МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Київський національний університет будівництва і архітектури

# ШТУЧНИЙ ІНТЕЛЕКТ ТА НЕЙРОННІ МЕРЕЖІ

Методичні вказівки
до виконання лабораторних робіт
для здобувачів другого (магістерського) рівня вищої освіти
спеціальності 126 «Інформаційні системи та технології»,
освітньо-професійних програм «Штучний інтелект. Когнітивні технології»,
«Інформаційні системи та технології»
всіх форм навчання

*У двох частинах*
***Частина перша***

Київ 2025

УДК 004.8
Ш94

Укладачі: О. О. Ільїн, д-р техн. наук, професор;
С. Д. Бушуєв, д-р техн. наук, професор

Рецензент О. А. Поплавський, канд. техн. наук, доцент

Відповідальний за випуск: С. Д. Бушуєв, д-р техн. наук, професор

*Затверджено на засіданні кафедри управління проектами, протокол № 6 від 11 грудня 2024 року.*

В авторській редакції.

**Штучний** інтелект та нейронні мережі [Електронний ресурс] : Ш94 методичні вказівки до виконання лабораторних робіт: у 2 ч. – Ч. 1 /уклад. О. О. Ільїн, С. Д. Бушуєв. – Київ : КНУБА, 2025. – 76с.

Містять зміст, завдання, порядок оформлення і вказівки до виконання лабораторних робіт першого модуля курсу.

Призначено для здобувачів другого (магістерського) рівня вищої освіти за спеціальністю «126 Інформаційні системи та технології» освітньо-професійних програм «Штучний інтелект. Когнітивні технології», «Інформаційні системи та технології».

**ЗМІСТ**

# ЗАГАЛЬНІ ПОЛОЖЕННЯ

Методичні вказівки до виконання лабораторних робіт з дисципліни «Штучний інтелект та нейронні мережі» містять описи лабораторних робіт згідно першого змістовного модуля дисципліни: «Модуль 1. Реалізація нейронних мереж засобами мови python». Вони призначені для надання здобувачам другого (магістерського) рівня необхідних теоретичних знань та практичних навичок з реалізації основних елементів штучних нейронних мереж (ШНМ) на мові python. Основна увага у цьому модулі приділяється моделюванню окремих складових нейронних мереж штатними засобами python.

Методичні вказівки по кожній окремій лабораторній роботі мають типову структуру та містять: мету роботи, необхідні теоретичні відомості, фрагменти кодів програм на мові python, інструкції з розробки повноцінної програми та особливості її запуску та дослідження результатів її роботи, завдання для самостійного виконання, вимоги до оформлення звіту про виконану роботу.

Під час виконання чергової лабораторної роботи, здобувачі мають ознайомитись з метою, теоретичними відомостями та дотримуючись інструкцій, виконати всі завдання першої частини роботи на мові python. У другій частині роботи слід проаналізувати умови задач та виконати завдання самостійної роботи. Хід виконання другої частини роботи (включаючи умови завдань, коди програм, результати їх роботи та висновки з аналізу отриманих результатів) має бути відображений у звіті. Звіти мають бути оформлені з дотриманням вимог до академічного письма.

Для виконання лабораторних робіт рекомендується використовувати середовище розробника Jupyter Notebook та оформлювати хід виконання роботи у вигляді «зошитів» (файли з розширенням ".ipynb").

# ЛАБОРАТОРНА РОБОТА №1

**Тема:** ***Основи програмування на мові Python. Бібліотеки NumPy, MatPlot, SciPy***

**Мета:** ознайомитись із командами та бібліотеками мови python, необхідними для моделювання нейронних мереж.

## *Частина 1.* Основні теоретичні відомості

Python – це високорівнева динамічно типізована багатопарадигмальна мова програмування [1]. Часто кажуть, що код Python майже схожий на псевдокод, оскільки він дозволяє висловлювати дуже потужні ідеї в невеликій кількості рядків коду, при цьому він дуже читабельний.

З 1 січня 2020 року Python офіційно припинив підтримку python2. Для виконання лабораторних робот по даній дисципліні будемо використовувати версію Python 3.7 або вище. Переконайтеся, що ви правильно встановили віртуальне середовище python3, перш ніж продовжити виконання лабораторної роботи. Ви можете ще раз перевірити свою версію Python у командному рядку після активації середовища, запустивши з командного рядка *python --version* або іншим чином (в залежності від середовища, в якому ви працюєте).

Використовуючи наведені нижче коди з підказками у вигляді коментарів, познайомтесь із основними конструкціями мови python, які стануть в нагоді під час вивчення дисципліни. Для цього намагайтесь набирати коди самостійно в власному середовищі IDE, запускайте та звіряйтесь із результатами, наведеними в коментарях. Після цього виконайте завдання та оформіть звіт.

Зверніть увагу на застосування таких бібліотек як Numpy, SciPy, Matplot. Для коректної роботи кодів із застосуванням функцій з цих бібліотек, ці бібліотеки треба встановити у ваш python. Механізм встановлення залежить від різновидів та версій IDE.

### *Основні типи даних*

Як і більшість мов, Python має кілька основних типів, включаючи цілі числа, числа з плаваючою точкою, логічні значення та рядки. Використання цих типів даних здійснюється в способи, аналогічними з інших мов

програмування. Відмітимо одразу, що типи даних під час декларування змінних, вказувати не треба.

*Числа (numbers):* цілі числа та числа з плаваючою точкою працюють так, як ви очікуєте від інших мов:

```
 1. x = 3
 2. print(type(x)) # Вивід "<class 'int'>"
 3. print(x)       # Вивід "3"
 4. print(x + 1)   # додавання; виводить "4"
 5. print(x - 1)   # віднімання; виводить "2"
 6. print(x * 2)   # множення; виводить "6"
 7. print(x ** 2)  # піднесення до ступеня; виводить "9"
 8. x += 1
 9. print(x)  # виводить "4"
10. x *= 2
11. print(x)  # виводить "8"
12. y = 2.5
13. print(type(y)) # виводить "<class 'float'>"
14. print(y, y + 1, y * 2, y ** 2) # виводить "2.5 3.5 5.0 6.25"
```

Зверніть увагу, що на відміну від багатьох мов, Python не має унарних операторів збільшення (x++) або зменшення (x--).

*Логічні значення (booleans):* Python реалізує всі звичайні оператори для логічної логіки, але використовує англійські слова, а не символи (&&, || тощо):

```
1. t = True
2. f = False
3. print(type(t)) # вивід "<class 'bool'>"
4. print(t and f) # логічне AND; вивід "False"
5. print(t or f)  # логічне OR; вивід "True"
6. print(not t)   # логічне NOT; вивід "False"
7. print(t != f)  # логічне XOR; вивід "True"
```

*Текстові рядки (Strings):* Python має дуже гарну підтримку роботи з рядками тексту:

```
1. hello = 'hello'    # Рядкові літерали можуть використовувати одинарні лапки
2. world = "world"    # або подвійні; не має значення
3. print(hello)       # вивід "hello"
4. print(len(hello))  # вивід "5"
```

```
5. hw = hello + ' ' + world  # Конкатенація рядків
6. print(hw)              # вивід "hello world"
7. hw12 = '%s %s %d' % (hello, world, 12)  # стиль форматування рядку sprintf
8. print(hw12)  # вивід "hello world 12"
```

Python є об'єктно-орієнтованою мовою програмування. Тому *рядкові об'єкти* мають купу корисних методів; наприклад:

```
1. s = "hello"
2. print(s.capitalize())  # вивід рядка з великої літери; вивід "Hello"
3. print(s.upper())       # Перетворює рядок у верхній регістр; вивід "HELLO"
4. print(s.rjust(7))      # Вирівнювання рядка по правому краю, доповнення
пробілами; вивід "  hello"
5. print(s.center(7))     # Центрує рядок, заповнюючи пробілами; вивід " hello "
6. print(s.replace('l', '(ell)'))  # Замінити всі екземпляри одного підрядка
іншим; вивід "he(ell)(ell)o"
8. print('  world '.strip())  # Видаляє пробіли на початку та в кінці; вивід
"world"
```

*Контейнери та ітеративна обробка даних*

Python містить кілька вбудованих типів контейнерів (структур даних): списки, словники, набори та кортежі.

*Списки (Lists).* Список в Python є еквівалентом масиву, але його розмір можна змінювати і він може містити елементи різних типів:

```
 1. xs = [3, 1, 2]    # створює список
 2. print(xs, xs[2])  # вивід "[3, 1, 2] 2"
 3. print(xs[-1])     # від'ємні індекси відраховуються з кінця списку; вивід "2"
 4. xs[2] = 'foo'     # списки можуть містити елементи різних типів
 5. print(xs)         # вивід "[3, 1, 'foo']"
 6. xs.append('bar')  # додає новий елемент у кінець списку
 7. print(xs)         # вивід "[3, 1, 'foo', 'bar']"
 8. x = xs.pop()      # видаляє та повертає останній елемент списку
 9. print(x, xs)      # вивід "bar [3, 1, 'foo']"
```

*Нарізка (slicing):* крім доступу до елементів списку по одному, Python забезпечує стислий синтаксис для доступу до *зрізів*; це називається «нарізкою»:

```
 1. nums = list(range(5))     # range - діапазон – це вбудована функція, яка
створює список цілих чисел
 2. print(nums)               # вивід "[0, 1, 2, 3, 4]"
 3. print(nums[2:4])          # повертає зріз з індексу 2 до 4 (не включно); вивід
"[2, 3]"
```

```
 4. print(nums[2:])          # повертає зріз від індексу 2 до кінця; вивід "[2,
3, 4]"
 5. print(nums[:2])          # повертає  зріз  початку до індексу 2 (не включно);
вивід "[0, 1]"
 6. print(nums[:])           # повертає зріз всього списку; вивід"[0, 1, 2, 3,
4]"
 7. print(nums[:-1])         # індекси для зрізу можуть бути від'ємними; вивід
"[0, 1, 2, 3]"
 8. nums[2:4] = [8, 9]       # призначити новий підсписок зрізу
 9. print(nums)              # вивід "[0, 1, 8, 9, 4]"
```

Підсписки часто застосовуються під час роботи з масивами з бібліотеки NumPy.

*Цикли (Loops):* Елементи списку можна перебирати за допомогою циклу таким чином:

```
1. animals = ['cat', 'dog', 'monkey']
2. for animal in animals:
3.     print(animal)
4. # вивід "cat", "dog", "monkey", кожний на своєму рядку.
```

Якщо вам потрібен доступ до *індексу* кожного елемента в тілі циклу, скористайтеся вбудованою функцією enumerate:

```
1. animals = ['cat', 'dog', 'monkey']
2. for idx, animal in enumerate(animals):
3.     print('#%d: %s' % (idx + 1, animal))
4. # вивід "#1: cat", "#2: dog", "#3: monkey", кожний на своєму рядку
```

*Перетворення списків (list comprehensions):* під час програмування ми часто хочемо перетворити один тип даних в інший. Як простий приклад розглянемо наступний код, який обчислює квадратні числа:

```
1. nums = [0, 1, 2, 3, 4]
2. squares = []
3. for x in nums:
4.     squares.append(x ** 2)
5. print(squares)   # вивід [0, 1, 4, 9, 16]
```

Цей код можна записати простіше застосувавши *перетворювач списків*

```
1. nums = [0, 1, 2, 3, 4]
```

```
2. squares = [x ** 2 for x in nums]
3. print(squares)   # вивід [0, 1, 4, 9, 16]
```

Перетворювачі списків також можуть включати умови:

```
1. nums = [0, 1, 2, 3, 4]
2. even_squares = [x ** 2 for x in nums if x % 2 == 0]
3. print(even_squares)  # вивід "[0, 4, 16]"
```

*Словник (dictionarie)* зберігає пару значень (key, value). Ви можете застосовувати його у такий спосіб:

```
 1. d = {'cat': 'cute', 'dog': 'furry'}  # Створіть новий словник з даними
 2. print(d['cat'])       # Отримати запис зі словника; вивід "cute"
 3. print('cat' in d)     # Перевірте, чи словник має даний ключ; вивід "True"
 4. d['fish'] = 'wet'     # Встановити запис у словнику
 5. print(d['fish'])      # Вивід "wet"
 6. # print(d['monkey'])  # KeyError: 'monkey' не є ключем у d
 7. print(d.get('monkey', 'N/A'))  # Отримати елемент за замовчуванням; вивід
"N/A"
 8. print(d.get('fish', 'N/A'))    # Отримати елемент за замовчуванням; вивід
"wet"
 9. del d['fish']         # Видалити елемент зі словника
10. print(d.get('fish', 'N/A')) # "fish" більше не є ключем; вивід "N/A"
```

*Циклічна обробка даних в словнику:* можна пройтись по усім ключам словника:

```
1. d = {'person': 2, 'cat': 4, 'spider': 8}
2. for animal in d:
3.     legs = d[animal]
4.     print('A %s has %d legs' % (animal, legs))
5. # вивід "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

А для того щоб отримати пару «ключ: значення», застосуйте метод *items*:

```
1. d = {'person': 2, 'cat': 4, 'spider': 8}
2. for animal, legs in d.items():
3.     print('A %s has %d legs' % (animal, legs))
4. # вивід "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

*Перетворювачі словників:* дають змогу спрощувати роботу з словниками. Наприклад:

```
1. nums = [0, 1, 2, 3, 4]
2. even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
3. print(even_num_to_square)  # вивід "{0: 0, 2: 4, 4: 16}"
```

*Набір (set)* – це невпорядкована сукупність різних елементів. Як простий приклад розглянемо наступне:

```
 1. animals = {'cat', 'dog'}
 2. print('cat' in animals)   # Перевірте, чи є елемент у наборі; вивід "True"
 3. print('fish' in animals)  # вивід "False"
 4. animals.add('fish')       # Додайте елемент до набору
 5. print('fish' in animals)  # вивід "True"
 6. print(len(animals))       # кількість елементів у наборі; вивід "3"
 7. animals.add('cat')        # Додавання елемента, який уже є в наборі, нічого не
робить
 8. print(len(animals))       # вивід "3"
 9. animals.remove('cat')     # Видалити елемент із набору
10. print(len(animals))       # вивід "2"
```

*Застосування циклів:* ітерація по набору має той самий синтаксис, що і ітерація по списку; але оскільки набори є невпорядкованими, ви не можете зробити припущення щодо порядку, в якому ви відвідуєте елементи набору:

```
1. animals = {'cat', 'dog', 'fish'}
2. for idx, animal in enumerate(animals):
3.     print('#%d: %s' % (idx + 1, animal))
4. # вивід "#1: fish", "#2: dog", "#3: cat"
```

*Перетворювачі наборів:* за аналогією до списків та словників, перетворювачі наборів спрощують роботу:

```
1. from math import sqrt
2. nums = {int(sqrt(x)) for x in range(30)}
3. print(nums)  # вивід "{0, 1, 2, 3, 4, 5}"
```

*Кортеж (Tuple)* – це (незмінний) упорядкований список значень. Кортеж багато в чому схожий на список; одна з найважливіших відмінностей полягає в тому, що кортежі можна використовувати як ключі в словниках і як елементи наборів, тоді як списки ні. Ось тривіальний приклад:

```
1. d = {(x, x + 1): x for x in range(10)}  # Створіть словник із ключами кортежу
2. t = (5, 6)        # Створіть кортеж
3. print(type(t))    # вивід "<class 'tuple'>"
```

```
4. print(d[t])       # вивід "5"
5. print(d[(1, 2)])  # вивід "1"
```

*Функції*

*Функції (functions)* у Python створюють за допомогою ключового слова *def:*

```
 1. def sign(x):
 2.     if x > 0:
 3.         return 'positive'
 4.     elif x < 0:
 5.         return 'negative'
 6.     else:
 7.         return 'zero'
 8.
 9. for x in [-1, 0, 1]:
10.     print(sign(x))
11. # вивід "negative", "zero", "positive"
```

Функції часто визначають з значеннями за замовчуванням для деяких вхідних аргументів наприклад:

```
1. def hello(name, loud=False):
2.     if loud:
3.         print('HELLO, %s!' % name.upper())
4.     else:
5.         print('Hello, %s' % name)
6.
7. hello('Bob') # вивід "Hello, Bob"
8. hello('Fred', loud=True)  # вивід "HELLO, FRED!"
```

*Бібліотека Numpy*

Numpy –це основна бібліотека для наукових обчислень на Python. Вона спрощує користування масивами, представляючи їх як об'єкти. Це значено спрощує математичні перетворення з матрицями, які також є масивами.

*Масиви.* Масив numpy – це таблиця (матриця) із даними, усі одного типу, індексована кортежем невід'ємних цілих чисел. Кількість вимірів – це *ранг масиву*; *форма масиву* - це кортеж цілих чисел, що задає розмір масиву вздовж кожного виміру.

