Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Кафедра автоматизації проектування енергетичних процесів і систем

Лабораторна робота №1

з дисципліни «МЕТОДОЛОГІЯ РОЗРОБКИ ІНТЕЛЕКТУАЛЬНИХ КОМП’ЮТЕРНИХ ПРОГРАМ»

«**Штучні нейронні мережі. Моделювання формальних логічних функцій. Прогнозування часових рядів**»

Виконала: студентка 3 курсу

ІАТЕ групи ТІ-01

Круть Катерина

Перевірив: проф. Мусієнко А. П.

**КИЇВ 2023**

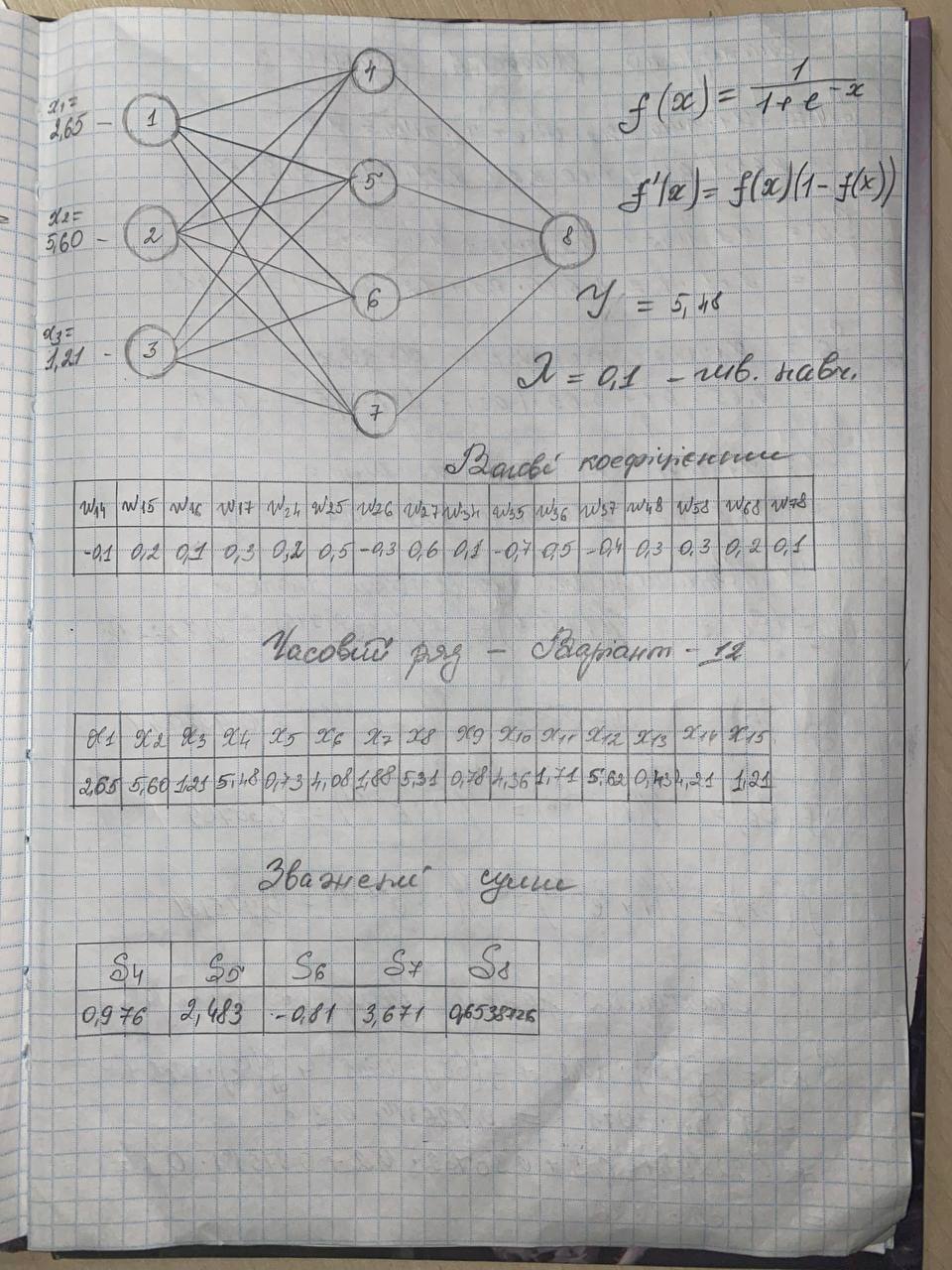
**ПОСТАНОВКА МЕТИ ДОСЛІДЖЕННЯ**

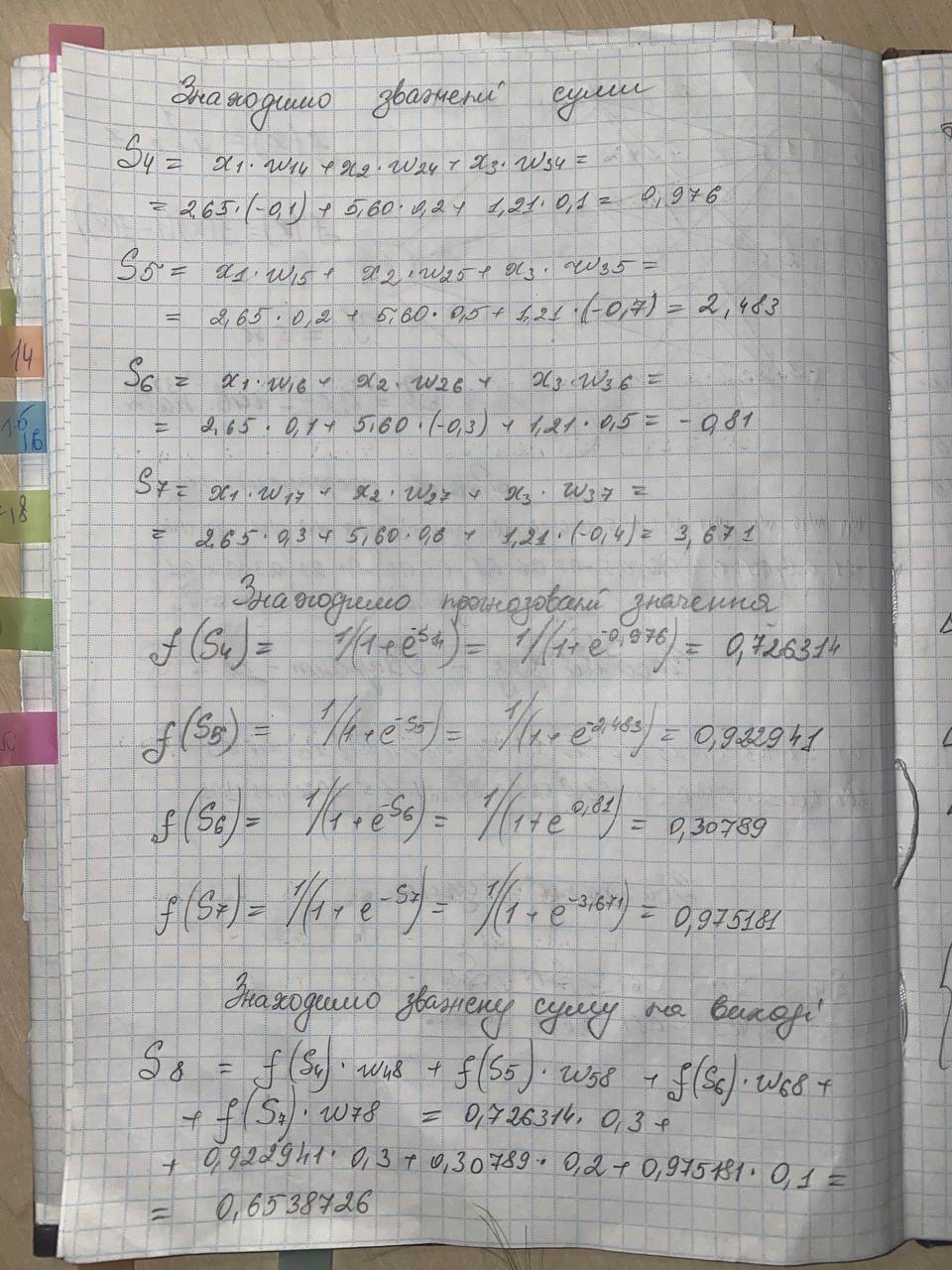
**Мета дослідження:** отримати початкові навички щодо створення штучних нейронних мереж, що здатні виконувати прості логічні функції, та нейронних мереж, що здатні прогнозувати часові ряди.

