M2177.003100
Deep Learning

[6: The Art of Neural Net Training]

Electrical and Computer Engineering
Seoul National University

(last compiled at 17:58:00 on 2018/10/02)

Outline

References

- *Deep Learning* by Goodfellow, Bengio and Courville ▸ Link
  - Chapter 8: Optimization for Training Deep Models
  - Chapter 11: Practical Methodology
  - Chapter 15: Representation Learning
- online resources:
  - *Deep Learning Specialization (coursera)* ▸ Link
  - *Stanford CS231n: CNN for Visual Recognition* ▸ Link
  - *Machine Learning Yearning* ▸ Link

Outline

Practical considerations

- successfully applying deep learning
  - requires more than knowing algorithms
- also need to know how to
  - choose an algorithm for a particular application
  - monitor and respond to feedback obtained from _____
- neural net training: so important/so expensive $\Rightarrow$ need careful tactics
  1. whether to gather more data
  2. increase/decrease model capacity
  3. add/remove regularization
  4. improve optimization
  5. improve approximate inference
  6. debug software implementation

Example: optimization

- a reasonable choice:
  - sgd with momentum with a decaying learning rate
  - popular decay schemes:
    - decay linearly to a fixed minimum
    - decay exponentially
    - decrease by a factor of 2–10 each time validation error plateaus
- reasonable alternative: _____
- batch normalization
  - can have a dramatic effect on optimization performance
  - can be omitted from the first baseline
  - but should be introduced quickly if optimization appears problematic

Example: regularization

- include regularization from the start
  - unless training set contains $+10$ million examples
- early stopping
  - should be used almost universally
- dropout: an excellent regularizer
  - easy to implement
  - compatible with many models and training algorithms
- _____
  - can reduce generalization error and allow dropout to be omitted[1]

[1] due to noise in estimate of statistics used to normalize each variable

Neural net training (iterative)

- phase 1: setup
  - data preprocessing
  - weight initialization
  - regularization
- phase 2: monitor training
  - covariate shift and normalization
  - parameter updates (optimization)
  - hyperparameter tuning
- phase 3: evaluate and improve
  - model ensembles
  - data augmentation, transfer learning
  - debugging

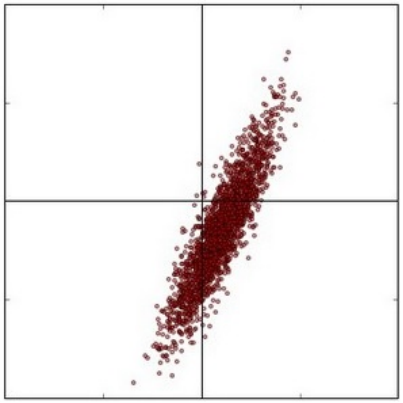
✳ **regularization**/**optimization**: will be covered separately

Outline

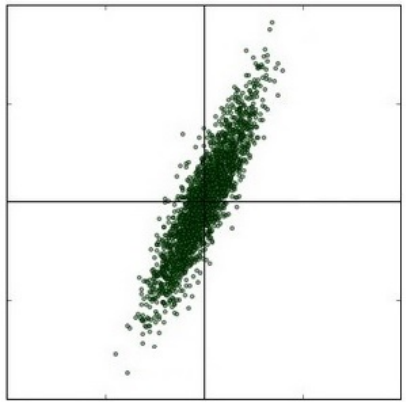
Data preprocessing

- common types
  - zero-centering
  - normalization
  - decorrelation
  - whitening

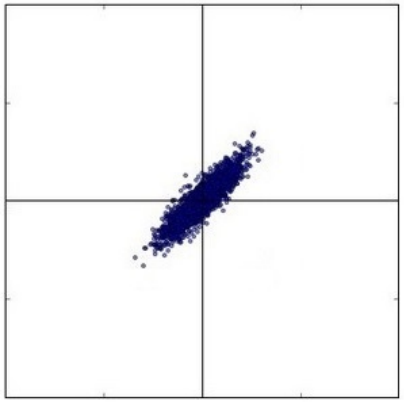
original



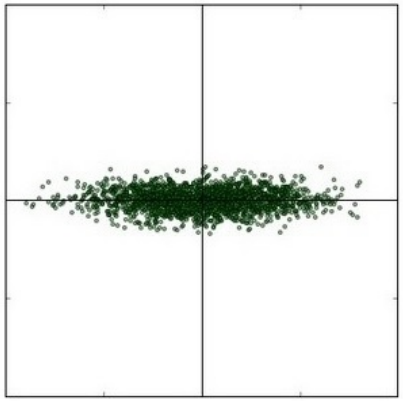
zero-centered



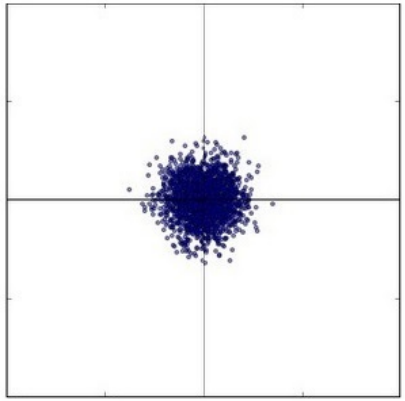
normalized



decorrelated

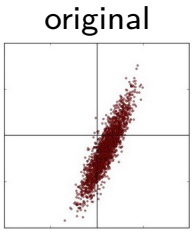


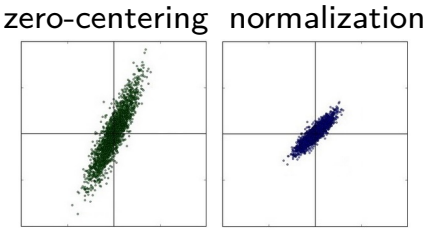
whitened

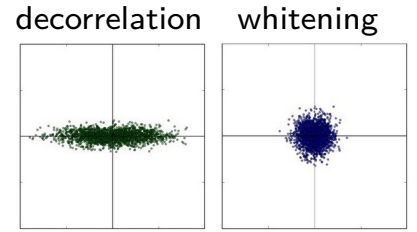


(source: cs231n)

- zero-centering
  - mean subtraction
- normalization
  - mean subtraction + division by standard deviation
- decorrelation
  - rotate data by the direction given by PCA
  - diagonal covariance matrix
- whitening
  - decorrelation + normalization
  - covariance matrix = identity matrix







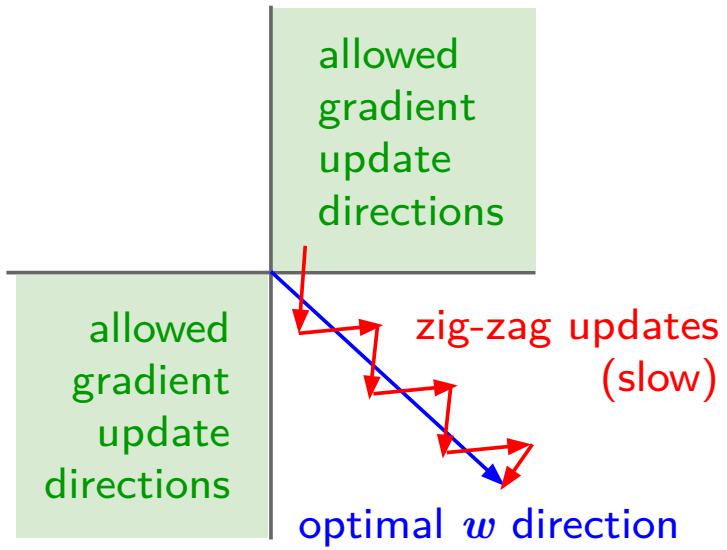
⁂ image data: typically _____ only

e.g. subtract per-image mean (AlexNet) or per-channel mean (VGGNet)

Why do preprocessing?

- can speed up training

1. importance of zero-centering
   - recall: all positive/negative inputs
     - ▷ cause _____ updates
     - $\Rightarrow$ slow convergence



(source: cs231n)

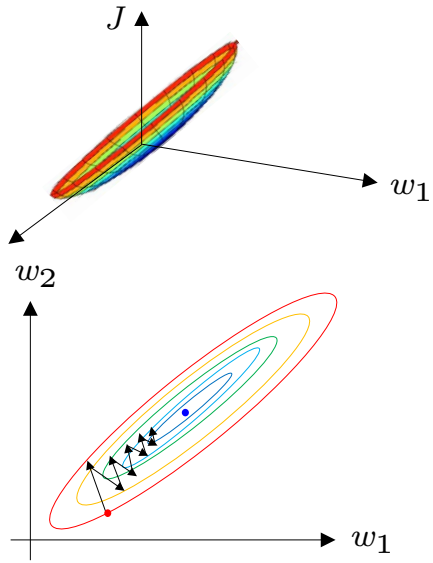
2. importance of normalization

unnormalized data

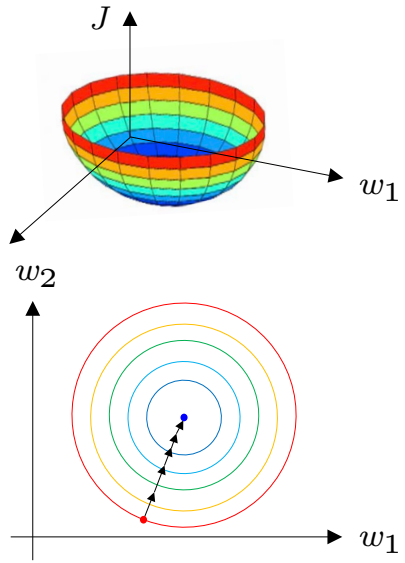
- may show wiggly behavior

$\Rightarrow$ difficult to optimize

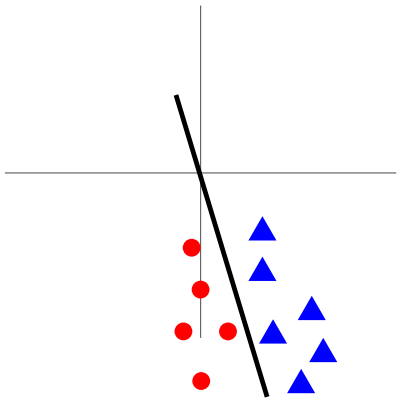
unnormalized



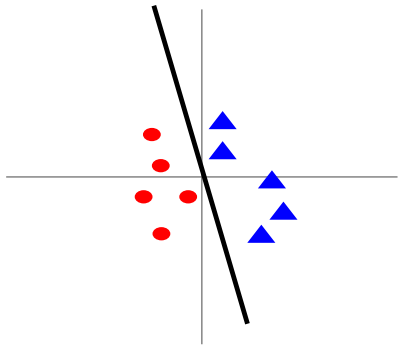
normalized



unnormalized



normalized



(source: cs231n, coursera)

