Обучение нейронной сети

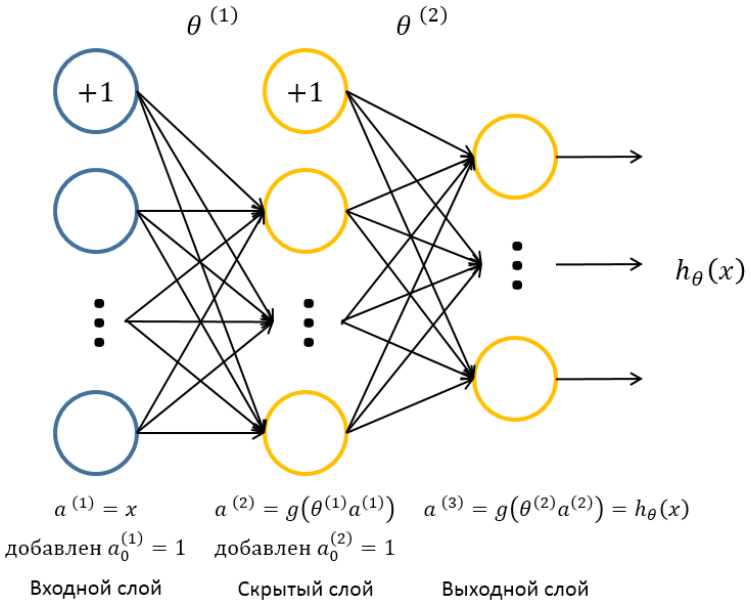
В предыдущей части работы вы реализовали алгоритм прямого распространения для вычисления значения на выходе нейронной сети и использовали его для распознавания рукописных цифр. Параметры нейронной сети были даны заранее. В этой части работы необходимо реализовать алгоритм обучения сети для автоматического вычисления оптимальных параметров.

Для успешного выполнения работы рекомендуется обратиться к материалам по обучению нейронной сети.

Теоретические сведения

Структура нейронной сети

Структура нейронной сети та же, что и в предыдущей чработе. Сеть состоит из трех слоев, средний из которых является скрытым. В первом (входном) слое 400 нейронов по числу распознаваемых признаков (распознаются изображения 20*20=400 пикселей). Во втором (скрытом) слое 25 нейронов. В третьем (выходном) – 10 нейронов по числу распознаваемых цифр от 0 до 9.



Формат входных и выходных данных

Обучающая выборка хранится в файле **training_set.mat** и состоит из 5000 рукописных цифр. Для загрузки данных в скрипт используйте функцию**scipy.io.loadmat**с именем файла в качестве параметра. Функция вернёт словарь, в котором нас интересуют массив по ключам **X** и **y**.

Матрица **X** представляет собой рукописные цифры. Она состоит из 5000 строк по числу тестовых примеров и 400 столбцов по числу точек на изображении. Каждое изображение цифры имеет размеры 20*20 точек, отсюда 400 столбцов в матрице **X**.

Вектор **y** – это вектор-столбец, состоящий из 5000 элементов. Каждый элемент определяет класс распознаваемой цифры. Цифре «0» соответствует класс 10, остальные классы соответствуют своим цифрам.

Матрицы весовых коэффициентов **Theta1** и **Theta2**, необходимые для проверки правильности функции стоимости, находятся в файле **weights.mat** по соответствующим ключам. Матрица **Theta1** связывает первый слой нейронной сети (400 входов + 1 нейрон смещения) со вторым слоем сети (25 нейронов), поэтому имеет размерность 25 строк на 401 столбец. Матрица **Theta2** связывает второй слой (25

нейронов + нейрон смещения) с третьим выходным слоем сети (10 нейронов), поэтому имеет размерность 10 строк на 26 столбцов.

Задание 1.

Реализовать загрузку обучающей выборки из файла **training_set.mat** в переменные **X** и **y**. Реализовать загрузку весовых коэффициентов из файла **weights.mat** в переменные **Theta1**и **Theta2**.

Задание 2.

Определить структуру нейронной сети:

input_layer_size – число нейронов во входном слое сети (400). Соответствует числу столбцов в матрице **X**.

hidden_layer_size – число нейронов в скрытом слое сети (25). Соответствует числу строк в матрице **Theta1**.

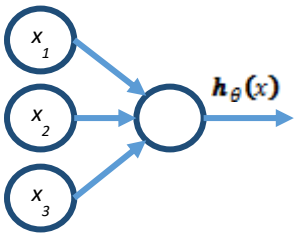
num_labels – число нейронов в выходном слое (10). Соответствует числу распознаваемых цифр, то есть количеству уникальных значений в векторе **y**. Массив уникальных значений можно получить с помощью функции **numpy.unique**.

