21. Training Neural Networks

STA3142 Statistical Machine Learning

Kibok Lee

Assistant Professor of

Applied Statistics / Statistics and Data Science

May {28, 30}, 2024

** Slides adapted from EECS498/598 @ Univ. of Michigan by Justin Johnson*

연세대학교
YONSEI UNIVERSITY

Assignment 5 (Final Exam Replacement)

- Due **Friday 6/14, 11:59pm**
- Topic: Convolutional Neural Networks
  - Derive gradients for NN layers
  - Implement layers for CNNs
  - Train a CNN classifier for MNIST digit recognition
- Please read the instruction carefully!
  - Submit one pdf and one zip file separately
  - Write your code only in the designated spaces
  - Do not import additional libraries
  - …
- If you feel difficult, consider to take option 2.

Overview

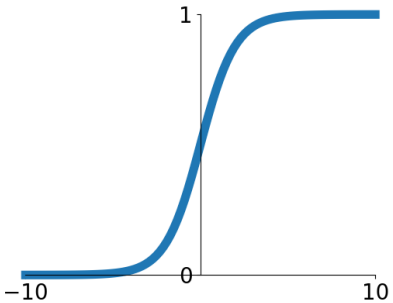
- Activation functions
- Data preprocessing
- Weight initialization
- Regularization
- Optimization
- Hyperparameter optimization
- Transfer learning

Activation Functions

Activation Functions

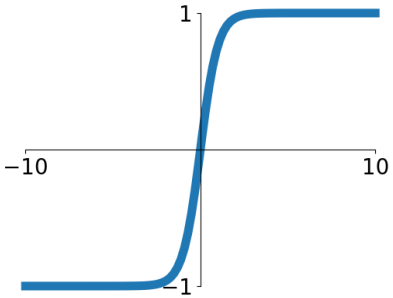
Sigmoid

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



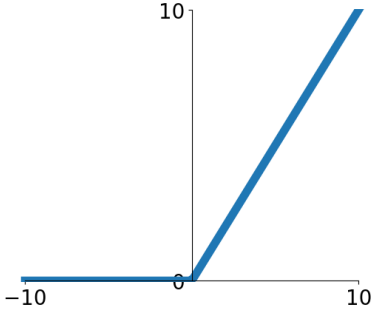
tanh

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



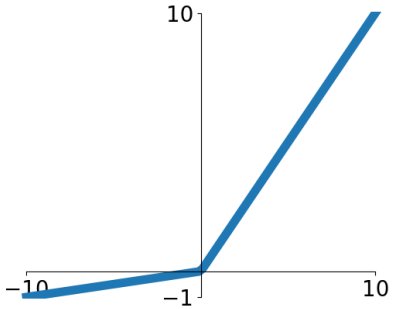
ReLU

$$\max(0, x)$$



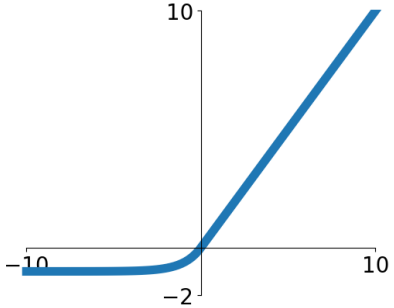
Leaky ReLU

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



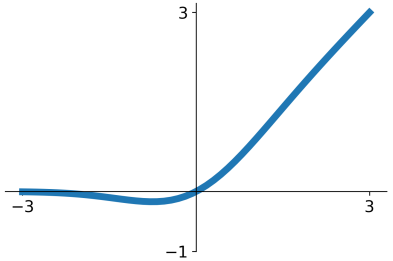
ELU

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

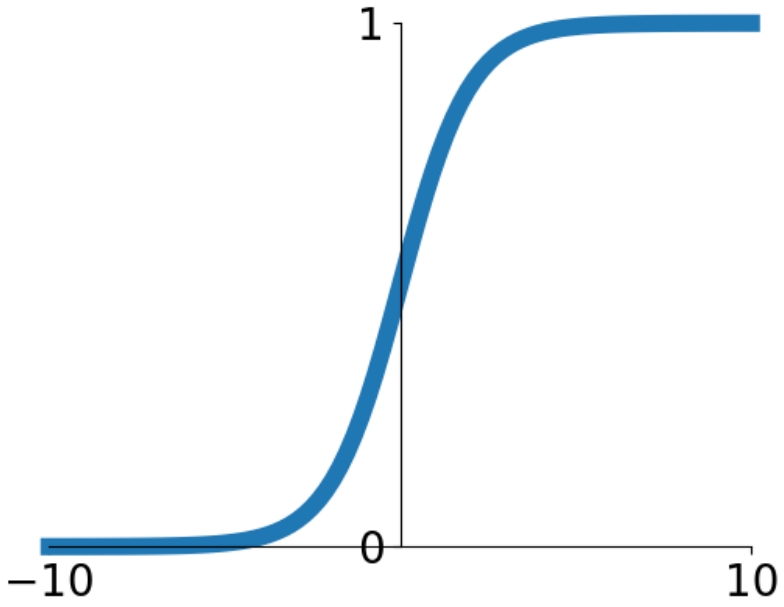


GELU

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



Activation Functions: Sigmoid



Sigmoid

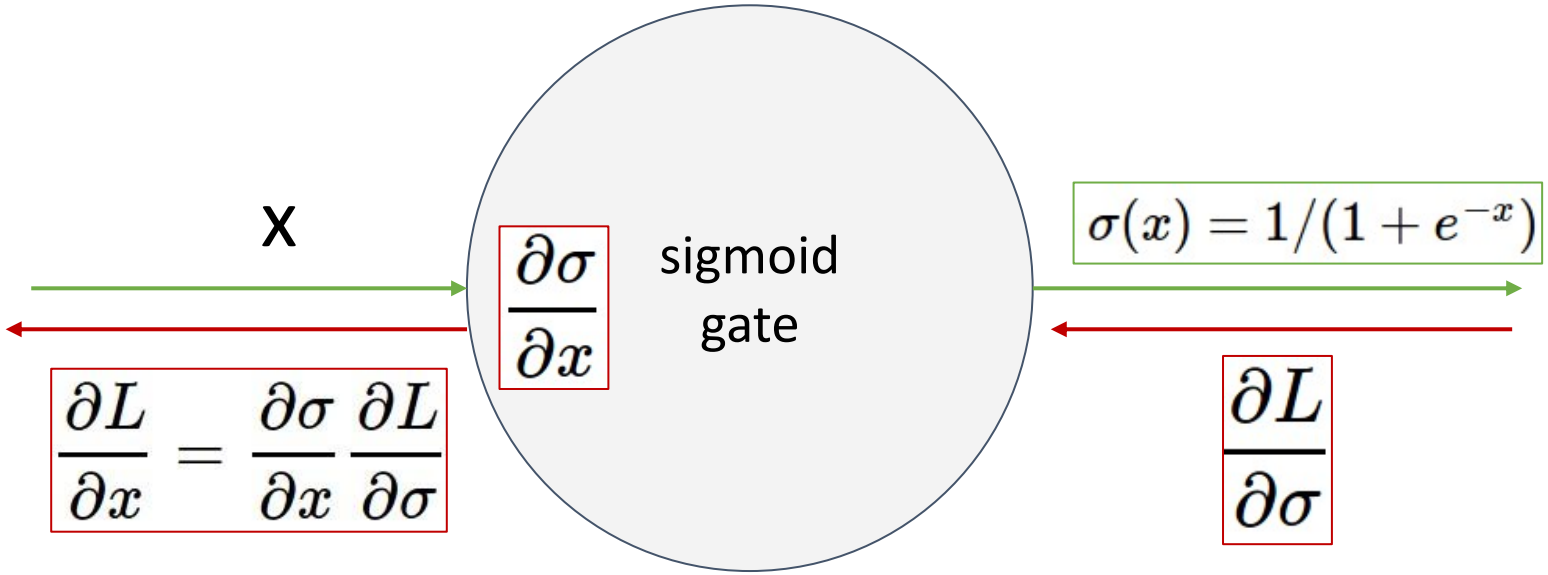
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

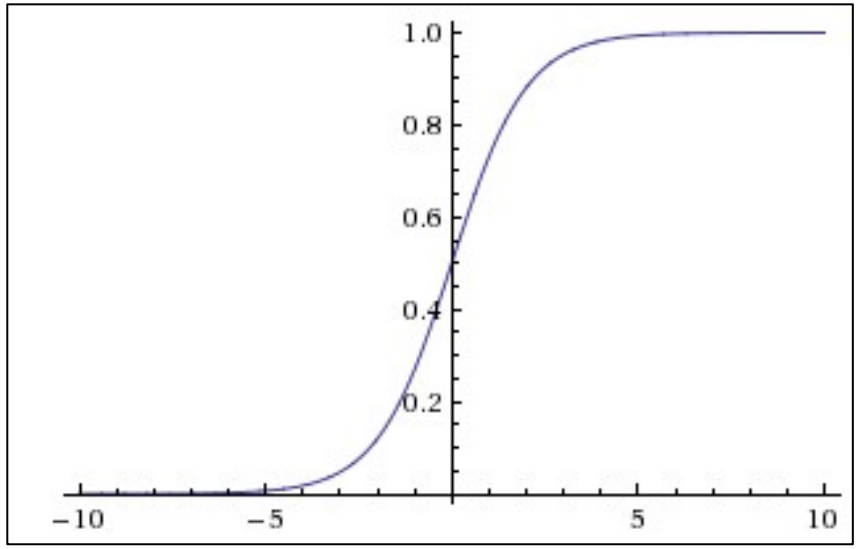
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients

Activation Functions: Sigmoid

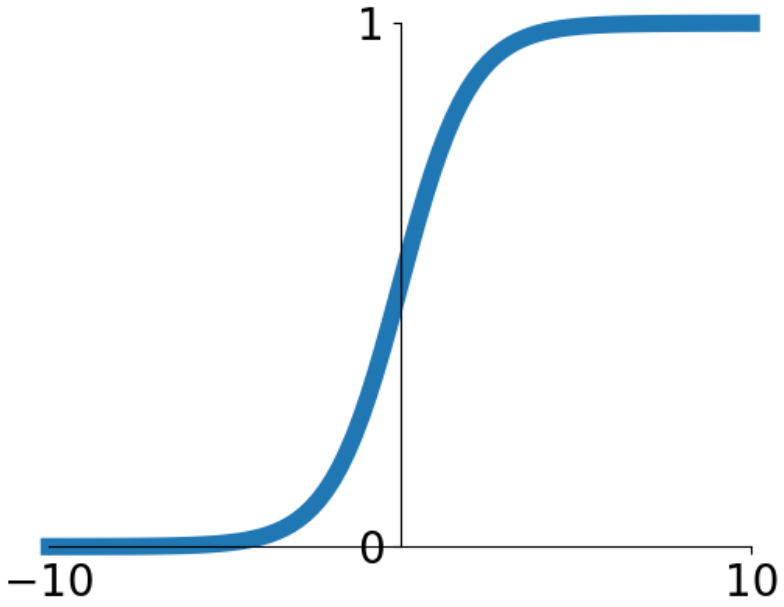




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Recall: $\frac{\partial}{\partial x}\sigma(x) = (1-\sigma(x))\sigma(x)$

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 $w^{(\ell)}$?

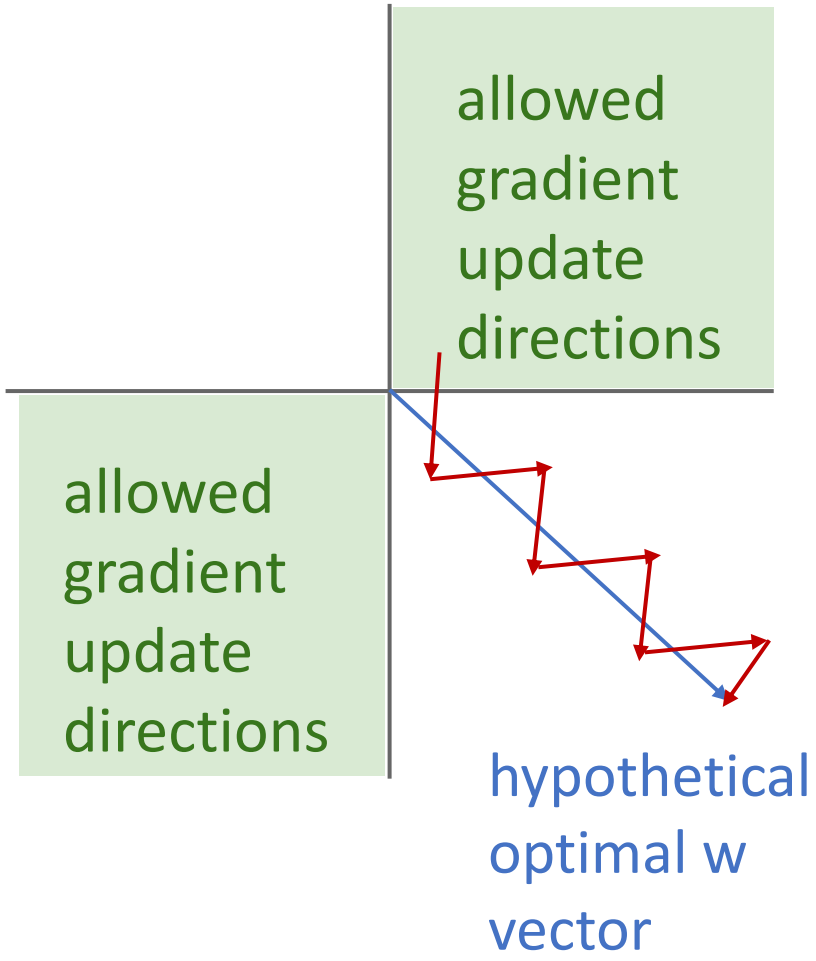
Gradients on all $w_{i,j}^{(\ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

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 $\partial L / \partial h_i^{(\ell)}$

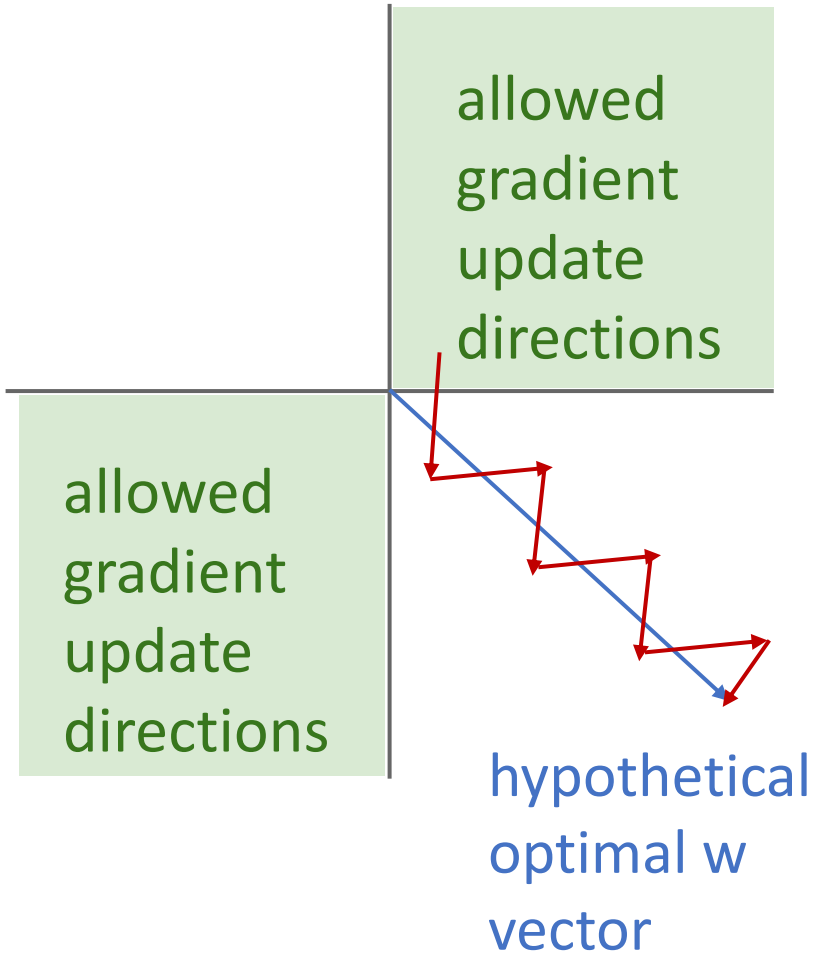
Gradients on rows of w can only point in some directions; needs to "zigzag" to move in other directions

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

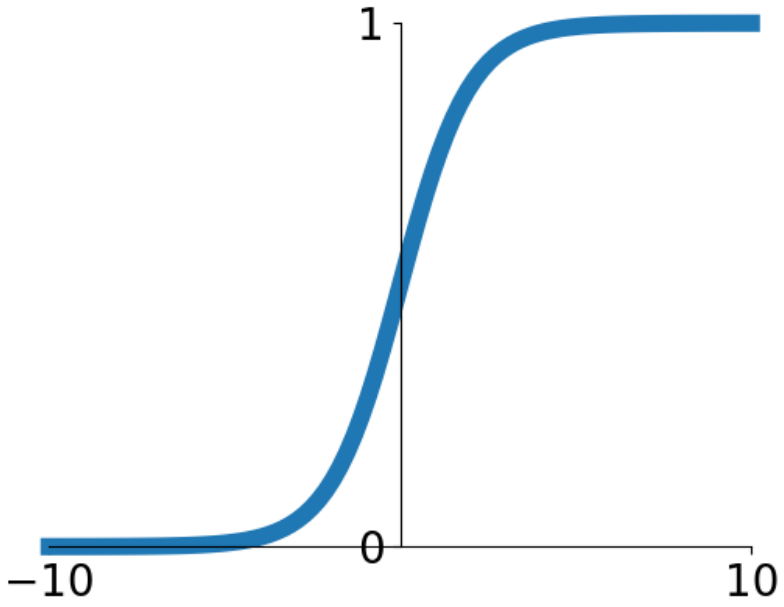


What can we say about the gradients on $w^{(\ell)}$?

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

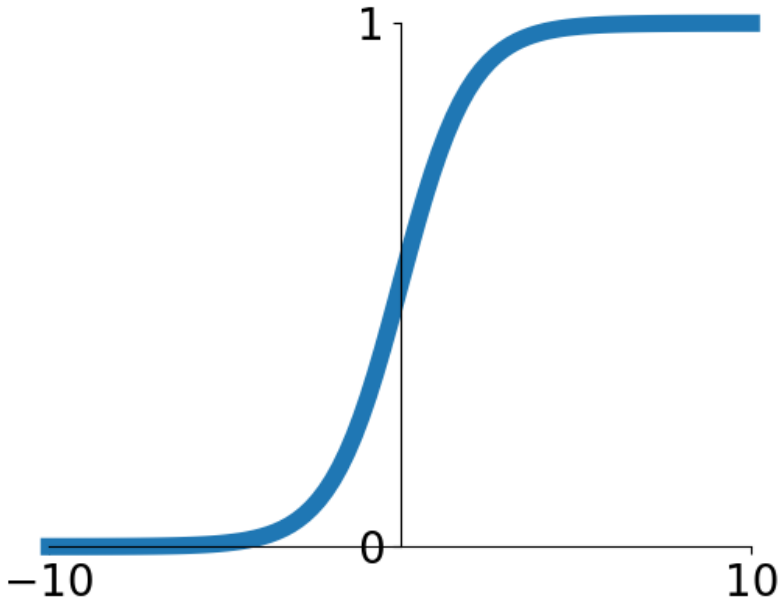
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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

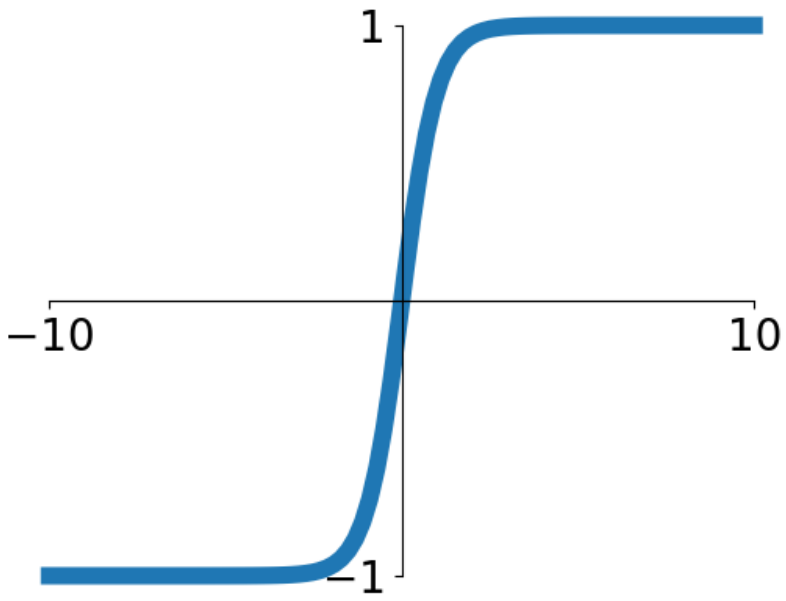
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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



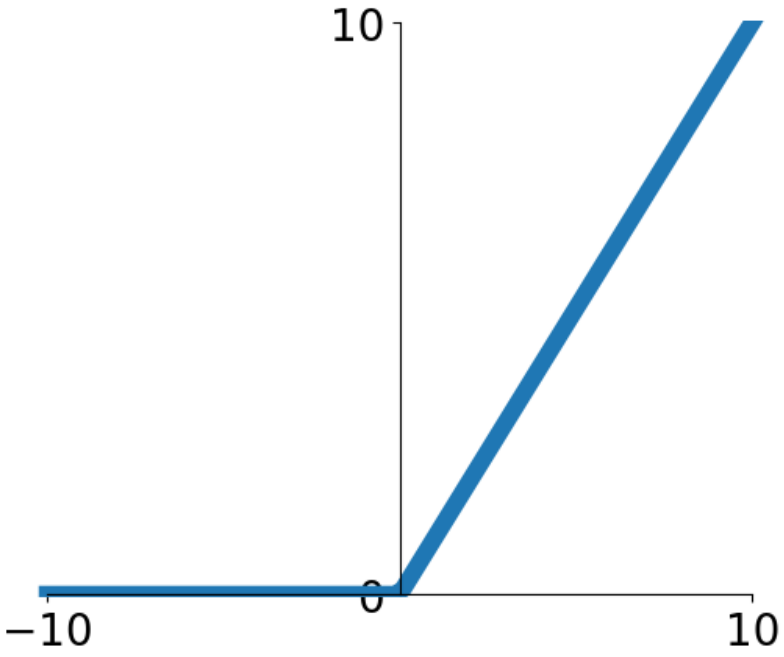
tanh(x)

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

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

Activation Functions: ReLU

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



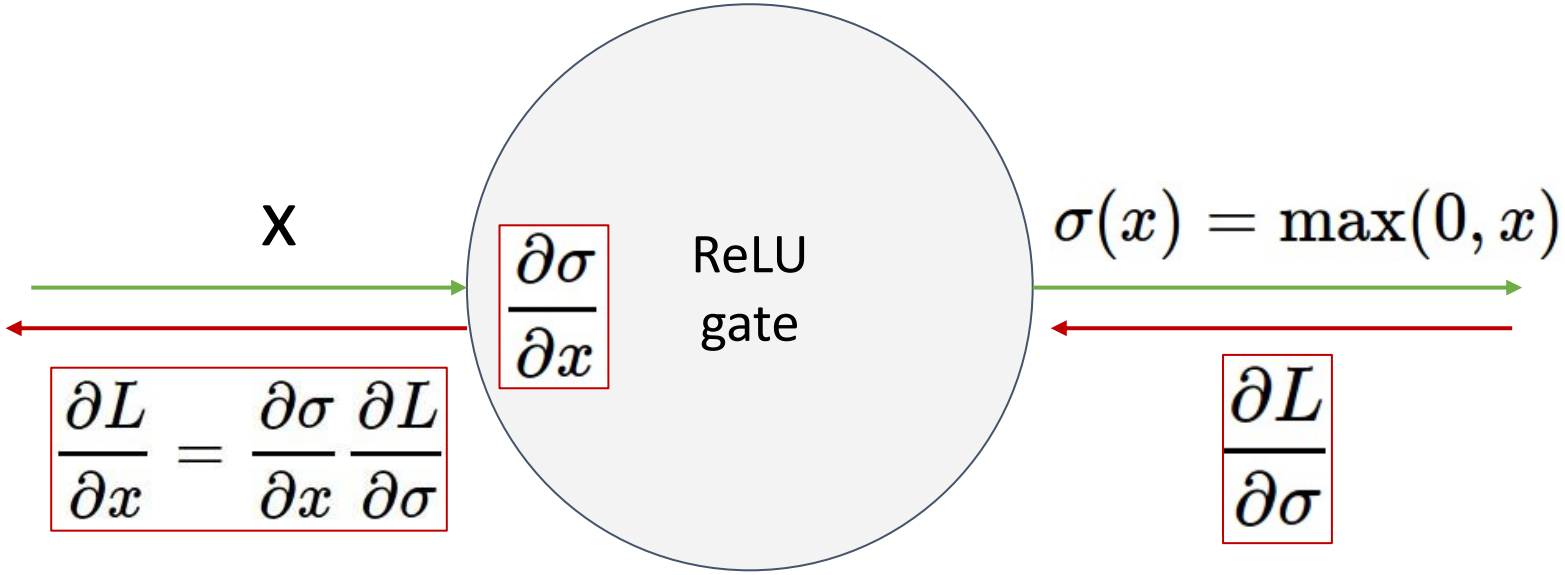
ReLU
(Rectified Linear Unit)

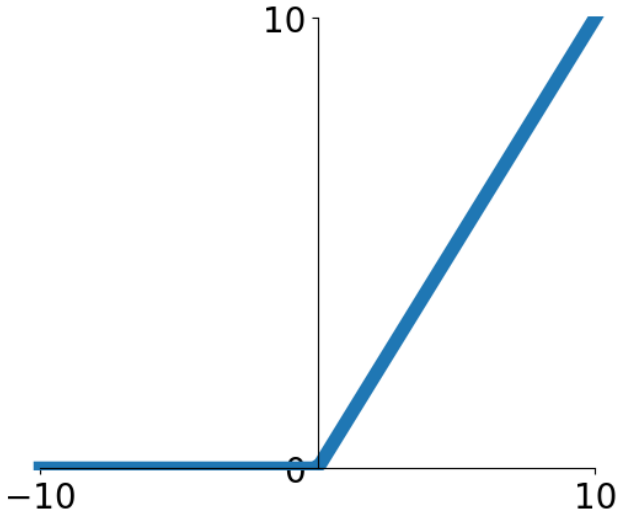
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g., 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

Activation Functions: ReLU

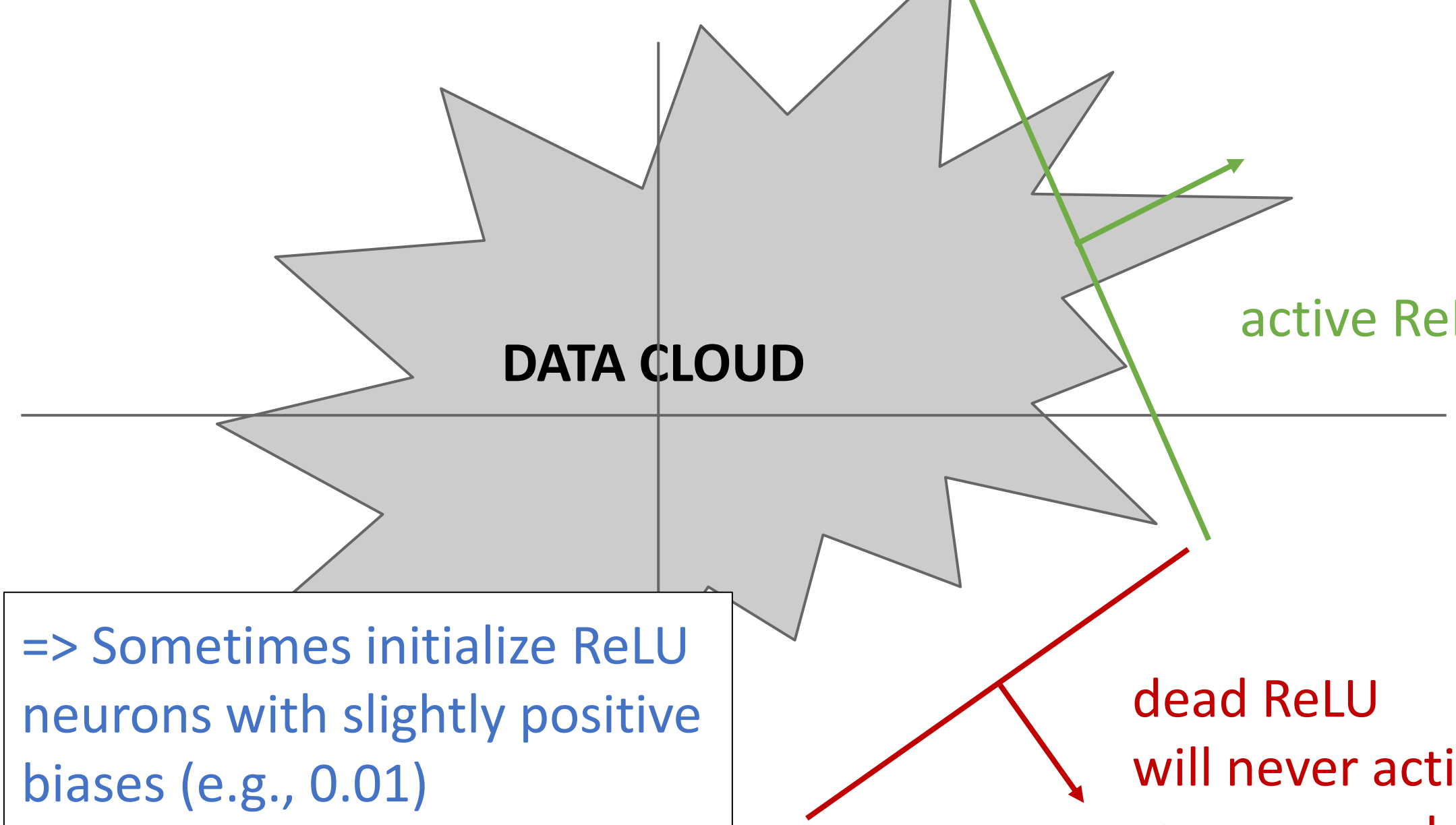




What happens when x = -10?

