

Lecture 10: Training Neural Networks (Part 1)

Reminder: A3

- Due Friday, October 9

Midterm Exam

We are still working out details! Will share more on Wednesday

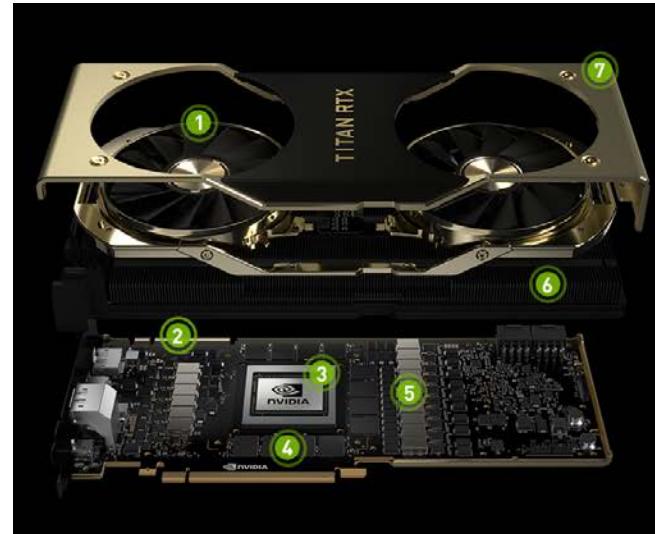
- Will (most likely) be online via <https://crabster.org/>
- Material up to Lecture 13 is fair game
- Mostly conceptual questions, no coding
- Some combination of:
 - True / False
 - Multiple choice
 - Short answer (math on paper)
- If you need accommodations, send your SSD letter to me

Last Time: Hardware and Software

CPU



GPU



TPU



**Static Graphs vs
Dynamic Graphs**

**PyTorch vs
TensorFlow**

Overview

1. One time setup

Activation functions, data preprocessing, weight initialization, regularization

2. Training dynamics

Learning rate schedules; large-batch training;
hyperparameter optimization

3. After training

Model ensembles, transfer learning

Overview

1. One time setup

Activation functions, data preprocessing, weight initialization, regularization

Today

2. Training dynamics

Learning rate schedules; large-batch training;
hyperparameter optimization

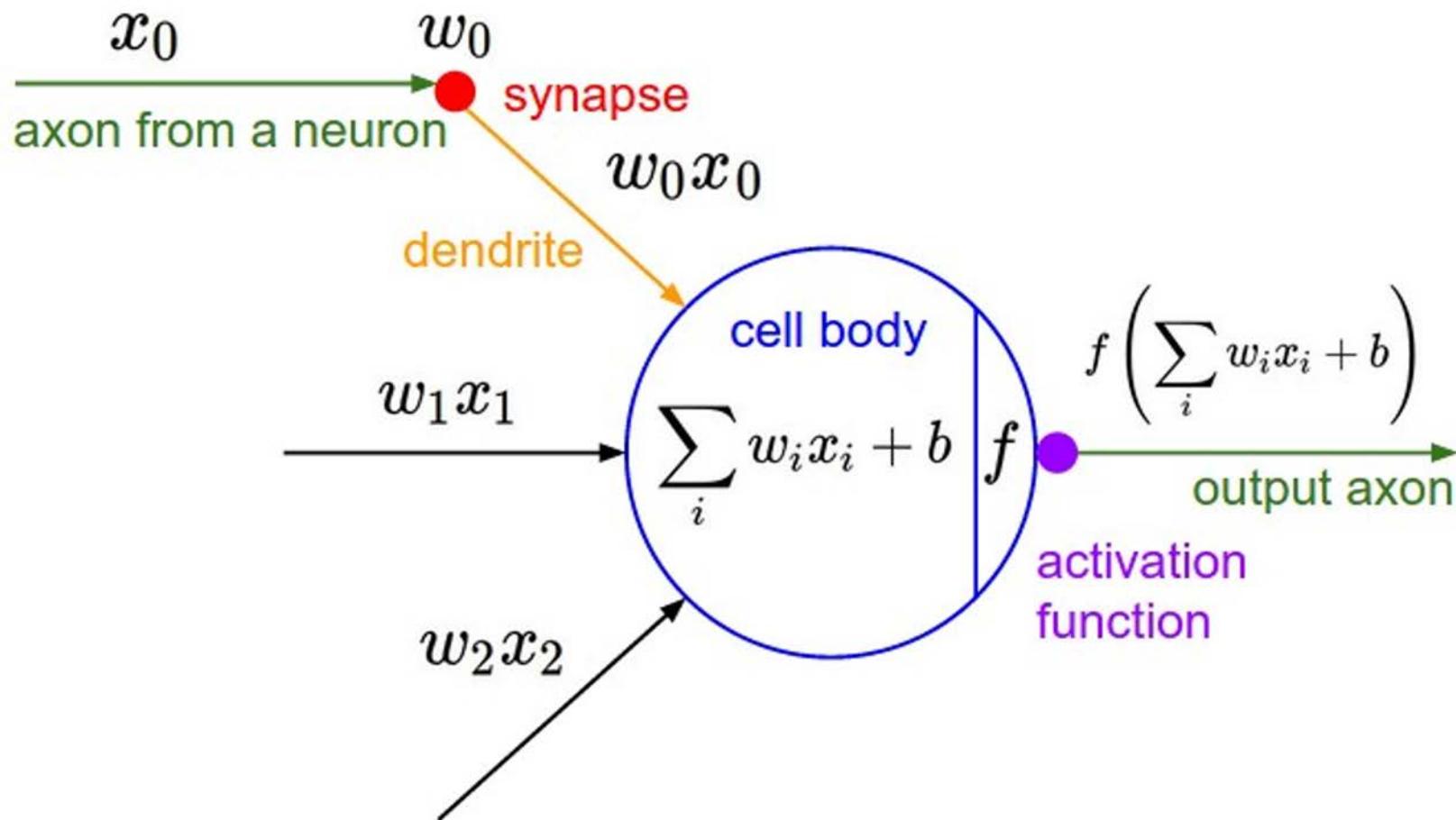
Next time

3. After training

Model ensembles, transfer learning

Activation Functions

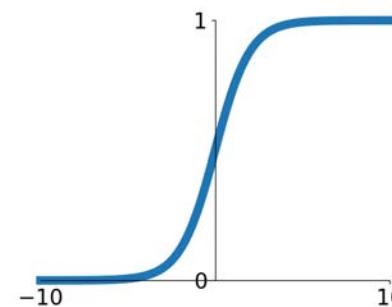
Activation Functions



Activation Functions

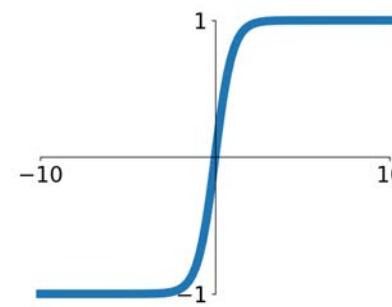
Sigmoid

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



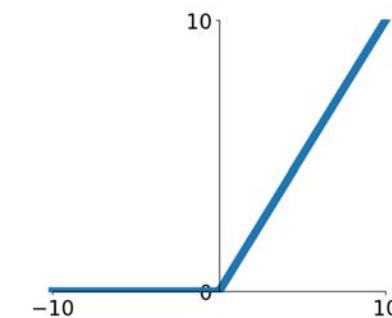
tanh

$$\tanh(x)$$



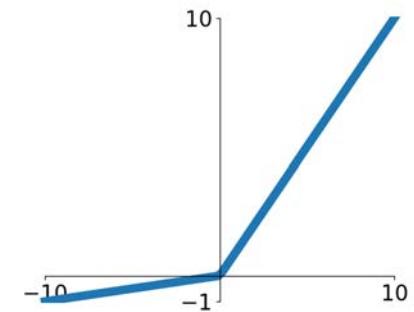
ReLU

$$\max(0, x)$$



Leaky ReLU

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

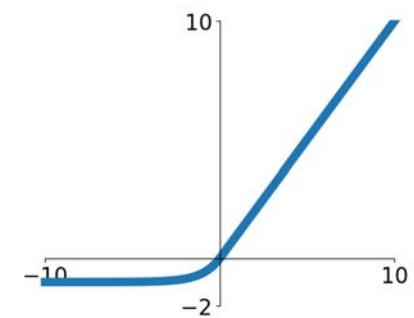


Maxout

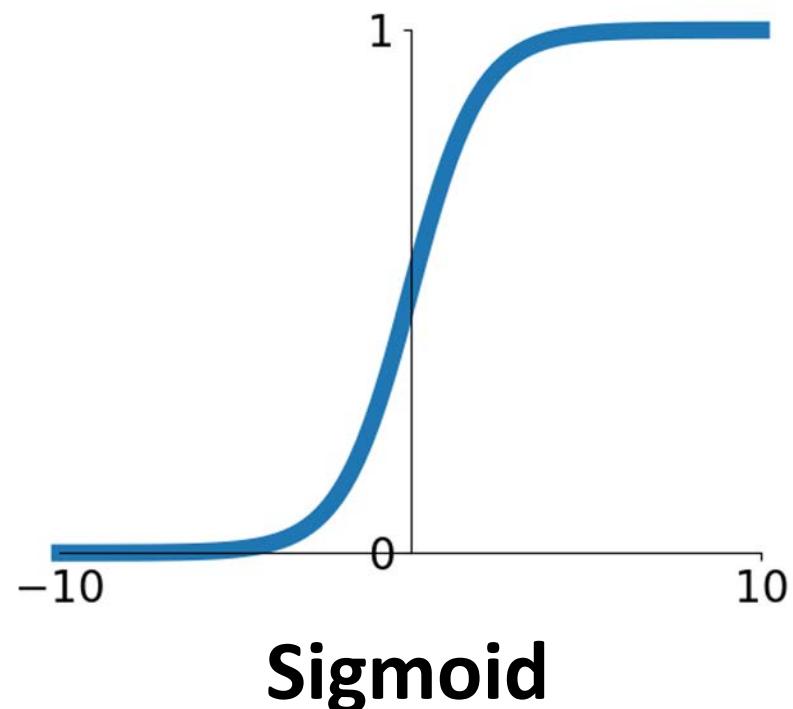
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



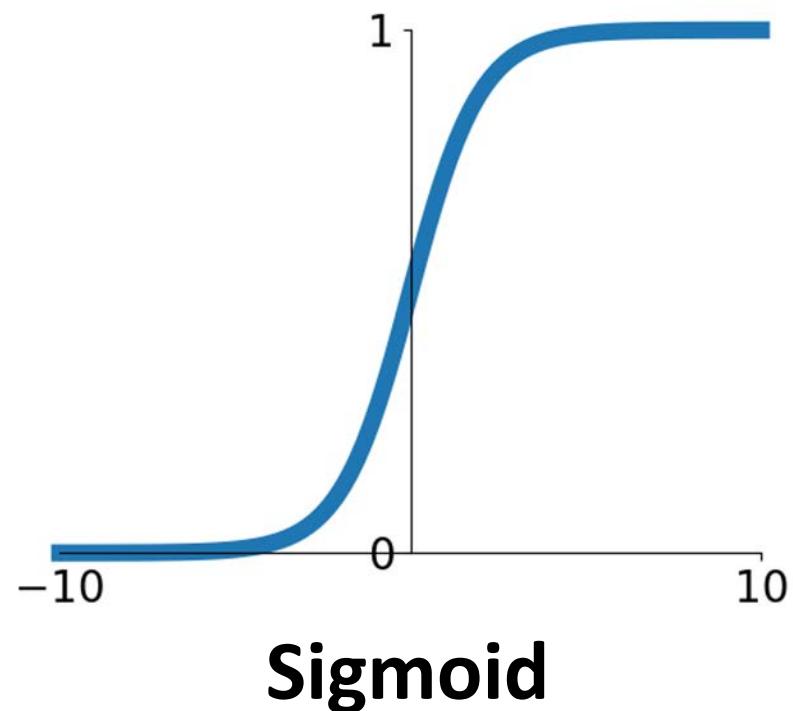
Activation Functions: Sigmoid



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

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

Activation Functions: Sigmoid



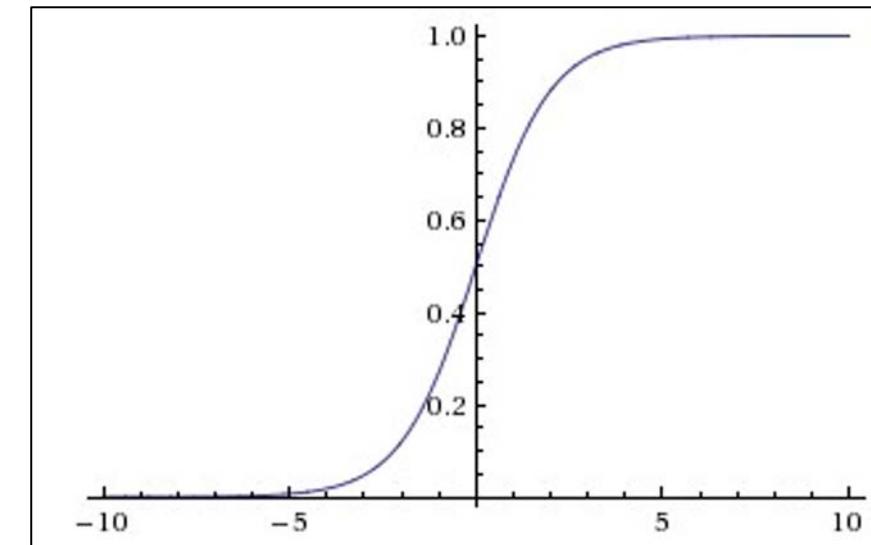
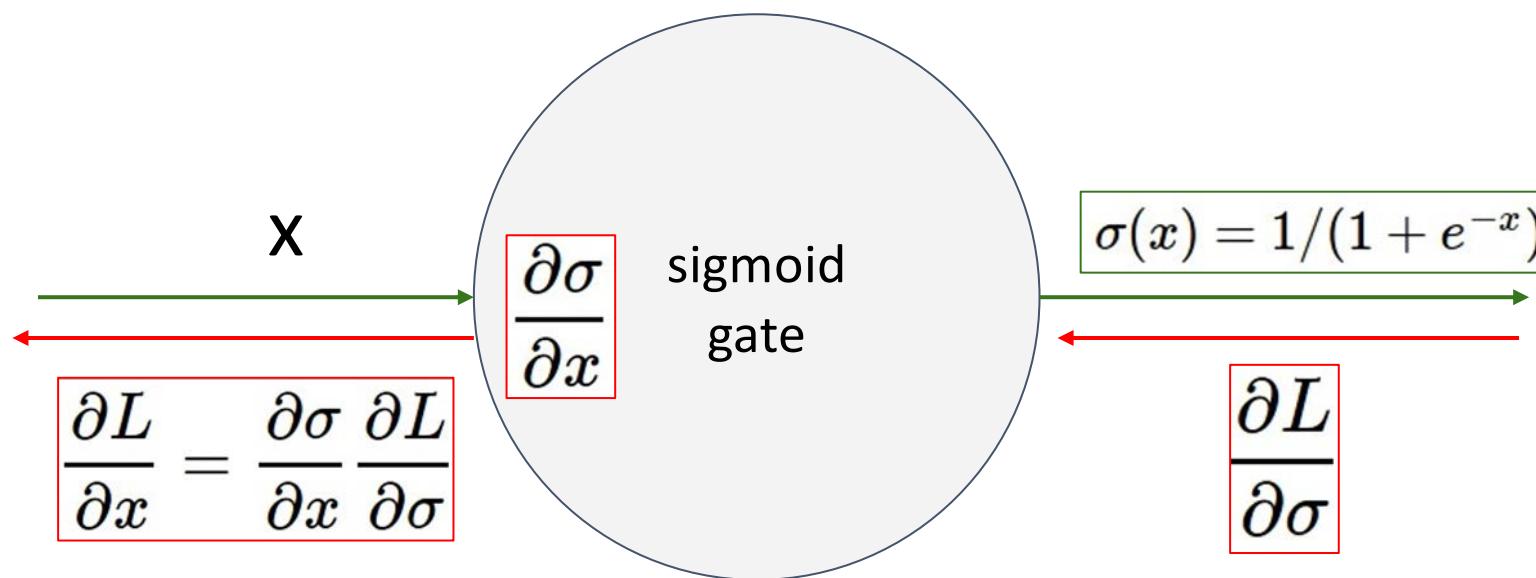
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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

3 problems:

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

Activation Functions: Sigmoid

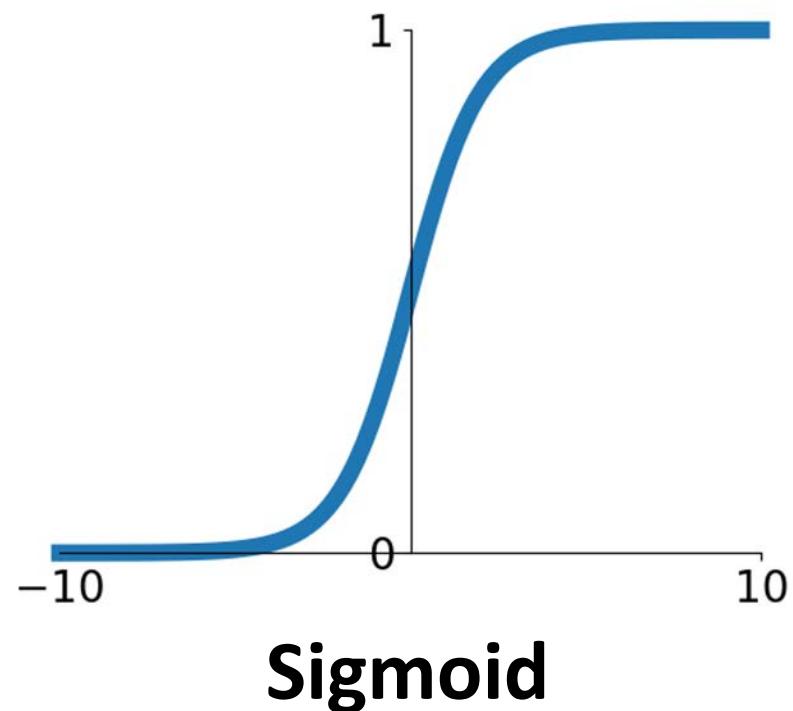


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

Activation Functions: Sigmoid



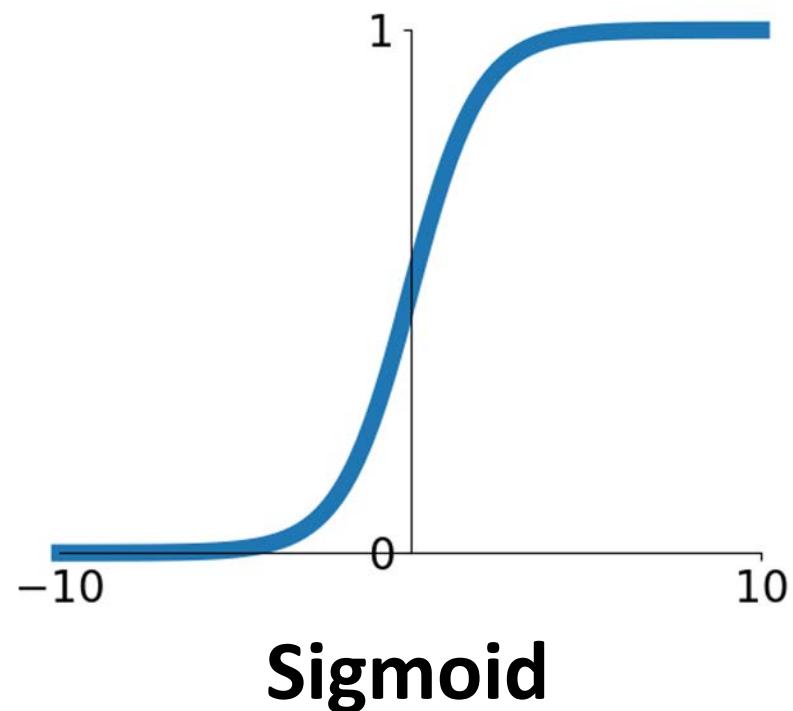
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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



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

- 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

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

$h_i^{(\ell)}$ is the i th 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 $w^{(\ell)}$?

