МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ. ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС

«ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ»

КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

Лабораторна робота №3

з дисципліни: “Нейронні мережі”

Виконали:

студенти групи КА-83

Самошин А.О.

Цепа О.Ю.

Перевірив:

Данилов В.Я.

КИЇВ 2020

Мета: отримати навички розв’язання практичних задач за допомогою мереж Кохонена.

Порядок виконання роботи

1. Реалізувати нейронну мережу мережу Кохонена.

2. За допомогою побудованої нейронної мережі розв’язати задачу класифікації зображень.

3. Результати роботи оформити звітом, який має містити: постановку задачі, навчальну вибірку даних, їх графічне представлення, спосіб кодування зображень для їх представлення нейронній мережі, результати роботи на тестовій множині даних із вказанням похибки, параметри нейронної мережі, що навчилася, вихідний код програми.

Теоретичні відомості

Карти Кохонена, що самоорганізуються — це спеціальний клас штучних НМ, робота яких базується *на конкурентному принципі навчання* (competitive learning): виходи нейронів конкурують між собою за право перейти в стан збудження. Виходом мережі вважається *нейрон-переможець* ("winner takes all"). Для реалізації конкурентного принципу навчання використовуються *латеральні гальмуючі зв'язки*[1] між нейронами. Ця ідея була започаткована Розенблаттом.

У картах самоорганізації КСО (Self-Organizing Maps — SOM) нейрони реалізуються у *вузлах* одномірної або двовимірної решітки. У процесі конкурентного навчання вони *вибірково* налаштовуються на різні вхідні образи (стимули), або класи вхідних образів. Позиції нейронів-переможців впорядковуються відносно інших.

Карти, що самоорганізуються, *формують* топографічне відображення вхідних образів, при якому просторове розташування (координати) нейронів решітки відбиває внутрішні статистичні властивості вхідних образів.

КСО поєднує два рівні адаптації:

– правила адаптації, сформовані на мікрорівні одного нейрона;

– формування експериментально більш ефективних і фізично досяжних властивостей на макрорівні шару.

КСО за своєю природою нелінійні, і їх можна розглядати як нелінійне узагальнення аналізу головних компонентів.

---

[1] Від'ємні зворотні зв'язки.

Розвиток КСО мотивований відмінною рисою людського мозку: входи від різних органів почуттів представлені топологічно упорядкованими обчислювальними відображеннями (картами). Зорова, звукова і дотикальна інформація відображається на різні області і топологічно впорядковується.

Обчислювальні відображення є основними будівельними блоками інфраструктури обробки інформації в нервовій системі людини.

Дві основні моделі відображення ознак

Принцип формування топографічної карти (Кохонен, 1990) полягає в наступному.

Просторове розташування вихідного нейрона топографічної карти відповідає конкретній властивості вхідних даних.

Цей принцип визначає нейробіологічну основу двох різних моделей відображення ознак: Кохонена й Уілшоу-Ван дер Мальсбурга (Willshow-von der Malsburg). В обох моделях вихідні нейрони упорядковані в двовимірні решітки. При такому типі топології кожен нейрон обов'язково має сусідів. Моделі розрізняються способом представлення вхідних образів.

Модель Уілшоу-Ван дер Мальсбурга

Це модель відображення ретини в зорову кору. В ній дві двовимірні решітки, з'єднані одна з одною, проектуються одна на одну (рис. 3.1).

Решітки зв'язані між собою адаптивними зв'язками, що навчаються за правилом Хебба (Hebb).

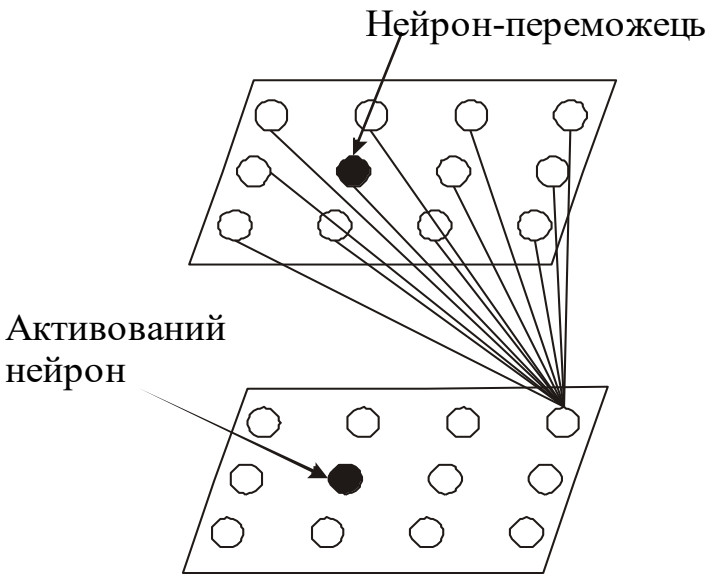


Рис. 3.1. Модель Уілшоу-Ван дер Мальсбурга

Основна ідея цієї моделі полягає в наступному. Геометрично близькі передсинаптичні нейрони кодуються з використанням кореляції їхньої електричної активності, і ці кореляції використовуються в постсинаптичній решітці, щоб сусідні передсинаптичні нейрони відповідали сусіднім постсинаптичним нейронам. Таким чином топологічно упорядковане відображення реалізується за рахунок самоорганізації. Розмірності обох решіток однакові.

