Lecture 7:
Training Neural Networks, Part I

Administrative: Project Proposal

Due yesterday, 4/19 on GradeScope

1 person per group needs to submit, but tag all group members

Personal announcement

Ranjay:
- I am defending my PhD on **Friday, April 23rd at 1pm PST**.
- Stanford CS defenses are public events.
- Join [CS 547](#)'s seminar if you want to watch it.
- If you are unable to find the zoom link to watch it and want to, send me an email by Thursday 3pm.

Administrative: A2

A2 is out, due Wednesday April 30th, 11:59pm

Where we are now...

Computational graphs

$$f = Wx \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

x

W

*

s (scores)

hinge loss

+

L

R

$$R(W)$$
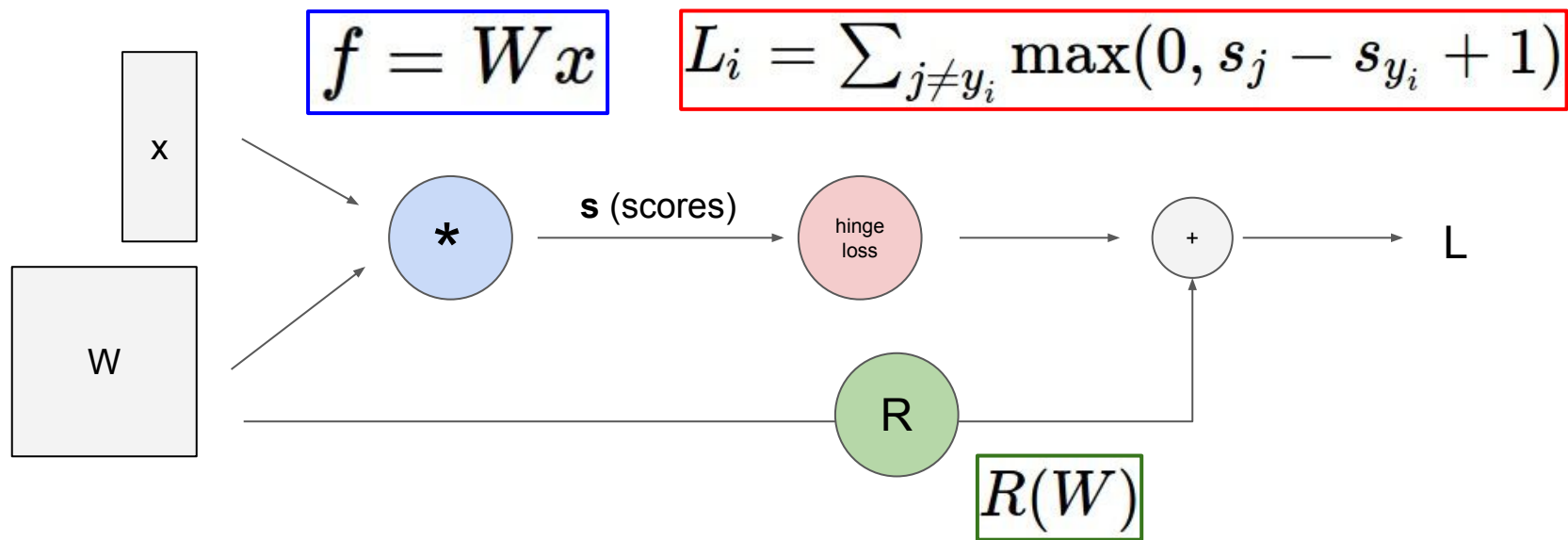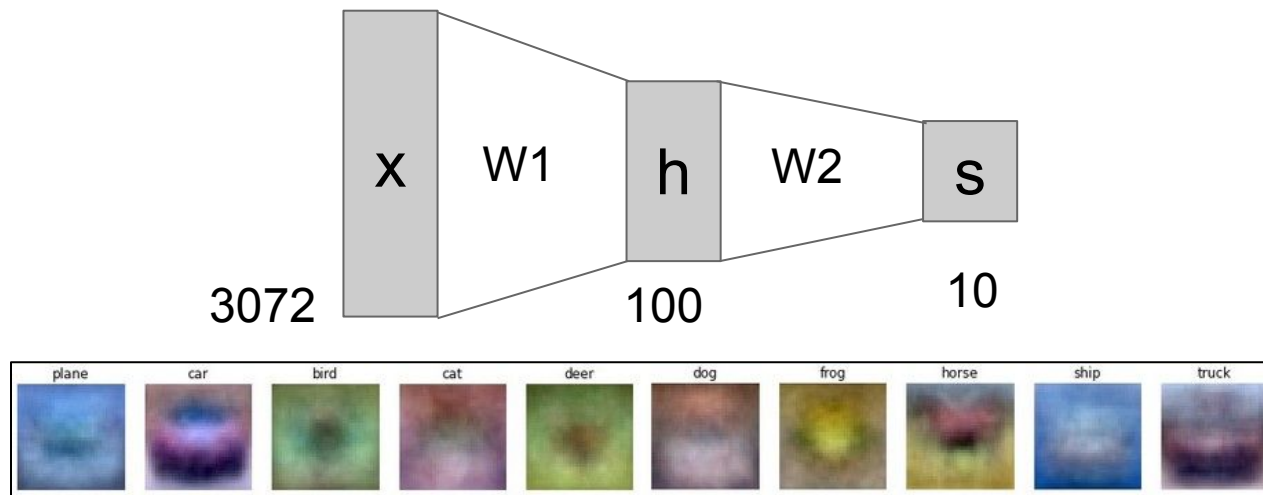
Where we are now...

Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$
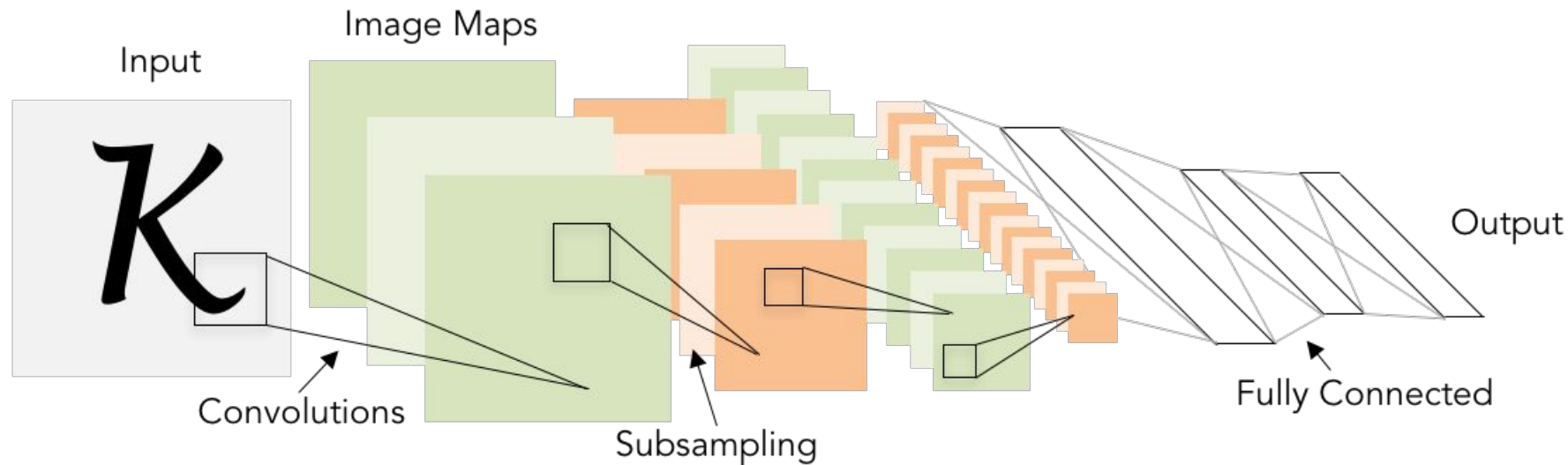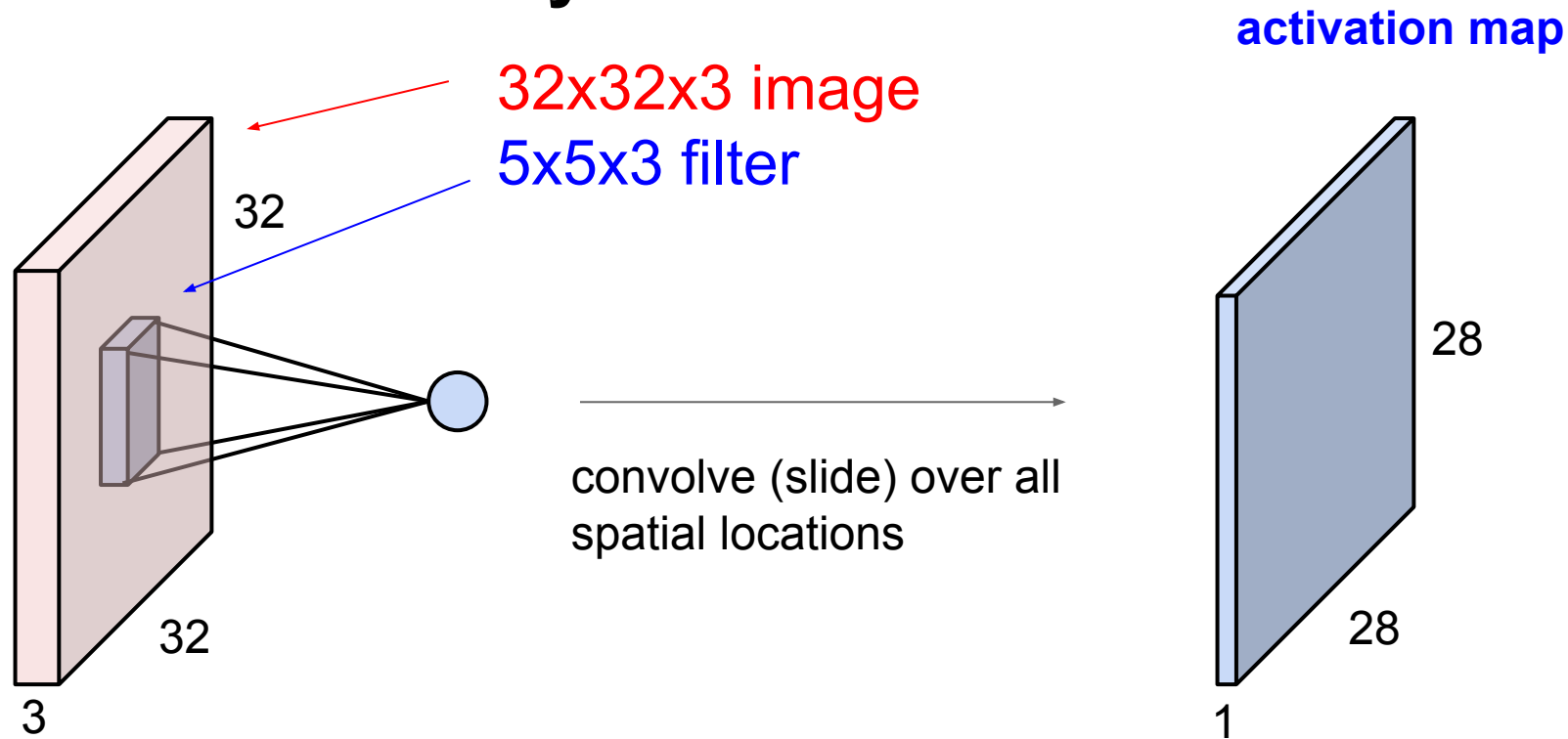
Where we are now...

Convolutional Neural Networks

Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1
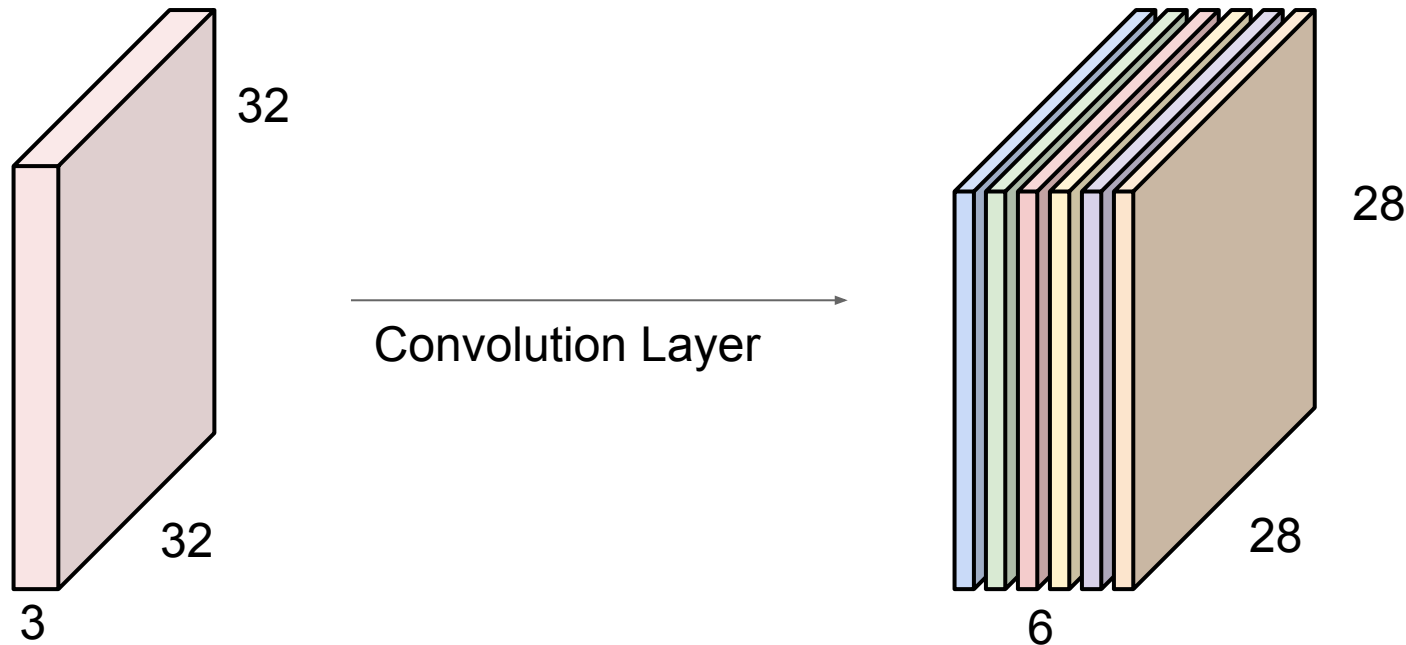
Where we are now...