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 i th 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 $w^{(\ell)}$?

Consider what happens when nonlinearity is always positive

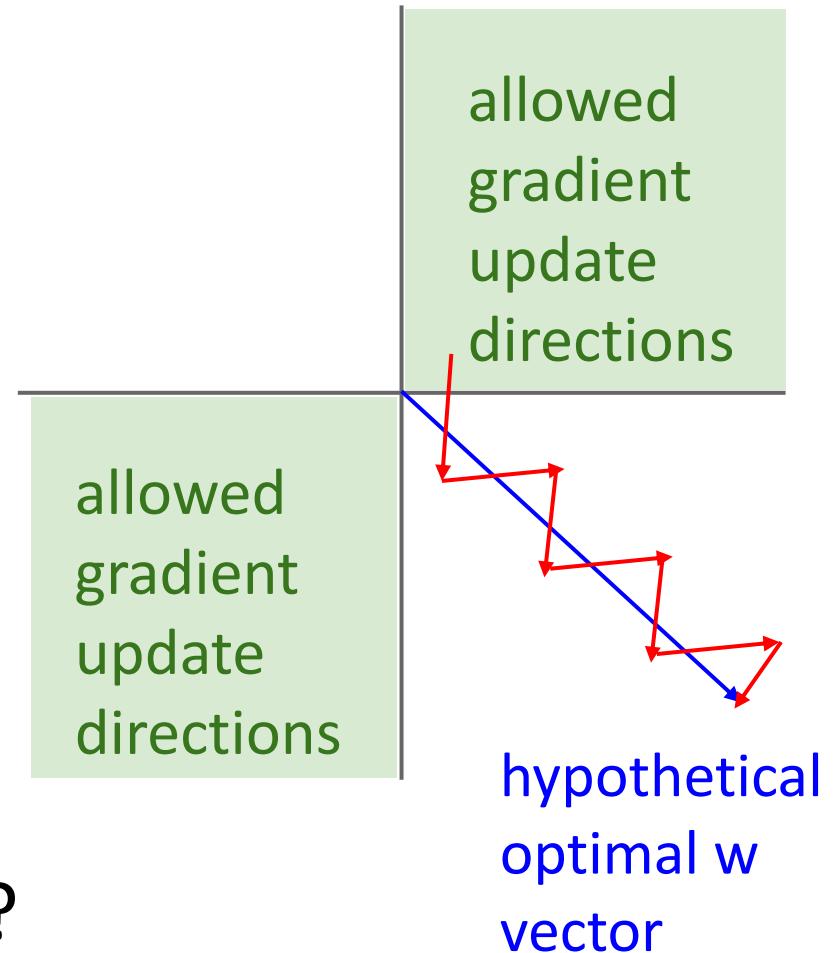
$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma(h_j^{(\ell-1)}) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the i th 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 $w^{(\ell)}$?

Always all positive or all negative :(



Consider what happens when nonlinearity is always positive

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

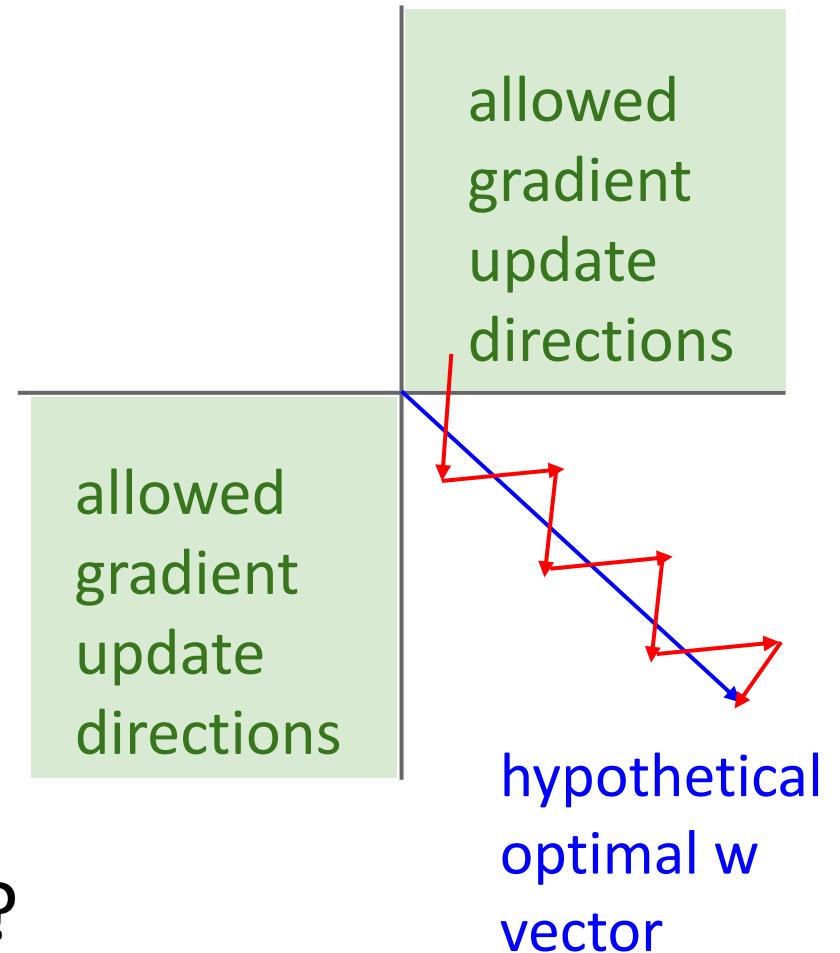
$h_i^{(\ell)}$ is the i th element of the hidden layer at layer ℓ (before activation)

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

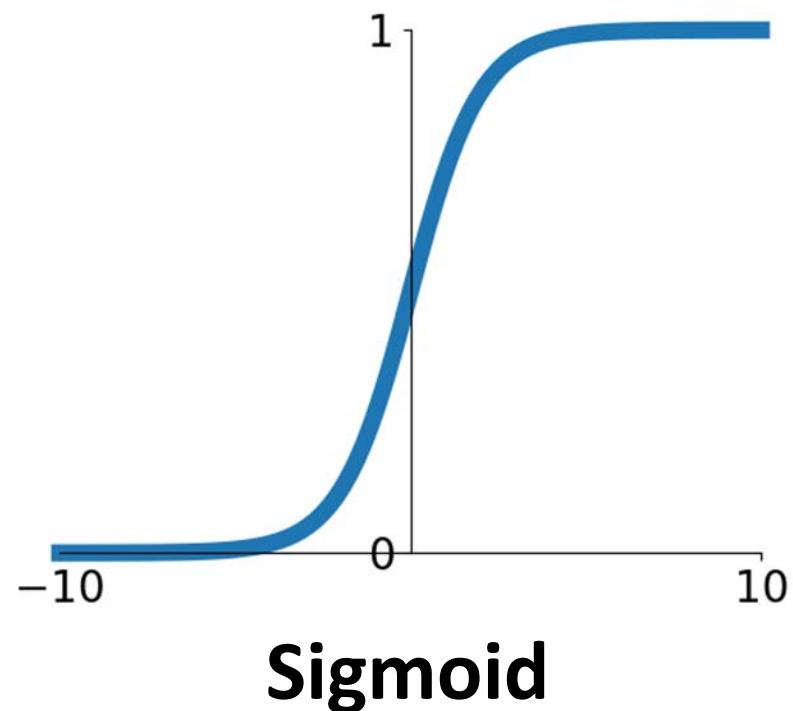
What can we say about the gradients on $w^{(\ell)}$?

Always all positive or all negative :(

(For a single element! Minibatches help)



Activation Functions: Sigmoid



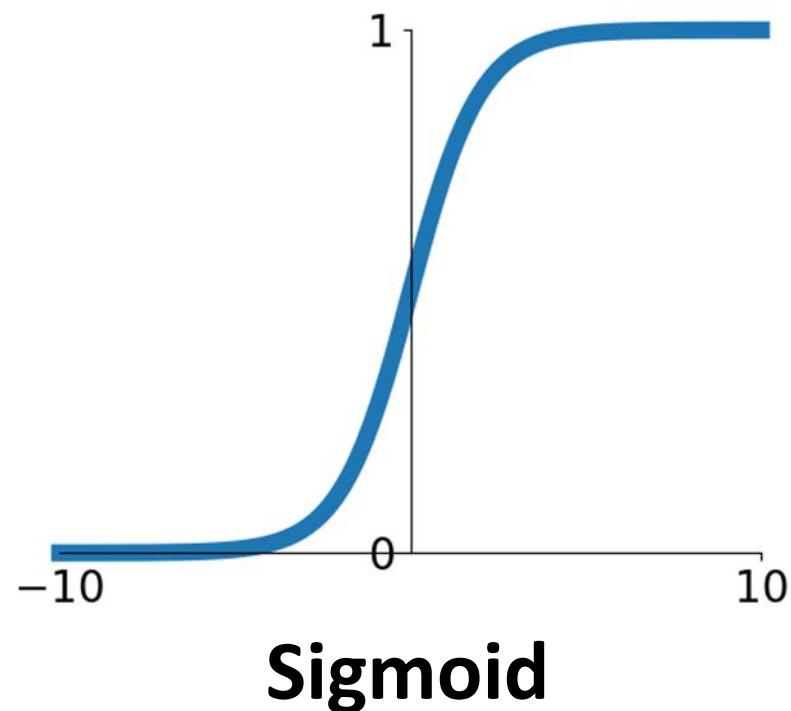
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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



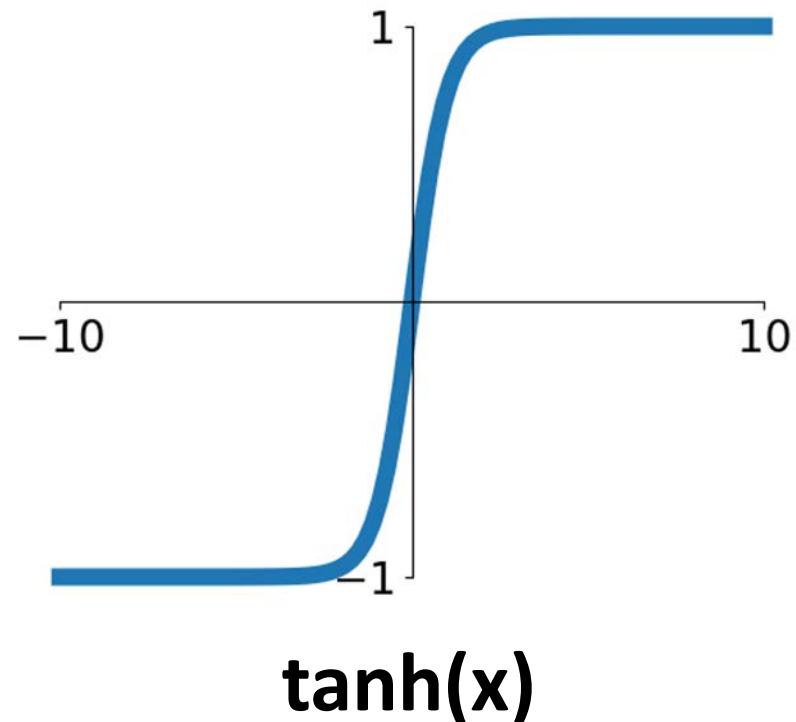
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- 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

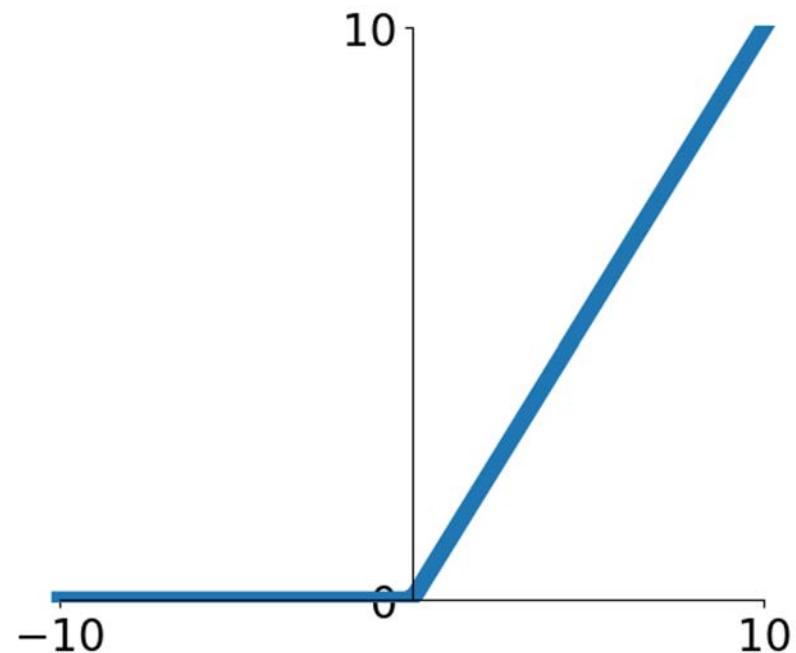
Activation Functions: Tanh



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

Activation Functions: ReLU

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

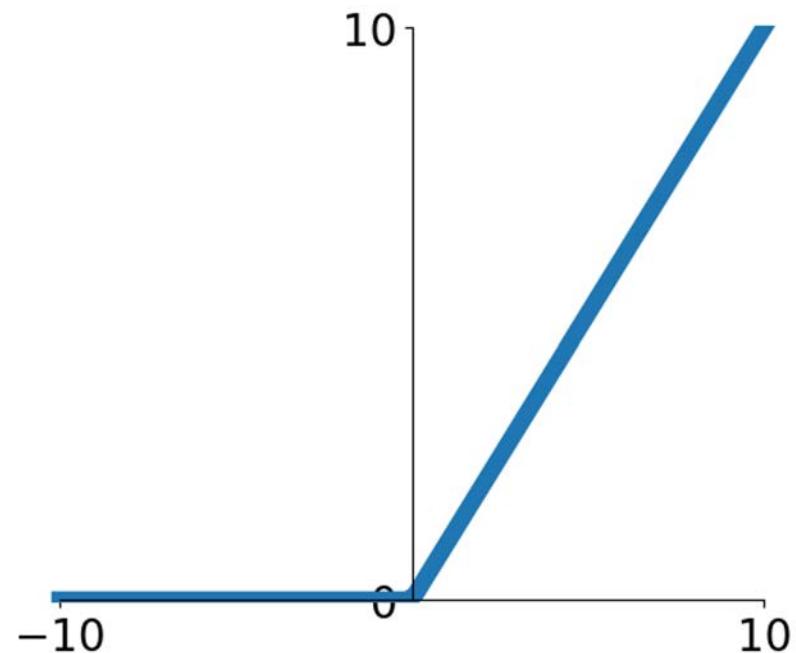


ReLU
(Rectified Linear Unit)

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

Activation Functions: ReLU

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

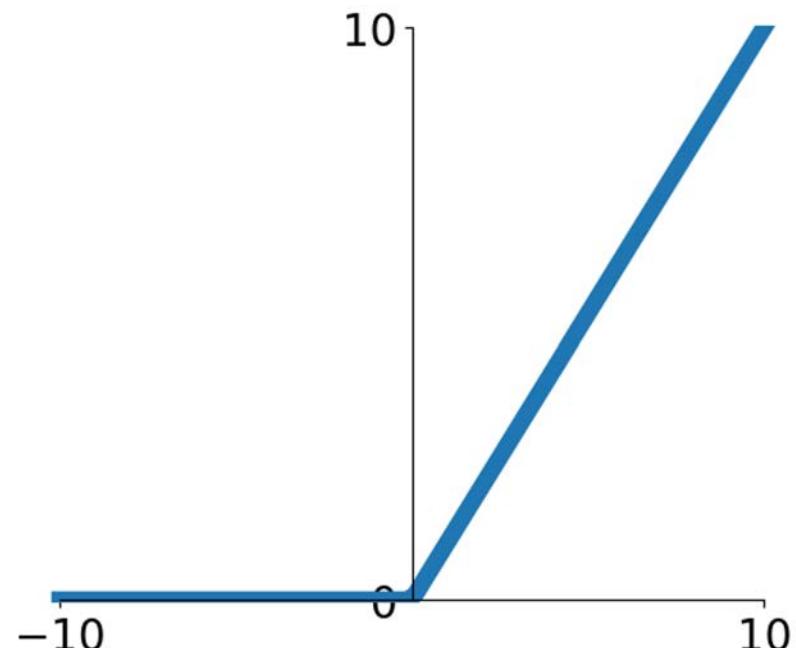


ReLU
(Rectified Linear Unit)

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

Activation Functions: ReLU

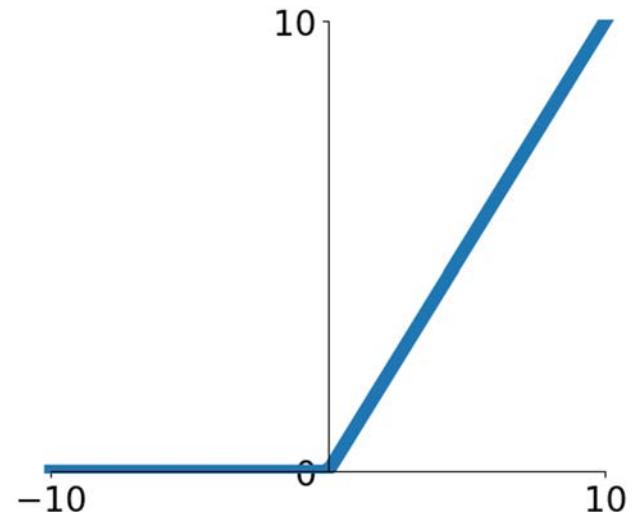
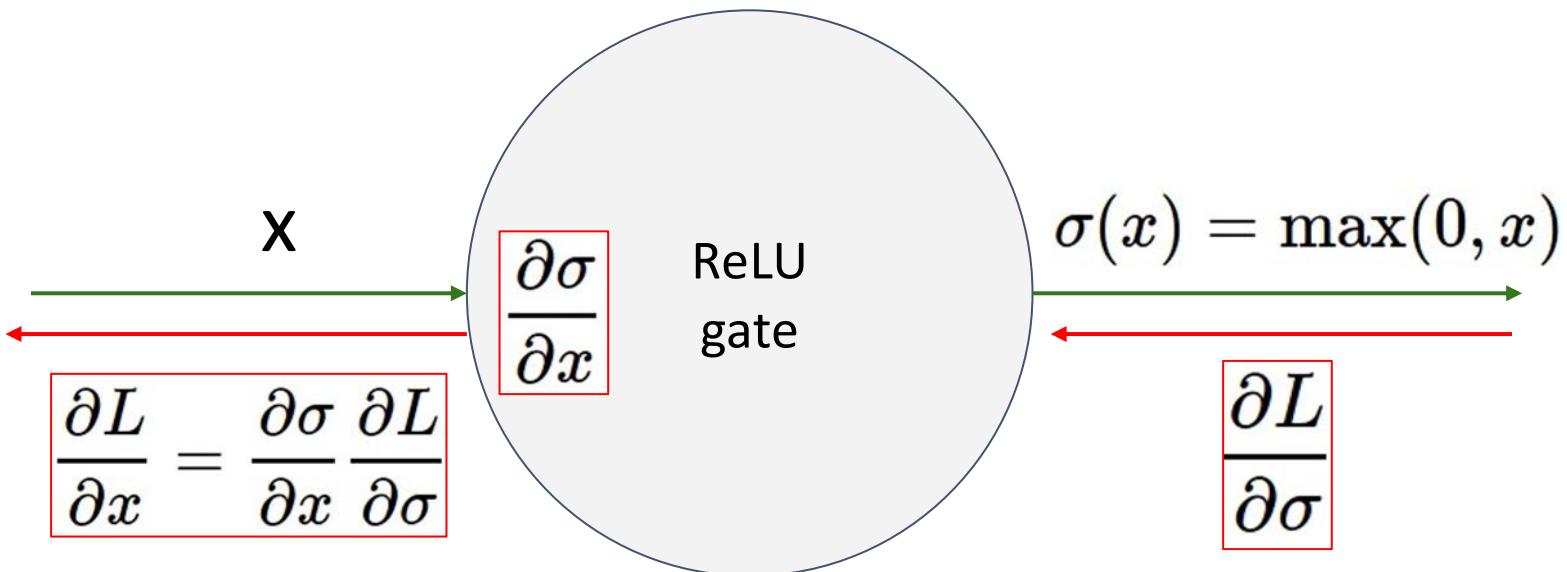
$$f(x) = \max(0, x)$$



ReLU
(Rectified Linear Unit)

- Does not saturate (in +region)
 - Very computationally efficient
 - Converges much faster than sigmoid/tanh in practice (e.g. 6x)
-
- Not zero-centered output
 - An annoyance:
hint: what is the gradient when $x < 0$?

