2주차_파이썬과 벡터화

link	https://akylys.notion.site/2-_-90e9f25e25544b3db0bba28b7b8ca74f?pvs=4

💡 벡터화 : 코드에서 *for문*을 없애는 것

딥러닝은 많은 데이터를 요구하기에, 빠르게 많은 데이터를 처리하는 것이 중요하다. 그렇기 때문에, 벡터화를 통해 빠르게 많은 데이터를 처리할 수 있게 하는 것이다.

벡터화란?

아래의 코드는 벡터화와 for loop의 속도 차이를 볼 수 있는 코드들이다.

```
import numpy as np
import time

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

우선 2개의 랜덤 array들을 만들어준 후, 각각의 값들을 더하기 위해 시간이 얼마나 소요되는지를 볼 수 있다.

```
tic = time.time()
c = np.dot(a,b)
toc = time.time()

print(c)
print("Vectorized Version: " + str(1000 * (toc-tic)) + 'ms')

# 평균적으로 1.5~2.0 ms 이 소요되었다.
# 내 컴퓨터에서는 약 1.0~1.5 ms 정도 소요되었다.
```

```
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')
```

```
# 평균적으로 400~500 ms 이 소요되었다.
# 내 컴퓨터에서는 약 200~300 ms 정도 소요되었다.
```

위 코드들을 통해 터화를 한 코드가 약 **300배 더** 오래걸린다는 것을 알 수 있다.

▼ *code screenshot*

```
[6]
1 import numpy as np
2 import time
3
4 a = np.random.rand(1000000)
5 b = np.random.rand(1000000)
```

```
[14]
1 tic = time.time()
2 c = np.dot(a,b)
3 toc = time.time()
4
5 print(c)
6 print("Vectorized Version: " + str(1000 * (toc-tic)) + 'ms')
```

```
249851.08819403293
Vectorized Version: 1.2803077697753906ms
```

```
[22]
1 tic = time.time()
2 for i in range(1000000):
3     c += a[i] * b[i]
4 toc = time.time()
5
6 print(c)
7 print("For Loop: " + str(1000 * (toc-tic)) + 'ms')
8
9 # 평균적으로 400~500 ms 이 소요되었다.
```

```
2248659.793746559
For Loop: 266.0660743713379ms
```

- 대규모 딥러닝 구현은 대부분 GPU에서 실행된다.
- GPU와 CPU 모두 SIMD(Simple Instruction Multiple Data)라는 병렬 명령어를 갖고 있다.
- SIMD를 통해 for문이 필요없는 파이썬이나 라이브러리 자체의 함수를 사용하면 병렬화의 장점을 통해 계산을 훨씬 더 빠르게 할 수 있게 한다.

SIMD는 병렬 프로세서의 한 종류로, 하나의 명령어로 여러 개의 값을 동시에 계산하는 방식입니다.

For loop 벡터화

아래는 로지스틱 회귀의 알고리즘이다. 2개의 for loop이 들어있고, 이 중 2번째 loop을 벡터화시켜보자.

```
J = 0; dw1 = 0; dw2 = 0; db = 0

# for loop - 1
for i in range(m):
  z[i] = w.T * x[i] + b
  a[i] = sigma(z[i])

  J += -( y[i]log(a[i]) + (1-y[i])log(1-a[i]) )
  dz[i] = a[i] - y[i]

  # for loop - 2
  dw1 += x1[i] * dz[i]
  dw2 += x2[i] * dz[i]

J = J/m
dw1 = dw1/m ; dw2 = dw2/m
db = db/m
```

아래는 for loop #2에 관련된 코드들을 벡터화시킨 코드다.