Convolutional Layer

activation map

32x32x3 image
5x5x3 filter

32

32

3

convolve (slide) over all
spatial locations

28

28

1

Where we are now...
Convolutional Layer

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

activation maps

Convolution Layer

32

32

3

28

28

6

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

Where we are now...

Learning network parameters through optimization
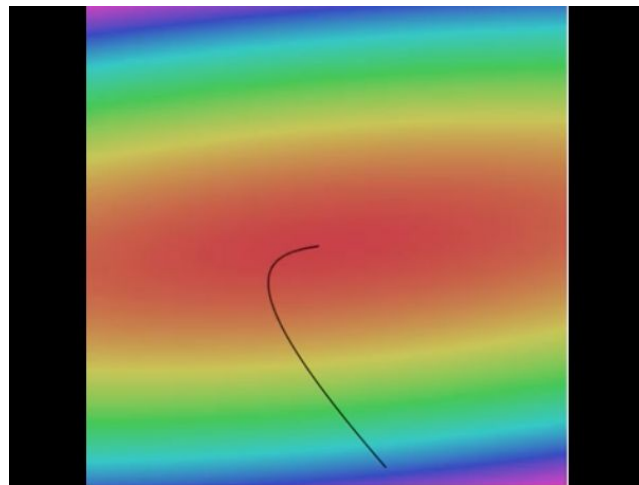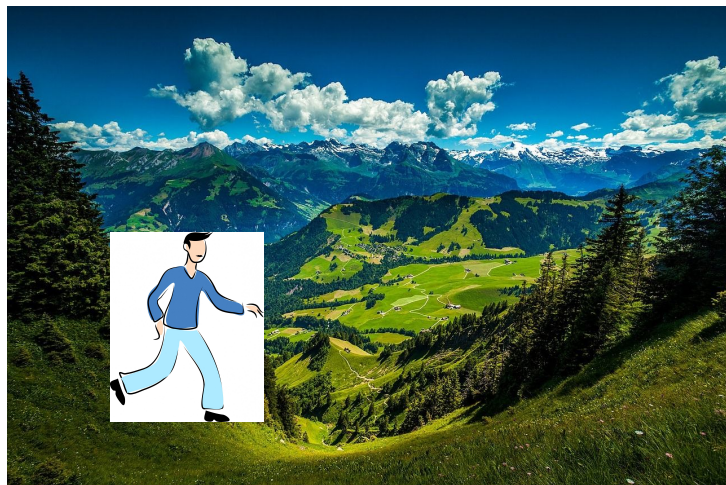

```
# Vanilla Gradient Descent

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

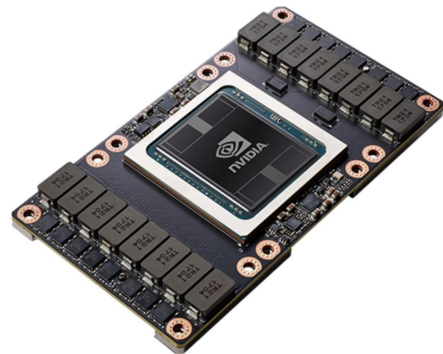Where we are now...

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

Today: Training Neural Networks

Overview

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

Part 1

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

Activation Functions

Activation Functions
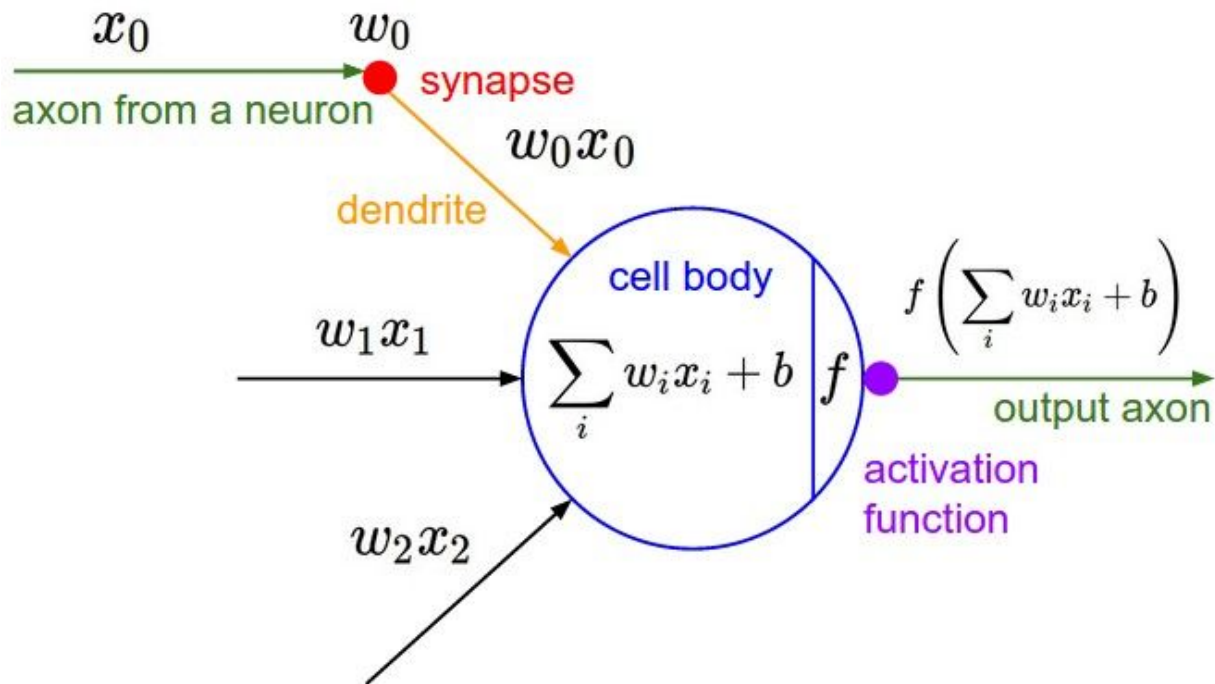
Activation Functions

Sigmoid

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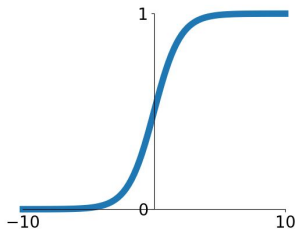
tanh

$\tanh(x)$
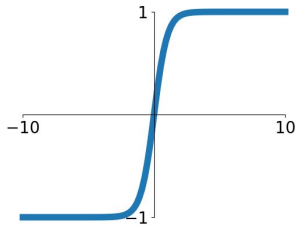
ReLU

$\max(0, x)$
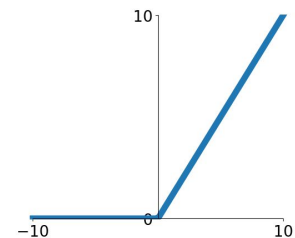
Leaky ReLU

$\max(0.1x, x)$
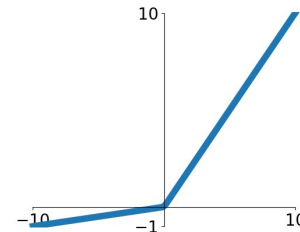
Maxout

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

ELU

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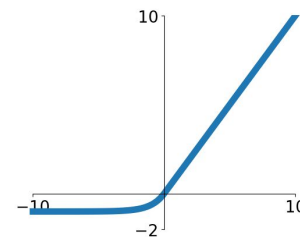
Activation Functions

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

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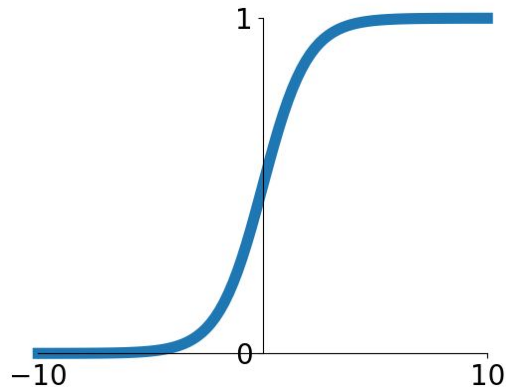
Sigmoid

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

x

$\sigma(x) = 1/(1 + e^{-x})$
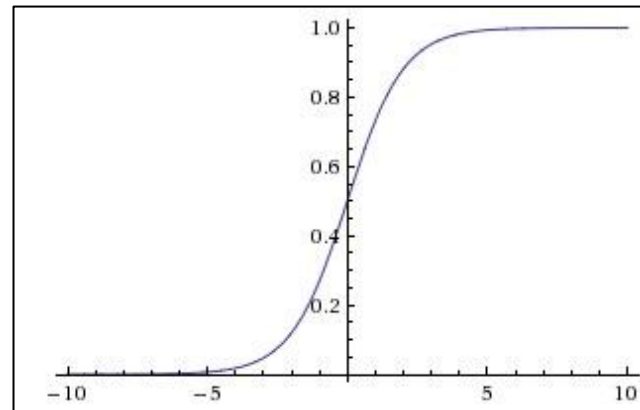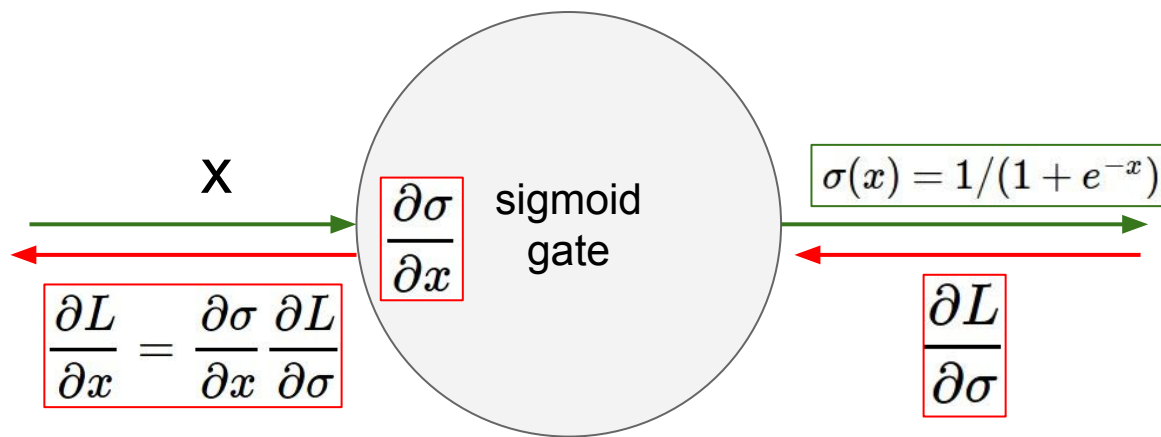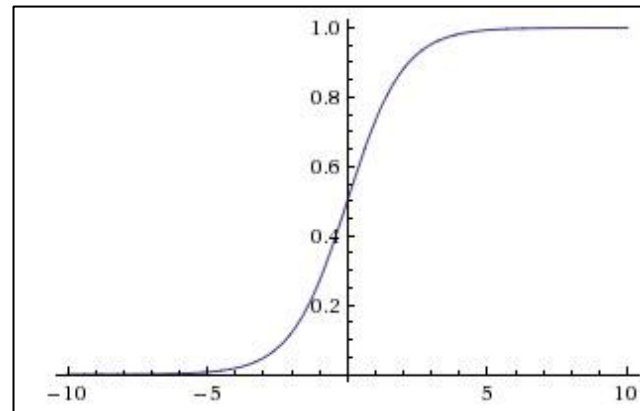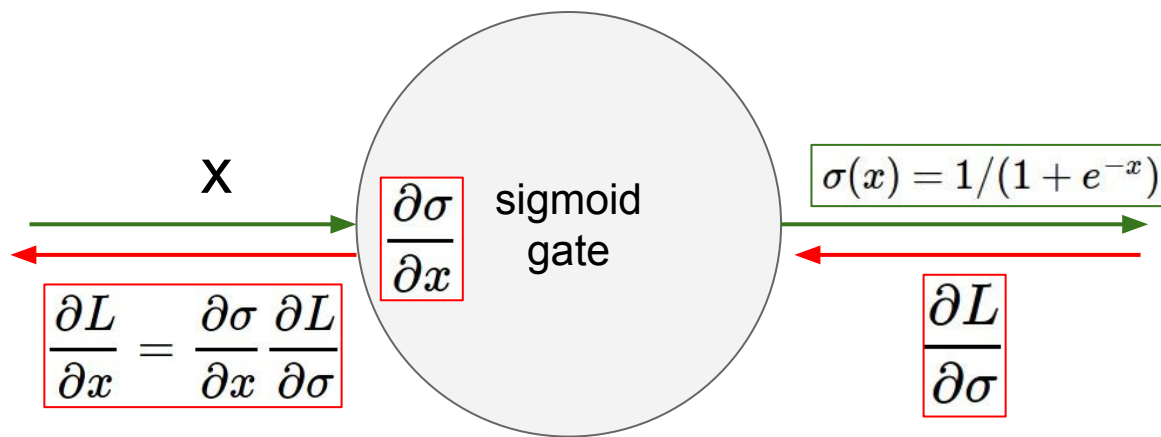
$\dfrac{\partial \sigma}{\partial x}$ sigmoid gate

$\dfrac{\partial L}{\partial x} = \dfrac{\partial \sigma}{\partial x} \dfrac{\partial L}{\partial \sigma}$

$\dfrac{\partial L}{\partial \sigma}$

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

x

$$\frac{\partial \sigma}{\partial x}$$ sigmoid gate

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

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial \sigma}$$

What happens when x = -10?

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

x

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

$\dfrac{\partial \sigma}{\partial x}$ sigmoid gate

$\dfrac{\partial L}{\partial x} = \dfrac{\partial \sigma}{\partial x}\dfrac{\partial L}{\partial \sigma}$

$\dfrac{\partial L}{\partial \sigma}$

What happens when x = -10?

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

$\sigma(x) = \sim 0$

$\dfrac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,) = 0(1 - 0) = 0$

x

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

sigmoid gate

$$\frac{\partial \sigma}{\partial x}$$

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial \sigma}$$

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

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

