2주차

날짜	@2024년 3월 19일
과제	강의 요약 출석 퀴즈 ✔ 캐글 필사
세부내용	[딥러닝 1단계] 2-2. 신경망과 로지스틱회귀 (계산 그래프~m개 샘플의 경사하강법) 3-1. 파이썬과 벡터화 (벡터화~로지스틱 회귀의 경사 계산을 벡터화 하기)
자료	[Week2] 캐글 필사 자료 [Week2] 출석퀴즈 C1_W2_note.pdf
과제물	[Week2] 캐글 필사.ipynb

신경망과 로지스틱회귀

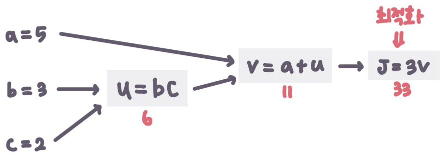
7️⃣ 계산 그래프

신경망의 계산

- 전방향 전파 : 신경망의 출력값을 계산
- 역방향 전파 : 경사나 도함수 계산
- 계산 그래프를 보면 신경망의 계산을 왜 이렇게 나누는지 알 수 있음

계산 그래프(Computation Graph)

- Let J(a,b,c) = 3(a+bc) → 3단계의 과정이 필요
  ① bc를 계산 : u = bc
  ② a+u를 계산 : v = a + u
  ③ J를 계산 : J = 3v



- 계산 그래프는 J 같은 특정한 출력값 변수를 최적화하고 싶을 때 유용
- 로지스틱 회귀의 경우 J는 비용함수
- 왼쪽에서 오른쪽 방향으로 J를 계산 ⇒ 전방향 전파

8️⃣ 계산 그래프로 미분하기

Computing derivatives (1)

- $\frac{dJ}{dv}$(v에 대한 J의 도함수) = 3 → `dv=3`
- $\frac{dJ}{da}$(a에 대한 J의 도함수) = $\frac{dJ}{dv} \cdot \frac{dv}{da}$ = 3 · 1 = 3 ⇒ chain rule → `da=3`
- $\frac{dv}{da}$ = 1
- $\frac{dJ}{dv}$를 계산한 것이 $\frac{dJ}{da}$를 계산하는 데 도움을 줌 ⇒ 역방향 계산의 한 단계
- $\frac{dFinalOutputVar}{dVar}$를 코드에서 구현할 때, 변수 이름은 dvar로 지정

Computing derivatives (2)

- dv=3 → da=3
- dv=3 → du= $\frac{dJ}{dv}$ · $\frac{dv}{du}$ =3 → db= $\frac{dJ}{du}$ · $\frac{du}{db}$ =3·2=6
- dv=3 → du=3 → dc= $\frac{dJ}{du}$ · $\frac{du}{dc}$ =3·3=9
- 오른쪽에서 왼쪽 방향으로 도함수를 계산 ⇒ 역방향 전파

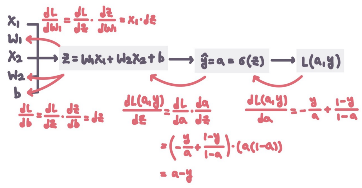
9️⃣ 로지스틱 회귀의 경사하강법

Logistic Regression Recap

- $z = w^T x + b$
- $\hat{y} = a = \sigma(z)$
- $L(a, y) = -(y \log(a) + (1-y)\log(1-a))$
- 목적 : 매개변수 w와 b를 변경해서 손실을 줄이는 것

Logistic Regression Derivatives

- `da` = $\frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$
- `dz` = $\frac{dL(a,y)}{dz} = \frac{dL}{da} \cdot \frac{da}{dz} = a - y$
- `dw1` = $\frac{dL}{dw_1} = x_1 dz$
- `dw2` = $\frac{dL}{dw_2} = x_2 dz$
- `db` = dz



- 도함수를 계산한 후 단일 샘플에 대한 파라미터 업데이트
    - $w_1 := w_1 - \alpha \cdot dw_1$
    - $w_2 := w_2 - \alpha \cdot dw_2$
    - $b := b - \alpha \cdot db$

🔟 m개 샘플의 경사하강법

비용함수 다시 살펴보기

- $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$ → 각 손실의 평균
- $a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$
- 단일 훈련 샘플 ($x^{(i)}$, $y^{(i)}$)를 사용했을 때
    - $\frac{d}{dw_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{d}{dw_i} L(a^{(i)}, y^{(i)})$
      → w_1에 대한 전체 비용 함수의 도함수 = w_1에 대한 각 손실 항 도함수의 평균

Logistic regression on m examples

- Let initialize J=0, dw1=0, dw2=0, db=0
- For i=1 to m → 훈련 세트 반복해 각 훈련 샘플에 대한 도함수를 계산하고 더함
    - $z^{(i)} = w^T x^{(i)} + b$
    - $a^{(i)} = \sigma(z^{(i)})$
    - $J += -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$
    - $dz^{(i)} = a^{(i)} - y^{(i)}$
    - $dw_1 += x_1^{(i)} dz^{(i)}$
    - $dw_2 += x_2^{(i)} dz^{(i)}$ ⇒ n=2개의 반복 (피처의 수가 늘어나면 반복 수도 증가)
    - $db += dz^{(i)}$
- $J /= m;\ dw_1 /= m;\ dw_2 /= m;\ db /= m$
- dw1, dw2, db는 값 저장에 사용 → dw1은 w에 대한 전체 비용 함수의 도함수와 같음
- $w_1 := w_1 - \alpha dw_1;\ w_2 := w_2 - \alpha dw_2;\ b := b - \alpha db$
- 약점 : m개의 훈련 샘플을 반복하는 for문과 n개의 특성을 반복하는 for문 필요 → 비효율적

    ⇒ Vectorization : 명시적인 for문을 제거할 수 있게 해줌

파이썬과 벡터화

1️⃣ 벡터화

벡터화(Vectorization)란?

