Lecture 6:

Activation Functions & Normalization Layers

Administrative: Assignment 1

Due today! 11:59pm

- K-Nearest Neighbor
- Linear classifiers: SVM, Softmax

Administrative: Assignment 2

Out soon

- Multi-layer Neural Networks,
- Image Features,
- Optimizers

Administrative: Fridays

This Friday

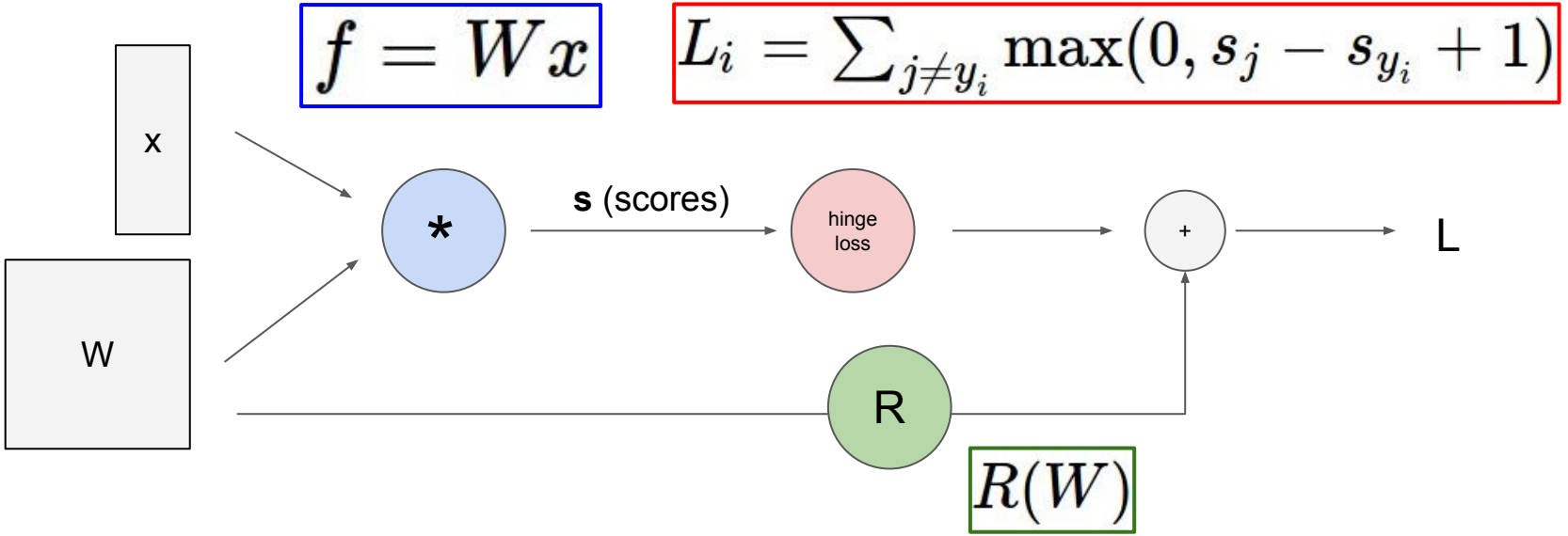
Convolutions & Vectorization

Project

Fill out partner form by eod if you want to!

Where we are now...

Computational graphs



Where we are now...

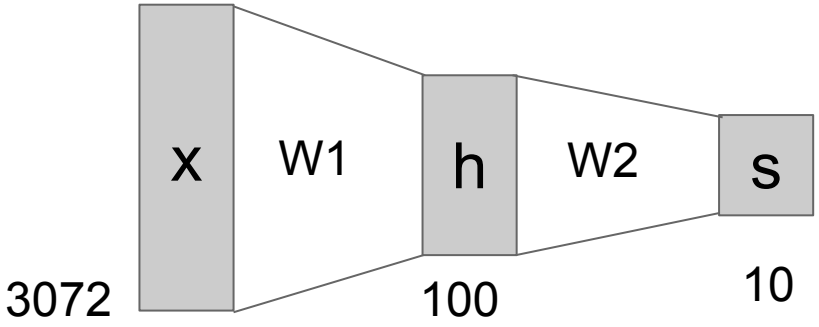
Neural Networks

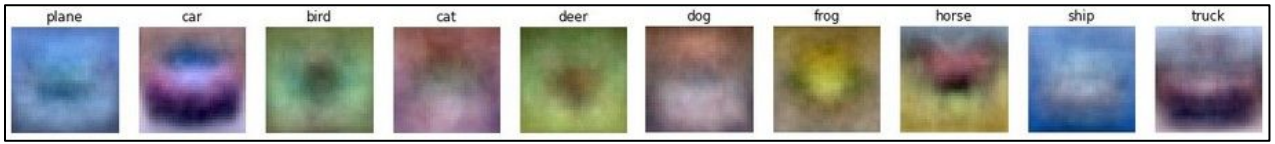
Linear score function:

$$f = Wx$$

2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$





Where we are now...

Convolutional Neural Networks

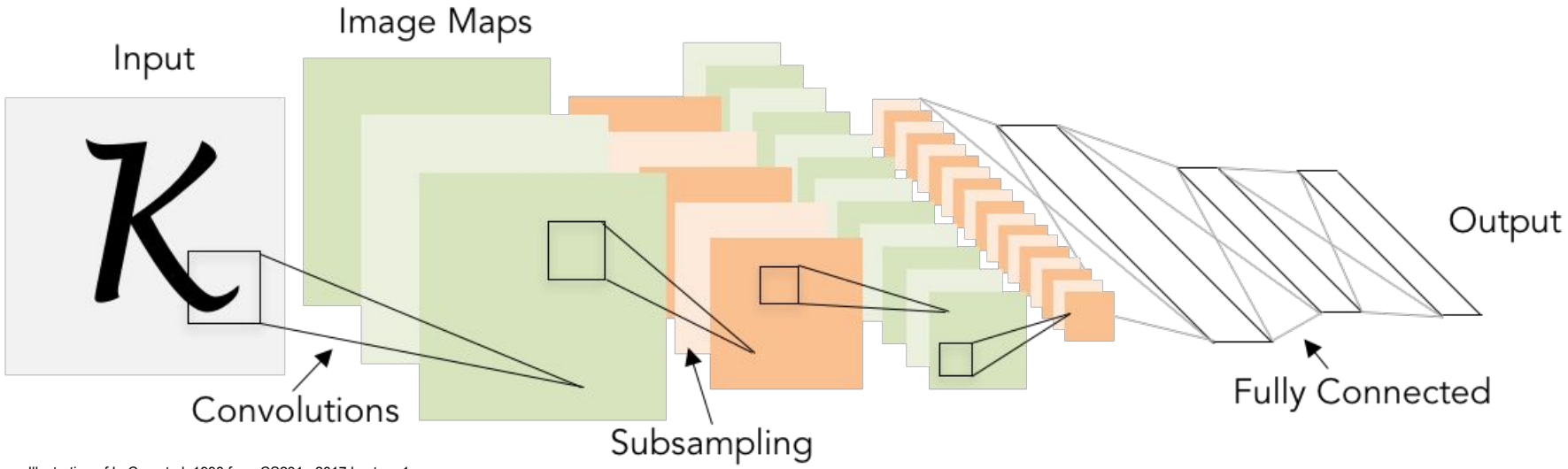
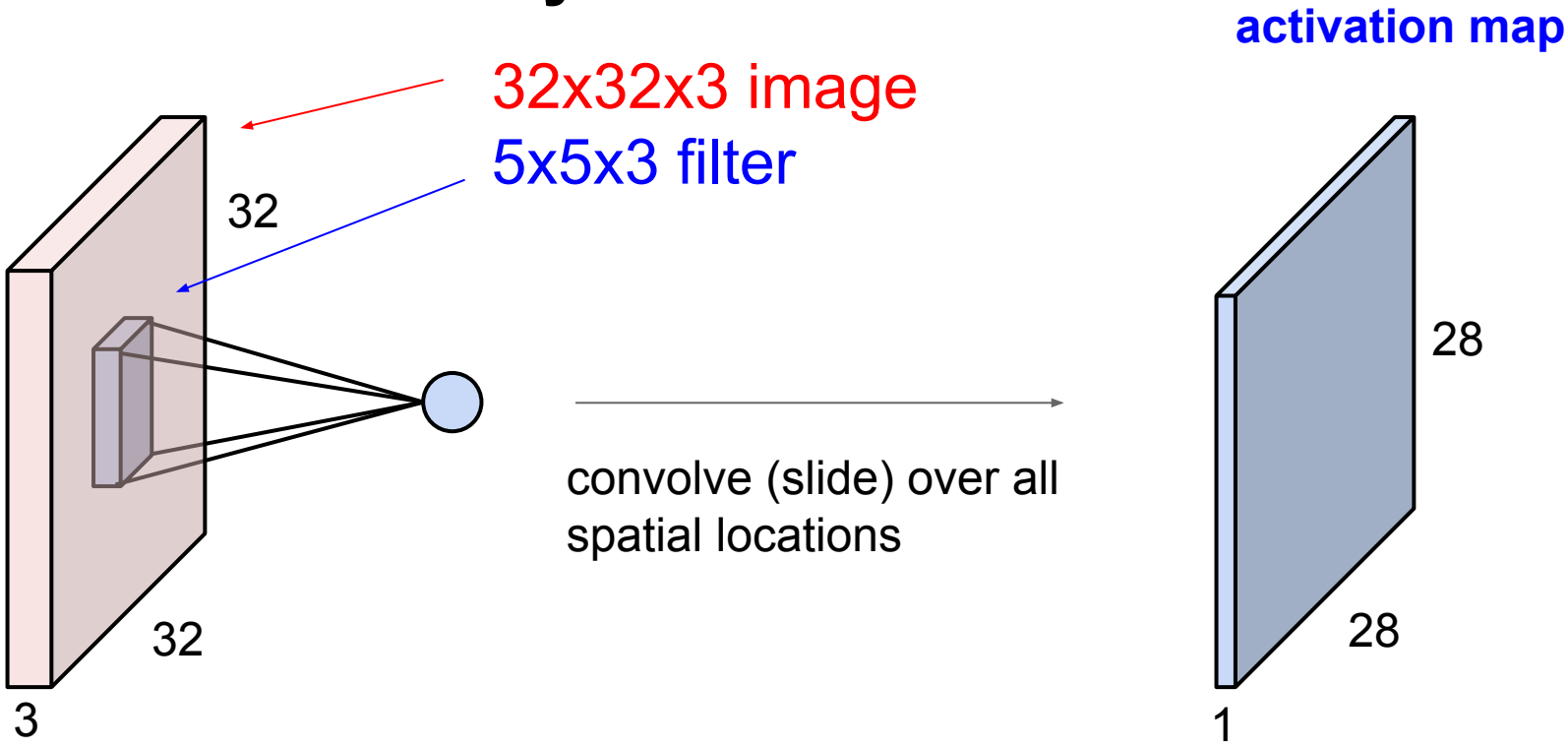


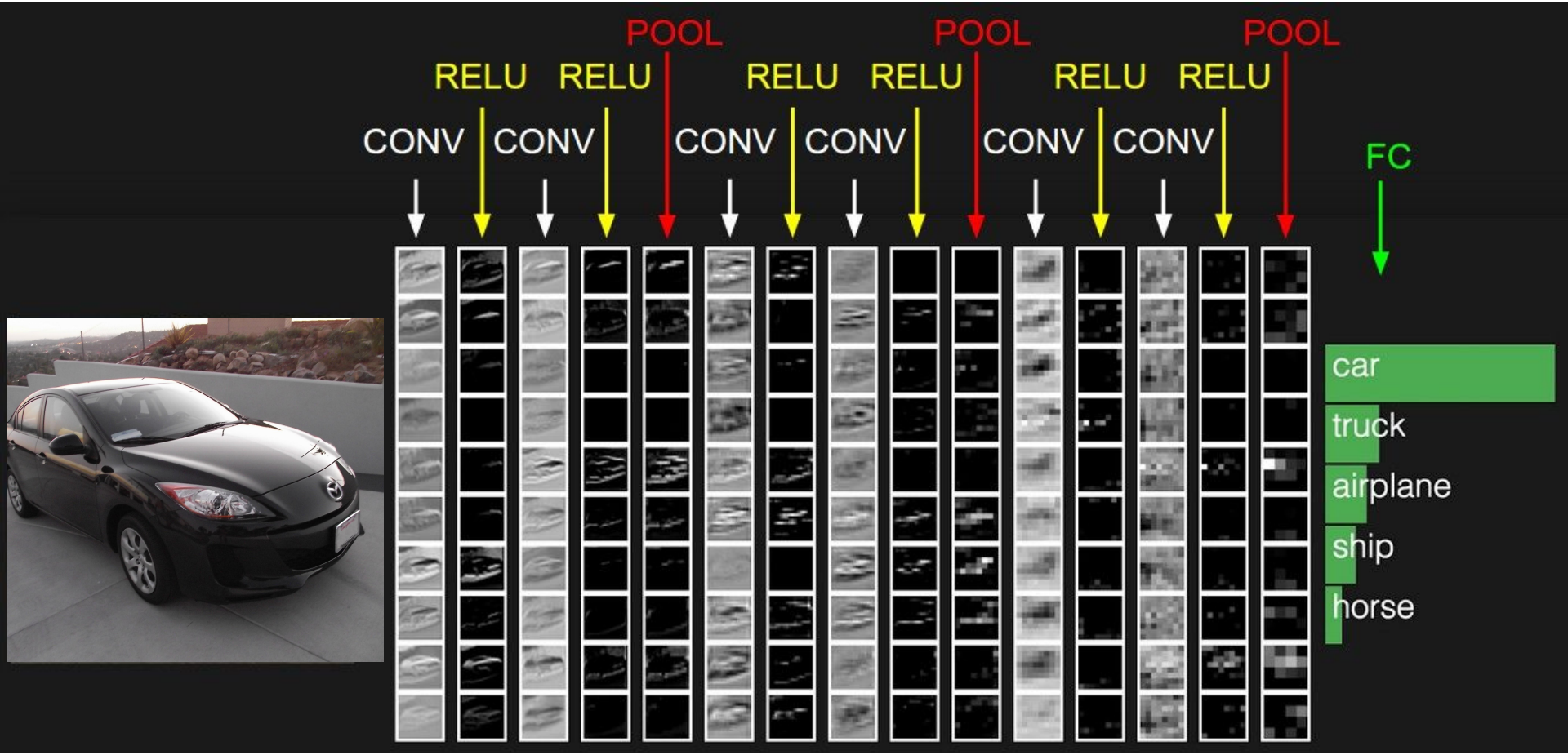
Illustration of LeCun et al. 1998 from CS231n 2017 Lecture 1

Where we are now...

Convolutional Layer



FOUR layers in total: CONV/ReLU/POOL/FC



Today:

- Activation Function
- Normalization layers

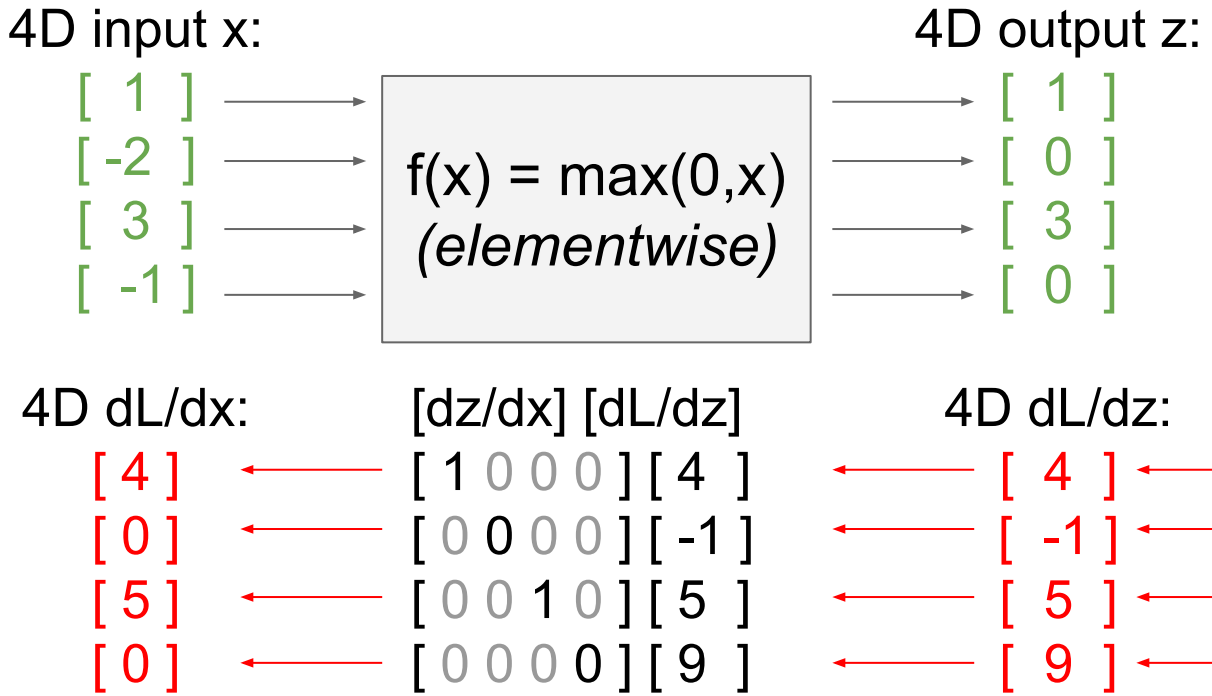
There are a few more steps before we start training

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

Today's agenda

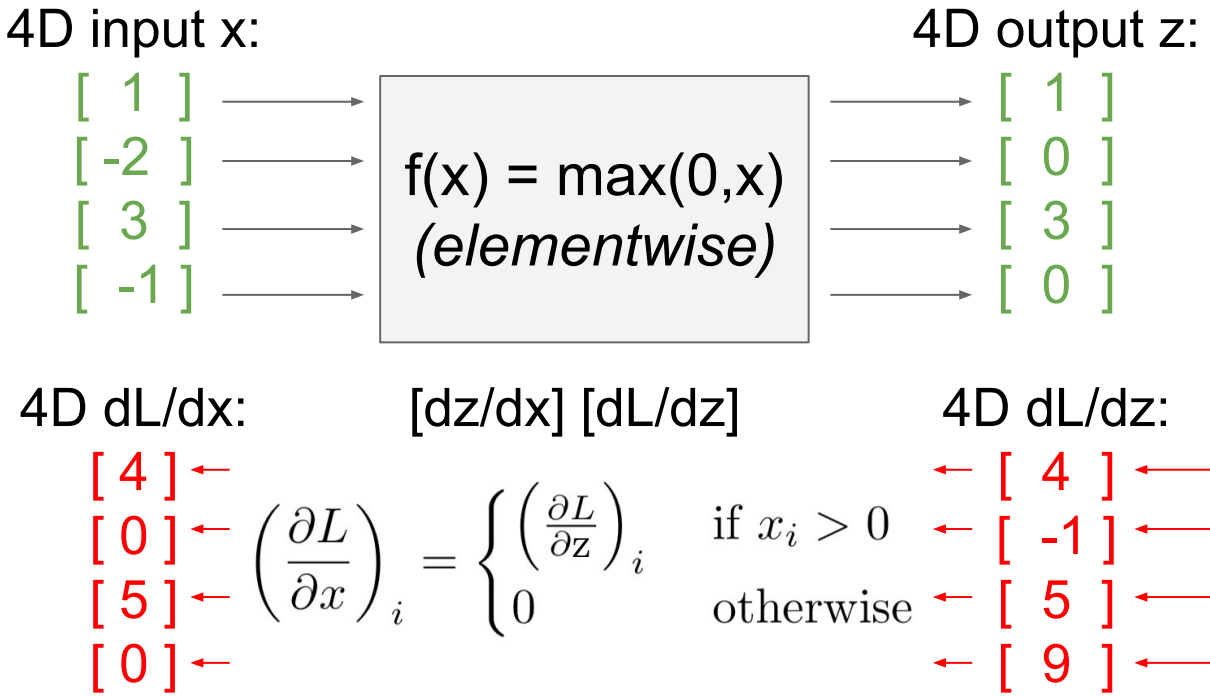
- Briefly revisit backprop
- Finish CNNs
- Activation Functions
- Data Preprocessing
- Weight Initialization
- Normalization Layers

Backprop with Vectors



Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication

Backprop with Vectors



Jacobian is **sparse**: off-diagonal entries always zero! Never **explicitly** form Jacobian -- instead use **implicit** multiplication

Backprop with Matrices

x: [N×D]

2	1	−3
−3	4	2

w: [D×M]

3	2	1	−1
2	1	3	2
3	2	1	−2

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

−1	−1	2	6
5	2	11	1

dL/dy: [N×M]

2	3	−3	9
−8	1	4	6

Backprop with Matrices

x: [N×D]

2	1	−3
−3	4	2

w: [D×M]

3	2	1	−1
2	1	3	2
3	2	1	−2

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

−1	−1	2	6
5	2	11	1

dL/dy: [N×M]

2	3	−3	9
−8	1	4	6

Jacobians:
dy/dx: [(N×D)×(N×M)]
dy/dw: [(D×M)×(N×M)]

For a neural net we may have
N=64, D=M=4096
Each Jacobian takes ~256 GB of memory! Must work with them implicitly!

Backprop with Matrices

x: [N×D]

2	1	−3
−3	4	2

w: [D×M]

3	2	1	−1
2	1	3	2
3	2	1	−2

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

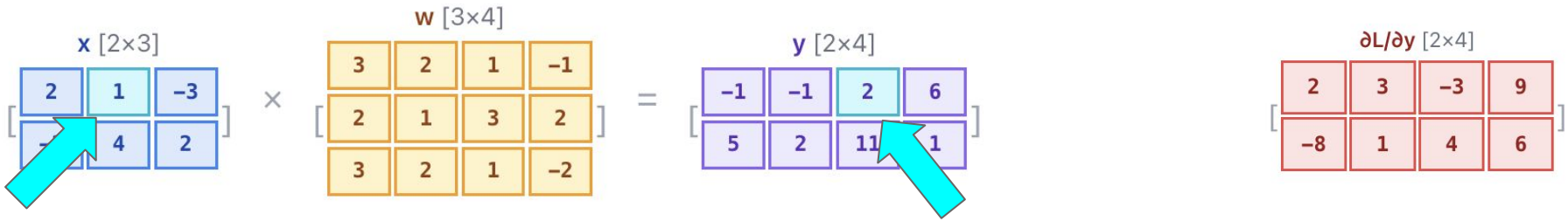
−1	−1	2	6
5	2	11	1

dL/dy: [N×M]

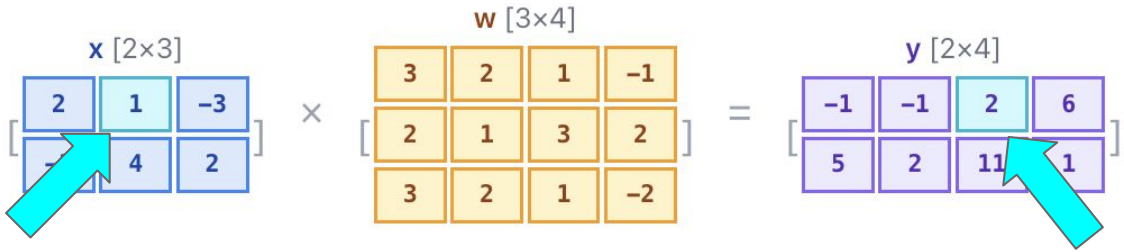
2	3	−3	9
−8	1	4	6

Jacobian is very large because it considers how EVERY element of the input x affects EVERY element of the output y

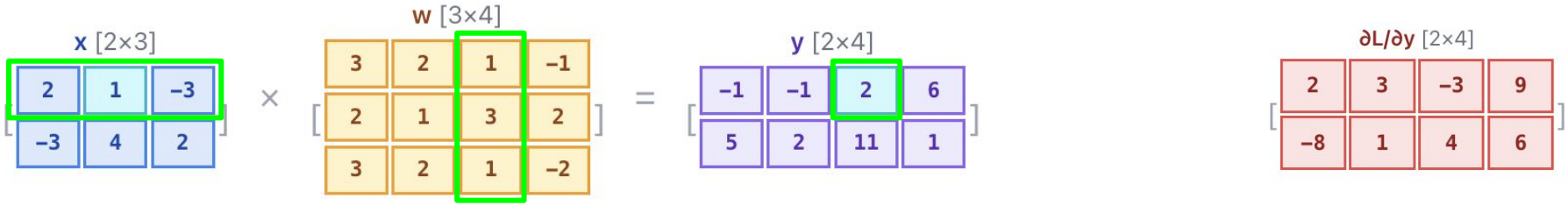
Can we get dL/dx more efficiently, by just looking at what parts of the input ACTUALLY affect which parts of the output?



