МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Южно-Уральский государственный университет"
(национальный исследовательский университет)

Образовательная программа
02.04.02 "Технологии разработки высоконагруженных систем"
по направлению подготовки
"Фундаментальная информатика и информационные технологии"
(степень "магистр")

ЗАДАНИЯ
ДЛЯ ПРАКТИЧЕСКИХ РАБОТ
и
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ИХ ВЫПОЛНЕНИЮ
по дисциплине
"Машинное обучение"

Разработчики:
Л.Б. Соколинский, доктор физ.-мат. наук, профессор
Р.С. Федянина, старший преподаватель

Челябинск-2019

ОГЛАВЛЕНИЕ

Задание 1. Нейронная сеть для распознавания рукописных цифр

Написать компьютерную программу на языке Python 3 создающую и обучающую нейронную сеть для распознавания рукописных цифр с использованием метода градиентного спуска и базы данных MNIST.

Исходные данные:

- Количество слоев нейронной сети: 3.
- Входные данные для нейронной сети: изображения размером 28×28 пикселов.
- Среда программирования: Python 3.6 и выше.
- Используемая библиотека: NumPy.

Методические указания к заданию 1

1. Организация среды разработки

Систему программирования на языке Python 3.6.4 для Windows можно загрузить с официального сайта https://www.python.org/downloads/windows/ (рекомендуется использовать Windows x86 executable installer). ***Перед установкой необходимо выбрать пункт "Add Python to PATH"***.

В качестве самоучителя по языку Python можно использовать ресурс https://pythonworld.ru/samouchitel-python.

2. Установка библиотеки NumPy

1. Запустите приложение «Командная строка» (для этого наберите `cmd` в окне поиска панели задач Windows).

```
pip3 install numpy
```

2. Запустите команду для установки пакета:

3. Установка рабочей папки проекта

Создайте каталог `NeuralNetwork`, в котором Вы будете хранить исходные тексты программ, создаваемых в ходе выполнения практических заданий (например, `C:\NeuralNetwork`). В каталоге `NeuralNetwork` создайте подкаталог `Network1`, в котором будут храниться исходные коды задания 1.

4. Создание нейронной сети

Запустите среду разработки (для запуска среды разработки IDLE, наберите idle в окне поиска панели задач Windows). Создайте новый файл для программы (меню `File/New`

File). Сохраните этот файл в каталоге Network1 под именем network (меню File/Save). Расширение .py будет подставлено по умолчанию.

Скопируйте в окно программы network.py следующие команды и впишите свои данные:

```
"""""""""""""""""""""""""""""""""""""""
network.py
Модуль создания и обучения нейронной сети для распознавания рукописных цифр
с использованием метода градиентного спуска.
Группа:<Указать номер группы>
ФИО:<Указать ФИО студента>
"""""""""""""""""""""""""""""""""""""""

#### Библиотеки
# Стандартные библиотеки

import random # библиотека функций для генерации случайных значений


# Сторонние библиотеки

import numpy as np # библиотека функций для работы с матрицами

""" ---Раздел описаний--- """
""" --Описание класса Network--"""
class Network(object): # используется для описания нейронной сети
    def __init__(self, sizes): # конструктор класса
                               # self – указатель на объект класса
                               # sizes – список размеров слоев нейронной
сети
        self.num_layers = len(sizes) # задаем количество слоев нейронной
сети
        self.sizes = sizes # задаем список размеров слоев нейронной сети
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]] # задаем
случайные начальные смещения
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1],
sizes[1:])] # задаем случайные начальные веса связей
""" --Конец описания класса Network--"""
""" --- Конец раздела описаний--- """

""" ---Тело программы--- """
net = Network([2, 3, 1]) # создаем нейронную сеть из трех слоев
""" ---Конец тела программы--- """

""" Вывод результата на экран: """
print('Сеть net:')
print('Количетво слоев:', net.num_layers)
for i in range(net.num_layers):
    print('Количество нейронов в слое', i,':',net.sizes[i])
for i in range(net.num_layers-1):
    print('W_',i+1,':')
    print(np.round(net.weights[i],2))
    print('b_',i+1,':')
    print(np.round(net.biases[i],2))
```

Сохраните файл network.py и выполните программу network. Для того чтобы запустить исполнение программы, выберите Run/Run Module (или нажмите F5). В результате будет создан объект класса Network, задающий трехуровневую нейронную сеть с соответствующими параметрами. При создании объекта класса Network веса и

смещения инициализируются случайным образом. Для инициализации этих величин используется функция `np.random.randn` из библиотеки `NumPy`. Данная функция, генерирует числа с нормальным распределением для массива заданной размерности.

Определение сигмоидальной функции

В качестве функции активации для нейронов сети используется сигмоидальная функция, вычисляющая выходной сигнал искусственного нейрона. Ниже представлен код, для определения функции. Добавьте этот код в раздел описаний программы `network.py`.

```
def sigmoid(z): # определение сигмоидальной функции активации
    return 1.0/(1.0+np.exp(-z))
```

Обратите внимание, что для описания сигмоидальной функции активации используется функция для вычисления экспоненты из библиотеки `NumPy`, это позволяет передавать массив в качестве входного параметра сигмоидальной функции. В этом случае функция экспоненты применяется поэлементно, то есть в векторизованной форме.

