8. Training Neural Networks

GEV6135 Deep Learning for Visual Recognition and Applications

Kibok Lee

Assistant Professor of

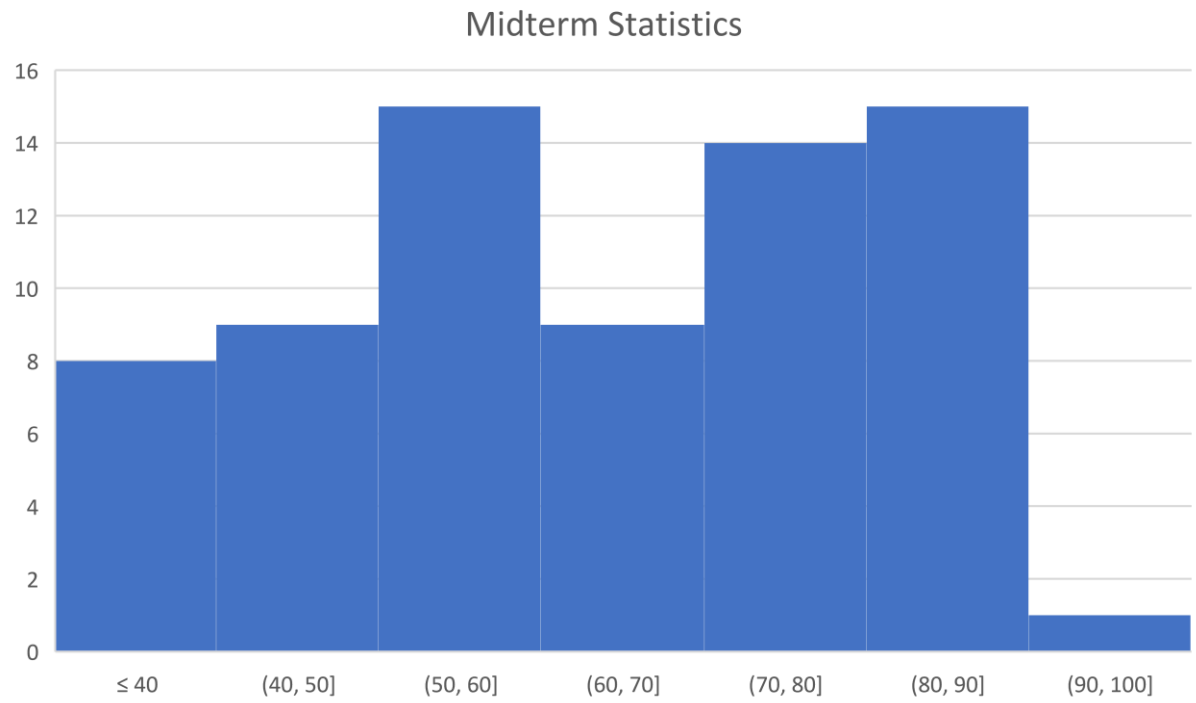
Applied Statistics / Statistics and Data Science

Oct 27, 2022

연세대학교
YONSEI UNIVERSITY

Midterm Statistics

- Mean ± std: 65.7 ± 13.9
- Median: 66



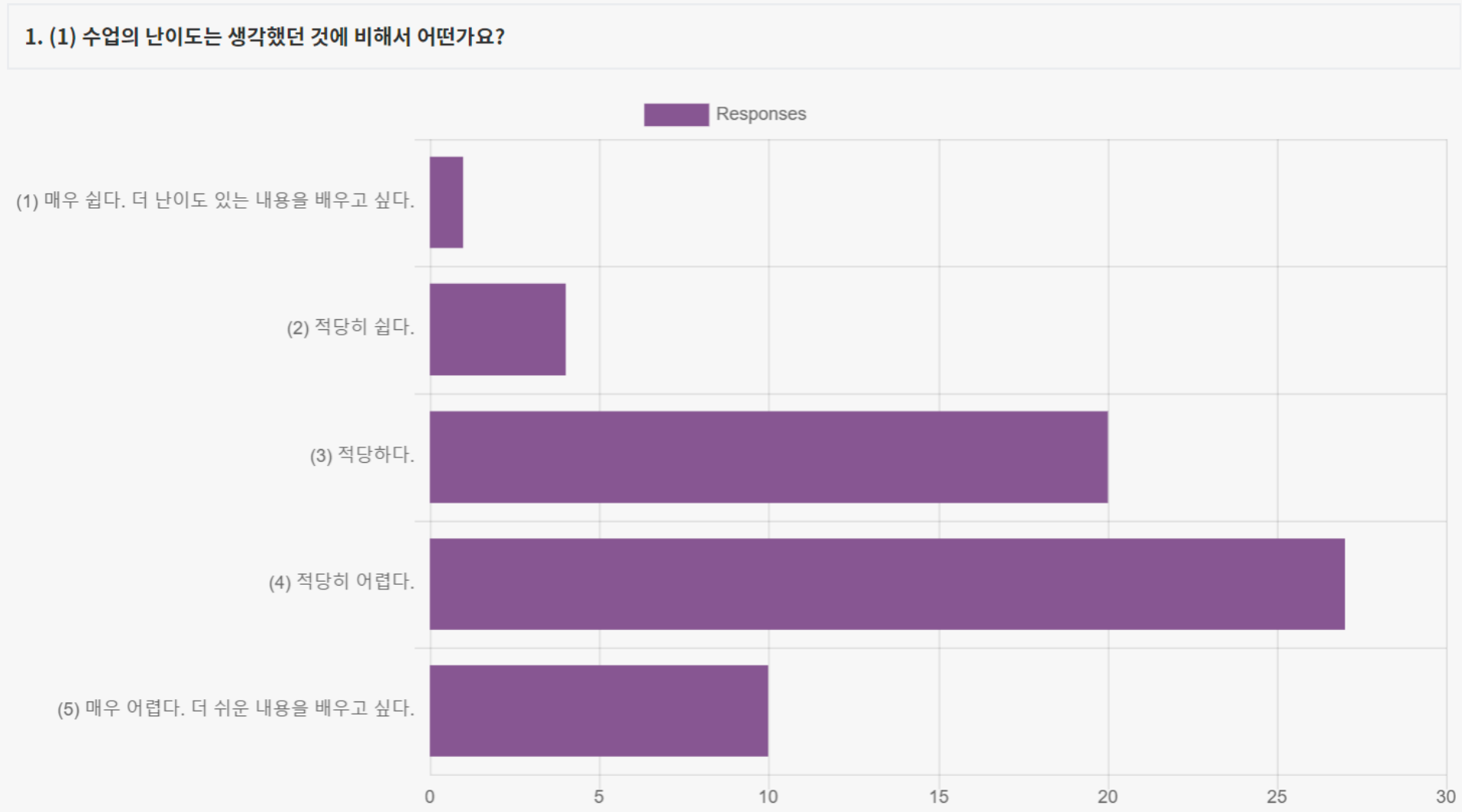
- Midterm: 15% of final score
- Final grade: (Conditional) absolute evaluation
  - Scaling up based on a curve if the statistics is low

Assignment 5

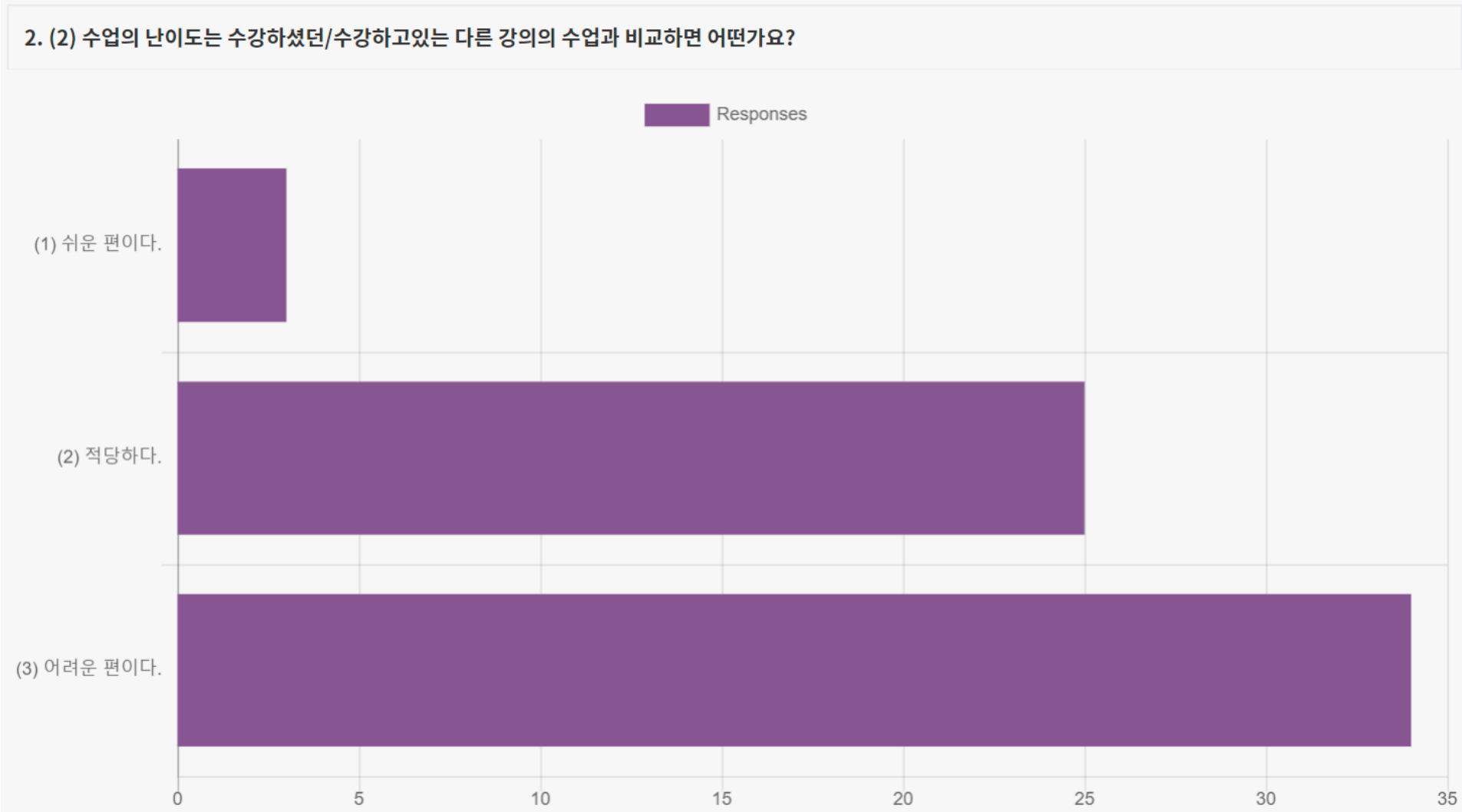
- Will be out around next week?
- Fully-connected networks
  - Modularized implementation (loss will be given!)
  - Dropout

- Before submitting your work, we recommend you
  - Re-download clean files
  - Copy-paste your solution to clean py
  - Re-run clean ipynb only once
- If you feel difficult, consider to take **option 2**.

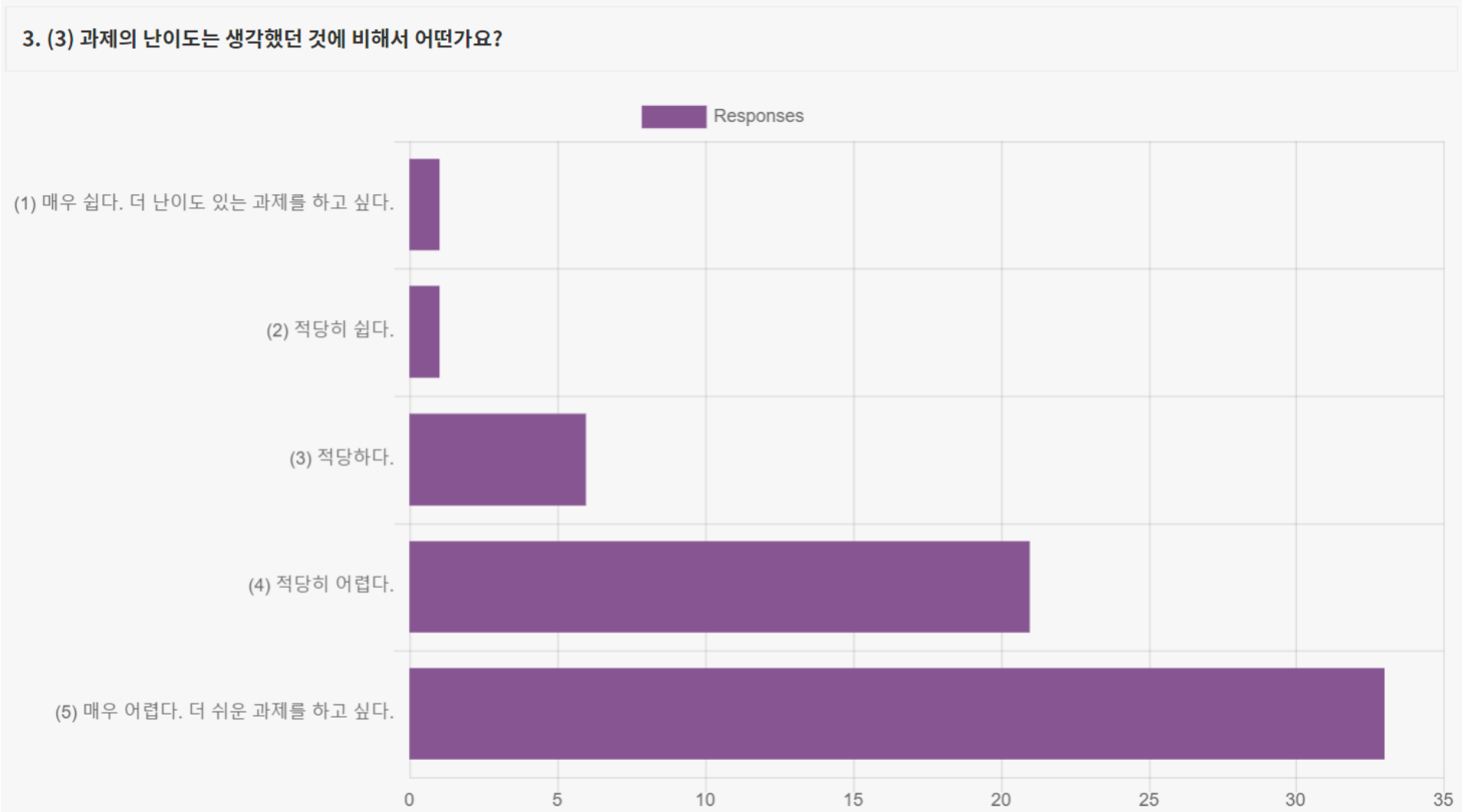
Survey Results: Q1



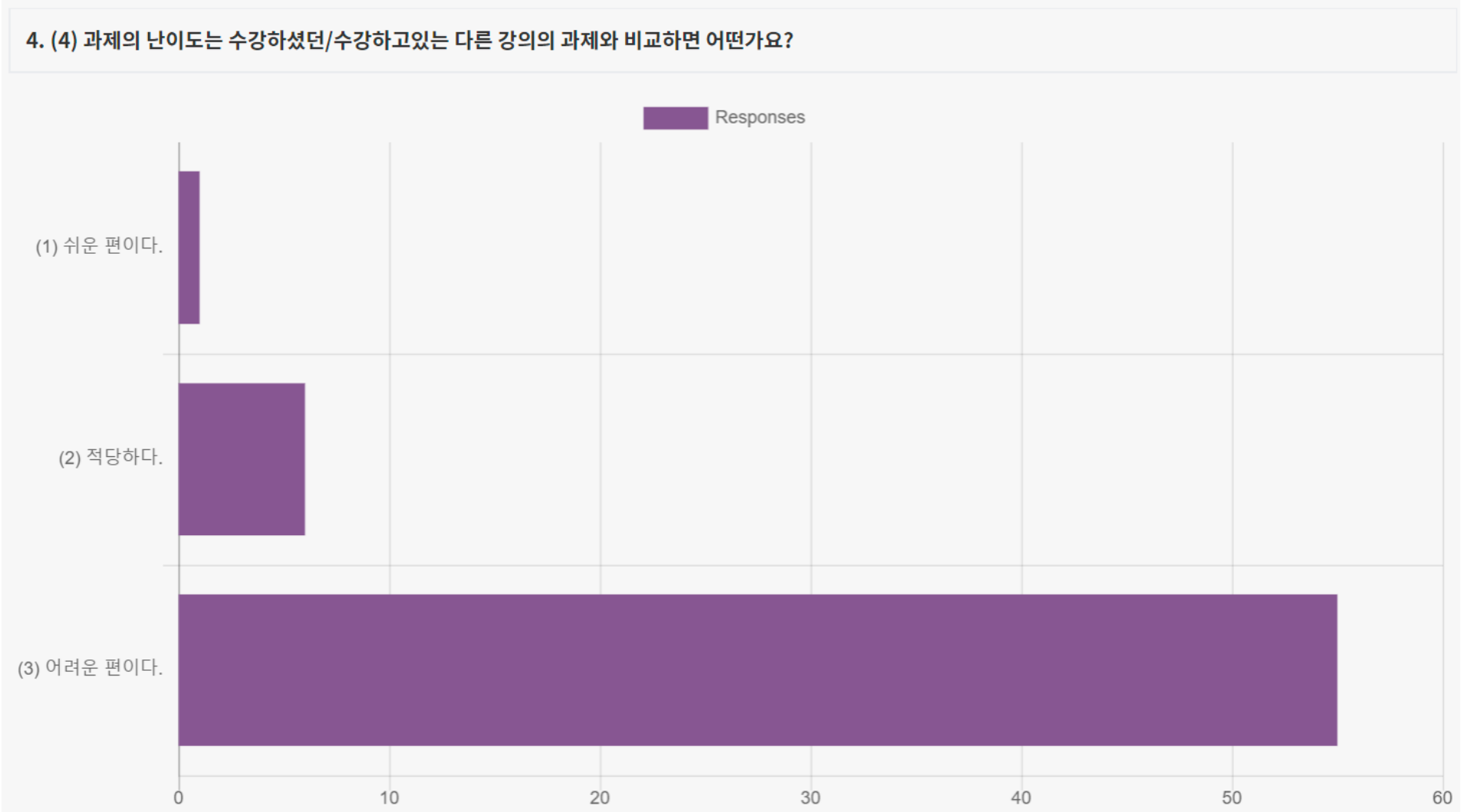
Survey Results: Q2



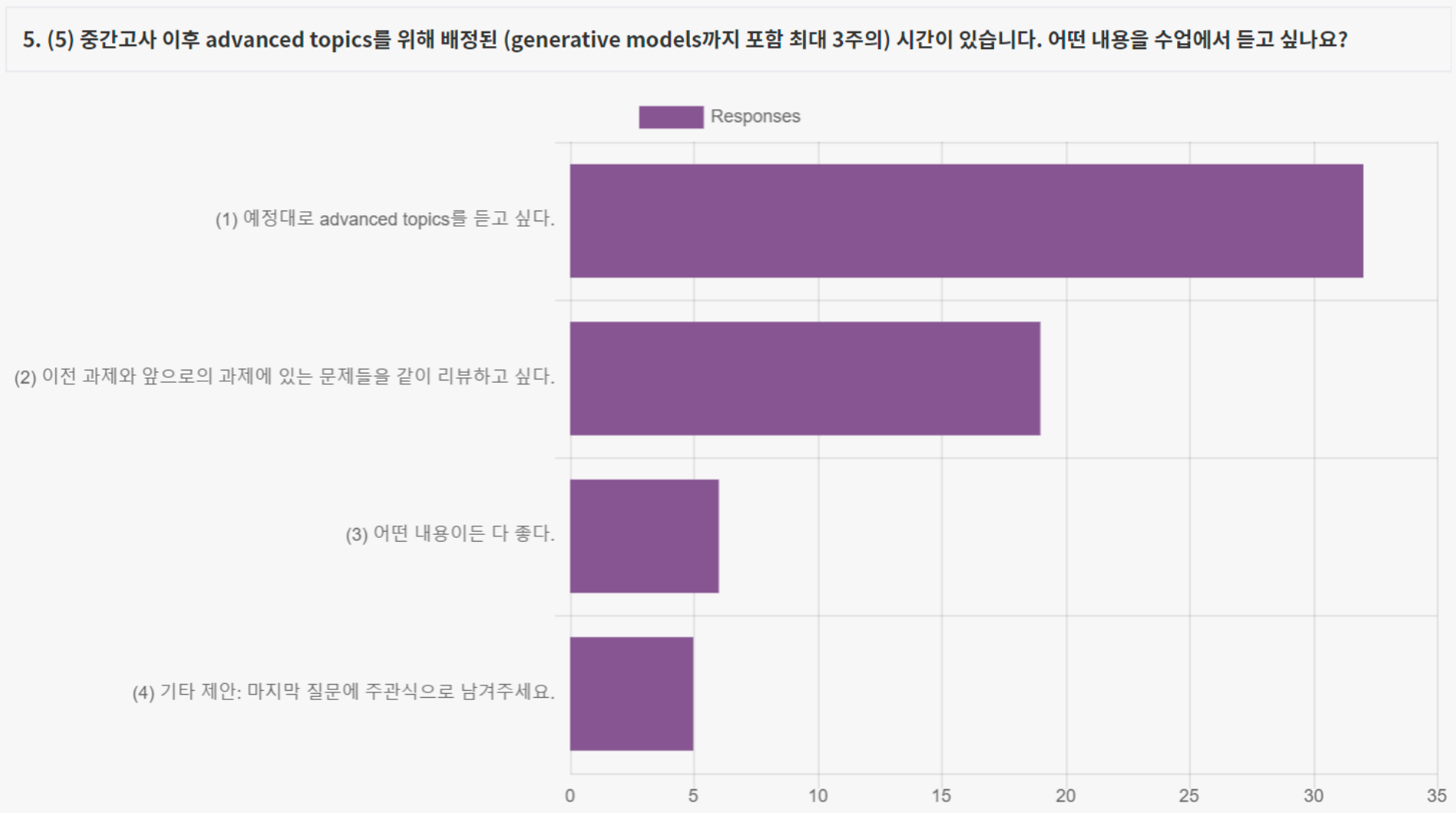
Survey Results: Q3



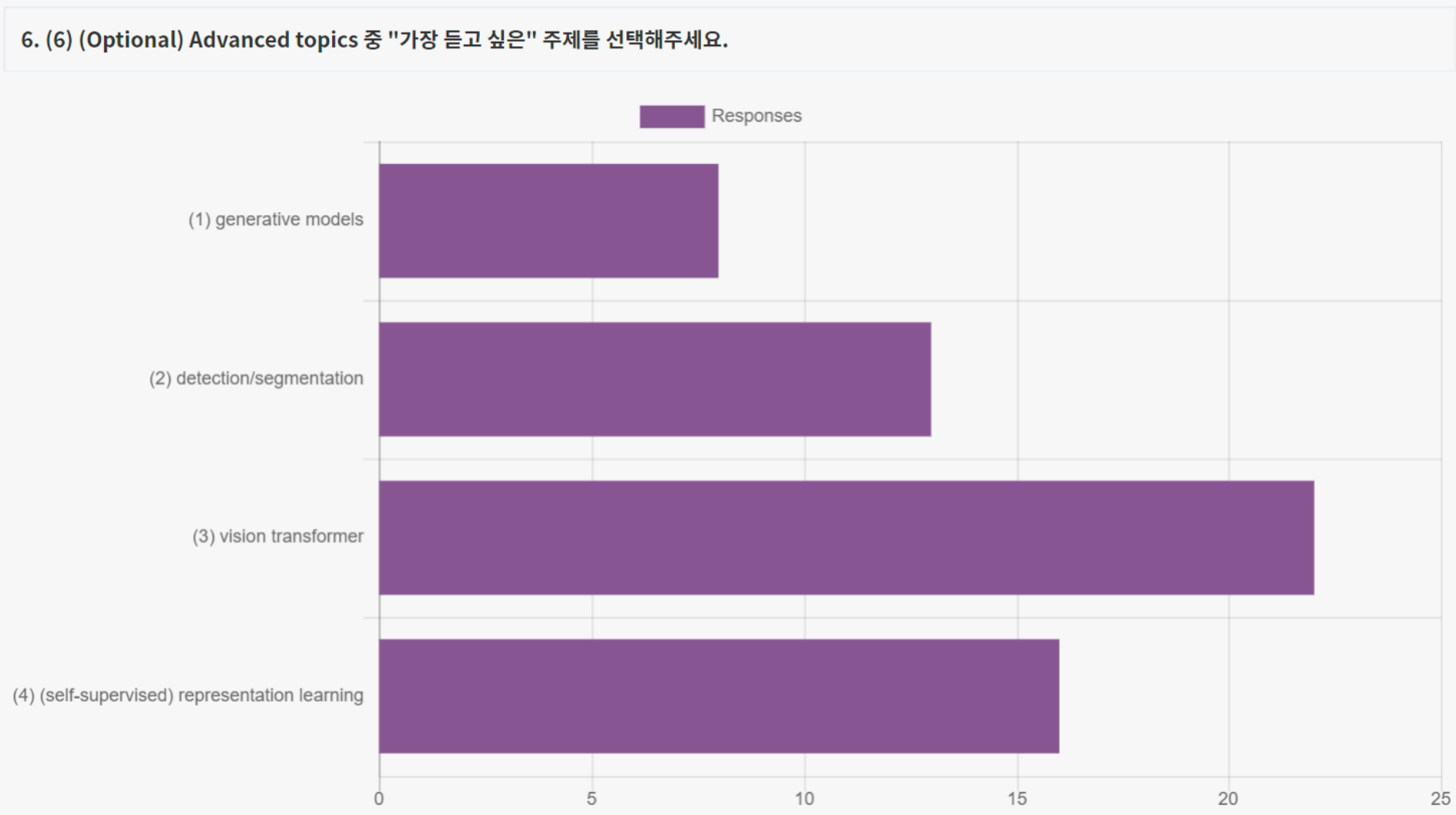
Survey Results: Q4



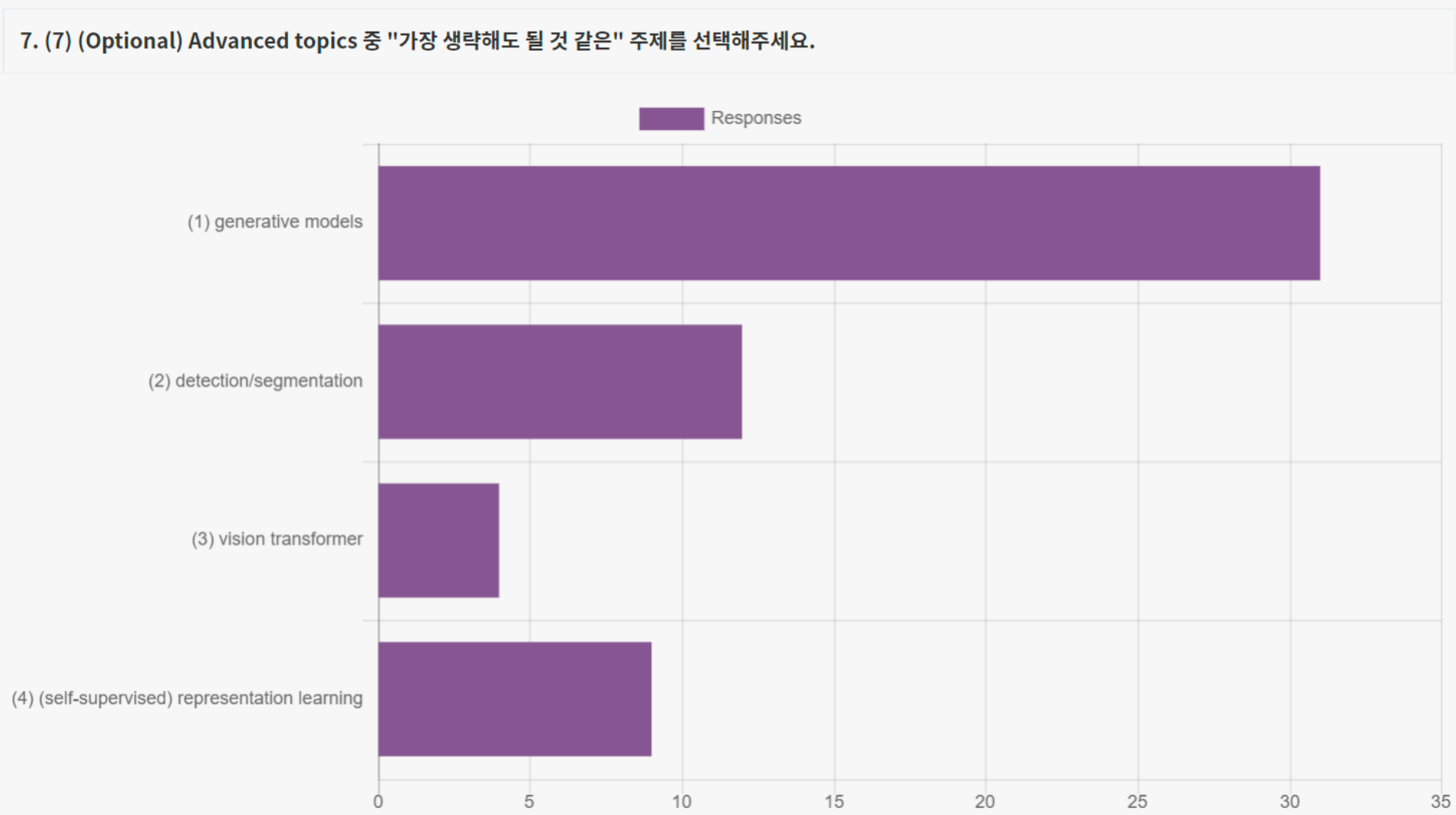
Survey Results: Q5



Survey Results: Q6



Survey Results: Q7



Survey Results: Q8

- Will discuss next week

- Note: feedback for assignments is available through

Grade/Attendance -> Grades

Schedule

- We have 15 weeks in total
- 3 intro + midterm + final
- 6 DL basics
  - Image classification, Linear classifiers, Optimization,
  - Neural Networks, Backpropagation, Convolutional Neural Networks
- 3 Practical & general topics
  - Training NNs (we are here), CNN architectures, DL software
- 3 Advanced topics (generative models -> GEK6194 GANs by Prof. Uh)
  - Object detection and image segmentation
  - Vision transformer
  - Representation learning

Schedule

- New survey: final exam vs. extended Lecture 8

- A: final exam
  - Assignments 60%, Midterm 15%, Final 15%, Attendance 10%
  - No class @ week 15
  - Final exam @ week 16

- B: extended Lecture 8
  - Assignments 70%, Midterm 20%, Attendance 10%
  - All schedules shifted & have a lecture @ week 15
  - No class @ week 16

Overview

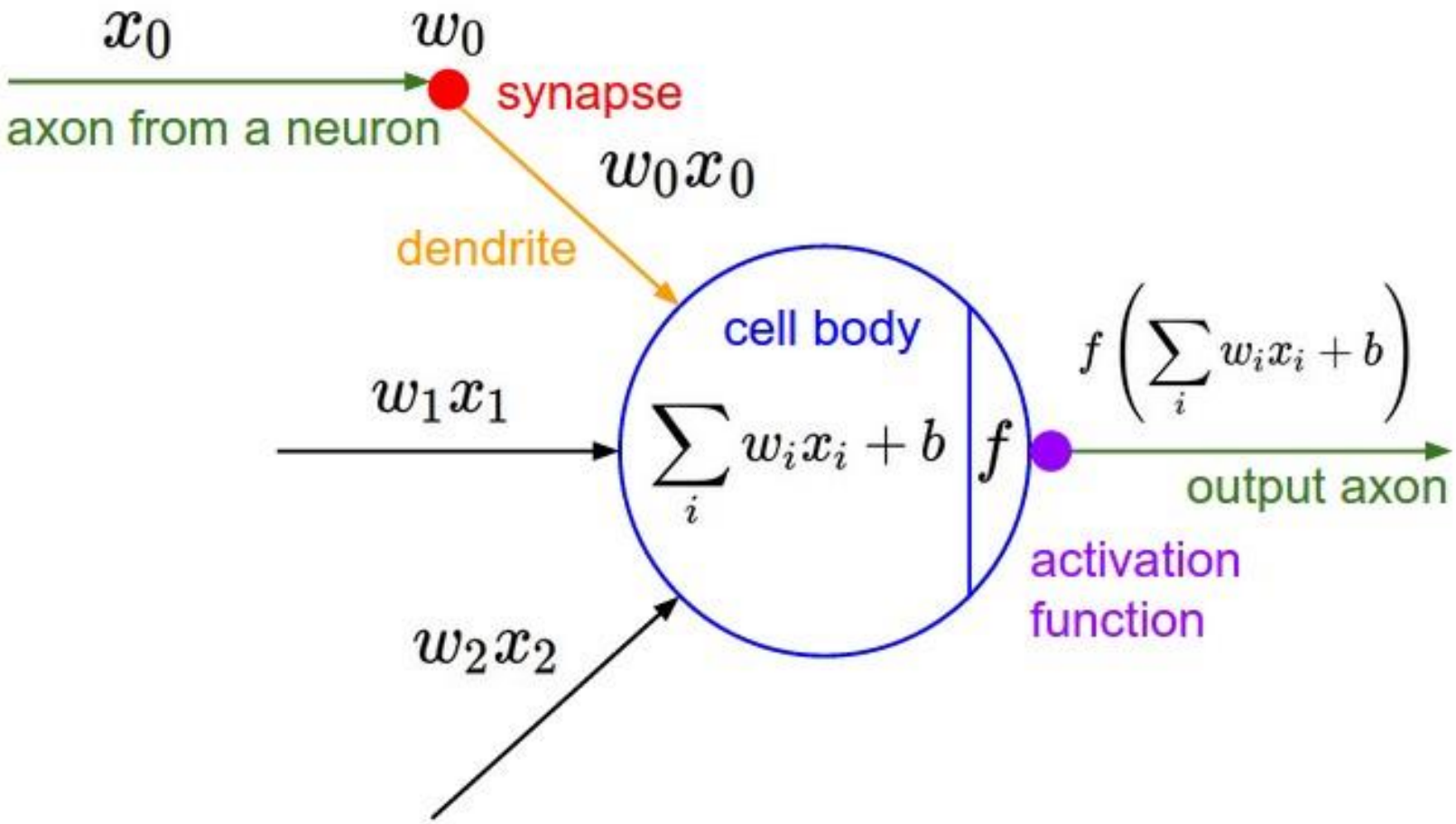
1. **One time setup**
   Activation functions, data preprocessing,
   weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules, large-batch training,
   hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

Overview

1. **One time setup**
   Activation functions, data preprocessing, weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules, large-batch training, hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