Модель Кохонена (1982)

Ця модель не відтворює нейробіологічні деталі, а лише відбиває основні властивості обчислювальних відображень. Вона є більш загальною, ніж перша, оскільки може виконувати *стиснення даних* (зменшення розмірності входів).

Модель Кохонена відноситься до класу алгоритмів векторного кодування. Вона забезпечує топологічне відображення, що оптимально розміщає фіксоване число векторів (слів коду) у вхідному просторі більш

високої розмірності, забезпечуючи, таким чином, стиснення даних (ри

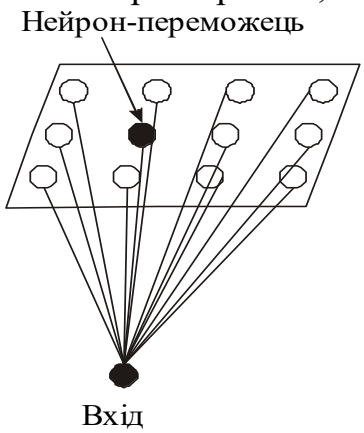


Рис. 3.2. Модель Кохонена

Карти самоорганізації

Основна ідея КСО — перетворити вхідний сигнал довільної розмірності в 1- або 2-мірну дискретну карту і виконати це перетворення адаптивно з урахуванням топологічного впорядкування (рис. 3.3).

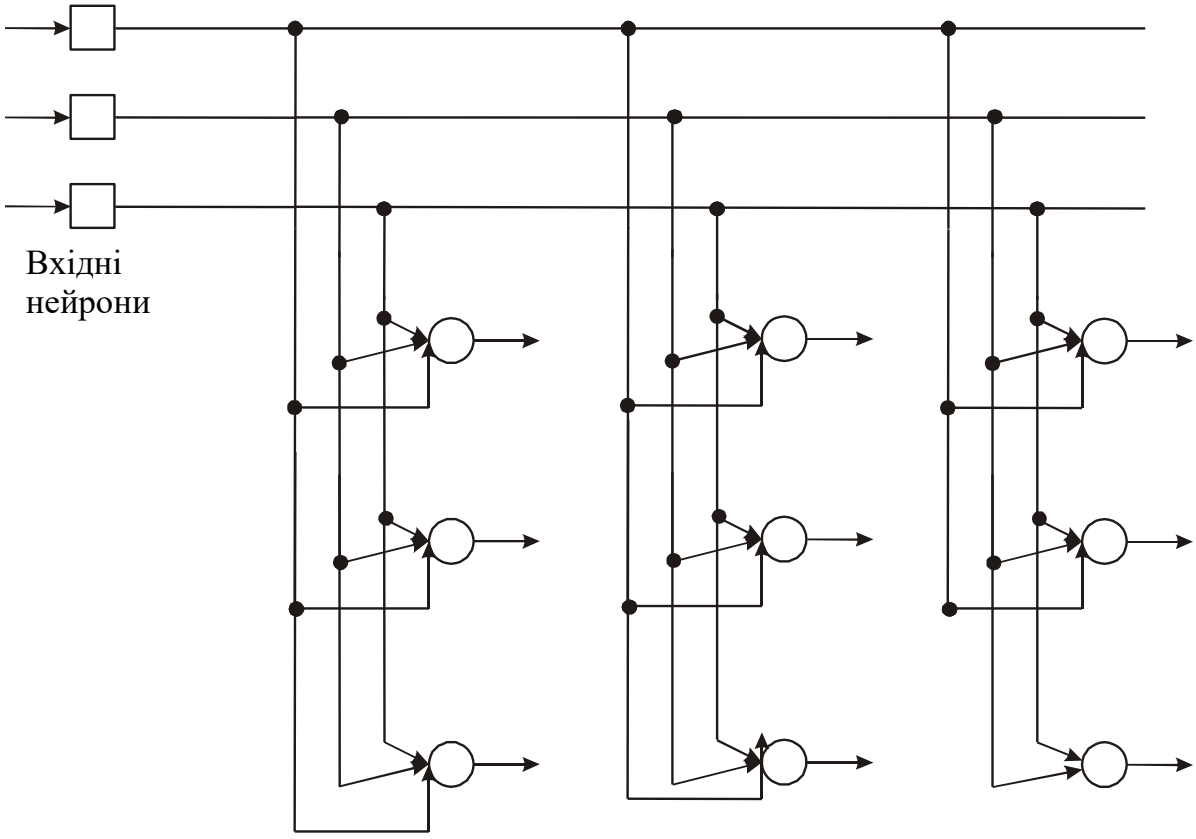


Рис. 3.3. Двовимірна решітка нейронів. Повнозв'язна мережа

Алгоритм самоорганізації складається з наступних етапів:

1. Ініціалізація синаптичних вагових коефіцієнтів у мережі (з використанням датчика випадкових чисел).

2. Конкуренція (competition). Для будь-якого вхідного образа і для всіх нейронів мережі обчислюється значення *дискримінантної функції*, що є основою конкуренції. Нейрон з *максимальним* значенням дискриминантної функції стає *переможцем*.

3. Кооперація. Нейрон-переможець визначає просторове розташування сусідніх збуджених нейронів.

4. Налаштування вагових коефіцієнтів (адаптація). Значення дискриминантної функції збуджених нейронів збільшується для даного образа шляхом налаштування вагових коефіцієнтів. При адаптації відгук нейрона-переможця на близький вхідний образ збільшується.

