Московский авиационный институт
(Национальный исследовательский университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа № 3

по курсу «Нейроинформатика»

Тема: Многослойные сети. Алгоритм обратного распространения ошибки.

Студент: Куликов А.В.
Группа: М80-408Б-17
Преподаватель: Аносова Н.П.
Дата: 6 ноября 2020
Оценка:

Москва, 2020

Цель работы:

Исследование свойств многослойной нейронной сети прямого распространения и алгоритмов ее обучения, применение сети в задачах классификации и аппроксимации функции.

Основные этапы работы:

1. Использовать многослойную нейронную сеть для классификации точек в случае, когда классы не являются линейно разделимыми.
2. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов первого порядка.
3. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов второго порядка.

Оборудование:

Процессор: AMD Ryzen 5 Mobile 3550H

Объем оперативной памяти: 8 Гб

Программное обеспечение:

Python 3.8.5

Сценарий выполнения работы:

Реализация линейной многослойной нейронной сети и алгоритмов обучения (файл multilayer_NN.py)

```
import numpy as np

def sigmoid(t, derivative=False):
    return sigmoid(t) * (1 - sigmoid(t)) if derivative else 1 / (1 + np.exp(-t))

def purelin(t, derivative=False):
    return np.full(t.shape, 1) if derivative else t

def tansig(t, derivative=False):
    return 1 / (np.cosh(t) ** 2) if derivative else np.tanh(t)

def relu(t, derivative=False):
    return np.where(t > 0, 1, 0) if derivative else np.maximum(0, t)

def softmax(t):
    return np.exp(t) / np.exp(t).sum()

def norm2(x):
    return np.sqrt((x**2).sum())
```

```
def mse(A, B):
    return ((A - B) ** 2).sum() / A.shape[0]

def rmse(A, B):
    return np.sqrt(mse(A, B))

def normalize(x, area):
    res = x.copy()
    if res.ndim == 1:
        for i, (min, max) in enumerate(area):
            res[i] = (res[i]) / (max - min)
    else:
        for i, (min, max) in enumerate(area):
            res[:,i] = (res[:,i]) / (max - min)

    return res

eps = 1e-8

class NeuralNetwork:
    def __init__(self, optimizer='traingd', area=None):
        self.area = area
        self.optimizer = optimizer
        self.W = []
        self.b = []
        self.f = []

    def add_layer(self, n_neurons, input_layer=False, activation_function=sigmoid
):
        if input_layer:
            self.prev_layer_n_neurons = n_neurons
        else:
            self.W.append(np.random.sample((self.prev_layer_n_neurons, n_neurons)
))
            self.b.append(np.random.random(n_neurons))
            self.f.append(activation_function)
            self.prev_layer_n_neurons = n_neurons

    def traingd(self, x_train, y_train, epochs):
        n_samples = x_train.shape[0]

        n_layers = len(self.W)
        self.a = [0] * n_layers
        self.o = [0] * n_layers

        for _ in range(epochs):

            dw = [np.zeros(w.shape) for w in self.W]
            db = [np.zeros(b.shape) for b in self.b]

            for x, y in zip(x_train, y_train):
```

```
                y_pred = self.predict(x, training=True)

                for i in range(n_layers-1, -1, -1):
                    o = x if i == 0 else self.o[i-1]

                    if i == n_layers-1:
                        delta = ((y_pred - y) * self.f[-1](self.a[-
1], derivative=True))
                    else:
                        delta = (delta @ self.W[i+1].T) * self.f[i](self.a[i], de
rivative=True)

                    dw[i] += o[np.newaxis].T @ delta[np.newaxis]
                    db[i] += delta

            for i in range(n_layers):
                self.W[i] -= self.lr * dw[i] / n_samples
                self.b[i] -= self.lr * db[i] / n_samples

    def trainrp(self, x_train, y_train, epochs):
        n_layers = len(self.W)
        self.a = [0] * n_layers
        self.o = [0] * n_layers

        eta_minus = 0.5
        eta_plus = 1.2

        dij_max = 50
        dij_min = 1e-6

        prev_dedw = [np.full(w.shape, 0.001) for w in self.W]
        prev_dedb = [np.full(b.shape, 0.001) for b in self.b]

        prev_dw = [np.full(w.shape, 0) for w in self.W]
        prev_db = [np.full(b.shape, 0) for b in self.b]

        dij_w = [np.full(w.shape, 0.1) for w in self.W]
        dij_b = [np.full(b.shape, 0.1) for b in self.b]

        for ep in range(epochs):
            print(f'epoch: {ep} RMSE: {rmse(y_train, self.predict(x_train, traini
ng=True))}')
            # dedw, dedb -- градиенты ошибки по весам и байесам соответственно
            dedw = [np.zeros(w.shape) for w in self.W]
            dedb = [np.zeros(b.shape) for b in self.b]

            for x, y in zip(x_train, y_train):
                y_pred = self.predict(x, training=True)

                delta = ((y_pred - y) * self.f[-1](self.a[-1], derivative=True))
```

```
                dedw[-1] += self.o[-2][np.newaxis].T @ delta[np.newaxis]
                dedb[-1] += delta

                for i in range(n_layers-2, -1, -1):
                    o = x if i == 0 else self.o[i-1]
                    delta = self.f[i](self.a[i], derivative=True) * (delta @ self
.W[i+1].T)
                    dedw[i] += o[np.newaxis].T @ delta[np.newaxis]
                    dedb[i] += delta

            for i in range(n_layers):

                prev_cur_dedw = (dedw[i] * prev_dedw[i]) >= 0
                prev_cur_dedb = (dedb[i] * prev_dedb[i]) >= 0

                dij_w[i] = np.clip(np.where(prev_cur_dedw, eta_plus, eta_minus) *
 dij_w[i], dij_min, dij_max)
                dij_b[i] = np.clip(np.where(prev_cur_dedb, eta_plus, eta_minus) *
 dij_b[i], dij_min, dij_max)

                prev_dw[i] = np.where(prev_cur_dedw, -
np.sign(dedw[i]) * dij_w[i], -prev_dw[i])
                prev_db[i] = np.where(prev_cur_dedb, -
np.sign(dedb[i]) * dij_b[i], -prev_db[i])

                self.W[i] += prev_dw[i]
                self.b[i] += prev_db[i]

                # если произведение производных меньше 0, то "обнуляем" производн
ую
                prev_dedw[i] = np.where(prev_cur_dedw > 0, dedw[i], 0)
                prev_dedb[i] = np.where(prev_cur_dedb > 0, dedb[i], 0)

    def traingdx(self, x_train, y_train, epochs):
        n_samples = x_train.shape[0]

        n_layers = len(self.W)
        self.a = [0] * n_layers
        self.o = [0] * n_layers

        vw = [np.zeros(w.shape) for w in self.W]
        vb = [np.zeros(b.shape) for b in self.b]