Ми можемо ініціалізувати масиви numpy із вкладених списків Python і отримувати доступ до елементів масиву за допомогою квадратних дужок:

```
import numpy as np

a = np.array([1, 2, 3])   # Створіть масив рангу 1
print(type(a))            # вивід "<class 'numpy.ndarray'>"
print(a.shape)            # вивід "(3,)"
print(a[0], a[1], a[2])   # вивід "1 2 3"
a[0] = 5                  # Змінити елемент масиву
print(a)                  # вивід "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Створіть масив рангу 2
print(b.shape)                     # вивід "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # вивід "1 2 4"
```

Numpy також містить багато функцій для створення масивів:

```
import numpy as np

a = np.zeros((2,2))   # Створіть масив що містить нулі
print(a)              # вивід "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Створіть масив що містить одиниці
print(b)              # вивід "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Створіть масив що містить 7
print(c)               # вивід "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Створіть матрицю тотожності 2x2
print(d)              # вивід "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Створіть масив, заповнений випадковими
значеннями
print(e)                     # Має вивести "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

*Індексація в масивах*

В numpy існує кілька способів роботи з індексами [5].

*Нарізка (slicing):* Подібно до списків Python, масиви numpy можна «нарізати». Оскільки масиви можуть бути багатовимірними, ви повинні вказати зріз для кожного виміру масиву:

```
import numpy as np

# Створіть наступний масив рангу 2 із формою (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Використовуйте «нарізку», щоб витягнути підмасив, що складається з перших
# 2 рядків і стовпців 1 і 2; b є наступним масивом з формою  (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# Зріз масиву є представленням в одні і ті ж самі данні; тому їх зміна
# призведе до зміни оригінального масиву
print(a[0, 1])   # вивід "2"
b[0, 0] = 77     # b[0, 0] є тим самим фрагментом даних що і a[0, 1]
print(a[0, 1])   # вивід "77"
```

Ви також можете поєднувати цілочисельне індексування з індексуванням фрагментів. Однак це призведе до масиву нижчого рангу, ніж вихідний масив.

```
import numpy as np

# Створіть наступний масив рангу 2 із формою (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Два способи доступу даних в середньому рядку масива.
# Поєднання цілих індексів із зрізами призведе до масиву нижчого рангу,
# тоді коли використання зрізів призведе до масиву того ж рангу,
# що і оригінальний масив:
row_r1 = a[1, :]    # представлення рангу 1 другого рядка a
row_r2 = a[1:2, :]  # представлення рангу 2 другого рядка a
print(row_r1, row_r1.shape)  # вивід "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # вивід "[[5 6 7 8]] (1, 4)"

# Ми можемо зробити те саме розрізнення під час доступу до стовпців масиву:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # вивід "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # вивід "[[ 2]
                             #         [ 6]
```

```
24.                             #          [10]] (3, 1)"
```

*Індексування цілочисельного масиву:* коли ви індексуєте масиви numpy за допомогою нарізки, отриманий масив завжди буде підмасивом вихідного масиву. Навпаки, індексація цілочисельного масиву дозволяє створювати довільні масиви, використовуючи дані з іншого масиву. Розглянемо приклад:

```
 1. import numpy as np
 2.
 3. a = np.array([[1,2], [3, 4], [5, 6]])
 4.
 5. # Приклад індексації цілочисельного масиву.
 6. # Повернений масив матиме форму (3,) та
 7. print(a[[0, 1, 2], [0, 1, 0]])  # виведе "[1 4 5]"
 8.
 9. # Наведений вище приклад індексації цілочисельного масиву еквівалентний цьому:
10. print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # вивід "[1 4 5]"
11.
12. # Використовуючи індексацію цілочисельного масиву, ви можете використовувати
13. # той самий елемент вихідного масиву:
14. print(a[[0, 0], [1, 1]])  # вивід "[2 2]"
15.
16. # Еквівалент попереднього прикладу індексування масиву цілих чисел
17. print(np.array([a[0, 1], a[0, 1]]))  # вивід "[2 2]"
```

Один корисний трюк з індексуванням цілочисельного масиву полягає у виборі або зміні одного елемента з кожного рядка матриці:

```
 1. import numpy as np
 2.
 3. # Створіть новий масив, з якого будемо вибирати елементи
 4. a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
 5.
 6. print(a)  # вивід "array([[ 1,  2,  3],
 7.           #                [ 4,  5,  6],
 8.           #                [ 7,  8,  9],
 9.           #                [10, 11, 12]])"
10.
11. # Створіть масив індексів
12. b = np.array([0, 2, 0, 1])
13.
14. # Виберіть по одному елементу з кожного рядка a, використовуючи індекси в b
15. print(a[np.arange(4), b])  # вивід "[ 1  6  7 11]"
16.
17. # Змініть один елемент із кожного рядка a, використовуючи індекси в b
18. a[np.arange(4), b] += 10
19.
```

```
print(a)  # вивід "array([[11,  2,  3],
          #               [ 4,  5, 16],
          #               [17,  8,  9],
          #               [10, 21, 12]])
```

*Логічне індексування масиву:* логічне індексування масиву дозволяє вибирати довільні елементи масиву. Часто цей тип індексування використовується для вибору елементів масиву, які задовольняють певну умову. Ось приклад:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # знаходження елементів, які більше 2;
                     # операція повертає масив numpy типу Booleans тієї ж
                     # форми як і a, де кожен слот bool_idx містить
                     # відповідь, чи більше елемент з a за 2.

print(bool_idx)      # вивід "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# Можна використати логічну індексацію масиву для побудови масиву рангу 1
# який міститиме елементи, що відповідатимуть значенню True
# з bool_idx
print(a[bool_idx])  # вивід "[3 4 5 6]"

# все це можна зробити коротшим записом:
print(a[a > 2])     # вивід "[3 4 5 6]"
```

*Типи даних в Numpy*

Кожен масив numpy є послідовністю елементів одного типу. Numpy пропонує великий набір числових типів даних, які можна використовувати для створення масивів. Numpy намагається вгадати тип даних, коли ви створюєте масив, але функції, які створюють масиви, зазвичай також включають необов'язковий аргумент для явного визначення типу даних. Ось приклад:

```
import numpy as np

x = np.array([1, 2])   # дозволяє numpy обрати тип даних самому
print(x.dtype)         # вивід "int64"

```

```
x = np.array([1.0, 2.0])   # дозволяє numpy обрати тип даних самому
print(x.dtype)             # вивід "float64"

x = np.array([1, 2], dtype=np.int64)   # примушує встановити заданий тип
print(x.dtype)                         # вивід "int64"
```

*Математика з масивами*

Основні математичні функції працюють по-елементно з масивами та доступні як перевантаження операторів, так і функції в модулі numpy:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# поелементне сумування; обидва створюють масив
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Поелементна різниця; обидва створюють масив
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Поелементний множення; обидва створюють масив
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Поелементний ділення; обидва створюють масив
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Поелементний квадратний корінь; створює масив
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

Зауважте, що оператор * працює як поелементне множення, не матричне множення. Замість нього ми застосовуємо функцію ***dot*** для

обчислення скалярного добутку векторів, для множення вектору на матрицю та для множення матриць. dot доступна як функція в модулі numpy так і як метод екземпляра для об'єктів масивів:

```
 1. import numpy as np
 2.
 3. x = np.array([[1,2],[3,4]]) # matrix x
 4. y = np.array([[5,6],[7,8]]) # matrix y
 5.
 6. v = np.array([9,10]) # vector v
 7. w = np.array([11, 12]) # vector w
 8.
 9. # Внутрішній добуток векторів; вивід 219
10. print(v.dot(w))
11. print(np.dot(v, w))
12.
13. # Матричний / векторний добуток; обидва створюють масив рангу 1 [29 67]
14. print(x.dot(v))
15. print(np.dot(x, v))
16.
17. # Матриця / матричний добуток; обидва створюють масив рангу 2
18. # [[19 22]
19. #  [43 50]]
20. print(x.dot(y))
21. print(np.dot(x, y))
```

Наряду з іншими цікавими функціями по роботі з масивами, є функція sum:

```
1. import numpy as np
2.
3. x = np.array([[1,2],[3,4]])
4.
5. print(np.sum(x))  # Обчисліть суму всіх елементів; вивід "10"
6. print(np.sum(x, axis=0))  # Обчисліть суму кожного стовпця; вивід "[4 6]"
7. print(np.sum(x, axis=1))  # Обчисліть суму кожного рядка; вивід "[3 7]"
```

Інші математичні функції наведені в документації [6].

Окрім обчислення математичних функцій за допомогою масивів, нам часто потрібно змінювати форму або іншим чином маніпулювати даними в масивах. Найпростішим прикладом цього типу операції є транспонування матриці; щоб транспонувати матрицю, просто використовуйте атрибут T об'єкта масиву:

```
 1. import numpy as np
 2.
 3. x = np.array([[1,2], [3,4]])
 4. print(x)    # вивід "[[1 2]
 5.             #          [3 4]]"
 6. print(x.T)  # вивід "[[1 3]
 7.             #          [2 4]]"
 8.
 9. # Зауважте, що транспонування масиву рангу 1 нічого не робить:
10. v = np.array([1,2,3])
11. print(v)    # вивід "[1 2 3]"
12. print(v.T)  # вивід "[1 2 3]"
```

Інші функції по маніпулюванню масивами наведені в [7].

*Бібліотека Matplotlib*

Matplotlib є бібліотекою для зображення графіків. Ми розглянемо модуль matplotlib.pyplot який пропонує систему графічного виводу аналогічної до MATLAB.

Побудова графіків. Набільш важливою функцією в matplotlib є plot. Вона дозволяє будувати графіки 2D. Ось простий приклад:

```
 1. import numpy as np
 2. import matplotlib.pyplot as plt
 3.
 4. # Обчисліть координати x і y для точок на синусоїді
 5. x = np.arange(0, 3 * np.pi, 0.1)
 6. y = np.sin(x)
 7.
 8. # Побудуйте точки за допомогою matplotlib
 9. plt.plot(x, y)
10. plt.show()  # ви маєте викликати plt.show() для появи графіку.
```

Результатом виконання коду буде створення нового вікна із графіком функції (див рис.1.1).

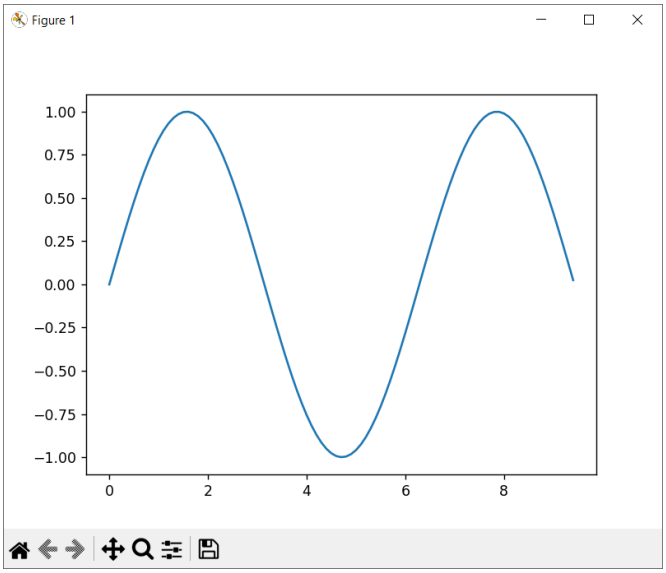


Рис.1.1. Графік функції, побудований за допомогою бібліотеки matplotlib.pyplot

Додавши трохи додаткового коду, ми можемо легко накреслити кілька ліній різних функцій на одному графіку одночасно, додати їм назви, підписи та мітки осей:

```
 1. import numpy as np
 2. import matplotlib.pyplot as plt
 3.
 4. # Обчисліть координати x і y для точок на синусі та косинусі
 5. x = np.arange(0, 3 * np.pi, 0.1)
 6. y_sin = np.sin(x)
 7. y_cos = np.cos(x)
 8.
 9. # Побудуйте точки за допомогою matplotlib
10. plt.plot(x, y_sin)
11. plt.plot(x, y_cos)
12. plt.xlabel('x axis label')
13. plt.ylabel('y axis label')
14. plt.title('Sine and Cosine')
15. plt.legend(['Sine', 'Cosine'])
16. plt.show()
```

Результат показаний на рис.1.2.

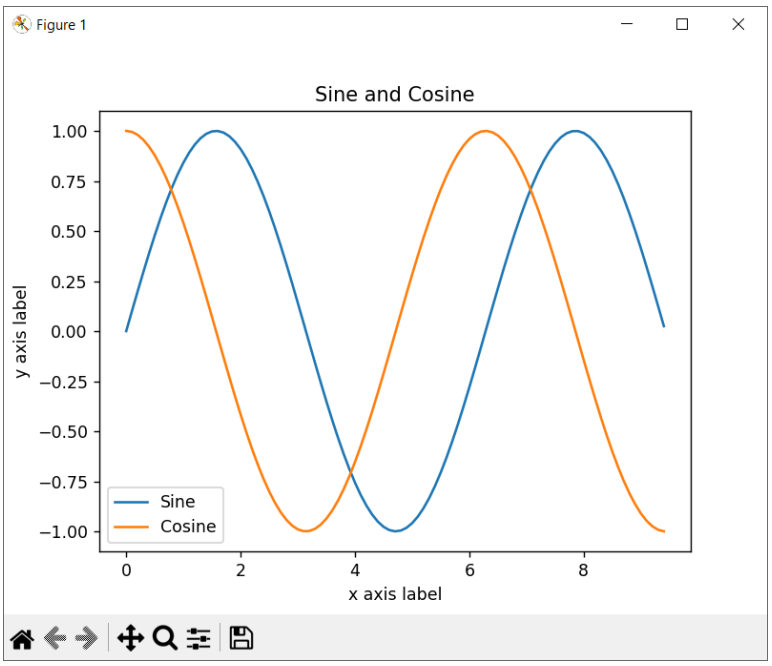


Рис.1.2. Вікно із графіками двох функцій, побудований
за допомогою бібліотеки matplotlib.pyplot

Більше інформації про застосування бібліотеки можете знайти в [9].

*Багато графіків (Subplots) на одному рисунку*

За допомогою функції subplot ви можете виводити кілька окремих графіків в межах одного малюнку (вікна графіку). Код нижче покаже як це робити:

```
 1. import numpy as np
 2. import matplotlib.pyplot as plt
 3.
 4. # Обчисліть координати x і y для точок на кривих синусів і косинусів
 5. x = np.arange(0, 3 * np.pi, 0.1)
 6. y_sin = np.sin(x)
 7. y_cos = np.cos(x)
 8.
 9. # Налаштуйте сітку фігури, яка має висоту 2 і ширину 1,
10. # і встановити перший такий subplot як активний.
11. plt.subplot(2, 1, 1)
12.
13. # зробіть перший графік
14. plt.plot(x, y_sin)
15. plt.title('Sine')
16.
```

```
17. # встановить другий subplot як активний та зробіть другий графік.
18. plt.subplot(2, 1, 2)
19. plt.plot(x, y_cos)
20. plt.title('Cosine')
21.
22. # показ всієї фігури
23. plt.show()
```

Результатом буде поява одного вікна, на якому показано два окремих графіки (рис.1.3)

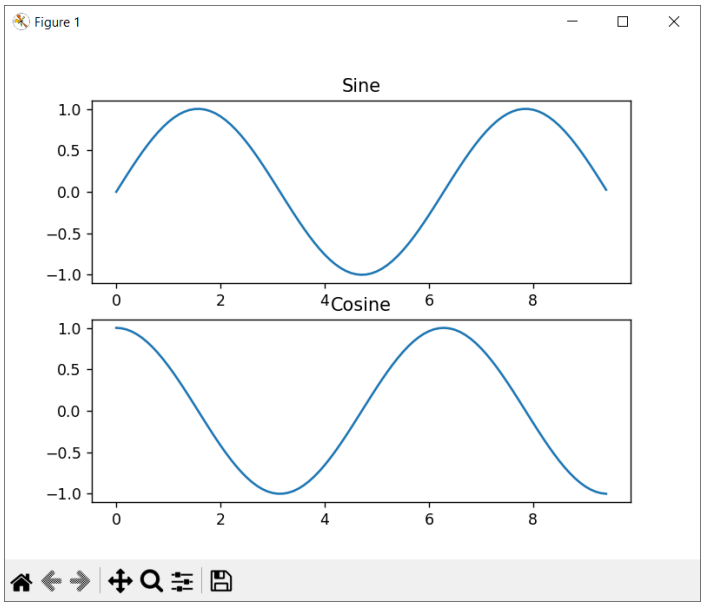


Рис.1.3. Вікно із двома окремими графіками двох функцій, побудований за допомогою бібліотеки matplotlib.pyplot

Ця бібліотека також дає змогу працювати з зображеннями. Для виводу зображення можна застосовувати функцію imshow. Розгляньте приклад:

```
 1. import numpy as np
 2. from matplotlib.pyplot import imread
 3. import matplotlib.pyplot as plt
 4.
 5. img = imread('assets/cat1.jpg')
 6. print(img.dtype, img.shape)  # Prints "uint8 (711, 469, 3)
 7.
14. img_tinted = img * [1, 0.95, 0.8]
15.
17. plt.subplot(1, 2, 1)
18. plt.imshow(img)
19.
```

```
21. plt.subplot(1, 2, 2)
22.
26. plt.imshow(np.uint8(img_tinted))
27. plt.show()
```

Результатом буде відображення одного вікна, в якому буде міститись два цифрових зображення рис.1.4).

