|  | **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 и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**Домашнее задание № 2**

**Разработка многослойного персептрона**

**на основе обратного распространения ошибки FFNN**

**ПО КУРСУ:**

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

Студент *Караник А. А.*

Преподаватель *Каганов Ю. Т.*

*Москва, 2024 г.*

**ОГЛАВЛЕНИЕ**

[1. Цель 3](#_Toc180489674)

[2. Постановка задачи 3](#_Toc180489675)

[3. Практическая реализация 4](#_Toc180489676)

[3. Вывод 9](#_Toc180489677)

# 1. Цель

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

# 2. Постановка задачи

1. Реализовать на языке высокого уровня многослойный персептрон и проверить его работоспособность на примере данных, выбранных из MNISTdataset.

2. Исследовать работу персептрона на основе использования различных целевых функций. (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера).

3. Исследовать работу многослойного персептрона с использованием различных методов оптимизации (градиентный, Флетчера-Ривза (FR), Бройдена-Флетчера-Гольдфарба-Шенно (BFGS)).

4. Провести исследование эффективности работы многослойного персептрона при изменении гиперпараметров (количества нейронов и количества слоев).

5. Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

# 3. Практическая реализация

Текст программы на языке программирования Python:

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

import numpy as np

NUMBER\_OF\_SAMPLES  = 60000

NUBMER\_OF\_TESTS = 10000

(trainX, trainY), (testX, testY) = mnist.load\_data()

trainX, trainY = trainX[:NUMBER\_OF\_SAMPLES], trainY[:NUMBER\_OF\_SAMPLES]

testX, testY = testX[:NUBMER\_OF\_TESTS], testY[:NUBMER\_OF\_TESTS]

trainX = trainX.reshape(trainX.shape[0], 28\*\*2) / 255.0

testX = testX.reshape(testX.shape[0], 28\*\*2) / 255.0

trainX.shape, trainY.shape, testX.shape, testY.shape

def ReLU(x):

    return np.maximum(0, x)

def dReLU(x):

    return np.where(x > 0, 1, 0)

def softmax(x):

    shiftx = x - np.max(x)

    exps = np.exp(shiftx)

    return exps / np.sum(exps)

def dsoftmax(x):

    s = softmax(x)

    return np.diagflat(s) - np.outer(s, s)

def MSE(y, y\_ref):

    return 0.5 \* np.mean((y - y\_ref)\*\*2)

def dMSE(y, y\_ref):

    delta = y - y\_ref

    return delta / delta.size

def KL\_divergence(y, y\_ref):

    return np.mean(y\_ref \* np.log((y\_ref + 1e-9) / (y + 1e-9)))

def KL\_divergence\_derivative(y, y\_ref):

    return -y\_ref / (y + 1e-9)

def categorical\_cross\_entropy(y, y\_ref):

    return -np.mean(y\_ref \* np.log(y + 1e-9))

def categorical\_cross\_entropy\_derivative(y, y\_ref):

   return y - y\_ref

class Perceptron:

    def \_\_init\_\_(self, input\_neurons, hidden\_neurons, output\_neurons,  hidden\_layers\_count, epochs=16, learning\_rate=0.003):

        self.initial\_weights = []

        self.initial\_biases = []

        self.activation\_functions = []

        self.activation\_derivatives = []

        self.epochs = epochs

        self.learning\_rate = learning\_rate

        layers = [input\_neurons] + [hidden\_neurons] \* hidden\_layers\_count + [output\_neurons]

        for i in range(1, len(layers)):

            W = np.random.randn(layers[i - 1], layers[i]) \* 0.3

            b = np.zeros(layers[i])

            self.initial\_weights.append(W)

            self.initial\_biases.append(b)

            self.activation\_functions.append(ReLU)

            self.activation\_derivatives.append(dReLU)

        self.activation\_functions[-1] = softmax

        self.activation\_derivatives[-1] = dsoftmax

    def predict(self, x):

        activation = x

        for W, b, f in zip(self.weights, self.biases, self.activation\_functions):

            activation = f(np.dot(activation, W) + b)

        return int(np.argmax(activation))

    def set\_learning\_rate(self, learning\_rate):

