МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

МЕТОДИ ГЛИБОКОГО НАВЧАННЯ НА РІЗНОРІДНИХ ДАНИХ

Лабораторний практикум

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня магістра
за освітньо-науковою програмою «Математичні методи моделювання,
розпізнавання образів та комп'ютерного зору»
спеціальності 113 «Прикладна математика»*

Київ
КПІ ім. Ігоря Сікорського
2022

Методи глибокого навчання на різнорідних даних. Лабораторний практикум [Електронний ресурс]: навч. посіб. для студ. спеціальності 113 «Прикладна математика» / А. Ю. Шелестов, Н. М. Куссуль, С. Ю. Дрозд, Г.О. Яйлимова; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 7 265 Кбайт). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 77 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 5 від 26.05.2022 р.) за поданням Вченої ради Навчально-наукового Фізико-технічного інституту Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (протокол № 4 від 18.04.2022 р.)

Електронне мережне навчальне видання

МЕТОДИ ГЛИБОКОГО НАВЧАННЯ НА РІЗНОРІДНИХ ДАНИХ

Лабораторний практикум

Автори: *Шелестов Андрій Юрійович*, д. техн. наук, проф.
Куссуль Наталія Миколаївна, д. техн. наук, проф.
Дрозд Софія Юріївна, студентка
Яйлимова Ганна Олексіївна, д-р філософії

Відповідальний редактор *Смирнов С.А.,* к.ф.-м.н., доц.

Рецензент *Лавренюк А.М.*, канд. техн. наук, доц. кафедри математичного моделювання та аналізу даних НН ФТІ Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського"

Навчальний посібник «Методи глибокого навчання на різнорідних даних. Лабораторний практикум» присвячено вивченню методів та алгоритмів глибинного навчання та аналізу гетерогенних геопросторових даних здобувачами вищої освіти за спеціальністю 113 «Прикладна математика» та може бути корисним для інших технічних спеціальностей. Метою навчальної дисципліни є формування у студентів здатностей засвоєння теоретичних положень, сучасних методів та алгоритмів аналізу гетерогенних і, зокрема, геопросторових даних, оволодіння практичними навичками використання сучасних підходів по використанню даних з різних джерел для розв’язання різних прикладних задач. Посібник містить необхідний теоретичний матеріал, приклади, а також завдання для виконання лабораторного практикуму.

ЗМІСТ

Лабораторна робота №1
Основи Numpy Python3

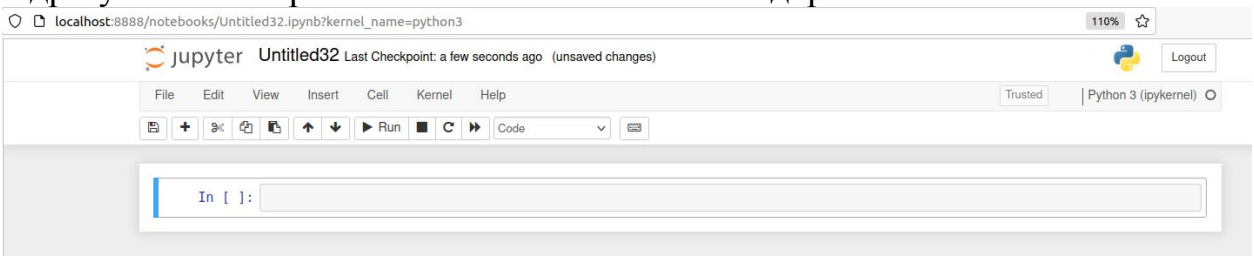
Мета роботи: отримати навички роботи з iPython Notebook та освоїти важливі функції бібліотеки Numpy

1.1. Теоретичні відомості

Знайомство з iPython Notebook

iPython Notebook — це інтерактивні середовища кодування, вбудовані у веб-сторінку. Для того, щоб розпочати роботу із iPython Notebook, необхідно установити на свою локальну машину та запустити *Jupyter Notebook*. Деталі установки Jupyter Notebook та запуску блокноту можна переглянути тут: https://www.8host.com/blog/ustanovka-jupyter-notebook-dlya-python-3/
Після запуску Jupyter Notebook необхідно створити файл блокноту:

Одразу після створення блокнот автоматично відкриється:



У вільній комірці можна писати програмний код Python

```
In [ ]:
### START CODE HERE ### (≈ 1 line of code)
test = None
### END CODE HERE ###
```

```
In [ ]:
print ("test: " + test)
```

Щоб запустити код у комірці, необхідно натиснути SHIFT+ENTER або кнопку на верхній панелі «Run».

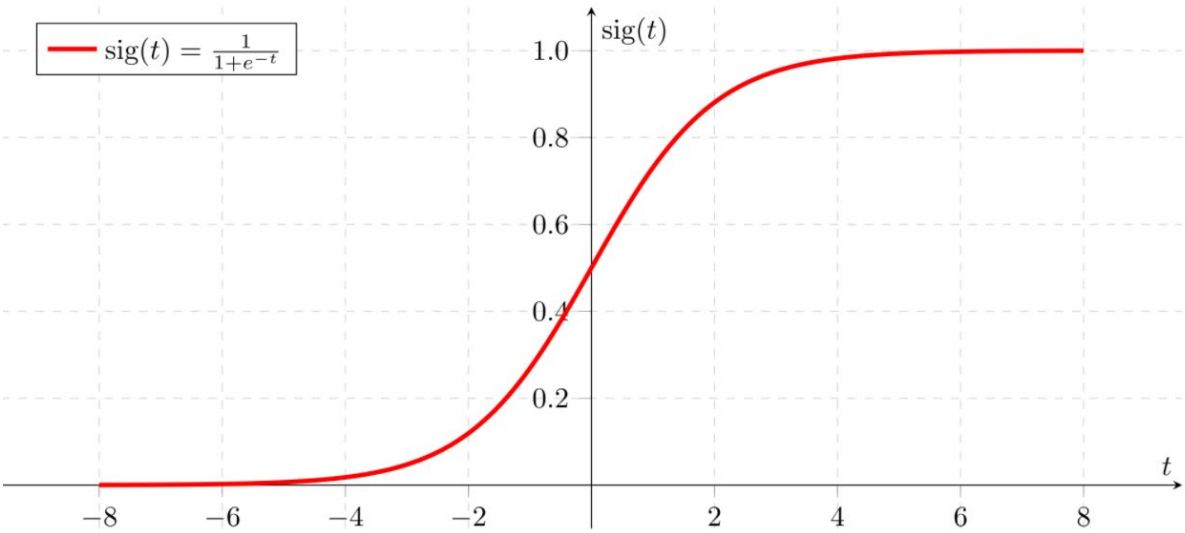
Для додавання нової комірки треба натиснути «+», а для видалення будь якої комірки використита «ножиці».
Щоб відмінити видалення комірки, треба натиснути “Ecs” – перехід у командний режим, а потім “Z”.
Комірки можна об’єднувати, міняти стилі шрифту, довати коментарі між комірками, тощо.

Побудова основних функцій за допомогою бібліотеки numpy

Numpy є основним пакетом для наукових обчислень на Python. Його підтримує велика спільнота (www.numpy.org). У цій лабораторній роботі ви дізнаєтеся кілька ключових функцій numpy, таких як np.exp(), np.log() і np.reshape(). Вам потрібно буде знати, як використовувати ці функції для майбутніх завдань.

Сигмовидна функція, np.exp()

Функція $sigmoid(x) = \frac{1}{1+e^{-x}}$ – *сигмовидна функція*, іноді також відома як логістична функція. Це нелінійна функція використовується не тільки в машинному навчанні (логістична регресія), але і в глибокому навчанні.



Цю функцію можна побудувати за допомогою біблотеки math.exp(). Проте у такому випадку у функцію не можна буде передати вектор значень. Тому для уникнення помилок, коли ми хочемо побудувати функцію не за одним числом, а для аргументу у вигляді вектору, наприклад, потрібно використовувати np.exp().
Розглянемо приклад побудови сигмовидної функції.

Приклад 1.1. Побудова сигмовидної функції за допомогою бібліотеки math

Перш ніж використовувати np.exp(), скористаємося функцією math.exp() для реалізації сигмовидної функції.
Побудуємо функцію, яка повертає сигмоїд дійсного числа x. Використаємо для цього math.exp(x) для експоненціальної функції.

```python
import math
def basic_sigmoid(x):
    """
    Аргументи:
    x – скаляр

    Повернення:
    s -- sigmoid(x)
    """
    s = (1 + math.exp(-x)
    return s
```

Насправді, ми рідко використовуємо «математичну» бібліотеку math в глибокому навчанні, оскільки вхідні дані для функцій цієї бібліотеки можуть бути лише реальними числами. Проте в глибокому навчанні ми переважно використовуємо матриці та вектори. Але коли ми захочемо передати у вище описану функцію з бібліотеки math вектор аргументів, то отримаємо помилку:

```
In [14]: ### One reason why we use "numpy" instead of "math" in Deep Learning ###
         x = [1, 2, 3]
         basic_sigmoid(x) # you will see this give an error when you run it, because x is a vector.

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_38048/2727964942.py in <module>
      1 ### One reason why we use "numpy" instead of "math" in Deep Learning ###
      2 x = [1, 2, 3]
----> 3 basic_sigmoid(x) # you will see this give an error when you run it, because x is a vector.

/tmp/ipykernel_38048/2564686151.py in basic_sigmoid(x)
     15
     16     ### START CODE HERE ### (≈ 1 line of code)
---> 17     s = 1 / (1 + math.exp(-x))
     18     ### END CODE HERE ###
     19

TypeError: bad operand type for unary -: 'list'
```

Бібліотека numpy дозволяє вирішити дану проблему. Тому використання numpy є необхідним та більш корисним.
При передачі вектора у функцію для обчислення сигмоїди за допомогою numpy, обчислення значення функції буде застосоване для кожної компоненти вектора-аргументу. Тобто на виході теж буде отриманий вектор зі значеннями функції відповідно до кожної компоненти вектора аргументу.

In fact, if $x = (x_1, x_2, \ldots, x_n)$ **is a row vector then** $np.exp(x)$ **will apply the exponential function to every element of x. The output will thus be:** $np.exp(x) = (e^{x_1}, e^{x_2}, \ldots, e^{x_n})$

Приклад 1.2. Побудова сигмовидної функції за допомогою бібліотеки numpy

```
import numpy as np
x = np.array([0., 1., 2.])
np.exp(x)
```

Вихід:

```
array([ 1. ,  2.71828183,  7.3890561 ])
```

Крім того, якщо x є вектором, то при застосуванні до нього операцій Python, такий як наприклад s =x+3 або s =1/x, у результаті на виході отримаємо вектор s такої ж розмірності, що і x.

```
In [16]: # example of vector operation
         x = np.array([1, 2, 3])
         print (x + 3)

         [4 5 6]
```

Примітка: Щоразу, коли потрібна додаткова інформація про якусь функцію numpy, рекомендуємо переглянути офіційну документацію.
Можна також створити нову клітинку в блокноті та написати документацію np.exp? (наприклад), щоб отримати швидкий доступ до документації.

Сигмовидний градієнт

Щоб оптимізувати функції втрат за допомогою зворотного поширення, потрібно буде обчислити градієнти. Давайте розглянемо етапи побудови коду для створення функції *сигмовидного градієнта*.
Формула така:

$$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Закодуємо цю функцію в два кроки:

1. Обчислення s як сигмоїду x за допомогою функції basic_sigmoid().
2. Обчислення $\sigma'(x)=s(1-s)$

Реалізація функції пропонується у якості вправи для самостійного виконання.

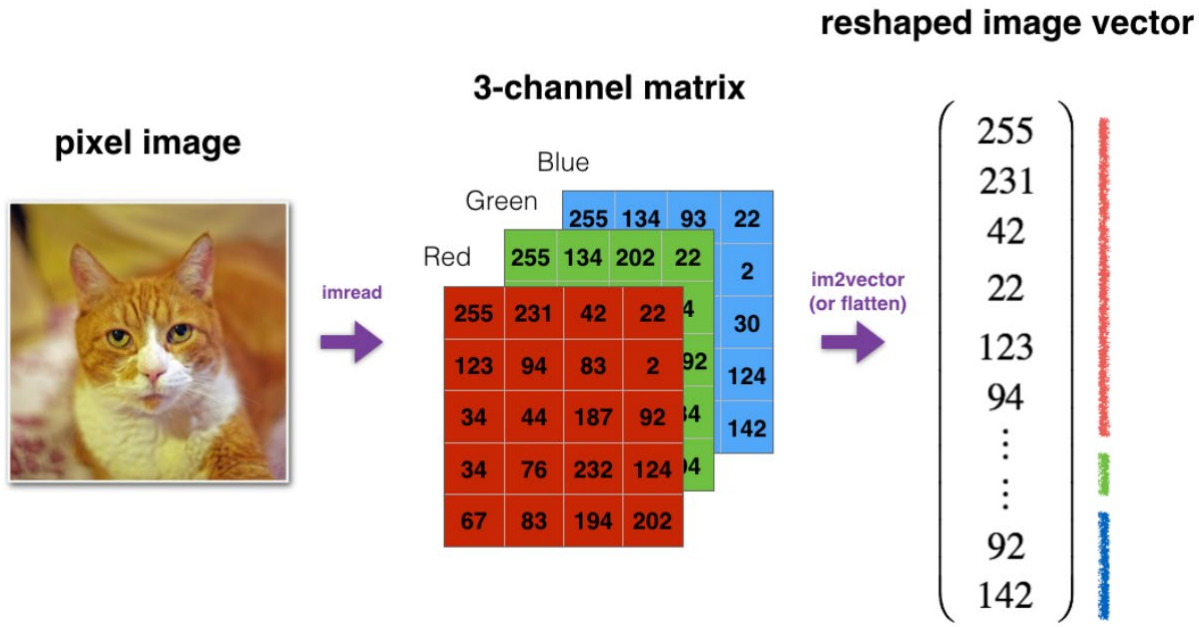
Зміна форми масивів numpy

Дві поширені функції numpy, які використовуються в глибокому навчанні - np.shape і np.reshape().

- X.shape використовується для отримання форми (розмірності) матриці/вектора X.
- X.reshape(...) використовується для зміни форми X в іншу розмірність.

Наприклад, в інформатиці зображення представляється тривимірним масивом фігур (довжина, висота, глибин = 3). Однак при зчитуванні зображення як

вхідних даних алгоритму, ми перетворюємо його у векторну форму (довжина∗висота∗ 3, 1). Іншими словами, відбувається «розгортання» або зміна форми 3D-масиву в одновимірний вектор.



Давайте виконаємо функцію image2vector(), яка приймає на вхід масив форми (довжина, висота, 3) і повертає вектор розмірності (довжина*висота*3, 1).
Щоб змінити розмірність масиву v форми (a, b, c) у вектор розмірності (a*b,c), потрібно зробити:

```
v = v.reshape((v.shape[0]*v.shape[1], v.shape[2])) # v.shape[0] = a ; v.shape[1] = b ; v.s
```

Не бажано кодувати розміри зображення як константу. Замість цього можна використовувати image.shape[0], що поверне значення довжини, image.shape[1], тощо.

Приклад 1.3. Зміна форми масиву numpy

```
def image2vector(image):
    """
    Аргументи:
    image -- масив numpy розмірністю (length, height, depth)
    Повернення:
    v - вектр розмірністю (length*height*depth, 1)
    """
    v = image.reshape(image.shape[0] * image.shape[1] * image.shape[2], 1);
    return v;
```

Вихід:

```
In [18]: image = np.array([[[ 0.67826139,  0.29380381],
                [ 0.90714982,  0.52835647],
                [ 0.4215251 ,  0.45017551]],

               [[ 0.92814219,  0.96677647],
                [ 0.85304703,  0.52351845],
                [ 0.19981397,  0.27417313]],

               [[ 0.60659855,  0.00533165],
                [ 0.10820313,  0.49978937],
                [ 0.34144279,  0.94630077]]])

         print ("image2vector(image) = " + str(image2vector(image)))

image2vector(image) = [[0.67826139 0.29380381]
 [0.90714982 0.52835647]
 [0.4215251  0.45017551]
 [0.92814219 0.96677647]
 [0.85304703 0.52351845]
 [0.19981397 0.27417313]
 [0.60659855 0.00533165]
 [0.10820313 0.49978937]
 [0.34144279 0.94630077]]
```

Нормалізація рядків

Ще одна поширена техніка, яку використовують в машинному навчанні та глибокому навчанні, — це нормалізація даних. Нормалізація потрібна для кращої продуктивності, оскільки градієнтний спуск сходиться швидше після нормалізації. Тут під нормалізацією мається на увазі зміну x на $\frac{x}{||x||}$ (ділення кожного вектора рядка x на його норму).

Приклад 1.4. Нормалізація даних

Нехай на вході матриця:

$$x = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \end{bmatrix}$$

Тоді:

$$\|x\| = np.linalg.norm(x, axis = 1, keepdims = True) = \begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix}$$

І:

$$x\_normalized = \frac{x}{\|x\|} = \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \end{bmatrix}$$

Зауважте, що ви можете розділити матриці різних розмірів, і це не викличе помилок: це називається *трансляцією*, і ви дізнаєтеся про це далі.

Трансляція та функція softmax()

Дуже важливою концепцією для розуміння в numpy є «*трансляція*». Вонв корисна для виконання математичних операцій між масивами різної форми. Повну інформацію про трансляцію можна прочитати в офіційній документації.
Реалізуємо функцію softmax() за допомогою numpy. Можна розглядати softmax() як про нормалізуючу функцію, яка використовується, коли алгоритму

потрібно класифікувати два або більше класів. Ви дізнаєтеся більше про softmax() згодом.

Приклад 1.5 Реалізація функції softmax()

```
def softmax(x):
  """
    Аргументи:
    x - масив розмірності (m,n)
    Повернення:
    s -- A numpy matrix equal to the softmax of x, of shape (m,n)
    """
    x_exp = np.exp(x)

  # Створення вектору x_sum що є сумою кожного рядка x_exp. Використаємо np.sum(..., axis = 1, keepdims = True).
    x_sum = np.sum(x_exp,axis = 1, keepdims = True)

  # Обчислення softmax(x) діленням x_exp на x_sum.
    s = x_exp / x_sum
  return s
```

In [71]:
```
x = np.array([
    [9, 2, 5, 0, 0],
    [7, 5, 0, 0 ,0]])
print("softmax(x) = " + str(softmax(x)))
```

Expected Output:

softmax(x)

[[9.80897665e-01 8.94462891e-04 1.79657674e-02 1.21052389e-04 1.21052389e-04] [8.78679856e-01 1.18916387e-01 8.01252314e-04 8.01252314e-04 8.01252314e-04]]

Примітка:
Якщо ви виведете розмірності x_exp, x_sum і s вище та повторно запустите комірку оцінки, ви побачите, що x_sum має форму (2,1), а x_exp та s мають форму (2,5). Ділення x_exp на x_sum не викликає помилок завдяки трансляції Python.

Векторизація

У глибокому навчанні часто мають справу з дуже великими наборами даних. Отже, неоптимальна для обчислень функція може стати вузьким місцем у алгоритмі і може призвести до створення моделі, опрацювати яку не вдасться і за 100 років. Щоб переконатися, що код є обчислювально ефективним, треба використати *векторизацію*. Наприклад, спробуємо визначити різницю між наступними реалізаціями крапкового/зовнішнього/елементного добутку.

