# Lecture 6:
# Training Neural Networks,
# Part I

# Разбор задачи с самостоятельной

Для заданной функции записать вычислительный граф, рассчитать прямое и обратное распространение по графу для заданного значения входов. При обратном распространении считать начальное значение градиента равным 1. При необходимости результаты, и промежуточные, и окончательные, округлять до второго знака.

*Вариант 2.*

$$f(\mathbf{x},\mathbf{y}) = \frac{x_0}{e^{-(x_0 y_0 + x_1 y_1)}}$$

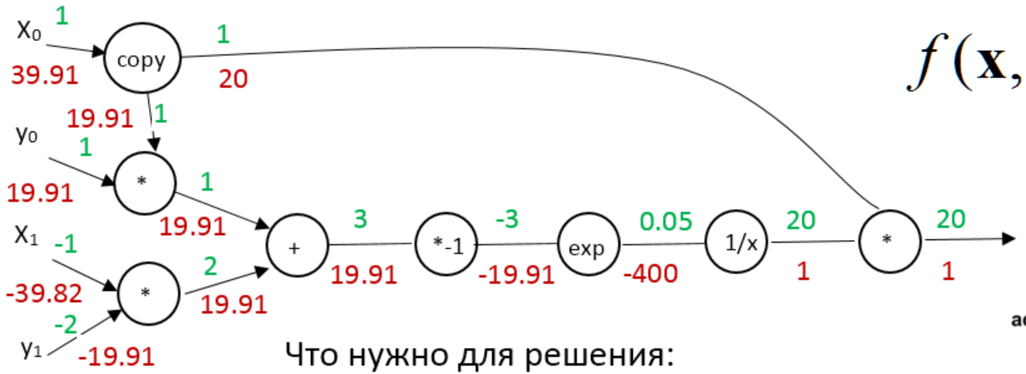
Рассчитать прямое распространение и градиенты

$\partial f / \partial x_0$, $\partial f / \partial x_1$, $\partial f / \partial y_0$, $\partial f / \partial y_1$

Начальные значения: $x_0 = 1$, $x_1 = -1$, $y_0 = 1$, $y_1 = -2$

# Разбор задачи с самостоятельной

Решение:




Задача:

$$f(\mathbf{x},\mathbf{y}) = \frac{x_0}{e^{-(x_0 y_0 + x_1 y_1)}}$$

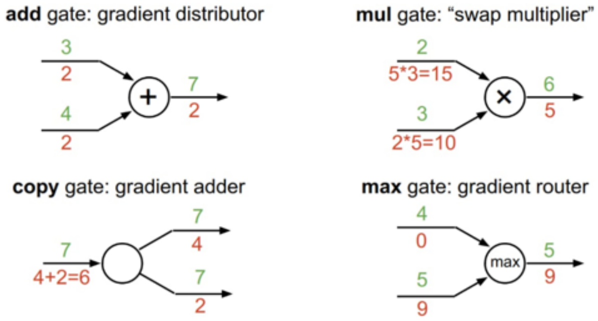
Что нужно для решения:

$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$

$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$

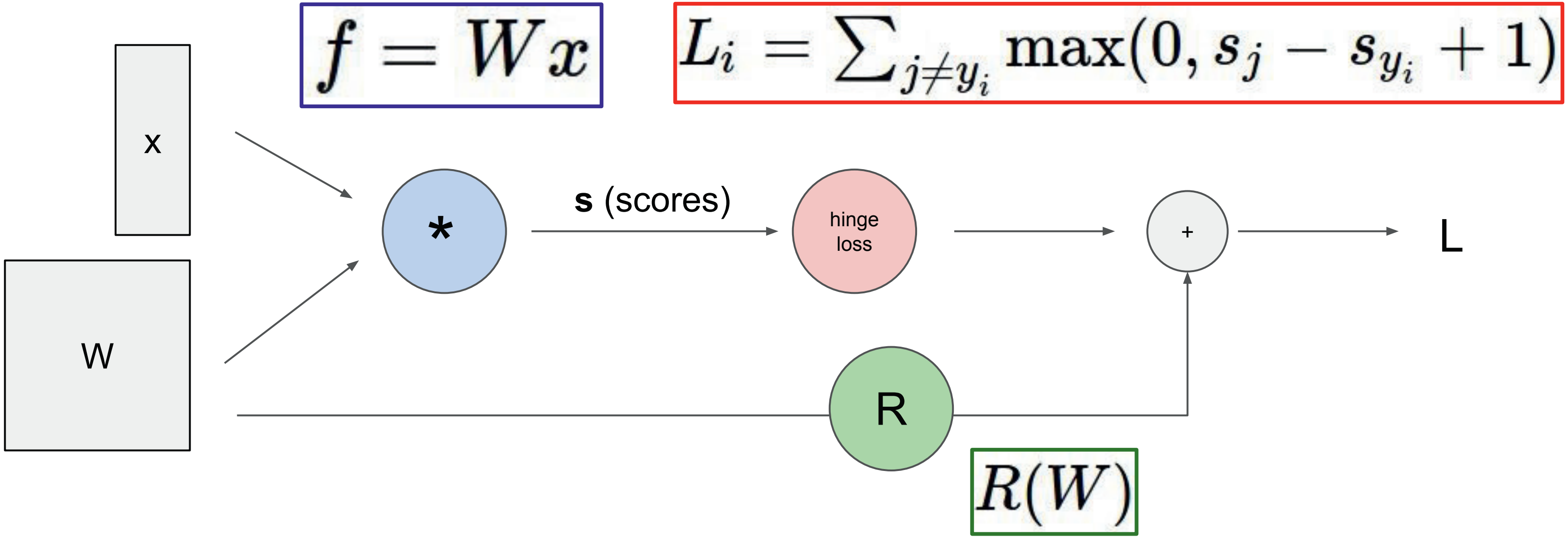
$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$



Where we are now...

## Computational graphs

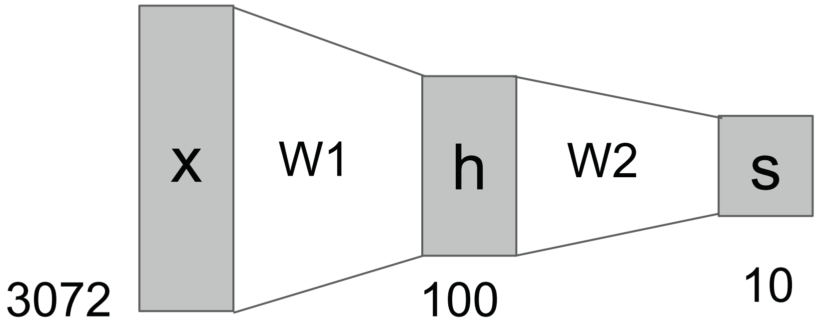


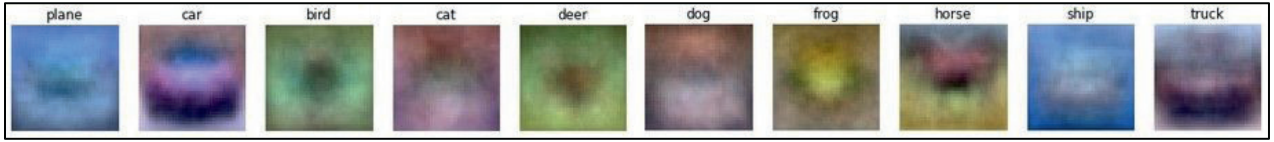
Where we are now...

## Neural Networks

Linear score function: $f = Wx$

2-layer Neural Network $f = W_2 \max(0, W_1 x)$



# Where we are now...

## Convolutional Neural Networks

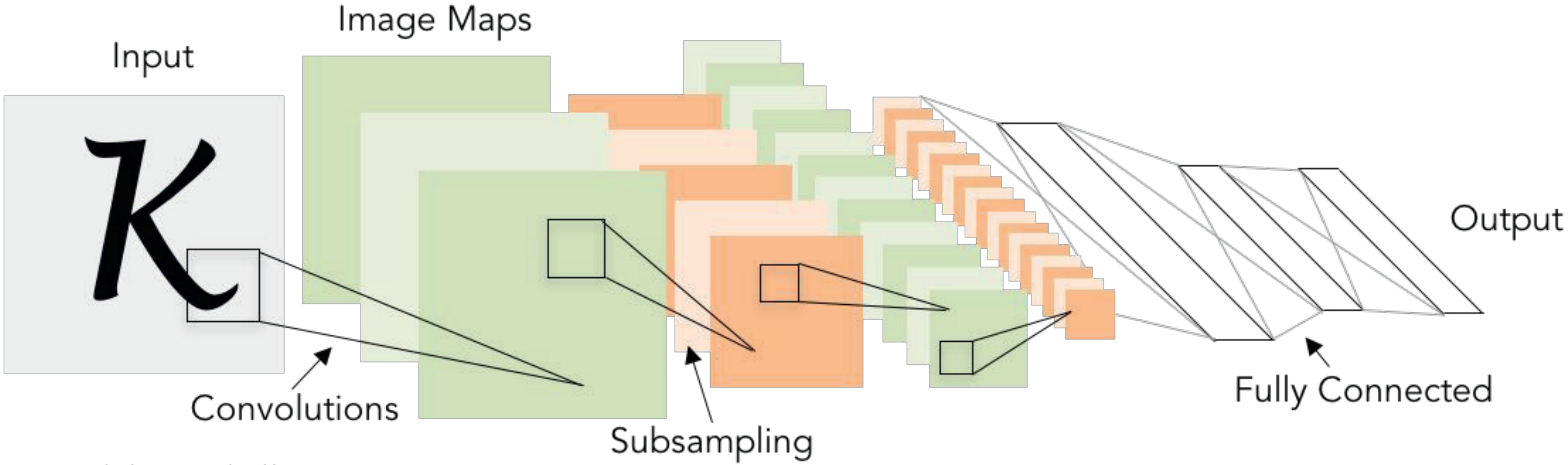
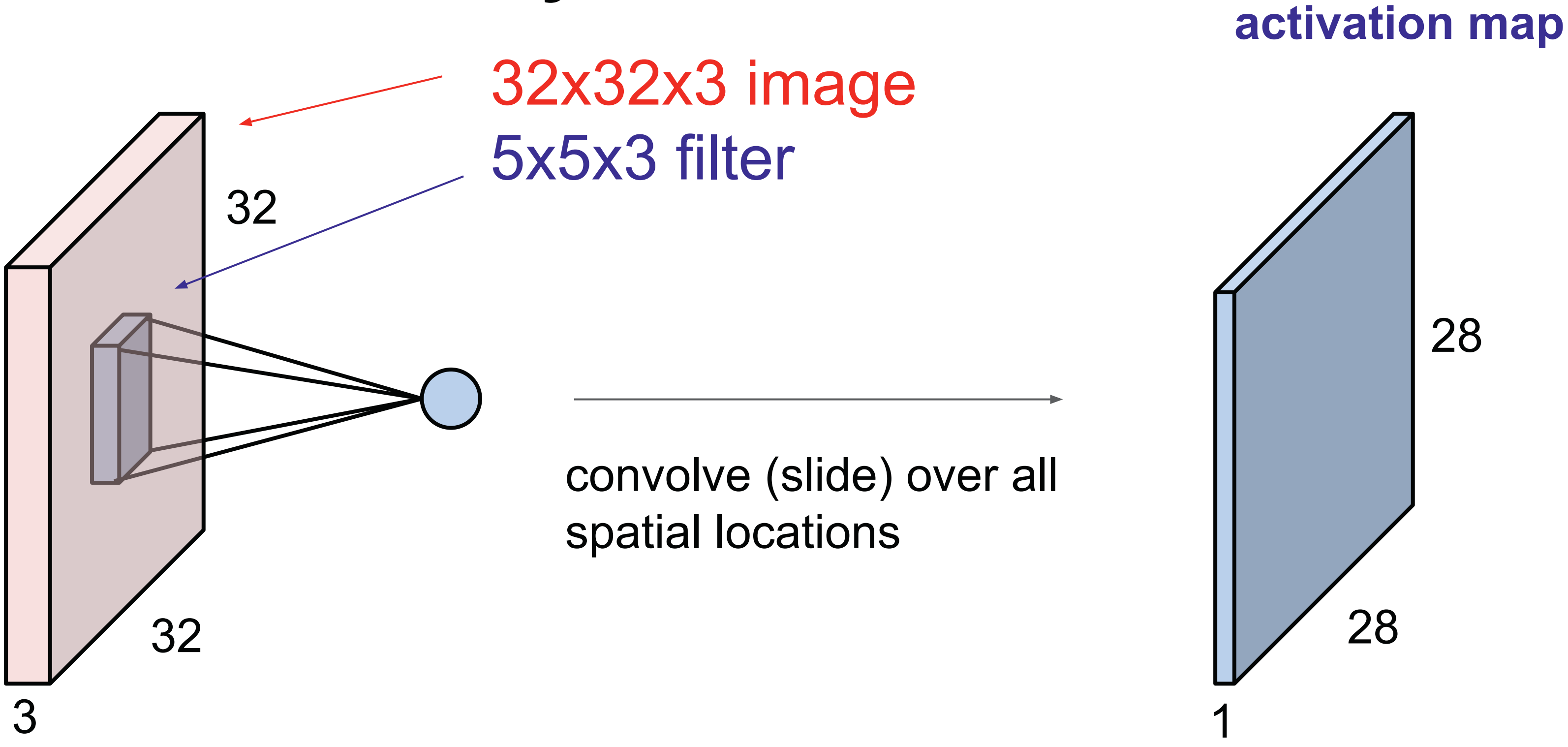



Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

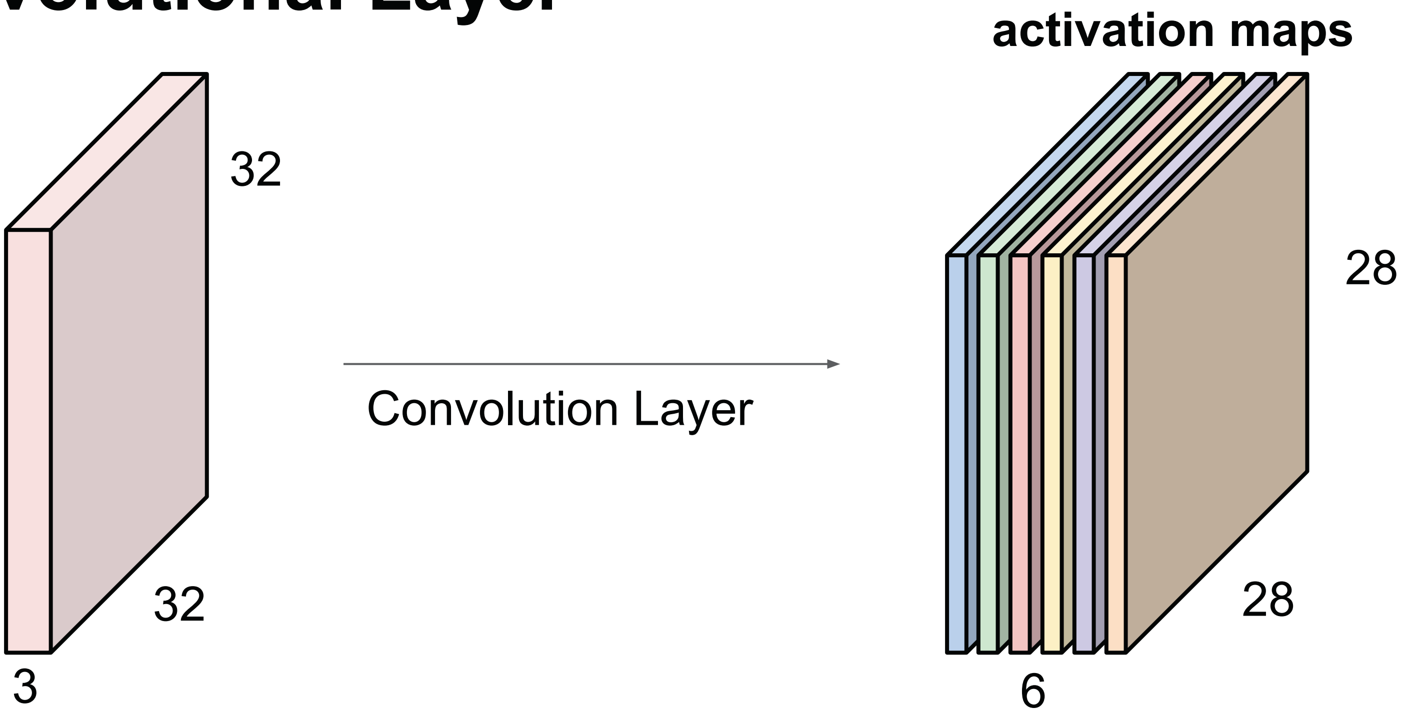
## Convolutional Layer



Where we are now...

