CS163: Deep Learning for Computer Vision

Lecture 09:
Training Neural Networks

Courtesy of Prof. Justin Johnson at Umich and the team of Stanford CS231n for the slides

Announcement

- Hope everything goes well in Assignment 2
  - Bonus point for top 20% of the students
  - Daily submission to the public leaderboard: 5
- Today's lecture will be useful for improving your models in assignment 2
  - Data augmentation, model ensemble, etc.

‘Alchemy’ of Deep Learning

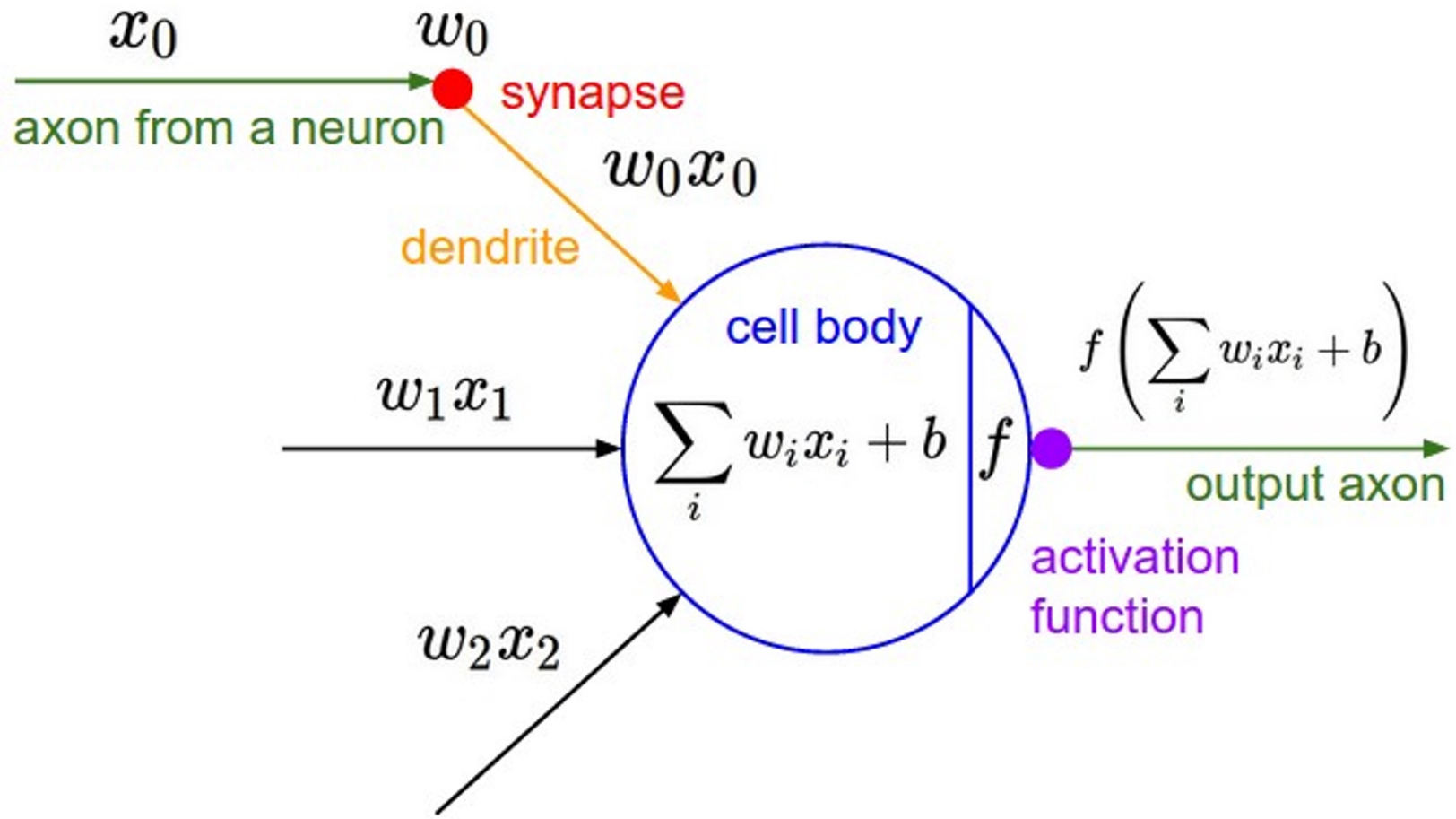
‘Chemistry’ of Deep Learning

Overview of training techniques

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

Activation Functions

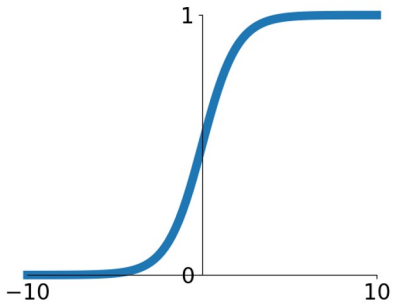
Activation Functions



Activation Functions

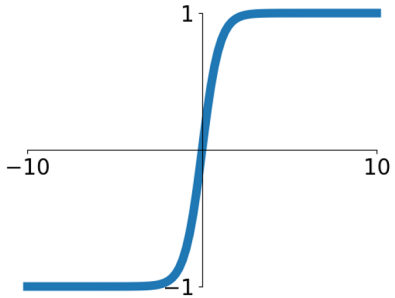
Sigmoid

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



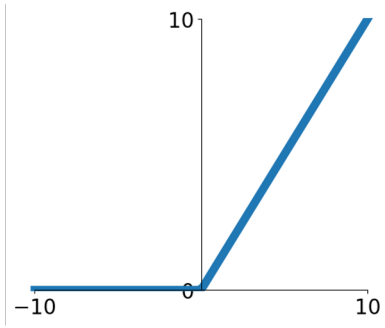
tanh

$$\tanh(x)$$



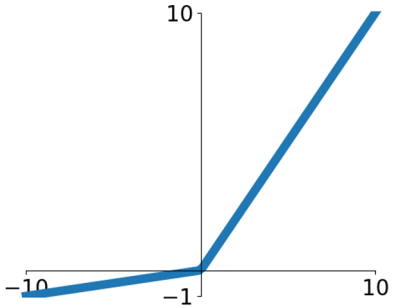
ReLU

$$\max(0, x)$$



Leaky ReLU

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

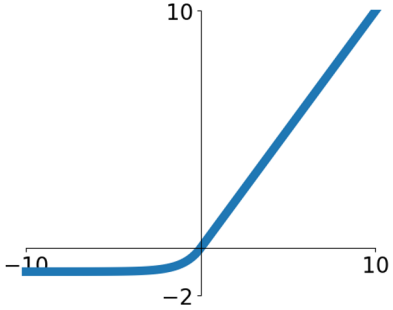


Maxout

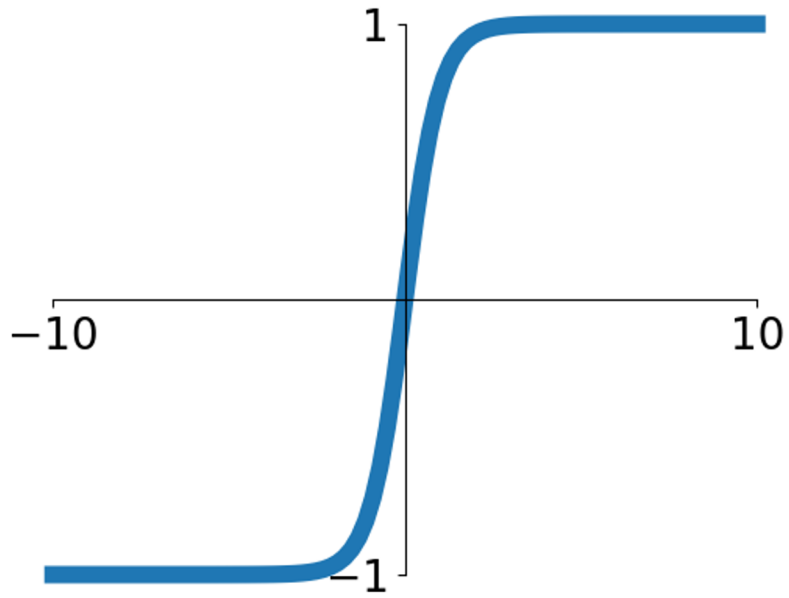
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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

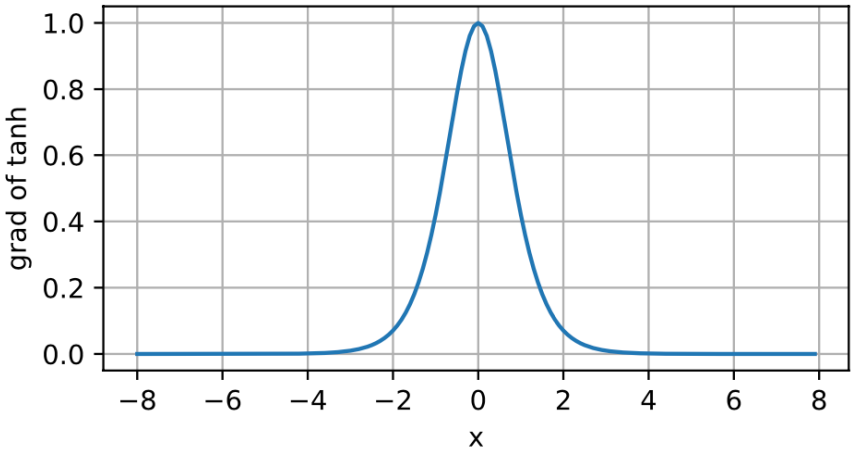


Activation Functions: Tanh



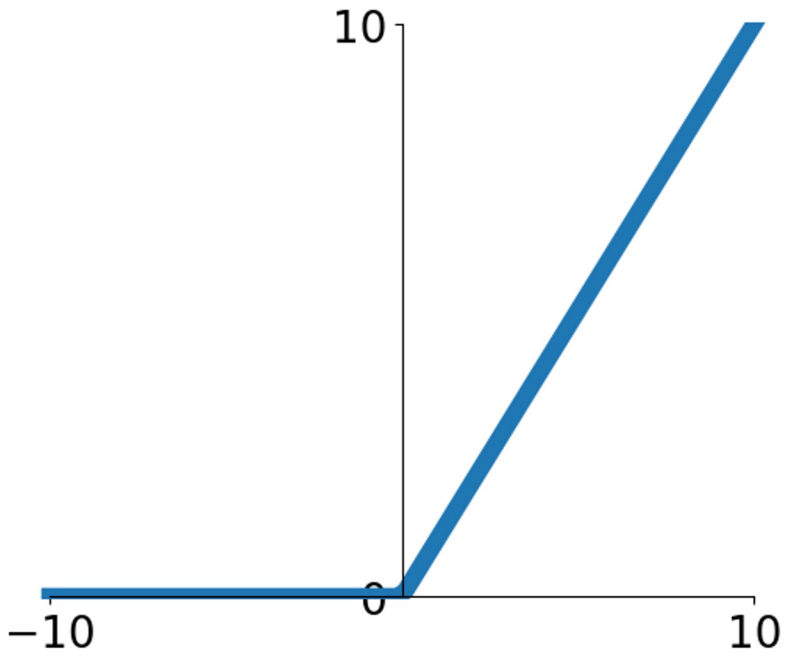
tanh(x)

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



Activation Functions: ReLU

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

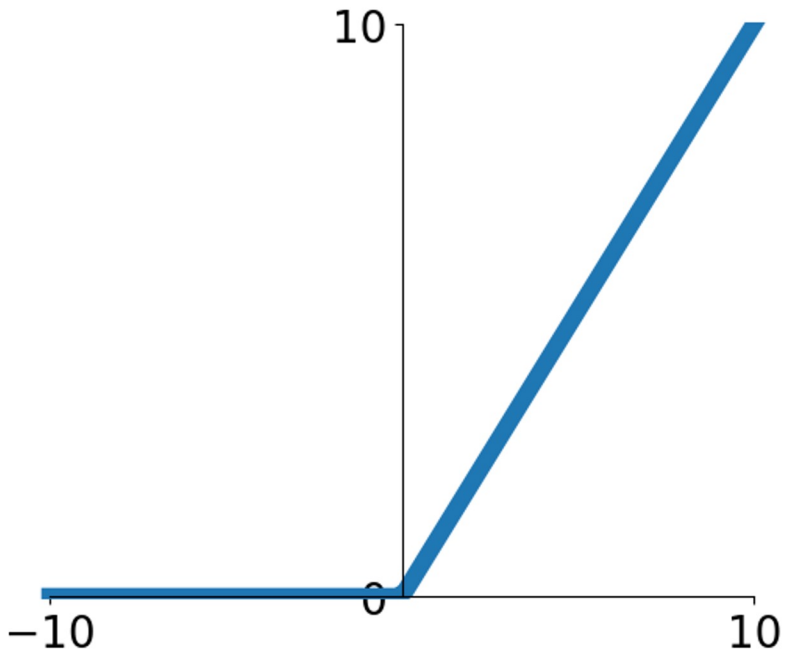


ReLU
(Rectified Linear Unit)

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

Activation Functions: ReLU

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



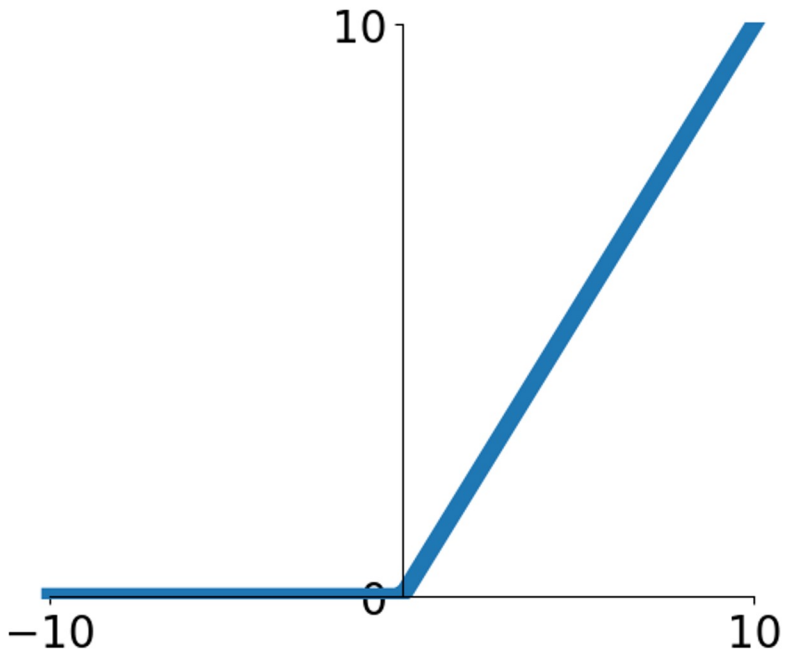
ReLU
(Rectified Linear Unit)

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

- Not zero-centered output

Activation Functions: ReLU

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



ReLU

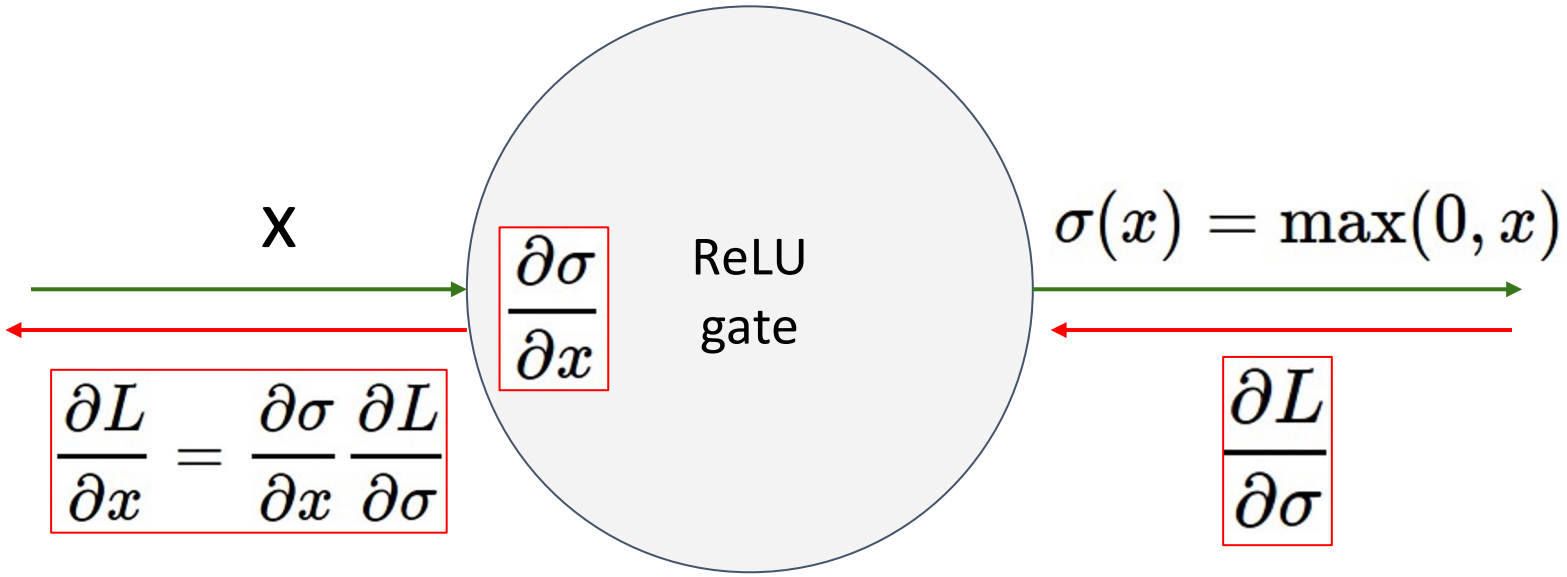
(Rectified Linear Unit)

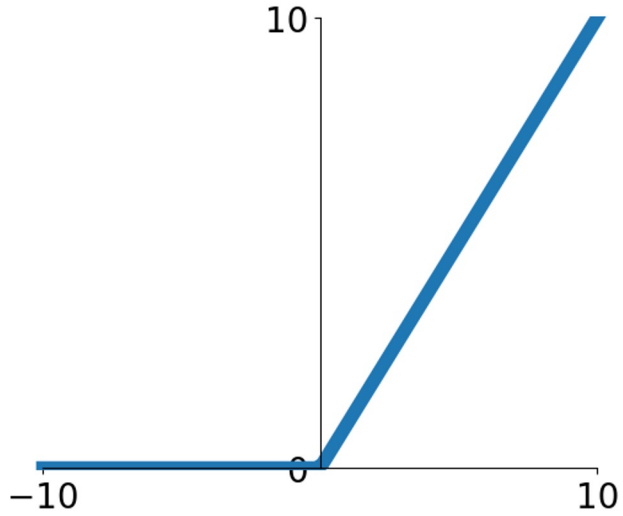
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Activation Functions: ReLU





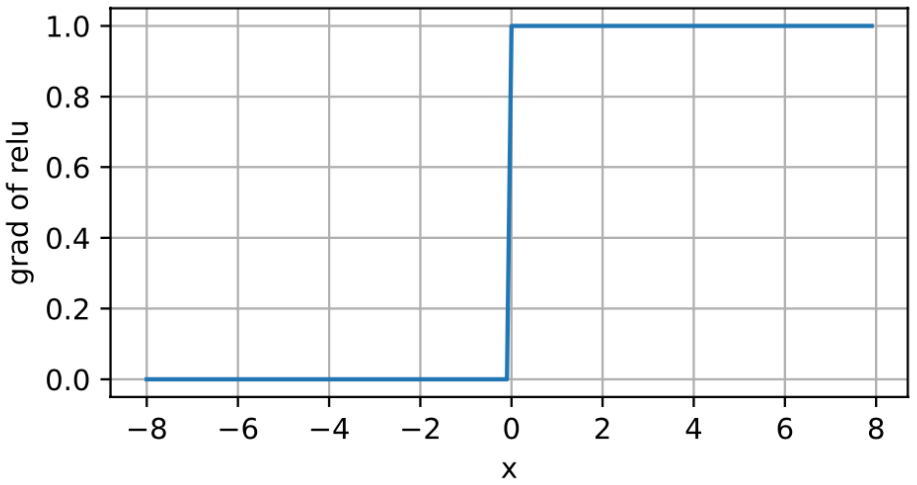
What happens when x = -8?

What happens when x = 0?

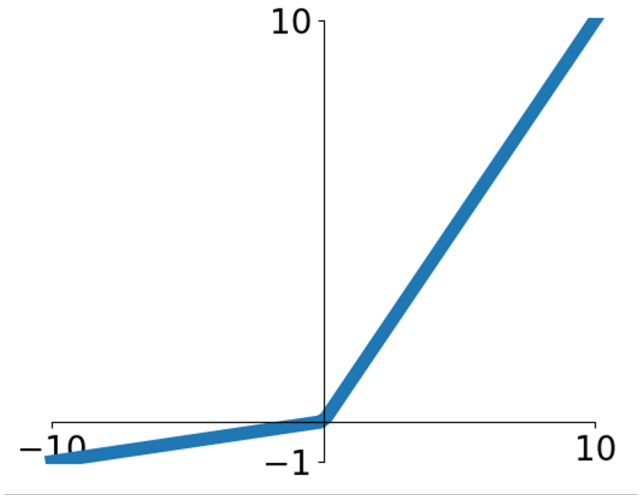
What happens when x = 8?

dead ReLU will never activate

=> never update



Activation Functions: Leaky ReLU



Leaky ReLU

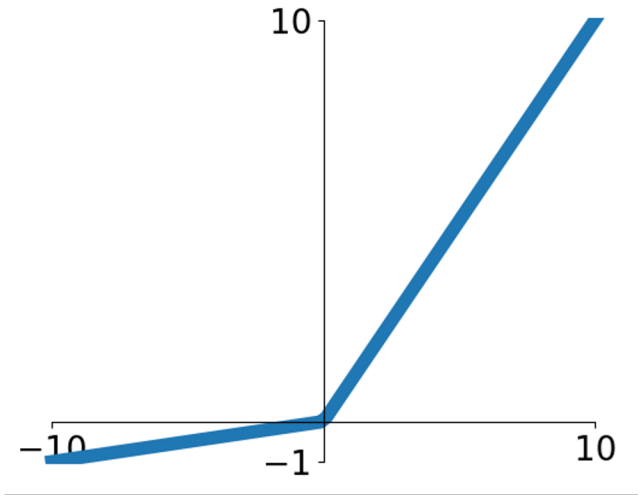
$f(x) = \max(\alpha x, x)$

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

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

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

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".**

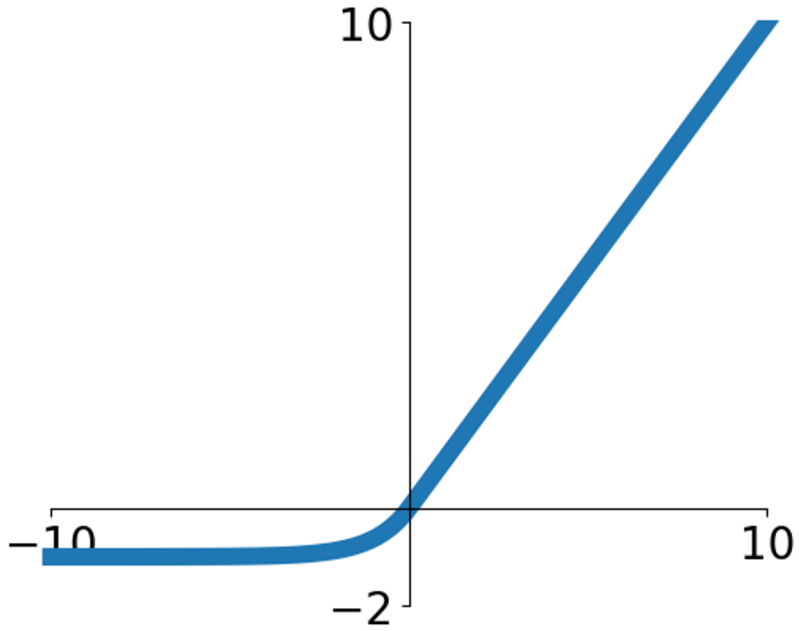
Parametric ReLU (PReLU)

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

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)



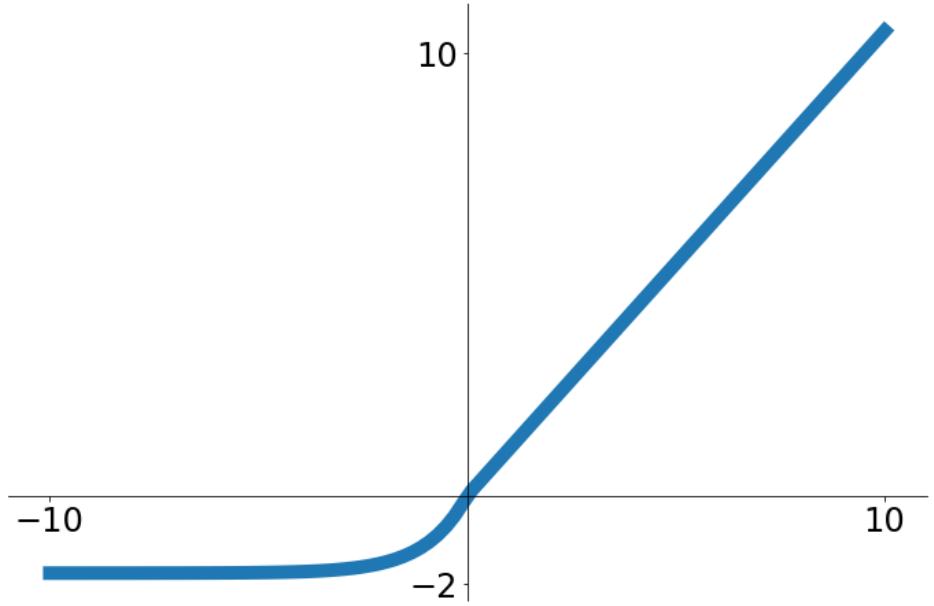
$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & if\ x \leq 0 \end{cases}$$

(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)



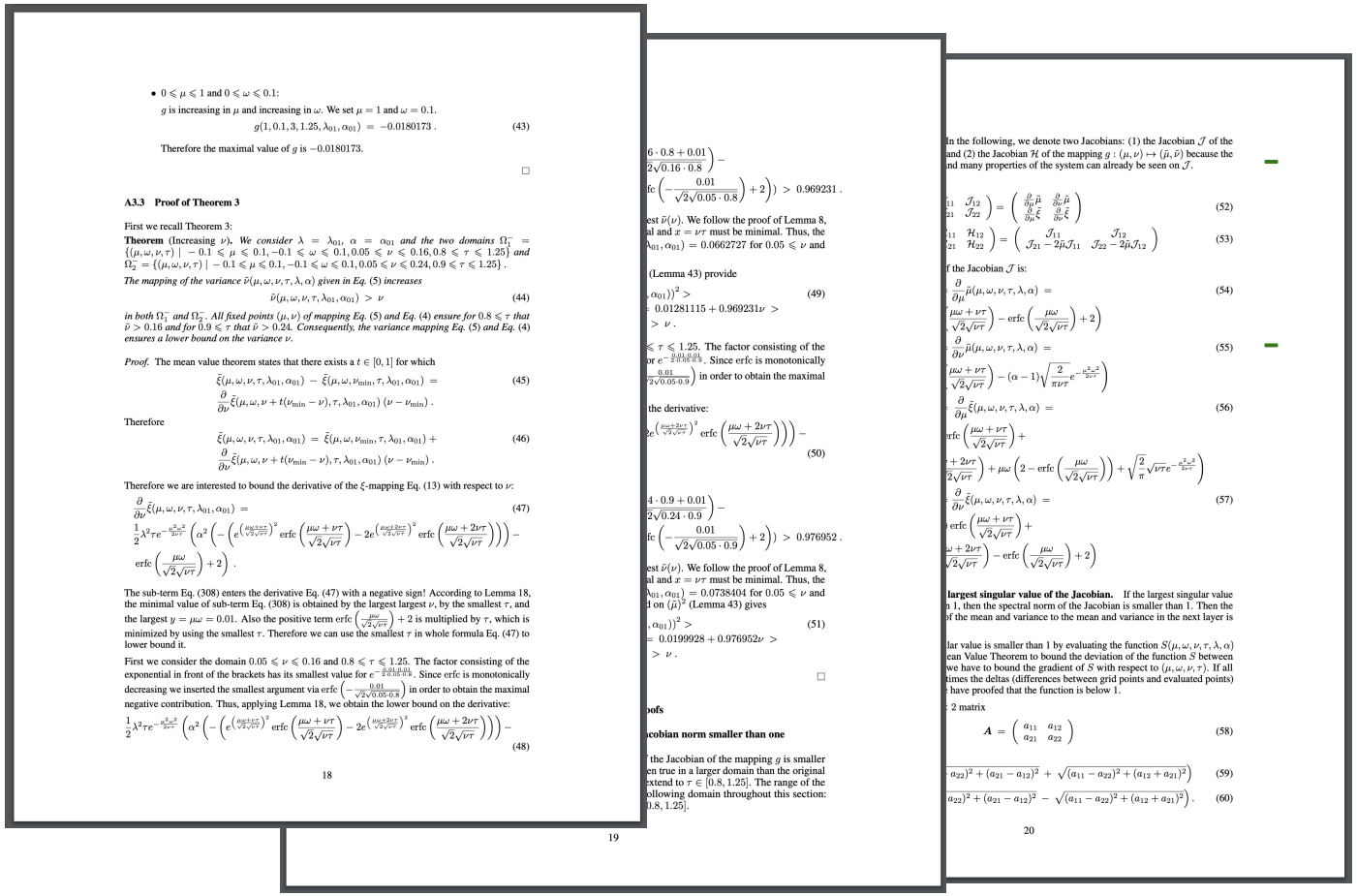
- Scaled version of ELU that works better for deep networks
- "Self-Normalizing" property; can train deep SELU networks without BatchNorm

