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 №3

«ОБУЧЕНИЕ ДВУХСЛОЙНОЙ ИНС РЕАЛИЗАЦИИ ФУНКЦИИ ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) С ПОМОЩЬЮ АЛГОРИТМА ОБРАТНОГО РАСПРОСТРАНЕНИЯ»

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

**СОДЕРЖАНИЕ**

[Цели работы……………………………………………………………………….](#_30j0zll) 3

[Результат работы…………………………………………………………………..](#_1fob9te) 4

[Заключение………………………………………………………………………...](#_3znysh7) 7

[Листинг кода………………………………………………………………………](#_2et92p0) 8

# Цели работы

Цель:обучение двухслойной нейронной сети для реализации функции "исключающее ИЛИ" (XOR) с использованием алгоритма обратного распространения ошибок.

В рамках данного лабораторного проекта будут рассмотрены следующие задачи:

1. Изучение алгоритма обратного распространения на примере обучения двухслойной нейронной сети для реализации функции XOR
2. Создание блок-схем алгоритма обратного распространения и вспомогательных функций
3. Выполнение ответов на вопросы из предоставленного файла.

# Результат работы

В ходе работы был изучен алгоритм обратного распространения на практическом примере, даны ответы на вопросы из предоставленного файла и подготовлены комментарии к представленному коду (листинг).

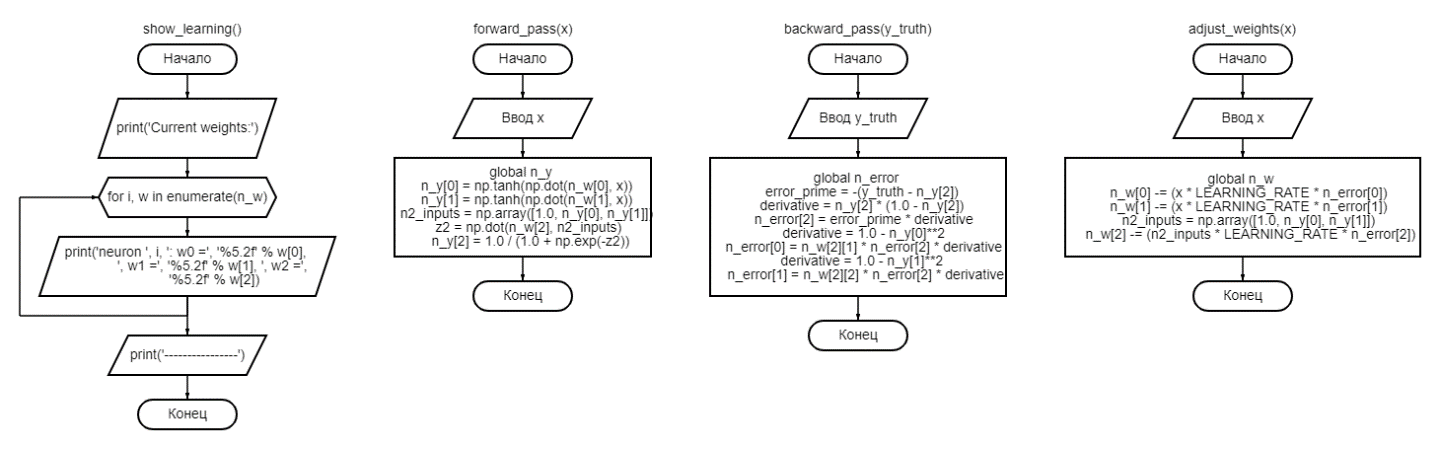


Рисунок 1 - Блок-схема, отражающая работу функций show learning(), forward\_pass(x), backward\_pass(y\_truth), adjust\_weights(x)

