# Практическая работа № 2

# Применение однослойной нейронной сети для решения задач регрессии экспериментальных данных

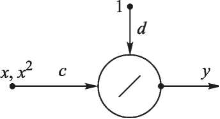
**Цель работы —** изучить возможности однослойных НС с правилом обучения Видроу — Хоффа в задачах регрессии экспериментальных данных.

***Постановка задачи.***

В зависимости от варианта работы с помощью НС с линейной функцией активации аппроксимировать функцию *f(х)* (найти коэффициенты *с, d****)*** по набору ее *N* дискретных значений, заданных равномерно на интервале [*a,b*]со случайными ошибками ******.

Выполнить расчет параметров *с, d* методом наименьших квадратов (МНК) и определить среднеквадратичное отклонение (СКО).

Провести обучение НС по правилу Видроу — Хоффа, экспериментальным путем подобрав оптимальный коэффициент обучения ******.

******

******

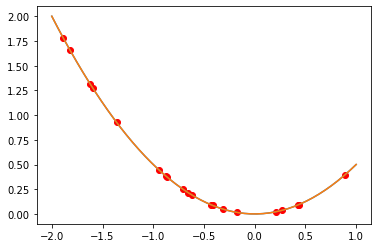
**Исходные данные:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| № варианта | *f(х)* | *c* | *d* | *a* | *b* | *N* | *A* |
| 6 | *cx2 +d* | 0,5 | 0 | -2 | 1 | 20 | 1 |

# Выполнение:

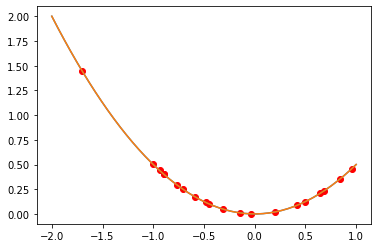
## Решим задачу регрессии через МНК на интервале [-2;1]. Параметр шума отсутствует, ошибка нулевая

c = 0,5; d = 0



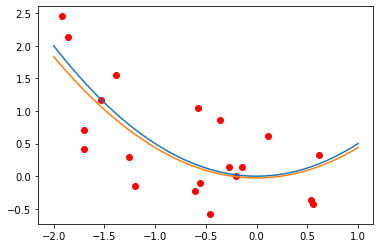
## Решим задачу регрессии через обучение нейрона. Количество эпох = 100, норма обучения = 0,025, Шум = 0

c = 0,499; d =1,49; E = 3.16480



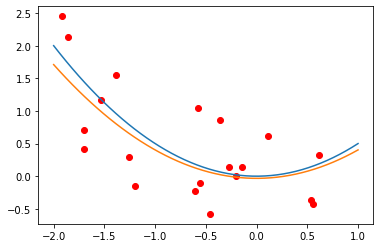
## Решим задачу регрессии через МНК. Параметр шума = 1

c = 0.465; d = -0.026; E = 0.12539



## Решим задачу регрессии через обучение нейрона. Количество эпох =100, норма обучения = 0,025, параметр шума = 1.

C = 0.435; d = -0.033; E = 0,3512994



# Выводы:

В результате проведения работы было исследовано функционирование однослойной НС с правилом обучения Видроу — Хоффа в задачах регрессии экспериментальных данных. Было проведено сравнение данного метода с методом наименьших квадратов.

# Контрольные вопросы

1. Объясните понятие регрессии экспериментальных данных.

Одномерная (простая) линейная регрессия – это метод, используемый для моделирования отношений между одной независимой входной переменной (переменной функции) и выходной зависимой переменной. Модель линейная. Более общий случай – множественная линейная регрессия, где создаётся модель взаимосвязи между несколькими входными переменными и выходной зависимой переменной. Модель остаётся линейной, поскольку выходное значение представляет собой линейную комбинацию входных значений. Также стоит упомянуть полиномиальную регрессию. Модель становится нелинейной комбинацией входных переменных, т. е. среди них могут быть экспоненциальные переменные: синус, косинус и т. п.

2. Поясните суть метода наименьших квадратов.

Задача заключается в нахождении коэффициентов линейной зависимости, при которых функция двух переменных а и b функция двух переменных

принимает наименьшее значение. То есть, при данных а и b сумма квадратов отклонений экспериментальных данных от найденной прямой будет наименьшей. Таким образом, решение сводится к нахождению экстремума функции двух переменных.

3. Сформулируйте нейросетевой подход к задачам регрессии.