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

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

Klambauer et al, Self-Normalizing Neural Networks, NeurIPS 2017
https://proceedings.neurips.cc/paper/2017/file/5d44ee6f2c3f71b73125876103c8f6c4-Paper.pdf

Activation Functions: Scaled Exponential Linear Unit (SELU)



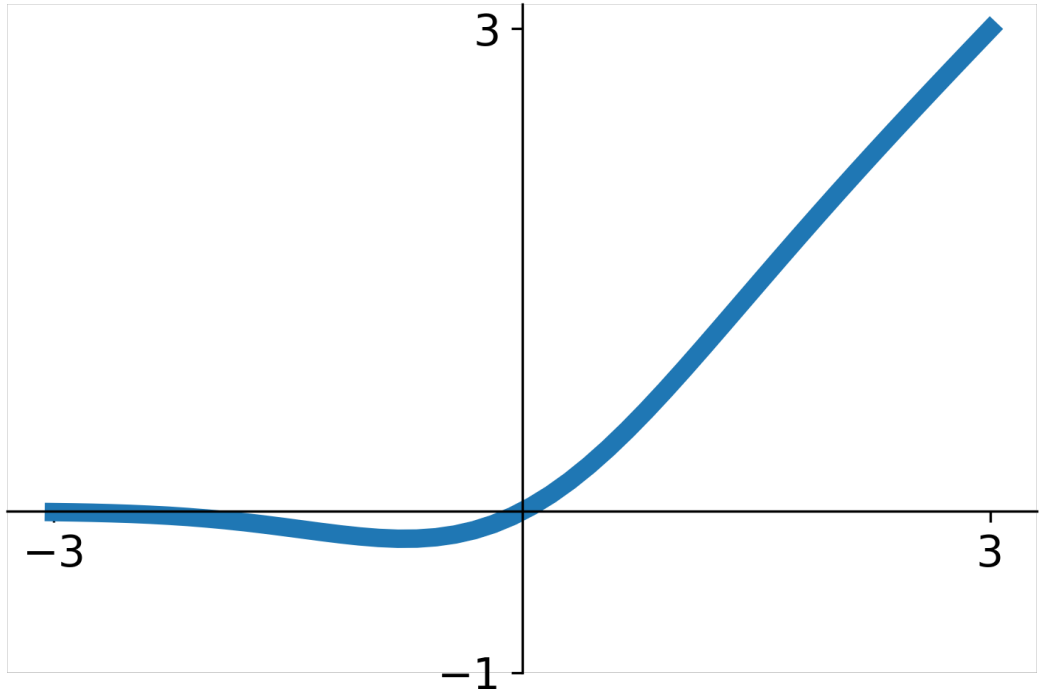
- Scaled version of ELU that works better for deep networks
- "Self-Normalizing" property; can train deep SELU networks without BatchNorm

Derivation takes
91 pages of math
in appendix…

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

Klambauer et al, Self-Normalizing Neural Networks, NeurIPS 2017
https://proceedings.neurips.cc/paper/2017/file/5d44ee6f2c3f71b73125876103c8f6c4-Paper.pdf

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, GPT-2, GPT-3)

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

Activation Functions: Gaussian Error Linear Unit (GELU)

https://github.com/karpathy/nanoGPT/blob/master/model.py

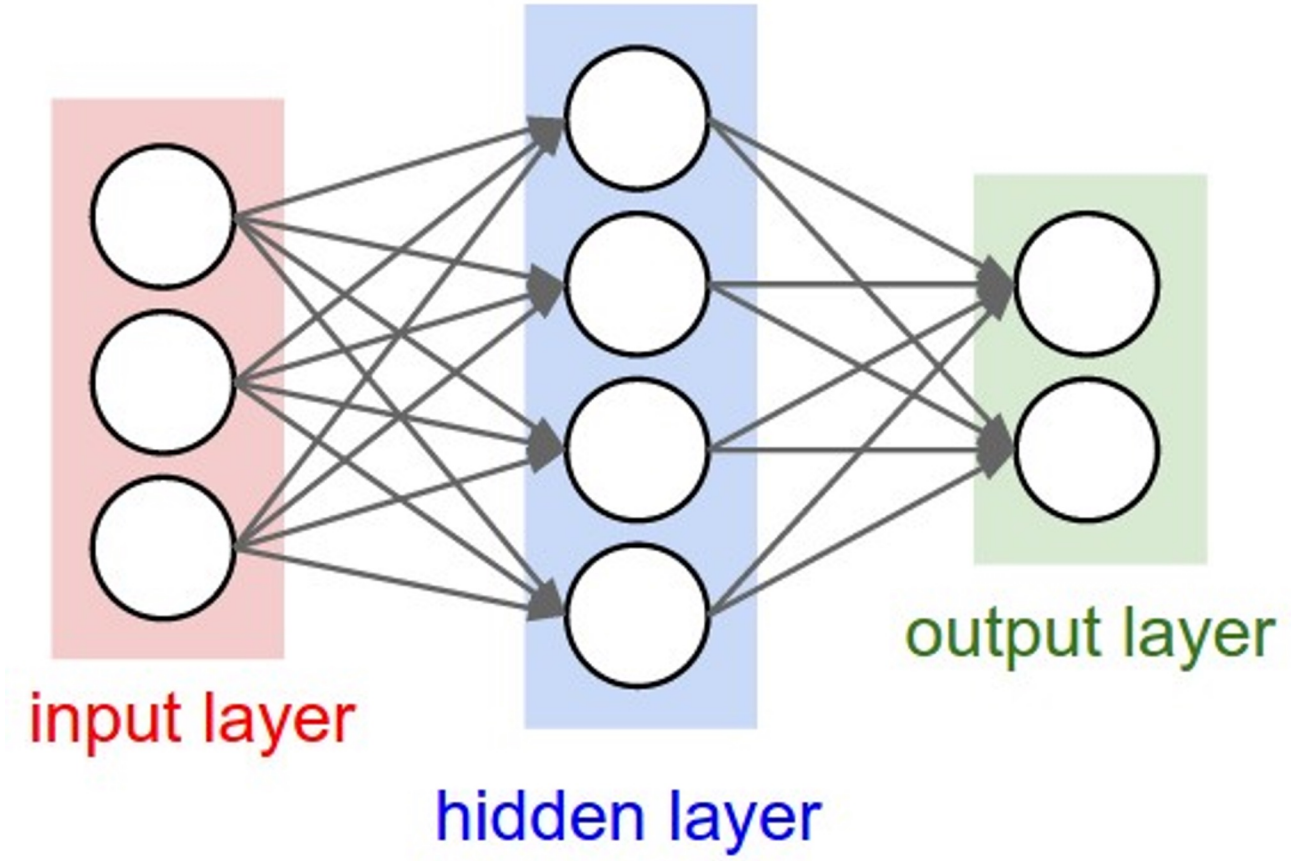
```
# @torch.jit.script # good to enable when not using torch.compile, disable when using (our default)
def new_gelu(x):
    """
    Implementation of the GELU activation function currently in Google BERT repo (identical to OpenAI GPT).
    Reference: Gaussian Error Linear Units (GELU) paper: https://arxiv.org/abs/1606.08415
    """
    return 0.5 * x * (1.0 + torch.tanh(math.sqrt(2.0 / math.pi) * (x + 0.044715 * torch.pow(x, 3.0))))
```

Activation Functions: Summary

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

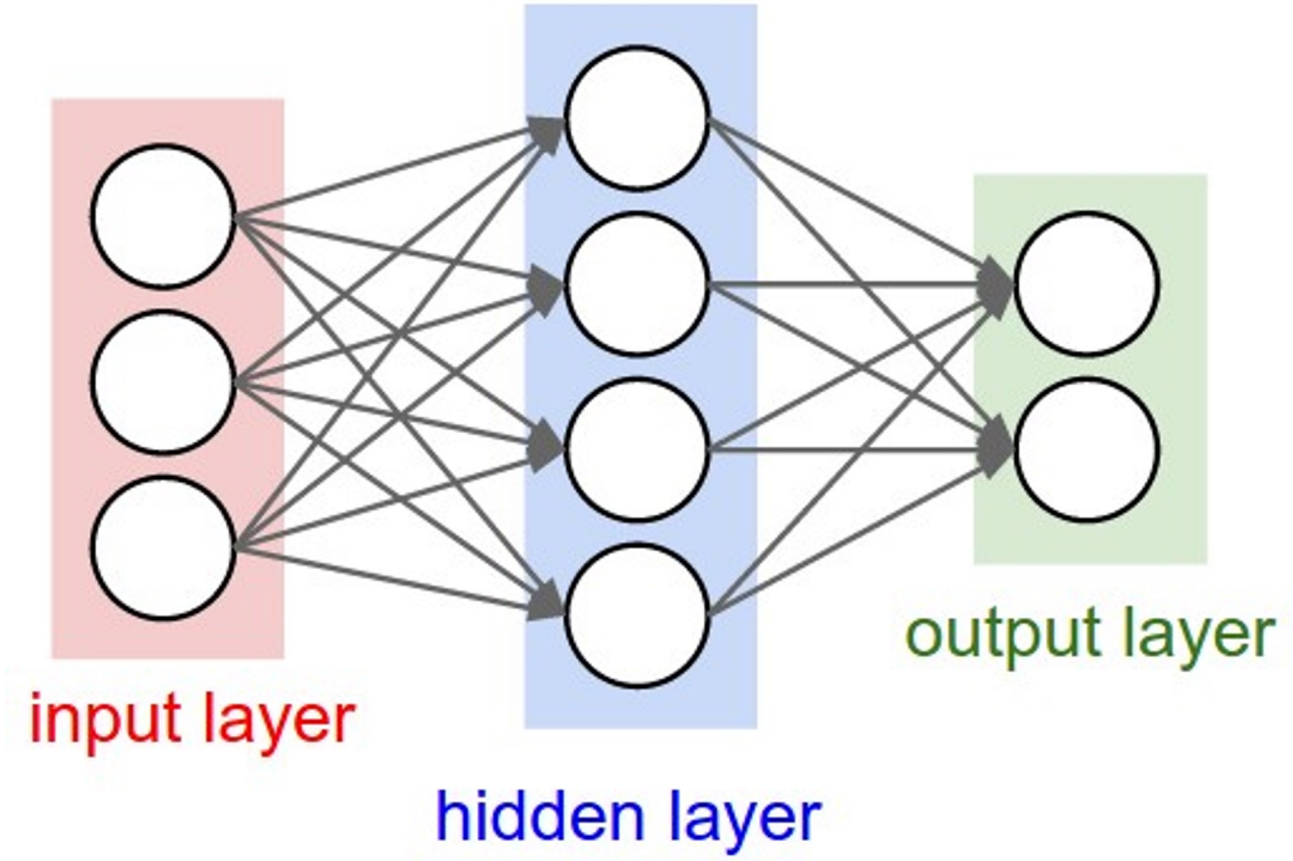
Weight Initialization

Weight Initialization



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

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)
```

Weight Initialization

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

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

Works ~okay for small networks, but problems 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

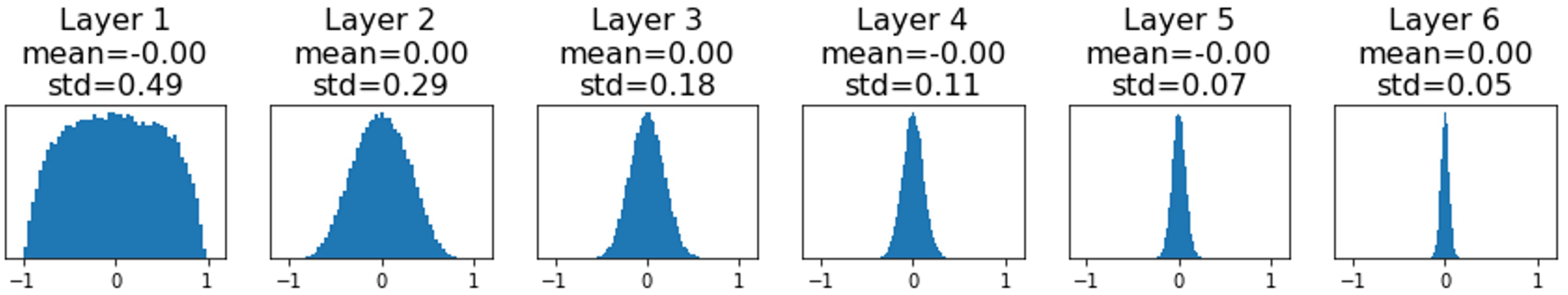
Weight Initialization: Activation Statistics

```
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?



Weight Initialization: Activation Statistics

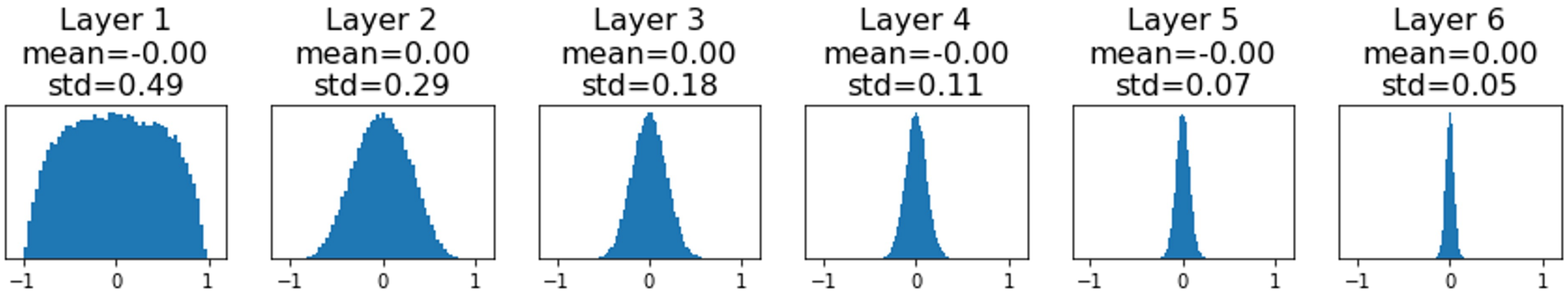
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

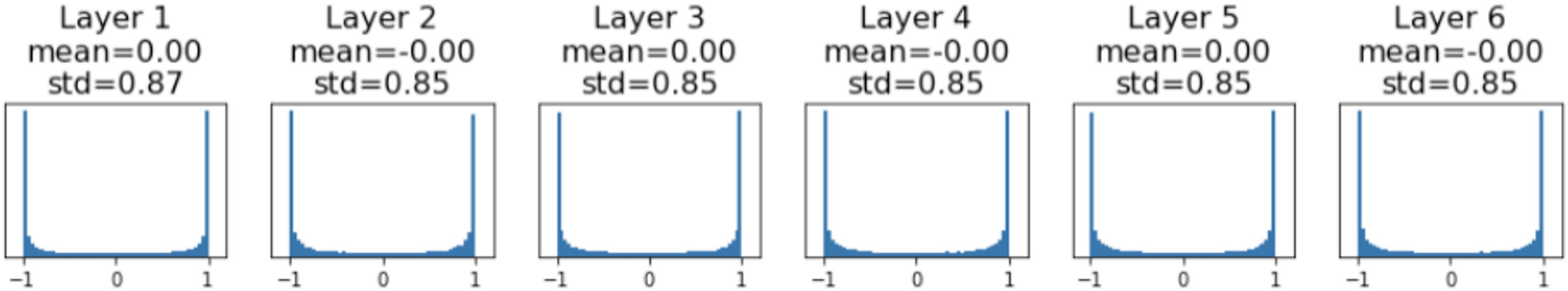
Weight Initialization: Activation Statistics

```
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?



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)
```

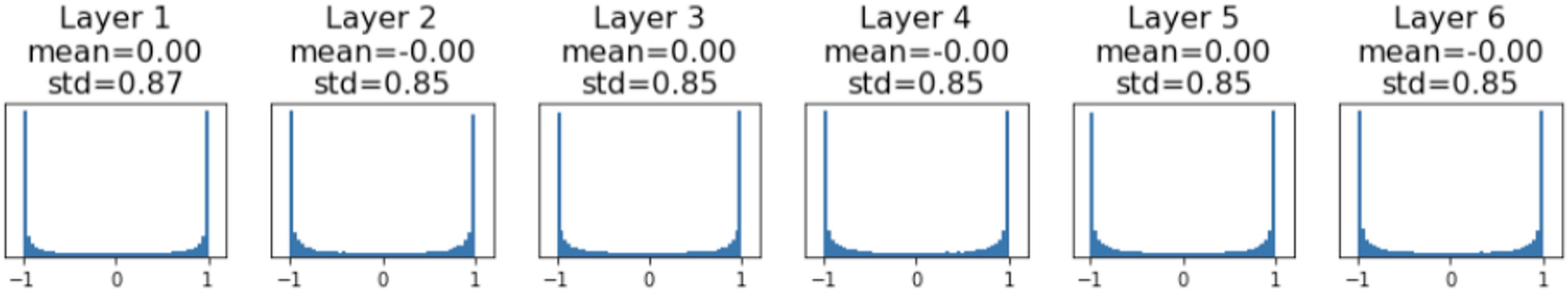
Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$



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)

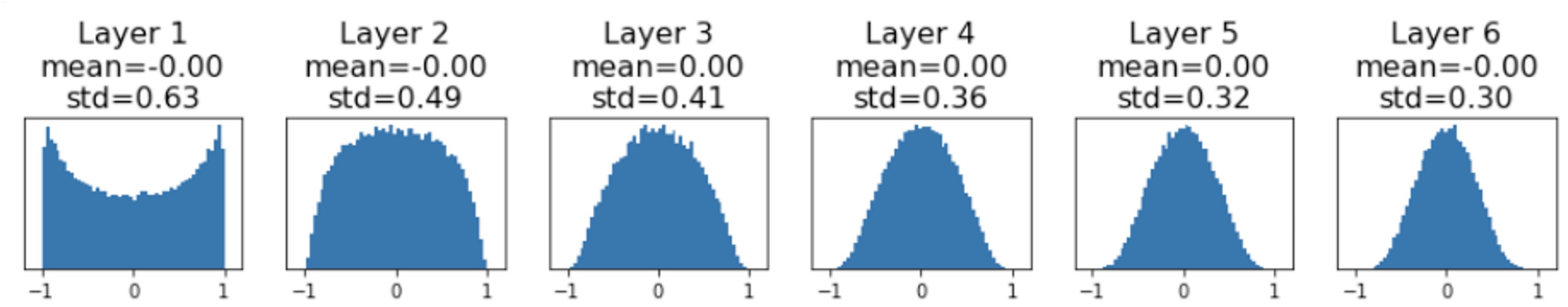
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

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!



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

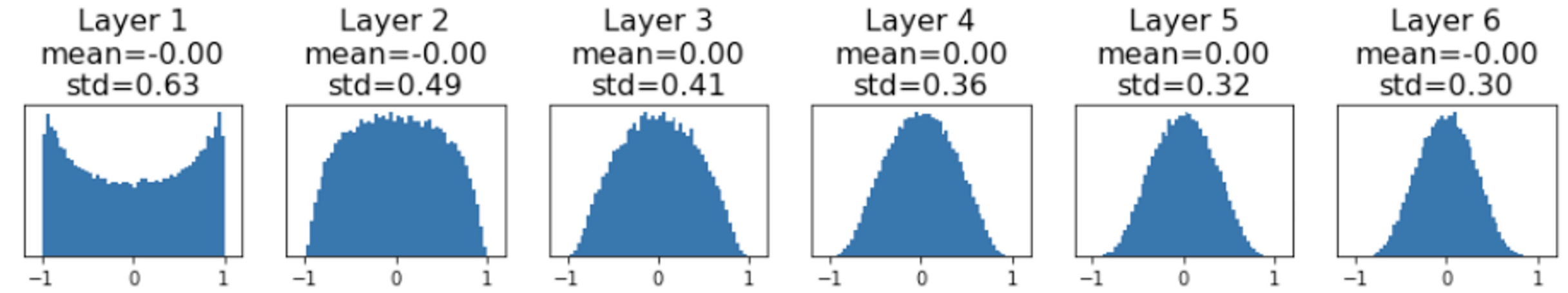
Weight Initialization: Xavier Initialization

```
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

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

Weight Initialization: Xavier Initialization

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

Derivation: Variance of output = Variance of input

y = Wx

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

$Var(y_i) = Din * Var(x_i w_i)$

[Assume x, w are iid]

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]

Weight Initialization: Xavier Initialization

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

Derivation: Variance of output = Variance of input

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

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

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

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

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

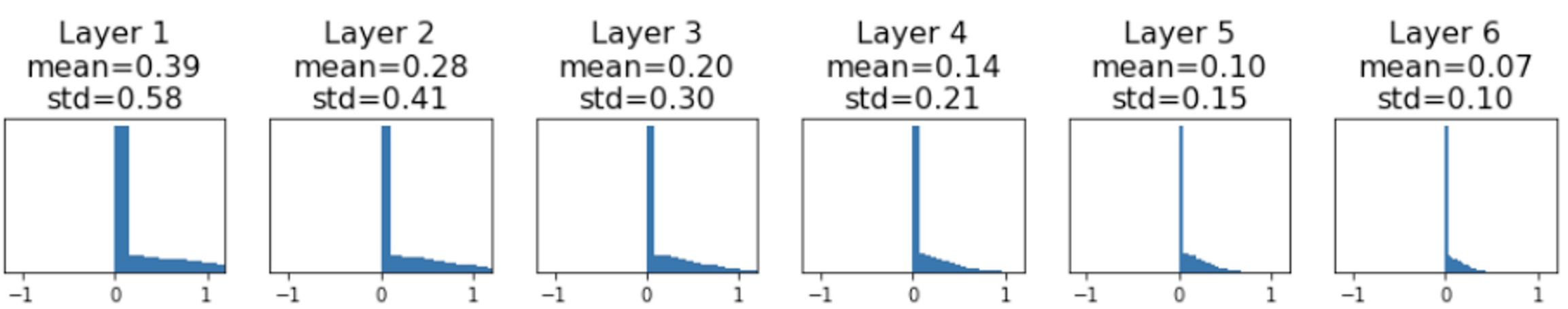
Weight Initialization: What about ReLU?

