|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |
| --- | --- |

ФАКУЛЬТЕТ Информатики, искусственного интеллекта и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**ЛАБОРАТОРНАЯ РАБОТА № 2**

***Разработка многослойного персептрона на основе обратного распространения ошибки***

**ПО КУРСУ:**

***«Теория искусственных нейронных сетей»***

Студент группы ИУ9-72Б: *Терентьева А.С.*

Преподаватель: *Каганов Ю.Т.*

*Москва, 2023 г.*

1. **Цель:**

Изучение многослойного персептрона, исследование его работы на основе использования различных методов оптимизации и целевых функций.

1. **Постановка задачи:**
2. Реализовать на языке высокого уровня многослойный персептрон и проверить его работоспособность на примере данных, выбранных из MNIST dataset.
3. Исследовать работу персептрона на основе использования различных целевых функций. (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера).
4. Исследовать работу многослойного персептрона с использованием различных методов оптимизации (градиентный, Флетчера-Ривза (FR), Бройдена-Флетчера-Гольдфарба-Шенно (BFGS))
5. **Реализация**

main.py:

import os

import numpy as np

from src.methods import gradient, FletcherReeves, bfgs, plt

import gzip

import struct

import random

n\_pixels = 784 # 28\*28

def relu(x):

return x if x > 0 else 0

def drelu(x):

return 1 if x > 0 else 0

def softmax(xs):

maxX = max(xs)

exp\_values = np.exp(xs - maxX)

sum\_exp\_values = np.sum(exp\_values)

return [ex / sum\_exp\_values for ex in exp\_values]

# среднквадратичная ф.п

def mse(y0, y):

return 1 / 2 \* (y0 - y) \*\* 2

def dmse(y0, y):

return y - y0

# перекрестная энтропия

def cross\_entr(y0, y):

"""epsilon = 1e-15

y = np.clip(y, epsilon, 1 - epsilon) # чтобы избежать деления на ноль"""

return -(y0 \* np.log(y) + (1 - y0) \* np.log(1 - y))

def dcross\_entr(y0, y):

'''epsilon = 1e-15

y = np.clip(y, epsilon, 1 - epsilon)'''

return -(y0 / y - (1 - y0) / (1 - y))

# дивергенция Кульбака-Лейблера

def kl\_divergence(y0, y):

return 0 if y0 == 0 else y0 \* np.log(y0 / y)

def dkl\_divergence(y0, y):

return np.log(y0 / y) + 1

class Layer:

def \_\_init\_\_(self, n\_neurons, n\_input, activation, derivative, lr):

self.n\_neurons = n\_neurons

self.n\_input = n\_input

# self.w = np.array([[1/n\_input for \_ in range(n\_input)] for \_ in range(n\_neurons)])

# self.w = np.array([[0.01 for \_ in range(n\_input)] for \_ in range(n\_neurons)])

self.w = np.array([[random.uniform(1 / (2 \* n\_input), 2 / n\_input)

for \_ in range(n\_input)] for \_ in range(n\_neurons)])

self.activation = activation

self.derivative = derivative

self.XW = []

self.out = []

self.lr = lr

self.prev\_grad = [] #FR

self.prev\_d = [] # FR

self.H = np.eye(n\_neurons \* n\_input) #BFGS

def forward(self, x):

self.XW = np.dot(self.w, x)

if self.activation == softmax:

self.out = self.activation(self.XW)

else:

self.out = [self.activation(xw) for xw in self.XW]

return self.out

class Perceptron:

def \_\_init\_\_(self, x\_train, y\_train):

self.layers = []

self.loss = mse

self.dloss = dmse

self.out = []

self.x\_train = x\_train

self.y\_train = y\_train

self.n\_train = len(x\_train)

self.lastDelta = []

def add\_layer(self, n\_neurons, n\_input=-1, func\_act=relu, dfunc\_act=drelu, lr=0.1):

if len(self.layers) == 0:

# создание 0го - входного слоя

l0 = Layer(n\_input, 0, func\_act, dfunc\_act, lr)

self.layers.append(l0)

n\_input = (

n\_input if n\_input > 0 else len(self.layers[-1].w)

) # кол-во нейронов предыдущего слоя

self.layers.append(Layer(n\_neurons, n\_input, func\_act, dfunc\_act, lr))

def set\_loss(self, lossfunc\_name):

if lossfunc\_name == "cross\_entr":

self.loss = cross\_entr

elif lossfunc\_name == "KL":

self.loss = kl\_divergence

def forward(self, x):

out = x[:]

