**Министерства науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования**

**«Заполярный Государственный Университет имени Н. М. Федоровского»**

**Кафедра информационных систем и технологий**

**Отчет**

**По дисциплине: «Нейронные сети»**

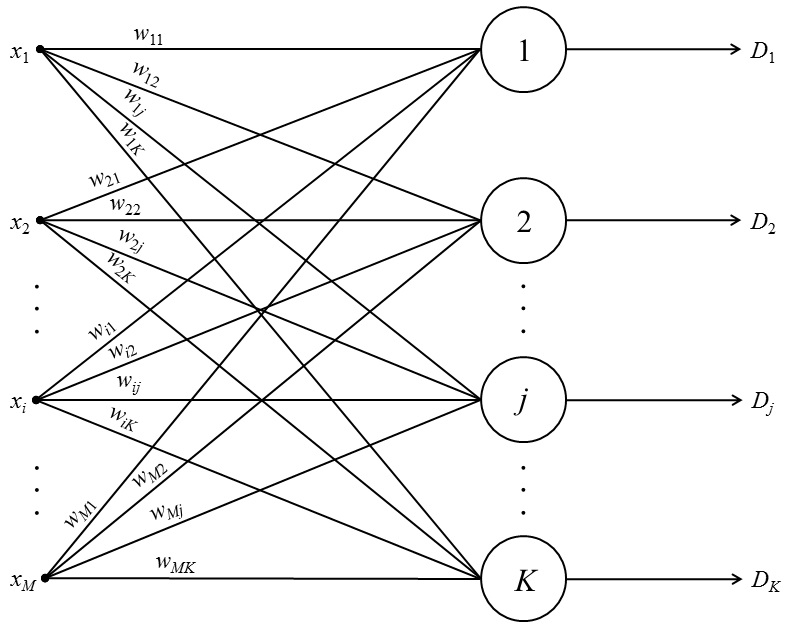
**«Лабораторная работа № 3. Самоорганизующиеся сети Кохонена»**

|  |  |  |  |
| --- | --- | --- | --- |
| Выполнил студент группы ИС-21 |  | Сидельников Максим Эдуардович | |
| Преподаватель: |  | Дыптан Елена Арнольдовна | |
|  |  |  | |
|  |  | | |
|  | Дата защиты: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  | Оценка: | | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Норильск 2024

1. Теоретическая часть

**Нейронные сети Кохонена** — класс нейронных сетей, основным элементом которых является слой Кохонена. Слой Кохонена состоит из адаптивных линейных сумматоров («линейных формальных нейронов»). Как правило, выходные сигналы слоя Кохонена обрабатываются по правилу «Победитель получает всё»: наибольший сигнал превращается в единичный, остальные обращаются в ноль.



1. Общая структура нейронной сети Кохонена.

Алгоритм обучения:

1. Инициализация сети. Задаем весовые коэффициенты матрицы W, как случайные малые значения (-0.3;0.3), α – начальный темп обучения, также вычисляем D0 – максимальное расстояние между векторами.

2. Предъявление сети нового входного сигнала X.

3 Вычисление расстояния от входа X до всех нейронов сети. Рассчитываем

4 Выбираем нейрон k с наименьшим расстоянием dk.

5 Настройка весов нейрона k и всех нейронов, находящихся от него на расстояние не превосходящим

6 Уменьшение значений α N и DN.

7 Шаги 2-6 повторяются до тех пор, пока веса не перестанут меняться (или, когда суммарное изменение всех весов не станет очень мало).

Функционирование сети: на вход подается сигнал, находятся расстояния между входным вектором и всеми нейронами сети, нейрон с наименьшим расстоянием считается победителем, таким образом считаем, к какому классу отнесем образ).