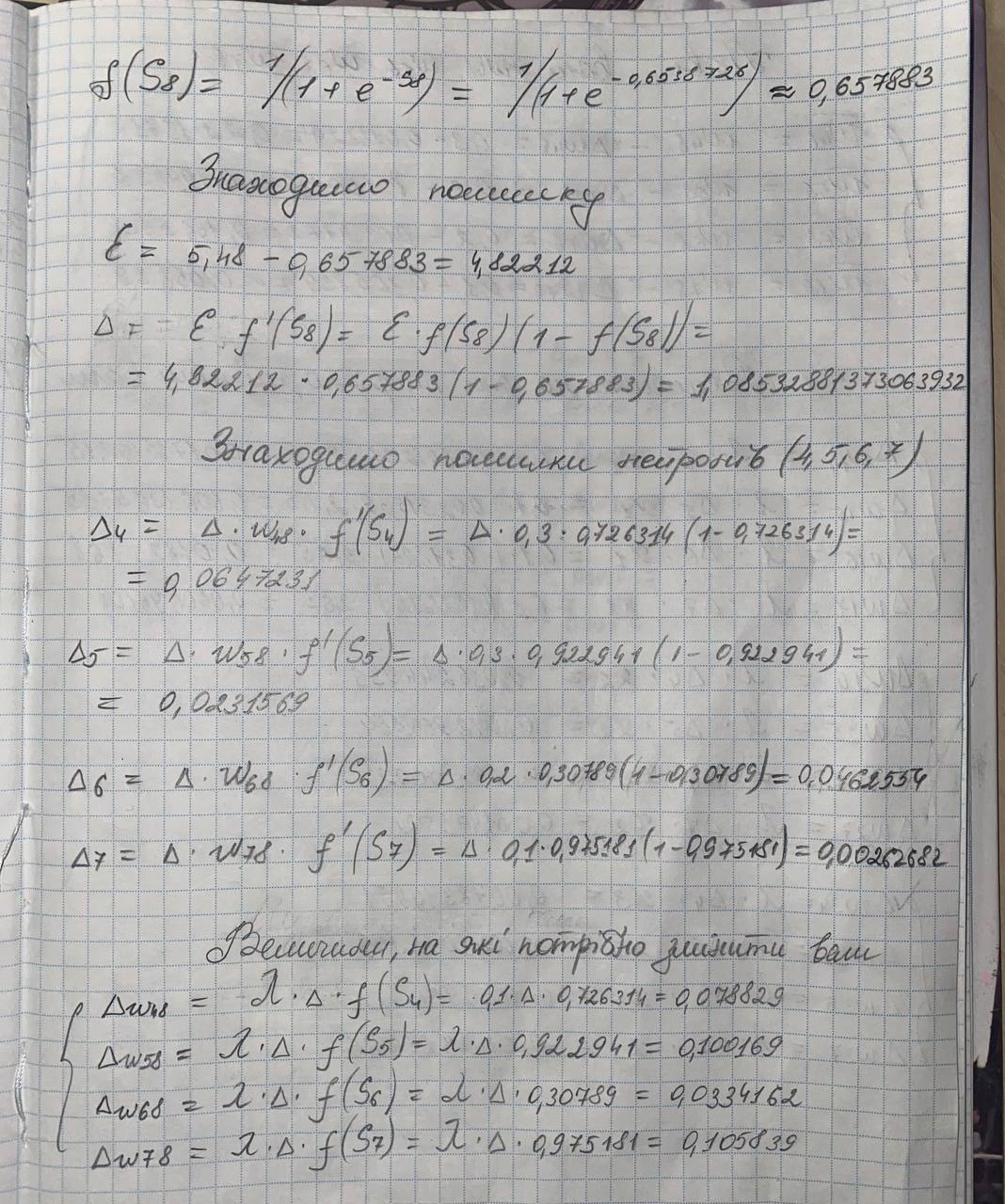
**Набір данних:**Изображение

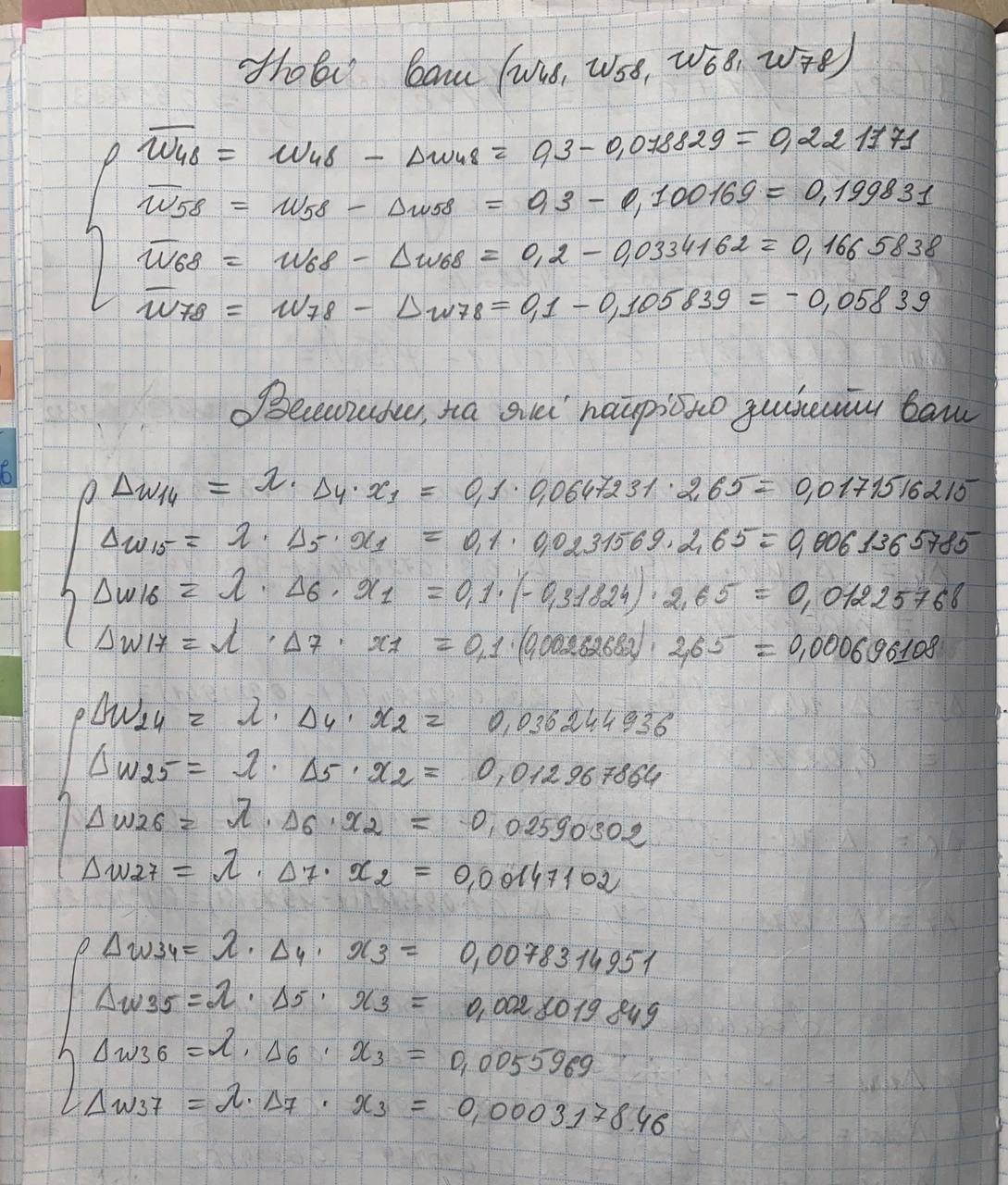
**ОПИС ДІЙ**

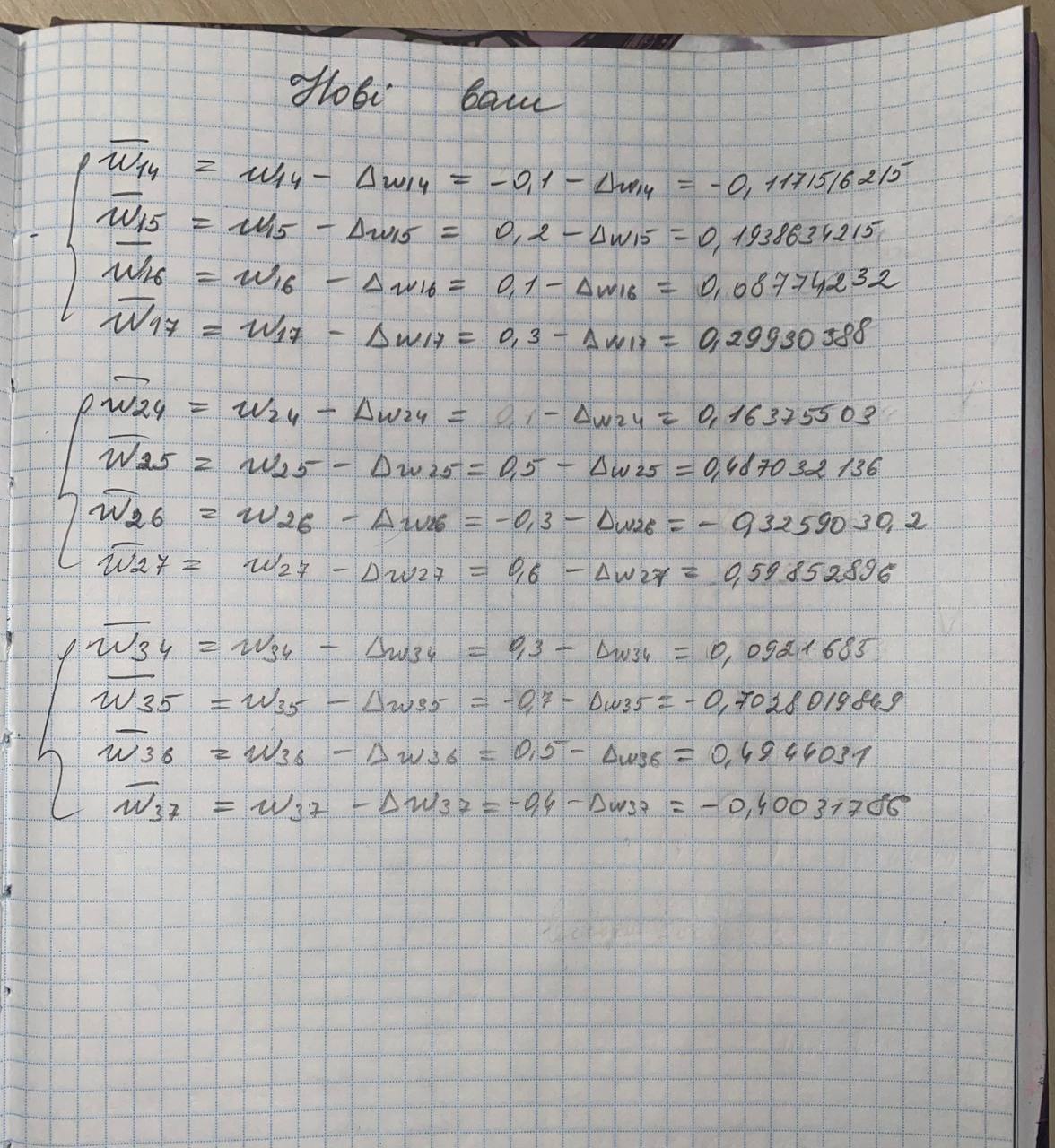
Алгоритм зворотного поширення на прикладі першої епохи

****

****

****

****

****

**КОПІЯ ВИКОНАНОЇ ПРОГРАМИ**

**Моделювання логічних функцій AND, NOT, OR**

import numpy as np

class Neuron:

def \_\_init\_\_(self, weights, name='Neuron'):

self.weights = weights

self.name = name

def get\_weights(self):

return self.weights

def set\_weights(self, w):

if isinstance(w, list):

self.weights = w

if w is None:

self.weights = [1, 1]

def get\_S(self, inputs):

return np.dot(np.array(self.weights), inputs)

def act\_F(self, weighted):

return weighted

def forward(self, inputs):

return self.act\_F(self.get\_S(inputs))

def \_\_str\_\_(self):

return f'{self.name}\nweights: {self.weights}\n'

class NeuronAND(Neuron):

def \_\_init\_\_(self, weights=None, name='Neuron AND'):

if isinstance(weights, list):

self.weights = weights

if weights is None:

self.weights = [1, 1]

super().\_\_init\_\_(weights)

self.weights = weights

self.name = name

def act\_F(self, weighted):

return 1 if weighted >= 1.5 else 0

def forward(self, inputs): # print(self.weights, inputs) print(np.dot(self.weights, inputs))

return self.act\_F(self.get\_S(inputs))

