



**МИНОБРНАУКИ РОССИИ**  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
**«МИРЭА - Российский технологический университет»**  
**РТУ МИРЭА**

---

**Институт Информационных Технологий**  
**Кафедра Вычислительной Техники**

**ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ**  
  
**по дисциплине**  
**«Проектирование и обучение нейронных сетей»**

Студент группы: ИКБО-04-22

Кликушин В.И.  
(Ф. И.О. студента)

Преподаватель

Семенов Р.Э.  
(Ф.И.О. преподавателя)

Москва 2024

# СОДЕРЖАНИЕ

|   |    |
|---|----|
| ВВЕДЕНИЕ .....                          | 4  |
| 1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА .....      | 6  |
| 1.1 Описание структуры .....            | 6  |
| 1.2 Постановка задачи.....              | 7  |
| 1.3 Алгоритм обучения.....              | 7  |
| 1.4 Программная реализация .....        | 8  |
| 1.5 Выводы по разделу.....              | 10 |
| 2 ДЕЛЬТА ПРАВИЛО .....                  | 11 |
| 2.1 Описание структуры .....            | 11 |
| 2.2 Постановка задачи.....              | 11 |
| 2.3 Алгоритм обучения.....              | 12 |
| 2.4 Программная реализация .....        | 13 |
| 2.5 Выводы по разделу.....              | 14 |
| 3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ ..... | 16 |
| 3.1 Описание структуры .....            | 16 |
| 3.2 Постановка задачи.....              | 16 |
| 3.3 Алгоритм обучения.....              | 17 |
| 3.4 Программная реализация .....        | 19 |
| 3.5 Выводы по разделу.....              | 21 |
| 4 СЕТЬ РАДИАЛЬНО-БАЗИСНЫХ ФУНКЦИЙ.....  | 22 |
| 4.1 Описание структуры .....            | 22 |
| 4.2 Постановка задачи.....              | 23 |
| 4.3 Алгоритм обучения.....              | 23 |
| 4.4 Программная реализация .....        | 24 |
| 4.5 Выводы по разделу.....              | 25 |
| 5 КАРТА КОХОНЕНА .....                  | 26 |
| 5.1 Описание структуры .....            | 26 |
| 5.2 Постановка задачи.....              | 27 |

|  |    |
|--|----|
| 5.3 Алгоритм обучения.....             | 27 |
| 5.4 Программная реализация .....       | 28 |
| 5.5 Выводы по разделу.....             | 31 |
| 6 СЕТЬ ВСТРЕЧНОГО РАСПРОСТРАНЕНИЯ..... | 32 |
| 6.1 Описание структуры .....           | 32 |
| 6.2 Постановка задачи.....             | 32 |
| 6.3 Алгоритм обучения.....             | 33 |
| 6.4 Программная реализация .....       | 33 |
| 6.5 Выводы по разделу.....             | 34 |
| 7 РЕКУРРЕНТНАЯ СЕТЬ .....              | 35 |
| 7.1 Описание структуры .....           | 35 |
| 7.2 Постановка задачи.....             | 36 |
| 7.3 Алгоритм обучения.....             | 36 |
| 7.4 Программная реализация .....       | 37 |
| 7.5 Выводы по разделу.....             | 38 |
| 8 СВЕРТОЧНАЯ СЕТЬ.....                 | 40 |
| 8.1 Описание структуры .....           | 40 |
| 8.2 Постановка задачи.....             | 41 |
| 8.3 Алгоритм обучения.....             | 41 |
| 8.4 Программная реализация .....       | 42 |
| 8.5 Выводы по разделу.....             | 43 |
| ЗАКЛЮЧЕНИЕ .....                       | 45 |
| ПРИЛОЖЕНИЯ.....                        | 47 |

# ВВЕДЕНИЕ

Современные технологии искусственного интеллекта (ИИ) находят все более широкое применение в различных сферах жизни и деятельности человека. Одной из ключевых областей ИИ является проектирование и обучение нейронных сетей, которые моделируют процессы, происходящие в человеческом мозге. Нейронные сети представляют собой мощный инструмент для решения задач классификации, регрессии, распознавания образов, обработки естественного языка и многих других.

В рамках данной курсовой работы рассматриваются основные алгоритмы обучения нейронных сетей, которые являются фундаментальными для понимания их функционирования. В частности, будут изучены такие методы, как обучение Хебба, дельта-правило, обратное распространение ошибки, радиально-базисные функции, карта Кохонена, сеть встречного распространения, рекуррентные сети и сверточные сети. Каждый из этих подходов имеет свои особенности и применяется для решения конкретных задач, что делает их изучение важным этапом в освоении теории и практики нейронных сетей.

Обучение Хебба, как один из первых алгоритмов обучения, основан на идее, что связь между нейронами усиливается, если они одновременно активны. Дельта-правило, являющееся обобщением обучения Хебба, позволяет минимизировать ошибку между желаемым и фактическим выходом сети. Обратное распространение ошибки, в свою очередь, представляет собой более сложный и мощный метод, который широко используется для обучения многослойных перцептронов.

Радиально-базисные функции (РБФ) и карта Кохонена применяются для решения задач кластеризации и визуализации данных. Сеть встречного распространения объединяет в себе свойства карт Кохонена и РБФ, что позволяет эффективно решать задачи ассоциативного обучения. Рекуррентные сети, в отличие от традиционных нейронных сетей, способны обрабатывать последовательности данных, что делает их незаменимыми для задач, связанных

с обработкой временных рядов и естественного языка. Сверточные сети (CNN) стали стандартом де-факто в задачах компьютерного зрения благодаря своей способности эффективно извлекать признаки из изображений.

Изучение перечисленных методов обучения нейронных сетей позволяет не только глубоко понять их принципы работы, но и выбрать наиболее подходящий подход для решения конкретной задачи. В данной курсовой работе будет проведено детальное исследование каждого из методов, их преимуществ и недостатков, а также их применение на практике.

Целью курсовой работы является систематизация знаний о различных алгоритмах обучения нейронных сетей, их сравнительный анализ и выявление наиболее эффективных подходов для решения задач в области искусственного интеллекта.

# 1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА

## 1.1 Описание структуры

В 1949 г. американский нейрофизиолог Д. Хебб (D.O.Hebb) изложил видение механизмов ассоциативной памяти в книге «Организация поведения» (The Organization of Behavior), предложил постулат обучения, ставший основой обучения некоторых нейронных сетей. В ходе наблюдений ученый установил, что по мере того как человек обучается различным задачам, связи в головном мозге постоянно изменяются и образуются ансамбли нейронов. Позднее в 1973г. была опубликована работа, подтверждающая физиологическую реализацию правила обучения Д. Хебба в области мозга – гиппокампус, играющего ключевую роль при запоминании информации и обучении.

### **Постулат Хебба:**

Если аксон клетки А находится на достаточно близком расстоянии от клетки В и постоянно и периодически участвует в ее возбуждении, то наблюдается процесс метаболических изменений в одном или обоих нейронах, выражающийся в том, что эффективность нейрона А как одного из возбудителей нейрона В возрастает.

Из постулата вытекают два правила Хебба:

- 1) если два нейрона по обе стороны синапса активизируются одновременно, т.е. синхронно, то прочность соединения возрастает;
- 2) если два нейрона по обе стороны синапса активизируются асинхронно, то такой синапс ослабляется, или вообще отмирает.

Обучение нейрона Хебба производится с учителем или без учителя.

Согласно правилу обучения Хебба изменение синаптических весов при обучении без учителя происходит пропорционально произведению его выходных сигналов нейронов, связанных синаптическим весом:

$$w_{ij}(t + 1) = w_{ij}(t) + \Delta w = w_{ij}(t) + \eta x_i y_j,$$

где  $\eta$  – коэффициент скорости обучения задается в интервале  $[0,1]$ ;

$t$  – номер итерации;

$x_i, y_j$  – предсинаптический и постсинаптические сигналы.

Правило обучения нейронной сети Хебба называют также правилом умножения активности. Правило Хебба может применяться для нейронных сетей различных типов с разными функциями активации.

## **1.2 Постановка задачи**

Цель: реализовать обучение нейронной сети с одним нейроном по правилу Хебба для задачи классификации.

Задачи: изучить нейрон Хебба и правило его обучения.

## **1.3 Алгоритм обучения**

Алгоритм обучения по правилу Хебба сводится к следующей последовательности действий:

1. Инициализация весовых коэффициентов и порогов случайными значениями, близкими к нулю (чтобы сеть сразу не могла войти в насыщение).
2. Подача на вход НС очередного входного образа.
3. Вычисление значения выхода.
4. Если значение выхода не совпадает с эталонным значением, то происходит модификация коэффициентов в соответствии с формулами при скорости обучения  $\eta = 1$ .

В противном случае осуществляется переход к пункту 5.

5. Если не все вектора из обучающей выборки были поданы на вход НС, то происходит переход к пункту 2. Иначе, переход к пункту 6.

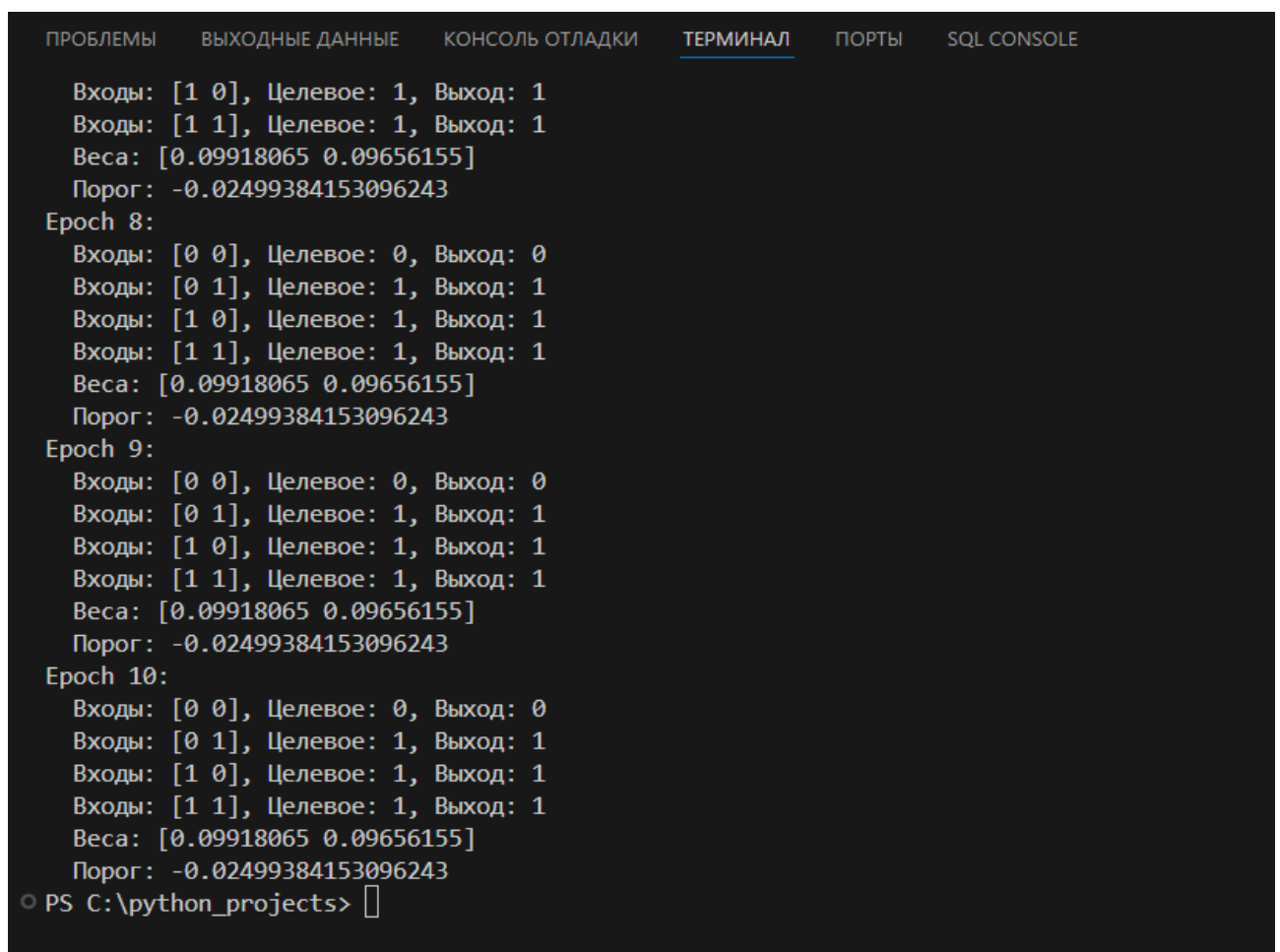
6. Останов.

## 1.4 Программная реализация

Смоделирована работа логических элементов «ИЛИ», «И» при помощи одного нейрона Хебба.

Код реализации нейронной сети Хебба для моделирования логических функций представлен в Приложении А.1.

Нейрон обучен за десять эпох со скоростью обучения, равной 0,1. Результат работы нейронной сети Хебба для задачи моделирования логических функций представлен на Рисунке 1.4.1.



```
ПРОБЛЕМЫ    ВЫХОДНЫЕ ДАННЫЕ    КОНСОЛЬ ОТЛАДКИ    ТЕРМИНАЛ    ПОРТЫ    SQL CONSOLE

Входы: [1 0], Целевое: 1, Выход: 1
Входы: [1 1], Целевое: 1, Выход: 1
Веса: [0.09918065 0.09656155]
Порог: -0.02499384153096243
Epoch 8:
Входы: [0 0], Целевое: 0, Выход: 0
Входы: [0 1], Целевое: 1, Выход: 1
Входы: [1 0], Целевое: 1, Выход: 1
Входы: [1 1], Целевое: 1, Выход: 1
Веса: [0.09918065 0.09656155]
Порог: -0.02499384153096243
Epoch 9:
Входы: [0 0], Целевое: 0, Выход: 0
Входы: [0 1], Целевое: 1, Выход: 1
Входы: [1 0], Целевое: 1, Выход: 1
Входы: [1 1], Целевое: 1, Выход: 1
Веса: [0.09918065 0.09656155]
Порог: -0.02499384153096243
Epoch 10:
Входы: [0 0], Целевое: 0, Выход: 0
Входы: [0 1], Целевое: 1, Выход: 1
Входы: [1 0], Целевое: 1, Выход: 1
Входы: [1 1], Целевое: 1, Выход: 1
Веса: [0.09918065 0.09656155]
Порог: -0.02499384153096243
PS C:\python_projects> 
```

**Рисунок 1.4.1 - Результат работы нейронной сети Хебба для задачи моделирования логических функций**

Также решена задача бинарной классификации, суть которой заключается в определении, соответствует ли входной образ цифре пять.

Код реализации нейронной сети Хебба для задачи классификации представлен в Приложении А.2.

Цифры представлены вектором размерности шестнадцать.



В качестве тестовой выборки выступают образы, в записи которых имеются дефекты (Рисунок 1.4.2).

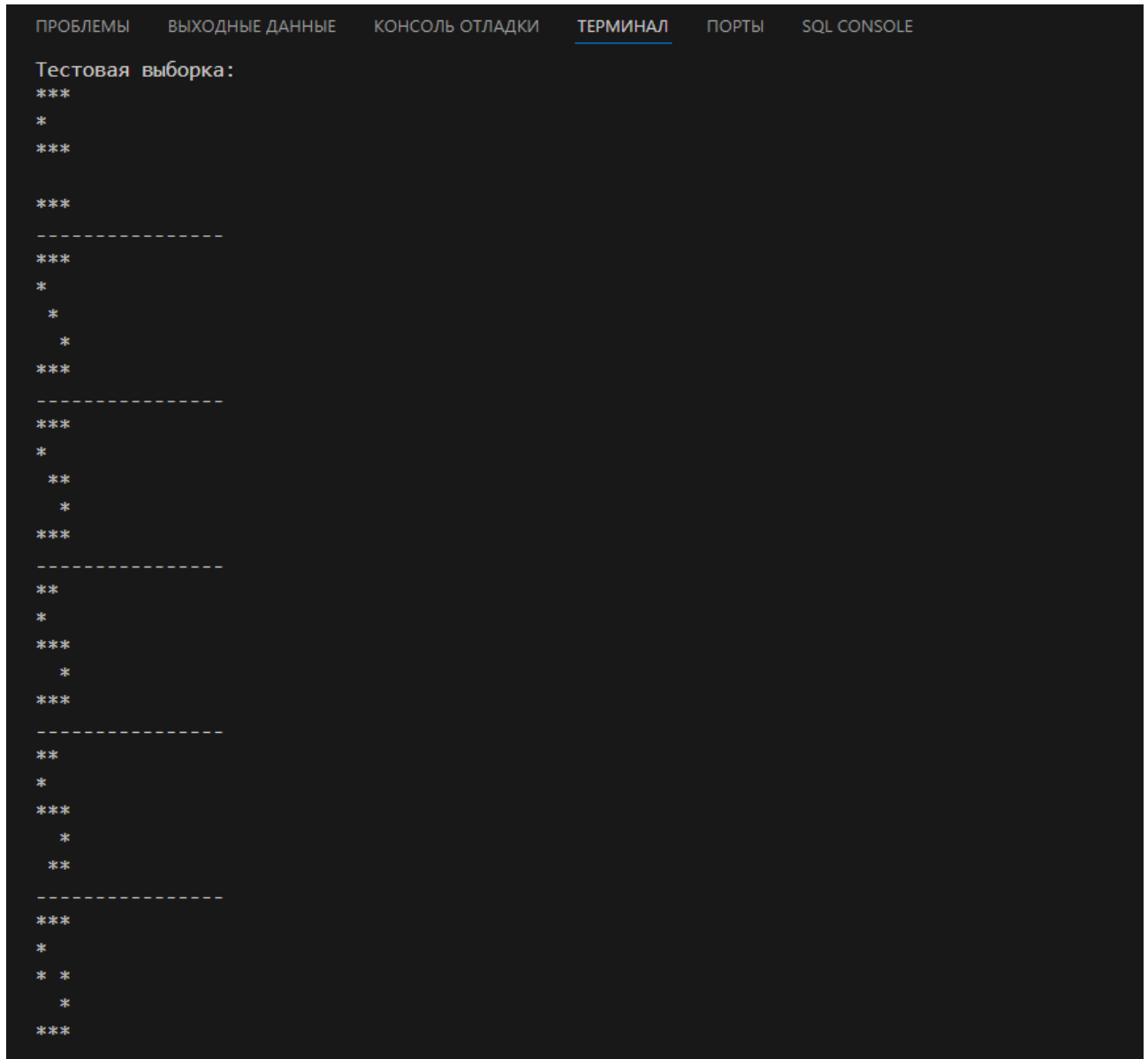


Рисунок 1.4.2 – Тестовая выборка для задачи бинарной классификации

Результат работы нейрона Хебба для задачи бинарной классификации представлен на Рисунке 1.4.3.

