

Лабораторная работа «Нейронная сеть для распознавания рукописных цифр»

Цель: написать компьютерную программу на языке Python, создающую и обучающую нейронную сеть для распознавания рукописных цифр.

Исходные данные:

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

Методические указания к лабораторной работе

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('Количество 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 *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 выводится 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') # загружаем 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])
```

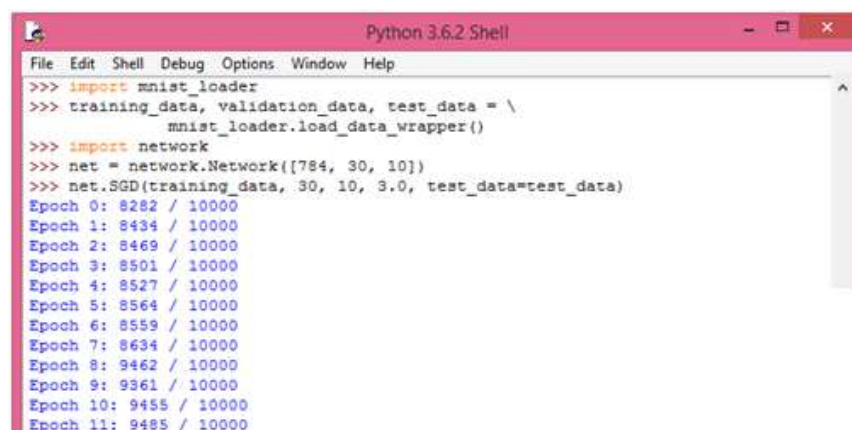
Параметры, указанные при вызове данного метода, определяют топологию создаваемой сети. Таким образом, в результате выполнения команды будет создана сеть, состоящая из трех слоев: входной слой сети состоит из 784-х нейронов; внутренний слой из 30 нейронов и выходной слой из 10 нейронов.

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%.



```
Python 3.6.2 Shell
File Edit Shell Debug Options Window Help
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
mnist_loader.load_data_wrapper()
>>> import network
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
Epoch 0: 8282 / 10000
Epoch 1: 8434 / 10000
Epoch 2: 8469 / 10000
Epoch 3: 8501 / 10000
Epoch 4: 8527 / 10000
Epoch 5: 8564 / 10000
Epoch 6: 8559 / 10000
Epoch 7: 8634 / 10000
Epoch 8: 9462 / 10000
Epoch 9: 9361 / 10000
Epoch 10: 9455 / 10000
Epoch 11: 9485 / 10000
```

Рис.2. Результат работы программы `Network1`.