3. 파이썬과 벡터화

날짜	@September 13, 2023

▼ 목차

벡터화

1️⃣ Vectorization

로지스틱 회귀에서 $z = w^T x + b$을 계산하는 과정을 non-vectorized와 vectorized의 경우로 나누어 표현해 보자. (이때, w와 x는 열 벡터이며 $w \in \mathbb{R}^{n_x}$, $x \in \mathbb{R}^{n_x}$이다.)

Case 1: Non-vectorized

```
z = 0
for i in range(n_x) :
  z += w[i] * x[i]
z += b
```

Case 2: Vectorized

`np.dot(w, x)` 를 이용해 $w^T x$를 직접 계산함

```
z = np.dot(w, x) + b
```

- 파이썬 코드로 확인해보면 vectorized version이 약 300배 이상 빠르다는 것을 알 수 있다.

In [1]:
```
import numpy as np
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)
```

Vectorized

In [2]:
```
tic = time.time()
c = np.dot(a, b)
toc = time.time()

print(c)
print("vectorized version: " + str(1000 * (toc - tic)) + "ms")
```

```
250055.71843720783
vectorized version: 17.566680908203125ms
```

Non-vectorized

In [3]:
```
c = 0
tic = time.time()
for i in range(1000000) :
    c += a[i] * b[i]
toc = time.time()

print(c)
print("for loop: " + str(1000 * (toc - tic)) + "ms")
```

```
250055.7184372137
for loop: 609.0946197509766ms
```

결론적으로, for문 대신 내장 함수나 다른 방식을 이용할 수 있다면 되도록 for문을 피하는 것이 좋음

2️⃣ Kernelization

위와 같이 for문 대신 NumPy 내장 함수 `np.dot` 을 사용하는 것이 빠른 이유는 SIMD(Single Instruction Multiple Data)의 이용에서 찾을 수 있다.

SIMD?

> 🔷 하나의 명령어로 동일한 형태/구조의 여러 데이터를 동시에 처리하는 병렬 프로세서의 한 종류로, 벡터화 연산을 가능하게 함

- 데이터를 병렬화하는 것이 벡터화 연산을 가능하게 하고, 이를 통해 계산 능력을 향상시킬 수 있다는 정도로 이해함! (아래는 관련 사이트)

  [Python] Numpy가 빠른 이유-1편 (하드웨어 관점에서, SIMD)
  데보션 (DEVOCEAN) 기술 블로그 , 개발자 커뮤니티이자 내/외부 소통과 성장 플랫폼
  https://devocean.sk.com/blog/techBoardDetail.do?ID=163631

더 많은 벡터화 예제

1️⃣ In Neural Network Programming

이전에는 열 벡터의 곱을 계산하는 과정을 살펴봤다면, 이번에는 행렬과 벡터의 곱 $u = Av$를 계산하는 과정을 살펴보자.

Case 1: Non-vectorized

$u_i = \sum_i \sum_j A_{ij} v_j$

```
u = np.zeros((n, 1))
for i . . . .
  for j . . . .
    u[i] += A[i][j] * v[j]
```

Case 2: Vectorized

`np.dot(A, v)` 를 이용해 Av를 직접 계산함

```
u = np.dot(A, v)
```

2️⃣ In Vectors and Matrix Valued Functions

열 벡터 v의 모든 원소에 대하여 지수 연산을 하는 과정을 살펴보자.

– – – –

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_n \end{bmatrix} \longrightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \cdot \\ \cdot \\ \cdot \\ e^{v_n} \end{bmatrix}$$

Case 1: Non-vectorized

```
u = np.zeros((n, 1))
for i in range(n) :
  u[i] = math.exp(v[i])
```

Case 2: Vectorized

내장함수 `np.exp(v)` 를 이용하면 바로 계산됨

```
u = np.exp(v)
```

그 외 NumPy 내장함수

```
np.log(v)    # 원소의 로그값 계산
np.abs(v)    # 원소의 절댓값 계산
v ** 2    # 모든 원소를 제곱한 벡터 반환
1 / v     # 원소의 역수로 이루어진 벡터 반환
```

3️⃣ In Logistic Regression Derivatives

앞서 배운 로지스틱 회귀의 도함수를 구하는 의사 코드를 살펴보자.

