



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

---

---

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

**Отчёт по практическим работам**  
**по дисциплине**  
**«Проектирование и разработка нейронных сетей»**

Студент группы: ИКБО-04-22

Егоров Л.А.  
(Ф.И.О. студента)

Принял старший преподаватель

Семёнов Р.Э.  
(Ф.И.О. преподавателя)

Москва 2024

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	3
1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА .....	4
1.1 Входные данные .....	4
1.2 Результат работы алгоритма .....	4
2 ДЕЛЬТА-ПРАВИЛО .....	6
2.1 Входные данные .....	6
2.2 Результат работы алгоритма .....	6
3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ .....	8
3.1 Входные данные .....	8
3.2 Результат работы алгоритма .....	8
4 НЕЙРОННЫЕ СЕТИ НА РАДИАЛЬНО-БАЗИСНЫХ ФУНКЦИЯХ .....	11
4.1 Входные данные .....	11
4.2 Результат работы алгоритма .....	11
5 КАРТЫ КОХОНЕНА .....	12
5.1 Входные данные .....	12
5.2 Результат работы алгоритма .....	12
6 НЕЙРОННЫЕ СЕТИ ВСТРЕЧНОГО РАСПРОСТРАНЕНИЯ .....	13
6.1 Входные данные .....	13
6.2 Результат работы алгоритма .....	13
7 РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ .....	14
7.1 Входные данные .....	14
7.2 Результат работы алгоритма .....	14
8 СВЁРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ .....	15
8.1 Входные данные .....	15
8.2 Результат работы алгоритма .....	15
ЗАКЛЮЧЕНИЕ .....	16
ПРИЛОЖЕНИЯ .....	17

# ВВЕДЕНИЕ

Нейронные сети, вдохновленные структурой человеческого мозга, совершили революцию в мире технологий, проникнув во все сферы нашей жизни. От распознавания лиц на смартфонах до прогнозирования погоды, от перевода языков до создания реалистичных изображений — нейросети стали неотъемлемой частью современного мира.

Нейронные сети активно применяются в различных областях, от медицины и финансов до развлечений и образования. За нейронными сетями лежит будущее во многих сферах жизнедеятельности общества.

Нейронные сети являются подклассом алгоритмов машинного обучения, и называются глубоким обучением. Цель данной дисциплины изучить принципы работы, архитектуры нейросетей и алгоритмы их оптимизации (обучения).

# 1 ОБУЧЕНИЕ ПО ПРАВИЛАМ ХЕББА

Правила Хебба, сформулированные в 1949, гласят:

1. Если сигнал персептрона неверен и равен нулю, то необходимо увеличить веса тех входов, на которые была подана единица.
2. Если сигнал персептрона неверен и равен единице, то необходимо уменьшить веса тех входов, на которые была подана единица.

В этой практической работе была построена модель персептрона и обучена в соответствии с правилами Хебба.

## 1.1 Входные данные

Для данной работы используется датасет с отказами сердца в зависимости от различных данных о здоровье пациента. Часть датасета представлена в Таблице 1.1.1.

Таблица 1.1.1 — *Datascem heart.csv*

age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	output
63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
41	0	1	130	204	0	0	172	0	1.4	2	0	2	1

Эти данные преобразованы с помощью Z-нормализации (1.1).

$$z = \frac{x - \mu}{\sigma} \quad (1.1)$$

## 1.2 Результат работы алгоритма

Нейрон обучен за 100 эпох с коэффициентом  $\alpha$ , равным 0.0005. Результат обучения представлен на Рисунках 1.2.1 – 1.2.2. Код программы представлен в Листинге А.1.