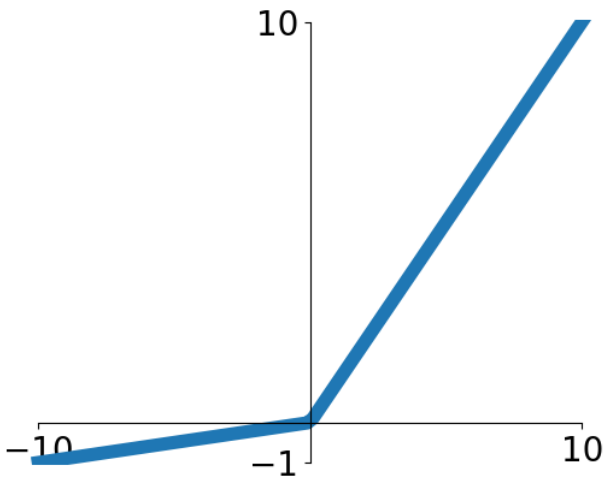
What happens when x = 0?

What happens when x = 10?



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

Activation Functions: Leaky ReLU



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

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

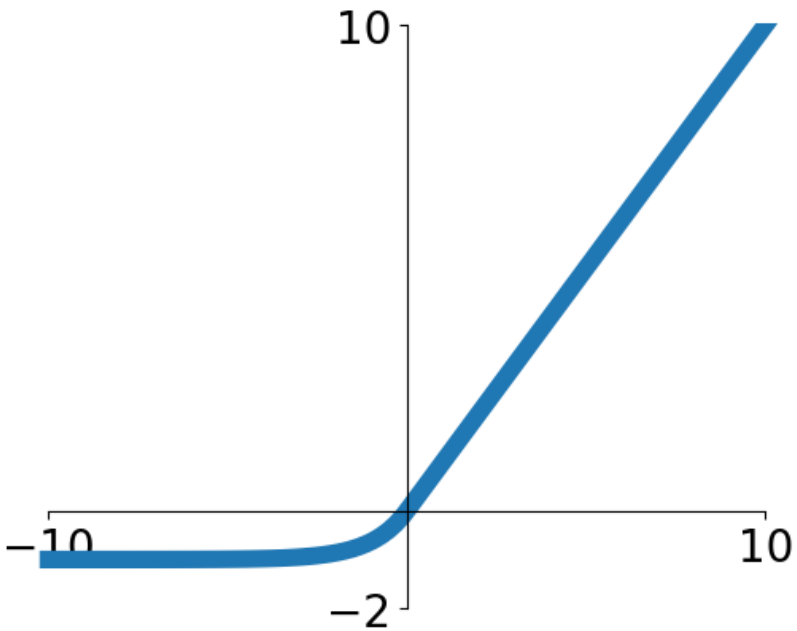
Parametric ReLU (PReLU)

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

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)



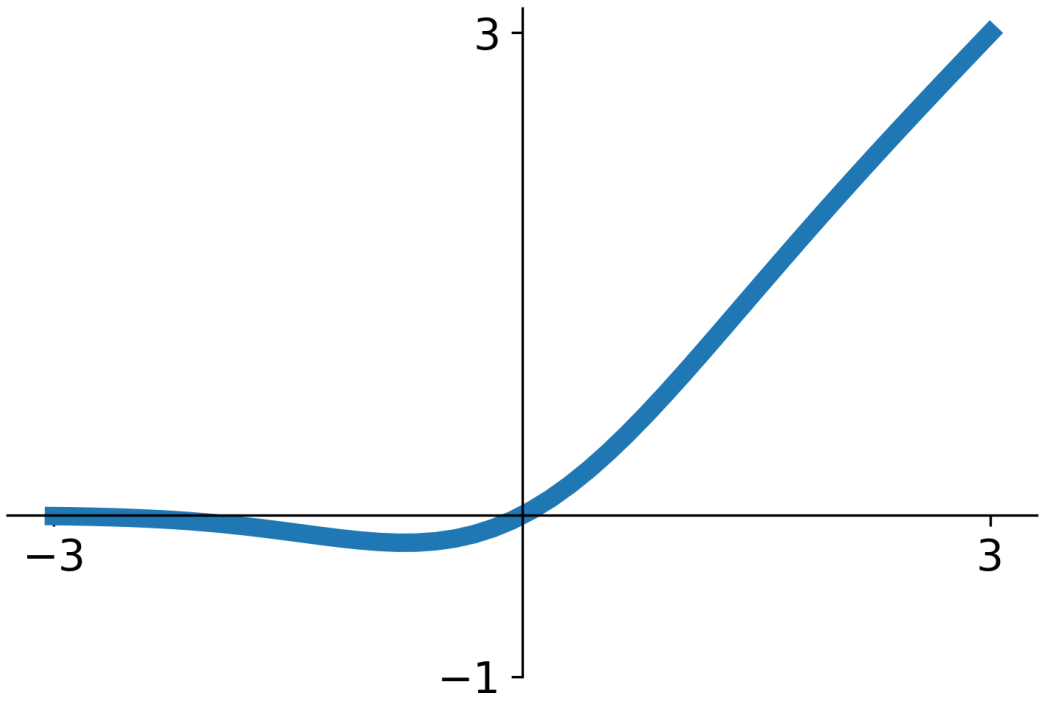
$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & if\ x \leq 0 \end{cases}$$

(Default $\alpha = 1$)

- All benefits of ReLU
- Closer to zero mean outputs
- Can be smooth in any regime (if $\alpha = 1$)

- Computation requires exp()

Activation Functions: Gaussian Error Linear Unit (GELU)



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

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

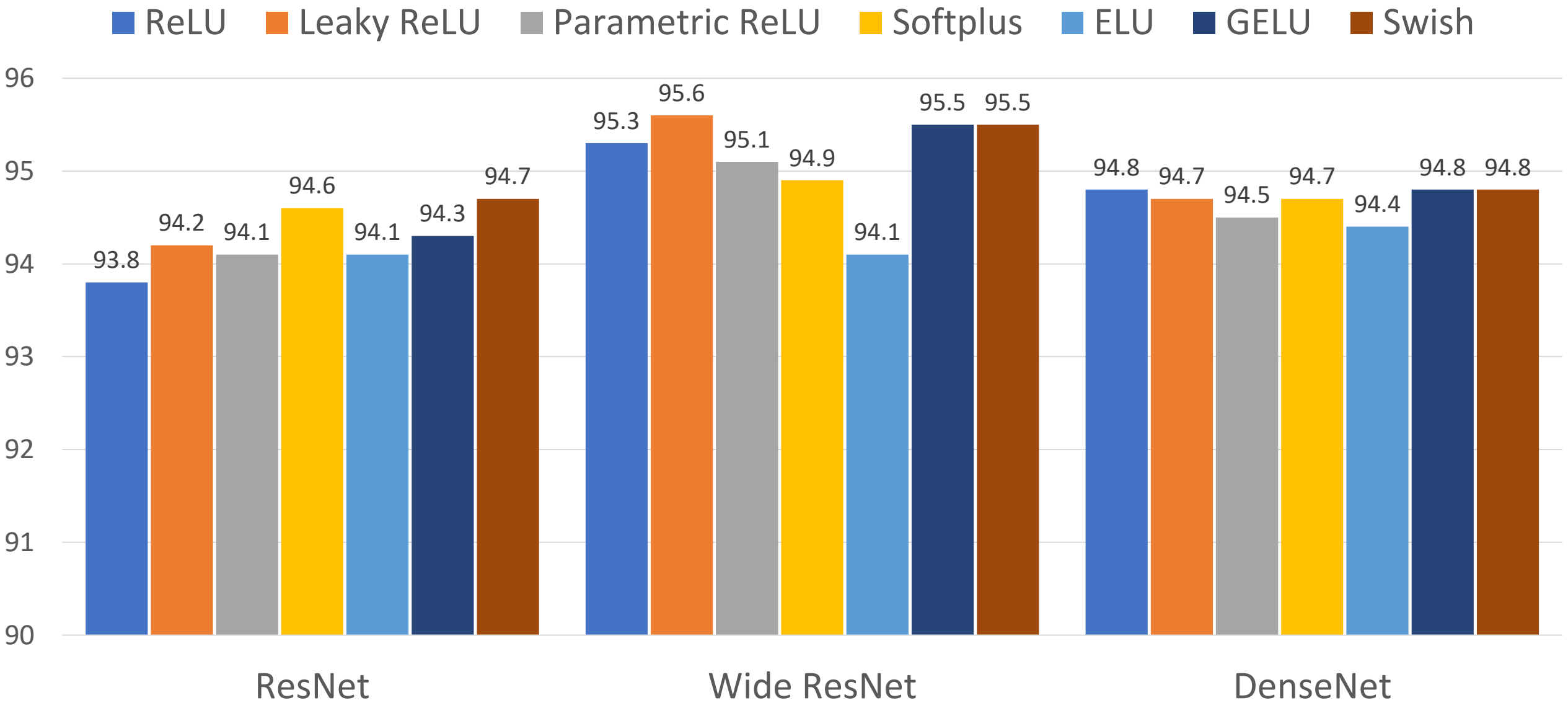
$$\approx x\sigma(1.702x)$$

- **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

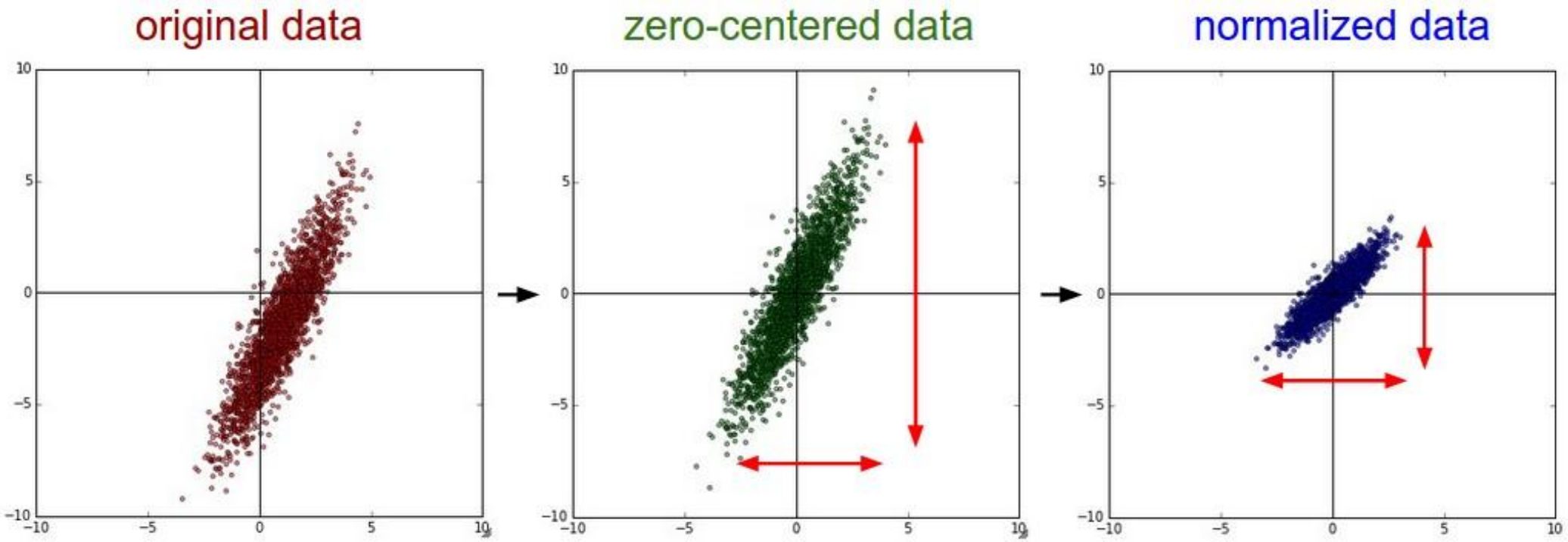
- Don't think too hard. Just use ReLU
- Try out Leaky ReLU / ELU / GELU
  if you need to squeeze that last 0.1%
- Don't use sigmoid or tanh

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



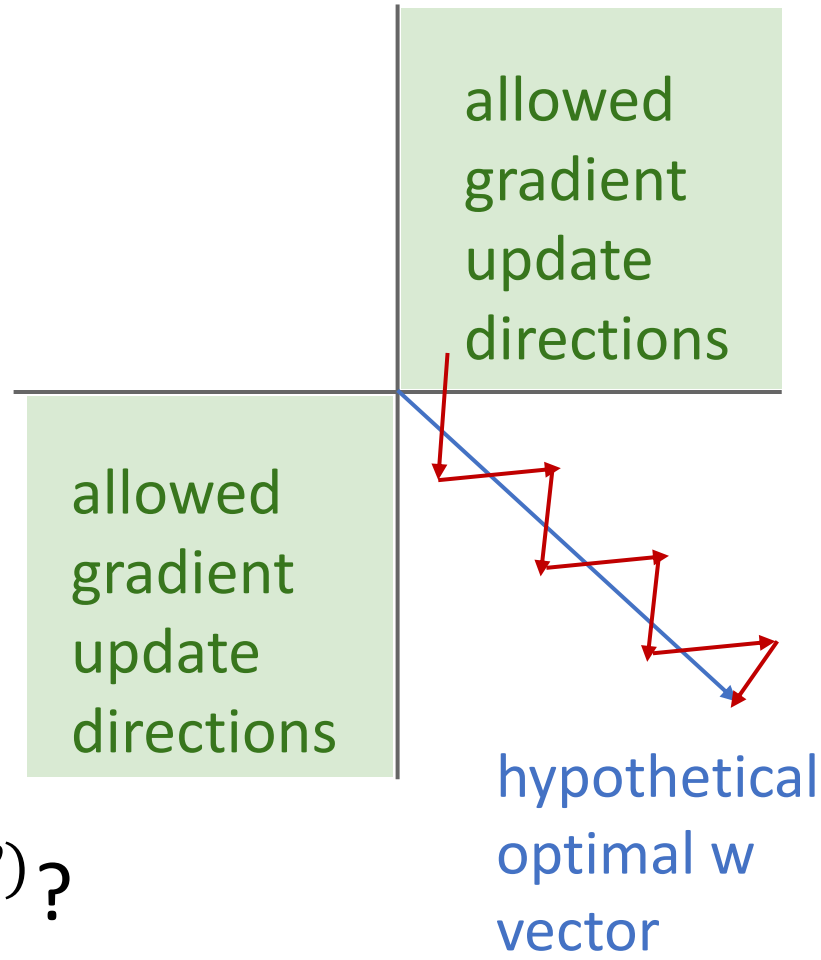

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

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

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

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



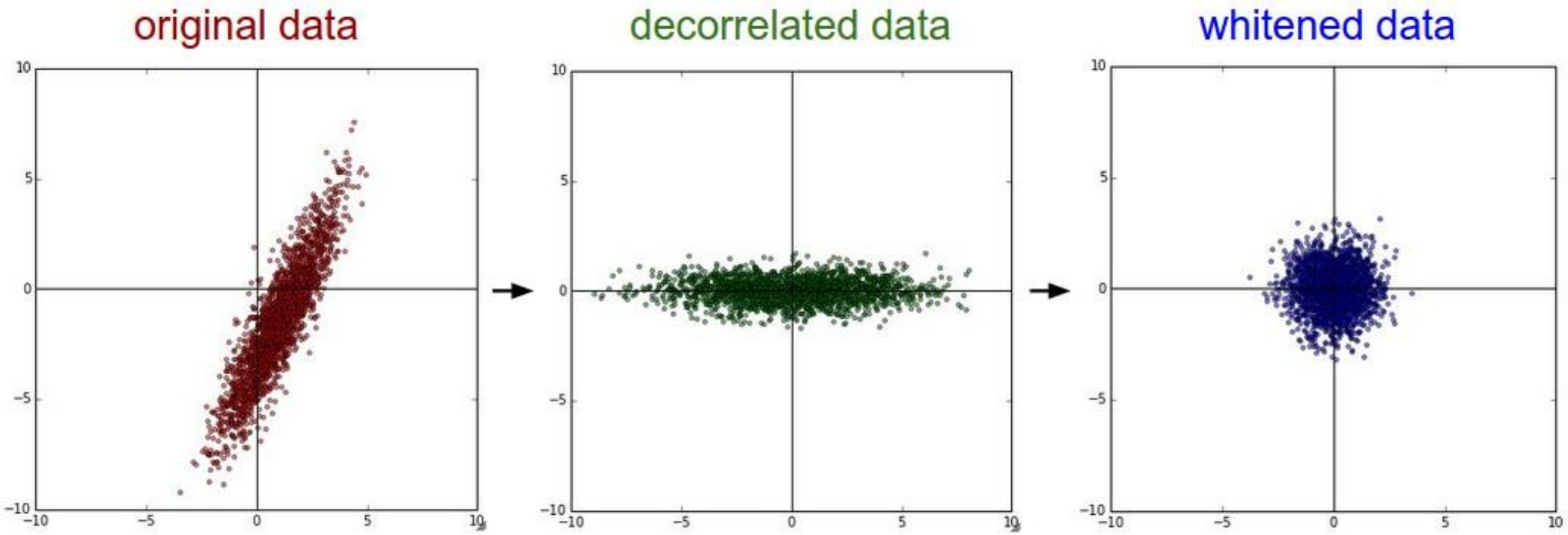
What can we say about the gradients on $w^{(\ell)}$?

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

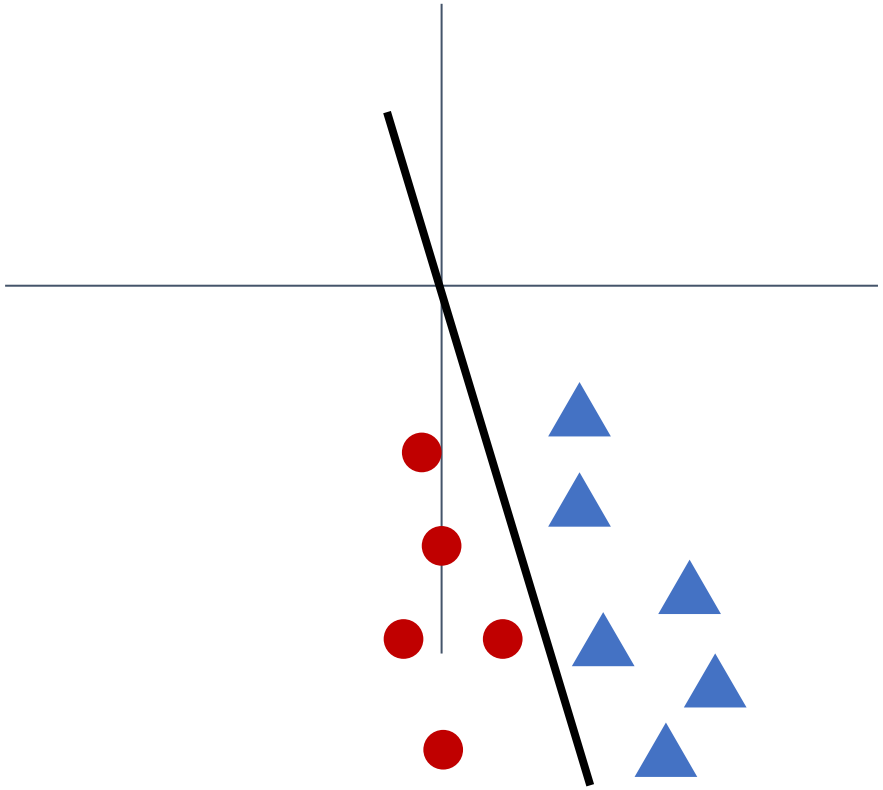


(data has diagonal covariance matrix)

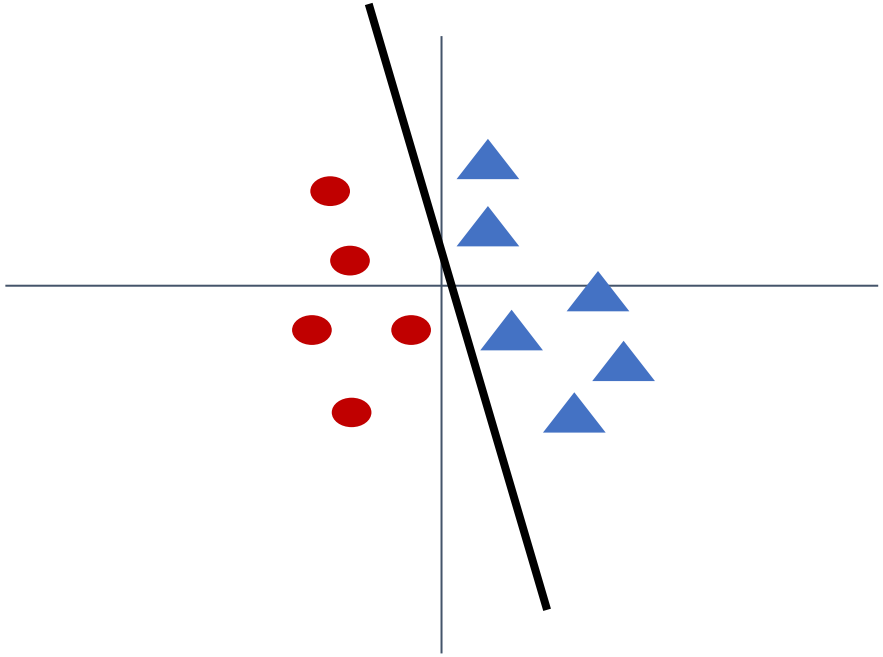
(covariance matrix is the identity matrix)

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

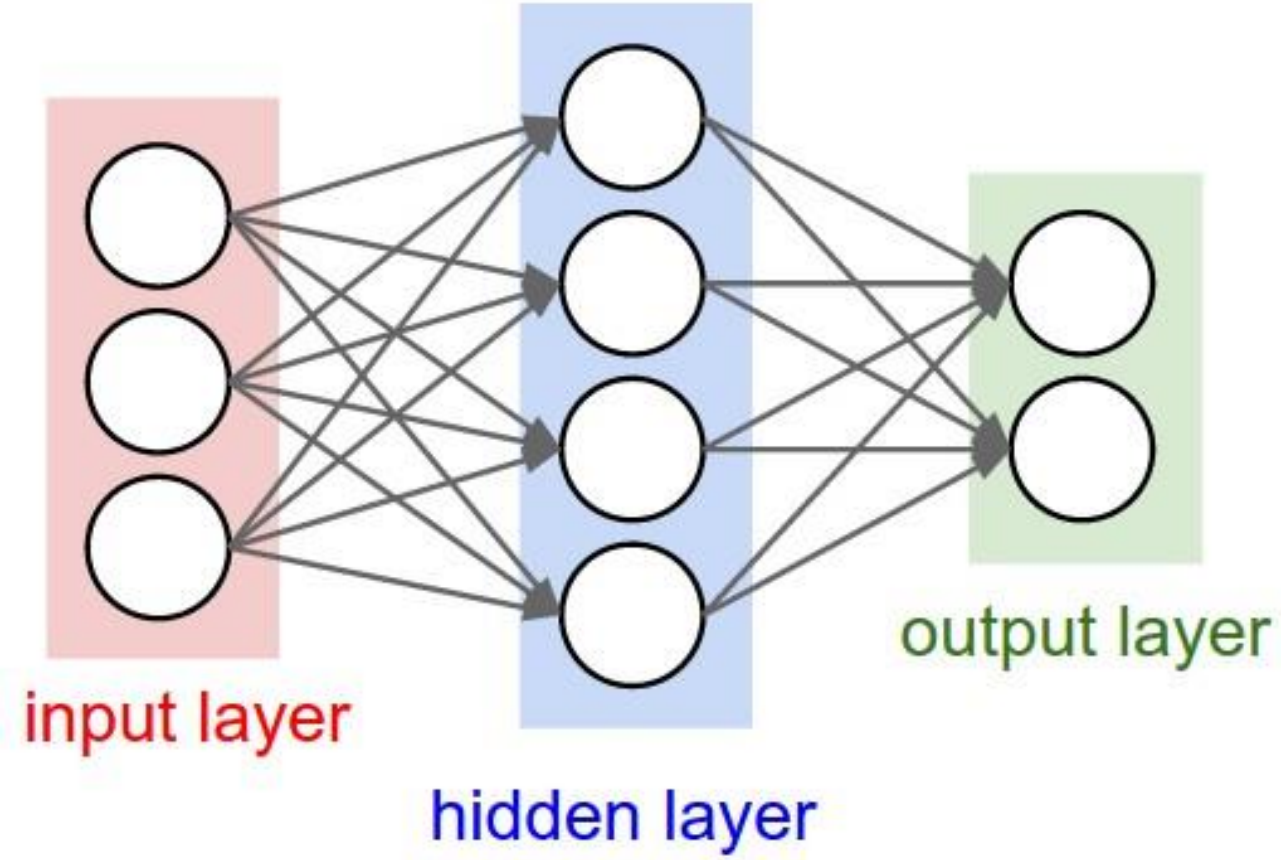
e.g., consider CIFAR-10 example with [32,32,3] images

- 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

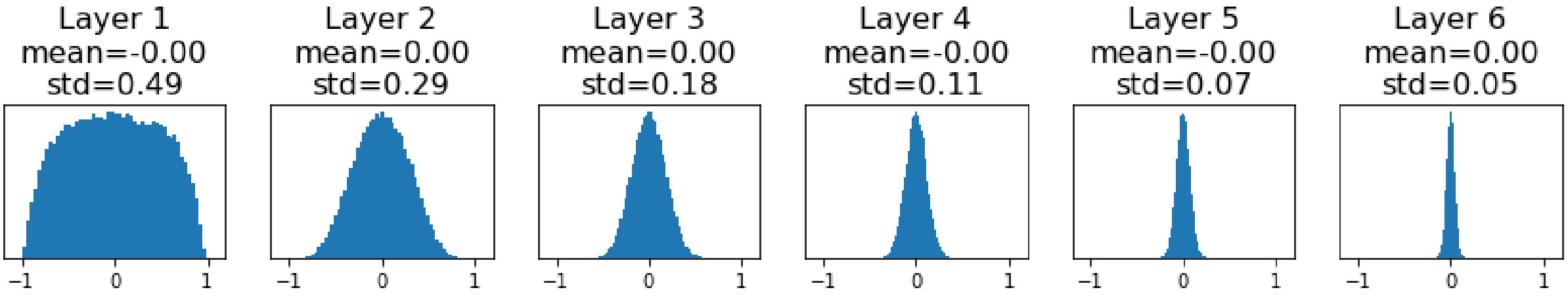
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

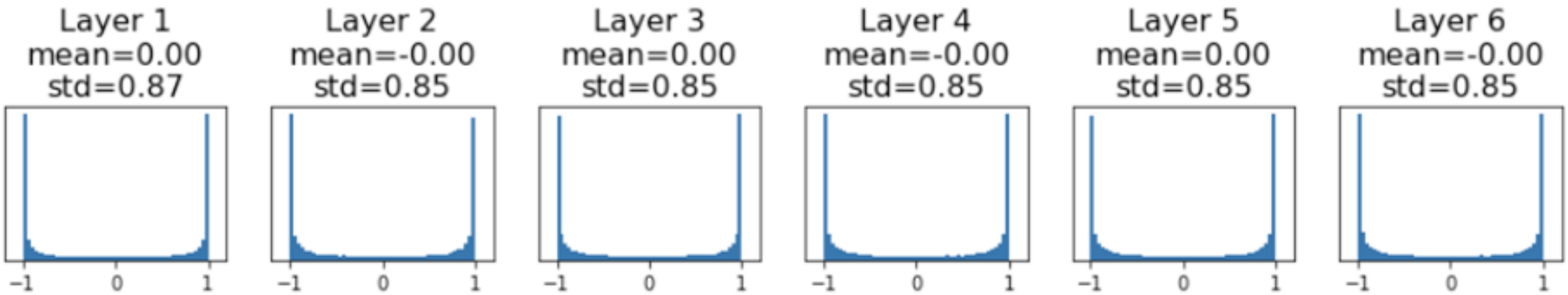
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



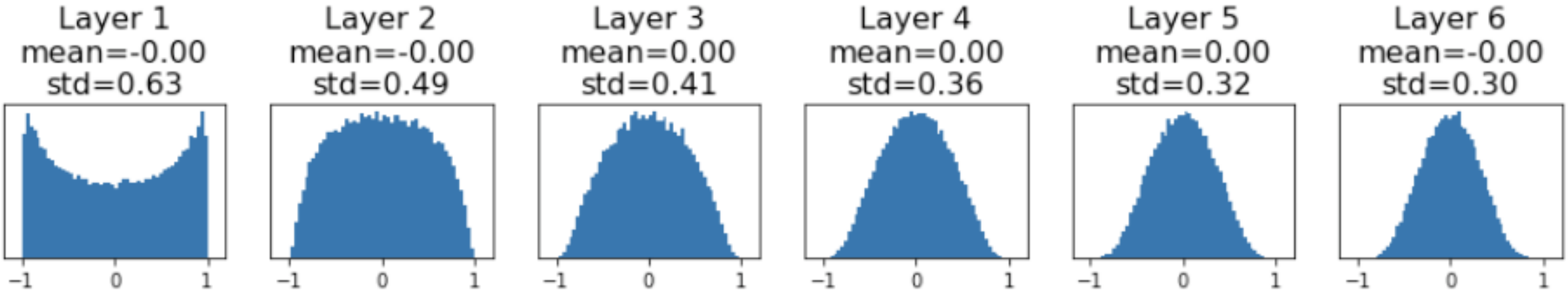
Weight Initialization: Xavier Initialization

```
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

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

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

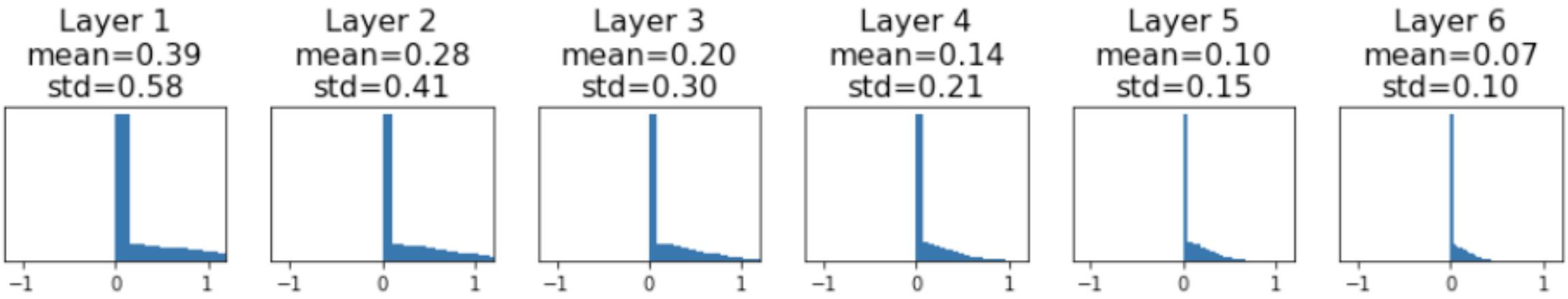
Weight Initialization: What about ReLU?