        cache_wgrad = [np.zeros(w.shape) for w in self.W]
        cache_bgrad = [np.zeros(b.shape) for b in self.b]

        for ep in range(epochs):
            print(f'epoch: {ep} RMSE: {rmse(y_train, self.predict(x_train, traini
ng=True))}')
            dw = [np.zeros(w.shape) for w in self.W]
            db = [np.zeros(b.shape) for b in self.b]
```

```
            for x, y in zip(x_train, y_train):
                y_pred = self.predict(x, training=True)

                for i in range(n_layers-1, -1, -1):
                    o = x if i == 0 else self.o[i-1]

                    if i == n_layers-1:
                        delta = ((y_pred - y) * self.f[-1](self.a[-
1], derivative=True))
                    else:
                        delta = (delta @ self.W[i+1].T) * self.f[i](self.a[i], de
rivative=True)

                    dw[i] += o[np.newaxis].T @ delta[np.newaxis]
                    db[i] += delta

            beta = 0.8

            for i in range(n_layers):
                db[i] /= n_samples
                dw[i] /= n_samples

                # моментики
                vw[i] = beta * vw[i] + (1 - beta) * dw[i]
                vb[i] = beta * vb[i] + (1 - beta) * db[i]

                # как в ADAGRAD'е
                cache_wgrad[i] += dw[i] ** 2
                cache_bgrad[i] += db[i] ** 2

                self.W[i] -
= self.lr *  1 / (np.sqrt(cache_wgrad[i]) + eps) * vw[i]
                self.b[i] -
= self.lr * 1 / (np.sqrt(cache_bgrad[i]) + eps) * vb[i]

    def trainoss(self, x_train, y_train, epochs):
        n_samples = x_train.shape[0]

        n_layers = len(self.W)
        self.a = [0] * n_layers
        self.o = [0] * n_layers

        prev_step_w = dw = [np.zeros(w.shape) for w in self.W]
        prev_step_b = [np.zeros(b.shape) for b in self.b]

        prev_dw = dw = [np.zeros(w.shape) for w in self.W]
        prev_db = [np.zeros(b.shape) for b in self.b]

        n_weights = 0
        for i in range(n_layers):
```

```
            n_weights += self.W[i].size + self.b[i].size

        for epoch in range(epochs):
            print(f'epoch: {epoch} RMSE: {rmse(y_train, self.predict(x_train, tra
ining=True))}')
            rmse_val = rmse(self.predict(x_train), y_train)

            if rmse_val < eps:
                break

            dw = [np.zeros(w.shape) for w in self.W]
            db = [np.zeros(b.shape) for b in self.b]

            for x, y in zip(x_train, y_train):
                y_pred = self.predict(x, training=True)

                for i in range(n_layers-1, -1, -1):
                    o = x if i == 0 else self.o[i-1]

                    if i == n_layers-1:
                        delta = ((y_pred - y) * self.f[-1](self.a[-
1], derivative=True))
                    else:
                        delta = (delta @ self.W[i+1].T) * self.f[i](self.a[i], de
rivative=True)

                    dw[i] += o[np.newaxis].T @ delta[np.newaxis]
                    db[i] += delta

            for i in range(n_layers):
                dw[i] /= n_samples
                db[i] /= n_samples

                if epoch == 0:
                    dwi = -dw[i]
                    dbi = -db[i]

                    prev_dw[i] = dw[i]
                    prev_db[i] = db[i]

                    prev_step_w[i] = self.lr * dwi
                    prev_step_b[i] = self.lr * dbi

                    self.W[i] += prev_step_w[i]
                    self.b[i] += prev_step_b[i]
                else:
                    yw = dw[i] - prev_dw[i]
                    yb = db[i] - prev_db[i]

                    yy = (yw ** 2).sum() + (yb ** 2).sum()
```

```
                    sy = np.clip((prev_step_w[i] * yw).sum() + (prev_step_b[i] *
yb).sum(), 1e-6, 100)

                    sg = (prev_step_w[i] * dw[i]).sum() + (prev_step_b[i] * db[i]
).sum()

                    yg = (dw[i] * yw).sum() + (db[i] * yb).sum()

                    Ac = -(1 + yy / sy) * sg / sy + yg / sy

                    Bc = sg / sy

                    dwi = -dw[i] + Ac * prev_step_w[i] + Bc * yw
                    dbi = -db[i] + Ac * prev_step_b[i] + Bc * yb

                    prev_dw[i] = dw[i]
                    prev_db[i] = db[i]

                    prev_W = self.W[i]
                    prev_B = self.b[i]

                    def func(alpha):
                        self.W[i] = prev_W + alpha * dwi
                        self.b[i] = prev_B + alpha * dbi

                        res =  rmse(y_train, self.predict(x_train, training=True)
)


                        self.W[i] = prev_W
                        self.b[i] = prev_B

                        return res

                    # простейший линейный поиск минимума
                    min_rmse = 1000
                    best_alpha = 0
                    for a in np.linspace(0, 2, 100):
                        val = func(a)
                        if val < min_rmse:
                            min_rmse = val
                            best_alpha = a

                    prev_step_w[i] = best_alpha * dwi
                    prev_step_b[i] = best_alpha * dbi

                    self.W[i] += prev_step_w[i]
                    self.b[i] += prev_step_b[i]

    def fit(self, x_train, y_train, learning_rate=0.01, epochs=200):
        x_normalized = x_train
        if self.area:
```

```
            x_normalized = normalize(x_train, self.area)

        self.lr = learning_rate
        if self.optimizer == 'traingd':
            self.traingd(x_normalized, y_train, epochs)
        elif self.optimizer == 'trainrp':
            self.trainrp(x_normalized, y_train, epochs)
        elif self.optimizer == 'traingdx':
            self.traingdx(x_normalized, y_train, epochs)
        elif self.optimizer == 'trainoss':
            self.trainoss(x_normalized, y_train, epochs)
        else:
            raise Exception(f"There is no optimizer '{self.optimizer}''")

    def predict(self, x, training=False):
        res = x
        if self.area and not training:
            res = normalize(x, self.area)

        n_layers = len(self.W)
        for i in range(n_layers):
            a = np.dot(res, self.W[i]) + self.b[i]
            self.a[i] = a
            res = self.f[i](a)
            self.o[i] = res

        return res

    def weights(self):
        return self.W

    def biases(self):
        return self.b
```

