Improving Deep Neural Network

: Hyperparameter tuning, Regularization and Optimization

[1-1. Setting up your Machine Learning Application]

< Train / Dev / Test sets >

- 신경망이 몇개의 층을 가지는지, 각 층이 몇개의 은닉층을 가지는지, 학습률과 활성화 함수는 무엇인지 등을 결정해 신경망을 훈련시켜야 한다.
- 좋은 하이퍼파라미터를 찾기 위해서는 사이클을 여러번 반복해야 한다.

* train / dev / test sets

Data	training sets	dev sets	test sets

- hold-out cross validation
- Development set 'dev'

- 과거에는, train : test = 70% : 30% 로 나누곤 했음.
- Big data 시대에는, 예를들어 데이터가 1,000,000개가 있다고 하면 dev set과 test set을 1% 만 사용해도 됨

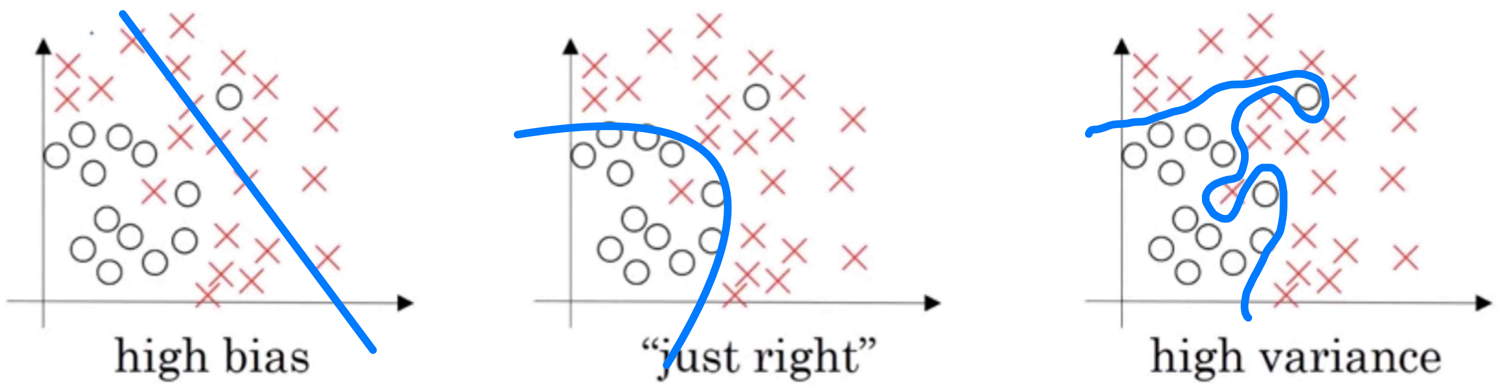
* Mismatched train / test distribution

ex) training sets: Cat pictures from webpages
Dev / test sets: Cat pictures from users using your app
→ 개발 셋과 테스트 셋이 같은 분포에서 나온건지 확인해야 함