Activation Functions

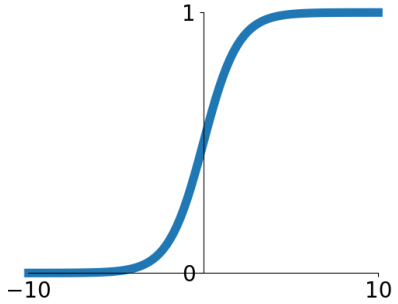
Activation Functions



Activation Functions

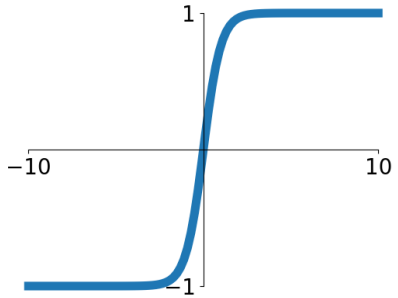
Sigmoid

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



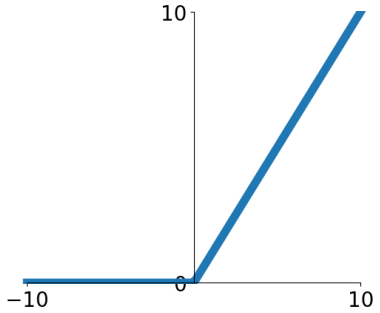
tanh

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$



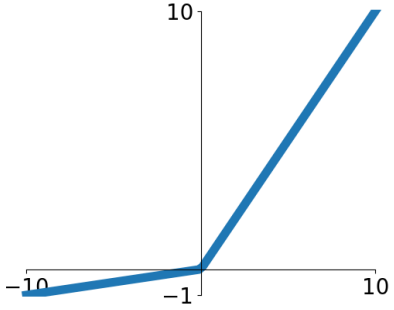
ReLU

$$\max(0, x)$$



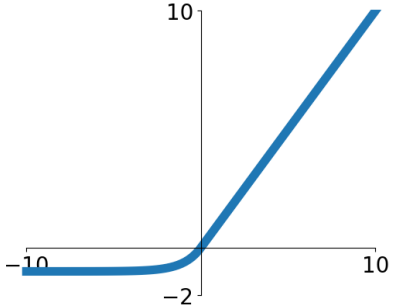
Leaky ReLU

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



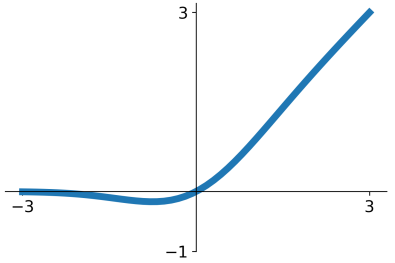
ELU

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

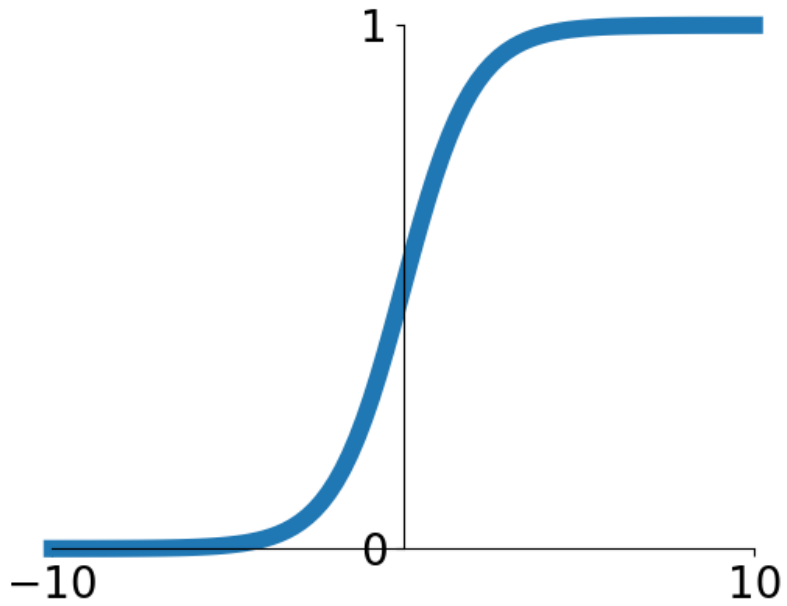


GELU

$$= 0.5x\left[1 + \text{erf}(x/\sqrt{2})\right]$$
$$\approx x\sigma(1.702x)$$



Activation Functions: Sigmoid



Sigmoid

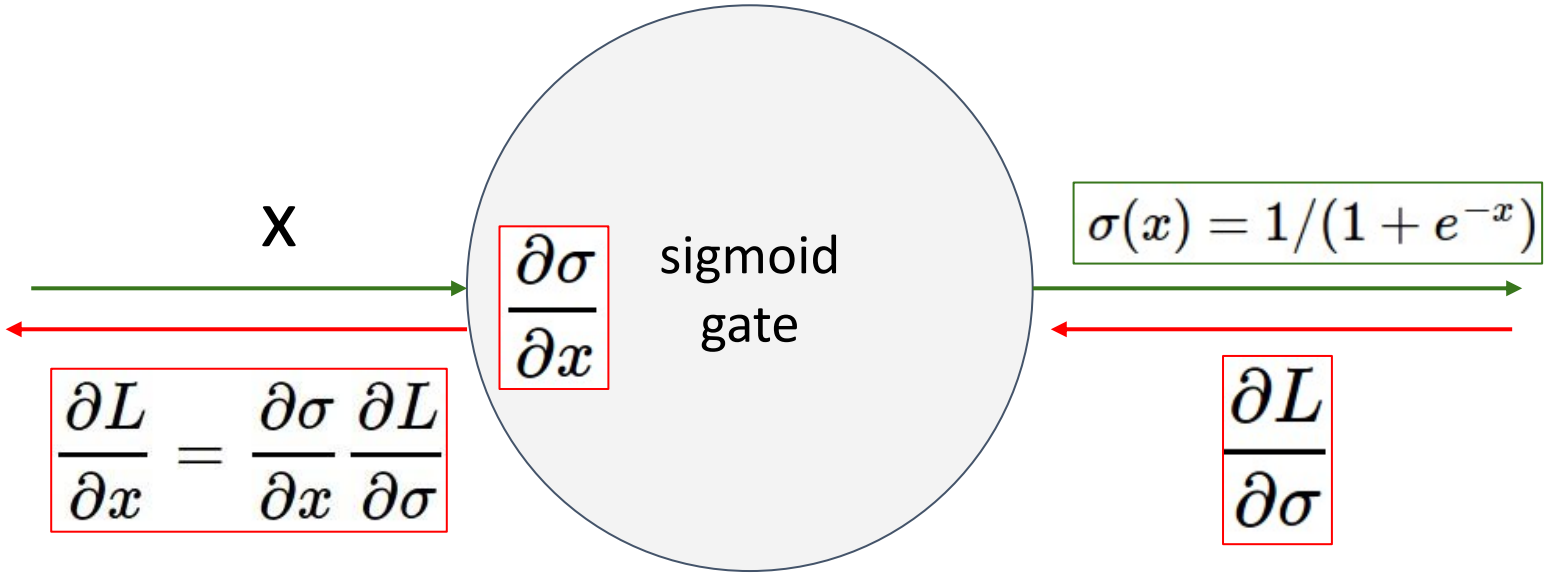
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

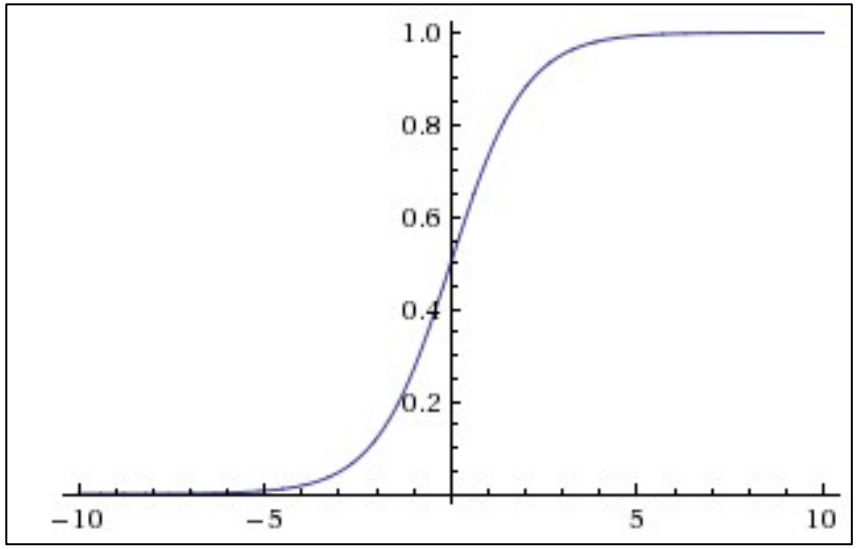
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients

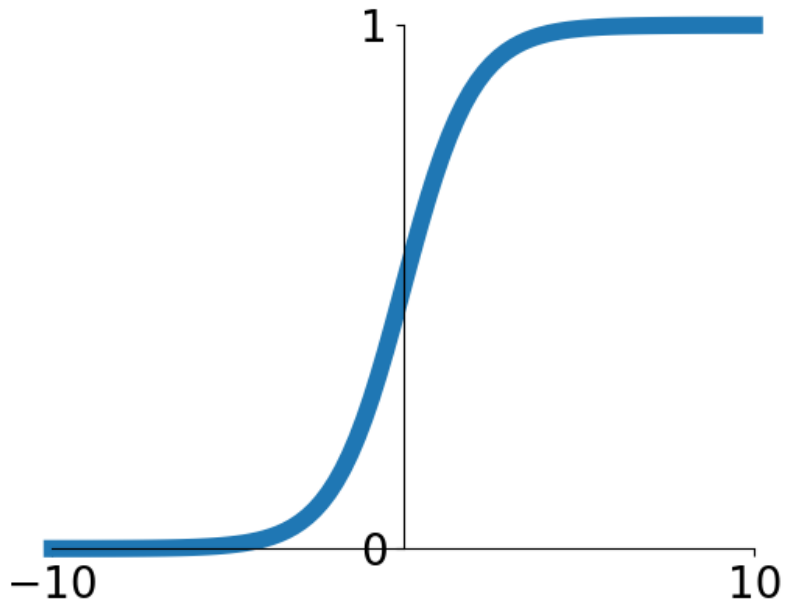
Activation Functions: Sigmoid





What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Activation Functions: Sigmoid



Sigmoid

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma\left(h_j^{(\ell-1)}\right) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, \mathrm{b}^{(\ell)}$ are the weights and bias of layer ℓ

Local Gradient / Upstream Gradient

$$\frac{\partial L}{\partial w_{i,j}^{(\ell)}} = \frac{\partial h_i^{(\ell)}}{\partial w_{i,j}} \cdot \frac{\partial L}{\partial h_i^{(\ell)}}$$

$$= \sigma\left(h_j^{(\ell-1)}\right) \cdot \frac{\partial L}{\partial h_i^{(\ell)}}$$

What can we say about the gradients on $w^{(\ell)}$?

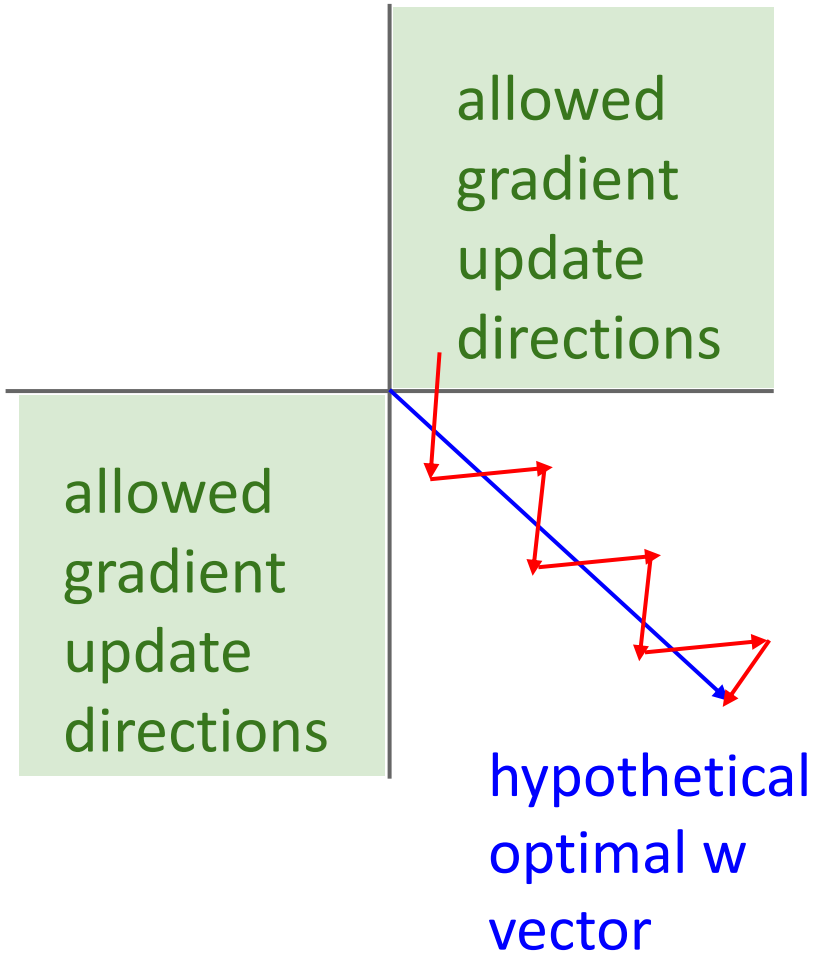
Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma\left(h_j^{(\ell-1)}\right) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, \mathrm{b}^{(\ell)}$ are the weights and bias of layer ℓ



What can we say about the gradients on $w^{(\ell)}$?

Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

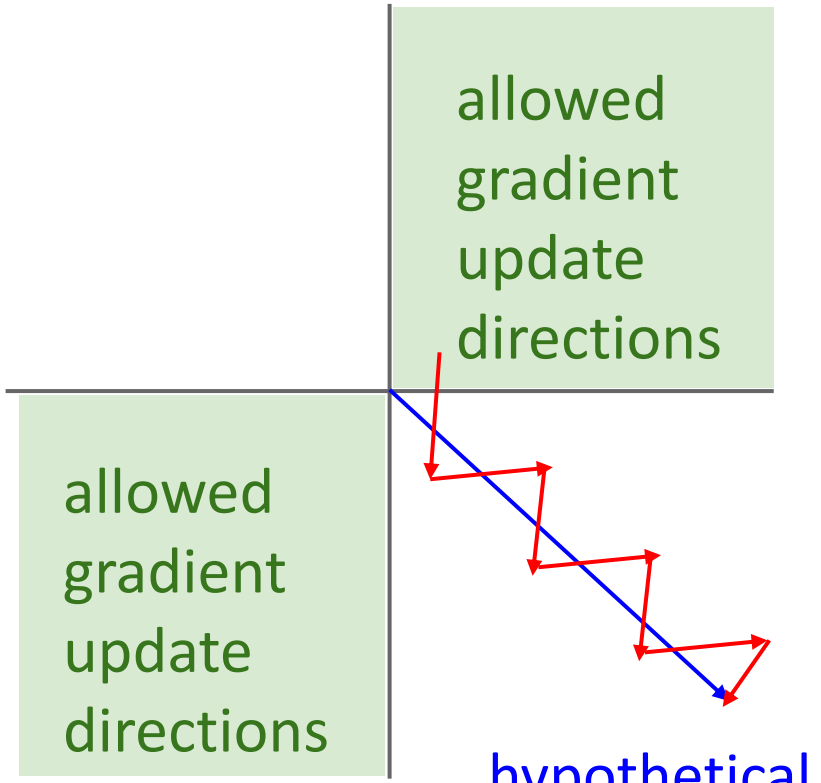
Gradients on rows of w can only point in some directions; needs to "zigzag" to move in other directions

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma\left(h_j^{(\ell-1)}\right) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)

$w^{(\ell)}, \mathrm{b}^{(\ell)}$ are the weights and bias of layer ℓ

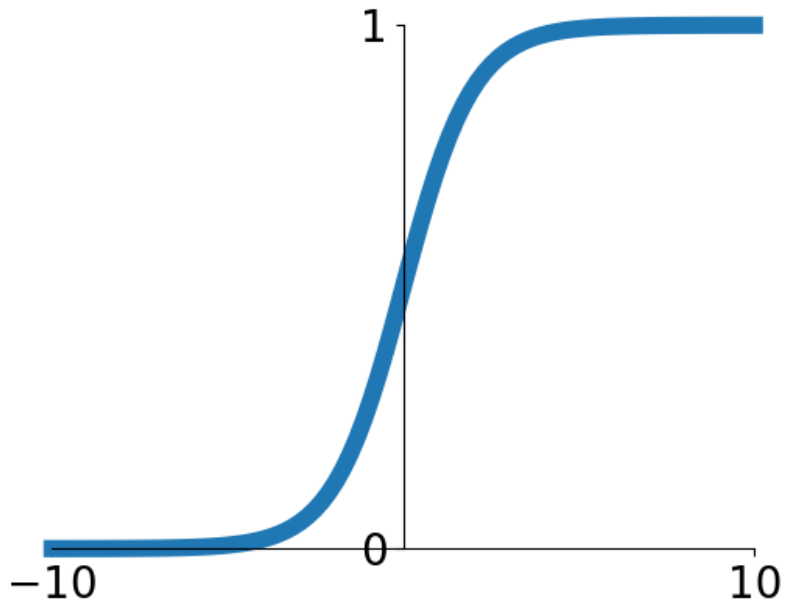


What can we say about the gradients on $w^{(\ell)}$?

Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

Not that bad in practice:
- Only true for a single example, minibatches help
- BatchNorm can also avoid this

Activation Functions: Sigmoid



Sigmoid

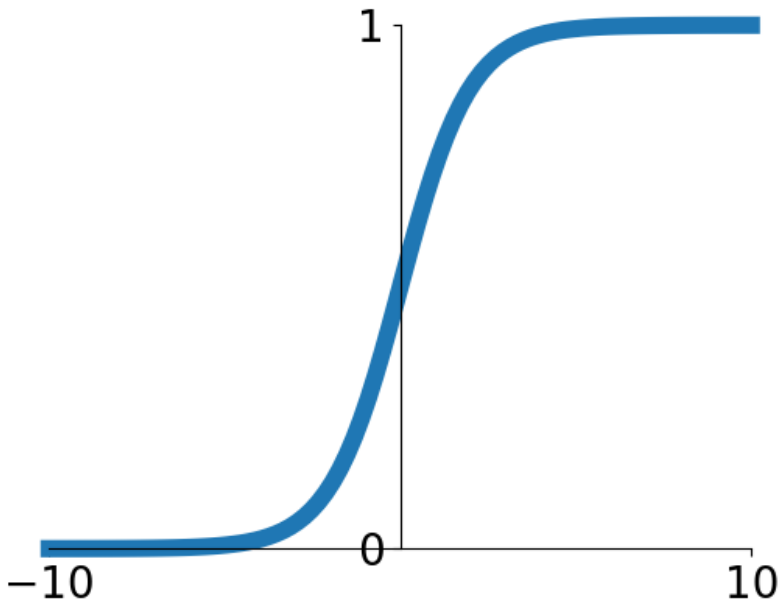
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions: Sigmoid



Sigmoid

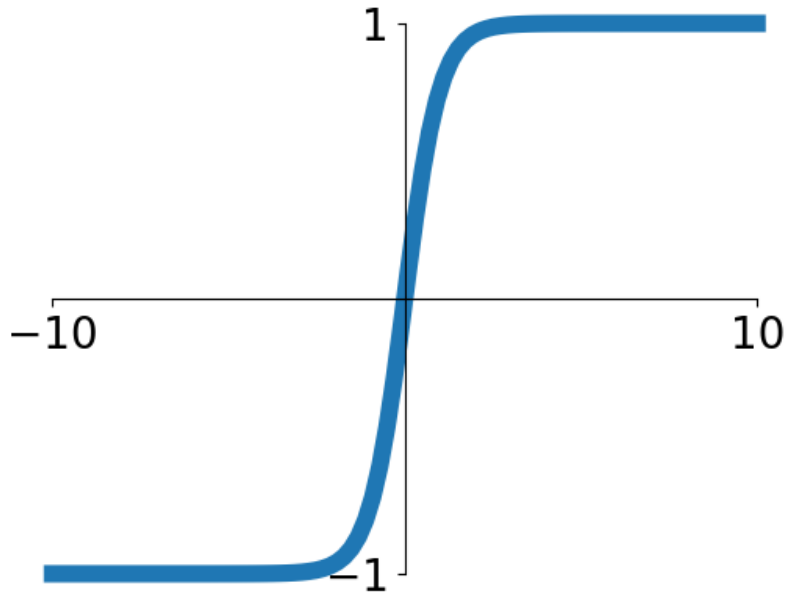
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems: **Worst problem in practice**

1. **Saturated neurons "kill" the gradients**
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Functions: Tanh



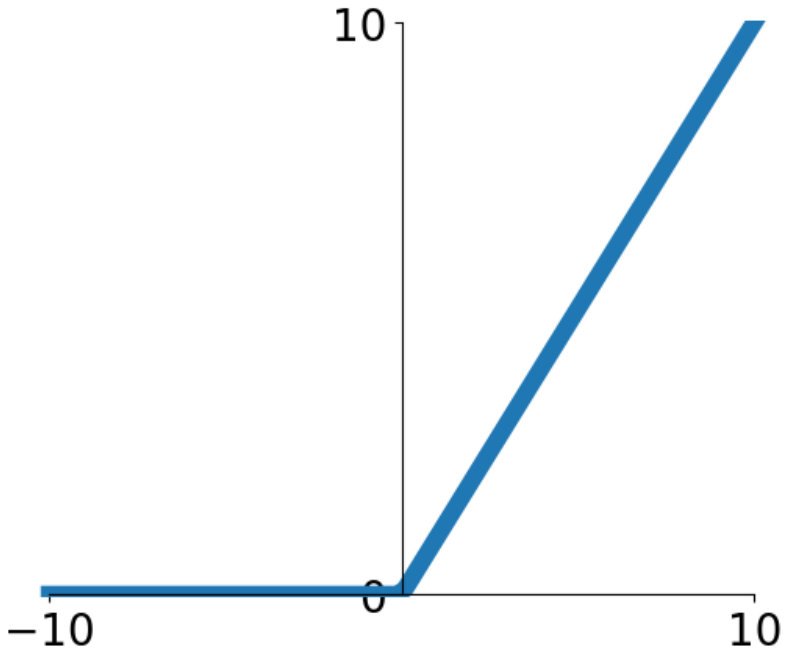
tanh(x)

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

Activation Functions: ReLU

$$f(x) = \max(0,x)$$



ReLU

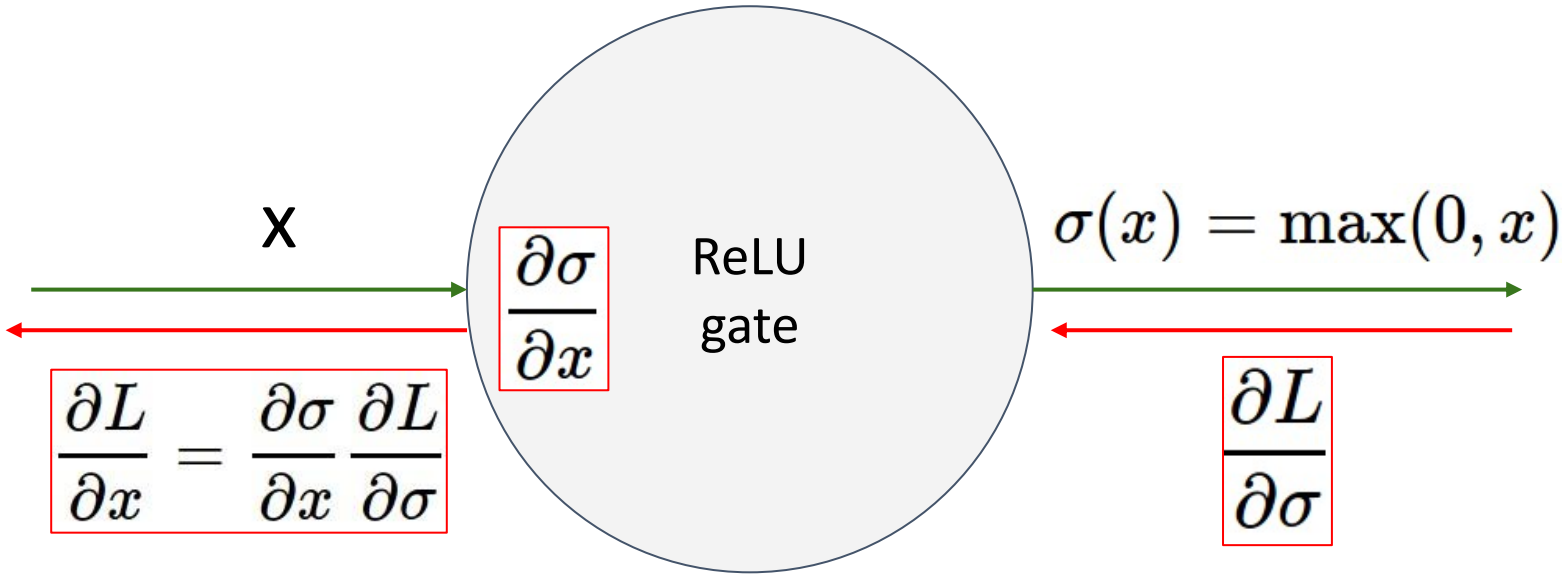
(Rectified Linear Unit)

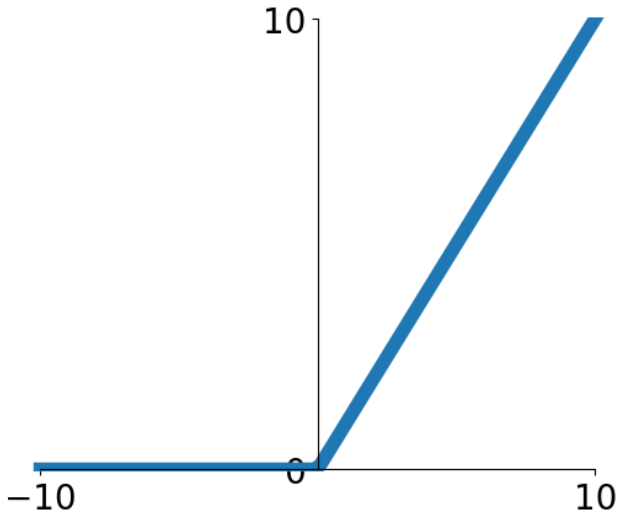
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

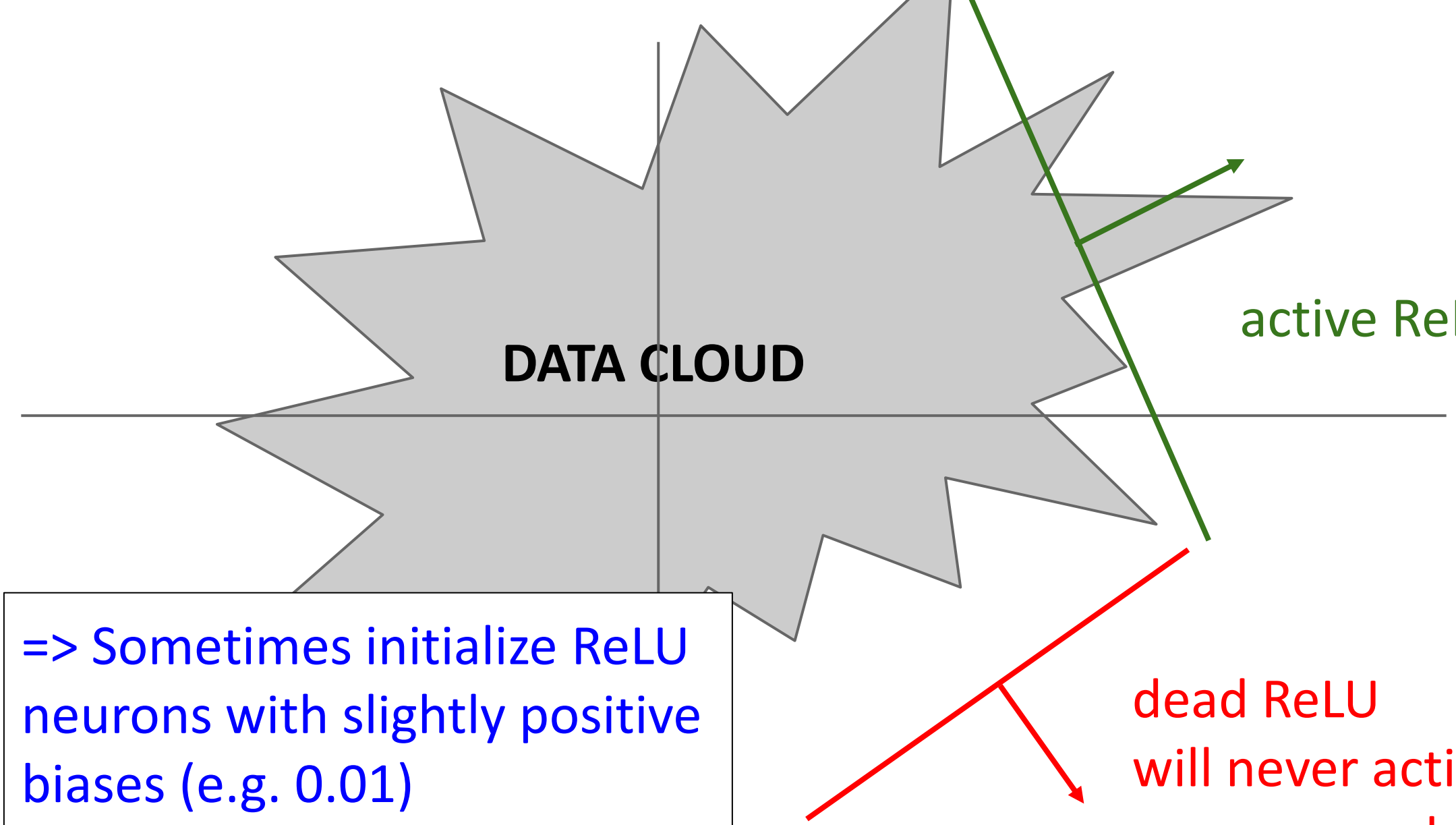
hint: what is the gradient when x < 0?

Activation Functions: ReLU



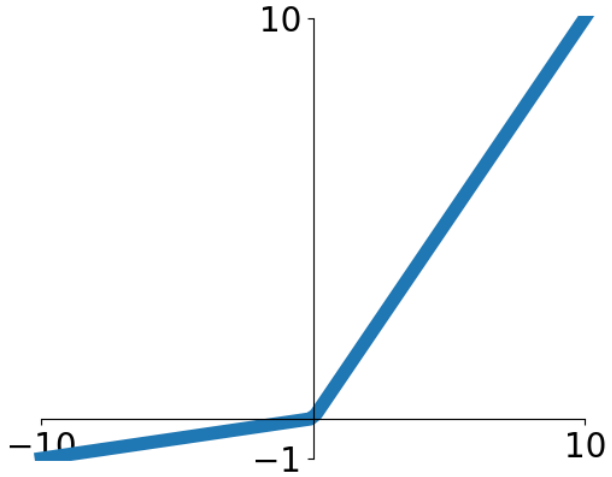


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?



=> Sometimes initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Activation Functions: Leaky ReLU



Leaky ReLU

$f(x) = \max(\alpha x, x)$

α is a hyperparameter, often $\alpha = 0.1$

Maas et al, "Rectifier Nonlinearities Improve Neural Network Acoustic Models", ICML 2013

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

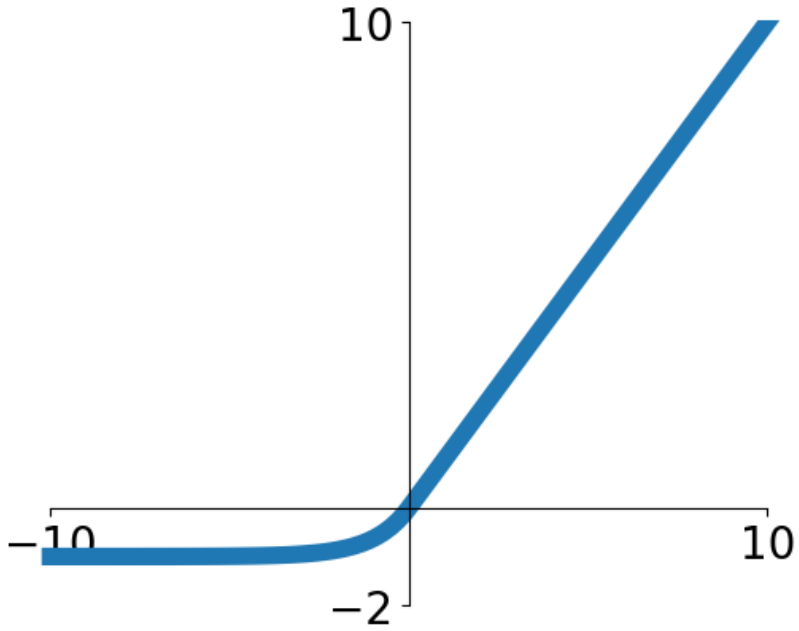
Parametric ReLU (PReLU)

$f(x) = \max(\alpha x, x)$

α is learned via backprop

He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Activation Functions: Exponential Linear Unit (ELU)



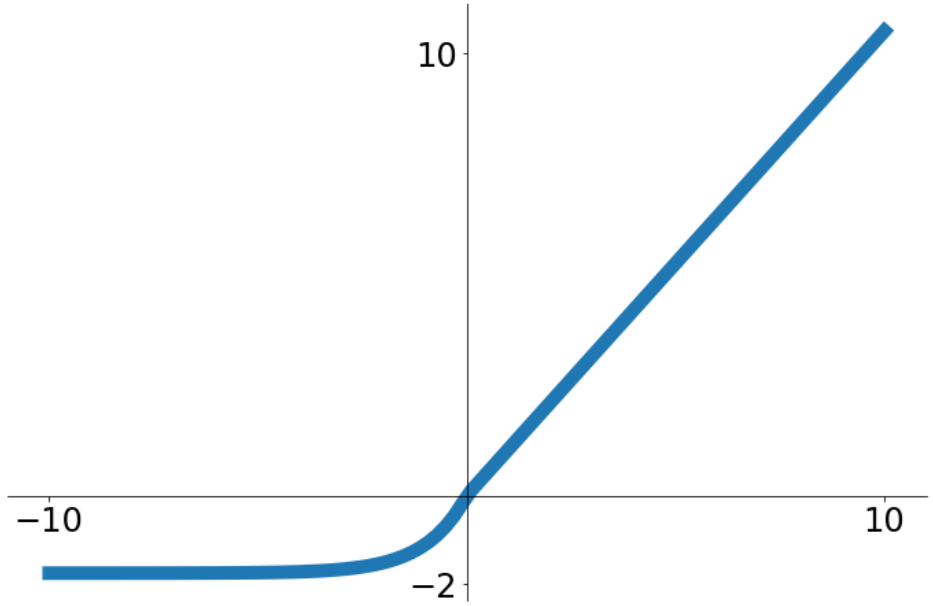
$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & if\ x \leq 0 \end{cases}$$

(Default alpha=1)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Activation Functions: Scaled Exponential Linear Unit (SELU)



- Scaled version of ELU that works better for deep networks
- "Self-Normalizing" property; can train deep SELU networks without BatchNorm

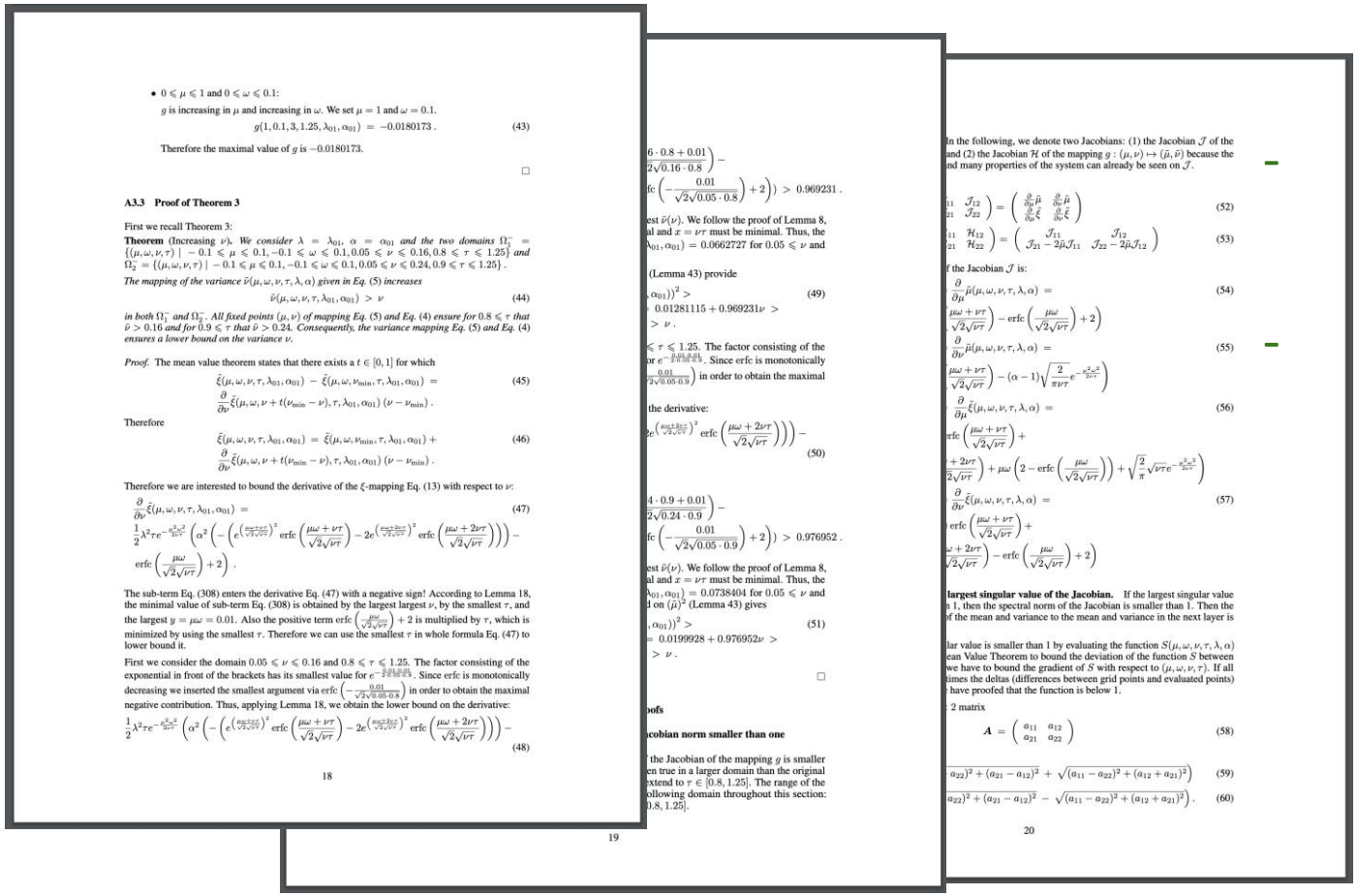
Designed for fully-connected networks; not working well with CNNs on ImageNet

$$selu(x) = \begin{cases} \lambda x & if\ x > 0 \\ \lambda\alpha(e^x - 1) & if\ x \leq 0 \end{cases}$$

$$\alpha = 1.67326324235437728481704299167 17$$
$$\lambda = 1.05070098735548049341933498529 46$$

Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017

Activation Functions: Scaled Exponential Linear Unit (SELU)