Метод feedforward

Добавьте метод `feedforward` в описание класса `Network`.

```
    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

Данный метод осуществляет подсчет выходных сигналов нейронной сети при заданных входных сигналах. Параметр $\boldsymbol{a}$ является массивом $n \times 1$, где n – количество нейронов входного слоя. Функция `np.dot` вычисляет произведение матриц. Для подсчета выходных значений нейронной сети, необходимо один раз вызвать метод `feedforward`, в результате чего выходные сигналы будут последовательно вычислены для всех слоев нейронной сети.

5. Обучение нейронной сети

Для реализации механизма обучения создаваемой нейронной сети добавим метод SGD, который реализует стохастический градиентный спуск. Метод имеет следующие параметры:

«`Training_data`» – обучающая выборка, состоящая из пар вида $(\vec{x}, \vec{y})$, где $\vec{x}$ – вектор входных сигналов, а $\vec{y}$ – ожидаемый вектор выходных сигналов;

«`epochs`» – количество эпох обучения;

«`mini_batch_size`» - размер подвыборки;

«eta» - скорость обучения;

«test_data» - (необязательный параметр); если данный аргумент не пуст, то программа после каждой эпохи обучения осуществляет оценку работы сети и показывает достигнутый прогресс.

Добавьте программный код метода SGD в раздел в описания класса Network:

```
    def SGD( # Стохастический градиентный спуск
          self             # указатель на объект класса
        , training_data    # обучающая выборка
        , epochs           # количество эпох обучения
        , mini_batch_size  # размер подвыборки
        , eta              # скорость обучения
        , test_data        # тестирующая выборка
        ):
        test_data = list(test_data) # создаем список объектов тестирующей
выборки
        n_test = len(test_data) # вычисляем длину тестирующей выборки
        training_data = list(training_data) # создаем список объектов
обучающей выборки
        n = len(training_data) # вычисляем размер обучающей выборки
        for j in range(epochs): # цикл по эпохам
            random.shuffle(training_data) # перемешиваем элементы обучающей
выборки
            mini_batches = [training_data[k:k+mini_batch_size] for k in
range(0, n, mini_batch_size)] # создаем подвыборки
            for mini_batch in mini_batches: # цикл по подвыборкам
                self.update_mini_batch(mini_batch, eta) # один шаг
градиентного спуска
            print ("Epoch {0}: {1} / {2}".format(j,
self.evaluate(test_data), n_test)) # смотрим прогресс в обучении
```

Данный программный код работает следующим образом. В начале каждой эпохи обучения элементы обучающей выборки перемешиваются (переставляются в случайном порядке) с помощью функции shuffle() из библиотеки random, после чего обучающая выборка последовательно разбивается на подвыборки длины mini_batch_size. Для каждой подвыборки выполняется один шаг градиентного спуска с помощью метода update_mini_batch (см. ниже). После того, как выполнен последний шаг градиентного спуска, т.е. выполнен метод update_mini_batch для последней подвыборки, на экран выводиться достигнутый прогресс в обучении нейронной сети, вычисляемый на тестовой выборке с помощью метода evaluate (см. ниже).

Анализируя программный код метода update_mini_batch можно увидеть, что основная часть вычислений осуществляется при вызове метода backprop (см. ниже). Данный метод класса Network реализует алгоритм обратного распространения ошибки, который является быстрым способом вычисления градиента стоимостной функции. Таким образом, метод update_mini_batch вычисляет градиенты для каждого прецедента $(\vec{x}, \vec{y})$ в подвыборке, а затем соответствующим образом обновляет веса и смещения

нейронной сети. Добавьте код метода `update_mini_batch` в раздел в описания класса `Network`:

```
    def update_mini_batch( # Шаг градиентного спуска
         self              # указатель на объект класса
         , mini_batch      # подвыборка
         , eta             # скорость обучения
         ):
        nabla_b = [np.zeros(b.shape) for b in self.biases] # список
градиентов dC/db для каждого слоя (первоначально заполняются нулями)
        nabla_w = [np.zeros(w.shape) for w in self.weights] # список
градиентов dC/dw для каждого слоя (первоначально заполняются нулями)
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y) # послойно
вычисляем градиенты dC/db и dC/dw для текущего прецедента (x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)] #
суммируем градиенты dC/db для различных прецедентов текущей подвыборки
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)] #
суммируем градиенты dC/dw для различных прецедентов текущей подвыборки
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)] #
обновляем все веса w нейронной сети
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)] # обновляем
все смещения b нейронной сети
```

Скопируйте в раздел описания класса `Network` программный код метода `backprop`, реализующего алгоритм обратного распространения:

```
    def backprop( # Алгоритм обратного распространения
         self   # указатель на объект класса
         , x    # вектор входных сигналов
         , y    # ожидаемый вектор выходных сигналов
         ):
        nabla_b = [np.zeros(b.shape) for b in self.biases] # список
градиентов dC/db для каждого слоя (первоначально заполняются нулями)
        nabla_w = [np.zeros(w.shape) for w in self.weights] # список
градиентов dC/dw для каждого слоя (первоначально заполняются нулями)