```
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 =(

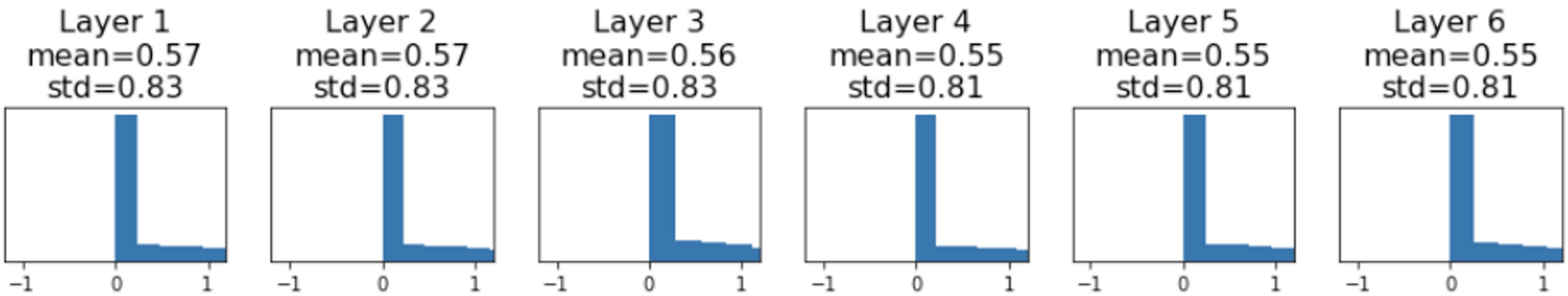


Weight Initialization: Kaiming Initialization

```
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.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

”Just right” – activations nicely scaled for all layers



See this paper for detailed derivation!

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

Weight Initialization: Kaiming Initialization

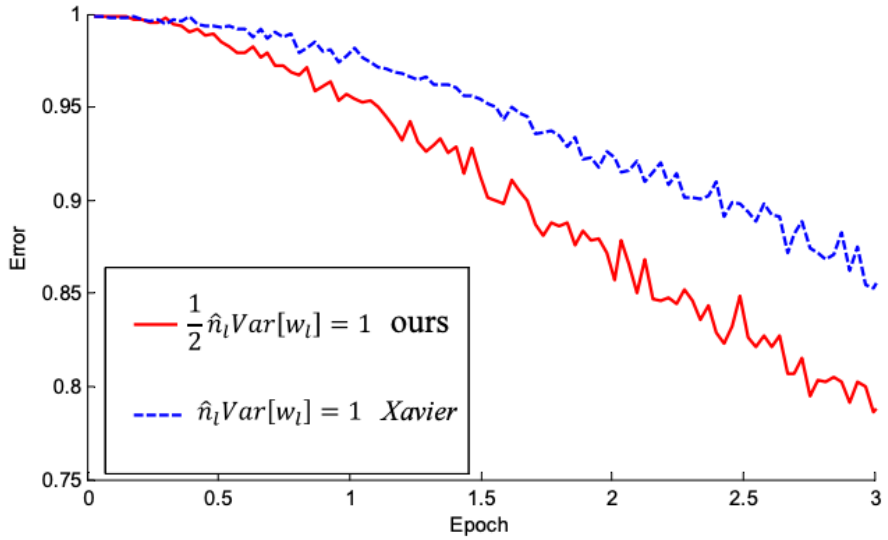


Figure 2. The convergence of a **22-layer** large model (B in Table 3). The x-axis is the number of training epochs. The y-axis is the top-1 error of 3,000 random val samples, evaluated on the center crop. We use ReLU as the activation for both cases. Both our initialization (red) and "*Xavier*" (blue) [7] lead to convergence, but ours starts reducing error earlier.

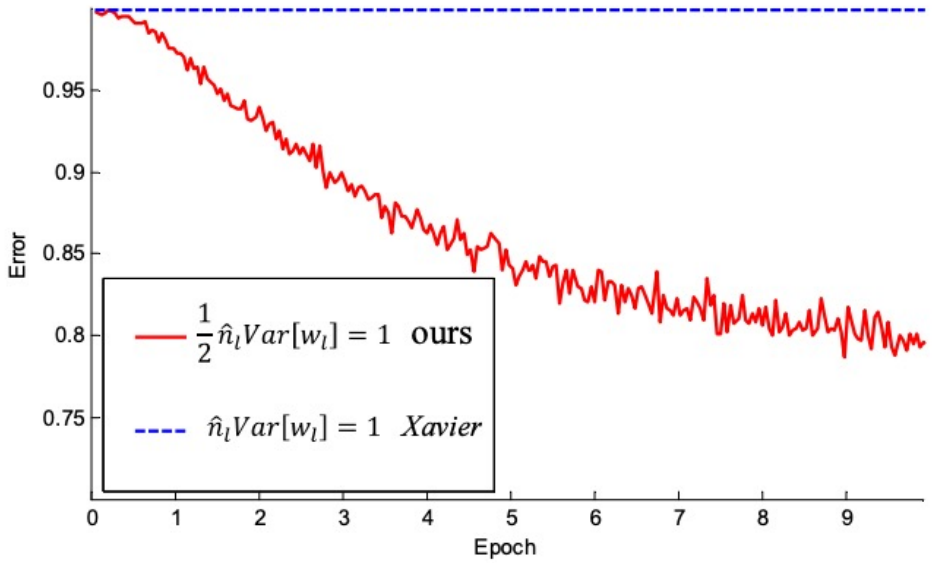
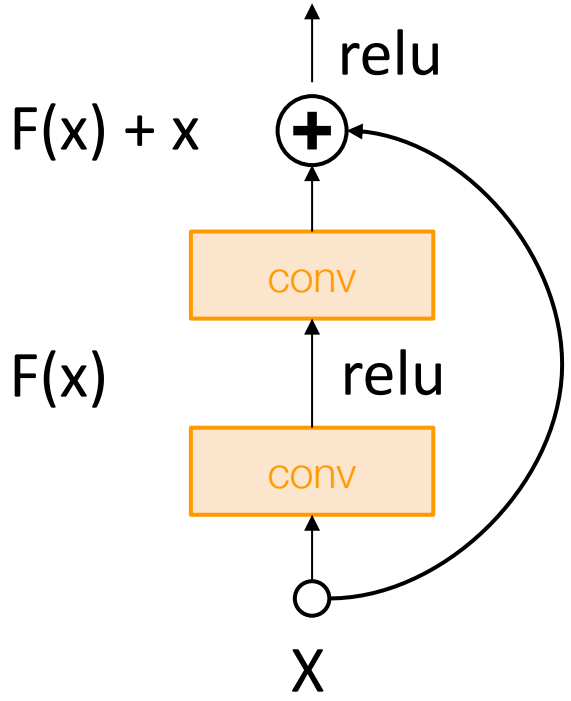


Figure 3. The convergence of a **30-layer** small model (see the main text). We use ReLU as the activation for both cases. Our initialization (red) is able to make it converge. But "*Xavier*" (blue) [7] completely stalls - we also verify that its gradients are all diminishing. It does not converge even given more epochs.

See this paper for detailed derivation!

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

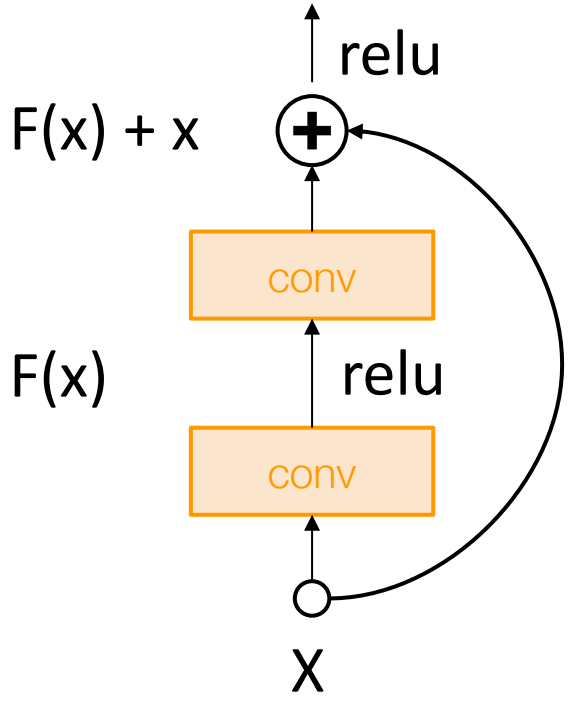
Weight Initialization: Residual Networks



Residual Block

If we initialize with Kaiming method :
then Var(F(x)) = Var(x)
But then Var(F(x) + x) > Var(x) variance grows with each block!

Weight Initialization: Residual Networks



Residual Block

If we initialize with Kaiming method :
then Var(F(x)) = Var(x)
But then Var(F(x) + x) > Var(x) variance grows with each block!

Solution: Initialize first conv with Kaiming, initialize second conv to zero. Then Var(x + F(x)) = Var(x)

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

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

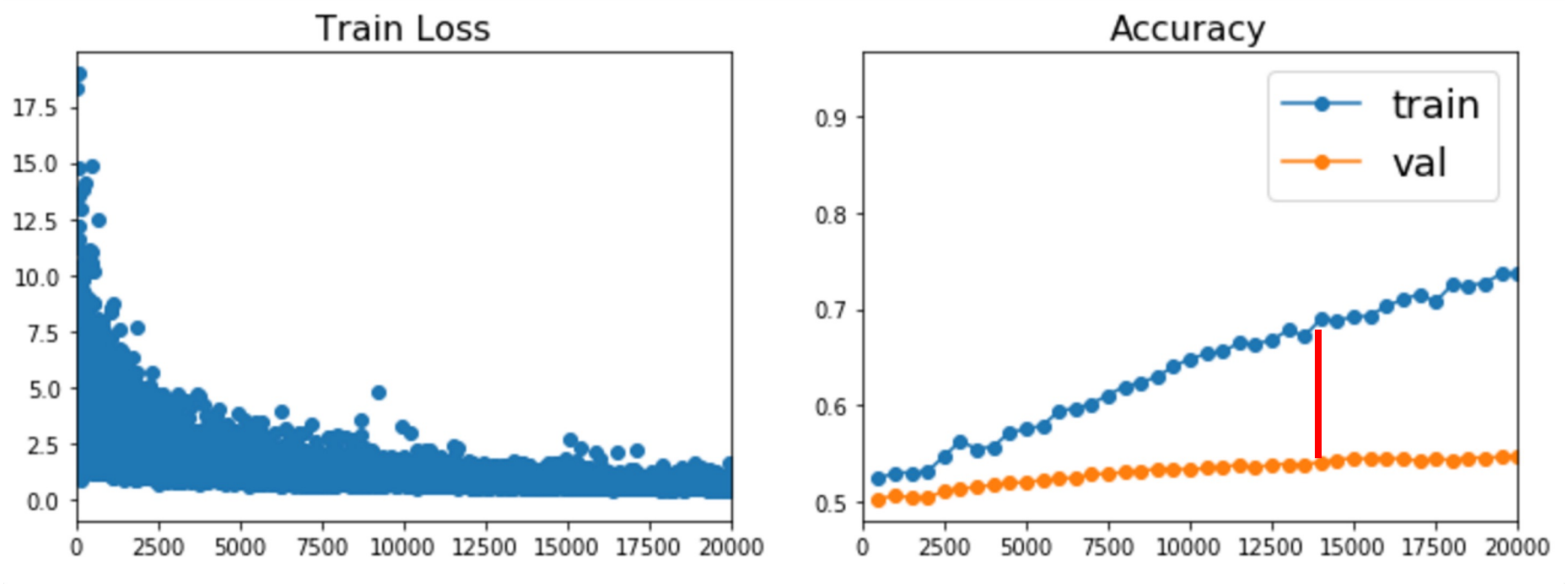
Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

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: Add term to the loss

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

In common use:

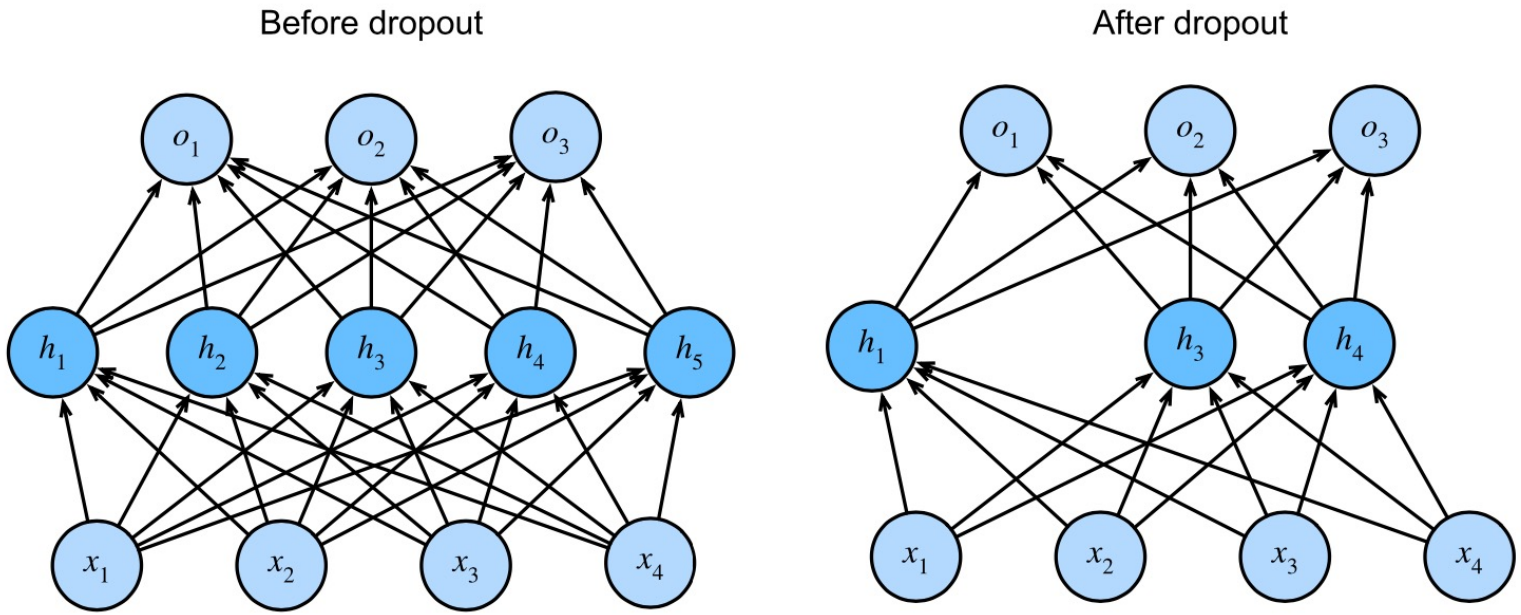
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

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



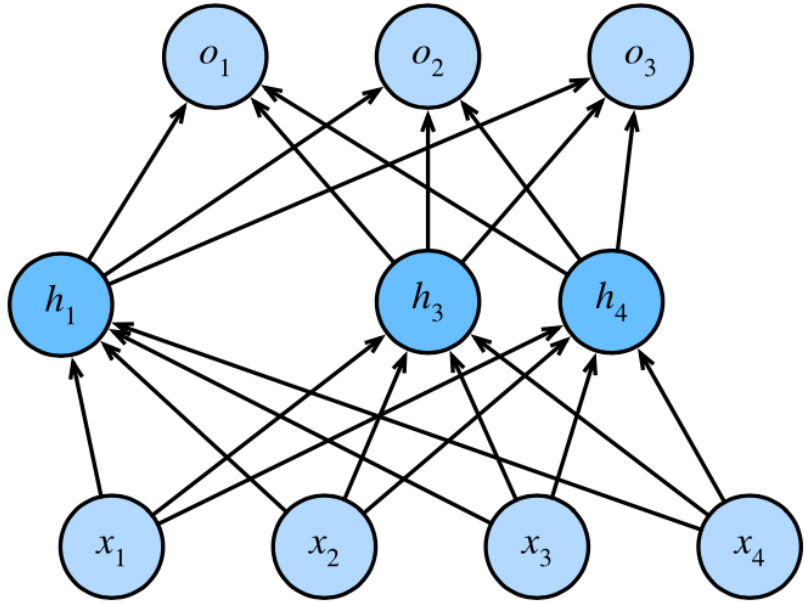
Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: Dropout

Randomly set weight as zero

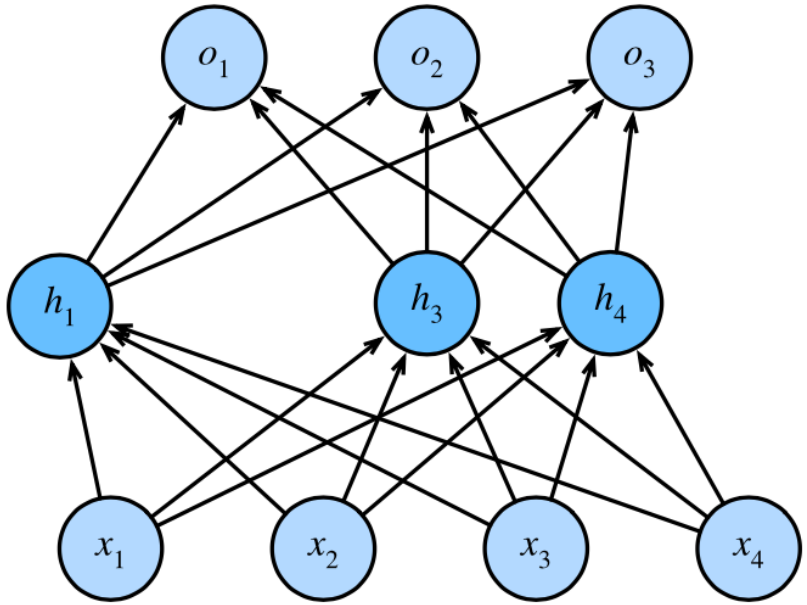
```
import torch
from torch import nn
from d2l import torch as d2l

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return torch.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

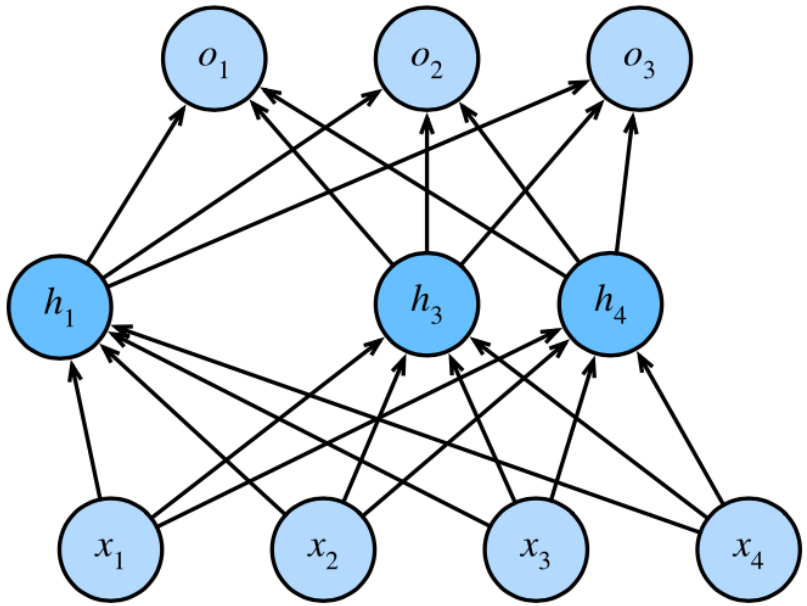


Regularization: Dropout

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 mask is one model

An FC layer with 4096 units has
$2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ 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

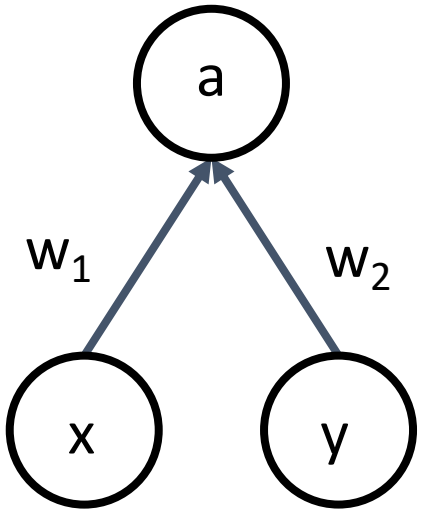
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

But this integral seems hard …

Dropout: Test Time

Want to approximate the integral

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



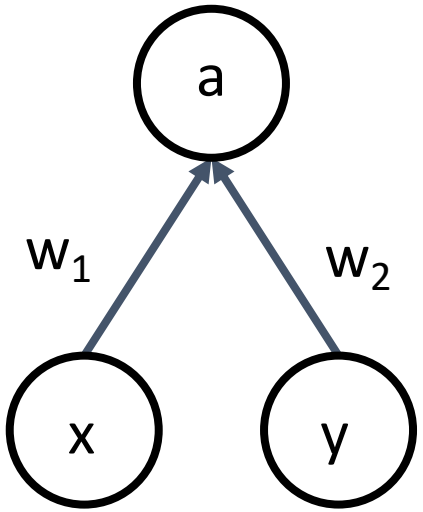
Consider a single neuron:

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

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: $E[a] = w_1 x + w_2 y$

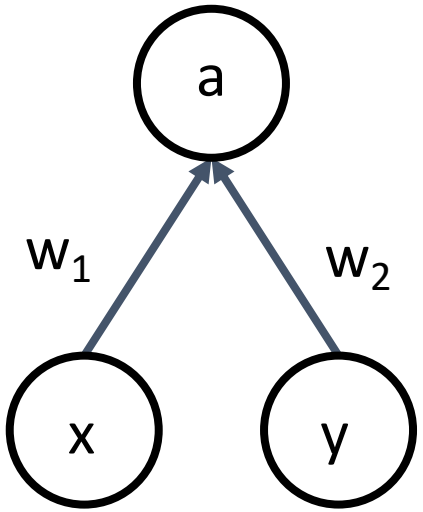
During training we have:

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

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:

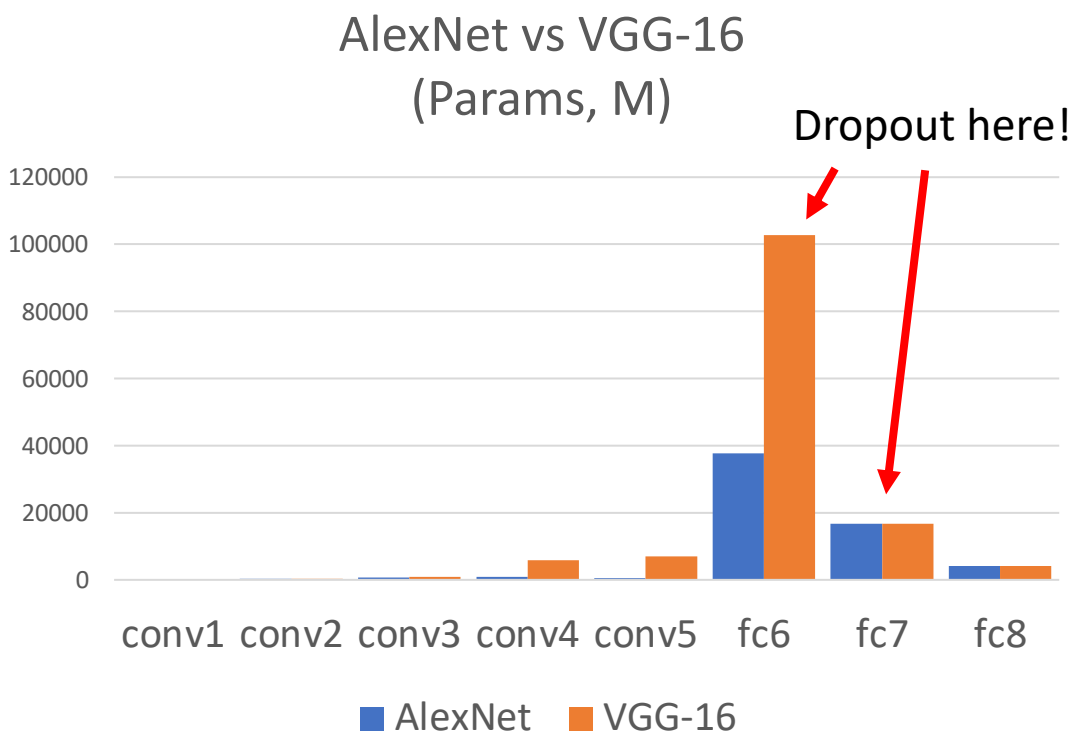
At test time we have: $E[a] = w_1 x + w_2 y$

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 in AlexNet and VGG

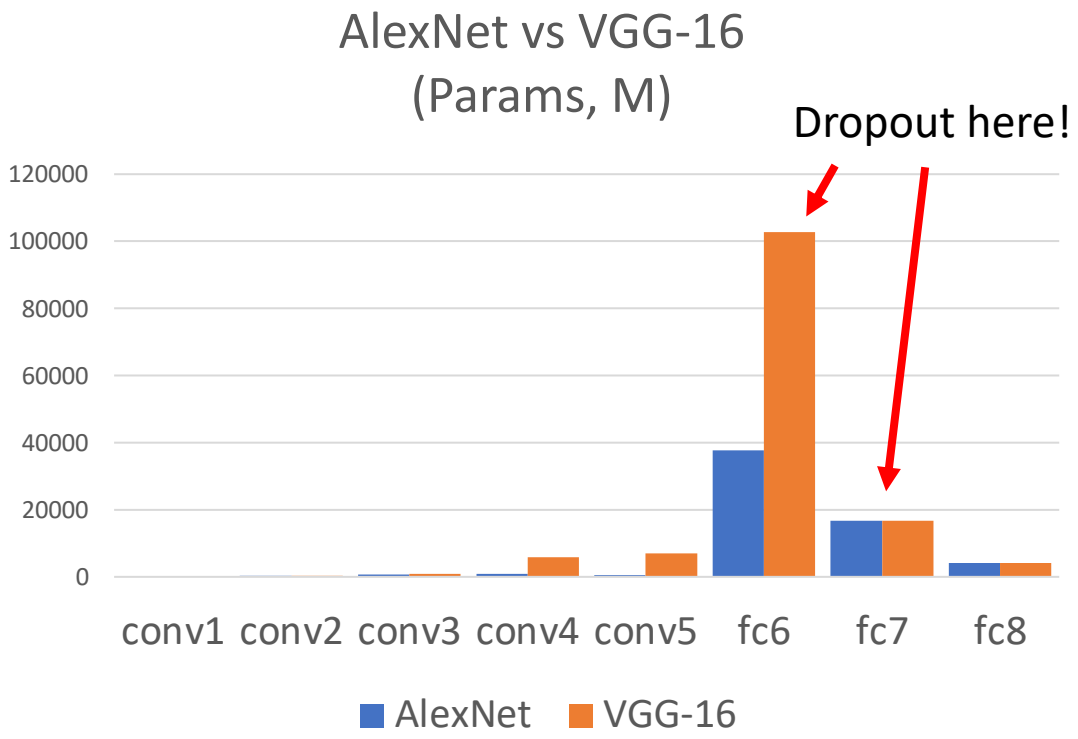
Recall AlexNet, VGG have most of their parameters in **fully-connected layers**; usually Dropout is applied there




```python
class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000, dropout: float = 0.5) -> None:
        super().__init__()
        _log_api_usage_once(self)
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )
```

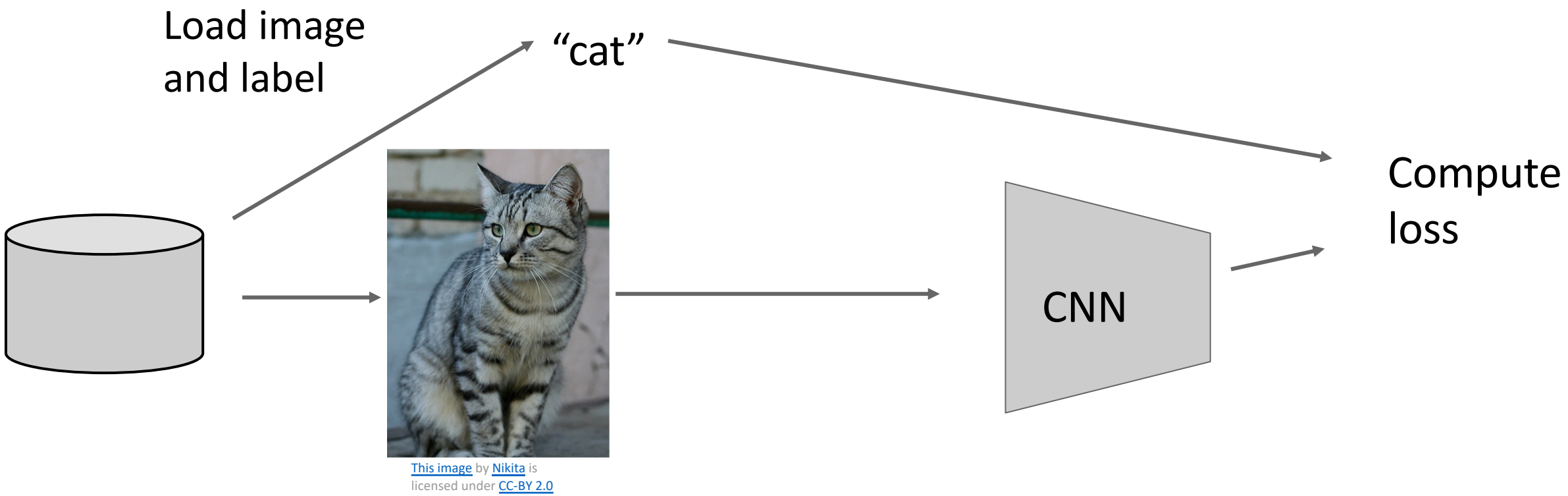
Dropout architectures

Recall AlexNet, VGG have most of their parameters in **fully-connected layers**; usually Dropout is applied there

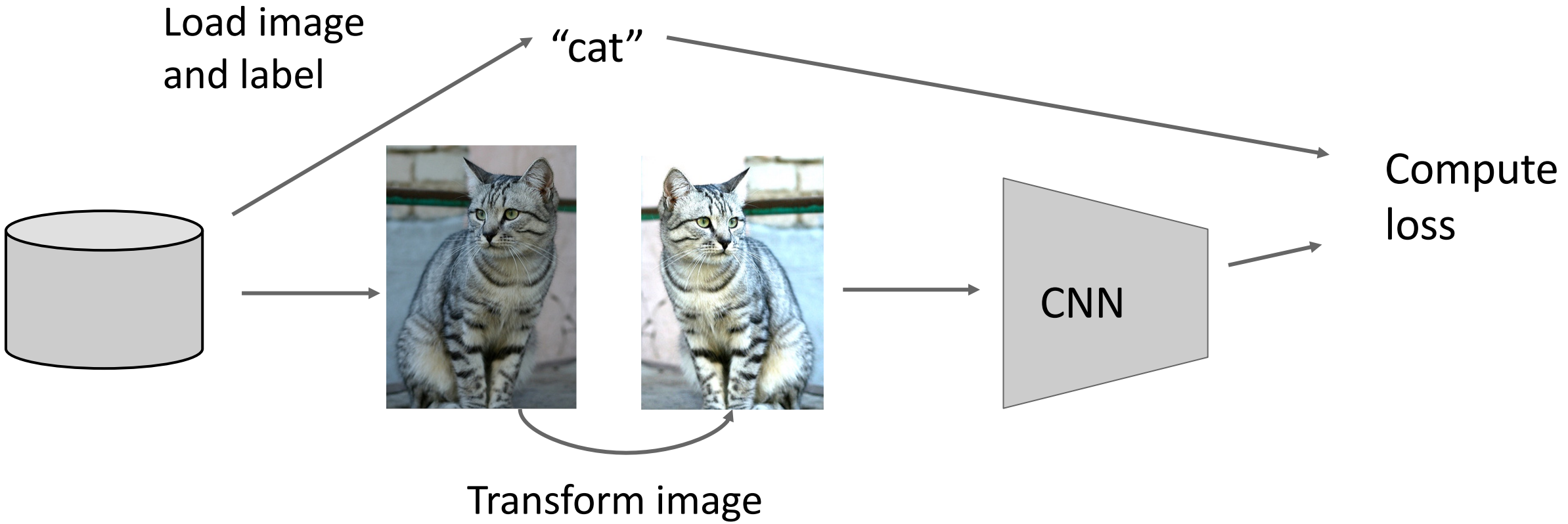


Later architectures (GoogLeNet, ResNet, etc) use global average pooling instead of fully-connected layers: they don't use dropout at all!