        self.learning\_rate = learning\_rate

    def set\_epochs(self, epochs):

        self.epochs = epochs

    def forward\_and\_backward(self, x, y\_ref, loss\_function, dloss\_function):

        grads\_w = [None] \* len(self.weights)

        grads\_b = [None] \* len(self.biases)

        activations = [x]

        inputs = []

        for (W, b, f) in zip(self.weights, self.biases, self.activation\_functions):

            weighted\_sum = np.dot(activations[-1], W) + b

            inputs.append(weighted\_sum)

            activations.append(f(weighted\_sum))

        y = activations[-1]

        loss = loss\_function(y, y\_ref)

        delta\_loss = dloss\_function(y, y\_ref)

        dZ = np.dot(delta\_loss, dsoftmax(inputs[-1]))

        grads\_w[-1] = np.outer(activations[-2], dZ)

        grads\_b[-1] = dZ

        for i in range(len(self.weights) - 2, -1, -1):

            dA = np.dot(dZ, self.weights[i + 1].T)

            dZ = dA \* self.activation\_derivatives[i](inputs[i])

            grads\_w[i] = np.outer(activations[i], dZ)

            grads\_b[i] = dZ

        return grads\_w, grads\_b, loss

    def validate(self, X, Y, loss\_function):

        correct\_predictions = 0

        total\_loss = 0.0

        for (x, y\_ref) in zip(X, Y):

            activation = x

            for W, b, f in zip(self.weights, self.biases, self.activation\_functions):

                activation = f(np.dot(activation, W) + b)

            total\_loss +=  loss\_function(activation, y\_ref)

            if int(np.argmax(activation)) == np.argmax(y\_ref):

                correct\_predictions += 1

        return total\_loss / len(X), correct\_predictions / len(X)

    def train(self, trainX, trainY, validate\_X, validate\_Y, loss\_function, dloss\_function):

        train\_one\_hot = np.array([np.array([int(i == y) for i in range(10)]) for y in trainY])

        validate\_one\_hot = np.array([np.array([int(i == y) for i in range(10)]) for y in validate\_Y])

        self.weights = [np.copy(w) for w in self.initial\_weights]

        self.biases = [np.copy(b) for b in self.initial\_biases]

        loss, accuracy = self.validate(validate\_X, validate\_one\_hot, loss\_function)

        losses = [loss]

        accuracies = [accuracy]

        for epoch in range(self.epochs):

            for (x, y\_ref) in zip(trainX, train\_one\_hot):

                weight\_grads, bias\_grads, loss = self.forward\_and\_backward(

                    x,

                    y\_ref,

                    loss\_function,

                    dloss\_function

                )

                for i in range(len(self.weights)):

                    self.weights[i] -= weight\_grads[i] \* self.learning\_rate

                    self.biases[i] -= bias\_grads[i] \* self.learning\_rate

            loss, accuracy = self.validate(validate\_X, validate\_one\_hot, loss\_function)

            losses.append(loss)

            accuracies.append(accuracy)

        return losses, accuracies

def launch(number\_of\_neurons, number\_of\_layers, epochs, learging\_rate):

    epochs\_axis = np.arange(epochs + 1)

    perceptron = Perceptron(28\*\*2, number\_of\_neurons, 10, number\_of\_layers, epochs, learging\_rate)

    mse\_loss, mse\_accuracy = perceptron.train(trainX, trainY, testX, testY, MSE, dMSE)

    cross\_entropy\_loss, cross\_entropy\_accuracy = perceptron.train(trainX, trainY, testX, testY,categorical\_cross\_entropy, categorical\_cross\_entropy\_derivative)