Конкуренція

Нехай m — розмірність вхідного простору (даних), $x=[x_1,...,x_m]^{\mathrm{T}}$ — вхідний образ. Вектор синаптичних вагових коефіцієнтів для j-го нейрона

$$\omega_j=[\omega_{j1},...,\omega_{jm}]^{\mathrm{T}}, \qquad j=\overline{1,l},$$

де l — число нейронів мережі.

Знайдемо нейрон-переможець

$$i(x)=\arg\max_{\overline{1,l}}\omega_j^{\mathrm{T}}x,$$

$$\max_j w_j^{\mathrm{T}}x \Leftrightarrow \min_j\left\|x-w_j\right\|,$$

$$i(x)=\arg\min_j\left\|x-\omega_j\right\|, \qquad j=\overline{1,l}\ . \qquad (3.1)$$

Вираз (3.1) описує процес конкуренції.

Неперервний простір вхідних образів відображається на дискретний простір нейронів у процесі конкуренції між нейронами мережі.

Кооперація

Нейрон-переможець визначає центр групи (окіл) нейронів, що беруть участь у кооперації. Як коректно визначити окіл? В нейробіології між збудженими нейронами існує *латеральна* взаємодія. Нейрон-переможець

сильніше впливає на топологічно близьких сусідів, ніж на більш віддалені нейрони.

Нехай d_{il} — латеральна відстань між нейроном-переможцем і збудженим нейроном. Топологічний окіл h_{ij} — це унімодальна функція латеральної відстані, що задовольняє двом вимогам:

– h_{ij} — симетрична щодо центра $d_{ij} = 0$;

– амплітуда h_{ij} монотонно зменшується зі збільшенням латеральної відстані і при $d_{ij} \to 0$, $d_{ij} \to 0$.

Цим умовам задовольняє функція Гаусса:

$$h_{ij}(x) = \exp\left(-\frac{d_{ij}^2}{2\sigma^2}\right) \qquad d_{ij}^2 = \left\|r_i - r_j\right\|^2,$$

де r_j — позиція збудженого нейрона j, r_i — дискретна позиція нейрона-переможця.

– Обидві координати визначаються в дискретному вихідному просторі.

– Ширина топологічного околу σ згодом зменшується.

Якщо n — дискретний час, то *експонентне* спадання забезпечує залежність

$$\sigma(n) = \sigma_0 \cdot e^{-\frac{n}{\tau_1}}, \qquad n = 0,1,2,\ldots,$$

де τ_1 — тимчасова константа, σ_0 — початкове значення.

Функція околу:

$$h_{j,i(x)}(n) = \exp\left(-\frac{d_{ij}^2}{2\sigma^2(n)}\right), \qquad n = 0,1,2\ldots \qquad (3.2)$$

Згодом окіл звужується, тому що його ширина зменшується. Нейрони околу будуть брати участь в адаптації вагових коефіцієнтів .

Ціль кооперації — скорелювати зміну вагових коефіцієнтів нейронів одного околу.

Адаптація

Адаптація полягає в зміні вагового коефіцієнта w_j в залежності від вхідного вектора x.

Вона базується на постулаті навчання Хебба:

Правильні зв'язки підсилюються, а хибні слабшають.

Однак у випадку самоорганізації це правило незастосовне, оскільки не відомий цільовий вихід. Якщо зв'язки будуть модифікуватися тільки убік посилення, то незабаром усі вони досягнуть насичення.

Модифікація правила Хебба полягає в використанні *забування*:

$g(y_j)\omega_j$,

де ω_j — синаптичні вагові коефіцієнти нейрона j, $g(y_j)$ — додатна скалярна функція від виходу y_j.

Єдина вимога до функції $g(y_j)$ — щоб залишковий член у її розкладі за формулою Тейлора був рівний нулю, тобто

$$g(y_j)|_{y_j=0} = 0. \qquad (3.3)$$

Модифікація вагових коефіцієнтів обчислюється за формулою:

$$\Delta w_j = \eta y_j x - g(y_j) w_j, \qquad (3.4)$$

де η — коефіцієнт швидкості навчання.

Для виконання умови (3.3), виберемо лінійну функцію

$g(y_j) = \eta y_j$.

Тоді (3.4) перепишеться так

$$\Delta w_j = \eta * y_j (x - w_j) = \eta * h_{ji(x)} (x - w_j).$$

Тоді при переході від моменту часу n до $n+1$ одержимо (Кохонен, 1997)

$$\omega_j(n+1) = \omega_j(n) + \eta(n) h_{j,i(x)}(n)(x - \omega_j(n)), \quad j = \overline{1, l}. \qquad (3.5)$$

Таким чином модифікуються вагові коефіцієнти всіх нейронів з околу нейрона-переможця i. Значення вагового вектора w_i нейрона-переможця i наближається до x. Вектори синаптичних вагових коефіцієнтів відслідковують розподіл вхідних векторів відповідно до вибору околу, забезпечуючи тим самим топологічне упорядкування карти ознак у вхідному просторі.

Для адаптації за формулою (3.5) необхідно мати 2 евристики:

– для вибору функції околу $h_{j,i(x)}(n)$ (див. (3.2))

– для швидкості навчання $\eta(n)$:

$$\eta_0 \approx 0.1 \quad \eta(n) = \eta_0 \cdot e^{-\frac{n}{\tau_2}}, \qquad n = 0,1,2,..., \qquad (3.6)$$

де τ_2 — ще одна часова константа.

