Нейронна мережа для розпізнавання рукописних цифр»

Мета: написати комп'ютерну програму на мові Python, створюючу та навчальну нейронну систему для розпізнавання рукописних цифр.
Вихідні дані:
• Кількість шарів нейронної мережі: 3.
• Вхідні дані зображення для нейронної мережі: розміром 28 × 28 пікселів.
• програмування: Python 3.6 і вище.
• Використовувана бібліотека: NumPy.

Методичні вказівки до лабораторної роботи

1) Організація середовища розробки

Систему програмування на мові Python 3.6.4 для Windows можна завантажити з офіційного сайту https://www.python.org/downloads/windows/ (рекомендуется использовать Windows x86 executable installer). ***Перед установкою необхідно вибрати пункт «Додати Python до PATH».***

В якості самоучителя на мові Python можна використовувати ресурс https://pythonworld.ru/samouchitel-python.

1) Встановлення бібілотеки NumPy
1. Запустіть додаток «Командна строка» (для цього виберіть cmd у вікні пошуку панелі завдань
Windows).

```
pip3 install numpy
```

1. Запустіть команду для втановлення пакету:

2) Встановлення робочої папки проєкту

3) Створіть каталог NeuralNetwork, у якому ви будете зберігати вихідні тексти програми, створені під час виконання практичних завдань (наприклад, C:\NeuralNetwork). У каталозі NeuralNetwork створіть підкаталог Network1, у якому будуть зберігатися вихідні коди завдання

1.Створення нейронної мережі

Запустіть середовище розробки (для запуску середовища розробки IDLE, виберіть idle у вікні панелі пошуку задач Windows). Створіть новий файл для програми (меню Файл/Новий файл). Збережіть цей файл у каталозі Network1 під іменем мережі (меню File/Save). Розширення .py буде встановлено за умовчанням.

Скопіюйте у вікні програми network.py наступні команди та впишіть дані

```
"""""""""""""""""""""""""""""""""""""""
network.py
Модуль створення та навчання нейронної
мережі для розпізнавання рукописних
цифр з використанням методу
градієнтного спуску.
Группа:<Указати номер групи>
ФИО:<Указати піб
студента>"""""""""""""""""""""""""""""""
"""""""""

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))
```

Збережіть файл network.py і виконайте програму network. Щоб запустити програму виконання, виберіть Run/Run Module (або натисніть F5). У результаті буде створено об'єкт класу Мережа, що задає трьохрівневу нейронну сеть з окремими параметрами. При створенні об'єкта класу мережі веса і розміщення ініціалізуються випадковим чином. Для ініціалізації цієї величини використовується функція np.random.randn з бібліотеки NumPy. Дана функція генерує числа з нормальним розподілом для масиву заданої розмірності.

Визначення сигмоїдальної функції

У якості функції активації для мережі нейронів використовується сигмоїдна функція, що вичисляє вихідний сигнал штучного нейрона. Ниже

представлений код для визначення функції. Додайте цей код у розділ описаної програми
network.py.

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

Зверніть увагу, що для опису сигмоїдальної функції активації використовується функція для вирахування експонентів з бібліотеки NumPy, що дозволяє передавати масив у якості вхідного параметра сигмоїдальної функції. У цьому випадку функція експонентів застосовується поелементно, тобто є у векторизованій формі.

Метод feedforward

Додайте метод feedforward в опис класу Network.

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

Цей метод здійснює підрахунок вихідних сигналів нейронної мережі при заданих вхідних сигналах. Параметр $\boldsymbol{a}$ є масивом $n \times 1$, де n – кількість нейронів вхідного шару. Функція np.dot вичисляє произведение матриці. Для підрахунку вихідних значень нейронної мережі необхідно один раз викликати метод feedforward, в результаті чого вихідні сигнали будуть послідовно вираховані для всіх шарів нейронної мережі.

4) Навчання нейронної мережі

Для реалізації механізму навчання створюваної нейронної мережі додаємо метод SGD, який реалізує стохастичний градієнтний спуск. Метод має такі параметри:

«Training_data» – навчальна вибірка, яка складається з пар виду $(\vec{x}, \vec{y})$, де $\vec{x}$ – вектор вхідних сигналів, а $\vec{y}$ – очікуємий вектор вихідних сигналів;

«epochs» – кількість епох навчання;

«mini_batch_size» - розмір підвибірки;

«eta» - швидкість навчання;

«test_data» - (необов'язковий параметр); якщо цей аргумент не пустий, то програма після кожної епохи навчання здійснює оцінку роботи мережі та показує досягнутий прогрес.

Додайте програмний код методу SGD в розділ опису класу Мережа:

```
    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
```

Цей програмний код працює таким чином. На початку кожної епохи навчання елементи навчальної виборки перемішуються (переставляються у випадковому порядку) за допомогою функції shuffle() із випадкової бібліотеки, після чого навчальна вибірка послідовно розбивається на підвиборки довжини mini_batch_size. Для кожної підвиборки виконується один крок градієнтного спуску за допомогою методу update_mini_batch (см. нижче). Після того, як виконано останній крок градієнтного спуску, т.е. виконується метод update_mini_batch для останньої підвиборки, на екрані якого виводиться досягнутий прогрес в навчанні нейронної мережі, що вираховується на тестовому виборі за допомогою методу evaluate (див. нижче).