Приклад 1.6 Векторизація

```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
```

```
tic = time.process_time()
dot = 0
for i in range(len(x1)):
    dot+= x1[i]*x2[i]
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1),len(x2))) # we create a len(x1)*len(x2) matrix with
only zeros
for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i]*x2[j]
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(1000*(toc
- tic)) + "ms")

### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))
for i in range(len(x1)):
    mul[i] = x1[i]*x2[i]
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation time =
" + str(1000*(toc - tic)) + "ms")

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Випадковий 3*len(x1) масив
tic = time.process_time()
gdot = np.zeros(W.shape[0])
for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j]*x1[j]
toc = time.process_time()
print ("gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")
```

```
x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc - tic))
+ "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation time = "
+ str(1000*(toc - tic)) + "ms")
```

```
### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")
```

Вихід:

```
dot = 278
 ----- Computation time = 0.17802000000011198ms
outer = [[ 81.  18.  18.  81.   0.  81.  18.  45.   0.   0.  81.  18.  45.   0.
    0.]
 [ 18.   4.   4.  18.   0.  18.   4.  10.   0.   0.  18.   4.  10.   0.
    0.]
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [ 63.  14.  14.  63.   0.  63.  14.  35.   0.   0.  63.  14.  35.   0.
    0.]
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [ 81.  18.  18.  81.   0.  81.  18.  45.   0.   0.  81.  18.  45.   0.
    0.]
 [ 18.   4.   4.  18.   0.  18.   4.  10.   0.   0.  18.   4.  10.   0.
    0.]
 [ 45.  10.  10.  45.   0.  45.  10.  25.   0.   0.  45.  10.  25.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
    0.]]
 ----- Computation time = 0.33455199999998797ms
elementwise multiplication = [ 81.   4.  10.   0.   0.  63.  10.   0.   0.   0.  81.   4.  25.   0.   0.]
 ----- Computation time = 0.19903899999995645ms
gdot = [ 17.79497022  19.82783769  19.47188495]
 ----- Computation time = 0.23477999999999ms
```

```
x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc - tic))
+ "ms")

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
```

```
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation time = "
+ str(1000*(toc - tic)) + "ms")

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot) + "\n ----- Computation time = " + str(1000*(toc -
tic)) + "ms")
```

Вихід:

```
dot = 278
 ----- Computation time = 0.16674600000010642ms
outer = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
 ----- Computation time = 0.15854599999998165ms
elementwise multiplication = [81  4 10  0  0 63 10  0  0  0 81  4 25  0  0]
 ----- Computation time = 0.12599999999984846ms
gdot = [ 17.79497022  19.82783769  19.47188495]
 ----- Computation time = 0.18404100000002366ms
```

Як ви могли помітити, векторизована реалізація набагато чистіша та ефективніша. Для більших векторів/матриць різниця в часі виконання стає ще більшою. Зауважте, що np.dot() виконує множення матриці-матриці або матриці-вектора. Це відрізняється від np.multiply() і оператора * (який еквівалентний .* у Matlab/Octave), які виконують поелементне множення.

Реалізація функції втрат L1 і L2

Втрата використовується для оцінки продуктивності моделі. Чим більша втрата, тим більш різняться прогнозовані значення (ȳ) з істинними значеннями

(y). У глибокому навчанні використовуються алгоритми оптимізації, такі як Gradient Descent, щоб навчити свою модель і мінімізувати витрати.

- Втрата L1 визначається як:

$$L_1(\hat{y}, y) = \sum_{i=0}^{m} |y^{(i)} - \hat{y}^{(i)}|$$

- Втрата L2 визначається як:

$$L_2(\hat{y}, y) = \sum_{i=0}^{m} (y^{(i)} - \hat{y}^{(i)})^2$$

1.2. Порядок виконання роботи

1.2.1. Проаналізувати умову задачі.
1.2.2. Розробити алгоритм та створити програму розв'язання задачі згідно з номером варіанту.
1.2.3. Результати роботи оформити протоколом.

1.3 Завдання для виконання

1. Реалізуйте сигмовидну функцію за допомогою numpy для аргументу x, що є дійсним числом, вектором або матрицею.
2. Реалізуйте функцію sigmoid_grad() для обчислення градієнта сигмоїдної функції відносно її входу x
3. Реалізуйте функцію normalizeRows(), щоб нормалізувати рядки матриці. Після застосування цієї функції до вхідної матриці x кожен рядок x має бути вектором одиничної довжини (мається на увазі довжина 1)
4. Реалізувати numpy векторизовану версію втрати L1. Варто скористатися функією abs(x) (пошук абсолютного значення x).
5. Реалізуйте numpy векторизовану версію втрати L2. Існує кілька способів реалізації втрати L2, але варто використати np.dot()

1.4. Контрольні запитання

1. Що таке блокноти ipython?
2. Що чого потрібно сигмовидна функція у машинному навчанні?
3. Яка основна бібліотека для математичних розрахунків та обробки даних використовується у машинному навчанні? Які її важливі функції ви знаєте?
4. Що таке трансляція?
5. Що таке векторизаія?
6. Що таке нормування даних і навіщо воно потрібне?
7. Що таке витрати? Яка різния між витратами L1 і L2?

Лабораторна робота №2
Логістична регресія із використанням мислення нейронної мережі

Мета роботи: познайомитися із поняттям логістичної регресії та отримати навички її побудови із використанням мислення нейронної мережі.

2.1. Теоретичні відомості

Поняття логістичної регресії

Логістична регресія — статистичний регресійний метод, що застосовують у випадку, коли залежна змінна є бінарною, тобто може набувати тільки двох значень (0 або 1). При запровадженні порогового значення може знаходити застосування у класифікації.

Прикладом може слугувати класифікація електронних листів на «спам» або «не спам». Метод також використовується у медицині, наприклад, для визначення чи є пухлина злоякісною, чи доброякісною.

Необхідні пакети

Для побудови моделі логістичної регресії спершу необхідно підключити всі необхідні пакети (бібліотеки).

- *numpy* є основним пакетом для наукових обчислень з Python.
- *h5py* — це звичайний пакет для взаємодії з набором даних, який зберігається у файлі H5.
- *matplotlib* - це відома бібліотека для побудови графіків на Python.
- *PIL* і *scipy* використовуються тут для перевірки моделі з вашим власним зображенням в кінці.

Приклад 2.1. Підключення необхідних бібліотек

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage

%matplotlib inline
```

Примітка: для встановлення відсутніх на локальній машині пакетів прямо через інтерфейс блокноту необхідно запустити команду: !pip install packet_name

Приклад 2.2. встановлення пакетів через інтерфейс Jupyter Notebook

```
In [2]: !pip install h5py

        /usr/lib/python3/dist-packages/secretstorage/dhcrypto.py:15: CryptographyDeprecationWarning: int_from_bytes is depre
        cated, use int.from_bytes instead
          from cryptography.utils import int_from_bytes
        /usr/lib/python3/dist-packages/secretstorage/util.py:19: CryptographyDeprecationWarning: int_from_bytes is deprecate
        d, use int.from_bytes instead
          from cryptography.utils import int_from_bytes
        Collecting h5py
          Downloading h5py-3.6.0-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (4.5 MB)
             |████████████████████████████████| 4.5 MB 4.4 MB/s eta 0:00:01     |████████████                    | 1.9 MB 4.
        4 MB/s eta 0:00:01
        Requirement already satisfied: numpy>=1.14.5 in /usr/lib/python3/dist-packages (from h5py) (1.17.4)
        Installing collected packages: h5py
        Successfully installed h5py-3.6.0
```

Огляд набору задач

Розглянемо набір даних ("data.h5"), що містить:

- навчальний набір зображень m_train, позначених як cat (y=1) або non-cat (y=0). Тобто зображення типу «кішки» та «не кішки»
- тестовий набір зображень m_test, позначених як cat або non-cat
- кожне зображення має форму (num_px, num_px, 3), де 3 для 3 каналів (RGB). Таким чином, кожне зображення є квадратним (висота = кількість_пікселей) і (ширина = кількість_пікселей).

Побудуємо простий алгоритм розпізнавання зображень, який зможе правильно класифікувати зображення як кішки чи не кішки. Давайте ближче познайомимося з набором даних. Завантажимо дані.

Приклад 2.3. Завантаження даних

```
def load_dataset():
    train_dataset = h5py.File('datasets/train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # майбутні
тренувальні дані
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) #
тренувальні мітки

    test_dataset = h5py.File('datasets/test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # майбутні
тестові дані
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # тестові
мітки

    classes = np.array(test_dataset["list_classes"][:]) # список класів

    train_set_y_orig = train_set_y_orig.reshape((1,
train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig,
test_set_y_orig, classes
```

```
# Завантаження даних (кіт/не кіт)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes =
load_dataset()
```

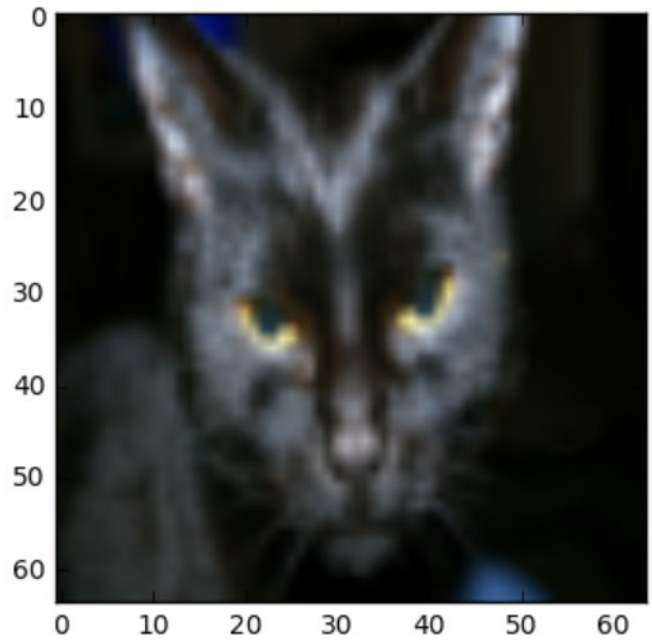
Ми додали «_orig» в кінці наборів даних зображень (навчання та тестування), тому що ми збираємося їх попередньо обробити. Після попередньої обробки ми отримаємо train_set_x і test_set_x (мітки train_set_y і test_set_y не

потребують попередньої обробки). Кожен рядок train_set_x_orig і test_set_x_orig є масивом, що представляє зображення. Запустимо наступний код, щоб переглянути зображення:

Приклад 2.4. Перегляд завантажених зображень

```
index = 25
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" +
classes[np.squeeze(train_set_y[:, index])].decode("utf-8") +  "' picture.")
```

```
y = [1], it's a 'cat' picture.
```



Щоб побачити інше зображення, потрібно змінити значення індексу.
Багато програмних помилок у глибокому навчанні виникають через невідповідність розмірів матриці/вектора. Щоб уникнути багатьох помилок, потрібно підтримувати розміри матриці/вектора прямими. Давайте визначимо розміри даних у нашому датасеті.

Приклад 2.5. Знаходження значень розмірностей масиву даних

```
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = test_set_x_orig.shape[1]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

Вихід:

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
```

```
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Для зручності тепер змінемо форму зображення (num_px, num_px, 3) у масив numpy (num_px∗ кількість_пікс∗3, 1). Після цього наш навчальний (і тестовий) набір даних являтиме собою масив numpy, де кожен стовпець представляє сплощене зображення. Повинні бути стовпці m_train (відповідно m_test).
Переформуємо набори навчальних і тестових даних, щоб зображення розміру (num_px, num_px, 3) були сплющені в окремі вектори розмірності (num_px∗кількість_пікс∗3, 1).

Приклад 2.6. Переформування даних

```
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -
1).T
test_set_x_flatten =  test_set_x_orig.reshape(test_set_x_orig.shape[0], -
1).T
print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

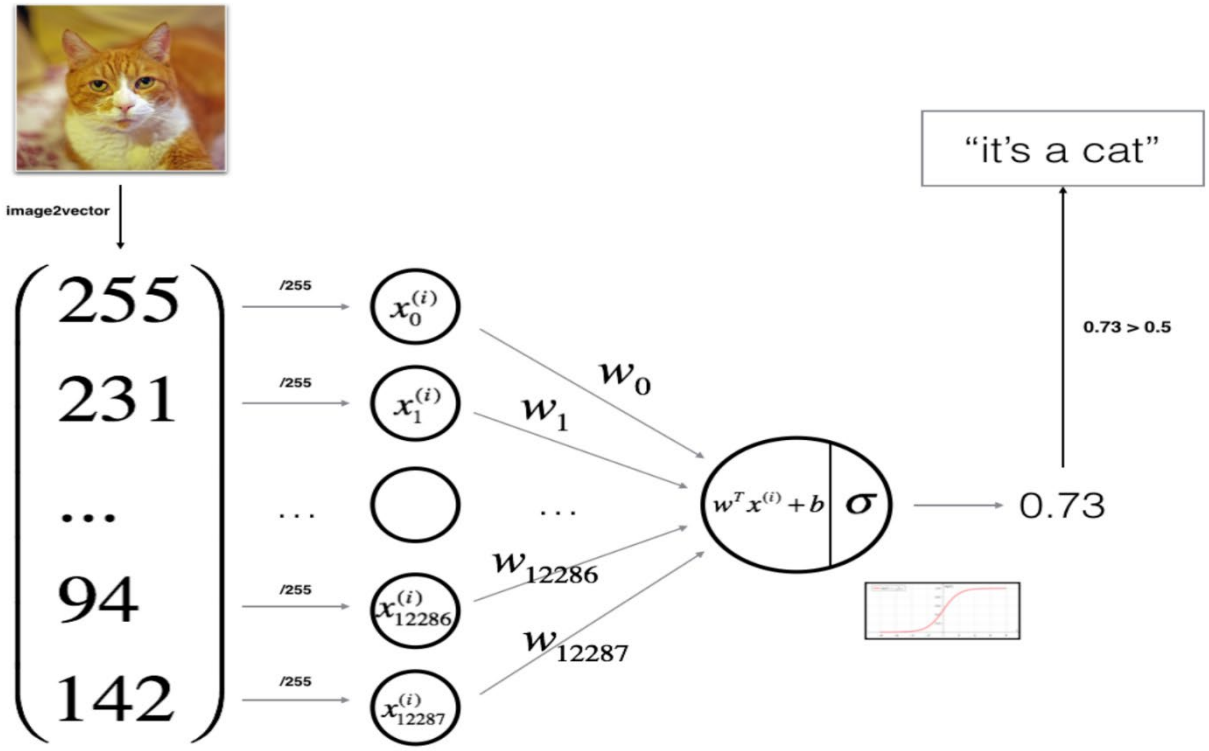
Для представлення кольорових зображень червоний, зелений і синій канали (RGB) мають бути вказані для кожного пікселя, тому значення пікселя насправді є вектором із трьох чисел у діапазоні від 0 до 255.
Одним із поширених етапів попередньої обробки в машинному навчанні є *центрування* та *стандартизація* набору даних, відмімання середнього значення всього масиву numpy з кожного прикладу, а потім ділення кожного прикладу на стандартне відхилення всього масиву numpy. Але для наборів даних зображень простіше і зручніше просто розділити кожен рядок набору даних на 255 (максимальне значення каналу пікселя).
Давайте стандартизуємо наш набір даних.

Приклад 2.7. Стандартизація даних

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

Загальна архітектура алгоритму навчання

Настав час розробити простий алгоритм, щоб відрізнити зображення котів від інших зображень. Побудуємо логістичну регресію, використовуючи мислення нейронної мережі. На наступному малюнку пояснюється, чому логістична регресія насправді є дуже простою нейронною мережею!



Математичний вираз алгоритму:
Для одного прикладу $x^{(i)}$:

$$z^{(i)} = w^T x^{(i)} + b$$
$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$
$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

Потім вартість обчислюється шляхом підсумовування всіх прикладів навчання:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

Побудова алгоритму

Наведемо основні кроки створення нейронної мережі:
1. Визначити структуру моделі (наприклад, кількість вхідних функцій)
2. Ініціалізувати параметри моделі
3. Розробити цикл:
   - Розрахувати поточні втрати (поширення вперед)
   - Розрахувати поточний градієнт (поширення назад)
   - Оновити параметри (градієнтний спуск)

Допоміжні функції

У якості допоміжної фунції необхідно реалізувати сигмовидну функцію, з якою ви познайомилися під час виконання лабораторної роботи 1. Для створення прогнозів необхідно обчислити сигмовидну функцію за формулою:

$$sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$$

Будемо використовувати np.exp().

Приклад 2.8. Обчислення сигновидної функції для прогнозування

```
def sigmoid(z):
    s = 1/(1 + np.exp(-z))
    return s
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

У якості параметра z тут може виступати дійсне число або масив довільної розмірності.

Параметри ініціалізації

Тепер реалізуємо параметри ініціалізації. Необхідно ініціалізувати вектор w як вектор нулів. Зробити це можна за допомою np.zeros(), про яку можна дізнатися більше в документації бібліотеки numpy.

Приклад 2.9. Ініціалізація масиву нулями

```
def initialize_with_zeros(dim):
    """
    Ця функція створює вектор нулів форми (dim, 1) для w і ініціалізує b до
    0
    Аргумент:
    dim - розмір потрібного вектора w (або кількість параметрів у цьому випадку)
    Повертає:
    w - ініціалізований вектор форми (dim, 1)
    b - ініціалізований скаляр (відповідає зміщенню)
    """
    w = np.zeros((dim, 1))
    b = 0
    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))
    return w, b
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

Вихід:

```
w = [[ 0.]
 [ 0.]]
b = 0
```

Пряме та зворотне поширення

Тепер, коли параметри ініціалізовано, можна виконати кроки «вперед» і «назад» для вивчення параметрів – *пряме та зворотне поширення*.

Створимо функцію для обчислення вартості функції та її градієнту propagate().

Алгоритм прямого поширення:

1. Отримуємо значення x
2. Обчислюємо:

$$A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$$

3. Обчислюємо вартість функції:

$$J = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$$

Для цього будемо використовувати дві формули:

$$\frac{\partial J}{\partial w} = \frac{1}{m}X(A-Y)^T$$
$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)})$$

Приклад 2.10. Обчислення вартості функції та її градієнту

```
# GRADED FUNCTION: propagate

def propagate(w, b, X, Y):
    """
    Реалізуйте функцію вартості та її градієнт для поширення, описаного вище
    Аргументи:
    w - ваги, числовий масив розміру (num_px * num_px * 3, 1)
    b - зміщення, скаляр
    X - дані розміру (кількість_px * кількість_px * 3, кількість прикладів)
    Y - справжній вектор "мітки" (містить 0, якщо не кіт, 1, якщо кіт) розміром
(1, кількість прикладів)
    Повернення функції:
    вартість - від'ємна вартість логарифмічної правдоподібності для логістичної
регресії
    dw - градієнт втрати по відношенню до w, таким чином така ж форма, що й w d
    b - градієнт втрати відносно b, такої ж розмірності, що й b """
#Пряме поширення
    m = X.shape[1]

    A = sigmoid(np.dot(w.T, X) + b)# Обчислення A
    cost = -1./m* np.sum(Y*np.log(A) + (1-Y)*np.log(1-A)) # Обчислення вартостей

    # Зворотнє поширення (знаходження градієнту)
    dw = 1./m*np.dot(X, (A-Y).T)
    db = 1./m*np.sum(A-Y)

    assert(dw.shape == w.shape)
    assert(db.dtype == float)

    grads = {"dw": dw,
             "db": db}

    return grads, cost
w, b, X, Y = np.array([[1],[2]]), 2, np.array([[1,2],[3,4]]), np.array([[1,0]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

Вихід:

```
dw = [[ 0.99993216]
 [ 1.99980262]]
db = 0.499935230625
cost = 6.00006477319
```

Оптимізація

Ми вже ініціалізували свої параметри та також можемо обчислити функцію вартості та її градієнт. Тепер можемо оновити параметри за допомогою градієнтного спуску.

Напишемо функцію оптимізації. Мета – навчити w і b шляхом мінімізації функції витрат функції J. Для параметра θ, правило оновлення таке $\theta = \theta - \alpha\, d\theta$, де α - це швидкість навчання.