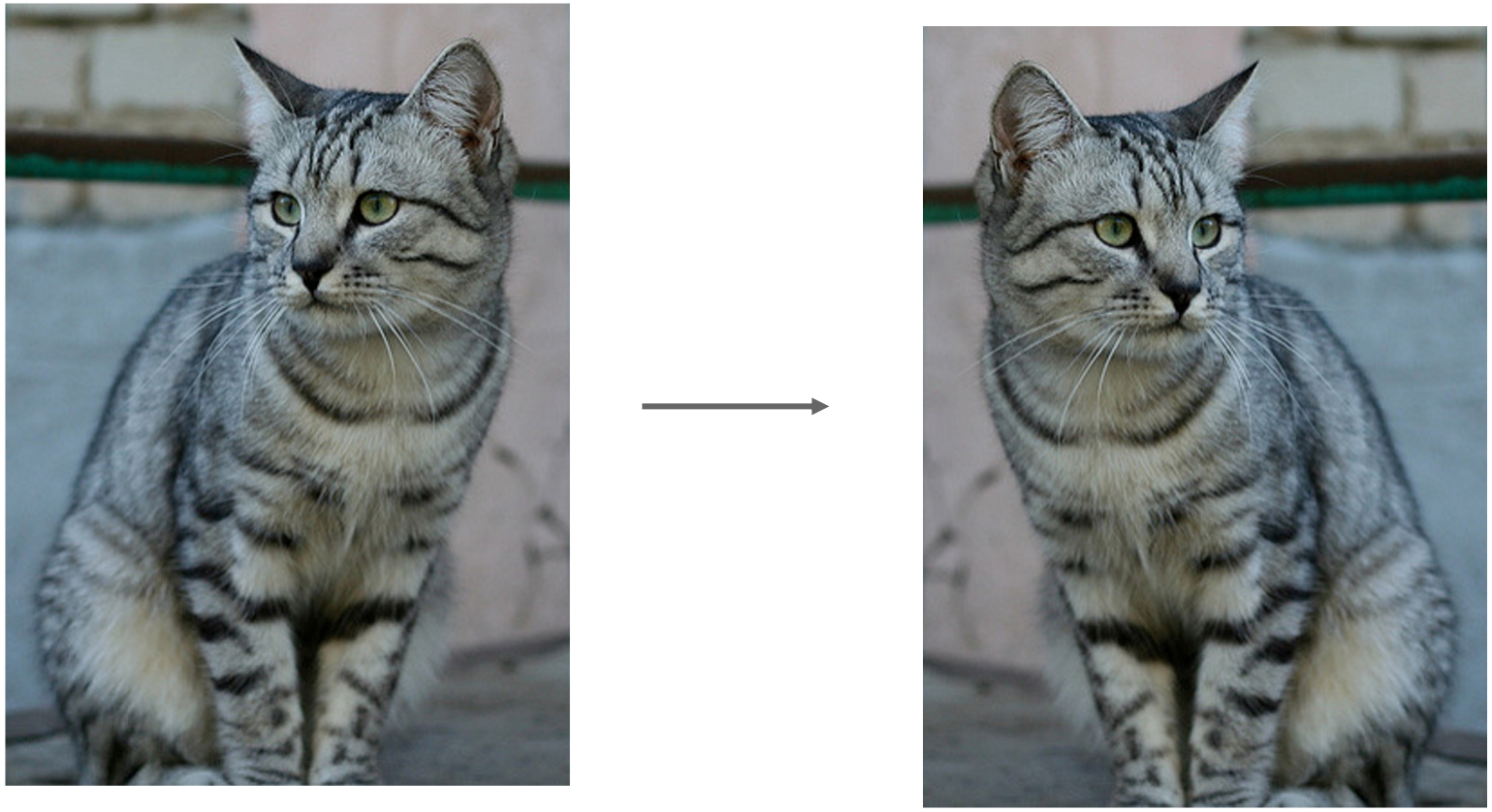
Data Augmentation



Data Augmentation



Data Augmentation: Horizontal Flips

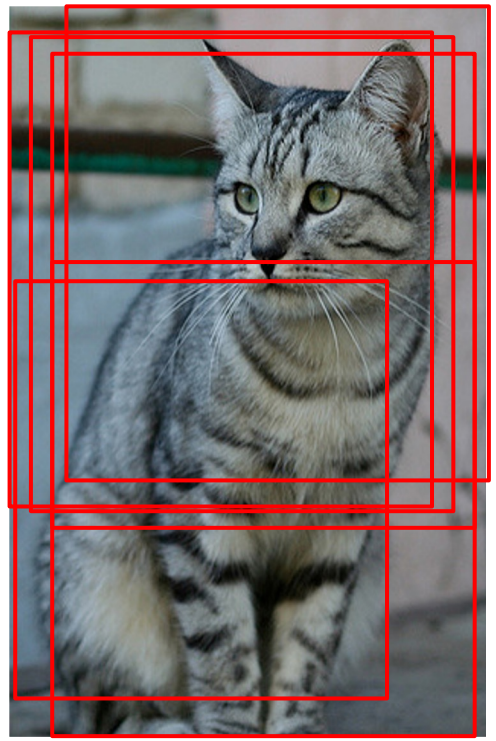


Data Augmentation: Random Crops and Scales

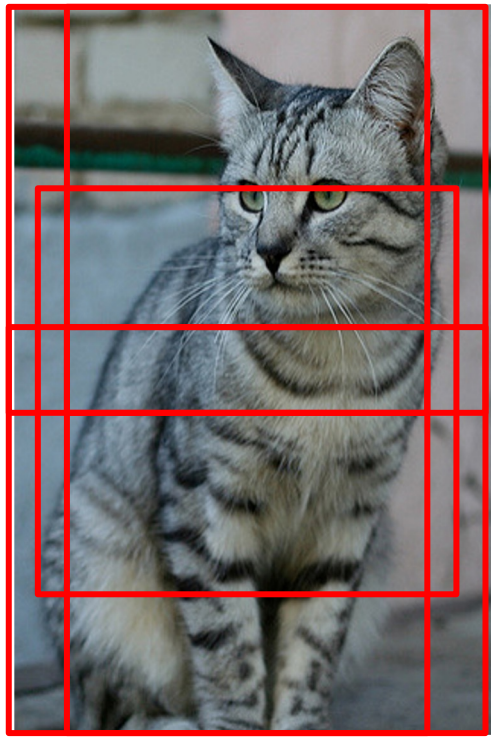
Training: sample random crops / scales

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 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: Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions,

… (go crazy)

Data Augmentation: Implementation in PyTorch

Training

```
transform_train = torchvision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height-to-width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new 224 x 224 image
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0 / 4.0, 4.0 / 3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # Randomly change the brightness, contrast, and saturation
    torchvision.transforms.ColorJitter(brightness=0.4, contrast=0.4,
                                       saturation=0.4),
    # Add random noise
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

Testing

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # Crop a 224 x 224 square area from the center of the image
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

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: Stochastic Depth

Training: Skip some residual blocks in ResNet

Testing: Use the whole network

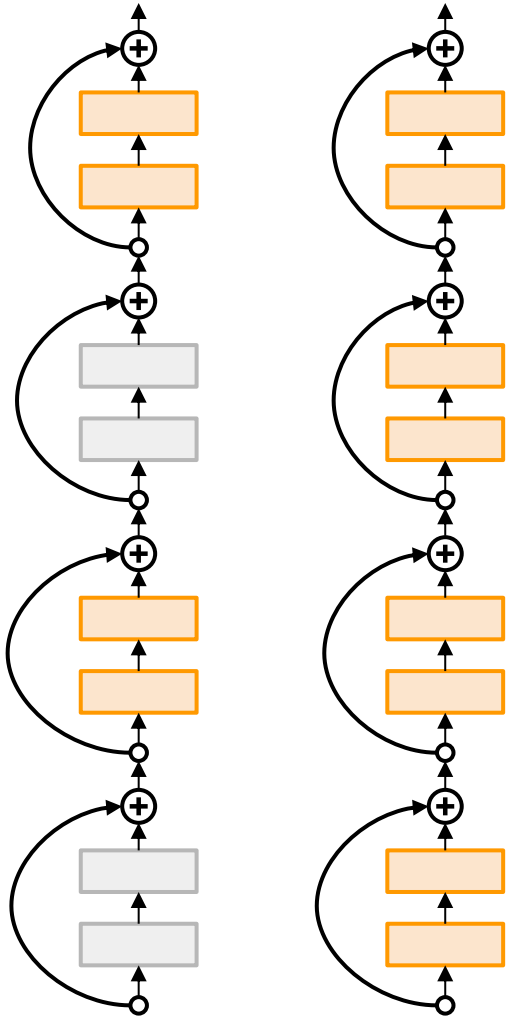
Examples:

Dropout

Batch Normalization

Data Augmentation

Stochastic Depth

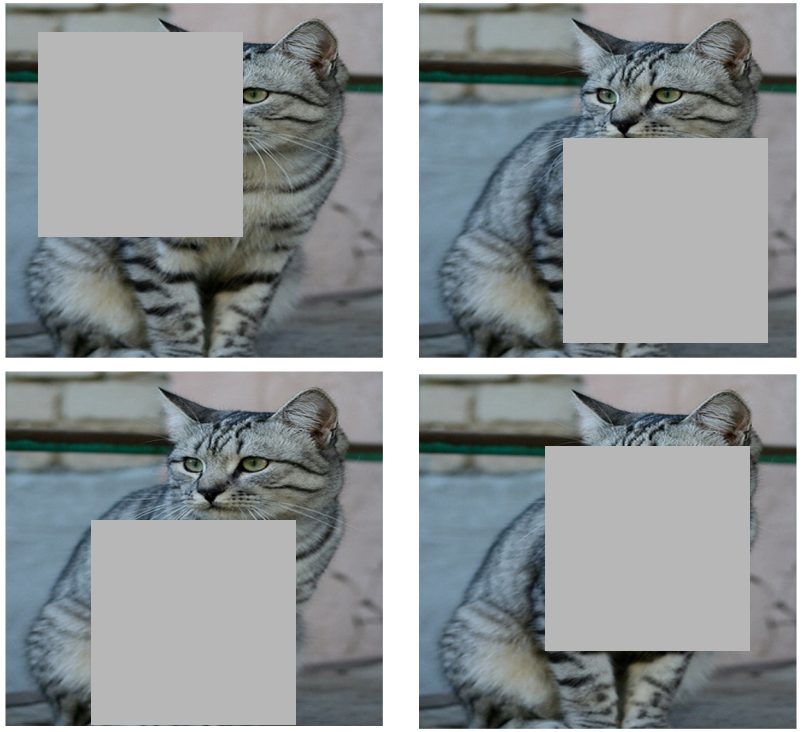


Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

Regularization: Cutout

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

Examples:
Dropout
Batch Normalization
Data Augmentation
Stochastic Depth
Cutout



Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017

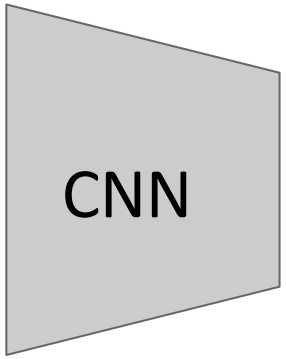
Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout
Mixup



Target label:
cat: 0.4
dog: 0.6

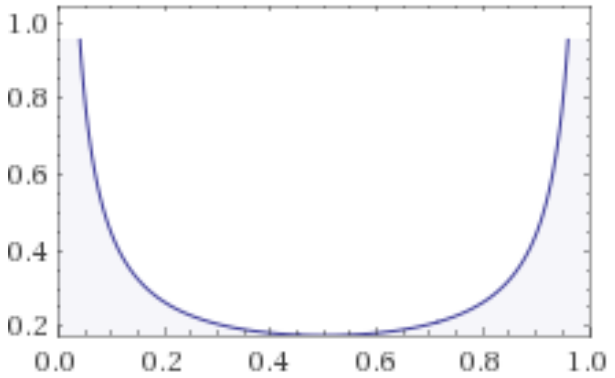
Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018
Why it works? some interesting analysis in the paper:
https://arxiv.org/pdf/1710.09412.pdf

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
Fractional Max Pooling
Stochastic Depth
Cutout
Mixup

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

CNN

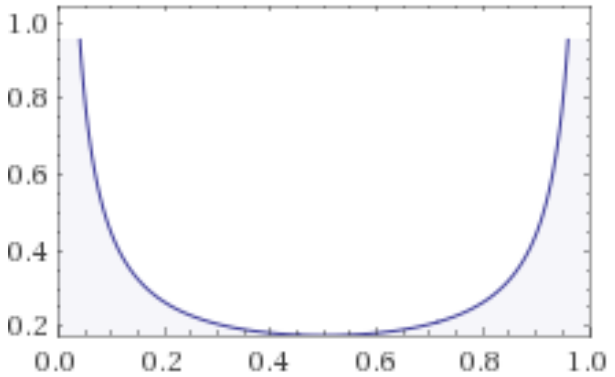
Target label:
cat: 0.4
dog: 0.6

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018
Why it works? some interesting analysis in the paper:
https://arxiv.org/pdf/1710.09412.pdf

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
Fractional Max Pooling
Stochastic Depth
Cutout
Mixup

Easy implementation