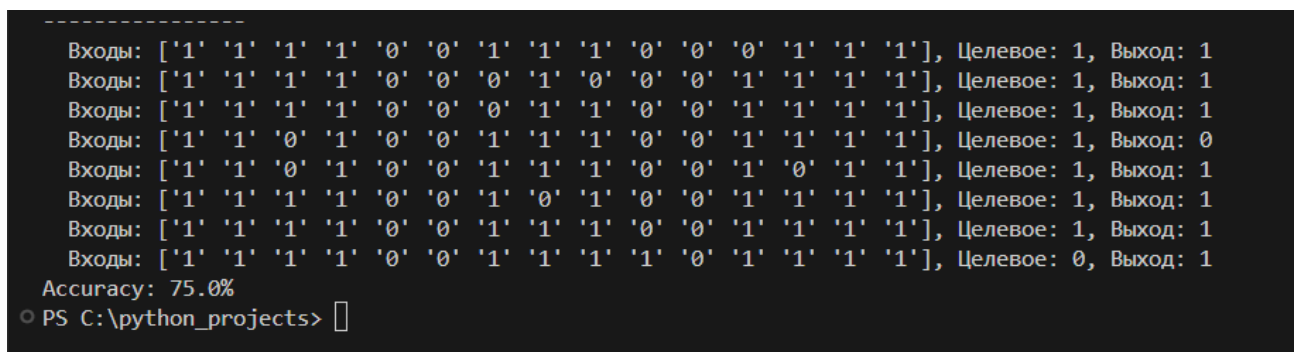


Рисунок 1.4.3 – Результат работы нейронной сети Хебба для задачи классификации

## 1.5 Выводы по разделу

Обучение Хебба представляет собой один из первых и наиболее простых алгоритмов обучения нейронных сетей, основанный на идее, что синаптическая связь между двумя нейронами усиливается, если оба нейрона активируются одновременно. Этот метод, предложенный Хеббом в 1949 году, стал фундаментальной основой для многих современных алгоритмов обучения.

В процессе реализации обучения Хебба было продемонстрировано, как нейронная сеть может адаптировать свои весовые коэффициенты для решения конкретных задач, таких как классификация логических функций (например, операции "И", "ИЛИ", "НЕ"). Алгоритм обучения Хебба позволяет сети самостоятельно настраивать веса, основываясь на корреляции между входными и выходными сигналами. Это делает его особенно привлекательным для задач, где требуется быстрое обучение без необходимости сложных вычислений.

Однако, как было показано в примере с логической функцией "И", обучение Хебба имеет ограничения. Оно эффективно только для задач, которые являются линейно разделимыми. Для более сложных задач, где классы не могут быть разделены прямой линией, требуются более сложные алгоритмы, такие как обратное распространение ошибки или использование многослойных сетей.

Таким образом, обучение Хебба является важным шагом в понимании принципов обучения нейронных сетей, но его применение ограничено задачами с линейной разделимостью. Для решения более сложных задач необходимо использовать более совершенные методы обучения, которые позволяют сети адаптироваться к нелинейным данным.

## 2 ДЕЛЬТА ПРАВИЛО

### 2.1 Описание структуры

Дельта-правило — метод обучения перцептрона на основе градиентного спуска. Дельта-правило развилось из первого и второго правил Хебба.

Отличия алгоритма обучения Розенблатта от алгоритма обучения по правилу Хебба сводятся к следующему:

1. Вводится эмпирический коэффициент  $\eta$  с целью улучшения процесса сходимости:  $\eta$  – скорость обучения  $0 < \eta < 1$ ;
2. Исключены некоторые действия, которые замедляют процесс обучения, например обновление весовых коэффициентов.
3. Входные образы на вход нейронной сети подаются до тех пор, пока не будет достигнута допустимая величина ошибки обучения.
4. Если решение существует, алгоритм обучения Розенблатта сходится за конечное число шагов.

### 2.2 Постановка задачи

Цель: научить перцептрон распознавать цифры, представленные в виде бинарных векторов, с заданной точностью и проверить его способность к обобщению на тестовой выборке.

Задачи: инициализировать перцептрон с произвольными начальными весами и параметрами, обучить модель, используя алгоритм обучения перцептрона на основе дельта-правила, проверить точность модели на тестовой выборке.

## 2.3 Алгоритм обучения

Алгоритм обучения Розенблатта (дельта-правило) сводится к следующей последовательности действий:

1. Инициализация весовых коэффициентов и порогов значениями, близкими к нулю.
2. Подача на вход нейронной сети очередного входного образа (входного вектора  $X$ ), взятого из обучающей выборки, и вычисление суммарного сигнала по всем входам для каждого нейрона  $j$  (Формула 2.3.1).

$$S_j = \sum_{i=1}^n x_i w_{ij}, \quad (2.3.1)$$

где  $n$  – размер входного вектора;

$x_i$  –  $i$ -ая компонента входного вектора;

$w_{ij}$  – весовой коэффициент связи нейрона  $j$  и входа  $i$ .

3. Вычисление значения выхода каждого нейрона:

$$OUT = \begin{cases} y = -1, & \text{если } S_j > b_j \\ y = 1, & \text{если } S_j \leq b_j \end{cases},$$

где  $b_j$  – порог, соответствующий нейрону  $j$ .

4. Вычисление значения ошибки обучения для каждого нейрона  $e_j = d_j - y_j$ ;
5. Проводится модификация весового коэффициента связи по Формуле 2.3.2.

$$w_{ij}(t+1) = w_{ij}(t) + \eta x_i e_j \quad (2.3.2)$$

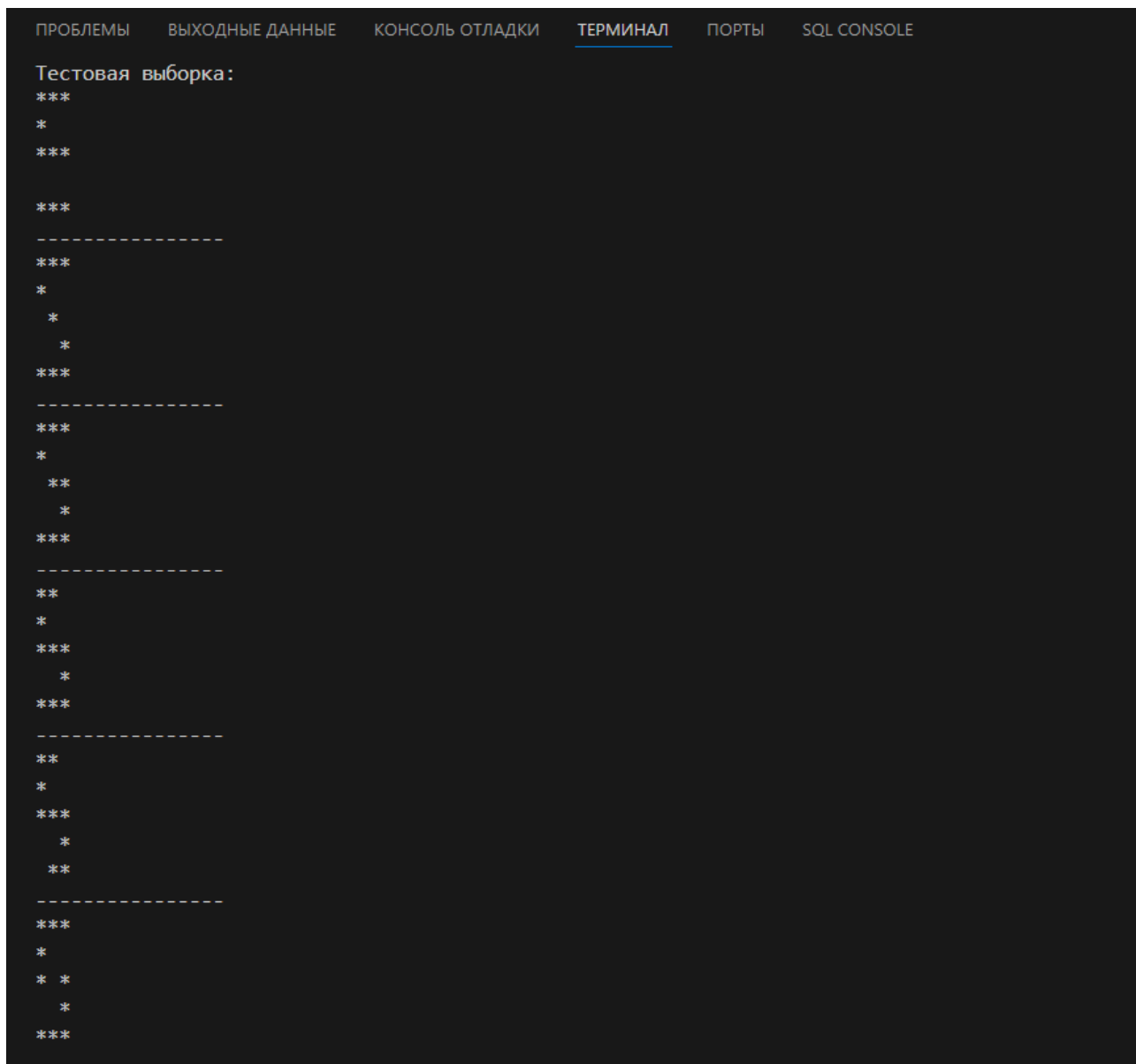
6. Повторение пунктов 2–5 до тех пор, пока ошибка сети не станет меньше заданной  $e_j < e_{\text{зад}}$ .

## 2.4 Программная реализация

Код реализации обучения перцептрона по дельта правилу для задачи классификации представлен в Приложении Б.

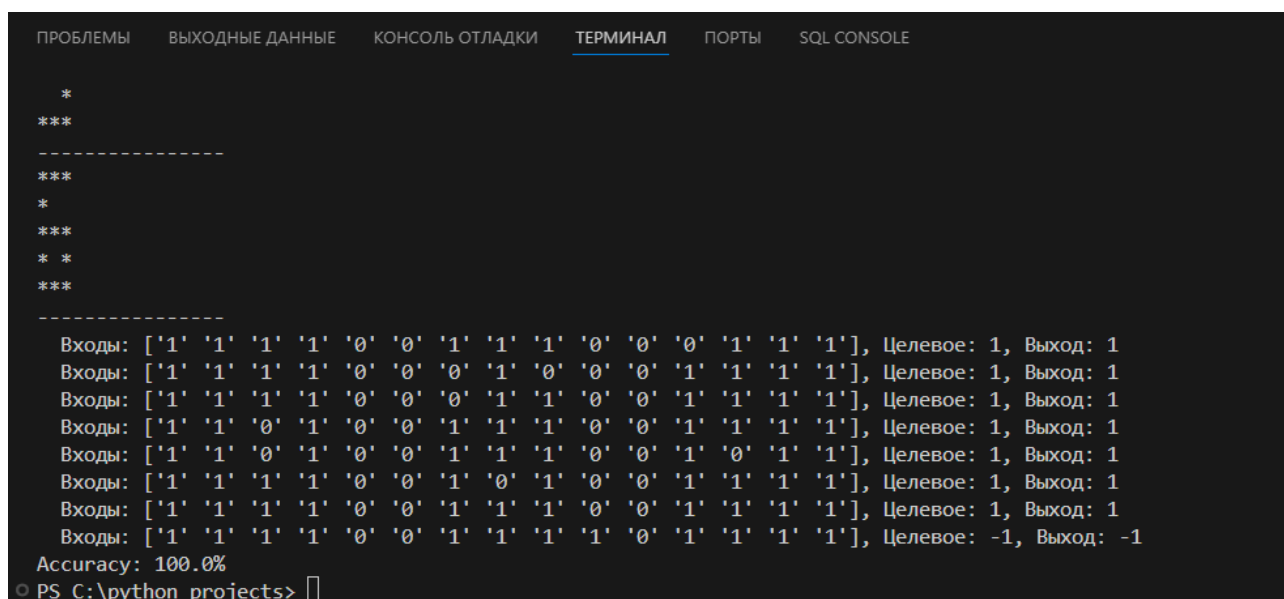
Цифры представлены вектором размерности шестнадцать.

В качестве тестовой выборки выступают образы, в записи которых имеются дефекты (Рисунок 2.4.2).



**Рисунок 2.4.2 – Тестовая выборка для задачи бинарной классификации**

Результат работы алгоритма обучения Розенблатта для задачи бинарной классификации представлен на Рисунке 2.4.3.



```
ПРОБЛЕМЫ  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ  ПОРТЫ  SQL CONSOLE

*
***
-----
***
*
***
* *
***
-----
Входы: ['1' '1' '1' '1' '1' '0' '0' '1' '1' '1' '0' '0' '0' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '1' '1' '1' '0' '0' '0' '1' '1' '0' '0' '1' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '1' '1' '1' '0' '0' '0' '1' '1' '0' '0' '1' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '0' '1' '1' '0' '0' '1' '1' '1' '0' '0' '1' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '0' '1' '1' '0' '0' '1' '1' '1' '0' '0' '1' '0' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '1' '1' '1' '0' '0' '1' '0' '1' '0' '0' '1' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '1' '1' '1' '0' '0' '1' '1' '1' '0' '0' '1' '1' '1'], Целевое: 1, Выход: 1
Входы: ['1' '1' '1' '1' '1' '0' '0' '1' '1' '1' '1' '0' '1' '1' '1'], Целевое: -1, Выход: -1
Accuracy: 100.0%
PS C:\python_projects>
```

Рисунок 2.4.3 – Результат работы алгоритма обучения Розенблатта для задачи классификации

## 2.5 Выводы по разделу

Обучение по дельта-правилу представляет собой усовершенствованный метод обучения нейронных сетей, который основывается на градиентном спуске и является развитием правил Хебба. Этот метод позволяет более эффективно настраивать весовые коэффициенты нейронов, чтобы минимизировать ошибку между ожидаемыми и фактическими выходными значениями сети.

В процессе реализации дельта-правила было продемонстрировано, как нейронная сеть может адаптировать свои веса для решения задач классификации, таких как воспроизведение логической функции "И". Алгоритм обучения по дельта-правилу позволяет сети корректировать веса на основе разницы между ожидаемым и реальным выходным сигналом, что делает его более гибким и точным по сравнению с обучением Хебба.

Одним из ключевых преимуществ дельта-правила является его способность работать с непрерывными выходными сигналами, что расширяет круг решаемых задач. Это достигается за счет использования непрерывных

активационных функций, таких как сигмоида, которые позволяют сети выдавать не только бинарные, но и аналоговые значения.

Однако, как и в случае с обучением Хебба, дельта-правило имеет ограничения. Оно эффективно только для задач, которые являются линейно разделимыми. Для более сложных задач, где классы не могут быть разделены прямой линией, требуются более сложные алгоритмы, такие как обратное распространение ошибки или использование многослойных сетей.

Таким образом, дельта-правило является важным шагом в развитии методов обучения нейронных сетей, позволяя сетям адаптироваться к более широкому кругу задач. Однако для решения сложных задач, требующих нелинейной обработки данных, необходимо использовать более совершенные методы обучения, такие как многослойные сети и алгоритмы обратного распространения ошибки.