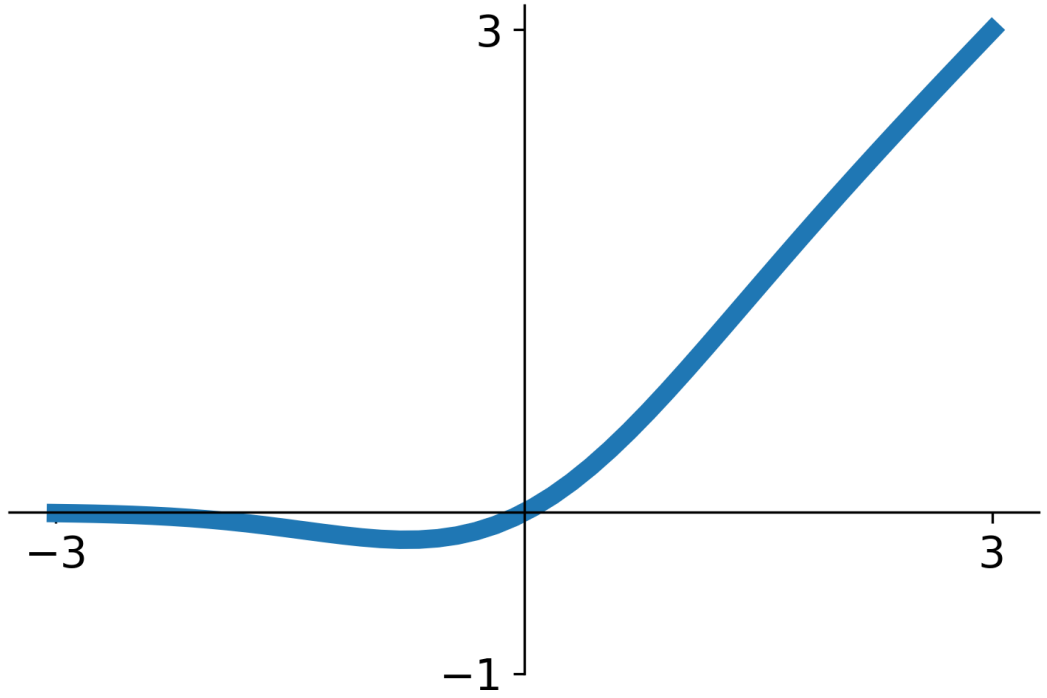
- Scaled version of ELU that works better for deep networks
- "Self-Normalizing" property; can train deep SELU networks without BatchNorm

Derivation takes
91 pages of math
in appendix…

$$\alpha = 1.6732632423543772848170429916717$$
$$\lambda = 1.0507009873554804934193349852946$$

Klambauer et al, Self-Normalizing Neural Networks, ICLR 2017

Activation Functions: Gaussian Error Linear Unit (GELU)



$$X \sim N(0,1)$$

$$gelu(x) = xP(X \leq x) = \frac{x}{2}\left(1 + \text{erf}\left(x/\sqrt{2}\right)\right)$$

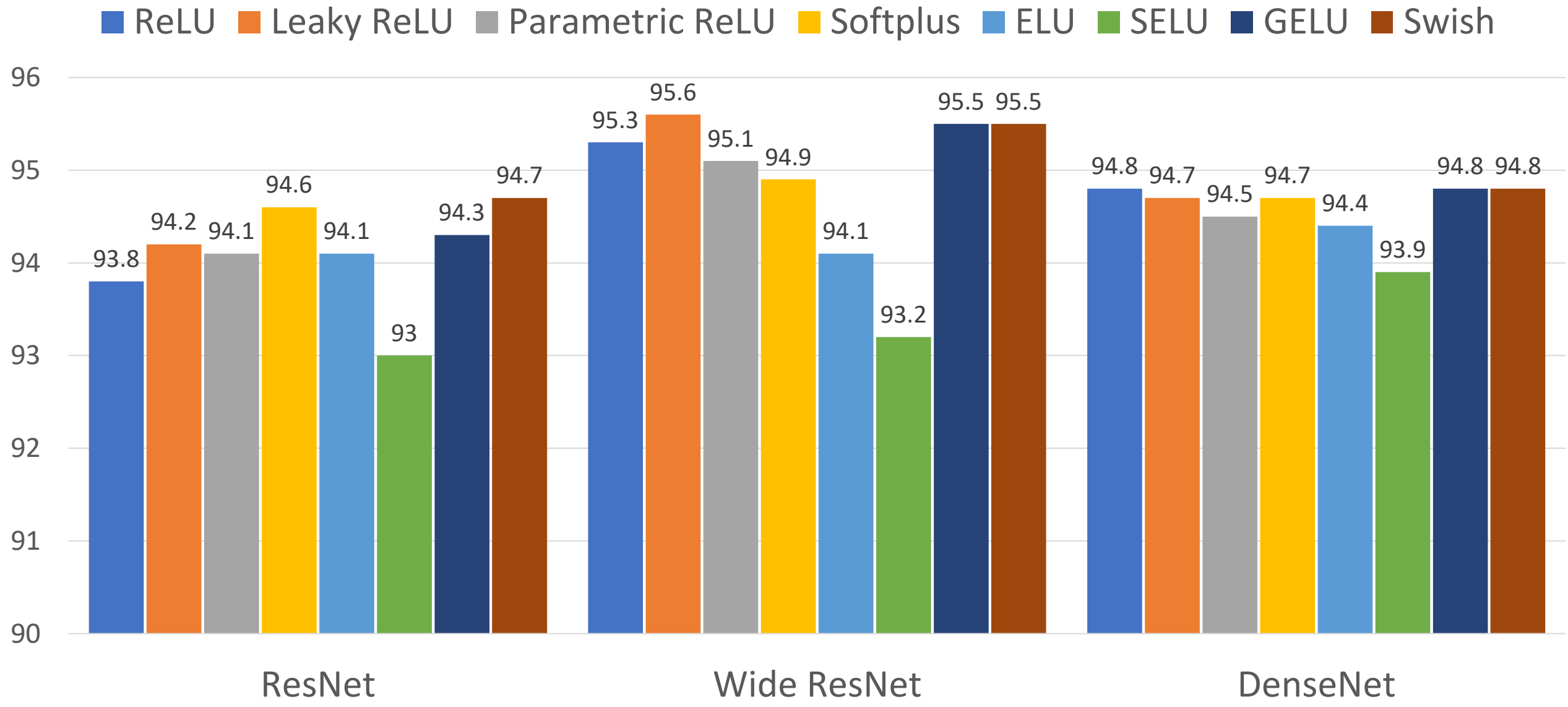
$$\approx x\sigma(1.702x)$$

- **Idea**: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, ViT)

Hendrycks and Gimpel, Gaussian Error Linear Units (GELUs), 2016

Accuracy on CIFAR10

Ramachandran et al, "Searching for activation functions", ICLR Workshop 2018



Activation Functions: Summary

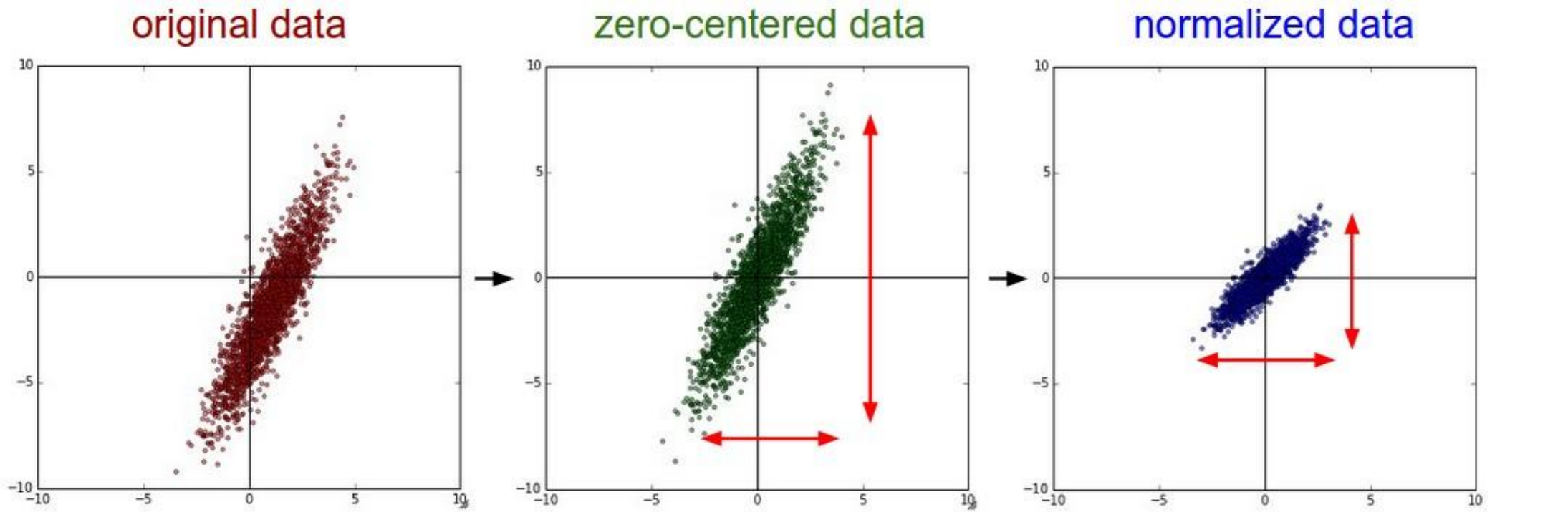
- Don’t think too hard. Just use ReLU
- Try out Leaky ReLU / ELU / SELU / GELU
  if you need to squeeze that last 0.1%
- Don’t use sigmoid or tanh

Some (very) recent architectures use GeLU
instead of ReLU, but the gains are minimal

Dosovitskiy et al, “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”, ICLR 2021
Liu et al, “A ConvNet for the 2020s”, arXiv 2022

Data Preprocessing

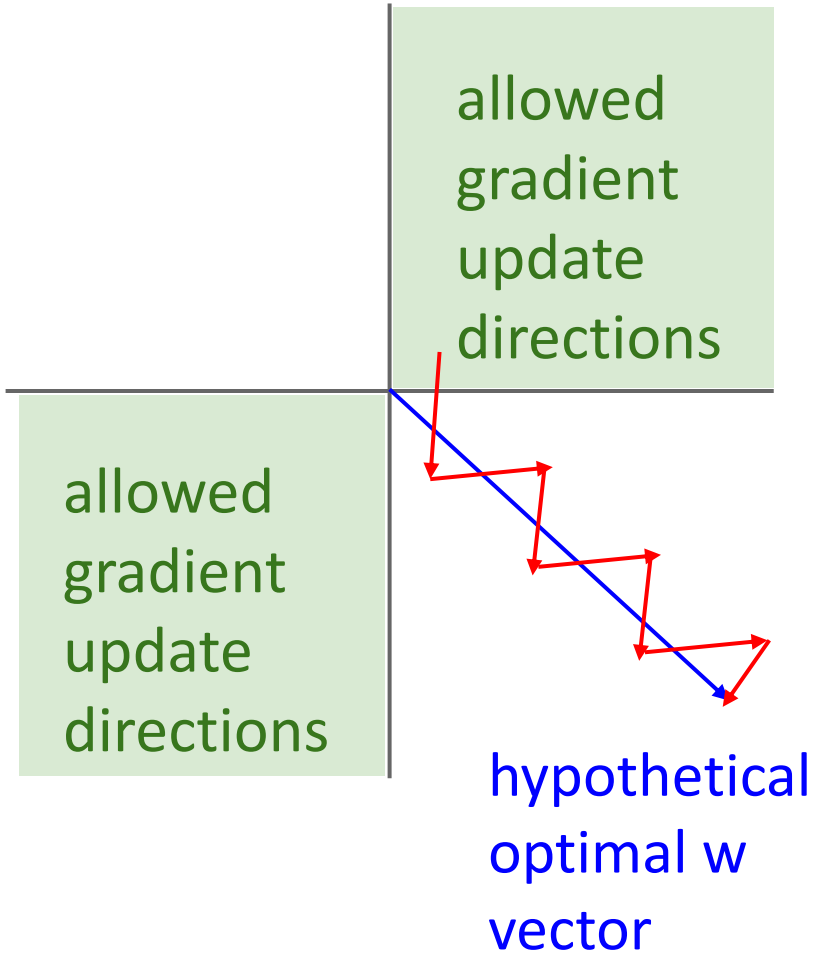
Data Preprocessing



(Assume X [NxD] is data matrix,
each example in a row)

Remember: Consider what happens when the input to a neuron is always positive...

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma\left(h_j^{(\ell-1)}\right) + b_i^{(\ell)}$$



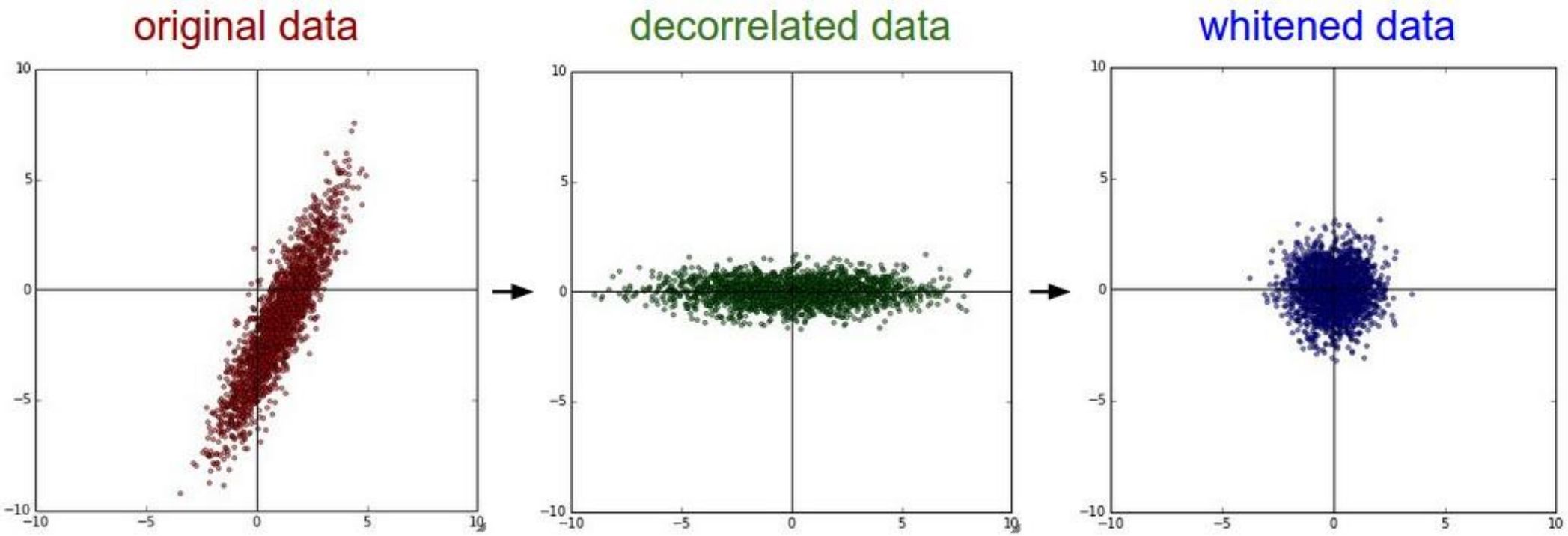
What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Data Preprocessing

In practice, you may also see **PCA** and **Whitening** of the data

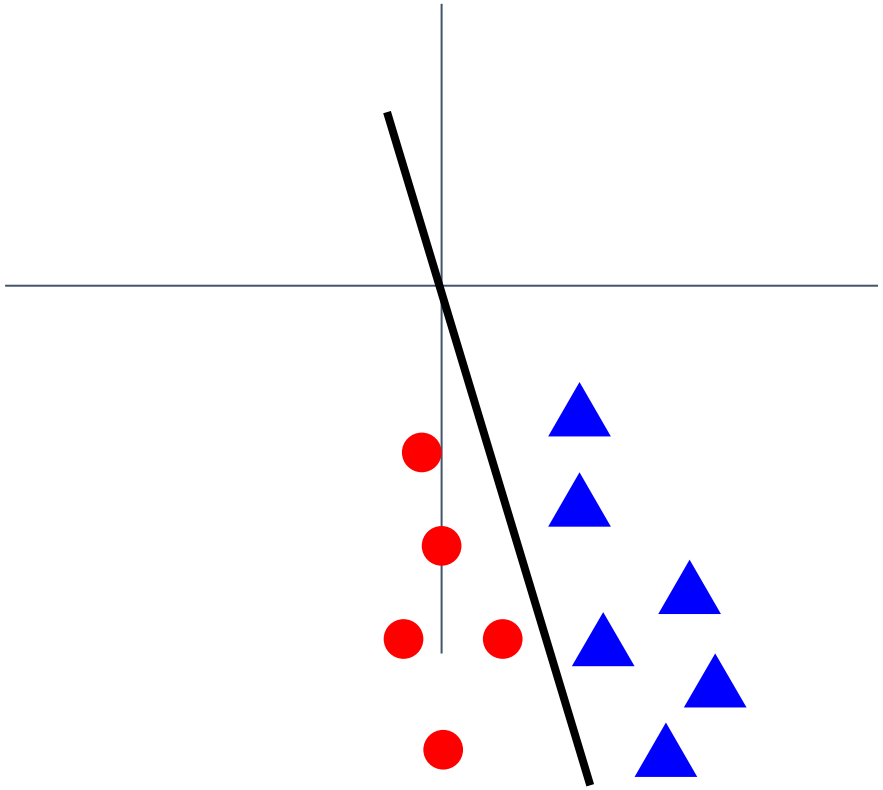


(data has diagonal covariance matrix)

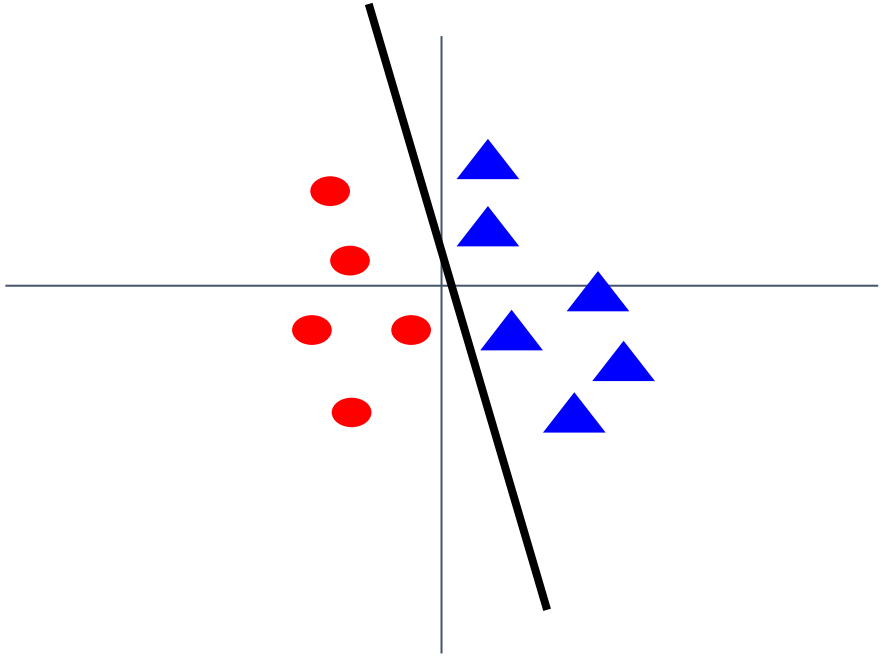
(covariance matrix is the identity matrix)

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data Preprocessing for Images

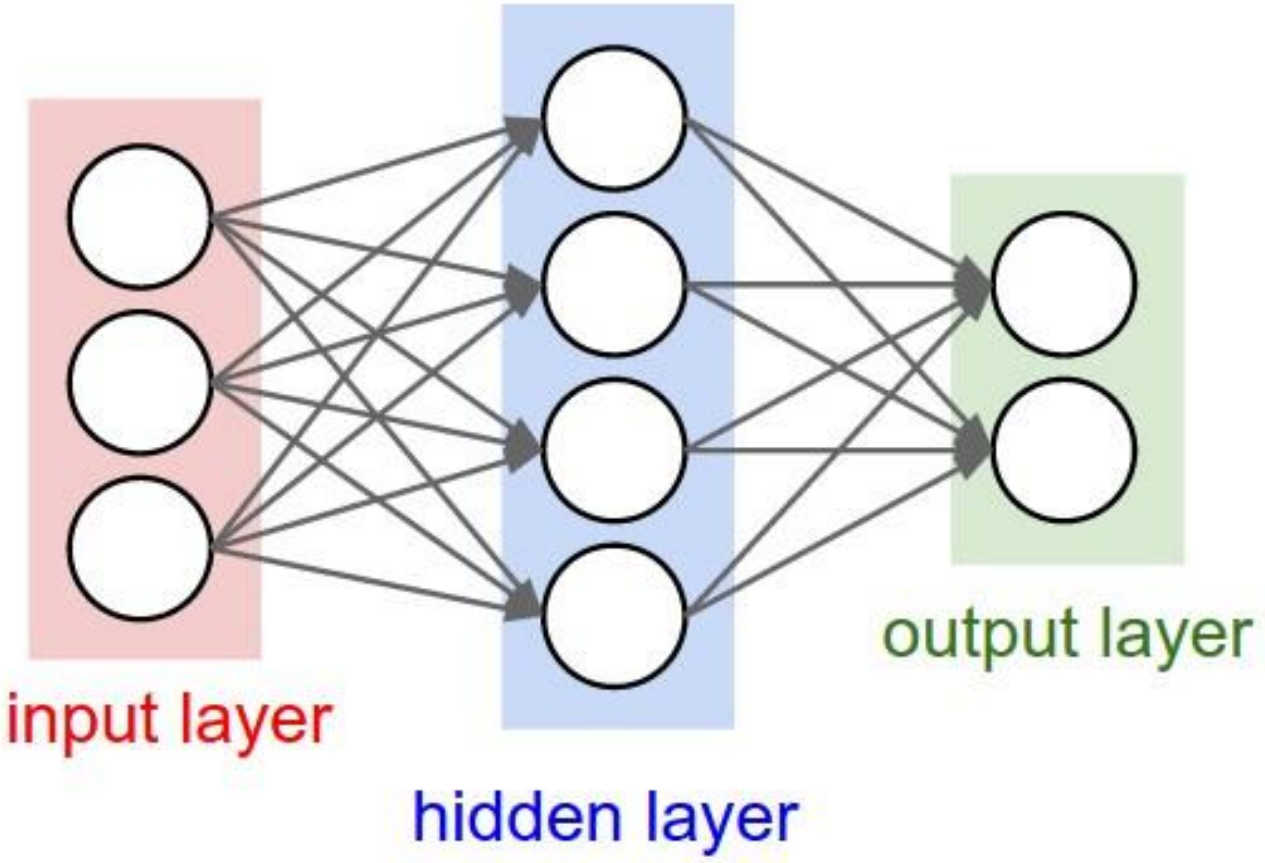
e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

Weight Initialization

Weight Initialization



Q: What happens if we initialize all W=0, b=0?

A: All outputs are 0, all gradients are the same! No “symmetry breaking”

Weight Initialization

Next idea: **small random numbers**
(Gaussian with zero mean, std=0.01)

```
W = 0.01 * np.random.randn(Din, Dout)
```

Works okayish for small networks, but problematic with deeper networks.

Weight Initialization: Activation Statistics

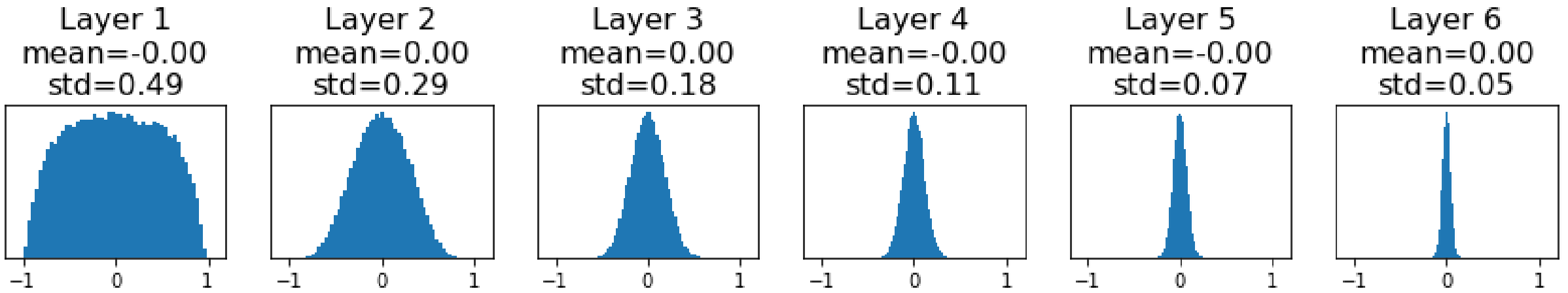
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(



Weight Initialization: Activation Statistics

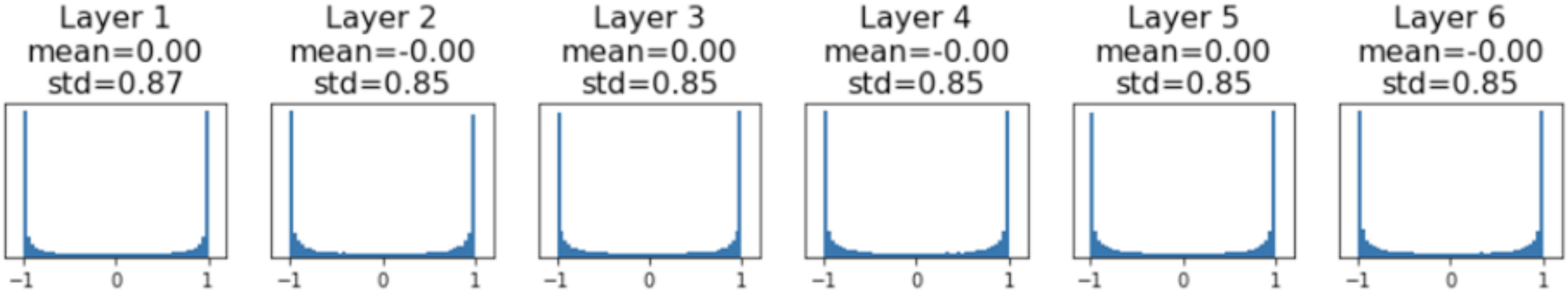
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



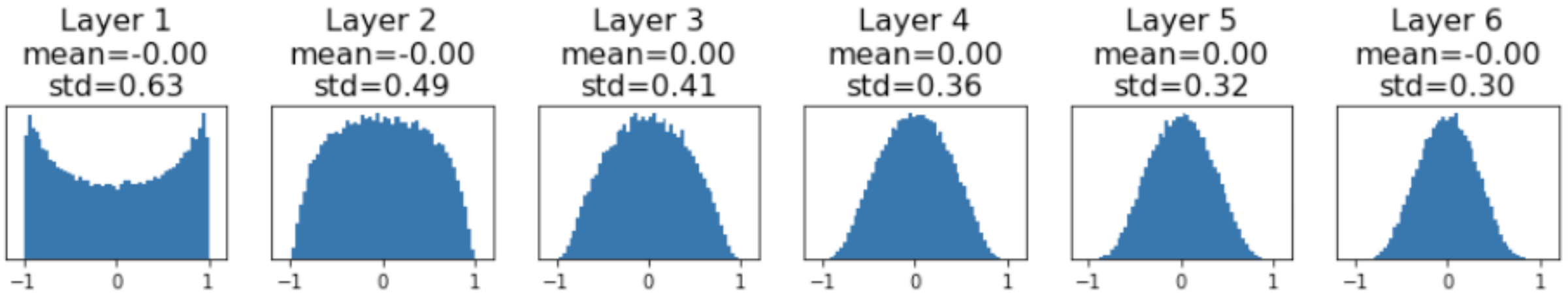
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$$y = Wx \qquad y_i = \sum_{j=1}^{Din} x_j w_j$$

$Var(y_i) = Din * Var(x_i w_i)$ [Assume x, w are iid]

$= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2 E[w_i]^2)$ [Assume x, w independent]

$= Din * Var(x_i) * Var(w_i)$ [Assume x, w are zero-mean]

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

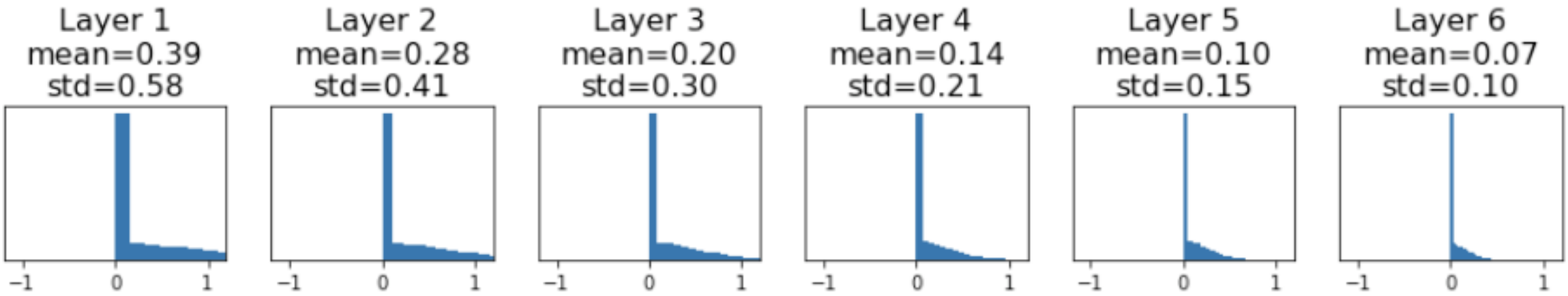
Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

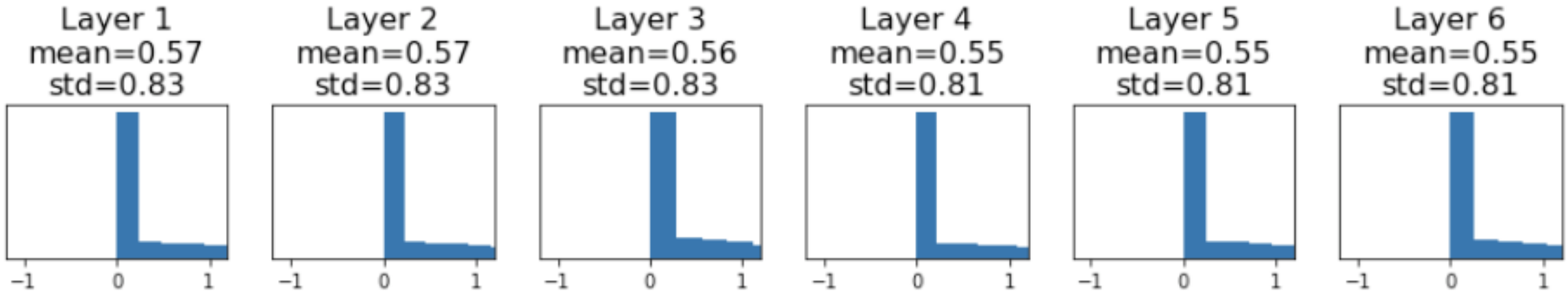


Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din/2)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

”Just right” – activations nicely scaled for all layers



He et al, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, ICCV 2015

Proper initialization is an active area of research

Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

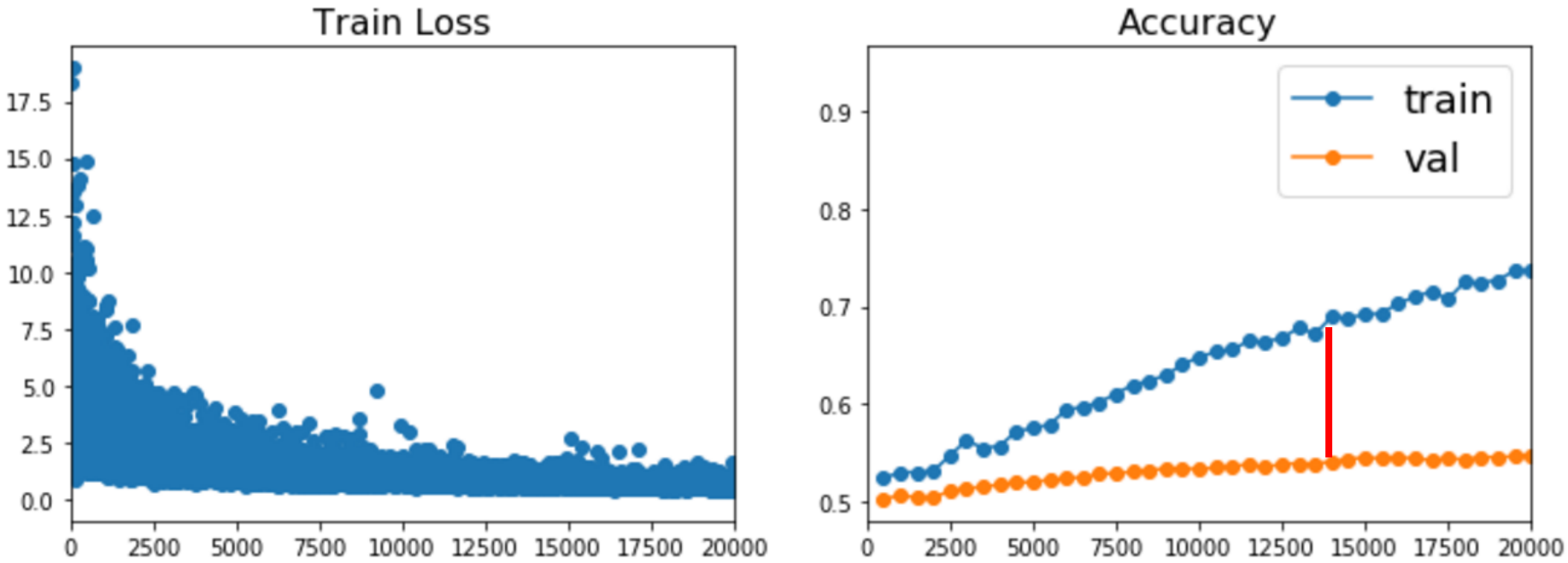
Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

Now your model is training … but it overfits!



