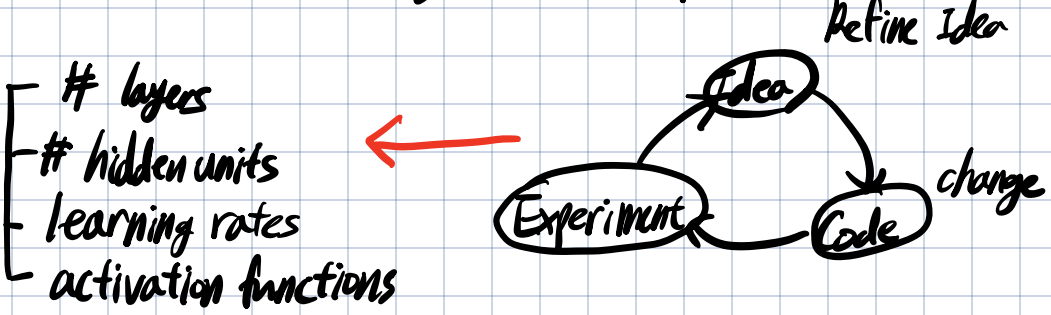
WEEK 1. Setting up your Machine Learning Application

1. Setting up your Machine Learning Applications

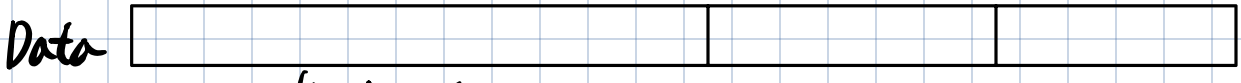
- Train / Dev / Test sets

- Applied ML is a highly iterative process



⇒ go round this cycle repeatedly & efficiently to find best hyperparameters

- Train / dev / test sets for efficient cycle



training set

Hold-out
cross validation
Development set

dev set

test

see which algorithm is better

- Previous era (Machine learning): split data to 70/30 % or 60/20/20%
- Big data era (1,000,000): dev & test set ↓ to quickly decide which algorithm is better → 98/1/1 % or 99/0.5/0.5 %

e.g. 1,000,000 → 10,000 is enough for dev set (10,000) and test set (10,000)

- Mismatched train/test distribution

e.g. training set: Cat pictures from webpages ⇒ high resolution

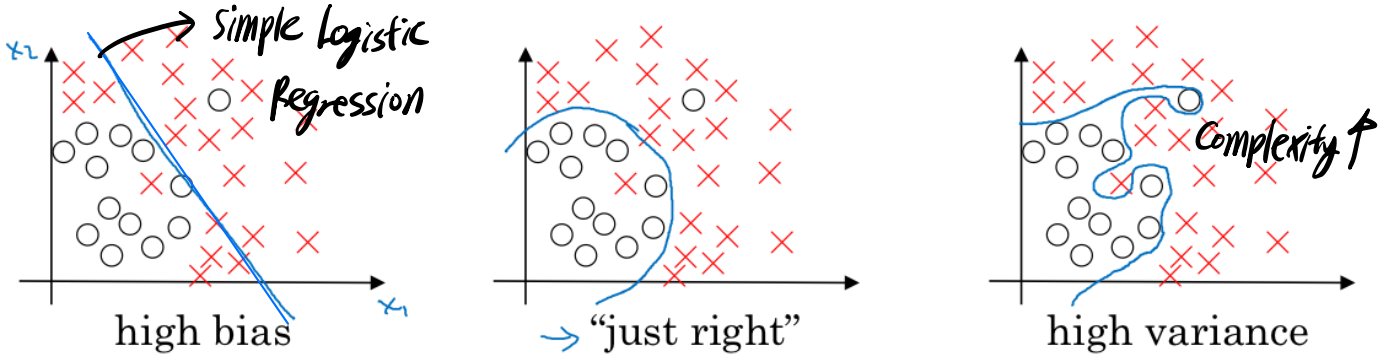
Dev/test sets: Cat pictures from users using your app ⇒ blurry / low resolution

⇒ distribution between these dataset can be different

[Make sure that dev and test sets come from the same distribution.
Not having a test set might be okay. (only dev set)
↳ giving you unbiased estimate of performance
⇒ if you don't need unbiased estimate, it's okay

- **Bias / Variance**

- less discussions in trend



Underfitting ⇓ Overfitting ⇓

⇒ In high dimension, it is hard to plot the decision boundary

e.g. cat classification Cat (y=1) vs Dog (y=0)

- 2 keys for bias & variance (Human: x%, Optimal (Bayes) ≈ 0%) ← standard

bias → Train set error:	1%	15%	15%	0.5%
Generalization ⇒ Variance → Dev set error:	11%	16%	30%	1%