Let’s look at how a single element of the input affects a single element of the output!



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

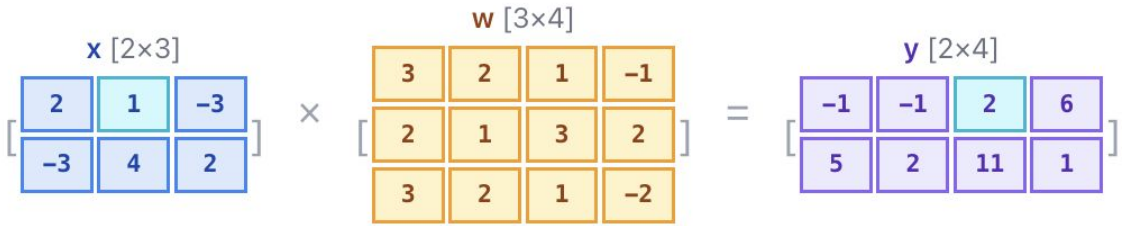


What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$



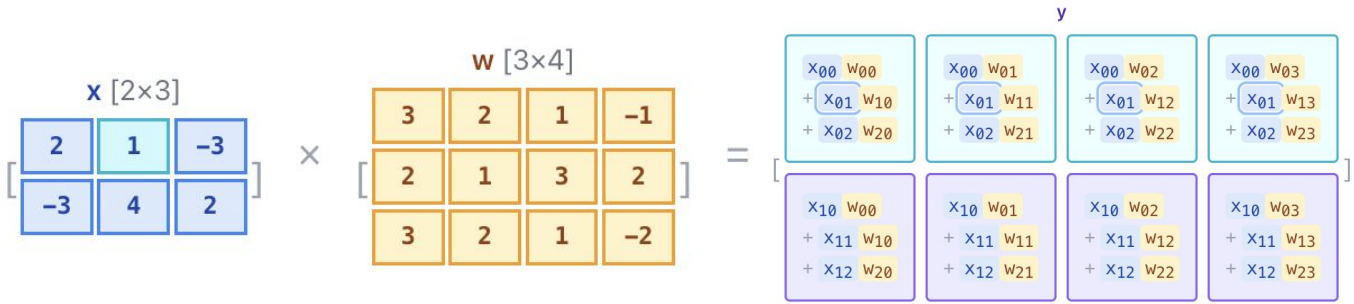
What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

What is $\frac{\partial y}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$



$\partial L/\partial y$ [2×4]

2	3	−3	9
−8	1	4	6

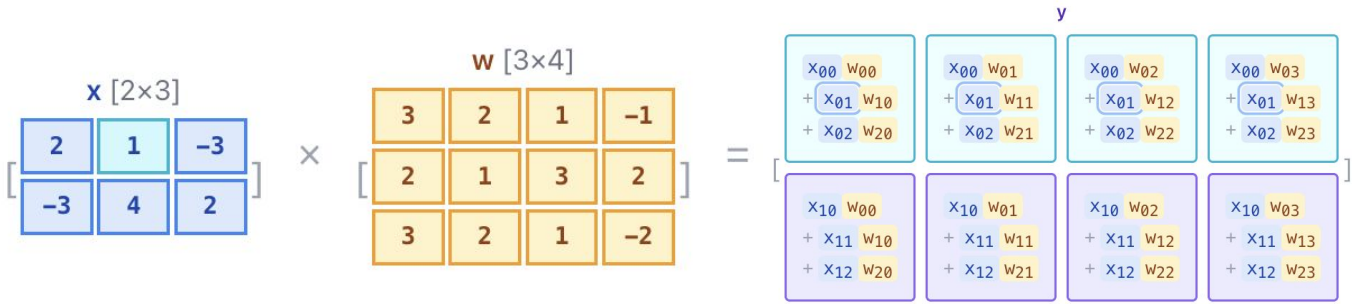
What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

What is $\frac{\partial y}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$



$\partial L/\partial y$ [2×4]

2	3	−3	9
−8	1	4	6

What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

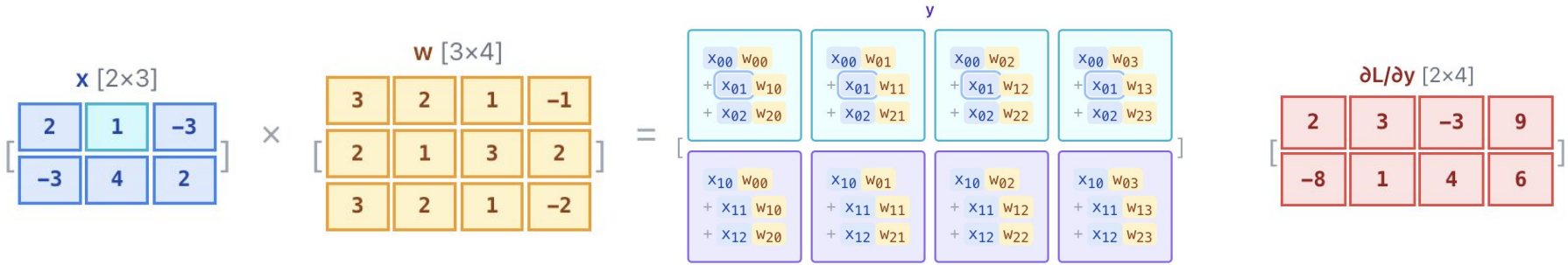
$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0 \text{ } (x_{0,1} \text{ doesn't affect row 1})$$



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

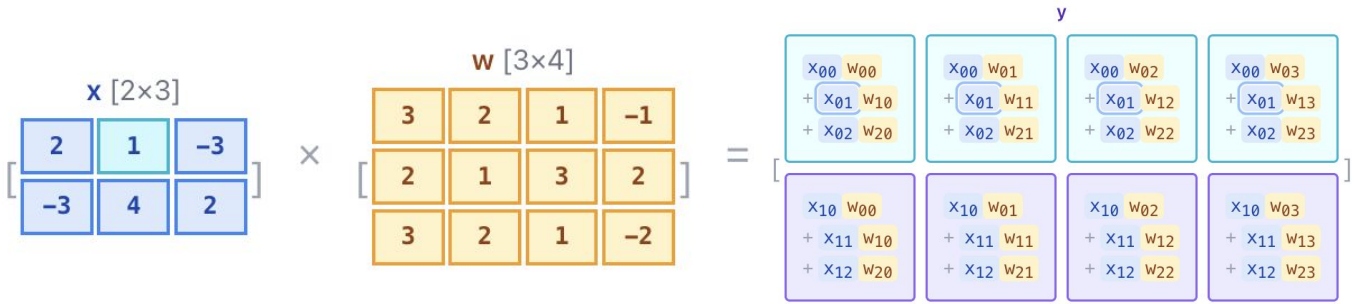
$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0 \text{ } (x_{0,1} \text{ doesn't affect row 1})$$

What is $\frac{\partial L}{\partial x_{0,1}}$?



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

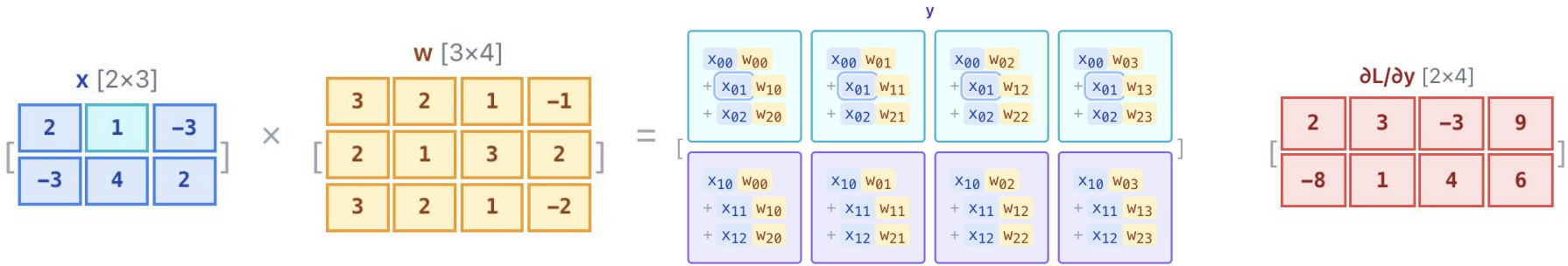
$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0 \text{ } (x_{0,1} \text{ doesn't affect row 1})$$

What is $\frac{\partial L}{\partial x_{0,1}}$?

Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

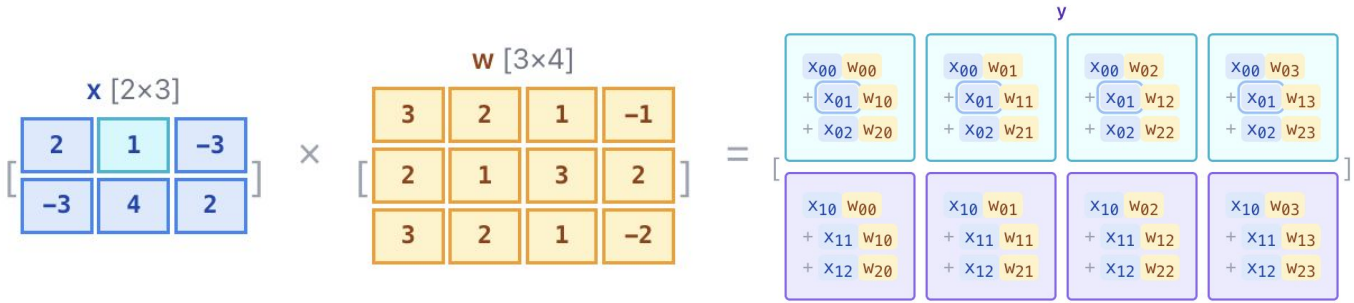
$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0 \text{ ($x_{0,1}$ doesn't affect row 1)}$$

What is $\frac{\partial L}{\partial x_{0,1}}$?

Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$
$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0$ ($x_{0,1}$ doesn't affect row 1)

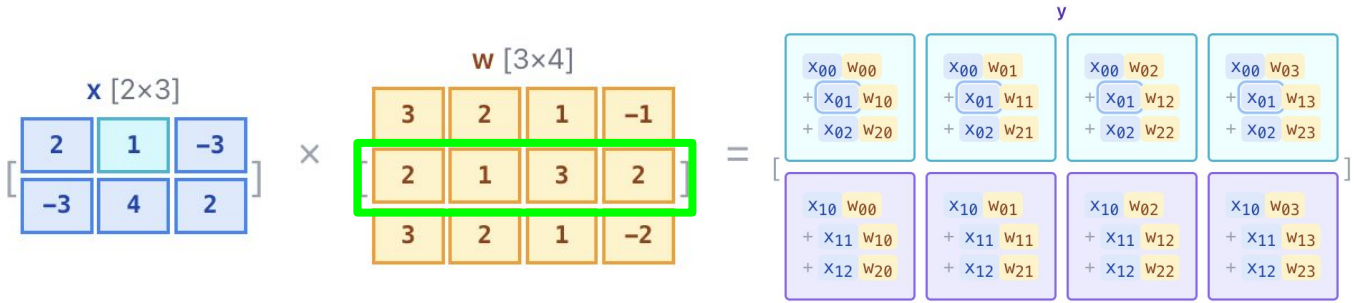
What is $\frac{\partial L}{\partial x_{0,1}}$?

Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$



What is $\frac{\partial y_{0,2}}{\partial x_{0,1}}$?

Looking at the formula for $y_{0,2}$, what's the partial derivative with respect to $x_{0,1}$?

$$y_{0,2} = x_{0,0} \cdot w_{0,2} + x_{0,1} \cdot w_{1,2} + x_{0,2} \cdot w_{2,2}$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

What is $\frac{\partial y}{\partial x_{0,1}}$?

There are multiple local gradients since $x_{0,1}$ affects multiple outputs:

$$\frac{\partial y_{0,0}}{\partial x_{0,1}} = w_{1,0} = 2$$

$$\frac{\partial y_{0,1}}{\partial x_{0,1}} = w_{1,1} = 1$$

$$\frac{\partial y_{0,2}}{\partial x_{0,1}} = w_{1,2} = 3$$

$$\frac{\partial y_{0,3}}{\partial x_{0,1}} = w_{1,3} = 2$$

$$\frac{\partial y_{1,m}}{\partial x_{0,1}} = 0 \quad (x_{0,1} \text{ doesn't affect row 1})$$

What is $\frac{\partial L}{\partial x_{0,1}}$?

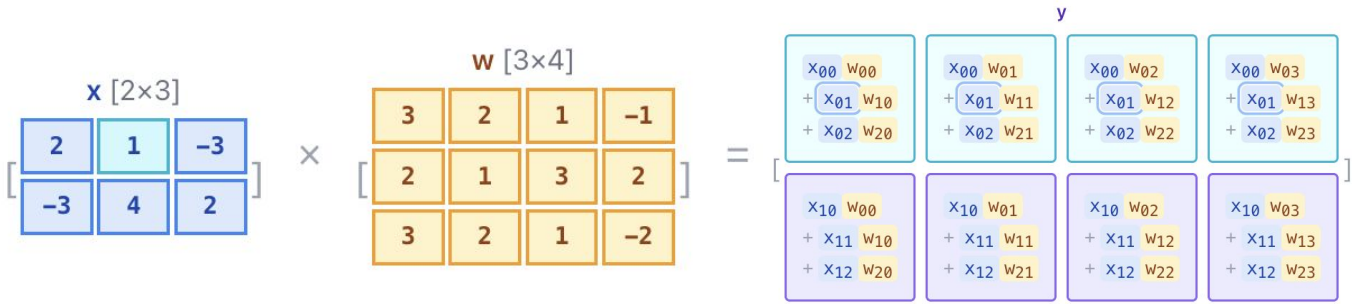
Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$= w[\text{row } 1] \cdot \frac{\partial L\ [\text{row } 0]}{\partial y}$$



$$\frac{\partial L}{\partial x_{0,1}} = \frac{\partial L}{\partial y}[\text{row } 0] \cdot w[\text{row } 1]$$

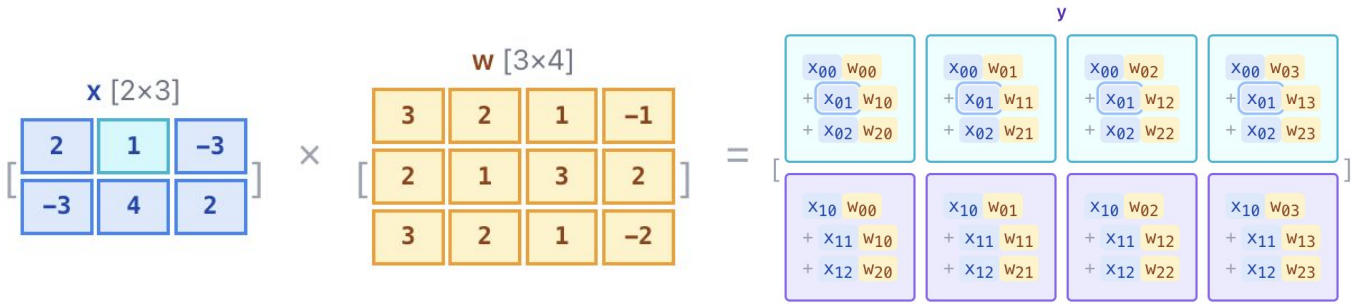
What is $\frac{\partial L}{\partial x_{0,1}}$?

Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$
$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$= w[\text{row } 1] \cdot \frac{\partial L}{\partial y}[\text{row } 0]$$



$$\frac{\partial L}{\partial x_{0,1}} = \frac{\partial L}{\partial y}[\text{row } 0] \cdot w[\text{row } 1]$$

$$\frac{\partial L}{\partial x_{n,d}} = \frac{\partial L}{\partial y}[\text{row } n] \cdot w[\text{row } d]$$

What is $\frac{\partial L}{\partial x_{0,1}}$?

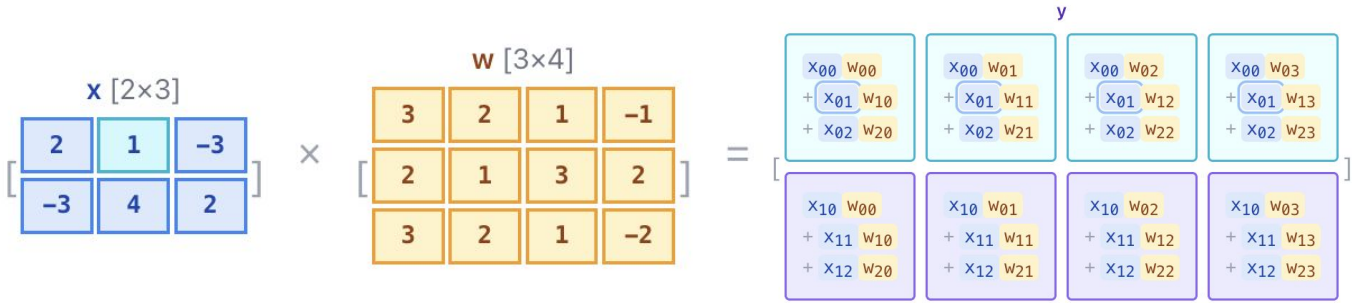
Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$= w[\text{row } 1] \cdot \frac{\partial L}{\partial y}[\text{row } 0]$$



$$\frac{\partial L}{\partial x_{0,1}} = \frac{\partial L}{\partial y}\text{[row 0]} \cdot w\text{[row 1]}$$

$$\frac{\partial L}{\partial x_{n,d}} = \frac{\partial L}{\partial y}\text{[row n]} \cdot w\text{[row d]}$$

$$\frac{\partial L}{\partial x}\ [2\times3] = \begin{bmatrix} \frac{\partial L}{\partial y_0}\cdot w[0] & \frac{\partial L}{\partial y_0}\cdot w[1] & \frac{\partial L}{\partial y_0}\cdot w[2] \\ \frac{\partial L}{\partial y_1}\cdot w[0] & \frac{\partial L}{\partial y_1}\cdot w[1] & \frac{\partial L}{\partial y_1}\cdot w[2] \end{bmatrix}$$

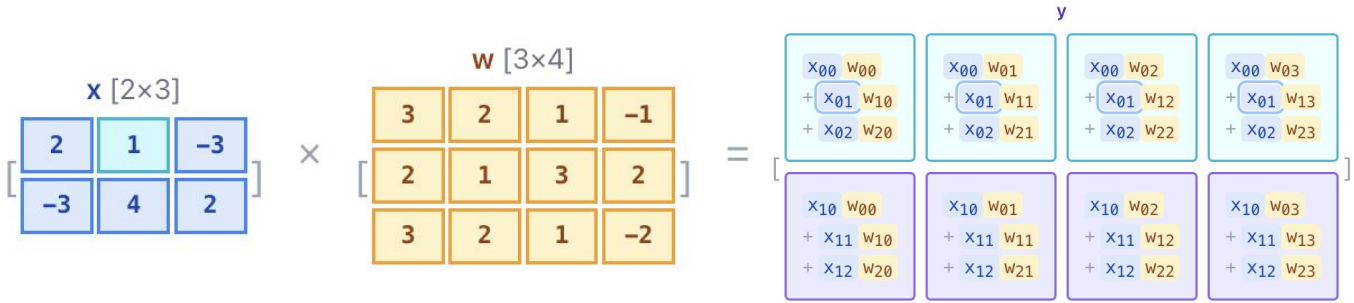
What is $\frac{\partial L}{\partial x_{0,1}}$?

Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$
$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

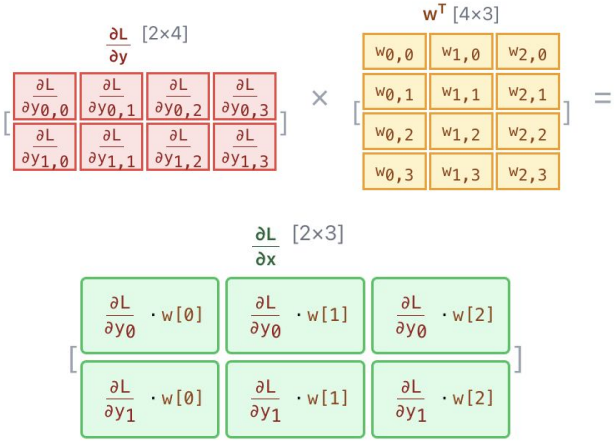
$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$= w\text{[row 1]} \cdot \frac{\partial L}{\partial y}\text{[row 0]}$$



$\partial L/\partial y$ [2×4]

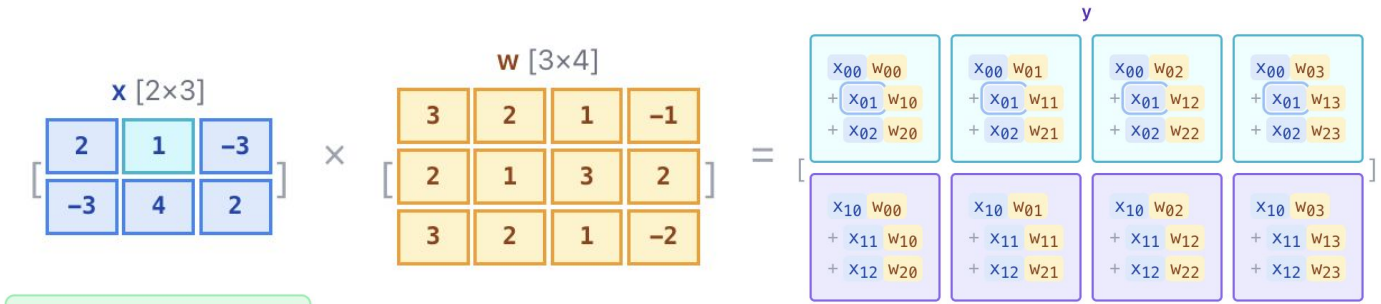
2	3	−3	9
−8	1	4	6



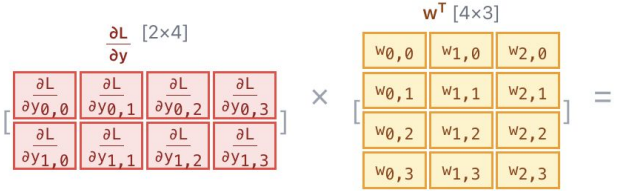
What is $\frac{\partial L}{\partial x_{0,1}}$?

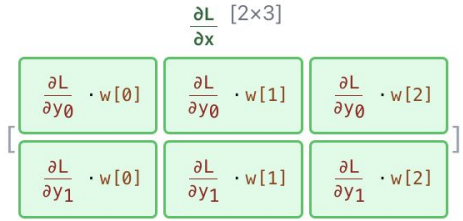
Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$
$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$
$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$
$$= \text{w[row 1]} \cdot \frac{\partial L}{\partial y}\text{[row 0]}$$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W^T$$





What is $\frac{\partial L}{\partial x_{0,1}}$?