# Convolutional Layer

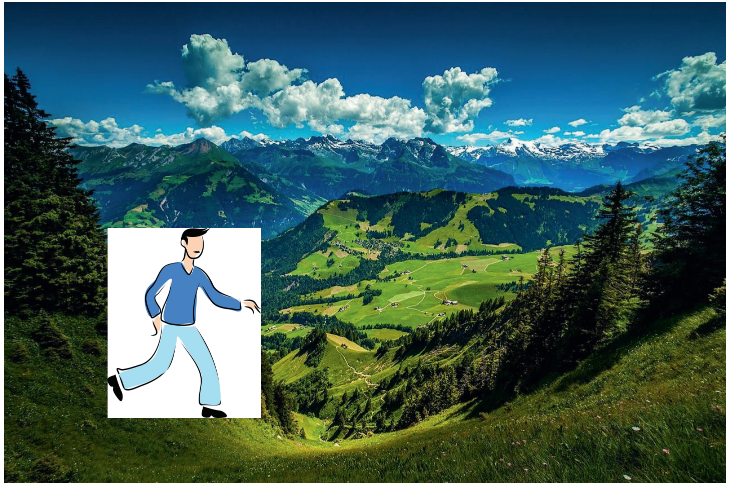
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

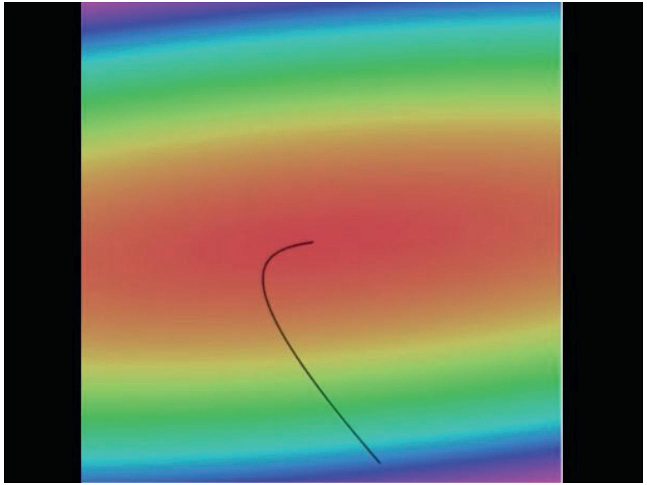



We stack these up to get a "new image" of size 28x28x6!

Where we are now...