        # определение переменных
        activation = x # выходные сигналы слоя (первоначально соответствует
выходным сигналам 1-го слоя или входным сигналам сети)
        activations = [x] # список выходных сигналов по всем слоям
(первоначально содержит только выходные сигналы 1-го слоя)
        zs = [] # список активационных потенциалов по всем слоям
(первоначально пуст)

        # прямое распространение
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b # считаем активационные потенциалы
текущего слоя
            zs.append(z) # добавляем элемент (активационные потенциалы
слоя) в конец списка
            activation = sigmoid(z) # считаем выходные сигналы текущего
слоя, применяя сигмоидальную функцию активации к активационным потенциалам
слоя
            activations.append(activation) # добавляем элемент (выходные
сигналы слоя) в конец списка
```

```
        # обратное распространение
        delta = self.cost_derivative(activations[-1], y) *
sigmoid_prime(zs[-1]) # считаем меру влияния нейронов выходного слоя L на
величину ошибки (BP1)
        nabla_b[-1] = delta # градиент dC/db для слоя L (BP3)
        nabla_w[-1] = np.dot(delta, activations[-2].transpose()) # градиент
dC/dw для слоя L (BP4)

        for l in range(2, self.num_layers):
            z = zs[-l] # активационные потенциалы l-го слоя (двигаемся по
списку справа налево)
            sp = sigmoid_prime(z) # считаем сигмоидальную функцию от
активационных потенциалов l-го слоя
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp #
считаем меру влияния нейронов l-го слоя  на величину ошибки (BP2)
            nabla_b[-l] = delta # градиент dC/db для l-го слоя (BP3)
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())#
градиент dC/dw для l-го слоя (BP4)
        return (nabla_b, nabla_w)
```

Скопируйте в раздел описания класса `Network` программный код метода `evaluate`, демонстрирующего прогресс в обучении:

```
    def evaluate(self, test_data): # Оценка прогресса в обучении
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

Указанный метод возвращает количество прецедентов тестирующей выборки, для которых нейронная сеть выдает правильный результат. Тестирующая выборка состоит из пар (x, y), где x – вектор размерности 784, содержащий изображение цифры, а y – целое числовое значение цифры, изображенной на картинке. Ответ нейронной сети определяется как номер нейрона в выходном слое, имеющего наибольшее значение функции активации. Метод `evaluate` вызывается в методе `SGD` после завершения очередной эпохи обучения.

Скопируйте в раздел описания класса Network программный код метода `cost_derivative`, вычисляющего вектор частных производных $\nabla C(\vec{a}^L) = \vec{a}^L - \vec{y}$:

```
    def cost_derivative(self, output_activations, y): # Вычисление частных
производных стоимостной функции по выходным сигналам последнего слоя
        return (output_activations-y)
```

Указанный метод вызывается в методе `backprop`.

Скопируйте в конец раздела описаний (после функции `sigmoid`) код функции `sigmoid_prime`, вычисляющей производную сигмоидальной функции:

```
def sigmoid_prime(z):# Производная сигмоидальной функции
    return sigmoid(z)*(1-sigmoid(z))
```

☐ Сохраните и закройте файл `network.py`.

6. Работа с базой данных MNIST

Для обучения нейронной сети будем использовать архив http://deeplearning.net/data/mnist/mnist.pkl.gz с сайта Лаборатории машинного обучения Университета Монреаля, сформированный на основе базы данных MNIST, который содержит 70 000 изображений рукописных цифр, разделенных на три набора:

1) `training_data` – набор из 50 000 изображений предназначен для обучения нейронных сетей;
2) `validation_data` – набор из 10 000 изображений предназначен для текущей оценки работы алгоритма обучения и подбора параметров обучения (используется в последующих лабораторных работах);
3) `test_data` – набор из 10 000 изображений предназначен для проверки работы нейронной сетей.

Каждый набор состоит из двух списков: списка изображений (в градациях серого) и соответствующего списка цифр в диапазоне от 0 до 9. Изображение представлено в виде одномерного numpy-массива размера $784 = 28 \times 28$ значений от 0 до 1, где 0 соответствует черному цвету пиксела, а 1 – белому.

☐ Загрузите архив http://deeplearning.net/data/mnist/mnist.pkl.gz и сохраните его в директории Network1.

Функции для работы с базой данных MNIST целесообразнее вынести в отдельный файл. Создайте новый файл `mnist_loader` и сохраните его в директории `Network1`. Скопируйте в окно программы `mnist_loader.py` следующие команды и впишите свои данные:

```
"""
mnist_loader.py
~~~~~~~~~~
Модуль для подключения и использования базы данных MNIST.

Группа:[Указать номер группы]
ФИО:[Указать ФИО студента]
"""
import gzip # библиотека для сжатия и распаковки файлов gzip и gunzip.
import pickle # библиотека для сохранения и загрузки сложных объектов
Python.
import numpy as np # библиотека для работы с матрицами

def load_data():

    f = gzip.open('mnist.pkl.gz', 'rb') # открываем сжатый файл gzip в
двоичном режиме
    training_data, validation_data, test_data = pickle.load(f,
encoding='latin1') # загружам таблицы из файла
    f.close() # закрываем файл
    return (training_data, validation_data, test_data)
```

Для использования базы данных MNIST в нашей программе необходимо скорректировать форматы наборы training_data, validation_data и test_data. Это делается в функции `load_data_wrapper`. Скопируйте в файл `mnist_loader` следующий программный код.