- before normalization
  - small change in weights

  $\Rightarrow$ big _____ in boundary
- after normalization
  - boundary less sensitive

  $\Rightarrow$ easier to optimize

Caution

- any preprocessing statistics (*e.g.* data mean)
  - must only be computed on training data
  - and then applied to dev/test data
- common pitfall:
  - compute mean and subtract it from every image across _____ dataset
  - and then split the data into train/dev/test splits
- correct way:
  - compute the mean only over training data
  - and then subtract it equally from all splits (train/dev/test)

Outline

Weight initialization

- reasonable assumption about final weight values
  - roughly half positive + half negative

  $\Rightarrow$ zero weight: "best guess" in expectation

- attempt #1: all zero initialization (don't do this)
  - every neuron computes the same output/gradient/weight update
  - no source of asymmetry between neurons

- for symmetry breaking
  - zero mean is fine but we need some _____ in weights

- common idea
  - initialize weights using zero mean Gaussian with some variance

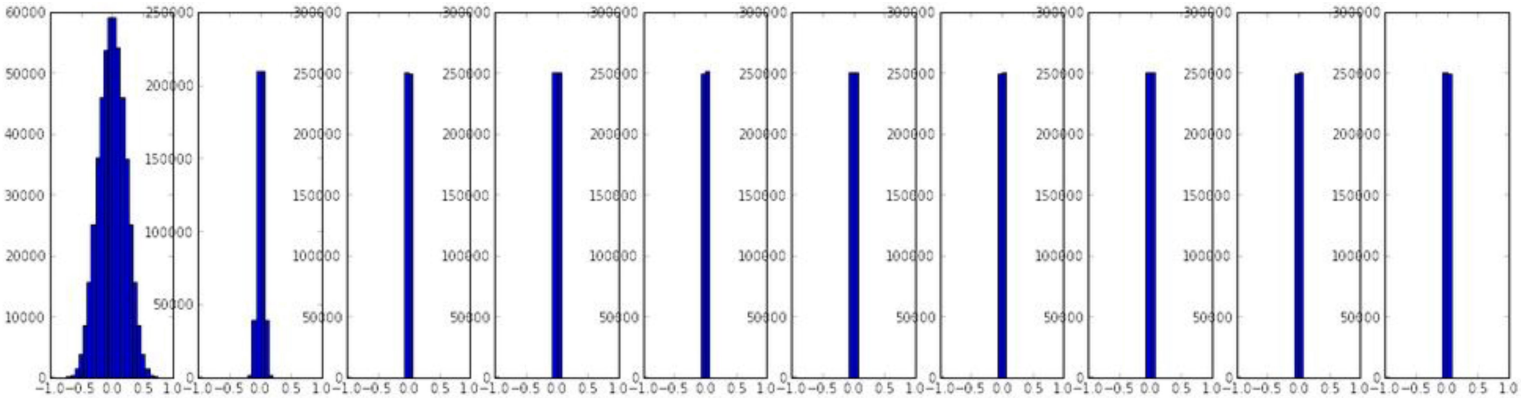
- attempt #2: initialization with small random numbers

  e.g. gaussian with zero mean and 0.01 std:

```
W = np.random.randn(fan_in, fan_out) * 0.01
```

  - works okay for small nets but problems with **deeper nets**

  e.g. activation distribution (10 layers, 500 units per layer, tanh nonlinearity):



  $\Rightarrow$ all activations rapidly become zero

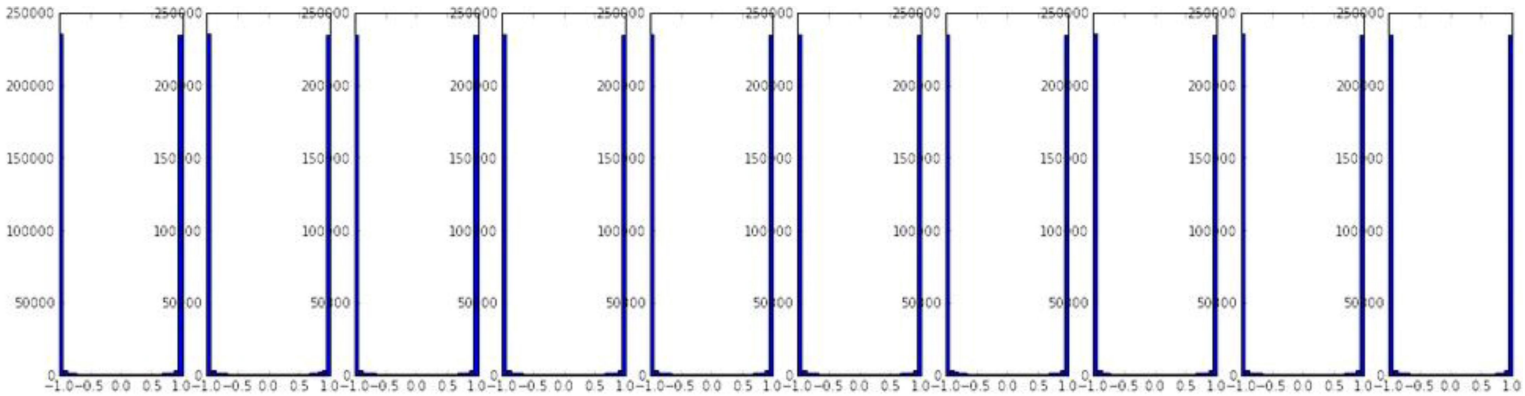
  - backward pass for $\boldsymbol{W}\boldsymbol{a}$ gate: $\frac{\partial J}{\partial \boldsymbol{W}} \propto \boldsymbol{a} \Rightarrow$ _____ killed

- attempt #3: initialization with larger random numbers

  e.g. gaussian with zero mean and some variance:

```
W = np.random.randn(fan_in, fan_out) * 2
```

  e.g. activation distribution in the same net as before



  - almost all neurons get _____ $(\pm 1) \Rightarrow$ gradients killed

- observation:
  - need initial weights that are just right (not too small, not too big)
  - Xavier/He initializations address this

Adjusting variance of initial weights

- assume a simple net: $y = \sum_{i=1}^{n} w_i x_i$
  - we don't want to make y too big/small (not to saturate)
  - to this end, need to make w_i _____ as n grows

- intuitively: setting $\mathrm{Var}(w) = \frac{1}{n}$ will do $\leftarrow n$: # inputs ("fan in")

- mathematically:
  - assume we want $\mathrm{Var}(y) = \mathrm{Var}(x)$

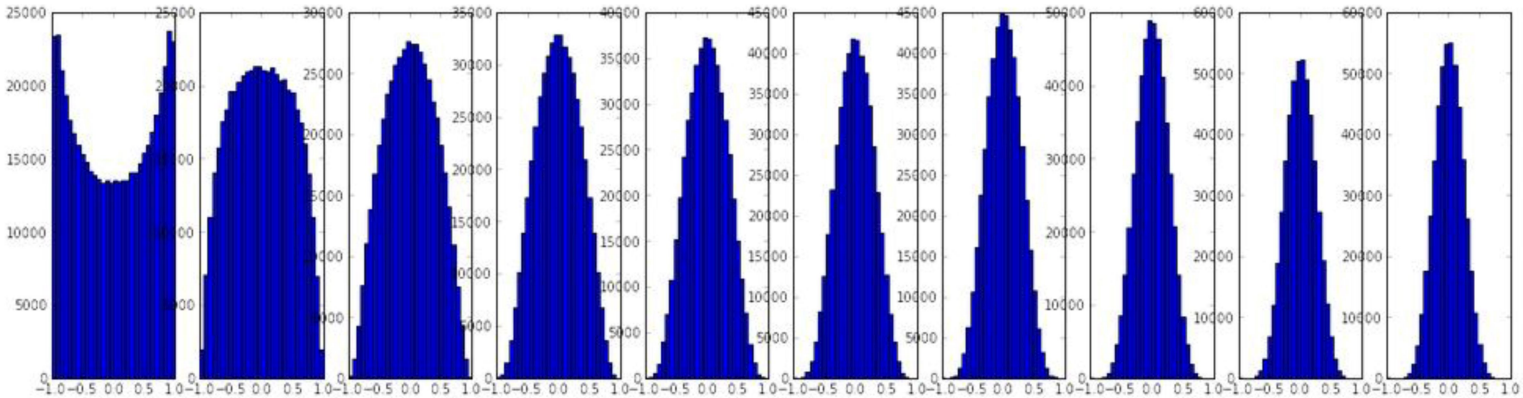
$$\begin{aligned}
\mathrm{Var}(y) &= \sum_{i=1}^{n} \mathrm{Var}(w_i x_i) \\
&= \sum_{i=1}^{n} \underbrace{[\mathbb{E}(w_i)]^2}_{=0} \mathrm{Var}(x_i) + \underbrace{[\mathbb{E}(x_i)]^2}_{=0} \mathrm{Var}(w_i) + \mathrm{Var}(w_i)\mathrm{Var}(x_i) \\
&= \sum_{i=1}^{n} \mathrm{Var}(w_i)\mathrm{Var}(x_i) = n\mathrm{Var}(w)\mathrm{Var}(x)
\end{aligned}$$

  - setting $\mathrm{Var}(w) = \frac{1}{n}$ will do

Xavier and He initialization

- Xavier initialization
  - reasonable choice for _____ type nonlinearity

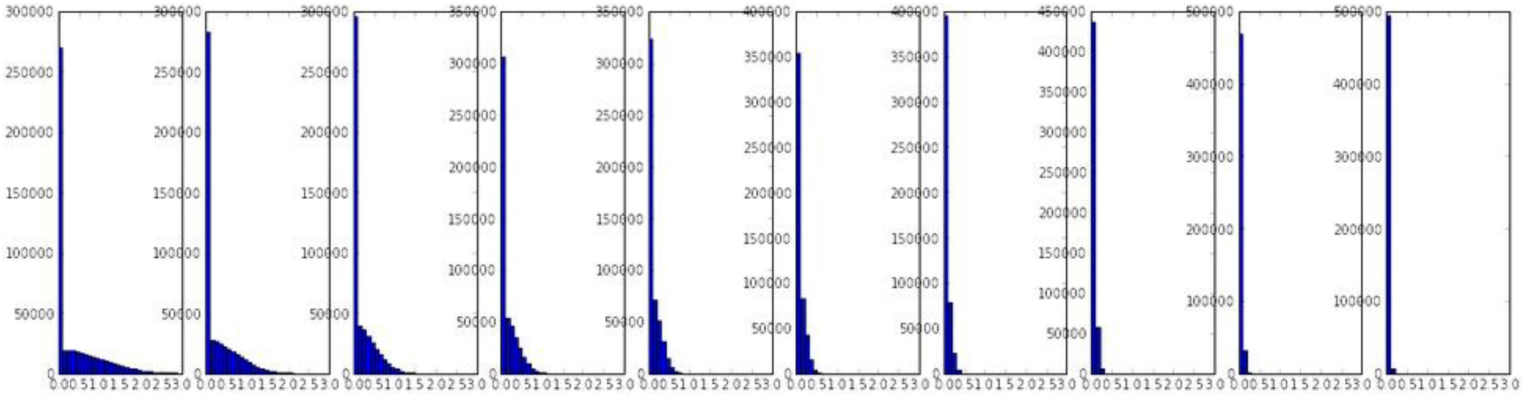
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```