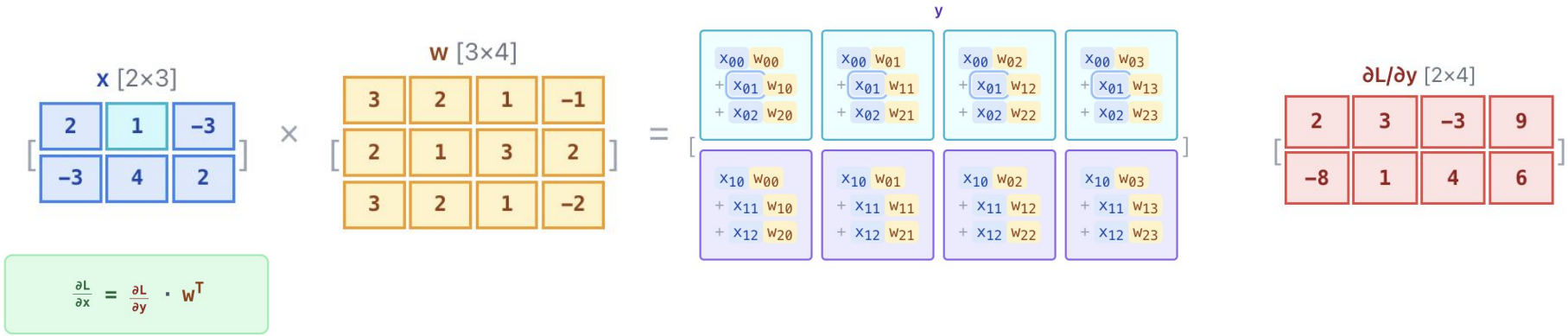
Sum of: (how much $x_{0,1}$ affects each y) × (how much each y affects L)

$$= \frac{\partial y_{0,0}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,0}} + \frac{\partial y_{0,1}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,1}} + \frac{\partial y_{0,2}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,2}} + \frac{\partial y_{0,3}}{\partial x_{0,1}} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$+ 0 \cdot \frac{\partial L}{\partial y_{1,0}} + 0 \cdot \frac{\partial L}{\partial y_{1,1}} + 0 \cdot \frac{\partial L}{\partial y_{1,2}} + 0 \cdot \frac{\partial L}{\partial y_{1,3}}$$

$$= w_{1,0} \cdot \frac{\partial L}{\partial y_{0,0}} + w_{1,1} \cdot \frac{\partial L}{\partial y_{0,1}} + w_{1,2} \cdot \frac{\partial L}{\partial y_{0,2}} + w_{1,3} \cdot \frac{\partial L}{\partial y_{0,3}}$$

$$= w[\text{row } 1] \cdot \frac{\partial L}{\partial y}[\text{row } 0]$$



$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot w^T$$

Summary: Two Ways to Compute the Same Thing

For $\mathbf{y = x \cdot w}$ where x is [N×D], w is [D×M], y is [N×M]:

Explicit Jacobian Approach

$$\frac{\partial L}{\partial x} = \text{Jacobian } \frac{\partial y}{\partial x} \cdot \frac{\partial L}{\partial y}$$

Jacobian size: **[(N·D) × (N·M)]** — mostly zeros!

Dimensions: (N·D) × (N·M) × (N·M) → (N·D)

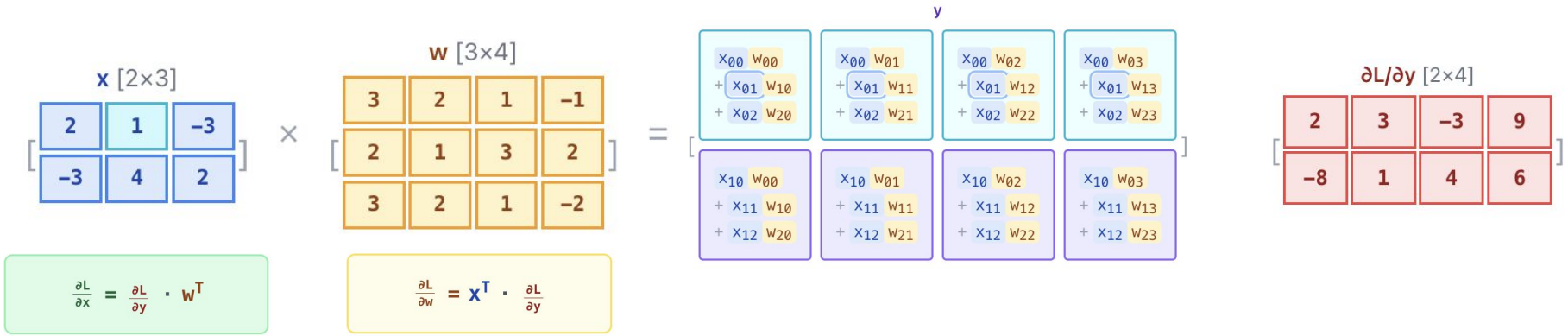
Large sparse matrix — wasteful

Implicit Matrix Multiply

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot w^T$$

Dimensions: (N × M) × (M × D) → (N × D)

Small dense matrix — efficient



Summary: Two Ways to Compute the Same Thing

For **y = x · w** where x is [N×D], w is [D×M], y is [N×M]:

Explicit Jacobian Approach

$$\frac{\partial L}{\partial x} = \text{Jacobian } \frac{\partial y}{\partial x} \cdot \frac{\partial L}{\partial y}$$

Jacobian size: **[(N·D) × (N·M)]** — mostly zeros!

Dimensions: (N·D) × (N·M) × (N·M) → (N·D)

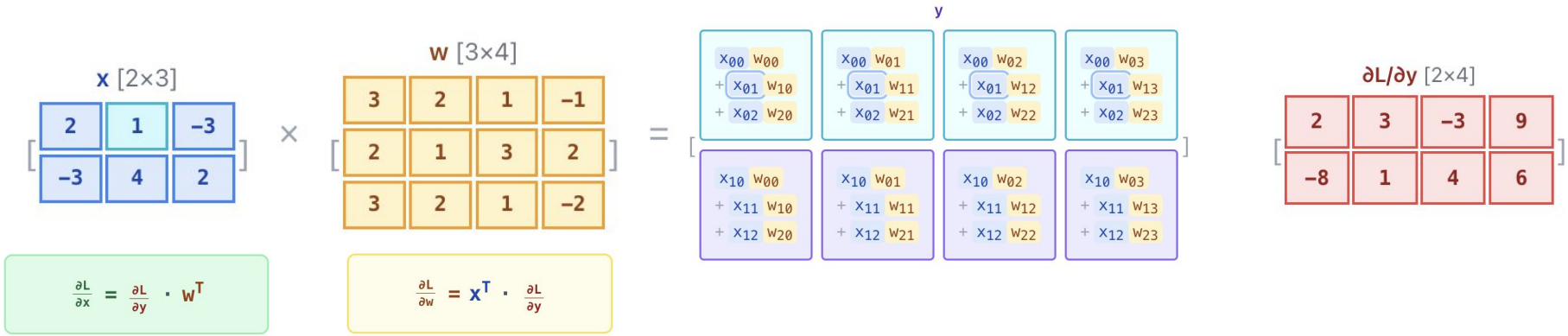
Large sparse matrix — wasteful

Implicit Matrix Multiply

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot w^T$$

Dimensions: (N × M) × (M × D) → (N × D)

Small dense matrix — efficient



These formulas are easy to remember: they are the only way to make shapes match up!

Summary: Two Ways to Compute the Same Thing

For **y = x · w** where x is [N×D], w is [D×M], y is [N×M]:

Explicit Jacobian Approach

$\frac{\partial L}{\partial x} = \text{Jacobian } \frac{\partial y}{\partial x} \cdot \frac{\partial L}{\partial y}$

Jacobian size: **[(N·D) × (N·M)]** — mostly zeros!

Dimensions: (N·D) × (N·M) × (N·M) → (N·D)

Large sparse matrix — wasteful

Implicit Matrix Multiply

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot w^T$

Dimensions: (N × M) × (M × D) → (N × D)

Small dense matrix — efficient

Backprop with Matrices

x: [N×D]

2	1	−3
−3	4	2

w: [D×M]

3	2	1	−1
2	1	3	2
3	2	1	−2

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y: [N×M]

−1	−1	2	6
5	2	11	1

dL/dy: [N×M]

2	3	−3	9
−8	1	4	6

[N×D] [N×M] [M×D]

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T$$

[D×M] [D×N] [N×M]

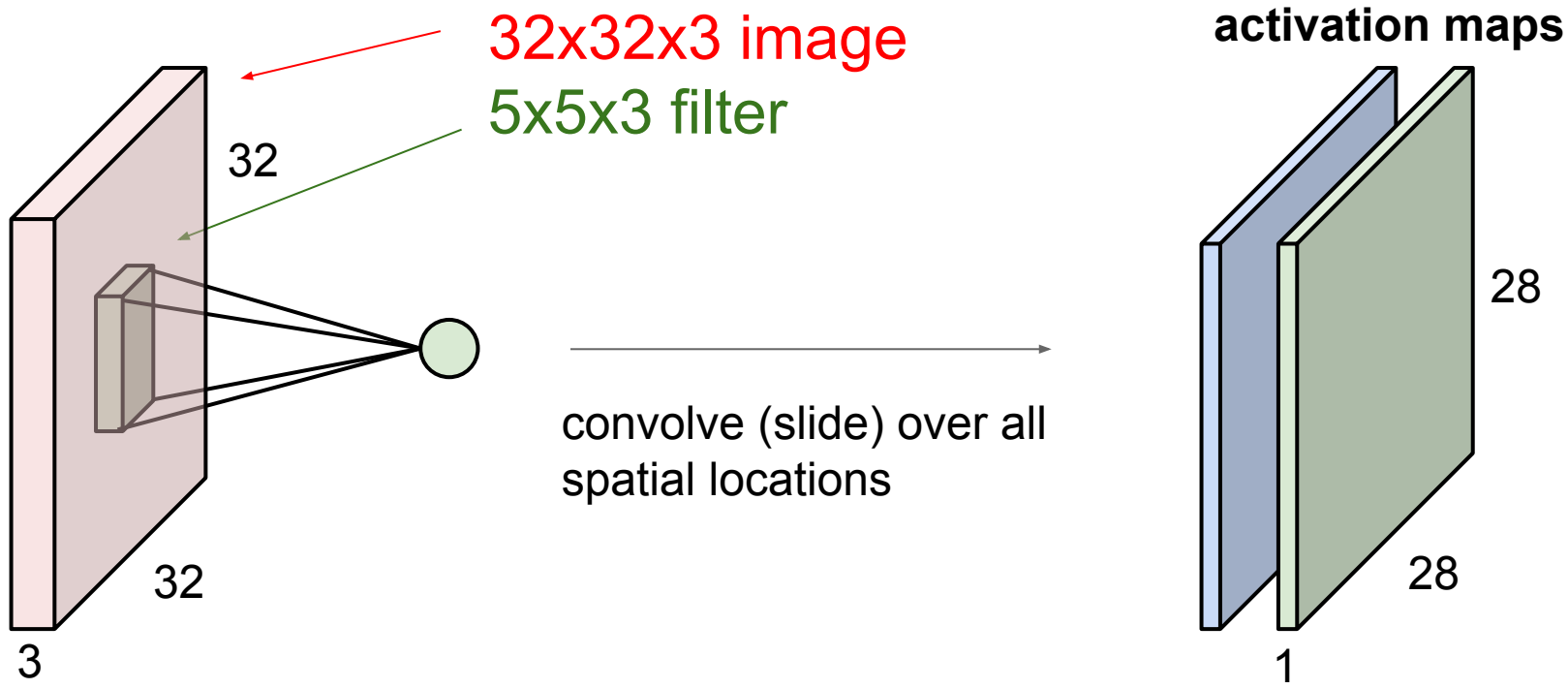
$$\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)$$

Today's agenda

- Briefly revisit backprop
- Finish CNNs
- Activation Functions
- Data Preprocessing
- Weight Initialization
- Normalization Layers

Convolution Layer

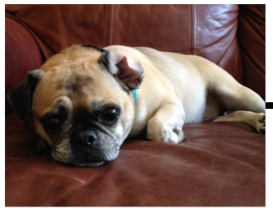
consider a second, green filter



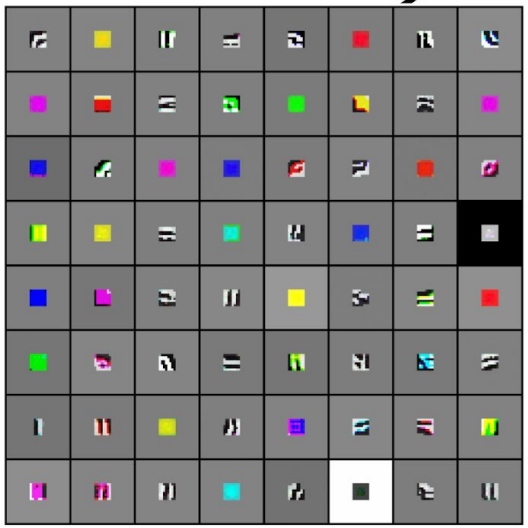
Preview

[Zeiler and Fergus 2013]

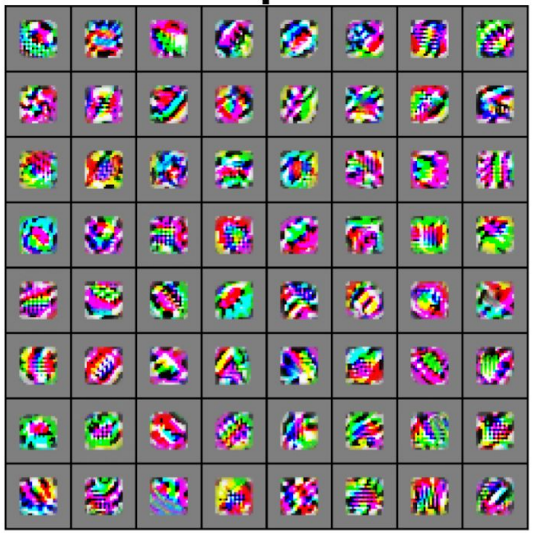
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



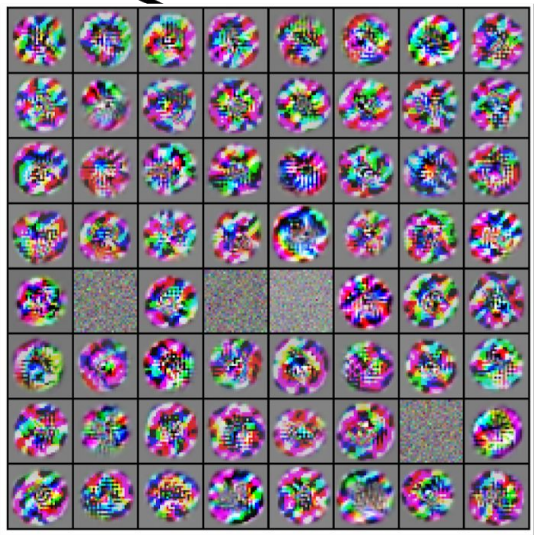
Low-level features → Mid-level features → High-level features → Linearly separable classifier



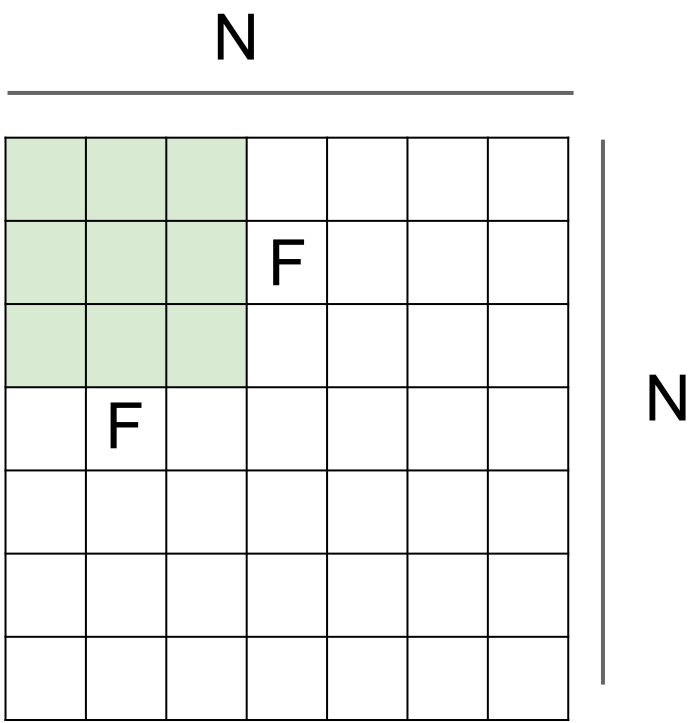
VGG-16 Conv1_1



VGG-16 Conv3_2



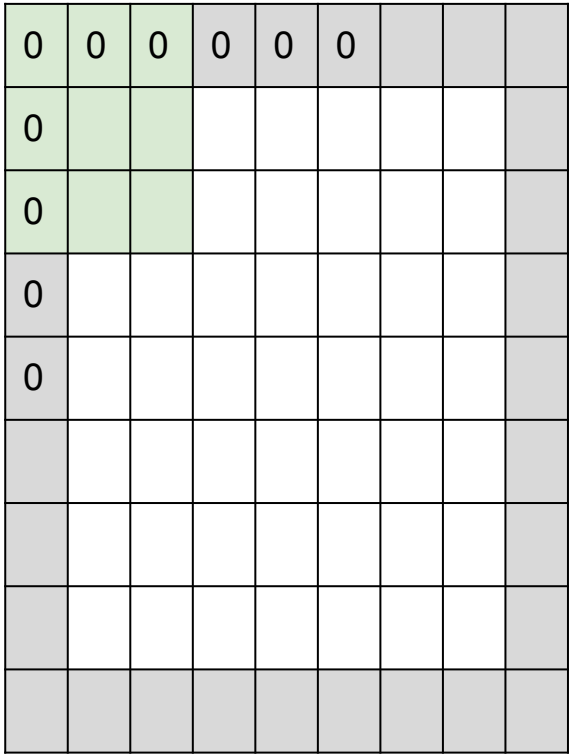
VGG-16 Conv5_3



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 :\

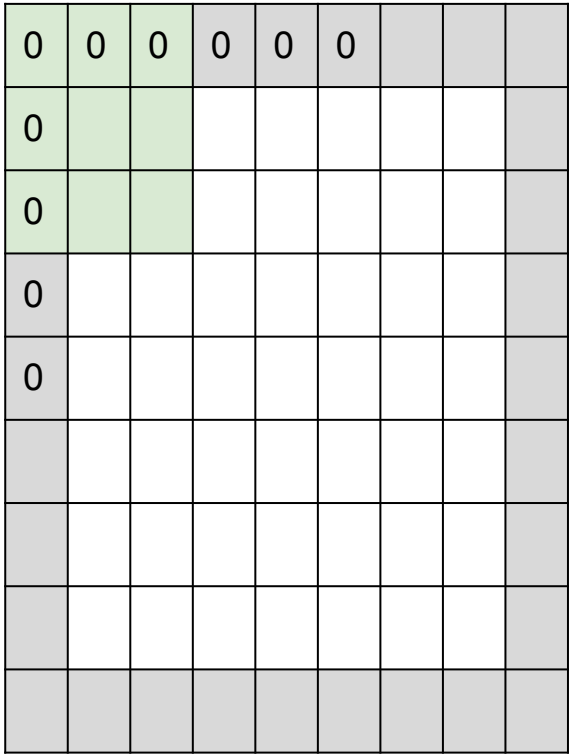
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)
(N - F) / stride + 1

In practice: Common to zero pad the border

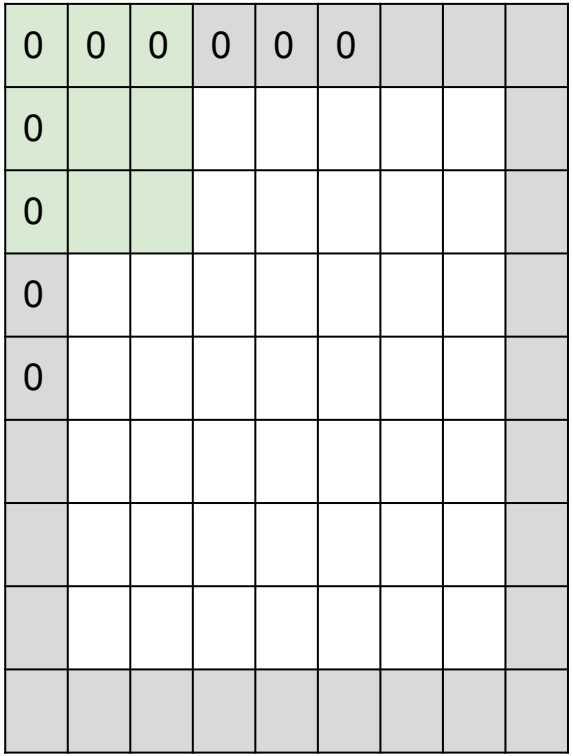


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

(recall:)
(N + 2P - F) / stride + 1

In practice: Common to zero pad the border

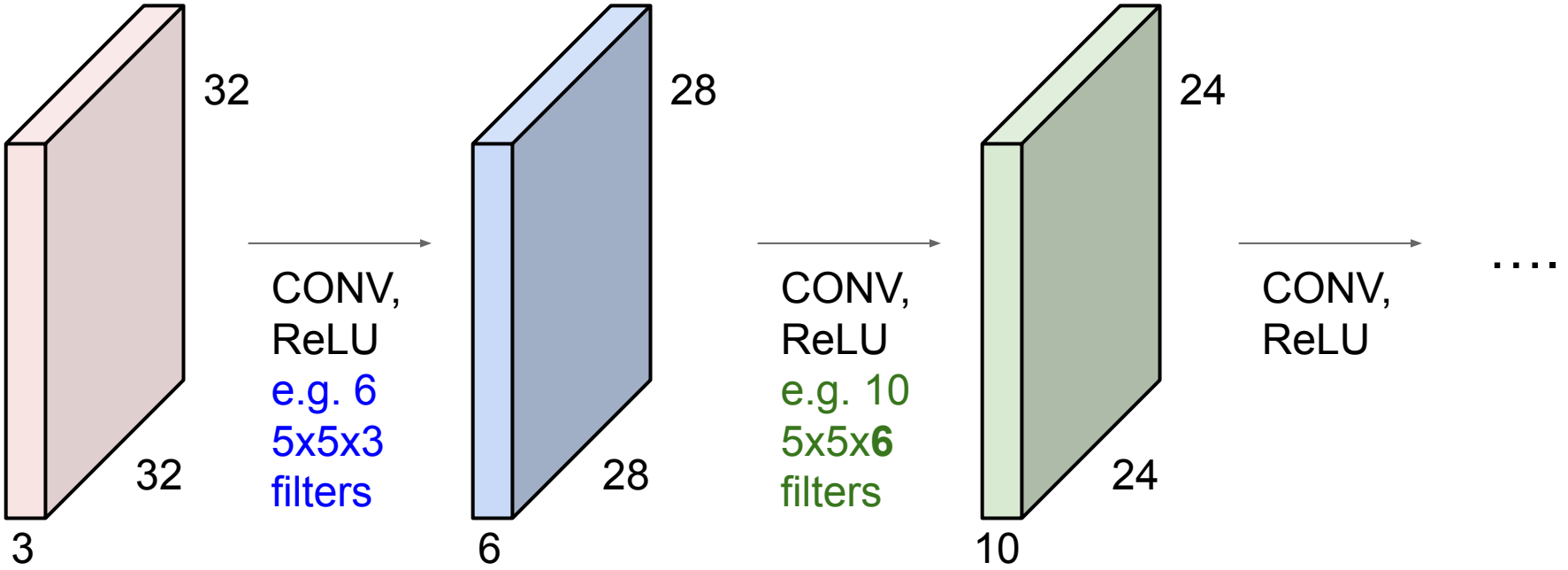


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

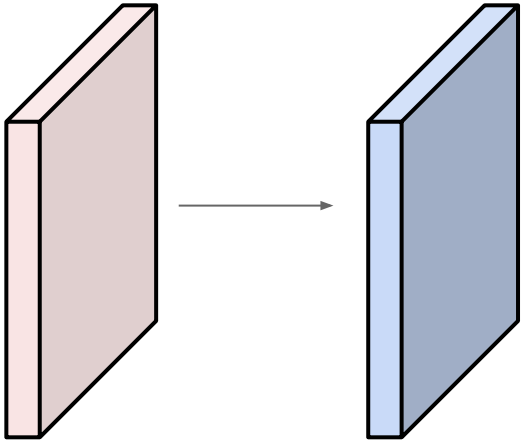
7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...).



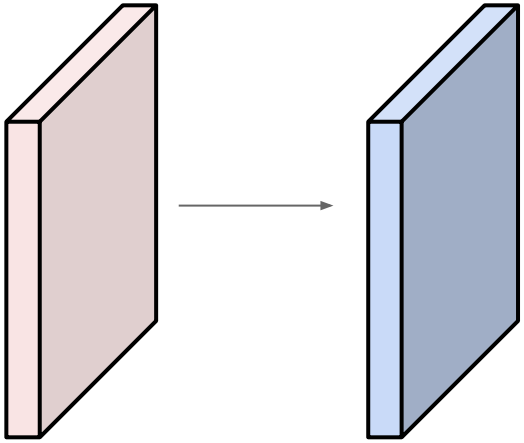
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxD.
What is D?