```
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 =(

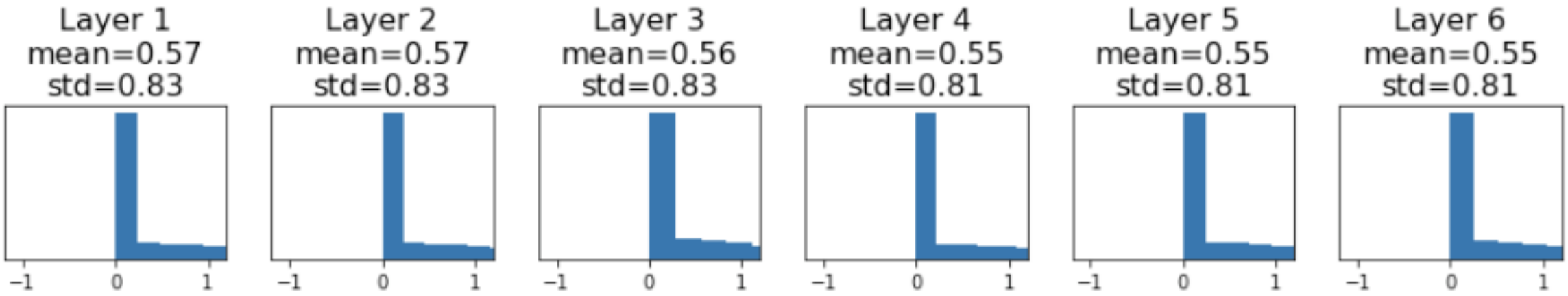


Weight Initialization: Kaiming / MSRA Initialization

```
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)
```

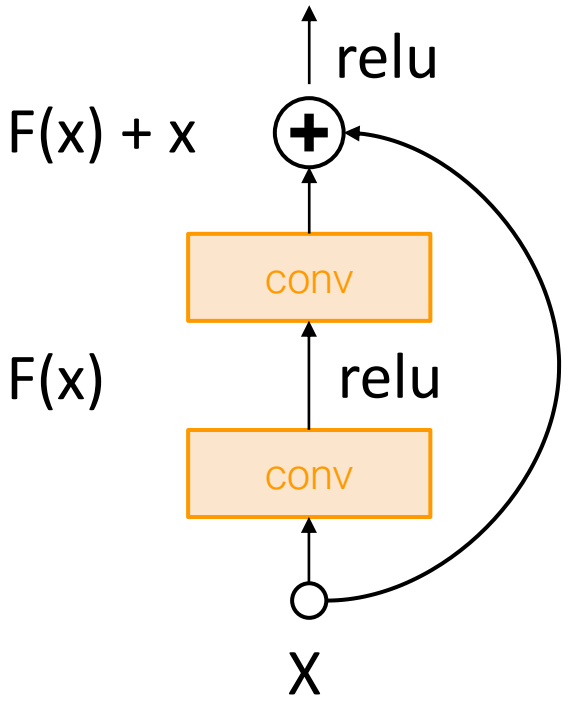
ReLU correction: std = sqrt(2 / Din)

"Just right" –activations nicely scaled for all layers



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

Weight Initialization: Residual Networks



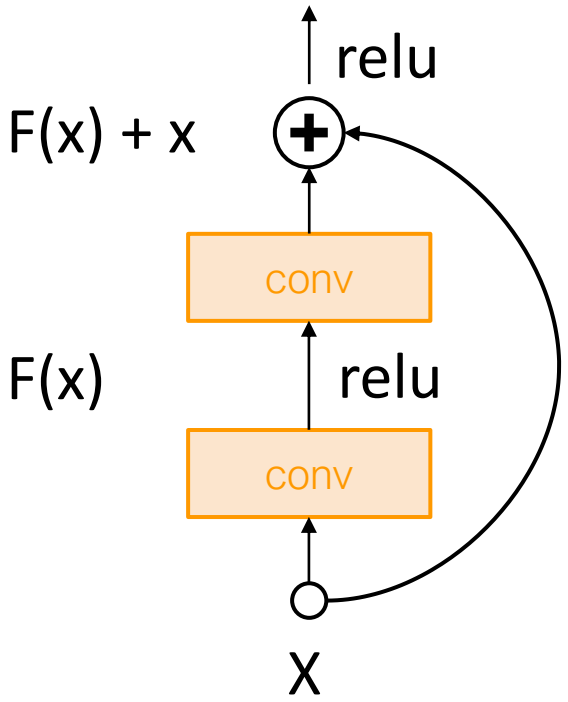
Residual Block

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

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

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

Weight Initialization: Residual Networks



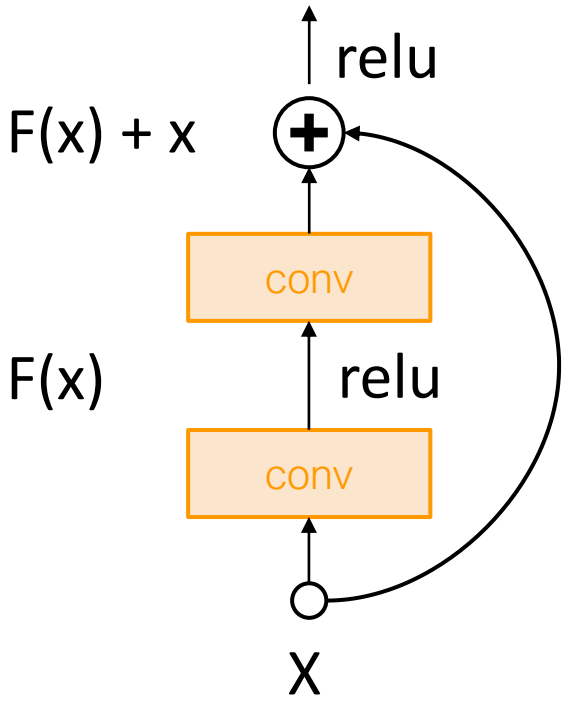
Residual Block

Proposed to train ResNet without BatchNorm; in fact, you don't have to worry about initialization too much when BatchNorm added

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

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

Aside: Weight Initialization for Residual Networks



Residual Block

Cf. $\gamma = 0$ **for the last BN** in each block has a similar effect, but not applicable if BN removed.

- PyTorch official implementation:

```
# Zero-initialize the last BN in each residual branch,
# so that the residual branch starts with zeros, and each residual block behaves like an identity.
# This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
if zero_init_residual:
    for m in self.modules():
        if isinstance(m, Bottleneck) and m.bn3.weight is not None:
            nn.init.constant_(m.bn3.weight, 0)  # type: ignore[arg-type]
        elif isinstance(m, BasicBlock) and m.bn2.weight is not None:
            nn.init.constant_(m.bn2.weight, 0)  # type: ignore[arg-type]
```

`zero_init_residual=False` by default

Zhang et al, "Fixup Initialization: Residual Learning Without Normalization", ICLR 2019
Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017
https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

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

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:

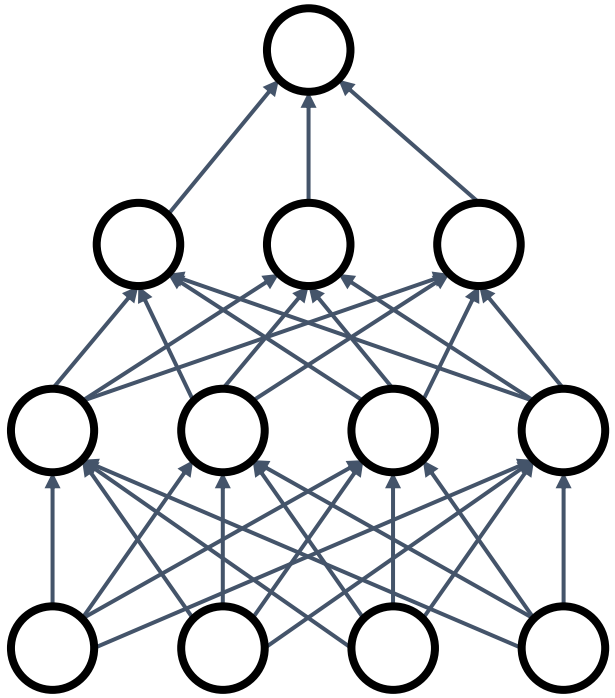
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

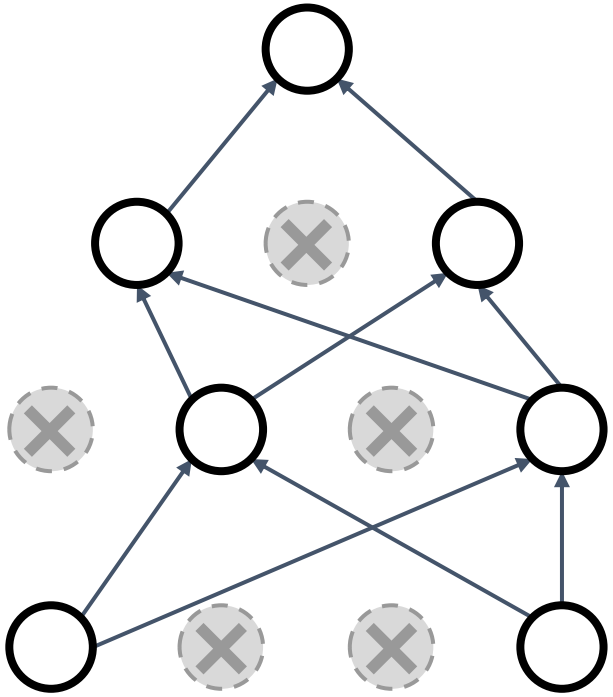
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

Recap: Dropout

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

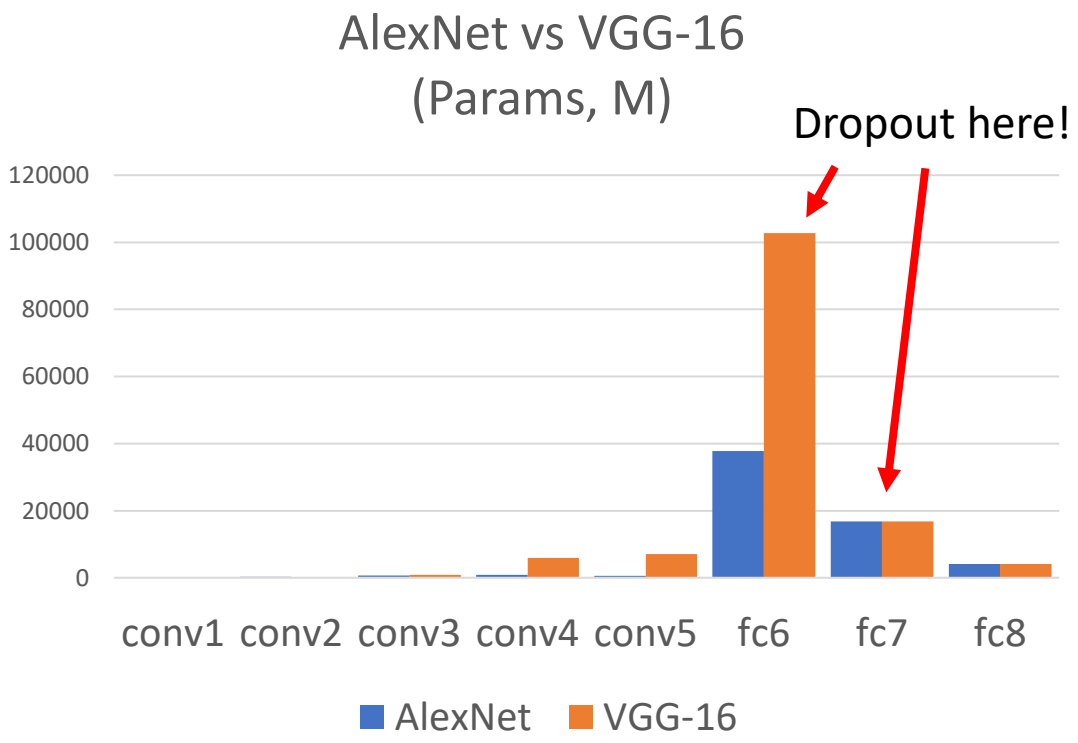




Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Dropout in CNNs

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!

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)

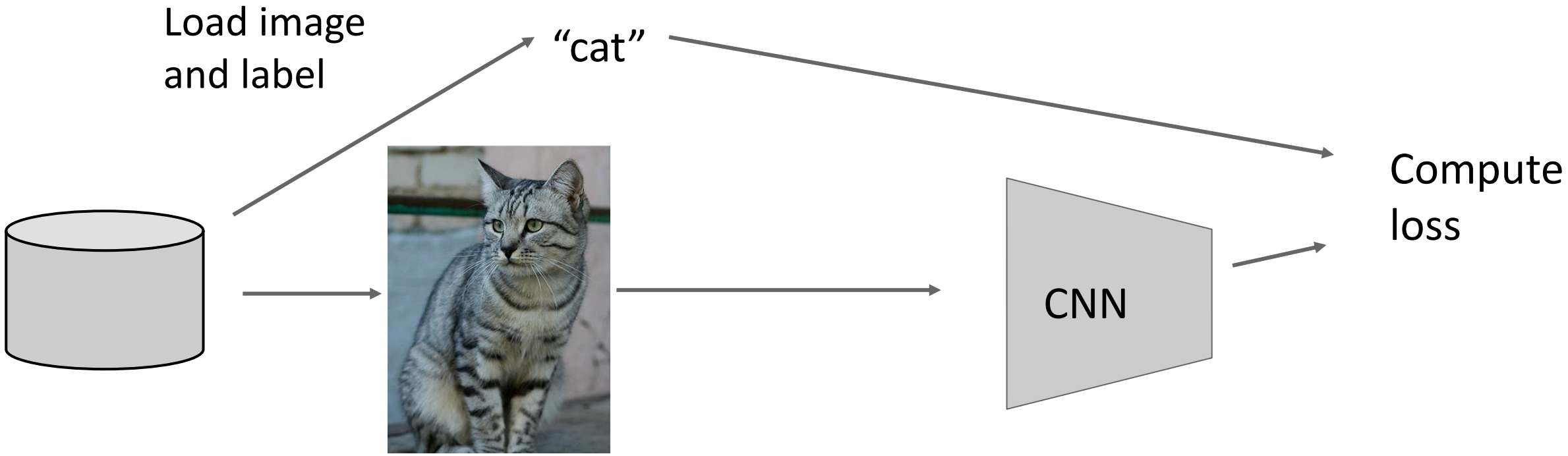
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

Example: Batch Normalization

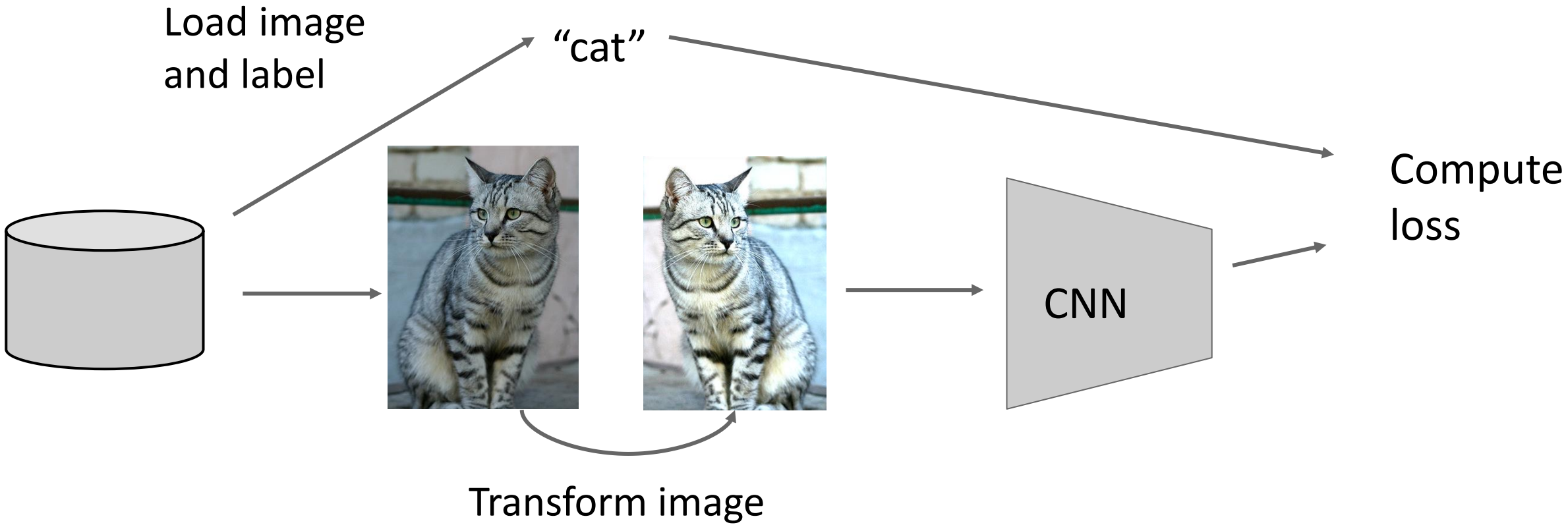
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Data Augmentation