Case 1: Non-vectorized

👾 J = 0, dw1 = 0, dw2 = 0, db = 0

for i = 1 to m :

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$

$\mathrm{d}z^{(i)} = a^{(i)}(1 - a^{(i)})$

dw1 += $x_1^{(i)} \mathrm{d}z^{(i)}$

dw2 += $x_2^{(i)} \mathrm{d}z^{(i)}$

db += $\mathrm{d}z^{(i)}$

J /= m, dw1 /= m, dw2 /= m, db /= m

- 위 코드에는 2개의 for문이 들어 있다. (초록 하이라이트 부분)
    - feature 수(n_x)만큼 dw가 존재하며, 위는 feature가 2개인 경우로 for문이 생략된 형태이다. (for j = 1 to n_x)

Case 2: Vectorized

위 코드의 2번째 for문을 지우고 dw를 벡터화시키는 과정을 살펴보자.

👾 J = 0, dw = np.zeros((n_x, 1)), db = 0

for i = 1 to m :

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$

$\mathrm{d}z^{(i)} = a^{(i)}(1 - a^{(i)})$

dw += $x^{(i)} \mathrm{d}z^{(i)}$

db += $\mathrm{d}z^{(i)}$

J /= m, dw /= m, db /= m

- dw1과 dw2를 `dw = np.zeros((n_x, 1))` 로 표현함으로써 for문을 하나 없앨 수 있다. (파란 하이라이트가 업데이트된 부분)

로지스틱 회귀의 벡터화

앞서 살펴본 코드에서 전체 훈련 세트에 대한 for문까지 제거하는 방법을 알아보자.

1️⃣ Forward Propagation

— 선형회귀식 — — 활성값 —

$$z^{(1)} = w^T x^{(1)} + b \qquad a^{(1)} = \sigma(z^{(1)})$$
$$z^{(2)} = w^T x^{(2)} + b \qquad a^{(2)} = \sigma(z^{(2)})$$
$$\vdots \qquad\qquad \vdots$$
$$z^{(m)} = w^T x^{(m)} + b \qquad a^{(m)} = \sigma(z^{(m)})$$

for문 사용

NumPy 내장함수 사용

- $Z = np.dot(w.T, X) + b$

* Broadcasting

상수 b가 $\underbrace{[b\ b \cdots b]}_{1 \times m}$ 되어 계산됨

- $\underset{n_x \times m}{X} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

- $Z = \underbrace{[z^{(1)}\ z^{(2)} \cdots z^{(m)}]}_{1 \times m} = w^T \underset{n_x \times m}{X} + \underbrace{[b\ b \cdots b]}_{1 \times m} = \underbrace{[w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w x^{(m)} + b]}_{1 \times m}$

- $A = \underbrace{[a^{(1)}\ a^{(2)} \cdots a^{(m)}]}_{1 \times m} = \sigma(Z)$

로지스틱 회귀의 경사 계산을 벡터화하기

1️⃣ Vectorizing Logistic Regression

$$dz^{(1)} = a^{(1)} - y^{(1)} \quad \cdots \quad dz^{(m)} = a^{(m)} - y^{(m)}$$

$$\Rightarrow A = [a^{(1)} \cdots a^{(m)}], \quad Y = [y^{(1)} \cdots y^{(m)}]$$

- $dZ = [dz^{(1)} \ \ dz^{(2)} \ \cdots \ dz^{(m)}]$

  $= [a^{(1)} - y^{(1)} \ \ a^{(2)} - y^{(2)} \ \cdots \ a^{(m)} - y^{(m)}]$

  $= A - Y$

- $db = \frac{1}{m}(dz^{(1)} + \cdots + dz^{(m)})$

  $= \frac{1}{m}\sum_{i=1}^{m} dz^{(i)} \Rightarrow \frac{1}{m}\text{np.sum}(dZ)$

- $dw = \frac{1}{m}(x^{(1)}dz^{(1)} + \cdots + x^{(m)}dz^{(m)})$

  $= \frac{1}{m}X(dZ)^T$

$$\underbrace{\begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix}}_{n_x \times m} \underbrace{\begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}}_{m \times 1}$$