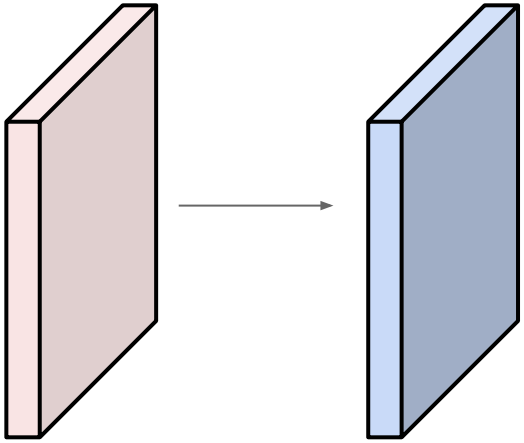
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxD.
What is D? **10**

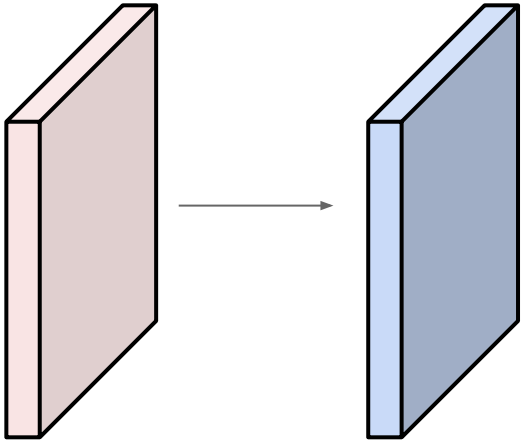
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxD.
What is D? **10**
What is H or W?

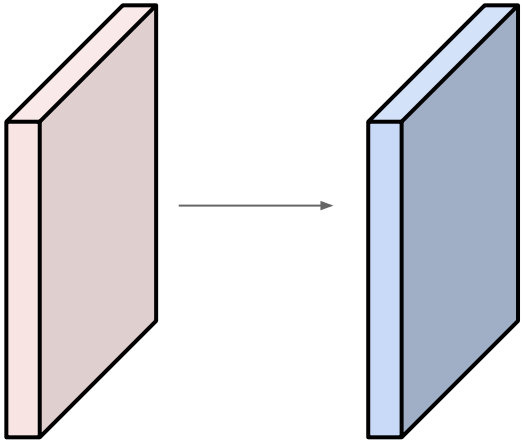
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxD.
What is D? 10
What is H or W? (32+2*2-5)/1+1 = 32

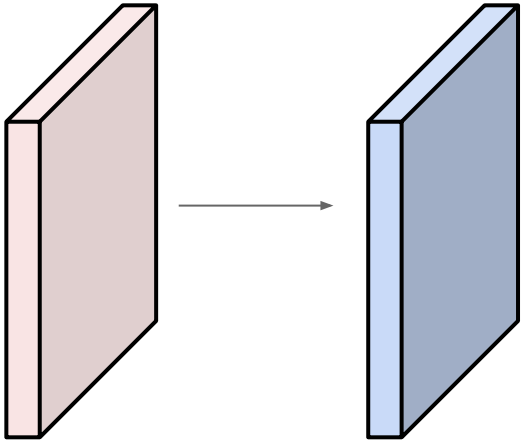
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Let's assume output size is HxWxD.
What is D? 10
What is H or W? (32+2*2-5)/1+1 = 32
So the total output size is: **32x32x10**

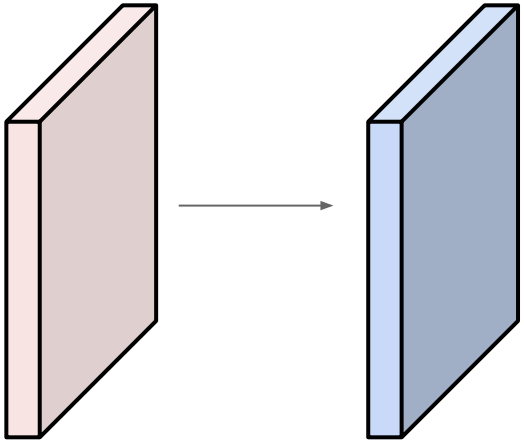
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Convolution layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K
where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: F^2KC and K biases

Convolution layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of W_2 x H_2 x K
where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

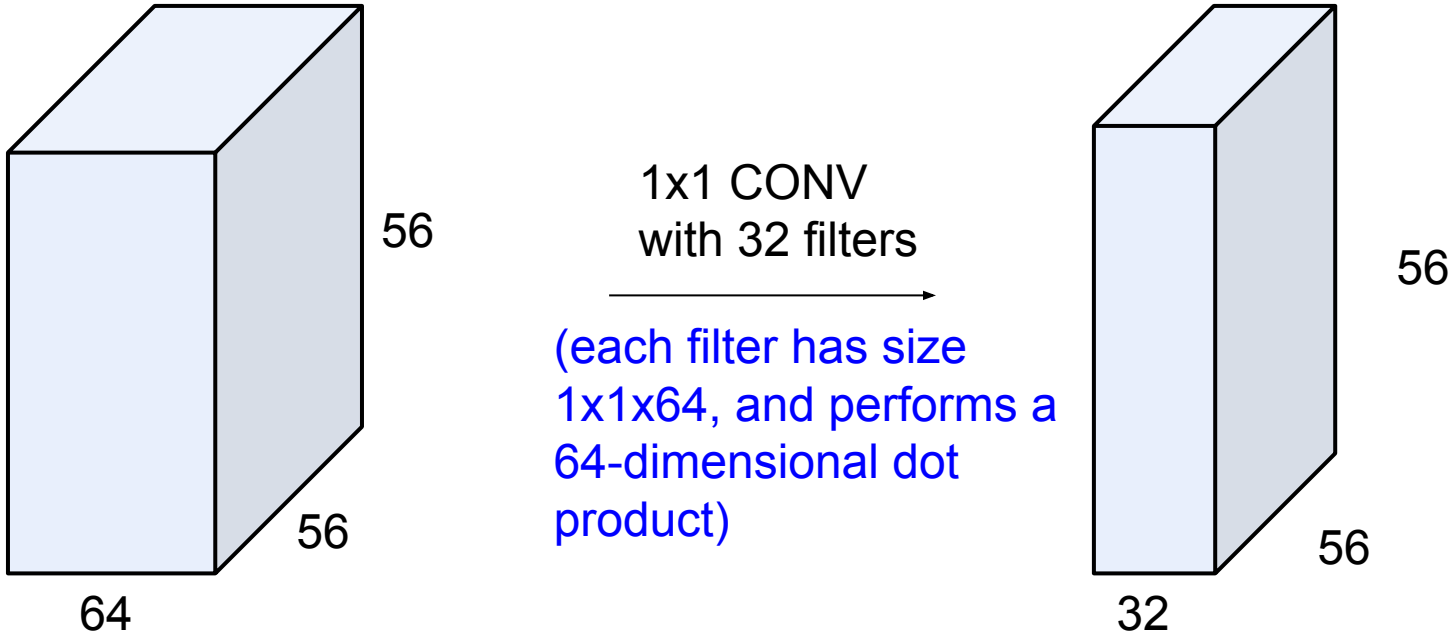
Number of parameters: F^2CK and K biases

Common settings:

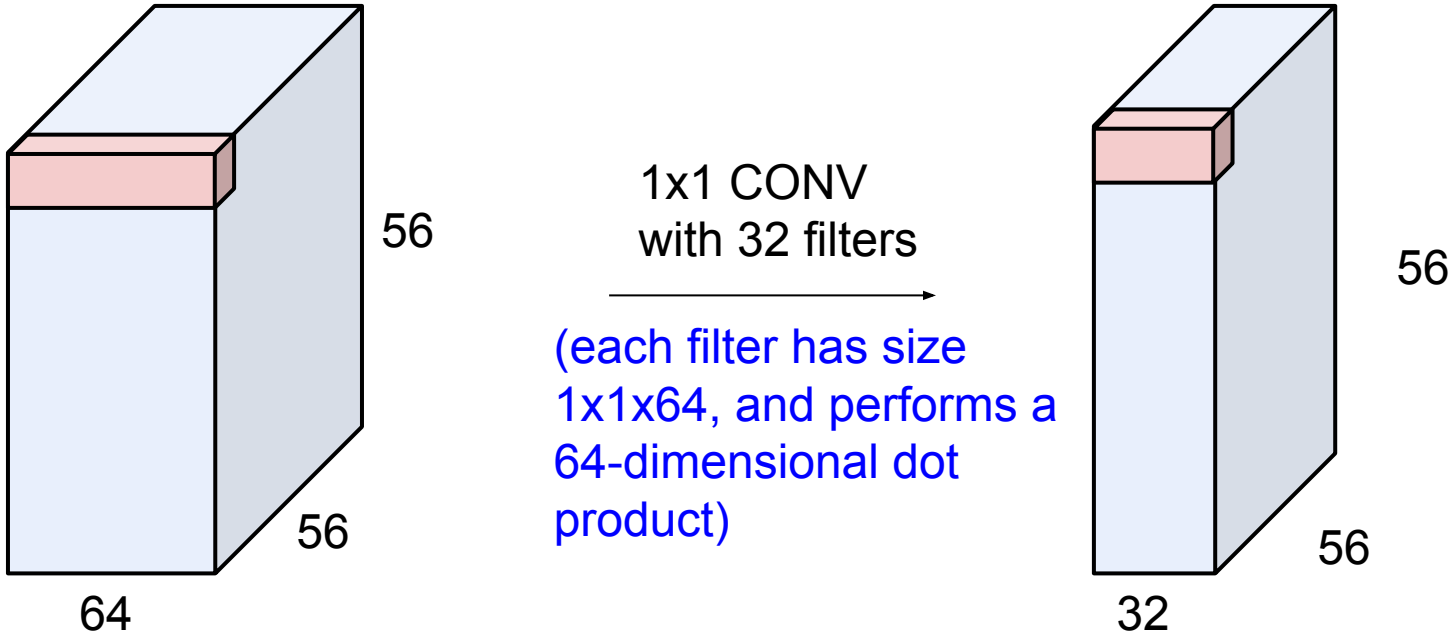
K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

(btw, 1x1 convolution layers are very useful)



(btw, 1x1 convolution layers are a very useful)



Example: CONV layer in PyTorch

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

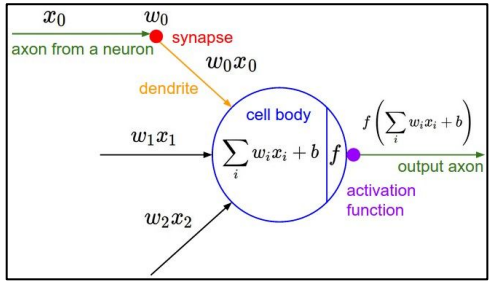
where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

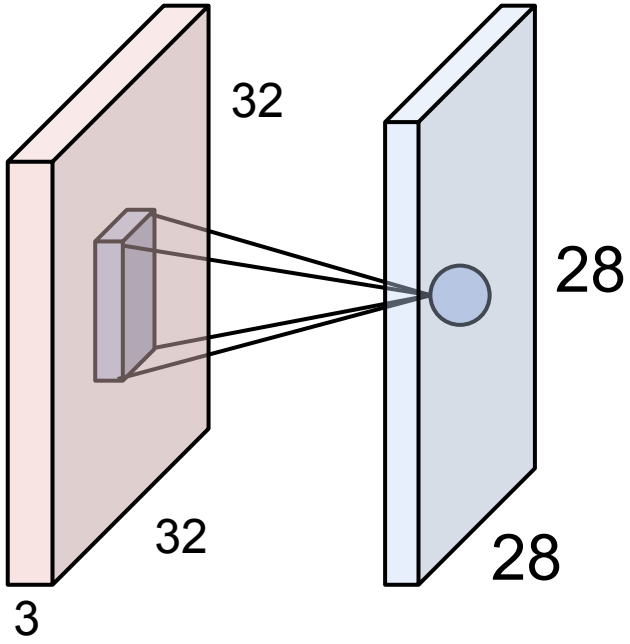
- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size: $\left\lfloor \frac{C_{\text{out}}}{C_{\text{in}}} \right\rfloor$.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

Receptive field



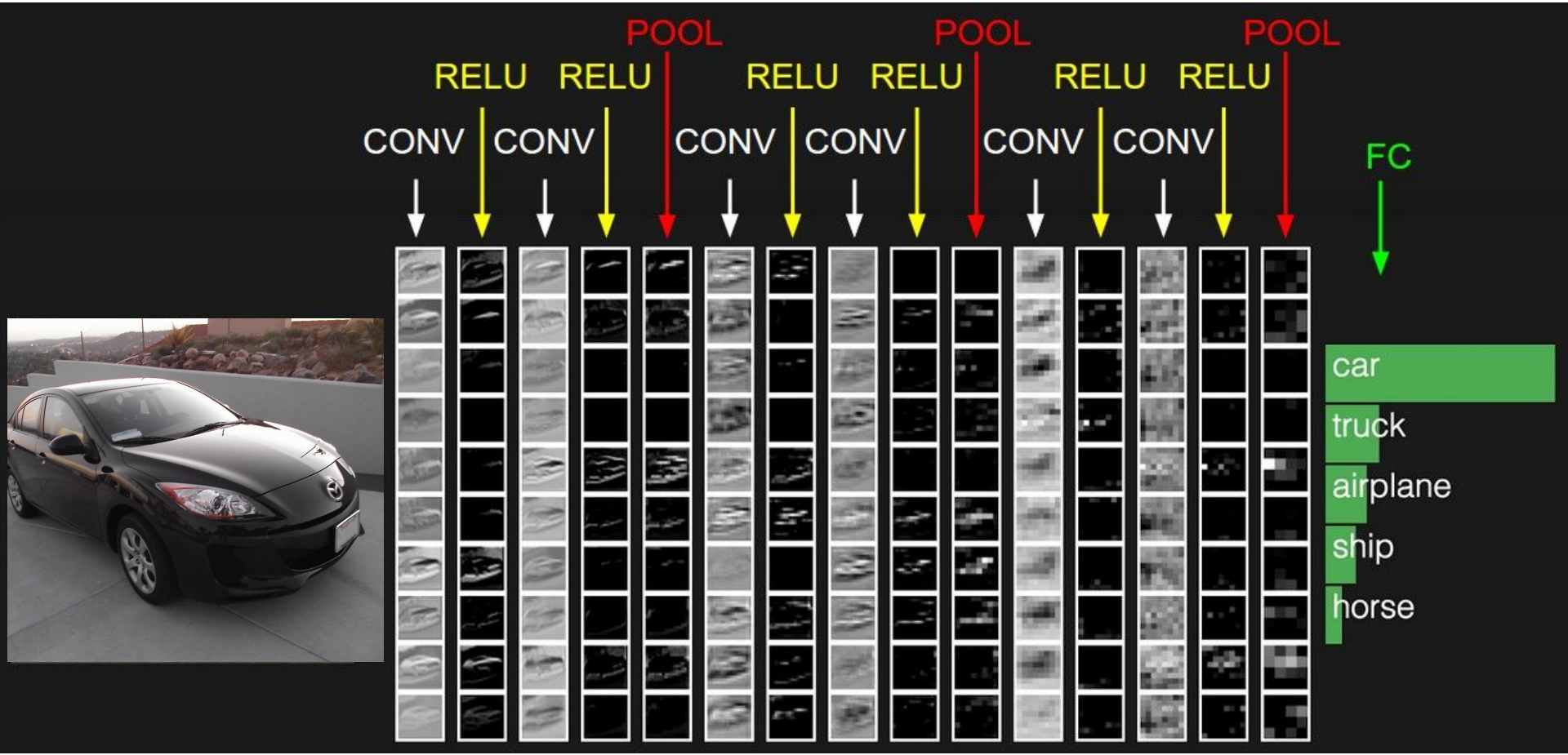


An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters

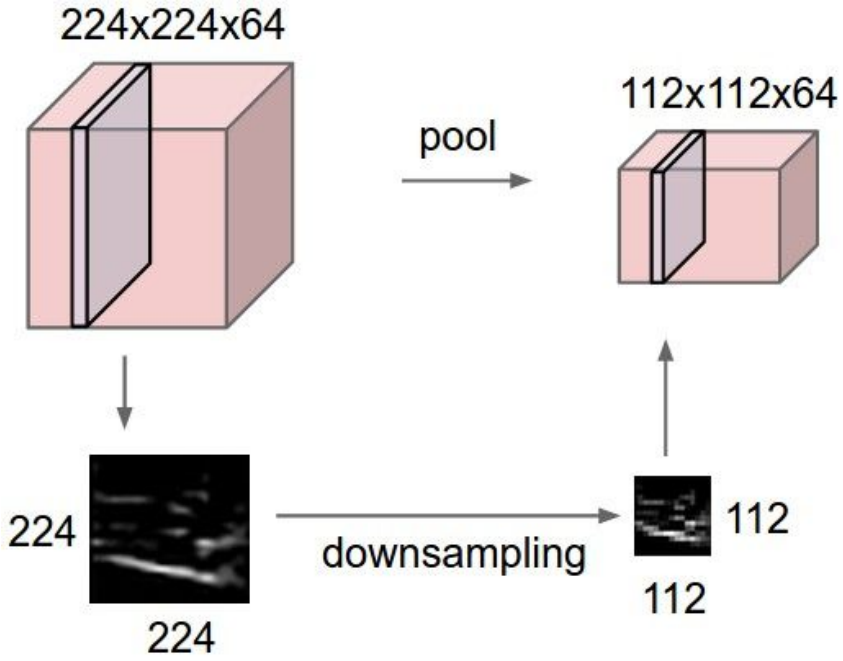
"5x5 filter" -> "5x5 receptive field for each neuron"

FOUR layers in total: CONV/ReLU/POOL/FC

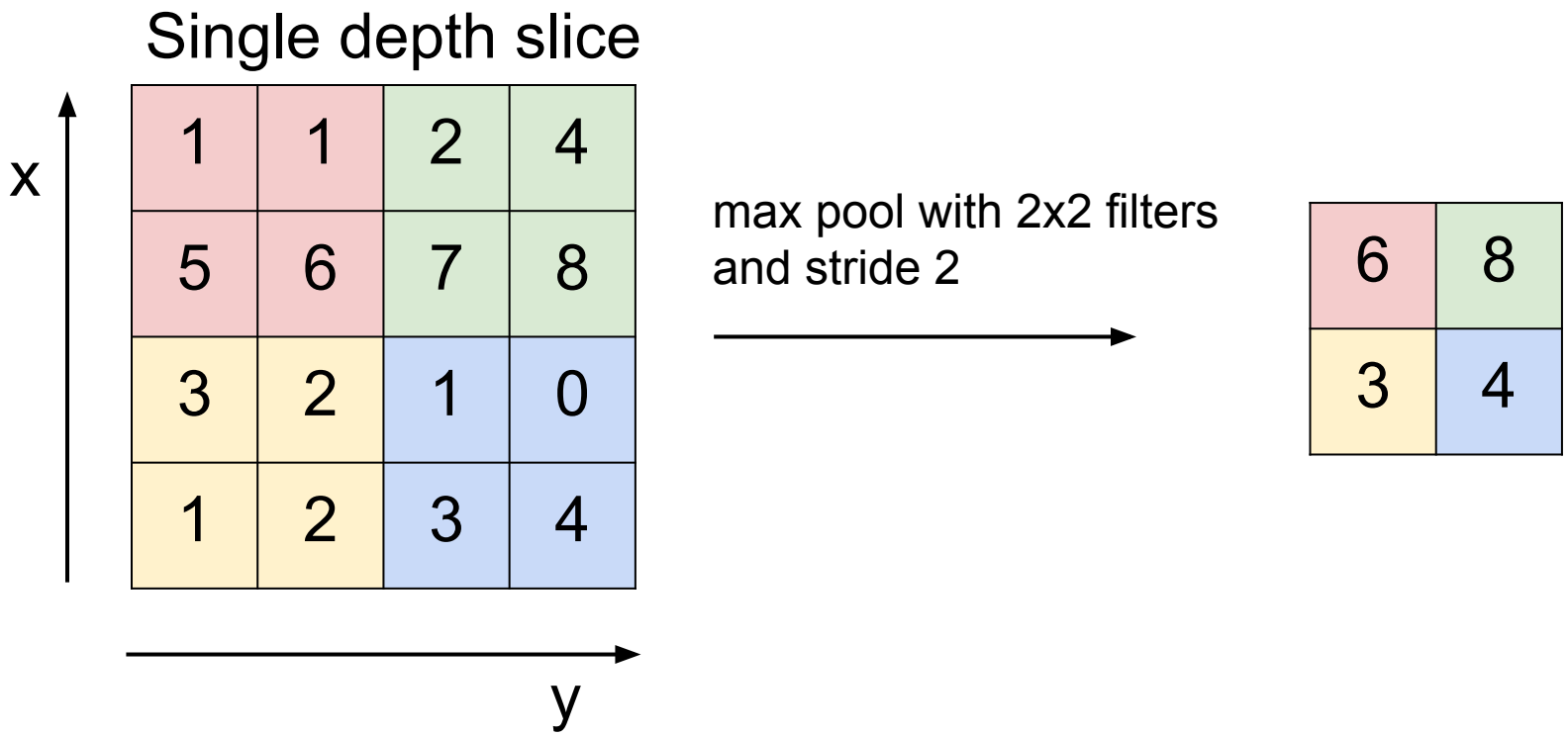


Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



MAX POOLING


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Pooling layer: summary

Let's assume input is W_1 x H_1 x C
Conv layer needs 2 hyperparameters:

- The spatial extent **F**
- The stride **S**

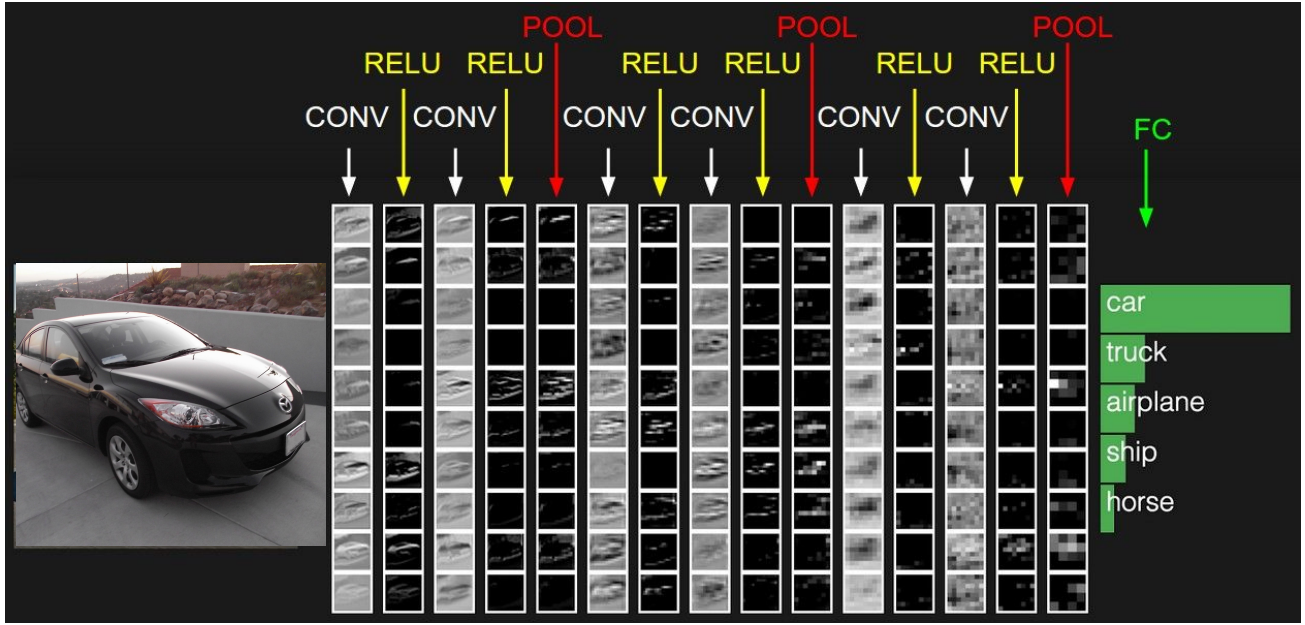
This will produce an output of W_2 x H_2 x C where:

- $W_2 = (W_1 - F)/S + 1$
- $H_2 = (H_1 - F)/S + 1$

Number of parameters: 0

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



Summary

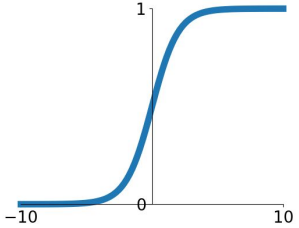
- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Between 2012-2016 architectures looked like
  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX
  where N is usually up to ~5, M is large, 0 <= K <= 2.
  - but recent advances such as ResNet/GoogLeNet have challenged this paradigm