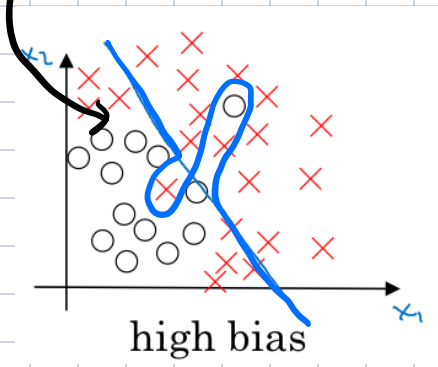
↓ 만약 이미지가 너무 흐려 안 보이면 x ⇒ 커진다

↳ low bias

Overfitting to training set (low generalization)
⇒ high variance

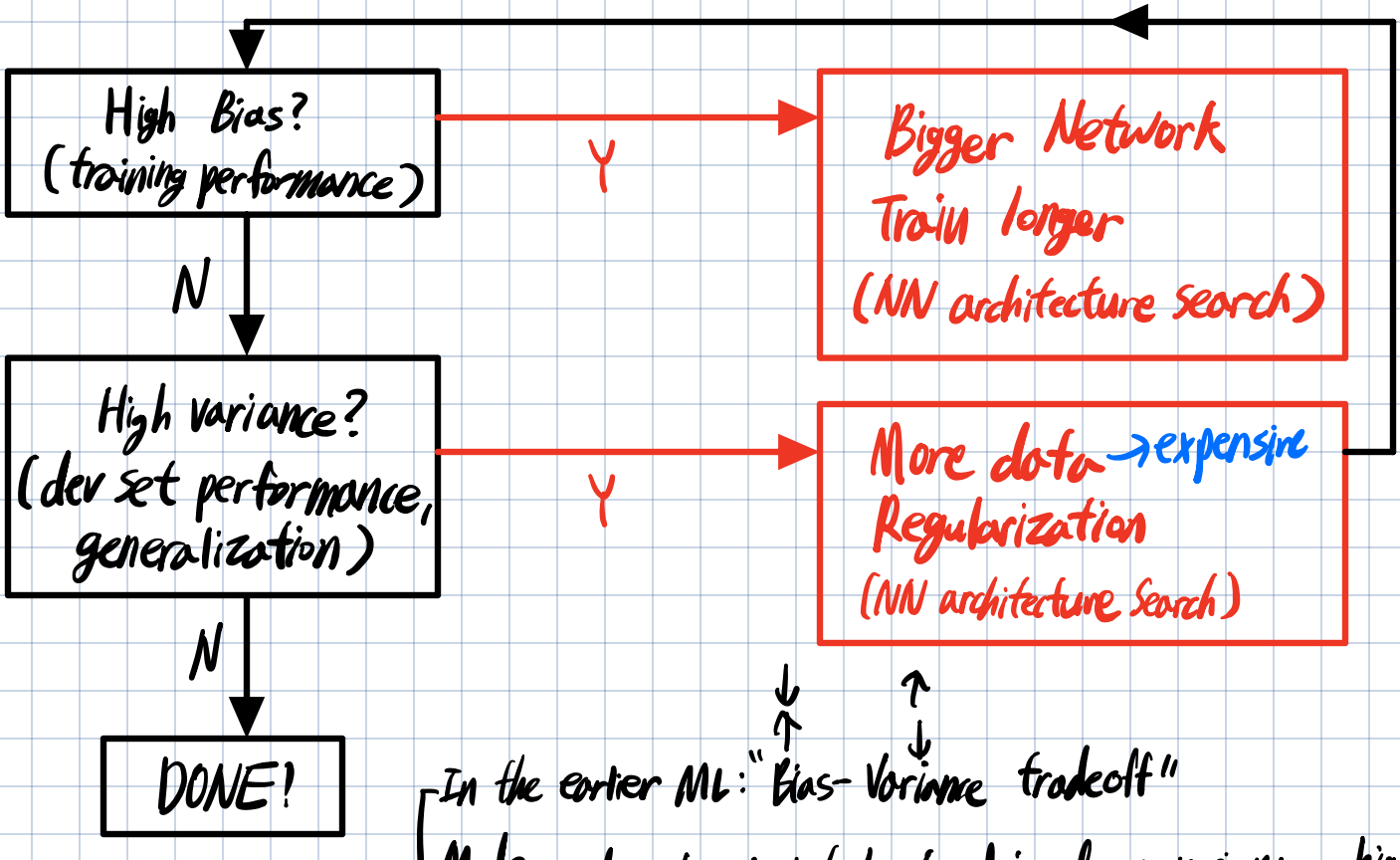
It didn't even fit training set (Underfitting)
⇒ high bias

high bias & high variance
⇓
overfitting to a part of data



& low variance

- Basic "recipe" for machine learning
  - After training initial model, check bias & variance by using training/dev set
    ⇒ Depending on its bias and variance, solution can be different



In the earlier ML: "Bias-Variance tradeoff"
Modern deep learning: tools to drive down variance or bias without hurting each other.

2. Regularizing your neural network.

• Regularization

- Overfitting due to High variance ⇒ Regularization
- Regularization of Logistic Regression

$w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, λ: regularization parameter (mbd) → hyperparameter used in cross-validation and dev set

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

- $\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y)$: cross-entropy cost
- omit ∵ almost all parameters are w to fit well (high dimensional)

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$ (Euclidean norm of w)

↳ L2 norm of vector w

- L1 Regularization $\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}\|w\|_1$ ⇒ w often be sparse = w vector has many zeros
- L2 Regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$

• Regularization of Neural Network

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

- Frobenius norm: norm of matrix w $(n^{[l-1]}, n^{[l]})$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2$$

↳ square of each element in matrix w

↳ weight update 할 때 w가 작아지게 만든다

- if λ is too large → oversmooth (model with high bias)

- Backpropagation

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

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

$= w^{[l]} - \alpha[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}]$ → extra term의 추가 penalty를 줌.

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

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

⇒ make $w^{[l]}$ a little bit smaller by multiplying something < 1

∴ "L2 Regularization = Weight decay"

• Why regularization reduces overfitting

- How does regularization prevent overfitting?