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    Ця функція оптимізує w і b за допомогою алгоритму градієнтного спуску

    Аргументи:
    w -- вартості, масиви розмірності (num_px * num_px * 3, 1)
    b -- зміщення, скаляр
    X - дані розмірністю (num_px * num_px * 3, number of examples)
    Y - фактичні  "значення" вектора (містять 0 якщо картинка без кота, 1 якщо
картинка з котом), розмірністю (1, number of examples)
    num_iterations - число ітераій оптимізаційного циклу
    learning_rate -- швидкість навчання правила оновлення градієнтного спуску
    print_cost -- True для друку втрати кожні 100 ітерацій

    Повернення функції:
    params - словник, що містить ваги w і зміщення b
    grads - словник, що містить градієнти ваг і зміщень відносно функії вартості
    costs - список всіх ваг, обрахованих під час оптимізаії , що будуть
використані для побудови графіку навчальної кривої
    """

    costs = []

    for i in range(num_iterations):


        # Обрахування вартості на градієнту
        grads, cost =propagate(w, b, X, Y)

        # Отримаємо похідні grads
        dw = grads["dw"]
        db = grads["db"]

        # Прави оновлення
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Запишемо вартості
        if i % 100 == 0:
            costs.append(cost)

        # Виведемо вартості кожних 100 ітерацій тестових прикладів
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
params, grads, costs = optimize(w, b, X, Y, num_iterations= 100, learning_rate =
0.009, print_cost = False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

Вихід:

```
w = [[ 0.1124579 ]
 [ 0.23106775]]
b = 1.55930492484
dw = [[ 0.90158428]
 [ 1.76250842]]
db = 0.430462071679
```

Попередня функція виведе вивчені w і b. Ми можемо використовувати w і b для прогнозування міток для набору даних X. Реалізуємо функцію передбачення. Побудова функції виконується у два кроки:

1. Обчислення:

$$\hat{Y} = A = \sigma(w^T X + b)$$

2. Перетворення записів а в 0 (якщо активація <= 0,5) або в 1 (якщо активація > 0,5), зберігаючи передбачення у векторі Y_prediction . За бажання можна скористатися конструкцією if/else в циклі for.

Приклад 2.11 Побудова функції прогнозування

Спрогнозуємо значення мітки, 0 чи 1, використовуючи вивчені параметри логістичної регресії (w, b)

```
def predict(w, b, X):
    '''
    Аргумети:
    w -- вартості, масиви numpy розміром (num_px * num_px * 3, 1)
    b -- зміщення, скаляр
    X - дані, розмірністю (num_px * num_px * 3, кількість прикладів)

    Повернення з функції:
    Y_prediction - масив numpy (вектор), що містить всі передбачення (0/1)
для прикладів в X
    '''
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)
    # Обчислимо вектор "A", передбачаючи ймовірність того, що на малюнку
буде присутній кіт
    A = sigmoid(np.dot(w.T, X) + b)
    for i in range(A.shape[1]):
        # перетворимо ймовірності A[0,i] у фактичні передбачення p[0,i]
        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
             Y_prediction[0, i] = 0

    assert(Y_prediction.shape == (1, m))

    return Y_prediction
print ("predictions = " + str(predict(w, b, X)))
```

Вихід:

```
predictions = [[ 1.  1.]]
```

Об'єднання всіх функцій в модель

Побудуємо структуровану загальну модель, зібравши всі будівельні блоки (функції, реалізовані в попередніх частинах) разом у правильному порядку.
Реалізуємо функцію моделі.
Використаємо такі позначення:
- ***Y_prediction_test*** для прогнозів на тестовому наборі
- ***Y_prediction_train*** для прогнозів на тестових даних
- ***w, cost, grads*** для результатів функії optimize().

Приклад 2.12 Побудова функції моделі

```
def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
learning_rate = 0.5, print_cost = False)

    # ініціалізуємо параметри нулями
    w, b = initialize_with_zeros(X_train.shape[0])

    # градієнтний спуск
    parameters, grads, costs = optimize(w, b, X_train, Y_train,
num_iterations, learning_rate, print_cost = print_cost)

    # Отримаємо параметри w і b зі словника "parameters"
    w = parameters["w"]
    b = parameters["b"]
```

```
    # Передбачення для навчального/тестового прикладів наборів
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)

    # Друк помилок для тренувального та тестового наборів
    print("train accuracy: {} %".format(100 -
np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    print("test accuracy: {} %".format(100 -
np.mean(np.abs(Y_prediction_test - Y_test)) * 100))


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

Запустимо наступну клітинку, щоб навчити свою модель.

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations =
2000, learning_rate = 0.005, print_cost = True)
```

На виході отримаємо:

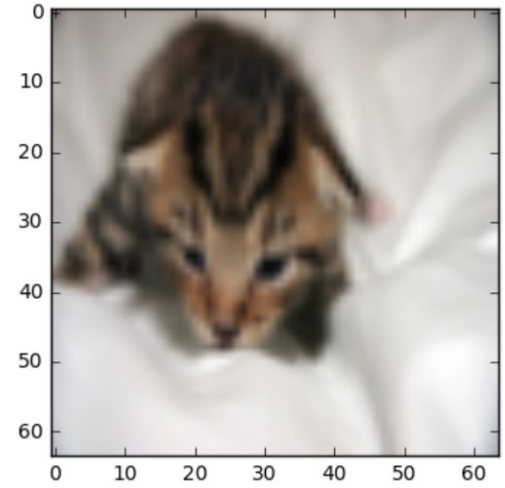
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214826
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
train accuracy: 99.04306220095694 %
test accuracy: 70.0 %
```

Точність навчання наближається до 100%. Це гарна перевірка працездатності: модель працює і має достатньо високу потужність, щоб відповідати навчальним даним.
Точність тесту 68%. Насправді це непогано для цієї простої моделі, враховуючи невеликий набір даних, який ми використовували, і що логістична регресія є лінійним класифікатором.
Модель явно переповнює навчальні дані. Пізніше в цій ви дізнаєтеся, як зменшити перенавчення, наприклад, за допомогою регуляризації.
Використовуючи наведений нижче код (і змінюючи індекс змінних) можна подивитися прогнози на зображеннях тестового набору.

Приклад 2.13 Прогнозування на тестовому наборі даних

```
# Приклад картинок, що класифікуються невірно.
index = 1
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a \""
+ classes[d["Y_prediction_test"][0,index]].decode("utf-8") +  "\" picture.")
```

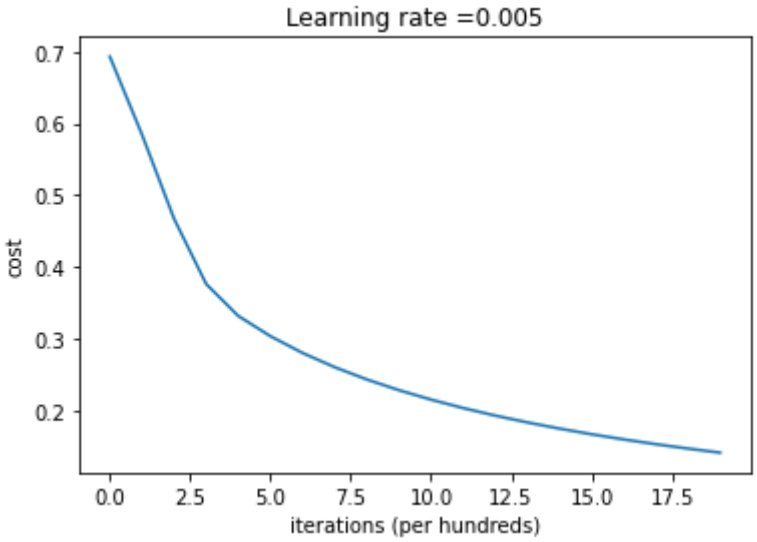
```
y = 1, you predicted that it is a "cat" picture.
```



Давайте також побудуємо графік функції вартості та градієнтів

Приклад 2.14 Побудова графіків функції вартості на градієнтів

```
# Графік навчальної кривої (з вартістю)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



Видно, вартість зменшується. Це показує, що параметри навчаються. Однак видно, що модель могла б бути більш тренованою на навчальному наборі. Спробуємо збільшити кількість ітерацій у клітинці вище та повторно запустити клітинку. Ви можете побачити, що точність навчального набору зростає, але точність тестового набору знижується. Це називається ***перенавчанням***.

2.2. Порядок виконання роботи

2.2.1. Проаналізувати умову задачі.

2.2.2. Підготувати власний набір зображень для класифікації згідно з варіантом (не менше 30 зображень)

1.2.3. Результати роботи оформити протоколом.

2.3 Завдання для виконання

Знайти набір даних (або підготувати власноруч) за темою, зазначеному у вашому варіанті. Побудувати модель для класифікації та класифікувати зображення на цьому наборі даних. Кількість картинок у наборі - не менше 30. Варіанти тем для пошуку наборів даних:

1. Птахи
2. Квіти
3. Собаки
4. Дерева
5. Будівлі
6. Автомобілі
7. Літаки
8. Кораблі
9. Ліжка

10. Вікна
11. Річки
12. Дороги

2.4. Контрольні запитання

1. Що таке логістична регресія і для чого вна потрібна?
2. Які пакети і навіщо потрібно імпортувати для побудови моделі логістичної регресії?
3. Як знайти розмірність даних і навізо це робити?
4. Що таке стандартизація? Що буд, якщо дані не стандартизувати?
5. Які допоміжні функціх треба реалізувати для побудови алгоритму?
6. Зо таке пряме та зворотне поширення?
7. Що таке градієнтний спуск?
8. Чому точність на навчальних даних більша ніж на тестових?
9. Як виявити перенавчання і що це взагалі таке?

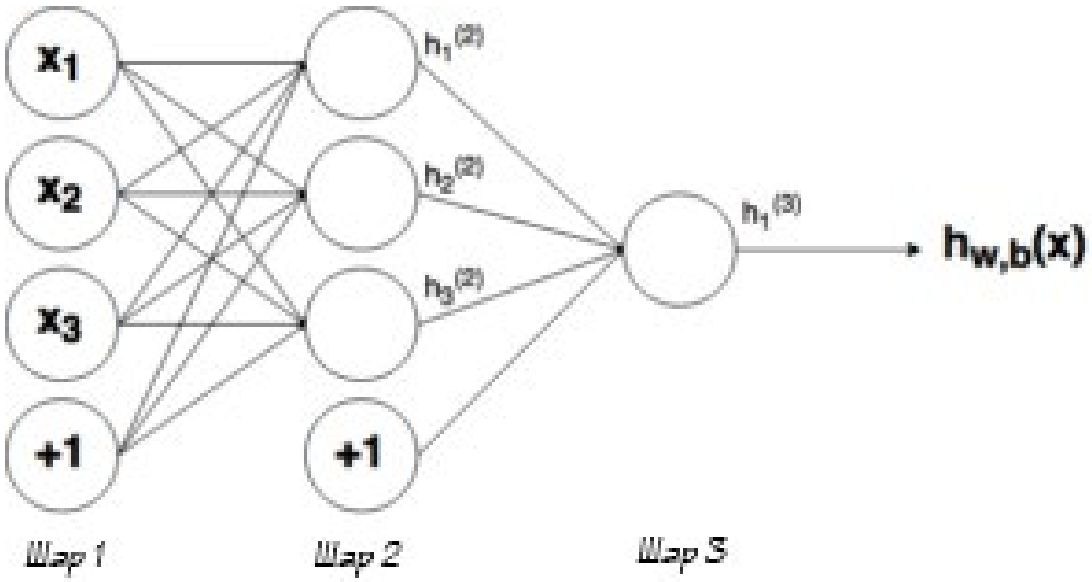
Лабораторна робота №3
Планарна класифікація даних з одним прихованим шаром

Мета роботи: побудувати нейронну мережу з прихованим шаром

3.1 Теоретичні відомості

Поняття нейронної мережі з прихованим шаром

У повній нейронній мережі знаходиться багато взаємозв'язаних між собою вузлів. Структури таких мереж можуть приймати міріади різних форм, але найпоширеніша складається з ***вхідного шару, прихованого шару*** та ***вихідного шару***. Приклад такої структури приведено нижче:



На рисунку вище можна побачити три шари мережі.

Шар 1 є ***вхідним шаром***, де мережа приймає зовнішні вхідні дані. Вхідний шар служить для розподілу даних по мережі і не робить ніяких обчислень. Виходи цього шару передають сигнали на входи наступного шару (прихованого або вихідного);

Шар 2 називають ***прихованим шаром***, цей шар не є частиною ні входу, ні виходу. Нейронні мережі можуть мати декілька прихованих шарів, і тоді вони називаються глибинними нейронними мережами. Приховані шари - шари звичайних нейронів, які передають сигнали від входу до виходу. Їх входом служить вихід попереднього шару, а вихід - входом наступного шару.

У даному прикладі було включено лише один шар для простоти.

Шар 3 є ***вихідним шаром***. Зазвичай він містить один нейрон, який видає результат розрахунків всієї нейронної мережі;

Необхідні пакети

Давайте спочатку імпортуємо всі пакети, які знадобляться під час виконання побудови нейронної мережі з прихованим шаром.

- ***numpy*** є основним пакетом для наукових обчислень з Python.
- ***sklearn*** надає прості та ефективні інструменти для інтелекту та аналізу даних.
- ***matplotlib*** — бібліотека для побудови графіків на Python.
- ***testCases*** надає кілька тестових прикладів для оцінки правильності функцій
- ***planar_utils*** надає різні корисні функції, які знадобляться при виконанні цього завдання

Приклад 3.1. Імпорт пакетів

```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
from testCases import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid,
load_planar_dataset, load_extra_datasets

%matplotlib inline

np.random.seed(1) # установіть початкове значення, щоб результати були
послідовними
```

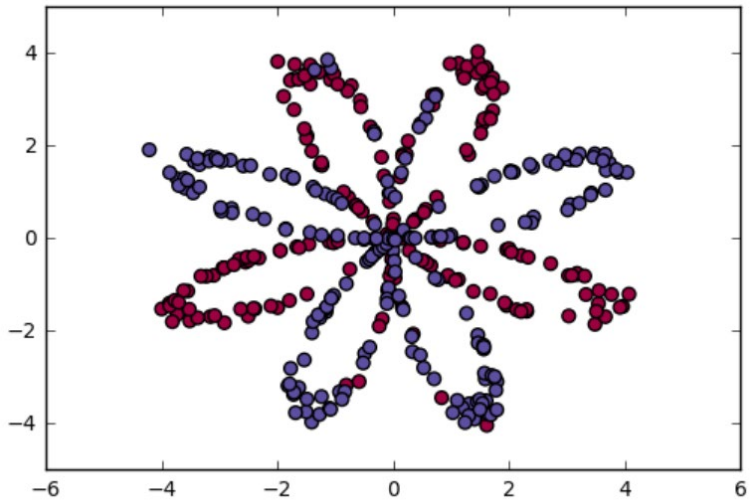
Завантаження даних

Спочатку давайте отримаємо набір даних, з яким будемо працювати. Завантажимо "квітковий" набір даних двох класів у змінні.

Далі візуалізуємо набір даних за допомогою matplotlib. Дані виглядають як «квітка» з деякими червоними (мітка y=0) і деякими синіми (y=1) точками. Наша мета — побудувати модель, яка відповідатиме цим даним. Іншими словами, ми хочемо, щоб класифікатор визначав області як червоні, так і як сині.

Приклад 3.2. Завантаження та візуалізація даних

```
X, Y = load_planar_dataset()
# візуалізація даних:
plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



Ми маємо:
- numpy-масив (матрицю) X, який містить функції (x1, x2)
- numpy-масив (вектор) Y, який містить мітки (червоний:0, синій:1).

Проведемо дослідження завантаженого датасету. Визгначимо форму масивів.

Приклад 3.3. Визначення форми масивів

```
shape_X = X.shape
shape_Y = Y.shape
m = X.shape[1]  # розмір тренувального набору
print ('The shape of X is: ' + str(shape_X))
print ('The shape of Y is: ' + str(shape_Y))
print ('I have m = %d training examples!' % (m))
```

Вихід:

```
The shape of X is: (2, 400)
The shape of Y is: (1, 400)
I have m = 400 training examples!
```

Проста логістична регресія

Перш ніж будувати повноцінну нейронну мережу, давайте спочатку подивимося, як логістична регресія діє на цю проблему. Для цього можна використовувати вбудовані функції sklearn. Навчимо класифікатор логістичної регресії для набору даних.

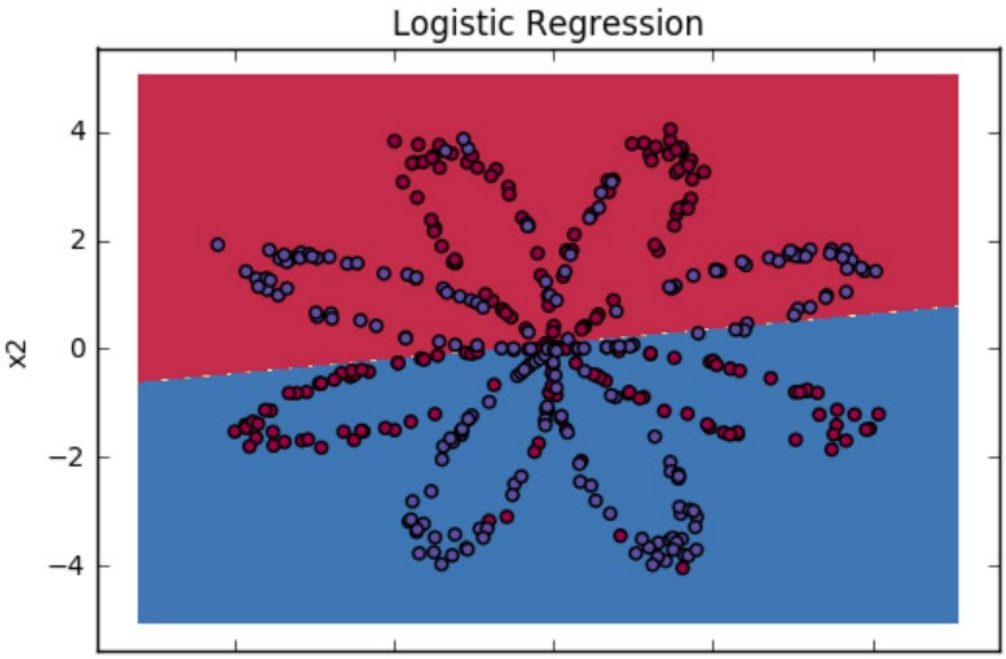
Приклад 3.4. Навчання класифікатора логістичної регресії

```
# Навчання класифікатора логістичної регресії
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);

# Побудуємо межу рішення для логістичної регресії
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
```

```
plt.title("Logistic Regression")

# Визначимо точність
LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' %
float((np.dot(Y,LR_predictions) + np.dot(1-Y,1-
LR_predictions))/float(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

Вихід:

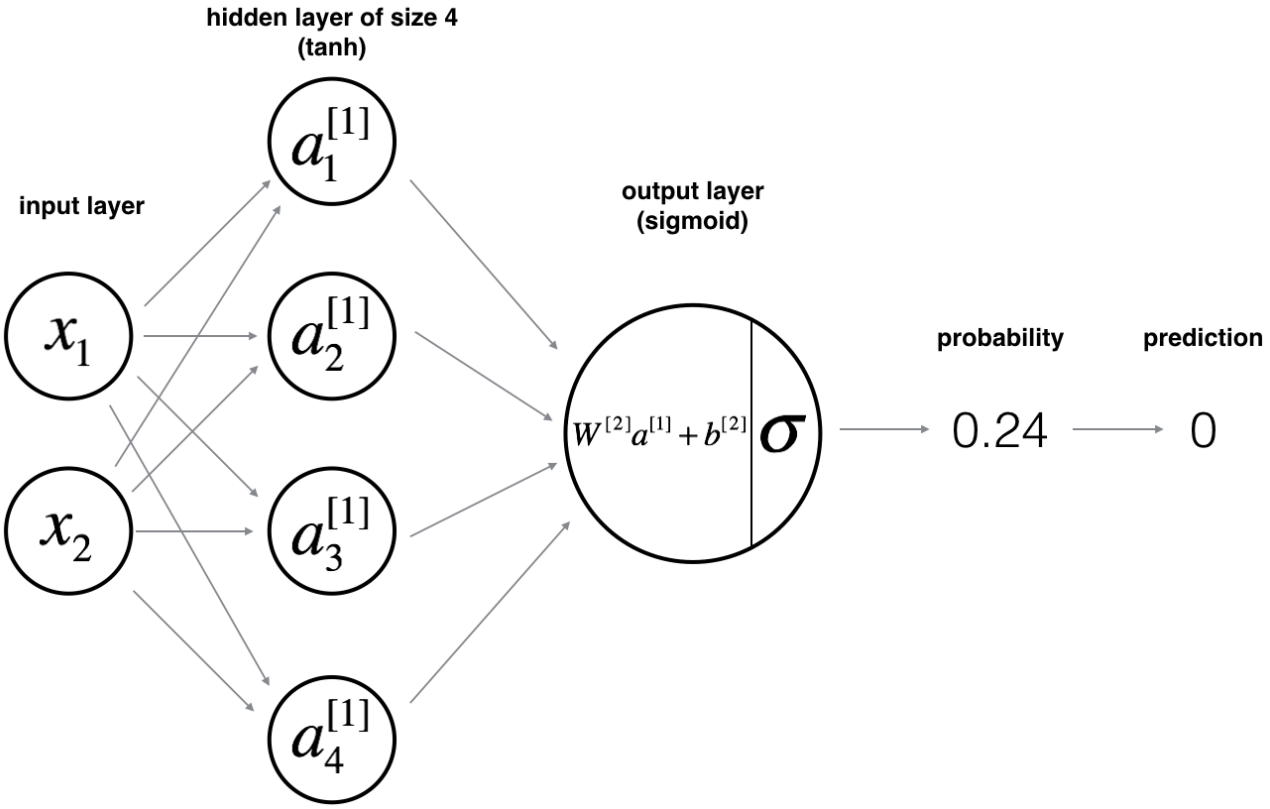
```
Accuracy of logistic regression: 47 % (percentage of correctly labelled datapoints)
```



Як видно, набір даних не є лінійно розділеним, тому логістична регресія не працює належним чином. Проте нейронна мережа має праювати краще.

