Національний авіаційний університет

Факультет комп’ютерних наук та технологій

**ЗВІТ**

**по лабораторній роботі No 5**

**Нейромережа прямого поширення**

Дисципліна: «Методи штучного інтелекту»

Кафедра: прикладної математики

ОС: бакалавр

Спеціальність: 113 «Прикладна математика»

ОПП: «Прикладне програмне забезпечення»

Виконав: здобувач вищої освіти 4 курсу. 451 групи

Архіпов Олексій Тімурович

Перевірив: Приставка Пилип Олександрович

Київ 2024

**Тема:** Нейромережа прямого поширення

**Завдання:**

* Реалізувати нейрону мережу прямого поширення з мінімум двома шарами.

**Теоретична частина**

Своєю назвою нейронні мережі завдячують аналогії зі структурою груп нейронів в мозку живих істот. Проте, таке порівняння лише частково вірне. На загал, нейронна мережа це – складна функція, що складається з великої кількості композицій інших (часто теж складних) функцій. Особливістю саме нейронних мереж є наявність нелінійних функцій активації σ (⋅) , що застосовуються до «елементів» мережі, які мають назву штучний нейрон

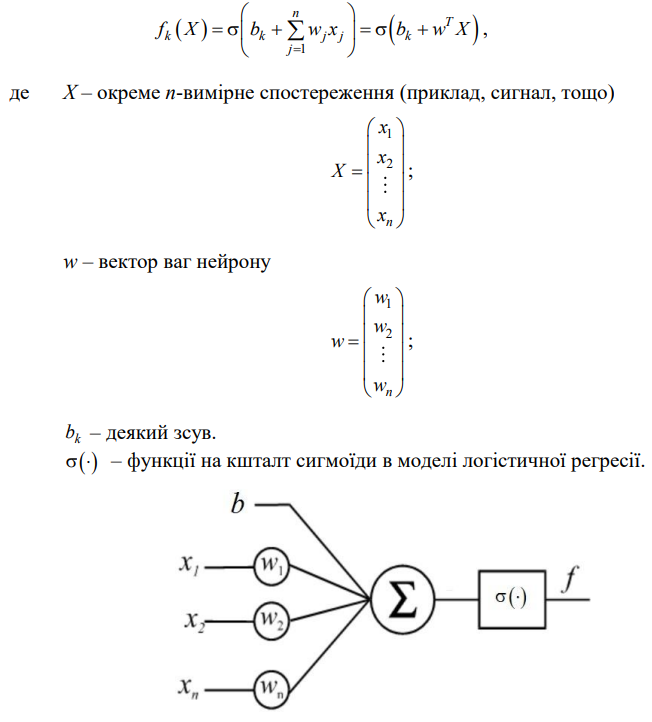
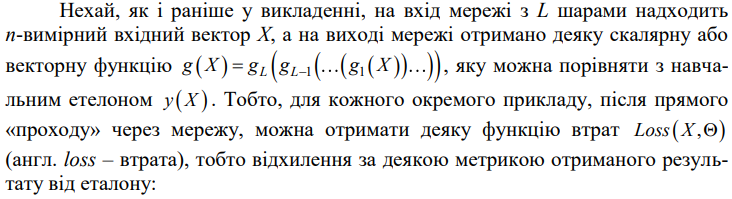


Рис.1. Схематичне представлення штучного нейрона

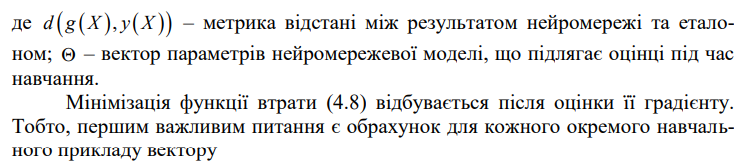
У загальному випадку багатошарову нейромережу можна описати складною функцією g ( X ) , в якій кожен шар мережі – своя вкладена функція:

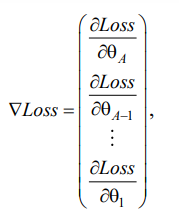


Перейдемо до розгляду методу зворотного поширення похибки, реалізація якого сприятиме використанню градієнтних методів для навчання нейромереж.