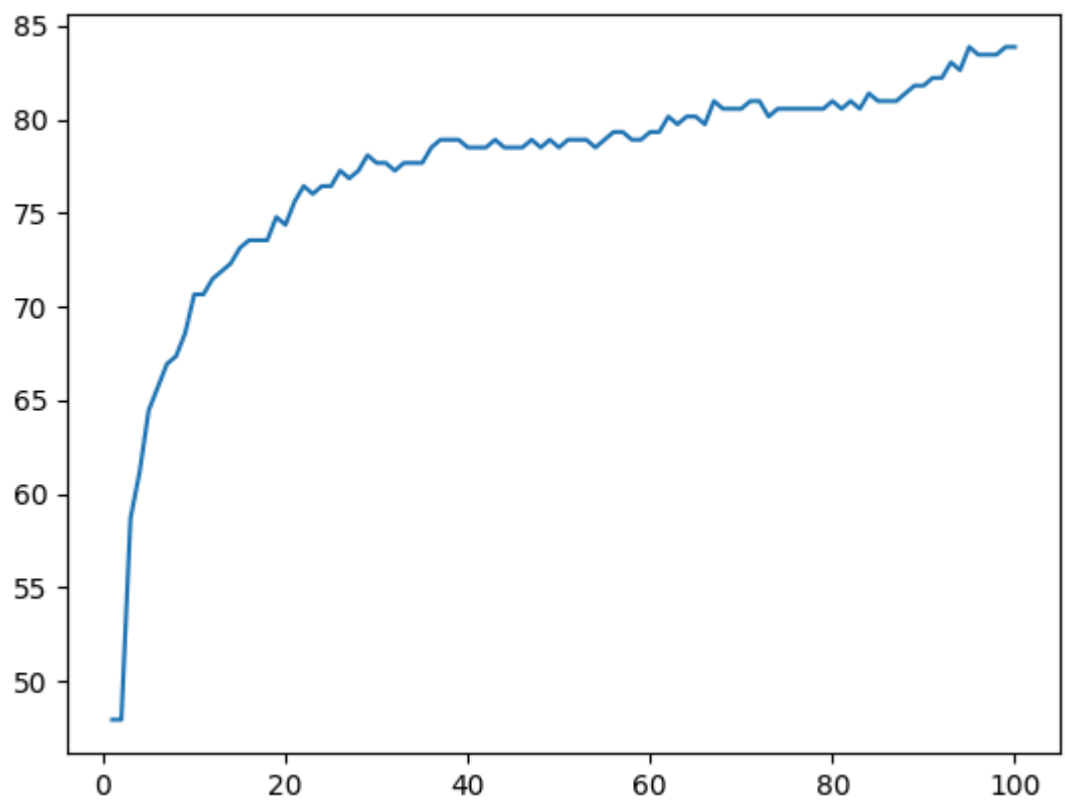


Рисунок 1.2.1 — Изменение точности

```

Epoch: 94/100
Epoch: 95/100
Epoch: 96/100
Epoch: 97/100
Epoch: 98/100
Epoch: 99/100
Epoch: 100/100
[0 0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1 1 0
 0 1 0 1 1 1 0 1 0 0 1 1 1 0 1 0 0 1 1 0 0 0 1 0]
Accuracy: 73.77%

```

Рисунок 1.2.2 — Точность на тестовом датасете

## 2 ДЕЛЬТА-ПРАВИЛО

Дельта правило является улучшенной версией алгоритма Хебба. По сравнению со своим предшественником дельта правило имеет параметр, отвечающий за скорость сходимости метода обучения, называемый скоростью обучения.

### 2.1 Входные данные

Для данной работы используется датасет с отказами сердца в зависимости от различных данных о здоровье пациента. Часть датасета представлена в Таблице 1.1.1.

Эти данные преобразованы с помощью Minmax нормализации (2.1).

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (2.1)$$

### 2.2 Результат работы алгоритма

Нейрон обучен за 100 эпох с коэффициентом  $\alpha$ , равным 0.0005. Результат обучения представлен на Рисунках 2.2.1 – 2.2.2. Код программы представлен в Листинге Б.1.

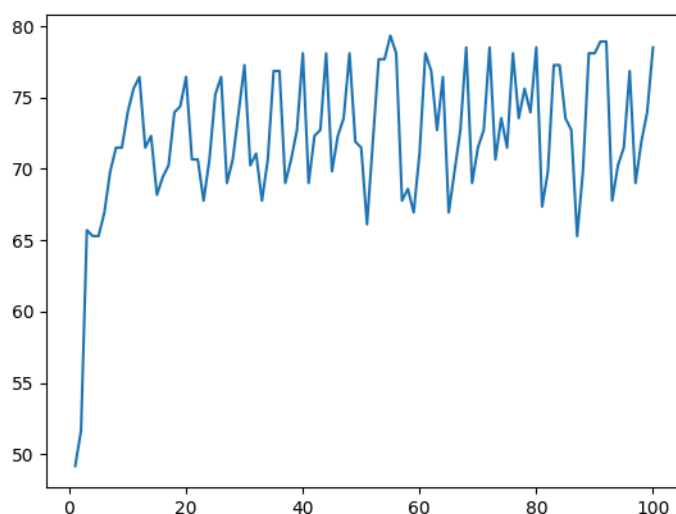


Рисунок 2.2.1 — Изменение точности

```
Epoch: 95/100  
Epoch: 96/100  
Epoch: 97/100  
Epoch: 98/100  
Epoch: 99/100  
Epoch: 100/100  
[1 0 0 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 1 1 1 1 1 0 1 0 1 1 1 1 1 1  
 1 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 0 0 0 1 0]  
Accuracy: 54.79709755442086%
```

Рисунок 2.2.2 — Точность на тестовом датасете

## 3 ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ОШИБКИ

Алгоритм обратного распространения ошибки (backpropagation) — это метод обучения искусственных нейронных сетей, который используется для минимизации ошибки предсказания путем корректировки весов сети. Он работает в два этапа: прямое распространение сигнала для вычисления выхода сети и обратное распространение ошибки для обновления весов на основе градиента функции потерь. Алгоритм эффективен и широко применяется в задачах машинного обучения, таких как классификация и регрессия.

### 3.1 Входные данные

Для данной работы используется датасет с отказами сердца в зависимости от различных данных о здоровье пациента. Часть датасета представлена в Таблице 1.1.1.

Эти данные преобразованы с помощью Minmax нормализации (2.1).

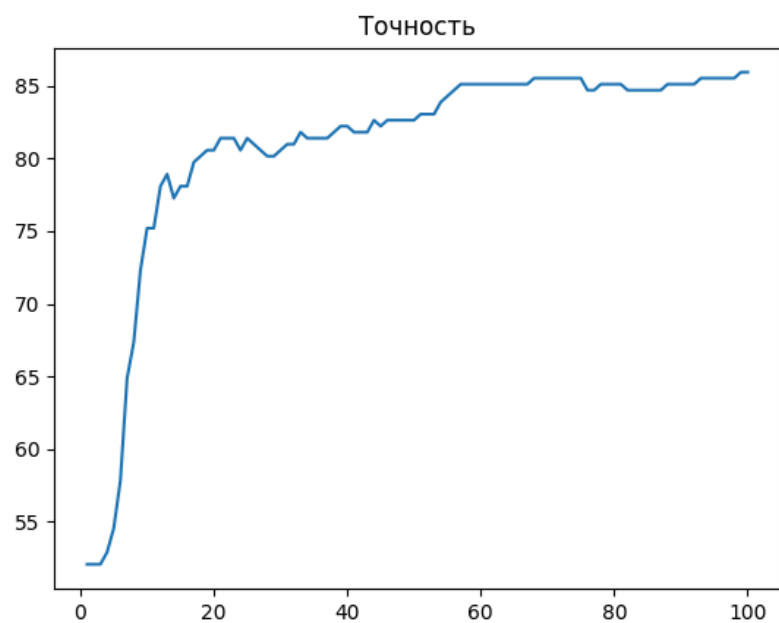
### 3.2 Результат работы алгоритма

Характеристики нейронной сети:

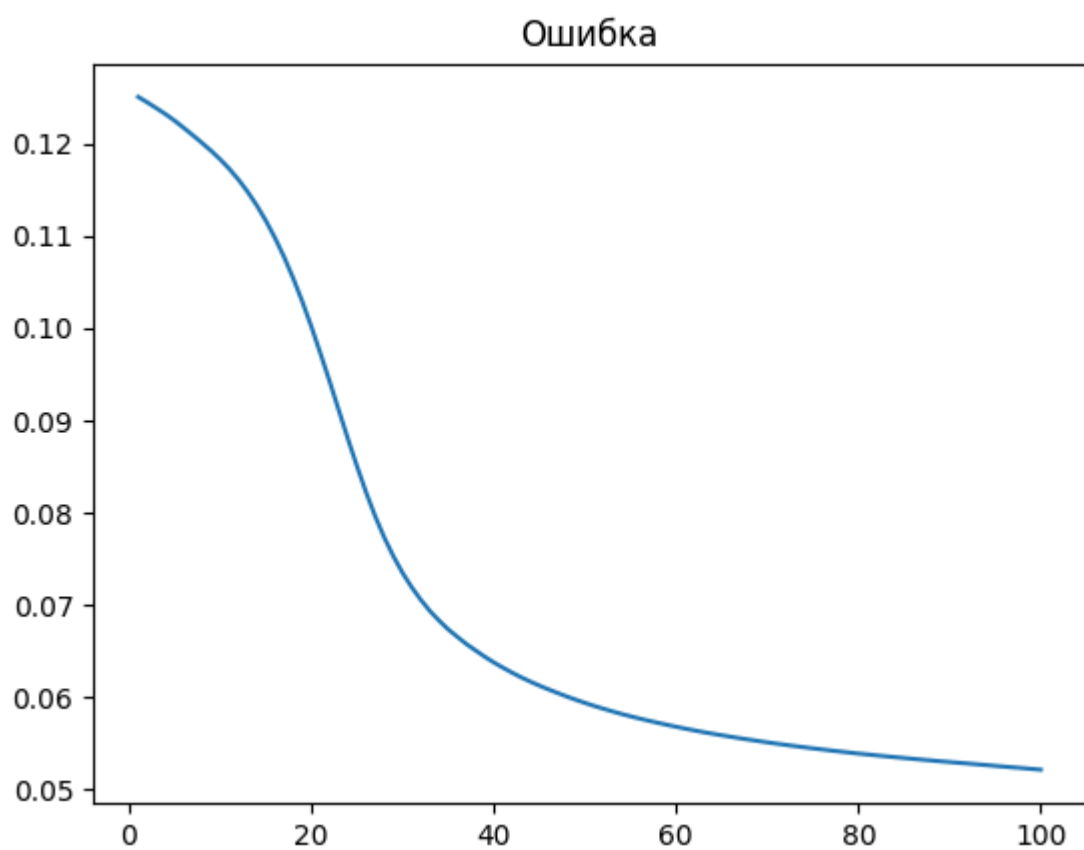
- количество скрытых слоёв: 2;
- нейронов в первом скрытом слое: 6;
- нейронов во втором скрытом слое: 3;
- активационные функции в скрытых слоях: *relu*;
- активационная функция на выходном слое: *sigmoid*;
- скорость обучения: 0.01;
- количество эпох: 100.

Результат обучения представлен на Рисунках 3.2.1 – 3.2.3. Код программы представлен в Листинге В.1.





**Рисунок 3.2.1 — Изменение точности**



**Рисунок 3.2.2 — Изменение ошибки**

```
Эпоха: 98/100
Ошибка: 0.052330846764291575
Точность: 85.53719008264463%
Эпоха: 99/100
Ошибка: 0.052242864544710235
Точность: 85.9504132231405%
Эпоха: 100/100
Ошибка: 0.05216094088067376
Точность: 85.9504132231405%
Точность на тестовом датасете: 81.9672131147541%
(лучше)
```

**Рисунок 3.2.3 — Точность на тестовом датасете**

## 4 НЕЙРОННЫЕ СЕТИ НА РАДИАЛЬНО-БАЗИСНЫХ ФУНКЦИЯХ

RBF-сети — это тип нейронных сетей, который использует радиально-базисные функции в качестве функций активации в скрытом слое. RBF-сети хорошо справляются с аппроксимацией нелинейных функций, и они менее чувствительны к шуму в данных.

### 4.1 Входные данные

Для данной работы используется датасет с отказами сердца в зависимости от различных данных о здоровье пациента. Часть датасета представлена в Таблице 1.1.1.

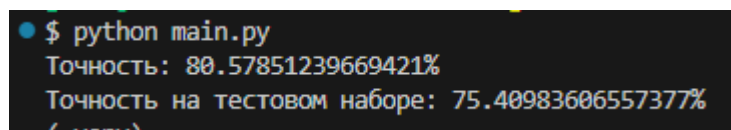
Эти данные преобразованы с помощью Minmax нормализации (2.1).