Модель нейронної мережі

Логістична регресія погано працювала на «наборі даних квітів». Ми збираємося тренувати нейронну мережу з одним прихованим шаром. Ось наша модель:



Математично:
Для одного прикладу x(i):

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1](i)}$$
$$a^{[1](i)} = \tanh(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2](i)}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Враховуючи прогнози на всіх прикладах, можемо обчислити вартість J наступним чином:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right)$$

Загальна методологія побудови нейронної мережі полягає в тому, щоб:

1. Визначити структуру нейронної мережі (кількість вхідних одиниць, кількість прихованих блоків тощо) .
2. Ініціалізувати параметри моделі
3. Розробити цикл:
    - Здійснити пряме поширення

- Обчислення втрати
- Реалізувати зворотне поширення, щоб отримати градієнти
- Оновити параметри (градієнтний спуск)

Визначення структури нейронної мережі

Спершу визначимо такі розмірності:

- - n_x: розмір вхідного шару
- - n_h: розмір прихованого шару (встановіть значення 4)
- - n_y: розмір вихідного шару

Приклад 3.5. Визначення структури нейронної мережі

```
def layer_sizes(X, Y):
    """
    Аргументи:
    X – вхідні дані розмірності (вхідний розмір, число прикладів)
    Y – мітки розмірності (вихідний розмір, число прикладів)

    Повернення:
    n_x – розмір вхідного шару
    n_h -- розмір прихованого шару
    n_y -- розмір вихідного шару

    """
    n_x = X.shape[0]
    n_h = 4
    n_y = Y.shape[0]
    return (n_x, n_h, n_y)
```

```
X_assess, Y_assess = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(X_assess, Y_assess)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the hidden layer is: n_h = " + str(n_h))
print("The size of the output layer is: n_y = " + str(n_y))
```

Вихід:

```
The size of the input layer is: n_x = 5
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 2
```

Ініціалізація параметрів моделі

Виконаємо функцію ініціалізації: initialize_parameters()

Приклад 3.6. Визначення структури нейронної мережі

```
def initialize_parameters(n_x, n_h, n_y):

    """
    Аргументи:
    n_x – розмір вхідного шару
    n_h -- розмір прихованого шару
    n_y -- розмір вихідного шару

    Повернення:
    params – словник, що містить такі параметри:
                    W1 – ваги матриі розмірності (n_h, n_x)
```

```
                    b1 - зміщення вектора розмірності (n_h, 1)
                    W2 - ваги матриі розмірності (n_y, n_h)
                    b2 -- зміщення вектора розмірності (n_y, 1)
    """

    np.random.seed(2) # ми налаштовуємо початкове значення, щоб ваш
результат відповідав нашим, хоча ініціалізація є випадковою.
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y,1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
n_x, n_h, n_y = initialize_parameters_test_case()

parameters = initialize_parameters(n_x, n_h, n_y)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Вихід:

```
W1 = [[-0.00416758 -0.00056267]
 [-0.02136196  0.01640271]
 [-0.01793436 -0.00841747]
 [ 0.00502881 -0.01245288]]
b1 = [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
W2 = [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 = [[ 0.]]
```

Цикл

Реалізуємо функцію forward_propagation().

Приклад 3.7. Реалізаія циклу функією forward_propagation().

```
def forward_propagation(X, parameters):
    """
    Аргументи:
    X - вхідні дані розмірності (n_x, m)
    parameters - рсловник, що містить параметри (вихідні дані з функції
ініціалізаії)

    Повернення:
    A2 - сигмовидних вихід другої активації
    cache - словник, що містить "Z1", "A1", "Z2" та "A2"
```

```
    """
    # Отримаємо параметри зі словника «параметриЄ»
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Реалізуємо пряме поширення для обчислення A2 (ймовірності)

    Z1 = np.dot(W1, X) + b1
    A1 =  np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
X_assess, parameters = forward_propagation_test_case()

A2, cache = forward_propagation(X_assess, parameters)

# Примітка: ми використовуємо середнє значення тут, щоб переконатися, що ваш результат відповідає нашим

print(np.mean(cache['Z1'])
,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cache['A2']))
```

Вихід

```
-0.000499755777742 -0.000496963353232 0.000438187450959 0.500109546852
```

Тепер, коли ми порахували $A^{[2]}$ (A2), що містить $a^{[2](i)}$ для кожного прикладу, можна обчислити вартості функції за наступною формулою:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right)$$

Виконаємо функцію compute_cost() , щоб обчислити вартість J.
Існує багато способів реалізувати втрату перехресної ентропії. Рекомендуємо робити це за такою формулою:

$$-\sum_{i=0}^{m} y^{(i)}\log(a^{[2](i)}):$$

Приклад 3.8. Обчислення вартості

```
def compute_cost(A2, Y, parameters):
    """
    Обчислення перехресної ентропійної вартості, наведеної в рівнянні
```

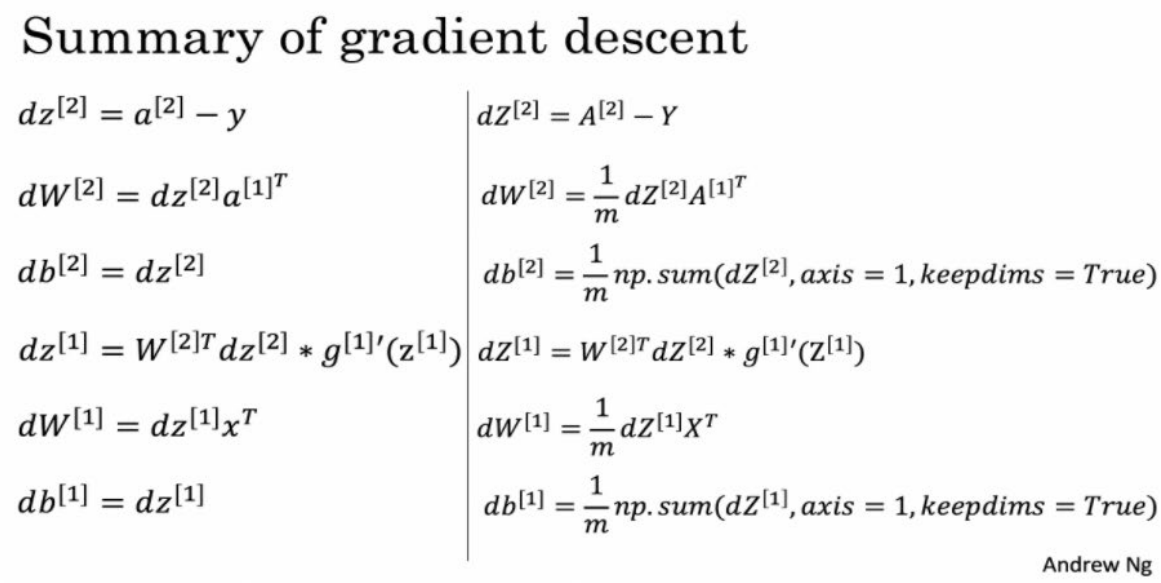
```
    Аргументи:
    A2 -- Сигмовидний вихід другої активації, рохмірності (1, number of
examples)
    Y -- "правдиві" мітки вектора, розмірності (1, number of examples)
    parameters - словник, що містить параметри W1, b1, W2 and b2

    Повернення:
    cost - перехресна ентропійна вартість за даними рівняння
    """

    m = Y.shape[1] # число прикладів

    # Обчислення перехресної ентропійної вартості

    logprobs = np.multiply(np.log(A2),Y) +  np.multiply(np.log(1-A2), (1-Y))
    cost = -1/m*np.sum(logprobs)

    cost = np.squeeze(cost)      # гарантує, що вартість відповідає
розмірності, яку ми очікуємо. Наприклад., перетворює [[17]] на 17
    assert(isinstance(cost, float))

    return cost
A2, Y_assess, parameters = compute_cost_test_case()

print("cost = " + str(compute_cost(A2, Y_assess, parameters)))
```

Вихід:

```
cost = 0.692919893776
```

Використовуючи кеш, обчислений під час прямого поширення, тепер можна реалізувати зворотне поширення.

Зворотне поширення зазвичай є найскладнішою (найбільш математичною) частиною глибокого навчання. Нижче слайд про зворотне поширення. Потрібно використовувати шість рівнянь праворуч на цьому слайді, щоб створити векторизовану реалізацію.



Приклад 3.9. Реалізація зворотного поширення

```
def backward_propagation(parameters, cache, X, Y):
    """
```

```
    Аргументи:
    parameters - словник з параметрами
    cache -- а словник, що містить "Z1", "A1", "Z2" and "A2".
    X - вхідні дані, розмірності (2, number of examples)
    Y -- "правдиві" мітки вектора, розірності (1, number of examples)

    Повернення:
    grads - словник, що містить градієнти з врахуванням різних параметрів
    """
    m = X.shape[1]

    # Спершу, визначимо W1 і W2 зі словника "parameters".
    W1 = parameters["W1"]
    W2 = parameters["W2"]

    # Визначимо також  A1 та A2 зі словника "cache".
    ### START CODE HERE ### (≈ 2 lines of code)
    A1 = cache["A1"]
    A2 =  cache["A2"]

    # Зворотнє поширення: вирахуємо dW1, db1, dW2, db2.

    dZ2= A2-Y
    dW2 = 1./m*np.dot(dZ2, A1.T)
    db2 = 1./m*np.sum(dZ2, axis = 1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = 1./m* np.dot(dZ1, X.T)
    db1 = 1./m*np.sum(dZ1, axis = 1, keepdims=True)


    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
parameters, cache, X_assess, Y_assess = backward_propagation_test_case()

grads = backward_propagation(parameters, cache, X_assess, Y_assess)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))
```

Вихід:

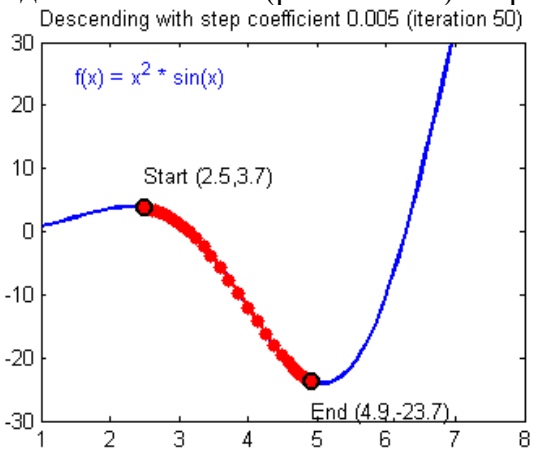
```
dW1 = [[ 0.01018708 -0.00708701]
 [ 0.00873447 -0.0060768 ]
 [-0.00530847  0.00369379]
 [-0.02206365  0.01535126]]
db1 = [[-0.00069728]
 [-0.00060606]
 [ 0.000364  ]
 [ 0.00151207]]
dW2 = [[ 0.00363613  0.03153604  0.01162914 -0.01318316]]
db2 = [[ 0.06589489]]
```

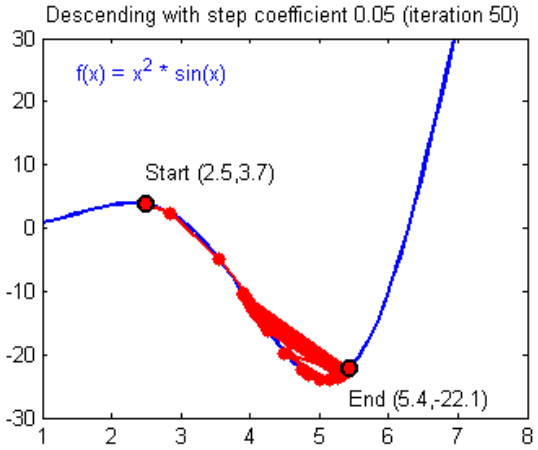
Застосуємо правило оновлення, використовуючи градієнтний спуск. Треба використовувати (dW1, db1, dW2, db2), щоб оновити (W1, b1, W2, b2). Загальне правило градієнтного спуску:

$$\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$$

де α - швидкість навчання і θ представляє параметр.

Алгоритми градієнтного спуску з хорошою швидкістю навчання (збіжність) і поганою швидкістю навчання (розбіжність) зображено нижче:





Приклад 3.10. Оновлення параметрів

```
def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Аргументи:
    parameters -- словник python, що містить параметри
    grads - словник python, що містить градієнти
    Повернення:
    parameters -- словник python, що містить оновлені параметри
    """
    # Визначимо кожен параметр зі словника "parameters"
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]

    # Визначимо кожен градієнт зі словника "grads"
    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    ## END CODE HERE ###

    # Оновило правила для кожного параметра

    W1 = W1 - dW1 * learning_rate
    b1 = b1 - db1 * learning_rate
    W2 = W2 - dW2 * learning_rate
    b2 = b2 - db2 * learning_rate

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}
```

```
    return parameters

parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Вихід:

```
W1 = [[-0.00643025  0.01936718]
 [-0.02410458  0.03978052]
 [-0.01653973 -0.02096177]
 [ 0.01046864 -0.05990141]]
b1 = [[ -1.02420756e-06]
 [  1.27373948e-05]
 [  8.32996807e-07]
 [ -3.20136836e-06]]
W2 = [[-0.01041081 -0.04463285  0.01758031  0.04747113]]
b2 = [[ 0.00010457]]
```

Об'єднання всіх функцій в модель

Тепер побудуємо свою модель нейронної мережі, яка використовує попередньонаписані функції в правильному порядку.

Приклад 3.11 Побудова моделі

```
def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
    """
    Аргументи:
    X - дані розмірності (2, number of examples)
    Y - мітки розмірності  (1, number of examples)
    n_h - розмір прихованого шару
    num_iterations - число ітераій в у циклі градієнтного спуску
    print_cost -- якщо True, друкувати вартість кожні 1000 ітерацій

    Повернення:
    parameters - параметри навченої моделі. Вони будуть використані для
передбачення пізніше.
    """

    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Ініціалізація параметрів, визначення W1, b1, W2, b2.
    Вхід: "n_x, n_h, n_y". Вихід = "W1, b1, W2, b2, parameters".
    parameters = initialize_parameters(n_x, n_h, n_y)
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
```

```
    # Цикл (градієнтний спуск)

    for i in range(0, num_iterations):


        # Поширення вперед. Входи: "X, parameters". Виходи: "A2, cache".
        A2, cache = forward_propagation(X, parameters)

        # Вартість функції. Вхлди: "A2, Y, parameters". Виходи: "cost".
        cost = compute_cost(A2, Y, parameters)

        # Зворотне поширення. Входи: "parameters, cache, X, Y". Виходи:
"grads".
        grads = backward_propagation(parameters, cache, X, Y)

        # Оновлення параметрів градієнтного спуску. вВходи: "parameters,
grads". Виходи: "parameters".
        parameters = update_parameters(parameters, grads)


        # Друк вартості кожні 1000 iteratioітерацій
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

```
_assess, Y_assess = nn_model_test_case()

parameters = nn_model(X_assess, Y_assess, 4, num_iterations=10000,
print_cost=False)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Вихід:

```
W1 = [[-4.23205691  5.31412598]
 [-7.53932104  1.20703336]
 [ 7.53821626 -1.20763364]
 [ 4.32939817 -5.27194484]]
b1 = [[ 2.30539936]
 [ 3.80834843]
 [-3.8102415 ]
 [-2.2821482 ]]
W2 = [[-6033.59327282 -6010.64275336  6009.92300745  6038.2072074 ]]
b2 = [[-53.22403038]]
```

Прогнозування

Тепер використаємо свою модель для прогнозування шляхом створення функції predict(). Використаємо пряме поширення, щоб передбачити результати. Згадаємо визначення прогнозування:

predictions = $y_{prediction} = 1\{\text{activation} > 0.5\} = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$

Щоб встановити для записів матриці X значення 0 і 1 на основі порогового значення, треба зробити:

```
X_new = (X > threshold)
```

Приклад 3.12. Прогнозування

```
def predict(parameters, X):
    """
    Використовуючи вивчені параметри, передбачимо клас для кожного прикладу
в X

    Аргументи:
    parameters - словник, що містить параметри
    X - вхідні дані, розмірністю (n_x, m)

    Повернення
    predictions - вектор передбачень нашої моделі (червоний: 0 / синій: 1)
    """

    # Обчислимо ймовірності використовуюси пряме поширення, і класифікуємо
як 0/1 використовуючи 0.5 як поріг.

    A2, cache = forward_propagation(X, parameters)
    predictions = A2 > 0.5


    return predictions
parameters, X_assess = predict_test_case()

predictions = predict(parameters, X_assess)
print("predictions mean = " + str(np.mean(predictions)))
```

Вихід:

```
predictions mean = 0.666666666667
```

Настав час запустити модель і подивитися, як вона працює на планарному наборі даних. Виконаємо наступний код, щоб перевірити свою модель з одним прихованим шаром з n_h прихованих одиниць.

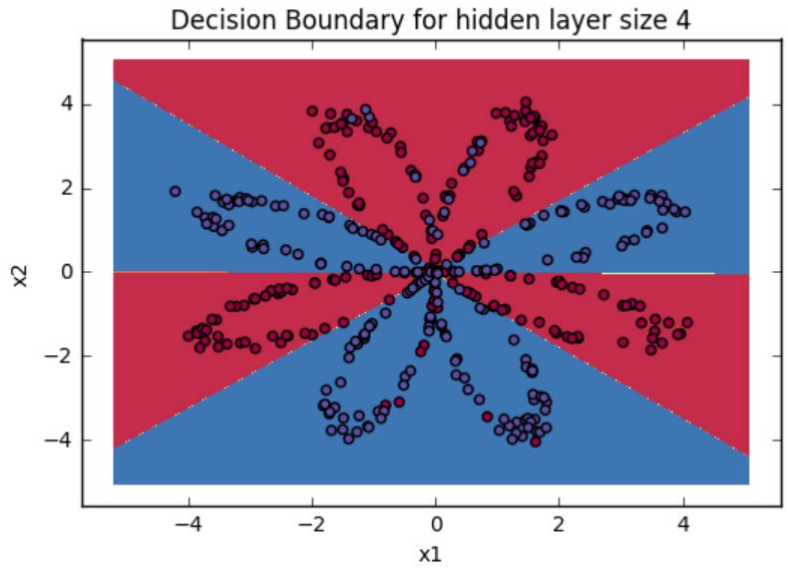
Приклад 3.13. Прогнозування на планарному наборі даних

```
# Будуємо модель з n_h-вимірним прихованим шаром
parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000,
print_cost=True)

# Позначимо межу рішень
plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
plt.title("Decision Boundary for hidden layer size " + str(4))
```

Вихід:

```
Cost after iteration 0: 0.684638
Cost after iteration 1000: 0.280383
Cost after iteration 2000: 0.268529
Cost after iteration 3000: 0.261708
Cost after iteration 4000: 0.257236
Cost after iteration 5000: 0.254139
Cost after iteration 6000: 0.251894
Cost after iteration 7000: 0.250204
Cost after iteration 8000: 0.248892
Cost after iteration 9000: 0.247846
```

```
<matplotlib.text.Text at 0x7fd7b114e438>
```




```
# Виведемо точність
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-
predictions.T))/float(Y.size)*100) + '%')
```

Вихід:

```
Accuracy: 91%
```

Точність дійсно висока в порівнянні з логістичною регресією. Модель вивчила візерунки листя квітки! Нейронні мережі здатні вивчати навіть дуже нелінійні межі рішень, на відміну від логістичної регресії.