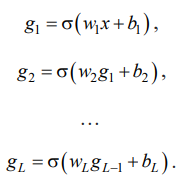


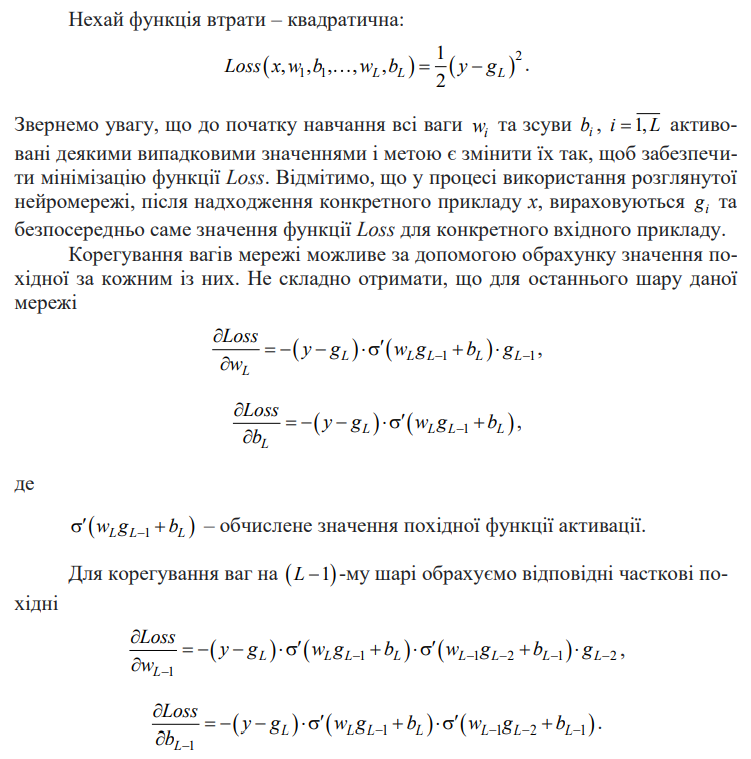


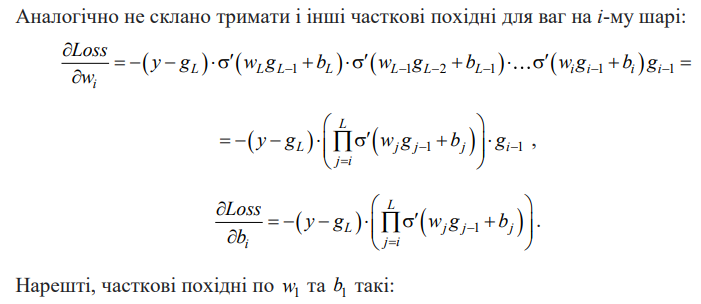


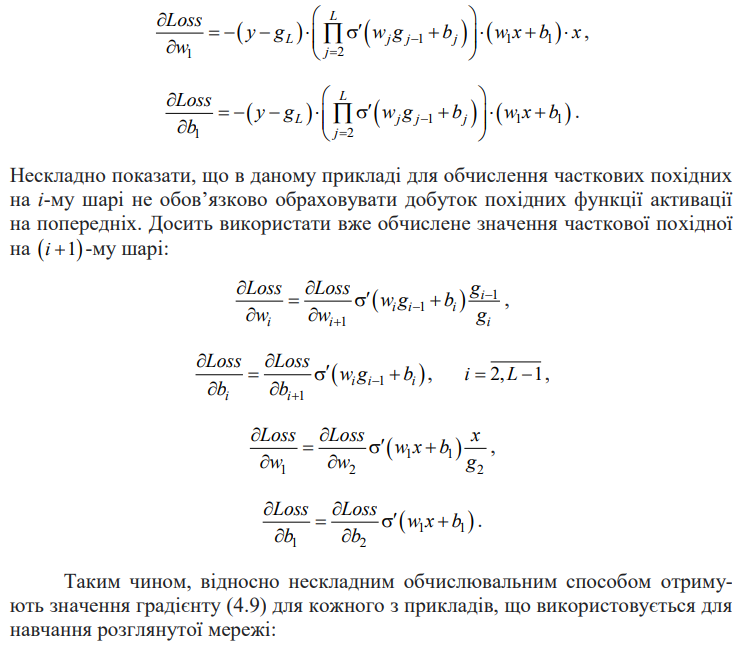
На перший погляд задача оцінки такого вектору градієнту видається складною, враховуючи потенційно велику кількість шарів нейромережі, кількість блоків нелінійної активації в кожному шарі, тощо. Проте, існує ефективний підхід до знаходження ∇Loss , який успішно реалізують в сучасному машинному навчанні. Алгоритм зворотного поширення – дозволяє передавати інформацію про вартість втрати назад по мережі для обчислення градієнта на всіх попередніх шарах мережі. Зауважимо, що під терміном «зворотне поширення» часто розуміють алгоритм навчання багатошарових нейронних мереж. Насправді зворотне поширення відноситься лише до обчислення градієнта, тоді як для навчання за допомогою такого градієнту застосовують інші алгоритми, наприклад, стохастичного градієнтного спуску. Крім того, іноді помилково вважають, що зворотне поширення стосується лише багатошарових нейронних мереж, хоча, фактично, мова йде про обчислення похідних будь-якої складної функції, за умови, що такі існують.

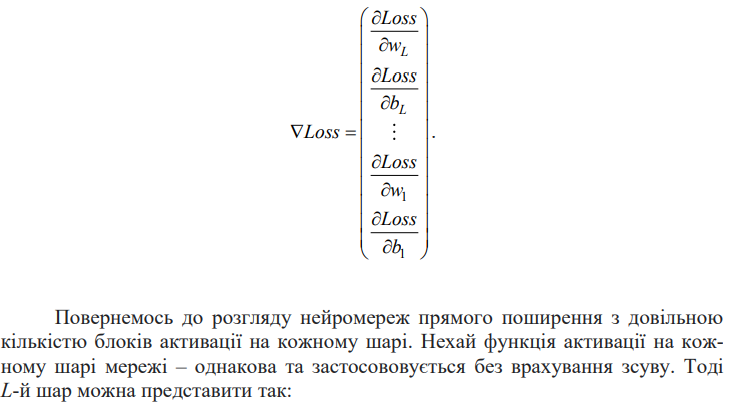
Розглянемо елементарний приклад мережі, що має L шарів та містить на кожному із них лише один блок. На вході та на виході – теж одне значення. Також нехай функція активації на кожному шарі буде однакововою. Тобто, можемо визначити мережу так:

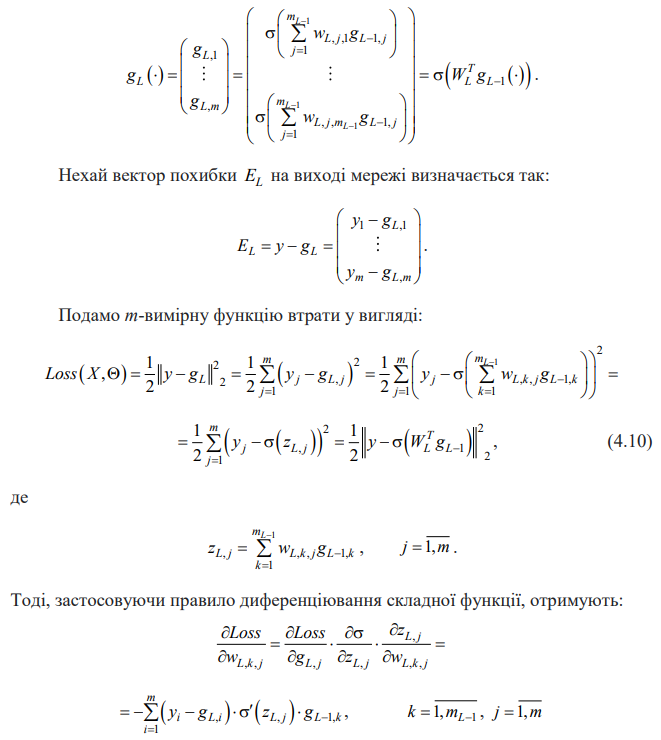


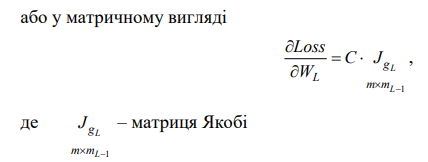


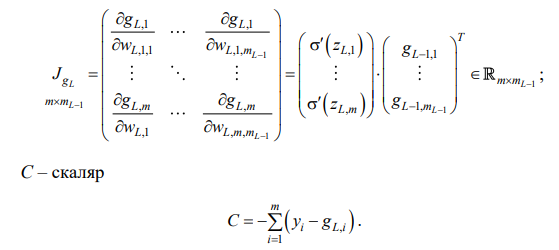


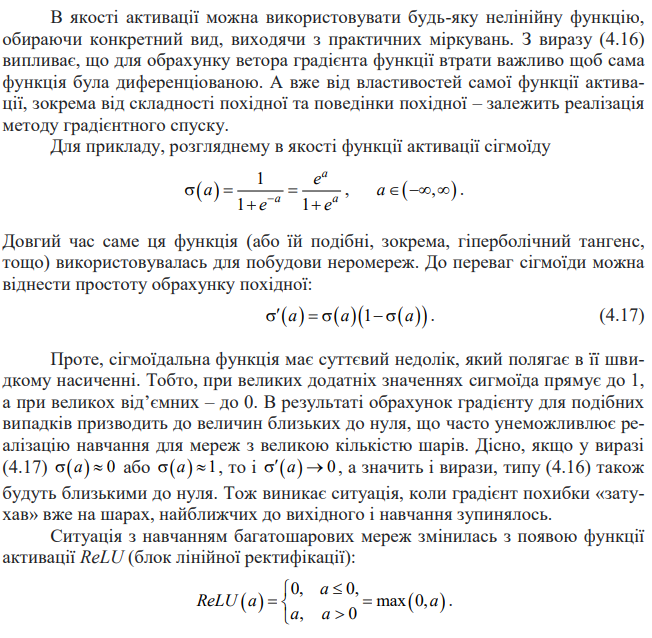


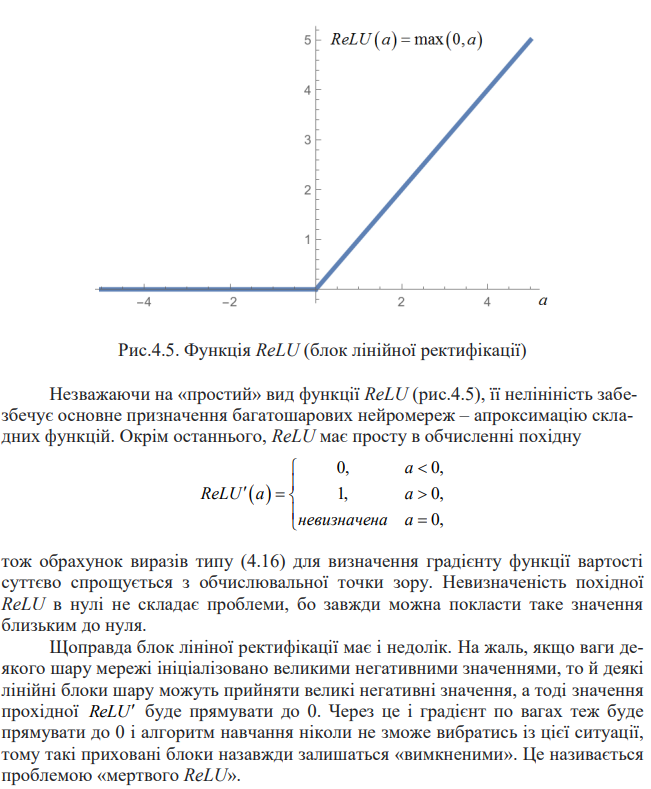






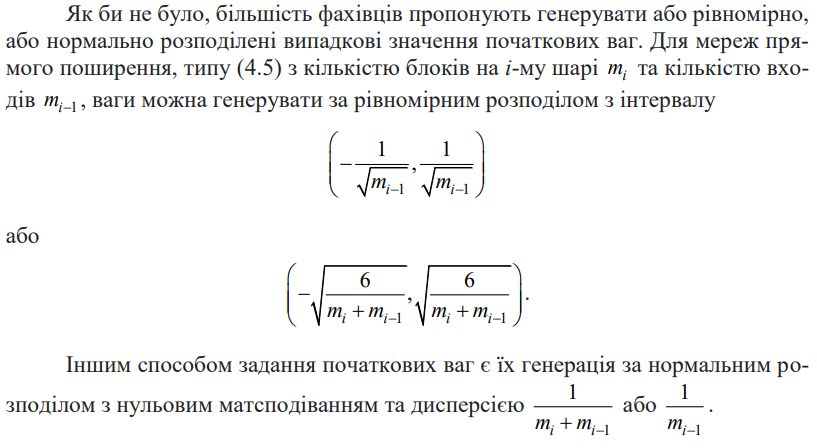






Початкова ініціалізація параметрів. Як для будь-якого ітеративного алгоритму, збіжність навчання нейронних мереж до деякого оптимального значення суттєво може залежати від початкового вибору параметрів, тобто значень вектору Θ. І оскільки цільова функція є неопуклою, від ініціалізації параметрів мережі може сильно залежати знайдене в кінцевому підсумку рішення, а також простота навчання функції, тобто легкість, з якою інформація передається вперед і назад в моделі. Основна ідея при такій ініціалізації полягає в тому, що початкові параметри повинні порушити симетрію між різними блоками. Якщо два прихованих блоки з однаковими функціями активації з'єднані з одими й тиж самими входами, то ці блоки повинні мати різні початкові параметри. Якщо початкові параметри однакові, то детермінований алгоритм навчання, що застосовується до детермінованої функції вартості та моделі, щоразу оновлюватиме ці блоки однаково. Тож зазвичай краще ініціалізувати кожен блок, так щоб він обчислював свою функцію інакше, ніж всі інші блоки і найкраще буде реалізувати таку ініціалізацію випадковим чином.

У типовому випадку зсуви Bi , блоків є евристично обрані константи, а випадково ініціалізуємо тільки ваги. Самі ж початкові ваги моделі ініціалізуються випадковими значеннями з нормальним або рівномірним розподілом, причому масштаб початкового розподілу може сильно впливати як на результат процедури оптимізації, так і на здатність мережі до узагальнення. Чим