# 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 перекрестнойэнтропии*, 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--- """

Добавьте в раздел описаний основной 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.

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***Постарайтесь ответить на вопрос:*** *Какой эффект достигается за счет указанных изменений при инициализации весов?*

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

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): # Cтоимостная 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): # Cтоимостная 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.

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***Постарайтесь ответить на вопросы:*** *Какая стоимостная функция использовалась в предыдущей версии программы? Какой эффект достигается благодаря использованию стоимостной функции на основе перекрестной энтропии?*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*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 (, где – вектор входных сигналов, а – 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. Если при вызове метода SGD, в качестве параметра, была передана оценочная выборка evaluation\_data, переменные monitor\_evaluation\_cost, monitor\_evaluation\_accuracy и др. равны «True», то на экран выводится информация о значениях функции потерь по всей проверочной выборке и достигнутый прогресс в обучении, вычисляемый 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 , где – вектор размерности 784, содержащий изображение цифры, а , 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. Так, например, когда метод accuracy вызывается для обучающей выборки, то 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, как и в программе network.py, 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 функций sigmoid и sigmoid\_prime в раздел описаний программы network2.py.

Для преобразования числа в вектор-столбец (10-мерный numpy массив), используется функция vectorized\_result. Добавьте код функции vectorized\_result в раздел описаний программы network2.py.

## 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)

Сохраните обученную сеть в файл.