МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФГБОУ ВО «Уральский государственный экономический университет»

В.П. Часовских

Интеллектуальные технологии и кибербезопасность цифрового предприятья

38.04.05 – бизнес-информатика направленность интеллектуальное управление цифровым предприятием»

Лабораторная работа № 4

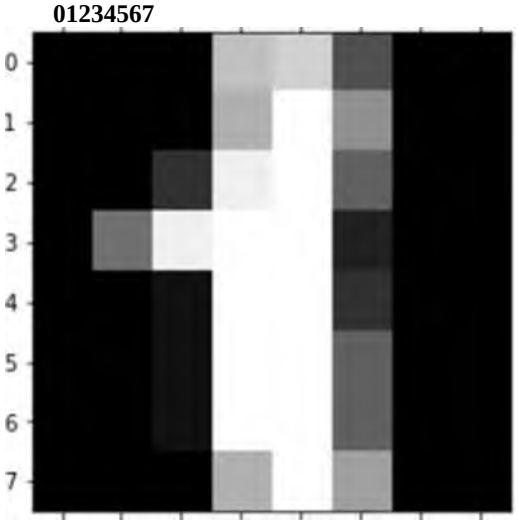
Нейронные сети

Екатеринбург 2022

Имплементация нейросети языке Python

Напишем код, который прогнозирует, основываясь на данных MNIST. База данных MNIST - это набор примеров в нейронных сетях и глубинном обучении. Она включает в себя изображения цифр, написанных от руки, с соответствующими ярлыками, которые объясняют, что это за число. Каждое изображение размером 8x8 пикселей. В этом примере мы используем сети данных MNIST для библиотеки машинного обучения scikit learn в языке программирования Python . Пример такого изображения можно увидеть под кодом:

```
from sklearn.datasets import load_digits digits = load_digits()
print(digits.data.shape) import matplotlib.pyplot as pit pit.gray()
pit.matshow(digits.images[1]) plt.show()
```



Код, который мы собираемся написать в нашей нейронной сети, будет анализировать цифры, которые изображают пиксели на изображении. Для начала, нам нужно отсортировать входные данные. Для этого мы сделаем две следующие вещи:

01. Масштабировать данные.
02. Разделить данные на тесты и учебные тесты.

1. Масштабирование данных

Почему нам нужно масштабировать данные? Во-первых, рассмотрим представление пикселей одного из сетов данных:

```
digits.data[0, : ]
Out[2]:

    array([0., 0., 5., 13., 9., 1., 0., 0., 0., 0., 13., 15., 10., 15., 5., 0., 0.,
       3., 15., 2., 0., 11., 8., 0., 0., 4., 12., 0., 0., 8., 8., 0., 0., 5., 8., 0.,
       0., 9., 8., 0., 0., 4., 11., 0., 1., 12., 7., 0., 0., 2., 14., 5., 10., 12.,
       0., 0., 0., 0., 6., 13., 10., 0., 0., 0.
    ])
```

Заметили ли вы, что входные данные меняются в интервале от 0 до 15? Достаточно распространенной практикой является масштабирование входных данных так, чтобы они были только в интервале от [0,1], или [1,1]. Это делается для более легкого сравнения различных типов данных в нейронной сети. Масштабирование данных можно легко сделать через библиотеку машинного обучения scikit learn:

```
X[0,:]
Out[3]:
array([ 0.        , -0.33501649, -0.04308102, 0.27407152, -0.66447751,
       -0.84412939, -0.40972392, -0.12502292, -0.05907756, -0.62400926,
        0.4829745 ,  0.75962245, -0.05842586,  1.12772113,  0.87958306,
       -0.13043338, -0.04462507,  0.11144272,  0.89588044, -0.86066632,
       -1.14964846,  0.51547187,  1.90596347, -0.11422184, -0.03337973,
        0.48648928,  0.46988512, -1.49990136, -1.61406277,  0.07639777,
        1.54181413, -0.04723238,  0.        ,  0.76465553,  0.05263019,
       -1.44763006, -1.73666443,  0.04361588,  1.43955804,  0.
       -0.06134367,  0.8105536 ,  0.63011714, -1.12245711, -1.06623158,
        0.66096475,  0.81845076, -0.08874162, -0.03543326,  0.74211893,
        1.15065212, -0.86867056,  0.11012973,  0.53761116, -0.75743581,
       -0.20978513, -0.02359646, -0.29908135,  0.08671869,  0.20829258,
       -0.36677122, -1.14664746, -0.5056698 , -0.19600752])
```

Стандартный инструмент масштабирования в scikit learn нормализует данные через вычитание и деление. Вы можете видеть, что теперь все данные находятся в интервале от -2 до 2. По же на счет выходных данных yy, то обычно нет необходимости их масштабировать.