Задание №1

```
import numpy as np
from multilayer_nn import *

from utils import *

import matplotlib.pyplot as plt

def rotate(point, alpha):
    c = np.cos(alpha)
    s = np.sin(alpha)

    rotation_matrix = np.array([[c, s], [-s, c]])
    return rotation_matrix @ point

def ellipse(a, b, alpha, x0, y0, n):
```

```
    t = np.linspace(0, 2*np.pi, n)

    c = np.cos(alpha)
    s = np.sin(alpha)

    points_x =  a * np.cos(t)
    points_y = b * np.sin(t)

    points_x_rotated = points_x * c + points_y * -s
    points_y_rotated = points_x * s + points_y * c

    points_x_rotated_shifted = points_x_rotated + x0
    points_y_rotated_shifted = points_y_rotated + y0

    return points_x_rotated_shifted, points_y_rotated_shifted

class1_n_points = 60
class2_n_points = 100
class3_n_points = 120

h = 0.025
n_elipse_points = int(2 * np.pi / h) + 1

# вариант 8
class1_points_x, class1_points_y = ellipse(0.2, 0.2, 0, -
0.25, 0.25, n_elipse_points)
class2_points_x, class2_points_y = ellipse(0.7, 0.5, -
np.pi/3, 0, 0, n_elipse_points)
class3_points_x, class3_points_y = ellipse(1, 1, 0, 0, 0, n_elipse_points)

class1_select = np.random.choice(n_elipse_points, class1_n_points, replace=False)
class2_select = np.random.choice(n_elipse_points, class2_n_points, replace=False)
class3_select = np.random.choice(n_elipse_points, class3_n_points, replace=False)

fig, ax = plt.subplots()

ax.scatter(class1_points_x[class1_select], class1_points_y[class1_select], c='red
', s=2)
ax.scatter(class2_points_x[class2_select], class2_points_y[class2_select], c='gre
en', s=2)
ax.scatter(class3_points_x[class3_select], class3_points_y[class3_select], c='blu
e', s=2)
ax.grid()

class1_X = np.column_stack([np.take(class1_points_x, class1_select), np.take(clas
s1_points_y, class1_select)])
class2_X = np.column_stack([np.take(class2_points_x, class2_select), np.take(clas
s2_points_y, class2_select)])
class3_X = np.column_stack([np.take(class3_points_x, class3_select), np.take(clas
s3_points_y, class3_select)])
```

```
class1_Y = np.tile(np.array([1, 0, 0]), (class1_n_points, 1))
class2_Y = np.tile(np.array([0, 1, 0]), (class2_n_points, 1))
class3_Y = np.tile(np.array([0, 0, 1]), (class3_n_points, 1))

n_samples = class1_n_points + class2_n_points + class3_n_points
permutation = np.random.permutation(n_samples)

X = np.take(np.vstack([class1_X, class2_X, class3_X]), permutation, axis=0)
Y = np.take(np.vstack([class1_Y, class2_Y, class3_Y]), permutation, axis=0)

x_train, y_train, x_test, y_test, x_valid, y_valid = split_dataset(X, Y, 0.2, 0.1
)

classifier = NeuralNetwork(optimizer='trainrp')

classifier.add_layer(2, input_layer=True)
classifier.add_layer(20, activation_function=tansig)

classifier.add_layer(3, activation_function=purelin)

classifier.fit(x_train, y_train, learning_rate=0.4, epochs=1500)

print("WEIGHTS:")
print(classifier.weights())
print("BIASES:")
print(classifier.biases())

y_pred = np.where(classifier.predict(x_train) > 0.5, 1, 0)
correctly_classified = row_wise_equal(y_pred, y_train)
print(f'{correctly_classified}/{x_train.shape[0]} ({correctly_classified / x_trai
n.shape[0]}) points of train set are classified correctly!')

y_pred = np.where(classifier.predict(x_test) > 0.5, 1, 0)
correctly_classified = row_wise_equal(y_pred, y_test)
print(f'{correctly_classified}/{x_test.shape[0]} ({correctly_classified / x_test.
shape[0]}) points of test set are classified correctly!')

y_pred = np.where(classifier.predict(x_valid) > 0.5, 1, 0)
correctly_classified = row_wise_equal(y_pred, y_valid)
print(f'{correctly_classified}/{x_valid.shape[0]} ({correctly_classified / x_vali
d.shape[0]}) points of validation set are classified correctly!')

x = np.arange(-1.2, 1.2, h)
y = np.arange(-1.2, 1.2, h)

dim_n_points = x.shape[0]

x_, y_ = np.meshgrid(x, y)

points = np.vstack([x_.flatten(), y_.flatten()]).T
```