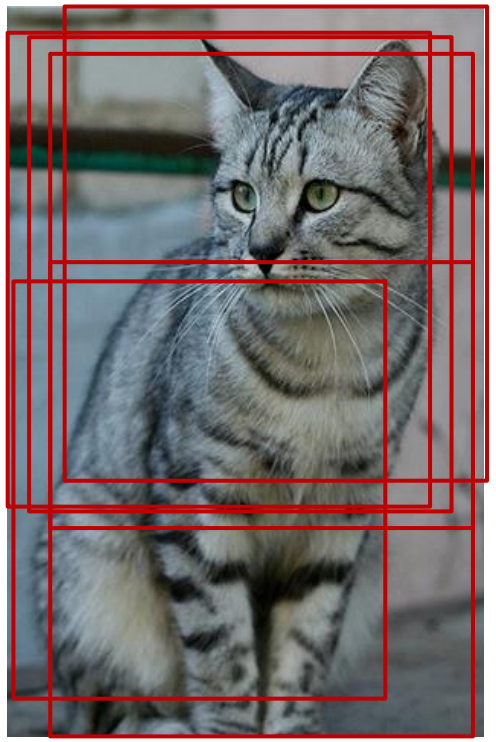


Data Augmentation



Data Augmentation: Horizontal Flips

Data Augmentation: Random Crops and Scales

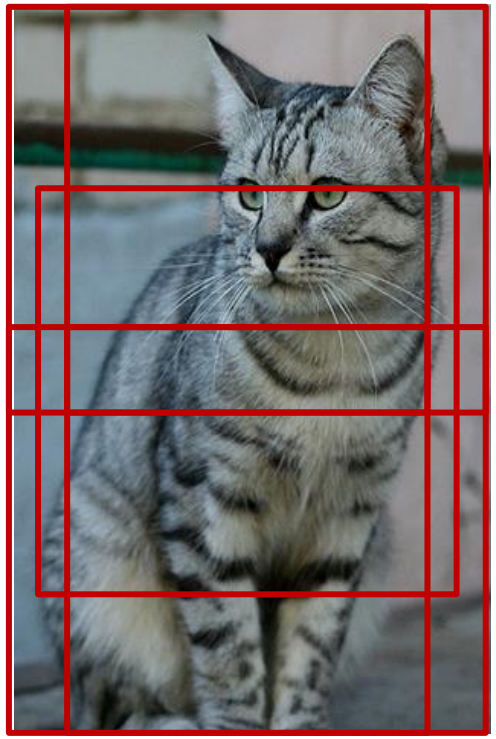


Training: sample random crops / scales

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

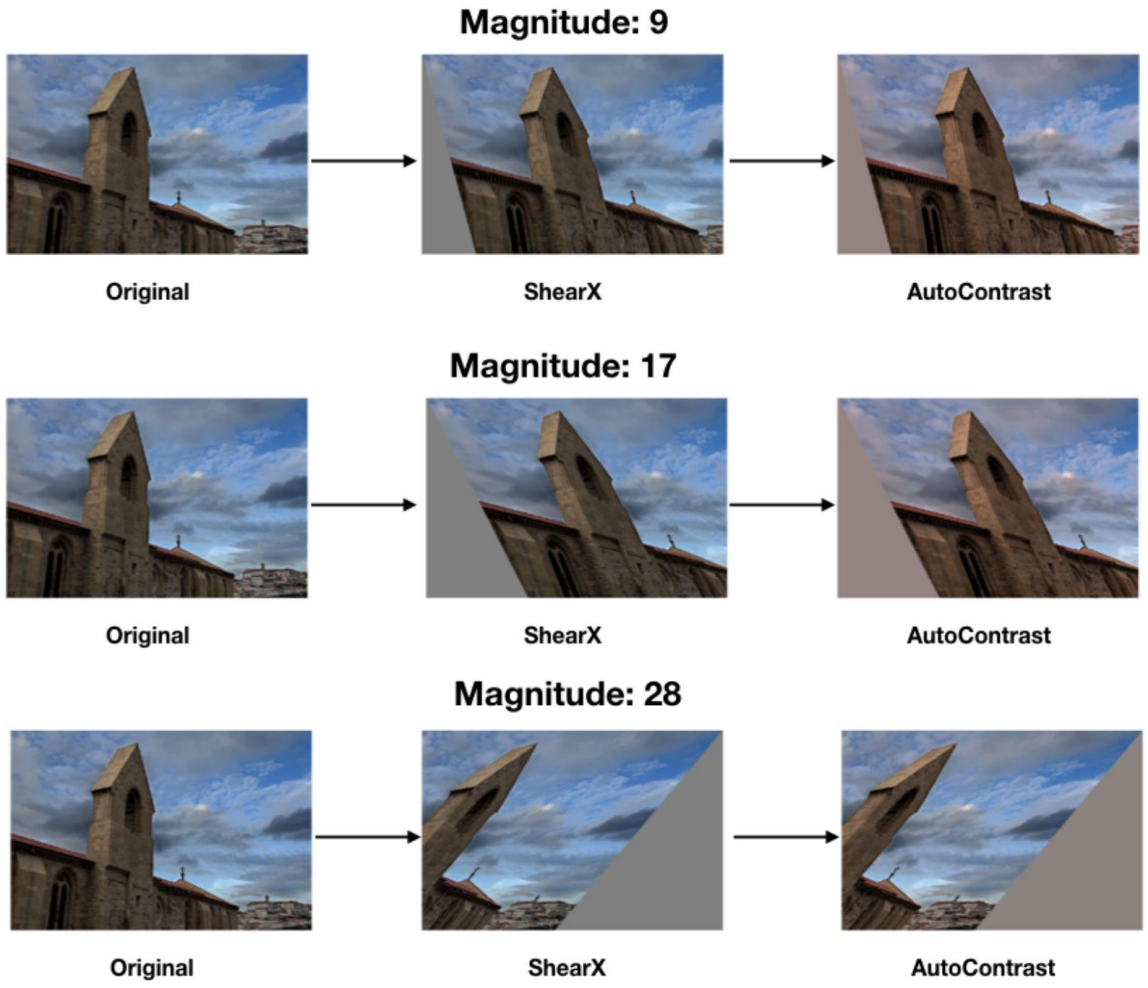
```
transforms = [
'Identity', 'AutoContrast', 'Equalize',
'Rotate', 'Solarize', 'Color', 'Posterize',
'Contrast', 'Brightness', 'Sharpness',
'ShearX', 'ShearY', 'TranslateX', 'TranslateY']

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

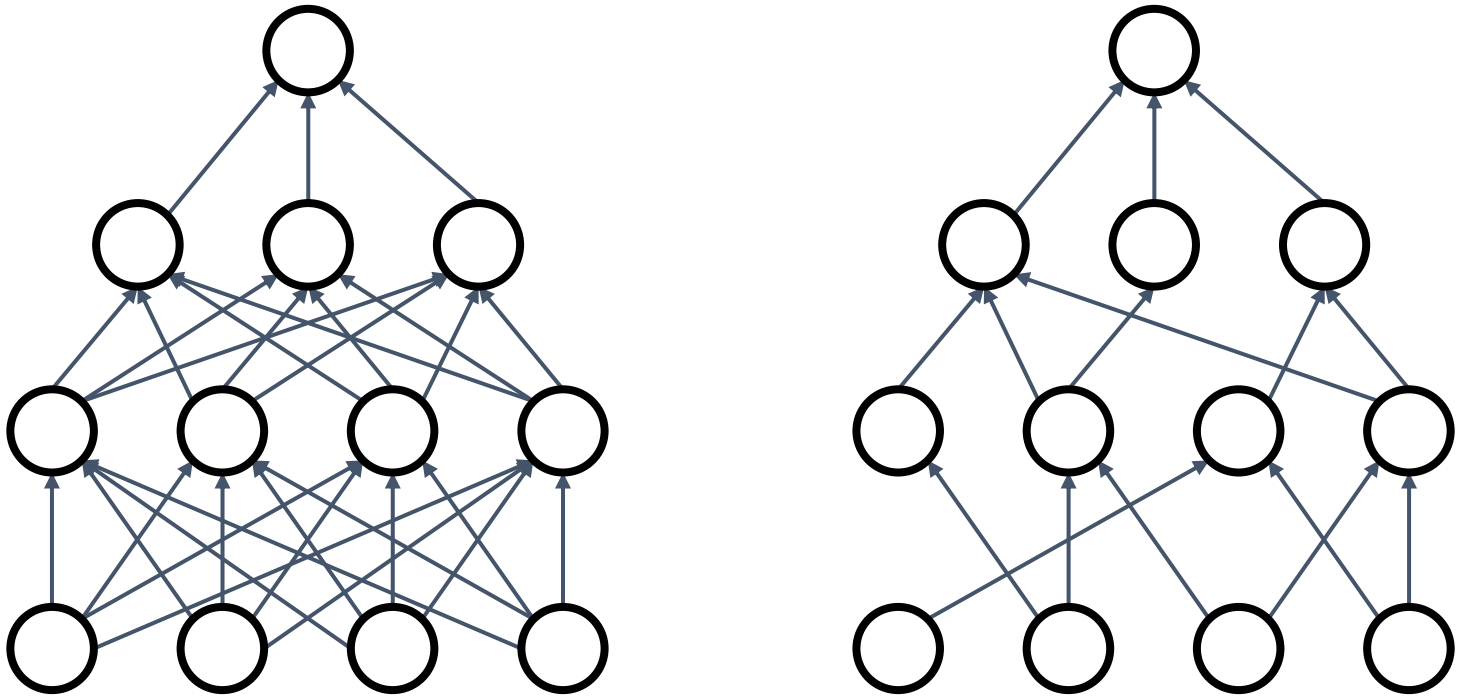
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



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

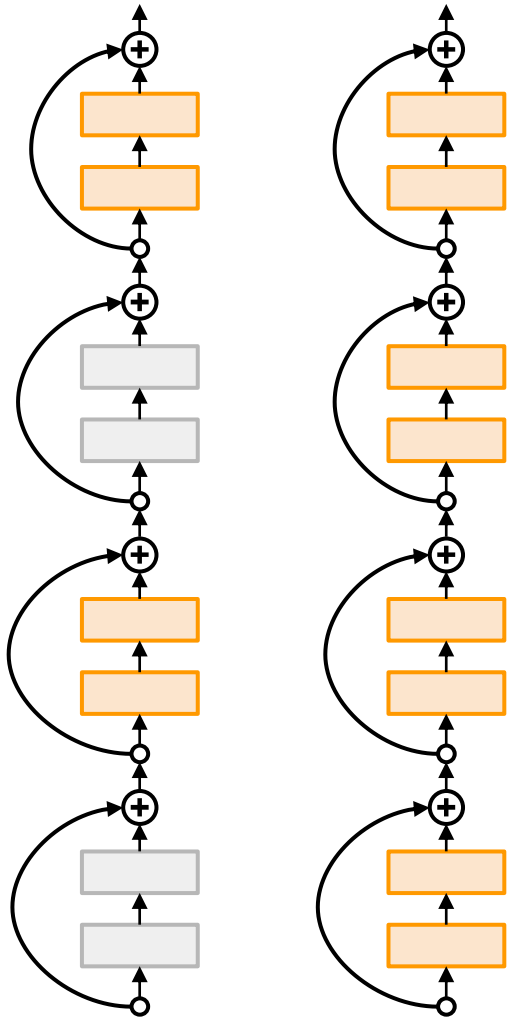
Regularization: Stochastic Depth

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

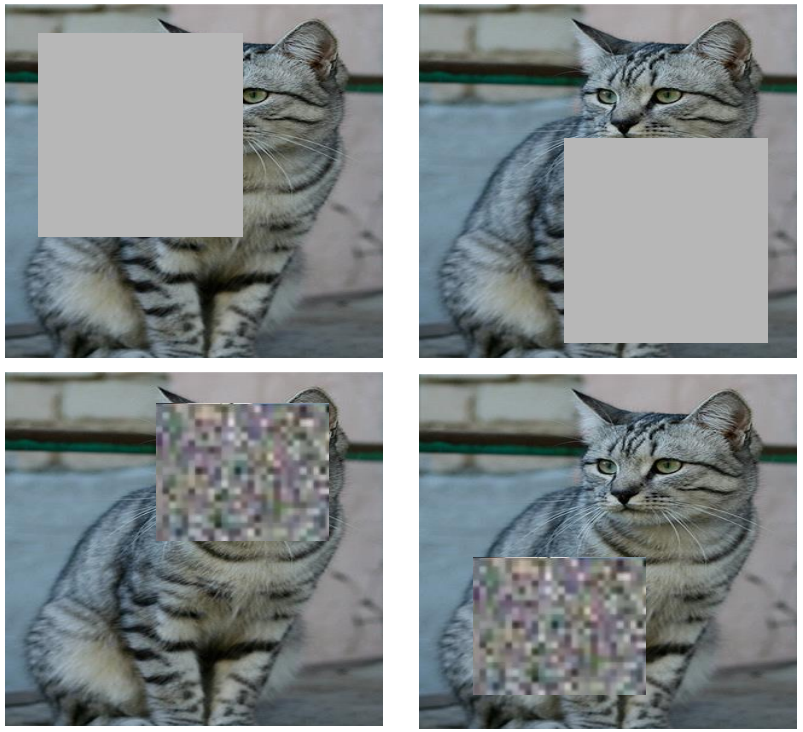


Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

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

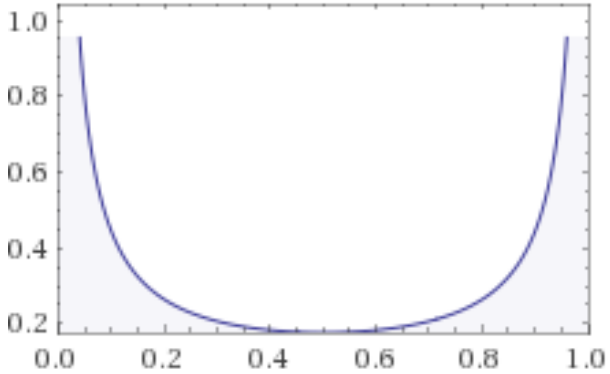


Replace random regions with mean value or random values

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

Regularization: Mixup

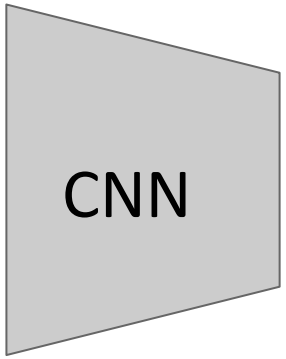
Training: Train on random blends of images
Testing: Use original images



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

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



Target label:
cat: 0.4
dog: 0.6

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

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

Regularization: CutMix

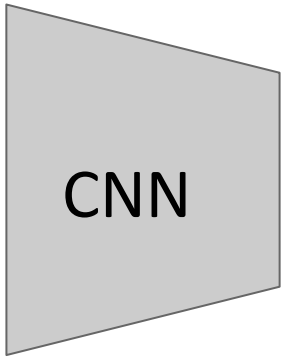
Training: Train on random blends of images

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 $1 - \frac{K-1}{K}\varepsilon$ on the correct category and ε/K on all other categories, with K categories and $\varepsilon \in (0,1)$. 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

Optimization

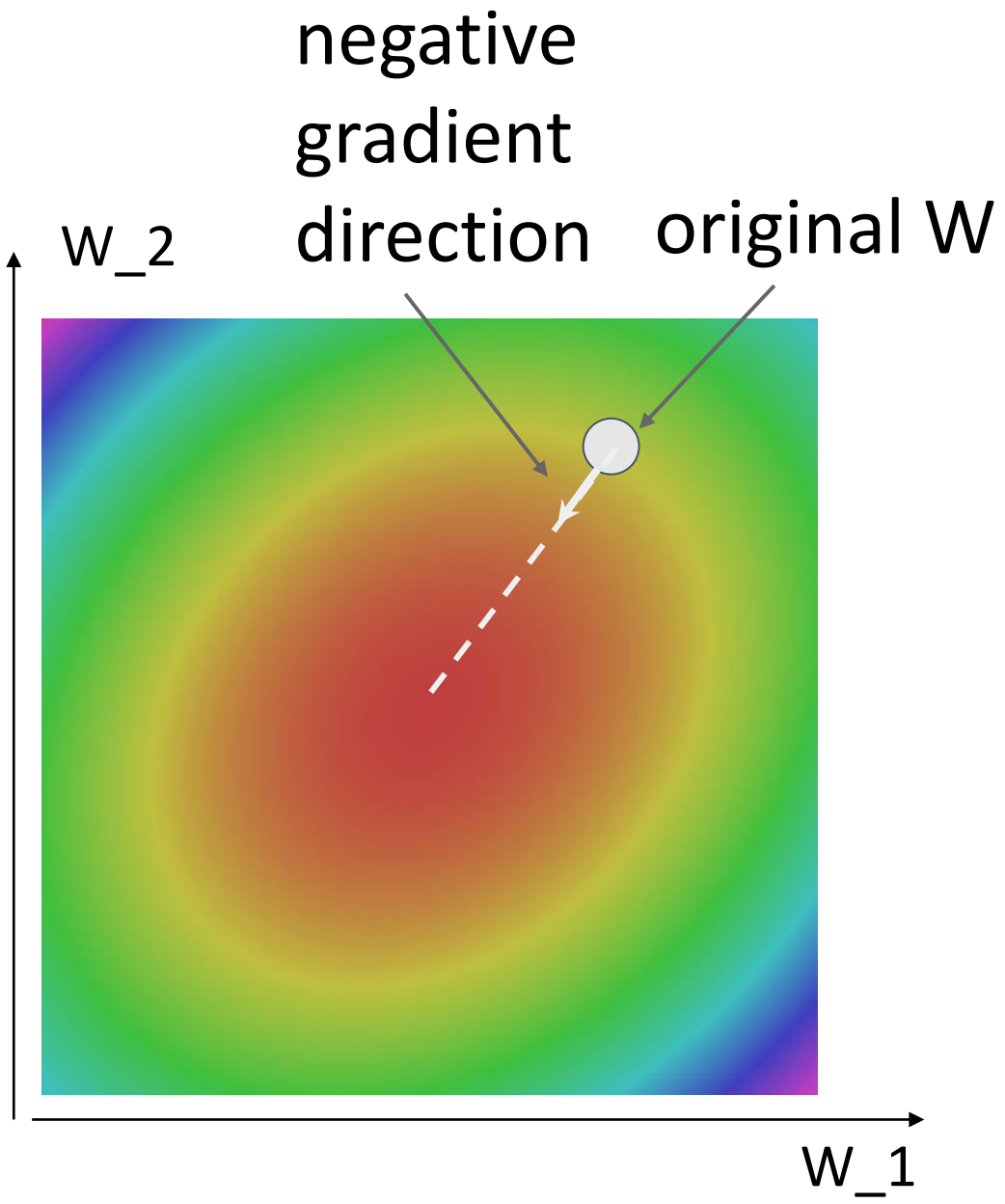
Recap: Gradient Descent

Iteratively step in the direction of the negative gradient (direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



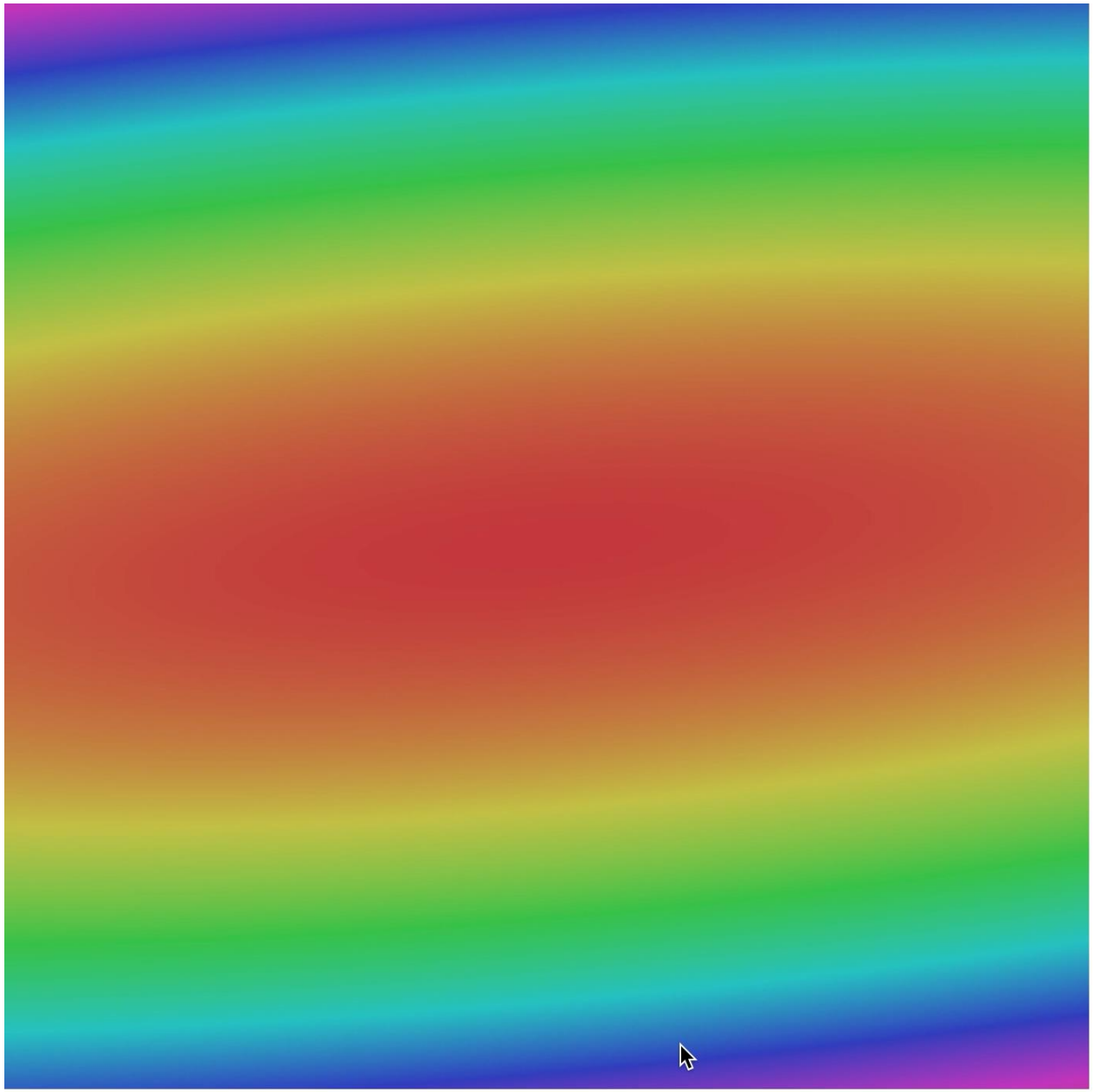
Gradient Descent

Iteratively step in the direction of the negative gradient
(direction of local steepest descent)

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
  dw = compute_gradient(loss_fn, data, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization method
- Number of steps
- Learning rate



Recap: Batch Gradient Descent (BGD)

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

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

Full sum expensive when N is large!

Recap: Stochastic Gradient Descent (SGD)

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

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

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples 32 / 64 / 128 common

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Recap: Stochastic Gradient Descent (SGD)

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

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

```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
  minibatch = sample_data(data, batch_size)
  dw = compute_gradient(loss_fn, minibatch, w)
  w -= learning_rate * dw
```

Note: Previously we called this mini-batch GD

Because mini-batch GD is also stochastic, we call this SGD in the context of deep learning

Hyperparameters:

- Weight initialization
- Number of steps
- Learning rate
- Batch size
- Data sampling

Recap: Stochastic Gradient Descent (SGD)

$$L(W) = \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda R(W)$$
$$\approx \frac{1}{N}\sum_{i=1}^{N} L(x_i, y_i, W) + \lambda R(W)$$

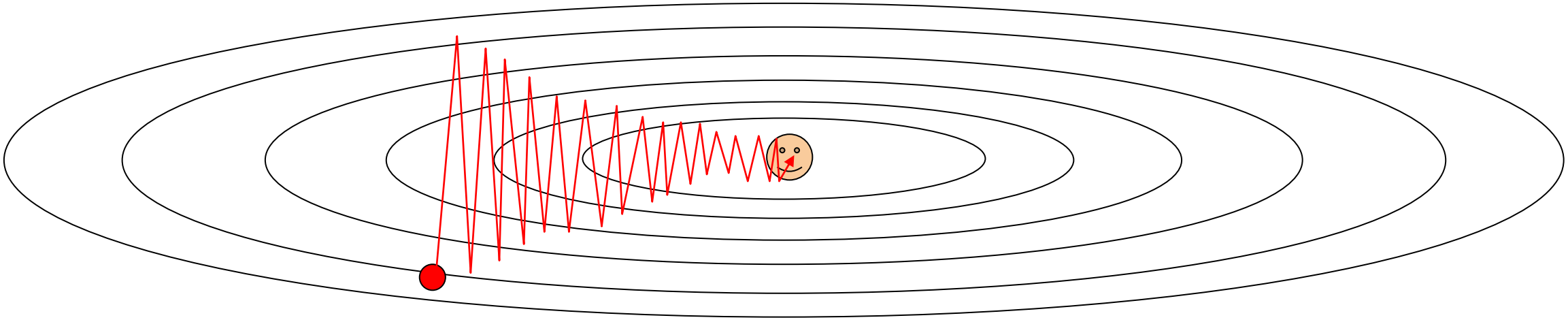
Think of loss as an expectation over the full **data distribution** p_{data}

Approximate expectation via sampling

$$\nabla_W L(W) = \nabla_W \mathbb{E}_{(x,y)\sim p_{data}}[L(x,y,W)] + \lambda \nabla_W R(W)$$
$$\approx \sum_{i=1}^{N} \nabla_w L_W(x_i, y_i, W) + \nabla_w R(W)$$

Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

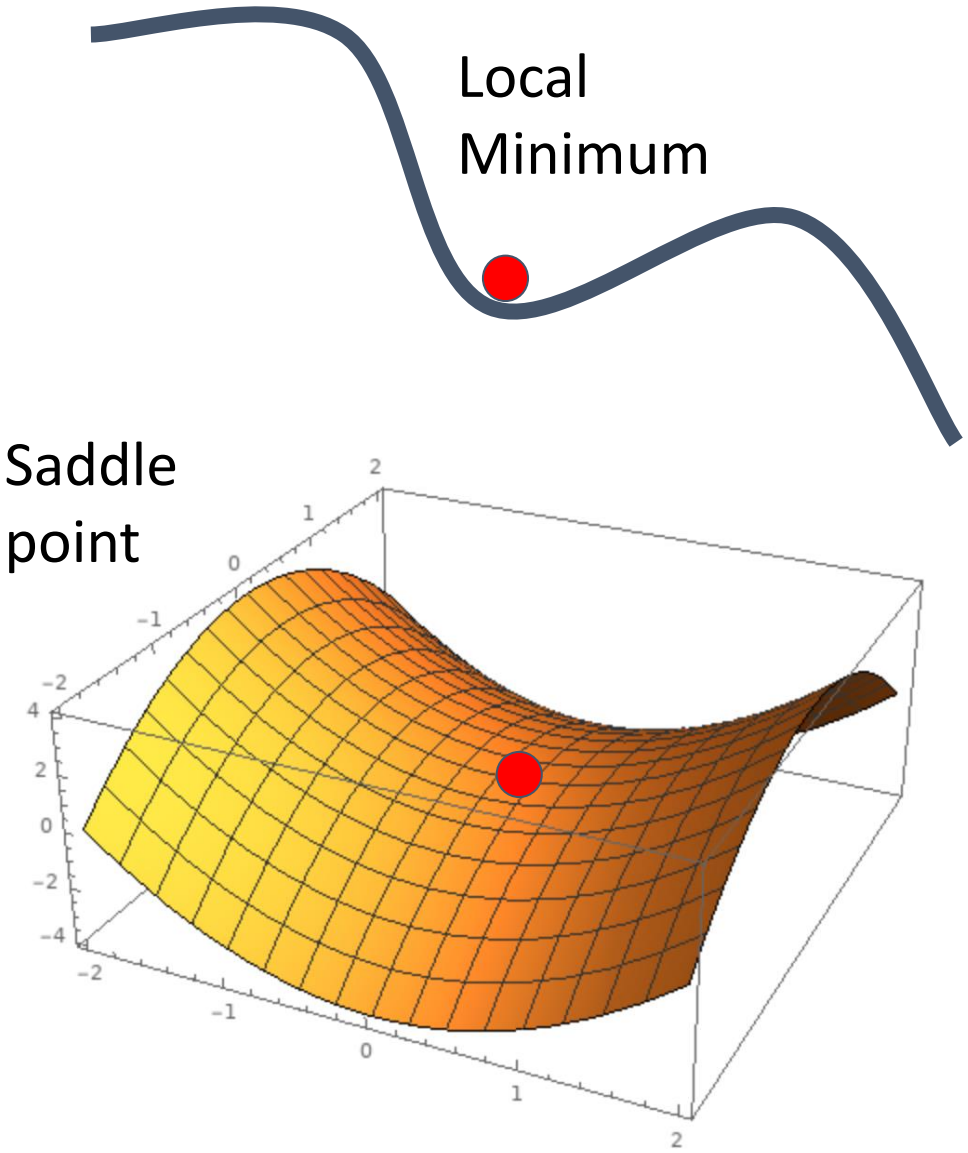


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

What if the loss function has a **local minimum** or **saddle point**?

Zero gradient, gradient descent gets stuck

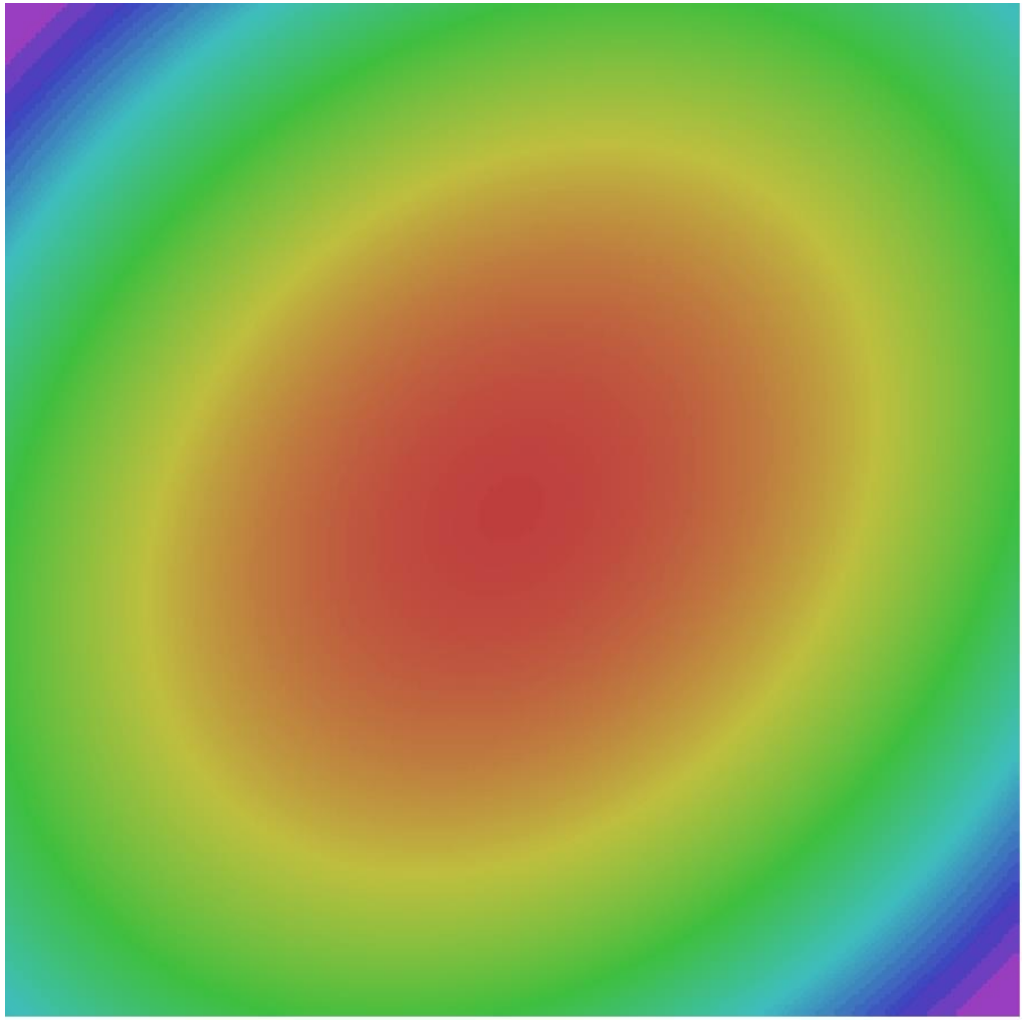


Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

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



SGD

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

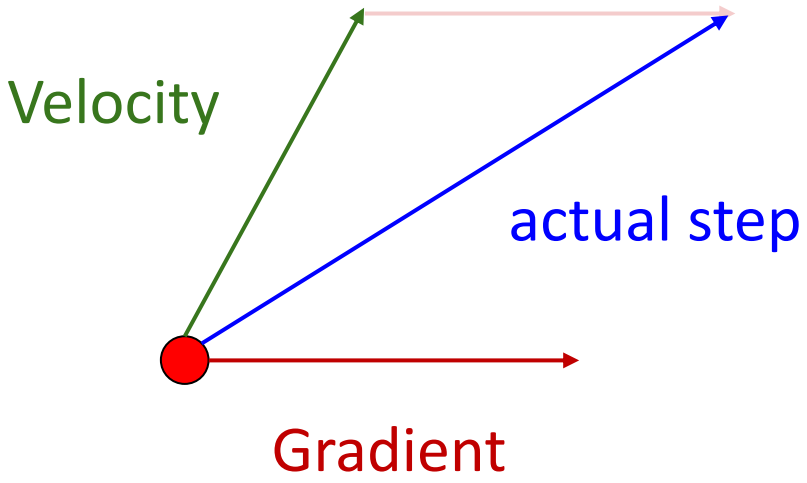
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

SGD+Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v - learning_rate * dw
  w += v
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

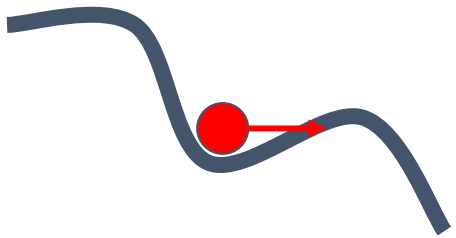
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

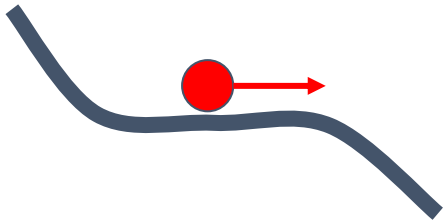
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

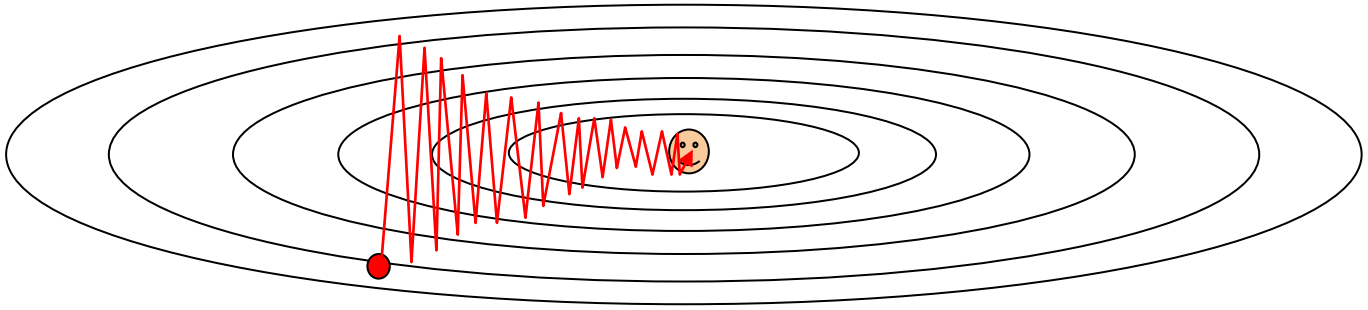
Local Minima



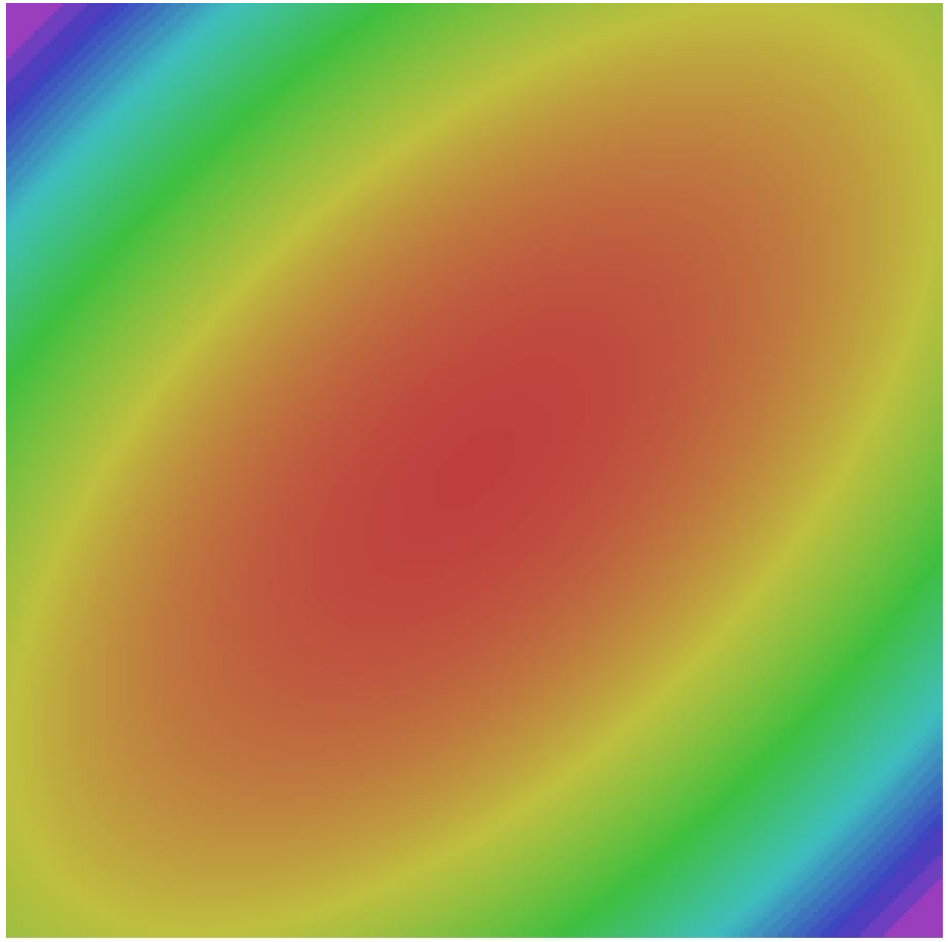
Saddle points



Poor Conditioning



Gradient Noise

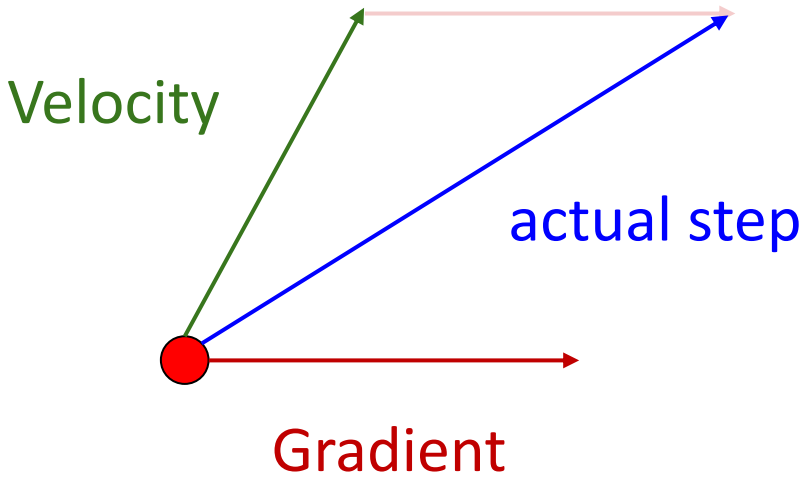


SGD SGD+Momentum

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

SGD + Momentum

Momentum update:

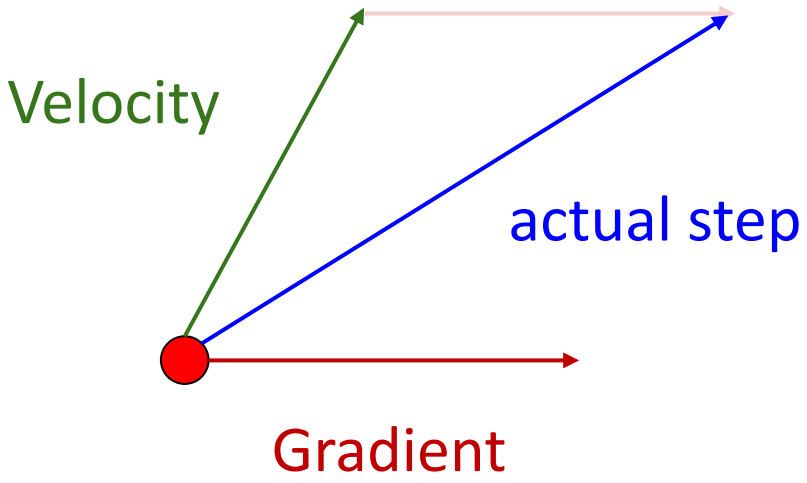


Combine gradient at current point with velocity to get step used to update weights

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

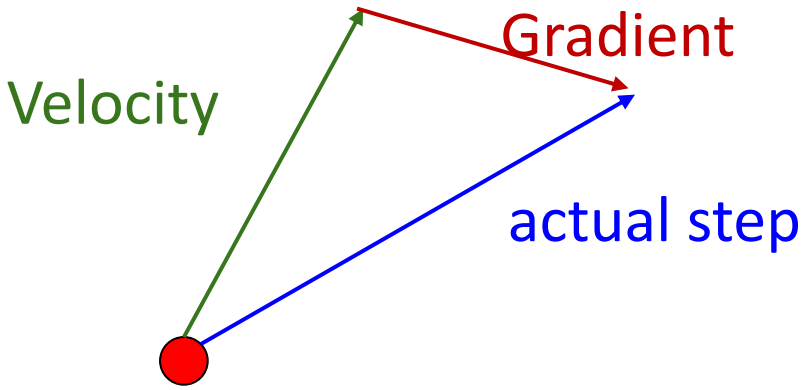
Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



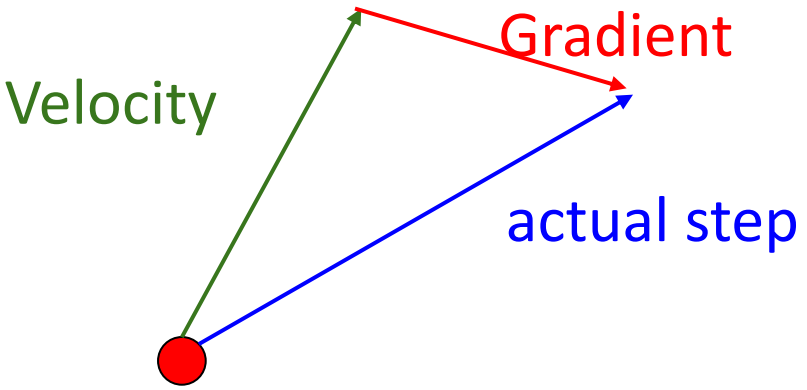
"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

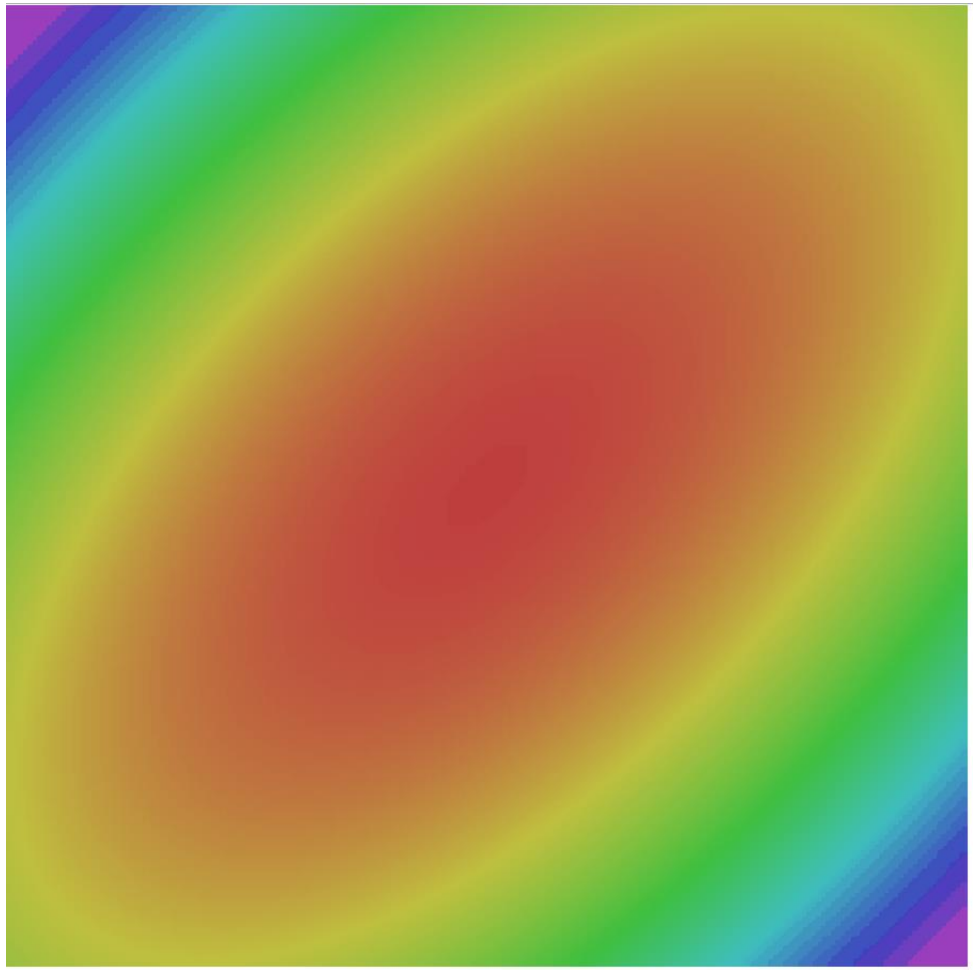
Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  old_v = v
  v = rho * v - learning_rate * dw
  w -= rho * old_v - (1 + rho) * v
```