  - variant (considers backprop direction)

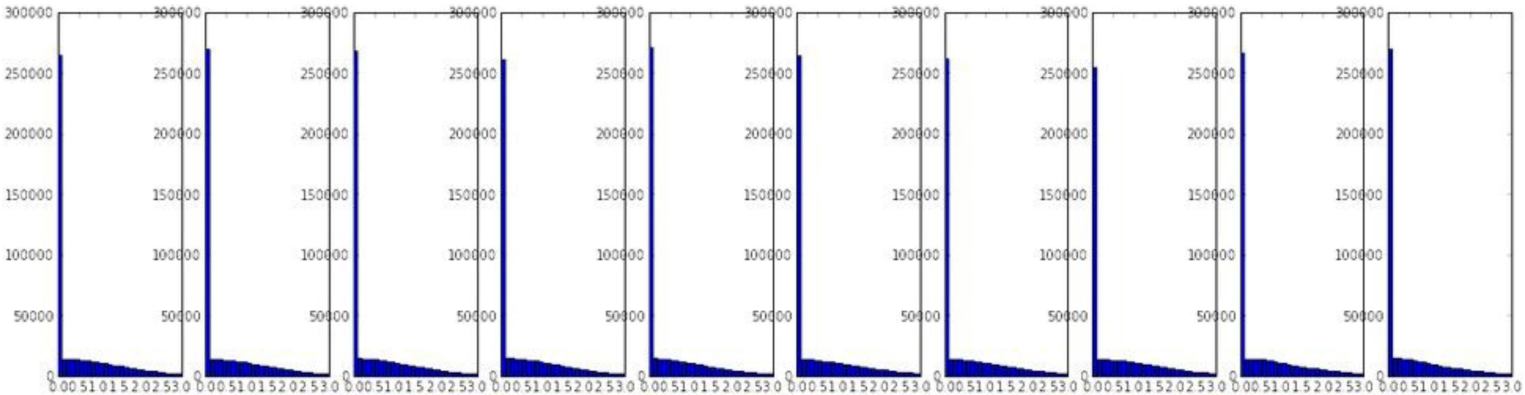
```
W = np.random.randn(fan_in, fan_out) / np.sqrt((fan_in + fan_out)/2)
```

- Xavier initialization: often does not work well for ReLU

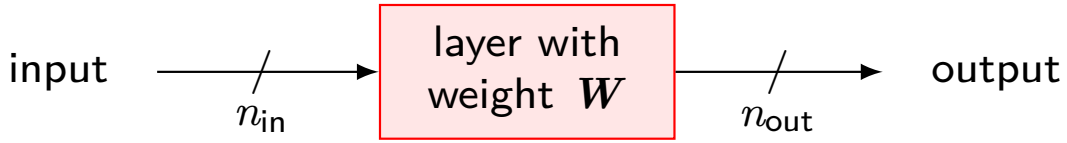


- He initialization: works well with _____ (current recommendation)

