МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ МОРСКОЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»  
 (СПБГМТУ)

| ФАКУЛЬТЕТ ЦИФРОВЫХ ПРОМЫШЛЕННЫХ ТЕХНОЛОГИЙ  КАФЕДРА КИБЕРФИЗИЧЕСКИХ СИСТЕМ |
| --- |

ЛАБОРАТОРНАЯ РАБОТА №4

«АЛГОРИТМ ОБРАТНОГО РАСПРОСТРАНЕНИЯ ОШИБКИ»

|  | |
| --- | --- |
|  | Выполнил  студент группы 20221 | | |
|  | Лаптев Иван Александрович | | |
|  | Проверила | | |
|  | Кайнова Татьяна Денисовна | | |
| Санкт-Петербург  2024 год | | |

**СОДЕРЖАНИЕ**

[Цели работы………………………………………………………………………. 2](#_gjdgxs)

[Результат работы…………………………………………………………………..3](#_1fob9te)

[Заключение………………………………………………………………………...](#_1t3h5sf) 7

[Листинг кода………………………………………………………………………](#_4d34og8) 7

# Цели работы

В ходе работы был реализован алгоритм обратного распространения ошибки для нейронной сети с двумя скрытыми слоями по 4 нейрона каждый. Программа обучается на бинарных данных, где на каждом шаге вычисляются выходы нейронов с использованием функции активации tanh. Затем производится расчет ошибок и обновление весов с помощью градиентного спуска. Обучение продолжается до тех пор, пока все примеры не будут правильно классифицированы.

# Результат работы

В результате выполнения работы была доработана программа из лабораторной работы №2, реализующая алгоритм обратного распространения ошибки (Листинг 1). Входные данные представлены в таблице истинности (Таблица 1). Для логической схемы была создана нейронная сеть с двумя скрытыми слоями, каждый из которых содержит 4 нейрона, и одним выходным слоем, который выводит количество пройденных эпох и веса каждого нейрона (Рисунок 2). Был построен график зависимости среднеквадратичной ошибки от числа эпох обучения (Рисунок 3). Результат работы программы представлен на рисунке (Рисунок 4)

Таблица 1 - Таблица истинности

| X1 | X2 | X3 | X4 | NOT(X1 AND X2) | X2 AND X3 | XOR | 5 AND 7 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |

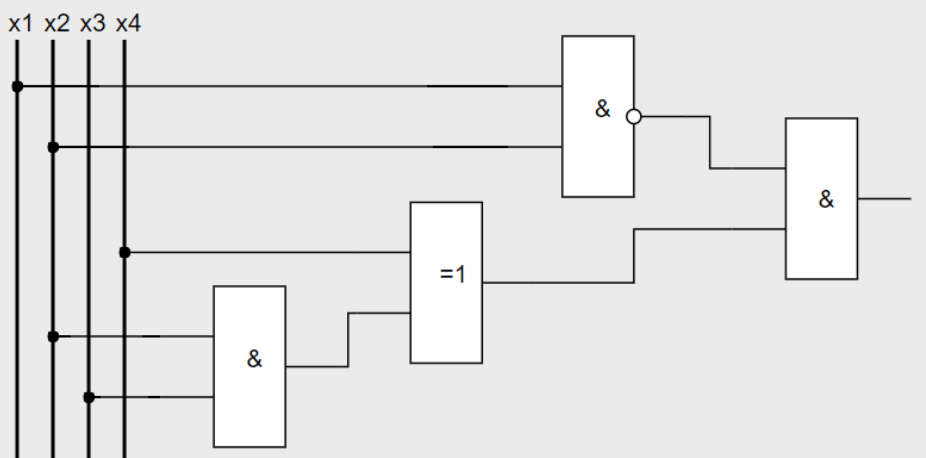
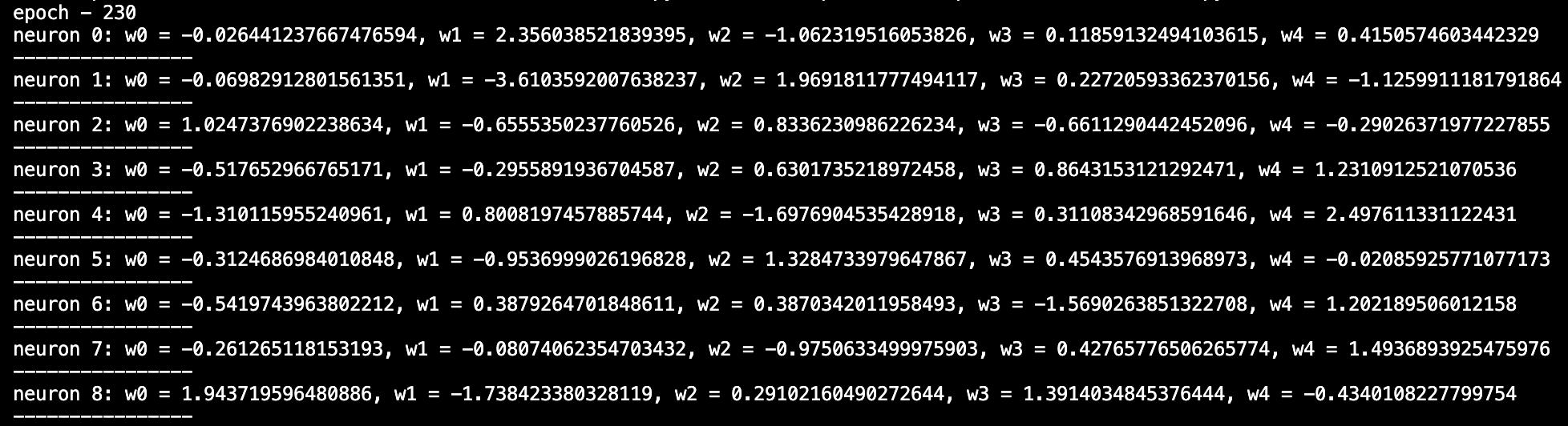