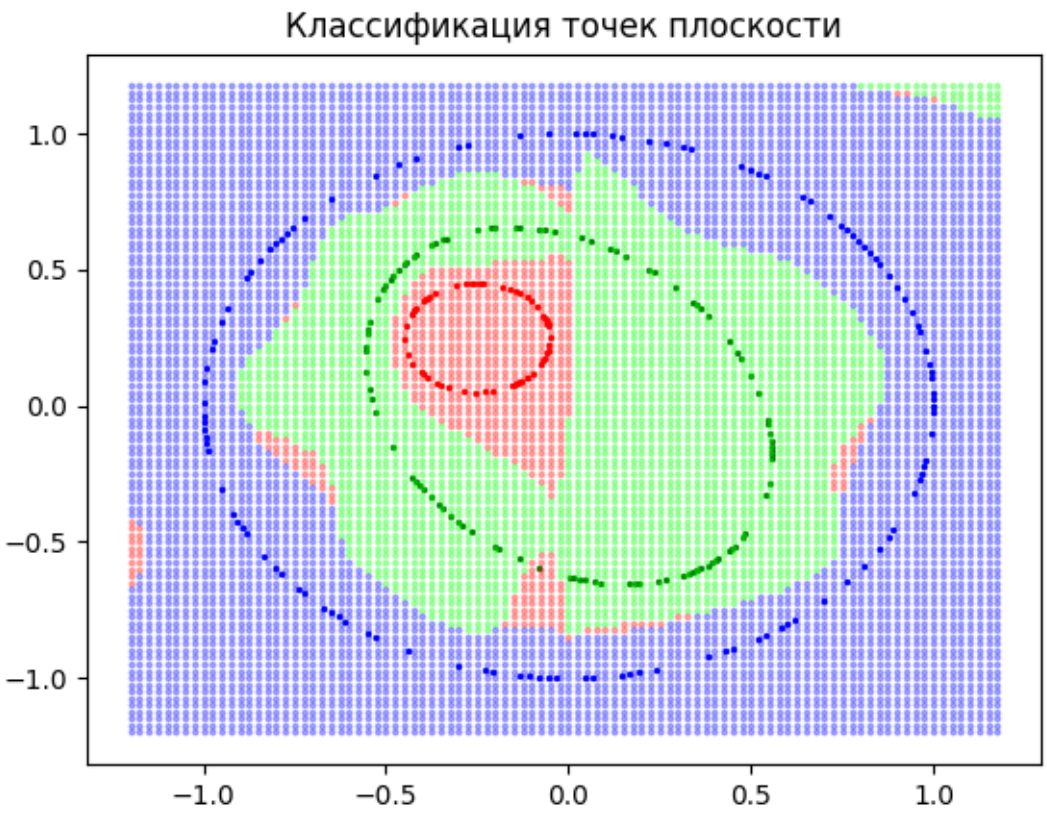
```
pred = np.where(classifier.predict(points) > 0.5, 1, 0)

class_labels = np.argmax(pred, axis=1)

class1_grid_points = points[class_labels == 0]
class2_grid_points = points[class_labels == 1]
class3_grid_points = points[class_labels == 2]

x_class1_grid = class1_grid_points[:,0]
y_class1_grid = class1_grid_points[:,1]

x_class2_grid = class2_grid_points[:,0]
y_class2_grid = class2_grid_points[:,1]

x_class3_grid = class3_grid_points[:,0]
y_class3_grid = class3_grid_points[:,1]

ax.scatter(x_class1_grid, y_class1_grid, color=(1, 0, 0, 0.3), s=3)
ax.scatter(x_class2_grid, y_class2_grid, color=(0, 1, 0, 0.3), s=3)
ax.scatter(x_class3_grid, y_class3_grid, color=(0, 0, 1, 0.3), s=3)

ax.grid()

ax.set_title('Классификация точек плоскости')
plt.savefig('1.png')
plt.show()
```