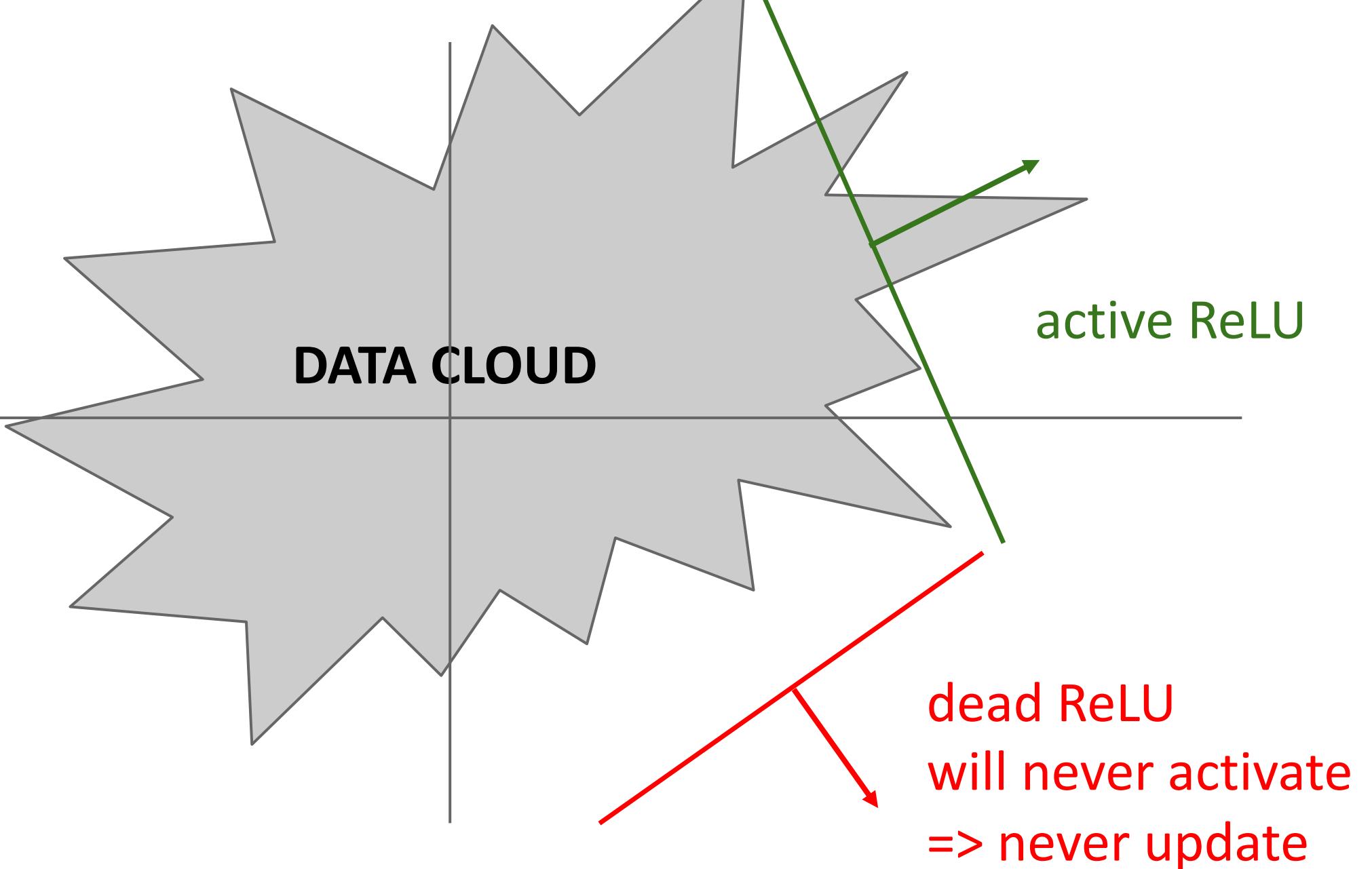
Activation Functions: ReLU

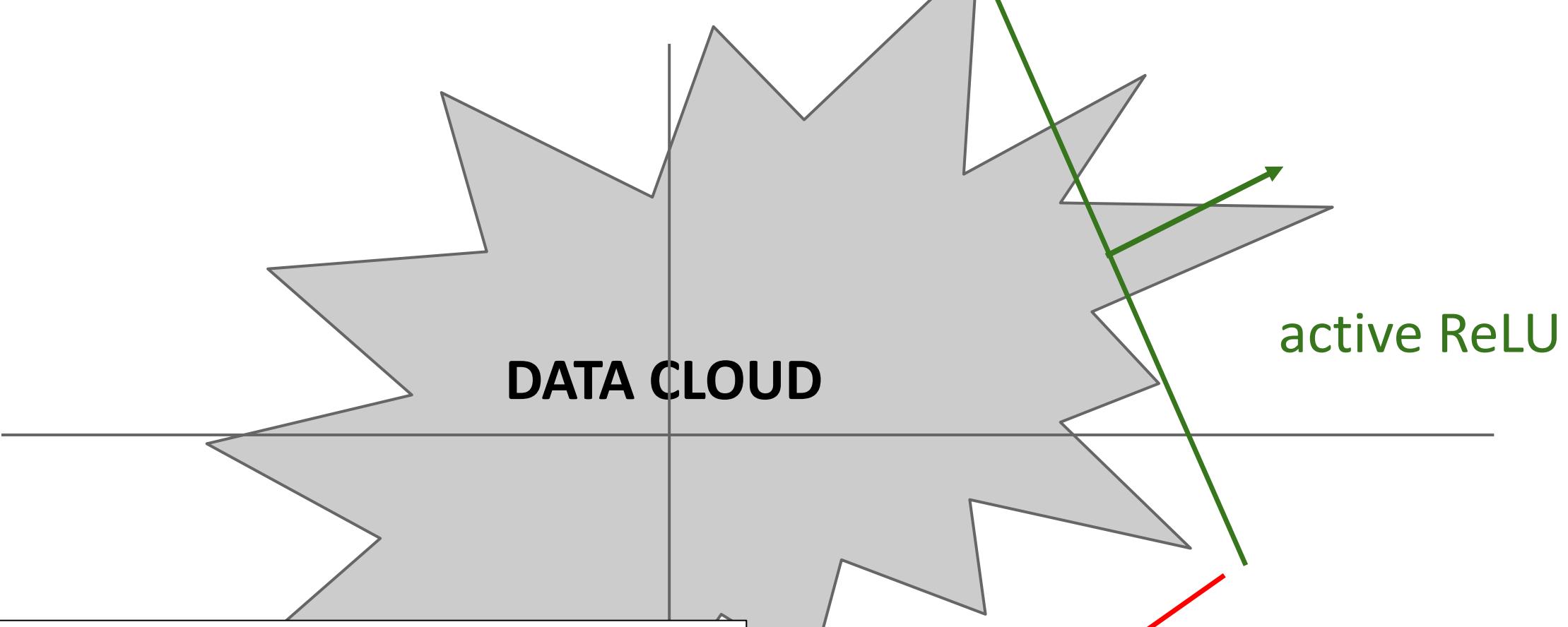


What happens when $x = -10$?

What happens when $x = 0$?

What happens when $x = 10$?

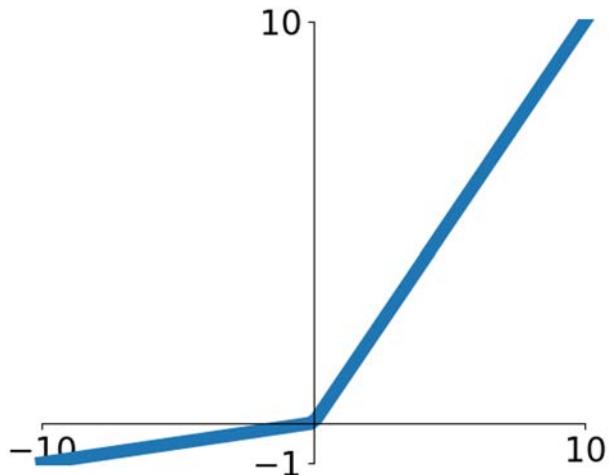




=> Sometimes initialize ReLU neurons with slightly positive biases (e.g. 0.01)

dead ReLU
will never activate
=> never update

Activation Functions: Leaky ReLU



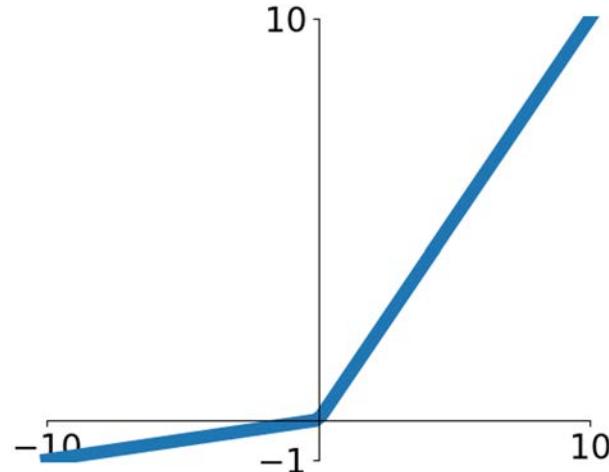
Leaky ReLU

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

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

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

Activation Functions: Leaky ReLU



Leaky ReLU

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

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

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

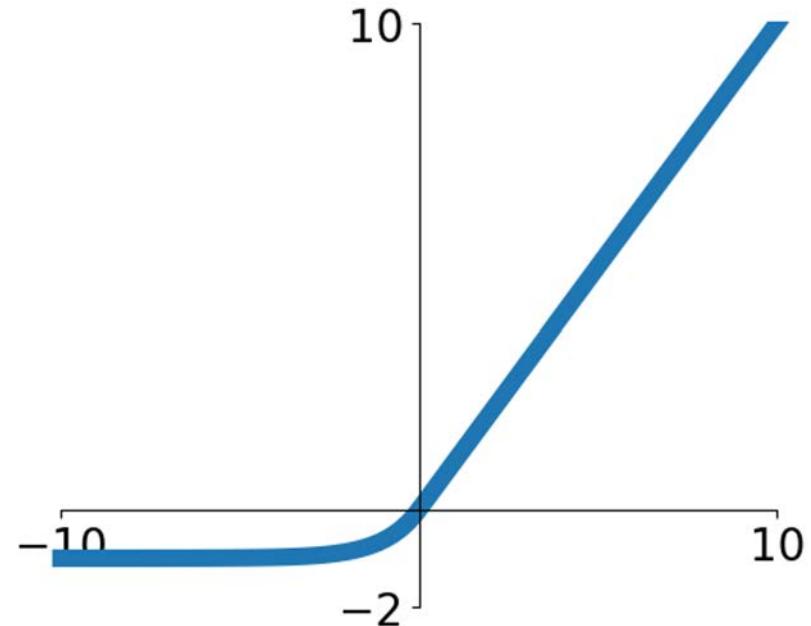
Parametric ReLU (PReLU)

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

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)



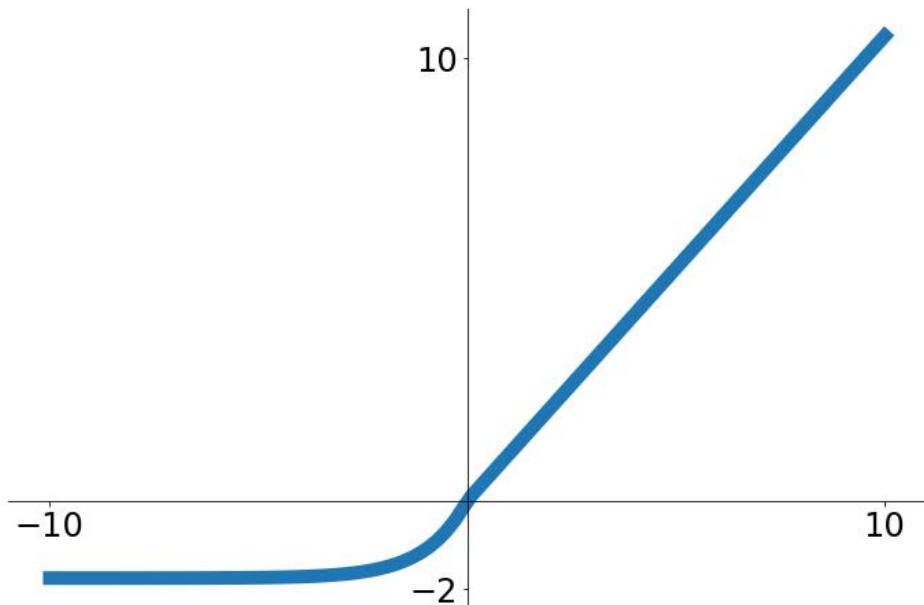
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

(Default alpha=1)

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

- Computation requires `exp()`

Activation Functions: Scaled Exponential Linear Unit (SELU)



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

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

$$\alpha = 1.6732632423543772848170429916717$$

$$\lambda = 1.0507009873554804934193349852946$$

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

Activation Functions: Scaled Exponential Linear Unit (SELU)

- $0 \leq \mu \leq 1$ and $0 \leq \omega \leq 0.1$:
 g is increasing in μ and increasing in ω . We set $\mu = 1$ and $\omega = 0.1$.

$$g(1, 0.1, 3, 1.25, \lambda_{01}, \alpha_{01}) = -0.0180173. \quad (43)$$

Therefore the maximal value of g is -0.0180173 . \square

A3.3 Proof of Theorem 3

First we recall Theorem 3:

Theorem (Increasing ν). We consider $\lambda = \lambda_{01}$, $\alpha = \alpha_{01}$ and the two domains $\Omega_1^- = \{(\mu, \omega, \nu, \tau) \mid -0.1 \leq \mu \leq 0.1, -0.1 \leq \omega \leq 0.1, 0.05 \leq \nu \leq 0.16, 0.8 \leq \tau \leq 1.25\}$ and $\Omega_2^- = \{(\mu, \omega, \nu, \tau) \mid -0.1 \leq \mu \leq 0.1, -0.1 \leq \omega \leq 0.1, 0.05 \leq \nu \leq 0.24, 0.9 \leq \tau \leq 1.25\}$.

The mapping of the variance $\tilde{\nu}(\mu, \omega, \nu, \tau, \lambda, \alpha)$ given in Eq. (5) increases

$$\tilde{\nu}(\mu, \omega, \nu, \tau, \lambda_{01}, \alpha_{01}) > \nu \quad (44)$$

in both Ω_1^- and Ω_2^- . All fixed points (μ, ν) of mapping Eq. (5) and Eq. (4) ensure for $0.8 \leq \tau$ that $\nu > 0.16$ and for $0.9 \leq \tau$ that $\nu > 0.24$. Consequently, the variance mapping Eq. (5) and Eq. (4) ensures a lower bound on the variance ν .

Proof. The mean value theorem states that there exists a $t \in [0, 1]$ for which

$$\tilde{\xi}(\mu, \omega, \nu, \tau, \lambda_{01}, \alpha_{01}) - \tilde{\xi}(\mu, \omega, \nu_{\min}, \tau, \lambda_{01}, \alpha_{01}) = \frac{\partial}{\partial \nu} \tilde{\xi}(\mu, \omega, \nu + t(\nu_{\min} - \nu), \tau, \lambda_{01}, \alpha_{01}) (\nu - \nu_{\min}). \quad (45)$$

Therefore

$$\tilde{\xi}(\mu, \omega, \nu, \tau, \lambda_{01}, \alpha_{01}) = \tilde{\xi}(\mu, \omega, \nu_{\min}, \tau, \lambda_{01}, \alpha_{01}) + \frac{\partial}{\partial \nu} \tilde{\xi}(\mu, \omega, \nu + t(\nu_{\min} - \nu), \tau, \lambda_{01}, \alpha_{01}) (\nu - \nu_{\min}). \quad (46)$$

Therefore we are interested to bound the derivative of the $\tilde{\xi}$ -mapping Eq. (13) with respect to ν :

$$\begin{aligned} \frac{\partial}{\partial \nu} \tilde{\xi}(\mu, \omega, \nu, \tau, \lambda_{01}, \alpha_{01}) &= \quad (47) \\ \frac{1}{2} \lambda^2 \tau e^{-\frac{\mu^2 \omega^2}{\nu^2 \tau^2}} \left(\alpha^2 \left(- \left(e^{\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}}} \operatorname{erfc} \left(\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) \right)^2 - 2e^{\left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right)^2} \operatorname{erfc} \left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) \right) \right) &- \\ \operatorname{erfc} \left(\frac{\mu \omega}{\sqrt{2} \sqrt{\nu \tau}} \right) + 2. \end{aligned}$$

The sub-term Eq. (308) enters the derivative Eq. (47) with a negative sign! According to Lemma 18, the minimal value of sub-term Eq. (308) is obtained by the largest largest ν , by the smallest τ , and the largest $y = \mu \omega = 0.01$. Also the positive term $\operatorname{erfc} \left(\frac{\mu \omega}{\sqrt{2} \sqrt{\nu \tau}} \right) + 2$ is multiplied by τ , which is minimized by using the smallest τ . Therefore we can use the smallest τ in whole formula Eq. (47) to lower bound it.