більші початкові ваги, тим сильніший ефект порушення симетрії, що допомагає уникнути надлишкових блоків. Великі початкові ваги допомагають також унеможливлювати «затухання» градієнту на проміжних шарах під час прямого чи зворотного розповсюдження, адже чим більші значення елементів матриць Wi , тим більшим є результат відповідних їх множень між собою. Однак, з іншого боку, якщо початкові ваги занадто великі, то може статися «вибухове» зростання значень градієнту під час прямого чи зворотного поширення. Крім того, великі ваги можуть стати причиною екстремальних значень, що веде до насичення функції активації (наприклад для сигмоїди) та повної втрати градієнта при поширенні через насичені блоки. Підходи до початкової ініціалізації параметрів з погляду регуляризації та оптимізації можуть дати зовсім різні способи ініціалізації мережі. З точки зору оптимізації, ваги повинні бути досить великими, щоб сприяти успішному розповсюдженню інформації. Але міркування регуляризації спонукають робити менші ваги, щоб не було тенденції до перенавчання.



**Результати роботи програми**

Запустивши навчання нейромережі з двома прихованими шарами. Задавши

* Крок навчання 0.0001
* Перший шар 784 нейронів
* Другий шар 128 нейронів
* Третій шар 128 нейронів
* Вихідний шар 10 нейронів.
* Кількість епох 100

Мета нейромережі навчитися розпізнавати рукописні числа з датасету MNIST.

Нижче наведені результати навчання нейромережі. А саме:

* Графік зміни втрат на тестовій та навчальній вибірках протягом навчання;
* Графік точності класифікацій на тестовій та навчальній вибірках протягом навчання;
* Та розписані результати попередніх графіків для кожної епохи.
* А також матриця якості класифікації.

