DeepRob

Lecture 10
Training Neural Networks I
University of Michigan | Department of Robotics

Project 2—Updates

- Instructions available on the website
  - Here: deeprob.org/projects/project2/
- two-layer neural network and R-CNN/two-stage detectors
- **Due Thursday, February 22th 11:59 PM EST**

Recap: Object Detection

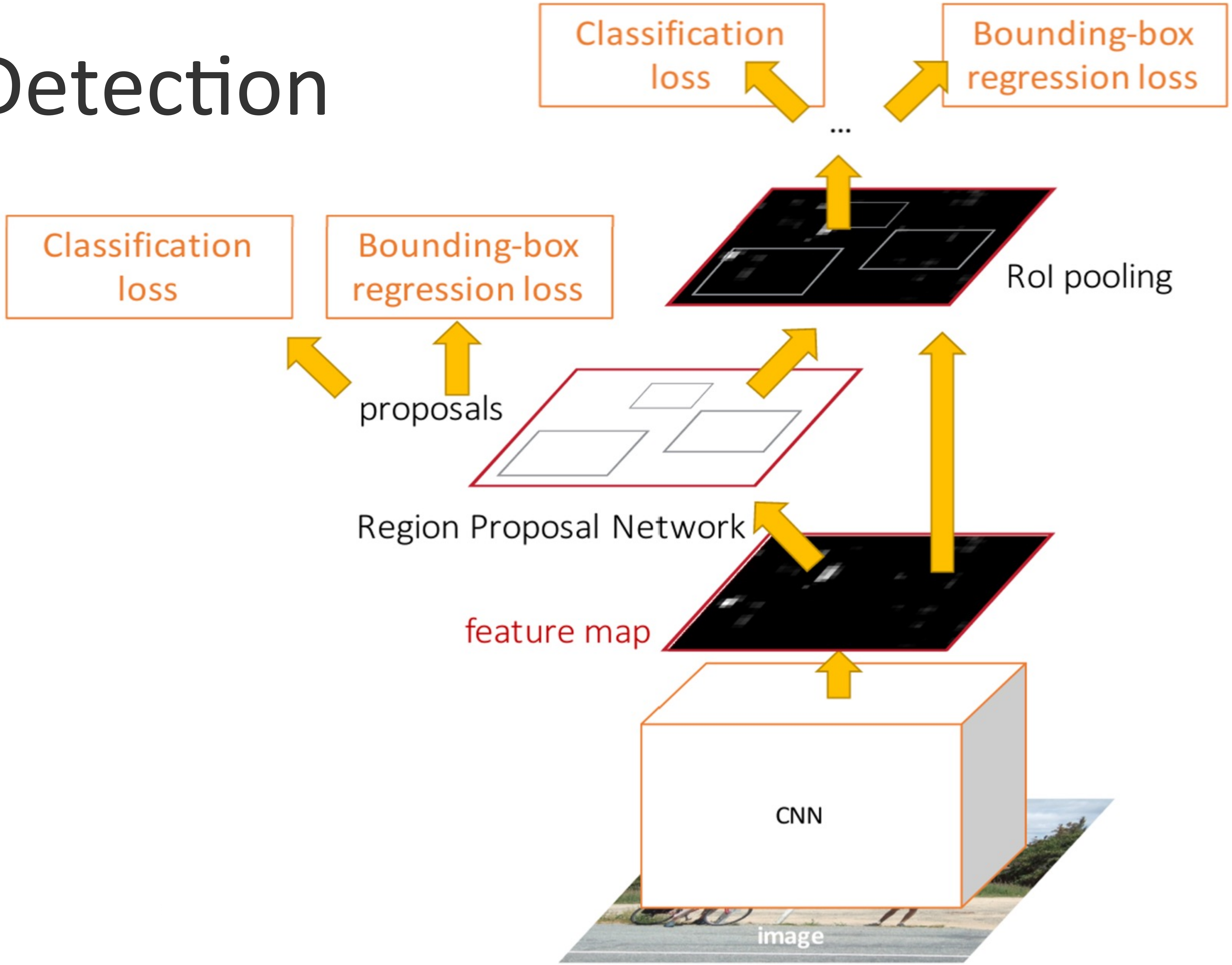
R-CNN

Fast R-CNN

Faster R-CNN

Mask R-CNN

...



Final Project Paper Selection Survey

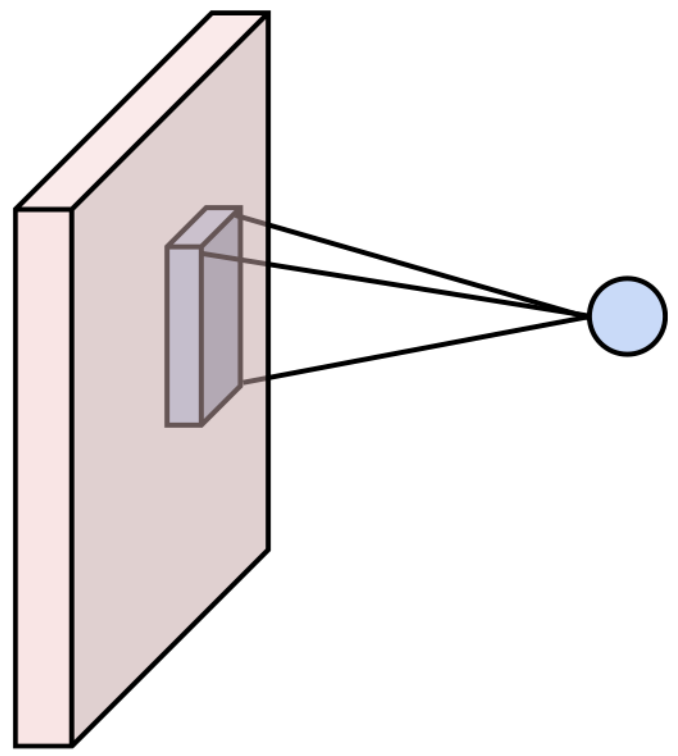
- Published as a gradescope quiz, 1 point
  - To gauge your areas of interest
  - Used for forming teams

https://deeprob.org/w24/papers/

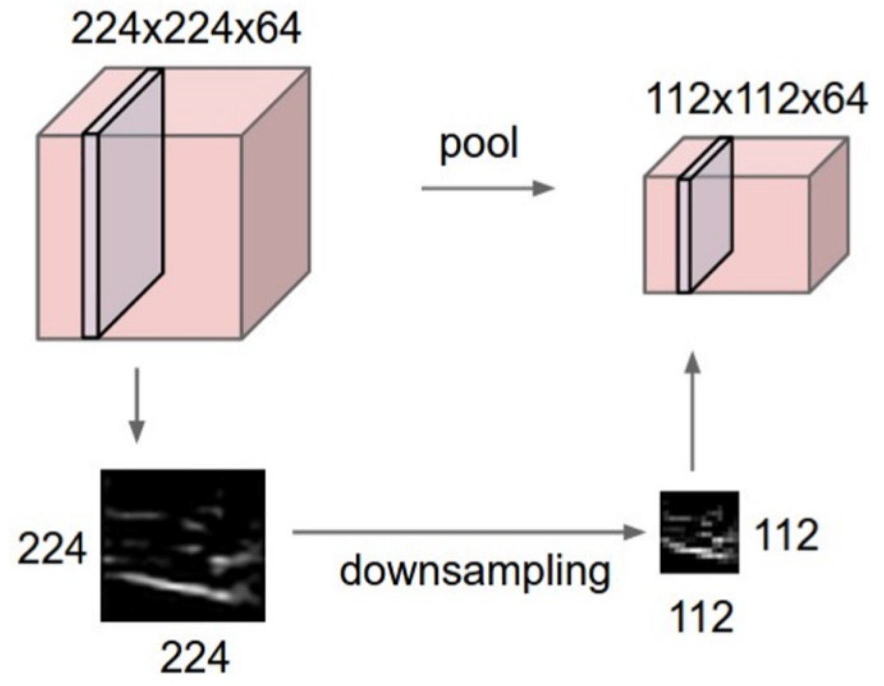
- **Due February 22rd 11:59 PM EST**

Components of Convolutional Networks

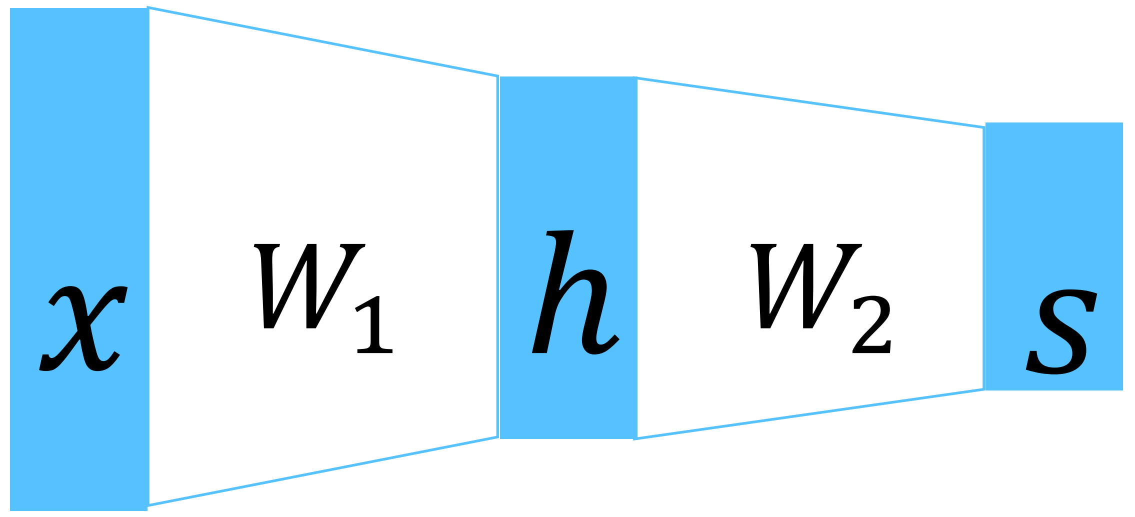
Convolution Layers



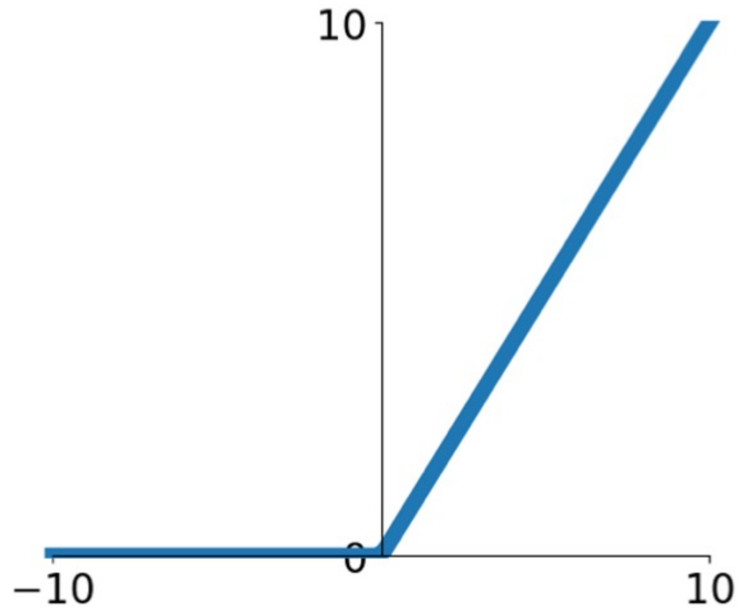
Pooling Layers



Fully-Connected Layers



Activation Function



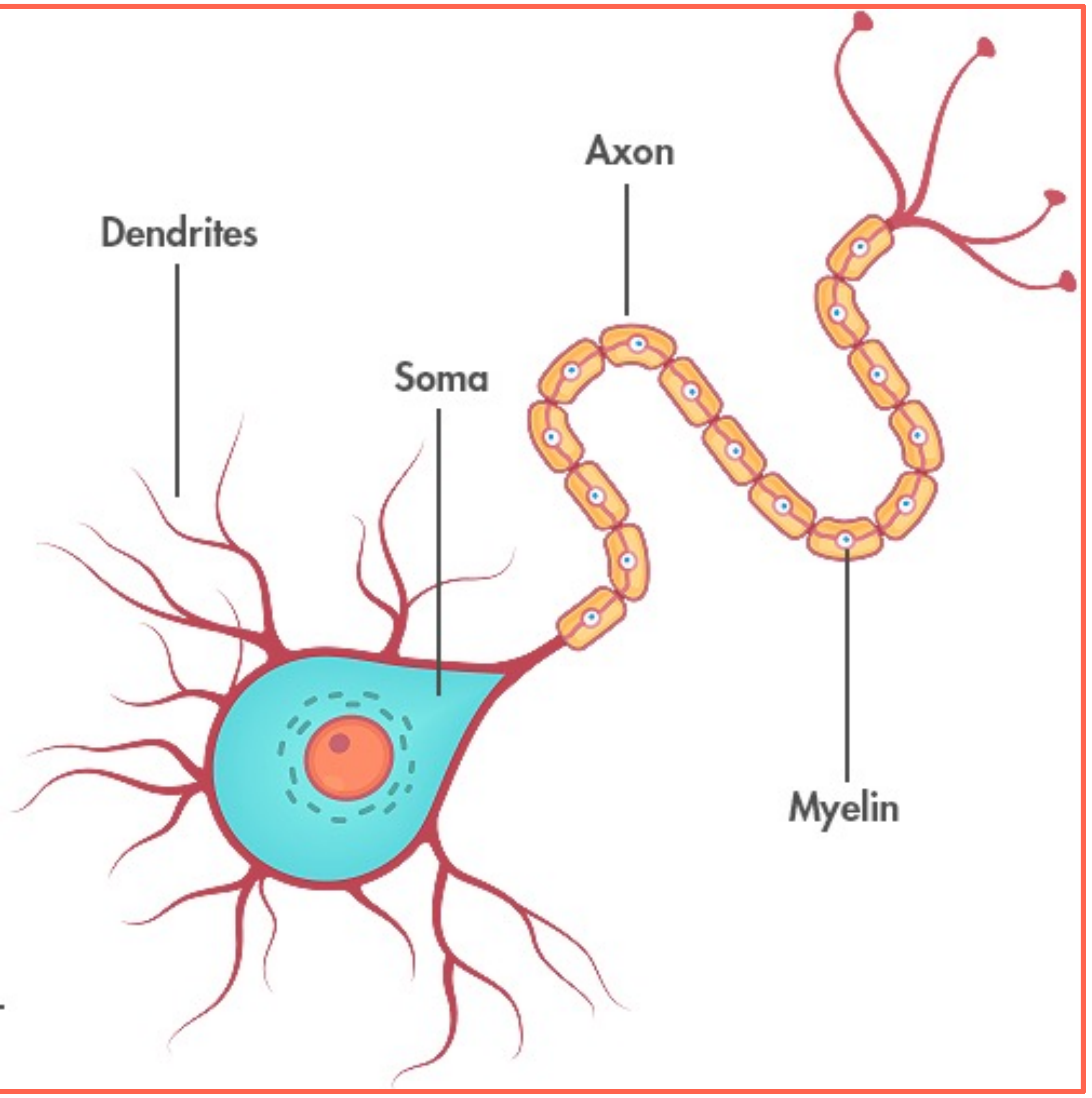
Normalization

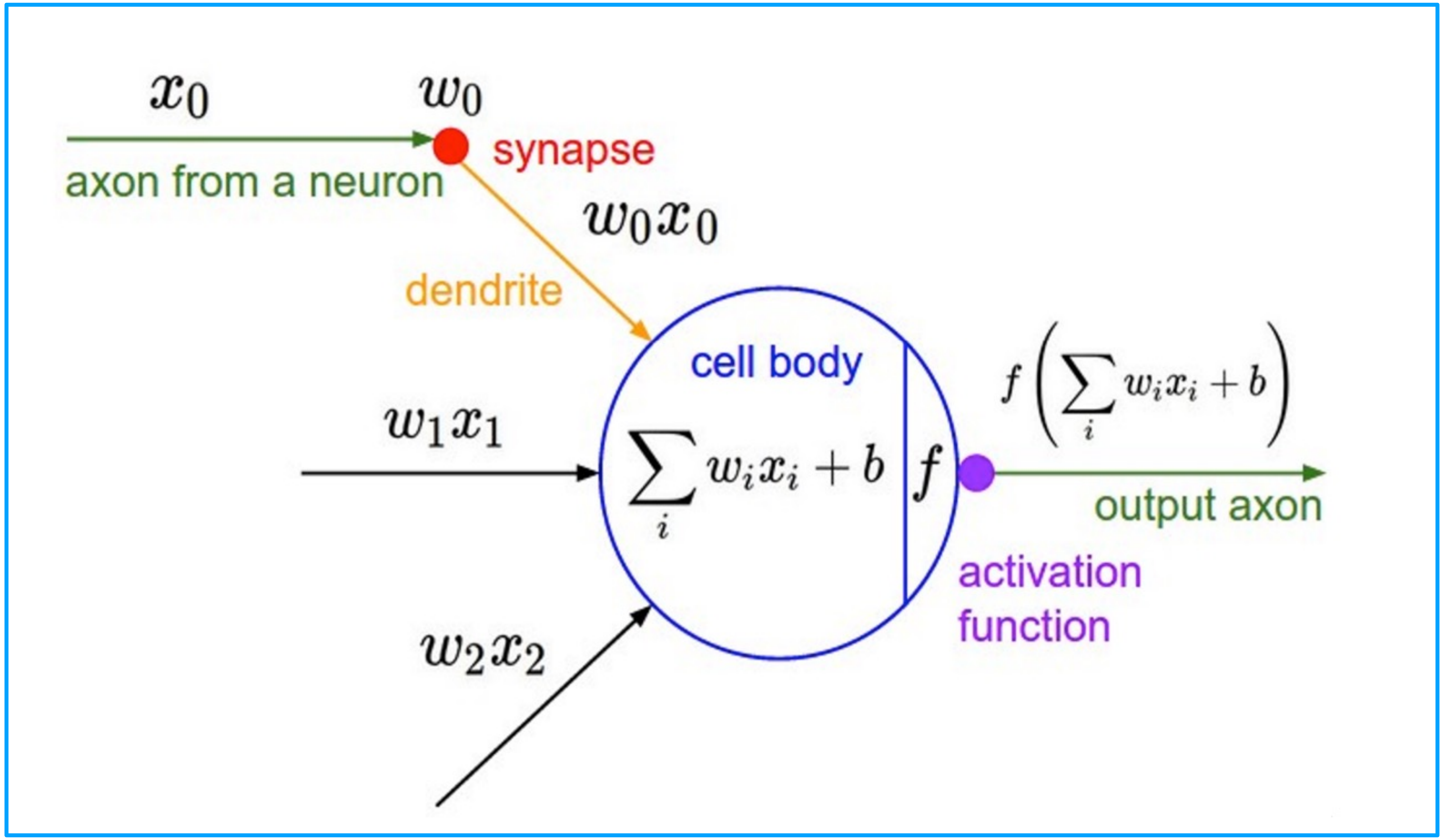
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Overview

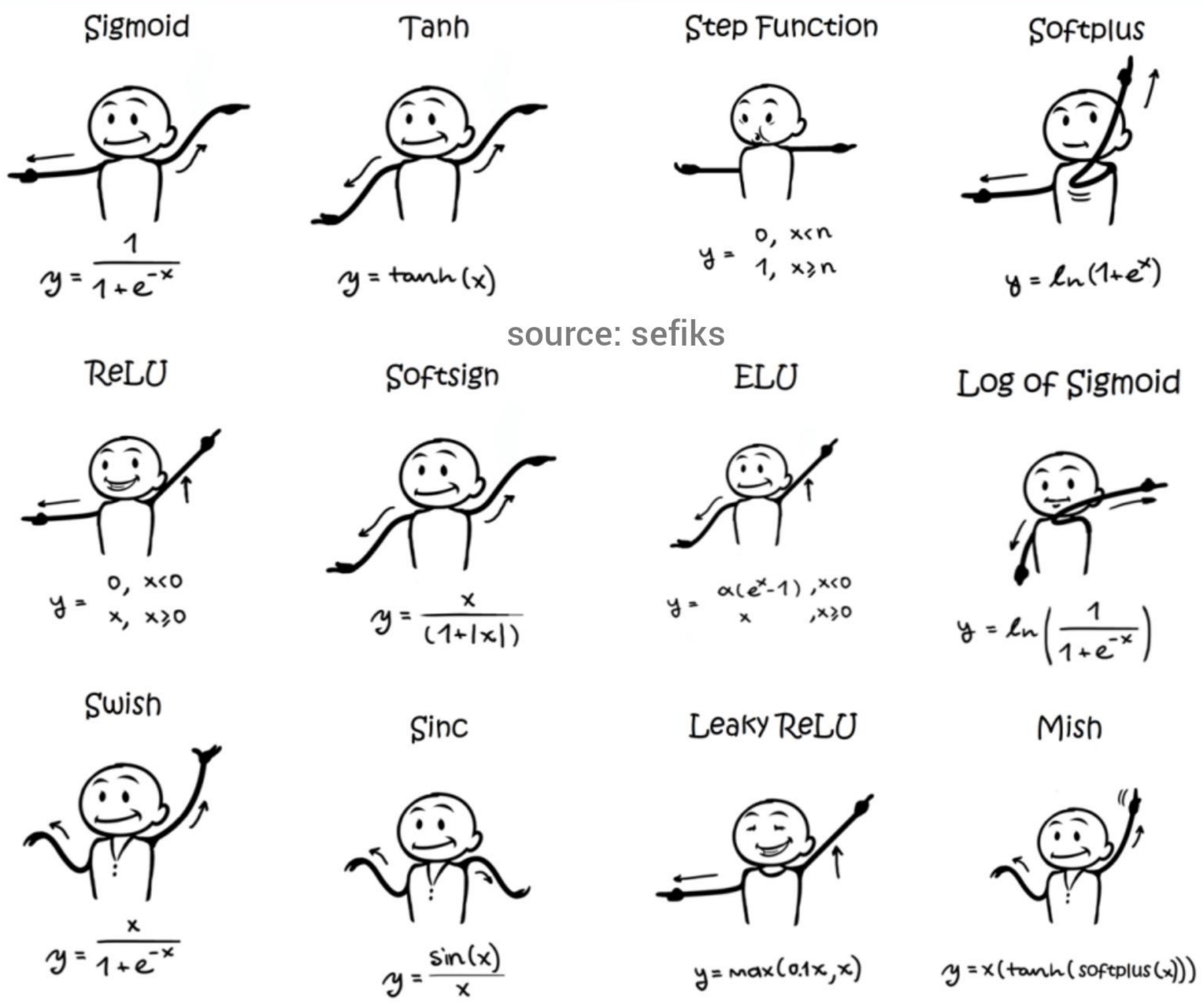
1. One time setup:
   - Activation functions, data preprocessing, weight initialization, regularization
2. Training dynamics:
   - Learning rate schedules; large-batch training; hyperparameter optimization
3. After training:
   - Model ensembles, transfer learning