Ці евристики можливо неоптимальні, але зазвичай адекватні.

Фази процесу адаптації

Фаза самоорганізації, або впорядковування, вимагає приблизно 1000 або більше ітерацій. Це залежить від $\eta(n)$ і $h_{j,i(x)}(n)$. У початковий момент часу $\eta(0) = 0.1$. Швидкість навчання повинна монотонно зменшуватися, але не спадати нижче $\eta(n) = 0{,}01$. Щоб це відбулося за 1000 ітерацій, з (3.6) випливає

$$\tau_2 = 1000.$$

Окіл $h_{j,i(x)}$ спочатку повинен включати майже всі нейрони решітки (з центром — нейрон-переможець), а згодом звужуватися.

Протягом фази упорядкування окіл $h_{j,i(x)}$ за 1000 ітерацій повинне "стиснутись" до декількох нейронів навколо переможця.

Для *двовимірних* решіток σ_0 повинне бути рівним "радіусу" решітки, а

$$\tau_1 = \frac{1000}{\log \sigma_0}.$$

Фаза збіжності потрібна для тонкого налаштування карти ознак. У загальному випадку число ітерацій цієї фази приблизно $500l$, де l — число нейронів у мережі. Для забезпечення високої статистичної точності параметр навчання $\eta(n)$ в процесі фази збіжності повинен залишатися $\approx 0{,}01$. $\eta(n)$ не повинен збігатись до нуля, щоб система не опинилась в *метастабільному* стані, що характеризує карту ознак з *топологічним* дефектом. В околі переможця повинно міститися 1 або 0 нейронів.

Резюме

Алгоритм КСО Кохонена працюють з наступними компонентами.

– Неперервний простір вхідних образів, згенерованих відповідно до деякого закону розподілу.

– Топологія мережі у формі решітки нейронів, що визначає дискретний вихідний простір.

– Змінна в часі функція околу $h_{j,i(x)}$, що визначає окіл нейрона-переможця.

– Змінний параметр $\eta(n)$, що зменшується в часі від η_0, але ніколи не досягає 0.

Алгоритм КСО включає наступні етапи.

1. Ініціалізація. Вагові коефіцієнти $\omega_j(0) \quad j=\overline{1,l}$ формуються як випадкові вектори. Вагові коефіцієнти $\{w_j(0)\}_{j=1}^{l}$ можна вибрати серед вхідних векторів $\{x_i\}_{i=1}^{N}$ випадковим чином.

2. Пред'явлення образа. Вибрати вектор вхідного простору $x \in R^m$ (випадковим чином).

3. Визначення переможця.

$$i(x) = \arg\min_j \eta(n)\left\|x(n) - w_j(n)\right\|, \qquad j=\overline{1,l} \ .$$

4. Модель налаштування вагових коефіцієнтів усіх нейронів.

$$w_j(n+1) = w_j(n) + \eta(n)\, h_{j,i(x)}(n)(x(n) - w_j(n),$$

де $h_{j,i(x)}$ — функція околу, центром якої є нейрон-переможець.

5. Перехід до п. 2, доки карта ознак не стабілізується.

1. **Навчальна вибірка даних**:
   Зображення задаються за допомогою масиву з 1 та -1 розмірністю 25, де 1 – зафарбований піксель, -1 – незафарбований, для графічного зображення символів створюємо на основі одновимірного масиву масив розмірності 5 х 5.
   Наприклад, буква N задається таким масивом:
   [1, -1, -1, -1, 1,
   1, 1, -1, -1, 1,
   1, -1, 1, -1, 1,
   1, -1, -1, 1, 1,
   1, -1, -1, -1, 1]

Графічне представлення:

Навчальна вибірка містить таку фразу: “Neural network”

2. **Тестова вибірка даних**

Було згенеровано 3 рівні зашумлення (2, 4 та 6 змінених пікселів). Результат роботи на тестовій множині (кількість змінених пікселів – кількість помилок):

0/25 – 0/14
2/25 – 0/14
4/25 – 7/14
6/25 – 8/14

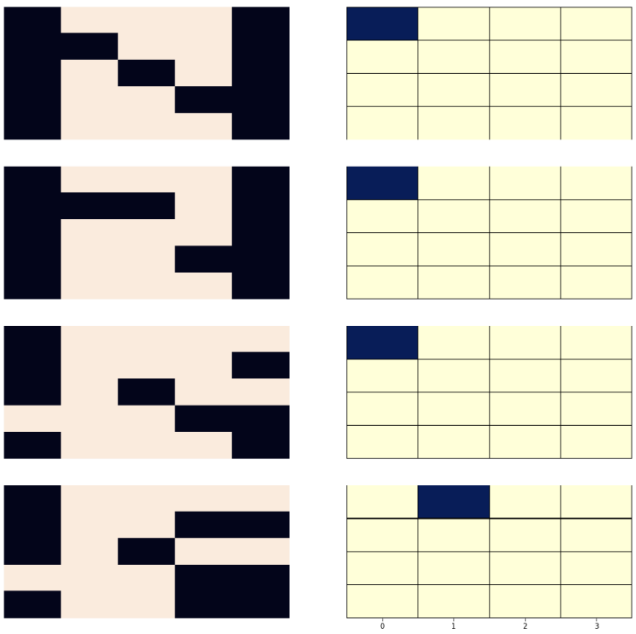
Як видно з результатів, нейронна мережа непогано навчилася класифікувати дані на навчальній вибірці, що і продемонстровано на тестовій, але є куди розвивати мережу.