# Learning network parameters through optimization





```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Where we are now...

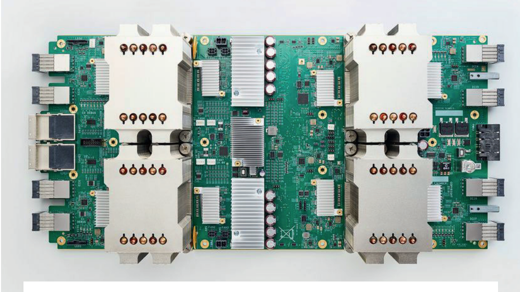
# Mini-batch SGD

Loop:

1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Where we are now...

# Hardware + Software



**PyTorch**

**TensorFlow**

# Next: Training Neural Networks

# Overview

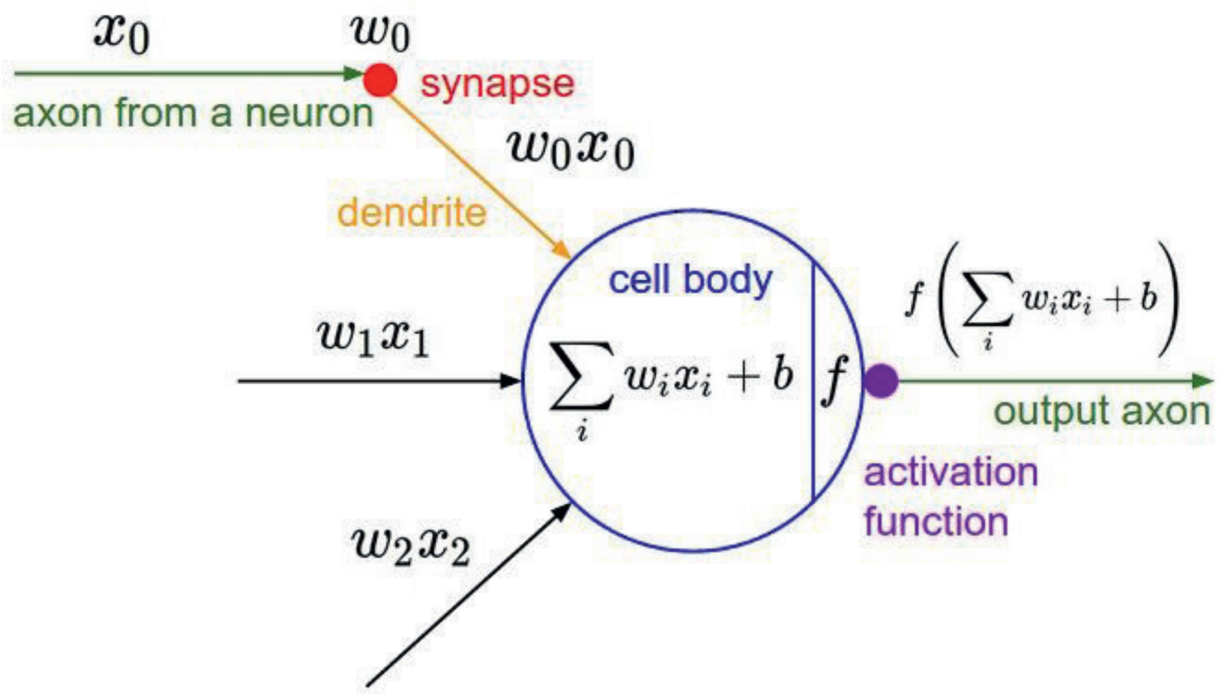
1. **One time setup**
   *activation functions, preprocessing, weight initialization, regularization, gradient checking*
2. **Training dynamics**
   transfer learning, *babysitting the learning process, parameter updates, hyperparameter optimization*
3. **Evaluation**
   *model ensembles, test-time augmentation*

# Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Transfer learning

# Activation Functions

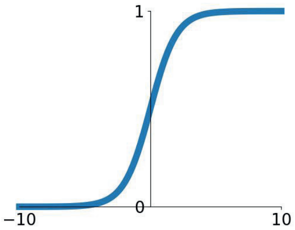
# Activation Functions



# Activation Functions

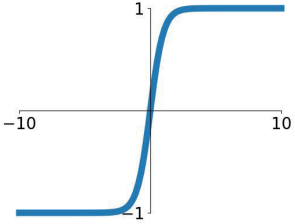
**Sigmoid**

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



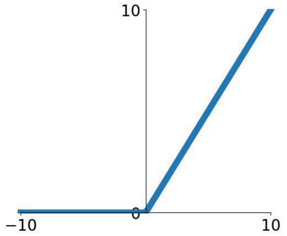

**tanh**

$$\tanh(x)$$



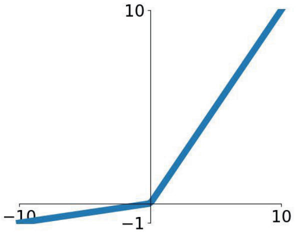

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

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

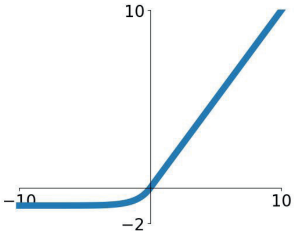



**Maxout**

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

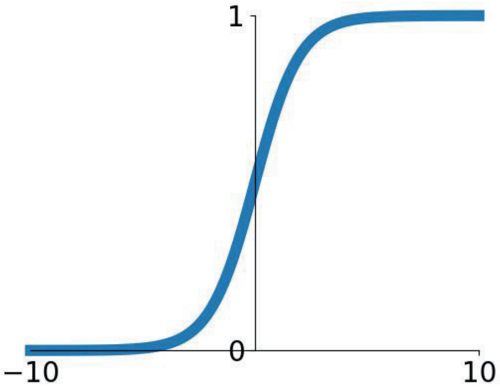
**ELU**

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



# Activation Functions

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

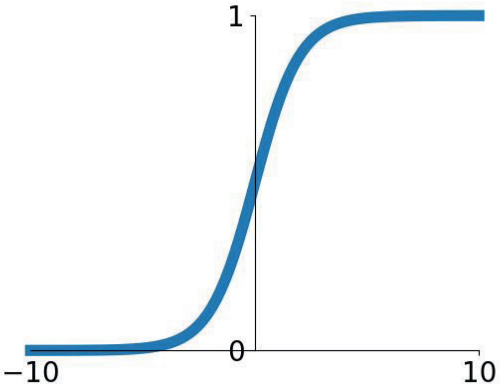



**Sigmoid**

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

# Activation Functions

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

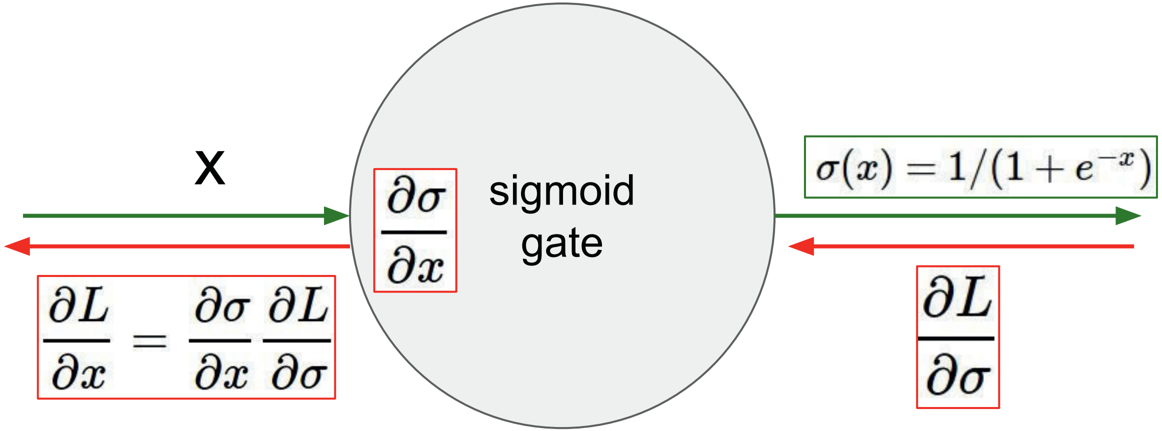


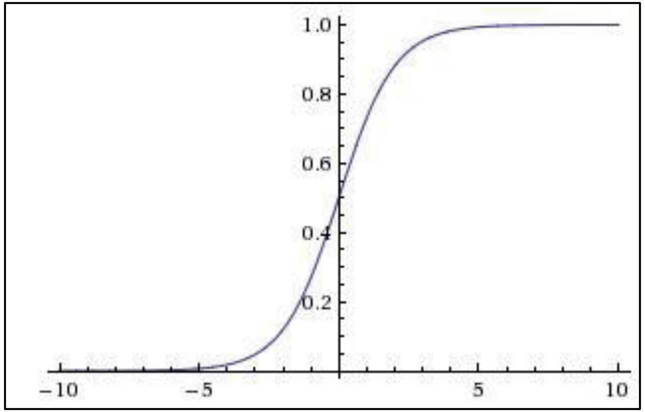

**Sigmoid**

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

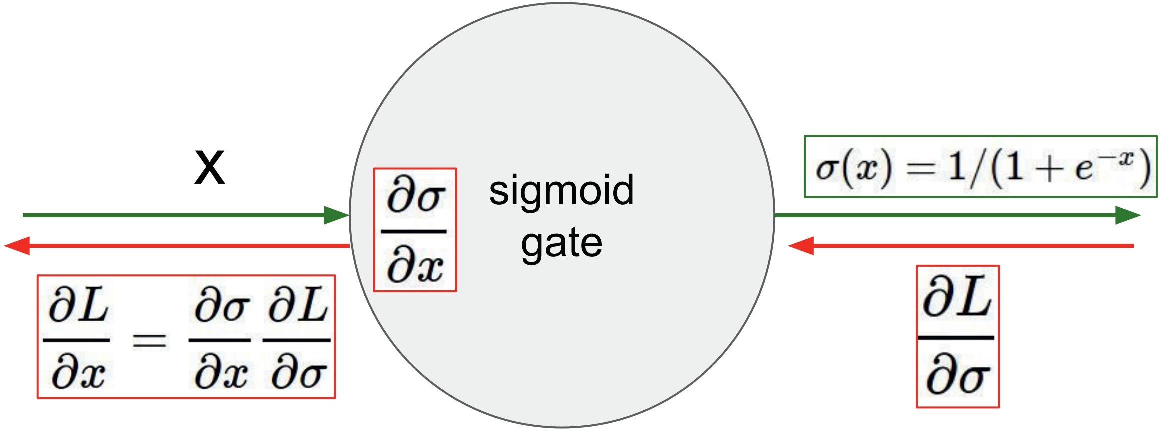
3 problems:

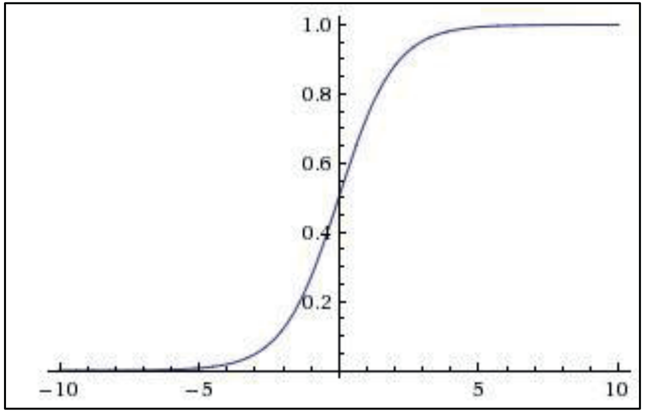
1. Saturated neurons "kill" the gradients



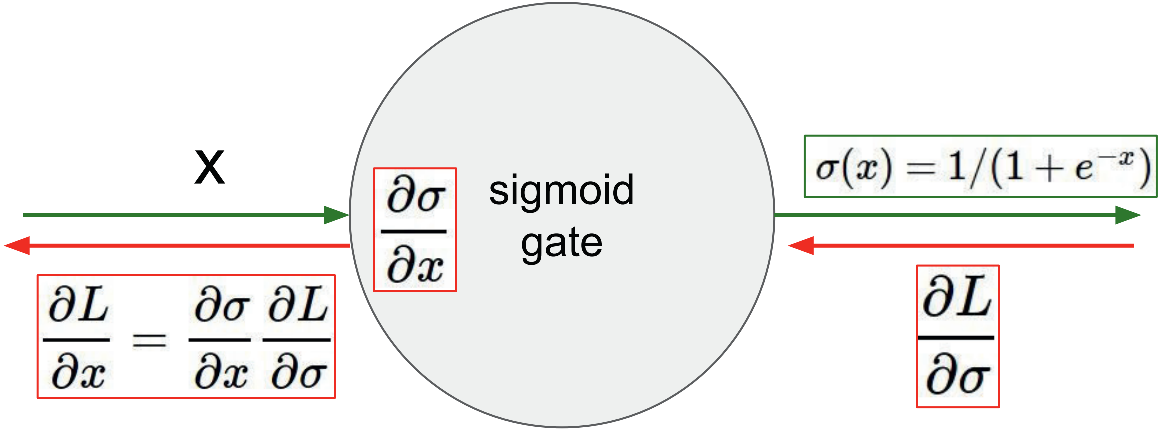

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

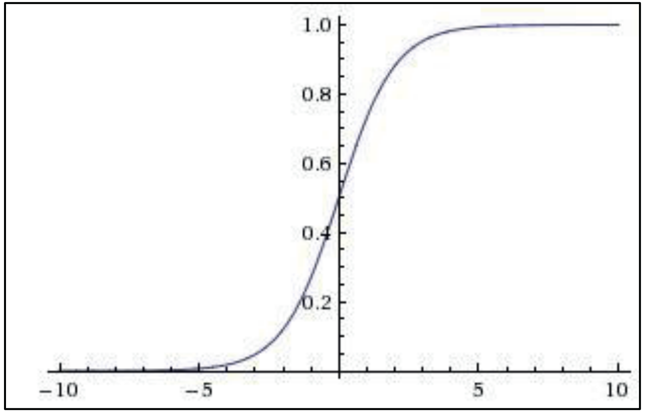



What happens when x = -10?

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

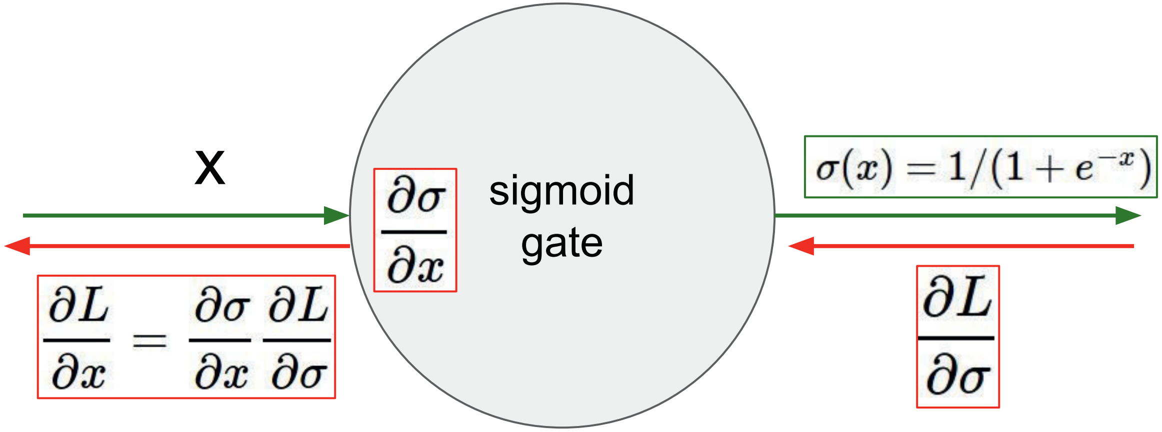


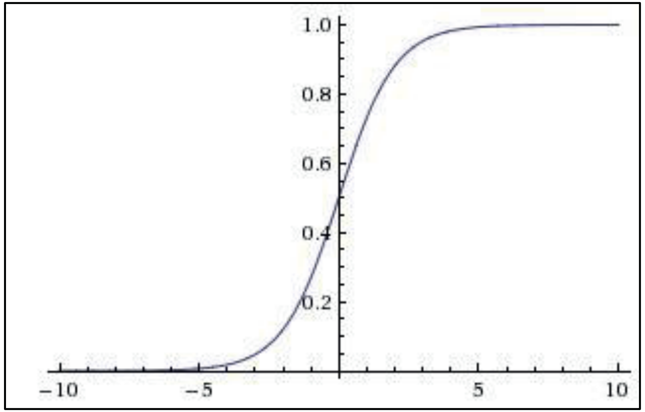




What happens when x = -10?

What happens when x = 0?

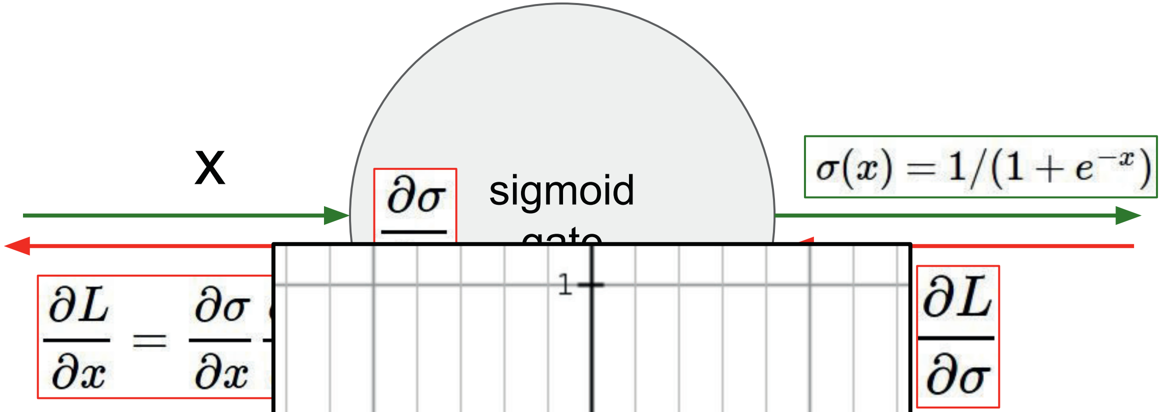
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

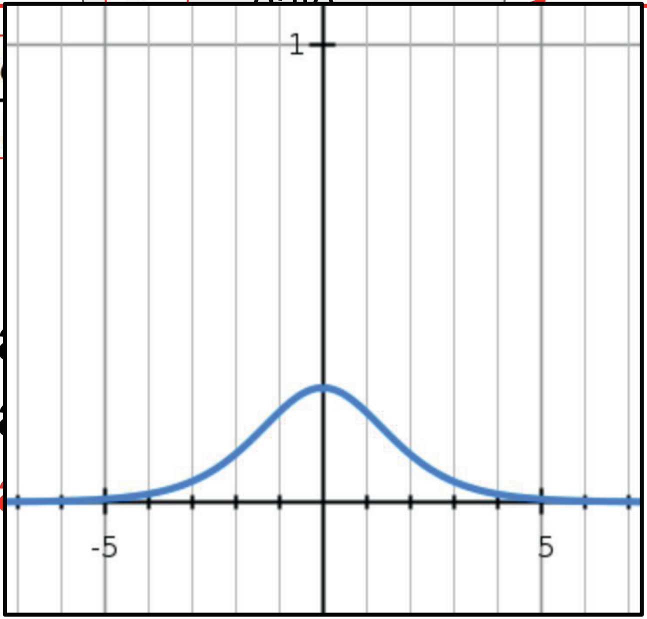


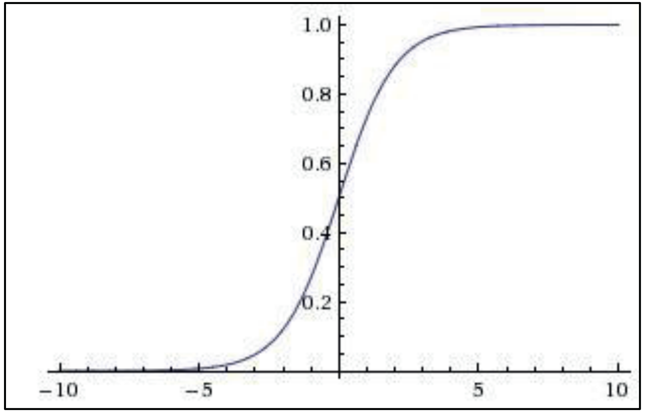




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

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




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

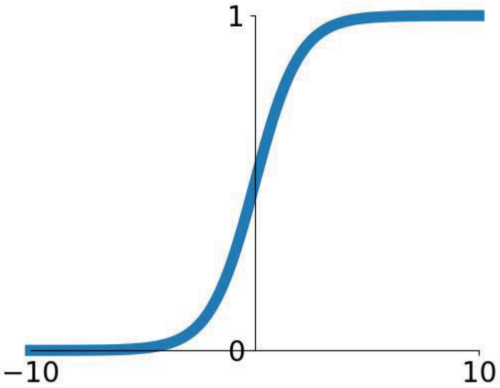
What ha

What ha

What ha

# Activation Functions

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




**Sigmoid**

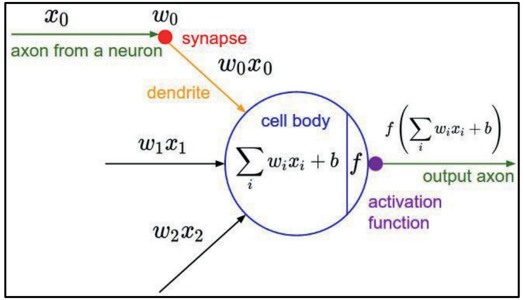
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

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

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

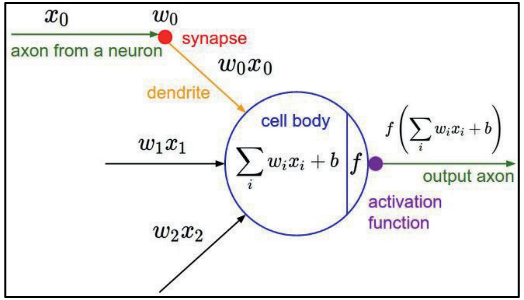
$$f\left(\sum_i w_i x_i + b\right)$$




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

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

$$f\left(\sum_i w_i x_i + b\right)$$

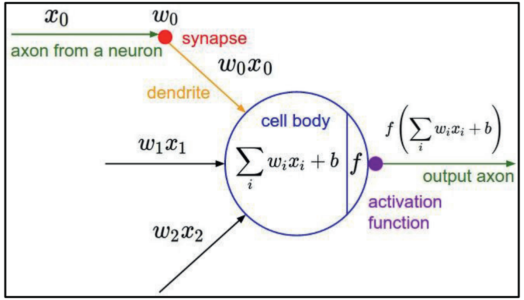



What can we say about the gradients on **w**?

$$\frac{\partial L}{\partial w} = \sigma(\textstyle\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))x \times upstream\_gradient$$

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

$$f\left(\sum_i w_i x_i + b\right)$$



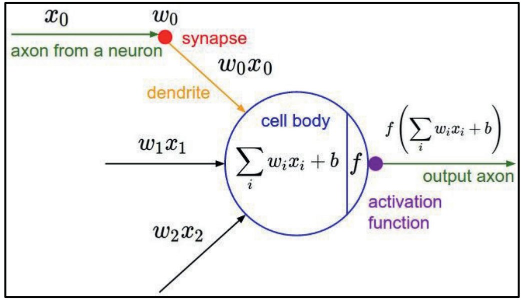

What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\textstyle\sum_i w_i x_i + b)(1 - \sigma(\textstyle\sum_i w_i x_i + b))} x \times upstream\_gradient$$

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

$$f\left(\sum_i w_i x_i + b\right)$$




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

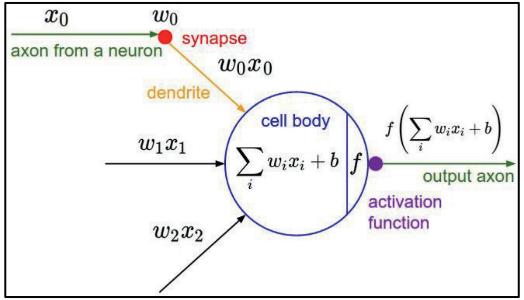
We know that local gradient of sigmoid is always positive

We are assuming x is always positive