    KL\_divergence\_loss, KL\_divergence\_accuracy = perceptron.train(trainX, trainY, testX, testY, KL\_divergence, KL\_divergence\_derivative)

    plt.figure()

    plt.title(f"Зависимость функции потерь от количества эпох (скрытых слоев: {number\_of\_layers}, нейронов: {number\_of\_neurons})")

    plt.xlabel("Количество эпох")

    plt.ylabel("Значение функции потерь")

    plt.plot(epochs\_axis, mse\_loss, label = "MSE")

    plt.plot(epochs\_axis, cross\_entropy\_loss, label = "Categorical cross entropy")

    plt.plot(epochs\_axis, KL\_divergence\_loss, label = "KL divergence")

    plt.legend()

    plt.show()

    plt.figure()

    plt.title(f"Зависимость точности от количества эпох (скрытых слоев: {number\_of\_layers}, нейронов: {number\_of\_neurons})")

    plt.xlabel("Количество эпох")

    plt.ylabel("Точность")

    plt.plot(epochs\_axis, mse\_accuracy, label = "MSE")

    plt.plot(epochs\_axis, cross\_entropy\_accuracy, label = "Categorical cross entropy")

    plt.plot(epochs\_axis, KL\_divergence\_accuracy, label = "KL divergence")

    plt.legend()