Activation Functions



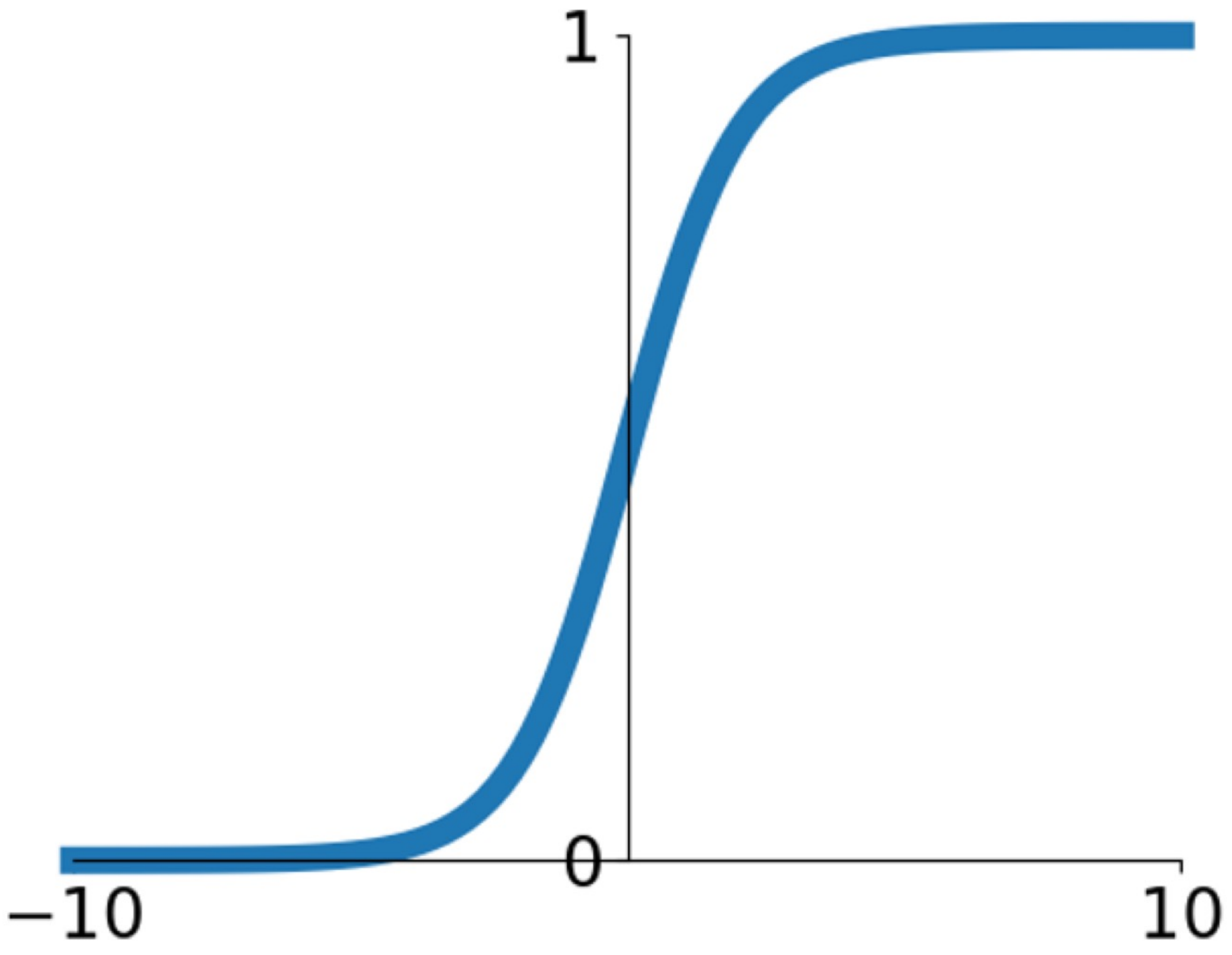


Dance Moves of Deep Learning Activation Functions



https://sefiks.com/2020/02/02/dance-moves-of-deep-learning-activation-functions/

Activation Functions: Sigmoid

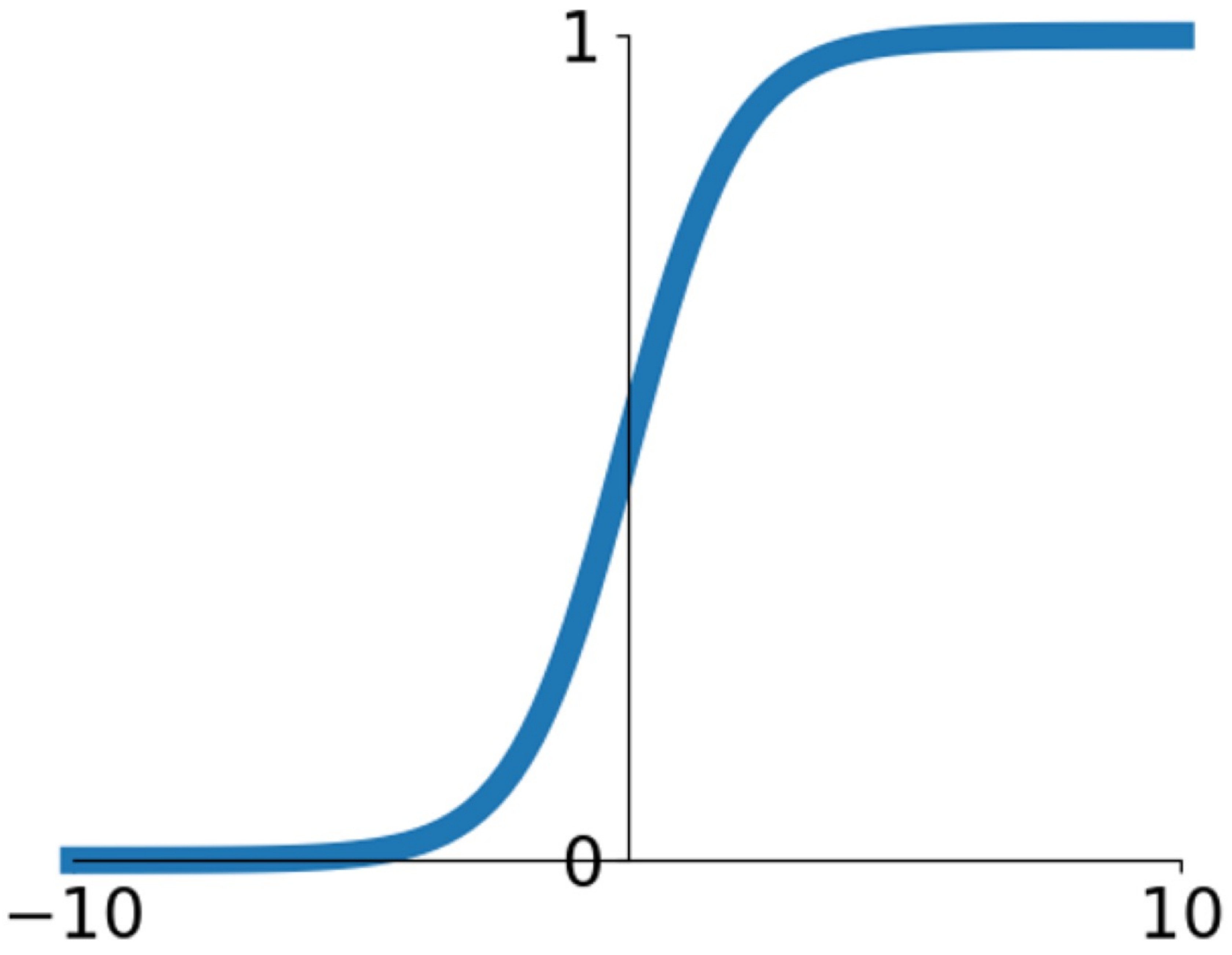


Sigmoid

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

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

Activation Functions: Sigmoid



Sigmoid

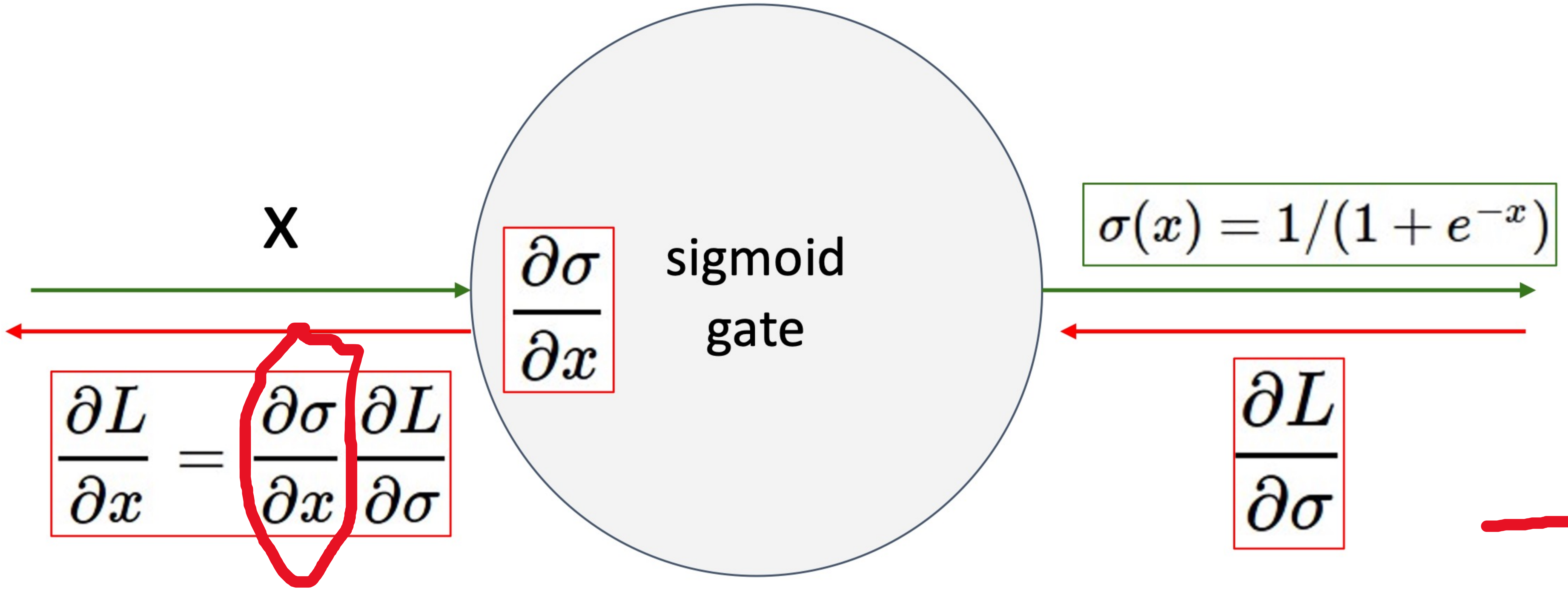
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

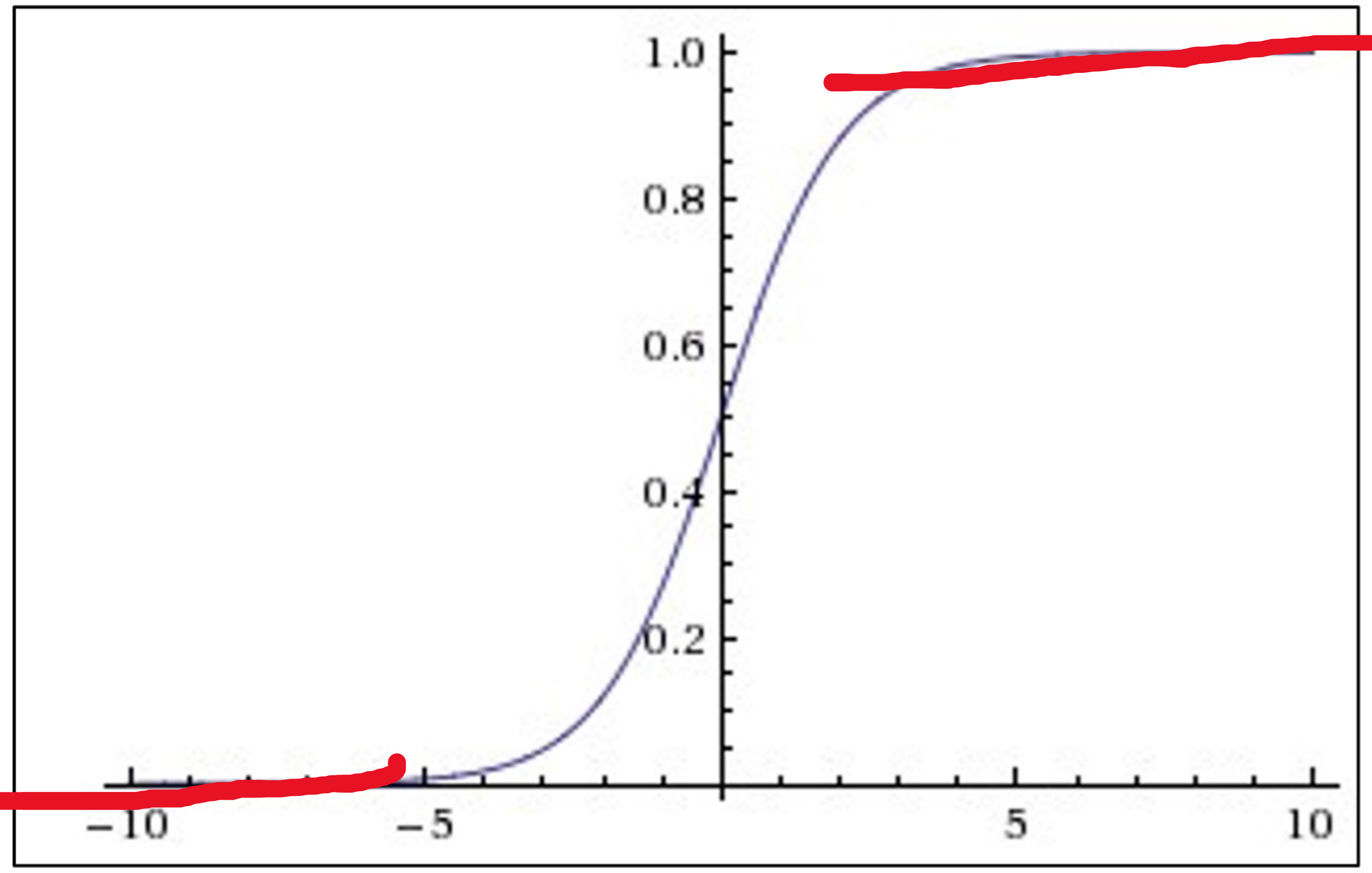
- Squashes numbers to range [0, 1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. **Saturated neurons “kill” the gradients**

Activation Functions: Sigmoid

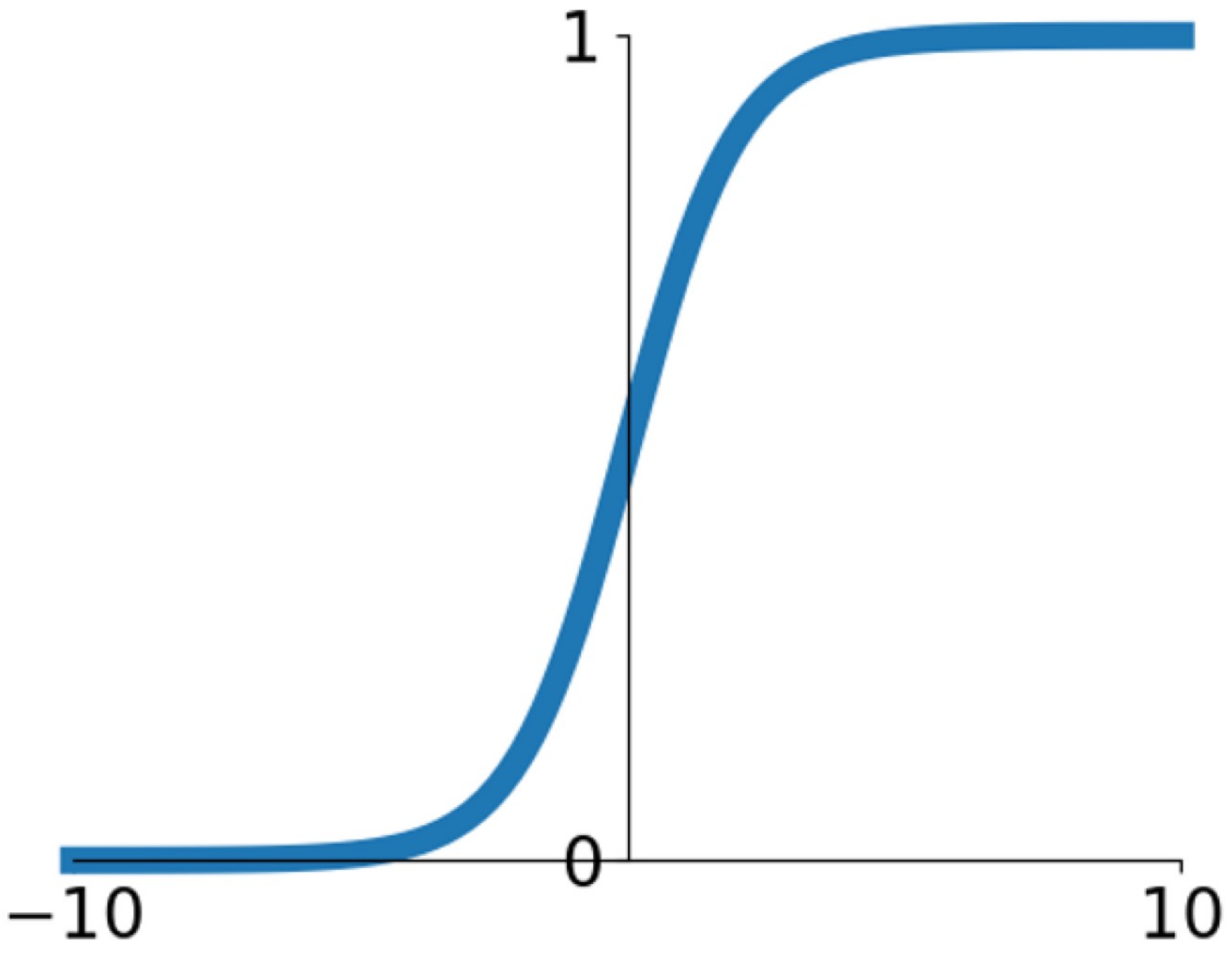




- What happens when x = -10?
- What happens when x = 10?

"sigmoid saturation problem"

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**

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

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

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

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

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

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

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

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

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

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

Local gradient · Upstream gradient

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

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

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

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)
$w^{(\ell)}, b^{(\ell)}$ 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 $\partial L / \partial h_i^{(\ell)}$

Local gradient · Upstream gradient

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

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

Activation Functions: Sigmoid

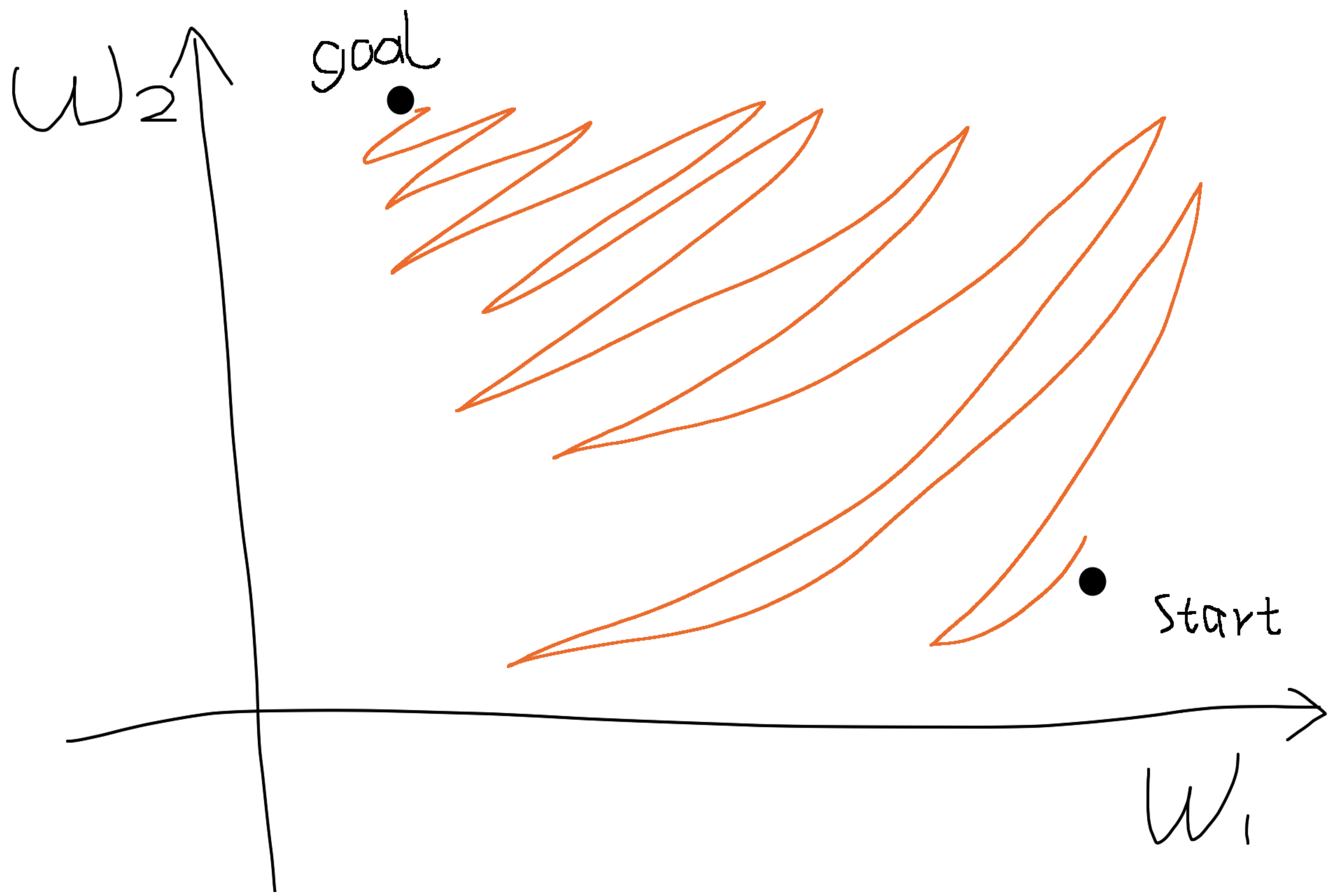
Consider what happens when nonlinearity is always positive

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

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)
$w^{(\ell)}, b^{(\ell)}$ 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 $\partial L / \partial h_i^{(\ell)}$



"zig-zagging dynamics"

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

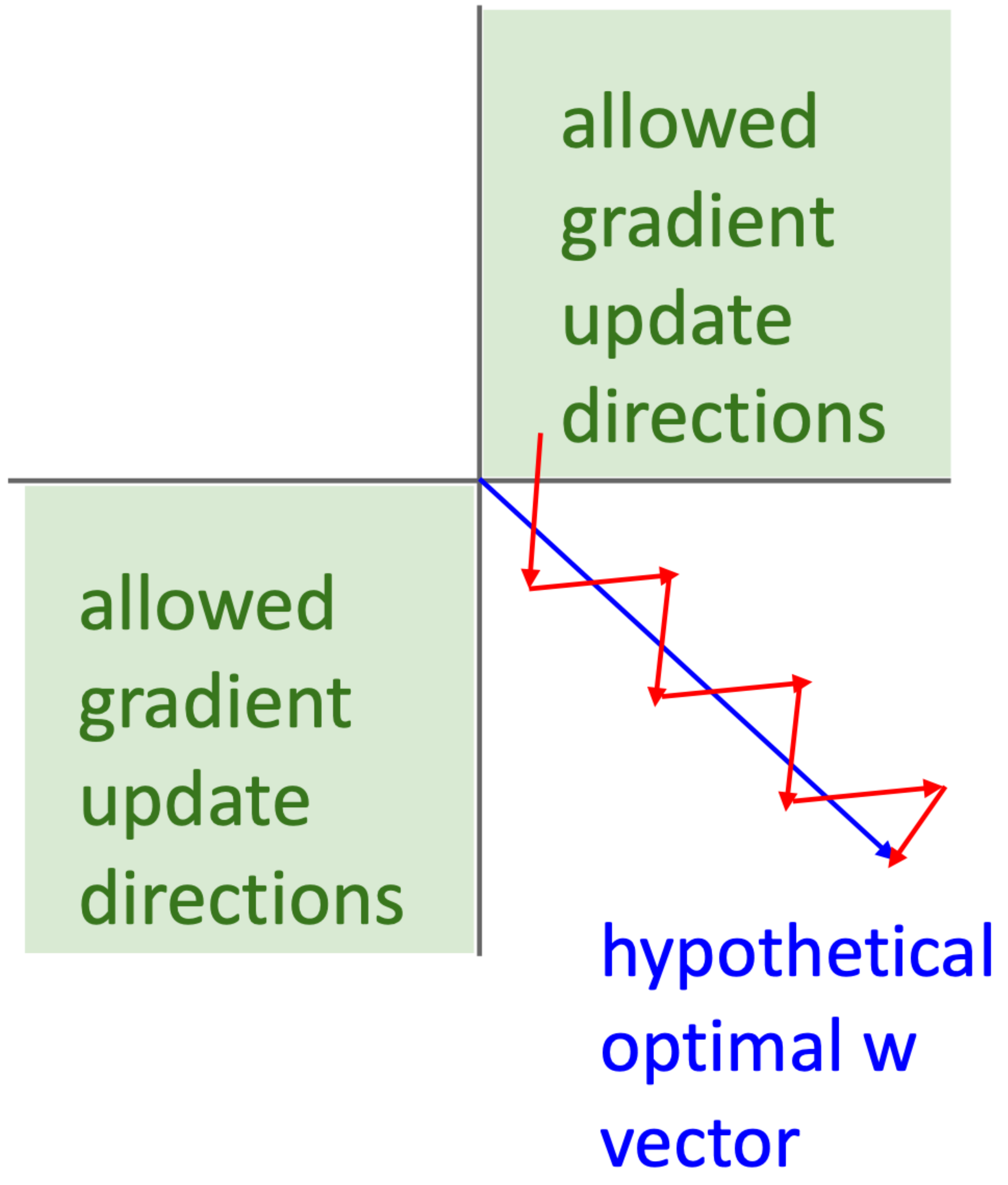
$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

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

$w^{(\ell)}, b^{(\ell)}$ 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

Activation Functions: Sigmoid

Consider what happens when nonlinearity is always positive

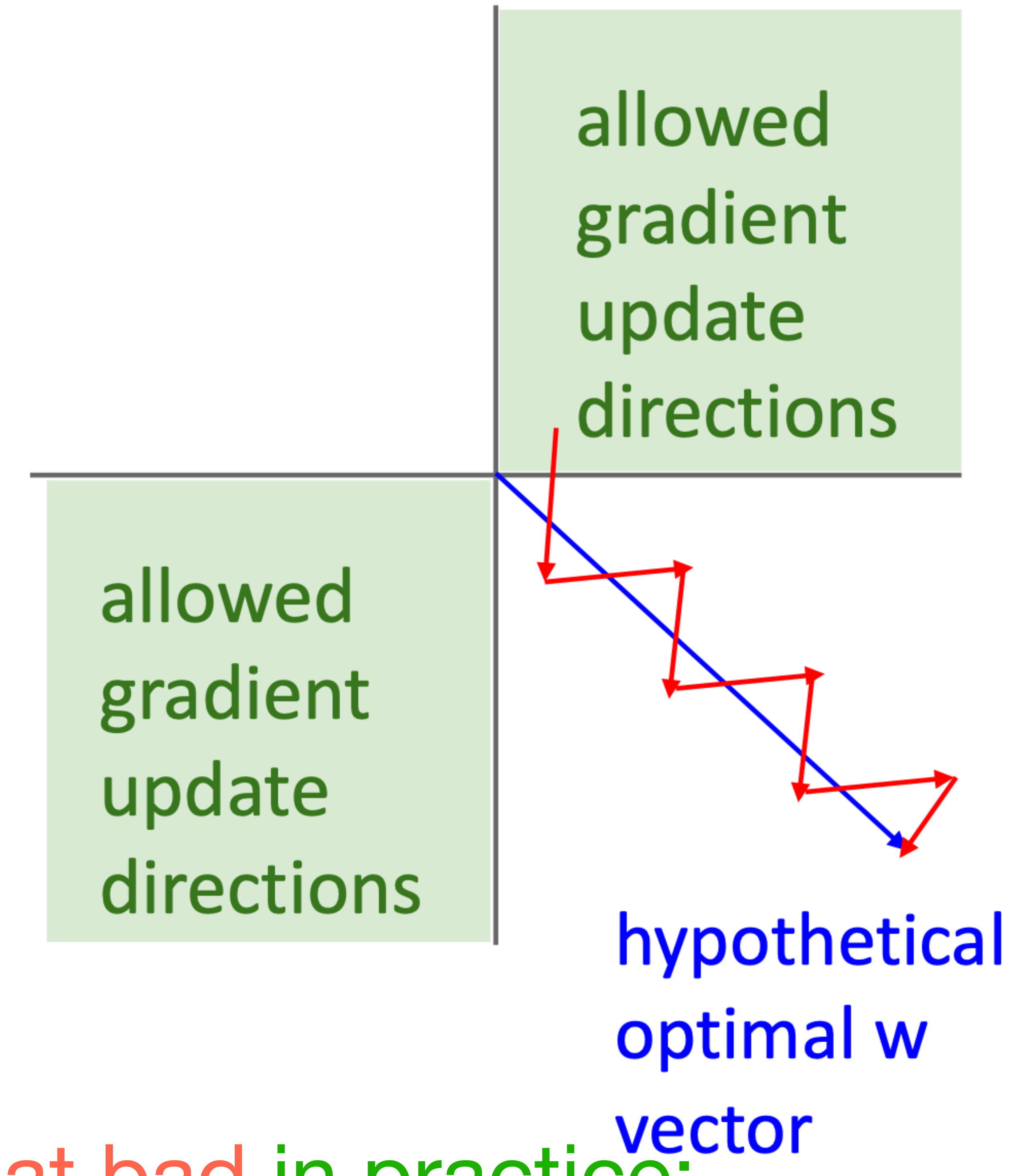
$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{\ell-1}) + b_i^{(\ell)}$$

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

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

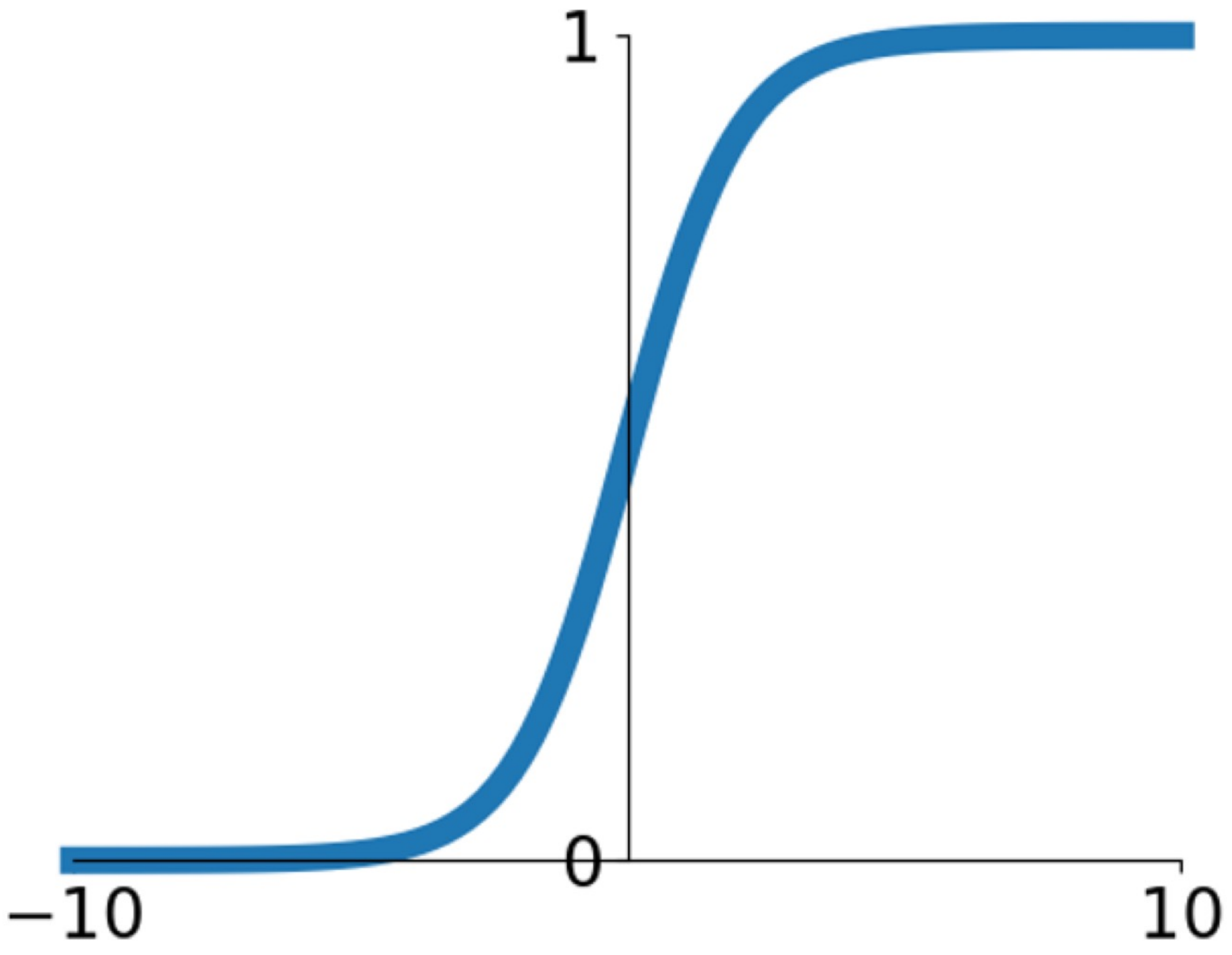


Not that bad in practice:

- Only true for a single example, mini batches help

- BatchNorm can also avoid this

Activation Functions: Sigmoid



Sigmoid

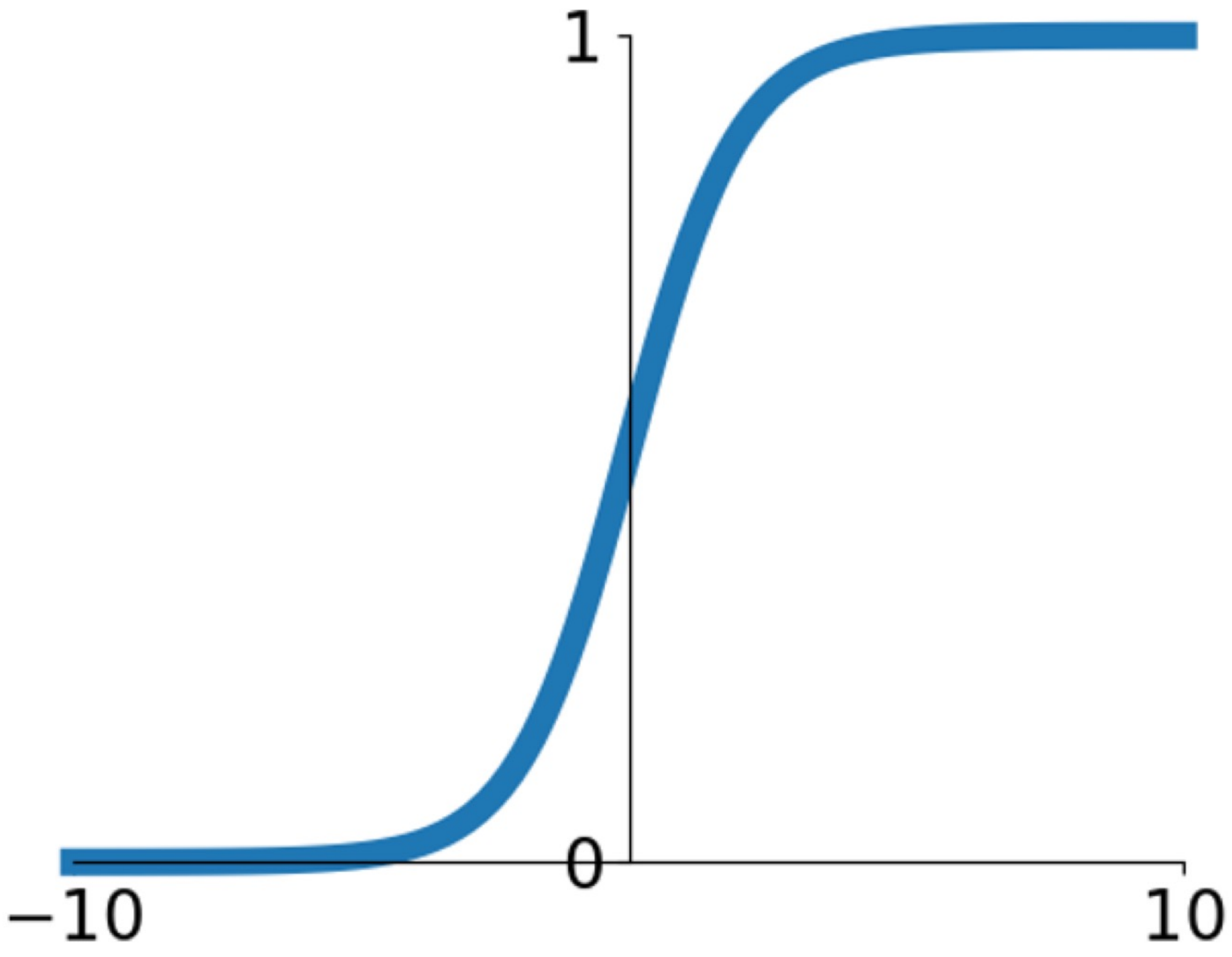
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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

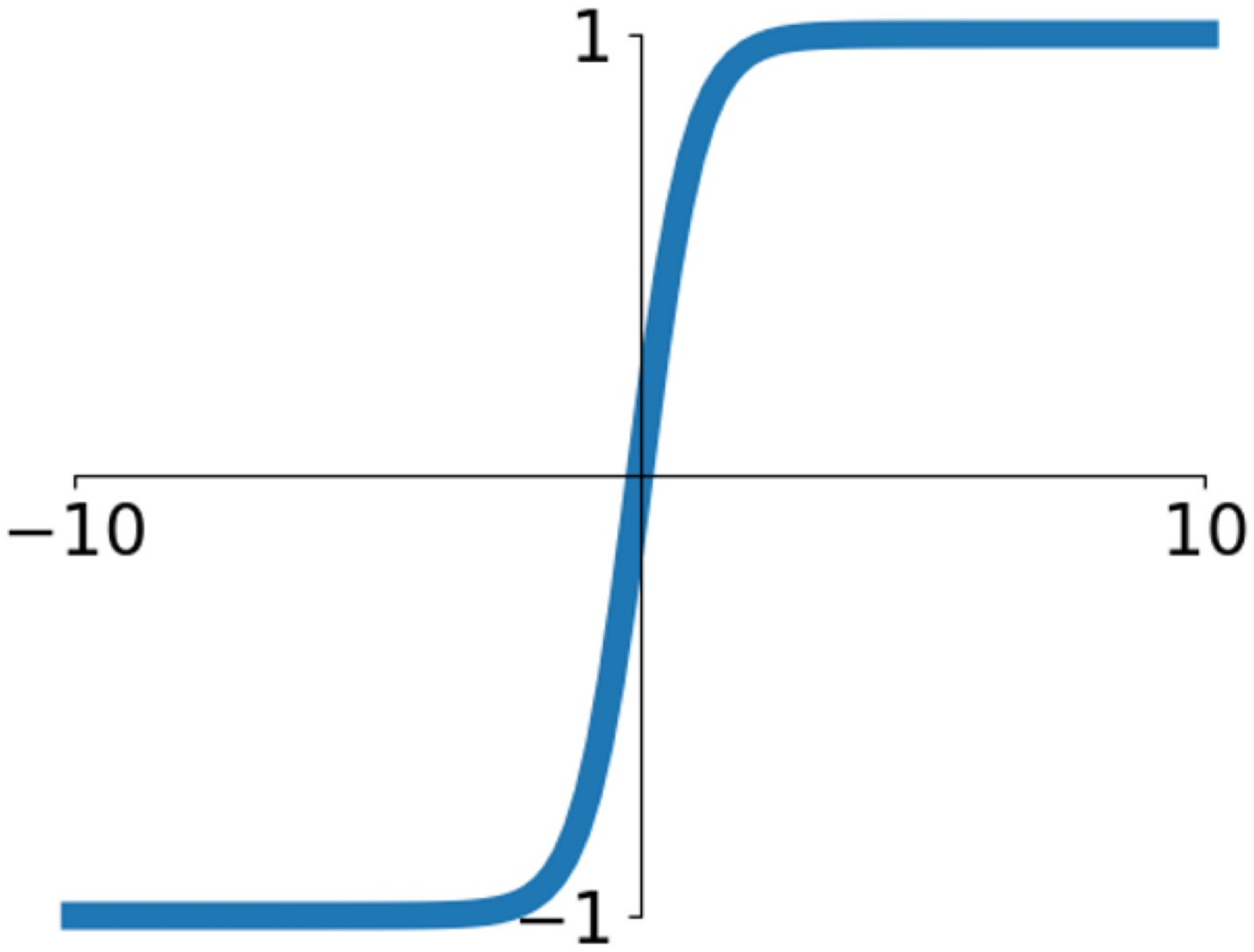
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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

Main issue in practice

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

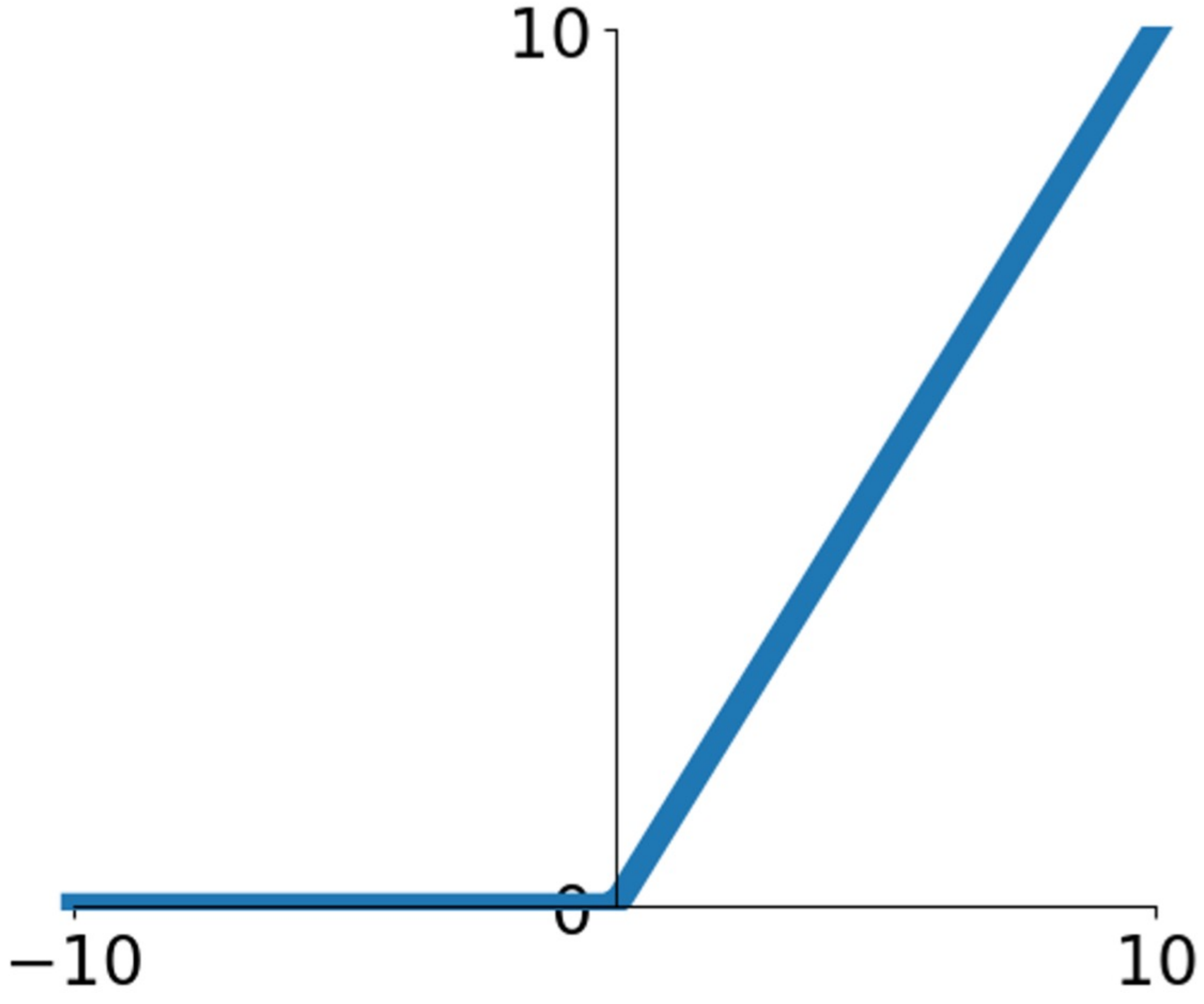
Activation Functions: tanh



tanh(x)

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

Activation Functions: ReLU



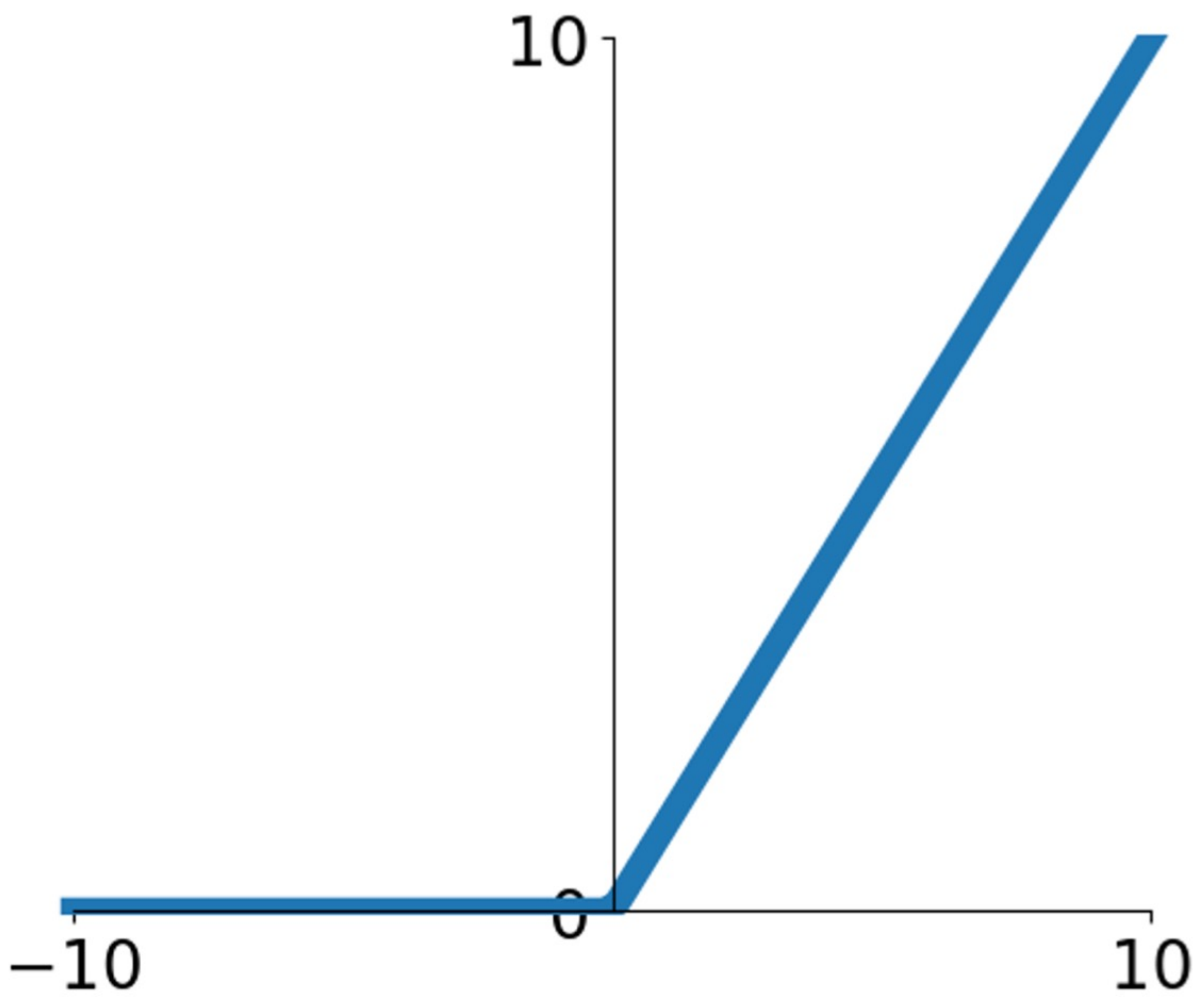
ReLU

(Rectified Linear Unit)

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

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

Activation Functions: ReLU



ReLU

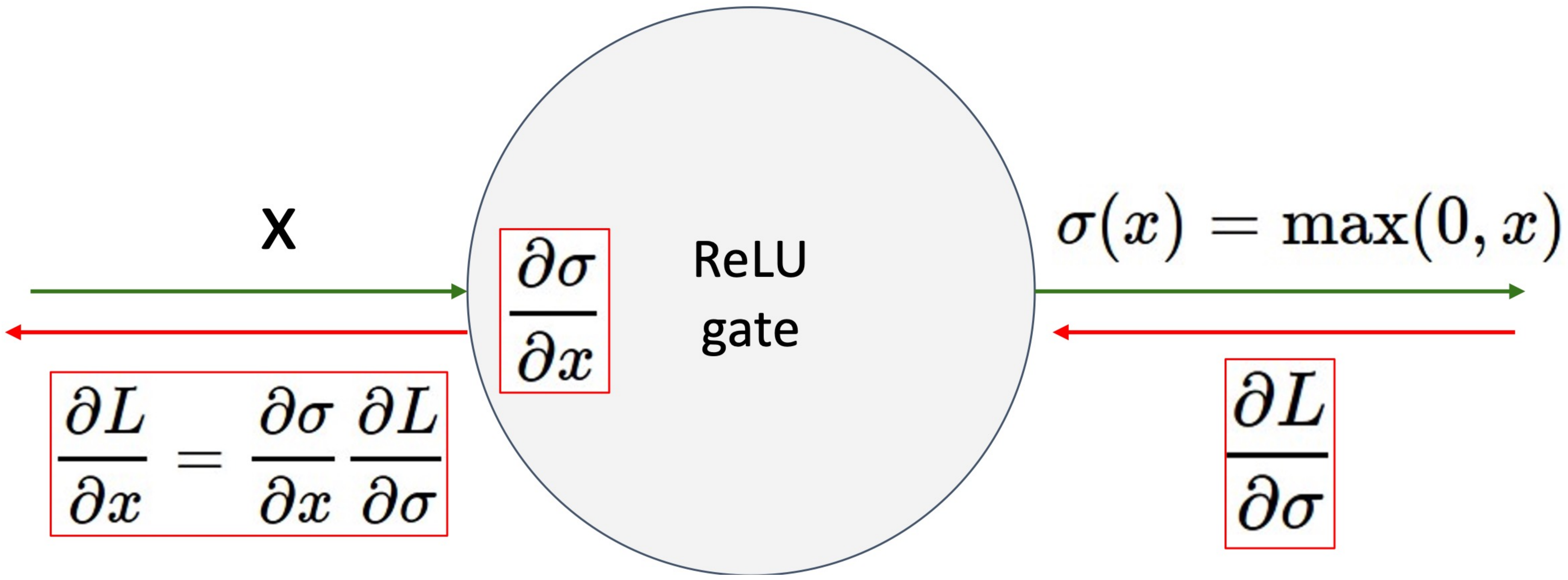
(Rectified Linear Unit)

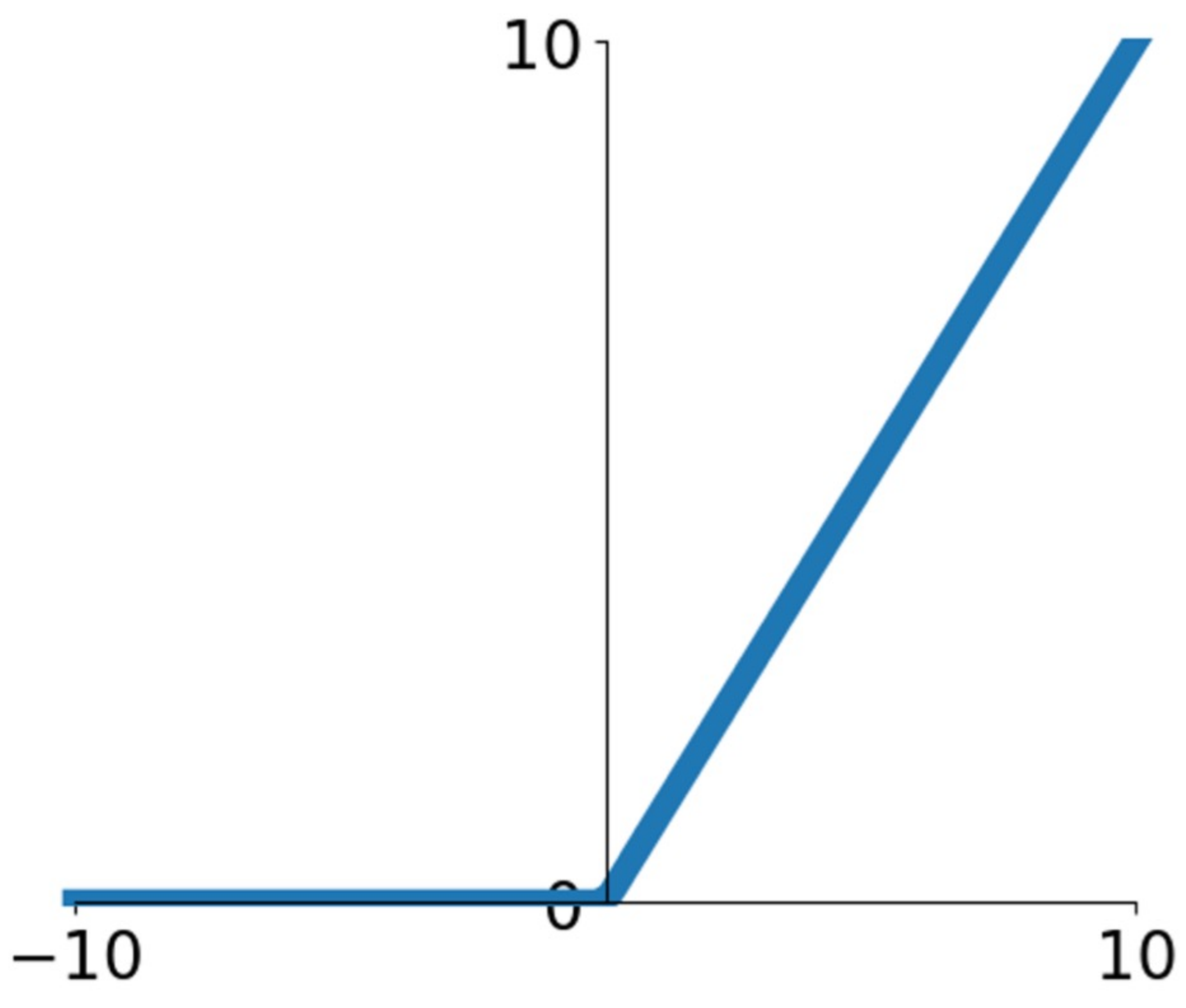
$$f(x) = max(0, x)$$

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid and tanh in practice (e.g. 6x)
- Not zero-centered output
- An annoyance:

what is the gradient when x<0?

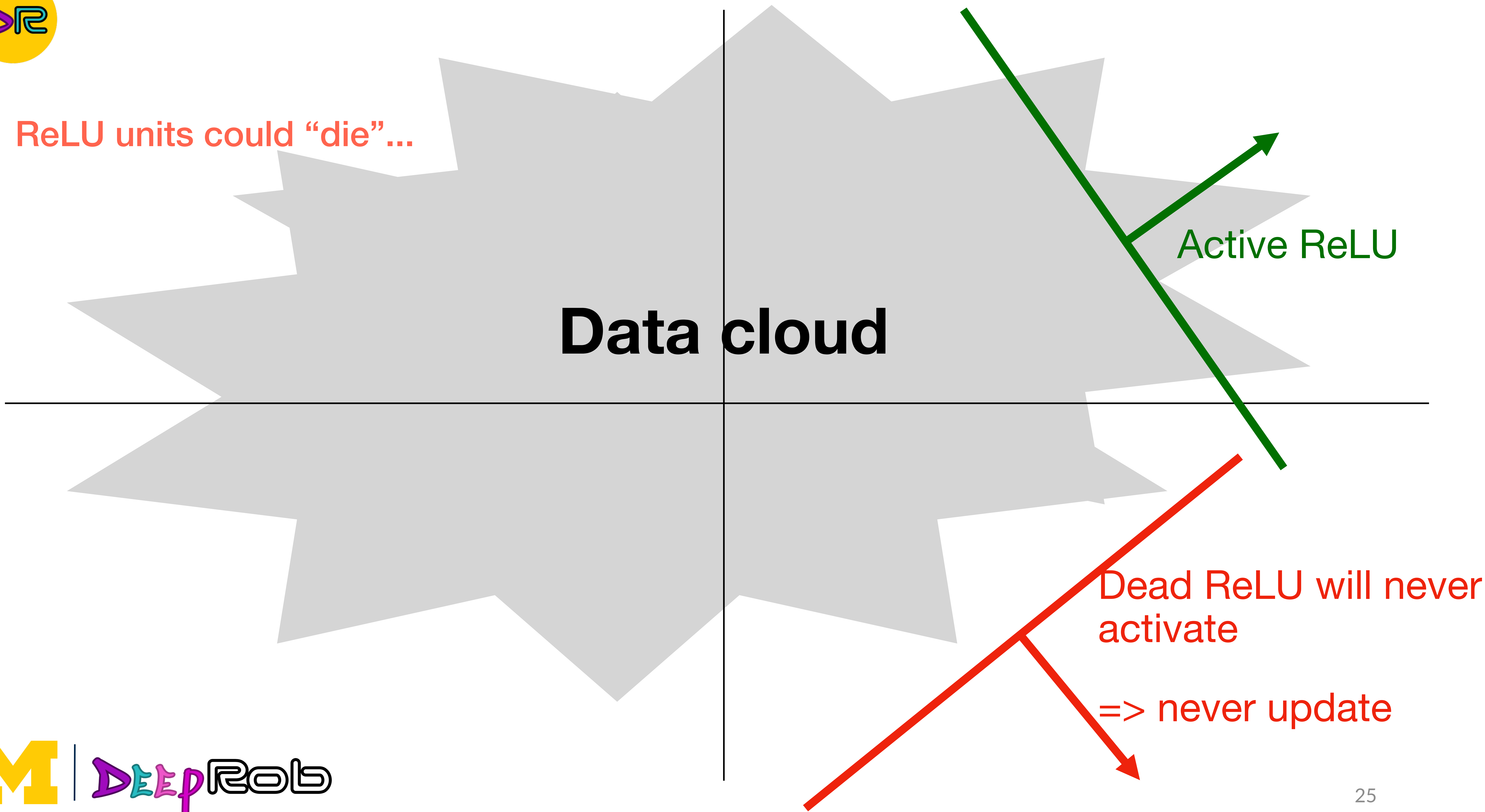
Activation Functions: ReLU

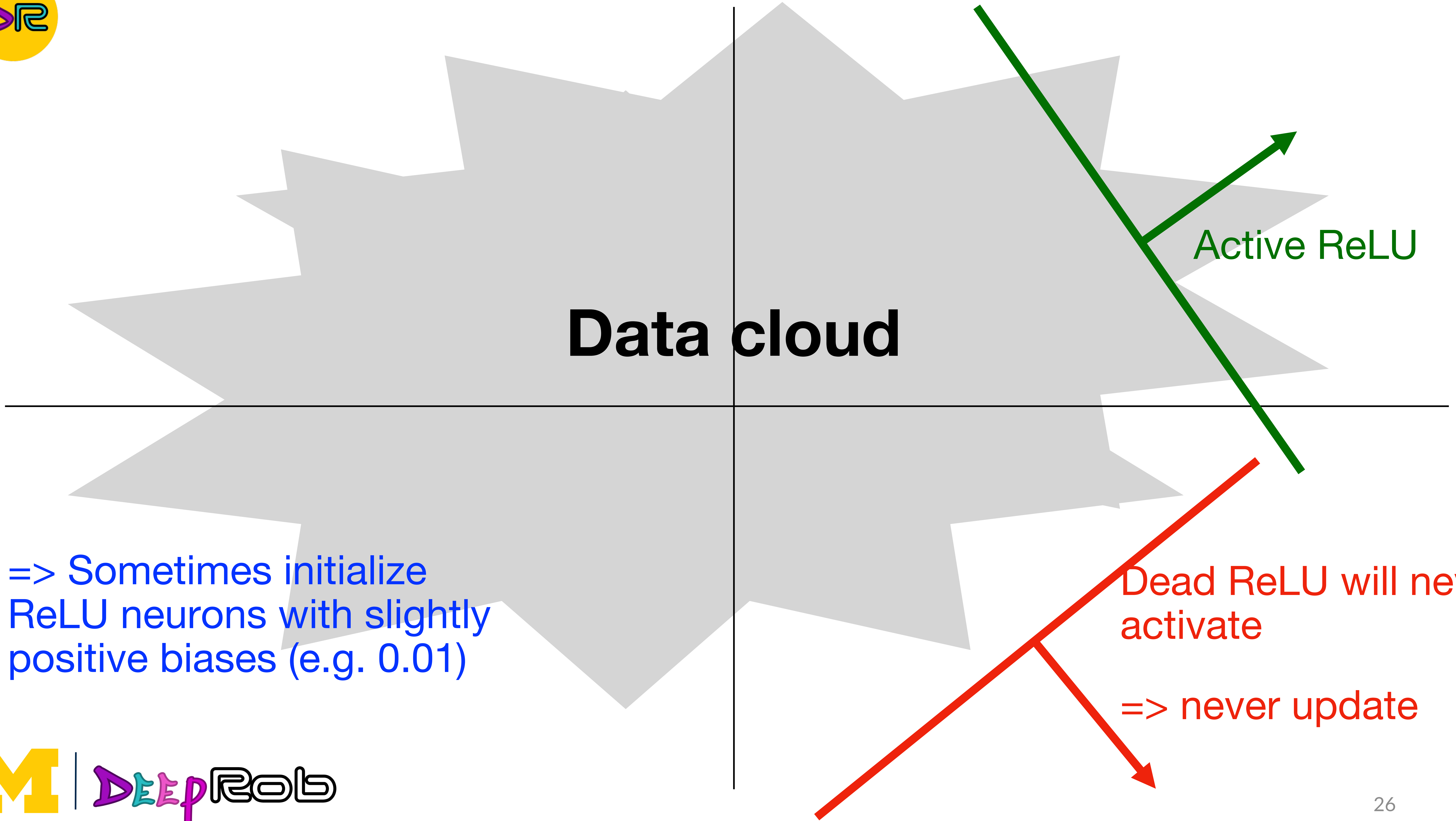




- What happens when x = -10?
- What happens when x = 10?

