[Week2]_문가을

2. **신경망과 로지스틱 회귀**

계산 그래프(Computation Graph)

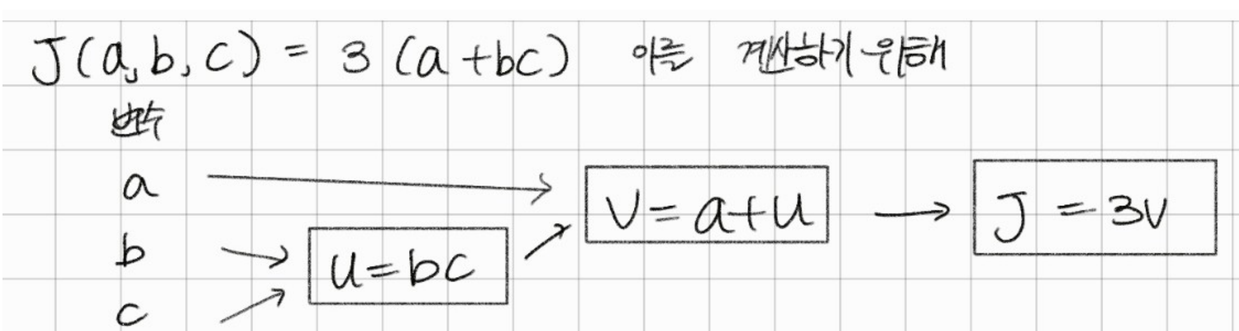
신경망을 계산하기 위해서는

- 전방향 패스 / 전방향 전파 : 신경망의 출력값을 계산하고,
- 역방향 패스 / 역방향 전파 : 경사나 도함수 계산

계산 그래프는 특정한 출력값 변수 최적화하고 싶을 때 유용

→ 로지스틱 회귀 같은 경우 비용함수를 최적화

- 계산 그래프 예시 :



왼쪽에서 오른쪽의 패스로 J값 계산

계산 그래프로 미분하기

미분의 연쇄법칙 (Chain rule)을 이용해 도함수 구하기.

→ 역방향으로 계산

위 계산 그래프 예시에서 dJ / da를 구하기

a → v → J

: a가 조금 증가했을 때 v가 변화하고, v의 변화는 J를 증가시킴.

따라서 a가 변화했을 때 J의 변화량 = (a를 밀었을 때 v의 변화량)*(v를 밀었을 때 J의 변화량)

$$\frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da}$$

코드를 작성할 때 표기법 :

최종 변수를 final output var, 미분하려고 하는 변수를 var라고 정의 할 때,

$$\frac{d\ Final\ output\ var}{d\ var} = d\ var$$

라고 간단히 표기함.

예) dJ / da → da라고 표기

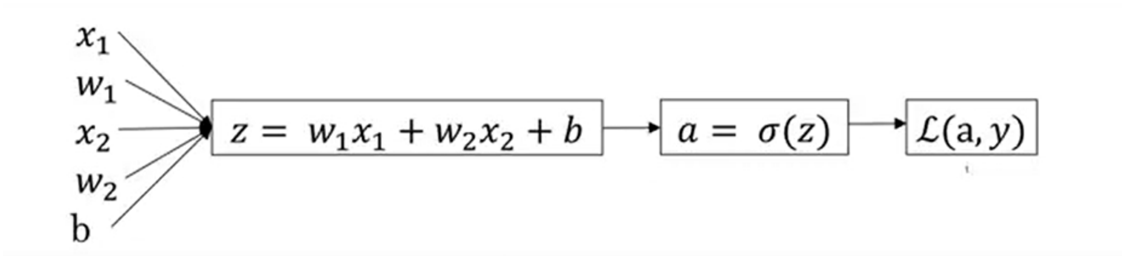
로지스틱 회귀의 경사하강법 (단일 샘플)

로지스틱 회귀에서 목적은 매개변수 w와 b를 변경해서 손실을 줄이는 것임.

→ 손실함수에 대한 도함수를 구해야 함.

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

이를 계산 그래프로 나타내면



$$da = \frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\begin{aligned} dz &= \frac{dL(a,y)}{dz} \\ &= \frac{dL}{da}\frac{da}{dz} \\ &= (-\frac{y}{a} + \frac{1-y}{1-a})(a(1-a)) \\ &= a - y \end{aligned}$$

$$\begin{aligned} dw_1 &= \frac{dL}{dw_1} \\ &= \frac{dL}{dz}\frac{dz}{dw} \\ &= dzx_1 \end{aligned}$$

$$\begin{aligned} db &= \frac{dL}{dz}\frac{dz}{db} \\ &= dz * 1 \\ &= dz \end{aligned}$$

$$\begin{aligned} w_1 &:= w_1 - \alpha dw_1 \\ w_2 &:= w_2 - \alpha dw_2 \\ b &:= b - \alpha db \end{aligned}$$

---

m개 샘플의 경사하강법

비용함수는 각 손실의 평균이기 때문에, **전체 비용 함수의 도함수도 각 손실의 도함수의 평균**이다.