Функция стоимости

Для подбора оптимальных значений параметров θ методом градиентного спуска, необходимо реализовать вычисление функции стоимости $J(\theta)$, значение которой затем нужно будет минимизировать (минимизировать ошибку нейронной сети на обучающей выборке).

Нейронная сеть напоминает логистическую регрессию. Вернее сказать, логистическая регрессия подобна одному нейрону из нейросети. Функция стоимости для логистической регрессии определяется так:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y log(h_\theta(x)) + (1-y) log(1-h_\theta(x)) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \qquad (1)$$

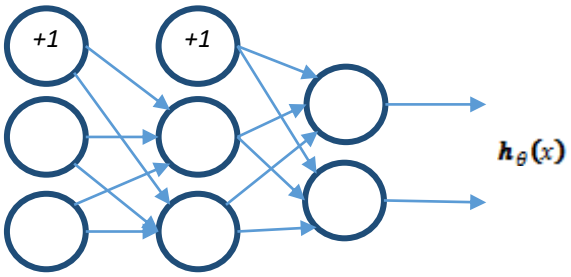


Последнее слагаемое в формуле – регуляризатор. Он нужен для того, чтобы бороться с переобучением (проблемой, когда модель оказалась подогнана под обучающую выборку, но на реальных данных не работает). Чем выше значение параметра λ, тем меньше явление переобучения, но при слишком больших значениях λ модель может оказаться недообученной.

Функция стоимости для нейронной сети очень похожа на функцию стоимости для логистической регрессии. Разница между ними обусловлена лишь тем, что логистическая регрессия по своей сути – это один нейрон, а нейронная сеть – множество нейронов. Сеть может содержать несколько (K) выходов и множество связей θ. Вот как выглядит функция стоимости для нейронной сети:

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2$$

(2)



Отличие формул 1 и 2 лишь в том, что в формуле 2 суммируем по всем K – всем выходам нейронной сети. Кроме того, в регуляризаторе учитываются все параметры θ , связывающие нейроны между собой.

При реализации функции стоимости вам неоднократно придётся умножать матрицы, поэтому внимательно ознакомьтесь с особенностями умножения матриц в библиотеке **NumPy**.

Замечания об умножении матриц в Python

Если умножаются матрица с размерами **(a,b)** на матрицу с размерами **(c,d)**, то размеры **b** и **c** должны совпадать. Итоговая матрица получится размерами **(a, d)**. Здесь **a** – количество строк в матрице, **d** – количество столбцов. Это правило полезно помнить, когда встаёт вопрос, в каком порядке умножать матрицы и нужно ли их транспонировать.

В библиотеке **NumPy** есть 2 типа матриц: **numpy.ndarray** и **numpy.matrixlib.defmatrix.matrix**. Операции умножения матриц и поэлементного умножения матриц для них реализуются по-разному. Таблица ниже демонстрирует различия.

создание	`>>> A = np.array([[1, 2, 3], [2, 3, 4]])` `>>> B = np.array([[3, 2, 1], [4, 3, 2]])`	`>>> A = np.matrix([[1, 2, 3], [2, 3, 4]])` `>>> B = np.matrix([[3, 2, 1], [4, 3, 2]])`

type	`>>> type(A)`
	`<class 'numpy.ndarray'>`
	`>>> type(B)`
	`<class 'numpy.ndarray'>`
		`>>> type(A)`
		`<class`
		`'numpy.matrixlib.defmatrix.matrix'>`
		`>>> type(B)`
		`<class`
		`'numpy.matrixlib.defmatrix.matrix'>`

умножение	`>>> np.dot(A, B.T)`
	`array([[10, 16],`
	`[16, 25]])`
		`>>> A * B.T`
		`matrix([[10, 16],`
		`[16, 25]])`

поэлементное умножение	`>>> A * B`
	`array([[3, 4, 3],`
	`[8, 9, 8]])`
		`>>>np.multiply(A, B)`
		`matrix([[3, 4, 3],`
		`[8, 9, 8]])`

Официальная документация рекомендует использовать тип **numpy.ndarray** при вычислениях.

Задание 3.

Перейти в файл **cost_function.py** и реализовать код вычисления ошибки нейронной сети.

В первую очередь необходимо определить **m** – количество примеров в обучающей выборке. Оно соответствует количеству строк в матрице **X**.

Затем необходимо вычислить значение на выходе нейронной сети для всех примеров обучающей выборки. Для этого воспользуйтесь приведёнными ниже формулами:

$$a^{(1)} = X$$

$$z^{(2)} = a^{(1)}\theta^{(1)T}$$

$$a^{(2)} = sigmoid\left(z^{(2)}\right)$$

$$a^{(2)} \rightarrow \text{прибавить единичный столбец функцией } add_{zero}{}_{feature\left(a^{(2)}\right)}$$

$$z^{(3)} = a^{(2)}\theta^{(2)T}$$

$$a^{(3)} = \boldsymbol{h}_\theta(x) = sigmoid\left(z^{(3)}\right)$$

Эти формулы позволяют в векторной форме вычислить отклик нейронной сети сразу для всех примеров матрицы **X**.