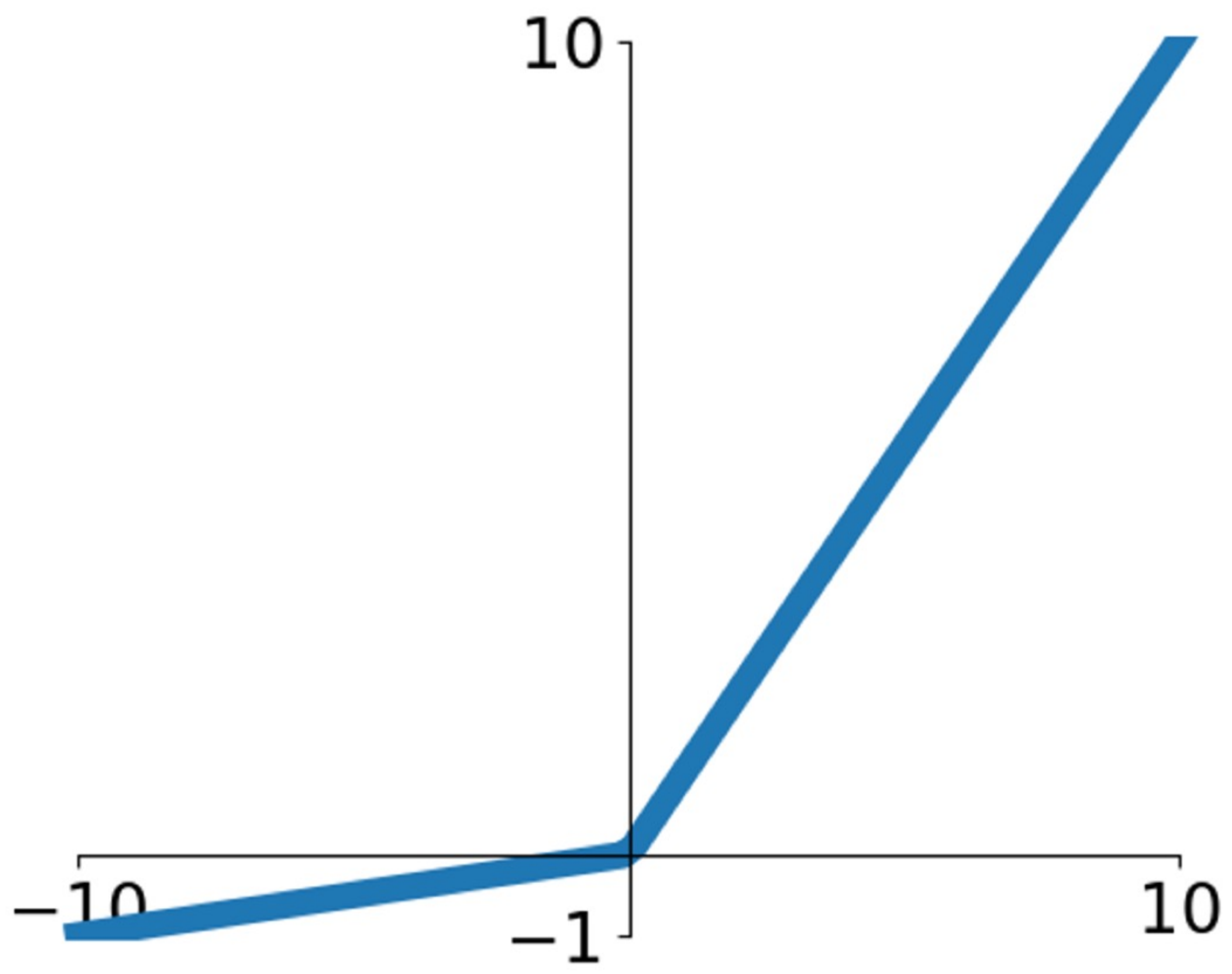
ReLU units could "die"...





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

Activation Functions: Leaky ReLU



Leaky ReLU

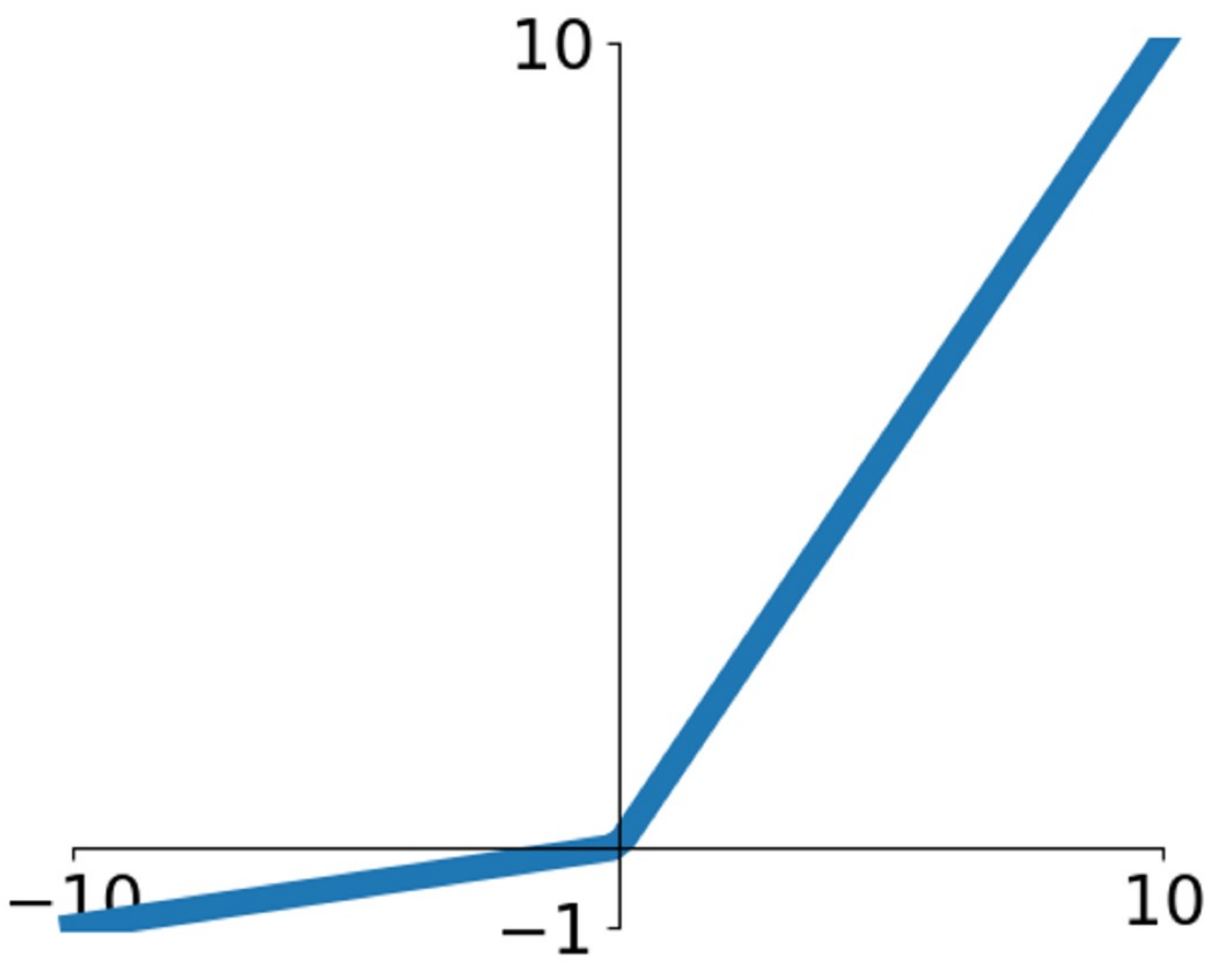
$f(x) = max(\alpha x, x)$

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

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid and 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



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid and 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

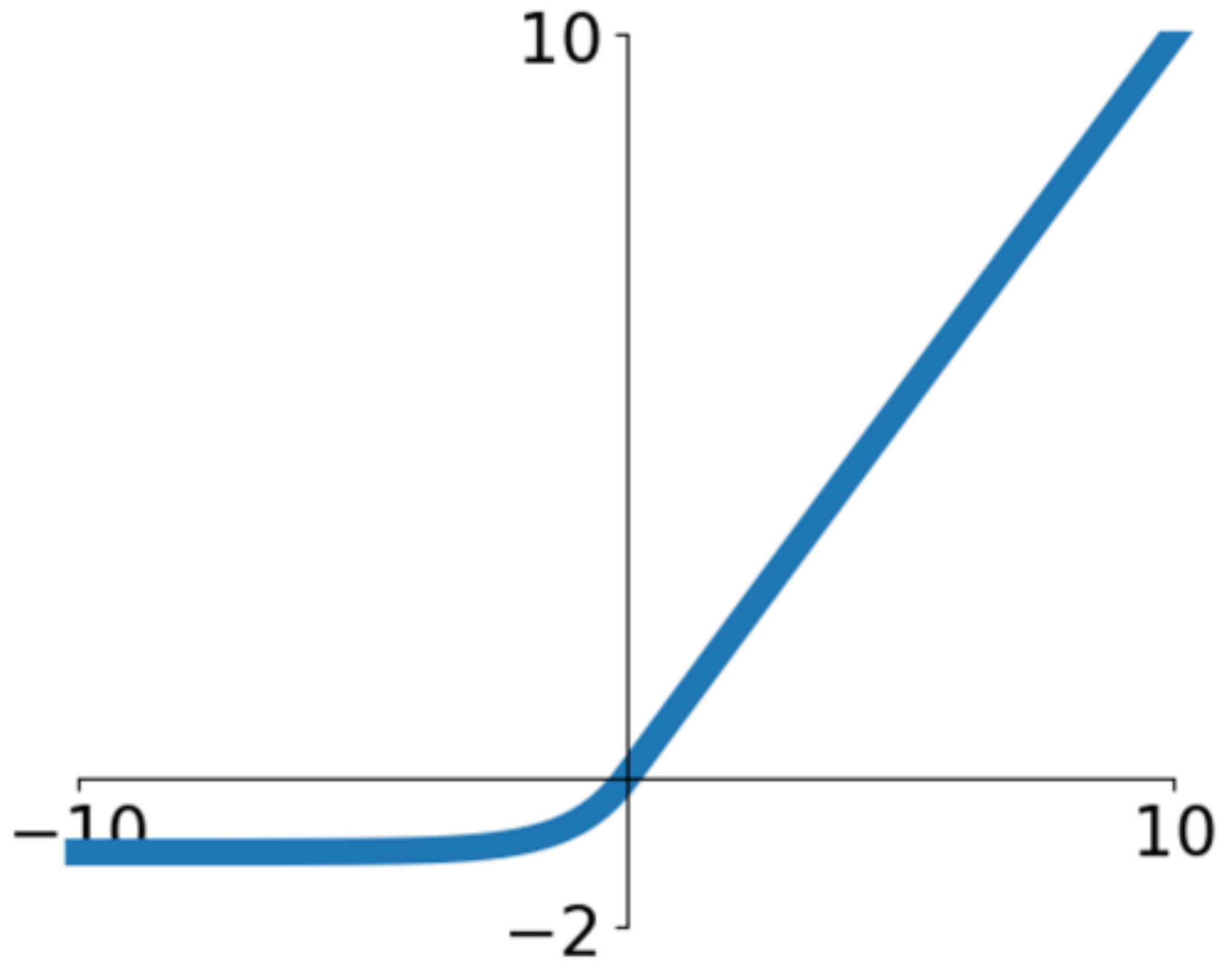
Parametric ReLU (PReLU)

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

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)



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

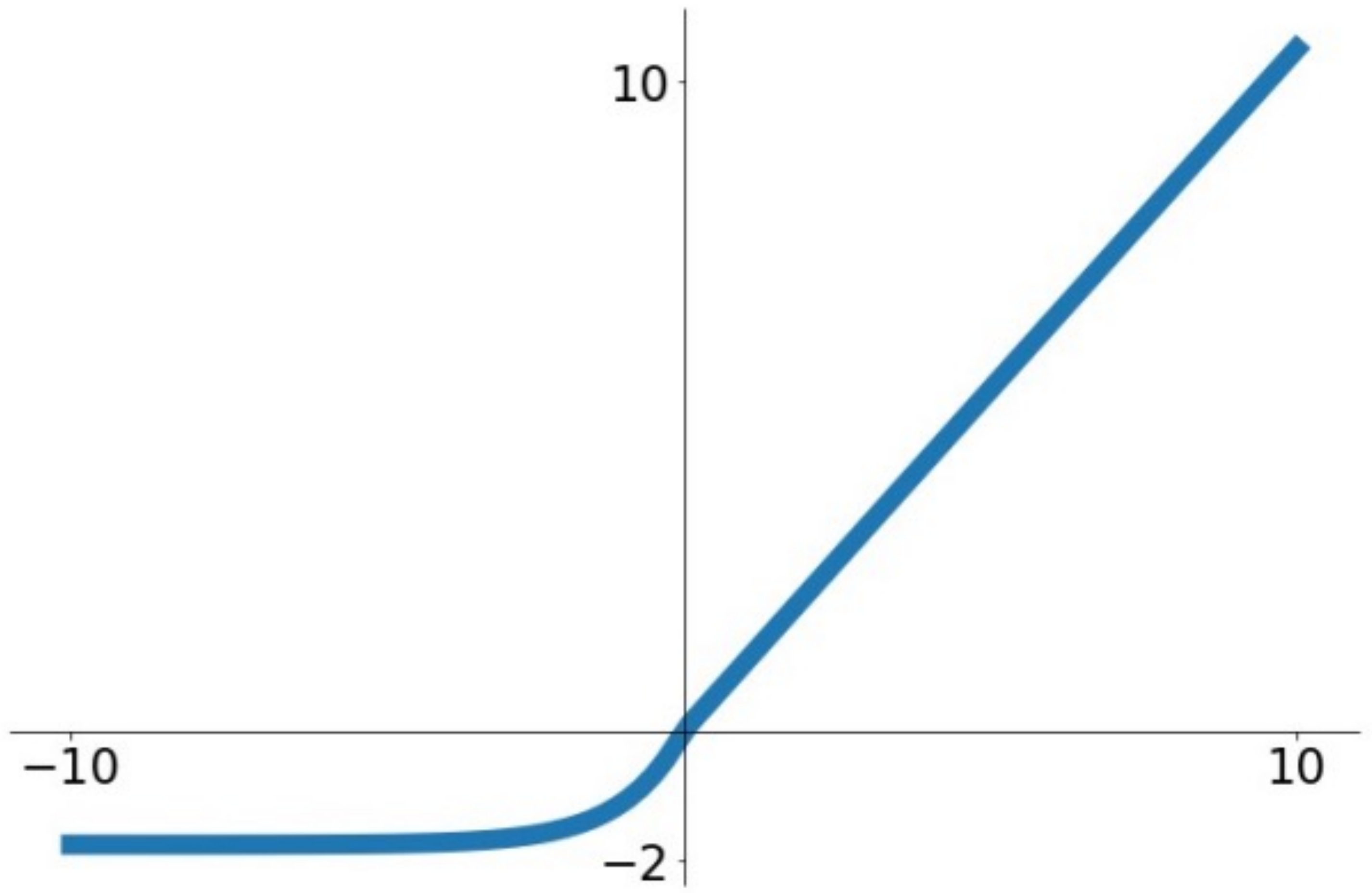
(Default $\alpha = 1$)

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

- Computation requires exp()

Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), ICLR 2016

Activation Functions: Scale Exponential Linear Unit (SELU)



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

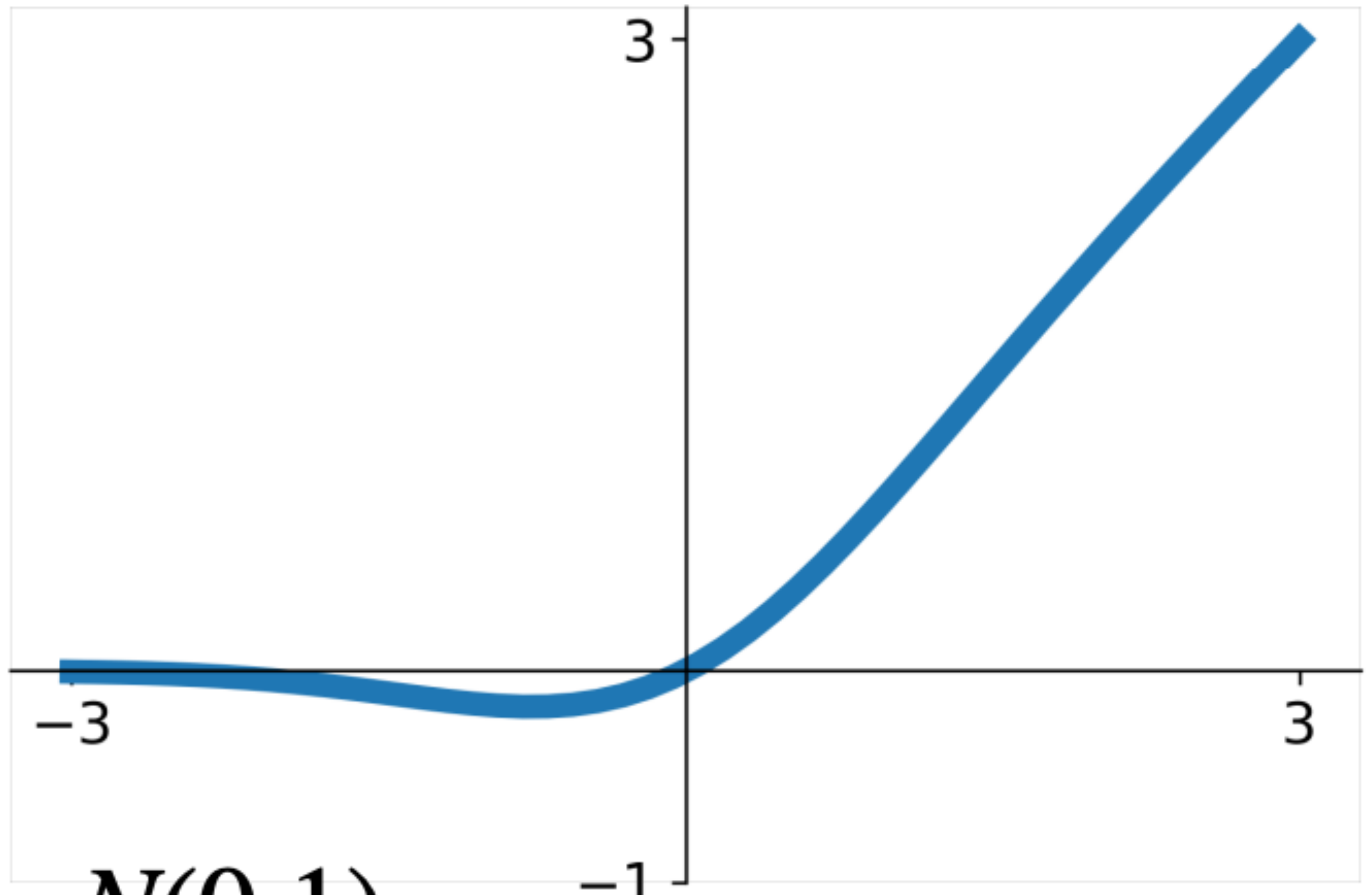
$$selu(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

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

derivation see original paper (91 pages…)

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

Activation Functions: Gaussian Error Linear Unit (GELU)



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

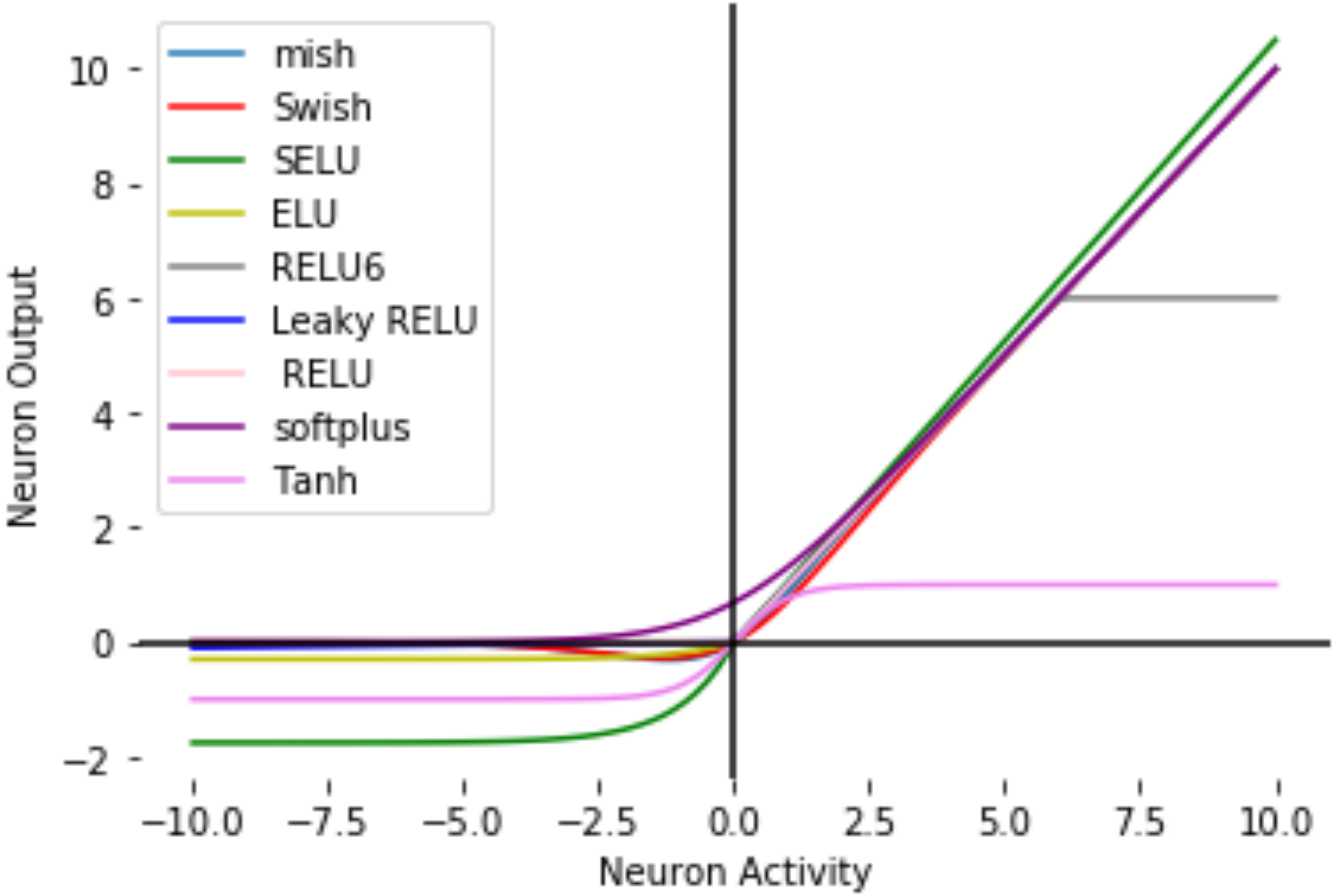
$$\text{gelu}(x) = xP(X \le x) = \frac{x}{2}(1 + \text{erf}(x/\sqrt{2}))$$