Activation Functions

Activation Functions

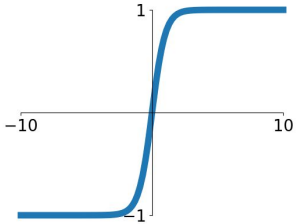
Sigmoid

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



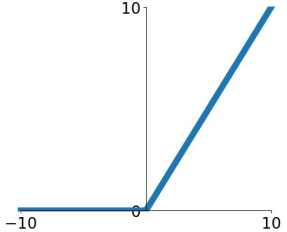
tanh

$$\tanh(x)$$



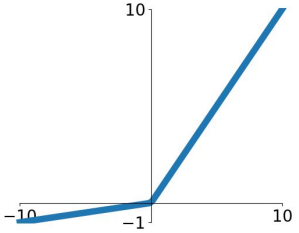
ReLU

$$\max(0, x)$$



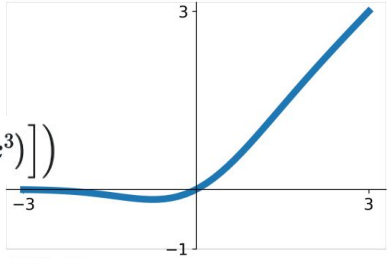
Leaky ReLU

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



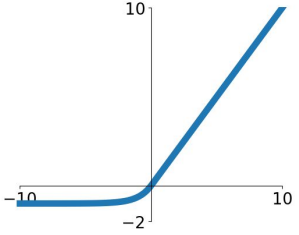
GeLU

$$0.5x\left(1+\tanh\left[\sqrt{2/\pi}\left(x+0.044715x^3\right)\right]\right)$$



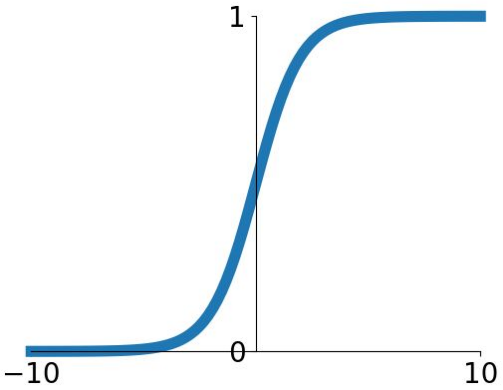
ELU

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



Sigmoid

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

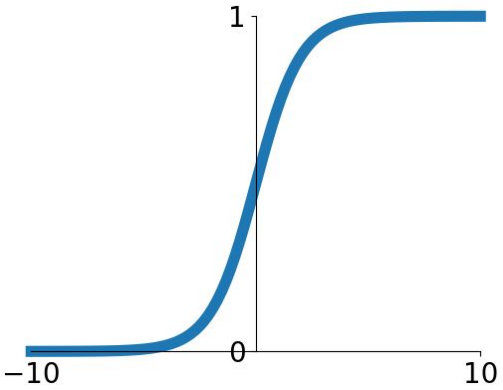


Sigmoid

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

Sigmoid

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

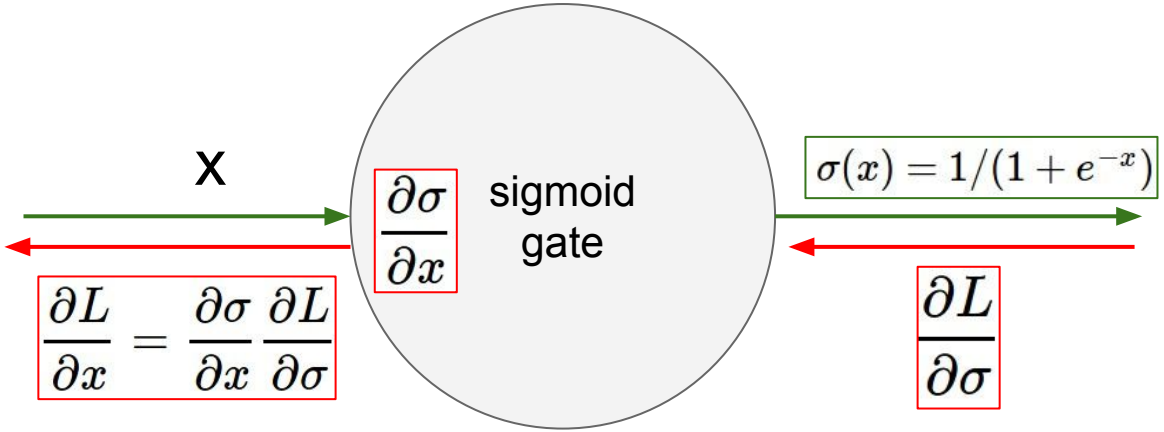


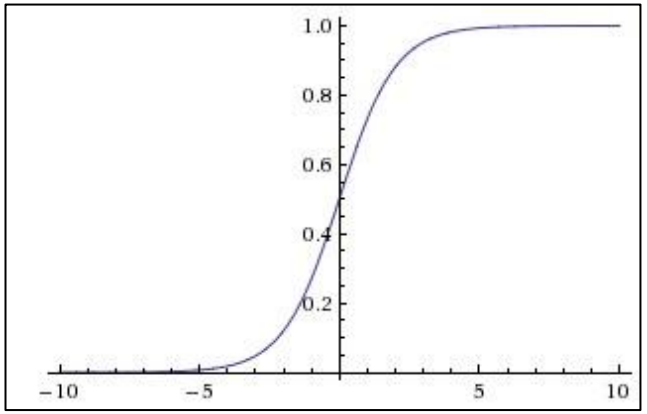
Sigmoid

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

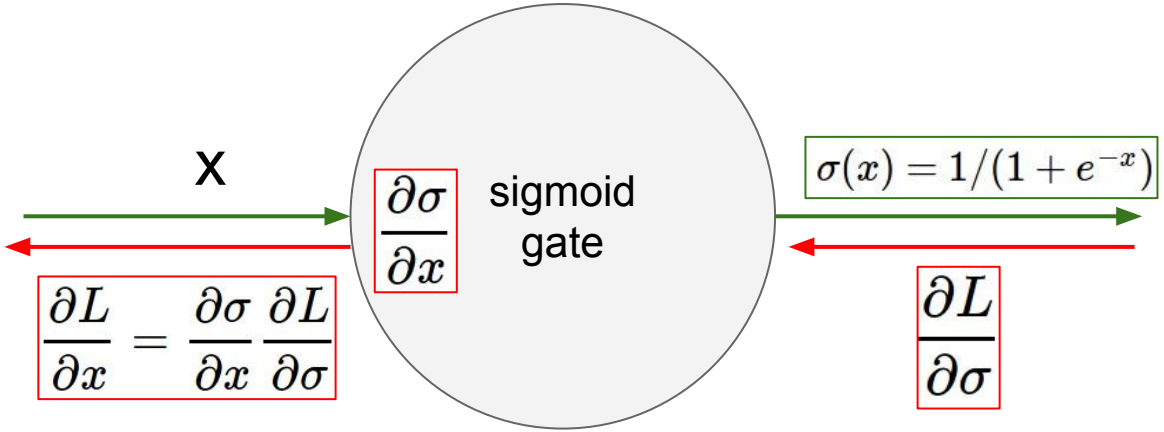
3 problems:

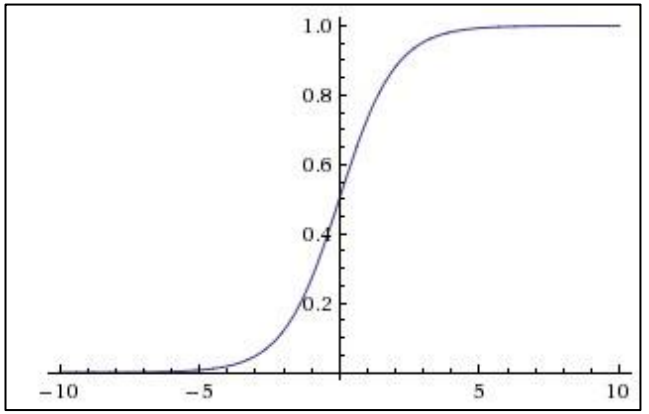
1. Saturated neurons “kill” the gradients





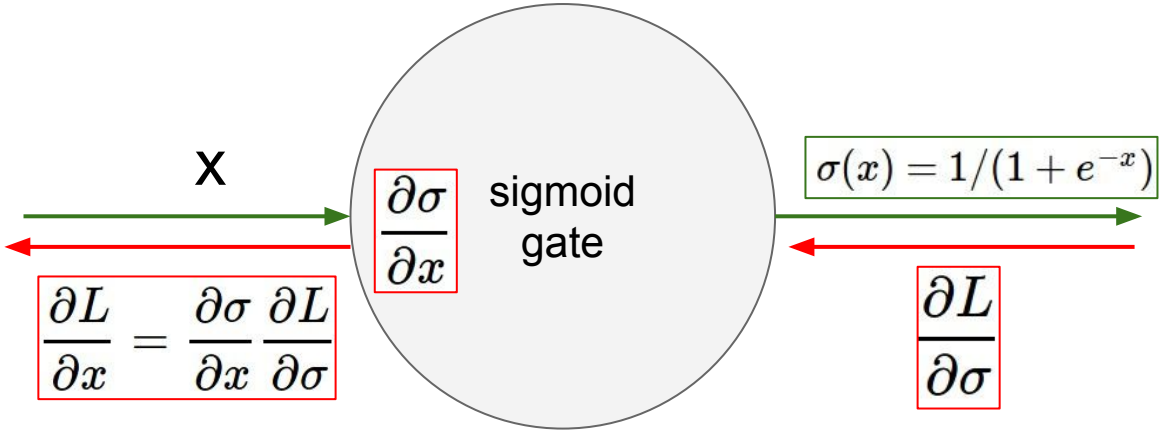
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

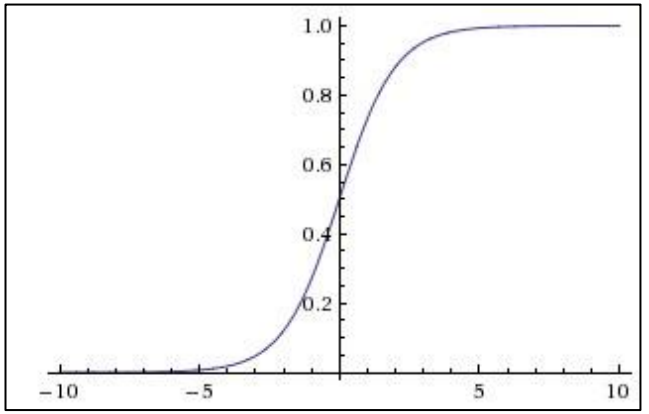




What happens when x = -10?

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



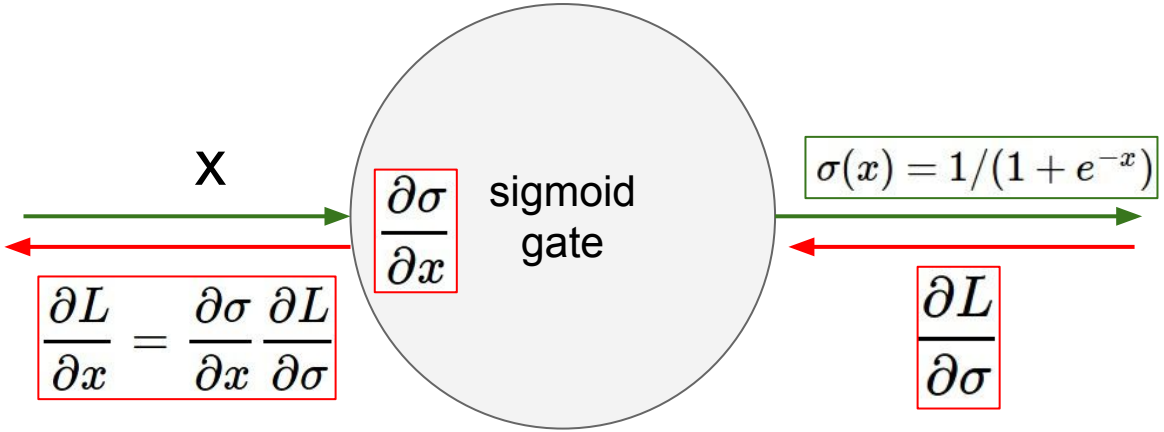


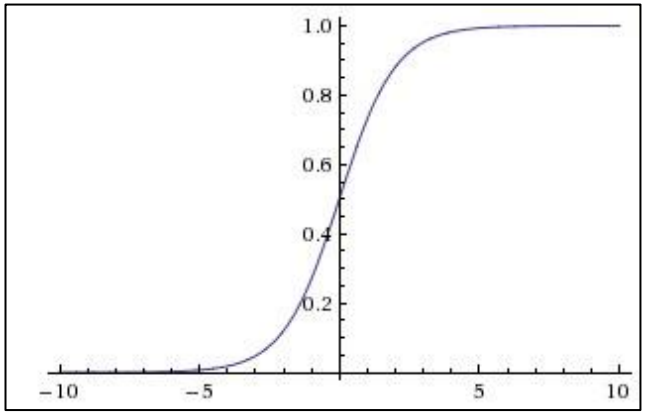
What happens when x = -10?

$$\sigma(x) = \sim 0$$

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

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

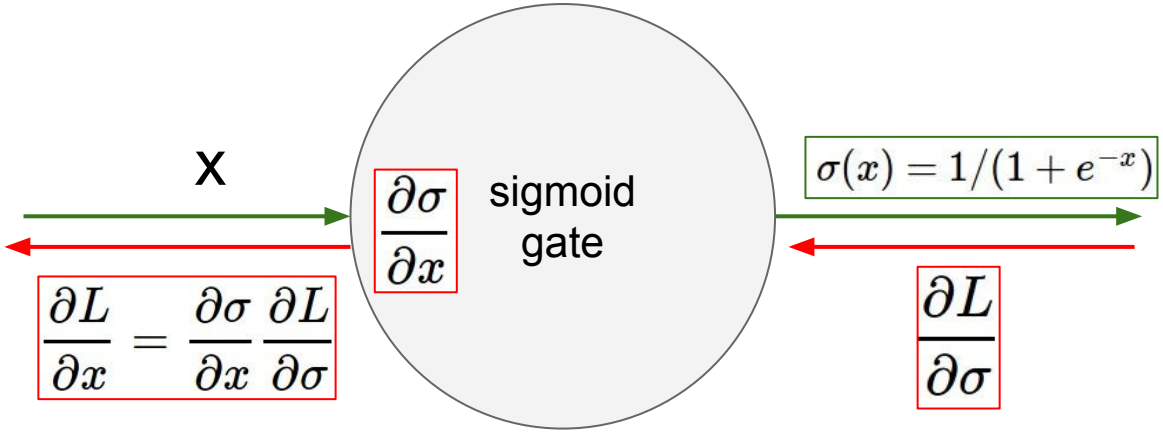


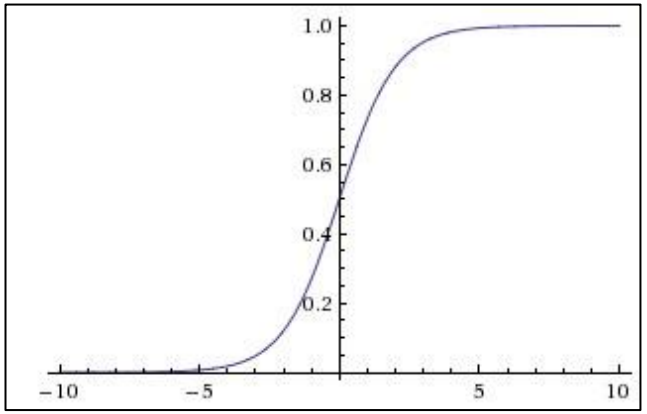


What happens when x = -10?

What happens when x = 0?

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



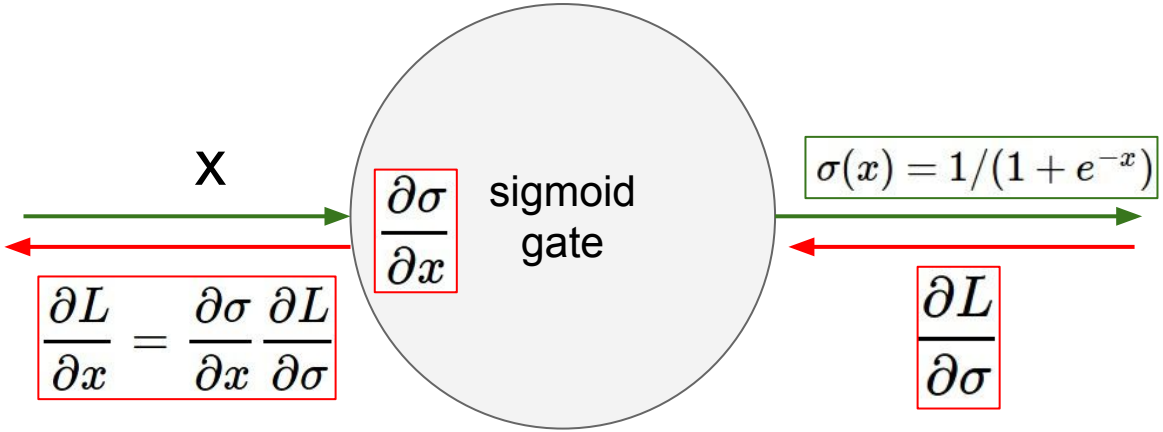


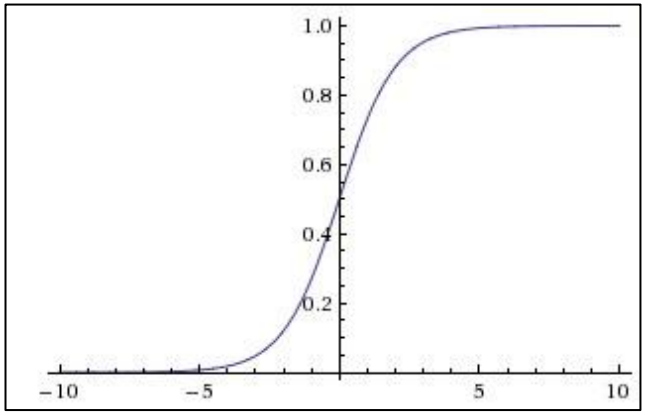
What happens when x = -10?

What happens when x = 0?

What happens when x = 10?

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





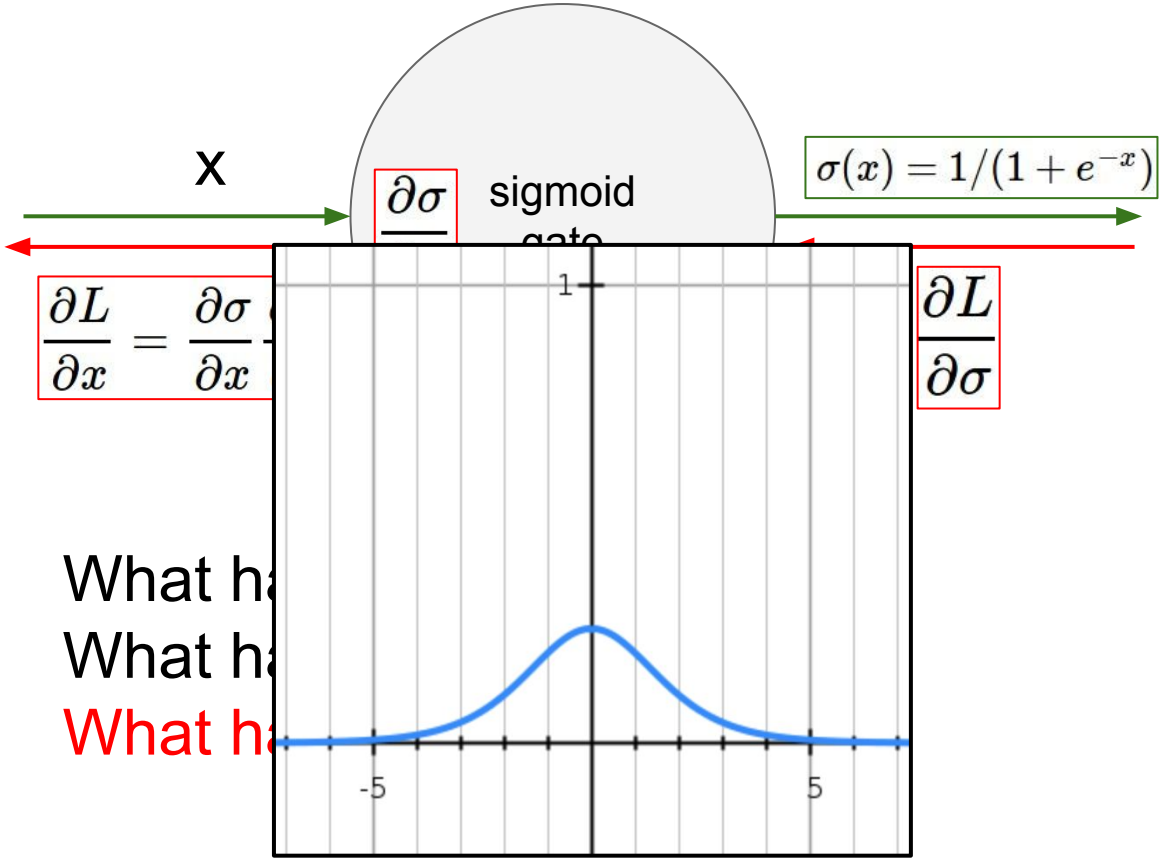
What happens when x = -10?

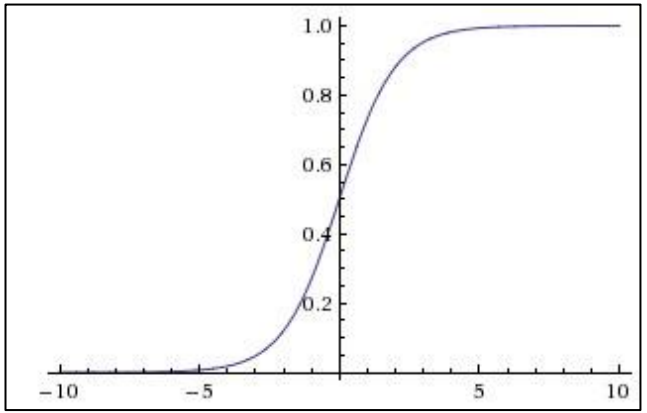
What happens when x = 0?

What happens when x = 10?

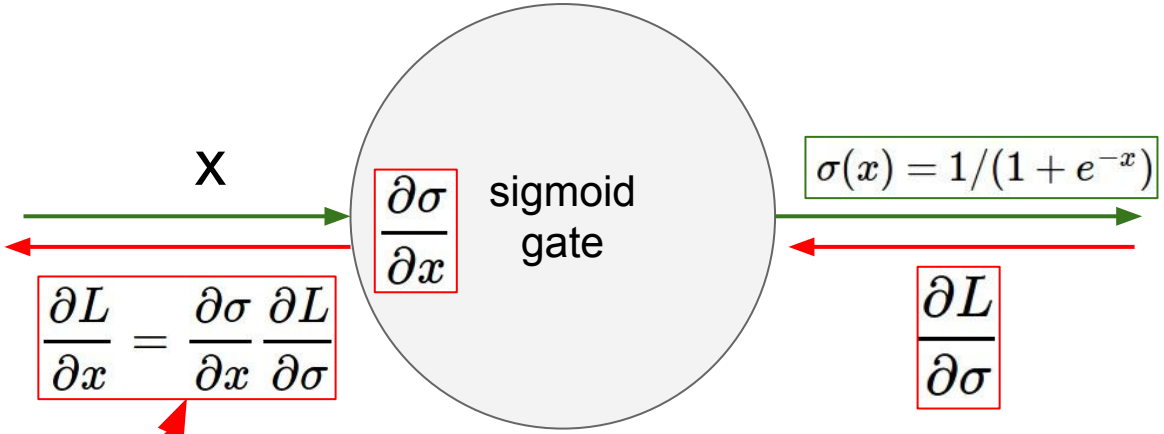
$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x))$$

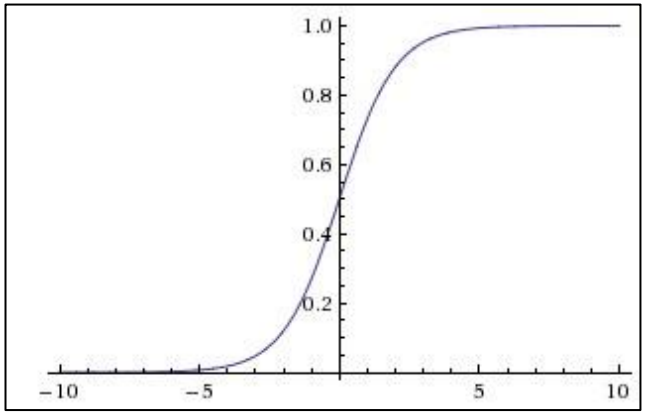
$$\sigma(x) = \sim 1 \qquad \frac{\partial \sigma(x)}{\partial x} = \sigma(x)\,(1 - \sigma(x)) = 1(1-1) = 0$$





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





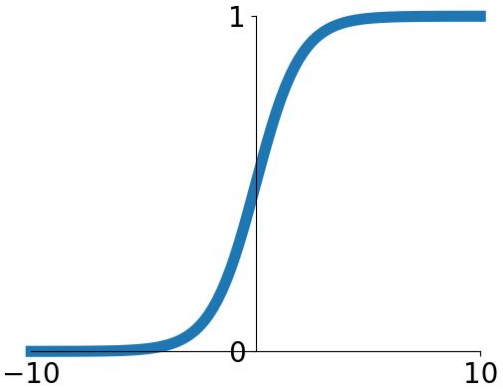
Why is this a problem?