Вывод программы:

```
…

epoch: 1499 RMSE: 0.254732628513507

WEIGHTS:

[array([[-1.10692498e+00,  2.34186869e+00,  3.01916976e-02,

        -1.70281480e+00, -1.22580140e-01, -3.05339770e+00,

         1.33026455e+00, -8.08628806e+00, -5.06716287e-01,

         1.58083524e+00, -5.24300793e-01,  6.56318089e-02,

         1.06644491e+00,  1.61252273e+00,  3.51524789e-01,

        -8.65830362e+01,  5.78514741e-03, -1.44850219e-01,

        -1.10509350e-02,  4.44549299e+00],

       [ 1.00117598e-01, -1.96944547e+00,  2.20353793e-01,

        -4.38581008e+00, -5.30454028e-02,  3.72672455e+00,

        -1.56892915e-01, -2.48006192e+00,  7.60062747e-04,
```

```
         2.90622568e+00, -1.38634988e-01,  3.03954624e-02,

        -2.01260643e+00,  2.22973642e+00,  1.38011254e-01,

         2.37031773e+00,  2.13689521e-01, -5.11665029e-01,

        -1.35765471e-01, -2.53871176e+00]]), array([[ 0.55531101,  0.21579871,
0.45830766],

       [-0.15841313, -0.92945903,  1.14336673],

       [-2.52544693,  1.64796235, -2.20190512],

       [-0.50027953,  0.91444832, -0.38565499],

       [ 0.03755916, -0.59670182,  2.02095679],

       [ 0.6175317 , -0.75625486,  0.17758165],

       [ 0.13291931,  1.26953331, -0.58335446],

       [-0.42297894,  0.52861523, -0.0931848 ],

       [ 0.34591063, -0.29543275,  0.24211673],

       [ 0.46009838, -1.96104936,  1.49843791],

       [-0.85728716, -0.15896036,  0.45395796],

       [ 0.15943096,  0.64242035, -0.44346207],

       [ 1.0869408 , -1.63265494,  0.66969435],

       [-0.59501217, -0.00763323,  0.62586606],

       [ 0.28631296,  0.23337207,  0.35003943],

       [ 0.23372211, -0.27983712,  0.04278729],

       [-0.4553773 ,  6.52571919, -3.59661907],

       [-0.65978515, -0.10253131,  0.11765914],

       [ 0.34469256,  0.33216011,  0.79771309],

       [-0.28655534,  1.18781664, -0.96849108]])]

BIASES:

[array([-1.06831019e-01, -2.49103931e+00,  1.66187680e-02, -3.87018957e-01,

         6.66013505e-01, -7.65234976e-01,  1.65191180e-01, -2.89369019e+00,

         2.12103077e-01, -2.47868370e+00,  1.00046349e-02, -3.60932409e+02,

         9.01273503e-01, -6.27378914e-01,  4.75922456e-01, -2.89009811e-01,

        -1.33477498e-01,  1.84075497e-01,  4.84146370e-01,  4.51383343e+00]),
array([-0.18889562, -0.4421576 ,  0.6194048 ])]

196/196 (1.0) points of train set are classified correctly!
```

```
55/56 (0.9821428571428571) points of test set are classified correctly!

28/28 (1.0) points of validation set are classified correctly!
```