```
# y1, y2 should be one-hot vectors
for (x1, y1), (x2, y2) in zip(loader1, loader2):
    lam = numpy.random.beta(alpha, alpha)
    x = Variable(lam * x1 + (1. - lam) * x2)
    y = Variable(lam * y1 + (1. - lam) * y2)
    optimizer.zero_grad()
    loss(net(x), y).backward()
    optimizer.step()
```

Dataset	Model	ERM	*mixup*
CIFAR-10	PreAct ResNet-18	5.6	**4.2**
	WideResNet-28-10	3.8	**2.7**
	DenseNet-BC-190	3.7	**2.7**
CIFAR-100	PreAct ResNet-18	25.6	**21.1**
	WideResNet-28-10	19.4	**17.5**
	DenseNet-BC-190	19.0	**16.8**

(a) Test errors for the CIFAR experiments.

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018
Why it works? some interesting analysis in the paper:
https://arxiv.org/pdf/1710.09412.pdf

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

Cutout

Mixup

Message for brewing your networks

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try mixup (and its extension such as CutMix and AugMix) for some classification datasets

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

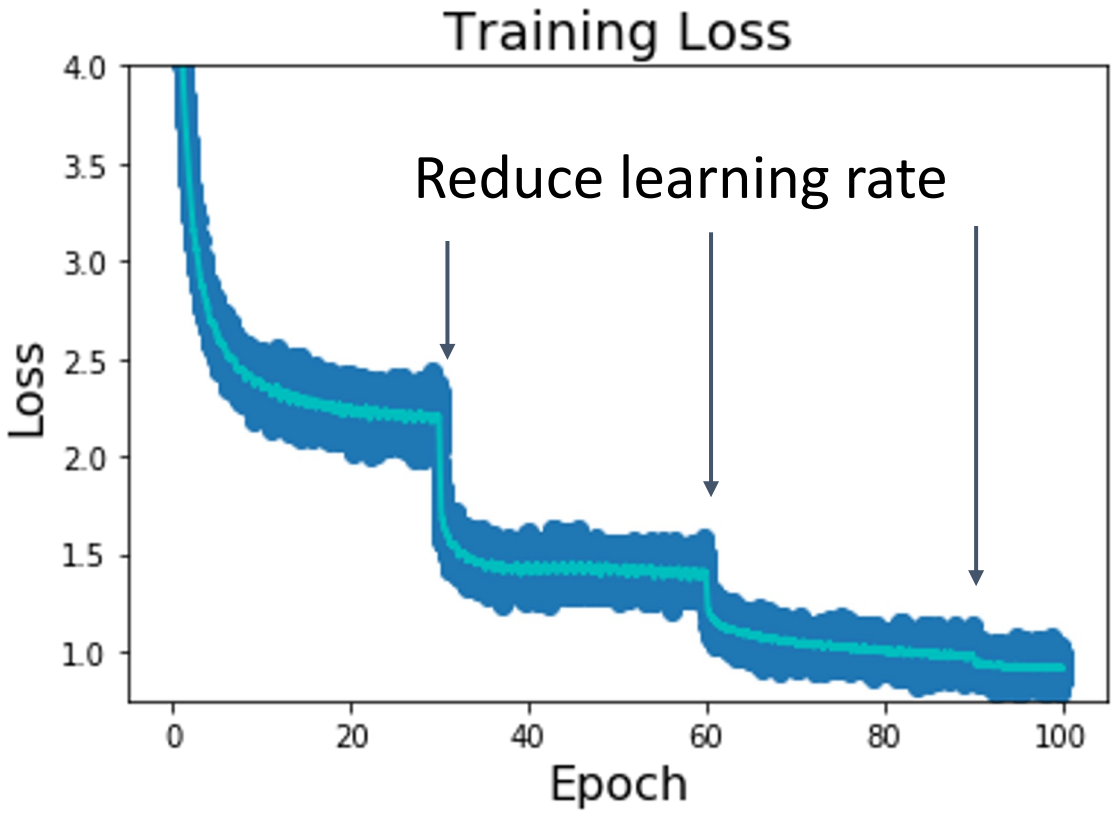
Overview

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

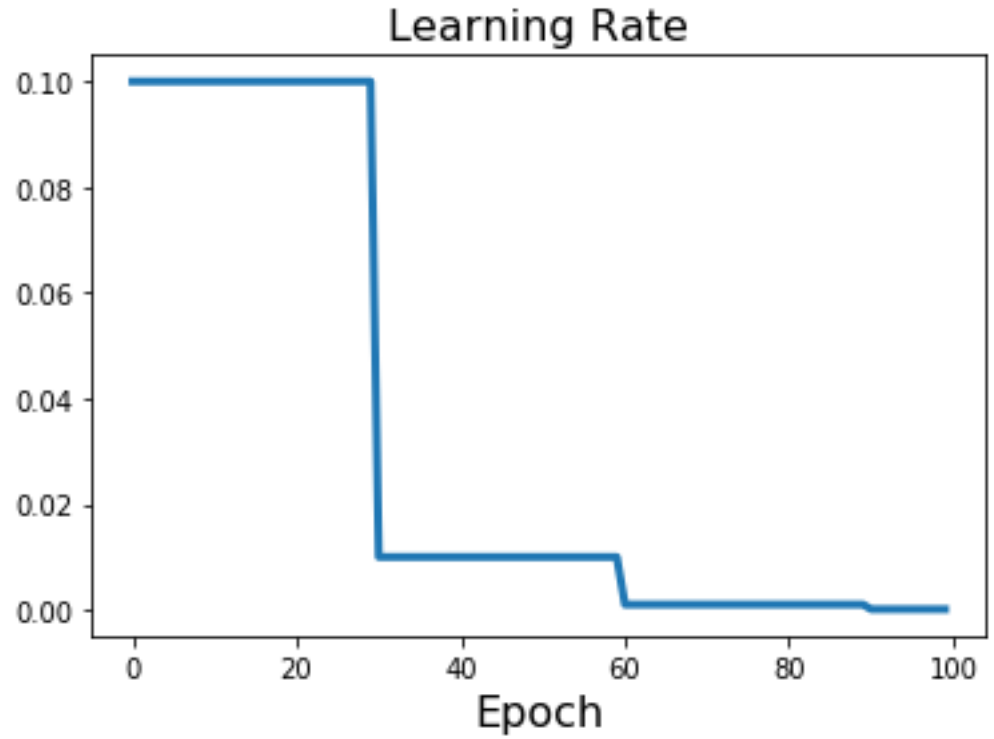
Next

Learning Rate Schedules

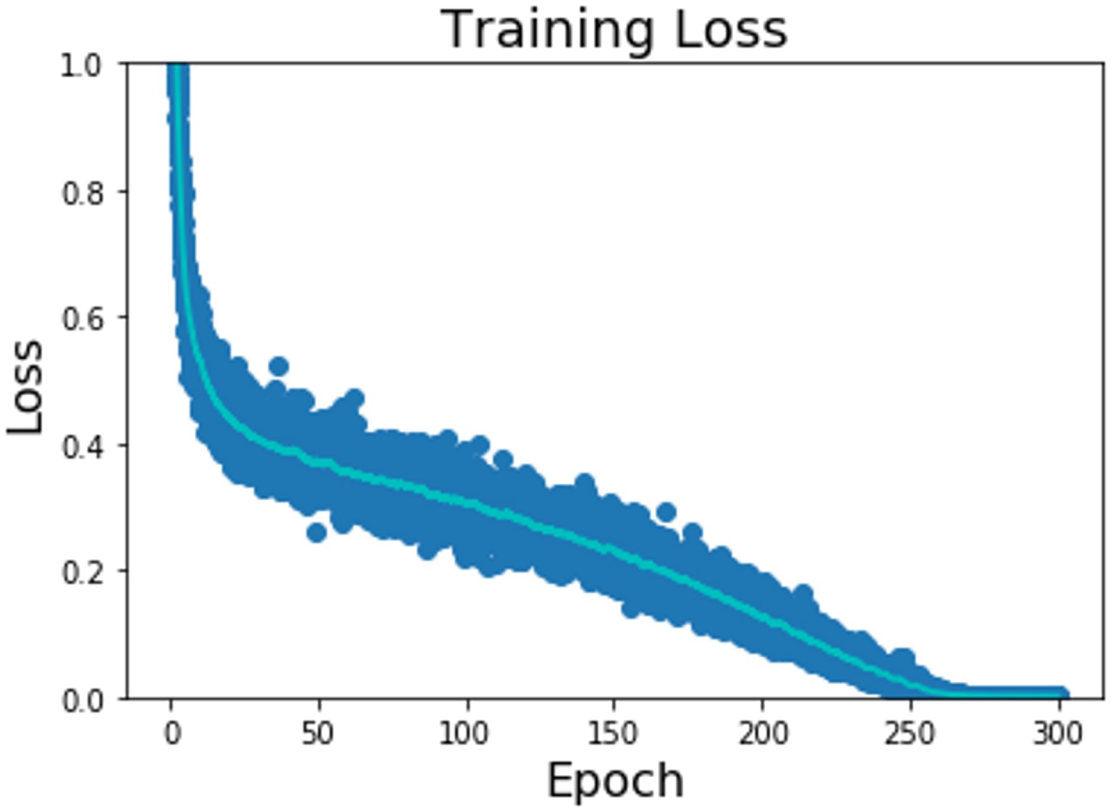
Learning Rate Decay: Step (most common)



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

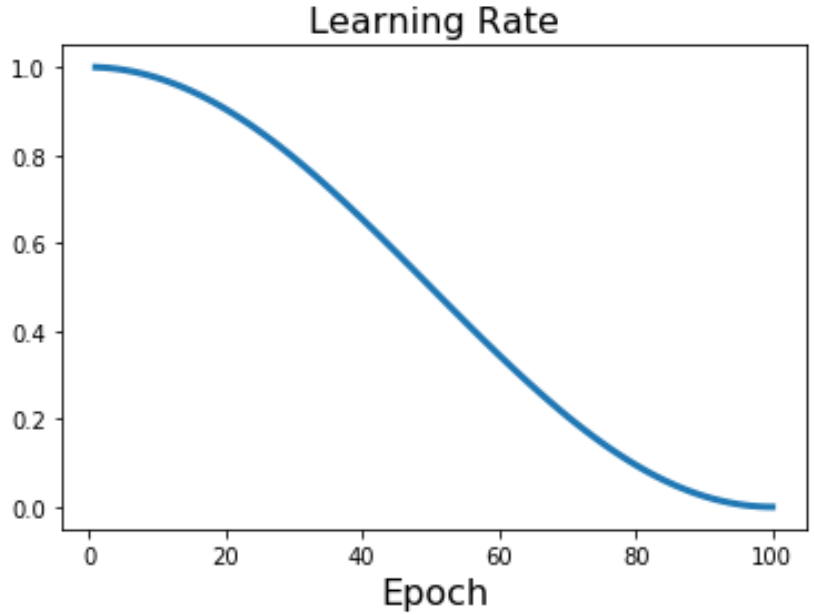


Learning Rate Decay: Cosine



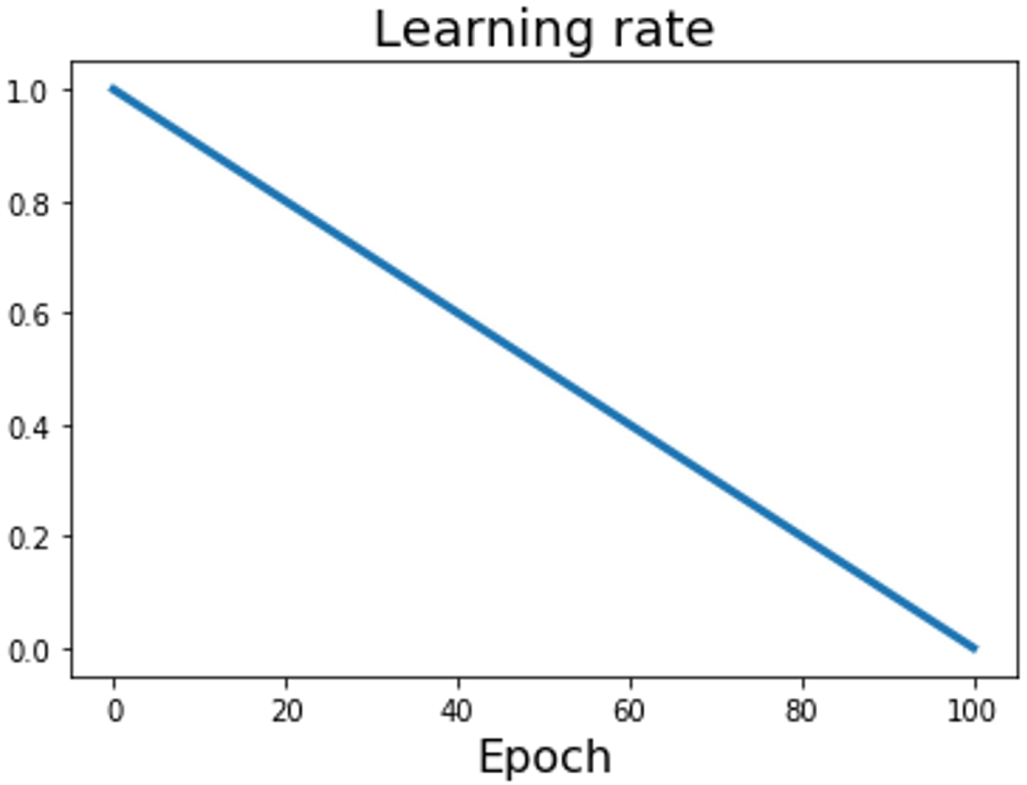
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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



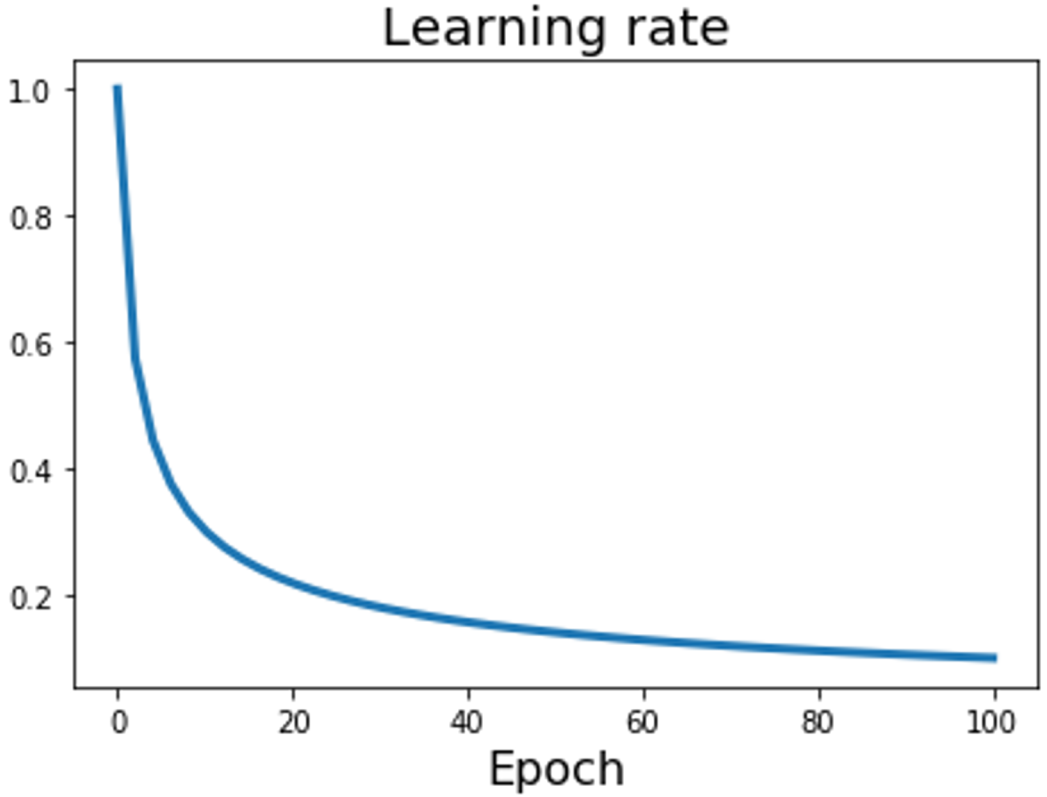
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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.

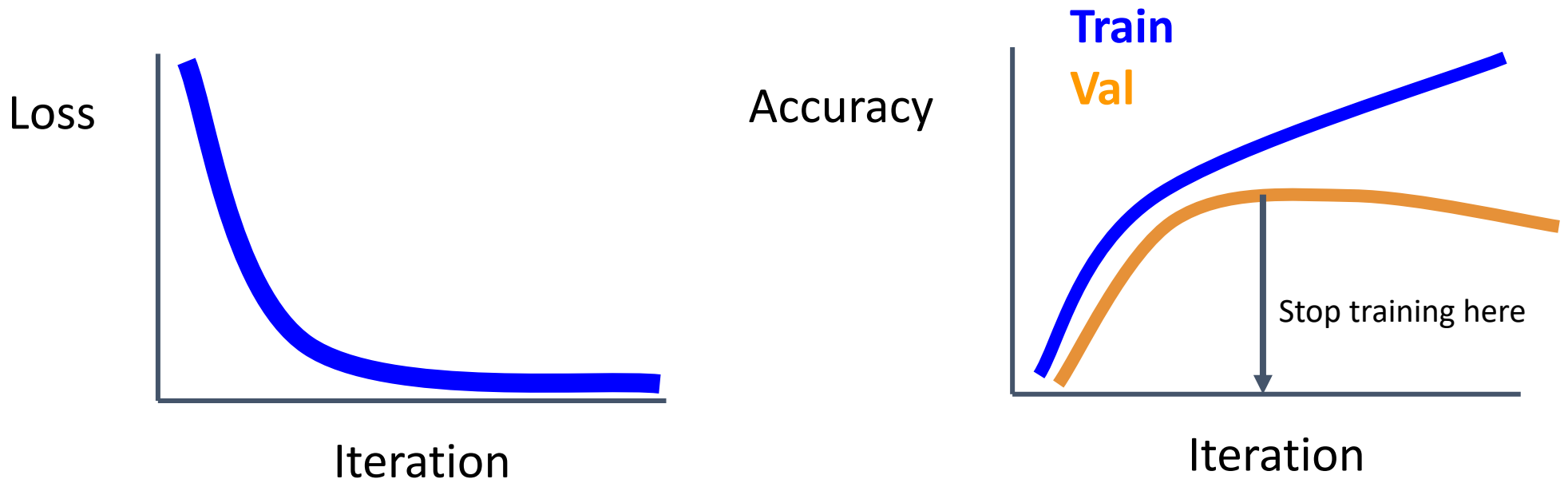
Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

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

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

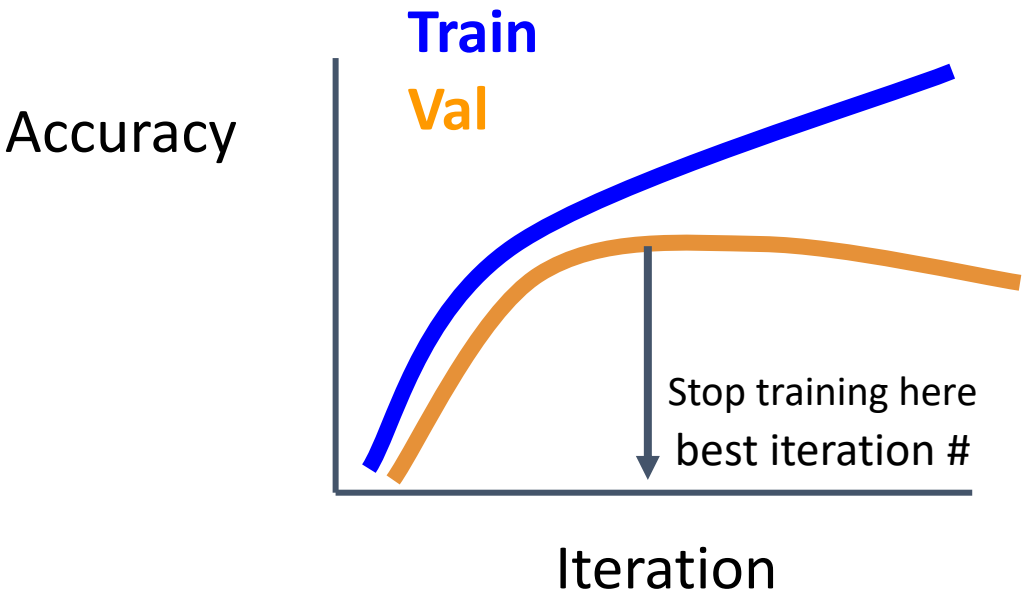
Vaswani et al, "Attention is all you need", NIPS 2017

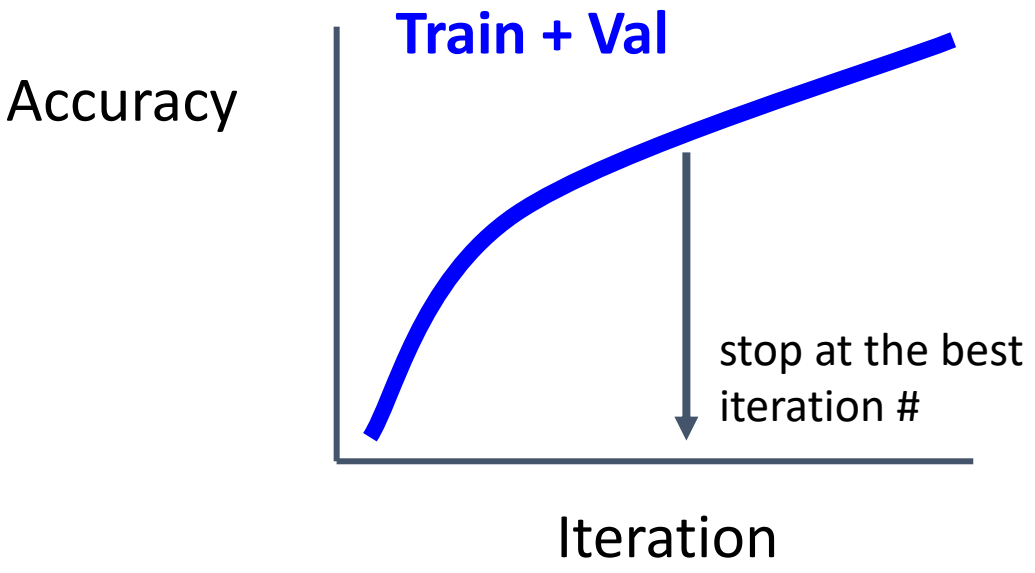
How long to train? **Early Stopping**



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!**

How long to train? **Early Stopping**





Combine the train and val sets to squeeze the most

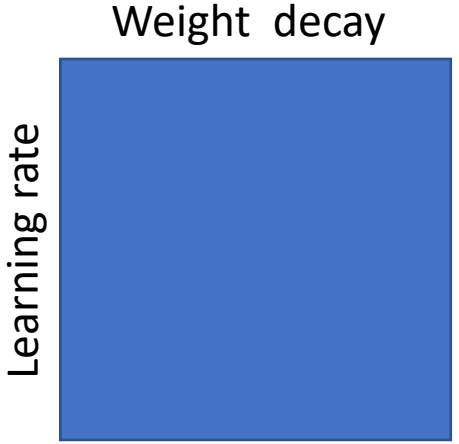
Choosing Hyperparameters

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



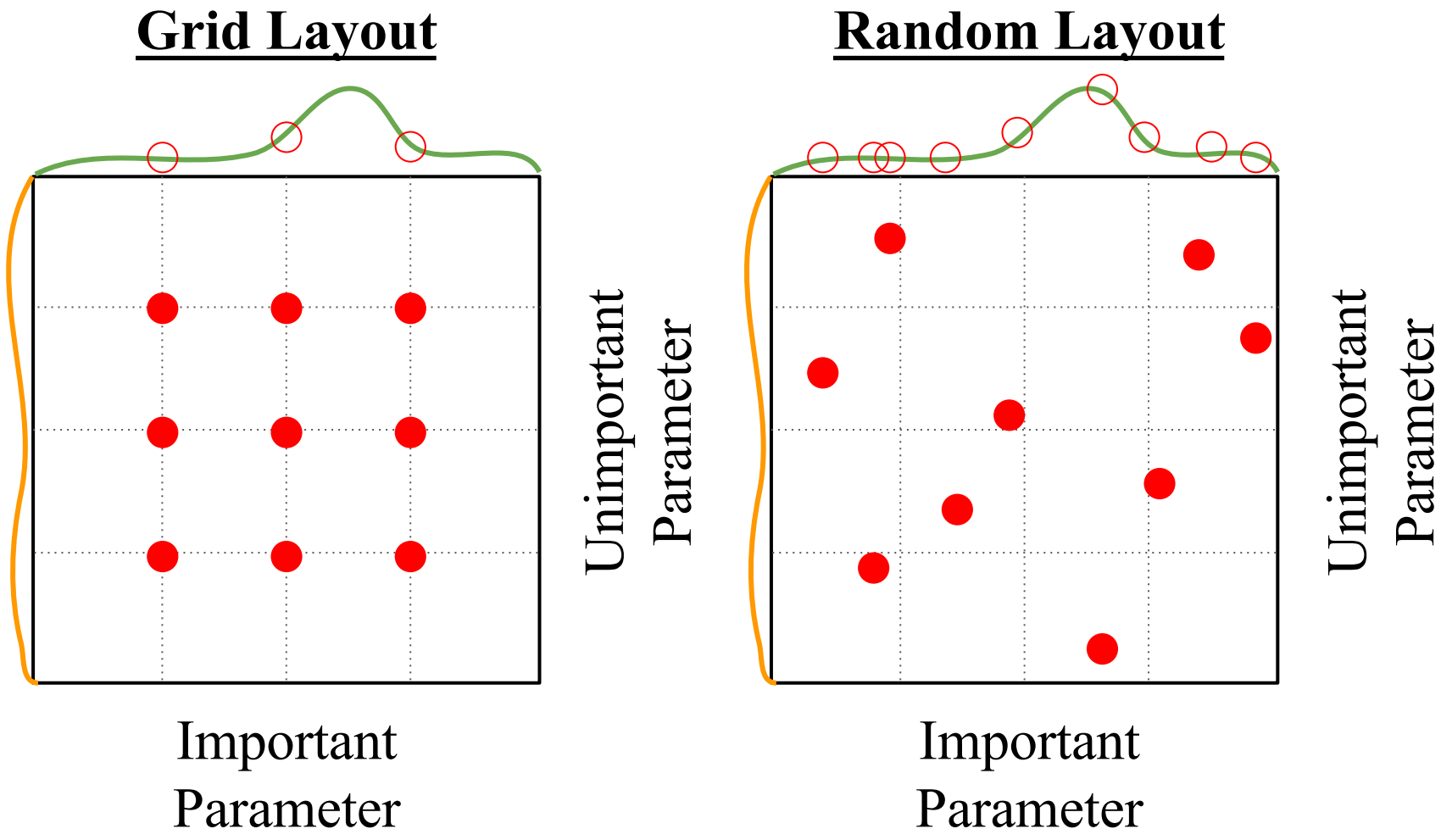
Choosing Hyperparameters: Random Search

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 $[1x10^{-4}, 1x10^{-1}]$

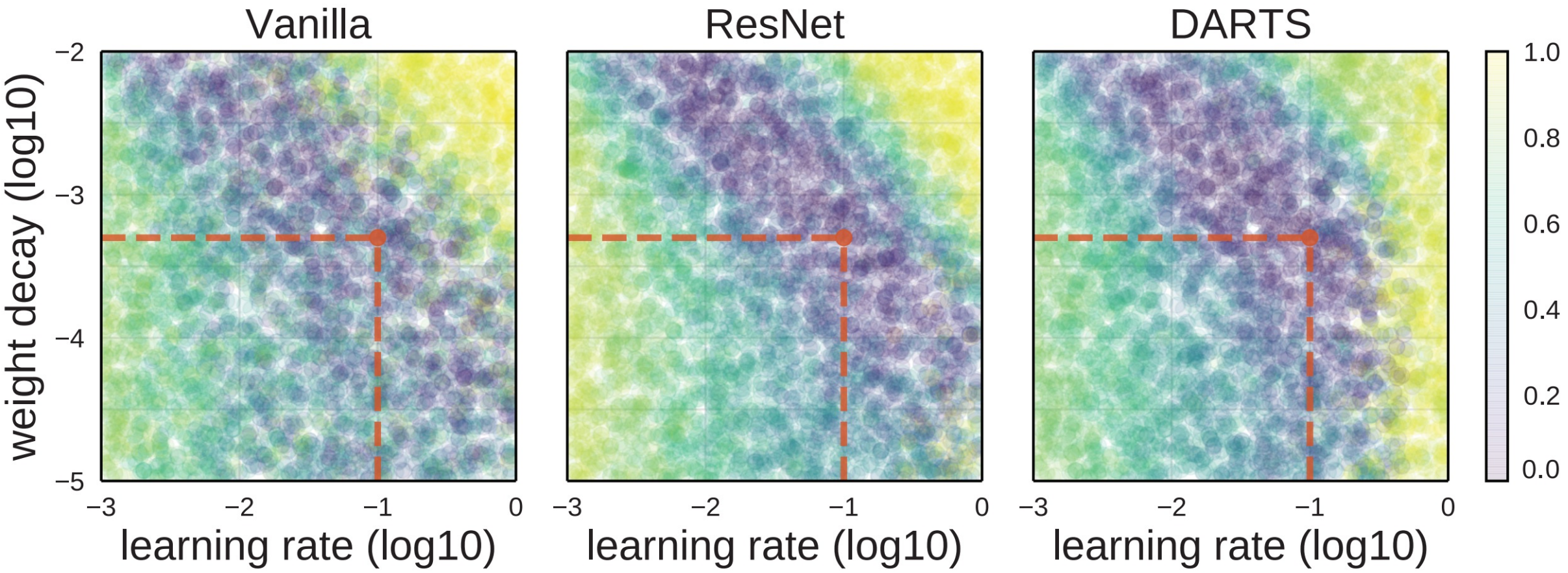
Run many different trials

Hyperparameters: Random vs Grid Search



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

Choosing Hyperparameters: Random Search



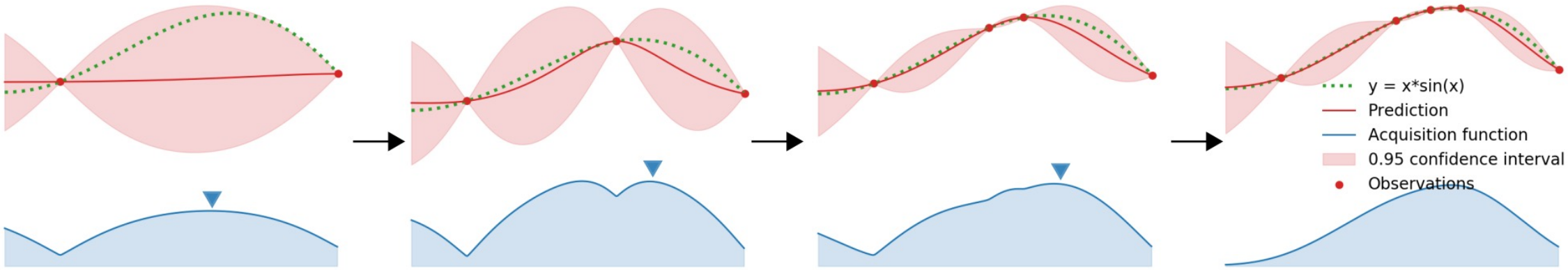
Training 5K models tells: One single learning rate and weight decay are effective across all three design spaces.

Radosavovic et al, "On Network Design Spaces for Visual Recognition", ICCV 2019