Рисунок 1 - Исходные данные

Рисунок 2 – Конечные значения весов

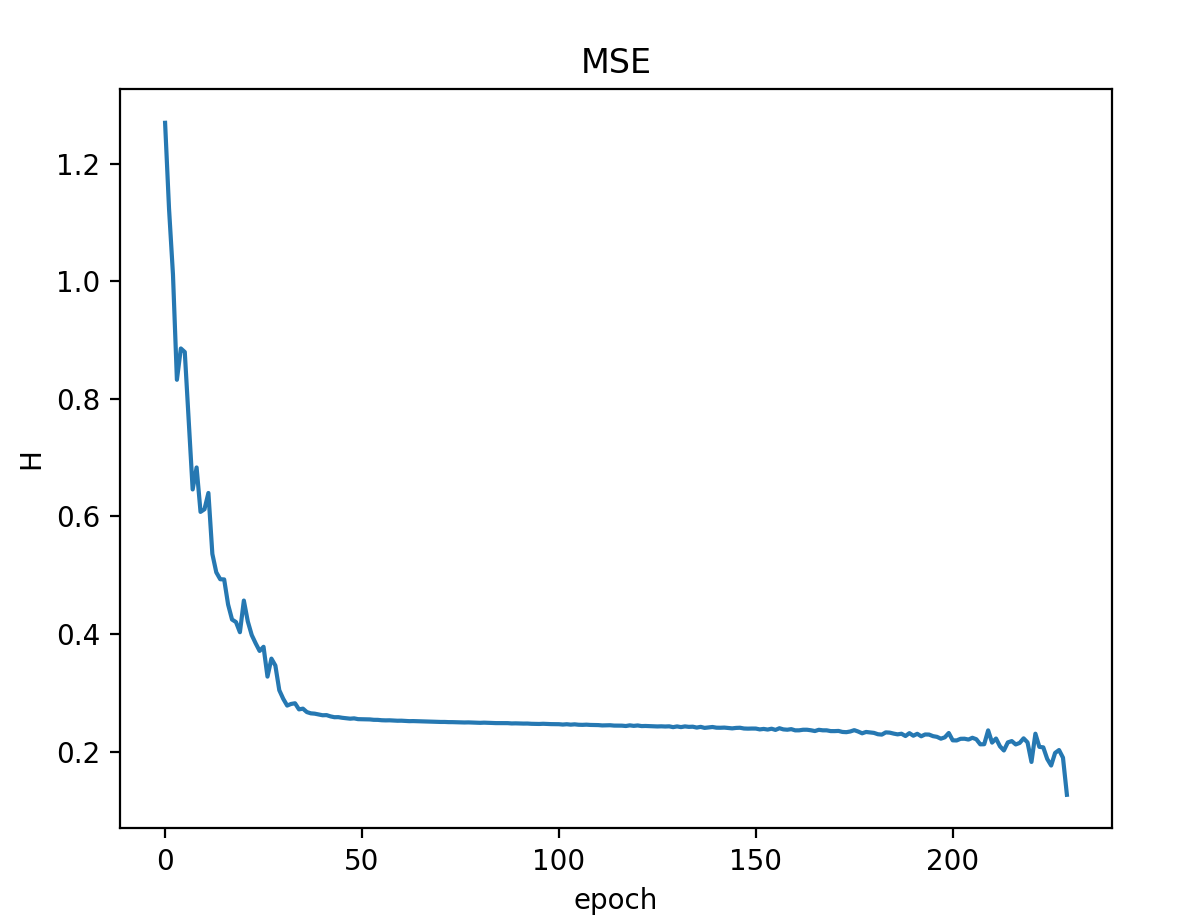


Рисунок 3 - Средняя квадратическая ошибка

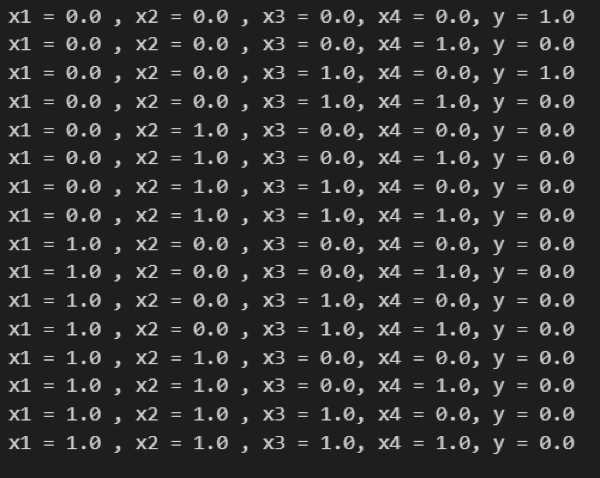


Рисунок 4 - Результат работы программы

Листинг 1 – Алгоритм обратного распространения ошибки

import matplotlib.pyplot as plt # Импорт библиотеки для построения графиков

import numpy as np # Импорт библиотеки для работы с массивами и математическими операциями

np.random.seed(6) # Устанавливаем начальное значение для генератора случайных чисел для воспроизводимости

# Функция для создания случайных весов для нейронов

def neuron\_w(input\_count):

weights = np.zeros(input\_count + 1) # Создаем массив весов, включая смещение (bias)

# Заполняем веса случайными значениями в диапазоне от -1 до 1

for i in range(1, (input\_count + 1)): # Проходим по всем весам, начиная с первого (смещение оставляем 0)

weights[i] = np.random.uniform(-1.0, 1.0) # Генерируем случайное число в диапазоне [-1, 1] для веса

return weights # Возвращаем массив весов

# Генерируем 9 нейронов с 4 входами и случайными весами для каждого из них

n\_w = [neuron\_w(4) for \_ in range(9)] # Для каждого нейрона генерируем веса, всего 9 нейронов

# Генерация входных данных для обучения (бинарные векторы)

x0 = [1 for \_ in range(16)] # Массив смещений, равный 1 (16 значений)

x1 = [1 if i // 8 == 1 else 0 for i in range(16)] # Вектор: 1, если i >= 8, иначе 0

x2 = [1 if i // 4 % 2 != 0 else 0 for i in range(16)] # Вектор: меняется каждые 4 элемента

x3 = [1 if i // 2 % 2 != 0 else 0 for i in range(16)] # Вектор: меняется каждые 2 элемента

x4 = [1 if i % 2 != 0 else 0 for i in range(16)] # Вектор: меняется на каждом элементе

