

Lecture 5:

Backprop + Convolutional

Neural Networks

Administrative: EdStem

Please make sure to check and read all pinned EdStem posts.

Administrative: Assignment 1

Due 1/22 11:59pm

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

Administrative: Fridays

This Friday

Convolutions & Vectorization

Administrative: Course Project

Project proposal due 1/29 11:59pm

“Is X a valid project for 493G1?”

- Anything related to deep learning or computer vision
- Maximum of 3 students per team
- Make a EdStem private post or come to TA Office Hours

More info on the website

Form to help make groups posted, please fill out by Thursday if interested.

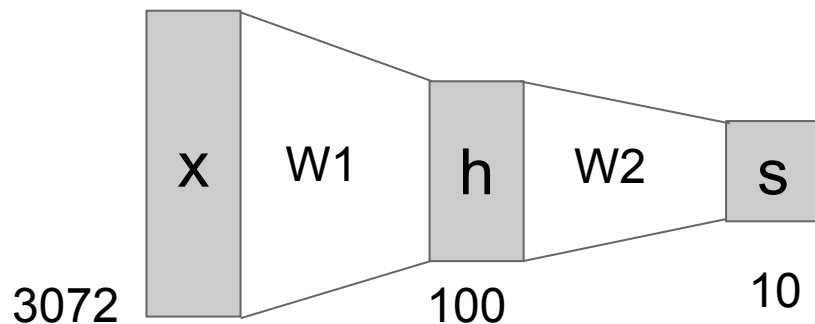
Last time: Neural Networks

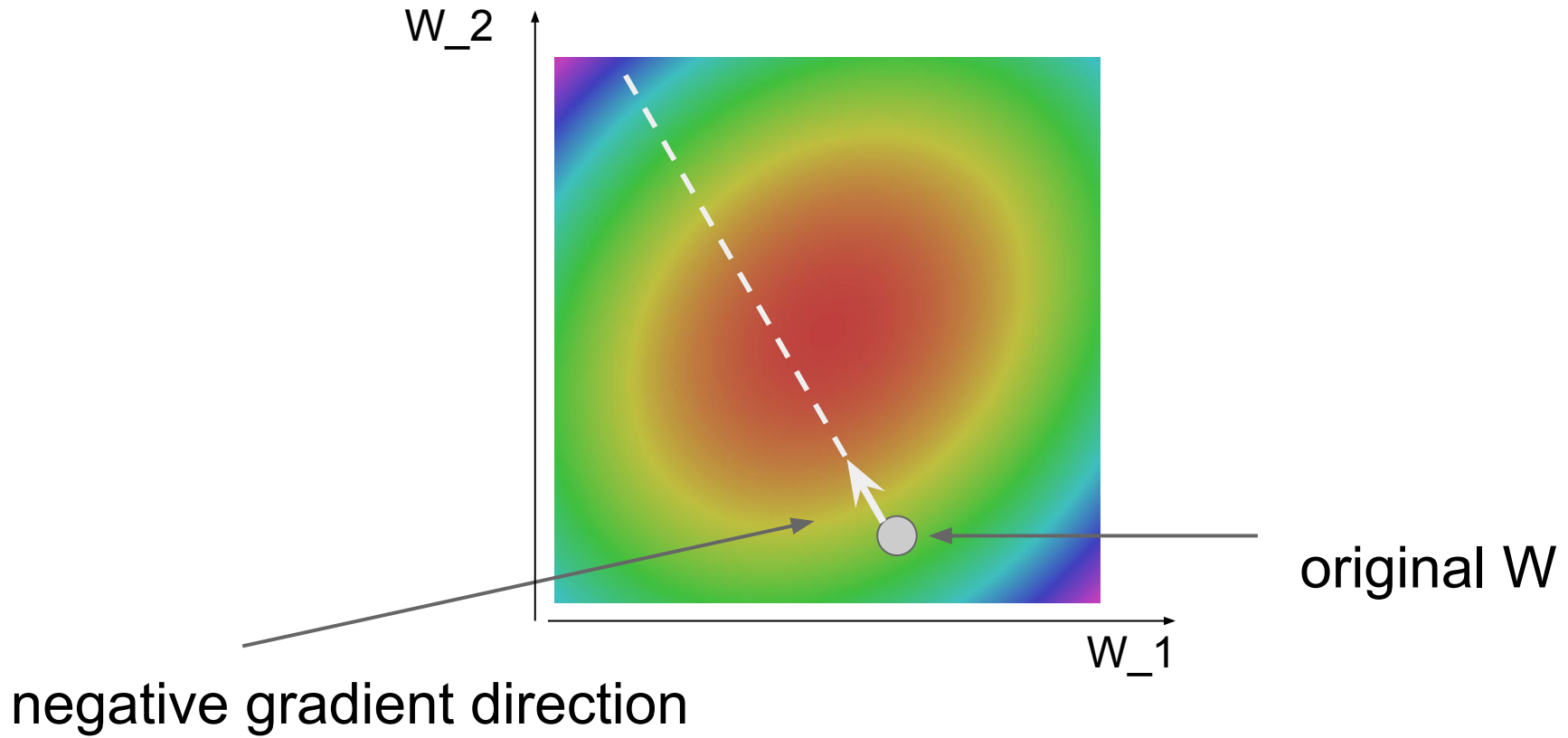
Linear score function:

$$f = Wx$$

2-layer Neural Network

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





Stochastic Gradient Descent (SGD)

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W) + \lambda R(W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive
when N is large!

Approximate sum
using a **minibatch** of
examples
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent
```