class NeuronNOT(Neuron):

def \_\_init\_\_(self, weight=None, name='Neuron NOT'):

if isinstance(weight, float):

self.weights = weight

if weight is None:

self.weights = -1.5

super().\_\_init\_\_(weight)

self.weights = weight

self.name = name

def set\_weights(self, w):

if isinstance(w, float):

self.weights = w

if w is None:

self.weights = 1

def get\_S(self, inputs):

return self.weights \* inputs

def act\_F(self, weighted):

return 1 if weighted >= -1 else 0

def forward(self, inputs):

return self.act\_F(self.get\_S(inputs))

class NeuronOR(Neuron):

def \_\_init\_\_(self, weight=None, name='Neuron OR'):

if isinstance(weight, list):

self.weights = weight

if weight is None:

self.weights = [1, 1]

super().\_\_init\_\_(weight)

self.weights = weight

self.name = name

def act\_F(self, weighted):

return 1 if weighted >= 0.5 else 0

def forward(self, inputs):

return self.act\_F(self.get\_S(inputs))

class NeuralNetworkLO:

def \_\_init\_\_(self, neuron, data):

self.neuron = neuron

self.data = data

self.result = []

def forward(self, inputs):

return self.neuron.forward(inputs)

def check\_data(self):

return [(i, self.forward(i)) for i in self.data]

def \_\_str\_\_(self):

if not self.result:

self.result = self.check\_data()

return f'{str(self.neuron)}\n' + '\n'.join([f'{i[0]} => {i[1]}' for i in self.result]) + '\n---------------\n'

data\_AND = [

(0, 0), # Expected output: 0

(0, 1), # Expected output: 0

(1, 0), # Expected output: 0

(1, 1) # Expected output: 1

]

data\_NOT = [

0, # Expected output: 1

1, # Expected output: 0

]

data\_OR = [

(0, 0), # Expected output: 0

(0, 1), # Expected output: 1

(1, 0), # Expected output: 1

(1, 1) # Expected output: 1

]

print(NeuralNetworkLO(NeuronAND([1, 1]), data\_AND))

print(NeuralNetworkLO(NeuronNOT(-1.5), data\_NOT))

print(NeuralNetworkLO(NeuronOR([1, 1]), data\_OR))

**Моделювання логічної функції XOR**

import numpy as np

def get\_connections\_n(neurones, input\_neurones):

return input\_neurones \* (neurones - input\_neurones) + input\_neurones \* input\_neurones

class Neuron:

def \_\_init\_\_(self, weights, name='Neuron'):

self.weights = weights

self.name = name

def get\_weights(self):

return self.weights

def set\_weights(self, w):

if isinstance(w, list):

self.weights = w

if w is None:

self.weights = [1, 1]

def get\_S(self, inputs):

return np.dot(np.array(self.weights), inputs)

def act\_F(self, weighted):

return weighted

def forward(self, inputs):

return self.act\_F(self.get\_S(inputs))

def \_\_str\_\_(self):

return f'{self.name}\nweights: {self.weights}\n'

class NeuronXOR(Neuron):

def \_\_init\_\_(self, weight=None, name='Neuron XOR'):

if isinstance(weight, list):

self.weights = weight

if weight is None:

self.weights = [1, 1]

super().\_\_init\_\_(weight)

self.weights = weight

self.name = name

def act\_F(self, weighted):

return 1 if weighted >= 0.5 else 0

def forward(self, inputs):

return self.act\_F(self.get\_S(inputs))

class NeuralNetworkXOR:

def \_\_init\_\_(self, data, weights=None, neurones\_n=3, input\_nn=2):

self.neurones\_n = neurones\_n

self.input\_nn = input\_nn

if isinstance(weights, list) and len(weights) == get\_connections\_n(self.neurones\_n, self.input\_nn):

self.weights = weights

if weights is None:

self.weights = [[1, -1], [-1, 1], [1, 1]]

self.neurones = self.initialize()

self.data = data

self.result = []

def initialize(self): # np.array\_split(np.array([1, -1, -1, 1, 1, 1]), self.neurones\_n)

return [NeuronXOR(i) for i in self.weights]

def forward(self, inputs):

return self.neurones[self.input\_nn:][0].forward([i.forward(inputs) for i in self.neurones[:self.input\_nn]])

def check\_data(self):

return [(i, self.forward(i)) for i in self.data]

def \_\_str\_\_(self):

if not self.result:

self.result = self.check\_data()

return ''.join([str(n) for n in self.neurones]) + '\n' + '\n'.join(

[f'{i[0]} => {i[1]}' for i in self.result]) + '\n---------------\n'

data\_XOR = [

(0, 0), # Expected output: 0

(0, 1), # Expected output: 1

(1, 0), # Expected output: 1

(1, 1) # Expected output: 1

]

print(NeuralNetworkXOR(data\_XOR))