x

$$\frac{\partial \sigma}{\partial x}$$ sigmoid gate

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

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x}\frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial \sigma}$$

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

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

$$\sigma(x) = \sim 1 \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)) = 1(1 - 1) = 0$$

x

sigmoid gate

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

$\frac{\partial \sigma}{\partial x}$

$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x}$

$\frac{\partial L}{\partial \sigma}$

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

What ha
What ha
What ha

x

$$\frac{\partial \sigma}{\partial x}$$

sigmoid gate

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

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x}\frac{\partial L}{\partial \sigma}$$

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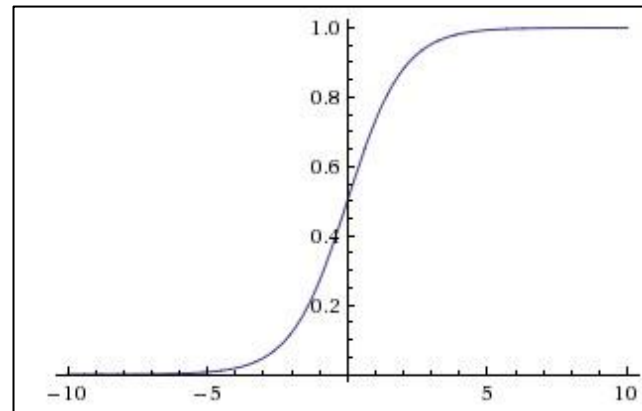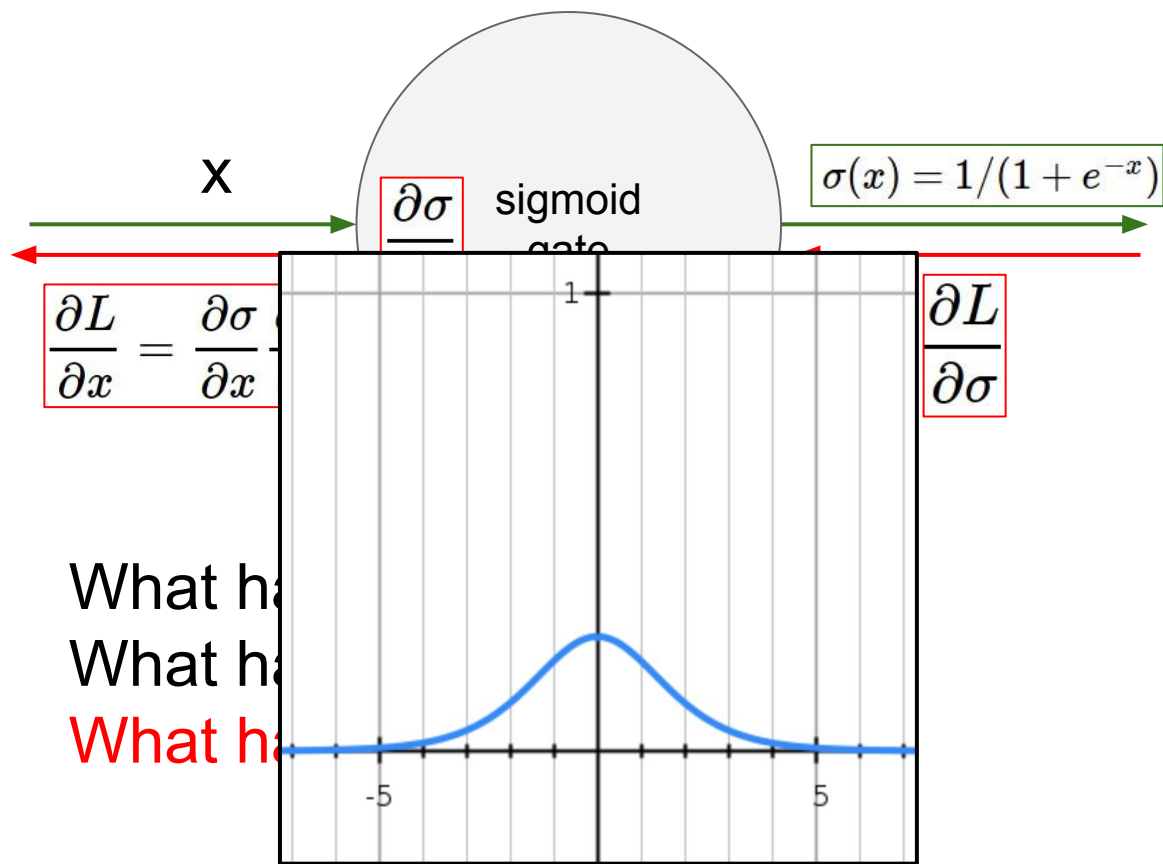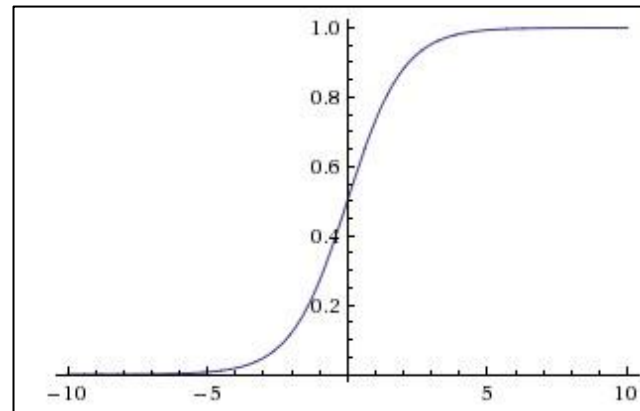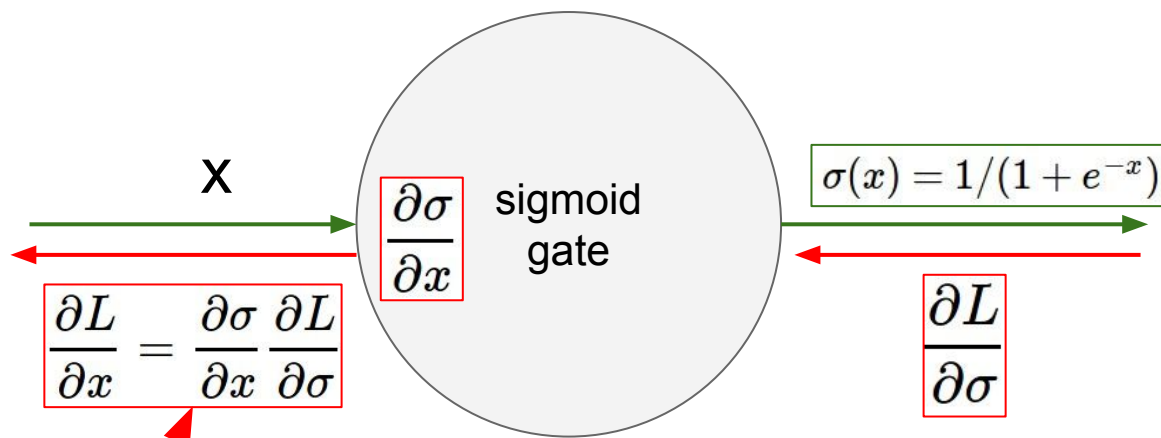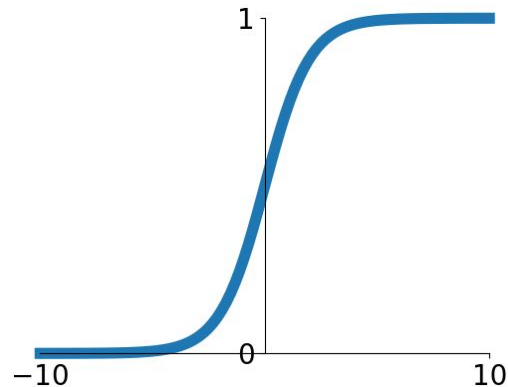
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)\,)$$

Why is this a problem?

If all the gradients flowing back will be zero and weights will never change

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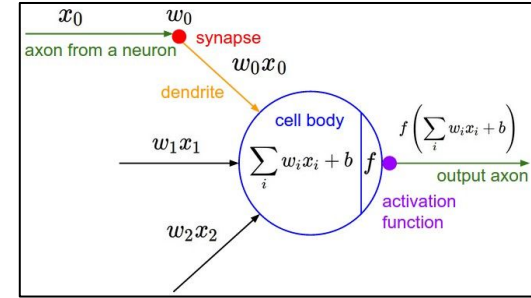
$$f\left(\sum_i w_i x_i + b\right)$$

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

Consider what happens when the input to a neuron is always positive...
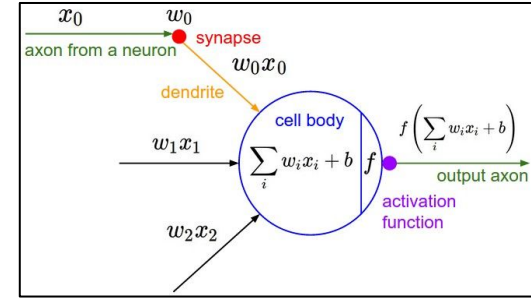
$$f\left(\sum_i w_i x_i + b\right)$$

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

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

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

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

Consider what happens when the input to a neuron is always positive...
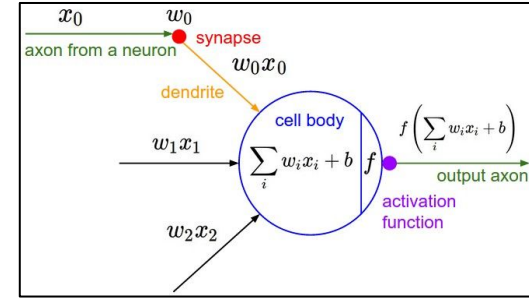
$$f\left(\sum_i w_i x_i + b\right)$$

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!

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

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

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

allowed gradient update directions

zig zag path

allowed gradient update directions

hypothetical optimal w vector
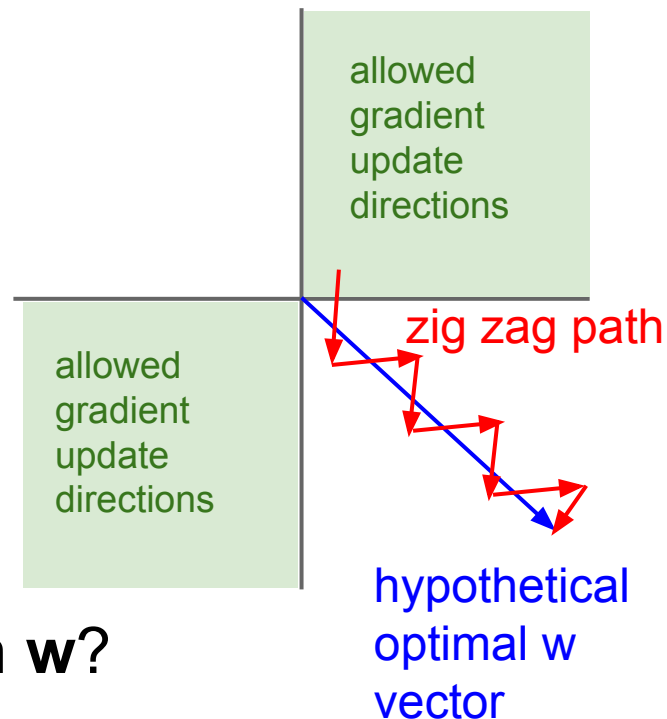
What can we say about the gradients on **w**?
Always all positive or all negative :(

Consider what happens when the input to a neuron is always positive...
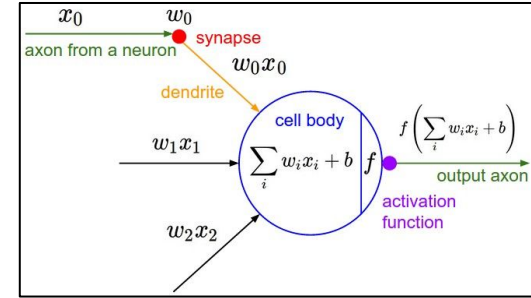
$$f\left(\sum_i w_i x_i + b\right)$$

allowed gradient update directions

zig zag path

allowed gradient update directions

hypothetical optimal w vector
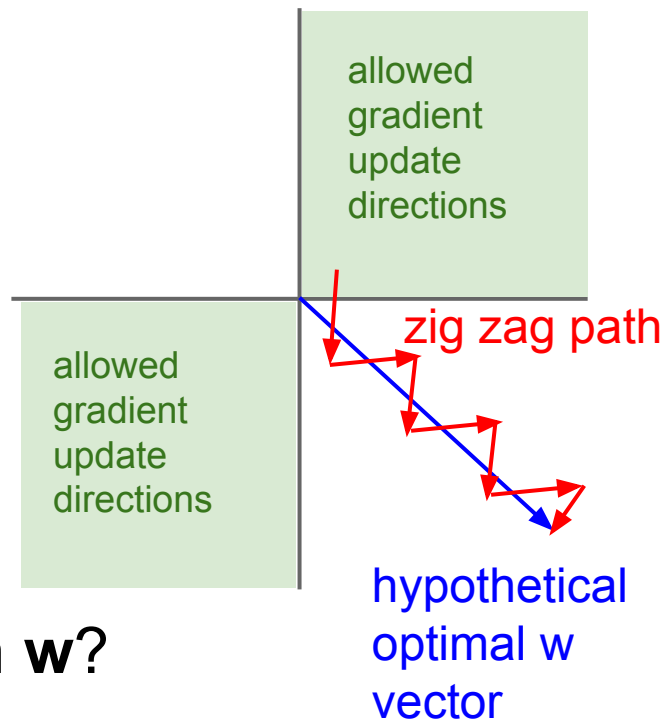
What can we say about the gradients on **w**?
Always all positive or all negative :(
(For a single element! Minibatches help)

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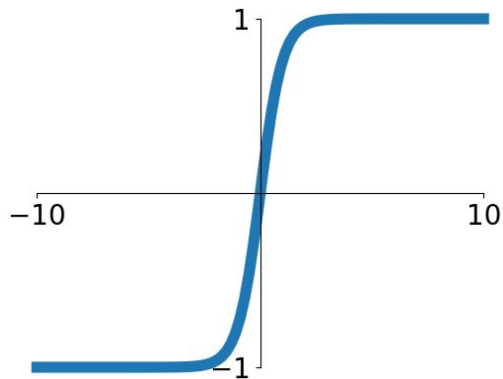
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