```
while True:
```

```
    data_batch = sample_training_data(data, 256) # sample 256 examples
```

```
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
```

```
    weights += - step_size * weights_grad # perform parameter update
```


(Bad) Idea: Derive

on paper $\nabla_W L$

$$s = f(x; W) = Wx$$

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

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

Problem: What if we want to change loss? E.g. use softmax instead of SVM? Need to re-derive from scratch =(

Problem: Not feasible for very complex models!

Convolutional network (AlexNet)

input image

weights

loss

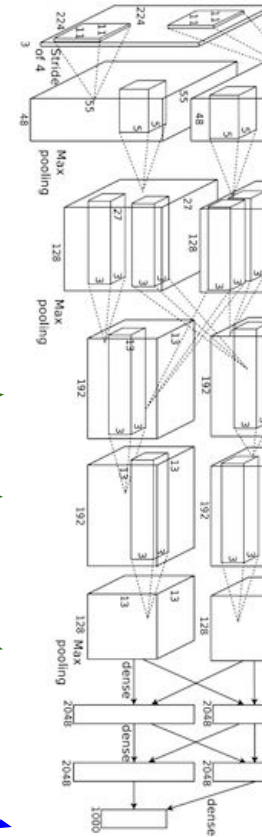


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Really complex neural networks!!

input image

loss

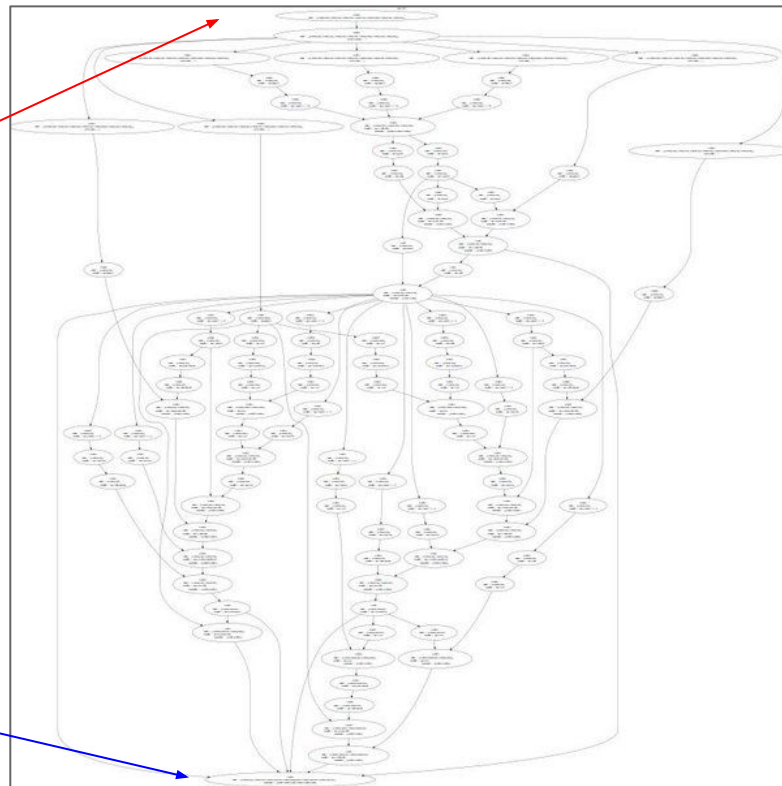
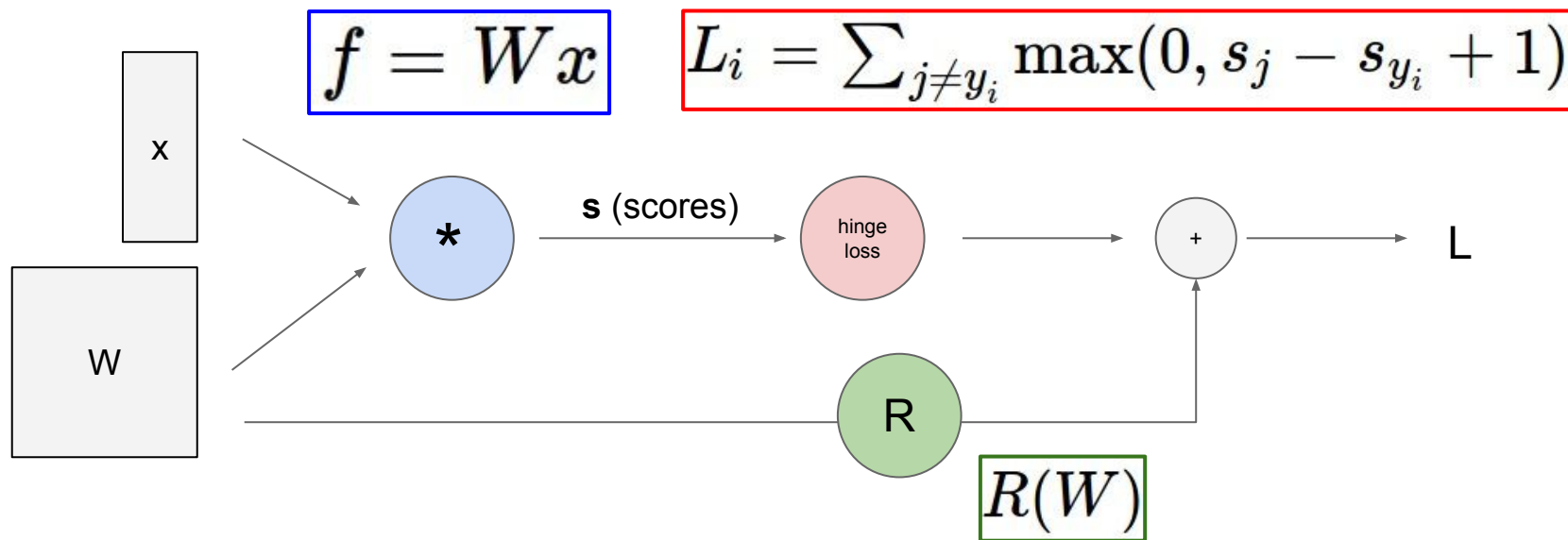


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

Better Idea: Computational graphs + Backpropagation



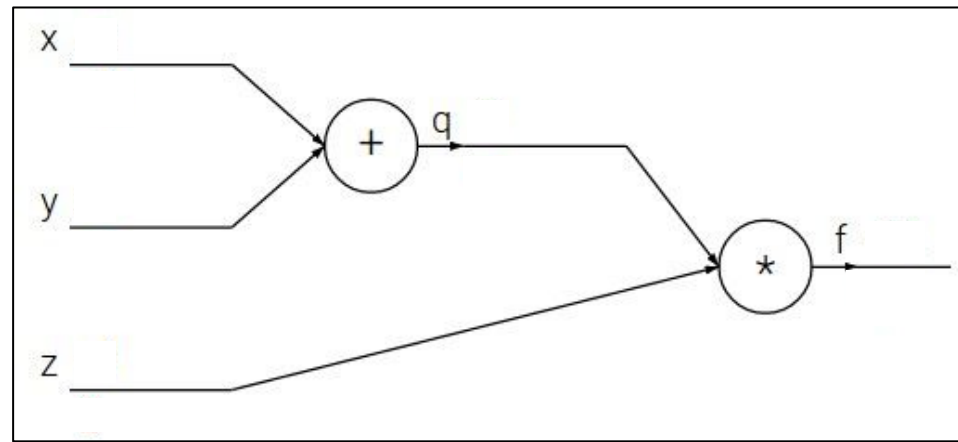
Solution: Backpropagation

Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

Backpropagation: a simple example

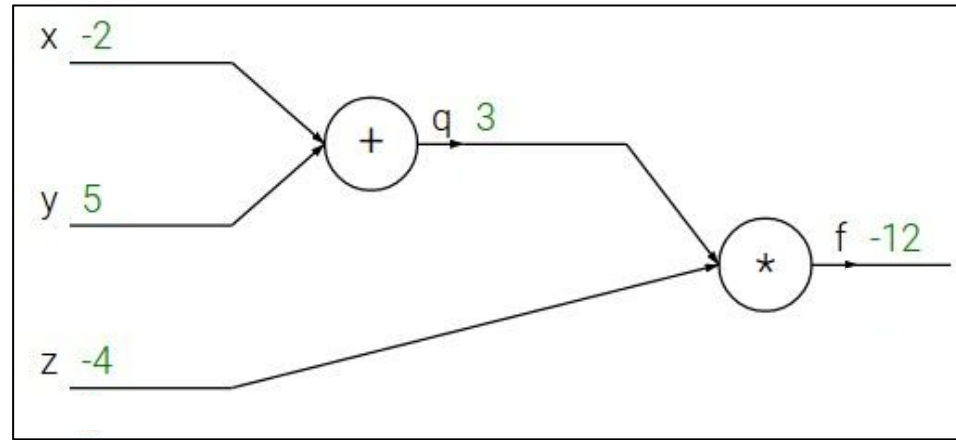
$$f(x, y, z) = (x + y)z$$



Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

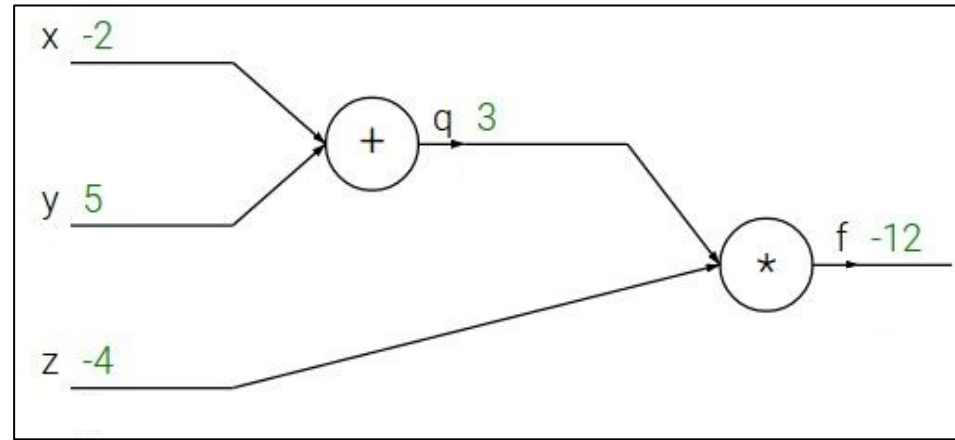


Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$



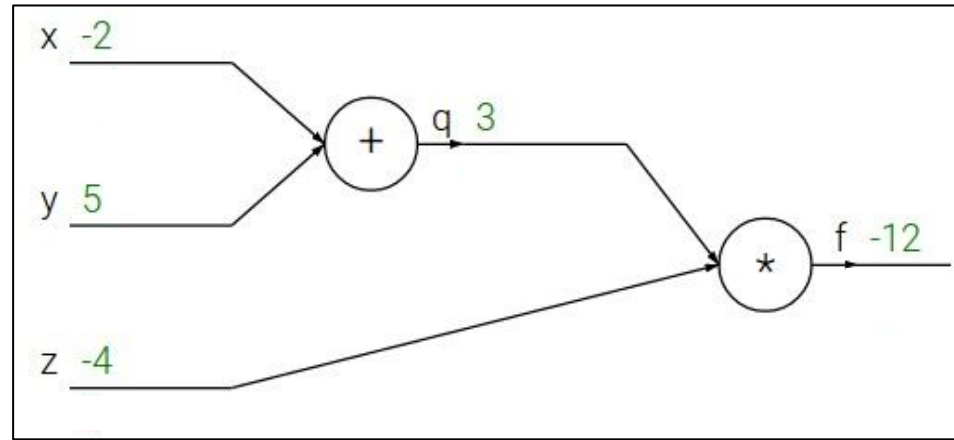
Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



Backpropagation: a simple example

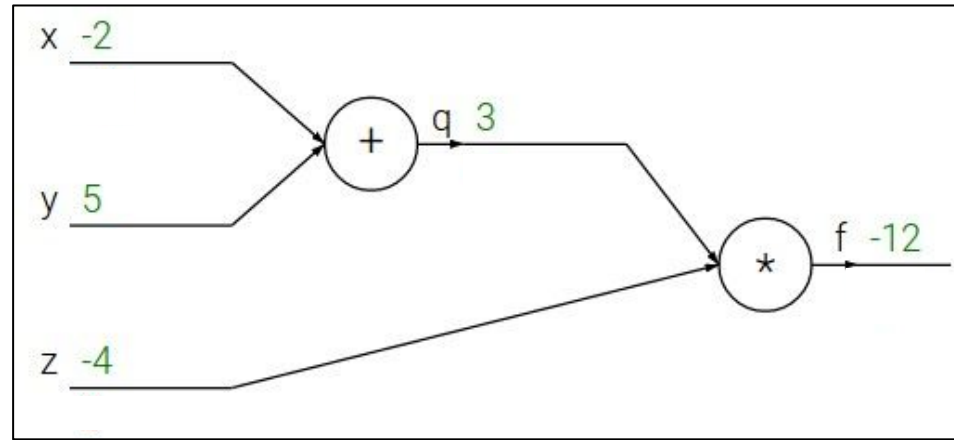
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

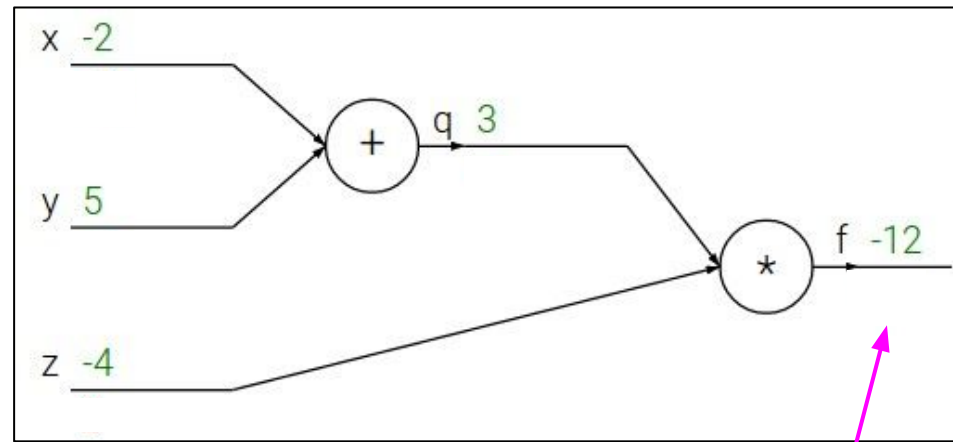
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial f}$$

Backpropagation: a simple example

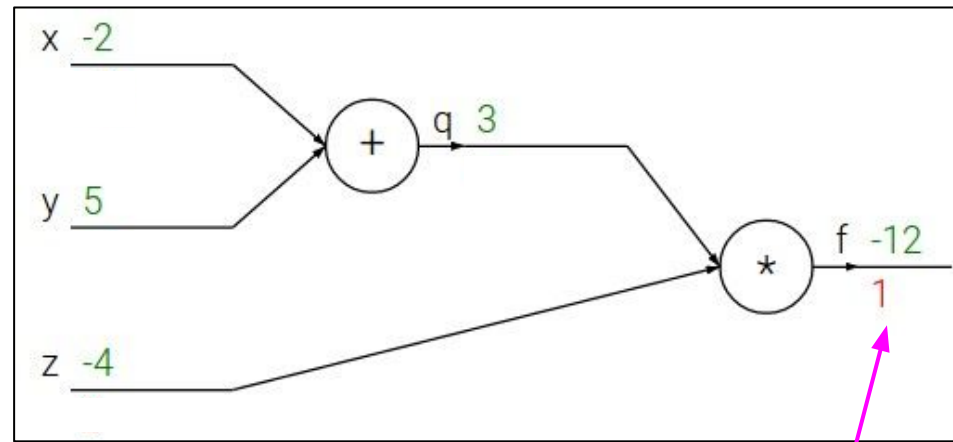
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial f}$$

A pink arrow points from this box to the output f of the multiplication node in the computational graph above.

Backpropagation: a simple example

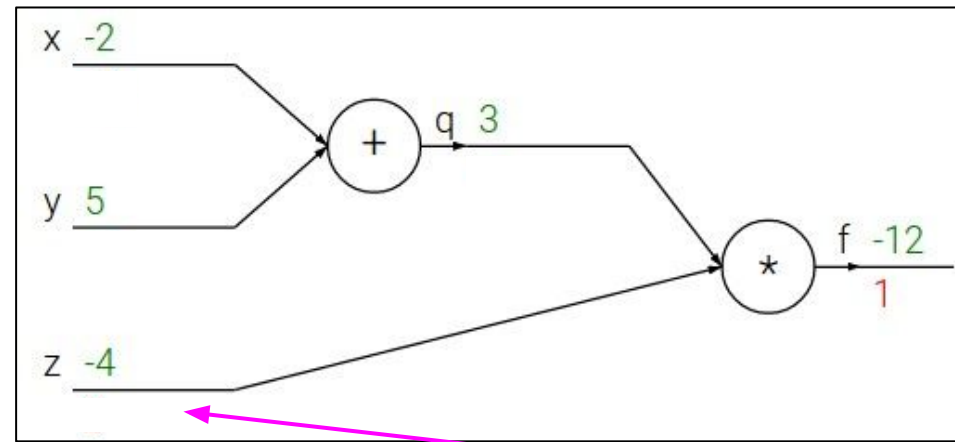
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

A magenta arrow points from this box to the input z of the multiplication node in the computational graph above.

Backpropagation: a simple example

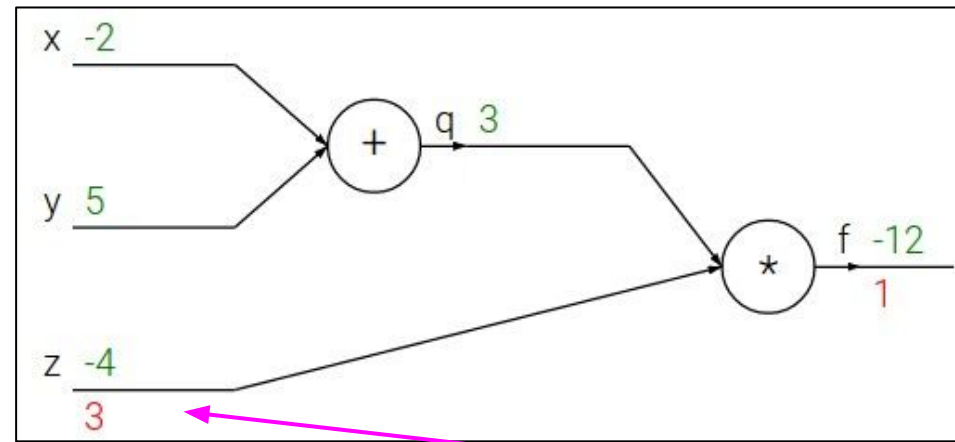
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

Backpropagation: a simple example

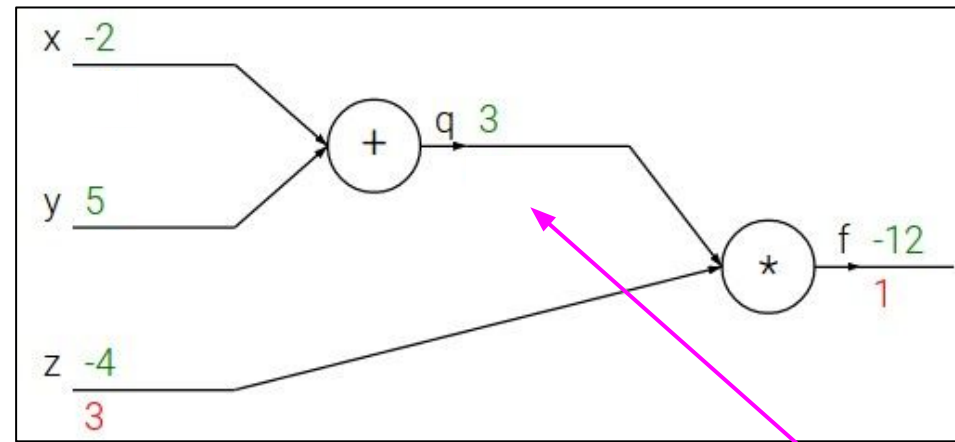
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial q}$$

Backpropagation: a simple example

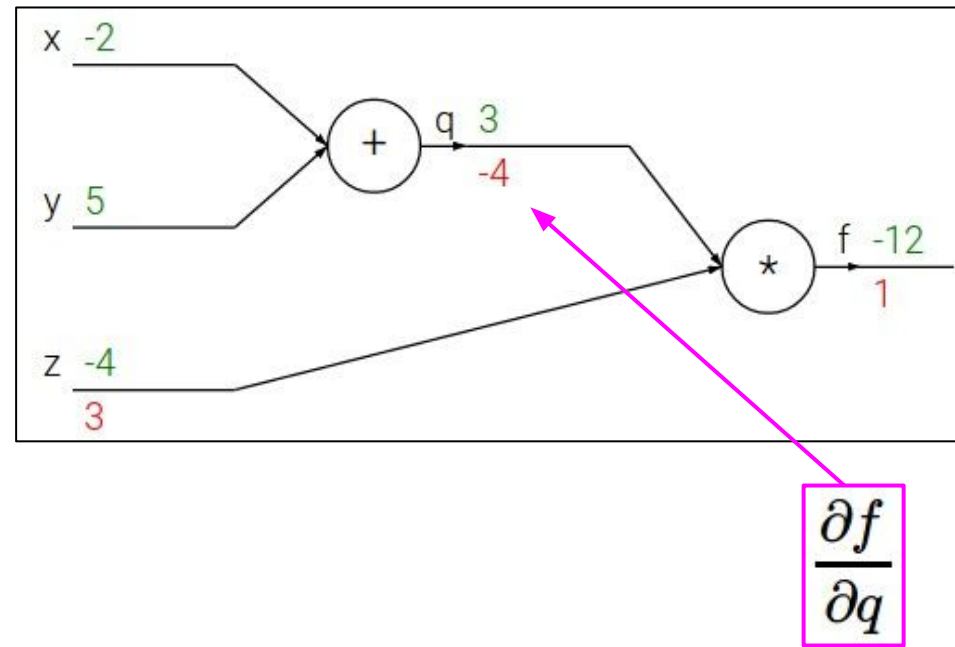
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation: a simple example

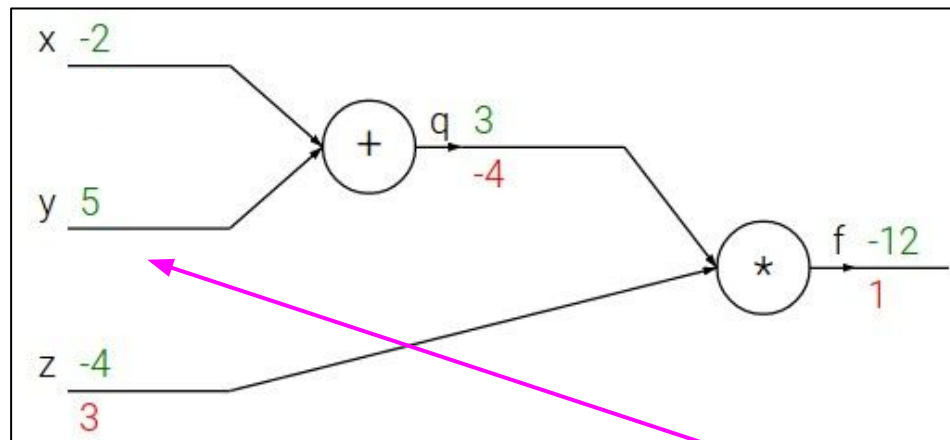
$$f(x, y, z) = (x + y)z$$

$$\text{e.g. } x = -2, y = 5, z = -4$$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

$$\text{Want: } \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$



$$\frac{\partial f}{\partial y}$$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Upstream
gradient

Local
gradient

Backpropagation: a simple example

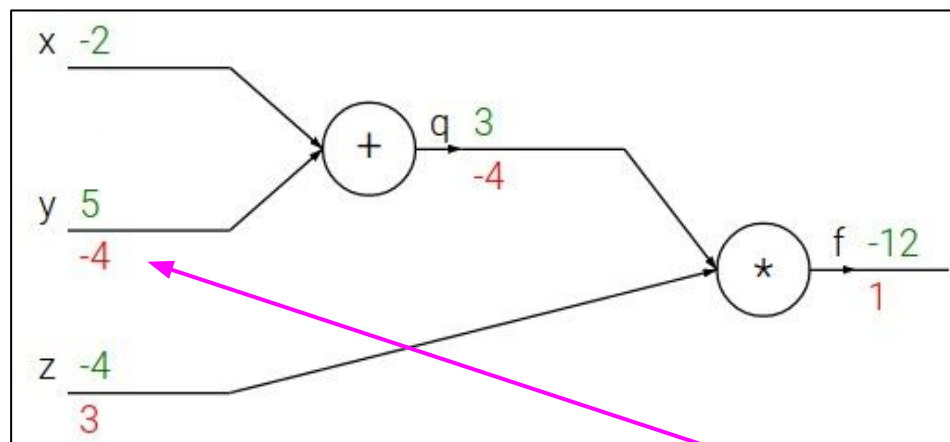
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial y}$$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Upstream
gradient

Local
gradient

Backpropagation: a simple example

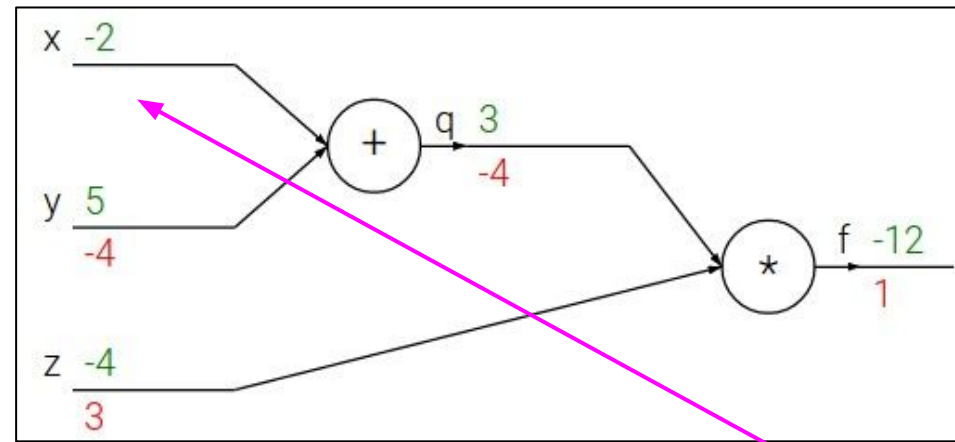
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial x}$$

Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Upstream
gradient

Local
gradient

Backpropagation: a simple example

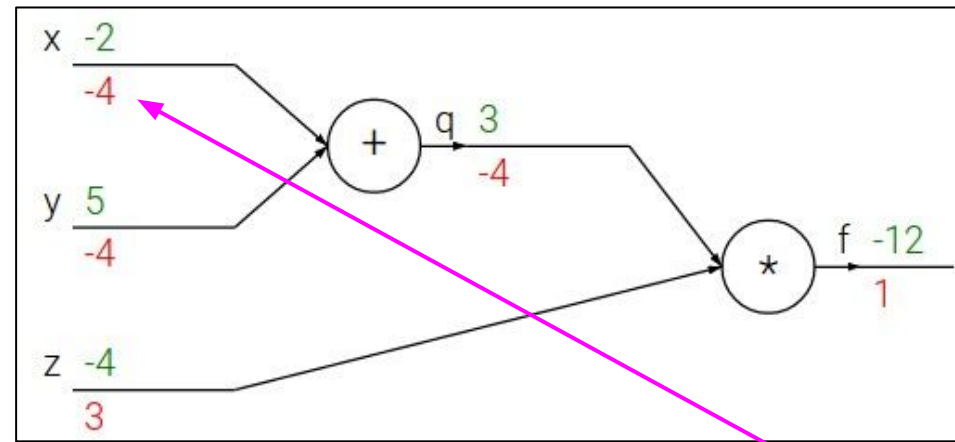
$$f(x, y, z) = (x + y)z$$

e.g. $x = -2$, $y = 5$, $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



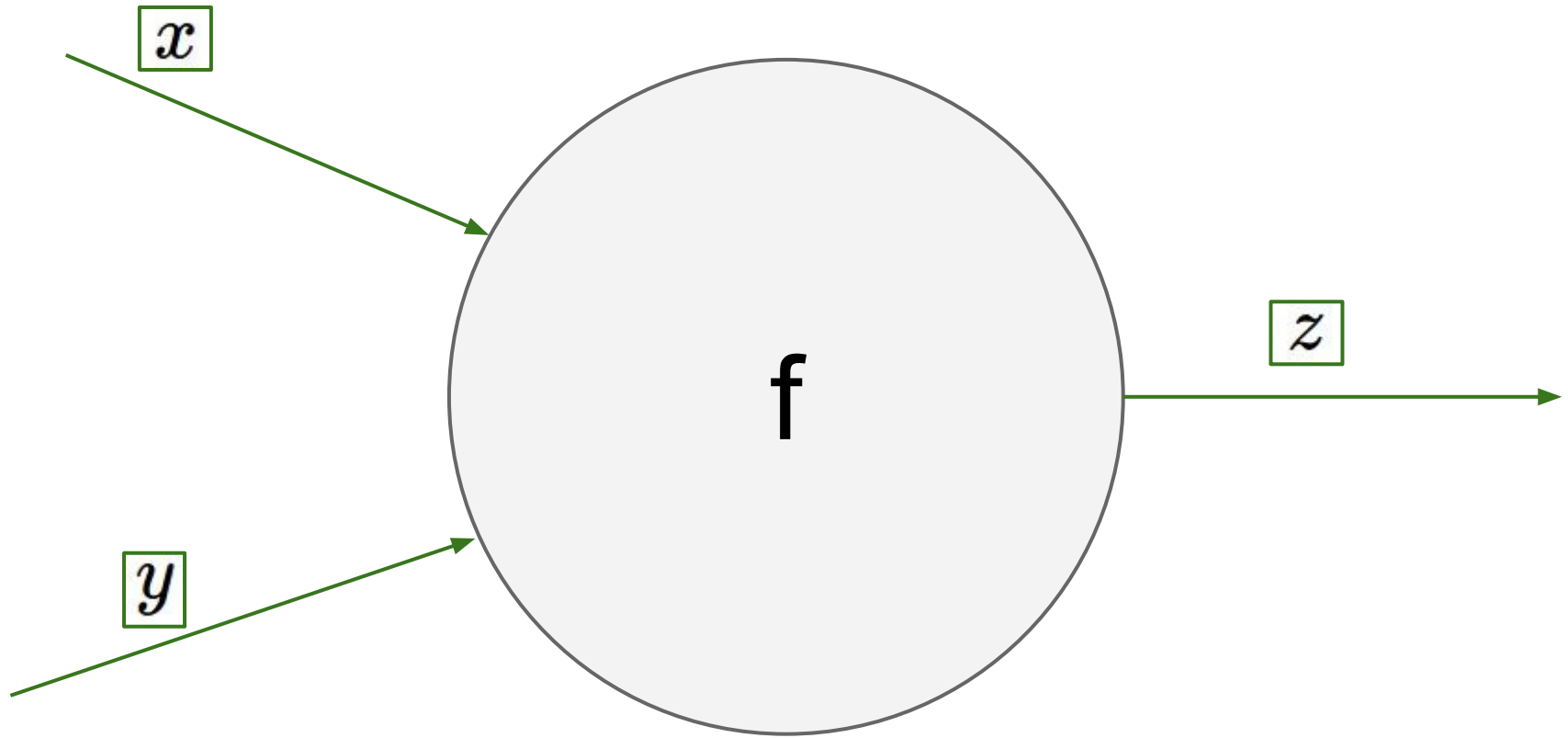
$$\frac{\partial f}{\partial x}$$

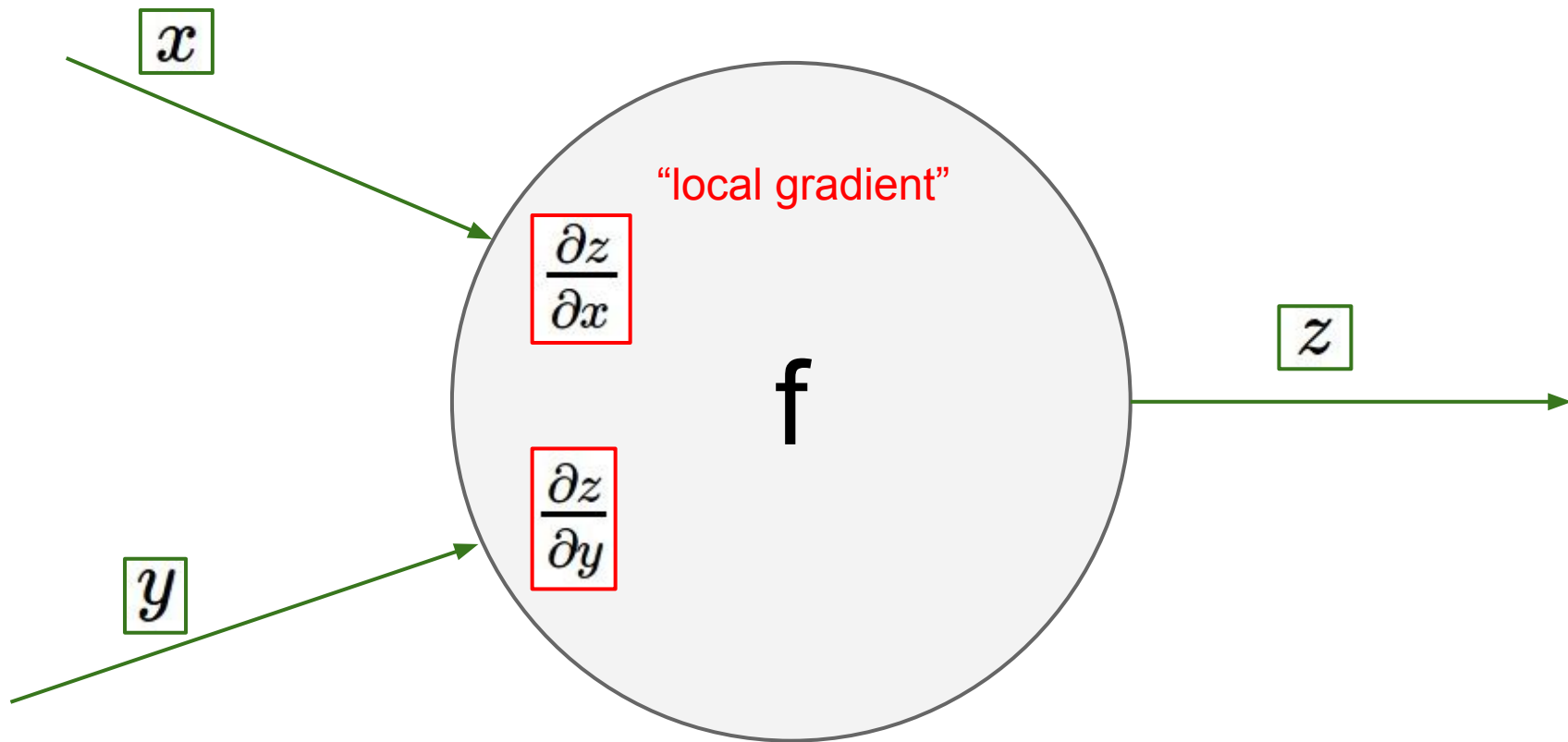
Chain rule:

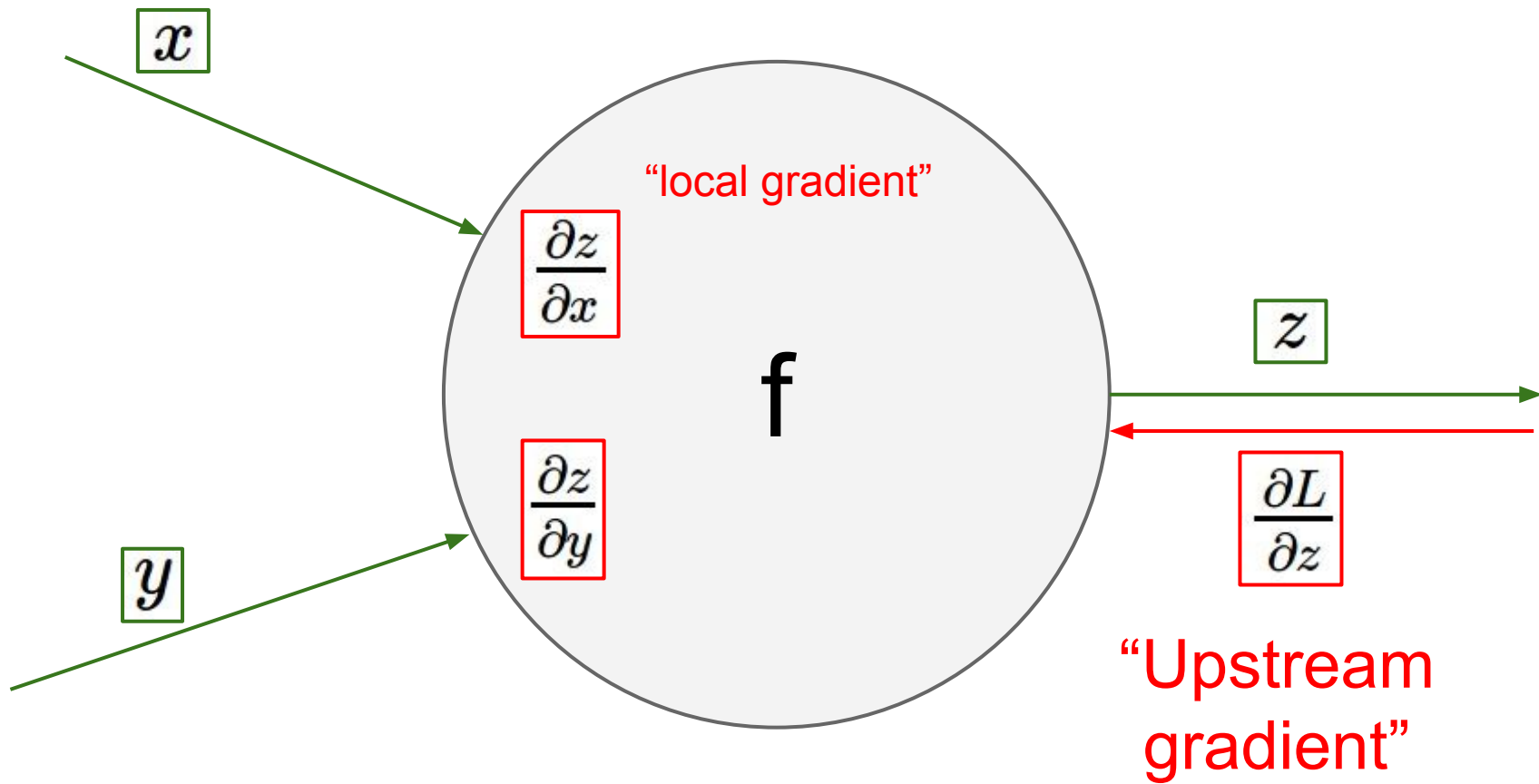
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

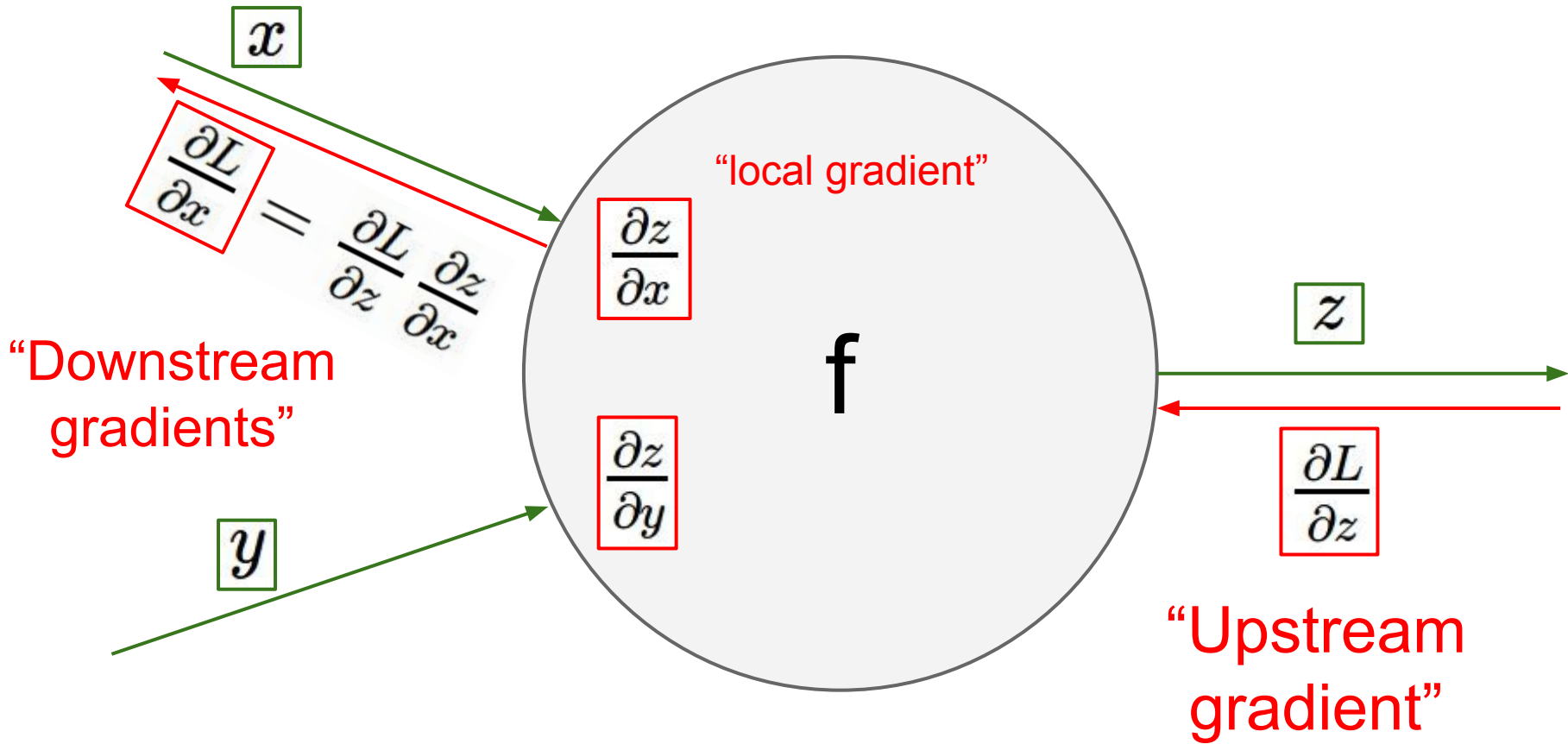
Upstream
gradient

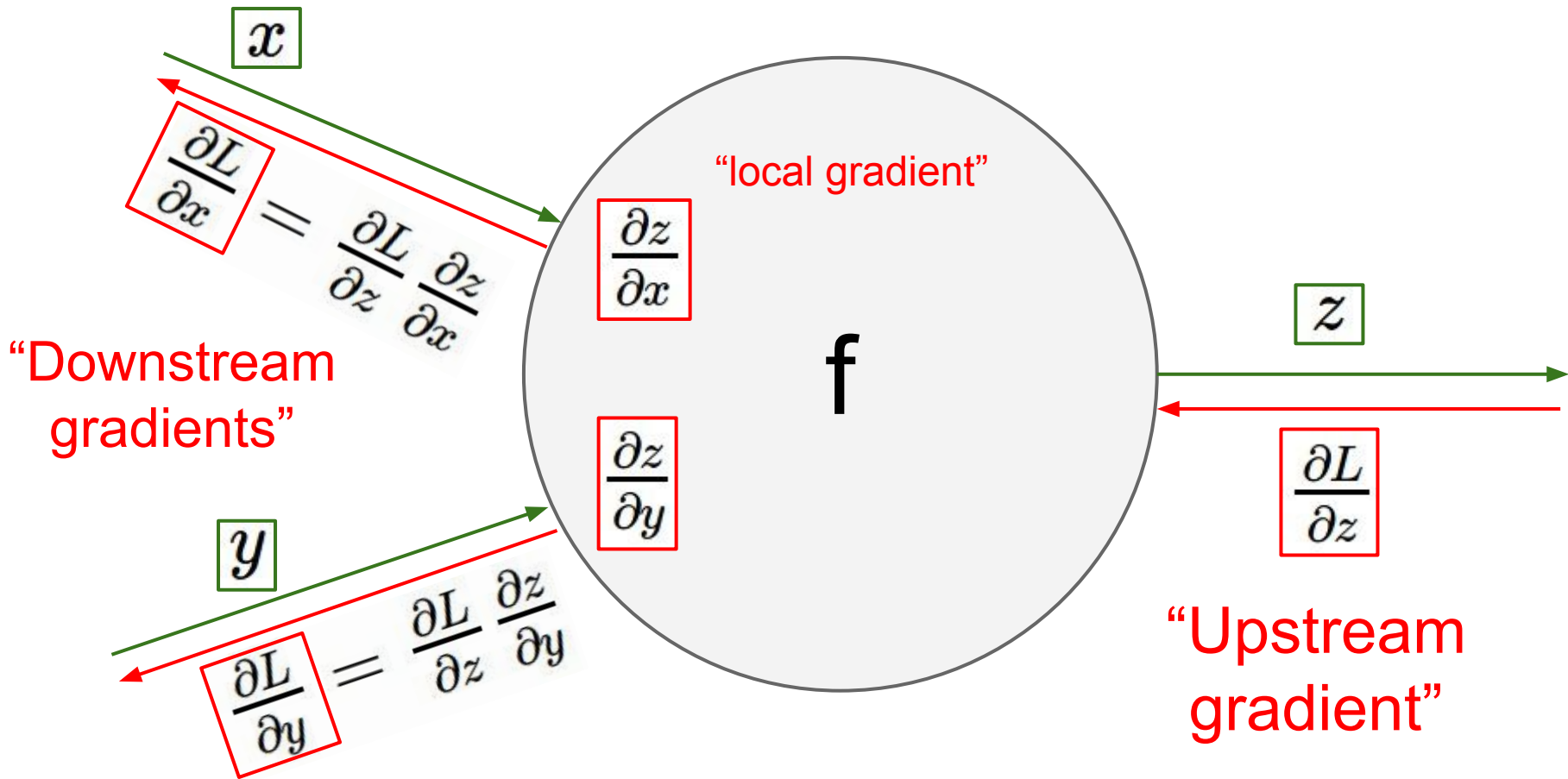
Local
gradient

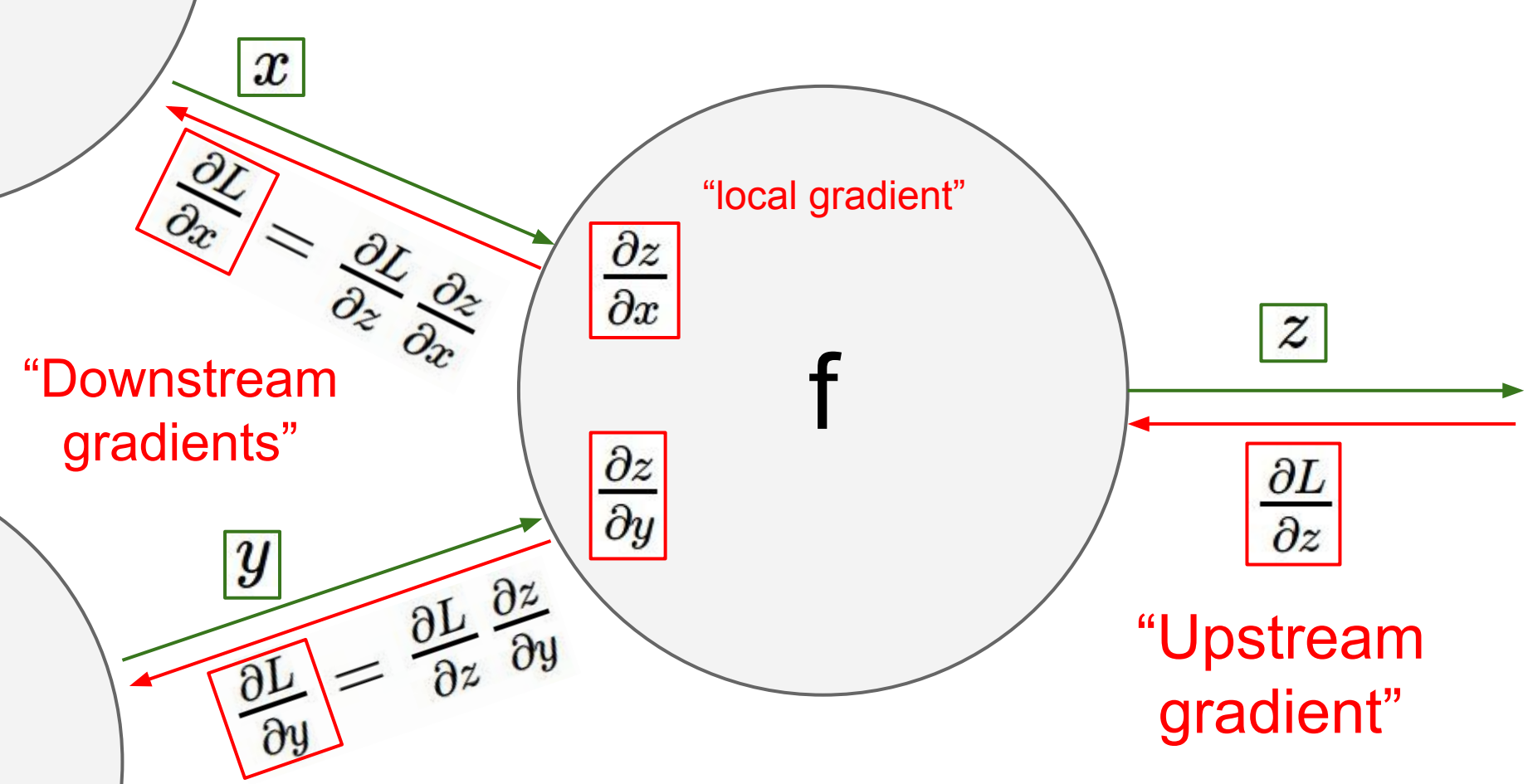




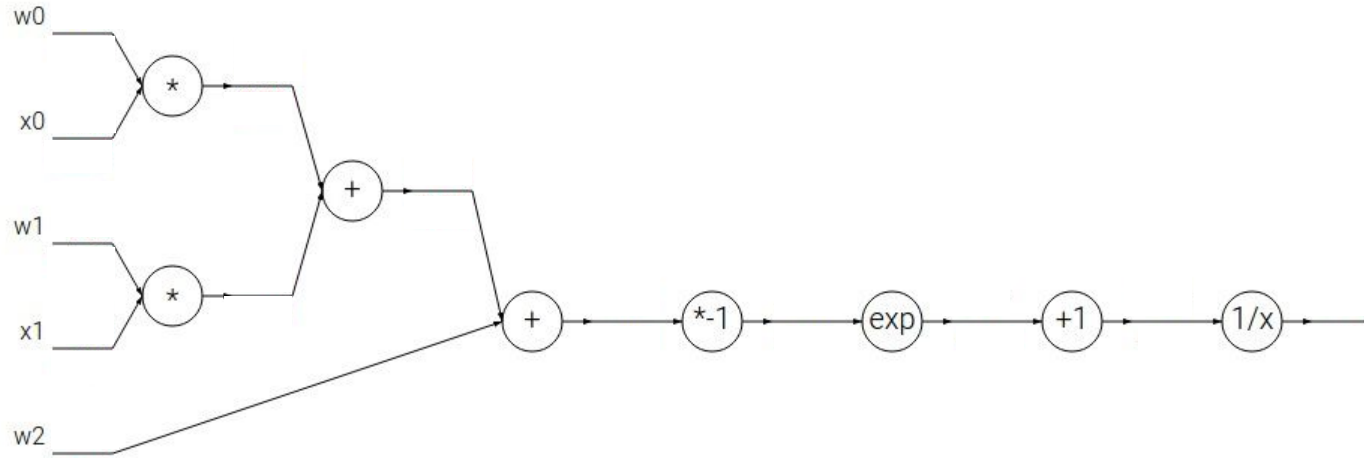






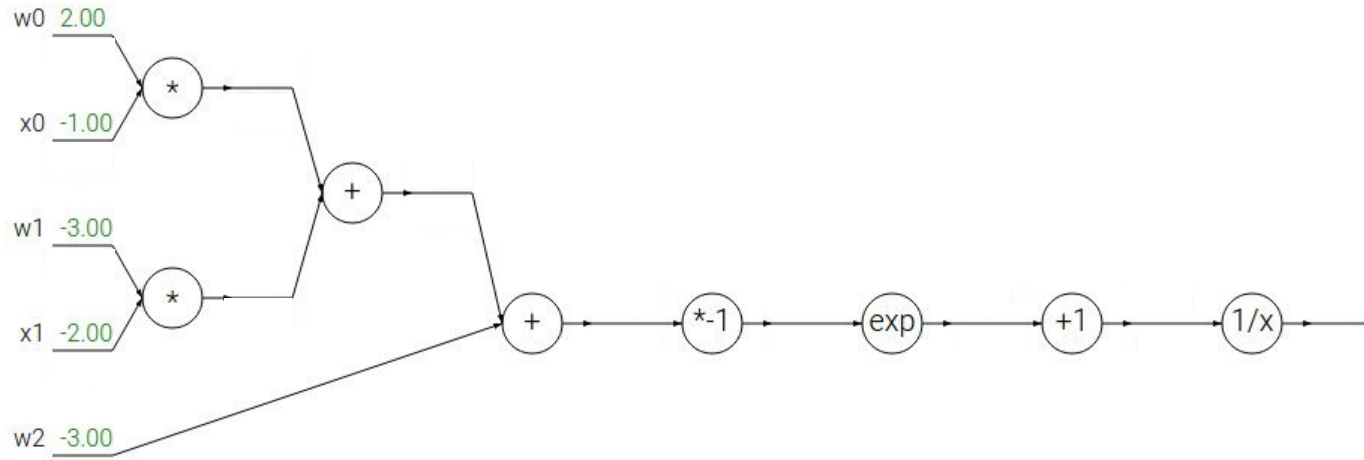


Another example: $f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