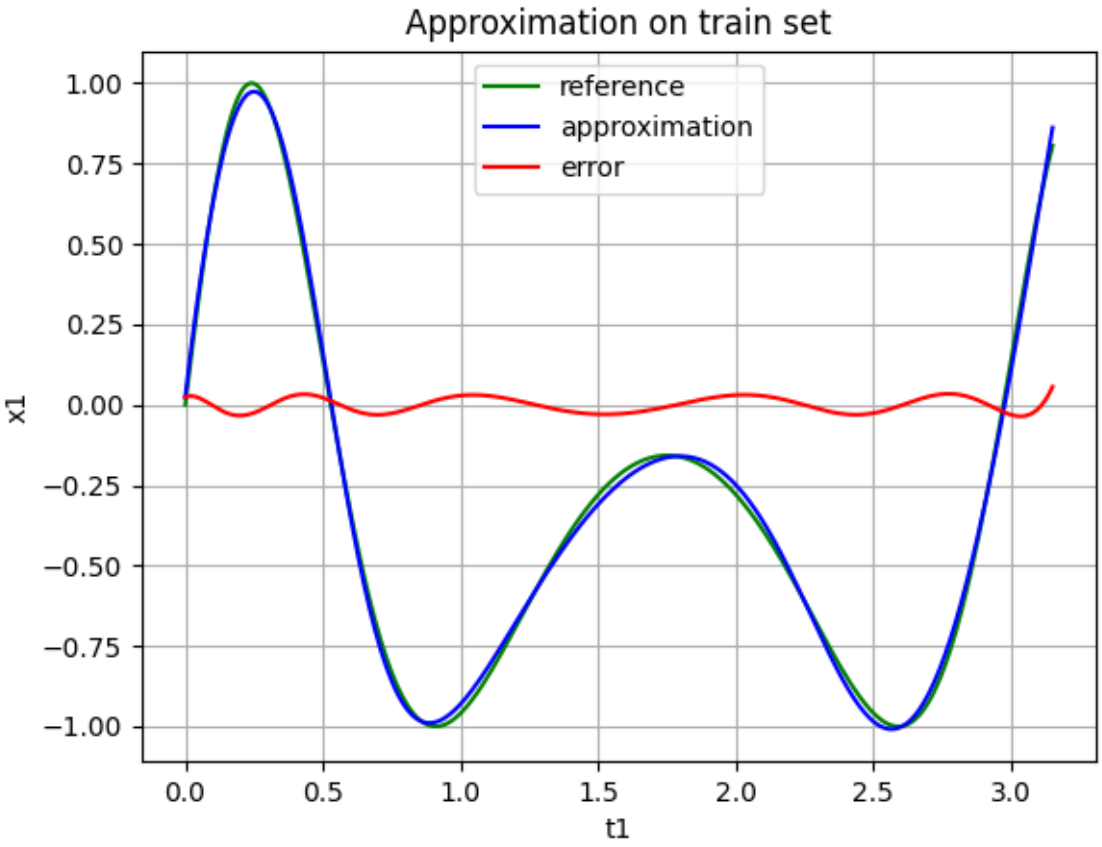
Задание №2

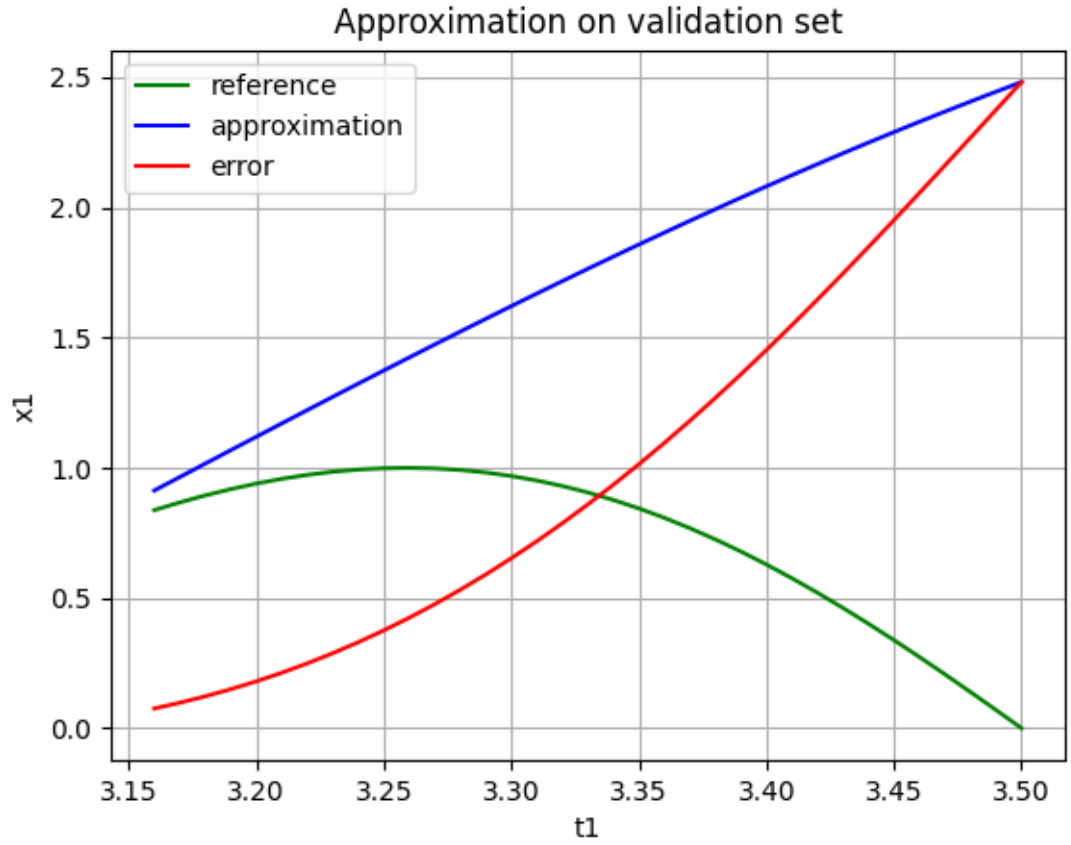
```
import numpy as np
import matplotlib.pyplot as plt

from multilayer_nn import *
from utils import *

def phi(t):
    return np.sin(-2*t*t + 7*t)

t_begin = 0
t_end = 3.5
h = 0.01

n_points = int((t_end - t_begin) / h) + 1
x = np.linspace(t_begin, t_end, n_points).reshape((-1,1))

y = phi(x)

x_train, y_train, x_test, y_test, x_valid, y_valid = split_dataset(x, y, 0.0, 0.1
)
```

```
approximator = NeuralNetwork(optimizer='traingdx')

approximator.add_layer(1, input_layer=True)
approximator.add_layer(10, activation_function=tansig)
approximator.add_layer(1, activation_function=purelin)

approximator.fit(x_train, y_train, learning_rate=2, epochs=9000)

print("WEIGHTS:")
print(approximator.weights())
print("BIASES:")
print(approximator.biases())

y_pred = approximator.predict(x_train)
print(f'RMSE on train set: {rmse(y_train, y_pred)}')

error = y_pred - y_train

fig, ax = plt.subplots()
ax.plot(x_train, y_train, 'g', label='reference')
ax.plot(x_train, y_pred, 'b', label='approximation')
ax.plot(x_train, error, 'r', label='error')

ax.set(xlabel='t1', ylabel='x1', title='Approximation on train set')
ax.grid()
ax.legend()

plt.savefig('2.png')
plt.show()

y_pred = approximator.predict(x_valid)
print(f'RMSE on validation set: {rmse(y_valid, y_pred)}')

error = y_pred - y_valid

fig, ax = plt.subplots()
ax.plot(x_valid, y_valid, 'g', label='reference')
ax.plot(x_valid, y_pred, 'b', label='approximation')
ax.plot(x_valid, error, 'r', label='error')

ax.set(xlabel='t1', ylabel='x1', title='Approximation on validation set')
ax.grid()
ax.legend()

plt.savefig('3.png')
plt.show()
```