⊕ test set이 없이 dev set만 있어도 됨

< Bias and Variance >

* 편향과 분산의 트레이드 오프



↳ under fitting

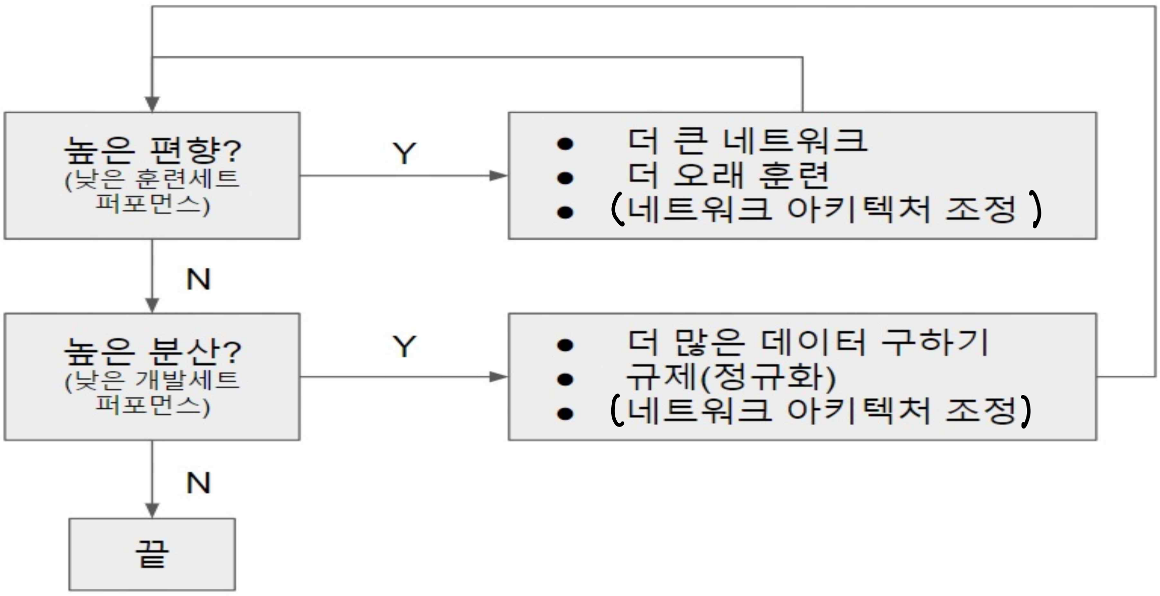
↳ over fitting

* 훈련 셋과 개발 셋의 관계

- 가정 : 인간 수준의 성능이 기본이 되어야 한다. 일반적으로 말하면, 베이지안 최적 오차가 0% 임

error	높은 분산 (과대적합)	높은 편향 (과소적합)	높은 편향 & 높은 분산	낮은 편향 & 낮은 분산
훈련 세트	1 %	15 %	15 %	0.5 %
개발 세트	11 %	11 %	30 %	1 %

<Basic Recipe for Machine Learning>



[1-2. Regularizing your Neural Network]

< Regularization >

* Logistic Regression

regularization parameter

생략

비용함수 : $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 + \cancel{\frac{\lambda}{2m}b^2}$, $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

L2 regularization : $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

L1 regularization : $\frac{\lambda}{2m}\sum_{i=1}^{m}|w| = \frac{\lambda}{2m}\|w\|_1$ → w will be sparse (= w가 꽤 0을 가지고 있음)

⇒ 모델 압축에 도움이 될 수 있음.

* Neural Network

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

→ Frobenius 노름 : $\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$ $(\because w : (n^{[l]}, n^{[l-1]}))$

- L2 정규화가 weight decay라고 불리는 이유

$w^{[l]} := w^{[l]} - \alpha dw^{[l]} = w^{[l]} - \alpha\{(\text{from backprop}) + \frac{\lambda}{m}w^{[l]}\}$

$= w^{[l]} - \frac{\alpha\lambda}{m}w^{[l]} - \alpha(\text{from backprop})$

$= (1 - \frac{\alpha\lambda}{m})w^{[l]} - \alpha(\text{from backprop})$

↳ 즉, 1보다 작은 값인 $(1-\frac{\alpha\lambda}{m})$가 곱해지기 때문

< Why Regularization Reduces Overfitting? >

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

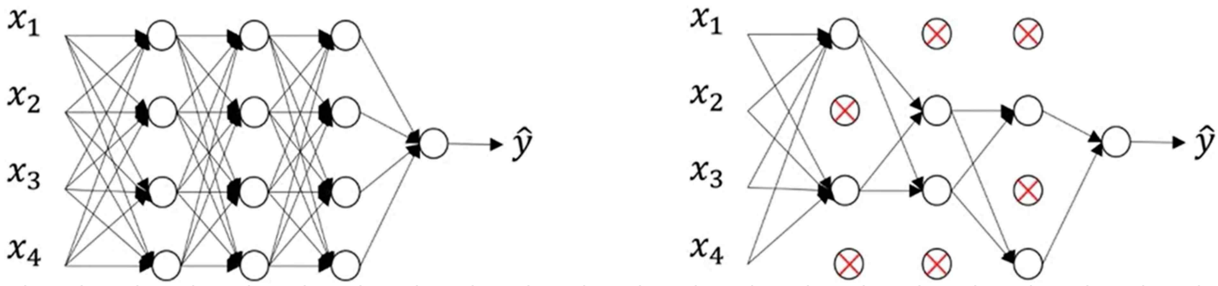
- 이때 λ를 매우 크게 하면, $w^{[l]} \approx 0$ 이 되고, (∵ 비용함수 최소화)

⇒ 그 결과 hidden units 의 개수가 작아져서

⇒ 작은 신경망이 되어, 과대적합이 일어나지 않는다.