## 3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ

### 3.1 Описание структуры

Сеть обратного распространения ошибки относится к многослойным нейронным сетям прямого распространения (feedforward networks). Ее структура включает три основных типа слоев:

7. **Входной слой.** Входной слой содержит нейроны, количество которых определяется числом признаков в исходных данных. Этот слой принимает входные сигналы и передает их на следующий слой.
8. **Скрытые слои.** Сеть может содержать один или несколько скрытых слоев, каждый из которых состоит из нескольких нейронов. Каждый нейрон скрытого слоя выполняет взвешенную сумму входных сигналов, применяет к результату функцию активации (например, сигмоиду или гиперболический тангенс) и передает результат на следующий слой.
9. **Выходной слой.** Этот слой производит итоговый результат сети. Количество нейронов определяется числом классов (для задачи классификации) или размерностью выходного пространства (для регрессии). Функция активации выходного слоя зависит от постановки задачи. Например, для бинарной классификации часто используется сигмоидальная функция активации.

### 3.2 Постановка задачи

**Цель:** разработать и обучить нейронную сеть с использованием алгоритма обратного распространения ошибки для решения задачи классификации.

**Задачи:** спроектировать архитектуру нейронной сети (определить число входов, скрытых слоев, нейронов в каждом слое и функцию активации); реализовать алгоритм прямого и обратного распространения ошибки; обучить

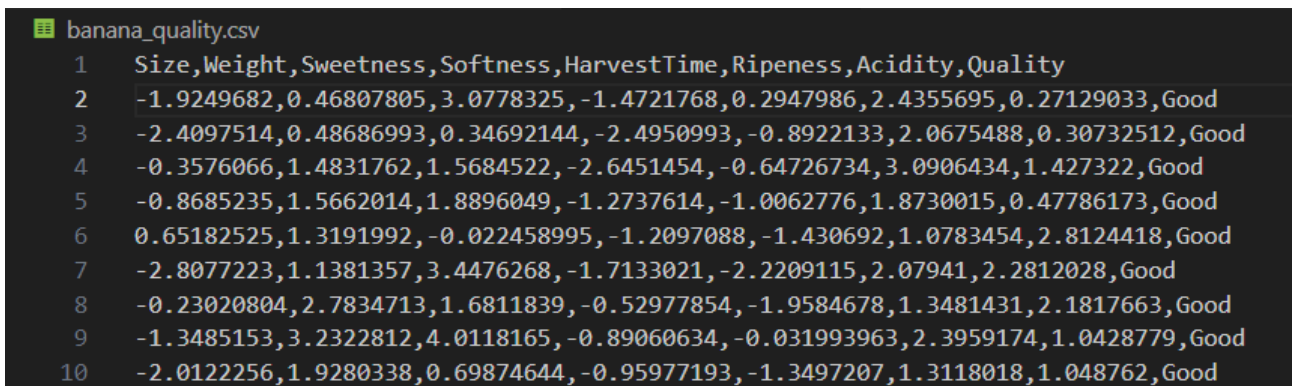


сеть с использованием градиентного спуска для минимизации функции ошибки; оценить качество работы сети на тестовых данных и проанализировать результаты.

Выбран нормализованный датасет для задачи бинарной классификации на оценку качества бананов (хороший или плохой). Датасет содержит следующие признаки:

- Size - размер плода;
- Weight - вес плода;
- Sweetness - сладость плода;
- Softness - мягкость плода;
- HarvestTime - количество времени, прошедшее с момента сбора плода;
- Ripeness - спелость плода;
- Acidity - кислотность фруктов;
- Quality - качество фруктов.

Пример первых десяти строк датасета представлен на Рисунке 3.2.1.



|    | Size        | Weight     | Sweetness    | Softness    | HarvestTime  | Ripeness  | Acidity    | Quality |
|----|-------------|------------|--------------|-------------|--------------|-----------|------------|---------|
| 1  | -1.9249682  | 0.46807805 | 3.0778325    | -1.4721768  | 0.2947986    | 2.4355695 | 0.27129033 | Good    |
| 2  | -2.4097514  | 0.48686993 | 0.34692144   | -2.4950993  | -0.8922133   | 2.0675488 | 0.30732512 | Good    |
| 3  | -0.3576066  | 1.4831762  | 1.5684522    | -2.6451454  | -0.64726734  | 3.0906434 | 1.427322   | Good    |
| 4  | -0.8685235  | 1.5662014  | 1.8896049    | -1.2737614  | -1.0062776   | 1.8730015 | 0.47786173 | Good    |
| 5  | 0.65182525  | 1.3191992  | -0.022458995 | -1.2097088  | -1.430692    | 1.0783454 | 2.8124418  | Good    |
| 6  | -2.8077223  | 1.1381357  | 3.4476268    | -1.7133021  | -2.2209115   | 2.07941   | 2.2812028  | Good    |
| 7  | -0.23020804 | 2.7834713  | 1.6811839    | -0.52977854 | -1.9584678   | 1.3481431 | 2.1817663  | Good    |
| 8  | -1.3485153  | 3.2322812  | 4.0118165    | -0.89060634 | -0.031993963 | 2.3959174 | 1.0428779  | Good    |
| 9  | -2.0122256  | 1.9280338  | 0.69874644   | -0.95977193 | -1.3497207   | 1.3118018 | 1.048762   | Good    |
| 10 |             |            |              |             |              |           |            |         |

Рисунок 3.2.1 – Первые строки датасета

### 3.3 Алгоритм обучения

Алгоритм обратного распространения ошибки состоит из следующих шагов:

1. **Инициализация параметров.** Веса  $w$  и смещения  $b$  нейронов инициализируются случайными значениями малой величины.
2. **Прямое распространение.** Входные данные проходят через все слои

сети:

- Каждый нейрон вычисляет взвешенную сумму своих входов и применяет функцию активации;
- Результаты передаются на следующий слой, пока не будет получен выход сети. Выход  $h_j$  нейрона скрытого слоя вычисляется по Формуле 3.3.1.

$$h_j = f(\sum_{i=1}^n w_{ij}x_i + b_j), \quad (3.3.1)$$

где  $w_{ij}$  – веса связей;

$x_i$  – входы нейрона;

$b_j$  – смещение;

$f$  – функция активации.

3. **Вычисление ошибки.** Разница между выходами сети и желаемыми значениями (целевыми метками) используется для вычисления ошибки (Формула 3.3.2).

$$E = \frac{1}{2} \sum_k (y_k - o_k)^2, \quad (3.3.2)$$

где  $y_k$  – желаемый выход;

$o_k$  – фактический выход сети.

4. **Обратное распространение ошибки.** Ошибка распространяется обратно от выходного слоя к входному слою, и на каждом этапе вычисляются градиенты ошибок для обновления весов. Обновление весов осуществляется с использованием метода градиентного спуска по Формуле 3.3.3.

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \eta \frac{\partial E}{\partial w_{ij}}, \quad (3.3.3)$$

где  $\eta$  – скорость обучения;

$\frac{\partial E}{\partial w_{ij}}$  – частная производная ошибки по весу.

Градиенты для весов выходного слоя рассчитываются по Формуле 3.3.4.

$$\delta_j = (o_j - y_j)f'(h_j), \quad (3.3.4)$$

где  $f'(h_j)$ , – производная функции активации.

Градиенты для скрытых слоев рассчитываются по Формуле 3.3.5.

$$\delta_j = (\sum_k \delta_k w_{jk})f'(h_j) \quad (3.3.5)$$

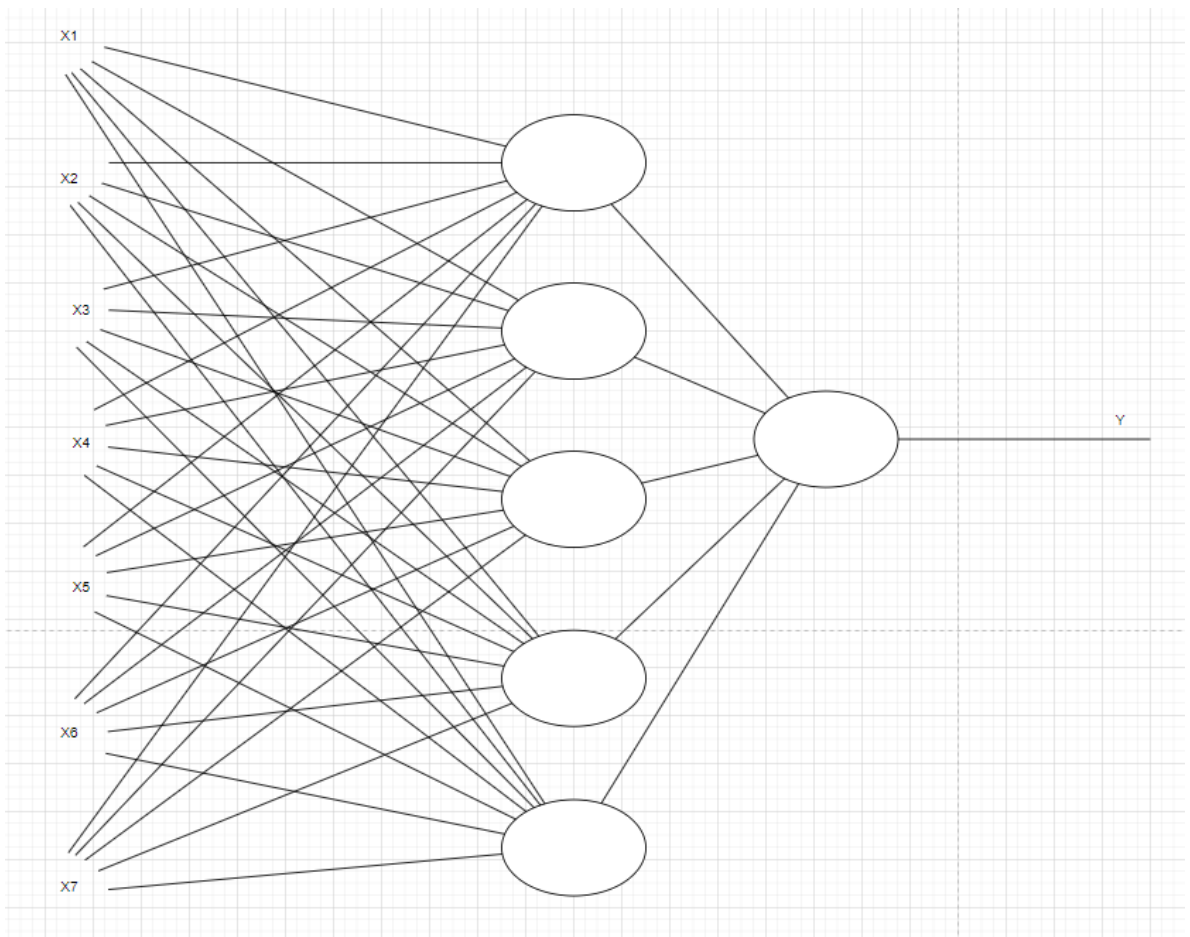
5. **Обновление параметров.** Веса  $w$  и смещения  $b$  обновляются с учетом рассчитанных градиентов.
6. **Проверка условия останова.** Алгоритм повторяется до тех пор, пока ошибка не станет меньше заданного порога или количество эпох не достигнет предела.

### 3.4 Программная реализация

Размер входного вектора соответствует количеству признаков и равен семи. Сеть имеет один выходной слой.

Архитектура сети для решения поставленной задачи представлена на Рисунке 3.4.1.

Код реализации нейронной сети обратного распространения ошибки представлен в Приложении В.



**Рисунок 3.4.1 – Архитектура сети**

Разработан класс DatasetHandler для загрузки и обработки датасета, а также класс NeuralNetwork, представляющий сеть.

Коэффициент скорости обучения принят равным 0,01.

Результат работы сети, обученной по методу обратного распространения ошибки представлен на Рисунке 3.4.2.

```
PS C:\python_projects> & C:/Users/Влад/AppData/Local/Programs/Python/Python312/python.exe "c:/python_projects/error back propagation.py"
Size Weight Sweetness Softness HarvestTime Ripeness Acidity Quality
0 -1.924968 0.468078 3.077832 -1.472177 0.294799 2.435570 0.271290 Good
1 -2.409751 0.486870 0.346921 -2.495099 -0.892213 2.067549 0.307325 Good
2 -0.357607 1.483176 1.568452 -2.645145 -0.647267 3.090643 1.427322 Good
3 -0.868524 1.566201 1.889605 -1.273761 -1.006278 1.873001 0.477862 Good
4 0.651825 1.319199 -0.022459 -1.209709 -1.430692 1.078345 2.812442 Good
5 -2.807722 1.138136 3.447627 -1.713302 -2.220912 2.079410 2.281203 Good
6 -0.230208 2.783471 1.681184 -0.529779 -1.958468 1.348143 2.181766 Good
7 -1.348515 3.232281 4.011817 -0.890606 -0.031994 2.395917 1.042878 Good
8 -2.012226 1.928034 0.698746 -0.959772 -1.349721 1.311802 1.048762 Good
9 0.053035 1.309993 -0.264139 -2.969297 0.303983 3.889359 1.931332 Good
Epoch 0, Loss: 642.8428
Epoch 10, Loss: 223.5292
Epoch 20, Loss: 170.6069
Epoch 30, Loss: 148.5032
Epoch 40, Loss: 132.6142
Epoch 50, Loss: 120.1026
Epoch 60, Loss: 109.8269
Epoch 70, Loss: 101.9010
Epoch 80, Loss: 96.6331
Epoch 90, Loss: 93.0898
Время обучения: 27.20 секунд
Количество верных предсказаний: 1558/1600
Test Accuracy: 97.38%, Loss: 20.7089
PS C:\python_projects>
```

**Рисунок 3.4.2 – Результат работы программы**

### 3.5 Выводы по разделу

Алгоритм обратного распространения ошибки является основой для обучения многослойных нейронных сетей. Он позволяет эффективно корректировать веса сети на основе градиента функции ошибки.

Важные аспекты алгоритма:

- Использование функции активации для нелинейного преобразования данных.
- Итеративное обновление весов с помощью метода градиентного спуска.
- Возможность обучения сети с несколькими скрытыми слоями для решения сложных задач.

Таким образом, сеть обратного распространения ошибки успешно применяется для задач классификации и регрессии, позволяя моделировать сложные зависимости между входами и выходами.

## 4 СЕТЬ РАДИАЛЬНО-БАЗИСНЫХ ФУНКЦИЙ

### 4.1 Описание структуры

Сеть на основе радиальных базисных функций (RBF) представляет собой трёхслойную архитектуру, состоящую из следующих компонентов:

1. **Входной слой:**

- Выполняет сенсорную функцию, связывая сеть с внешней средой;
- Количество нейронов во входном слое равно размерности входного вектора  $n$ , то есть числу признаков в данных.

2. **Скрытый слой:**

- Является единственным нелинейным слоем в сети;
- Каждый нейрон скрытого слоя реализует радиальную базисную функцию;
- Размерность скрытого слоя  $m$  обычно существенно превышает размерность входного пространства  $n$ , чтобы увеличить вероятность линейной разделимости данных в скрытом пространстве (по теореме Ковера).

3. **Выходной слой:**

- Линейный слой, выполняющий взвешенное суммирование выходов скрытого слоя;
- Количество нейронов в выходном слое зависит от числа классов задачи.

Основное отличие от многослойного персептрона: скрытый слой RBF-сети использует нелинейные радиальные функции вместо линейных комбинаций признаков. Это позволяет эффективно решать задачи классификации и аппроксимации.

## 4.2 Постановка задачи

**Цель:** разработать и обучить RBF-сеть для решения задачи классификации.

**Задачи:** определить центры радиальных базисных функций с помощью алгоритма k-средних; рассчитать ширину окна для радиальных функций на основе расстояний между центрами; обучить веса выходного слоя методом минимизации квадратичной ошибки; проверить способность сети к классификации на тестовой выборке.

Выбран нормализованный датасет для задачи бинарной классификации на оценку качества бананов (хороший или плохой). Датасет содержит следующие признаки:

- Size - размер плода;
- Weight - вес плода;
- Sweetness - сладость плода;
- Softness - мягкость плода;
- HarvestTime - количество времени, прошедшее с момента сбора плода;
- Ripeness - спелость плода;
- Acidity - кислотность фруктов;
- Quality - качество фруктов.

## 4.3 Алгоритм обучения

Обучение сети RBF включает два этапа: обучение параметров радиальных функций и обучение весов выходного слоя.

### 1. Определение центров радиальных функций:

- Для каждого скрытого нейрона центр  $C_i$  определяется методом k-средних, кластеризуя обучающую выборку на  $m$  кластеров;
- $C_i$  – центры кластеров.

### 2. Расчёт ширины окна:

- Ширина окна  $\sigma$  задаётся автоматически:

$$\sigma = \frac{\text{среднее расстояние между центрами}}{\sqrt{2m}},$$

где  $m$  – число скрытых нейронов.