- $z = w^T x + b$ → $w, x \in \mathbb{R}^{n_x}$
- Non-vectorized

    z = 0
    for i in range(n_x):
      z += w[i] * x[i]
    z += b

- Vectorized

    z = np.dot(w,x) + b ⇒ 계산이 더 빠름

```
import numpy as np

# a라는 배열 생성 후 출력
a = np.array([1,2,3,4])
print(a)
# [1 2 3 4]

# 벡터화 예시
import time

a = np.random.rand(1000000)  # 백만 차원의 배열 생성
b = np.random.rand(1000000)

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

print(c)
```

```
print("Vectorized version:" + str(1000*(toc-tic)) + "ms")
# 250222.13354707597
# Vectorized version:5.106210708618164ms

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")
# 250222.13354708292
# for loop:540.3635501861572ms
```

- 같은 값을 계산했지만 시간 효율이 다름

GPU와 CPU

- GPU와 CPU 모두 SIMD(Single Instruction Multiple Data) 병렬 명령어 존재
- np.dot을 사용하거나 for문이 필요 없는 함수를 사용할 때 Numpy가 병렬화 장점을 통해 계산을 훨씬 빠르게 수행할 수 있음

2️⃣ 더 많은 벡터화 예제

Neural Network Programming Guideline

- 가능한 한 for문을 쓰지 않는 것
- u = Av 계산의 정의 : $u_i = \sum_i \sum_j A_{ij} v_j$
  - non-vectorized version → 2개의 for문 존재

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

  - vectorized version → `u = np.dot(A,v)`

Vectors and Matrix Valued Functions

- 메모리 상에 벡터 v가 있다고 가정, 원소마다 지수 연산을 하길 원함
- non-vectorized

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

- vectorized

  ```
  import numpy as np
  ```

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

- 파이썬 numpy에서 많은 벡터 함수를 제공하고 있음

  ex) `np.log(v), np.abs(v), np.max(v,0), v**2, 1/v`

Logistic Regression Derivatives

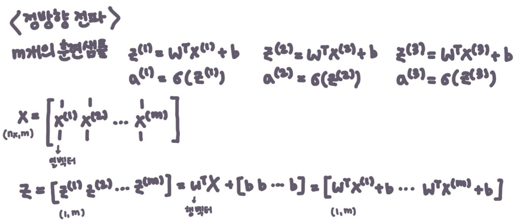
- for문 하나 제거해보

```
J=0, dw=np.zeros((n_x,1)), db=0
for i=1 to n:
    z[i] = w^T * x[i] + b
    a[i] = sigmoid(z[i])
    J += -(y[i]*np.log(yhat[i]) + (1-y[i])*np.log(1-yhat[i]))
    dz[i] = a[i] * (1-a[i])
    dw += x[i] * dz[i]
  db += dz[i]
J/=m; dw/=m; db/=m
```

3️⃣ 로지스틱 회귀의 벡터화

Vectorizing Logistic Regression

- $Z = \left[z^{(1)} \, \cdots \, z^{(m)}\right] = w^T X + \left[b \, \cdots \, b\right] = \left[w^T x^{(1)} + b \, \cdots \, w^T x^{(m)} + b\right]$
- Python) `Z = np.dot(w.T,X) + b`
- 파이썬이 자동으로 b를 (1,m) 벡터로 만들어 계산을 수행 ⇒ Broadcasting
- $A = \left[a^{(1)} \, \cdots \, a^{(m)}\right] = \sigma(Z)$



4️⃣ 로지스틱 회귀의 경사 계산을 벡터화 하기

Vectorizing Logistic Regression

- $dZ = \left[dz^{(1)} \, dz^{(2)} \, \cdots \, dz^{(m)}\right]$ → 1 x m 행렬
- $A = \left[a^{(1)} \, \cdots \, a^{(m)}\right] = \sigma(Z)$
- $Y = \left[y^{(1)} \, \cdots \, y^{(m)}\right]$

  ⇒ $dZ = A - Y = \left[a^{(1)} - y^{(1)} \, \cdots \, a^{(m)} - y^{(m)}\right]$
- $db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$ = `np.sum(dZ)`
- $dw = \frac{1}{m} X (dZ)^T = \frac{1}{m} \left[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)}\right]$ → m x 1 행렬

Implementing Logistic Regression

```
for iter in range(1000):  # 경사 하강법 반복
    Z = np.dot(w.T, X) + b
    A = sigmoid(Z)
    dZ = A - Y
    dw = np.dot(X, dZ.T) / m
    db = np.sum(dZ) / m
```

```
w -= lr * dw
b -= lr * db
```