```
dw = np.zeors((n_x,1))

dw += x[i] * dz[i]
dw /= m
```

로지스틱 회귀의 벡터화

정방향 전파

각각의 데이터의 z값과 a값을 계산하기 위해 모든 데이터를 행렬의 형태로 만들어준다.

$w^T = \begin{bmatrix} w^{(1)} & w^{(2)} & ... & w^{(m)} \end{bmatrix}$

$X = \begin{bmatrix} x^{(1)} & x^{(2)} & ... & x^{(m)} \\ \end{bmatrix}$, where X is a (N_x, m) matrix

$b = \begin{bmatrix} b & b & ... & b \end{bmatrix}$, where b is a $(1 \times m)$ matrix

이제 $z = w^T x + b$ 를 행렬곱으로 표현할 수 있다. 만약 행렬곱을 사용해 계산한다면, for loop을 사용할 필요가 없다.

$Z = w^T X + b$ 를 파이썬으로 구현한다면 `Z = np.dot(w.T, x) + b` 가 된다.

여기서 `b`는 행렬이 아닌 하나의 상수인데, 이 수식에서 파이썬이 자동으로 하나의 행렬로 만들어준다 (= broadcasting 브로드캐스팅).

경사하강법

정방향 전파를 했던 것 처럼, 경사하강법을 계산할 때에도 데이터를 행렬으로 변환시켜 계산해준다.

$$dZ = \begin{bmatrix} dz^{(1)} & dz^{(2)} & \dots & dz^{(m)} \end{bmatrix}$$
$$A = \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix}$$
$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix}$$
$$\rightarrow dZ = A - Y$$

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$
$$db = \frac{1}{m} \texttt{np.sum}(dz)$$

$$dw = \frac{1}{m} X dz^T$$
$$= \frac{1}{m} \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$
$dw = \frac{1}{m} \left[x^{(1)} dz^{(1)} + x^{(2)} dz^{(2)} + \cdots + x^{(m)} dz^{(m)} \right]$, where it is a $(n \times 1)$ matrix

원래 알고리즘은 아래와 같았다 (위에서 2번째 for loop을 벡터화 시킨 것 포함):

```
J = 0; dw = np.zeors((n_x,1)); db = 0

for i in range(m):
  z[i] = w.T * x[i] + b
  a[i] = sigma(z[i])
```

```
  J += -( y[i]log(a[i]) + (1-y[i])log(1-a[i]) )
  dz[i] = a[i] - y[i]

  dw += x[i] * dz[i] # 위에서 바꿔준 것

J = J/m; dw /= m; db = db/m
```

벡터화를 한다면 이러한 형태로 다시 표현할 수 있다.

```
Z = np.dot(w.T, X) + b    # w.T * X + b
A = sigma(Z)
dz = A - Y
dw = 1/m * X * dZ.T
db = 1/m np.sum(dZ)

w := w-(alpha)dw
w := w-(alpha)db
```

** 위 코드블럭들은 파이썬의 형태가 아닌 pseudocode*

만약 경사하강법을 여러번 시행하고 싶다면, 이 코드를 for loop을 통해 시행해야한다. (이 for loop을 없앨 방법을 없다)

파이썬

Broadcasting

> 💡 The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. It does this without making needless copies of data and usually leads to efficient algorithm implementations. *— 출처 : Numpy*

```
import numpy as np
A = np.array([[56., 0, 4.4, 68],
              [1.2, 104, 52, 8],
              [1.8, 135, 99, 0.9]])