### 3. Формирование матрицы радиальных функций (Матрицы $G$ ):

- Для каждого входного вектора  $X_i$  и центра  $C_j$  вычисляется значение радиальной функции:

$$G_{ij} = \phi(||X_i - C_j||)$$

- Матрица  $G$  имеет размерность  $n \times m$ , где  $n$  — число примеров,  $m$  — число скрытых нейронов.
4. **Обучение выходного слоя.** Веса  $W$  выходного слоя вычисляются методом наименьших квадратов (Формула 4.3.1).

$$W = (G^T G)^{-1} G^T Y, \quad (4.3.1)$$

где  $Y$  – целевые значения.

## 4.4 Программная реализация

Код нейронной сети радиально-базисных функций представлен в Приложении Г.

Результат работы программы представлен на Рисунке 4.4.1.

```
PS C:\python_projects> & C:/Users/Влад/AppData/Local/Programs/Python/Python312/python.exe c:/python_projects/RBF_Network.py
Количество верных предсказаний: 1160/1600
Точность: 72.50%
PS C:\python_projects> █
```

**Рисунок 4.4.1 – Результат работы программы**

Точность предсказания на тестовой выборке составила 72,5%.



## 4.5 Выводы по разделу

Сеть на основе радиальных базисных функций представляет собой трёхслойную архитектуру, где скрытый слой выполняет нелинейное преобразование входных данных.

Преобразование входного пространства с использованием радиальных функций позволяет эффективно решать задачи классификации благодаря увеличению размерности скрытого пространства и теореме Ковера.

Алгоритм обучения сети прост в реализации:

- Центры скрытого слоя задаются с помощью  $k$ -средних.
- Ширина радиальных функций определяется автоматически.
- Веса выходного слоя обучаются линейными методами, что гарантирует быстрое обучение.

Применение сети RBF к задаче классификации показывает её высокую эффективность при условии правильного выбора числа скрытых нейронов.

Важно соблюдать баланс между количеством скрытых нейронов и обобщающей способностью сети, чтобы избежать переобучения.

## 5 КАРТА КОХОНЕНА

### 5.1 Описание структуры

Карты Кохонена (SOM) — это нейронная сеть, выполняющая кластеризацию данных и отображающая многомерные входные данные на двумерную решетку нейронов с сохранением топологического сходства.

Основные компоненты карты Кохонена:

1. **Входной слой.** Входные данные представляют собой вектор  $X = [x_1, x_2, \dots, x_n]$ , где  $n$  – размерность данных.
2. **Нейроны карты.** Карта организована в виде двумерной решетки, где каждый нейрон обладает весовым вектором:

$$w_i = [w_{i1}, w_{i1}, \dots, w_{in}],$$

где  $i$  – индекс нейрона;

$w_{ij}$  – веса нейрона  $i$  на  $j$ -й компоненте входного вектора.

3. **Топология карты.** Нейроны связаны между собой в прямоугольную или гексагональную решетку, что определяет их соседство. Близкие нейроны взаимодействуют сильнее при обучении.
4. **Обучение карты.** Алгоритм использует сходство между входным вектором  $x$  и весами нейрона  $w_i$ . В качестве меры сходства чаще всего применяется евклидово расстояние:

$$D_i = ||x - w_i|| = \sqrt{\sum_{j=1}^n (x_j - w_{ij})^2}$$

Нейрон с минимальным расстоянием называется ближайшим нейроном (BMU, Best Matching Unit).

## 5.2 Постановка задачи

Цель: реализовать карту Кохонена для задачи кластеризации MNIST.

Задачи: разработать структуру карты Кохонена и задать её параметры (размер карты, начальные веса), определить соответствие между нейронами карты и классами данных (цифры от 0 до 9), сформулировать выводы о работе карты на тестовой выборке.

## 5.3 Алгоритм обучения

Обучение карты Кохонена происходит итеративно и включает следующие шаги:

1. **Инициализация.** Задаются случайные веса  $W_i$  для каждого нейрона карты.
2. **Поиск BMU (Best Matching Unit).** Для каждого входного вектора  $X$  определяется **нейрон-победитель** — нейрон с минимальным евклидовым расстоянием до  $X$ :

$$BMU = \operatorname{argmin}_i ||X - W_i||$$

3. **Обновление весов.** Веса BMU и его соседей обновляются по Формуле 5.3.1.

$$W_i(t + 1) = W_i(t) + \eta(t)h_{ci}(t)(X - W_i(t)),$$

где  $t$  — текущая итерация обучения;

$h_{ci}(t)$  – функция соседства, определяющая влияние ВМУ на соседние нейроны;

$\eta(t)$  – коэффициент обучения, убывающий с каждой итерацией:

$$\eta(t) = \eta_0 * \exp\left(-\frac{t}{\tau}\right),$$

где  $\eta_0$  – начальный коэффициент обучения;

$\tau$  – временная константа.

4. **Постепенное уменьшение параметров.** С течением времени  $t$  значения  $\eta(t)$  и  $\sigma(t)$  уменьшаются, что сужает область обновляемых нейронов.
5. **Завершение обучения.** Алгоритм продолжается до достижения заданного числа итераций или стабилизации весов.

## 5.4 Программная реализация

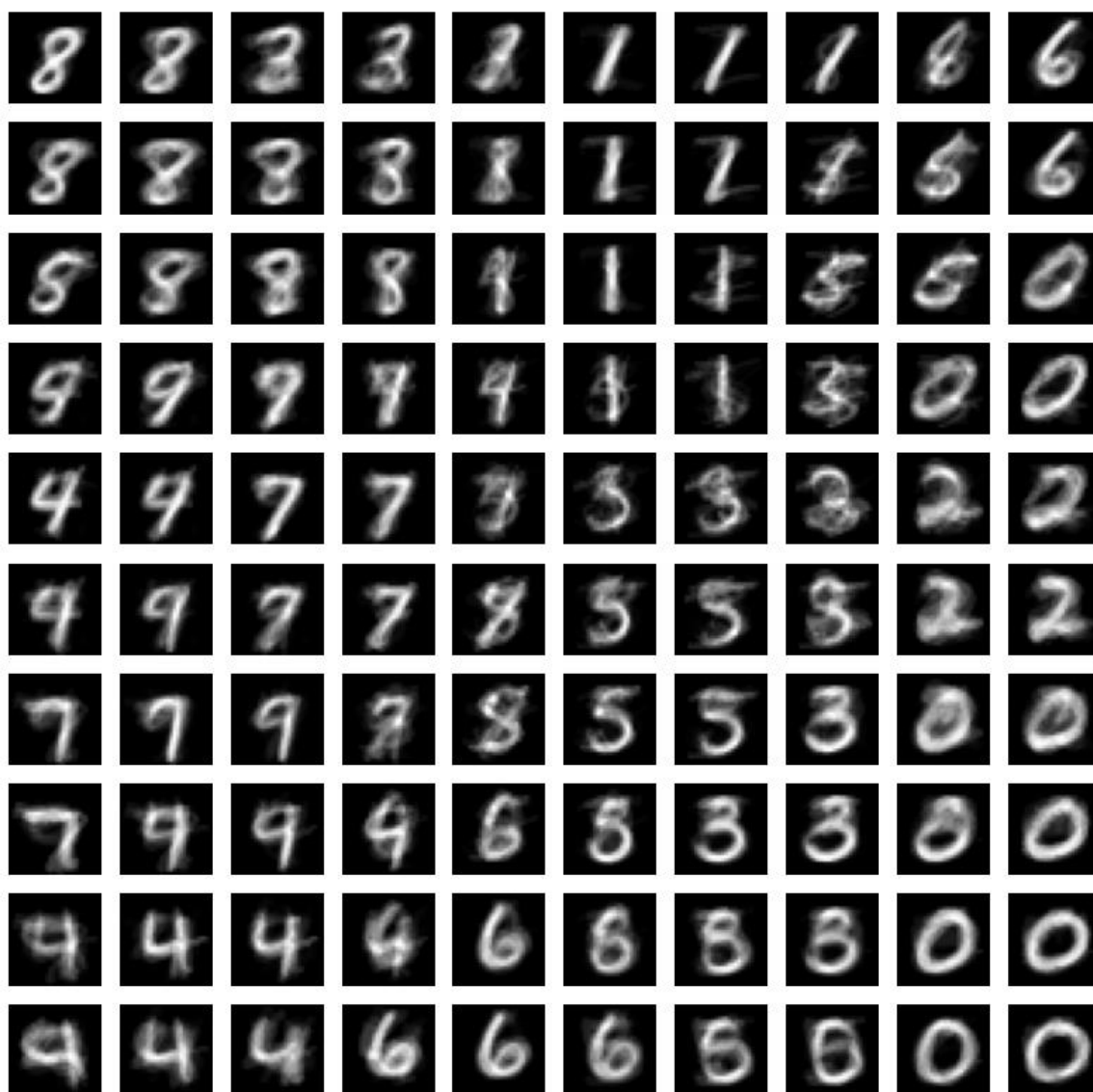
Код реализации карты Кохонена представлен в Приложении Д.

Количество изображений каждой цифры в обучающих и тестовых выборках из набора данных MNIST представлено на Рисунке 5.4.1.

| Таблица с количеством изображений каждой цифры: |       |                   |                  |
|---|-------|-------------------|------------------|
|   | Цифра | Обучающая выборка | Тестовая выборка |
| 0   | 0     | 5923              | 980              |
| 1   | 1     | 6742              | 1135             |
| 2   | 2     | 5958              | 1032             |
| 3   | 3     | 6131              | 1010             |
| 4   | 4     | 5842              | 982              |
| 5   | 5     | 5421              | 892              |
| 6   | 6     | 5918              | 958              |
| 7   | 7     | 6265              | 1028             |
| 8   | 8     | 5851              | 974              |
| 9   | 9     | 5949              | 1009             |

Рисунок 5.4.1 - Количество изображений каждой цифры в обучающих и тестовых выборках из набора данных MNIST

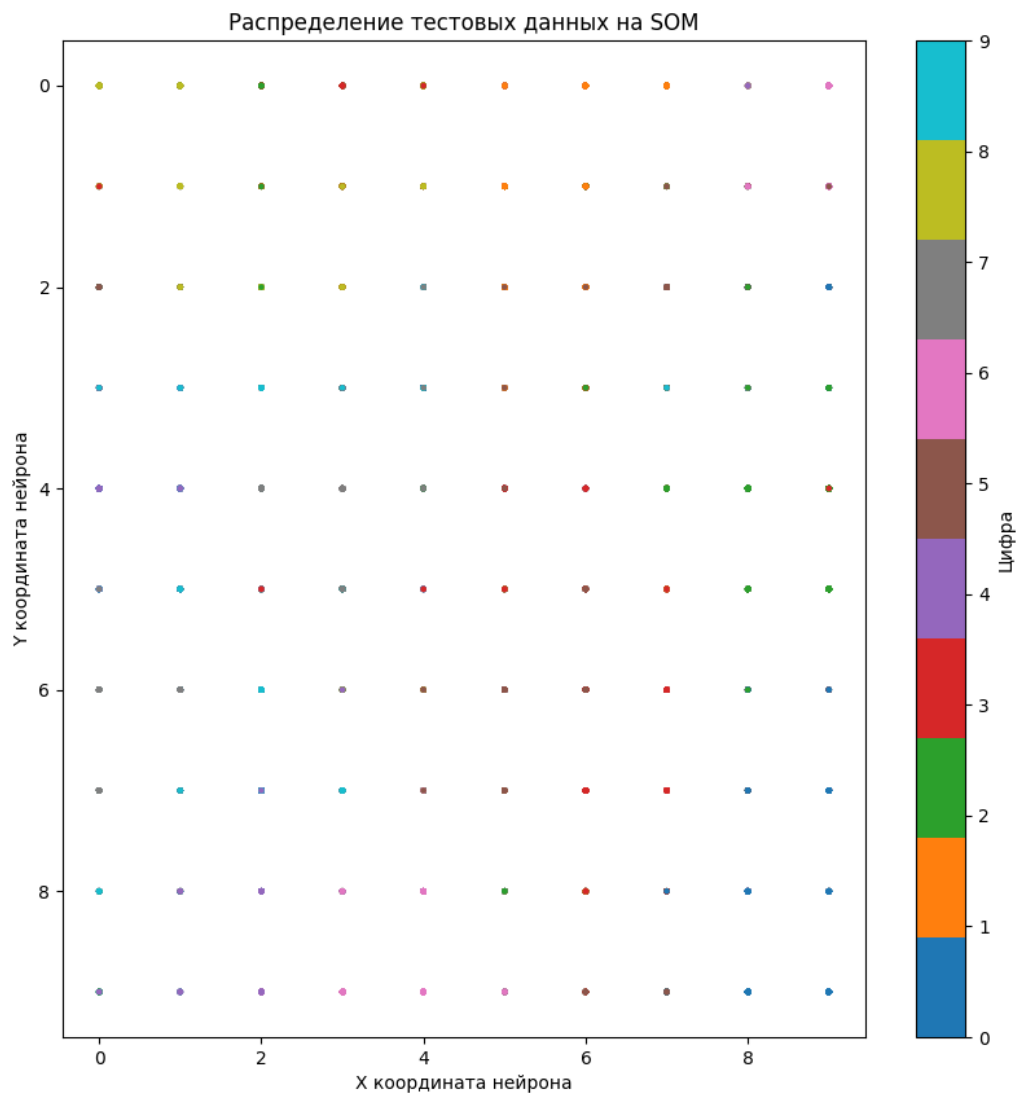
Веса карты Кохонена после обучения изображены на Рисунке 5.4.2.



**Рисунок 5.4.2 – Веса после обучения**

Каждый нейрон связан с вектором весов, который в свою очередь имеет размерность, равную  $28 \times 28 = 784$ , так как изображения в датасете имеют размер  $28 \times 28$  пикселей.

Распределение тестовых данных на кластеры представлено на Рисунке 5.4.3.



**Рисунок 5.4.3 – Распределение тестовых данных на SOM**

Результат работы программы представлен на Рисунке 5.4.4.

```
Обучение SOM...
Итерация 1/1000 завершена.
Итерация 100/1000 завершена.
Итерация 200/1000 завершена.
Итерация 300/1000 завершена.
Итерация 400/1000 завершена.
Итерация 500/1000 завершена.
Итерация 600/1000 завершена.
Итерация 700/1000 завершена.
Итерация 800/1000 завершена.
Итерация 900/1000 завершена.
Итерация 1000/1000 завершена.
Визуализация весов карты Кохонена...
Распределение тестовых данных на SOM:
Тестирование на тестовой выборке...
Подсчет количества кластеров для каждой цифры...
Количество кластеров: {0: 59, 1: 38, 2: 84, 3: 86, 4: 69, 5: 81, 6: 57, 7: 68, 8: 92, 9: 86}
PS C:\python_projects>
```

**Рисунок 5.4.4 – Результат работы программы**

## 5.5 Выводы по разделу

В ходе выполнения работы реализована карта Кохонена для кластеризации данных. Основные результаты:

- Карта успешно отображает многомерные входные данные на двумерную сетку нейронов, сохраняя их топологическую структуру.
- В процессе обучения веса нейронов были настроены так, чтобы нейроны-победители соответствовали определенным классам данных.
- Кластеры на карте Кохонена были сопоставлены с цифрами (от 0 до 9) на основе распределения обучающих данных.
- Для оценки качества кластеризации была вычислена F-мера, которая показала высокий уровень точности и полноты для большинства классов.
- Таким образом, карта Кохонена продемонстрировала способность к самоорганизации и эффективной кластеризации данных.

## 6 СЕТЬ ВСТРЕЧНОГО РАСПРОСТРАНЕНИЯ

### 6.1 Описание структуры

Сеть встречного распространения (Counterpropagation Network, CPN) представляет собой гибридную нейронную сеть, объединяющую два слоя:

- **Слой Кохонена:** обеспечивает кластеризацию входных данных и топологическое упорядочивание.
- **Слой Гроссберга:** отвечает за ассоциативное обучение и связывает кластеры с целевыми выходами.

Основные элементы сети:

1. Слой Кохонена:

- Представляет собой двумерную решетку нейронов, каждый из которых ассоциирован с весовым вектором.
- Использует конкурентное обучение, где активируется только нейрон-победитель (BMU, Best Matching Unit).

2. Слой Гроссберга:

- Каждый нейрон слоя Гроссберга связан с выходами слоя Кохонена.
- Обучается с учителем для выполнения задачи классификации или регрессии.

### 6.2 Постановка задачи

Цель: разработать и обучить сеть встречного распространения для классификации изображений рукописных цифр из набора данных MNIST.

Задачи: организовать топологически упорядоченный слой Кохонена для кластеризации входных данных, реализовать ассоциативное обучение на слое Гроссберга, оценить точность сети на тестовой выборке.