Замечания, повышающие качество нейронной сети:

Если предварительно провести единичную нормировку всех векторов, а также если после каждой итерации процесса обучения осуществлять нормировку весов каждого нейрона столбца матрицы W, то в качестве меры близости входных векторов и весовых векторов нейронов сети можно скалярное произведение между ними. Наименьшим будет расстояние до того нейрона, скалярное произведение до которого максимально. Активационная функция у сети в таком случае будет тождественная

1. Практическая часть

Задача: сделать программу классификации образов цифр от 0 до 9. Реализация на языке программирования python с преобразованием картинок попиксельно во входной вектор. Во время обучения сети в случайном порядке подаются образы. Количество нейронов зависит от количества цифр выборки, в данном случае 10. По итогу обучения сети каждый нейрон должен получить своё представление об одной из цифр выборки. При тестировании, нейрон считается победителем если расстояние между его весами и образом наименьшее из всех.

Присутствует возможность загрузки и сохранения скорректированных весов нейронной сети.

В зависимости от изменения параметров обучения нейронная сеть реагирует по-разному.

**Листинг**

**import numpy as np**

**import os**

**import tkinter as tk**

**from tkinter import Canvas, messagebox, Toplevel, Label, OptionMenu, StringVar**

**from PIL import Image, ImageDraw**

**import matplotlib.pyplot as plt**

**np.random.seed(41)**

**#Класс с самоорганизующейся картой Кохонена (SOM потому что Self-organized map, ес ай эм фром инглэнд :D)**

**class SOM:**

**def \_\_init\_\_(self, input\_dim, num\_neurons, learning\_rate=0.1, radius=2.0, convergence\_threshold=0.1, max\_epochs=1000):**

**self.input\_dim = input\_dim**

**self.num\_neurons = num\_neurons**

**self.learning\_rate = learning\_rate**

**self.radius = radius**

**self.convergence\_threshold = convergence\_threshold**

**self.max\_epochs = max\_epochs**

**self.weights = np.random.uniform(-0.3, 0.3, (input\_dim, num\_neurons))**

**def update\_weights(self, x, winner\_idx):**

**# Проходим по каждому нейрону**

**for i in range(self.weights.shape[1]):**

**# Считаем расстояние от победившего нейрона до текущего**

**distance = np.linalg.norm(np.array([winner\_idx]) - np.array([i]))**

**# Если нейрон находится в радиусе влияния**

**if distance <= self.radius:**

**# Считаем влияние (функция Гаусса)**

**influence = np.exp(-distance\*\*2 / (2 \* (self.radius\*\*2)))**

**# Обновляем веса текущего нейрона**

**self.weights[:, i] += self.learning\_rate \* influence \* (x - self.weights[:, i])**

**def train(self, data):**

**# Начинаем обучение**

**epoch = 0**

**flag = True**

**indices = np.random.permutation(len(data))**

**data\_shuffled = data[indices]**

**# Пока не сойдется или не превысит max\_epochs**

**while flag and epoch < self.max\_epochs:**

**# Копируем текущие веса перед их обновлением**

**prev\_weights = self.weights.copy()**

**# Проходим по каждому образцу данных**

**for x in data\_shuffled:**

**# Считаем расстояния до всех нейронов**

**distances = np.linalg.norm(self.weights - x[:, np.newaxis], axis=0)**

**# Ищем нейрон-победитель (с минимальным расстоянием)**

**winner\_idx = np.argmin(distances)**

**# Обновляем веса нейронов**

**self.update\_weights(x, winner\_idx)**

**# Считаем изменение весов**

**weight\_change = np.linalg.norm(self.weights - prev\_weights)**

**print(f"Эпоха: {epoch}, скорость обучения: {self.learning\_rate:.6f}, радиус: {self.radius:.6f}, изменение весов: {weight\_change:.6f}")**