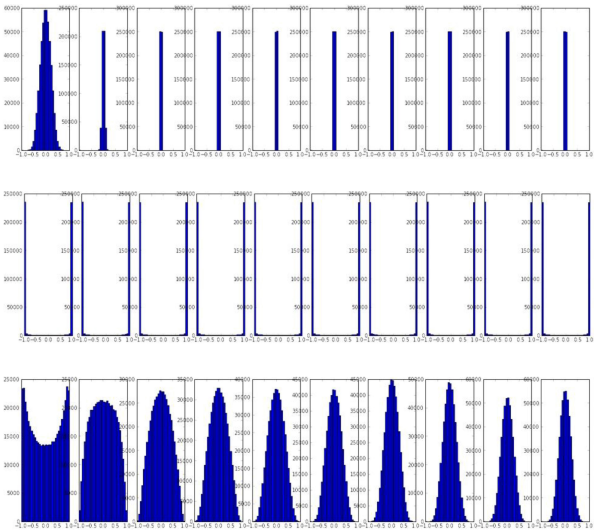
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2)
```



- comparison:



method	$\mathrm{Var}(w)$	use with	code
Xavier	$\frac{1}{n_{\text{in}}}$	sigmoid	`W = np.random.randn(n_in, n_out)/np.sqrt(n_in)`
Xavier	$\frac{2}{n_{\text{in}} + n_{\text{out}}}$	sigmoid	`W = np.random.randn(n_in, n_out)/np.sqrt((n_in + n_out)/2)`
He	$\frac{2}{n_{\text{in}}}$	ReLU	`W = np.random.randn(n_in, n_out)/np.sqrt(n_in/2)`



- initialization too small $\Rightarrow$ no learning
  - activations/gradients go to zero
- initialization too big $\Rightarrow$ no learning
  - activations saturate (tanh)/gradients to zero
- initialization just right $\Rightarrow$ learning proceeds nicely
  - nice distribution of _____ at all layers

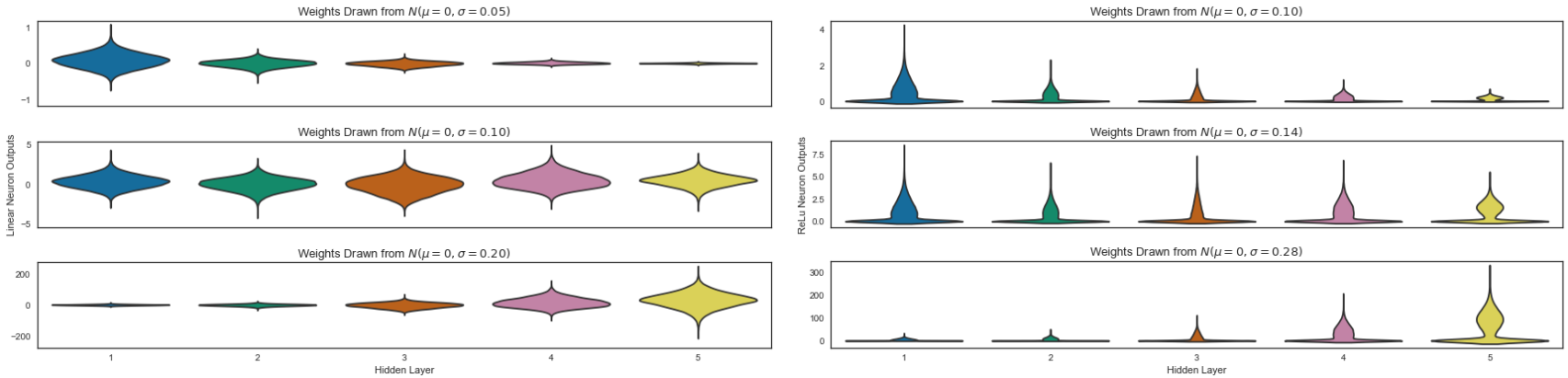
Additional techniques

- sparse initialization: another way to break symmetry
  - weight matrix is initialized to a sparse matrix

  e.g. mostly zero but some elements sampled from a small gaussian

- bias initialization
  - biases: commonly set to ____
  - rationale: adjusting weights is sufficient for symmetry breaking

- proper initialization: active area of research!



(source: https://intoli.com/blog/neural-network-initialization/)

Alternative idea

- what we've done so far:
  - adjust initial weights $\rightarrow$ activations can have a nice distribution
- how about
  - directly making activations have a nice distribution?

  ↑ *e.g.* zero mean, unit variance Gaussian
- this is the idea behind _____ normalization

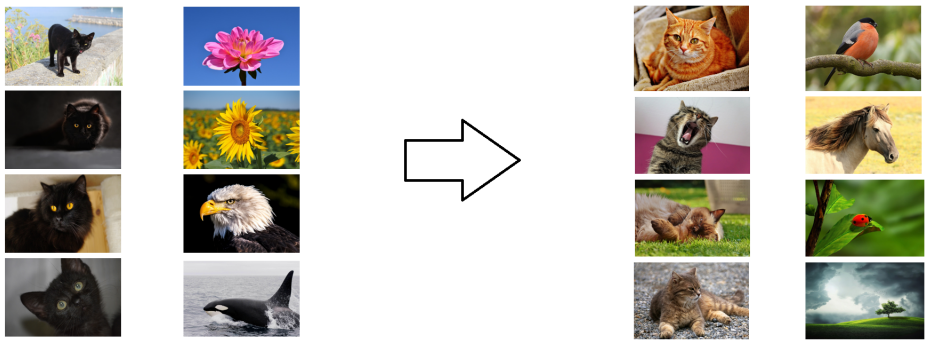
Outline

Covariate shift

- synonyms
  - input variable, independent variable, predictor variable, regressor, covariate, controlled variable, manipulated variable, explanatory variable, exposure variable (reliability theory), risk factor (medical statistics), feature (machine learning), control variable (econometrics)

- covariate shift
  - change of the _____ distribution to a learning system

  e.g. training: web images → test: smartphone images

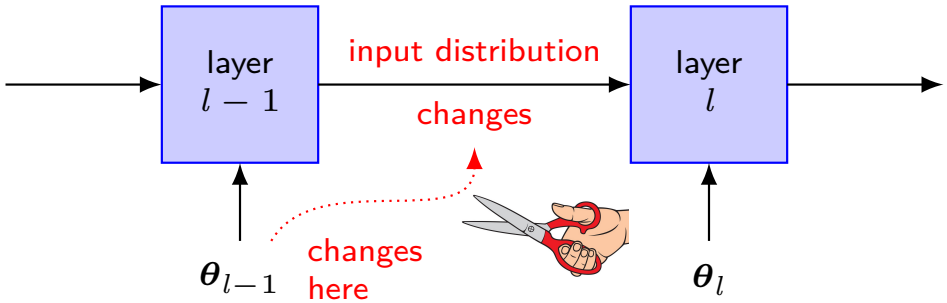


(source: coursera)

  - cure: domain adaptation

Internal covariate shift

- distribution of each layer's inputs changes during training
  - as parameters of previous layers change



- internal covariate shift complicates NN training:
  - slows down training by requiring
    - lower learning rates
    - careful parameter _____
  - makes it hard to train models with saturating nonlinearities

- cure: cut the interaction $\rightarrow$ how?

Whitening

- normalization (zero mean, unit variance) + decorrelation
- has been long known: if inputs are whitened
  - network training converges faster (LeCun et al., 1998)
- likewise, how about whitening each layer inputs?
  - their distributions: always fixed (zero mean, unit variance Gaussian)

  $\Rightarrow$ reduced internal covariate shift $\Rightarrow$ faster training
- but full whitening of each layers' inputs
  - computationally expensive (covariance matrix and its inverse)
  - not everywhere differentiable
- simplified alternative: batch normalization
  - _____-wise normalization (*i.e.* no decorrelation) of layer inputs

Outline

Batch normalization (Ioffe and Szegedy, 2015)

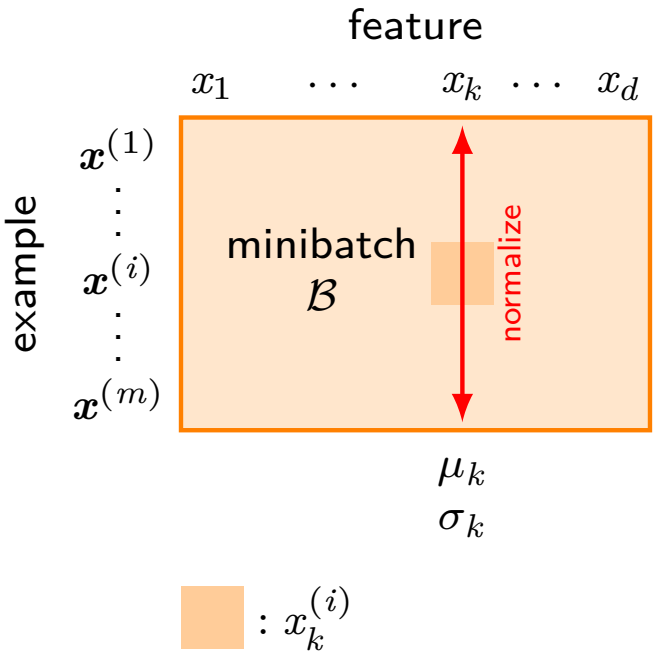
- alleviates many issues with properly initializing neural nets
  - very popular these days
- idea: explicitly forces the activations throughout a net
  - to take on a unit gaussian distribution at the beginning of training
- core observation: this is possible because
  - normalization is a simple _____ operation

- implementation: create and insert a BN layer
  - FC/conv layer $\rightarrow$ BN layer $\rightarrow$ nonlinearity layer
- interpretation
  - $\underbrace{\text{BN}}_{\uparrow}$ = doing preprocessing at every layer of the net

    but integrated into the net itself in a differentiable manner

BN operation #1: normalization

- normalize each scalar feature independently (not whitening features jointly)[2]
  - speeds up convergence (even without _____)



- for feature k
  - minibatch mean

$$\mu_k = \frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}$$

  - minibatch variance