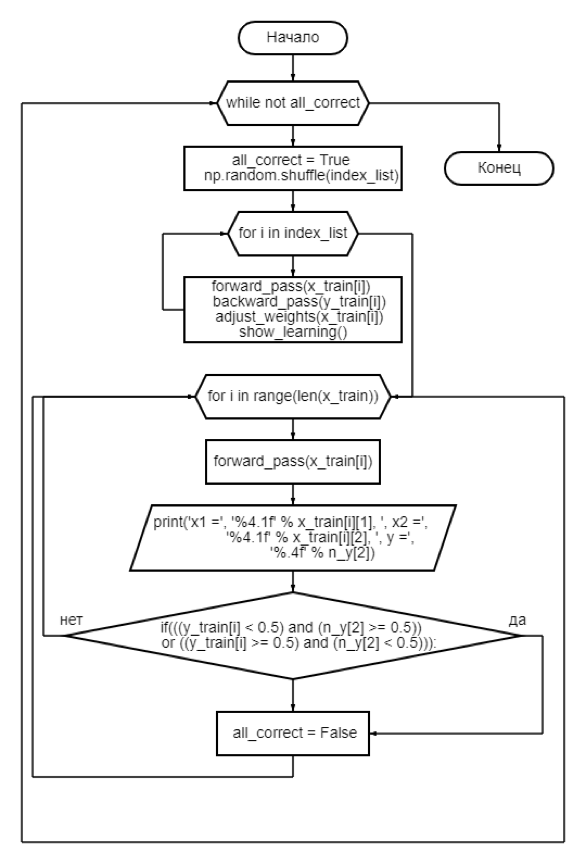


Рисунок 2 - Блок схема, отражающая алгоритм обучения нейронной сети

Ответы на вопросы.

**Вопрос 1.** Мы вызываем np.random.seed вместо просто random.seed Зачем?

Поскольку используется np.random, для установки seed необходимо применять ту же библиотеку, так как модули random и numpy работают независимо, и у каждого из них своя функция seed

**Вопрос 2.** В обучающих примерах мы изменили истинное значение на значение от 0,0 до 1,0, поскольку для выходного нейрона мы используем теперь логистическую сигмоидальную функцию в качестве функции активации, а её выходной диапазон не доходит до -1,0. Почему?

Рассмотрим сигмоидальную функцию. F(x) = 1/(1+exp(-x)). При любых значениях x знаменатель функции будет положительным, а значит и вся функция тоже. Поэтому её выходной диапазон не доходит до -1,0.

# Заключение

В ходе работы был подробно изучен алгоритм обратного распространения на примере обучения двухслойной нейронной сети для реализации функции XOR. Были разработаны блок-схемы, иллюстрирующие принцип работы алгоритма и вспомогательных функций. Кроме того, выполнены ответы на поставленные вопросы, что позволило закрепить теоретические знания и применить их на практике.

# Листинг кода

Листинг 1 – Алгоритм

# Импорт библиотеки numpy для работы с массивами и математическими операциями

import numpy as np

# Установка случайного начального состояния для генератора случайных чисел.

# Это позволяет получать одни и те же результаты при каждом запуске программы.

np.random.seed(3)

# Установка константы скорости обучения (шаг изменения весов во время обучения)

LEARNING\_RATE = 0.1

# Список индексов для упорядочивания или перемешивания обучающих данных

index\_list = [0, 1, 2, 3]

# Входные данные для обучения, где каждый массив представляет отдельный пример (включая "смещение" в виде первого элемента = 1.0)

x\_train = [np.array([1.0, -1.0, -1.0]), # Пример 1

np.array([1.0, -1.0, 1.0]), # Пример 2

np.array([1.0, 1.0, -1.0]), # Пример 3

np.array([1.0, 1.0, 1.0])] # Пример 4

# Ожидаемые результаты (выходы) для каждого соответствующего примера из x\_train

y\_train = [0.0, 1.0, 1.0, 0.0]

# Функция для создания и инициализации весов нейрона

def neuron\_w(input\_count):

# Создание массива весов с нулями. (input\_count + 1, потому что добавляется вес смещения)

weights = np.zeros(input\_count + 1)

# Для каждого входа (кроме смещения) присваиваем случайный вес из диапазона [-1.0, 1.0]

for i in range(1, (input\_count + 1)):

weights[i] = np.random.uniform(-1.0, 1.0)

# Возврат массива весов нейрона

return weights

# Инициализация весов для трех нейронов (два входных и один выходной)

n\_w = [neuron\_w(2), neuron\_w(2), neuron\_w(2)]

# Список для хранения выходных значений каждого нейрона (обновляется во время forward\_pass)

n\_y = [0, 0, 0]

# Список для хранения ошибок (градиентов) каждого нейрона, вычисляемых в backward\_pass

n\_error = [0, 0, 0]

# Функция для отображения текущих значений весов нейронов (используется для отладки)

def show\_learning():