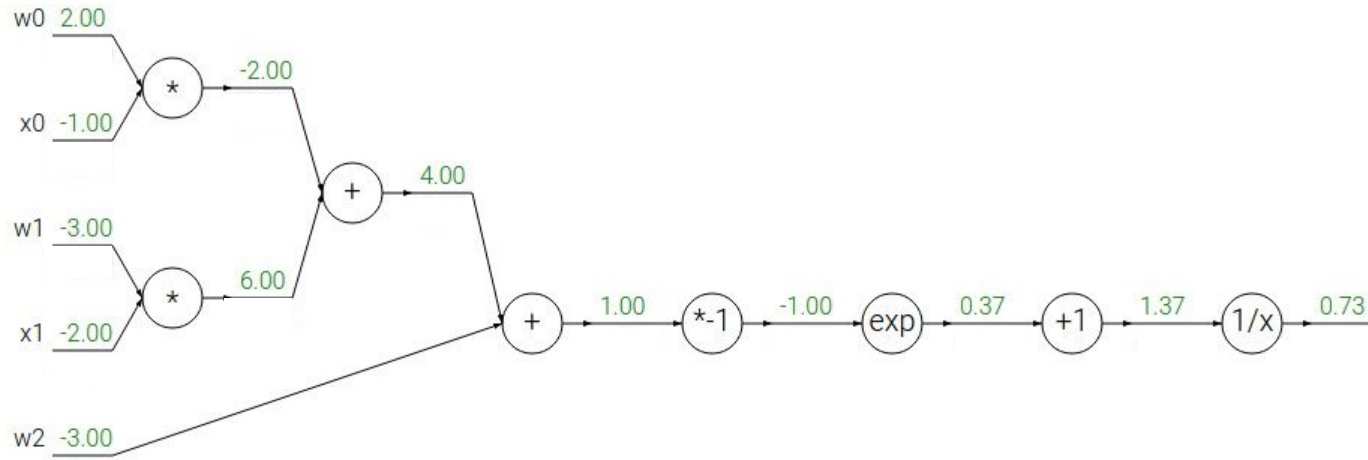
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



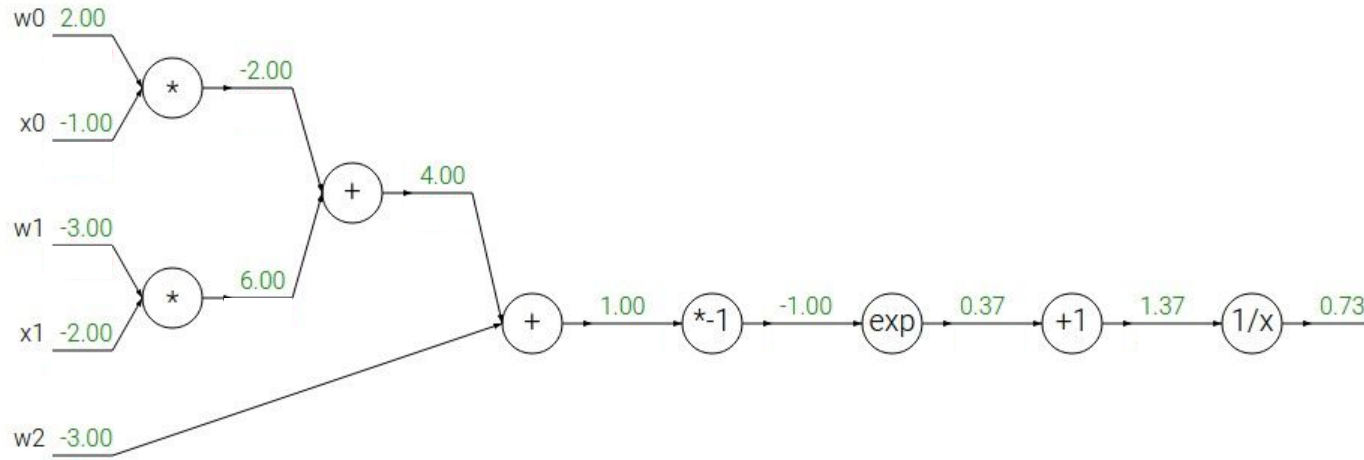
Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



Another example:

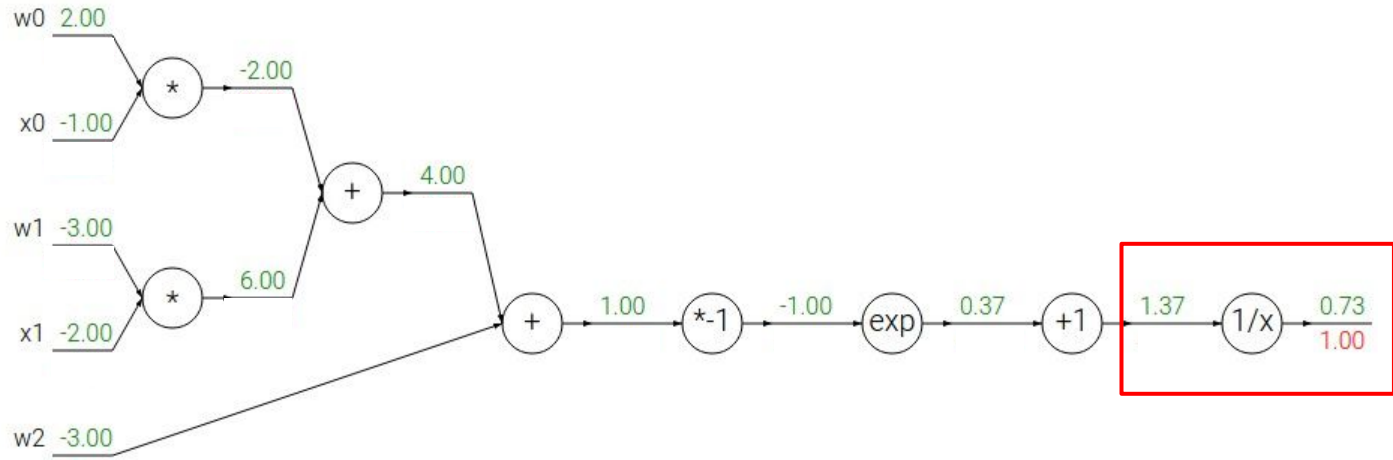
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	→	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	→	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	→	$\frac{df}{dx} = a$		$f_c(x) = c + x$	→	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

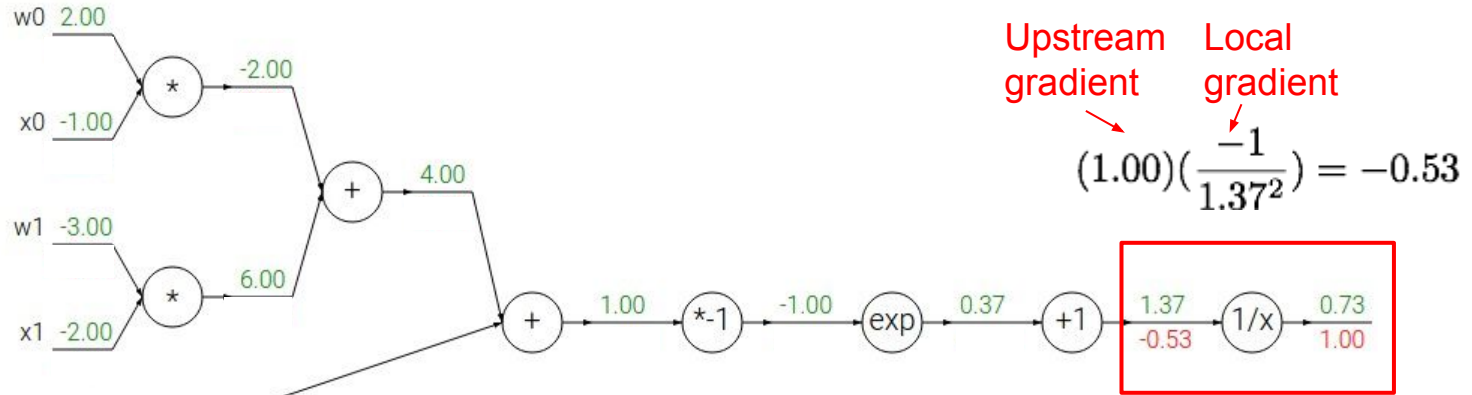
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

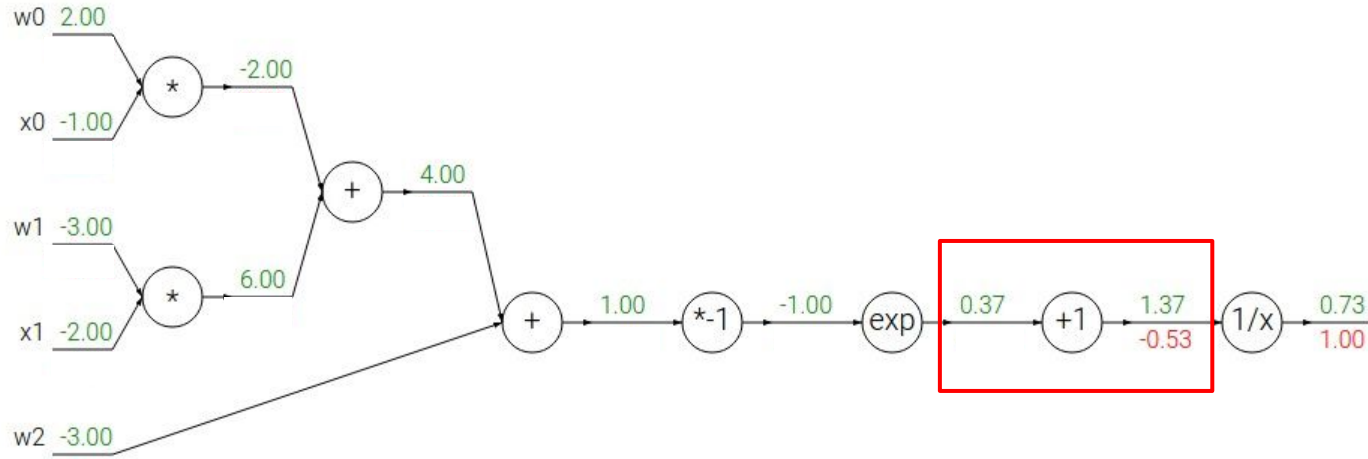
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

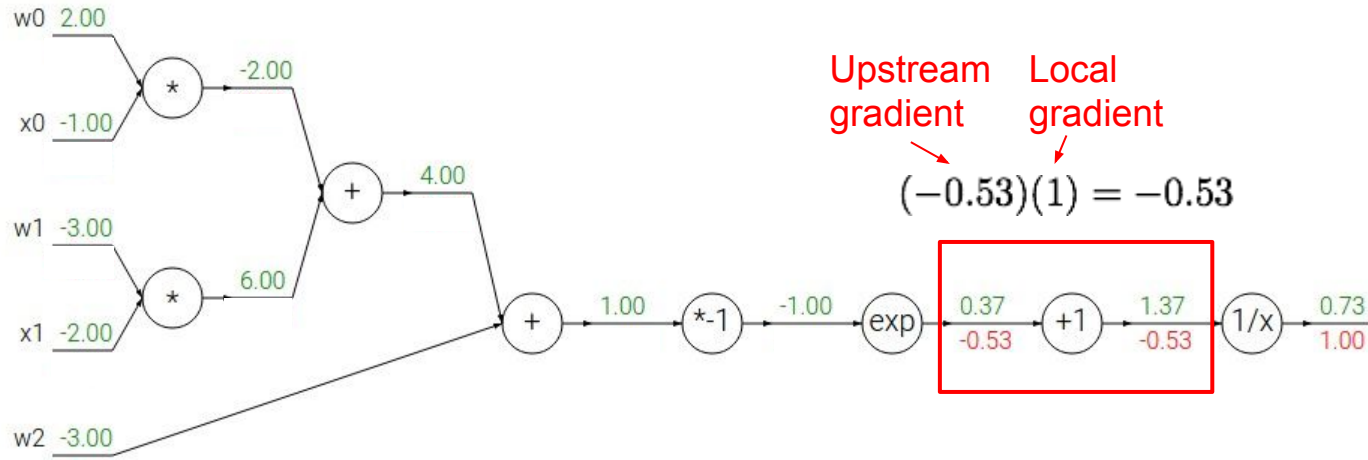
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

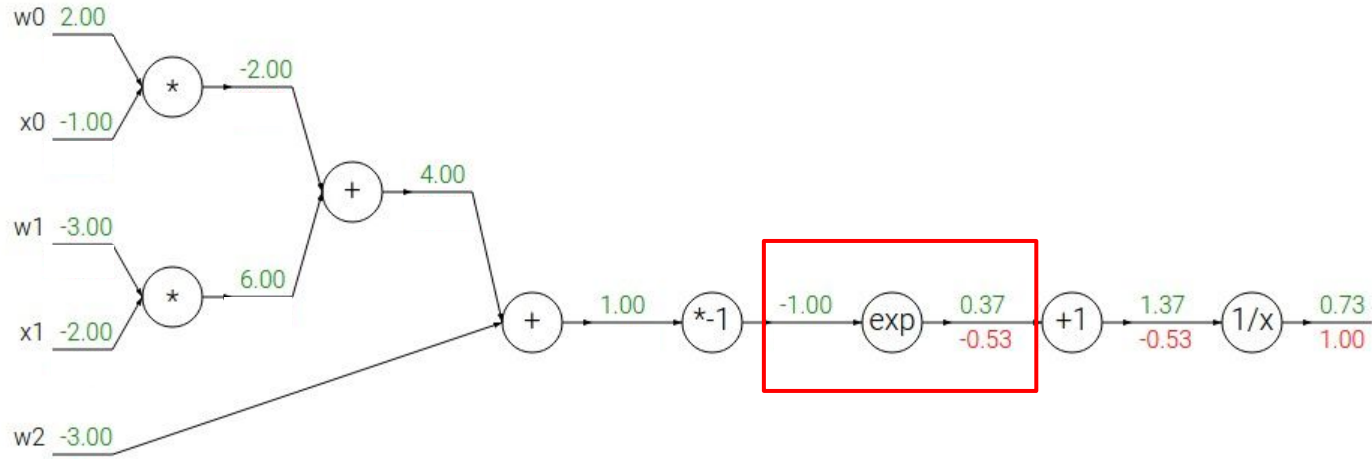
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

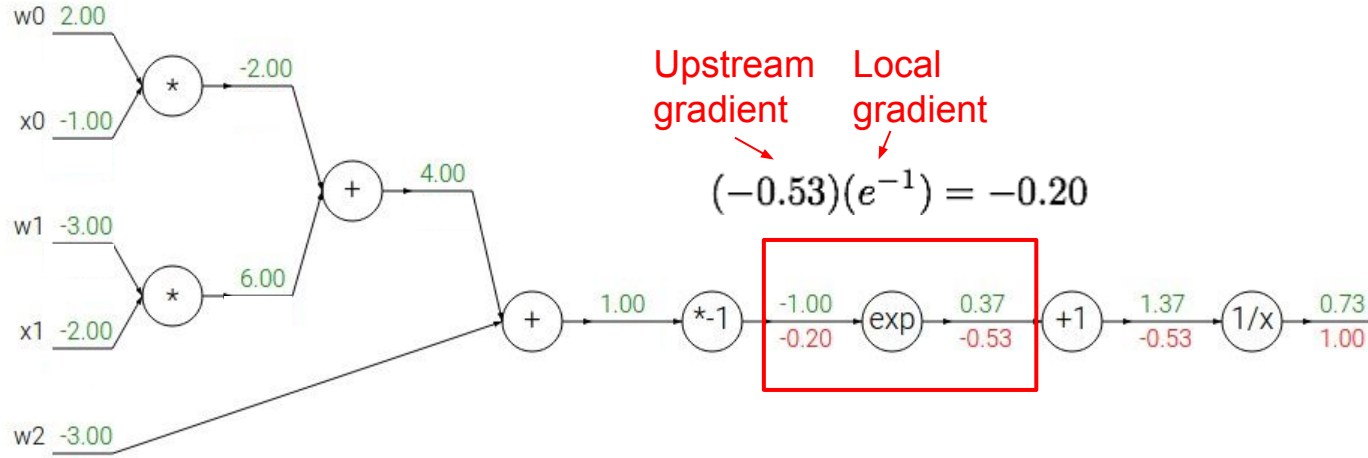
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



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

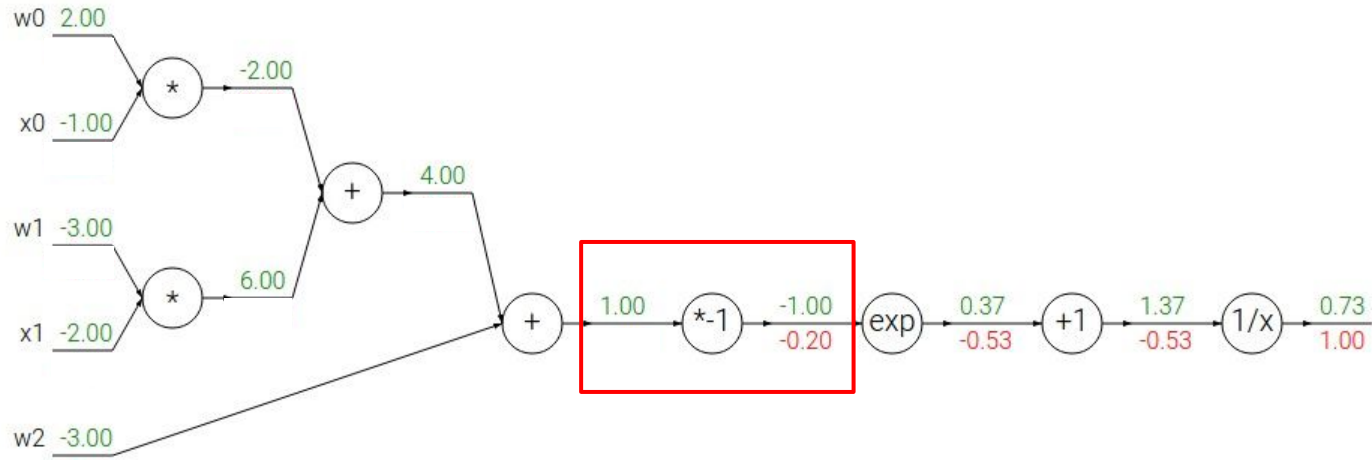
$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

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

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

Another example:

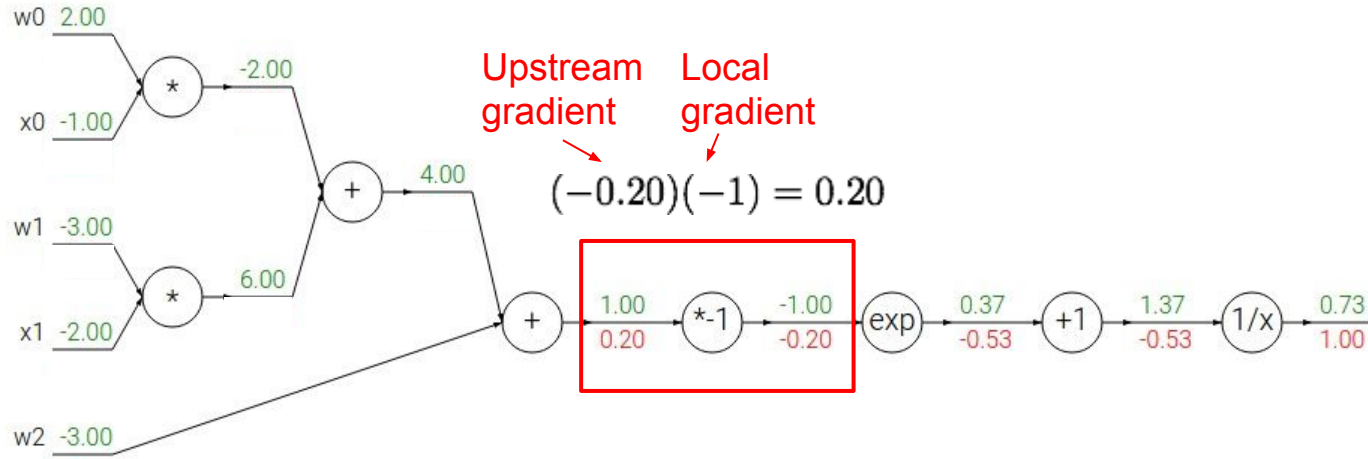
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

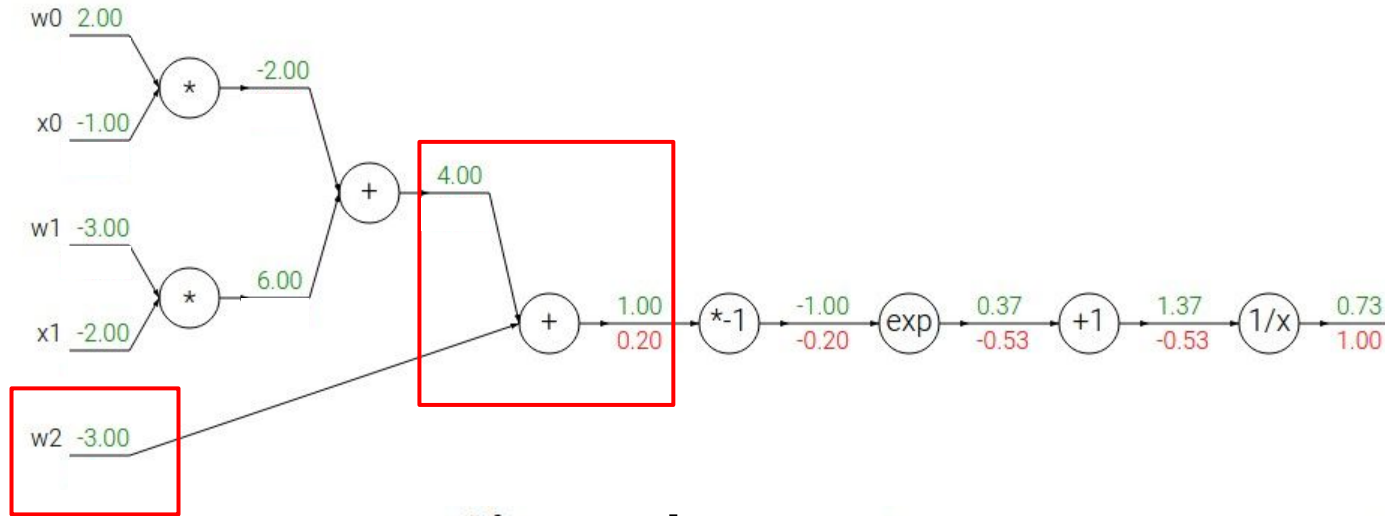
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

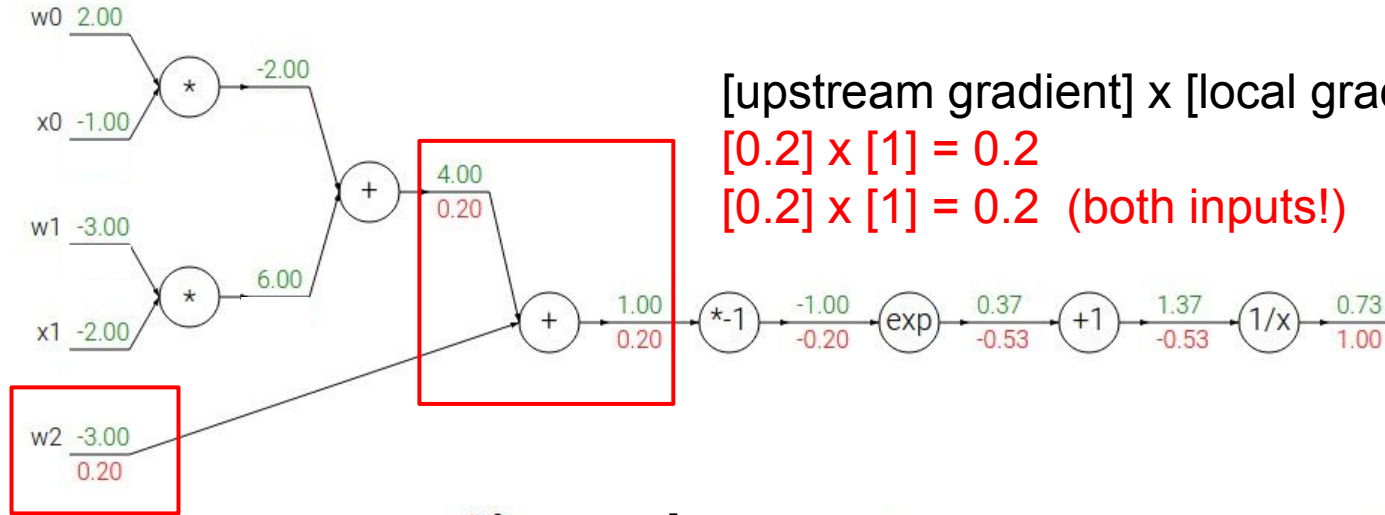
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



[upstream gradient] x [local gradient]

$$[0.2] \times [1] = 0.2$$

$$[0.2] \times [1] = 0.2 \text{ (both inputs!)}$$

$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

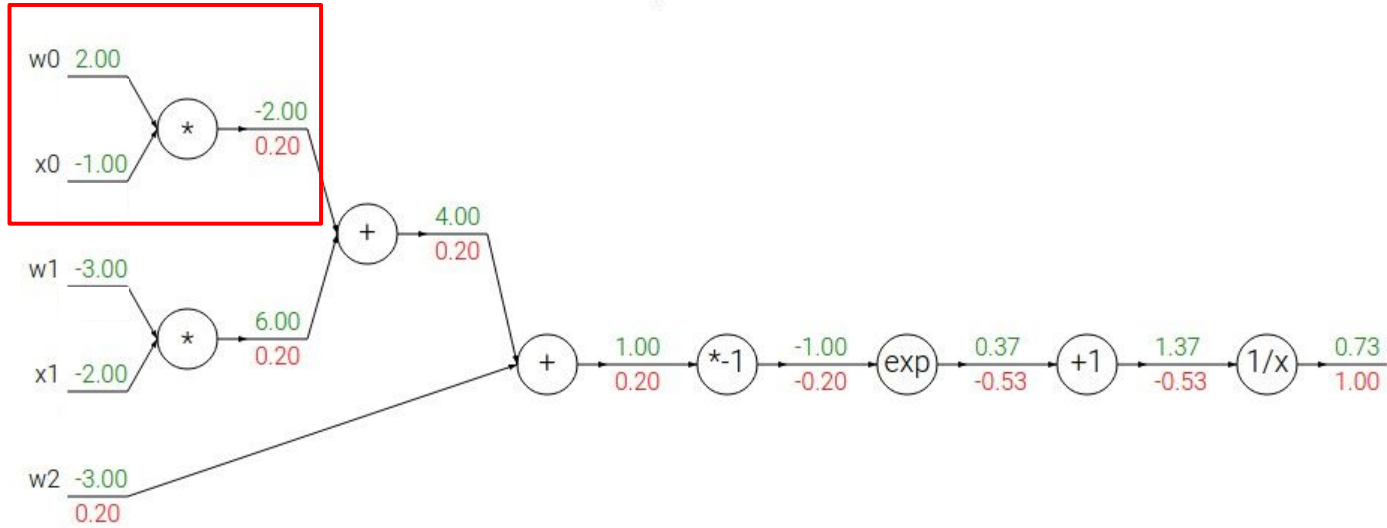
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

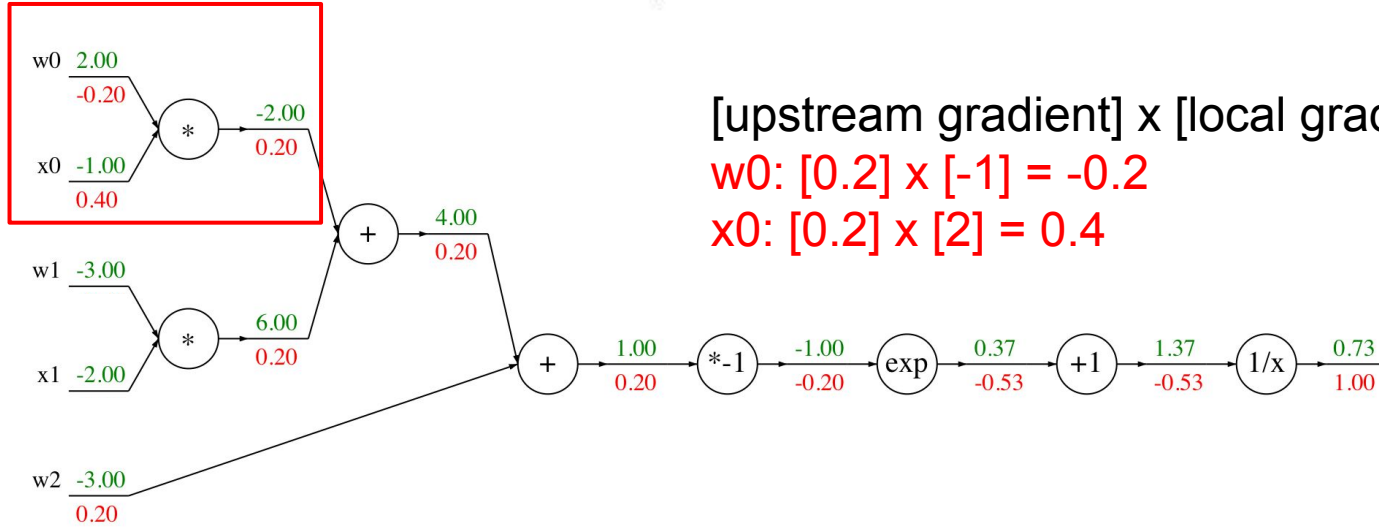
$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$



[upstream gradient] x [local gradient]

$$w_0: [0.2] \times [-1] = -0.2$$

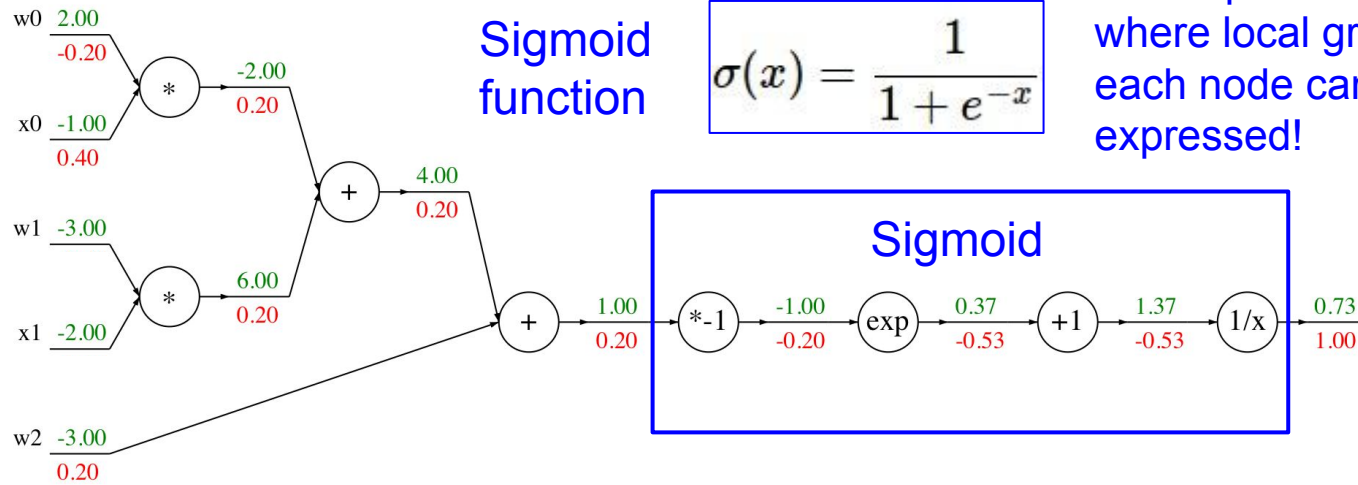
$$x_0: [0.2] \times [2] = 0.4$$

$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$		$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$		$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

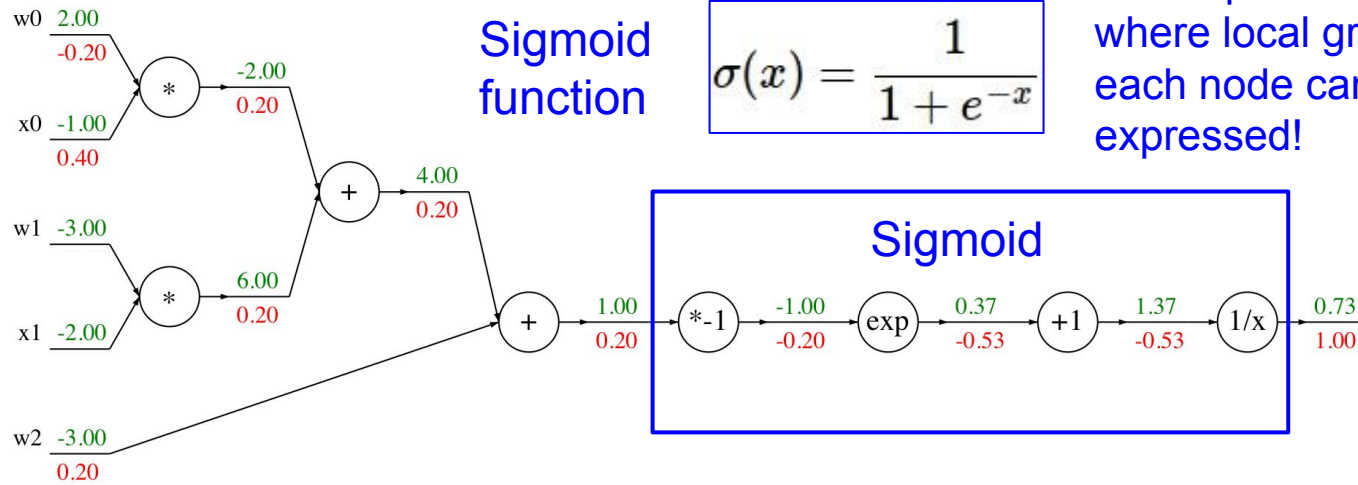
Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



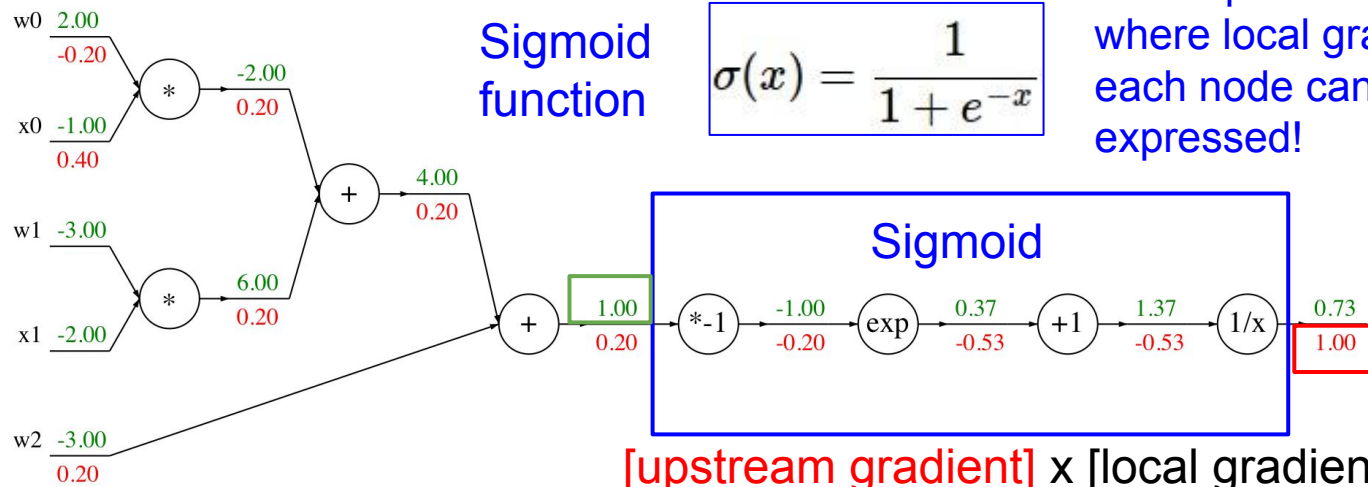
Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$$

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



[upstream gradient] x [local gradient]
 $[1.00] \times [(1 - 1/(1+e^1)) (1/(1+e^1))] = 0.2$

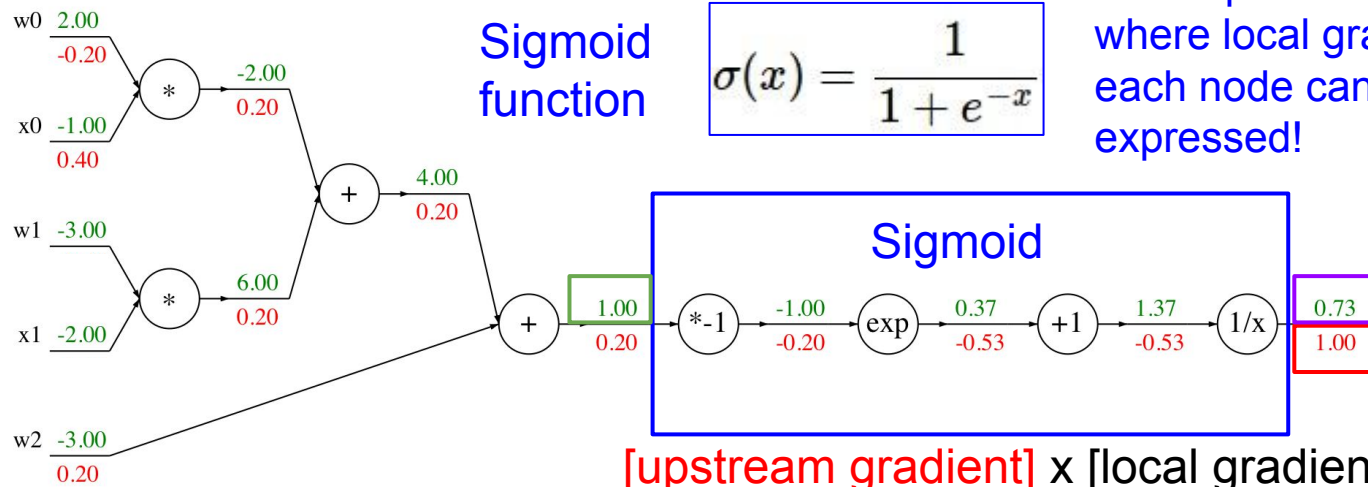
Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

Another example:

$$f(w, x) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2x_2)}}$$

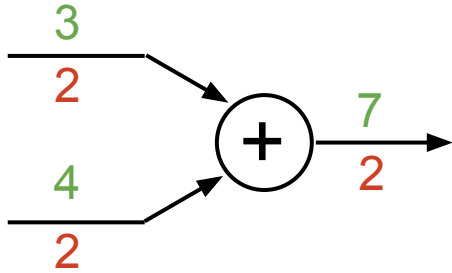
Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!



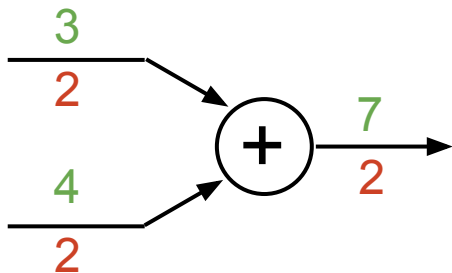
Sigmoid local gradient:

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1 + e^{-x})^2} = \left(\frac{1 + e^{-x} - 1}{1 + e^{-x}} \right) \left(\frac{1}{1 + e^{-x}} \right) = (1 - \sigma(x)) \sigma(x)$$

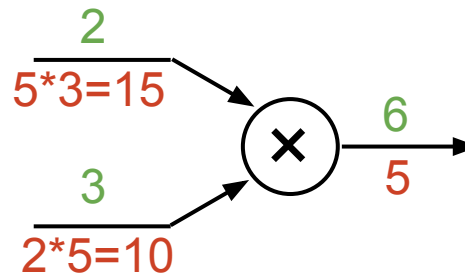
add gate: gradient distributor



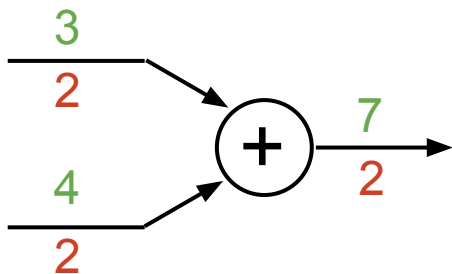
add gate: gradient distributor



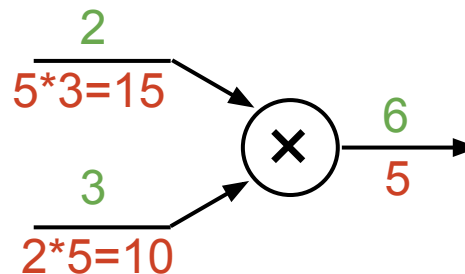
mul gate: “swap multiplier”



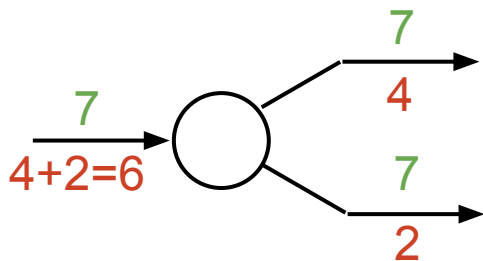
add gate: gradient distributor



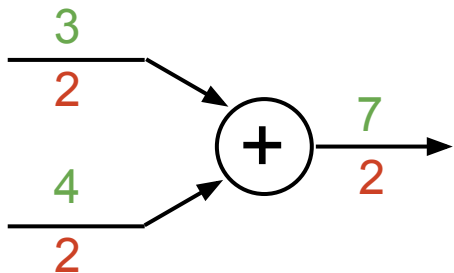
mul gate: “swap multiplier”



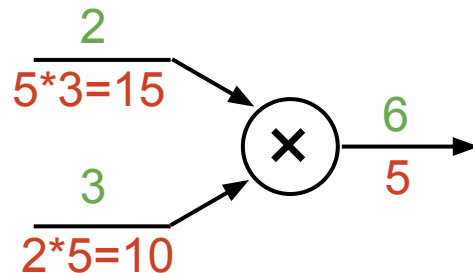
copy gate: gradient adder



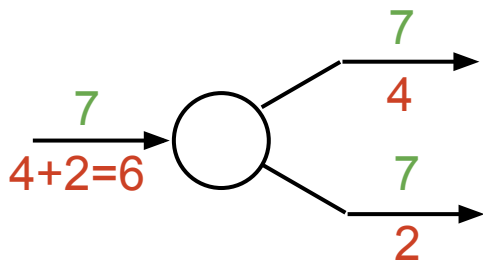
add gate: gradient distributor



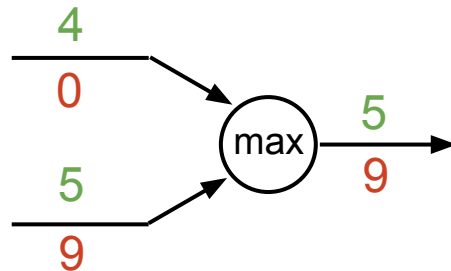
mul gate: “swap multiplier”



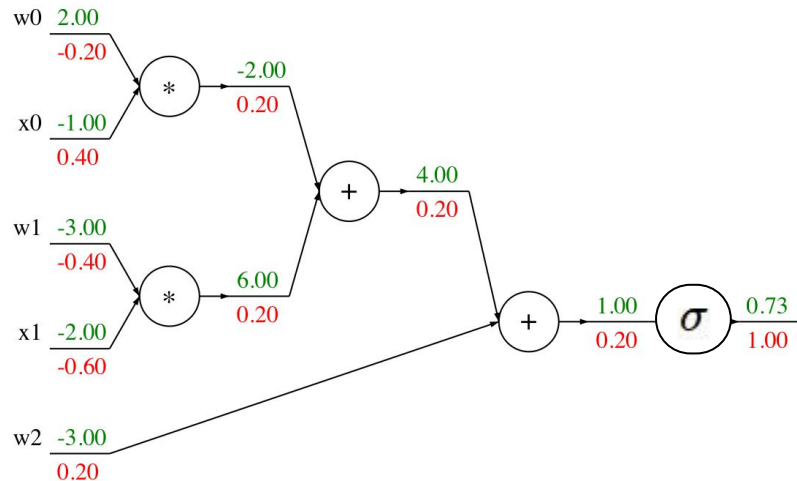
copy gate: gradient adder



max gate: gradient router



Backprop Implementation: “Flat” code



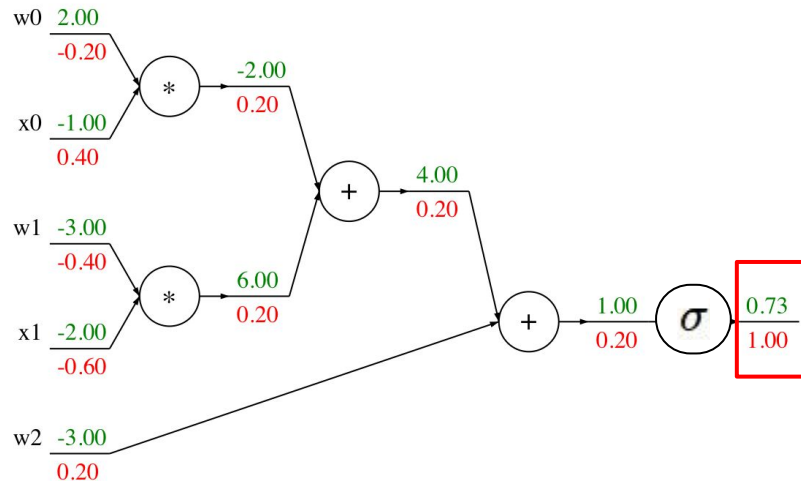
Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

Backward pass:
Compute grads

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

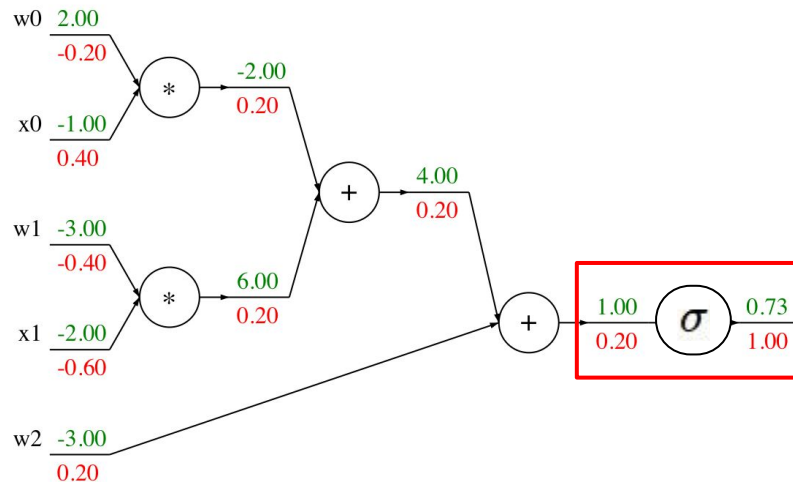
```
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