Nesterov Momentum



SGD

SGD+Momentum

Nesterov

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

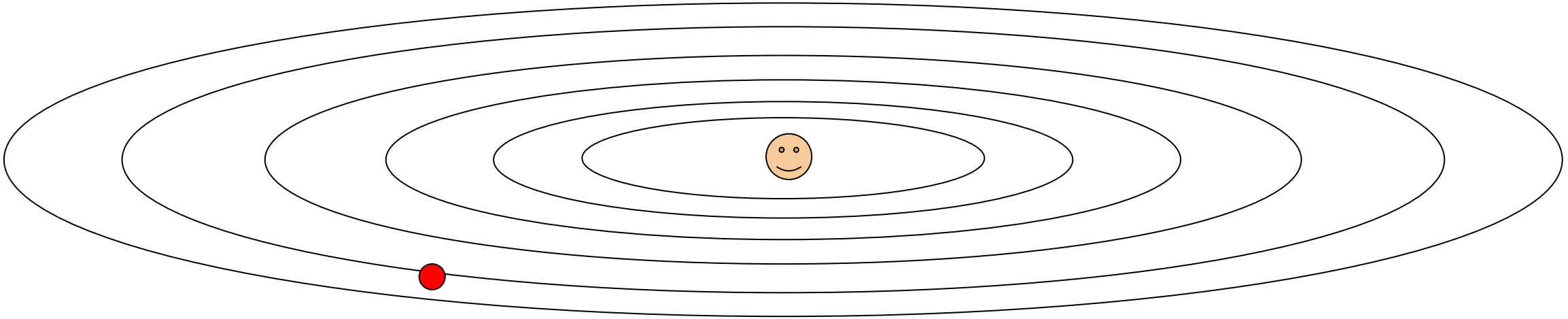
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates"
or "adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```



Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

RMSProp: “Leaky Adagrad”

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

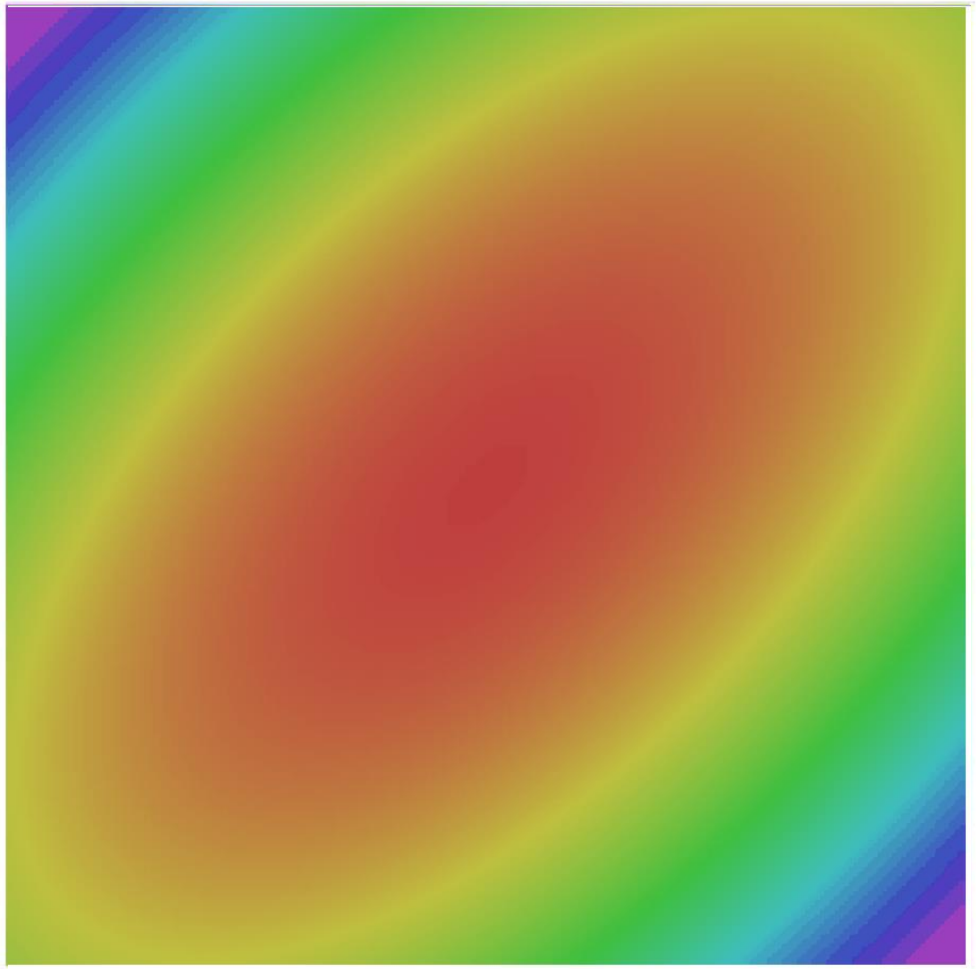
AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Tieleman and Hinton, 2012

RMSProp



SGD

SGD+Momentum

RMSProp

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```

SGD+Momentum

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

Bias correction

Q: What happens at t=0?
(Assume beta2 = 0.999)

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Momentum

AdaGrad / RMSProp

Bias correction

Bias correction for the fact that first and second moment estimates start at zero

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam (almost): RMSProp + Momentum

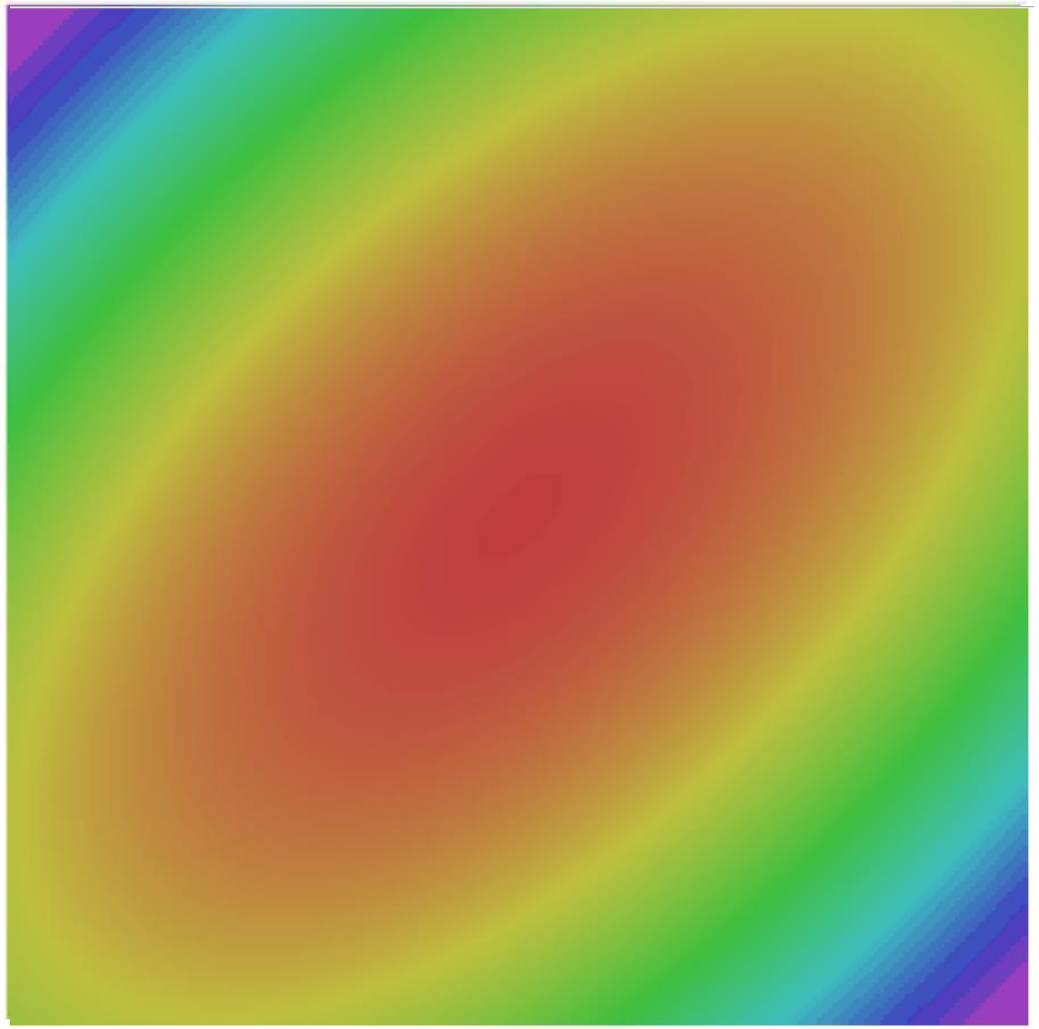
```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  moment1_unbias = moment1 / (1 - beta1 ** t)
  moment2_unbias = moment2 / (1 - beta2 ** t)
  w -= learning_rate * moment1_unbias / (moment2_unbias.sqrt() + 1e-7)
```

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Adam



SGD

SGD+Momentum

RMSProp

Adam

Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	✗	✗	✗	✗
SGD+Momentum	✓	✗	✗	✗
Nesterov	✓	✗	✗	✗
AdaGrad	✗	✓	✗	✗
RMSProp	✗	✓	✓	✗
Adam	✓	✓	✓	✓

L2 Regularization vs Weight Decay

Optimization Algorithm

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

L2 Regularization

$$L(w) = L_{data}(w) + \boldsymbol{\lambda \|w\|^2}$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + \boldsymbol{2\lambda w_t}$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

Weight Decay

$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + \boldsymbol{2\lambda w_t}$$
$$w_{t+1} = w_t - \alpha s_t$$

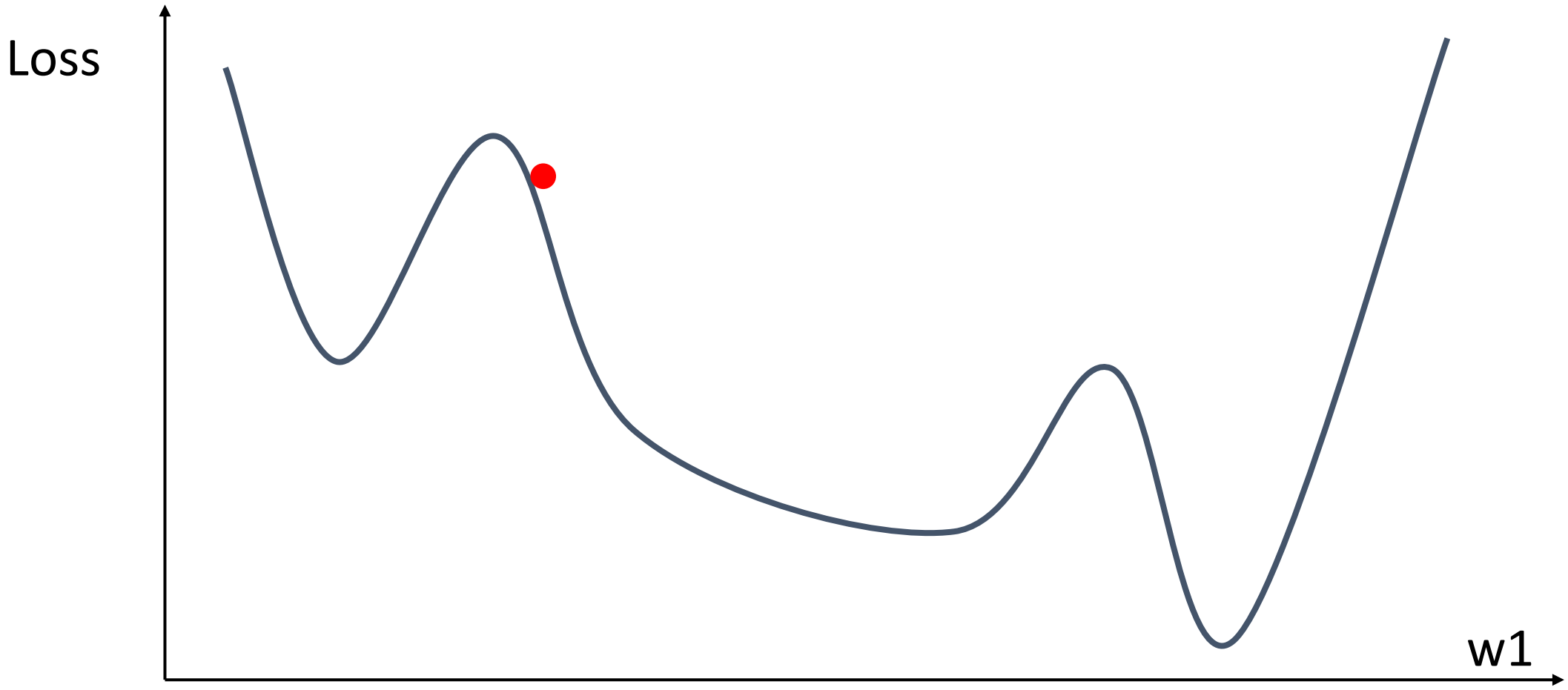
AdamW: Decoupled Weight Decay

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

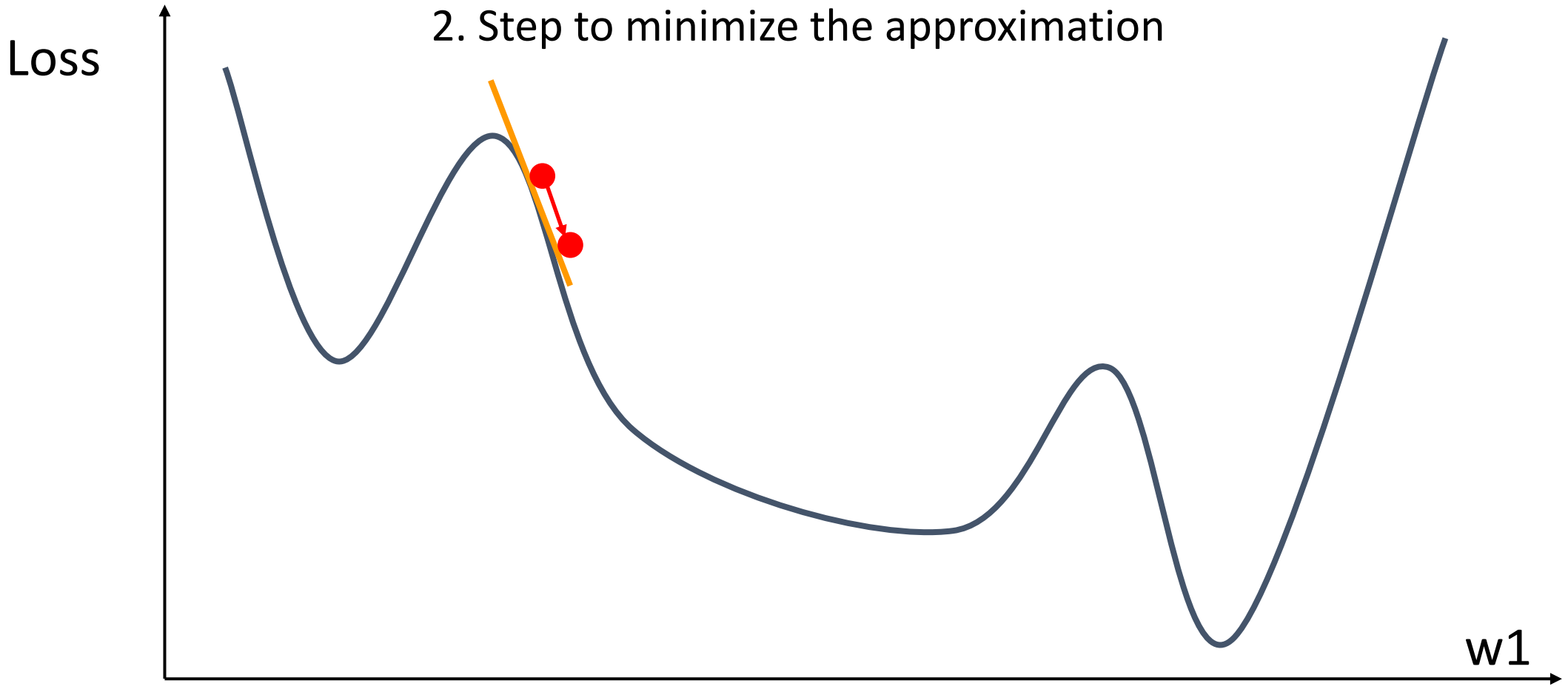
Loshchilov and Hutter, "Decoupled Weight Decay Regularization", ICLR 2019

Recap: First-Order Optimization



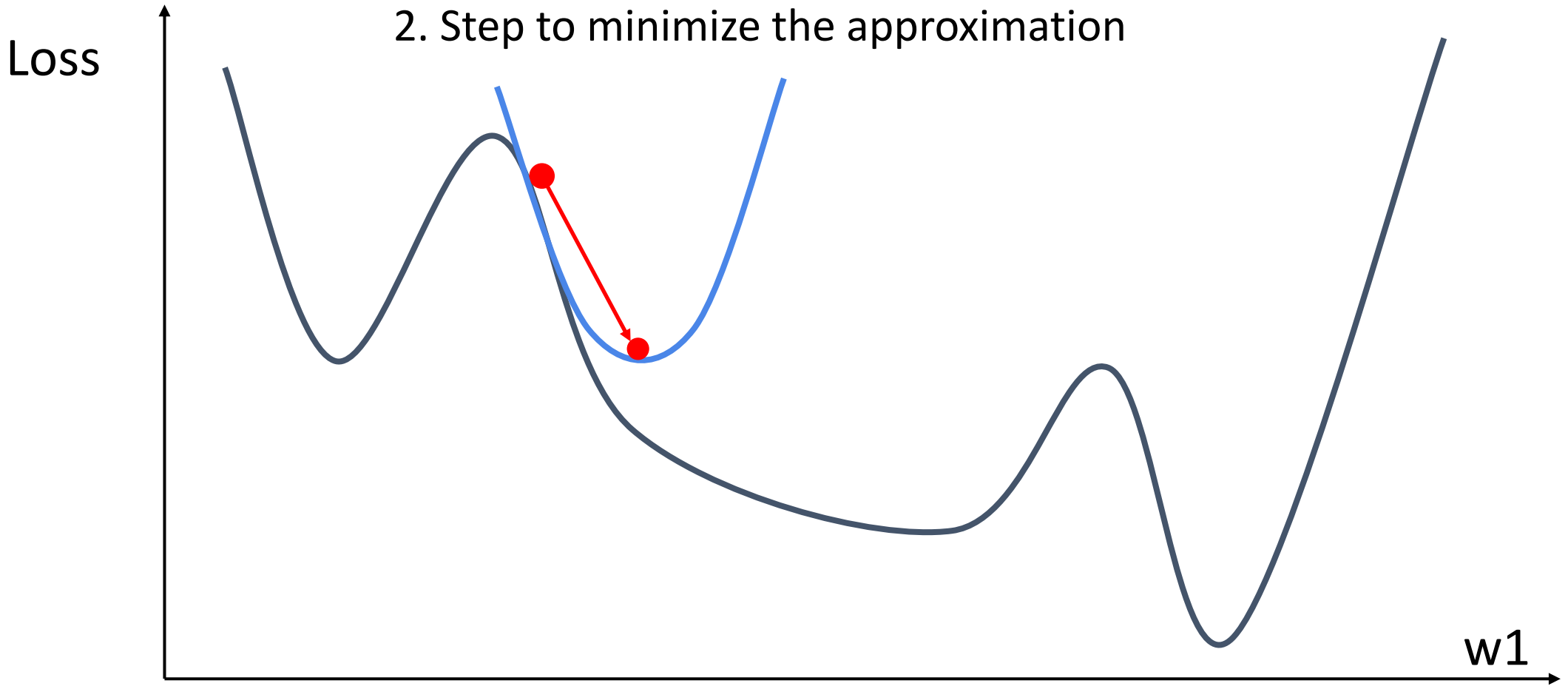
Recap: **First-Order** Optimization

1. Use gradient to make linear approximation
2. Step to minimize the approximation



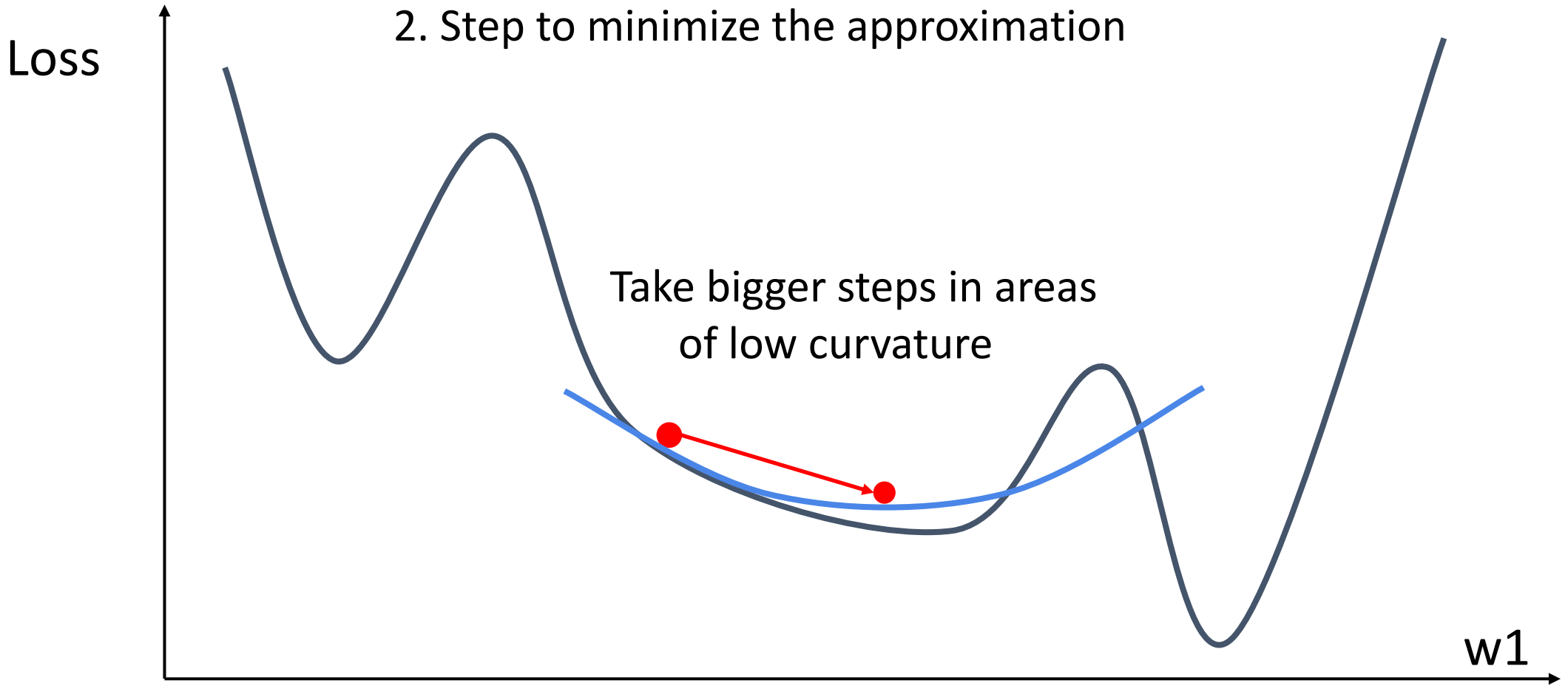
Recap: **Second-Order** Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Recap: **Second-Order** Optimization

1. Use gradient and Hessian to make quadratic approximation
2. Step to minimize the approximation



Recap: **Second-Order** Optimization

Second-Order Taylor Expansion:

$$L(w) \approx L(w_0) + (w - w_0)^\intercal \nabla_w L(w_0) + \frac{1}{2}(w - w_0)^\intercal \mathbf{H}_w L(w_0)(w - w_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

Q: Why is this impractical?

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Second-Order Optimization

$$w^* = w_0 - \mathbf{H}_w L(w_0)^{-1} \nabla_w L(w_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian but only a few vectors.