< Dropout Regularization >

* Dropout Regularization



- 드롭아웃 방식 : 신경망의 각각의 층에 대해 노드를 삭제하는 확률을 설정하는 것. 삭제할 노드를 선정한 후, 삭제된 노드의 들어가는 링크와 나가는 링크를 모두 삭제함.
  → 더 작고 간소화된 네트워크가 만들어지고, 이 작아진 네트워크로 훈련함.

* Implementing dropout

ex) layer = 3 , keep_prob = 0.8

d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob

a3 = np.multiply(a3, d3)   # a3 *= d3

$\boxed{\text{a3 /= keep\_prob}}$ → Inverted dropout technique

: dropout을 적용하기 전과 동일하게 활성화 값의 기대값으로 맞춰주기 위해 노드를 삭제후 얻은 활성화 값에 keep.prob (삭제하지 않을 확률)을 나눈다

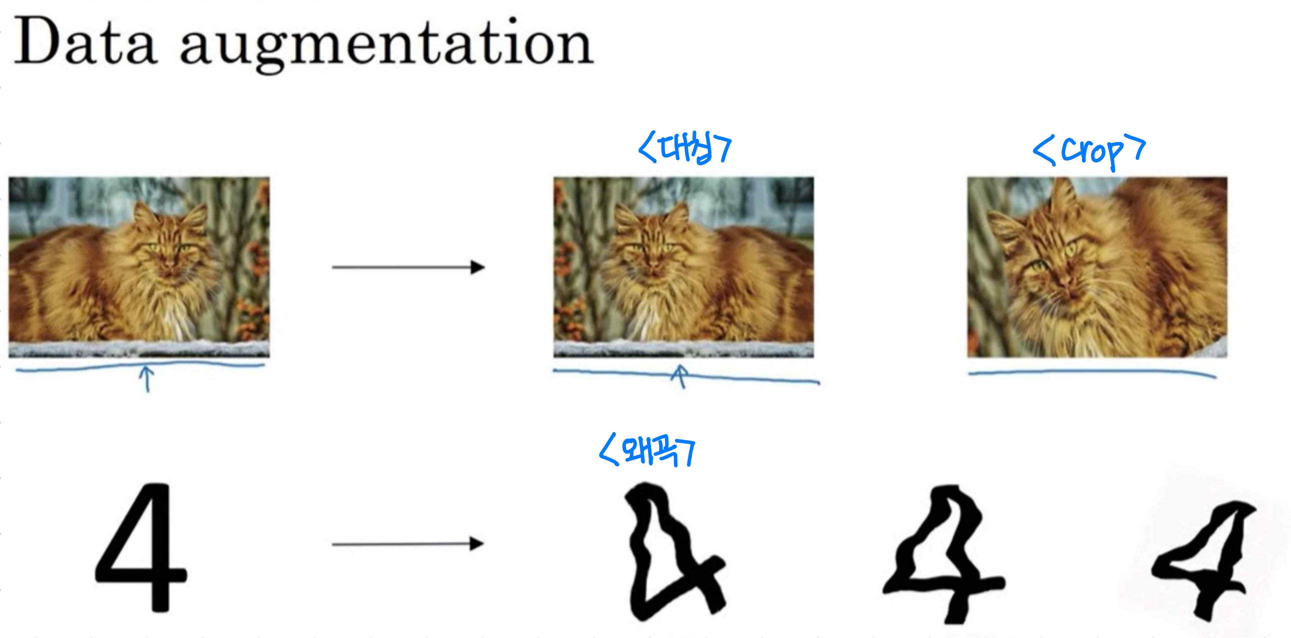
⊕ test time 에서는 dropout을 진행하지 않는다.