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

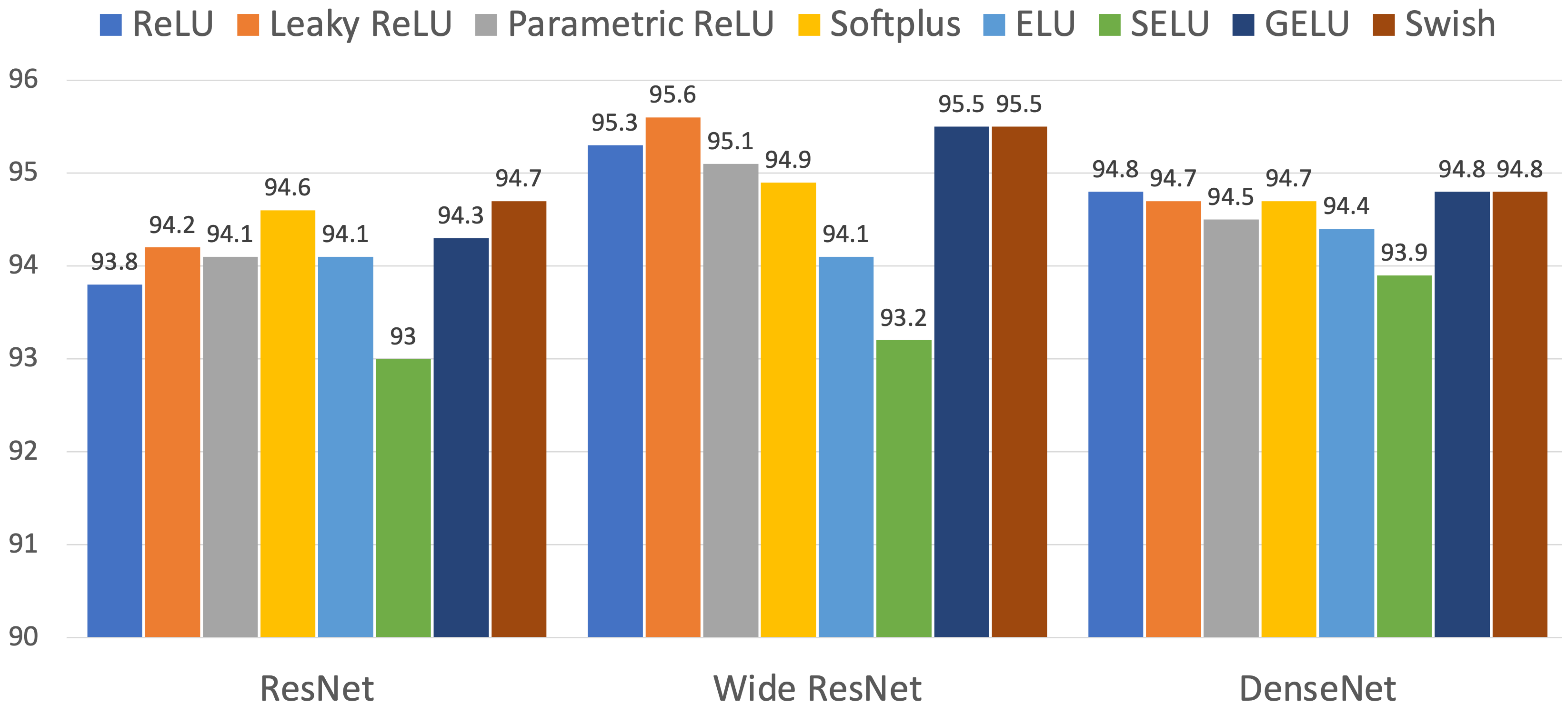
- **Idea**: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, ViT)

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

Activation Functions



Accuracy on CIFAR10



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

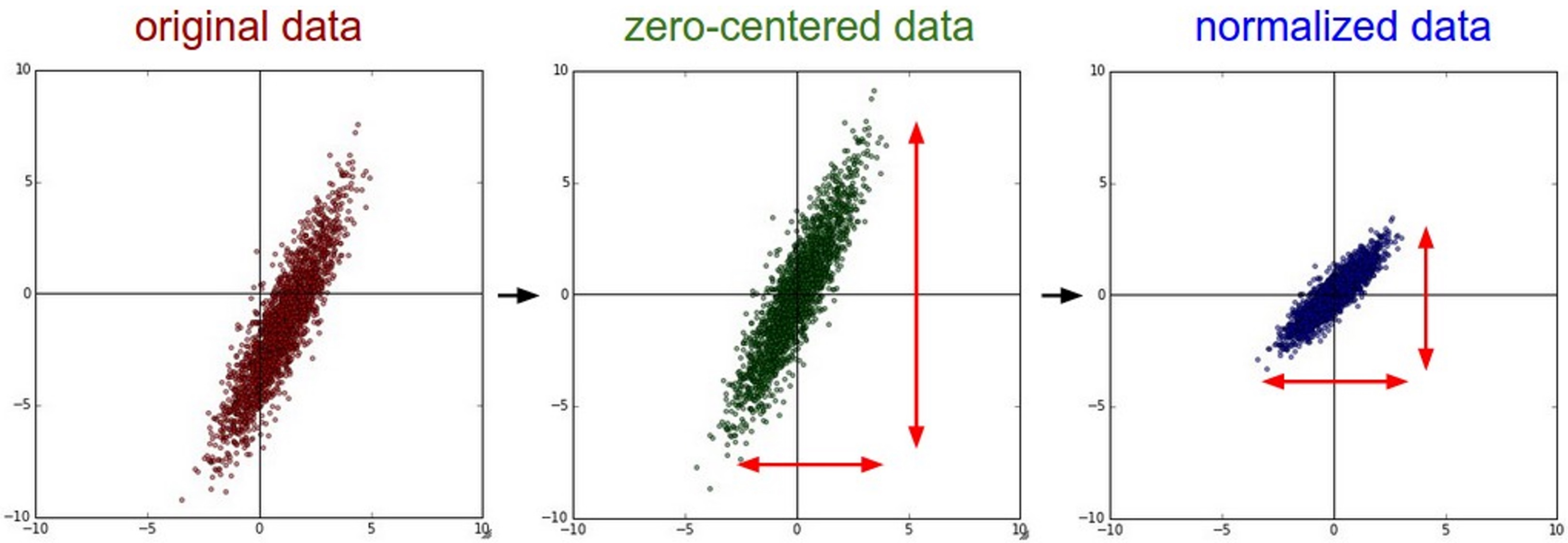
Activation Functions: Summary

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

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

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

Data preprocessing



See batchnorm

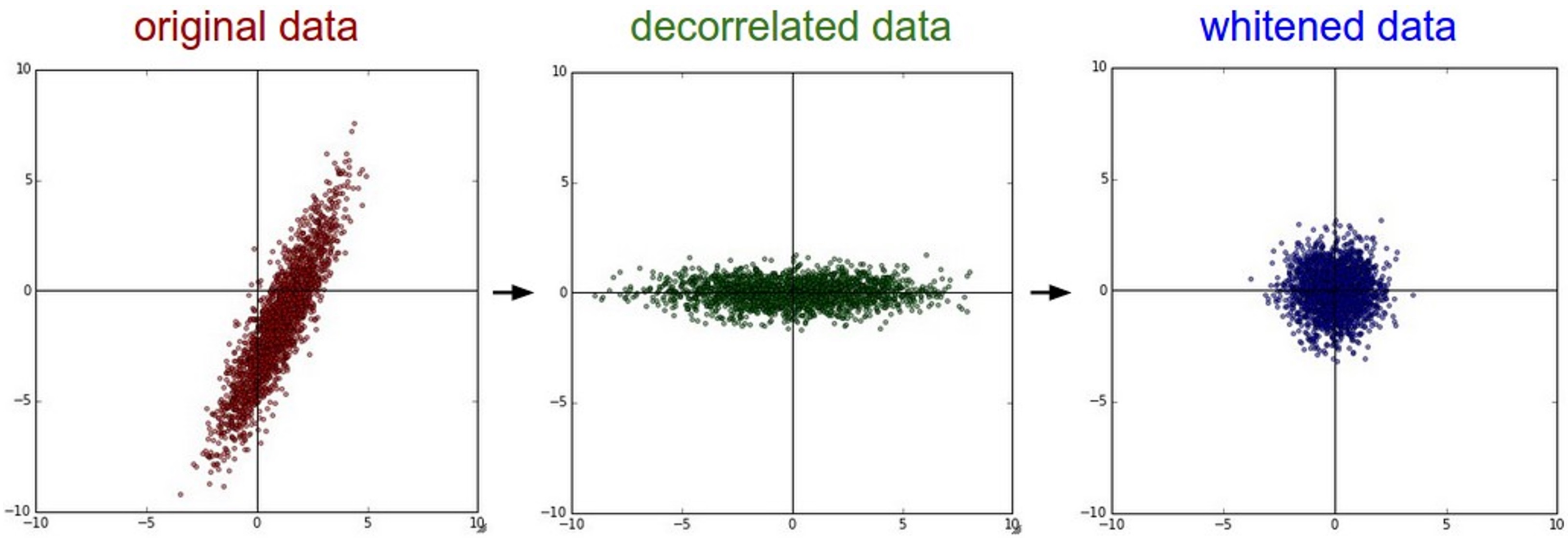
```
X -= np.mean(X, axis = 0)
```

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

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

Data preprocessing

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

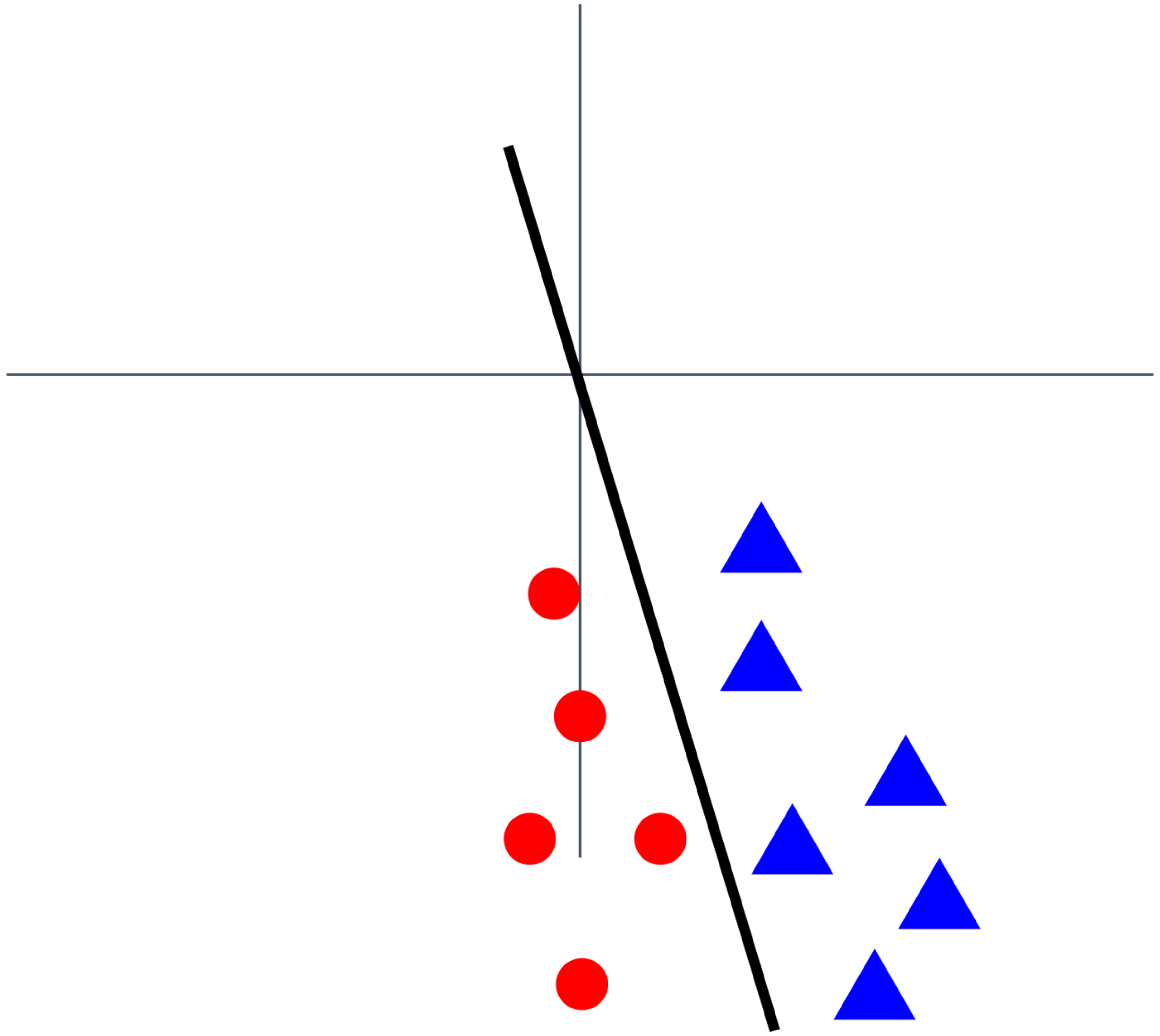


(Data has diagonal covariance matrix)

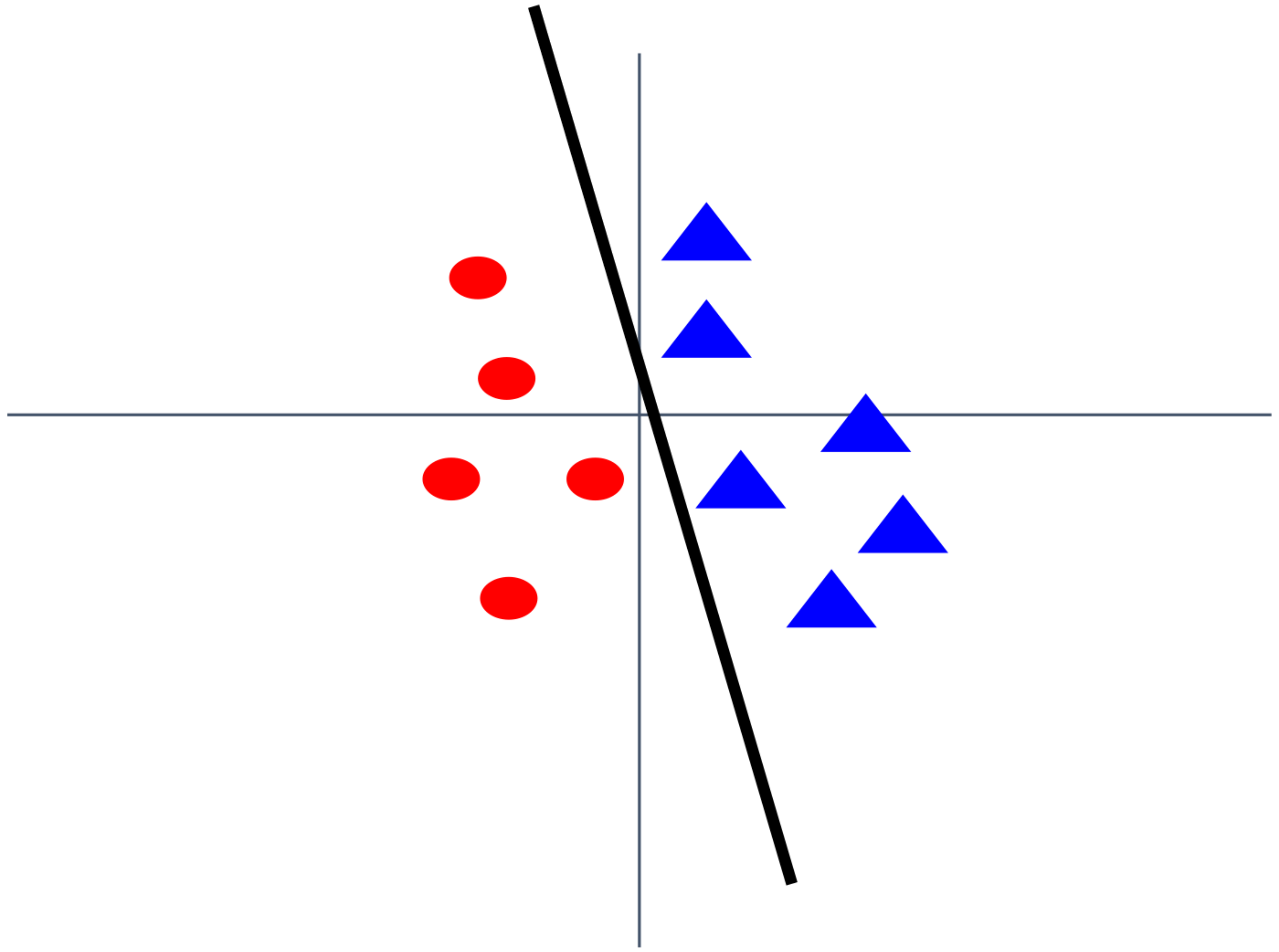
(Covariance matrix is the identity matrix)

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

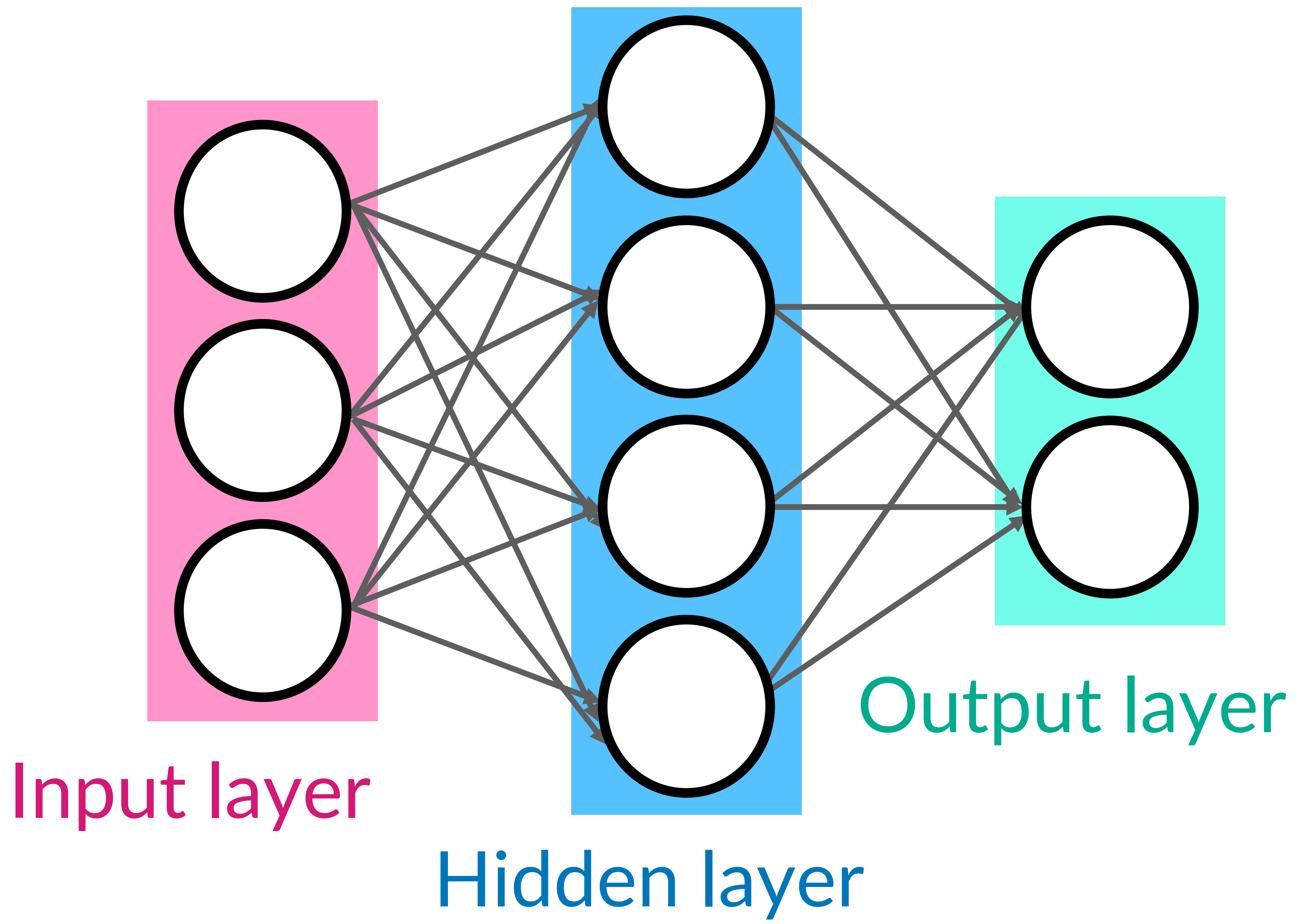
- Subtract the mean image (e.g. AlexNet)
  (mean image = [32, 32, 3] array)

- 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



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

A: All outputs are 0, all gradients are the same!

"symmetry breaking" problem

https://www.pinecone.io/learn/weight-initialization/

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.

“vanishing gradient” problem

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

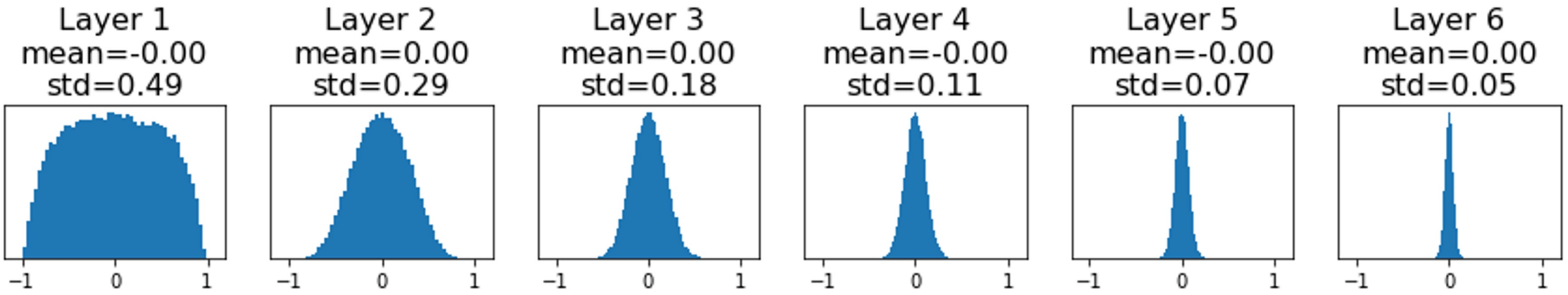
Weight initialization: Activation statistics

```
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

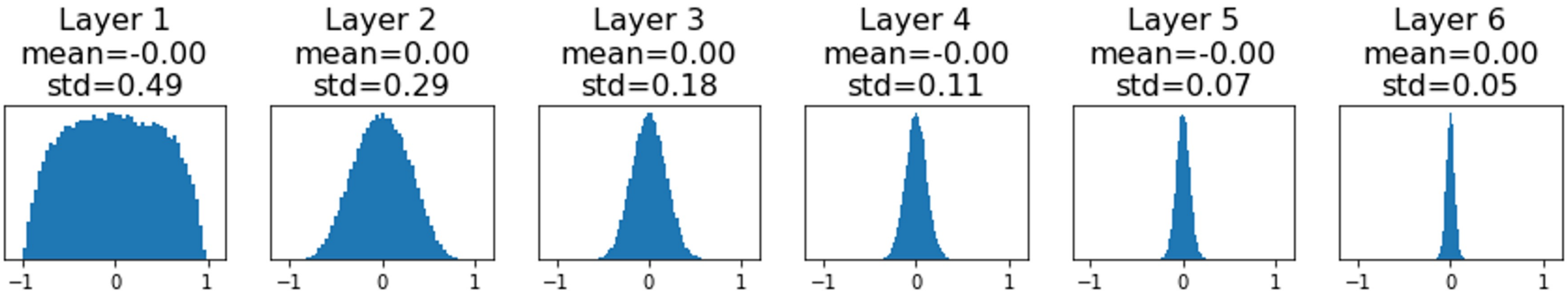
```python
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

All activations tend to be zero for deeper network layers

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

A: All zero, no learning :(



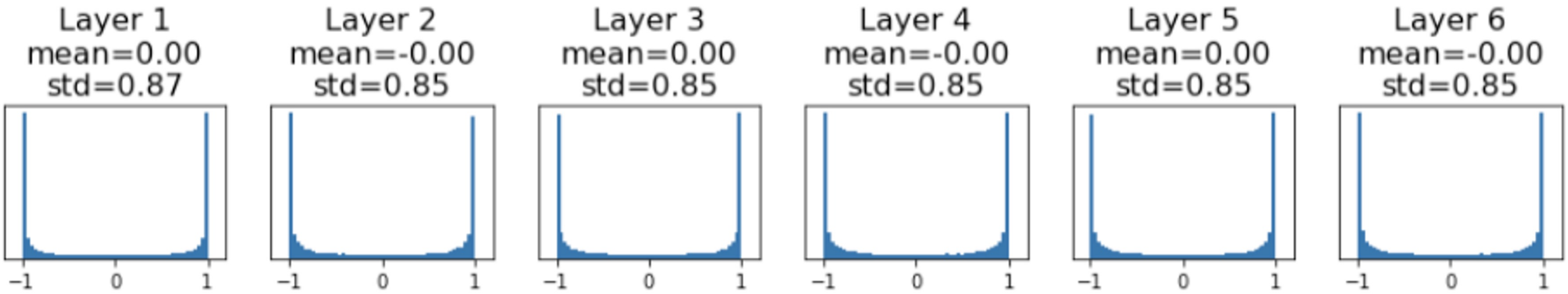
Weight initialization: Activation statistics

```
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?



Weight initialization: Activation statistics

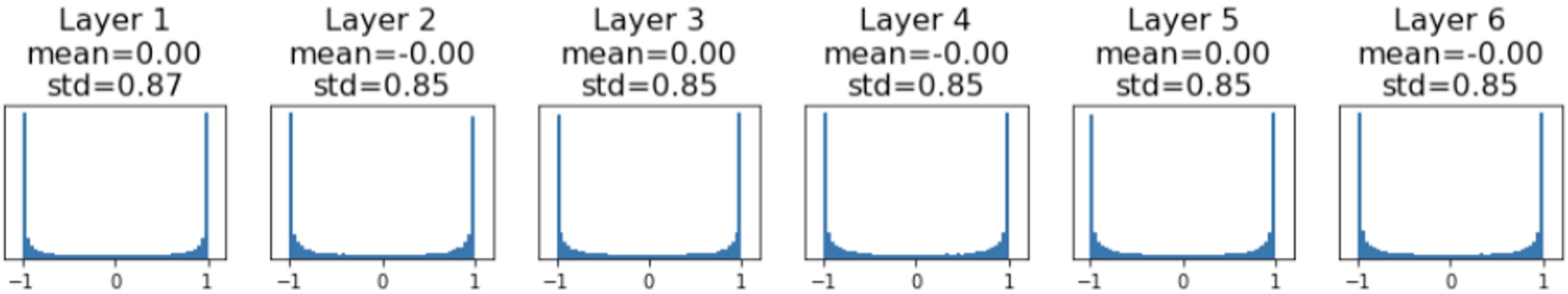
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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!

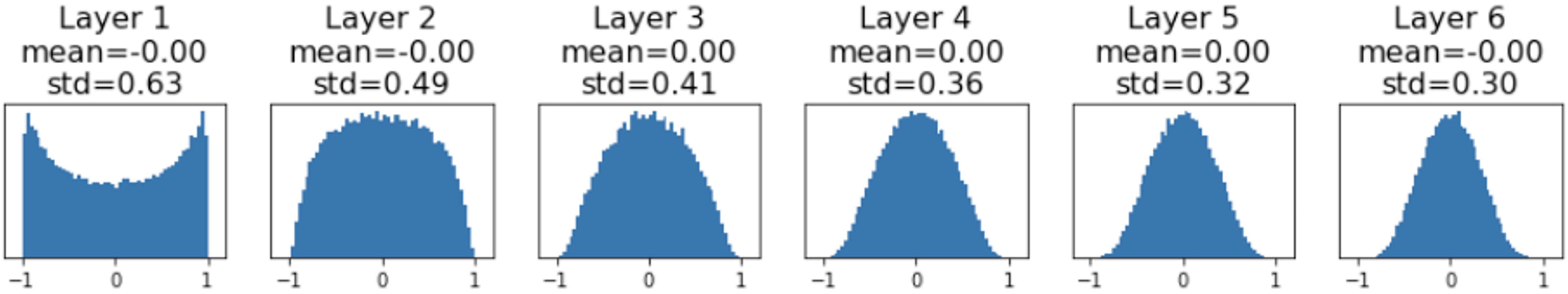
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

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!



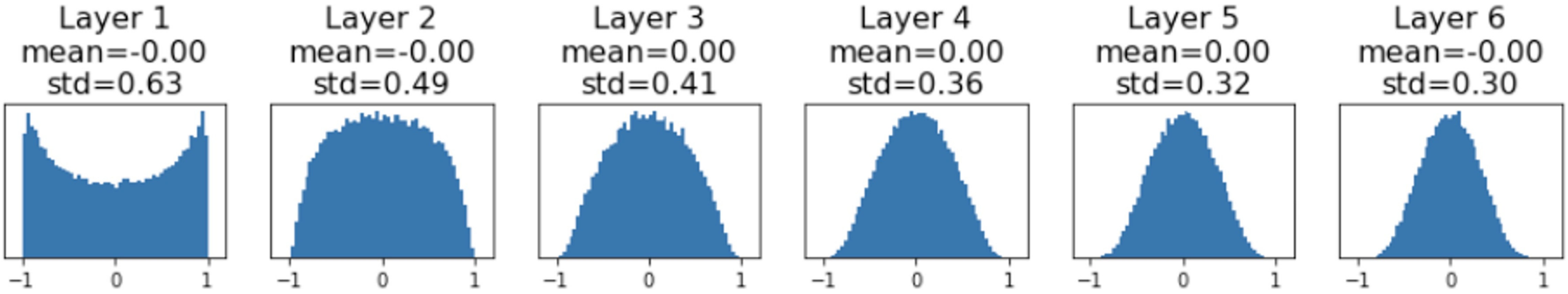
Weight initialization: Xavier Initialization