$$= \underbrace{\left[x^{(1)}dz^{(1)} + \cdots + x^{(m)}dz^{(m)} \right]}_{n_x \times 1}$$

위 내용을 반영한 로지스틱 회귀 경사 하강법의 의사 코드는 다음과 같다.

> 👾 dw = np.zeros((n_x, 1)), db = 0
>
> Z = w^TX + b = np.dot(w.T, X) + b
>
> A = σ(Z)
>
> dZ = A - Y
>
> dw = $\frac{1}{m}$ X dZ^T
>
> db = $\frac{1}{m}$ np.sum(dZ)
>
> w := w - αdw
>
> b := b - αdb

- 경사 하강법을 여러 번 반복하는 경우에는 가장 바깥에 for문을 쓸 수밖에 없음

파이썬의 브로드캐스팅

1️⃣ Broadcasting Example

- e.g. 각 음식 100g에 들어 있는 탄수화물, 단백질, 지방의 칼로리량

$$\begin{array}{rc} & \begin{array}{cccc} \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \end{array} \\ \begin{array}{r} \text{Carb} \\ \text{Protein} \\ \text{Fat} \end{array} & \begin{bmatrix} 56.0 & 0.0 & 4.4 & 68.0 \\ 1.2 & 104.0 & 52.0 & 8.0 \\ 1.8 & 135.0 & 99.0 & 0.9 \end{bmatrix} \end{array}$$

파이썬 코드로 각 영양소가 주는 칼로리의 백분율을 계산해 보자.

```
In [1]: import numpy as np

        A = np.array([[56.0, 0.0, 4.4, 68.0],
                      [1.2, 104.0, 52.0, 8.0],
                      [1.8, 135.0, 99.0, 0.9]])

        print(A)

[[ 56.    0.    4.4  68. ]
 [  1.2 104.   52.    8. ]
 [  1.8 135.   99.    0.9]]
```

```
In [2]: cal = A.sum(axis=0)    # 열 더하기
        print(cal)

[ 59.   239.   155.4  76.9]
```

```
In [3]: percentage = 100 * A / cal.reshape(1, 4)
        print(percentage)

[[94.91525424  0.          2.83140283 88.42652796]
 [ 2.03389831 43.51464435 33.46203346 10.40312094]
 [ 3.05084746 56.48535565 63.70656371  1.17035111]]
```

- `cal = A.sum(axis=0)`
  - `axis=0` : 열 더하기 (세로로 더하기)
  - `axis=1` : 행 더하기 (가로로 더하기)
- `percentage = 100 * A / cal.reshape(1, 4)`
  - 상수와 행렬, 벡터의 연산이 파이썬 브로드캐스팅을 통해 이루어지고 있다.
  - `reshape` 함수: 행렬의 차원이 확실하지 않을 때 필요한 차원으로 바꾸어줄 수 있다.
    - 위 코드의 경우 `cal` 의 차원이 (1, 4)임이 확실하므로 굳이 쓰지 않아도 됨!