Base case

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
    grad_w2 = grad_s3
    grad_s2 = grad_s3
    grad_s0 = grad_s2
    grad_s1 = grad_s2
    grad_w1 = grad_s1 * x1
    grad_x1 = grad_s1 * w1
    grad_w0 = grad_s0 * x0
    grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

Sigmoid

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
```

```
    grad_w2 = grad_s3
```

```
    grad_s2 = grad_s3
```

```
    grad_s0 = grad_s2
```

```
    grad_s1 = grad_s2
```

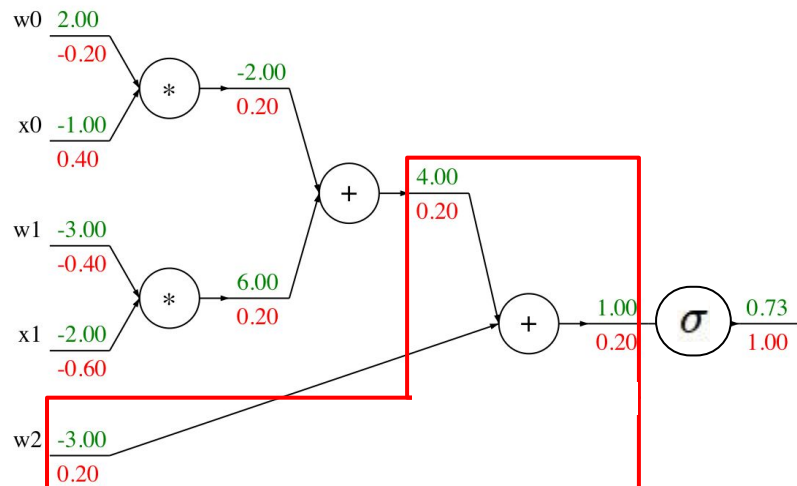
```
    grad_w1 = grad_s1 * x1
```

```
    grad_x1 = grad_s1 * w1
```

```
    grad_w0 = grad_s0 * x0
```

```
    grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



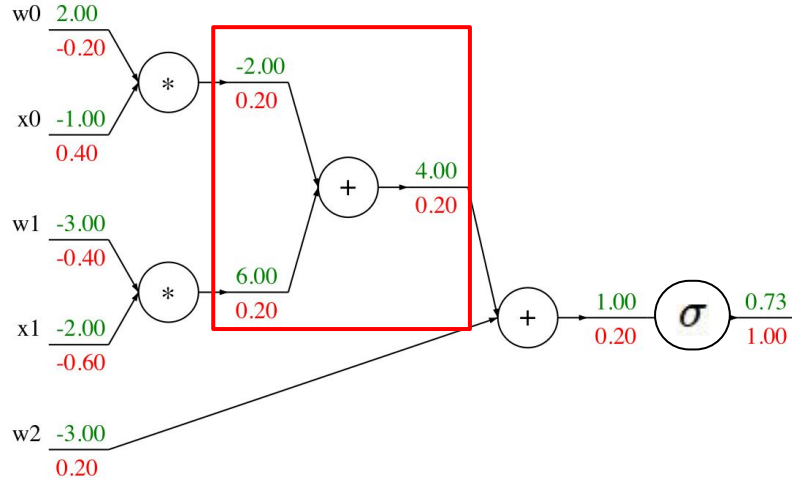
Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

Add gate

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
    grad_L = 1.0
```

```
    grad_s3 = grad_L * (1 - L) * L
```

```
    grad_w2 = grad_s3
```

```
    grad_s2 = grad_s3
```

```
    grad_s0 = grad_s2
```

```
    grad_s1 = grad_s2
```

```
    grad_w1 = grad_s1 * x1
```

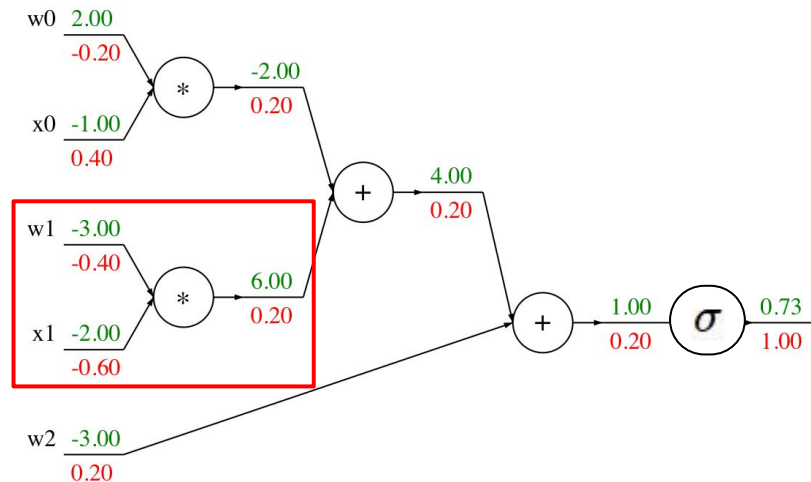
```
    grad_x1 = grad_s1 * w1
```

```
    grad_w0 = grad_s0 * x0
```

```
    grad_x0 = grad_s0 * w0
```

Add gate

Backprop Implementation: “Flat” code



Forward pass:
Compute output

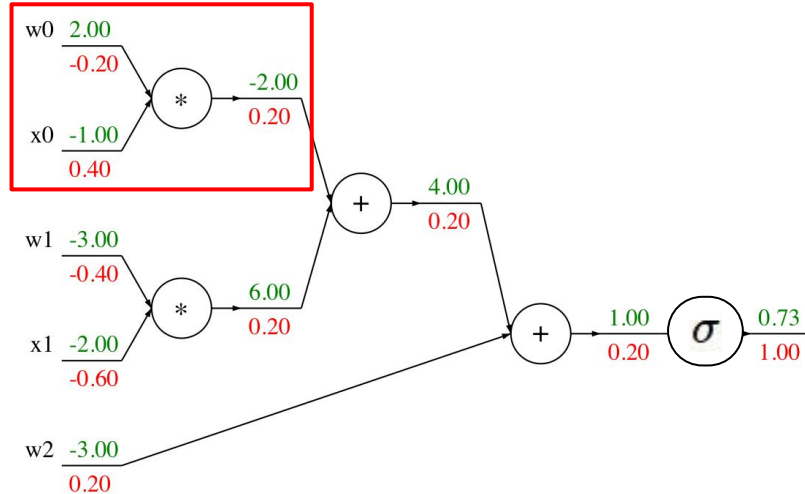
```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
    s1 = w1 * x1
    s2 = s0 + s1
    s3 = s2 + w2
    L = sigmoid(s3)
```

```
grad_L = 1.0
grad_s3 = grad_L * (1 - L) * L
grad_w2 = grad_s3
grad_s2 = grad_s3
grad_s0 = grad_s2
grad_s1 = grad_s2
grad_w1 = grad_s1 * x1
grad_x1 = grad_s1 * w1
grad_w0 = grad_s0 * x0
grad_x0 = grad_s0 * w0
```

Multiply gate

Backprop Implementation: “Flat” code



Forward pass:
Compute output

```
def f(w0, x0, w1, x1, w2):
```

```
    s0 = w0 * x0
```

```
    s1 = w1 * x1
```

```
    s2 = s0 + s1
```

```
    s3 = s2 + w2
```

```
    L = sigmoid(s3)
```

```
grad_L = 1.0
```

```
grad_s3 = grad_L * (1 - L) * L
```

```
grad_w2 = grad_s3
```

```
grad_s2 = grad_s3
```

```
grad_s0 = grad_s2
```

```
grad_s1 = grad_s2
```

```
grad_w1 = grad_s1 * x1
```

```
grad_x1 = grad_s1 * w1
```

```
grad_w0 = grad_s0 * x0
```

```
grad_x0 = grad_s0 * w0
```

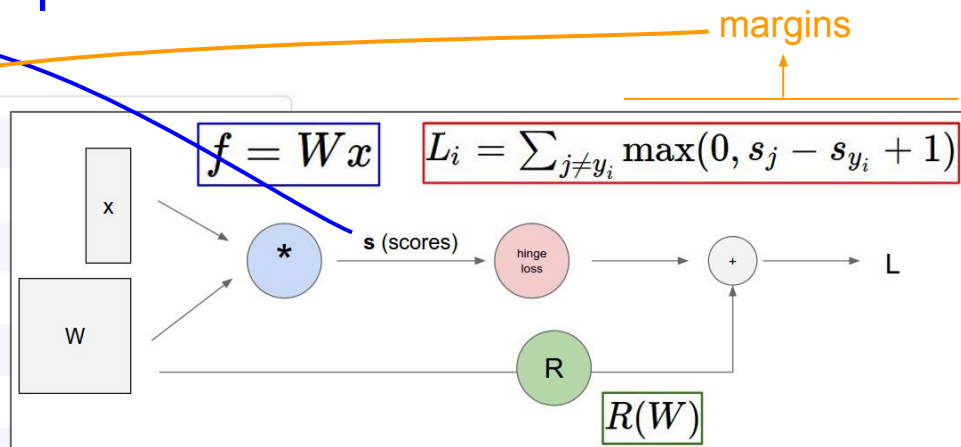
Multiply gate

“Flat” Backprop: Do this for assignment 1!

Stage your forward/backward computation!

E.g. for the SVM:

```
# receive W (weights), X (data)
# forward pass (we have 8 lines)
scores = #...
margins = #...
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss
# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```



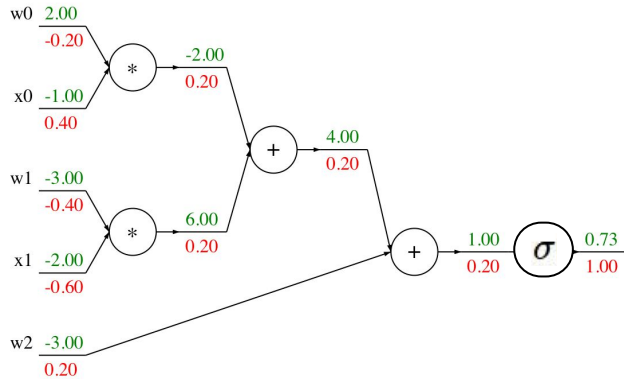
“Flat” Backprop: Do this for assignment 1!

E.g. for two-layer neural net:

```
# receive W1,W2,b1,b2 (weights/biases), X (data)
# forward pass:
h1 = #... function of X,W1,b1
scores = #... function of h1,W2,b2
loss = #... (several lines of code to evaluate Softmax loss)
# backward pass:
dscores = #...
dh1,dW2,db2 = #...
dW1,db1 = #...
```

Backprop Implementation: Modularized API

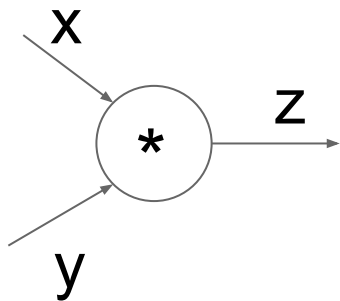
Graph (or Net) object (*rough pseudo code*)



```
class ComputationalGraph(object):  
    #...  
    def forward(inputs):  
        # 1. [pass inputs to input gates...]  
        # 2. forward the computational graph:  
        for gate in self.graph.nodes_topologically_sorted():  
            gate.forward()  
        return loss # the final gate in the graph outputs the loss  
    def backward():  
        for gate in reversed(self.graph.nodes_topologically_sorted()):  
            gate.backward() # little piece of backprop (chain rule applied)  
        return inputs_gradients
```

Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code



(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y)  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z):  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y
```

Need to stash
some values for
use in backward

Upstream
gradient

Multiply upstream
and local gradients

Example: PyTorch operators

pytorch / pytorch

Watch

1,221

★ Unstar

26,770

Y Fork

6,340

< Code

Issues 2,286

Pull requests 561

Projects 4

Wiki

Insights

Tree: 517c7c9861

pytorch / aten / src / THNN / generic /

Create new file

Upload files

Find file

History

ezyang and facebook-github-bot

Canonicalize all includes in PyTorch. (#14849)

Latest commit 517c7c9 on Dec 8, 2018

..

AbsCriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

BCECriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

ClassNLLCriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

Col2Im.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

ELU.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

FeatureLPPooling.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

GatedLinearUnit.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

HardTanh.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

Im2Col.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

IndexLinear.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

LeakyReLU.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

LogSigmoid.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

MSECriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

MultiLabelMarginCriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

MultiMarginCriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

RRReLU.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

Sigmoid.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SmoothL1Criterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SoftMarginCriterion.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SoftPlus.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SoftShrink.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SparseLinear.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SpatialAdaptiveAveragePooling.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SpatialAdaptiveMaxPooling.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SpatialAveragePooling.c

Canonicalize all includes in PyTorch. (#14849)

4 months ago

SpatialClassNLLCriterion.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialConvolutionMM.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialDilatedConvolution.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialDilatedMaxPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialFractionalMaxPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialFullDilatedConvolution.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialMaxUnpooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialReflectionPadding.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialReplicationPadding.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialUpSamplingBilinear.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
SpatialUpSamplingNearest.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
THNN.h	Canonicalize all includes in PyTorch. (#14849)	4 months ago
Tanh.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
TemporalReflectionPadding.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
TemporalReplicationPadding.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
TemporalRowConvolution.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
TemporalUpSamplingLinear.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
TemporalUpSamplingNearest.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricAdaptiveAveragePooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricAdaptiveMaxPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricAveragePooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricConvolutionMM.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricDilatedConvolution.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricDilatedMaxPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricFractionalMaxPooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricFullDilatedConvolution.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricMaxUnpooling.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricReplicationPadding.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricUpSamplingNearest.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
VolumetricUpSamplingTrilinear.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago
linear_upsampling.h	Implement nn.functional.interpolate based on upsample. (#8591)	9 months ago
pooling_shape.h	Use integer math to compute output size of pooling operations (#14405)	4 months ago
unfold.c	Canonicalize all includes in PyTorch. (#14849)	4 months ago

PyTorch sigmoid layer

Forward

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



```
void THNN_(Sigmoid_updateOutput)(
    THNNState *state,
    THTensor *input,
    THTensor *output)
{
    THTensor_(sigmoid)(output, input);
}
```

```
void THNN_(Sigmoid_updateGradInput)(
    THNNState *state,
    THTensor *gradOutput,
    THTensor *gradInput,
    THTensor *output)
{
    THNN_CHECK_NELEMENT(output, gradOutput);
    THTensor_(resizeAs)(gradInput, output);
    TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
        scalar_t z = *output_data;
        *gradInput_data = *gradOutput_data * (1. - z) * z;
    );
}
```

#endif

[Source](#)

PyTorch sigmoid layer

Forward

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

```
void THNN_(Sigmoid_updateOutput)(
    THNNState *state,
    THTensor *input,
    THTensor *output)
{
    THTensor_(sigmoid)(output, input);
}
```

```
void THNN_(Sigmoid_updateGradInput)(
    THNNState *state,
    THTensor *gradOutput,
    THTensor *gradInput,
    THTensor *output)
{
    THNN_CHECK_NELEMENT(output, gradOutput);
    THTensor_(resizeAs)(gradInput, output);
    TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
        scalar_t z = *output_data;
        *gradInput_data = *gradOutput_data * (1. - z) * z;
    );
}
```

#endif

```
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a))))}; },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>((scalar_t)(0)) - a;
                a = a.exp();
                a = Vec256<scalar_t>((scalar_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Forward actually
defined elsewhere...

```
return (1 / (1 + std::exp((-a))));
```

[Source](#)

PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
```

```
4
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10     THTensor_(sigmoid)(output, input);
11 }
```

Forward

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

```
12
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
```

Backward

$$(1 - \sigma(x)) \sigma(x)$$

```
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>((scalar_t)(0)) - a;
                a = a.exp();
                a = Vec256<scalar_t>((scalar_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Forward actually defined [elsewhere...](#)

[Source](#)

So far: backprop with scalars

What about vector-valued functions?

Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector to Scalar

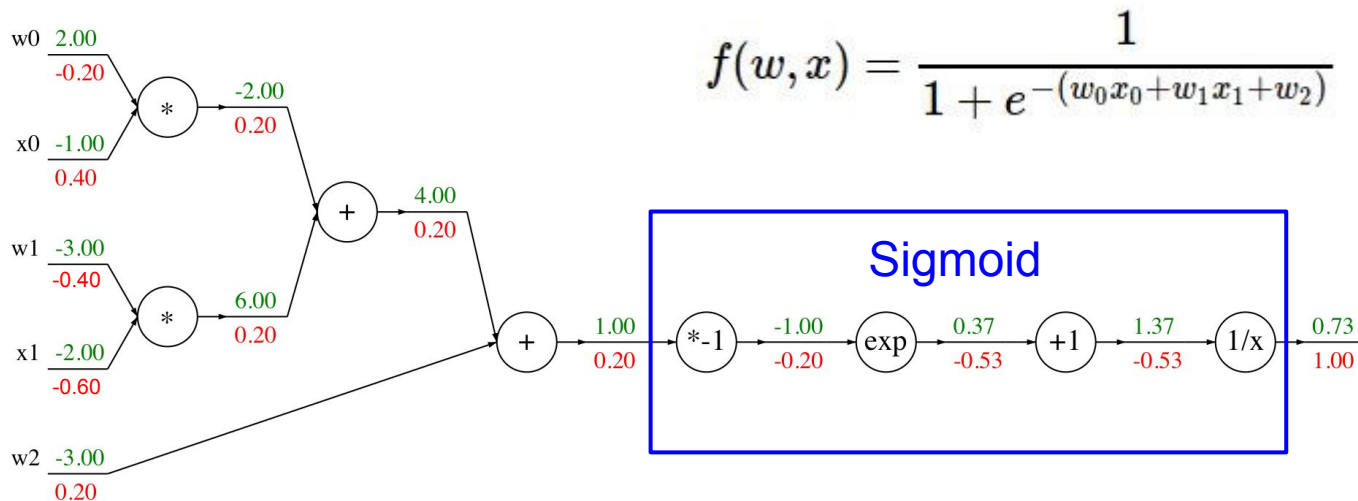
$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will y change?

Returning to this example



$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

Vector to Scalar

$$\begin{bmatrix} -1.00 \\ -2.00 \end{bmatrix} x \in \mathbb{R}^N, y \in \mathbb{R} \quad 0.73$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n} \quad \begin{bmatrix} 0.40 \\ -0.60 \end{bmatrix}$$

Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If x changes by a small amount, how much will y change?

Vector to Scalar

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is **Gradient**:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will y change?

