

# Neural Network and Deep Learning

# Kaggle

## 1. 이전 대회의 Solution 참고하기

↳ 두 번째 대회라면 그 대회의 이전 대회 참고하기

## 2. 대회 관련 블로그, 정보, 가수 참고하기 ~ (둘은)

↳ 이 대회에 필요한 테크닉은 무엇인지,  
인기있는 가수의 특징인지 파악하기

## 3. 베이스라인 구축하기

↳ Kaggle 팀사를 꼭 해보기. → 상위 3개,  
3번식 팀A

## 4. 모델 구현 및 평가를 계속 반복하기

결국 tf, pytorch는  
다 악에서  
언제 쓰기 좋은지  
모아야 함

## 5. 1일 1논문

그냥 많이 읽기

처음에는 한 논문에 떠들어 걸린다

## 6. 머신러닝을 어느정도 알아야 솔루션 - ML vs 알지 않겠지...?

### 7. 대학원 될수?

머신러닝에 흥미, 통계 필요하다보니

논문의 영역에 대응.

학부는 4년. + 경력 4년이 필요하다보니

석사를 거쳐 드.

직군 별로 차이는 있음

→ 대학원 가면  
프로젝트 경험 이  
생각해 줄은  
시간을 확보면 그게  
더 나을 수 있다는 의미.  
(6년에 Kaggle 30개...)

정크로드 계속 보면  
후에는 변하고  
변하는

(현장화(X))

## 8. 주말 ML 븕트캠프로 ML 뉴비의 취업 학습 분위기 조성

인터뷰는 이제 블로그 수준이 늘면 Engineer, Developer 뿐으로는 꼭 석사 X  
Research 밟고 이 과정에서 합격야

## 최신 ML 트렌드

- 양자학적 ML  
Quantum
- Big-model
- 가상현실 등

공부할 때 주변 이야기, 생각에 귀 기울고 읊어보기..



# ★ General Methodology of Deep Learning ★

① [ Initialize Parameters  
    Define hyperparameters

② Loop for num\_iterations

- ① Forward Propagation
- ② Compute Cost function
- ③ Backward Propagation
- ④ Update Parameters

(using parameters and grads from back prop)

③ Use trained parameters to predict labels

## Matrix Product

$a = (4 \times 3)$  배열

$b = (3 \times 2)$  배열

① dot product  $\Rightarrow$   $\sum x_i \cdot y_i$

`np.dot(x1, x2)`

`c = np.dot(a, b)`

`c.shape  $\Rightarrow$  (4, 2)`

② outer product  $\Rightarrow$   $m \times n$  행렬

`np.outer(x1, x2)`

`c = np.outer(a, b)`

`c.shape  $\Rightarrow$  (12, 6)`

③ element-wise product  $\Rightarrow$

`np.multiply(x1, x2)`

`c = a * b`

`c = np.multiply(a, b)`

Error  $\Rightarrow$  Cannot Broadcast.

# Broadcasting in Python

파이썬 코드를 더 빠르게 풀리는 방법

〈General Principle〉

```
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]])
print(A)
```

```
[[ 56.      0.      4.4    68. ]
 [ 1.2     104.    52.     8.   ]
 [ 1.8     135.    99.     0.9]]
```

```
cal = A.sum(axis=0) # axis = 0 means sum of one vertical line
print(cal)
```

```
[ 59. 239. 155.4 76.9]
```

```
percentage = 100 * A/cal.reshape(1,4) # 3x4 matrix devide 1x4 matrix
print(percentage) # careful to reshape. It wants a vertical line
[[94.91525424 0. 2.83140283 88.42652796]
 [ 2.03389831 43.51464435 33.46203346 10.40312094]
 [ 3.05084746 56.48535565 63.70656371 1.17035111]]
```

$$(m,n) \xrightarrow{?} (l,n) \rightarrow (m,n)$$

column vec

$$(m,1) + R \rightarrow \begin{bmatrix} 1+R \\ 2+R \\ \vdots \\ n+R \end{bmatrix}$$

$$(l,m) + R \rightarrow [l+R \ 2+R \ \dots +R]$$

〈 0 || 제 1 〉

① 실수 하나

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$$

〈 Broadcastin g의 약점 〉

큰 유동성 때문에

강제하기 힘든 버그나  
매우 이상하게 생긴 버그가  
생길 수 있다

② 가로 1줄 벡터

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$$

$(2,3) \rightarrow (2,3)$

예로 세로줄 벡터 + 가로줄 벡터는  
오류지만 broadcasting  
그 계산값을 반복하여.  
또한 그러나 생긴 버그를  
줄이는 방법은 다음 장에.

③ 세로 1줄 벡터

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$$

# A Note on Python / Numpy Vectors

```
import numpy as np  
  
a = np.random.randn(5) # 5개의 랜덤 숫자를 생성  
print(a)  
print(a.shape) # rank 1 array : neither row vector or column vector  
print(a.T) # a looks like similar to a.T ↗ ??. 이상하니까 쓰지마  
[ 0.10726102 -0.8333651  0.96262561 -0.35502006  0.93974614]  
(5,)  
[ 0.10726102 -0.8333651  0.96262561 -0.35502006  0.93974614]  
  
print(np.dot(a, a.T)) # it's not vector. it's a number  
2.641812434703152  
  
a가 세로 벡터가 되게 해줌  
a = np.random.randn(5,1) # create a that it was 5x1 vector  
print(a.shape)  
print(a)  
print(a.T) ↗ row vector  
print(np.dot(a,a.T)) # result isn't number, it's vector  
  
(5, 1)  
[[ 0.07835525]  
[-1.43662294]  
[-0.69501606]  
[ 1.33774209]  
[ 0.44835058]]  
[[ 0.07835525 -1.43662294 -0.69501606  1.33774209  0.44835058]]  
[[ 0.00613954 -0.11256695 -0.05445816  0.10481911  0.03513062]  
[-0.11256695  2.06388548  0.99847602 -1.92183098 -0.64411072]  
[-0.05445816  0.99847602  0.48304732 -0.92975224 -0.31161085]  
[ 0.10481911 -1.92183098 -0.92975224  1.7895539  0.59977744]  
[ 0.03513062 -0.64411072 -0.31161085  0.59977744  0.20101824]]
```

<Python / numpy vectors>

①  $\begin{cases} \text{세로} = \text{np.random.randn}(5, 1) \\ \text{가로} = \text{np.random.randn}(1, 5) \end{cases}$   
∴ 꼭  $(\times n, n \times 1)$  vector를 쓰기!

② assert ( $a.shape == (5,1)$ )  
코드를 기록하는 장점이 있으니  
자주 쓰기

③ reshape를 너무 두려워하지 말기

+ )  $a.shape = (3, 4), b.shape = (4, 1)$

for i in range(3):  
 for j in range(4):  
  $c[i][j] = a[i][j]c[j] + b[j]$

vectorization

$\Rightarrow c = a + b.T$

# Loss Function

모델의 성능 판단에 사용

loss가 클수록, 실제값  $y$ 와 예측값  $\hat{y}$ 의 차이가 커진다

(L1) loss

$$L_1(\hat{y}, y) = \sum_{i=0}^{m-1} |y^i - \hat{y}^i|$$

(L2) loss

$$L_2(\hat{y}, y) = \sum_{i=0}^{m-1} (y^i - \hat{y}^i)^2$$

※ L2 구현에는 여러 방법이 있는데,  
 $np.dot()$ 도 유용함

```
def L1(yhat, y):
    loss = np.sum(abs(yhat-y))
    return loss

yhat = np.array([.9, .02, .01, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L1 = " + str(L1(yhat,y)))

L1 = 1.1
```

※  $X = [x_1, x_2 \dots x_n]$  일 때  
 $np.dot(X, X) = \sum_{j=0}^n x_j^2$  같은 이론

```
def L2(yhat, y):
    loss = np.sum((y-yhat)**2)
    return loss

yhat = np.array([.9, .2, .1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L2 = " + str(L2(yhat,y)))

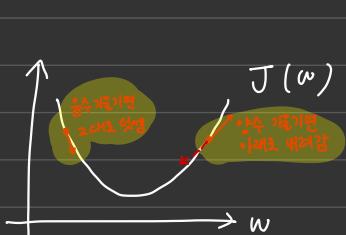
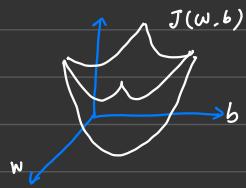
L2 = 0.43
```

Gradient Descent :  $J(w, b)$  을 최소화하는  $w, b$  찾기

$$\text{Recap} : \hat{y} = \sigma(w^T x + b), \sigma(z) = \frac{1}{1+e^{-z}}$$

$$J(w, b) = -\frac{1}{n} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m} \left( \sum_{i=1}^m y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)}) \log (1-\hat{y}^{(i)}) \right)$$



반복 {   
 $w = w - \alpha \frac{dJ(w)}{dw}$  }   
 learning rate  
 slope of a function at the point

$$J(w, b)$$

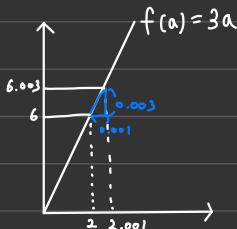
$$w = w - \alpha \boxed{\frac{dJ(w, b)}{dw}} = dw$$

$$b = b - \alpha \boxed{\frac{dJ(w, b)}{db}} = db$$

$\therefore$  Sometimes convex function has multiple local optima

# 이적분 (Intuition about Derivatives)

이적분이 익숙하다면  
넓적도 되는 내용

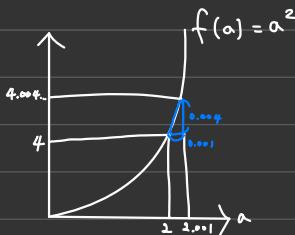


만약  $a=2$  라면  $f(a)=6$

$$a=2.001 \quad f(a) = 6.003$$

$$\therefore \text{Slope} = \frac{\text{상승량}}{\text{내려} \cdot \text{내려}} = \frac{0.003}{0.001} = 3$$

↑↑↑ 3배씩 증가합니다



$$\frac{a=2}{a=2.001} \quad f(a)=4 \quad f(a) \approx 4.004 \Rightarrow \frac{0.004}{0.001} = 4 \dots$$

$$\left( \begin{array}{l} a=5 \\ a=5.001 \end{array} \right) \quad \begin{array}{l} f(a)=25 \\ f(a) \approx 25.000 \end{array} \Rightarrow \frac{0.000}{0.001} = 10 \dots$$

$\xrightarrow{a \times 2}$

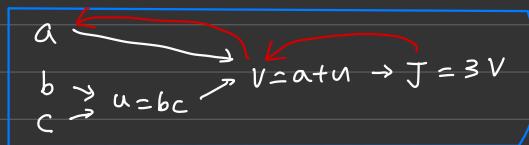
$\therefore$  제곱 배 만큼 증가합니다  $a^2$

$\therefore$  가울기는  $a$ 의 2배 만큼 커진다  $2a$

## Computation Graph

$$J(a, b, c) = 3(a + bc)$$

$$\begin{array}{c} u = bc \\ v = a + u \\ J = 3v \end{array}$$



최종 목적은  $\underbrace{J}_{\text{미분식}}$  계산하기

앞쪽에서  
오른쪽으로 통과 : forward  
↳ 반대 : backward

## Computing derivatives - Back Propagation

위의 계산식에서,

$$\boxed{\frac{dJ}{dv}} = 3 : \text{만약 } v=11 \text{ 이라면 } J=33 \Rightarrow \frac{6.003}{0.001} = 3$$

$v$ 가 변할 때  
 $J$ 의 변화량

$$\boxed{\frac{dJ}{da}} = \frac{dJ}{dv} \times \frac{dv}{da} = 3 \times 1 : \text{만약 } a=5 \text{ 일 때 } \underbrace{a=5.001}_{v=11 \text{에서}} \text{로 증가시키면 } \underbrace{v=11.001}_{\text{되고}} \text{이 되고 } \frac{dv}{da} = 1$$

$a$ 가 변할 때  
 $J$ 의 변화량

$J=33$ 에서  $J=33.003$ 이 된다

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

$$+) \frac{dJ}{db} = \frac{dJ}{du} \cdot \frac{du}{db} = 6$$

$$\begin{cases} b=3 \rightarrow 3.001 \\ u=b \cdot c = 6 \rightarrow 6.002 \\ J=33.006 \end{cases} \Rightarrow \frac{6.002}{0.001} = 2$$

$$\downarrow \frac{0.006}{0.001} = 6$$

## Sigmoid Gradient

back propagation을 이용하여

loss 함수를 최적화하기 위해선

gradient를 계산해야 한다

### ① Sigmoid Derivative

Sigmoid 함수의  $x$ 에 대한 기울기를 구해보자

$$\begin{aligned}\text{sigmoid-derivative}(x) &= \sigma'(x) \\ &= \sigma(x)(1 - \sigma(x))\end{aligned}$$

# Logistic Regression Gradient Descent

Logistic Regression : (loss 값을 줄이기 위해  
w와 b를 감소시키기)

$$z = w^T x + b$$

$$\text{예측값 } \hat{y} = a = \sigma(z)$$

$$L(a, y) = -(y \log(a) + (1-y) \log(1-a))$$

= loss의 미분값

$$\begin{aligned} & \left. \begin{array}{l} x_1 \\ w_1 \\ x_2 \\ w_2 \\ b \end{array} \right\} \rightarrow z = w_1 x_1 + w_2 x_2 + b \rightarrow \hat{y} = a = \sigma(z) \rightarrow L(a, y) \\ & \quad "da" = \frac{dL}{da} = \frac{dL(a, y)}{da} \\ & \quad = -\frac{y}{a} + \frac{1-y}{1-a} \\ & \quad = \frac{dL}{da} \left( \frac{\partial a}{\partial z} \right) \end{aligned}$$

$$\begin{aligned} \frac{dL}{dw_1} &= \frac{dL}{da} \left( \frac{\partial a}{\partial z} \right) = x_1 da \\ dw_1 &= x_1 da \\ db &= da \end{aligned} \quad \left. \begin{array}{l} w_1 = w_1 - \alpha da \\ w_2 = w_2 - \alpha da \\ b = b - \alpha da \end{array} \right.$$

1회의 training 데이터에 대해  
로지스틱 회귀분석을 위한  
가중치 강화를 도입하기 위해  
derivative를 산출하는 방법

# Logistic Regression on m examples

Cost function

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

$$\text{이분} \left\{ \begin{array}{l} a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b) \\ \text{각각의 loss함들의 평균} \end{array} \right.$$

$$\frac{\partial}{\partial w_i} J(w, b) = \frac{1}{m} \sum_{i=1}^m \underbrace{\frac{\partial}{\partial w_i} L(a^{(i)}, y^{(i)})}_{d w_i^{(i)} \sim \text{이전 슬라이드에}} \quad \text{이것 하나하나 계산하는 방식이 나와있음}$$

$(x^{(i)}, y^{(i)})$   
1개 일 때

$d w_1^{(i)}, d w_2^{(i)}, d w_3^{(i)}$ 의 정운

# Logistic Regression on m examples

답변:  $J=0, dw=0, dw_2=0, db=0$

이전 트레이닝 데이터에 대해서 for loop은 사용하지 않음

각각의 데이터에 대해서 이를 값들 구간 뒤 더하기

For  $i=1$  to  $m$

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

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

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

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

$$dw_1 += x_1^{(i)} da^{(i)} \quad ]$$

$$dw_2 += x_2^{(i)} da^{(i)} \quad ]$$

$$db += da^{(i)}$$

$J, dw_1, dw_2, db$

모두 각각 나누기  $m$

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$$w_1 = w_1 - \alpha dw_1$$

$$w_2 = w_2 - \alpha dw_2$$

$$b = b - \alpha db$$

+ for loop 없이 하는 방법은...

Vectorization을 이용

$m=2개인 경우$   
가장하여 진행 중

# Vectorization

: for loop 코드를 제거하는 코드

로직스틱 회귀 분석에서는

$$w = [ ; ], \quad x = [ ; ]$$

$z = w^T x + b$  를 계산해야 한다

이 때  $w$ 와  $x$ 는 서로  $n$  벡터이다

$w \in R^n$ ,  $x \in R^{n \times 1}$  일 때 좋다.

만약 non-vectorized라면 느림

$$z = 0$$

```
for i in range(n-x):
    z += w[i] * x[i]
```

$$z += b$$

Vectorized라면

$$z = \underbrace{np.dot(w, x)}_{w^T x} + b$$

```
import numpy as np

# make an array
a = np.array([1,2,3,4])

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")
```

```
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") # really slow

250021.0082094741
for loop:609.7338199615479ms
```

+) CPU와 GPU 모두

parallelization instruction이 있다

이제 SIMD로 구현해보기

single instruction  
multiple data의 특징이다

이전 일련의 기능을 이용하면,  
parallelism 덕분에 for 안 쓸

이 기능은 GPU에 대해 특별히

## More Vectorization Example

즉, 가능한 for-loop을 피하는 방법

ex1)

$$u = Av$$

행렬 A의 곱셈으로 벡터 u를 계산하고 싶을 때

행렬 곱셈을 하면

$$u_i = \sum_j A_{ij} v_j$$

이는 non-vector 벡터로 나타내면

$$u = np.zeros(n, 1)$$

for ... i

for ... j

$$u[i] += A[i, j] * v[j]$$

→ vector 버전으로,

$$u = np.dot(A, v) \quad \text{혹시!}$$

ex2)

Vectors and Matrix valued functions

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

지수의 방법을 벡터 v의 모든 요소에 적용하고 싶다면

u가 벡터가 되게 하면,

$$\text{즉, } u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}. \text{ 이는 non-vector 벡터.}$$

처음에는 u의 벡터를 0으로 초기화한 뒤 for-loop에서 하나씩 계산

$$u = np.zeros((n, 1))$$

for i in range(n):

$$u[i] = math.exp(v[i])$$

→ vector 버전

import numpy as np

$$v = np.array([10, 20, 30])$$

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

+ numpy의 벡터값 함수들

• np.log(v) : element-wise  $\log$

• np.abs(v) : 절대값

• np.maximum(v, 0) : v의 모든 원소의 최대값을 0으로 해줌

• v\*\*2 : element-wise의 제곱

• 1/v : inverse (역행렬)

## Logistic Regression Derivatives $\Rightarrow$ Vectorization

$$J = 0, \quad \cancel{dw_1 = 0}, \quad \cancel{dw_2 = 0}, \quad b = 0 \quad \xrightarrow{\text{dw} = np.zeros((n-x, 1))}$$

for  $i = 1$  to  $m$ :

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

$$\alpha^i = \sigma(z^i)$$

$$J += -[y^i \log \alpha^i + (1-y^i) \log(1-\alpha^i)]$$

파란색으로 바꾸면

중간중간 반복되는 부분을

줄일 수 있다.

전체 for문을 없애는 방법  
이후에 나온다

$$\begin{aligned} \text{for } j = 1 \text{ to } m: \\ dw_1 &+= \dots \\ dw_1 &+= x_1^i dz^i \\ dw_2 &+= x_2^i dz^i \end{aligned} \quad \xrightarrow{\text{dw} += x^i dz^i}$$

$$db += dz^i$$

$$\xrightarrow{\text{dw} /= m}$$

$$J = J/m, \quad dw_1 = dw_1/m, \quad db = db/m$$

•  $\because dz^i$ 의 의미

이전 강의에서  $dz = \alpha(1-\alpha)$  라고 하였는데

이는 축약된 형태입니다.

$$\text{제대로 표기하면 } \frac{da}{dz} = \alpha(1-\alpha) \text{ 입니다.}$$

또, 그 전 강의에서의  $dz^i$ 는

$$\frac{dl}{dz} = \alpha - y \text{ 를 쓰겠습니다.}$$

이를 명확히 하기 위해  $\frac{da}{dz}$  와  $\frac{dl}{dz}$  의 관계를 다음과 같습니다.

$$\frac{dl}{dz} = \frac{dl}{da} \times \frac{da}{dz}$$

$$\frac{dl}{dz} = \frac{\alpha - y}{\alpha(1-\alpha)} \times \alpha(1-\alpha) = \alpha - y$$

## Initialize Parameters with Zeros

```
def initialize_with_zeros(dim):
    w = np.zeros( (dim,1) )
    b = 0.0
    return w, b

dim = 2
w, b = initialize_with_zeros(dim)
assert type(b) == float
print("w =", w)
print("b =", b)
```

```
w = [[0.]
      [0.]]
b = 0.0
```

# Vectorizing Logistic Regression

로지스틱 회귀분석에 vectorization을 적용하여

전체 트레이닝 데이터에 대해 prediction 구하기!

a: activation

① forward propagation

$$\begin{array}{l} \text{설명문자} \\ \text{설명문자} \\ \left( \begin{array}{l} z^1 = w^T x^1 + b \\ a^1 = \sigma(z^1) \end{array} \right) \quad \left( \begin{array}{l} z^2 = w^T x^2 + b \\ a^2 = \sigma(z^2) \end{array} \right) \quad \left( \begin{array}{l} z^3 = w^T x^3 + b \\ a^3 = \sigma(z^3) \end{array} \right) \quad \dots m \geq H \end{array}$$

이 과정을 for문으로  
m번 반복하는 과정  
for loop

for-loop을 m번 a(activation) 구하기!

$$X = \begin{bmatrix} | & | & | \\ x^1 & x^2 & \dots & x^m \\ | & | & & | \end{bmatrix}$$

설명문자  
설명문자  
 $(n_{x,m})$  행렬

① z 구하기

$$(1, m)$$

z의 각 행렬은 예전  
각 행렬은 행렬

$$z = [z^1 \ z^2 \ \dots \ z^m] = w^T X + [b \ b \ \dots \ b]$$

$$\begin{aligned} &= w^T \begin{bmatrix} | & | & | \\ x^1 & x^2 & \dots & x^m \\ | & | & & | \end{bmatrix} \\ &= \underbrace{w^T x^1 + b}_{=z^1} \ \underbrace{w^T x^2 + b}_{=z^2} \ \dots \ \underbrace{w^T x^m + b}_{m \geq H} \end{aligned}$$

② A 구하기

$$A = [a^1 \ a^2 \ \dots \ a^m] = \sigma(z)$$

$$\therefore z = np.dot(w^T, x) + b$$

$(1, 1)$  행렬은 실수.

파이썬은 자동으로 이를

$(1 \times m)$  행렬로 바꾸고,

이를 broadcasting이라고 함

## Vectorizing Logistic Regression

로지스틱 회귀분석에 vectorization을 적용하는 법

전체 트레이닝 데이터에 대해

for-loop 없이 Gradient Descent 적용해보자

먼저 가중치 구하는 과정부터!

$$\text{for } dz' = a^1 - y^1 \quad dz^2 = a^2 - y^2 \quad \dots \quad m \text{ 까지}$$

$$\left. \begin{aligned} dZ &= [dz^1 \ dz^2 \ \dots \ dz^m] \\ &\quad (1, m) \text{ 까지 벡터} \\ A &= [a^1 \ a^2 \ \dots \ a^m] \\ Y &= [y^1 \ \dots \ y^m] \end{aligned} \right\} \Rightarrow \begin{aligned} dZ &= A - Y \\ &= [a^1 - y^1 \ a^2 - y^2 \ \dots] \end{aligned}$$

$$\left. \begin{aligned} \text{for } dw &= 0 \\ dw &+= x^1 dz^1 \\ dw &+= x^2 dz^2 \\ &\vdots \\ dw &+= x^m dz^m \\ dw &/= m \end{aligned} \right\} \Rightarrow \begin{aligned} dw &= \frac{1}{m} X dz^\top \\ &= \frac{1}{m} \begin{bmatrix} | & | & \dots & | \\ x^1 & x^2 & \dots & x^m \\ | & | & \dots & | \end{bmatrix} \begin{bmatrix} dz^1 \\ dz^2 \\ \vdots \\ dz^m \end{bmatrix} \\ &= \frac{1}{m} \underbrace{\left[ x^1 dz^1 + \dots + x^m dz^m \right]}_{(1 \times n)} \end{aligned}$$

$$\left. \begin{aligned} \text{for } db &= 0 \\ db &+= dz^1 \\ db &+= dz^2 \\ &\vdots \\ db &+= dz^m \\ db &/= m \end{aligned} \right\} \Rightarrow \begin{aligned} db &= \frac{1}{m} \sum_{i=1}^m dz^i \\ &= \frac{1}{m} \text{np.sum}(dz) \end{aligned}$$

# Propagate by Python

<공식>

$$A = \sigma(w^T X + b) = [a^1 \ a^2 \ \dots \ a^n]$$

$$\text{cost func } J = -\frac{1}{m} \sum_{i=1}^m (y^i \log a^i + (1-y^i) \log(1-a^i))$$

$$dw = \frac{\partial J}{\partial w} = \frac{1}{m} X(A-Y)^T$$

$$db = \frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^m (a^i - y^i)$$

<매개변수>

w: 가중치 . numpy array

b: bias . 스칼라

X: data의 크기 . (num\_px \* num\_px \* 3, 입력 개수)

Y: 실제 참값 . vector

```
def propagate(w, b, X, Y):
    m = X.shape[1]

    # Forward Propagation (from X to Cost)
    A = sigmoid(np.dot(w.T, X) + b)
    cost = -(1/m) * np.sum((Y*np.log(A) + (1-Y)*np.log(1-A))) )

    # Backward Propagation (to find gradient)
    dw = (1/m) * np.dot(X, (A-Y).T)
    db = (1/m) * np.sum(A-Y)

    cost = np.squeeze(np.array(cost))
    grads = {"dw":dw, "db":db}
    return grads, cost
```

<리턴값>

cost : negative log-likelihood cost  
for logistic regression

dw : W에 대한 loss의 기울기 . w의 동일한 shape

db : b에 " "

Logistic Regression에 이전 계산식 적용하기  $\Rightarrow$  전체 for문도 삭제 가능한 방법

$$J=0, dw_1=0, dw_2=0, b=0$$

방법

$$\text{for } i = 1 \text{ to } m : z^i = w^T x^i + b$$

$$a^i = \sigma(z^i) \rightarrow A = \sigma(z)$$

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

$$\text{for } j = 1 \text{ to } n : dz^i = a^i - y^i \rightarrow dZ = A - Y$$

$$\left. \begin{array}{l} dw_1 += x_1^i dz^i \\ dw_2 += x_2^i dz^i \end{array} \right] \rightarrow dw += x^i * dz^i \Rightarrow dw = \frac{1}{m} X dZ^T$$

$$db += dz^i$$

$$J = J/m, dw_1 = dw/m, dw_2 = dw/m,$$

$$db = db/m$$

$\therefore$  Gradient descent

$$w = w - \alpha \frac{dw}{\text{learning rate}}$$

$$b = b - \alpha \frac{db}{\text{learning rate}}$$

Gradient Descent (반복 정리).

가로(강화) $\frac{1}{2}$  1000번하고 끝나면

이 빨간 상자 $\frac{1}{2}$

for - 1000번 해주어야 함

## Optimize

- cost func  $J$ 를 최소화하는  
 $w$ 와  $b$ 를 학습해보자
- 업데이트 규칙은  $\star = \star - \alpha \frac{\partial J}{\partial \star}$ 을 해보자  
 $\alpha$ 는 learning rate

```
def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009, print_cost=False):
    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        # cost and gradient calculation
        grads, cost = propagate(w, b, X, Y)

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # Update Rule
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)
        # Print the cost every 100 training iterations
        if print_cost:
            print("Cost after iteration %i: %f" % (i, cost))

    params = {"w":w, "b":b}
    grads = {"dw":dw, "db":db}

    return params, grads, costs
```

# Predict

$$\hat{Y} = \sigma(w^T X + b) \quad (= A) \xrightarrow{\approx} \text{예상 결과}$$

$A(\text{activation}) \approx 0.5$ 보다 크면 1, 작으면 0  
 $Y_{\text{prediction}}$ 에 저장

```
def predict(w, b, X):
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Predicting the probabilities of a cat in the picture
    A = sigmoid(np.dot(w.T,X)+b)

    for i in range(A.shape[1]):
        # Convert probabilities A[0,i] to actual predictions p[0,i]
        if A[0,i] > 0.5:
            Y_prediction[0,i] = 1
        else:
            Y_prediction[0,i] = 0

    return Y_prediction
```

Mode | : 이전까지 만든 함수를 모두 사용해서 만들기

```
def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning_rate=0.5, print_cost=False):
    # Initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])
    # Gradient Descent
    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)
    # Retrieve parameters w and b from dictionary "params"
    w, b = params["w"], params["b"]
    # Predict train/test set examples
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_test = predict(w, b, X_test)

    if print_cost:
        print("train accuracy : {} %".format(100-np.mean(np.abs(Y_prediction_train-Y_train))*100))
        print("test accuracy : {} %".format(100-np.mean(np.abs(Y_prediction_test-Y_test))*100))

    d = {"costs": costs,
          "Y_prediction_test": Y_prediction_test,
          "Y_prediction_train": Y_prediction_train,
          "w": w,
          "b": b,
          "learning_rate": learning_rate,
          "num_iterations": num_iterations
         }
    return d
```

training accuracy가 거의 100% 가까이 보인다.

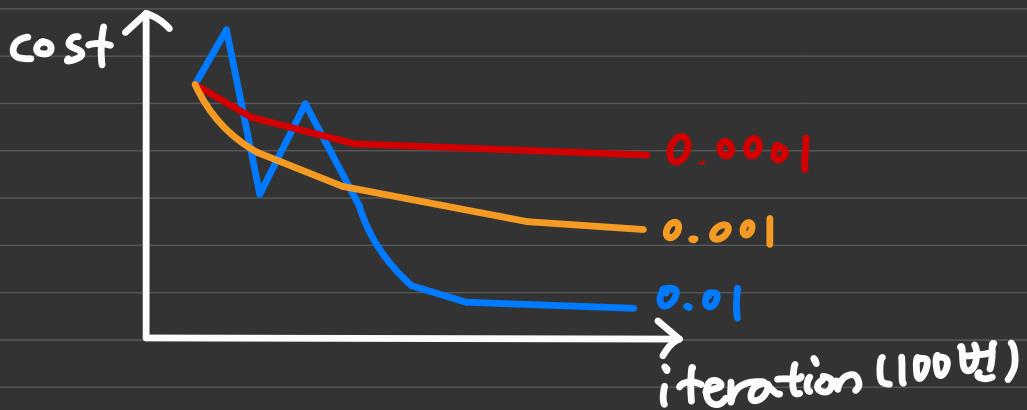
이는 train data의 overfit일 확률이 높다

이제 regularization을 배우면 overfit을 줄일 수 있다

특히 train accuracy는 증가하는데

test accuracy가 감소할 때, 이는 확실히 overfitting이다

## About Learning Rate



- learning rate가 클 수록,  
cost는 진동하며 감소한다.  
심지어 분산될 수도 있다 (수렴X)
- cost가 낮다고 무조건 좋은 모델이 아니다.  
너무 낮다면 overfit인지 확인해보아야 한다  
특히 train accuracy >>> test accuracy일 때!

↳ 이 때는 overfit을 줄이는 것과 관련된 기술은 사용해야 한다

Logistic Regression에 Cost function을 쓰면 좋은 이유

$$\hat{y} = \sigma(w^T x + b) \quad \text{when } \sigma(z) = \frac{1}{1+e^{-z}}$$
$$= P(y=1|x)$$

입력값  $x$ 에 대해  $y=1$ 일 확률을 의미

즉,  $y=1$ 이면  $P(y|x) = \hat{y}$   
 $x$ 가  $\hat{y}$ 일 때의  $y$  확률

$y=0$  일 때  $P(y|x) = 1-\hat{y}$

우의 공식을 바탕으로... 정리하면

$$P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$$
$$\hookrightarrow y=1 \text{ 일 때 } \hat{y}^1 (1-\hat{y})^0 = \hat{y}$$
$$\hookrightarrow y=0 \text{ 일 때 } \hat{y}^0 (1-\hat{y})^1 = 1-\hat{y}$$

이 함수는 단순 증가하기 때문에

최대화를 통해  $P(y|x)$ 를 최적화해보자

↑  
 $\log P(y|x) = \log \hat{y}^y (1-\hat{y})^{(1-y)}$  사실 loss 함수의 마이너스 값이 있으니  
 $= y \log \hat{y} + (1-y) \log (1-\hat{y})$   
 $= -L(\hat{y}, y)$  ↓

∴ loss 함수를 최소화시킨다는 의미는

$\log$ 을 최대화시키는 것과 동일

## Normalizing Rows

normalization을 거치면

Gradient Descent가 빠르게 수렴하여 성능↑

여기서 normalization 이란,

$X \frac{1}{2}$ , 각 가로벡터의  $X$ 를 그  $X$ 의 norm 값으로 나눈 값으로 고쳐하는 것

$$\therefore X \rightarrow \frac{X}{\|X\|}$$

예로  $X = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \end{bmatrix}$  일 때

$$\|X\| = \text{np.linalg.norm}(X, axis=1, keepdims=True)$$

$$= \begin{bmatrix} 5 \\ \sqrt{56} \end{bmatrix} \quad \text{broadcasting으로 원의 크기에 맞춰서 나누기}$$

(~8 데이터)  
 $\begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 6 & 8 & 0 \end{bmatrix}$ 로 저장  
∴ 1 2 3 4 5 6 7 8  
↔  
+1 axis=0  
column-wise  
 $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 1 & 0 \end{bmatrix}$   
∴ 1 5 2 6 3 7 4 8

이 때  $X$ 의 정규화 값은

$$X_{\text{normalized}} = \frac{X}{\|X\|} = \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \end{bmatrix}$$

```
def normalize_rows(x):
    # normalizes each row of the matrix x (to have unit length)
    x_norm = np.linalg.norm(x, axis=1, keepdims=True)
    x = np.divide(x, x_norm) # /= doesn't support vector
    print(x_norm.shape, x.shape)
    return x
    broadcasting

x = np.array([[0,3,4], [1,6,4]])
print("normalizeRows(x) = " + str(normalize_rows(x)) )

(2, 1) (2, 3)
normalizeRows(x) = [[0.          0.6         0.8        ]
[0.13736056 0.82416338 0.54944226]]
```

$X_{\text{norm}}$ 과  $X$ 의 shape가 서로 다르다

그 이유는  $X_{\text{norm}}$ 은  $X$ 의 row vector  $\vec{x}_i$ 의 norm을 데려온 가령이 때문

그래서 row의 수는 서로 같지만,  $X_{\text{norm}}$ 의 column 수는 1이

## Softmax

두 개 이상의 집단으로 분류해야 할 때 사용하는  
normalizing function의 일종

(정의)

- $X$  가  $(1 \times n)$  차원에 속할 때

$$\text{softmax}(x) = \text{softmax}([x_1, x_2, \dots, x_n])$$

$$= \left[ \frac{e^{x_1}}{\sum_j e^{x_j}}, \dots, \frac{e^{x_n}}{\sum_j e^{x_j}} \right]$$

내용  
 $m = \text{train data의 수}$   
이상화된 예제 보기!  
 $M$ : row  
 $n$ : column

- $X$  가  $(m \times n)$  차원에 속할 때

$x$ 의  $i$  번째 가로줄,  $j$  번째 세로줄에 위치한 원소에 맵핑하기

$$\text{softmax}(x) = \text{softmax} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{e^{x_{11}}}{\sum_j e^{x_{1j}}} & \dots & \frac{e^{x_{1n}}}{\sum_j e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_j e^{x_{2j}}} & \dots & \vdots \\ \vdots & & \vdots \\ \frac{e^{x_{m1}}}{\sum_j e^{x_{mj}}} & \dots & \frac{e^{x_{mn}}}{\sum_j e^{x_{mj}}} \end{bmatrix} = \begin{bmatrix} \text{softmax}(x_{11} \text{ first row}) \\ \text{softmax}(x_{12} \text{ second row}) \\ \vdots \\ \text{softmax}(x_{1n} \text{ nth row}) \end{bmatrix}$$

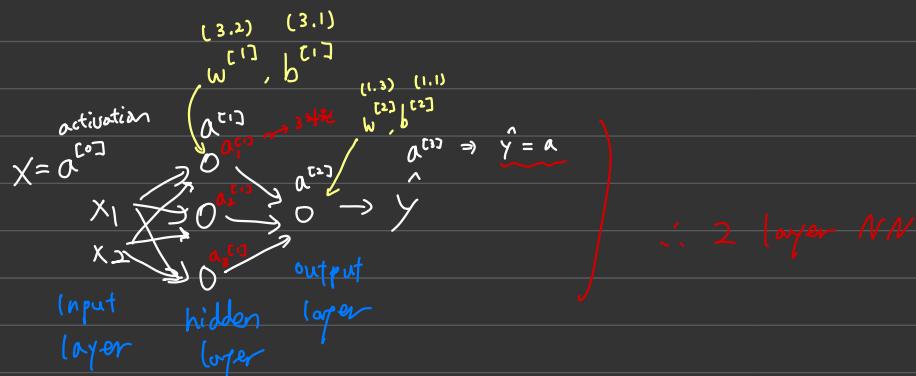
<Shape>

```
def softmax(x):
    (2,5) ← x_exp = np.exp(x)
    (2,1) ← x_sum = np.sum(x_exp, axis=1, keepdims=True) # sum each row of x_exp
    (2,5) ← s = np.divide(x_exp, x_sum) # automatically broadcasting
    return s

t_x = np.array([[0,2,5,0,0], [7,5,0,0,0]])
print("softmax(x) = " + str(softmax(t_x)))

softmax(x) = [[6.29714138e-03 4.65299309e-02 9.34578645e-01 6.29714138e-03
 6.29714138e-03]
 [8.78679856e-01 1.18916387e-01 8.01252314e-04 8.01252314e-04
 8.01252314e-04]]
```

# Neural Network Representation



입력  $X$ 를 표기하는 다른 방법은

$$\alpha^{[0]}$$

activation을 상정

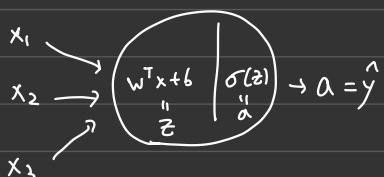
서로 다른 신경망들의 총합

각각 이어지는 신경망 층들로 전달되는 값을 의미

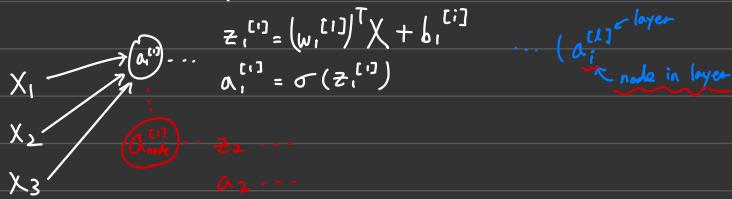
hidden layer에서는 activation을 생성

$$\alpha^{[1]}$$

$\therefore$  정리



# Hidden Layer



node가 4개일 때

∴ 계산식 정리

$$\begin{aligned} z_1^{(l)} &= (w_1^{(l)})^T x + b_1^{(l)} \\ z_2^{(l)} &= (w_2^{(l)})^T x + b_2^{(l)} \\ z_3^{(l)} &= (w_3^{(l)})^T x + b_3^{(l)} \\ z_4^{(l)} &= (w_4^{(l)})^T x + b_4^{(l)} \end{aligned}$$

$$\hat{y} = \alpha_1^{(l)} = \sigma(z_1^{(l)})$$

$$\alpha_2^{(l)} = \sigma(z_2^{(l)})$$

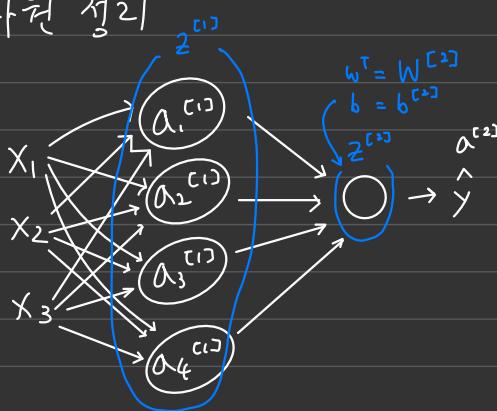
$$\alpha_3^{(l)} = \sigma(z_3^{(l)})$$

$$\alpha_4^{(l)} = \sigma(z_4^{(l)})$$

$$\therefore \underline{\underline{z}} = \underbrace{\begin{bmatrix} -(w_1^{(l)})^T \\ -(w_2^{(l)})^T \\ -(w_3^{(l)})^T \\ -(w_4^{(l)})^T \end{bmatrix}}_{W^{(l)}} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \underbrace{\begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ b_3^{(l)} \\ b_4^{(l)} \end{bmatrix}}_b$$

$$\alpha^{(l)} = \begin{bmatrix} \alpha_1^{(l)} \\ \vdots \\ \alpha_4^{(l)} \end{bmatrix} = \sigma(\underline{\underline{z}})$$

차원 정리



X가 3개일 때

$$\underline{\underline{z}}^{(l+1)} = W^{(l+1)} \alpha^{(l)} + b^{(l+1)}$$

$$\alpha^{(l+1)} = \sigma(z^{(l+1)})$$

로짓스틱 회귀 분석의  
hidden layer의  
각 node가 하는 일

$$\underline{\underline{z}}^{(l+1)} = W^{(l+1)} \alpha^{(l)} + b^{(l+1)}$$

$$\alpha^{(l+1)} = \sigma(z^{(l+1)})$$

output layer의  
Regression

∴ 4개로 정리 가능!

```
def layer_sizes(X, Y):
    n_x = X.shape[0] # the size of input layer
    n_h = 4 #the size of the hidden layer
    n_y = Y.shape[0] # the size of output layer
    return n_x, n_h, n_y

def initialize_parameters(n_x, n_h, n_y):
    # seed를 설정 -> your output matches ours although the initialization is random
    np.random.seed(2)

    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))

    parameters = {"W1": W1, "b1":
                  "W2": W2, "b2": b2}
    return parameters
```

(m)  
여러 개의 데이터에 적용시키기/ 위한 Vectorizing

이전 공식을 살짝 변형하여 모든 샘플에서 결과값을 산출할 수 있도록 해보자

$$\begin{aligned}
 & X \longrightarrow a^{[1]} = \hat{y} \\
 & \left( \begin{array}{l} X^{(1)} \longrightarrow a^{[2](1)} = \hat{y}^{(1)} \\ \vdots \\ X^{(m)} \longrightarrow a^{[2](m)} = \hat{y}^{(m)} \end{array} \right) \Rightarrow \begin{array}{l} \text{for } i=1 \text{ to } m, \\ z^{[1](i)} = W^{[1](i)} a^{[0](i)} + b^{[1](i)} \\ a^{[1](i)} = \sigma(z^{[1](i)}) \\ z^{[2](i)} = W^{[2](i)} a^{[1](i)} + b^{[2](i)} \\ a^{[2](i)} = \sigma(z^{[2](i)}) \end{array}
 \end{aligned}$$

여러 개의 데이터를 세로로 병합한  
train data를 사용하는 경우

$$X = \begin{bmatrix} | & | & | \\ X^{(1)} & \dots & X^{(m)} \\ | & & | \end{bmatrix} \text{ 일 때, } z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & \dots & z^{[1](m)} \\ | & & | \end{bmatrix}$$

그리고 A 행렬은  
train data를 가로로 indexing 함  
그래서 가로 인덱스가  
train set을 의미

$$\begin{cases} z^{[1]} = W^{[1]} X^{[0]} + b^{[1]} \\ A^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]} A^{[1]} + b^{[2]} \\ A^{[2]} = \sigma(z^{[2]}) \end{cases}$$

• | 때  $X^{[0]}$  을 다시금 적용해보면  
결국, 첨자의 index가 증가하여 반복되는 과정임을 알 수 있다

# Activation Functions

ex) sigmoid

앞의 공식에 sigmoid 대신  
다른 activation func 적용할 수 있다

+1) sigmoid의 뿐이가

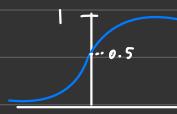
0 ~ 1 사이에서 이진분류에 적합

output이 0.5보다 작을 때 0으로  
크면 1로 분류

tanh도 좋지만 범위가 -1 ~ 1이라  
불편

sigmoid

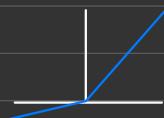
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Leaky

ReLU

$$\max(0.1x, x)$$

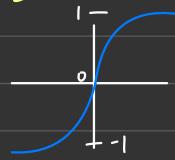


만수도 표현이 되기 어려워  
sigmoid보다 나음

tanh

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

z가 너무 커지면 기울기가 0



Maxout

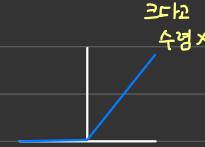
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Rectified

linear unit

ReLU

$$\max(0, x)$$



두 가지 분류해야 할 때 벌조임

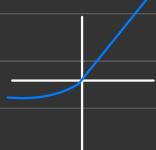
실제로 음수일 때는 다른 일어

많이 없어서

ReLU가

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



∴ 사실 모든 힘스를 적용해보며

실행해보셔야 힘^^

# 비선형 Activation function이 필요한 이유

activation function이 선형이라면,

$f(z) = z$ 로 표현할 수 있다.

