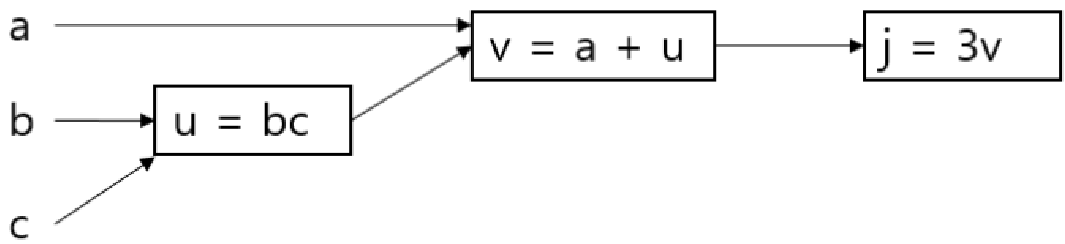
2주차 강의 요약

01. 계산 그래프

- $J(a,b,c) = 3(a+bc)$의 계산 그래프 만드는 과정-왼쪽에서 오른쪽으로 전방향 전파(forward pass)를 거쳐 $J(w,b)$ 비용 함수를 계산

---

1. $u = bc$
2. $v = a + u$
3. $J = 3v$



02. 계산 그래프로 미분하기

- 미분의 연쇄법칙(chain rule)이란 합성함수의 도함수에 대한 공식
- 합성함수의 도함수(derivative)는 합성함수를 구성하는 함수의 미분을 곱함으로써 구할 수 있음(겉미분*속미분)
- 기본적인 아이디어: 오른쪽에서 왼쪽으로 역전파(backpropagation) 진행→x가 바뀜에 따라 J는 어떻게 바뀌는가?→$\frac{dJ}{dx}$
  - $v = a + u \rightarrow J = 3v$
  - $\frac{dJ}{da} = \frac{dJ}{du}\frac{du}{da}$
- 코드 작성 시 편의를 위해 아래와 같이 도함수를 정의함
  - 최종변수를 Final output var, 미분하려는 변수를 var이라고 정의

$$\frac{dFinaloutputvar}{dvar} = dvar$$

- 위의 예시로 계산한 결과

$$dv = \frac{dJ}{dv} = 3$$

$$du = \frac{dJ}{dv}\frac{dv}{du} = 3 * 1 = 3$$

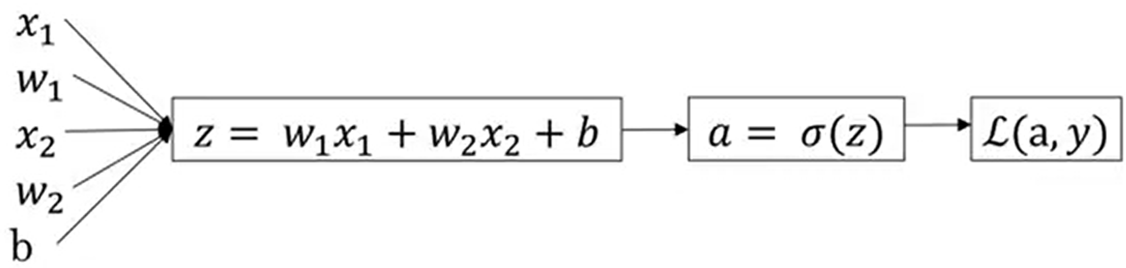
$$da = \frac{dJ}{dv}\frac{dv}{da} = 3 * 1 = 3$$

$$db = \frac{dJ}{du}\frac{du}{db} = 3 * 2 = 6$$

$$dc = \frac{dJ}{du}\frac{du}{dc} = 3 * 3 = 9$$

03. 로지스틱 회귀의 경사하강법

- 단일 샘플에 대한 경사하강법



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

$$dz = a - y$$

$$dw_1 = \frac{dL}{dw_1} = x_1dz, dw_2 = \frac{dL}{dw_1} = x_2dz$$

$$db = \frac{dL}{db} = dz$$

➡️ **결과**

$$w_1 = w_1 - \alpha dw_1, w_2 = w_2 - \alpha dw_2, b : b - \alpha db$$

04. m개 샘플의 경사하강법

- 단일 샘플이 아닌 **m개의 샘플**에 대한 경사하강법
- 로지스틱 회귀에서 비용 함수는 다음과 같이 표현됨

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{i=m}(L(a^{(i)},y^{(i)}))when(x^{(i)},y^{(i)})$$

- **코드**

```
J=0, dw1=0, dw2=0, db=0
for i in range(1, m+1):
        z(i) = wT x(i) + b
        a(i) = σ(z(i))
        J += -[y(i) log a(i) + (1-y(i)) log(1-a(i))]
        dz(i) = a(i) - y(i)
        dw1 += x1(i) dz(i)
        dw2 += x2(i) dz(i)

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

Details

dw_1, dw_2, db는 값을 저장하는 데 사용되고 있음. 이 식들는 첨자 (i)가 사용되지 않는데, 이는 식들이 훈련 세트 전체를 합한 값을 저장하고 있기 때문임. 반면 $dz^{(i)}$는 훈련 샘플 하나 당$(x^{(i)}, y^{(i)})$의 dz이기 때문에 첨자 (i)를 사용함.

➡️ 위 코드를 한 번 실행하면 한 단계의 경사 하강법을 실행하는 것. 따라서 훈련을 위해선 경사 하강법을 여러 번 실행해야 함.

그러나, 이 방법엔 문제가 있음

for 문을 두 개 써야 한다는 점. 첫 번째 for문은 m개의 샘플 데이터를 도는 데, 두 번째 for 문은 n개의 특성을 도는 데 쓰임→이런 명시적인 for문은 알고리즘을 비효율적으로 만듦(계산 속도가 느려짐)→**명시적인 for문 없이 코드를 구현해야 큰 데이터 집합도 처리 가능!**

✅ **vectorization(벡터화): 명시적인 for문을 제거해 큰 데이터 집합을 용이하게 처리하는 방**

05. 벡터화(vectorization)

- 벡터화의 예시

문제: $z = W^T X + b$ $\{ W \in \mathbb{R}^{1,n-x},\ X \in \mathbb{R}^{n-x,m} \}$

Non-Vectorized

