[Week 3] 1. Broadcasting

Broadcasting은 파이썬의 코드 실행시간을 줄여주는 한 방법이다.

파이썬에서 브로드캐스팅 하는 과정을 알아보자.

```
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) #같은 열의 요소끼리 다 더함
percentage = 100 *A/cal.reshape(1,4)
print(percentage)
```

The purpose of `cal.reshape(1,4)` is to convert the array `cal` into a shape that allows for broadcasting.

When you perform operations between arrays of different shapes, NumPy uses a mechanism called broadcasting to make the arrays compatible. Broadcasting essentially allows **arrays with different shapes to be treated as if they had the same shape** by implicitly expanding the smaller array to match the shape of the larger one.

위에서 실행한 코드는 파이썬의 브로드캐스팅을 활용하였다. (3,4)인 행렬 A를 (1,4)의 행렬로 나누었음을 알 수 있다. 여기서 파이썬은 자동으로 (1,4)의 행렬을 (3,4)행렬로 복제한 후 나누어준 것이다.

예를 들어 아래의 그림과 같이 (4,1)의 행렬과 숫자 100을 더한다고 했을 때에도, 숫자 100이 (4,1)의 벡터로 바뀌어 모든 원소에 더해진다.

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} \quad 100 \quad = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

이러한 **broadcasting은 행벡터와 열벡터 모두에 적용**된다!

+(m,n) 행렬이 있고, (1,n) 행렬의 사칙연산 연산이 있다면? 파이썬은 (1,n)행렬을 m번 복사해서 (m,n)행렬로 만들어준 후, 사칙연산을 수행할 것 이다. 쉽게 이야기 하면, 행벡터 혹은 열벡터가 있을 때, 해당 벡터 외에 다른 행렬의 크기에 맞추어 복사된 후 연산이 수행된다는 것이다.

NumPy 를 사용하면서 헷갈리기 쉬운 오류들

1. Never use “rank 1 array”

```
import numpy as np
a = np.random.randn(5)
print(a)
print(a.shape) #(5,)로 나타남
print(a.T) #transpose 가 먹히지 않음
print(np.dot(a,a.T) #값이 1개가 나옴.



a=np.random.randn(5,1)
print(a.T) # 앞의 a.T 는 대괄호가 하나였으나, 여기는 두개. 왜냐하면 앞의
print(np.dot(a,a.T)) #우리가 원하는 벡터의 외적을 할 수 있게 됨

assert(a.shape==(5,1)) # 우리가 행렬의 차원을 제대로 알지 못하는 경우가
```

- shape 메소드를 사용했을 때 (n,)의 형태가 나온다면 이것은 “rank 1 array”이다. 이는 행벡터나 열벡터가 아닌, 다른 자료구조이다. 따라서 계산이 이상해질 수 있기 때문에 rank 1 array는 사용하지 않는 것을 추천한다.
- 아래의 `a=np.random.randn(5,1)` 처럼, (5,1)의 행렬을 만들어서 사용해야 한다. 혹은 rank1 배열을 reshape 함수를 사용하여 행렬로 바꾸어 사용한다.
- assert - 행렬과 배열의 차원을 확인할 수 있음.
- reshape- 행렬과 벡터를 필요한 차원으로 변환.

[Week 3] 2. Cost Function of Logistic Regression, MLE

ˆy = σ(w^T * x + b) 가 로지스틱 회귀에서 예측값을 도출할 때 사용하는 함수이다. 로지스틱 회귀는 이진분류를 위한 알고리즘이고, 0또는 1로 분류한다. 우리는 y=1일 확률을 ˆy 이라고 하기로 했으므로, y=0일 확률은 1-ˆy 이다.

$if y = 1 : P(y|x) = \hat{}y$

$if y = 0 : P(y|x) = 1 - \hat{}y$

두 식을 하나로 합치면 아래와 같다.

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

*(1-y)는 지수이다.

그리고, 로그 함수는 **단조적**인 을 지닌 함수이기 때문에 P(y|x)을 최대화 하는 것은 logP(y|x)을 최대화 하는 것과 같다. 따라서 로그를 씌워주면 우리가 [Week 1].2 에서 보았던 식이 나오게 된다.

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

근데 왜 앞에 마이너스가 붙으냐?

왜냐하면 우리는 cost function을 최소화 하고 싶기 때문이다. 확률(log*P(y|x)*) 을 최대화 시키는 것은 -1 을 곱한 확률 (−log*P(y|x)*)를 최소화와 등치이다. 따라서 훈련 샘플 하나의 손실 함수는 아래와 같다.