Second-Order Optimization: **L-BFGS**

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, “On optimization methods for deep learning, ICML 2011”
Ba et al, “Distributed second-order optimization using Kronecker-factored approximations”, ICLR 2017

Optimization in Practice

- Conventionally, we use

  minibatch SGD + momentum + some sensibly changing learning rate

  - This is typically what is meant by **SGD**;
    people often do not talk about momentum and changing lr, but they are there

- **Adam** is a good default choice
  - In many cases, **SGD** can outperform **Adam**, but it may require more tuning

- If you can afford to do full batch updates, then try out **L-BFGS**
  (and don't forget to disable all sources of noise)

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

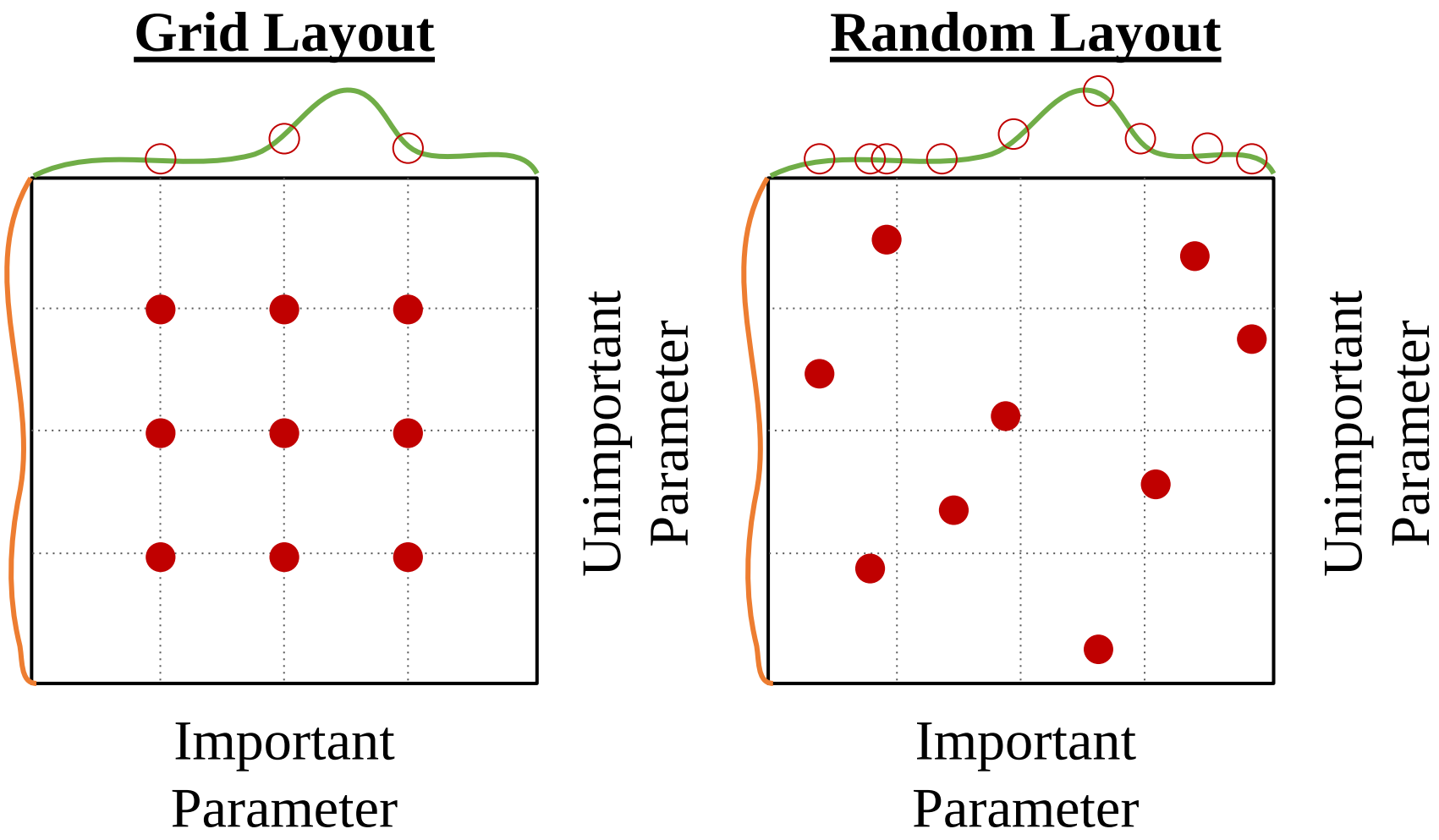
Choosing Hyperparameters: Random Search

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

Example:
Weight decay: log-uniform on $[1\text{x}10^{-4}, 1\text{x}10^{-1}]$
Learning rate: log-uniform on $[1\text{x}10^{-4}, 1\text{x}10^{-1}]$

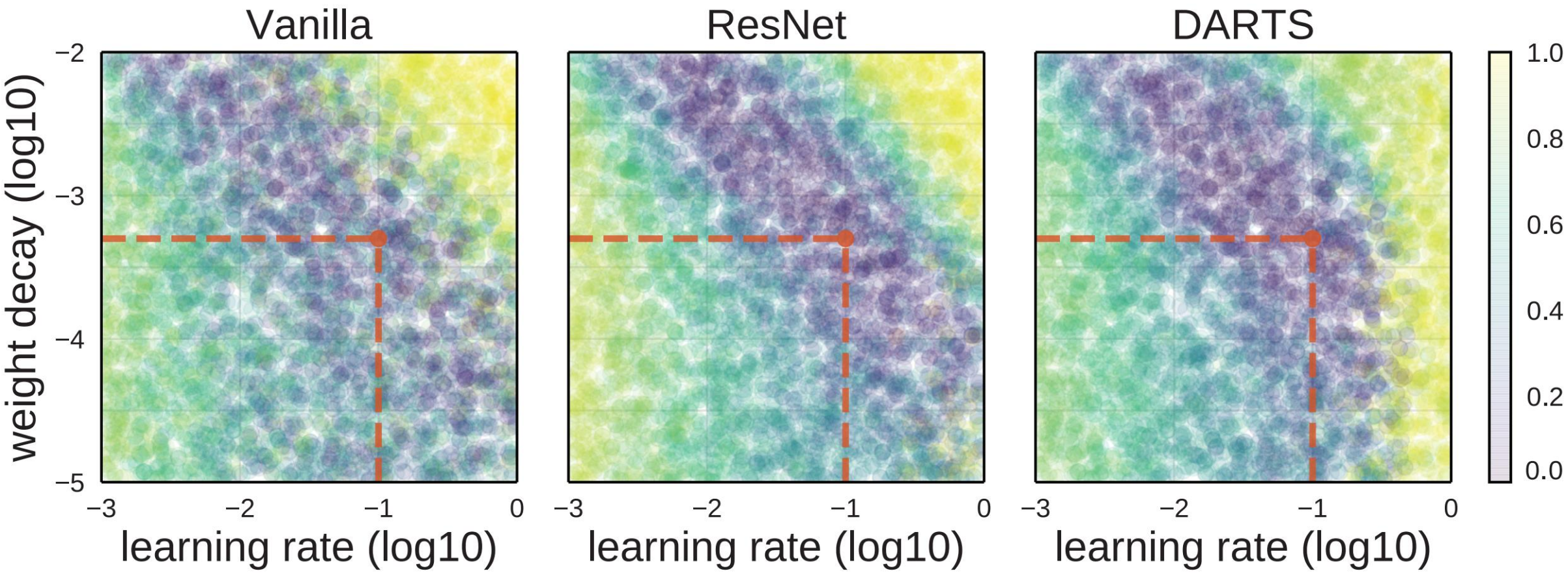
Run many different trials

Hyperparameters: Random vs. Grid Search



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

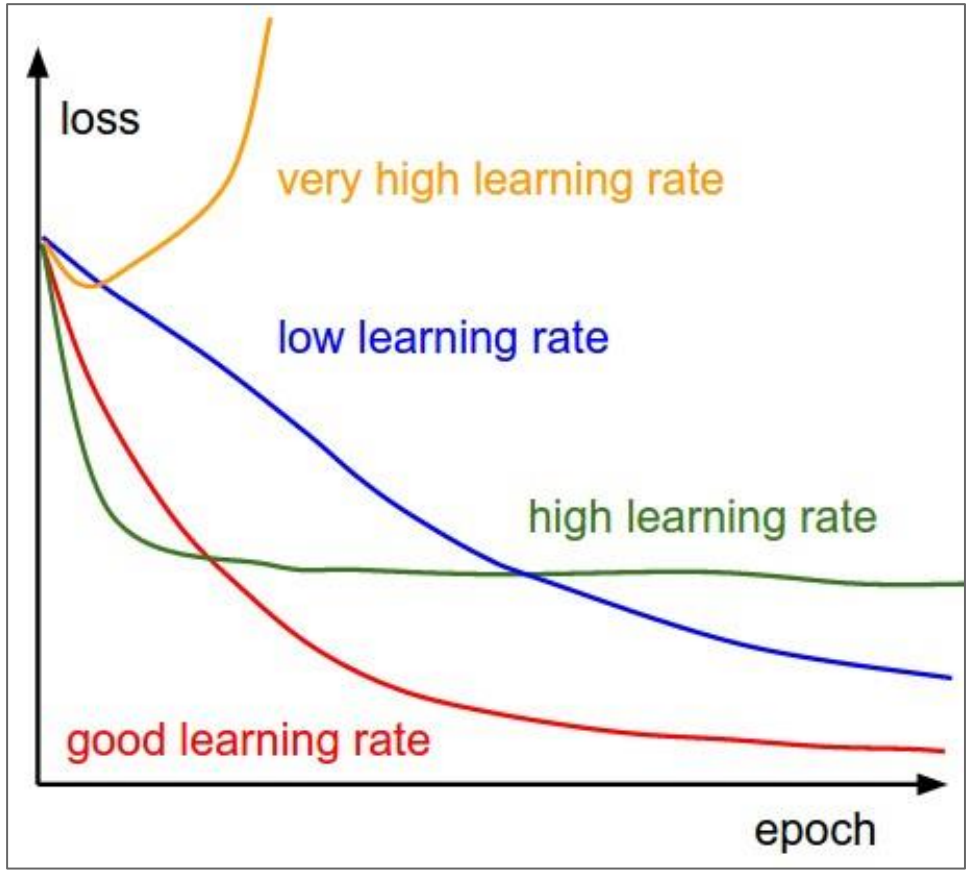
Choosing Hyperparameters by Random Search



Radosavovic et al, “On Network Design Spaces for Visual Recognition”, ICCV 2019

Choosing Hyperparameters: Learning Schedule

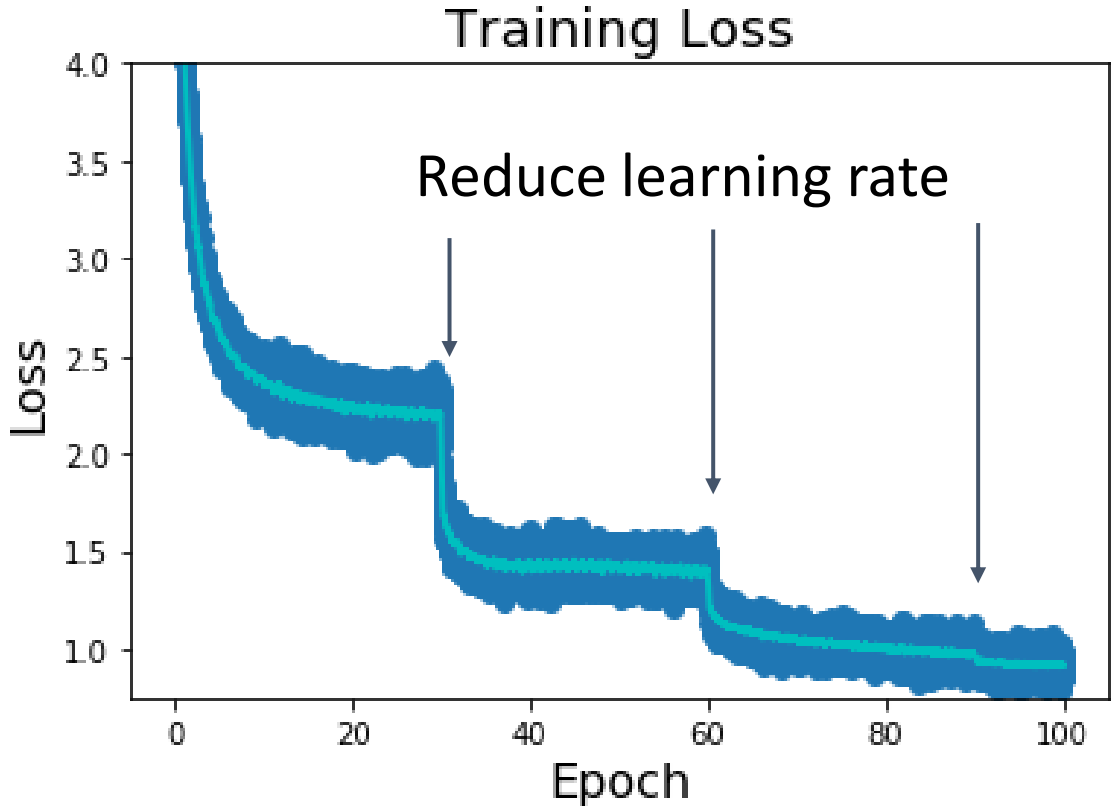
SGD, SGD+Momentum, Adagrad, RMSProp, Adam
all have **learning rate** as a hyperparameter.



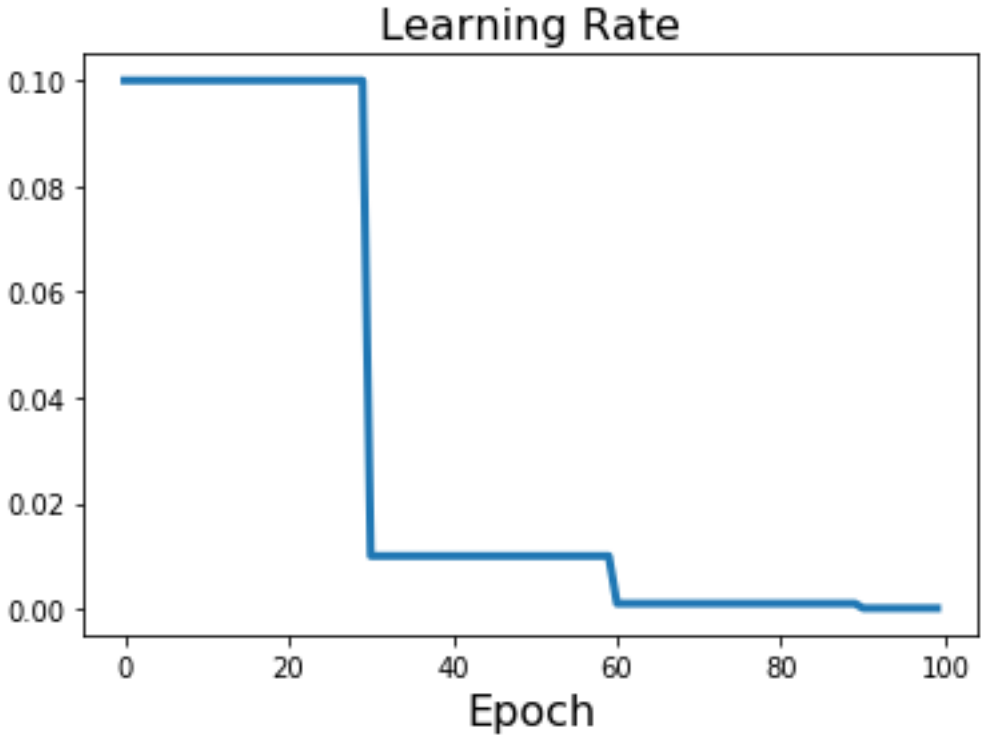
Q: Which one of these learning rates is best to use?

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

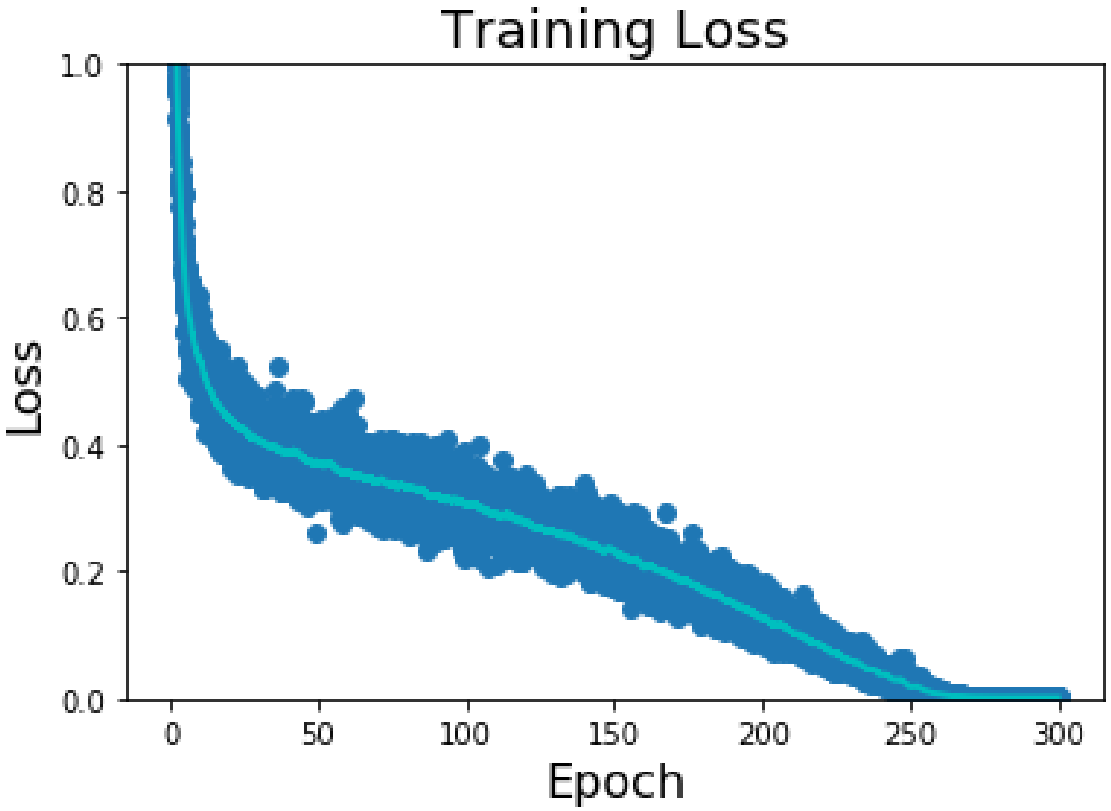
Learning Rate Decay: Step



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

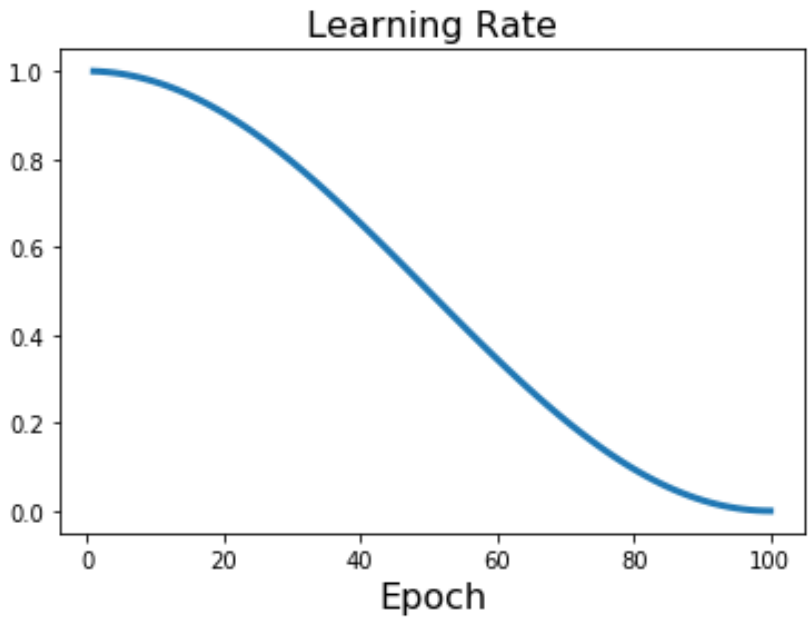


Learning Rate Decay: Cosine



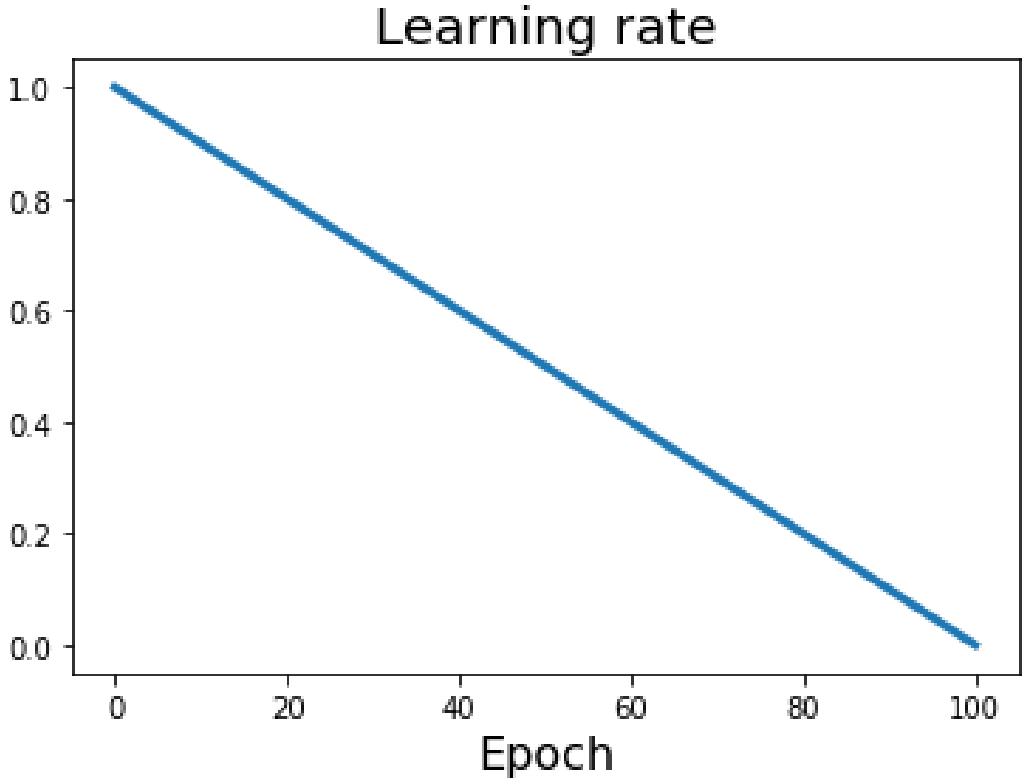
Step: Reduce learning rate at a few fixed points. e.g., for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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



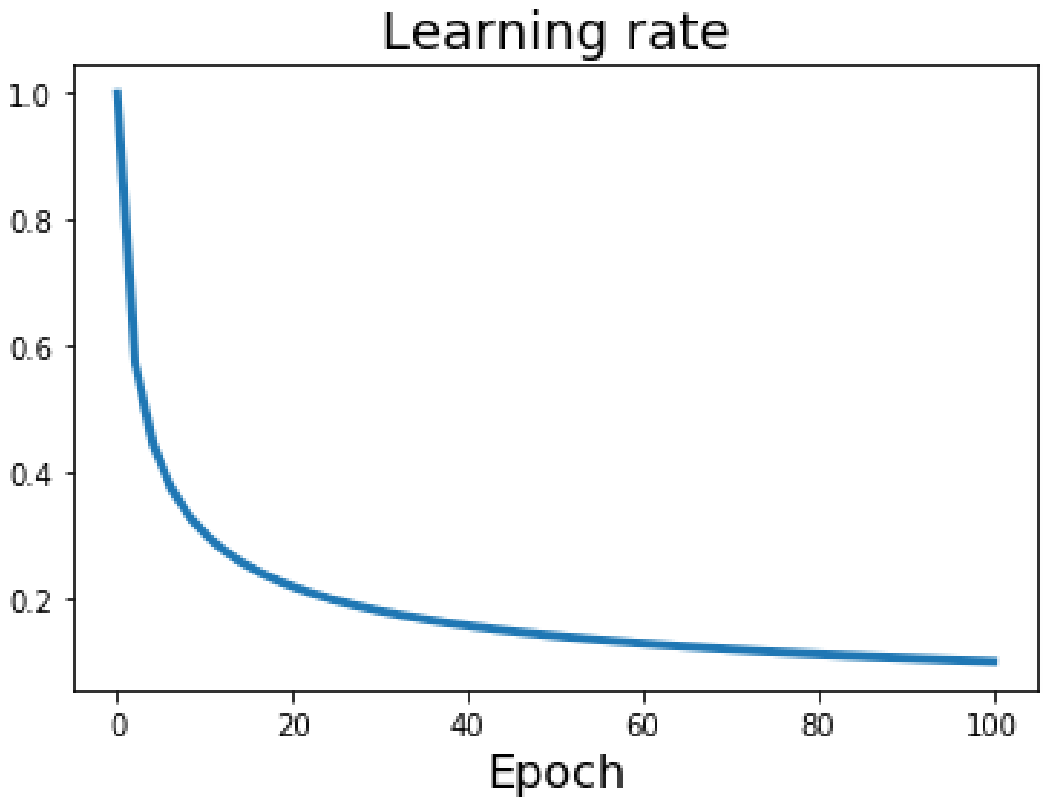
Step: Reduce learning rate at a few fixed points. e.g., for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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.

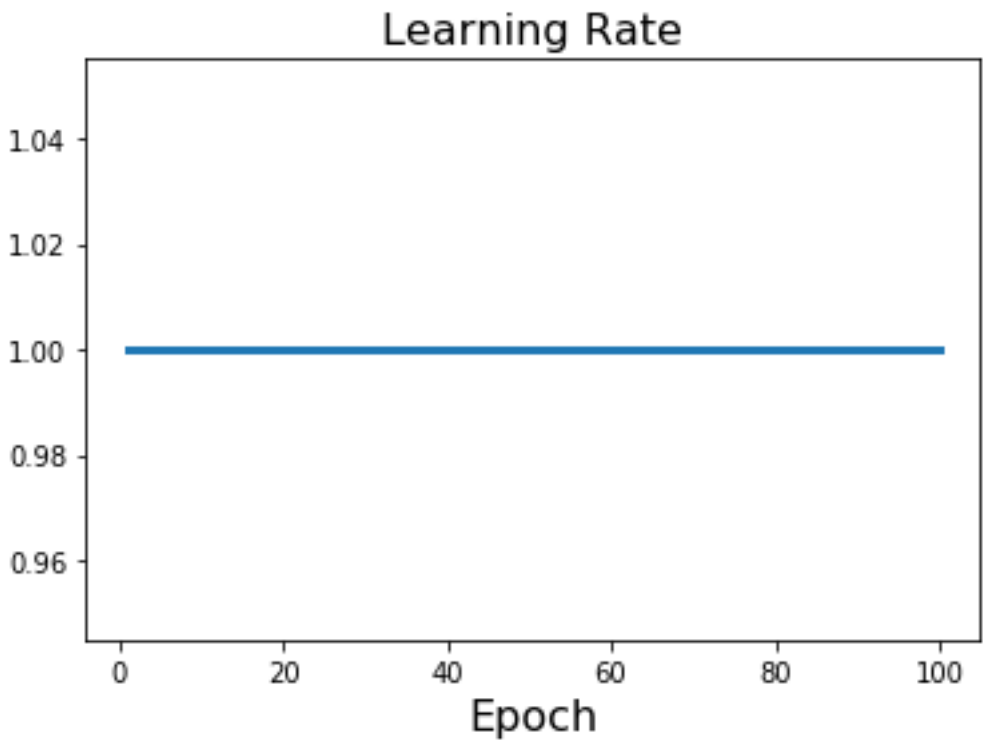
Cosine: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$

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

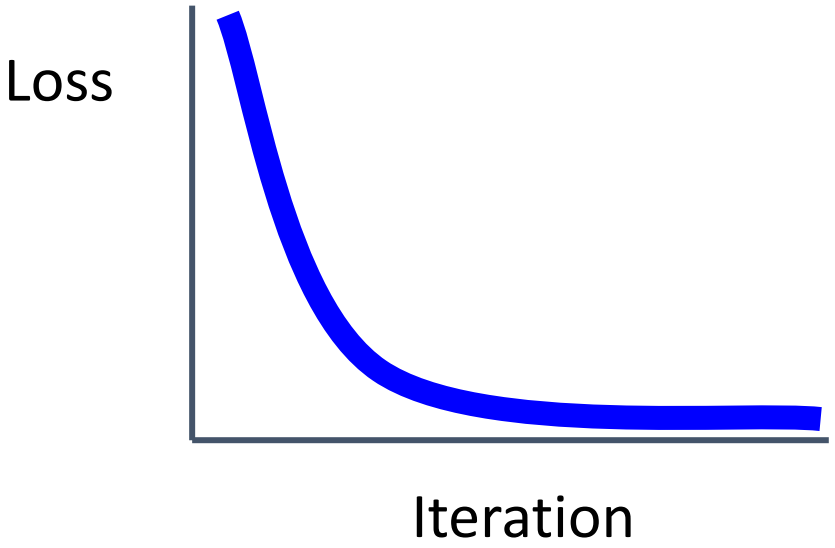
Linear: $\alpha_t = \alpha_0\left(1 - \frac{t}{T}\right)$