# Формируем обучающий набор данных, комбинируя все входные вектора

x\_train = np.array(list(zip(x0, x1, x2, x3, x4))) # Создаем обучающий набор данных (16 строк по 5 признаков)

# Заданные целевые значения для обучения нейронной сети (классы: 1 или -1)

y\_train = np.array([1, -1, 1, -1, 1, 1, 1, 1, 1, -1, -1, 1, -1, -1, -1, -1]) # Целевые метки для каждого примера

# Класс для нейронной сети

class Network:

LEARNING\_RATE = 0.1 # Устанавливаем скорость обучения (параметр, регулирующий шаг обновления весов)

def \_\_init\_\_(self, w: list, x\_train: list, y\_train: list) -> None:

"""

Инициализация нейронной сети:

- w: веса сети

- x\_train: входные данные

- y\_train: целевые значения

"""

self.w = w # Инициализация весов

self.x\_train = x\_train # Инициализация входных данных

self.y\_train = y\_train # Инициализация целевых значений

self.n\_y = [0 for \_ in range(len(w))] # Список для хранения выходных значений нейронов

self.n\_error = [0 for \_ in range(len(w))] # Список для хранения ошибок нейронов

self.index\_list = [i for i in range(len(self.x\_train))] # Индексы всех примеров для случайной выборки

self.epoch = 0 # Счетчик количества эпох (итераций)

self.weighted\_sum = 0 # Переменная для вычисления взвешенной суммы входов

self.MSE = [] # Массив для хранения значений ошибки на каждой эпохе (среднеквадратичная ошибка)

def show\_learning(self):

"""

Функция для вывода текущих значений весов на каждой эпохе.

"""

print(f'epoch - {self.epoch}') # Выводим номер текущей эпохи

for i, w in enumerate(n\_w): # Для каждого нейрона выводим значения его весов

print(f'neuron {i}: w0 = {w[0]}, w1 = {w[1]}, w2 = {w[2]}, w3 = {w[3]}, w4 = {w[4]}') # Выводим веса каждого нейрона

print('----------------') # Разделитель между нейронами

def forward\_pass(self, x):

"""

Прямой проход через нейронную сеть:

1. Вычисляем выходы нейронов на каждом слое.

2. Используем функцию активации tanh для преобразования линейной комбинации входов в выход.

"""

for i in range(4): # Проходим по входному слою (4 нейрона)

self.n\_y[i] = np.tanh(np.dot(self.w[i], x)) # Для каждого нейрона вычисляем выход, используя функцию активации tanh

self.layer2\_input = np.array([1.0] + [self.n\_y[i] for i in range(4)]) # Добавляем смещение для второго слоя

for i in range(4): # Проходим по второму слою нейронов (4 нейрона)

self.n\_y[i + 4] = np.tanh(np.dot(self.w[i + 4], self.layer2\_input)) # Рассчитываем выход для каждого нейрона второго слоя

# Для третьего слоя (выходного слоя)

self.output\_layer\_input = np.array([1.0] + [self.n\_y[i + 4] for i in range(4)]) # Добавляем смещение для выходного слоя

self.n\_y[8] = np.tanh(np.dot(self.w[8], self.output\_layer\_input)) # Рассчитываем выход для последнего нейрона

def backward\_pass(self, y):

"""

Обратный проход:

1. Рассчитываем ошибку для выходного нейрона.

2. Обновляем ошибки для каждого слоя в сети, используя производную от активационной функции tanh.

"""

self.error\_prime = -(y - self.n\_y[8]) # Рассчитываем ошибку для выходного нейрона

self.derivative = 1.0 - self.n\_y[8] \*\* 2 # Производная от функции активации tanh

self.n\_error[8] = self.error\_prime \* self.derivative # Ошибка для выходного нейрона

for i in range(4): # Процесс для второго слоя нейронов

self.derivative = 1.0 - self.n\_y[i + 4] \*\* 2 # Производная от tanh для каждого нейрона второго слоя

self.n\_error[i + 4] = self.w[8][i + 1] \* self.n\_error[8] \* self.derivative # Ошибка для нейронов второго слоя

for i in range(4): # Процесс для первого слоя нейронов

self.weight\_sum = 0 # Сумма ошибок для каждого нейрона первого слоя

for j in range(4): # Суммируем ошибки для всех нейронов второго слоя

self.weight\_sum += self.n\_error[j + 4] \* self.w[i][j + 1]

self.derivative = 1.0 - self.n\_y[i] \*\* 2 # Производная от tanh для нейронов первого слоя

self.n\_error[i] = self.weight\_sum \* self.derivative # Ошибка для нейронов первого слоя

def adjust\_weights(self, x):

"""

Корректировка весов на основе ошибки:

1. Обновляем веса каждого слоя нейронной сети с помощью градиентного спуска.