$$\sigma_k = \frac{1}{m}\sum_{i=1}^{m}(x_k^{(i)} - \mu_k)^2$$

  - normalized $x_k^{(i)}$

$$\widehat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

[2] for consistency, the notations used here differ from those in the original paper

BN operation #2: linear transform

- simple normalization of each input of a layer: problematic
  - it may change what the layer can represent

    i.e. will reduce the representation power of the layer
  - hidden units will always have zero mean and unit variance

    e.g. sigmoid inputs: will be put in linear regime of nonlinearity

- solution: introduce two parameters γ_k, β_k for each activation $x_k^{(i)}$

$$y_k^{(i)} = \gamma_k \widehat{x}_k^{(i)} + \beta_k \equiv \mathrm{BN}_{\gamma_k,\beta_k}(x_k^{(i)})$$

- these parameters
  - _____ along with original model parameters by sgd
  - restore the representation power of the network[3]

[3] *e.g.* by setting $\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$, we can recover original activations (if that were the optimal thing to do)

Batch norm operations altogether

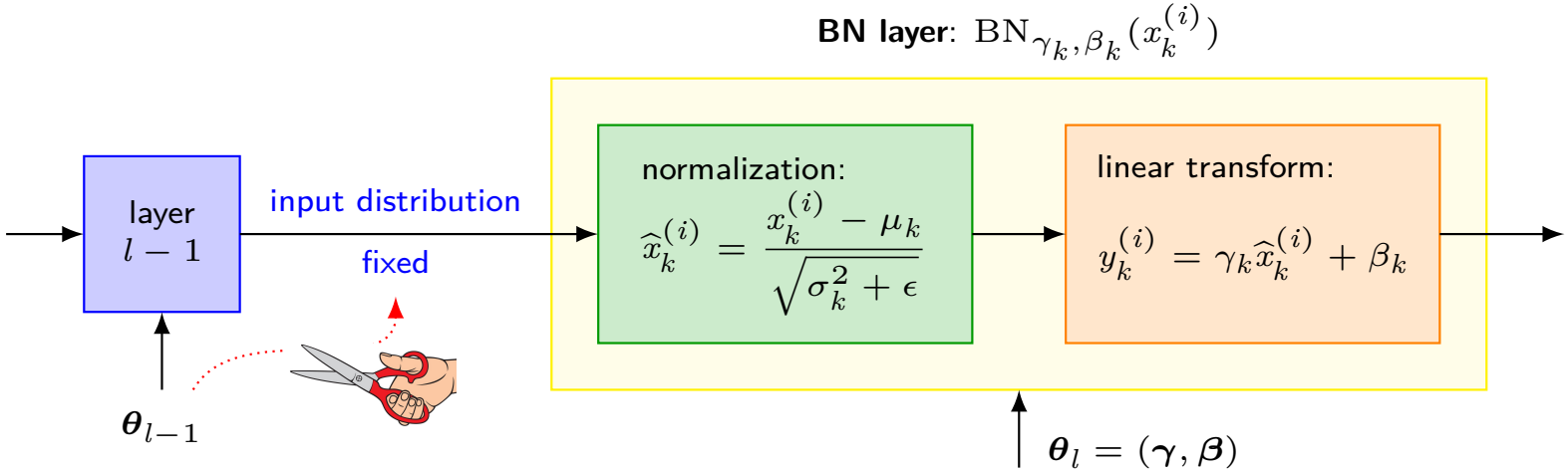
- op #1: normalization
  - to fix the distribution of inputs to each layer

  $\Rightarrow$ cut high-order interactions between different layers

  $\Rightarrow$ _____ learning dynamics $\Rightarrow$ speed up training

- op #2: linear transform
  - to restore representation power of the network

    (don't get confused why this is needed)



BN at test time

- at test time BN layer functions differently:
  - the mean/std are not computed based on the batch
  - instead, a single ____ empirical mean of activations during training is used
  - this is to get a deterministic result for each test example

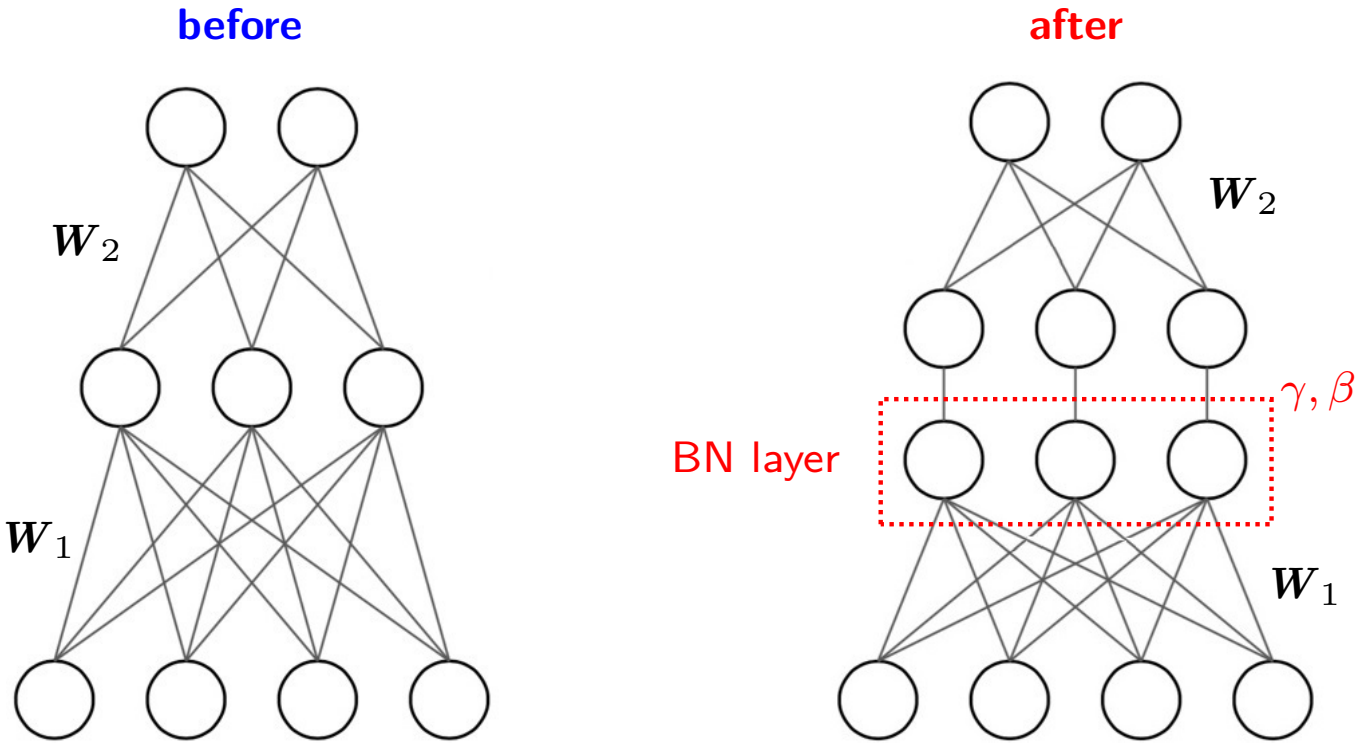
- for example: μ and σ can be estimated
  - using exponentially weighted moving average

↑ collected during training time across mini-batches

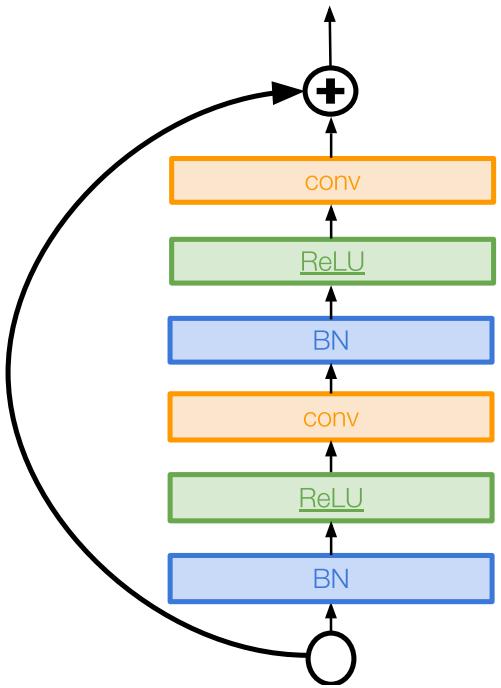
minibatches:	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$	$\cdots$		
	$\downarrow$	$\downarrow$	$\downarrow$			
mean series:	$\mu^{(\mathcal{B}_1)}$	$\mu^{(\mathcal{B}_2)}$	$\mu^{(\mathcal{B}_3)}$	$\cdots$	$\longrightarrow$	μ for test
variance series:	$\sigma^{(\mathcal{B}_1)}$	$\sigma^{(\mathcal{B}_2)}$	$\sigma^{(\mathcal{B}_3)}$	$\cdots$	$\longrightarrow$	σ for test

Implementation

- inserted to a net as "BN layer"
  - learnable parameters: ____



- usually inserted
  - _____ linear layers (FC, conv) and before nonlinearity (ReLU)
- parameters of BN layer
  - after FC layer: one pair of (γ, β) per neuron (per activation)
  - after conv layer: one pair of (γ, β) per feature map
- example: ResNet block



(source: He et al.)

BN layer summary

- input
  - $\boldsymbol{X}$: $\underbrace{m}_{\text{batch size}} \times \underbrace{d}_{\#\text{ of features}}$
- learnable parameters
  - $\boldsymbol{\gamma}, \boldsymbol{\beta} : 1 \times d$
- intermediates
  - $\boldsymbol{\mu}, \boldsymbol{\sigma} : 1 \times d$
  - $\widehat{\boldsymbol{X}} : m \times d$
- output
  - $\boldsymbol{Y} : m \times d$

- training time

$$\mu_k = \frac{1}{m}\sum_{i=1}^{m} x_k^{(i)}$$

$$\sigma_k = \frac{1}{m}\sum_{i=1}^{m}(x_k^{(i)} - \mu_k)^2$$

$$\widehat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

$$y_k^{(i)} = \gamma_k \widehat{x}_k^{(i)} + \beta_k$$

- test time

$\mu_k =$ (running) average from training

$\sigma_k =$ (running) average from training

$$\widehat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

$$y_k^{(i)} = \gamma_k \widehat{x}_k^{(i)} + \beta_k$$

Benefits of BN

- allows higher learning rates $\Rightarrow$ accelerated training
  - why? less covariate shift, less exploding/vanishing gradients
- regularization effect
  - if you use dropout + BN, you may not need L2 regularization
- allows training with saturating nonlinearities (*e.g.* sigmoid)
  - BN prevents them from getting stuck in saturating ranges
- reduces dependency on initialization

어느정도 regularization은
되지만 원래 이용도는
아니었음

Outline

Limitations of BN

- key limitation of BN
  - dependency on mini-batch

    (mean and variance will differ for each mini-batch)

- this dependency causes two main problems