### 4.2 Результат работы алгоритма

Характеристики нейронной сети:

- количество радиально-базисных функций: 10;
- скорость обучения: 0.01;
- количество эпох: 100.

Результат обучения данной нейронной сети представлен на Рисунке 4.2.1. Код программы представлен в Листинге Г.1



```
$ python main.py
Точность: 80.57851239669421%
Точность на тестовом наборе: 75.40983606557377%
( mean)
```

Рисунок 4.2.1 — Точность RBF-сети

## 5 КАРТЫ КОХОНЕНА

SOM (Карты Кохонена) — это тип нейронных сетей, который используется для неконтролируемого обучения и визуализации данных. Карты самоорганизации работают на основе конкуренции между нейронами и подходят для группировки 2-мерных данных, такие как изображения. Также их можно применять и для данных больших размерностей, поскольку карта Кохонена переведёт эти данные на двумерную сетку.

### 5.1 Входные данные

Для данной работы используется датасет с отказами сердца в зависимости от различных данных о здоровье пациента. Часть датасета представлена в Таблице 1.1.1.

Эти данные преобразованы с помощью Minmax нормализации (2.1).

### 5.2 Результат работы алгоритма

Характеристики нейронной сети:

- размер сетки:  $10 \times 10$ .

Рисунок 5.2.1 - Матрица весов до обучения

Рисунок 5.2.2 - Матрица весов после обучения

## **6 НЕЙРОННЫЕ СЕТИ ВСТРЕЧНОГО РАСПРОСТРАНЕНИЯ**

CPN - это тип нейронных сетей, используемых для решения задач классификации и аппроксимации функций. Они отличаются от традиционных нейронных сетей, таких как многослойные перцептроны, своим архитектурным строением и принципом работы.

### **6.1 Входные данные**

В качестве задачи выберем аппроксимацию функции, которая является отображением из экспоненциального распределения в нормальное распределение. Нейросеть будет иметь 3 входа и 3 выхода. Возьмем 1000 синтетически сгенерированных примеров и разобьем их на пакеты по 16 штук.

### **6.2 Результат работы алгоритма**

Скорость обучения для слоя Кохонена возьмем 0.7, для слоя Гроссберга 0.1, количество эпох 2. Кривая обучения представлена ниже.

Рисунок 6.2.1 – Кривая обучения нейросети

## 7 РЕКУРРЕНТНЫЕ НЕЙРОННЫЕ СЕТИ

Рекуррентные нейросети являются более современной архитектурой нейронных сетей. Их основным преимуществом является возможность запоминать предыдущие значения во входных данных, что является важным свойством для анализа последовательностей – текста, временных рядов, звука и тд. В данной работе реализуем рекуррентную сеть на базе 2 GRU слоев (100 и 50 нейронов) и между ними функцию активации ELU.

### 7.1 Входные данные

В качестве задачи возьмём регрессию. Синтетически сгенерируем набор данных, который содержит 25000 образца с 100 признаками. Все данные масштабируем к диапазону  $[0;1]$  по формуле:

Данные будем подавать в пакетах по 32 образца. Разделим данные на обучающую и тестовую выборки в соотношении 80% и 20% соответственно.

### 7.2 Результат работы алгоритма

В качестве метрики выберем среднюю квадратичную ошибку, так как она чувствительна к большим ошибкам и уменьшает влияние более маленьких ошибок. А скорость обучения выберем равную 0.010, количество эпох равное 25 и будем увеличивать скорость обучения на 10% каждую эпоху. Скрытые состояния будем сохранять в нейросети. На графиках ниже представлены кривые обучения нейросети (верхняя за цикл обучения, нижняя за цикл теста) – на оси абсцисс – эпоха, на оси ординат – средняя ошибка за эпоху.

Рисунок 7.2.1 – Кривая обучения нейросети

## 8 СВЁРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ

Сверточные нейронные сети (CNN) - это тип нейронных сетей, специально разработанный для обработки данных с пространственной структурой, например, изображений, аудио и текстов. Ключевыми особенностями являются: слой свертки (извлекают ключевую информацию) и слой пулинга (уменьшают размерность данных, убирая избыточную информацию). В данной работе предлагает сделать вариационный автокодировщик (VAE) для генерации рукописных цифр. Главная особенность VAE в том, что кодировщик VAE переводит входные данные в латентное нормальное распределение, а декодировщик из нормального распределения восстанавливает первоначальные данные.

### 8.1 Входные данные

Вариационный автокодировщик обучается без учителя, так как в процессе обучения он учится восстанавливать первоначальные данные, то есть на выходе у VAE должны быть входные данные. В качестве набора данных возьмем набор из 1797 одноцветных изображений рукописных цифр 8 на 8 пикселей. Входные данные приведем к диапазону  $[0;1]$  по формуле:  $x\_scaled = x/x\_max$  Данные разобьем на пакеты по 16 образцов.

### 8.2 Результат работы алгоритма

В качестве метрики выберем ELBO, которая состоит из 2 частей – ошибка реконструкции (в данном случае Log Loss) и расстояние Кульбака — Лейблера. А скорость обучения выберем равную 0.002, количество эпох равное 100. На рисунке ниже представлен результат генерации цифры из случайного вектора.

Рисунок 8.2.1 – Результат генерации рукописной цифры

## **ЗАКЛЮЧЕНИЕ**

В результате практических работ были изучены алгоритмы и оптимизации нейронных сетей и их архитектуры.

Важно отметить, что глубокое обучение — это динамично развивающаяся область. Постоянные исследования и разработки новых алгоритмов и архитектур нейронных сетей продолжаются, и полученные в рамках данной дисциплины знания являются отправной точкой для дальнейшего изучения и освоения новых технологий.