```
z=0
for i in range(n_x):
        z+=W^T[i] * X[i]
z+=b
```

Vectorized

```
z= np.dot(W^T, X)+b
```

⇒ much faster computation!

- SIMD(Single Instruction Multiple Data): 병렬 프로세서의 한 종류로, 하나의 명령어로 여러 개의 값을 동시에 계산하는 방식. 이는 벡터화 연산을 가능하게 함. CPU와 GPU를 이용한 계산에 모두 적용할 수 있음.

06. 더 많은 벡터화 예제

- 컴퓨터의 계산 효율성을 위해서 가능하면 for문을 피하는 것이 좋음
- 벡터화를 위해 자주 쓰는 numpy 함수
    - np.dot(a,b) # inner product
    - np.exp(v) # exponential
    - np.log(v) #log v
    - np.abs(v) #absolute value
    - np.maximum(v,0) # v와 0중 큰 값을 반환
    - ** #squared value
    - 1/v #inverse of v
    - np.zeros(m,n) # (m,n) 짜리 0 행
- 로지스틱 회귀에는 두 개의 for문이 존재(m개의 훈련 데이터셋 학습/n개의 특징 업데이트)→아래 코드는 n개의 특징을 업데이트하는 for문을 벡터화로 대체한 예

＊ 2번째 for문 벡터화하기

$J = 0,\ \boxed{dw_1 = 0,\ dw_2 = 0},\ db = 0$ …… $dW = 0$

for i in range(1, m+1):

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

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

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

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

$\left.\begin{array}{l} dw_1 += x_1^{(i)} dz^{(i)} \\ dw_2 += x_2^{(i)} dz^{(i)} \end{array}\right\}$ … $dW += x^{(i)} dz^{(i)}$

$J /= m,\ \boxed{dw_1 /= m,\ dw_2 /= m},\ db /= m$

… $dW /= m$

07. 로지스틱 회귀의 벡터

- 벡터화를 통해 m개의 샘플의 forward pass를 동시에 계산하는 방법-for문을 아예 쓰지 않음
  - 원래대로라면 아래의 식은 for문을 통해 i를 변화시켜 가며 계산해야 했음
    - $z^{(i)} = w^T x^{(i)} + b$
    - $a^{(i)} = \sigma(z^{(i)})$
  - 하지만 벡터화를 사용하면 다음과 같이 간결하게 계산할 수 있음
    - $Z = [z^{(1)}, z^{(2)} \dots z^{(m)}] = np.dot(np.transpose(W), X) + b$

$W^T = [\, w_1 \ w_2 \ \cdots \ w_{n\_x} \,],\quad X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix} \in \mathbb{R}^{n\_x, m}$

$\in \mathbb{R}^{1 \times n\_x}$

$W^T X = [\, w_1 x_1^{(1)} + w_2 x_2^{(1)} + \cdots + w_{n\_x} x_{n\_x}^{(1)},\ \cdots \,]$

$\in \mathbb{R}^{1 \times m}$

✅ $np.dot(np.transpose(w), x)$는 (1,m)크기의 행렬과 상수 b를 더해 오류가 날 것 같지만, 파이썬이 자동적으로 상수 b를 (1,m) 크기의 행렬로 브로드캐스팅 해주기 때문에 오류가 발생하지 않음

- $A = [a^{(1)}, a^{(2)} \dots a^{(m)}] = \sigma(Z)$

08. 로지스틱 회귀의 경사 계산을 벡터화 하기

- 벡터화를 통해 m개의 샘플에 대한 경사 계산을 동시에 하는 방법

$$dZ = [\, dz^{(1)} \quad dz^{(2)} \cdots \quad dz^{(m)} \,] \in \mathbb{R}^{1 \times m}$$

$$A = [\, a^{(1)}, a^{(2)} \cdots a^{(m)} \,], \quad Y = [\, y^{(1)}, y^{(2)} \cdots y^{(m)} \,]$$

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

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

$$dw \xRightarrow{\text{벡터화}} \frac{1}{m} X dZ^T = \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

$$= \frac{1}{m} [\, x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)} \,] \in \mathbb{R}^{n_x, 1}$$

* 경사 하강법 vectorization (1 time)

non-vec	vec
$z^{(i)} = w^T x^{(i)} + b$	$Z = np.dot(W^T, X) + b$
$a^{(i)} = \sigma(z^{(i)})$	$A = \sigma(Z)$
$dz^{(i)} = a^{(i)} - y^{(i)}$	$dZ = A - Y$
$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$	$dW = \frac{1}{m} X dZ^T$
$db \mathrel{+}= dz^{(i)}$	$db = \frac{1}{m} np.sum(dZ)$
	$w: \; w - \alpha dW$
	$b: \; b - \alpha db$

그러나 경사 하강을 여러 번 한다면, 이때는 어쩔 수 없이 for문을 써야 함