print(A)
```

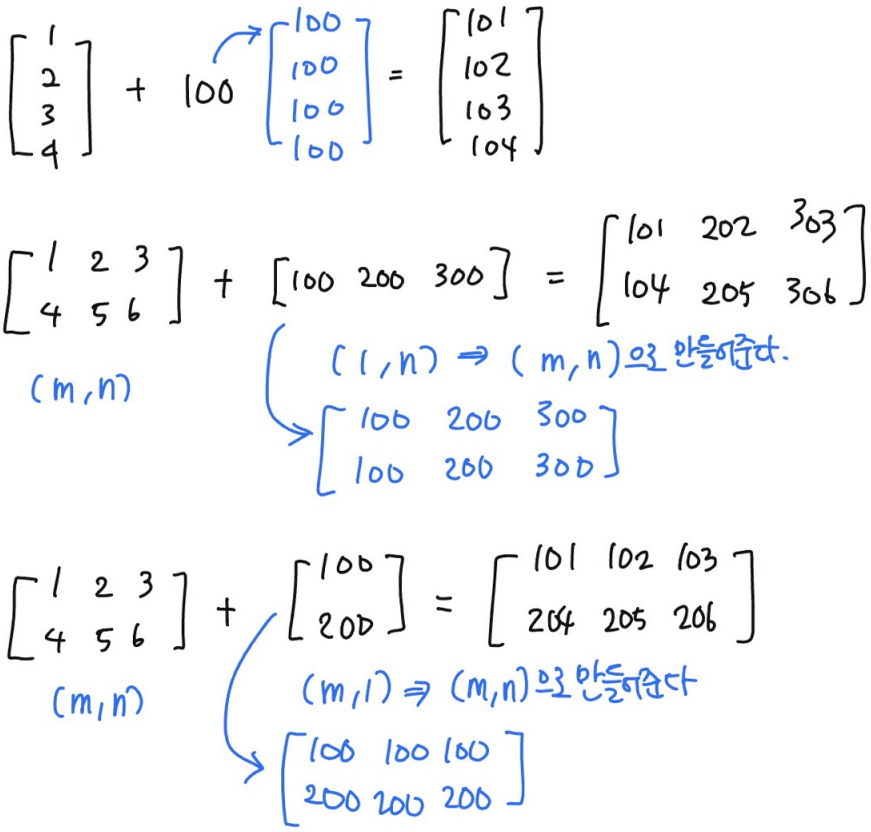
```
cal = A.sum(axis=0) # axis=0 means to sum vertically (새로)
print(cal)

percentage = 100 *A / cal.reshape(1,4)  # reshape이 꼭 필요하지 않다.
                                        # 이미 cal을 만들 때 (1,4)형태로 되어있었다.
print(percentage)                       # 하지만 변수의 shape이 확실하지 않을 때는
                                        # reshape을 통해 확실히 하는 것도 좋은 방법
```

▼ *code screenshot*

```
[2]  1 A = np.array([[56., 0, 4.4, 68],
     2              [1.2, 104, 52, 8],
     3              [1.8, 135, 99, 0.9]])
     4
     5 print(A)
```

```
[8]  1 cal = A.sum(axis=0) # axis=0 means to sum vertically (새로)
     2 print(cal)
     3
     4 percentage = 100 *A / cal.reshape(1,4)  # reshape이 꼭 필요하지 않다.
     5                                         # 이미 cal을 만들 때 (1,4)형태로 되어있었다.
     6 print(percentage)                       # 하지만 변수의 shape이 확실하지 않을 때는
     7                                         # reshape을 통해 확실히 하는 것도 좋은 방법
```

```
[ 59.  239.  155.4  76.9]
[[94.91525424  0.          2.83140283 88.42652796]
 [ 2.03389831 43.51464435 33.46203346 10.40312094]
 [ 3.05084746 56.48535565 63.70656371  1.17035111]]
```



Numpy

파이썬이 익숙하지 않다면 다양한 오류들을 겪을 수 있다.

```
a = np.random.randn(5)
print(a)
# [ 0.14744071 -1.95836193  1.26972689 -1.08794637  0.57066189]

a.shape # rank 1 array
# (5,)

print(a.T)
# [ 0.14744071 -1.95836193  1.26972689 -1.08794637  0.57066189]

print(np.dot(a,a.T)) # 행렬이 아닌 상수가 나온다.
# 6.978408890578056
```

이렇게 array를 만들면, 예상과는 다른 결과들이 나올 수 있다. 이런 오류들을 피하기 위해, array를 만들 때, `(5)`가 아닌 `(5,1)`을 넣어준다.