Вывод программы:

```
WEIGHTS:

[array([[ 2.25519508,  3.12822284, -2.30310663, -2.08897788,  1.28043563,

        -0.90086104, -1.65677596, -2.27271799,  1.29389652, -1.28775643]]),
array([[-1.5160284 ],

       [-2.72361944],

       [-1.65175989],

       [-1.70214951],

       [-1.23670403],

       [-1.50396104],

       [-3.34670612],

       [-1.68916531],

       [-0.49248688],

       [-1.55497855]])]

BIASES:

[array([-5.3323364 , -1.53969777, -0.63141305, -0.71822547,  2.02172584,

       -0.68328741,  5.15180215, -0.62634476,  1.34272394,  1.08992946]),
array([-1.57042693])]

RMSE on train set: 0.02259888712151485

RMSE on validation set: 1.2555591847630128
```


Approximation on train set
reference
approximation
error
x1
t1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0


Approximation on validation set
reference
approximation
error
x1
t1
2.5
2.0
1.5
1.0
0.5
0.0
3.15
3.20
3.25
3.30
3.35
3.40
3.45
3.50

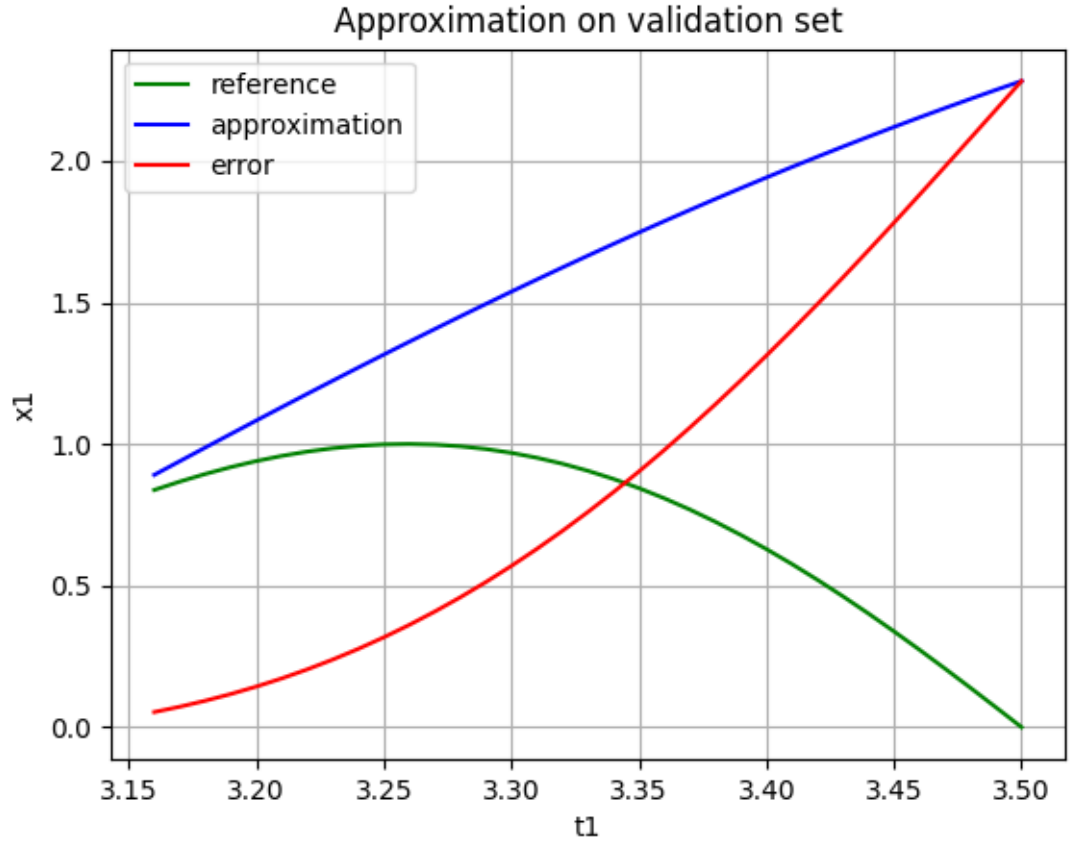
Задание №3

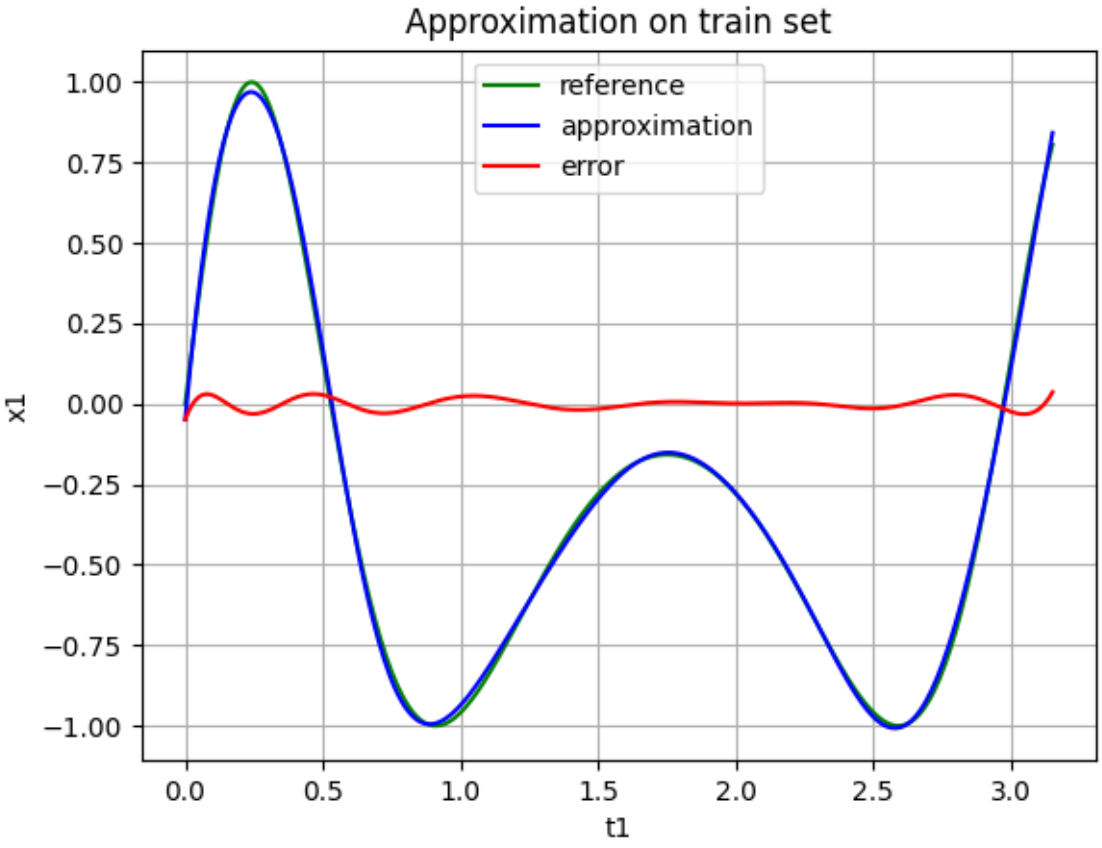
```
import numpy as np
import matplotlib.pyplot as plt

from multilayer_nn import *
from utils import *

def phi(t):
    return np.sin(-2*t*t + 7*t)

t_begin = 0
t_end = 3.5
h = 0.01

n_points = int((t_end - t_begin) / h) + 1
x = np.linspace(t_begin, t_end, n_points).reshape((-1,1))

y = phi(x)

x_train, y_train, x_test, y_test, x_valid, y_valid = split_dataset(x, y, 0.0, 0.1
)

approximator = NeuralNetwork(optimizer='trainoss')

approximator.add_layer(1, input_layer=True)
approximator.add_layer(10, activation_function=tansig)
approximator.add_layer(1, activation_function=purelin)

approximator.fit(x_train, y_train, learning_rate=0.2, epochs=1000)

print("WEIGHTS:")
print(approximator.weights())
print("BIASES:")
print(approximator.biases())

y_pred = approximator.predict(x_train)
print(f'RMSE on train set: {rmse(y_train, y_pred)}')

error = y_pred - y_train

fig, ax = plt.subplots()
ax.plot(x_train, y_train, 'g', label='reference')
ax.plot(x_train, y_pred, 'b', label='approximation')
ax.plot(x_train, error, 'r', label='error')

ax.set(xlabel='t1', ylabel='x1', title='Approximation on train set')