Vector to Vector

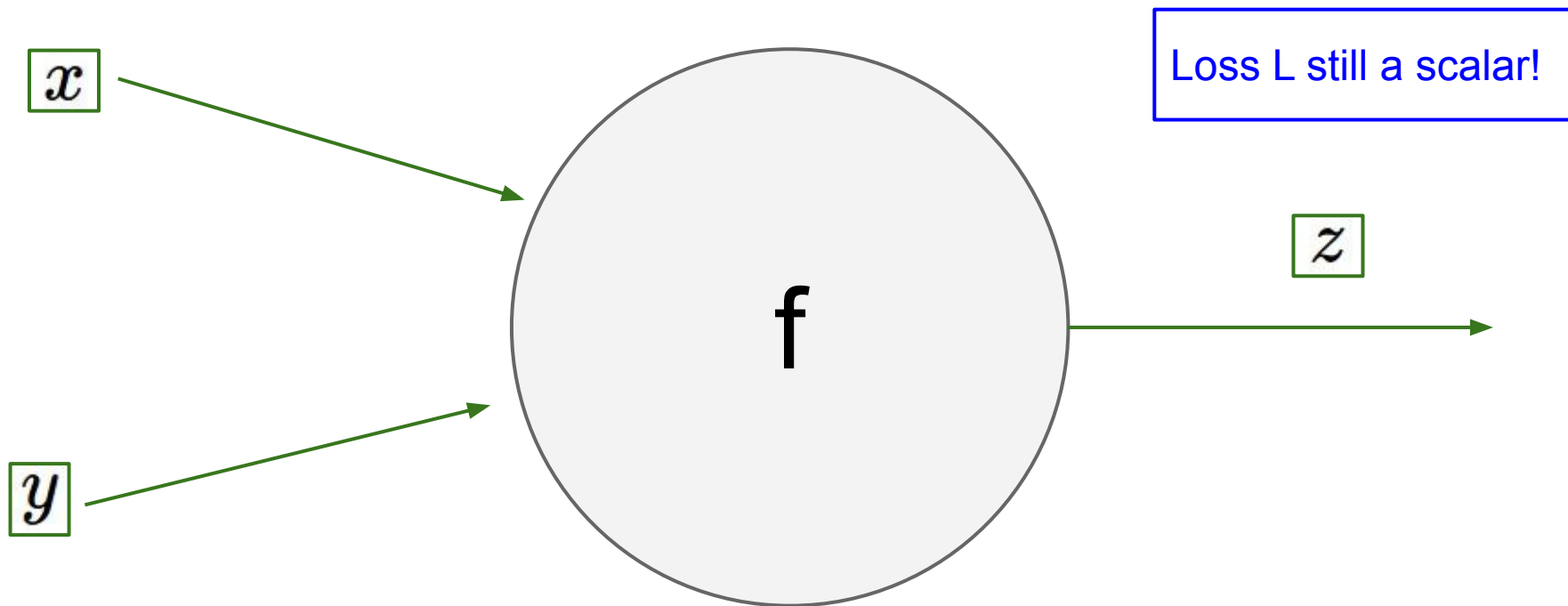
$$x \in \mathbb{R}^N, y \in \mathbb{R}^M$$

Derivative is **Jacobian**:

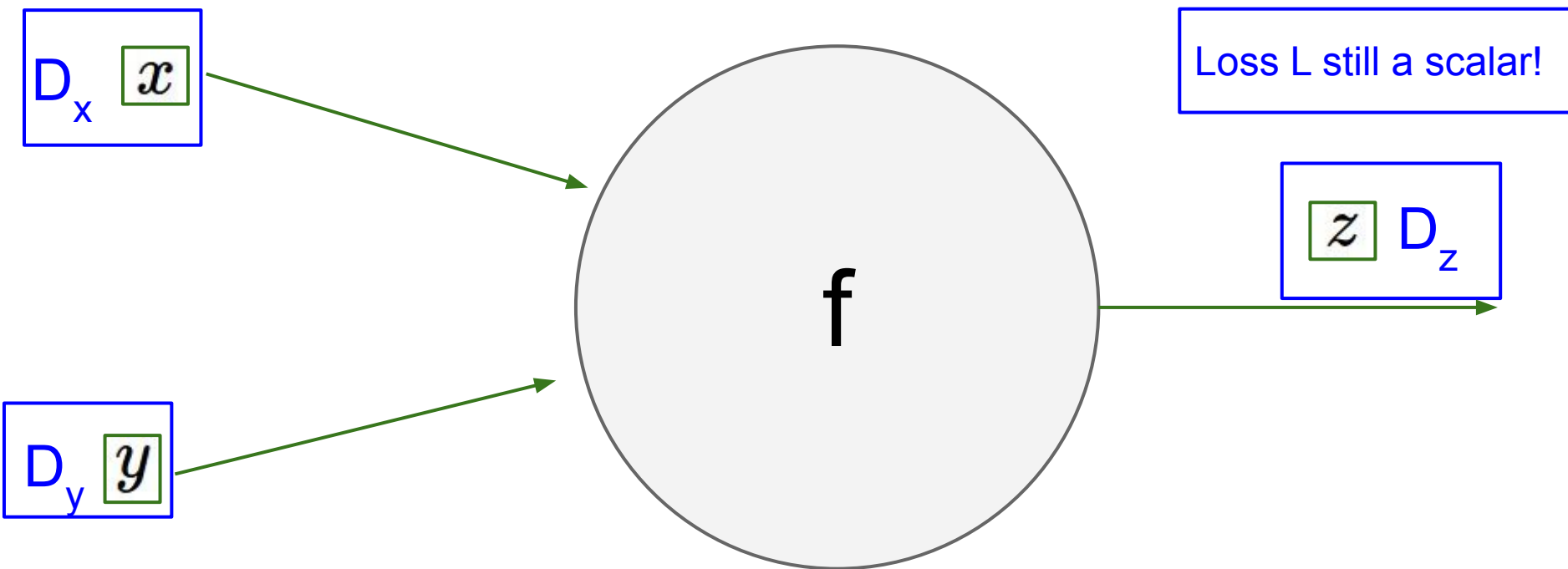
$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

For each element of x , if it changes by a small amount then how much will each element of y change?

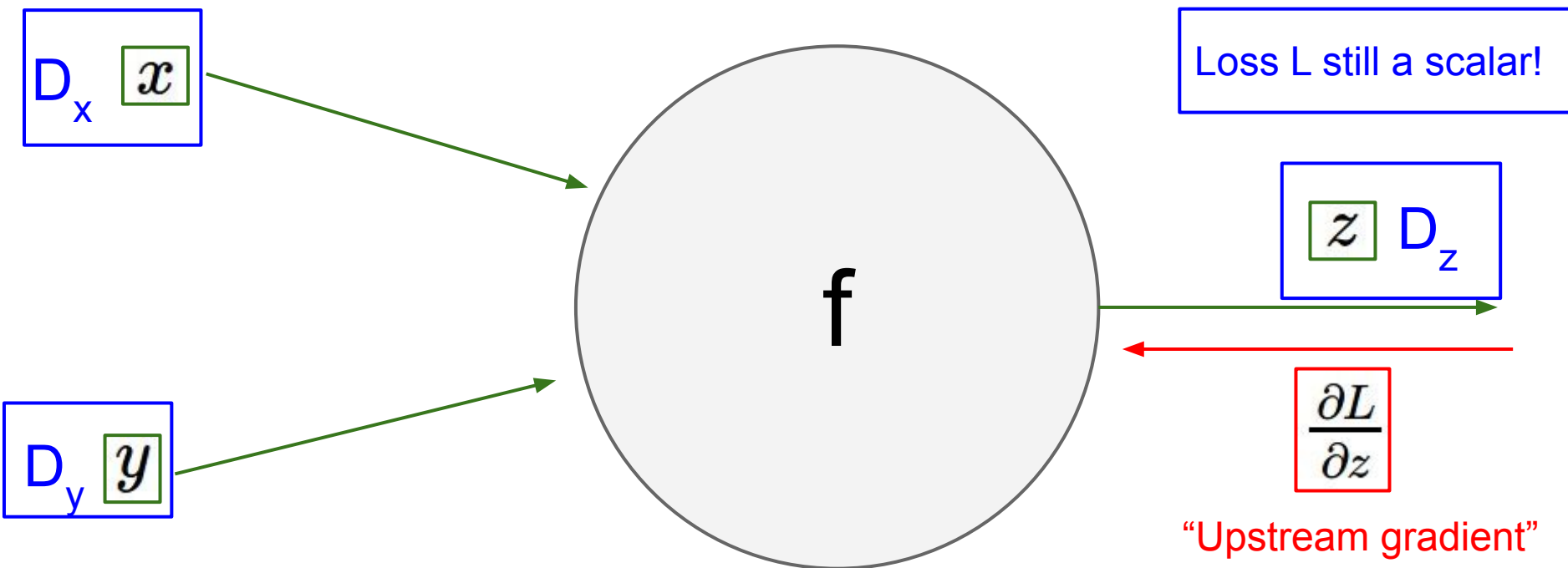
Backprop with Vectors



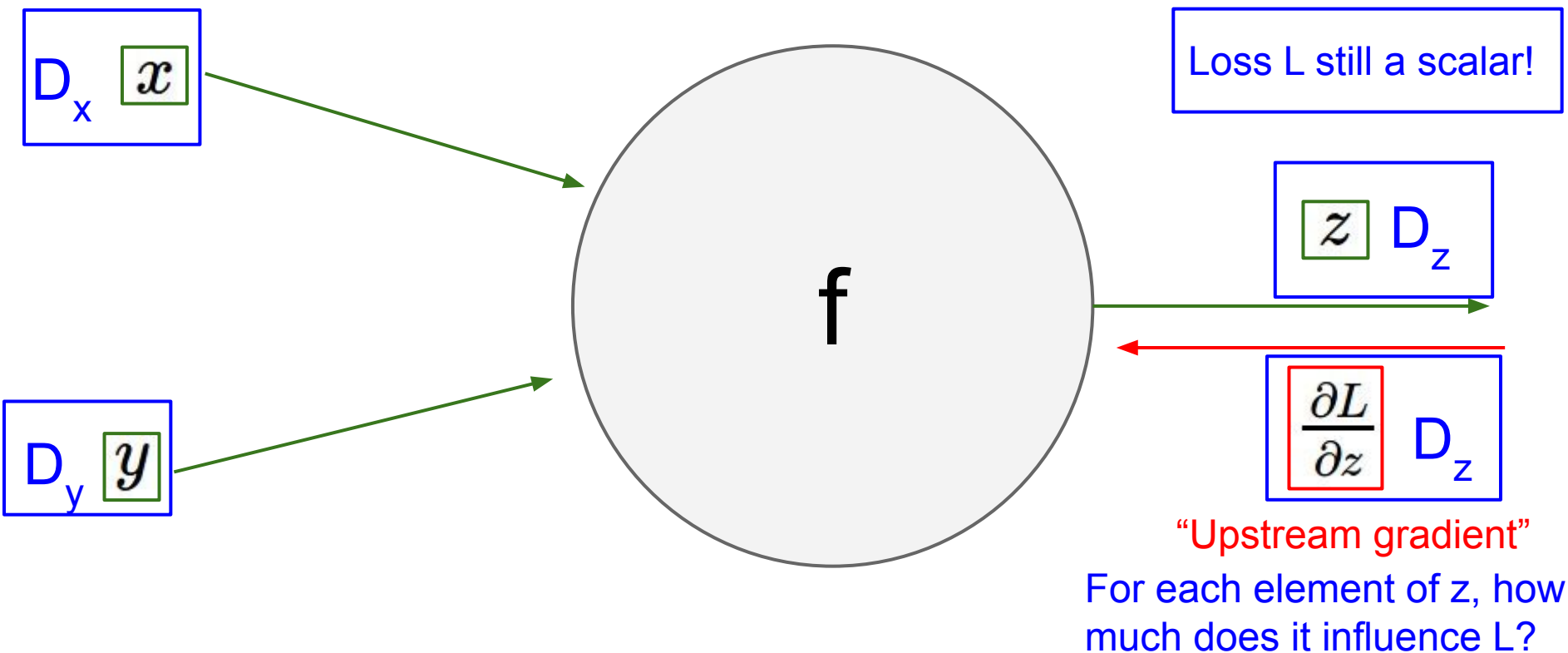
Backprop with Vectors



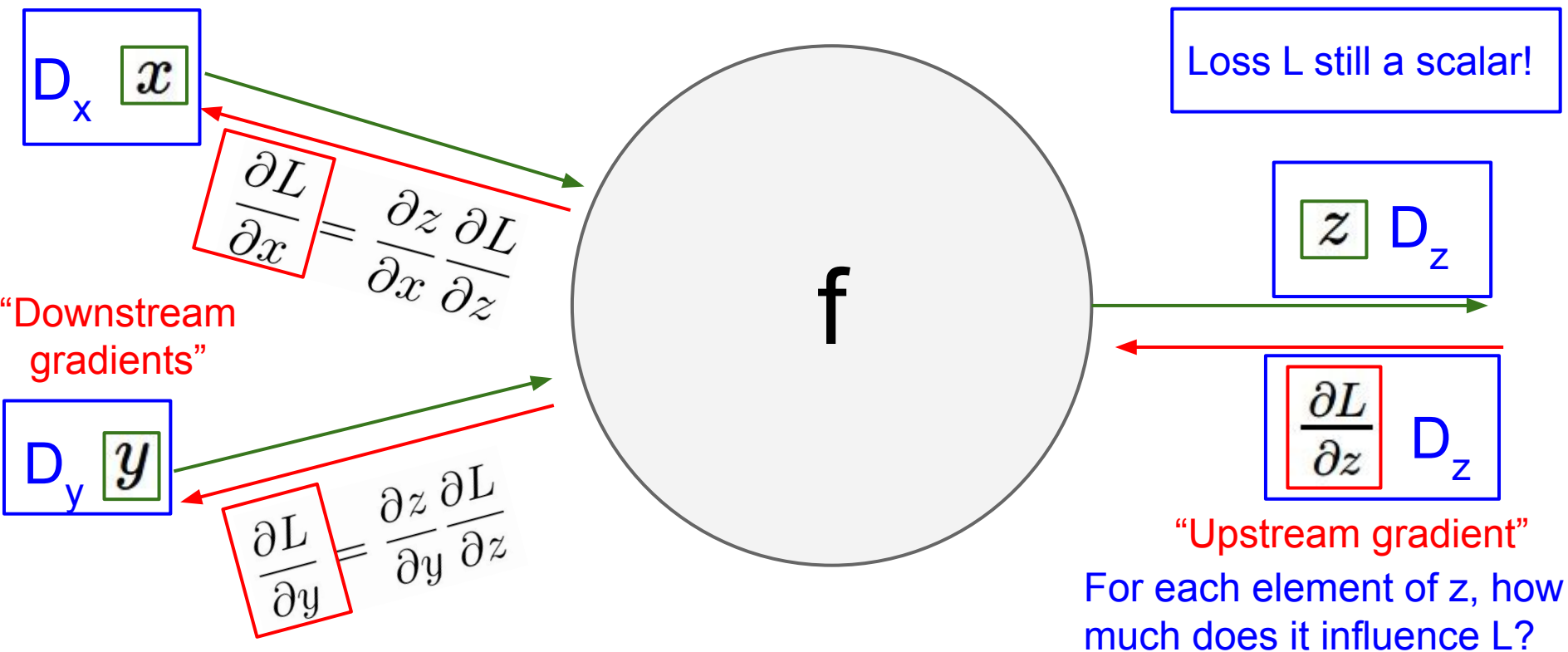
Backprop with Vectors



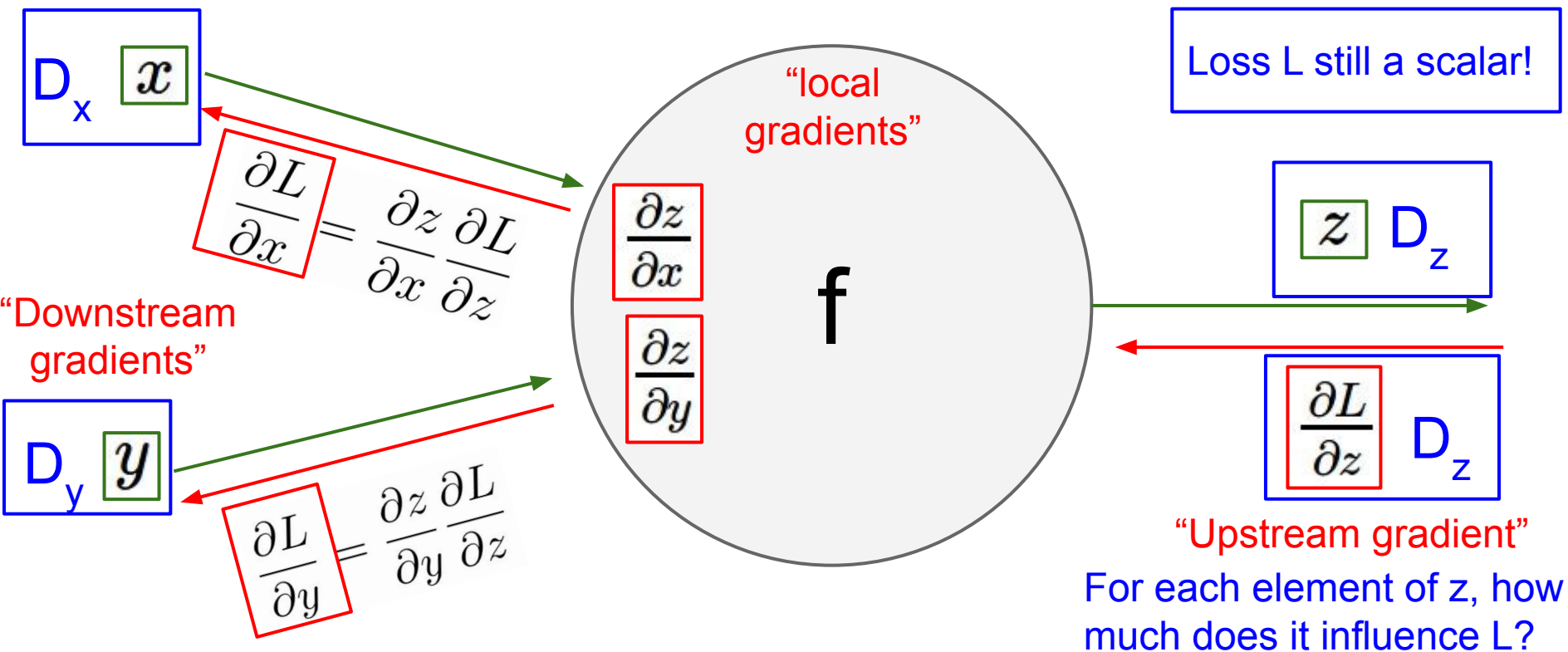
Backprop with Vectors



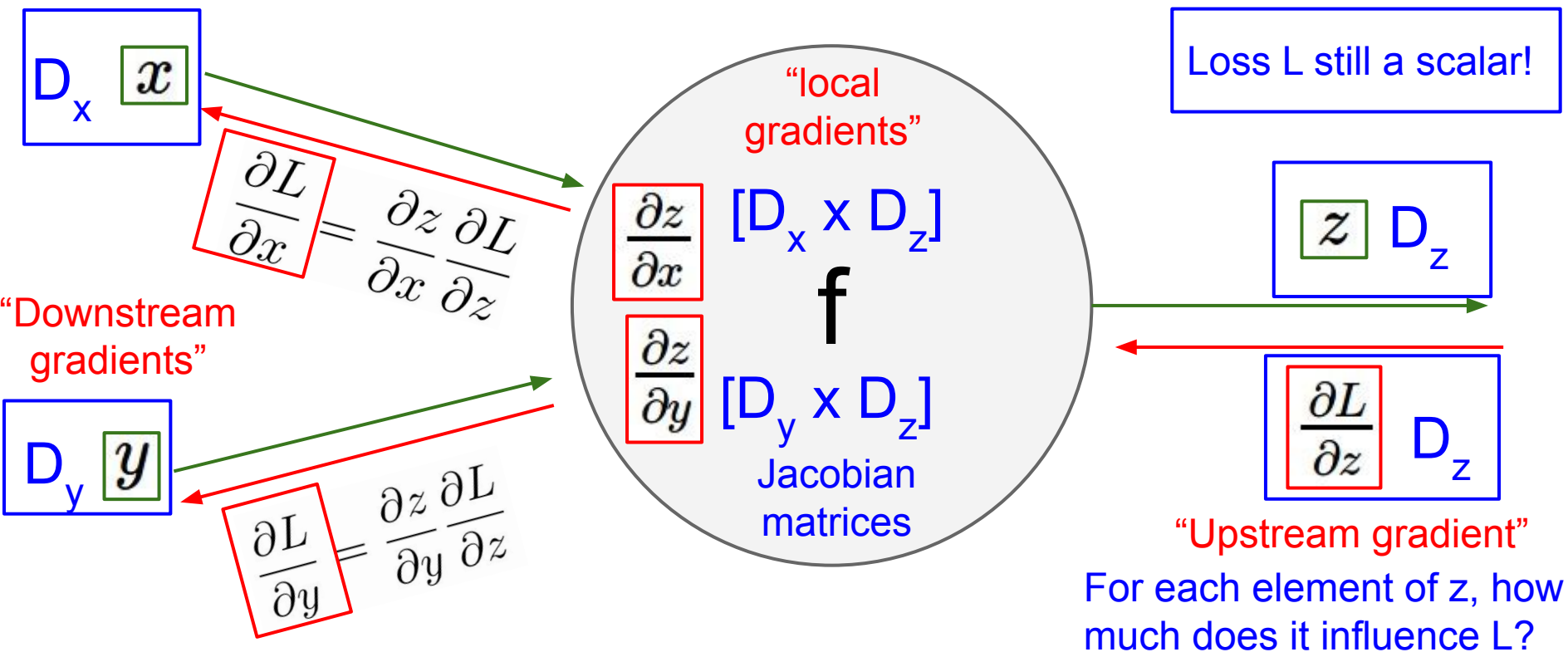
Backprop with Vectors



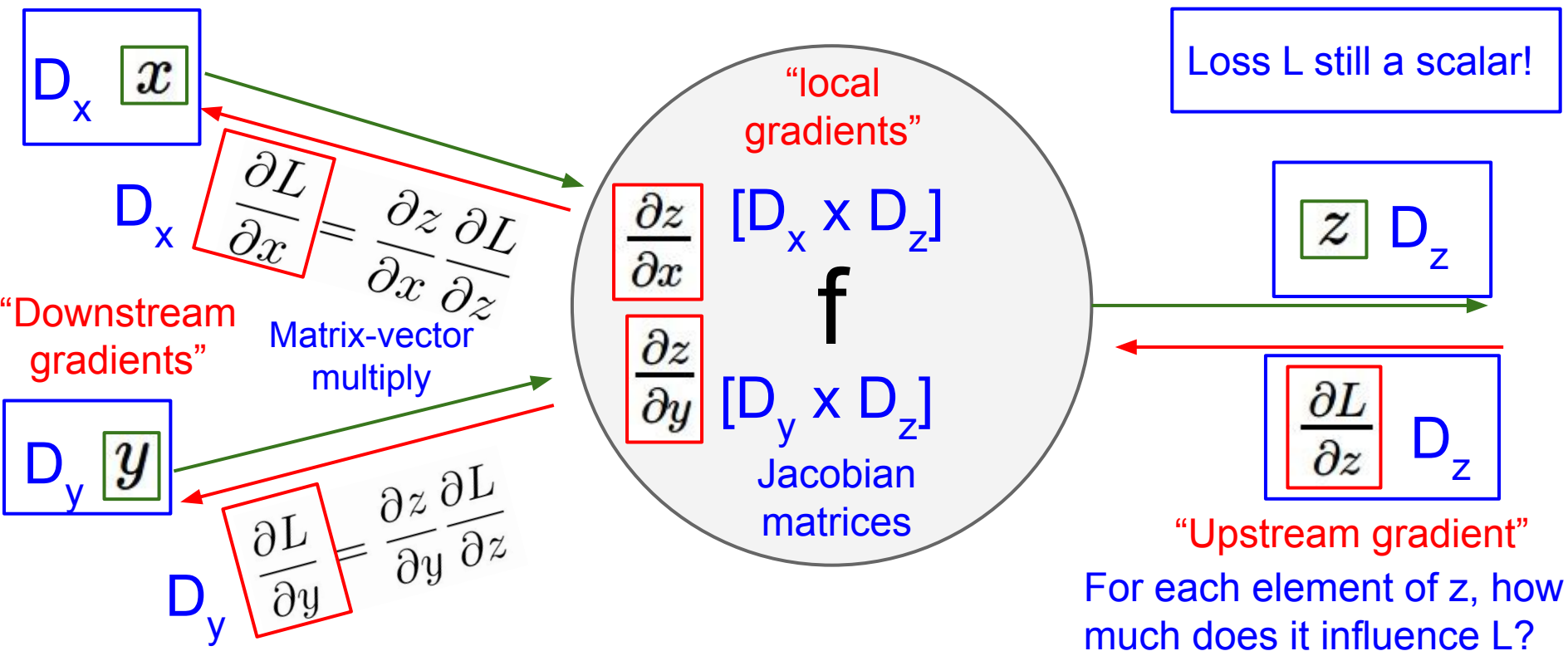
Backprop with Vectors



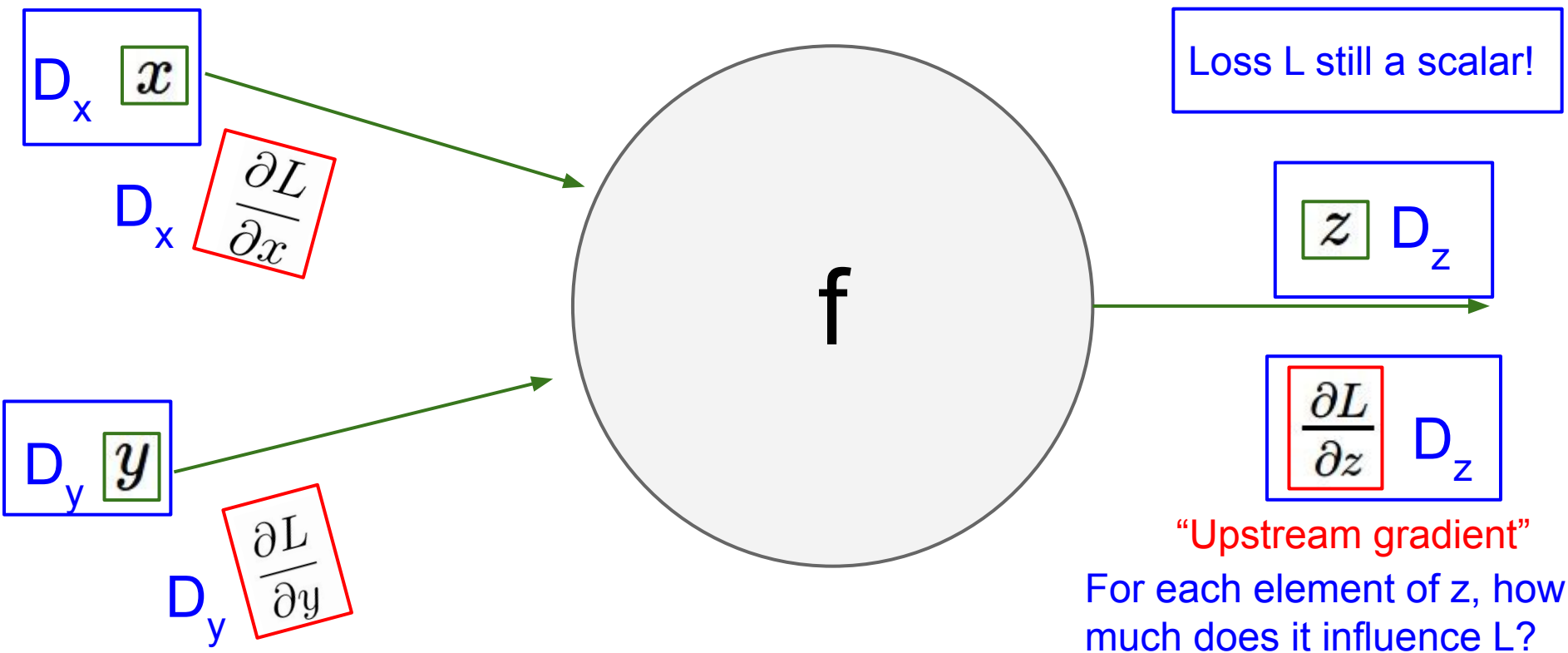
Backprop with Vectors



Backprop with Vectors



Gradients of variables wrt loss have same dims as the original variable



Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

$$f(x) = \max(0, x) \\ (\textit{elementwise})$$

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

$$f(x) = \max(0, x) \\ (\textit{elementwise})$$

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

4D dL/dz:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

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

(*elementwise*)

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

Jacobian dz/dx

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

4D dL/dz :

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

$$f(x) = \max(0, x) \\ (\text{elementwise})$$

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

$[dz/dx] [dL/dz]$

$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix}$

$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \end{bmatrix}$

$\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix}$

$\begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 9 \end{bmatrix}$

4D dL/dz:

$\begin{bmatrix} 4 \end{bmatrix}$

$\begin{bmatrix} -1 \end{bmatrix}$

$\begin{bmatrix} 5 \end{bmatrix}$

$\begin{bmatrix} 9 \end{bmatrix}$

Upstream
gradient

Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

$$f(x) = \max(0, x) \\ (\text{elementwise})$$

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

4D dL/dx:

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

$[dz/dx] [dL/dz]$

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

4D dL/dz:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

Backprop with Vectors

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

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

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

(*elementwise*)

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

4D dL/dx:

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

$[dz/dx] [dL/dz]$

$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

4D dL/dz:

$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

Backprop with Vectors

4D input x:

$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix}$

$$f(x) = \max(0, x) \quad (\text{elementwise})$$

4D output z:

$\begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$

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

4D dL/dx :

$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix}$

$[dz/dx] [dL/dz]$

$$\left(\frac{\partial L}{\partial x} \right)_i = \begin{cases} \left(\frac{\partial L}{\partial z} \right)_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

4D dL/dz :

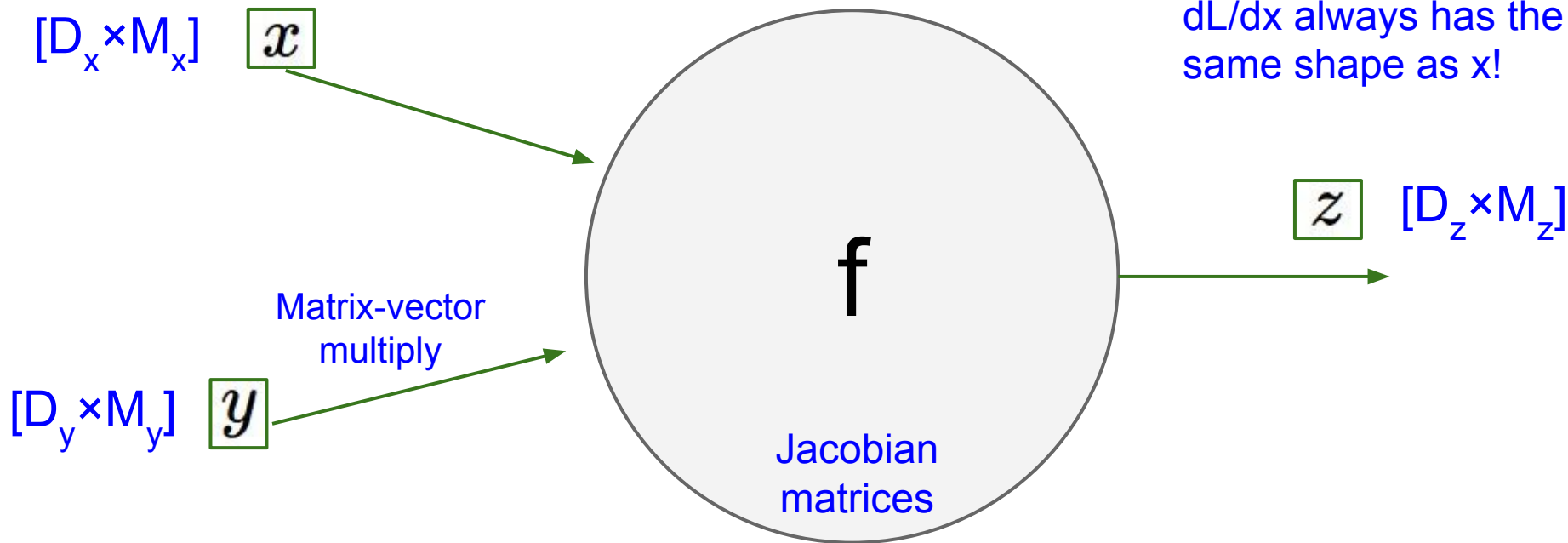
$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$

Upstream
gradient

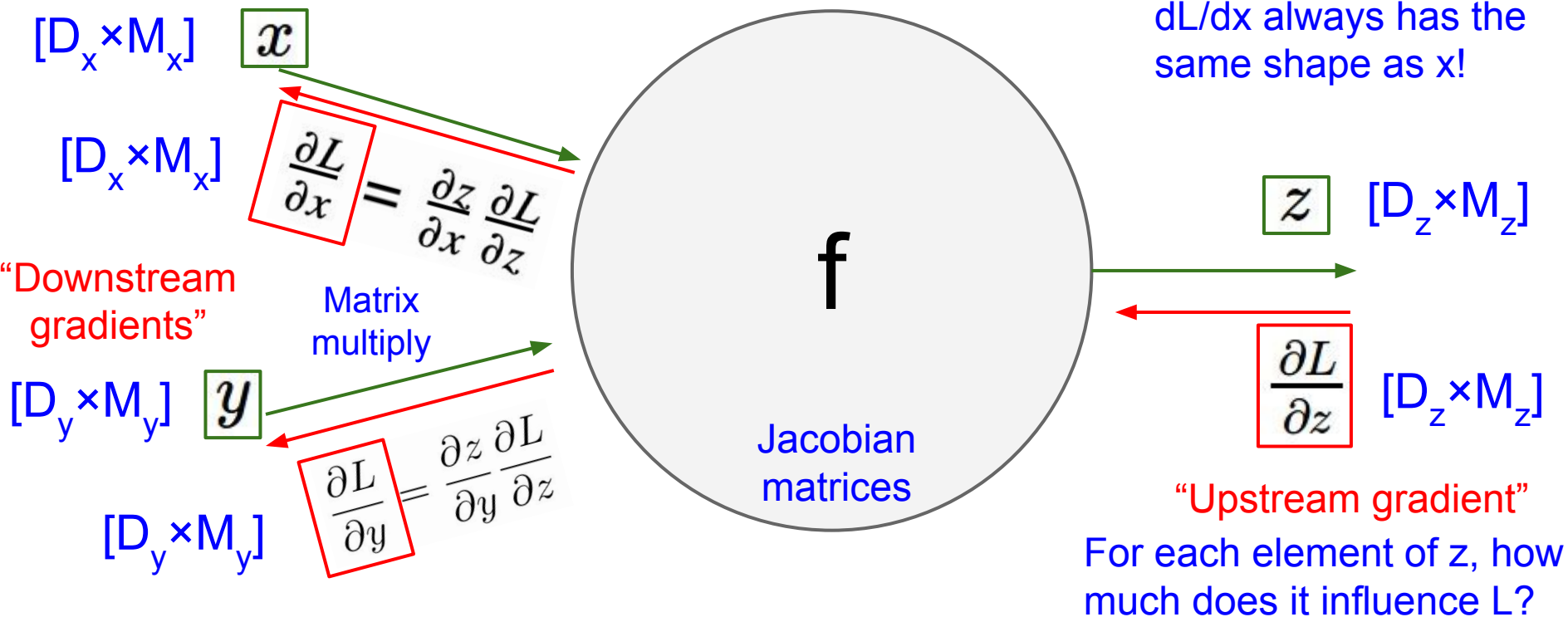
Backprop with Matrices (or Tensors)

Loss L still a scalar!

dL/dx always has the same shape as x !