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

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

Sigmoid

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



Sigmoid

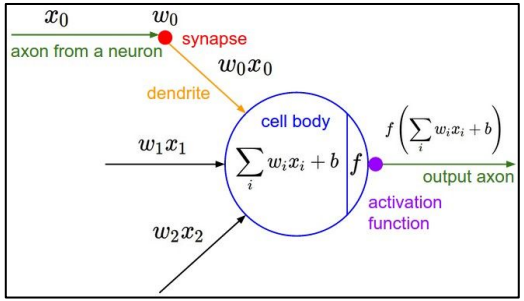
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

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...

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



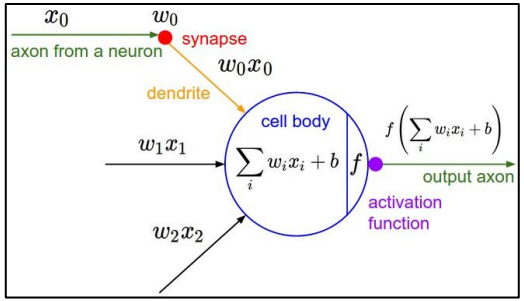
What can we say about the gradients for that neuron (i.e. a row of **w)**?

Neuron/row = (vector size n)
Input = x (vector size n, all positive)
Output = z (what shape?)
dL/dz = (what shape?)
dL/dw = (what shape?)

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \cdot \frac{\partial L}{\partial z}$$

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 for that neuron (i.e. a row of **w)**?

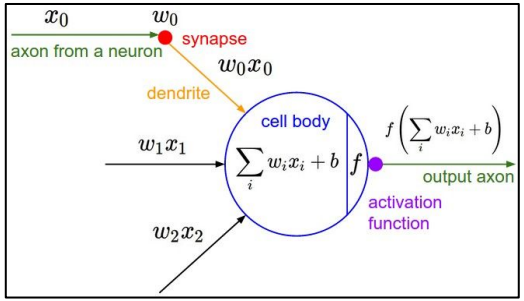
Neuron/row = (vector size n)
Input = x (vector size n, all positive)
Output = z (what shape?)
dL/dz = (what shape?)
dL/dw = (what shape?)

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \cdot \boxed{\frac{\partial L}{\partial z}}$$

Scalar (+ or -)

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 for that neuron (i.e. a row of **w)**?

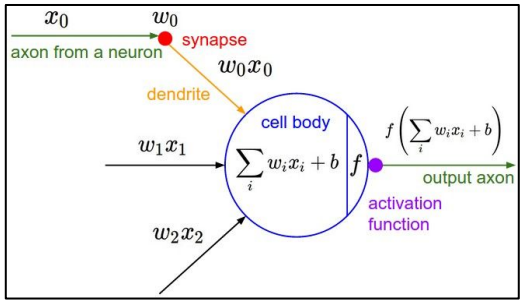
Neuron/row = (vector size n)
Input = x (vector size n, all positive)
Output = z (what shape?)
dL/dz = (what shape?)
dL/dw = (what shape?)

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \cdot \frac{\partial L}{\partial z}$$

X (all +) Scalar (+ or -)

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 for that neuron (i.e. a row of **w)**?

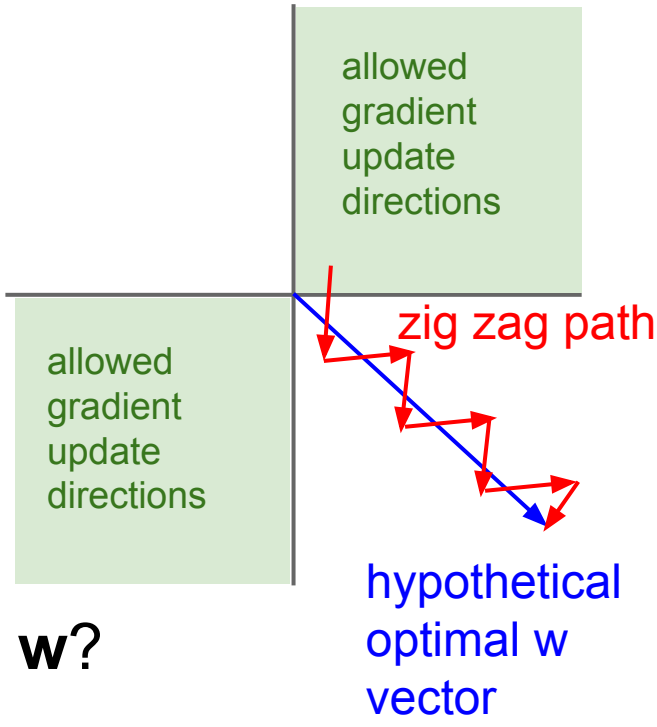
Neuron/row = (vector size n)
Input = x (vector size n, all positive)
Output = z (what shape?)
dL/dz = (what shape?)
dL/dw = (what shape?)

$$\frac{\partial L}{\partial w} = \frac{\partial z}{\partial w} \cdot \frac{\partial L}{\partial z}$$

(all + or -) X (all +) Scalar (+ or -)

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

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

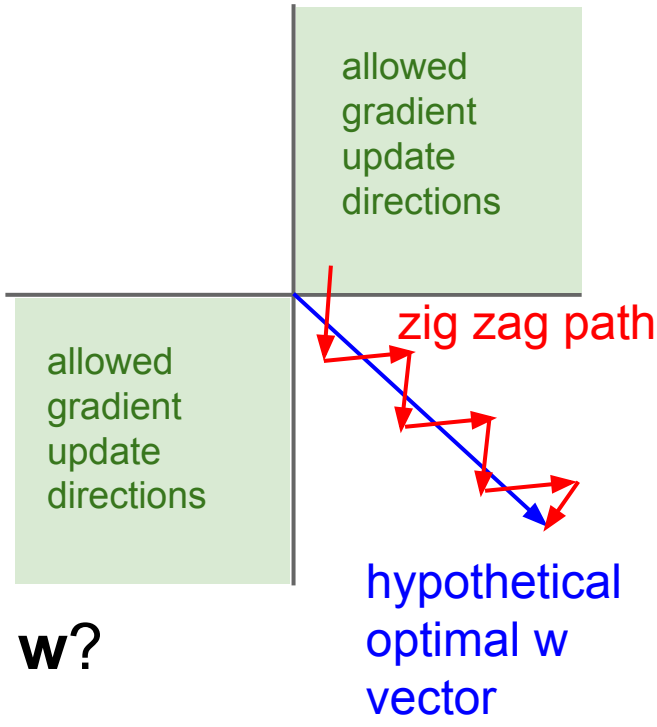


What can we say about the gradients on **w**?

Always all positive or all negative :(

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

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



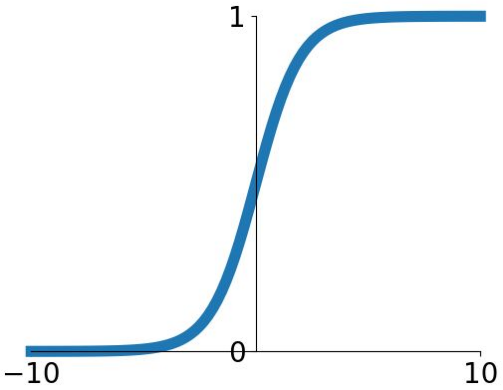
What can we say about the gradients on **w**?

Always all positive or all negative :(

(For a single element! Minibatches help)

Sigmoid

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



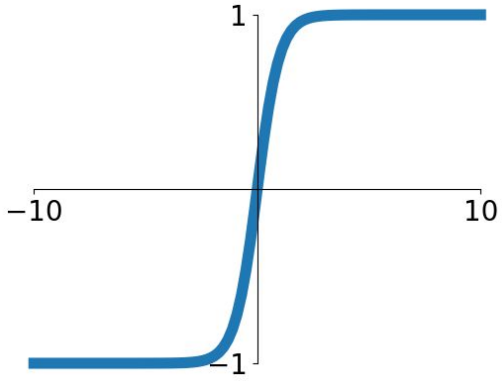
Sigmoid

- 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

Tanh

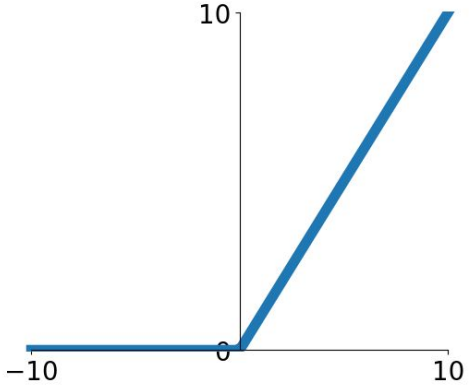


tanh(x)

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

[LeCun et al., 1991]

ReLU



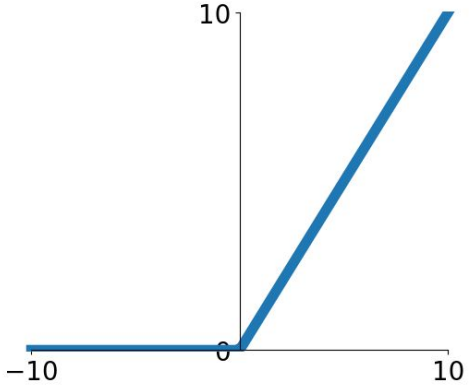
ReLU
(Rectified Linear Unit)

- 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]

ReLU



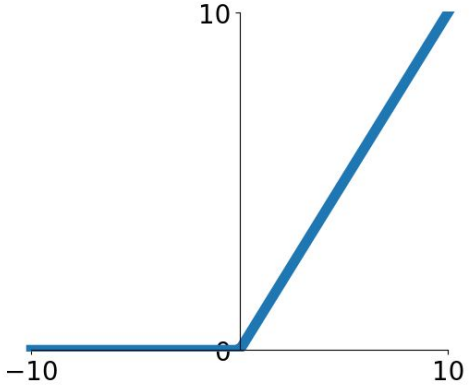
ReLU
(Rectified Linear Unit)

- 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

ReLU



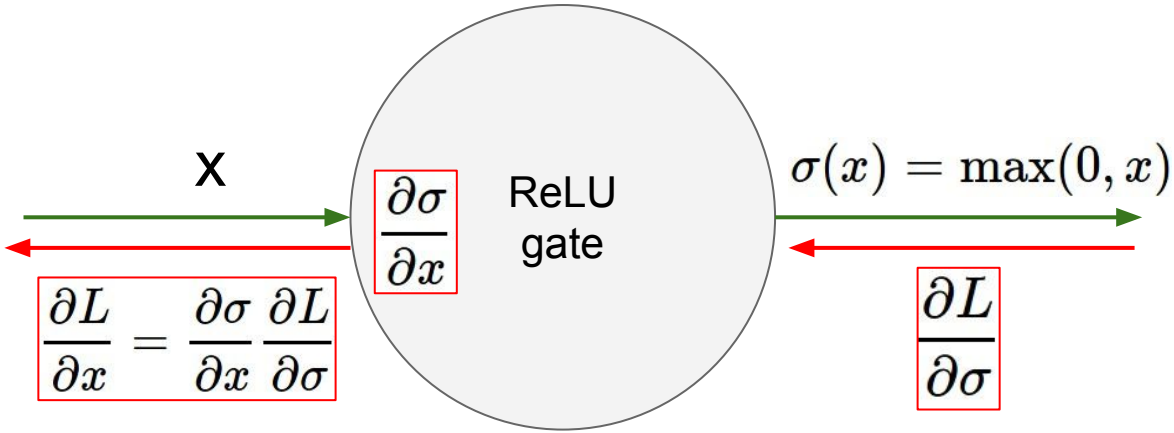
ReLU
(Rectified Linear Unit)

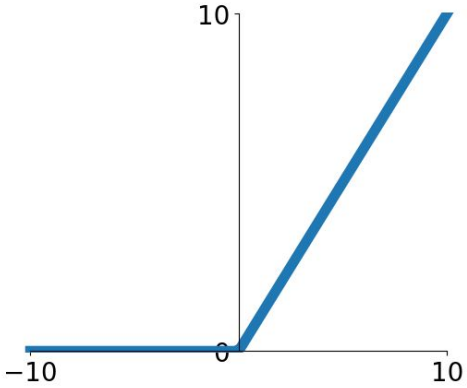
- Computes **f(x) = max(0,x)**

- 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:

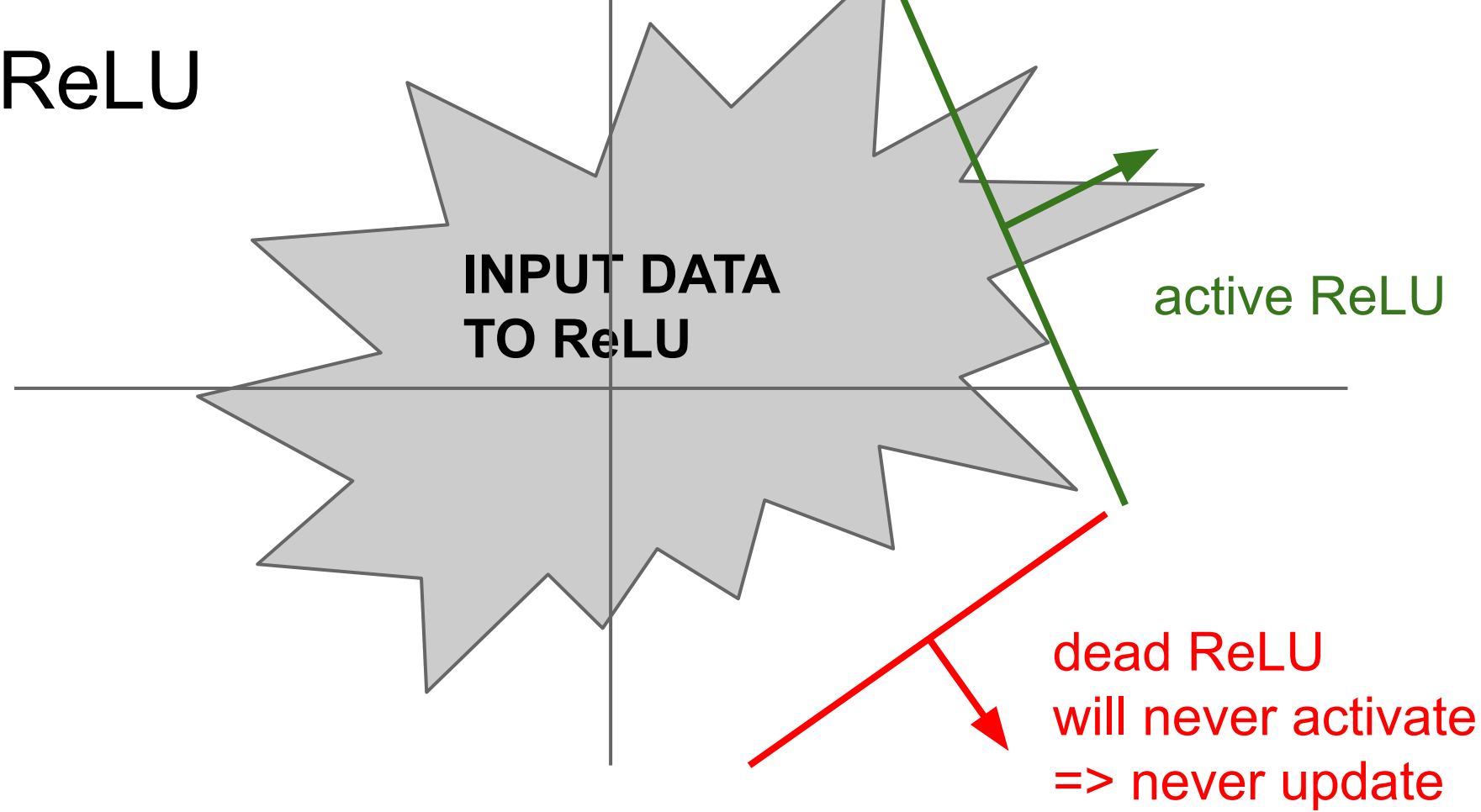
hint: what is the gradient when x < 0?



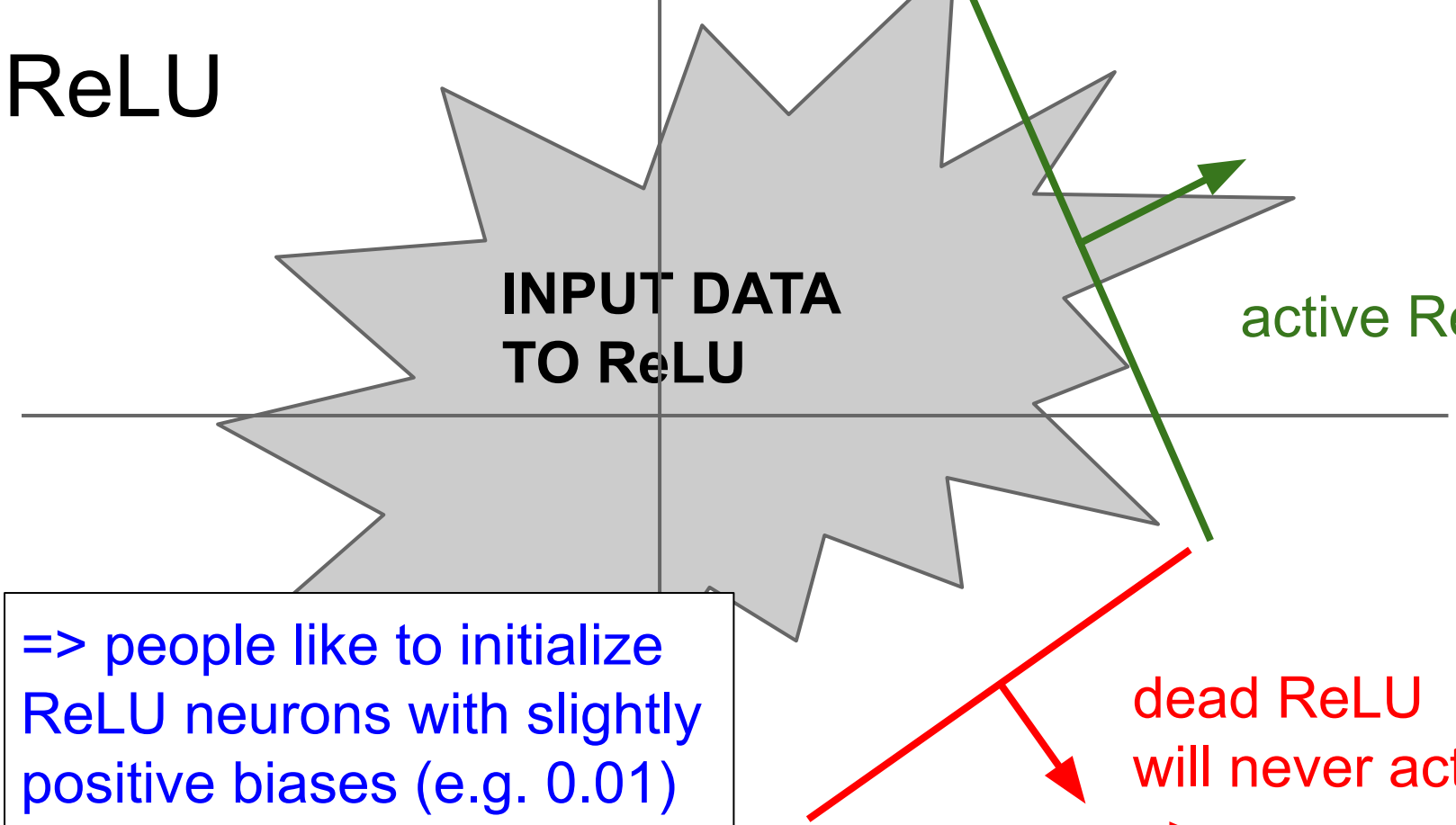


What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

ReLU


INPUT DATA
TO ReLU
active ReLU
dead ReLU
will never activate
=> never update

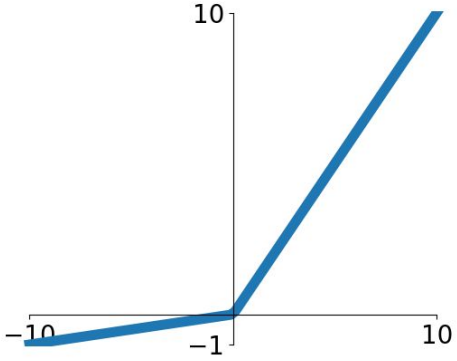
ReLU



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

Leaky ReLU

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



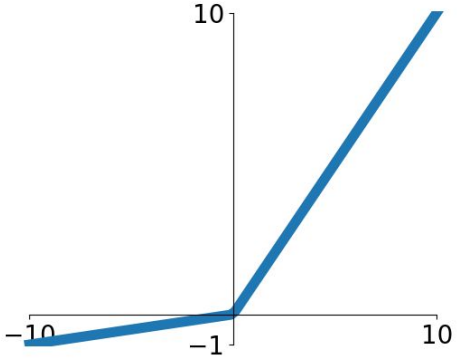
- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

Leaky ReLU

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

Leaky ReLU

[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)

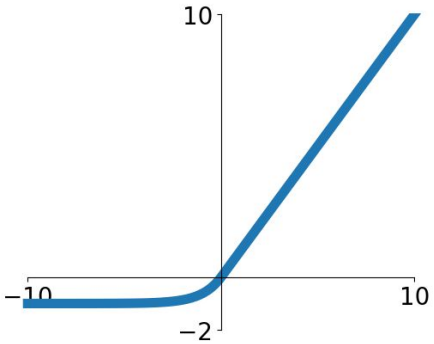
$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

ELU

[Clevert et al., 2015]

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs

- Computation requires exp()
- Negative saturation can kill gradients for large negative

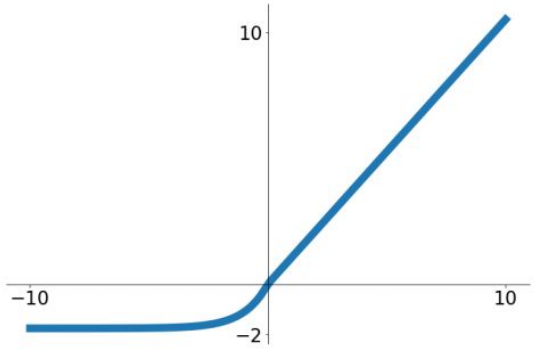
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

(Alpha default = 1)

SELU

[Klambauer et al. ICLR 2017]

Scaled Exponential Linear Units (SELU)



- Scaled versionof 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”

[Goodfellow et al., 2013]