ax.grid()
ax.legend()

plt.savefig('4.png')
```

```
plt.show()

y_pred = approximator.predict(x_valid)
print(f'RMSE on validation set: {rmse(y_valid, y_pred)}')

error = y_pred - y_valid

fig, ax = plt.subplots()
ax.plot(x_valid, y_valid, 'g', label='reference')
ax.plot(x_valid, y_pred, 'b', label='approximation')
ax.plot(x_valid, error, 'r', label='error')

ax.set(xlabel='t1', ylabel='x1', title='Approximation on validation set')
ax.grid()
ax.legend()

plt.savefig('5.png')
plt.show()
```

Вывод программы:

```
WEIGHTS:
[array([[-2.75800605,  1.00360221,  3.29886962, -3.55298838, -1.092554  ,
        -1.66780477,  0.83672884,  1.75077141, -1.42015736,  5.01231704]]),
array([[ 1.078872  ],
       [ 0.68529593],
       [-2.35116654],
       [-0.13094346],
       [-1.67990375],
       [-3.19247372],
       [ 0.33565885],
       [-0.83030163],
       [-0.53610836],
       [ 2.04889734]])]
BIASES:
[array([ 6.79325011, -0.05030957, -1.71652344,  5.54528985,  0.90676715,
        5.07337943, -0.06520697, -3.46589284, -0.99746615,  0.50264769]),
array([-0.93052901])]
RMSE on train set: 0.016690766269254118
RMSE on validation set: 1.1399546347902756
```


Approximation on validation set
reference
approximation
error
x1
t1
2.0
1.5
1.0
0.5
0.0
3.15
3.20
3.25
3.30
3.35
3.40
3.45
3.50


Approximation on train set
reference
approximation
error
x1
t1
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Выводы:

Многослойная нейронная сеть — нейронная сеть, состоящая из входного, выходного и расположенными между ними скрытыми слоями нейронов.

Помимо входного и выходного слоев эти нейронные сети содержат промежуточные, скрытые слои. Такие сети обладают гораздо большими возможностями, чем однослойные нейронные сети

Например, с помощью трехслойной НС, в которой скрытые слои содержат элементы с сигмоидальными функциями, а выходной слой — элементы с линейной активационной функцией, можно с любой наперед заданной точностью аппроксимировать любую функцию.

Для обучения таких нейронных сетей применяется алгоритм обратного распространения ошибки. Причем алгоритм обратного распространения ошибки может использовать различные оптимизаторы.