이를 공식에 적용하면,

$$\begin{cases} z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[1]} = z^{[1]} \\ z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \\ a^{[2]} = z^{[2]} \end{cases}$$

$$\begin{aligned} a^{[1]} &= z^{[1]} = W^{[1]}x + b^{[1]} \\ a^{[2]} &= z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \end{aligned} \quad \text{이다}$$

정리하면

$$a^{[2]} = \underbrace{(W^{[2]}W^{[1]})}_{W'}x + \underbrace{(W^{[2]}b^{[1]} + b^{[2]})}_{b'}$$

$$= W'x + b' \quad \text{이다}$$

이려면 신경망이 여러 개의 층으로 구성되어도

결국의 일차원의 계산이

선형 방식으로 구현되기 때문에

이를 방지하기 위해

non-linear activation func 사용

```
def forward_propagation(X, parameters):
    W1 = parameter["W1"]
    b1 = parameter["b1"]
    W2 = parameter["W2"]
    b2 = parameter["b2"]

    Z1 = np.dot(W1,X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2,A1) + b2
    A2 = sigmoid(Z2)

    assert (A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1, "A1": A1,
              "Z2": Z2, "A1": A1}

    return A2, cache

def compute_cost(A2,Y):
    m = Y.shape[1]
    logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2),1-Y)
    cost = -(1/m) * np.sum(logprobs)

    # make sure cost is the demension we expect - [[17]]->17
    # np.squeeze() to remove redundant dimensions
    # You can also cast the array as a type float using float()
    cost = float(np.squeeze(cost))

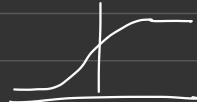
    return cost
```

## Derivatives of Activation Functions

Back propagation  $\curvearrowleft$  activation func의 기울기가 필요

$\langle \text{sigmoid} \rangle$

$$\frac{1}{1+e^{-z}} = \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right)$$



$$= \sigma(z) (1 - \sigma(z)) \Rightarrow$$

기울기가 0에 가깝고

기울기가 0에서 멀다는

사실을 알 수 있다

$\langle \tanh \rangle$



$$(\tanh)' = 1 - (\tanh(z))^2$$

0과 가까워질수록

activation의 기울기는 커진다

반면에 0에 가까울 때는 0이다

$\langle \text{ReLU} \rangle$

$$(\max(0, z))' = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

$\langle \text{Leaky ReLU} \rangle$

$$(\max(0.1z, z))' = \begin{cases} 0.1 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

## (Binary Classification)

# Gradient Descent for Neural Networks

## For One hidden layer

기울기 강화의 back propagation의 작동방식(공식) 설명

$$\eta_x = n^{[0]} \sim n^{[1]} \Rightarrow 1 \text{ output } i \text{ 쪽을}$$

Gradient Descent :

반복 {

• 매개변수 :  $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$   
 shape:  $(n^{[1]}, n^{[2]})$        $(n^{[2]}, n^{[1]})$        $(n^{[2]}, 1)$   
 $\quad \quad \quad (1, 1)$

• Cost func :  $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]})$

$$= \frac{1}{m} \sum_{i=1}^m L(\hat{y}_i, y_i)$$

predict 예측 :  $\hat{y}^{(i)} \quad (i=1 \sim m)$

이 계산에 대한  
구식은 성장하는  
 $\frac{dw^{[1]}}{db^{[1]}} = \frac{dJ}{dw^{[1]}}$       ... , 1, 2, ...

$$\begin{cases} w^{[1]} = w^{[1]} - \alpha dw^{[1]} \\ b^{[1]} = b^{[1]} - \alpha db^{[1]} \end{cases} \quad \dots, 1, 2, \dots$$

derivatives 미분 계산

먼저

Forward propagation  $\frac{\partial J}{\partial z}$ 을 보자

$$z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(z^{[1]})$$

$$z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]})$$

Back Propagation

$$dz^{[2]} = A^{[2]} - Y$$

$$dw^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} np.sum(dz^{[2]}, axis=1, keepdims=True)$$

$$dz^{[1]} = \underbrace{W^{[2]T} dz^{[2]} \otimes}_{(n^{[2]}, m)} \underbrace{\sigma'(z^{[1]})}_{(n^{[1]}, m)}$$

element-wise  $\otimes$       keepdims=True  
 (n,)인 rank 1 array      (n, 1)로 바꿈

$$dw^{[1]} = \frac{1}{m} dz^{[1]} X^T$$

output layer  $L_1$       hidden layer  $L_2$   
 사용한 activation 함수의 미분값.

$$db^{[1]} = \frac{1}{m} np.sum(dz^{[1]}, axis=1, keepdims=True)$$

여기서 sigmoid가 이전 풀 출력을 가정.

이전에는  $b$ 가 실수인  $(1, 1)$ 이라 원활하지만  
 이제는  $(n^{[1]}, 1)$ 의 벡터가 되는데  
 꼭 적기!

Y는

모든 모기의 데이터는  
 가로로 정렬한 상태  
 $Y = [y^{(1)} \dots y^{(m)}]$

```
def backward_propagation(parameters, cache, X, Y):
    m = X.shape[0]

    W1 = parameters["W1"]
    W2 = parameters["W2"]

    A1 = cache["A1"]
    A2 = cache["A2"]

    dZ2 = A2-Y
    dW2 = (1/m) * np.dot(dZ2, A1.T)
    db2 = (1/m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (1-np.power(A1,2))
    dW1 = (1/m) * np.dot(dZ1, X.T)
    db1 = (1/m) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": W1, "db1": db1,
              "dW2": W2, "db2": db2}

    return grads
```

```
def update_parameters(parameters, grads, learning_rate = 1.2):
    W1 = copy.deepcopy(parameters["W1"])
    b1 = copy.deepcopy(parameters["b1"])
    W2 = copy.deepcopy(parameters["W2"])
    b2 = copy.deepcopy(parameters["b2"])

    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    # Update rule for each parameters
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {"W1": W1, "b1": b1,
                  "W2": W2, "b2": b2}

    return parameters
```

# Random Initialization

신경망을 변경할 때 weight를 임의로 초기화하는 것은 중요

로지스틱 회귀에서 weight = 0 초기화는 랜덤이지만

모든 신경망에서 그렇게 하면 가중치 강화에서 이상해집니다

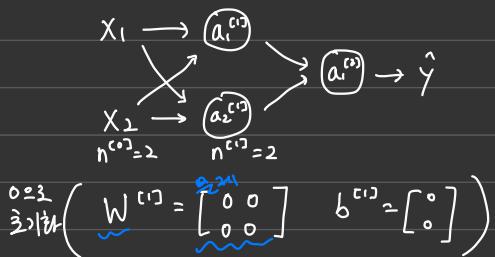
왜 그런지 살펴보자

로지스틱 회귀는 hidden layer가 있어서.

로지스틱 회귀의 기본값은

x가 상수 벡터가 아니라면

0이 아님으로



$$\text{if } a_1^{(1)} = a_2^{(1)},$$

back prop에서

$$dZ_1^{(1)} = dZ_2^{(1)} \text{ 이 된다}$$

$$\text{이로 인해 } W^{(2)} = [0 \ 0]$$

$$\text{그럼 } dW^{(1)} = \begin{bmatrix} \Delta & \Delta \\ \Delta & \Delta \end{bmatrix} \text{ 은 즐마라 같은 값을 가진다}$$

이 때 W를 update하면

$$W^{(1)} = W^{(1)} - \alpha dW^{(1)} \text{ 예시로 }$$

$$W = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ 는 첫번째 줄과 두번째 줄이 같아질 때}$$

모든 W값을 0으로 하면,

그 때의 hidden node는

모든 출력은 활성화 함수를 산출하고

이는 output unit에 같은 영향을 주기 때문에

반복하면 두 개의 X에 대해

모든 차트로 나와버린다  
계속 같은 식, 같은 나오면 hidden node를

여러 개로 두 이유가  
무엇이 때문이다

## Random Initialize

앞의 수식에 대비

$$W^{(1)} = np.random.randn(2, 2) * 0.01$$

인 이유

$$b^{(1)} = np.zeros(2, 1)$$

만약 weight 너무 크면

$$W^{(2)} = np.random.rand(1, 2) * 0.01$$

~~f~~ tanh 만 생각하세요

$$b^{(2)} = 0$$

$Z = W^{(1)}x + b^{(1)}$  가 너무 커져서

계산이 많아 속도가 느려지는 단점..

# NN\_model in Python

```
def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):
    # n_h : nums of hidden layers
    # Too many n_h , it will be overfitting

    np.random.seed(3)
    n_x = layer_sizes(X,Y)[0]
    n_y = layer_sizes(X,Y)[2]

    parameters = initialize_parameters(n_x, n_h, n_y)

    # Loop (gradient descent)
    for i in range(0, num_iterations):
        # Forward propagation
        A2, cache = forward_propagation(X, parameters)
        # Cost function
        cost = compute_cost(A2, Y)
        # Back propagation
        grads = backward_propagation(parameters, cache, X, Y)
        # Gradient Descent parameter update
        parameters = update_parameters(parameters, grads, learning_rate=1.2)

        if print_cost and i % 1000 == 0:
            print("Cost after iteration %i: %f" % (i, cost))

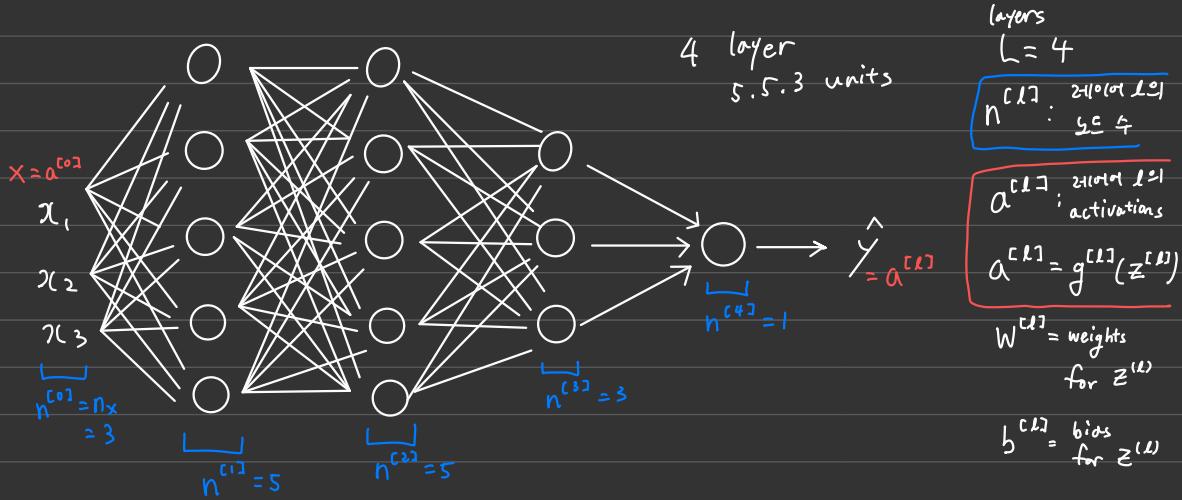
    return parameters
```

```
def predict(parameters, X):
    A2, cache = forward_propagation(X, parameters)
    predictions = A2 > 0.5
    return predictions
```

```
p = predict(parameters, X)
print('Accuracy: %d' % float((np.dot(Y, p.T)+np.dot(1-Y, 1-p.T))/float(Y.size)*100)+"%)
```

# Deep Neural Network

hidden layer 수가 훨씬 많아진 Network



```
def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(1) # random의 무작위성을 위해
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h,1)) * 0.01
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y,1))

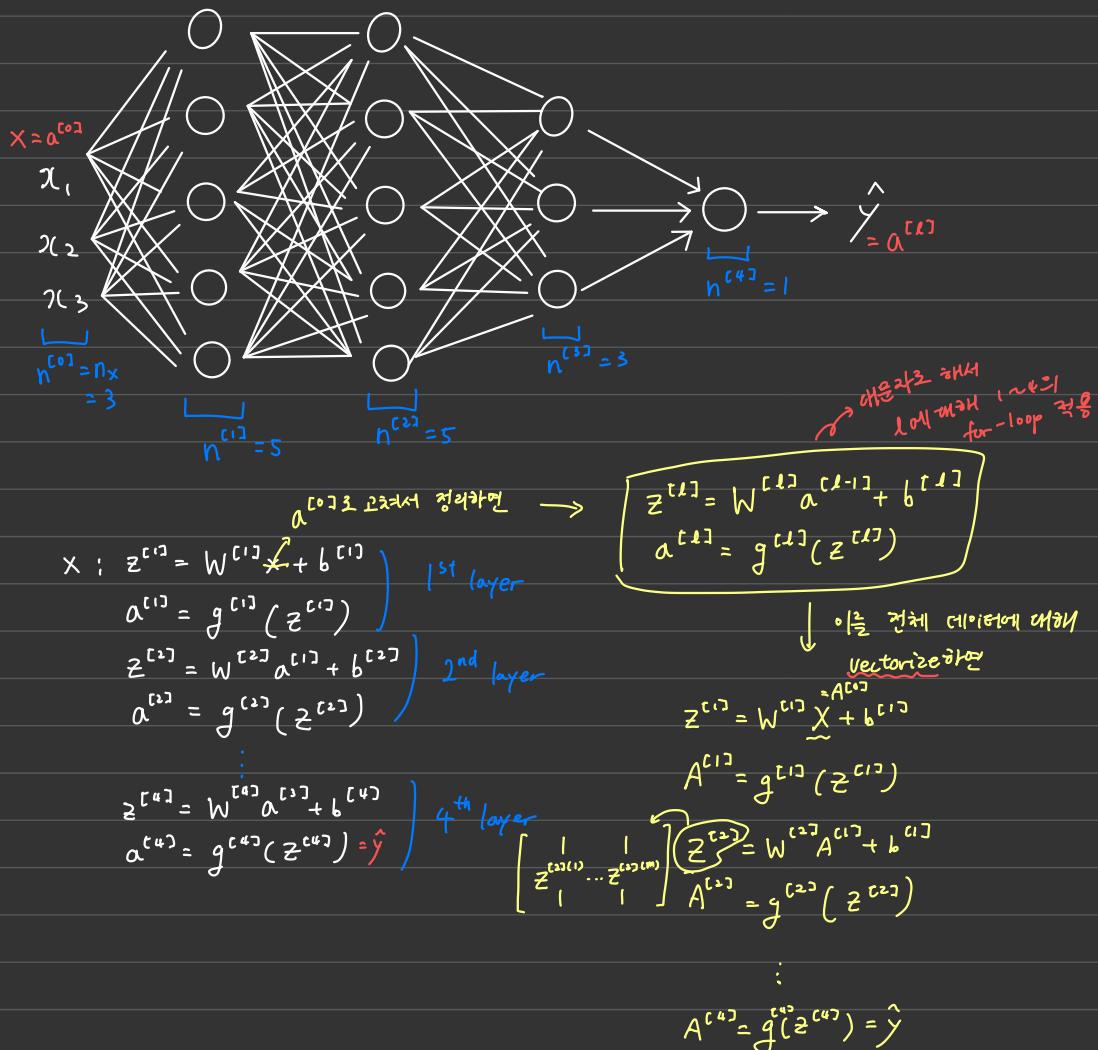
    parameters = {"W1":W1, "b1":b1, "W2":W2, "b2":b2}
    return parameters
```

```
def initialize_parameters_deep(layer_dims):
    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)

    for l in range(1, L):
        parameters["W"+str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) * 0.01
        parameters["b"+str(l)] = np.zeros((layer_dims[l], 1)) * 0.01
    return parameters
```

# Forward Propagation in a deep Network

1개의 train data 인 x에 대해서 알아보기



```
def linear_forward(A,W,b):
    # Implement the linear part of layer's forward propagation
    Z = np.dot(W,A) + b
    cache = (A, W, b)
    return Z, cache
```



```
def linear_activation_forward(A_prev, W, b, activation):
    # Implement the forward propagation
    # for linear->activation layer

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)
    return A, cache
```



```
def L_model_forward(X, parameters):
    # Implement forward propagation
    # for [linear->relu]*(L-1) -> linear -> sigmoid computation
    caches = []
    A = X
    L = len(parameters) // 2

    # Implement [linear -> relu]*(L-1)
    for l in range(1, L): # loop starts at 1 because layer 0 is the input
        A_prev = A
        A, cache = linear_activation_forward(A_prev,
                                              parameters["W"+str(l)],
                                              parameters["b"+str(l)],
                                              "relu")

    # Implement [linear -> sigmoid]
    AL = linear_activation_forward(A,
                                   parameters["W"+str(L)], parameters["b"+str(L)], "sigmoid")
    caches += (cache,)

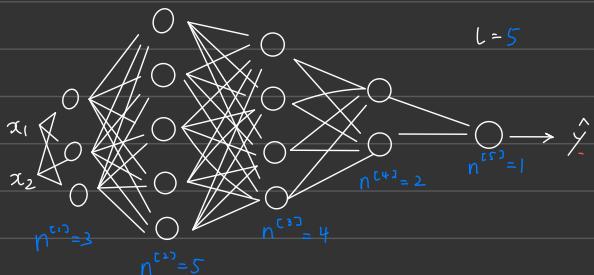
    return AL, caches
```



```
def compute_cost(AL, Y):
    m = Y.shape[1]

    # compute loss from AL and Y
    cost = -(1/m) * np.sum( np.multiply(Y, np.log(AL)) + np.multiply(1-Y, np.log(1-AL)))
    cost = np.squeeze(cost) # To make sure your cost's shape ([17])->17
    return cost
```

# Getting your Matrix Dimensions Right



< Parameters  $W^{[l]}, b^{[l]}$  >

먼저  $Z, W, X$ 의 차원을 고려해보면

그는 첫번째 hidden layer의 activation을 결과로  
(3, 1)이다

$X$ 는 2개가 있으니 (2, 1)이다

그럼  $\begin{bmatrix} Z \\ \vdots \end{bmatrix} = \begin{bmatrix} W \end{bmatrix} \times \begin{bmatrix} X \end{bmatrix}$  가 되어야 한다  
 $W$ 는 (3, 2)이 되어야 한다  
(3, 1) - (n<sup>[1]</sup>, 1)

즉,  $Z^{[1]} = W^{[1]}X + b^{[1]}$   
(n<sup>[1]</sup>, 1) (n<sup>[2]</sup>, n<sup>[1]</sup>) (n<sup>[1]</sup>, 1)이다

이 나아가

$W^{[1]} : (n^{[1]}, n^{[0]})$ 이므로

$W^{[2]} : (5, 3)$ 은  $(n^{[2]}, n^{[1]})$ 이다

마지막  $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ 는  
(5, 1) (5, 3) (3, 1) + (n<sup>[1]</sup>, 1) ... (n<sup>[2]</sup>, 1)

그럼  $W^{[3]} = (4, 5)$ ,  $W^{[4]} = (2, 4)$

,  $W^{[5]} = (1, 2)$ 이다

여기 계산에서  $b$ 를 추적해보면

$b^{[4]} = (n^{[4]}, 1)$ 이 된다

Back propagation 적용 시에

$dW^{[l]} = (n^{[l]}, n^{[l-1]})$

$db^{[l]} = (n^{[l]} - 1)$  이 되어야 함

여러 개의 데이터에서

근데  $X$ 의 차원은 달라지니,

그 경우 살펴보면,

$$\underbrace{Z^{[1]}}_{(n^{[1]}, m)} = \underbrace{W^{[1]}}_{(n^{[1]}, n^{[0]})} \times \underbrace{X}_{(n^{[0]}, m)} + \underbrace{b^{[1]}}_{(n^{[1]}, m)}$$

즉,  $X, b$ 의 두번째 차원의 값이  $m$ 으로 변한다

이 때  $A^{[0]} = X$ 이므로

$dA^{[0]} = (n^{[0]}, m)$ 이다

또한  $dZ^{[l]}, dA^{[l]} : (n^{[l]}, m)$ 이 된다

$\therefore W^{[l]} = (n^{[l]}, n^{[l-1]})$

```

def linear_backward(dZ, cache):
    # Implement the linear portion of backward propagation
    # for a sing layer (layer l)

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = (1/m) * np.dot(dZ, A_prev.T)
    db = (1/m) * np.sum(dZ, axis=1, keepdims=True) # sum by rows
    dA_prev = np.dot(W.T, dZ)

    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    # Implement the backward propagation
    # for linear -> activation layer

    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        # to help you implement, the func have been provided
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    # Implement the backward propagation
    # for [linear->relu]*(L-1) -> linear -> sigmoid group
    grads = {}
    L = len(caches) # number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after thisline, Y has same shape as AL

    # Initializing the back propagation
    dAL = -(np.divide(Y,AL) - np.divide(1-Y,1-AL))

    # Lth layer (sigmoid->linear) gradients
    current_cache = caches[L-1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_cache, "sigmoid")
    grads["dA"+str(L-1)] = dA_prev_temp
    grads["dW"+str(L)] = dW_temp
    grads["db"+str(L)] = db_temp

    # Loop from l = L-2 to 0
    for i in range(L-2, -1, -1):
        # lth layer : [relu -> linear] gradients
        current_cache = caches[i]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA"+str(i+1)],
                                                                    current_cache, "relu")

        grads["dA"+str(i+1)] = dA_prev_temp
        grads["dW"+str(i)] = dW_temp
        grads["db"+str(i)] = db_temp

    return grads

def update_parameters(params, grads, learning_rate):
    # update parameters using gradient descent

    parameters = params.copy()
    L = len(parameters) // 2

    # Update rule for each parameter. Use a for loop
    for i in range(L):
        parameters["W"+str(i+1)] = parameters["W"+str(i+1)] - learning_rate * grads["dW"+str(i+1)]
        parameters["b"+str(i+1)] = parameters["b"+str(i+1)] - learning_rate * grads["db"+str(i+1)]

    return parameters

```

왜 심층신경망이 여러 문제에서 같은 신경망보다 잘 작동할까?

① 심층신경망은 무엇을 계산하는가

ex) 얼굴 사진을 넣었다면

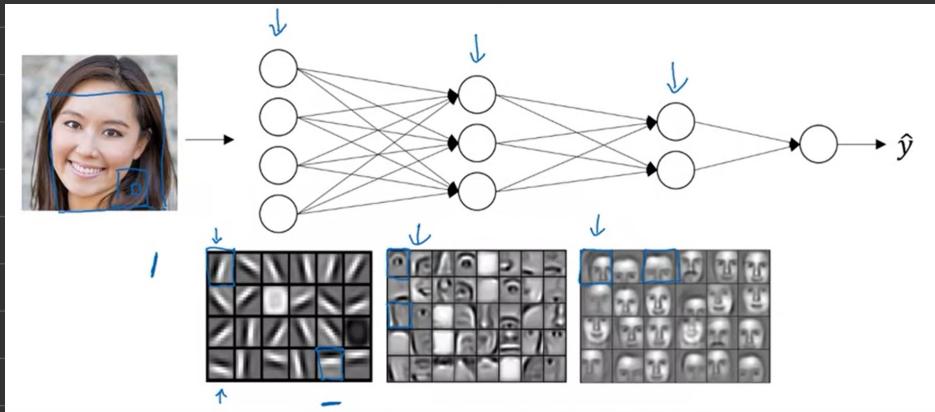
첫번째 layer는 사진을 각각 다른 하나의 batch 속에서

문서리가 어디쯤 있는지 알아낸다

두번째 layer에서는

눈,코,입 등을 찾아내고

세번째 layer는 그것들이 얼굴의 어디에 위치한지 알아낸다



ex) 모티오

1단계 : 저음, 고음 영역 강조를 통한 Wave 강조

또는 백색소음 인지

2단계 : 기본적인 음성 단위 phoneme 인지

3단계 : phoneme를 측정하여 단어 → 문장 → 문구 인식

왜 심층신경망이 여러 문제에서 얇은 신경망보다 잘 작동할까?

## ② Circuit theory and deep Learning

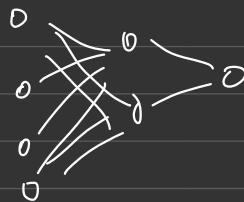
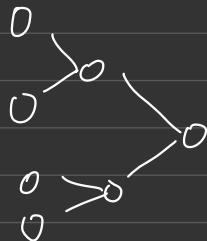
비공식적으로, 이 함수들은 "작지안" 같은 network를 계산.

여기서 "작아"는 숨겨진 노드가 비교적 적다는 뜻.

같은 함수를 얕은 네트워크로 계산하면,

즉, hidden layer가 충분하지 않으므로

계산하기 위해서 hidden layer마다 아주 많은 노드가 필요하다



이제 circuit의 더 효율적인지도...  
theory

# Building Blocks of Deep Neural Networks

## Forwards and Backwards Functions

Layer  $l$ :  $W^{[l]}, b^{[l]}$

Forward  
Input:  $a^{[l-1]}$   
output:  $a^{[l]}$   
Cache:  $z^{[l]}$

$$z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(z^{[l]})$$

$$= dA^{[l]}$$

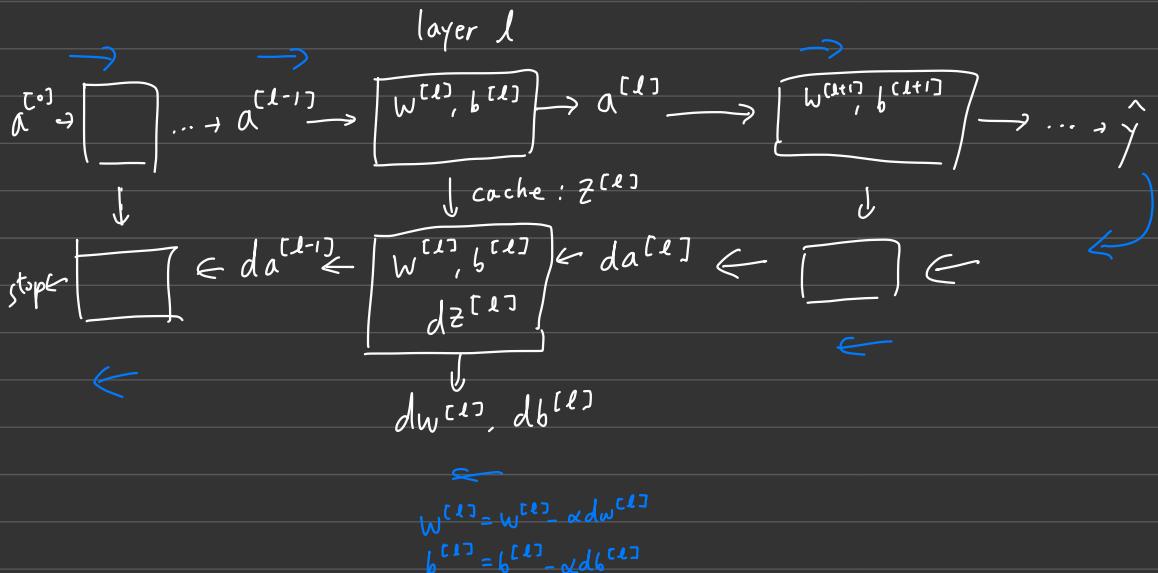
Backward  
Input:  $da^{[l]}$   
output:  $da^{[l-1]}, \dots, dW^{[2]}, db^{[2]}$   
Cache:  $z^{[l]}$

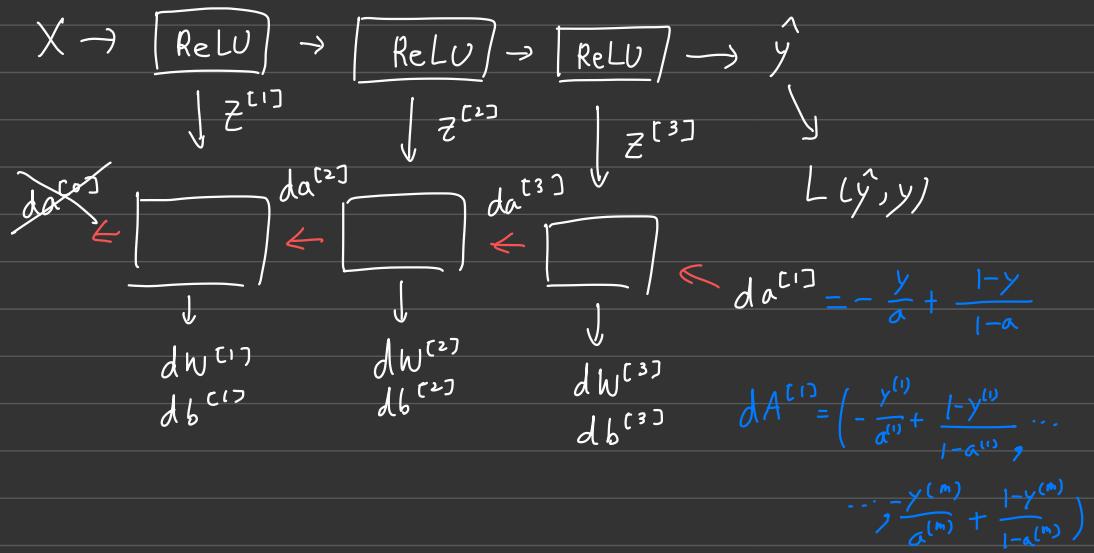
$$dz^{[l]} = (W^{[l+1]})^T dz^{[l+1]} \times g^{[l]}'(z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dz^{[l]} (A^{[l-1]})^T$$

$$db^{[l]} = \frac{1}{m} \text{np.sum}(dz^{[l]}, \sim)$$

$$dA^{[l-1]} = (W^{[l]})^T dz^{[l]}$$





# Hyper parameter

ex) learning rate  $\alpha$   
num of iterations  
hidden layer  $L$   
hidden units  $n^{(0)}, n^{(1)}, \dots$   
activation func 선택

이들은  
parameter인  
 $w, b$ 을 결정짓기(?)에  
hyperparameter라고 함

- + ) momentum
- mini batch size
- regularizations ---

hyperparameter를 다양하게 조정해보면

cost가 작아지는 확인하는 과정을  
수없이 많이 하셔야 합니다.  
업무를 진행하는 중에서도 수정해야 하는 경우가 생깁니다  
더 좋은 hyperparameter가 무엇인지 알아가기 위해 원래  
CPU, GPU, Network가 정해져 있지 않기에  
정답은 없으니 다양히 시도해보세요

```

import numpy as np
import matplotlib.pyplot as plt
import h5py

def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ;
    stored for computing the backward pass efficiently
    """

    A = np.maximum(0,Z)
    assert(A.shape == Z.shape)

    cache = Z
    return A, cache

def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiency

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiency

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ

def initialize_parameters(n_x, n_h, n_y):
    """
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    W1 = np.random.randn(n_h, n_x)*0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.01
    b2 = np.zeros((n_y, 1))

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters

def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
                    WL -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(1)
    parameters = {}
    L = len(layer_dims)           # number of layers in the network

    for l in range(1, L):
        parameters["W" + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) / np.sqrt(layer_dims[l-1])
        parameters["b" + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters["W" + str(l)].shape == (layer_dims[l], layer_dims[l-1]))
        assert(parameters["b" + str(l)].shape == (layer_dims[l], 1))

    return parameters

def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation parameter
    cache -- a python dictionary containing "A", "W" and "b"; stored for computing the backward pass efficiently
    """

    Z = W.dot(A) + b

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache

def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR>ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation value
    cache -- a python dictionary containing "linear_cache" and "activation_cache";
            stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache

```

```

def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]**(L-1)->LINEAR->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
        every cache of linear_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
        the cache of linear_sigmoid_forward() (there is one, indexed L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2           # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation = "relu")
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], activation = "sigmoid")
    caches.append(cache)

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches

def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat))

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    # Compute loss from al and y.
    cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect
    assert(cost.shape == ())

    return cost

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (A_l, W_l, b_l, Z_l, A_{l-1}) given the gradients of the cost with respect to Z_l dZ_l.

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_{prev}, W, b) coming from the forward propagation

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_{prev}
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1./m * np.dot(dZ,A_prev.T)
    dW = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db

def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward propagation for current layer l
    activation -- the activation to be used in this layer, stored as a text string ("relu" or "sigmoid")

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as dA
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """

    linear_cache, activation_cache = cache
    dA_prev, dW, db = linear_backward(dA, linear_cache)

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db

def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR computation

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forward)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
        every cache of linear_activation_forward() with "relu" (there are L-1 such caches, indexed 1 to L-2)
        the cache of linear_activation_forward() with "sigmoid" (there is one, indexed L-1)
    Returns:
    grads -- A dictionary with the gradients
        grads["dA" + str(l)] = ...
        grads["dW" + str(l)] = ...
        grads["db" + str(l)] = ...
    """

    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1-Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "current_cache = caches[L-1]"
    grads["dA" + str(L-1)] = dAL
    grads["dW" + str(L)] = linear_activation_backward(dAL, caches[L-1], activation = "sigmoid")

    for l in reversed(range(L-1)):
        # lth layer: (RELU) -> LINEAR gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l+1)], current_cache, activation = "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l+1)] = dW_temp
        grads["db" + str(l+1)] = db_temp

    return grads

def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_backward
    Returns:
    parameters -- python dictionary containing your updated parameters
        parameters["W" + str(l)] = ...
        parameters["b" + str(l)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network
    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate * grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * grads["db" + str(l+1)]

    return parameters

def predict(X, y, parameters):
    """
    This function is used to predict the results of a L-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model
    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2 # number of layers in the neural network
    p = np.zeros((1,m))

    # Forward propagation
    probas, caches = L_model_forward(X, parameters)
    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    #print results
    #print ("predictions: " + str(p))
    #print ("true labels: " + str(y))
    print("Accuracy: " + str(np.sum((p == y))/m))

    return p

```

```

# each image is size of (64,64,3)
# Reason to set flatten, X has m examples. each example is one column vector
# That why we have to reshape 64,64,4 to 1 line vector
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T
test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T
# '-1' makes reshape flatten the remaining dimensions

# Standardize data to have feaeture values between 0 and 1
train_x = train_x_flatten/255
test_x = test_x_flatten/255

# shape
# train_x : (12288, 209) #12288 = 64*64*3

# 2-layer NN
n_x = 12288
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
learning_rate = 0.0075

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
    np.random.seed(1)
    grads = {}
    costs = {} # to keep track of the cost
    m = X.shape[1] # number of examples
    (n_x, n_h, n_y) = layers_dims

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    for i in range(0, num_iterations):
        # Forward propagation: linear->relu->linear->sigmoid
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")

        # Compute cost
        cost = compute_cost(A2, Y)

        # Initializing backward propagation
        dA2 = -(np.divide(Y, A2) - np.divide(1-Y, 1-A2))

        # Backward propagation
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigmoid")
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")

        grads["dW1"] = dW1
        grads["db1"] = db1
        grads["dW2"] = dW2
        grads["db2"] = db2

        # Update parameters
        parameters = update_parameters(parameters, grads, learning_rate)
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        if print_cost and i%100==0 or i==num_iterations-1:
            print("Cost after iteration {}: {}".format(i,np.squeeze(cost)))
    if i%100 == 0 or i == num_iterations:
        costs.append(cost)

    return parameters, costs

def L_layer_Deep_NN(X, Y, layers_dims, learning_rate=0.0075, num_iterations=3000, print_cost=False):
    np.random.seed(1)
    costs = []
    parameters = initialize_parameters_deep(layers_dims)

    for i in range(num_iterations):
        AL,caches = L_model_forward(X, parameters)
        cost = compute_cost(AL, Y)
        grads = L_model_backward(AL, Y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)

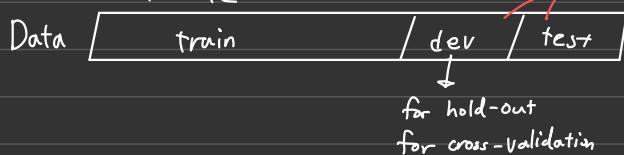
        if print_cost and i%100==0 or i==num_iterations-1:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if i%100==0 or i==num_iterations:
            costs.append(cost)

    return parameters, costs

```

# How to Select Train/dev/test Sets

## ① 각 데이터의 비율



이 비율은  $60/20/20$ 으로 시작해보는 게 좋음  
 $70/0/30$

그런데 1,000,000보다 data가 많다면

$98/1/1$  만 해도 충분함  
 $99.5/0.25/0.25$  ) 이런 경우도  
 $99.5/0.4/0.1$  ) 존재함



## ② 각 데이터 별 수집 방식

train set은 웹에서 긁어온 고양이 사진

dev set은 사용자가 카메라로 직접 찍은 사진

이 때 test set은 dev set 크기와 동일하면 좋음

## ③ test set이 없어도 고민할 수 있다

test set의 목표는 마지막 네트워크에서 bias 없는 성능 추정치를 제공하는 것  
만약 bias 추정치가 필요하지 않다면 필요X

만약 dev set만 있다면

train 속에 다른 모델 구조를 시도해 봄

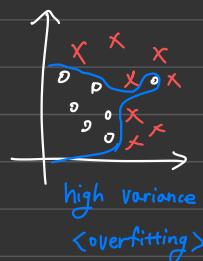
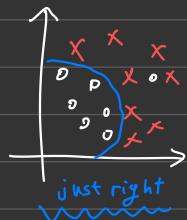
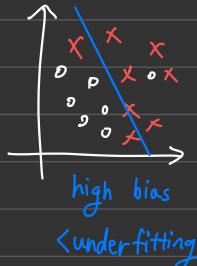
데이터를 dev set에 맞춰가며 더 이상 성능에 애쓰는

bias 없는 추정치를 주지 않음

보통 이 때의 dev set을 test set이라고 부른다

전략

## Bias / Variance



안목

train set error: 1%  
dev set error: 11% ] =  $\frac{\text{high variance}}{\text{variance}}$  = overfitting  $\rightarrow$  Increasing the regularization parameter lambda

$\frac{15\%}{16\%}$  ] =  $\frac{\text{high bias}}{\text{bias}}$  = underfitting

$\frac{15\%}{30\%}$  ] =  $\frac{\text{high bias}}{\text{high variance}}$   $\rightarrow$

$\frac{0.5\%}{1\%}$  ] =  $\frac{\text{low bias}}{\text{low variance}}$

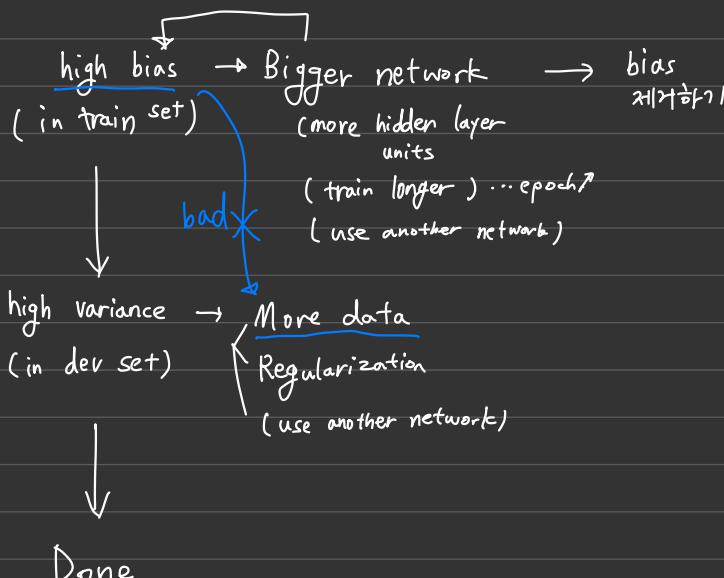


Optimal  $\lambda$ ,  
인간 기준으로 판단했을 때

비슷한 accuracy가 나오게 해야 함.

ex) 인간 탐색력을 15%로 향으면  
15% accuracy도 나쁘지  
않은 값이 됨

# Basic Recipe for machine Learning



"bias vs variance tradeoff"

두 가지를 동시에 원정히 줄일 수 없다는 것

그래도 위 조건을 만족한 큰 네트워크에서

적절히 regularization을 거치면, 데이터는 추가하면

bias의 변화 없이 <sup>기본</sup> 향상 variance를 줄일 수 있다

## Regularization

overfitting) 의심된다면 high variance

이 때 regularization을 해볼 수 있음

$n^{[l]}$ : 현재 layer의 unit 수  
 $n^{[L-1]}$ : 이전 layer 수

python : lambda

dev set이나 교차 검증으로 설치

train set의 parameter가 적어지게 하기 위해

두 표준을 설정한 후 고친해보면서

무엇이 최선인지 확인해보기

### Logistic Regression

$$\min_{w, b} J(w, b)$$

$$w \in \mathbb{R}^{n_x}$$

$$b \in \mathbb{R}$$

$\lambda$ : regularization parameter [생각. 포함해도 됨]

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2 + \frac{\lambda}{m} b^2$$

만약에 사용

$$L_2 \text{ Regularization} : \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

여기서  $w$ 만 일변수

$b$ 를 생각해도 되는 이유

:  $w \in \mathbb{R}^{n_x \times 1}$ , 꽤 높은 차수의 측정 벡터.