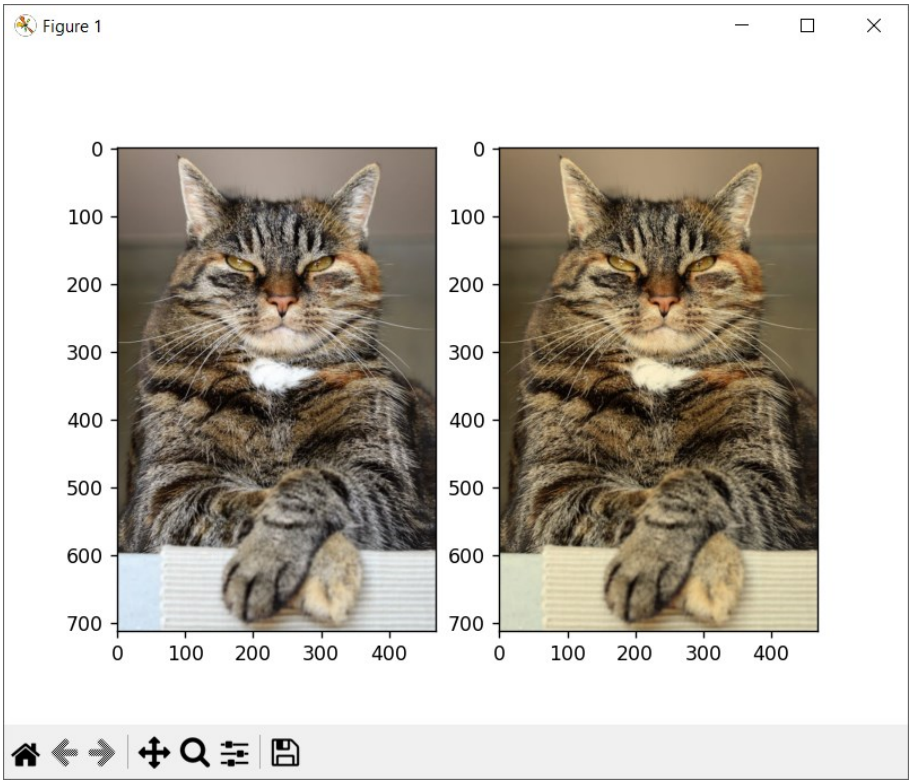


Рис.1.4. Вікно із двома зображеннями (ліворуч-оригінальне, праворуч – із тоновою корекцією)

### *Частина 2.* Завдання для самостійного виконання

1. Виконайте всі коди з теоретичної частини в Вашому середовищі розробнику Python; переконайтесь що вони працюють вірно згідно з коментарями в тексті коду. Для виводу зображень, оберіть з власних, але невеликого розміру.

2. Створіть програму, в якій визначить функцію з одним вхідним параметром x, яка буде здійснювати розрахунок за формулою:

$$y(x) = \frac{1}{1+\exp^{-x}} \qquad (1.1)$$

Розрахуйте значення y(x) для $x \in [-10; 10], \Delta x = \frac{1}{N}$, де N задається через консоль. Сформуйте вектор x та відповідний йому вектор y, виведіть їх у консоль.

3. Намалюйте графік функції (1.1) за допомогою бібліотеки matplot. Мінімальне та максимальне значення x а також параметр N задайте через консоль.

4. Створіть два вектори та матрицю за допомогою бібліотеки Numpy:

$$a = [0.8 \quad 1.2 \quad -2.1 \quad 0.45]$$
$$b = [0 \quad -0.45 \quad 0.21 \quad 1.3]$$

$$w_1 = \begin{bmatrix} 0.23 & 0.1 & -1.2 & 0.33 \\ 0.6 & 0.1 & 0.1 & 0.7 \\ 0.3 & -0.1 & -0.43 & 0.9 \\ 0.83 & -0.13 & 0.16 & 0.12 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} -0.1 & 0 & .2 & 0.7 \\ 0.2 & -0.61 & -0.21 & 0.17 \\ -0.3 & .1 & 0.3 & 0.19 \\ 0.8 & 0.3 & 0.6 & -0.2 \end{bmatrix}$$

Напишіть програму, яка здійсніть операції:

- додавання векторів a та b;
- скалярний добуток векторів a та b;
- множення вектору a на матрицю w1;
- множення матриці b на вектор w2;
- множення матриці w1 на матрицю w2;
- транспонування матриці w2;

Результати операцій збережіть як нові данні (числа, вектори чи матриці) та виведіть у консоль.

5. Оформити звіт, в якому відобразити:

- ✓ Постановку завдання (завдання 2-4).
- ✓ Короткі теоретичні відомості
- ✓ Код програми із коментарями.
- ✓ Результати роботи програми.
- ✓ Висновки.

*Запитання до лабораторної роботи*

1. Яким чином в мові python можна представити скаляр?

2. Яким чином в мові python можна представити множину даних? Вектор? Матрицю?

3. Для чого призначена бібліотека Numpy? В яких ситуаціях її використання має переваги?

4. Для чого призначена бібліотека matplot?

5. Який синтаксис визначення функції?

## ЛАБОРАТОРНА РОБОТА №2

**Тема:** ***Моделювання елементів нейронних мереж на мові Python***

**Мета:** Навчитися розробляти датасет та моделювати окремі елементи нейронної мережі на мові Python.

### *Частина 1.* Основні теоретичні відомості

Глибинне навчання – це різновид машинного навчання, в якому застосовуються нейронні мережі з прихованими шарами всередині. Штучна нейронна мережа або просто – нейронна мережа (НМ) - це набір математичних формул та алгоритмів, які можна запрограмувати. Відповідна програма під час її роботи буде приблизно моделювати когнітивні процеси всередині мозку живої істоти. Найменшим елементом, з якого складається нервова тканина мозку, є нейрон. Нейрони зв'язані між собою за допомогою синапсів. В процесі навчання штучної нейронної мережі, одні зв'язки між нейронами укріплюються, інші послабляються, для досягнення бажаного результату управління. Зв'язки між нейронами в моделі можна упорядкувати, розмістивши нейрони у шари. Схема трьохрівневої нейронної мережа представлена на рис.2.1. Кожний нейрон зображений у вигляді кружка сірого кольору. Лінії між нейронами моделюють синапси.

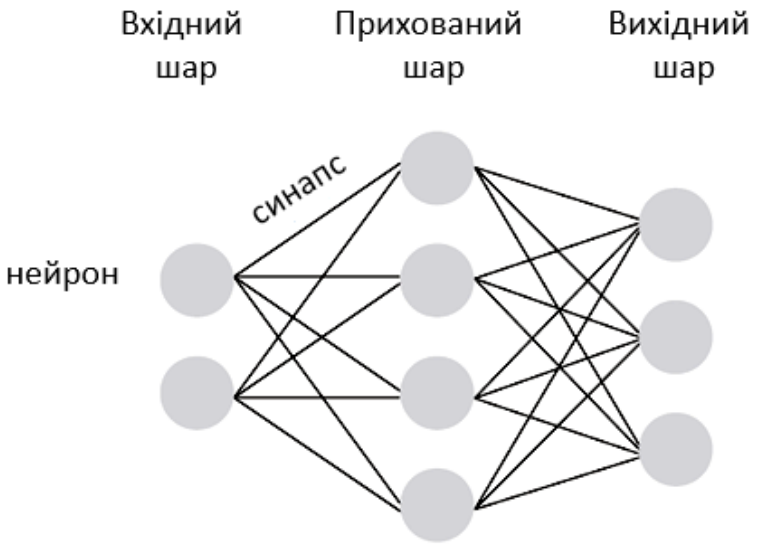


Рис. 2.1. Схема повнозв'язаної нейронної мережі з трьох шарів

На рис.2.1 наведена схема НМ, в якій нейрони згруповані по 3 шарах. Перший шар називається «вхідним», останній – «вихідним», а проміжні – «прихованими». Проміжних шарів може бути більше одного (тому навчання в таких мережах має назву «глибинне»). В даній мережі кожний нейрон одного шару має зв'язки з усіма нейронами наступного шару. Тому така мережа має назву повнозв'язаною.

Розглянемо структуру окремого нейрону, показану на рис.2.2.

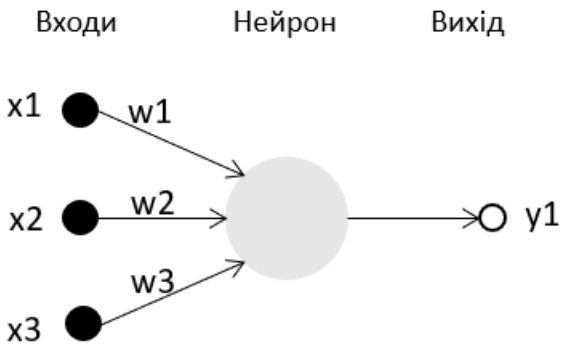


Рис. 2.2. Структура окремого нейрону

Нерон має входи (один чи більше), саме тіло нейрону та вихід нейрону (один чи більше). На рис. 2.2 показаний нейрон з 3 входами та одним виходом (вихід нейрону є синапсом). Можливі конфігурації нейрону та найпростіших мереж на його основі, можуть бути такими:

- нейрон з одним входом та одним виходом;
- нейрон з кількома входами та одним виходом;
- одношарова мережа з одним входом та кількома виходами;
- одношарова мережа з кількох нейронів, що мають кілька входів та виходів;
- мережа з одним прихованим шаром, кількома входами та виходами.

В цій лабораторній роботі розглянемо основні аспекти моделювання всіх зазначених елементів НМ на мові Python.

*Модель нейрона «один вхід-один вихід»*

Розглянемо структуру найпростішого нейрону, який складається з входу, тіла та виходу (рис. 2.3)

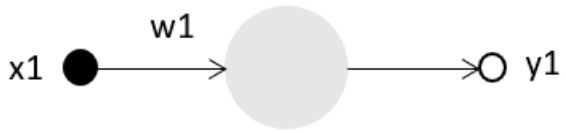


Рис.2. 3. Структура нейрону з одним входом та одним виходом

На схемі нейрону стрілками показано напрямок передачі електричного сигналу: від входу через тіло до виходу. Розповсюдження сигналу в такому напрямку має назву *пряме розповсюдження* сигналу. І нейронні мережі, які складаються з нейронів із таким напрямком розповсюдження вхідного сигналу, мають назву мережі з прямими розповсюдженням.

Робота штучного нейрону взагалі полягає в тому, щоб трансформувати вхідний сигнал $x_1$ у вихідний $y_1$. Він це здійснює за допомогою особливостей роботи тіла нейрону, яку можна також описати математичною залежністю $y = f(x, w)$. В найпростішому варіанті це може бути множення величини вхідного значення $x_1$ на параметр нейрону $w_1$. Параметр $w_1$ має назву *ваговий коефіцієнт*. Він характеризує властивості перетворення сигналу на вході нейрона у сигнал на виході. У нейрона кожний вхід, скільки б їх не було, має свій власний ваговий коефіцієнт.

Під час навчання нейронної мережі (а це є основний етап створення моделі нейронної мережі) основною задачею є встановлення значень всіх вагових коефіцієнтів нейронів мережі у конкретні потрібні значення, які дозволяють моделі НМ в цілому вирішити поставлену перед нею задачу.

Для нейрону з рис.2.3 принцип такого перетворення сигналу можна описати формулою (2.1):

$$y_1 = x_1 w_1. \tag{2.1}$$

Нейронні мережі створюють для вирішення різних задач, де потрібне моделювання певних когнітивних функцій (людини). Але на рівні математики все зводиться до вирішення задач прогнозування та класифікації (якщо ми говоримо про нейронні мережі, які навчаються з вчителем). Тому інтерпретувати формулу (2.1) можна як отримання прогнозного значення $y_1$ відповідно до вхідного $x_1$. Вірне прогнозування можливе тільки тоді, коли мережа під час навчання обрала вірний коефіцієнт $w_1$. Для мережі з рис.2.3 важко знайти практичне застосування, бо прогнозування на базі одного нейрону практично не має доцільності. Реальні мережі мають мільйони

нейронів та сотні шарів. Але ми наразі розглядаємо базові конфігурації, з яких такі мережі складаються та відповідні кодові конструкції на мові Python. Тому для наближення до реальності і для такого простого випадку, застосуємо навчальне приклад: нехай мережі на рис. 2.3 прогнозує імовірність виграшу у спортивному змаганні.

Мережа може вірно прогнозувати тільки тоді, коли вона *навчена*. Процес навчання здійснюється на основі сукупності даних для навчання, які утворюють навчальні набори даних, так звані *датасети* (datasets). Навчальний датасет містить значення як для входів, так і для виходів мережі. Тому він і названий – для навчання. Коли ми навчили мережу на цьому наборі даних, її можна використовувати для прогнозування і на основі інших даних, які не входили у цей набір. Але під час навчання обов'язково мають бути дані як для входу мережі, так і ті, що мали б бути на виході.

Отже, якщо говоримо про отримання імовірності виграшу у спортивному змаганні, маємо такий датасет, що характеризує історію програшів та виграшів спортивної команди у минулому [10] (табл.2.1).

*Таблиця 2.1*

**Датасет для навчання**

| | #1 | #2 | #3 | #4 |
|---|---|---|---|---|
| | toes | wlrec | nfans | win_or_loose_binary |
| № | **number of toes** | **win/loose** | **number of fans** | **probability of win (fact!)** |
| 1 | 8.5 | 0.65 | 1.2 | 1 |
| 2 | 9.5 | 0.8 | 1.3 | 1 |
| 3 | 9.9 | 0.8 | 0.5 | 0 |
| 4 | 9 | 0.9 | 1 | 1 |
| | | | | |
| | input | | | output |

Цей датасет є таблицею, яка містить колонки – *ознаки* (features) розглядаємого явища, та рядки – значення цих ознак для одного періоду часу. В цьому випадку ознаками є:

- number_of_toes: середня кількість ігор, зіграних спортивною командою перед поточною грою сезону;
- win/lose – відношення виграшів/програшів ігор перед поточною грою;
- number of fans – кількість болільників на поточній грі;
- probability of win (fact!) – чи була виграна дана ігра (1) чи програна (0).

Як ви бачите, дана таблиця містить історичні данні, які вже відомі: результати чотирьох ігор (4 рядки таблиці), які відбувались на протязі року,

в кожному його сезоні – зима, весна, літо, осінь. Але під час навчання даної мережі ми будемо вважати, що відомими є тільки одна ознака – перша колонка (середня кількість зіграних ігор перед початком поточної гри - number_of_toes). І мережа повинна навчитись *прогнозувати*, тобто отримувати значення останньої ознаки – імовірність виграшу поточної ігри.

Для мережі з рис. 2.3, в якості входу, оберемо ознаку number_of_toes. Прогнозувати будемо імовірність виграшу ігри - ознаку probability of win (fact!). Зрозуміло, що вхідних даних замало для реального практичного виконання такої задачі. Але ми акцентуємо увагу на розробці коду моделі на мові python.

Модель цієї мережі представлена у лістингу 2.1.

Лістинг 2.1. Модель мережі нейрону «один вхід-один вихід» [10]

```
1. def neural_network(input, weight):
2.     prediction = input * weight
3.     return prediction
4.
5. number_of_toes = [8.5, 9.5, 10, 9]
6.
7. weight = 0.1
8.
9. input = number_of_toes[0]
10. pred = neural_network(input,weight)
11.
12. print(pred)
```

*Коментарі по рядках коду:*

1-3: задаємо функцію, яка повертає результат прогнозу; вхідними параметрами є значення входу та велична вагового коефіцієнту;

5: задаємо список number_of_toes, який містить значення відповідної ознаки (див. табли.2.1) за 4 сезони;

7: задаємо ваговий коефіцієнт (один);

9: обираємо значення для входу з можливих значень ознаки number_of_ toes; індекс 0 дозволяє обрати значення ознаки для першої гри сезону;

10: обчислюємо прогноз для обраного вхідного значення та вагового коефіцієнту мережі;

12: виводимо значення прогнозу в консоль.

Результат роботи даного коду, виконаний в середовищі Jupiter Notebook, показаний на скріншоті (рис.2.4)

```
0.8500000000000001
```

Рис. 2.4. Результат виконання коду з лістингу 2.1

*Аналіз роботи коду*

Якщо порівняти результат з значенням ознаки probability of win (fact!) з датасету, для першої гри сезону, то там ми маємо значення 1. Тобто, можна стверджувати, що нейронна мережа виконала прогнозування вірно. Якщо ж в якості входу обрати ознаки інших ігор сезону (індекси 1-3 в коді), то там вже результати можуть не співпадати. Це цілком природньо, зважуючи на простоту даної «архітектури» мережі. Для співпадіння з результатом для інших сезонів, ми мали б змінити ваговий коефіцієнт. Але внаслідок того, що коефіцієнт один, то для обраного сезону результат буде вірним, а для інших - вже ні. З цього «замкненого» кола можна вибратись тільки шляхом збільшення загальної кількості нейронів у шарі.

*Модель нейрона мережі «кілька входів - один вихід»*

Попередня нейронна мережа здійснювала прогноз одного значення спираючись на одне вхідне значення. Питання – чи достовірний такий прогноз – в даному випадку риторичне, але його можна поставити для нейронної мережі з будь-якою кількістю входів. Що треба, щоб підвищити достовірність прогнозу? Один із способів – надати мережі більше вхідної інформації, вхідних даних. Для цього треба збільшити кількість входів. Для цього розглянемо структуру найпростішого нейрону, який складається з кількох входів, тіла та виходу (рис.2.5.)