First we consider the domain $0.05 \leq \nu \leq 0.16$ and $0.8 \leq \tau \leq 1.25$. The factor consisting of the exponential in front of the brackets has its smallest value for $e^{-\frac{\mu^2 \omega^2}{\nu^2 \tau^2}}$. Since ν is monotonically decreasing we inserted the smallest argument via $\operatorname{erfc} \left(\frac{\mu \omega}{\sqrt{2} \sqrt{\nu \tau}} \right) + 2$ in order to obtain the maximal negative contribution. Thus, applying Lemma 18, we obtain the lower bound on the derivative:

$$\frac{1}{2} \lambda^2 \tau e^{-\frac{\mu^2 \omega^2}{\nu^2 \tau^2}} \left(\alpha^2 \left(- \left(e^{\left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right)^2} \operatorname{erfc} \left(\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) - 2e^{\left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right)^2} \operatorname{erfc} \left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) \right) \right) \right) = \quad (48)$$

18

□

$$\begin{aligned} &\frac{6 - 0.8 + 0.01}{2\sqrt{0.16 - 0.8}} - \\ &\operatorname{erfc} \left(\frac{0.01}{\sqrt{2\sqrt{0.05 - 0.8}}} + 2 \right) \rangle 0.969231. \end{aligned}$$

est $\tilde{\nu}(\nu)$. We follow the proof of Lemma 8, al and $x = \nu \tau$ must be minimal. Thus, the $\alpha_{01}, \alpha_{01}) = 0.0662727$ for $0.05 \leq \nu$ and

(Lemma 43) provide

$$\begin{aligned} &(\alpha_{01})^2 > \quad (49) \\ &0.0128115 + 0.969231\nu > \\ &> \nu. \end{aligned}$$

$\leq \tau \leq 1.25$. The factor consisting of the or $e^{-\frac{\mu^2 \omega^2}{\nu^2 \tau^2}}$. Since erfc is monotonic in $\frac{0.01}{\sqrt{2\sqrt{0.05 - 0.9}}}$ in order to obtain the maximal

$$\begin{aligned} &\text{the derivative:} \\ &2e^{\left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right)^2} \operatorname{erfc} \left(\frac{\mu \omega + 2\nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) - \\ &\frac{4 - 0.9 + 0.01}{2\sqrt{0.24 - 0.9}} - \end{aligned}$$

$$\begin{aligned} &\operatorname{erfc} \left(\frac{0.01}{\sqrt{2\sqrt{0.05 - 0.9}}} + 2 \right) \rangle 0.976952. \end{aligned}$$

est $\tilde{\nu}(\nu)$. We follow the proof of Lemma 8, al and $x = \nu \tau$ must be minimal. Thus, the $\alpha_{01}, \alpha_{01}) = 0.0738404$ for $0.05 \leq \nu$ and α on $(\bar{\mu})$ (Lemma 43) gives

$$\begin{aligned} &(\alpha_{01})^2 > \quad (51) \\ &0.0199928 + 0.976952\nu > \\ &> \nu. \end{aligned}$$

□

softs

cobian norm smaller than one

The Jacobian of the mapping g is smaller in true in a larger domain than the original extend to $\tau \in [0.8, 1.25]$. The range of the following domain throughout this section: $[0.8, 1.25]$.

19

In the following, we denote two Jacobians: (1) the Jacobian \mathcal{J} of the mapping $\mathcal{J} : (\mu, \nu) \mapsto (\tilde{\mu}, \tilde{\nu})$ because the and many properties of the system can already be seen on \mathcal{J} .

$$\begin{aligned} &\begin{pmatrix} \frac{\partial}{\partial \mu} \tilde{\mu} \\ \frac{\partial}{\partial \nu} \tilde{\mu} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial \mu} \tilde{\mu} & \frac{\partial}{\partial \nu} \tilde{\mu} \\ \frac{\partial}{\partial \mu} \tilde{\nu} & \frac{\partial}{\partial \nu} \tilde{\nu} \end{pmatrix} \quad (52) \\ &\begin{pmatrix} \frac{\partial}{\partial \mu} \tilde{\nu} \\ \frac{\partial}{\partial \nu} \tilde{\nu} \end{pmatrix} = \begin{pmatrix} \tilde{\mathcal{J}}_{11} & \tilde{\mathcal{J}}_{12} \\ \tilde{\mathcal{J}}_{21} & \tilde{\mathcal{J}}_{22} - 2\tilde{\mu} \tilde{\mathcal{J}}_{11} \end{pmatrix} \quad (53) \end{aligned}$$

of the Jacobian \mathcal{J} is:

$$\begin{aligned} &\frac{\partial}{\partial \mu} \tilde{\mu}(\mu, \omega, \nu, \tau, \lambda, \alpha) = \quad (54) \\ &\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} - \operatorname{erfc} \left(\frac{\mu \omega}{\sqrt{2} \sqrt{\nu \tau}} \right) + 2 \end{aligned}$$

$$\begin{aligned} &\frac{\partial}{\partial \nu} \tilde{\mu}(\mu, \omega, \nu, \tau, \lambda, \alpha) = \quad (55) \\ &\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} - (\alpha - 1) \sqrt{\frac{2}{\pi \nu \tau}} e^{-\frac{\mu^2 \omega^2}{2\nu \tau}} \end{aligned}$$

$$\begin{aligned} &\frac{\partial}{\partial \mu} \tilde{\nu}(\mu, \omega, \nu, \tau, \lambda, \alpha) = \quad (56) \\ &\operatorname{erfc} \left(\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) + \end{aligned}$$

$$\begin{aligned} &\frac{\mu \omega + 2\nu \tau}{2\sqrt{\nu \tau}} + \mu \omega \left(2 - \operatorname{erfc} \left(\frac{\mu \omega}{2\sqrt{2}\sqrt{\nu \tau}} \right) \right) + \sqrt{\frac{2}{\pi}} \sqrt{\nu \tau} e^{-\frac{\mu^2 \omega^2}{2\nu \tau}} \end{aligned}$$

$$\begin{aligned} &\frac{\partial}{\partial \nu} \tilde{\nu}(\mu, \omega, \nu, \tau, \lambda, \alpha) = \quad (57) \\ &\operatorname{erfc} \left(\frac{\mu \omega + \nu \tau}{\sqrt{2} \sqrt{\nu \tau}} \right) + \end{aligned}$$

$$\begin{aligned} &\frac{\omega + 2\nu \tau}{2\sqrt{\nu \tau}} - \operatorname{erfc} \left(\frac{\mu \omega}{\sqrt{2} \sqrt{\nu \tau}} \right) + 2 \end{aligned}$$

largest singular value of the Jacobian. If the largest singular value is 1, then the spectral norm of the Jacobian is smaller than 1. Then the of the mean and variance to the mean and variance in the next layer is

lar value is smaller than 1 by evaluating the function $S(\mu, \omega, \nu, \tau, \lambda, \alpha)$ can Value Theorem to bound the deviation of the function S between two adjacent grid points of ν with respect to (μ, ω, ν, τ) . If all hanes the deltas (differences between grid points and evaluated points) have proofed that the function is below 1.

2 matrix

$$\begin{aligned} &\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad (58) \\ &a_{22}^2 + (a_{21} - a_{12})^2 + \sqrt{(a_{11} - a_{22})^2 + (a_{12} + a_{21})^2} \quad (59) \\ &a_{22}^2 + (a_{21} - a_{12})^2 - \sqrt{(a_{11} - a_{22})^2 + (a_{12} + a_{21})^2}. \quad (60) \end{aligned}$$

20

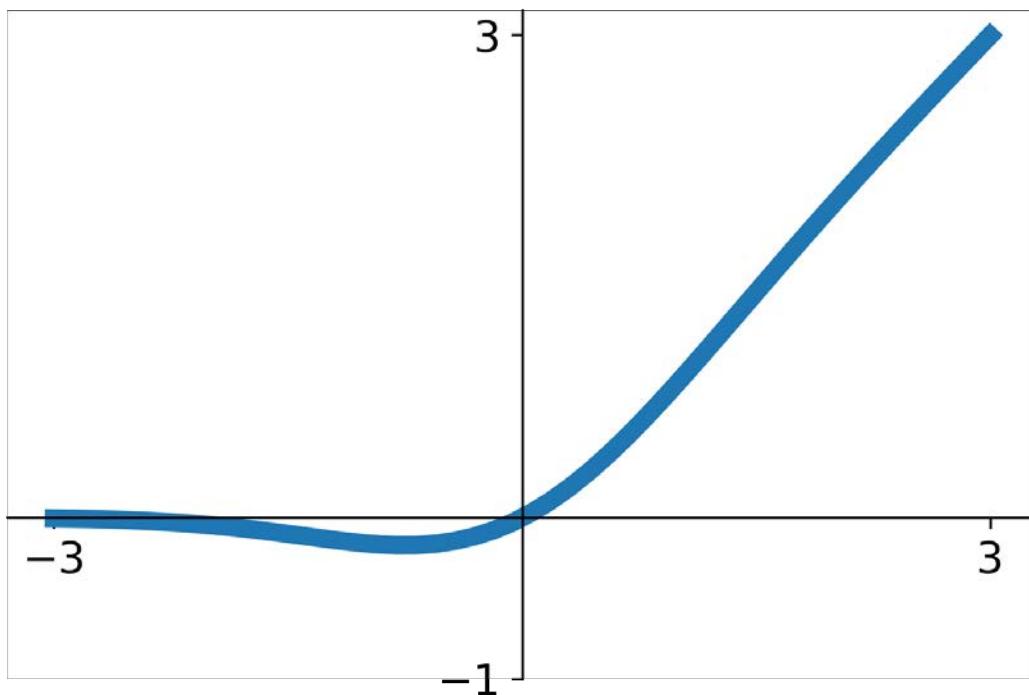
Scaled version of ELU that works better for deep networks
“Self-Normalizing” property;
can train deep SELU networks without BatchNorm

**Derivation takes
91 pages of math
in appendix...**

$$\begin{aligned} \alpha &= 1.673263242354377284170429916717 \\ \lambda &= 1.0507009873554804934193349852946 \end{aligned}$$

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

Activation Functions: Gaussian Error Linear Unit (GELU)



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

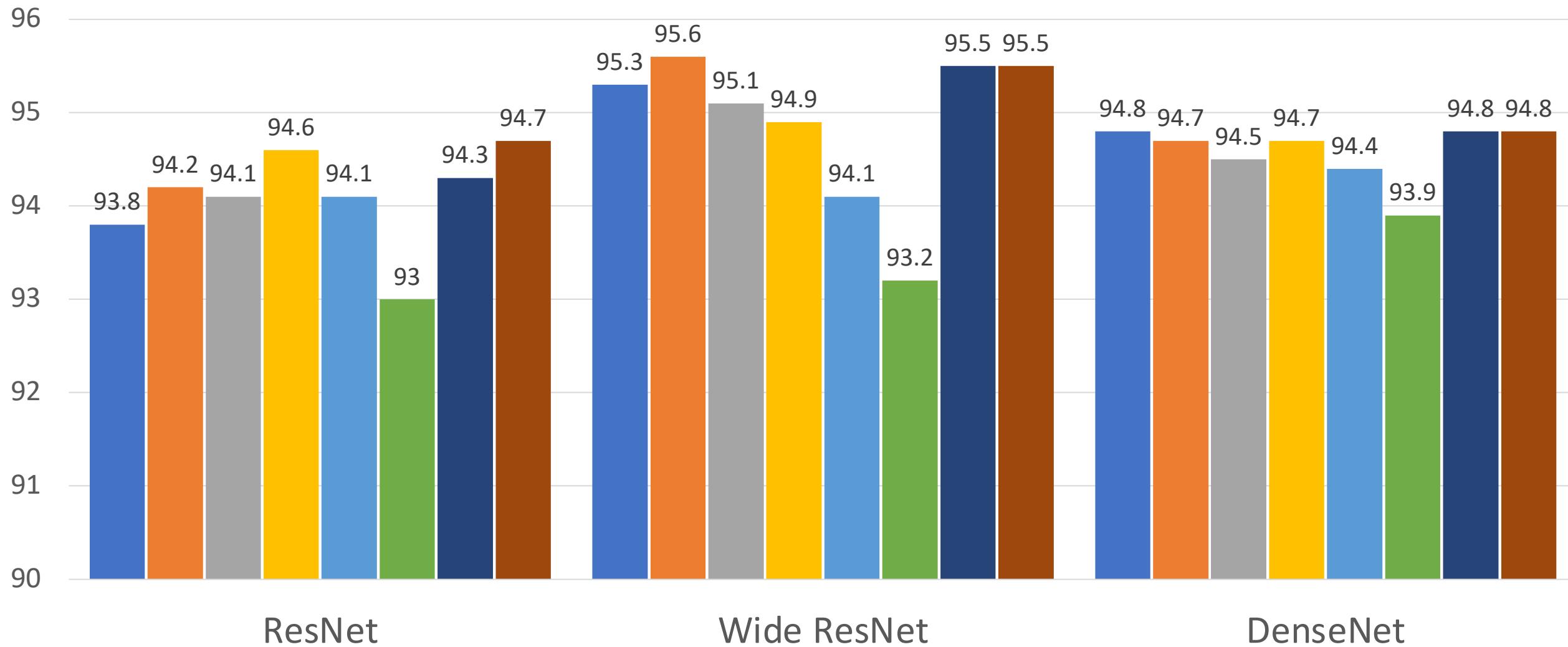
$$\begin{aligned} gelu(x) &= xP(X \leq x) = \frac{x}{2}(1 + \text{erf}(x/\sqrt{2})) \\ &\approx x\sigma(1.702x) \end{aligned}$$

- Idea: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, GPT-2, GPT-3)