# Вывод информации о весах каждого нейрона

print('Current weights:')

for i, w in enumerate(n\_w):

print('neuron ', i, ': w0 =', '%5.2f' % w[0], # Вывод веса смещения (w0)

', w1 =', '%5.2f' % w[1], # Вывод первого веса

', w2 =', '%5.2f' % w[2]) # Вывод второго веса

print('----------------')

# Функция прямого прохода через сеть (вычисление выходных значений нейронов)

def forward\_pass(x):

global n\_y # Объявляем глобальную переменную, чтобы изменять ее внутри функции

# Вычисление выходного значения первого нейрона с использованием функции активации tanh

n\_y[0] = np.tanh(np.dot(n\_w[0], x))

# Вычисление выходного значения второго нейрона

n\_y[1] = np.tanh(np.dot(n\_w[1], x))

# Создание массива входов для третьего нейрона (включая смещение и выходы двух предыдущих нейронов)

n2\_inputs = np.array([1.0, n\_y[0], n\_y[1]])

# Вычисление линейной комбинации входов для третьего нейрона

z2 = np.dot(n\_w[2], n2\_inputs)

# Применение функции активации сигмоиды для получения выходного значения третьего нейрона

n\_y[2] = 1.0 / (1.0 + np.exp(-z2))

# Функция обратного прохода (вычисление ошибок для каждого нейрона)

def backward\_pass(y\_truth):

global n\_error # Используем глобальную переменную для изменения значений ошибок нейронов

# Вычисление разницы между истинным значением и выходом третьего нейрона

error\_prime = -(y\_truth - n\_y[2])

# Вычисление производной сигмоидальной функции для третьего нейрона

derivative = n\_y[2] \* (1.0 - n\_y[2])

# Вычисление градиента ошибки для третьего нейрона

n\_error[2] = error\_prime \* derivative

# Вычисление производной tanh для первого нейрона

derivative = 1.0 - n\_y[0]\*\*2

# Вычисление градиента ошибки для первого нейрона, используя вес из второго слоя

n\_error[0] = n\_w[2][1] \* n\_error[2] \* derivative

# Вычисление производной tanh для второго нейрона

derivative = 1.0 - n\_y[1]\*\*2

# Вычисление градиента ошибки для второго нейрона

n\_error[1] = n\_w[2][2] \* n\_error[2] \* derivative

# Функция для корректировки весов на основе ошибок (обучение сети)

def adjust\_weights(x):

global n\_w # Глобальные веса нейронов, которые нужно обновить

# Коррекция весов первого нейрона

n\_w[0] -= (x \* LEARNING\_RATE \* n\_error[0])

# Коррекция весов второго нейрона

n\_w[1] -= (x \* LEARNING\_RATE \* n\_error[1])

# Формирование входов для третьего нейрона (смещение и выходы двух предыдущих нейронов)

n2\_inputs = np.array([1.0, n\_y[0], n\_y[1]])

# Коррекция весов третьего нейрона

n\_w[2] -= (n2\_inputs \* LEARNING\_RATE \* n\_error[2])

# Основной цикл обучения сети

all\_correct = False # Флаг, показывающий, сошлась ли сеть

while not all\_correct: # Пока сеть не обучена полностью

all\_correct = True # Предполагаем, что все обучено правильно

# Перемешиваем порядок обучающих примеров для случайности

np.random.shuffle(index\_list)

for i in index\_list: # Проходим по каждому примеру

forward\_pass(x\_train[i]) # Выполняем прямой проход

backward\_pass(y\_train[i]) # Выполняем обратный проход (вычисление ошибок)

adjust\_weights(x\_train[i]) # Корректируем веса

show\_learning() # Отображаем текущие веса для отладки

for i in range(len(x\_train)): # Проверяем, обучена ли сеть

forward\_pass(x\_train[i]) # Выполняем прямой проход

# Печатаем входные данные и результат

print('x1 =', '%4.1f' % x\_train[i][1], ', x2 =',

'%4.1f' % x\_train[i][2], ', y =',

'%.4f' % n\_y[2])

# Проверяем, правильно ли предсказан результат

if(((y\_train[i] < 0.5) and (n\_y[2] >= 0.5))

or ((y\_train[i] >= 0.5) and (n\_y[2] < 0.5))):

all\_correct = False # Если хотя бы один пример неверный, продолжаем обучение