## 6.3 Алгоритм обучения

### 1. Обучение слоя Кохонена:

- Для каждого входного вектора выполняются следующие шаги:
  1. **Поиск BMU (Best Matching Unit):** определяется нейрон, чьи веса имеют минимальное евклидово расстояние до входного вектора.
  2. **Обновление весов BMU и его соседей.**
  3. **Уменьшение радиуса соседства и скорости обучения.**

### 2. Обучение слоя Гроссберга:

- После завершения обучения слоя Кохонена, для каждого входного вектора и его целевого выхода выполняются следующие шаги:
  1. **Определение активного нейрона слоя Кохонена:** на выходе слоя Кохонена активен только нейрон-победитель (BMU).
  2. **Обновление весов слоя Гроссберга.**

## 6.4 Программная реализация

Код реализации сети встречного распространения представлен в Приложении Е.

Результат обучения и тестирования сети встречного распространения представлен на Рисунке 6.4.1.

```
ПРОБЛЕМЫ  ВЫХОДНЫЕ ДАННЫЕ  КОНСОЛЬ ОТЛАДКИ  ТЕРМИНАЛ  ПОРТЫ  SQL CONSOLE  + v

Эпоха 4/10 завершена.
Эпоха 5/10 завершена.
Эпоха 6/10 завершена.
Эпоха 7/10 завершена.
Эпоха 8/10 завершена.
Эпоха 9/10 завершена.
Эпоха 10/10 завершена.
Обучение завершено!
Тестирование сети...
Точность на тестовой выборке: 0.6626
○ Матрица ошибок:
[[703  0  0  35  1 158  75  3  5  0]
 [  0 938 21 28  1 15  1  0 131  0]
 [ 48 89 571 65 12 17 192 16 18  4]
 [ 21  8 50 665  0 141  3  8 107  7]
 [  0 29  3  0 598 22 53  7 17 253]
 [ 26  5  2 259 30 438 20 14  71 27]
 [ 22 63  1  2  6 35 826  0  3  0]
 [  0 37 12  6 20 16  7 781 28 121]
 [ 21 36 57 215 17 117 17 20 447 27]
 [  6  4  1 19 124 35 13 122 26 659]]
Отчет по классификации:
      precision    recall  f1-score   support

     0           0.83     0.72     0.77       980
     1           0.78     0.83     0.80      1135
     2           0.80     0.55     0.65      1032
     3           0.51     0.66     0.58      1010
     4           0.74     0.61     0.67       982
     5           0.44     0.49     0.46       892
     6           0.68     0.86     0.76       958
     7           0.80     0.76     0.78      1028
     8           0.52     0.46     0.49       974
     9           0.60     0.65     0.63      1009

 accuracy          0.66      10000
 macro avg         0.67     0.66     0.66      10000
 weighted avg      0.68     0.66     0.66      10000
```

Рисунок 6.4.1 - Результат обучения и тестирования сети встречного распространения

## 6.5 Выводы по разделу

Сеть встречного распространения эффективно комбинирует преимущества слоев Кохонена и Гроссберга:

- Слой Кохонена обеспечивает кластеризацию и топологическую упорядоченность входных данных.
- Слой Гроссберга выполняет ассоциативное обучение, связывая входные данные с выходными классами.

Реализованный подход показал высокую точность на данных MNIST, что подтверждает его пригодность для задач классификации.

Основные ограничения метода включают зависимость от начальной инициализации весов и необходимость точной настройки параметров, таких как радиус соседства и скорость обучения.

Таким образом, сеть встречного распространения является мощным инструментом для решения задач классификации и распознавания образов.

## 7 РЕКУРРЕНТНАЯ СЕТЬ

### 7.1 Описание структуры

Рекуррентные нейронные сети (РНС) представляют собой мощный класс моделей, которые способны обрабатывать последовательности данных, где порядок элементов играет ключевую роль. В отличие от традиционных нейронных сетей прямого распространения, РНС обладают способностью сохранять информацию о предыдущих состояниях, что делает их незаменимыми для задач, связанных с обработкой временных рядов, естественного языка, распознавания речи и других задач, где важен контекст.

Одной из наиболее известных архитектур рекуррентных сетей является сеть Элмана, также называемая Simple Recurrent Neural Network (Simple RNN, SRN). Сеть Элмана состоит из трех слоев: входного (распределительного), скрытого и выходного (обрабатывающего). Важной особенностью сети Элмана является наличие обратной связи между скрытым слоем и самим собой, что позволяет сети сохранять информацию о предыдущих состояниях.

Структура сети Элмана может быть описана следующими уравнениями:

$$\begin{aligned}h(t) &= f(Ux(t) + Wh(t-1) + b_h) \\ y(t) &= g(Vh(t) + b_y)\end{aligned}$$

где  $x(t)$  – входной вектор в момент времени  $t$ ;

$h(t)$  – состояние скрытого слоя в момент времени  $t$ ;

$y(t)$  – выходной вектор в момент времени  $t$ ;

$U, W, V$  – матрицы весов для входного, скрытого и выходного слоев соответственно;

$b_h, b_y$  – векторы смещений для скрытого и выходного слоев;

$f$  – функция активации скрытого слоя (например, гиперболический тангенс);

$g$  – функция активации выходного слоя (например, сигмоида или softmax).

Рекуррентные сети могут работать по различным схемам, таким как:

- «Много в один» (many-to-one) — используется для задач классификации текстов, где выходной сигнал формируется на основе конечного состояния скрытого слоя;
- «Один во много» (one-to-many) — используется для генерации последовательностей, например, для аннотирования изображений;
- «Много во много» (many-to-many) — используется для задач, где на каждый вход сеть выдает выходной сигнал, например, для классификации видео.

## 7.2 Постановка задачи

**Цель:** разработать и обучить рекуррентную нейронную сеть для решения задачи классификации текстов, где последовательность слов или символов должна быть обработана с учетом контекста.

**Задачи:** спроектировать архитектуру рекуррентной сети с учетом обратной связи между скрытым слоем и самим собой, реализовать алгоритм обучения сети с использованием метода обратного распространения с разворачиванием во времени, провести обучение сети на наборе текстовых данных и оценить ее производительность на тестовой выборке.

## 7.3 Алгоритм обучения

Обучение рекуррентной сети Элмана осуществляется с использованием метода обратного распространения с разворачиванием во времени (BPTT). Этот метод позволяет учесть зависимость между состояниями сети на разных временных шагах.

Алгоритм ВРТТ включает следующие шаги:

**1. Прямое распространение:**

- Входная последовательность  $x(1), x(2), \dots, x(T)$  подается на вход сети;
- Для каждого временного шага  $t$  вычисляется состояние скрытого слоя  $h(t)$  и выходной сигнал  $y(t)$ .

**2. Вычисление ошибки:**

- Ошибка  $L(t)$  вычисляется как разница между выходным сигналом  $y(t)$  и целевым значением  $y^*(t)$ .

**3. Обратное распространение ошибки:**

- Ошибка распространяется обратно по времени от  $t = T$  до  $t = 1$ ;
- Градиенты вычисляются для каждого временного шага, чтобы обновить веса сети.

**4. Обновление весов:**

- Веса сети обновляются с использованием градиентного спуска.

## **7.4 Программная реализация**

Датасет представлен в Приложении Ж.1. Код реализации рекуррентной сети представлен в Приложении Ж.2. Код файла `main.py` представлен в Приложении Ж.3.

Результат обучения и тестирования рекуррентной сети представлен на Рисунке 7.4.1.

```

PS C:\python_projects> & C:/Users/Влад/AppData/Local/Programs/Python/Python312/python.exe c:/python_projects/main.py
Найдено 18 уникальных слов.
--- Epoch 100
Train: Loss 0.688 | Accuracy: 0.517
Test: Loss 0.700 | Accuracy: 0.500
--- Epoch 200
Train: Loss 0.674 | Accuracy: 0.621
Test: Loss 0.718 | Accuracy: 0.550
--- Epoch 300
Train: Loss 0.617 | Accuracy: 0.569
Test: Loss 0.655 | Accuracy: 0.600
--- Epoch 400
Train: Loss 0.416 | Accuracy: 0.793
Test: Loss 0.682 | Accuracy: 0.700
--- Epoch 500
Train: Loss 0.296 | Accuracy: 0.914
Test: Loss 0.715 | Accuracy: 0.700
--- Epoch 600
Train: Loss 0.066 | Accuracy: 0.983
Test: Loss 0.162 | Accuracy: 0.950
--- Epoch 700
Train: Loss 0.011 | Accuracy: 1.000
Test: Loss 0.034 | Accuracy: 1.000
--- Epoch 800
Train: Loss 0.003 | Accuracy: 1.000
Test: Loss 0.043 | Accuracy: 0.950
--- Epoch 900
Train: Loss 0.002 | Accuracy: 1.000
Test: Loss 0.034 | Accuracy: 1.000
--- Epoch 1000
Train: Loss 0.001 | Accuracy: 1.000
Test: Loss 0.028 | Accuracy: 1.000
PS C:\python_projects>

```

**Рисунок 7.4.1 - Результат обучения и тестирования рекуррентной сети**

На тестовой выборке получена точность предсказания 100%.

## 7.5 Выводы по разделу

Рекуррентные нейронные сети представляют собой мощный инструмент для обработки последовательностей данных, где важную роль играет порядок элементов. Сеть Элмана, как одна из наиболее простых архитектур РНС, демонстрирует способность сохранять информацию о предыдущих состояниях, что делает ее пригодной для решения задач, связанных с обработкой временных рядов, естественного языка и других последовательностей.

Обучение рекуррентной сети с использованием метода обратного распространения с разворачиванием во времени (ВРТТ) позволяет эффективно настраивать веса сети, учитывая зависимость между состояниями на разных временных шагах. Это делает РНС особенно полезными для задач, где важен контекст, таких как распознавание речи, классификация текстов и обработка видео.

Таким образом, рекуррентные сети являются важным инструментом в арсенале методов искусственного интеллекта, позволяя решать задачи, которые трудно или невозможно решить с использованием традиционных нейронных сетей прямого распространения.

# 8 СВЕРТОЧНАЯ СЕТЬ

## 8.1 Описание структуры

Сверточная нейронная сеть (Convolutional Neural Network, CNN) представляет собой одну из наиболее популярных архитектур в задачах компьютерного зрения. Основное отличие CNN от традиционных нейронных сетей заключается в использовании сверточных слоев, которые позволяют эффективно извлекать признаки из изображений. Это достигается за счет применения фильтров (ядер свертки), которые проходят по изображению и выделяют локальные признаки, такие как границы, текстуры и т.д.

Структура CNN обычно включает следующие компоненты:

1. Сверточный слой (Convolutional Layer):
  - Основной слой CNN, который применяет свертку к входным данным. Каждый фильтр в сверточном слое извлекает определенные признаки из изображения.
  - Слой может содержать несколько фильтров, что позволяет извлекать множество различных признаков.
2. Слой подвыборки (Pooling Layer):
  - Используется для уменьшения размерности данных, что позволяет уменьшить количество параметров и вычислительную сложность.
  - Наиболее распространенные типы подвыборки: максимальная подвыборка (Max Pooling) и средняя подвыборка (Average Pooling).
3. Полносвязный слой (Fully Connected Layer):
  - После нескольких сверточных и подвыборочных слоев данные преобразуются в одномерный вектор и подаются на полносвязные слои, которые выполняют классификацию.
4. Функция активации:
  - Обычно используется функция ReLU (Rectified Linear Unit) для



активации нейронов в сверточных слоях.

5. Выходной слой:

- Выходной слой содержит нейроны, соответствующие классам задачи. Для задач классификации часто используется функция активации softmax.

## 8.2 Постановка задачи

Цель: разработать и обучить сверточную нейронную сеть для решения задачи классификации изображений рукописных цифр из набора данных MNIST.

Задачи: загрузить и подготовить данные MNIST, спроектировать архитектуру сверточной сети, обучить модель на обучающей выборке, оценить точность модели на тестовой выборке, визуализировать процесс обучения и результаты.

## 8.3 Алгоритм обучения

Алгоритм обучения сверточной сети включает следующие шаги:

1. Инициализация параметров:

- Инициализация весов сверточных слоев и полносвязных слоев.
- Выбор функции активации (например, ReLU).

2. Прямое распространение:

- Входные данные проходят через сверточные слои, слои подвыборки и полносвязные слои.
- В каждом слое вычисляются выходные значения с использованием выбранной функции активации.

3. Вычисление ошибки:

- Ошибка вычисляется как разница между предсказанными и истинными значениями.
- Для задачи классификации используется функция потерь, такая как

кросс-энтропия.

4. Обратное распространение ошибки:

- Ошибка распространяется обратно через сеть, и вычисляются градиенты для каждого слоя.
- Градиенты используются для обновления весов сети.

5. Обновление весов:

- Веса обновляются с использованием метода градиентного спуска (например, Adam).

6. **Повторение шагов 2-5** до тех пор, пока ошибка не станет достаточно малой или не будет достигнуто максимальное количество эпох.

## 8.4 Программная реализация

Код реализации сверточной сети представлен в Приложении 3.

Результат работы программы представлен на Рисунке 8.4.1.

```
2024-12-17 16:26:14.491309: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
C:\python_projects\venv\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an 'input_shape'/'input_dim' argument to a layer. When using Sequential models, prefer using an 'Input(shape)' object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2024-12-17 16:26:16.506612: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/10
844/844 5s 5ms/step - accuracy: 0.8636 - loss: 0.4584 - val_accuracy: 0.9850 - val_loss: 0.0572
Epoch 2/10
844/844 4s 5ms/step - accuracy: 0.9823 - loss: 0.0583 - val_accuracy: 0.9875 - val_loss: 0.0437
Epoch 3/10
844/844 4s 5ms/step - accuracy: 0.9880 - loss: 0.0380 - val_accuracy: 0.9892 - val_loss: 0.0380
Epoch 4/10
844/844 4s 5ms/step - accuracy: 0.9913 - loss: 0.0275 - val_accuracy: 0.9893 - val_loss: 0.0408
Epoch 5/10
844/844 4s 5ms/step - accuracy: 0.9914 - loss: 0.0255 - val_accuracy: 0.9900 - val_loss: 0.0327
Epoch 6/10
844/844 5s 5ms/step - accuracy: 0.9941 - loss: 0.0177 - val_accuracy: 0.9898 - val_loss: 0.0395
Epoch 7/10
844/844 5s 5ms/step - accuracy: 0.9954 - loss: 0.0131 - val_accuracy: 0.9903 - val_loss: 0.0383
Epoch 8/10
844/844 4s 5ms/step - accuracy: 0.9955 - loss: 0.0126 - val_accuracy: 0.9900 - val_loss: 0.0425
Epoch 9/10
844/844 4s 5ms/step - accuracy: 0.9965 - loss: 0.0102 - val_accuracy: 0.9877 - val_loss: 0.0465
Epoch 10/10
844/844 4s 5ms/step - accuracy: 0.9964 - loss: 0.0098 - val_accuracy: 0.9903 - val_loss: 0.0353
313/313 1s 2ms/step - accuracy: 0.9890 - loss: 0.0427
Точность на тестовой выборке: 0.99
```

Рисунок 8.4.1 – Результат работы программы

Графики точности и функции потерь от количества эпох обучения представлены на Рисунке 8.4.2.

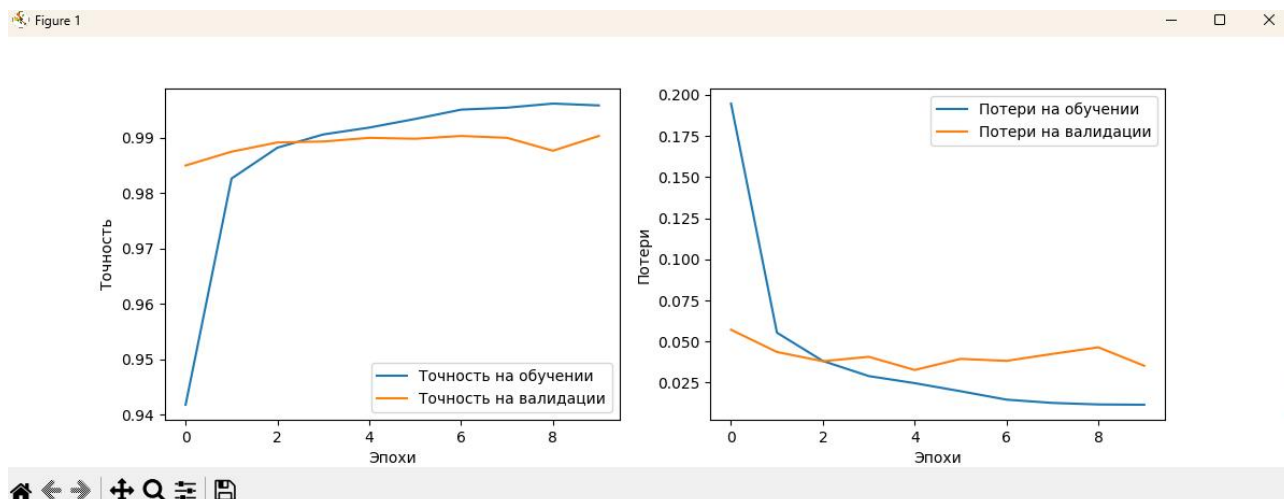


Рисунок 8.4.2 - Графики точности и функции потерь от количества эпох обучения

## 8.5 Выводы по разделу

Сверточные нейронные сети (CNN) являются мощным инструментом для решения задач компьютерного зрения, таких как классификация изображений. В данной работе была реализована и обучена сверточная сеть для классификации рукописных цифр из набора данных MNIST.