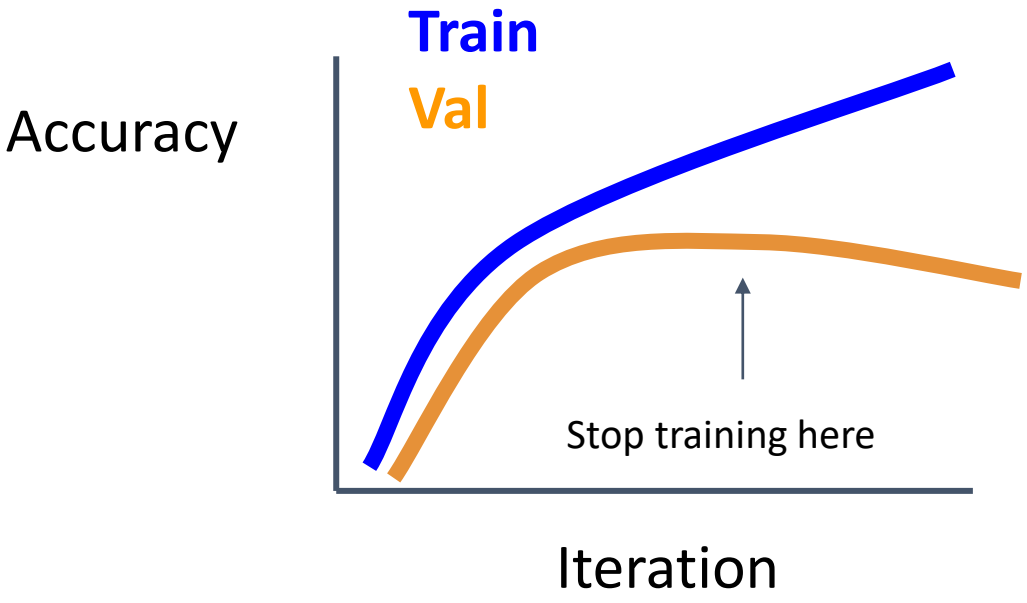
Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

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

(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

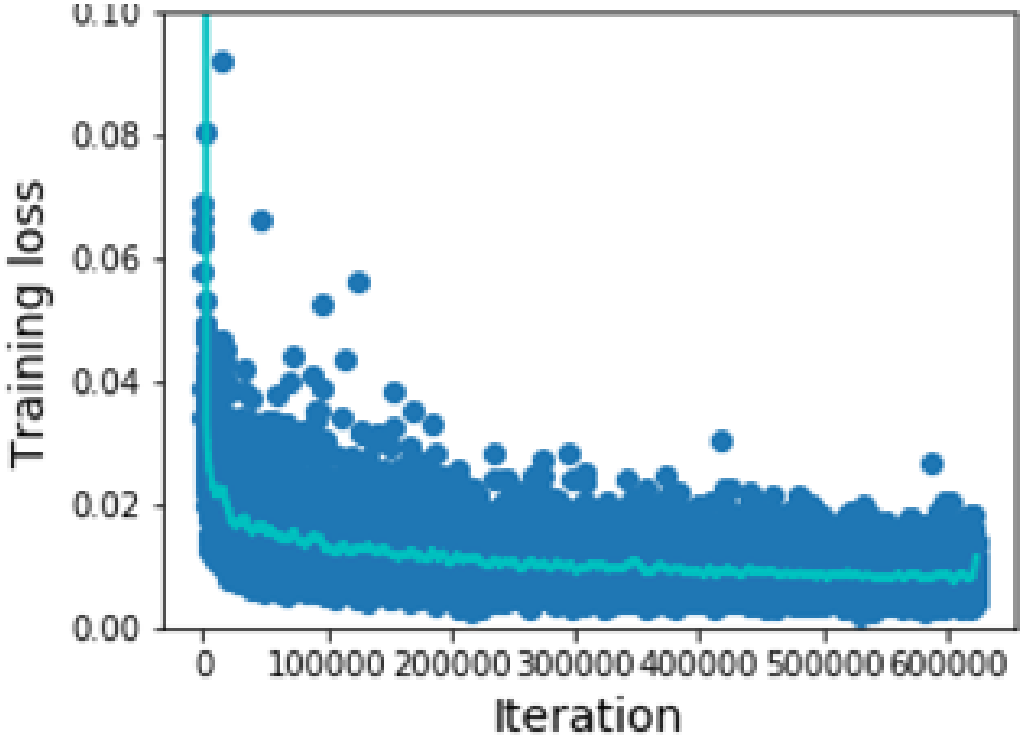
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer

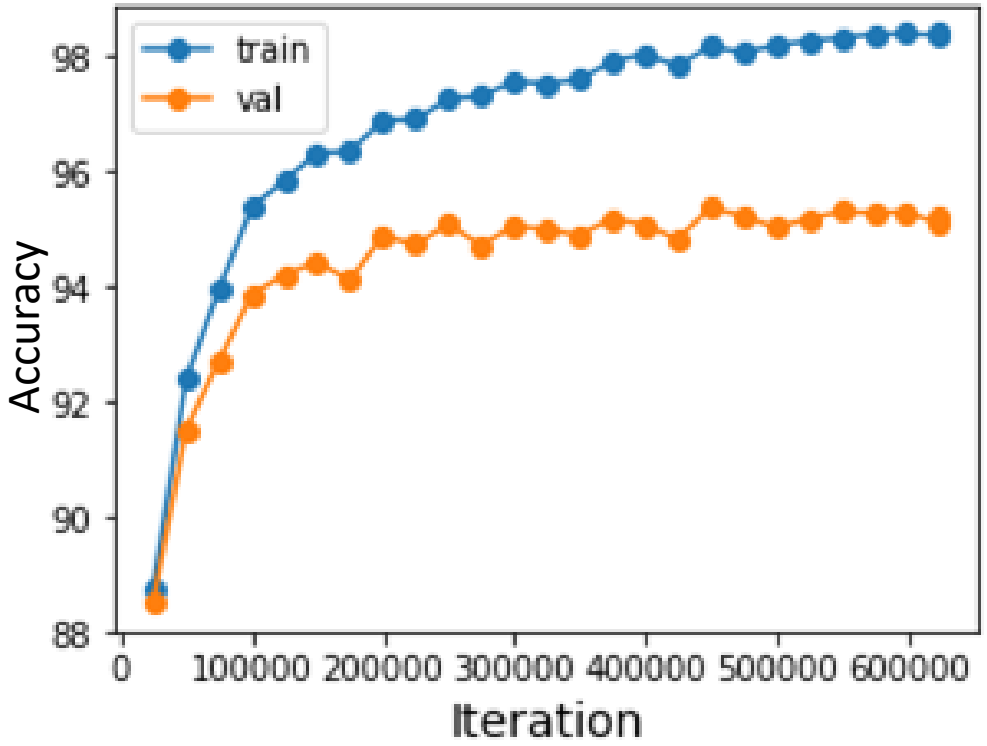
Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Choosing Hyperparameters

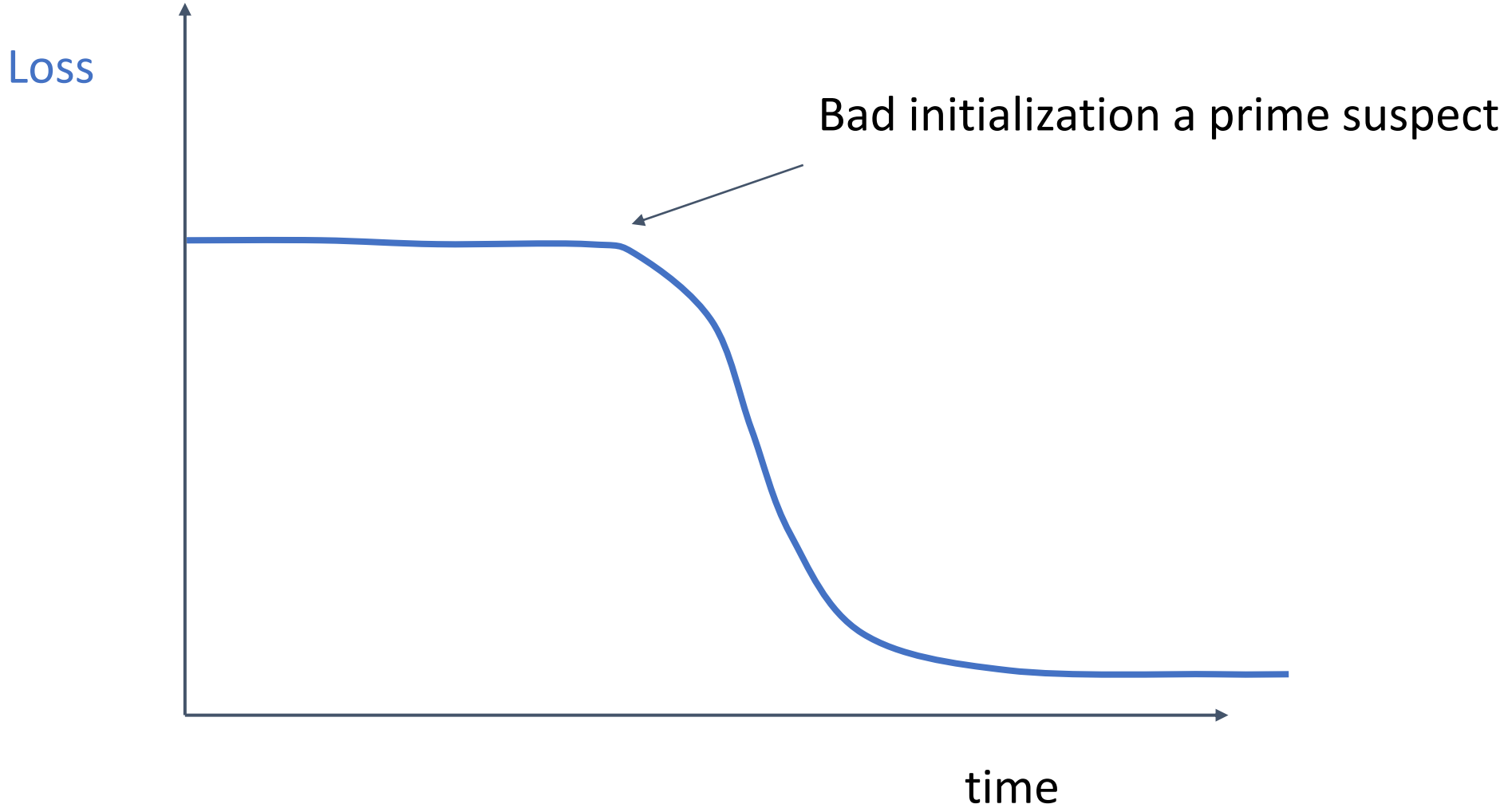
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at learning curves

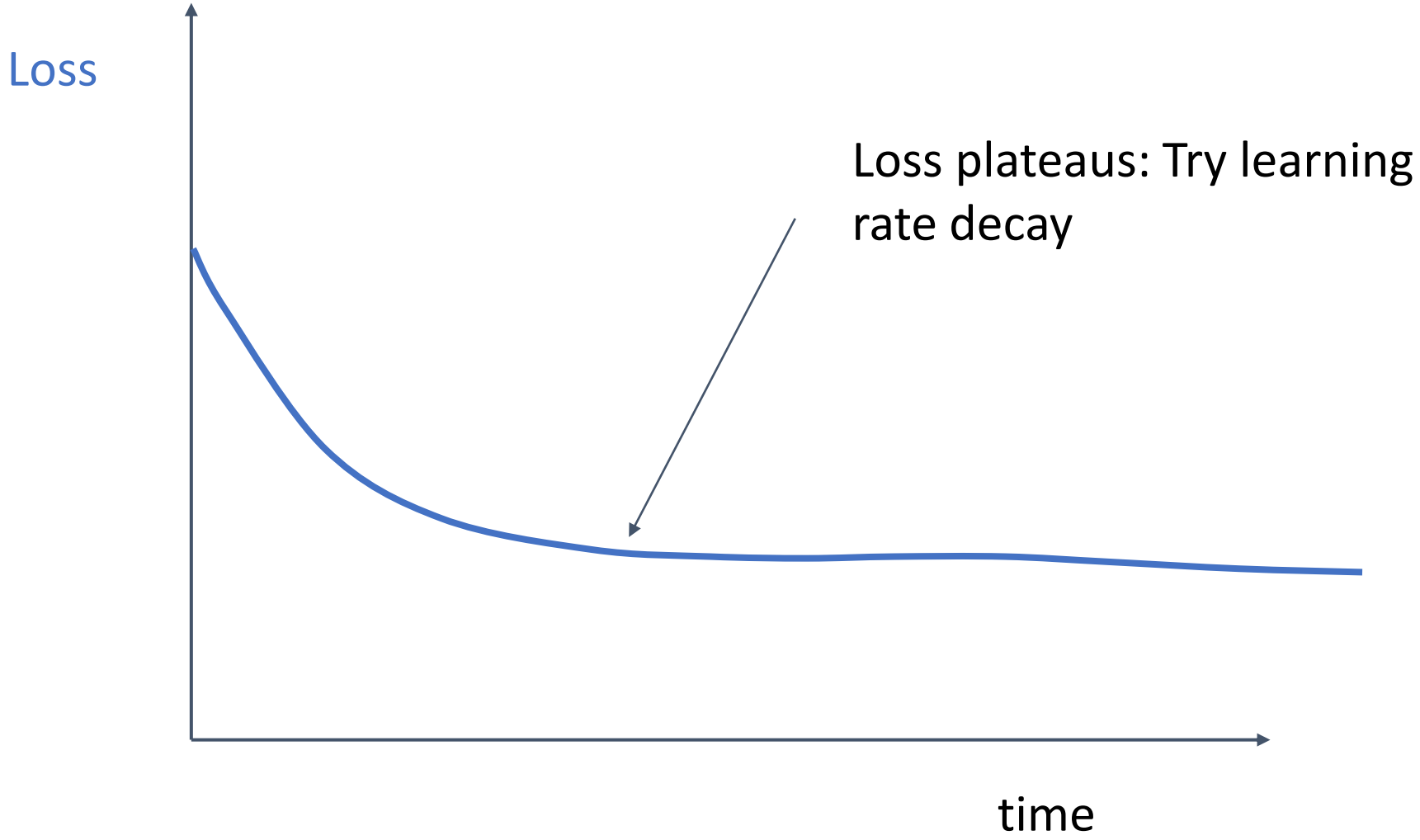
Look at Learning Curves!

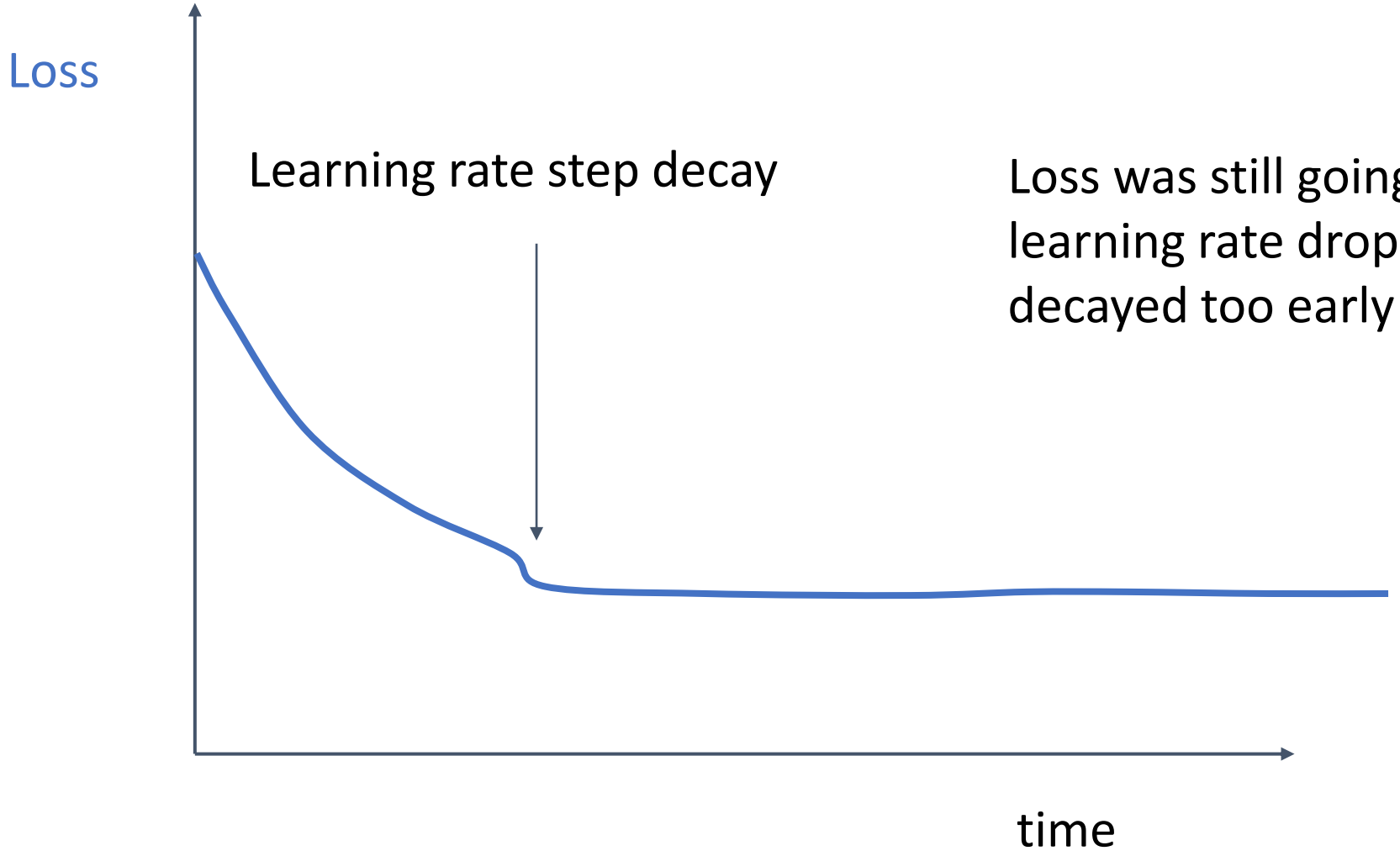




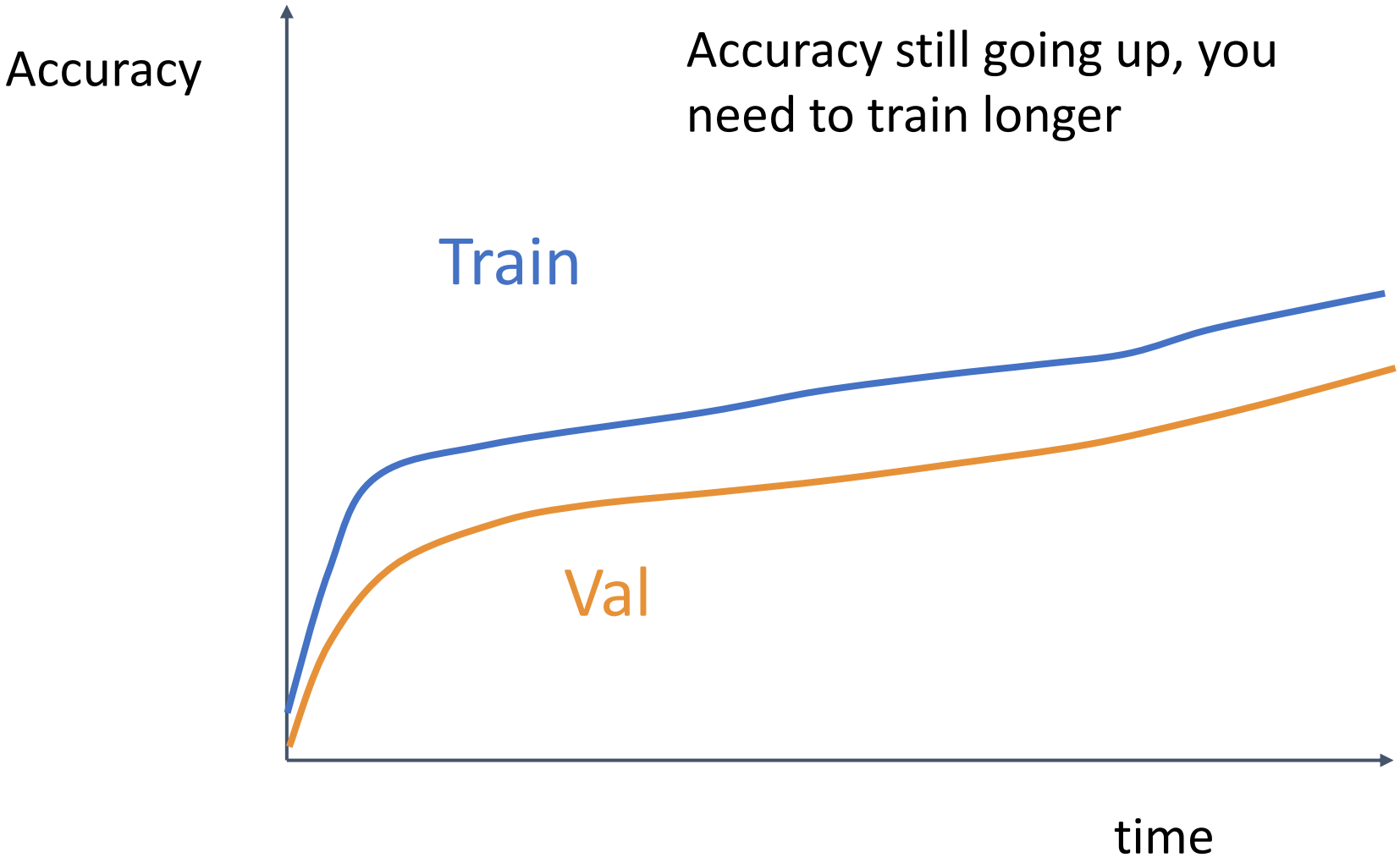
Losses may be noisy, use a scatter plot and also plot moving average to see trends better

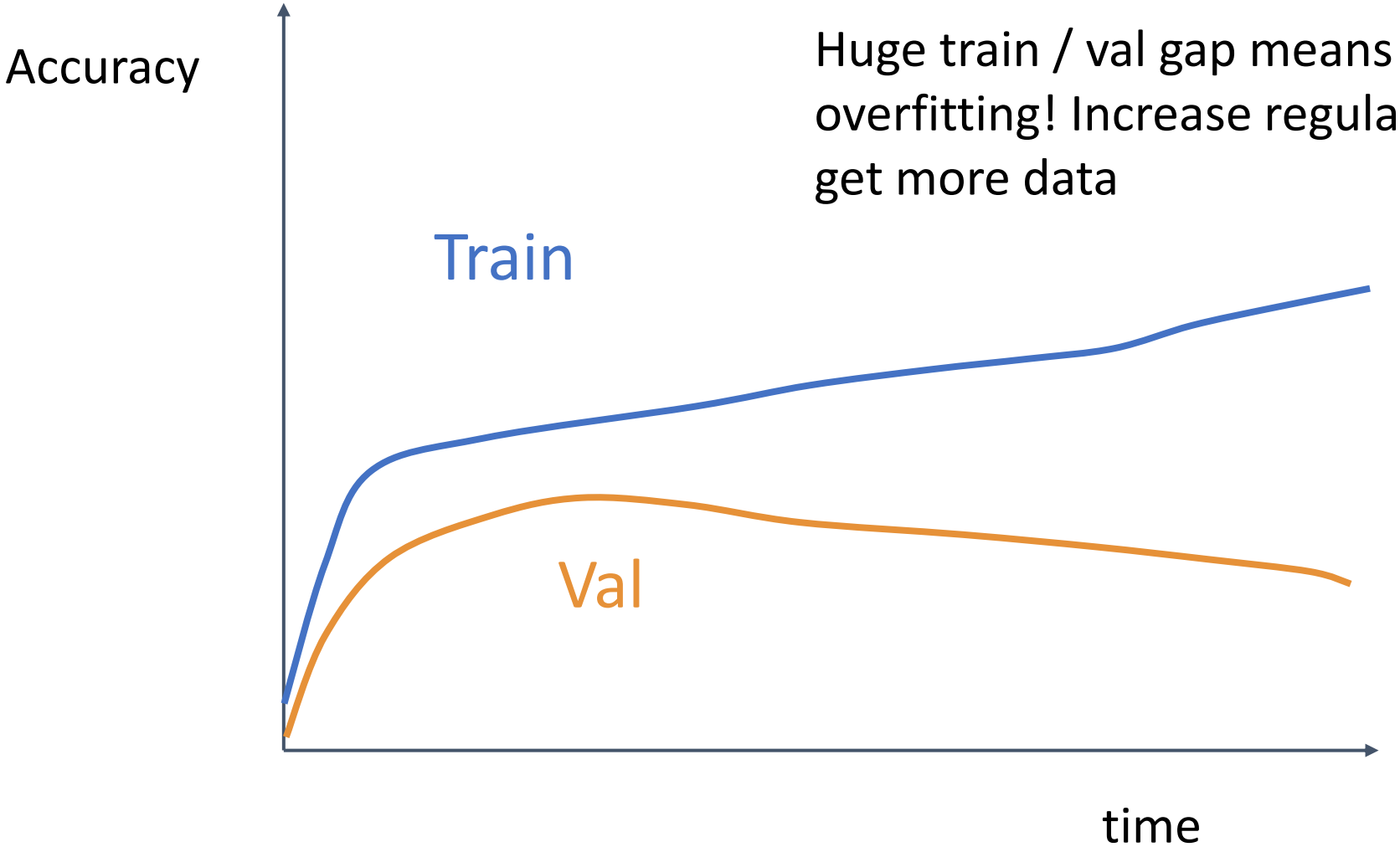

Loss
Bad initialization a prime suspect
time


Loss
Loss plateaus: Try learning rate decay
time

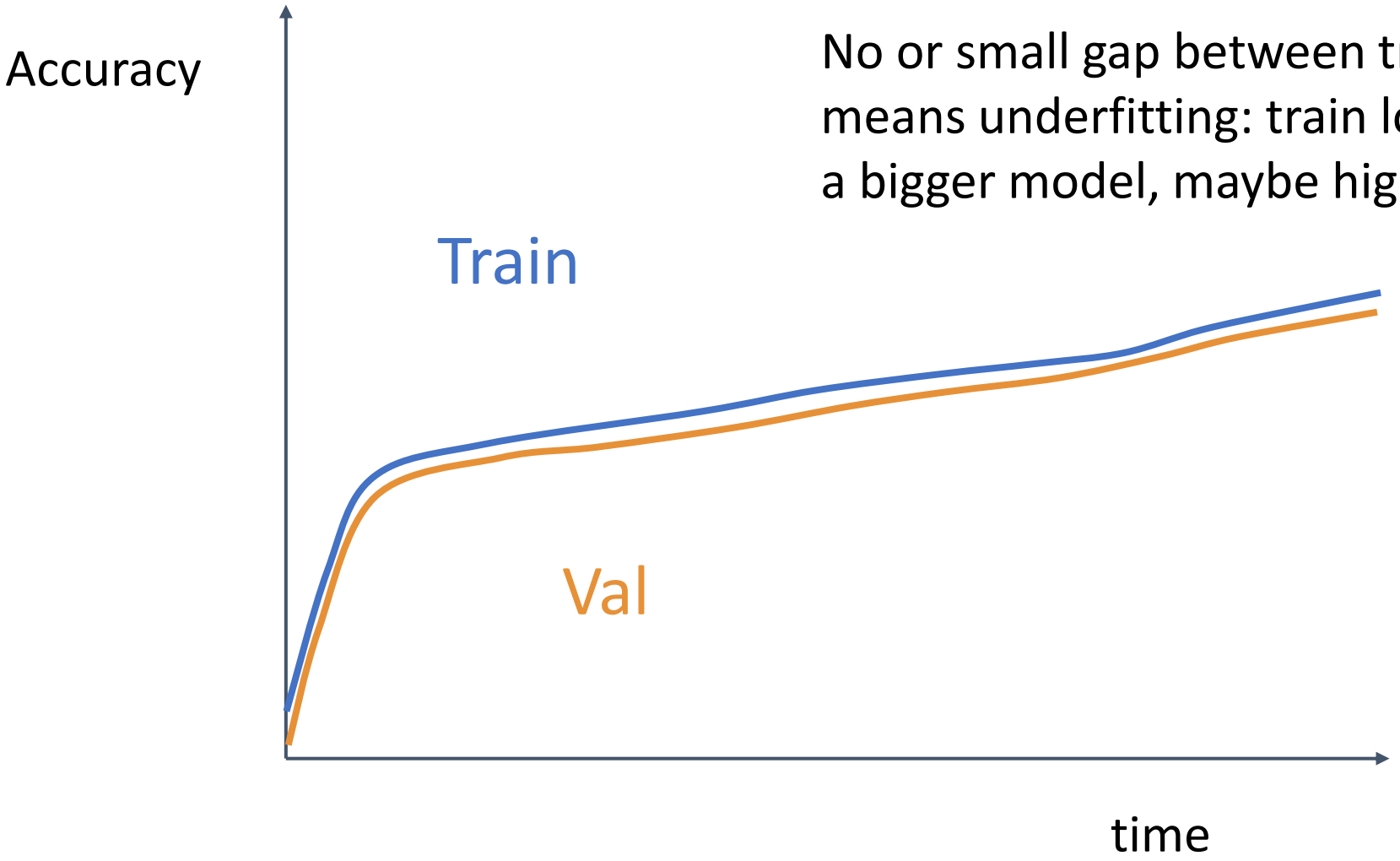


Loss was still going down when learning rate dropped, you decayed too early!