# self.layers[0].out = out[:]

i = 1

for l in self.layers[1:]:

out = l.forward(out)

# print(i, out)

i += 1

self.out = out

return out

def gradient(self):

gradient(self)

def FletcherReeves(self):

FletcherReeves(self)

def BFGS(self):

bfgs(self)

def find\_answ(res):

num = 0

min = 1

for i in range(len(res)):

if abs(1 - res[i]) < min:

min = abs(1 - res[i])

num = i

return num

def countErr(self):

all = 0

for i in range(self.n\_train): # обучение на n тестах

e = 0

res = self.forward(self.x\_train[i])

for j in range(10): # 10 нейронов

e += self.loss(self.y\_train[i][j], res[j])

e /= 10 # среднее

all += e

all /= self.n\_train

return all

def checkCorrectness(self, x, y):

# проверка ответов

num = len(x)

correct\_num = 0

for i in range(num):

res = [0] \* 10

mas = [0] \* 10

res = self.forward(x[i])

for j in range(10):

mas[j] = round(res[j], 2)

predicted = np.argmax(res)

expected = np.argmax(y[i])

if expected == predicted:

correct\_num += 1

if i > num - 5:

print(mas)

print(y[i])

print(expected, "--->", predicted, "\n")

print(correct\_num / num \* 100, "%% correctness")

def create\_Y\_ans(Y\_first):

Y\_res = []

for y in Y\_first:

mas = np.zeros(10)

mas[y] = 1

Y\_res.append(mas)

return Y\_res

data\_folder = os.path.join(os.getcwd(), "data")

# load compressed MNIST gz files and return numpy arrays

def load\_data(filename, label=False):

with gzip.open(filename) as gz:

struct.unpack("I", gz.read(4))

n\_items = struct.unpack(">I", gz.read(4))

if not label:

n\_rows = struct.unpack(">I", gz.read(4))[0]

n\_cols = struct.unpack(">I", gz.read(4))[0]

res = np.frombuffer(

gz.read(n\_items[0] \* n\_rows \* n\_cols), dtype=np.uint8)

res = res.reshape(n\_items[0], n\_rows \* n\_cols)

else:

res = np.frombuffer(gz.read(n\_items[0]), dtype=np.uint8)

res = res.reshape(n\_items[0], 1)

return res

# note we also shrink the intensity values (X) from 0-255 to 0-1. This helps the model converge faster.

X\_train = load\_data(os.path.join(

data\_folder, "train-images.gz"), False) / 255.0

Y\_train = load\_data(os.path.join(

data\_folder, "train-labels.gz"), True).reshape(-1)

X\_test = load\_data(os.path.join(data\_folder, "test-images.gz"), False) / 255.0

Y\_test = load\_data(os.path.join(

data\_folder, "test-labels.gz"), True).reshape(-1)

train\_len = len(X\_train)

n\_tests = 500

X\_first = X\_train[:n\_tests]

Y\_first = Y\_train[:n\_tests]

Y\_res = create\_Y\_ans(Y\_first)

perc1 = Perceptron(X\_first, Y\_res)

perc1.add\_layer(n\_input=n\_pixels, n\_neurons=10, lr=0.01)

W1 = [row[:] for row in perc1.layers[-1].w]

#perc1.add\_layer(n\_neurons=10, lr=0.01)

perc1.add\_layer(n\_neurons=10, func\_act=softmax, lr=0.01)

W2 = [row[:] for row in perc1.layers[-1].w]

perc1.set\_loss('cross\_entr')

#perc1.set\_loss('KL')

#perc1.gradient()

perc1.FletcherReeves()

#perc1.BFGS()

perc1.checkCorrectness(X\_first, Y\_res)

Y\_res2 = create\_Y\_ans(Y\_test)

#perc1.checkCorrectness(X\_test[:500], Y\_res2[:500])

plt.legend()

plt.show()

methods.py:

import numpy as np

import matplotlib.pyplot as plt

def gradient(perc):

x = perc.x\_train

y = perc.y\_train

num = perc.n\_train

epohs = []

errors = []

for step in range(5):

if step % 1 == 0:

print(step)

epohs.append(step)

errors.append(perc.countErr())

for i in range(num): # обучение на n тестах

perc.layers[0].out = x[i]

out = perc.forward(x[i])