특히 변동이 심할 때 더욱 좋은

반면에  $b$ 는 그저 단수.

그러나 거의 모든 매개변수들이  $b$ 보다  $w$ 에 있음

$$L_1 \text{ Regularization} : \underbrace{\frac{\lambda}{2m} \sum_{j=1}^m |w_j|}_{L_1 \text{에서 추가}} = \frac{\lambda}{2m} \|w\|_1, \rightarrow L_1 \text{ 쓰면}$$

$w$ 는 회백색집.

이는  $w$  안에 수많은 0이 있음을 의미

이제 한 층연에서

이것이 모델 입출력에 도움이 됨.

파라미터 세트는 이고, 모델을 저장할 메모리가 덜 필요

### Neural Network

gradient  $\leftarrow$  이것이 함께

학습하려면?

$$J(w^{[0]}, b^{[0]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m (L(\hat{y}^{(i)}, y^{(i)})) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|^2$$

$$L_2 : \|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2 \rightarrow \text{그지 matrix 요소들의 제곱합}$$

$$w : (n^{[0]}, n^{[1]}, \dots, n^{[L]})$$

$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} W^{[l]}$$

$$W^{[l]} = w^{[l]} - \alpha dw^{[l]}$$

$$W^{[l]} (1 - \frac{\alpha \lambda}{m}) = W^{[l]} \cdot \alpha (\text{from back}) + \frac{\lambda}{m} W^{[l]}$$

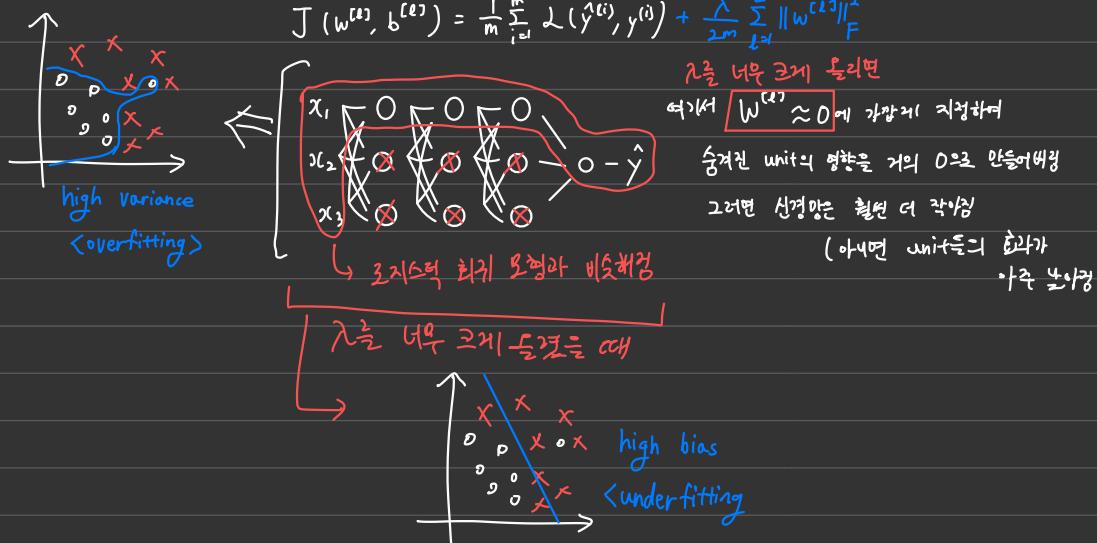
$L_2$  Regularization  $\leftarrow$

weight decay라고 불리는 이유

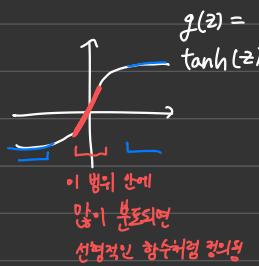
$(W^{[l]})$ 에 1보다 작은 수를 곱해서  
작게 만들고

# Why Regularization Reduce Overfitting

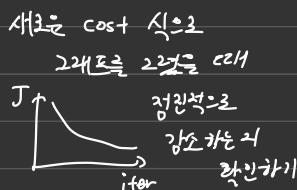
①



②



## 2-1 Debugging



만약  $\lambda \uparrow$ ,  $\Rightarrow W^{(l)} \downarrow$   
왜냐면 cost에서 더 큰 값을 갖기 때문에  
이 때  $J^{(l)} = W^{(l)} a^{(l+1)} + b^{(l+1)} \cdot 1/m$   
 $W^{(l)}$ 도 작아짐.

그럼 빨간 꽂에 몰리게 되어

선형 모델이 됨

모든 layer가 선형이면

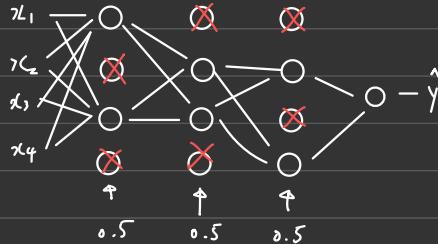
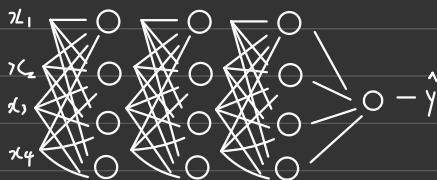
전체 네트워크도 선형이 됨

그래서 비선형의 overfit은 불가능



# Dropout Regularization

각 layer 별로 unit을 제거하는 확률을 설정



도입 방법 종류

① Inverted Dropout : keep-prob의 숫자가 어떤 드롭아웃 기법값을 유지

- layer  $l=3$
- $d_3 = \text{np.random.rand}(a_3, \underbrace{\text{shape}[0]}, \underbrace{a_3, \text{shape}[1]}_{\text{random}}) < \underbrace{\text{keep\_prob}}_{\text{숫자: } 0.8}$

hidden unit이 유지될 확률  
숫자: 0.8

\* keep-prob  
 $0.5 \rightarrow 0.6$ 로 바꾸면

→ 정규화 효과  
→ train set error↓

- $a_3 = \text{np.multiply}(a_3, d_3)$   $\# a_3 = a_3 \cdot d_3$
- 3 번째 layer의 activation

shape  
 $d_3 = a_3$   
random 사용!!  
 숫자: 0.8

예측  
\* test 시에는 drop-out 적용 X

결과값이 임의의 숫자가 되면  
좋지 않기 때문.  
해결책이 noise를 더 할 뿐  
keep-prob을 하면  $a_3$ 의  
기댓값이 변하지 않으니 괜찮

$\Rightarrow a_3 / \text{keep\_prob}$

만약 4 번째 layer의 50개의 unit이 끊다면

$a_3$ 은  $50 \times 1$  or  $50 \times m$  차원.

50개에 대해서 80%가 유지된다면  
10개는 삭제되며 그들의 값을 0으로 채움

$$z^{[4]} = W^{[4]} \cdot \underbrace{a^{[3]}}_{\text{이 때 20%가 0이든}} + b^{[4]}$$

이 때 20%가 0이든

10개가 0이 됨

그러면  $z^{[4]}$  자체가 강조하지 않기 위해

0.8을 나눠서 다시 20%만큼 증가하게 하여

$a_3$ 의 기댓값이 변하지 않도록 해줌

\* 꼭 keep-prob

모든 레이어에 똑같이

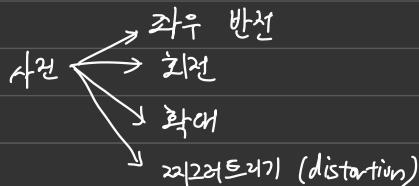
할 필요 X

# for Remove Overfitting

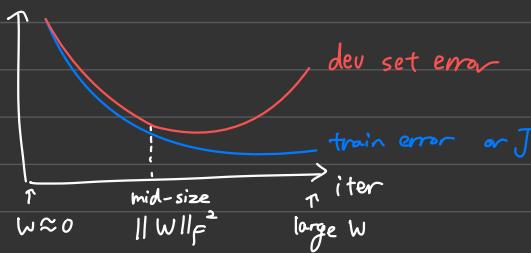
## Other Regularization Methods

① L2 ② dropout

③ Data Augmentation



### + Early Stopping



$w$ 에 대해 train을 stop

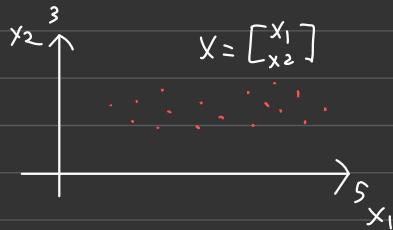
<단점>

cost  $J$  최적화로  
↳ 가중치 강화, 오선정  
RMSPROP, Adam 등  
overfitting 해결  
- regularization

이 줄을 하는 순간에 stop하는  
어느 것인가 문제인지 알기 어려움

그러나 parameter  $\lambda$  를  
다양하게 사용해 보는 편

# Normalizing Input

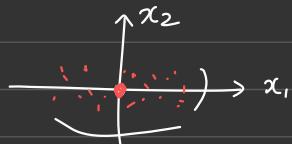


〈 Normalize train set 〉

way 1) Subtract mean

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$x := x - \mu$$

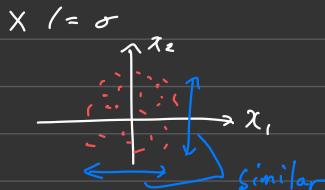


way 2) normalize variance

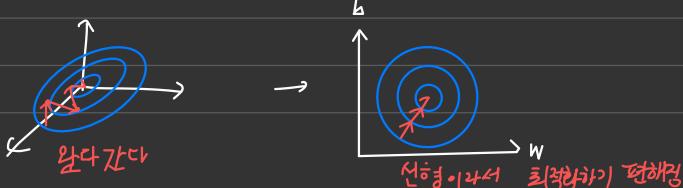
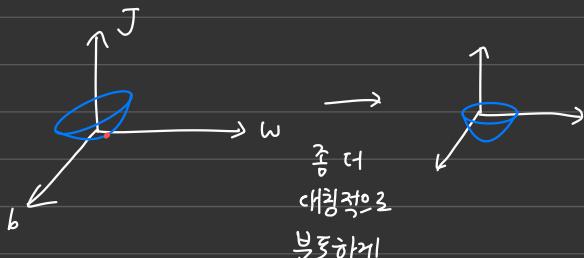
$x_i$  이  $x$  보다 넓은지 분포

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m x^{(i)} - \mu$$

element-wise



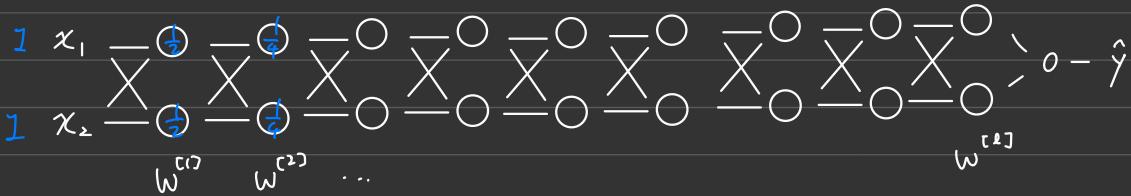
Why Nonnormalize inputs?



심층 신경망을 교육시키는 것의 가장 큰 문제점은

## Vanishing / Exploding Gradients

: 대미터가 사라지거나 기울기가 폭발적으로 증가하는 경우, 이를 끝이나 기울기가 굉장히 작아지는 경우



linear activation func       $b^{[L]} = 0$   
 $g(z) = z$

$$y = w^{[L]} \cdot w^{[L-1]} \cdots w^{[2]} \cdot w^{[1]} \cdot x$$
$$\underbrace{z^{[1]}}_{\alpha^{[1]}} = g(z^{[1]}) = z^{[1]}$$
$$\alpha^{[2]} = g(z^{[2]}) = g(w^{[2]} \alpha^{[1]})$$

$$w^{[L]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \text{이라면}$$
$$\underbrace{\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}}_{\downarrow}^{l-1 \text{번 반복}} \cdot x$$

이러면  $n$ 이 레이어의 개수 일 때  
 $y$ 는  $1.5^n$  개를 가지면서 네트워크를

그리고  $\begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$  대신  $\begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$  라면  $\frac{1}{2}$ 씩 작아진다.

0.5는 너무 빠르게 감소해서 0.9도 괜찮다.

# Weight Initializing for Deep Networks : (단계별 X) 앞서 vanishing gradient 문제 해결법

## ① Random initialization에서 조금 더 괜찮거나, 조성스럽게 선택하기

일단, Unit 간에 대해 살펴보면

$$y = g(z) = g(w_1x_1 + w_2x_2 + \dots + w_nx_n + b)$$

\$n\$ : 입력 개수  
일단 뷰어  
 $w_i$ 는 학습하려면 각 항이 적을 수록 좋음

$x_1, x_2, x_3, x_4 \rightarrow O \rightarrow \hat{y}$

성능 선형화에선  $a^{(l)}$

이제  $w_i$ 의 편차 =  $\frac{1}{n}$ 로 설정

$$w^{(l)} = np.random.randn(shape) * np.sqrt(\frac{1}{n^{(l-1)}})$$

그런데 이미 입력이 평균 0, 편차 1이면

근데 비슷한 scale을 갖게되어 문제 해결은 안 되지만  
그래도 vanishing gradient 문제는 도와줌

왜냐면 (보다 너무 크거나 작게 하지 않기) 막아주기 때문.

너무 빠르게 0이 되거나 학습이 끝나기 때문

만약 ReLU를 쓴다면  $y_n$  대신

$w_i$ 의 편차 =  $\frac{2}{n}$ 로 하면 좋음

$$g^l(z) = \text{relu}(z)$$

만약  $\tanh$ 을 쓴다면

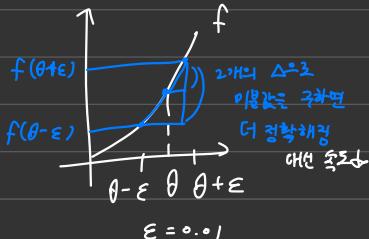
$$\text{Xavier initialization} = \sqrt{\frac{1}{n^{(l-1)}}} \cdot \frac{2}{\pi} \text{ 사용}$$

$$\text{이는 } \sqrt{\frac{2}{n^{(l-1)} + n^{(l)}}} \text{ 공식을 사용}$$

## Gradient Checking

back prop-1 잘 이루어지고 있나 확인하는 방법

일단 기울기의 산출값을 구하는 방법을 보자



$$\frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon} \approx g(\theta)$$

이 때  $f(\theta) = \theta^3$  이라면

$$f(\theta) = 3.0001 \approx 3$$

$$\therefore f'(\theta) = 3\theta^2$$

( $\theta, \theta + \epsilon$ 만 고려하면 3.0301로 예상↑)

Gradient Checking 적용 방법 : 적용

1st : 모든 parameter를 big vector 형식으로 재정비

$$\begin{pmatrix} d\omega \\ db \end{pmatrix}$$

same shape  $\begin{pmatrix} w^{(1)}, \dots, w^{(L)} \\ b^{(1)}, \dots, b^{(L)} \end{pmatrix} \rightarrow \theta$

이를 위해 모든  $w$ 를 연결하여

Vectorize reshape

2nd :  $\frac{dW^{(1)}}{db^{(1)}} \dots \frac{dW^{(L)}}{db^{(L)}}$  big vector  $\theta$  형식으로.

3rd : Grad Check

for each  $i$  :

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

$$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

4th :  $d\theta_{approx}$ 와  $d\theta$ 의 shape가 일치하는가

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

이 식을 적용할 때,  $\epsilon = 10^{-7}$ 을 추천.

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$$

$\leq 10^{-7}$  이면 딥러닝의 derivative approximation이 맞을 가능성이 높음

# Gradient Checking

## 〈주의사항〉

① training 시 적용X. 흔히 DeBug 툴로만

모든 i에 대해서  $d\theta_{approx}^{[i]}$  구하는 것은 오래 걸림

② 만약 알고리즘이 실패했다면,  
bug의 원인인 components를 살펴보기

$d\theta_{approx}$  와  $d\theta$ 가 많이 다르면

|번제의 다른 값들을 보면  
즉,  $\underbrace{d\theta_{approx}^{[i]}}$ ,  $\underbrace{d\theta^{[i]}}$  중 어느 쪽 문제인가|

③ Remember Regularization  $\frac{\lambda}{2}$

$$J(\theta) = \frac{1}{m} \sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_k \|w^{[k]}\|_F^2 \text{ 일 때}$$

$d\theta = J \frac{\partial}{\partial \theta}$  이을한 거임

④ Doesn't work with dropout

grad check는 dropout에서 작동하지 않음

dropout 적용이 필요한지 알려면

$keep\_prob = 1.0$ 으로 해보기 시도하기

Run at

⑤ Random initialization

$w, b$ 가 0에서 초기 범위값을 랜덤으로 초기화

# Initialize

~~Zero initialize~~

```
for l in range(1, len(layers_dims)): # Never Use It !!  
    parameters["W"+str(l)] = np.zeros((layers_dims[l], layers_dims[l-1]))  
    parameters["b"+str(l)] = np.zeros((layers_dims[l], 1))
```

fail  
to break symmetry

~~Random initialize~~

```
for l in range(1, len(layers_dims)):  
    parameters["W"+str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * 10  
    parameters["b"+str(l)] = np.zeros((layers_dims[l], 1))
```

too  
large Weights

Xavier initialize

```
for l in range(1, len(layers_dims)):  
    parameters["W"+str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * np.sqrt(2/layers_dims[l-1])  
    parameters["b"+str(l)] = np.zeros((layers_dims[l], 1))
```

Recommend



- Different initialization  $\Rightarrow$  different results
- No initialize  $\rightarrow$  too large values...
- Xavier is good for Network with ReLU
- Random Initialize : cost starts very high
  - ↳ Initializing with small random values should do better!

# Regularization

## ① L2 Regularization

λ 3 dev set 을 조정.  $\lambda \rightarrow$  over smooth

L2는 경계선을 부드럽게 해줌.

Γ 를 거쳐서 모든 가중치 값을 더 작게 만듬

2t  
정

- cost 계산
  - regularization 흑은 cost에 더해짐
- back prop
  - weight 행렬에 대해, gradient 속에 흑을 추가
- Weight end up smaller ("weight decay")
  - + Weights are pushed to smaller values

```
compute_cost_with_regularization(A3, Y, parameters, lambd):
    """
    Implement the cost function with L2 regularization. See formula (2) above.

    Arguments:
    A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    parameters -- python dictionary containing parameters of the model

    Returns:
    cost -- value of the regularized loss function (formula (2))
    """
    m = Y.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]

    cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost
    # (= 1 lines of code)
    # L2_regularization_cost =
    # YOUR CODE STARTS HERE
    L2_regular_cost = (1/m)*(lambd/2)*(np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))

    # YOUR CODE ENDS HERE
    cost = cross_entropy_cost + L2_regular_cost

    return cost
```

```
def backward_propagation_with_regularization(X, Y, cache, lambd):
    """
    Implements the backward propagation of our baseline model to which we added an L2 regularization term.

    Arguments:
    X -- input dataset, of shape (input size, number of examples)
    Y -- "true" labels vector, of shape (output size, number of examples)
    cache -- cache output from forward_propagation()
    lambd -- regularization hyperparameter, scalar
    """
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2, Z3, A3, W3, b3) = cache
    dZ3 = A3 - Y
    dW3 = 1./m * np.multiply(dZ3, np.int64(A2 > 0))
    dW3 = 1./m * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T) + (lambd/m)*W2
    db2 = 1. / m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T) + (lambd/m)*W1
    db1 = 1. / m * np.sum(dZ1, axis=1, keepdims=True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
                "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
                "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients
```

## ② Dropout

Randomly shut down some neurons in each iter

뉴런을 제거하면 새로운 모델이 되는 것과 마찬가지.

특이한 뉴런 몇 가지에 반응하지 않도록 함.

```
def forward_propagation_with_dropout(X, parameters, keep_prob = 0.5):
    np.random.seed(1)

    # retrieve parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    # First layer
    # LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID
    Z1 = np.dot(W1, X) + b1
    A1 = relu(Z1)
    D1 = np.random.rand(A1.shape[0], A1.shape[1]) # Step 1: initialize matrix D1 = np.random.rand(..., ...)
    D1 = (D1 < keep_prob).astype(int) # Step 2: convert entries of D1 to 0 or 1 (using keep_prob)
    A1 = A1 * D1 #np.multiply(A1, D) # Step 3: shut down some neurons of A1
    A1 /= keep_prob # Step 4: scale the value of neurons that haven't been shut down

    # Second layer
    Z2 = np.dot(W2, A1) + b2
    A2 = relu(Z2)
    D2 = np.random.rand(A2.shape[0], A2.shape[1]) # Step 1: initialize matrix D2 = np.random.rand(..., ...)
    D2 = (D2 < keep_prob).astype(int) # Step 2: convert entries of D2 to 0 or 1 (using keep_prob)
    A2 = A2 * D2 #np.multiply(A2, D2) # Step 3: shut down some neurons of A2
    A2 /= keep_prob # Step 4: scale the value of neurons that haven't been shut down

    Z3 = np.dot(W3, A2) + b3
    A3 = sigmoid(Z3)
    cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)
    return A3, cache
```

~~~~~ 3. 나누라는 것은  
결국 X2를 하는 것

Dropout =  
training에서 forward / backward  
모두 적용해야 함.  
단 test 할 때는 안 해!

```
def backward_propagation_with_dropout(X, Y, cache, keep_prob):
    m = X.shape[1]
    (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3) = cache

    dZ3 = A3 - Y
    dW3 = 1./m * np.dot(dZ3, A2.T)
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims=True)
    dA2 = np.dot(W3.T, dZ3)
    dA2 *= D2 # Step 1: Apply mask D2 to shut down the same neurons as during the forward propagation
    dA2 /= keep_prob # Step 2: Scale the value of neurons that haven't been shut down

    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T)
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dA1 *= D1 # Step 1: Apply mask D1 to shut down the same neurons as during the forward propagation
    dA1 /= keep_prob # Step 2: Scale the value of neurons that haven't been shut down

    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T)
    db1 = 1./m * np.sum(dZ1, axis=1, keepdims=True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3, "dA2": dA2,
                 "dZ2": dZ2, "dW2": dW2, "db2": db2, "dA1": dA1,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients
```

# Gradient Check

Gradient Descent의 Bug을 잡기 위해 사용

Back prop에서 gradient를 계산할 때  $\frac{\partial J}{\partial \theta}$  를 이용

이 때  $J$ 는 forward prop의 loss func에서 나온 값

forward prop은 구현하기 위해서  $J$ 를 정확히 계산했라고 다를 생각함  
이를 확인하기 위해  $\frac{\partial J}{\partial \theta}$  를 계산

$\frac{\partial J}{\partial \theta}$  를 계산하는 법으로 미분식의 정의를 이용해보라는 것

forward prop에서 계산한 미분값과, 평균값  $(+\theta, -\theta)$  시 대해서 계산한 값의 차이를 보자  
 $= \text{gradapprox}$        $= \text{grad}$

```
# GRADED FUNCTION: forward_propagation
def forward_propagation(x, theta):
    """
    Implement the linear forward propagation (compute J) presented in Figure 1 (O)
    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well
    Returns:
    J -- the value of function J, computed using the formula J(theta) = theta * x
    """
    # (approx. 1 line)
    # J =
    # YOUR CODE STARTS HERE
    J = theta * x
    # YOUR CODE ENDS HERE.
    return J
```

```
# GRADED FUNCTION: backward_propagation
def backward_propagation(x, theta):
    """
    Computes the derivative of J with respect to Theta (see Figure 1).
    Arguments:
    x -- a real-valued input
    theta -- our parameter, a real number as well
    Returns:
    dtheta -- the gradient of the cost with respect to theta
    """
    # (approx. 1 line)
    # dtheta =
    # YOUR CODE STARTS HERE
    dtheta = x
    # YOUR CODE ENDS HERE
    return dtheta
```

## Exercise 3 - gradient\_check

To show that the `backward_propagation()` function is correctly computing the gradient  $\frac{\partial J}{\partial \theta}$ , let's implement gradient checking.

### Instructions:

- First compute 'gradapprox' using the formula above (1) and a small value of  $\epsilon$ . Here are the steps to follow:

- $\theta^* = \theta + \epsilon$
- $\theta^* = \theta - \epsilon$
- $J^+ = J(\theta^*)$
- $J^- = J(\theta^-)$
- $\text{gradapprox} = \frac{J^+ - J^-}{2\epsilon}$

- Then compute the gradient using backward propagation, and store the result in a variable 'grad'

- Finally, compute the relative difference between 'gradapprox' and the 'grad' using the following formula:

$$\text{difference} = \frac{\| \text{grad} - \text{gradapprox} \|_2}{\| \text{grad} \|_2 + \| \text{gradapprox} \|_2} \quad (2)$$

You will need 3 steps to compute this formula:

1. compute the numerator using `np.linalg.norm(...)`
2. compute the denominator. You will need to call `np.linalg.norm(...)` twice.
3. divide them.

- If this difference is small (say less than  $10^{-7}$ ), you can be quite confident that you have computed your gradient correctly.

Otherwise, there may be a mistake in the gradient computation.

```
def gradient_check(x, theta, epsilon=1e-7, print_msg=False):
    """
    Implement the backward propagation presented in Figure 1.
    Arguments:
    x -- a float input
    theta -- our parameter, a float as well
    epsilon -- tiny shift to the input to compute approximated gradient with formula(1)
    Returns:
    difference -- difference (2) between the approximated gradient and the backward propagation gradient. F!
    """
    # Compute gradapprox using left side of formula (1). epsilon is small enough, you don't need to worry about theta_minus + theta + epsilon
    theta_plus = theta + epsilon
    theta_minus = theta - epsilon
    J_plus = J(theta_plus)
    J_minus = J(theta_minus)
    gradapprox = (J_plus - J_minus) / (2 * epsilon) # 2번의 상식적으로 구한 값

    # Check if gradapprox is close enough to the output of backward_propagation()
    grad = backward_propagation(x, theta) # 아래 코드로 구한 값
    if np.allclose(grad, gradapprox, rtol=1e-7):
        print("All good!")
    else:
        print("\u274d\u274d\u274d There is a mistake in the backward propagation! difference = " + str(difference))

    if print_msg:
        if difference > 2e-7:
            print("\u274d\u274d\u274d There is a mistake in the backward propagation! difference = " + str(difference))
        else:
            print("\u274d\u274d\u274d Your backward propagation works perfectly fine! difference = " + str(difference))

    return difference
```

# N-차원에서의 Gradient Checking

```
def gradient_check_n(parameters, gradients, X, Y, epsilon=1e-7, print_msg=False):
    """
    Checks if backward_propagation_n computes correctly the gradient of the cost function
    by comparing them to an "approximated gradient" computed using the forward propagation
    module.

    Arguments:
    parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", ...
    grads -- output of backward_propagation_n, contains gradients of the cost function
    X -- input datapoint, of shape (input size, 1)
    Y -- true "label"
    epsilon -- tiny shift to the input to compute approximated gradient with formula(1)

    Returns:
    difference -- difference (2) between the approximated gradient and the backward propagation gradient
    """

    # Set-up variables
    parameters_values, _ = dictionary_to_vector(parameters)

    grad = gradients_to_vector(gradients)
    num_parameters = parameters_values.shape[0]
    J_plus = np.zeros((num_parameters, 1))
    J_minus = np.zeros((num_parameters, 1))
    gradapprox = np.zeros((num_parameters, 1))

    # Compute gradapprox
    for i in range(num_parameters):

        # Compute J_plus[i]. Inputs: "parameters_values, epsilon". Output = "J_plus[i]"
        # "_" is used because the function you have to outputs two parameters
        theta_plus = np.copy(parameters_values)
        theta_plus[i] += epsilon
        J_plus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(theta_))

        # Compute J_minus[i]. Inputs: "parameters_values, epsilon". Output = "J_minus[i]"
        theta_minus = np.copy(parameters_values)
        theta_minus[i] -= epsilon
        J_minus[i], _ = forward_propagation_n(X, Y, vector_to_dictionary(theta_))

