



M2177.003100

Deep Learning

[8: The Art of Neural Net Training]

Electrical and Computer Engineering
Seoul National University

© 2020 Sungroh Yoon. this material is for educational uses only. some contents are based on the material provided by other paper/book authors and may be copyrighted by them.

(last compiled at 21:04:00 on 2020/10/12)

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

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](#)

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

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 experiments
- 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: Adam
- **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
- batch normalization
 - ▶ can reduce generalization error and allow dropout to be omitted¹

¹due 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 dynamics
 - ▶ covariate shift and normalization
 - ▶ parameter updates (optimization)
 - ▶ hyperparameter tuning
- phase 3: evaluate and improve
 - ▶ model ensembles
 - ▶ data augmentation, transfer learning
 - ▶ debugging



* **regularization/optimization:** covered separately

image source: <https://images.app.goo.gl/cAfEjLVqbLCariQo6>

Outline

Introduction

Preprocessing and Initialization

Data Preprocessing

Weight Initialization

Normalization

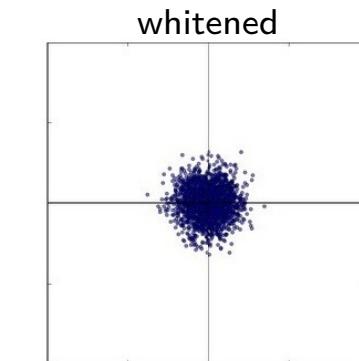
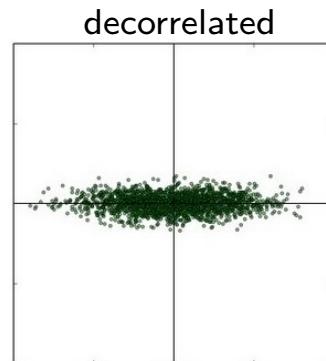
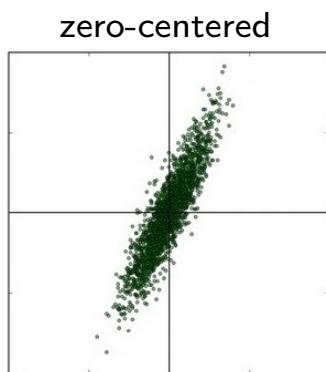
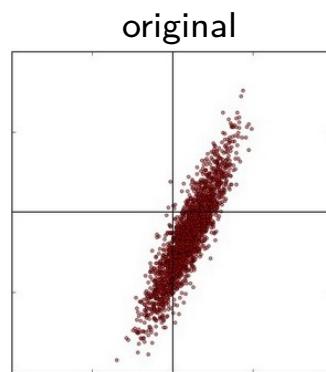
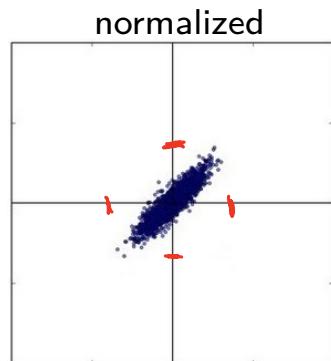
Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

Data preprocessing

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



+ ==

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

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

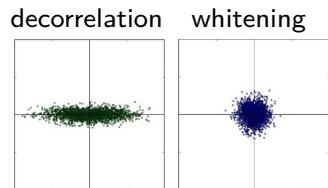
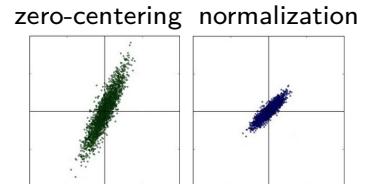
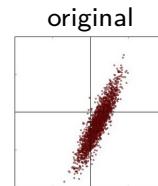


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

Why do preprocessing?

- can speed up training

1. importance of zero-centering

- ▶ recall: all positive/negative inputs
 - ▷ cause zig-zag updates
 - ⇒ slow convergence

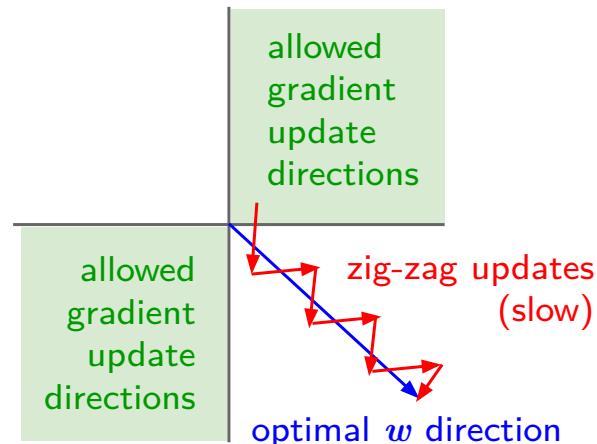
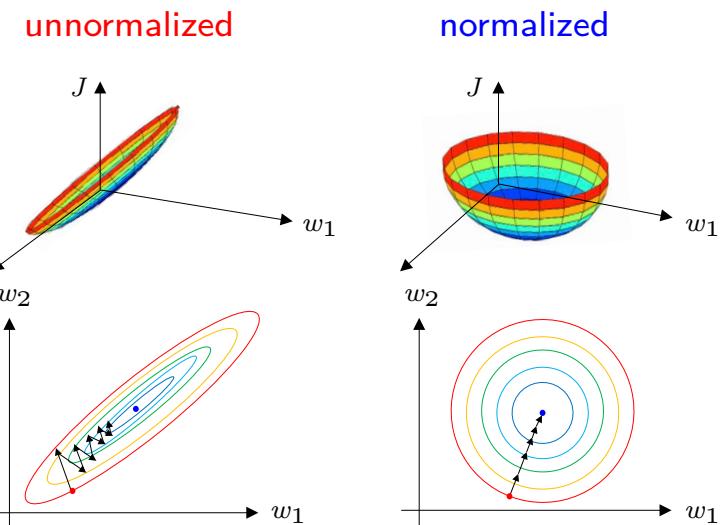


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

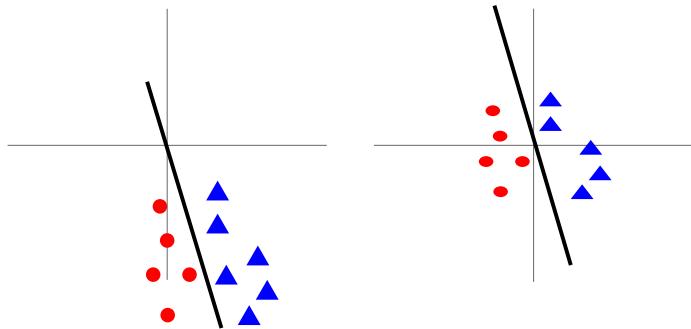
2. importance of normalization

- unnormalized data

- ▶ may show wiggly behavior
- ⇒ difficult to optimize



- unnormalized



- normalized

- before normalization
 - ▶ small change in weights
 - ⇒ big swing in boundary
- after normalization
 - ▶ boundary less sensitive
 - ⇒ easier to optimize

image sources: Ng, Deep Learning (Coursera), <https://www.coursera.org/specializations/deep-learning>; Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017)

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 **entire 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