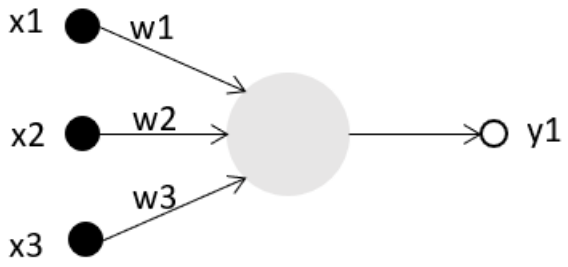


Рис. 2.5. Структура нейрону з кількома входами та одним виходом

Такий нейрон, теоретично, має робити більш точний прогноз. Для його отримання треба, щоб нейрон врахував вхідні значення з урахуванням вагових коефіцієнтів кожного вхідного зв'язку окремо. Це він робить за допомогою наступної формули (2.2):

$$y_1 = x_1 w_1 + x_2 w_2 + x_3 w_3 = \sum_{i=1}^{3} x_i w_i \qquad (2.2)$$

Формула (2.2) дає зважену суму вхідних значень.

Для мережі з рис. 2.4, в якості входів, оберемо ознаки:
- вхід x1: number_of_toes: середня кількість ігор, зіграних перед поточною грою;
- вхід x2: win/lose – відношення виграшів/програшів ігор перед поточною грою;
- вхід x3: number of fans – кількість болільників на поточній ігрі.

Прогнозувати будемо імовірність виграшу ігри- ознаку probability of win (fact!). Код моделі такої мережі на мові python міститься в лістингу 2.2.

Лістинг 2.2. Модель мережі нейрону «три входи-один вихід» [10]

```
def neural_network(input, weights):
    pred = w_sum(input,weights)
    return pred

def w_sum(a,b):
    output = 0
    for i in range(len(a)):
        output += (a[i] * b[i])
    return output

weights = [0.1, 0.2, 0]

toes =  [8.5, 9.5, 9.9, 9.0]
wlrec = [0.65, 0.8, 0.8, 0.9]
nfans = [1.2, 1.3, 0.5, 1.0]

input = [toes[0],wlrec[0],nfans[0]]

pred = neural_network(input,weights)
print(pred)
```

*Коментарі по ключових рядках коду:*

1-3: задаємо функцію, яка повертає результат прогнозу; вхідними параметрами є значення входів, заданих у вигляді списку input, та величини вагових коефіцієнтів weights (також список);
2: застосовуємо метод w_sum(input,weights), який відповідає за обчислення зваженої суми входів та вагових коефіцієнтів, які у вигляді списків подаємо вхідними параметрами;
5-9: реалізація функції обчислення зваженої суми:
11: задаємо вагові коефіцієнти у вигляді списку з трьох елементів;
13-15: задаємо значення ознак, які будемо подавати на входи, також у вигляді списків;
17: формуємо вектор вхідних даних input як список ознак за однаковим індексом (індекс 0 дозволяє обрати вхідних ознак для першої гри сезону);
19: обчислюємо прогноз для векторів входів та вагових коефіцієнтів мережі;
12: виводимо значення прогнозу в консоль.

Результат роботи даного коду, виконаний в середовищі Jupiter Notebook, показаний на скріншоті (рис. 2.5).

```
0.9800000000000001
```

Рис. 2.5 Результат виконання коду з лістингу 2.2

*Аналіз роботи коду*

Якщо порівняти результат з значенням ознаки probability of win (fact!) з датасету, для першої гри сезону, то там ми маємо значення 1. Отримане значення 0.98 на 2% відрізняється від істинного, тобто можна стверджувати, що нейронна мережа виконала прогнозування вірно. Якщо ж в якості входу обрати ознаки інших ігор сезону (індекси 1-3 в коді, спробуйте обрати самостійно та порівняти з істинними), то там вже результати можуть не співпадати. Для співпадіння прогнозних значень з істинними, мережу треба навчати (цей процес та відповідний код будемо розглядати в наступних роботах).

*Модель фрагменту нейронної мережі «один вхід – кілька виходів»*

Розглянемо ситуацію, коли в мережі має бути тільки один вхід та декілька виходів (рис.2.6).

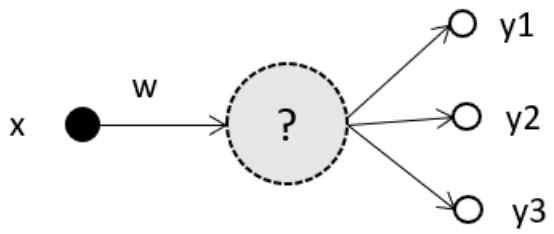


Рис. 2.6. Модель мережі, в якій один вхід та декілька виходів

З погляду класичного нейрону, такого бути не може: нейрон дає тільки один вихід. Тому схема на рис.2.6 не є вірною. Коректна схема, коли треба змоделювати архітектуру «один вхід та декілька виходів», має складатись з кількох нейронів, кожний з яких дає один вихід (рис. 2.7).

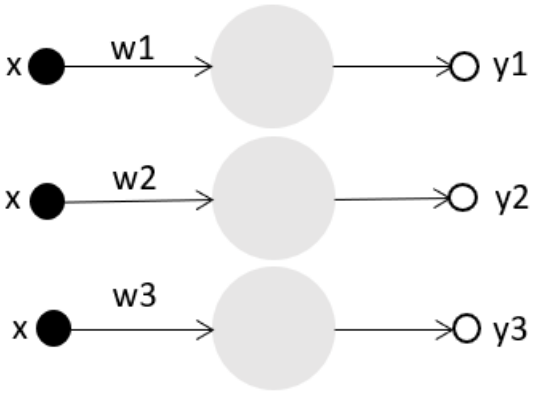


Рис. 2.7. Мережа типу «три входи-три виходів»

На схемі показані три нейрони, які мають різні вагові коефіцієнти, але на вхід яких подається одна і таж сама ознака. І формула, за якою обчислюємо значення виходів, така ж сама, як і для моделі мережі №1 (формула 2.1).

Призначимо в якості вхідного значення ознаку win/lose – відношення загальної кількості перемог до поразок. А мережа має здійснити прогноз трьох значень (в дужках наведені відповідні бінарні значення на виходах, наведені ознаки взяті як приклад):

- вихід $y1$: відсоток травм членів команди (значення від 0 до 1);
- вихід $y2$: перемога (1) чи поразка (0)?
- вихід $y3$: емоційний стан - радість (1) чи засмучення (0).

Код, який реалізує даний вид мережі, приведений у лістингу 2.3. Звертаємо увагу, що в нашому оригінальному датасеті відсутні данні для

співставлення з виходами $y2$ та $y3$. Тому таку мережу натренувати на існуючому датасеті не можливо. Також слід зазначити, що на виході ми фактично отримуємо вектор прогнозів (бо їх кілька). Тому вихідні значення доцільно зберігати, як і вагові коефіцієнти, у виді списку python.

Лістинг 2.3. Модель мережі «один вхід-кілька виходів» [10]

```
 1. def neural_network(input, weights):
 2.     pred = ele_mul(input,weights)
 3.     return pred
 4.
 5. def ele_mul(number,vector):
 6.     output = [0,0,0]
 7.     for i in range(len(vector)):
 8.         output[i] = number * vector[i]
 9.     return output
10.
11. weights = [0.3, 0.2, 0.9]
12. wlrec = [0.65, 0.8, 0.8, 0.9]
13.
14. input = wlrec[0]
15.
16. pred = neural_network(input,weights)
17. print(pred)
```

*Коментарі по ключових рядках коду:*

1-3: задаємо функцію neural_network, яка повертає результат прогнозу; вхідними параметрами є значення входу (скаляр), та величини вагових коефіцієнтів weights (вектор);

2: застосовуємо метод ele_mul (input,weights);

5-9: реалізація функції ele_mul: на вході маємо число number та вектор vector, які треба перемножити поелементно;

11: задаємо вагові коефіцієнти у вигляді списку з трьох елементів;

13-15: задаємо значення ознак, які будемо подавати на вход (ознака wlrec), також у вигляді списку;

17: формуємо одне вхідне значення input (значення ознаки wlrec за індексом 0);

16: обчислюємо прогноз для векторів входів та вагових коефіцієнтів мережі;

17: виводимо значення прогнозу в консоль.

Результат роботи даного коду, виконаний в середовищі Jupiter Notebook, показаний на скріншоті (рис.2.8).

```
[0.195, 0.13, 0.5850000000000001]
```

Рис. 2.8 Результат виконання коду з лістингу 3

*Аналіз роботи коду.* Фактично, даний код моделює роботу трьох різних нейронів, які працюють за формулою 2.1. Але на які подається один і той же вхідний сигнал.

*Модель нейронної мережі «кілька входів – кілька виходів»*

Розглянемо ситуацію, коли в мережі є кілька входів та декілька виходів (рис.2.9). Тобто, мережа на основі трьох ознак має здійснювати три різні прогнози. Фактично, така мережа має складатись з кількох окремих нейронів (трьох), на кожний з яких подаються данні трьох ознак окремо. Математична формула для обчислення кожного виходу має вигляд формули 2.2 (зважена сума входів).

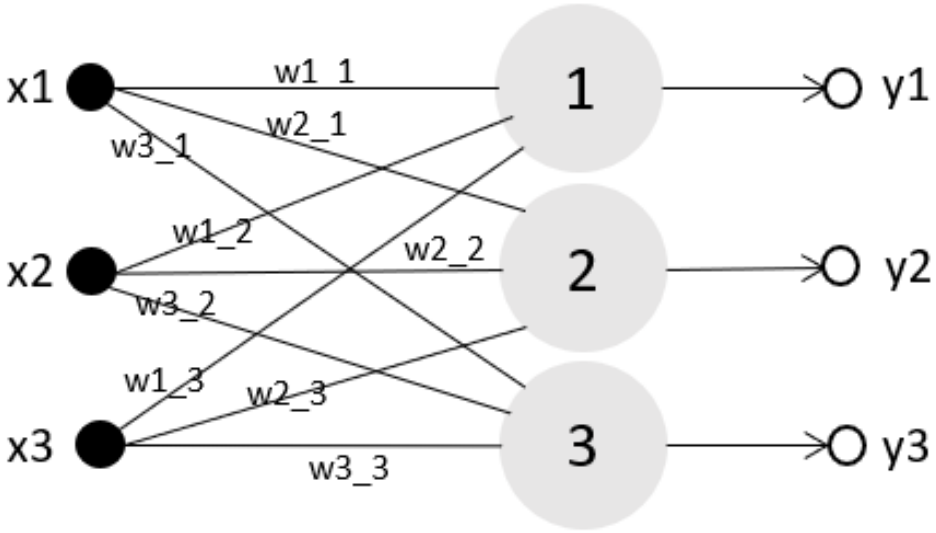


Рис. 2.9. Модель нейронної мережі «кілька входів – кілька виходів»

В якості вхідних ознак оберемо вже відомі:

- вхід x1: number_of_toes: середня кількість ігор, зіграних перед поточною грою;
- вхід x2: win/lose – відношення виграшів/програшів ігор перед поточною грою;
- вхід x3: number of fans – кількість болільників на поточній ігрі.

А в якості виходів такі прогнозні величини:

- вихід $y1$: відсоток травм членів команди (значення від 0 до 1);
- вихід $y2$: перемога (1) чи поразка (0)?
- вихід $y3$: емоційний стан - радість (1) чи засмучення (0).

В коді реалізації цієї архітектури цікавим буде вирішення задачі способу збереження сукупності вагових коефіцієнтів: їх загальна кількість – 9 штук. Існують різні способи їх групування, найкращим – матричне представлення. Матриця буде складатись з 3 рядків та 3 стовпчиків (колонок). Кожний рядок міститиме вагові коефіцієнти, які характеризують зв'язки, спрямовані в 1,2,3 нейрони відповідно. Схема матриці зображена у формулі (2.3).

$$\begin{bmatrix} w1\_1 & w1\_2 & w1\_3 \\ w2\_1 & w2\_2 & w2\_3 \\ w3\_1 & w3\_2 & w3\_3 \end{bmatrix} \quad (2.3)$$

Тобто, на 1-му рядку знаходяться ваги зав'язків, які ведуть до 1-го нейрону від всіх входів. На 2-му рядку знаходяться ваги зав'язків, які ведуть до 2-го нейрону, від всіх входів. І так далі. Якщо представити ваги як $w\_i\_j$, то і означає рядок (до якого нейрону), j означає колонку (від якого нейрону).

З врахуванням наведеного, код моделі буде наступним (лістинг 2. 4).

Лістинг 2.4. Модель мережі «кілька входів – кілька виходів» [10]

```
def neural_network(input, weights):
    pred = vect_mat_mul(input,weights)
    return pred

def vect_mat_mul(vect,matrix):
    output = [0,0,0]
    for i in range(len(vect)):
        output[i] = w_sum(vect,matrix[i])
    return output

def w_sum(a,b):
    output = 0
    for i in range(len(a)):
        output = output + (a[i] * b[i])
    return output

# matrix of weights
# columns contains weights FROM the same input
```

```
19. # rows containg weights TO one output
20.
21.             #toes % win # fans
22. weights = [ [0.1, 0.1, -0.3], # hurt?
23.             [0.1, 0.2, 0.0], # win?
24.             [0.0, 1.3, 0.1] ] # sad?
25.
26. # dataset
27. toes =  [8.5, 9.5, 9.9, 9.0]
28. wlrec = [0.65,0.8, 0.8, 0.9]
29. nfans = [1.2, 1.3, 0.5, 1.0]
30.
31. # forming input vector
32. input = [toes[0],wlrec[0],nfans[0]]
33.
34. #calculate prediction
35. pred = neural_network(input,weights)
36. print(pred)
```

*Коментарі по ключових рядках коду:*

1-3: задаємо функцію neural_network, яка повертає результат прогнозу; вхідними параметрами є значення входу (вектор), та матриці вагових коефіцієнтів; функція має повернути вектор прогнозних значень;

2: виклик функції vect_mat_mul, на вхід якої подаємо тіж самі вектор та матрицю;

5-9: функція vect_mat_mul формує вектор, кожний елемент якого є результатом зваженої суми вагів з одного рядка матриці вагів, та всіх входів мережі; це здійснюється функцією w_sum;

22-24: формування матриці вагів, де кожний рядок містить ваги, які йдуть від усіх входів до кожного нейрону; кожний стовпчик означає вхід, від якого йдуть ваги;

26-29: отримуємо датасет (тільки частина для входів;

32: формуємо вектор входу;

35-36: отримуємо прогнозне значення на виходах.

Результат роботи даного коду представлений на рис. 2.10.

```
[0.555, 0.9800000000000001, 0.9650000000000001]
```

Рис.2.10 Результат виконання коду з лістингу 2.4

## *Частина 2.* Завдання для самостійного виконання

1. Уважно дослідить улаштування датасету та кодів всіх моделей НМ з теоретичної частини в Вашому середовищі розробнику Python.

2. Створіть власний датасет згідно приведених нижче тем, на основі якого можна навчити Вашу нейронну мережу здійснювати прогнози. З назви тематики визначить суть явище, яке має бути описане в часовому аспекті набором характеристик (ознак). Наприклад, тематика «погода» може бути описана датасетом, показаним у табл.2.2:

*Таблиця 2.2*

**Структура датасету для теми «погода»**

| Часовий інтервал | Назви характеристик | | | |
|---|---|---|---|---|
| | Середньоденна температура (градуси Цельсія) | Рівень вологості (%) | Прогноз дощу (1-буде, 0-не буде) | Прогнозний рівень опадів (мм) |
| Понеділок | 28 | 34 | 0 | 0 |
| Вівторок | 26 | 78 | 1 | 36 |
| Середа | 25 | 12 | 0 | 2 |
| Четвер | 30 | 67 | 1 | 10 |

Датасет може включати як прямі характеристики явища (температура, вологість, тиск і т.п.), так і похідні. Тобто, такі які опосередковано можуть впливати на погоду, або бути її складовою: фаза Місяця, наявність озонових «дірок», плям на Сонці і т.п.

Тематика:

- погода
- футбольна гра
- навчальне заняття
- ціна валюти (або бікоіну) на ринку
- настрій домашньої кішки
- ігра в шахи
- рослина у горщику на вікні
- *(або запропонуйте свій варіант)*

3. Реалізуйте у коді python на базі власного датасету одну з архітектур мереж, розглянутих у лабораторній роботі (оберіть самостійно).

5. Оформити звіт, в якому відобразити:

- ✓ Постановку завдання (завдання 2-3).
- ✓ Короткі теоретичні відомості згідно розробленого Вами датасету та реалізованої мережі
- ✓ Код програми із коментарями.
- ✓ Результати роботи програми.
- ✓ Висновки.

*Запитання до лабораторної роботи*

1. Що таке нейрон, з чого складається штучний нейрон?
2. Яка математична модель нейрона?
3. Що таке пряме розповсюдження?
4. Які конфігурації нейронів та найпростіших поєднань нейронів ви знаєте?
5. Як реалізується в коді вектор вхідних ознак? вектор виходів? матриця вагів?

## ЛАБОРАТОРНА РОБОТА №3

**Тема:** ***Реалізація схем контрольованого навчання на мові python та порівняння їх ефективності***

**Мета:** навчитися реалізовувати пряме розповсюдження у нейронні мережі, різні варіанти контрольованого навчання та оцінювати їх ефективність.