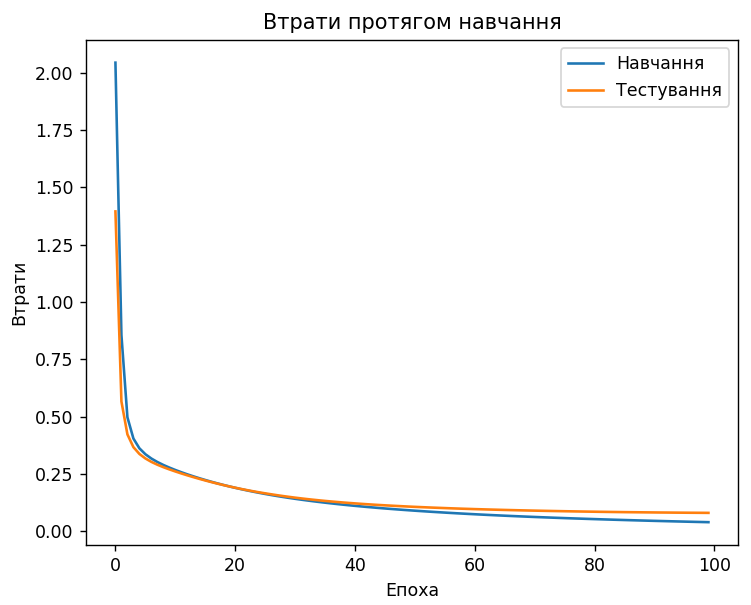


Рис.1. Графік втрат протягом навчання

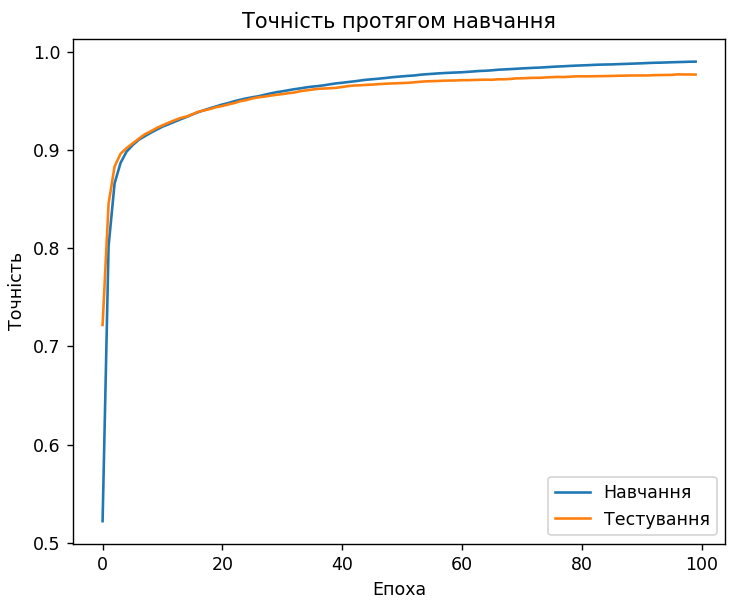


Рис.2. Графік точності протягом навчання

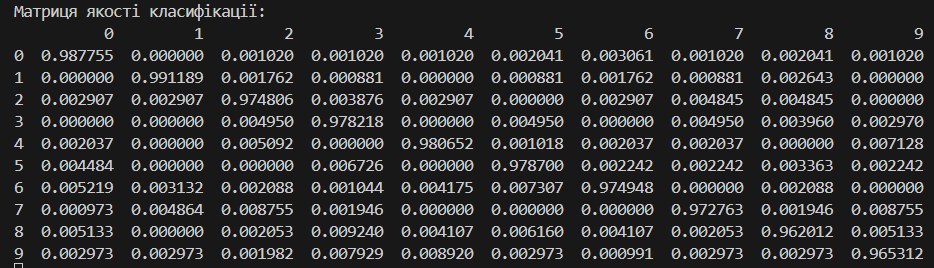


Рис.3. Матриця якості класифікації

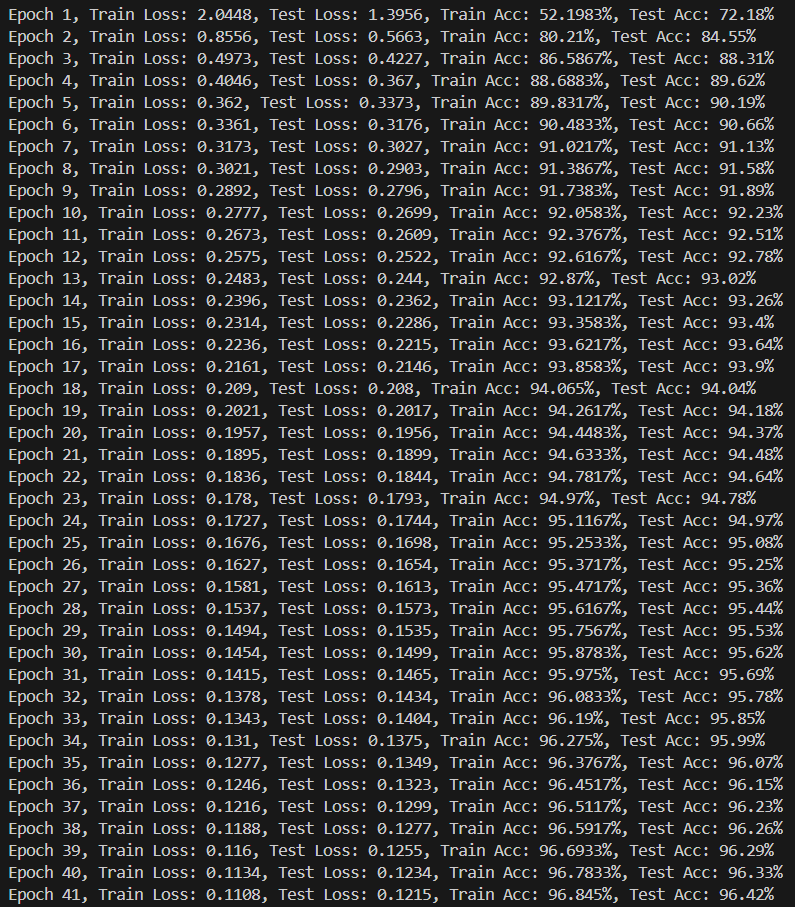


Рис.4. Результати кожної епохи починаючи з 1 закінчуючи 41

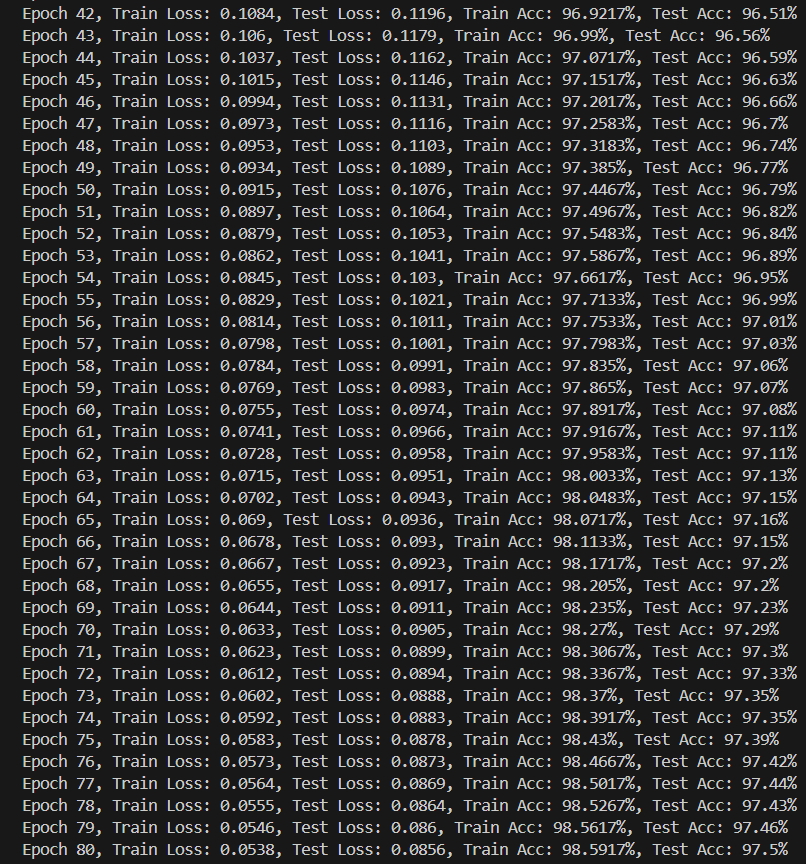


Рис.5. Результати кожної епохи починаючи з 42 закінчуючи 80

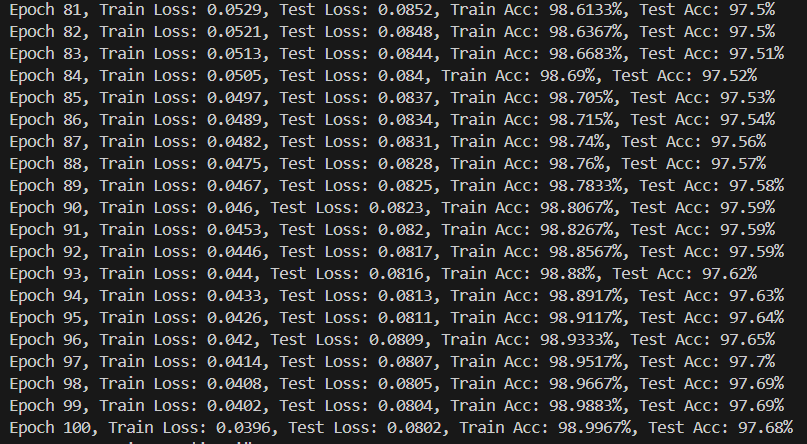
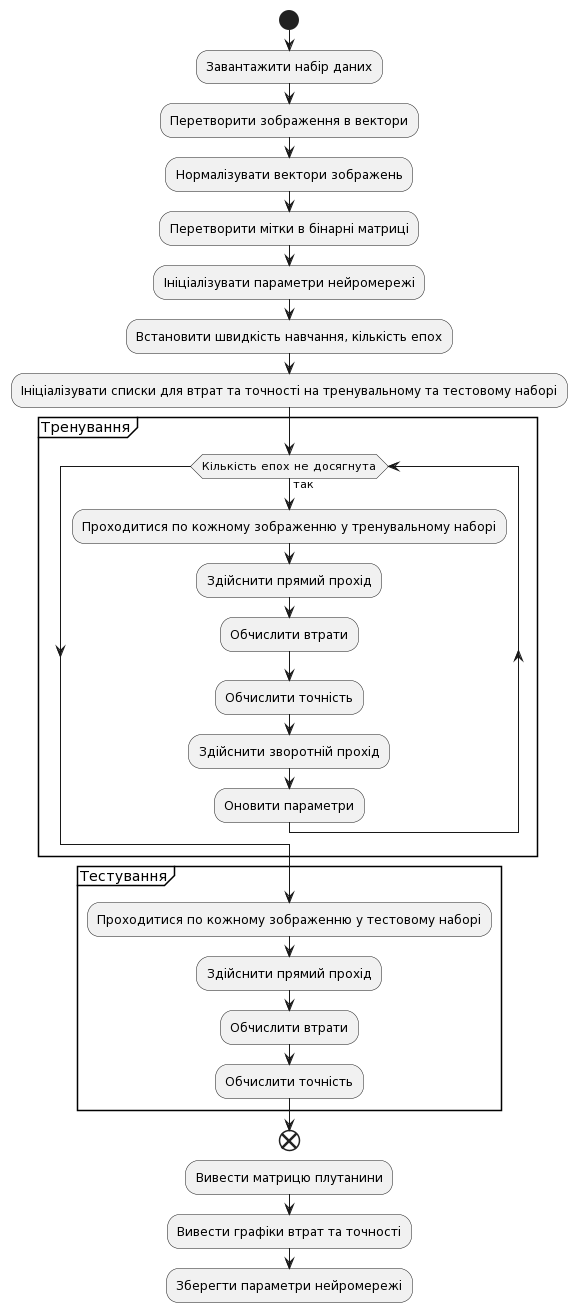


Рис.6. Результати кожної епохи починаючи з 81 закінчуючи 100

Як видно з результатів, нейромережа навчилася досить добре розпізнавати рукописні числа від 0 до 9 на датасеті MNIST.

**Блок-схема**



**Висновок**

* Я реалізував нейрону мережу прямого поширення з двома шарами.

**Список використаної літератури**

* Приставка П.О. Методи штучного інтелекту / Електронний посібник. – К. Національний авіаційний університет, кафедра прикладної математики, 2022 р. – 64 с.

**Код програми**

import numpy as np

from datasets import load\_from\_disk

import pickle