Introduction

Preprocessing and Initialization

Data Preprocessing

Weight Initialization

Normalization

Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

Weight initialization

- reasonable assumption about final weight values
 - ▶ roughly half **positive** + half **negative**
⇒ **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 **Variance** in weights
- common idea
 - ▶ initialize weights using **zero mean** Gaussian with **some variance**

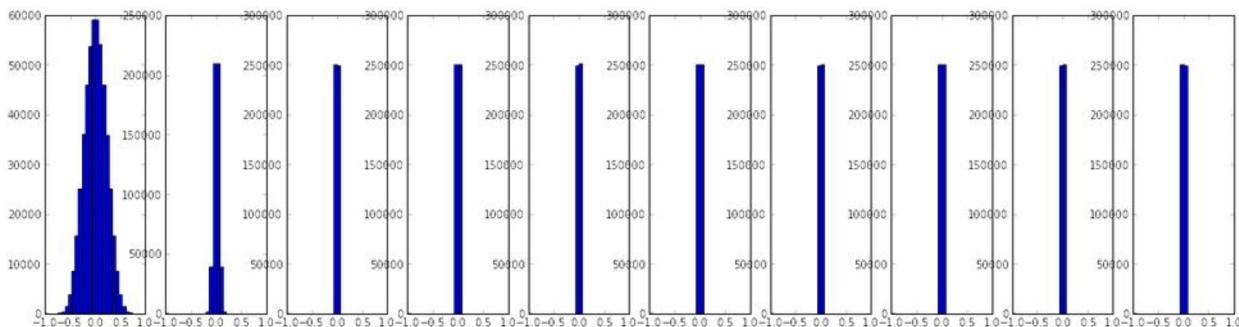
- attempt #2: initialization with **small** random numbers

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):



⇒ all activations rapidly become zero



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

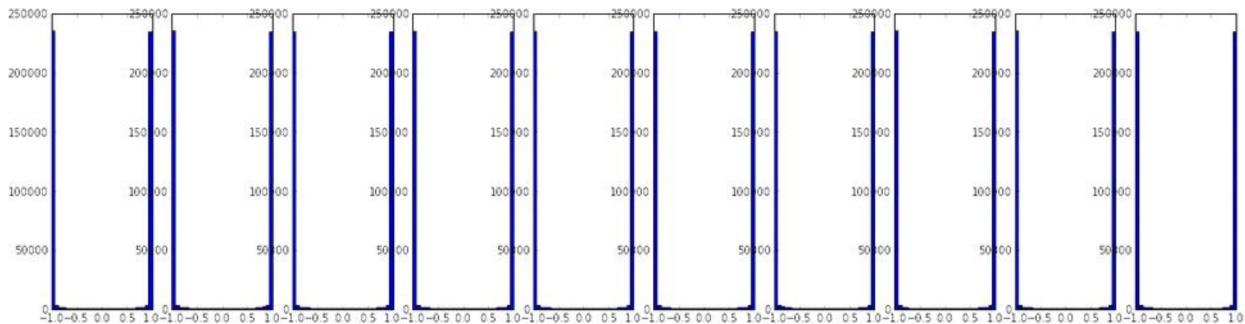
image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

- 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 saturated (± 1) \Rightarrow gradients killed

- observation:

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

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

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 small as n grows
- intuitively: setting $\text{Var}(w) = \frac{1}{n}$ will do $\leftarrow n: \# \text{ inputs ("fan in")}$
- mathematically:
 - assume we want $\text{Var}(y) = \text{Var}(x)$

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

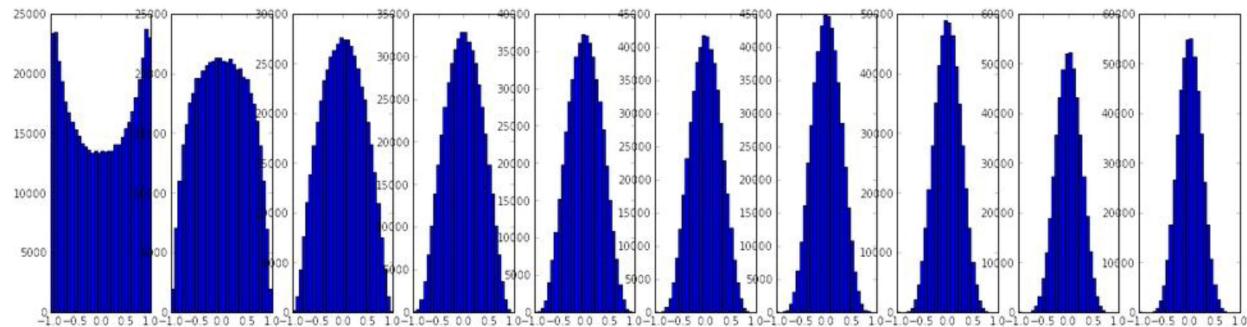
- setting $\text{Var}(w) = \frac{1}{n}$ will do

Xavier and He initialization

- Xavier initialization

- ▶ reasonable choice for sigmoid type nonlinearity

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

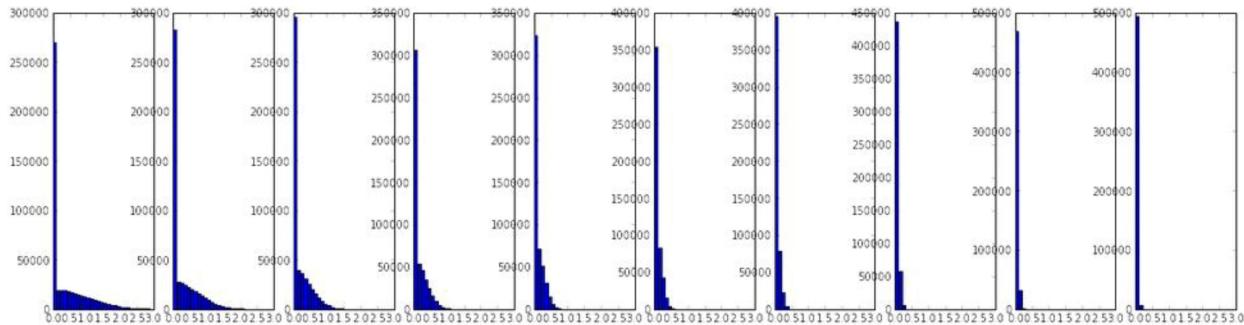


- ▶ variant (considers backprop direction)