### *Частина 1.* Основні теоретичні відомості

З [10,11] вам відомо, що існує три схеми реалізації контрольованого навчання:

- стохастичний градієнтний спуск (Stochastic Gradient Descent, SGD);
- пакетний режим (Batch);
- міні-пакетний режим (Mini Batch).

Розглянемо реалізацію кожного з них для тренування нейронної мережі, яка складається з одного шару та одного нейрону в шарі (рис. 3.1). Тренування мереж більш складного характеру буде базуватись на аналогічних принципах та кодових фрагментах.

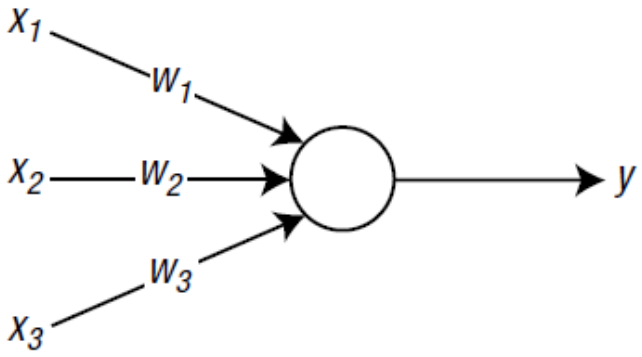


Рис. 3.1. Схема нейрону для тренування

На виході буде застосована сигмовидна функція активації, яка обчислюється за формулою 3.1 та візуально має вигляд, показаний на рис. 3.2.

$$\varphi(x) = \frac{1}{1+e^{-x}} \qquad (3.1)$$

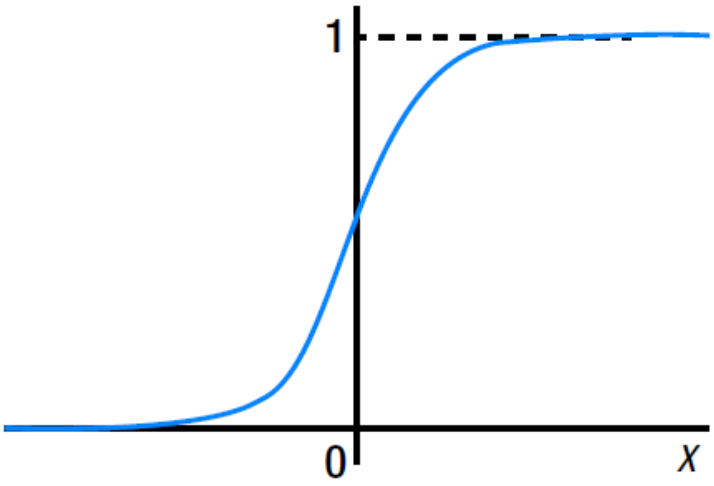


Рис. 3.2. Графік сигмовидної функції активації

Для тренування мережі скористаємось датасетом №1, який представлений у таблиці 3.1.

*Таблиця 3.1*

**Датасет №1**

| x1 | x2 | x3 | correct y |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Правилом навчання даної мережі буде дельта-правило (delta rule), а процес навчання буде базуватись на таких математичних перетвореннях (3.2-3.4), детальний опис яких наведений у [11]:

$$\delta_i = \varphi(v_i)\big(1 - \varphi(v_i)\big)e_i \tag{3.2}$$

$$\Delta\omega_{ij} = \alpha\delta_i x_i \tag{3.3}$$

$$\omega_{ij} \leftarrow \omega_{ij} + \Delta\omega_{ij} \tag{3.4}$$

Навчання за дельта-правилом будемо реалізовувати для двох схем навчання: SGD та пакетний режим. Розглянемо їх окремо. Для всіх фрагментів коду застосовується бібліотека numpy, тому першим рядком має йти її імпорт:

```
1. import numpy as np
```

*Реалізація SGD методу навчання*

Спільним для всіх методів навчання є функція активації, код який наведений нижче.

```
1. def Sigmoid(x):
2.     return 1.0 / (1.0 + np.exp(-x))
```

Реалізація формул (3.2-3.4) у схемі SGD відбуватиметься у наступному коді [12]:

```
 1. def DeltaSGD(W, X, D):
 2.     alpha = 0.9
 3.
 4.     N = 4
 5.     for k in range(N):
 6.         x = X[k, :].T
 7.         d = D[k]
 8.
 9.         v = np.matmul(W, x)
10.         y = Sigmoid(v)
11.
12.         e     = d - y
13.         delta = y*(1-y) * e
14.
15.         dW = alpha*delta*x
16.
17.         W[0][0] = W[0][0] + dW[0]
18.         W[0][1] = W[0][1] + dW[1]
19.         W[0][2] = W[0][2] + dW[2]
20.
21.     return W
```

Функція DeltaSGD(W, X, D) має три вхідні параметри: матрицю вагів W, X – частина датасету для тренування (колонки $x_1, x_2, x_3$ з датасету №1), D – частина датасету з вірними виходами мережі (колонка correct у з датасету №1).

*Коментарі по коду:*

2: задаємо коефіцієнт швидкості навчання;

6: взяти з матриці X всі значення k-того рядка (це буде вектор-рядок);

7: взяти з вектора-стовпчика D значення k-го рядка;

9: здійснити множення матриці вагів W на вектор вхідних значень x (отримуємо зважену суму v);
10: обрахувати вихід нейрону y, тобто значення сигмовидної функції для отриманої зваженої суми v;
12: обчислюємо абсолютне значення похибки (істинний вихід мінус отриманий результат);
13: обчислюємо значення delta;
15: обраховуємо поправки для вагових коефіцієнтів у вигляді вектору dW;
17-19: корегуємо вагові коефіцієнти для 0-2 вхідних сигналів 0-го нейрону.

Таким чином виклик функції DeltaSGD(…) здійснює навчання в межах однієї епохи, тобто модифікує вагову матрицю для всіх точок датасету один раз.

Для того, щоб натренувати мережі на конкретній кількості епох, треба помістити виклик функції DeltaSGD(…) в цикл та задати кількість ітерацій. Для цього призначений код функції TestDeltaSGD() для налаштування мережі та здійснення тренування на заданій кількості епох [12]:

```
 1. def TestDeltaSGD():
 2.     X = np.array([[0, 0, 1],
 3.                   [0, 1, 1],
 4.                   [1, 0, 1],
 5.                   [1, 1, 1]])
 6.
 7.     D = np.array([[0],
 8.                   [0],
 9.                   [1],
10.                   [1]])
11.
12.     W = 2*np.random.random((1, 3)) - 1
13.
14.     for _epoch in range(40000):
15.         W = DeltaSGD(W, X, D)
16.
17.     N = 4
18.     for k in range(N):
19.         x = X[k,:].T
20.         v = np.matmul(W, x)
21.         y = Sigmoid(v)
22.         print(y)
```

*Коментарі по коду:*
2-5: обираємо фрагмент датасету для тренування та записуємо у матрицю X;

7-10: обираємо фрагмент датасету з вірними значеннями виходу (колонка correct y)
12: створюємо вихідну (не «навчену») матрицю вагових коефіцієнтів (значення від -1 до 1);
14-15: тренуємо мережі на 40_000 епохах;
17-22: перевіряємо як працює натренована мережа, подавши на вхід всі точки датасету для тренування

У результате виклику функції TestDeltaSGD() ви маєте отримати результат, близький до наступного (скріншот на рис. 3.3)

```
[0.00504953]
[0.00411405]
[0.99664428]
[0.99588052]
```

Рис. 3.3 Результат роботи натренованої мережі методом SGD

*Реалізація пакетного методу навчання*

Реалізація пакетного навчання здійснюється у функції DeltaBatch(W, X, D), код приведений нижче [12]:

```
 1. def DeltaBatch(W, X, D):
 2.     alpha = 0.9
 3.     dWsum = np.zeros(3)
 4.
 5.     N = 4
 6.     for k in range(N):
 7.         x = X[k, :].T
 8.         d = D[k]
 9.
10.         v = np.matmul(W, x)
11.         y = Sigmoid(v)
12.
13.         e     = d - y
14.         delta = y*(1-y) * e
15.         dW = alpha * delta * x
16.
17.         dWsum = dWsum + dW
18.
19.     dWavg = dWsum / N
20.
21.     W[0][0] = W[0][0] + dWavg[0]
```

```
22.     W[0][1] = W[0][1] + dWavg[1]
23.     W[0][2] = W[0][2] + dWavg[2]
24.
25.     return W
```

Особливістю як методу, так і реалізації, є спосіб однократної зміни всіх вагових коефіцієнтів після навчання на всіх точках. Для цього у р. 3 вводиться нова матриця вагових коефіцієнтів, яка заповнюється нулями. В циклі навчання по всіх точках датасету (р.5-17) застосовується аналогічний до DeltaSGD код. З тією різницею, що зміни вагів dW додаються до матриці dWsum. Після проходження всіх точек датасету, в р19 обчислюються середні значення змін вагів, і в р.21-23 здійснюється оновлення матриці вагів. Оновлена матриця повертається в точку виклику.

Для навчання застосовується функція TestDeltaBatch(), код наведений нижче [12]:

```
 1. def TestDeltaBatch():
 2.     X = np.array([[0, 0, 1],
 3.                   [0, 1, 1],
 4.                   [1, 0, 1],
 5.                   [1, 1, 1]])
 6.
 7.     D = np.array([[0],
 8.                   [0],
 9.                   [1],
10.                   [1]])
11.
12.     W = 2*np.random.random((1, 3)) - 1
13.
14.     for _epoch in range(40000):
15.         W = DeltaBatch(W, X, D)
16.
17.     N = 4
18.     for k in range(N):
19.         x = X[k,:].T
20.         v = np.matmul(W, x)
21.         y = Sigmoid(v)
22.         print(y)
```

Для запуску навчання треба викликати зазначену функцію `TestDeltaBatch()`, в результаті чого маєте отримати вивід в консоль (рис. 3. 4):

```
[0.01020539]
[0.00829801]
[0.99323761]
[0.99168025]
```

Рис. 3.4 Результат роботи натренованої мережі методом SGD

*Порівняння швидкості навчання методом SGD та пакетним*

Для того, щоб оцінити швидкості навчання за різними схемами, розглянемо функцію ComparsionSGDvsBatch(). Для її роботи потрібна бібліотека matplotlib, тож не забудьте про імпорт:

```
1. import matplotlib.pyplot as plt
```

Код функції ComparsionSGDvsBatch() [12]:

```
 1. def ComparsionSGDvsBatch():
 2.     X = np.array([[0, 0, 1],
 3.                   [0, 1, 1],
 4.                   [1, 0, 1],
 5.                   [1, 1, 1]])
 6.
 7.     D = np.array([[0],
 8.                   [0],
 9.                   [1],
10.                   [1]])
11.
12.     E1 = np.zeros(1000)
13.     E2 = np.zeros(1000)
14.
15.     W1 = 2*np.random.random((1, 3)) - 1
16.     W2 = np.array(W1)
17.
18.     for epoch in range(1000):
19.         W1 = DeltaSGD(W1, X, D)
20.         W2 = DeltaBatch(W2, X, D)
21.
22.         es1 = 0
23.         es2 = 0
24.         N   = 4
25.         for k in range(N):
26.             x = X[k, :].T
27.             d = D[k]
28.
29.             v1 = np.matmul(W1, x)
```

```
            y1 = Sigmoid(v1)
            es1 = es1 + (d - y1)**2

            v2 = np.matmul(W2, x)
            y2 = Sigmoid(v2)
            es2 = es2 + (d - y2)**2

        E1[epoch] = es1/N
        E2[epoch] = es2/N


    SGD,   = plt.plot(E1, 'r')
    Batch, = plt.plot(E2, 'b:')
    plt.xlabel("Epoch")
    plt.ylabel("Average of Training Error")
    plt.legend([SGD, Batch], ['SGD', 'Batch'])

    plt.show()
```

Виклик функції ComparsionSGDvsBatch() призведе до появи вікна з графіком, з якого можна зробити висновки щодо ефективності методів навчання. Можливий вигляд графіку наведений на рис. 3.5

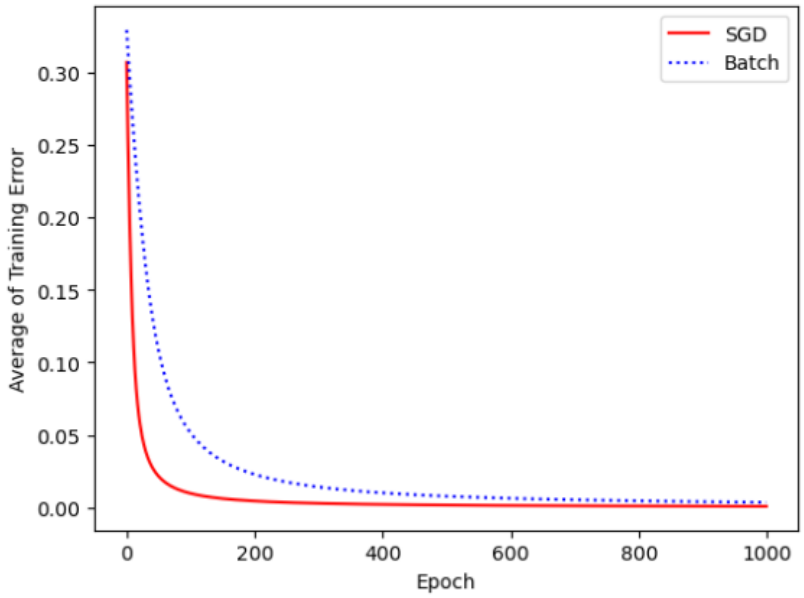


Рис. 3.5. Порівняння швидкості навчання методів SGD та пакетного

## *Частина 2.* Завдання для самостійного виконання

1. Здійсніть всі схеми навчання, наведені у теоретичній частині. Також здійсніть порівняння їх ефективності, зробіть висновки.

2. Дослідить, як кількість епох навчання впливає на точність отриманих прогнозних даних на виході нейронної мережі. Почніть дослідження з 40_000 епох у сторону як зменшення, так і збільшення. Зробіть висновок.

3. Яку кількість епох треба зробити в SGD схемі навчання, щоб отримати результат навчання з точністю 5 %?

4. Яку кількість епох треба зробити в пакетній схемі навчання, щоб отримати результат навчання з точністю 5 %?

5. Який вплив здійснює на навчання параметр alpha? Для двох різних значень alpha (в допустимих межах цього параметру) виведіть криві залежності похибки від часу для режимів навчання SGD та пакетного швидкостей (аналогічні до рис. 3.5). Зробіть висновок.

6. Оформити звіт, в якому відобразити:

- ✓ Постановку завдання (завдання 2-5).
- ✓ Короткі теоретичні відомості згідно схем навчання
- ✓ Коди програм
- ✓ Результати роботи програм
- ✓ Висновки для кожного завдання

Питання до лабораторної роботи

1. Що таке нейрон, з чого складається штучний нейрон?
2. Яка математична модель нейрона?
3. Що таке пряме розповсюдження?
4. Як улаштована схема навчання SGD?
5. Як працює пакетне навчання?
6. Як співвідносяться епохи навчання в SGD та пакетному режимах?

## ЛАБОРАТОРНА РОБОТА №4

**Тема:** ***Реалізація алгоритму зворотного розповсюдження для багатошарових мереж на python та його оптимізація***

**Мета:** Навчитися моделювати зворотне розповсюдження на мові python.

### *Частина 1.* Основні теоретичні відомості

*Задача про XOR та застосування алгоритму зворотного розповсюдження*

З [10,11] вам відомо, що для навчання багатошарових мереж, застосовується алгоритм зворотного розповсюдження (back-propagation algorithm). В цій лабораторній роботі ми розглянемо реалізацію цього алгоритму на базі мови python, для нескладної мережі, схема якої представлена на рис.4.1.

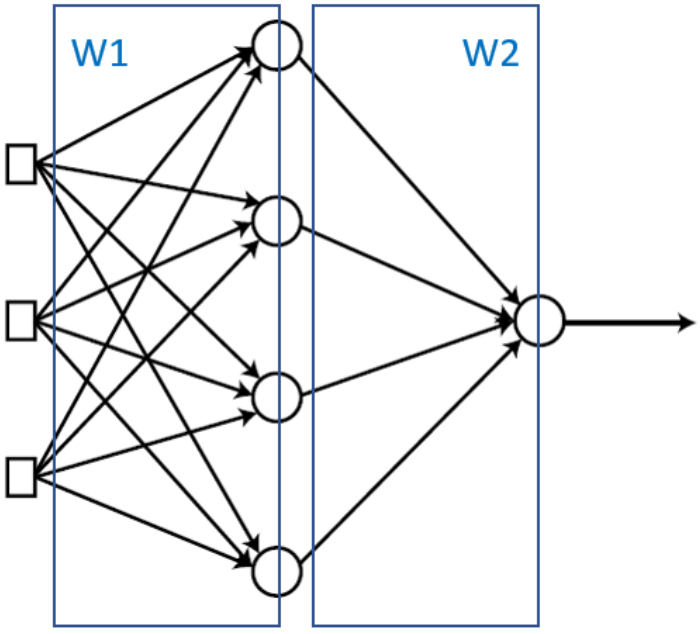


Рис. 4.1 Схема устрою мережі з одним прихованим шаром

В мережі, показаної на рис. 4.1, є три шари: вхідний, який має 3 вхідні вузла; вихідній, який має один нейрон; проміжний (прихований), який містить 4 нейрони. Матриця вагових коефіцієнтів W1 містить коефіцієнти прихованого шару, а W2 - вихідного.