More Examples

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} \cancel{100} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

$$\underset{(m,n)\ (2,3)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(1,n) \rightsquigarrow (m,n)\quad (2,3)}{\begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(m,1) \rightsquigarrow (m,n)}{\begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix}} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$

2️⃣ General Principle

$$\underset{\text{matrix}}{(m,n)} \qquad \begin{matrix}+\\-\\ *\\ /\end{matrix} \qquad \begin{matrix}(1,n) \rightsquigarrow (m,n)\\ (m,1) \rightsquigarrow (m,n)\end{matrix}$$

$$(m,1) \quad + \quad \mathbb{R}$$

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + 100 = \begin{bmatrix}101 & 102 & 103\end{bmatrix}$$

Matlab/Octave: bsxfun

파이썬과 NumPy 벡터

💠 랭크 1 배열(rank 1 array)은 벡터와는 별개의 자료구조로 결과가 직관적이지 않게 하므로, 그 대신 행 벡터 또는 열 벡터를 사용하는 것이 좋음

파이썬 코드를 통해 알아보자.

In [1]:
```
import numpy as np
```

Rank 1 Array

In [2]:
```
# 가우시안 분포를 따르는 변숫값 5개를 배열 a에 저장
a = np.random.randn(5)
print(a)
```

```
[ 0.48691983  1.02068969  0.38199277 -0.26611354  1.17556678]
```

이때 a는 행 벡터도 열 벡터도 아닌 rank 1 배열이 되어, 전치 또는 내적 연산이 제대로 이루어지지 않는다.

In [3]:
```
print(a.shape)
```

```
(5,)
```

In [4]:
```
print(a.T)
```

```
[ 0.48691983  1.02068969  0.38199277 -0.26611354  1.17556678]
```

In [5]:
```
print(np.dot(a, a.T))
```

```
2.8775904899838127
```

Vector

따라서 다음과 같이 설정해야 한다.

In [6]:
```
a = np.random.randn(5, 1)    # 열 벡터
print(a)
```

```
[[-1.81816688]
 [ 0.52605925]
 [-0.22418967]
 [ 0.41807433]
 [-0.01811298]]
```

In [7]:
```
print(a.T)
```

```
[[-1.81816688  0.52605925 -0.22418967  0.41807433 -0.01811298]]
```

In [8]:
```
print(np.dot(a, a.T))
```

```
[[ 3.30573079e+00 -9.56463494e-01  4.07614237e-01 -7.60128906e-01
   3.29324150e-02]
 [-9.56463494e-01  2.76738329e-01 -1.17937050e-01  2.19931868e-01
  -9.52849907e-03]
 [ 4.07614237e-01 -1.17937050e-01  5.02610094e-02 -9.37279481e-02
   4.06074241e-03]
 [-7.60128906e-01  2.19931868e-01 -9.37279481e-02  1.74786149e-01
  -7.57257084e-03]
 [ 3.29324150e-02 -9.52849907e-03  4.06074241e-03 -7.57257084e-03
   3.28079940e-04]]
```

전치와 내적 연산이 잘 동작함을 확인할 수 있다.

다음과 같이 `assert` 함수를 통해 차원을 확인하거나, `reshape` 함수를 이용해 rank 1 배열을 행 또는 열 벡터로 바꿔줄 수 있다.

assert

In [9]:
```
a = np.random.randn(5)
assert(a.shape == (5, 1))    # AssertionError!
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [9], in <cell line: 2>()
      1 a = np.random.randn(5)
----> 2 assert(a.shape == (5, 1))

AssertionError:
```

In [10]:
```
a = np.random.randn(5, 1)
assert(a.shape == (5, 1))
```

reshape

In [11]:
```
a = np.random.randn(5)
print(a)
```

```
[ 0.53824925  0.05645422  0.40674856 -1.01842304 -0.55106096]
```

In [12]:
```
a = a.reshape(5, 1)
print(a)
```

```
[[ 0.53824925]
 [ 0.05645422]
 [ 0.40674856]
 [-1.01842304]
 [-0.55106096]]
```

로지스틱 회귀의 비용함수 설명

$$\hat{y} = \sigma(w^T + b)$$
$$where \ \ \sigma(z) = \frac{1}{1+e^{-z}}$$
$$Interpret \ \ \hat{y} = P(y=1|x)$$

- $\hat{y}$는 주어진 x에 대해 y가 1일 확률이다.