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

⇒ reduce L2/Ferbenius norm ⇒ less overfitting

- Intuitions

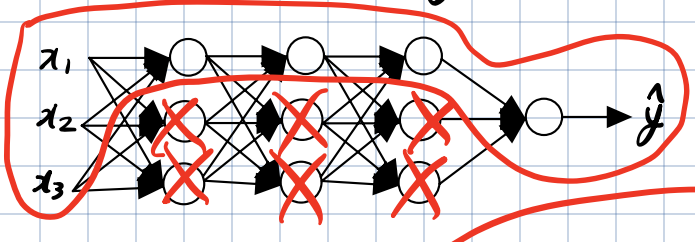
① λ big: make $w^{[l]} \approx 0$ ⇒ set many hidden units 0

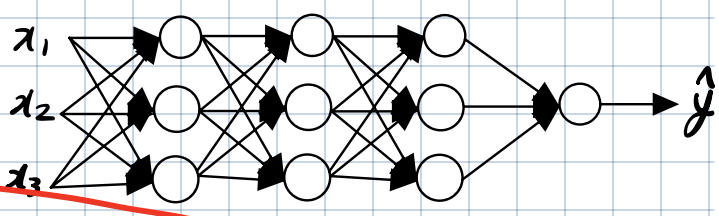
⇒ reduces impacts of hidden units

⇒ close to LR unit (simpler network)

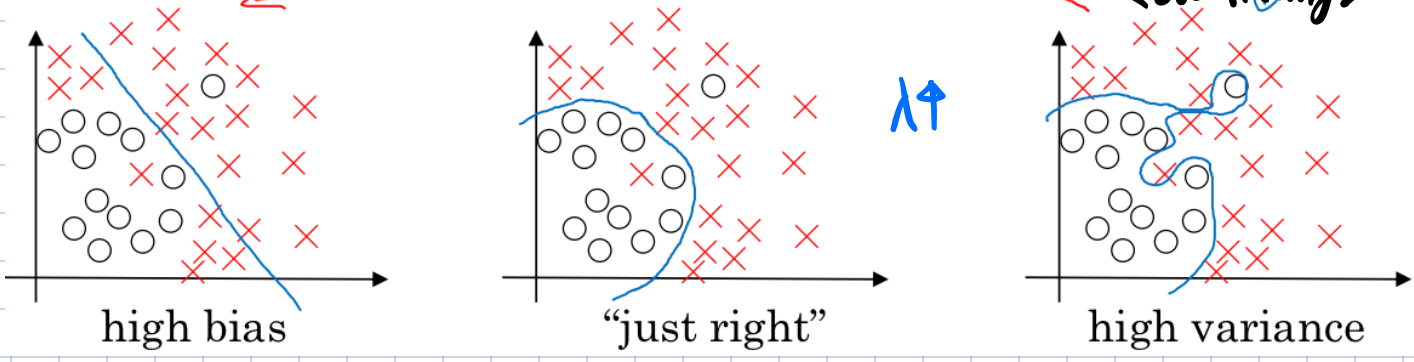
⇒ reduces overfitting

- It can hurt training set performance but ultimately it gives better test accuracy.



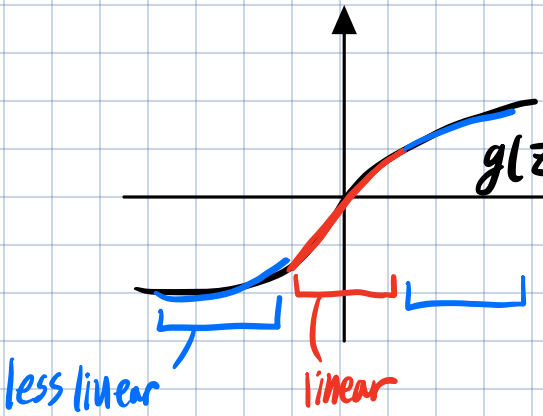


<overfitting>



② If z takes on only small ranges of parameters

⇒ just using linear region of tanh function



to minimize Cost function

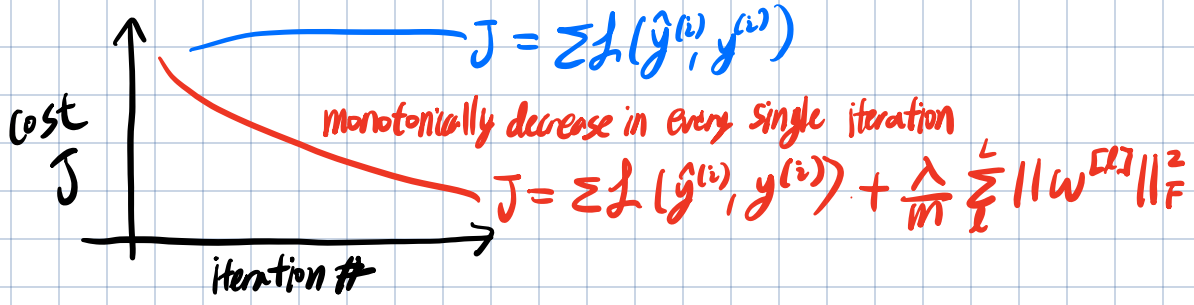
if $\lambda\uparrow : w^{[l]}\downarrow \rightarrow z^{[l]}\downarrow = w^{[l]}a^{[l-1]}+b^{[l]}$