        # Compute gradapprox[i]
        gradapprox[i] = (J_plus[i]-J_minus[i]) / (2*epsilon)

    # Compare gradapprox to backward propagation gradients by computing difference
    numerator = np.linalg.norm(grad-gradapprox)
    denominator = np.linalg.norm(grad) + np.linalg.norm(gradapprox)
    difference = numerator / denominator
```

Gradient checking은 뒤에

그러니 train마다 적용할 수 있다

오직 code가 옳은지 확인용으로 쓰고,

종료시킬 때,

back prop으로 학습을

다시 이어서 진행하자

# Fast!

## Batch VS Mini-Batch for Grad Descent : 학습 속도↑ 정확도↑

만약  $m = 5,000,000$  일 때

mini-batch 는 1000 개씩 배정된다면

$$X = \begin{bmatrix} x^{(1)} & \cdots & x^{(1000)} & x^{(1001)} & \cdots & x^{(2000)} & \cdots & x^{(m)} \end{bmatrix}_{(n_x, m)}$$

$x^{\{1\}}$        $x^{\{2\}}$        $x^{\{5,000\}}$

$$Y = \begin{bmatrix} y^{(1)} & \cdots & y^{(1000)} & y^{(1001)} & \cdots & y^{(2000)} & \cdots & y^{(m)} \end{bmatrix}_{(1, m)}$$

$y^{\{1\}}$        $y^{\{2\}}$        $y^{\{5,000\}}$

$X^{(t)}$  : i 번째 training data

$Z^{[l]}$  : 신경망의 l 번째 layer

$X^{\{t\}}, Y^{\{t\}}$  :  $(n_x, \text{mini-batch}) \xrightarrow{\exists \gamma} \text{여기서 } (n_x, 1000)$

### Mini-batch Gradient Descent

for  $t = 1, \dots, 5000$  :

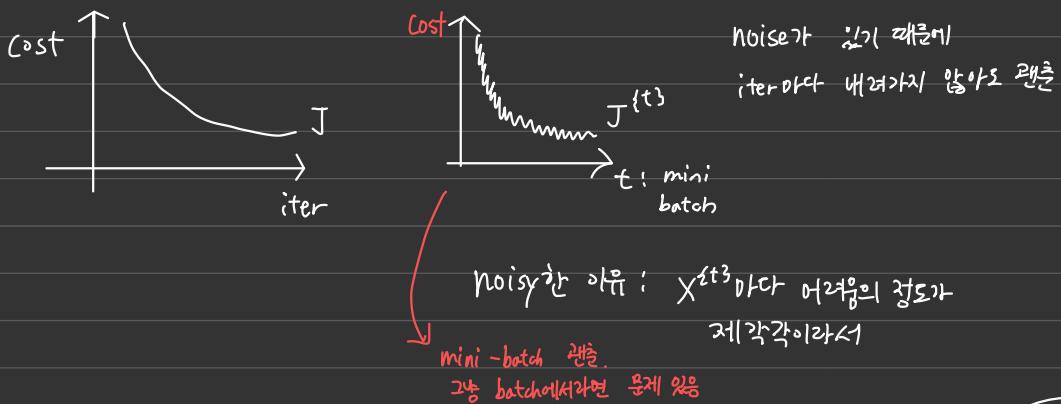
- forward prop on  $X^{\{t\}}$
- $Z^{[l]} = W^{[l]} X^{\{t\}} + b^{[l]}$
- $A^{[l]} = g^{[l]}(Z^{[l]})$  Vectorize
- $A^{[T]} = g^{[T]}(Z^{[T]})$  from  $X^{[t]}, Y^{[t]}$
- Compute cost  $J = \underbrace{\frac{1}{1000} \sum_{i=1}^1 L(\hat{y}^{(i)}, y^{(i)})}_{\text{Ex mini-batch size}} + \underbrace{\frac{\lambda}{2 \cdot 1000} \sum_l \|W^{[l]}\|_F^2}_{\text{from } X^{[t]}, Y^{[t]}}$

- Backprop to compute gradient cost  $J^{[t]}$  (using  $(X^{\{t\}}, Y^{\{t\}})$ )

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha dW^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha db^{[l]} \end{aligned}$$

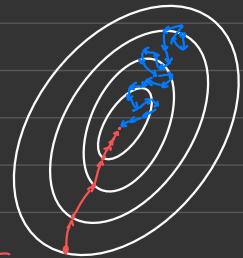
# Understanding Mini-batch

## Batch Grad Descent      Mini-Batch GD



Batch GD에서 cost  $J$ 를 작게 하려면  
 ① mini-batch GD ② better random initialize  
 ③ Adam

< Choosing Mini-Batch Size >



if mini-batch size =  $m$ , Batch Grad Descent  $\Leftarrow$   $(X^{t3}, Y^{t3}) = (X, Y)$

$\Rightarrow$  I : Stochastic Grad Descent  $\Leftarrow$  Every Examples are  
 speed down it own mini-batch

실제로는 1과  $m$  사이의 값을 이용 : Fast Learning

• Vectorization을 갖게됨

1개보다 1000개를 동시에 처리하면 빠름

• 끝까지 기다릴 필요 없이 중간 과정 확인 가능

∴ small train set ( $\leq 2000$ ) : batch-size

typical mini-batch size :  $64, 128, 256, 512, 1024$

use mini-batch on CPU/GPU

+ ) 각각의 mini-batch  $\frac{2}{2}$   
 핸들하게 하는 게 좋음  
 ex) Shuffle, Partition  
 $X^{t3}$  순서  
 키어서 썹기  
 미리 미니  
 배치 안  
 넣기

## 지수적 이동 평균 가중치: Exponentially Weighted Averages

그 날의 온도라면

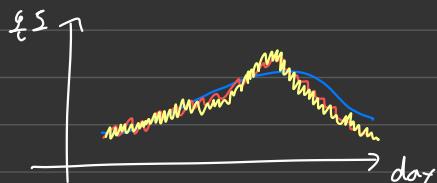
$$\text{근식: } V_t = \underbrace{\beta \cdot V_{t-1} + (1-\beta) \theta_t}_{= 0.9 \text{ 라면}} \text{ 일 때}$$

$= 0.9$  라면, 이전 10일 동안 평균 기온

$$\beta \approx \frac{1}{1-\beta} \text{ 만큼} \quad = 0.98 \text{ 라면, 이전 } \frac{1}{1-0.98} = 50 \text{ 일 동안 } "$$

$= 0.5$  라면, 이전 2일 동안

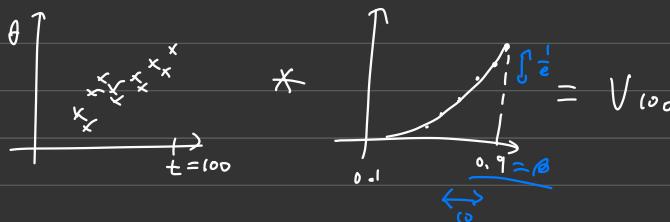
총총



$\therefore \beta$ 가 커지면 곡선은 부드러워지고  
데이터의 전체적인 트렌드를 잘 따랐지만  
latency가 생김

$$\begin{aligned} V_{100} &= 0.9 V_{99} + 0.1 \theta_{100} \\ &= 0.1 \theta_{100} + (0.1 \theta_{99} + (0.1 \theta_{98} + V_{97} \dots)) \\ &\approx 0.1 \theta_{100} + 0.1 \times 0.9 \theta_{99} + 0.1 (0.9)^2 \theta_{98} + 0.1 (0.9)^3 \theta_{97} \dots \end{aligned}$$

이를 그려보면



코드로 표현하면,

$$\begin{aligned} \rightarrow \underline{V_\theta = 0} \quad &\text{변수는 한 개만 만들어서} && \rightarrow \text{그 중간의 값을 모두 저장하고} \\ \text{반복 } \{ &\text{누적 저장} && \text{있어야는 과정이} \\ &\text{Get next } \theta_t && \text{더 많은 메모리가 필요하고} \\ &V_\theta = \beta \cdot V_\theta + (1-\beta) \theta_t && \text{연산 부담도 커짐} \end{aligned}$$

}

## Bias Correction in Exponentially Weighted Average

$$V_t = \beta V_{t-1} + (1 - \beta) \theta_t$$

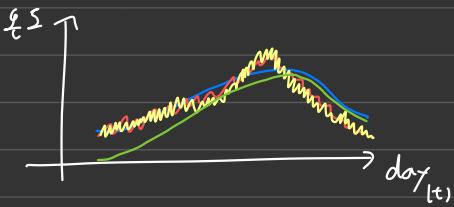
$$V_0 = 0$$

$$V_1 = 0.98 V_0 + 0.02 \theta_1$$

$$V_2 = 0.98 \underbrace{V_1}_{= 0.0196 \theta_1} + 0.02 \theta_2$$

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

실제로 이 공식을 쓰면  
화면색 곡선이 아님  
 $\beta = 0.98$   
아주 빨리 시작하는 초록색  
(0으로 시작)



↓ 개선

$$V_t = \frac{V_t}{1 - \beta^t}$$

$$t=2 : 1 - \beta^t = 0.0396$$

$$\frac{V_2}{0.0396} = \frac{"0.0196\theta_1 + 0.02\theta_2"}{0.0396}$$

∴ training 초반부의

추정치를 개선하고 싶다면

bias correction을 하자

때때로 이를 사용하지 않고

추정치가 데이터를 따라갈 때까지

기다리는 경우도 있다

∴ t가 커질 수록 초록색과 화면색의 차이는 줄어들고 점점 겹쳐짐

이는 bias correction이 해낸 것

$$V_1 = -$$

$$V_2 = \frac{V_1}{1 - \beta^2} = \frac{1.5}{0.95}$$

$$V_t = 0.5x$$

$$V_1 = 0.5x V_0 + 0.5x \theta_1$$

$$V_1 = 5$$

$$V_2 = 0.5x V_1 + 0.5x \theta_2$$

$$2.5 + 5$$

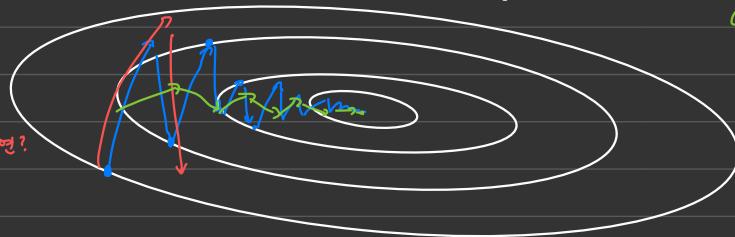
10

# Gradient Descent with Momentum

훨씬 적은 최적화로 빠르게 도달

Minibatch

: 변동폭이 GD를  
느리게 하여  
더 큰 학습 속도를  
사용하지 않도록 해줌



GD + Momentum

세(3) 느리게  
가로로 빠르게  
학습하기

< Momentum > : 기울기 강화의 결과를  
복드릴지 해증

t 번 만족 :

작은 경지 / 초기  
mini batch

계산 -  $dW, db$  on 현재 batch

$$\text{기존 평균치 구하기} \leftarrow V_{dw} = \beta V_{dw} + (1-\beta) dW$$

$$V_{db} = \beta V_{db} + (1-\beta) db$$

이 모멘텀이라는 속도를 의미

update  $\left\{ \begin{array}{l} W = W - \alpha V_{dw} \\ b = b - \alpha V_{db} \end{array} \right.$

버리면 몸을  
기죽시키는 것



지그재그에서

가로 빠르게 가려면

→ 가로 방향에서의 평균은  
아직 꽤 큰 값일 것.

세(3) 느리게 진행하려면

- ↘ ↗ - 양수, 음수의 균형을 잡아  
평균이 거의 0으로 되게 함

그러므로 GD와 모멘텀이

세로 방향에서 훨씬 더 작은 변동으로 가게 함

# RMS prop

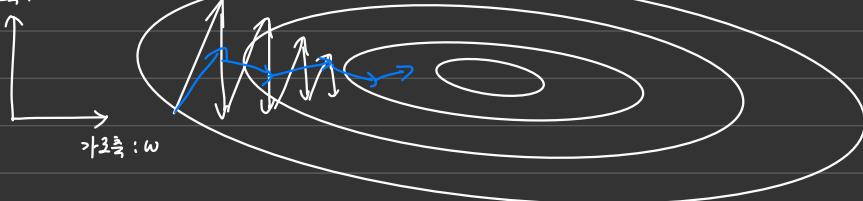
Root - mean - square - propagation

가중기 강화의 속도를 증가시킴

b(세로)로 느리게

w(가로)로 바르게 해보자

세로축 : b



t번 반복:

Compute  $dW, db$  on 현재 <sup>mini</sup> batch  
element-wise

$SdW = \beta SdW + (1-\beta) dW^2$  : 가로축 빠르게를 위해  $SdW$ 는 비교적 작은 값으로

$Sdb = \beta Sdb + (1-\beta) db^2$  : 세로축 느리게 ..  $Sdb$ 는 // 크게

$$W = W - \alpha \frac{dw}{\sqrt{SdW}}$$

$$b = b - \alpha \frac{db}{\sqrt{Sdb}}$$

learning-rate를 고려하면

세로 방향으로 갈라지 않으면서

더 빠른 학습을 진행할 수 있음

만약 분포가  
0에 너무 가까운면 문제...

이는 다음 장에서!

(분포에 +δ을 더보정)  
 $\delta=10^{-6}$

실제로는 고차원 이미지

$$w_1, w_2, \dots, w_n \uparrow$$

일 것임

그러니  $dW$ 는 굉장히 고차원의 벡터.  
 $db$

모든,

변동이 있는 곳에선

미분의 제한의 기준 평균값이 커져서

절대적으로 변동을 무리하게 함

# Adam Optimization

= RMS prop + Momentum

= Adaptive stands for adaptive momentum estimation

$$\text{초기화: } V_{dw} = 0, S_{dw} = 0 \\ V_{db} = 0, S_{db} = 0$$

1번 반복:

Comput  $dw, db$  using ~~전부~~<sup>mini!</sup> batch

$$\begin{array}{l} \text{Momentum} \\ \text{-like update} \end{array} \left( \begin{array}{l} V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw \\ V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \end{array} \right)$$

$$\begin{array}{l} \text{RMS prop} \\ \text{-like update} \end{array} \left( \begin{array}{l} S_{dw} = \beta_2 S_{dw} + ((1 - \beta_2) dw^2) \\ S_{db} = \beta_2 S_{db} + ((1 - \beta_2) db^2) \end{array} \right)$$

bias correction  $\frac{\partial}{\partial t}$

$$\Rightarrow \left( \begin{array}{l} V_{dw}^{\text{correct}} = V_{dw} / (1 - \beta_1 t) \\ V_{db}^{\text{correct}} = V_{db} / (1 - \beta_1 t) \end{array} \right)$$

$$\left( \begin{array}{l} S_{dw}^{\text{correct}} = S_{dw} / (1 - \beta_2 t) \\ S_{db}^{\text{correct}} = S_{db} / (1 - \beta_2 t) \end{array} \right)$$

$$\text{Finally} \quad \left( \begin{array}{l} W = W - \alpha \frac{V_{dw}^{\text{correct}}}{\sqrt{S_{dw}^{\text{correct}}} + \epsilon} \\ b = b - \alpha \frac{V_{db}^{\text{correct}}}{\sqrt{S_{db}^{\text{correct}}} + \epsilon} \end{array} \right)$$

조절하려는 Hyper parameter

( $\frac{\partial}{\partial t}$ )

$\alpha$ : 어느 범위 내의 값은 모드 시도  $\rightarrow$  1회

$\beta_1$ : 0.9 mean of 0.9들로 : first momentum

$\beta_2$ : 0.999 ( $dw^2$ )

제곱들의 exponentially weighted average

: second momentum

for Adam

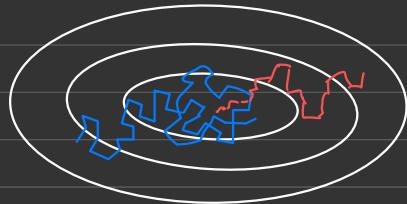
(선택)

$\epsilon: 10^{-8}$

# Learning Rate Decay

학습 속도를 높이기 위한.

시간이 지날 수록 천천히 learning rate는 감소시킴



수렴하지 못하고

그 주변에서 맴돌고 있을 때 사용

1 epoch = 1 pass through data



$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch\_num}} \times \alpha_0$$

if  $\alpha_0 = 0.2$ ,  $\text{decay-rate} = 1$

then

| Epoch | $\alpha$ | Hyperparameter |
|-------|----------|----------------|
| 1     | 0.1      | 시작             |
| 2     | 0.067    |                |
| 3     | 0.05     | 감소             |
| 4     | 0.04     |                |

< Other learning Rate decay formula >

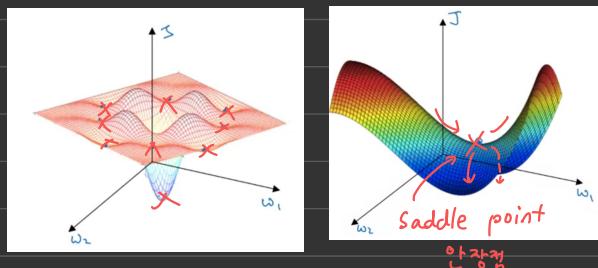
- $\alpha = \underbrace{0.95}_{\text{less than 1}}^{\text{epoch\_num}} \times \alpha_0 \Rightarrow \text{exponentially decay}$

- $\alpha = -\frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{E}} \cdot \alpha_0$ 
  - ↳ 어떤 학습
  - ↳ mini batch의 수

- descrete staircase :  $\alpha \left| \begin{array}{ccccccc} t \\ \hline \end{array} \right. = \frac{\alpha}{t}$

# 국소 최적값 Local Optima 문제점 $\rightarrow$ Not for NN

## 1. Local Optima in NN



→ 높기가 0인 점은 모두 안장점.  
그런데 그 점 국대정 (concave)거나  
국소정 (convex)일 수 있다.

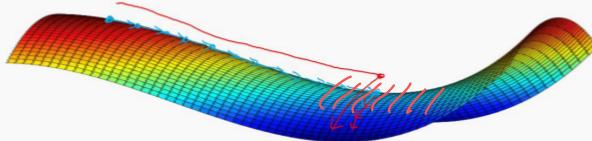
데이터 많을 때는

국소 최적값보단

안장점에 안착할 확률이 높음

NN에서의 문제점은? : Plateaus (정체구간)  
학습 속도를 느리게 함

### Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Andrew Ng

∴ 데이터가 많거나  
NN이 큰 경우  
국소 최적값에 갇힐 확률 커다  
비용 함수  $J$ 는  
비교적 고차원의 공간에서  
정의된다

실제 문제는 plateaus이며  
학습 속도를 느리게 한다

여기가 바로 momentum,  
RMSprop,  
Adam 등이다

도움이 되는 부분이다  
(이때 plateaus를 벗어날 수 있도록)

# Hyperparameter Tuning

| 1순위                                  | 2순위                                             | 3순위                                                      |
|--------------------------------------|-------------------------------------------------|----------------------------------------------------------|
| $\alpha$ : learning-rate<br>→ 0.9 추천 | $\beta$ : momentum term<br>num. of hidden units | num. of layers<br>learning rate decay<br>mini-batch size |

$\rightarrow |E|$

Adam에서

$$\beta_1 : 0.9$$

$$\beta_2 : 0.999$$

$$\epsilon : 10^{-8}$$

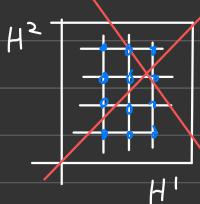
Adam에서 분모에 들어감

Momentum의 hyper parameter  $\beta$ 는

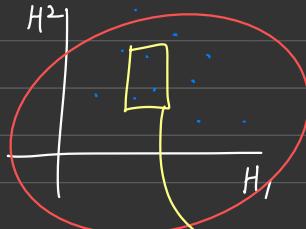
0.9 ~ 0.99 사이를 추천

$$\begin{cases} r = \text{np.random.rand}() \\ \text{beta} = 1 - 10^{**}(-r-1) \end{cases}$$

표를 그리지 말고, 다양한 값을 대입해보며 확인하자!



H: hyperparameter



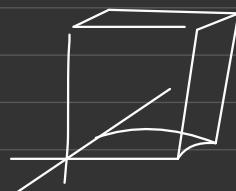
세세하게  
다양한 값을  
확인할 수 있음.  
교정에 있는 값만  
확인하지 않고!

hyperparameter의 수가 적으면 괜찮지만

많을 때, 어떤 값으로 학습할 때 좋은지 알기 어렵다

잘 작동하는 곳을 찾으면  
확대하여 또 다시

Sampling 으로 해봄으로써  
가장 좋은 값을 찾을 수 있다



Hyperparameter가  
3개일 때

# Using an Appropriate Scale to pick Hyperparameters

## ① Picking Hyperparameters at Random : 특정 범위에서 랜덤!

hidden Unit  $\approx$  50 - 100 사이를 고려하기

layer  $\approx$  2-4 사이를 고려하기

## ② Appropriate Scale for hyperparameters

$\alpha \approx 0.0001 \sim 1$  사이를 고려할 때  
너무 많으니

log Scale을 써보자  
good

$$r = -4 * np.random.rand() \quad (-4 < r < 0)$$
$$\alpha = 10^r \quad (10^{-4} < \alpha < 10)$$

$r \in$  특정  $(n_1, n_2)$  범위로 잡아보면서  
좋은 범위에 집중하기



## + ) Hyperparameter $\beta$ for Exponentially weighted averages

$$\beta = \underbrace{0.9}_{\text{이든 마저}} \sim \underbrace{0.999}_{\downarrow 1000\text{-번}} :$$

10개의 값에서  
평균을 구하는 것과 같다  
(10일의 평균 기준)

$\beta$ 가 1에 근접할 수록  
초2만  $\beta$ 의 변화에도  
결과값이 민감하게 반응

$\underbrace{0.9005 \sim 0.999}_{0.9 \sim 0.995}$  >에서  
마지막이 더 민감함

$1-\beta$ 의 범위로 고려하면,  $0.1 \sim 0.001$ 임  
 $10^{-1} \sim 10^{-3}$  : 모든 원래 범위에  $\beta$ 는  
'원래' 범위라는 점!

$$-3 < r < 1$$

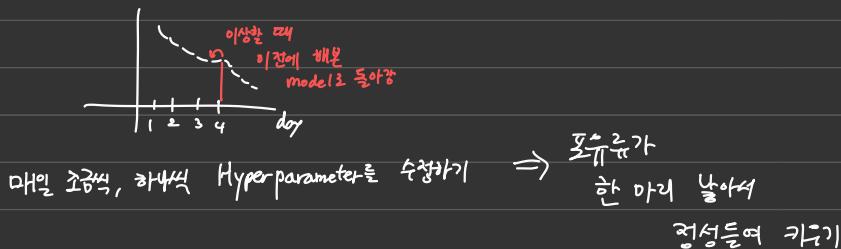
$$1-\beta = 10^r$$

$$\beta = 1 - 10^r$$

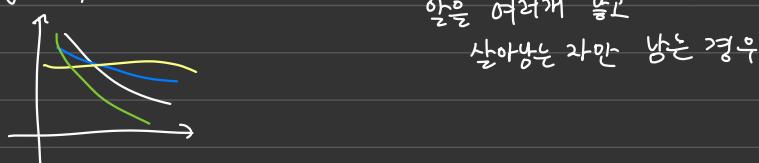
## 실제로 Hyperparameter Tuning 와 주의해야 할 점

몇 개월마다 한 번씩 다시 하기

- Babysitting one model : CPU/GPU 부족으로  
모델을 하나밖에 테스트하는 것



- Training many models in parallel  $\Rightarrow$  개구리가



# Normalizing Activations in a Network

## Batch Normalization

: hyperparameter 찾기하고 신경망을 훈련하게 함

logistic 회귀에선 정규화는 입력값에 대한 것

$$M = \frac{1}{m} \sum_i x^{(i)} \quad \text{평균 구하고,}$$

$$x = x - M \quad \text{입력값에서 평균을 빼서}$$

$$\sigma^2 = \frac{1}{m} \sum_i (x^{(i)})^2 \quad \text{편차를 계산}$$

$$X = X / \sigma^2 \quad \text{편차대로 데이터를 정규화함}$$



Deeper Model에서는

입력값  $x$ 만 아니라 layer마다  $a$ 도 ~~있~~

있는데

$a$ 의 분산을 정규화하면 그 다음의  $w, b$  파라미터의 training을 효율적으로, 잘

이것이 바로 Batch Normalization

g.D, momentum  
RMSprop, Adam  
등으로 업데이트 할 수 있음

Implement Batch Norm on One Hidden Layer

NN에 몇 개의 intermediate values가 있다.

그리고 어떤 layer의 hidden unit 수는  $Z^{[L]^{(1)}}, \dots, Z^{[L]^{(m)}}$  이다

그 layer를 대하여

$$\textcircled{1} \quad U^{[L]} = \frac{1}{m} \sum_i Z^{[L]^{(i)}}$$

$$\textcircled{2} \quad \sigma^2_Z = \frac{1}{m} \sum_i (Z^{[L]^{(i)}} - U^{[L]})^2 \quad \text{분산(편차) 계산}$$

$$\textcircled{3} \quad Z^{[L]^{(i)}}_{\text{norm}} = \frac{Z^{[L]^{(i)}} - U^{[L]}}{\sqrt{(\sigma^2_Z)^2 + \epsilon}} \quad \begin{array}{l} \text{평균에서 뺀 값은} \\ \text{분산으로 나누어 정규화 값 계산} \\ \text{증가} \end{array}$$

이전 평균 0, 분산 1인

데이터가 됨

$$\textcircled{4} \quad \tilde{Z}^{(i)} = \sigma_Z \cdot Z_{\text{norm}}^{(i)} + \beta \quad \begin{array}{l} \text{평균 } 0, \text{ 분산 } 1 \text{이 되도록} \\ \text{정규화된 병위가 } \tilde{Z}^{(i)} \text{로 바뀜} \end{array}$$

$\tilde{Z}^{(i)}$ 의 평균값이 0 외에  
올라가는 경우로 되게 함

만약  $r = \sqrt{\sigma^2_Z + \epsilon}$ 라면 원래와 동일

$$\beta = M$$

그 때는  $\tilde{Z}^{(i)} = \tilde{Z}^{(i)}$ 임.

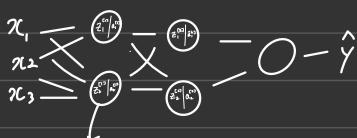
$\hookrightarrow$  숫자적으로 안정감을 주기 위해  $\epsilon$ 을 더함  
 $\sigma^2_Z$ 이 0이 되는 경우가 있어서

$\beta = \sigma \circ \text{grad}_{\theta}$  gradient optimizerch  $\frac{\partial \phi}{\partial \theta}$

# Finding Batch Norm to a Network

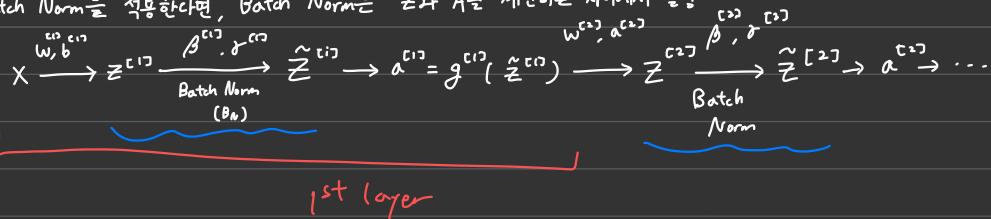
Deep

; tf.nn.batch\_normalization



각각의 unit을  
1st.  $\tilde{z}^{[1]}$  2nd  $\alpha^{[1]}$  를 구함

Batch Norm을 적용한다면, Batch Norm은  $\alpha$ 와  $A$ 를 계산하는 사이에서 발생



이 때,

$$\text{parameter : } \left. \begin{array}{l} W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]} \\ \beta^{[1]}, \gamma^{[1]}, \dots, \beta^{[L]}, \gamma^{[L]} \end{array} \right\} \quad d\beta^{[L]} \text{ 계산 후} \\ \beta = \beta^{[L]} - \alpha d\beta^{[L]} \quad \text{업데이트 가능.}$$

(Adam, RMSprop, 모멘텀 등으로도 업데이트)

설명을, Batch Norm은 mini-batch와 함께 적용됨

$$\text{첫 번째 mini-batch } X^{[1]} \xrightarrow{W^{[0]}, b^{[0]}} z^{[1]} \xrightarrow{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow \alpha^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \rightarrow \tilde{z}^{[2]} \dots$$

$$\text{두 번째 mini-batch } X^{[2]} \rightarrow z^{[2]} \xrightarrow{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow \dots$$

⋮

Parameter :  $W^{[0]}, b^{[0]}, \beta^{[1]}, \gamma^{[1]}$

mini-batch에 대해 전 신경망의 흐름은 무시됨.

Batch Normalization 계산값은 별개로 계산해서.

그리고  $b$ 는 parameter로 안 됨거나 안 넣어야.

$$\text{이 때 } z^{[k]} = W^{[k]} \alpha^{[k-1]} + b^{[k]}$$

$$z_{\text{norm}}^{[k]} \leftarrow \frac{z^{[k]}}{\sqrt{n^{[k]}}} \Rightarrow \tilde{z}^{[k]} = \gamma^{[k]} \cdot z_{\text{norm}}^{[k]} + \beta^{[k]}$$

영향을 주는 parameter

# Implementing Gradient Descent using Batch Norm

for  $t=1 \dots \text{num of mini-batches}$

•  $X^{[t]}$ 에 대해 forward prop 계산

↳ 각각의 hidden layer의 Batch Norm 사용;  $Z^{[t]} \rightarrow \begin{cases} \hat{Z}^{[t]} \\ \text{정규화된 상태} \end{cases}$ 로 표시

• Back prop:  $dW^{[t]}$ ,  $d\gamma^{[t]} \cancel{, d\beta^{[t]}, d\sigma^{[t]}}$

• Update parameters :  $\left. \begin{array}{l} W^{[t]} = W^{[t]} - \alpha dW^{[t]} \\ \beta^{[t]} = \beta^{[t]} - \alpha d\beta^{[t]} \\ \gamma^{[t]} = \dots \end{array} \right)$

기울기 강화  $\alpha$ .

$\alpha$ 는 momentum

RMSprop  
Adam

과도 잘 작동함

# Why does Batch Norm Work?

1. 한 쪽으로 치우친 데이터를 가운데로 옮겨서 계산 절약

2. Regularization의 효과  $\Rightarrow$  다음 장에...

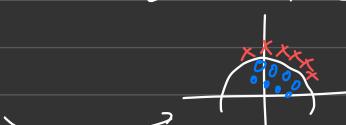
3. Batch Norm은 Weight를 생성

10번재 layer의 weight가 첫 번째 layer의 weight보다 안정적

흑백 사진 고양이 모델



컬러 사진 고양이 모델



고양이 판별은 같지만  
데이터의 분포가 다를 때,  
covariate shift로써  
다시 training 해야함



layer가 여러 개일 때 발생하는 문제와 유사



$W^{[3]}, b^{[3]}$ 로 들어오는  $a^{[2]}$ 는  
이전  $W, b$ 에 따라 값이 항상 변함!  
즉, covariate shift 문제가 발생



Batch Norm  $\Downarrow$  hidden unit의 분포가  
slowly 같다 이동하는 양을 줄여줌!

$$\begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} \xrightarrow{\quad \quad} \begin{array}{c} \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array}$$

$$\tilde{z}_1 \quad \tilde{z}_2$$

$\tilde{z}_1$ 과  $\tilde{z}_2$  값은 변할 수 있음

But, Batch Norm이  
 $\mu^{[2]}$ 와  $\sigma^{[2]}$ 의 "평균값"과  
"분산"은  
똑같을 것이라고 보장해줌  
 $(\beta_2 \text{와 } \gamma_2)$ 에 의해 지배되는 값으로

Coupling  $\Leftarrow$  중간의 layer에

영향을 줄 수 있는  
값을 제한시킴.

layer들이

그나마

독립적으로

작동할 수 있도록!

$\therefore$  이전 layer의 parameter가  
업데이트되면서

영향을 줄 수 있는

값을 제한시킴.

$\therefore$  Batch Norm은

입력값이 변해서

생기는 문제를 해결

# Batch Norm As Regularization

(14, 128 ...)

- 각 mini-batch는 해당 mini-batch에 대해서만 계산된 평균과 분산으로 조정된다

⇒ 전처 데이터에서 계산한 것이다

많은 noise가 함유됨

⇒ 이는  $\bar{x}^{[l]} \rightarrow \hat{x}^{[l]}$ 로 가는 과정에서도

noisy한 평균/분산 때문에, noisy가 생김

⇒ hidden layer에 noise를 더해줌

dropout 같은, 아주 작은 regularization 효과가 생김

(noise가 작아서 효과도 ↓)

⇒ Batch Normal + Dropout을 함께 쓰면 효과가 좀 더 나옴

⇒ 아니면 mini-batch의 크기를 키움 ( $64 \rightarrow 512$ )

※ 설명은 했지만,

Batch Norm은 Regularization 옵션을 쓰지마라!

Regularization 옵션을 쓰세요

## Test 시에서의 Batch Norm

Batch Norm은 mini-batch 뿐만 아니라함.

그런데 test 시에 데이터를 한 개씩 처리해야 할 수 있다.

이 때 Network를 어떻게 변경시킬 수 있는지 알아보자

1. 평균 구하기 위해  $z^{(i)}$ 의 mini-batch의 합 구하기

$$M = \frac{1}{m} \sum_i z^{(i)}$$

$\hookrightarrow$  mini-batch 속 data의 개수

(개수를 기준으로  $\Rightarrow$  빛의  $\mu$ 와  $\sigma$ 를 예측하여 적용)

평균/분산을  $\cdot 1$  때 exponentially weighted average를 이용하여

구할 수 있음!

따라

2. 표준차

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - M)^2$$

3. Znorm

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - M}{\sqrt{\sigma^2 + \epsilon}}$$

$$4. \tilde{z}^{(i)} = \gamma \cdot z_{\text{norm}}^{(i)} + \beta$$

이는 hidden layer의 그들의 평균값의

추정치

$\sigma^2$ 은 각 layer에서의  $\sigma^2$ 의 평균으로 추정치로 구함

그림으로 보면

$$\begin{aligned} & X_1^{(t+3)}, X_2^{(t+3)}, \dots \\ & (M^{(13)}[t], M^{(23)}[t], \dots) \Rightarrow \mu \\ & (\sigma^{(13)}[t], \sigma^{(23)}[t], \dots) \Rightarrow \sigma^2 \end{aligned}$$

이렇게 구한  $M$ 과  $\sigma^2$ 을 전해

$$\rightarrow z_{\text{norm}} = \frac{z - M}{\sqrt{\sigma^2 + \epsilon}}$$

$$\rightarrow \tilde{z} = \gamma \cdot z_{\text{norm}} + \beta$$

# Softmax Regression

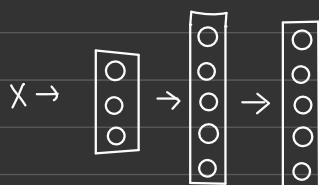
- logistic 회귀의 일반화된 버전

- 1 of 복수의 클래스로 예측 가능

↳ 고양이, 강아지, 명아리, X 주변

↳ 4종류의  $\hat{y}$ 는  $(4, 1)$  shape

## SoftMax layer



$$z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]} \quad \dots \quad (4, 1)$$

Softmax Activation func :  $t = e^{z^{[L]}} \quad \dots \quad (4, 1)$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{i=1}^4 t_i}, \quad a_i^{[L]} = \frac{t_i}{\sum_{i=1}^4 t_i}$$

4개의 확률 합계

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$$

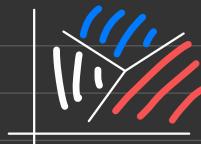
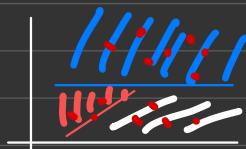
$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$$

$$a^{[L]} = \frac{t}{148.4}$$

## Softmax Examples

$$\begin{aligned} x_1 &\sim \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \hat{y} \\ x_2 &\sim \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{aligned} \quad z^{[1]} = W^{[1]} x + b^{[1]}$$

$$a^{[1]} = \hat{y} = g(z^{[1]})$$



decision boundary 2

unit이 3개  $\rightarrow$  3개의 class를  $\frac{W}{2}$

linear한 모델  $\frac{W}{2}$  가는  
전체

# Understanding softmax

multi-task

Learning 예제 2

$$z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix} \quad \dots \quad c=4$$

$\leftarrow$  softmax  $\leftrightarrow$  hard max

$$t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} \Rightarrow g(z^{[L]}) = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$$

$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$  큰 값만 1로,  
나머지는 0으로

## Loss Function

$\hat{y}$ 의 손실 I

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \xrightarrow{\text{cat}} \text{cat} \quad \alpha^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{array}{l} \text{2번이 비율이 적어} \\ \text{61% 같은 고양이로 분류 못 함} \end{array}$$

$$\gamma_1 = \gamma_2 = \gamma_3 = 0 \rightarrow$$

$$\underline{L(\hat{y}, y)} = -\sum_{j=1}^3 y_j \log \hat{y}_j = -\sum_{j=1}^3 \hat{y}_j = -\log \hat{y}_2$$

$\cdot \hat{y}_2$ 의 값 커지게 되

TensorFlow에서 여러 개인 경우, Shape는 어떻게 될까

$$Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$$

$$= \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & m \end{bmatrix} \quad (4, m)$$

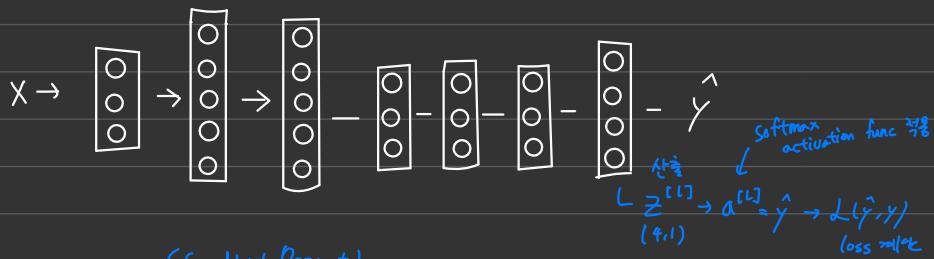
$$\hat{Y} = [\hat{y}^{(1)}, \dots, \hat{y}^{(m)}]$$

$$= \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix} \quad (4, m)$$

## Gradient Descent with Softmax

$$\text{cost } J(w^{(i)}, b^{(i)}, \dots) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

목표는 gradient descent로 cost를 최소화하는 것



(Gradient Descent)

Backprop :

from loss layer  $\rightarrow d\zeta^{(i)} = \hat{y} - y$  \*

$\frac{\partial J}{\partial z^{(i)}}$

# Deep Learning Framework

개발 뿐만 아니라 반복 수행 업무, 생산 배치에도 용이한 Framework 선택하기

## TensorFlow

### Motivating problem

$$\text{cost } J(w) = w^2 - 10w + 25 \text{ 가 } \frac{\partial}{\partial w} \text{를 } 0 \text{ 이 } \rightarrow \text{최소화 } J(w) = 0$$
$$= (w-5)^2 \text{ 일 때 } w=5 \text{ 가 최소가 된다.}$$

np.random.randn()  
14.58 ↴ tensor flow에서 해보면

$J(w, b)$ 를 최소화하는  $w, b$ 를 찾아야 함

```
w = tf.Variable(0, dtype = tf.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

# define a single training step at this func
def train_step():
    # forward prop을 적으면
    # tensorflow가 자동적으로 backprop or Grad Descent 함
    with tf.GradientTape() as tape: # cost를 기록함 - 미분값도 알아서 계산해줌
        # cost function ↪ 미분값 계산, backward하면서 기록
        cost = w ** 2 - 10*w+25
    trainable_variables = [w]
    grads = tape.gradient(cost, trainable_variables)
    optimizer.apply_gradients(zip(grads, trainable_variables))

print(w)

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0>

train_step()
print(w) # 0 → 0.1 up

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=0.0999999>

for i in range(1000):
    train_step()
print(w)
# 1000번 돌리니 cost의 최솟값인 5가 나온다!!

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```

tf를 써면  
그간 cost func와 optimizer만 정해주면 된다

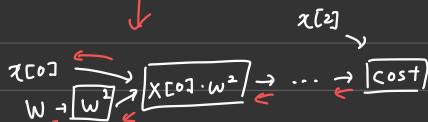
```
# "another version" of how to implement all this
w = tf.Variable(0, dtype=tf.float32)
x = np.array([1.0, -10.0, 25.0], dtype=np.float32)
optimizer = tf.keras.optimizers.Adam(0.1)

def training(x, w, optimizer):
    def cost_fn(): # minimize cost
        return x[0]*w ** 2 + x[1]*w + x[2]
    for i in range(1000):
        optimizer.minimize(cost_fn, [w])
    return w

# print(w)
# optimizer.minimize(cost_fn, [w]) # take a one step of Adam

w = training(x, w, optimizer)
print(w) # 0 → 0.1 up

<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=5.000001>
```



Tensorflow가 자동적으로 진행하면서  
모든 필요한 Backward 계산을 알아냄 → back prop은 꼭 필요X

## <tf.Tensor>

tensor는 numpy array와 같은 것

↪ 차이점 - tensor는 ...

①  $\text{for-loop}$   $\text{for}$

직접적으로 `data`에 접근 불가  $\Rightarrow$  "iter"을 이용하여

"element-spec" = 3

shape과 dtype 확인 가능

"next"를 확인 가능

`print(next(iter(~)))`

② 하나를 바꾸려면 `map()`에

함수를 넣어 각 원소에 적용

ex) `normalize()`

## <tf.Variable>

변수들의 상태를 저장해줌. 초기에  $\text{data}$  type과 shape를 정해야 함

만약 `dtype`을 생략하면 그저 "Tensor" 또는 "convert\_to\_tensor"가 대신 결정함

직접 정해주는 게 가장 좋음

## <tf.constant>

`tf.Variable`과 다르게 상태를 바꿀 수 없음

<tf.sigmoid> ① `tf.cast(, tf.float32)`

<tf.softmax> ② `tf.keras.activations.sigmoid(z)`

(int) label이 가진  
집단의 수  $\lceil$  세로줄 축은  
[ 0 차원에서 생성 ]  
비교 등  
 $\lceil$  (int) Categorical  
labels

## <tf.one-hot>

`def one-hot-matrix (tf.reshape(tf.one-hot(label, depth, axis=0), (depth,))`

`y-test.map(one-hot-matrix)`

## <tf.linalg.matmul>

행렬 곱 연산 사용 (`np.dot()`)

+ 0이 2로 나누어 끝나면

<tf. kernels. initializers. GlorotNormal (seed=1)>

<tf.zeros ()>  
<tf.ones ()>

$$\text{평균 } 0, \text{ 표준편차 } = \sqrt{\frac{2}{\text{fan-in} + \text{fan-out}}} \text{ 인 정규 분포 생성}$$

fan-in은 입력 단위 수, fan-out은 출력 unit의 수. 이는 weight tensor

→  $W = \text{tf. Variable}(\text{initializer}(\text{shape}=(\cdot, \cdot)))$

## Tensorflow 2.0 NN 설계하기

## ① forward prop 구현

## ①-2 : Cost 계산

```
Computes the cost

Arguments:
logits -- output of forward propagation (output of the last LINEAR unit), of shape (6, num_examples)
labels -- "true" labels vector, same shape as Z3

Returns:
Cost -- Tensor of the cost function
"""

logits = tf.transpose(logits)
labels = tf.transpose(labels)
cost = tf.reduce_mean(tf.keras.losses.categorical_crossentropy(labels, logits, from_logits=True))
```

ଲେଇଟର

y\_true y\_pred  
from forward prop

② grad 최수하고 모델 훈련시키기

```

def model(X_train, Y_train, X_test, Y_test, learning_rate = 0.0001,
          num_epochs = 1500, minibatch_size = 32, print_cost = True):
    """
    Implements a three-layer tensorflow neural network:
    LINEAR->RELU->LINEAR->RELU->LINEAR->SOFTMAX.

    Arguments:
    X_train -- training set of shape (input size = 12288, num of train = 1080)
    Y_train -- test set, of shape (output size = 6, num of train = 1080)
    X_test -- training set, of shape (input size = 12288, num of train = 120)
    Y_test -- test set, of shape (output size = 6, num of test = 120)
    learning_rate -- learning rate of the optimization
    num_epochs -- num of epochs of the optimization loop
    minibatch_size -- size of a minibatch
    print_cost -- True to print the cost every 10 epochs
    """

```

```

# Do the training loop
for epoch in range(num_epochs):
    epoch_cost = 0.

    # We need to reset object
    # to start measuring from 0 the accuracy each epoch
    train_accuracy.reset_states()

    for minibatch_X, minibatch_Y in minibatches:
        with tf.GradientTape() as tape:
            # 1. predict
            Z3 = forward_propagation(tf.transpose(minibatch_X), parameters)

            # 2. loss
            minibatch_cost = compute_cost(Z3, tf.transpose(minibatch_Y))

        # We accumulate the accuracy of all the batches
        train_accuracy.update_state(tf.transpose(Z3), minibatch_Y)

        trainable_variables = [W1, b1, W2, b2, W3, b3]
        grads = tape.gradient(minibatch_cost, trainable_variables)
        optimizer.apply_gradients(zip(grads, trainable_variables))
        epoch_cost += minibatch_cost

    # We divide the epoch cost over the number of samples
    epoch_cost /= m

    # Print the cost every 10 epochs
    if print_cost == True and epoch % 10 == 0:
        print ("Cost after epoch %i: %f" % (epoch, epoch_cost))
    print("Train accuracy:", train_accuracy.result())
    # We evaluate the test set every 10 epochs
    # to avoid computational overhead
    for minibatch_X, minibatch_Y in test_minibatches:
        Z3 = forward_propagation(tf.transpose(minibatch_X), parameters)
        test_accuracy.update_state(tf.transpose(Z3), minibatch_Y)
        print("Test accuracy:", test_accuracy.result())

    costs.append(epoch_cost)
    train_acc.append(train_accuracy.result())
    test_acc.append(test_accuracy.result())
    test_accuracy.reset_states()

return parameters, costs, train_acc, test_acc

```

`#_train = Y_train.batch(batch_size, drop_remainder=True).prefetch(8)`  
`# loads memory faster`

batch train에 이걸 추가적으로 적용하면  $\rightarrow$  disk에서 읽을 때 생기는  
 extra.step을 더 빨르게

## ML 개선 방법

- Collect more data
- Collect more diverse training set
  - ↪ Data Augmentation
- Train Algorithm longer with gradient descent
- Try Adam instead of gradient descent
- Try Bigger Network
- Try Smaller Network
  - ↓
  - Try Dropout
- Add L<sub>2</sub> Regularization
- Network Architecture
  - Activation functions
  - \* hidden units, layers.

많은 방법이 존재함.

어떤 상황에 유효한지 먼저 해보면

좋을지를 이제부터 알아볼 예정

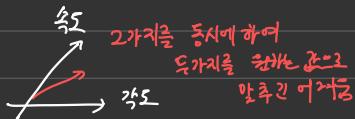
# Orthogonalization

자동하는 특정 각도와 속도로 운행할 수 있다

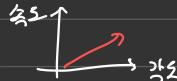
현실에선 1개의 방향 조정 장치, 불개의 속도 조절 장치를 따로 가짐

직교화한 두 가지를 한 번에 표현

일차원적으로 방향 조정과,  
이차원적으로 속도 조절



그런데 그자가 직교화한라면 속도를



지도학습 시 확인해야 할 것

1. train set에서 제대로 작동하는지 확인  
→ 인간과 준하는 성능인지 비교  
[ 이들을 조절할 장치 (손잡이) ]  
중복 구현하지 않고  
되도록 적은 수로 두어야 함  
(+ 최적화)  
ex) 직사각형 크기 조절이  
각 봉에 대해서만 X  
전체 크기가 변화도록 O
- = 2. dev set에서 "  
[ ] → 모니터 조절장치 ]  
쓰자  
ex) 더 큰 dev set  
구하기
3. cost 적용된 test set에서 "  
[ ]  
devel set or  
비주기  
분포가 잘 맞음
4. 실세계에서 잘 되는지  
[ ]  
더 큰 dev set  
구하기  
cost 계산을  
잘 못 함

## Single Number Evaluation Metric

실수(R) 평가 지표를 만들자!

accuracy  $\left[ \begin{array}{l} \text{정밀도} \\ \text{precision : for train} \\ \text{재현율} \\ \text{recall : for dev / test} \end{array} \right]$  이것들로 평가 지표를 만들지 않음

어떤 classifier는 precision에서, 어떤 건 recall에서 더 좋기 때문.

여러 classifier의 hyperparameter를 시도하고, classifier도 여러 개 시도해보며 계측 반복하기

### <Classifier 고르기>

precision과 recall을 합쳐 새로운 평가 지표 생성.

이제 두개 보통 F1 score를 이용.