```
def load_data_wrapper():

    tr_d, va_d, te_d = load_data() # инициализация наборов данных в формате
MNIST
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]] #
преобразование массивов размера 1 на 784 к массивам размера 784 на 1
    training_results = [vectorized_result(y) for y in tr_d[1]] #
представление цифр от 0 до 9 в виде массивов размера 10 на 1
    training_data = zip(training_inputs, training_results) # формируем
набор обучающих данных из пар (x, y)
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]] #
преобразование массивов размера 1 на 784 к массивам размера 784 на 1
    validation_data = zip(validation_inputs, va_d[1]) # формируем набор
данных проверки из пар (x, y)
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]] #
преобразование массивов размера 1 на 784 к массивам размера 784 на 1
    test_data = zip(test_inputs, te_d[1]) # формируем набор тестовых данных
из пар (x, y)
    return (training_data, validation_data, test_data)
```

Данная функция преобразует `training_data` в список, содержащий 50 000 пар (x, y), где x является 784-мерным numpy-массивом, содержащим входное изображение, а y – это 10-мерный numpy-массив, представляющий собой вектор, у которого координата с порядковым номером, соответствующим цифре на изображении, равняется единице, а остальные координаты нулевые. Аналогичные преобразования делаются для наборов `validation_data` и `test_data`.

Для преобразования числа в вектор-столбец (10-мерный numpy массив), используется следующая функция `vectorized_result`. Скопируйте ее программный код в файл `mnist_loader`.

```
def vectorized_result(j):

    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

☐ Сохраните и закройте файл mnist_loader.py.

7. Запуск программы

В среде разработки IDLE последовательно выполните следующие команды для установки рабочего каталога на примере `C:\NeuralNetwork`:

```
>>>import os
>>>os.chdir ('C:\\NeuralNetwork\\Network1')
```

Следующие команды используются для подключения модуля mnist_loader и инициализации наборов данных для обучения нейронной сети:

```
>>>import mnist_loader
>>>training_data, validation_data, test_data =
mnist_loader.load_data_wrapper()
```

Подключите созданный Вами модуль `network.py`:

```
>>>import network
```

При этом выполниться написанная в нем программа, выводящая информацию о нейронной сети.

Создайте нейронную сеть для распознавания рукописных цифр:

```
>>>net = network.Network([784, 30, 10])
```

Параметры, указанные при вызове данного метода, определяют топологию создаваемой сети. Таким образом, в результате выполнения команды будет создана сеть, состоящая из трех слоев: входной слой сети состоит из 784-х нейронов; внутренний слой из 30 нейронов и выходной слой из 10 нейронов.

Запустите процедуру обучения созданной нейронной сети, включающую 30 эпох:

```
>>>net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Параметры, указанные при вызове метода `SGD`: обучающая выборка, количество эпох обучения, размер подвыборки, скорость обучения, тестирующая выборка.

Обучение может занять несколько минут. В ходе обучения будет выдаваться информация о пройденных эпохах (см. рис. 2). Для каждой эпохи выводится отношение количества правильно распознанных цифр к общему количеству цифр в тестовой выборке. Например, запись `Epoch 6: 9374 / 10000` говорит о том, что в результате эпохи обучения с номером 6 достигнута точность распознавания $\frac{9374}{10000} \approx 0.94$, что составляет 94%.

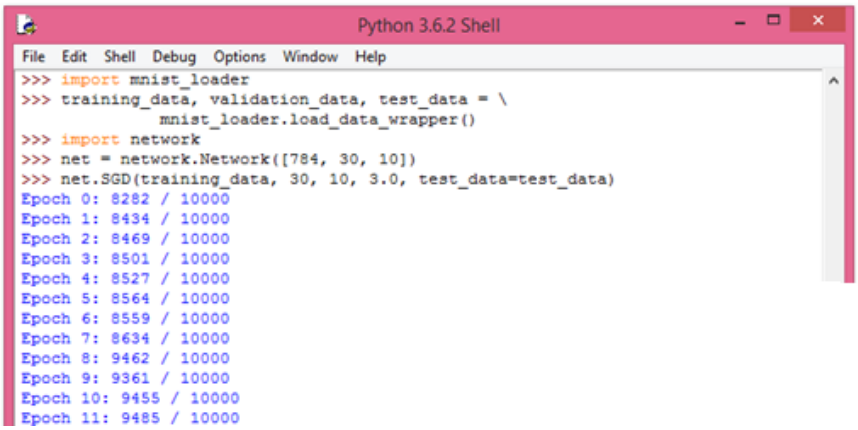

Рис.2. Результат работы программы Network1.

Задание 2. Моделирование конфигурации нейронной сети и параметров скорости её обучения

Используя написанную ранее программу для распознавания рукописных цифр, создайте и обучите несколько нейронных сетей. Создаваемые сети должны иметь разную топологию. Для каждой сети попытайтесь подобрать оптимальные параметры для запуска процедуры обучения методом градиентного спуска.

Примеры запусков:

```
>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)

>>> net = network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 1000, 0.001, test_data=test_data)

>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)

>>> net = network.Network([784, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

Ответьте на вопросы:

1) Какие параметры влияют на обучение нейронной сети? Объясните характер их влияния?