< Understanding Dropout >

- 드롭아웃은 랜덤으로 노드를 삭제시키기 때문에, 하나의 특성에 의존하지 못하게 만듦으로서 가중치를 다른곳에 분산
- 드롭아웃의 keep.prop 확률은 층마다 다르게 설정 가능
- 비용함수가 단조감소인지 우선 확인한 후, 드롭아웃을 사용해야 함

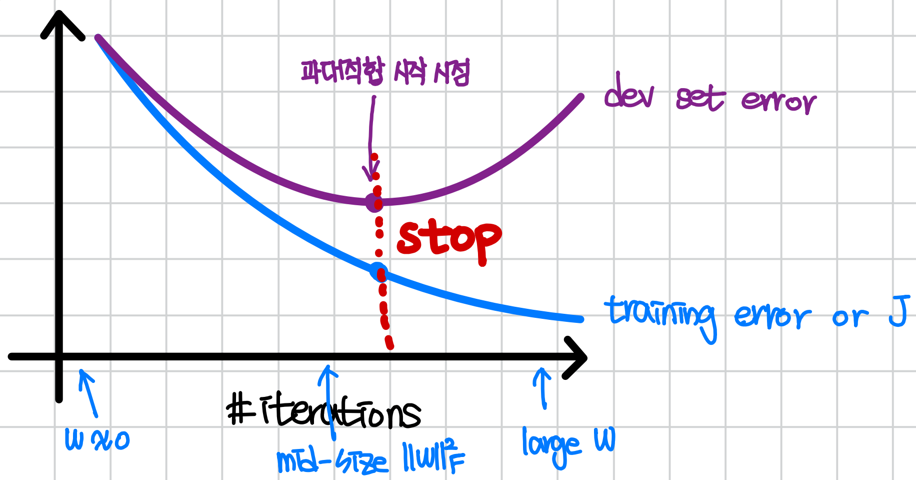
<Other Regularization Methods>

* Data augmentation



- 이미지의 경우, 더 많은 트레이닝 셋을 사용하여 과대적합을 해결할 수 있음
- 대칭, 확대, 왜곡, 회전 등을 이용하여 새로운 데이터 생성
- 이런 가짜이미지는 완전히 새로운 샘플보다 더 적은 정보를 추가하지만, 비용이 들지 않는 장점

* Early stopping



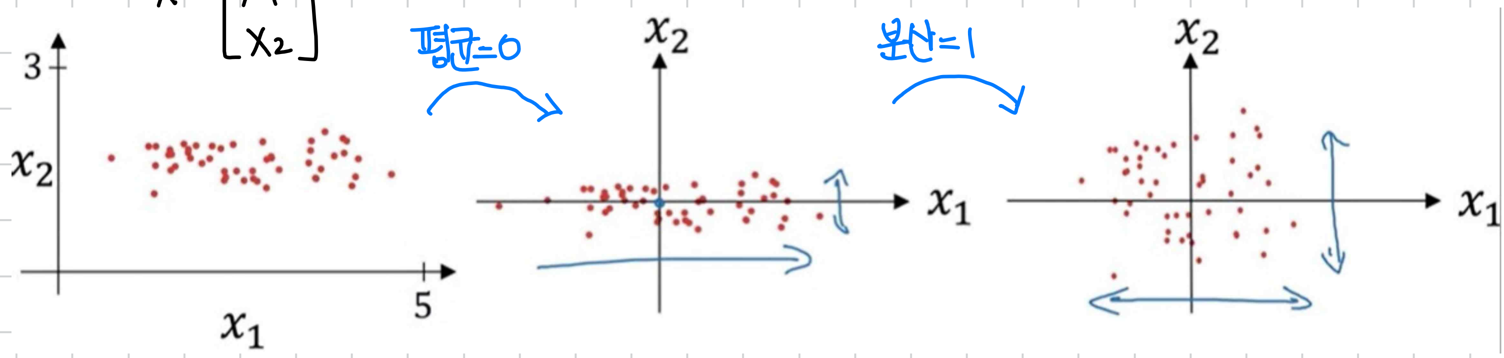
- early stopping: 신경망이 dev set의 오차 저점 부근, 즉 가장 잘 작동하는 점에서 훈련 stop
- 단점: training 시, ① 비용함수 최적화 ② 과대적합 하지 않도록 이 두가지는 별개의 일 (Orthogonalization) 이므로 다른 접근법을 사용해야 하지만, early stopping은 두가지를 섞어서 최적의 조건을 못 찾을 수도 있음.