  1. a lower limit on the batch size
  2. difficult to apply to _____ connections in RNN

recurrent

- BN: less well-suited for recurrent models, generative models, and deep RL

볼 수 있음
online learning: 실시간으로
데이터가 계속 들어옴

more details:

->

- dependency on mini-batch can be a problem in settings such as
  - on-line learning (batch size of 1)
  - small-batch training (due to memory or other constraints)
  - generative models/reinforcement learning (highly sensitive to ____)
- recurrent activations of each time-step will have different statistics
  - we have to fit a separate BN layer for each time-step
  - ⇒ more complicated model, more memory

batch사이즈가
작은 경우에
주로 문제됨
(그럼에도
불구하고 대부분
잘됨)

noise

cnn 파라미터
sharing -> filter
공유
rnn 파라미터
sharing ->
timewise
sharing

Batch-independent normalization techniques

- methods that do not suffer from batch size issue
  - weight norm (OpenAI: Kingma) ▸ Link
  - layer norm (Hinton) ▸ Link
  - instance norm
  - group norm (FAIR: He) ▸ Link

파란색 블럭이 mean, std계산하는 대상

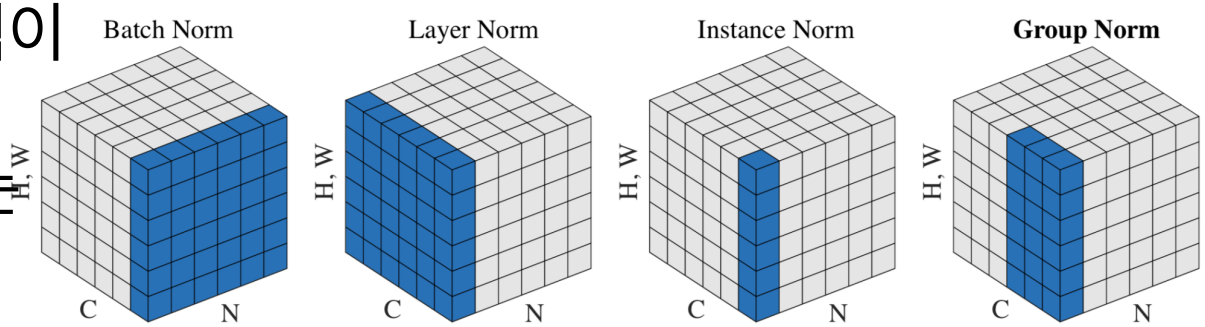


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

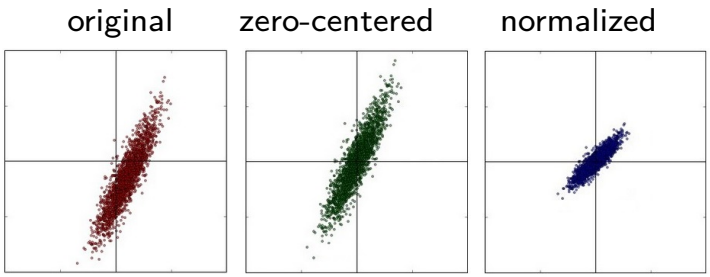
- comparison

sequential

  - LN/IN: better for _____ (RNN, LSTM) or generative models (GAN)
  - GN: better for visual recognition

Decorrelated batch normalization (DBN)

- batch normalization

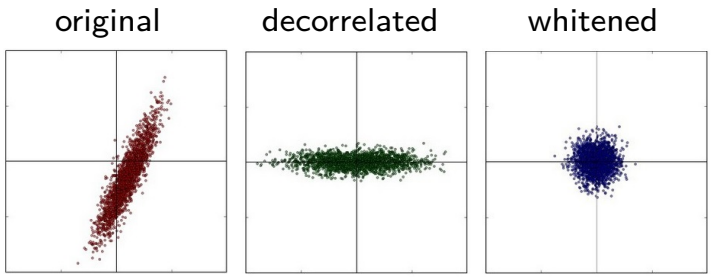


decorrelate 안함 -> approximate whitening, 대신 계산이 쉬움

- normalizes data
- cannot correct correlations among features

$$\widehat{x}_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}$$

- decorrelated batch normalization



- whitens data
- can correct correlations
- needs full covariance matrix of minibatch 잡은 mini-batch에 대해서는 사용가능

$$\widehat{\boldsymbol{x}}^{(i)} = \Sigma^{-\frac{1}{2}}(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})$$

Weight normalization (WN)

- reparameterization of the weight vectors in a neural net
  - _____ the length of those weight vectors from their direction

decouple(끊다)

1: reparameterize the weights $\boldsymbol{w}$ of any layer

$$\boldsymbol{w} = \frac{g}{||\boldsymbol{v}||}\boldsymbol{v}$$

  - separates[4] norm of $\boldsymbol{w}$ from its direction

weight의 크기와 방향을 분리

2: then optimize both g (norm) and $\boldsymbol{v}$ (direction) using gradient descent

- this change of learning dynamics by decoupling norm and direction
  - makes optimization easier $\Rightarrow$ speeds up convergence of sgd

weight를 바꾸므로 mini-batch와 independent함

- no dependency between examples in a minibatch

  $\Rightarrow$ applicable to recurrent, generative, and deep RL

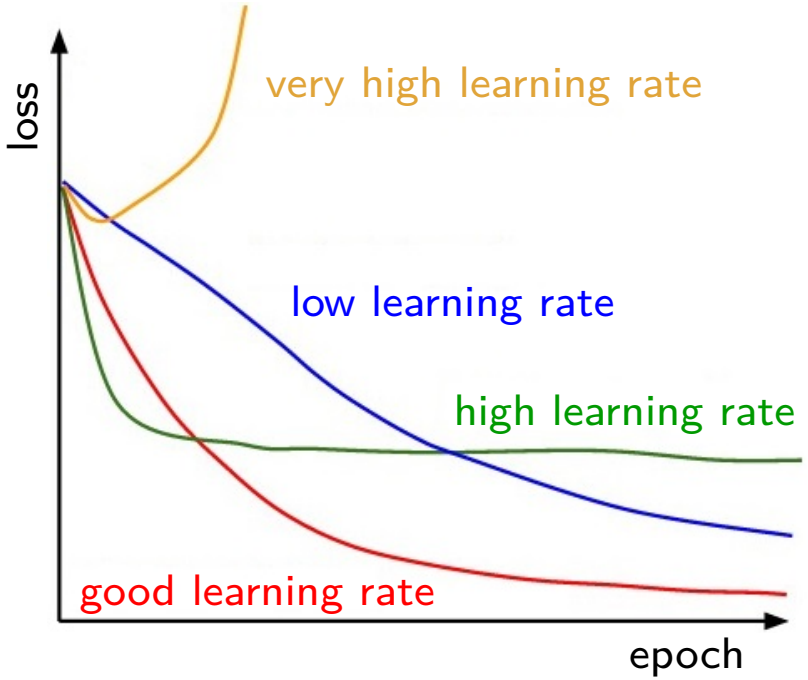
[4] this has effect of fixing the norm of weight vector $\boldsymbol{w}$: we now have $||\boldsymbol{w}|| = g$, independent of parameters $\boldsymbol{v}$

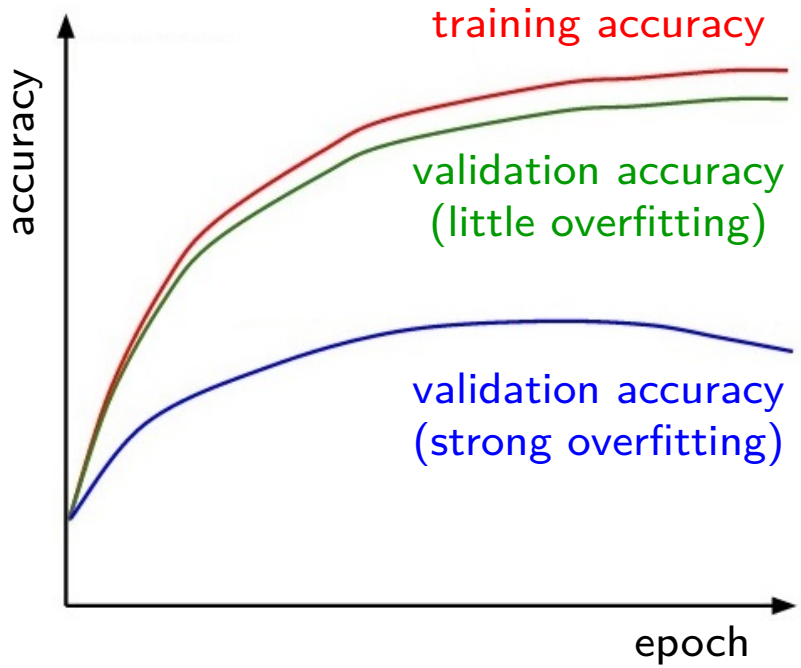
Outline

dnn, cnn-> 배치놈 사용,
rnn, rl ... -> 대안들 사용
고려

Monitoring performance curve

- provides valuable insight
  - into hyperparameter tuning
  - into under/overfit detection

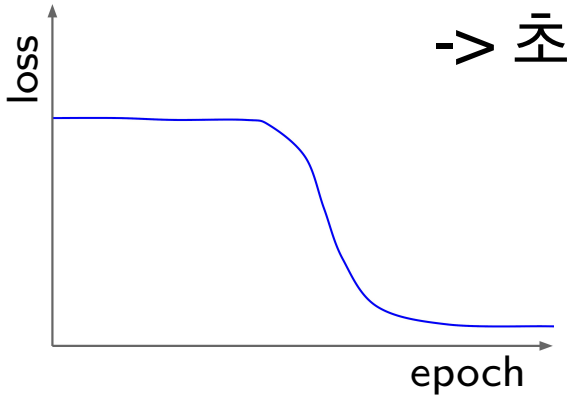




(source: cs231n)

  - and many more!

- example 1:
  - assume you observe the following:



-> 초기화가 잘못된 경우

⇒ prime suspect: bad initialization

- example 2:
  - track the following ratio during training

$$\frac{||\ \text{weight updates}\ ||}{||\ \text{weights}\ ||}$$

recommended value	source
∼0.1%–1%	cs231n
∼1%	Bottou (2015)

  - significant deviation from recommendation: indicates too fast/slow move

Outline

Parameters vs hyperparameters

- **parameters** (= model parameters)
  - what is learned by training

  e.g. weights and biases in a neural net

- **hyperparameters** (= meta-parameters, free parameters)
  - the "knobs" one "turns" during training

  e.g. learning rate, # layers, # hidden units, minibatch size
  - affect algorithm behavior/running time & memory/model quality
  - formally: hyperparameter tuning = *model selection*

- tuning priority
  - top: learning rate
  - 2nd: momentum, # hidden units, minibatch size
  - 3rd: # layers, learning rate decay

- effect of various hyperparameters on model capacity

현대적 조언:
복잡한 모델쓰고
regularization
잘하는게 좋음빵

params 갯수는
늘어남, 대신
output이 작아짐