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

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

- Xavier initialization: often does not work well for ReLU



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

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

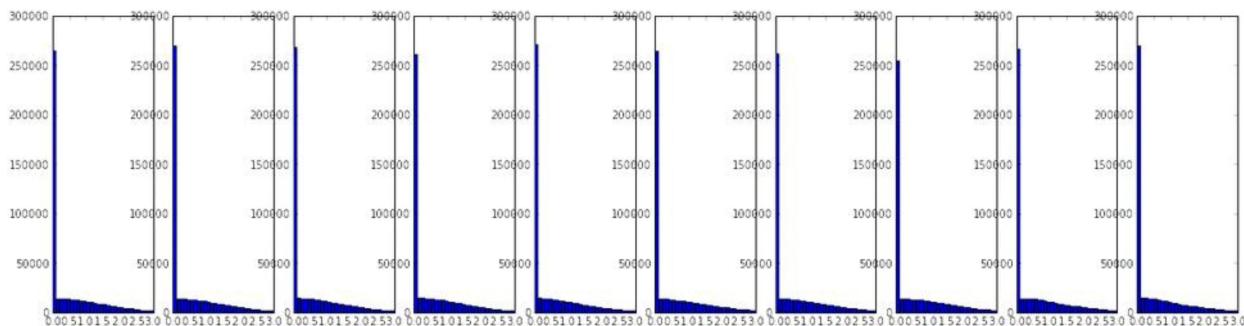
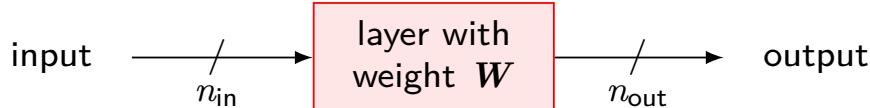
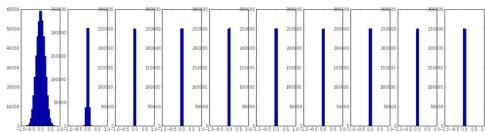


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), <http://cs231n.stanford.edu/2017/index.html>

- comparison:



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



- 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 activations at all layers

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017)

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 zero
 - ▶ rationale: adjusting weights is sufficient for symmetry breaking
- proper initialization: active area of research!

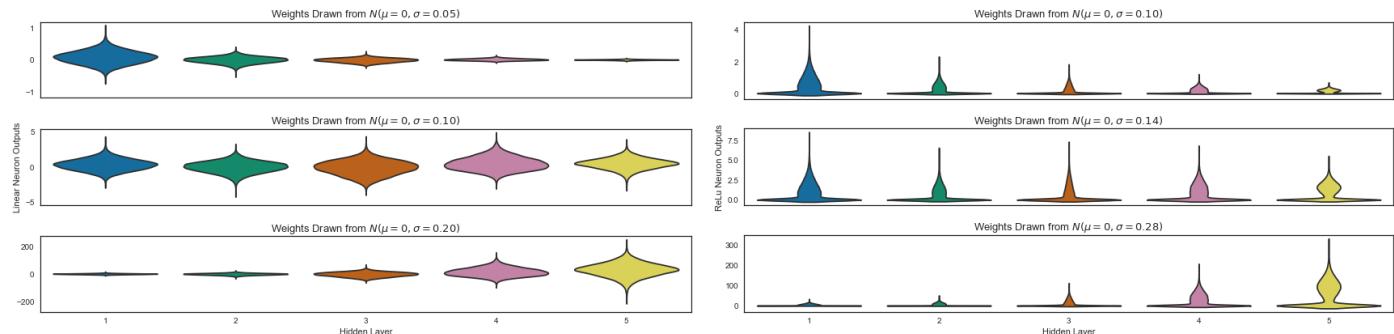


image source: <https://intoli.com/blog/neural-network-initialization/>

Alternative idea

- what we've done so far:
 - ▶ adjust initial weights → 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 batch normalization



image source: <https://images.app.goo.gl/Pb73urF8hs6LYxRu7>

Outline

Introduction

Recent Techniques

Preprocessing and Initialization

Monitoring and Tuning Training

Normalization

Data Scarcity and Transfer Learning

Covariate Shift

Summary

Batch Normalization

Covariate shift

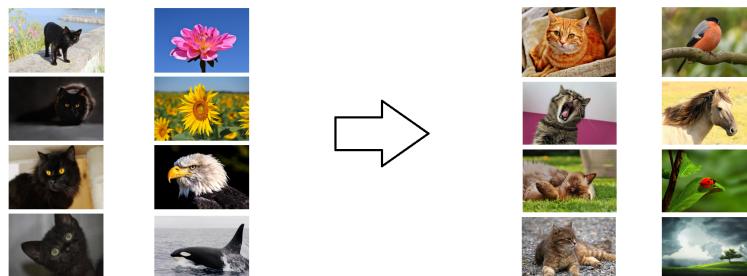
- synonyms

- ▶ **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 input distribution to a learning system

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

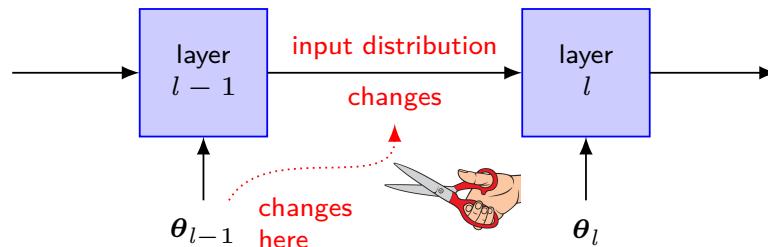


- ▶ cure: domain adaptation

image source: Ng, Deep Learning (Coursera), <https://www.coursera.org/specializations/deep-learning>

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 initialization
 - ▶ makes it hard to train models with saturating nonlinearities
- cure: **cut the interaction** → how?

scissors clipart from: <https://images.app.goo.gl/6a45RbcHRwwvgAp16>

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

Outline

Introduction

Recent Techniques

Preprocessing and Initialization

Monitoring and Tuning Training

Normalization

Covariate Shift

Data Scarcity and Transfer Learning

Batch Normalization

Summary

Batch normalization (Ioffe and Szegedy, 2015)

▶ Paper

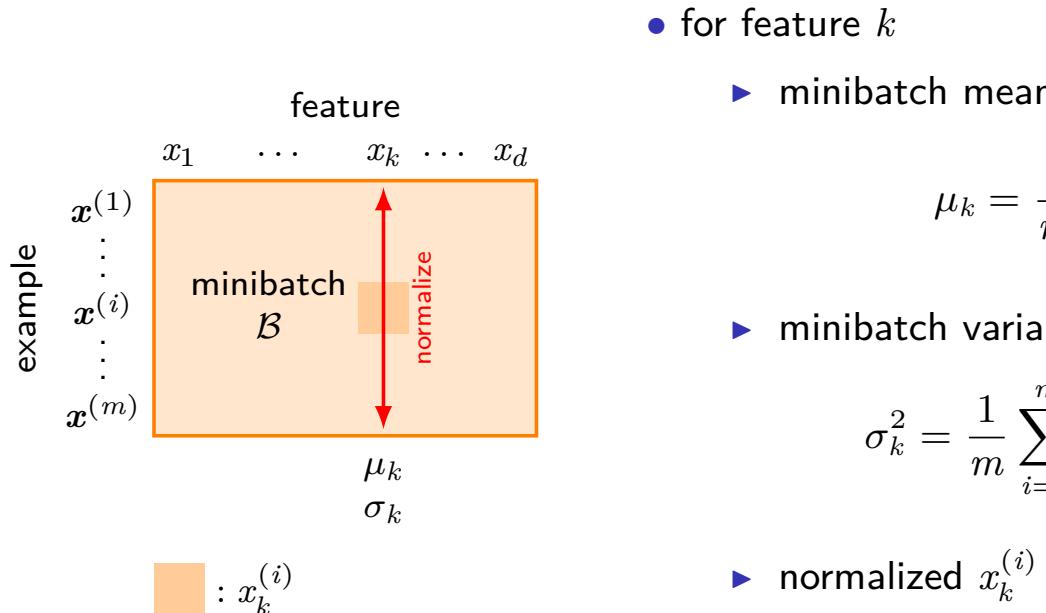
- alleviates many issues with properly initializing neural nets
 - ▶ very popular these days
- idea: explicitly forces the activations throughout a net
 - ▶ to take on a unit gaussian distribution at the beginning of training
- core observation: this is possible because
 - ▶ normalization is a simple differentiable operation
- implementation: create and insert a BN layer
 - ▶ FC/conv layer → BN layer → nonlinearity layer
- interpretation
 - ▶ BN = doing preprocessing at every layer of the net
 - ↑
but integrated into the net itself in a differentiable manner