https://arxiv.org/pdf/1905.13214.pdf

Choosing Hyperparameters: Bayesian Optimization

It can be modeled as a sequential design strategy for global optimization of black-box functions (no gradient can be accessed)

$$P(f|S) = \frac{P(S|f)P(f)}{P(S)}$$



https://towardsdatascience.com/bayesian-optimization-and-hyperparameter-tuning-6a22f14cb9fa

A visual tutorial: https://distill.pub/2020/bayesian-optimization/

https://arxiv.org/pdf/2104.12365.pdf

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); modify 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

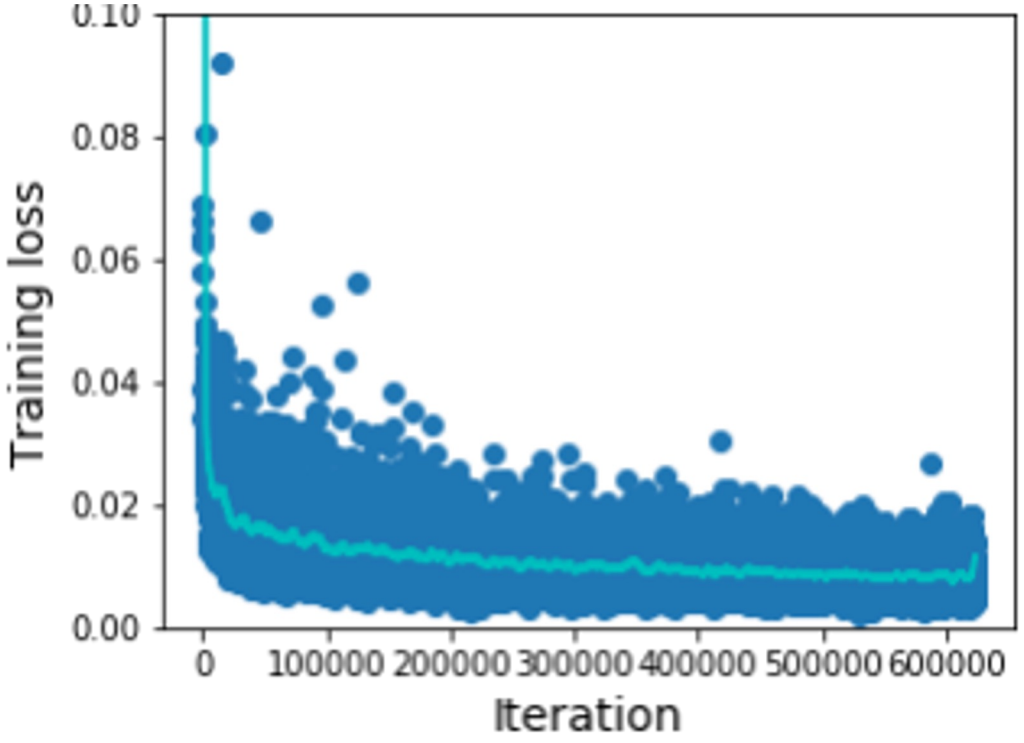
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer

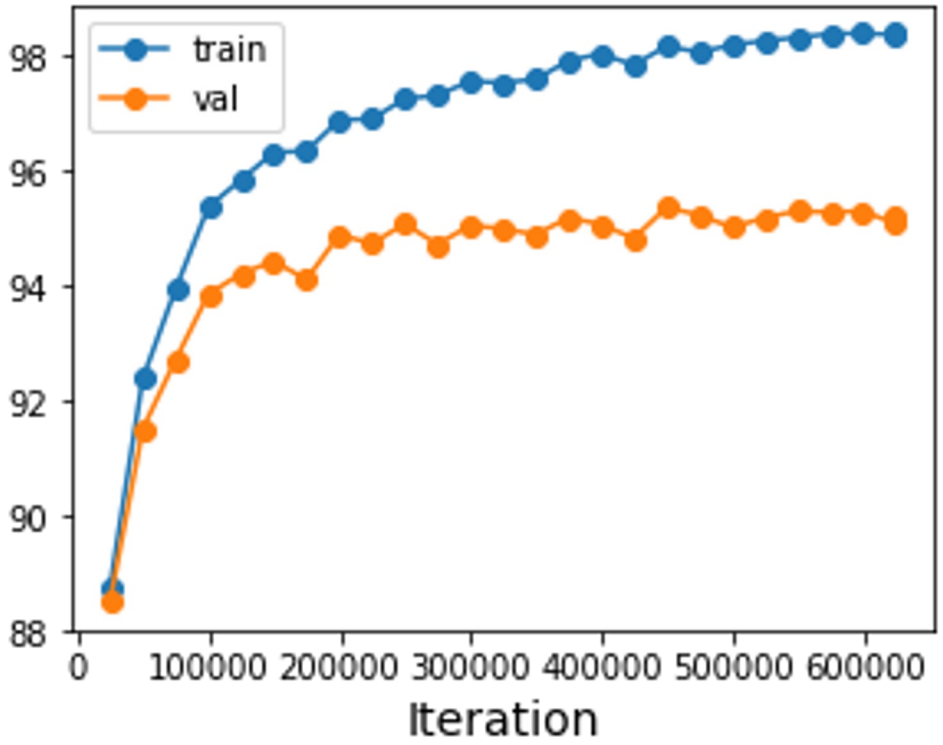
Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

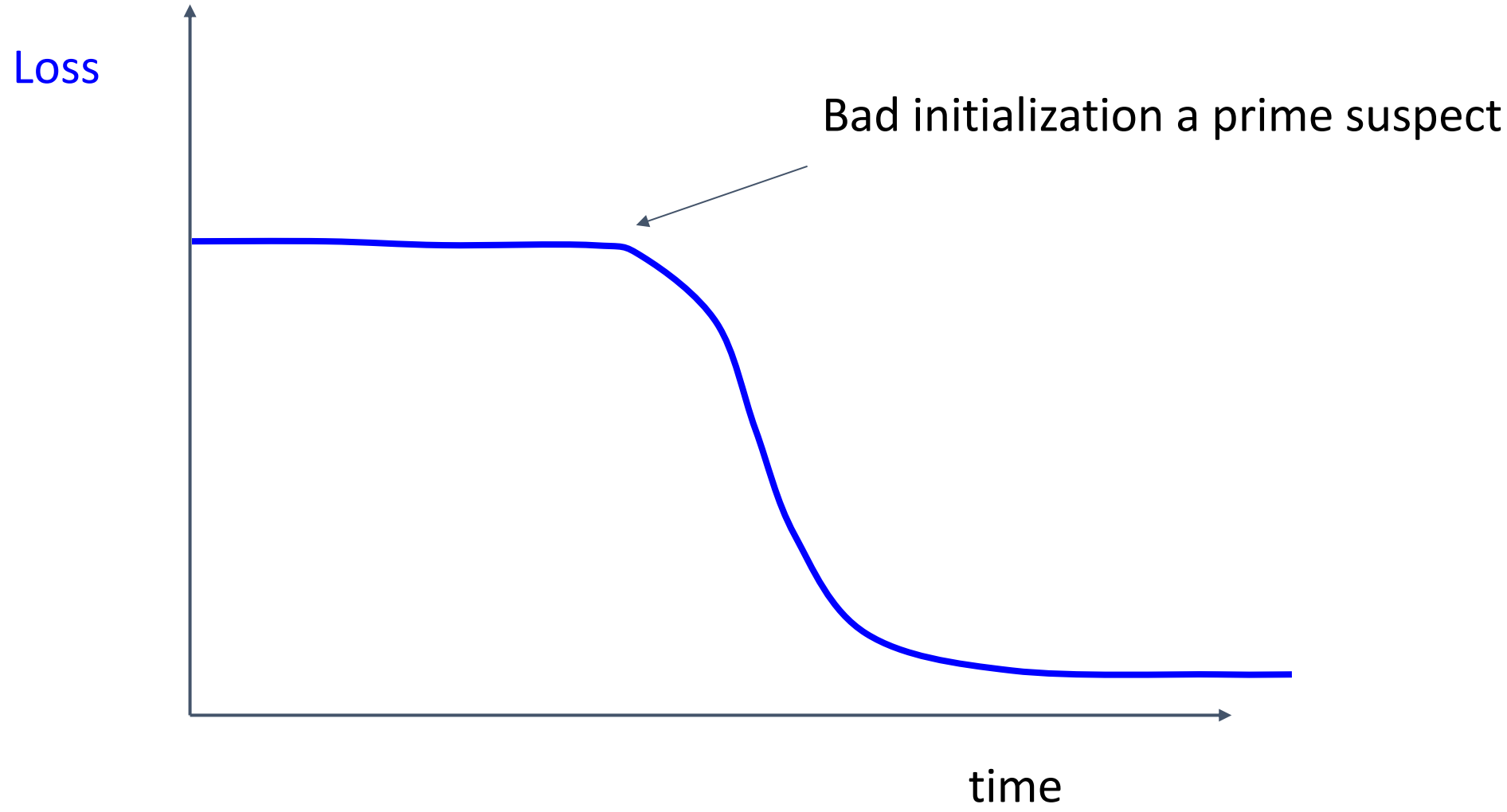
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at learning curves

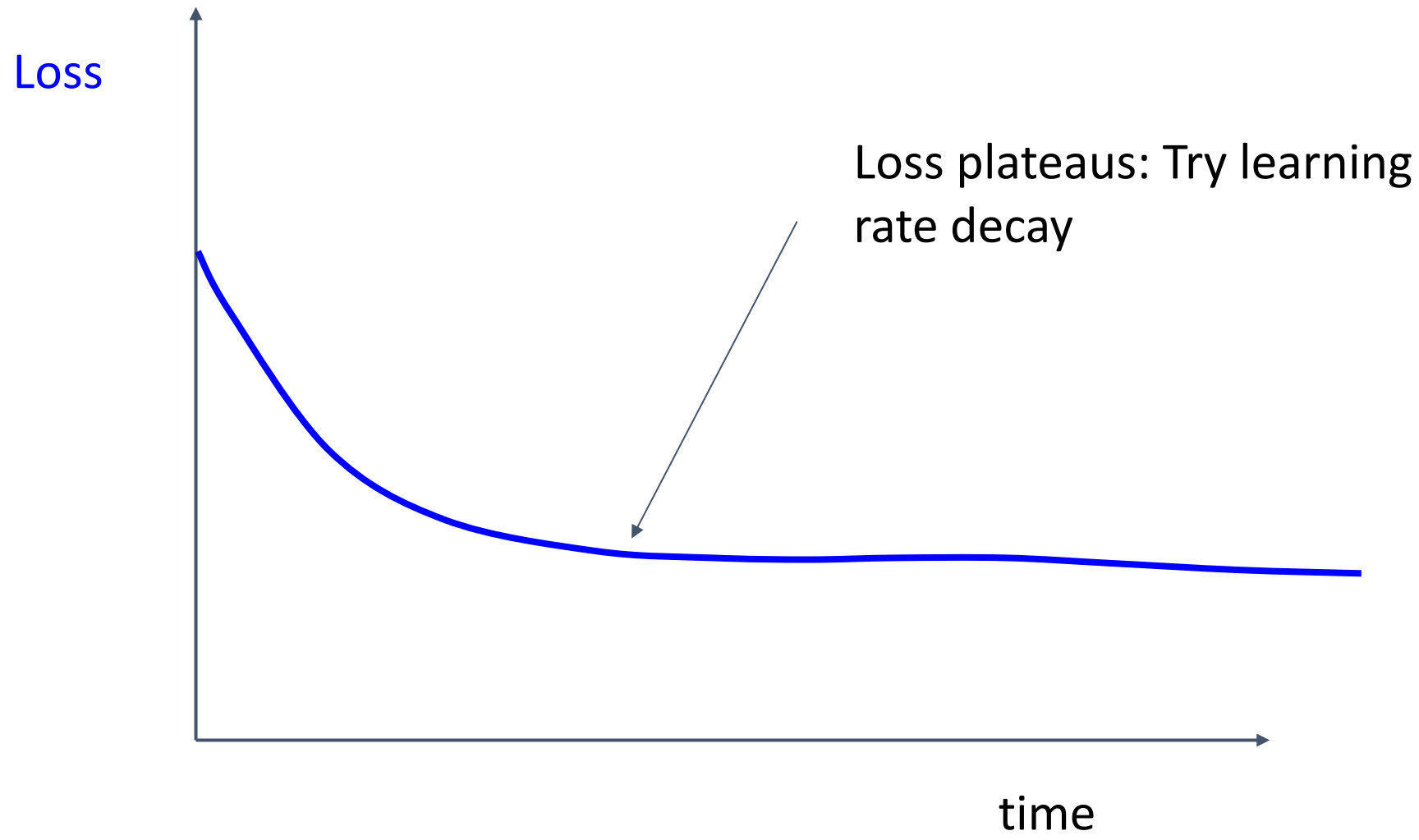
Look at Learning Curves!

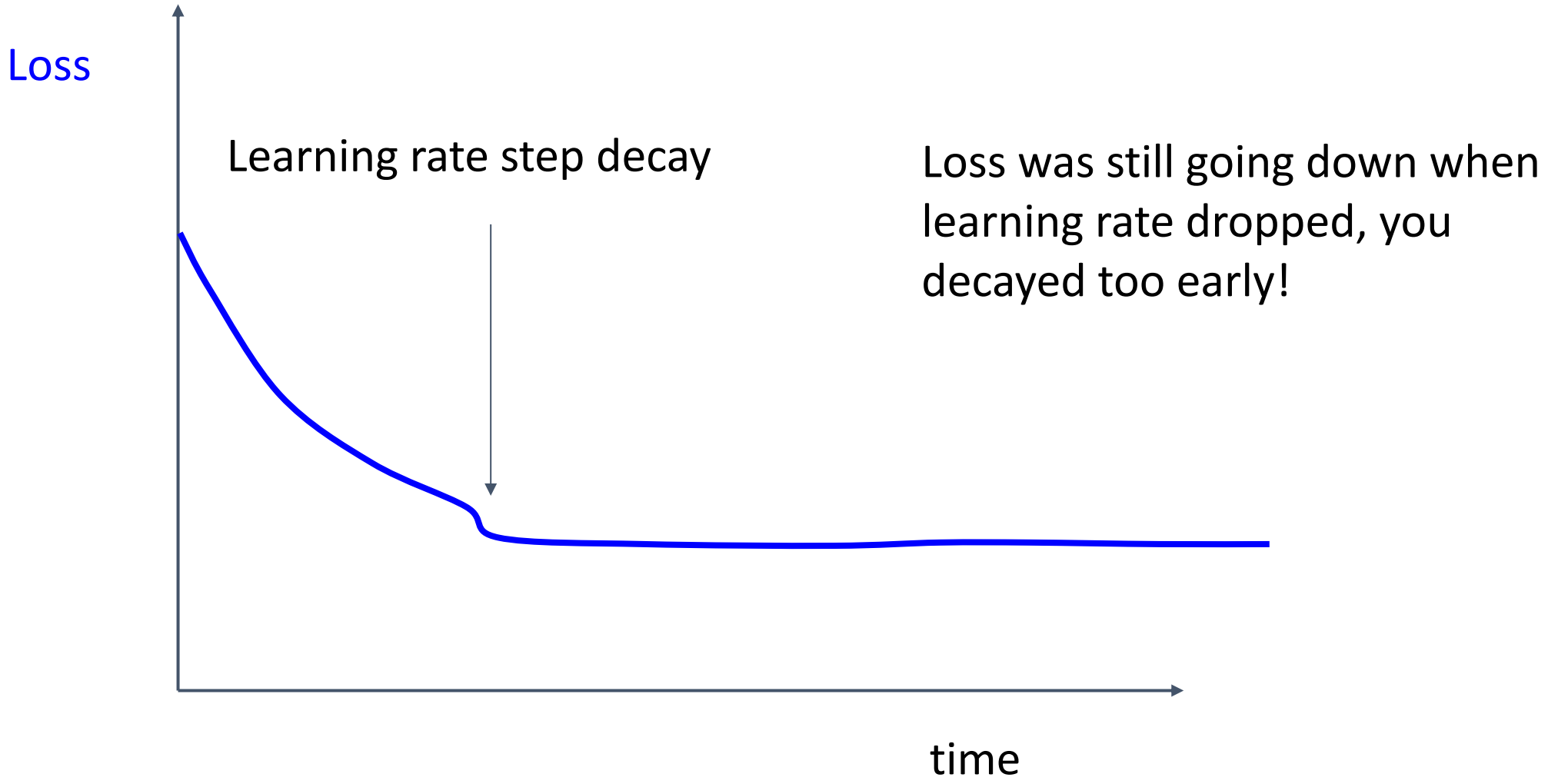


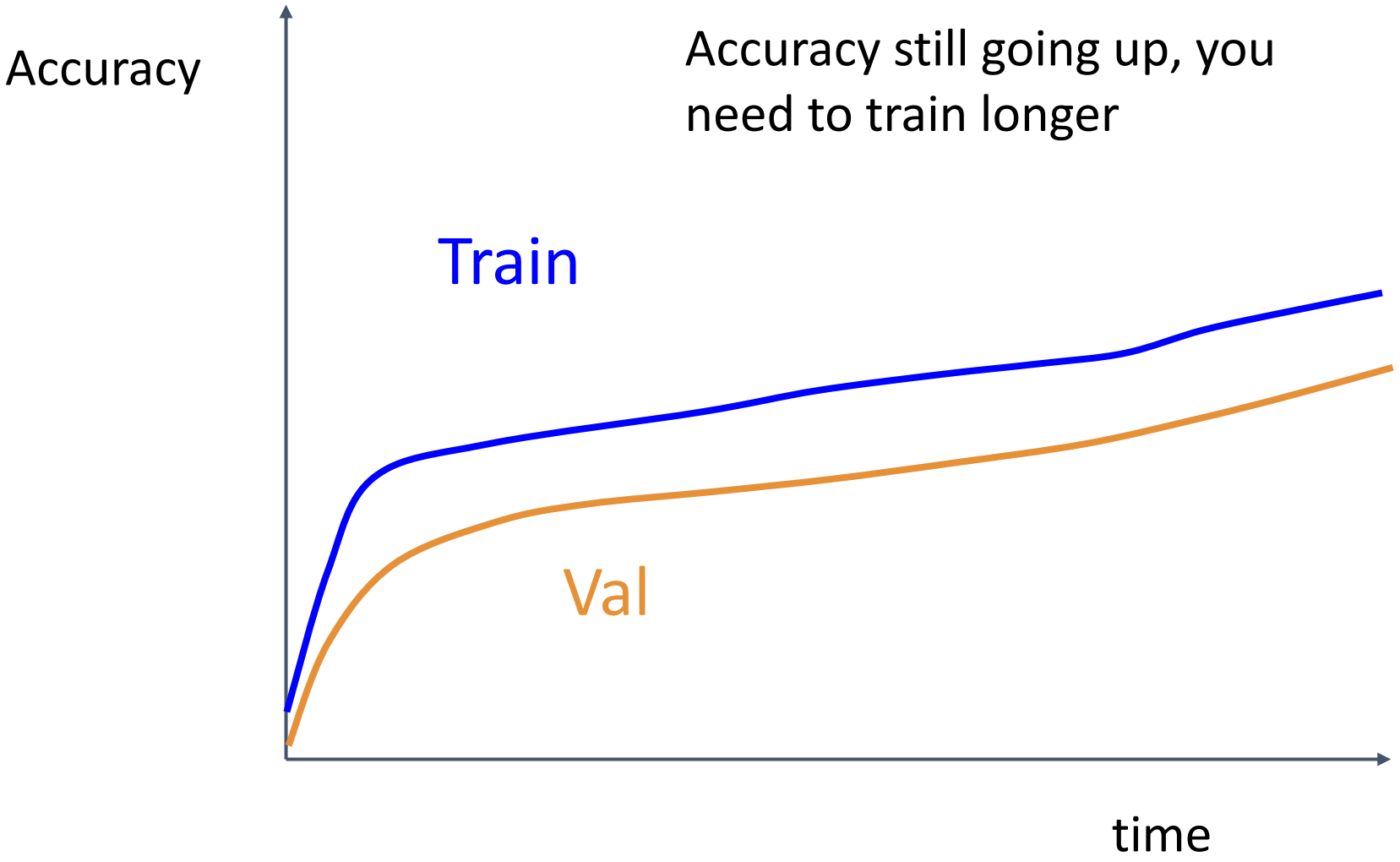


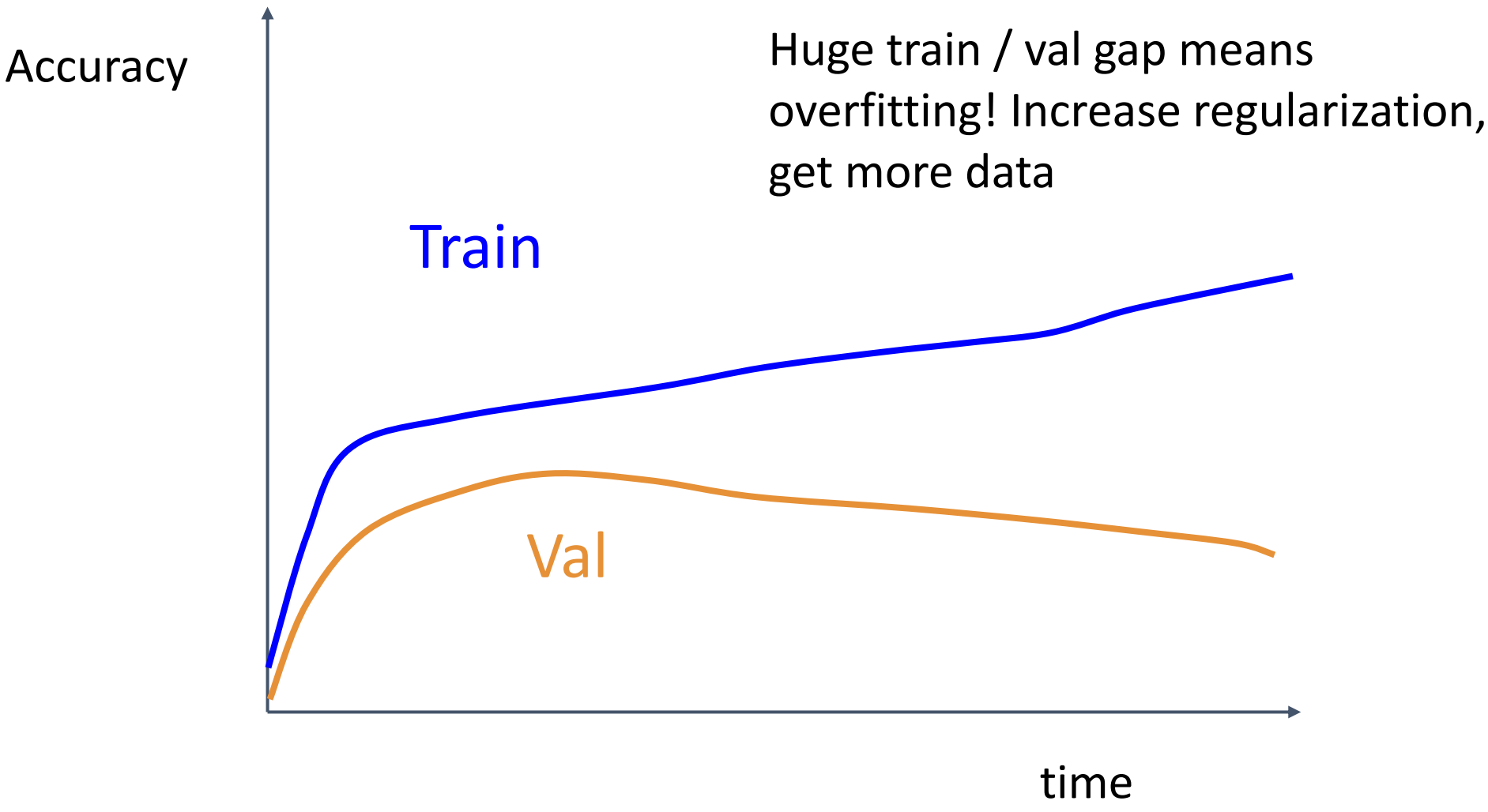
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

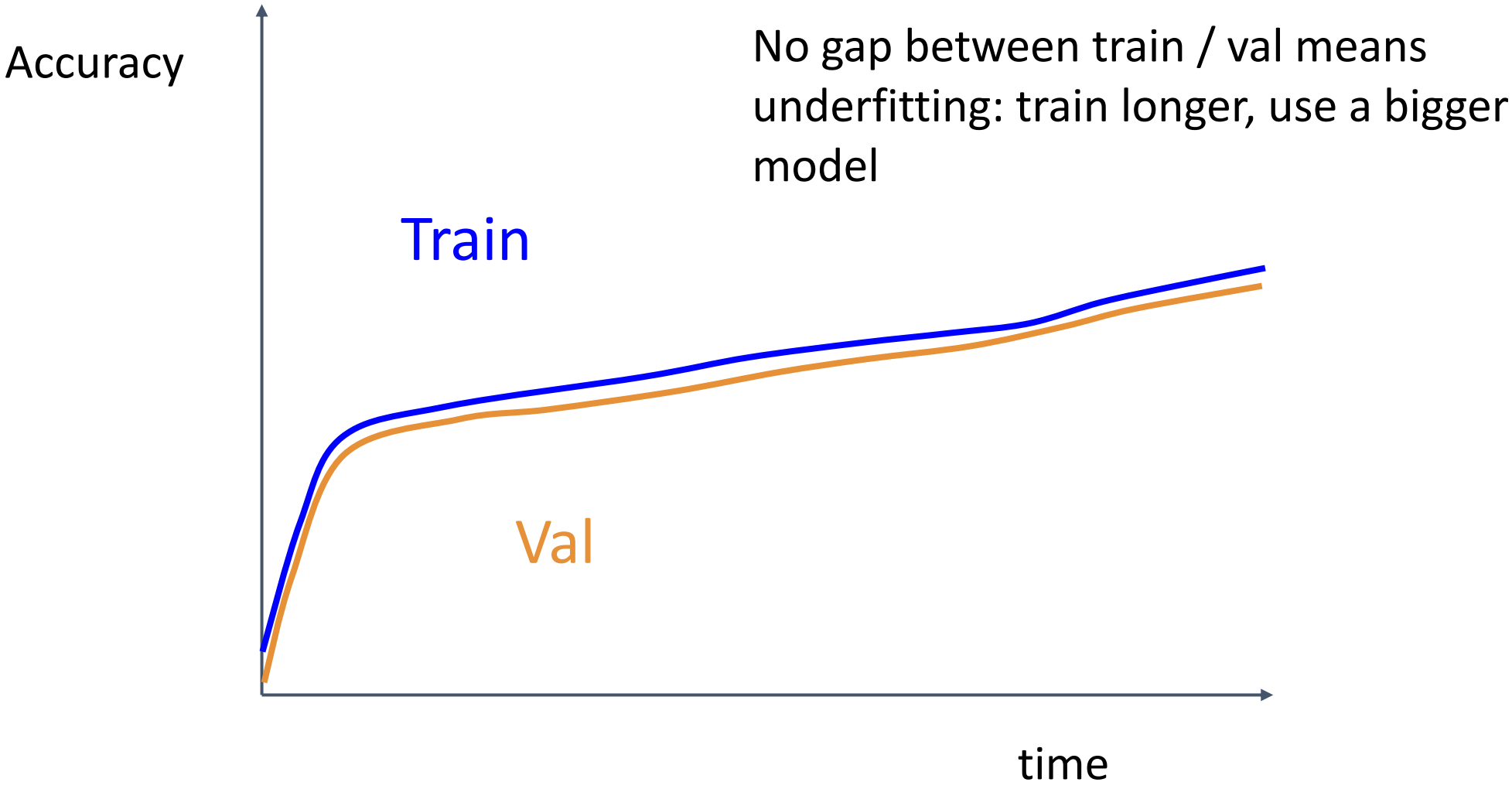

Loss
Bad initialization a prime suspect
time


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 gap between train / val means underfitting: train longer, use a bigger model
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 as a DJ:

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

neural networks practitioner
music = loss function

Overview

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

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

Transfer Learning

Transfer Learning

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

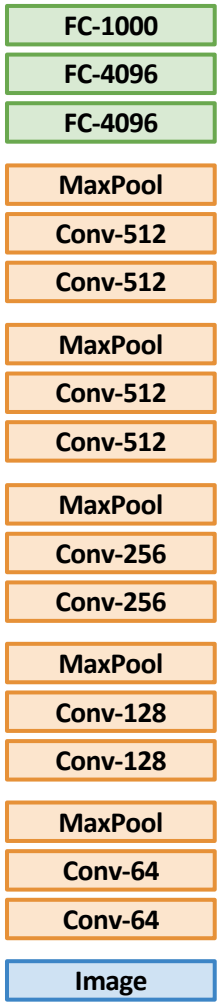
Transfer Learning

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

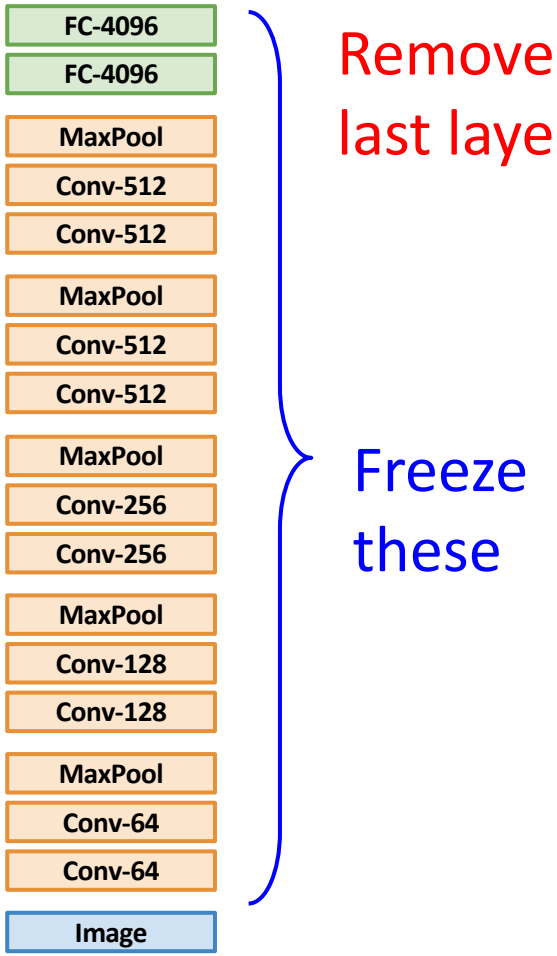
BUSTED

Transfer Learning with CNNs

1. Train on Imagenet



2. Use CNN as a feature extractor



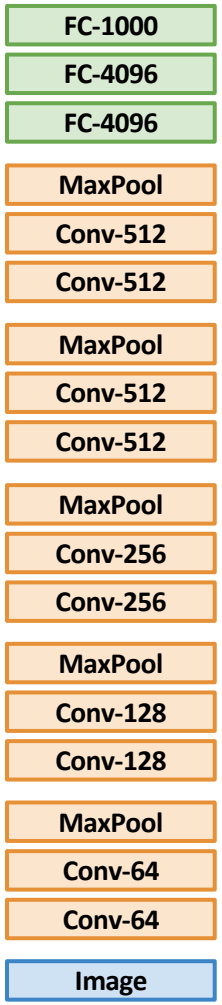
Remove last layer

Freeze these

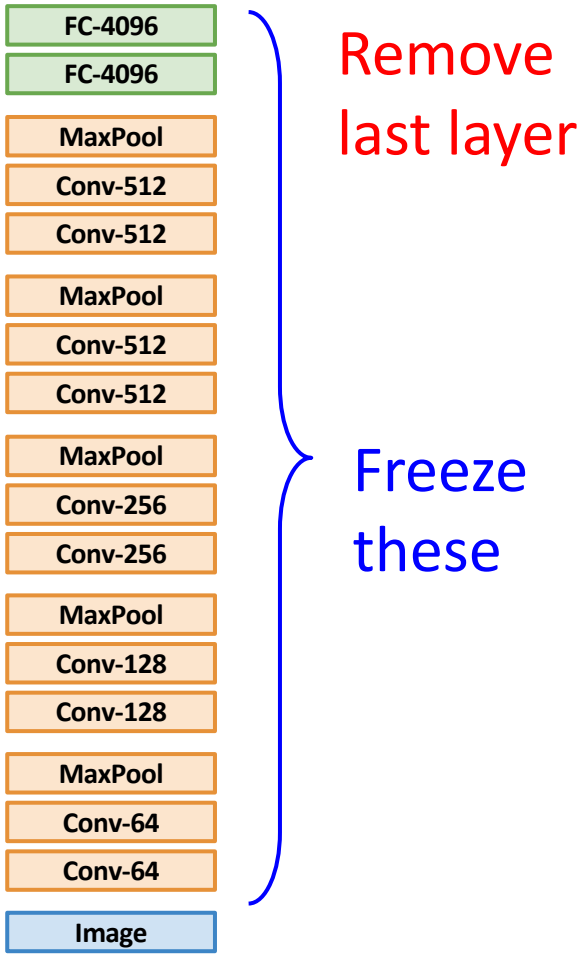
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs

1. Train on Imagenet



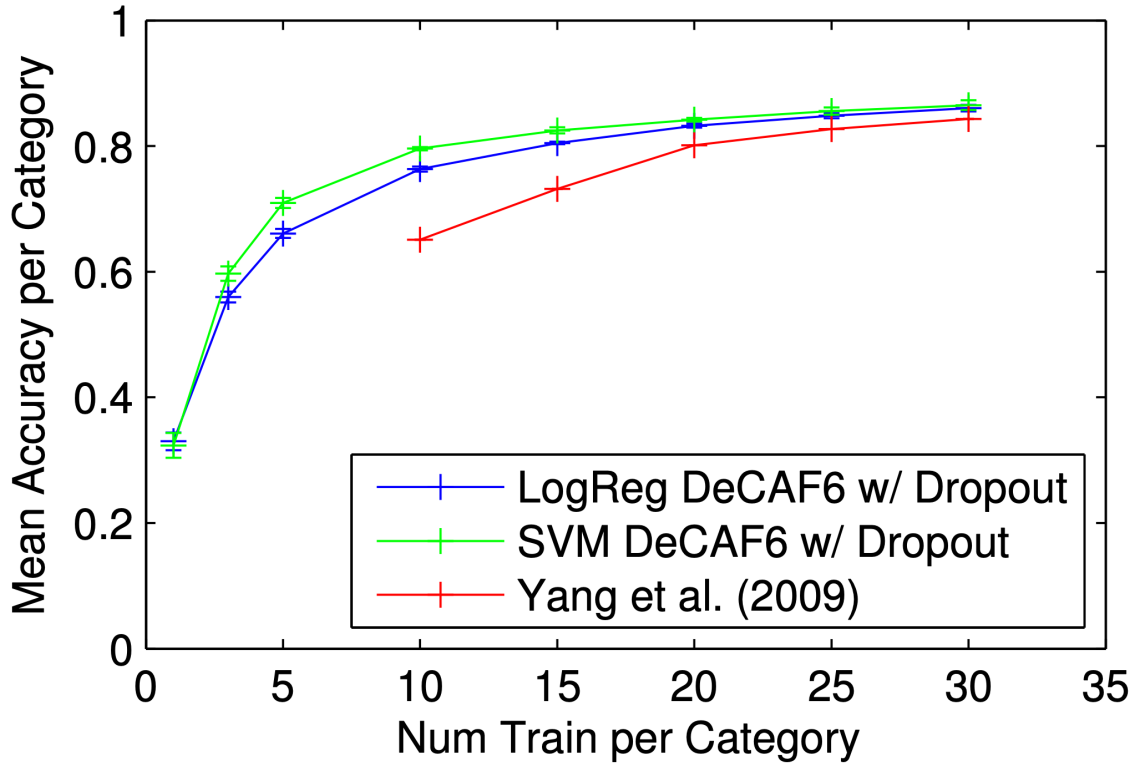
2. Use CNN as a feature extractor



Remove last layer

Freeze these

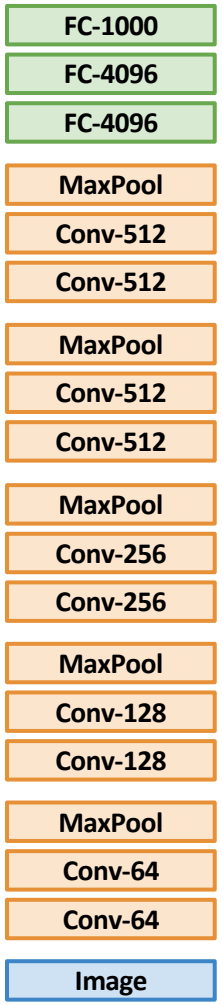
Classification on Caltech-101



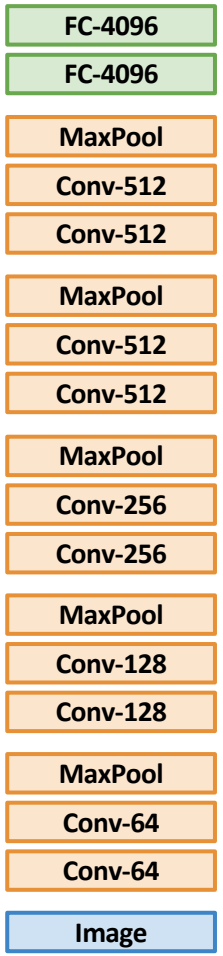
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs

1. Train on Imagenet



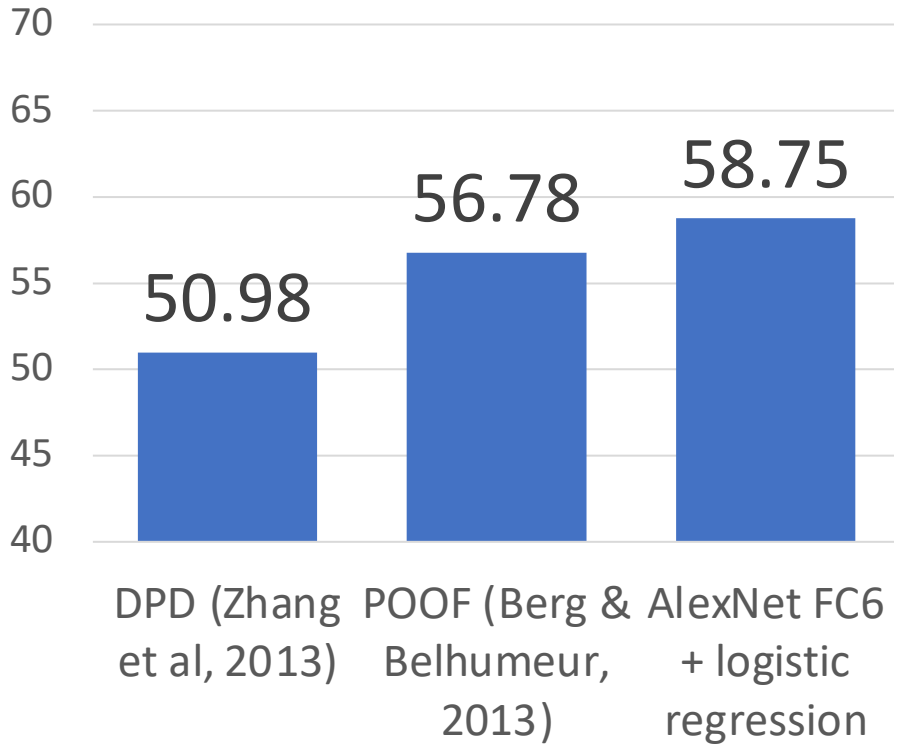
2. Use CNN as a feature extractor



Remove last layer

Freeze these

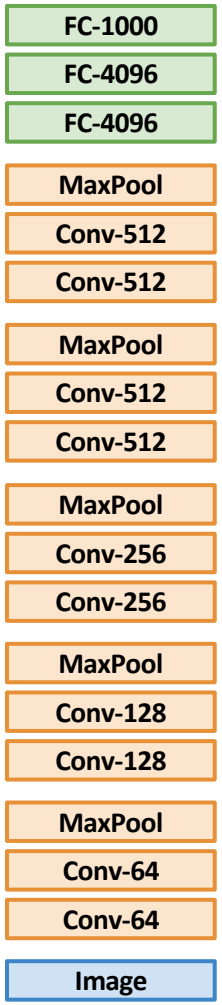
Bird Classification on Caltech-UCSD



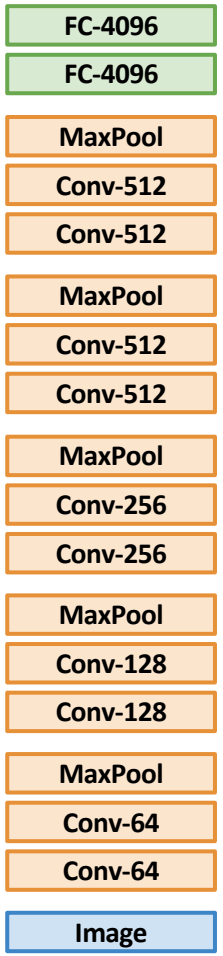
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs

1. Train on Imagenet



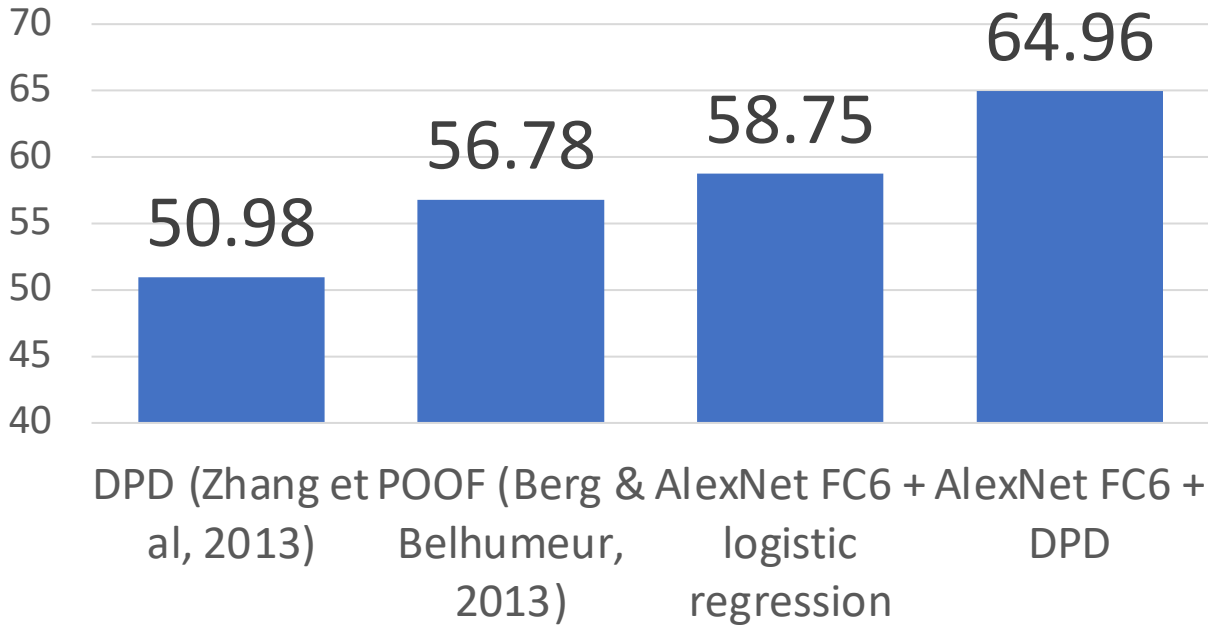
2. Use CNN as a feature extractor



Remove last layer

Freeze these

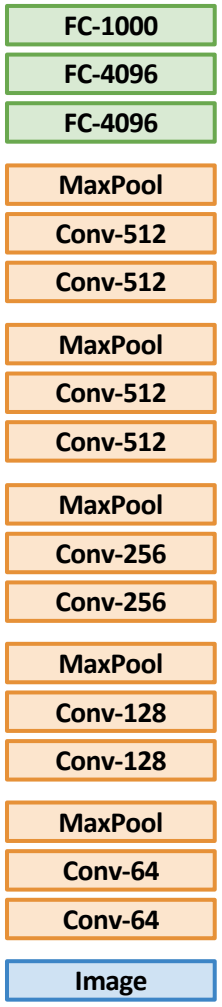
Bird Classification on Caltech-UCSD



