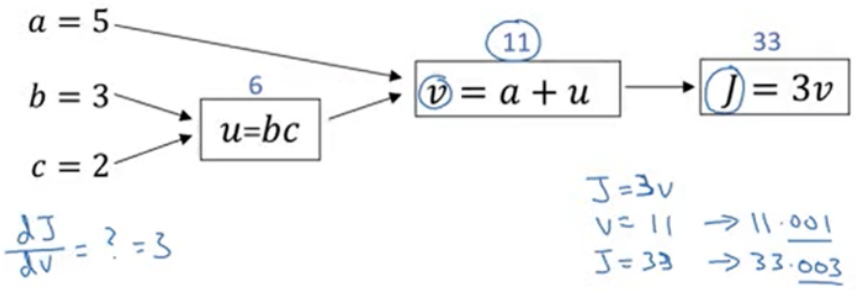
[Week 2] 1. Chain Rule

Cost function J에 대한 도함수를 작성하는 방법

dJ/dv : v의 값을 아주 조금 바꾸면 J의 값이 어떻게 바뀔 것인가를 나타냄

dJ/da : J를 0.001만큼 증가시키면 a는 0.003 만큼 증가함. 즉, 도함수는 3이다.



Chain Rule

a의 변화가 computation graph의 오른쪽으로 전파되어 최종적으로 J도 변화시킴. 즉, a가 바면 v를 바뀌고, v가 바뀌면 J가 변함. 이때 v는 dv/da에 의한 양 만큼 증가한다. 미적분에는 연쇄법칙이라는 게 존재하는데, 아래와 같다.

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx},$$

and

$$\left.\frac{dz}{dx}\right|_x = \left.\frac{dz}{dy}\right|_{y(x)} \cdot \left.\frac{dy}{dx}\right|_x,$$

• 미분의 연쇄법칙이란 합성함수의 도함수에 대한 공식이다. 합성함수의 도함수는 합성함수를 구성하는 함수의 미분을 곱함으로써 구할 수 있다. 위의 규칙을 적용하여 dJ/da의 값을 $dJ/dv, dv/da$의 값을 곱하여 구할 수 있다.

표기법 : $dFinalOutput/dvar$인경우 dvar라고 표기함.

ex) dJ/da → da, dJ/dv → dv

[Week 2] 2.Gradient Descent

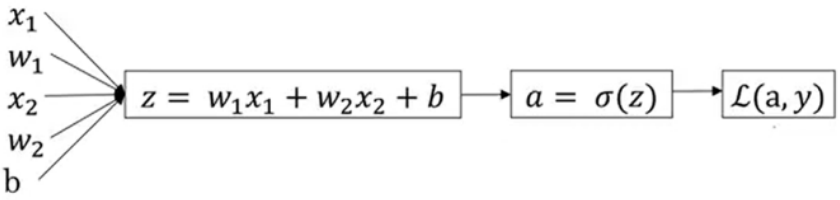
Gradient Descent of Logistic Regression

로지스틱 회귀를 구현하는 데 필요한 도함수를 계산하는 방법에 대해 알아보자

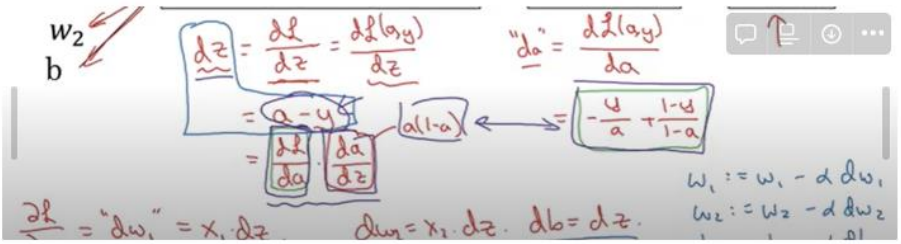
핵심 : 공식 구현 방법에 대해 파악하기

[복습]

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



우리는 w,b값을 바꿔가면서 L의 값을 최소한으로 줄이고자 하는게 목표이다. 역방향으로 연산을 해보자.



$$dw_1 = \frac{dL}{dw_1} = x_1 dz$$

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

우리는 단일 샘플에 대한 한번의 연산만을 해보았지만 실제 로지스틱 모델에서는 m개의 훈련 샘플을 가진 세트 전체를 훈련해야 한다. 이를 아래에서 알아보자.