⇒ z takes small range of value, g(z) is almost linear

⇒ Every hidden layer ≈ linear ⇒ linear network

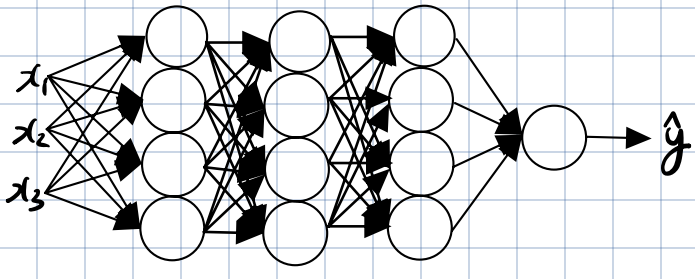
⇒ simpler network ⇒ overfitting↓

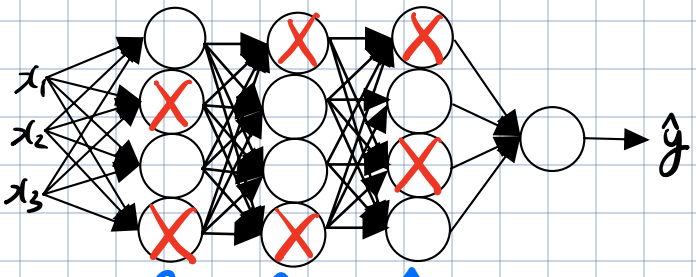
- Tips for implementation of gradient descent



• Dropout Regularization

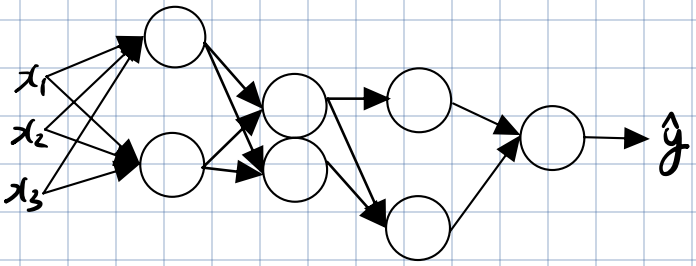
— Let's train this overfitting network





e.g. 0.5 chance of keeping / removing nodes

① go through each of the layers in the network and set some probability of eliminating the node of neural network



② remove every in/outgoing links from removed nodes and perform backpropagation in the smaller Network

③ repeat for every training samples

-Implementing dropout ("Inverted dropout")

e.g. `l=3 keep_prob=0.8`

↳ layer probability of hidden unit to be kept for layer 3

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

↳ dropout vector for layer 3 — d3는 각각의 샘플 & 각 유닛에 대해 [0.8의 확률로 1, 0.2의 확률로 0] 이 된다.

zero out hidden unit

`a3 = np.multiply(a3, d3)` or `a3 *= d3`

`a3 /= keep_prob` scaling step

inverted-dropout technique

⇒ keep_prob을 어떤 값으로 설정하든 keep_prob을 다시 나눠줘서 a3의 기대값을 같게 유지

즉, 최종 cost가 dropout을 사용하지 않았을 때와 동일한 값을 유지하도록 마지막에 나눠준다.

e.g. 50 units in 3rd layer —keep_prob=0.8→ 10 units shut off

$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$

↑ 20% of elements are zeroed-out

⇒ dropout 전 A3의 기대값은 A3 그 자체, but dropout 후 A3의 기대값은 원래 값의 80%이다. ⇒ keep_prob으로 나눠 기대값 유지

⇒ d3는 경사하강법 한 번 반복마다 0이 되는 은닉 유닛들이 달라진다. (첫번째 훈련 세트의 d ≠ 두번째 ㅇ)

— Making predictions at test time

$a^{[0]} = X \ (= X\_test)$

test에서는 no drop out ⇒ random하게 hidden unit을 zero-out하지 않음

$z^{[1]} = w^{[1]}a^{[0]} + b^{[1]}$ (output $\hat{y}$이 random하게 나오면 X, dropout을 적용하면 test의 noise만 커진다.)