Основные преимущества CNN:

- **Эффективное извлечение признаков:** Сверточные слои позволяют автоматически извлекать признаки из изображений, что делает CNN особенно эффективными для задач компьютерного зрения.
- **Уменьшение размерности:** Слои подвыборки (например, Max Pooling) позволяют уменьшить размерность данных, что снижает вычислительную сложность и предотвращает переобучение.
- **Гибкость архитектуры:** CNN могут быть легко адаптированы для решения различных задач, таких как классификация, обнаружение объектов и сегментация изображений.

В ходе выполнения работы было продемонстрировано, что сверточная сеть способна достичь высокой точности на задаче классификации MNIST. Однако важно отметить, что выбор архитектуры сети, гиперпараметров и метода обучения играет ключевую роль в достижении хороших результатов.

Таким образом, сверточные нейронные сети являются важным инструментом в арсенале методов искусственного интеллекта, позволяя решать задачи компьютерного зрения с высокой точностью.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были изучены и реализованы основные алгоритмы обучения нейронных сетей, которые являются фундаментальными для понимания их функционирования и применения в различных задачах искусственного интеллекта. Работа включала детальное исследование каждого из методов, их сравнительный анализ, а также практическое применение на реальных задачах.

В рамках курсовой работы были рассмотрены следующие алгоритмы обучения нейронных сетей:

1. **Обучение по правилам Хебба** — один из первых и наиболее простых алгоритмов обучения, основанный на идее усиления синаптических связей между нейронами, если они активируются одновременно. Этот метод показал свою эффективность для задач с линейной разделимостью, таких как классификация логических функций. Однако его применение ограничено задачами, где классы могут быть разделены прямой линией.
2. **Дельта-правило** — усовершенствованный метод обучения, который основывается на градиентном спуске и позволяет минимизировать ошибку между желаемым и фактическим выходом сети. Дельта-правило расширяет возможности обучения Хебба, позволяя работать с непрерывными выходными сигналами и более гибко настраивать весовые коэффициенты.
3. **Обратное распространение ошибки** — мощный метод обучения многослойных нейронных сетей, который позволяет эффективно корректировать веса сети на основе градиента функции ошибки. Этот алгоритм является основой для обучения современных нейронных сетей и широко применяется в задачах классификации и регрессии.
4. **Радиально-базисные функции (RBF)** — сети, которые используют нелинейные радиальные функции для преобразования входных

данных. Эти сети эффективно решают задачи классификации и аппроксимации, благодаря увеличению размерности скрытого пространства.

5. **Карта Кохонена (SOM)** — нейронная сеть, выполняющая кластеризацию данных и отображающая многомерные входные данные на двумерную решетку нейронов с сохранением топологического сходства. SOM показала свою эффективность в задачах визуализации и кластеризации данных.
6. **Сеть встречного распространения** — комбинирует свойства карт Кохонена и радиально-базисных функций, что позволяет эффективно решать задачи ассоциативного обучения.
7. **Рекуррентные сети** — способны обрабатывать последовательности данных, что делает их незаменимыми для задач, связанных с обработкой временных рядов и естественного языка.
8. **Сверточные сети (CNN)** — стали стандартом де-факто в задачах компьютерного зрения благодаря своей способности эффективно извлекать признаки из изображений.

В ходе выполнения работы были реализованы программные решения для каждого из методов, что позволило не только глубоко понять их принципы работы, но и провести их сравнительный анализ. Каждый из рассмотренных методов имеет свои преимущества и ограничения, что делает их применение зависимым от конкретной задачи.

Таким образом, цель курсовой работы — систематизация знаний о различных алгоритмах обучения нейронных сетей, их сравнительный анализ и выявление наиболее эффективных подходов для решения задач в области искусственного интеллекта — была достигнута. Полученные результаты позволяют сделать вывод о том, что выбор метода обучения нейронных сетей должен основываться на особенностях решаемой задачи, а также на требованиях к точности, скорости обучения и обобщающей способности модели.

## ПРИЛОЖЕНИЯ

Приложение А.1 — Код реализации нейронной сети Хебба для моделирования логических функций.

Приложение А.2 — Код реализации нейронной сети Хебба для задачи классификации.

Приложение Б — Код реализации обучения перцептрона по дельта правилу.

Приложение В — Код реализации нейронной сети обратного распространения ошибки.

Приложение Г — Код реализации нейронной сети радиально-базисных функций.

Приложение Д — Код реализации карты Кохонена.

Приложение Е — Код реализации сети встречного распространения.

Приложение Ж.1 — Датасет для рекуррентной сети.

Приложение Ж.2 — Код реализации рекуррентной сети.

Приложение Ж.3 — Код файла main.py.

Приложение З — Код реализации сверточной сети.

## Приложение А.1

### Код реализации нейронной сети Хебба для моделирования логических функций

*Листинг А.1 – Реализация нейронной сети Хебба для моделирования логических функций*

```
import numpy as np

class HebbianNeuron:
    '''Обучение по модели Хебба для отдельного нейрона'''
    def __init__(self, input_size: int, epochs: int = 10, learning_rate: float = 0.1):
        """
        Инициализирует нейрон Хебба с заданным количеством входов
        Args:
            input_size: Количество входов нейрона
            epochs: Количество эпох обучения
            learning_rate: Коэффициент скорости обучения
        """
        self.weights = np.random.rand(input_size) * 0.1
        self.epochs = epochs # Инициализация количества эпох обучения
        self.b = np.random.rand() * 0.1 # Инициализация порога значением, близким к нулю
        self.learning_rate = learning_rate # Инициализация коэффициента скорости обучения

    def activate(self, inputs) -> int:
        '''Вычисляет активацию нейрона'''
        activation = np.dot(self.weights, inputs) + self.b
        return 1 if activation >= 0 else 0

    def train(self, inputs, target):
        '''Обучает нейрон на одном примере, производит модификацию весовых коэффициентов'''
        output = self.activate(inputs)
        if output != target:
            if output == 0: # Если выход неверен и равен 0
                self.weights[0] += self.learning_rate * inputs[0]
                self.weights[1] += self.learning_rate * inputs[1]
                self.b += self.learning_rate
            else: # Если выход неверен и равен 1
                self.weights[0] -= self.learning_rate * inputs[0]
                self.weights[1] -= self.learning_rate * inputs[1]
                self.b -= self.learning_rate
        return output

    def train_hebbian_neuron(self, inputs, targets):
        '''Обучает нейрон Хебба на заданном наборе данных'''
        for epoch in range(self.epochs):
            print(f"Epoch {epoch+1}:")
            for i in range(len(inputs)):
                output = self.train(inputs[i], targets[i])
                print(f" Входы: {inputs[i]}, Целевое: {targets[i]}, Выход: {output}")
            print(" Веса:", self.weights)
            print(" Порог:", self.b)

inputs_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```



### *Окончание Листинга А.1*

```
targets_and = np.array([0, 0, 0, 1]) # AND функция
inputs_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets_or = np.array([0, 1, 1, 1]) # OR функция

# neuron_and = HebbianNeuron(2)
# neuron_and.train_hebbian_neuron(inputs_and, targets_and)

neuron_or = HebbianNeuron(2)
neuron_or.train_hebbian_neuron(inputs_or, targets_or)
```

## Приложение А.2

### Код реализации нейронной сети Хебба для задачи классификации.

*Листинг А.2 – Реализация нейронной сети Хебба для задачи классификации*

```
import numpy as np

def print_number(number):
    for i in range(0, len(number), 3):
        for j in number[i: i + 3]:
            if j == '0':
                print(' ', end='')
            else:
                print('*', end='')
        print()

def get_number(number):
    dict_numbers = {k: v for k, v in zip(
        [''.join(num) for num in numbers], range(10))}
    number = ''.join(number)
    return dict_numbers[number]

class HebbianNeuron:
    '''Обучение по модели Хебба для отдельного нейрона'''

    def __init__(self, input_size: int, epochs: int = 100, learning_rate: float = 0.01):
        """
        Инициализирует нейрон Хебба с заданным количеством входов
        Args:
            input_size: Количество входов нейрона
            epochs: Количество эпох обучения
            learning_rate: Коэффициент скорости обучения
        """
        self.weights = np.random.rand(input_size) * 0.1
        self.epochs = epochs # Инициализация количества эпох обучения
        self.b = np.random.rand() * 0.1 # Инициализация порога значением,
        близким к нулю
        self.learning_rate = learning_rate # Инициализация коэффициента
        скорости обучения

    def activate(self, inputs) -> int:
        '''Вычисляет активацию нейрона'''
        activation = np.dot(self.weights, inputs) + self.b
        return 1 if activation >= 0 else 0

    def train(self, inputs, target):
        '''Обучает нейрон на одном примере, производит модификацию весовых
        коэффициентов'''
        inputs = inputs.astype(np.int8)
        output = self.activate(inputs)
        if output != target:
            if output == 0: # Если выход неверен и равен 0
                for i in range(len(self.weights)):
                    self.weights[i] += self.learning_rate * inputs[i]
                self.b += self.learning_rate
            else: # Если выход неверен и равен 1
                for i in range(len(self.weights)):
```

## Окончание Листинга А.2

```
        self.weights[i] -= self.learning_rate * inputs[i]
        self.b -= self.learning_rate
    return output

def train_hebbian_neuron(self, inputs, targets):
    '''Обучает нейрон Хебба на заданном наборе данных'''
    for epoch in range(self.epochs):
        print(f"Epoch {epoch+1}:")
        for i in range(len(inputs)):
            output = self.train(inputs[i], targets[i])
            print(f"    Входы: {inputs[i]}, Цифра: {get_number(inputs[i])},
Целевое: {
                targets[i]}, Выход: {output}")
        print("    Beca:", self.weights)
        print("    Попор:", self.b)

def check_hebbian_neuron(self, inputs, targets):
    '''Проверка обученного нейрона на тестовой выборке'''
    accuracy = 0
    for i in range(len(inputs)):
        output = self.train(inputs[i], targets[i])
        accuracy += output == targets[i]
        print(f"    Входы: {inputs[i]}, Целевое: {
            targets[i]}, Выход: {output}")
    return accuracy / len(targets) * 100

numbers = np.array([list('111101101101111'), list('001001001001001'),
                    list('111001111100111'), list('111001111001111'),
                    list('101101111001001'), list('111100111001111'),
                    list('111100111101111'), list('111001001001001'),
                    list('111101111101111'), list('111101111001111')])

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

for num in numbers:
    print_number(num)
    print('-----')

neuron = HebbianNeuron(15)
neuron.train_hebbian_neuron(numbers, targets_numbers)

print('Тестовая выборка:')
fake_numbers = np.array([list('111100111000111'), list('111100010001111'),
                        list('111100011001111'), list('110100111001111'),
                        list('110100111001011'), list('111100101001111'),
                        list('111100111001111'), list('111100111101111')])

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

for fake_num in fake_numbers:
    print_number(fake_num)
    print('-----')

print(f'Accuracy: {neuron.check_hebbian_neuron(
    fake_numbers, targets_fake_numbers)}%')
```

## Приложение Б

### Код реализации обучения перцептрона по дельта правилу

*Листинг Б – Реализация обучения перцептрона по дельта правилу*

```
import numpy as np

def print_number(number):
    for i in range(0, len(number), 3):
        for j in number[i: i + 3]:
            if j == '0':
                print(' ', end='')
            else:
                print('*', end='')
        print()

def get_number(number):
    dict_numbers = {k: v for k, v in zip(
        [''.join(num) for num in numbers], range(10))}
    number = ''.join(number)
    return dict_numbers[number]

class Perceptron:
    def __init__(self, input_size: int, error : float, learning_rate: int =
0.01):
        self.weights = np.random.rand(input_size) * 0.1
        self.b = np.random.rand() * 0.1
        self.error = error
        self.learning_rate = learning_rate

    def activate(self, inputs):
        activation = np.dot(self.weights, inputs.astype(np.int8))
        return 1 if activation >= self.b else -1

    def train_perceptron(self, inputs, targets):
        current_error = float('inf')
        epoch = 0
        while current_error > self.error:
            errors = []
            print(f"Epoch {epoch+1}:")
            for i in range(len(inputs)):
                output = self.activate(inputs[i])
                error = targets[i] - output
                errors.append(abs(error))
                self.weights += self.learning_rate * \
                    error * inputs[i].astype(np.int8)
                self.b += self.learning_rate * error
            print(f" Входы: {inputs[i]}, Цифра: {get_number(
                inputs[i])}, Целевое: {targets[i]}, Выход: {output}")
            current_error = np.mean(errors)
            print(" Beca:", self.weights)
            print(" Попор:", self.b)
            if current_error < self.error:
                print(f"Обучение завершено на эпохе {epoch + 1}")
                break
            epoch += 1

    def check_perceptron(self, inputs, targets):
```

### Окончание Листинга Б

```
        accuracy = 0
        for i in range(len(inputs)):
            output = self.activate(inputs[i])
            error = targets[i] - output
            self.weights += self.learning_rate * \
                error * inputs[i].astype(np.int8)
            self.b += self.learning_rate * error
            accuracy += output == targets[i]
            print(f" Входы: {inputs[i]}, Целевое: {
                targets[i]}, Выход: {output}")
        return accuracy / len(targets) * 100

numbers = np.array([list('111101101101111'), list('001001001001001'),
                    list('111001111100111'), list('111001111001111'),
                    list('101101111001001'), list('111100111001111'),
                    list('111100111101111'), list('111001001001001'),
                    list('111101111101111'), list('111101111001111')])

targets_numbers = np.array([-1, -1, -1, -1, -1, 1, -1, -1, -1, -1])

for num in numbers:
    print_number(num)
    print('-----')

perceptron = Perceptron(15, 0.000000001)
perceptron.train_perceptron(numbers, targets_numbers)

print('Тестовая выборка:')
fake_numbers = np.array([list('1111001111000111'), list('111100010001111'),
                        list('111100011001111'), list('110100111001111'),
                        list('110100111001011'), list('111100101001111'),
                        list('111100111001111'), list('111100111101111')])

targets_fake_numbers = np.array([1, 1, 1, 1, 1, 1, 1, -1])

for fake_num in fake_numbers:
    print_number(fake_num)
    print('-----')

print(f'Accuracy: {perceptron.check_perceptron(
    fake_numbers, targets_fake_numbers)}%')
```

## Приложение В

### Код реализации нейронной сети обратного распространения ошибки

#### *Листинг В – Реализация нейронной сети обратного распространения ошибки*

```
import pandas as pd
from sklearn.model_selection import train_test_split
import time
import random
import math

class DatasetHandler:
    '''Класс для загрузки, разбиения и отображения данных.'''
    def __init__(self):
        '''Конструктор класса DatasetHandler.'''
        self.data = None
        self.train_data = None
        self.test_data = None

    def load_data(self, file_name: str):
        '''
        Считывает датасет из файла.
        Параметры:
            file_name (str): Путь к файлу с данными.
        '''
        self.data = pd.read_csv(file_name)

    def split_data(self, test_size: float = 0.2, random_state: int = 42):
        '''
        Разделяет датасет на обучающую и тестовую выборки.
        Параметры:
            test_size (float): Доля тестовой выборки (от 0 до 1).
            random_state (int): Случайное состояние для воспроизводимости.
        '''
        if self.data is None:
            raise ValueError("Данные не загружены. Используйте load_data().")
        features = self.data.iloc[:, :-1].values
        targets = self.data.iloc[:, -1].apply(lambda x: 1 if x == "Good" else 0).values
        self.train_data, self.test_data = train_test_split(
            list(zip(features, targets)), test_size=test_size,
            random_state=random_state
        )

    def show_sample(self, n: int = 10):
        '''
        Отображает несколько строк датасета.
        Параметры:
            n (int): Количество строк для отображения.
        '''
        if self.data is None:
            raise ValueError("Данные не загружены. Используйте load_data().")
        print(self.data.head(n))

class NeuralNetwork:
    '''Класс нейронной сети с одним скрытым слоем.'''
    def __init__(self, input_size: int, hidden_size: int, output_size: int,
        learning_rate: float = 0.01):
        '''
```

### Продолжение Листинга В

```
Конструктор класса NeuralNetwork.
Параметры:
    input_size (int): Размер входного слоя.
    hidden_size (int): Размер скрытого слоя.
    output_size (int): Размер выходного слоя.
    learning_rate (float): Скорость обучения.
'''
self.input_size = input_size
self.hidden_size = hidden_size
self.output_size = output_size
self.learning_rate = learning_rate

self.weights_input_hidden = [
    [random.uniform(-1, 1) for _ in range(hidden_size)] for _ in
range(input_size)]
self.weights_hidden_output = [
    [random.uniform(-1, 1) for _ in range(output_size)] for _ in
range(hidden_size)]
self.bias_hidden = [random.uniform(-1, 1) for _ in range(hidden_size)]
self.bias_output = [random.uniform(-1, 1) for _ in range(output_size)]

@staticmethod
def sigmoid(x: float) -> float:
    '''
    Сигмоидальная (логистическая) функция активации.
    Параметры:
        x (float): Входное значение.
    Возвращает:
        float: Результат применения функции.
    '''
    return 1 / (1 + math.exp(-x))

@staticmethod
def sigmoid_derivative(x: float) -> float:
    '''
    Производная сигмоидальной функции.
    Параметры:
        x (float): Значение, к которому применяется производная.
    Возвращает:
        float: Производная функции.
    '''
    return x * (1 - x)

def feedforward(self, inputs: list[float]) -> list[float]:
    '''
    Выполняет прямое распространение данных через сеть.
    Параметры:
        inputs (list[float]): Входные значения.
    Возвращает:
        list[float]: Выходные значения сети.
    '''
    self.inputs = inputs
    self.hidden_inputs = [
        sum(inputs[i] * self.weights_input_hidden[i][j] for i in
range(self.input_size)) + self.bias_hidden[j]
        for j in range(self.hidden_size)
    ]
    self.hidden_outputs = [self.sigmoid(x) for x in self.hidden_inputs]
    self.final_inputs = [
        sum(self.hidden_outputs[j] * self.weights_hidden_output[j][k] for j
in range(self.hidden_size)) + self.bias_output[k]
```

### Продолжение Листинга В

```
        for k in range(self.output_size)
    ]
    self.final_outputs = [self.sigmoid(x) for x in self.final_inputs]
    return self.final_outputs

def backpropagate(self, targets: list[float]):
    """
    Выполняет обратное распространение ошибки.