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

Accuracy on CIFAR10

ReLU Leaky ReLU Parametric ReLU Softplus ELU SELU GELU Swish



ResNet

Wide ResNet

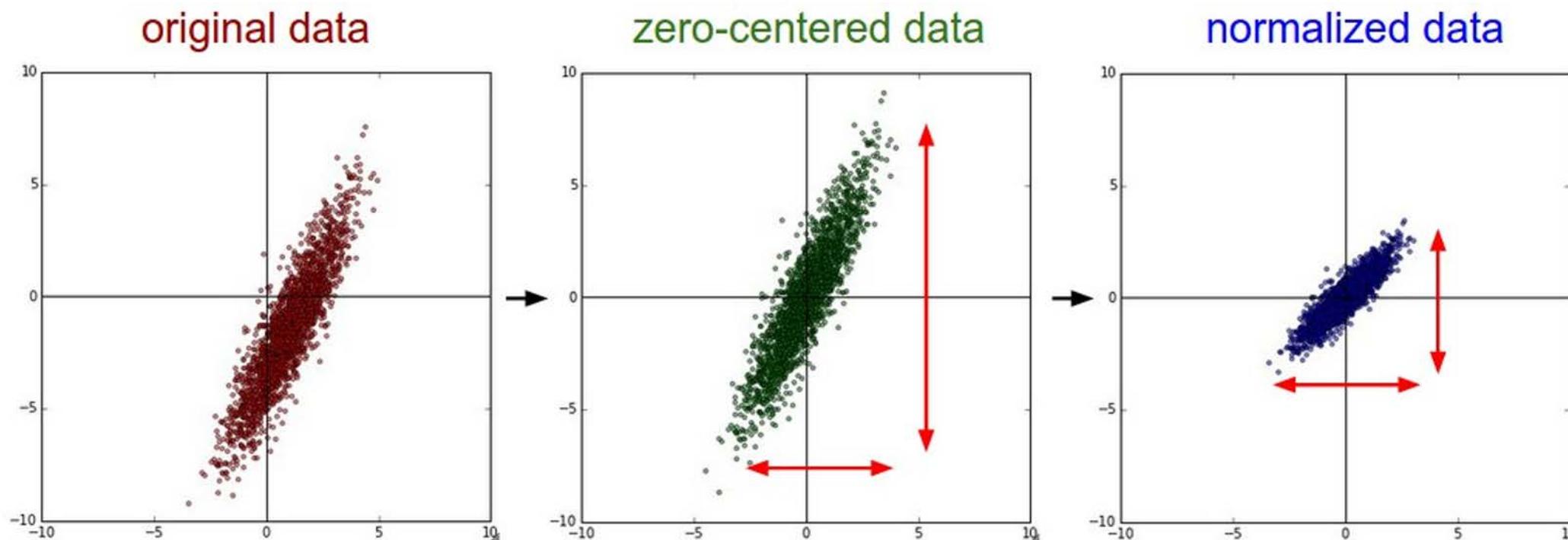
DenseNet

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

Data Preprocessing

Data Preprocessing



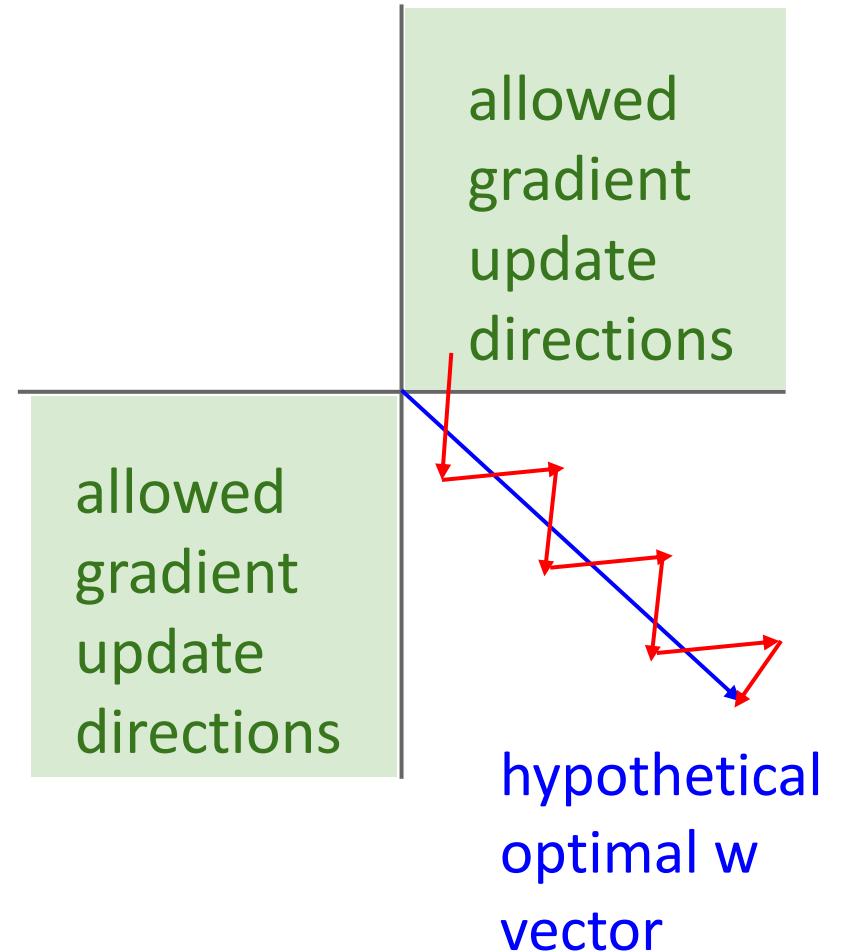
```
X -= np.mean(X, axis = 0)
```

```
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...

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

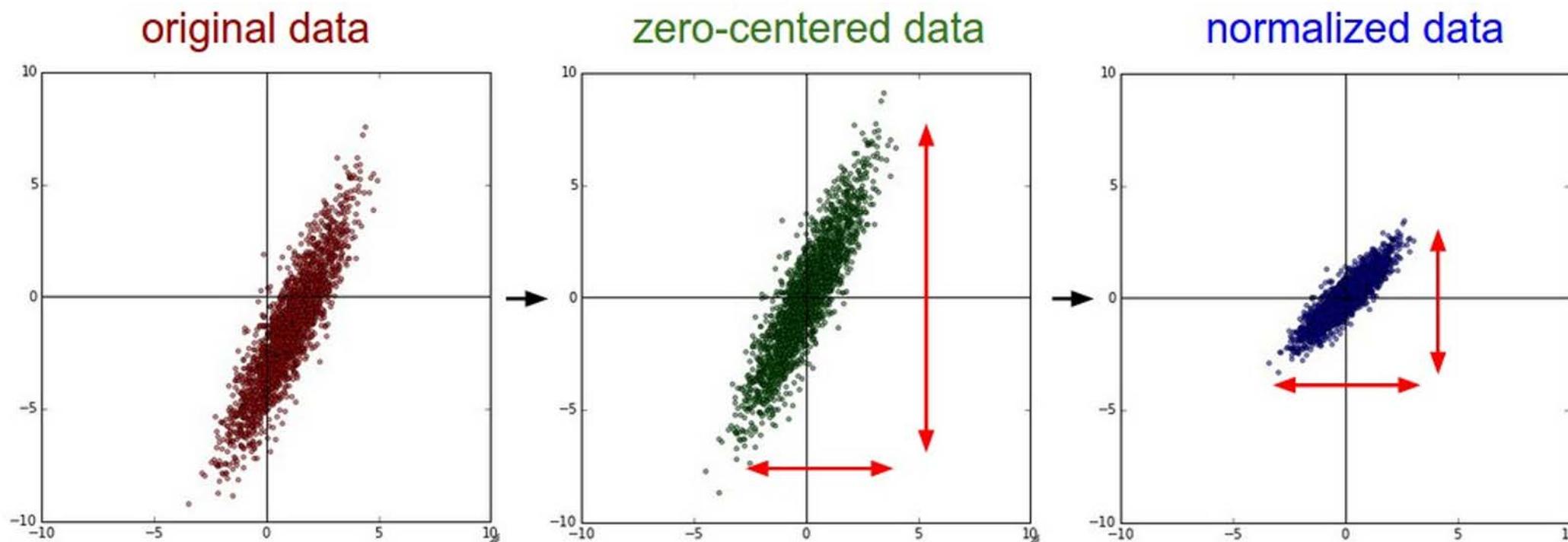


What can we say about the gradients on w ?

Always all positive or all negative :(

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

Data Preprocessing



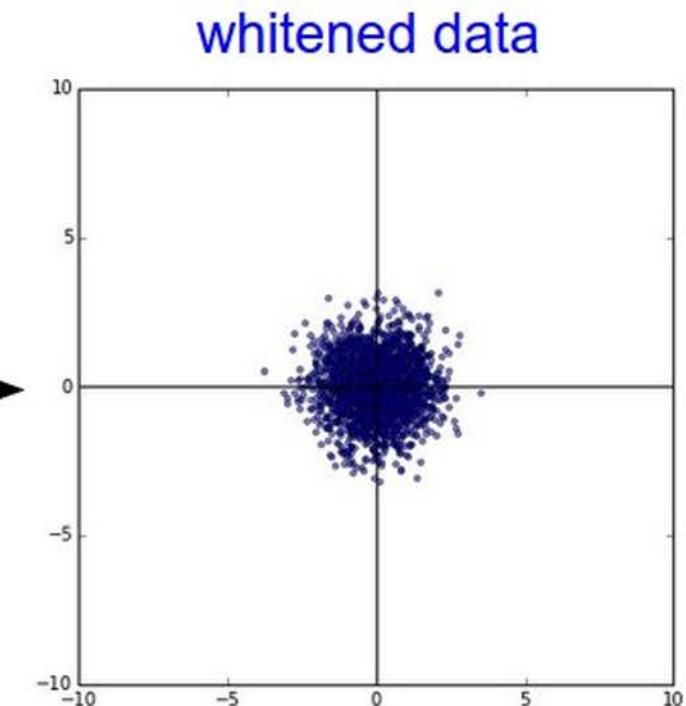
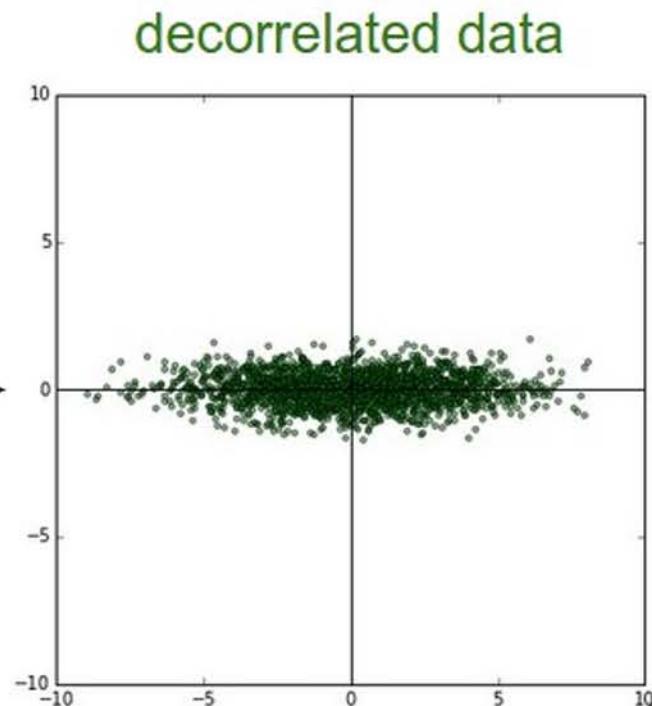
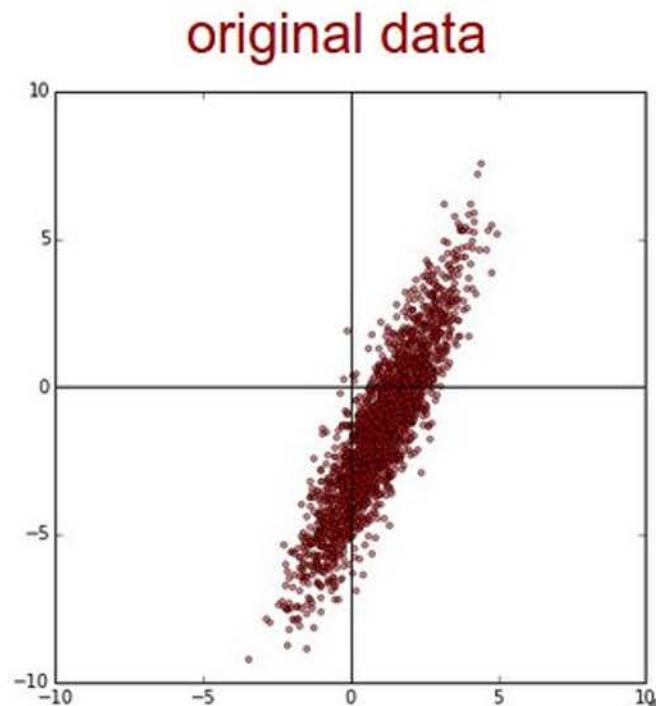
```
X -= np.mean(X, axis = 0)
```

```
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

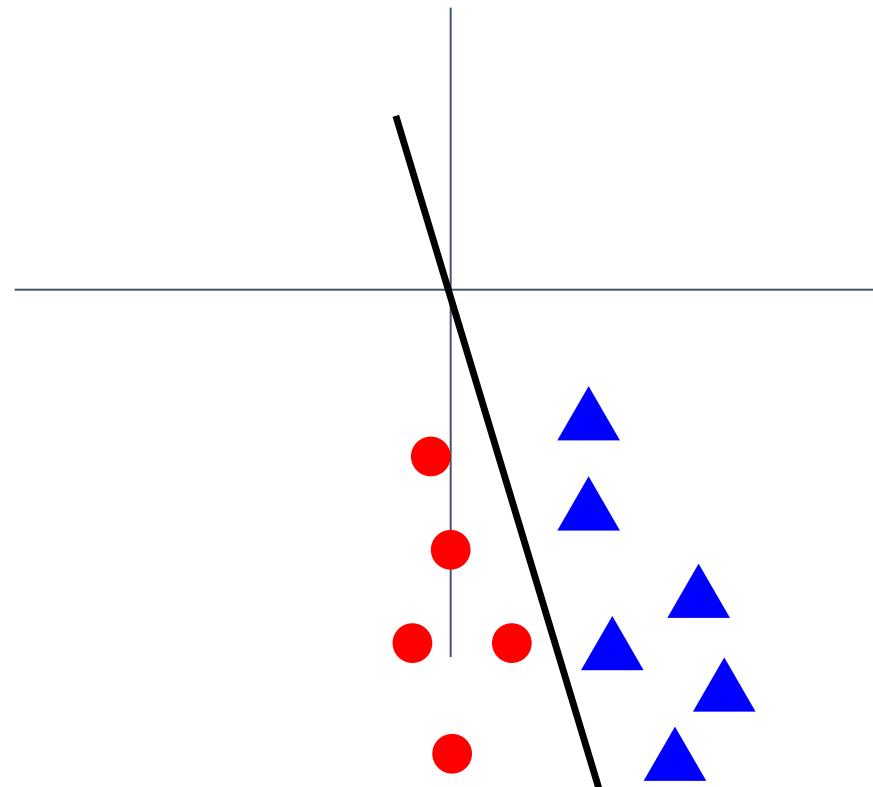


(data has diagonal
covariance matrix)

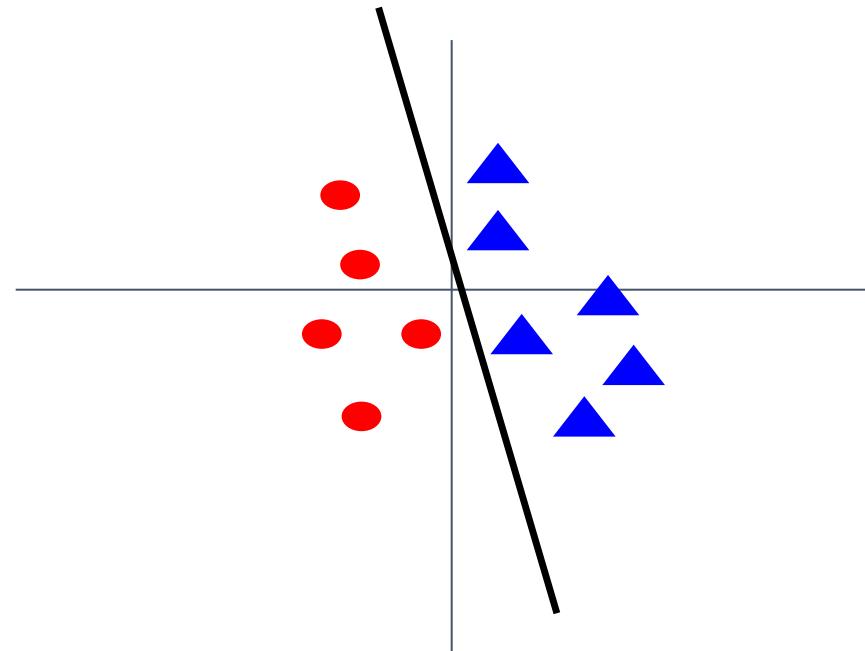
(covariance matrix is
the identity matrix)

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

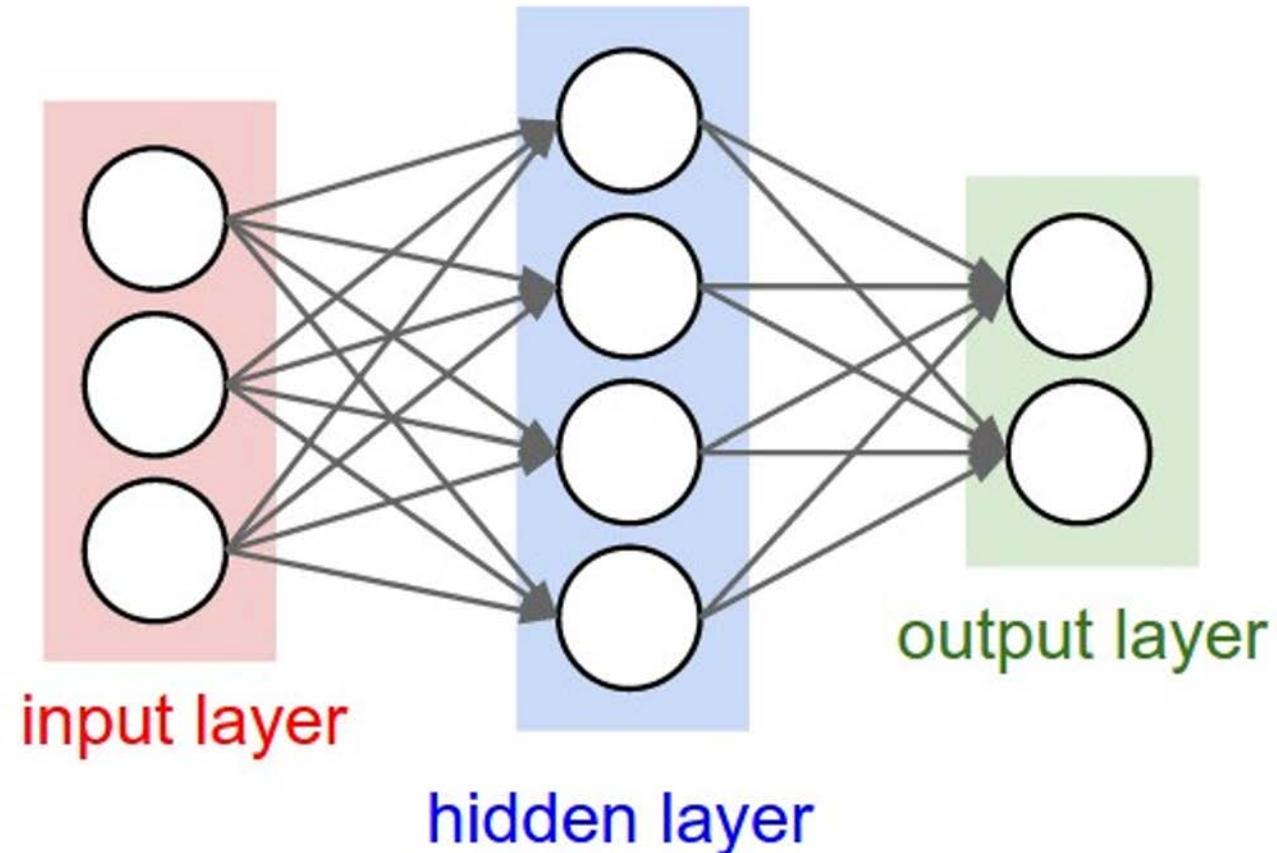
e.g. consider CIFAR-10 example with [32,32,3] images

- 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

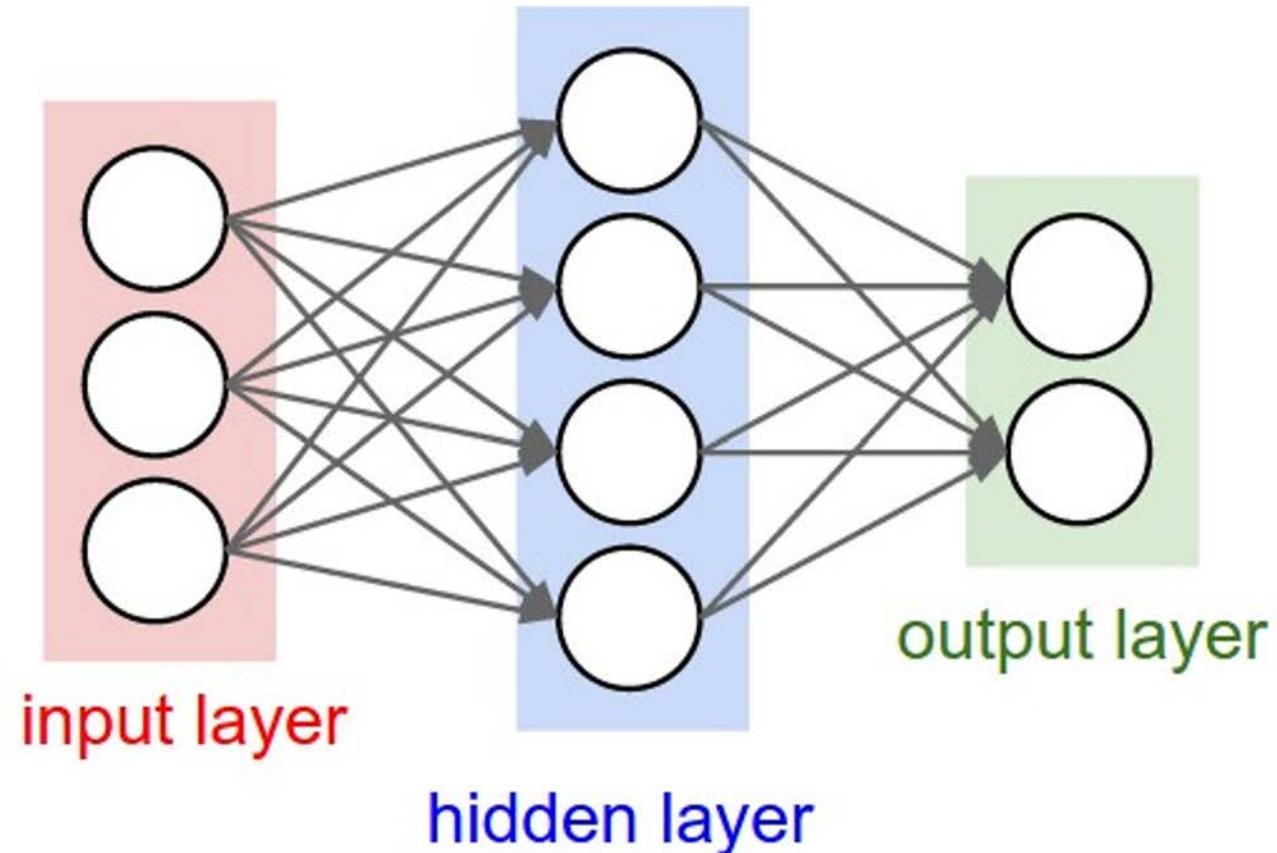
Weight Initialization

Weight Initialization



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

Weight Initialization



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

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

Weight Initialization

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

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

Weight Initialization

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

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

Works ~okay for small networks, but
problems with deeper networks.

Weight Initialization: Activation Statistics

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []                  net with hidden size 4096
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)
```

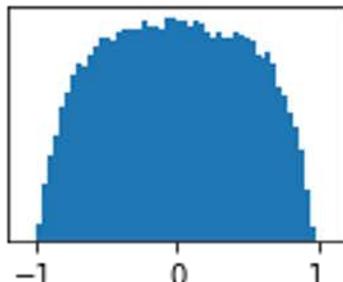
Weight Initialization: Activation Statistics

```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
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)
```

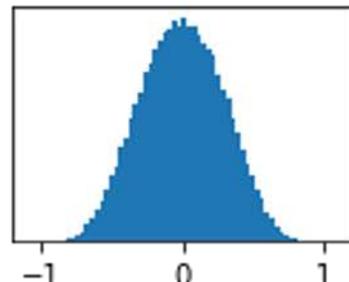
All activations tend to zero for deeper network layers

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

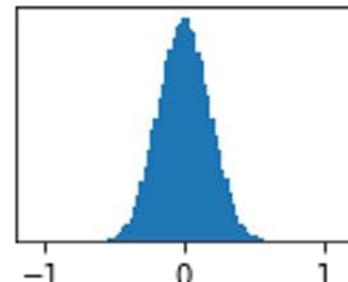
Layer 1
mean=-0.00
std=0.49



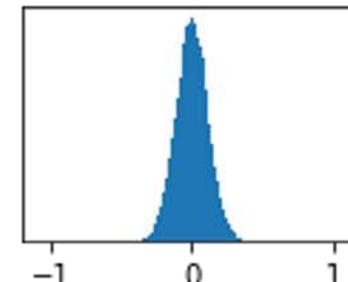
Layer 2
mean=0.00
std=0.29



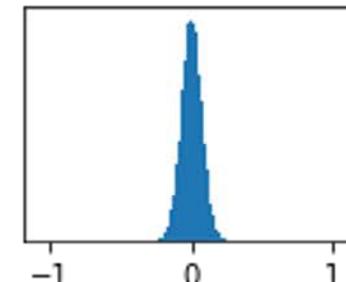
Layer 3
mean=0.00
std=0.18



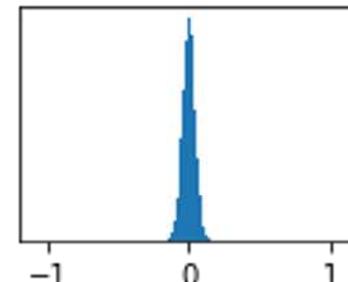
Layer 4
mean=-0.00
std=0.11



Layer 5
mean=-0.00
std=0.07



Layer 6
mean=0.00
std=0.05



Weight Initialization: Activation Statistics

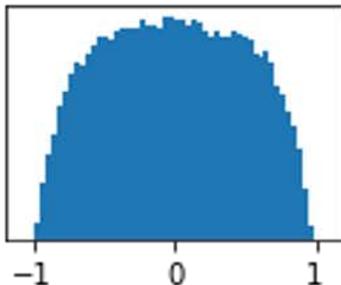
```
dims = [4096] * 7      Forward pass for a 6-layer  
hs = []                  net with hidden size 4096  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = 0.01 * np.random.randn(Din, Dout)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

All activations tend to zero for deeper network layers

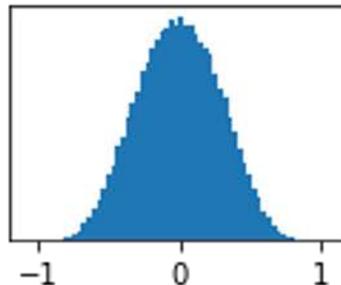
Q: What do the gradients dL/dW look like?