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

$z^{[2]} = w^{[2]}a^{[1]} + b^{[2]}$

$a^{[2]} = \cdots$

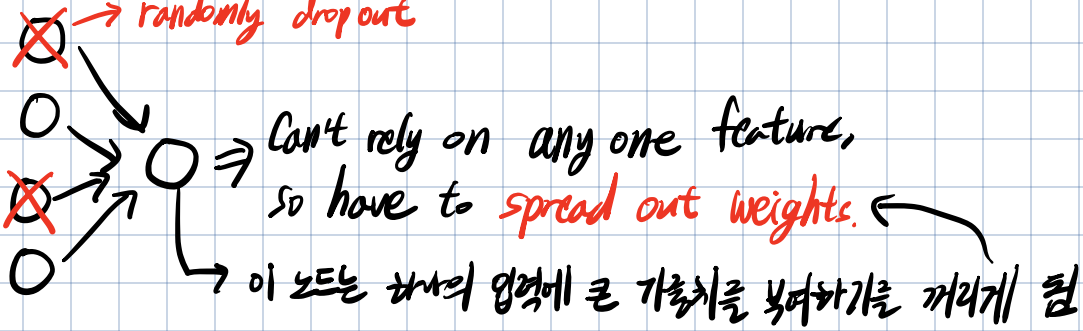
↓

$\hat{y}$

test에서 drop-out을 구현하지 않아도 크기는 변하지 않기 때문에 /= keep_prob scaling을 할 필요 X

• Understanding dropout

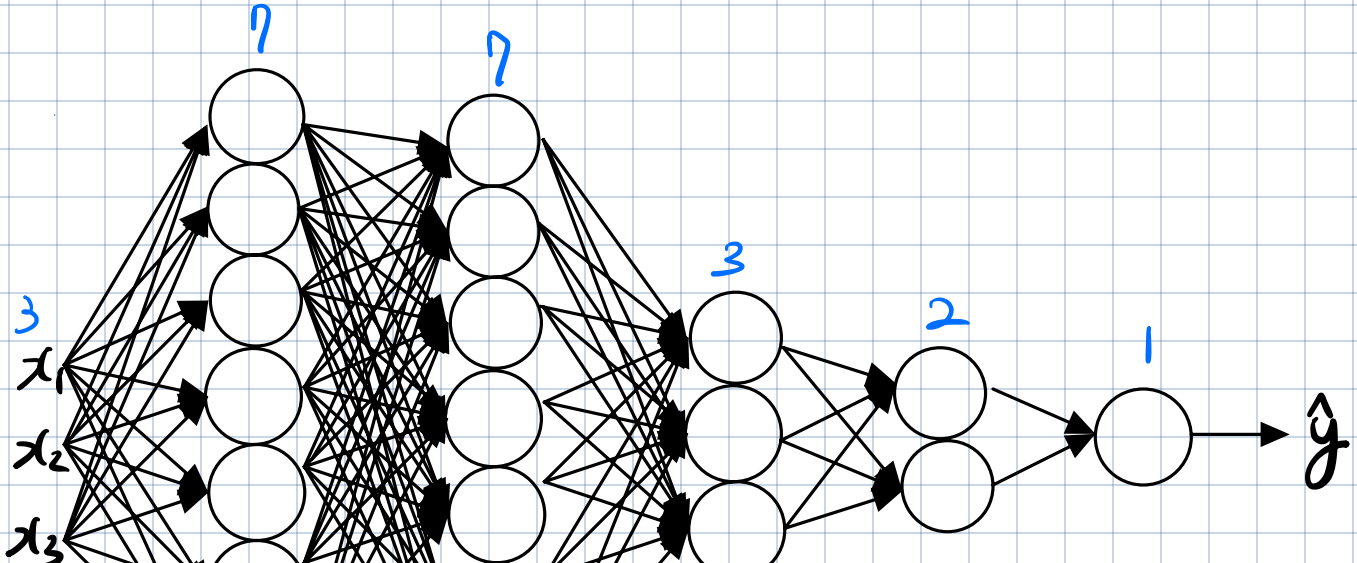
— Why does drop-out Work?

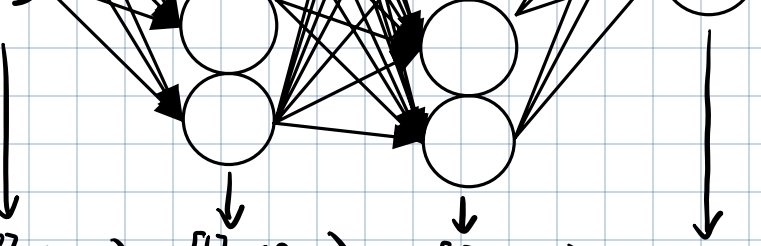


⇒ Can't rely on any one feature, so have to spread out weights. ← 이 노드는 하나의 입력에 큰 가중치를 부여하기를 꺼리게 됨

⇒ weight를 spread-out 함으로써 $\|w^{[2]}\|^2$이 줄어든다. ← L2 정규화처럼 가중치를 줄여 overfitting ⊗

— keep-prob을 각 층에 다르게 설정할 수 있다. (but, more hyperparameter for CV)





keep-prob ↑ ⇒ Regularization effect ↓

Size : $W^{[0]} = (3,7)$ $W^{[1]} = (7,3)$ $W^{[2]} = (7,7)$ $W^{[3]} = (3,7)$

keep-prob : 1.0 (input 층에서는 dropout 안하거나 1에 가깝게 설정)

0.7 (the biggest parameter matrix ⇒ keep-prob가 상대적으로 더 낮은 keep-prob로 설정 발아야 함 (powerful dropout)) 0.5

0.7 (다대다형의 우려가 작은 층에서는) 1.0 1.0 (not using dropout)

- Computer vision에서 drop-out 구현을 많이 한다.

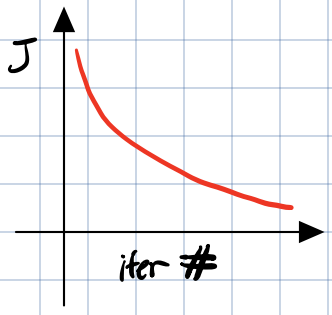
  ∵ 아주 많은 pixel 값 사용 → no enough data

  ⇒ 하지만 이 분야 외에서는 over-fitting 전에는 drop-out 사용 X

- Downside of dropout

  ⇒ 반복마다 무작위로 노드 일부를 제거하기 때문에 cost function J의 하강을 확인하기 어려움 (따라서 디버깅 어려움)

  (dropout을 디 설정하여 drop out을 끄고 J를 확인 후 다시 turn-on)