$$\frac{\partial L}{\partial w} = \boxed{\sigma(\sum_i w_i x_i + b)(1 - \sigma(\sum_i w_i x_i + b))}\boxed{x} \times upstream\_gradient$$

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

$$f\left(\sum_i w_i x_i + b\right)$$




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

We know that local gradient of sigmoid is always positive

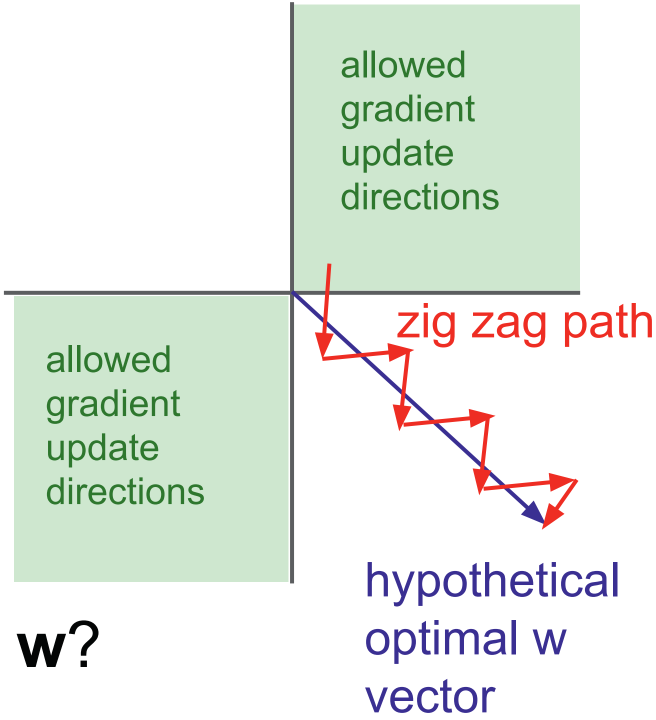
We are assuming x is always positive

So!! Sign of gradient **for all w$_i$** is the same as the sign of upstream scalar gradient!

$$\boxed{\frac{\partial L}{\partial w}} = \boxed{\sigma(\textstyle\sum_i w_i x_i + b)(1-\sigma(\sum_i w_i x_i + b))}\boxed{x} \times \boxed{upstream\_gradient}$$

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

$$f\left(\sum_i w_i x_i + b\right)$$

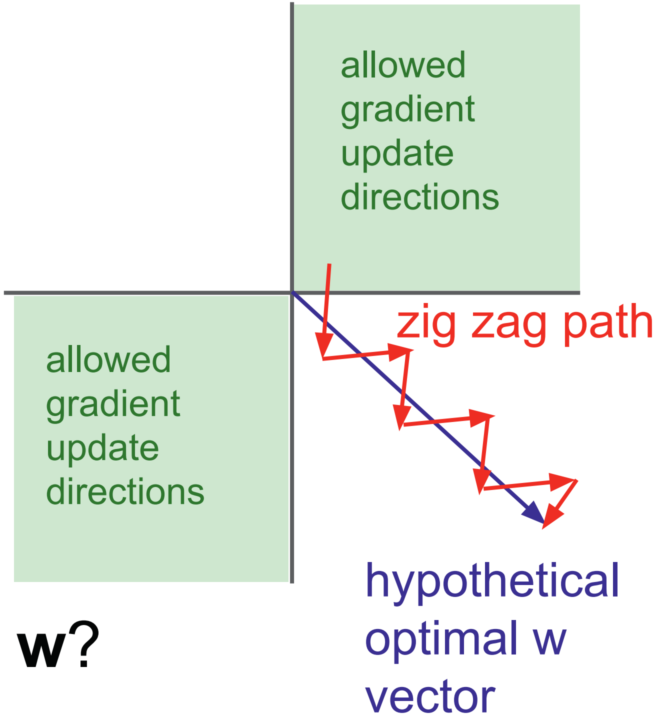



What can we say about the gradients on **w**?

Always all positive or all negative :(

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

$$f\left(\sum_i w_i x_i + b\right)$$



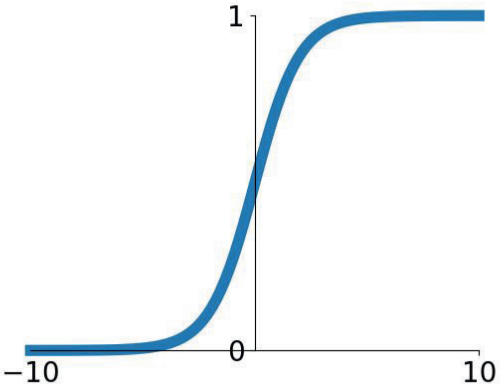

What can we say about the gradients on **w**?

Always all positive or all negative :(

(For a single element! Minibatches help)

# Activation Functions

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



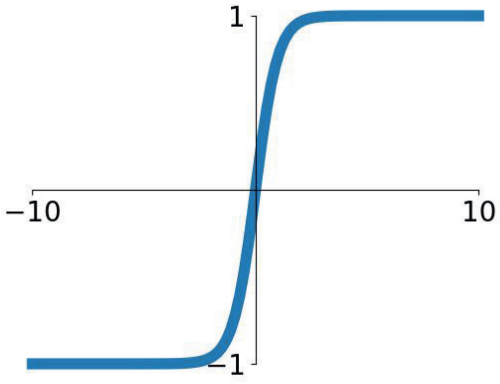

**Sigmoid**

- 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

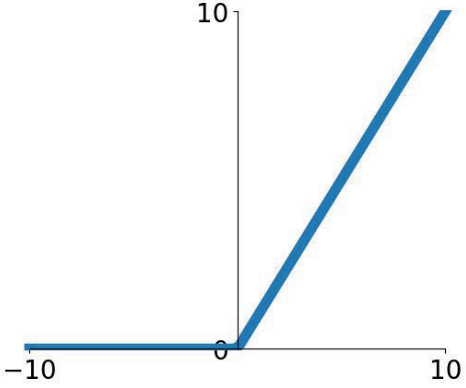



**tanh(x)**

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

[LeCun et al., 1991]

# Activation Functions



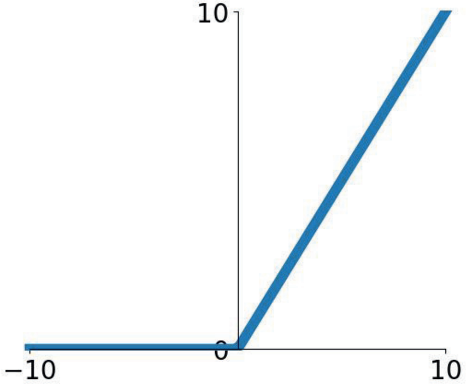

**ReLU**
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

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

[Krizhevsky et al., 2012]

# Activation Functions



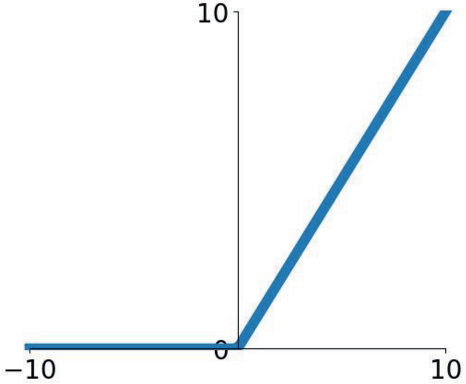

**ReLU**
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- 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



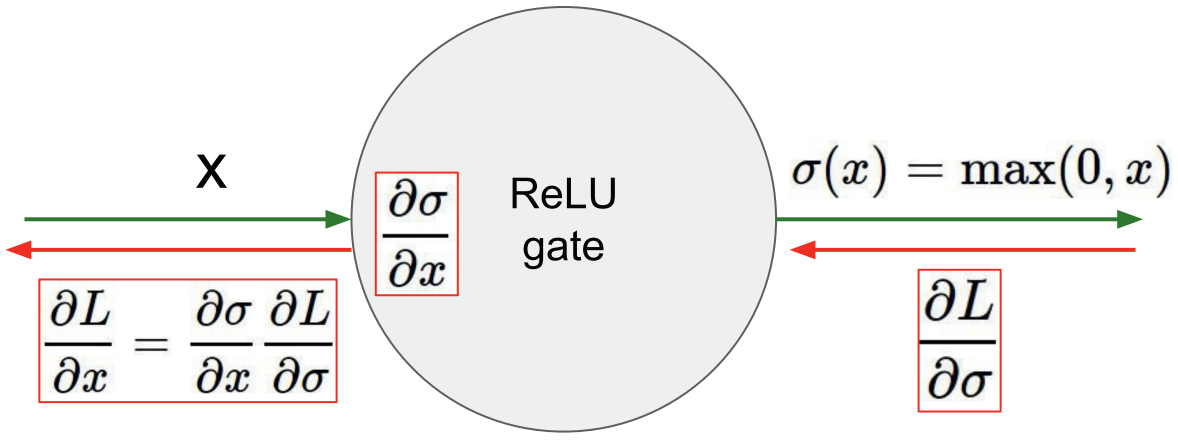

**ReLU**
(Rectified Linear Unit)

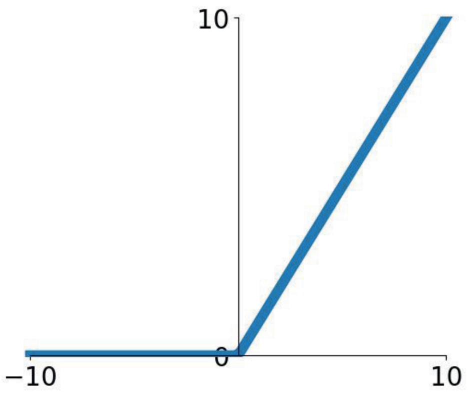
- Computes **f(x) = max(0,x)**

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

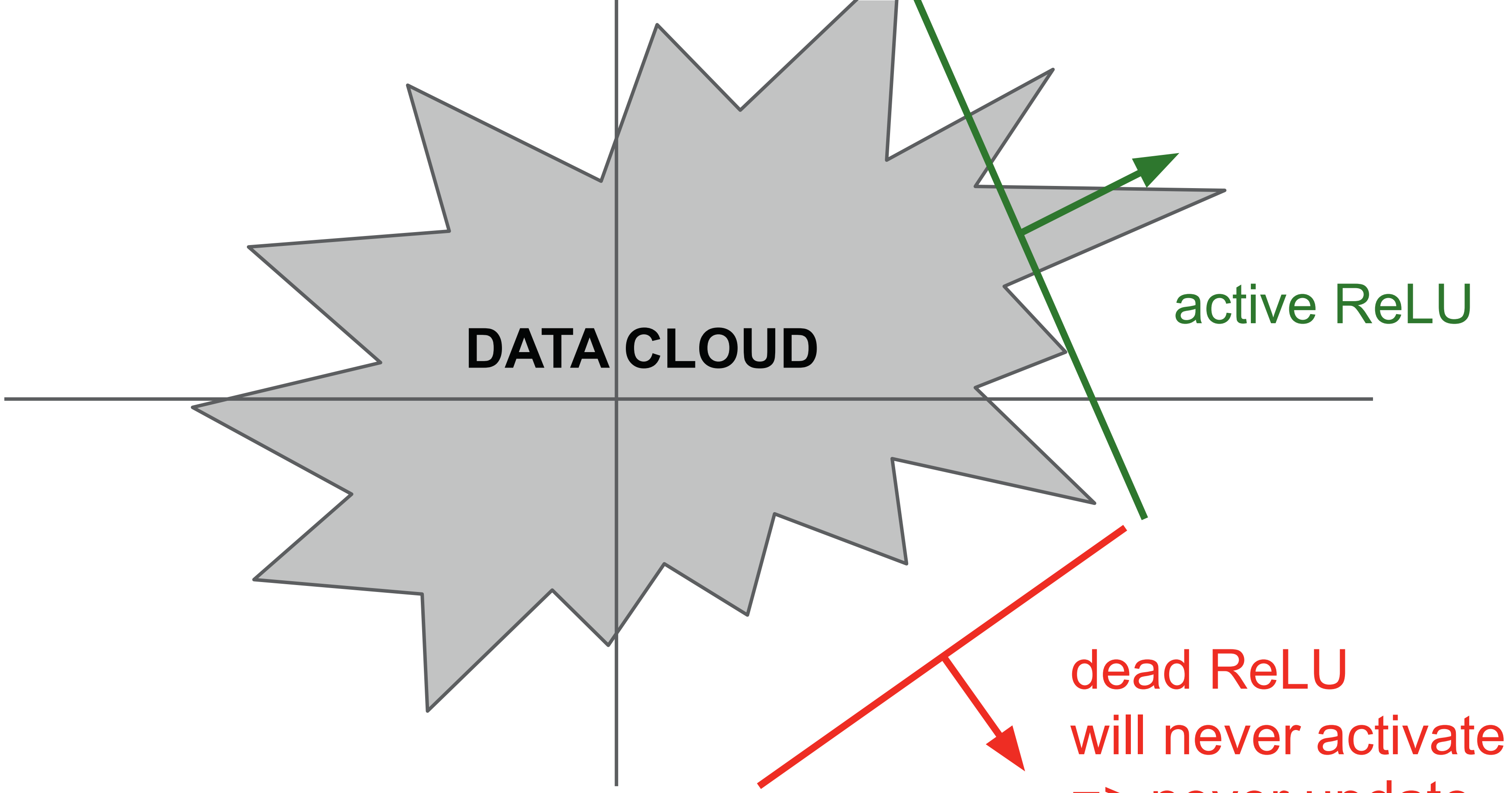
- Not zero-centered output
- An annoyance:

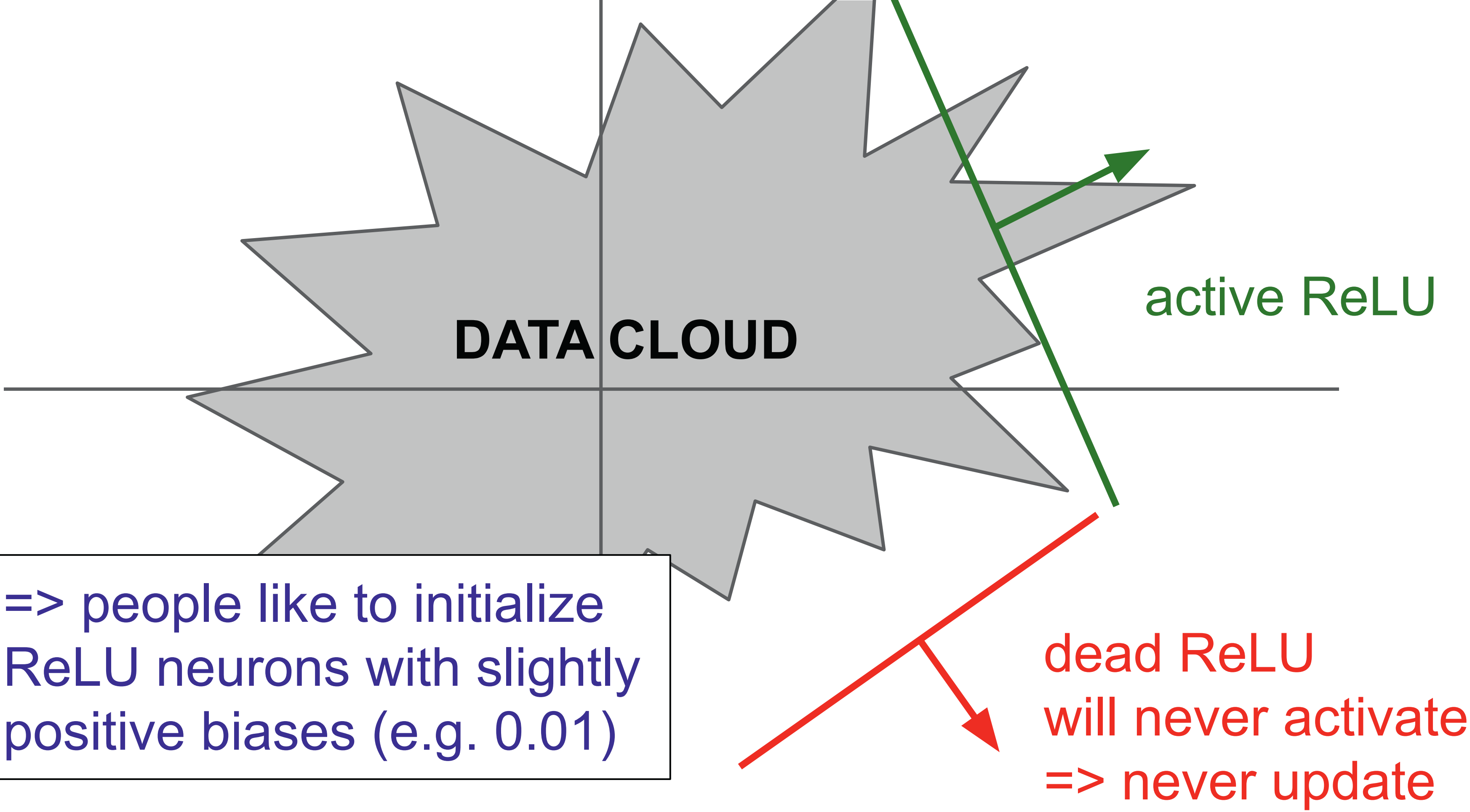
hint: what is the gradient when $x < 0$?




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

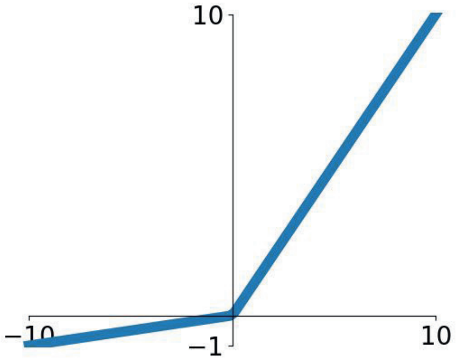

DATA CLOUD
active ReLU
dead ReLU
will never activate
=> never update




=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

# Activation Functions

[Mass et al., 2013]
[He et al., 2015]



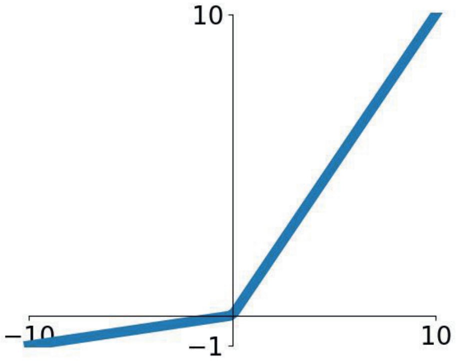

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

# Activation Functions

[Mass et al., 2013]
[He et al., 2015]




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

**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

**Parametric Rectifier (PReLU)**

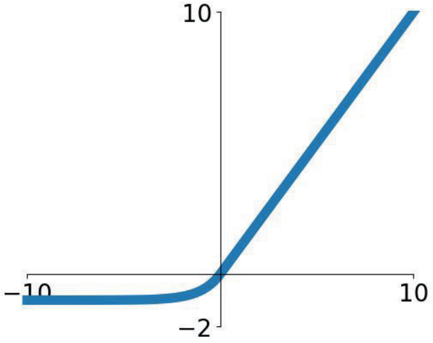
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha
(parameter)

# Activation Functions

[Clevert et al., 2015]

## Exponential Linear Units (ELU)




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

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

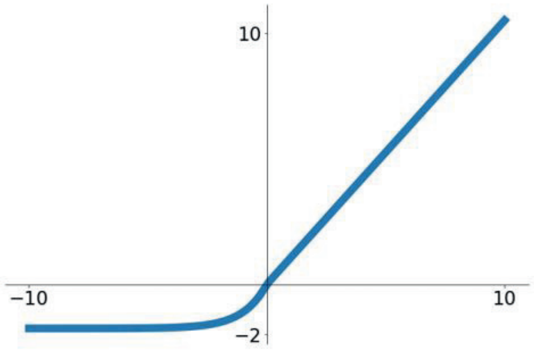
(Alpha default = 1)

- Computation requires exp()

# Activation Functions

[Klambauer et al. ICLR 2017]

## Scaled Exponential Linear Units (SELU)




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

$\alpha$ = 1.6733, $\lambda$ = 1.0507

- Scaled versionof ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

# Maxout "Neuron"

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

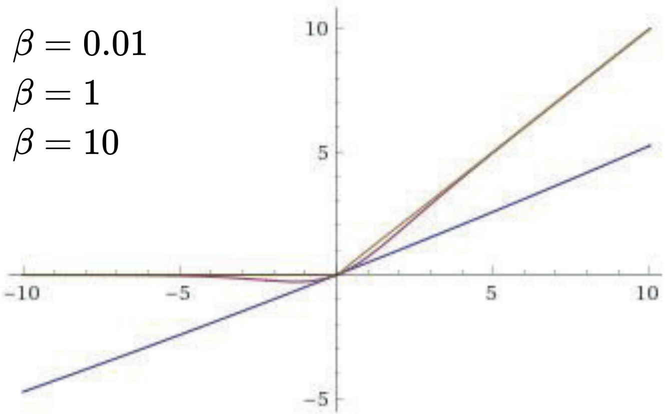
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

# Activation Functions

[Ramachandran et al. 2018]

## Swish




$$f(x) = x\sigma(\beta x)$$

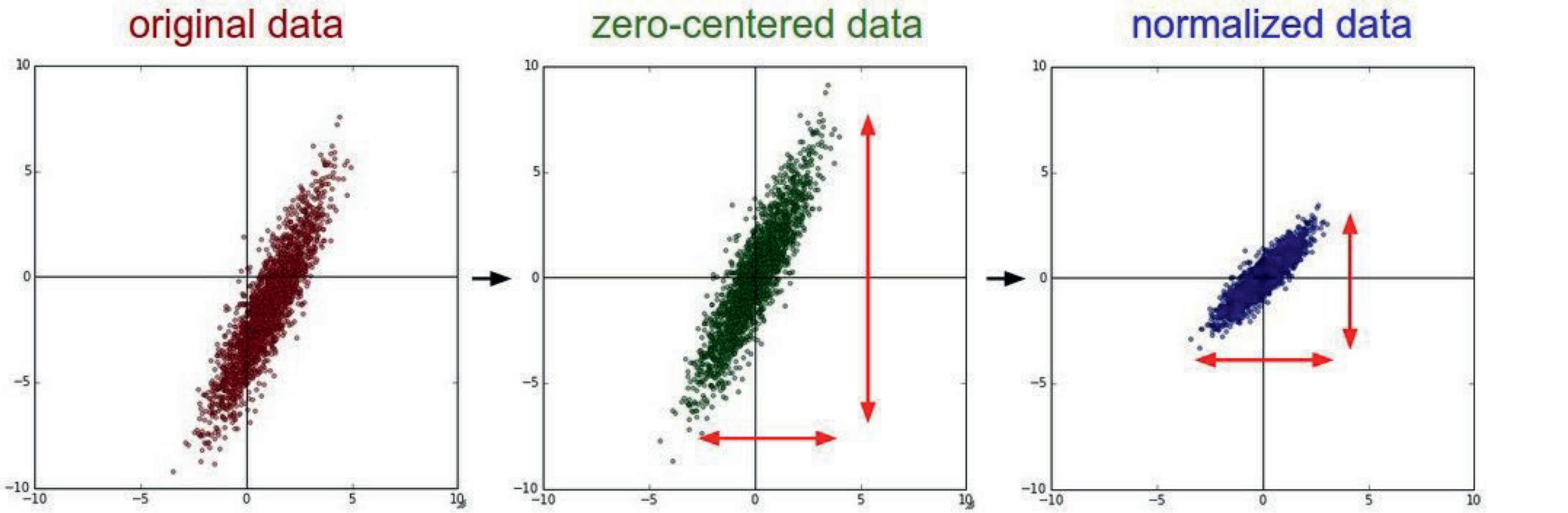
- They trained a neural network to generate and test out different non-linearities.
- Swish outperformed all other options for CIFAR-10 accuracy

# TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU / SELU
    - To squeeze out some marginal gains
- Try PReLU with smaller learning rate
- Don't use sigmoid or tanh

# Data Preprocessing

# Data Preprocessing

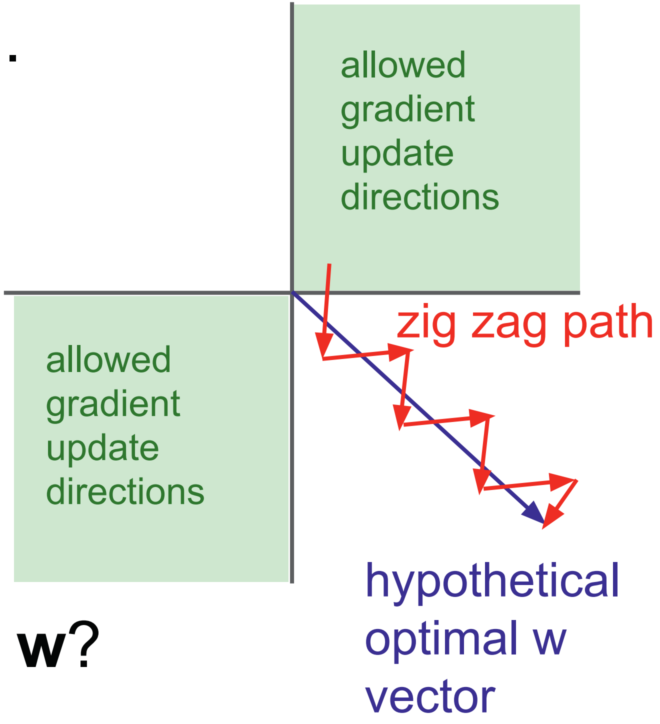