**# Проверяем, сошлись ли веса (меньше порога сходимости)**

**if weight\_change < self.convergence\_threshold:**

**print(f"Training converged after {epoch} epochs")**

**flag = False**

**# Уменьшаем скорость обучения и радиус**

**self.learning\_rate \*= 0.9**

**self.radius \*= 0.9**

**epoch += 1**

**def predict(self, x):**

**# Считаем расстояния до всех нейронов**

**distances = np.linalg.norm(self.weights - x[:, np.newaxis], axis=0)**

**# Возвращаем индекс победившего нейрона**

**return np.argmin(distances)**

**def save\_weights(self, filename):**

**# Сохраняем веса в файл**

**np.save(filename, self.weights)**

**def load\_weights(self, filename):**

**# Загружаем веса из файла**

**self.weights = np.load(filename)**

**#Класс для обработки картинок (центрирование)**

**class ImageProcessor:**

**@staticmethod**

**def get\_object\_bounds(image):**

**image\_array = np.array(image)**

**non\_empty\_columns = np.where(image\_array.min(axis=0) < 255)[0]**

**non\_empty\_rows = np.where(image\_array.min(axis=1) < 255)[0]**

**if non\_empty\_columns.any() and non\_empty\_rows.any():**

**upper, lower = non\_empty\_rows[0], non\_empty\_rows[-1]**

**left, right = non\_empty\_columns[0], non\_empty\_columns[-1]**

**return left, upper, right, lower**

**else:**

**return None**

**@staticmethod**

**def center\_object(image):**

**bounds = ImageProcessor.get\_object\_bounds(image)**

**if bounds:**

**left, upper, right, lower = bounds**

**object\_width = right - left**

**object\_height = lower - upper**

**horizontal\_padding = (image.width - object\_width) // 2**

**vertical\_padding = (image.height - object\_height) // 2**

**cropped\_image = image.crop(bounds)**

**centered\_image = Image.new("L", (image.width, image.height), "white")**

**centered\_image.paste(cropped\_image, (horizontal\_padding, vertical\_padding))**

**return centered\_image**

**return image**

**@staticmethod**

**def load\_images(folder):**

**images = []**

**labels = []**

**for filename in os.listdir(folder):**

**img\_path = os.path.join(folder, filename)**

**try:**

**with Image.open(img\_path) as img:**

**img = img.convert("L")**

**img = ImageProcessor.center\_object(img)**

**img = img.resize((100, 100))**

**images.append(np.asarray(img).flatten() / 255.0)**

**label = int(filename[0])**

**labels.append(label)**

**except Exception as e:**

**print(f"Error loading {img\_path}: {e}")**

**return np.array(images), np.array(labels)**

**#Визуал**

**class TestCanvas(tk.Tk):**

**def \_\_init\_\_(self, som, neuron\_assignments):**

**super().\_\_init\_\_()**

**self.som = som**

**self.neuron\_assignments = neuron\_assignments**

**self.canvas = Canvas(self, width=280, height=280, bg="white")**

**self.canvas.pack()**

**self.bind("<B1-Motion>", self.draw)**

**self.image = Image.new("L", (280, 280), "white")**

**self.draw\_image = ImageDraw.Draw(self.image)**

**test\_button = tk.Button(self, text="Проверить", command=self.test\_image)**

**test\_button.pack(side=tk.BOTTOM)**

**clear\_button = tk.Button(self, text="Очистить", command=self.clear\_canvas)**

**clear\_button.pack(side=tk.BOTTOM)**

**save\_weights = tk.Button(self, text="Сохранить веса", command=self.save\_weights)**

**save\_weights.pack(side=tk.BOTTOM)**

**research = tk.Button(self, text="Провести эксперименты", command=self.research)**

**research.pack(side=tk.BOTTOM)**

**assign\_neurons = tk.Button(self, text="Назначить нейроны", command=self.assign\_neurons)**

**assign\_neurons.pack(side=tk.BOTTOM)**

**graf = tk.Button(self, text="Показать графики", command=self.graf)**

**graf.pack(side=tk.BOTTOM)**

**def draw(self, event):**

**x, y = event.x, event.y**

**self.canvas.create\_oval(x-7, y-7, x+7, y+7, fill='black')**

**self.draw\_image.ellipse([x-7, y-7, x+7, y+7], fill='black')**

**def test\_image(self):**

**centered\_image = ImageProcessor.center\_object(self.image)**

**inverted\_image = centered\_image.resize((100, 100))**

**img\_array = np.array(inverted\_image) / 255.0**

**img\_array = img\_array.flatten()**

**min\_index = self.som.predict(img\_array)**

**assigned\_class = self.neuron\_assignments[min\_index]**

**messagebox.showinfo("Результат", f"Это похоже на класс {assigned\_class}!")**

**self.clear\_canvas()**

**def clear\_canvas(self):**

**self.canvas.delete("all")**

**self.image = Image.new("L", (280, 280), "white")**

**self.draw\_image = ImageDraw.Draw(self.image)**

**def research(self):**

**folder\_path = "testing"**

**re\_images, re\_labels = ImageProcessor.load\_images(folder\_path)**

**wrong\_count = 0**

**for img, label in zip(re\_images, re\_labels):**

**flag = self.som.predict(img)**

**assigned\_class = self.neuron\_assignments[flag]**

**if assigned\_class != label:**

**wrong\_count += 1**

**messagebox.showinfo("Результат", f"Кол-во ошибок: {wrong\_count}, процент ошибок: {(wrong\_count/len(re\_images))\*100:.2f}%")**

**def graf(self):**

**fig, axes = plt.subplots(2, 5, figsize=(15, 6))**

**for i, ax in enumerate(axes.flat):**

**if i < self.som.weights.shape[1]:**

**ax.imshow(self.som.weights[:, i].reshape(100, 100), cmap='gray')**

**ax.set\_title(f"Нейрон {i}")**

**ax.axis('off')**

**plt.tight\_layout()**

**plt.show()**

**def save\_weights(self):**

**filename = f"weights-koh{self.som.learning\_rate}.npy"**

**self.som.save\_weights(filename)**

**messagebox.showinfo("Результат", f"Веса сохранены в {filename}!")**

**def assign\_neurons(self):**

**assign\_window = Toplevel(self)**

**assign\_window.title("Назначить нейроны")**

**labels = []**

**options = [str(i) for i in range(10)]**

**class\_vars = []**

**for i in range(10):**

**Label(assign\_window, text=f"Нейрон {i}:").grid(row=i, column=0, padx=10, pady=5)**

**class\_var = StringVar(assign\_window)**

**class\_var.set(self.neuron\_assignments[i])**

**class\_menu = OptionMenu(assign\_window, class\_var, \*options)**

**class\_menu.grid(row=i, column=1, padx=10, pady=5)**

**class\_vars.append(class\_var)**

**def save\_assignments():**

**for i, class\_var in enumerate(class\_vars):**

**self.neuron\_assignments[i] = int(class\_var.get())**

**messagebox.showinfo("Результат", "Назначения нейронов сохранены!")**

**assign\_window.destroy()**

**save\_button = tk.Button(assign\_window, text="Сохранить", command=save\_assignments)**

**save\_button.grid(row=10, column=0, columnspan=2, padx=10, pady=10)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**folder\_path = "dataset"**

**input\_dim = 10000**

**num\_neurons = 10**

**som = SOM(input\_dim, num\_neurons)**

**neuron\_assignments = {i: i for i in range(num\_neurons)}**

**try:**

**som.load\_weights(f"weights-koh{som.learning\_rate}.npy")**

**print("Weights loaded successfully")**

**except FileNotFoundError:**

**images, \_ = ImageProcessor.load\_images(folder\_path)**

**som.train(images)**

**som.save\_weights(f"weights-koh{som.learning\_rate}.npy")**

**print("Weights saved successfully after training")**

**app = TestCanvas(som, neuron\_assignments)**

**app.title("Тест нейронной сети.")**

**app.mainloop()**