**Моделювання для часового ряду**

import numpy as np

"""

hidden = neurones

inp = input neurones

out = output neurones

"""

def get\_connections\_n(hidden, inp, out):

return inp \* hidden + hidden \* out

def f(x):

return 1 / (1 + np.exp(-x))

def df(x):

return x \* (1 - x)

class Neuron:

def \_\_init\_\_(self, weights\_, name='Neuron'): # inputs\_n

self.weights = weights\_

self.name = name

self.delta = None

self.neuron\_fS = None

def get\_weights(self):

return self.weights

def set\_weights(self, w):

if isinstance(w, list):

self.weights = w

if w is None:

self.weights = [1, 1, 1]

def set\_delta(self, delta):

self.delta = delta

def get\_S(self, inputs):

return np.dot(np.array(self.weights), inputs)

def f(self, x):

return 1 / (1 + np.exp(-x))

def df(self, x):

return x \* (1 - x) # return 1 / (1 + np.exp(-x)) \* (1 - 1 / (1 + np.exp(-x)))

def set\_neuron\_fS(self, fs):

self.neuron\_fS = fs

def forward(self, inputs):

return self.f(self.get\_S(inputs))

def \_\_str\_\_(self):

return f'{self.name}\nweights: {self.weights}\nf(S): {self.neuron\_fS}\ndelta: {self.delta}'

class NeuralNetwork:

def \_\_init\_\_(self, weights\_=None, iterations=1000, lmd=0.1, input\_size=3, hidden\_size=4, output\_size=1):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.iterations = iterations

self.lmd = lmd

self.conn\_n = get\_connections\_n(self.hidden\_size, self.input\_size, self.output\_size)

self.weights = weights\_ if isinstance(weights\_, list) and len(weights\_) == self.conn\_n else np.random.randn(

self.conn\_n)

self.hidden\_neurones = []

self.output\_neuron = None

def set\_iterations(self, iterations):

self.iterations = iterations

def set\_lmd(self, lmd):

self.lmd = lmd

def set\_weights(self, w):

if isinstance(w, list) and len(w) == self.conn\_n:

self.weights = w

def get\_hidden\_weights(self):

return np.array\_split(self.weights[:len(self.weights) - self.hidden\_size], self.hidden\_size)

def get\_output\_weights(self):

return self.weights[self.conn\_n - self.hidden\_size:]

def initialize\_hidden(self):

weights\_ = self.get\_hidden\_weights()

self.hidden\_neurones = [Neuron(weights\_[i], f'Neuron {self.input\_size + i + 1}') for i in range(len(weights\_))]

def initialize\_output(self):

weights\_ = self.get\_output\_weights()

if self.output\_size == 1:

self.output\_neuron = Neuron(weights\_, f'Neuron {self.input\_size + self.output\_size + self.hidden\_size}')

# else:

# weights\_ = np.array\_split(weights\_, self.output\_size)

# n = self.input\_size + self.output\_size + self.hidden\_size

# self.output\_neuron = [Neuron(weights\_[i], f'Neuron {n + 1}') for i in range(len(weights\_))]

def get\_hidden\_fS(self, inputs):

for neuron in self.hidden\_neurones:

neuron.set\_neuron\_fS(neuron.forward(inputs))

return [neuron.neuron\_fS for neuron in self.hidden\_neurones]

def get\_out\_fS(self, hidden\_fs):

self.output\_neuron.set\_neuron\_fS(self.output\_neuron.forward(hidden\_fs))

return self.output\_neuron.neuron\_fS

def set\_out\_delta(self, expected):

error = expected - self.output\_neuron.neuron\_fS

delta = error \* df(self.output\_neuron.neuron\_fS)

self.output\_neuron.set\_delta(delta)

def set\_hidden\_delta(self):

for i in range(len(self.hidden\_neurones)):

neuron = self.hidden\_neurones[i]

neuron.set\_delta(self.output\_neuron.delta \* self.output\_neuron.weights[i] \* df(neuron.neuron\_fS))

def adjust\_weights\_output(self, hidden\_fs):

return np.array(self.output\_neuron.weights) + self.output\_neuron.delta \* self.lmd \* np.array(hidden\_fs)

def adjust\_weights(self, hidden\_fs, inputs):

hidden\_weights = np.array(

[np.array(i.weights) + i.delta \* self.lmd \* np.array(inputs) for i in self.hidden\_neurones]).flatten()

self.weights = np.concatenate([hidden\_weights, self.adjust\_weights\_output(hidden\_fs)])

def check(self, inputs):

self.initialize\_hidden()

self.initialize\_output()

return self.get\_out\_fS(self.get\_hidden\_fS(np.array(inputs) / 10)) \* 10

def test(self, data\_):

return (data\_[1], self.check(data\_[0]))

def train(self, data\_):

inputs = np.array(data\_[0]) / 10

expected = data\_[1] / 10

for iteration in range(self.iterations):

self.initialize\_hidden()

self.initialize\_output()

hidden\_fs = self.get\_hidden\_fS(inputs)

out\_fS = self.get\_out\_fS(hidden\_fs)

error = expected - out\_fS

delta = error \* df(out\_fS)

self.output\_neuron.set\_delta(delta)

self.set\_hidden\_delta()

self.adjust\_weights(hidden\_fs, inputs)

data = [2.65, 5.60, 1.21, 5.48, 0.73, 4.08, 1.88, 5.31, 0.78, 4.36, 1.71, 5.62, 0.43, 4.21, 1.21]

# [w14, w24, w34, w15 .. w68, w78]

weights = [-0.1, 0.2, 0.1, 0.2, 0.5, -0.7, 0.1, -0.3, 0.5, 0.3, 0.6, -0.4, 0.3, 0.3, 0.2, 0.1]

nn = NeuralNetwork(weights)