이는 precision P와 recall R의 평균수치.

$$= \left( \frac{2}{\frac{1}{P} + \frac{1}{R}} \right) : \text{Harmonic mean}$$

즉, P와 R의 평균값

### <또 다른 예시>

| Algorithm | US | China | India | Other | Average |
|-----------|----|-------|-------|-------|---------|
| A         | 3% | 7%    | 5%    | 9%    | 6       |
| B         | 5% | 6%    | 5%    | 10%   | 6.5     |
| C         | 2  | 3     | 4     | 5     | 3.5     |
| D         | 5  | 8     | 7     | 2     | 5.25    |
| E         | 4  | 5     | 2     | 4     | 3.75    |
| F         | 7  | 11    | 8     | 12    | 9.5     |

수많은 소리를 보기 일단 평균을 계산하여 한정하기

• 1회

↳ 평균이란 하나의 상수를 계산하여 평균

3.5

# satisficing      Optimizing

## 최소한의 충족과 최적화 방법

< 또 다른 Cat+ 블루 예제 >

| Classifier | 이 둘은 경쟁자끼리<br>공정 평가 수치를 만들 수 있지만, 일관적인 느낌 |              |
|------------|-------------------------------------------|--------------|
|            | (F1 score 등)<br>Accuracy                  | Running time |
| A          | 90                                        | 80ms         |
| B          | 92                                        | 95ms         |
| C          | 95                                        | 150ms        |

Satisficing metric [ 정확도는 적당히 좋으면 됨 ]      이이치 편법에 최소 100ms 이내 소요 ] = optimizing metric

< Wake word for 음성인식하기 >

trigger word에 잘 반응하는가?  
우작위로 갑자기 기기가 스스로 인식하는 경우 ) → 두 수치를 경쟁하여 대로 지표 생성

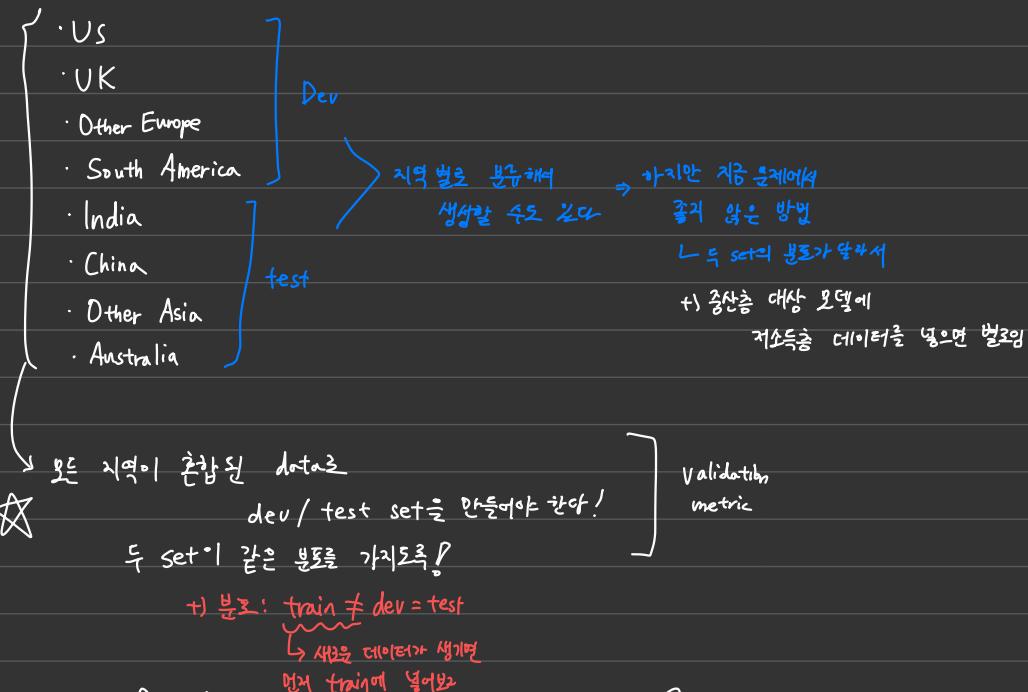
⇒ 그 결과를 바탕으로

고객간 대화 | 기기의 | 끊임없이 경지하는 등으로 사용자

hold out cross-validation set

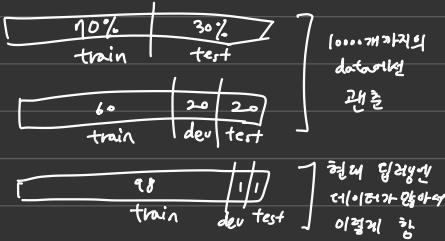
## Train / Dev / Test Distribution

Cat Classifier를 만들어서 아래 지역에 적용한다고 하자



## Size of dev

### Old way of splitting data



## Size of test

높은 confidence를 얻기 위해  
test set은 충분히 크게 잡자  
data를 분류해서 만들어 짐승에 이용한다면  
그것은 dev set.  
그러나 test set을 생략하기도 하지만  
꼭 적용해보자

## When to change dev/test sets and metrics

특정 목표를 향해 진행하다가 잘못되었을 깨달았을 때  
목표를 이동시킬 수 있음

예로 고양이 판별  
A, B 알고리즘 중 A의 오류가 적었지만

B가 훨씬 알고리즘을 잘 걸려면, 어느 걸 쓸지 담이 때

(S.1.1) dev set / test set 바꿔야 함

(S.1.2) 평가 metric 바꿈

$$\text{포도노 이미지} \leftarrow \sum_{i=1}^{m_{\text{dev}}} W^{(i)} \{ y_{\text{pred}}^{(i)} + y^{(i)} \} \Rightarrow \begin{array}{l} \text{분류를 못한 사례들이} \\ \text{총 개수를 합하기} \end{array}$$

판단용 가중치를 추가함

$$W^{(i)} = \begin{cases} 1 & \text{일반} \\ 0 & \text{포도노} \end{cases}$$

이걸 수정하는 수도 있음

사실인 것의 수를 세는 항수

prediction value (0 or 1)

(S.1.3) Weighting 항수 추가하기

∴ Orthogonalization for cat img : ~~제작~~

1st : 목표 설정에 따른 평가 지표 정의하기

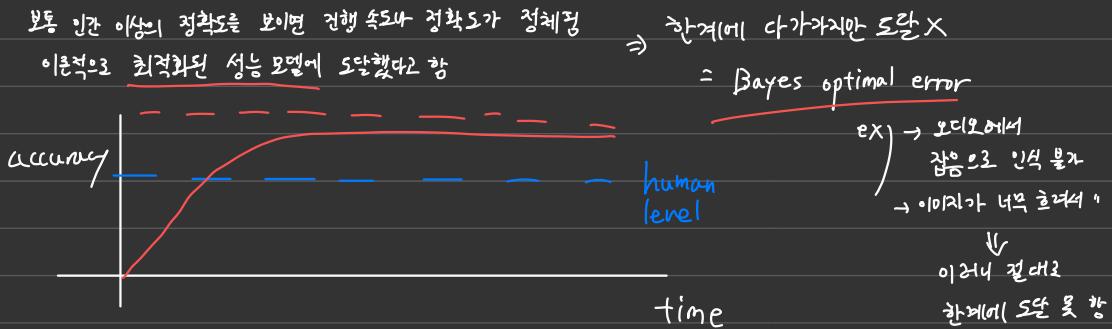
2nd : 평가 지표에 따른 목표로 조준하고 맞춰보기

∴ 그러나 dev/test set 분포와 크기를 잘 설정하고

목표에 따른 평가 지표를 잘 세워야 함

## Why Human-level Performance?

- 이유 - ① 딥러닝의 발전으로 인해, 머신러닝이 더 좋아질 때가 있어서  
② 머신러닝을 설계하고 수행절차를 설정하는 과정이 많이 발전해서



그리고, 인간 완전 성능을 초과할 때 적용할 만한 도구가 없음

## Avoidable Bias

인간 성별에 따라 어느 정도의 정확성을 가지면 좋은지 알아보자

예로, 고양이 분류에서 정확도 다음과 같다면,

i. 훈련  
차이를 의미 [ 인간 : 1% ] train error : 8% > 이 훈련 차이가 크면  
이는 Bayes dev error : 10% ⇒ bias와 편차를 떼여야 함. 여기서 bias부터 고려해보자  
error보다 낮아질 수 없으

< bias 피하기 in train set >

1. 더 큰 신경망을 train하기
2. 더 오랜 시간동안 train하기

또 다른 예로, 흐린 이미지를 대상으로 하여 인간의 오류도 높다면

7.5%보다  
더 낮게 [ 인간 : 7.5% ]  
차리고 dev error : 10% > 이 끗의 차이를  
줄일 수 있도록 해야함 ⇒ (설명) 불필요에 충분! ⇒ train data를 더 수집  
노력하자마자 ~ ⇒ regularization

# Understanding "Human-level Performance"

(방사선)

ex) 의학 이미지 분류 사례

여기서  
일반인은 3% 오류, 의사 1% 오류를 보임  
경험 있는 의사 0.1%, --- ) 이걸 때 '인간'의 기준은?  
Proxy 또는 Bayes Error로 접근해보자

Proxy 또는 Bayes Error의 추정 값을 청하는 경우,

경험 있는 의사를 기준으로 할 때, Bayes Error  $\leq 0.1$

$\therefore$  bias와 편차(분산) 복적으로

인간 성능 지표를 정의할 땐 최고의 인간의 탄란의 오류를 기준으로 하자

그런데 논문의 목적이나 시스템에 도입하려고 한다면

목표 지점에 따라 다르게 정의해야 함

예시 2

인간  
 $1\%$   
 $0.1\%$   
 $0.5\%$   
 train : 5%  
 dev : 6%

일단 bias가 더 큰 dev set이기  
 높으면 dev set에 overfitが多い 학습

$\uparrow$   
 available bias  
 약  $4 - 4.5\% \Rightarrow$  뭘로 하면  
 차이가 크지 않다면  
 중립화해 고려할 필요X

예시 3

인간 : 0.5%  
 train : 0.1% ] 인간 < train 으로  
 인간 여러 설정해!  
 dev : 0.8%  
 $0.1 < 0.2$  이므로  
 available bias에  
 적용해보기

< 정리 >

- Human-level error : bias + variance 관란에 이용

available bias < 인간  
 variance < train  
 variance < test

# Surpassing human-level performance

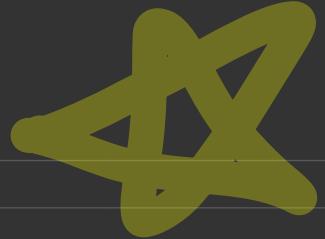
으사 : 0.5% )  $\rightarrow$  둘다 train 보다 커서  $\Rightarrow$  그 이상으로 뛰어나게  
 인간 : 1% ) bias 와 variance 중  
 특성을 정도로 지 결정 불가 발전시키는 것이 어려울 뿐  
 train : 0.3% 양축쳤다면 괜찮게 설계한 것  
 dev : 0.4%

실제 세계에서 다루는 인간 능가 수준의 문제들

1. 대출 승인 여부 판단  
 2. 온라인 맞춤 광고  
 3. 제품 추천  
 4. 이동 시간 계산

⇒ 이들은 구조화된 데이터를 활용  
 자연적으로 인지(X). 인공적으로 생성한 데이터 기반.  
 ⇒ 방대한 양의 데이터를 다룬  
 ↳ 이걸 때 심지어 인간의 생각도 패턴화하여 대처함

# Improving your Model Performance



기본적인 시도

1. train set을 잘 대상화하기

= 낮은 avoidable bias를 찾는다

⇒ train error 와

Bayes error 를 비교하면 ⇒ 2 경우에  
avoidable bias 확인

2. Organization을 고려한 모델

조절 장치를 이용하여 충분 시간 늘리기

(Variance 확인)

< 편차가 문제라면 >

3. Regularization

1. 더 많은 데이터 수집

4. 더 많은 data 수집

2. regularization

5. 더 큰 모델로 충분시키기

└ dropout

6. 더욱 향상된 알고리즘 쓰기

└ data augmentation

↳ Adam, RMSprop, Momentum

3. hyper parameter 조정

4. 세트로 구조 찾기

7. 더 나은 구조를 찾기

↳ 더 좋은 hyper parameter

↳ layer 수

↳ unit 수

↳ activation func 종류

↳ 모델 구조 변경 (Chaining)

# Carrying Out Error Analysis

< Look at dev set to evaluate ideas >

ex) 고양이 분류

accuracy : 90%.

dev error : 10%. - 이 수치가 예상보다 높다고 걱정하자

↳ 분류를 못한다는 의미 : 예로, 강아지를 고양이로 간주해버림

⇒ 강아지에 대해 일관성을 강화해볼 수 있음

강아지에 대해 focus 할지 결정할지 고민하기

이는 수치들이 걸리고 결과도 좋지 않을 수 있음

그러면,

→ 대략 100개의 잘못 분류된 dev 샘플을 수동으로 정사화하라 ) 5~10분 밖에 안 걸림

그 중에서 몇 개가 잘못 분류된 걸지 알아내 확인

① 예로, 100개 중 5%가 강아지 이미지라면,

5개만 수정할 수 있음. 이 때 강아지에 집중하면

(0% → 9.5%로 오류가 줄어듦.

이제 복습하다면 강아지에 집중하면 낫겠지

"ceiling on performance"

오류 이미지가

전체 암으면 거기에 집중하라고

② 100개 중 50개가 강아지 이미지라면

이 때는 강아지에 집중하여

(0 → 5%로 오류가 감소)

ex2) 고양이 계열 - 사자, 토끼, 치타 분류

↳ 작은 고양이, 징고양이로 인식되는 것을 막음 or 초깃한 이미지도 잘 되도록

⇒ 엑셀로 오류 분석 테이블을 만들자! (몇 시간안 걸아-넣어 ^^)

| Image | Dog | Great Cat | Blur | 비교                      |
|-------|-----|-----------|------|-------------------------|
| 1     | v   |           |      | 분류                      |
| 2     |     | v         | v    | 이는 분류는 사자               |
| :     |     | v         |      |                         |
| :     | :   | :         | :    |                         |
| Total | 8%  | 43%       | 61%  | ⇒ 이를 통해 어디에 focus 할지 알면 |

카메라 필터도  
하나의 고려 항목으로

넓이도 넓을 듯

# Cleaning Up Incorrectly Labeled Data

처음부터 잘못된 입력값 : incorrect label

강아지는 인데 1로 된 이미지 가짜

1st. Train set에서 정상 작동하는가

↳ 딥러닝은 Random Error에 강함 → 정상 작동이면 놔둬도 OK

→ 엑셀로 표 만들자 ^^

| Image | ... | Incorrect label | 1%<br>고등비그먼<br>배경을 잘못인식 |
|-------|-----|-----------------|-------------------------|
| i     | ... | 6%              |                         |

↳ (cost →)  
dev set에서 현저히 개선될 수 있다면 시간을 들여 data를 수정하자.  
그게 아니면 시간 낭비가 될 지나...

일단 이 세 가지를 고려하자

Overall dev set error --- 10%  
Errors due to incorrect labels --- 0.6%  
Errors due to other causes --- 9.4%

2%

0.6% → 전체의 0.3%가 오류

1.4%

↳ dev set의

incorrect label 수정하기

잘못된 label은 accuracy/error  $\frac{1}{2}$

너무 신뢰하지 않기

Be Cautious to correct

Incorrect dev/test set

① 같은 분포를 가진 dev와 test에

똑같은 수정 과정을 적용하기

② 틀린 데이터를 유심히 살펴보기

↳ 수정해야 할 것이 있는지 보기

보델이

단순히 운영체제

알고리즘 작동부록

↳ 틀린 것만 고치고 나면 bias estimates 와 노드를 신호

③ train과 dev/test의 분포가 달라짐에 유의하기  $\Rightarrow$  다음 장에 계속

Build your First System Quickly, then Iterate

⇒ 여러 가지 고려할 사항에 우선순위를 잡다가  
지지부진하지 않고 일단 만들고 테스트하기를 빨리하되

⇒ 즉, (st) dev/test set 준비, 평가 지표 정하기  
⇒ 목표를 어디에 둘지 결정하기

2nd) 빠르게 첫 모델 만들기

3rd) Bias / Variance 분석과 Error 분석으로  
다음에 무엇부터 할지 정하기

# Training and testing on different distributions

목표: 훈련으로 짹은 고양이 패밀리

↳ 아마존에서 짹은 고양이 사진: 10,000장

↳ 웹 크롤링한 고양이 사진: 200,000장

비주

목표에 맞는 데이터가

2) 목적할 때는

다른 분포의 데이터가

무작위로 섞어서 학습시킴

train | dev | test  
205,000 | 5000 | 5000



(주천 1)

목표에 부합하는 데이터를 모두 dev/test로

train set: 총 200,000장

dev/test: 모바일 5000장 씩

↳ 단점: train과 dev가 다른 분포.

그리고 저거보다 훨씬 나음

↓ (장점: train/dev/test 모두 같은 분포)

단점: dev set에 목표에 맞지 않는 데이터 많

↳ 목표에 속하지 않기에 비주천 ㅠㅠ

# Bias and Variance with Mismatched Data Distributions

bias와 variance는 업무 순위를 결정하는 함.

그런데 train과 dev set이 다른 분포에서 왔다면 bias와 variance를 분석하는 방법이 달라짐

ex) 고양이 분류

인간 (Bayes error): 0%

train: 1%      ) train과 dev가 서로 같은 분포에서 왔다면

dev: 10%      이는 아주 큰 variance를 가진 문제임

train  $\neq$  dev 라면,

즉, train set을 잘 일반화 시키지 못하고

dev에선 초반부에만 잘 함

한쪽만 편향이 잘 될 수도 있음

1. 알고리즘이 train data는 봤지만 dev data는 본 적 X

2. dev와 test의 분포가 달라서 생긴 오류인지

얼만큼이 dev를 못 봐서 생긴 오류인지 알 수 없음  $\Rightarrow$  variance

개선

"training-dev set" 사용 정의

서로 일부 추출하여 만들기에 동일한 분포도.

그러나 이는 훈련시킬 때 안됨. 오류 분석 시에 함께 참고할 용도



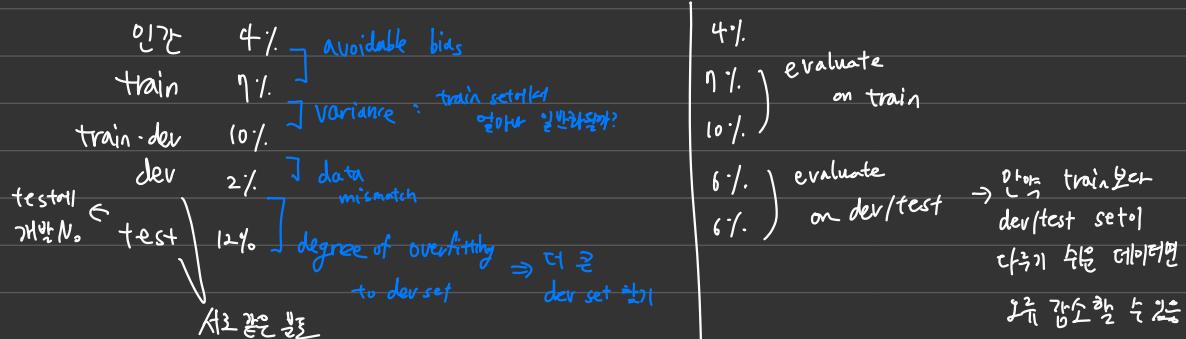
훈련 안 된 데이터  $\leftarrow$  ex) train error: 1%      variance  
                        train-dev error: 1%      이렇게 넓어가면서 오류가 증가함.  
                        dev: 10%      즉, 같은 분포로 추정해도  
                        데이터에 일반화가 잘 되지 않음

1%.      dev에서 2%라면  
1.5%.      데이터 mismatch 문제.  
10%.      train-dev와 dev는 서로 다른 분포이다.

인간: 0%      인간보다 못한 성능이라서  
train: 10%      "Avoidable bias problem".  
train-dev: 11%      high bias setting  $\frac{1}{2}$  같애임  
dev: 12%       $\frac{1}{2}$

0%.      Avoidable bias가 높음  
10%.       $\Rightarrow$  train-dev도 각각 별로  
11%.      variance  
12%.      Data-mismatch

## < Bias / Variance on Mismatched train and dev/test set >



## < More General Formulation >

| error       |  | 多样한 훈련 data          |            | 차량 백미러 관련<br>인증에 따른 데이터를 훈련하는 경우 훈련 데이터 |                  | ex) 목표<br>백미러에<br>응답을<br>기반으로 |                                 |
|-------------|--|----------------------|------------|-----------------------------------------|------------------|-------------------------------|---------------------------------|
| Human       |  | 4%                   | (Train) 7% | avoidable bias                          | 6%               |                               |                                 |
| + trained   |  | (Training-dev error) | 10%        | variance                                | (Dev/test error) | 6%                            | 여기서 back prop 적용X<br>그게 맞는 문제는? |
| not trained |  |                      |            | data-mismatch                           |                  |                               |                                 |

## < How to handle Data-Mismatch >

⇒ cross val

# Addressing "Data Mismatch"

1. 일일이 <sup>작정</sup> error를 분석해보고 train과 dev/test의 차이를 이해하기

test set이 overfit하지 않으면, 오류 분석을 통해, dev set만 수동으로 확인해야 함

⇒ 예2, dev ≠ train의 문제가 아니라

dev set이 소용이 많은 것일 수 있으니

⇒ dev set을 파악하면 dev와 train set이  
어떻게 다른지 알 수 있음

⇒ dev/test와  
우사<sup>3</sup>은 데이터<sup>2</sup>를  
수집하기)  
↳ 소용 많은 데이터 수집  
  
<sup>[1st]</sup> for dev  
<sup>[2nd]</sup> for train

or  
train set이 고의적으로  
알아듣지 못하게 or 너무 크게  
말한 데이터 추가

↓ (작동차)

v 목소리나 일어날 법한 소음을 합성하여 사용

⇒ 높은 풍질의 합성본인치 추적하기

⇒ 인간 눈에서만 놓다고 판단하는 데이터면 안 됨 꼬

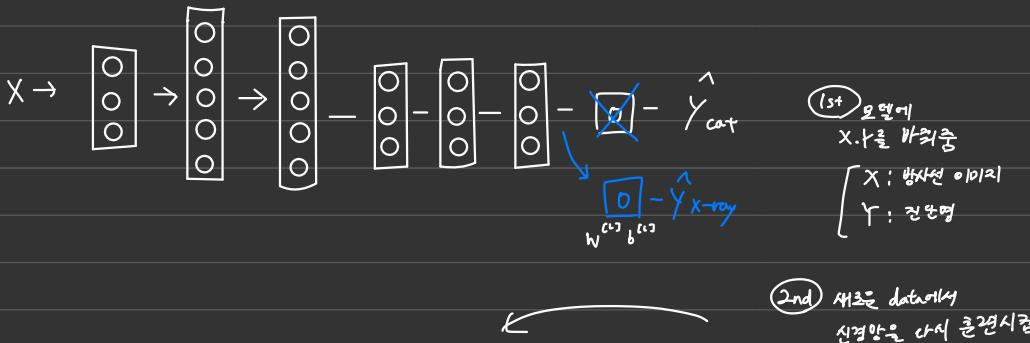
v 차량 소용 알고리즘 개발하기 (1000시간의 자동차 소용 필요)

## Transfer Learning

딥러닝의 장점은, 배운 지식을 다른 업무에 적용 가능하다는 점

예로 고양이 완판 알고리즘으로

부분적으로 X-ray 읽는 알고리즘을 재생산할 수도 있음



if 방식은 이이이(데이터가 적다면,  
(or 2개짜리)  
이-지-막 러이이이) W, b 만은 다시 훈련  
나이이 parameter는 그대로 두고.)

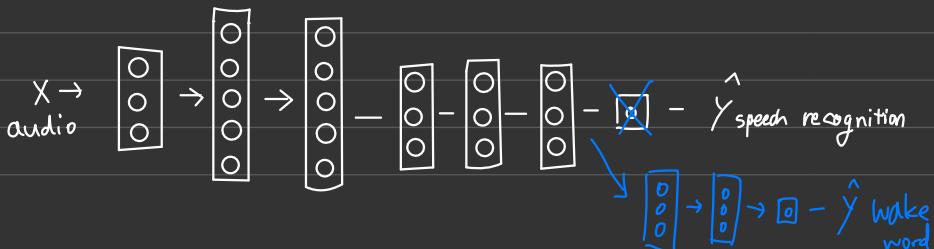
Transfer  
(earning 시  
사용되는 용어)

[ "pre-training" → "fine tuning"  
이미지 임식까지만      방사선 이미지로 특징화 ]

edge 을, positive object 을 놓치하는 특성과 함께  
수 많은 low-level 특성 덕분에强大이 된다

즉, 이미지에서 선, 점, 육선을 바탕으로 놓고 할 수 있게 되었으니.

< 또 다른 예시 - 오디오 > — 여러 개의 새끼는 흥을 불일 수도!



Transfer Learning 을 쓰면 좋은 시점

(low-level)

기초적인 인식 데이터는 많는데,

그와 관련된 특정 분야로 축협을 때 데이터가 적어지는 경우

쓰지 말아야 하는 상황

데이터가 더 많아지는 case라면 굳이?

# Multi-task Learning

transfer learning A업무  $\rightarrow$  B업무라는 순차성이 있지만

Multi-task는 두 업무가 동시에 시작

$\Rightarrow$  여러 가지 길을 배우게 하여 다른 업무도 도와주게 함

(transfer 보여  
직제 쓸)

ex) 자율주행차

보행자, 다른 차량, 표지판, 교통신호 등을 잘 감지해야 함



이들이 복합적으로 이미지에 나타나는 정도 유의!

그러니  $y^{(i)}$ 는 4개의 레이블이 필요  $\Rightarrow$  이 사진이 차량이다! 가 아님

$$\text{그럼 } Y = \begin{bmatrix} y_1^{(i)} & \dots & y_n^{(i)} \end{bmatrix}$$

차량도 있고, 신호도 있고 식으로  
돌려야 할 때!

그래서 하나의 이미지가 4개의 레이블을 가짐

$$\text{이 때 Loss} = \hat{y}^{(i)} - \text{평균 Loss} = \frac{1}{m} \sum_{i=1}^m \sum_{j=1}^4 (\hat{y}_j^{(i)} - y_j^{(i)})$$

즉각  
logistic loss

$$Y = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} - y_j^{(i)} \log \hat{y}_j^{(i)} - (1 - y_j^{(i)}) \log (1 - \hat{y}_j^{(i)})$$

딱 1만 덧셈

이런 일은  $\rightarrow$  이런 multi-learning 대안

4개를 각각 따로 훈련시키기 보단

한 개의 NN이 4개의 일을 하도록 해야 한다.

1, 0, ?  $\Rightarrow$  hard max

Multi-learning의 장점으로

몇 개만 제대로 인식하면 서먹을 수 있다는 점~

그리고 판정 예측하고 ?도 각각 봄 가능~

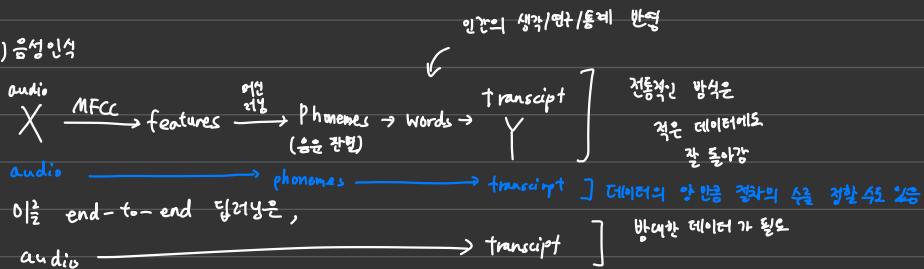
Multi-Learning을 적용해야 하는 상황 조건

1. 여러 레이블을 한 번에 (low-level)라 연결될 땅인가?
2. 종종 각 레이블에 대한 data 양이 비슷한 때
3. 모든 것에서 잘 하도록 큰 신경망을 운영해야 할 때

# What is "End-to-end Deep Learning"?

여러 단계를 거쳐서 하나의 NN으로 명령하는 것

ex) 음성 인식



ex) 얼굴 인식

1st) 이미지 상에 어디에 얼굴이 위치해 있는지 알아내는 봉출기 먼저 만들기



2nd) 그 얼굴이 누구인지 판별하는 봉출기 만들기

↳ 등록된 10000장의 이미지와 빠르게 비교하여 찾기

3rd) X → Y  
이미지 → 신체

여기서 end-to-end Good! : X-Y 데이터가  
아주 많아서

ex) 기계 번역

영어 → 텍스트 추출 → ... → 프랑스어

ex) X-ray로 아이의 나이 추정하기

1st) 골격 구조 및 인식하는 것부터

2nd) 평균 뼈 길이를 계산하여 나이 추정

↳ 아이들 손 끝에 대학 통계치를 이미 가지고 있다면

데이터가 많이 필요하지 않음

→ 이 문제는 골이 end-to-end 한 쪽도 꿈

# When to Use End-to-End Deep Learning?

(장점)

1. 원夙한 학습으로 데이터가 할 수 있도록 하는 것

ex) 모델이 운동학을 바탕으로 소리내어를 흡적 X

어떤 풀현을 해석해 할 수 있음

2. 세세한 사항의 설계가 더 적다는 것

→ 작업을 단순화함

(단점)

1. 많은 데이터 필요

↳ X-Y 맵핑까지 되어야 함

2. 세세한 사항의 설계 방식이 생략될다는 것

Ex) 자율주행

여기선 planning을 이용하는 게 더 나음

이미지  
[ ]  
→ 모형자  
→ 자동차

↓  
> 경로 → 가속 및 제동

T  
:

여기엔

Deep Learning을 쓰지만

[주의]

- 복잡에 맞는 X-Y를 잘 가려야 함

- 이 방식이 가장 좋은 방법 X  $\Rightarrow$  복잡한 접근 방식이나 많고 원夙한 학습!

# Convolutional NN

Computer vision ↗ 아주 큰 데이터...! ↗ 이를 효율적으로 처리하자!

## Edge Detecting

- 수직선 / 수평선 감지

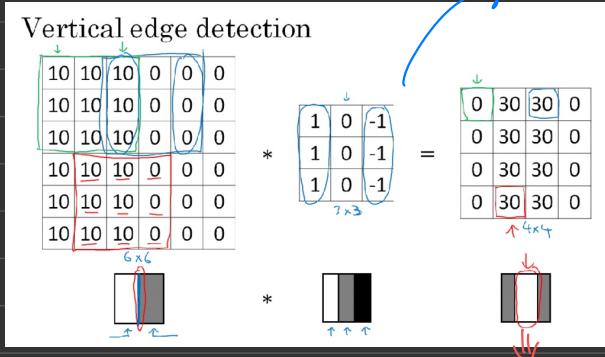
전체 이미지에 필터(kernel)을 convolve 하기

\*를 흑색이 아닌 convolve하고 보기!

수평선 필터는

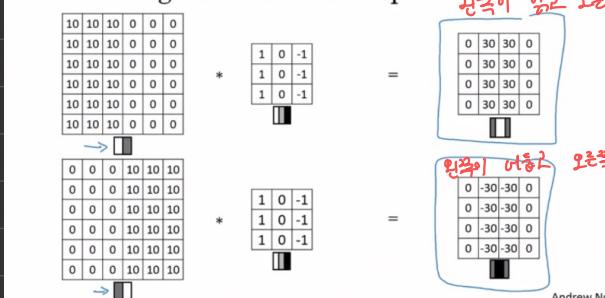
$\begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$  → 왼쪽이 밝으면  
양수: 흰색 선

아래쪽에  
검증되어  
음수가 나오면  
검은색 선



이미지 중앙에  
수직선이 있음을 감지

Vertical edge detection examples



왼쪽이 밝고 오른쪽이 어두우면

+30 ⇒ 밝은 선을 찾았어

왼쪽이 어두우고 오른쪽이 밝으면

-30 ⇒ 어두운 선을 찾았어

이는  $\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$  필터와

Andrew Ng

Vertical and Horizontal Edge Detection

|   |   |    |
|---|---|----|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

Vertical

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 10 | 10 | 10 | 0  | 0  | 0  |
| 10 | 10 | 10 | 0  | 0  | 0  |
| 10 | 10 | 10 | 0  | 0  | 0  |
| 0  | 0  | 0  | 0  | 10 | 10 |
| 0  | 0  | 0  | 10 | 10 | 10 |
| 0  | 0  | 0  | 10 | 10 | 10 |

\*

|    |    |    |
|----|----|----|
| 1  | 1  | 1  |
| 0  | 0  | 0  |
| -1 | -1 | -1 |

Horizontal

|    |    |    |
|----|----|----|
| 1  | 1  | 1  |
| 0  | 0  | 0  |
| -1 | -1 | -1 |

=

|   |    |   |   |   |   |
|---|----|---|---|---|---|
| 0 | 10 | 0 | 0 | 0 | 0 |
| 0 | 0  | 0 | 0 | 0 | 0 |
| 0 | 0  | 0 | 0 | 0 | 0 |
| 0 | 0  | 0 | 0 | 0 | 0 |
| 0 | 0  | 0 | 0 | 0 | 0 |
| 0 | 0  | 0 | 0 | 0 | 0 |

Andrew Ng

최고우선에의  
나타내는  
Horizontal 흐름 공간에서  
1000x1000 이미지에  
거의 안 보임

Filter ...

$$\begin{matrix} 1 & 0 & 1 \\ 2 & 0 & -2 \\ 1 & 0 & 1 \end{matrix}$$

*<Sobel filter>*

중앙 가로줄에 강조를 주어

증여 경고하게 만듬

$$\begin{matrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{matrix}$$

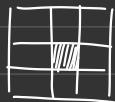
*<sharr filter>*

실제로는... 이 filter의 숫자마저  
parameter로 두어

back prop 하며 Update한 수 있는  
↳ 45°, 70° 모서리도 잘 인식하게...

이걸 90°로 돌리면 수직선 감지 ~ ^~

★  $(f, f)$  : 홀수 개만큼으로 만드는 이유!



홀수개면 중앙에 특징점을 가지기 때문.

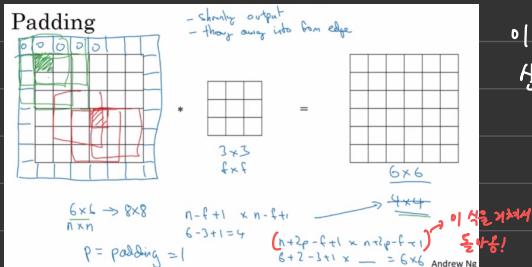
3x3이 흔하고

5x5, 7x7도 종종 쓰임

불이다

# Padding

앞에서 필터 쪽에 약아진 이미지를 원래 크기로 되돌려?!



이 때 가장자리 픽셀들은

상대적으로 중심부보다 적게 사용됨

⇒ 해결 방안

① 전체 이미지에 가장자리 만들기

; 한 경계면  $p=1$   
padding

= "deep border"

얼마나 덧붙일 수 있는가?

## ① Valid Convolution

: no padding ( $p=0$ )

$$(n, n) * (f, f) \rightarrow (n - f + 1, n - f + 1)$$

## ② Same Convolution

: pad. 입력과 출력의 크기가  
같게 하려고

$$\left\{ \begin{array}{l} (n + 2p - f + 1), (n + 2p - f + 1) \\ \hookrightarrow (n, n) 이미지에 p만큼 padding 하면 - \\ \text{입력 } n = n + 2p - f + 1 \text{ 을 계산하면} \\ \therefore p = \frac{f-1}{2} \\ \therefore \text{padding은 필터크기의 } \frac{f-1}{2} \text{ 만!} \end{array} \right.$$

# Strided Convolutions

Strided convolution

$\begin{matrix} 2 & 3 & 7 & 4 & 6 & 2 & 9 \\ 6 & 6 & 9 & 8 & 7 & 4 & 3 \\ \vdots & & & & & & \\ 3 & 4 & 8 & 3 & 8 & 9 & 7 \\ 7 & 8 & 3 & 6 & 6 & 3 & 4 \\ 4 & 2 & 1 & 8 & 3 & 4 & 6 \\ 3 & 2 & 4 & 1 & 9 & 8 & 3 \\ 0 & 1 & 3 & 9 & 2 & 1 & 4 \end{matrix} \quad * \quad \begin{matrix} 3 & 4 & 4 \\ 1 & 0 & 2 \\ -1 & 0 & 3 \end{matrix} = \begin{matrix} 91 & 100 & 83 \\ 69 & 91 & 117 \\ 44 & 71 & 74 \\ 3 & 3 & 3 \end{matrix}$   
 stride = 2       $\lfloor \frac{n}{s} \rfloor = \text{floor}(2)$        $\text{Unk}$   
 $n \times n \quad * \quad f \times f$   
 padding p      strides s  
 $s=2$   
 $\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$   
 $\frac{2+0-3}{2} + 1 = \frac{4}{2} + 1 = 3$   
 Andrew Ng

stride 1, padding 영역은 포함하지 않도록 주자!

↑ 만큼 padding한  
 $(n,n)$ 이면  
 $(f,f)$  필터가  
↓ 만큼 stride를 쓰면

Output의 shape는

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

교차 상관관계

+ Cross - Correlation vs Convolution의 차이점

Technical note on cross-correlation vs convolution

Convolution in math textbook:

$(A * B) * C = A * (B * C)$  을 이용

$\begin{matrix} 2 & 3 & 7 & 4 & 6 & 2 \\ 6 & 6 & 9 & 8 & 7 & 4 \\ 3 & 4 & 8 & 3 & 8 & 9 \\ 7 & 8 & 3 & 6 & 6 & 3 \\ 4 & 2 & 1 & 8 & 3 & 4 \\ 3 & 2 & 4 & 1 & 9 & 8 \end{matrix} \quad * \quad \begin{matrix} (3) & (4) & (5) \\ 1 & 0 & 2 \\ -1 & 9 & 7 \end{matrix} = \begin{matrix} 7 & 9 & -1 \\ 2 & 0 & 1 \\ 5 & 4 & 3 \end{matrix}$   
 $(A \times B) * C = A \times (B \times C)$   
 Andrew Ng

↓  
 그러면 수식/수평  
 둘 다 축복임      이렇게 중복되는 과정을  
 전부하고 진행한 상태

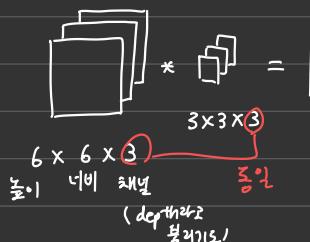
그러나 지금까지의 설명은 cross-correlation임.

그런데 DNN에서 흔히들 이를 convolution이라고 함

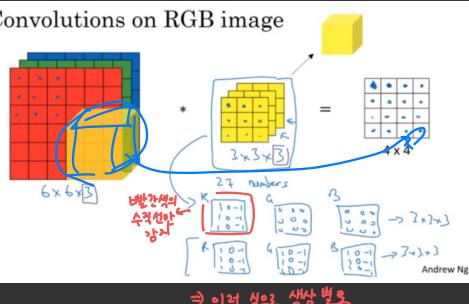
# Convolution over 3차원

RGB 이미지는  $6 \times 6$ 에서  $6 \times 6 \times 3$ 이 됨

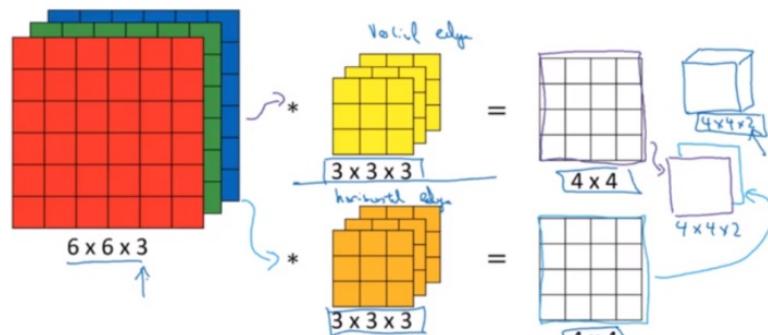
2중 필터도  $3 \times 3$ 에서  $3 \times 3 \times 3 = 1$  됨



## Convolutions on RGB image



## Multiple filters



Andrew Ng

여러 종류의 filter를 적용하고 싶다면

각각의 filter를 통과한 후 합침.

∴ 결과의 차원 수는

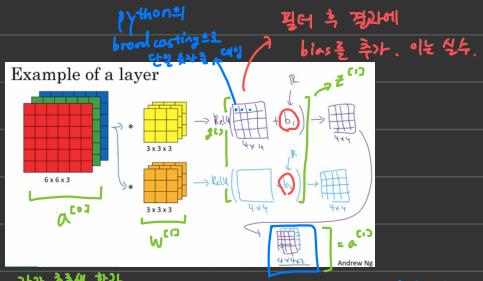
적용한 filter의 수와 동일하다

$$\therefore n \times n \times n_c * f \times f \times n_c \rightarrow (n-f+1) \times (n-f+1) \times n'_c$$

$\underbrace{6 \quad 6 \quad 3}_{=}$        $\underbrace{6 \quad 6 \quad 3}_{=}$       4      4      2

필터의 개수

# Look up One Layer of a Convolution



ex)  $3 \times 3 \times 3$  크기의 필터 10개가 있다면  
이 layer의 parameter의 수?

$$\begin{aligned} & 3 \times 3 \times 3 \\ & + \text{bias} \\ & = 28 \text{ parameter} \end{aligned} \quad \left. \begin{array}{l} \cdots 10^2 N \\ \downarrow \\ \therefore 280^2 N \end{array} \right)$$

## Convolution 관련 (포기법 정리)

$L$ : convolution layer

$f^{[L]}$ : filter size

$p^{[L]}$ : padding

$s^{[L]}$ : stride

$n_c^{[L]}$ : num. of filters

높이 너비 채널 (필터 개수)  
Input:  $n_h^{[L-1]} \times n_w^{[L-1]} \times n_c^{[L-1]}$

Output:  $n_h^{[L]} \times n_w^{[L]} \times n_c^{[L]}$

각각의 차원 계산 공식

$$n_h^{[L]} = \left\lfloor \frac{n_h^{[L-1]} + 2p^{[L]} - f^{[L]}}{s^{[L]}} + 1 \right\rfloor$$

¶

[차원 계산 공식]

Each filter is,  $f^{[L]} \times f^{[L]} \times n_c^{[L]}$

입력 차원 크기

= 필터 크기 크기

Activations:  $A^{[L]} \rightarrow n_h^{[L]} \times n_w^{[L]} \times n_c^{[L]}$

$A^{[L]} \rightarrow m \times n_h^{[L]} \times n_w^{[L]} \times n_c^{[L]}$

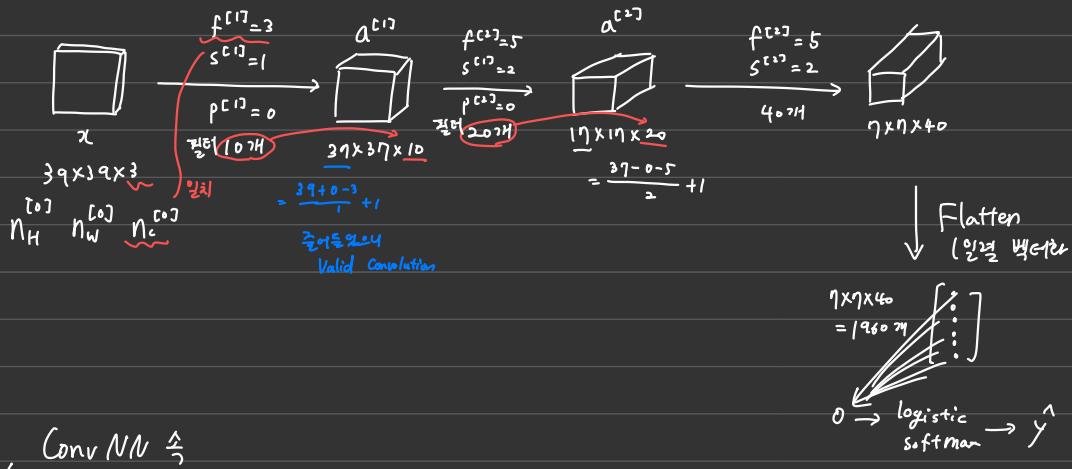
필터의

Weights:  $(f^{[L]} \times f^{[L]} \times n_c^{[L]}) \times n_c^{[L]}$   
= 필터의 수만큼!

bias:  $n_c^{[L]} = (1, 1, 1, n_c^{[L]})$

# Simple Convolutional Network Example

이전 장은 One layer. 이번에는 심층 Convolution NN에 대해서!



- Convolution (Conv)
- Pooling (POOL)
- Fully Connected (FC)

+ 오답 정리

입력:  $300 \times 300$  RGB 0/1/2  
필터:  $5 \times 5$  100개

hidden layer 속  
parameter 개수는?

↓

$$\therefore \text{weights} = 25 \times 3 = 75$$

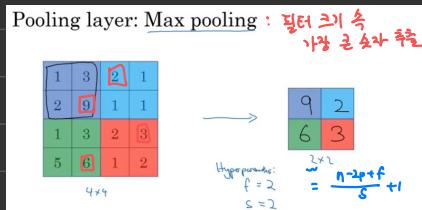
bias = 필터당 1개씩

$$\therefore (75+1) \times 100$$

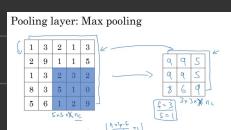
# Pooling Layers

크기를 줄여서 계산 속도를 높이고,  
조금 더 견고하게 감지하는 기능을 만들기 위해.

## < Max Pooling >



크기 2인 filter 2 stride 2인,  
Max Pooling한 결과



입력  
3차원이면

Max Pooling의 결과도  
3차원

\* Pooling은 Back prop 때 영향  
계집!

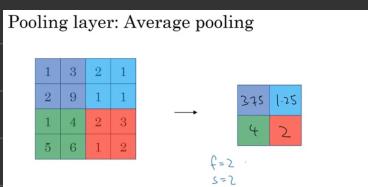
loss = | 흐름은 Back prop에서 일어남

Pooling layer는 입력 값을 선별/수정하기에

그 과정을 거친 것들을 대상으로

Back prop을 해야함  
(이유)

## < Average Pooling > : 각 필터 속 평균값을 이용



## < Hyperparameter of Pooling >

$f$ : filter size

$s$ : stride

효과: 2배 이상  
높이와 너비를 줄임

+ )  $p$ : padding  
드릴게 이용

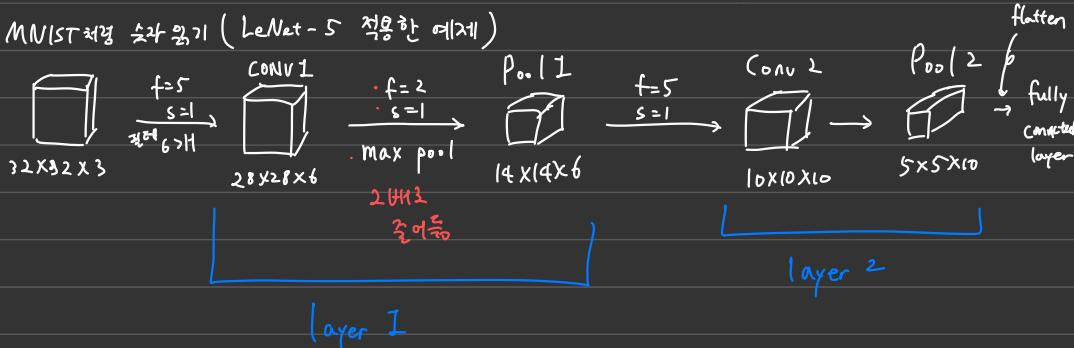
\*: Max Pooling은 No!!