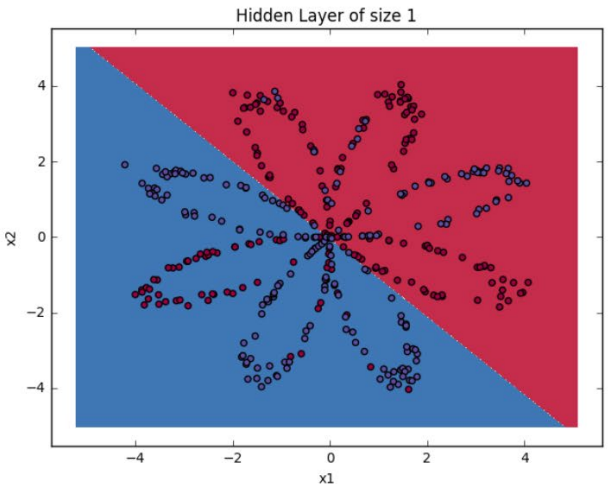
Від моделі з одним прихованим шаром можна перейти до побудови моделей з кількома прихованими шарами.

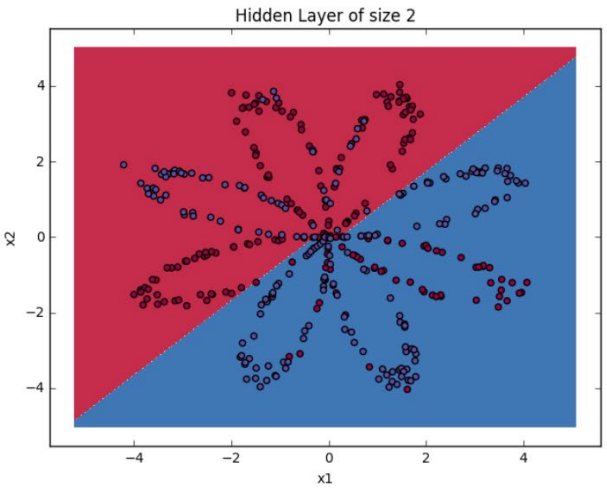
Приклад 3.14. Побудова моделі з кількома прихованими шарами

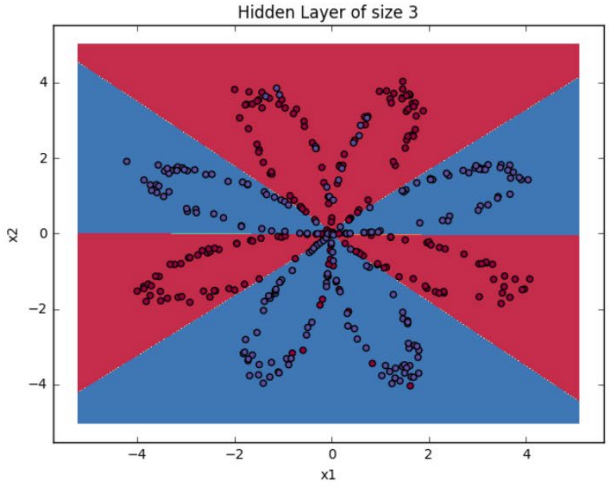
```
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i+1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations = 5000)
    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-
predictions.T))/float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

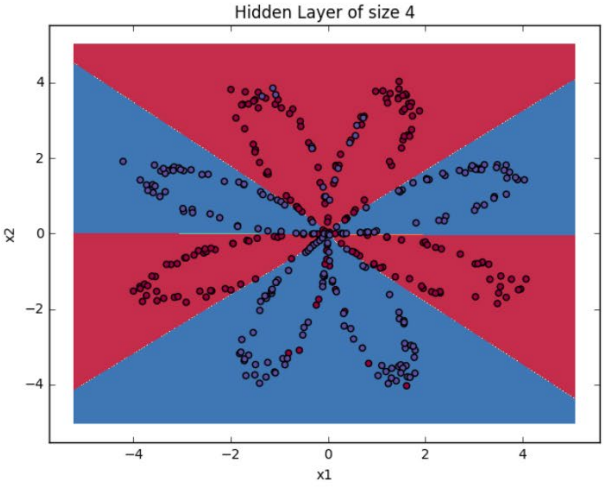
Вихід:

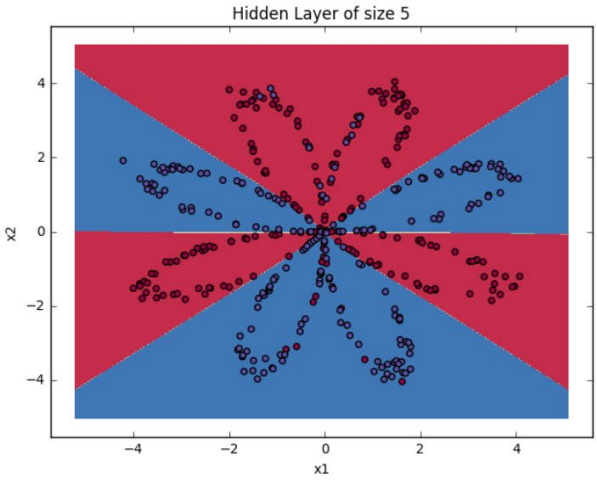
```
Accuracy for 1 hidden units: 66.25 %
Accuracy for 2 hidden units: 67.25 %
Accuracy for 3 hidden units: 90.75 %
Accuracy for 4 hidden units: 90.75 %
Accuracy for 5 hidden units: 91.0 %
Accuracy for 20 hidden units: 90.5 %

Accuracy for 50 hidden units: 90.75 %
```

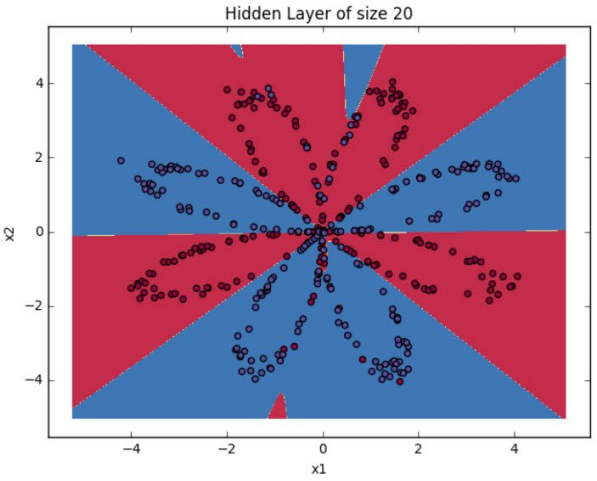

Hidden Layer of size 1
x1
x2

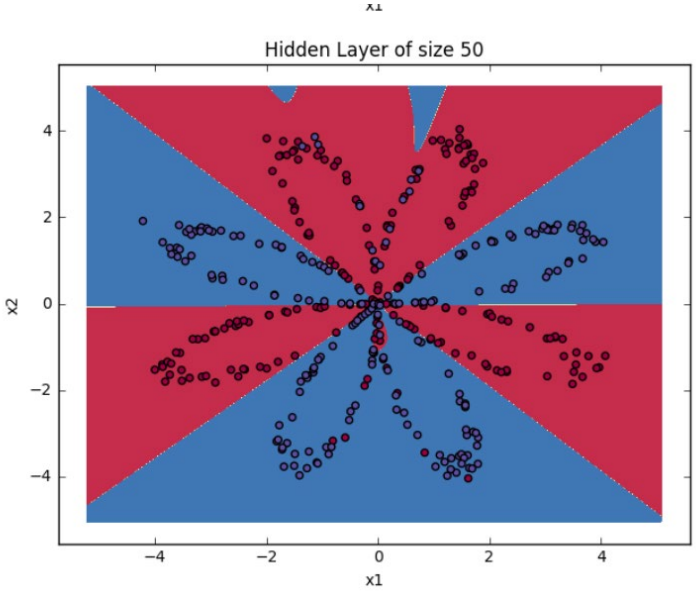

Hidden Layer of size 2
x1
x2


Hidden Layer of size 3
x1
x2


Hidden Layer of size 4
x1
x2


Hidden Layer of size 5
x1
x2


Hidden Layer of size 20
x1
x2



Більші моделі (з більшою кількістю прихованих блоків) можуть краще відповідати навчальному набору, поки в кінцевому підсумку не виникне переповнення даних. Здається, найкращий розмір прихованого шару становить близько n_h = 5. Дійсно, значення тут, здається, добре відповідає даним, не завдаючи при цьому помітного перенавчання. Пізніше буде розглянуто регуляризацію, яка дозволяє використовувати дуже великі моделі (наприклад, n_h = 50) без сильного перенавчання.

3.2. Порядок виконання роботи

3.2.1. Проаналізувати умову задачі.
3.2.2. Виконати завдання згідно свого варіанту.
3.2.3. Результати роботи оформити протоколом.

3.3 Завдання для виконання

Побудувати моделі з декількома прихованими шарами(не менше 7), визначити продуктивність для кожної моделі та визначити кількість шарів для найточнішого прогнозування для набору даних, зазначеному у вашому варіанті.
Варіанти наборів даних:

1. noisy_circles
2. noisy_moons
3. blobs
4. gaussian_quantiles
5. no_structure

3.4 Контрольні питання

1. Що таке нейронна мережа з прихованим шаром?
2. За що відповідає кожен шар нейронної мережі?
3. Чому нейронна мережа з прихованим шаром корисніша за просту логістичну регресію?
4. Якою є модель нейронної мережі з прихованим шаром?
5. Що таке алгоритми градієнтного спуску з хорошою швидкістю навчання (збіжність) і поганою швидкістю навчання?
6. Як визначити найкращу кількість шарів для прогнозування?

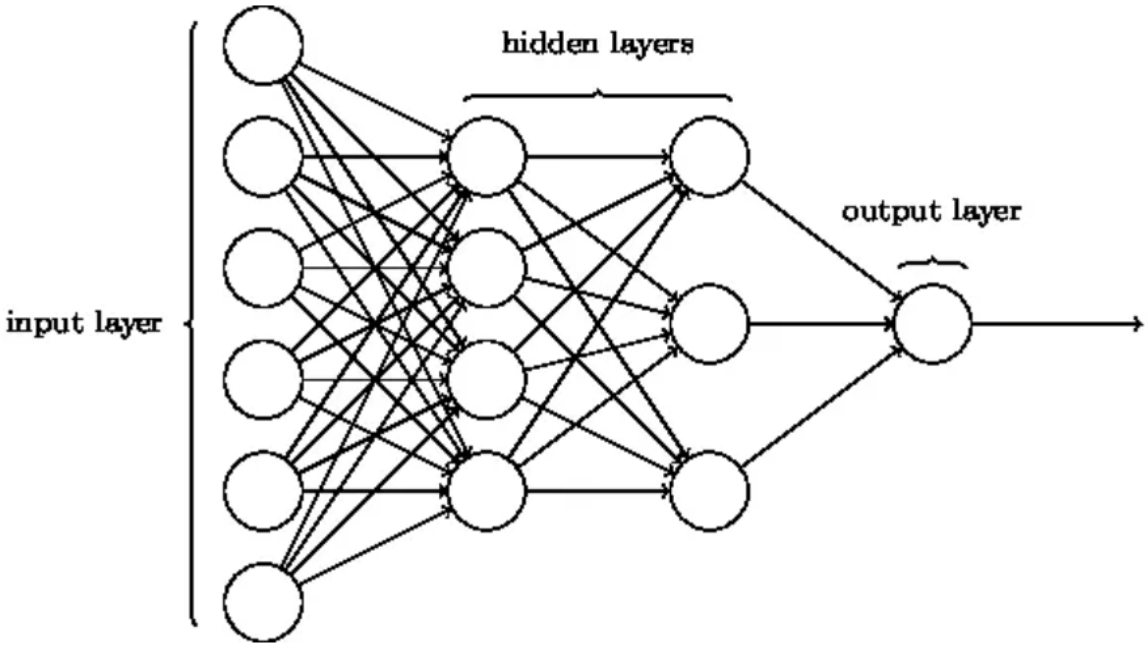
Лабораторна робота №4
Побудова глибокої нейронної мережі

Мета роботи: побудувати глибоку нейронну мережу.

4.1 Теоретичні відомості

Поняття глибокої нейронної мережі

Глибока нейронна мережа - це нейронна мережа з більш ніж одним прихованим шаром. Глибокі нейронні мережі використовують складне математичне моделювання для обробки даних складними способами.



Необхідні пакети

Щоб розпочати побудову глибокої нейронної мережі як завжди перш за все треба імпортувати необхідні для подальшої роботи пакети.
Під час виконання цього завдання знадобляться:

- **numpy** - є основним пакетом для наукових обчислень з Python.
- **matplotlib** - це бібліотека для побудови графіків на Python.
- **dnn_utils** - надає деякі необхідні функції для цієї лабораторної роботи.
- **testCases** - надає кілька тестових випадків для оцінки правильності функцій
- **np.random.seed(1)** - використовується для забезпечення узгодженості всіх випадкових викликів функцій. Це допоможе нам оцінити вашу роботу.

Приклад 4.1. Імпорт пакетів

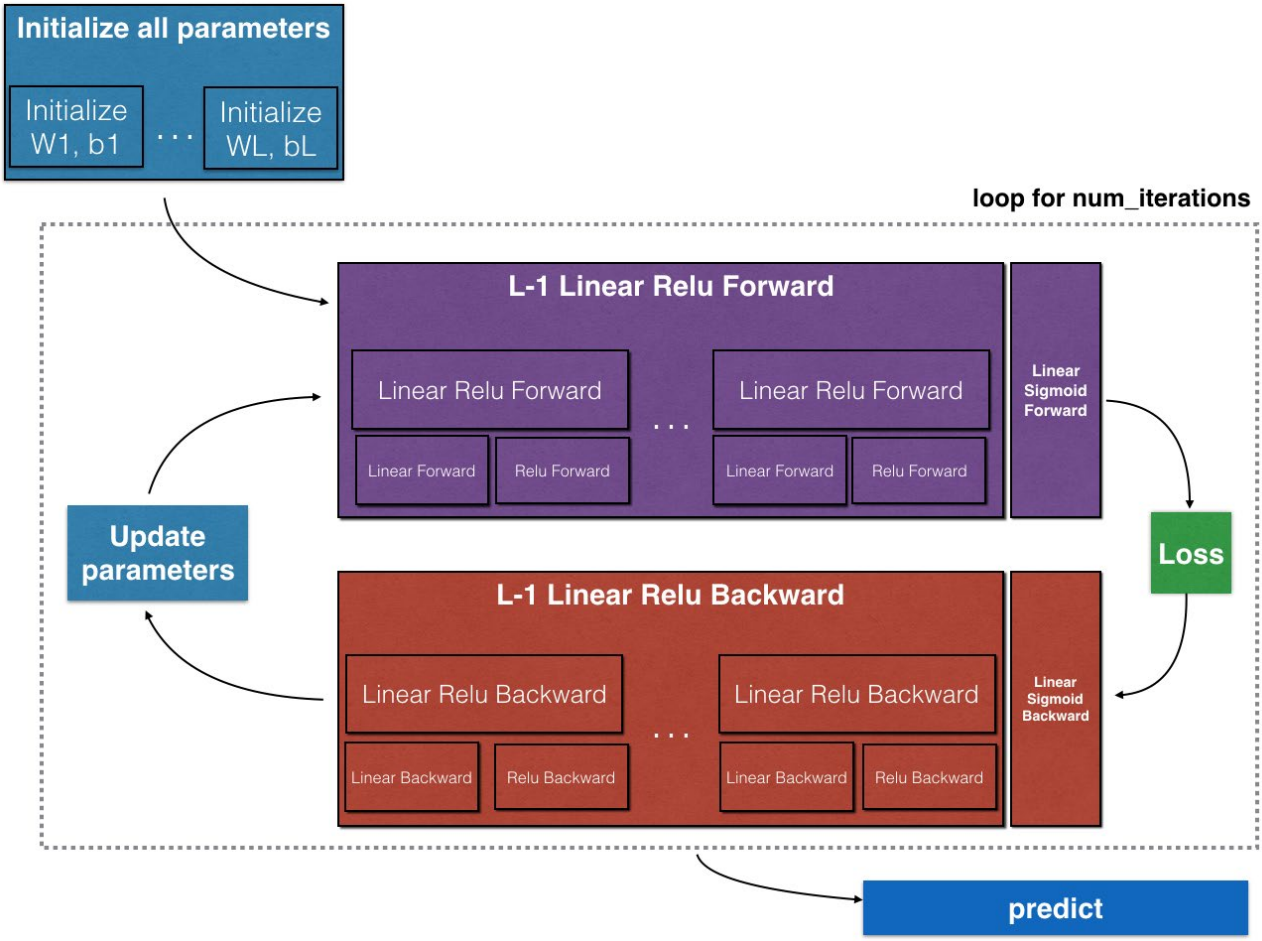
```
import time
import numpy as np
```

```
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
from dnn_app_utils_v3 import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # встановлення розмірів графіків за замовчуванням
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

Схема завдання

Щоб побудувати свою нейронну мережу, ми будемо реалізовувати кілька «допоміжних функцій». Ці допоміжні функції будуть використані і для побудови двошарової нейронної мережі та L-шарової нейронної мережі. Надамо схему для виконання цього завдання:

1. Ініціалізувати параметри для двошарової мережі та для L-шарової нейронної мережа.
2. Реалізувати модуль прямого поширення (показаний фіолетовим кольором на малюнку нижче).
   - Завершити ЛІНІЙНУ частину кроку прямого поширення шару (результати в $Z^{[l]}$).
   - Створити функцію АКТИВАЦІЇ (relu/sigmoid).
   - Об'єднати попередні два кроки в нову [LINEAR->ACTIVATION] функцію.
   - Скласти функцію [LINEAR->RELU] вперед L-1 раз (для шарів з 1 по L-1) і додати [LINEAR- > SIGMOID] в кінці (для останнього шаруL). Це дає нову функцію L_model_forward.
3. Обчислити втрату.
4. Реалізувати модуль зворотного поширення (позначений червоним на малюнку нижче).
   - Завершити ЛІНІЙНУ частину кроку зворотного поширення шару.
   - Обчислити градієнт функції ACTIVATE(relu_backward/sigmoid_backward)
   - Об'єднайте попередні два кроки в нову функцію [LINEAR->ACTIVATION] назад.
   - Складись [LINEAR->RELU] назад L-1 раз і додати [LINEAR->SIGMOID] назад у новій функції L_model_backward
5. Оновити параметри.



Зауважимо, що для кожної прямої функції існує відповідна зворотна функція. Ось чому на кожному кроці модуля вперед зберігаємо деякі значення в кеші. Кешовані значення корисні для обчислення градієнтів. У модулі зворотного поширення будемо використовувати кеш для обчислення градієнтів.

Ініціалізація

Напишемо дві допоміжні функції, які ініціалізують параметри для нашої моделі. Перша функція буде використовуватися для ініціалізації параметрів для двошарової моделі. Друга узагальнить цей процес ініціалізації для L-шарової моделі.

2-шарова нейронна мережа

Проведемо створення та ініціалізацію параметрів 2-шарової нейронної мережі. Структура моделі така: LINEAR -> RELU -> LINEAR -> SIGMOID
Використаємо випадкову ініціалізацію для вагових матриць. Скористаємося np.random.randn(shape)*0.01 з правильною розмірністю. Використаємо нульову ініціалізацію для зміщень. Використаємо np.zeros(shape) .