# последний слой

l = perc.layers[-1]

lastDelta = [perc.dloss(y[i][j], l.out[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(lastDelta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[-2].out])

perc.layers[-1].w += l.lr \* gradient

# скрытые слои

# for l in perc.layers[1:-1:-1]:

for l\_i in range(len(perc.layers) - 2, 0, -1):

l = perc.layers[l\_i]

# средневзвешенная delta выходов

sum = np.dot(perc.layers[l\_i + 1].w.T, lastDelta)

delta = [sum[j] \* l.derivative(l.XW[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(delta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[l\_i - 1].out])

perc.layers[l\_i].w += l.lr \* gradient

plt.plot(epohs, errors, label=f"{perc.loss.\_\_name\_\_}")

#plt.show()

def FletcherReeves(perc):

x = perc.x\_train

y = perc.y\_train

num = perc.n\_train

epohs = []

errors = []

for step in range(5):

if step % 1 == 0:

print(step)

epohs.append(step)

errors.append(perc.countErr())

for i in range(num): # обучение на n тестах

perc.layers[0].out = x[i]

out = perc.forward(x[i])

for l\_i in range(len(perc.layers) - 1, 0, -1):

l = perc.layers[l\_i]

if (l\_i == len(perc.layers) - 1):

# последний слой

lastDelta = [perc.dloss(y[i][j], l.out[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(lastDelta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[-2].out])

#perc.layers[-1].w += l.lr \* gradient

else:

# скрытые слои

sum = np.dot(perc.layers[l\_i + 1].w.T, lastDelta)

delta = [sum[j] \* l.derivative(l.XW[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(delta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[l\_i - 1].out])

#perc.layers[l\_i].w += l.lr \* gradient

if len(l.prev\_grad) == 0 or len(l.prev\_d) == 0:

l.prev\_grad = [row[:] for row in gradient]

l.prev\_d = l.lr \* gradient

W = [[min(gradient[i][j] \*\* 2 / (l.prev\_grad[i][j] \*\* 2 + 1e-6), 1) for j in range(l.n\_input)] for i in range (l.n\_neurons)]

d = l.prev\_grad - np.array(W) \* l.prev\_d

l.w += l.lr \* d

l.prev\_grad = [row[:] for row in gradient]

l.prev\_d = np.array([row[:] for row in d])

plt.plot(epohs, errors, label=f"{perc.loss.\_\_name\_\_}")

#plt.show()

def bfgs(perc):

x = perc.x\_train

Y = perc.y\_train

num = perc.n\_train

epohs = []

errors = []

for step in range(4):

if step % 1 == 0:

print(step)

epohs.append(step)

errors.append(perc.countErr())

for i in range(num): # обучение на n тестах

perc.layers[0].out = x[i]

out = perc.forward(x[i])

for l\_i in range(len(perc.layers) - 1, 0, -1):

l = perc.layers[l\_i]

if (l\_i == len(perc.layers) - 1):

# последний слой

lastDelta = [perc.dloss(Y[i][j], l.out[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(lastDelta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[-2].out])

#perc.layers[-1].w += l.lr \* gradient

else:

# скрытые слои

sum = np.dot(perc.layers[l\_i + 1].w.T, lastDelta)

delta = [sum[j] \* l.derivative(l.XW[j]) for j in range(l.n\_neurons)]

lastDelta = np.array(delta)

gradient = -np.dot(np.transpose([lastDelta]), [perc.layers[l\_i - 1].out])

#perc.layers[l\_i].w += l.lr \* gradient

if len(l.prev\_grad) == 0:

l.prev\_grad = np.array([row[:] for row in gradient])

perc.layers[l\_i].w += l.lr \* gradient

else:

p = -np.dot(l.H, gradient.reshape(-1, 1))

I = np.eye(l.n\_neurons \* l.n\_input)

s = l.lr \* p

l.w += s.reshape(l.n\_neurons, l.n\_input)

y = gradient - l.prev\_grad

#s = s.reshape(-1, 1)

y = y.reshape(-1, 1)

rho = 1 / np.dot(y.T, s)

A = I - rho \* np.dot(s, y.T)

B = I - rho \* np.dot(y, s.T)

l.H = np.dot(A, np.dot(l.H, B)) + rho \* np.dot(s, s.T)

l.prev\_grad = [row[:] for row in gradient]