Regularization

Regularization

Regularization: Add term to the loss

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

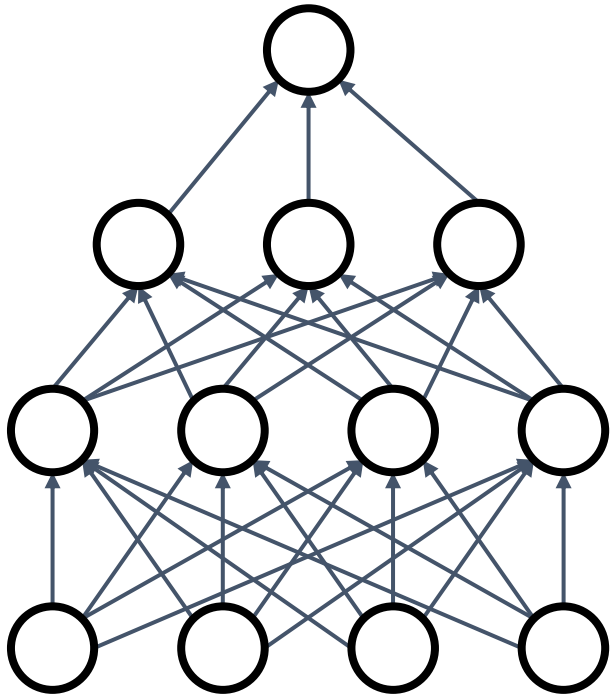
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

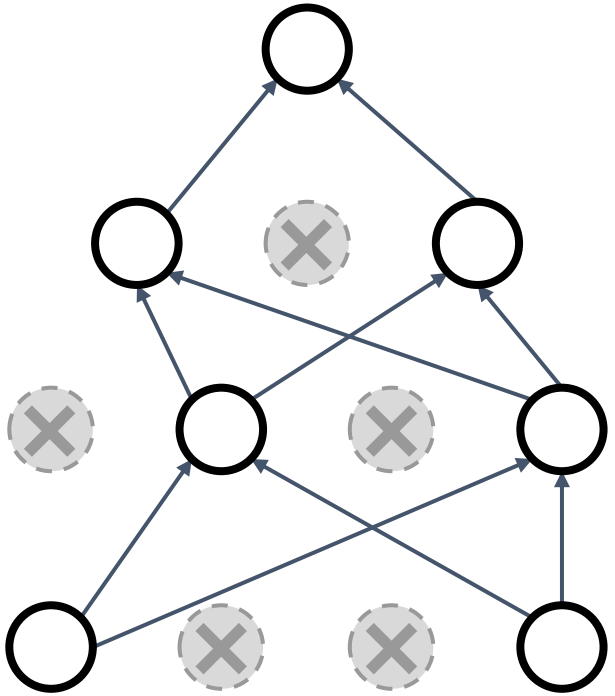
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



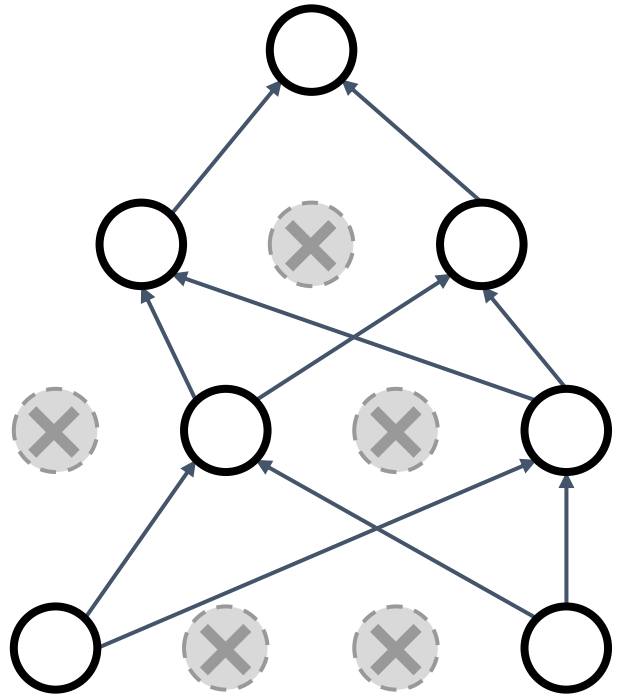


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

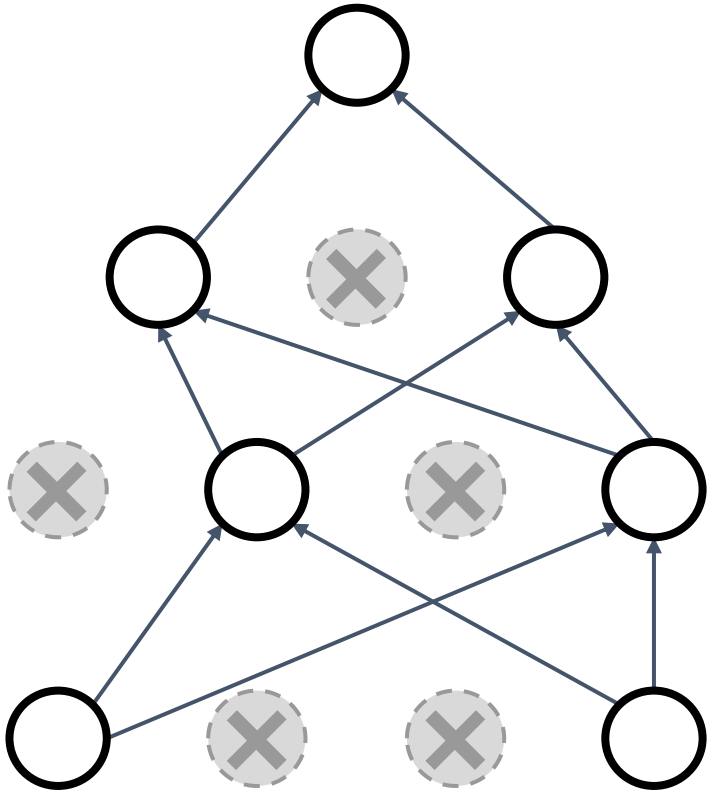
Regularization: Dropout

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout

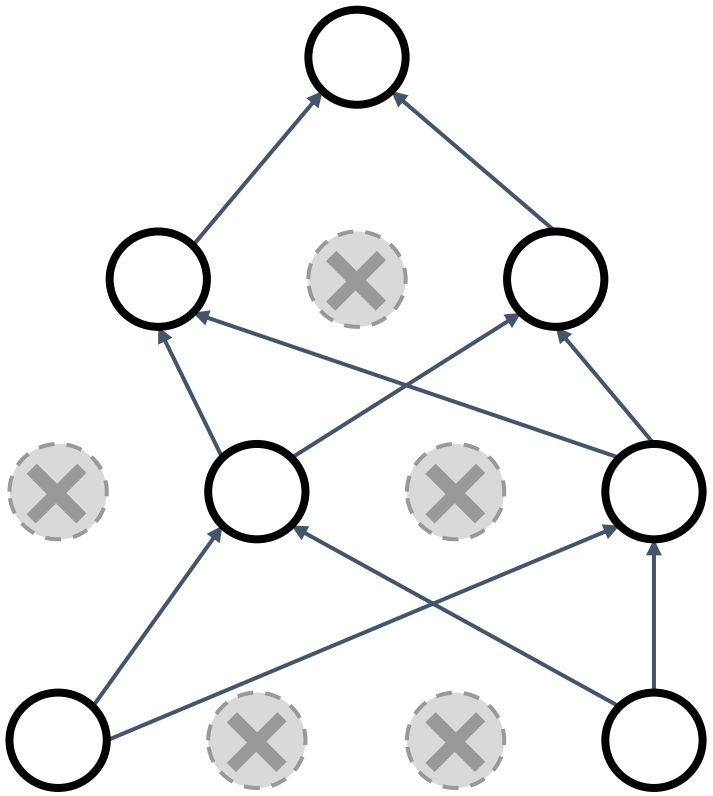


Regularization: Dropout



Forces the network to have a redundant representation; Prevents **co-adaptation** of features

Regularization: Dropout



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary-masked one is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only ~ 10^{82} atoms in the universe...

Dropout: Test Time

Output (label) Input (image)

Dropout makes our output random! $\boldsymbol{y} = f_W(\boldsymbol{x}, \boldsymbol{z})$ Random mask

Want to "average out" the randomness at test-time

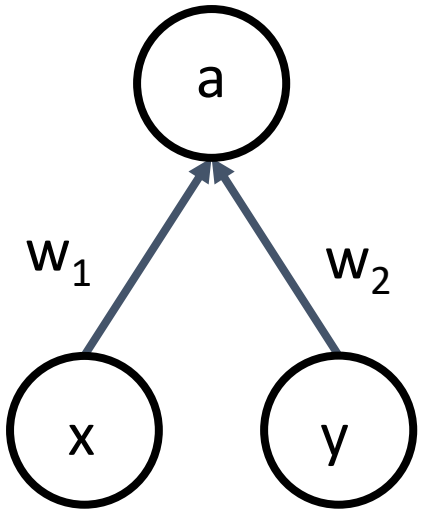
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

But this integral seems hard …

Dropout: Test Time

Want to approximate the integral

$$y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$$



Consider a single neuron:

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

At test time, drop nothing and **multiply** by dropout probability

Dropout: Test Time

```
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

More common: “Inverted dropout”

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Drop and scale during training

test time is unchanged!

Dropout in CNNs

- Dropout prevents co-adaptation of features
  - Good for fully-connected layers

- Dropout is not good for convolutional layers
  - Correlation among features introduced by convolution cannot be removed
  - Empirically, training conv layers with dropout does not perform well

- Dropout is not good with batch normalization
  - Dropout makes the output distribution bimodal
    - Additional modality at 0
  - If you want to try out, apply dropout after batch normalization

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

For ResNet and later, often L2 and Batch Normalization are the only regularizers!

Testing: Average out randomness (sometimes approximate)

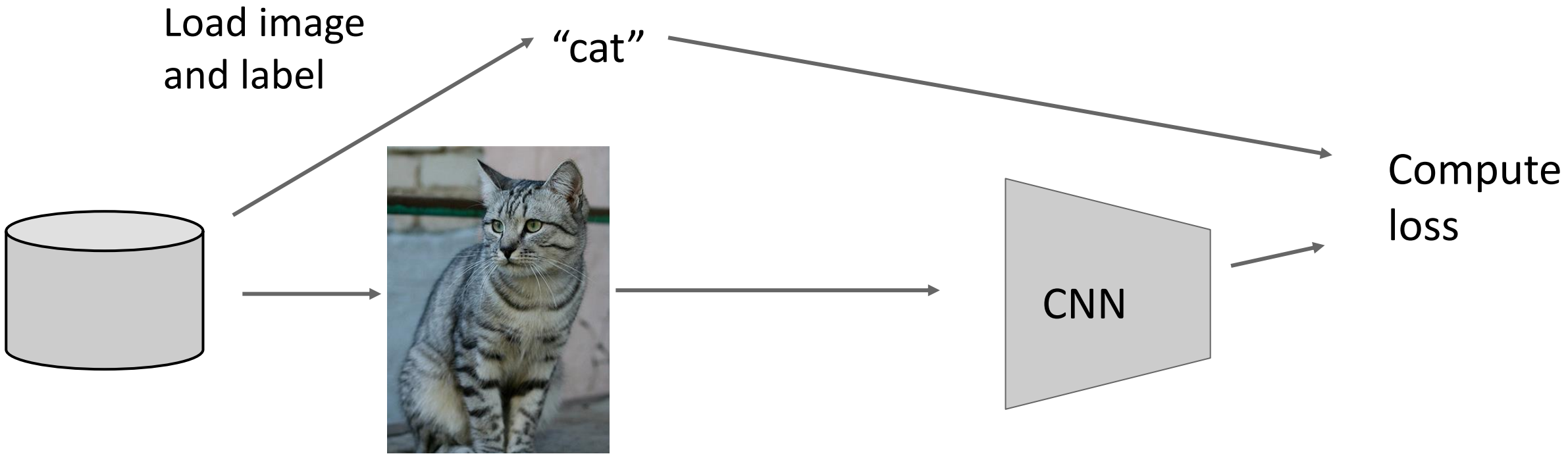
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

Example: Batch Normalization

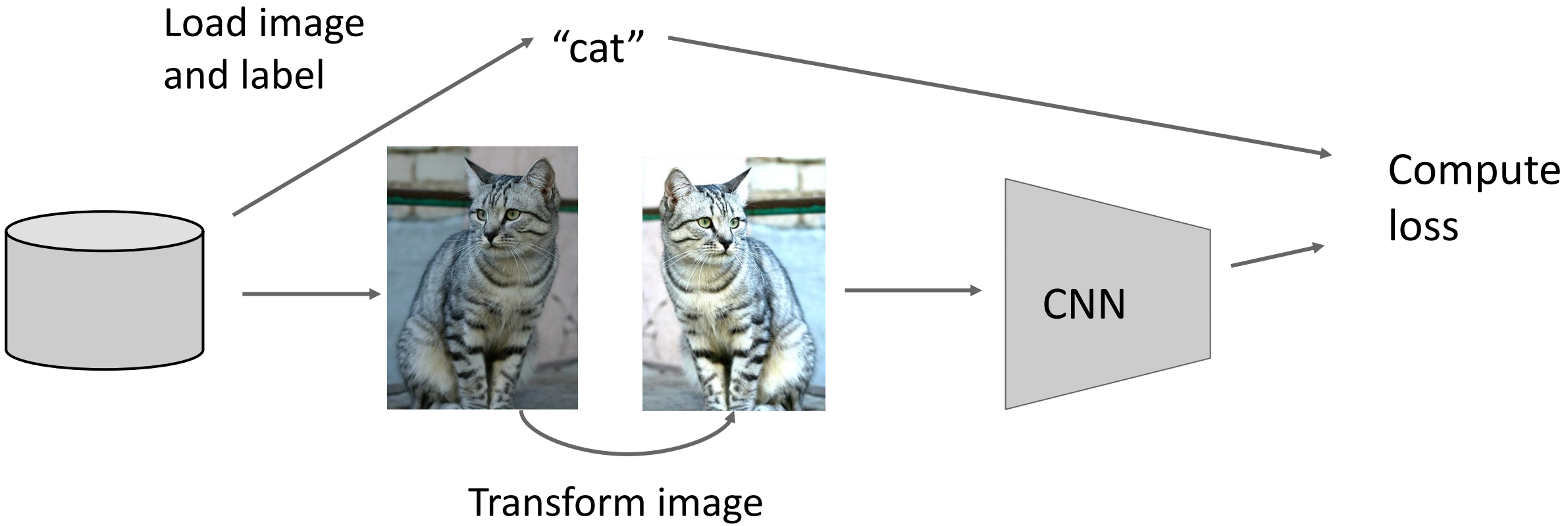
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

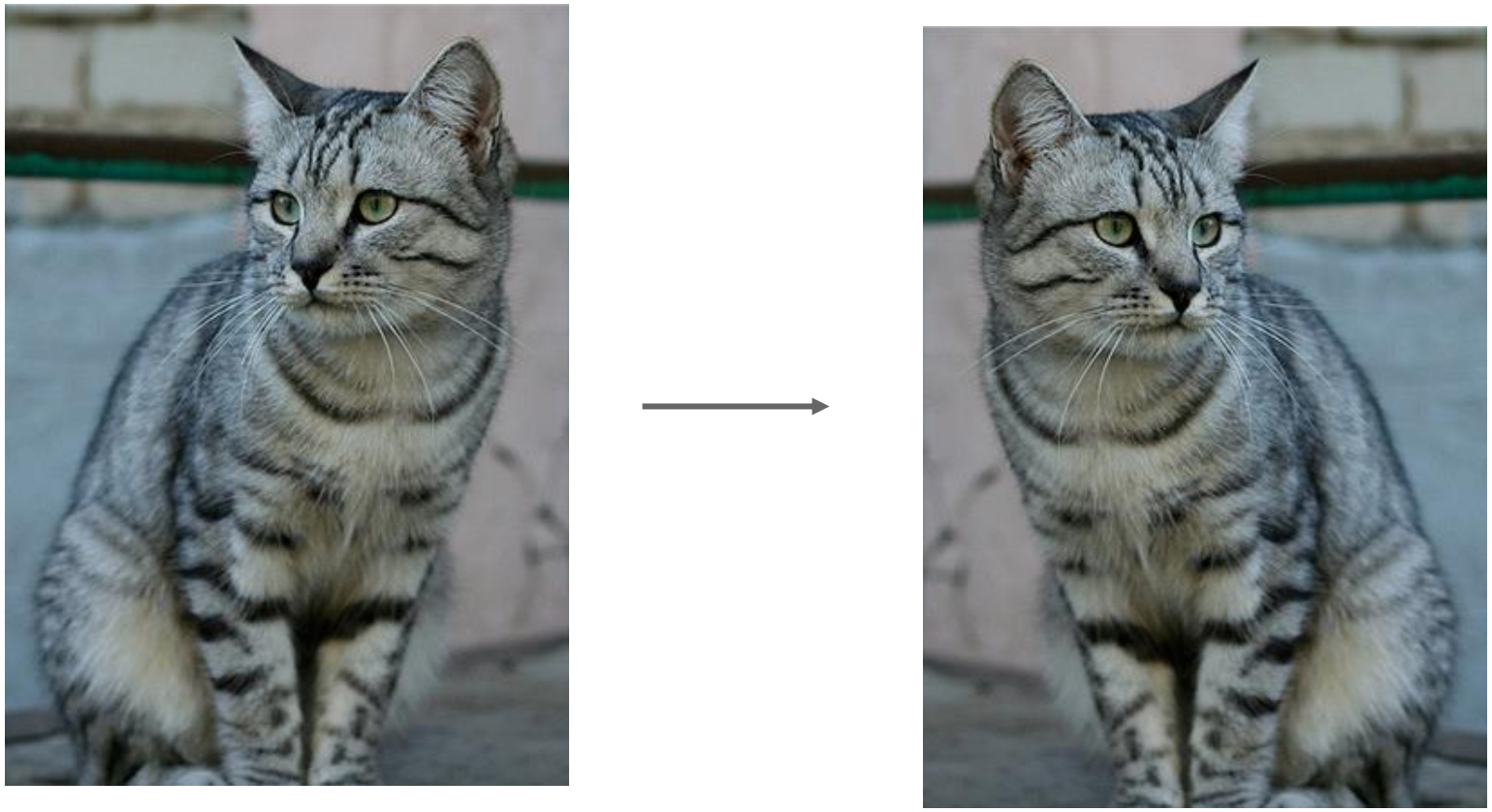
Data Augmentation



Data Augmentation



Data Augmentation: Horizontal Flips

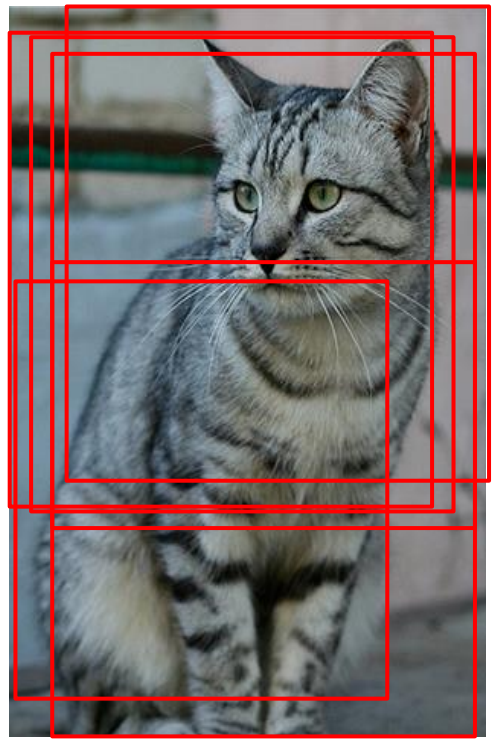


Data Augmentation: Random Crops and Scales

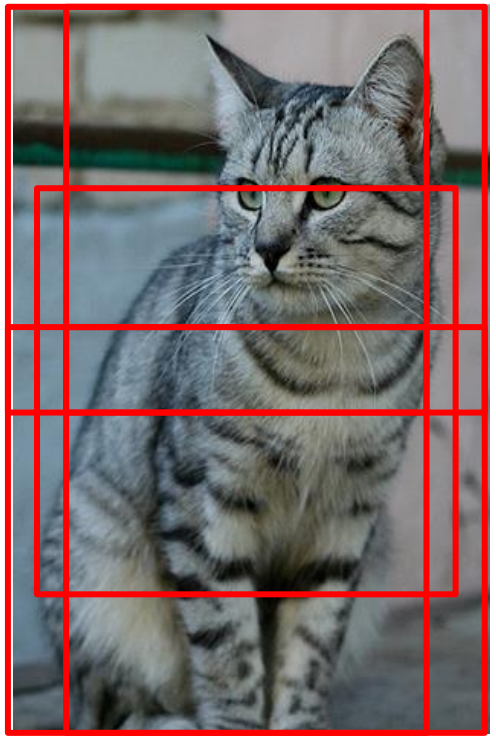
Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data Augmentation: Random Crops and Scales



Training: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

Testing: average (predictions from) a fixed set of crops

ResNet:

1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation: Color Jitter

Simple: Randomize
contrast and brightness

More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

(Used in AlexNet, ResNet, etc)

Data Augmentation: RandAugment

Apply random combinations of transforms:

- **Geometric**: Rotate, translate, shear
- **Color**: Sharpen, contrast, brightness, solarize, posterize, color

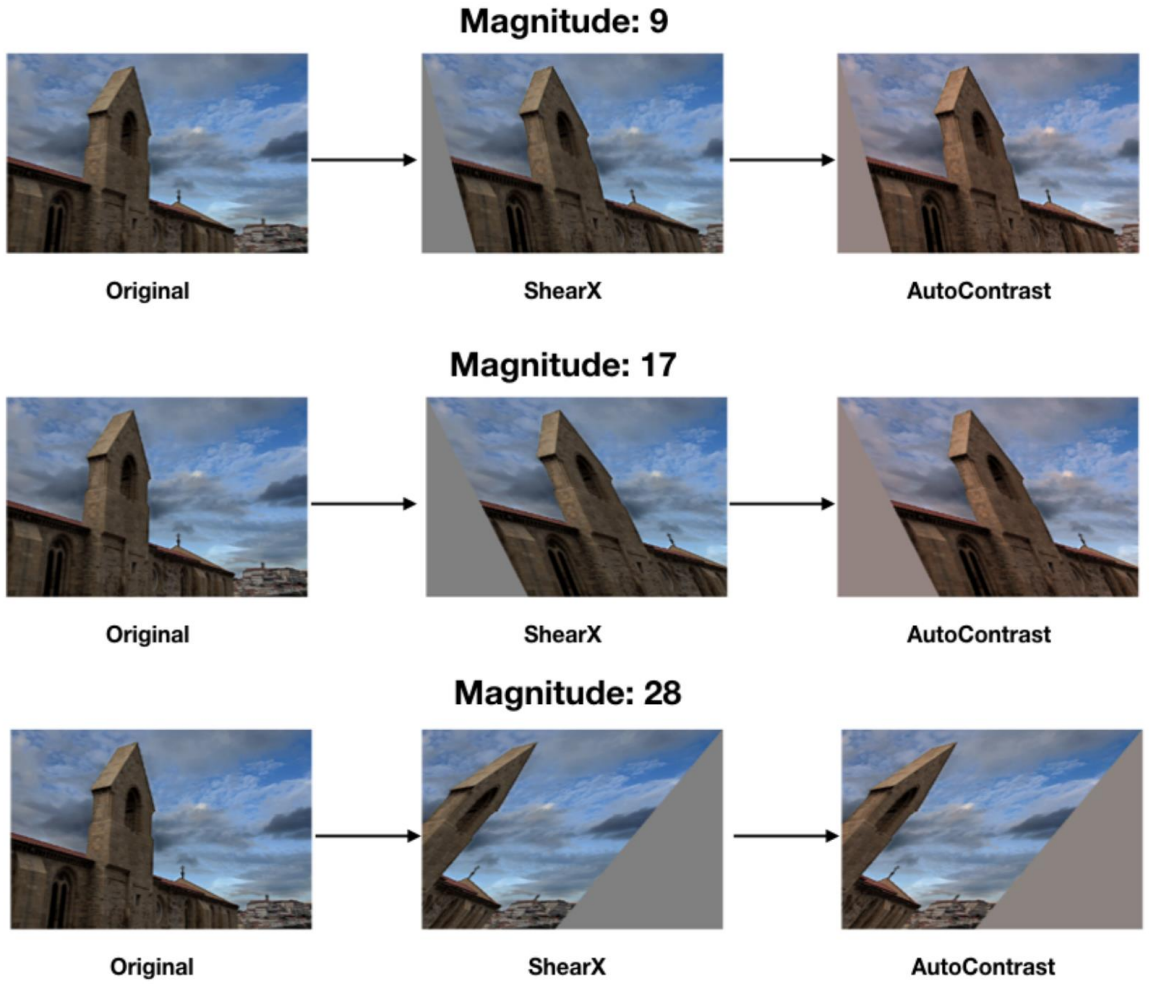
```
transforms = [
'Identity', 'AutoContrast', 'Equalize',
'Rotate', 'Solarize', 'Color', 'Posterize',
'Contrast', 'Brightness', 'Sharpness',
'ShearX', 'ShearY', 'TranslateX', 'TranslateY']

def randaugment(N, M):
"""Generate a set of distortions.

   Args:
     N: Number of augmentation transformations to
          apply sequentially.
     M: Magnitude for all the transformations.
"""