- Other regularization methods

① Data augmentation

- 데이터를 다양하게 변형 / 편집하여 새로운 데이터 얻기

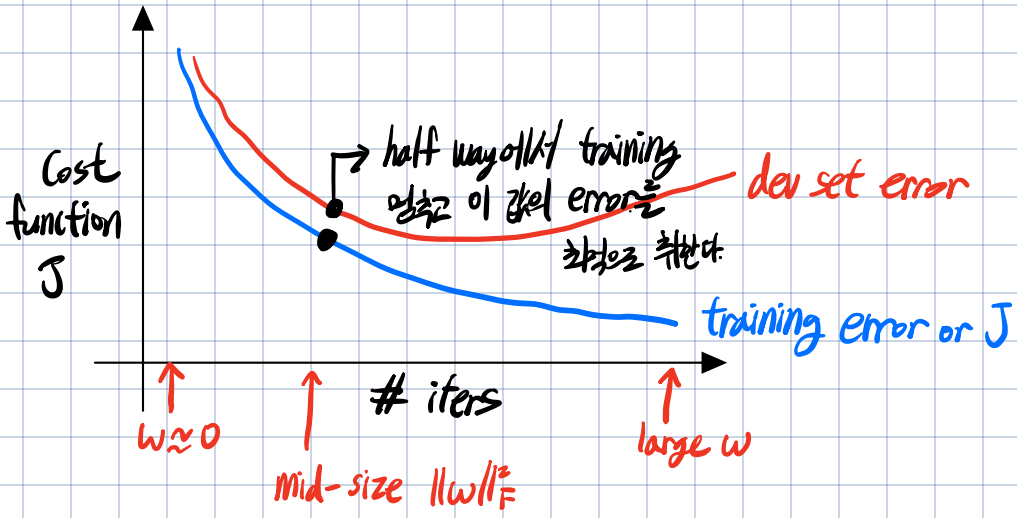
  e.g. 이미지의 회전, 확대, 편집

  ⇒ 중복된 사항이 많아져 좋지 않을 수 있으며, 독창적인 새로운 이미지를 구하는 것보다는 bad.

② Early Stopping

- Early stopping은 training 중 최소의 loss가 나오면 stop

⇒ L2 정규화와 비슷하게 w norm이 작은 상황을 선택함으로써 cost function을 optimize한다.



- L2 regularization vs early stopping

	L2	Early Stopping
Good	모델을 따로 훈련할 수 있어 J최적화 가능	한번의 경사하강법으로 small w, mid w, big w를 얻을 수 있어 많은 값을 시도해볼 필요 없다.
Bad	λ에 많은 값 시도 필요 ⇒ computationally expensive	J 최적화를 망치게 됨 (orthogonalization X) ⇒ ①,②를 독립적으로 수행 못함

Orthogonalization
- ① optimize J : Gradient descent, momentum ... ⇒ optimize w & b
- ② Avoid overfitting : augmentation, getting more data, ... ⇒ reduce variance

이 둘은 완전히 다르다

→ 이 둘을 한 번에

think one thing at a time ⇒ but early stopping mix these two things up

⇒ 컴퓨터 감당 가능하면 L2 Regularization 이용하는 것을 추천

3. Setting up your optimization problem

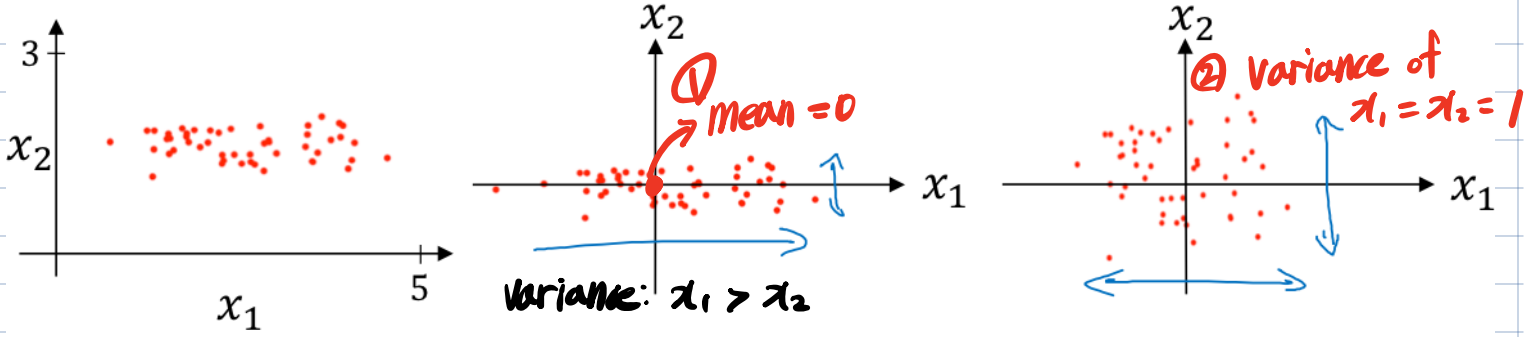
- Normalizing inputs

- How to speed up training? ⇒ Normalize the inputs!