2) Какова максимальная точность распознавания, которую вам удалось достичь при обучении нейронной сети (с указанием топологии нейронной сети)?
3) Как повлияло изменение топологии нейронной сети на качество ее обучения?
4) Как повлияло изменение параметров запуска метода градиентного спуска на качество обучения нейронной сети?

Задание 3. Стоимостная функция на основе перекрестной энтропии

Написать компьютерную программу на языке Python 3 создающую и обучающую нейронную сеть для распознавания рукописных цифр с использованием метода градиентного спуска и стоимостной функции на основе перекрестной энтропии.

Методические указания к заданию 3

1. Создание нейронной сети

В каталоге `NeuralNetwork` создайте подкаталог `Network2`. Запустите среду разработки IDLE. Создайте новый файл для программы и сохраните этот файл в каталоге `Network2` под именем `network2`.

Скопируйте в окно программы `network2.py` следующие команды и впишите свои данные.

```
"""""""""""""""""""""""""""""""""""""""""""
network2.py
Модуль создания и обучения нейронной сети для распознавания рукописных цифр
на основе метода стохастического градиентного спуска для прямой нейронной
сети и стоимостной функции на основе перекрестной энтропии, регуляризации и
улучшеннного способа инициализации весов нейронной сети.

Группа:<Указать номер группы>
ФИО:<Указать ФИО студента>
"""""""""""""""""""""""""""""""""""""""""""

#### Библиотеки
# Стандартные библиотеки
import json # библиотека для кодирования/декодирования данных/объектов
Python
import random # библиотека функций для генерации случайных значений
import sys # библиотека для работы с переменными и функциями, имеющими
отношение к интерпретатору и его окружению

# Сторонние библиотеки
import numpy as np # библиотека функций для работы с матрицами


""" ---Раздел описаний--- """



""" --- Конец раздела описаний--- """
```

Добавим в раздел описаний основной класс программы – класс Network для определения нейронной сети. Скопируйте код в раздел описаний network2.py:

```
""" --Описание класса Network--"""
class Network(object):
    def __init__(  # конструктор класса
        self       # указатель на объект класса
        , sizes    # список размеров слоев нейронной сети
        , cost=CrossEntropyCost  # стоимостная функция (по умолчанию будет
использоваться функция перекрестной энтропии)
        ):
        self.num_layers = len(sizes) # задаем количество слоев нейронной
сети
        self.sizes = sizes # задаем список размеров слоев нейронной сети
        self.default_weight_initializer() # метод инициализации начальных
весов связей и смещений по умолчанию
        self.cost=cost # задаем стоимостную функцию

    def default_weight_initializer(self): # метод инициализации начальных
весов связей и смещений
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]] #
задаем случайные начальные смещения
        self.weights = [np.random.randn(y, x)/np.sqrt(x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])] #
задаем случайные начальные веса связей
```

Обратите внимание, в методе default_weight_initializer начальные смещения задаются аналогично программе network.py, а инициализация весов связей осуществляется иначе. Для задания начальных значений весов связей число, возвращаемое функцией np.random.randn, делится на квадратный корень от числа входных сигналов нейрона.

В дополнение к методу default_weight_initializer добавим в раздел описаний в класс Network метод large_weight_initializer, для этого скопируйте нижеприведенный код в раздел описаний network2.py. В методе large_weight_initializer начальные смещения и веса задаются как в программе Network.

```
    def large_weight_initializer(self):
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]] #
задаем случайные начальные смещения
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])] #
задаем случайные начальные веса
```

Определение стоимостных функций

Для определения среднеквадратичной стоимостной функции создайте класс QuadraticCost, для этого скопируйте в раздел описаний программы network2.py

код представленный ниже. Обратите внимание раздел «Определение стоимостных функции» должен предшествовать описанию класса `Network`.

```
""" -- Определение стоимостных функции --"""

class QuadraticCost(object): # Определение среднеквадратичной стоимостной
функции

    @staticmethod
    def fn(a, y): # Стоимостная функция
        return 0.5*np.linalg.norm(a-y)**2

    @staticmethod
    def delta(z, a, y):  # Мера влияния нейронов выходного слоя на величину
ошибки
        return (a-y) * sigmoid_prime(z)
```

Для определения стоимостной функции на основе перекрестной энтропии создайте класс `CrossEntropyCost`, для этого скопируйте в раздел описаний программы `network2.py` следующий код:

```
class CrossEntropyCost(object):  # Определение стоимостной функции на основе
перекрестной энтропии

    @staticmethod
    def fn(a, y): # Стоимостная функция
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def delta(z, a, y): # Мера влияния нейронов выходного слоя на величину
ошибки
        return (a-y)
```

Обратите внимание! Использование функции `np.nan_to_num` позволяет гарантировать правильную обработку очень маленьких и слишком больших чисел.