2. Используем скорость обучения (LEARNING\_RATE).

"""

for i in range(4): # Обновляем веса для первого слоя

self.w[i] -= (x \* Network.LEARNING\_RATE \* self.n\_error[i]) # Обновление веса для каждого нейрона первого слоя

self.w[i + 4] -= (self.layer2\_input \* Network.LEARNING\_RATE \* self.n\_error[i + 4]) # Обновление весов для второго слоя

self.w[8] -= (self.output\_layer\_input \* Network.LEARNING\_RATE \* self.n\_error[8]) # Обновление веса для выходного слоя

def learning(self):

"""

Основной цикл обучения:

1. Выполняем прямой и обратный проход на каждом шаге.

2. Обновляем веса и проверяем, достигнута ли необходимая точность.

"""

all\_correct = False # Флаг, показывающий, что все ответы сети правильные

while not all\_correct: # Повторяем обучение до тех пор, пока все ответы не будут правильными

self.H = 0 # Сумма ошибок на текущей эпохе

all\_correct = True # Изначально считаем, что все ответы правильные

self.epoch += 1 # Увеличиваем счетчик эпох

np.random.shuffle(self.index\_list) # Случайным образом перемешиваем индексы данных для обучения

for i in self.index\_list: # Проходим по всем примерам в обучающем наборе

self.forward\_pass(self.x\_train[i]) # Прямой проход

self.backward\_pass(self.y\_train[i]) # Обратный проход

self.adjust\_weights(self.x\_train[i]) # Обновление весов

self.H += (self.y\_train[i] - self.n\_y[8]) \*\* 2 # Добавляем квадрат ошибки для текущего примера

self.H /= len(self.x\_train) # Среднеквадратичная ошибка (MSE) на текущей эпохе

self.MSE.append(self.H) # Сохраняем MSE для анализа

# Проверка, все ли ответы правильные

for i in range(len(self.x\_train)):

self.forward\_pass(self.x\_train[i]) # Прямой проход для примера

output = 1 if self.n\_y[8] >= 0 else -1 # Преобразуем выход в бинарное значение (1 или -1)

if self.y\_train[i] != output: # Если выход не совпадает с целевым значением, то ошибка

all\_correct = False # Если хотя бы один ответ неправильный, продолжаем обучение

self.show\_learning() # Выводим текущие веса после обучения

def show\_MSE(self):

"""

Функция для отображения графика MSE (среднеквадратичной ошибки).

"""

plt.title('MSE') # Заголовок графика

plt.xlabel('epoch') # Подпись оси X (номер эпохи)

plt.ylabel('H') # Подпись оси Y (значение MSE)

plt.plot([i for i in range(len(self.MSE))], self.MSE) # Построение графика

plt.show() # Отображение графика

# Запуск обучения нейронной сети

if \_\_name\_\_ == '\_\_main\_\_':

nn = Network(n\_w, x\_train, y\_train) # Создаем объект сети

nn.learning() # Запускаем процесс обучения

nn.show\_MSE() # Показываем график ошибки на протяжении обучения

# Заключение

Работа показала, что алгоритм обратного распространения эффективно минимизирует ошибку и позволяет достичь высокой точности классификации на заданных данных.

В ходе лабораторной работы были изучены такие вопросы как:

1. Изучен и реализован алгоритм обратного распространения ошибки для многослойной нейронной сети.
2. Проведено обучение сети, состоящей из двух скрытых слоев и выходного слоя.
3. Рассчитана и визуализирована среднеквадратичная ошибка (MSE) для анализа качества обучения.
4. Получены веса нейронов, позволяющие корректно классифицировать входные данные из обучающей выборки.