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

① 모든 훈련 샘플에 대해 mean 빼기 (zero-out mean)

② 모든 훈련 샘플에 대해 variance 나누기



Use same μ & σ^2 to normalize your test set

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

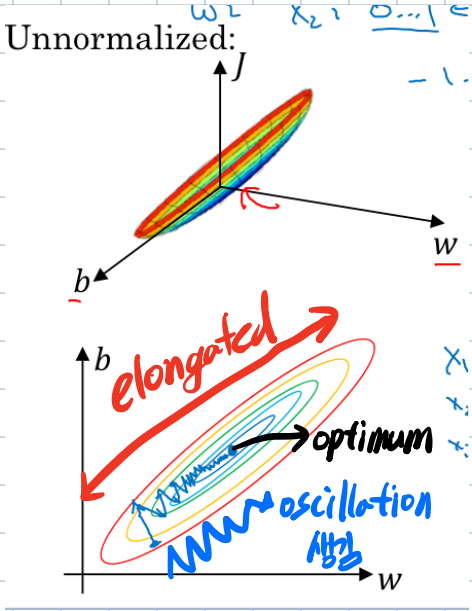
② $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)**2}$ (element-wise)

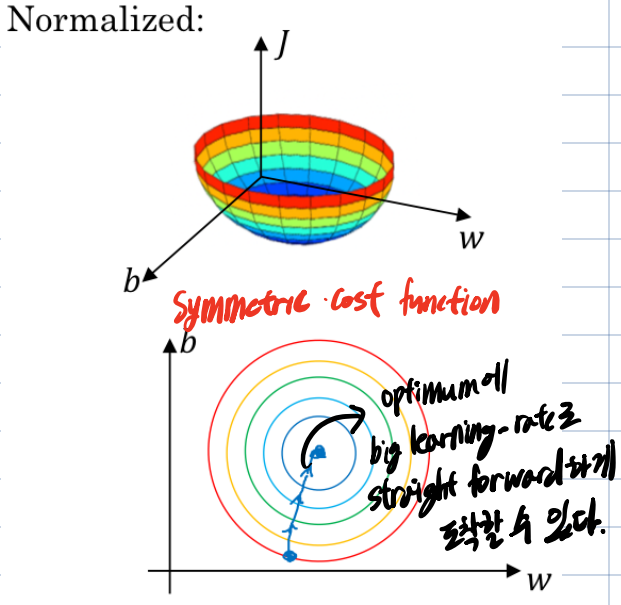
$x := x - \mu$

$x /= \sigma^2$

- Why do we normalize input features?

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

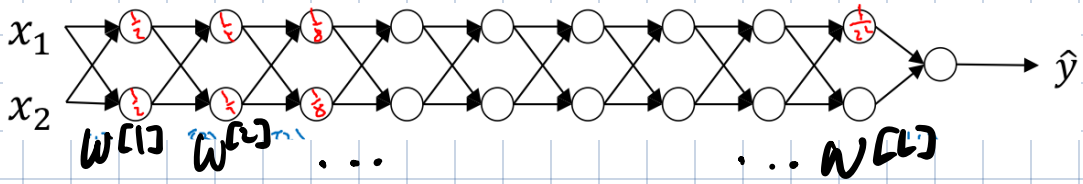




⇒ feature 간 scale이 너무 다르면 w_1 & w_2의 비가 너무 달라져 elongated cost function을 갖게 되기에 매우 작은 learning_rate로 gradient descent 수행 (시간 오래 걸림)

• Vanishing / exploding gradients

— 굉장히 deep한 network 훈련 시 derivative가 기하급수적으로 작아진다.



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

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

$z^{[1]} = w^{[1]} x$

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

$a^{[2]} = g(z^{[2]}) = g(w^{[2]} a^{[1]})$

⇒ $\hat{y}$은 모든 w 행렬의 곱이 됨

if $w^{[l]} > I$ ⇒ Deep NN makes activations explode

e.g. $w^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$ ⇒ $\hat{y} = w^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x$

$= 1.5^{(L-1)} x$

if $w^{[l]} < I$ ⇒ Deep NN makes activations very small

e.g. $w^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ ⇒ $\hat{y} = 0.5^{(L-1)} x$

⇒ you can apply this intuitive to derivative explode / vanish

• Weight initialization for deep networks

- Partial solution for gradient vanishing / exploding

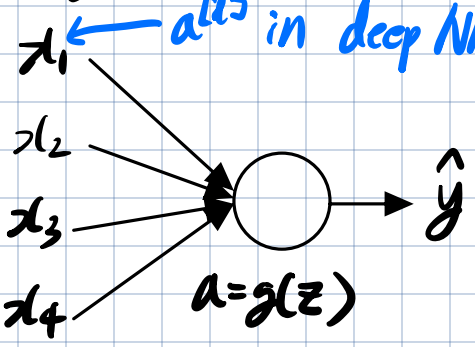
⇒ better choice of random initialization for NN

⇒ Good initialization leads to...

① Speeding up the convergence of gradient descent

② Increase the odds of gradient descent converging to a lower training & generalization error

- Single neuron example



# of input feature

$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$ (b → 0)

⇒ to make z not too big / small...