```
a = np.random.randn(5,1)
print(a)
```

```
# [[ 1.16849245]
# [ 1.70998856]
# [ 0.64074319]
# [-0.3727467 ]
# [-0.9635918 ]]

a.shape # rank 1 array
# (5, 1)

print(a.T)
# [[ 1.16849245  1.70998856  0.64074319 -0.3727467  -0.9635918 ]]

print(np.dot(a,a.T)) # 행렬이 아닌 상수가 나온다.
# [[ 1.36537461  1.99810872  0.74870358 -0.4355517  -1.12594975]
# [ 1.99810872  2.92406086  1.09566352 -0.63739259 -1.64773095]
# [ 0.74870358  1.09566352  0.41055184 -0.23883491 -0.61741489]
# [-0.4355517  -0.63739259 -0.23883491  0.1389401   0.35917566]
# [-1.12594975 -1.64773095 -0.61741489  0.35917566  0.92850916]]
```

위에 첫 코드가 오류가 났던 이유는, shape이 `5,`인 array는 랭크1 배열 (rank 1 array)이기 때문이다.

- 랭크1 배열은 행 벡터나 열 벡터가 아닌 다른 자료구조이다. 그렇기 때문에 결과가 직관적이지 않게 되는 것이다.
- 프로그래밍 예제나 신경망을 구현할 때 이 랭크1 배열을 사용하는 것을 삼가는 것이 좋다.
- 만약 랭크1 배열을 만들게 된다면, `reshape` 함수를 활용해 `(5,1)`으로 바꿔주는 것이 좋다.

▼ *code screenshot*

```
[9]  1 a = np.random.randn(5)
     2 print(a)

[ 0.14744071 -1.95836193  1.26972689 -1.08794637  0.57066189]

[10]  1 a.shape # rank 1 array

(5,)

[11]  1 print(a.T)

[ 0.14744071 -1.95836193  1.26972689 -1.08794637  0.57066189]

[13]  1 print(np.dot(a,a.T)) # 행렬이 아닌 상수가 나온다.

6.978408890578056

[19]  1 a = np.random.randn(5,1)
      2 print(a)

[[ 1.16849245]
 [ 1.70998856]
 [ 0.64074319]
 [-0.3727467 ]
 [-0.9635918 ]]

[20]  1 a.shape

(5, 1)

[21]  1 print(a.T) # [[ 가 2개인 것을 확인할 수 있다.

[[ 1.16849245  1.70998856  0.64074319 -0.3727467  -0.9635918 ]]

[22]  1 print(np.dot(a,a.T))