[1-3. Setting Up your Optimization Problem]

<Normalizing Inputs>

* Normalizing traing sets

ex) $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$



① 평균을 0으로 만든다

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

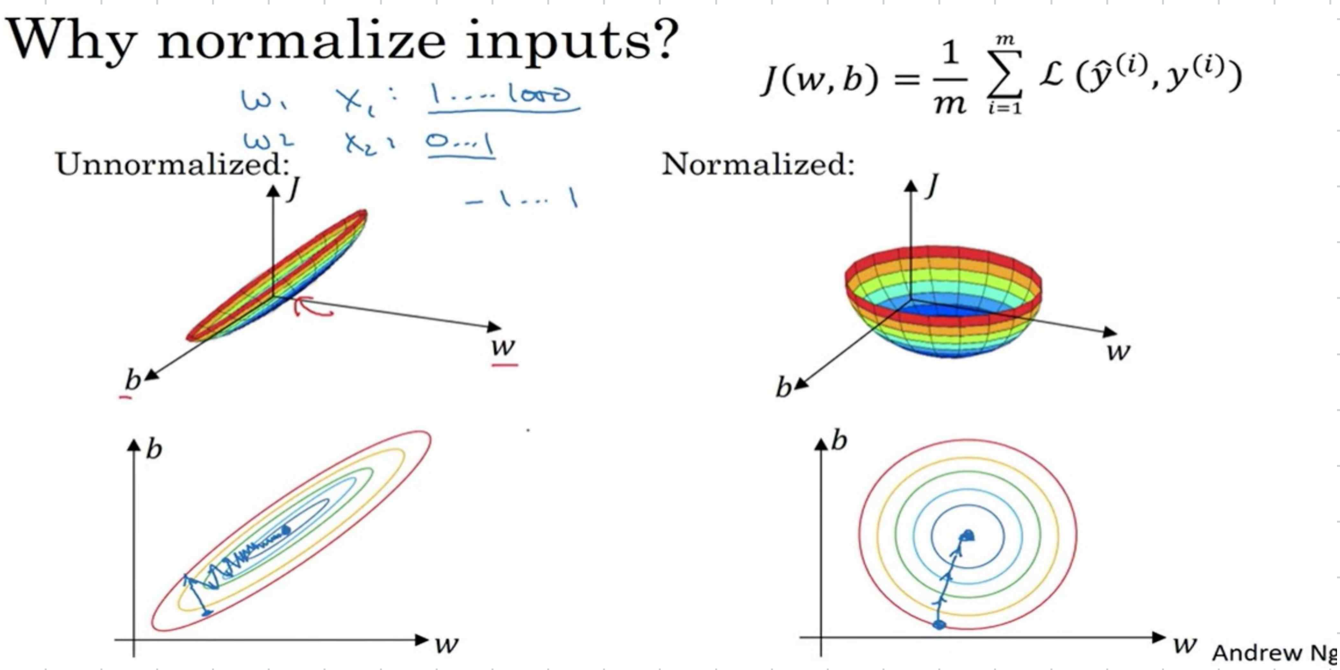
$$x := x - \mu$$

② 분산을 1로 만든다.

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

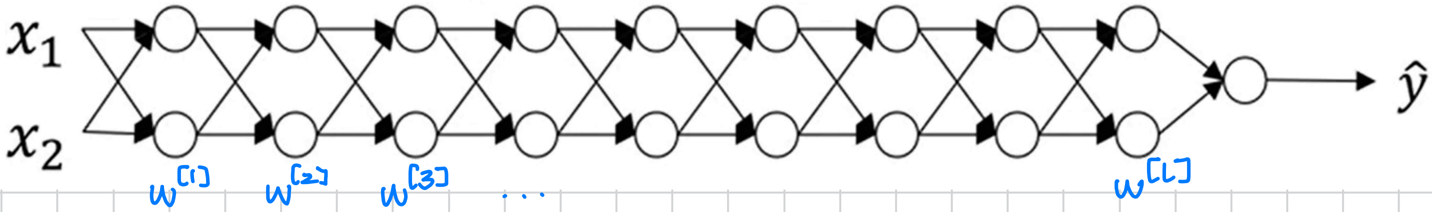
$$x := \frac{x}{\sigma^2}$$

- 테스트 셋을 정규화할 때, train set에 사용한 μ, σ를 사용해야 함



→ 정규화를 통해 비용함수의 모양이 더 둥글고 최적화하기 쉬워져서 학습 알고리즘이 빨리 실행됨

< Vanishing / Exploding Gradients >



if) $g(z) = z$, $b^{[l]} = 0$

$\rightarrow \hat{y} = w^{[L]} \cdot w^{[L-1]} \cdots w^{[3]} \cdot w^{[2]} \cdot w^{[1]} x$