  sampled_ops = np.random.choice(transforms, N)
  return [(op, M) for op in sampled_ops]
```

Cubuk et al, “RandAugment: Practical augmented data augmentation with a reduced search space”, NeurIPS 2020

Data Augmentation: RandAugment

Apply random combinations of transforms:

- **Geometric**: Rotate, translate, shear
- **Color**: Sharpen, contrast, brightness, solarize, posterize, color



Cubuk et al, "RandAugment: Practical augmented data augmentation with a reduced search space", NeurIPS 2020

Data Augmentation: Get creative for your problem!

Data augmentation encodes **invariances** in your model

Think for your problem: what changes to the image should **not** change the network output?

May be different for different tasks!

Regularization: A common pattern

Training: Add some randomness

Testing: Marginalize over randomness

Examples:

Dropout

Batch Normalization

Data Augmentation

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Regularization: DropConnect

Training: Drop random connections between neurons (set weight=0)

Testing: Use all the connections

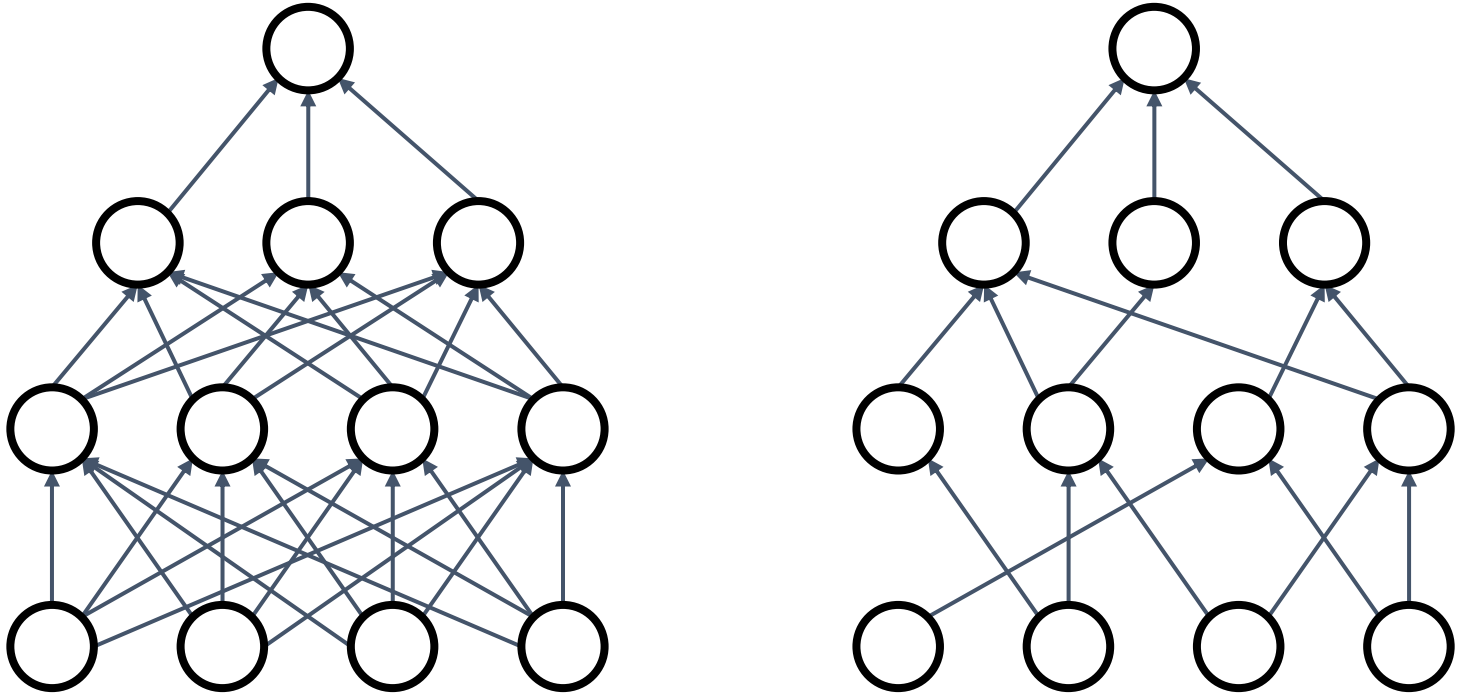
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013

Regularization: Stochastic Depth

Training: Skip some residual blocks in ResNet

Testing: Use the whole network

Examples:

Dropout

Batch Normalization

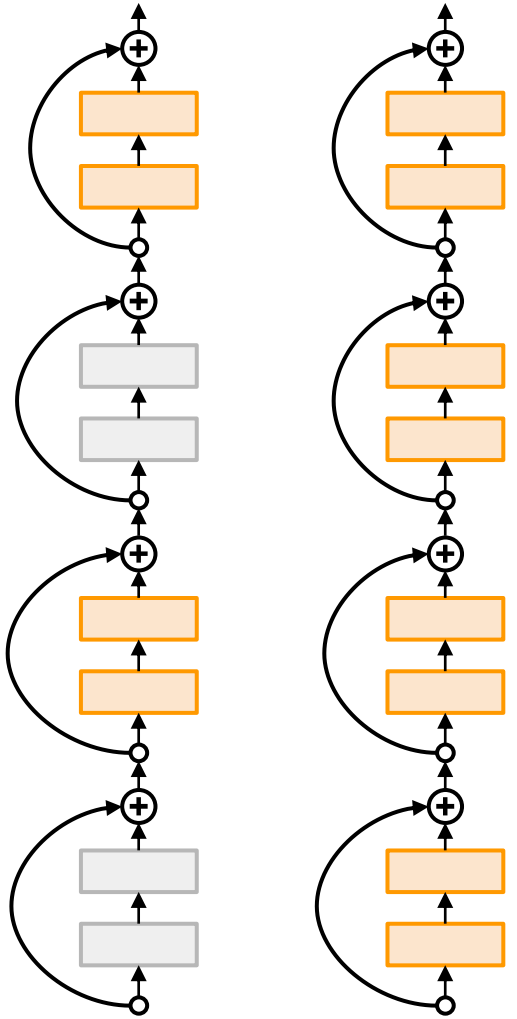
Data Augmentation

DropConnect

Stochastic Depth

Starting to become common in recent architectures!

- Pham et al, "Very Deep Self-Attention Networks for End-to-End Speech Recognition", INTERSPEECH 2019
- Tan and Le, "EfficientNetV2: Smaller Models and Faster Training", ICML 2021
- Fan et al, "Multiscale Vision Transformers", ICCV 2021
- Bello et al, "Revisiting ResNets: Improved Training and Scaling Strategies", NeurIPS 2021
- Steiner et al, "How to train your ViT? Data, Augmentation, and Regularization in Vision Transformers", arXiv 2021

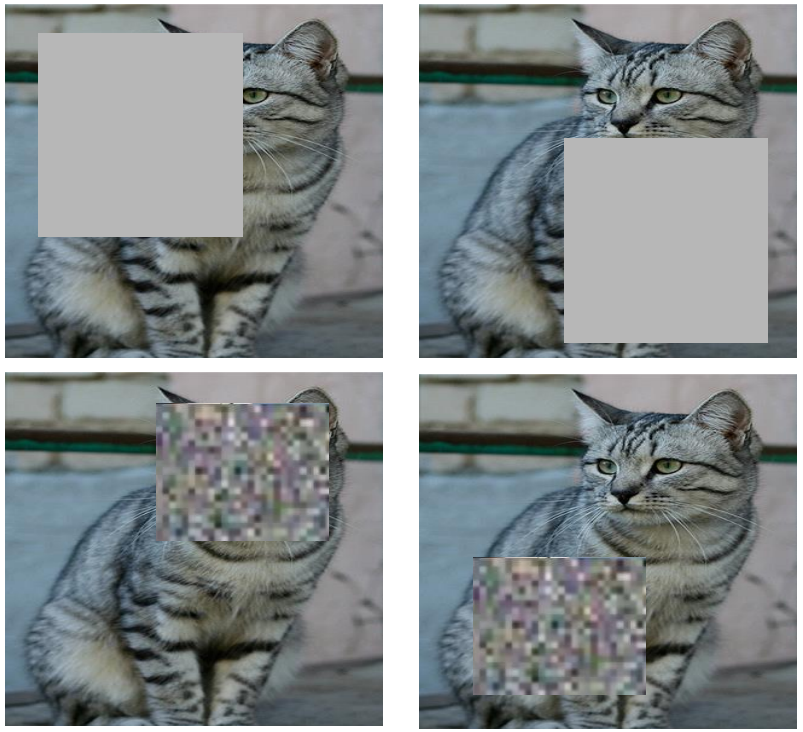


Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

Regularization: CutOut

Training: Set random images regions to 0
Testing: Use the whole image

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Stochastic Depth
Cutout / Random Erasing



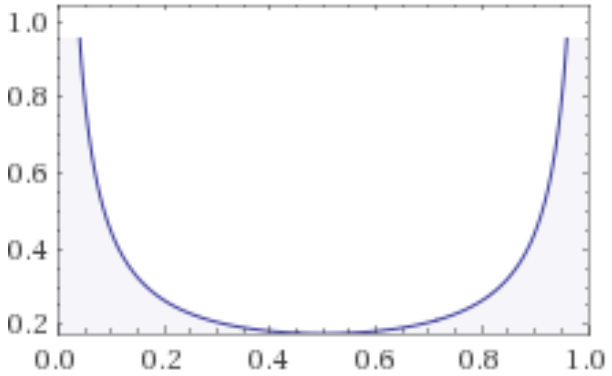
Replace random regions with mean value or random values

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017
Zhong et al, "Random Erasing Data Augmentation", AAAI 2020

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images



Sample blend probability from a beta distribution Beta(a, b) with a=b≈0 so blend weights are close to 0/1

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Stochastic Depth
Cutout / Random Erasing
Mixup

CNN

Target label:
cat: 0.4
dog: 0.6

Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Regularization: CutMix

Yun et al, "CutMix: Regularization Strategies to Train Strong Classifiers with Localizable Features", ICCV 2019

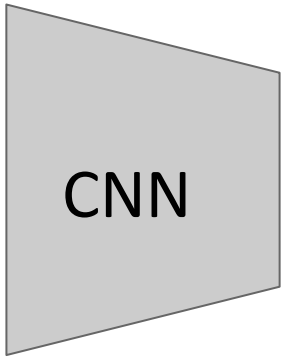
Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix

Replace random crops of one image with another: e.g. 60% of pixels from cat, 40% from dog



Target label:
cat: 0.6
dog: 0.4

Regularization: Label Smoothing

Szegedy et al, "Rethinking the Inception Architecture for Computer Vision", CVPR 2015

Training: Change target distribution

Testing: Take argmax over predictions

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix
Label Smoothing

Target Distribution

Standard Training	Label Smoothing
Cat: 100%	Cat: 90%
Dog: 0%	Dog: 5%
Fish: 0%	Fish: 5%

Set target distribution to be $1 - \frac{K-1}{K}\epsilon$ on the correct category and ϵ/K on all other categories, with K categories and $\epsilon \in (0,1)$.
Loss is cross-entropy between predicted and target distribution.

Regularization: Summary

Training: Add randomness

Testing: Marginalize over randomness

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix
Label Smoothing

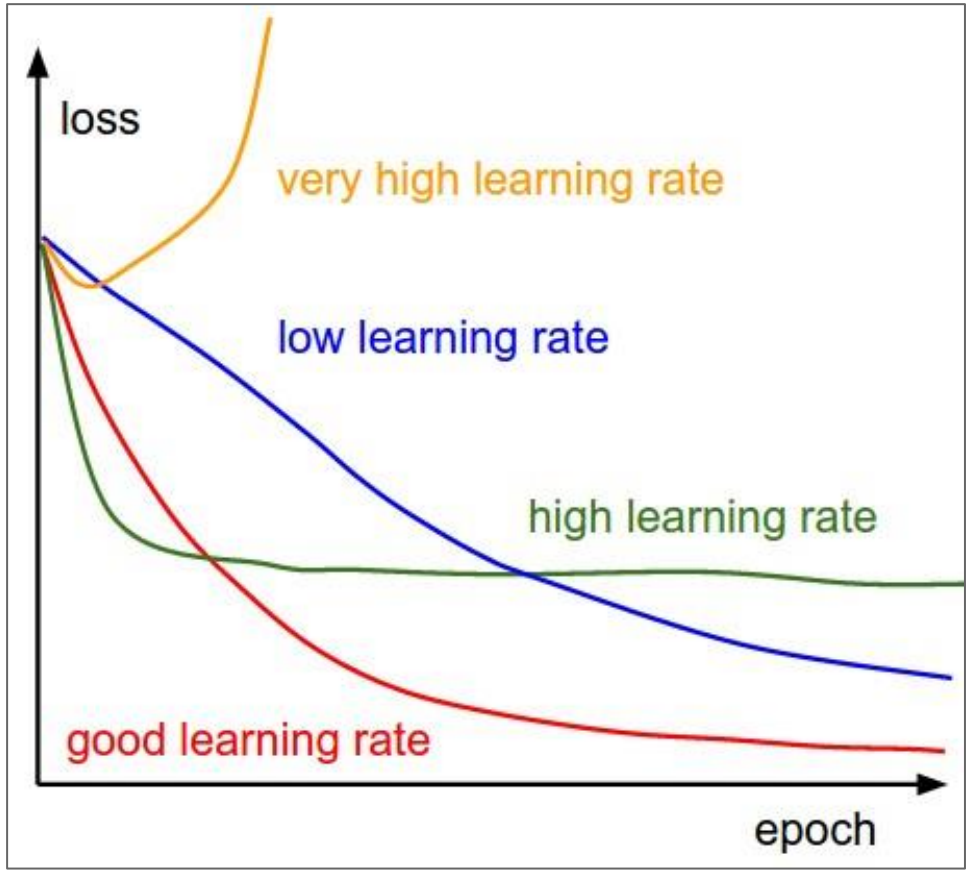
- Use Dropout for large fully-connected layers
- Data augmentation always a good idea
- Use BatchNorm for CNNs (but not ViTs)
- Try Cutout, MixUp, CutMix, Stochastic Depth, Label Smoothing to squeeze out a bit of extra performance

Overview

1. **One time setup**
   Activation functions, data preprocessing, weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules, large-batch training, hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

Learning Rate Schedules

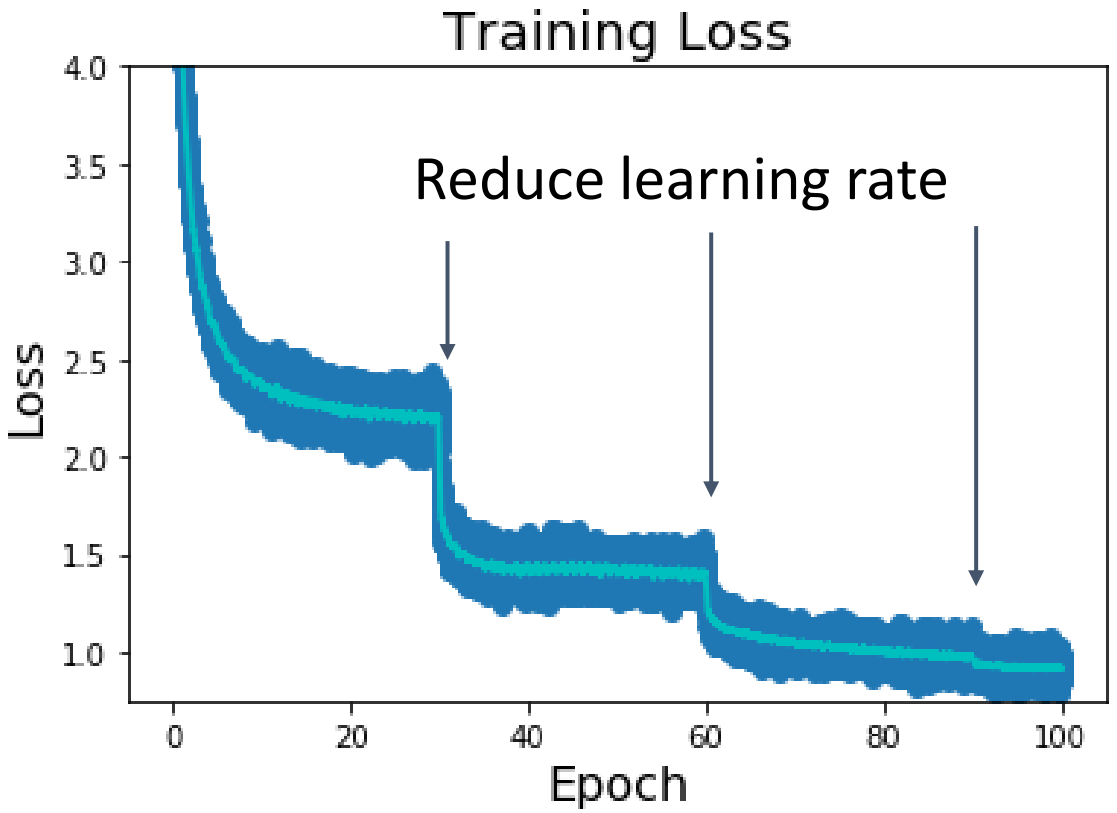
SGD, SGD+Momentum, Adagrad, RMSProp, Adam
all have **learning rate** as a hyperparameter.



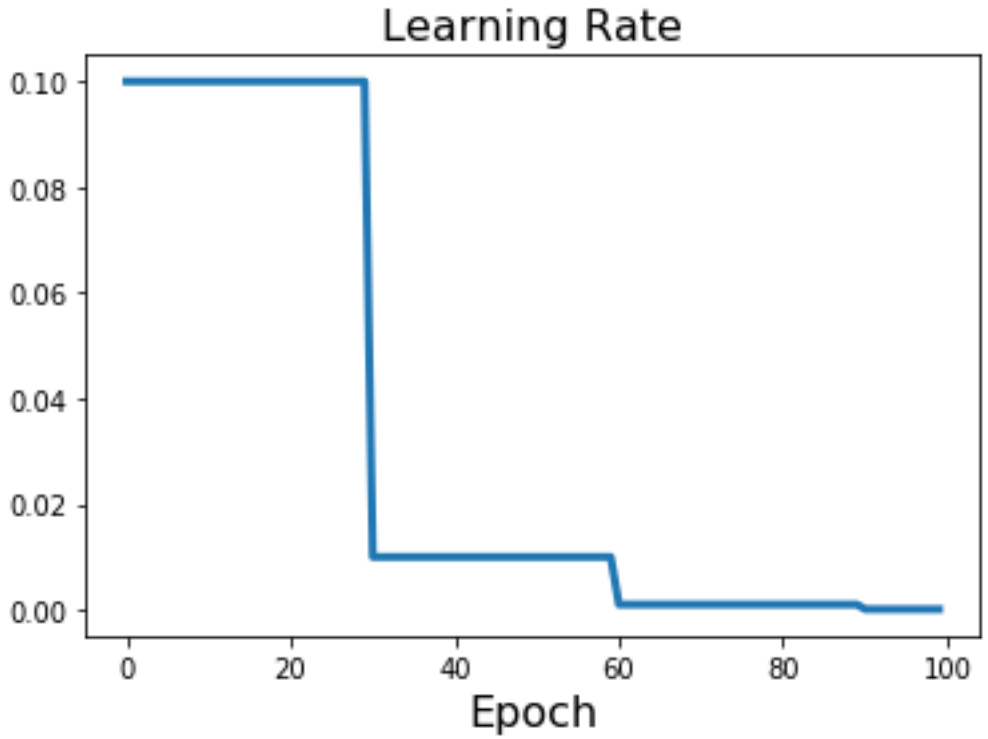
Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time

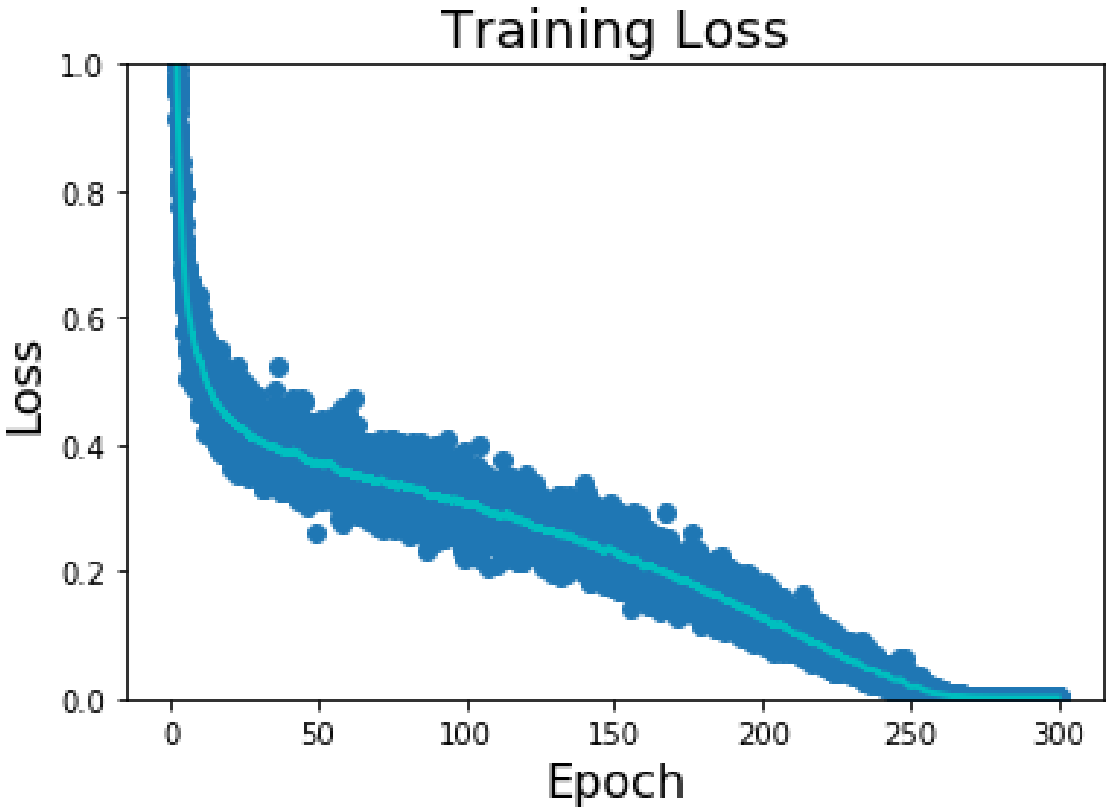
Learning Rate Decay: Step



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

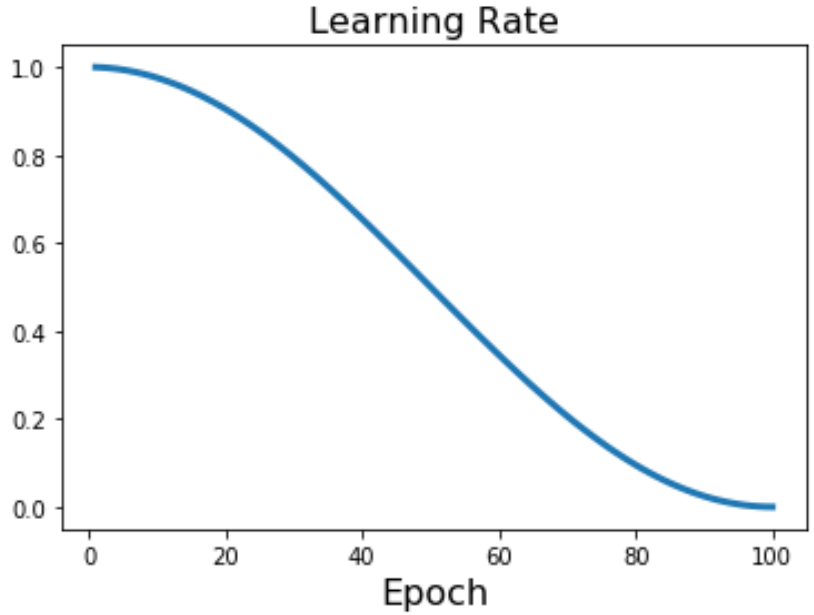


Learning Rate Decay: Cosine



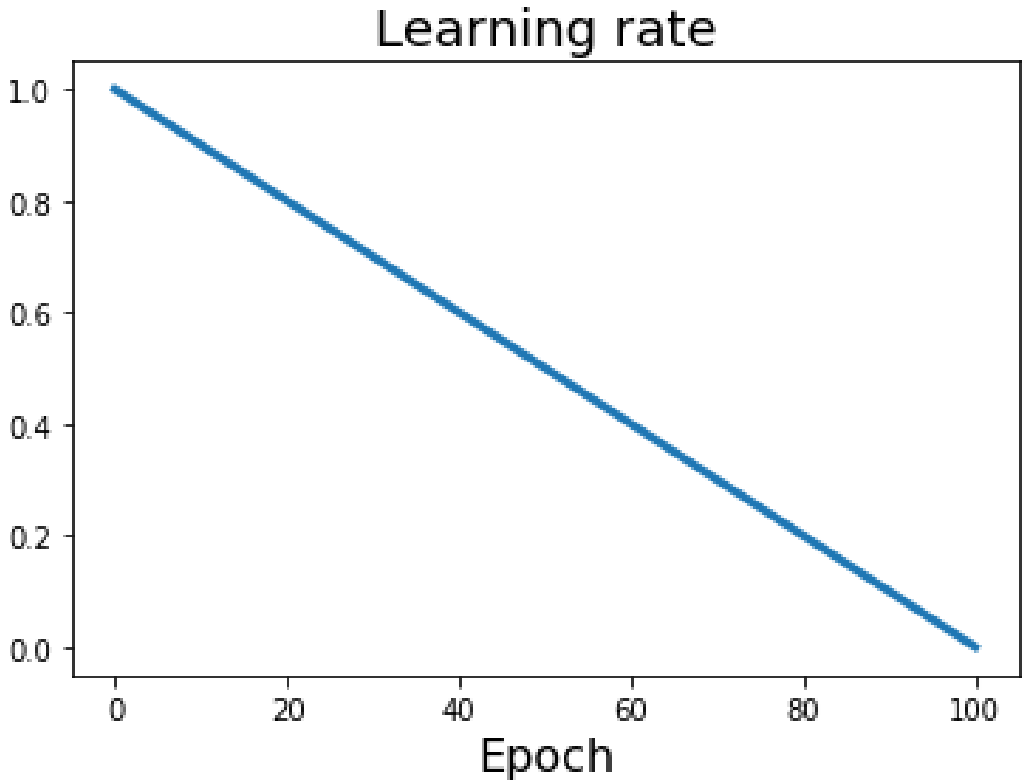