Далее необходимо вычислить усреднённую ошибку нейронной сети по формуле 2 (без учёта регуляризатора). Матрицы **y**и $\boldsymbol{h}_\theta(x)$ имеют размерность 5000×10. Это означает, что каждая строка содержит информацию по отдельному примеру из выборки, а каждый элемент строки содержит информацию об отдельном выходе нейронной сети для этого примера. Ошибка для отдельного примера и выхода нейронной сети может быть посчитана по формуле:

$$-y_k^{(i)} \cdot log\left(\boldsymbol{h}_\theta\left(x^{(i)}\right)_k\right) - \left(1 - y_k^{(i)}\right) \cdot log\left(1 - \boldsymbol{h}_\theta\left(x^{(i)}\right)_k\right)$$

Здесь **i** – индекс по строкам матриц y и $\boldsymbol{h}_\theta(x)$. **k** – индекс по столбцам этих матриц. Но так как в библиотеке **numpy** поддерживаются поэлементные операции над матрицами, то не нужно для вычислений ошибок реализовывать два вложенных цикла (по **i**и по **k**). **Numpy** может вычислить ошибку для всех элементов матриц сразу. В конце останется только найти сумму элементов полученной матрицы и поделить её на **m**.

На последнем этапе необходимо учесть регуляризатор. Для его вычисления необходимо матрицы **Theta1** и **Theta2** поэлементно возвести в квадрат, затем найти сумму их элементов. Полученная сумма умножается на коэффициент $\frac{\lambda}{2m}$ (см. формулу 2).

В конце необходимо к ошибке по всем примерам прибавить регуляризатор и вернуть результат.

Если функция **cost_function** реализована правильно, она должна вернуть результат **0.287629165** для **lambda_coef=0** и **0.384487796** для **lambda_coef=1**.

Алгоритм обратного распространения

Помимо вычисления функции стоимости $J(\theta)$, в алгоритме градиентного спуска требуется вычислять частные производные от функции стоимости по параметрам $\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$. Это нужно для того, чтобы корректно изменять значения параметров θ нейронной сети.

Для вычисления частных производных используется алгоритм обратного распространения ошибки. Идея состоит в том, что мы пропускаем примеры из обучающей выборки через нейронную сеть, а затем сравниваем её результат с ожидаемыми ответами из обучающей выборки. Эта разница далее распространяется по нейронной сети в обратную сторону, чтобы увидеть, какова была ошибка в каждом нейроне сети. Затем параметры нейронной сети корректируются так, чтобы ошибки уменьшились. Этот

процесс итеративный и продолжается до тех пор, пока не достигнут заданный уровень ошибки или не выполнено заданное количество итераций.

На псевдокоде алгоритм обратного распространения ошибки можно описать так:

Дана обучающая выборка: {($x^{(1)}$, $y^{(1)}$), ($x^{(2)}$, $y^{(2)}$), …, ($x^{(m)}$, $y^{(m)}$)}

Задаем $\Delta_{ij}^{(l)} = \mathbf{0}$ для всех l, i, j – это нужно для взятия производных $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$

Цикл по i от 1 до m (по всем примерам из обучающей выборки):

Задаём $a^{(1)} = x^{(i)}$

С помощью прямого распространения вычисляем $a^{(l)}$ для $l = 2, 3, \ldots, L$

Используя $y^{(i)}$, вычисляем ошибку в последнем слое $\delta^{(L)} = a^{(L)} - y^{(i)}$

Вычисляем $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

$$\Rightarrow \quad \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Рассмотрим подробнее алгоритм обратного распространения ошибки, приведенный выше.

С помощью прямого распространения мы нашли значения на выходе каждого нейрона сети $a^{(l)}$. Теперь нам нужно найти ошибку в каждом нейроне. Обозначим:

$\delta_j^{(l)}$ –это ошибка в j-м нейроне l-го слоя нейронной сети.

Вычислить значения ошибок на любом слое сети можно так (на примере нейронной сети из 3-х слоев):

$\delta_j^{(3)} = a_j^{(3)} - y_j = h_\Theta(x) - y_j$ или $\delta^{(3)} = a^{(3)} - y$

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'\left(z^{(2)}\right), \text{ где } g'(z^{(2)}) = a^{(2)} .* \left(1 - a^{(2)}\right)$$

$\delta^{(1)}$ считать не нужно, так как первый слой сети – это входные данные.