∴ larger n, smaller w_i

① Set $Var(w_i) = \frac{1}{n}$

layer l에 들어가는 # of input features

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

② if activation func = ReLU, set $Var(x) = \frac{2}{n}$

$w^{[l]} = np.random.randn(shape) * np.sqrt\left(\frac{2}{n^{[l-1]}}\right)$

⇒ 각각의 w를 1보다 너무 크거나 작아지지 않게 설정하여 소실 / 폭발 ⊗

③ Other activation Variants

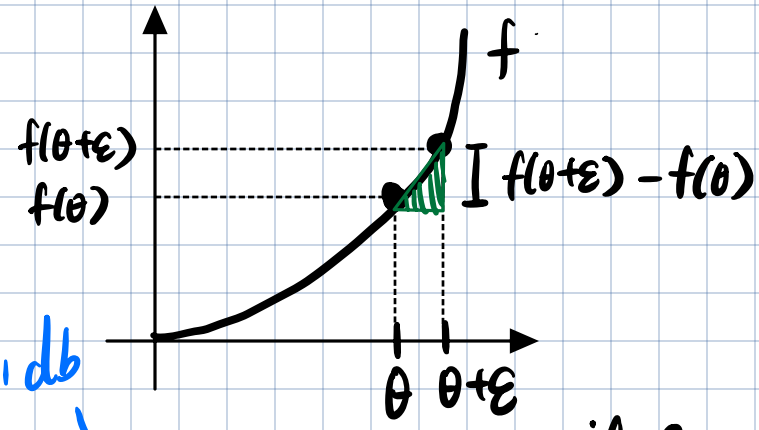
tanh : $\sqrt{\frac{1}{n^{[l-1]}}}$ or $\sqrt{\frac{2}{n^{[l-1]}}}$

⇒ Xavier initialization

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

⇒ hyperparameter 탐색의 우선순위로 고려할 수준 정도 가능

- **Numerical approximation of gradients**
  - Check implementation of backprop is correct
  - One-side difference



dw, db

$$g(\theta) = \frac{d}{d\theta} f(\theta) = f'(\theta)$$

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

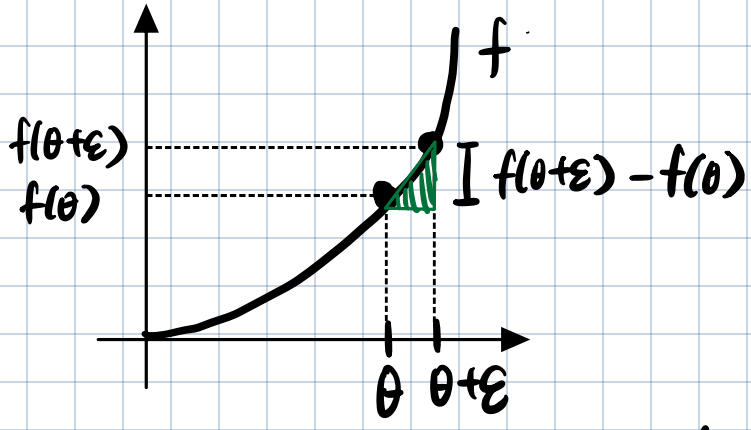
J

$f(\theta) = \theta^3, \ \theta \in \mathbb{R}$

if $\theta = 1, \ \varepsilon = 0.01,$

$$\Rightarrow \begin{cases} g(\theta) = 3 \cdot (1)^2 = 3 \\ \dfrac{(1.01)^3 - 1^3}{0.01} = 3.0301 \end{cases} \quad 0.0301$$

  - two-side difference



$$g(\theta) = \frac{d}{d\theta} f(\theta) = f'(\theta)$$

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

J

$f(\theta) = \theta^3, \ \theta \in \mathbb{R}$

if $\theta = 1, \ \varepsilon = 0.01,$

$$\Rightarrow \begin{cases} g(\theta) = 3 \cdot (1)^2 = 3 \\ \dfrac{(1.01)^3 - (0.99)^3}{0.01} = 3.0001 \approx 3 \end{cases} \quad 0.0001$$

⇒ two-side difference가 one-side보다 $g(\theta)$이 $f(\theta)$의 derivative에 대한 correct implementation 더욱 confidence를 준다.

• Gradient Checking

- Check if you implemented backprop well for NN

① $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ 합쳐 reshape하여 하나의 큰 θ vector를 만든다. (↳ concatenate)

⇒ Cost function J 또한 w, b에 대한 함수가 아닌 θ에 대한 함수로

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

② $dW^{[1]}, db^{[1]}, \cdots, dW^{[L]}, db^{[L]}$를 합쳐 reshape ⇒ $d\theta$ vector

③ Check if $d\theta$ the gradient of $J(\theta)$?

now, $J(\theta) = J(\theta_1, \theta_2, \theta_3, \cdots)$ → θ 모든 각각에 대해 check

for each i use two-side difference

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

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

↳ check by euclidean distance

$$\text{set } \varepsilon = 10^{-7}, \quad \frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx \#$$

- 10^{-9} ⇒ great
- 10^{-5} ⇒ okay, but check that none of the components of vector are too large

10^{-3} ⇒ Worry, 특정 i에서 $d\theta_{approx}[i]$와 $d\theta[i]$의 차이가 크게 나는 곳을 추적할 필요 있음.

- **Gradient checking implementation notes**

① Debugging에만 사용하고 training에는 사용 X

∵ expensive computation

② algorithm이 grad check에 실패 ⇒ Component 관찰

⇒ 특정 l에서 어떤 $db^{[l]}$과 $dw^{[l]}$에서 θ와 멀어지는지 check

③ Regularization이 J에 포함되어 있다는 것을 기억하기

$$J(\theta) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_l ||w^{[l]}||^2$$

$d\theta$ = grad of J w.r.t θ that contains 정규화항

④ Dropout하고 반복에는 쓰지 말 것.

∵ Dropout은 node의 subset을 없애기 때문에 계산이 어렵다.
keep-prob = 1로 설정해서 turn-off 후 grad check

⑤ random initialization 할 (몇번 training 할에) 실행

∵ $w, b \approx 0$ in random initialization ⇒ network를 깊게 훈련하여 w & b가 0에서 멀어질 시간을 줌