## ПРИЛОЖЕНИЯ

- Приложение А — Реализация обучения по правилам Хебба
- Приложение Б — Реализация дельта-правила
- Приложение В — Реализация обратного распространения ошибки
- Приложение Г — Реализация сети на радиально-базисных функциях
- Приложение Д — Реализация карты Кохонена
- Приложение Е — Реализация сети встречного распространения
- Приложение Ж — Реализация рекуррентной нейронной сети
- Приложение И — Реализация свёрточной нейронной сети

## Приложение А

### Реализация обучения по правилам Хебба

*Листинг А.1 — Код файла main.py*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

class StupidNeuralNetwork:

    def __init__(self):
        self.weights: np.ndarray = None
        self.b: np.ndarray = None
        self.lr = 0.0005
        self.epochs = 100

    def init_weights(self, input_size: int, output_size: int):
        if self.weights is None:
            self.weights = np.zeros(shape=(input_size, output_size))
        if self.b is None:
            self.b = -np.random.random()

    def train(self, dataset: np.ndarray, target: np.array):
        self.dataset = dataset
        self._init_weights(self.dataset.shape[1], 1)

        accuracy_list = []
        for epoch in range(self.epochs):
            print(f"Epoch: {epoch + 1}/{self.epochs}")
            for row, target_value in zip(self.dataset, target):
                result = row @ self.weights + self.b
                result = (result > 0).astype(np.int8)
                if result != target_value:
                    if result == 0:
                        self.weights += self.lr * np.reshape(row, (-1, 1))
                        self.b += self.lr
                    else:
                        self.weights -= self.lr * np.reshape(row, (-1, 1))
                        self.b -= self.lr
            epoch_result = self.dataset @ self.weights + self.b
            epoch_result = (epoch_result > 0).astype(np.int8).reshape(-1)
            accuracy = (epoch_result == target).mean() * 100
            accuracy_list.append(accuracy)

        plt.plot(range(1, self.epochs + 1), accuracy_list)
        plt.show()

    def test(self, data: np.ndarray, target: np.array):
        result = data @ self.weights + self.b
        result = (result > 0).astype(np.int8).reshape(-1)
        print(result)
        accuracy = (result == target).mean() * 100
        return accuracy

class Dataset:

    def __init__(self) -> None:
        self._max: pd.DataFrame = None
        self._min: pd.DataFrame = None
        self._std: pd.DataFrame = None
        self._mean: pd.DataFrame = None
        self.weights: np.ndarray = None

    def normalize_data(self, df: pd.DataFrame):
        self.df = df
        if self._max is None:
```

### Окончание Листинга A.1

```
        self._max = self.df.max()
    if self._min is None:
        self._min = self.df.min()
    if self._std is None:
        self._std = self.df.std()
    if self._mean is None:
        self._mean = self.df.mean()

    # self.df = (self.df - self._min) / (self._max - self._min)
    # self.df = (self.df > 0.5).astype(np.int8)
    self.df = (self.df - self._mean) / self._std
    self.df = (self.df > 0).astype(np.int8)

    def prepare_dataset(self):
        self.train_target = self.df['output'].to_numpy()
        self.train_data = self.df.drop(columns=['output']).to_numpy()
        self.train_data, self.test_data, self.train_target, self.test_target = (
            train_test_split(self.train_data,
                             self.train_target,
                             shuffle=True,
                             random_state=78498,
                             test_size=0.2))

    def main():
        df = pd.read_csv('../dataset/heart.csv')
        dataset = Dataset()
        dataset.normalize_data(df)
        dataset.prepare_dataset()

        nn = StupidNeuralNetwork()
        nn.train(dataset.train_data, dataset.train_target)
        accuracy = nn.test(dataset.test_data, dataset.test_target)
        print(f'Accuracy: {accuracy:.2f}%')

if __name__ == '__main__':
    main()
```

## Приложение Б

### Реализация дельта-правила

*Листинг Б.1 — Код файла main.py*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

class StupidNeuralNetwork:

    def __init__(self):
        self.weights: np.ndarray = None
        self.b: np.ndarray = None
        self.lr = 0.01
        self.epochs = 100

    def init_weights(self, input_size: int, output_size: int):
        if self.weights is None:
            self.weights = np.random.random(size=(input_size, output_size))
        if self.b is None:
            self.b = -np.random.random(size=(output_size, 1))

    def train(self, dataset: np.ndarray, target: np.ndarray):
        self.dataset = dataset

        self._init_weights(self.dataset.shape[-1], target.shape[-1])

        accuracy_list = []
        batch_size = len(target)
        for epoch in range(self.epochs):
            print(f"Epoch: {epoch + 1}/{self.epochs}")
            for row, target_value in zip(np.split(self.dataset,
self.dataset.shape[0] // batch_size),
np.split(target, target.shape[0] // batch_size)):
                result = row @ self.weights + self.b
                result = (result > 0).astype(np.int8)

                loss = target_value - result
                self.weights += self.lr * (row.T @ loss)
                self.b += self.lr * loss.mean(axis=0)
            epoch_result = self.dataset @ self.weights + self.b
            epoch_result = (epoch_result > 0).astype(np.int8)
            accuracy = (epoch_result == target).mean() * 100
            accuracy_list.append(accuracy)

        plt.plot(range(1, self.epochs + 1), accuracy_list)
        plt.show()

    def test(self, data: np.ndarray, target: np.array):
        result = data @ self.weights + self.b
        result = (result > 0).astype(np.int8).reshape(-1)
        print(result)
        accuracy = (result == target).mean() * 100
        return accuracy

class Dataset:

    def __init__(self) -> None:
        self._max: pd.DataFrame = None
        self._min: pd.DataFrame = None
        self._std: pd.DataFrame = None
        self._mean: pd.DataFrame = None
        self.weights: np.ndarray = None

    def normalize_data(self, df: pd.DataFrame):
        self.df = df
```