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $$\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$$



Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", ICCV 2019
Radosavovic et al, "On Network Design Spaces for Visual Recognition", ICCV 2019
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Learning Rate Decay: Linear



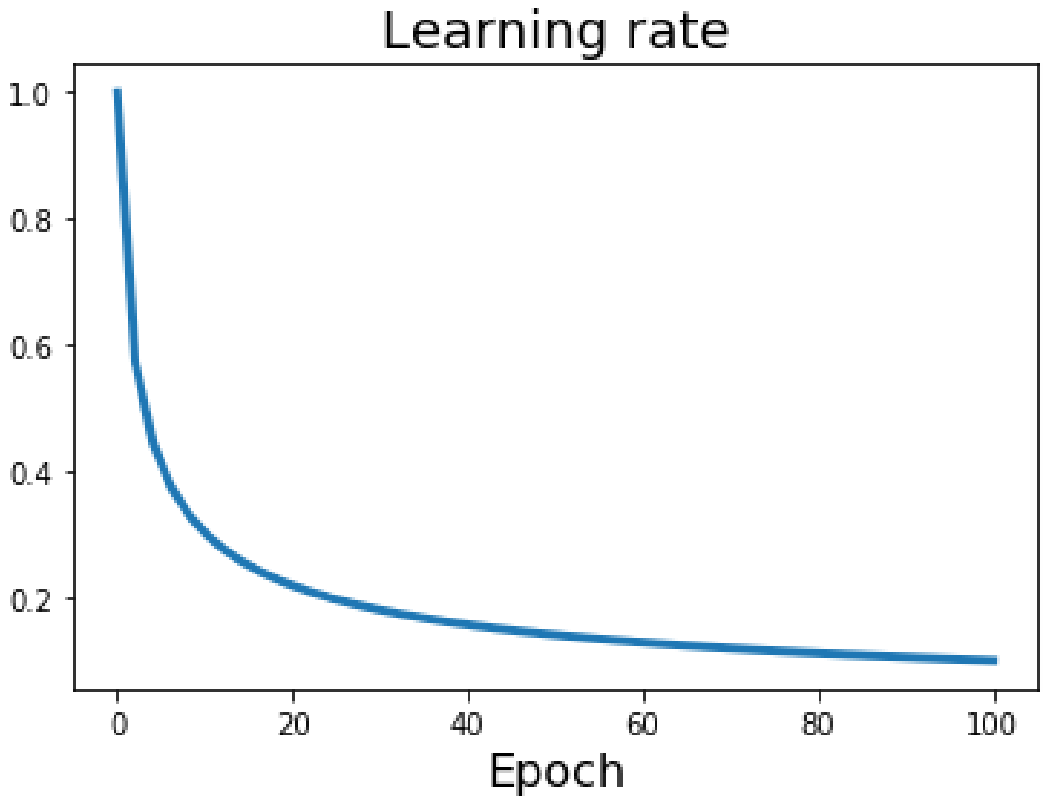
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

Linear: $\alpha_t = \alpha_0\left(1 - \frac{t}{T}\right)$

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", NAACL 2018
Liu et al, "RoBERTa: A Robustly Optimized BERT Pretraining Approach", 2019
Yang et al, "XLNet: Generalized Autoregressive Pretraining for Language Understanding", NeurIPS 2019

Learning Rate Decay: Inverse Sqrt



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

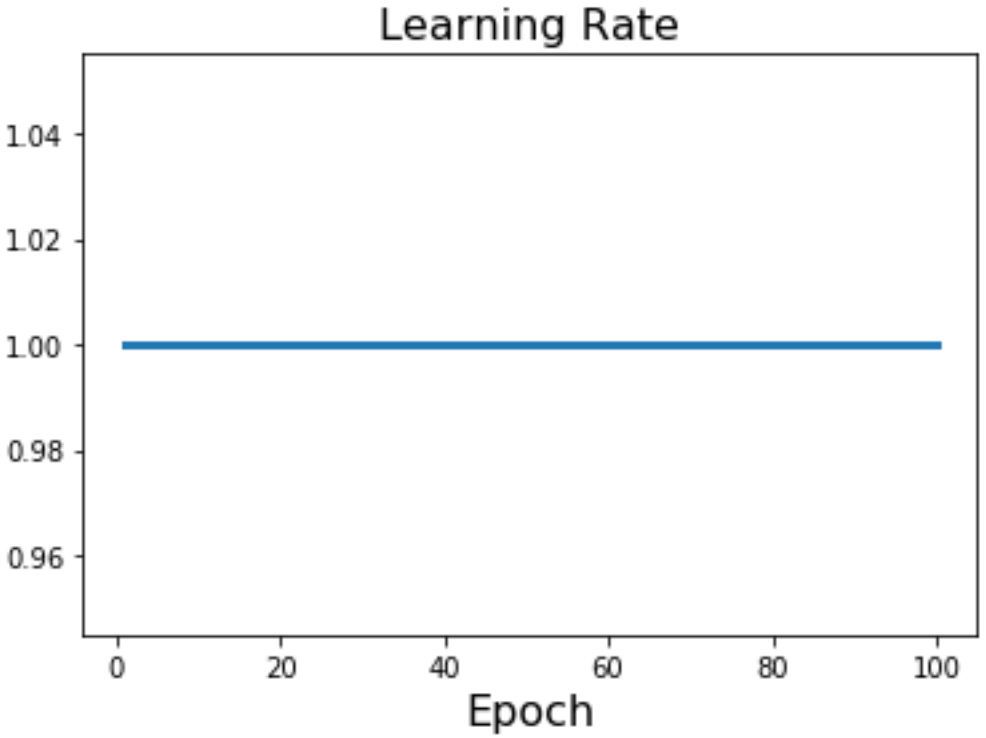
Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

Linear: $\alpha_t = \alpha_0\left(1 - \frac{t}{T}\right)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Vaswani et al, "Attention is all you need", NIPS 2017

Learning Rate Decay: Constant



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

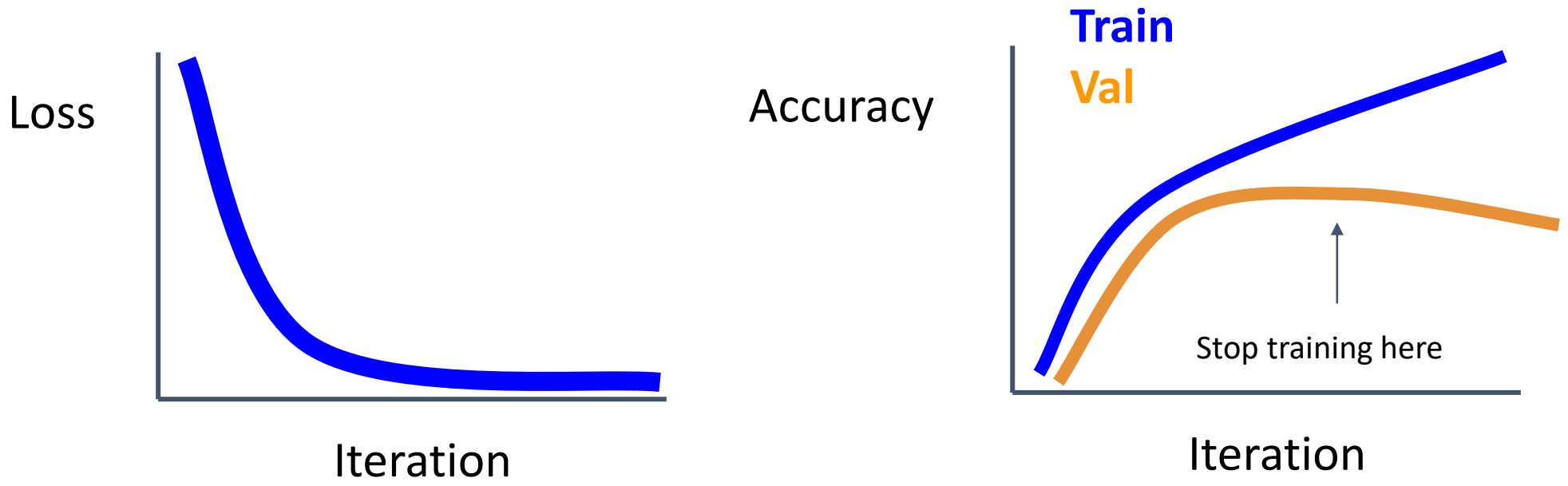
Linear: $\alpha_t = \alpha_0\left(1 - \frac{t}{T}\right)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

Constant: $\alpha_t = \alpha_0$

Brock et al, "Large Scale GAN Training for High Fidelity Natural Image Synthesis", ICLR 2019
Donahue and Simonyan, "Large Scale Adversarial Representation Learning", NeurIPS 2019

How long to train? **Early Stopping**



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val. **Always a good idea to do this!**

Choosing Hyperparameters

Choosing Hyperparameters: Grid Search

Choose several values for each hyperparameter
(Often space choices log-linearly)

Example:
Weight decay: [$1x10^{-4}$, $1x10^{-3}$, $1x10^{-2}$, $1x10^{-1}$]
Learning rate: [$1x10^{-4}$, $1x10^{-3}$, $1x10^{-2}$, $1x10^{-1}$]

Evaluate all possible choices on this
hyperparameter grid

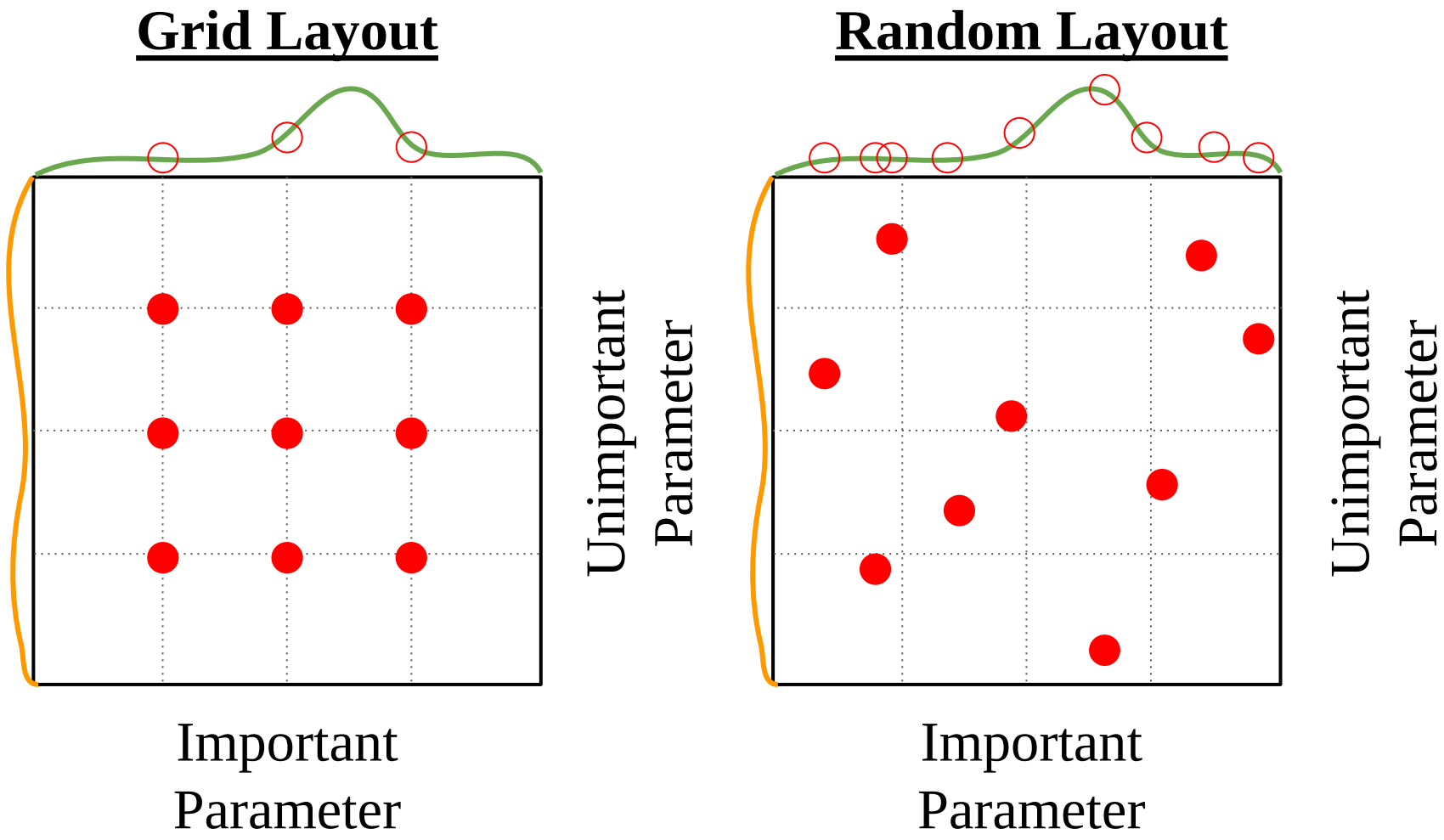
Choosing Hyperparameters: Random Search

Choose several values for each hyperparameter
(Often space choices log-linearly)

Example:
Weight decay: log-uniform on [$1x10^{-4}$, $1x10^{-1}$]
Learning rate: log-uniform on [$1x10^{-4}$, $1x10^{-1}$]

Run many different trials

Hyperparameters: Random vs Grid Search



Bergstra and Bengio, "Random Search for Hyper-Parameter Optimization", JMLR 2012

Choosing Hyperparameters

(without tons of GPUs)

Choosing Hyperparameters

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. log(C) for softmax with C classes

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization. Turn off regularization.

Loss not going down? LR too low, bad initialization
Loss explodes to Inf or NaN? LR too high, bad initialization

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Choosing Hyperparameters

Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

Choosing Hyperparameters

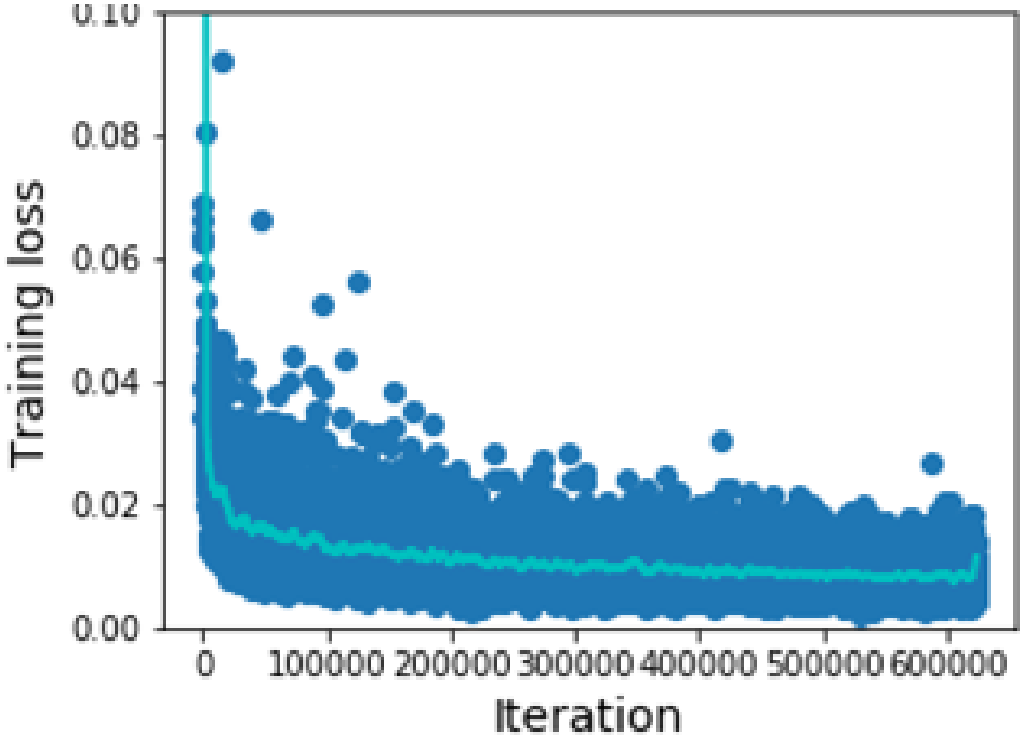
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer

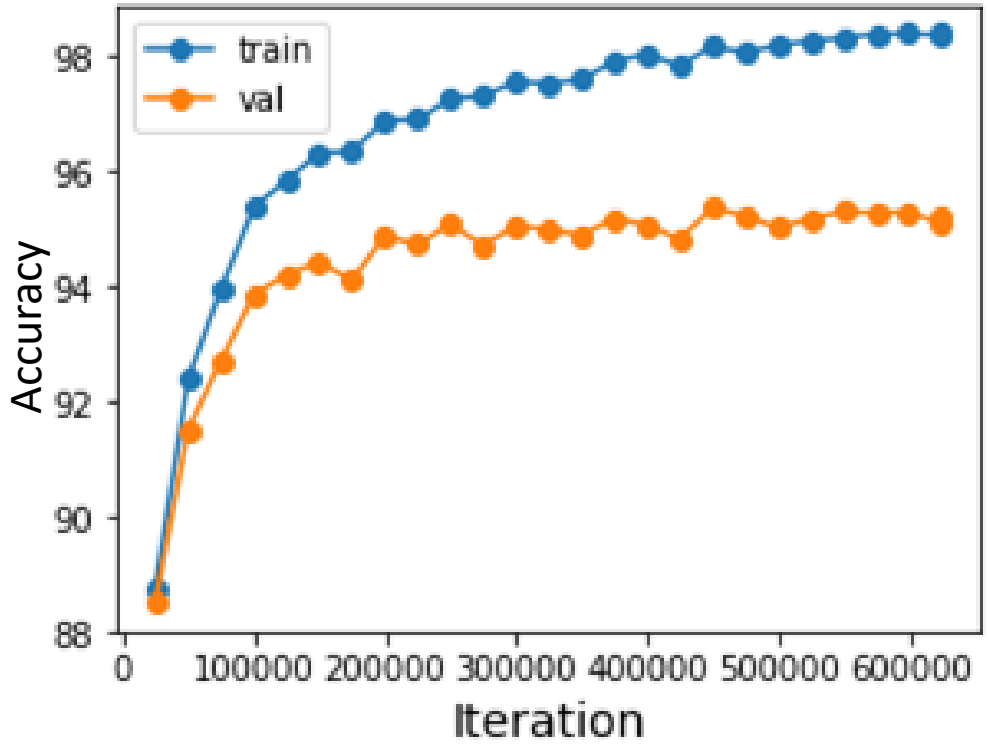
Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

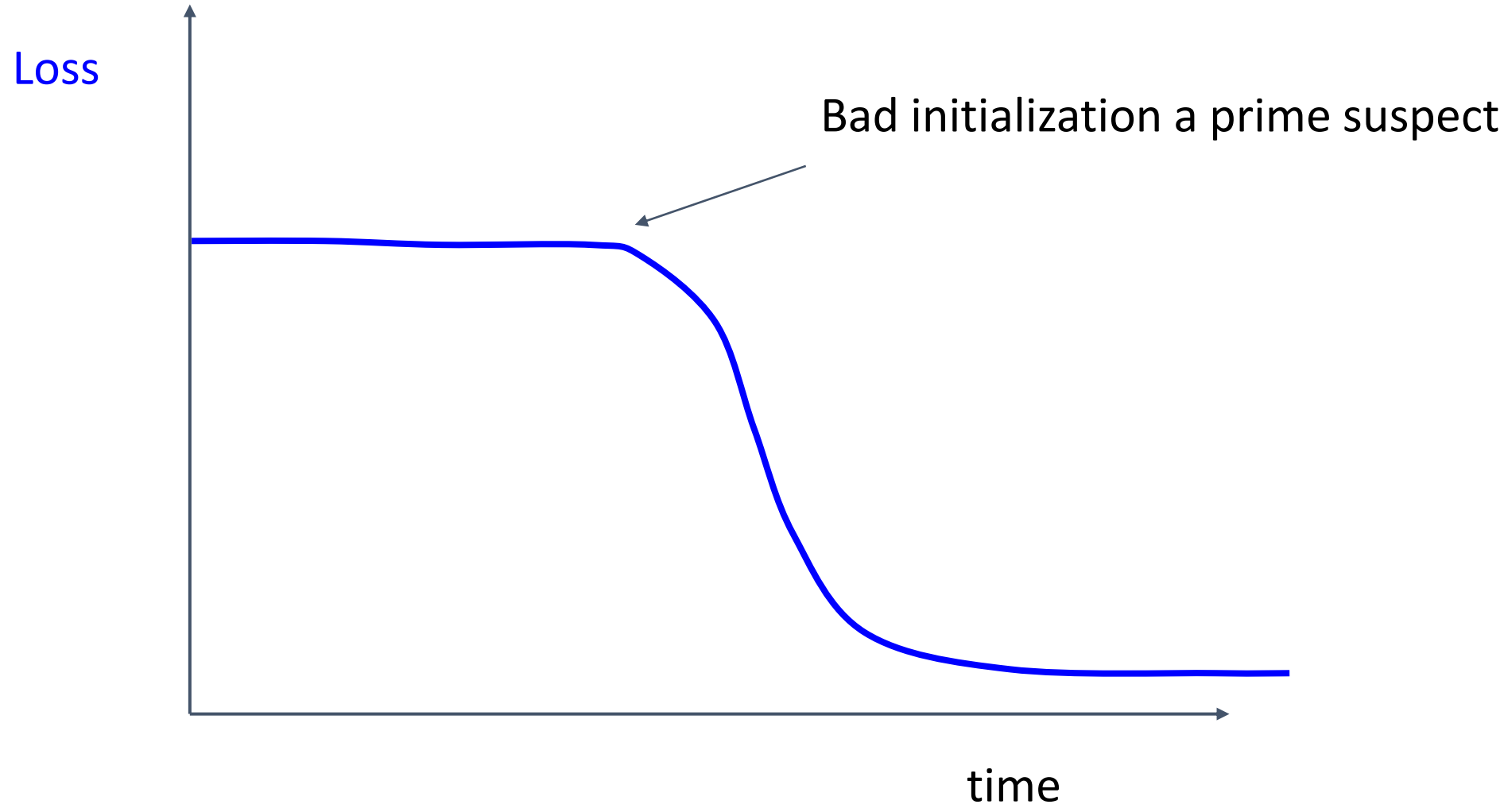
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at learning curves

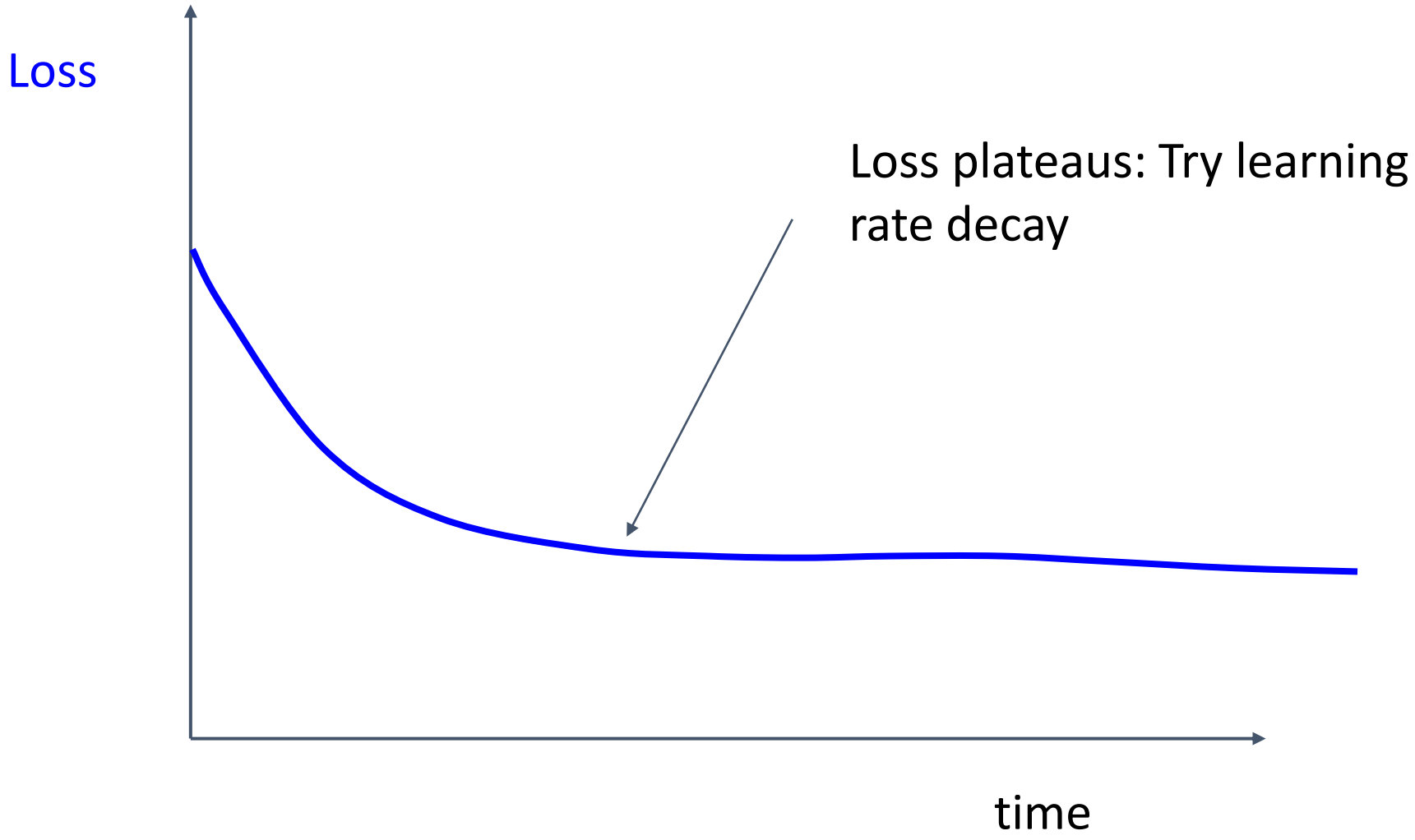
Look at Learning Curves!

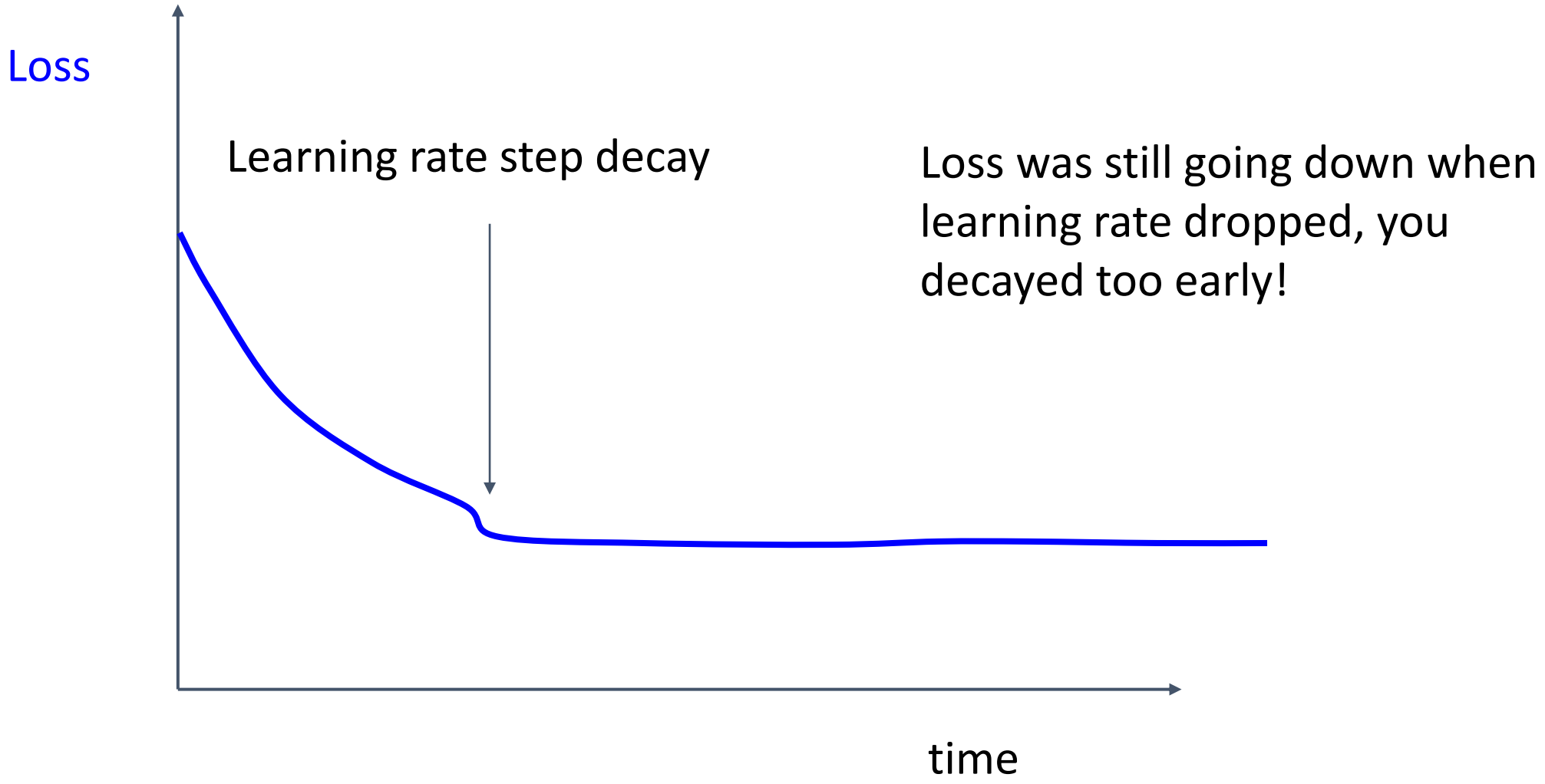


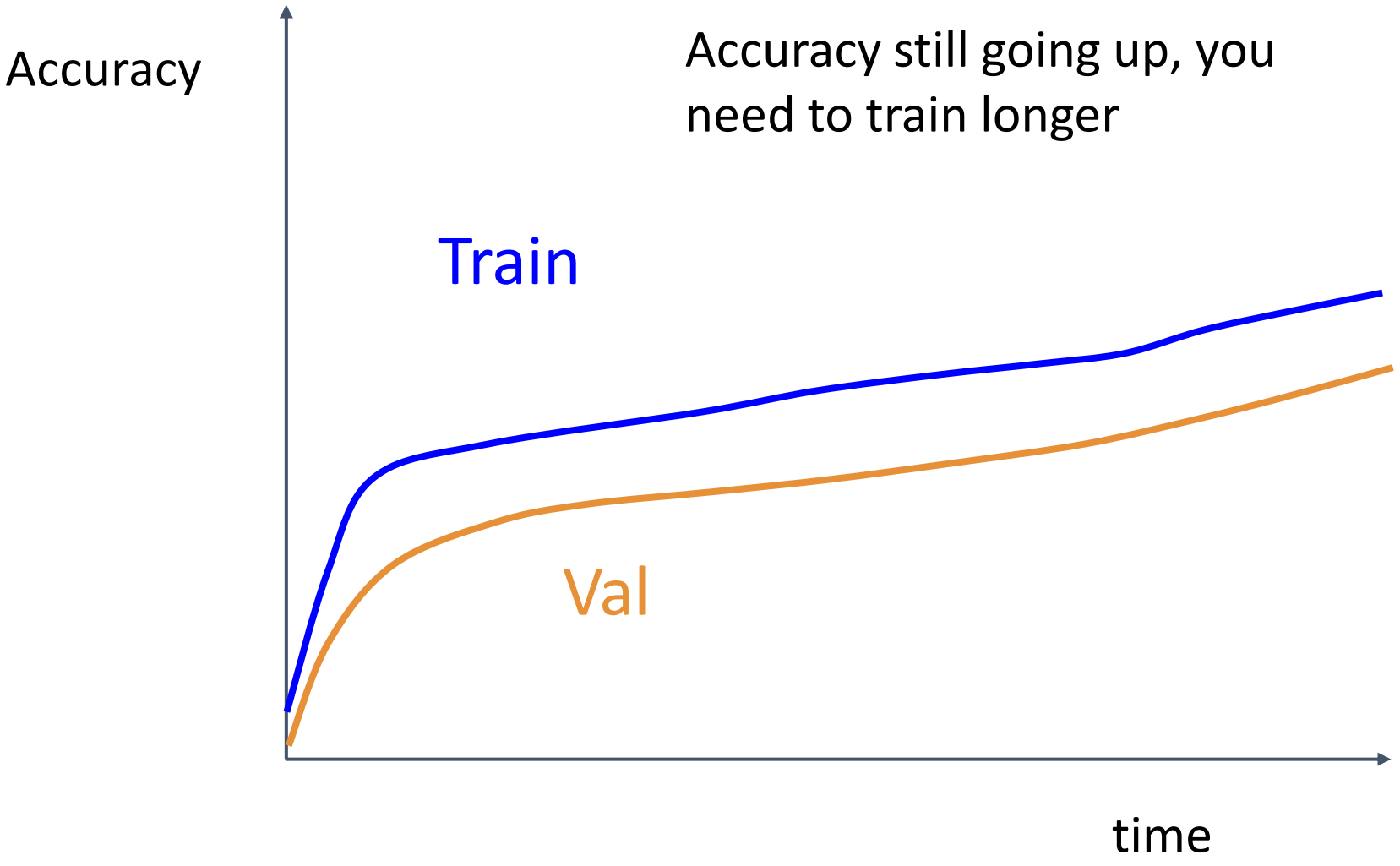


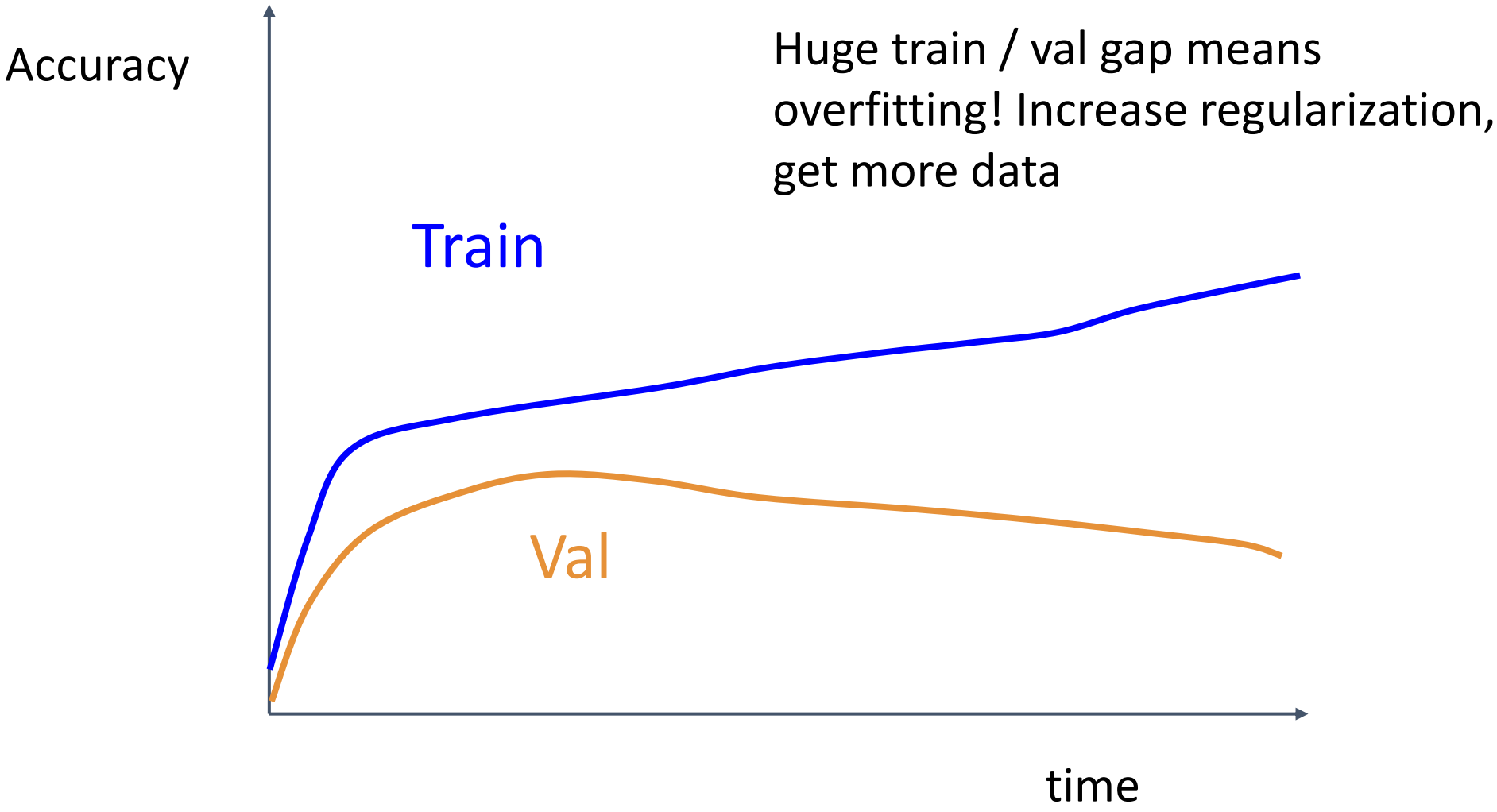
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

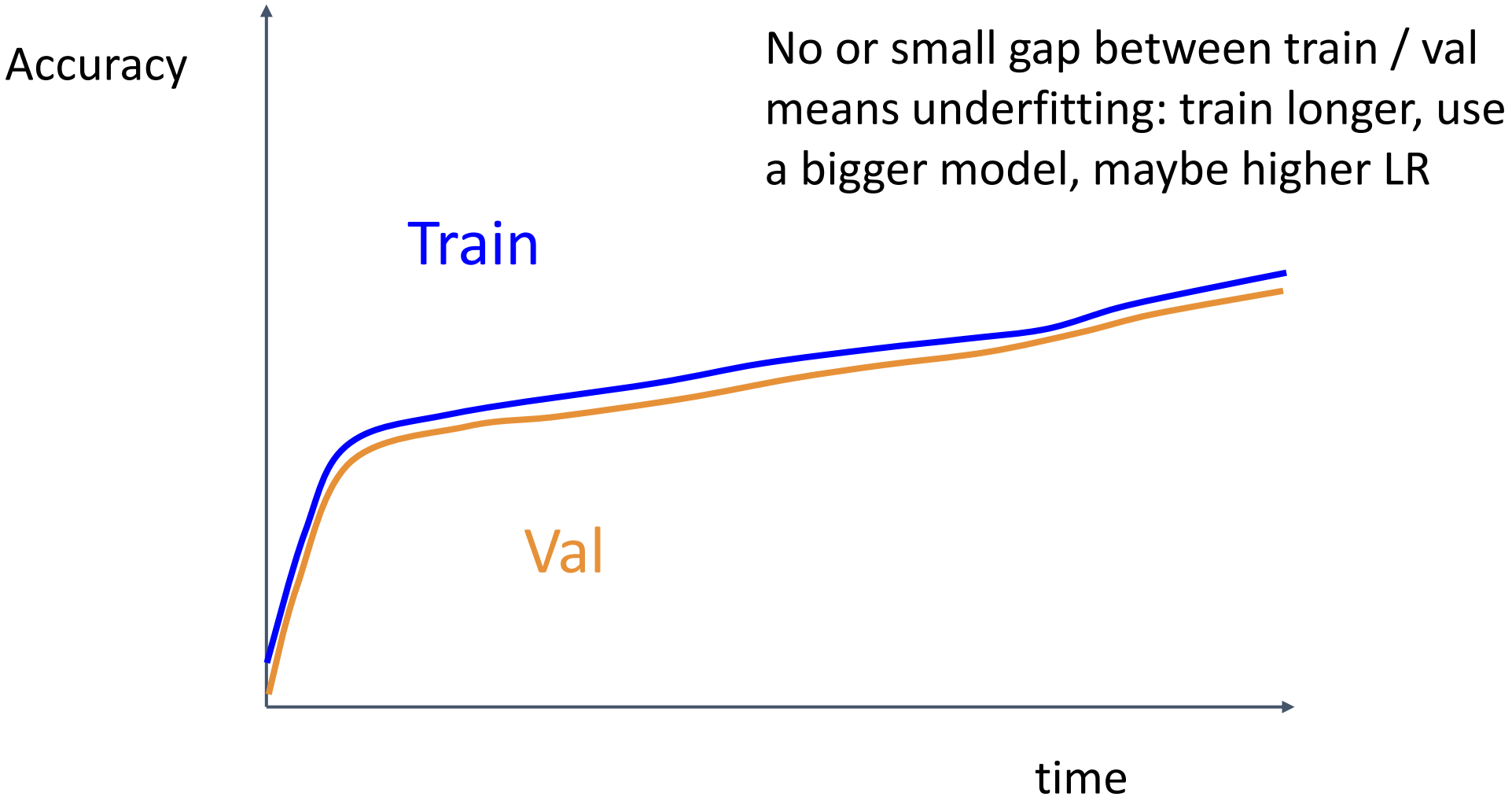

Loss
Bad initialization a prime suspect
time


Loss
Loss plateaus: Try learning rate decay
time


Loss
Learning rate step decay
Loss was still going down when learning rate dropped, you decayed too early!
time


Accuracy
Accuracy still going up, you need to train longer
Train
Val
time


Accuracy
Huge train / val gap means overfitting! Increase regularization, get more data
Train
Val
time


Accuracy
No or small gap between train / val means underfitting: train longer, use a bigger model, maybe higher LR
Train
Val
time

Choosing Hyperparameters

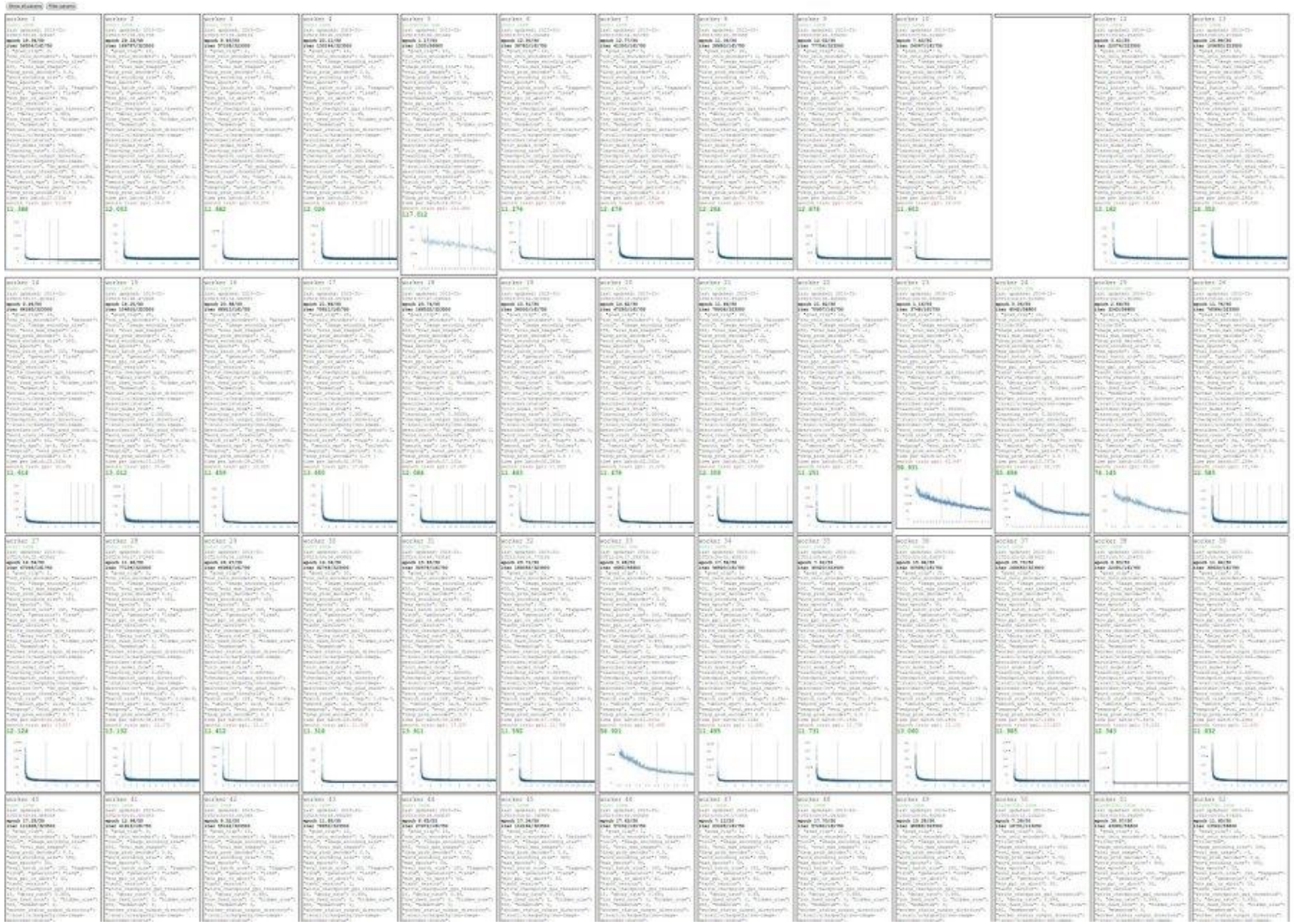
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Step 7: GOTO step 5

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

A large validation “command center”



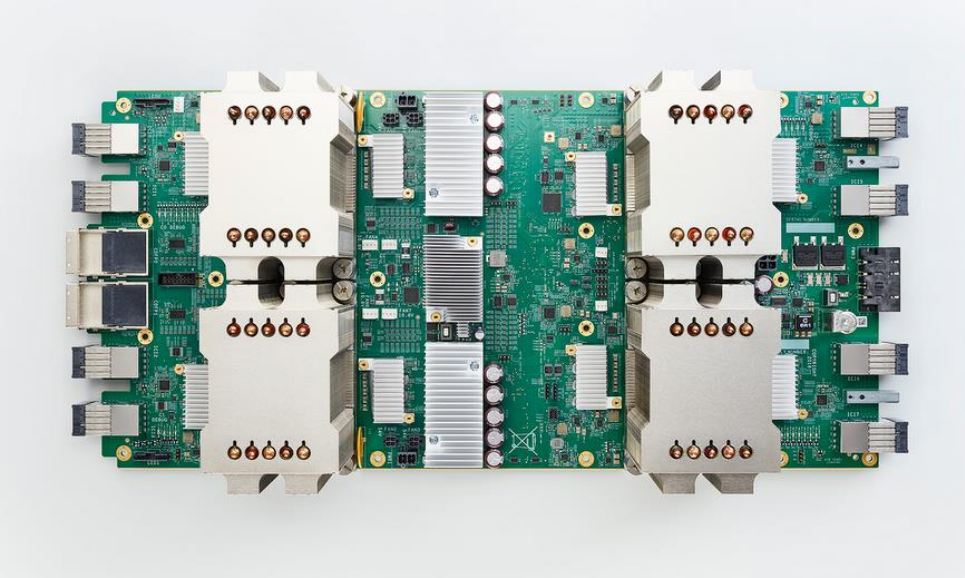
Track ratio of weight update / weight magnitude

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

Distributed Training and Large-Batch Training

Beyond individual devices



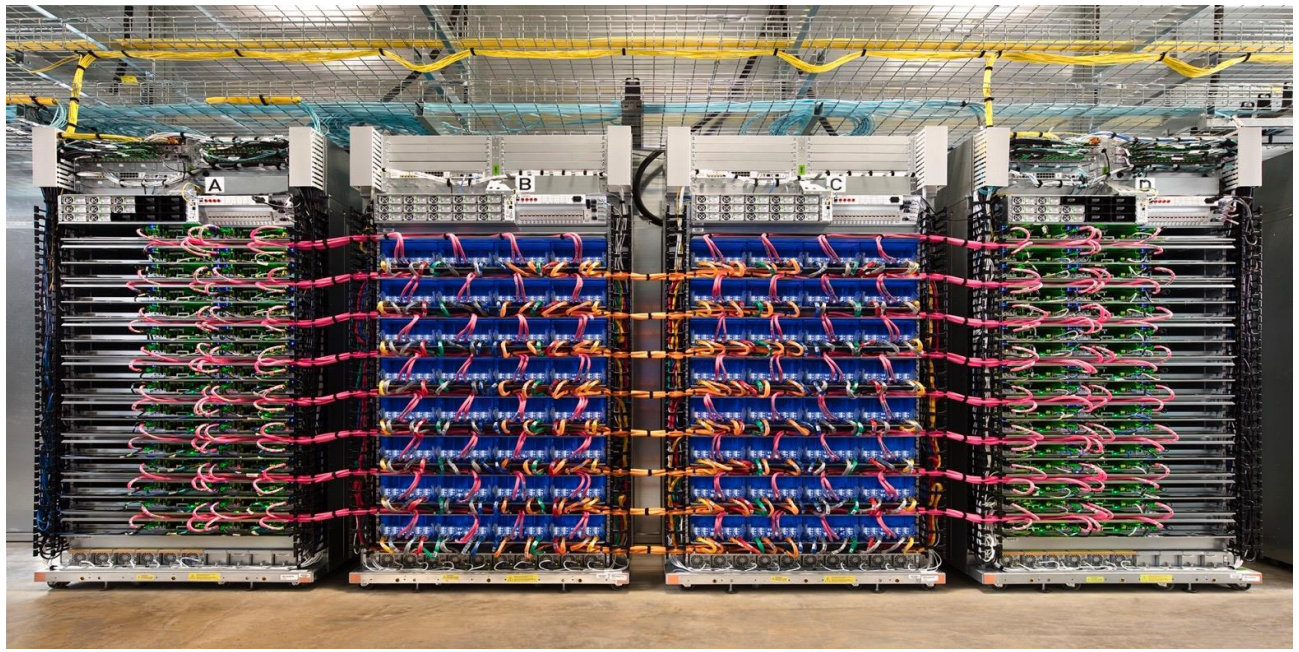
Cloud TPU v2

180 TFLOPs

64 GB HBM memory

$4.50 / hour

(free on Colab!)



Cloud TPU v2 Pod

64 TPU-v2

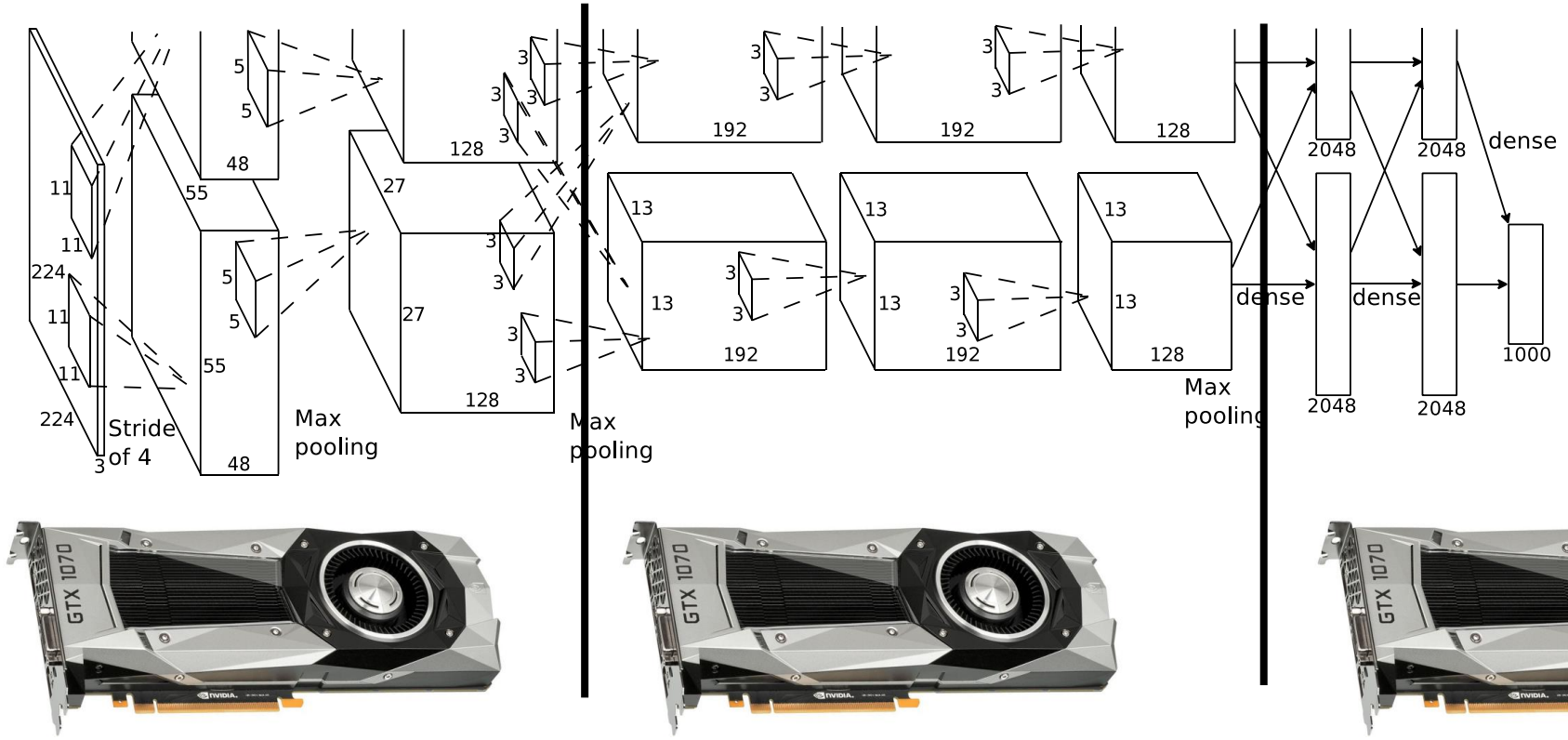
11.5 PFLOPs

$384 / hour

Model Parallelism: Split Model Across GPUs

Idea #1: Run different layers on different GPUs