image source: <https://images.app.goo.gl/ok7ppbJg6C938pt2A>

BN operation #1: normalization

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



- for feature k

- ▶ minibatch mean

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

- ▶ minibatch variance

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

- ▶ normalized $x_k^{(i)}$

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

²for 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 $x_k^{(i)}$

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

→ 2d γ & β parameter

- these parameters
 - ▶ learned along with original model parameters by sgd
 - ▶ restore the representation power of the network³

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

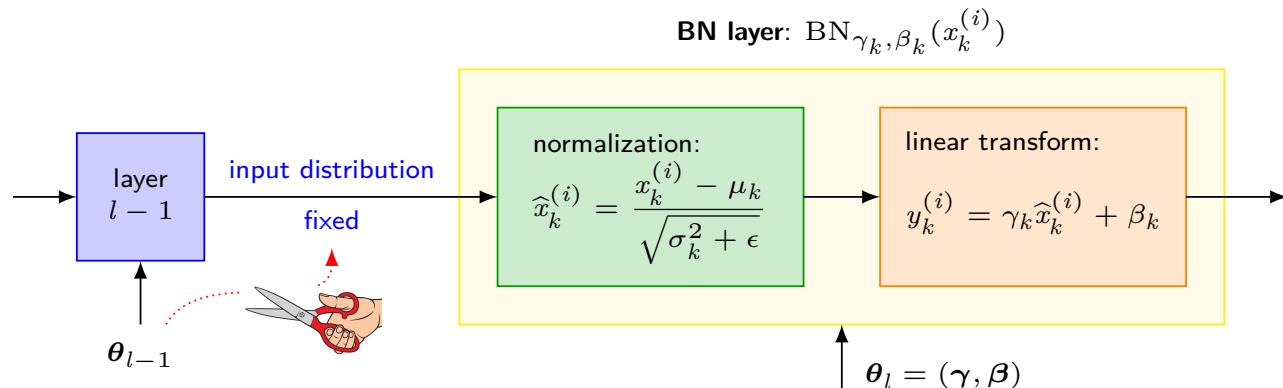
Batch norm operations altogether

- op #1: normalization

- ▶ to fix the distribution of inputs to each layer
- ⇒ cut high-order interactions between different layers
- ⇒ Simplifies learning dynamics ⇒ speed up training

- op #2: linear transform

- ▶ to restore representation power of the network
(don't get confused why this is needed)



scissors clipart from: <https://images.app.goo.gl/6a45RbcHRwwvgAp16>

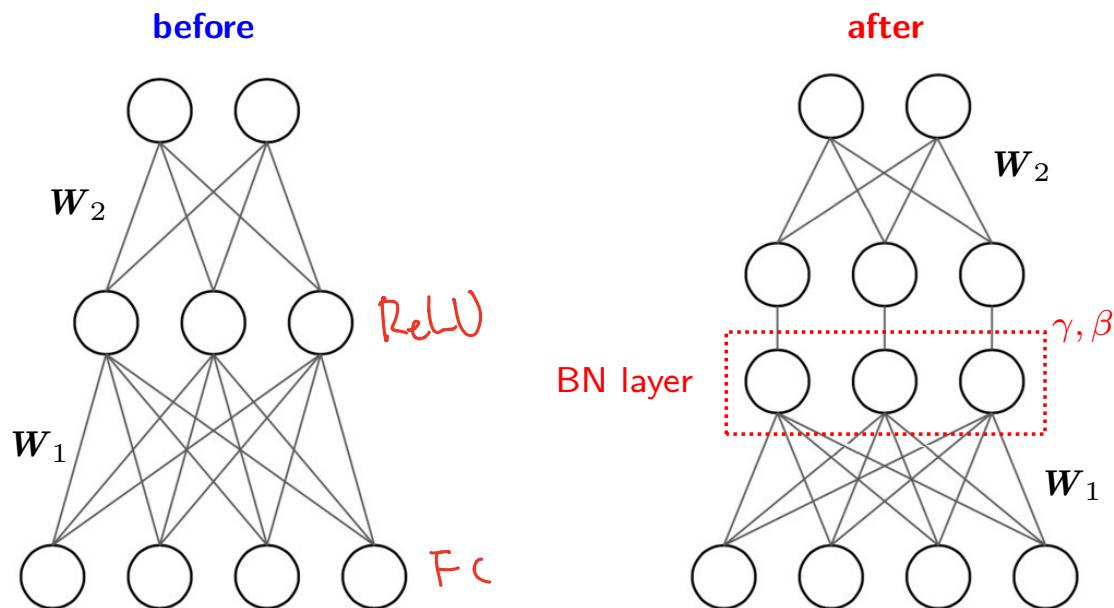
BN at test time

- at test time BN layer functions differently:
 - ▶ the mean/std are not computed based on the batch
 - ▶ instead, a single fixed 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	\dots	
	\downarrow	\downarrow	\downarrow		
mean series:	$\mu^{(\mathcal{B}_1)}$	$\mu^{(\mathcal{B}_2)}$	$\mu^{(\mathcal{B}_3)}$	\dots	$\longrightarrow \mu$ for test
variance series:	$\sigma^{(\mathcal{B}_1)}$	$\sigma^{(\mathcal{B}_2)}$	$\sigma^{(\mathcal{B}_3)}$	\dots	$\Longrightarrow \sigma$ for test

Implementation

- inserted to a net as “BN layer”
 - ▶ learnable parameters: γ, β



- usually inserted
 - ▶ after 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

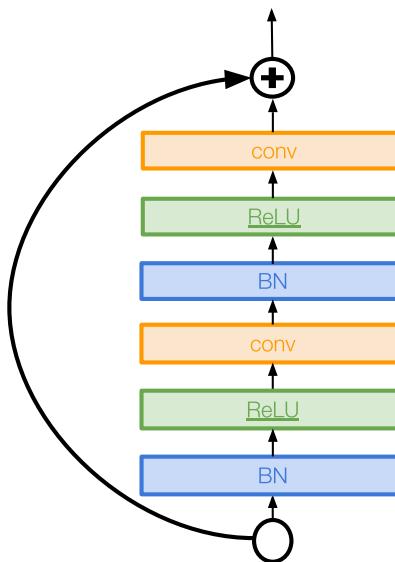


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

BN layer summary

- input

- ▶ $X : \underbrace{m}_{\text{batch size}} \times \underbrace{d}_{\# \text{ of features}}$