보통 padding된 값이 0이 없음

# CNN 예시

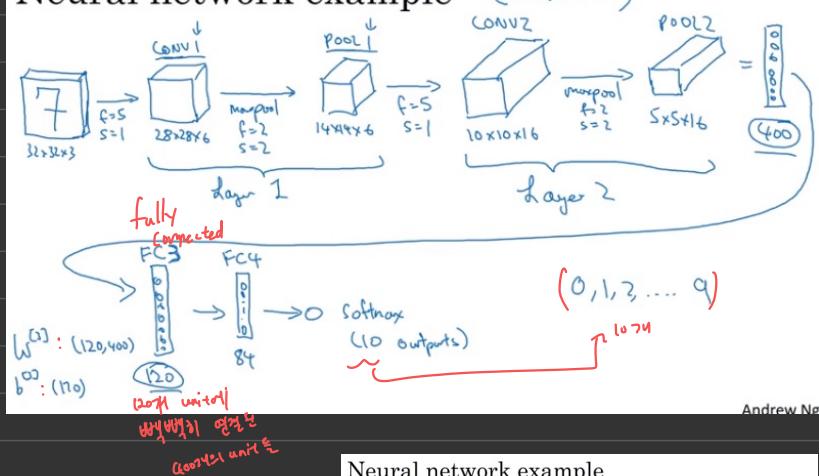
$n_H, n_W \downarrow$

- MNIST 처리 속도 끌기 (LeNet-5 적용한 예제)



One layer는 CONV + POOL이다!

## Neural network example (LeNet-5)



Andrew Ng

## Neural network example

|                  | Activation shape | Activation Size                                                | # parameters              |
|------------------|------------------|----------------------------------------------------------------|---------------------------|
| Input:           | (32, 32, 3)      | ~ 3,072                                                        | 0                         |
| CONV1 (f=5, s=1) | (28, 28, 6)      | 6,272 ( $\frac{28}{5} \times \frac{28}{5} \times 6$ ) = 6,272  | 0                         |
| POOL1            | (14, 14, 6)      | 1,568                                                          | 0                         |
| CONV2 (f=5, s=1) | (10, 10, 16)     | 1,600 ( $\frac{14}{5} \times \frac{14}{5} \times 16$ ) = 1,600 | 0                         |
| POOL2            | (5, 5, 16)       | 400                                                            | 0                         |
| FC3              | (120, 1)         | 120                                                            | $120 \times 400 = 48,000$ |
| FC4              | (84, 1)          | 84                                                             | $84 \times 120 = 10,080$  |
| Softmax          | (10, 1)          | 10                                                             | $10 \times 84 = 840$      |

conv layer  
48,000개의 parameter  
MAX pooling  
parameter가 줄어듬  
fully connected  
layer의 parameter는 줄어듬

# Why Convolution is Useful?

## 1. Parameter Sharing

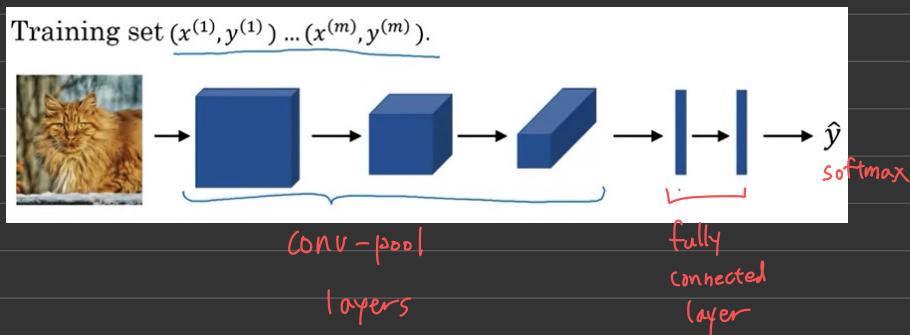
수직선 감지 같은 feature detector가 가장 유용한 부분을 추출해줌

## 2. Sparsity of connections

각 layer 속, 각각의 출력값은

입력의 아주 작은 일부분에만 영향 받음

## Putting it Together



이 속의 수 많은 parameter는,  $W, b$ 에 대비

$$\text{Cost } J = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

$W, b$ 를 찾기 위한 gradient descent를 하면 된다.

여기서 Adam 등을 적용해보!

# Case Study

<98>

## Classic Networks

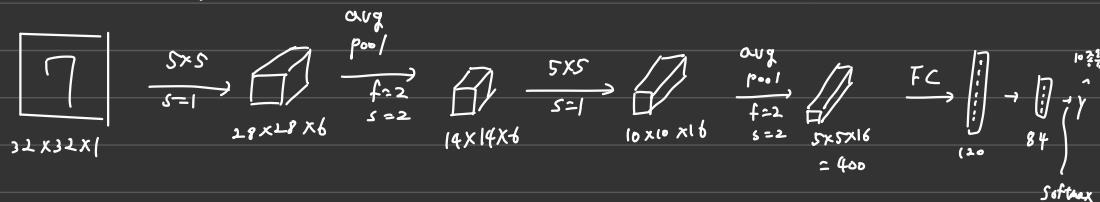
- LeNet-5
- AlexNet
- VGG

• ResNet

• Inception

# Classic Network

$\langle \text{LeNet-5} \rangle$  : parameter 60000개

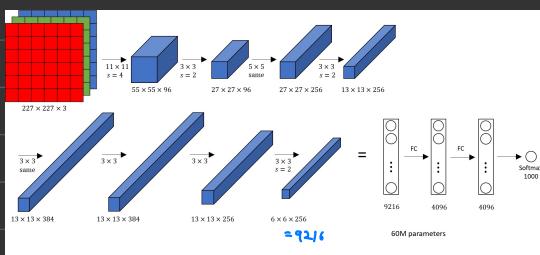


특징

①  $n_h, n_w \downarrow$      $n_c \uparrow$

②  $\boxed{\text{conv} \rightarrow \text{pool}} \times 4$   $\rightarrow$  FC X 여러번  $\rightarrow$  output

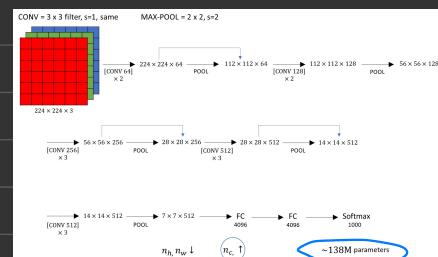
$\langle \text{AlexNet} \rangle$  : 6천만개 parameter



$\rightarrow$  ReLU 를 사용

$\rightarrow$  2개의 GPU를 사용하는 방법

$\langle \text{VGG-16} \rangle$  : 신경망을 단순화하는 예



## ResNet : Residual Network

### • Residual Block (잔차 풀기)

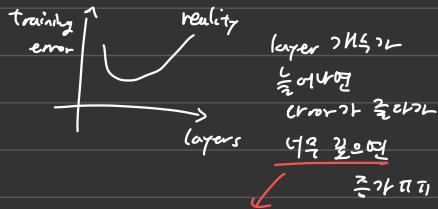
skip 하지 않고  
반복되는 block을 -



Skip connection ↓

$$\begin{aligned}
 a^{[l]} &\rightarrow \text{Linear} \rightarrow \text{ReLU} \rightarrow a^{[l+1]} \rightarrow \text{Linear} \rightarrow \text{ReLU} \rightarrow a^{[l+2]} \\
 z^{[l+1]} &= W^{[l+1]} \cdot a^{[l]} + b^{[l+1]} \quad a^{[l+1]} = g(z^{[l+1]}) \quad \dots \quad a^{[l+2]} = g(z^{[l+2]}) \\
 a^{[l+2]} &= g(z^{[l+2]} + a^{[l+1]}) \quad \text{Residual connection}
 \end{aligned}$$

잔차 유통을 많이 사용하여, 경쳐 쌓아나감



ResNet이  
더 깊은 것으로 가지 않  
gradient가 사라지거나  
혹발하는 것을 막아줌

# Why ResNet Works so Well?

always  $a > 0$

$$\text{원본: } X \rightarrow \boxed{\text{Big NN}} \rightarrow a^{[l]}$$

(다음 레이어 흡수로 ~)

$$\begin{aligned} \text{ResNet: } X &\rightarrow \boxed{\text{Big NN}} \rightarrow a^{[l]} \xrightarrow{\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}} \xrightarrow{\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}} a^{[l+2]} \\ &\quad \text{입/출력의 차이를 } V \text{ 를 합쳐서 차기 단계, } W_b \text{ 추가} \\ &\Rightarrow a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) \\ &= g(W^{[l+2]} a^{[l+2]} + b^{[l+2]} + a^{[l]}) \end{aligned}$$

그것이 좋은 차원이라고  
가정해야 Re-Net이 성립

$\therefore$  ResNet이 잘 되는 이유

: 추가적인 layer들

복잡한 copy 를 학습하는 거

이전보다 수워진 것 같아서

이 때 만약  $W^{[l+2]} = b^{[l+2]} = 0$  이면,

$a^{[l+2]} = g(a^{[l]})$  이 됨

$g$  가 ReLU라면

$\underbrace{a^{[l+2]}}_{a^{[l]}} = a^{[l]}$  이 됨

그거 copy하는 훌륭한 전략처럼

ResNet은

+

2개 차원 차이를 처리하기

모든 0으로  
해결할 수도.

$$\begin{aligned} a^{[l+2]} &= g(z^{[l+2]} + W_b \cdot a^{[l]}) \\ \xrightarrow[128]{256 \times} & \end{aligned}$$

# Networks in Networks and $1 \times 1$ Convolutions

$1 \times 1$  Convolution은 어떻게 동작하는가?

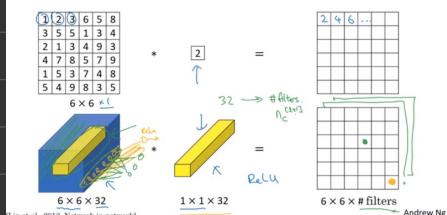
Why does a  $1 \times 1$  convolution do?

$$\begin{array}{|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 6 & 5 & 8 \\ \hline 3 & 5 & 5 & 1 & 3 & 4 \\ \hline 2 & 1 & 3 & 4 & 9 & 3 \\ \hline 4 & 7 & 8 & 5 & 7 & 9 \\ \hline 1 & 5 & 3 & 7 & 4 & 8 \\ \hline 5 & 4 & 9 & 8 & 3 & 5 \\ \hline \end{array} \quad * \quad \begin{array}{|c|} \hline 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|c|c|c|c|c|} \hline 2 & 4 & 6 & \dots & & \\ \hline \end{array}$$

$6 \times 6$

그저 스칼라 곱과 같음

Why does a  $1 \times 1$  convolution do?



특정 위치에서 element-wise 곱을 하여

ReLU를 하면 스칼라 값이 나온다

그런데 3차원일 때는 이야기가 달라짐

3차원의 수는 filter  $n_c^{[L+1]}$  과 동일하여

그에 따라 output은 filter 수만큼 생성

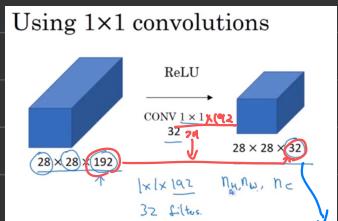
$\nearrow$   $(1 \times 1)$  conv :  $n_c \downarrow$   
 $\nearrow$  2D pool :  $n_h, n_w \downarrow$

$\langle (1 \times 1)$  Convolution  $\rangle$  | 유용한 상황> : non-linearity를 적용하여  
Volume를 증가/줄이/강도하고 싶을 때!

높이라 네비는 pooling과 stride 3

줄일 수 있음

채널 수를 줄이고 싶을 때 이걸 쓰라!

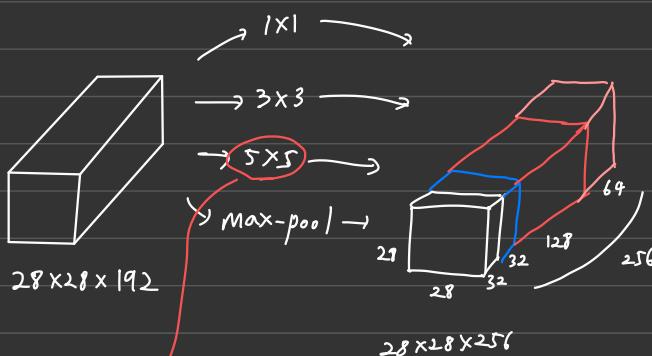


만약 Volume을 유지하고 싶다면

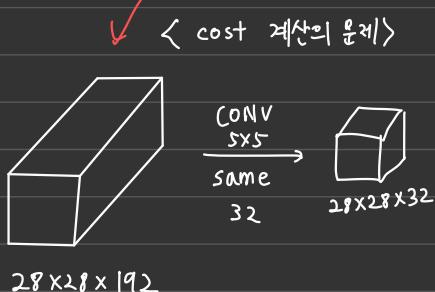
filter를 192개로 유지

그럼 ReLU 적용은 되어서 효과는 없음

# Inception Network Motivation



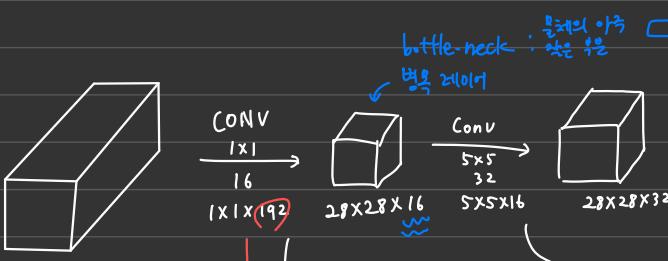
원하는 filter 크기와  
원하는 pooling을 선택하여 활용



output에 32개의 채널이 있고  
각 필터는 5x5x192 = 1억 2천만 켤레  
32개의 필터가 필요  
 $(28 \times 28 \times 32) \times (5 \times 5 \times 192) = 1억 2천만 켤레$

120m

< 1x1 convolution으로 계산량을 줄여보자 >



bottle-neck : 특징의 압축  
층 위로  
층 크기를 다시 늘리기 전까  
representation을 살피 만들

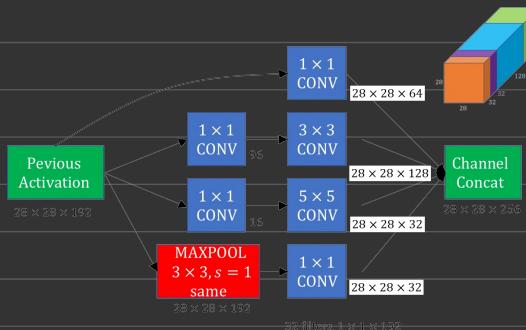
∴ cost 계산 비율

$$(28 \times 28 \times 16) \times 192 = 2.4 \text{ million} + (28 \times 28 \times 32 \times 5 \times 5 \times 192) = 12.4m \\ = 10.0 \text{ million}$$

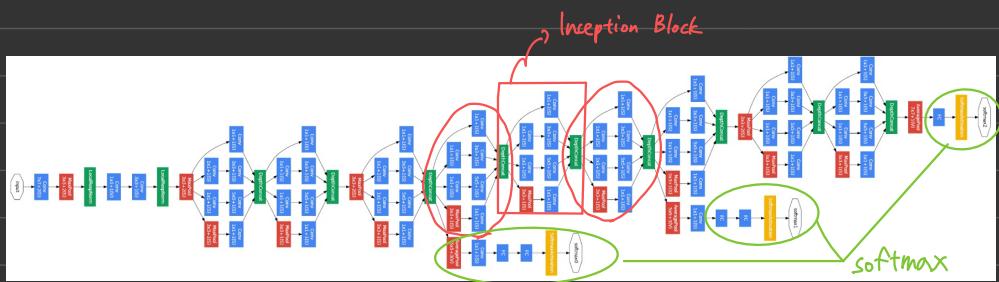
75%  
25%

# Inception Network

< Inception module >



< Inception Network > : 위의 모듈을 여러 번 반복한 것



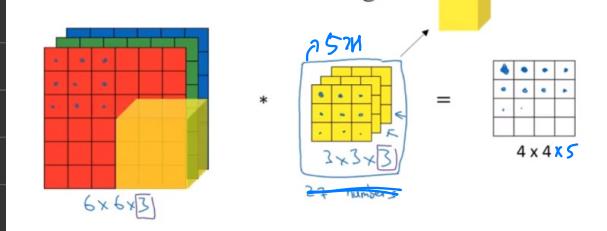
# Mobile Net

for NN for mobile phone

less powerful CPU & GPU

<Normal Conv>

Convolutions on RGB image



$$n \times n \times n_c \quad f \times f \times n_c \quad n_{out} \times n_{out} \times n_c'$$

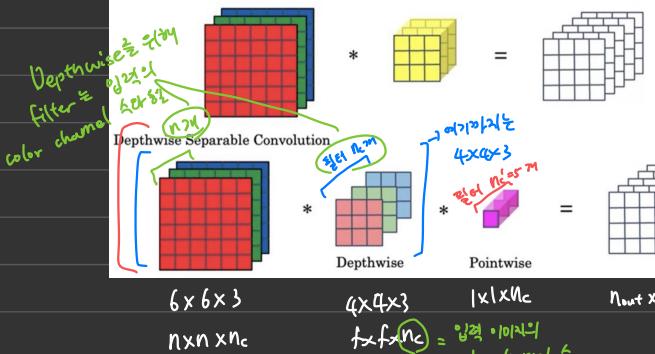
Computational cost

$$2160 = 3 \times 3 \times 3 \times 4 \times 4 \times 5$$

Depthwise Separable Convolution = 2번의 Conv를 수행

Normal Convolution



Depth wise :  $432 = 3 \times 3 \times 4 \times 4 \times 3$

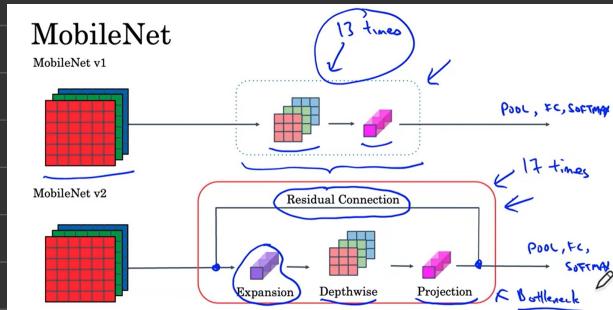
$$(4 \times (4 \times 3) + 1) \times 1 \times n_c' \text{ 를 적용}$$

$$\therefore \text{cost} = \underbrace{240}_{\substack{\text{filter param} \\ \sim}} \times \underbrace{1 \times 1 \times 3}_{\substack{\text{filter position} \\ \sim}} \times \underbrace{4 \times 4}_{\substack{\text{filter} \\ \sim}} \times 5$$

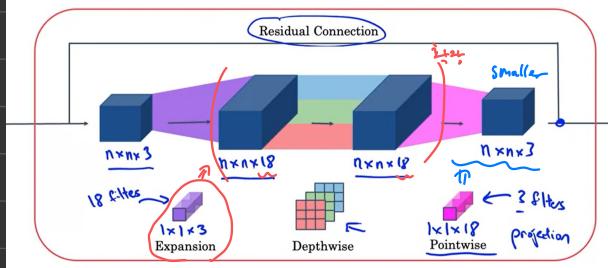
2160에서 432+240 = 672 만큼 감소

# MobileNet Architecture

조금 더 발전된 MobileNet v2 구조



## MobileNet v2 Bottleneck



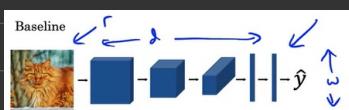
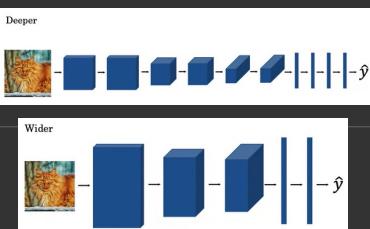
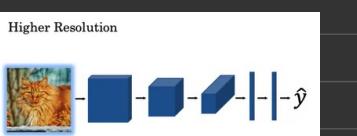
# Efficient Net

서로 다른 종류의 통과 차원에 맞추어 차등적으로 크기를 늘리고 줄임

이 중에서 선택적 scale up/down

$$\begin{cases} r : \text{resolution} \\ d : \text{depth} \\ w : \text{wider} \end{cases}$$

각 비율은  
기본마다 다른  
관련 논문을 봐보니



# Using Open-Source Implementation by Github

⇒ Github에서 원성분을 다운받아 실행하면서 하세요.  
설명서와 함께 풀어서 읽으려고 하면 힘들어요.

(코드 복잡하거나  
가능치도 다운받자)

## Transfer Learning

이미 만들어진 유명한 dataset

: ImageNet, MS Coco, pascal 등

보내기 대상이 되는

만들어진 것 가져와서

하는 것이 Transfer Learning

데이터 전처리 ↓

마지막 layer까지

softmax 안 바라보는 것으로

시작하면 시간 단축

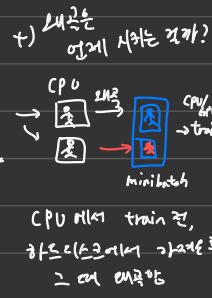
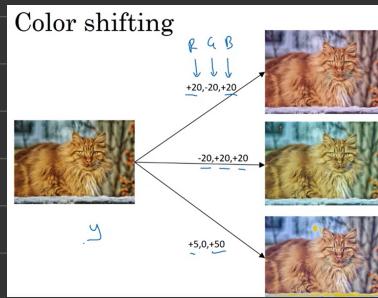
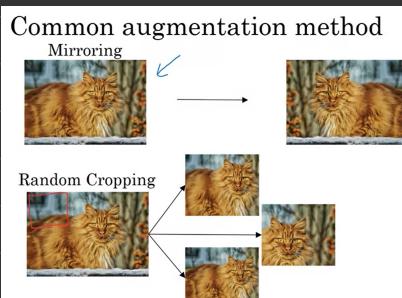
데이터가 많다면

여 않은 layer는 경계 바라보기

기억하고 gradient descent는 하지마

• last layer는 바라볼 수도 있음

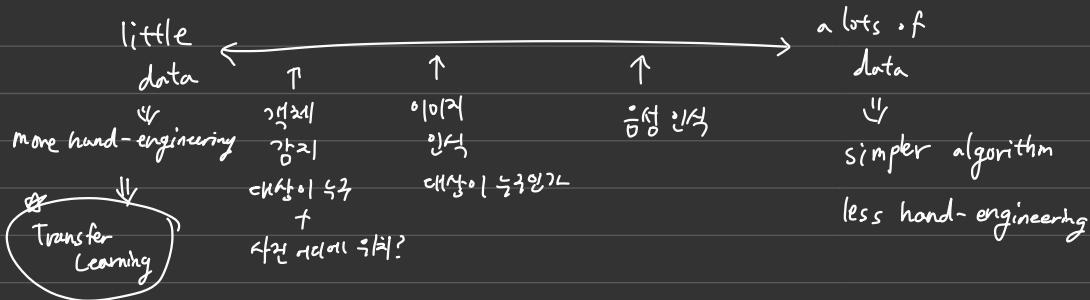
## Data Augmentation



### RGB 샘플링 방법

- PCA (Principle Component Analysis) 48
- 이미지가 주로 빛장/탁장 차이로 인해 그 색상을 줄여서 RGB가 필요로 되도록 함

## Computer Vision 은 현재



- 정보의 출처
    - label이 지정된 data ( $x, y$ )
    - Hand Engineering - feature와 규칙을 설계하여 학습

# Tips for winning Competitions

→ Ensemble

열 개의 NN을 독립적으로 훈련하여  
(3~15개) 그 결과들의 평균값 구하기

7 ↳ 기업에서 사용X

→ Multi-crop at test time  $\Rightarrow$  NNOI 시점은 절대로 되지 않지만 시간이 흐르면 절여

↳ data augmentation = test set all  $\frac{2}{3}$ .

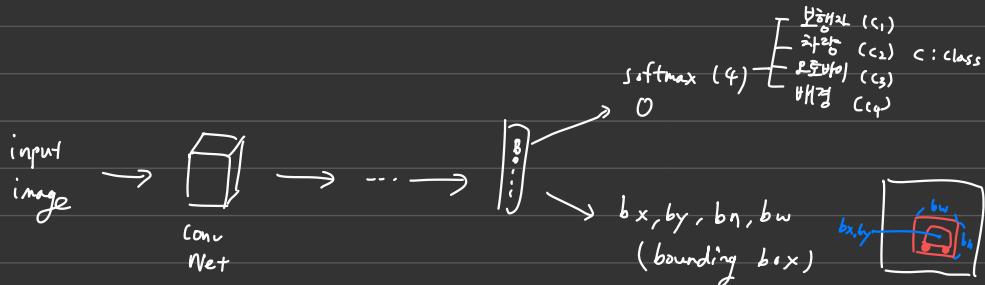
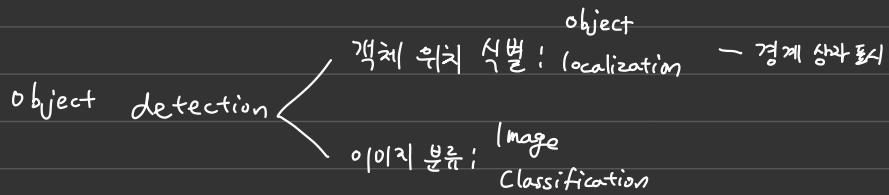
### 각각의 허전의 결과들의 평균값 구하기

↳ 10 - Crop

for corners crops with different color filter



# Detection Algorithms



$y \in$  가능한 한다면

$$y = \begin{bmatrix} \text{logistic regression} \\ P_C \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix}$$

나쁜 품질  
있어야!  
있는가!

mean-squared error



$$\begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

↳ don't care

loss 함수 정의하기

$$\begin{aligned} L(\hat{y}, y) &= (\hat{y}_1 - y_1)^2 + \dots + (\hat{y}_8 - y_8)^2 \quad \dots y_1 = 1 \\ &= (\hat{y}_1 - y_1)^2 \quad \dots y_1 = 0 \end{aligned}$$

~~training set~~  
bounding box 를요!

예측한  $b_x$  가 실제  $b_x$  가 아니면  
정답인가? 아니면?

## Landmark Detection

= 이미지의 주요 지점을 X.Y 좌표로 신경망이 출력하게 만드는 것



양쪽 눈의 좌표를 알고 싶다면?

→ 자세 감지하면,

$l_{1x}, l_{1y}$

$l_{2x}, l_{2y}$

$l_{3x}, l_{3y}$

$l_{4x}, l_{4y}$

:

정 개수  $N+1$ 번

$2N$  만큼의 unit이

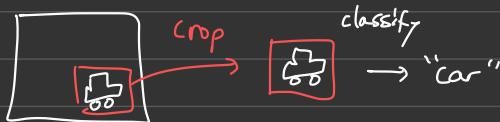
output에 있음

각 관절마다 좌표생성

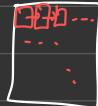
AR에 사용됨

원하는 만큼 좌표 생성

## Object Detection



crop  $\Rightarrow$  Sliding window 방법이 %



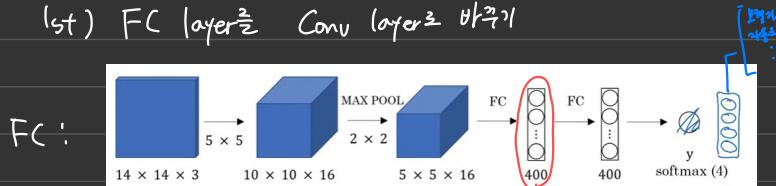
window 크기를 점점 크게 하면서  $\Rightarrow$  cost 비용  
넓 많아지면

↳ 특히 이미지가  
넓은 경우 때

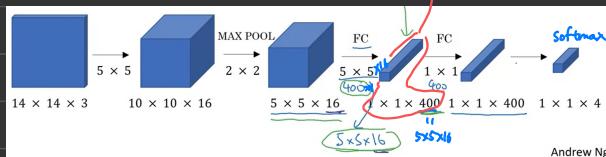
# Sliding Window $\frac{2}{2}$

Convolutional 하지 실행해보자

1st) FC layer  $\frac{2}{2}$  Conv layer  $\frac{2}{2}$  를  $\frac{2}{2}$ 가



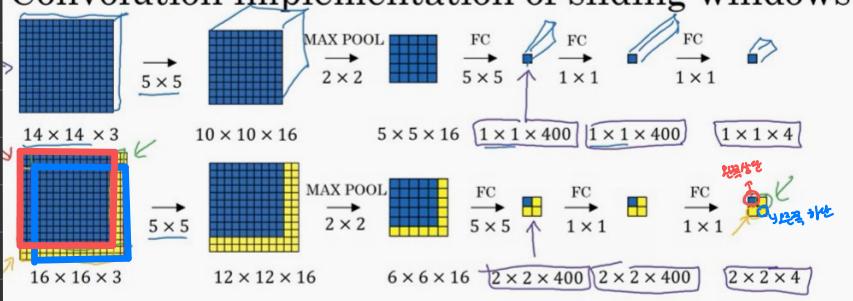
Conv :



2nd)

Convolution implementation of sliding windows

train set :



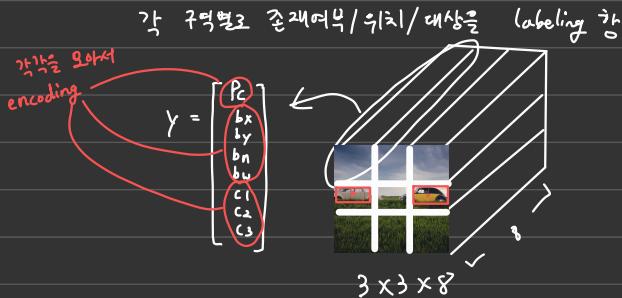
test set :

$\downarrow$   
+2 pixel

이전 씩스로  
한 번에/  
여러 구역을 계산

# Bounding Box Predictions 개선하기

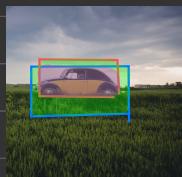
## 1. YOLO Algorithm



## 2. YOLO 조정 방법 - IoU : Intersection Over Union

IoU가 하는 일

겹쳐진 Bounding Box의 영역을 계산



교차영의 크기를 계산

$$IoU = \frac{\text{교차 영역}}{\text{전체 영역}} \geq 0.5 \text{ 일 때}$$

옳다고 판단

0.7, 0.6을 넘지 않도록  
기준으로 해도

# Non-max Supression <sup>여제</sup> Ex

: 알고리즘이 하나의 객체를 여러 번 감지했는지, 한 번만 감지했는지 알아보는 것

→ (st) 탐지와 관련된 확률을 조사

2nd) 그 중 가장 큰 것을 뺏기 처리

3rd) 나머지 boxes를 어동지 처리하여 선택되지 않음을 표시



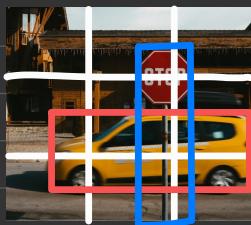
+ ) 이 때 0.6 이하는 무조건 삭제하는 방법을 추가할 수도 있음  
" " ]

$$\begin{array}{ccccccccc} & & -2 & & -1 & & +2 & & -3 \\ & & -4 & & -4 & & +4 & & +4 \\ & & 1 & & 1 & & = 2 & & \end{array}$$

# Anchor Box

여러 개의 cell에 Overlapping 된 객체 감지

두 객체가  
동일한 Grid에  
나타났을 때 하는 일



1st) Anchor Box 모양을 미리 정하기



$$2nd) \quad y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \\ p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

각각을 모아서  
encoding

for Anchor Box 1

for Anchor Box 2

3rd) IoU 값을 가진 Anchor Box를 선택

(grid cell, anchor box) 쌍의 값을 가짐.

만약 Anchor Boxes는 2개인 경우

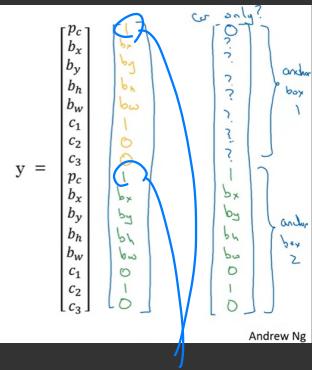
객체가 3개라면

인식을 잘 못하게 됨. 즉ly.

Anchor box example

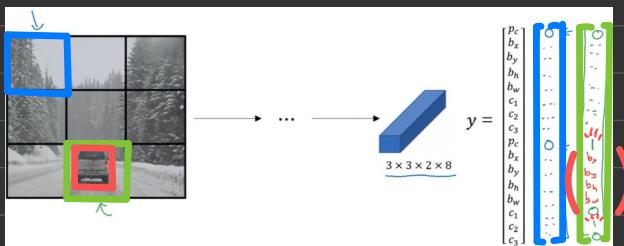
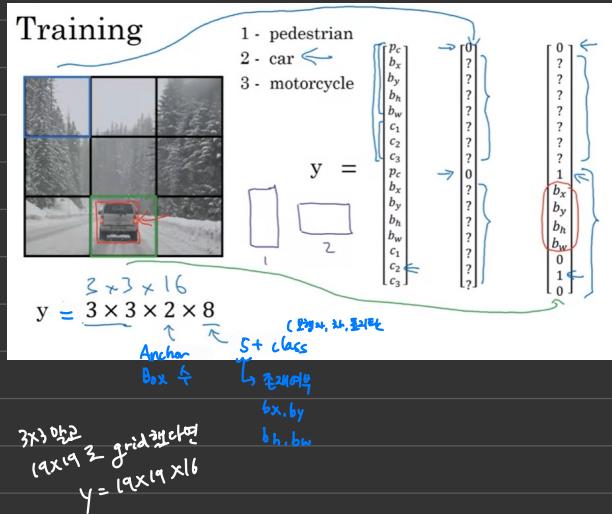


Anchor box 1:      Anchor box 2:



각각의 cell의  
두 객체 모두 있는지를 알 수 있음

# Yolo Algorithm



## Non-Max Suppressed 적용

각 cell마다 2개의 Bounding box를 얻음

IoU 낮은 상자는 제거

각 집단 별로 non-max suppressed를 적용하여 최종 예측하기

= 보행자, 차량, 표지판



## + ) Region Algorithm

전체 이미지를 sliding window하지 않고

Segmentation Algorithm으로 배경이 되는 구역은 skip

특정 부분에 대해서 짐작적으로 sliding window를 실행  
(총격전 아님)

Region proposal: R-CNN



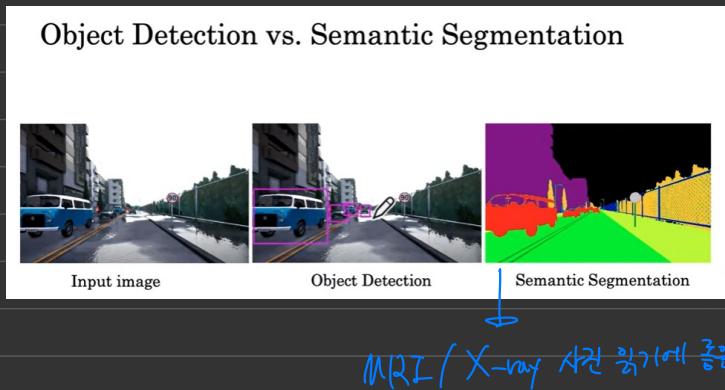
더욱 정확한

=> Bounding Box를 얻을 수  
있음

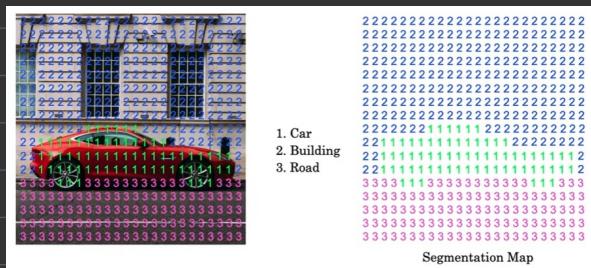
특정 지역을 제안하는 속도를 높이기 위해

이 제안에 Convolutional Network를 사용

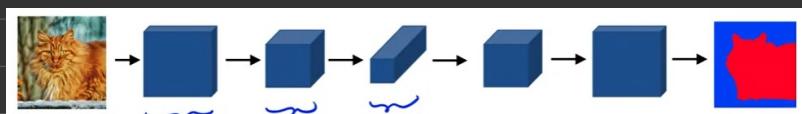
# Semantic Segmentation with U-Net



Per-pixel class labels



Deep Learning for Semantic Segmentation



Smaller

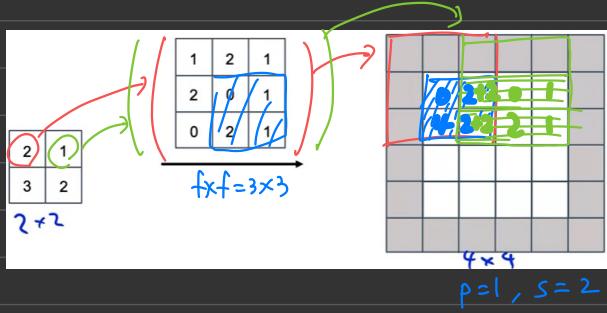
Back to Original Size By Transpose Conv

# Transpose Convolutions

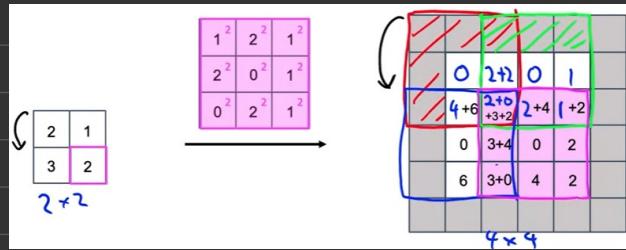
Normal Conv

$\square \times \text{tiny} = \text{small}$

smaller  $\times$  small  $\rightarrow$  bigger

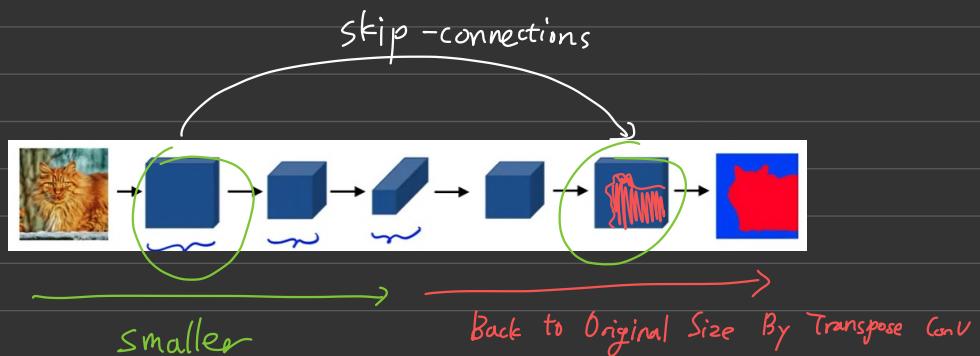


경치진 딕셀을  
느낄까요? 좋지



# U-Net Architecture Intuition

고해상도 이미지에서  
unit들은 어떤 일을 하나?



누락된 것은 정교하고 세밀한 공간 정보 뿐.

resolution

이는 activation layers가 더 높은 공간 해상도를 가지고 있어야  
되거나 놓일 수 있는 정보임.

이제 skip connection이 하는 일은

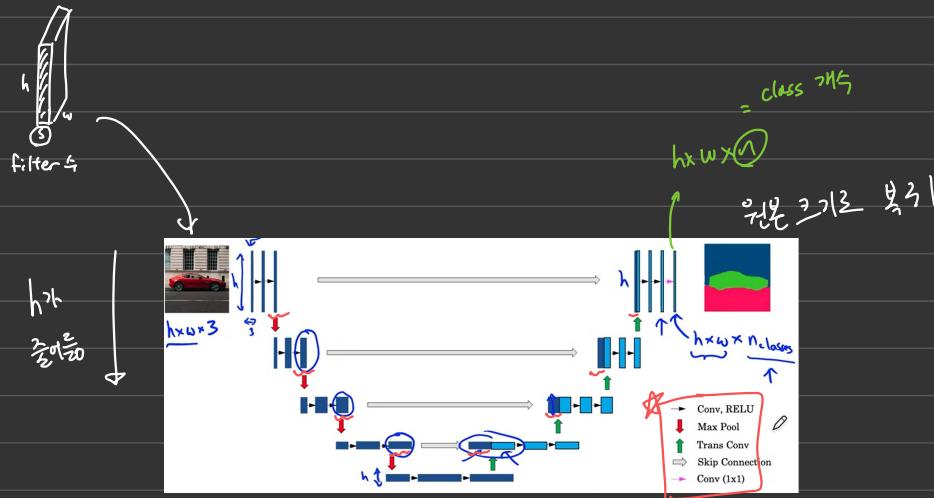
NN이 모든 픽셀을 강제할 정도의 높은 해상도와

지수준의 feature 정보를 '알아챌 수 있도록' 하는 것

또한 골바로 그 다음 layer로 진행되는 것을 잠시 멈춤

# U-Net Architecture

설명 2방법 U-Net 구조



Skip Connection은  
마지막 버전을 복원함

# Face Recognition vs Face Verification

K명에 대한 DB, 이미지 풀

K명에 속한 사람이라면 ID 출력

I : K

입력: 이미지, 이름/ID

출력: 2 이름에 2 이미지가 맞는지 확인

1 : 1

## One Shot Learning

이미지 한 장으로도 학습시키기...

<Learning "Similarity" func>

$$d(\text{img}_1, \text{img}_2) = \text{이미지 간 차이 정도}$$

$$\begin{cases} \leq c \rightarrow \text{"same person"} \\ > c \rightarrow \text{"different"} \end{cases} \quad \text{Verification}$$



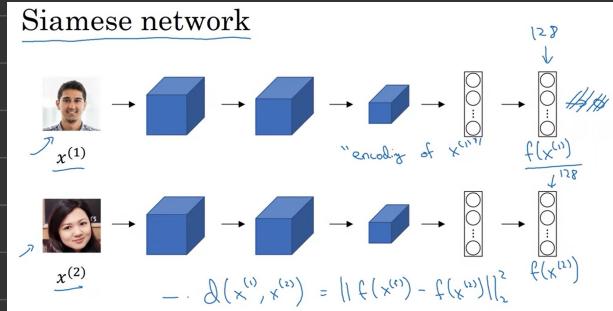
$f(x^{(i)})$  가 사람수 만큼 존재

$$NN \in \frac{\|f(x^{(i)}) - f(x^{(w)})\|^2}{2}$$

이해 학습해나감

같은 사람이라면 거리가 더 길어지지.