import matplotlib.pyplot as plt

import pandas as pd

mnist\_dataset = load\_from\_disk("mnist\_dataset")

images = mnist\_dataset["train"]["image"]

labels = mnist\_dataset["train"]["label"]

images2 = mnist\_dataset["test"]["image"]

labels2 = mnist\_dataset["test"]["label"]

images\_as\_vectors = np.array([np.reshape(image, -1) for image in images]) /255.0

labels\_as\_array = np.array(labels)

labels\_as\_array = np.eye(10)[labels\_as\_array]

images\_as\_vectors\_test = np.array([np.reshape(image, -1) for image in images2]) /255.0

labels\_as\_array\_test2 = np.array(labels2)

labels\_as\_array\_test = np.eye(10)[labels\_as\_array\_test2]

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    s = sigmoid(x)

    return s \* (1 - s)

def softmax(x):

    x\_exp\_shifted = np.exp(x - np.max(x))

    return x\_exp\_shifted / np.sum(x\_exp\_shifted)

def softmax\_derivative(x):

    s = softmax(x)

    return s \* (1 - s)

def relu(x):

    return np.maximum(x,0)

def relu\_derivative(x):

    return (x >= 0).astype(float)

def forward(X, P):

    hidden\_input1 = np.dot(X, P['w1']) + P['b1']

