# 2주차

# 3. 파이썬과 벡터화

## Vectorization

### 벡터화

- 코드에서 for문을 없애는 방법
- 로지스틱 회귀에서 $w^Tx$를 구하는데 for문 사용 → 계산 느림
- 벡터화로 $w^Tx$ 직접 구현 `z = np.dot(w,x) + b`

### 시간 비교

```
import time

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

tic=time.time() #현재 시간
c=np.dot(a,b)
toc=time.time()

print(c)
print("Vectorized version:" +str(1000*(toc-tic))+"ms") #밀리초 단위로 표현
```

- 결과: Vectorized version:2.147674560546875ms

```
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")
```

- 결과: for loop:473.3731746673584ms

→ Vectorized version이 훨씬 빠르다

- 대규모 딥러닝 구현 GPU 사용 방금 cpu

- SIMD (single instruction mutiple data) : 병렬 명령어, numpy가 병렬화의 장점을 통해 계산을 훨씬 빠르게 할 수 있게 해줌
- 가능하면 for문은 사용 x

# More vectorization examples

## 행렬 곱 u=Av 계산할때

- 벡터화하지 않은 구현

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

  - 2중 for문 사용

- 벡터화한 구현

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

## 행렬 지수 연산

$$v = \begin{pmatrix} v1 \\ ... \\ vn \end{pmatrix} \rightarrow u = \begin{pmatrix} e^{v1} \\ ... \\ e^{vn} \end{pmatrix}$$

- 벡터화하지 않은 구현

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

- 벡터화한 구현

```
import numpy as np
u=np.exp(v)
```

  - np.log(v) : 원소의 로그값 구함
  - np.abs(v) : 절대값

- np.maximum(v,0) : v와 0중 더 큰 값 반환
- v**2 : 원소 제곱, 1/v : 원소의 역수

### 로지스틱 회귀

$$
\begin{aligned}
&J = 0,\ ^{1}\ dw1 = 0,\ dw2 = 0,\ db = 0 \\
&\text{for } i = 1 \text{ to } m: \\
&\quad z^{(i)} = w^T x^{(i)} + b \\
&\quad a^{(i)} = \sigma(z^{(i)}) \\
&\quad J \mathrel{+}= -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})] \\
&\quad dz^{(i)} = a^{(i)}(1 - a^{(i)}) \\
&2\ \ dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)} \\
&\quad dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)} \\
&\quad db \mathrel{+}= dz^{(i)} \\
&J = J/m,\ ^{3}\ dw_1 = dw_1/m,\ dw_2 = dw_2/m,\ db = db/m
\end{aligned}
$$

- 두번째 for문 제거
- 1 : dw를 벡터로 만듬 → dw=np.zeros((n_x,1))
- 2 : dw+=x^(i)*dz^(i)
- 3 : dw/=m

## Vectorizing Logistic Regression

- 정방향 전파 : m개의 훈련 샘플에 대해 예측값 계산

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

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

- 벡터화를 이용해 for문 사용 없이 계산
  - `Z = np.dot(np.transpose(W), X) + b`
  - Broadcasting : b는 (1,1) 행렬인 실수이지만 자동으로 (1,m) 행렬로 broadcasting 되어 오류 발생 x
- 모든 z를 동시에 계산하고 sigmoid 함수 구현으로 모든 a도 동시에 계산

# Vectorizing Logistic Regression's Gradient Computation

## 역방향 전파 벡터화

- 역방향 전파

  $$dz^{(i)} = a^{(i)} - y^{(i)}$$

- m개의 훈련 샘플에 대한 for문이 남음
  - db = 1/m*np.sum(dz)
  - dw =1/m*X*dz^T

## 로지스틱 회귀 벡터화

$$
\begin{aligned}
Z &= w^T X + b \\
&= np.dot(w.T, X) + b \\
A &= \sigma(Z) \\
dZ &= A - Y \\
dw &= \frac{1}{m} X dZ^T \\
db &= \frac{1}{m} np.sum(dZ)
\end{aligned}
$$

$$
\begin{aligned}
w &:= w - \alpha dw \\
b &:= b - \alpha db
\end{aligned}
$$

- 경사하강법을 여러번 반복하고 싶다면 for문 필요

# Broadcasting in Python

```
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]])
```

```
cal=A.sum(axis=0)
percentage = 100*A/cal.reshape(1,4)
```

- axis = 0 : 파이썬에게 세로로 더하라고 알려줌
  - 가로축 : axis=0
- (3,4) 행렬 A를 (1,4) 행렬로 나눔
  - 코드 첫줄이 실행된 이후 변수 cal은 이미 (1,4)행렬
- 행렬 차원 확실 x → reshape 사용
  - 상수 시간 → 호출 저렴

> 💡 (m,n) +-*/ (1,n) → (m,n)
>                 (m,1) → (m,n)
> [ 1 2 3 ] + 100 = [ 101 102 103 ]

## A note on python/numpy vectors

- `a=np.random.randn(5)`
  - a.shape = (5,)
  - rank 1 배열
  - 열 벡터, 행 벡터 둘 다 x
  - `a = a.reshape((5,1))`로 (5,1)배열로 변경해서 사용을 권장
- `a=np.random.randn(5,1)`
  - 열 벡터
- `a=np.random.randn(1,5)`
  - 행 벡터
- `assert(a.shape==(5,1))`
  - 행렬과 배열의 차원을 확인

> 💡 rank 1 배열을 사용하지 말고 열 벡터 (n,1) 행렬 / 행 벡터 (1,n) 행렬을 사용하자

# Explanation of logistic regression cost function

- 로지스틱 회귀 : $\hat{y} = \sigma(w^T x + b)$ when $\sigma(z) = \frac{1}{1+e^{-z}}$
- $\hat{y} = P(y = 1|x)$
  - if y=1 : $P(y|x) = \hat{y}$
  - if y=0 : $P(y|x) = 1 - \hat{y}$
- 위 두가지 경우를 하나의 수식으로 나타내면

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

  - if y=1 : $P(y|x) = \hat{y}$
  - if y=0 : $P(y|x) = 1 - \hat{y}$
- 로그 함수는 강한 단조 증가 함수 → $logP(y|x)$를 최대화 = $P(y|x)$최대화

$$logP(y|x) = log\hat{y}^y (1 - \hat{y})^{(1-y)} = ylog\hat{y} + (1 - y)log(1 - \hat{y}) = -L(\hat{y}, y)$$

  - 손실함수의 음수와 동일 → 로지스틱 회귀에서는 손실 함수를 최소화하고 싶어하기 때문
    - 손실 함수 최소화 = 확률의 로그값 최대화

## m개 훈련 세트 손실함수

- 손실함수
  - 훈련 샘플들이 독립동일분포라고 가정
  - 전체 샘플에 대한 확률은 각 확률의 곱

$$logP(labelsintrainingset) = log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

- 최대 우도 추정 → 이 값을 최대화하는 매개 변수 찾아야함
  - 비용은 최소화
- 비용 함수

$$J(w, b) = -logP(labelsintrainingset) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$