-> capacity가
줄어드는 효과

hyperparameter	increases capacity when ...	reason	caveats
# hidden units h	increased	increasing h increases representational capacity	increasing h increases both time and memory of essentially every operation
learning rate ϵ	tuned optimally	improper ϵ results in a model with low effective capacity ($\because$ optimization failure)	
convolution kernel width k	increased	increasing k increases the # parameters in the model	a wider kernel gives a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. wider kernels require more memory for parameter storage and increased runtime, but a narrower output reduces memory cost.
implicit zero padding	increased	adding implicit zeros before convolution keeps the representation size large	increased time and memory cost of operations
weight decay coefficient c	decreased	decreasing c frees the model parameters to become larger	
dropout rate	decreased	dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

How to choose hyperparameters

- two basic approaches:

1. manual tuning
   - we need to understand
     - what hyperparameters do and
     - how ML models achieve good generalization
2. automated selection: meta-learning, autoML
   - reduces the need to understand the above ideas
   - but often much more computationally costly

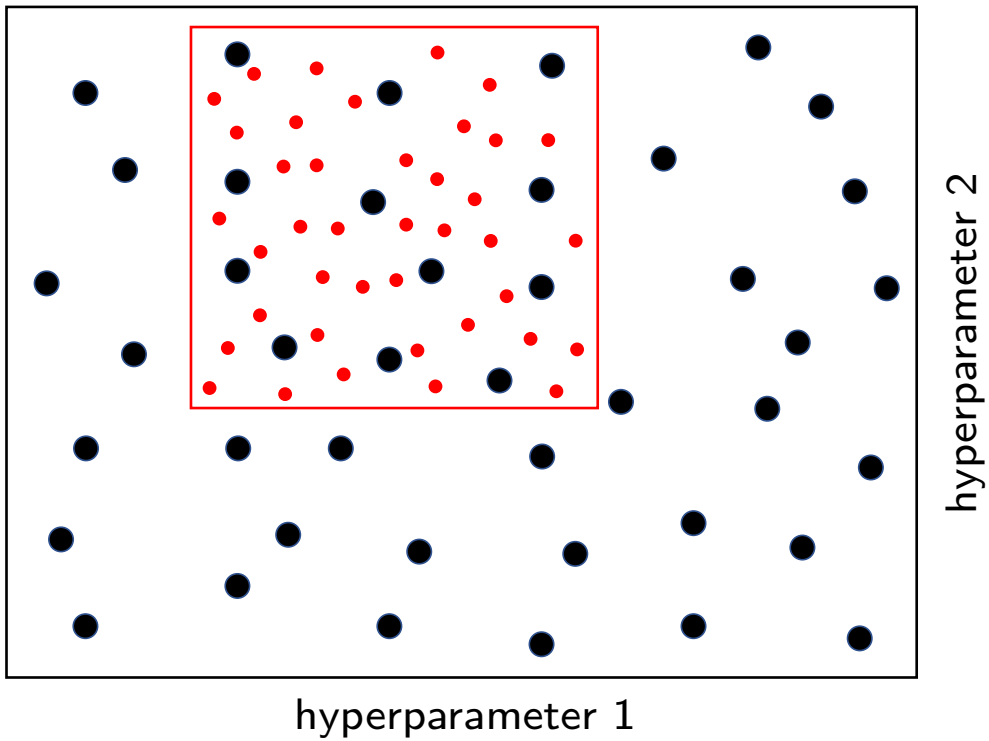
Manually tuning hyperparameters

validation
error가 제일
좋은것을 사용

- basic strategy
  - plug in a set of hyperparameter values
  - do cross-validation to choose the best one
- challenge
  - search space: enormous
- known heuristics
  1: coarse-to-fine sampling
  2: random sampling instead of grid search
  3: use appropriate distribution for sampling

1: Coarse-to-fine sampling scheme

- cross-validation in stages
  1. sample coarsely in a large region (spending only a few epochs each)
  2. then focus on a smaller region, sampling more densely inside

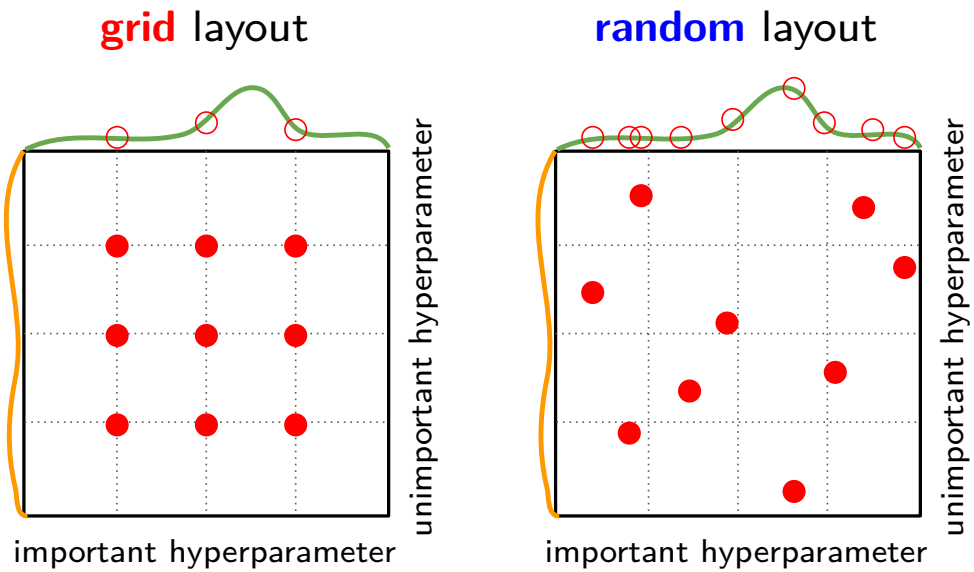


(source: coursera)

2: Random sampling over grid search

특정 hp를 고정시키면 나머지에 대하여 2^n 계산 낭비쬬

- grid search
  - wastes computation
    - ↑ wasted amount: exponential in # of non-influential hyperparameters
- random search: allows richer exploration (exponentially more efficient)
  - tests a unique value of every influential hyper-p on nearly every trial



(source: Bergstra and Bengio)

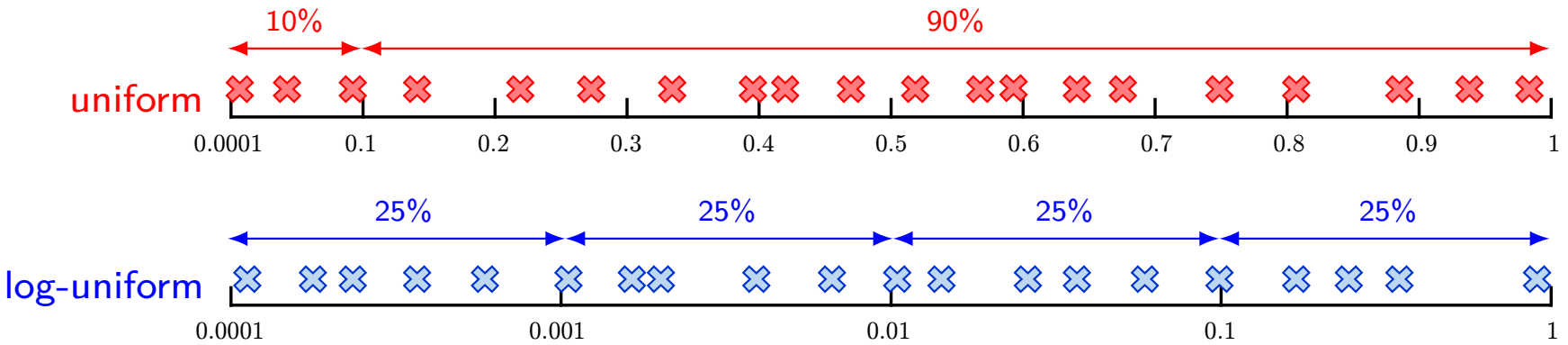
3: Appropriate distribution for sampling

- **binary**/**discrete** hyperparameters
  - sample from Bernoulli or multinoulli distribution

- **positive real-valued** hyperparameters
  - sample from a uniform distribution on log-scale

e.g. two ways to sample ϵ in range of [0.0001, 1]

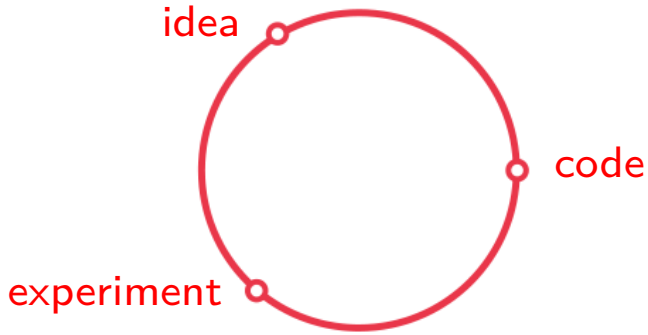
1. uniform distribution: $\epsilon \sim \mathrm{U}(0.0001, 1)$ (linear)
2. log-uniform distribution: $\epsilon' \sim \mathrm{U}(\log 10^{-4}, \log 10^{0}) \rightarrow \epsilon = 10^{\epsilon'}$ better



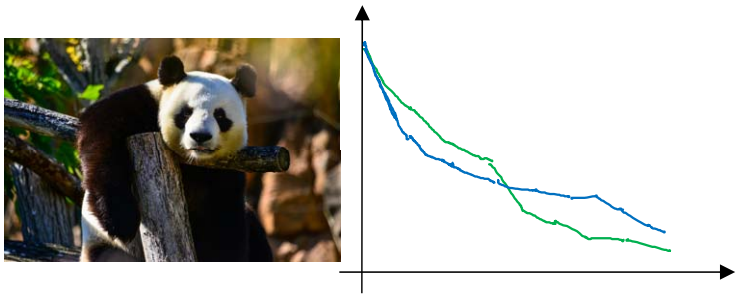
Re-test hyperparameters occasionally

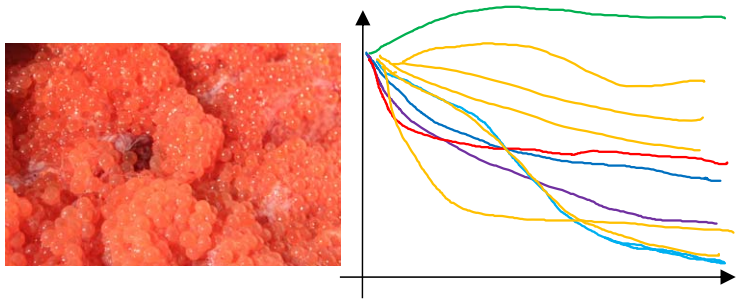
hp 변경해야하는 경우가 많음

- common in many applications
  - NLP, vision, speech, logistics, ...
- intuitions do get stale
  - need to re-evaluate occasionally



- two schools of thoughts (depending on computational resources)
  - "panda": babysit one model
  - "caviar": train many models in parallel





(source: coursera)

Outline

When your ML system performs poorly

- it is difficult to tell the reason:
  - algorithm itself, or
  - implementation bug
- ML systems: challenging to debug
  1. no prior knowledge on intended behavior of the algorithm
  2. multiple adaptive components → if one part breaks, other parts can adapt
- most debugging strategies for neural nets:
  - designed to get around these two difficulties

Debugging strategies

- we design either
  - a case
    - that is so _____ → correct behavior actually can be predicted
  - a test
    - that exercises one part of implementation in isolation

simple

regression test
-> 원래 잘 돌아가던게 변경 후 안돌아 갈 수 있으므로 테스트 해봐야함

- important debugging tests include (see textbook for details):
  - T1 visualize the model in action
  - T2 visualize the worst mistakes
  - T3 reasoning using train and test error