    Параметры:
        targets (list[float]): Целевые значения.
    """
    output_errors = [targets[k] - self.final_outputs[k] for k in
range(self.output_size)]
    output_gradients = [
        output_errors[k] * self.sigmoid_derivative(self.final_outputs[k])
for k in range(self.output_size)
    ]

    hidden_errors = [
        sum(self.weights_hidden_output[j][k] * output_gradients[k] for k in
range(self.output_size))
        for j in range(self.hidden_size)
    ]
    hidden_gradients = [
        hidden_errors[j] * self.sigmoid_derivative(self.hidden_outputs[j])
for j in range(self.hidden_size)
    ]

    for j in range(self.hidden_size):
        for k in range(self.output_size):
            self.weights_hidden_output[j][k] += self.learning_rate *
output_gradients[k] * self.hidden_outputs[j]

    for i in range(self.input_size):
        for j in range(self.hidden_size):
            self.weights_input_hidden[i][j] += self.learning_rate *
hidden_gradients[j] * self.inputs[i]

    for k in range(self.output_size):
        self.bias_output[k] += self.learning_rate * output_gradients[k]
    for j in range(self.hidden_size):
        self.bias_hidden[j] += self.learning_rate * hidden_gradients[j]

def train(self, training_data: list[tuple[list[float], float]], epochs: int
= 100):
    """
    Обучает нейронную сеть на предоставленных данных.
    Параметры:
        training_data (list[tuple[list[float], float]]): Обучающая выборка.
        epochs (int): Количество эпох обучения.
    """
    for epoch in range(epochs):
        total_loss = 0
        for inputs, targets in training_data:
            outputs = self.feedforward(inputs)
            self.backpropagate([targets])
            total_loss += sum((targets - outputs[k]) ** 2 for k in
range(len(outputs))) * 0.5
        if epoch % 10 == 0:
            print(f"Epoch {epoch}, Loss: {total_loss:.4f}")
```



```
def test(self, test_data: list[tuple[list[float], float]]) -> tuple[float, float]:
    '''
    Тестирует нейронную сеть.
    Параметры:
        test_data (list[tuple[list[float], float]]): Тестовая выборка.
    Возвращает:
        tuple[float, float]: Точность (в процентах) и ошибка.
    '''
    correct_predictions = 0
    total_loss = 0
    for inputs, targets in test_data:
        outputs = self.feedforward(inputs)
        prediction = round(outputs[0])
        correct_predictions += int(prediction == targets)
        total_loss += 0.5 * (targets - outputs[0]) ** 2
    accuracy = (correct_predictions / len(test_data)) * 100
    print(f"Количество верных предсказаний: {correct_predictions}/{len(test_data)}")
    print(f"Test Accuracy: {accuracy:.2f}%, Loss: {total_loss:.4f}")
    return accuracy, total_loss

if __name__ == "__main__":
    dataset_handler = DatasetHandler()
    dataset_handler.load_data("banana_quality.csv")
    dataset_handler.split_data(test_size=0.2)
    dataset_handler.show_sample()

    train_data = dataset_handler.train_data
    test_data = dataset_handler.test_data

    start = time.perf_counter()
    nn = NeuralNetwork(input_size=7, hidden_size=5, output_size=1,
learning_rate=0.01)
    nn.train(train_data, epochs=100)
    print(f"Время обучения: {time.perf_counter() - start:.2f} секунд")
    nn.test(test_data)

# Size - размер плода
# Weight - вес плода
# Sweetness - сладость плода
# Softness - мягкость плода
# HarvestTime - количество времени, прошедшее с момента сбора плода
# Ripeness - спелость плода
# Acidity - кислотность фруктов
# Quality - качество фруктов
```

## Приложение Г

### Код реализации нейронной сети радиально-базисных функций

#### *Листинг Г – Реализация нейронной сети радиально-базисных функций*

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score

class DatasetHandler:
    '''Класс для загрузки, разбиения и отображения данных.'''

    def __init__(self):
        '''Конструктор класса DatasetHandler.'''
        self.data = None
        self.train_data = None
        self.test_data = None

    def load_data(self, file_name: str):
        '''
        Считывает датасет из файла.
        Параметры:
            file_name (str): Путь к файлу с данными.
        '''
        self.data = pd.read_csv(file_name)

    def split_data(self, test_size: float = 0.2, random_state: int = 42):
        '''
        Разделяет датасет на обучающую и тестовую выборки.
        Параметры:
            test_size (float): Доля тестовой выборки (от 0 до 1).
            random_state (int): Случайное состояние для воспроизводимости.
        '''
        if self.data is None:
            raise ValueError("Данные не загружены. Используйте load_data().")
        features = self.data.iloc[:, :-1].values
        targets = self.data.iloc[:, -1].apply(lambda x: 1 if x == "Good" else 0).values
        self.train_data, self.test_data = train_test_split(
            list(zip(features, targets)), test_size=test_size,
            random_state=random_state
        )

    def show_sample(self, n: int = 10):
        '''
        Отображает несколько строк датасета.
        Параметры:
            n (int): Количество строк для отображения.
        '''
        if self.data is None:
            raise ValueError("Данные не загружены. Используйте load_data().")
        print(self.data.head(n))

class RBFNetwork:
    '''
    Класс радиальной базисной сети (RBF-сети).
    '''
```

```

'''
def __init__(self, n_hidden_neurons: int, sigma: float = None):
    '''
    Конструктор класса RBFNetwork.
    Параметры:
        n_hidden_neurons (int): Количество нейронов скрытого слоя.
        sigma (float): Ширина окна радиальной функции. Если None,
        рассчитывается автоматически.
    '''
    self.n_hidden_neurons = n_hidden_neurons
    self.sigma = sigma # Радиус функции.
    self.centers = None
    self.weights = None # Веса выходного слоя.

def _rbf_function(self, X: np.ndarray, center: np.ndarray) -> np.ndarray:
    '''
    Радиальная базисная функция (Гауссова функция).
    Параметры:
        X (np.ndarray): Входные данные.
        center (np.ndarray): Центр RBF.
    Возвращает:
        np.ndarray: Значения RBF для каждого входного вектора.
    '''
    return np.exp(-np.linalg.norm(X - center, axis=1)**2 / (2 *
self.sigma**2))

def fit(self, X: np.ndarray, y: np.ndarray):
    '''
    Обучение RBF-сети.
    Параметры:
        X (np.ndarray): Обучающие входные данные.
        y (np.ndarray): Обучающие метки классов.
    '''
    # Этап 1: Выбор центров RBF с помощью алгоритма K-means.
    kmeans = KMeans(n_clusters=self.n_hidden_neurons,
random_state=52).fit(X)
    self.centers = kmeans.cluster_centers_

    # Этап 2: Определение ширины окна "сигма".
    if self.sigma is None:
        distances = [np.linalg.norm(c1 - c2) for i, c1 in
enumerate(self.centers)
                    for c2 in self.centers[i + 1:]]
        self.sigma = np.mean(distances) / \
            np.sqrt(2 * self.n_hidden_neurons)

    # Этап 3: Формирование матрицы RBF (матрицы G).
    G = np.zeros((X.shape[0], self.n_hidden_neurons))
    for i, center in enumerate(self.centers):
        G[:, i] = self._rbf_function(X, center)

    # Этап 4: Вычисление весов выходного слоя (метод наименьших квадратов).
    self.weights = np.linalg.pinv(G) @ y

def predict(self, X: np.ndarray) -> np.ndarray:
    '''
    Предсказание классов для входных данных.
    Параметры:
        X (np.ndarray): Входные данные.
    Возвращает:

```

```

        np.ndarray: Предсказанные метки классов.
    """
    # Формирование матрицы RBF для входных данных.
    G = np.zeros((X.shape[0], self.n_hidden_neurons))
    for i, center in enumerate(self.centers):
        G[:, i] = self._rbf_function(X, center)

    # Рассчитываем выходные значения сети.
    predictions = G @ self.weights
    return (predictions >= 0.5).astype(int)

def train(self, train_data: list[tuple[np.ndarray, int]], epochs: int = 1):
    """
    Обучает RBF-сеть.
    Параметры:
        train_data (list[tuple[np.ndarray, int]]): Обучающая выборка.
        epochs (int): Количество эпох (для RBF обычно 1 достаточно).
    """
    features, targets = zip(*train_data)
    X = np.array(features)
    y = np.array(targets)
    self.fit(X, y)

def test(self, test_data: list[tuple[np.ndarray, int]]) -> None:
    """
    Тестирует RBF-сеть и выводит результаты.
    Параметры:
        test_data (list[tuple[np.ndarray, int]]): Тестовая выборка.
    """
    features, targets = zip(*test_data)
    X = np.array(features)
    y = np.array(targets)

    predictions = self.predict(X)

    correct_predictions = np.sum(predictions == y)
    total = len(y)
    accuracy = accuracy_score(y, predictions)

    print(f"Количество верных предсказаний: {correct_predictions}/{total}")
    print(f"Точность: {accuracy * 100:.2f}%")

if __name__ == "__main__":
    dataset_handler = DatasetHandler()
    dataset_handler.load_data("banana_quality.csv")
    dataset_handler.split_data(test_size=0.2)
    train_data = dataset_handler.train_data
    test_data = dataset_handler.test_data
    train_features, train_labels = zip(*train_data)
    test_features, test_labels = zip(*test_data)
    train_features = np.array(train_features)
    train_labels = np.array(train_labels)
    test_features = np.array(test_features)
    test_labels = np.array(test_labels)

    rbf_net = RBFNetwork(n_hidden_neurons=5)
    rbf_net.train(list(zip(train_features, train_labels)))

    rbf_net.test(list(zip(test_features, test_labels)))

```

## Приложение Д

### Код реализации карты Кохонена

#### *Листинг Д – Реализация карты Кохонена*

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import MinMaxScaler

class MNISTData:
    '''Класс для загрузки и обработки данных MNIST.'''
    def __init__(self):
        '''Конструктор класса MNISTData.'''
        self.data: np.ndarray = None
        self.labels: np.ndarray = None

    def load_data(self) -> None:
        '''Загрузка данных MNIST и их нормализация.'''
        print("Загрузка данных MNIST...")
        mnist = fetch_openml('mnist_784', version=1)
        self.data = mnist.data.astype(np.float32)
        self.labels = mnist.target.astype(int)

        scaler = MinMaxScaler()
        self.data = scaler.fit_transform(self.data)
        print("Данные успешно загружены и нормализованы.")

    def get_training_data(self) -> tuple[np.ndarray, np.ndarray]:
        '''
        Получение обучающей выборки.
        Возвращает:
            tuple[np.ndarray, np.ndarray]: Кортеж, содержащий данные и метки
            обучающей выборки.
        '''
        return self.data[:60000], self.labels[:60000]

    def get_test_data(self) -> tuple[np.ndarray, np.ndarray]:
        '''
        Получение тестовой выборки.
        Возвращает:
            tuple[np.ndarray, np.ndarray]: Кортеж, содержащий данные и метки
            тестовой выборки.
        '''
        return self.data[60000:], self.labels[60000:]

    def get_number_of_each_digit(self) -> pd.DataFrame:
        '''
        Получение количества изображений каждой цифры в обучающей и тестовой
        выборках.
        Возвращает:
            pd.DataFrame: Таблица с количеством изображений каждой цифры.
        '''
        train_counts =
pd.Series(self.labels[:60000]).value_counts().sort_index()
        test_counts = pd.Series(self.labels[60000:]).value_counts().sort_index()
        table = pd.DataFrame({
            'Цифра': range(10),
            'Обучающая выборка': train_counts.values,
```

```

        'Тестовая выборка': test_counts.values
    })
    print(table)
    return table

class SOM:
    '''Класс для реализации нейронной сети Кохонена (Self-Organizing Map).'''
    def __init__(self, grid_shape: tuple[int, int], input_dim: int,
learning_rate: float = 0.1, radius: float = None):
        '''
        Конструктор класса SOM.
        Параметры:
            grid_shape (tuple[int, int]): Размер сетки нейронов (строки,
столбцы).
            input_dim (int): Размерность входных данных.
            learning_rate (float): Скорость обучения.
            radius (float): Радиус соседства нейронов. Если не указан,
используется половина максимального размера сетки.
        '''
        self.grid_shape = grid_shape
        self.input_dim = input_dim
        self.learning_rate = learning_rate
        self.radius = radius if radius is not None else max(grid_shape) / 2
        self.weights = np.random.random((grid_shape[0], grid_shape[1],
input_dim))
        self.time_constant = 1000 / np.log(self.radius)

    def train(self, data: np.ndarray, num_iterations: int) -> None:
        '''
        Обучение сети Кохонена.
        Параметры:
            data (np.ndarray): Обучающие данные.
            num_iterations (int): Количество итераций обучения.
        '''
        for k in range(num_iterations):
            sample = data[np.random.randint(0, data.shape[0])]
            bmu_idx = self._find_bmu(sample)
            self._update_weights(sample, bmu_idx, k, num_iterations)
            if (k + 1) % 100 == 0 or k == 0:
                print(f"Итерация {k + 1}/{num_iterations} завершена.")

    def _find_bmu(self, sample: np.ndarray) -> tuple[int, int]:
        '''
        Поиск Best Matching Unit (BMU) — нейрона с наименьшим расстоянием до
входного вектора.
        Параметры:
            sample (np.ndarray): Входной вектор.
        Возвращает:
            tuple[int, int]: Индексы BMU в сетке нейронов.
        '''
        distances = np.linalg.norm(self.weights - sample, axis=2)
        return np.unravel_index(np.argmin(distances), self.grid_shape)

    def _update_weights(self, sample: np.ndarray, bmu_idx: tuple[int, int], k:
int, num_iterations: int) -> None:
        '''
        Обновление весов нейронов.
        Параметры:
            sample (np.ndarray): Входной вектор.
            bmu_idx (tuple[int, int]): Индексы BMU.
            k (int): Текущая итерация обучения.
        '''

```

```

        num_iterations (int): Общее количество итераций обучения.
    """
    lr = self.learning_rate * np.exp(-k / num_iterations)
    radius_decay = self.radius * np.exp(-k / self.time_constant)

    for x in range(self.grid_shape[0]):
        for y in range(self.grid_shape[1]):
            distance = np.linalg.norm(np.array([x, y]) - np.array(bmu_idx))
            if distance < radius_decay:
                influence = np.exp(-distance**2 / (2 * (radius_decay**2)))
                self.weights[x, y] += lr * influence * (sample -
self.weights[x, y])

    def visualize_weights(self) -> None:
        '''Визуализация весов нейронов в виде изображений.'''
        fig, axes = plt.subplots(self.grid_shape[0], self.grid_shape[1],
figsize=(10, 10))
        for i in range(self.grid_shape[0]):
            for j in range(self.grid_shape[1]):
                axes[i, j].imshow(self.weights[i, j].reshape(28, 28),
cmap='gray')
                axes[i, j].axis('off')
        plt.show()

    def test(self, test_data: np.ndarray, test_labels: np.ndarray) -> None:
        '''
        Тестирование сети Кохонена на тестовой выборке.
        Параметры:
            test_data (np.ndarray): Тестовые данные.
            test_labels (np.ndarray): Метки тестовых данных.
        '''
        print("Тестирование на тестовой выборке...")
        bmu_positions = []
        for sample in test_data:
            bmu_idx = self._find_bmu(sample)
            bmu_positions.append(bmu_idx)

        bmu_positions = np.array(bmu_positions)
        plt.figure(figsize=(10, 10))
        plt.scatter(bmu_positions[:, 1], bmu_positions[:, 0], c=test_labels,
cmap='tab10', s=5)
        plt.colorbar(label='Цифра')
        plt.gca().invert_yaxis()
        plt.title('Распределение тестовых данных на SOM')
        plt.xlabel('X координата нейрона')
        plt.ylabel('Y координата нейрона')
        plt.show()

    def count_clusters(self, data: np.ndarray, labels: np.ndarray) -> dict[int,
int]:
        '''
        Подсчет количества кластеров для каждой цифры.
        Параметры:
            data (np.ndarray): Данные для кластеризации.
            labels (np.ndarray): Метки данных.
        Возвращает:
            dict[int, int]: Словарь, где ключи — цифры, а значения — количество
кластеров для каждой цифры.
        '''
        print("Подсчет количества кластеров для каждой цифры...")
        cluster_counts = {}

```

### *Окончание Листинга Д*