```
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```

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

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

$$f\left(\sum_i w_i x_i + b\right)$$

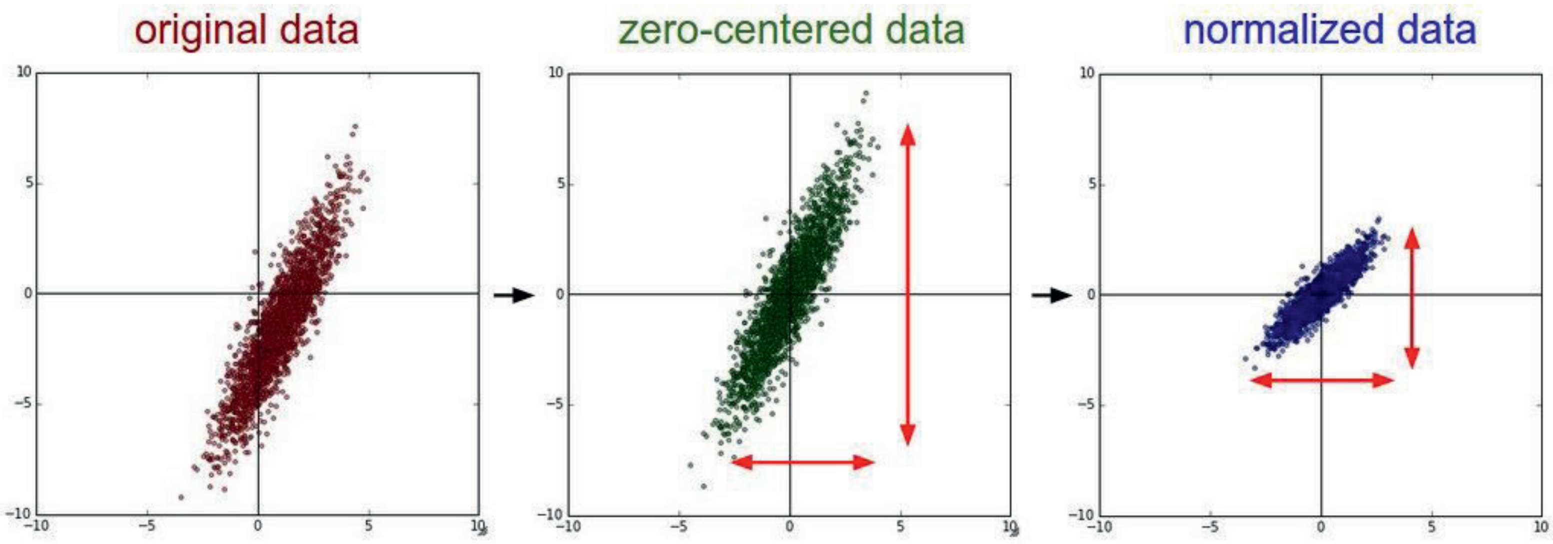



What can we say about the gradients on **w**?

Always all positive or all negative :(

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

# Data Preprocessing



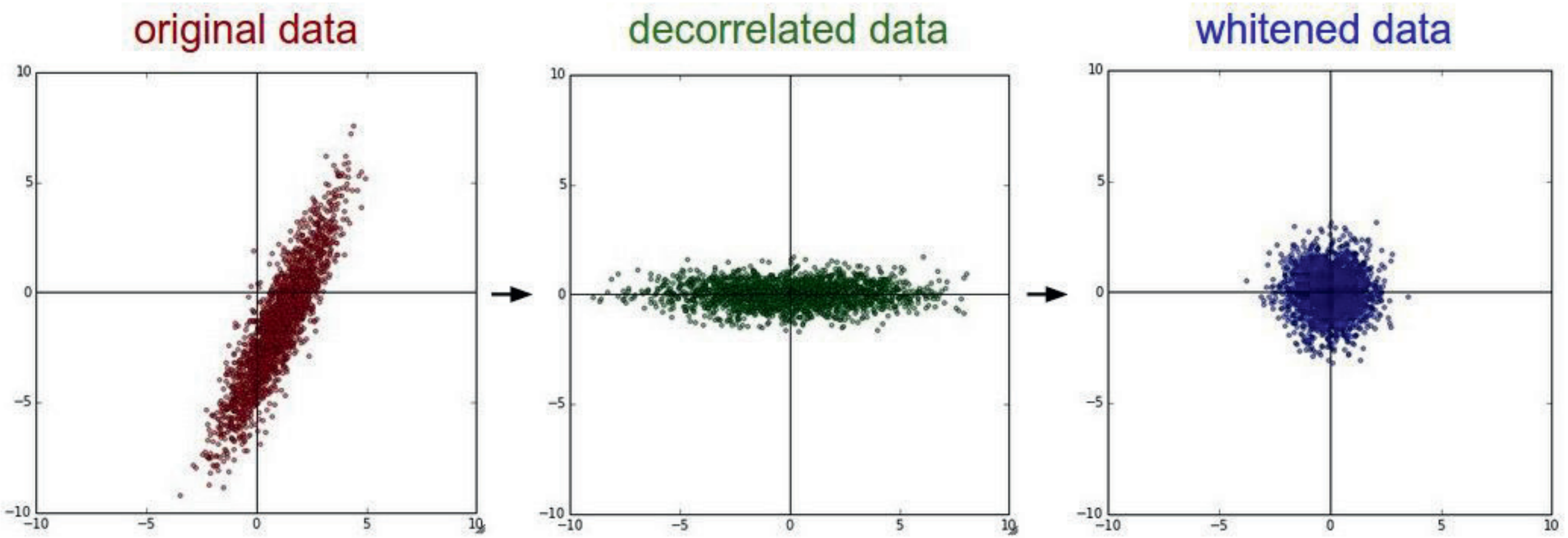

```
X -= np.mean(X, axis = 0)
```

```
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

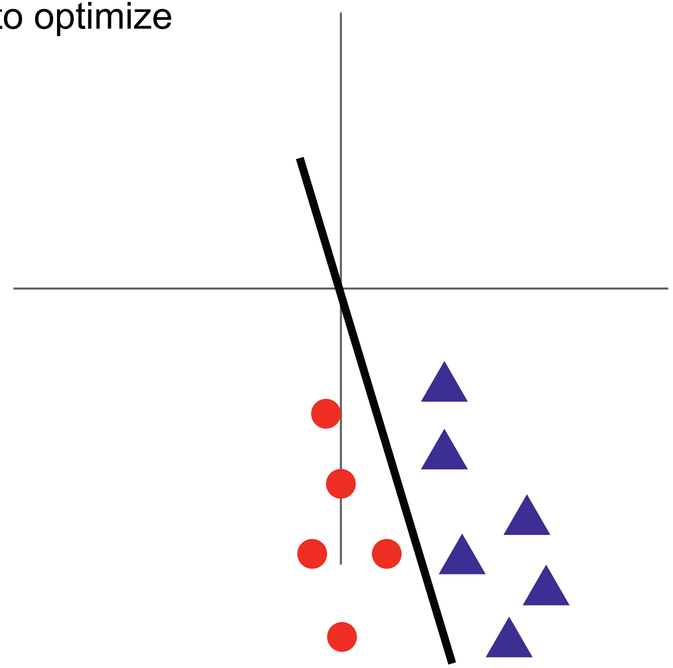



(data has diagonal covariance matrix)

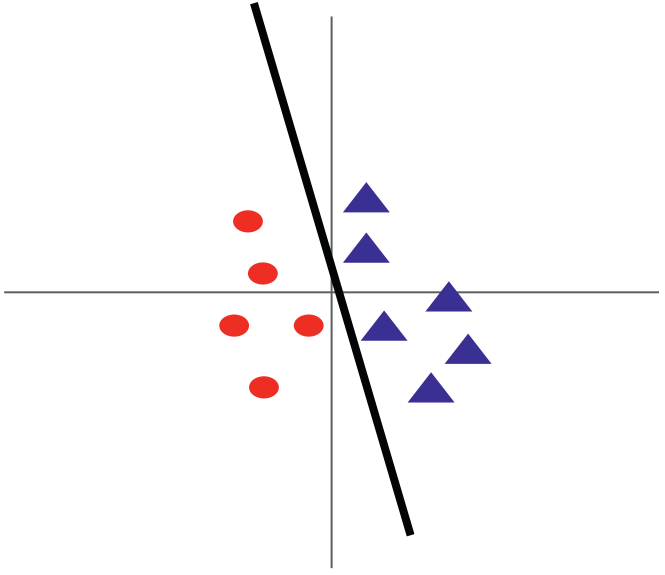
(covariance matrix is the identity matrix)

# 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



# **TLDR: In practice for Images:** center only

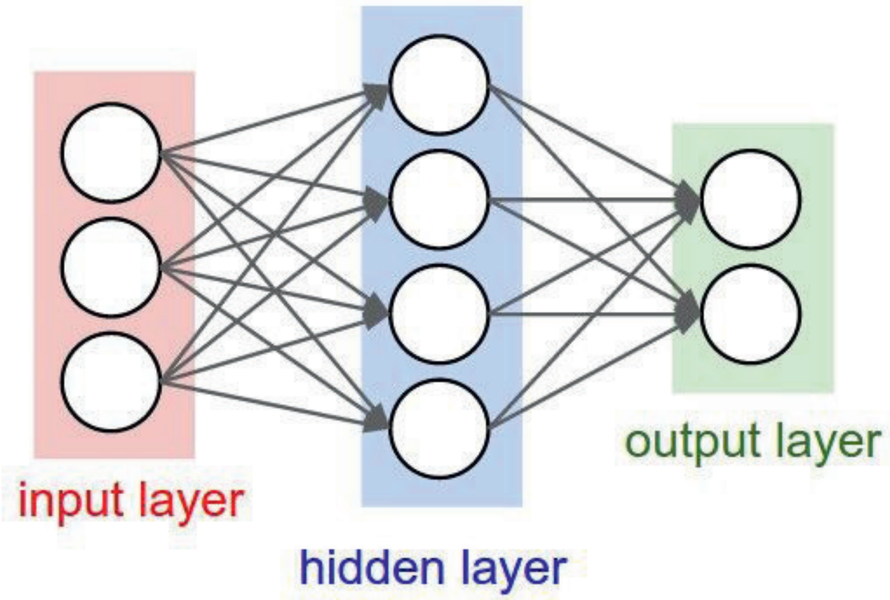
e.g. consider CIFAR-10 example with [32,32,3] images

- 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 when W=constant init is used?



- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

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

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

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

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

# Weight Initialization: Activation statistics

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

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

What will happen to the activations for the last layer?

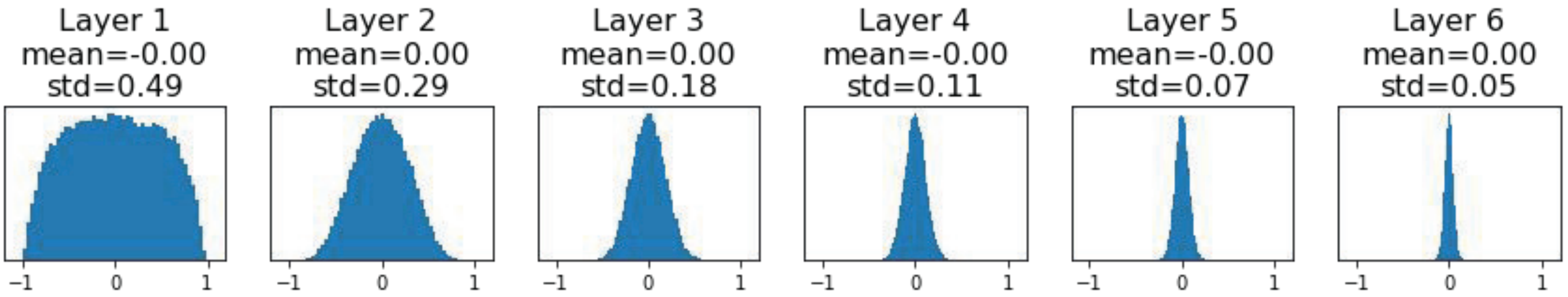
# Weight Initialization: Activation statistics

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

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

All activations tend to zero for deeper network layers

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



# Weight Initialization: Activation statistics

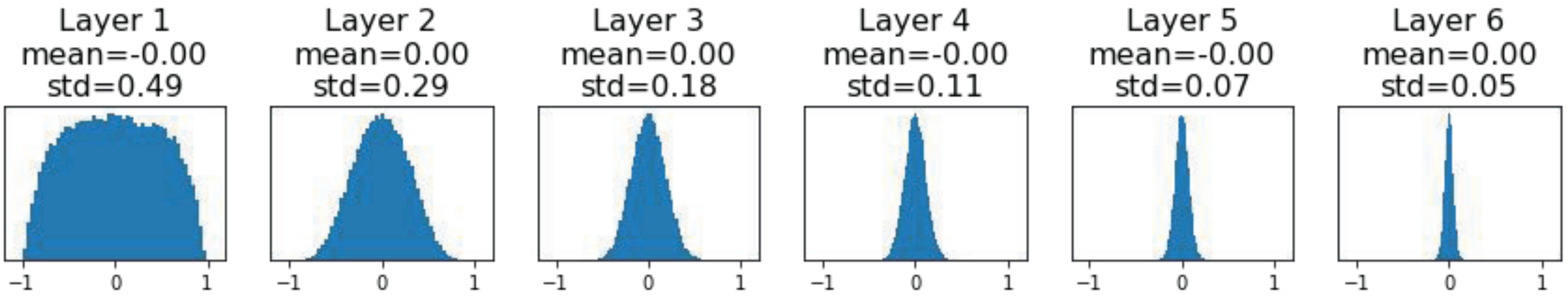
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

All activations tend to zero for deeper network layers

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

**A**: All zero, no learning =(



# Weight Initialization: Activation statistics

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

Increase std of initial weights from 0.01 to 0.05

What will happen to the activations for the last layer?

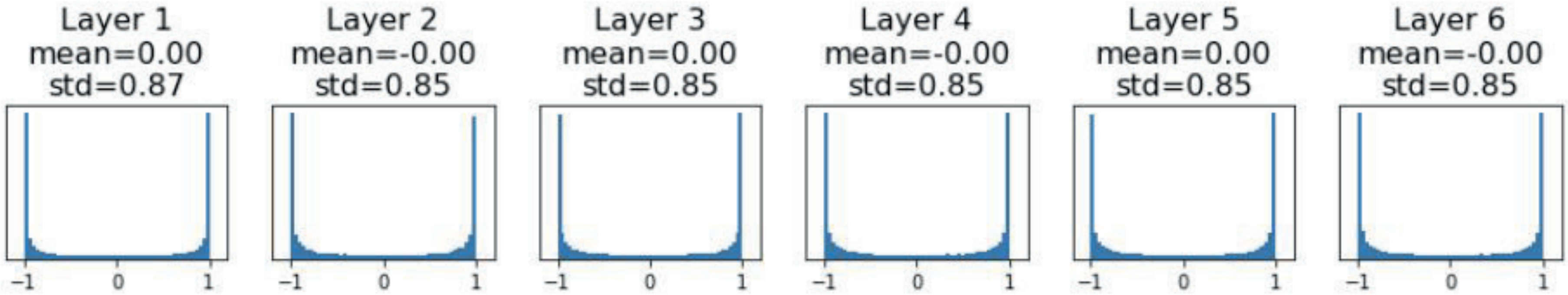
# Weight Initialization: Activation statistics