МНК дает уравнение прямой, наилучшим образом аппроксимирующей набор данных. Для нахождения такой прямой можно применить другой подход, основанный на использовании НС с линейной функцией активации и обучении, т. е. настройке ее весов с,d по некоторой выборке. Поскольку сеть однослойная, обучение можно реализовать по правилу Видроу — Хоффа.

Каждая эпоха обучения k=1,2,... включает в себя цикл последовательного предъявления всех образцов обучающей выборки i=1,N на вход НС. Предъявление одного обучающего образца внутри эпохи является элементарным шагом обучения, во время которого вектор весовых коэффициентов w=(c,d) корректируется согласно правилу Видроу — Хоффа:

где η — норма обучения, η∈(0,1].

# Приложения

import numpy as np

from copy import copy

import matplotlib.pyplot as plt

import math

from dataclasses import dataclass

from typing import List, Iterable, TYPE\_CHECKING, Callable, Optional

import types

@dataclass

*class* Point:

    x: *float*

    y: *float*

*def* mnk(*data*: *List*[*Iterable*[*List*[*float*]]]) -> (*float*, *float*):

    """

    Возвращает значения c, d, расчитанные по методу наименьших квадратов

    Принимает в себя список точек

    """

    points = [Point(x, y) for x, y in data]

    n = len(points)

    c = (n \* sum((point.x \* point.y for point in points)) - sum((point.x for point in points)) \* sum((point.y for point in points))) \

        / (n \* (sum((math.pow(point.x, 2) for point in points))) - math.pow(sum((point.x for point in points)), 2))

    d = (sum((point.y for point in points)) - c \* sum((point.x for point in points))) / n

    return c, d

*def* af(*self*: "*NeuralNetwork*", *x*: *float*) -> *float*:

    return x

*def* df(*self*: "*NeuralNetwork*", *net*: *float*) -> *float*:

    return 1.0

*def* just\_x(*self*: "*NeuralNetwork*", *x*: *int*) -> *float*:

    return *float*(x)

*class* NeuralNetwork:

*def* \_\_init\_\_(*self*,

*learning\_rate*: *float*,

*activation\_function*: *Callable*[["*NeuralNetwork*", *float*], *float*],

*error\_function*: *Callable*[["*NeuralNetwork*", *int*], *float*] = just\_x,

*c*: *float* = 0.0,

*d*: *float* = 0.0):

        """

        Инициализируем веса

        Переданные функции активации и ее производная должны первым параметром принимать параметр self,

        чтобы превратить их в метод.

        """

        self.weights: List[*float*] = [c, d]  # c, d

        # параметр learning\_rate должен иметь значение от 0 не включительно до 1 включительно

        if not 0.0 < learning\_rate <= 1.0:

            raise *ValueError*('Поставьте параметр learning\_rate от 0 до 1 (необходимо по сабжекту)')

        self.learning\_rate: *float* = learning\_rate

        self.activation\_function = types.MethodType(activation\_function, self)

        self.error\_function = types.MethodType(error\_function, self)

        self.current\_nn\_output: Optional[List[*float*]] = None

*def* get\_net(*self*, *x*: *float*) -> *float*:

        return self.weights[0] \* x + self.weights[1]

*def* \_get\_output\_data(*self*,

*input\_data*: *List*[*float*]) -> List[*float*]:

        return [self.activation\_function(self.get\_net(x)) for x in input\_data]

*def* change\_weights(*self*,

*input\_train\_data*: *List*[*float*],

*errors*: *List*[*float*]) -> None:

        self.weights[0] += sum(self.learning\_rate \*

                               self.error\_function(errors[j]) \*

                               input\_train\_data[j]

                               for j in range(len(input\_train\_data)))

        self.weights[1] += sum(self.learning\_rate \*

                               self.error\_function(errors[j])

                               for j in range(len(input\_train\_data)))

*def* epoch(*self*,

*input\_data*: *List*[*float*],

*expected\_output\_data*: *List*[*float*]) -> *float*:

        self.current\_nn\_output = self.\_get\_output\_data(input\_data)

        errors = [expected\_output\_data[i] - self.current\_nn\_output[i] for i in range(len(self.current\_nn\_output))]

        self.change\_weights([x for x in input\_data], errors)

        return sum([x \*\* 2 for x in errors])

*def* predict(*self*,

*input\_data*: *List*[*float*]) -> List[*float*]:

        return [self.c \* x + self.d for x in input\_data]

    @property

*def* c(*self*):

        return self.weights[0]