Мережу будемо тренувати на датасеті, який показаний у табл.4.1. Якщо розглянути тільки входи x1,x2, та y, то маємо таблицю істинності для логічної операції XOR. Звідси походить назва нашої задачі.

*Таблиця 4.1*

**Датасет для навчання (x1-x3 – вхідні значення, y- істинний вихід)**

| x1 | x2 | x3 | y |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |

Для прихованого та вихідного шарів будемо застосовувати сигмовидну функцію активації. Також застосуємо SGD методику навчання, як саму просту для реалізації. Хоча можна застосувати і пакетний або напів-пакетний методи, але це можете реалізувати самостійно. Нижче наведений код, який реалізує весь процес навчання.

Лістинг 4.1. Реалізація алгоритму зворотного розповсюдження [12]

```
import numpy as np

def Sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def BackpropXOR(W1, W2, X, D):
    alpha = 0.9

    N = 4
    for k in range(N):
        x = X[k, :].T
        d = D[k]

        v1 = np.matmul(W1, x)
        y1 = Sigmoid(v1)
        v  = np.matmul(W2, y1)
        y  = Sigmoid(v)

        e     = d - y
        delta = y*(1-y) * e

        e1     = np.matmul(W2.T, delta)
        delta1 = y1*(1-y1) * e1

        dW1 = (alpha*delta1).reshape(4, 1) * x.reshape(1, 3)
```

```
26.         W1  = W1 + dW1
27.
28.         dW2 = alpha * delta * y1
29.         W2  = W2 + dW2
30.
31.     return W1, W2
32.
33. def TestBackpropXOR():
34.     X = np.array([[0, 0, 1],
35.                   [0, 1, 1],
36.                   [1, 0, 1],
37.                   [1, 1, 1]])
38.
39.     D = np.array([[0],
40.                   [1],
41.                   [1],
42.                   [0]])
43.
44.     W1 = 2*np.random.random((4, 3)) - 1
45.     W2 = 2*np.random.random((1, 4)) - 1
46.
47.     for _epoch in range(10000):
48.         W1, W2 = BackpropXOR(W1, W2, X, D)
49.
50.     N = 4
51.     for k in range(4):
52.         x  = X[k ,:].T
53.         v1 = np.matmul(W1, x)
54.         y1 = Sigmoid(v1)
55.         v  = np.matmul(W2, y1)
56.         y  = Sigmoid(v)
57.         print(y)
58.
59. TestBackpropXOR()
```

*Коментарі по коду:*

6-31: визначення функції BackpropXOR(W1, W2, X, D), яка здійснює зміну матриць вагових коефіцієнтів один раз, згідно алгоритму зворотного розповсюдження; W1, W2 – це матриці вагових коефіцієнтів відповідних рівнів мережі (див. рис.4.1), X та D – вектор входів та вектор коректних відповідей (з датасету); функція повертає оновлені матриці;

14-15: обчислюються вектори v1 та y1, які містять зважені суми та виходи першого прихованого шару

16-17: обчислюються вектори v та y, які містять зважену суми та вихід вихідного шару мережі (один вихідний нейрон).

19-20: обчислення похибки e та $\delta$ у вихідному шарі;

22-23: обчислення похибки e1 та $\delta 1$ у прихованому шарі, за алгоритмом зворотного розповсюдження; обчислення відбувається згідно формули 4.1:

$$\begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \end{bmatrix} = W_2^T \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} \tag{4.1}$$

25-26: обчислення поправок для вагових коефіцієнтів першого прихованого шару та внесення поправок до вагової матриці W1;

28-29: обчислення поправок для вагових коефіцієнтів другого, вихідного шару, та внесення поправок до вагової матриці W2;

33-57: визначення функції TestBackpropXOR(), яка здійснює всю роботу по навчанню мережі та виводить у р.57 результат її роботи при подачі на вхід вектору X з датасету;

34-37: задаємо вхідні значення для мережі, для всіх точок датасету;

39-42: задаємо вектор вірних відповідей з датасету, для всіх точок датасету;

44-45: ініціалізуємо матриці вагових коефіцієнтів W1 та W2 випадковими числами з діапазону (-1;1);

47-48: здійснюємо навчання на 10_000 епохах;

50-56: перевіряємо роботу мережі, подаючи послідовно на неї всі точки датасету;

57: вивід відповіді мережі.

У результаті виконання наведеного коду, в консолі отримуємо такий вивід:

```
[0.00714616]
[0.99025643]
[0.98814186]
[0.01551047]
```

Рис. 4.2 Результат роботи натренованої мережі
на основі алгоритму зворотного розповсюдження

Якість навчання можна перевірити, порівнявши вектор з рис. 4.2. та істинні відповіді з датасету з таблиці 4.1. Отримані значення достатньо близькі до ідеальних.

Точність можна порахувати, обрахувавши абсолютну (формула 4.2) та відносні похибки (формула 4.3):

$$\Delta y = \left| y_{real} - y_{expected} \right|, \tag{4.2}$$

$$\delta y = \frac{\left| y_{expected} - y_{real} \right|}{y_{real}} \times 100\%, \tag{4.3}$$

де $y_{real}$ – значення на виході мережі (яке повертає натренована мережа); $y_{expected}$ – істинне (очікуване) значення, яке беремо з датасету; $\delta y$ – відносна похибка, у відсотках.

Наприклад, для другої точки, $\delta y = \frac{1-0.99025643}{0.99025643} \times 100\% \cong 0.98\%$

Отже, мережа дозволяє отримувати результати на тренувальних даних з точністю на рівні одного відсотка.

*Застосування імпульсу в алгоритмі зворотного розповсюдження*

Для оновлення вагових коефіцієнтів можна застосувати імпульс (momentum) [11], який буде спрямовувати корегування вагових коефіцієнтів в певному напрямку. Його застосування може покращити стабільність навчання та підвищити швидкість навчального процесу.

Лістинг коду для реалізації алгоритму навчання із зворотнім наведений нижче.

Лістинг 4.2. Реалізація алгоритму зворотного розповсюдження із застосуванням імпульсу [12]

```
 1. import numpy as np
 2.
 3.
 4. def Sigmoid(x):
 5.     return 1.0 / (1.0 + np.exp(-x))
 6.
 7. def BackPropMmt(W1, W2, X, D):
 8.     alpha = 0.9
 9.     beta  = 0.9
10.
11.     mmt1 = np.zeros_like(W1)
12.     mmt2 = np.zeros_like(W2)
13.
```

```
    N = 4
    for k in range(N):
        x = X[k, :].T
        d = D[k]

        v1 = np.matmul(W1, x)
        y1 = Sigmoid(v1)
        v  = np.matmul(W2, y1)
        y  = Sigmoid(v)

        e     = d - y
        delta = y*(1-y) * e

        e1     = np.matmul(W2.T, delta)
        delta1 = y1*(1-y1) * e1

        dW1  = (alpha*delta1).reshape(4, 1) * x.reshape(1, 3)
        mmt1 = dW1 + beta*mmt1
        W1   = W1 + mmt1

        dW2  = alpha * delta * y1
        mmt2 = dW2 + beta*mmt2
        W2   = W2 + mmt2

    return W1, W2

def TestBackpropMmt():
    X = np.array([[0, 0, 1],
                  [0, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]])

    D = np.array([[0],
                  [1],
                  [1],
                  [0]])

    W1 = 2*np.random.random((4, 3)) - 1
    W2 = 2*np.random.random((1, 4)) - 1

    for _epoch in range(10_000):
        W1, W2 = BackPropMmt(W1, W2, X, D)

    N = 4
    for k in range(N):
        x  = X[k, :].T
        v1 = np.matmul(W1, x)
        y1 = Sigmoid(v1)
        v  = np.matmul(W2, y1)
        y  = Sigmoid(v)
        print(y)
```

```
65.
66. if __name__ == '__main__':
67.     TestBackpropMmt()
```

*Коментарі по коду*

7-38: визначення функції BackPropMmt(W1, W2, X, D), яка по суті та результату роботи аналогічна функції BackpropXOR(W1, W2, X, D);

9 – додається новий коефіцієнт beta, який за значення менше 1, і регулює вплив попередніх значень «моментів імпульсу»;

31: розрахунок поправки dW1 для вагових коефіцієнтів (як і в минулому прикладі);

32: розрахунок фактичної поправки mmt1 на основі dW1, але з урахуванням моменту;

33: зміна вагових коефіцієнтів для матриці W1 на величину mmt1;

34-36: аналогічні дії і для матриці W2.

В результаті виконання наведеного коду, в консолі отримуємо такий вивід:

```
[0.00481611]
[0.99395737]
[0.99265064]
[0.01142023]
```

Рис. 4.3. Результат роботи натренованої мережі на основі алгоритму зворотного розповсюдження та застосуванням моменту

Як можна побачити, величина похибки для другого підходу менша, при тій самі кількості епох. Це значить, що для досягнення однієї і тієї ж точності, з використанням моменту мережа досягає швидше.

*Використання крос-ентропійної функції під час обчислення похибки мережі*

Функція втрат (cost function), і більш конкретно – крос-ентропійна функція, застосовується на рівні всієї нейронної мережі для оцінки того, наскільки якісно модель прогнозує виходи на основі заданих вхідних даних [11]. Для бінарної класифікації, а в нас задача XOR фактично зведена до банірної класифікації – на виходах два значення 0 або 1, крос-ентропійна функція має вигляд:

$$E = -d\ln(y) - (1-d)\ln(1-y), \tag{4.4}$$

де $d$ – істине значення виходу,

$y$ – те, що дає мережа.

Коли ми використовуємо сигмоїдну функцію активації для вихідного нейрона та крос-ентропійну функцію втрат, дельта дорівнює похибці через те, як ці дві функції поєднуються в процесі диференціювання. Як наслідок, маємо що

$$\delta = e \tag{4.5}$$

Але це «дельта» для шару W2. Для шару W1 вона залишається як і у попередніх випадках. Отже, код реалізації алгоритму зворотного розповсюдження в цьому випадку буде наступним:

Лістинг 4.4. Реалізація алгоритму зворотного розповсюдження під час використання функції втрат у виді крос-ентропійної функції [12]

```
import numpy as np


def Sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def BackpropCE(W1, W2, X, D):
    alpha = 0.9

    N = 4
    for k in range(N):
        x = X[k, :].T
        d = D[k]

        v1 = np.matmul(W1, x)
        y1 = Sigmoid(v1)
        v  = np.matmul(W2, y1)
        y  = Sigmoid(v)

        e     = d - y
        delta = e

        e1     = np.matmul(W2.T, delta)
        delta1 = y1*(1-y1) * e1

        dW1 = (alpha*delta1).reshape(4, 1) * x.reshape(1, 3)
```

```
27.         W1  = W1 + dW1
28.
29.         dW2 = alpha * delta * y1
30.         W2  = W2 + dW2
31.
32.     return W1, W2
33.
34. def TestBackpropCE():
35.     X = np.array([[0, 0, 1],
36.                   [0, 1, 1],
37.                   [1, 0, 1],
38.                   [1, 1, 1]])
39.
40.     D = np.array([[0],
41.                   [1],
42.                   [1],
43.                   [0]])
44.
45.     W1 = 2*np.random.random((4, 3)) - 1
46.     W2 = 2*np.random.random((1, 4)) - 1
47.
48.     for _epoch in range(10000):
49.         W1, W2 = BackpropCE(W1, W2, X, D)
50.
51.     N = 4
52.     for k in range(N):
53.         x  = X[k, :].T
54.         v1 = np.matmul(W1, x)
55.         y1 = Sigmoid(v1)
56.         v  = np.matmul(W2, y1)
57.         y  = Sigmoid(v)
58.         print(y)
59.
60. if __name__ == '__main__':
61.     TestBackpropCE()
```

*Коментарі по коду (суттєві зміни)*

7-32: визначення функції BackpropCE(W1, W2, X, D), яка по суті та результату роботи аналогічна функції BackpropXOR(W1, W2, X, D) з лістингу 4.1;

20: обчислення «дельти» за формулою (4.5).

У результаті виконання наведеного коду, в консолі отримуємо такий вивід:

```
[0.00051258]
[0.99978372]
[0.99959669]
[0.00025083]
```

Рис. 4.4. Результат роботи натренованої мережі на основі алгоритму зворотного розповсюдження та крос-ентропійної функції обчислення похибки

Стосовно точності навчання можете зробити висновки самостійно.

*Порівняння функцій втрат*

Якщо розглянути коди з лістингів 4.1 та 4.3, то вони відрізняються тільки в одному рядку. По суті алгоритму – вони використовують різні функції втрат на вихідному шарі. Цікаво дослідити, як обрання функції втрат впливає на якість навчання. Для цього застосуємо код з лістингу 4.5.

Лістинг 4.5. Порівняння реалізацій алгоритму зворотного розповсюдження при використанні різних функцій втрат (сума квадратів різниць та крос-ентропійна функція) [12]

```
 1. import numpy as np
 2. import matplotlib.pyplot as plt
 3.
 4. def Sigmoid(x):
 5.     return 1.0 / (1.0 + np.exp(-x))
 6.
 7. def BackpropCE(W1, W2, X, D):
 8.     alpha = 0.9
 9.
10.     N = 4
11.     for k in range(N):
12.         x = X[k, :].T
13.         d = D[k]
14.
15.         v1 = np.matmul(W1, x)
16.         y1 = Sigmoid(v1)
17.         v  = np.matmul(W2, y1)
18.         y  = Sigmoid(v)
19.
20.         e     = d - y
21.         delta = e
22.
23.         e1     = np.matmul(W2.T, delta)
24.         delta1 = y1*(1-y1) * e1
```

```
25.
26.         dW1 = (alpha*delta1).reshape(4, 1) * x.reshape(1, 3)
27.         W1  = W1 + dW1
28.
29.         dW2 = alpha * delta * y1
30.         W2  = W2 + dW2
31.
32.     return W1, W2
33.
34.
35. def BackpropXOR(W1, W2, X, D):
36.     alpha = 0.9
37.
38.     N = 4
39.     for k in range(N):
40.         x = X[k, :].T
41.         d = D[k]
42.
43.         v1 = np.matmul(W1, x)
44.         y1 = Sigmoid(v1)
45.         v  = np.matmul(W2, y1)
46.         y  = Sigmoid(v)
47.
48.         e     = d - y
49.         delta = y*(1-y) * e
50.
51.         e1     = np.matmul(W2.T, delta)
52.         delta1 = y1*(1-y1) * e1
53.
54.         dW1 = (alpha*delta1).reshape(4, 1) * x.reshape(1, 3)
55.         W1  = W1 + dW1
56.
57.         dW2 = alpha * delta * y1
58.         W2  = W2 + dW2
59.
60.     return W1, W2
61.
62.
63. def test():
64.     X = np.array([[0, 0, 1],
65.                   [0, 1, 1],
66.                   [1, 0, 1],
67.                   [1, 1, 1]])
68.
69.     D = np.array([[0],
70.                   [1],
71.                   [1],
72.                   [0]])
73.
74.     E1 = np.zeros(1000)
75.     E2 = np.zeros(1000)
```

```
 76.
 77.     W11 = 2*np.random.random((4, 3)) - 1
 78.     W12 = 2*np.random.random((1, 4)) - 1
 79.     W21 = np.array(W11)
 80.     W22 = np.array(W12)
 81.
 82.     for _epoch in range(1000):
 83.         W11, W12 = BackpropCE(W11, W12, X, D)
 84.         W21, W22 = BackpropXOR(W21, W22, X, D)
 85.
 86.         es1 = 0
 87.         es2 = 0
 88.         N   = 4
 89.         for k in range(N):
 90.             x = X[k, :].T
 91.             d = D[k]
 92.
 93.             v1  = np.matmul(W11, x)
 94.             y1  = Sigmoid(v1)
 95.             v   = np.matmul(W12, y1)
 96.             y   = Sigmoid(v)
 97.             es1 = es1 + (d - y)**2
 98.
 99.             v1  = np.matmul(W21, x)
100.             y1  = Sigmoid(v1)
101.             v   = np.matmul(W22, y1)
102.             y   = Sigmoid(v)
103.             es2 = es2 + (d - y)**2
104.
105.         E1[_epoch] = es1 / N
106.         E2[_epoch] = es2 / N
107.
108.
109.     CE,  = plt.plot(E1, 'r')
110.     SSE, = plt.plot(E2, 'b:')
111.     plt.xlabel('Epoch')
112.     plt.ylabel('Average of Training error')
113.     plt.legend([CE, SSE], ["Cross Entropy", "Sum of Squared Error"])
114.     plt.show()
115.
116. test()
```

Результат роботи коду показаний на рис. 4.5.