Метод feedforward

Добавьте в описание класса `Network` метод `feedforward`, который осуществляет подсчет выходных сигналов нейронной сети при заданных входных сигналах. Реализация данного метода не отличается от реализации в программе `network.py`

Метод стохастического градиентного спуска

Реализация метода SGD для программы `network2.py` отличается от реализации в программе `network.py` незначительно. Основное отличие заключается в добавлении механизма регуляризации и возможности вывода дополнительной информации по завершению каждой эпохи обучения, такой как: значение стоимостной функции, рассчитанное на оценочной и/или обучающей выборке, достигнутый прогресс в обучении, рассчитанный на оценочной и/или обучающей выборке.

Новая версия метода SGD имеет большее количество входных параметров. Первые четыре параметра такие же, как и в программе `network2.py`: «`Training_data`», «`epochs`», «`mini_batch_size`», «`eta`». Подробное описание этих переменных приведено в методических указаниях к заданию 1.

Следующие шесть параметров – новые:

«`lmbda`» - параметр L2-регуляризации (параметр сглаживания), по умолчанию равен 0, что соответствует случаю, когда метод градиентного спуска применяется без регуляризации;

«`evaluation_data`» - (необязательный параметр); оценочная выборка, состоящая из пар вида $(\vec{x}, \vec{y})$, где $\vec{x}$ – вектор входных сигналов, а $\vec{y}$ – ожидаемый вектор выходных сигналов;

«`monitor_evaluation_cost`» - (необязательный параметр/флаг); по умолчанию равен «false». Если данный параметр равен «true», то программа после каждой эпохи обучения осуществляет оценку работы сети и выводит значение стоимостной функции на наборе «`evaluation_data`».

«`monitor_evaluation_accuracy`» - (необязательный параметр/флаг); по умолчанию равен «false». Если данный параметр равен «true», то программа после каждой эпохи обучения осуществляет оценку работы сети и показывает достигнутый прогресс, рассчитанный на наборе «`evaluation_data`».

«`monitor_training_cost`» - (необязательный параметр/флаг); по умолчанию равен «false». Если данный параметр равен «true», то программа после каждой эпохи обучения осуществляет оценку работы сети и выводит значение стоимостной функции на наборе «`training_data`».

«`monitor_training_accuracy`» - (необязательный параметр/флаг); по умолчанию равен «false». Если данный параметр равен «true», то программа после каждой эпохи обучения осуществляет оценку работы сети и показывает достигнутый прогресс, рассчитанный на наборе «`training_data`».

Добавьте программный код метода `SGD` в раздел в описания класса `Network`:

```
    def SGD(self, training_data, epochs, mini_batch_size, eta,
            lmbda = 0.0 # параметр сглаживания L2-регуляризации
            , evaluation_data=None # оценочная выборка
            , monitor_evaluation_cost=False # флаг вывода на экран информа-
ции о значении стоимостной функции в процессе обучения, рассчитанном на
оценочной выборке
            , monitor_evaluation_accuracy=False # флаг вывода на экран ин-
формации о достигнутом прогрессе в обучении, рассчитанном на оценочной
выборке
            , monitor_training_cost=False # флаг вывода на экран информации
о значении стоимостной функции в процессе обучения, рассчитанном на
обучающей выборке
            , monitor_training_accuracy=False # флаг вывода на экран инфор-
мации о достигнутом прогрессе в обучении, рассчитанном на обучающей выборке
            ):
        if evaluation_data:
            evaluation_data = list(evaluation_data)
            n_data = len(evaluation_data)
        training_data = list(training_data)
        n = len(training_data)
        evaluation_cost, evaluation_accuracy = [], []
        training_cost, training_accuracy = [], []
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(
                    mini_batch, eta, lmbda, len(training_data))
            print ("Epoch %s training complete" % j)
            if monitor_training_cost:
                cost = self.total_cost(training_data, lmbda)
                training_cost.append(cost)
                print ("--Cost on training data: {}".format(cost))
            if monitor_training_accuracy:
                accuracy = self.accuracy(training_data, convert=True)
                training_accuracy.append(accuracy)
                print ("--Accuracy on training data: {} / {}".format(
                    accuracy, n))
            if monitor_evaluation_cost:
                cost = self.total_cost(evaluation_data, lmbda,
convert=True)
                evaluation_cost.append(cost)
                print ("--Cost on evaluation data: {}".format(cost))
            if monitor_evaluation_accuracy:
                accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print ("--Accuracy on evaluation data: {} / {}".format(
                    self.accuracy(evaluation_data), n_data))
            print
        return evaluation_cost, evaluation_accuracy, \
            training_cost, training_accuracy
```

Данный программный код работает аналогично программе `network.py`. В начале каждой эпохи обучения элементы обучающей выборки перемешиваются (переставляются в случайном порядке) с помощью функции `shuffle()` из библиотеки `random`, после чего обучающая выборка последовательно разбивается на подвыборки длины `mini_batch_size`. Для каждой подвыборки выполняется один шаг градиентного

спуска с помощью метода update_mini_batch (см. ниже). После того, как выполнен последний шаг градиентного спуска, т.е. выполнен метод update_mini_batch для последней подвыборки, на экран выводиться сообщение о завершении соответствующей эпохи. В случае, если при вызове метода SGD, в качестве параметра, была передана оценочная выборка evaluation_data и переменные monitor_evaluation_cost, monitor_evaluation_accuracy и др. равны «True», то на экран выводится информация о значение функции потерь по всей выборке и достигнутый прогресс в обучении нейронной сети, вычисляемый на соответствующей выборке с помощью метода evaluate (см. ниже).