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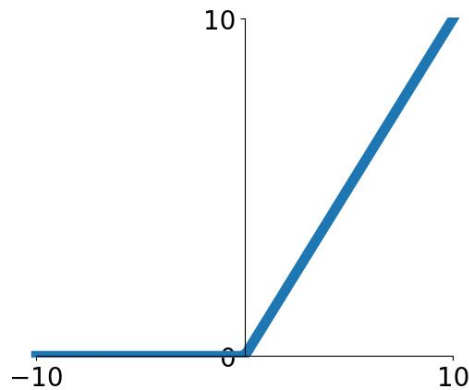
- Computes **f(x) = max(0,x)**

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

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]

Activation Functions
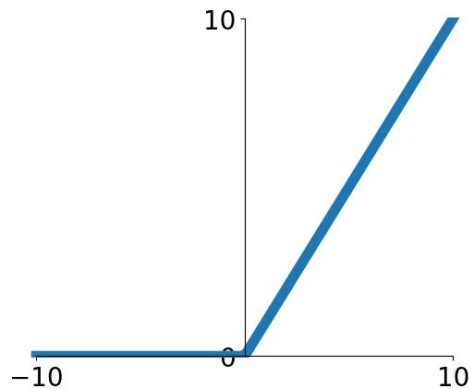
ReLU
(Rectified Linear Unit)

- 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
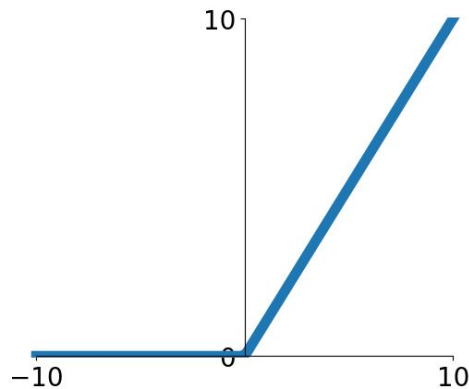
ReLU
(Rectified Linear Unit)

- 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?
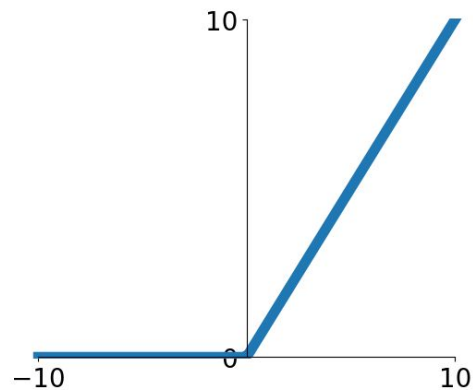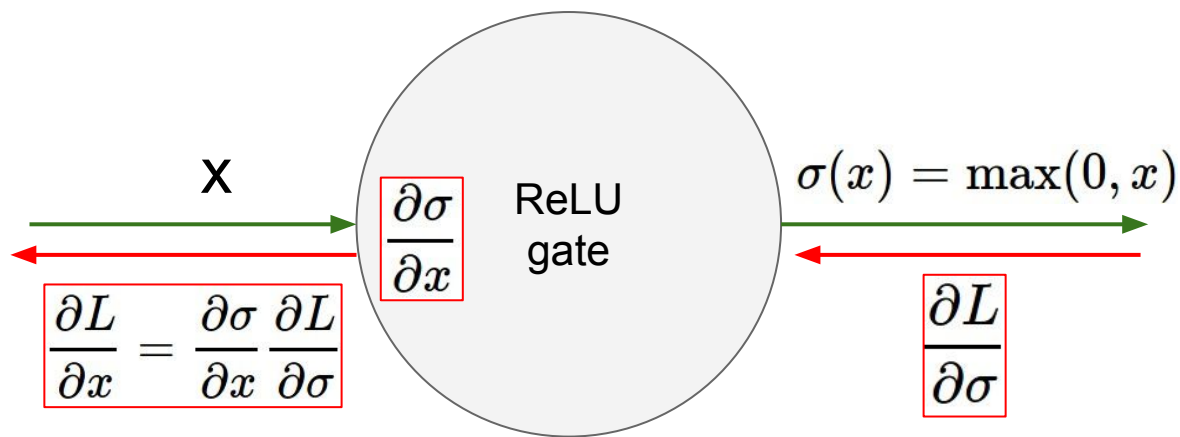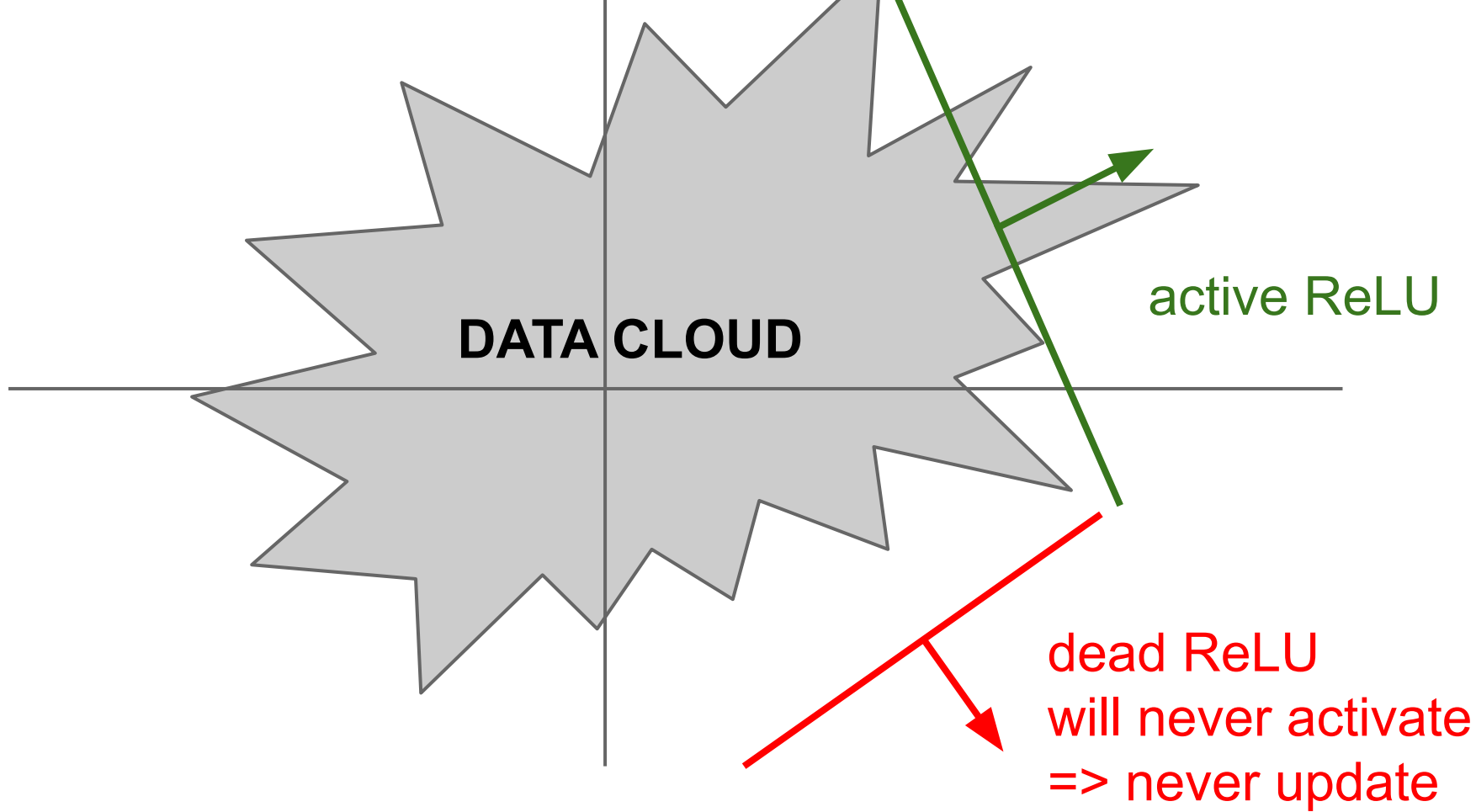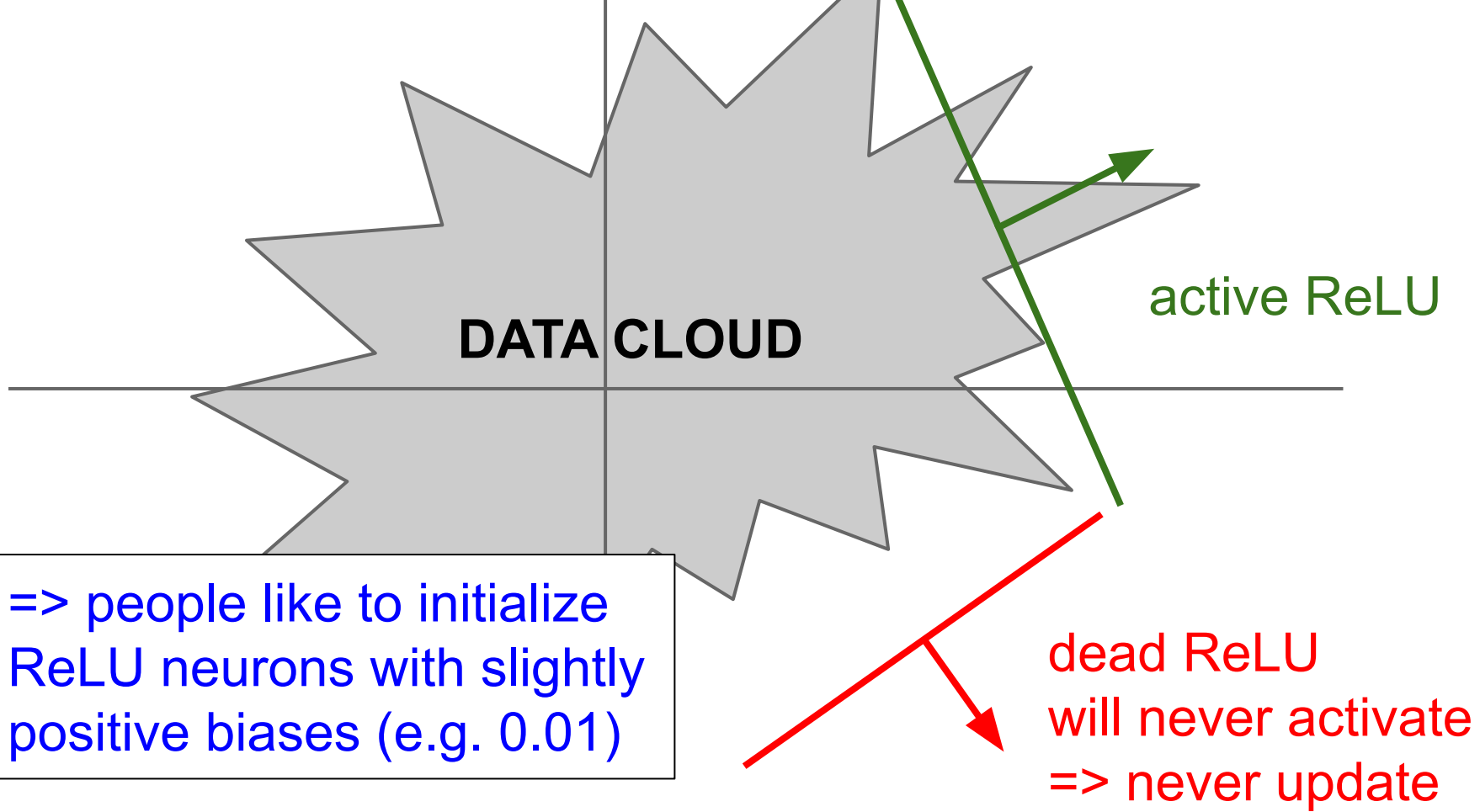
$$x$$

$$\frac{\partial \sigma}{\partial x}$$ ReLU gate

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

$$\frac{\partial L}{\partial x} = \frac{\partial \sigma}{\partial x} \frac{\partial L}{\partial \sigma}$$

$$\frac{\partial L}{\partial \sigma}$$

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

DATA CLOUD

active ReLU

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

dead ReLU
will never activate
=> never update

Activation Functions

[Mass et al., 2013]
[He et al., 2015]
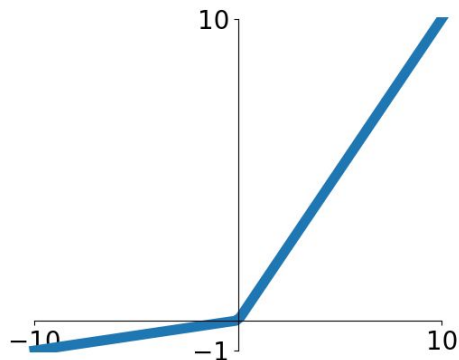
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

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

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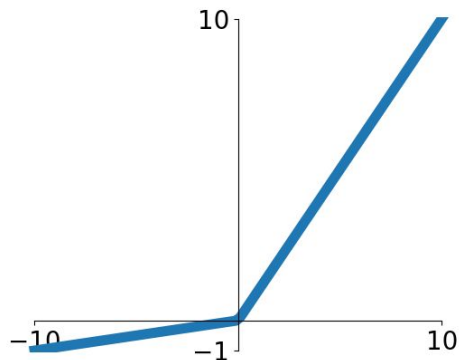
[Mass et al., 2013]
[He et al., 2015]

- 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

Exponential Linear Units (ELU)

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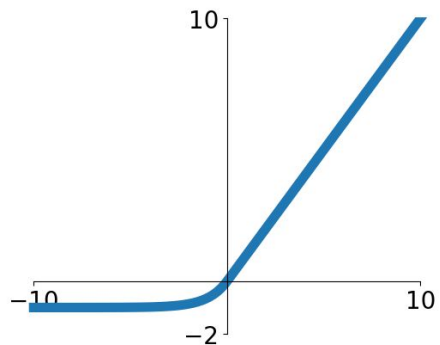
(Alpha default = 1)

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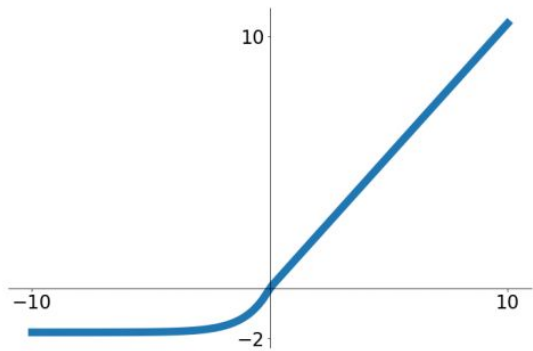
- Scaled version of ELU that works better for deep networks
- "Self-normalizing" property;
- Can train deep SELU networks without BatchNorm
  - (will discuss more later)

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

α = 1.6733, λ = 1.0507

Maxout "Neuron"

- 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

Swish

$\beta = 0.01$
$\beta = 1$
$\beta = 10$
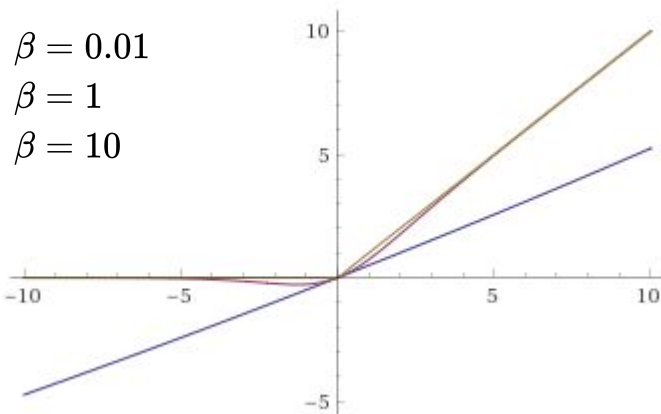
$$f(x) = x\sigma(\beta x)$$

- 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
- Don't use sigmoid or tanh

Data Preprocessing

Data Preprocessing

original data   zero-centered data   normalized data

`X -= np.mean(X, axis = 0)`   `X /= np.std(X, axis = 0)`
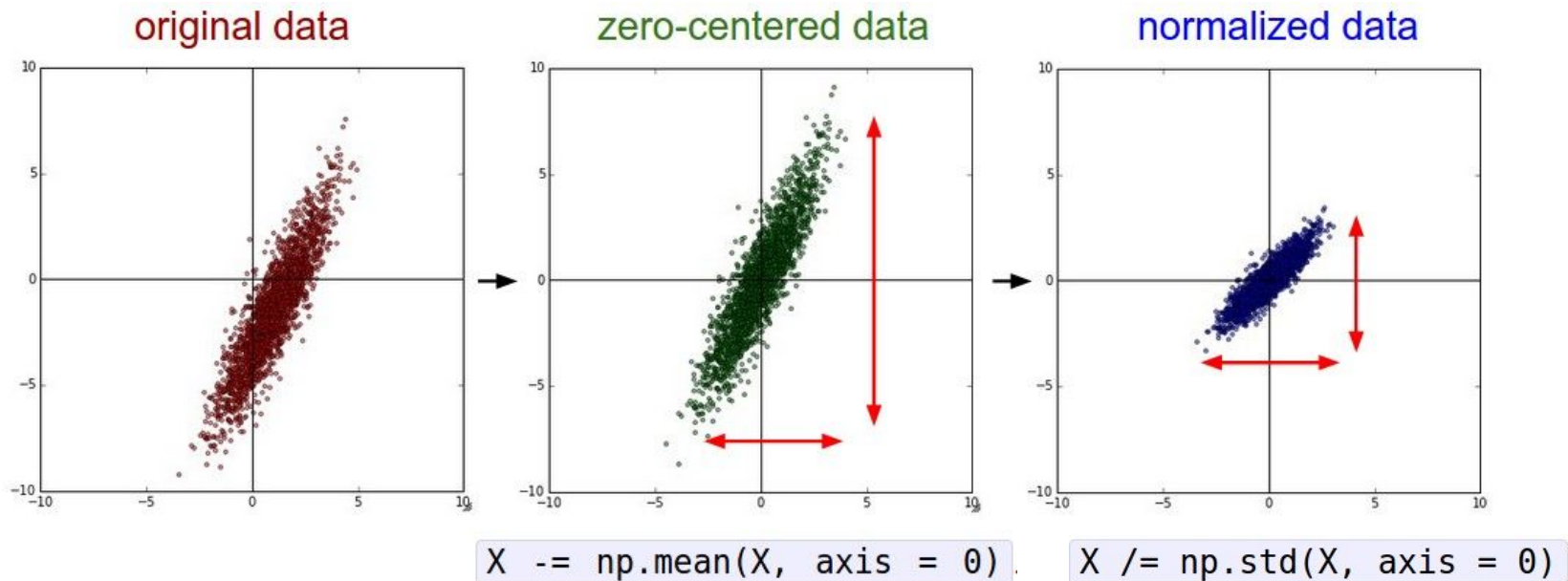
(Assume X [NxD] is data matrix,
each example in a row)

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

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

allowed gradient update directions

allowed gradient update directions

zig zag path

hypothetical optimal w vector
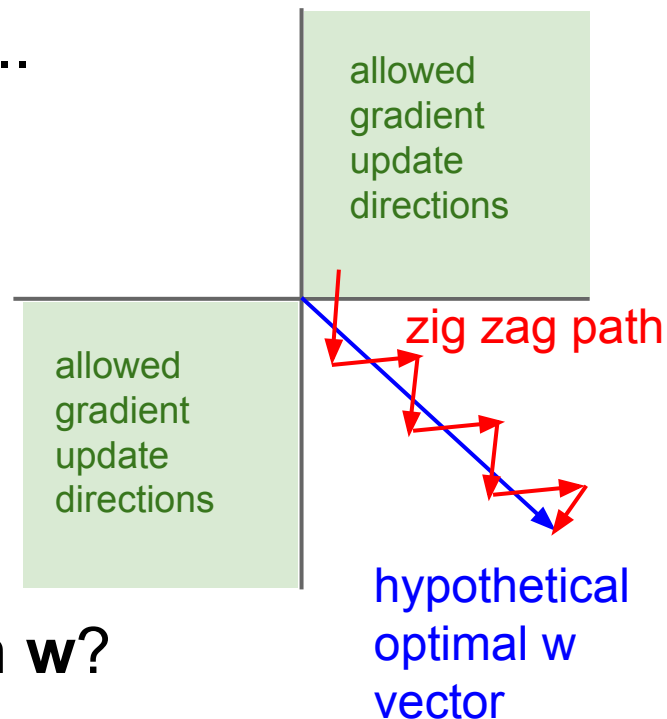
What can we say about the gradients on **w**?
Always all positive or all negative :(
(this is also why you want zero-mean data!)

Data Preprocessing

original data   zero-centered data   normalized data

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