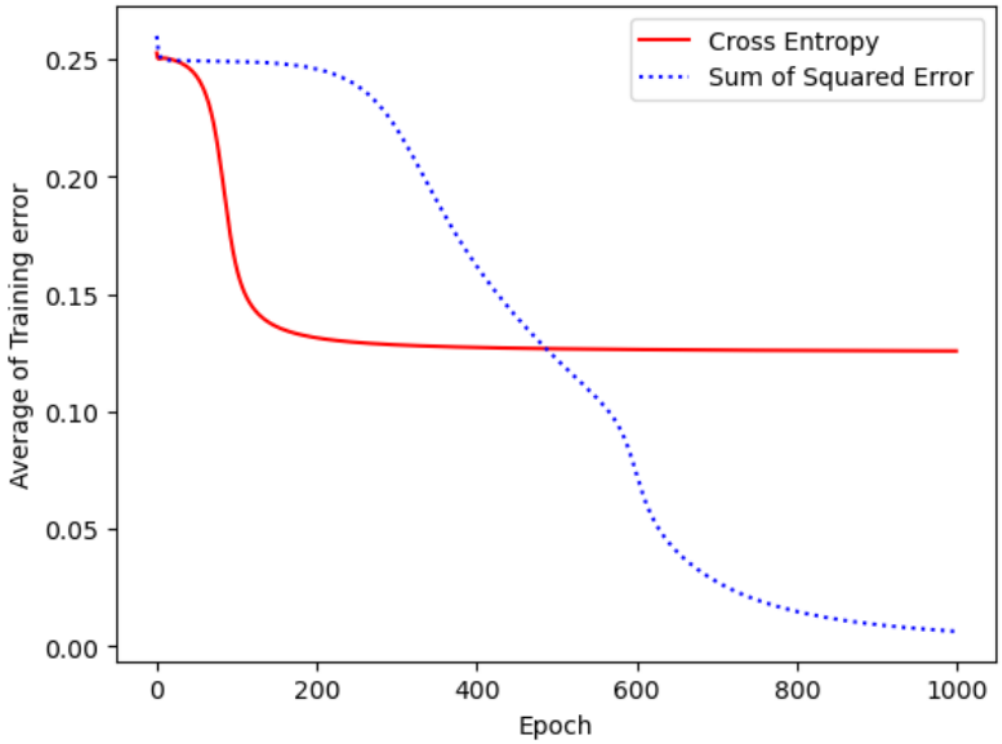


Рис. 4.5. Порівняння якості роботи мереж під час застосування різних функцій для обчислення похибок (1000 епох)

### *Частина 2.* Завдання для самостійного виконання

1. Здійсніть всі схеми навчання, які наведені у теоретичній частині. Також здійсніть порівняння ефективності.

2. Дослідить, як кількість епох навчання впливає на точність отриманих даних на виході нейронної мережі у випадку застосування звичайного зворотного розповсюдження. Зробіть висновок.

3. На що впливає параметр beta у лістингу 2 (тобто застосування моменту)? Порівняйте результат тренування мережі при застосуванні різних beta.

4. Яку кількість епох треба зробити під час застосування крос-ентропійної функції, щоб досягти такої точності навчання, яку ви досягаєте при застосуванні звичайної середньоквадратичної похибки на 500 епохах?

5. Зробіть висновок відносно результату порівняння різних функцій обчислення похибок (лістинг 4.5). В чому можливі нюанси, зробіть висновок.

6. (опціонально) Спробуйте сформулювати задачу, для розв’язання якої вам буде достатньо мережі, розглянутою в даній лабораторній роботі. Кількість нейронів у вхідному шарі може бути іншою. Розробіть датасет,

оберіть та застосуйте функцію обчислення похибки та алгоритм зворотного навчання (з або без «моменту»). Протестуйте роботу мережі та зробіть висновки.

7. Оформити звіт, в якому відобразити:

- ✓ Постановку завдання (завдання 2-5, 6 - опціонально).
- ✓ Короткі теоретичні відомості
- ✓ Коди програм
- ✓ Результати роботи програм
- ✓ Висновки для кожного завдання

*Запитання до лабораторної роботи*

1. Що таке шар нейронів?
2. Яка математична модель шару нейронів?
3. Що таке зворотне розповсюдження похибки?
4. В чому суть алгоритму зворотного розповсюдження?
5. Для чого застосовуються техніки оптимізації, як наприклад «момент»?
6. Що таке функція втрат, які вам відомі?
7. Як характеризується крос-ентропійна функція втрат?

# ЛАБОРАТОРНА РОБОТА №5

**Тема:** ***Застосування нейронних мереж у задачах класифікації***

**Мета:** Навчитися розробляти нейронні мережі для бінарної та багатокласової класифікації на мові python.

## *Частина 1.* Основні теоретичні відомості

Розглянемо дві задачі на класифікацію: бінарна класифікація та багатокласова класифікація.

### *Задача бінарної класифікації*

Бінарна класифікація за допомогою нейронних мереж може бути застосована для таких задача, як спім-фільтри (визначення, чи є лист спамом чи ні), автоматичне погодження кредиту (надати кредит або ні). Її головною особливістю є кількість класів, до яких можна віднести розглядаєму сутність - лист, заявку на кредит або щось інше. Таких класів два, тому і назва - бінарна.

Кількість вхідних ознак, які подаються на вхід мережі, залежить від задачі, і вони впливають на устрій вхідного шару. Для прикладу, оберемо дві ознаки, тобто вхідний шар матиме два входи. Вихідний шар для даного типу класифікації може містити як один так і два нейрони. Розглянемо ситуацію, коли він містить один вихідний нейрон. Кількість проміжних шарів може бути довільна, як і кількість нейронів у них. Для спрощення, оберемо один проміжний шар, який містить 4 нейрони. Отже, схема мережі наведена на рис. 5.1.

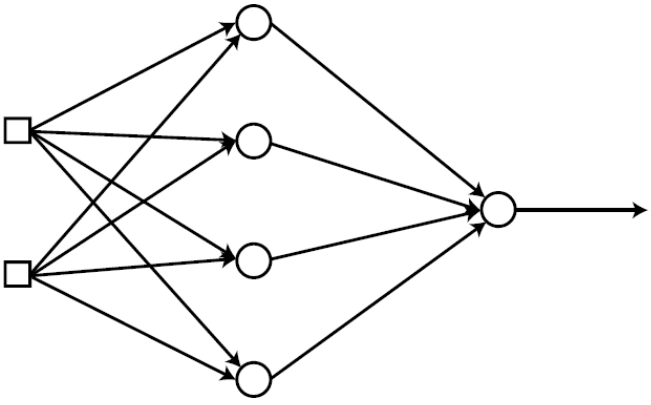

Рис. 5.1. Схема устрою НМ для бінарної класифікації

Для того, щоб закодувати у вихідному сигналі ознаку належності до одного з двох класів, користуються наступним принципом: якщо на виході є значення «0», то данні ознак описують сутність, яка належіть до класу 1; якщо на виході значення «1» – то сутність належить до класу 2.

Для того, щоб вихід мережі надавав потрібні значення від 0 до 1, застосуємо сигмовидну функцію активації, вигляд якої показаний на рис. 5.2.

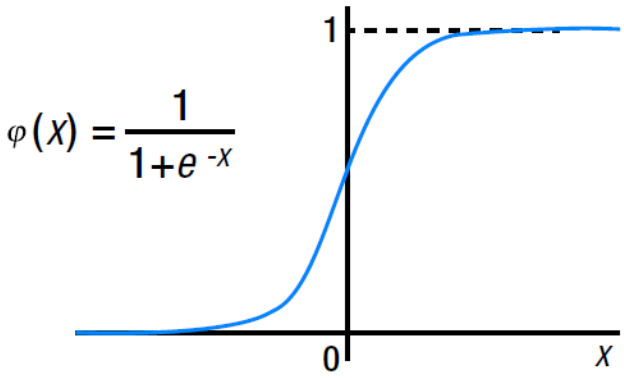


Рис. 5.2. Сигмовидна функція активації

Для обчислення функції втрат, застосуємо крос-ентропійну функцію втрат:

$$E = -d\ln(y) - (1-d)\ln(1-y) \qquad (5.1)$$

де $d$ – істине значення (з тренувального датасету), $y$ – поточне значення, яке дає мережа на виході.

Алгоритм роботи такої мережі та код реалізації на мові Python аналогічний до того, який був у попередній лабораторній роботі.

*Задача мультикласової класифікації*

Задача багатокласової (мультикласової) класифікації полягає у віднесені сутності, яка описується набором вхідних ознак, до одного з класів, яких може бути більше 2. Кількість класів, до яких може бути віднесена сутність, залежить від задачі.

Якщо, наприклад, ми маємо датасет, показаний на рис. 5.3, то в ньому кожний набір вхідних ознак (x1,x2) класифікується за одним з трьох класів (y).

| x1 | x2 | y |
|---|---|---|
| 5 | 7 | class 1 |
| 8 | 9 | class 3 |
| 2 | 3 | class 3 |
| 4 | 5 | class 2 |
| 1 | 5 | class 1 |
| 8 | 9 | class 2 |

Рис. 5.3. Датасет, який відносить данні до одного з трьох класів

На відміну від бінарної класифікації, де для кодування одного з двох класів достатньо двох цифр – 0 або 1, і відповідно, одного нейрону у вихідному шарі з сигмовидною функцією активації, для багатокласової класифікації це не підходить. В багатокласовій класифікації треба застосувати більше даних для кодування класів та, відповідно, більшу кількість нейронів у вихідному шарі.

Класи кодують за принципом прямого кодування «1 до N», де кожний клас кодується набором з N бінарних значень. Таким чином, «class 1» можна представити набором[1,0,0], «class 2» - [0,1,0], «class 3» - [0,0,1].
Відповідно до цього принципу, датасет матиме такий вигляд (рис.5.4), де кожна цифра з наведених вище наборів, представлена в додаткових колонках y1,y2,y3 [13].

| x1 | x2 | y1 | y2 | y3 | Origianl y |
|---|---|---|---|---|---|
| 5 | 7 | 1 | 0 | 0 | class 1 |
| 8 | 9 | 0 | 0 | 1 | class 3 |
| 2 | 3 | 0 | 0 | 1 | class 3 |
| 4 | 5 | 0 | 1 | 0 | class 2 |
| 1 | 5 | 1 | 0 | 0 | class 1 |
| 8 | 9 | 0 | 1 | 0 | class 2 |

Рис. 5.4. Оновлена структура датасету, з закодованими класами

Але для того, щоб на виході мережі отримувати комбінацію із трьох бінарних значень, потрібно 3 нейрони. Виходячи з цього, для розглядаємого прикладу схема вихідного шару нейронної мережі може бути така (рис. 5.5) (відсутність прихованого шару на схемі не принципова).

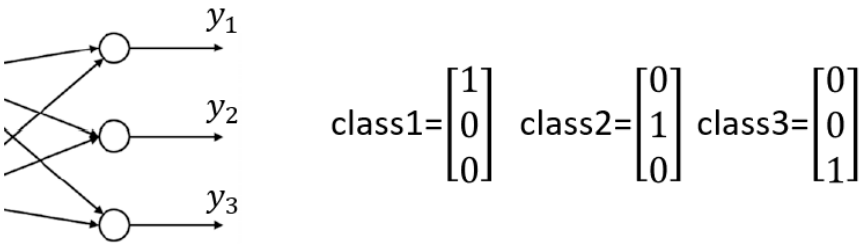


Рис. 5.5. Схема вихідного шару НМ для класифікації
до одного з трьох класів

Таким чином, отримавши на виході значення [y1=1, y2=0, y3=0], або просто [1,0,0], ми будемо знати, що мережа віднесла подані на вхід значення до «class 1» (див. рис.5.5, де вказані значення для зручності записані у виді вектора-стовпчика). І так для всіх інших вхідних значень.

Але для такого кодування у вихідному шарі, коли тільки один нейрон може мати максимальне значення 1, а всі інші мають давати 0, сигмовидна функція активації вже не підходить. Головна причина - вона не може забезпечити врахування впливів виходів окремих нейронів між собою [11,12]. Це може здійснити так звана soft-max (софт-макс) функція, яка для загального випадку N вихідних нейронів, може бути записана у вигляді:

$$y_i = \sigma(v_i) = \frac{e^{v_i}}{\sum_{j=1}^{N} e^{v_j}}, \tag{4.4}$$

де $v_i$ –- це зважена сума і-го вихідного нейрону, N – кількість вихідних нейронів.

Для випадку N=3 (у вихідному шарі 3 нейрони), ця функція матиме вигляд такий:

$$y_i = \sigma(v_i) = \frac{e^{v_i}}{e^{v_1}+e^{v_2}+e^{v_3}}, \tag{4.5}$$

де $i = 1,2,3$.

Як правило, загальна похибка обчислюється за формулою перехресної ентропії (4.1).

Виходячи з таких міркувань, розглянемо приклад багатокласової класифікації, призначений для розпізнавання цифр, представлених у виді бінарних зображень (рис. 5.6).

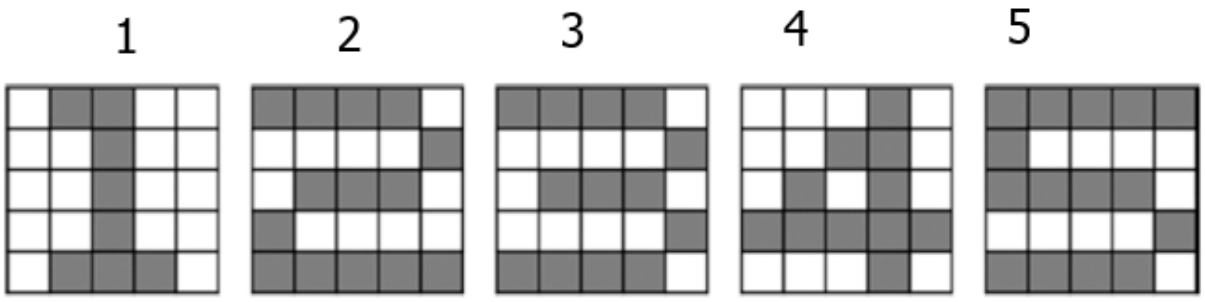


Рис. 5.6. Зображення цифр для розпізнавання

Кодування зображення для датасету відбуватиметься шляхом представлення зображення у виді матриці ознак 5x5=25 елементів, де в положеннях чорних пікселів буде стояти 1, а білих – 0. (рис. 5.7)

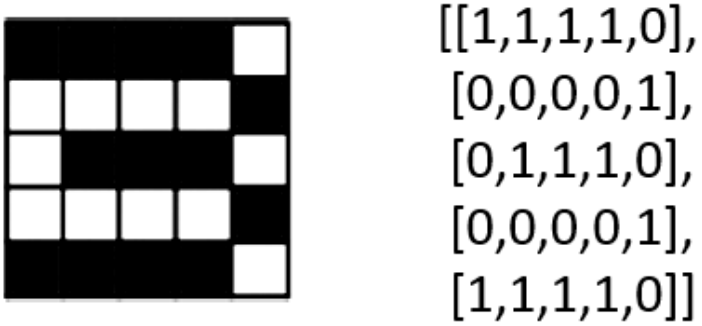


Рис. 5.7. Принцип кодування ознак цифри «3»

Відповідно до принципів, обговорених вище, схема мережі та кодування класів може бути таким (рис. 5.8) [13].

Вхідний шар містить 25 входів, по одному для кожного пікселя зображення. Проміжний шар містить 50 нейронів та сигмовидну функцію в якості активаційної. Вихідний шар містить 5 нейронів та софт-макс функцію активації.

Код навчання та тестування роботи мережі наведений у кількох лістингах нижче, але під час запуску він має бути в одному блокноті Jupyter.

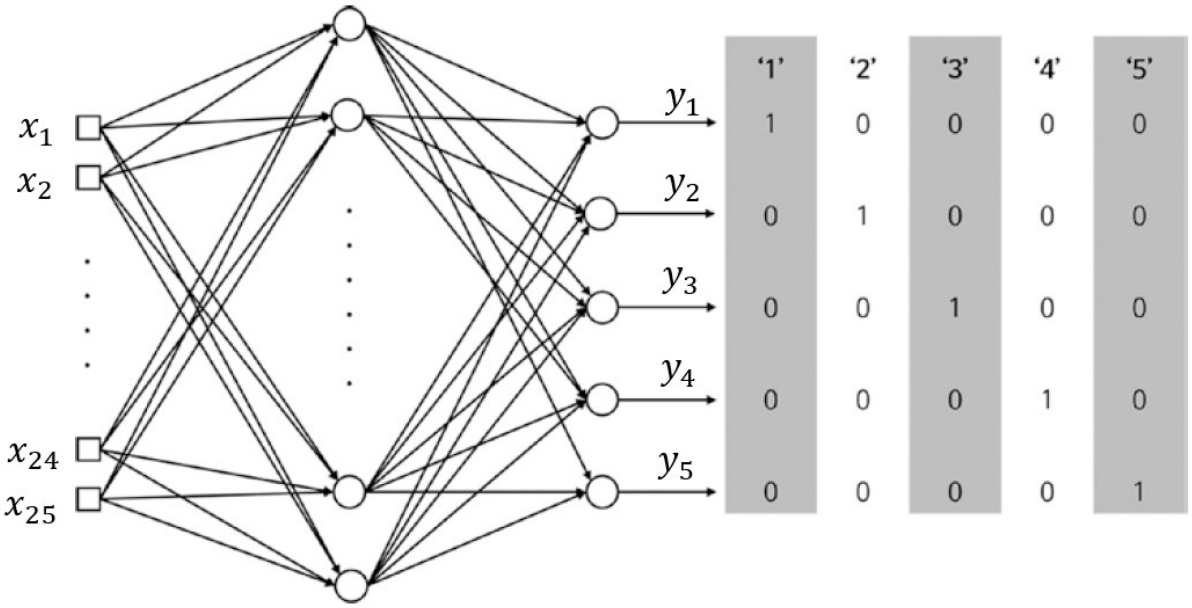


Рис. 5.8. Схема НМ для класифікації цифр від 1 до 5 (ліворуч)
та схема кодування класів (праворуч)

Наведений нижче перший фрагмент коду містить декларування бібліотеки NumPy та кількох функцій активації, які застосовуються в роботі різних шарів мережі [12].

```
 1. import numpy as np
 2.
 3. def Sigmoid(x):
 4.     return 1.0 / (1.0 + np.exp(-x))
 5.
 6. def Softmax(x):
 7.     x  = np.subtract(x, np.max(x))        # prevent overflow
 8.     ex = np.exp(x)
 9.
10.     return ex / np.sum(ex)
```

*Коментарі до фрагменту коду:*

р.3-4: декларування сигмовидної функції активації, аргументом є скалярне число;

р.6-10: декларування софт-макс функції активації, аргументом є вектор; Обчислення функції здійснюється за формулою (5.4). Але в реалізації під час програмування є нюанс: якщо значення вектора $v_i$ (в коді це вектор x) великі, то експонента $e^{x_i}$ також може бути дуже великою, що призведе до переповнення (overflow) і виклику помилок в обчисленнях. Наприклад, значення $e^{1000}$ є надто великим для комп'ютерних систем. В цьому випадку

застосовується використання np.subtract(x, np.max(x)), що дозволяє стабілізувати чисельні обчислення, зменшуючи ризик переповнення, при цьому не змінюючи результат softmax. Це працює так:

1. Здійснюється віднімання максимального значення: з кожного елемента вектора віднімається максимальне значення, це зменшує значення всіх елементів, зберігаючи їх відносні різниці. Наприклад, якщо x=[1000,1001,1002], то то після виконання xx=np.subtract(x, np.max(x)) , xx буде містити [-2,-1,0]

2. Обчислюються експоненти ex = np.exp(xx), тобто ex буде містити $[e^{-2}, e^{-1}, e^{-0}]$.