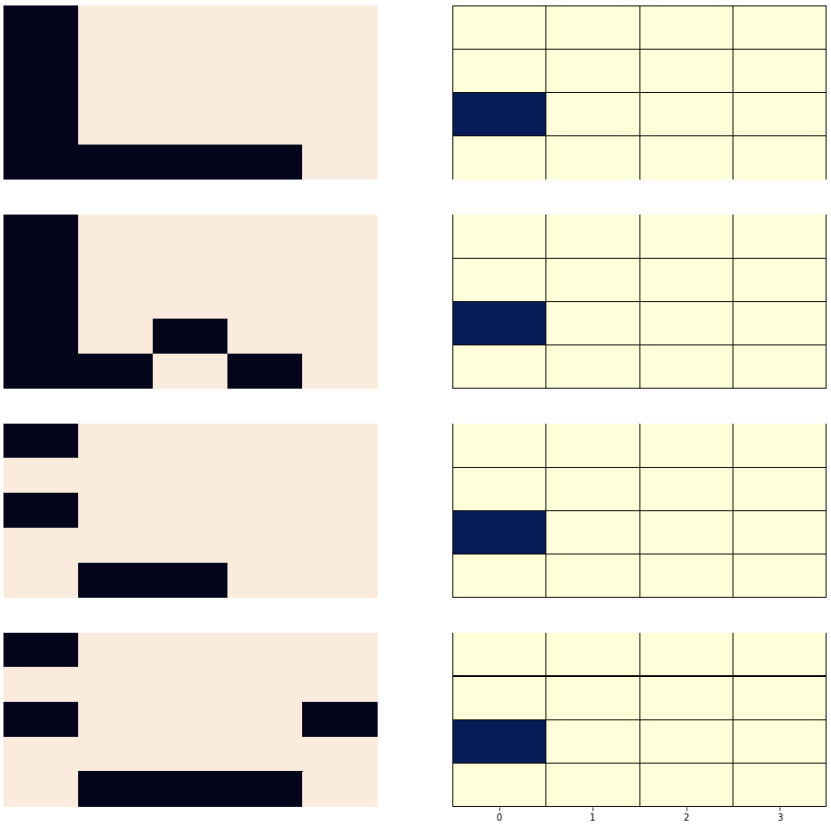
Нейронна мережа містить 16 нейронів, що утворюють двовимірну решітку розмірності 4 x 4.

3. Результати роботи програми

Визначення нейрона переможця та активація нейронів його околу до фази збіжності.

Після фази збіжності сусідні нейрони не активуються:





Кількість ітерацій:

Фаза адаптації: 1000

Фаза збіжності: 8000

Лістинг програми

```
import numpy as np
import random as rd
import seaborn as sb
import matplotlib.pyplot as plt
import math
import copy
I = [[1, -1, -1, -1, 1,
    1, 1, -1, -1, 1,
    1, -1, 1, -1, 1,
    1, -1, -1, 1, 1,
    1, -1, -1, -1, 1], #N

    [1, 1, 1, 1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1],  #E

    [1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
```

```
 1, -1, -1, -1, 1,
 1, -1, -1, 1, 1,
 1, 1, 1, -1, 1],  #U

 [1, 1, 1, 1, -1,
 1, -1, -1, 1, -1,
 1, -1, -1, 1, -1,
 1, 1, 1, 1, -1,
 1, -1, -1, -1, 1],  #R

 [-1, -1, 1, -1, -1,
 -1, 1, -1, 1, -1,
 1, -1, -1, -1, 1,
 1, 1, 1, 1, 1,
 1, -1, -1, -1, 1], #A

 [1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 1, 1, 1, 1, -1], #L

[-1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1],  #space

[1, -1, -1, -1, 1,
 1, 1, -1, -1, 1,
 1, -1, 1, -1, 1,
 1, -1, -1, 1, 1,
 1, -1, -1, -1, 1], #N

[1, 1, 1, 1, -1,
 1, -1, -1, -1, -1,
 1, 1, 1, 1, -1,
 1, -1, -1, -1, -1,
 1, 1, 1, 1, -1],  #E

 [1, 1, 1, 1, 1,
 -1, -1, 1, -1, -1,
 -1,- 1, 1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1],  #T

 [1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
 1, -1, 1, -1, 1,
 1, -1, 1, -1, 1,
 -1, 1, -1, 1, -1],  #W
```

```
    [-1, 1, 1, 1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, -1],  #O

    [1, 1, 1, 1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, 1, -1,
    1, 1, 1, 1, -1,
    1, -1, -1, -1, 1],  #R

    [1, -1, -1, -1, 1,
    1, -1, -1, 1, -1,
    1, 1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, -1, 1]  #K

    ]


I2 = [[1, -1, -1, -1, 1,
    1, 1, 1, -1, 1,
    1, -1, -1, -1, 1,
    1, -1, -1, 1, 1,
    1, -1, -1, -1, 1], #N

    [1, 1, -1, 1, -1,
    1, 1, -1, -1, -1,
    1, 1, 1, 1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1],  #E

    [1, -1, -1, -1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, 1,
    1, 1, 1, -1, 1],  #U

    [1, 1, 1, -1, -1,
    1, -1, -1, 1, -1,
    1, -1, 1, 1, -1,
    1, 1, 1, 1, -1,
    1, -1, -1, -1, 1],  #R

    [-1, -1, 1, -1, -1,
    -1, 1, -1, 1, -1,
    1, -1, -1, -1, 1,
    1, 1, 1, 1, -1,
    1, -1, -1, 1, 1], #A

    [1, -1, -1, -1, -1,
```

```
 1, -1, -1, -1, -1,
 1, -1, -1, -1, -1,
 1, -1, 1, -1, -1,
 1, 1, -1, 1, -1], #L

[-1, -1, -1, -1, -1,
 -1, -1, -1, 1, -1,
 -1, -1, -1, -1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, -1, -1, -1],  #space

[1, -1, -1, -1, 1,
 1, 1, -1, 1, 1,
 1, -1, 1, -1, 1,
 1, -1, -1, -1, 1,
 1, -1, -1, -1, 1], #N

[1, 1, 1, 1, -1,
 1, -1, -1, -1, -1,
 1, 1, -1, 1, -1,
 1, 1, -1, -1, -1,
 1, 1, 1, 1, -1],  #E

 [1, 1, 1, 1, -1,
 -1, -1, 1, -1, -1,
 -1,- 1, 1, 1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, -1, -1],  #T

 [1, -1, -1, -1, 1,
 1, -1, -1, 1, 1,
 1, -1, 1, -1, -1,
 1, -1, 1, -1, 1,
 -1, 1, -1, 1, -1],  #W

 [-1, 1, 1, 1, -1,
 1, -1, -1, 1, 1,
 1, -1, -1, -1, -1,
 1, -1, -1, -1, 1,
 -1, 1, 1, 1, -1],  #O

 [1, 1, 1, 1, -1,
 1, -1, -1, 1, -1,
 1, -1, -1, -1, -1,
 1, 1, 1, 1, 1,
 1, -1, -1, -1, 1],  #R

 [-1, -1, -1, -1, 1,
 1, -1, 1, 1, -1,
 1, 1, 1, -1, -1,
 1, -1, -1, 1, -1,
 1, -1, -1, -1, 1]  #K
```

```
    ]

I4 = [[1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    1, -1, 1, -1, -1,
    -1, -1, -1, 1, 1,
    1, -1, -1, -1, 1], #N

    [1, 1, 1, 1, -1,
    -1, -1, -1, -1, -1,
    1, 1, 1, 1, -1,
    -1, -1, -1, -1, -1,
    -1, 1, -1, 1, -1],  #E

    [-1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    1, 1, 1, -1, 1],  #U

    [1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1,
    1, -1, -1, 1, -1,
    -1, 1, -1, -1, -1,
    1, -1, -1, -1, 1],  #R

    [-1, -1, 1, -1, -1,
    -1, -1, -1, -1, -1,
    1, -1, -1, -1, 1,
    -1, 1, 1, 1, 1,
    -1, -1, -1, -1, 1], #A

    [1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1,
    1, -1, -1, -1, -1,
    -1, -1, -1, -1, -1,
    -1, 1, 1, -1, -1], #L

   [-1, -1, -1, -1, -1,
    -1, 1, -1, 1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, -1, -1, -1],  #space

   [1, -1, -1, -1, 1,
    -1, 1, -1, -1, 1,
    -1, -1, -1, -1, 1,
    1, -1, -1, 1, 1,
    -1, -1, -1, -1, 1], #N

   [-1, 1, 1, -1, -1,
```

```
    -1, -1, -1, -1, -1,
    1, 1, 1, -1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1],  #E

    [1, -1, 1, 1, 1,
    -1, -1, -1, -1, -1,
    -1,- 1, -1, -1, -1,
    -1, -1, 1, -1, -1,
    -1, -1, -1, -1, -1],  #T

    [1, -1, -1, -1, 1,
    -1, -1, -1, -1, 1,
    1, -1, 1, -1, -1,
    -1, -1, 1, -1, 1,
    -1, 1, -1, -1, -1],  #W

    [-1, 1, -1, -1, -1,
    1, -1, -1, -1, 1,
    1, -1, -1, -1,-1,
    1, -1, -1, -1, 1,
    -1, 1, 1, -1, -1],  #O

    [-1, 1, -1, 1, -1,
    1, -1, -1, 1, -1,
    1, -1, -1, -1, -1,
    1, 1, 1, 1, -1,
    -1, -1, -1, -1, 1],  #R

    [1, -1, -1, -1, 1,
    1, -1, -1, 1, -1,
    -1, 1, -1, -1, -1,
    1, -1, -1, -1, -1,
    -1, -1, -1, -1, 1]  #K

    ]