```
        for digit in range(10):
            digit_data = data[labels == digit]
            bmu_positions = set()
            for sample in digit_data:
                bmu_idx = self._find_bmu(sample)
                bmu_positions.add(bmu_idx)
            cluster_counts[digit] = len(bmu_positions)
        print("Количество кластеров:", cluster_counts)
        return cluster_counts

if __name__ == "__main__":
    mnist_data = MNISTData()
    mnist_data.load_data()

    print("Таблица с количеством изображений каждой цифры:")
    mnist_data.get_number_of_each_digit()

    train_data, train_labels = mnist_data.get_training_data()

    som = SOM(grid_shape=(10, 10), input_dim=784, learning_rate=0.5)
    print("Обучение SOM...")
    som.train(train_data, num_iterations=1000)

    print("Визуализация весов карты Кохонена...")
    som.visualize_weights()

    test_data, test_labels = mnist_data.get_test_data()
    print("Распределение тестовых данных на SOM:")
    som.test(test_data, test_labels)

    cluster_counts = som.count_clusters(train_data, train_labels)
```



## Приложение Е

### Код реализации сети встречного распространения

*Листинг Е – Реализация сети встречного распространения*

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

class MNISTData:
    '''Класс для загрузки и обработки данных MNIST.'''
    def __init__(self):
        self.data: np.ndarray = None
        self.labels: np.ndarray = None

    def load_data(self) -> None:
        '''Загрузка данных MNIST и их нормализация.'''
        print("Загрузка данных MNIST...")
        mnist = fetch_openml('mnist_784', version=1)
        self.data = mnist.data.astype(np.float32)
        self.labels = mnist.target.astype(int)

        scaler = MinMaxScaler()
        self.data = scaler.fit_transform(self.data)
        print("Данные успешно загружены и нормализованы.")

    def get_training_data(self) -> tuple[np.ndarray, np.ndarray]:
        '''Получение обучающей выборки.'''
        return self.data[:60000], self.labels[:60000]

    def get_test_data(self) -> tuple[np.ndarray, np.ndarray]:
        '''Получение тестовой выборки.'''
        return self.data[60000:], self.labels[60000:]

class CPN:
    '''Сеть встречного распространения с слоями Кохонена и Гроссберга.'''
    def __init__(self, kohonen_shape=(10, 10), input_dim=784, output_dim=10,
learning_rate=0.1):
        '''
        Конструктор сети.
        Параметры:
            kohonen_shape (tuple): Размер сетки слоя Кохонена.
            input_dim (int): Размерность входного вектора.
            output_dim (int): Количество классов (меток).
            learning_rate (float): Скорость обучения.
        '''
        self.kohonen_shape = kohonen_shape
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.learning_rate = learning_rate

        # Инициализация весов
        self.kohonen_weights = np.random.random((kohonen_shape[0],
kohonen_shape[1], input_dim))
        self.grossberg_weights = np.zeros((kohonen_shape[0], kohonen_shape[1],
output_dim))

    def train(self, X_train, y_train, num_epochs=10):
        '''Обучение сети на данных.'''
```

```

print("Начало обучения сети встречного распространения...")
for epoch in range(num_epochs):
    for i, sample in enumerate(X_train):
        # Слой Кохонена: находим BMU
        bmu_idx = self._find_bmu(sample)

        # Обновляем веса слоя Кохонена
        self._update_kohonen_weights(sample, bmu_idx)

        # Обновляем веса слоя Гроссберга
        self._update_grossberg_weights(y_train[i], bmu_idx)

    print(f"Эпоха {epoch + 1}/{num_epochs} завершена.")
print("Обучение завершено!")

def predict(self, X):
    '''Предсказание меток для входных данных.'''
    predictions = []
    for sample in X:
        # Найти BMU на слое Кохонена
        bmu_idx = self._find_bmu(sample)

        # Получить предсказание на слое Гроссберга
        grossberg_output = self.grossberg_weights[bmu_idx]
        predicted_label = np.argmax(grossberg_output)
        predictions.append(predicted_label)
    return np.array(predictions)

def _find_bmu(self, sample):
    '''Находим нейрон с минимальным расстоянием на слое Кохонена (BMU).'''
    distances = np.linalg.norm(self.kohonen_weights - sample, axis=2)
    return np.unravel_index(np.argmin(distances), self.kohonen_shape)

def _update_kohonen_weights(self, sample, bmu_idx):
    '''Обновление весов слоя Кохонена с использованием функции соседства.'''
    x, y = bmu_idx
    radius = 2
    for i in range(self.kohonen_shape[0]):
        for j in range(self.kohonen_shape[1]):
            distance = np.linalg.norm(np.array([i, j]) - np.array([x, y]))
            if distance < radius:
                influence = np.exp(-distance**2 / (2 * radius**2))
                self.kohonen_weights[i, j] += self.learning_rate * influence
    * (sample - self.kohonen_weights[i, j])

def _update_grossberg_weights(self, target_label, bmu_idx):
    '''Обновление весов слоя Гроссберга.'''
    x, y = bmu_idx
    target_output = np.zeros(self.output_dim)
    target_output[target_label] = 1
    self.grossberg_weights[x, y] += self.learning_rate * (target_output -
self.grossberg_weights[x, y])

if __name__ == "__main__":
    # Загрузка данных
    mnist_data = MNISTData()
    mnist_data.load_data()

    # Получение обучающих и тестовых данных
    X_train, y_train = mnist_data.get_training_data()
    X_test, y_test = mnist_data.get_test_data()

```

### *Окончание Листинга E*

```
# Создание и обучение сети встречного распространения
cpn = CPN(kohonen_shape=(10, 10), input_dim=784, output_dim=10,
learning_rate=0.1)
cpn.train(X_train, y_train, num_epochs=10)

# Тестирование сети
print("Тестирование сети...")
y_pred = cpn.predict(X_test)
print("Точность на тестовой выборке:", accuracy_score(y_test, y_pred))
print("Матрица ошибок:")
print(confusion_matrix(y_test, y_pred))
print("Отчет по классификации:")
print(classification_report(y_test, y_pred))
```

## Приложение Ж.1

### Датасет для рекуррентной сети

*Листинг Ж.1 – Датасет для рекуррентной сети*

```
train_data = {
    'good': True,
    'bad': False,
    'happy': True,
    'sad': False,
    'not good': False,
    'not bad': True,
    'not happy': False,
    'not sad': True,
    'very good': True,
    'very bad': False,
    'very happy': True,
    'very sad': False,
    'i am happy': True,
    'this is good': True,
    'i am bad': False,
    'this is bad': False,
    'i am sad': False,
    'this is sad': False,
    'i am not happy': False,
    'this is not good': False,
    'i am not bad': True,
    'this is not sad': True,
    'i am very happy': True,
    'this is very good': True,
    'i am very bad': False,
    'this is very sad': False,
    'this is very happy': True,
    'i am good not bad': True,
    'this is good not bad': True,
    'i am bad not good': False,
    'i am good and happy': True,
    'this is not good and not happy': False,
    'i am not at all good': False,
    'i am not at all bad': True,
    'i am not at all happy': False,
    'this is not at all sad': True,
    'this is not at all happy': False,
    'i am good right now': True,
    'i am bad right now': False,
    'this is bad right now': False,
    'i am sad right now': False,
    'i was good earlier': True,
    'i was happy earlier': True,
    'i was bad earlier': False,
    'i was sad earlier': False,
    'i am very bad right now': False,
    'this is very good right now': True,
    'this is very sad right now': False,
    'this was bad earlier': False,
    'this was very good earlier': True,
    'this was very bad earlier': False,
    'this was very happy earlier': True,
    'this was very sad earlier': False,
    'i was good and not bad earlier': True,
    'i was not good and not happy earlier': False,
```

### *Окончание Листинга Ж.1*

```
'i am not at all bad or sad right now': True,  
'i am not at all good or happy right now': False,  
'this was not happy and not good earlier': False,  
}  
  
test_data = {  
    'this is happy': True,  
    'i am good': True,  
    'this is not happy': False,  
    'i am not good': False,  
    'this is not bad': True,  
    'i am not sad': True,  
    'i am very good': True,  
    'this is very bad': False,  
    'i am very sad': False,  
    'this is bad not good': False,  
    'this is good and happy': True,  
    'i am not good and not happy': False,  
    'i am not at all sad': True,  
    'this is not at all good': False,  
    'this is not at all bad': True,  
    'this is good right now': True,  
    'this is sad right now': False,  
    'this is very bad right now': False,  
    'this was good earlier': True,  
    'i was not happy and not good earlier': False,  
}
```

## Приложение Ж.2

### Код реализации рекуррентной сети

*Листинг Ж.2 – Код реализации рекуррентной сети*

```
import numpy as np
from numpy.random import randn

class RNN:
    '''Простая рекуррентная нейронная сеть (RNN) с архитектурой "многие-в-
    один".'''

    def __init__(self, input_size: int, output_size: int, hidden_size: int =
64):
        '''
        Инициализация параметров RNN.

        Параметры:
            input_size (int): Размер входных данных (размер словаря).
            output_size (int): Размер выходных данных (например, количество
классов).
            hidden_size (int): Размер скрытого слоя.
        '''
        # Инициализация весов
        self.Whh = randn(hidden_size, hidden_size) / 1000 # Веса скрытого слоя
        self.Wxh = randn(hidden_size, input_size) / 1000  # Веса между входом и
скрытым слоем
        self.Why = randn(output_size, hidden_size) / 1000 # Веса между скрытым
слоем и выходом

        # Инициализация смещений
        self.bh = np.zeros((hidden_size, 1)) # Смещение скрытого слоя
        self.by = np.zeros((output_size, 1)) # Смещение выходного слоя

    def forward(self, inputs: list[np.ndarray]) -> tuple[np.ndarray,
np.ndarray]:
        '''
        Прямой проход через нейронную сеть.
        Параметры:
            inputs (list[np.ndarray]): Список входных данных, каждый элемент -
one-hot вектор.

        Возвращает:
            tuple[np.ndarray, np.ndarray]: Кортеж из итогового выхода и скрытых
состояний.
        '''
        h = np.zeros((self.Whh.shape[0], 1)) # Начальное скрытое состояние

        self.last_inputs = inputs # Сохраняем входные данные
        self.last_hs = {0: h} # Словарь для хранения скрытых состояний

        # Выполнение каждого шага RNN
        for i, x in enumerate(inputs):
            h = np.tanh(self.Wxh @ x + self.Whh @ h + self.bh) # Обновление
скрытого состояния
            self.last_hs[i + 1] = h # Сохраняем скрытое состояние

        # Вычисление выхода
        y = self.Why @ h + self.by
        return y, h
```

```

def backprop(self, d_y: np.ndarray, learn_rate: float = 2e-2) -> None:
    """
    Обратное распространение ошибки для обновления весов.

    Параметры:
        d_y (np.ndarray): Градиент ошибки по выходу (dL/dy).
        learn_rate (float): Коэффициент обучения.
    """
    n = len(self.last_inputs)

    # Вычисление градиентов по весам
    d_Why = d_y @ self.last_hs[n].T
    d_by = d_y

    # Инициализация градиентов для скрытых слоев
    d_Whh = np.zeros(self.Whh.shape)
    d_Wxh = np.zeros(self.Wxh.shape)
    d_bh = np.zeros(self.bh.shape)

    # Вычисление градиента по скрытому состоянию для последнего шага
    d_h = self.Why.T @ d_y

    # Обратное распространение ошибки через временные шаги
    for t in reversed(range(n)):
        # Промежуточное значение для вычисления градиентов
        temp = ((1 - self.last_hs[t + 1] ** 2) * d_h)

        # Градиенты по параметрам
        d_bh += temp
        d_Whh += temp @ self.last_hs[t].T
        d_Wxh += temp @ self.last_inputs[t].T

        # Обновление градиента по скрытому состоянию для предыдущего шага
        d_h = self.Whh @ temp

    # Обновление весов с помощью градиентного спуска
    for d in [d_Wxh, d_Whh, d_Why, d_bh, d_by]:
        np.clip(d, -1, 1, out=d) # Обрезка градиентов для предотвращения
        взрывных градиентов

    # Применяем обновления весов
    self.Whh -= learn_rate * d_Whh
    self.Wxh -= learn_rate * d_Wxh
    self.Why -= learn_rate * d_Why
    self.bh -= learn_rate * d_bh
    self.by -= learn_rate * d_by

```

## Приложение Ж.3

### Код реализации файла main.py

*Листинг Ж.3 – Код реализации файла main.py*

```
import numpy as np
import random
from RNN import RNN
from data import train_data, test_data

vocab = list(set([w for text in train_data.keys() for w in text.split(' ')]))
vocab_size = len(vocab)
print(f'Найдено {vocab_size} уникальных слов.')

word_to_idx = {w: i for i, w in enumerate(vocab)}
idx_to_word = {i: w for i, w in enumerate(vocab)}

def createInputs(text: str) -> list[np.ndarray]:
    """
    Создает список одноразрядных векторов для представления слов текста.
    Параметры:
        text (str): Входной текст.
    Возвращает:
        list[np.ndarray]: Список векторов формы (vocab_size, 1).
    """
    inputs = []
    for w in text.split(' '):
        v = np.zeros((vocab_size, 1))
        v[word_to_idx[w]] = 1
        inputs.append(v)
    return inputs

def softmax(xs: np.ndarray) -> np.ndarray:
    """
    Применяет функцию Softmax к входному массиву.
    Параметры:
        xs (np.ndarray): Входной массив.
    Возвращает:
        np.ndarray: Результат применения функции Softmax.
    """
    return np.exp(xs) / np.sum(np.exp(xs), axis=0)

rnn = RNN(vocab_size, 2)

def processData(data: dict[str, bool], backprop: bool = True) -> tuple[float, float]:
    """
    Рассчитывает потери и точность RNN для заданных данных.
    Параметры:
        data (dict[str, bool]): Словарь с текстами и их метками (True/False).
        backprop (bool): Указывает, выполнять ли обратное распространение
    ошибки.
    Возвращает:
        tuple[float, float]: Потери и точность.
    """
    items = list(data.items())
    random.shuffle(items)
    loss = 0
```



### Окончание Листинга Ж.3

```
num_correct = 0

for x, y in items:
    inputs = createInputs(x)
    target = int(y)

    out, _ = rnn.forward(inputs)
    probs = softmax(out)

    loss -= np.log(probs[target, 0])
    num_correct += int(np.argmax(probs) == target)

    if backprop:
        d_L_d_y = probs
        d_L_d_y[target] -= 1
        rnn.backprop(d_L_d_y)

return loss / len(data), num_correct / len(data)

for epoch in range(1000):
    train_loss, train_acc = processData(train_data)

    if epoch % 100 == 99:
        print(f'--- Epoch {epoch + 1}')
        print(f'Train:\tLoss {float(train_loss):.3f} | Accuracy:
{float(train_acc):.3f}')

        test_loss, test_acc = processData(test_data, backprop=False)
        print(f'Test:\tLoss {float(test_loss):.3f} | Accuracy:
{float(test_acc):.3f}')
```

## Приложение 3

### Код реализации сверточной сети

*Листинг 3 – Код реализации сверточной сети*

```
import tensorflow as tf
from tensorflow.keras import layers, models, datasets
import matplotlib.pyplot as plt

# Загрузка и подготовка данных MNIST
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Нормализация

# Добавление канала цвета (для совместимости с CNN)
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

# Преобразование меток в формат one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Построение модели CNN
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax') # Выходной слой для классификации
])

# Компиляция модели
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=10,
                   batch_size=64, validation_split=0.1)

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Точность на тестовой выборке: {test_acc:.2f}")

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Точность на обучении')
plt.plot(history.history['val_accuracy'], label='Точность на валидации')
plt.xlabel('Эпохи')
plt.ylabel('Точность')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Потери на обучении')
plt.plot(history.history['val_loss'], label='Потери на валидации')
plt.xlabel('Эпохи')
plt.ylabel('Потери')
plt.legend()

plt.show()
```