Приклад 4.2. Ініціалізація параметрів моделі

```
def initialize_parameters(n_x, n_h, n_y):
    """
    Аргументи:
    n_x - розмір вхідного шару
    n_h - розмір прихованого шару
    n_y - розмір вихідного шару

    Повернення:
    parameters - словник, що містить параметри :
                    W1 - ваги матриці, розмірності (n_h, n_x)
```

```
                    b1 - зміщення вектора, розмірності (n_h, 1)
                    W2 - ваги матриці, розмірності (n_y, n_h)
                    b2 - зміщення вектора, розмірності  (n_y, 1)
    """

    np.random.seed(1)


    W1 = np.random.randn(n_h,n_x)*0.01
    b1 = np.zeros([n_h,1])
    W2 = np.random.randn(n_y,n_h)*0.01
    b2 = np.zeros([n_y,1])

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

```
parameters = initialize_parameters(3,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Вихід:

```
W1 = [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.01744812 -0.00761207]]
b2 = [[0.]]
```

L-шарова нейронна мережа

Ініціалізація для більш глибокої нейронної мережі з L шарів є складнішою, оскільки існує набагато більше вагових матриць і векторів зміщення. При виконанні initialize_parameters_deep(), треба переконатися, що розміри збігаються між кожним шаром. Згадайте, що $n^{[l]}$ - кількість одиниць у шарі l. Таким чином, наприклад, якщо розмір нашого вхідного X є (12288, 209) (з m = 209 зразків), тоді:

	Shape of W	**Shape of b**	**Activation**	**Shape of Activation**
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]} X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
⋮	⋮	⋮	⋮	⋮
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]} A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Пам’ятайте, коли ми обчислюємо WX + b в python, здійснюється трансляція. Наприклад, якщо:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix}$$

Тоді WX + b:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix}$$

Тепер реалізуємо ініціалізацію для нейронної мережі L-шару.

Структура моделі така: *[LINEAR -> RELU] × (L−1) -> LINEAR -> SIGMOID*. Тобто модель має L −1 шари, які використовують функцію активації ReLU, а потім вихідний шар із сигмоподібною функцією активації.

Використаємо випадкову ініціалізацію для вагових матриць. Скористаємося np.random.rand(shape) * 0.01 .

Використаємо ініціалізацію нулями для зміщень.

Будемо зберігати $n^{[l]}$ одиниць у різних шарах у змінній layer_dims. Наприклад, для "моделі класифікації планарних даних" з минулої лабораторної layer_dims було б [2,4,1]: Були два входи, один прихований шар з 4 прихованими одиницями і вихідний шар з 1 вихідною одиницею. Це означає, що розмірність W1 була (4,2), b1 була (4,1), W2 була (1,4) і b2 була (1,1

Ось реалізація для L-1 - одношарової нейронної мережі.

Приклад 4.3. Реалізація ініціалізації параметрів для L-1 - одношарової нейронної мережі

```
if L == 1:
      parameters["W" + str(L)] = np.random.randn(layer_dims[1], layer_dim
s[0]) * 0.01
      parameters["b" + str(L)] = np.zeros((layer_dims[1], 1))
```

Тепер перенесемо реалізацію ра модель з більшою кількістю прихованих шарів.

Приклад 4.4. Реалізація ініціалізації параметрів для L-шарової нейронної мережі

```
def initialize_parameters_deep(layer_dims):
    """
    Аргументи:
    layer_dims - список, що містить розмірності кодного шару у мережі

    Повернення:
    parameters - словник, що містить параметри "W1", "b1", ..., "WL", "bL":
    Wl - ваги матриці, розмірності (layer_dims[l], layer_dims[l-1])
    bl - зміщення вектора, розмірності (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)            # число шарів у мережі

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims
[l-1]) * 0.01
        parameters['b' + str(l)] = np.zeros([layer_dims[l],1])

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[
l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

```
parameters = initialize_parameters_deep([5,4,3])
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

Вихід:

```
W1 = [[ 0.01788628  0.0043651   0.00096497 -0.01863493 -0.00277388]
 [-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]
 [-0.01313865  0.00884622  0.00881318  0.01709573  0.00050034]
 [-0.00404677 -0.0054536  -0.01546477  0.00982367 -0.01101068]]
b1 = [[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
W2 = [[-0.01185047 -0.0020565   0.01486148  0.00236716]
 [-0.01023785 -0.00712993  0.00625245 -0.00160513]
 [-0.00768836 -0.00230031  0.00745056  0.01976111]]
b2 = [[ 0.]
 [ 0.]
 [ 0.]]
```

Модуль прямого поширення

Побудова лінійної частини прямого поширення

Тепер, коли ми ініціалізували свої параметри, можна зробити модуль прямого поширення. Почнемо з реалізації деяких основних функцій, які будемо

використовувати пізніше під час реалізації моделі. Виконаэмои три функції в такому порядку:
- LINEAR
- LINEAR -> ACTIVATION де ACTIVATION буде або ReLU або Sigmoid.
- [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID (уся модель)
Лінійний прямий модуль (векторизований на всіх прикладах) обчислюється за таким рівнянням:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \quad ,$$

де $A^{[0]} = X$.
Побудуємо лінійну частину прямого поширення. Нагадування: математичне представлення цієї одиниці:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

Приклад 4.5 Побудова лінійної частини прямого поширення

```
def linear_forward(A, W, b):
    """
    Аргументи:
    A – активація з попереднього шару (або вхідні дані): (розмір попереднього шарyr, число прикладів)
    W – ваги матриці: масив розмірності (розмір поточного шару, розмір попереднього шару)
    b – зміщення вектора, масив форми (розмір поточного шару, 1)

    Повернення:
    Z – вхід активаційної функції, так званих преактиваційний параметр
    cache – словник, що містить "A", "W" and "b" ;  зберігається для ефективного обчислення зворотного проходу
    """
    """

    Z = np.dot(W,A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache
```

```
A, W, b = linear_forward_test_case()

Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z))
```

Вихід:

```
Z = [[ 3.26295337 -1.23429987]]
```

Лінійна активація вперед

У цьому блоці будемо вористовувати дві функції активації:

- ***Сигмоїд:***

$$\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}.$$

Ця функція повертає два елементи: значення активації "A" і "кеш", який містить "Z" (це те, що ми будемо вводити у відповідну функцію назад). Щоб скористатися нею, можна просто викликати:

```
A, activation_cache = sigmoid(Z)
```

- ***ReLU:***

$$A = RELU(Z) = max(0, Z)$$

Щоб скористатися нею, можна просто викликати:

```
A, activation_cache = relu(Z)
```

Виконаємо пряме поширення LINEAR->ACTIVATION шару. Математичне співвідношення $A^{[l]} = g(Z^{[l]}) = g(W^{[l]}A^{[l-1]} + b^{[l]})$, де активація g може бути sigmoid() або relu().

Приклад 4.6. Виконання прямого поширення вперед

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Аргументи:
    A_prev – активацыя з попереднього шару (або вхідні
дані): (size of previous layer, number of examples)
    W – ваги ватриці: масив розмірності (розмір поточного шару, розмір
попереднього шару)
    b -- зміщення, масив розмірності (розмір поточного шару, 1)
    activation – активація, що буде використана в цьому шарі, зберігається
як строка: "sigmoid" або "relu"

    Повернення:
    A – вихід з функії активації, також званих постактиваційне значення
    cache – словник, що містить "linear_cache" та "activation_cache";
             зберігається для ефективного обчислення зворотного проходу
    """

    if activation == "sigmoid":
        # Вхід: "A_prev, W, b". Вихід: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        # Вхід: "A_prev, W, b". Вихід: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache

A_prev, W, b = linear_activation_forward_test_case()

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activat
ion = "sigmoid")
print("With sigmoid: A = " + str(A))

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activat
ion = "relu")
print("With ReLU: A = " + str(A))
```

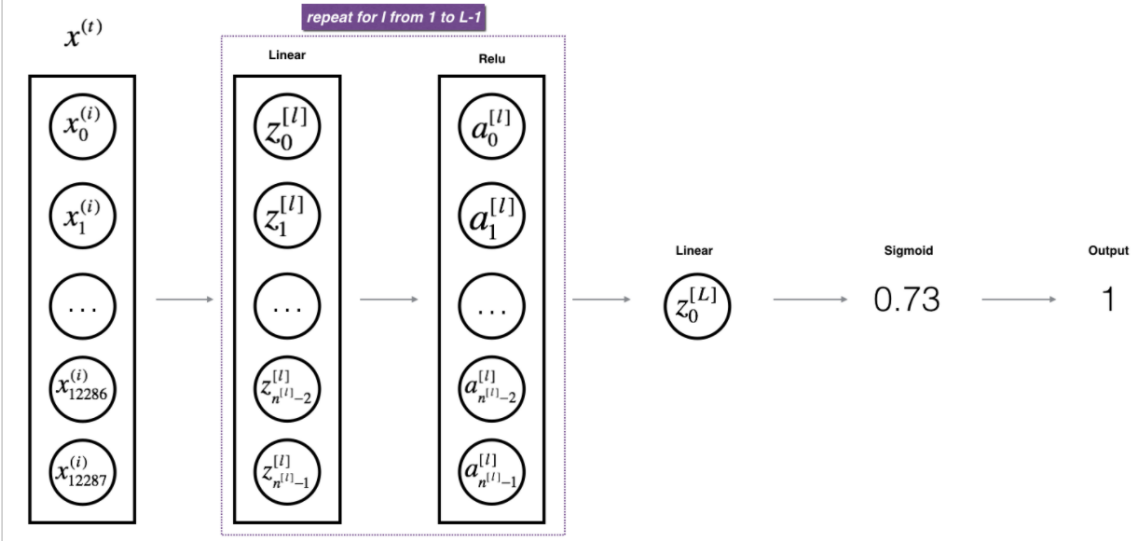
Вихід:

```
With sigmoid: A = [[ 0.96890023  0.11013289]]
With ReLU: A = [[ 3.43896131  0.        ]]
```

Примітка: у глибокому навчанні обчислення LINEAR->ACTIVATION враховується як один шар у нейронній мережі, а не два шари.

L-шарова модель

Для ще більшої зручності під час впровадження L-шаровлої нейронної сітки, знадобиться функція, яка повторює попередню (linear_activation_forward() **з RELU) L −1 раз, потім слідує з однією** linear_activation_forward() з SIGMOID



Виконаємо пряме поширення вищевказаної моделі.

Приклад 4.7. Функція прямого поширення

```
def L_model_forward(X, parameters):
    """
    Аргументи:
    X - дані, масив розмірності (вхідний розмір, число прикладів)
    parameters - вихід з initialize_parameters_deep()

    Повернення:
    AL - останнє пост-активаційне значення.
    caches - список кешів, що містить::
                кожен кеш з linear_activation_forward() (їх там L-
1, індексація з 0 до L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2                  # число шарів в рейронній сітці

    # Реалізація [LINEAR -> RELU]*(L-1). Додавання "кешу" списку кешів.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev,parameters['W'+str(l)],p
arameters['b'+str(l)],'relu')
```

```
        caches.append(cache)

    # Реалізація LINEAR -> SIGMOID. Додавання "кешу" списку кешів.
        AL, cache = linear_activation_forward(A, parameters['W'+str(l+1)], p
arameters['b'+str(l+1)],'sigmoid')
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches
```

```
X, parameters = L_model_forward_test_case_2hidden()
AL, caches = L_model_forward(X, parameters)
print("AL = " + str(AL))
print("Length of caches list = " + str(len(caches))
```

Вихід:

```
AL = [[ 0.03921668  0.70498921  0.19734387  0.04728177]]
Length of caches list = 3
```

Чудово! Тепер у нас є повне пряме поширення, яке приймає вхідний X і виводить вектор-рядок AL, що містить наші прогнози. Він також записує всі проміжні значення в «кеші». Використовуючи AL, ми зможемо розрахувати вартість наших прогнозів.

Функція витрат

Тепер потрібно розрахувати вартість, щоб перевірити, чи дійсно модель навчається.

Обчислимо вартість крос-ентропії, використовуючи таку формулу:

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right))$$

Приклад 4.8. Реалізація функції витрат

```
def compute_cost(AL, Y):
    """
    Аргументи:
    AL - вектор ймовірності наших
передбачень, розмірності (1, number of examples)
    Y - правдиві "мітки" вектора (наприкладе: містить 0 якщо не кіт, 1 -
кіт), розмірності (1, number of examples)

    Повернення:
    cost -- перехресна ентропійна вартість
    """

    m = Y.shape[1]

    # Обчислимо втрати від aL і y.
    cost = (-1/m) * np.sum( np.dot(Y, np.log(AL).T) + np.dot((1-
Y), np.log(1-AL).T))
    cost = np.squeeze(cost)       # Перетворює, наприклад, [[17]] на 17).
    assert(cost.shape == ())

    return cost
```

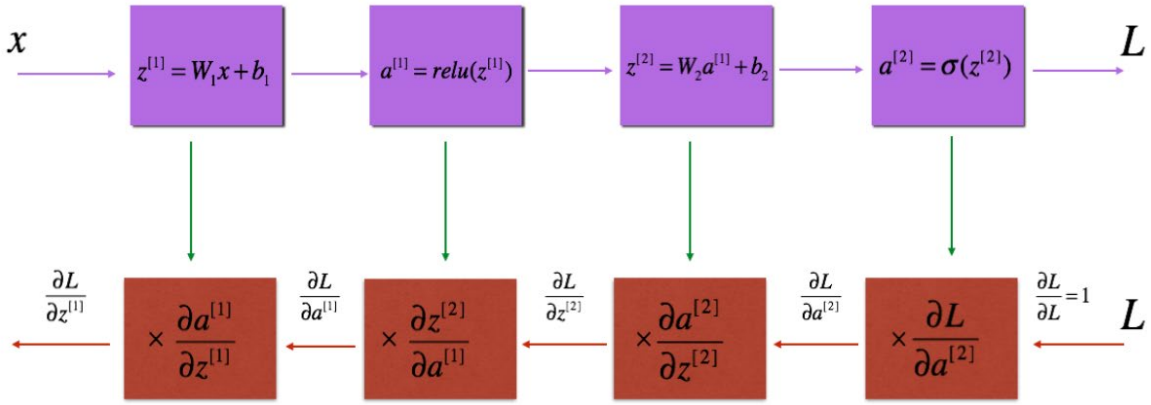
```
Y, AL = compute_cost_test_case()

print("cost = " + str(compute_cost(AL, Y)))
```

Вихід:

```
cost = 0.414931599615
```

Модуль зворотного поширення

Як і у випадку прямого поширення, нам потрібно реалізувати допоміжні функції для зворотного поширення. Пам'ятайте, що зворотне поширення використовується для обчислення градієнта функції втрат щодо параметрів.



* Фіолетові блоки представляють пряме поширення, а червоні блоки – зворотне.*

Тепер, подібно до прямого поширення, побудуємо зворотне поширення в три кроки:

- LINEAR backward
- LINEAR -> ACTIVATION назад, де ACTIVATION ON обчислює похідну ReLU або sigmoid активації
- [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID backward (уся модель)

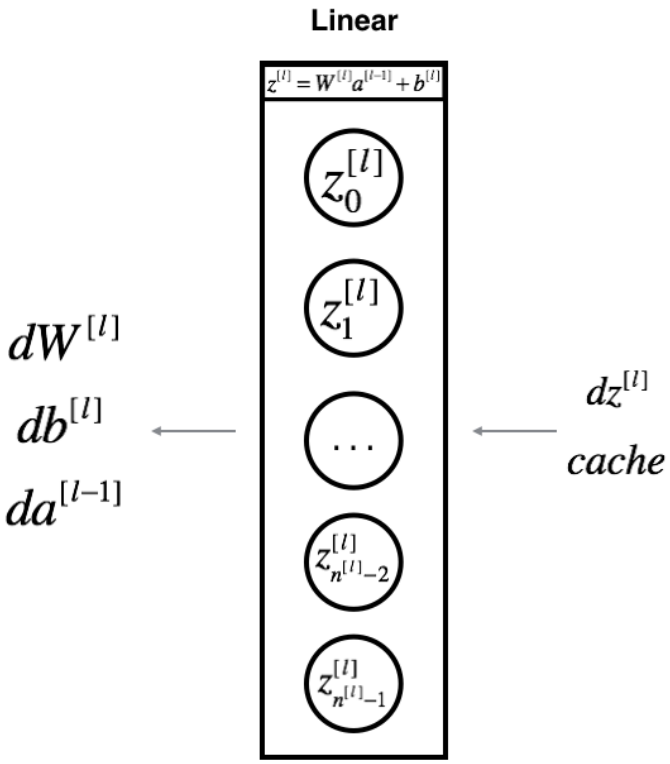
Побудова лінійної частини зворотного поширення

Для шару *l,* лінійна частина така:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

(з наступною активацією).

Припустимо, ми вже обчислили похідну. $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$. Ми хочемо отримати ($dW^{[l]}$,$db^{[l]}$,$dA^{[l-1]}$)



Три виходи ($dW^{[l]}$,$db^{[l]}$,$dA^{[l]}$) обчислюються за допомогою входу $dZ^{[l]}$.Ось потрібні формули:

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

Використовуюси 3 формули вище, реалізуємо функцію linear_backward().

Приклад 4.9. Реалізація функція linear_backward().

```
def linear_backward(dZ, cache):
    """
    Аргументи:
    dZ - градієнт вартостей with по відношенню до лінійного
виходу (поточного шару l)
    cache -- набір значень (A_prev,W, b), що надходять від прямого поширення
в поточному шарі

Повернення:
    dA_prev - Градієнт вартості по відношенню до активації (попереднього
шару l-1), такої ж розмірності, як A_prev
      dW -- Градієнт вартості по відношенню до W (поточний шар l), такої ж
розмірності, як W
    db -- Градієнт вартості щодо b (поточний шар l), такої ж розмірності як
b
    """
    A_prev, W, b = cache
```

```
    m = A_prev.shape[1]

    dW = 1/m * np.dot(dZ, A_prev.T)
    db = np.sum(dZ, axis=1, keepdims=True)/m
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

```
# Налаштуємо деякі тестові входи
dZ, linear_cache = linear_backward_test_case()

dA_prev, dW, db = linear_backward(dZ, linear_cache)
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

Вихід:

```
dA_prev = [[ 0.51822968 -0.19517421]
 [-0.40506361  0.15255393]
 [ 2.37496825 -0.89445391]]
dW = [[-0.10076895  1.40685096  1.64992505]]
db = [[ 0.50629448]]
```

Лінійна зворотна активація

Далі створимо функцію, яка об'єднає дві допоміжні функції: linear_backward() і зворотній крок для активації linear_activation_backward

Надамо дві зворотні функції:

- sigmoid_backward : реалізує зворотне поширення для блоку SIGMOID. Викликається так:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- relu_backward : реалізує зворотне поширення для блоку RELU. Викликається так:

```
dZ = relu_backward(dA, activation_cache)
```

Якщо g(.) є функцією активації, sigmoid_backward і relu_backward обчислюються так:

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$

Приклад 4.10. Реалізація функції лінійного зворотнього поширення

```
def linear_activation_backward(dA, cache, activation):
    """
    Аргументи:
    dA - градієнт після активації для поточного шару l
    cache - набір значень (linear_cache, activation_cache), які ми
зберігаємо для ефективного обчисленя зворотного поширення
activation - активація, яка буде використовуватися на поточному шарі,
зберігається у вигляді текстового рядка: "sigmoid" або "relu"

    Повернення:
    dA_prev - Градієнт вартості по відношенню до активації (попереднього
шару l-1), такої ж розмірності, як A_prev
```

```
    dW -- Градієнт вартості по відношенню до W (поточний шар l), такої ж
розмірності, як W
    db -- Градієнт вартості щодо b (поточний шар l), такої ж розмірності як
b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    elif activation == "sigmoid":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    return dA_prev, dW, db
```

```
dAL, linear_activation_cache = linear_activation_backward_test_case()

dA_prev, dW, db = linear_activation_backward(dAL, linear_activation_cache, a
ctivation = "sigmoid")
print ("sigmoid:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db) + "\n")

dA_prev, dW, db = linear_activation_backward(dAL, linear_activation_cache, a
ctivation = "relu")
print ("relu:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

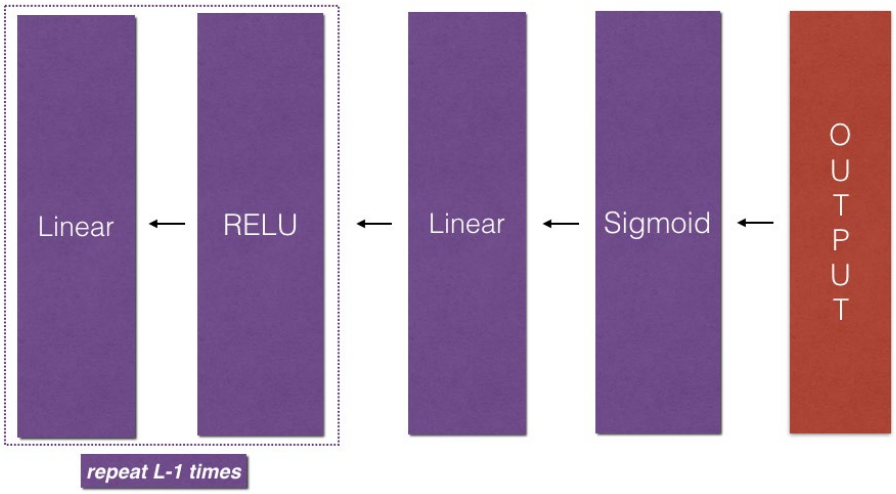
Вихід:

```
sigmoid:
dA_prev = [[ 0.11017994  0.01105339]
 [ 0.09466817  0.00949723]
 [-0.05743092 -0.00576154]]
dW = [[ 0.10266786  0.09778551 -0.01968084]]
db = [[-0.05729622]]

relu:
dA_prev = [[ 0.44090989  0.        ]
 [ 0.37883606  0.        ]
 [-0.2298228   0.        ]]
dW = [[ 0.44513824  0.37371418 -0.10478989]]
db = [[-0.20837892]]
```

Зворотна L-модель

Тепер реалізуємо зворотну функцію для всієї мережі. Згадаємо, що коли ми впроваджували L_model_forward(), на кожній ітерації зберігаєли кеш, який містить (X, W, b і z). В зворотньому модулі поширення, використаємо ці змінні

для обчислення градієнтів. Тому в L_model_backward(), треба перебирати всі приховані шари назад, починаючи з шару L. На кожному кроці будемо використовувати кешовані значення для шару l для зворотного поширення через шар l. На малюнку 5 нижче показано перехід назад:



Ініціалізація зворотного поширення: для зворотного поширення через цю мережу ми знаємо, що вихід: $A^{[L]}=\sigma(Z^{[L]})$ Таким чином, ваш код повинен обчислювати $dAL = \frac{\partial \mathcal{L}}{\partial A^{[L]}}$.