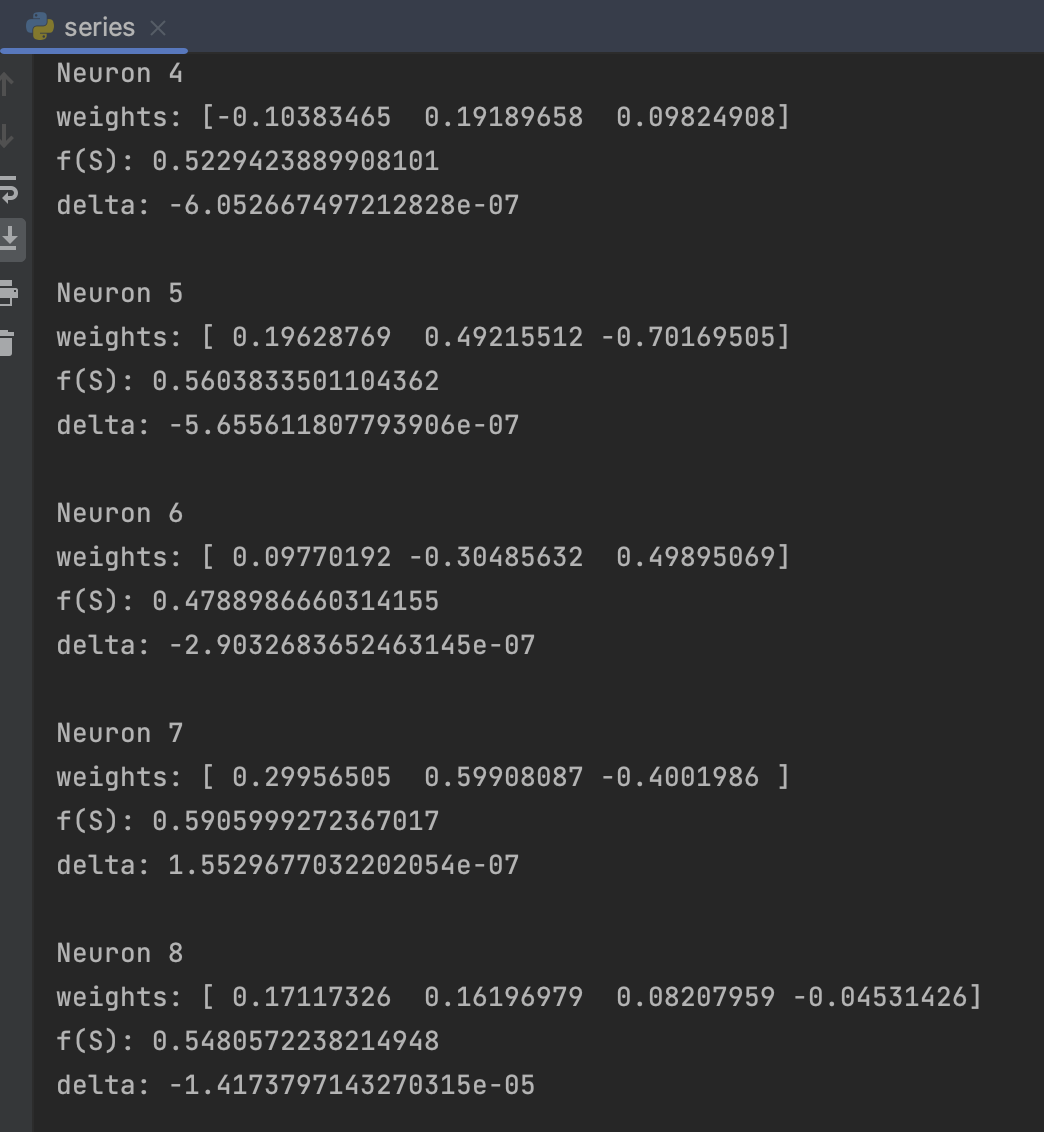
nn.train(([2.65, 5.60, 1.21], 5.48))