Аналізуючи програмний код методу update_mini_batch, можна побачити, що основна частина вираховується в результаті виклику методу backprop (див. нижче). Цей метод класу мережі реалізує алгоритм зворотного поширення помилок, який є швидким способом вирахування градієнта стоїмкової функції. Таким чином, метод update_mini_batch вираховує всі градієнти для кожного прецеденту $(\vec{x}, \vec{y})$ у підвиборці, а також оновлює масу та зміщення нейронної мережі. Додайте код методу update_mini_batch в розділ опису класу Мережа:

```
    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
```

Скопіюйте в розділ опис класу

```
    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 класу NETWORK програмний код методу оцінки, що демонструє прогрес в навчанні:

```
    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)
```

5) Робота з базою даних MNIST

Для навчання нейронної мережі будемо використовувати архів http://deeplearning.net/data/mnist/mnist.pkl.gz на сайті Лабораторії машинного навчання Університету Монреаля, сформований на основі бази даних MNIST, який містить 70 000 зображень рукописних цифр, розділених на три набори:

`training_data` – набір із 50 000 зображень, призначений для навчання нейронних сетей;

`validation_data` – набір із 10 000 зображень, призначений для поточної оцінки роботи алгоритму навчання та підбору параметрів навчання (використовується в останніх лабораторних роботах);

`test_data` – набір із 10 000 зображень призначений для перевірки роботи нейронної мережі.

Кожен набір складається з двох списків: списку зображень (у градаціях серого) і відповідного списку цифр у діапазоні від 0 до 9. Зображення представлено у вигляді одномерного масиву numpy розміром 784 = 28 × 28 значень від 0 до 1, де 0 відповідає чорному кольору піксела, а 1 – білому.

Функції для роботи з базою даних MNIST целесообразно винести в окремий файл. Створіть новий файл mnist_loader і збережіть його в директорії Network1. Скопіюйте у вікні програми mnist_loader.py наступні команди та

впишіть свої дані:

```
"""
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 використання бази даних MNIST в нашій програмі необхідно відкоригувати формати наборів training_data, validation_data і test_data. Це робиться у функції load_data_wrapper. Скопіюйте у файл mnist_loader наступний програмний 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)
```

Дана функція перетворює training_data в список, що містить 50 000 пар (x, y), де x є 784-мерним numpy-масивом, що містить вхідне зображення, а y – це 10-мірний numpy-массив, що представляє собою вектор, у якого координати з порядковим номером, відповідною цифрою на зображенні, вирівнюється одиницею, а інші координати нульові. Аналогічні перетворення виконуються для наборів validation_data і test_data.

Для перетворення чисел у вектор-столбец (10-мірний масив numpy) використовується наступна функція vectorized_result. Скопіюйте її програмний код у файл mnist_loader.

```
def vectorized_result(j):

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

□ Збережіть і закрийте mnist_loader.py.

7) Запуск програми

У середовищі розробки IDLE послідовно виконайте наступні команди для встановлення робочого каталогу на прикладі C:\NeuralNetwork:

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

Наступні команди використовуються для підключення модуля mnist_loader та ініціалізації наборів даних для навчання нейронної мережі:

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

Подключите модуль `network.py`:

```
>>>import network
```

При цьому виконується написана в ньому програма, що виводить інформацію про нейронну мережу.

Создайте нейронную сеть для распознавания рукописных цифр:

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

Параметри, вказані при виклику даного методу, визначають топологію створюваної мережі. Таким чином, в результаті виконання команди буде створена сеть, що складається з трьох шарів: вхідний шар мережі складається з 784-х нейронів; внутрішній шар з 30 нейронів і вихідний шар з 10 нейронів.

Запустіть процедуру навчання створеної нейронної мережі, що включає 30 років:

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

Параметри, указані при виклику методу SGD: навчальна вибірка, кількість епохи навчання, розмір підвиборки, швидкість навчання, тестувальна вибірка.

Навчання может занять несколько минут. В ході навчання буде видаватися інформація про пройдені епохи (см. рис. 2). Для кожної епохи виводиться відношення кількості правильно розпізнаних цифр до загальної кількості цифр у тестовому виборі. Наприклад, запис `Epoch 6: 9374 / 10000` говорить про те, що, в результаті епохи навчання з номером 6 досягнута точність розпізнавання $\frac{9374}{10000} \approx 0.94$, що складає 94%.

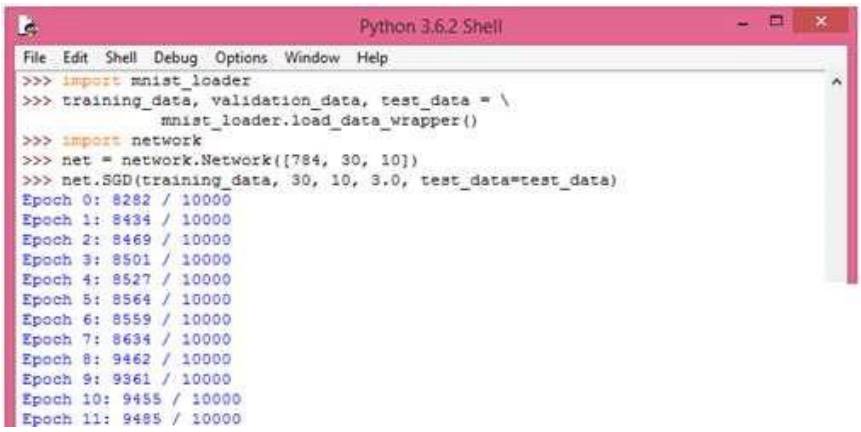

Рис.2.