I6 = [[1, -1, -1, -1, -1,
    1, -1, -1, 1, 1,
    1, -1, 1, -1, -1,
    -1, -1, -1, 1, 1,
    1, -1, -1, 1, 1], #N

    [1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1,
    1, 1, 1, 1, -1,
    -1, -1, -1, 1, -1,
    -1, 1, -1, 1, -1],  #E

    [-1, -1, -1, -1, -1,
    1, -1, -1, 1, 1,
    1, -1, -1, -1, -1,
```

```
 1, -1, -1, -1, 1,
 1, 1, 1, 1, 1],  #U

 [1, 1, 1, 1, 1,
 -1, -1, -1, 1, 1,
 1, -1, -1, 1, -1,
 -1, 1, -1, -1, -1,
 1, -1, -1, -1, 1],  #R

 [-1, -1, 1, 1, -1,
 -1, -1, -1, -1, 1,
 1, -1, -1, -1, 1,
 -1, 1, 1, 1, 1,
 -1, -1, -1, -1, 1], #A

 [1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1,
 1, -1, -1, -1, 1,
 -1, -1, -1, -1, -1,
 -1, 1, 1, 1, -1], #L

[1, -1, -1, -1, -1,
 -1, 1, -1, 1, -1,
 -1, -1, 1, -1, -1,
 -1, -1, 1, 1, -1,
 -1, -1, -1, -1, -1],  #space

[1, -1, -1, 1, 1,
 -1, 1, -1, -1, 1,
 -1, -1, 1, -1, 1,
 1, -1, -1, 1, 1,
 -1, -1, -1, -1, 1], #N

[-1, 1, 1, -1, -1,
 -1, 1, -1, -1, -1,
 1, 1, 1, -1, -1,
 1, 1, -1, -1, -1,
 1, 1, 1, 1, -1],  #E

 [1, 1, 1, 1, 1,
 -1, -1, -1, -1, -1,
 -1,- 1, -1, -1, -1,
 -1, 1, 1, -1, -1,
 -1, -1, -1, -1, -1],  #T

 [1, -1, -1, -1, 1,
 -1, -1, -1, -1, 1,
 1, 1, 1, -1, 1,
 -1, -1, 1, -1, 1,
 -1, 1, -1, -1, -1],  #W

 [-1, 1, -1, 1, -1,
```

```
    1, -1, -1, -1, 1,
    1, -1, -1, -1,-1,
    1, -1, -1, -1, 1,
    -1, 1, 1, -1, 1],  #O

    [-1, 1, -1, 1, 1,
    1, -1, -1, 1, -1,
    1, -1, 1, -1, -1,
    1, 1, 1, 1, -1,
    -1, -1, -1, -1, 1],  #R

    [1, -1, -1, -1, 1,
    1, -1, -1, 1, -1,
    -1, 1, -1, -1, -1,
    1, -1, -1, 1, -1,
    -1, -1, 1, -1, 1]  #K

    ]

Coord = [[0, 0], [0,1], [0,2], [0,3],
        [1, 0], [1,1], [1,2], [1,3],
        [2, 0], [2,1], [2,2], [2,3],
        [3, 0], [3,1], [3,2], [3,3]]
W = np.random.uniform(0,1, (16, 25))
#print(W[0])
nu0 = 0.1
t2 = 1000
sg = 2

def noise(I = []):
    print(I[0])
    I2=copy.deepcopy(I)
    for i in range(len(I)):
        for j in range(3):
            r = rd.randint(0,len(I[0])-1)
            I2[i][r]*=-1
    #print(I[0])
    return I2

def t1():
    return 1000/(math.log(sg))
def nu(n):
    return nu0*math.exp(-n/t2)

def sgn(n):
    return sg*math.exp(-n/t1())

def h(n,i,j) :

    return math.exp(-math.pow(-np.linalg.norm(np.array(Coord[i])-
np.array(Coord[j])),2)/(2*math.pow(sgn(n),2)))
```

```
def test(I=[],W=[]):
        T = []
        for q in range(len(I)):
            x = I[q]
            n=15
            min = np.linalg.norm(I[q]-W[15])
            for i in range(15):
                j = np.linalg.norm(I[q]-W[i])
                if j < min:
                    min = j
                    n = i
            T.append(n)
        return T



for k in range(1000):
    x = rd.randint(0, 11)
    min = np.linalg.norm(I[x]-W[15])
    n=15
    for i in range(15):
        j = np.linalg.norm(I[x]-W[i])
        if j < min:
            min = j;
            n = i
    for z in range(len(W)):
            W[z] = W[z] + nu(k)*h(k,n,z)*(I[x]-W[z])



for k in range(8000):
    x = rd.randint(0, 11)
    min = np.linalg.norm(I[x]-W[11])
    n=15
    for i in range(15):
        j = np.linalg.norm(I[x]-W[i])
        if j < min:
            min = j;
            n = i
    for z in range(len(W)):
            W[z] = W[z] + 0.1*h(k,n,z)*(I[x]-W[z])

#for i in range(16):
#    print(W[i])



Test=test(I,W)
Test2=test(I2,W)
Test4=test(I4,W)
Test6=test(I6,W)
```

```
er2 = []
er4 = []
er6 = []


for ii in range(len(I)):
    ht = []
    ht2 =[]
    ht4 =[]
    ht6 =[]
    x=I[ii]
    y=I2[ii]
    z=I4[ii]
    v=I6[ii]

    num = 0
    n = Test[ii]
    i=j=0
    for i in range(4):
        ht.append([])
        for j in range(4):
            ht[i].append(h(k,n,num))
            num+=1
    num = 0
    n = Test2[ii]
    n1 = Test4[ii]
    n2 = Test6[ii]
    i=j=0
    for i in range(4):
        ht2.append([])
        ht4.append([])
        ht6.append([])
        for j in range(4):
            ht2[i].append(h(k,n,num))
            ht4[i].append(h(k,n1,num))
            ht6[i].append(h(k,n2,num))
            num+=1

    f, ax=plt.subplots(4, 2, figsize = (16,16))
    sb.heatmap(np.reshape(I[ii],(5,5)), cmap = sb.cm.rocket_r, ax=ax[0][0], cbar=False,
yticklabels=False, xticklabels=False )
    sb.heatmap(ht, ax = ax[0][1], cmap="YlGnBu", linewidths=0.1, linecolor = 'black',
cbar=False, xticklabels=False, yticklabels=False )
    sb.heatmap(np.reshape(I2[ii],(5,5)), cmap = sb.cm.rocket_r, ax=ax[1][0], cbar=False,
yticklabels=False, xticklabels=False  )
    sb.heatmap(ht2, ax = ax[1][1], cmap="YlGnBu", linewidths=0.1, linecolor = 'black',
cbar=False, xticklabels=False, yticklabels=False )
    sb.heatmap(np.reshape(I4[ii],(5,5)), cmap = sb.cm.rocket_r, ax=ax[2][0], cbar=False,
yticklabels=False, xticklabels=False  )
    sb.heatmap(ht4, ax = ax[2][1], cmap="YlGnBu", linewidths=0.1, linecolor = 'black',
cbar=False, xticklabels=False, yticklabels=False )
```