```
X /= np.std(X, axis = 0)
```
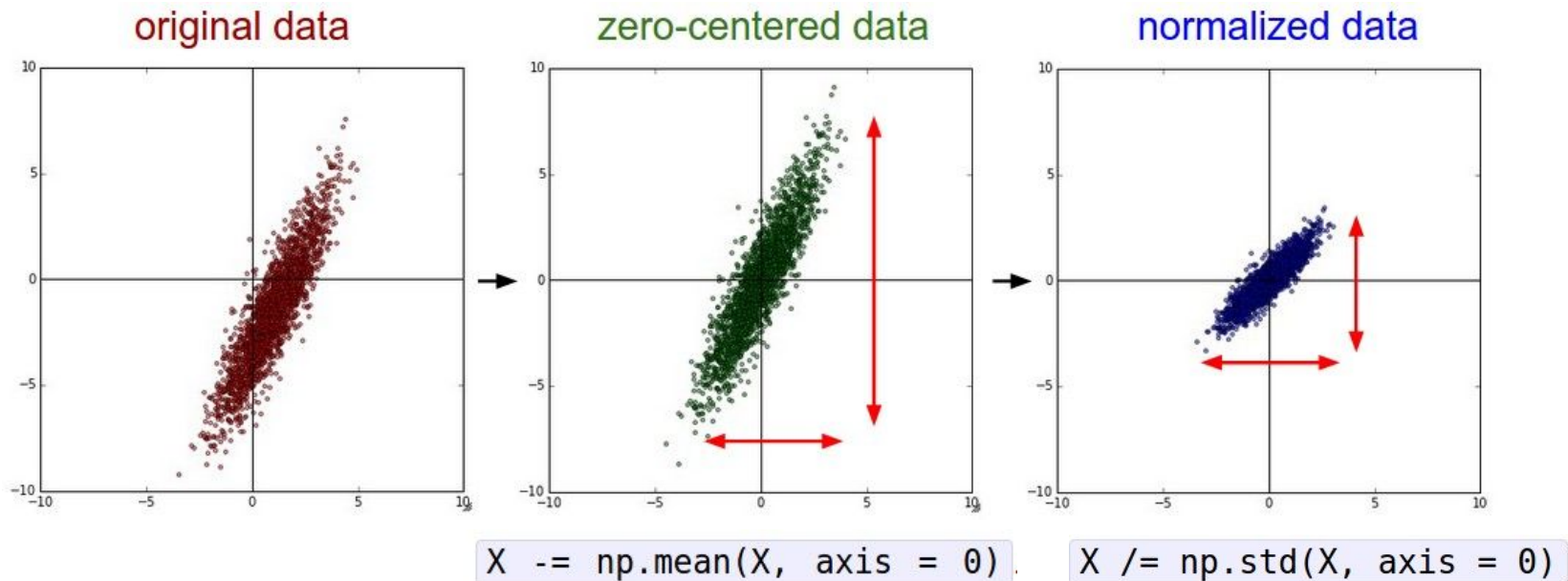
(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing

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

original data | decorrelated data | whitened data

(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)
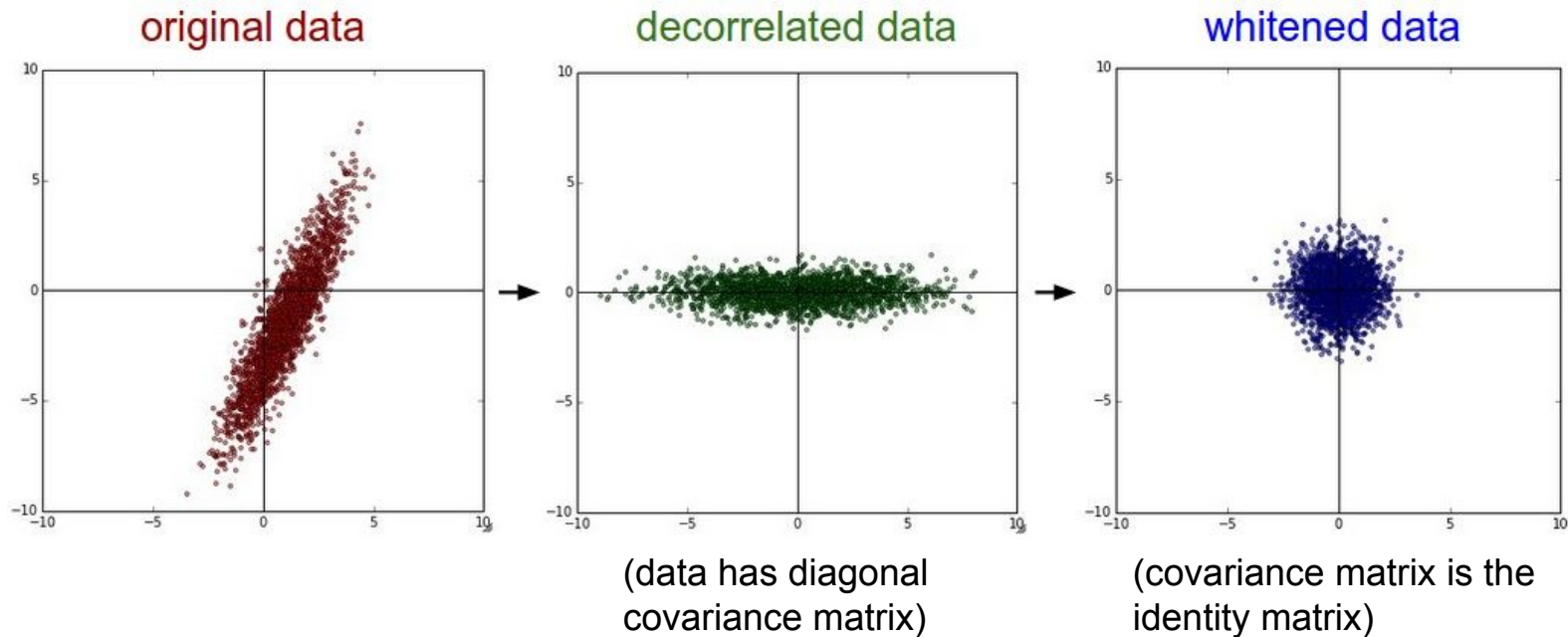
Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize
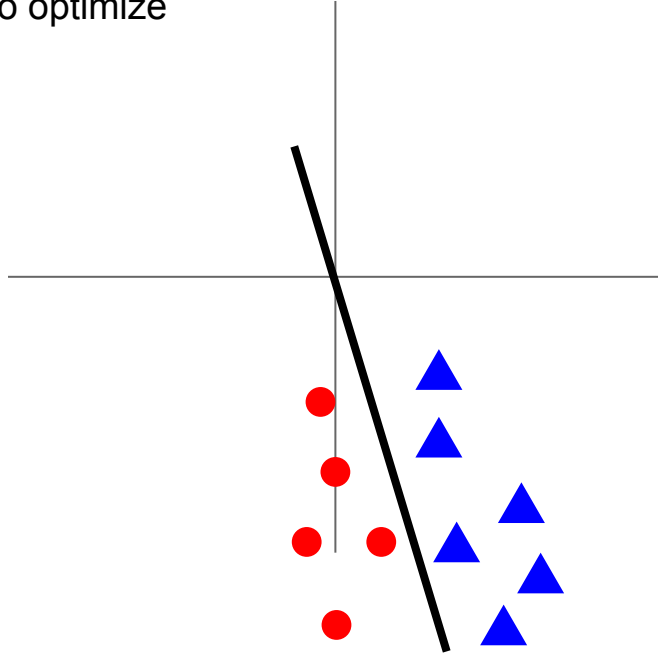
After normalization: less sensitive to small changes in weights; easier to optimize
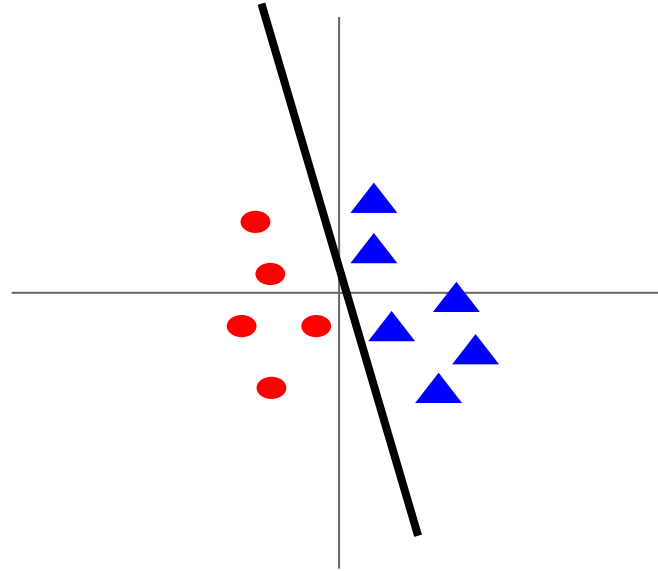
TLDR: **In practice for Images:** center only

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?

input layer

hidden layer

output layer

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