```
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 x input_channels



Weight initialization: Xavier Initialization

"Xavier" initialization:
$std = 1/\sqrt{Din}$

Derivation: Variance of output = Variance of input

$$y = Wx$$

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

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

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

$= Din \times Var(x_i) \times Var(w_i)$ [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

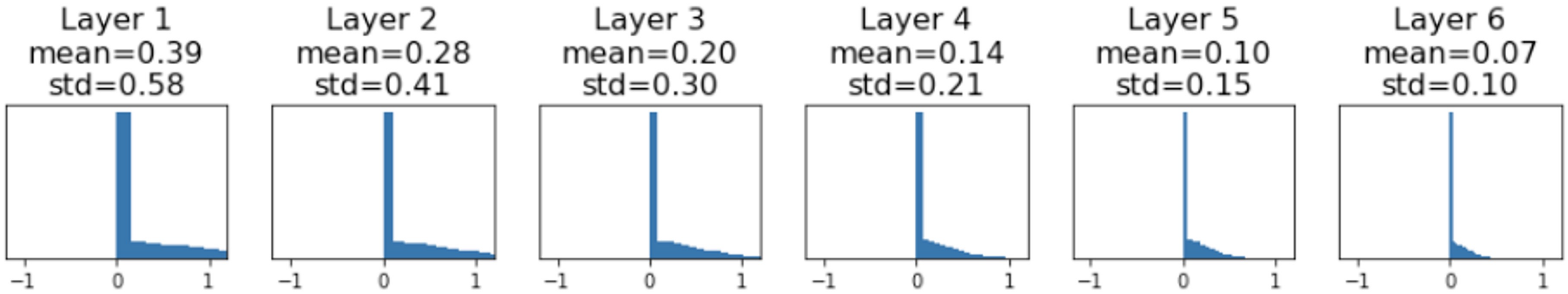
Weight initialization: What about ReLU?

```
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 :(

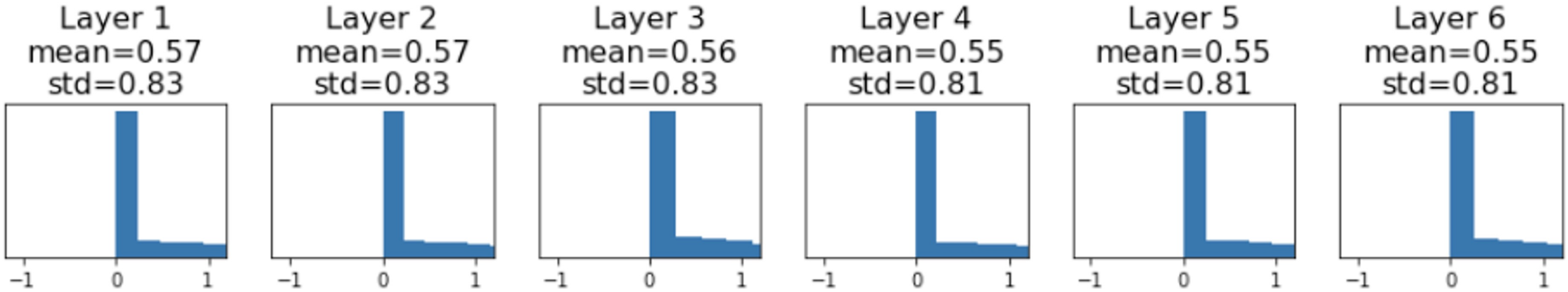


Weight initialization: Kaiming / MSRA initialization

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

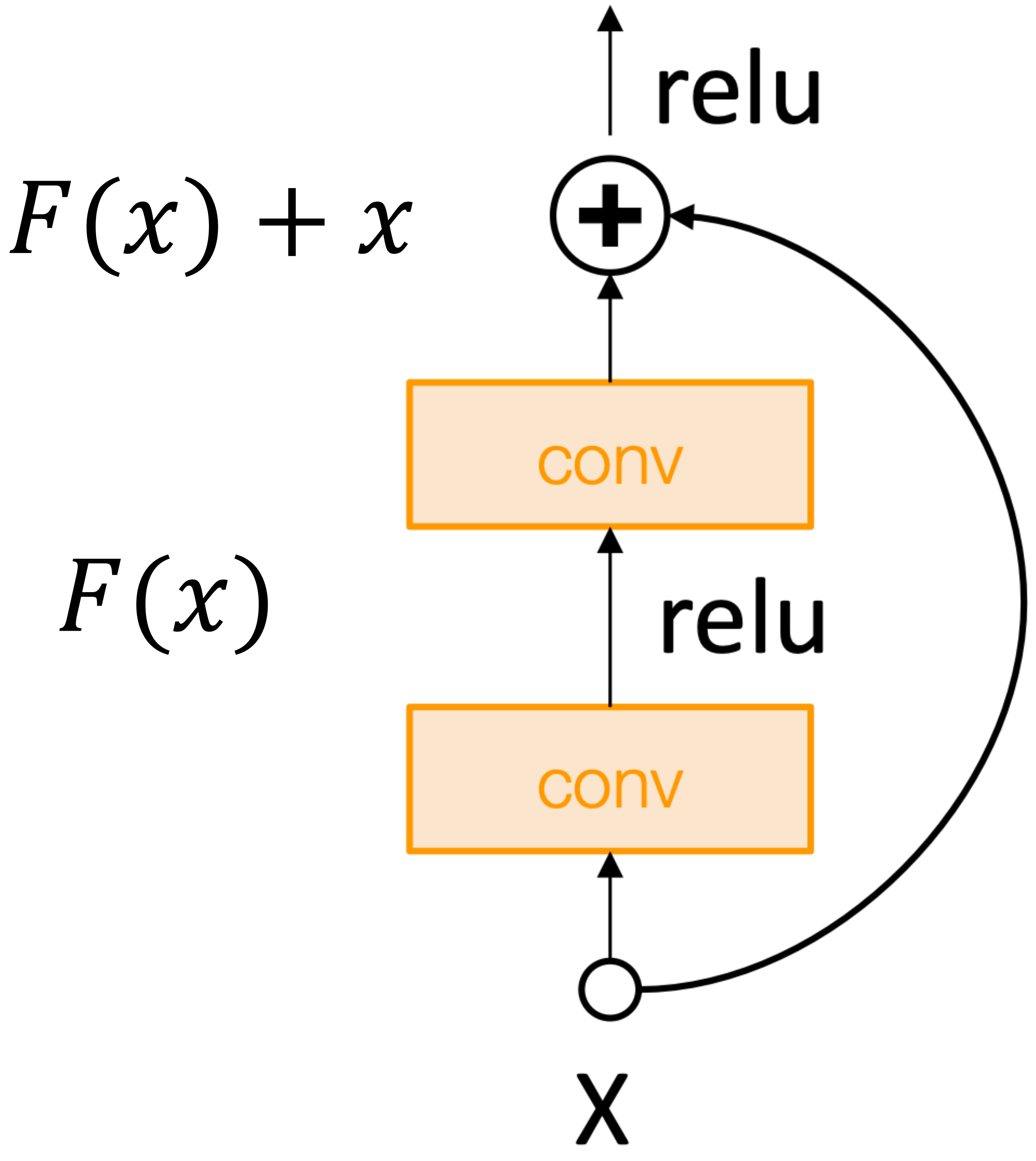
ReLU correction: std = sqrt(2 / Din)

"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

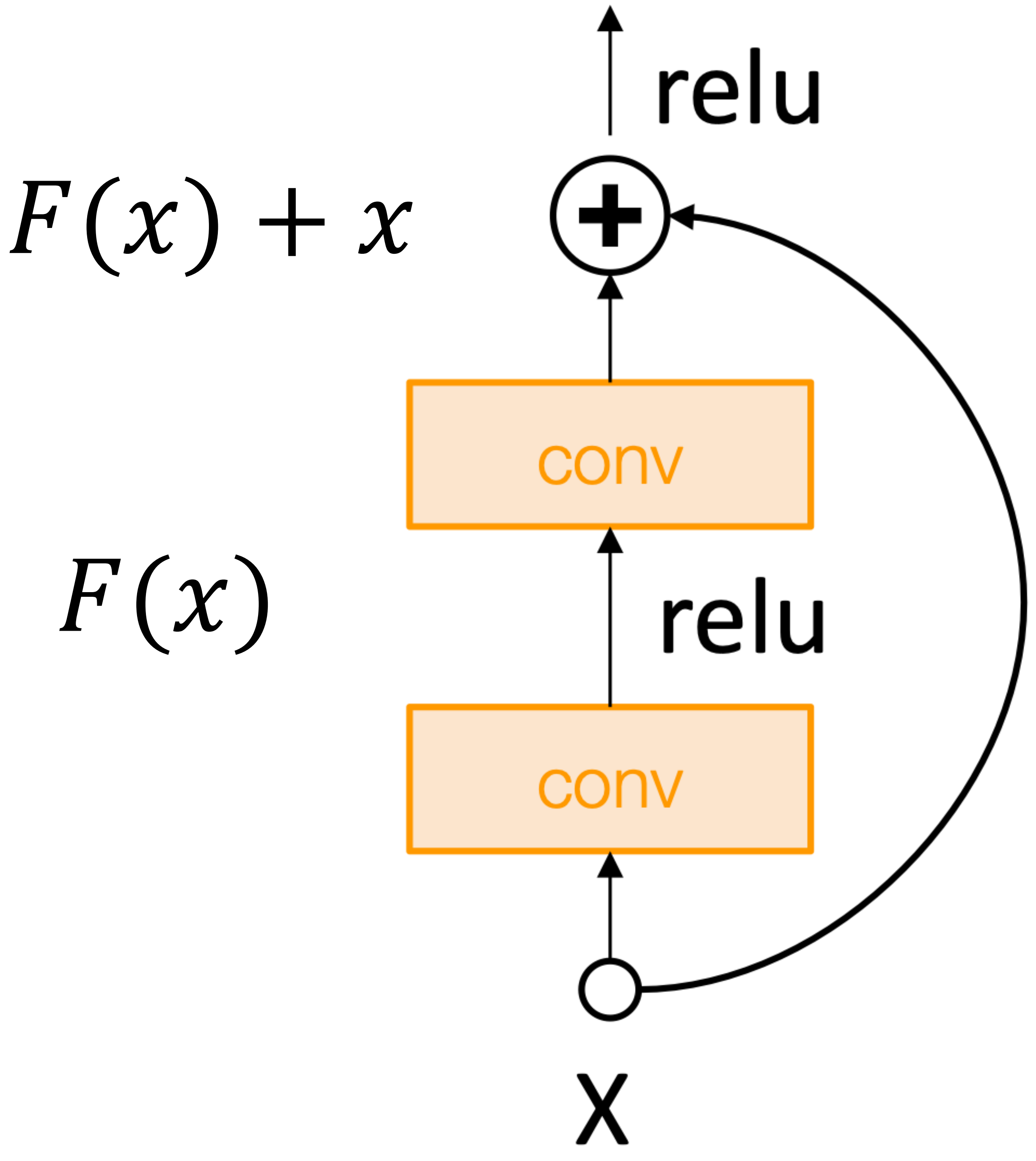


Residual Block

If we initialize with MSRA:
then $Var(F(x)) = Var(x)$

But then $Var(F(x) + x) > Var(x)$ variance grows with each block!

Weight initialization: Residual Networks



Residual Block

If we initialize with MSRA: then $Var(F(x)) = Var(x)$

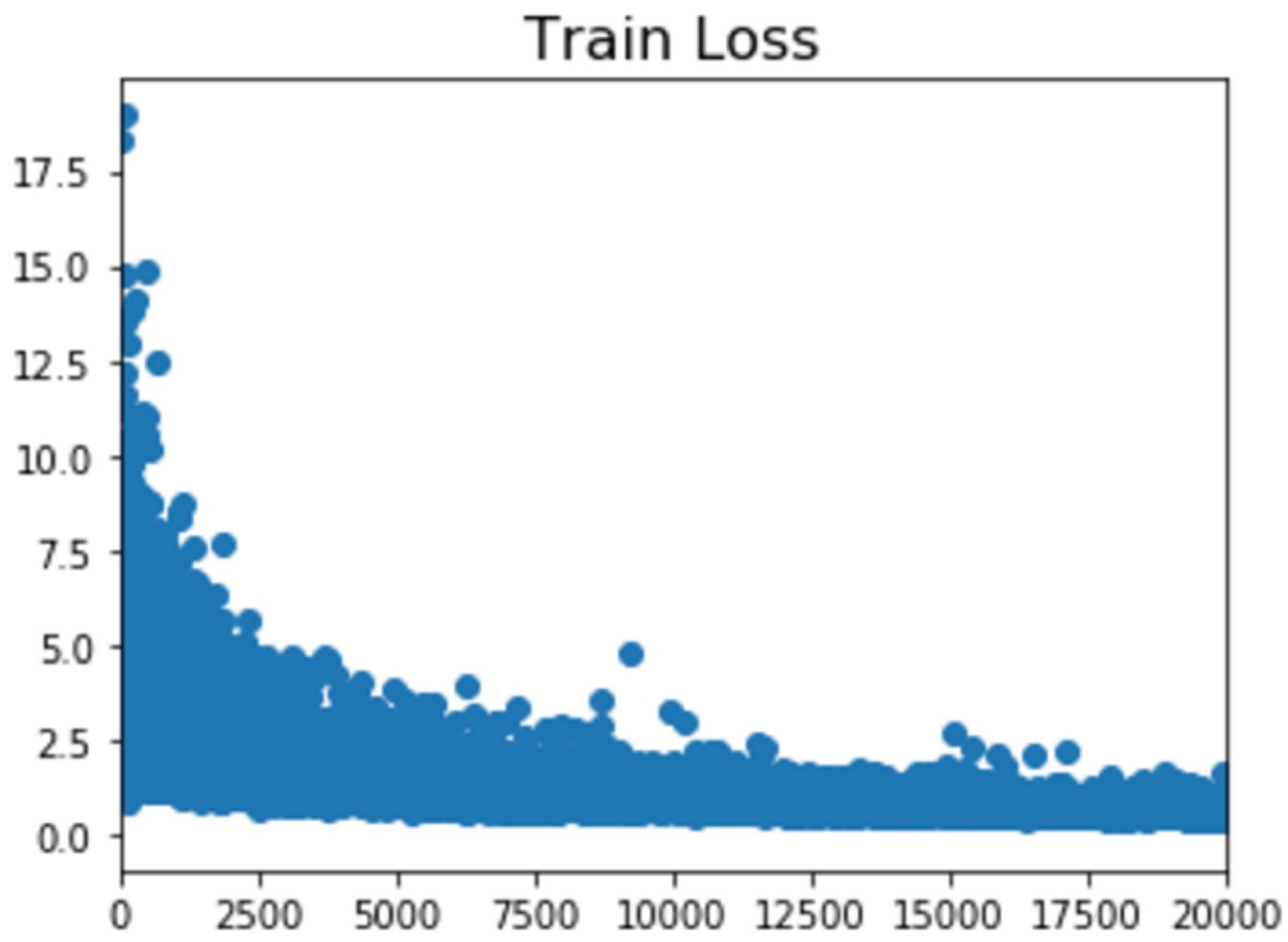
But then $Var(F(x) + x) > Var(x)$ variance grows with each block!

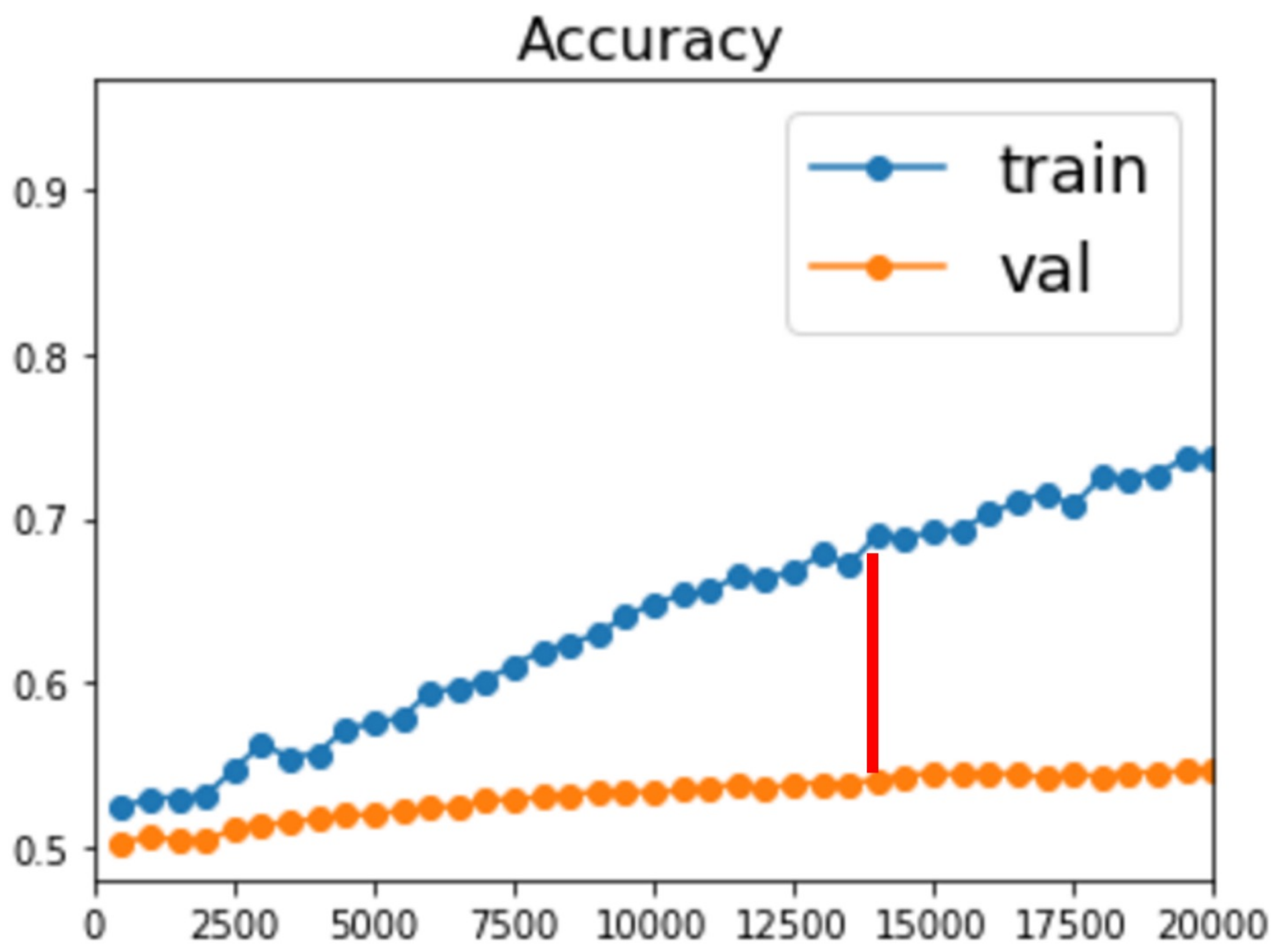
Solution: Initialize first conv with MSRA, initialize second conv to zero. Then $Var(F(x) + x) = Var(x)$

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

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

In common use:

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

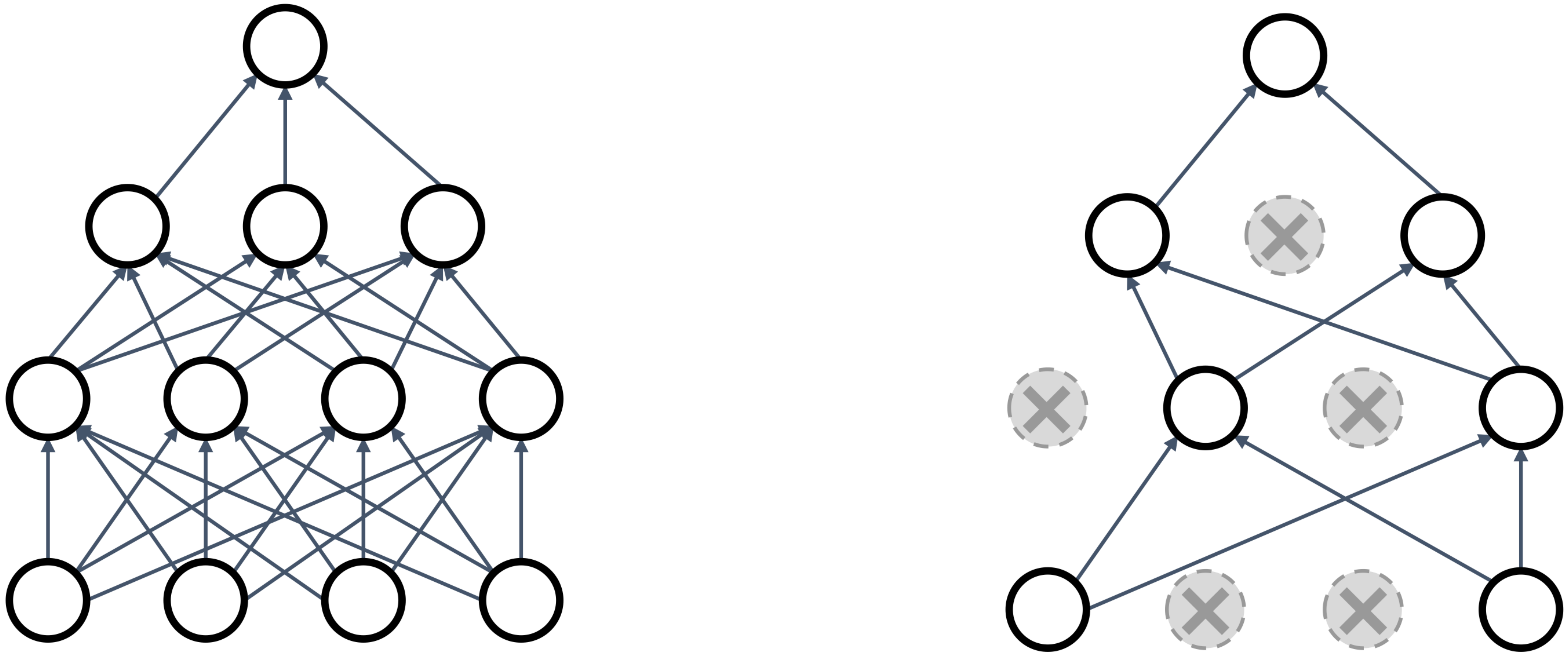
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

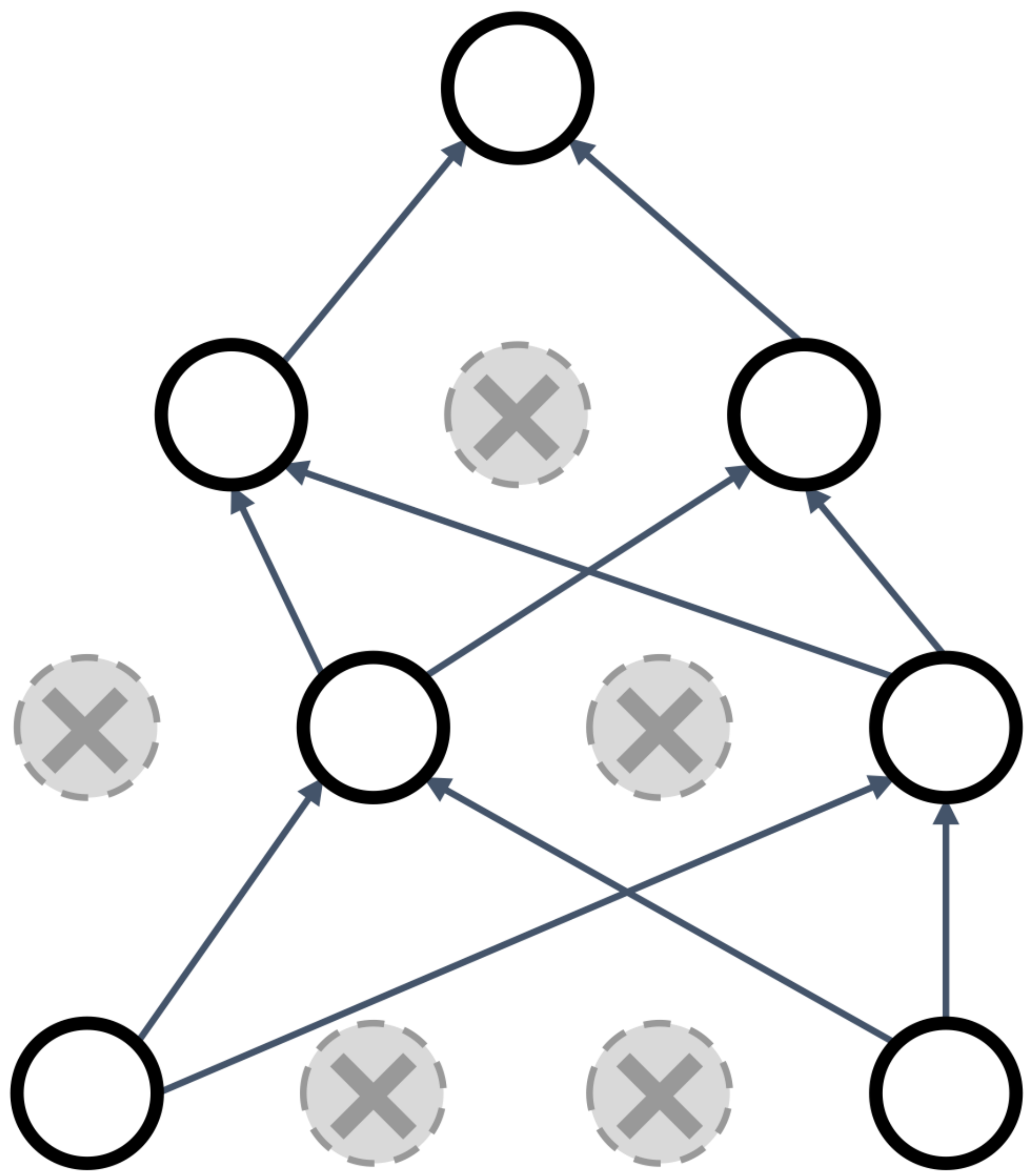


Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

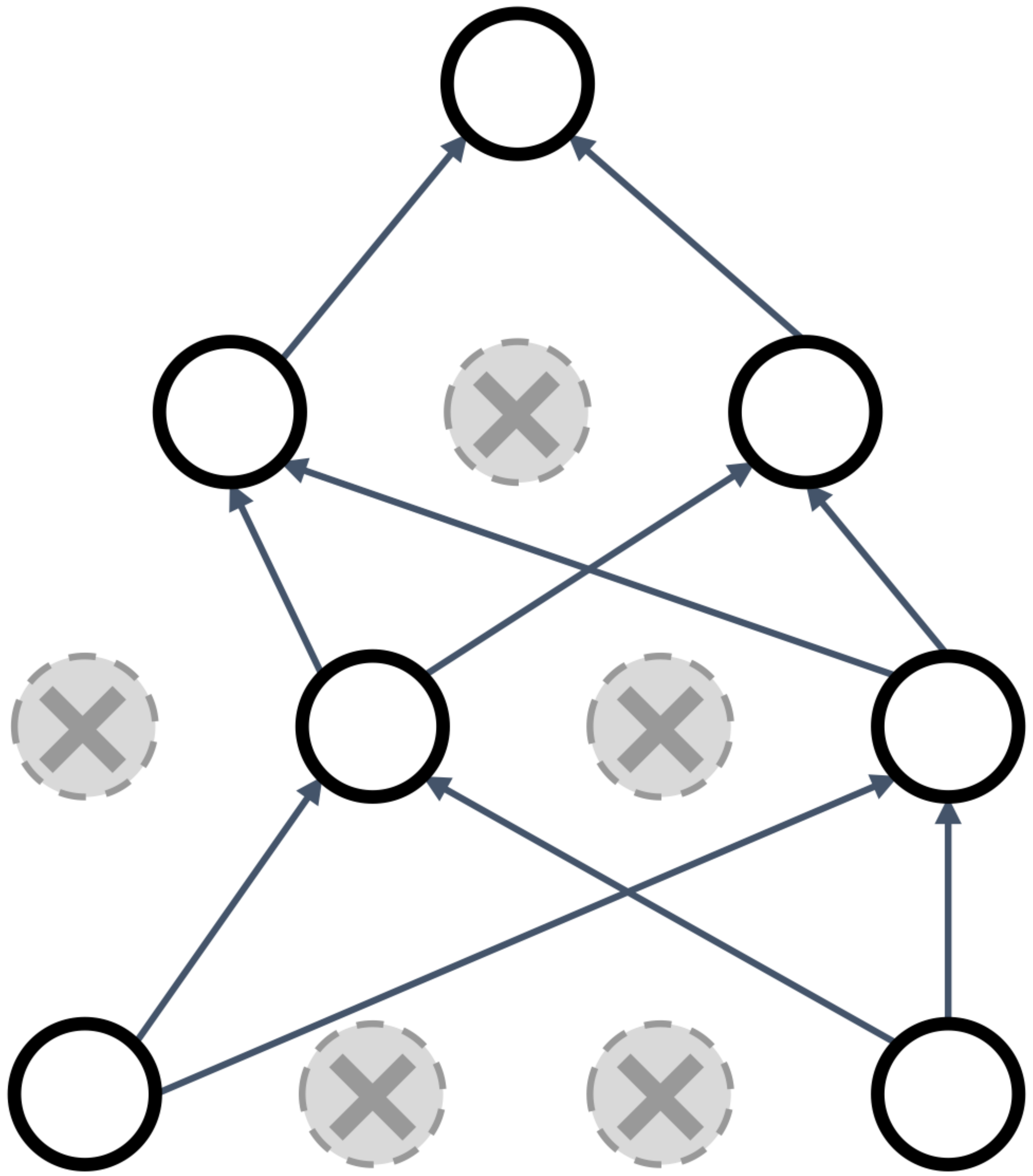
Regularization: Dropout

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

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

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

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

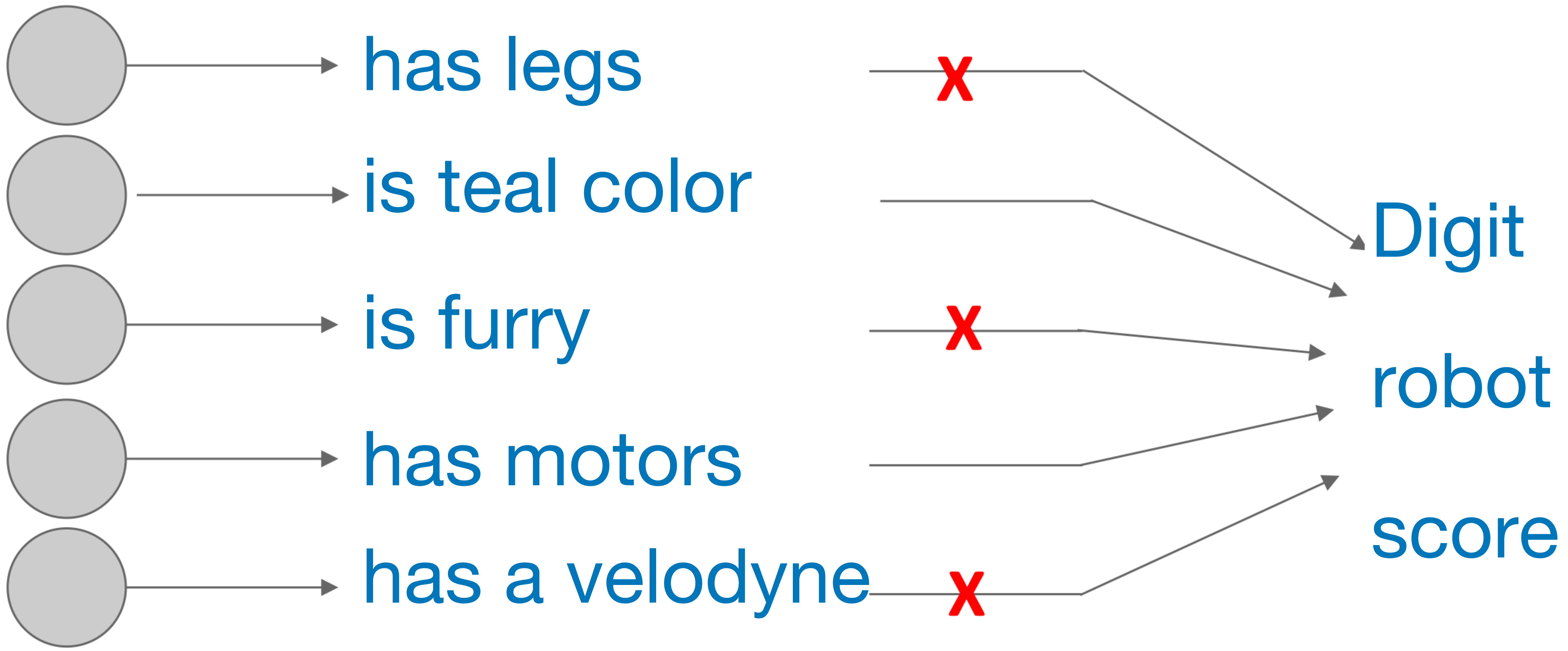
Example forward pass with a 3-layer network using dropout



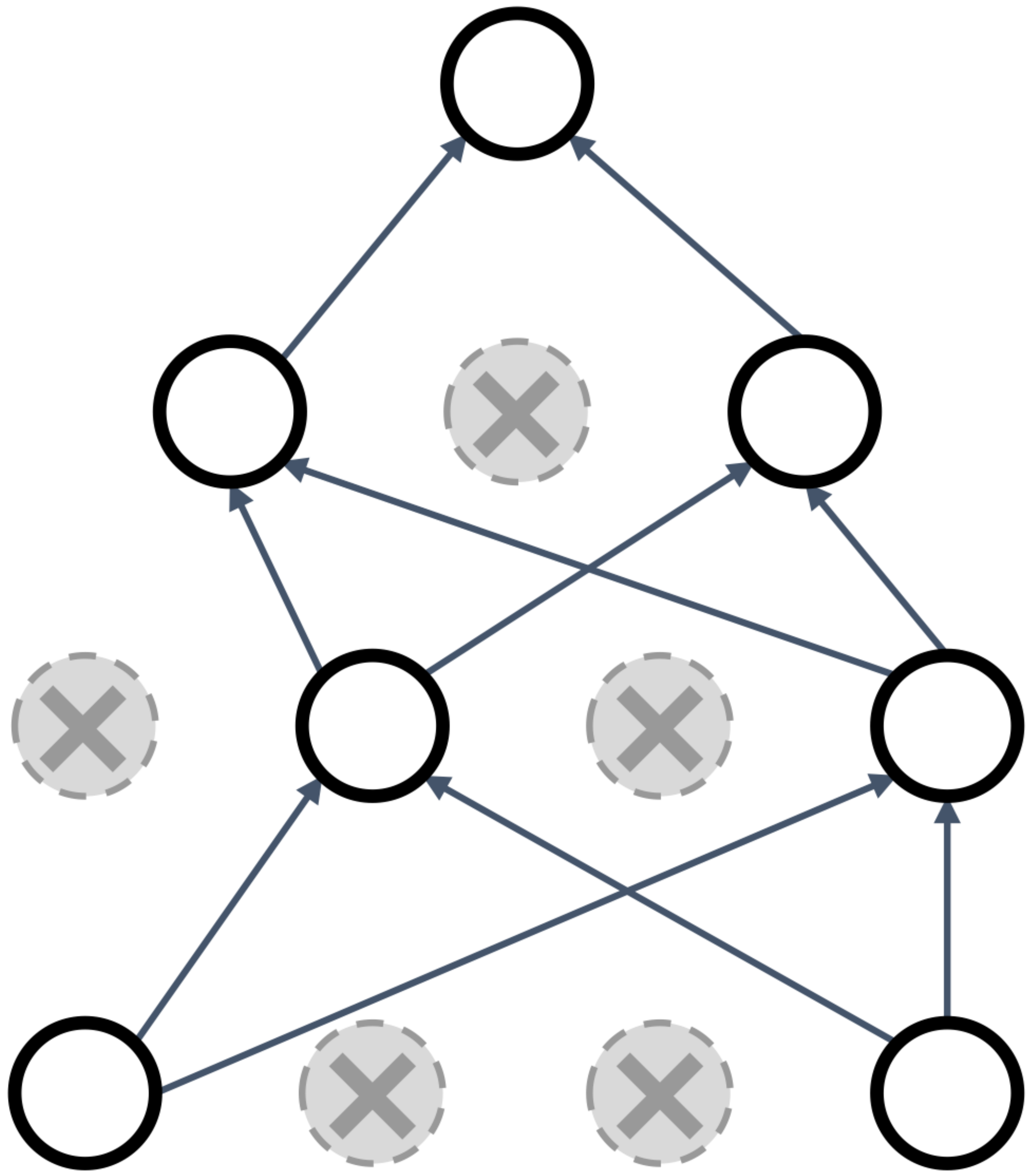
Regularization: Dropout



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



Regularization: Dropout



Another interpretation:

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

Each binary 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

Dropout makes our output random!

$$\underbrace{y}_{\text{Output label}} = f_W(\underbrace{x}_{\text{Input image}}, \overbrace{z}^{\text{Random mask}})$$

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