Для цього скористайтеся цією формулою (отриманою за допомогою обчислення, у яке вам не слыд заглиблюватися):

```
dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL)) # похідна вартості по відношенню до AL
```

Потім ви можете використовувати цей градієнт dAL після активації, щоб продовжувати рухатися назад. Як видно на попередньому малюнку, тепер можна подавати dAL у реалізовану нами функцію LINEAR->SIGMOID (яка використовуватиме кешовані значення, збережені функцією L_model_forward()). Після цього треба використовувати цикл for для перебору всіх інших шарів за допомогою функції LINEAR->RELU назад. Треба зберігати кожні dA, dW і db у словнику grads. Для цього скористаємося цією формулою:

$$grads[" \, dW \, " + str(l)] = dW^{[l]}$$

Наприклад, для l=3 повинно зберігатися $dW^{[l]}$ в grads["dW3"].

Виконаємо зворотне поширення для *[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID* моделі.

Приклад 4.11. Реалізація зворотнього поширення для *[LINEAR->RELU] × (L-1) -> LINEAR -> SIGMOID* моделі

```
def L_model_backward(AL, Y, caches):
    """
    Аргументи:
    AL -- вектор ймовірності, вихід прямого поширення (L_model_forward())
```

```
    Y -- правдиві "мітки" у формі вектора (містить 0 якщо не кіт, 1 якщо
кіт)
    caches - список кешів, що містить:
                кожен
кеш linear_activation_forward() з "relu" (це caches[l], для l в межах(L-
1) тобто l = 0...L-2)
кеш linear_activation_forward() з "sigmoid" (це caches[L-1])

    Повернення:
    grads - словник градієнтів
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # число шарів
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # після цього рядка, Y стає такої ж розмірності
як AL

    # Ініціалізація зворотнього поширенння
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # L-тий шар (SIGMOID > LINEAR) градієнтів.
Вхід: "AL, Y, caches". Вихід: "grads["dAL"], grads["dWL"], grads["dbL"]

    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linea
r_activation_backward(dAL, current_cache, "sigmoid")

    for l in reversed(range(L-1)):
        # l-ший шар: (RELU -> LINEAR) градієнтів.
        # Вхід: "grads["dA" + str(l + 2)], caches". Вихід: "grads["dA" + str
(l + 1)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["d
A" + str(l + 2)], current_cache, "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
    return grads
```

```
AL, Y_assess, caches = L_model_backward_test_case()
grads = L_model_backward(AL, Y_assess, caches)
print_grads(grads)
```

Вихід:

```
dW1 = [[ 0.41010002  0.07807203  0.13798444  0.10502167]
 [ 0.          0.          0.          0.        ]
 [ 0.05283652  0.01005865  0.01777766  0.0135308 ]]
db1 = [[-0.22007063]
 [ 0.        ]
 [-0.02835349]]
dA1 = [[ 0.          0.52257901]
 [ 0.         -0.3269206 ]
 [ 0.         -0.32070404]
 [ 0.         -0.74079187]]
```

Оновлення параметрів

У цьому розділі оновимо параметри моделі, використовуючи градієнтний спуск:

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha\, db^{[l]}$$

,

де α це швидкість навчання. Після обчислення оновлених параметрів збережемо їх у словнику параметрів.

Приклад 4.12. Оновлення параметрів

```
def update_parameters(parameters, grads, learning_rate):
    """

    Аргументи:
    parameters - словник , що містить параметри
    grads – словник, що містить градієнти, виходт L_model_backward()

    Повернення:
    parameters - словник , що містить оноавлені параметри
                 parameters["W" + str(l)] = ...
                 parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # число шарів в нейронній мережі

    # Оновимо правила кодного параметра в циклі:
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] -
 learning_rate * grads["dW" + str(l + 1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] -
 learning_rate * grads["db" + str(l + 1)]
        return parameters
```

```
parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads, 0.1)

print ("W1 = " + str(parameters["W1"]))
print ("b1 = " + str(parameters["b1"]))
print ("W2 = " + str(parameters["W2"]))
print ("b2 = " + str(parameters["b2"]))
print ("W3 = " + str(parameters["W3"]))
print ("b3 = " + str(parameters["b3"]))
```

Вихід:

```
W1 = [[ 1.72555789  0.3700272   0.07818896]
 [-1.8634927  -0.2773882  -0.35475898]
 [-0.08274148 -0.62700068 -0.04381817]
 [-0.47721803 -1.31386475  0.88462238]]
b1 = [[-0.07593768]
 [-0.07593768]
 [-0.07593768]
 [-0.07593768]]
W2 = [[ 0.71838378  1.70957306  0.05003364 -0.40467741]
 [-0.54535995 -1.54647732  0.98236743 -1.10106763]
 [-1.18504653 -0.2056499   1.48614836  0.23671627]]
b2 = [[-0.08616376]
 [-0.08616376]
 [-0.08616376]]
W3 = [[-0.88352436 -0.7129932   0.62524497]
 [-0.02025258 -0.76883635 -0.23003072]]
b3 = [[ 0.08416196]
 [ 0.08416196]]
```

4.2. Порядок виконання роботи

4.2.1. Проаналізувати умову задачі.

4.2.2. Виконати задачу.

4.2.3. Результати роботи оформити протоколом.

4.3 Завдання для виконання

Побудувати глибоку нейронну мережу за прикладом, наведеним у теоретичних відомостях.

4.4 Контрольні питання

1. Що таке глибока нейронна мережа?
2. Які допоміжні функції потрібно створити для її реалізації?
3. Опишіть схему ініціалізації.
4. Опишіть модуль прямого поширення.
5. Опишіть функцію витрат.
6. Опишіть модуль зворотного поширення.

Лабораторна робота №5
Глибока нейронна мережа для класифікації зображень: застосування

Мета роботи: застосуйте глибоку нейронну мережу для класифікації зображень.

5.1 Теоретичні відомості

Під час виконання цієї роботи ми будемо використовувати функції, які ми реалізували в попередный лабораторній роботі, щоб побудувати глибоку мережу та застосувати її для класифікації cat vs non-cat. Сподіваємося, ви побачите покращення точності порівняно з попередньою реалізацією логістичної регресії.

Необхідні пакети

Давайте спочатку імпортуємо всі пакунки, які знадобляться під час виконання цього завдання, а саме:

- **numpy** - є основним пакетом для наукових обчислень з Python.
- **matplotlib** - це бібліотека для побудови графіків на Python.
- **dnn_utils** - надає деякі необхідні функції для цієї лабораторної роботи.
- **testCases** - надає кілька тестових випадків для оцінки правильності функцій
- **np.random.seed(1)** - використовується для забезпечення узгодженості всіх випадкових викликів функцій. Це допоможе нам оцінити вашу роботу.

Приклад 5.1. Імпорт пакетів

```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
from dnn_app_utils_v2 import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # вставлення розмірів графіків зпа замовчуванням
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

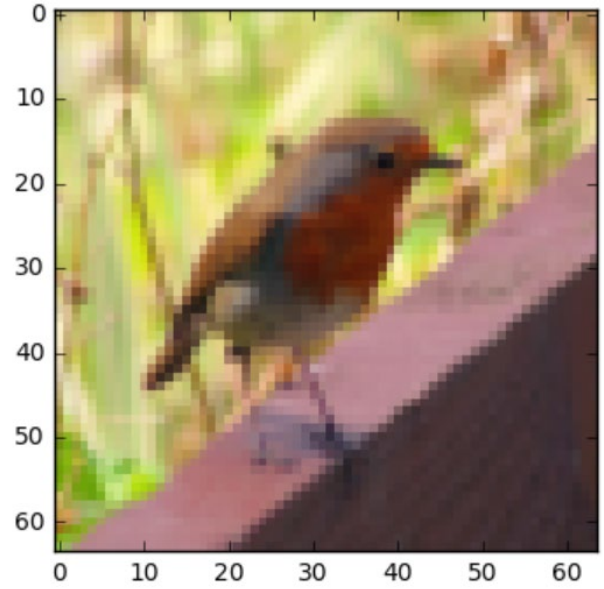
Завантаження даних

Ми будемо використовувати той самий набір даних "Cat vs non-Cat", що й у "Логістичній регресії як нейронної мережі" (лабораторна робота 2). Модель, яку ми тоді створили, мала 70% точності на тестових даних при класифікації зображень котів і не котів. Сподіваємося, нова модель буде працювати краще!
Набір даних ("data.h5") містить:
- навчальний набір зображень m_train, позначених як cat (1) або non-cat (0)
- тестовий набір зображень m_test, позначених як cat і non-cat
- кожне зображення має форму (num_px, num_px, 3), де 3 для 3 каналів (RGB).
Завантажимо дані та дослідимо їх:

Приклад 5.2. Завнтаження та дослідження вхідгних даних

```
train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0,index]) + ". It's a " +
classes[train_y[0,index]].decode("utf-8") +  " picture.")
```

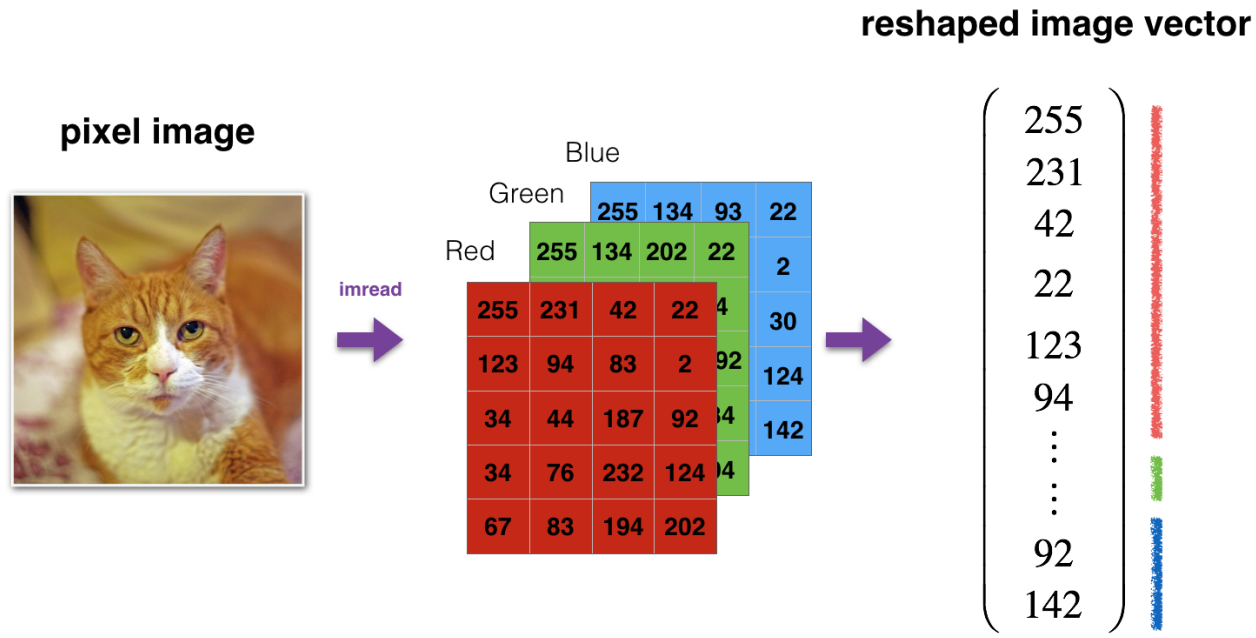
```
y = 0. It's a non-cat picture.
```




```
# Дослідження даних
m_train = train_x_orig.shape[0]
num_px = train_x_orig.shape[1]
m_test = test_x_orig.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ",
3)")
print ("train_x_orig shape: " + str(train_x_orig.shape))
print ("train_y shape: " + str(train_y.shape))
print ("test_x_orig shape: " + str(test_x_orig.shape))
print ("test_y shape: " + str(test_y.shape))
```

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
```

Як зазвичай, зміним форму та стандартизуємо зображення перед подачею їх у мережу.



Приклад 5.3. Зміна форми та стандартизація даних

```
# Зміна форми тренувальних та тестових прикладів
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T
"-1" змушує зміну форми вирівнювати решту розмірів
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

# Стандартизуємо дані, щоб мати значення функції від 0 до 1.
train_x = train_x_flatten/255.
test_x = test_x_flatten/255.

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))
```

Вихід:

```
train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

12,288 еквівалентно 64×64×3 , що є розмірм одного переформованого вектора зображення.

Архітектура моделі

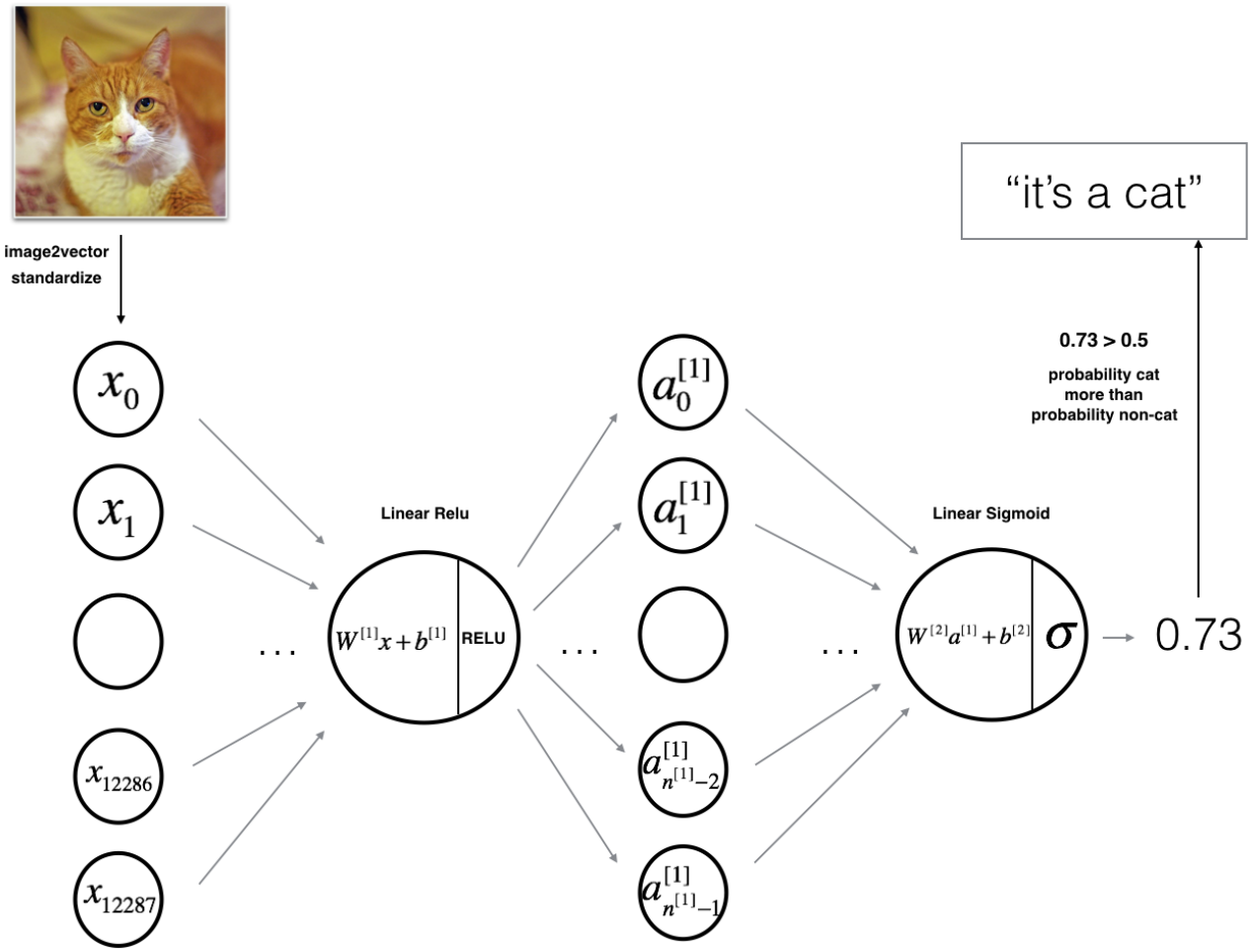
Тепер, ми вивчили набір даних, настав час побудувати глибоку нейронну мережу, щоб відрізняти зображення котів від зображень інших. Створимо дві різні моделі:

- 2-шарова нейронна мережа.

- L-шарова глибока нейронна мережа.

Потім порівняємо продуктивність цих моделей, а також спробуємо різні значення для L. Давайте подивимося на дві архітектури.

2-шарова глибока нейронна мережа



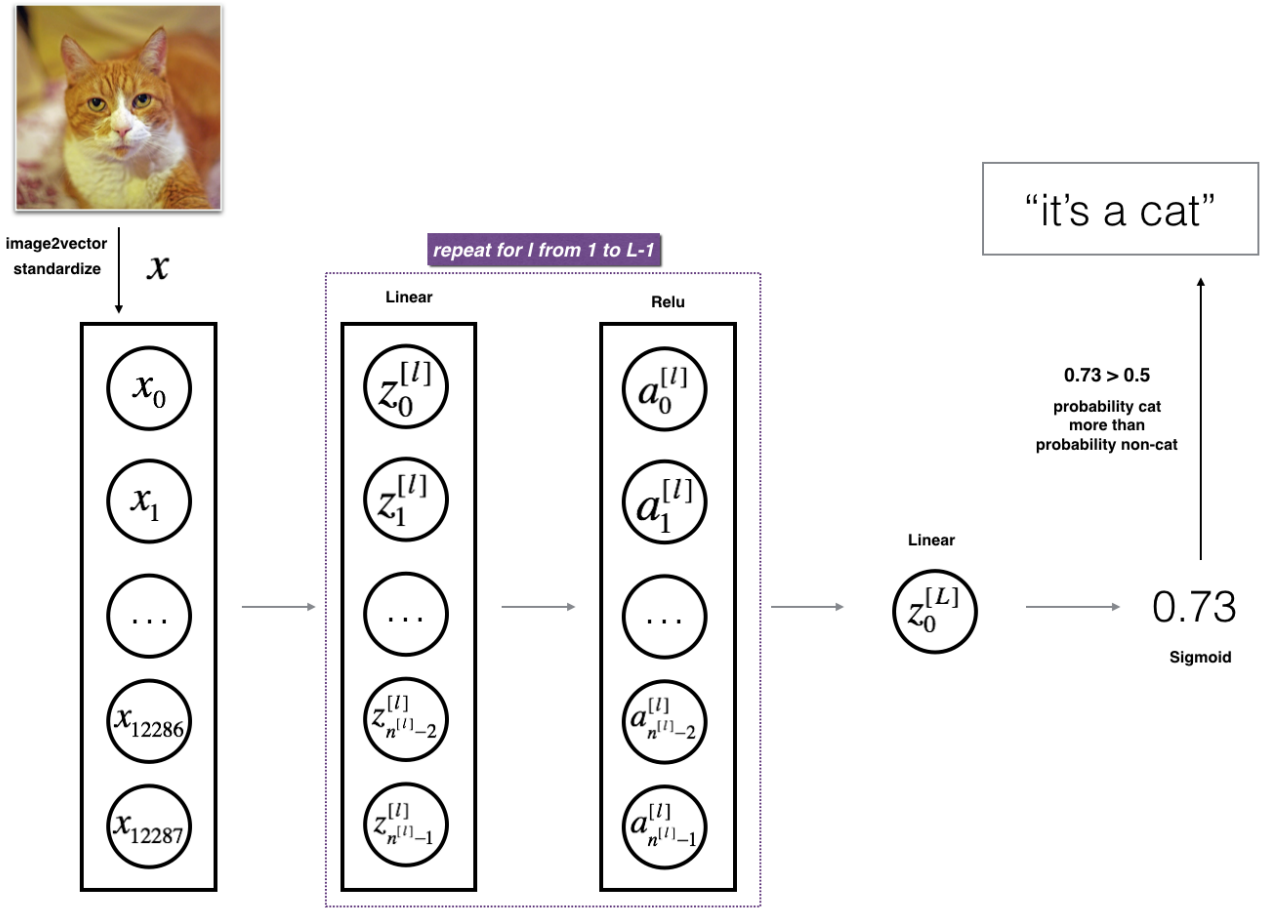
Модель можна узагальнити так: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

Детальна архітектура на малюнку:

- Вхідним є зображення (64,64,3), яке вирівнюється до вектора розміру (12288, 1).
- Відповідний вектор $[x_0,x_1,...,x_{12287}]^T$, потім множиться на вагову матрицю $W^{[1]}$ розміру $(n^{[1]},12288)(n^{[1]},12288)$.
- Потім додамо член зміщення та приймаємо його relu, щоб отримати такий вектор: $[a^{[1]}_0,a^{[1]}_1,...,a^{[1]}_{n[1]-1}]^T$
- Потім повторюємо той самий процес.
- Множимо отриманий вектор на W[2] і додаємо своє зміщення. Нарешті, будемо мати сигмовидну форму результату. Якщо вона більше 0,5, класифікуємо зоюраження як кішку.

L-шарова глибока нейронна мережа

Важко уявити глибоку нейронну мережу з L шарами з наведеним вище представленням. Однак ось спрощене представлення мережі:



Модель можна узагальнити так: ***[LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID***

Детальна архітектура малюнка:

- Вхідним є зображення (64,64,3), яке вирівнюється до вектора розміру (12288,1).
- Відповідний вектор $[x_0,x_1,...,x_{12287}]^T$, потім множиться на вагову матрицю $W^{[1]}$ розміру $(n^{[1]},12288)(n^{[1]},12288)$ і потім додаємо перехоплення $b^{[1]}$.. Результат називається лінійною одиницею.
- Далі беремо relu лінійної одиниці. Цей процес можна повторити кілька разів для кожного
  ($W^{[l]}$, $b^{[l]}$) в залежності від архітектури моделі.
- Нарешті, ви беремо сигмовидну форму кінцевої лінійної одиниці. Якщо вона більше 0,5, класифыкуэмо зображення як кішку.

Загальна методика

Як зазвичай, будемо дотримуватися методології глибокого навчання для побудови моделі:

1. Ініціалізація параметрів / Визначення гіперпараметрів
2. Цикл для num_iterations:

a. Прямек поширення
b. Обчислити
c. Зворотне поширення
d. Оновлення параметрів (використовуючи параметри та градації з backprop)

3. Використаємо навчені параметри для прогнозування міток функції витрат розмноження

Давайте тепер реалізуємо ці дві моделі!

Двошарова нейронна мережа

Тепер використаємо допоміжні функції, які ми реалізували в попередній лабораторній роботі, щоб побудувати 2-шарову нейронну мережу з такою структурою: *LINEAR -> RELU -> LINEAR -> SIGMOID*.

Функції, які можуть знадобитися, та їх вхідні дані:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