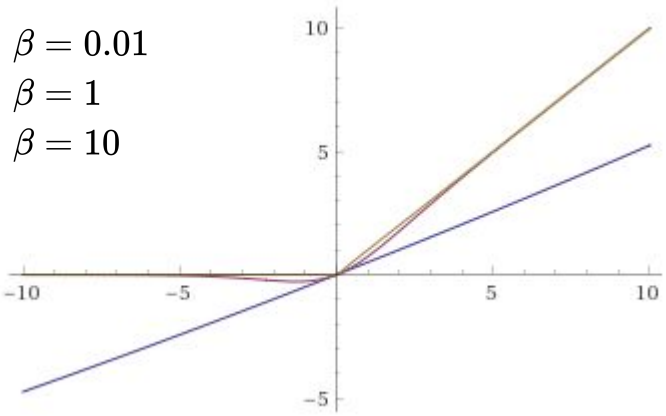
- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/weights :(

Swish

[Ramachandran et al. 2018]



Swish

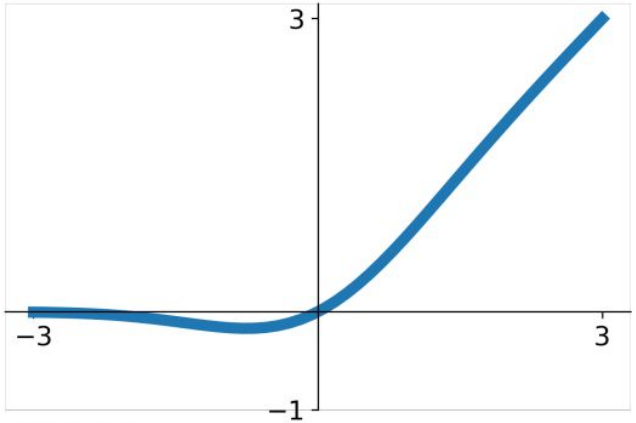
$$f(x) = x\sigma(\beta x)$$

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

GeLU

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

Gaussian Error Linear Units

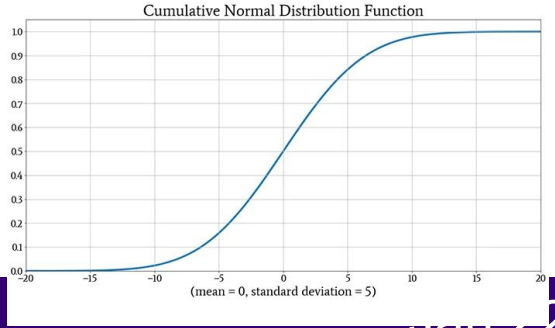


$$X \sim N(0,1)$$

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

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

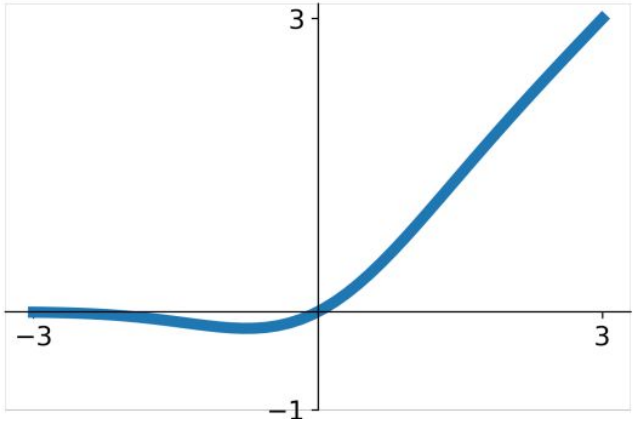
- Idea: combine the best parts of sigmoid and relu.
- GELU transitions more smoothly
- It weights each input according to the Gaussian (Normal) CDF



GeLU

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

Mathematically, $\mathrm{GELU}(x) = x \times \Phi(x)$



where $\Phi(x)$ is the cumulative distribution function (CDF) of a standard normal distribution $N(0,1)$

$$\Phi(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

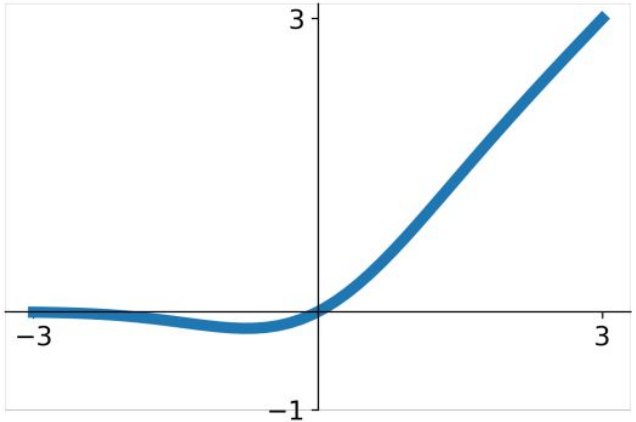
It is approximated as

$$\mathrm{GELU}(x) \approx 0.5\,x\left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715\,x^3\right)\right)\right]$$

Activation Functions

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

GeLU



$X \sim N(0,1)$

$$gelu(x) = xP(X \leq x) = \frac{x}{2}\left(1 + \text{erf}\left(x/\sqrt{2}\right)\right)$$
$$\approx x\sigma(1.702x)$$

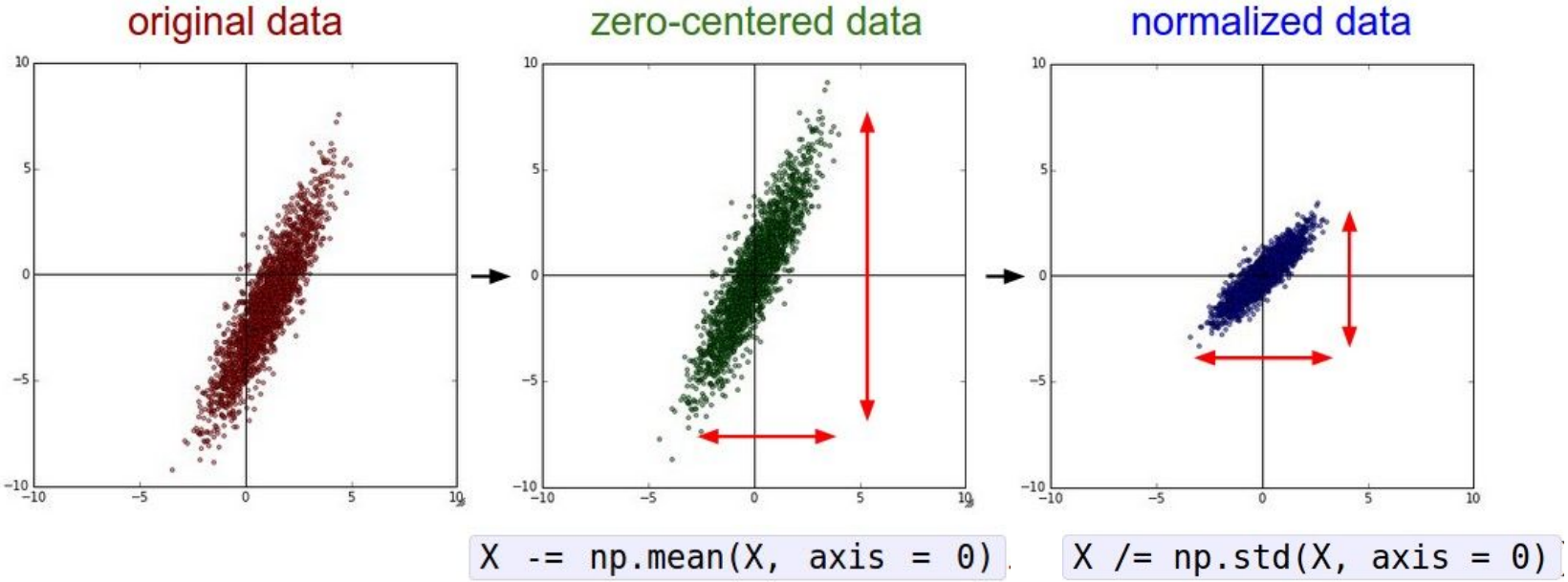
- More continuous version of ReLU. It scales the input x by a probability factor dependent on x
- Avoid dead neurons
- Empirically stable training
- Very common in Transformers (BERT, GPT, ViT)

TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Use GeLU when using transformers
- Try out Leaky ReLU / Maxout / ELU / SELU
  - To squeeze out some marginal gains
- Don’t use sigmoid or tanh

Data Preprocessing

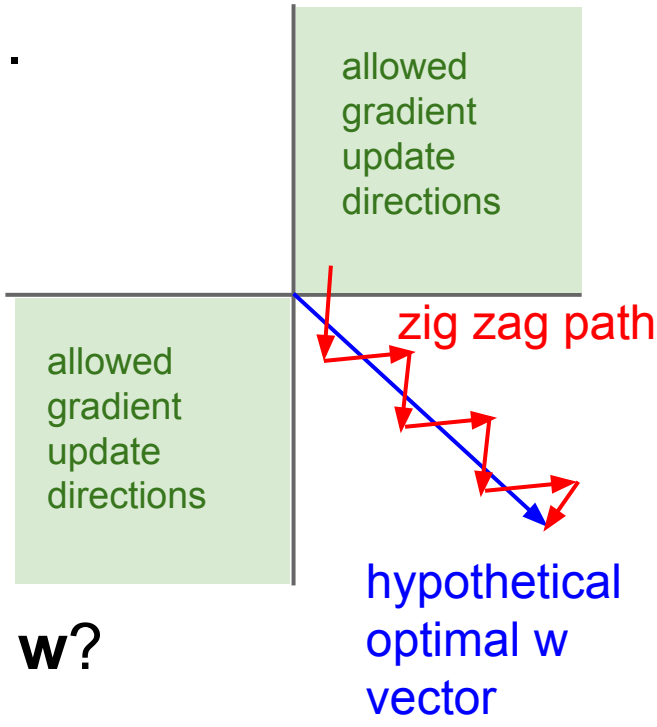
Data Preprocessing



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

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

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



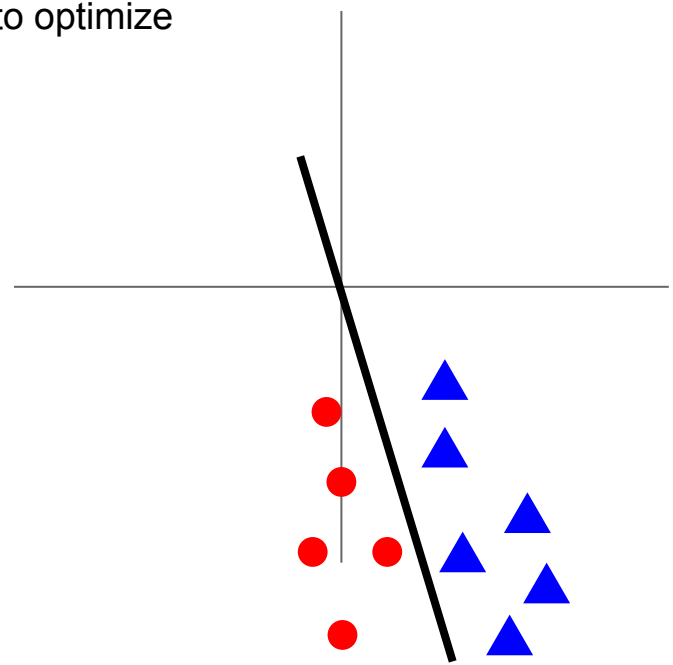
What can we say about the gradients on **w**?

Always all positive or all negative :(

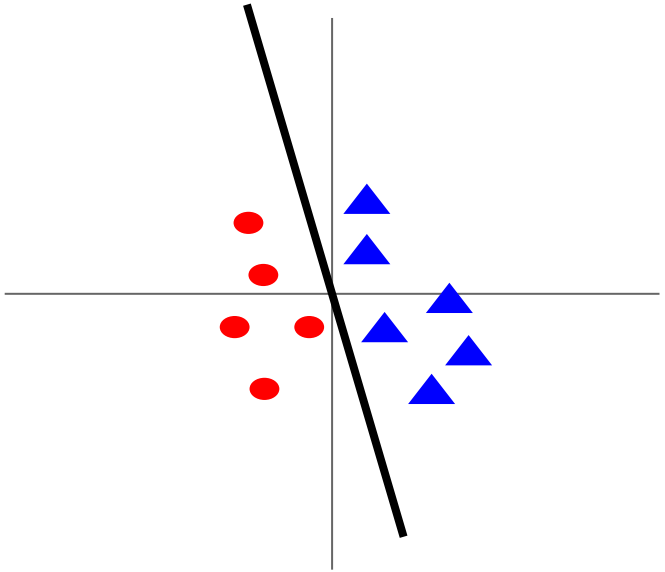
(this is also why you want zero-mean data!)

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize

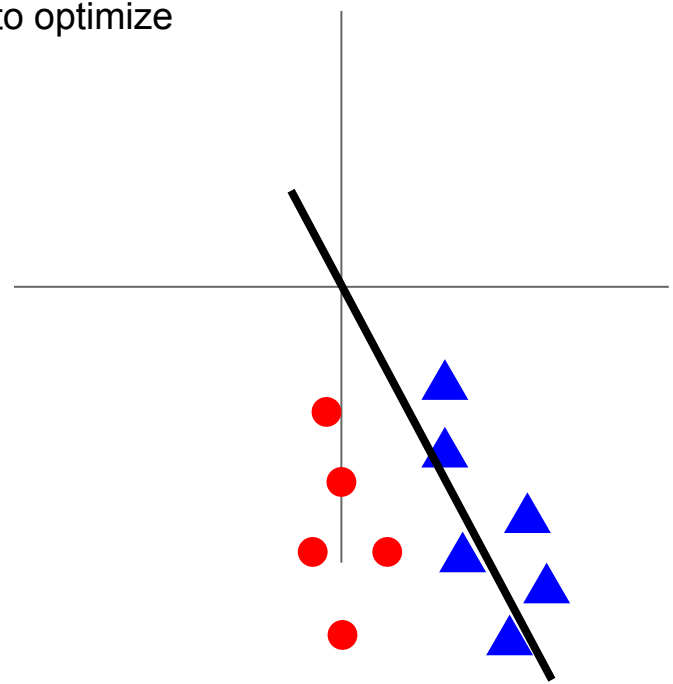


After normalization: less sensitive to small changes in weights; easier to optimize

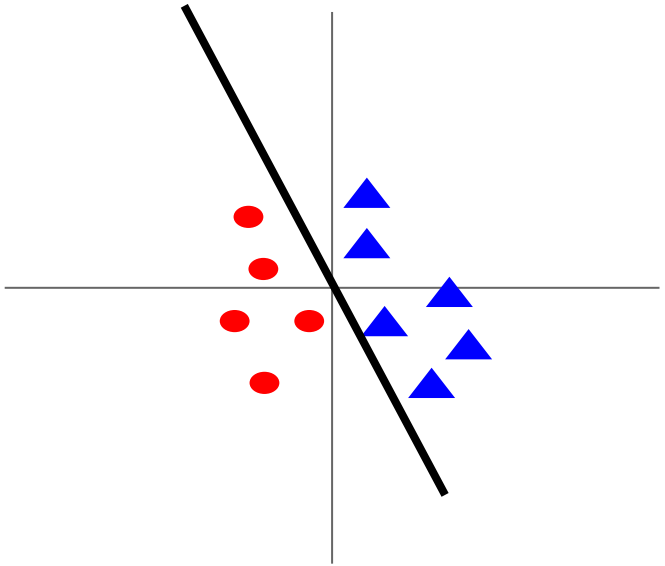


Data Preprocessing

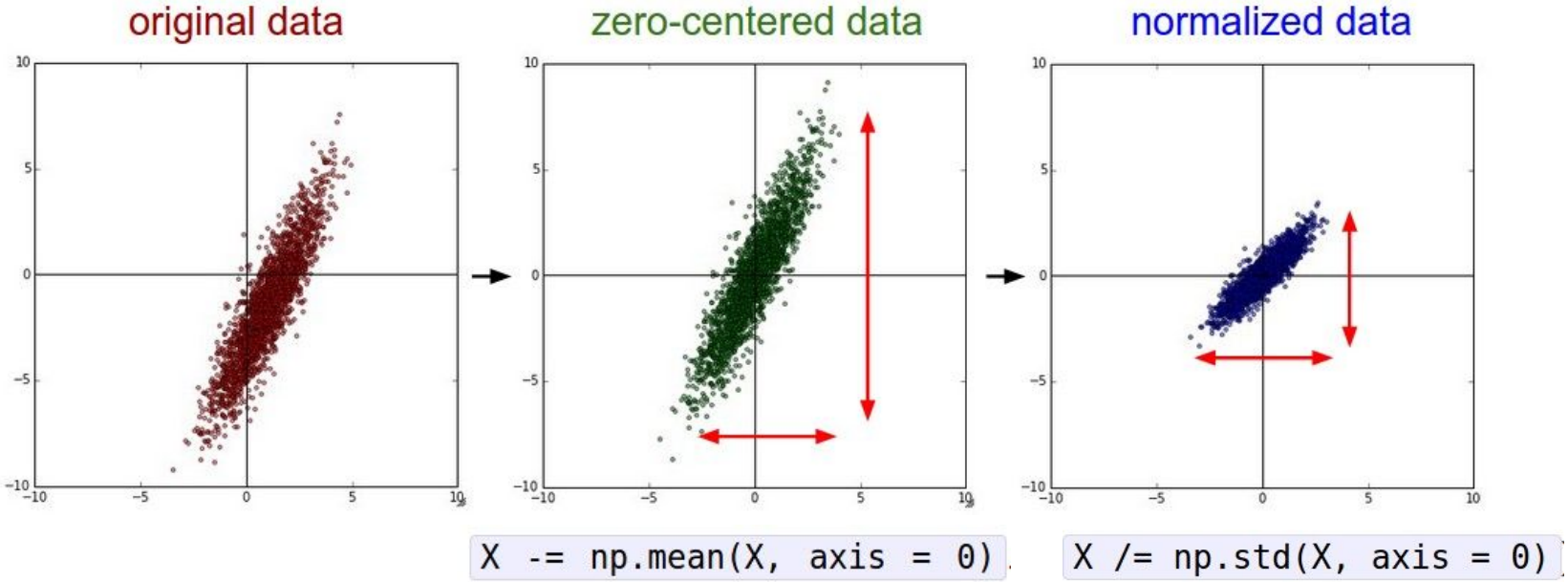
Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



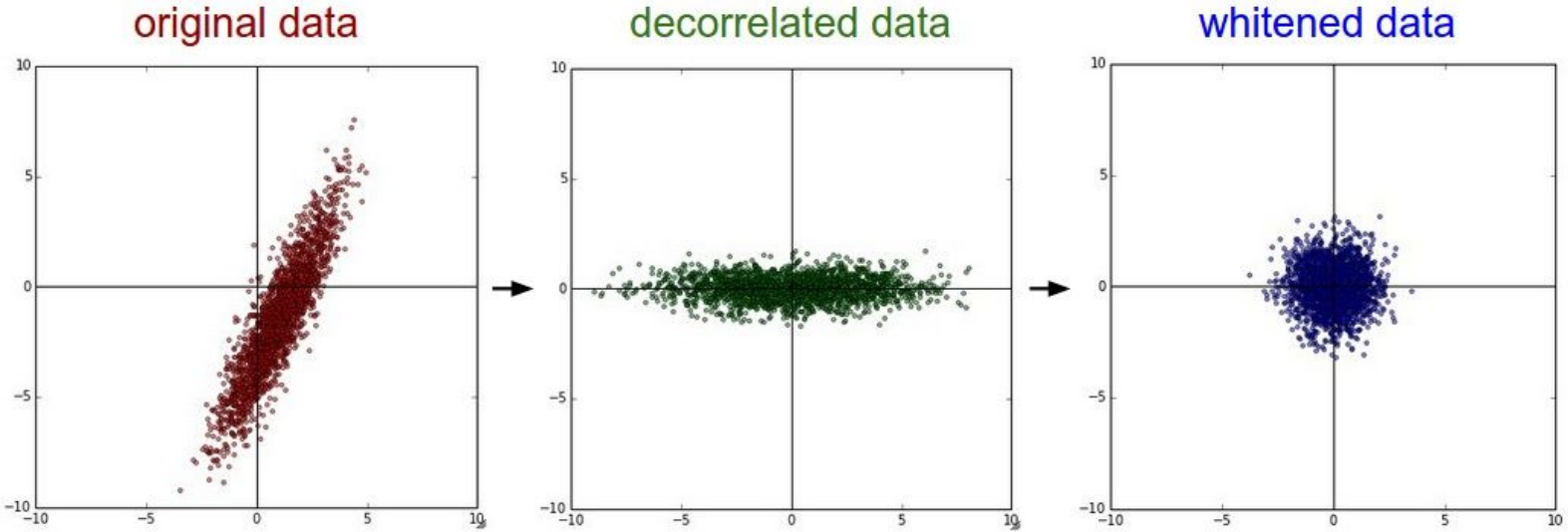
Data Preprocessing



(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



TLDR: In practice for Images: center only

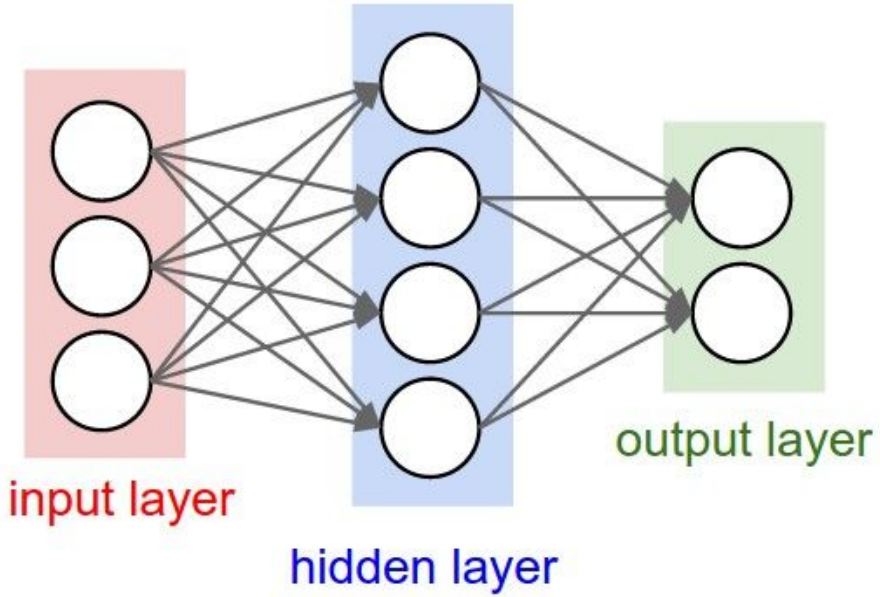
e.g. consider CIFAR-10 example with [32,32,3] images

- 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?