다른 사람은 사람이라면 더 가깝게



# Triplet loss

차이가 있을 때 틀릴 때 놀다 0인 상황을

비교하기 위해

margin  $\alpha$  를 더함  $\rightarrow$  원하는  
parameter  $\rightarrow$  더해서 정확도 ↑

## Learning Objective



$$\text{Want: } \frac{\|f(A) - f(P)\|^2}{d(A, P)} + \alpha \leq \frac{\|f(A) - f(N)\|^2}{d(A, N)}$$

$$\frac{\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha}{d(A, N)} \leq 0 \quad \text{margin} \quad f(\cdot) = \vec{o}$$

Schroff et al., 2015, FaceNet: A unified embedding for face recognition and clustering]

Andrew Ng

## Loss function

$A, P, N$  3개가 되어야만  
positive negative

Choosing triplets  
 $A, P, N$

훈련 동안 만약  $A, P, N$ 이 목록으로 등장하면  
 $d(A, P) + \alpha \leq d(A, N)$  을 만족함  
성능을 위해 되도록 훈련하기 힘든 3쌍으로 구성하기를?

$$L(A, P, N) = \max \left( \|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0 \right)$$

$$J = \sum_{i=1}^m L(A^{(i)}, P^{(i)}, N^{(i)})$$

생각해보면

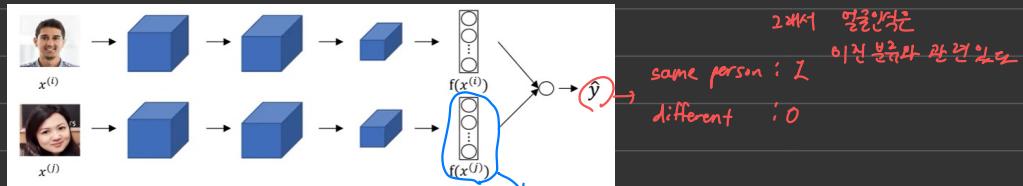
$N \leftrightarrow A$  는

서로 바꿔서 넣으면  
또 다른 세트로 data set  
되기도!

training set : 10K 장의 이미지

다른 사람이  
한국 오직 블로그 쓰라 ~ ^~

# Face Verifications and Binary Classification



"Logistic Regression"으로 이전 분류를 풀어

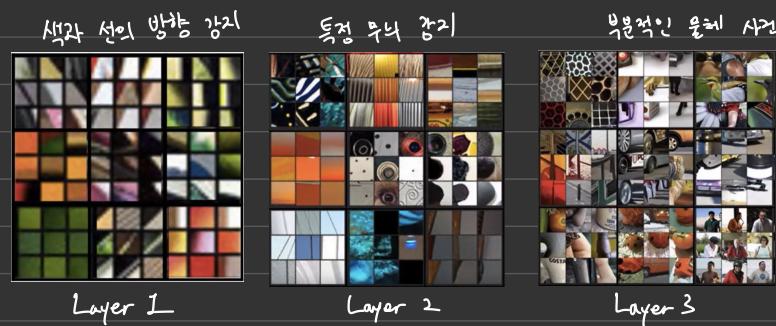
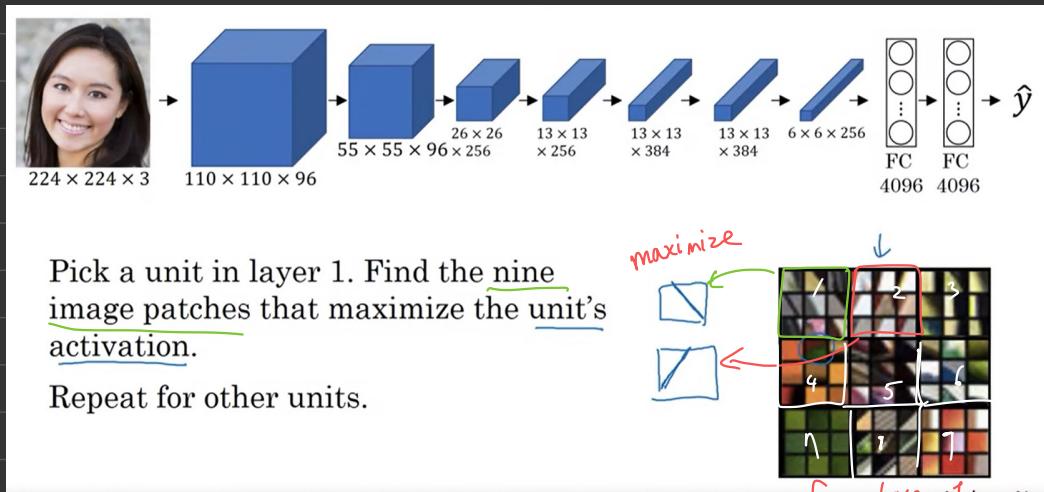
각 이미지에 따른  $f(x)$ 을  
매번 계산하는 것이 아니라  
미리 계산해두고 DB에 저장해놓기 때문에 필요 없음

"pre compute"

$$\hat{y} = \sigma\left(\sum_{k=1}^{128} w_k (f(x^{(i)})_k - f(x^{(j)})_k) + b\right)$$

$$x^2 = \frac{(f(x^{(i)})_k - f(x^{(j)})_k)^2}{f(x^{(i)})_k + f(x^{(j)})_k}$$

# What are Deep ConvNets Learning?



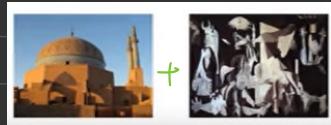
# Cost func for Neural Style Transfer



$$J(G) = \alpha \cdot J_{\text{content}}(C, G) + \beta \cdot J_{\text{style}}(S, G)$$

자기 221 ↘

< Find the generated img  $G$  >



1st)  $G$ 를 목적函으로 초기화

$$G : 100 \times 100 \times \underbrace{3}_{RGB}$$



2nd) Gradient descent 3  $J(G)$ 을 최소화하기

$$\underbrace{G}_G = G - \frac{\varepsilon}{2} \nabla J(G)$$

$\hookrightarrow$  특징 간지 간 상관관계(영향)

예술적으로 보이는 학설에 대해 훈련할 뿐,  
parameter를 훈련하지 않음!



① Content Cost function :  $J_{content}(C, G)$

가정 T이이 훈련된 ConvNet 사용 (ex - VGG)

$l$ : hidden layer

L 2 차원에서 l번쨰 2차원의 activation

$$= \alpha^{[\ell](c)}, \alpha^{[\ell](G)}$$

만약 이들이 서로 비슷하다면 두 이미지는 같은 Content임

$$\therefore \text{J content}(C, G) = \sum_i ||\alpha^{[e] \in C} - \alpha^{[e] \in G}||^2$$

② Style Cost func :  $J_{style}(S, G)$

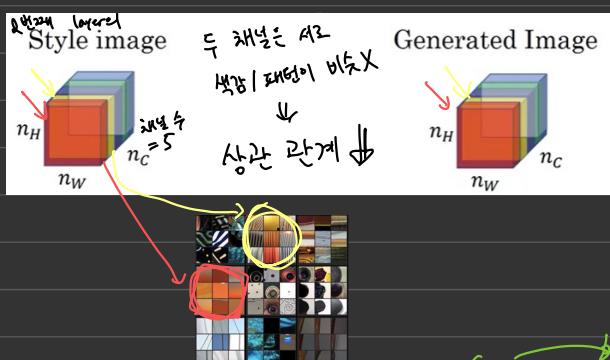
## < 0(0)2[0]서 "style" 의 의미 >

$\equiv$  채널들을 가로지르는 activations 간의 상관관계

1월22일 2010년 1/21

∴ 높은 수준의 이미지의 질감 표현이

이미지의 부모들에서 함께 발생하는,



## <Style Matrix>

$a_{i,j,k}^{[l]}$  = (i, j, k)의 activation

$$G^{[\ell]} = n_c^{[\ell]} \times n_c^{[\ell]}$$

$$G^{(L)(S)} = \sum_i \sum_j a_{ijk} \cdot a_{ijl}^{(L)(S)} \quad | \sim \text{체밀수}$$

Gram matrix

3

$$J_{style}(S, G) = \sum_k \lambda^{[k]} \cdot J_{style}^{[k]}(S, G)$$

↓ hyperparameters

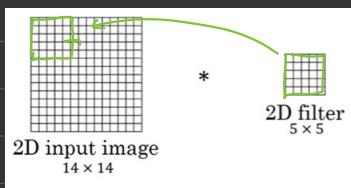
NN에서 다른 layer를 사용하게 함.  
즉, 초기 레이어에는 간단한 패턴을  
추가로 복잡한 더 강자하도록.

$$\therefore J_{style}^{[L]}(S, G) = r'' \left\| G_{[L]^{(S)}} - G_{[L]^{(G)}} \right\|_F^2 \\ = \frac{1}{\left( \sum_{k=1}^m \sum_{k'=1}^m \sum_{c=1}^C \right)^2} \sum_{k=1}^m \sum_{k'=1}^m \left( G_{kk'}^{[L]^{(S)}} - G_{kk'}^{[L]^{(G)}} \right)^2$$

# 1D and 3D Generalizations

$$\text{공식: } \lfloor \frac{n^{[L-1]} - f + 2 \cdot p}{s} \rfloor + 1$$

## 2D Conv

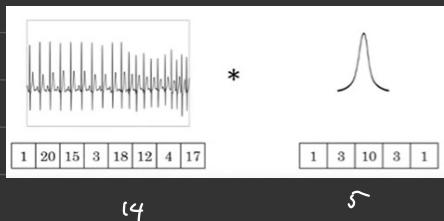


RGB  
 $14 \times 14 \times 3 * 5 \times 5 \times 3$

$\rightarrow 10 \times 10 \times 16$  : 필터가 16개라면  
 $(14 - (5-1))$

$10 \times 10 \times 16 * 5 \times 5 \times 16$   
 $\rightarrow 6 \times 6 \times 32$  : 필터가 32개라면

1D conv ex) 심장 박동

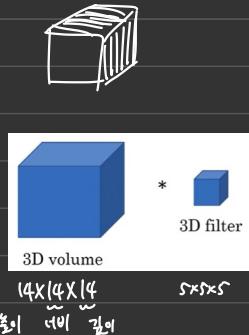


$14 \times 1 * 5 \times 1$

$\rightarrow 10 \times 16$  : 필터가 16개라면

$10 \times 16 * 5 \times 16$   
 $\rightarrow 6 \times 32$  : 필터가 32개라면

3D : 2D의 예외 slice  $\frac{1}{2}$  경계값은 놓



$14 \times 14 \times 14 \times 1 * 5 \times 5 \times 5 \times 1$

RGB 등등  $\neq$  필터 등  
 차례로 수는 서로 같아야 함  
 $n_c$

$\rightarrow 10 \times 10 \times 10 \times 16 * 5 \times 5 \times 5 \times 16$  : 16개의 필터가 있다면

$\rightarrow 6 \times 6 \times 6 \times 32$  : 32개의 필터가 있는 경우

# Sequence Model

지금까지 본 모델은 스스로 만드는 방법을 배운다

Ex. of Sequence data : 오두 (X, Y)로 나타내어

Supervised Learning을 할 수 있다

· 음성 인식 : 시간을  
음파  $\rightarrow$  단어 나열 풀방

· 음악 생성 :  $\emptyset \rightarrow$  음표와 나열

· 감정 분류 : "이 영화는 뿐만 아니라"  $\Rightarrow$  ☆☆☆☆☆

· DNA : AGGCC--  $\Rightarrow$  AG<sup>|||||</sup>CC<sup>T</sup> : 어느 부분이 단백질과  
일치하는지 와콤보 볼여

· 가게 번역 프로그램  $\rightarrow$  영어

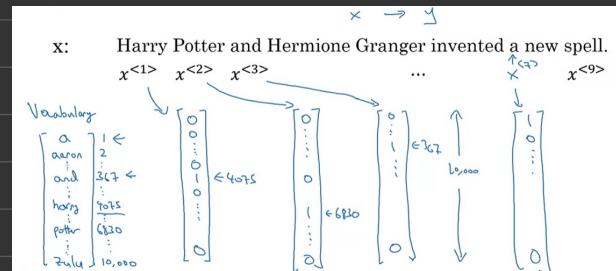
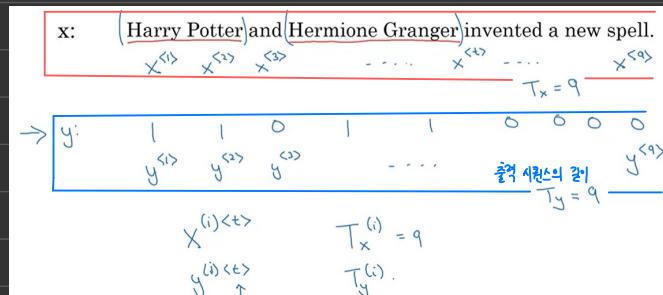
# Notation for Sequence Model

ex) 문장에서 사람 이름 찾는 시퀀스 모델 - Named Entity Recognition

$$x \rightarrow y$$

한 문장  
각 단어의 유무

1 0 0 1 0 0 ...  
0 0 1 0 0 0 ...



i 번째 train data  
t 번째 단어

각 단어가 Vocabulary 21스톤 안의 뭘 번째 있는지 찾아서

그것만 (1) 하는 one-hot coding을 적용

그렇게  $x \rightarrow y$  데코더로하여 NLP 풀리기

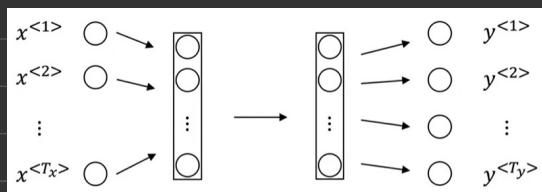
순환 신경망

# RNN : Recurrent Neural Network Model

Standard

NN 모델 만들기

$I \leftarrow X \rightarrow \text{수집망}$

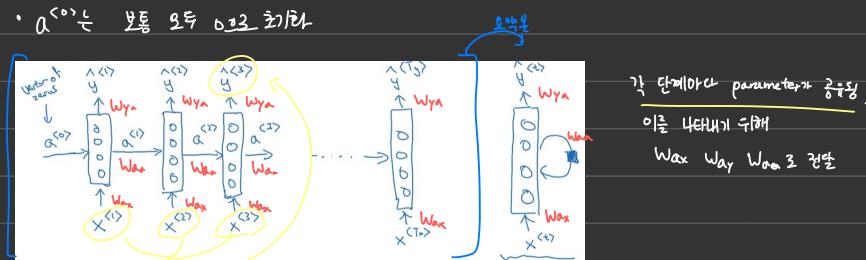


<문제점>

- 1. 입력과 출력의 단어/문장 길이가 불일치
- 2. 여러 사람의 이름을 동시에 감지하기 어렵다  
(한 사람만 가능)

## RNN

- 두 번째로 유치한 단어를 예상할 때 이전의 예상값을 바탕으로 전개
- $T_x, T_y$  길이 동일함
- $a^{<0>}$ 는 보통 모두 0으로 초기화



덕분에 이전 모든  $X$ 가 현재  $\hat{y}$ 에 적용됨

$\Rightarrow$  RNN 단점은 과거에 대해서만 봄다는うこと

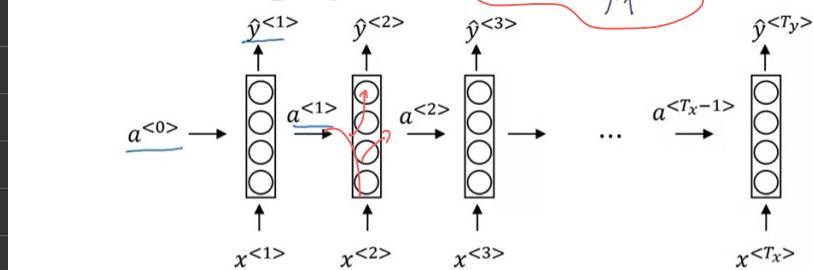
ex) Teddy is a Teacher 어색

Teddy가 선생님이라는 것은 뒷 단어를 봤어야 알 수 있었음

$\Rightarrow$  보완: Bidirectional NN (BiNN)

# RNN - Forward Prop

## Forward Propagation



$$a^{<t>} = g_1(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a) \leftarrow \text{tanh / ReLU}$$

$$\hat{y}^{<t>} = g_2(W_{ya} a^{<t>} + b_y) \leftarrow \text{sigmoid}$$

(0 or 1 : prevent Vanishing gradient)

$$\begin{cases} a^{<t>} = g(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a) \\ \hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y) \end{cases}$$

단승화해보면,

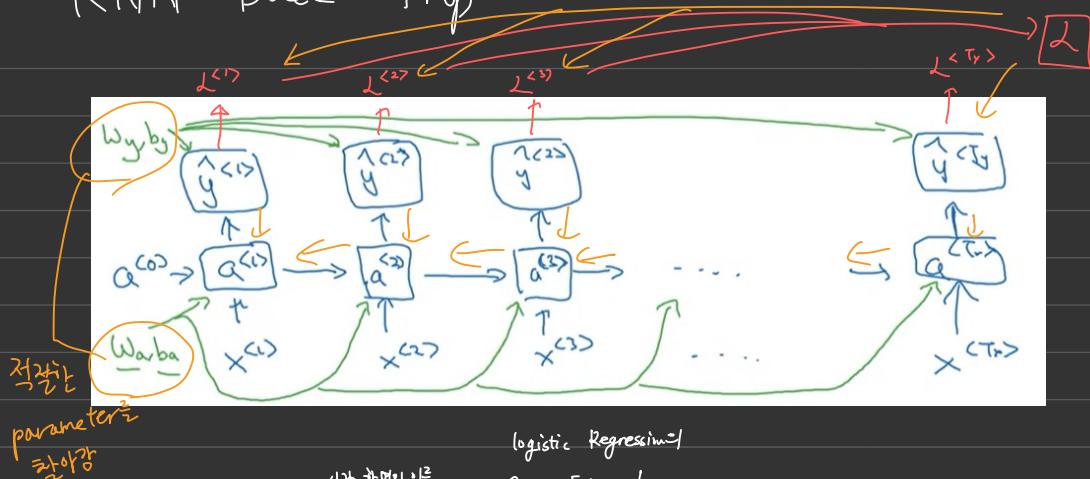
$$a^{<t>} = g(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y)$$

$$\begin{aligned} [a^{<t-1>} \ x^{<t>}] &= \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} \in \mathbb{R}^{100 \times 100} \\ a^{<t>} &= g(W_{aa} [a^{<t-1>} \ x^{<t>}] + b_a) \\ &= [W_{aa} \ W_{ax}] \begin{bmatrix} a^{<t-1>} \\ x^{<t>} \end{bmatrix} + b_a \\ &= W_{ya} a^{<t>} + b_y \end{aligned}$$

시그모이드 활성화기 적용 가능

# RNN - Back Prop



$$L^{(t)}(\hat{y}^{(t)}, y^{(t)}) = -y^{(t)} \log \hat{y}^{(t)} - (1-y^{(t)}) \log (1-\hat{y}^{(t)})$$

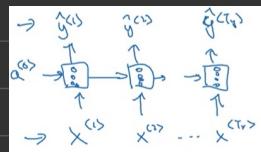
$$\Downarrow$$

$$L(\hat{y}, y) = \sum_{t=1}^T L^{(t)}(\hat{y}^{(t)}, y^{(t)})$$

# RNN의 예시 버전

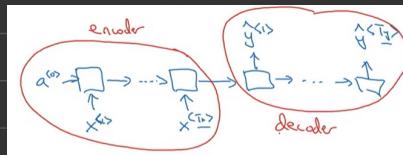
## ① Many - to - Many :

입력 여러 개, 출력 여러 개



→ 입출력 길이가 다른 경우

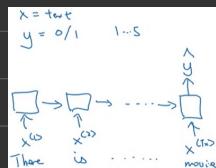
: 기계 번역



번역 문 → 번역 문

## ② Many - to - one :

간접보석으로 별장 탐방하기

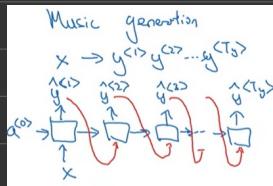


## ③ one - to - one : 원래는



## ④ one - to - Many : 음악 생성

(입력이 null 일 수도,  
zeros vector 일 수도 있다)



RNN의 출력 확률값을 이용하여

$\hat{y}^{(t)}$  word를 무작위로 sampling 함

이렇게 선택된 단어는 다음 단계로 넘어감

# Language Model and Sequence Generation

< Language Model이란? >

음성 인식에서

pair of pair의 구조는 의미로 해석하는 힘

문장에 따른 그 단어 확률을 계산하는 language Model이 있음

$$= P(y^{<1>} , y^{<2>} , \dots , y^{<T>})$$

문장 속 각 단어

## Language Modeling with an RNN

training set : 거대한 영어 말뭉치 (문장을)

↓

1st) Tokenize ... (.) 문장도 단어로 치환할 수도 있음

↓

2nd) Mapping One-hot - 각 단어 벡터로

$\langle \text{EOS} \rangle$ 로 문장 구별  
end of sentence

↓

unknown

3rd) Vocabulary에 없는 단어 처리  $\rightarrow \langle \text{UNK} \rangle$  토큰으로 처리

## RNN model

해당 위치에 어떤 단어가 나타날지 예측

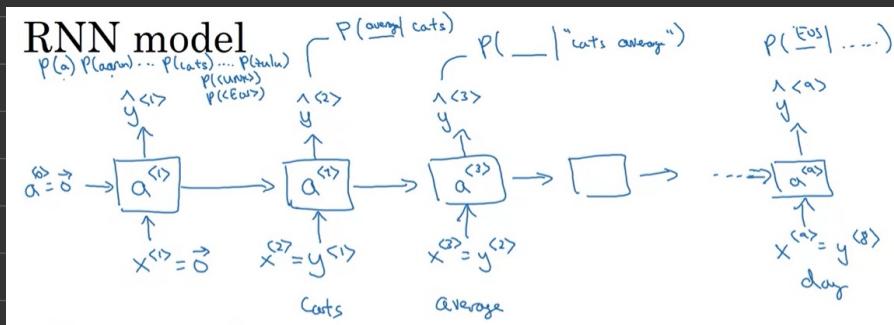
↳ 이전의 패러다임을 이용하여 ↑ 계산

$\frac{\text{Cat} \text{ is cute 라면}}{P(\text{Cat})}$   $T \vdash y \in \text{Cheese}$  는 첫번째에 Cat이란 단어가 놓을 때  $\equiv$   $P(\text{is}/\text{Cat})$   
두번째로 올 단어는?

$$\therefore P(y^{(1)}, y^{(2)}, y^{(3)})$$

$$= P(y^{<1>}) \times P(y^{<2>} | y^{<1>}) \times P(y^{<3>} | y^{<1>}, y^{<2>})$$

$$\begin{aligned} \hat{y} &: \text{Softmax prediction} & \text{Softmax loss func} \\ y &: \text{true word} & \rightarrow \mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>} \\ & & \mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) \end{aligned}$$



## ❖ 가중치가 아님

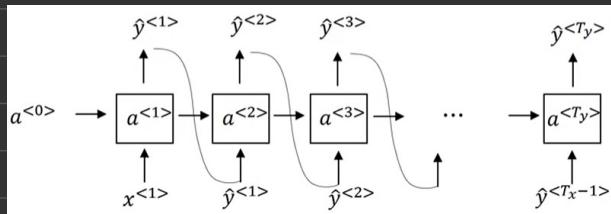
4 자체를 넘김.

✓ ⌈ 하나만 놓여도 나머지는 알아서 쭉쭉 만들기 ⌋

$\vec{0}$ : zero vector

$a$ : Softmax

# Sampling Novel Sequences



경우 진행하다가 <EOS>면 Stop

<UNK>가 포함되면 안되는 경우

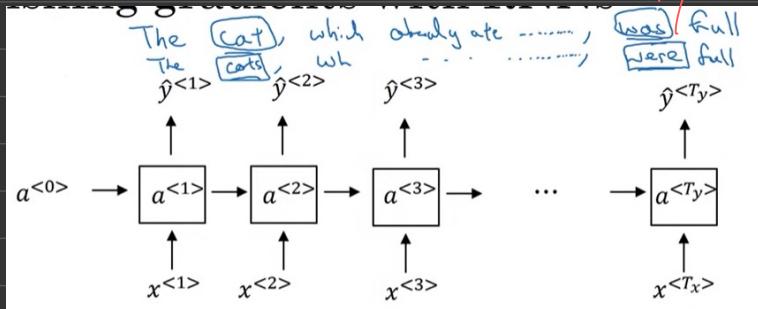
resampling하여 <UNK> 없는  
문장 만 뽑도록 함

단어 예측도 넘어 char 단위 예측도 가능

2 예측 vocabulary = [a, b, c, ..., z, , ]  
ASCII code

↳  
↳ 한자, 세로 스파이스 글자를 찾을 가능

# Vanishing Gradients with RNNs



→ 올장의 속한부 예측 성능 안 좋음 //

(단수/복수 구분 등)

$y_t$  이전의 모든  $x$ 에  
영향을 받는데 이를

back prop 때 업데이트할 수  
있기 때문

상자에 NaN이 나올 수도...

Exploding Gradient는 Gradient Clipping을 해라

→ weights & activation

threshold 보다 큰 gradients vector에 대해서

NaN ("Not a Number")

너무 크지 않게 re-scale해라

가 되어버리는 문제의 원인

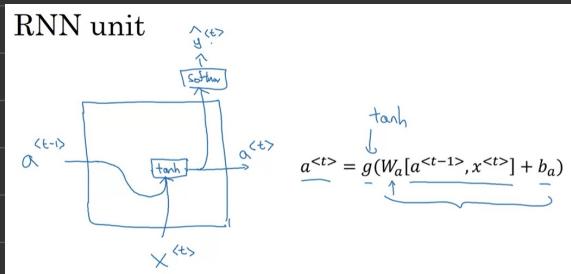
몇 가지의 최대값으로 바꿔주는 것

Vanishing Gradient는 GRUs를 해라

↳ 멀리 떨어져 있는 단어들이

서로 영향을 알 수 있게 해야 ~~하는~~ 하는데

# Gated Recurrent Unit (GRU)



## GRU (simplified)

C : memory cell 주어가 봄 / 단수인지 등을 저장

$$C^{<t>} = a^{<t>} \text{ activation value}$$

$$\tilde{C}^{<t>} = \tanh(W_c[C^{<t-1>}, x^{<t>}] + b_c)$$

$\Gamma_u$  : gate value

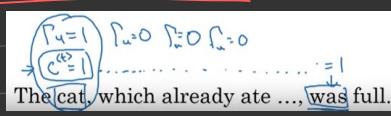
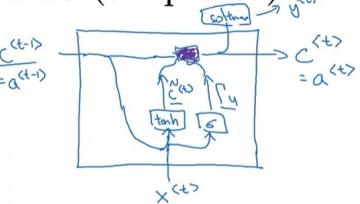
$$\Gamma_u = \sigma(W_u[C^{<t-1>}, x^{<t>}] + b_u) \quad \text{sigmoid (0 or 1)}$$

$$C^{<t>} = \underbrace{\Gamma_u * \tilde{C}^{<t>}}_{\text{element-wise}} + (1 - \Gamma_u) * C^{<t-1>}$$

$$\Gamma_u = 1 \text{ or } 0 \rightarrow 1$$

$C^{<t>} = C^{<t-1>}$  이 되면 값을 그대로 유지하면서 기록할 수 있음

## GRU (simplified)



주어에서  $\Gamma_u=1$ 은 단수를 대입  
memory cell을 update

## Full GRU

$\tilde{C}^{<t>} \leftarrow C^{<t-1>} \text{의 상관관계 } 1 \rightarrow 0$

왜 간단한 버전은 안 되고

$\Gamma_r$ 은,  $r$ 은 즉 썩 잘 .

(r : relevance  
u : update)

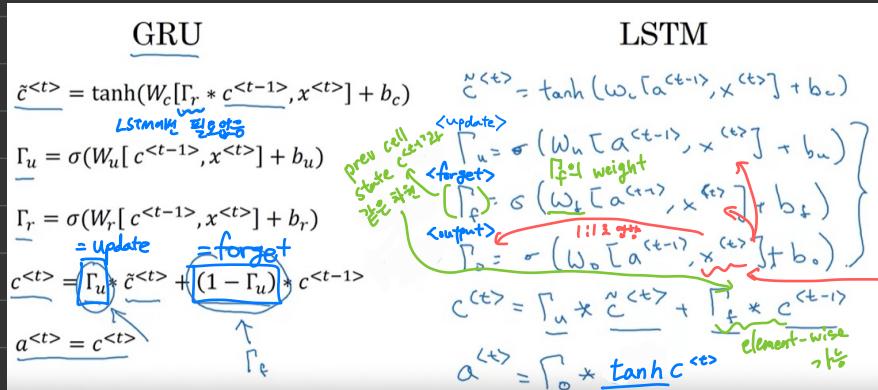
$\tilde{C}^{<t>} :$  replacing the memory cell

$$\begin{aligned} \tilde{C}^{<t>} &= \tanh(W_c[\boxed{\Gamma_r} * c^{<t-1>}, x^{<t>}] + b_c) \\ \Gamma_u &= \sigma(W_u[C^{<t-1>}, x^{<t>}] + b_u) \\ \Gamma_r &= \sigma(W_r[C^{<t-1>}, x^{<t>}] + b_r) \\ c^{<t>} &= \Gamma_u * \tilde{C}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \end{aligned}$$

# LSTM

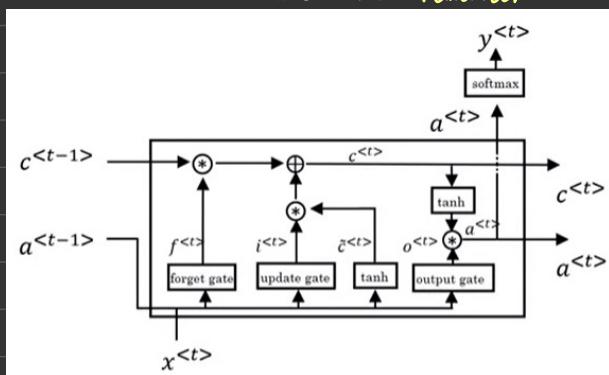
long short term memory

$x^{<t>} :$  current input  
 $a^{<t-1>} :$  prev hidden state



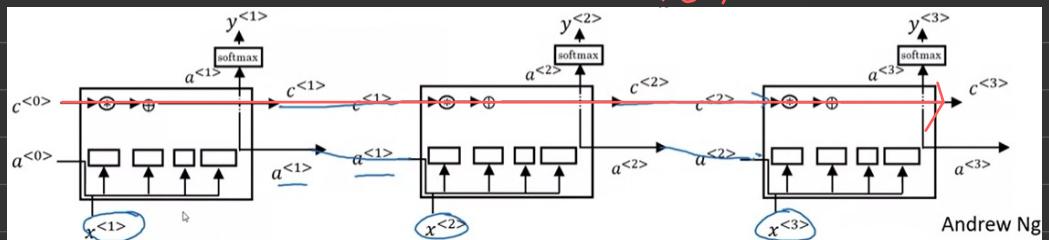
$\Gamma_f : 0 \sim 1$  < close to 0: "forget" stored state in previous state  
 close to 1: "remember"

각각의  $c^{<t>}$   
 넓으면  
 'poophole connection'  
 (LSTM 한정 내용)



단어  
 10000개의 Vocabulary  
 100차원의  $a^{<t>}$   
 $\Gamma_u$ 의 차원 = 100  
 = LSTM 속  
 hidden unit

이전 단계 ( $<\cdot>$ 를 유지하여 가진다)

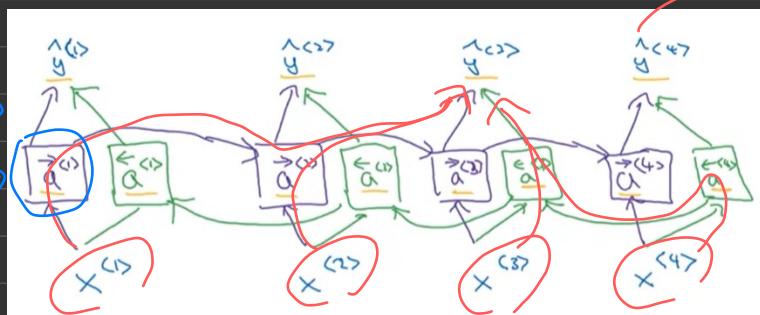


Andrew Ng

## Bi-directional RNN

추가한  
forward prop 시에 backward connection이 적용  
 $\hat{y}^{<t>} = g(W_y [\alpha^{<t>} \tilde{\alpha}^{<t>}] + b_y)$

<Block 3>  
RNN  
GRU  
LSTM  
⋮

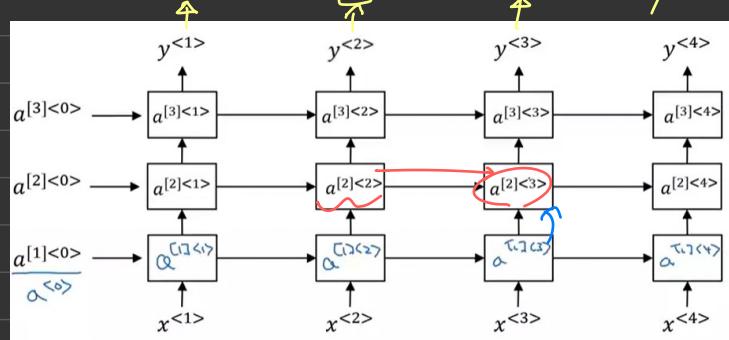


: Acyclic graph

: 한 방향이 길장을 끌어올 때까지 가기 위해 처리하는 단계

<Deep RNN>

[ ] layer  
<1> 시기 3



$$a^{<1>} = g(W_a^{<1>} [a^{<2>} \tilde{a}^{<2>}], b_a^{<1>})$$

# Word Representation

상상적 맥락을 고려하여 단어 선별을 하는 법

|              | Man<br>(5391) | Woman<br>(9853) | King<br>(4914) | Queen<br>(7157) | Apple<br>(456) | Orange<br>(6257) |
|--------------|---------------|-----------------|----------------|-----------------|----------------|------------------|
| Gender       | -1            | 1               |                |                 |                |                  |
| Royal        | 0.01          | 0.62            | 0.93           | 0.95            |                |                  |
| Age          | 0.03          | 0.02            | 0.7            | 0.69            | 0.03           | -0.02            |
| Food         | 0.04          | 0.01            | 0.02           | 0.01            | 0.95           | 0.97             |
| size<br>cost | ⋮             | ⋮               | ⋮              | ⋮               | ⋮              | ⋮                |

Vocabulary 속  
그 단어의 인덱스

↑ one-hot 벡터로 그자체만  
그 단어에 해당되는 것임

Apple과 orange는 비슷한 경향 가짐

그럼 Apple, orange 두 가지

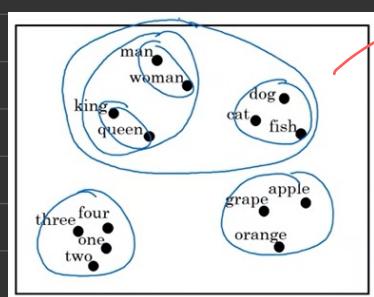
juice라는 단어가 몇 확률이

높다는 것을 학습

## Visualizing Word Embedding

< t-SNE 알고리즘 > - 단어별로 그룹화되는 경향 파악

: non-linear dimensionality reduction technique



300개의 정의라면

1개의 정당 299개의 오류 1개의 1을 찾기기에

300차원

203로 표현할 수도 있고



3차원도 가능

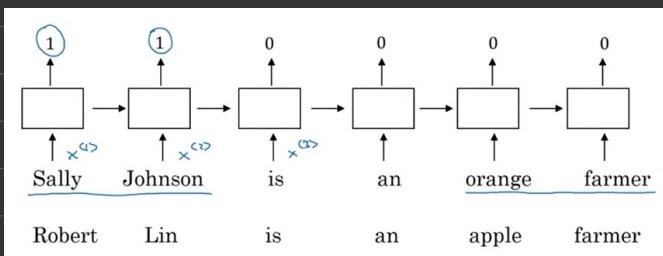
= 언어의 벡터화

## Word Embeddings

아래 문장에서 Sally Johnson이 사장임을 아는 방법은,

1. 뒤에 오는 orange farmer도 사람을 가리킨다는 걸
2. 다른 문장에서도 이 위치가 사람으로 고려되었다는 걸 등으로 추측

+1) farmer 자리에 cultivator가 들어오면,  
이 둘은 embedding하여 같은 관계↑



## Transfer Learning & Word Embedding

1. 아주 큰 문단에서 word embedding을 학습  
(또는 이미 훈련된 embedding을 온라인에서 다운로드)

Task A  
(data  $\theta_B$ )

Word Vector < Vocab  
보통 50~400 차원

2. 그 embedding을 새로운 작업으로 활용  
① 각 10,000차원의 one-hot vector 대신  
300차원의 Dense vector를 사용

Task B  
(data  $\psi_B$ )

+1) 성능 가능

3. embed한 word 육체를 미세하게 조정하기  
  ↑  
  새로운 데이터에 대처

+1) 일상 인식에서  
각 얼굴마다 생성한  
 $f(x^{(i)})$  encodings  
embedding 처리 가능

# Properties of Word Embeddings

Man:Woman as Boy:Girl  
 Ottawa:Canada as Nairobi:Kenya  
 Big:Bigger as Tall:Taller  
 Yen:Japan as Ruble:Russia

Man and Woman then King and ?

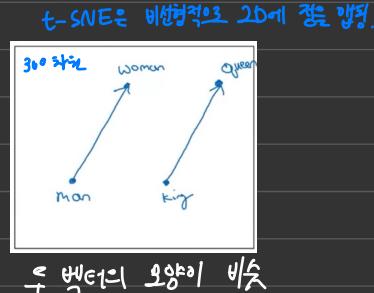
사람은 Queen이라고 떠오름  
 컴퓨터도 그렇게 하려면?

|        | Man<br>(5391) | Woman<br>(9853) | King<br>(4914) | Queen<br>(7157) | Apple<br>(456) | Orange<br>(6257) |
|--------|---------------|-----------------|----------------|-----------------|----------------|------------------|
| Gender | -1            | 1               | -0.95          | 0.97            | 0.00           | 0.01             |
| Royal  | 0.01          | 0.02            | 0.93           | 0.95            | -0.01          | 0.00             |
| Age    | 0.03          | 0.02            | 0.70           | 0.69            | 0.03           | -0.02            |
| Food   | 0.09          | 0.01            | 0.02           | 0.01            | 0.95           | 0.97             |

$$e_{\text{man}} - e_{\text{woman}} = \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad e_{\text{king}} - e_{\text{queen}} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\therefore e_{\text{man}} - e_{\text{woman}} \approx e_{\text{king}} - e_{\text{?}}$$

?  $\approx$  queen



$$? \text{ 찾기}; \arg\max \underbrace{\text{similarity}(e_?, e_{\text{king}} - e_{\text{man}} + e_{\text{woman}})}_{\text{유사성 최대화}}$$

30~75%

## Cosine similarity

$$\frac{u \cdot v}{\|u\|_2 \|v\|_2} = \frac{u^T \cdot v}{\|u\|_2 \|v\|_2}$$

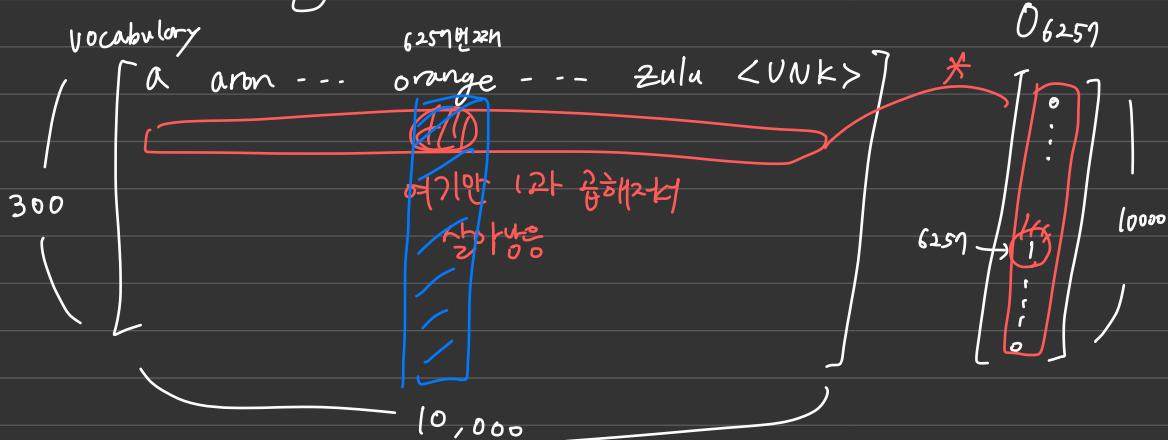


$$\cos \phi \Rightarrow \phi$$

$$\|u - v\|^2 \quad 90^\circ \text{ 라면} \\ \text{코사인 유사성은 } 0$$

$$180^\circ \text{ 라면} \\ \text{유사성 반대} \rightarrow -1$$

# Embedding Matrix



01 matrix를 E라고 하면

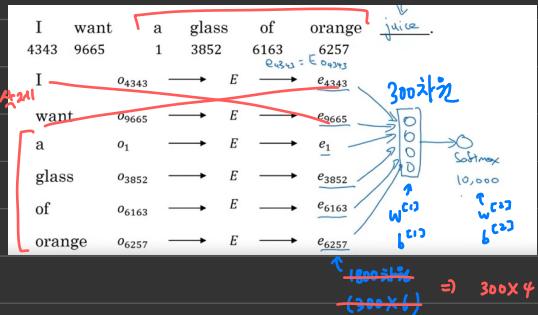
$$E \cdot O_{625} = \boxed{\boxed{}} \quad \begin{array}{l} \text{이런식으로 해방 단어} \\ \text{column은 칸} \end{array}$$

$(300, 10000) \cdot (10000, 1) \quad (300, 1)$

# Learning Word Embeddings

NLP

이전 단어만 이용하여 다음 단어 예측



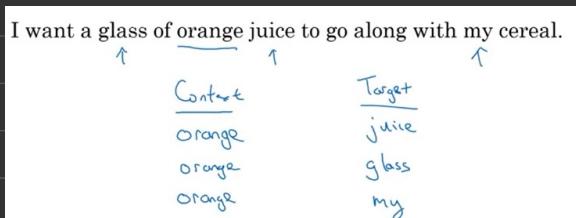
Other context / target pairs

- Context
  - type 1) 마지막 4개 단어 ↗
  - type 2) 4 words on left & right
    - ex - a glass of orange in to go along with
    - 필자의 연결성을 보는
  - type 3) 마지막 1개 단어
  - type 4) Nearby 1 word
    - ex) ~ glass ~ ?
    - ~( 중요하다고 고려된 단어 중 가까운 것 )

Word2Vec : 서로 가까이 놓여 있는  $c, t$ 을 선택 (혹은  $c$ 와  $t$ 보다 이전의 풀보드)

Skip-Grams : 전단위기 모델

문자위로 문맥의 중심이 되는 단어를 선택



Model : 문맥 단위에 오는 단어 예측

그럼 context에서  $c$ 를 생략한 방법은?