- learnable parameters

- ▶ $\gamma, \beta : 1 \times d$

- intermediates

- ▶ $\mu, \sigma : 1 \times d$
- ▶ $\widehat{X} : m \times d$

- output

- ▶ $Y : m \times d$

- training time

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

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

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

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

- test time

$\mu_k = \text{(running average from training)}$ fixed

$\sigma_k^2 = \text{(running average from training)}$

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

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

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**

Outline

Introduction

Preprocessing and Initialization

Normalization

Covariate Shift

Batch Normalization

Recent Techniques

Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

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 (≥ 16)
 2. difficult to apply to recurrent connections in RNN
- BN: less well-suited for recurrent models, generative models, and deep RL

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 noise)
- 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

Batch-independent normalization techniques

- methods that do not suffer from batch size issue

- ▶ weight norm (OpenAI: Kingma) [► Paper](#)
- ▶ layer norm (Hinton) [► Paper](#)
- ▶ instance norm [► Paper](#)
- ▶ group norm (FAIR: He) [► Paper](#)

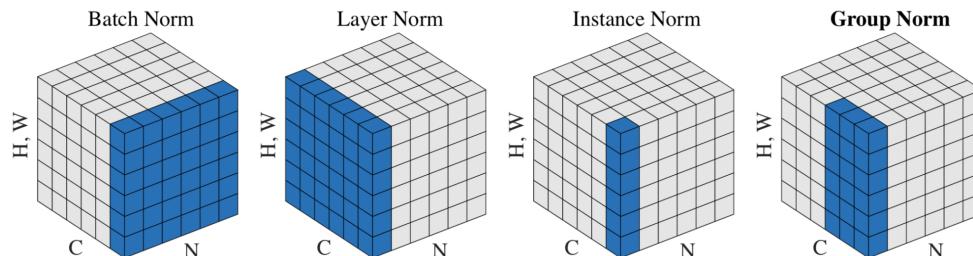


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

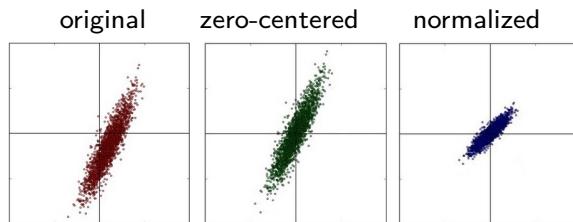
- comparison

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

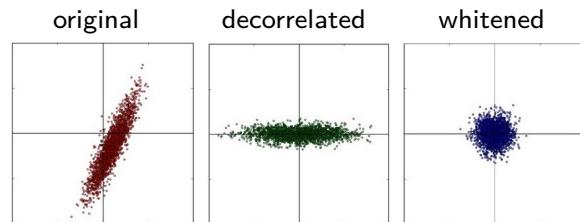
image source: Wu, Yuxin, and Kaiming He. "Group normalization." *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018. <https://arxiv.org/pdf/1803.08494.pdf>

Decorrelated batch normalization (DBN)

- batch normalization



- decorrelated batch normalization



- ▶ normalizes data
 - ▶ cannot correct correlations among features
- ▶ whitens data
 - ▶ can correct correlations
 - ▶ needs full covariance matrix of minibatch

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

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

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

Weight normalization (WN)

- reparameterization of the weight vectors in a neural net
 - ▶ decouples the **length** of those weight vectors from their **direction**

1: reparameterize the weights w of any layer

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

▶ separates⁴ **norm** of w from its **direction**

2: then optimize both g (**norm**) and v (**direction**) using gradient descent

- this change of learning dynamics by decoupling **norm** and **direction**
 - ▶ makes optimization easier \Rightarrow speeds up convergence of sgd
- no dependency between examples in a minibatch
 - \Rightarrow applicable to recurrent, generative, and deep RL

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

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

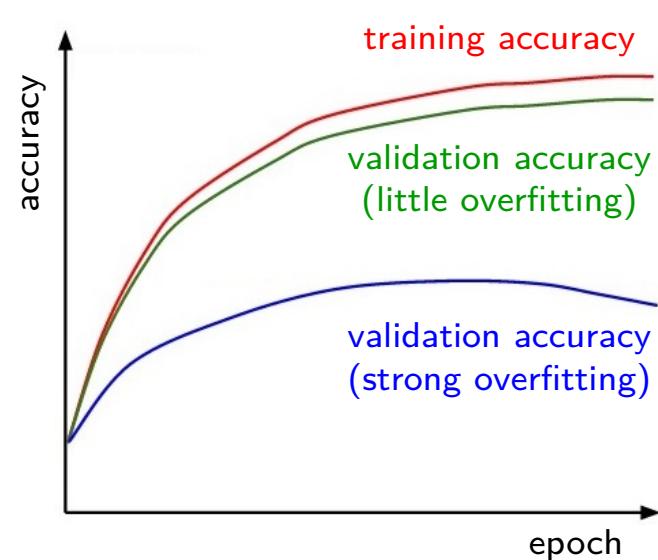
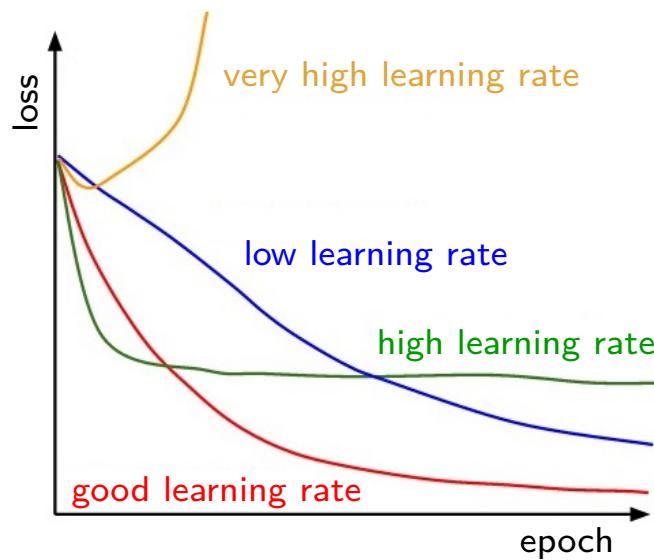
Hyperparameter Tuning
Debugging Strategies

Data Scarcity and Transfer Learning

Summary

Monitoring performance curve

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

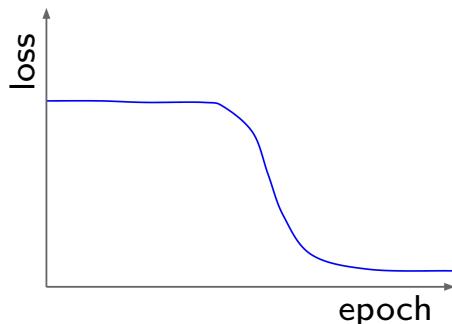


- ▶ and many more!