A: All zero, no learning =(

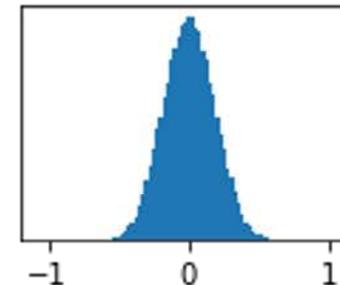
Layer 1
mean=-0.00
std=0.49



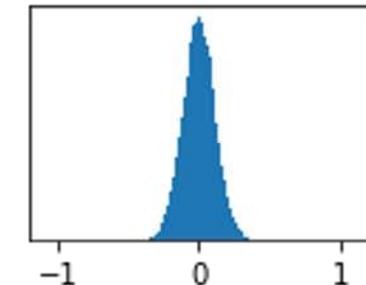
Layer 2
mean=0.00
std=0.29



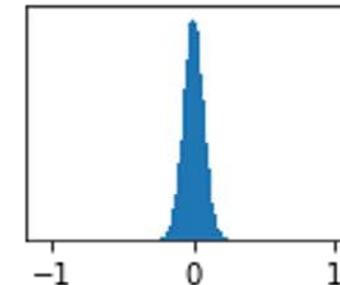
Layer 3
mean=0.00
std=0.18



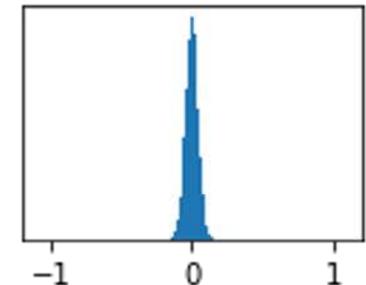
Layer 4
mean=-0.00
std=0.11



Layer 5
mean=-0.00
std=0.07



Layer 6
mean=0.00
std=0.05



Weight Initialization: Activation Statistics

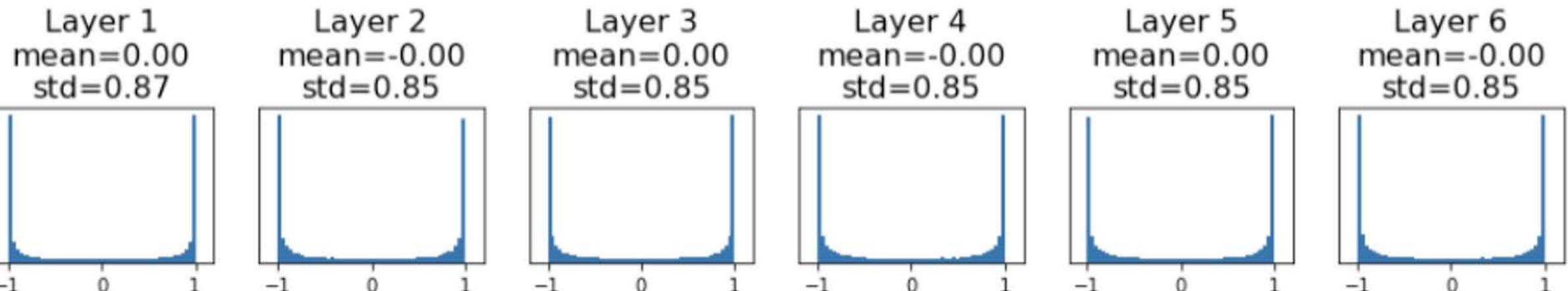
```
dims = [4096] * 7    Increase std of initial weights
hs = []              from 0.01 to 0.05
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Weight Initialization: Activation Statistics

```
dims = [4096] * 7      Increase std of initial weights  
hs = []                  from 0.01 to 0.05  
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)
```

All activations saturate

Q: What do the gradients look like?



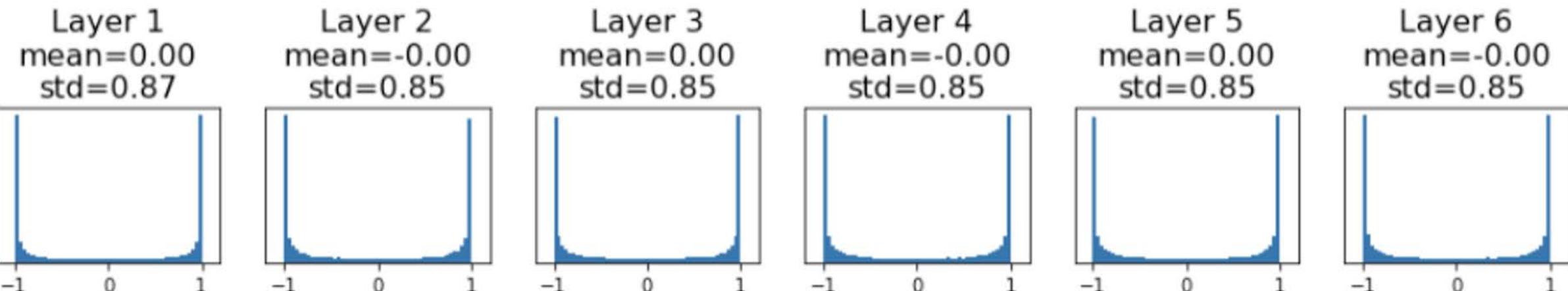
Weight Initialization: Activation Statistics

```
dims = [4096] * 7      Increase std of initial weights  
hs = []                  from 0.01 to 0.05  
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)
```

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Weight Initialization: Xavier Initialization

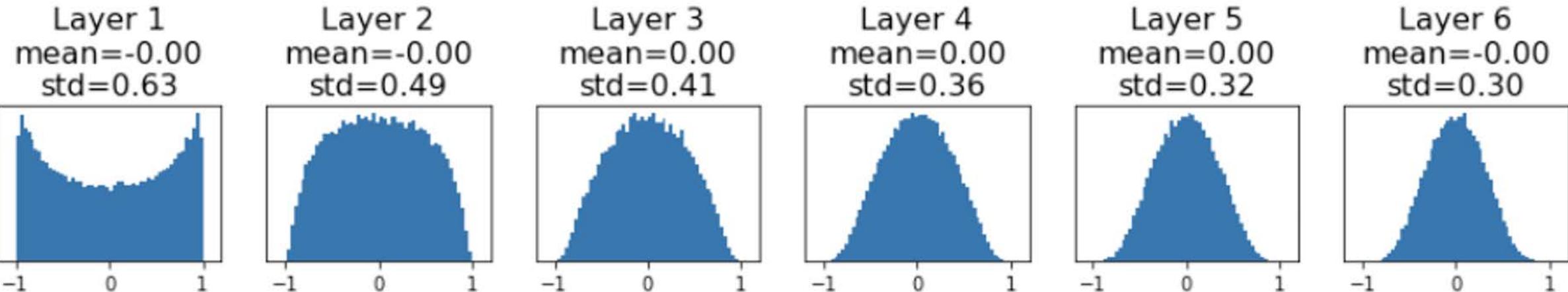
```
dims = [4096] * 7           "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
x = np.random.randn(16, dims[0])  
for Din, Dout in zip(dims[:-1], dims[1:]):  
    W = np.random.randn(Din, Dout) / np.sqrt(Din)  
    x = np.tanh(x.dot(W))  
    hs.append(x)
```

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

Weight Initialization: Xavier Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
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)
```

“Just right”: Activations are nicely scaled for all layers!



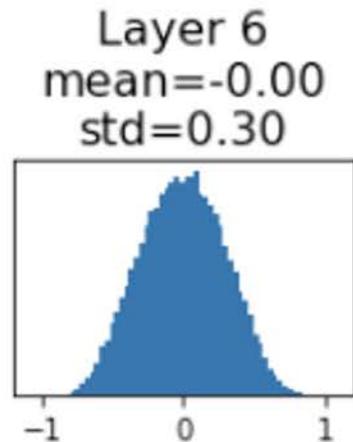
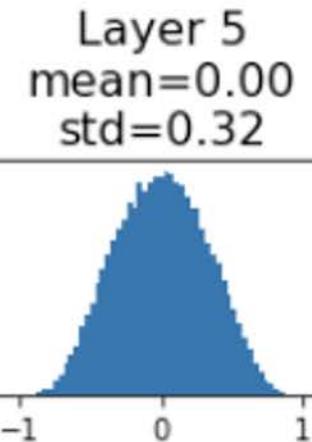
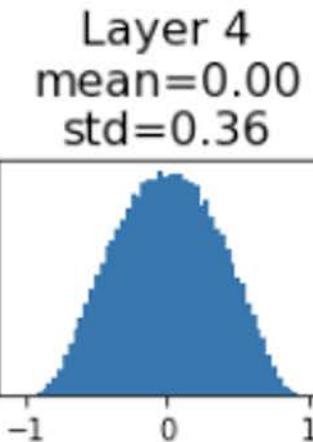
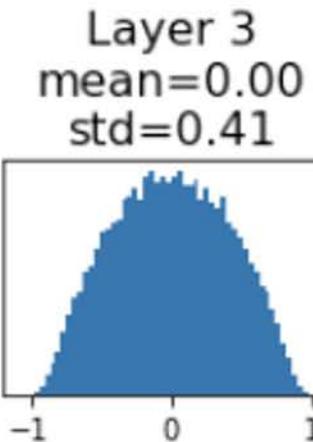
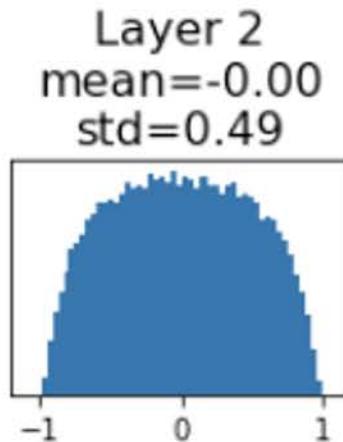
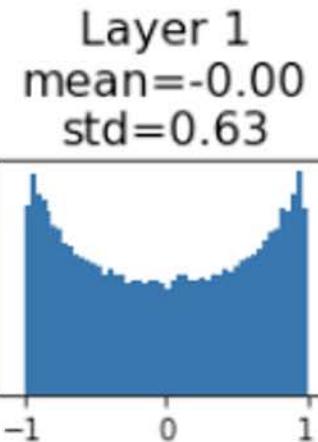
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

Weight Initialization: Xavier Initialization

```
dims = [4096] * 7          "Xavier" initialization:  
hs = []                      std = 1/sqrt(Din)  
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)
```

“Just right”: Activations are nicely scaled for all layers!

For conv layers, Din is $\text{kernel_size}^2 * \text{input_channels}$



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

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

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

Weight Initialization: Xavier Initialization

“Xavier” initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$$\mathbf{y} = \mathbf{Wx}$$

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

$$\text{Var}(y_i) = \text{Din} * \text{Var}(x_i w_i)$$

[Assume x, w are iid]

Weight Initialization: Xavier Initialization

“Xavier” initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$$\mathbf{y} = \mathbf{Wx}$$

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

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

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

Weight Initialization: Xavier Initialization

“Xavier” initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$$\mathbf{y} = \mathbf{Wx}$$

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

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

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

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

Weight Initialization: Xavier Initialization

“Xavier” initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$$\mathbf{y} = \mathbf{Wx}$$

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

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

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

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

If $\text{Var}(w_i) = 1/\text{Din}$ then $\text{Var}(y_i) = \text{Var}(x_i)$

Weight Initialization: What about ReLU?

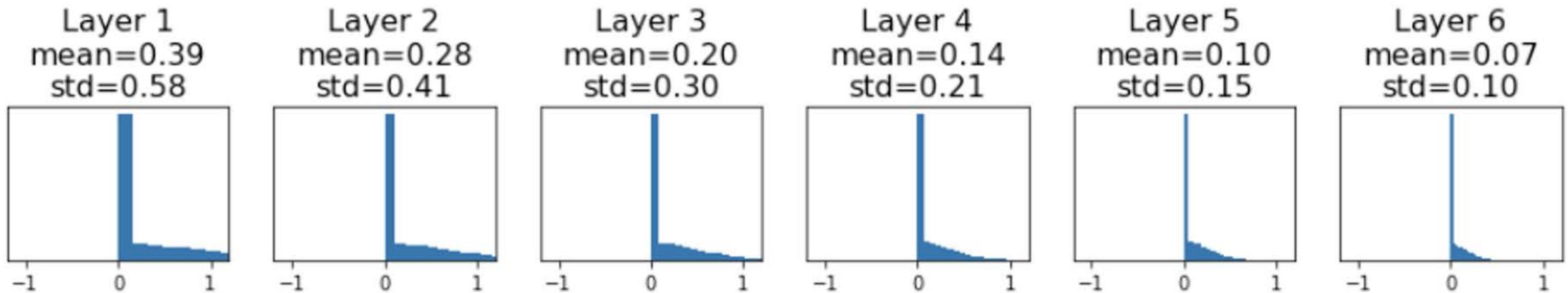
```
dims = [4096] * 7      Change from tanh to ReLU
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
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)
```

Xavier assumes zero centered activation function

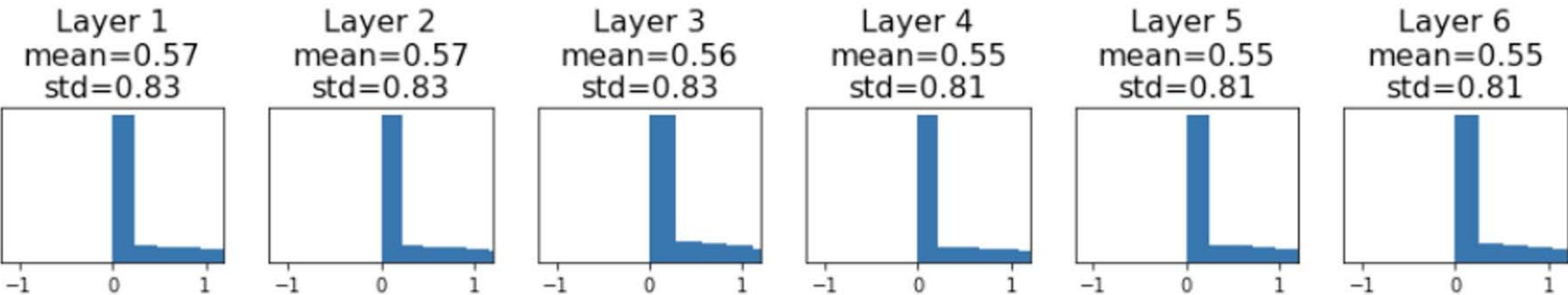
Activations collapse to zero again, no learning =(