Аккумулируем ошибки в переменных $\boldsymbol{\Delta^{(1)}}$ и $\boldsymbol{\Delta^{(2)}}$ по всем примерам выборки.

Затем вычисляем частные производные по формулам:

$$D^{(1)} = \frac{\Delta^{(1)}}{m} + \begin{cases} \lambda\theta^{(1)}, & j \neq \mathbf{0} \\ 0, & j = \mathbf{0} \end{cases}$$

$$D^{(2)} = \frac{\Delta^{(2)}}{m} + \begin{cases} \lambda\theta^{(2)}, & j \neq \mathbf{0} \\ 0, & j = \mathbf{0} \end{cases}$$

Обратите внимание, что регуляризатор считается для всех нейронов, кроме нейрона смещения ($j \neq \mathbf{0}$).

Вычисленные частные производные представляют собой матрицы, имеющие ту же размерность, что и матрицы **Theta1** и **Theta2**. Частные производные используются затем при градиентном спуске для обновления значений матриц **Theta1** и **Theta2**.

Задание 4.

Необходимо реализовать вычисление производной от сигмоиды, которая используется для вычисления ошибок в l-том слое нейронной сети. Для этого откройте файл **sigmoid_gradient.py** и реализуйте функцию **sigmoid_gradient**.

Вычисление производной сигмоиды выполняется по формуле:

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

Где

$$g(z) = sigmoid(z) = \frac{1}{1 + e^{-z}}$$

Обратите внимание, что в первой формуле при перемножении $g(z)$ на $(1 - g(z))$ используется поэлементное умножение матриц.

Если функция реализована вами правильно, то производная сигмоиды в точках **[-1, -0.5, 0, 0.5, 1]** должна выдавать следующий результат:

```
[ 0.19661193  0.23500371  0.25        0.23500371  0.19661193]
```

Кроме того, постройте график сигмоиды с одинаковым масштабом по осям абсцисс и ординат и в указанных точках **(-1, -0.5, 0, 0.5, 1)** проведите касательные к графику. Убедитесь, что тангенс наклона касательных соответствует значениям, которые выдал скрипт.

На заключительном этапе необходимо реализовать функцию **gradient_function**, которая будет использована при градиентном спуске для оптимизации весов нейронной сети. Для оптимизации используется функция **minimize** из библиотеки **scipy.optimize**, которая за 100 итераций вычисляет оптимальные коэффициенты **Theta**. Код оптимизации уже реализован в файле **run.py**, вам не нужно писать его самостоятельно.

Задание 5.

Вам необходимо реализовать функцию **gradient_function**, которая будет использоваться на каждой итерации градиентного спуска.

Для начала определите количество примеров в выборке и сохраните в переменной **m**, так же, как в **cost_function**.

Затем реализуйте вычисление отклика нейронной сети для всех примеров из выборки. Код в точности соответствует коду в функции **cost_function**.

Затем рассчитайте ошибки в третьем слое сети (выходном) и во втором слое (скрытом). Используйте для этого следующие формулы:

$$\delta^{(3)} = a^{(3)} - y$$

$$\delta^{(2)} = \delta^{(3)}\theta^{(2)} .* g'(z^{(2)})$$

Оператор **.***–это поэлементное умножение. Обратите внимание, что когда будете перемножать $\delta^{(3)}\theta^{(2)}$, необходимо отбросить первый столбец полученной матрицы, так как он имеет отношение к нейрону смещения и для алгоритма обратного распространения ошибки не интересен.

Далее вычислите $D^{(1)}$ и $D^{(2)}$ – частные производные, необходимые для градиентного спуска. Используйте для этого формулы:

$$D^{(1)} = \frac{\delta^{(2)T} a^{(1)}}{m}$$

$$D^{(2)} = \frac{\delta^{(3)T} a^{(2)}}{m}$$

В заключение необходимо учесть регуляризатор. Все элементы матриц $D^{(1)}$ и $D^{(2)}$, кроме первого столбца, поэлементно умножьте на $\frac{\lambda}{m}\theta^{(1)}$ и $\frac{\lambda}{m}\theta^{(2)}$ соответственно. В матрицах $\theta^{(1)}$ и $\theta^{(2)}$ первый столбец также нужно отбросить.

Если вы реализовали код функции **gradient_function** без ошибок, после запуска скрипта вы должны увидеть сообщение:

```
Точность нейронной сети на обучающей выборке: 98.8
```

Число может отличаться из-за случайной инициализации начальных параметров, но должно быть близко к **99%**. Итоговые коэффициенты, полученные при обучении нейронной сети, находятся в матрицах **Theta1** и **Theta2**. Нейронная сеть, распознающая рукописные цифры, успешно обучена!