Gradient Descent on m examples

$$J(w, b) = \Sigma L(\hat{}y(i), y(i))/m$$

비용함수를 다시 살펴보자. 비용함수는 각 샘플의 손실에 대한 평균이다. 위의 예제에서 단일 훈련샘플에 대해 도함수를 구했던 것처럼, 전체 데이터셋에 대해 구하고 평균을 구하면 경사하강법에 사용할 gradient를 구할 수 있다.

경사하강법의 1 step의 코드이다. 경사하강법을 여러번 진행하여 파라미터를 여러번 업데이트 하려면 아래의 작업을 반복해야 한다.

J=0; dw_1=0; dw_2=0; db=0; 이라고 하고,

for i=1 to m

z^i = w^T*x^i + b

a^i = σ(z^i)

J += - (ylogˆy + (1-y)log(1-ˆy))

dz^i= a^i - y^i

dw_1 += x_1^i * dz^i

dw_2 += x_2^i * dz^i //파라미터가 w1,w2만 있다고 가정한 식이다. 더 있다면 dw_3 ...

J/=m; dw_1/=m; dw_2/=m; db/=m; // 평균 연산

w1 = w1-α*dw_1

w2 = w2-α*dw_2

b = b- α*db

- dw_1,dw_2,db는 각각의 매개변수에 대한 비용함수 J의 도함수를 계산한 것이다. 또한 이 값들은 값을 저장하는데 사용하고 있다. 이 값들은 첨자가 없는데, 그 이유는 전체 훈련 세트의 값을 반영하였기 때문이다.

단점

- for문을 두개 만들어야 한다. (전체 데이터셋, n개의 feature가 있는 경우)

⇒ for문은 알고리즘을 비효율적으로 만든다. 따라서 우리는 for문없이 구현해야 더 큰 데이터 집합을 처리하기 용이하기 때문에 **vectorization**을 통해 for문을 사용하지 않고 처리하게 되었다.

[Week 2] 3. Vectorization

Vectorization

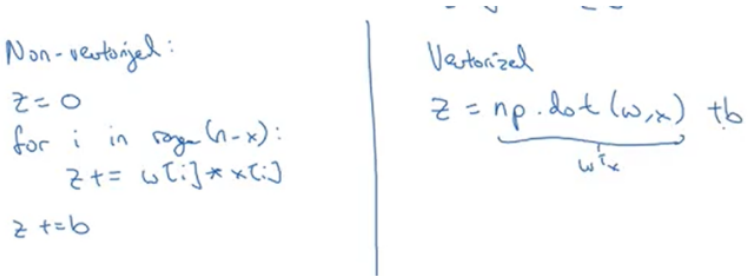
큰 데이터 세트를 학습시키기 때문에 코드가 빠르게 실행되는 것이 중요하다.

로지스틱 회귀에서는 아래의 함수 연산을 수행했었다.

$$\hat{}y = \sigma(w^T * x + b)$$

해당 연산이 벡터화 되었을 때와 for문을 사용할 때를 예시로 비교해보자.

Non-vectorized vs Vectorized



Vector로 만들어 병렬 연산(행렬곱)을 통해 훨씬 빠른 계산이 가능하다.

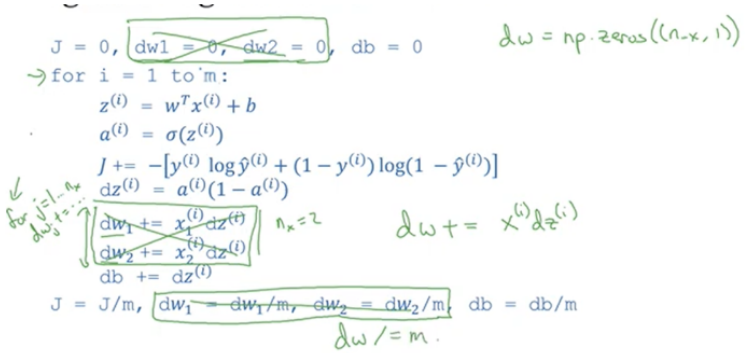
SIMD : Single Instruction Multiple Data의 줄임말. GPU는 SIMD의 빠른 연산이 가능하다.