В код метода update_mini_batch так же необходимо внести изменения, связанные с добавлением в network2.py механизма регуляризации. Добавьте код метода update_mini_batch в раздел в описания класса Network:

```
    def update_mini_batch( # Шаг градиентного спуска
         self              # указатель на объект класса
         , mini_batch      # подвыборка
         , eta             # скорость обучения
         , lmbda           # параметр сглаживания L2-регуляризации
         , n               #
         ):
        nabla_b = [np.zeros(b.shape) for b in self.biases] # список
градиентов dC/db для каждого слоя (первоначально заполняются нулями)
        nabla_w = [np.zeros(w.shape) for w in self.weights] # список
градиентов dC/dw для каждого слоя (первоначально заполняются нулями)
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y) # послойно
вычисляем градиенты dC/db и dC/dw для текущего прецедента (x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)] #
суммируем градиенты dC/db для различных прецедентов текущей подвыборки
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)] #
суммируем градиенты dC/dw для различных прецедентов текущей подвыборки
        self.weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)] #
обновляем все веса w нейронной сети
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)] # обновляем
все смещения b нейронной сети
```

Скопируйте в раздел описания класса Network программный код метода backprop, реализующего алгоритм обратного распространения:

```
    def backprop(# Алгоритм обратного распространения
        self   # Указатель на объект класса
        , x    # Вектор входных сигналов
        , y    # Ожидаемый вектор выходных сигналов
        ):

        nabla_b = [np.zeros(b.shape) for b in self.biases] # список
градиентов dC/db для каждого слоя (первоначально заполняются нулями)

        nabla_w = [np.zeros(w.shape) for w in self.weights] # список
градиентов dC/dw для каждого слоя (первоначально заполняются нулями)


        # Определение переменных
        activation = x # Выходные сигналы слоя (первоначально соответствует
выходным сигналам 1-го слоя или входным сигналам сети)
        activations = [x] # Список выходных сигналов по всем слоям
(первоначально содержит только выходные сигналы 1-го слоя)
        zs = [] # Список активационных потенциалов по всем слоям
(первоначально пуст)

        # Прямое распространение
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b # Считаем активационные потенциалы
текущего слоя
            zs.append(z) # Добавляем элемент (активационные потенциалы
слоя) в конец списка
            activation = sigmoid(z) # Считаем выходные сигналы текущего
слоя, применяя сигмоидальную функцию активации к активационным потенциалам
слоя

            activations.append(activation) # Добавляем элемент (выходные
сигналы слоя) в конец списка

        # Обратное распространение
        delta = (self.cost).delta(zs[-1], activations[-1], y) # Считаем
меру влияния нейронов выходного слоя L на величину ошибки (BP1)
        nabla_b[-1] = delta # Градиент dC/db для слоя L (BP3)
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())# Градиент
dC/dw для слоя L (BP4)

        for l in range(2, self.num_layers):
            z = zs[-l] # Активационные потенциалы l-го слоя (двигаемся по
списку справа налево)
            sp = sigmoid_prime(z) # Считаем сигмоидальную функцию от
активационных потенциалов l-го слоя
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp #
Считаем меру влияния нейронов l-го слоя  на величину ошибки (BP2)
            nabla_b[-l] = delta # Градиент dC/db для l-го слоя (BP3)
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w) # Градиент dC/dw для l-го слоя (BP4)
```

Определение прогресса в обучении

Для определения прогресса в обучении нейронной сети используется функция `accuracy` . результатом работы данной функции является число равное количеству правильно распознанных рукописных цифр. Скопируйте в раздел описания класса `Network` программный код метода `accuracy`, представленный ниже:

```
    def accuracy(# Оценка прогресса в обучении
         self # Указатель на объект класса
         , data # Набор данных (выборка)
         , convert=False # Признак необходимости изменять формат
представления результата работы нейронной сети
         ):
        if convert:
            results = [(np.argmax(self.feedforward(x)), np.argmax(y))
                       for (x, y) in data]
        else:
            results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in data]
        return sum(int(x == y) for (x, y) in results)
```

Данный метод может быть вызван для обучающей, проверочной или тестовой выборки. Обе выборки состоят из пар (x, y), где x – вектор размерности 784, содержащий изображение цифры, а y, в зависимости от выборки, либо целое числовое значение цифры, изображенной на картинке (в случае проверочной выборки), либо вектор размерности 10 – ожидаемый выходной результат (в случае обучающей выборки). Ответ нейронной сети определяется как номер нейрона в выходном слое, имеющего наибольшее значение функции активации.

Флаг `convert` указывает на то, как должны обрабатываться данные, это обусловлено отличием форматов возможных входных данных метода `accuracy`. Таким образом, когда метод `accuracy` вызывается для обучающей выборки, ожидаемый выходной результат определяется как номер координаты вектора, имеющей наибольшее значение, это означает, что флаг `convert` должен равняться «true».

Метод `accuracy` вызывается в методе `SGD` после завершения очередной эпохи обучения в случае если установлен соответствующий флаг.