2 Создание тестов и учебных наборов данных

В машинном обучении появляется такой феномен, который называется "переобучением". Это происходит, когда модели, во время учебы, становятся слишком запутанными - они достаточно хорошо обучены, но когда им передаются новые данные, которые они никогда на "видели", то результат, который они выдают, становится плохим. Иными словами, модели генерируются не очень хорошо. Чтобы убедиться, что мы не создаем слишком сложные модели, обычно набор данных разбивают на учебные наборы и тестовые наборы. Учебный набором данных, на которых модель будет учиться, а тестовый набор - это данные, на которых модель будет тестироваться после завершения обучения. Количество учебных данных должно быть всегда больше тестовых данных. Обычно они занимают 60-80% от набора данных.

Опять же, scikit learn легко разбивает данные на учебные и тестовые наборы:

```
from sklearn.model_selection import train_test_split
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
```

В этом случае мы выделили 40% данных на тестовые наборы и 60% соответственно на обучение. Функция train_test_split в scikit learn добавляет данные рандомно в различные базы данных - то есть, функция не берет первые 60% строк для учебного набора, а то, что осталось, использует как тестовый.

3 Настройка выходного слоя

Для того, чтобы получать результат - числа от 0 до 9, нам нужен выходной слой. Более- менее точная нейросеть, как правило, имеет выходной слой с 10 узлами, каждый из которых выдает число от 0 до 9. Мы хотим научить сеть так, чтобы, например, при цифре 5 на изображении, узел с цифрой 5 в исходном слое имел наибольшее значение. В идеале, мы бы хотели иметь следующий вывод: [0, 0, 0, 0, 0,1,0, 0, 0, 0]. Но на самом деле мы можем получить что-то похожее на это: [0.01,0.1, 0.2, 0.05, 0.3, 0.8, 0.4, 0.03, 0.25, 0.02]. В таком случае мы можем взять крупнейших индекс в исходном массиве и считать это нашим полученным числом.

В данных MNIST нужны результаты от изображений записаны как отдельное число. Нам нужно конвертировать это единственное число в вектор, чтобы его можно было сравнивать с исходным слоем с 10 узлами. Иными словами, если результат в MNIST обозначается как "1", то нам нужно его конвертировать в вектор: [0,1, 0, 0, 0, 0, 0, 0, 0, 0]. Такую конвертацию осуществляет следующий код:

```
import numpy as np

def convert_y_to_vect(y):
    y_vect = np.zeros((len(y), 10)) for i in
    range(len(y)):
        y_vect[i, y[i]] = 1
    return y_vect y_v_train = convert_y_to_vect(y_train)

y_v_test = convert_y_to_vect(y_test)
y_train[0], y_v_train[0]
0ut[8]:
(1, array([ 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]))
```

Этот код конвертирует "1" в вектор [0,1.0, O, 0, 0, 0, 0, 0, 0].

4 Создаем нейросеть

Следующим шагом является создание структуры нейронной сети. Для входного слоя, мы знаем, что нам нужно 64 узла, чтобы покрыть 64 пикселей изображения. Как было сказано ранее, нам нужен выходной слой с 10 узлами. Нам также потребуется скрытый слой в нашей сети. Обычно, количество узлов в скрытых слоях не менее и не больше количества узлов во входном и выходном слоях. Объявим простой список на языке Python , который определяет структуру нашей сети:

```
nn_structure = [64, 30, 10]
```

Мы снова используем сигмоидальную активационную функцию, так что сначала нужно объявить эту функцию и ее производную:

```
def f(x):
    return 1 / (1 + np.exp(-x)) def fderiv(x):
    return f(x) * (1 - f(x))
```

Сейчас мы не имеем никакого представления, как выглядит наша нейросеть. Как мы будем ее учить? Вспомним наш алгоритм из предыдущих разделов:
Рандомно инициализируем веса для каждого слоя W(l1 Когда итерация страницы итерации:

01. Зададим AW и ЛЬ начальное значение ноль.
02. Для экземпляров от 1 до т: а. Запустите процесс прямого распространения через все П| слоев. Храните вывод активационной функции в h^6. Найдите значение 6(П|) выходного слоя. Обновите AW^n Ab(1' для каждого слоя.
03. Запустите процесс градиентного спуска, используя:

гп

Значит первым этапом является инициализация весов для каждого слоя. Для этого мы используем словари в языке программирования Python (обозначается через {}). Рандомные значения предоставляются весам для того, чтобы убедиться, что нейросеть будет работать правильно во время обучения. Для рандомизации мы используем random_sample из библиотеки numpy. Код выглядит следующим образом:

```
import numpy.random as r def setup_and_init_weights(nn_structure):
    W = {} b = {} for 1 in range(l, len(nn_structure)):
        W[l] = r.random_sample((nn_structure[l]> nn_structure[l-l])) b[l] = r.random_sample((nn_structure[l]>))
    return W, b
```

Следующим шагом является присвоение двум переменным AW и ЛЬ нулевых начальных значений (они должны иметь такой же размер, что и матрицы весов и смещений)

```
def init_tri_values(nn_structure):
    tri_W = {}
    trib = {}
    for 1 in range(l, len(nn_structure)):

        tri_W[l] = np.zeros((nn_structure[l], nn_structure[l-l])) tri_b[l] = np.zeros((nn_structure[l],))
    return tri_W, tri_b
```

Далее запустим процесс прямого распространения через нейронную сеть:

```
def feed_forward(x, W, b):
    h = {1: x}
    z = {}
    for l in range(1, len(W) + 1):



    #Если первый слой, то весами является x, в противном случае
    #Это выход из последнего слоя if l == 1:
            node_in = x else:
            node_in = h[l]
        z[l+1] = W[l].dot(node_in) + b[l] # z^A(l+1) = W^A(l)*h^A(l) + b^A(l)
        h[l+1] = f(z[l+1]) # h^A(l) = f(z^A(l))
    return h, z
```

И наконец, найдем выходной слой 6 (n|) и значение 6в скрытых слоях для запуска обратного распространения:

```
def calculate_out_layer_delta(y, h_out, z_out):

    # delta^A(nl) = ~(y_i - h_i^A(nl)) * f'(z_i^A(nl)) return -(y-h_out) * f_deriv(z_out)
```

```
def calculate_hidden_delta(delta_plus_l, w_l, z_l):

    # delta^A(l) = (transpose(W^A(l)) * delta^A(l+l)) * f'(z^A(l)) return np.dot(np.transpose(wl),
    delta_plus_l) * f_deriv(z_l)
```

Теперь мы можем соединить все этапы в одну функцию:

```
def train_nn(nn_structure, Xл y, ^ег_пит=3000, alpha=0.25):
    W, b = setup_and_init_weights(nn_structure) ent = 0 m = len(y) avg cost func =
    [] print('Начало градиентного спуска для {} итераций'.format(iter_num)) while ent
    1:
                        delta[l] = calculate_hidden_delta(delta[l+l], W[l], z[ #
                    triWA(l) = triWA(l) + deltaA(l+l) * transpose(hA(l)) tri_W[l]
                    += np.dot(delta[1+1][:,np.newaxis], np.transpose( # tribA(l) =
                    tribA(l) + deltaA(l+l) tri_b[l] += delta[1+1]
        # запускает градиентный спуск для весов в каждом слое for 1 in
        range(len(nn_structure) - 1, 0, -1):
            W[l] += -alpha * (1.0/m * tri_W[l])
```

Функция сверху должна быть немного объяснена. Во-первых, мы не задаем лимит работы градиентного спуска, основываясь на изменениях или точности функции оценки. Вместо этого, мы просто запускаем её с фиксированным числом итераций (3000 в нашем случае), а затем наблюдаем, как меняется общая функция оценки с прогрессом в обучении. В каждой итерации градиентного спуска, мы перебираем каждый учебный экземпляр (range (len (y)) и запускаем процесс прямого распространения, а после него и обратное распространение. Этап обратного распространения является итерацией через слои, начиная с выходного слоя к началу - range (len (nn_structure), 0,1). Мы находим среднюю оценку на исходном слое (I == len (nn.structure)). Мы также обновляем значение AW и AЬ с пометкой tri_W и tri_b, для каждого слоя, кроме исходного (исходный слой не имеет никакого связи, который связывает его со следующим слоем).