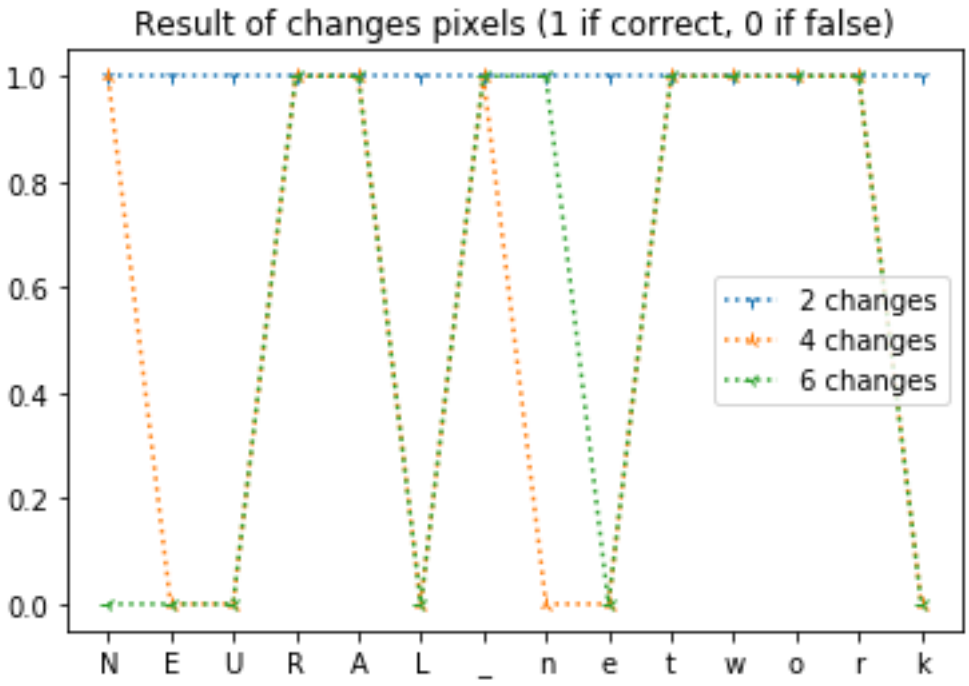
```
    sb.heatmap(np.reshape(I6[ii],(5,5)), cmap = sb.cm.rocket_r, ax=ax[3][0], cbar=False, yticklabels=False, xticklabels=False  )
    sb.heatmap(ht6, ax = ax[3][1], cmap="YlGnBu", linewidths=0.1, linecolor = 'black', cbar=False, yticklabels=False )

    plt.show();

    if (ht==ht2):
        er2.append(1)
    else: er2.append(0)
    if (ht==ht4):
        er4.append(1)
    else: er4.append(0)
    if (ht==ht6):
        er6.append(1)
    else: er6.append(0)
words = ['N','E', 'U', 'R', 'A', 'L', '_', 'n', 'e', 't', 'w', 'o', 'r', 'k']
print(f'2 Changes | {er2} | {er2.count(1)} symbols is correct | {er2.count(0)} symbols is false | Accuracy is {er2.count(1)/14}')
print(f'4 Changes | {er4} | {er4.count(1)} symbols is correct | {er4.count(0)} symbols is false | Accuracy is {er4.count(1)/14}')
print(f'6 Changes | {er6} | {er6.count(1)} symbols is correct | {er6.count(0)} symbols is false | Accuracy is {er6.count(1)/14}')
plt.plot(words, er2, label='2 changes', linestyle=':', marker='1')
plt.plot(words, er4, label='4 changes', linestyle=':', marker='2')
plt.plot(words, er6, label='6 changes', linestyle=':', marker='3')
plt.title('Result of changes pixels (1 if correct, 0 if false)')
plt.legend()
plt.show()
```

Результати роботи програми

2 Changes | [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] | 14 symbols is correct | 0 symbols is false | Accuracy is 1.0

4 Changes | [1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0] | 8 symbols is correct | 6 symbols is false | Accuracy is 0.5714285714285714

6 Changes | [0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0] | 8 symbols is correct | 6 symbols is false | Accuracy is 0.5714285714285714



Висновки

В результаті виконання лабораторної роботи ми програмно реалізували нейронну мережу Кохонена на мові програмування Python, що розв'язує задачу розпізнавання зашумлених символів.