[[ 1.36537461  1.99810872  0.74870358 -0.4355517  -1.12594975]
 [ 1.99810872  2.92406086  1.09566352 -0.63739259 -1.64773095]
 [ 0.74870358  1.09566352  0.41055184 -0.23883491 -0.61741489]
 [-0.4355517  -0.63739259 -0.23883491  0.1389401   0.35917566]
 [-1.12594975 -1.64773095 -0.61741489  0.35917566  0.92850916]]
```

각 배열의 shape을 확실히 하기 위해서 `assert` 함수를 사용해주면 좋다.

가정 설정문(assert)은 뒤의 조건이 True가 아니면 AssertError을 발생하게 하는 함수다.

assert는 개발자가 프로그램을 만드는 과정에 관여한다. 원하는 조건의 변수 값을 보증받을 때까지 assert로 테스트 할 수 있다.

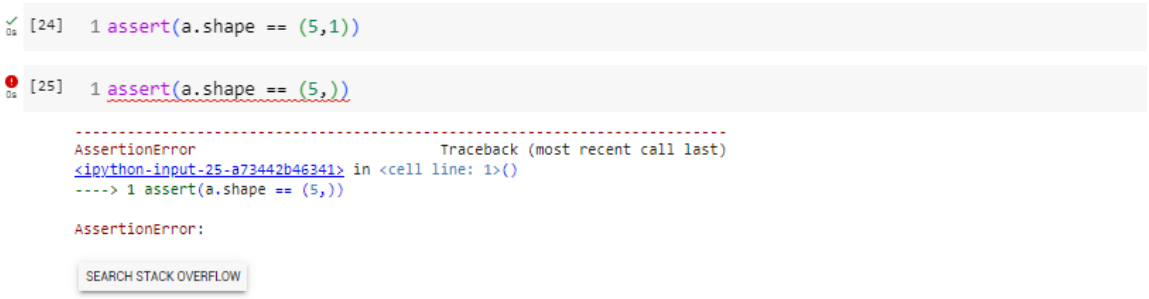
이는 단순히 에러를 찾는것이 아니라 값을 보증하기 위해 사용된다.

예를 들어 함수의 입력 값이 어떤 조건의 참임을 보증하기 위해 사용할 수 있고 함수의 반환 값이 어떤 조건에 만족하도록 만들 수 있다. 혹은 변수 값이 변하는 과정에서 특정 부분은 반드시 어떤 영역에 속하는 것을 보증하기 위해 가정 설정문을 통해 확인 할 수도 있다.

이처럼 실수를 가정해 값을 보증하는 방식으로 코딩 하기 때문에 이를 '방어적 프로그래밍'이라 부른다. *— 출처 : 위키독스*

```
assert(a.shape == (5,1))

assert(a.shape == (5,))
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-25-a73442b46341> in <cell line: 1>()
----> 1 assert(a.shape == (5,))

AssertionError:
```

▼ *code screenshot*



로지스틱 회귀의 비용함수

손실함수

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

$\hat{y} = P(y=1|x)$

- If $y=1$: $P(y|x) = \hat{y}$
- If $y=0$: $P(y|x) = 1-\hat{y}$
  - 이 두 개의 가정은 결국 $P(y|x)$을 의미한다.

위 두 개의 등식을 합치면, 아래의 식으로 표현할 수 있다.

$P(y|x)$의 정의

$$P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$$

- $y=1$: $(1-\hat{y})$은 0승이 되기 때문에 1이 되어 사라지고 $\hat{y}$만 남는다

- $y = 0$: $\hat{y}$은 0승이 되기 때문에 1이 되어 사라지고 $(1 - \hat{y})$만 남는다

또한 로그함수는 **강한 단조 증가 함수(strictly monotonically increasing function)**이기 때문에 $\log P(y|x)$를 최대화 하는 것은 $P(y|x)$를 최대화 하는 것과 같은 결과를 준다.

$$\begin{aligned}\log p(y|x) &= \log \hat{y}^{y}(1-\hat{y})^{(1-y)} \\ &= y \log \hat{y} + (1-y)\log(1-\hat{y}) \\ &= -\mathcal{L}(\hat{y}-y) \text{ , where } \mathcal{L} \text{ is the loss function.}\end{aligned}$$

💡 마지막 줄에 $-$가 있는 이유는, 로그함수를 최대화하기 위해서는 비용함수를 최소화해야하기 때문이다.

비용함수

- 훈련 샘플이 `iid`, 즉 독립동일분포임을 가정해야한다.

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

💡 **최대 우도 추정의 원칙 (Maximum Likelihood Estimation)**
$-\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$를 최대화 시키는 매개변수을 찾는 것

Cost : $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

- 최소화하는 것이기 때문에 $-$를 없애고, 스케일을 맞추기 위해 편의상 $\frac{1}{m}$이라는 비례 계수를 추가한 것.

- 즉, 훈련샘플이 `iid` *(identical and identically distributed)*일 때, 비용함수인 $J(w,b)$를 최소화하므로, 로지스틱 회귀 모델의 최대 우도 추청을 한 것이다.