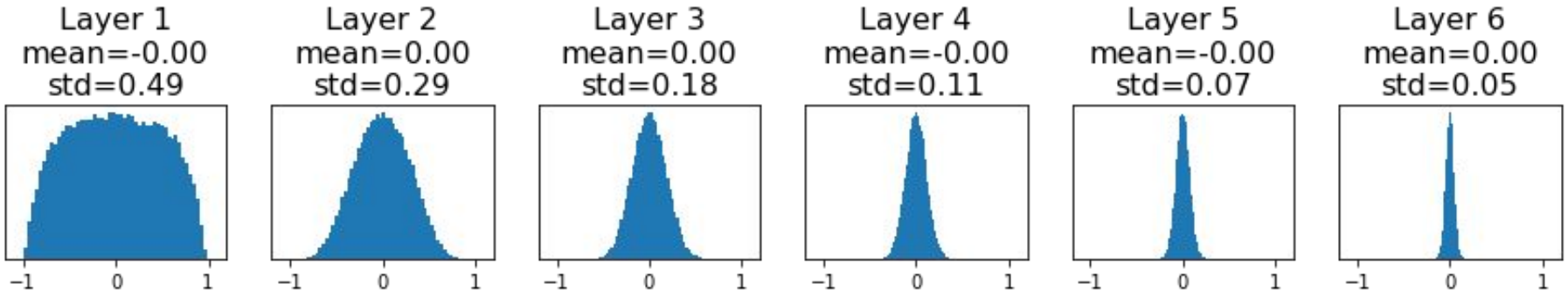
Weight Initialization: Activation statistics

```
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

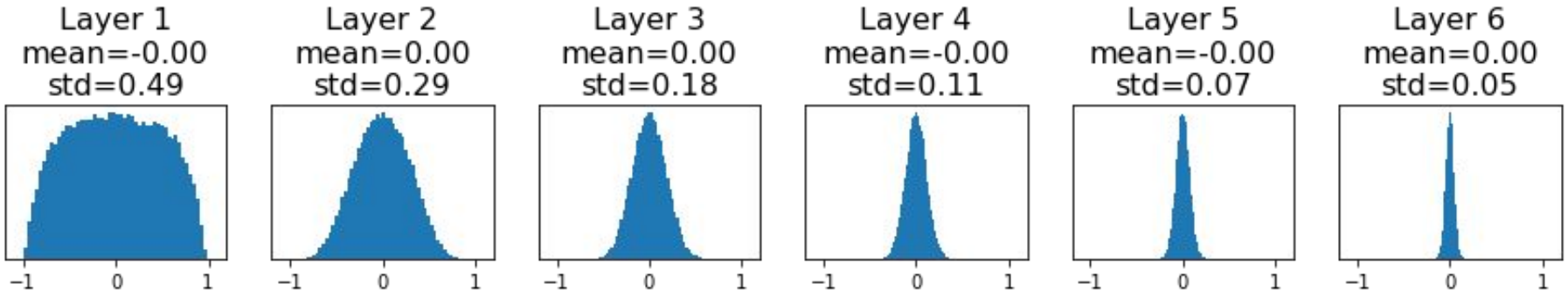
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 when the weights are initialized with larger values?

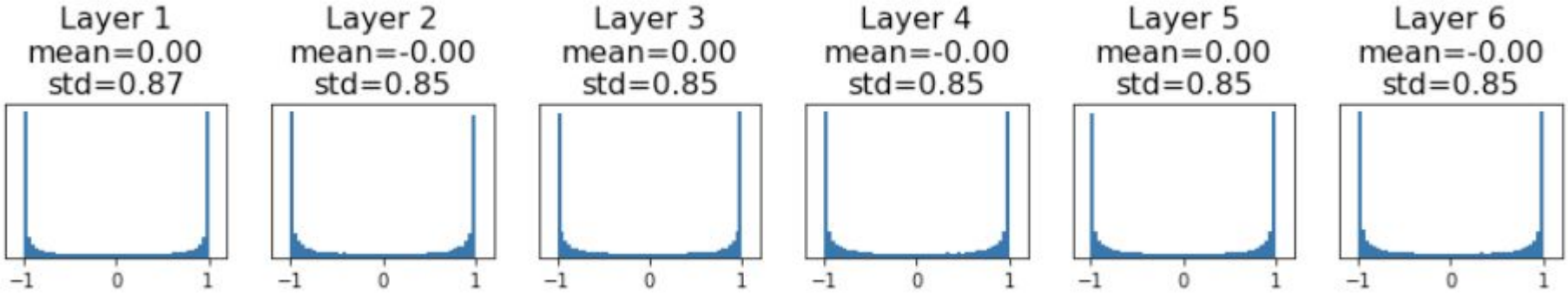
Weight Initialization: Activation statistics

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

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Weight Initialization: Activation statistics

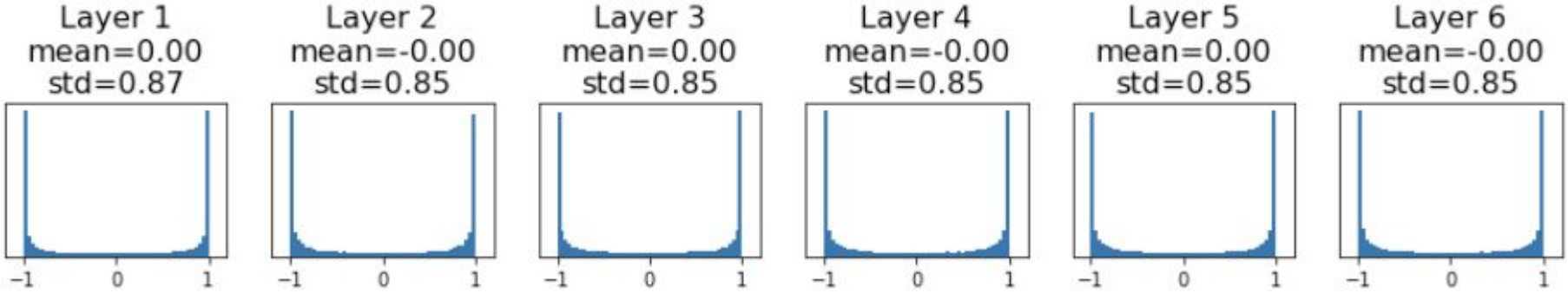
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Weight Initialization: “Xavier” Initialization

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

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

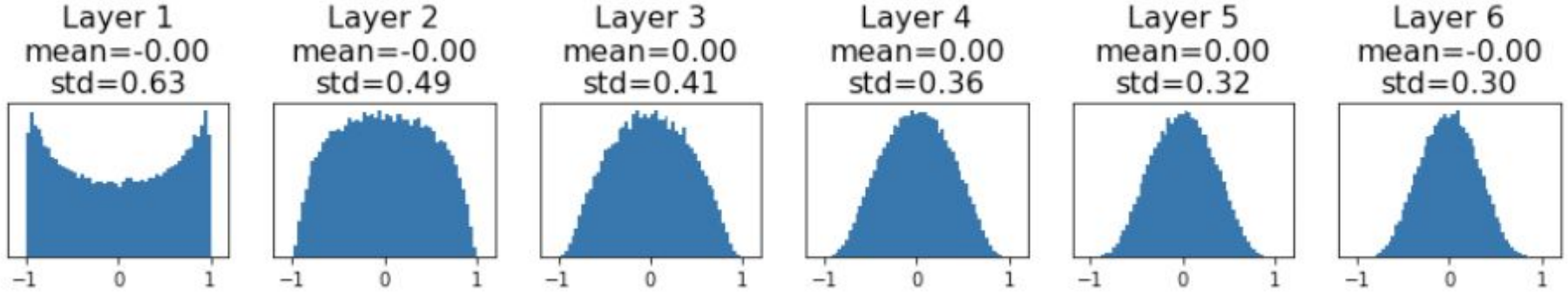
Glorot and Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AISTAT 2010

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!



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

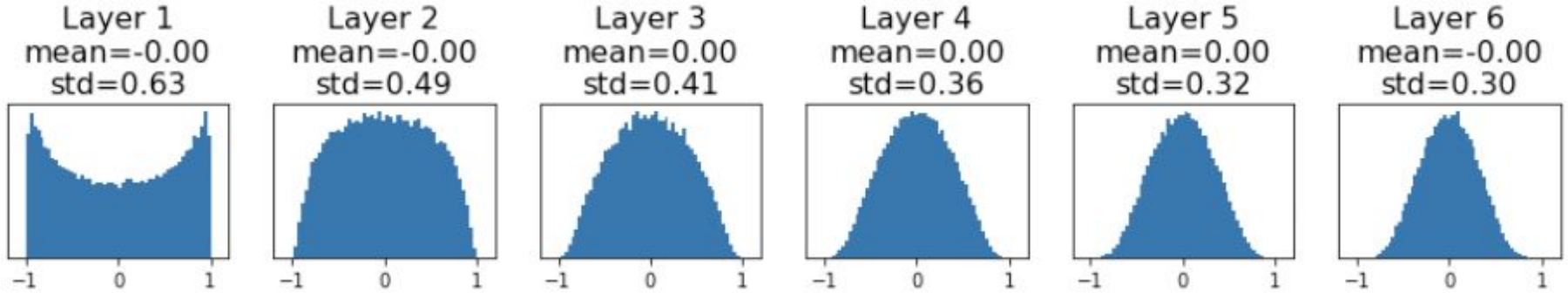
Weight Initialization: “Xavier” Initialization

```
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 filter_size^2 * input_channels

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

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: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

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: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

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: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

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

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$
[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: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

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

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

$Var(y) = Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din})$

$= Din\ Var(x_iw_i)$

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

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

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

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

[Assume all x_i, w_i are iid]

So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

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

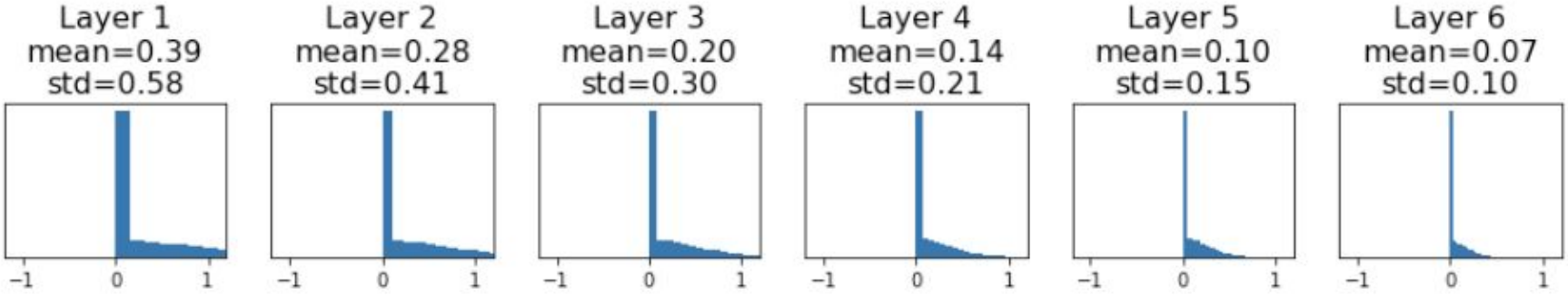
Weight Initialization: What about ReLU?

```
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 =(

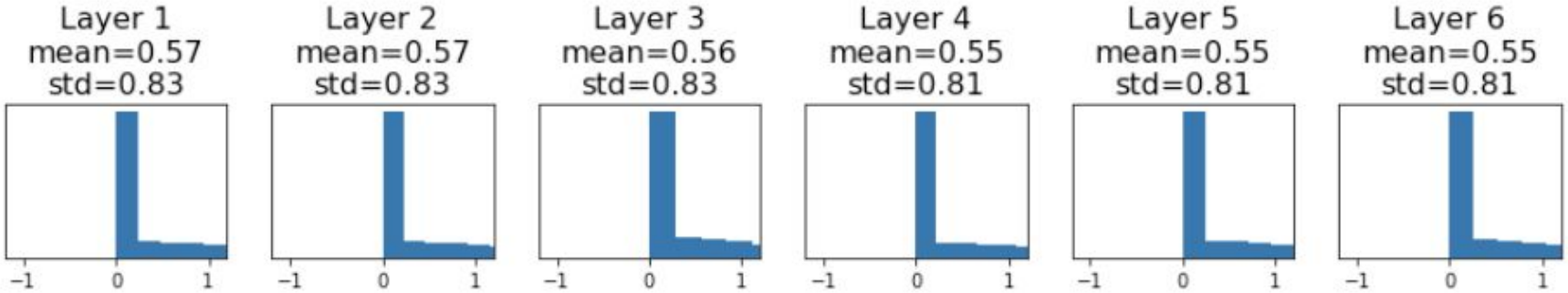


Weight Initialization: Kaiming / MSRA Initialization

```
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:

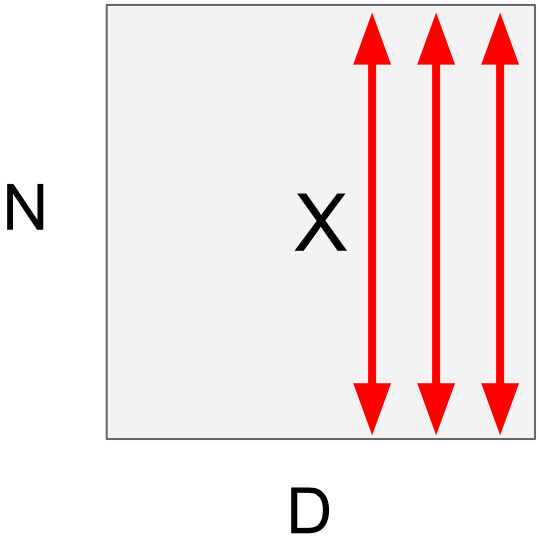
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

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

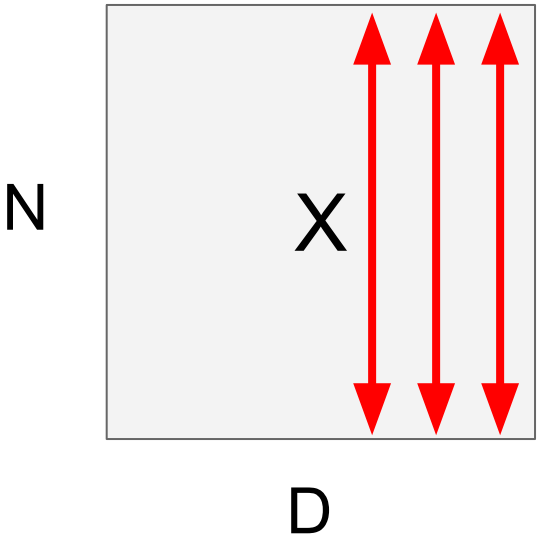
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

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

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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

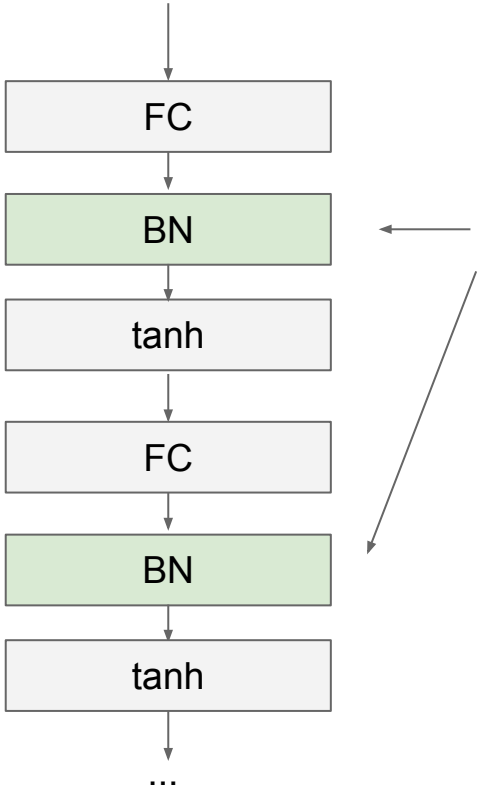
Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

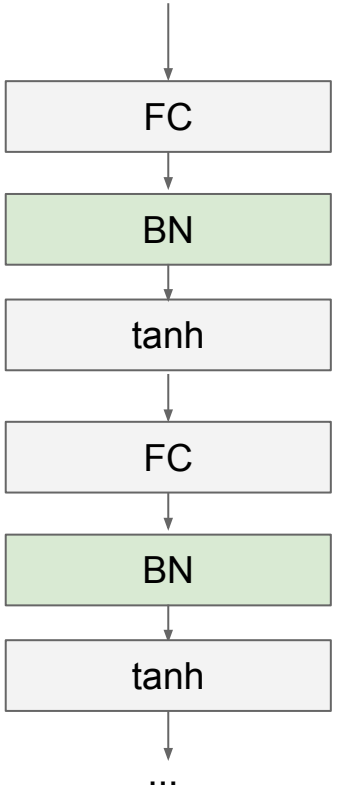


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

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 debugging!

Batch Normalization for convolutions

Batch Normalization for
fully-connected networks

$x: N \times D$

Normalize

$\mu, \sigma: 1 \times D$

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

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

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

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

Normalize

$\mu, \sigma: 1 \times C \times 1 \times 1$

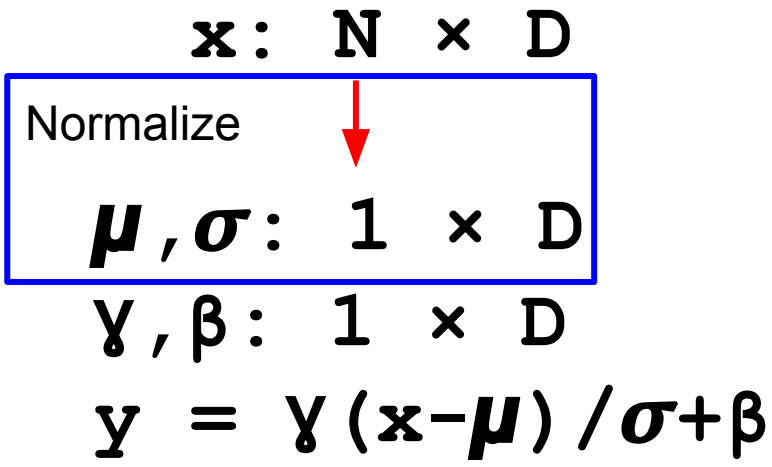
$\gamma, \beta: 1 \times C \times 1 \times 1$

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

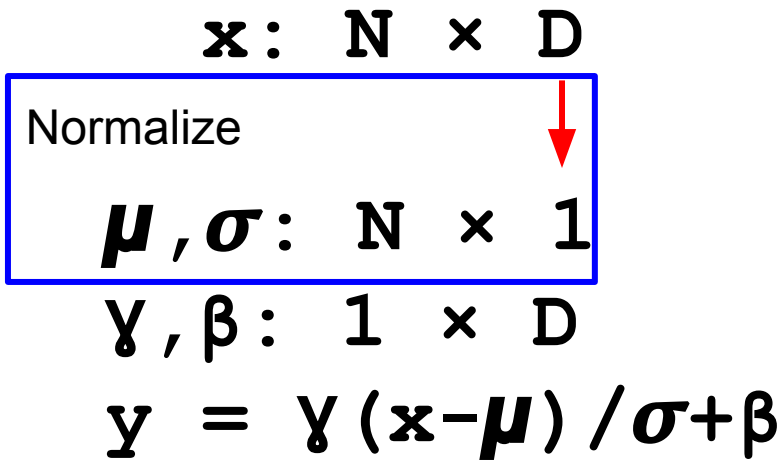
Layer Normalization

Layer Normalization for MLPs

Batch Normalization for fully-connected networks



Layer Normalization for fully-connected networks
Same behavior at train and test!
Often used in transformers

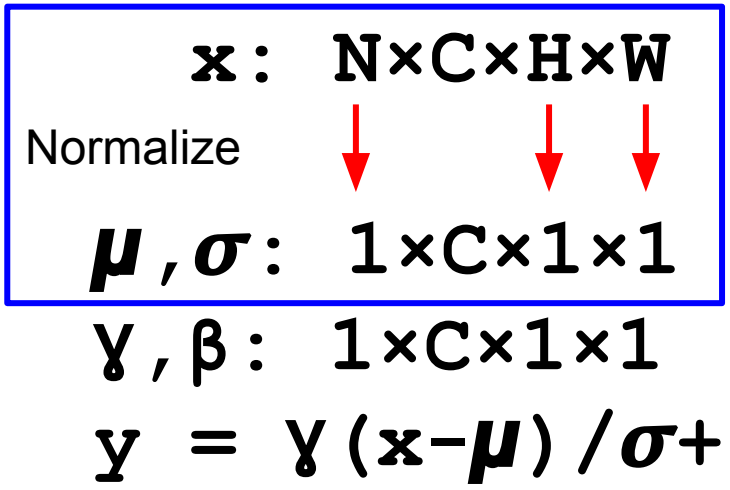


Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

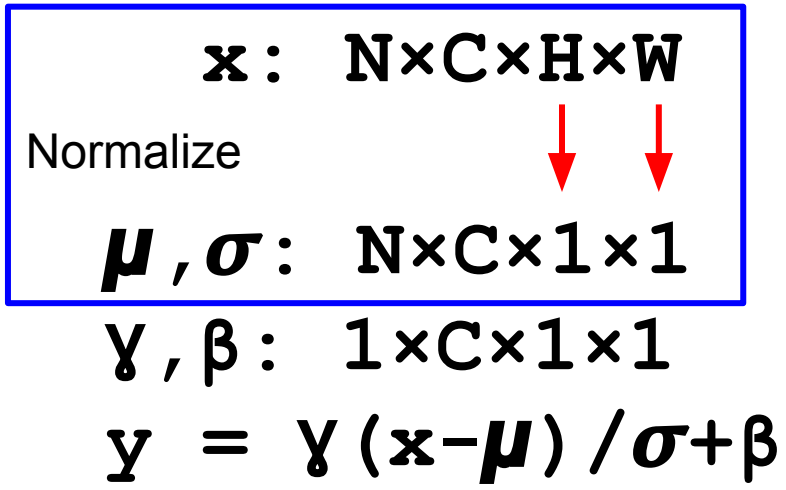
Instance Normalization

Instance Normalization for Convolutions

Batch Normalization for convolutional networks

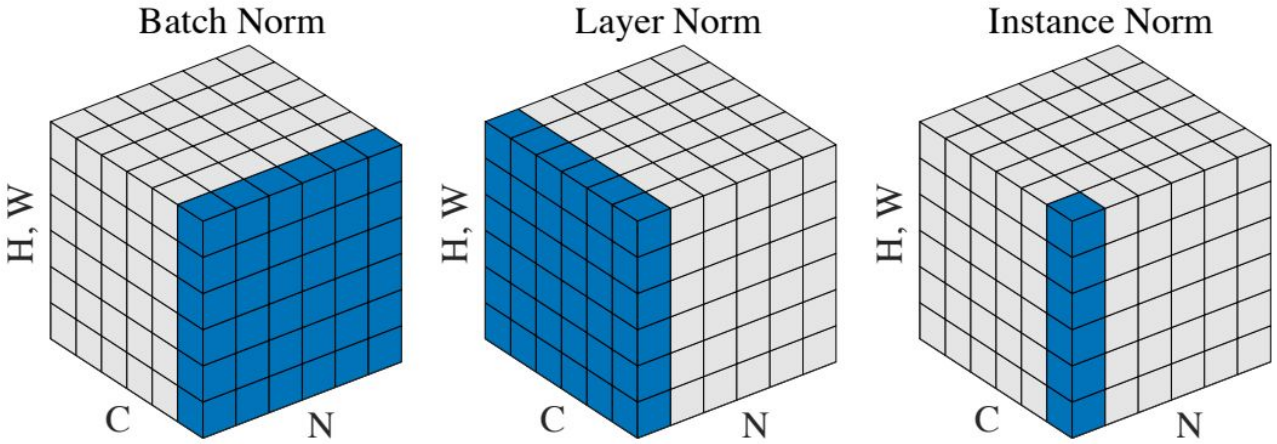


Instance Normalization for convolutional networks
Same behavior at train / test!



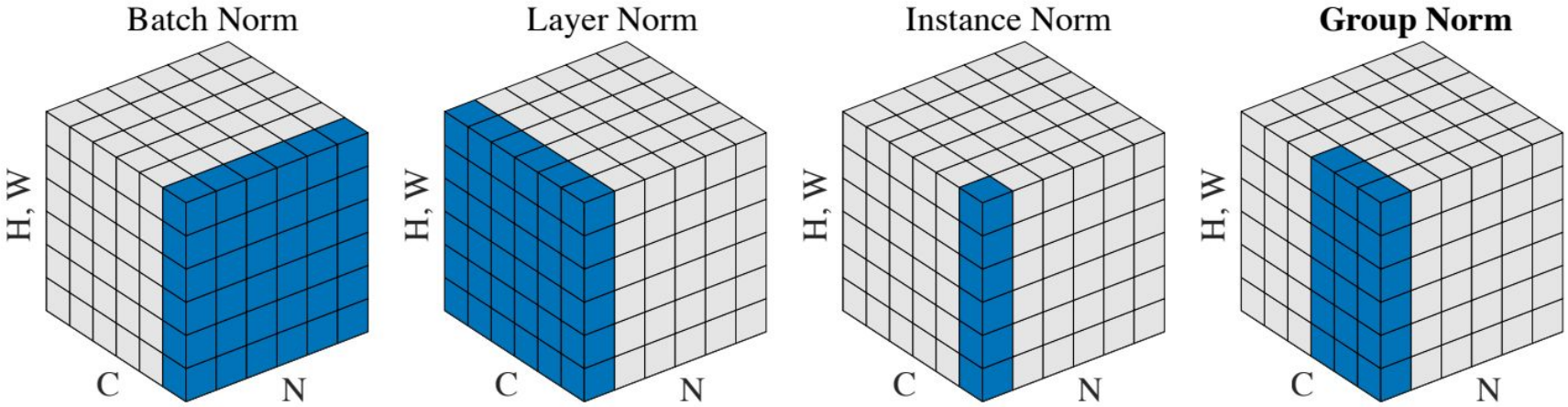
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



Wu and He, “Group Normalization”, ECCV 2018

Group Normalization



Wu and He, “Group Normalization”, ECCV 2018

Summary

TLDRs

We looked in detail at:

- Activation Functions (use ReLU or GeLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use this!)
- Layer Normalization (used in transformers!)

Next time: Optimizers

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