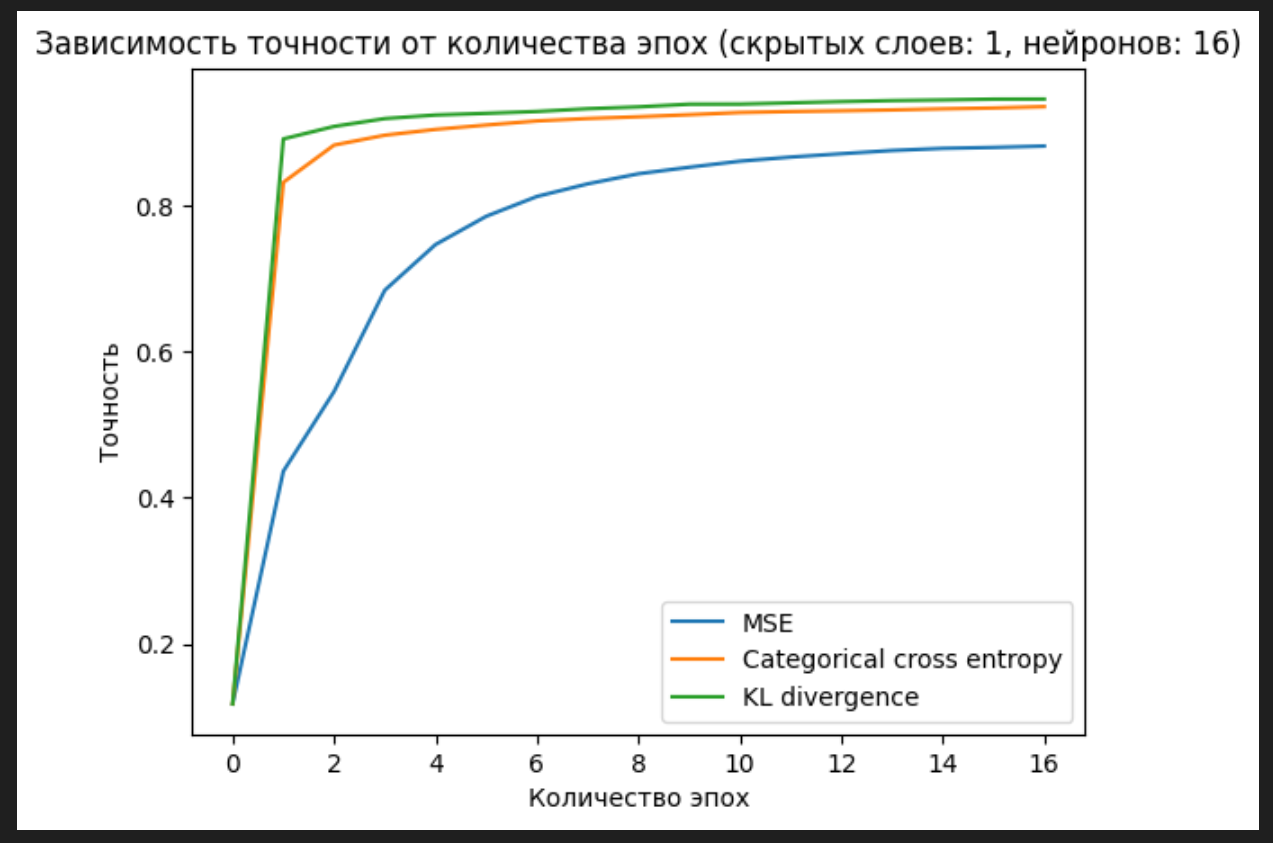
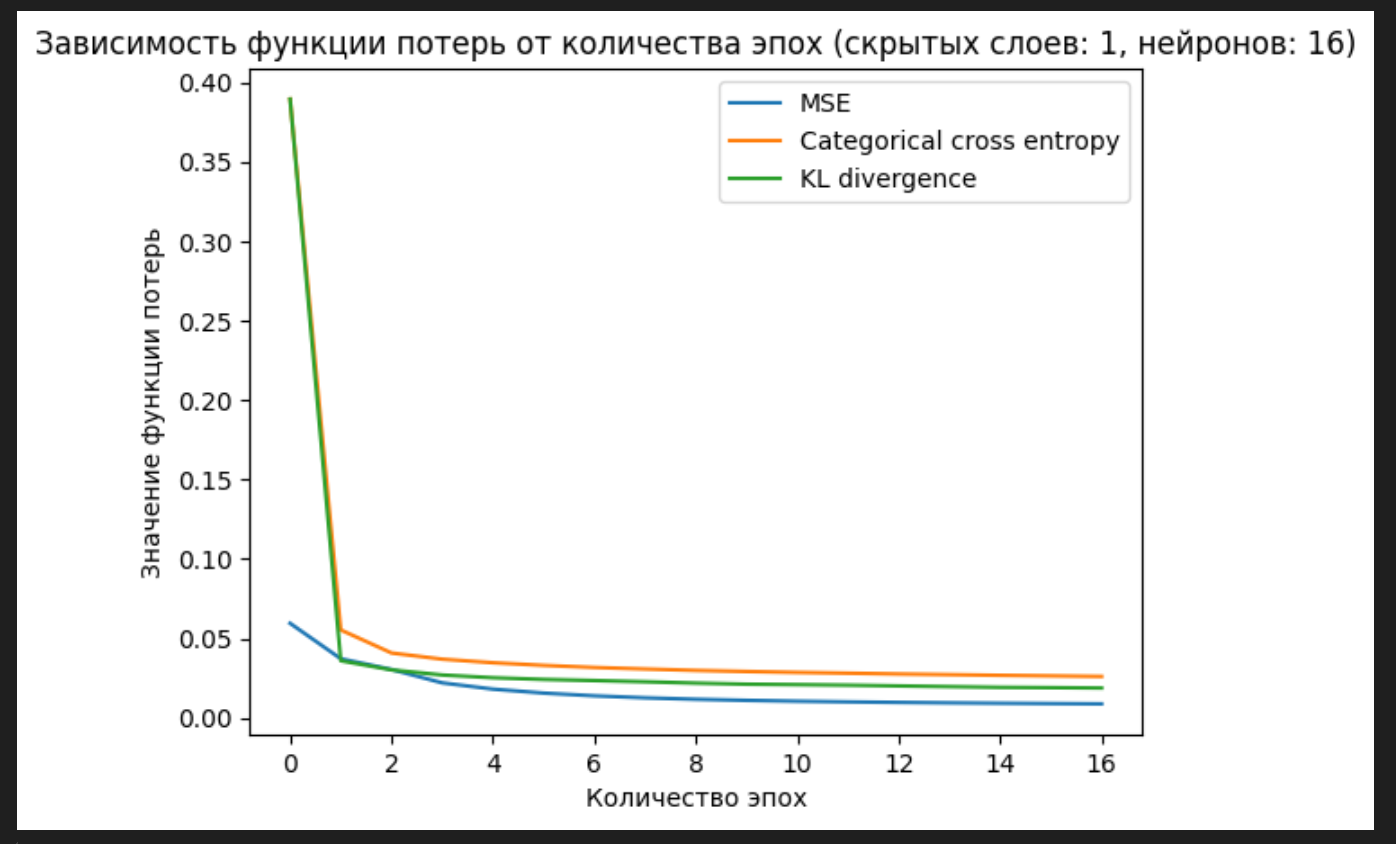
    plt.show()