→ 단일 훈련 샘플의 도함수를 구해 그 평균을 구하면 경사 하강법에 사용할 전체적인 경사를 구할 수 있음.

경사하강법을 사용한 로지스틱 회귀를 구현하는 알고리즘(코딩)

- 경사하강법 한 단계

- 초기화 $J=0$, $dw_1=0$, $dw_2=0$, $db=0$
- 훈련세트 반복해 각 훈련 샘플에 대한 도함수를 계산하고 이를 더하기

for $i=1$ to m

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

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

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

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

$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

$$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$

n이 2라

$$db \mathrel{+}= dz^{(i)}$$

- 각 훈련 샘플의 도함수를 더한 값에서 샘플 수로 나눠 평균 구하기

$J \mathrel{/}= m$; $dw_1 \mathrel{/}= m$; $dw_2 \mathrel{/}= m$; $db \mathrel{/}= m$

- 매개변수 업데이트

$$w_1 := w_1 - \alpha dw_1$$

$$w_2 := w_2 - \alpha dw_2$$

$$b := b - \alpha db$$

경사 하강법을 진행하려면 위를 계속 반복해야 함.

- 위 방식의 약점 → 2개의 for문을 사용한다는 점.

m개의 훈련 샘플을 반복하기 위한 것과 n개의 특성을 반복하기 위한 것.

데이터 집합이 크기 때문에, 명시적인 for문은 알고리즘을 비효율적으로 만듦. (계산 속도가 느려짐)

→ for 문을 사용하지 않기 위해 벡터화(vectorization)가 필요함

3. 파이썬과 벡터화

벡터화(Vectorization)

벡터화하여 계산한 것과 for문을 사용해 계산한 것의 시간 비교

```python
import time

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

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

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

print('--------------------------------------')

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")
```

```
249728.84245611727
Vectorized version : 4.889726638793945ms
--------------------------------------
249728.84245611803
for loop :542.1316623687744ms
```

GPU와 CPU 모두 SIMD라고 불리는 병렬 명령어가 있음.

SIMD (Single Instruction Multiple Data) : 하나의 명령어로 여러 개의 값을 동시에 계산하는 방식.

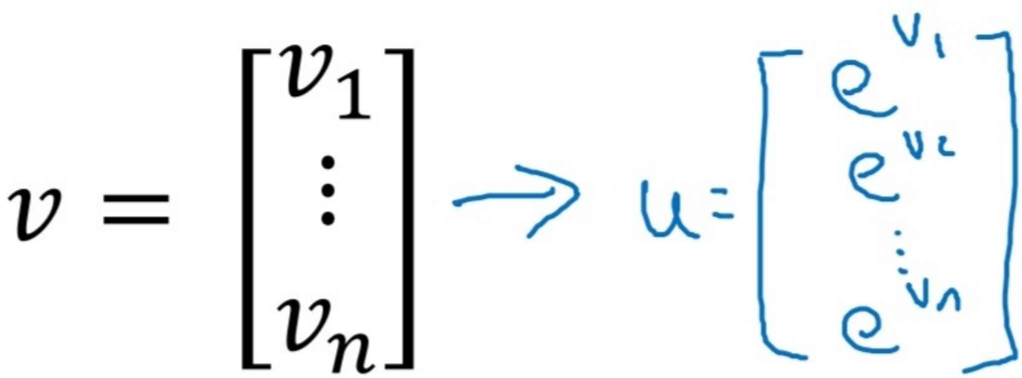
→ for문이 필요 없는 함수를 사용할 때, 병렬화의 장점을 통해 계산을 훨씬 빠르게 할 수 있음.

더 많은 벡터화 예제

컴퓨터의 계산 효율성을 위해서 가능하면 for loop를 피하는 것이 좋음

→ numpy 내장함수 이용

예 :