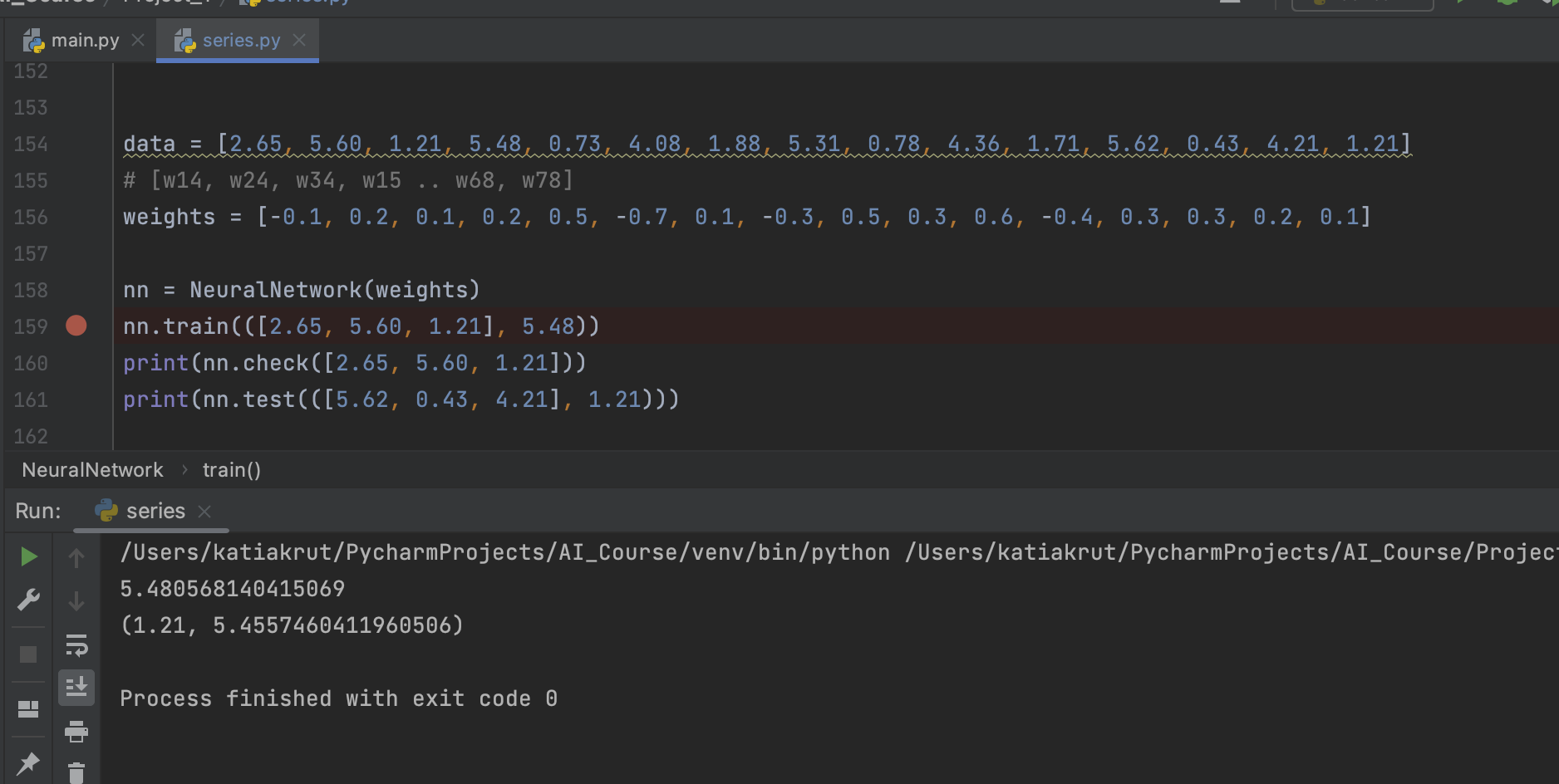
print(nn.check([2.65, 5.60, 1.21]))

print(nn.test(([5.62, 0.43, 4.21], 1.21)))

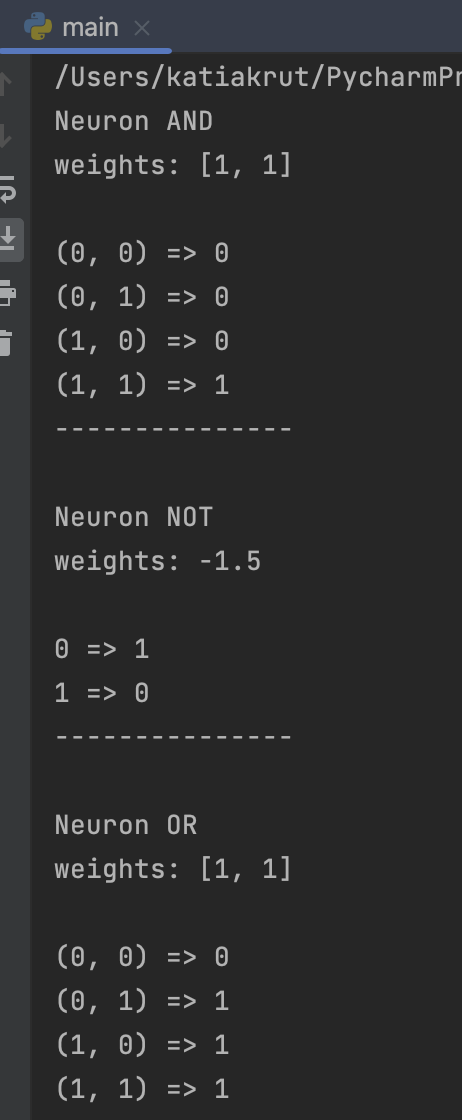
**РЕЗУЛЬТАТИ НАВЧАННЯ ТА ТЕСТУВАННЯ НЕЙРОННОЇ МЕРЕЖІ**

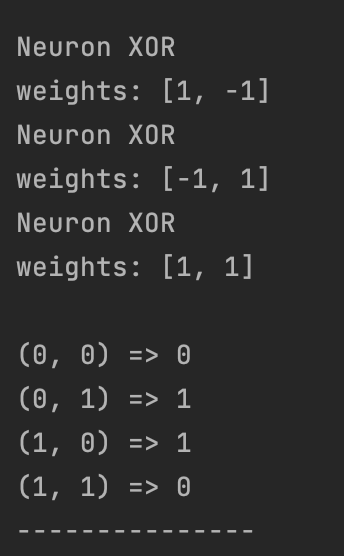
**Для часового ряду**

****



**Для логічних функцій**

****

****

**Висновки:** В ході виконання даної лабораторної роботи було отримано відомості про одно та багатошарові штучні нейронні мережі. Навчилися використовувати навчання з вчителем із використанням зворотного поширення похибки. Створена програма дозволяє на основі тренувальних даних в подальшому генерувати значення з доволі високою точністю за відносно невеликий час, проте чим точніші значення ми хочемо отримати, тим довший час виконання роботи. Реалізували моделі простих нейронних мереж, що здатні виконувати прості логічні функції.