```
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

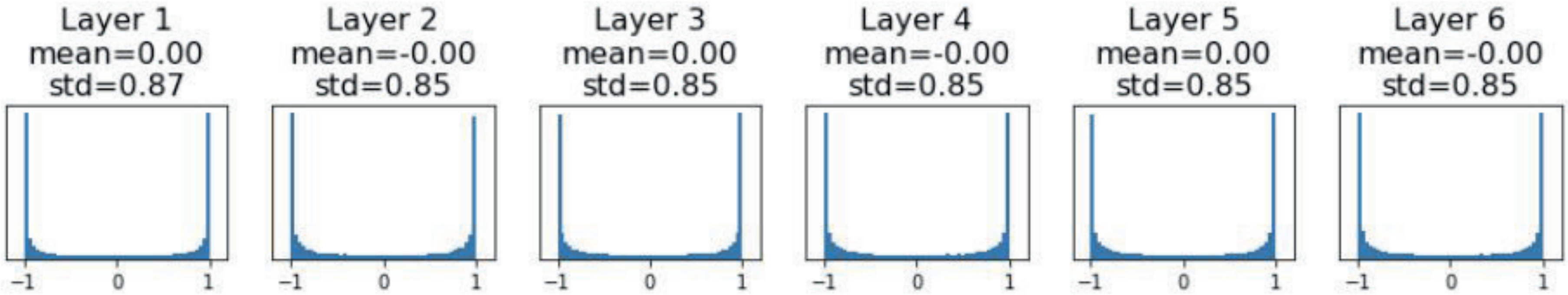
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



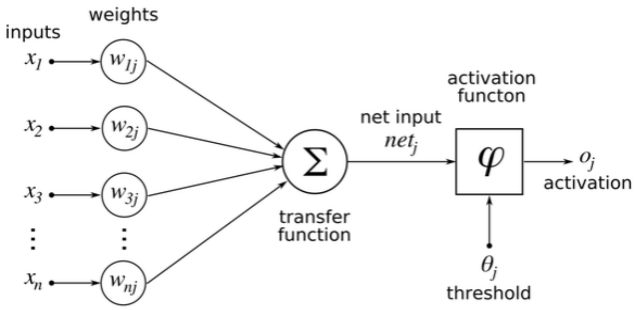
# Weight Initialization: “Xavier” Initialization

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

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

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

# Обоснование инициализации Ксавьера




$s = \sum_i^n w_i x_i$

$$\mathrm{Var}(s) = \mathrm{Var}\left(\sum_i^n w_i x_i\right)$$

$$= \sum_i^n \mathrm{Var}(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 \mathrm{Var}(x_i) + [E(x_i)]^2 \mathrm{Var}(w_i) + \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= \sum_i^n \mathrm{Var}(x_i)\mathrm{Var}(w_i)$$

$$= (n\mathrm{Var}(w))\,\mathrm{Var}(x)$$

$$\mathrm{Var}(aX) = a^2\mathrm{Var}(X)$$

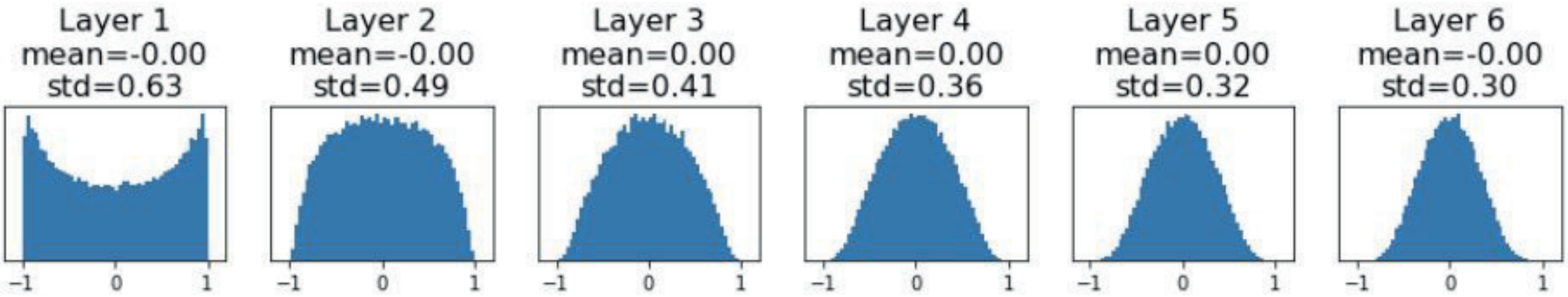
```
w = np.random.randn(n) / sqrt(n)
```

# Weight Initialization: "Xavier" Initialization

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

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

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




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

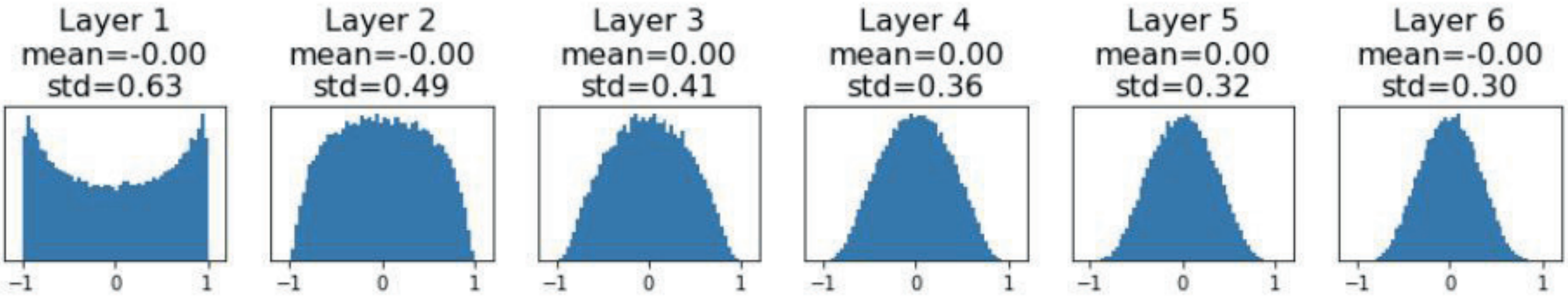
# Weight Initialization: “Xavier” Initialization

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

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

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

For conv layers, Din is filter_size$^2$ * input_channels




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

# Weight Initialization: “Xavier” Initialization

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

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

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

For conv layers, Din is $\text{filter\_size}^2 * \text{input\_channels}$

$y = Wx$
$h = f(y)$

**Derivation:**

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

[Assume x, w are iid]

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

# Weight Initialization: “Xavier” Initialization

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

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

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

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

**Derivation:**

$y = Wx$
$h = f(y)$

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

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

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

# Weight Initialization: “Xavier” Initialization

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

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

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

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

$y = Wx$
$h = f(y)$

**Derivation:**

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

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

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

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

# Weight Initialization: "Xavier" Initialization

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

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

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

For conv layers, Din is $kernel\_size^2$ * input_channels

$y = Wx$
$h = f(y)$

**Derivation:**

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

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

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

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

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

# Weight Initialization: What about ReLU?

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

Change from tanh to ReLU

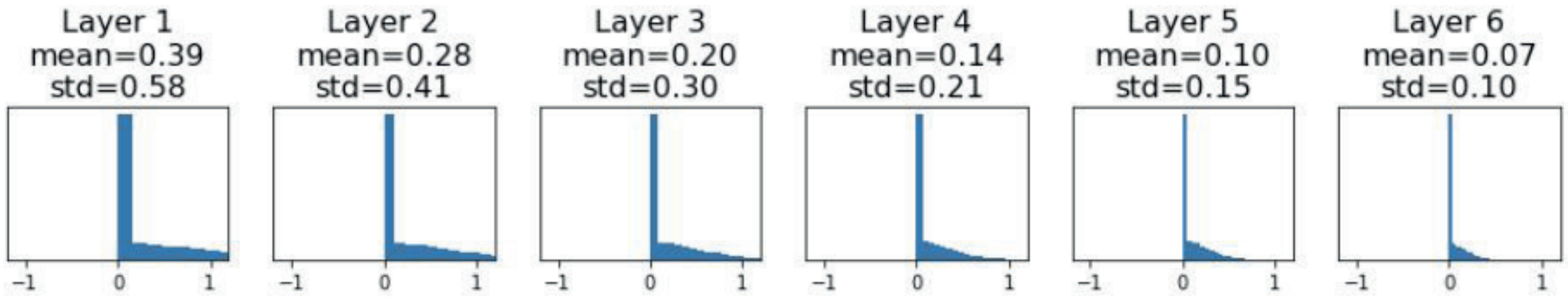
# Weight Initialization: What about ReLU?

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

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

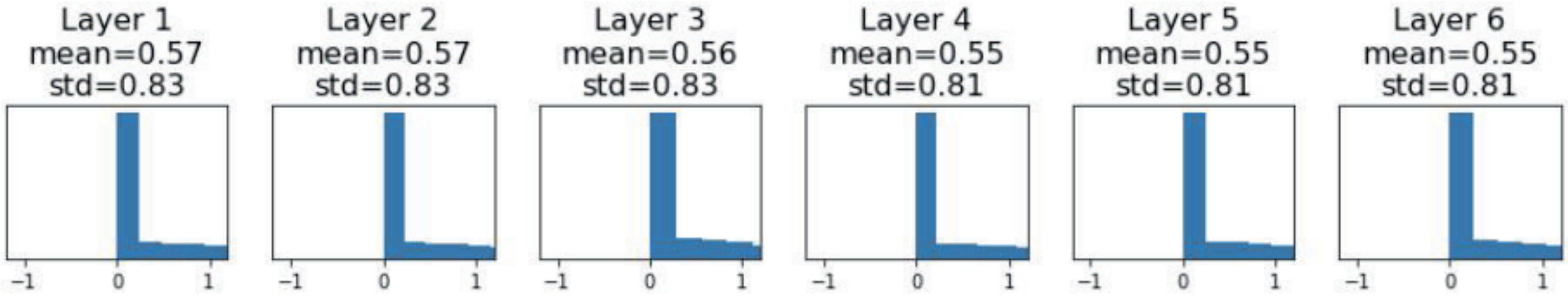


# Weight Initialization: Kaiming / MSRA Initialization

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

ReLU correction: std = sqrt(2 / Din)

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




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

# Proper initialization is an active area of research…

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

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

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

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

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

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

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

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

# Batch Normalization

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

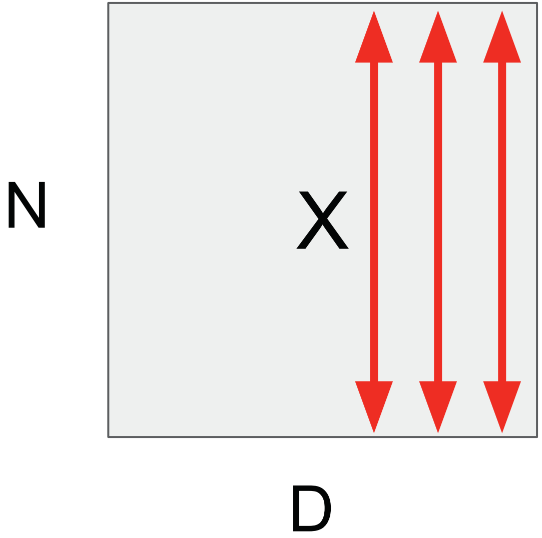
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** $x : N \times D$




$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

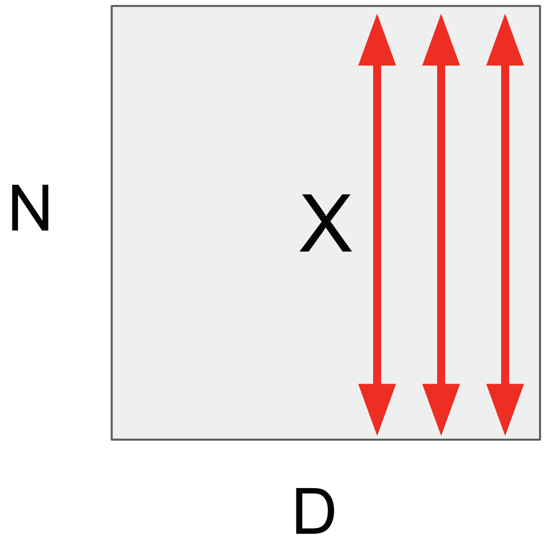
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** $x : N \times D$




$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

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

Normalized x, Shape is N x D

Problem: What if zero-mean, unit variance is too hard of a constraint?