$$If \ \ y = 1: \ \ p(y|x) = \hat{y}$$
$$If \ \ y = 0: \ \ p(y|x) = 1 - \hat{y}$$
$$\Longrightarrow \ p(y|x) = \hat{y}^y(1-\hat{y})^{1-y}$$

- 이때 주어진 x에 대해 y가 0일 확률은 $1-\hat{y}$로 나타낼 수 있다.
- 두 식을 하나로 합친 것이 마지막 줄인데, $y=0$과 $y=1$을 대입해 보면 원하는 결과가 나옴을 알 수 있다.

$$\begin{aligned}\log p(y|x) &= \log(\hat{y}^y(1-\hat{y})^{1-y}) \\ &= y\log\hat{y} + (1-y)\log(1-\hat{y}) \\ &= -L(\hat{y}, y)\end{aligned}$$

- 로그함수는 강한 단조 증가 함수로, $\log p(y|x)$를 최대화하는 것은 $p(y|x)$를 최대화하는 것과 같다.
- 식을 풀어 보면 로지스틱 회귀 손실함수의 음수 값이 도출되는데, 이는 우도(확률)를 최대화하는 것이 손실함수 값(비용)을 최소화하는 것과 같기 때문이다.

$$\begin{aligned}&\log p(labels\ in\ training\ set) \\ &= \log\prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\ &= \sum_{i=1}^{m}\log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})\end{aligned}$$

- 모든 training sample이 독립동일분포라고 가정했을 때 전체 sample에 대한 확률은 각 확률의 곱으로 표현되며, 최대 우도 추정을 통해 training set의 타깃 확률을 최대화하는 변수를 찾게 된다.
- 로그를 씌워보면, 로지스틱 회귀 모델의 최대 우도 추정은 결국 비용함수 $J(w, b)$를 최소화하는 것과 같음을 알 수 있다.

출석퀴즈 오답노트

▼ 2. 다음 중 뉴런이 계산하는 것은?

- 선형함수(z = Wx + b)를 계산한 후 활성화 함수 (O)
- 입력 x를 선형적으로 확장하는 함수 g(Wx + b) 계산 (X)

▼ 7, 8. 아래 상황에서 `c` 의 shape는?

```
a = np.random.randn(4, 3)
b = np.random.randn(3, 2)
c = a * b

a = np.random.randn(3, 3)
b = np.random.randn(3, 1)
c = a * b
```

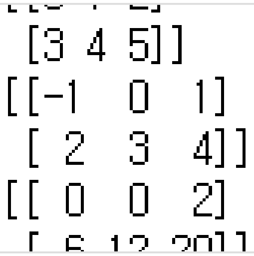
- 두 경우 모두 사이즈가 매치되지 않기 때문에 에러가 난다!

- 행렬 곱셈 관련 연산을 진행할 때 `*` , `.dot` , `@` 를 헷갈려서는 안 된다.

[Numpy]행렬곱(@)과 내적(dot) 그리고 별연산(*)

numpy array의 곱연산에 대해서 알아보도록 하겠습니다. 곱연산에는 총 세가지 연산이 있는데요. 선형대수에서 배우는 행렬의 곱을 하는 행렬곱(@)과 내적, 스칼라

https://seong6496.tistory.com/110



Operator	Shape	비고
`*` 연산	(n, m) * (n, m) = (n, m) — Broadcasting— (1, m) * (n, m) = (n, m) (m, 1) * (m, n) = (m, n) (m, n) * (m, 1) = (m, n) (n, m) * (1, m) = (n, m)	같은 shape인 경우에만 연산이 가능하며, 행렬과 벡터의 연산 경우 브로드캐스팅이 이루어짐
내적(`.dot`)	(n, m).dot((m, k)) = (n, k)	
행렬곱(`@`)	(n, m) @ (m, k) = (n, k)	2차원 행렬의 경우 내적과 행렬곱의 결과가 같아 어떤 것을 사용해도 괜찮음