의미 없는

1. 문자위로 설정 ... 전자사가 걸릴 수 있음
2. 덜 일반적인 단어들 중에서 목적어/보어

Vocab size = 10000

$X \longrightarrow Y$

Content

$c$  : "orange"  
(6257)

Target

$t$  : "juice"  
(4834)



$c$ 에 대한  
one-hot 벡터

$E$  : embedding matrix

① : 이 모델의 parameter

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$$

$\theta_t$  :  $t$ 와 관련된 parameter

<방법>

계층적 softmax

$$\therefore L(\hat{y}, y) = - \sum_{i=1}^{10,000} y_i \log \hat{y}_i$$

$y$ 는 one-hot 벡터  
4834번 째인 [2]

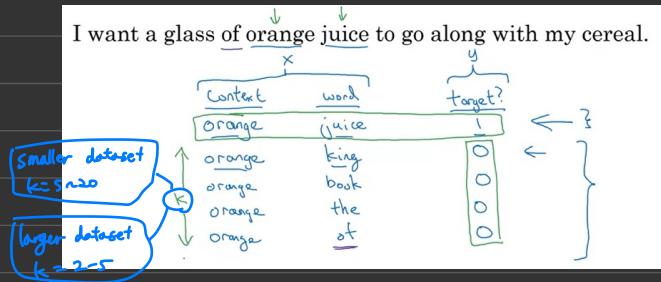
$\downarrow$  10,000개 이상의 단어라면  
합산하는데 오래 걸림.

$\Rightarrow$  시간 단축  
 $\log 1/1/3$

이는 softmax 분류의 문제임.

이 트리 구조에  
여러 알고리즘이 존재

## Negative Sampling

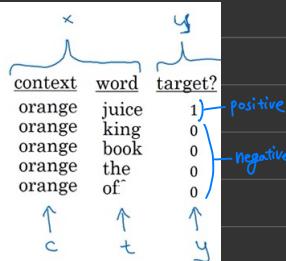


context와 연관있는 단어라면 target에 고를 듯이

## Model

Softmax:  $p(t|c) = \frac{e^{\theta_j^T e_c}}{\sum_{j=1}^{10,000} e^{\theta_j^T e_c}}$

logistic Regression Model



$O_{6257} * E \rightarrow C_{6257}$

## How to Select Negative Words?

1. 물관/문장의 중간 부분 단어를 수집  
→ 추출하여...  
2. 사람의 개인적인 경험에 기반하여 추출  
→ 논문이라는 단어 빈도의  $\frac{1}{4}$  제작에

$$P(w_i) = \frac{f(w_i)^{\frac{3}{4}}}{\sum_{j=1}^{1000} f(w_j)^{\frac{3}{4}}} \sim \frac{\text{정답 기반 무작위}}{\text{완전 무작위} \dots \text{Vocabulary} \in \{0, 1, 2, \dots, 1000\}}$$

# GloVe Word Vectors

global vectors for word representation :  $i, j$ 가 얼마나 자주 나타나는지 둘러보기

$X_{ij}$  : content  $i$ 에서  $j$ 가 나타난 빈도

만약 target이 context의 10개 단어 내로 포함하면  
 $X_{ij} = X_{ji}$ 의 대칭적 구조를 갖게됨

context 뒤에 오는 target 하나 학습기본

비대칭적으로  $X_{ij} \neq X_{ji}$

## Model

$f(x_{ij}) = 0$  라면  $x_{ij} = 0$   
 log  $x_{ij}$ 에서  $x_{ij}$ 가 0이면 음의 목한대로 가버리는  
 그 항을 삭제 시켜주기 위해 추가함

$$\text{minimize} \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(x_{ij}) (\underbrace{\phi_i^T e_j + b_i + b_j'}_{\text{Weighting term}} - \underbrace{\log x_{ij}}_0)^2$$

절대 0은 안됨!

L 영역에서 자주 나타나는 단어  
 this is, of, a ...에

가중치를 적게 주어 절제됨

무작위로

$\theta$ 와  $e$ 를 초기화하여

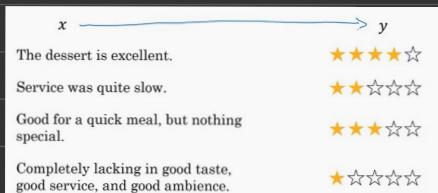
gradient descent를 따라 최소화하여

$$\Rightarrow e_w^{(\text{final})} = \frac{e_w + \theta_w}{2}$$

모든 단어를 다 읽으면 평균을 얻어냄

서로 대칭적으로 1/2 가짐

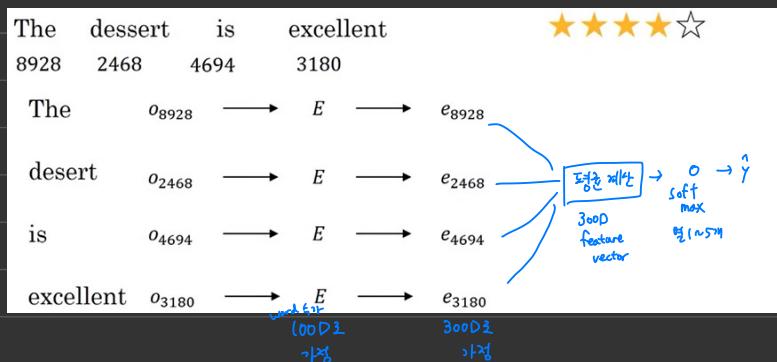
# 감성 분류 : Sentiment Classification



$x \rightarrow y$  라벨링이 된  
○ 네이버로 훈련하여  
SNS의 글의 평가도 알아낼 수 있다

Qtox 라벨링된 데이터가 적다면?

Vocabulary에 10,000개 뿐이라면? (실제로 10,001,000개 필요)



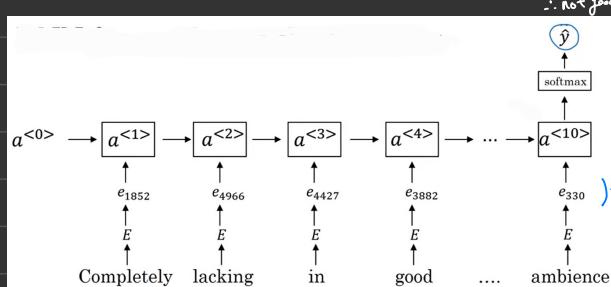
word embedding을 적용해보기  
dataset에 없는 단어가  
단적으로 들어와도 모델에 적용 가능

단점: 단어 순서 무시하고 단순 나열로 입력하기 어려움

예로 'good' 같은 단어가 끝이나면 positive로 인식할 수 있다

ex) "Completely lacking in good taste, good service"

이로 인해 RNN 적용



# Debiasing Word Embedding

### └ 서법, 인종 차별 등

## The problem of bias in word embeddings

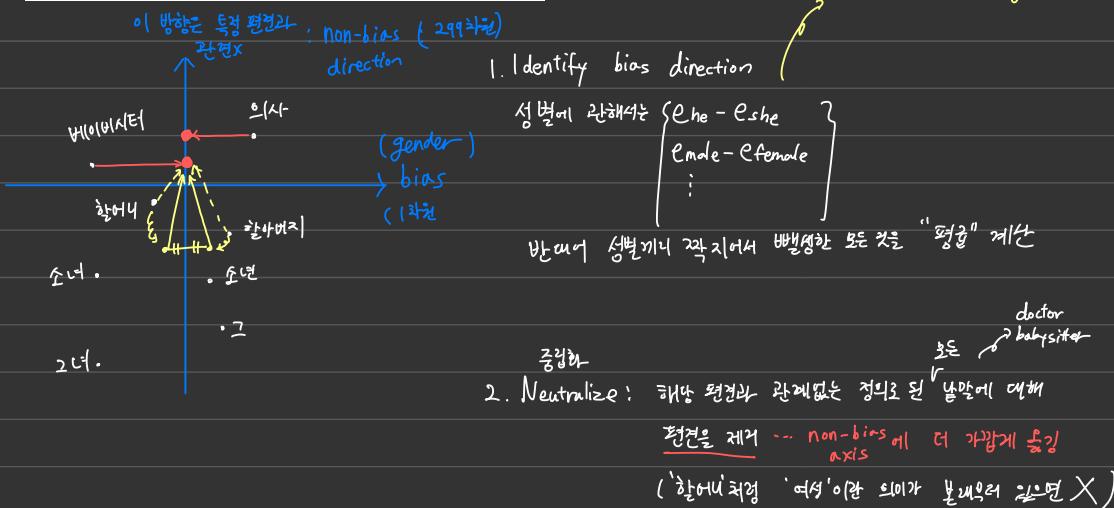
## Man:Woman as King:Queen

Man:Computer\_Programmer as Woman:Homemaker X

## Father: Doctor as Mother: Nurse X

Word embeddings can reflect gender, ethnicity, age, sexual orientation, and other biases of the text used to train the model.

$$\begin{aligned} e_{\text{boy}} - e_{\text{girl}} &\simeq e_{\text{brother}} - e_{\text{sister}} \\ e_{\text{boy}} - e_{\text{brother}} &\simeq e_{\text{girl}} - e_{\text{sister}} \end{aligned}$$



3. Equalize pairs : 관련 있는 단어끼리 짹짜이어서 성별을 표현하도록 함

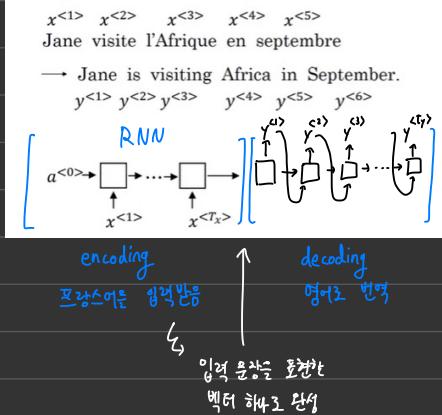
할머니 - 할아버지 ⇒ 이 두 단어가 baby sitter doctor 안아로보터  
소녀 - 소년 같은 거리에 놓이자 바꼭기

→ 동일한 거리에 있는 두 정으로  
둘 다 올길

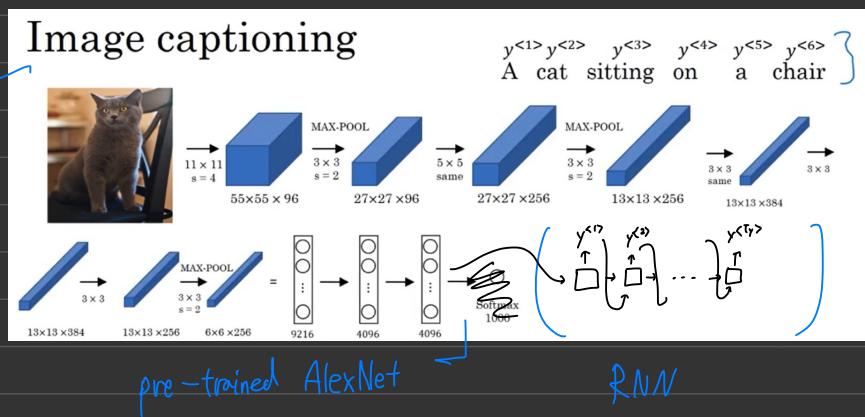
# Basic Models : 자연어 번역의 예

## < Audio Translation >

### Sequence to sequence model

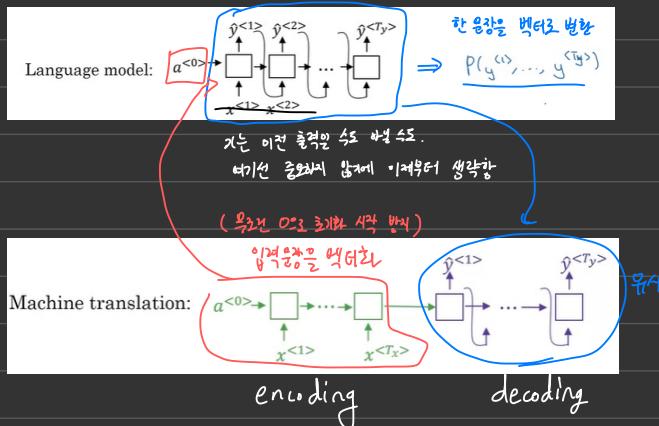


## < Image Captioning >



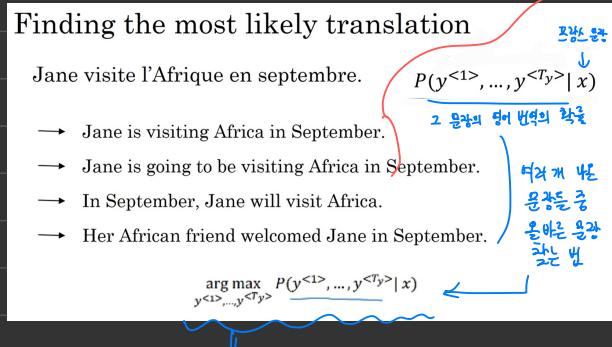
# Picking the Most Likely Sentence

7) 계 번역은 조건부 언어 모델을 만드는 것



가장 잘 된 번역 고르기 : 가장 마지막 단어에 따라  
 확률을 주제화하기.

Greedy Search



가장 가능성 있는 단어씩 선택

별로 효과 X

최적 문장도 절제 산출 --

근사 알고리즘 : 이 문장의 조건 확률을  
 극대화하는 문장으로 선택

(최적X, 최선O)

# Beam Search Algorithm

전화기록

최적의 값 X 를 찾고 싶다!

→ 영어 번역의 첫번째 단어 고르기 :  $P(y^{<1>} | x)$

이 때  $B$  (Beam width)  $\rightarrow$  가능한 단어 수만을

가능성 있는 단어 후보군을 선택

lots of memory

$\frac{B}{2} \rightarrow$  slower

large  $B$  : better result

small  $B$  : faster

worse result

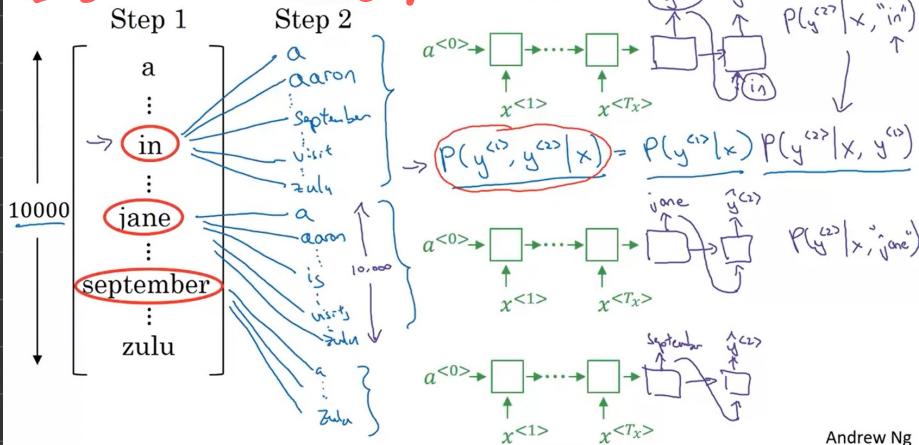
→ 두 번째 단어

$B=3$ 이자 3개 단어 선택

Step 1

Step 2

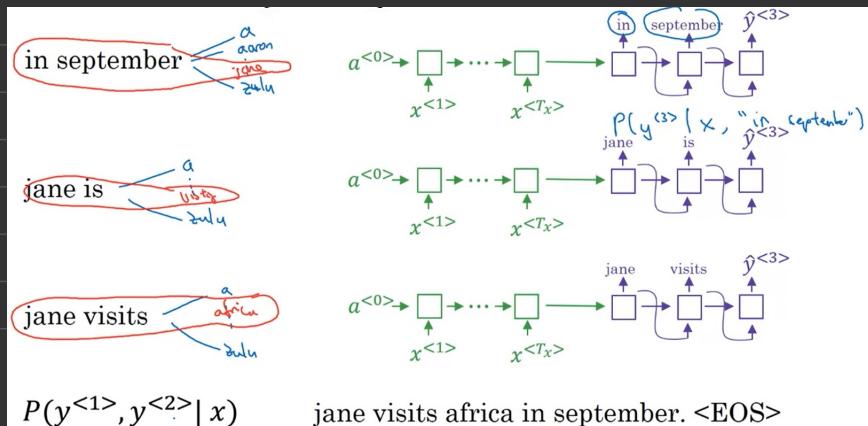
( $B=3$ )



Andrew Ng

Beam Width는 network의 복사본을 객관화하여 재사용

$\therefore (0000)^B \rightarrow (0000)^3$  번의 단어 연결/선택 과정을 줄임



# Beam Search 정제하기

$\langle \text{Length Normalization} \rangle = \text{Sentence Normalization}$  : 이걸 해야 출력 문장이 짧아지지 않음

$$\arg\max_y \prod_{t=1}^{T_y} P(y^{(t)} | x, y^{(1)}, \dots, y^{(t-1)}) \leq P(y^{(1)} \dots y^{(T_y)} | x) \leq P(y^{(1)} | x) P(y^{(2)} | x, y^{(1)}) \dots$$

확률은 항상 (보라 각기어, 이) 숫자를 계속 곱하다보면 원래 값이 너무 작아져서 흐트럼.

$$\downarrow \log \text{ 추가}$$

반올림 오류도 예방. 최대화하면서 단조증으로 증가  $\cancel{+/-}$

$$\arg\max_y \prod_{t=1}^{T_y} (\log P(y^{(t)} | x, y^{(1)}, \dots, y^{(t-1)}))$$

$\downarrow \frac{1}{T_y} \Rightarrow$  각 단어의 확률을 기록하여 평균 계산  $\Rightarrow$  정규화  
한 번역에도 잘 버리기 귀하

$$\text{단어의 수} \rightarrow \underbrace{\frac{1}{T_y}}_{\alpha} \arg\max_y \prod_{t=1}^{T_y} \log P(y^{(t)} | x, y^{(1)}, \dots, y^{(t-1)})$$

s

이외에 더 브드려는  
방법을 쓰기도

$$\downarrow \alpha \text{ 추가}$$

$$\frac{1}{T_y} \arg\max_y \prod_{t=1}^{T_y} (\log P(y^{(t)} | x, y^{(1)}, \dots, y^{(t-1)}))$$

$\alpha = 0.7 \dots \alpha = 1$  이면  $T_y$  개로 정규화  
 $\alpha = 0 \dots 1$  면 정규화 적용

문서 분석

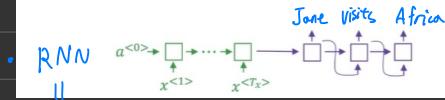
# Error Analysis in Beam Search

Jane visite l'Afrique en septembre.

Human: Jane visits Africa in September. ( $\hat{y}^*$ )

Algorithm: Jane visited Africa last September. ( $\hat{y}$ ) - bad

$\rightarrow$  RNN이 문제인가  
Beam이 문제인가  $\rightarrow$  BP but it will be slow



encoder + decoder

$\Rightarrow$  RNN으로 2하는 것,

$p(y^*|x)$ 에 대한  $p(\hat{y}|x)$

문제는 끝났고, 고려해야 할 수도 있는

Case 1 사양  
 $p(y^*|x) > p(\hat{y}|x)$

이는 Beam Search가  $\hat{y}$ 를 선택했을 때

$y^*$ 가  $p(y|x)$ 보다 높다면

Beam Search의 문제

그러면 Beam Search는  $P(y|x)^{\frac{1}{2}}$

최대한하는 것이 올바르기 때문

$\rightarrow$  Beam Width 조정

Case 2  $p(y^*|x) \leq p(\hat{y}|x)$

$y^*$ 의 번역이  $\hat{y}$ 보다 나은데

RNN 예측이  $P(y^*|x) < P(\hat{y}|x)$ 라면  
RNN의 문제 |  $\hat{y}$ 의 출현률 축소

$\rightarrow$  정규화 추가 for deeper layer

$\rightarrow$  Get more train set

$\rightarrow$  다른 신경망 사용 . . .

# Bleu Score

$\hat{=} MT$   
 베이스 시스템은 정확성 측정 방법 : 어떤가  
 이전과 가까운 수록 높은 점수 정하는가?

기제 번역 결과 속 단어가

$$\text{몇 개의 음절 속에 들어 있는지 확인} \quad (\text{precision}) = \frac{\text{나타낸 봄의 수}}{\text{한 음절 속에}} \quad \text{단어}$$

$\Rightarrow$  각 음절에 대한 나타난 헛스가 (modified precision)

## Evaluating machine translation

French: Le chat est sur le tapis.

둘다  
옳은 번역

Reference 1: The cat is on the mat.  
 Reference 2: There is a cat on the mat.

Bleu  
 bilingual evaluation diversity

MT output: the the the the the the.

Precision:  $\frac{7}{7}$

Modified precision:  $\frac{2}{7} \leftarrow \text{Count}_{clip}(\text{"the"})$   
 $\leftarrow \text{Count}(\text{"the"})$

(단어(쌍))

1-3 4-5-1 나타낸 음이에 대비한 Bleu 점수로 표기

## Bleu Score on bigrams

Example: Reference 1: The cat is on the mat.  $\leftarrow$

Reference 2: There is a cat on the mat.

| MT 속 bigrams |       | $\frac{\text{정답}}{\text{나타낸}}$ in MT |       | $\frac{\text{정답}}{\text{나타낸}}$ in Reference |      |
|--------------|-------|--------------------------------------|-------|---------------------------------------------|------|
|              | Count |                                      | Count |                                             | Clip |
| the cat      | 2     |                                      | 1     |                                             |      |
| cat the      | 1     |                                      | 0     |                                             |      |
| cat on       | 1     |                                      | 1     |                                             |      |
| on the       | 1     |                                      | 1     |                                             |      |
| the mat      | 1     |                                      | 1     |                                             |      |

$$P_1 = \frac{\sum_{\substack{\text{정답도} \\ \text{unigram } \in y}} \text{Count}_{clip}(\text{unigram})}{\sum_{\substack{\text{정답도} \\ \text{unigram } \in y}} \text{Count}(\text{unigram})}$$

MT가 정답 1 ~ 정답 24  
 정확히 일치하면  
 $P_1, P_2, \dots, P_n = 1.0$

S

$$P_n = \frac{\sum_{\substack{\text{n-gram} \\ \text{정답도}}} \text{Count}_{clip}(n\text{-gram})}{\sum_{\substack{\text{n-gram} \\ \text{정답도}}} \text{Count}(n\text{-gram})}$$

# Bleu Detail

$p_n = \text{Bleu score on } n\text{-gram only}$

$$\text{Bleu score 향치기} : \underbrace{\text{BP}}_{\substack{\text{exponentiation을 적용하여 linear을 벗어남} \\ \text{그전에 이는 기하 급수적으로 증가해서 BP로 조정}}}, \underbrace{\exp\left(\frac{1}{4} \sum_{n=1}^4 p_n\right)}$$

brevity penalty (가짜로 처리)

장점

$$\text{BP} \begin{cases} 1 & \text{if MT\_output\_length} > \underbrace{\text{reference\_output\_length}}_{\text{정확성 산출}} \\ \exp(1 - \text{ref\_output\_len} / \text{MT\_output\_len}) & \text{otherwise} \end{cases}$$

매우 짧은 번역에선 높은 정확성 산출.

둘다면 짧은 모든 단어가 이미 정확한 번역에 포함.

긴 문장에서도 잘 되기 위해 BP에 위의 공식을 적용

MT가 더 길면 그를 적용.

그렇게 짧은 번역을 이제

# Attention model intuition



encoder에서 문장을 끝까지 읽어야만 decoder에서 번역 가능.

그러나 인간은 부분적으로 알고 번역하면서 진행한다

인 문장을 외우는 게 더 어렵기에

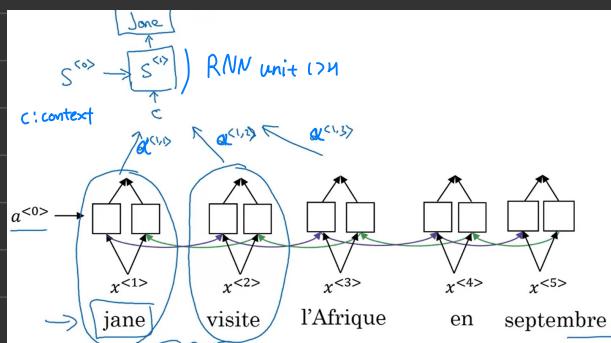
둘 다 그늘은 문장에서만 높은 정확성을 보임

이걸 기계에 적용하면 인 문장에서도 정확성↑

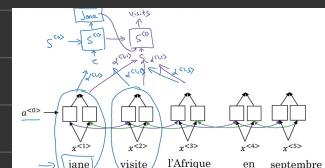
GRU, LSTM으로도 가능

양방향 RNN 사용 → 각 단어를 둘러싼 맥락 파악에 좋음  
 ↳ 첫단어를 선택하는데 마지막 단어까지 확인할 필요X

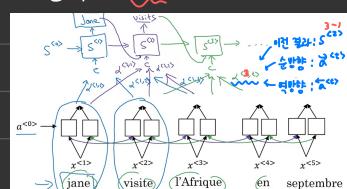
(단계)-



2단계 - 이전 출력이 다음 입력으로



3단계 → EOS를 만날 때까지



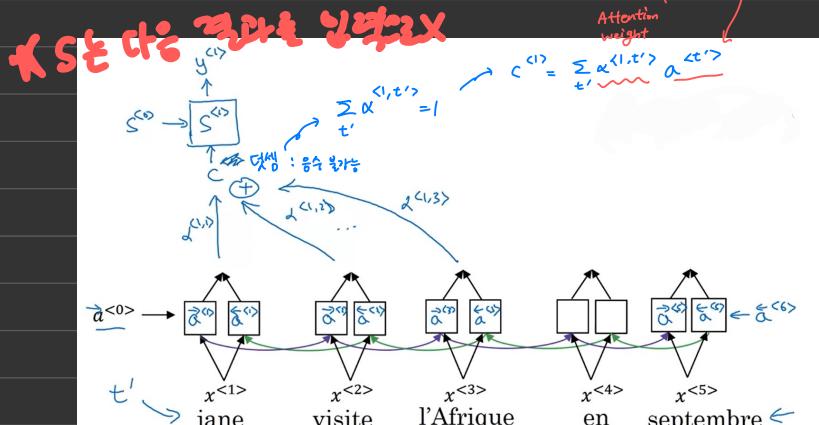
## Attention Model

$$\vec{\alpha} : \text{forward}, \quad \overleftarrow{\alpha} : \text{backward}, \quad \alpha^{<\leftrightarrow>} = (\vec{\alpha}^{<\leftrightarrow>}, \overleftarrow{\alpha}^{<\leftrightarrow>})$$

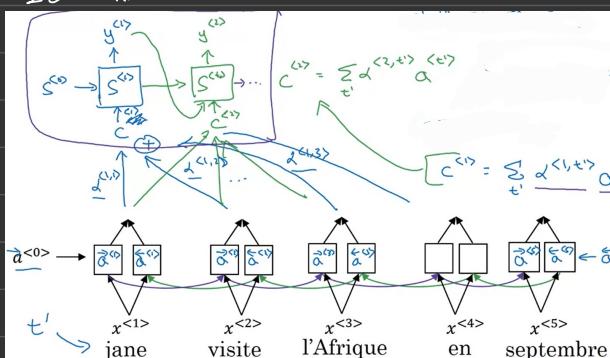
양방향이라 원쪽에  $\vec{\alpha}^{(c)}$ , 오른쪽에  $\vec{\alpha}^{(e)}$  넣으

$$\begin{aligned} \text{포함 } & \text{문장이면 } t' \text{로 표현} \\ \rightarrow & \underline{\alpha^{ct'}} = (\vec{\alpha}^{ct'}, \underline{\alpha}^{ct'}) \\ \alpha^{ct'} &= (\vec{\alpha}^{ct'}, \underline{\alpha}^{ct'}) \end{aligned}$$

? 가  $\alpha^{ctr}$ ?에



$$|\gamma_{\text{true}}| = \frac{b_i}{\beta_{NN} \cdot |\eta|}$$



## 6) 호텔의 + ) 단점

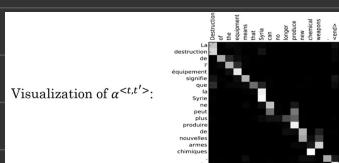
입력 단어  $T_x$ , 출력 단어  $T_y$  개  
 일 때  $\alpha^{(t,t')} = T_x \times T_y$  개 필요

### +) 적용 예시 - 형식 변환

$$\langle \text{Computing Attention } a^{(t,t')} \rangle = y^{(t)} \mapsto a^{(t,t')}_{\text{out}}$$

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=l}^{T_x} \exp(e^{<t,t'>})}$$

## 7. 예 대회 모두 뛰는 것



$$+) \sum_{t'} a^{<t,t'>} = 1$$

(Note  $\Sigma$  is over  $t'$ )

# Speech Recognition



단어 하나씩 정확히 음성 기반으로 번역 X

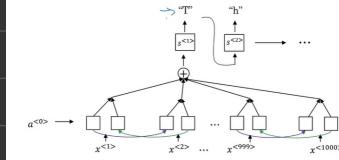
3000h 정도의 dataset 부터 훈련이 끝나기 시작

100,000h 까지 가능도.

방법 1. 양방향 RNN 계열



Attention model for speech recognition



방법 2) CTC cost for speech recognition

L Connectionist temporal classification

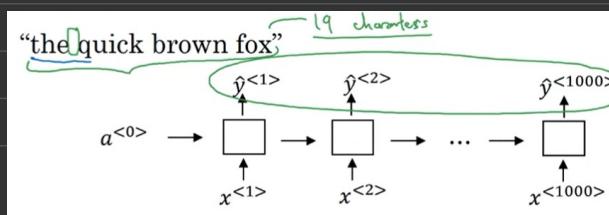
: Connection의 확장자

$x$ 와  $y$ 의 수가 같은 상황에 단방향 RNN으로 설명하나, 실제로 양방향 사용

Ex) 10초 분량의 오디오를 100Hz, 초당 100개의 셀룰로 전달하면, 1000개의 입력이 됨

기본 규칙: blank는 처리되지 않은 반복된 문자를 제거

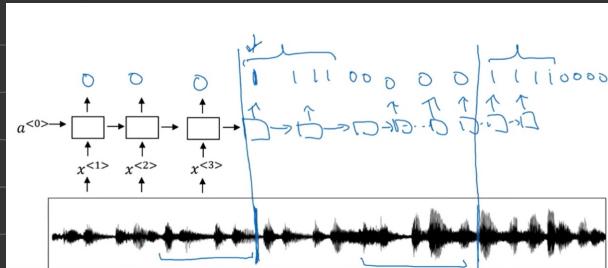
t t t - h \_ e e e \_ \_ w \_ - - g g g - -  $\Rightarrow$  the w g  
space blank



# Trigger Word Detection

입력 데이터 중이가 불규칙하다는 단점

Trigger Word 예시안 13 2 또는 0으로



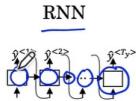
$x^{<t>} :$  Features of the audio at time t

ex) spectrogram features

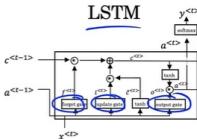
# Transformer Network Intuition

## Transformer Network Motivation

increased complexity  
sequential



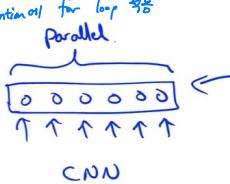
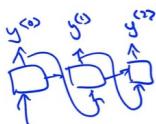
GRU



한 문장은 병렬적으로 입력  $\rightarrow$  이 방식은 이미지에서도 적용 가능

## Transformer Network Intuition

- Attention + CNN
  - Self-Attention
  - Multi-Head Attention : self-Attention for long 距離 parallel



# Self-Attention

$k$ : qualities of words given a  $Q$   
 $v$ : specific representations of word given a  $Q$

## Self-Attention Intuition

$$A(q, K, V) = \text{attention-based vector representation of a word}$$

calculate for each word  $A^{<1>} \dots A^{<5>}$

RNN Attention

$$\alpha^{<t,t'} = \frac{\exp(e^{<t,t'}))}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'}))}$$

$x^{<1>} \downarrow$   
Jane

$x^{<2>} \downarrow$   
visite

Word-embedding  
 $x^{<3>} \downarrow$   
l'Afrique

Transformers Attention

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$

$x^{<4>} \downarrow$   
en

$x^{<5>} \downarrow$   
septembre

softmax-like

'아프리카'라는 단어가 "2번째로 큰 매트릭스" or "여기" 등

어떤 맥락으로 사용되었는지 알기 위해

문장 속 다른 단어에 대해서 계산

vector

$q$ : query  
 $k$ : key  
 $v$ : value

각 단어마다  $g^{<3>}$  같은

단어를 찾을

단어가 있으면

제일 큰  $k$ 를  $A$ 에 넣을 수 있게 한 경우

$\rightarrow A$ : attention value

$$q^{<3>} = W_q x^{<3>}$$

$$k^{<3>} = W_k x^{<3>}$$

$$v^{<3>} = W_v x^{<3>}$$

$W$ : attention weight matrix

• 잘 정의하도록 학습

## Self-Attention

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$

$A^{<3>}$

| Query (Q)               | Key (K)                 | Value (V)               |
|-------------------------|-------------------------|-------------------------|
| $q^{<1>} \dots q^{<5>}$ | $k^{<1>} \dots k^{<5>}$ | $v^{<1>} \dots v^{<5>}$ |

[Vaswani et al. 2017, Attention Is All You Need]

단어가 주어졌을 때,  
단어 값을 합산하여 해당 단어와 관련된

attention 정도를 맵팅 함으로써

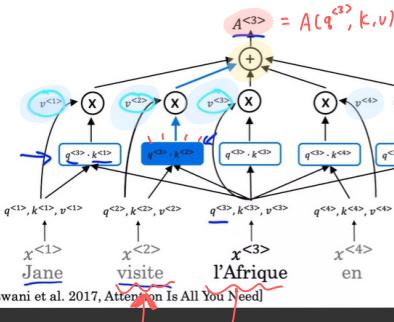
단어의 맥락을 계산하는데 이를 단어의 '%

matrix of 차원

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{dk}}\right)V$$

## Self-Attention

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$



| Query (Q)               | Key (K)                 | Value (V)               |
|-------------------------|-------------------------|-------------------------|
| $q^{<1>} \dots q^{<5>}$ | $k^{<1>} \dots k^{<5>}$ | $v^{<1>} \dots v^{<5>}$ |

문맥을

∴ 2번 Africa에 대해

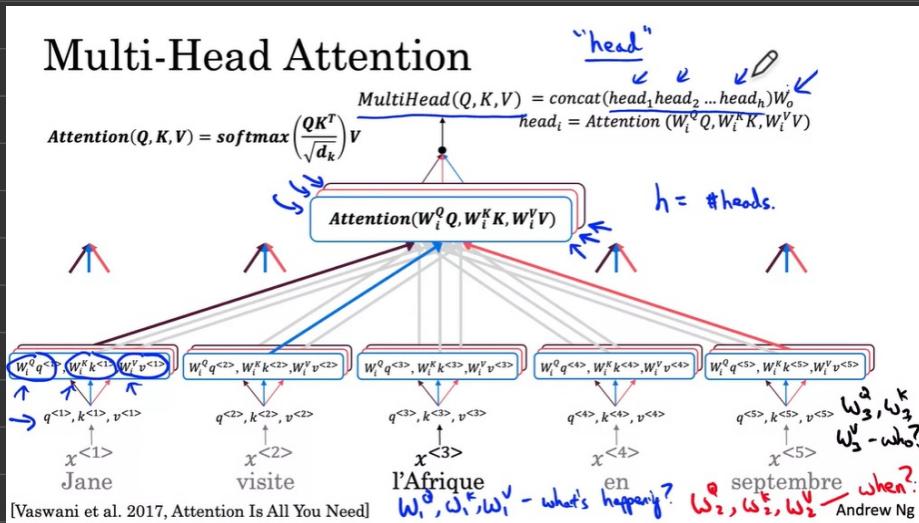
"아프리카" 방을 "까지" 알게됨

# Multi-Head Attention

$W$ 를 2개 이상으로 하여

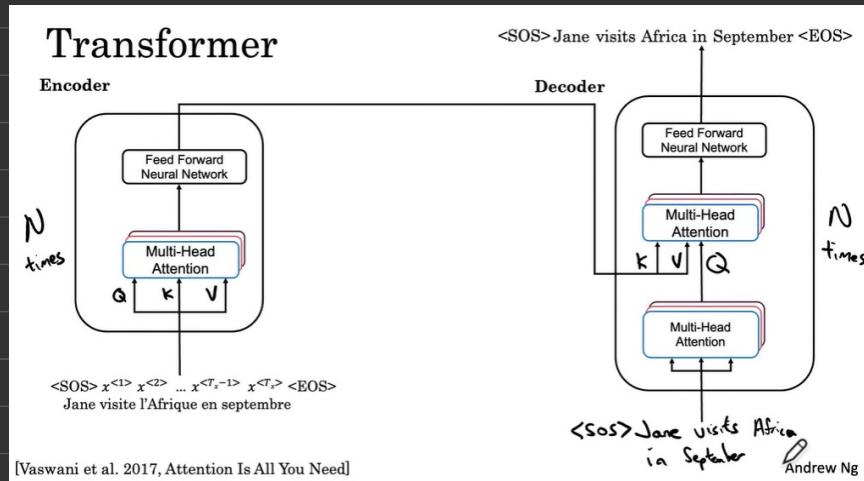
for  $i \in \{1, 2, 3\}$

- $W_i \in \text{What's happening} \rightarrow \text{간식}$  같은 것
- $W_2 \in \text{when}$
- $W_3 \in \text{who}$

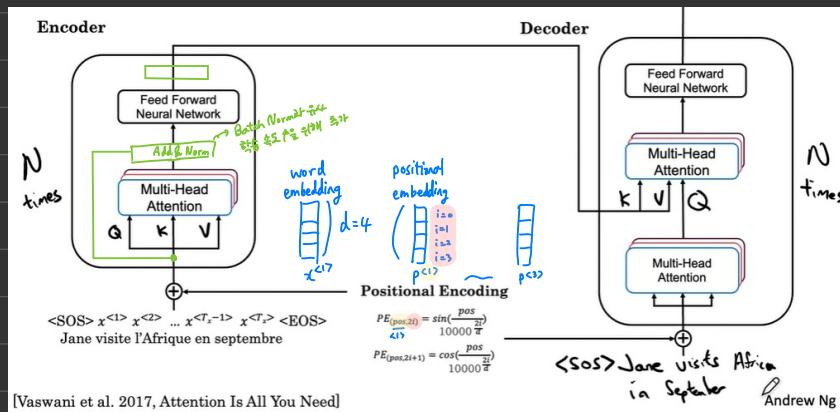


# Transformer Network

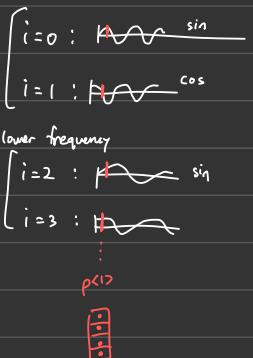
Attention 방식  
CNN style의 processing을 위한 것



## Transformer + Positional Encoding



$\sin, \cos$ 은  $i \sim 1$  사이로 만들면  
그냥 그대로呗 word embeddings와  
positional embeddings를 합해呗  
크게 대조되지 않음



encoding  
< 좋은 positional algorithm은 가능한 >

1. 각 time-step (문장에서 단어의 위치)에 대해 고정한 encoding을 활용함

2. 각 time-step 간 거리는 모든 문장 길이에 대해 일관성이 있어야 함

3. 같은 풀장도 일반화할 수 있음

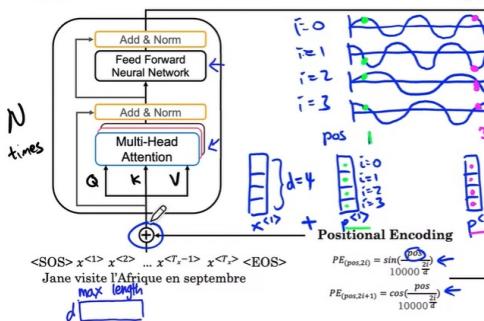
$d$ : word positional embeddings의 차원

$pos$ : 단어의 위치

$i$ : positional encoding의 차원

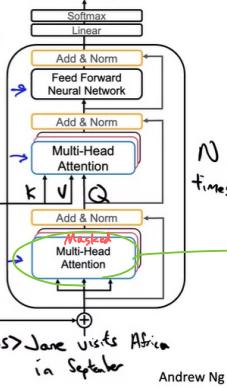
# Transformer Details

Encoder



$<\text{SOS}> \text{Jane visits Africa in September } <\text{EOS}>$

Decoder



$$\text{softmax} \left( \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} + \mathbf{V} \right) V$$

mask  
key와 value - 범위를 줄여  
softmax 품질 scale down

영어 - 프랑스 번역 시 추가  
이전에는 한 번에 한 단어씩 예측  
한 단어씩 할 필요가 없음

즉 mask가 하는 일은

완벽히 번역된 것처럼

반복적으로 위장하는 것

Mask

입력이

Type 1) Padding mask - 최대 문장 길이를 초과 시

0을 채우거나, 잘라내서,

transformer model에 동일한 길이로 전달

→ 0은 softmax에서 영향을 주기 어려움

-1e9를 사용

Type 2) look-ahead mask

위에서 사용한 방식