이를 하기 위한 코드

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

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

로지스틱 회귀 경사 하강법에 벡터화 적용

$dw = np.zeros((n_x, 1))$

- 초기화 $J = 0$, $\cancel{dw_1 = 0}$, $\cancel{dw_2 = 0}$, $db = 0$
- 훈련 세트 반복해 각 훈련 샘플에 대한 도함수를 계산하고 이를 더하기

$$\text{for } i = 1 \text{ to } m$$

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

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

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

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

$$\left.\begin{aligned} &\cancel{dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}} \\ &\cancel{dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}} \end{aligned}\right\}_{n이\ 2라} \longrightarrow dw \mathrel{+}= x^{(i)} dz^{(i)}$$

$$db \mathrel{+}= dz^{(i)}$$

- 각 훈련 샘플의 도함수를 더한 값에서 샘플 수로 나눠 평균 구하기

$$J \mathrel{/}= m \ ; \ \cancel{dw_1 \mathrel{/}= m} \ ; \ \cancel{dw_2 \mathrel{/}= m} \ ; \ db \mathrel{/}= m$$

$$dw \mathrel{/}= m$$

- 매개변수 업데이트

$$w_1 := w_1 - \alpha dw_1$$

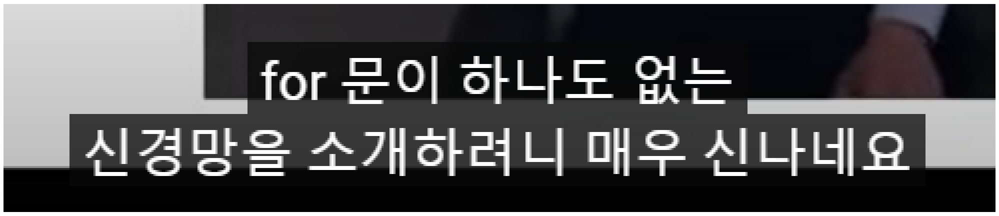
$$w_2 := w_2 - \alpha dw_2$$

$$b := b - \alpha db$$

→ 훈련 세트를 반복하기 위한 for문이 남아 있음.

로지스틱 회귀의 벡터화

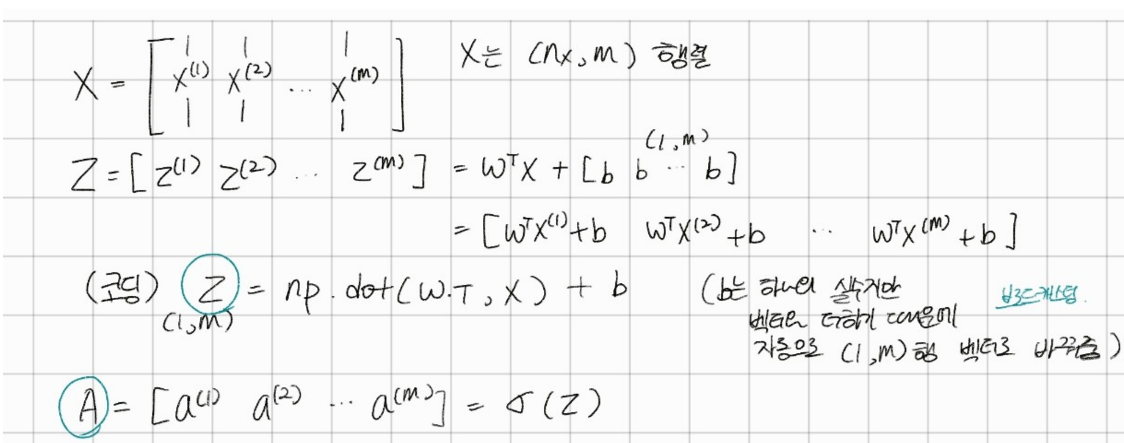
for문을 하나도 사용하지 않고 구현.



(신나신 교수님..^-^)

- $z^{(i)} = W^T x^{(i)} + b$
- $a^{(i)} = \sigma(z^{(i)})$

위를 벡터화를 이용해 계산하는 방법



→ z와 a를 한 번에 계산할 수 있음.

로지스틱 회귀의 경사 계산을 벡터화 하기

- dZ, dw, db 구하기

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

$$A = \sigma(Z)$$

$$dZ = A - Y$$

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

$$db = \frac{1}{m} np.sum(dZ)$$

- for문을 사용한 경사 하강법

$J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$

for $i = 1$ to m

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

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

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

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

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ n이 2개

$db \mathrel{+}= dz^{(i)}$

$J /= m$; $dw_1 /= m$; $dw_2 /= m$; $db /= m$

$w_1 := w_1 - \alpha dw_1$

$w_2 := w_2 - \alpha dw_2$

$b := b - \alpha db$

- 벡터화한 경사하강법

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

$$A = \sigma(Z)$$

$$dZ = A - Y$$

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

$$db = \frac{1}{m} np.sum(dZ)$$

$$w_1 := w_1 - \alpha dw_1$$

$$w_2 := w_2 - \alpha dw_2$$

$$b := b - \alpha db$$

→ 여러번 경사하강법을 진행하고 싶다면 이 전체를 for문 돌리면 됨.