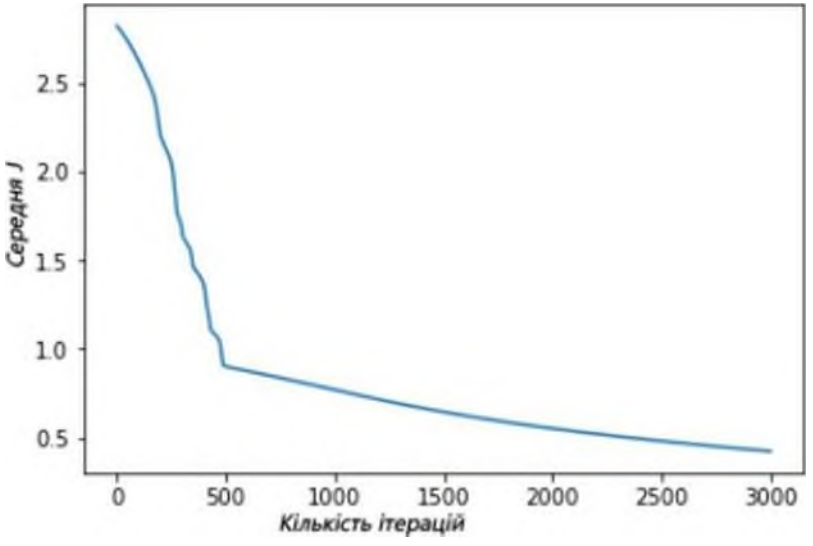
И наконец, после того, как мы прошлись по всем учебным экземплярам, накапливая значение tri_W и tri_b, мы запускаем градиентный спуск и меняем значения весов и смещений:

После окончания процесса, мы возвращаем полученные вес и смещение со средней оценкой для каждой итерации. Теперь время вызвать функцию. Ее работа может занять несколько минут, в зависимости от компьютера.

```
W, b, avg_cost_func = train_nn(nnstructure, X_train, y_v_train)
```

Мы можем увидеть, как функция средней оценки уменьшилась после итерационной работы градиентного спуска:

```
pit.plot(avg_cost_func)
pit.ylabel('Средняя J')
pit.xlabel('Количество итераций')
pit.show()
```



Выше изображен график, где показано, как за 3000 итераций нашего градиентного спуска функция средней оценки снизилась и маловероятно, что подобная итерация изменит результат.

Для этого можно использовать функцию numpy.argmax, она возвращает индекс элемента массива с наибольшим значением:

```
def predict_y(W, b, X, n_layers):
    m = X.shape[0]
    y = np.zeros((m,))
    for i in range(m):

        h, z = feed_forward(X[i, :], W, b) y[i] = np.argmax(h[n_layers])
    return y
```

Теперь, наконец, мы можем оценить точность результата (процент раз, когда сеть выдала правильный результат), используя функцию accuracy_score из библиотеки scikit learn:

```
from sklearn.metrics import accuracy_score y_pred = predict_y(W, b, X_test, 3)
accuracy_score(y_test, y_pred)*100
```

Мы получили результат 86% точности. Звучит довольно неплохо? На самом деле, нет, это довольно низкая точностью. В наше время точность алгоритмов глубинного обучения достигает 99.7%, мы немного отстали.

Теперь мы можем соединить все этапы в одну функцию:

```
                delta[l] = calculate_hidden_delta(delta[l+l], W[l], z[ # triW^A(l) = triW^A(l) + delta^A(l+l) * transpose(h^A(l)) tri_W[l] += np.dot(delta[1+1][:,np.newaxis], np.transpose(l # trib^A(l) = trib^A(l) + delta^A(l+l) tri_b[l] += delta[1+1] # запускает градиентный спуск для весов в каждом слое for 1 in range(len(nn_structure) - 1, 0, -1):

        W[l] += -alpha * (1.0/m * tri_W[l]) b[l] += -alpha * (1.0/m * tri_b[l]) # завершает расчеты общей оценки avg_cost = 1.0/m * avg_cost avg_cost_func.append(avg_cost) ent += 1 return W, b, avg_cost_func
```