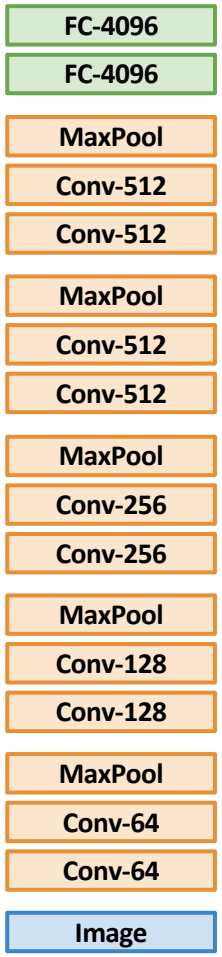
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

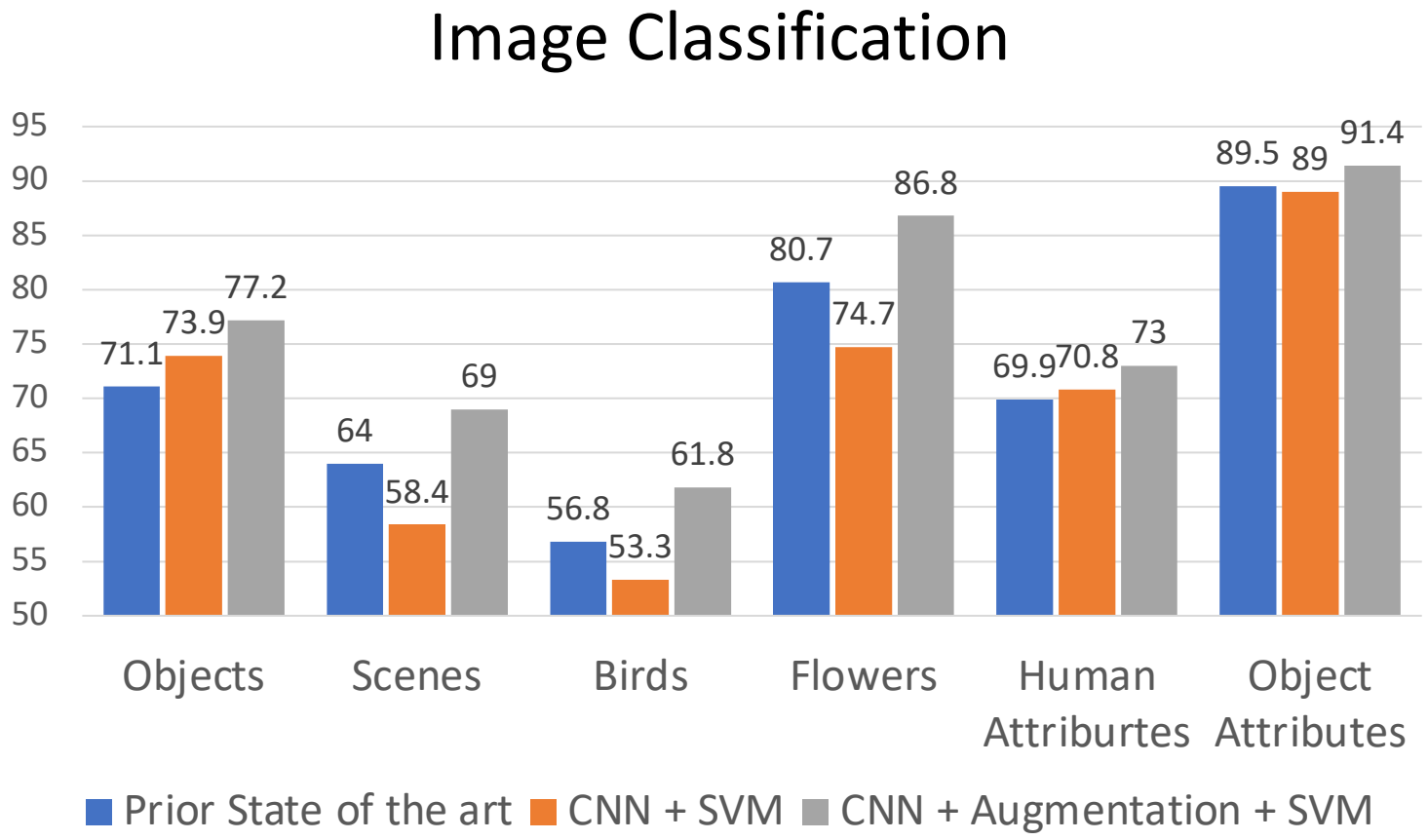
Transfer Learning with CNNs

1. Train on Imagenet



2. Use CNN as a feature extractor

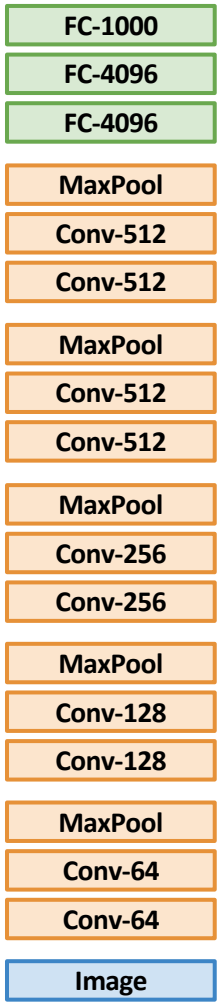




Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

1. Train on Imagenet



2. Use CNN as a feature extractor

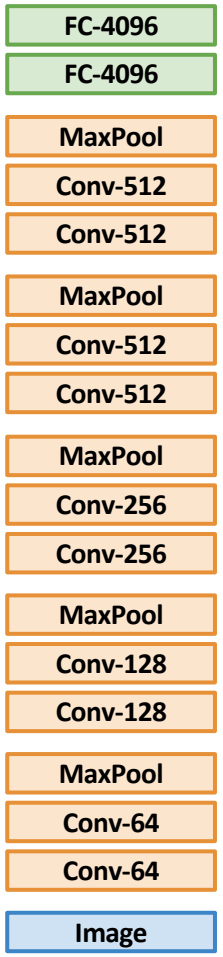
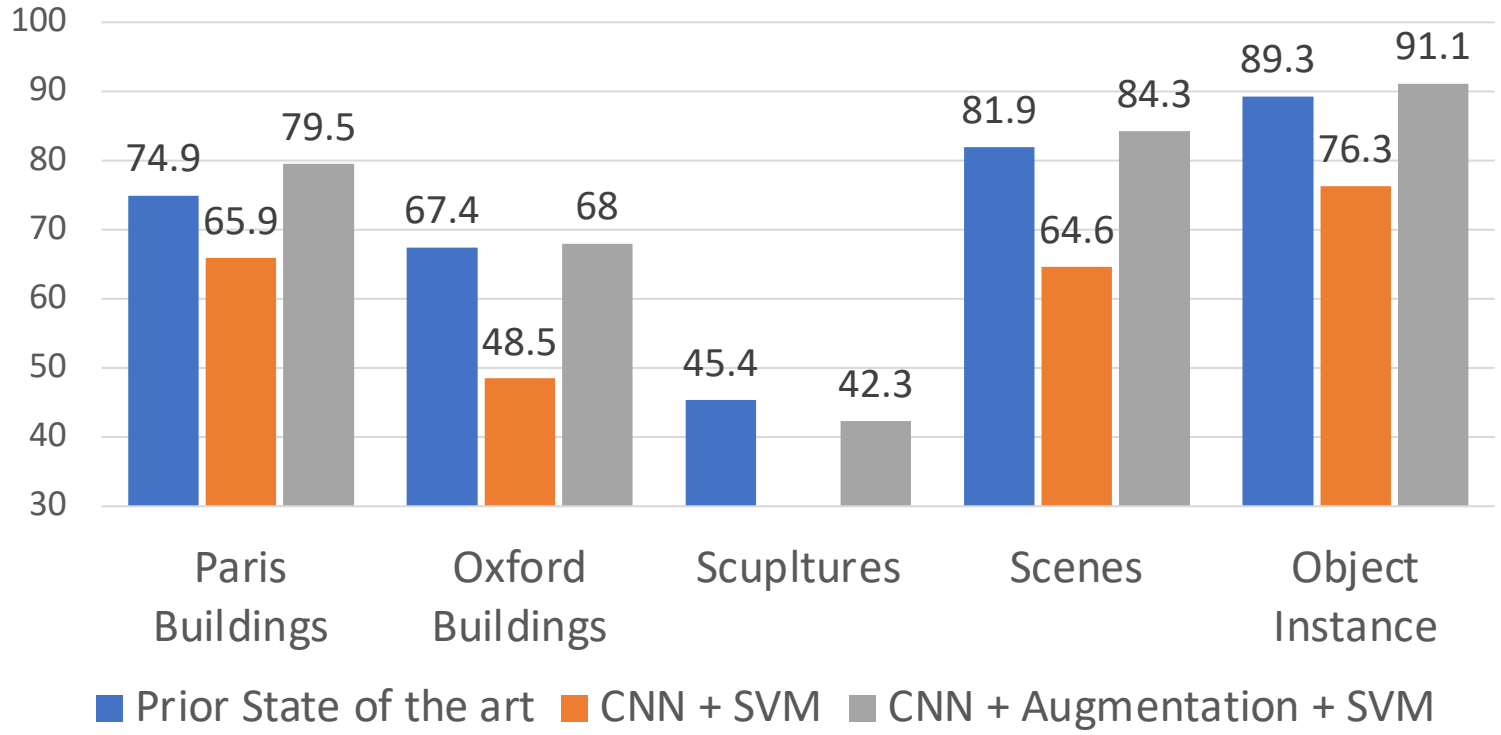


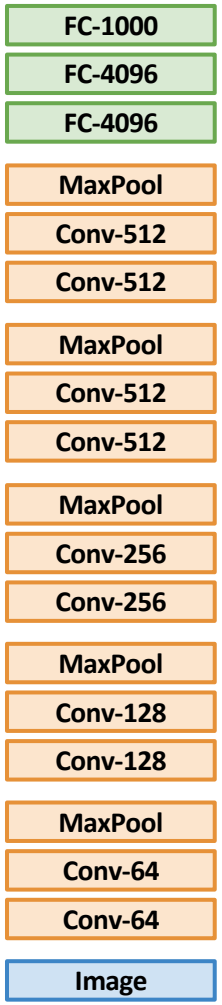
Image Retrieval: Nearest-Neighbor



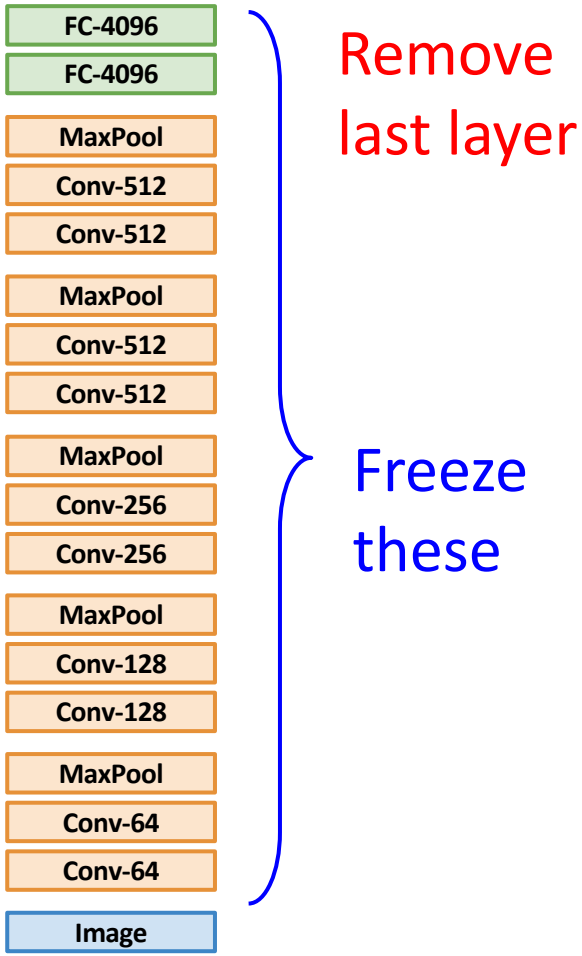
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

1. Train on Imagenet



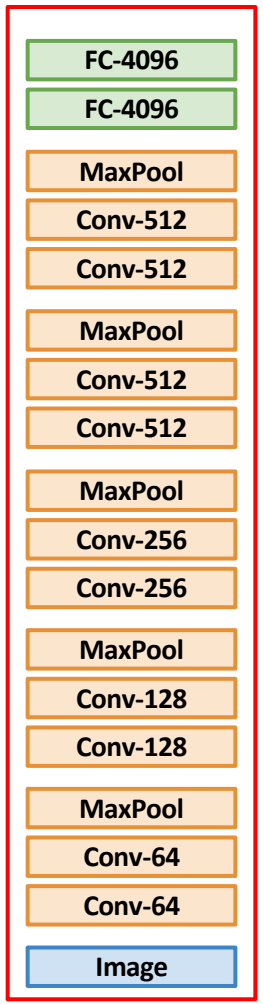
2. Use CNN as a feature extractor



Remove last layer

Freeze these

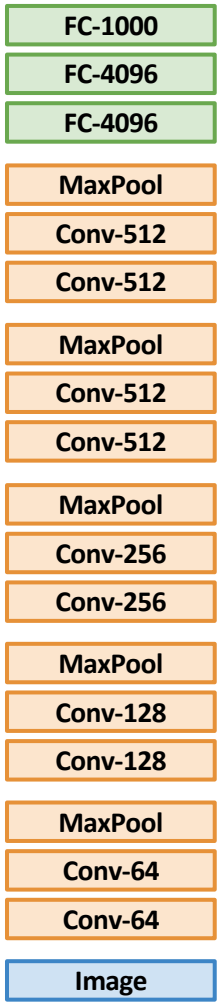
3. Bigger dataset:
Fine-Tuning



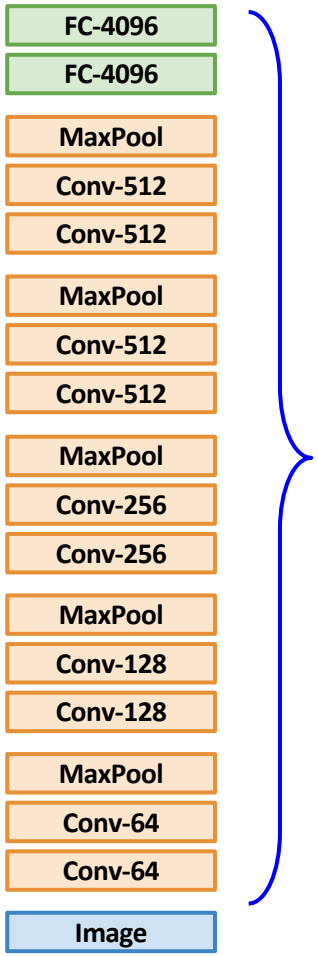
Continue training CNN for new task!

Transfer Learning with CNNs

1. Train on Imagenet



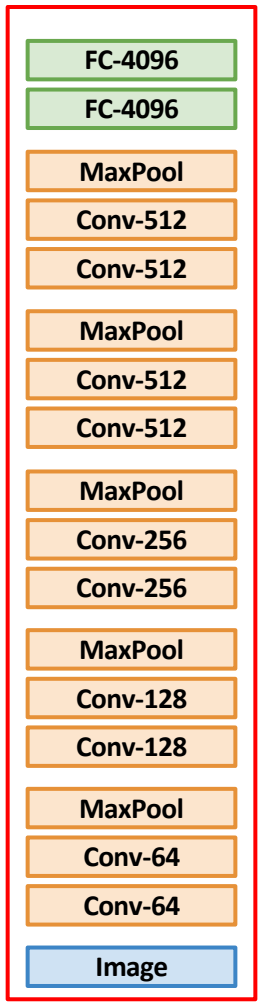
2. Use CNN as a feature extractor



Remove last layer

Freeze these

3. Bigger dataset:
Fine-Tuning



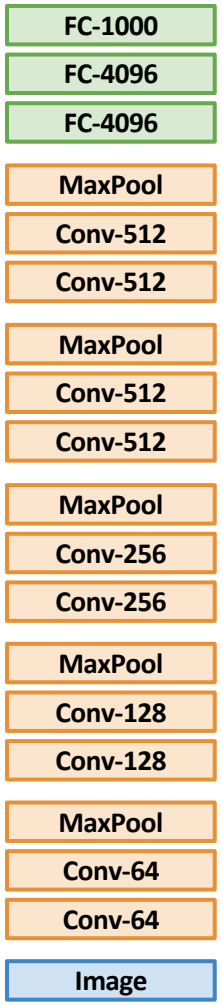
Continue training CNN for new task!

Some tricks:

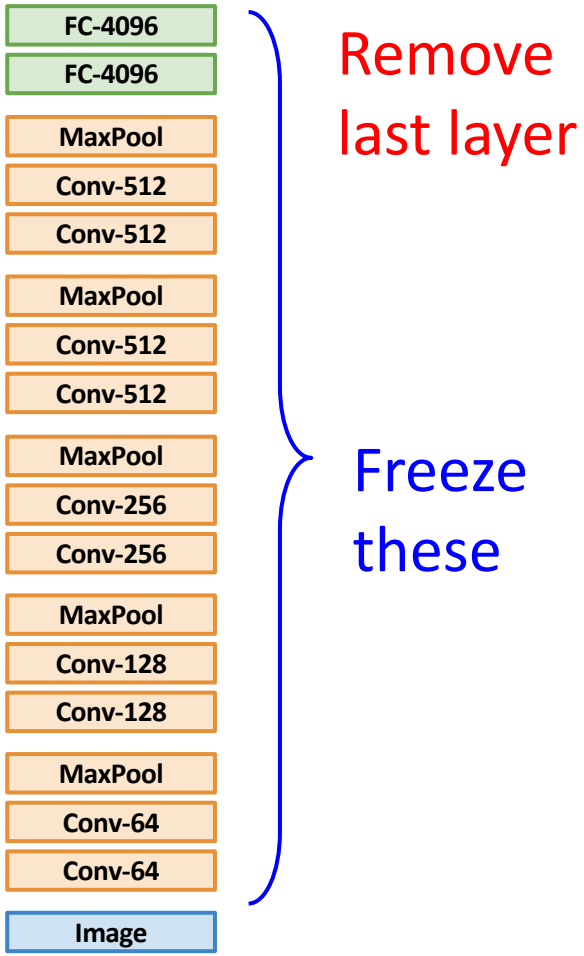
- Train with feature extraction first before fine-tuning
- Lower the learning rate: use ~1/10 of LR used in original training
- Sometimes freeze lower layers to save computation

Transfer Learning with CNNs

1. Train on Imagenet



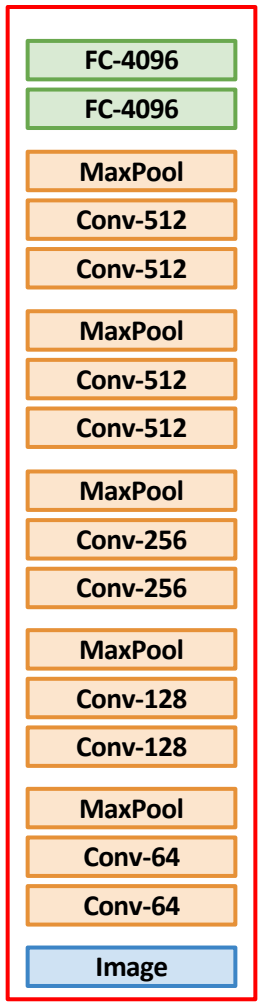
2. Use CNN as a feature extractor



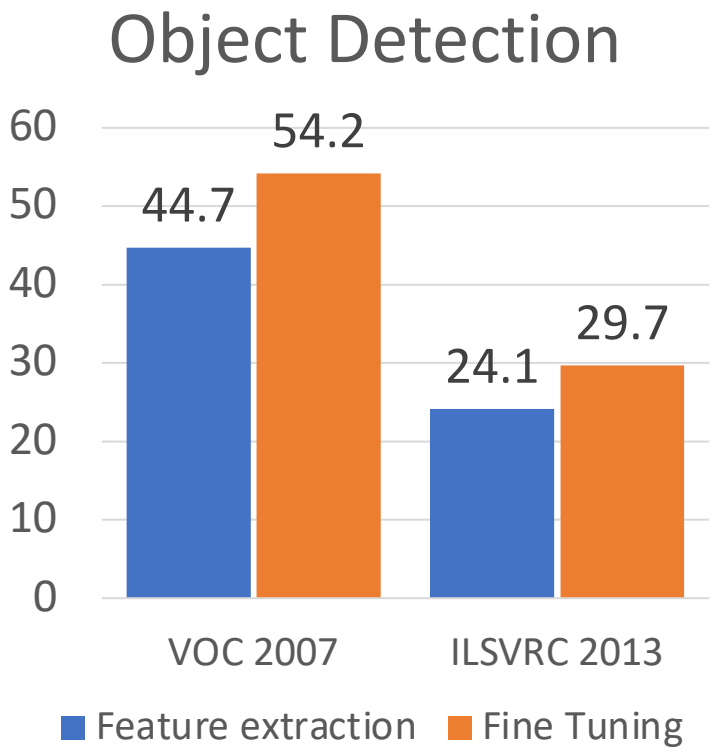
Remove last layer

Freeze these

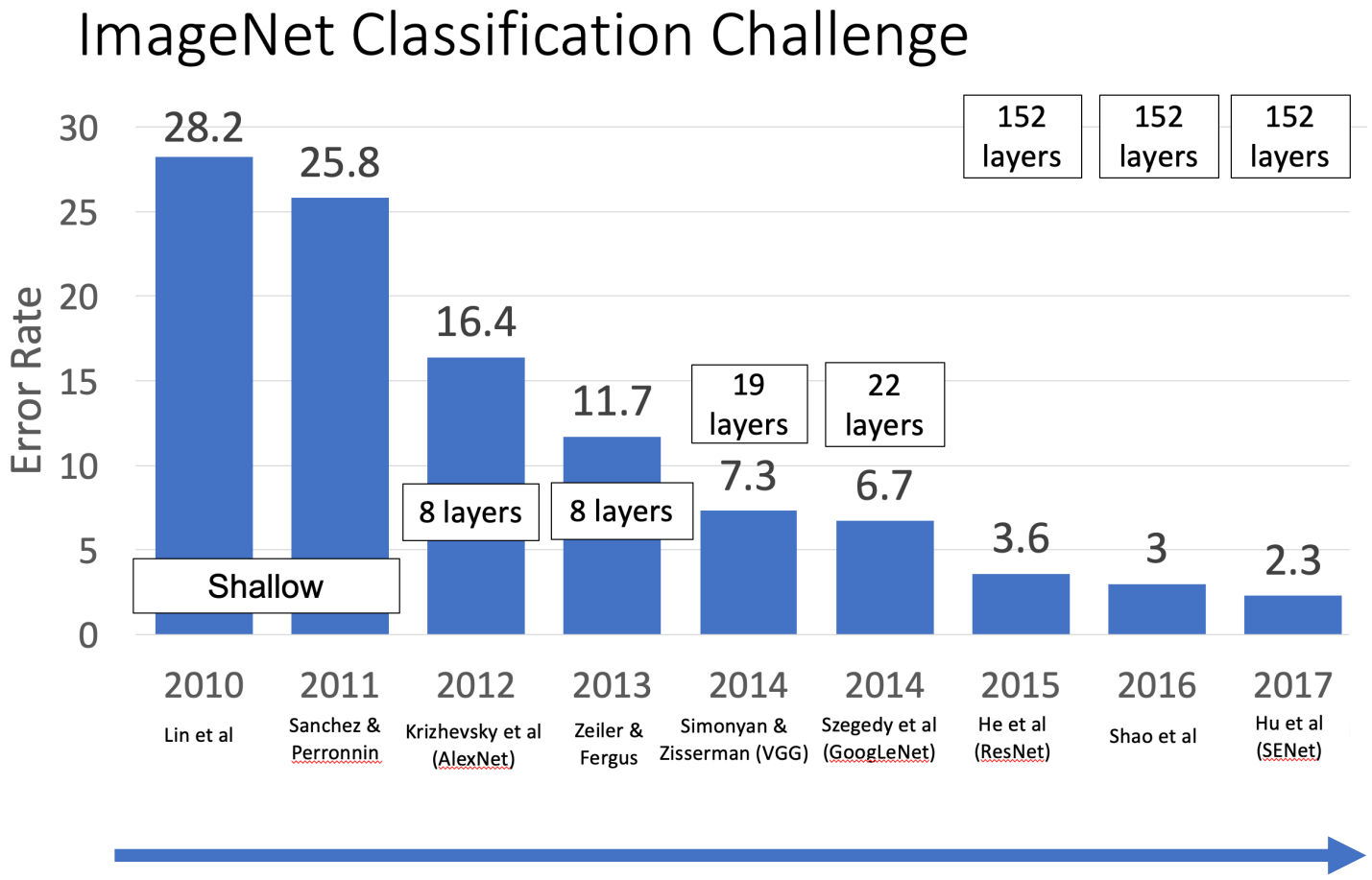
3. Bigger dataset:
Fine-Tuning



Continue training CNN for new task!

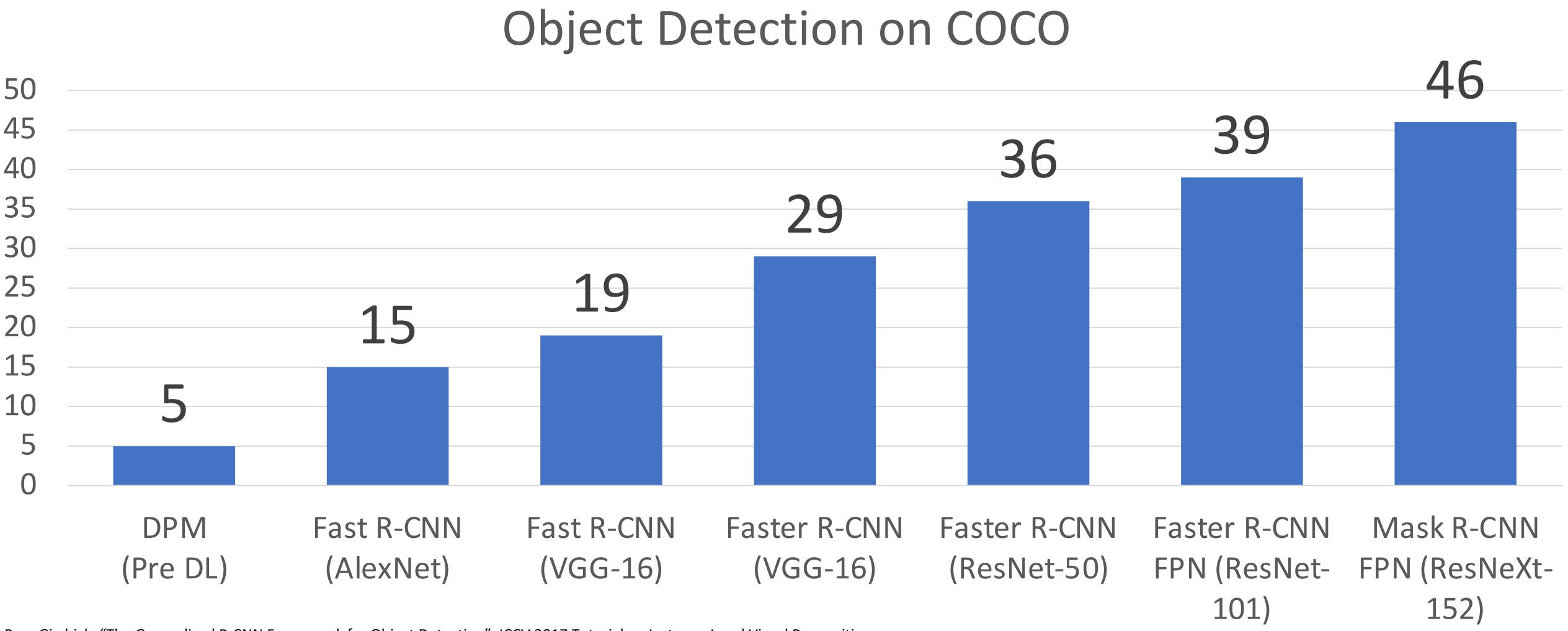


Transfer Learning with CNNs: Architecture Matters!



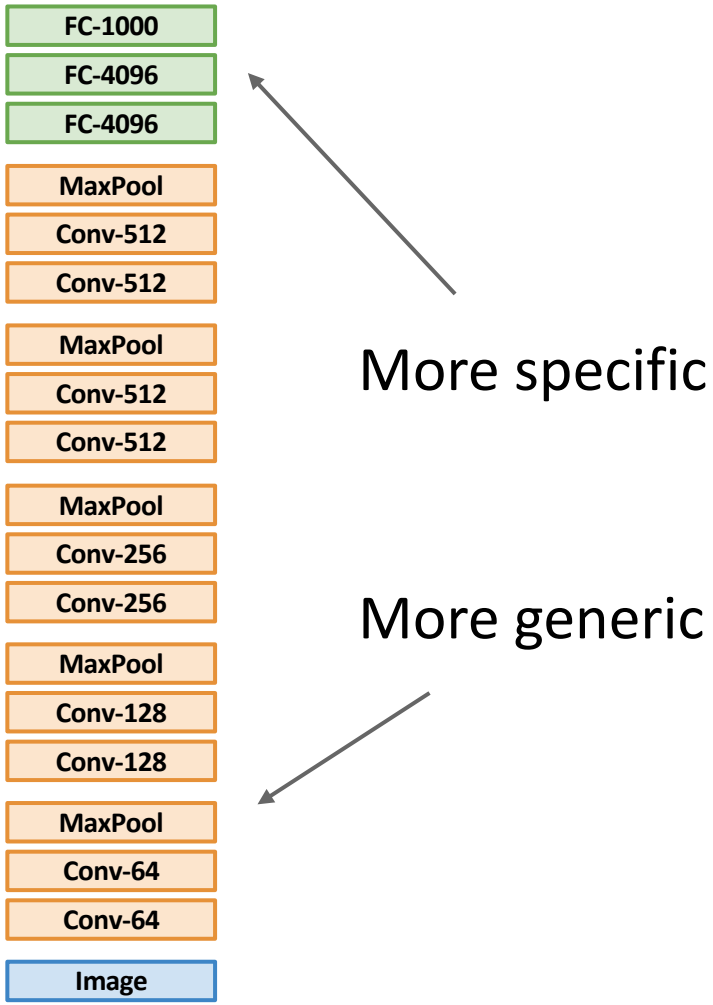
Improvements in CNN architectures lead to improvements in many downstream tasks thanks to transfer learning!

Transfer Learning with CNNs: Architecture Matters!



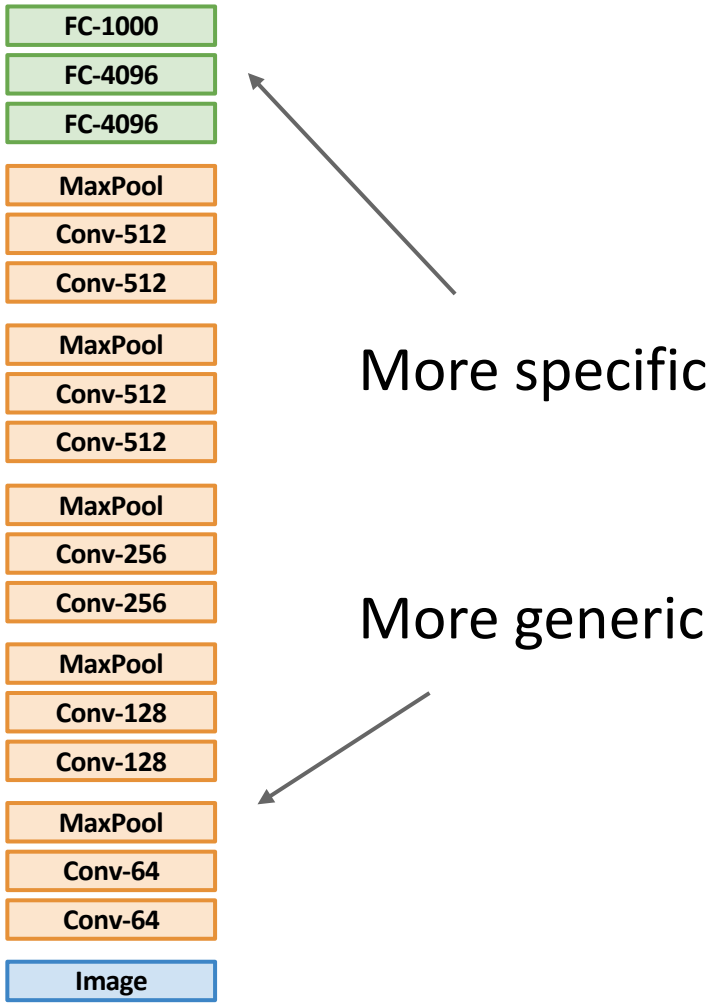
Ross Girshick, "The Generalized R-CNN Framework for Object Detection", ICCV 2017 Tutorial on Instance-Level Visual Recognition

Transfer Learning with CNNs



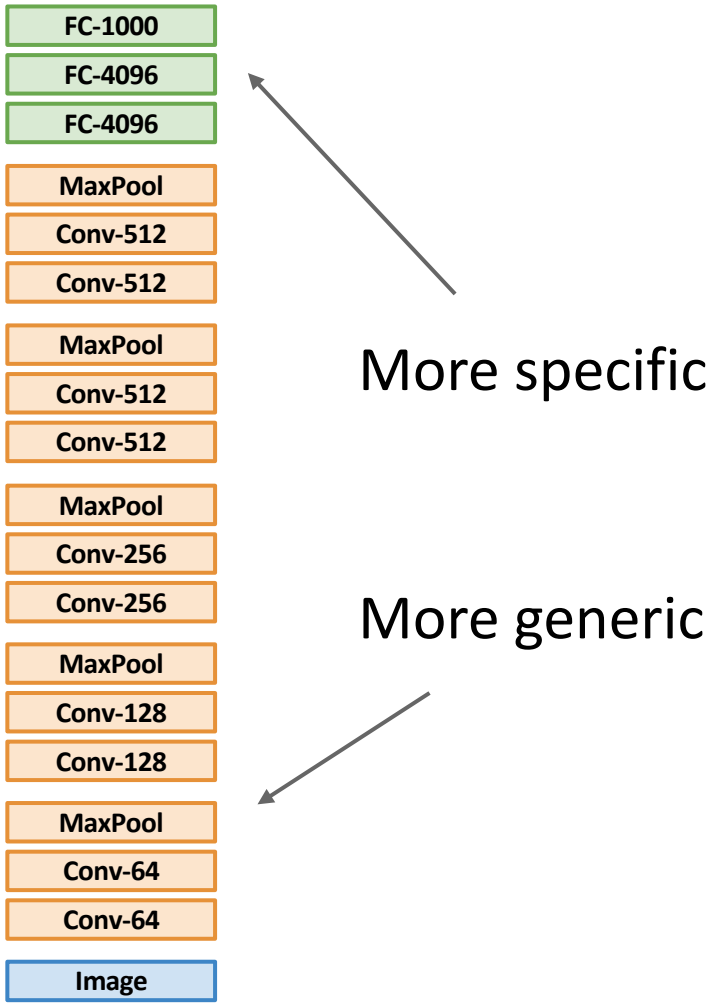
	Dataset similar to ImageNet	**Dataset very different from ImageNet**
very little data (10s to 100s)	?	?
quite a lot of data (100s to 1000s)	?	?

Transfer Learning with CNNs



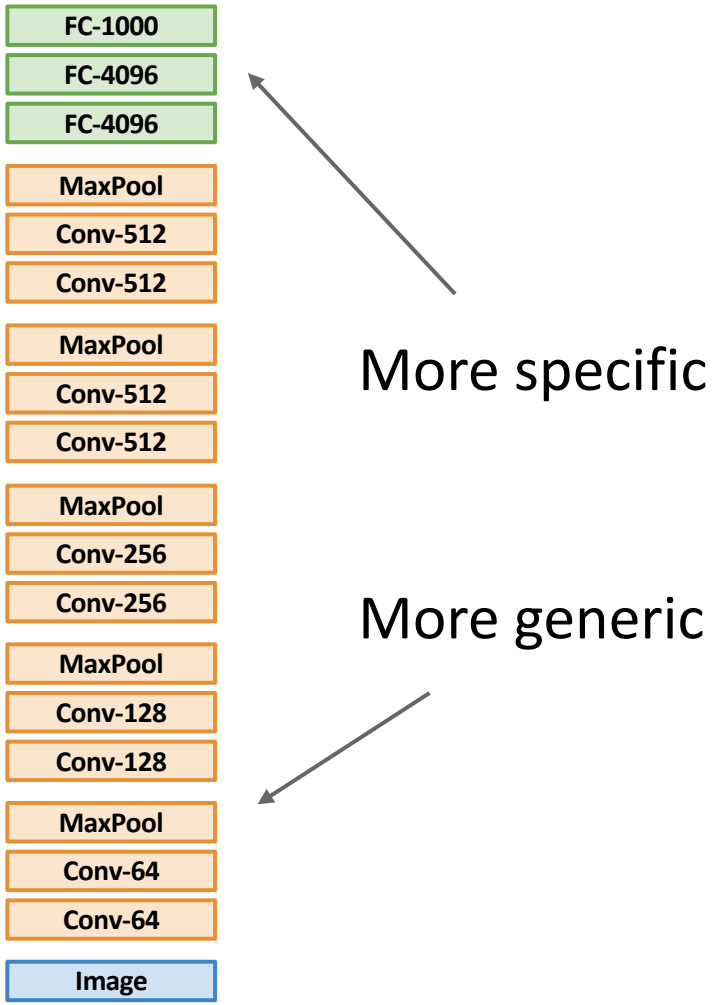
	Dataset similar to ImageNet	**Dataset very different from ImageNet**
very little data (10s to 100s)	Use Linear Classifier on top layer	?
quite a lot of data (100s to 1000s)	Finetune a few layers	?

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	?
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

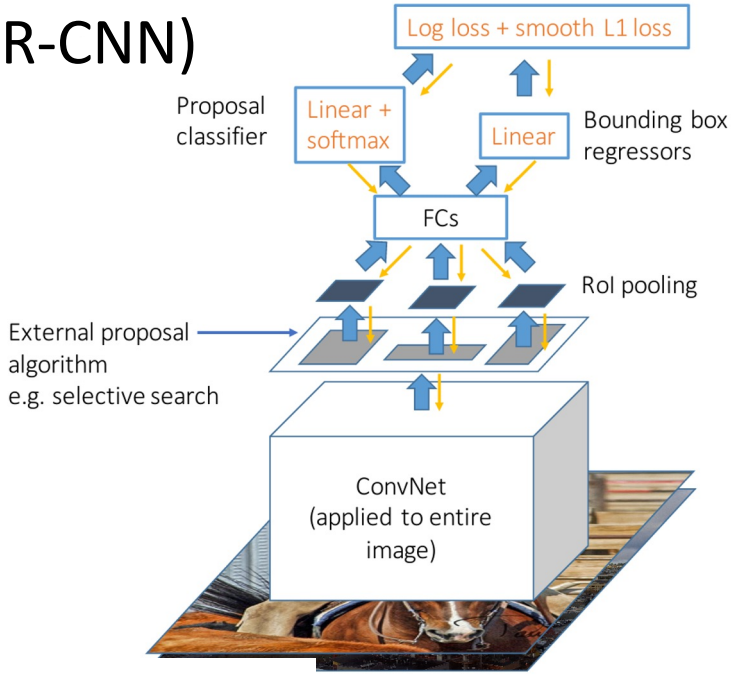
Transfer Learning with CNNs

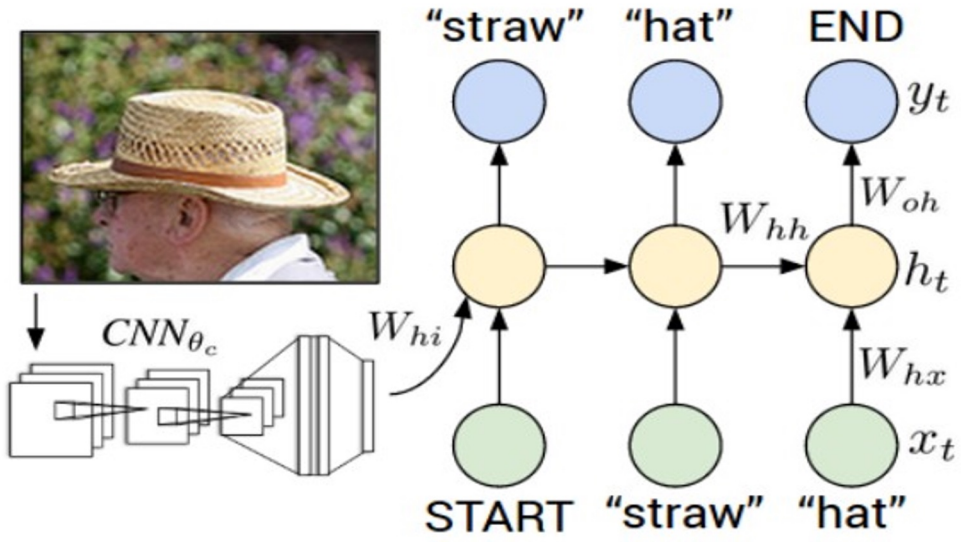


	Dataset similar to ImageNet	**Dataset very different from ImageNet**