    hidden\_output1 = relu(hidden\_input1)

    hidden\_input2 = np.dot(hidden\_output1, P['w2']) + P['b2']

    hidden\_output2 = relu(hidden\_input2)

    output\_input = np.dot(hidden\_output2, P['wout']) + P['bout']

    output\_output = softmax(output\_input)

    result = {

        "g1": hidden\_output1,

        "g2": hidden\_output2,

        "g3": output\_output,

        "t1": hidden\_input1,

        "t2": hidden\_input2,

    }

    return result

def initialize\_parameters(input\_size, hidden\_size1, hidden\_size2, output\_size):

    a1 = 1/pow(input\_size, 0.5)

    weights\_hidden1 = np.random.uniform(low=-a1, high=a1, size=(input\_size, hidden\_size1)) #\* 0.01

    bias\_hidden1 = np.zeros((1, hidden\_size1))

    a2 = 1/pow(hidden\_size1, 0.5)

    weights\_hidden2 = np.random.uniform(low=-a2, high=a2,size=(hidden\_size1, hidden\_size2)) #\* 0.01

    bias\_hidden2 = np.zeros((1, hidden\_size2))

    a3 = 1/pow(hidden\_size2, 0.5)

    weights\_output = np.random.uniform(low=-a3, high=a3, size=(hidden\_size2, output\_size)) #\* 0.01

    bias\_output = np.zeros((1, output\_size))