### Окончание Листинга Б.1

```
        if self._max is None:
            self._max = self.df.max()
        if self._min is None:
            self._min = self.df.min()
        if self._std is None:
            self._std = self.df.std()
        if self._mean is None:
            self._mean = self.df.mean()

        self.df = (self.df - self._min) / (self._max - self._min)
        self.df = (self.df > 0.5).astype(np.int8)
        # self.df = (self.df - self._mean) / self._std
        # self.df = (self.df > 0).astype(np.int8)

    def prepare_dataset(self):
        self.train_target = self.df['output'].to_numpy().reshape((-1, 1))
        self.train_data = self.df.drop(columns=['output']).to_numpy()
        self.train_data, self.test_data, self.train_target, self.test_target = (
            train_test_split(self.train_data,
                            self.train_target,
                            shuffle=True,
                            random_state=78498,
                            test_size=0.2))

def main():
    df = pd.read_csv('../dataset/heart.csv')
    dataset = Dataset()
    dataset.normalize_data(df)
    dataset.prepare_dataset()

    nn = StupidNeuralNetwork()
    nn.train(dataset.train_data, dataset.train_target)
    accuracy = nn.test(dataset.test_data, dataset.test_target)
    print(f'Accuracy: {accuracy}%')

if __name__ == '__main__':
    main()
```

## Приложение В

### Реализация обратного распространения ошибки

Листинг В.1 — Код файла *main.py*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

class StupidLayer:
    def __init__(self, size: tuple[int], lr: float = 0.01, activate: str = 'relu'):
        self.input_size = size[0]
        self.output_size = size[1]
        self.weights = np.random.random(size=(input_size, output_size)) - 0.5
        self.b = np.random.random(size=(output_size, )) - 0.5
        self.lr = lr

        activate_func = {'relu': lambda x: np.maximum(0, x),
                          'sigmoid': lambda x: 1 / (1 + np.exp(-x))}
        derivative = {'relu': lambda x: (x > 0).astype(np.int8),
                      'sigmoid': lambda x: x * (1 - x)}
        self.activate = activate_func[activate]
        self.deriv = derivative[activate]
        self.input: np.ndarray
        self.output: np.ndarray

    def forward(self, data: np.ndarray):
        self.input = data.copy()
        self.output = self.input @ self.weights + self.b
        self.output = self.activate(self.output)
        return self.output

    def back_prop(self, next_loss: np.ndarray) -> np.ndarray:
        self.delta = self.deriv(self.output) * next_loss
        return self.delta @ self.weights.T

    def calculate_weights(self):
        self.weights -= self.lr * self.input.T @ self.delta
        self.b -= self.lr * self.delta.sum(axis=0)

class StupidNeuralNetwork:
    def __init__(self, epochs=100, batch_size: int | None = None):
        self.layers = []
        self.epochs = epochs
        self.batch_size = batch_size

    def create_layers(self, input_size: int, output_size: int):
        self.layers = [StupidLayer((input_size, 6), activate='relu'),
                        StupidLayer((6, 3), activate='relu'),
                        StupidLayer((3, output_size), activate='sigmoid')]

    def forward(self, row: np.ndarray):
        for layer in self.layers:
            row = layer.forward(row)
        return row

    def backward(self, target: np.ndarray, result: np.ndarray):
        loss_deriv = (result - target)
        for layer in reversed(self.layers):
            loss_deriv = layer.back_prop(loss_deriv)

        for layer in self.layers:
            layer.calculate_weights()

    def train(self, dataset: np.ndarray, target: np.ndarray):
        self.create_layers(dataset.shape[1], target.shape[1])
```

```
        accuracy_list = []
        epoch_loss_list = []
        self.batch_size = len(dataset) if self.batch_size is None else
self.batch_size

    for epoch in range(self.epochs):
        print(f"Эпоха: {epoch + 1}/{self.epochs}")
        epoch_loss = []
        for i in range(0, len(dataset), self.batch_size):
            X_batch = dataset[i:i + self.batch_size]
            y_batch = target[i:i + self.batch_size]

            result = self.forward(X_batch)
            loss = 0.5 * ((y_batch - result) ** 2).mean()
            epoch_loss.append(loss)

            self.backward(y_batch, result)

        avg_loss = np.mean(epoch_loss)
        print(f'Ошибка: {avg_loss}')
        epoch_loss_list.append(avg_loss)

        predictions = self.forward(dataset)
        predictions = (predictions >= 0.5).astype(np.int8)
        accuracy = (predictions == target).mean() * 100
        print(f'Точность: {accuracy}%')
        accuracy_list.append(accuracy)

    plt.plot(range(1, self.epochs + 1), accuracy_list)
    plt.title('Точность')
    plt.show()

    plt.plot(range(1, self.epochs + 1), epoch_loss_list)
    plt.title('Ошибка')
    plt.show()

def test(self, data: np.ndarray, target: np.ndarray):
    result = self.forward(data)
    result = (result >= 0.5).astype(np.int8)
    accuracy = (result == target).mean() * 100
    print(f'Точность на тестовом датасете: {accuracy}%')
    return accuracy

class Dataset:

    def __init__(self) -> None:
        self._max: pd.DataFrame = None
        self._min: pd.DataFrame = None
        self._std: pd.DataFrame = None
        self._mean: pd.DataFrame = None
        self.weights: np.ndarray = None

    def normalize_data(self, df: pd.DataFrame):
        self.df = df
        if self._max is None:
            self._max = self.df.max()
        if self._min is None:
            self._min = self.df.min()
        if self._std is None:
            self._std = self.df.std()
        if self._mean is None:
            self._mean = self.df.mean()

        self.df = (self.df - self._min) / (self._max - self._min)
        # self.df = (self.df > 0.5).astype(np.int8)
        # self.df = (self.df - self._mean) / self._std
        # self.df = (self.df > 0).astype(np.int8)

    def prepare_dataset(self):
        self.train_target = self.df['output'].to_numpy().reshape((-1, 1))
        self.train_data = self.df.drop(columns=['output']).to_numpy()
```

### *Окончание Листинга В.1*