```python
W = 0.01 * np.random.randn(Din, Dout)
```
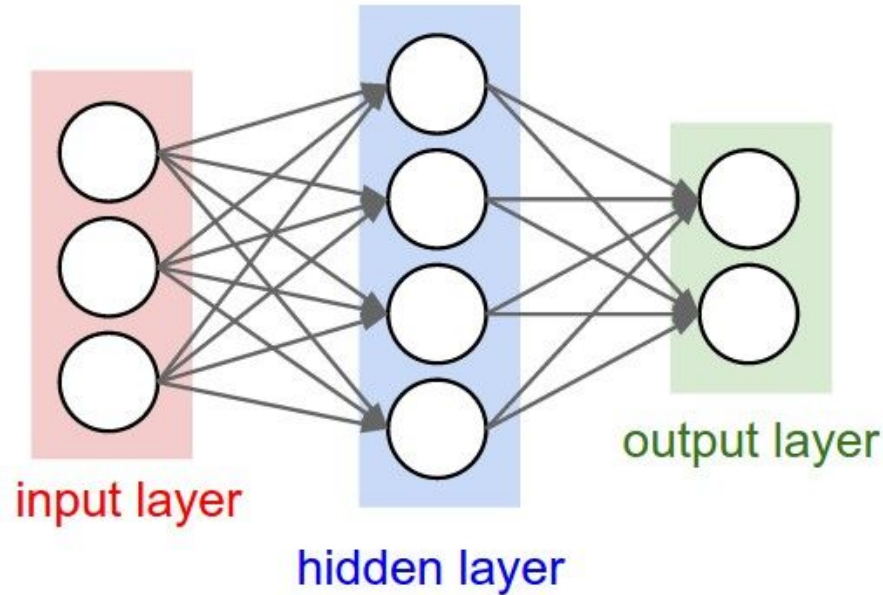
- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
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 =(

| 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
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?

Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
mean=0.00 std=0.87	mean=-0.00 std=0.85	mean=0.00 std=0.85	mean=-0.00 std=0.85	mean=0.00 std=0.85	mean=-0.00 std=0.85

Weight Initialization: Activation statistics

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

| Layer 1
mean=0.00
std=0.87 | Layer 2
mean=-0.00
std=0.85 | Layer 3
mean=0.00
std=0.85 | Layer 4
mean=-0.00
std=0.85 | Layer 5
mean=0.00
std=0.85 | Layer 6
mean=-0.00
std=0.85 |

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

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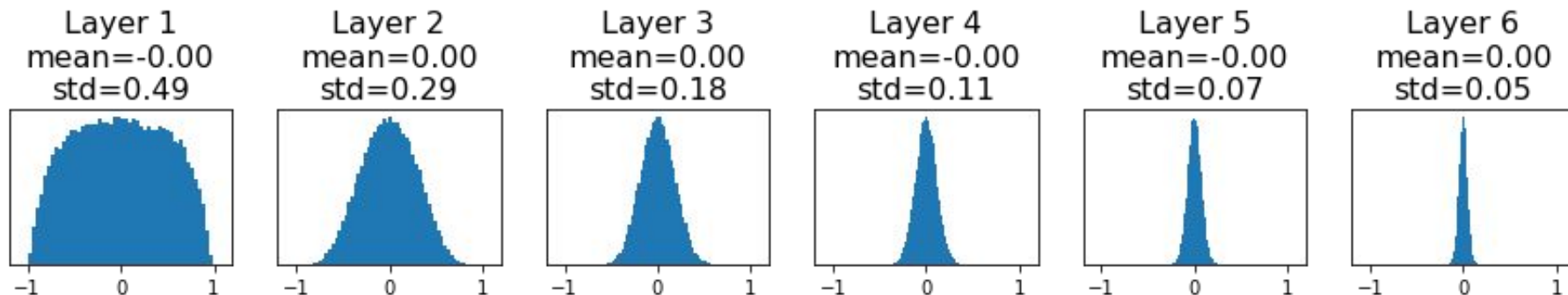
"Xavier" initialization:
std = 1/sqrt(Din)

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

Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
mean=-0.00	mean=-0.00	mean=0.00	mean=0.00	mean=0.00	mean=-0.00
std=0.63	std=0.49	std=0.41	std=0.36	std=0.32	std=0.30

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

Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
mean=-0.00	mean=-0.00	mean=0.00	mean=0.00	mean=0.00	mean=-0.00
std=0.63	std=0.49	std=0.41	std=0.36	std=0.32	std=0.30

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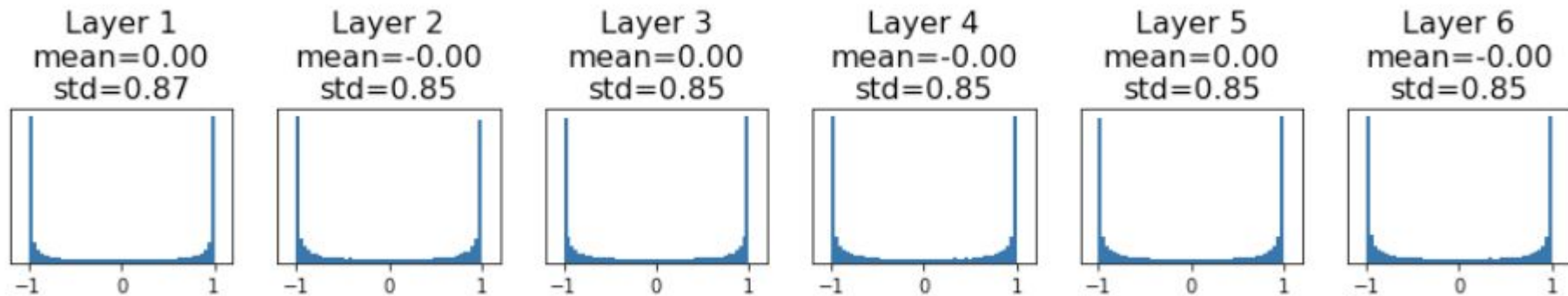
"Xavier" initialization:
std = 1/sqrt(Din)

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}$

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

Weight Initialization: "Xavier" Initialization

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

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

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + ... + x_{Din} w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

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

Weight Initialization: "Xavier" Initialization

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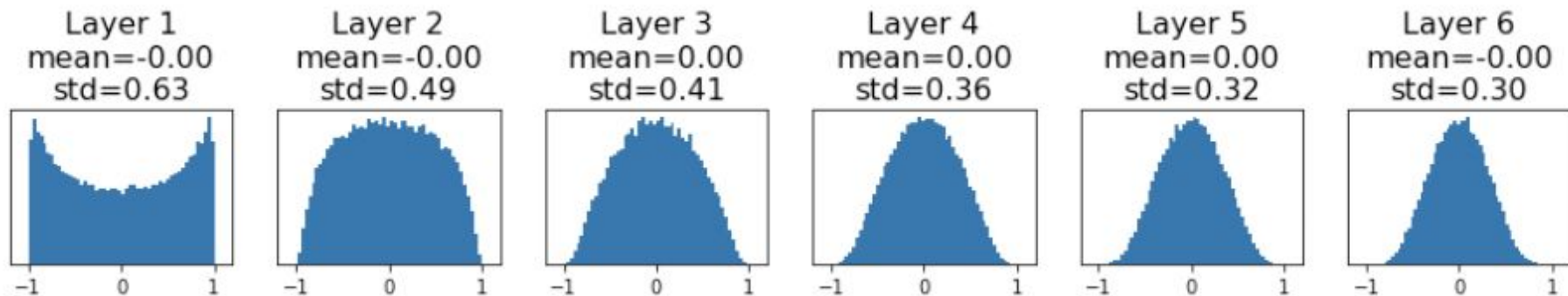
"Xavier" initialization:
std = 1/sqrt(Din)

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + ... + x_{Din} w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

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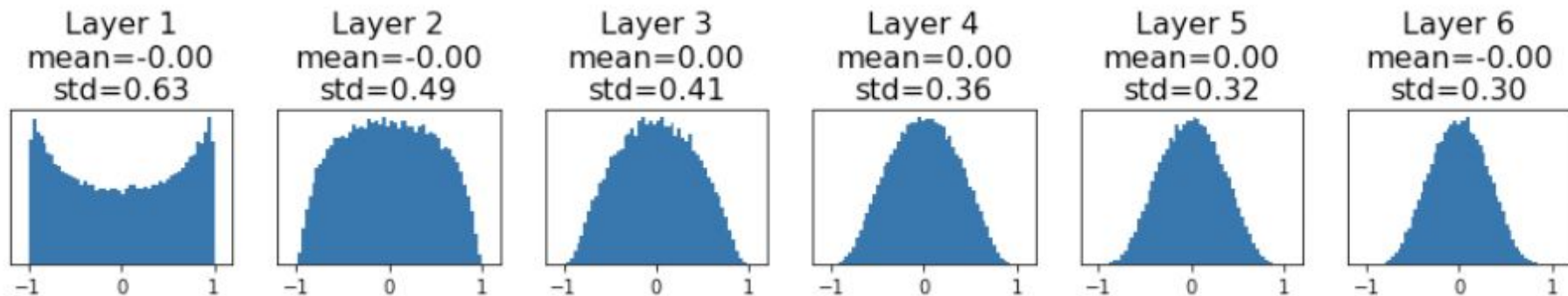
Weight Initialization: "Xavier" Initialization