3. Здійснюється обчислення вектору вихідних значень як ex / np.sum(ex).

Це можно довести строго математично, тобто той факт, що:

якщо $x` = x - \max(x)$, то $softMax(x_i`) = \frac{e^{x_i}}{\sum_j e^{x_i}} = softMax(x_i)$.

Наступний фрагмент коду демонструє реалізацію функції MultiClass(W1, W2, X, D), призначеної для одного тренування мережі [12].

```
#
#
#
def MultiClass(W1, W2, X, D):
    alpha = 0.9

    N = 5
    for k in range(N):
        x = np.reshape(X[:,:,k], (25, 1))
        d = D[k, :].T

        v1 = np.matmul(W1, x)
        y1 = Sigmoid(v1)
        v  = np.matmul(W2, y1)
        y  = Softmax(v)

        e     = d - y
        delta = e

        e1     = np.matmul(W2.T, delta)
        delta1 = y1*(1-y1) * e1

        dW1 = alpha * delta1 * x.T
        W1  = W1 + dW1

        dW2 = alpha * delta * y1.T
        W2  = W2 + dW2
```

```
28.
29.     return W1, W2
```

*Коментарі до фрагменту коду:*

р.4: заголовок функції, в якості параметрів передаються: матриці вагових коефіцієнтів W1,W2 (для прихованого та вихідного шару), масив вхідних ознак X («зображення» для тренування, кожне «зображення» кодується у двовимірному масиві) та масив вірних відповідей (рис.5.8, таблиця праворуч);

р.7-8: організовується цикл по кількості зображень в тренувальному датасеті (N=5);

р.9: в x записується у вигляді розгорнутого у одновимірний масив, чергового k-того зображення з тренувального датасету X;

р.10: отримання вектору вірних відповідей виходу мережі для k-го зображення;

р.12: обчислення зваженої суми після першого (прихованого) шару, як добуток матриці вагових коефіцієнтів W1 на вектор вхідних ознак x;

р.13: обчислення вихідного сигналу y1 після першого (прихованого) шару під дією сигмовидної функції активації;

р.14: обчислення зваженої суми нейронів вихідного шару як добуток матриці вагових коефіцієнтів W2 на вектор вхідних ознак y1;

р.15: обчислення вихідного сигналу y на виході мережі під дією софт-макс функції активації;

р.17-18: обчислення похибки між отриманим сигналом y та дійсним значенням з тренувального датасету d та обчислення delta;

р.20-21: реалізація алгоритму оберненого розповсюдження і обчислення e1 та delta1 для прихованого шару;

р.23-24: обчислення поправки dW1 для матриці вагів W1 та корегування матриці W1;

р.26-27: обчислення поправки dW2 для матриці вагів W2 та корегування матриці W2;

р.29: повернення оновлених матриць W1 та W2.

Слід звернути увагу, що функція MultiClass здійснює однократне тренування шляхом одного проходження по всіх сємплах (точках) датасету.

Наступний фрагмент коду визначає функцію для тренування мережі на зазначених на рис.5.6 зображеннях [12].

```
1. # Training net and validating on digits from training dataset
```

```
def TestMultiClass():
    X = np.zeros((5, 5, 5))

    X[:, :, 0] = [[0,1,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,0,1,0,0],
                  [0,1,1,1,0]]

    X[:, :, 1] = [[1,1,1,1,0],
                  [0,0,0,0,1],
                  [0,1,1,1,0],
                  [1,0,0,0,0],
                  [1,1,1,1,1]]

    X[:, :, 2] = [[1,1,1,1,0],
                  [0,0,0,0,1],
                  [0,1,1,1,0],
                  [0,0,0,0,1],
                  [1,1,1,1,0]]

    X[:, :, 3] = [[0,0,0,1,0],
                  [0,0,1,1,0],
                  [0,1,0,1,0],
                  [1,1,1,1,1],
                  [0,0,0,1,0]]

    X[:, :, 4] = [[1,1,1,1,1],
                  [1,0,0,0,0],
                  [1,1,1,1,0],
                  [0,0,0,0,1],
                  [1,1,1,1,0]]

    D = np.array([[[1,0,0,0,0]],
                  [[0,1,0,0,0]],
                  [[0,0,1,0,0]],
                  [[0,0,0,1,0]],
                  [[0,0,0,0,1]]])


    W1 = 2*np.random.random((50, 25)) - 1
    W2 = 2*np.random.random(( 5, 50)) - 1

    # Training on 10_000 epoches
    for _epoch in range(10000):
        W1, W2 = MultiClass(W1, W2, X, D)


    # Validating on digits from training dataset
    N = 5
    for k in range(N):
        x  = np.reshape(X[:, :, k], (25, 1))
        v1 = np.matmul(W1, x)
```

```
55.         y1 = Sigmoid(v1)
56.         v  = np.matmul(W2, y1)
57.         y  = Softmax(v)
58.
59.         print("--------------------")
60.         print("Y = {}: ".format(k+1))
61.         displayDigit(X[:, :, k])
62.         print(y)
63.
64.     return W1, W2
```

*Коментарі до фрагменту коду:*

р.3: створення та ініціалізація нулями тривимірного масиву X, який міститиме 5 двовимірних масивів розміром 5x5;
р.5-9: додавання в масив X зображення цифри «1»;
р.11-15: додавання в масив X зображення цифри «2»;
р.17-21: додавання в масив X зображення цифри «3»;
р.23-27: додавання в масив X зображення цифри «4»;
р.29-33: додавання в масив X зображення цифри «5»;
р.35: створення масиву списків, кожен з списків містить істинні значення виходів мережі (у виді векторів-рядків) для кожного класу зображень (схема кодування класів наведена на рис.5.8);
р.42-43: створення матриць вагів W1,W2 та ініціалізація випадковими значеннями від -1 до 1;
р.45-47: ітеративне тренування мережі на протязі 10000 епох;
р.50-62: валідація роботи мережі шляхом подачі на її входи зображень з тренувального датасету, данні з виходів мережі виводяться в консоль;
р.64: повернення моделі натренованої мережі (матриці W1,W2);
р.61: застосування функції displayDigit, яка виводить в консоль вміст двовимірного масиву, який містить «зображення» цифри.

Наступний фрагмент містить функцію displayDigit, яка виводить в консоль «зображення» цифри в більш зручному виді, аніж комбінацією 0 та 1 [12].

```
1. def displayDigit(digit):
2.     symbol='*'
3.     #print(digit[0,2])
4.     for i in range(5):
5.         for j in range(5):
6.             if digit[i,j]>0.5:
7.                 print(symbol+' ',end='')
8.             else:
9.                 print('  ',end='')
```

```
10.         print()
```

*Коментарі до фрагменту коду:*

р.1: подача на вхід функції двовимірного масиву digit, який містить зображення цифри;

р2: визначення, яким знаком буде ображатись в консолі чорний піксель;

р.4-10: вивід зображення в консоль, де символом symbol буде зображатись чорний піксель, а символом ‘ ‘ буде виводитись білий піксель.

Останній великий фрагмент коду містить функцію RealMultiClass(), яка реалізує не тільки тренування мережі, а і перевірку її роботи шляхом подачі зображень цифр, які не були присутні в тренувальному датасеті [12].

```
 1. #
 2. def RealMultiClass():
 3.     # Train network
 4.     W1, W2 = TestMultiClass()
 5.
 6.     # Creating digits for classification
 7.     X = np.zeros((5, 5, 5))
 8.
 9.     X[:, :, 0] = [[0,0,1,1,1],
10.                   [0,1,0,1,0],
11.                   [1,1,1,1,0],
12.                   [0,0,0,1,0],
13.                   [0,0,1,1,1]]
14.
15.     X[:, :, 1] = [[0,1,1,0,0],
16.                   [0,0,1,0,0],
17.                   [0,0,1,0,0],
18.                   [0,0,1,0,0],
19.                   [0,1,1,1,0]]
20.
21.
22.     N = 2
23.     for k in range(N):
24.         x  = np.reshape(X[:, :, k], (25, 1))
25.         v1 = np.matmul(W1, x)
26.         y1 = Sigmoid(v1)
27.         v  = np.matmul(W2, y1)
28.         y  = Softmax(v)
29.
30.         print("------------------")
31.         print("N = {}: ".format(k+1))
32.         displayDigit(X[:, :, k])
33.         print(y)
```

*Коментарі по коду (основні)*

функція на початку роботи здійснює тренування мережі (р.4), далі створюється масив X, який міститиме кілька зображень цифр для класифікації. Цифри задаються:  спотворена «4» в р.9-13 та спотвореня «1» в р.15-19. В циклі (р.23-33) здійснюється почергова подача на вхід натренованої мережі цих цифр та вивід результатів класифікації в консоль.

Для запуску програми, слід викликати або функцію TestMultiClass() (для простого тренування мережі) або функцію RealMultiClass() для тренування та класифікації інших цифр.

```
1. if __name__ == '__main__':
2.     #TestMultiClass()
3.     RealMultiClass()
```

Якщо викликати останню, як показано в фрагменті коду вище, то в консолі буде вивід як на рис.5.9. Це результат тренування мережі на 10000 епохах (зображення цифр та результати на виході мережі під підписами Y=1, Y=2, Y=3, Y=4, Y=5) та результат класифікації двох зображень цифр, які не були присутні в датасеті (під підписами N=1,N=2, зображення та результат класифікації мережі). Цифрами на зеленому фоні позначені: 1-5 – зображення цифр та результати тренування мережі; 6-7 – зображення цифр та результати їх класифікації.

Точність тренування при тренуванні на 10000 епохах достатньо велика. Наприклад, в виводі (1) показано, що класифікація цифри «1» як саме одиничка, відбувається з імовірністю 9.99988054e-01. Аналогічна точність досягається і для інших цифр тренувального датасету.

Якщо розглянути класифікацію останніх двох цифр, яких немає в навчальному датасеті, наприклад цифру з позначкою «6» на рис. 5.9, то імовірність розпізнавання в зображенні цифри «4» досить низька: 0.07. Імовірність того, що це цифра «1» складає 0.0055, цифра «2» - 0.3, цифра «3» - 0.000075, цифра «4» - 0.07, цифра «5» - 0.62 . Тобто, імовірність того, що на зображенні цифра «4» (як воно і є) – 7 %. Мережа вважає з імовірністю 62 %, що на зображенні показана цифра «5», і з імовірність того що це цифра «2» - 30 %.

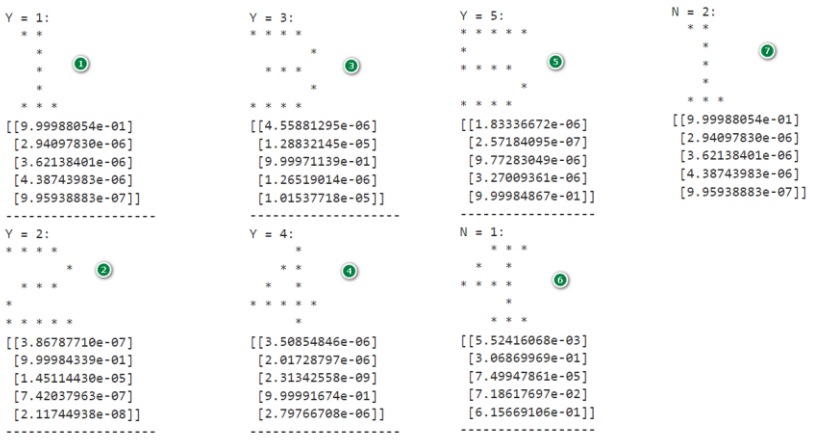


Рис. 5.9. Вивід в консоль результатів тренування мережі

## *Частина 2.* Завдання для самостійного виконання

1. З наведених в теоретичних відомостях фрагментів коду, скомпонуйте програму, яке буде моделювати навчання нейронної мережі класифікувати зображення цифр.

2. Здійсніть тренування мережі та перевірте якість розпізнавання на зображеннях тренувального датасету

3. Додайте в функцію RealMultiClass зображення цифр 1-5, які не були включені в тренувальний датасет. Оцініть ефективність їх розпізнавання.

4. Проведіть наступне дослідження: для однієї цифри створіть кілька варіацій її зображення, із різними особливостями в структурі. Додайте їх до навчального датасету функції TestMultiClass. Здійсніть навчання мережі на цьому датасеті. Здійсніть перевірку роботи мережі щодо якості класифікації цифр, для яких ви додали варіанти написання. Зробіть висновки.

5. Оформити звіт, в якому відобразити:

- ✓ Постановку завдання
- ✓ Коди програм
- ✓ Результати роботи програм
- ✓ Висновки для кожного завдання

*Запитання до лабораторної роботи*

1. Що таке класифікація, які види існують

2. Чому для бінарної класифікації достатньо одного нейрону у вихідному шарі?

3. Чим відрізняється софт-макс функція активації від сигмовидної? В яких випадках слід застосовувати ту чи іншу функції?

4. Яка причина застосування саме софт-макс функції активації у вихідному шарі під час багатокласової класифікації

## СПИСОК ЛІТЕРАТУРИ


1. Y. Daniel Liang  Introduction to programming using Python // Pearson, 2013.
2. Python Numpy Tutorial (with Jupyter and Colab). – Режим доступу: https://cs231n.github.io/python-numpy-tutorial/
3. aCode. Уроки Python. Режим доступу: https://acode.com.ua/lessons-python/
4. OneCompiler. Режим доступу: https://onecompiler.com/python
5. NumPy. Array creation. – Режим доступу: https://numpy.org/doc/stable/user/basics.creation.html#arrays-creation
6. NumPy. Mathematical functions. – Режим доступу: https://numpy.org/doc/stable/reference/routines.math.html
7. NumPy. Array manipulation routines. – Режим доступу: https://numpy.org/doc/stable/reference/routines.array-manipulation.html
8. SciPy API. Режим доступу: https://docs.scipy.org/doc/scipy/reference/
9. Matplotlib. Pyplot API. – Режим доступу: https://matplotlib.org/2.0.2/api/pyplot_api.html
10. Andrew W. Trask grokking Deep Learning // Manning Publications, 2019.
11. Phil Kim MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence // Apress, 2017.
12. Apress Source Code. Режим доступу: https://github.com/Apress/matlab-deep-learning
13. Електронний курс дисципліни на освітньому сайті КНУБА. – Режим доступу: http://org2.knuba.edu.ua/

Навчально-методичне видання

# ШТУЧНИЙ ІНТЕЛЕКТ ТА НЕЙРОННІ МЕРЕЖІ

Методичні вказівки
до виконання лабораторних робіт
для здобувачів другого (магістерського) рівня вищої освіти
спеціальності 126 «Інформаційні системи та технології»,
освітньо-професійних програм «Штучний інтелект. Когнітивні технології»,
«Інформаційні системи та технології»
всіх форм навчання

*У двох частинах*
***Частина перша***

Укладачі: **Ільїн** Олег Олександрович,
**Бушуєв** Сергій Дмитрович

Комп'ютерне верстання *А. П. Селівестрової*

Ум. друк. арк. 4,42. Обл.-вид. арк. 4,75
Електронний документ. Вид № 2/V-25.

Виконавець і виготовлювач
Київський національний університет будівництва і архітектури

Проспект Повітряних Сил, 31, Київ, Україна, 03680

Свідоцтво про внесення до Державного реєстру суб'єктів
видавничої справи ДК № 808 від 13.02.2002 р