```
        self.train_data, self.test_data, self.train_target, self.test_target = (
            train_test_split(self.train_data,
                            self.train_target,
                            shuffle=True,
                            random_state=78498,
                            test_size=0.2))

def main():
    df = pd.read_csv('../dataset/heart.csv')
    dataset = Dataset()
    dataset.normalize_data(df)
    dataset.prepare_dataset()

    nn = StupidNeuralNetwork(epochs=100, batch_size=1)
    nn.train(dataset.train_data, dataset.train_target)
    nn.test(dataset.test_data, dataset.test_target)

if __name__ == '__main__':
    main()
```



## Приложение Г

### Реализация сети на радиально-базисных функциях

Листинг Г.1 — Код файла *main.py*

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans

class RBFNetwork:

    def __init__(self, input_dim, hidden_dim, output_dim, lr=0.01, epochs=100):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.lr = lr
        self.epochs = epochs

    def init_centers(self, dataset: np.ndarray):
        kmeans = KMeans(n_clusters = self.hidden_dim, random_state = 78498)
        kmeans.fit_predict(dataset)
        self.centers = kmeans.cluster_centers_

    @staticmethod
    def gaussian_rbf(x, center, beta):
        return np.linalg.norm(x - center)

    def compute_rbf_layer(self, X):
        # Рассчитываем радиально-базисные функции для входного слоя
        beta = 1.0 # Параметр ширины RBF
        RBF_output = np.zeros((X.shape[0], self.hidden_dim))
        for i, sample in enumerate(X):
            for j, center in enumerate(self.centers):
                RBF_output[i, j] = self.gaussian_rbf(sample, center, beta)
        return RBF_output

    def fit(self, X, y):
        RBF_output = self.compute_rbf_layer(X)
        self.weights = np.linalg.pinv(RBF_output) @ y

    def forward(self, X):
        # Вычисляем выход RBF слоя и итоговый выход сети
        RBF_output = self.compute_rbf_layer(X)
        output = RBF_output @ self.weights
        return output

    def train(self, dataset: np.ndarray, target: np.ndarray):
        self.init_centers(dataset)
        self.fit(dataset, target)
        predictions = self.forward(dataset)

        predictions = (predictions >= 0.5).astype(np.int8)
        accuracy = (predictions == target).mean() * 100
        print(f'Точность: {accuracy}%')

    def predict(self, X):
        output = self.forward(X)
        return output

class Dataset:
    def __init__(self):
        self._max = None
        self._min = None

    def normalize_data(self, df):
        if self._max is None:
```

### Окончание Листинга Г.1

```
        self._max = df.max()
    if self._min is None:
        self._min = df.min()

    self.df = (df - self._min) / (self._max - self._min)

    def prepare_dataset(self):
        self.train_target = self.df['output'].to_numpy().reshape((-1, 1))
        self.train_data = self.df.drop(columns=['output']).to_numpy()
        self.train_data, self.test_data, self.train_target, self.test_target
= train_test_split(
        self.train_data,
        self.train_target,
        shuffle=True,
        random_state=78498,
        test_size=0.2
    )

def main():
    df = pd.read_csv('../dataset/heart.csv')
    dataset = Dataset()
    dataset.normalize_data(df)
    dataset.prepare_dataset()

    rbf_net = RBFNetwork(input_dim=dataset.train_data.shape[1],
                          hidden_dim=10, # Количество радиально-базисных функций
                          output_dim=1,
                          lr=0.01,
                          epochs=100)

    rbf_net.train(dataset.train_data, dataset.train_target)

    predictions = rbf_net.predict(dataset.test_data)
    predictions = (predictions >= 0.5).astype(np.int8)
    accuracy = (predictions == dataset.test_target).mean() * 100
    print(f"Точность на тестовом наборе: {accuracy}%")

if __name__ == '__main__':
    main()
```

## Приложение Д

### Реализация карты Кохонена

#### Листинг Д.1 — Код файла main.py

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import matplotlib.pyplot as plt

class SOM:
    def __init__(self, map_size, input_size, sigma=0.3, learning_rate=0.5):
        self.map_size = map_size
        self.input_size = input_size
        self.sigma = sigma
        self.learning_rate = learning_rate
        self.weights = np.random.rand(map_size[0], map_size[1], input_size)

    def find_bmu(self, x):
        distances = np.linalg.norm(self.weights - x, axis=2)
        return np.unravel_index(np.argmin(distances), distances.shape)

    def neighborhood_function(self, bmu, iteration):
        t1 = 1000
        t2 = 1000
        sigma_t = self.sigma * np.exp(-iteration / t1)
        learning_rate_t = self.learning_rate * np.exp(-iteration / t2)
        dist_sq = np.sum((np.indices(self.map_size).T - np.array(bmu)).T **
2, axis=0)
        return learning_rate_t * np.exp(-dist_sq / (2 * sigma_t ** 2))