Problem: GPUs spend lots of time waiting



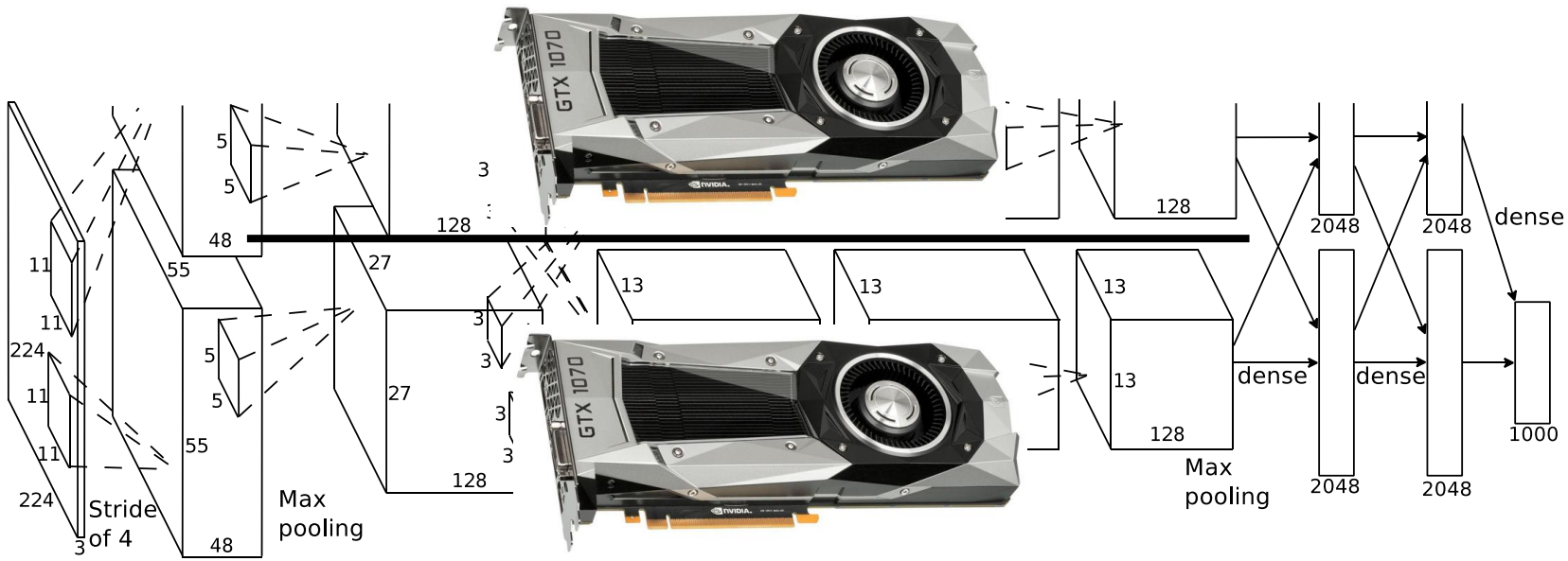
This image is in the public domain

Model Parallelism: Split Model Across GPUs

Idea #2: Run parallel branches of model on different GPUs

Problem: Synchronizing across GPUs is expensive;

Need to communicate **activations** and **grad activations**

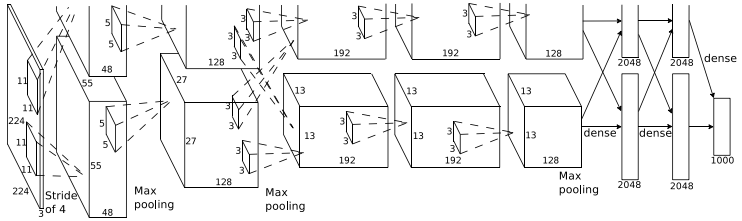


This image is in the public domain

Data Parallelism: Copy Model on each GPU, split data

Forward: compute loss

N / 2 images



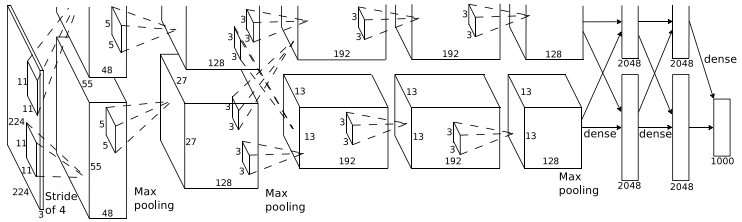
Backward: Compute gradient

Batch of N images

Exchange gradients, sum, update

Forward: compute loss

N / 2 images



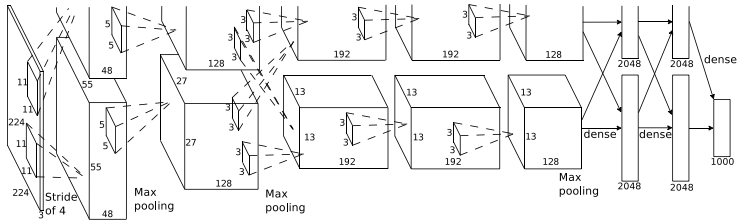
Backward: Compute gradient

GPUs only communicate once per iteration, and only exchange grad params

Data Parallelism: Copy Model on each GPU, split data

Batch of N images

N / K images

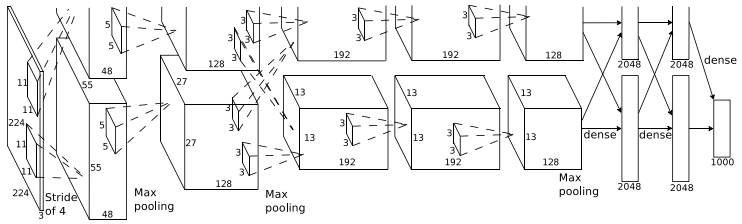


Scale up to K GPUs (within and across servers)

...

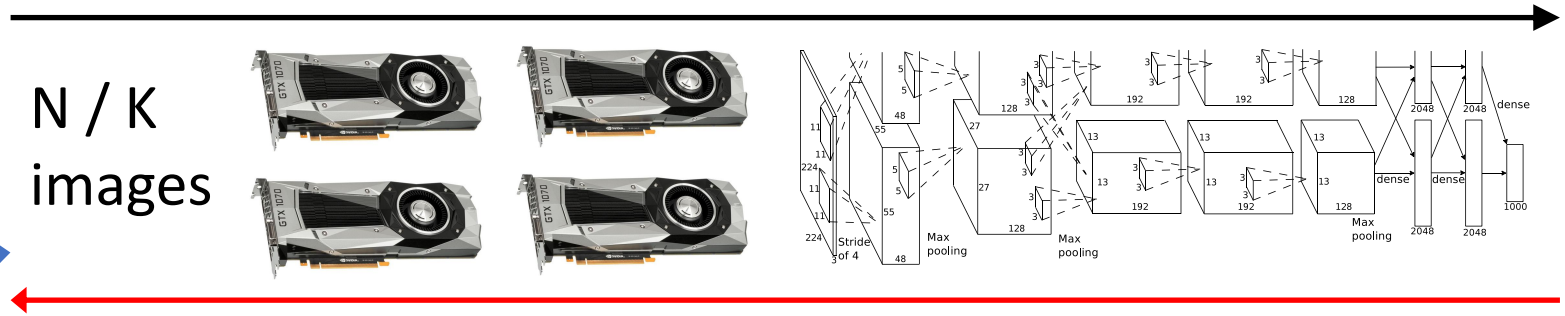
Main mechanism for distributing across many GPUs today

N / K images



Mixed Model + Data Parallelism

Problem: Need to train with very large minibatches!

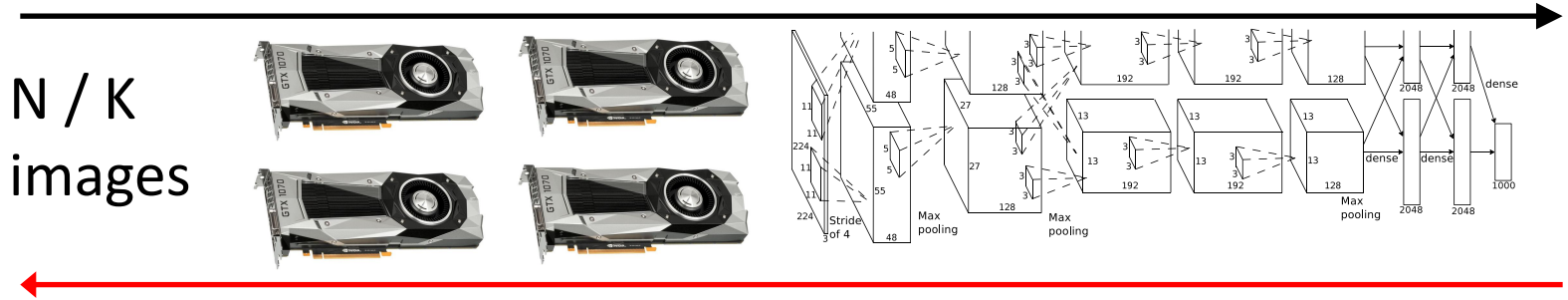


Batch of N images

Model parallelism **within** each server

...

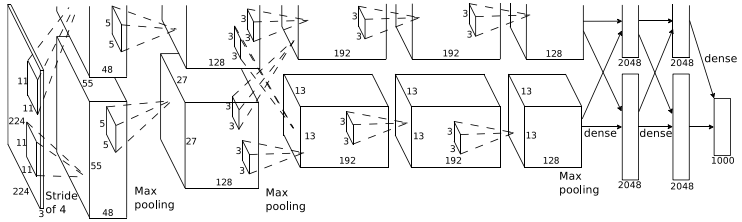
Data parallelism **across** K servers



Example: https://devblogs.nvidia.com/training-bert-with-gpus/

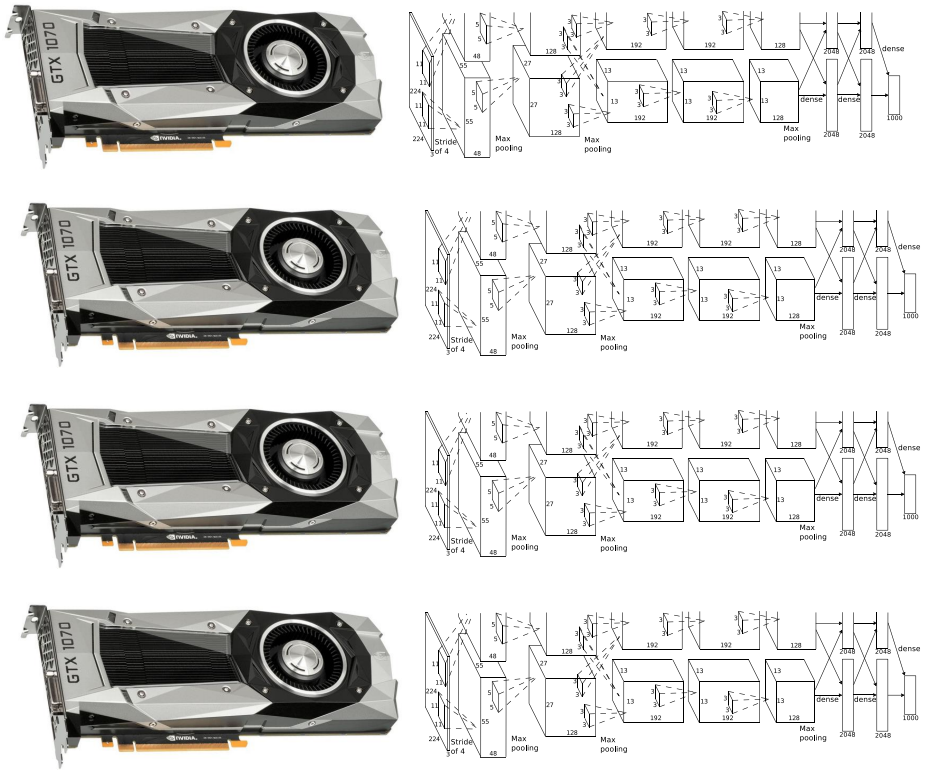
Large-Batch Training

Suppose we can train a good model with one GPU



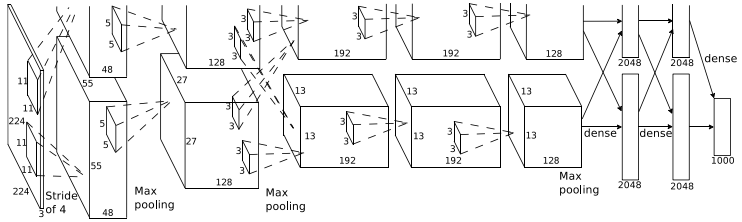
Goal: Train for same number of epochs, but use larger minibatches.
We want model to train K times faster!

How to scale up to data-parallel training on K GPUs?

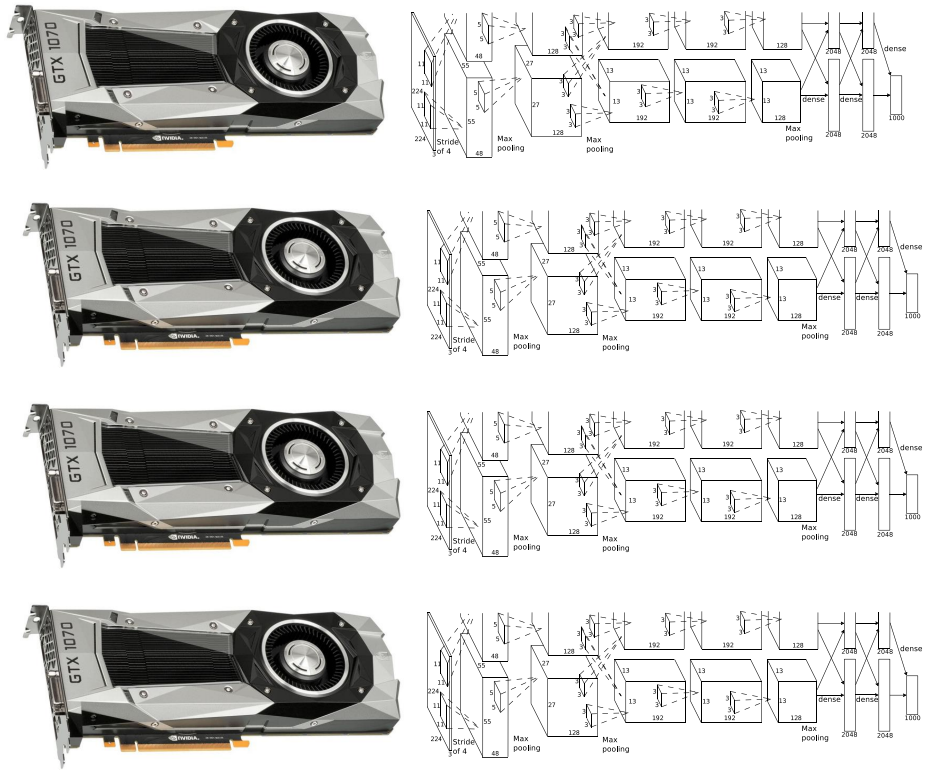


Large-Batch Training: Scale Learning Rates

Single-GPU model:
batch size N, learning rate α

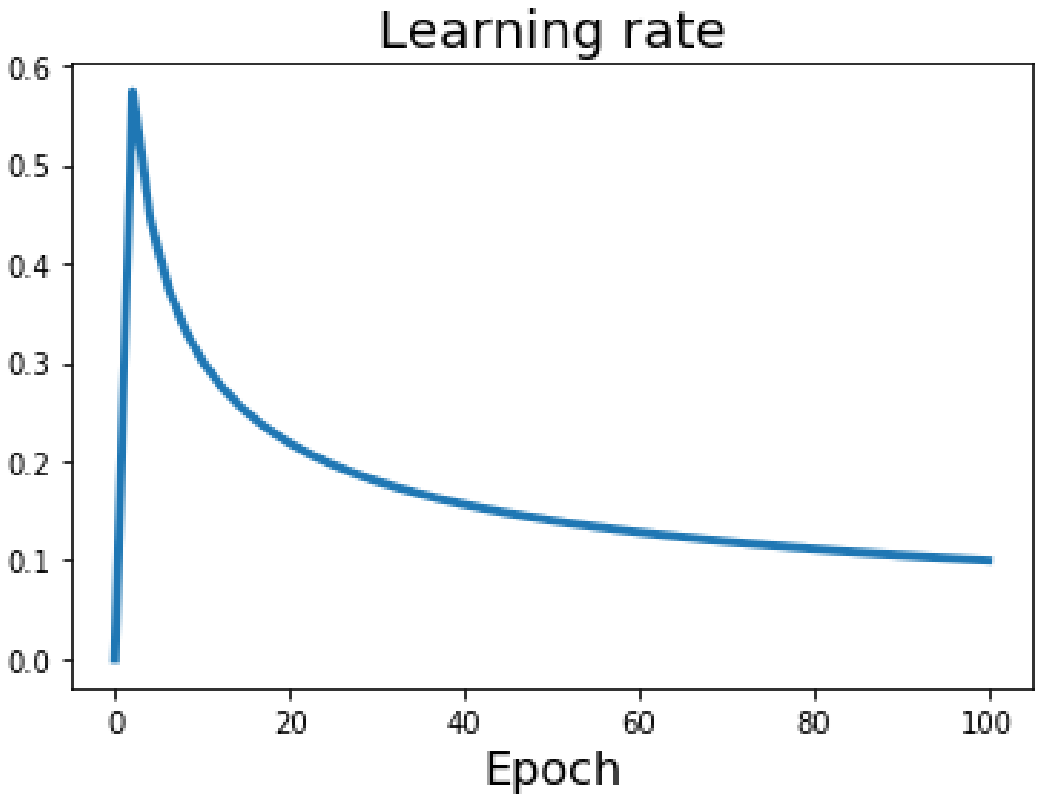


K-GPU model:
batch size KN, learning rate Kα



Alex Krizhevsky, “One weird trick for parallelizing convolutional neural networks”, arXiv 2014
Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

Large-Batch Training: Learning Rate Warmup



High initial learning rates can make loss explode; linearly **increasing** learning rate from 0 over the first ~5000 iterations can prevent this

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

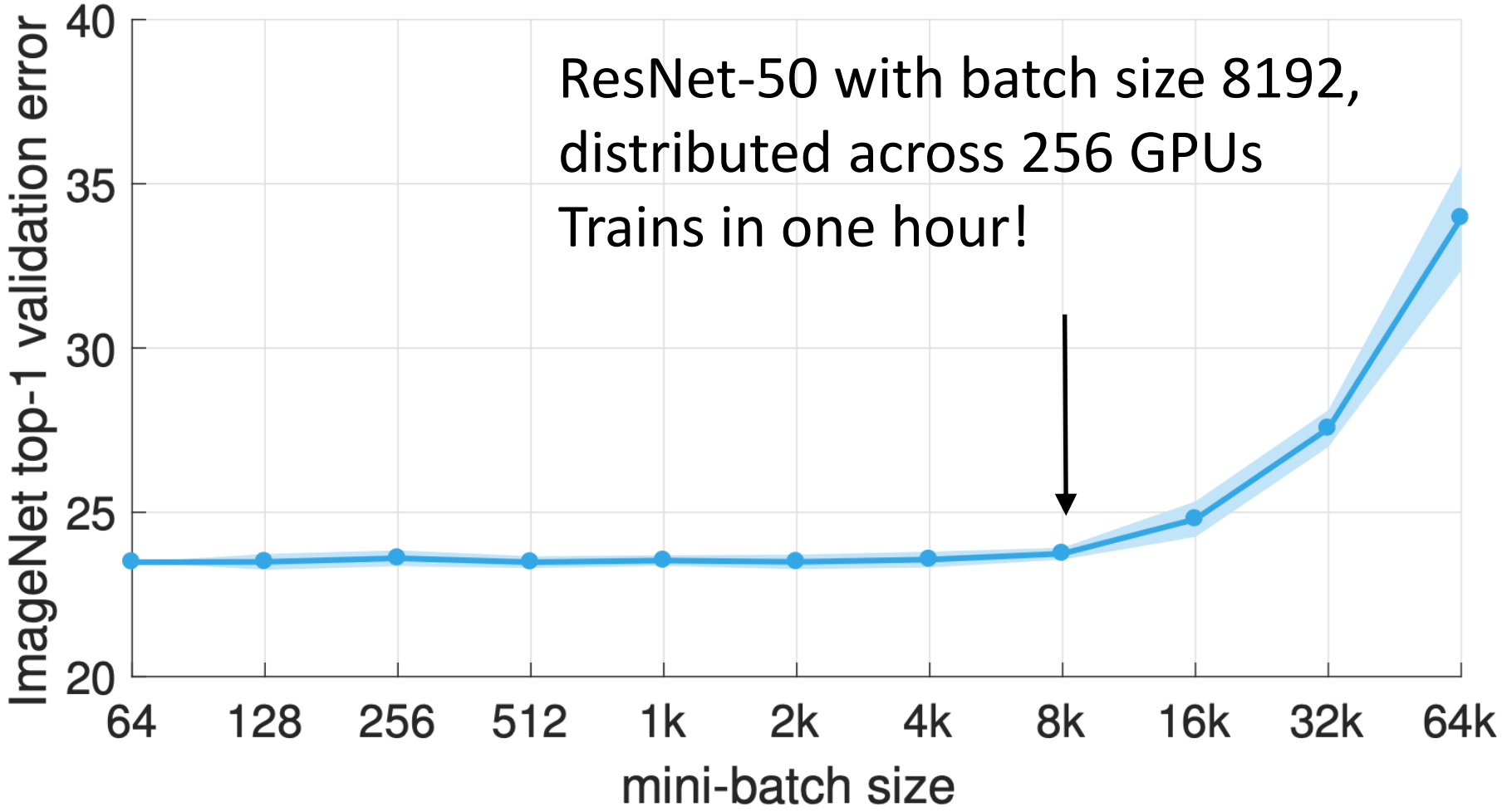
Large-Batch Training: Other Concerns

Be careful with **weight decay** and **momentum**, and **data shuffling**

For Batch Normalization, only normalize **within a GPU**
Often use **SyncBatchNorm** for synchronizing batch statistics over GPUs

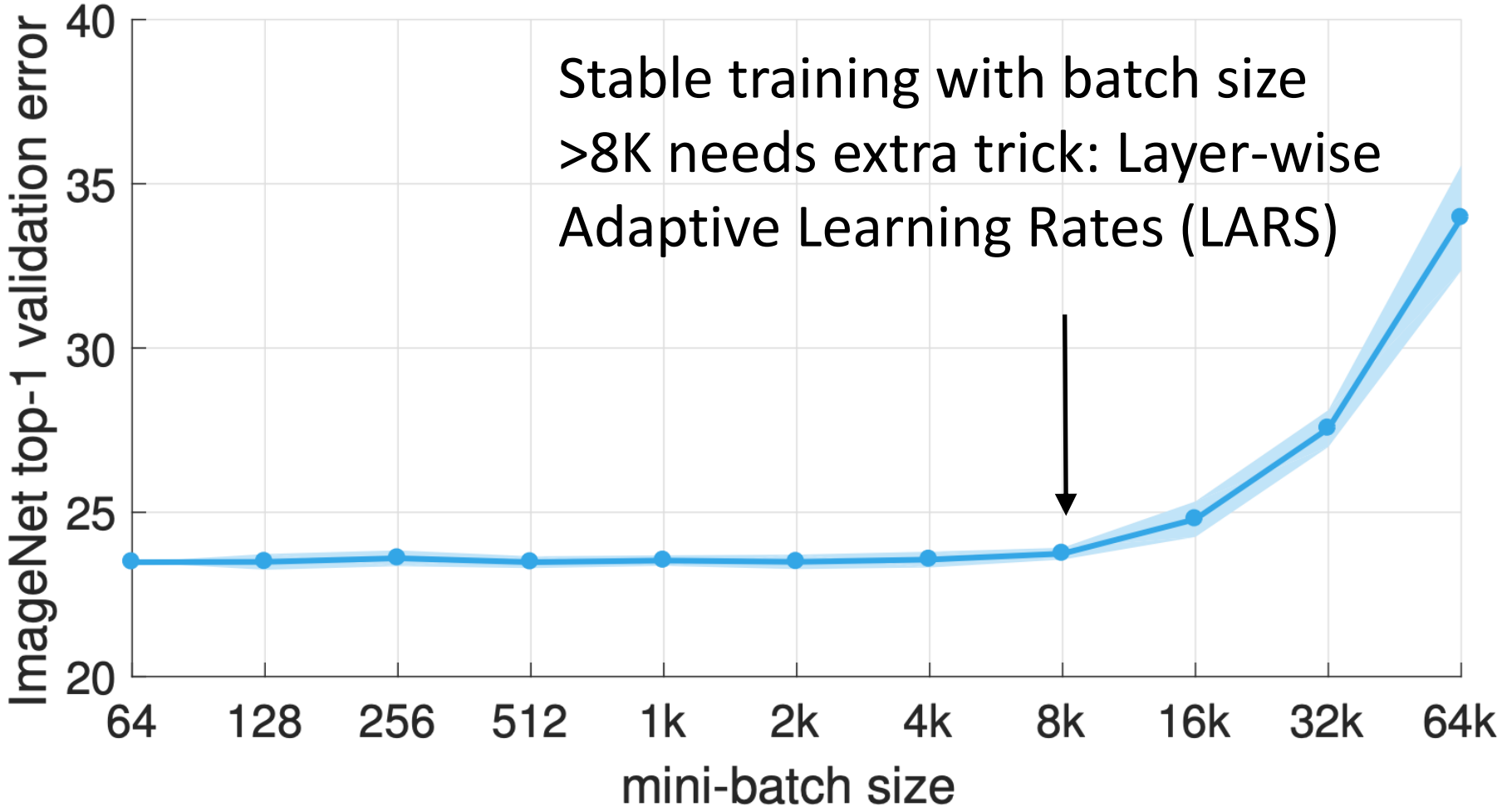
Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

Large-Batch Training: ImageNet in One Hour!



Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

Large-Batch Training: ImageNet in One Hour!



Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

You et al, "Large Batch Training of Convolutional Networks", arXiv 2017

Large-Batch Training

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, 2017
Batch size: 8192; 256 P100 GPUs; 1 hour

Codreanu et al, “Achieving deep learning training in less than 40 minutes on imagenet-1k”, 2017
Batch size: 12288; 768 Knight’s Landing devices; 39 minutes

You et al, “ImageNet training in minutes”, 2017
Batch size: 16000; 1600 Xeon CPUs; 31 minutes

Akiba et al, “Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes”, 2017
Batch size: 32768; 1024 P100 GPUs; 15 minutes

MLPerf v0.7 (NVIDIA): 1840 A100 GPUs; **46 seconds**

MLPerf v0.7 (Google): TPU-v3-8192; Batch size 65536; **27 seconds!**

https://mlperf.org/

Overview

1. **One time setup**
   Activation functions, data preprocessing,
   weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules, large-batch training,
   hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

Model ensembles

Model Ensembles

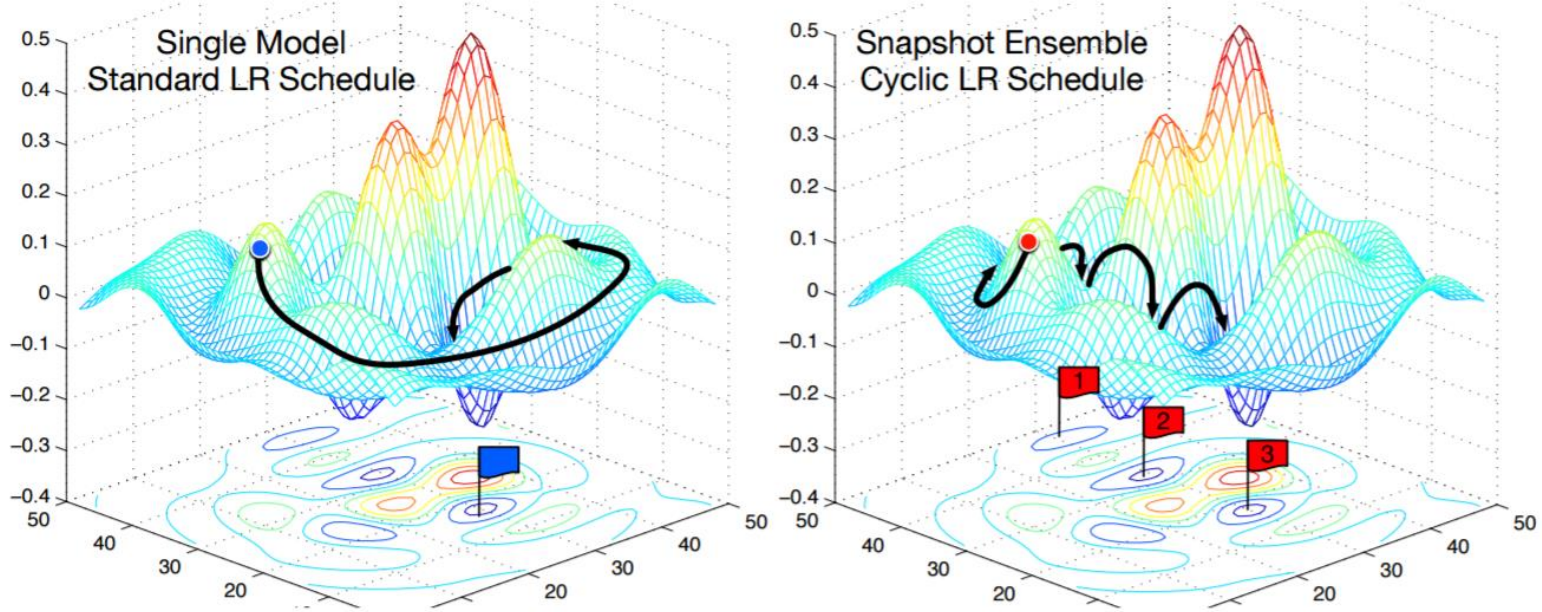
1. Train multiple independent models
2. At test time average their results

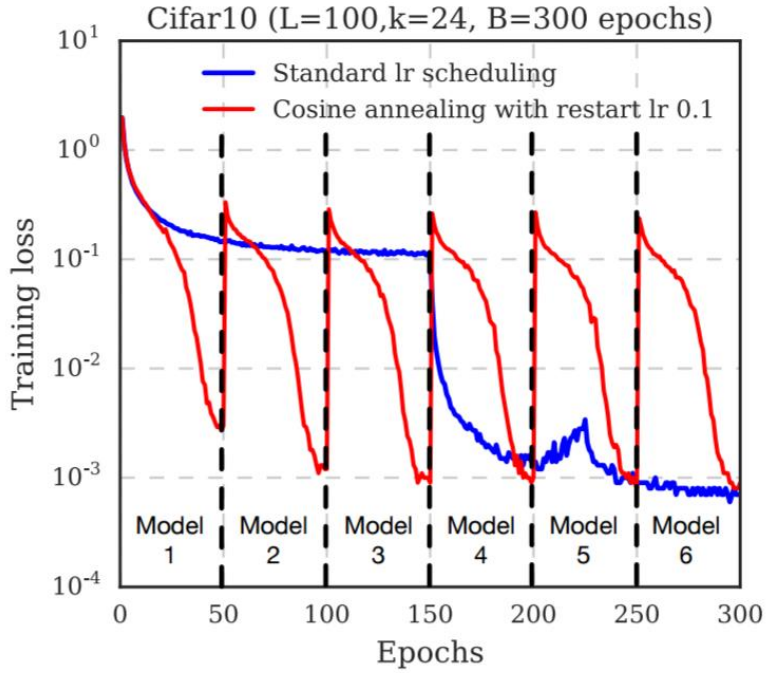
   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!





Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, “Acceleration of stochastic approximation by averaging”, SIAM Journal on Control and Optimization, 1992.
Karras et al, “Progressive Growing of GANs for Improved Quality, Stability, and Variation”, ICLR 2018
Brock et al, “Large Scale GAN Training for High Fidelity Natural Image Synthesis”, ICLR 2019

Transfer Learning

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

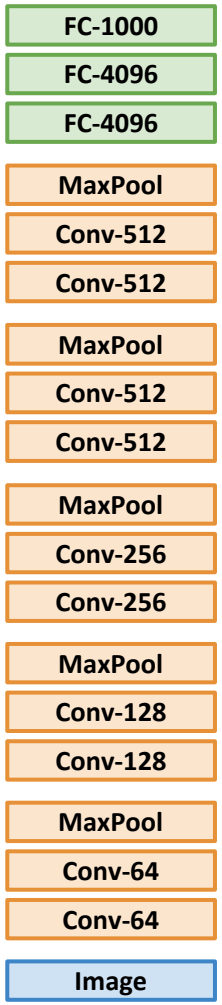
Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

Not Really

Transfer Learning with CNNs: Feature Extraction

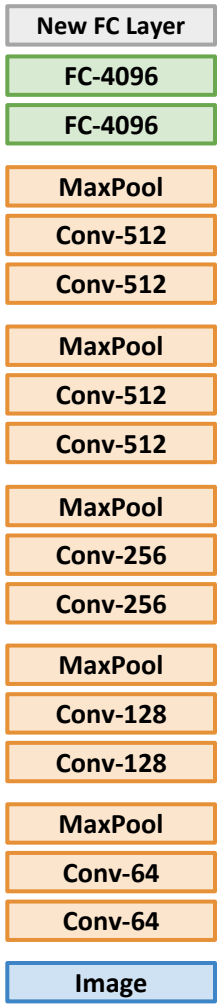
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

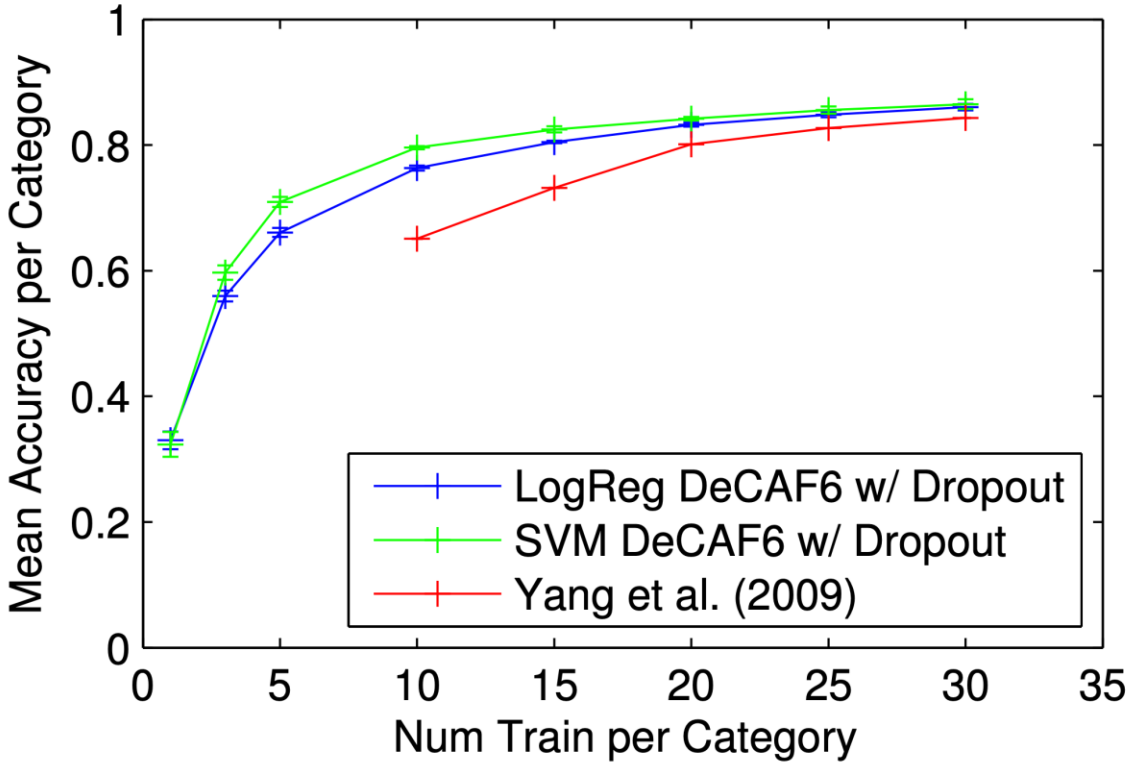