Weight Initialization: Kaiming / MSRA Initialization

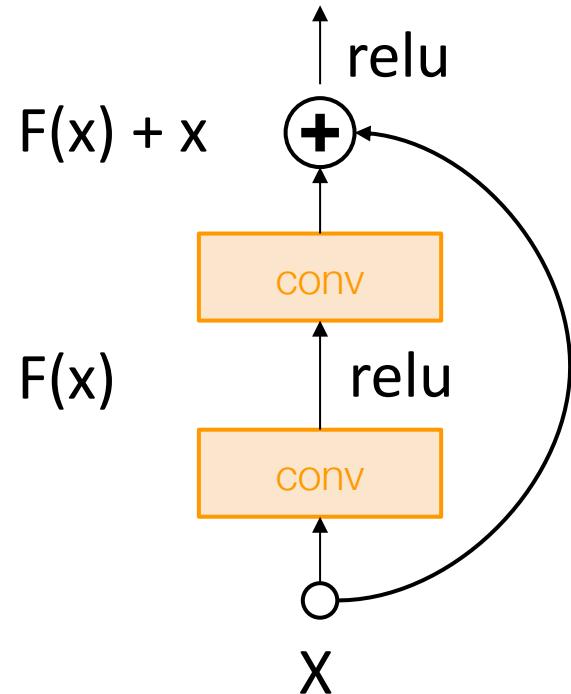
```
dims = [4096] * 7 # ReLU correction: std = sqrt(2 / Din)
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

"Just right" – activations nicely scaled for all layers



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

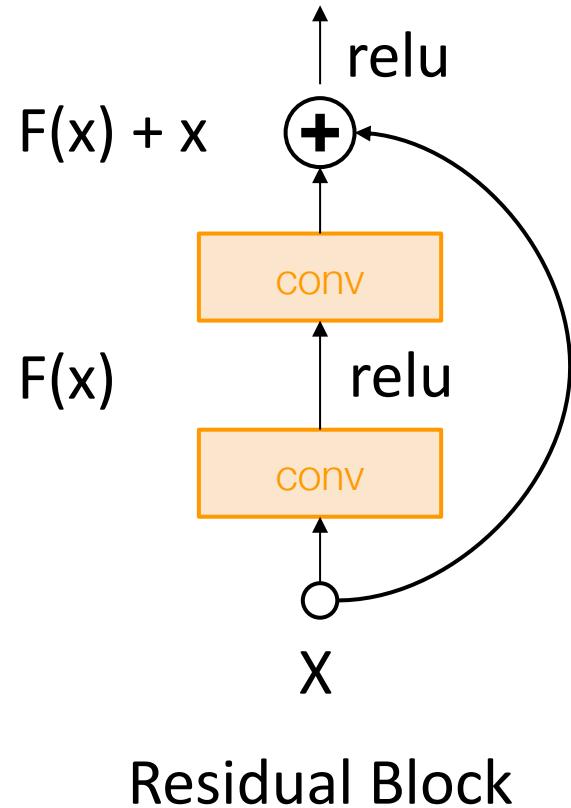
Weight Initialization: Residual Networks



Residual Block

If we initialize with MSRA: then $\text{Var}(F(x)) = \text{Var}(x)$
But then $\text{Var}(F(x) + x) > \text{Var}(x)$ – variance grows
with each block!

Weight Initialization: Residual Networks



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

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

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

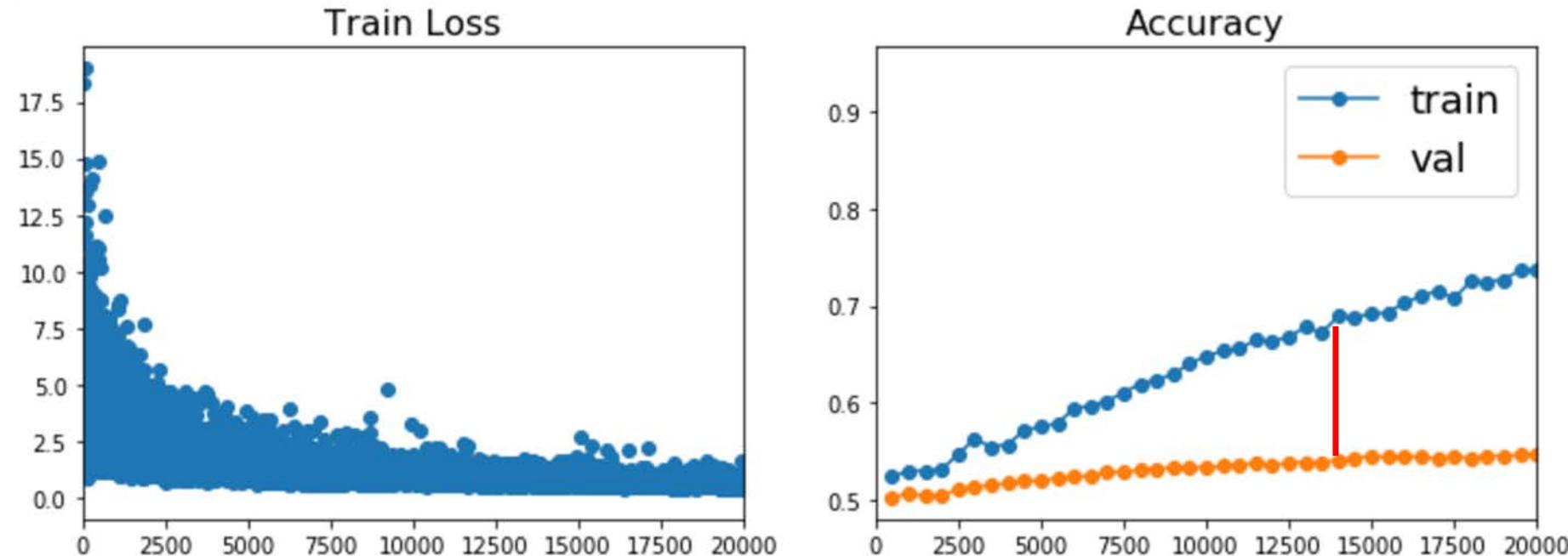
Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

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

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

In common use:

L2 regularization

L1 regularization

Elastic net (L1 + L2)

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

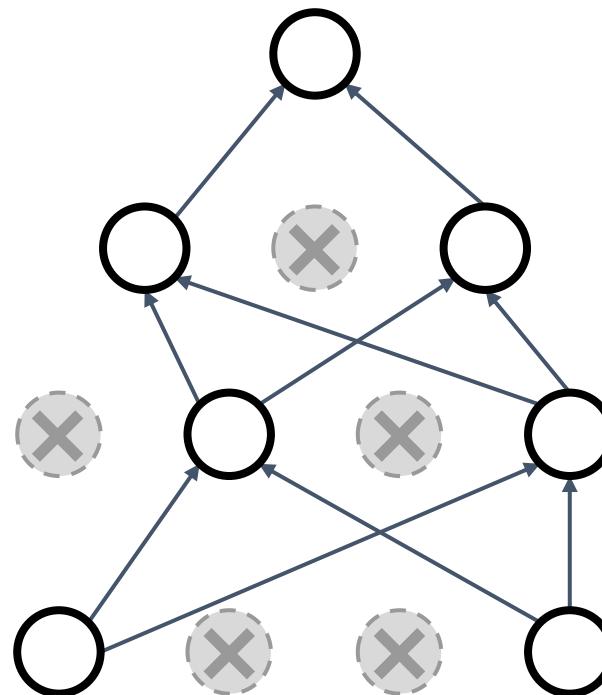
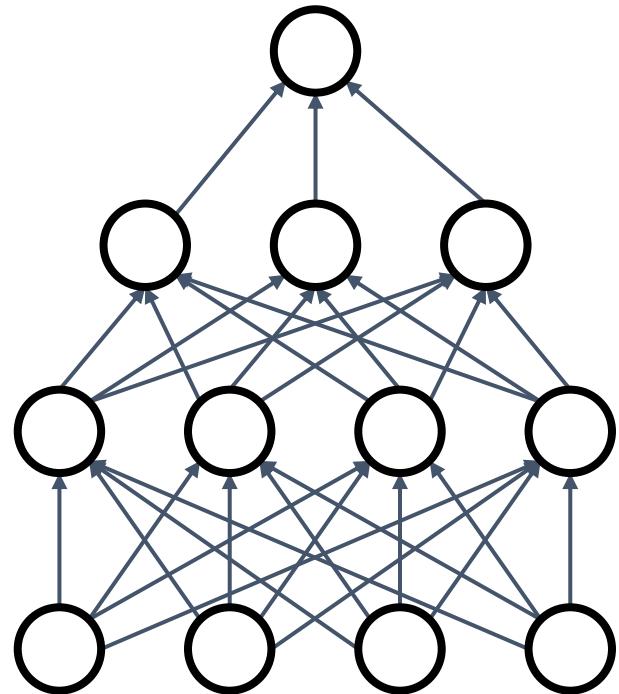
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

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

Regularization: Dropout

In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

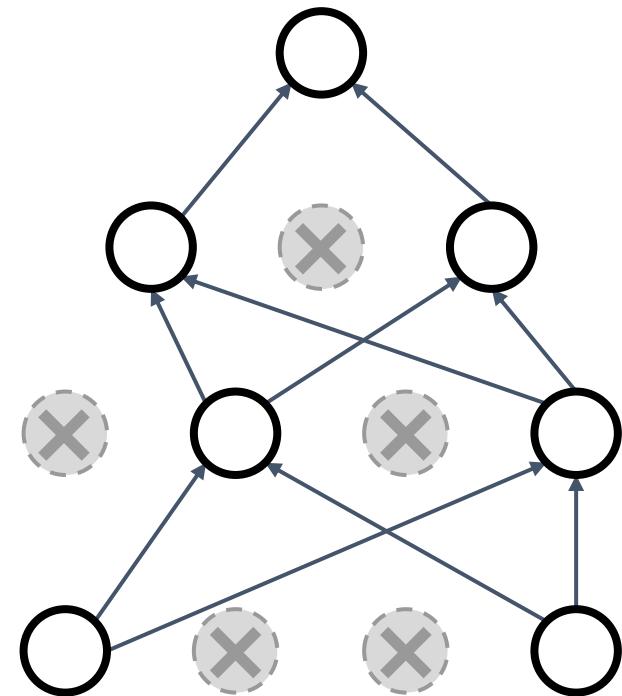
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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

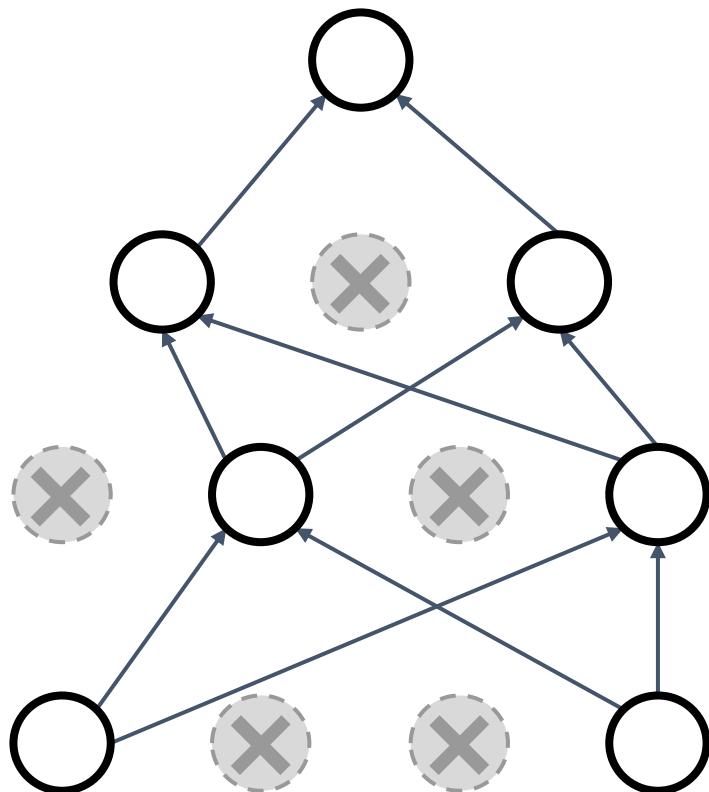
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
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

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

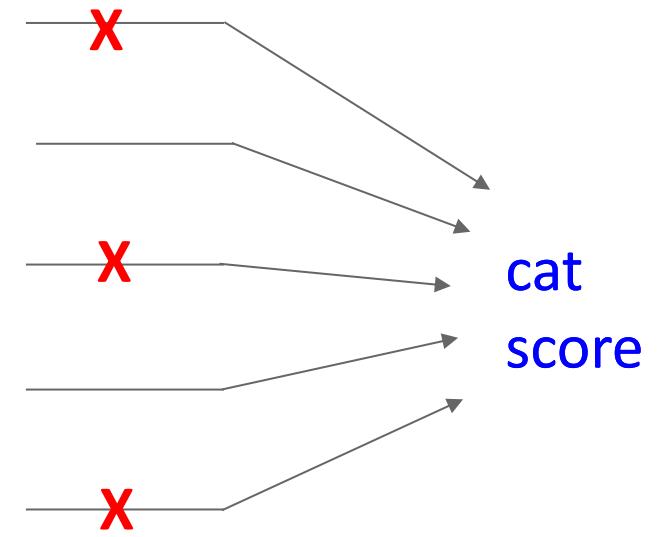
Example forward pass with a 3-layer network using dropout



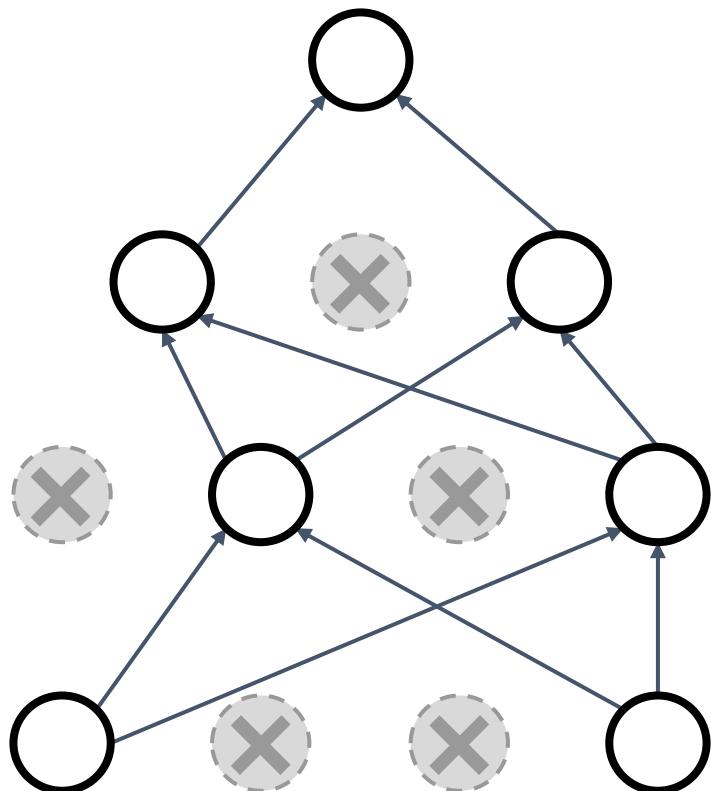
Regularization: Dropout



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!

Output
(label) Input
(image)

$$\mathbf{y} = f_W(\mathbf{x}, \mathbf{z})$$

Random
mask

Want to “average out” the randomness at test-time

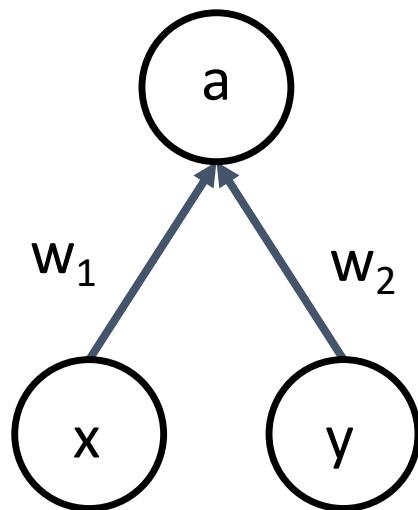
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

But this integral seems hard ...

Dropout: Test Time

Want to approximate
the integral

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



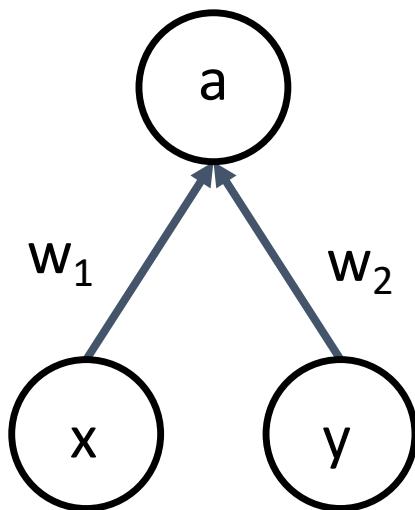
Consider a single neuron:

At test time we have: $E[a] = w_1x + w_2y$

Dropout: Test Time

Want to approximate
the integral

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



Consider a single neuron:

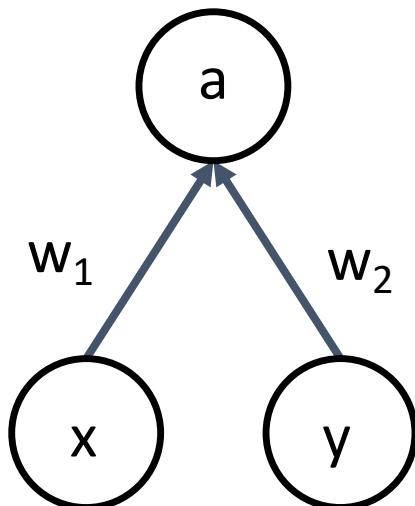
At test time we have: $E[a] = w_1x + w_2y$

$$\begin{aligned} \text{During training we have: } E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

Dropout: Test Time

Want to approximate
the integral

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



Consider a single neuron:

At test time we have: $E[a] = w_1x + w_2y$

During training we have: $E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y)$
 $\quad\quad\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y)$
 $\quad\quad\quad = \frac{1}{2}(w_1x + w_2y)$

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

Dropout: Test Time

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

At test time all neurons are active always

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

output at test time = expected output at training time

Dropout Summary

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

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

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

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

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

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

drop in forward pass

scale at test time

More common: “Inverted dropout”

```
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)

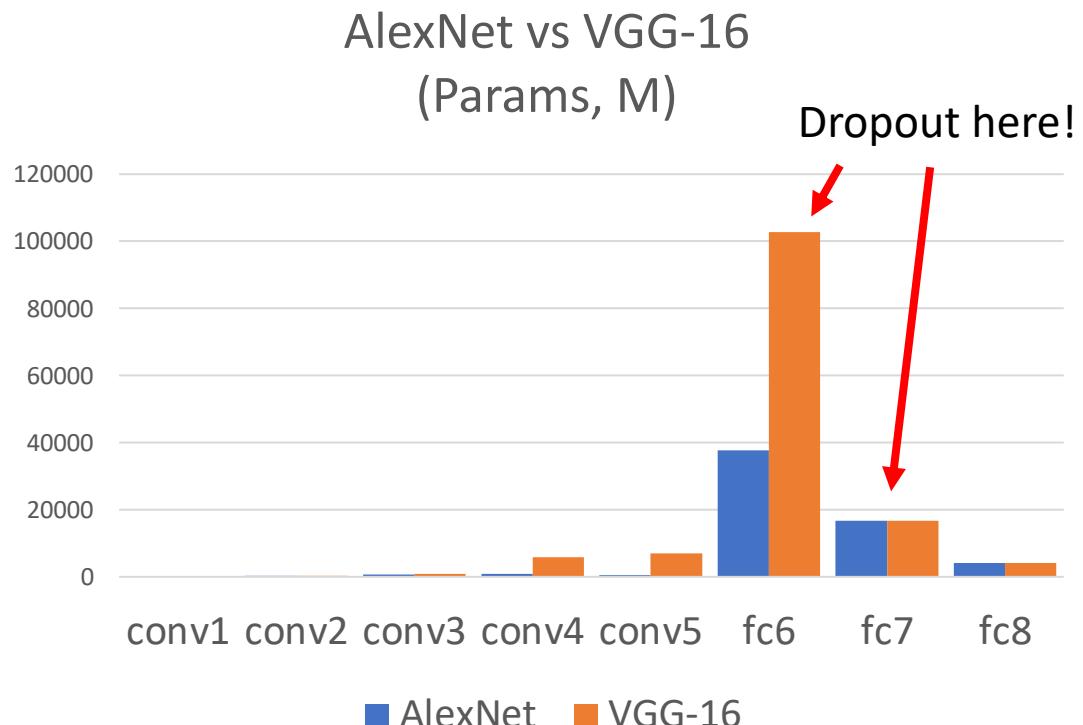
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

Drop and scale
during training

test time is unchanged!