    def train(self, data, iterations):
        """ Обучение SOM """
        for iteration in tqdm(range(iterations)):
            for x in data:
                bmu = self.find_bmu(x)
                nh_func = self.neighborhood_function(bmu, iteration)
                self.weights += nh_func[:, :, np.newaxis] * (x - self.weights)

    def visualize(self, data):
        plt.figure(figsize=(7, 7))
        plt.pcolor(np.linalg.norm(self.weights, axis=2).T, cmap='bone_r',
alpha=0.2)
        plt.colorbar()

        for x in data:
            bmu = self.find_bmu(x)
            plt.text(bmu[1] + 0.5, bmu[0] + 0.5, '.', color=plt.cm.Red(x[0]),
fontdict={'weight': 'bold', 'size': 11})

        plt.show()

class Dataset:
    def __init__(self):
        self._max = None
        self._min = None

    def normalize_data(self, df):
        if self._max is None:
            self._max = df.max()
        if self._min is None:
            self._min = df.min()

        self.df = (df - self._min) / (self._max - self._min)

    def prepare_dataset(self):
        self.train_target = self.df['output'].to_numpy().reshape((-1, 1))
```

### Окончание Листинга Д.1

```
        self.train_data = self.df.drop(columns=['output']).to_numpy()
        self.train_data, self.test_data, self.train_target, self.test_target
= train_test_split(
            self.train_data,
            self.train_target,
            shuffle=True,
            random_state=78498,
            test_size=0.2
        )

def main():
    df = pd.read_csv('../dataset/heart.csv')
    dataset = Dataset()
    dataset.normalize_data(df)
    dataset.prepare_dataset()
    som_shape = (10, 10)
    som = SOM(som_shape, dataset.train_data.shape[1], sigma=0.3,
learning_rate=0.5)
    som.train(dataset.train_data, 100)
    som.visualize(dataset.train_data)

if __name__ == '__main__':
    main()
```

## Приложение Е

### Реализация сети встречного распространения

Листинг Е.1 — Код файла *main.py*

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import accuracy_score

class CPN:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights_kohonen = np.random.rand(input_size, hidden_size)

        self.weights_grossberg = np.random.rand(hidden_size, output_size)

    def train_kohonen(self, data, epochs=100, learning_rate=0.1):
        for epoch in range(epochs):
            for x in data:
                bmu_index = np.argmax(np.dot(x, self.weights_kohonen))
                self.weights_kohonen[:, bmu_index] += learning_rate * (x -
self.weights_kohonen[:, bmu_index])

    def train_grossberg(self, data, labels, epochs=100, learning_rate=0.1):
        for epoch in range(epochs):
            print(f'Epoch: {epoch + 1}/{epochs}')
            for x, label in zip(data, labels):
                hidden_output = np.dot(x, self.weights_kohonen)
                hidden_output = hidden_output == np.max(hidden_output)
                hidden_output = hidden_output.astype(int)

                output = np.dot(hidden_output, self.weights_grossberg)

                error = label - output
                self.weights_grossberg += learning_rate * np.outer(hidden_output,
error)

    def predict(self, data):
        predictions = []
        for x in data:
            hidden_output = np.dot(x, self.weights_kohonen)
            hidden_output = hidden_output == np.max(hidden_output)
            hidden_output = hidden_output.astype(int)

            output = np.dot(hidden_output, self.weights_grossberg)
            predictions.append(np.argmax(output))
        return np.array(predictions)

def main():
    iris = load_iris()
    X = iris.data
    y = iris.target

    X = (X - X.mean(axis=0)) / X.std(axis=0)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

    encoder = OneHotEncoder(sparse_output=False)
    y_train_one_hot = encoder.fit_transform(y_train.reshape(-1, 1))
    y_test_one_hot = encoder.transform(y_test.reshape(-1, 1))

    input_size = X_train.shape[1]
```

### *Окончание Листинга E.1*

```
hidden_size = 10
output_size = y_train_one_hot.shape[1]
cpn = CPN(input_size, hidden_size, output_size)
cpn.train_kohonen(X_train, epochs=100, learning_rate=0.1)
cpn.train_grossberg(X_train, y_train_one_hot, epochs=100, learning_rate=0.1)

predictions = cpn.predict(X_test)
accuracy = accuracy_score(y_test, predictions)
print(f"Точность на тестовом наборе: {accuracy:.2f}")

if __name__ == '__main__':
    main()
```

## Приложение Ж

### Реализация рекуррентной нейронной сети

*Листинг Ж.1 — Код файла main.py*

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense
from tensorflow.keras.preprocessing import sequence

max_features = 5000
maxlen = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)

x_train = sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = sequence.pad_sequences(x_test, maxlen=maxlen)

model = Sequential()

model.add(Embedding(input_dim=max_features,
                    output_dim=32, input_length=maxlen))

model.add(SimpleRNN(units=32, return_sequences=False))

model.add(Dense(units=1, activation='sigmoid'))

model.compile(optimizer='adam',
              loss='binary_crossentropy', metrics=['accuracy'])

model.summary()

batch_size = 64
epochs = 3

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
          validation_data=(x_test, y_test))

loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Точность на тестовых данных: {accuracy:.4f}")
```

## Приложение II

### Реализация свёрточной нейронной сети

*Листинг II.1 — Код файла main.py*

```
import tensorflow as tf
from tensorflow import keras
from keras import layers, models
from keras.datasets import mnist
from keras.utils import to_categorical

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1))

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

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

model.fit(x_train, y_train, epochs=5, batch_size=64, validation_split=0.1)

test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Точность на тестовых данных: {test_acc}")
```