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

- example 1:

- ▶ assume you observe the following:



⇒ 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

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017), <http://cs231n.stanford.edu/2017/index.html>

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

Hyperparameter Tuning
Debugging Strategies

Data Scarcity and Transfer Learning

Summary

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

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	

table source: I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016

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 NAS

- ▶ reduces the need to understand the above ideas
- ▶ but often much more computationally costly

Manually tuning hyperparameters

- 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

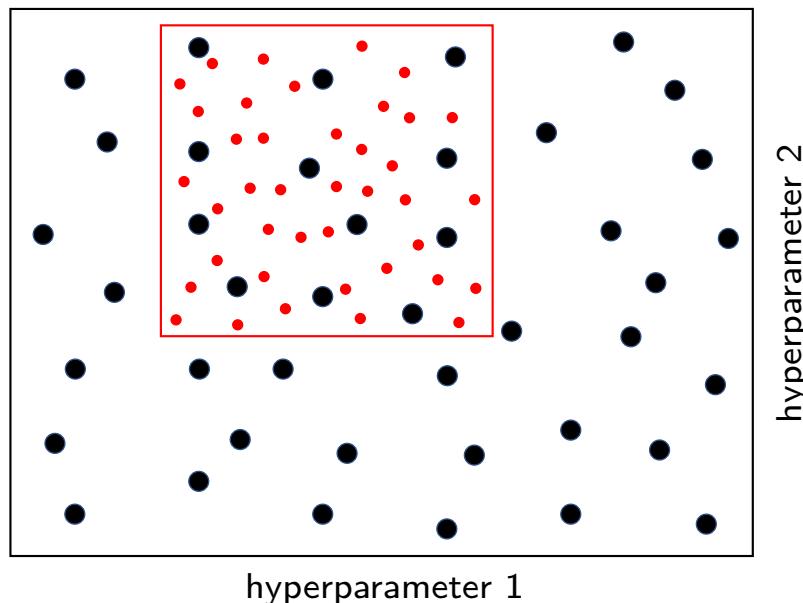


image modified from: Ng, Deep Learning (Coursera), <https://www.coursera.org/specializations/deep-learning>

2: Random sampling over grid search

- **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

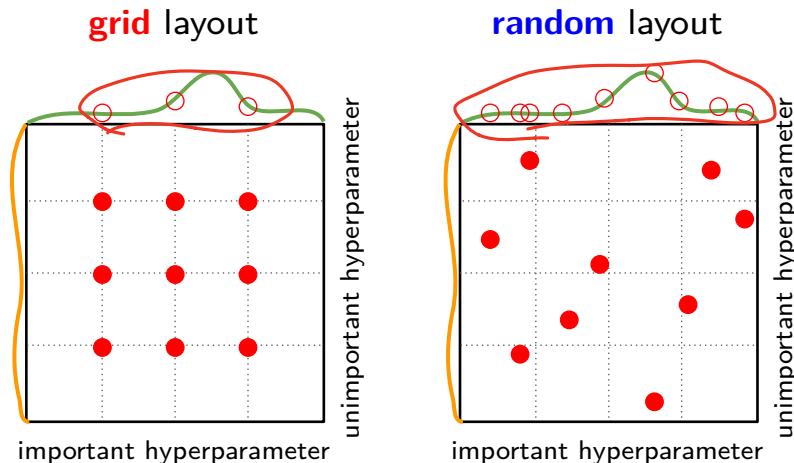


image modified from: Bergstra and Bengio. "Random search for hyper-parameter optimization." *The Journal of Machine Learning Research* 13.1 (2012): 281-305. <https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>

3: Appropriate distribution for sampling

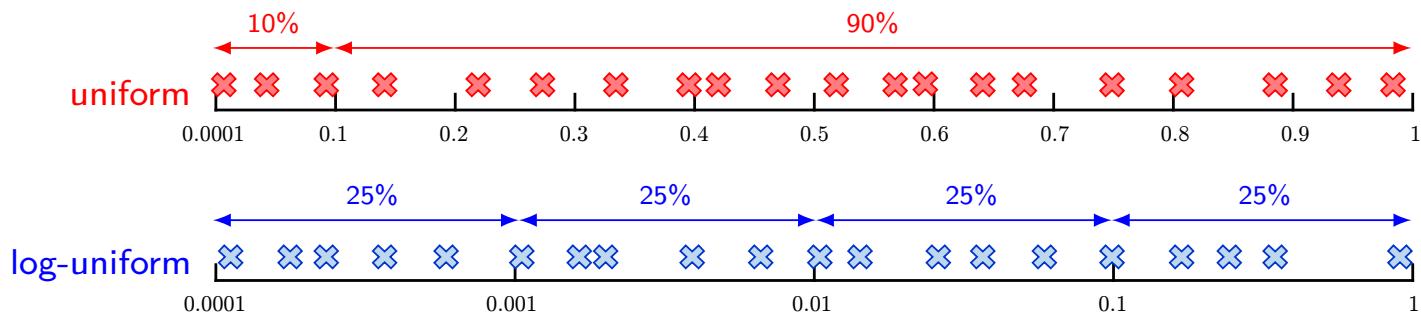
- **binary/discrete** hyperparameters
 - ▶ sample from Bernoulli or multinoulli distribution

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

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

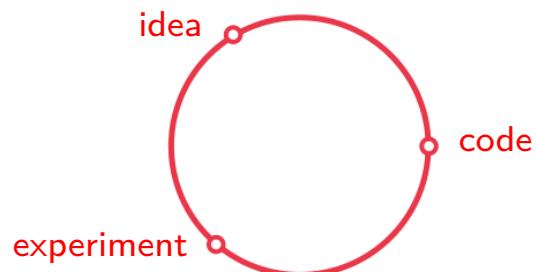
1. **uniform** distribution: $\epsilon \sim U(0.0001, 1)$
2. **log-uniform** distribution: $\epsilon' \sim U(\log 10^{-4}, \log 10^0) \rightarrow \epsilon = 10^{\epsilon'}$

better



Re-test hyperparameters occasionally

- 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

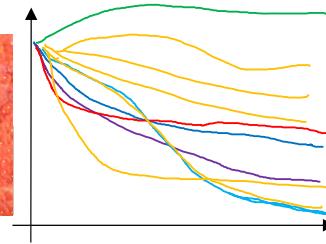
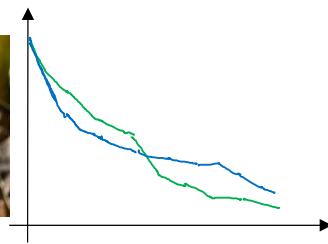


image source: Ng, Deep Learning (Coursera), <https://www.coursera.org/specializations/deep-learning>

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

Hyperparameter Tuning
Debugging Strategies

Data Scarcity and Transfer Learning

Summary

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 Simple → correct behavior actually can be predicted
 - ▶ a **test**
 - ▷ that exercises one part of implementation in **isolation**
- 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