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

+ ⠿

그렇다면 하나의 샘플이 아닌 m개의 샘플에 대한 cost function에 대해 구해보자.

+ ⠿ 비용함수는 m개 훈련 세트 중, 각 샘플 (*x(i)*)이 주어졌을 때, 샘플에 해당하는 라벨(*y(i)*)값이 1 혹은 0 이 될 확률의 곱으로 구할 수 있다. (단, 샘플이 각각 독립적일 때)

$$P(labels\ in\ training\ set) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$$

$$\log P(labels\ in\ training\ set) = \log \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) = -\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

$$J(w, b) = -\log P(labels\ in\ training\ set) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

Maximum likelihood estimation(MLE)

최대 우도 추정은 확률 분포를 사용하는 모델의 파라미터를 추정하는 데 주로 사용된다. 관측된 데이터가 특정 확률 분포에서 파생되었다고 가정하고, 이 데이터를 가장 잘 설명하는 모수 값(파라미터)을 찾는 과정이다.

최대 우도 추정 절차

1. **우도 함수 설정**: 주어진 데이터에 대한 확률 밀도 함수(또는 확률 질량 함수)를 설정한다.
2. **로그 우도 함수 계산**: 우도 함수를 로그 변환하여 계산하면 계산이 용이해지므로 변환한다. 또한, 로그 우도 함수를 최적화할 때 곱셈 연산을 덧셈 연산으로 바꿀 수 있어 편리하다.
3. **최적화**: 로그 우도 함수를 최대화하는 최적화 알고리즘(예: 경사 하강법, 뉴턴-랩슨 등)을 사용하여 최적의 파라미터 값을 찾는다.
4. **추정된 파라미터**: 최대 우도 추정을 통해 추정된 파라미터는 주어진 데이터에 대한 확률 분포를 가장 잘 설명하는 값.

[Week 3] 3. Computation of Neural Network output

우리가 그동안 배웠던 신경망 연산은 각 노드에서 z 값과 a값을 각각 연산한다.

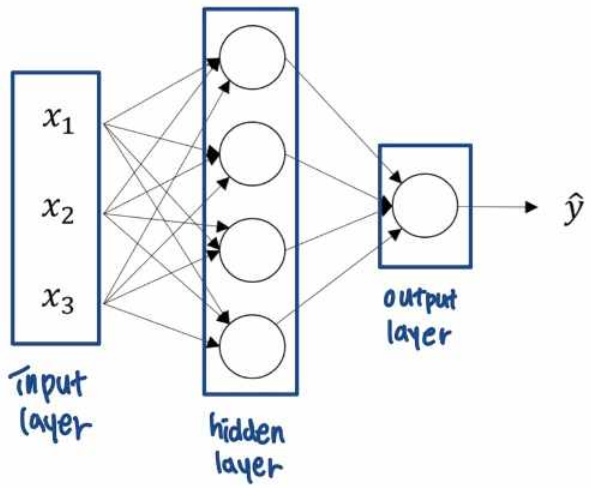
앞으로 사용할 Notation : $z^{[1]} = W^{[1]} + b^{[1]}$

위첨자의 []안의 숫자는 해당 행렬이 속한 layer를 의미하고, 아래첨자의 ()안의 숫자는 각 노드의 번호를 의미한다. 헷갈리지 말 것!

로지스틱 회귀와의 차이점에 주목하자.

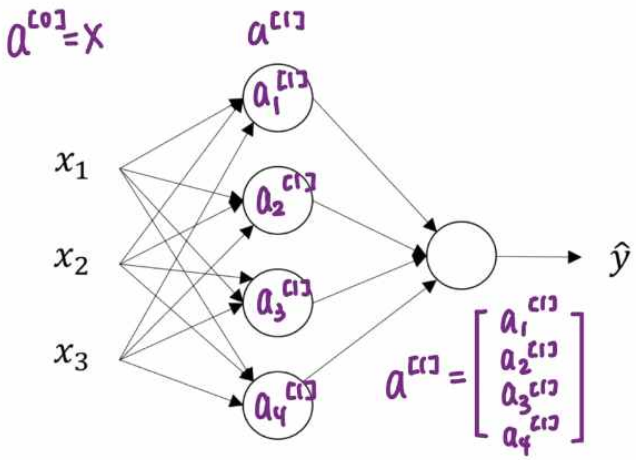
로지스틱 회귀는 z값과 a값의 연산을 딱 한번 수행했다면, Neural Network에서 **는 z와 a를 여러 번 계산한 후, loss function을 구한다.**

Neural Network Representation



hidden layer의 값은 training set에 기록되지 않는다. 따라서 우리는 input value와 output value는 알 수 있지만 hidden layer의 값들은 알 수 없다.