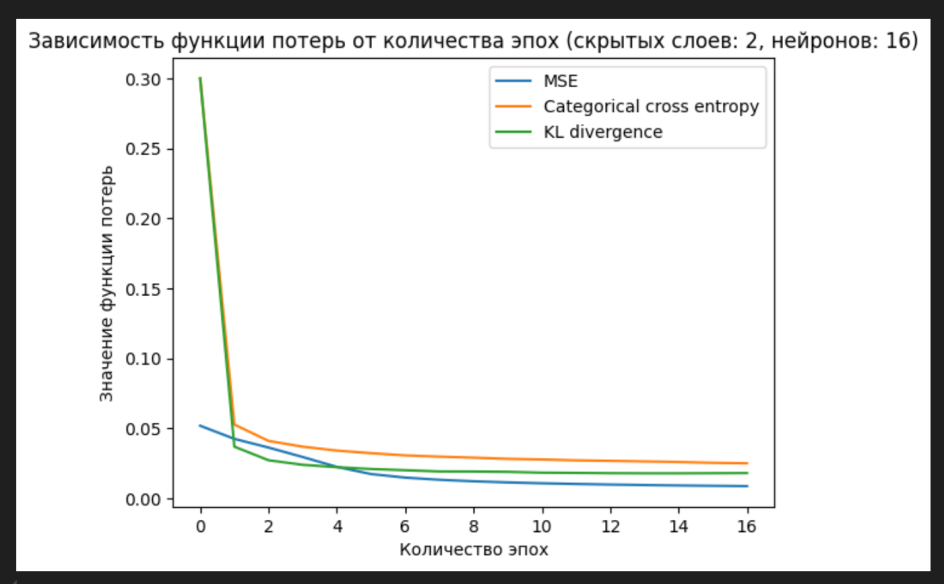
    return

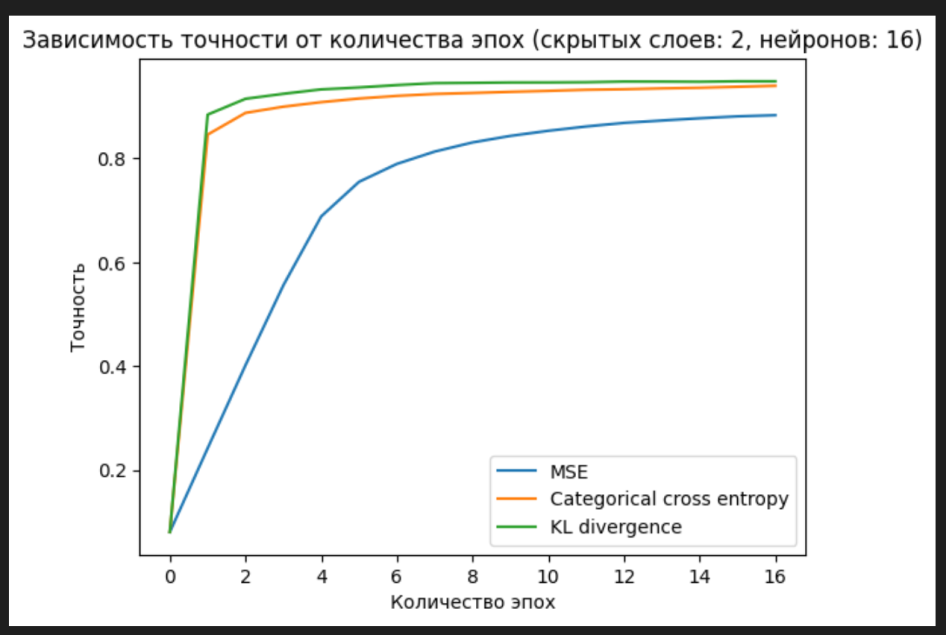
for number\_of\_layers in [1, 2, 3]:

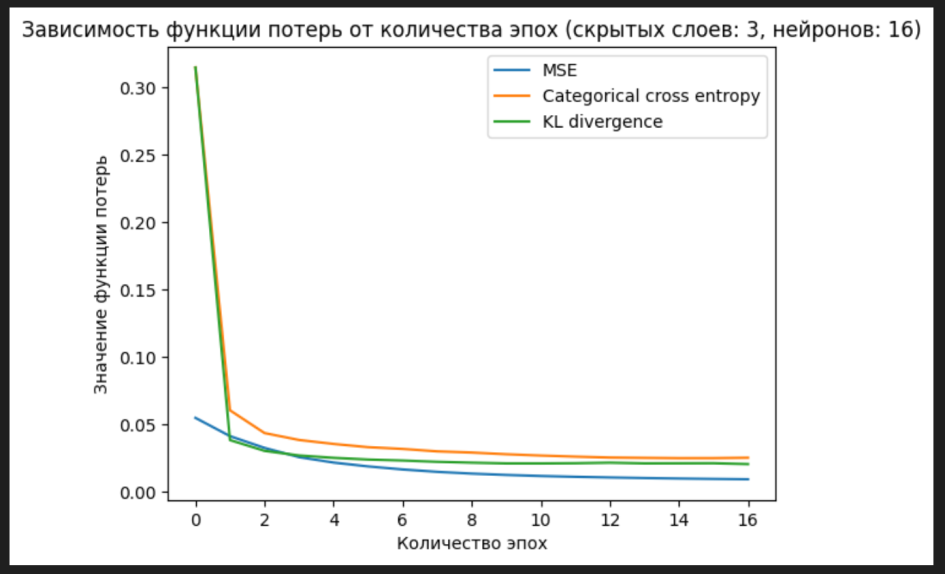
    launch(16, number\_of\_layers, 16, 0.003)

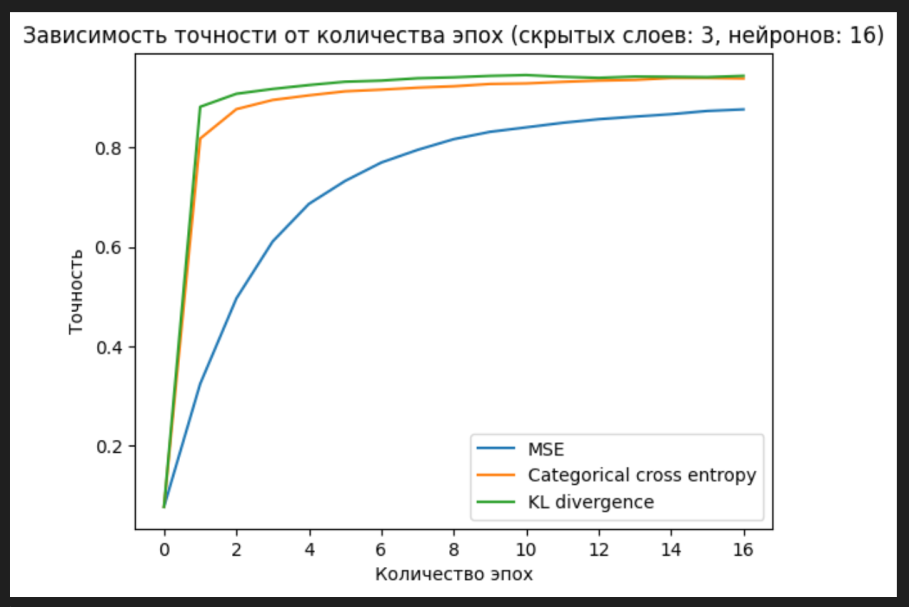
Результаты экспериментов (в виде графиков функций зависимости функции потерь и точности от числа эпох):











# 3. Вывод

В данной работе был разработан многослойный персептрон для распознавания рукописных цифр в виде 28x28 матриц. Разные целевые функции, такие как среднеквадратичная ошибка, перекрестная энтропия и дивергенция Кульбака-Лейблера, влияют на точность и стабильность работы модели. Дивергенция Кульбака-Лейблера и перекрестная энтропия обеспечили лучшую точность по сравнению со среднеквадратичной ошибкой, при этом дивергенция Кульбака-Лейблера привела к более быстрому и стабильному росту точности. Эффективность модели также зависит от числа нейронов и слоев: большие значения уменьшают количество эпох для обучения, но увеличивают время вычислений.