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



Huge train / val gap means overfitting! Increase regularization, get more data



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

Choosing Hyperparameters

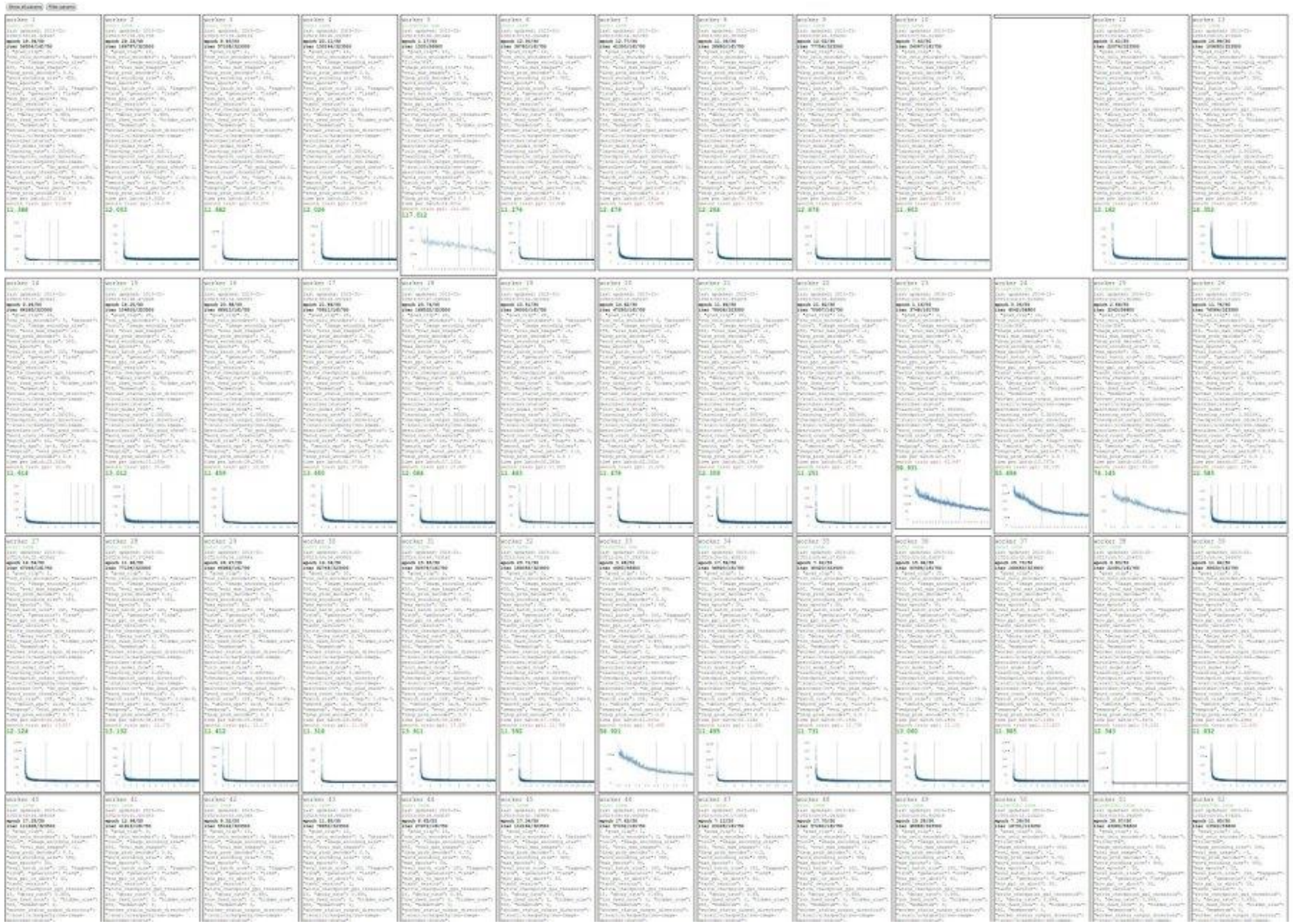
Step 1: Check initial loss
Step 2: Overfit a small sample
Step 3: Find LR that makes loss go down
Step 4: Coarse grid, train for ~1-5 epochs
Step 5: Refine grid, train longer
Step 6: Look at loss curves
Step 7: GOTO step 5

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

Transfer Learning

Transfer Learning

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

Transfer Learning

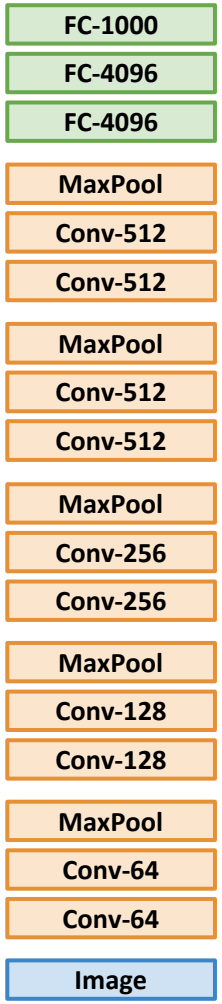
"You need a lot of a data if you want to train/use CNNs"

Not Really

Transfer Learning with CNNs: Feature Extraction

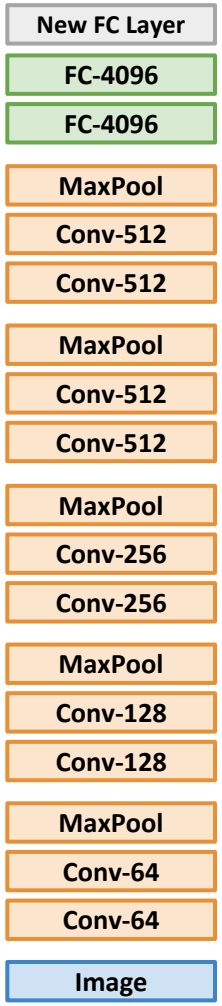
1. Train on ImageNet

2. Use CNN as a feature extractor



Remove last layer

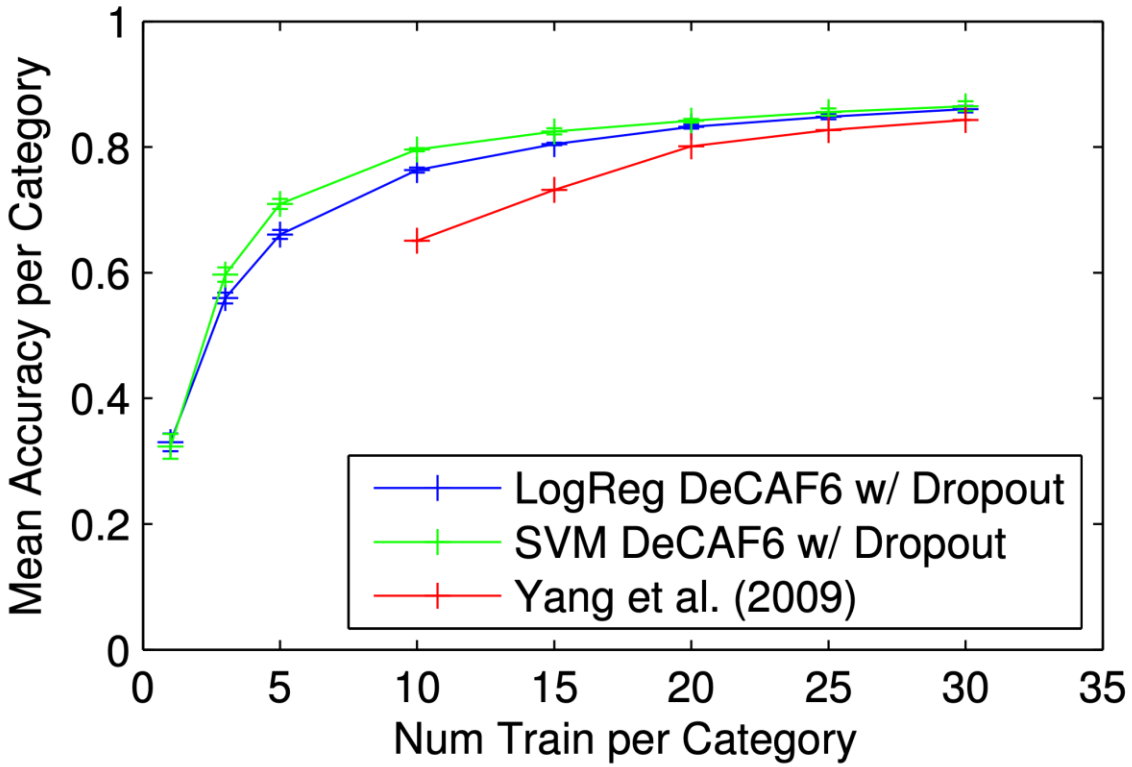
Initialize from ImageNet model



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

Freeze these

Classification on Caltech-101

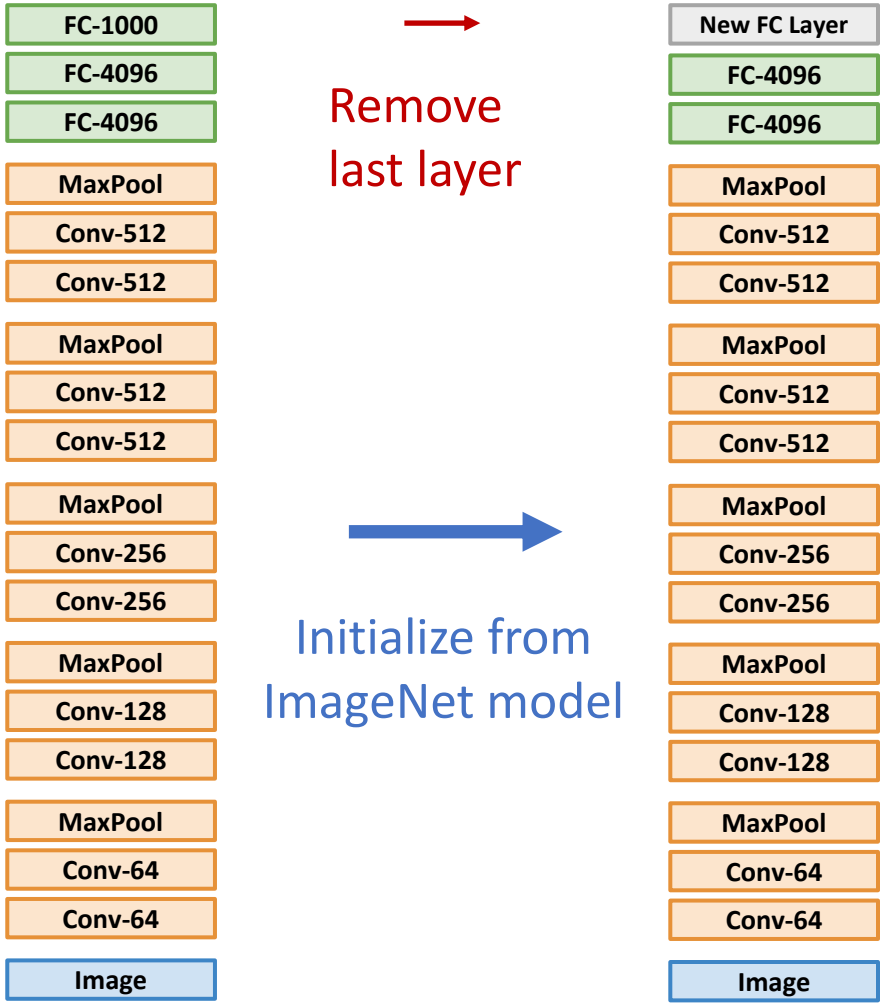


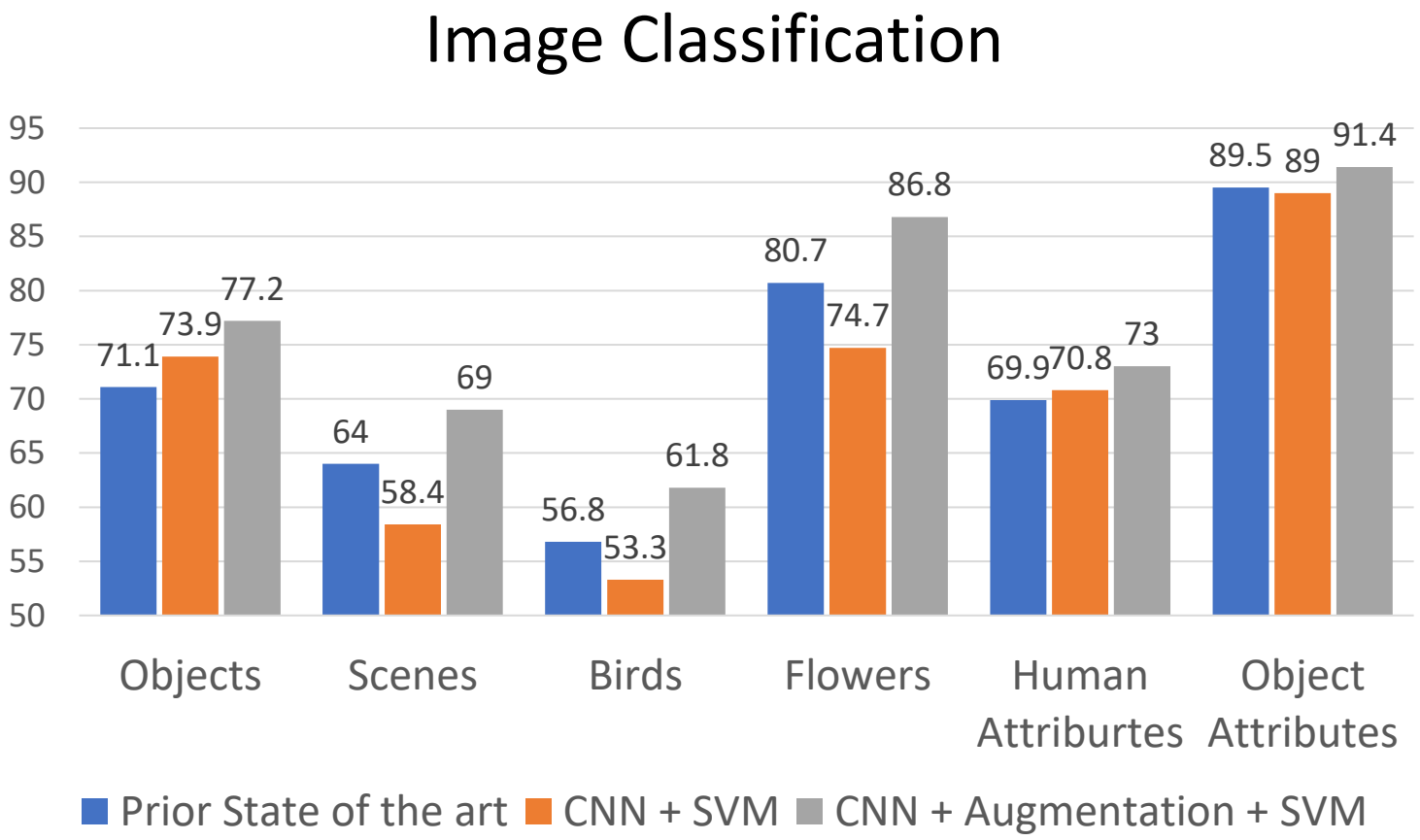
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs: Feature Extraction

1. Train on ImageNet

2. Use CNN as a feature extractor

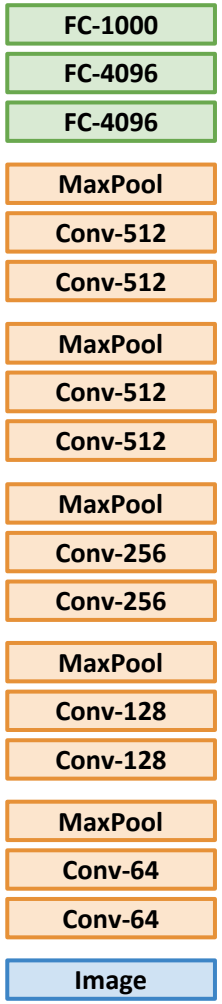




Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs: Fine-Tuning

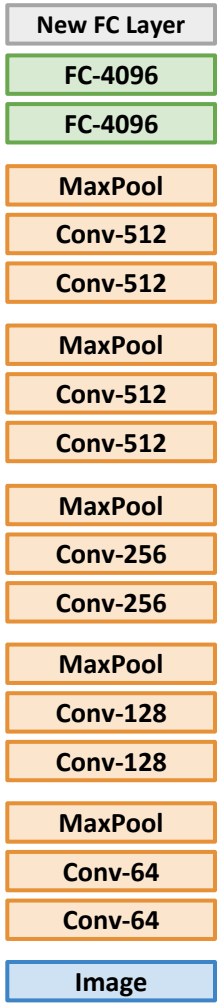
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

2. Fine-tune CNN



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

Continue training entire model for new task

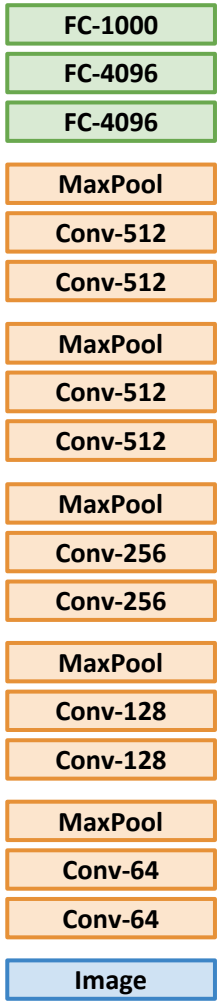
Some tricks:

- 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

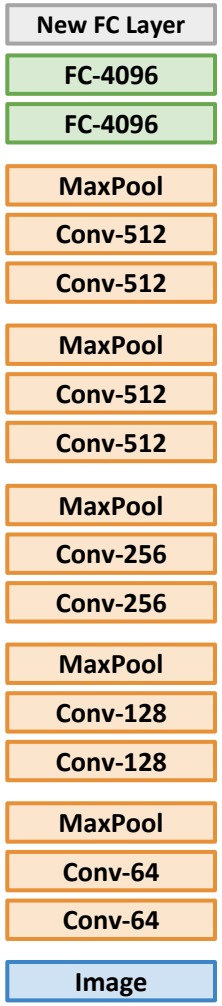
1. Train on ImageNet



Remove last layer

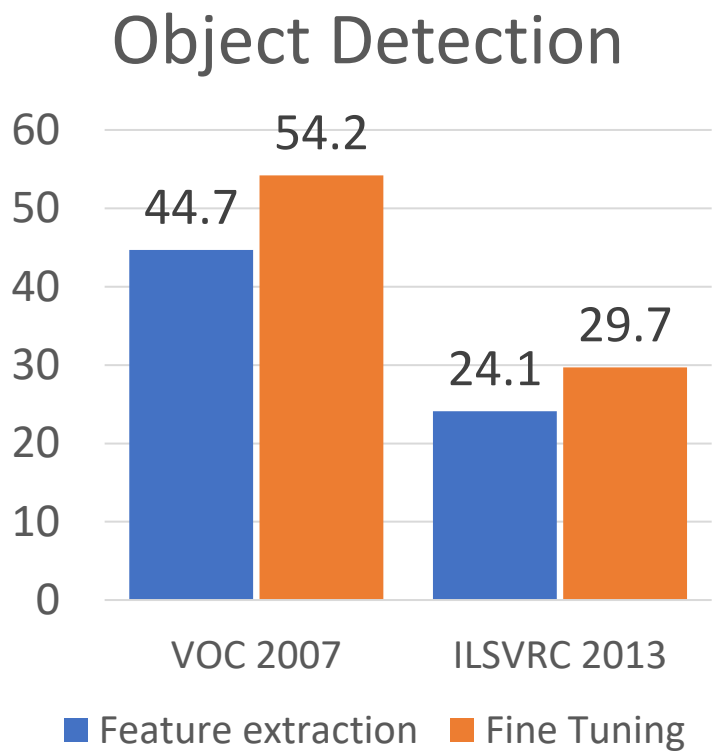
Initialize from ImageNet model

2. Fine-tune CNN



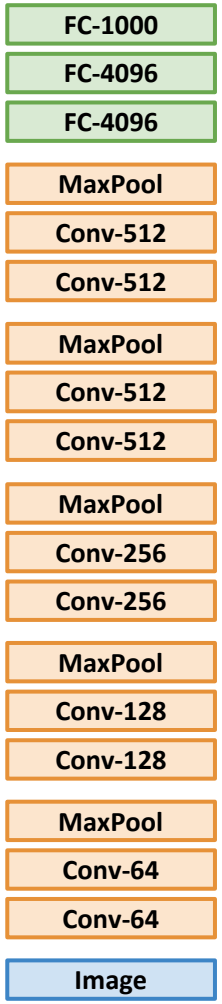
e.g., Add randomly initialized final FC layer for new task

Continue training entire model for new task



Transfer Learning with CNNs: Fine-Tuning

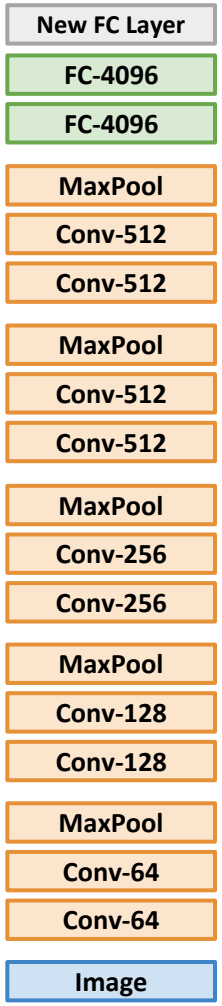
1. Train on ImageNet



Remove last layer

Initialize from ImageNet model

2. Fine-tune CNN



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

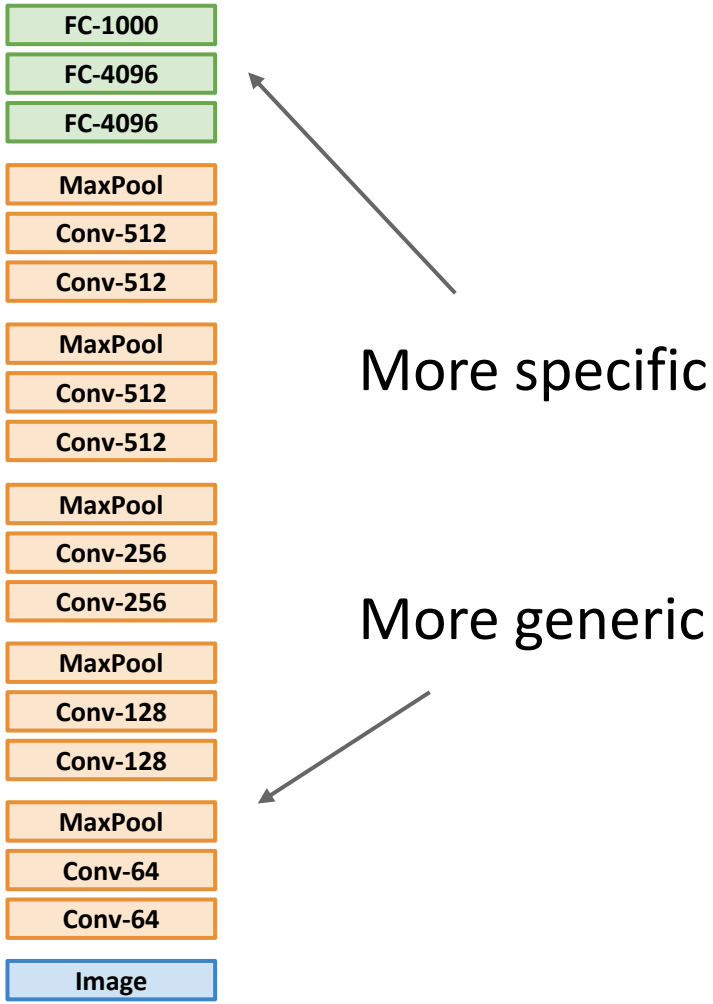
Continue training entire model for new task

Compared with Feature Extraction, Fine-Tuning:

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

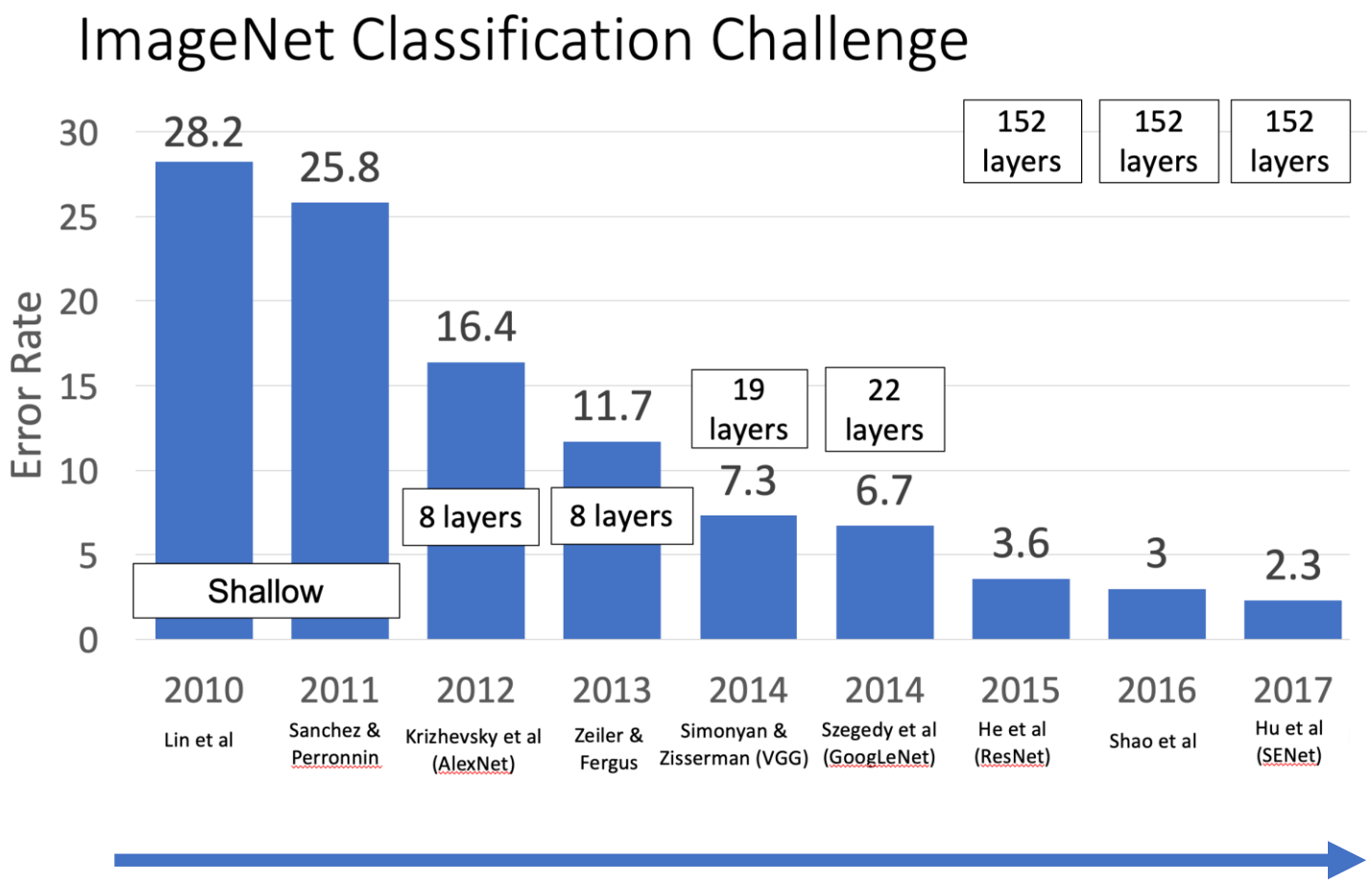
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

Transfer Learning with CNNs



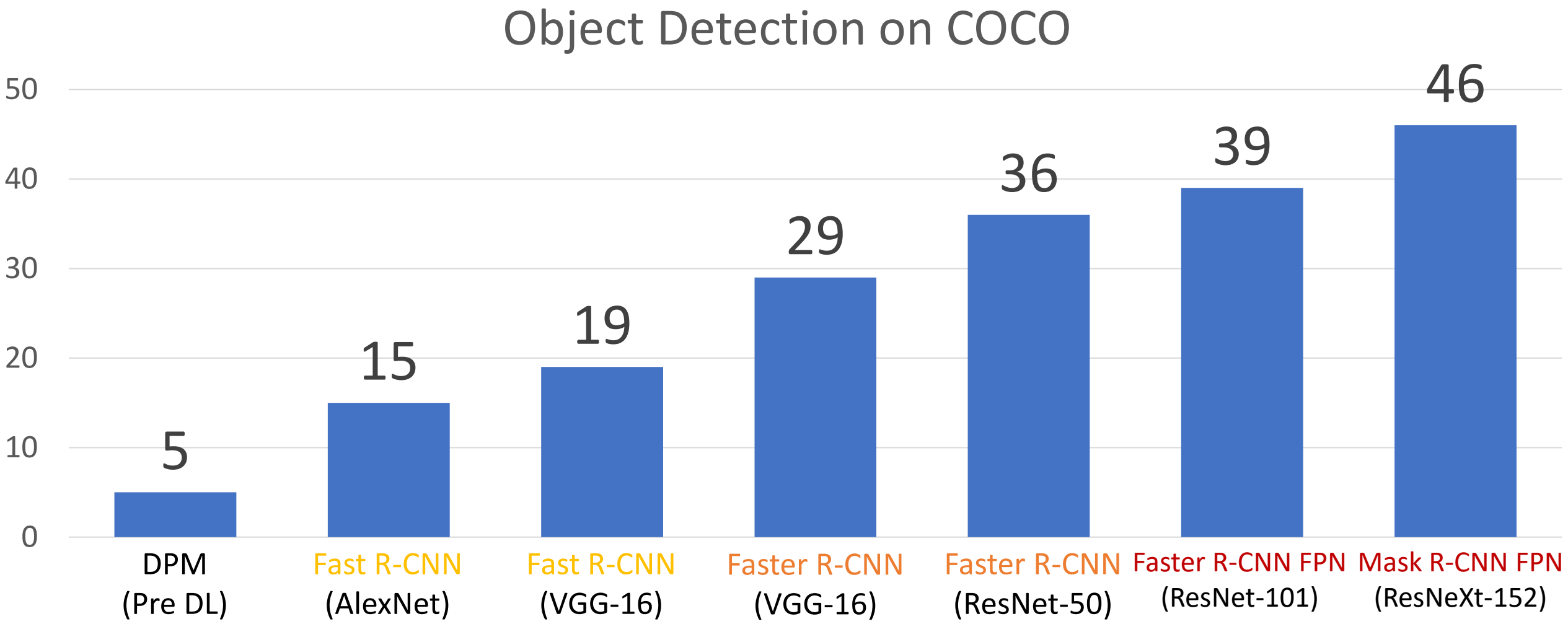
	Dataset similar to ImageNet	**Dataset very different from ImageNet**
very little data (10s to 100s)	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data (100s to 1000s)	Finetune a few layers	Finetune a larger number of layers

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

Recap

- Activation functions
- Data preprocessing
- Weight initialization
- Regularization
- Optimization
- Hyperparameter optimization
- Transfer learning

Next: Recurrent Neural Networks