X= a^[0] : 우리가 입력값을 X라고 표현했었는데, a는 활성값을 의미하고 신경망 층들이 다음층으로 전달해주는 값을 의미한다. 따라서 a^[0]는 입력층의 activation이라고 부른다. 다음 층인 hidden layer에서는 activation a^[1]을 만든다.



로지스틱 회귀와의 차이점을 본다면, 로지스틱 회귀에서는 layer가 하나만 있기 때문에 위첨자를 사용하지 않았으나, 신경망에서는 hidden layer가 여러개로 구성될 수 있기 때문에 위첨자를 사용하여 어떤 층에서 만들어진 건지 표기해주어야 한다.

위의 그림은 몇개의 layer를 가진 NN일까?

정답은 2 layer NN이다. 왜냐하면 NN에서는 input layer를 빼고 갯수를 세기 때문이다. 어떻게 보면 input layer를 0번째 층이라고 부르기 때문일지도?

위의 그림에서 a^[1]은 w^[1]와 b^[1]과 관련이 있고, w는 (4,3) 행렬이 되고 , b는 (4,1) 벡터가 된다. 왜냐하면 input feature가 3개이고, hidden layer의 노드가 4개이기 때문이다. 이는 나중에 행렬의 차원을 다루면 더 자세하게 이야기해보자.

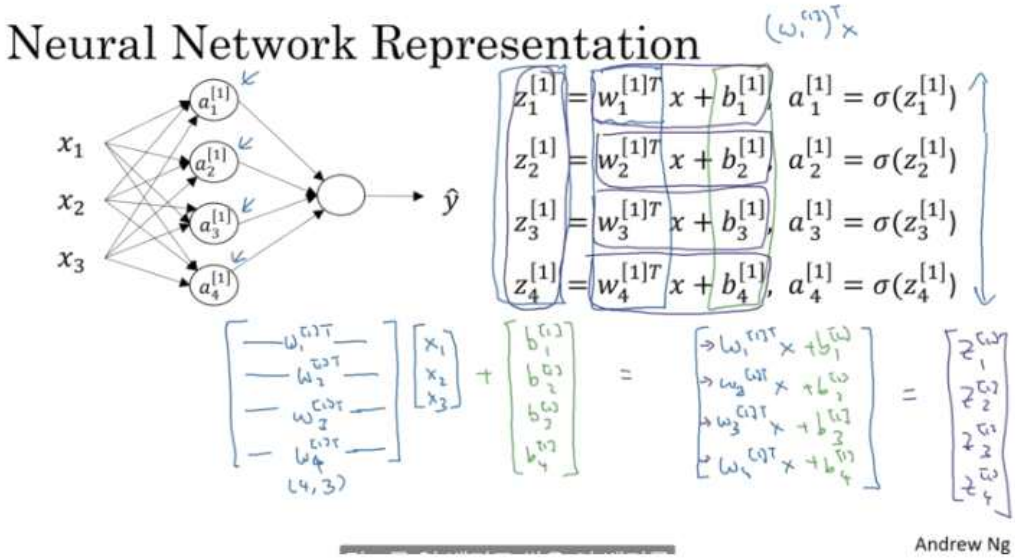
Computation of Neural Network Output

로지스틱 회귀와 유사하게 z연산과 a 연산(시그모이드)을 수행한다. 그러나 차이점은 신경망은 이러한 연산을 여러번 수행한다는 것이다.

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

하지만 이러한 연산은 for문을 써서 하나씩 진행하면 비효율적이므로, 4가지 연산을 벡터화해보자.

갑자기 헷갈려서 정리 : w1이 feature x1,x2,x3에 대한 각각의 가중치를 모두 가지고 있는 것이다. x1은 실제 값이고 w1은 가중치(파라미터)임.



먼저 z를 벡터로 계산하는 법을 알아보자. w를 행렬로 쌓으면(벡터를 여러개를 쌓으면 행렬이 되니까) 된다. 각 노드는 그 노드에 상응하는 w가 있다. 위의 그림에 따르면$w1[1]^T, w2[1]^T, w3[1]^T, w4[1]^T$ 인 행벡터 4개를 쌓으면 (4,3) 행렬인 W^[1]을 얻을 수 있다. 같은 방식으로 b^[1]도 구할 수 있다. 아래의 그림을 보면 벡터화를 통해 진행한 연산이 위의 4개의 수식과 같음을 알 수 있다.

Tip) 한 층에 노드가 여러개이면 세로로 쌓는다. 노드들을 세로로 쌓아 행렬을 만들기!