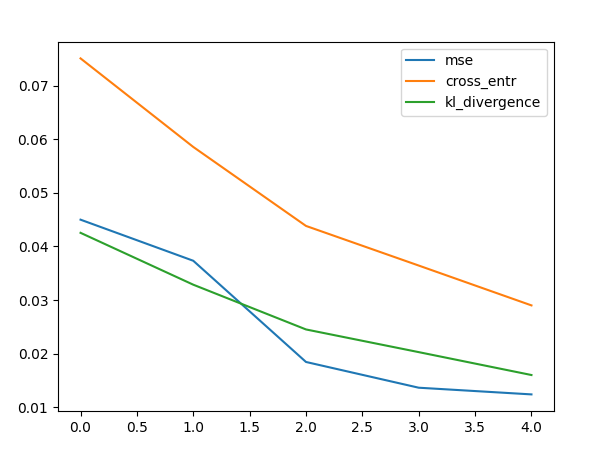
plt.plot(epohs, errors, label=f"{perc.loss.\_\_name\_\_}")

#plt.show()

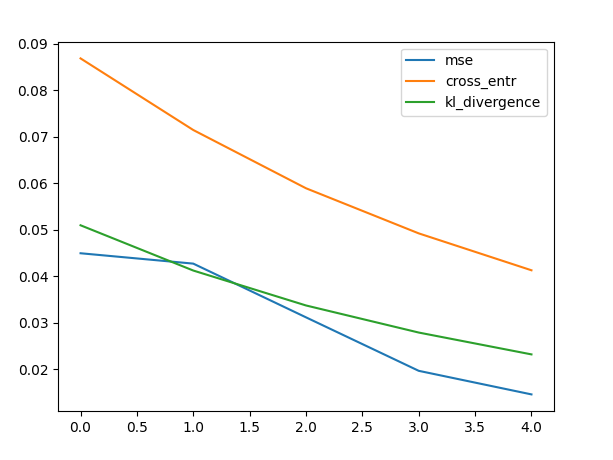
1. **Результат работы**

Графики зависимости ошибки от количества эпох:

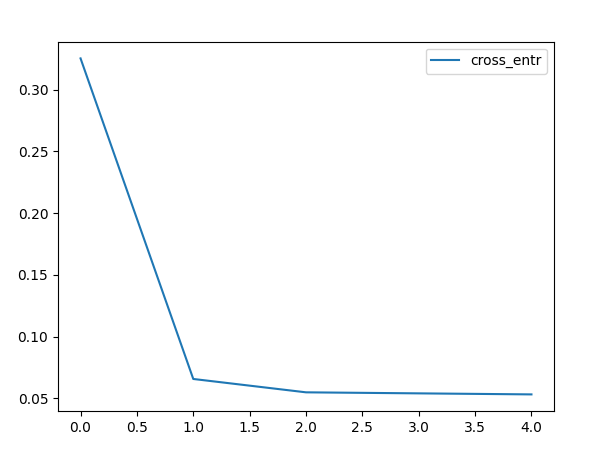
1. Градиентный спуск:



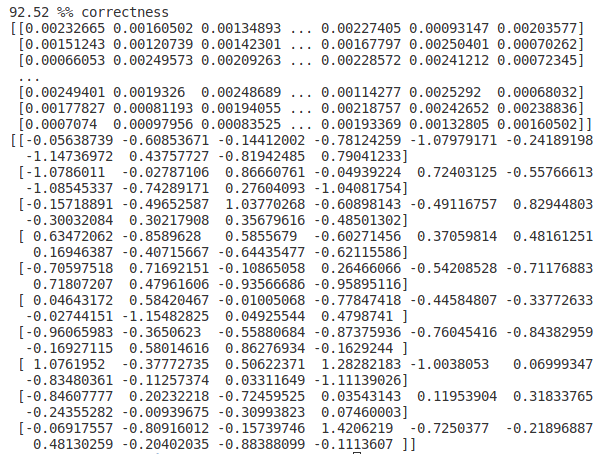
2. Сопряженные градиенты (метод Флетчера-Ривза):



3. Квазиньютоновский метод (BFGS):



Вывод найденных весов (на 2х слоях):



1. **Выводы**

В ходе выполнения лабораторной работы был изучен принцип работы многослойного персептрона, была написана его реализация на языке программирования Python.

В ходе эксперимента по исследованию работы программы на основе различных методов оптимизации (градиентный, Флетчера-Ривза, BFGS) и целевых функций (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера), были сделаны следующие выводы:

1. Самым точным методом оказался метод Флетчера-Ривза.
2. Быстрее всего персептрон обучался при методе градиентного спуска.
3. Самым медленным методом оказался метод BFGS, однако ему необходимо меньшее количество эпох для обучения.