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

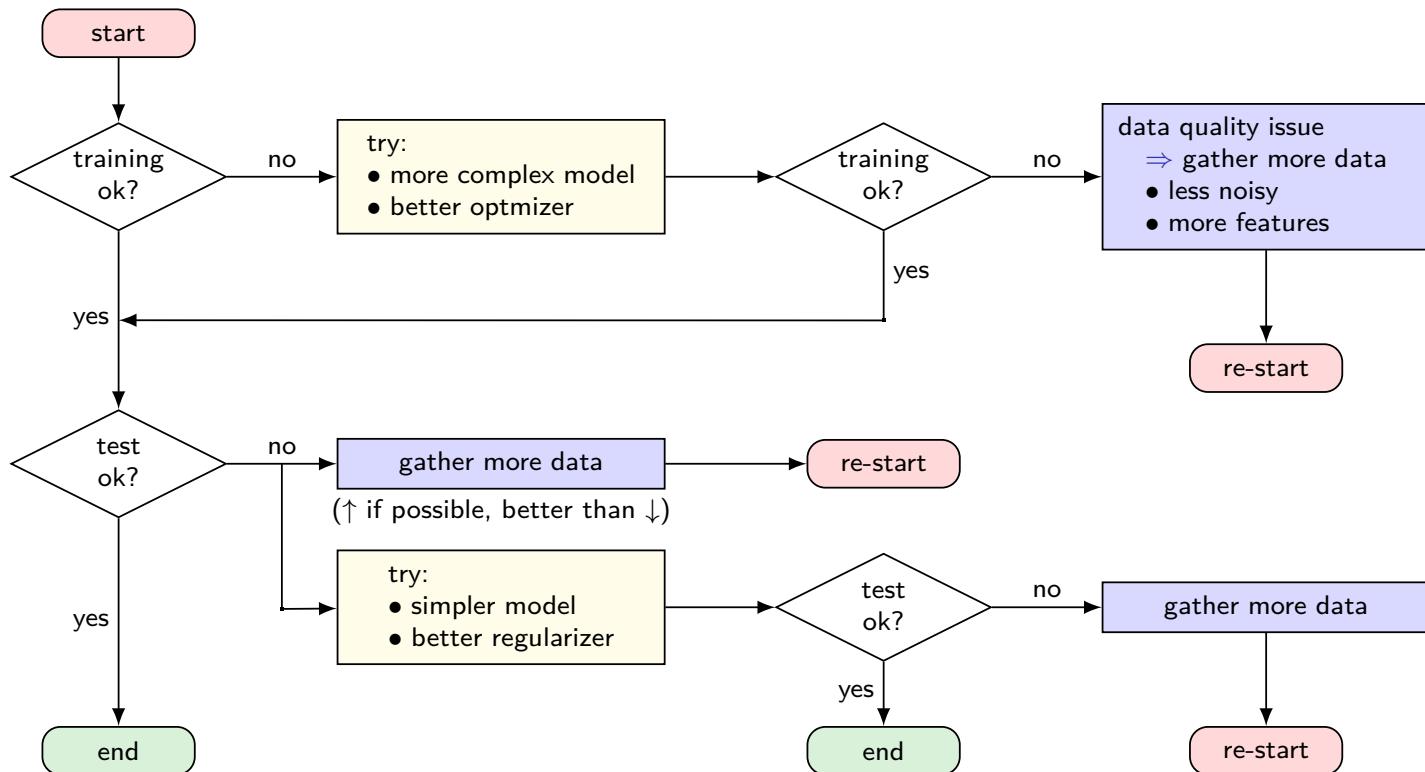
Data Scarcity and Transfer Learning

Summary

Data, data, data

- some datasets grow rapidly in size (faster than computing power)
 - ⇒ 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



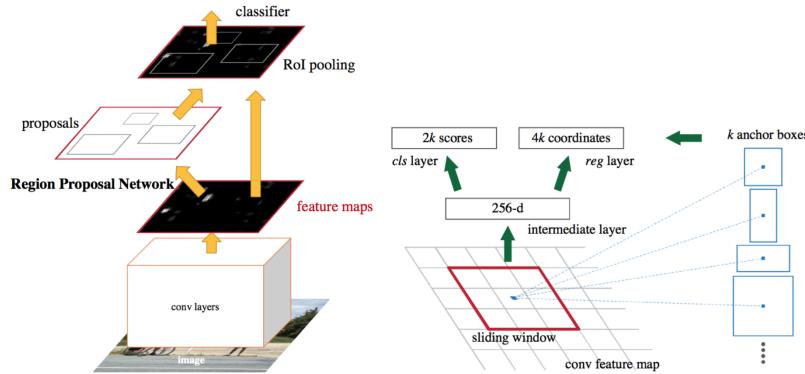
- how much more data? try data sizes on a **logarithmic scale**
e.g. double # of examples between experiments

Recycling models/algorithms

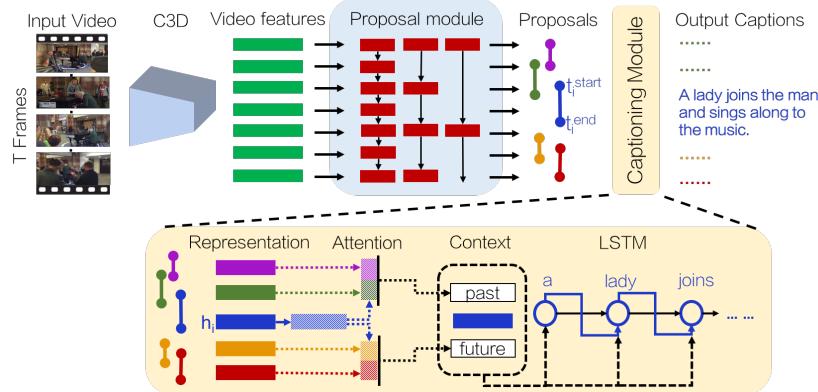
- if your task is similar to another task studied extensively
 - ▶ copy the model/algorithm known to perform best on that task
- you may even want to **copy a trained model** from that task
 - e.g. use features from a CNN to other tasks
 - ↑
trained on ImageNet
- Transfer learning and domain adaptation
 - ▶ 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 \leftrightarrow supervised task (page 68)

Examples

- transfer learning with CNNs: pervasive



► object detection
(faster R-CNN)



► video captioning

image sources: Ren, et al. "Faster R-CNN: Towards real-time object detection with region proposal networks." *Advances in Neural Information Processing Systems*. 2015; Krishna et al. "Dense-captioning events in videos." *Proceedings of the IEEE International Conference on Computer Vision*. 2017.

Model zoos

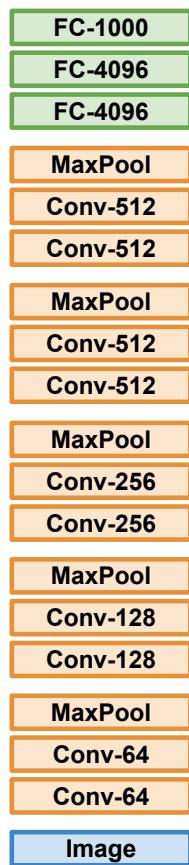
- many deep learning frameworks
 - ▶ provide a model zoo of [pretrained models](#)
- e.g. PyTorch [▶ Link](#)
- TensorFlow [▶ Link](#)
- Caffe [▶ Link](#)
- ▶ before start training from scratch, should consider using them