2. Use CNN as a feature extractor



e.g., Add randomly initialized final FC layer for new task

Freeze these

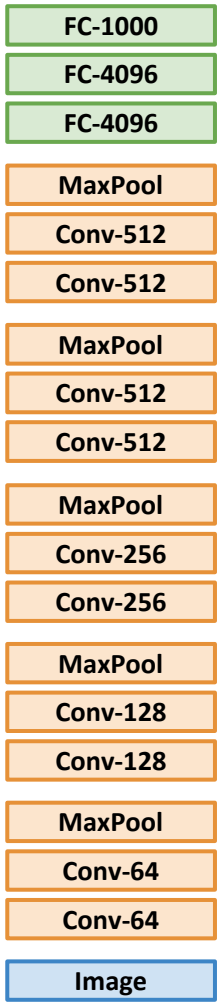
Classification on Caltech-101



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs: Feature Extraction

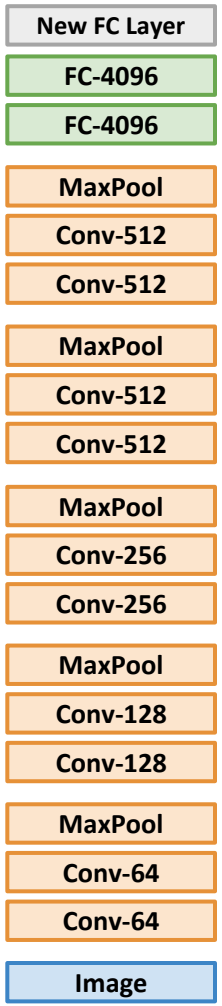
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

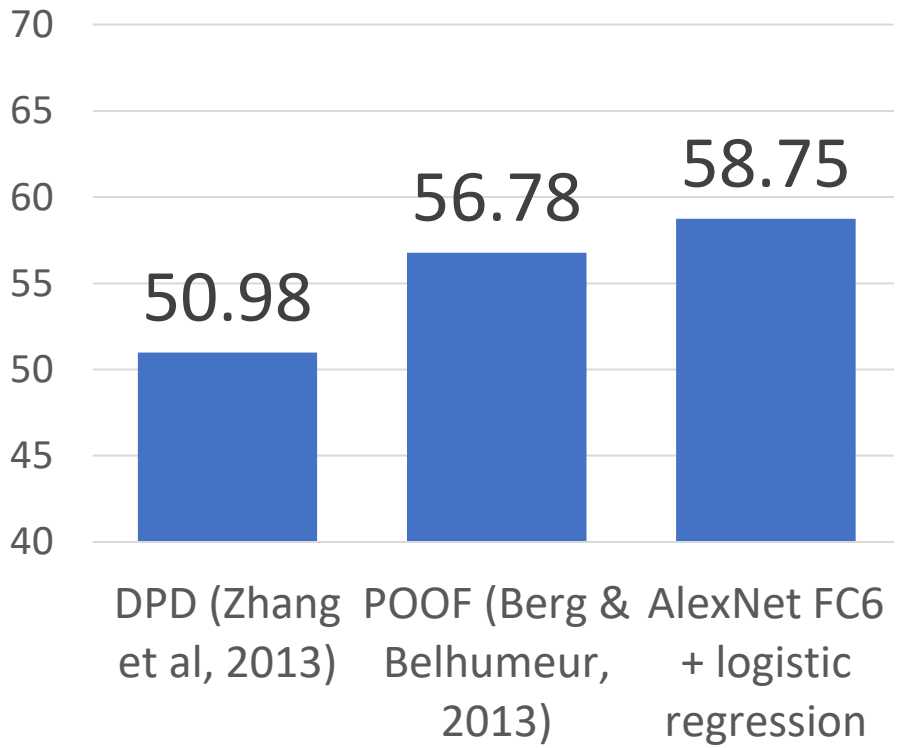
2. Use CNN as a feature extractor



e.g., Add randomly initialized final FC layer for new task

Freeze these

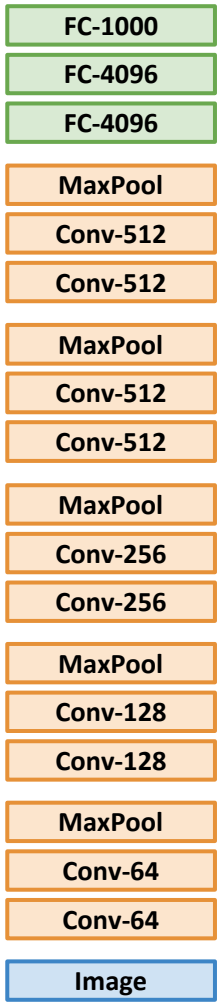
Bird Classification on Caltech-UCSD



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs: Feature Extraction

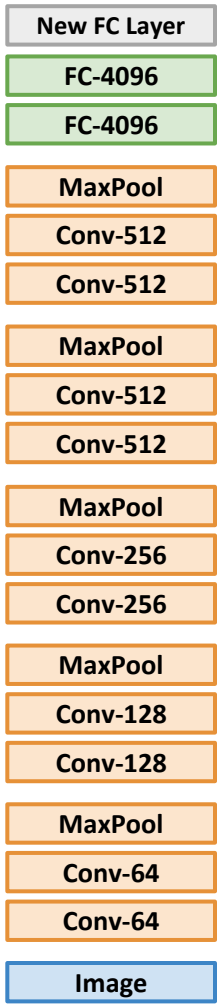
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

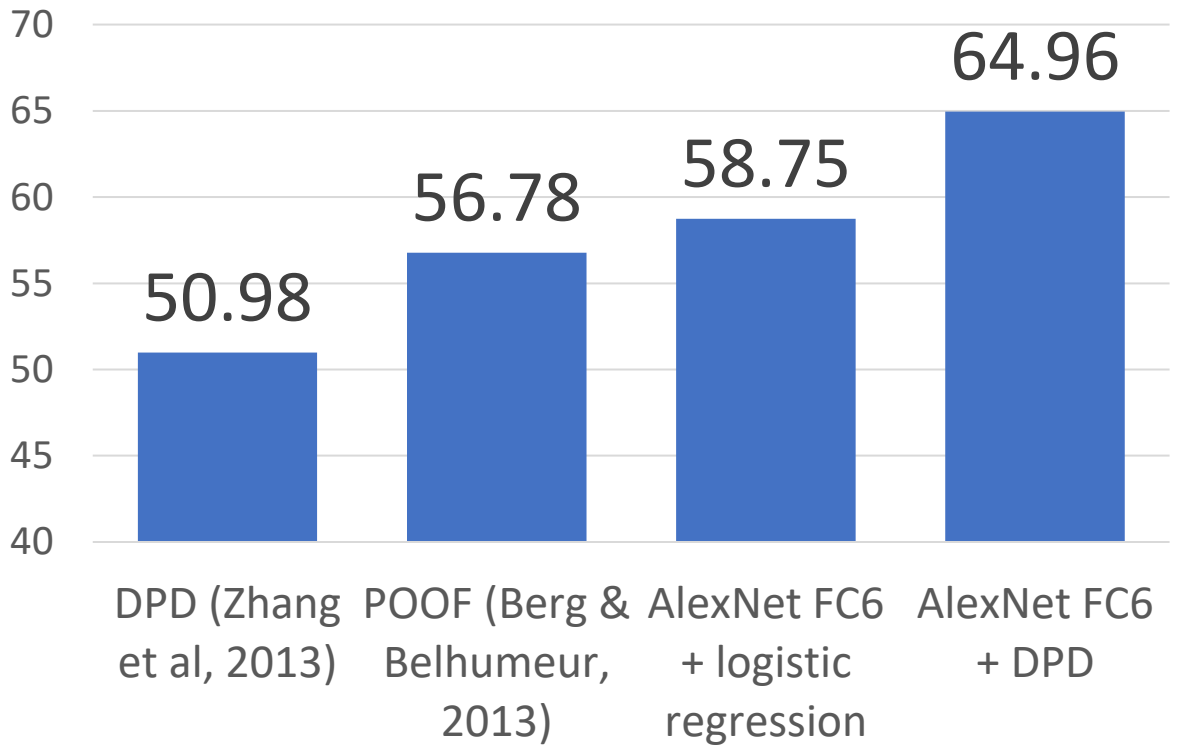
2. Use CNN as a feature extractor



e.g., Add randomly initialized final FC layer for new task

Freeze these

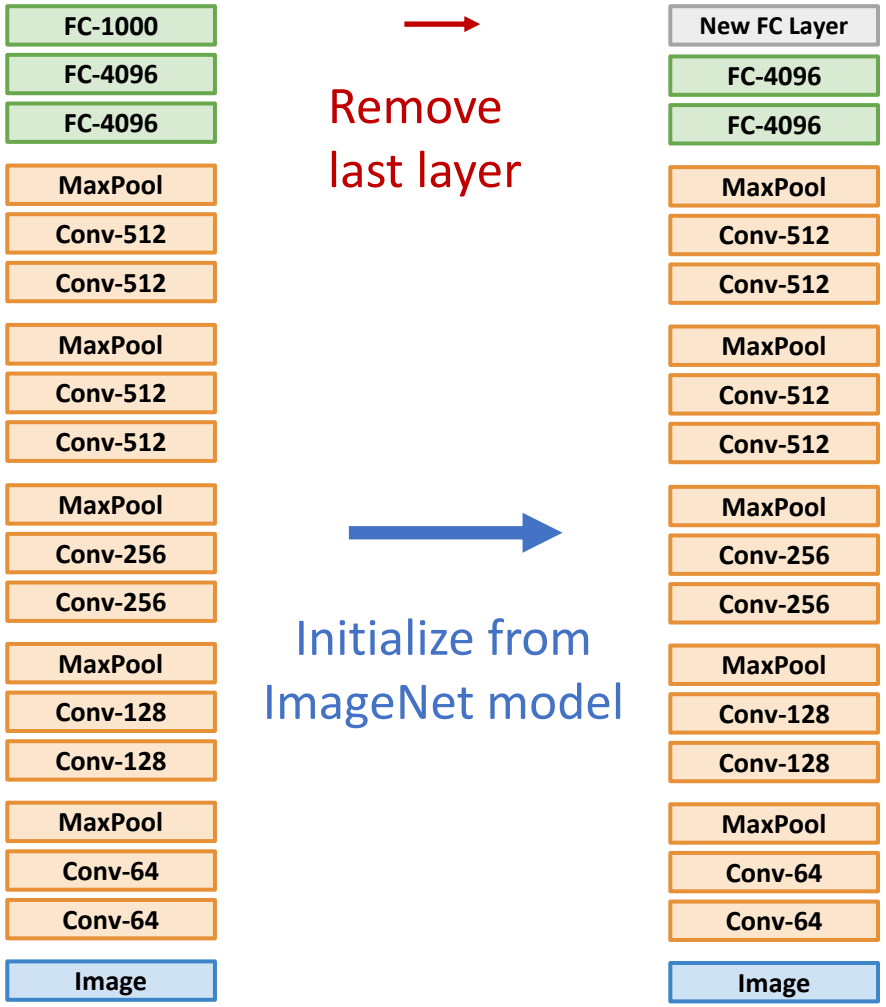
Bird Classification on Caltech-UCSD



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs: Feature Extraction

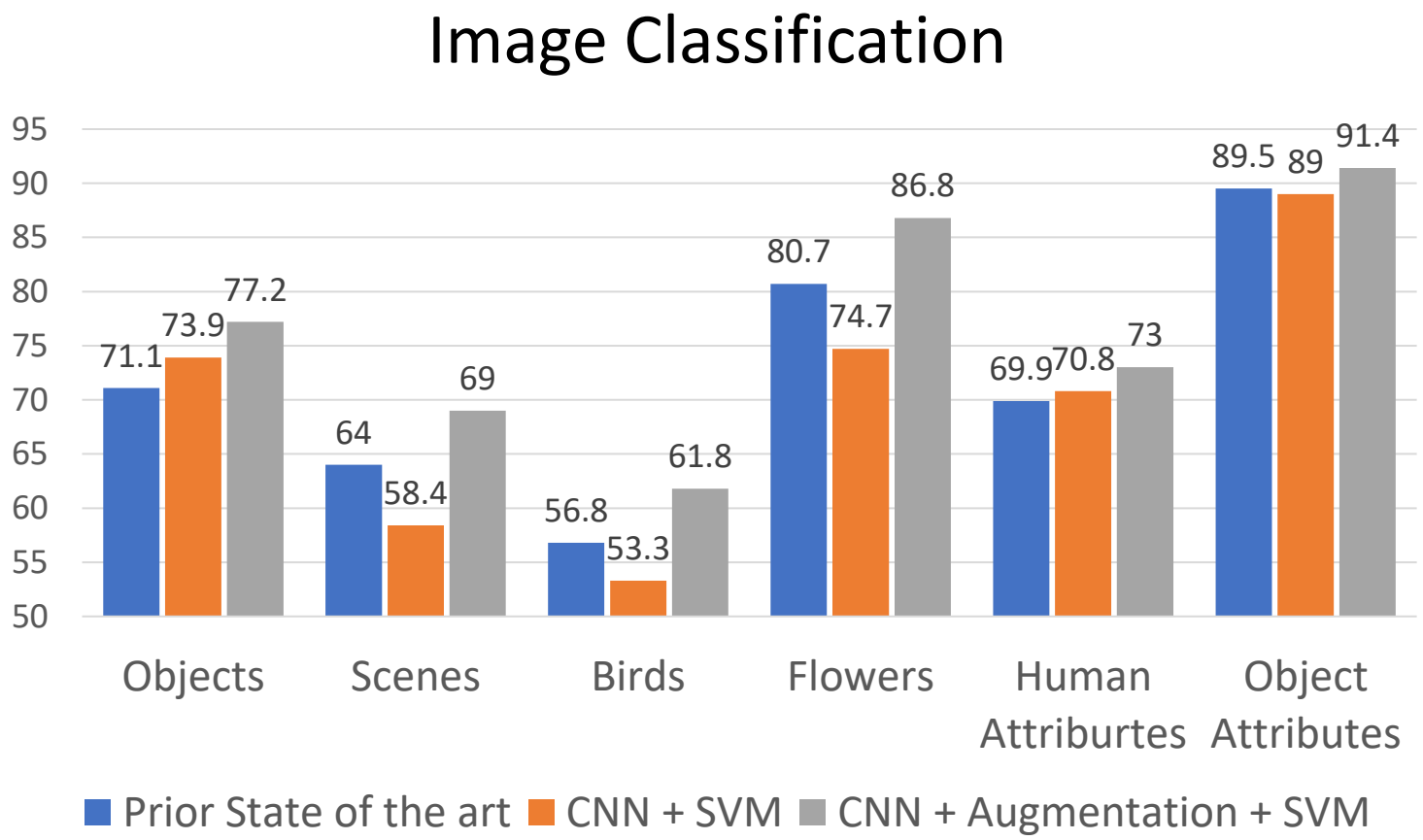
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

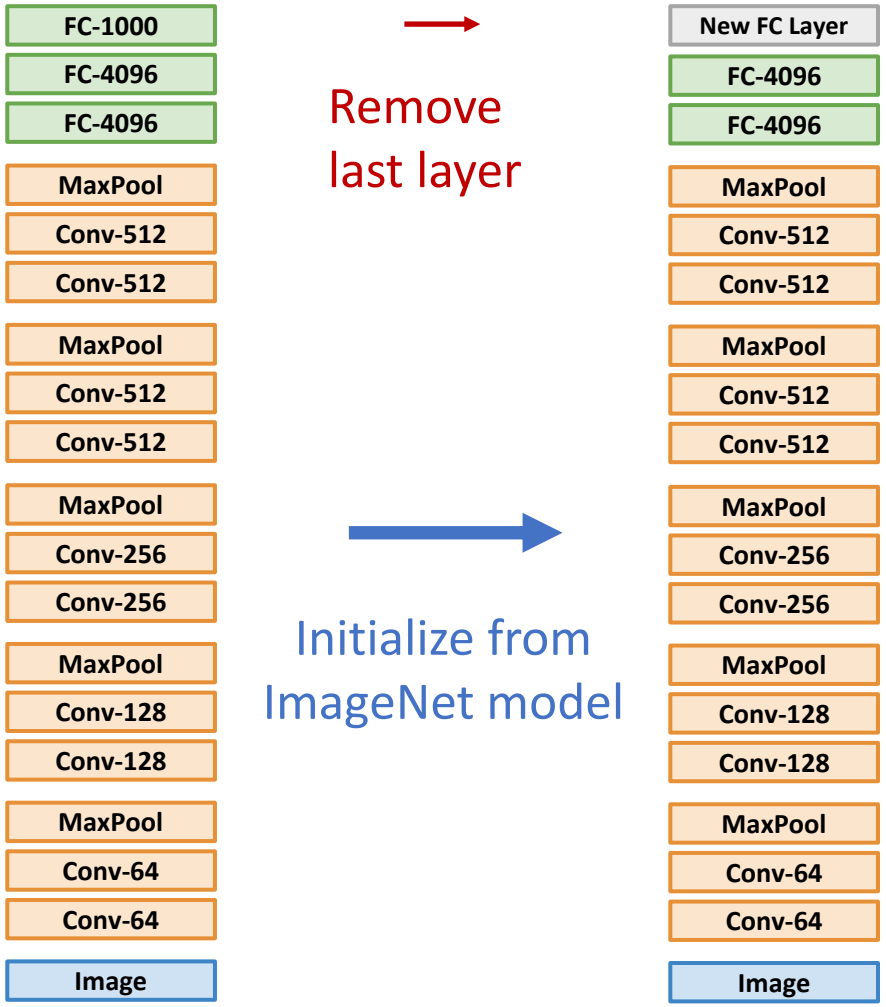
2. Use CNN as a feature extractor



Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs: Feature Extraction

1. Train on ImageNet

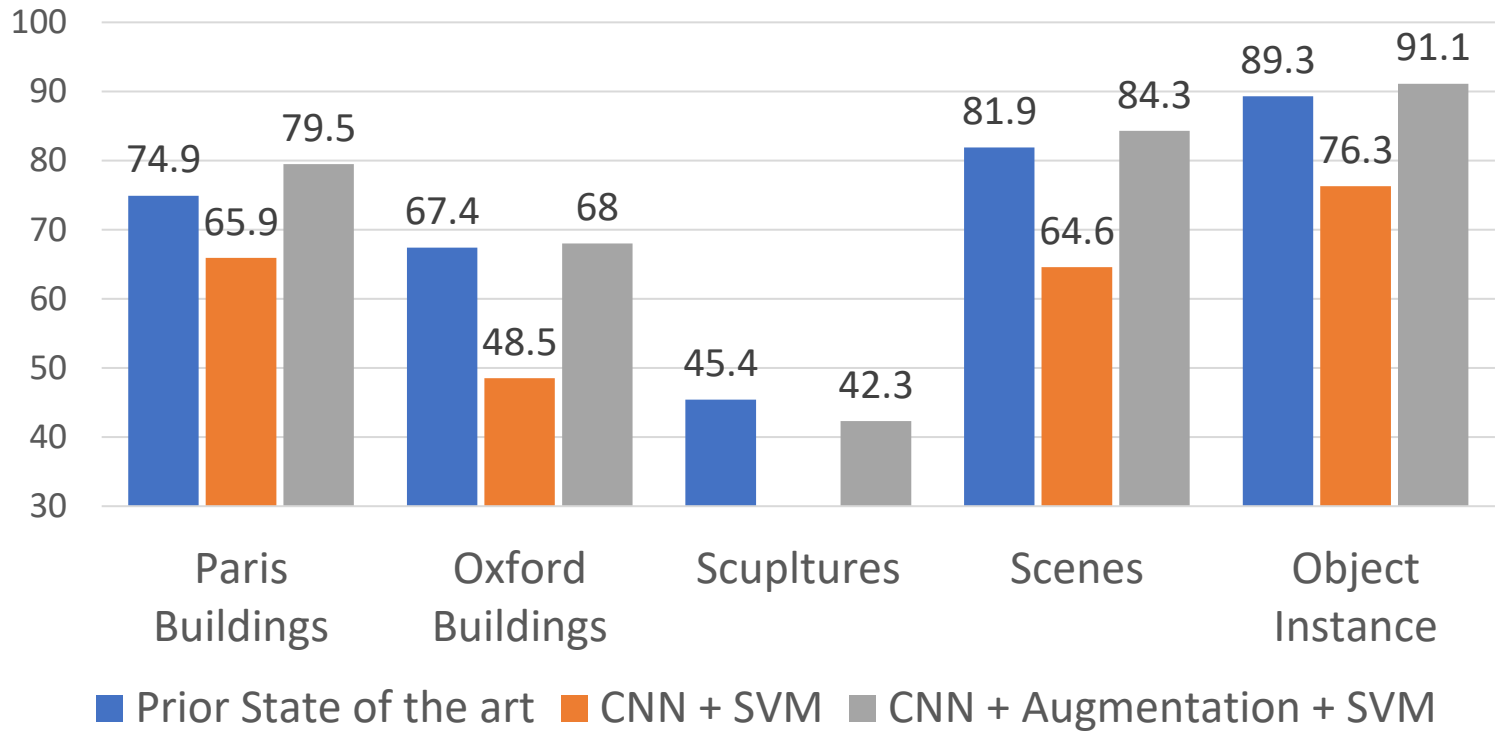


Remove last layer

Initialize from ImageNet model

2. Use CNN as a feature extractor

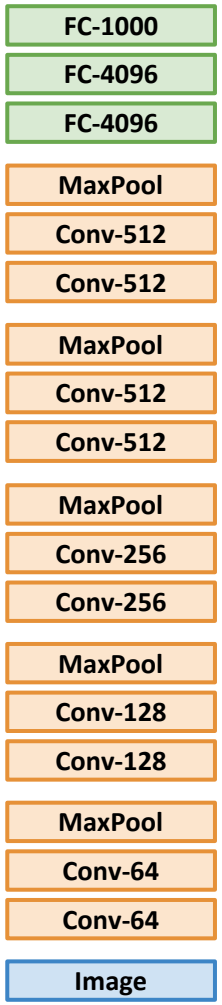
Image Retrieval: Nearest-Neighbor



Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs: Fine-Tuning

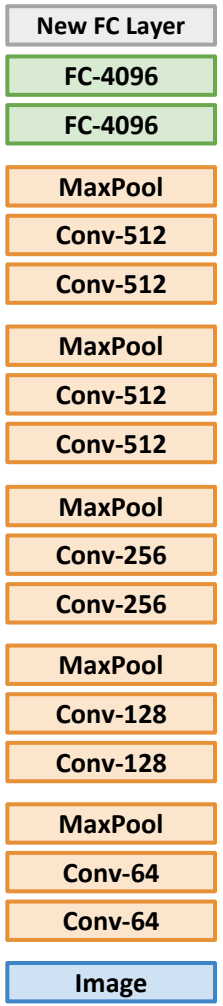
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

2. Fine-tune CNN



Add randomly initialized final FC layer for new task

Continue training entire model for new task

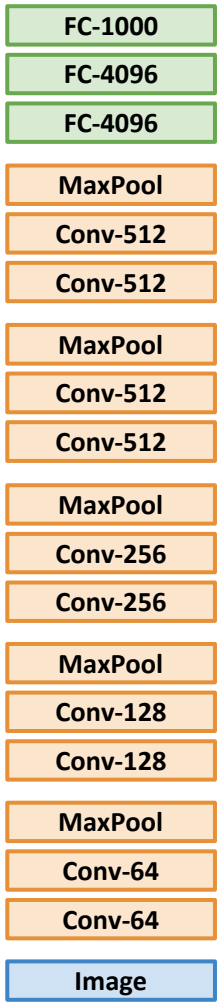
Some tricks:

- Train with frozen feature extraction first before fine-tuning
- Lower the learning rate: use ~1/10 of LR used in original training
- Sometimes freeze lower layers to save computation
- Train with BatchNorm in “test” mode

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014

Transfer Learning with CNNs: Fine-Tuning

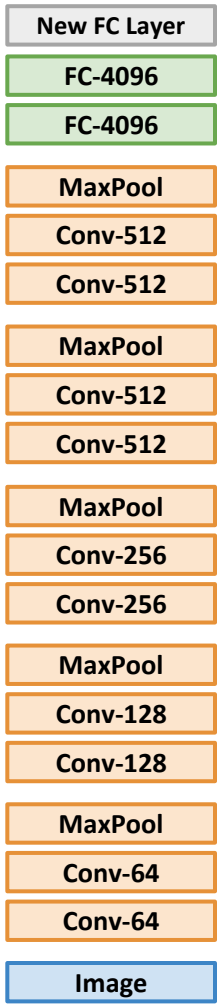
1. Train on ImageNet



Remove last layer

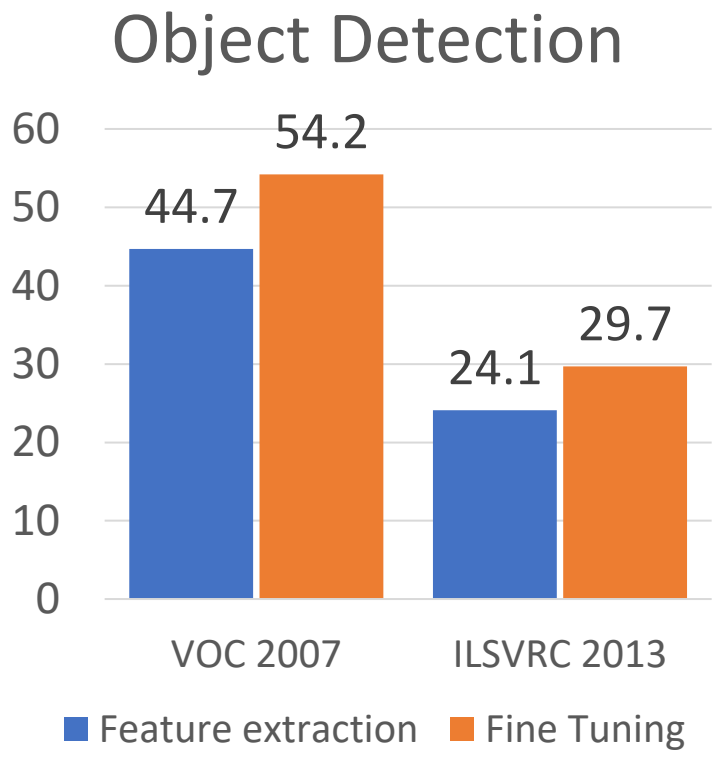
Initialize from ImageNet model

2. Fine-tune CNN



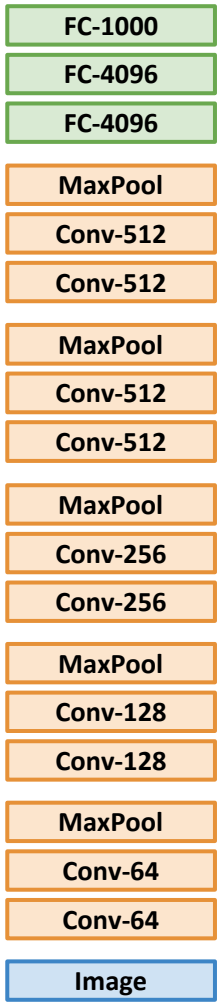
Add randomly initialized final FC layer for new task

Continue training entire model for new task



Transfer Learning with CNNs: Fine-Tuning

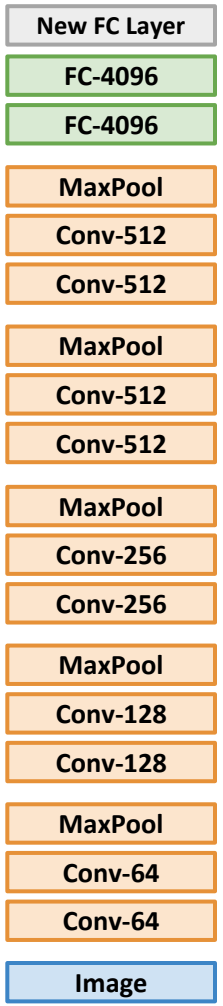
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

2. Fine-tune CNN



Add randomly initialized final FC layer for new task

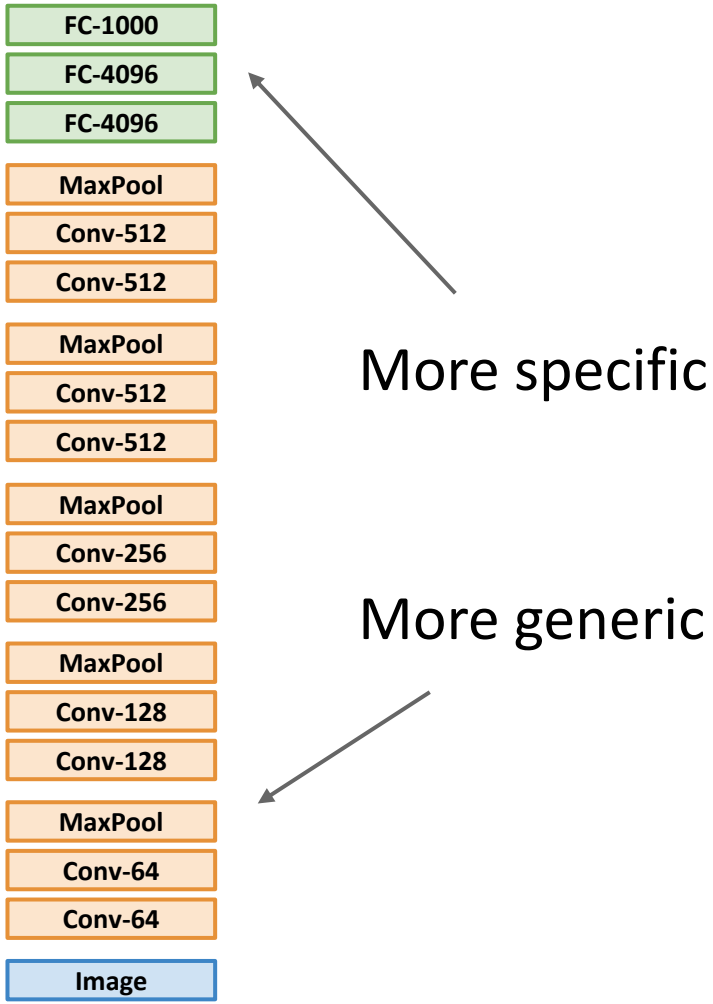
Continue training entire model for new task

Compared with Feature Extraction, Fine-Tuning:

- Requires more data
- Is more computationally expensive
- Can give higher accuracies

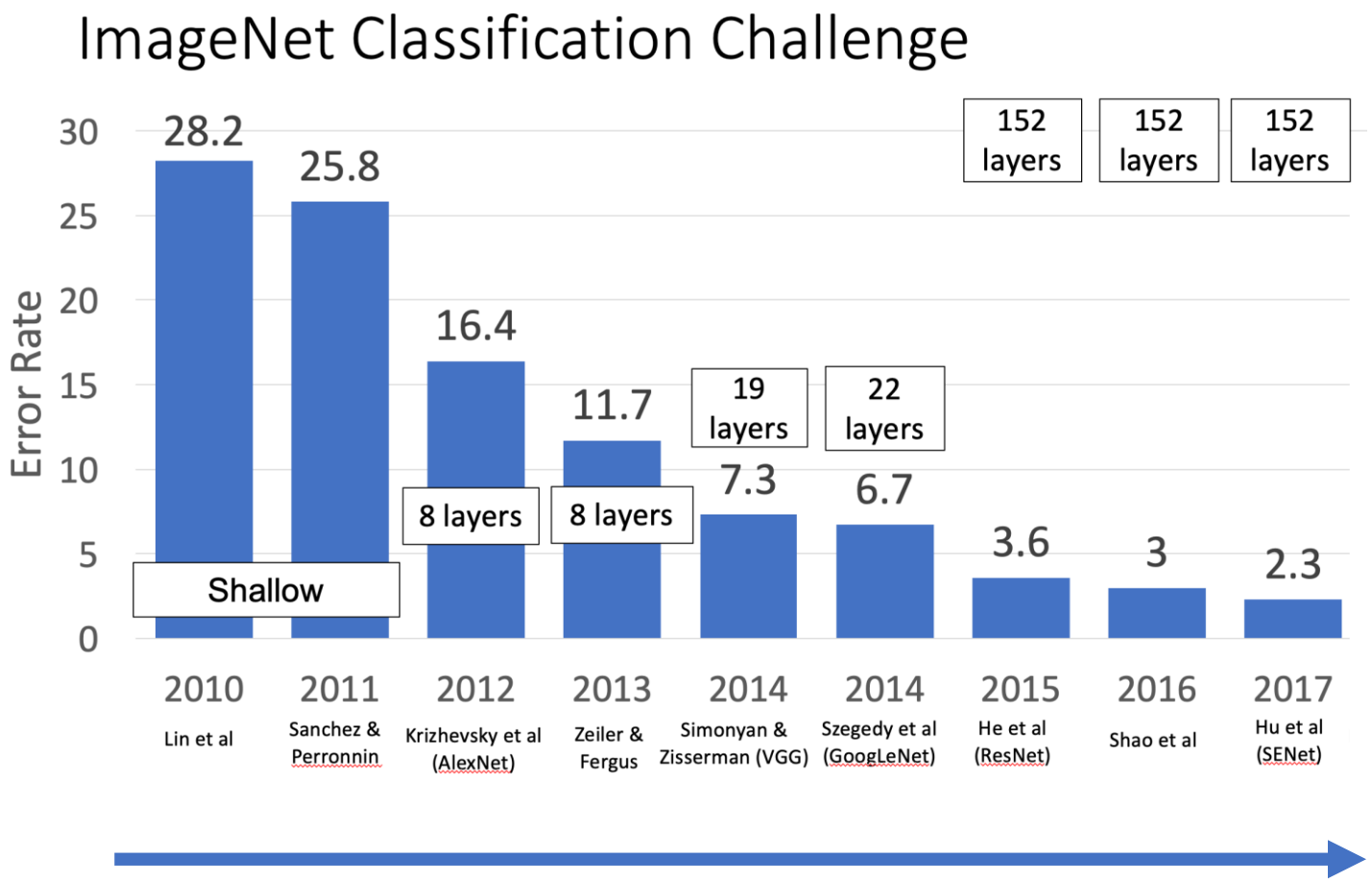
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs



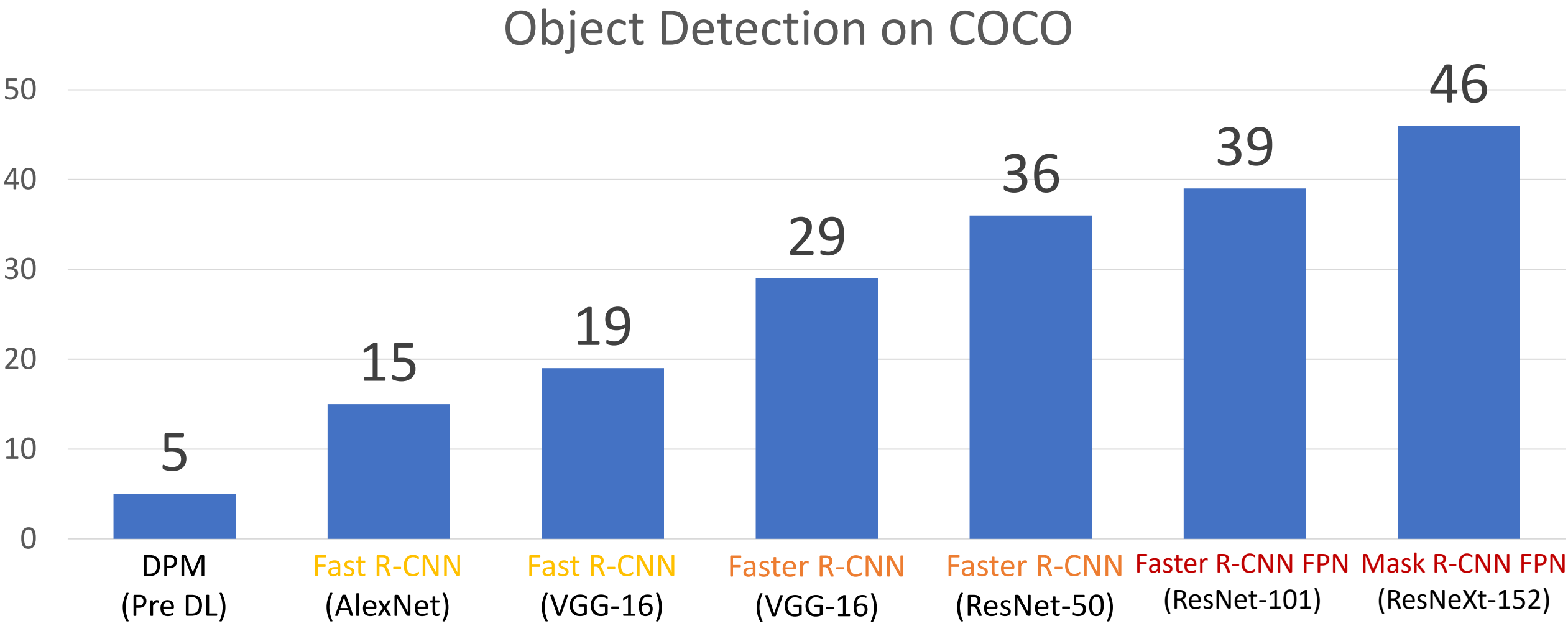
	Dataset similar to ImageNet	**Dataset very different from ImageNet**
very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Transfer Learning with CNNs: Architecture Matters!



Improvements in CNN architectures lead to improvements in many downstream tasks thanks to transfer learning!

Transfer Learning with CNNs: Architecture Matters!

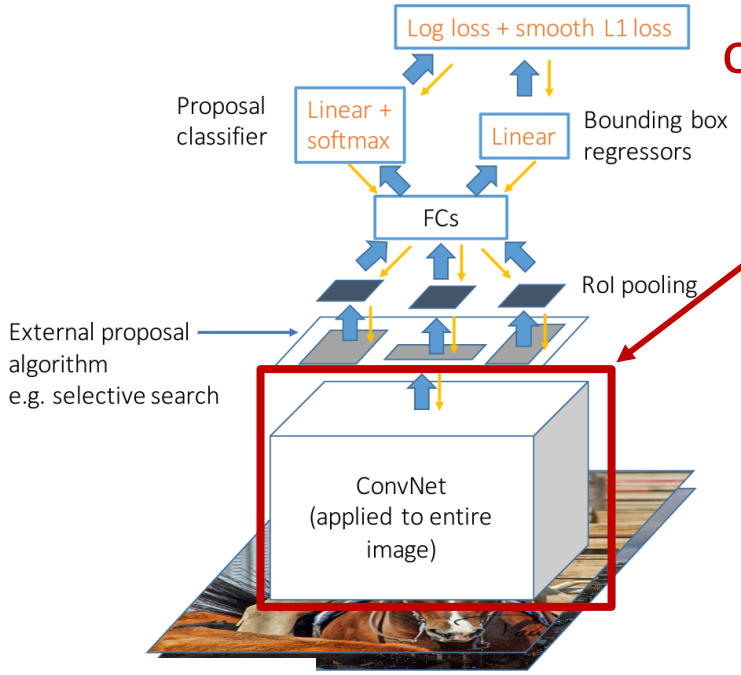


Ross Girshick, "The Generalized R-CNN Framework for Object Detection", ICCV 2017 Tutorial on Instance-Level Visual Recognition

Transfer learning is pervasive!

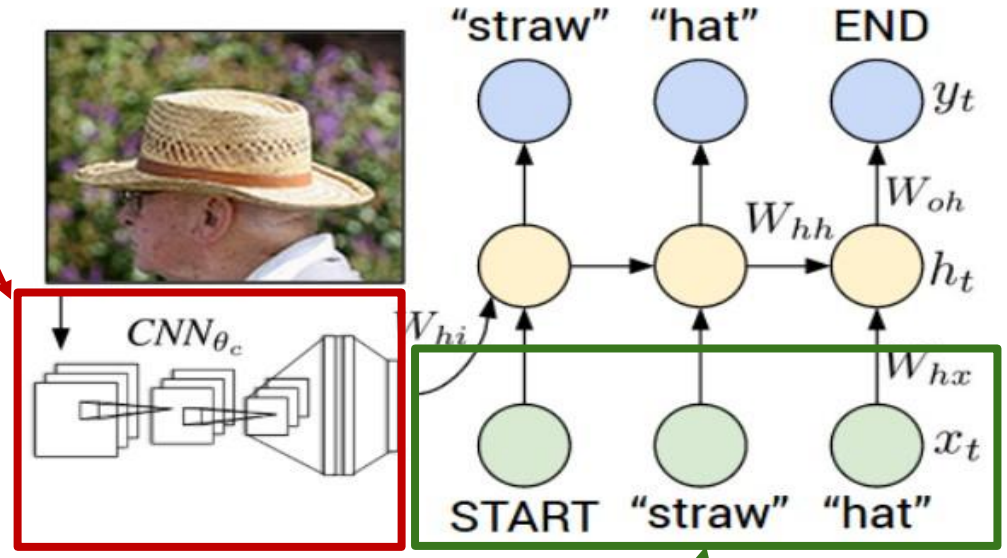
It's the norm, not the exception

Object Detection
(Fast R-CNN)



CNN pretrained
on ImageNet

Image Captioning

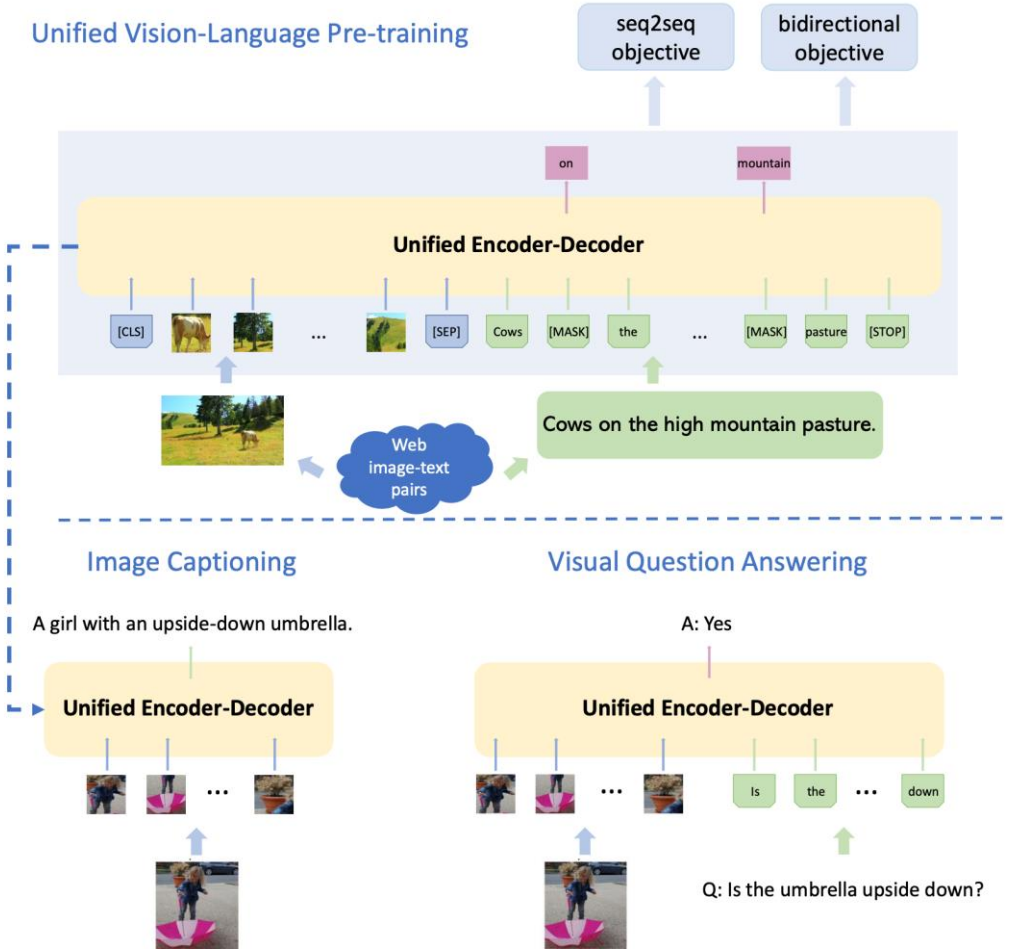


Word vectors pretrained
with word2vec

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning is pervasive!
It's the norm, not the exception

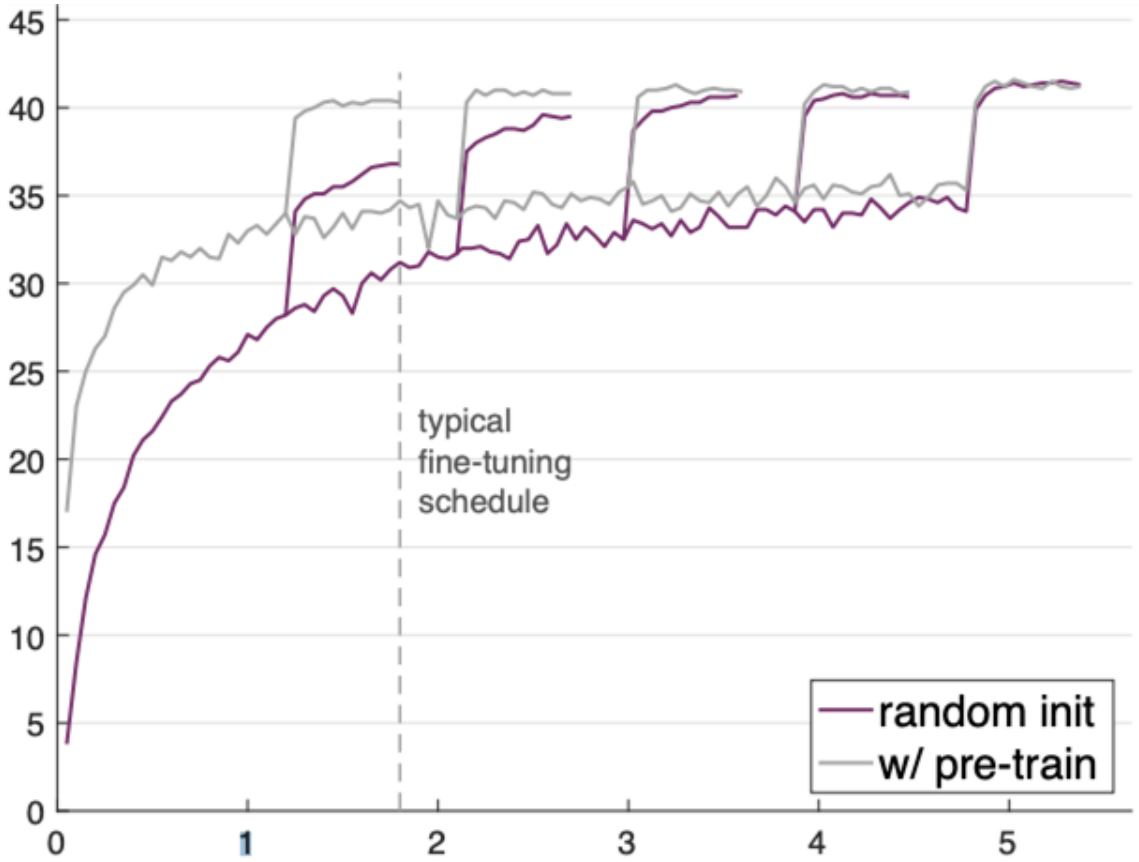


1. Train CNN on ImageNet
2. Fine-Tune (1) for object detection on Visual Genome
3. Train BERT language model on lots of text
4. Combine (2) and (3), train for joint image / language modeling
5. Fine-tune (5) for image captioning, visual question answering, etc.

Zhou et al, "Unified Vision-Language Pre-Training for Image Captioning and VQA", AAAI 2020

Transfer Learning can help you converge faster

COCO object detection



If you have enough data and train for much longer, random initialization can sometimes do as well as transfer learning

He et al, "Rethinking ImageNet Pre-Training", ICCV 2019

Recap

1. **One time setup**
   Activation functions, data preprocessing,
   weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules, large-batch training,
   hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

Next: CNN Architectures