  - T4 fit a tiny dataset
  - T5 compare back-propagated derivatives to numerical derivatives
  - T6 monitor histograms of activations and gradient

Outline

Data, data, data

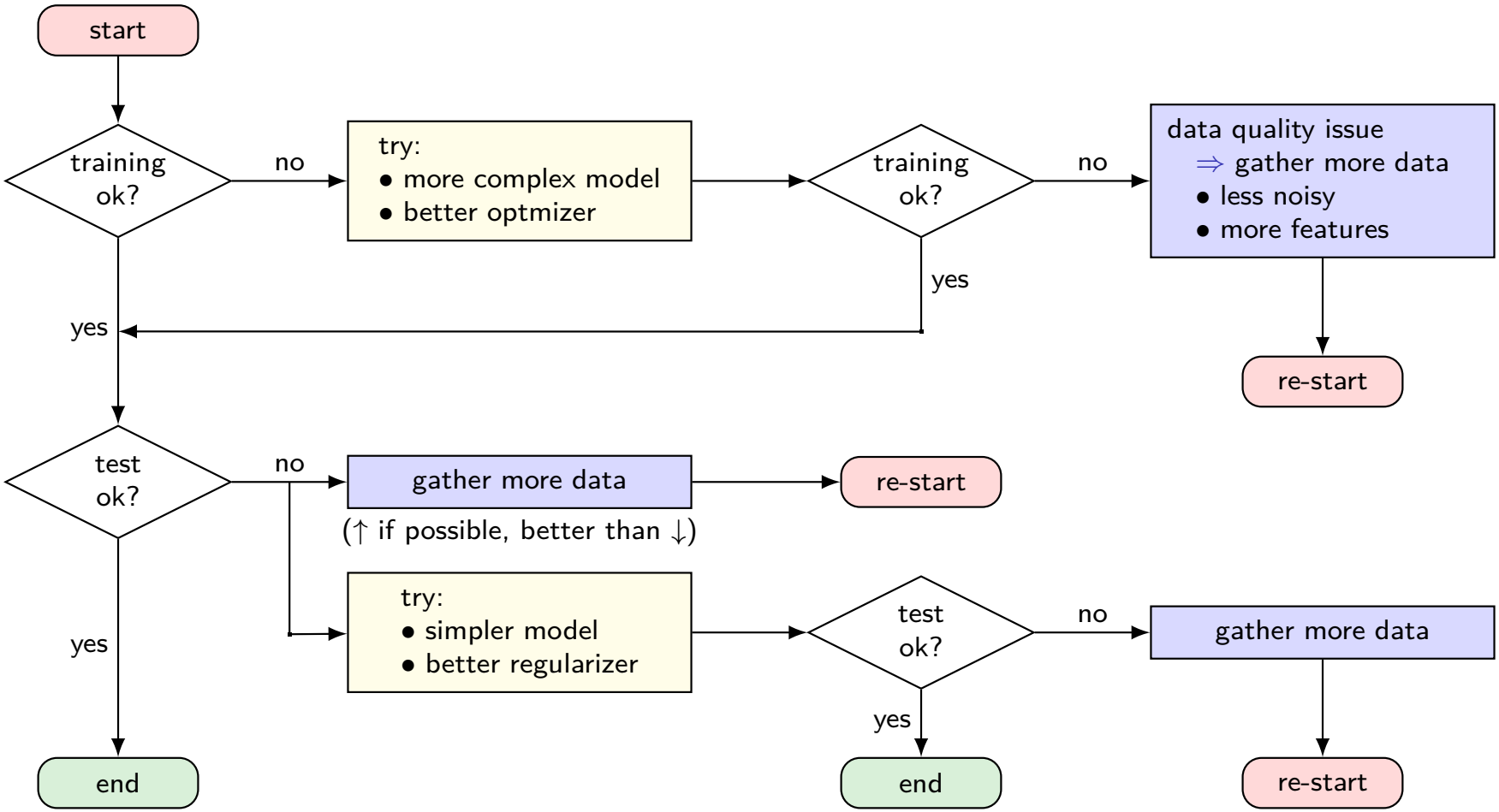
data가 너무 많으면 fitting 자체가
이슈일 수 있음

- some datasets grow rapidly in size (faster than computing power)
  - $\Rightarrow$ it is becoming more common for ML applications to
    - ▷ use each training example only once or
    - ▷ even make an incomplete pass through training set

- when using an extremely large training set:
  - ▶ overfitting is not an issue
  - ▶ sounder fitting/computational efficiency become predominant concerns

- but more often than not:
  - ▶ we often suffer from limited amount of training data

Deciding whether to gather more data



- how much more data? try data sizes on a **logarithmic** scale

  e.g. double # of examples between experiments

Recycling models/algorithms

- if your task is similar to another task studied extensively
  - copy the model/algorithm known to perform best on that task
- you may even want to copy a trained model from that task

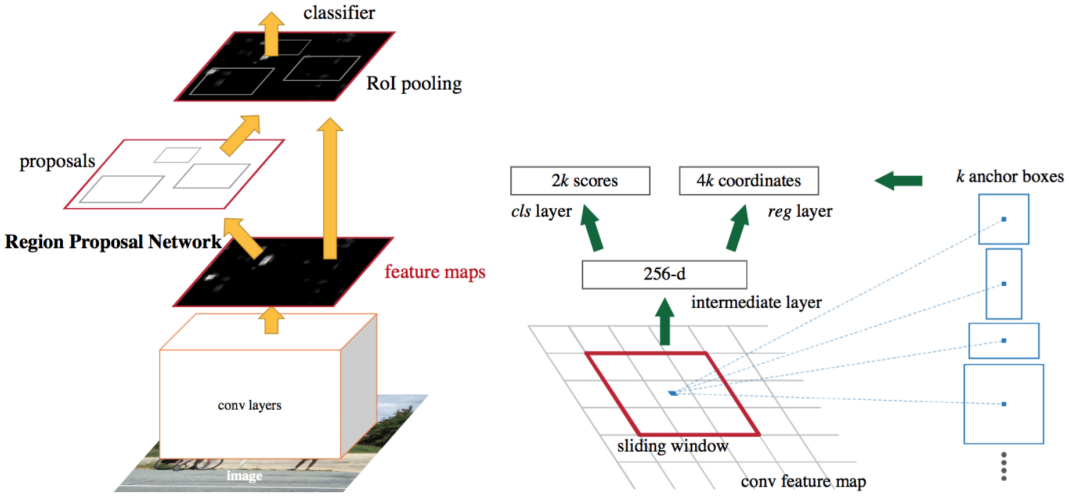
  e.g. use features from a CNN to other tasks

  ↑ trained on ImageNet
- transfer learning and domain adaptation
  - what has been learned in one setting ("source domain")
  - is used in another setting ("target domain")
- this generalizes the idea of representation transfer
  - unsupervised task ↔ supervised task (page 68)

supervised를 위해 pre-process로 unsupervised 이용하는 경우도 있음

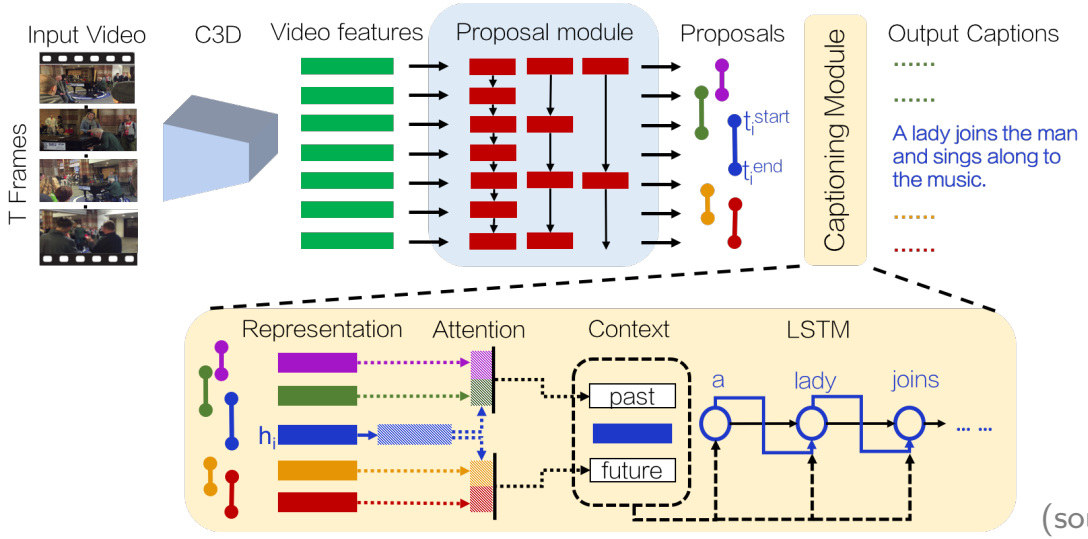
Examples

- transfer learning with CNNs: pervasive



- object detection (faster R-CNN)

(source: Ren, He, Girshick, Sun)



- video captioning

(source: Krishna, Hata, Ren, Fei-Fei, Niebles)

Model zoos

- many deep learning frameworks
  - provide a model zoo of pretrained models

  e.g. Caffe ▸ Link

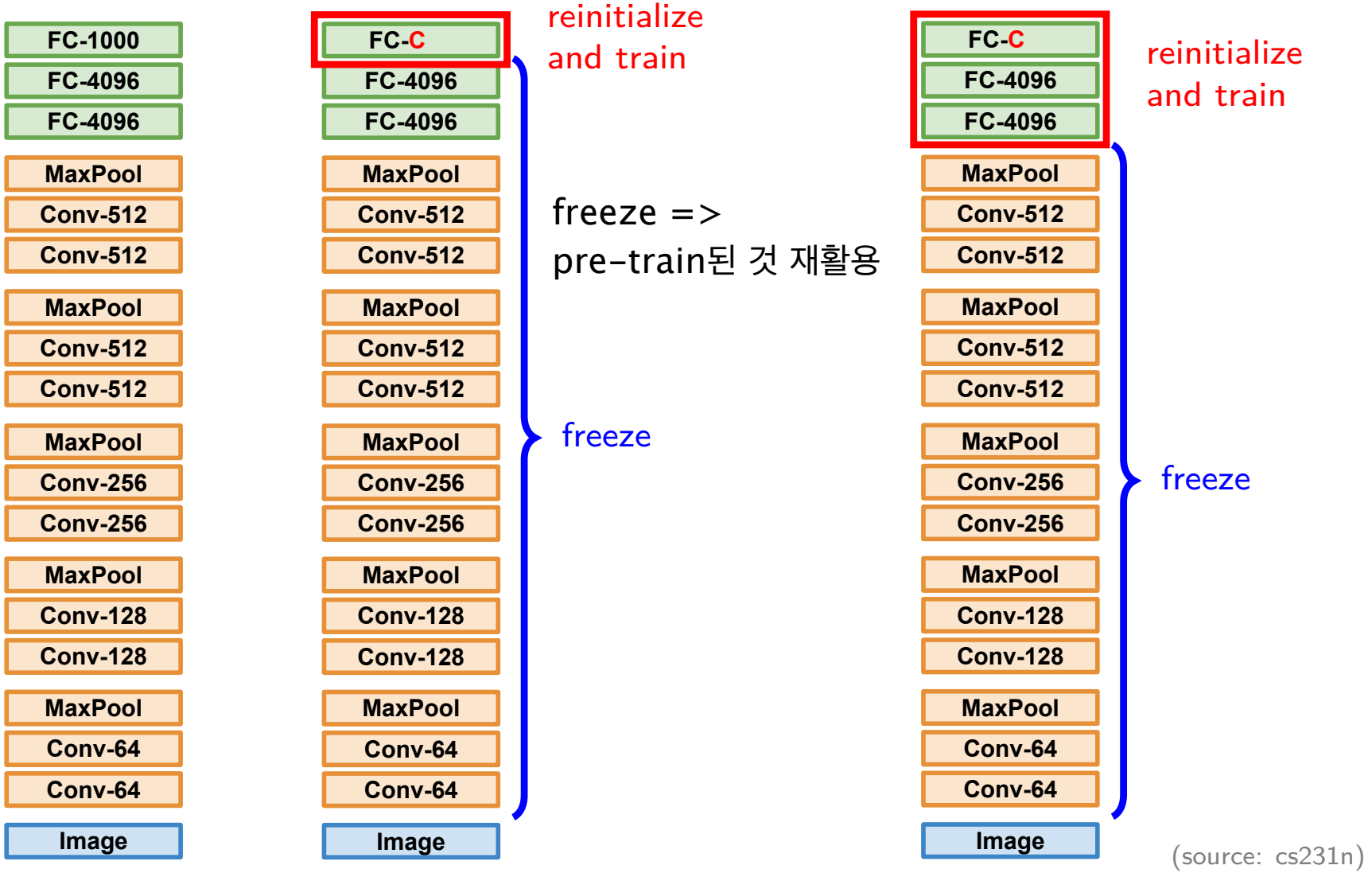
  TensorFlow ▸ Link

  PyTorch ▸ Link
  - before start training from scratch, should consider using them

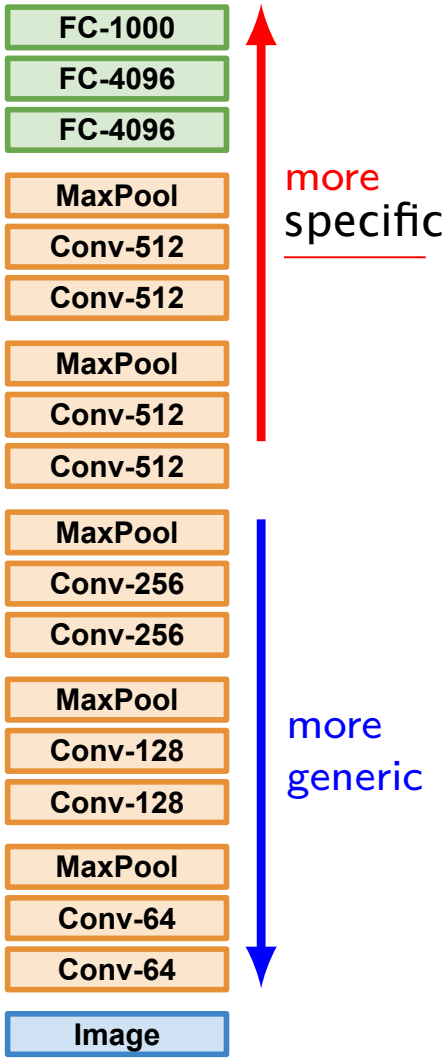
Recycling CNN features

- ※ bigger dataset ⇒ retrain more layers
- ※ lower learning rate when finetuning
  (*e.g.* start with 0.1 of original rate)

- pretrained
- new task: small data
- new task: bigger data



(source: cs231n)



(source: cs231n)

light한 linear classifier사용

	similarity to source domain	
	very similar	very different
target domain data: very little	use linear classifier on top layer	(you're in trouble) try linear classifier from different stages
target domain data: quite a lot	finetune a few layers	finetune many layers

Beginning by unsupervised learning

ex) label이 별로 없는 raw data가 많은 경우 -> unsupervised learning으로 pre-process

- a common question:
  - whether to begin by using unsupervised learning
  - answer: domain specific

- some domains (such as NLP): nlp등에서 유용
  - benefit tremendously from unsupervised learning

  e.g. learning unsupervised word embeddings

- other domains (such as computer vision):
  - current unsupervised learning techniques bring no benefit
  - except in semi-supervised setting with very few labeled examples

Extreme forms of transfer learning

사람한테는 쉽지만
computer에게는 어려운 task

- one-shot learning
  - only one labeled example of the transfer task is given
- zero-shot learning (aka zero-data learning)
  - no labeled examples are given at all

"검은 타이 정장 훈남"

"긴 머리 청바지 그녀"

Outline

Summary

- neural net training: often extremely time-consuming

  $\Rightarrow$ effective strategies needed (more an art than a science)

- phase 1: setup
  - preprocessing: zero-centering, normalization, decorrelation, whitening
  - weight initialization: Xavier and He initialization
  - regularization $\leftarrow$ will be covered separately

- phase 2: monitor training dynamics
  - parameter updates (optimization) $\leftarrow$ will be covered separately
  - batch normalization to alleviate covariate shift issue
  - hyperparemter tuning (log-scale random search often preferred)

- phase 3: evaluate and improve
  - difficulty of debugging ML systems: lack of prior knowledge, adaptivity
  - transfer learning and model zoo: for speedy development