```
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

Let: $y = x_1 w_1 + x_2 w_2 + ... + x_{Din} w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1 w_1 + x_2 w_2 + ... + x_{Din} w_{Din})$
[substituting value of y]

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

Let: $y = x_1w_1 + x_2w_2 + ... + x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1 + x_2w_2 + ... + x_{Din}w_{Din})$
$= Din \, Var(x_iw_i)$
[Assume all x_i, w_i 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 filter_size^2 * input_channels

Let: $y = x_1w_1 + x_2w_2 + ... + x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2) = ... = Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$Var(y) = Var(x_1w_1 + x_2w_2 + ... + x_{Din}w_{Din})$
$= Din \ Var(x_iw_i)$
$= Din \ Var(x_i) \ Var(w_i)$

[Assume all x_i, w_i 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:
$\text{std} = 1/\text{sqrt}(\text{Din})$

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

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \ldots = \text{Var}(x_{Din})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \ldots + x_{Din} w_{Din})$
$= \text{Din} \, \text{Var}(x_i w_i)$
$= \text{Din} \, \text{Var}(x_i) \, \text{Var}(w_i)$
[Assume all x_i, w_i are iid]

So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/\text{Din}$

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 =(

Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6
mean=0.39	mean=0.28	mean=0.20	mean=0.14	mean=0.10	mean=0.07
std=0.58	std=0.41	std=0.30	std=0.21	std=0.15	std=0.10
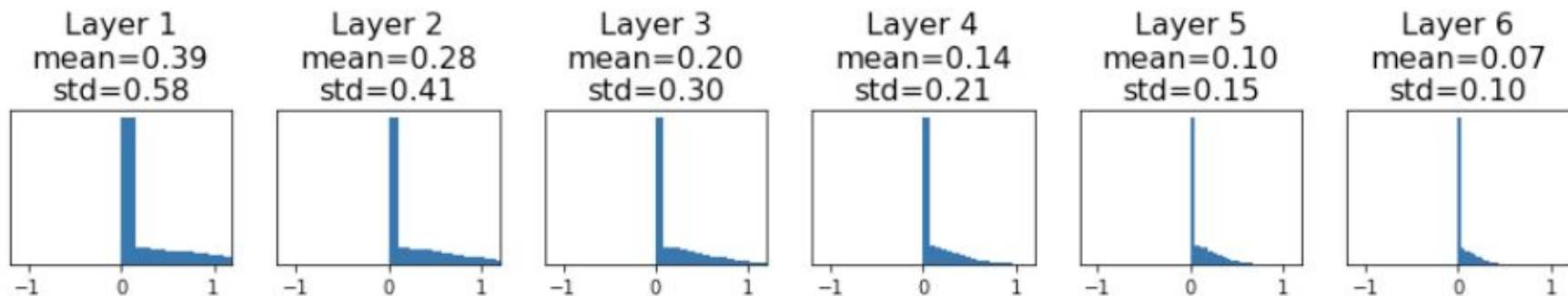
Weight Initialization: Kaiming / MSRA Initialization

```
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!

Layer 1 mean=0.57 std=0.83   Layer 2 mean=0.57 std=0.83   Layer 3 mean=0.56 std=0.83   Layer 4 mean=0.55 std=0.81   Layer 5 mean=0.55 std=0.81   Layer 6 mean=0.55 std=0.81
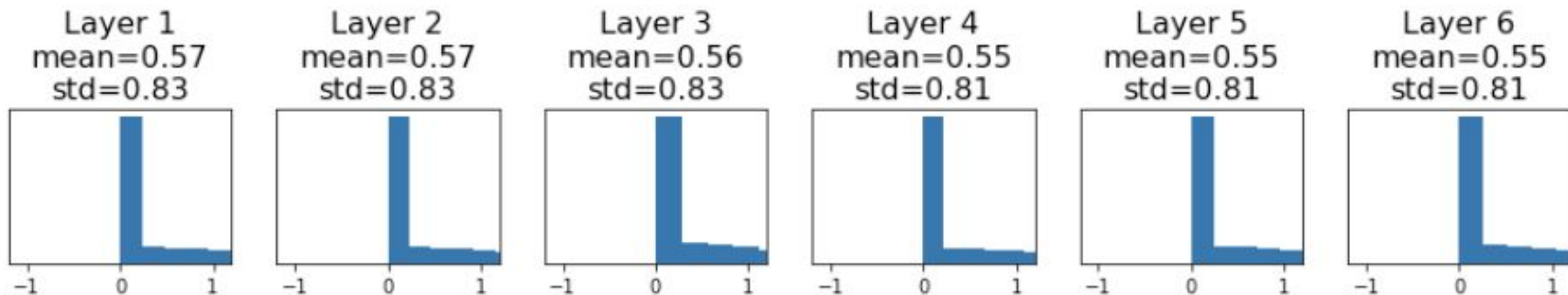
He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

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

"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
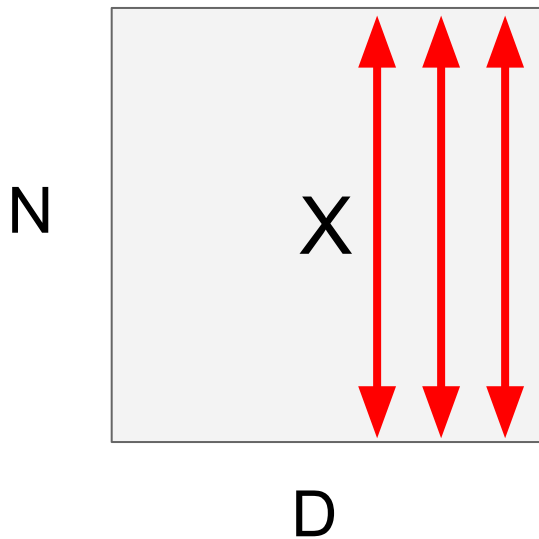
Input: $x : N \times D$

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

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

Input: $x : N \times D$

N

X

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

Input: $x : N \times D$

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

Per-channel mean, shape is D

Learnable scale and shift parameters:

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

Per-channel var, shape is D

$$\gamma, \beta : D$$

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

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

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

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$
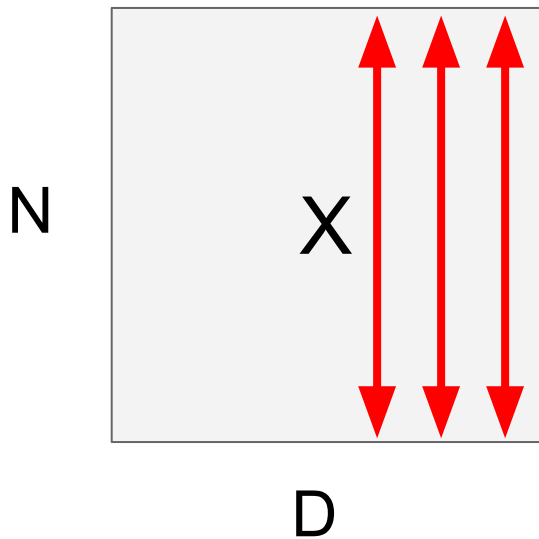
Per-channel var, shape is D

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

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: $x : N \times D$

$\mu_j = $ (Running) average of values seen during training   Per-channel mean, shape is D

Learnable scale and shift parameters:

$\gamma, \beta : D$

$\sigma_j^2 = $ (Running) average of values seen during training   Per-channel var, shape is D

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

$\hat{x}_{i,j} = \dfrac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$ 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
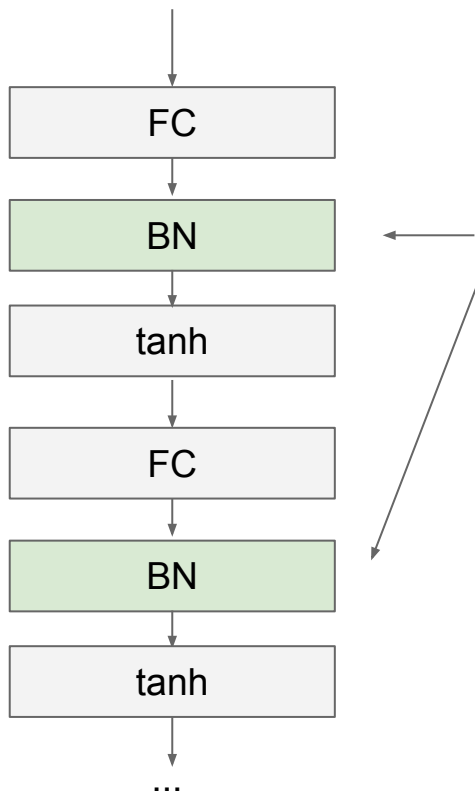
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

Batch Normalization

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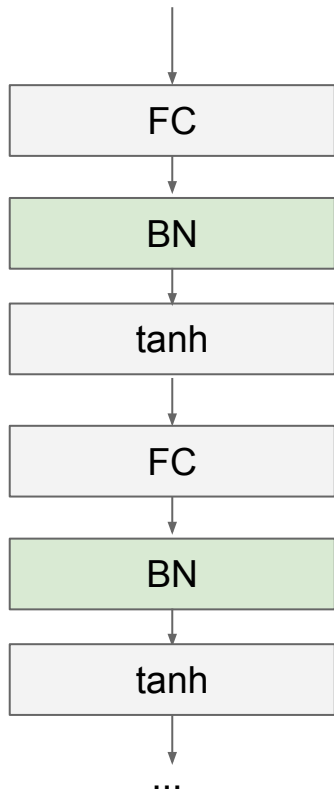
- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training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

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$$x: N \times D$$

Normalize ↓

$$\mu, \sigma: 1 \times D$$
$$\gamma, \beta: 1 \times D$$
$$y = \gamma(x-\mu)/\sigma+\beta$$

$$x: N \times C \times H \times W$$

Normalize ↓ ↓ ↓

$$\mu, \sigma: 1 \times C \times 1 \times 1$$
$$\gamma, \beta: 1 \times C \times 1 \times 1$$
$$y = \gamma(x-\mu)/\sigma+\beta$$

Layer Normalization

Batch Normalization for
fully-connected networks

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

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

Normalize ↓

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

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

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

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

Normalize ↓

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

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

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

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks

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

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

Normalize

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

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

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

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

Normalize

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

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

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

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

Comparison of Normalization Layers
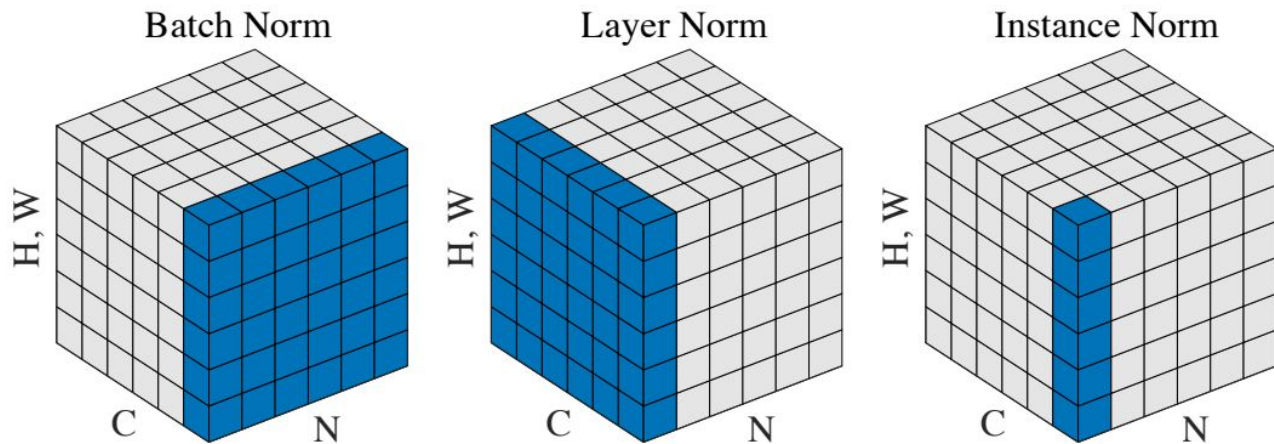
Wu and He, "Group Normalization", ECCV 2018

Group Normalization
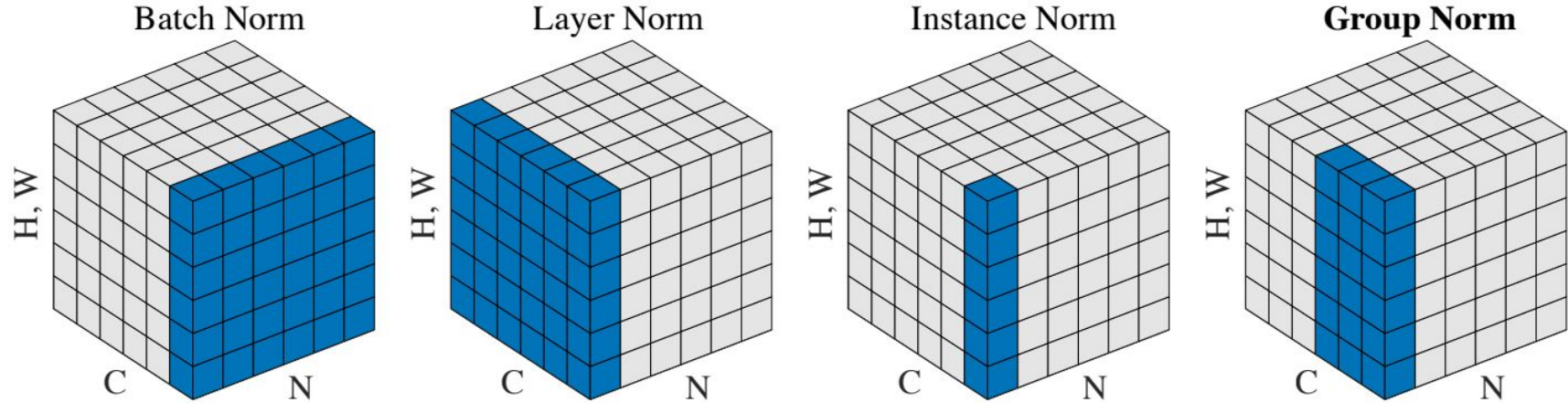
Wu and He, "Group Normalization", ECCV 2018

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