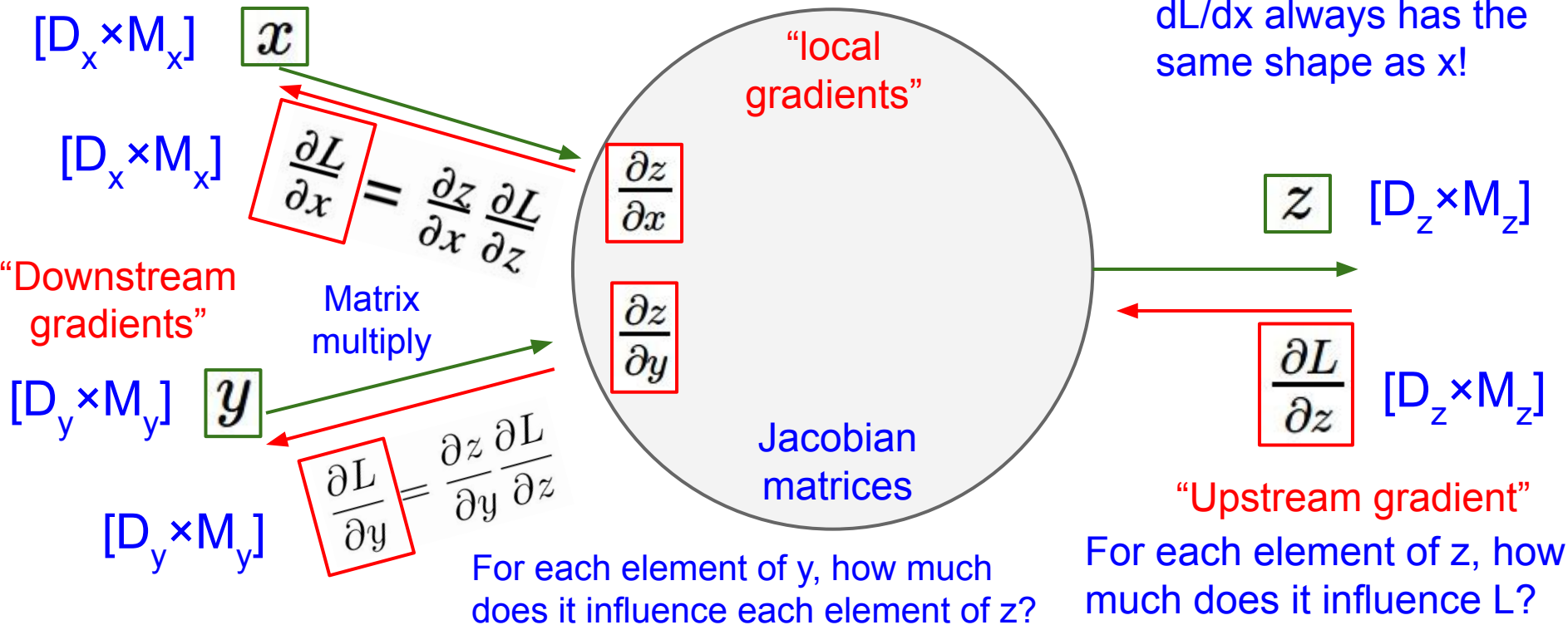
Backprop with Matrices (or Tensors)



Backprop with Matrices (or Tensors)

Loss L still a scalar!

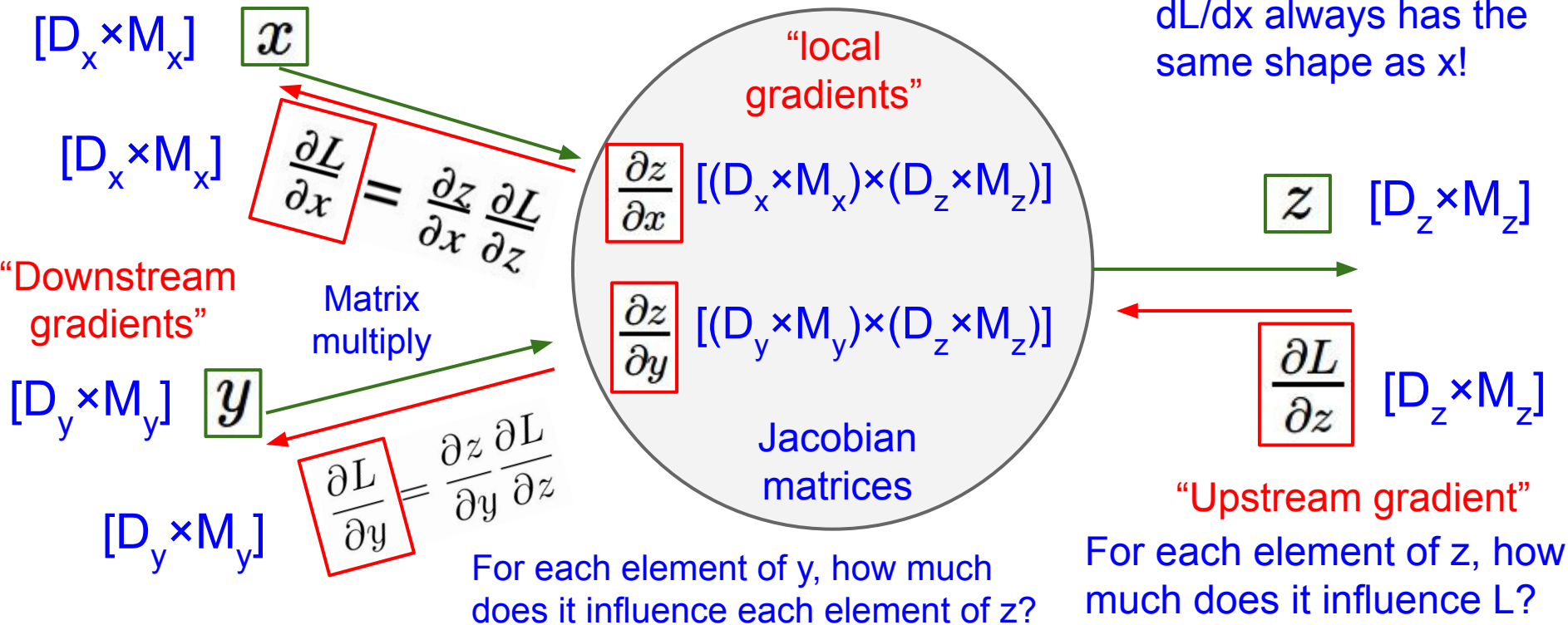
dL/dx always has the same shape as x !



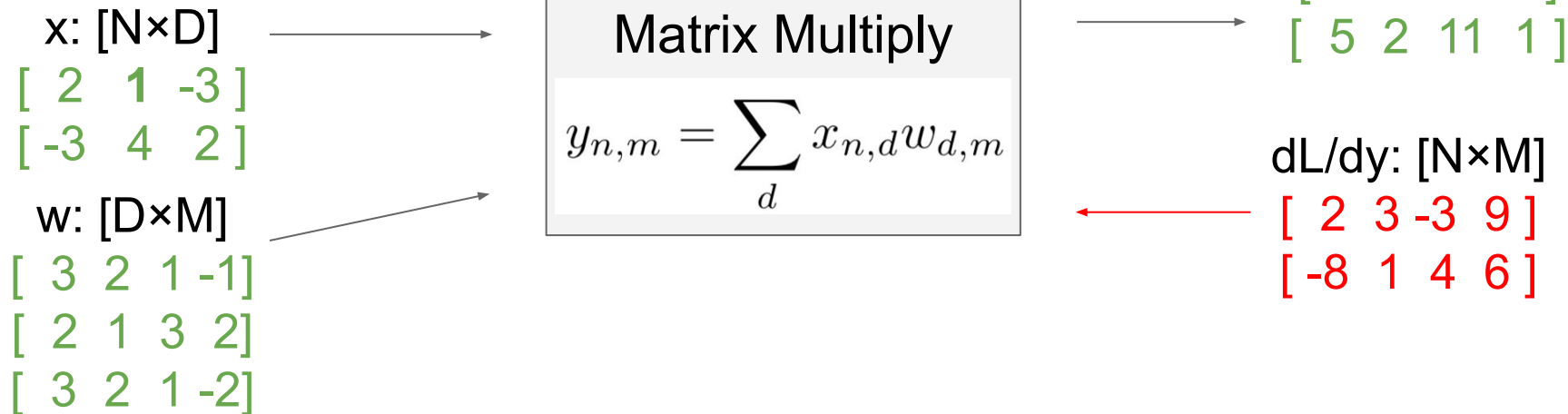
Backprop with Matrices (or Tensors)

Loss L still a scalar!

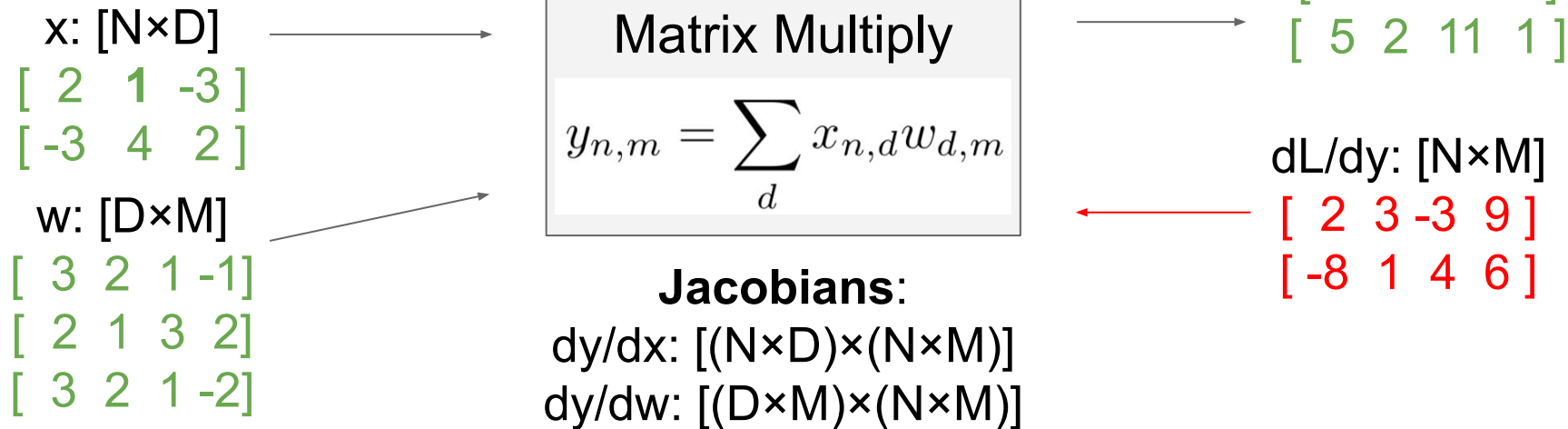
dL/dx always has the same shape as x !



Backprop with Matrices

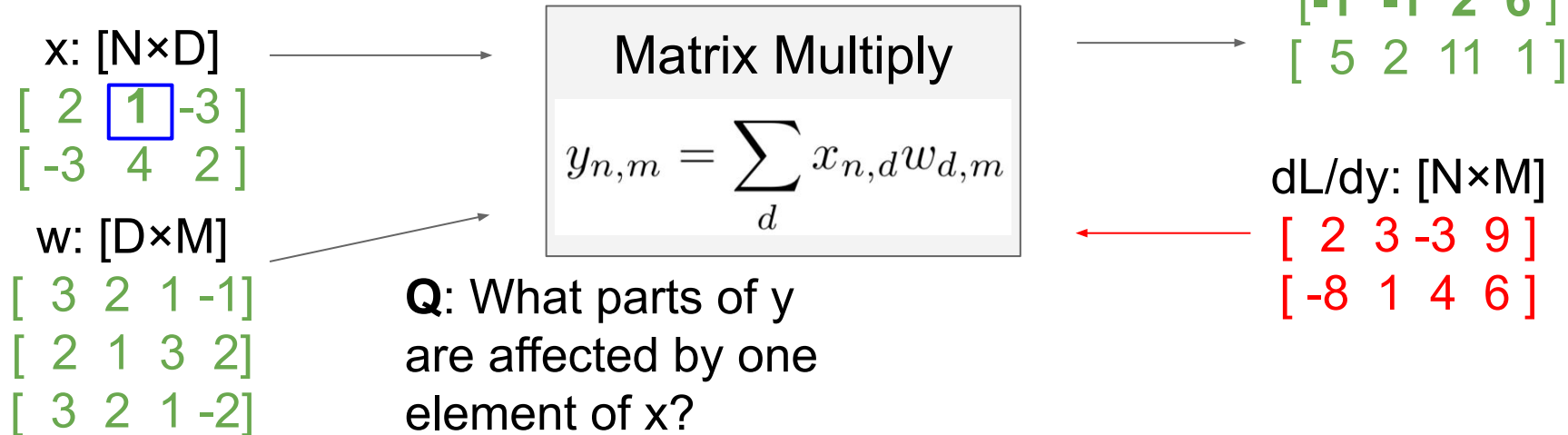


Backprop with Matrices



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



Backprop with Matrices

$$x: [N \times D]$$
$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$$w: [D \times M]$$
$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

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

$$y: [N \times M]$$
$$\begin{bmatrix} \boxed{-1} & \boxed{-1} & \boxed{2} & \boxed{6} \\ 5 & 2 & 11 & 1 \end{bmatrix}$$

$$dL/dy: [N \times M]$$
$$\begin{bmatrix} \boxed{2} & \boxed{3} & \boxed{-3} & \boxed{9} \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

Q: What parts of y are affected by one element of x ?

A: $x_{n,d}$ affects the whole row $y_{n,\cdot}$.

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

Backprop with Matrices

$$x: [N \times D]$$

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$$w: [D \times M]$$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

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

$$y: [N \times M]$$

$$\begin{bmatrix} -1 & -1 & \boxed{2} & 6 \\ 5 & 2 & 11 & 1 \end{bmatrix}$$

$$dL/dy: [N \times M]$$

$$\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

Q: What parts of y are affected by one element of x ?

A: $x_{n,d}$ affects the whole row $y_{n,\cdot}$.

Q: How much does $x_{n,d}$ affect $y_{n,m}$?

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

Backprop with Matrices

$$x: [N \times D]$$

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$$w: [D \times M]$$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & \boxed{3} & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

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

$$y: [N \times M]$$

$$\begin{bmatrix} -1 & -1 & \boxed{2} & 6 \\ 5 & 2 & 11 & 1 \end{bmatrix}$$

$$dL/dy: [N \times M]$$

$$\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

Q: What parts of y are affected by one element of x ?

A: $x_{n,d}$ affects the whole row $y_{n,\cdot}$.

Q: How much does $x_{n,d}$ affect $y_{n,m}$?

A: $w_{d,m}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

Backprop with Matrices

$$x: [N \times D]$$

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$$w: [D \times M]$$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & \boxed{3} & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

$[N \times D] \quad [N \times M] \quad [M \times D]$

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

Matrix Multiply

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

$$y: [N \times M]$$

$$\begin{bmatrix} -1 & -1 & \boxed{2} & 6 \\ 5 & 2 & 11 & 1 \end{bmatrix}$$

$$dL/dy: [N \times M]$$

$$\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

Q: What parts of y are affected by one element of x ?

A: $x_{n,d}$ affects the whole row $y_{n,\cdot}$.

Q: How much does $x_{n,d}$ affect $y_{n,m}$?

A: $w_{d,m}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

Backprop with Matrices

$$x: [N \times D]$$

$$\begin{bmatrix} 2 & \boxed{1} & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$$w: [D \times M]$$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & \boxed{3} & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

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

$$y: [N \times M]$$

$$\begin{bmatrix} -1 & -1 & \boxed{2} & 6 \\ 5 & 2 & 11 & 1 \end{bmatrix}$$

$$dL/dy: [N \times M]$$

$$\begin{bmatrix} \boxed{2} & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

By similar logic:

$$[N \times D] \quad [N \times M] \quad [M \times D]$$

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

$$[D \times M] \quad [D \times N] \quad [N \times M]$$

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

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

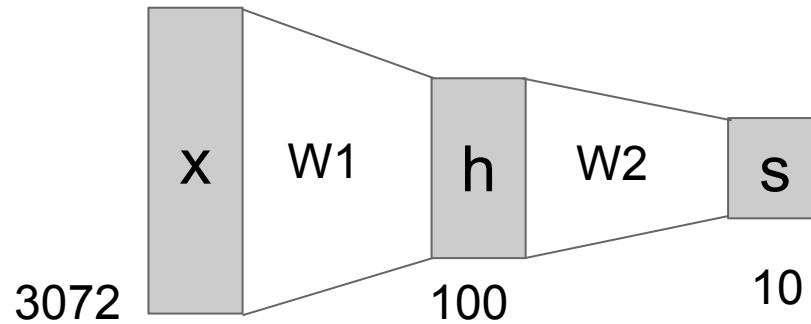
Wrapping up: Neural Networks

Linear score function:

$$f = Wx$$

2-layer Neural Network

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



Next: Convolutional Neural Networks

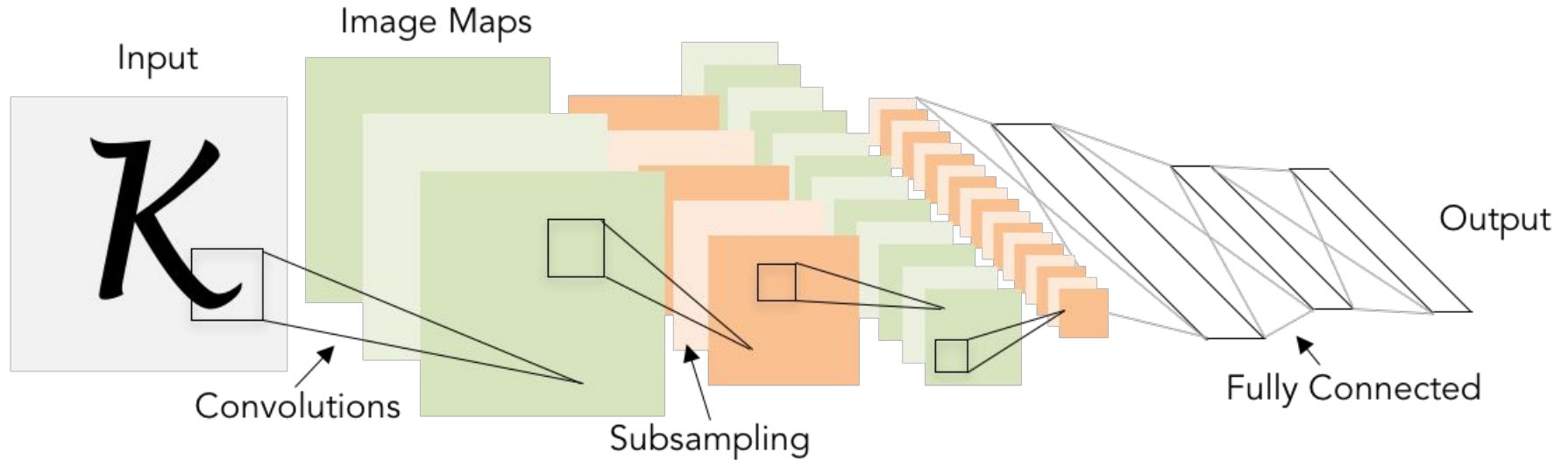
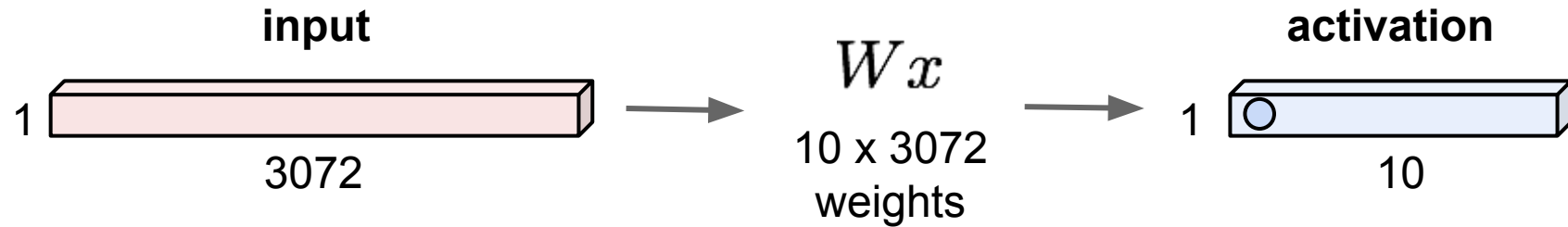


Illustration of LeCun et al. 1998

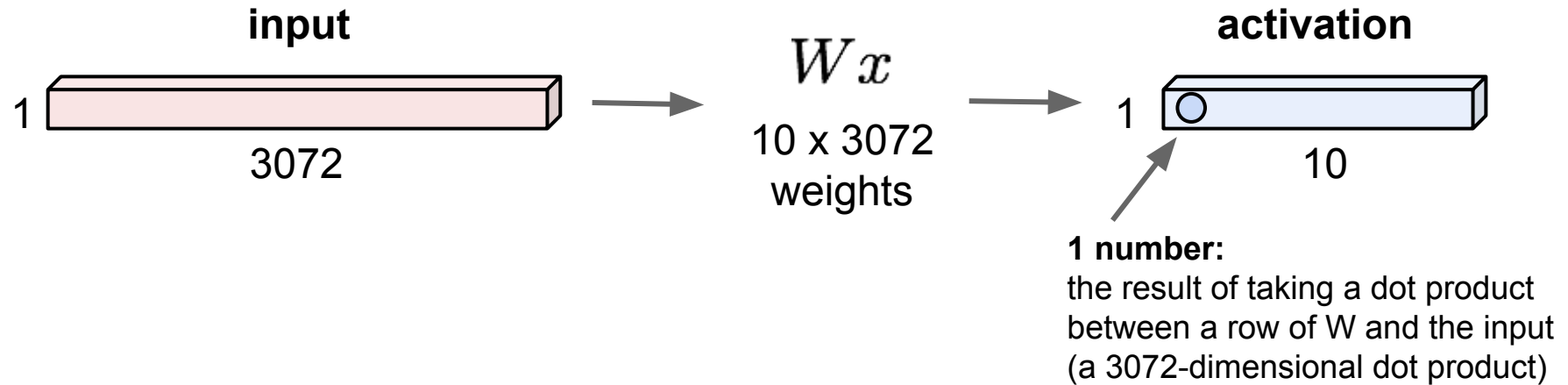
Recap: Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



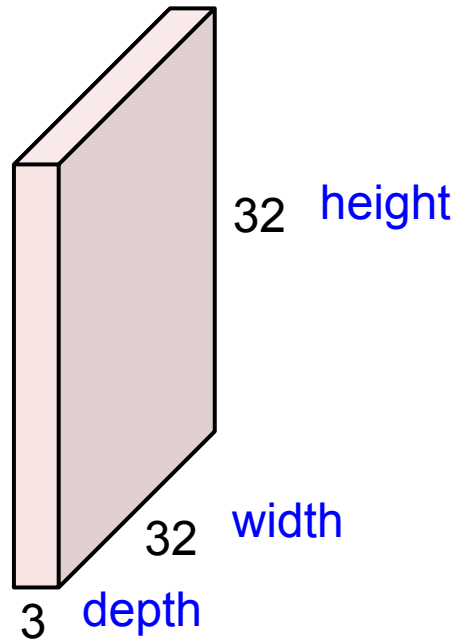
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1



Convolution Layer

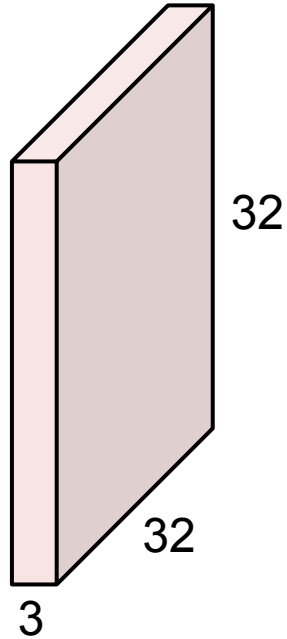
32x32x3 image $\rightarrow$ preserve spatial structure



Main idea: only look at small patches of an image

Convolution Layer

32x32x3 image



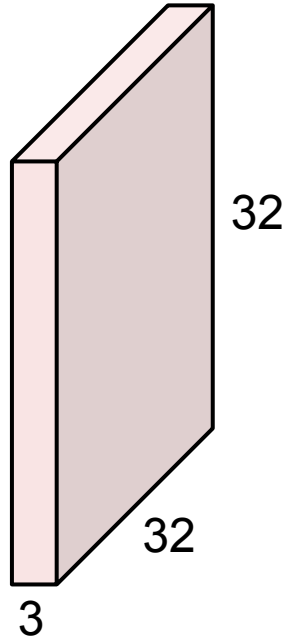
5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

32x32x3 image



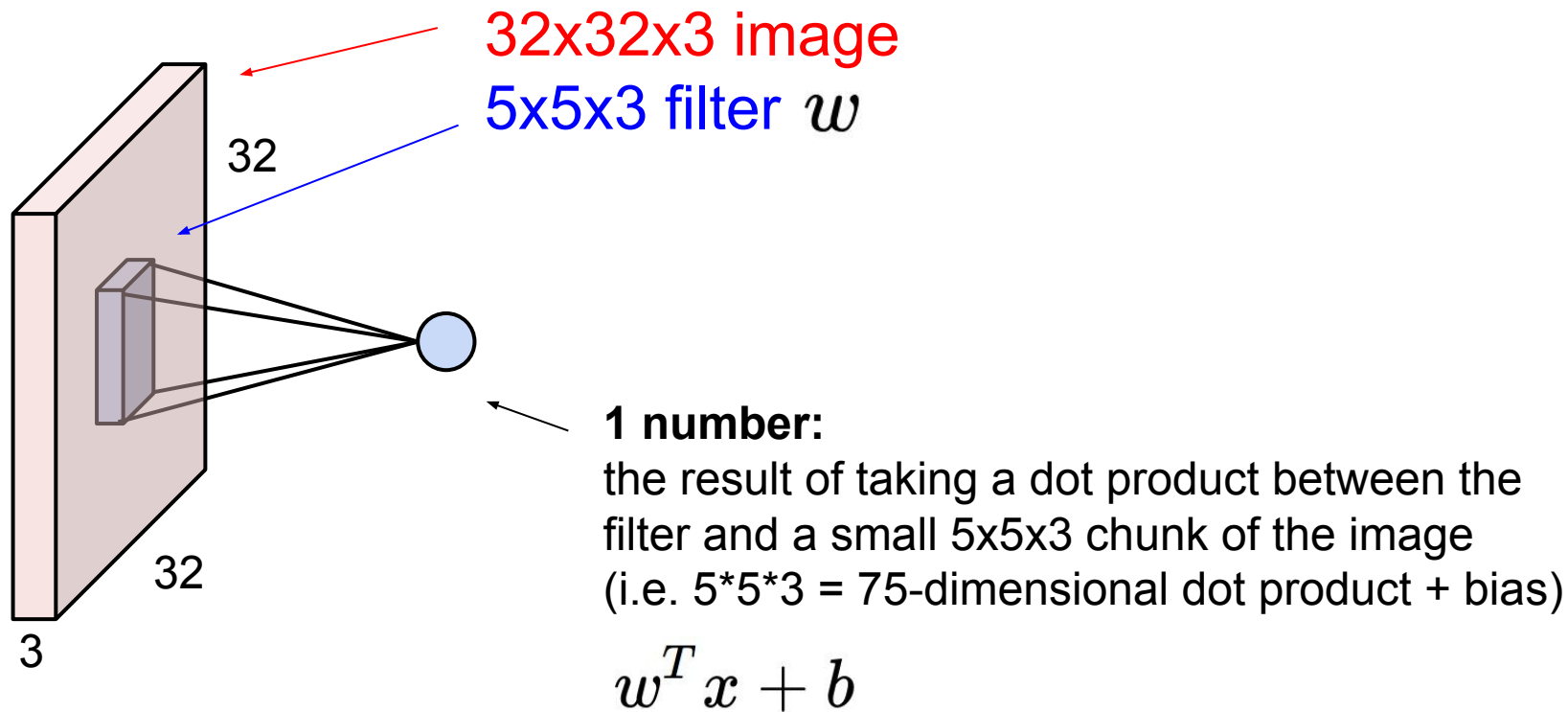
Filters always extend the full depth of the input volume