Подсчет значения функции потерь по всей выборке

Метод `total_cost` так же вызывается в методе `SGD` после завершения очередной эпохи обучения в случае если установлен соответствующий флаг. В данном методе осуществляется подсчет значения функции потерь по всему набору данных `data`. Выбор значения для параметра `convert` определяется по аналогии с методом `total_cost`. Скопируйте в раздел описания класса `Network` программный код метода `total_cost`:

```
    def total_cost((# Значение функции потерь по всей выборке
         self # Указатель на объект класса
         , data # Набор данных (выборка)
         , lmbda # Параметр сглаживания L2-регуляризации
         , convert=False # Признак необходимости изменять формат
представления результата работы нейронной сети
         ):
        cost = 0.0
        data = list(data)
        for x, y in data:
            a = self.feedforward(x)
            if convert: y = vectorized_result(y)
            cost += self.cost.fn(a, y)/len(data)
        cost += 0.5*(lmbda/len(data))*sum(
            np.linalg.norm(w)**2 for w in self.weights)
        return cost
```

Для того, чтобы иметь возможность сохранять обученную сеть добавим метод `save`. Имя файла, в который сохраняется сеть передается в качестве параметра. Скопируйте в раздел описания класса `Network` программный код метода `save`:

```
    def save(self, filename): # Запись нейронной сети в файл
        data = {"sizes": self.sizes,
                "weights": [w.tolist() for w in self.weights],
                "biases": [b.tolist() for b in self.biases],
                "cost": str(self.cost.__name__)}
        f = open(filename, "w")
        json.dump(data, f)
        f.close()
```

Для того чтобы загрузить нейронную сеть из файла добавим метод `load`. Добавьте этот код в раздел описаний программы `network2.py`.

```
def load(filename): # Загрузка нейронной сети из файла

    f = open(filename, "r")
    data = json.load(f)
    f.close()
    cost = getattr(sys.modules[__name__], data["cost"])
    net = Network(data["sizes"], cost=cost)
    net.weights = [np.array(w) for w in data["weights"]]
    net.biases = [np.array(b) for b in data["biases"]]
    return net
```

Определение сигмоидальной функции

В качестве функции активации для нейронов сети используется сигмоидальная функция, вычисляющая выходной сигнал искусственного нейрона. Ниже представлен код, для определения функции (`sigmoid`) и подсчета производной сигмоидальной функции (`sigmoid_prime`). Добавьте этот код в раздел описаний программы `network2.py`.

```
def sigmoid(z): # Определение сигмоидальной функции активации
    return 1.0/(1.0+np.exp(-z))


def sigmoid_prime(z): # Производная сигмоидальной функции
    return sigmoid(z)*(1-sigmoid(z))
```

Для преобразования числа в вектор-столбец (10-мерный numpy массив), используется функция `vectorized_result`. Добавьте этот код в раздел описаний программы `network2.py`.

```
def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

2. Запуск программы

Запустите среду разработки IDLE, если она не была запущена. Если среда уже запущена, осуществите перезапуск среды, выбрав пункт `Shell/Restart Shell` (`Ctrl+F6`). В среде разработки IDLE последовательно выполните следующие команды для установки рабочего каталога на примере `C:\NeuralNetwork`:

```
    >>>import os
    >>>os.chdir ('C:\\NeuralNetwork\\Network2')
```

Следующие команды используются для подключения модуля mnist_loader и инициализации наборов данных для обучения нейронной сети:

```
    >>>import mnist_loader
    >>>training_data, validation_data, test_data =
mnist_loader.load_data_wrapper()
```

Подключите созданный Вами модуль `network2.py`:

```
    >>>import network2
```

Создайте нейронную сеть для распознавания рукописных цифр:

```
    >>> net = network2.Network([784, 30, 10],
cost=network2.CrossEntropyCost)
```

Запустите процедуру обучения созданной нейронной сети, включающую 30 эпох:

```
>>> net.SGD(training_data, 30, 10, 0.5, lmbda =
5.0,evaluation_data=validation_data, monitor_evaluation_accuracy=True,
monitor_evaluation_cost=True, monitor_training_accuracy=True,
monitor_training_cost=True)
```

Задание 4. Выбор топологии нейронной сети и подбор параметров обучения

Используя программу для распознавания рукописных цифр, написанную в задании 3, создайте и обучите несколько нейронных сетей. Создаваемые сети должны иметь разную топологию. Для каждой сети попытайтесь подобрать оптимальные параметры для запуска процедуры обучения методом градиентного спуска.

Примеры запусков:

```
>>> net = network2.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 10.0, lmbda = 1000.0,
evaluation_data=validation_data, monitor_evaluation_accuracy=True)
```

Ответьте на вопросы:

5) Какие параметры влияют на обучение нейронной сети? Объясните характер их влияния?
6) Какова максимальная точность распознавания, которую вам удалось достичь при обучении нейронной сети (с указанием топологии нейронной сети)?
7) Как повлияло изменение топологии нейронной сети на качество ее обучения?
8) Как повлияло изменение параметров запуска метода градиентного спуска на качество обучения нейронной сети?

Задание 5. Функция активации Softmax

Задание 6. Сверточные нейронные сети (Theano)