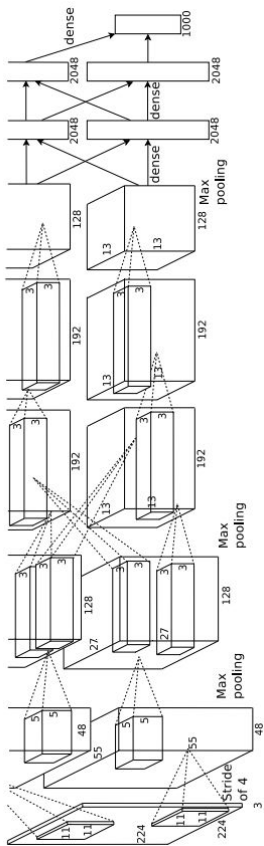
Transfer Learning with CNNs
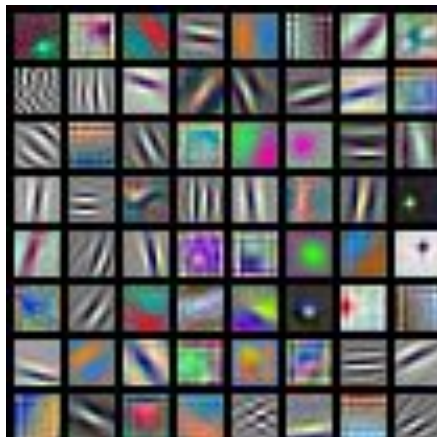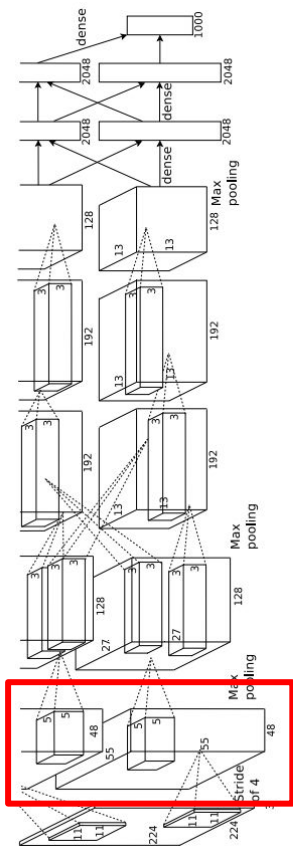
AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)

Transfer Learning with CNNs

Test image   L2 Nearest neighbors in <u>feature</u> space

(More on this in Lecture 13)

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

| FC-1000 |
| FC-4096 |
| FC-4096 |

| MaxPool |
| Conv-512 |
| Conv-512 |

| MaxPool |
| Conv-512 |
| Conv-512 |

| MaxPool |
| Conv-256 |
| Conv-256 |

| MaxPool |
| Conv-128 |
| Conv-128 |

| MaxPool |
| Conv-64 |
| Conv-64 |

| Image |

Transfer Learning with CNNs

1. Train on Imagenet

| FC-1000 |
| FC-4096 |
| FC-4096 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-256 |
| Conv-256 |
| MaxPool |
| Conv-128 |
| Conv-128 |
| MaxPool |
| Conv-64 |
| Conv-64 |
| Image |

2. Small Dataset (C classes)

| FC-C |
| FC-4096 |
| FC-4096 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-256 |
| Conv-256 |
| MaxPool |
| Conv-128 |
| Conv-128 |
| MaxPool |
| Conv-64 |
| Conv-64 |
| Image |

Reinitialize this and train

Freeze these

Transfer Learning with CNNs

1. Train on Imagenet

FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

2. Small Dataset (C classes)

FC-C
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

Reinitialize this and train

Freeze these

Finetuned from AlexNet

DPD (Zhang et al., 2013)	POOF (Berg & Belhumer, 2013)	AlexNet FC6 + logistic regression	AlexNet FC6 + DPD
50.98	56.78	58.75	64.96

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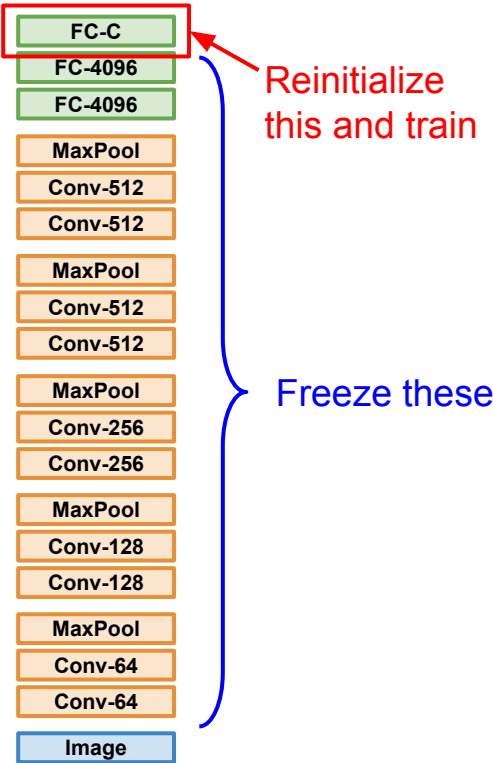
Transfer Learning with CNNs

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
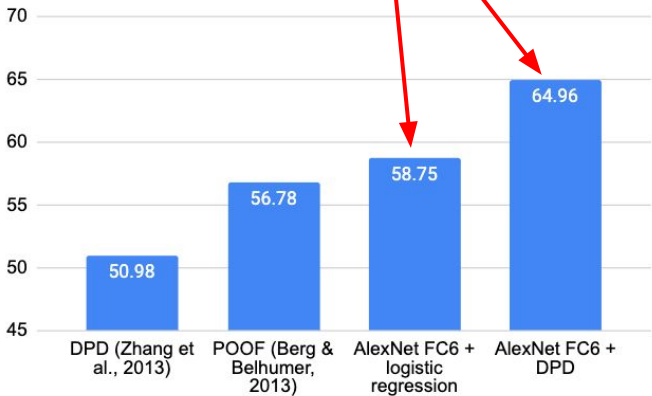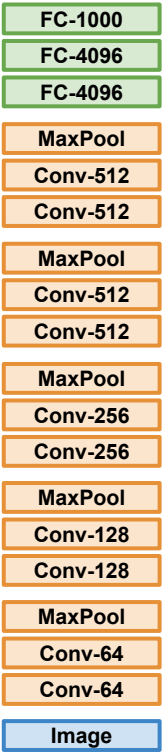
1. Train on Imagenet

| FC-1000 |
| FC-4096 |
| FC-4096 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-256 |
| Conv-256 |
| MaxPool |
| Conv-128 |
| Conv-128 |
| MaxPool |
| Conv-64 |
| Conv-64 |
| Image |

2. Small Dataset (C classes)

| FC-C |
| FC-4096 |
| FC-4096 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-256 |
| Conv-256 |
| MaxPool |
| Conv-128 |
| Conv-128 |
| MaxPool |
| Conv-64 |
| Conv-64 |
| Image |

Reinitialize this and train

Freeze these

3. Bigger dataset

| FC-C |
| FC-4096 |
| FC-4096 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-512 |
| Conv-512 |
| MaxPool |
| Conv-256 |
| Conv-256 |
| MaxPool |
| Conv-128 |
| Conv-128 |
| MaxPool |
| Conv-64 |
| Conv-64 |
| Image |

Train these

With bigger dataset, train more layers

Freeze these

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

FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

More specific

More generic

	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?

FC-1000
FC-4096
FC-4096
MaxPool
Conv-512
Conv-512
MaxPool
Conv-512
Conv-512
MaxPool
Conv-256
Conv-256
MaxPool
Conv-128
Conv-128
MaxPool
Conv-64
Conv-64
Image

More specific

More generic

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

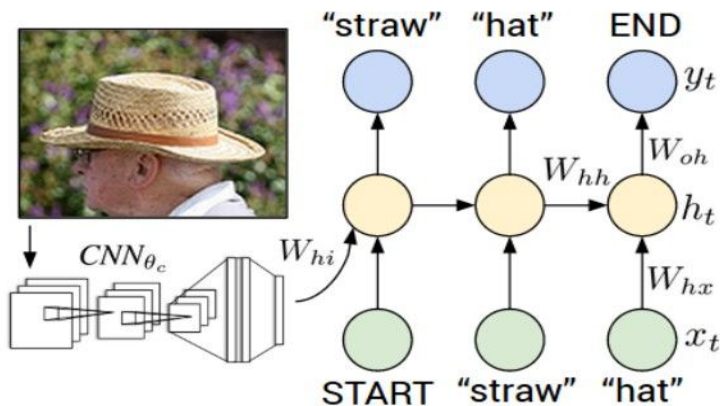|	**very similar dataset**	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

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
Figure copyright Ross Girshick, 2015. Reproduced with permission.

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

Log loss + smooth L1 loss

Proposal
classifier

Linear +
softmax

Linear

Bounding box
regressors

FCs

RoI pooling

External proposal
algorithm
e.g. selective search

ConvNet
(applied to entire
image)

CNN_{θ_c}

"straw" "hat" END

V_{hi}

W_{hh}

W_{oh}

y_t

h_t

W_{hx}

x_t

START "straw" "hat"

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

Log loss + smooth L1 loss

Proposal classifier

Linear + softmax

Linear

Bounding box regressors

FCs

RoI pooling

External proposal algorithm
e.g. selective search

ConvNet
(applied to entire image)

Word vectors pretrained with word2vec

CNN_{θ_c}

"straw" "hat" END

y_t

W_{hh}

W_{oh}

h_t

V_{hi}

W_{hx}

x_t

START "straw" "hat"

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

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
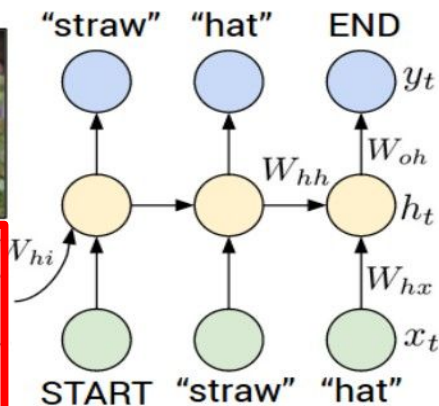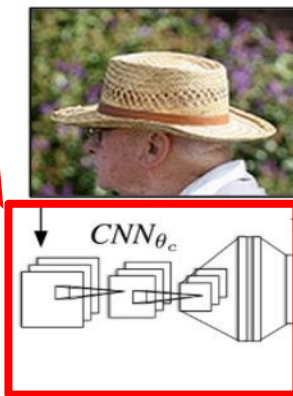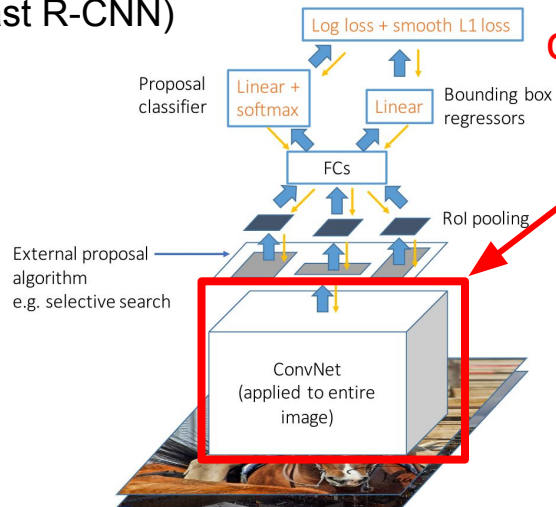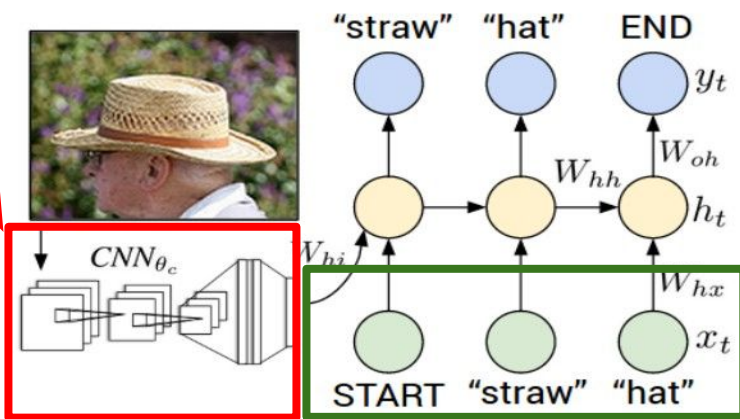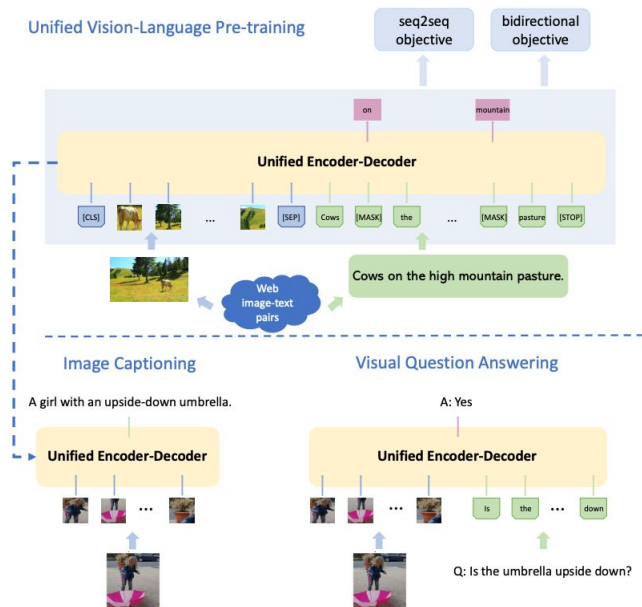Figure copyright IEEE, 2015. Reproduced for educational purposes.

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

Unified Vision-Language Pre-training

seq2seq objective

bidirectional objective

Unified Encoder-Decoder

on   mountain

[CLS] ... [SEP] Cows [MASK] the ... [MASK] pasture [STOP]

Web image-text pairs

Cows on the high mountain pasture.

Image Captioning

A girl with an upside-down umbrella.

Unified Encoder-Decoder

Visual Question Answering

A: Yes

Unified Encoder-Decoder

Is the ... down

Q: Is the umbrella upside down?

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

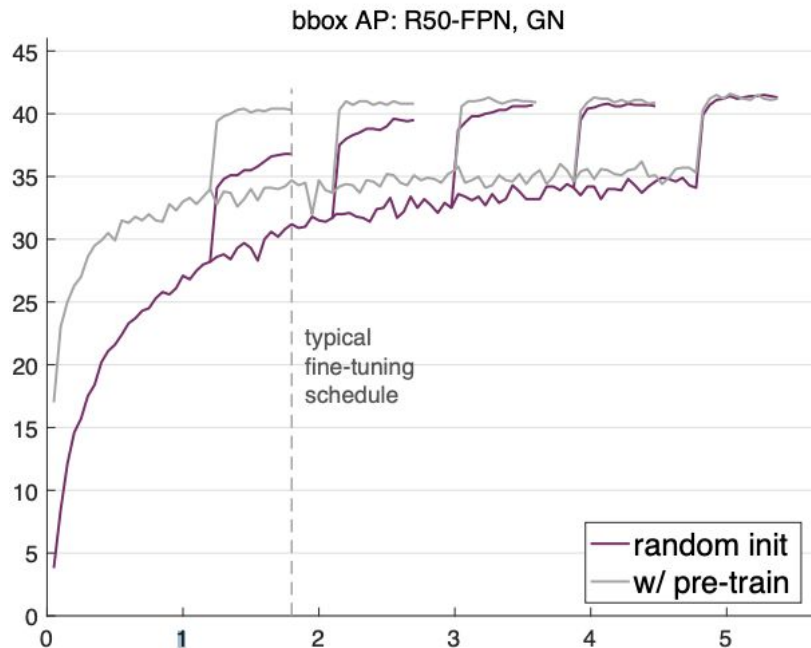
Object detection on MSCOCO
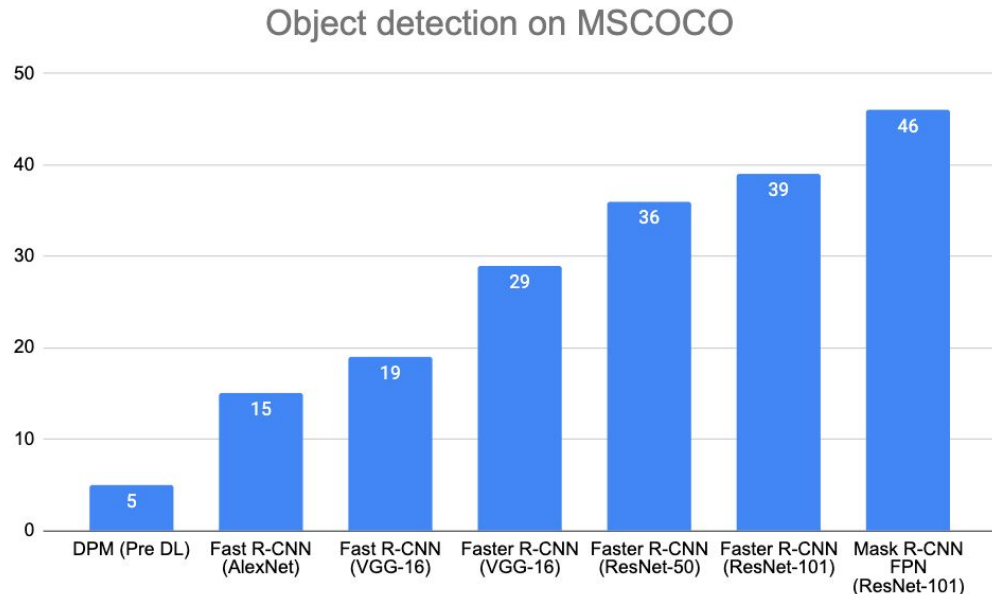
We will discuss different architectures in detail in two lectures

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

Transfer learning with CNNs is pervasive…
But recent results show it might not always be necessary!

bbox AP: R50-FPN, GN

typical fine-tuning schedule

— random init
— w/ pre-train

He et al, "Rethinking ImageNet Pre-training", ICCV 2019
Figure copyright Kaiming He, 2019. Reproduced with permission.

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

Custom layers

Pretrained layers

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