    @property

*def* d(*self*):

        return self.weights[1]

if \_\_name\_\_ == '\_\_main\_\_':

    data = []

    epochs = 10

    epoch\_output = []

    a = -2

    b = 1

    c = 0.5

    d = 0

    N = 20

    A = 1

    x\_points = np.random.uniform(low=a, high=b, size=N)

    rA = np.random.normal(scale=A, size=x\_points.size)

    y\_points = c \* x\_points \*\* 2 + d + rA

    \_c, \_d = mnk(*list*(zip([x \*\* 2 for x in x\_points], y\_points)))

    true\_x = np.linspace(a, b)

    true\_y = c \* true\_x \*\* 2 + d

    mnk\_x = np.linspace(a, b)

    mnk\_y = \_c \* mnk\_x \*\* 2 + \_d

    nn = NeuralNetwork(learning\_rate=0.5, activation\_function=af)

    for i in range(epochs):

        nn.epoch(input\_data=[x \*\* 2 for x in x\_points], expected\_output\_data=*list*(y\_points))

        x = np.linspace(a, b, 100)

        y = c \* x \*\* 2 + d

a = -2

b = 1

c = 0.5

d = 0

N = 20

x\_points = np.random.uniform(low=a, high=b, size=N)

y\_points = c \* x\_points \*\* 2 + d

\_c, \_d = mnk(*list*(zip([x \*\* 2 for x in x\_points], y\_points)))

true\_x = np.linspace(a, b)

true\_y = c \* true\_x \*\* 2 + d

mnk\_x = np.linspace(a, b)

mnk\_y = \_c \* mnk\_x \*\* 2 + \_d

plt.plot(true\_x, true\_y)

plt.scatter(x\_points, y\_points, c='red')

plt.plot(mnk\_x, mnk\_y)

print("c = ", \_c, "d = ", \_d)

nn = NeuralNetwork(learning\_rate=0.025, activation\_function=af)

epochs = 100

for i in range(epochs):

    nn.epoch(input\_data=[x \*\* 2 for x in x\_points], expected\_output\_data=y\_points)

nn\_x = np.linspace(a, b)

nn\_y = nn.c \* nn\_x \*\* 2 + nn.d

true\_x = np.linspace(a, b)

true\_y = c \* true\_x \*\* 2 + d

plt.plot(true\_x, true\_y)

plt.plot(nn\_x, nn\_y)

plt.scatter(x\_points, y\_points, c='red')

print(*f*"Среднеквадратичная ошибка: {sum([e \*\* 2 for e in [(c \* x \*\* 2 + d) - (nn.c \* x \*\* 2 + nn.d) for x in x\_points]])}")

print("c =", nn.c, "d =", nn.d)

A = 1

x\_points = np.random.uniform(low=a, high=b, size=N)

rA = np.random.normal(scale=A, size=x\_points.size)

y\_points = c \* x\_points \*\* 2 + d + rA / 2

\_c, \_d = mnk(*list*(zip([x \*\* 2 for x in x\_points], y\_points)))

true\_x = np.linspace(a, b)

true\_y = c \* true\_x \*\* 2 + d

mnk\_x = np.linspace(a, b)

mnk\_y = \_c \* mnk\_x \*\* 2 + \_d

plt.plot(true\_x, true\_y)

plt.scatter(x\_points, y\_points, c='red')

plt.plot(mnk\_x, mnk\_y)

print(*f*"Среднеквадратичная ошибка: {sum([e \*\* 2 for e in [(c \* x \*\* 2 + d) - (\_c \* x \*\* 2 + \_d) for x in x\_points]])}")

print("c =", \_c, "d =", \_d)

A = 1

nn = NeuralNetwork(learning\_rate=0.025, activation\_function=af)

epochs = 100

rA = np.random.normal(scale=A, size=y\_points.size)

for i in range(epochs):

    nn.epoch(input\_data=[x \*\* 2 for x in x\_points], expected\_output\_data=y\_points + rA / 2)

nn\_x = np.linspace(a, b)

nn\_y = nn.c \* nn\_x \*\* 2 + nn.d

true\_x = np.linspace(a, b)

true\_y = c \* true\_x \*\* 2 + d

plt.plot(true\_x, true\_y)

plt.plot(nn\_x, nn\_y)

plt.scatter(x\_points, y\_points, c='red')

print(*f*"Среднеквадратичная ошибка: {sum([e \*\* 2 for e in [(c \* x \*\* 2 + d) - (nn.c \* x \*\* 2 + nn.d) for x in x\_points]])}")

print("c =", nn.c, "d =", nn.d)