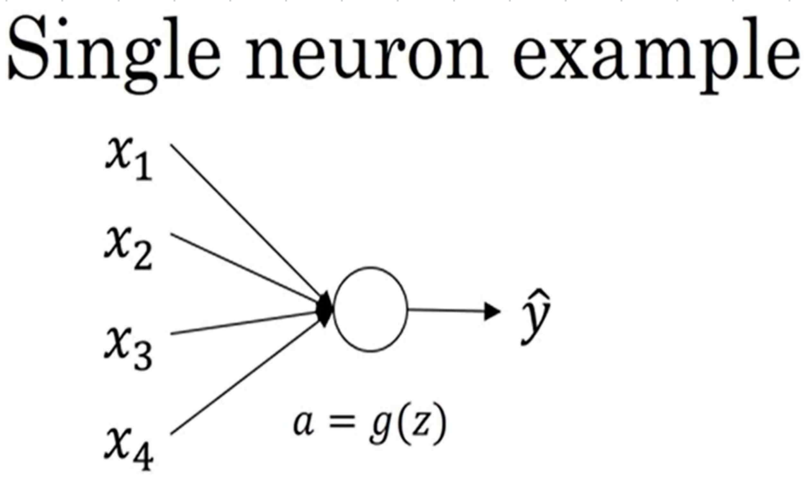
① $W = 1.5E \rightarrow w^{[l]} > I$

: $\hat{y} = w^{[L]} \cdot 1.5^{[L-1]} E x$ ⟶ 신경망이 깊어질수록 $\hat{y}$이 기하급수적으로 "커짐"

② $W = 0.5E \rightarrow w^{[l]} < I$

: $\hat{y} = w^{[L]} \cdot 0.5^{[L-1]} E x$ ⟶ 신경망이 깊어질수록 $\hat{y}$이 기하급수적으로 "작아짐"

< Weight Initialization for Deep Networks >



$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + \cancel{b}$ 0가정

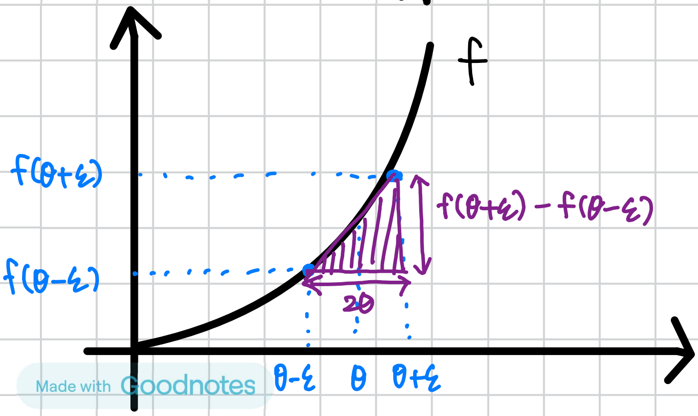
$\rightarrow Var(W_i) = \frac{1}{n}$ 입력특성의 개수

$W^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$

i) ReLU $\rightarrow V(W_i) = \frac{2}{n^{[l-1]}}$

ii) tanh $\rightarrow V(W_i) = \frac{1}{n^{[l-1]}}$ or $\frac{2}{n^{[l-1]} + n^{[l]}}$

< Numerical Approximation of Gradients > → 역전파를 알맞게 구현했는지 확인하고자!



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

$\rightarrow f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$

< Gradient Checking >

① $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$을 big vector θ의 하나로 concatenate

: $J(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}) = J(\theta)$

② $dW^{[1]}, db^{[1]}, \cdots, dW^{[L]}, db^{[L]}$을 big vector $d\theta$의 하나로 concatenate

→ for each i :

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

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

③ 수치미분과 일반미분의 값을 비교해서 유클리디안 거리가 10^{-7}보다 작은지 확인

→ 유클리디안 거리 : $\dfrac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx 10^{-7} \Rightarrow$ great!

< Gradient checking implementation notes >

Made with Goodnotes