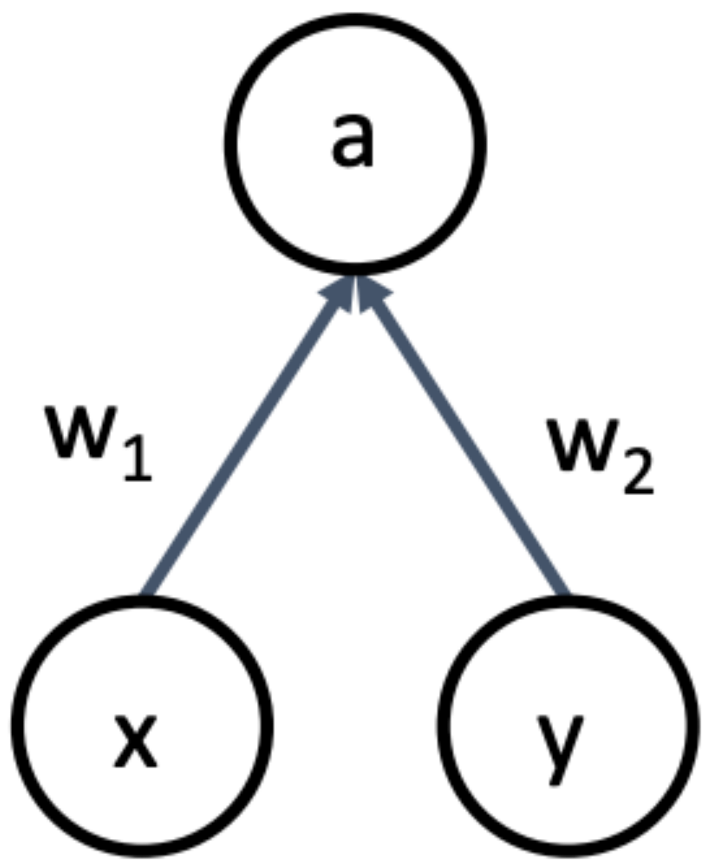
$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

But this integral seems hard…

Dropout: Test time

Want to approximate the integral

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z)f(x, z)dz$$



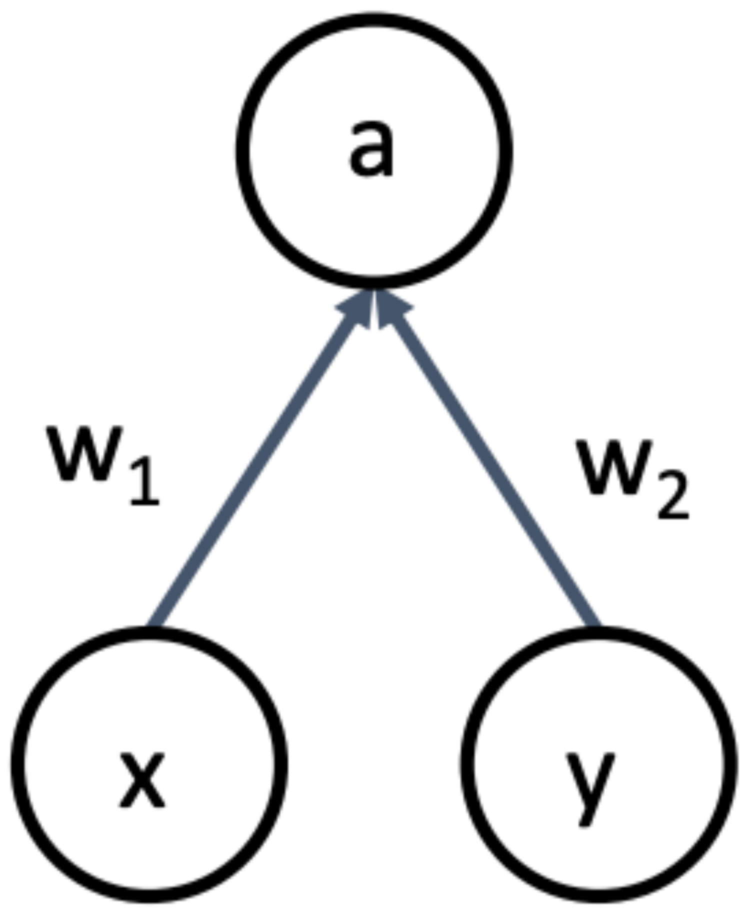
Consider a single neuron:

At test time we have: $\mathbb{E}[a] = w_1x + w_2y$

Dropout: Test time

Want to approximate the integral

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$



Consider a single neuron:

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

During training time we have:

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

$$= \frac{1}{2}(w_1 x + w_2 y)$$

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

Dropout: Test time

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

At test time all neurons are active always

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

Output at test time = Expected output at training time

Dropout Summary

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

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

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

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

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

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

Drop in forward pass

Scale at test time

More common: “Inverted dropout”

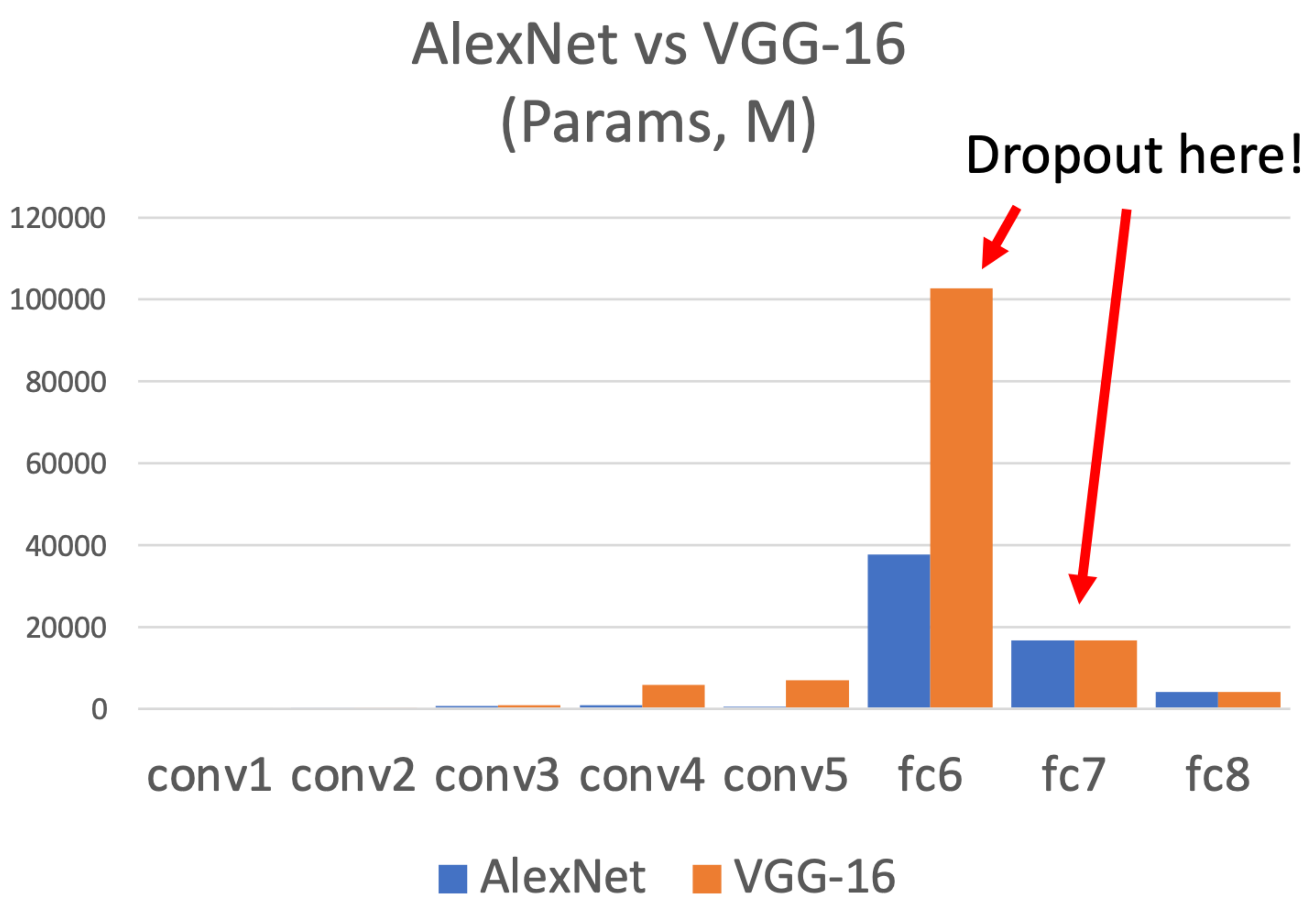
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

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

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

Drop and scale during training

test time is unchanged!

Dropout 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!

Regularization: A common pattern

Training: Add some kind of randomness

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

Testing: Average out randomness (sometimes approximate)

$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

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)

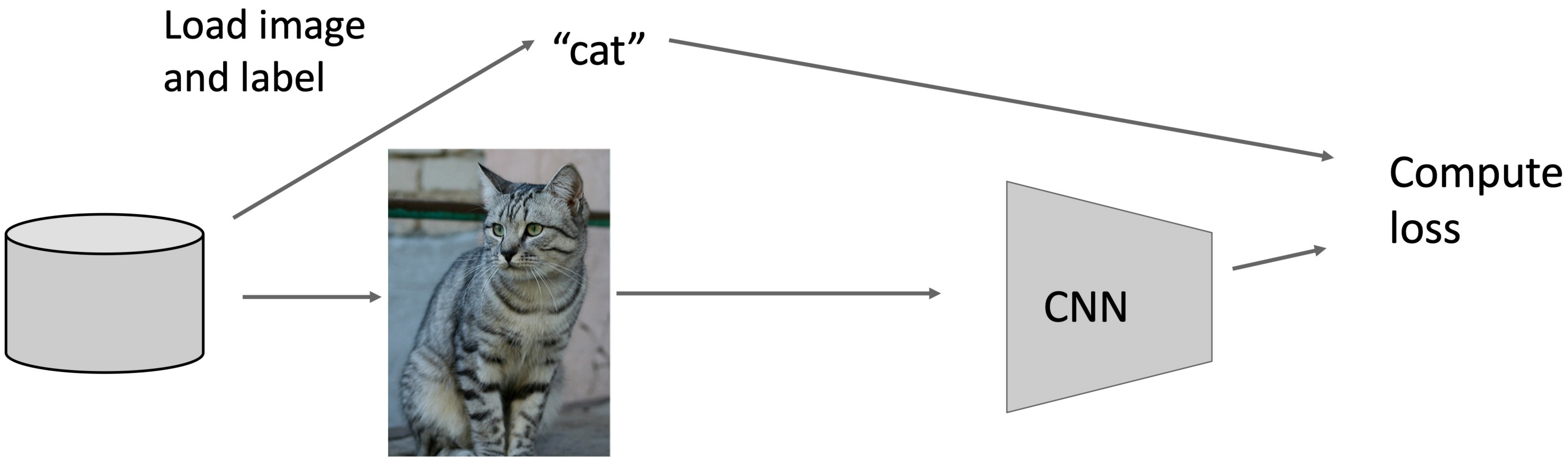
$$y = f(x, z) = \mathbb{E}_z[f(x, z)] = \int p(z) f(x, z) dz$$

Example: Batch Normalization

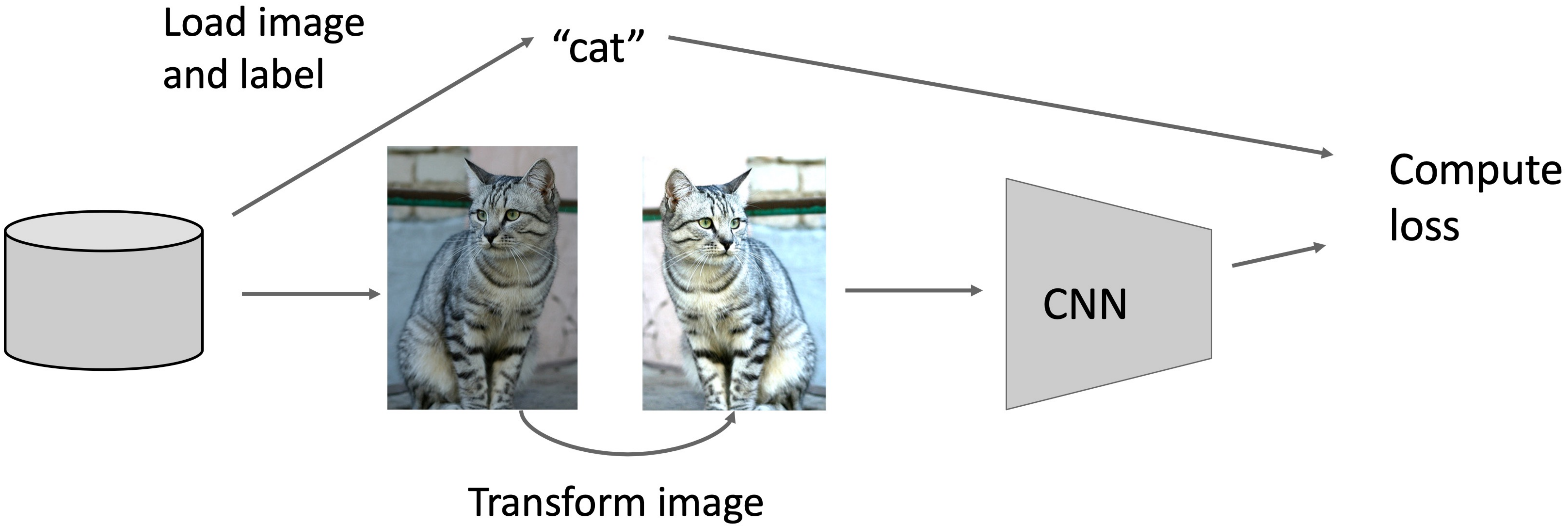
Training: Normalize using stats from random mini batches

Testing: Use fixed stats to normalize

Data augmentation



Data augmentation

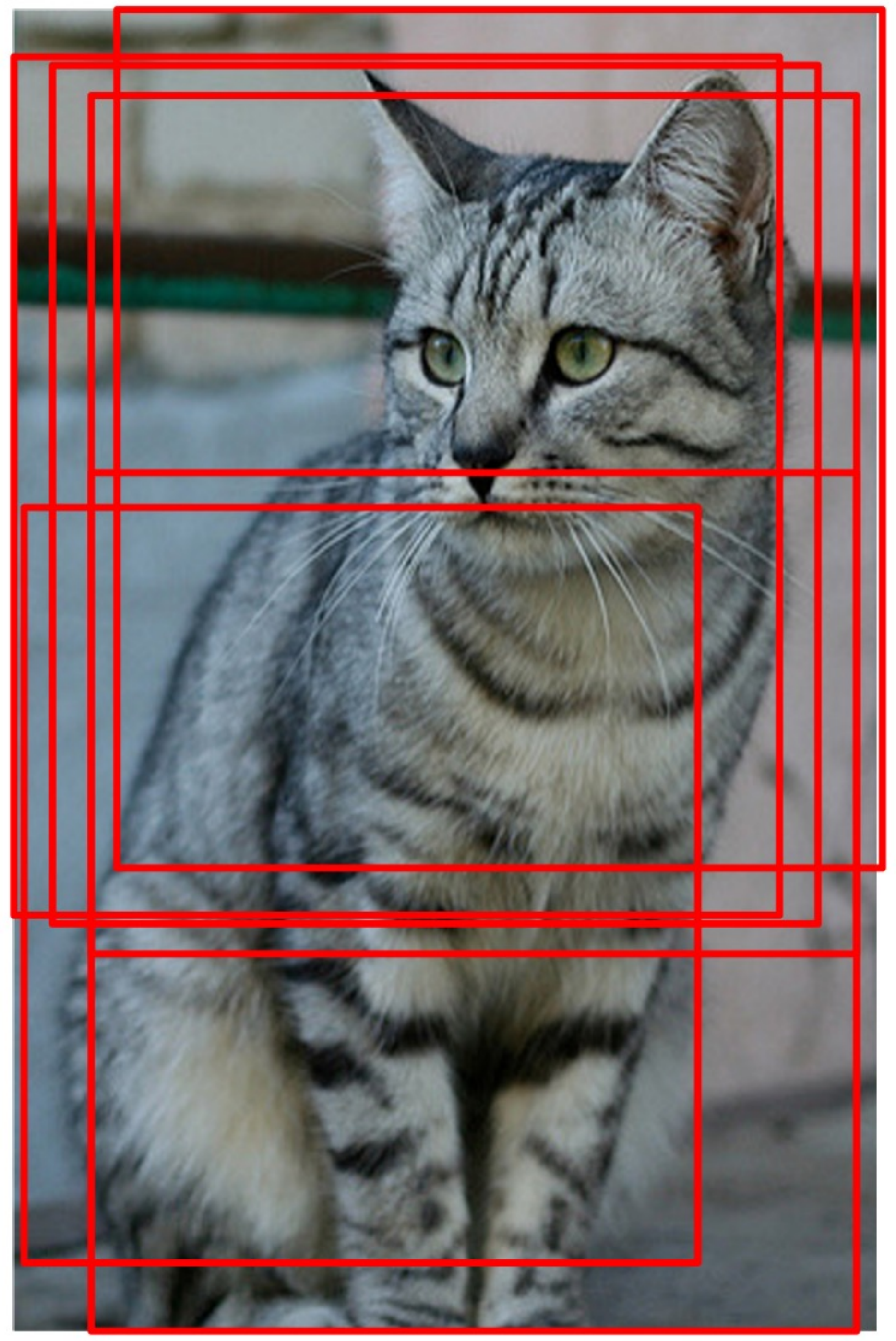


Data augmentation: Horizontal Flips

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



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: RandAugment

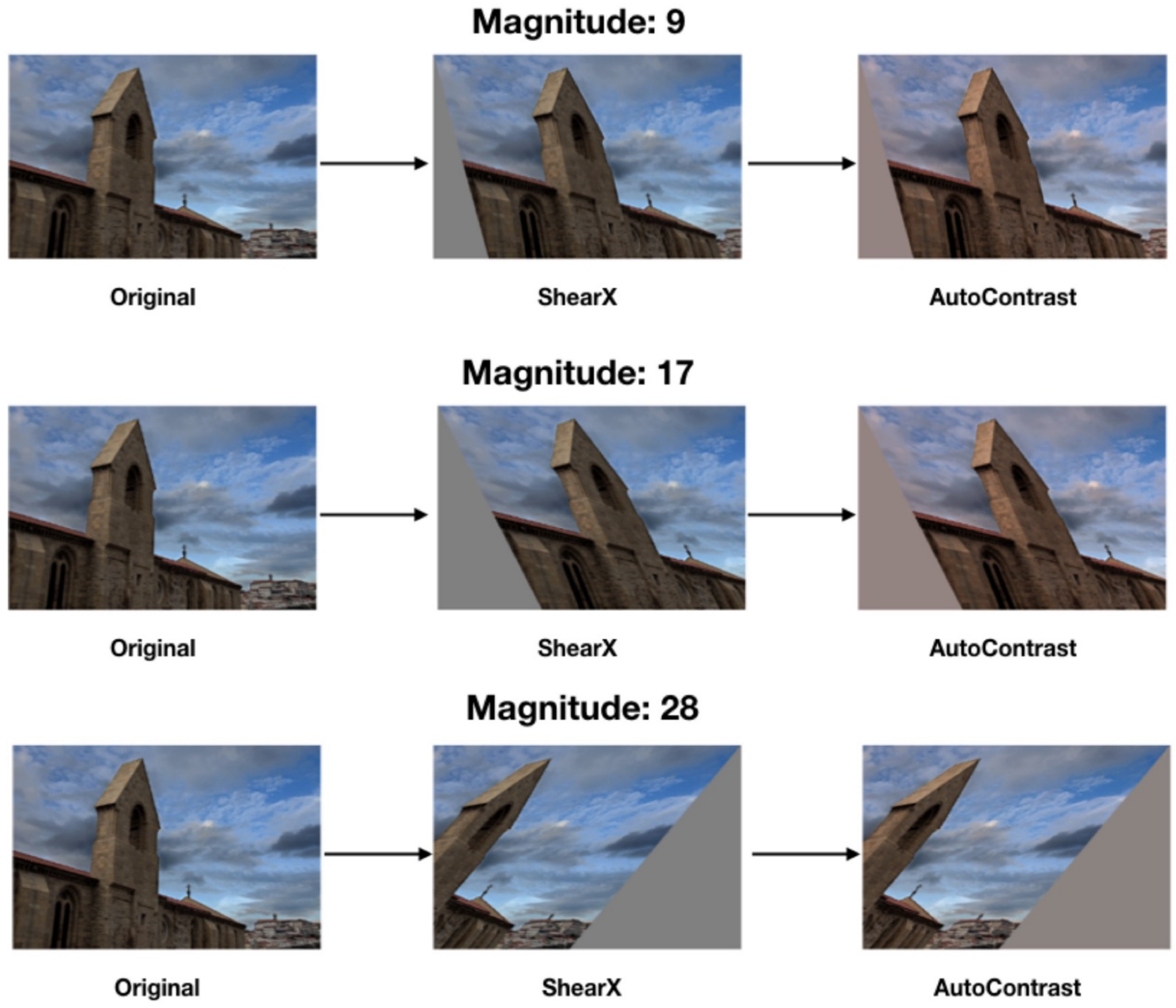
```python
transforms = [
'Identity', 'AutoContrast', 'Equalize',
'Rotate', 'Solarize', 'Color', 'Posterize',
'Contrast', 'Brightness', 'Sharpness',
'ShearX', 'ShearY', 'TranslateX', 'TranslateY']