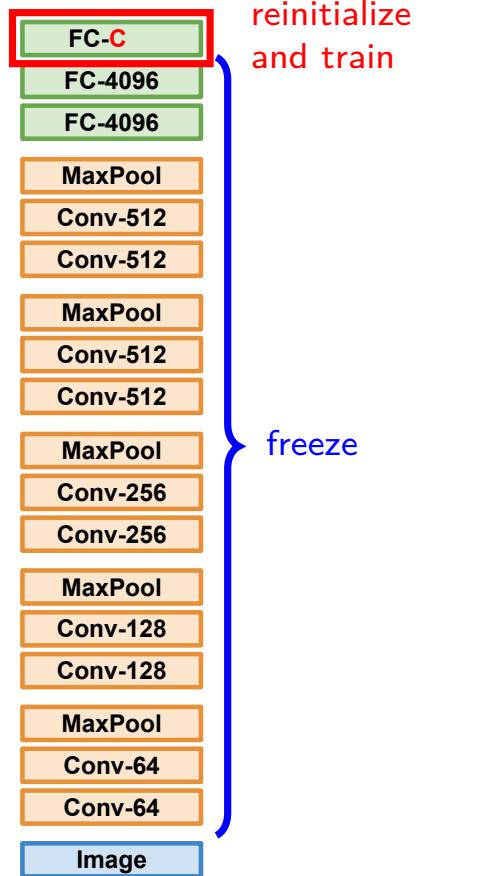
image source: <https://images.app.goo.gl/NVfdKcZR4XXEsc8i7>

Recycling CNN features

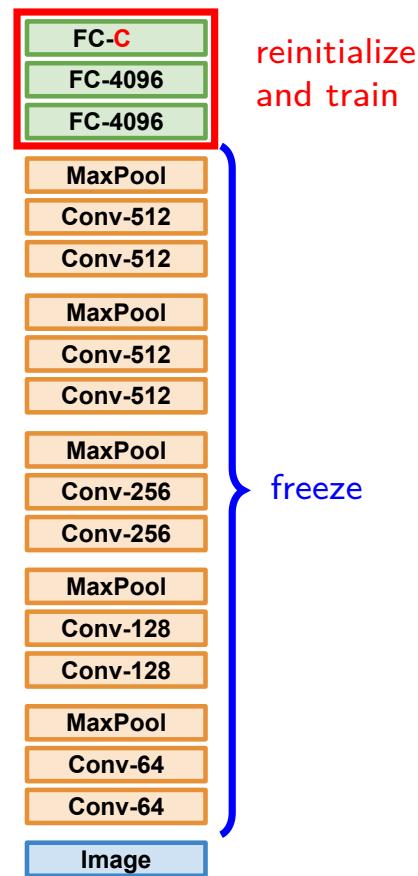
- pretrained



- new task: small data



- new task: bigger data



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

image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017)

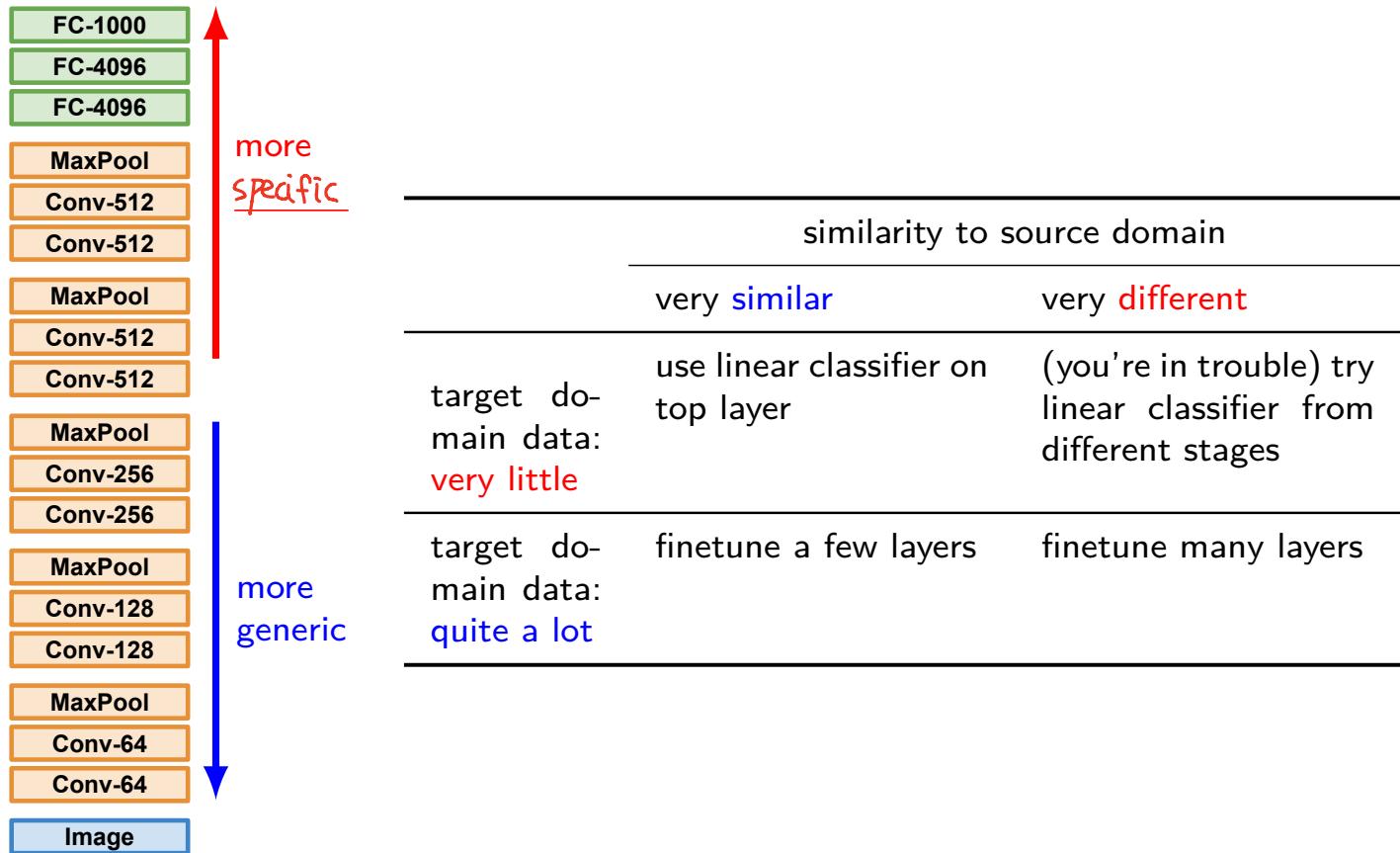


image source: Fei-Fei Li, J. Johnson, S. Yeung, CS231n: Convolutional Neural Networks for Visual Recognition (2017),
<http://cs231n.stanford.edu/2017/index.html>

Beginning by unsupervised learning

- a common question:
 - ▶ whether to begin by using unsupervised learning
 - ▶ answer: domain specific
- some domains (such as **NLP**):
 - ▶ 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

- 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 짐장 훈남”



“긴 뒷걸음 청바지 그네”



image sources: <https://images.app.goo.gl/oXfzL9qVJJ8kjsbXA>; <https://images.app.goo.gl/UuXgLdKp3i4i39L38>

Outline

Introduction

Preprocessing and Initialization

Normalization

Monitoring and Tuning Training

Data Scarcity and Transfer Learning

Summary

Summary

- neural net training: often extremely time-consuming
 - ⇒ 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 ← will be covered separately
- phase 2: monitor training dynamics
 - ▶ parameter updates (optimization) ← will be covered separately
 - ▶ batch normalization to alleviate covariate shift issue
 - ▶ hyperparameter 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