```
### Константне визначення моделі ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

Приклад 5.4. Побудова двошарової нейронної мережі

```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075,
num_iterations = 3000, print_cost=False):
    """

    Аргументи:
    X - вхіжні дані, розмірності (n_x, число прикладів)
    Y - правдиві "мітки" вектора (містять 0 якщо cat, 1 якщо non-cat),
розмірності (1, число прикладів)
    layers_dims - ромірність шару (n_x, n_h, n_y)
    num_iterations - число ітерацій оптимізаційного циклу
    learning_rate - швидкість навчання правила оновлення градієнтного спуску
    print_cost -- Якщо
встановлено значення True, це буде друкувати вартість кожні 100 ітерації

    Повернення:
```

```
    parameters - словник, що містить W1, W2, b1, і b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                                     # щоб відстежувати вартість
    m = X.shape[1]                                  # кількість прикладів
    (n_x, n_h, n_y) = layers_dims

    # Ініціалізуємо словник параметрів, викликавши одну з функцій, які ми
реалізували раніше

    parameters = initialize_parameters(n_x, n_h, n_y)

    # Отримаємо W1, b1, W2 і b2 зі словника параметрів.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Цикл (градієнтний спуск)

    for i in range(0, num_iterations):

        # Пряме поширення: LINEAR -> RELU -> LINEAR -> SIGMOID. Вхідні дані:
"X, W1, b1". Вихідні дані: "A1, cache1, A2, cache2".
        A1, cache1 = linear_activation_forward(X, W1, b1, activation=
"relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation=
"sigmoid")

        # Обчислення вартості
        cost = compute_cost(A2, Y)


        # Ініціалізаія зворотного поширення
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Зворотне поширення. Вхідні дані: "dA2, cache2, cache1". Вихідні
дані: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
        ### START CODE HERE ### (≈ 2 lines of code)
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation =
"sigmoid")
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation =
"relu")

        # Встановлення grads['dWl'] на dW1, grads['db1'] на db1,
grads['dW2'] днао dW2, grads['db2'] на db2
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Оновлення параметрів.
        )
        parameters = update_parameters(parameters, grads, learning_rate)

        # Отримання W1, b1, W2, b2 з параметрів
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
```

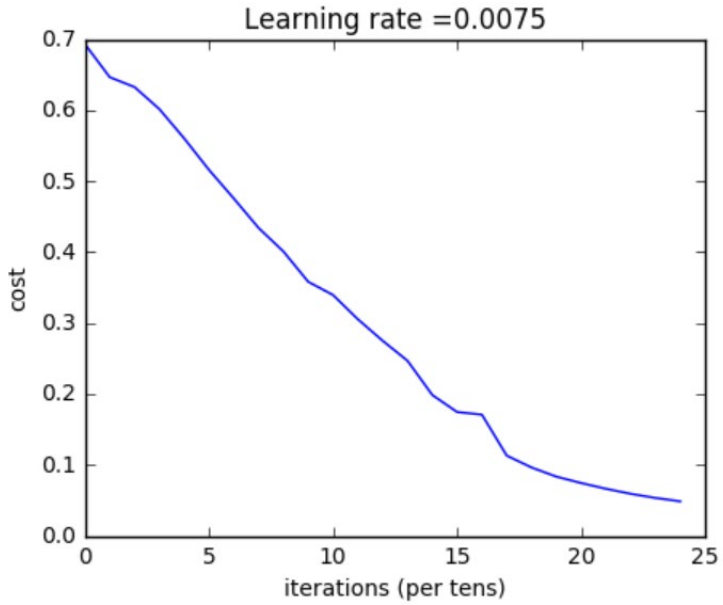
```
        b2 = parameters["b2"]

        # Друк вартості кожні 100 навчальних прикладів
        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # Графік вартості

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

Запустимо код нижче, щоб навчити параметри. Подивимося, чи працює модель. Вартість має зменшуватися. Для виконання 2500 ітерацій може знадобитися до 5 хвилин.

```
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y,
num_iterations = 2500, print_cost=True)
```

Вихід:

```
Cost after iteration 0: 0.693049735659989
Cost after iteration 100: 0.6464320953428849
Cost after iteration 200: 0.6325140647912678
Cost after iteration 300: 0.6015024920354665
Cost after iteration 400: 0.5601966311605748
Cost after iteration 500: 0.515830477276473
Cost after iteration 600: 0.4754901313943325
Cost after iteration 700: 0.43391631512257495
Cost after iteration 800: 0.4007977536203886
Cost after iteration 900: 0.35807050113237987
Cost after iteration 1000: 0.3394281538366413
Cost after iteration 1100: 0.30527536361962654
Cost after iteration 1200: 0.2749137728213015
Cost after iteration 1300: 0.24681768210614827
Cost after iteration 1400: 0.1985073503746611
Cost after iteration 1500: 0.17448318112556593
Cost after iteration 1600: 0.1708076297809661
Cost after iteration 1700: 0.11306524562164737
Cost after iteration 1800: 0.09629426845937163
Cost after iteration 1900: 0.08342617959726878
Cost after iteration 2000: 0.0743907870431909
Cost after iteration 2100: 0.06630748132267938
Cost after iteration 2200: 0.05919329501038176
Cost after iteration 2300: 0.05336140348560564
Cost after iteration 2400: 0.048554785628770226
```



Добре, що ми створили векторизовану реалізацію. Інакше тренування могло б зайняти в 10 разів більше часу. Тепер ми можемо використовувати навчені параметри для класифікації зображень із набору даних. Щоб побачити свої прогнози на навчальних та тестових наборах, запустимо нступний код.

Приклад 5.5. Запуск двошарової моделі на тренувальних та тестових даних

```
predictions_train = predict(train_x, train_y, parameters)
```

Вихід:

```
Accuracy: 1.0
```

```
predictions_test = predict(test_x, test_y, parameters)
```

Вихід:

```
Accuracy: 0.72
```

Примітка

Ви можете помітити, що запуск моделі на меншій кількості ітерацій (скажімо, 1500) дає кращу точність у тестовому наборі. Це називається «раннє припинення». Рання зупинка – це спосіб запобігти переобладнанню. Наша двошарова нейронна мережа має кращу продуктивність (72%), ніж реалізація логістичної регресії (70%, завдання 2 тиждень). Давайте подивимося, що буде при застосучанні моделі з більшою кількістю шарів.

L-шарова нейронна мережа

Використаємо допоміжні функції, які ми реалізували раніше, щоб побудувати таку структуру: [*[LINEAR -> RELU]××(L-1) -> LINEAR -> SIGMOID*.

Функції, які нам можуть знадобитися, та їх вхідні дані:

```
def initialize_parameters_deep(layer_dims):
```

```
    ...
    return parameters
def L_model_forward(X, parameters):
    ...
    return AL, caches
def compute_cost(AL, Y):
    ...
    return cost
def L_model_backward(AL, Y, caches):
    ...
    return grads
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

```
### Константи ###
layers_dims = [12288, 20, 7, 5, 1] #  5-layer model
```

Приклад 5.6. Запуск багатошарової моделі на тренувальних та тестових даних

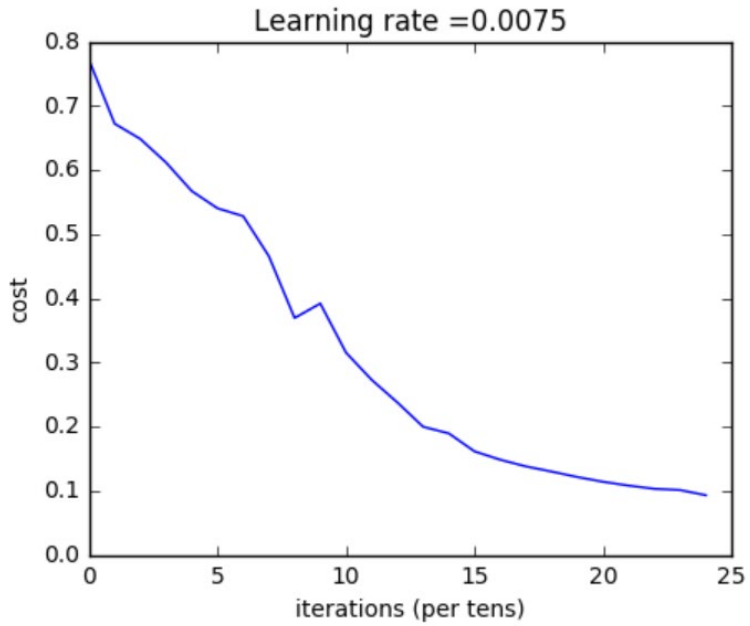
```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations
= 3000, print_cost=False):#lr was 0.009
    """
    Аргументи:
    X - вхіжні дані, розмірності (n_x, число прикладів)
    Y - правдиві "мітки" вектора (містять 0 якщо cat, 1 якщо non-cat),
розмірності (1, число прикладів)
    layers_dims - ромірність шару (n_x, n_h, n_y)
    num_iterations - число ітерацій оптимізаційного циклу
    learning_rate - швидкість навчання правила оновлення градієнтного спуску
    print_cost -- Якщо
встановлено значення True, це буде друкувати вартість кожні 100 ітерації

    Повернення:
    parameters - словник, що містить W1, W2, b1, і b2
    """

    np.random.seed(1)
    costs = []                         # відстежуйте вартість
    # Параметри ініціалізації.
    parameters = initialize_parameters_deep(layers_dims)

    # цикл(градієнтний спуск)
    for i in range(0, num_iterations):

        # Пряме поширення: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
                AL, caches = L_model_forward(X, parameters)

        # Обччислення вартості.

        cost = compute_cost(AL, Y)
        ###

        # Зворотне поширення.
        grads = L_model_backward(AL, Y, caches)

        # Оновлення параметрів.
        )
        parameters = update_parameters(parameters, grads, learning_rate)
```

```
        # Друкування вартості кодні 1000 ітерацій прикладів.
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # Графік вартості
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

Тепер будемо навчати модель як 4-шарову нейронну мережу. Запустимо клітинку нижче, щоб навчити свою модель. Вартість має зменшуватися на кожній ітерації. Для виконання 2500 ітерацій може знадобитися до 5 хвилин.

```
parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations =
2500, print_cost = True)
```

```
Cost after iteration 0: 0.771749
Cost after iteration 100: 0.672053
Cost after iteration 200: 0.648263
Cost after iteration 300: 0.611507
Cost after iteration 400: 0.567047
Cost after iteration 500: 0.540138
Cost after iteration 600: 0.527930
Cost after iteration 700: 0.465477
Cost after iteration 800: 0.369126
Cost after iteration 900: 0.391747
Cost after iteration 1000: 0.315187
Cost after iteration 1100: 0.272700
Cost after iteration 1200: 0.237419
Cost after iteration 1300: 0.199601
Cost after iteration 1400: 0.189263
Cost after iteration 1500: 0.161189
Cost after iteration 1600: 0.148214
Cost after iteration 1700: 0.137775
Cost after iteration 1800: 0.129740
Cost after iteration 1900: 0.121225
Cost after iteration 2000: 0.113821
Cost after iteration 2100: 0.107839
Cost after iteration 2200: 0.102855
Cost after iteration 2300: 0.100897
Cost after iteration 2400: 0.092878
```




```
pred_train = predict(train_x, train_y, parameters)
```

Вихід:

```
Accuracy: 0.985645933014
```

```
pred_test = predict(test_x, test_y, parameters)
```

Вихід:

```
Accuracy: 0.8
```

Наша 4-шарова нейронна мережа має кращу продуктивність (80%), ніж 2-шарова нейронна мережа (72%) на тому ж наборі тестів. Це хороша продуктивність для цього завдання.

Аналіз результатів

Давайте подивимося на деякі зображення, які модель L-шару класифікувала неправильно.

```
print_mislabeled_images(classes, test_x, test_y, pred_test)
```

Деякі типи зображень, які модель, як правило, погано показують, включають:

- Тіло кота в незвичній позі
- Кішка з'являється на тлі схожого кольору
- Незвичайний котячий забарвлення і вид
- Кут камери
- Яскравість картинки
- Варіація масштабу (кіт дуже великий або маленький на зображенні)

5.2. Порядок виконання роботи

5.2.1. Проаналізувати умову задачі.
5.2.2. Виконати задачу.
5.2.3. Результати роботи оформити протоколом.

5.3. Завдання для виконання

1. Побудувати глибоку багатошарову нейронну мережу за прикладом, наведеним у теоретичних відомостях для власного датасету(завантажити у папку «images» власні зображення).
2. Протестувати нейронні мережі з іншою кількістю шарів і порівняти результати.

5.4. Контрольні питання

1. Опишіть архітектуру моделі з двома шарами.
2. Чим архітектура багатошарової моделі відрізняється від архітектури двошарової моделі?
3. Опишіть структури моделей.
4. Яка модель і чому має кращу продуктиність?
5. Чому деякі зображення красифікуються неправильно?

Додаткові джерела інформації

1. A. Shelestov, B. Yailymov, H.Yailymova, L. Shumilo, M. Lavreniuk Advanced Method of Land Cover Classification Based on High Spatial Resolution Data and Convolutional Neural Network. Proceedings of International Conference on Applied Innovation in IT. — Volume 10, Issue 1 — pp. 125-132. doi:10.25673/76943.
2. Leonid Shumilo, Mykola Lavreniuk, Nataliia Kussul, Bella Shevchuk Automatic Deforestation Detection based on the Deep Learning in Ukraine. 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications. – 2021. – Cracow (virtual format). – P. 337 – 342. DOI: 10.1109/IDAACS53288.2021.9661008.
3. Leonid Shumilo, Nataliia Kussul, Mykola Lavreniuk U-net model for logging detection based on the Sentinel-1 and Sentinel-2 data. 2021 IEEE International Geoscience and Remote Sensing Symposium (IGARSS). – 2021. – Brussels (virtual format). – pp. 4680-4683, doi: 10.1109/IGARSS47720.2021.9554885.
4. M. Hosseini, H.McNairn, et all. A Comparison between Support Vector Machine and Water Cloud Model for Estimating Crop Leaf Area Index. Remote Sensing. – 2021. – Vol. 13, No. 7. – P. 1-20. DOI: 10.3390/rs13071348
5. Nataliia Kussul, Mykola Lavreniuk, Leonid Shumilo Deep recurrent neural network for crop classification task based on Sentinel-1 and Sentinel-2 imagery
6. IGARSS 2020 – 2020 IEEE International Geoscience and Remote Sensing Symposium. – Waikoloa, HI, USA. – P. 6914-6917. DOI: 10.1109/IGARSS39084.2020.9324699

7. Deininger, K., Ali, D. A., Kussul, N., Lavreniuk, M., & Nivievskyi, O. Using Machine Learning to Assess Yield Impacts of Crop Rotation: Combining Satellite and Statistical Data for Ukraine. Policy research working papers. – 2020. DOI: 10.1596/1813-9450-9306
8. Andrii Shelestov, Mykola Lavreniuk, Vladimir Vasiliev, Leonid Shumilo, Andrii Kolotii, Bohdan Yailymov, Nataliia Kussul, Hanna Yailymova Cloud Approach to Automated Crop Classification Using Sentinel-1 Imagery. IEEE Transactions on Big Data – 2020. – Vol. 6, No. 3. – 572-582 pp – DOI: 10.1109/TBDATA.2019.2940237