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]
```

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: RandAugment



Apply random combinations of transforms:

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

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?

Maybe different for different tasks!

Regularization: A common pattern

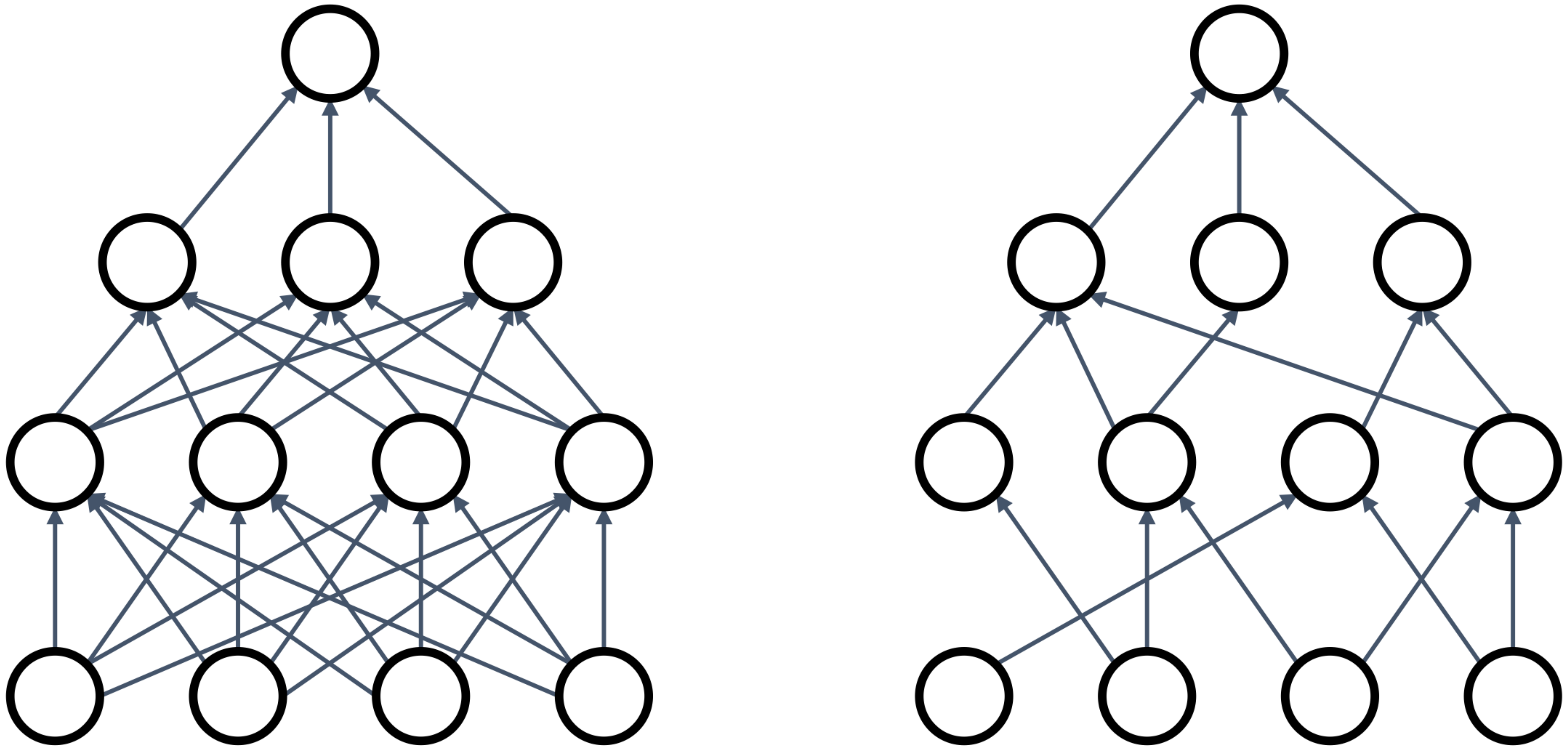
Training: Add some randomness
Testing: Marginalize over randomness

Examples:
Dropout
Batch Normalization
Data Augmentation

Regularization: DropConnect

Training: Drop random connections between neurons (set weight=0)
Testing: Use all the connections

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect

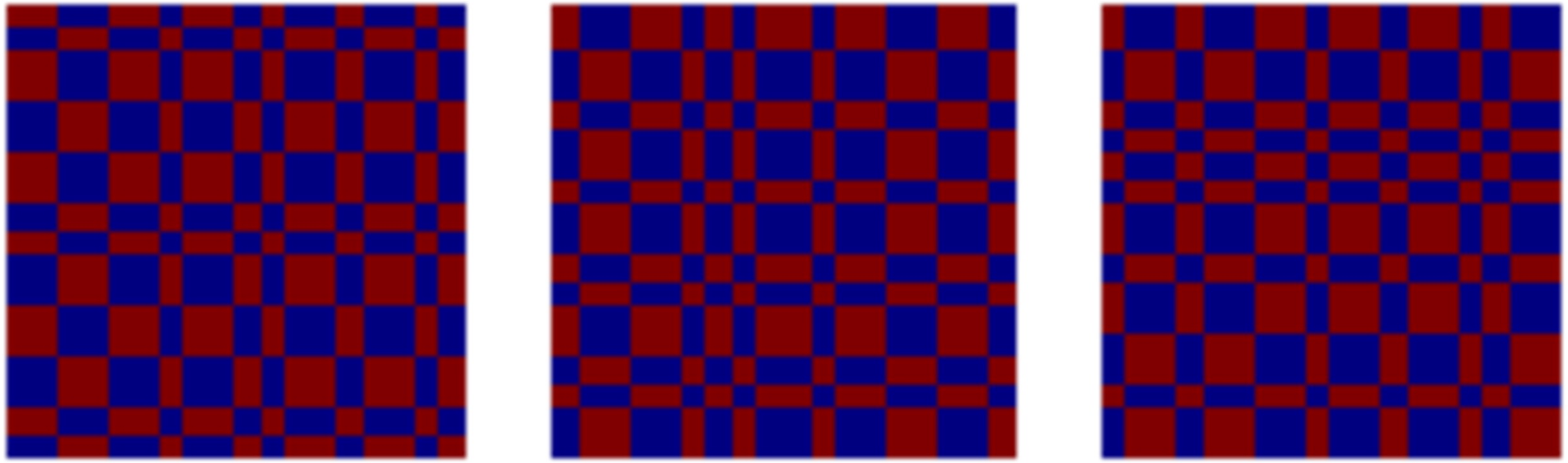


Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

Regularization: Fractional Pooling

Training: Use randomized pooling regions
Testing: Average predictions over different samples

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

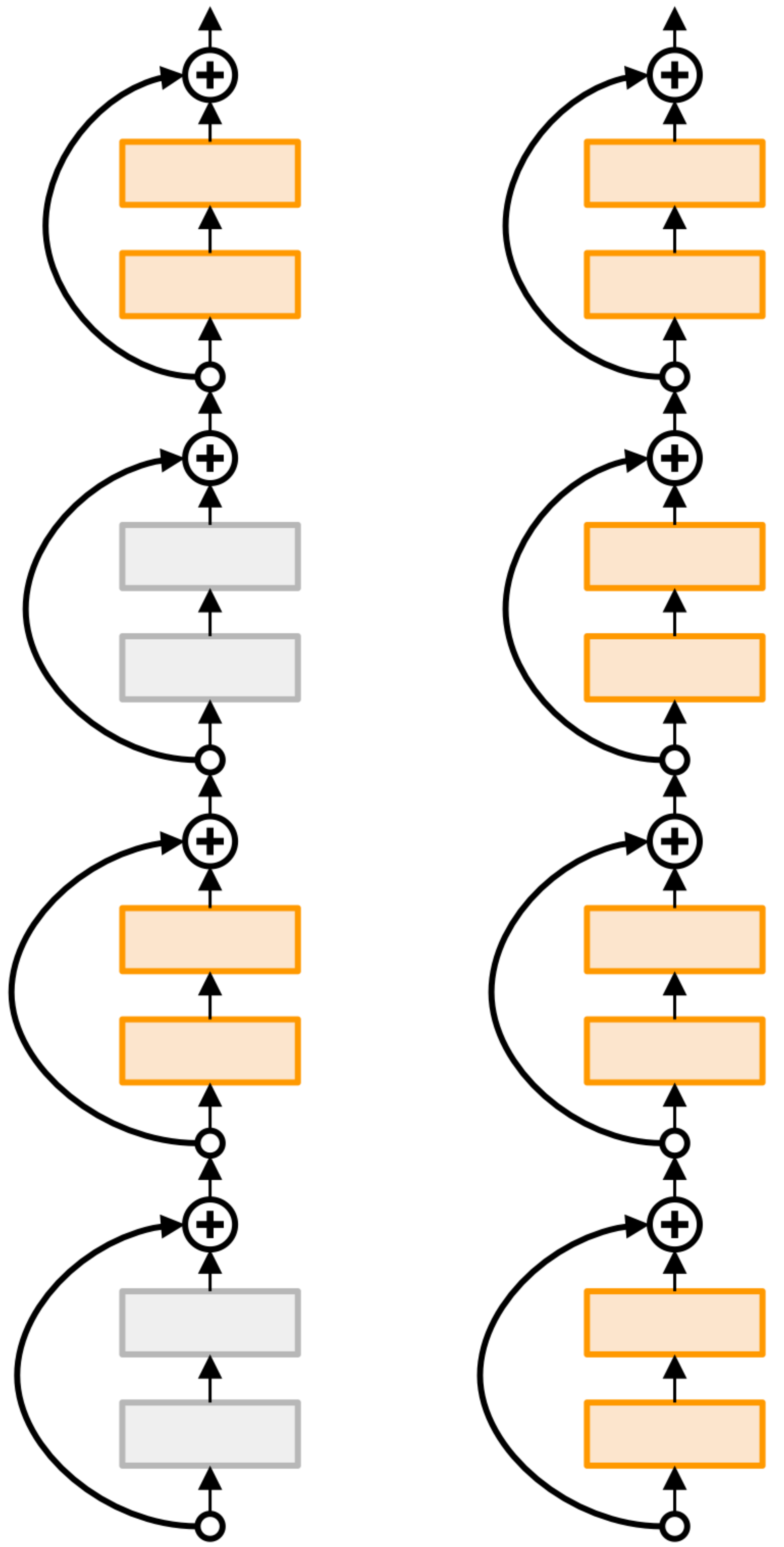
Regularization: Stochastic Depth

Training: Skip some residual blocks in ResNet
Testing: Use the whole network

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
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

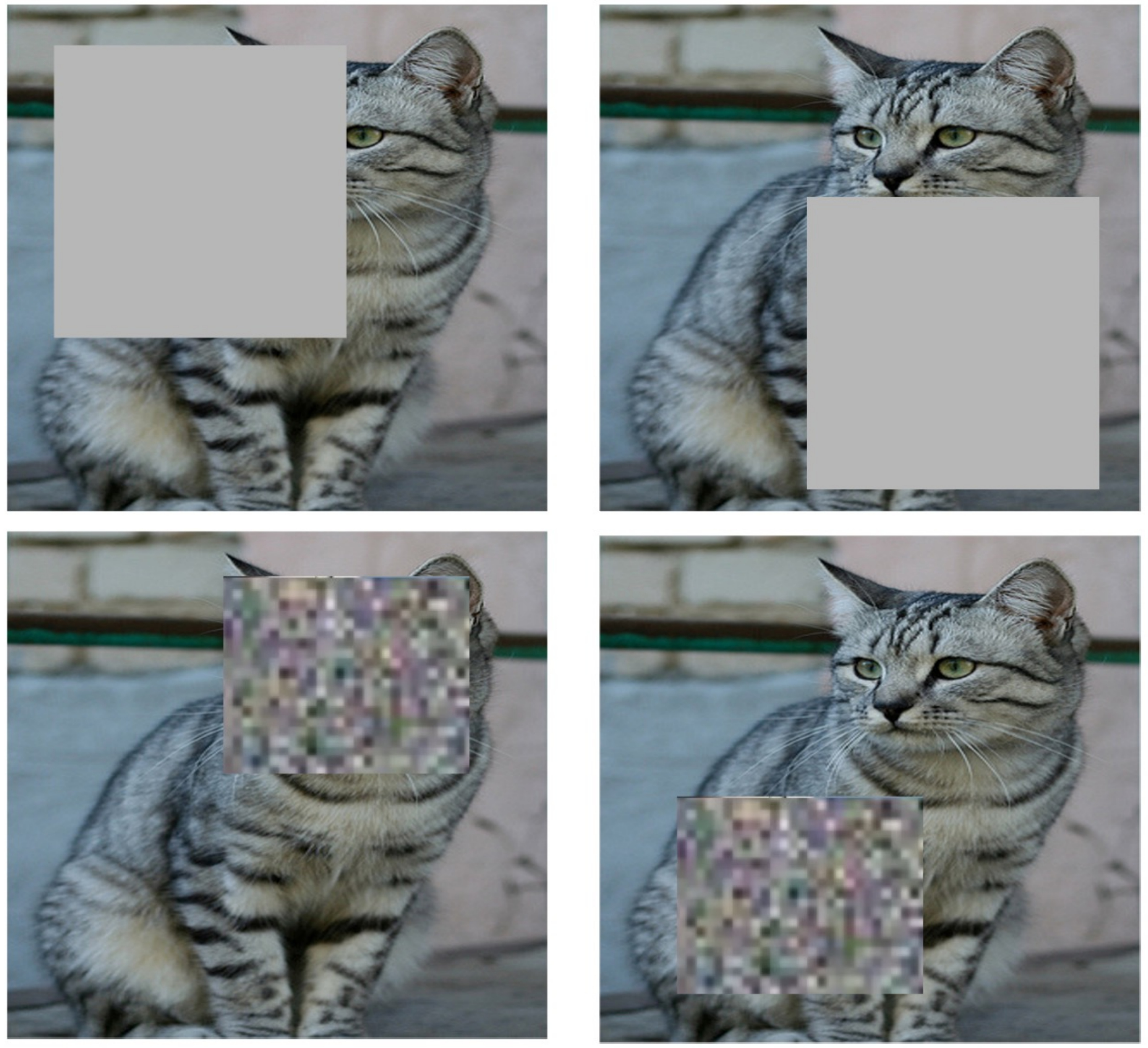


Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

Regularization: CutOut

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

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



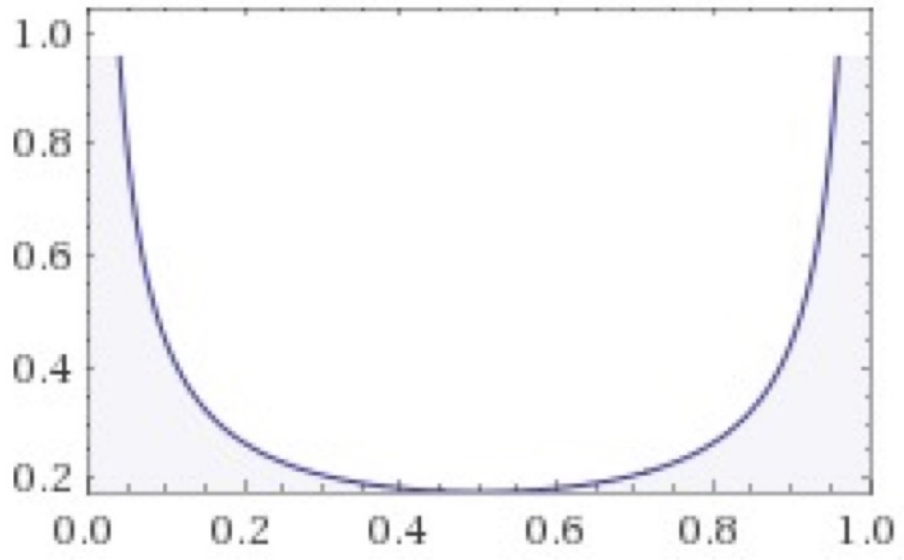
Replace random regions with mean value or random values

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

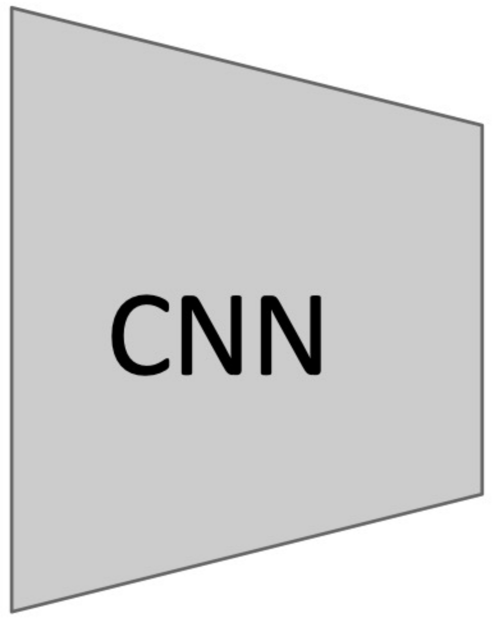
Regularization: Mixup

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

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Erasing
Mixup



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



Target label:
cat: 0.4
dog: 0.6

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

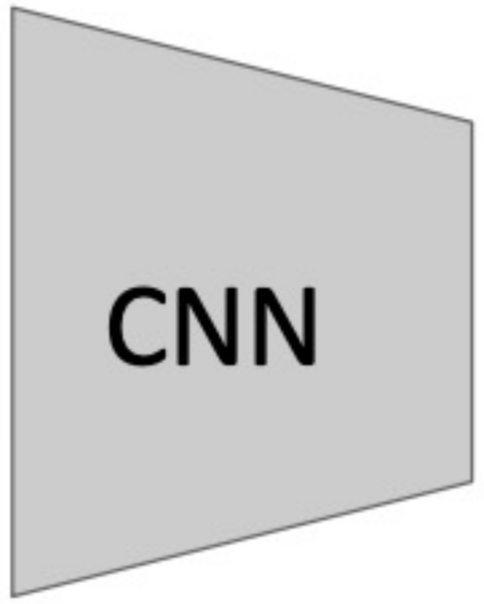
Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Regularization: Mixup

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

Another example

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Erasing
Mixup



Target label:
Pretzels: 0.6
Robot: 0.4

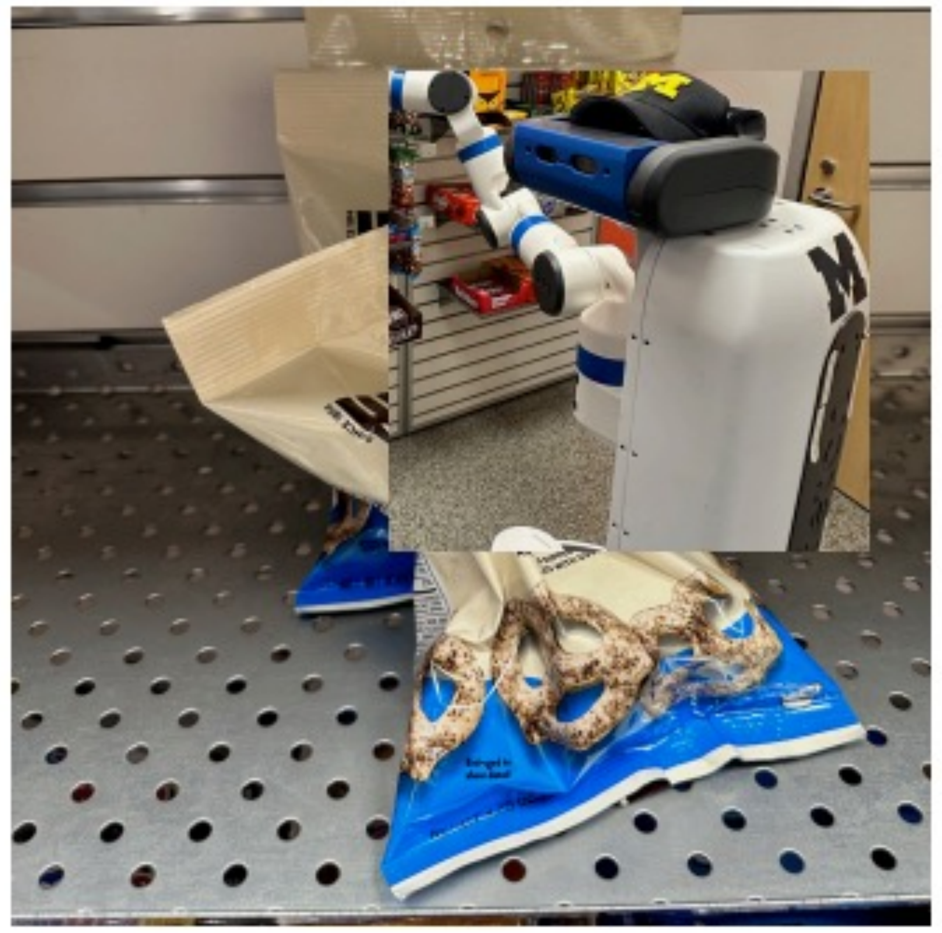
Randomly blend the pixels of pairs of training images, e.g. 60% pretzels, 40% robot

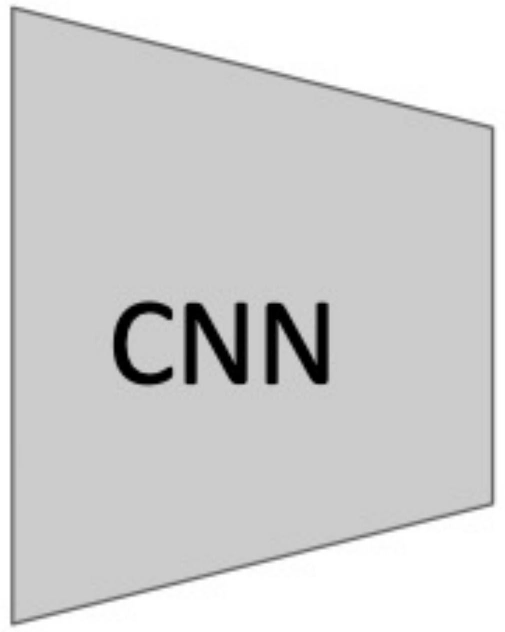
Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

Regularization: CutMix

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

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix





Target label:
Pretzels: 0.6
Robot: 0.4

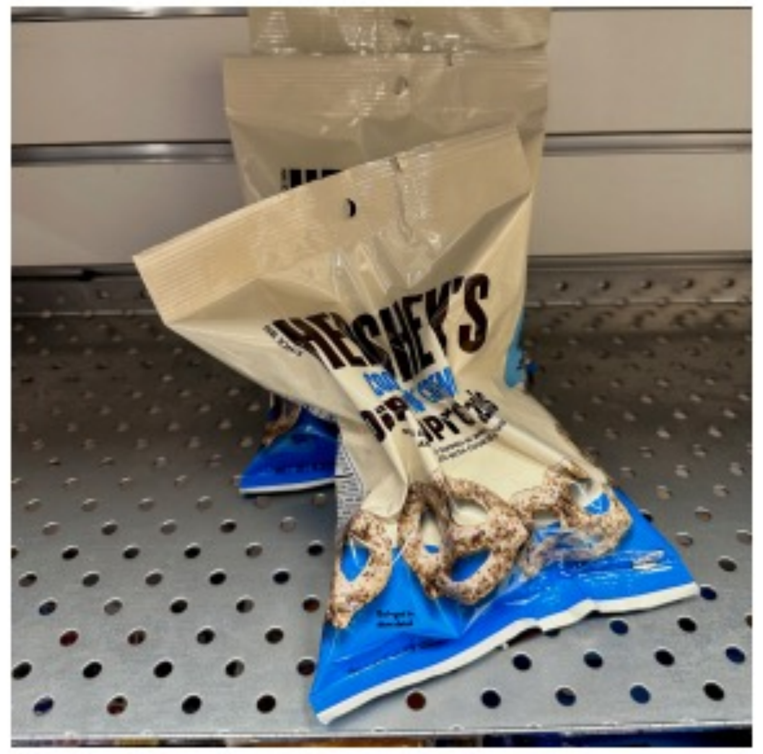
Replace random crops of one image with another, e.g. 60% of pixels from pretzels, 40% from robot

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

Regularization: Label Smoothing

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

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix
Label Smoothing



Standard Training	Label Smoothing
Pretzels: 100%	Pretzels: 90%
Robot: 0%	Robot: 5%
Sugar: 0%	Sugar: 5%

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

Loss is cross-entropy between predicted and target distribution.

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

Regularization: Summary

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

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Erasing
Mixup / CutMix
Label Smoothing

- Use DropOut for large fully-connected layers
- Data augmentation is 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

Summary

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

DeepRob

Lecture 10
Training Neural Networks I
University of Michigan | Department of Robotics