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$\gamma, \beta : D$

Learning $\gamma = \sigma$,
$\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

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

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$\gamma, \beta : D$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

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

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization: Test-Time

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

During testing batchnorm becomes a linear operator! Can be fused with the previous fully-connected or conv layer

$$\mu_j =$$ (Running) average of values seen during training

Per-channel mean, shape is D

$$\sigma_j^2 =$$ (Running) average of values seen during training

Per-channel var, shape is D

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

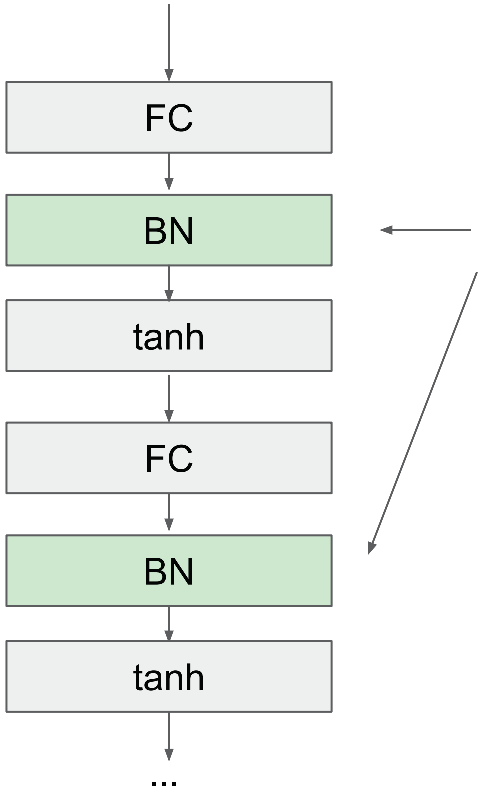
Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization

[Ioffe and Szegedy, 2015]

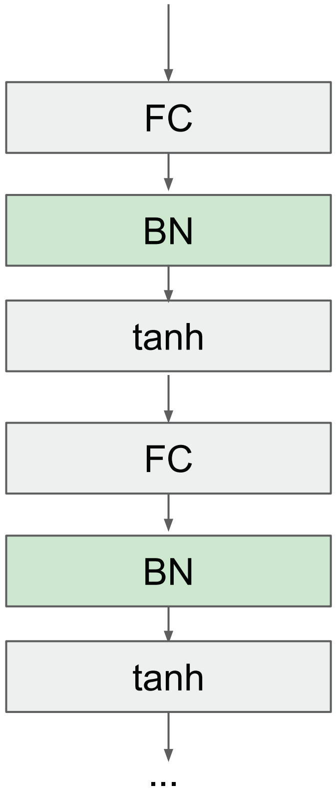



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

# Batch Normalization

[Ioffe and Szegedy, 2015]




- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

# Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

$\mathbf{x}: N \times D$

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

$\mathbf{x}: N \times C \times H \times W$

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

# Layer Normalization

**Batch Normalization** for fully-connected networks

$\mathbf{x}: N \times D$

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times D$

$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times D$

$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$

**Layer Normalization** for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$\mathbf{x}: N \times D$

Normalize

$\boldsymbol{\mu}, \boldsymbol{\sigma}: N \times 1$

$\boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times D$

$\mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta}$

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

# Instance Normalization

**Batch Normalization** for convolutional networks

$$\mathbf{x}: \ N \times C \times H \times W$$

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ 1 \times C \times 1 \times 1$$

$$\gamma, \beta: \ 1 \times C \times 1 \times 1$$

$$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$$

**Instance Normalization** for convolutional networks
Same behavior at train / test!

$$\mathbf{x}: \ N \times C \times H \times W$$

Normalize

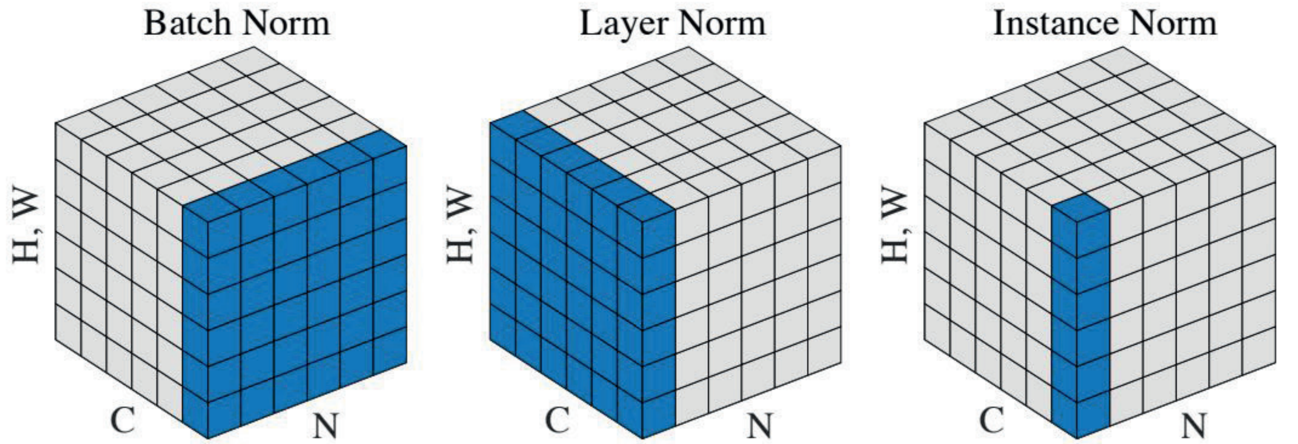
$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ N \times C \times 1 \times 1$$

$$\gamma, \beta: \ 1 \times C \times 1 \times 1$$

$$\mathbf{y} = \gamma(\mathbf{x}-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$$

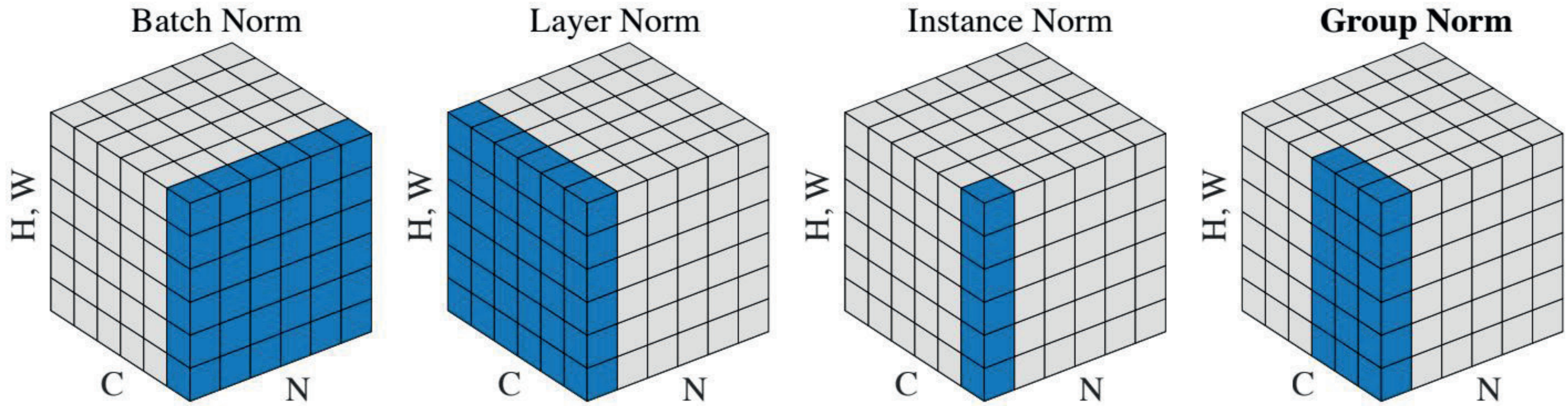
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

# Comparison of Normalization Layers




Wu and He, “Group Normalization”, ECCV 2018

# Group Normalization




Wu and He, “Group Normalization”, ECCV 2018

# Transfer learning

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

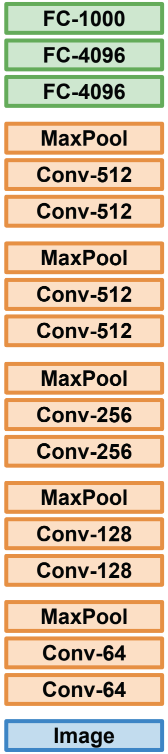
"You need a lot of a data if you want to train/use CNNs"

BUSTED

# Transfer Learning with CNNs

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

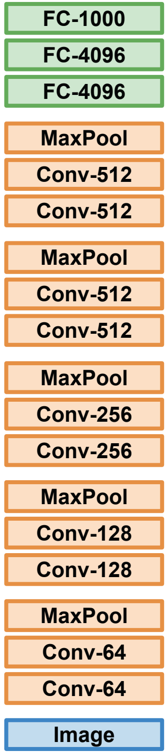
1. Train on Imagenet



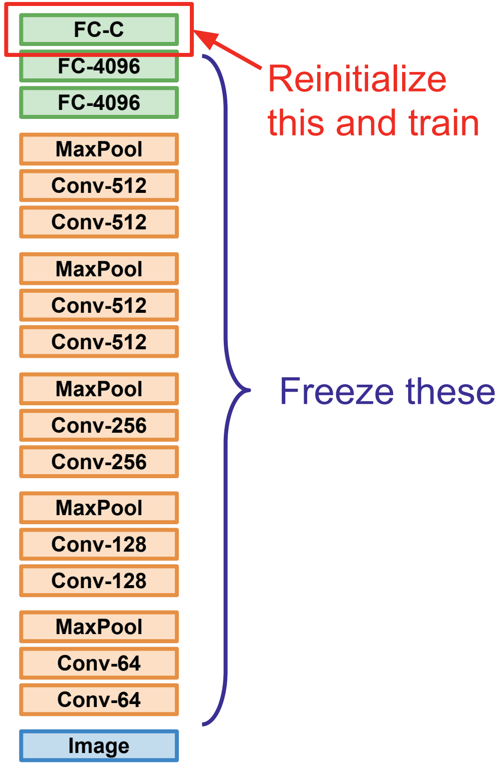
# Transfer Learning with CNNs

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

1. Train on Imagenet



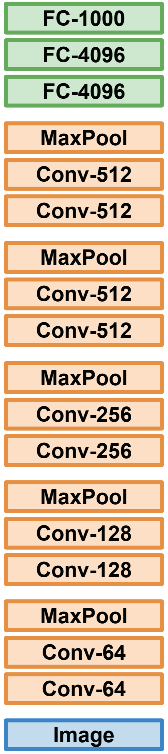

2. Small Dataset (C classes)



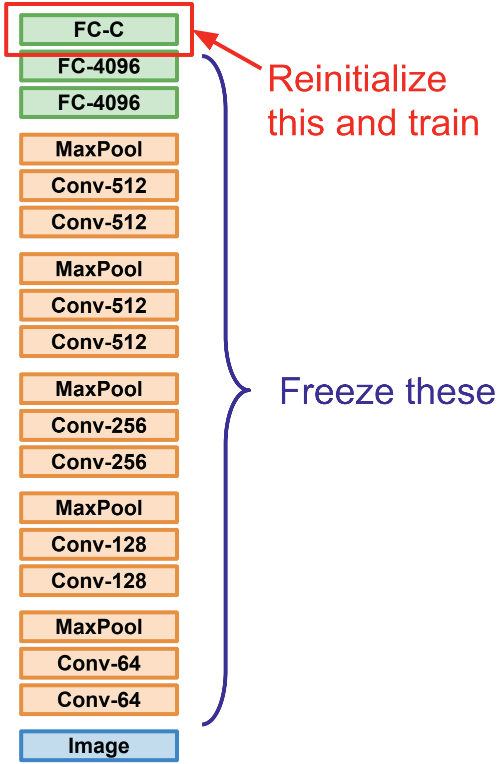
# Transfer Learning with CNNs

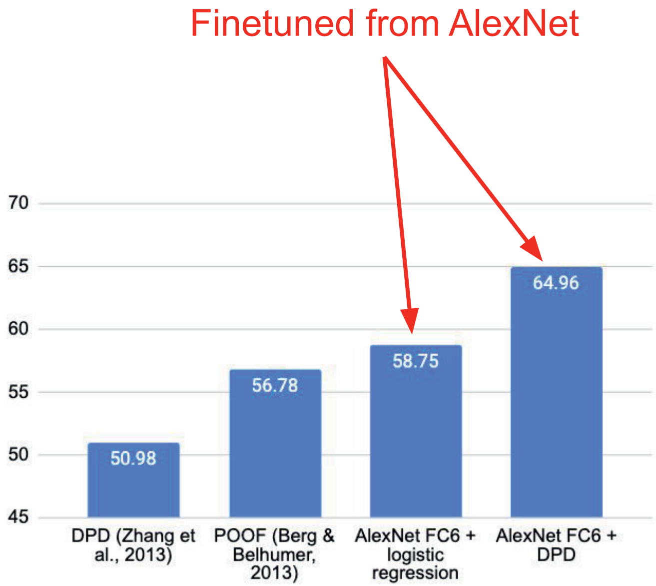
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet




2. Small Dataset (C classes)



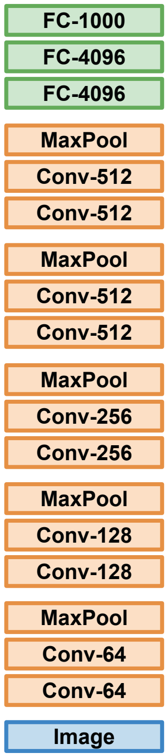




Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

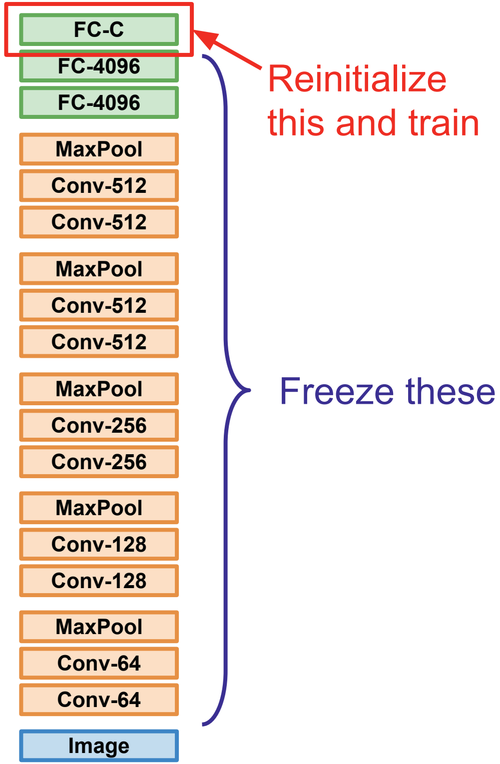
# Transfer Learning with CNNs

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

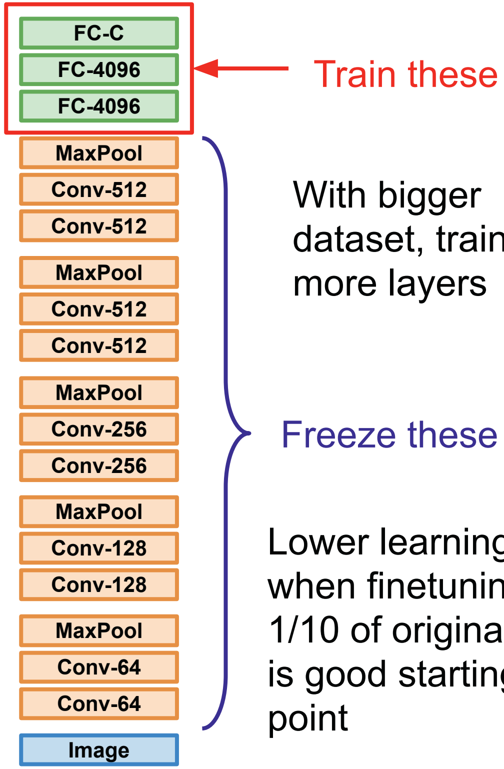
## 1. Train on Imagenet




## 2. Small Dataset (C classes)

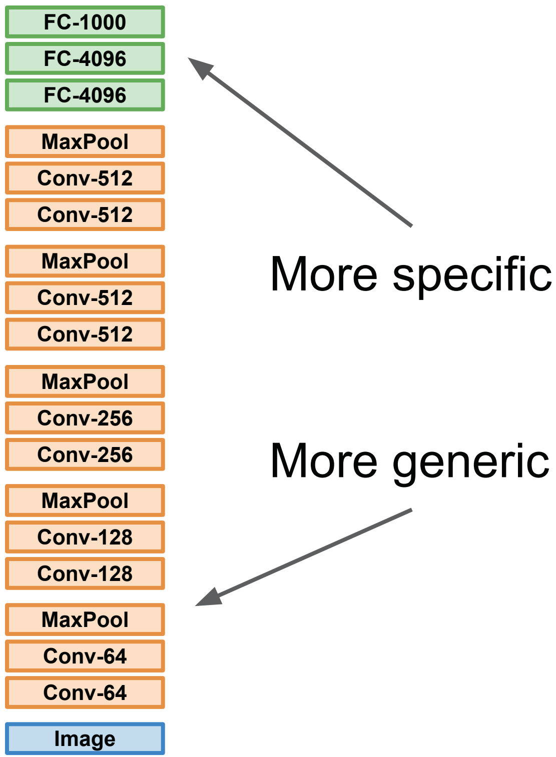



## 3. Bigger dataset

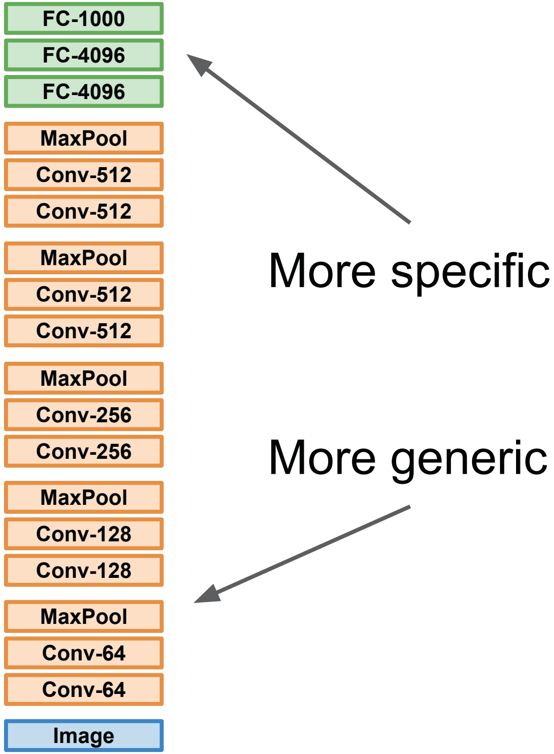



With bigger dataset, train more layers

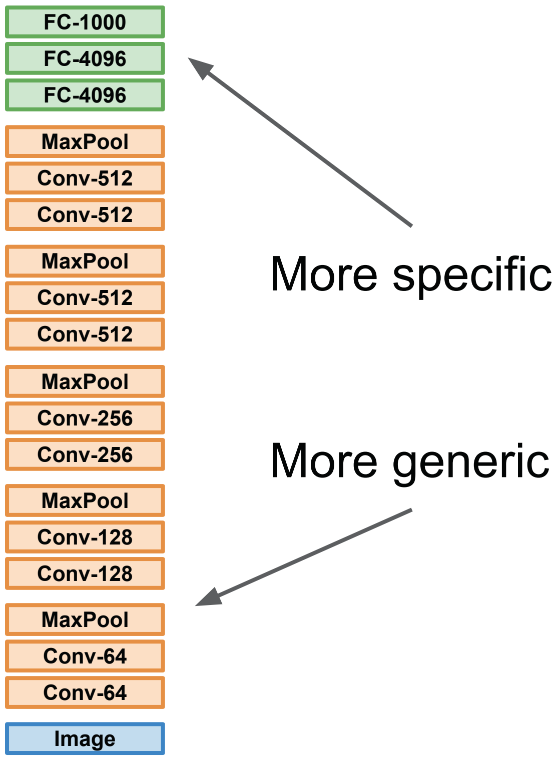
Lower learning rate when finetuning; 1/10 of original LR is good starting point




| | very similar dataset | very different dataset |
|---|---|---|
| **very little data** | ? | ? |
| **quite a lot of data** | ? | ? |




| | **very similar dataset** | **very different dataset** |
|---|---|---|
| **very little data** | Use Linear Classifier on top layer | ? |
| **quite a lot of data** | Finetune a few layers | ? |




| | **very similar dataset** | **very different dataset** |
|---|---|---|
| **very little data** | Use Linear Classifier on top layer | You're in trouble… Try linear classifier from different stages |
| **quite a lot of data** | Finetune a few layers | Finetune a larger number of layers |

# Transfer learning with CNNs is pervasive…
## (it's the norm, not an exception)

Object Detection
(Fast R-CNN)

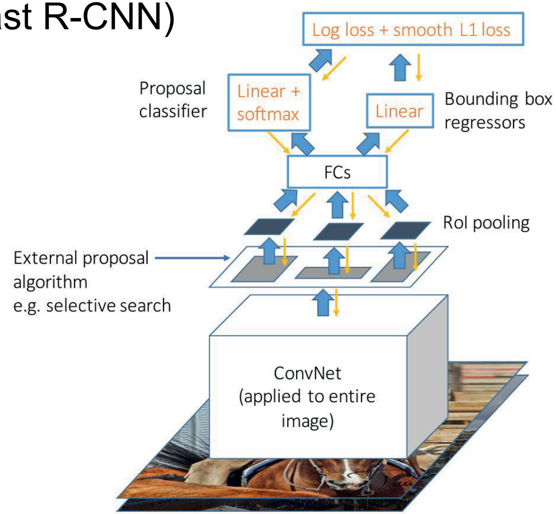
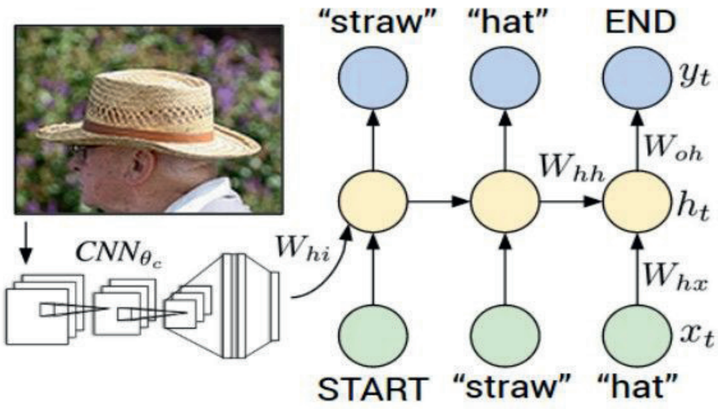



Image Captioning: CNN + RNN

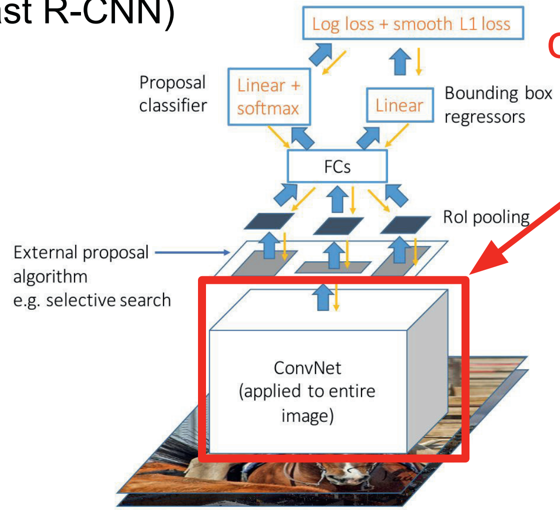



Girshick, "Fast R-CNN", ICCV 2015


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

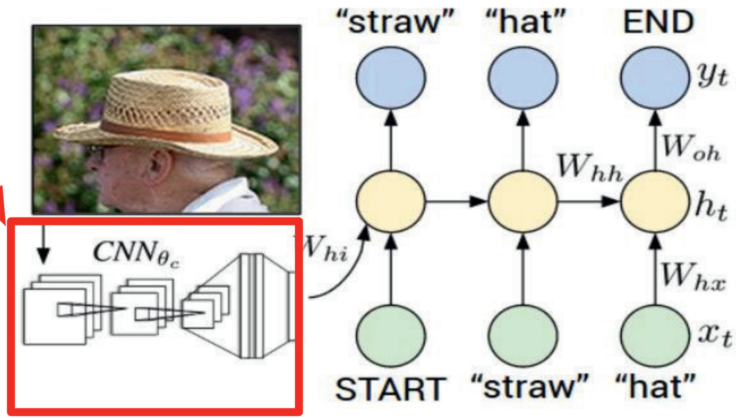
# Transfer learning with CNNs is pervasive…
## (it's the norm, not an exception)

Object Detection
(Fast R-CNN)




CNN pretrained on ImageNet

Image Captioning: CNN + RNN

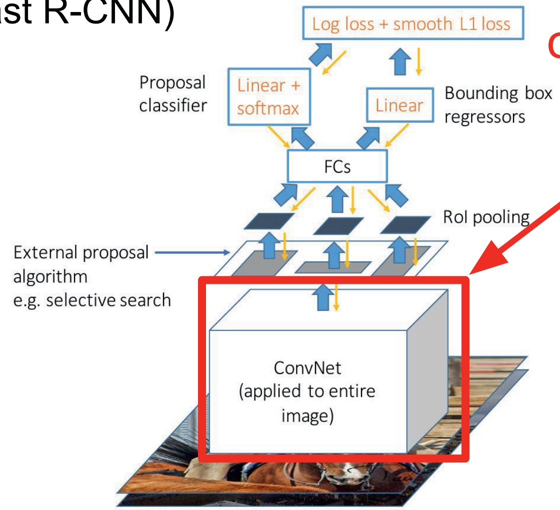



Girshick, "Fast R-CNN", ICCV 2015


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

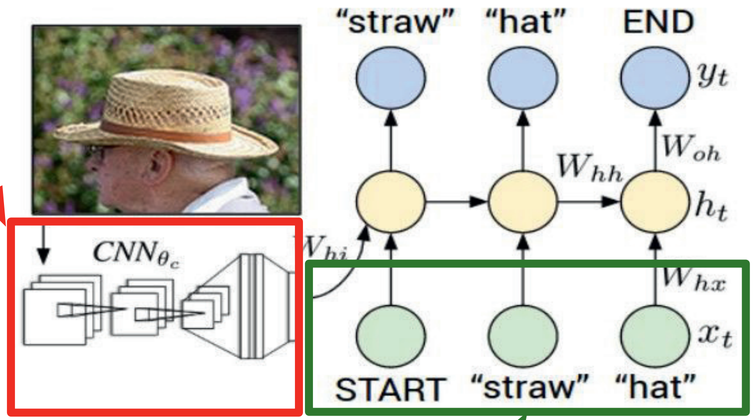
# Transfer learning with CNNs is pervasive…
## (it’s the norm, not an exception)

Object Detection
(Fast R-CNN)




CNN pretrained on ImageNet

Image Captioning: CNN + RNN

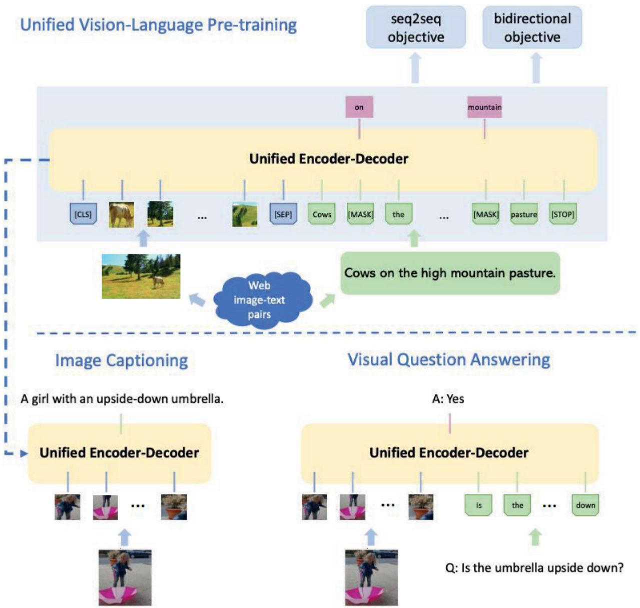



Word vectors pretrained with word2vec

Girshick, “Fast R-CNN”, ICCV 2015


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

# Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

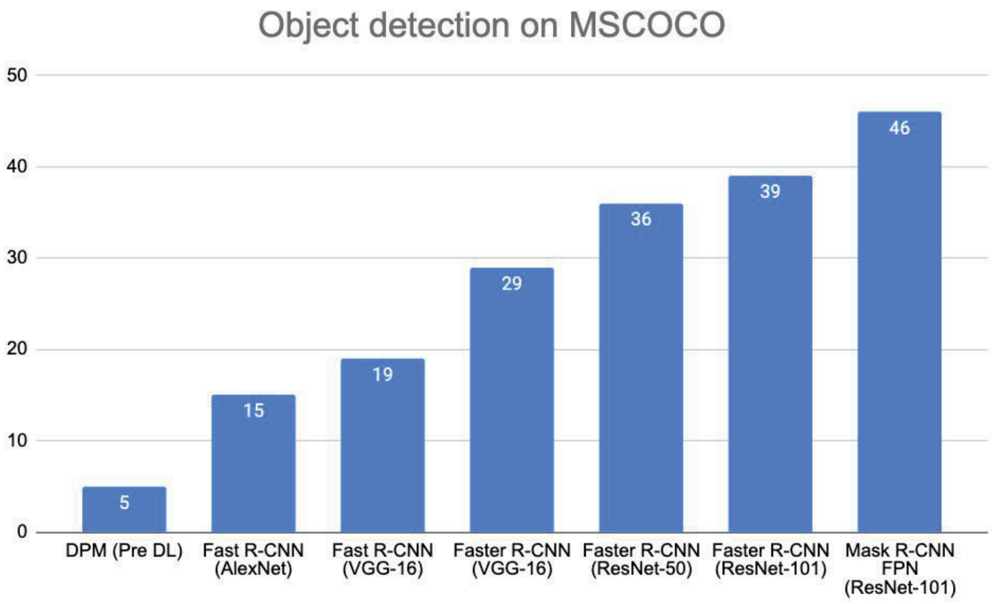



1. Train CNN on ImageNet
2. Fine-Tune (1) for object detection on Visual Genome
3. Train BERT language model on lots of text
4. Combine(2) and (3), train for joint image / language modeling
5. Fine-tune (4) for image captioning, visual question answering, etc.

Zhou et al, “Unified Vision-Language Pre-Training for Image Captioning and VQA” CVPR 2020
Figure copyright Luowei Zhou, 2020. Reproduced with permission.

Krishna et al, “Visual genome: Connecting language and vision using crowdsourced dense image annotations” IJCV 2017
Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding” ArXiv 2018

# Transfer learning with CNNs - Architecture matters

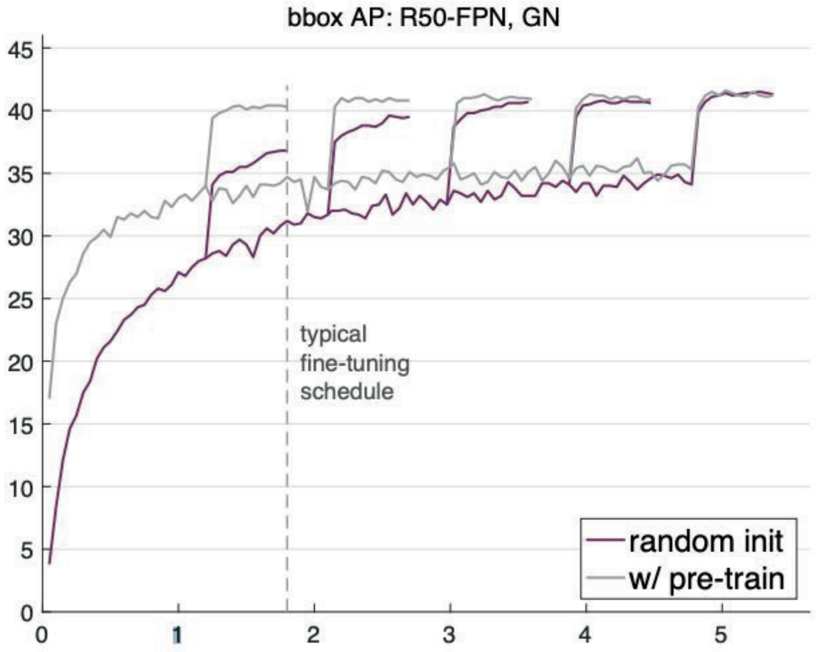



Girshick, "The Generalized R-CNN Framework for Object Detection", ICCV 2017 Tutorial on Instance-Level Visual Recognition

We will discuss different architectures in detail in two lectures

# Transfer learning with CNNs is pervasive…

## But recent results show it might not always be necessary!




He et al, “Rethinking ImageNet Pre-training”, ICCV 2019


Training from scratch can work just as well as training from a pretrained ImageNet model for object detection

But it takes 2-3x as long to train.

They also find that collecting more data is better than finetuning on a related task

# Takeaway for your projects and beyond:

Transfer learning be like


Source: AI & Deep Learning Memes For Back-propagated Poets

# Takeaway for your projects and beyond:

Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision

# Summary

TLDRs

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use this!)
- Transfer learning (use this if you can!)

# Next time:
# Training Neural Networks, Part 2

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Babysitting learning
- Evaluation (Ensembles etc.)
- Hyperparameter Optimization
- Transfer learning / fine-tuning