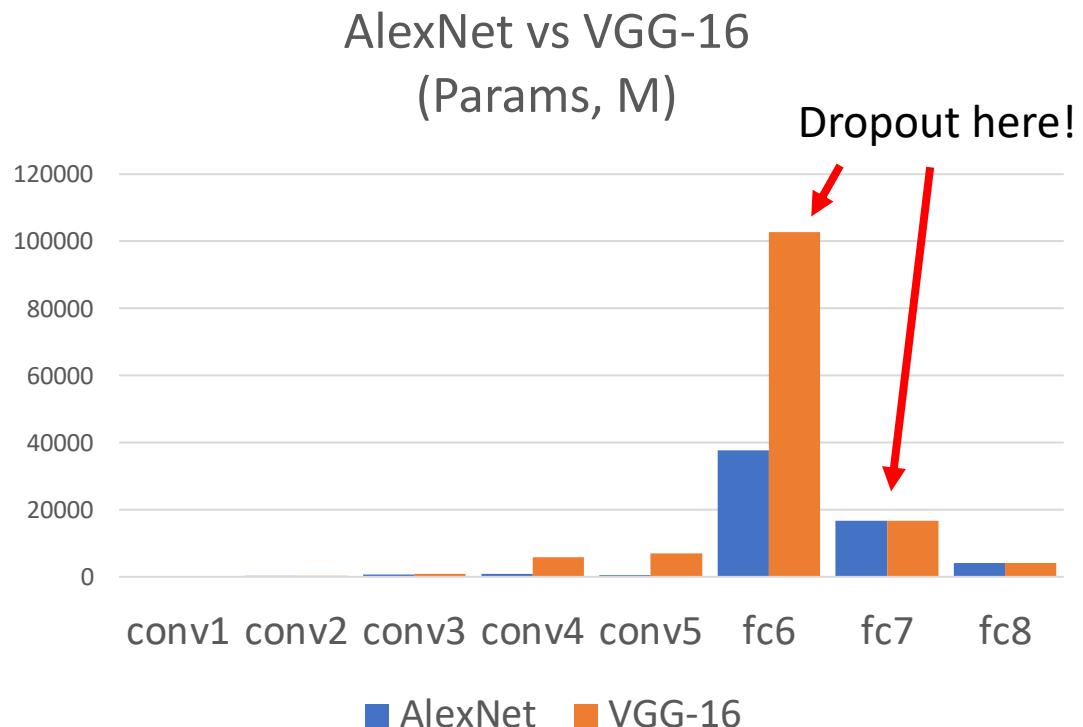
Dropout architectures

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



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)

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

Regularization: A common pattern

Training: Add some kind of randomness

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

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

Regularization: A common pattern

Training: Add some kind of randomness

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

Testing: Average out randomness
(sometimes approximate)

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

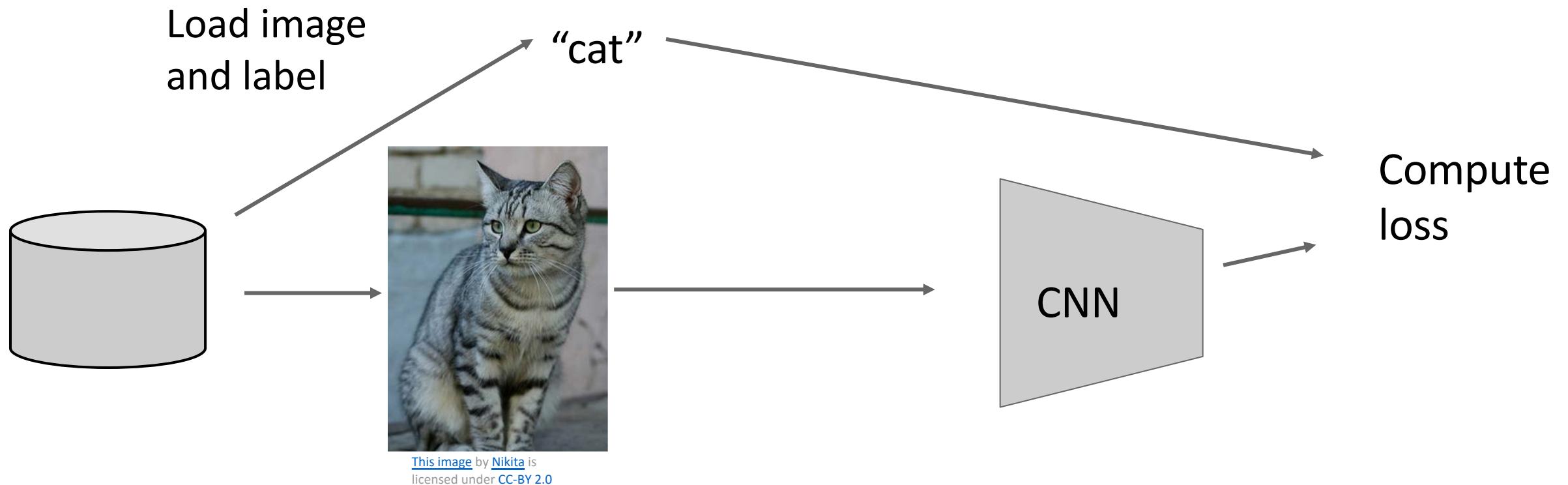
For ResNet and later,
often L2 and Batch
Normalization are
the only regularizers!

Example: Batch
Normalization

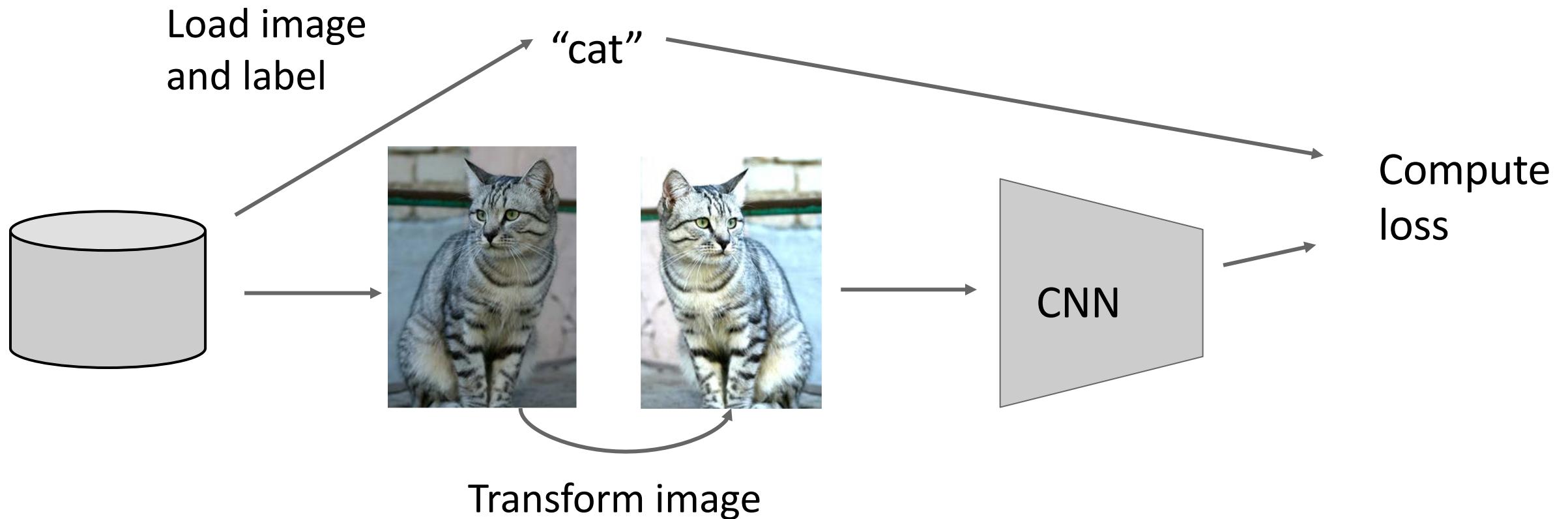
Training: Normalize
using stats from
random minibatches

Testing: Use fixed
stats to normalize

Data Augmentation



Data Augmentation



Data Augmentation: Horizontal Flips

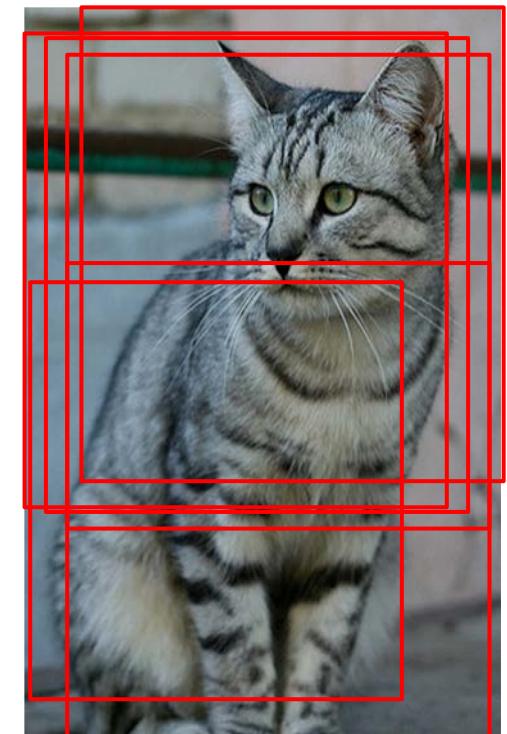


Data Augmentation: Random Crops and Scales

Training: sample random crops / scales

ResNet:

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

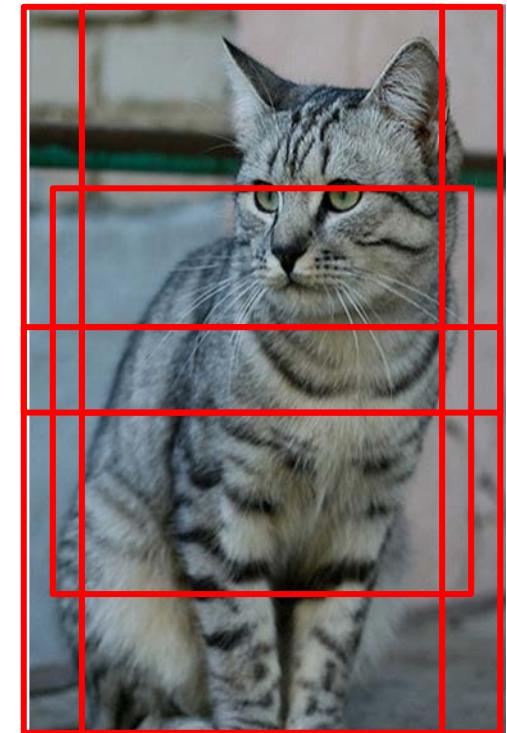


Data Augmentation: Random Crops and Scales

Training: sample random crops / scales

ResNet:

1. Pick random L in range $[256, 480]$
2. Resize training image, short side = L
3. Sample random 224×224 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 10 224×224 crops: 4 corners + center, + flips

Data Augmentation: Color Jitter

Simple: Randomize
contrast and brightness



More Complex:

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

(Used in AlexNet, ResNet, etc)

Data Augmentation: Get creative for your problem!

Random mix/combinations of :

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

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

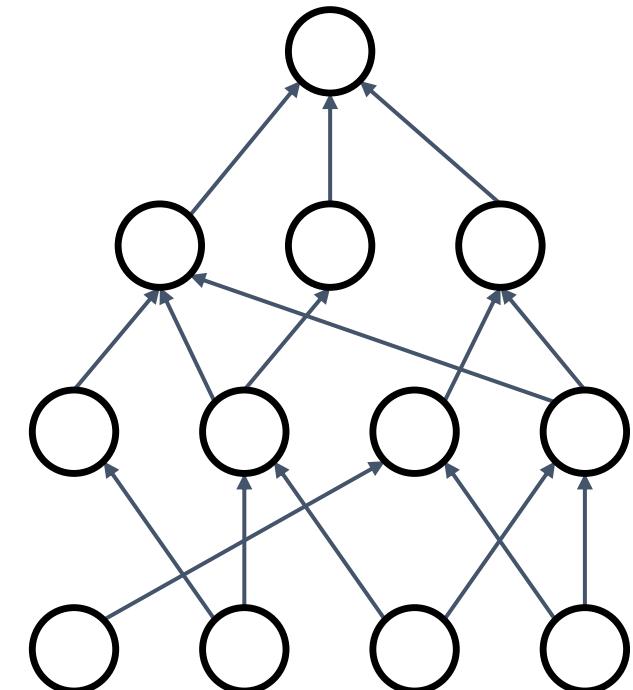
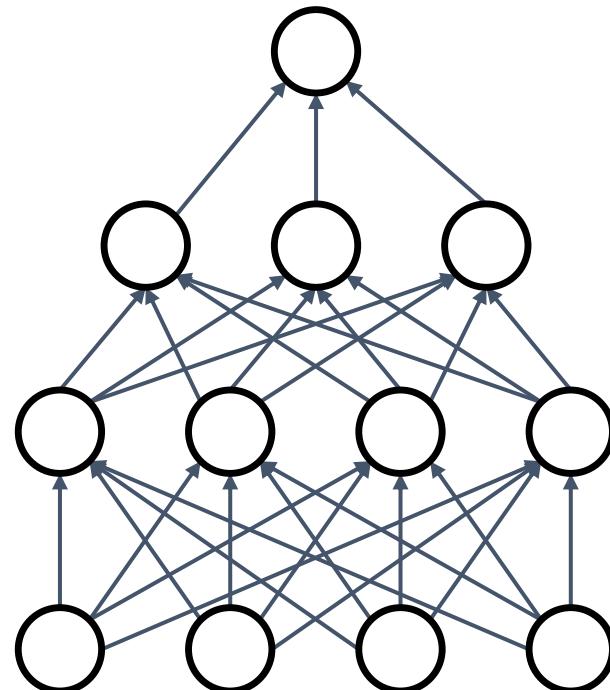
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Regularization: Fractional Pooling

Training: Use randomized pooling regions

Testing: Average predictions over different samples

Examples:

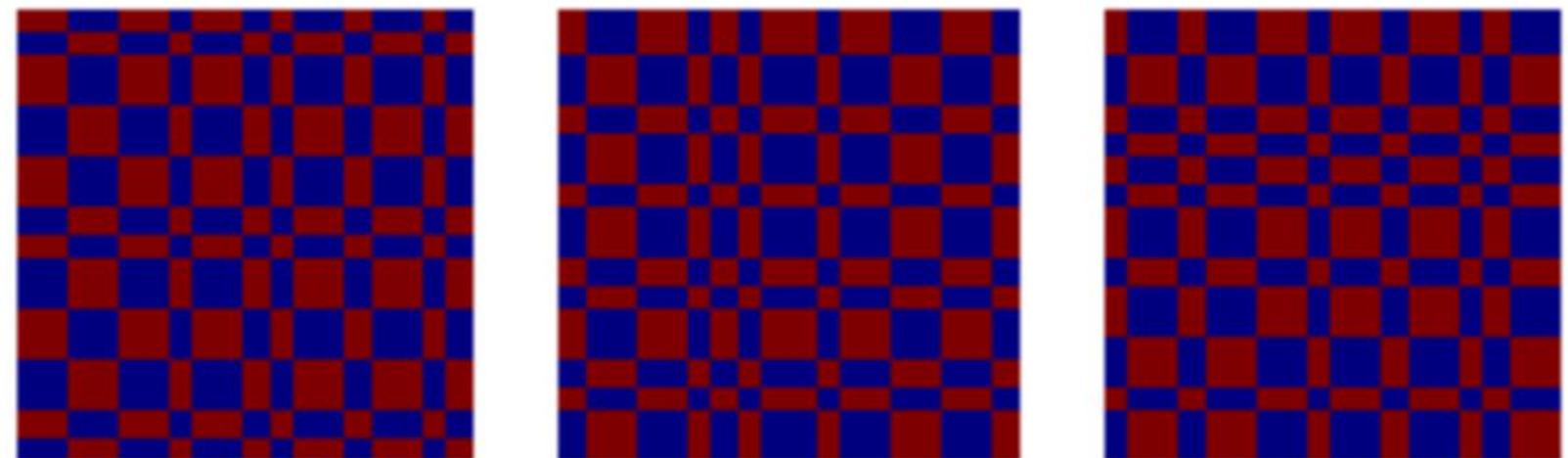
Dropout

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

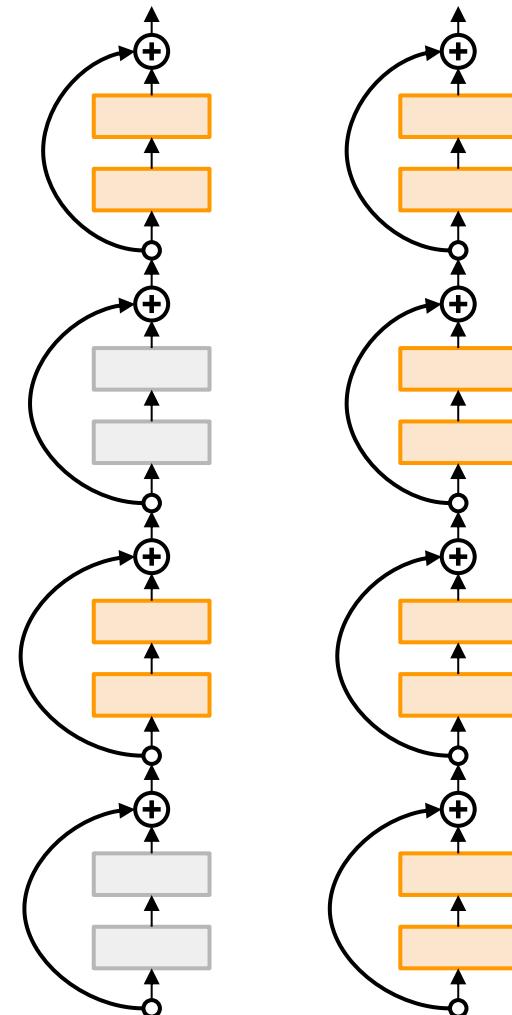
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth



Regularization: Stochastic Depth

Training: Set random images regions to 0

Testing: Use the whole image

Examples:

Dropout

Batch Normalization

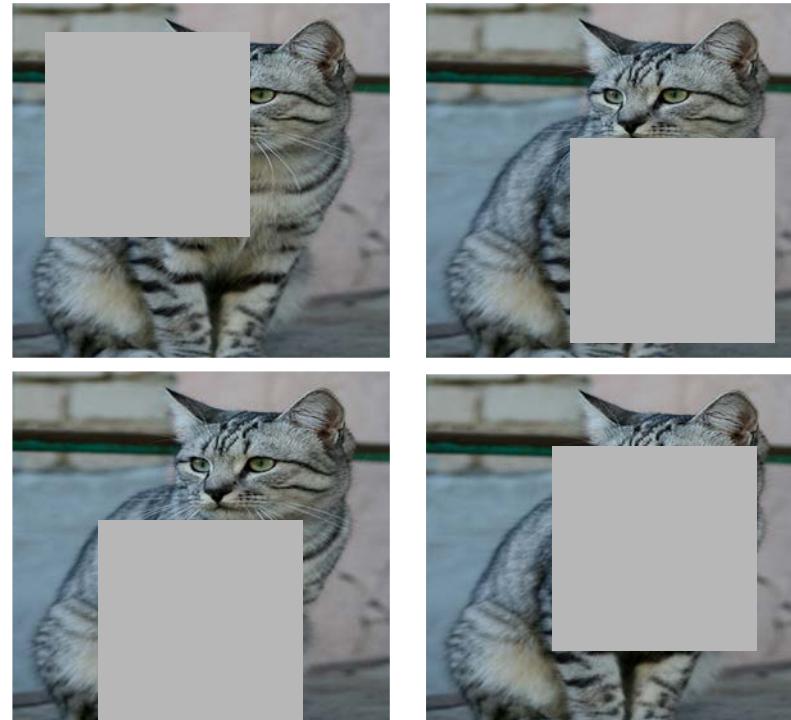
Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout



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

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

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

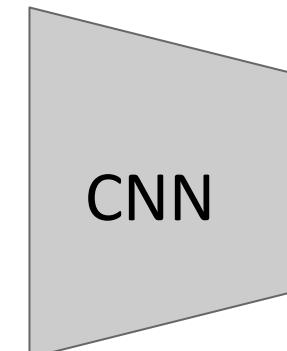
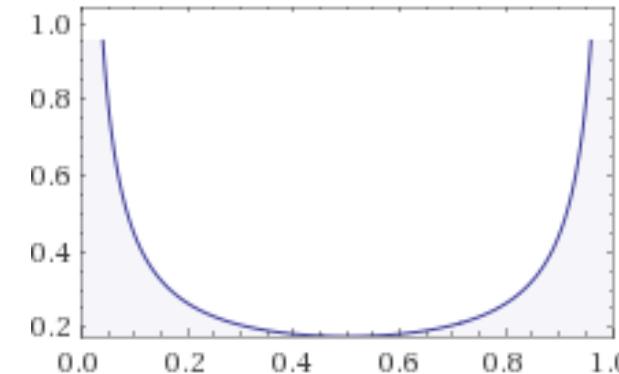
DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout

Mixup



Target label:
cat: 0.4
dog: 0.6

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

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

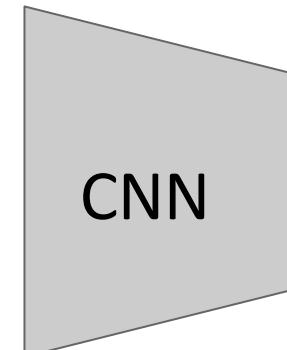
DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout

Mixup



Target label:
cat: 0.4
dog: 0.6

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

Regularization: Mixup

Training: Train on random blends of images

Testing: Use original images

Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout

Mixup

- Consider dropout for large fully-connected layers
- Batch normalization and data augmentation almost always a good idea
- Try cutout and mixup especially for small classification datasets

Summary

1. One time setup

Activation functions, data preprocessing, weight initialization, regularization

Today

2. Training dynamics

Learning rate schedules; large-batch training;
hyperparameter optimization

Next time

3. After training

Model ensembles, transfer learning

Next time:
Training Neural Networks
(part 2)