very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different layers
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

Transfer learning is pervasive!

Object Detection (Fast R-CNN)





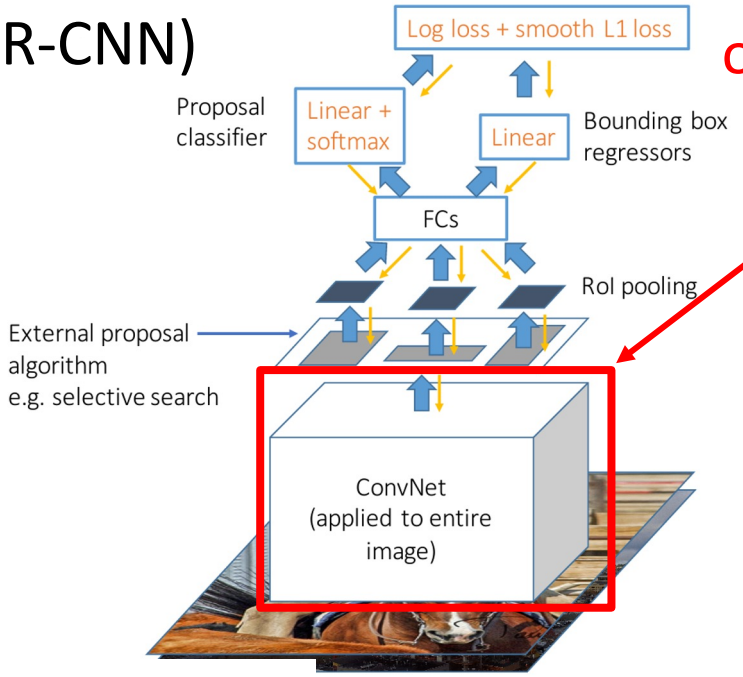
Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

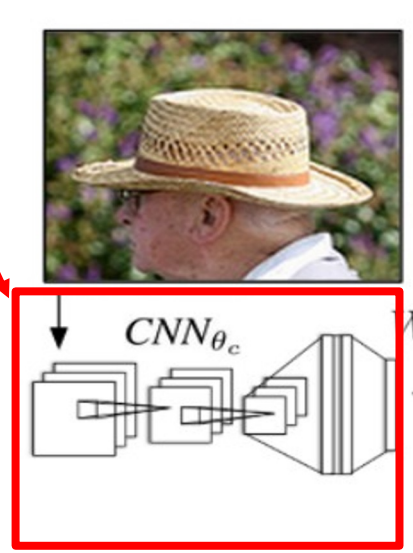
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

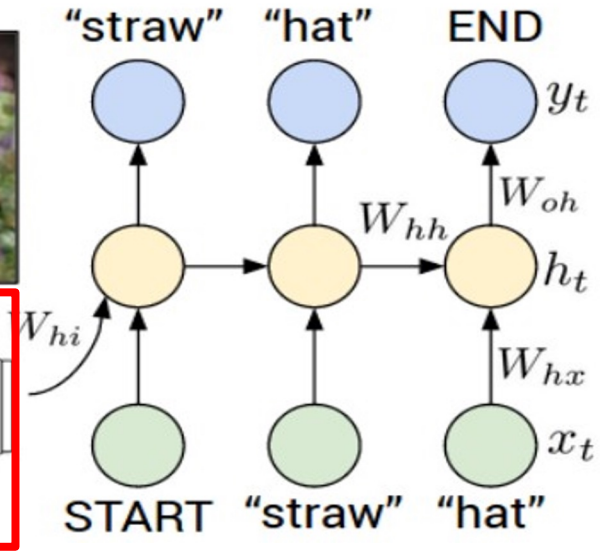
Transfer learning is pervasive!

Object Detection (Fast R-CNN)

CNN pretrained on ImageNet





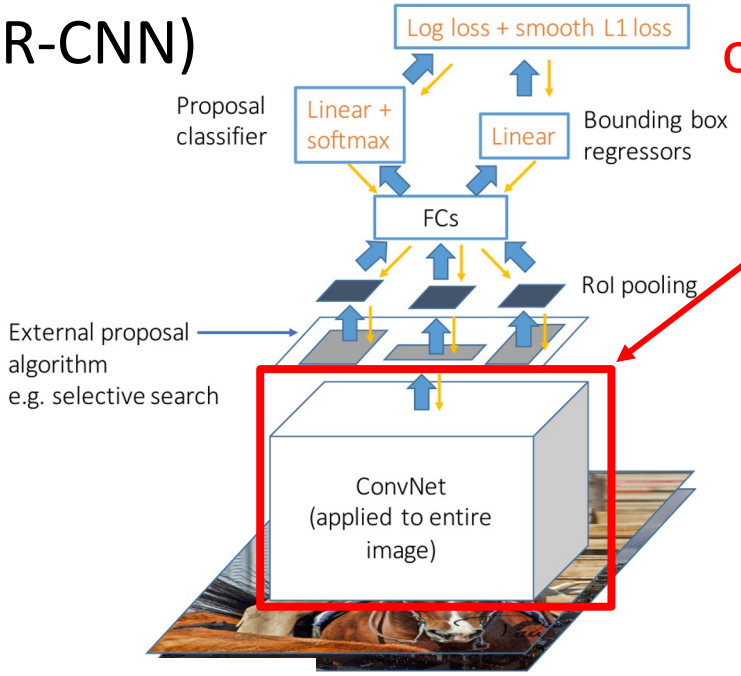


Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

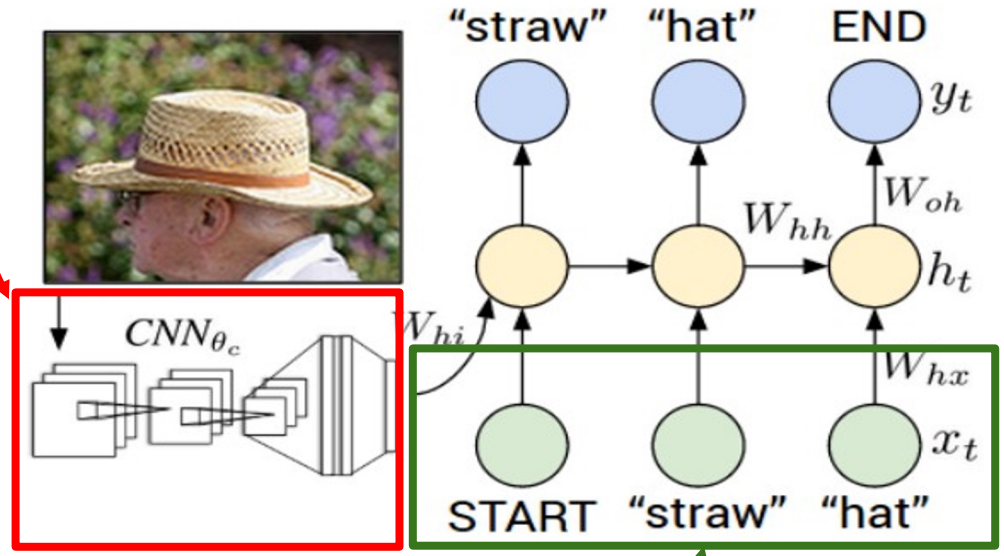
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning is pervasive!

Object Detection (Fast R-CNN)



CNN pretrained on ImageNet



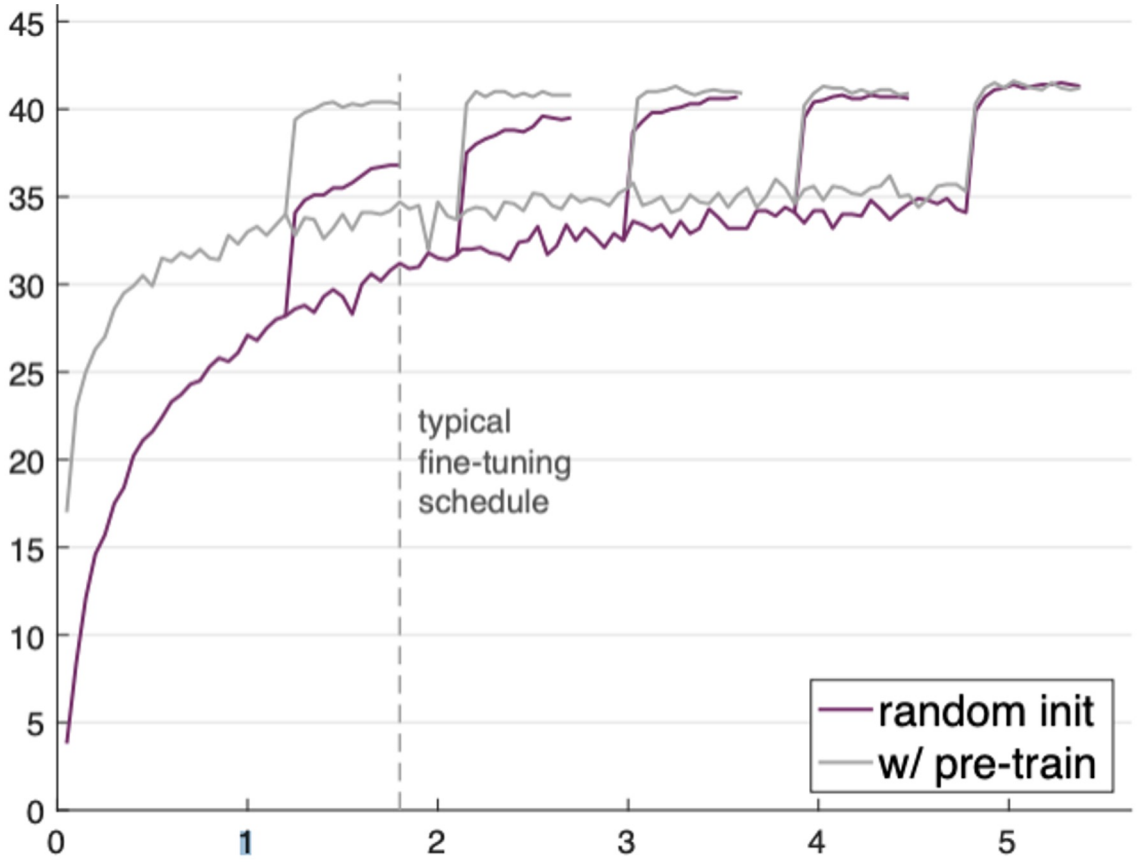
Word vectors pretrained with word2vec

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!
But Kaiming has questioned it

COCO object detection



Training from scratch can work as well as pretraining on ImageNet!

… If you train for 3x as long

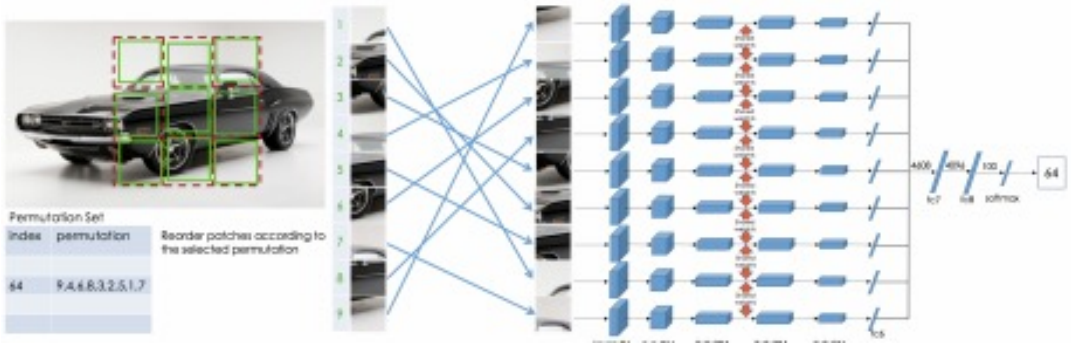
He et al, "Rethinking ImageNet Pre-Training", ICCV 2019

Maybe before training on ImageNet, we pretrain the network on some other (ultra-large) dataset???

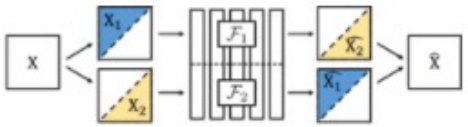
Not enough labels or people or money to annotate images…
How about training network on (ultra-large) unlabeled data (like any images on Instagram or web)?

Unsupervised Representation Learning: Training CNN without image labels.

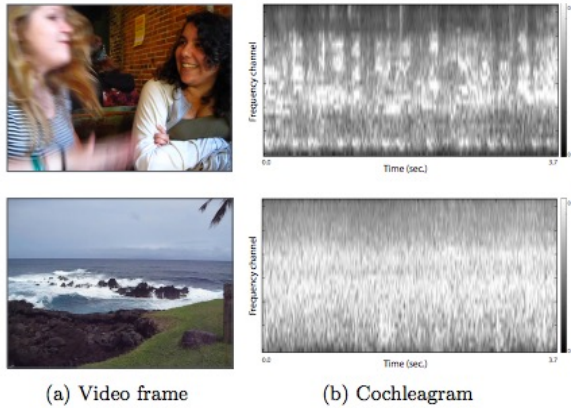
Context prediction, ICCV'15



Solving puzzle, ECCV'16



Colorization, ECCV'16 and CVPR'17



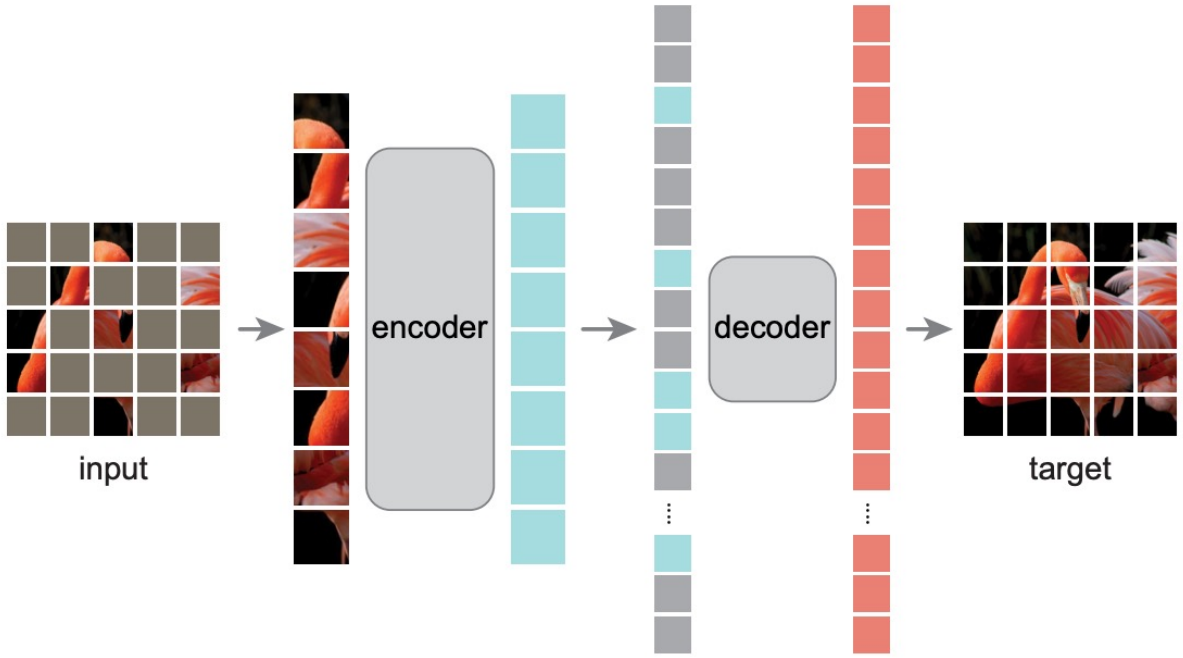
Audio prediction, ECCV'16

Unsupervised learning remains as a hot topic

[Submitted on 11 Nov 2021 (v1), last revised 19 Dec 2021 (this version, v3)]

Masked Autoencoders Are Scalable Vision Learners

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, Ross Girshick



https://arxiv.org/pdf/2111.06377.pdf

Webly supervised learning

Can we treat ImageNet feature as lower bound?

Google's JET-300M data

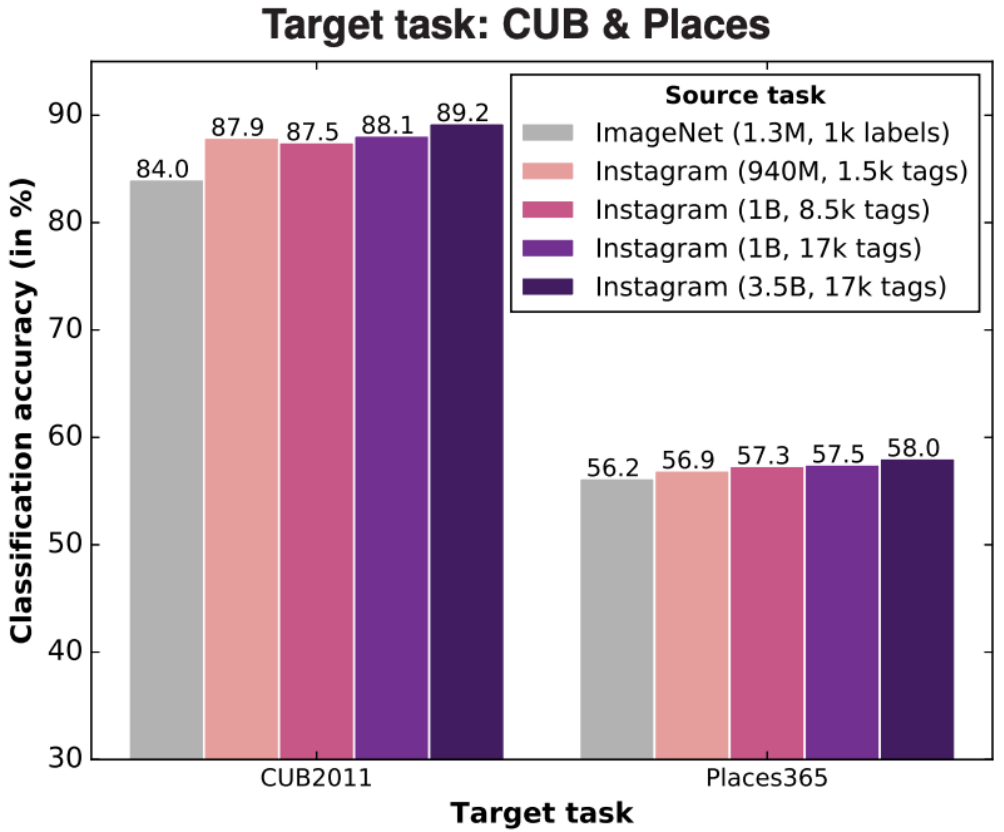
Initialization	Top-1 Acc.	Top-5 Acc.
MSRA checkpoint [16]	76.4	92.9
Random initialization	77.5	93.9
Fine-tune from JFT-300M	**79.2**	**94.7**

Table 1. Top-1 and top-5 classification accuracy on the ImageNet 'val' set (single model and single crop inference are used).

Method	mAP@0.5	mAP@[0.5,0.95]
He *et al.* [16]	53.3	32.2
ImageNet	53.6	34.3
300M	56.9	36.7
ImageNet+300M	**58.0**	**37.4**
Inception ResNet [38]	56.3	35.5

Revisiting Unreasonable Effectiveness of Data in Deep Learning Era. Chen Sun et al. ICCV'17. Google

Facebook's Instagram hashtag 1B



Exploring the Limits of Weakly Supervised Pretraining. D.Hahajan et al. ECCV'18.

Recap

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

Next Time: Visualization and Understanding Neural Networks