    parameters = {

    "w1": weights\_hidden1,

    "b1": bias\_hidden1,

    "w2": weights\_hidden2,

    "b2": bias\_hidden2,

    "wout": weights\_output,

    "bout": bias\_output

    }

    return parameters

def Loss(gl, y):

    loss = -np.dot(y, np.log(gl.T + 0.00001))

    return np.squeeze(loss)

def back\_propagation(G, y, p, X):

    dE\_dt3 = G['g3'] - y

    dE\_dw3 = np.dot(G['g2'].T, dE\_dt3)

    dE\_db3 = dE\_dt3

    dE\_dg2 = np.dot(dE\_dt3, p['wout'].T)

    dE\_dt2 = dE\_dg2 \* relu\_derivative(G['t2'])

    dE\_dw2 = np.dot(G['g1'].T, dE\_dt2)

    dE\_db2 = dE\_dt2

    dE\_dg1 = np.dot(dE\_dt2, p['w2'].T)

    dE\_dt1 = dE\_dg1 \* relu\_derivative(G['t1'])

    X = np.array(X)

    X = X.reshape(X.size, 1)

    dE\_dw1 = np.dot(X, dE\_dt1)

    dE\_db1 = dE\_dt1

    DW = {

    "dE\_dw3": dE\_dw3,

    "dE\_db3": dE\_db3,

    "dE\_dw2": dE\_dw2,

    "dE\_db2": dE\_db2,

    "dE\_dw1": dE\_dw1,

    "dE\_db1": dE\_db1

    }

    return DW

def upgarde\_parameters(dw, alpha):

    parameters['w1'] -= alpha \* dw['dE\_dw1']

    parameters['w2'] -= alpha \* dw['dE\_dw2']

    parameters['wout'] -= alpha \* dw['dE\_dw3']

    parameters['b1'] -= alpha \* dw['dE\_db1']

    parameters['b2'] -= alpha \* dw['dE\_db2']

    parameters['bout'] -= alpha \* dw['dE\_db3']

input\_size = images\_as\_vectors.shape[1]

hidden\_size1 = 128

hidden\_size2 = 128

output\_size = 10

parameters = initialize\_parameters(input\_size, hidden\_size1, hidden\_size2, output\_size)

learning\_rate = 0.0001

epochs = 100

c = 0

train\_losses, test\_losses = [], []

train\_accuracies, test\_accuracies = [], []

for epoch in range(epochs):

    train\_loss = []

    correct\_train = 0

    for i in range(images\_as\_vectors.shape[0]):

        G = forward(images\_as\_vectors[i], parameters)

        loss = Loss(G['g3'], labels\_as\_array[i])

        train\_loss.append(loss)

        pred = np.argmax(G['g3'])

        correct\_train += (pred == np.argmax(labels\_as\_array[i]))

        dw = back\_propagation(G, labels\_as\_array[i], parameters, images\_as\_vectors[i])

        upgarde\_parameters(dw, learning\_rate)

    train\_losses.append(np.mean(train\_loss))

    train\_acc = correct\_train / images\_as\_vectors.shape[0]

    train\_accuracies.append(train\_acc)

    test\_loss = []

    correct\_test = 0

    for i in range(images\_as\_vectors\_test.shape[0]):

        G = forward(images\_as\_vectors\_test[i], parameters)

        loss = Loss(G['g3'], labels\_as\_array\_test[i])

        test\_loss.append(loss)

        pred = np.argmax(G['g3'])

        correct\_test += (pred == np.argmax(labels\_as\_array\_test[i]))

    test\_losses.append(np.mean(test\_loss))

    test\_acc = correct\_test / images\_as\_vectors\_test.shape[0]

    test\_accuracies.append(test\_acc)

    print(f'Epoch {epoch+1}, Train Loss: {round(train\_losses[-1],4)}, Test Loss: {round(test\_losses[-1],4)}, Train Acc: {round(train\_acc\*100,4)}%, Test Acc: {round(test\_acc\*100,4)}%')

class\_counts = np.zeros(10, dtype=int)

for label in labels\_as\_array\_test2:

    class\_counts[label] += 1

confusion\_matrix = np.zeros((10, 10), dtype=float)

for i in range(images\_as\_vectors\_test.shape[0]):

    G = forward(images\_as\_vectors\_test[i], parameters)

    predicted\_label = np.argmax(G['g3'])

    true\_label = labels\_as\_array\_test2[i]

    confusion\_matrix[true\_label, predicted\_label] += 1.0 / class\_counts[true\_label]

confusion\_df = pd.DataFrame(confusion\_matrix, index=range(10), columns=range(10))

print("Матриця якості класифікації:")

print(confusion\_df)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(train\_losses, label='Навчання')

plt.plot(test\_losses, label='Тестування')

plt.title('Втрати протягом навчання')

plt.xlabel('Епоха')

plt.ylabel('Втрати')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(train\_accuracies, label='Навчання')

plt.plot(test\_accuracies, label='Тестування')

plt.title('Точність протягом навчання')

plt.xlabel('Епоха')

plt.ylabel('Точність')

plt.legend()

plt.tight\_layout()

plt.show()

parameters\_file\_path = "parameters.pkl"

with open(parameters\_file\_path, 'wb') as f:

    pickle.dump(parameters, f)