결론 : 컴퓨터의 계산 효율성을 위해서 가능하면 "for loop" 을 피하는 것이 좋다.

방법 : 파이썬 NumPy 등 내장 함수를 이용한다!(하나의 호출만을 사용하므로 편리함)

Vectorizing Logistic Regression

저번에 배웠던 Logistic Regression의 벡터화를 진행해보자.



그러나 아직 m개의 training data의 연산 대한 loop가 하나 남아 있다. For 문을 하나도 쓰지 않고 구현하려면 어떻게 해야 할까?

Forward Propagation

x는 training input을 열로 쌓은 행렬임을 기억하자.

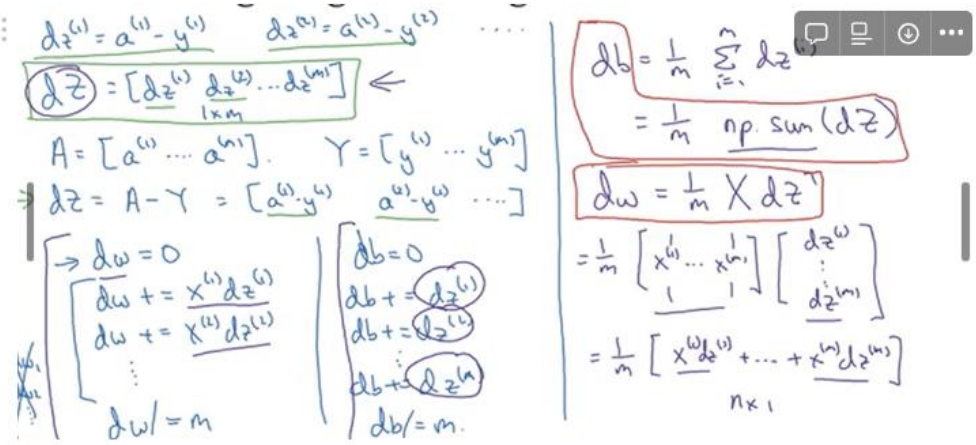
$$Z = np.dot(np.transpose(W), X) + b$$

우리는 z= np.dot(w*T,X)+b 라는 한줄의 코드로 Z를 연산할 수 있다. 즉, z와 a를 하나씩 계산하기 위해 m개의 훈련샘플을 순환하는 대신에 한줄의 코드와 시그마 연산으로 모든 a를 동시에 계산할 수 있다.

** 여기서 (1,m) 크기의 행렬과 상수 b를 더하기에 오류가 날 것 같지만, 파이썬이 자동적으로 상수를 (1,m) 크기의 행렬로 broadcasting 해주기에 오류가 발생하지 않는다.

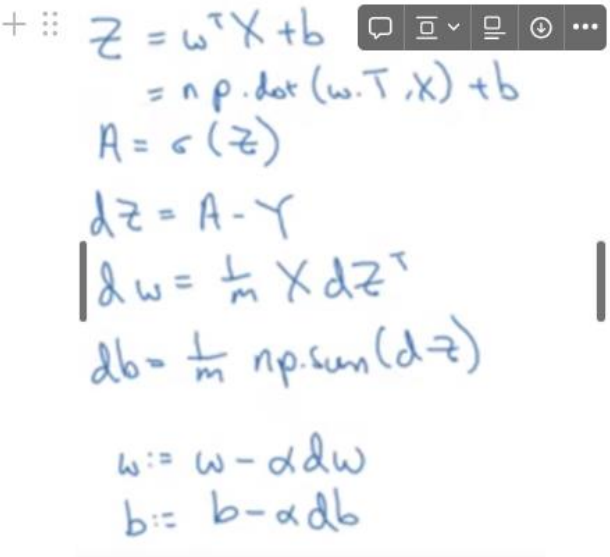
Vectorizing Logistic Regression's Gradient Computation

이번에는 경사계산까지 하는 방법을 알아보자.



빨간색 부분을 보자. 위에서와 같은 방식으로 우리는 행렬곱 연산으로 for loop을 대신할 수 있다.

최종적으로 vectorized 된 Logistic Regression 연산은 아래와 같다.



하지만 경사하강법 자체를 여러번 반복하고 싶다면 for문을 밖에 사용해 주어야 한다.