5x5x3 filter

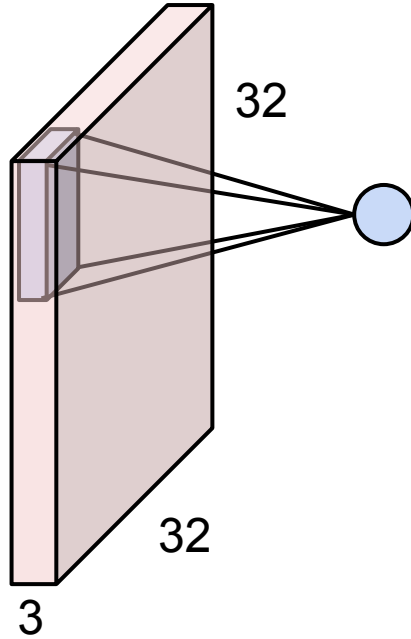


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

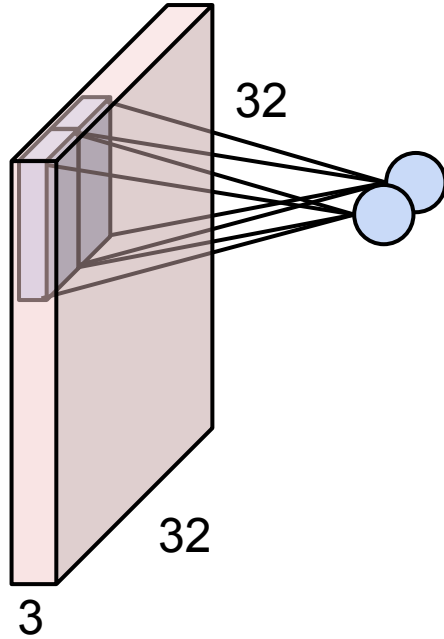
Convolution Layer



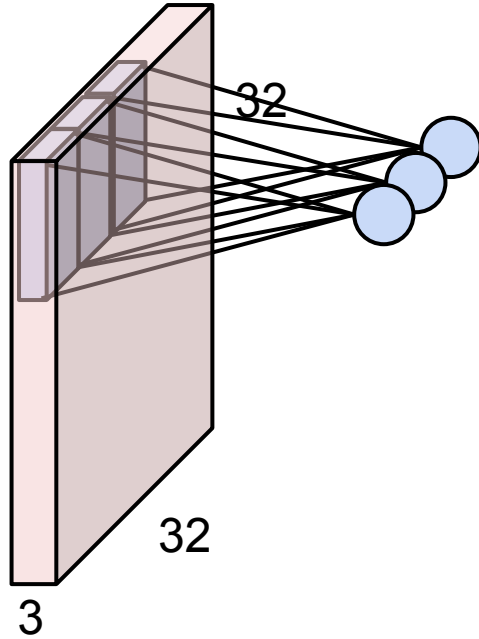
Convolution Layer



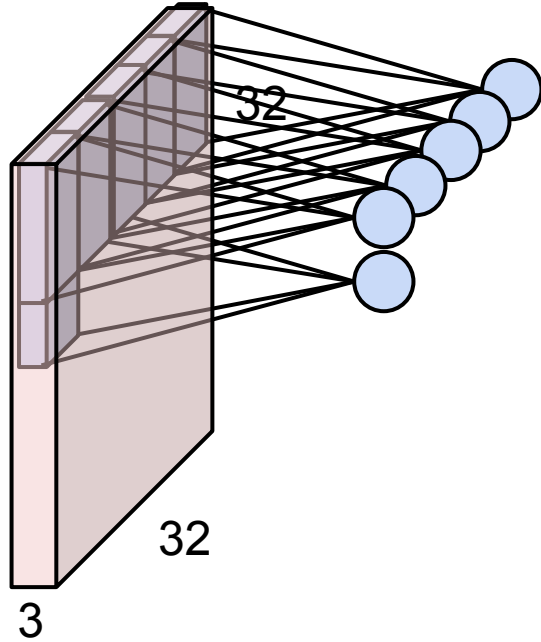
Convolution Layer



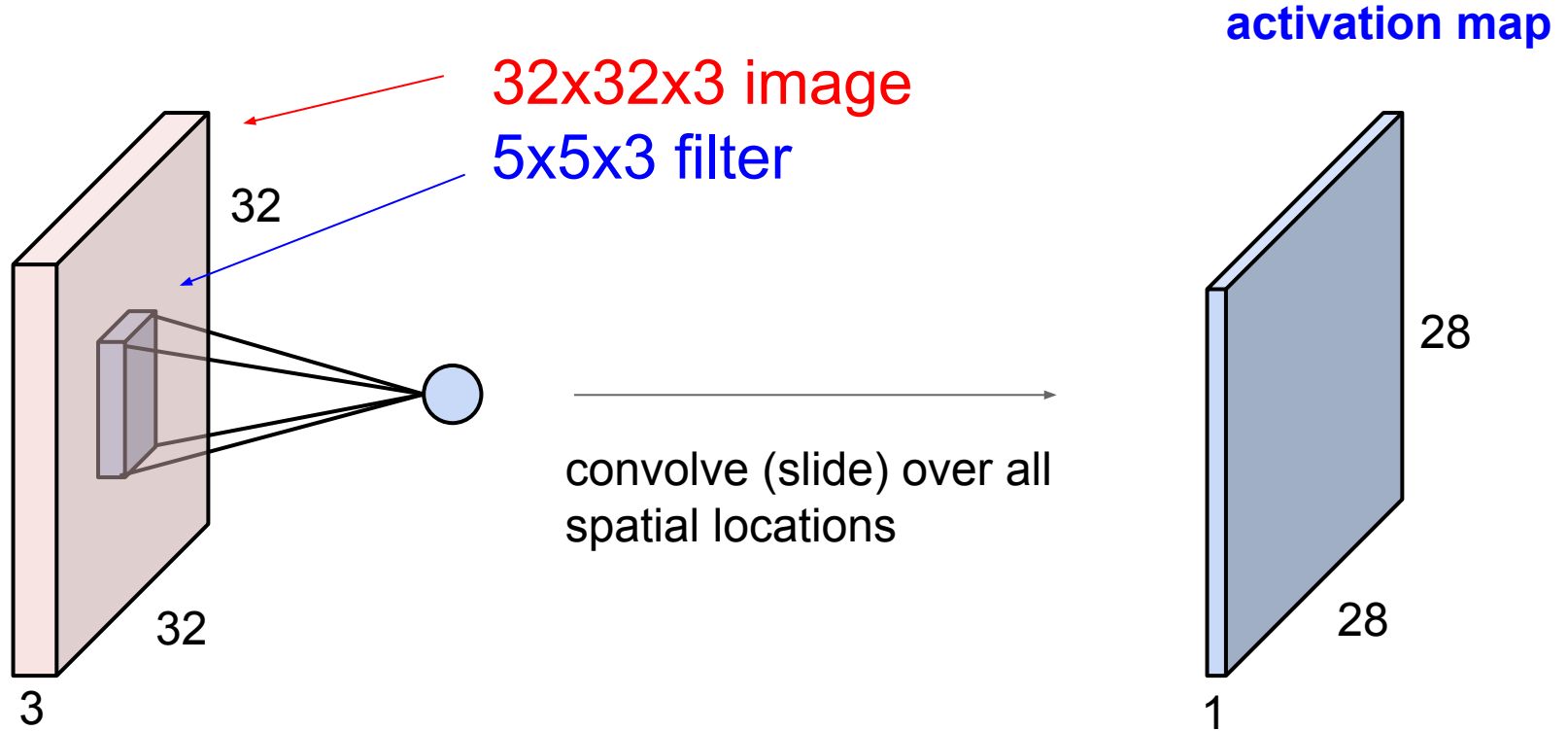
Convolution Layer



Convolution Layer



Convolution Layer

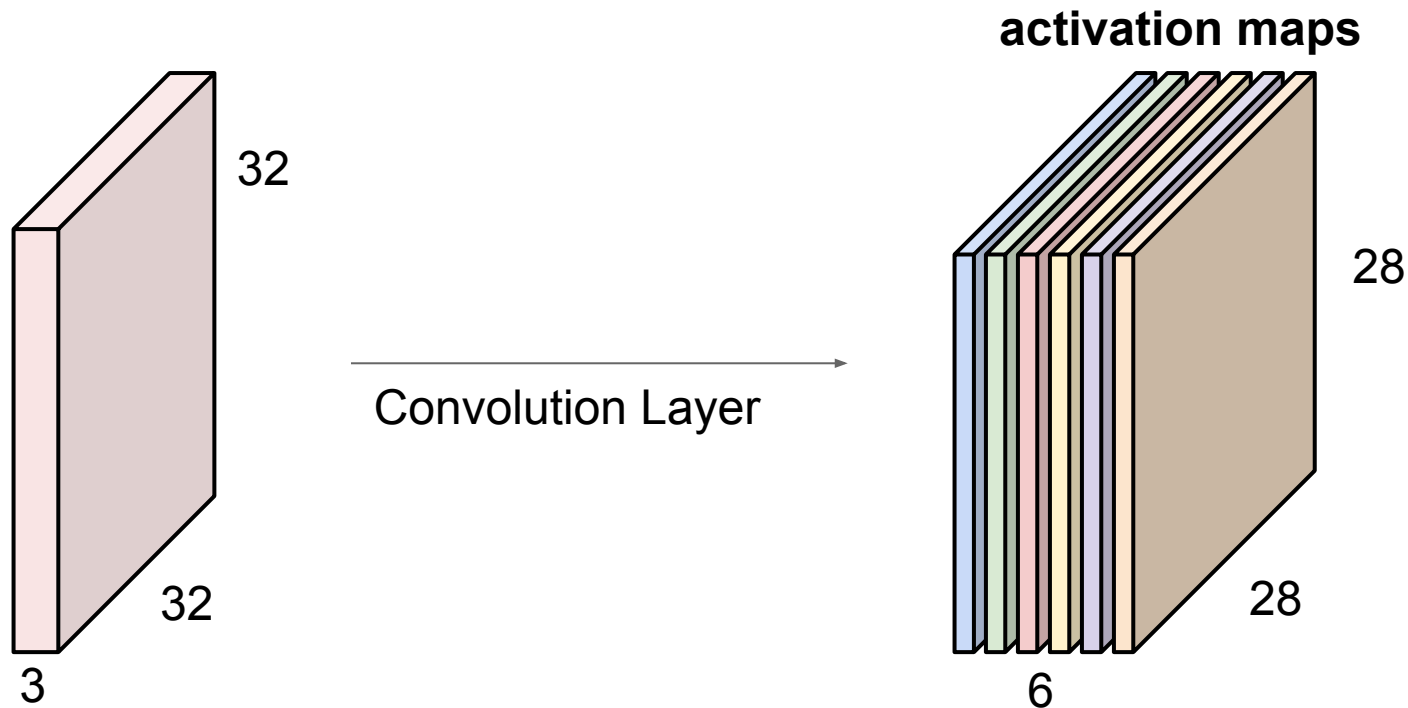


Convolution Layer

consider a second, **green** filter

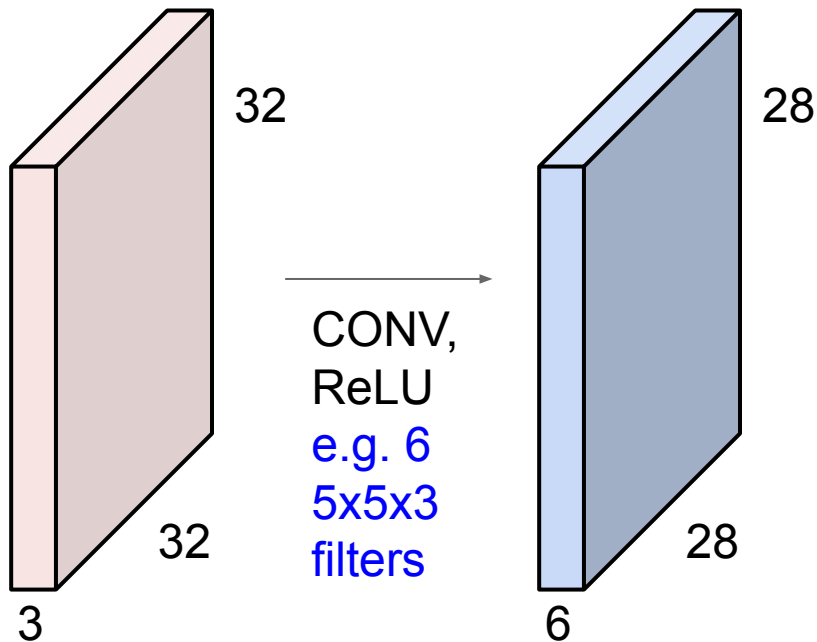


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

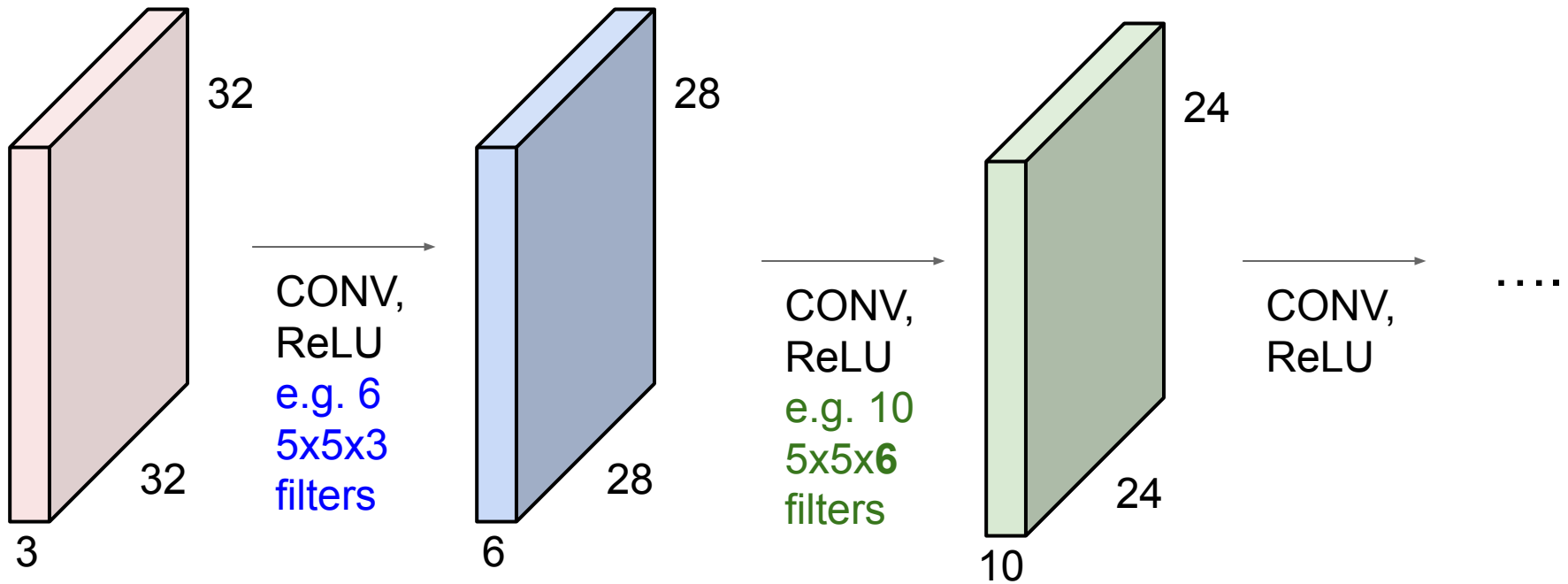


We stack these up to get a “new image” of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



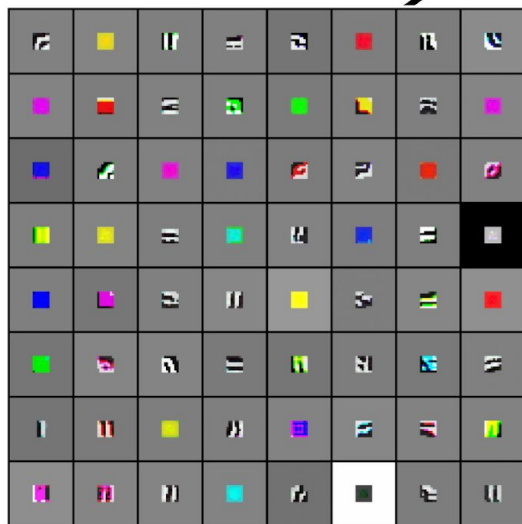
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



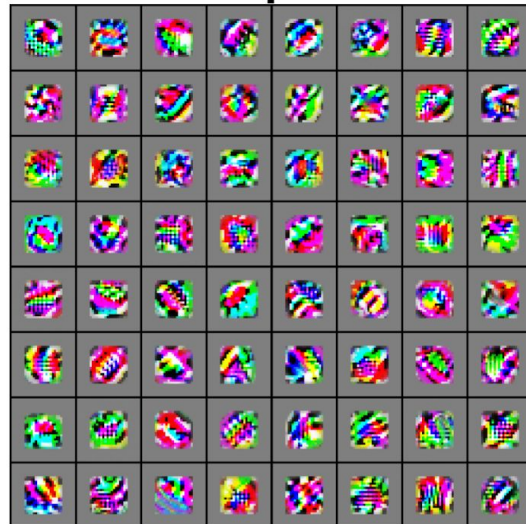
Preview

[Zeiler and Fergus 2013]

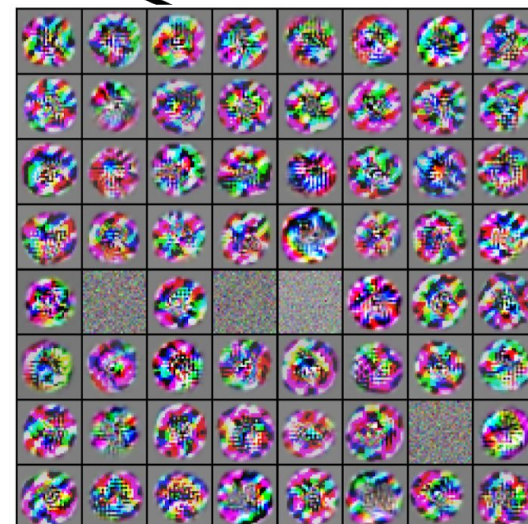
Visualization of VGG-16 by Lane McIntosh. VGG-16 architecture from [Simonyan and Zisserman 2014].



VGG-16 Conv1_1

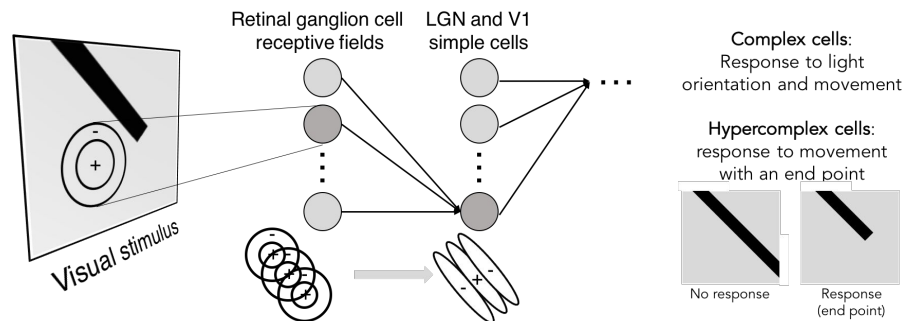
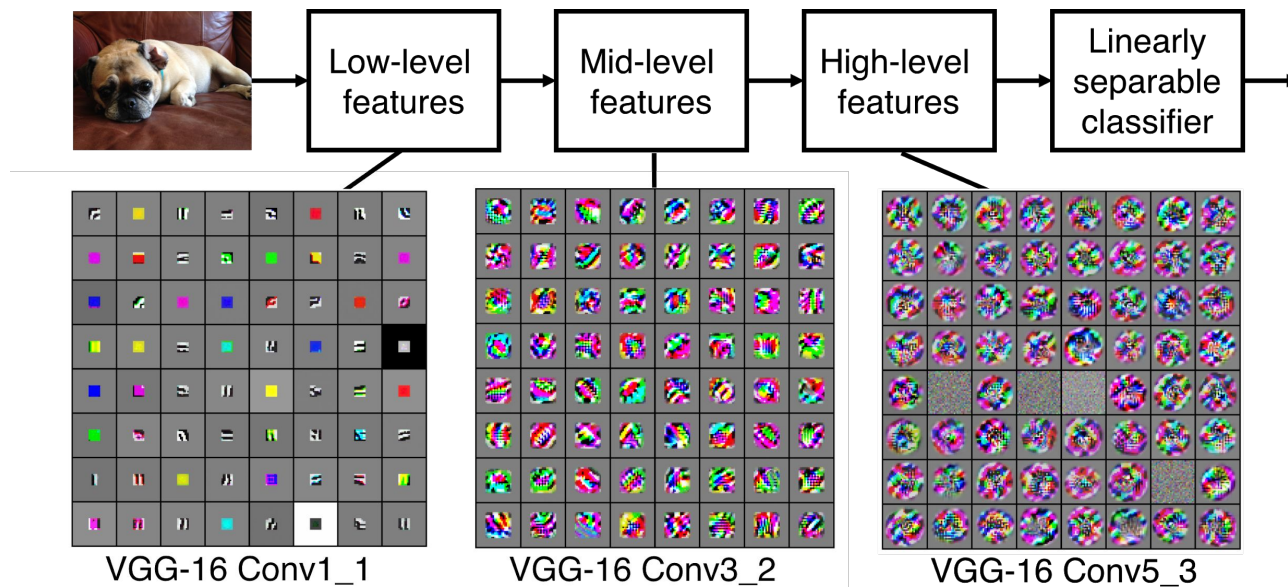


VGG-16 Conv3_2



VGG-16 Conv5_3

Preview





one filter =>
one activation map

example 5x5 filters
(32 total)

Activations:

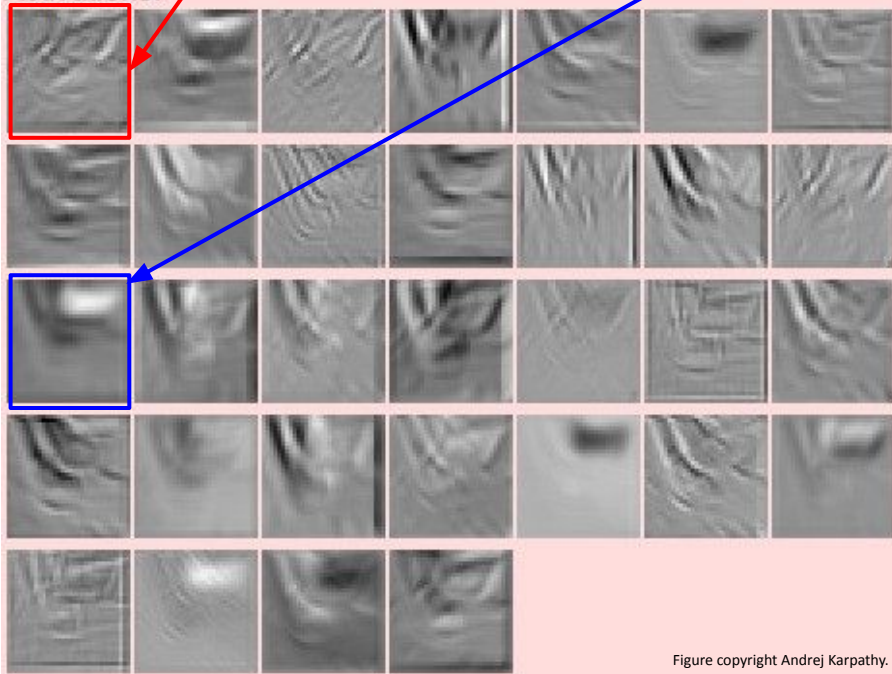


Figure copyright Andrej Karpathy.

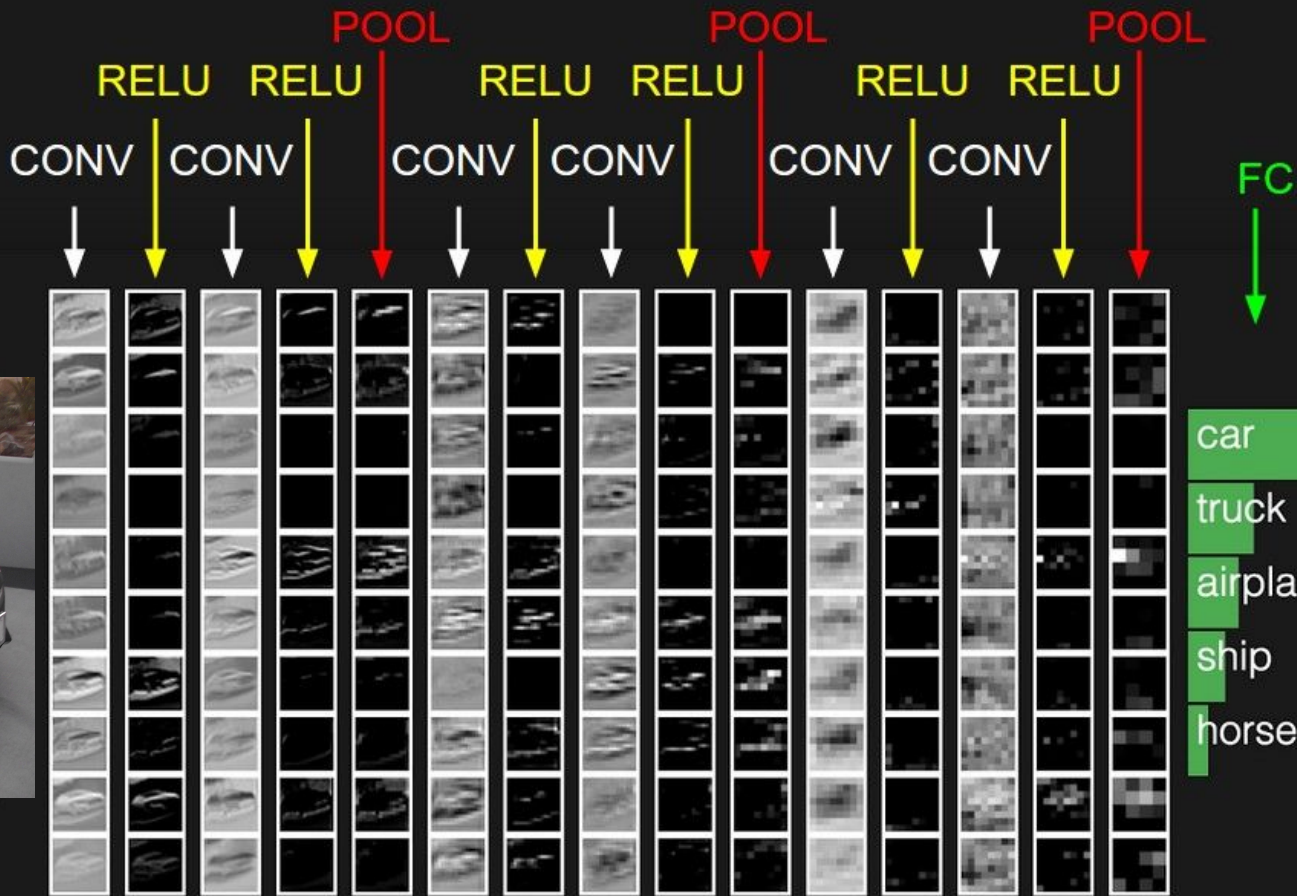
We call the layer convolutional
because it is related to convolution
of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1,y-n_2]$$

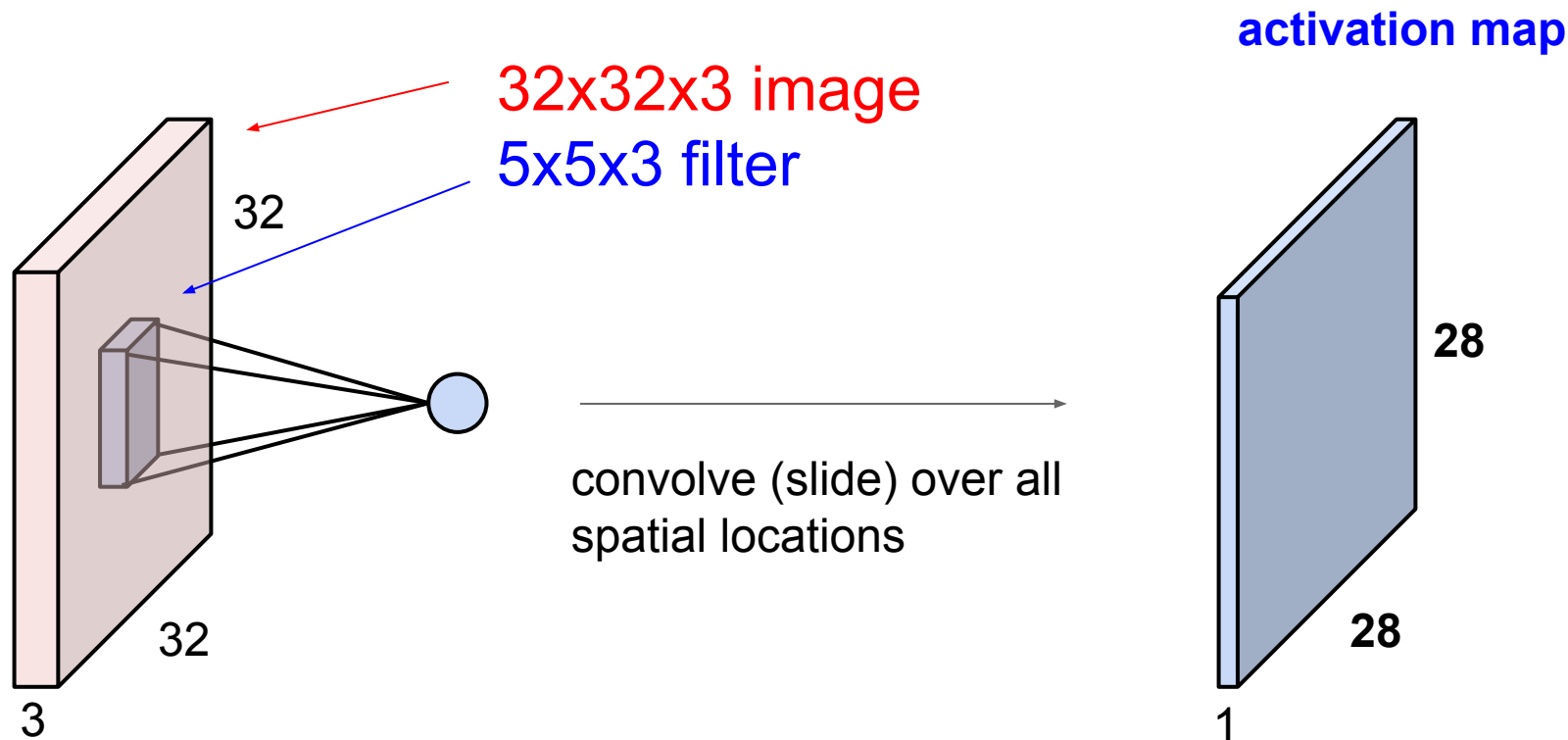


elementwise multiplication and sum of
a filter and the signal (image)

preview:

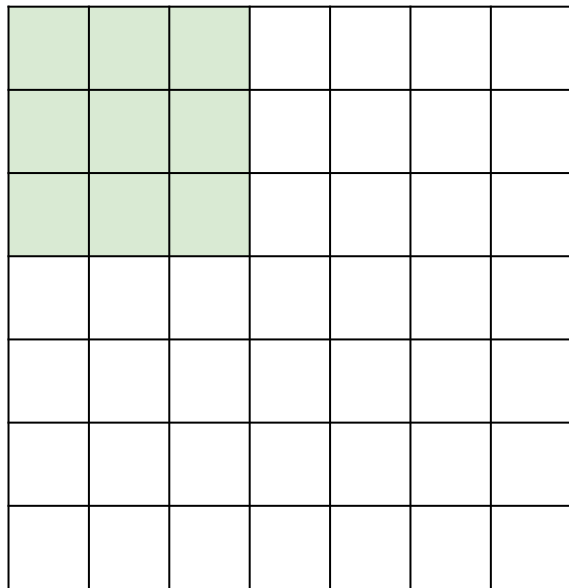


A closer look at spatial dimensions:



A closer look at spatial dimensions:

7

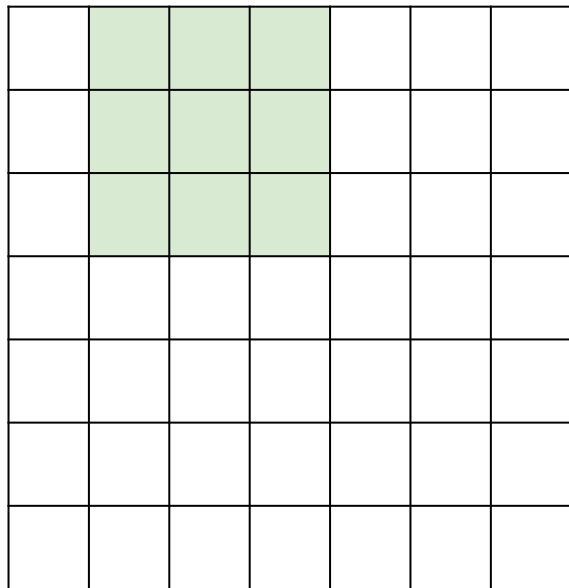


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7

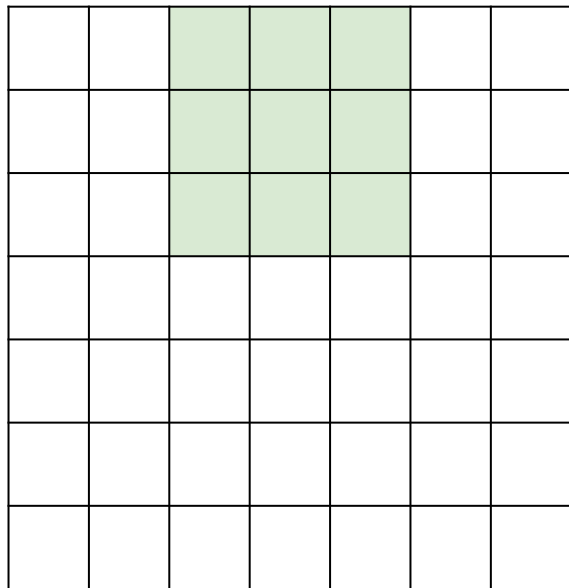


7

7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:

7

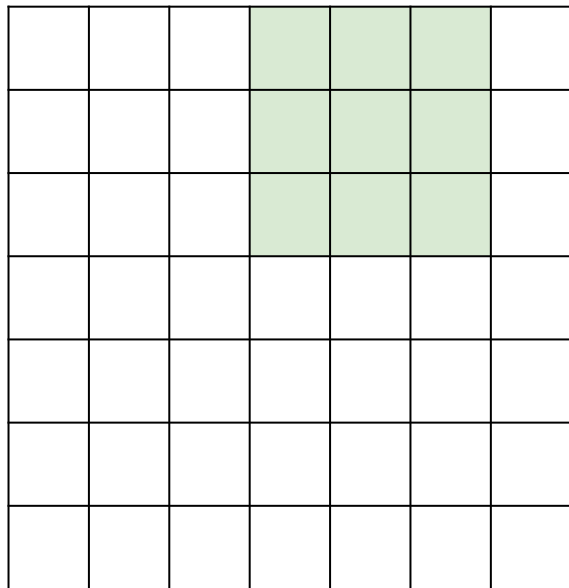


7x7 input (spatially)
assume 3x3 filter

7

A closer look at spatial dimensions:

7



7

7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:

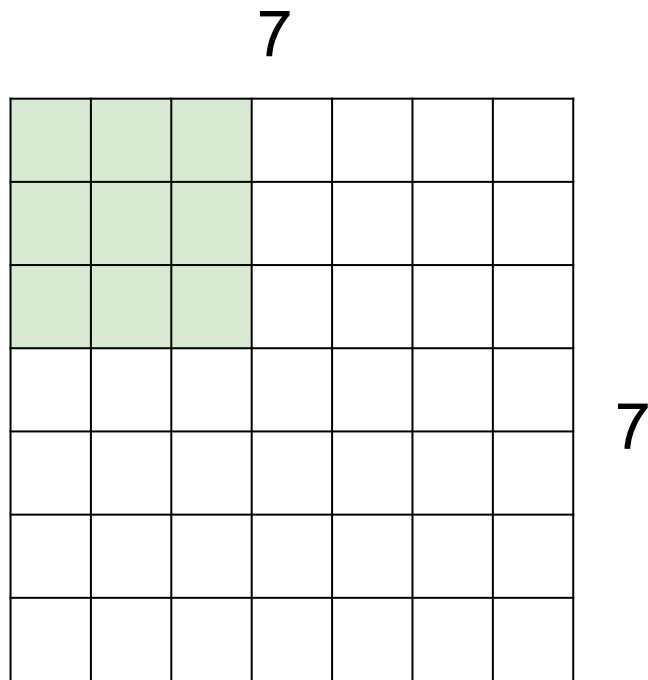
7

7

7x7 input (spatially)
assume 3x3 filter

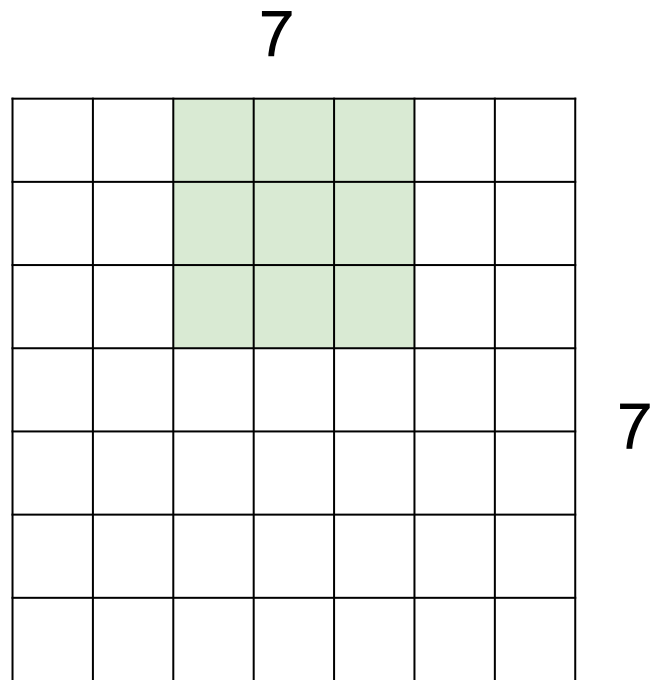
=> 5x5 output

A closer look at spatial dimensions:



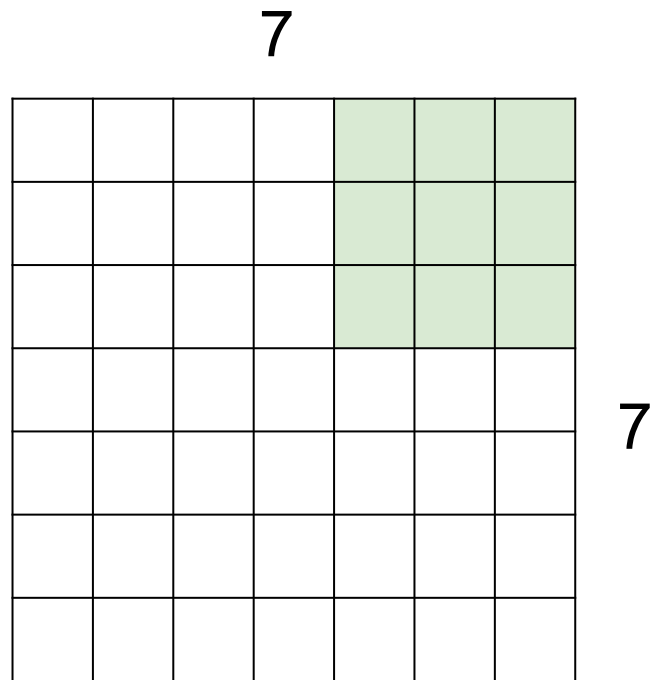
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



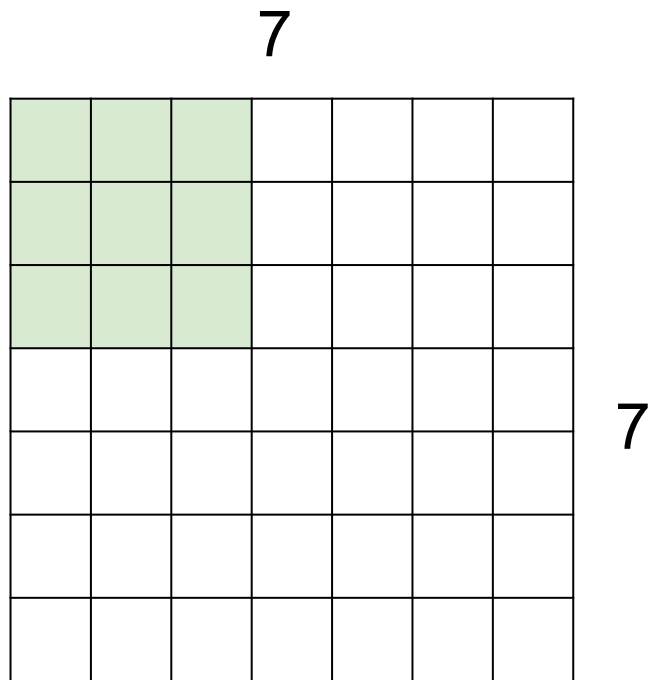
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



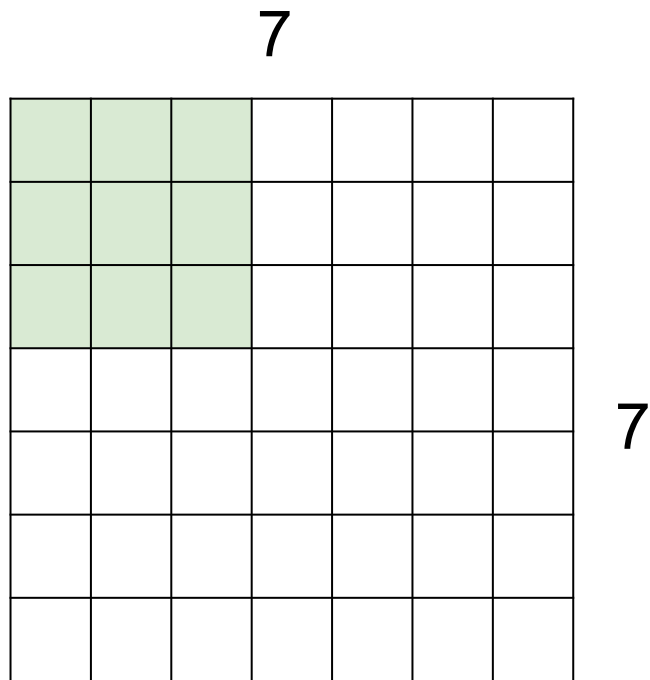
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



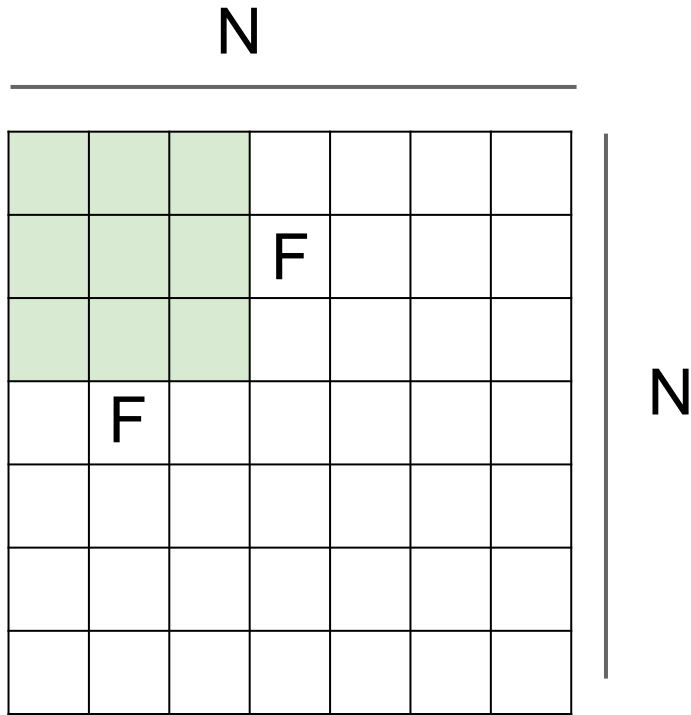
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
 $(N - F) / \text{stride} + 1$

e.g. $N = 7, F = 3$:

stride 1 $\Rightarrow (7 - 3) / 1 + 1 = 5$

stride 2 $\Rightarrow (7 - 3) / 2 + 1 = 3$

stride 3 $\Rightarrow (7 - 3) / 3 + 1 = 2.33 \therefore \backslash$

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output?

(recall:)

$$(N - F) / \text{stride} + 1$$

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output?

7x7 output!

(recall:)

$$(N + 2P - F) / \text{stride} + 1$$

In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

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 $F \times F$, and zero-padding with $(F-1)/2$. (will preserve size spatially)

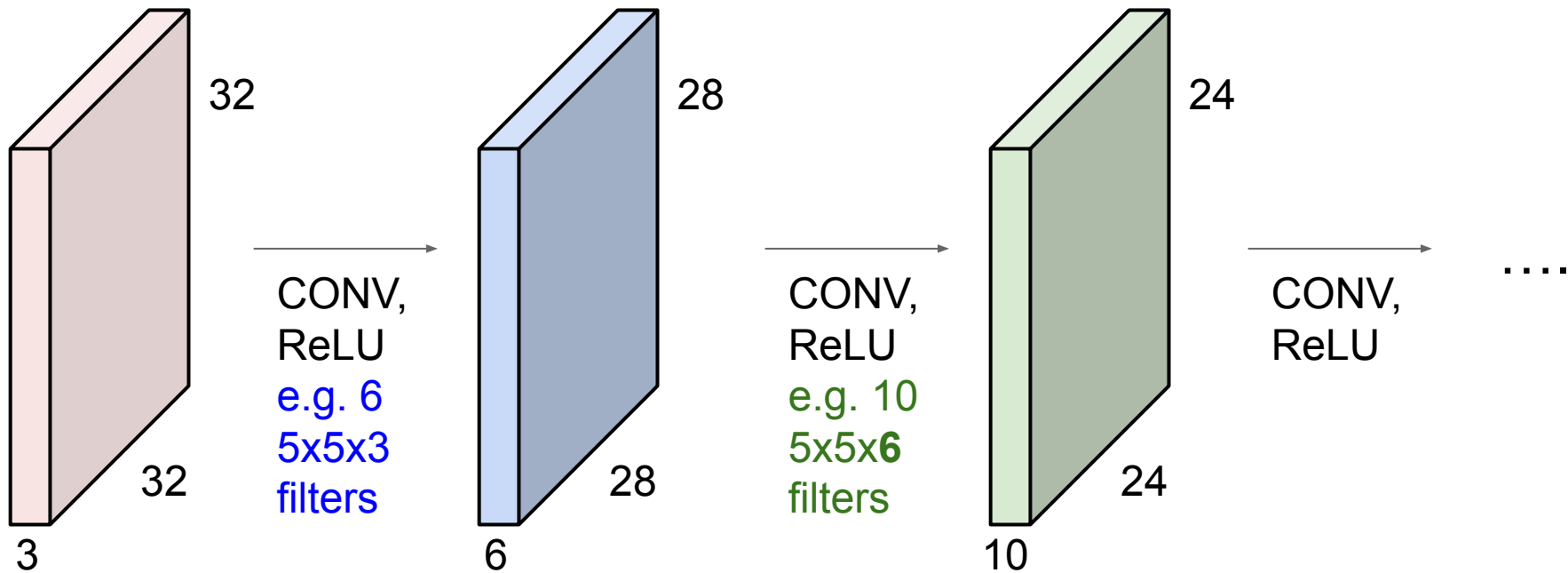
e.g. $F = 3 \Rightarrow$ zero pad with 1

$F = 5 \Rightarrow$ zero pad with 2

$F = 7 \Rightarrow$ zero pad with 3

Remember back to...

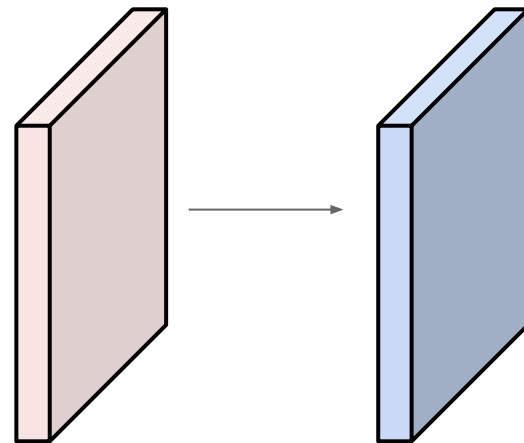
E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!
(32 -> 28 -> 24 ...).



Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2



Let's assume output size is $H \times W \times D$.
What is D ?

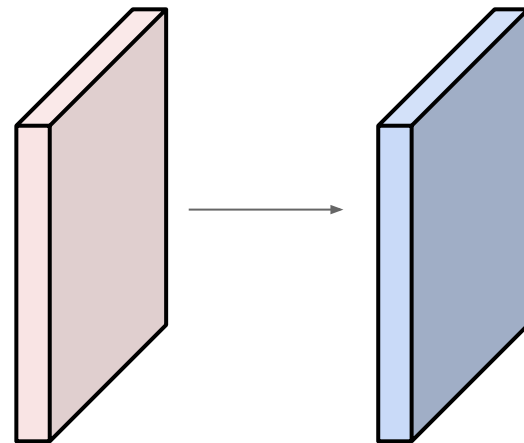
Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Let's assume output size is $H \times W \times D$.

What is D ? **10**



Examples time:

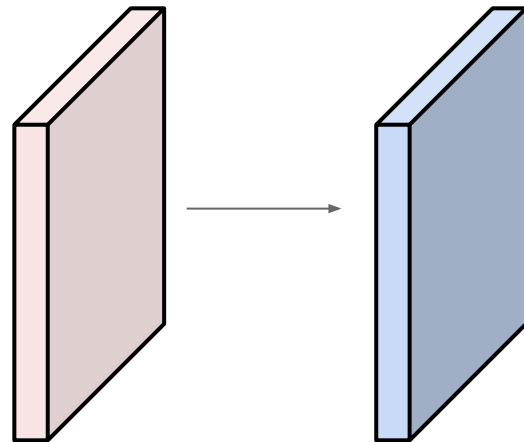
Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Let's assume output size is $H \times W \times D$.

What is D ? **10**

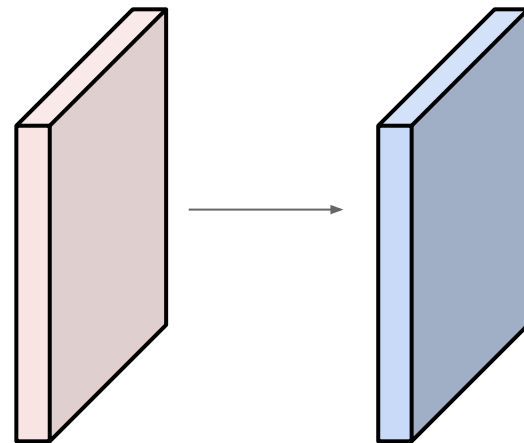
What is H or W ?



Examples time:

Input volume: **32x32x3**

10 **5x5** filters with stride **1**, pad **2**



Let's assume output size is $H \times W \times D$.

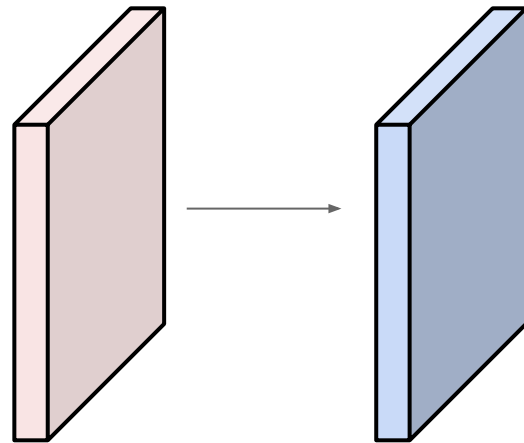
What is D ? 10

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 $H \times W \times D$.

What is D ? **10**

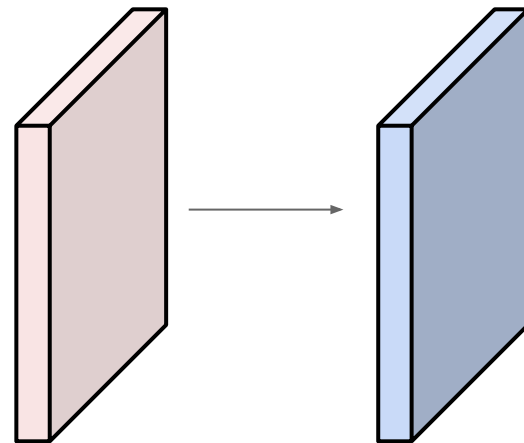
What is H or W ? $(32 + 2 * 2 - 5) / 1 + 1 = 32$

So the total output size is: **32x32x10**

Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

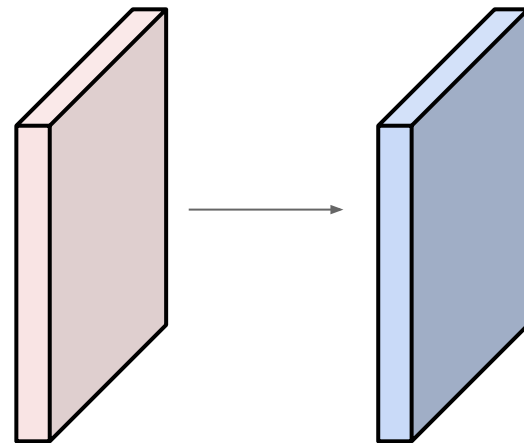


Number of parameters in this layer?

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)

$\Rightarrow 76*10 = 760$

Convolution layer: summary

Let's assume input is $W_1 \times H_1 \times C$

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 $W_2 \times H_2 \times K$
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

Common settings:

Let's assume input is $W_1 \times H_1 \times C$

Conv layer needs 4 hyperparameters:

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

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

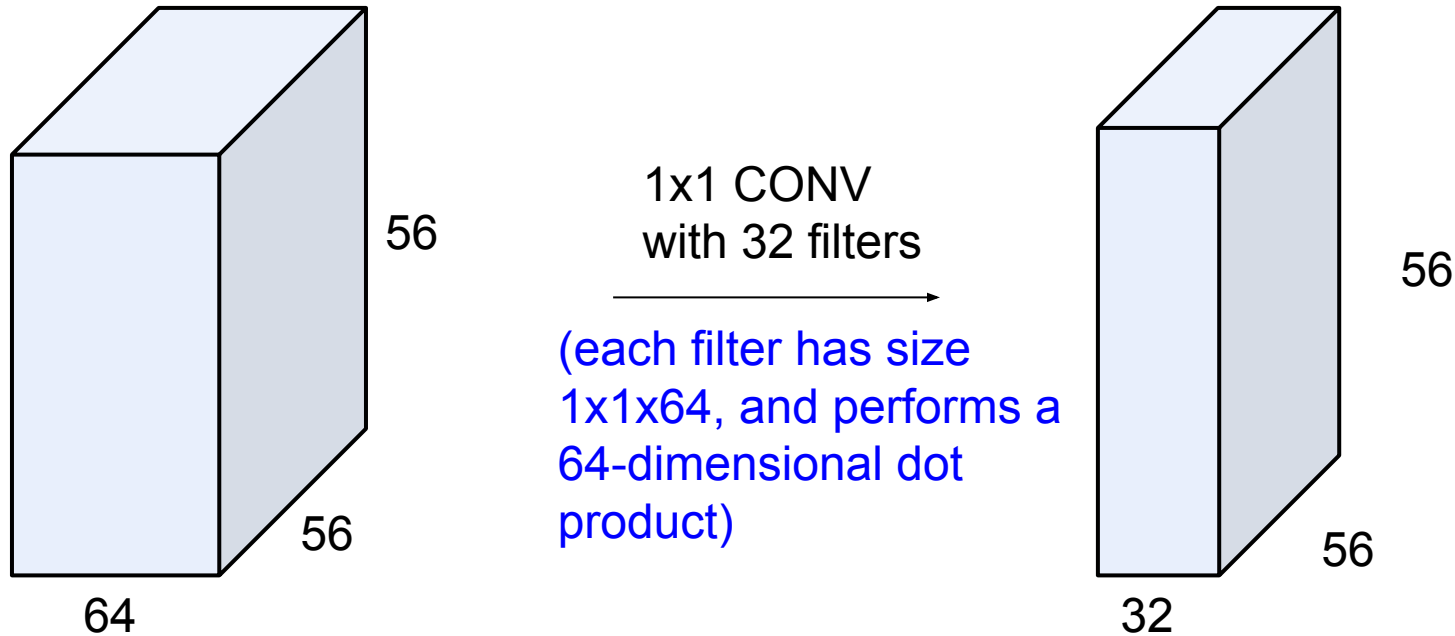
- **F** = 3, **S** = 1, **P** = 1
- **F** = 5, **S** = 1, **P** = 2
- **F** = 5, **S** = 2, **P** = ? (whatever fits)
- **F** = 1, **S** = 1, **P** = 0

This will produce an output of $W_2 \times H_2 \times K$
where:

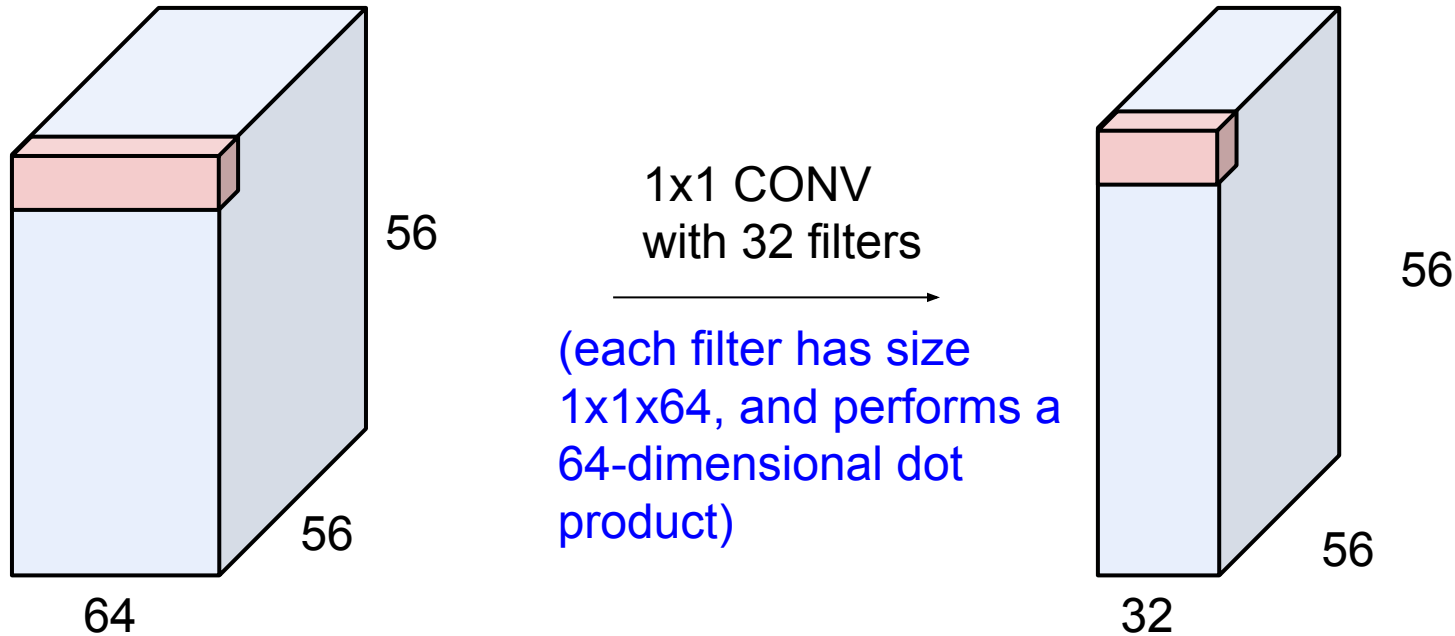
- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: F^2CK and K biases

(btw, 1x1 convolution layers are very useful)



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



Example: CONV layer in PyTorch

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 $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

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

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: $\begin{bmatrix} C_{out} \\ C_{in} \end{bmatrix}$.

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

PyTorch is licensed under [BSD 3-clause](#).

Conv layer needs 4 hyperparameters:

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

Example: CONV layer in Keras

Conv layer needs 4 hyperparameters:

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

Conv2D

[\[source\]](#)

```
keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, d:
```

2D convolution layer (e.g. spatial convolution over images).

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If `use_bias` is True, a bias vector is created and added to the outputs. Finally, if `activation` is not None, it is applied to the outputs as well.

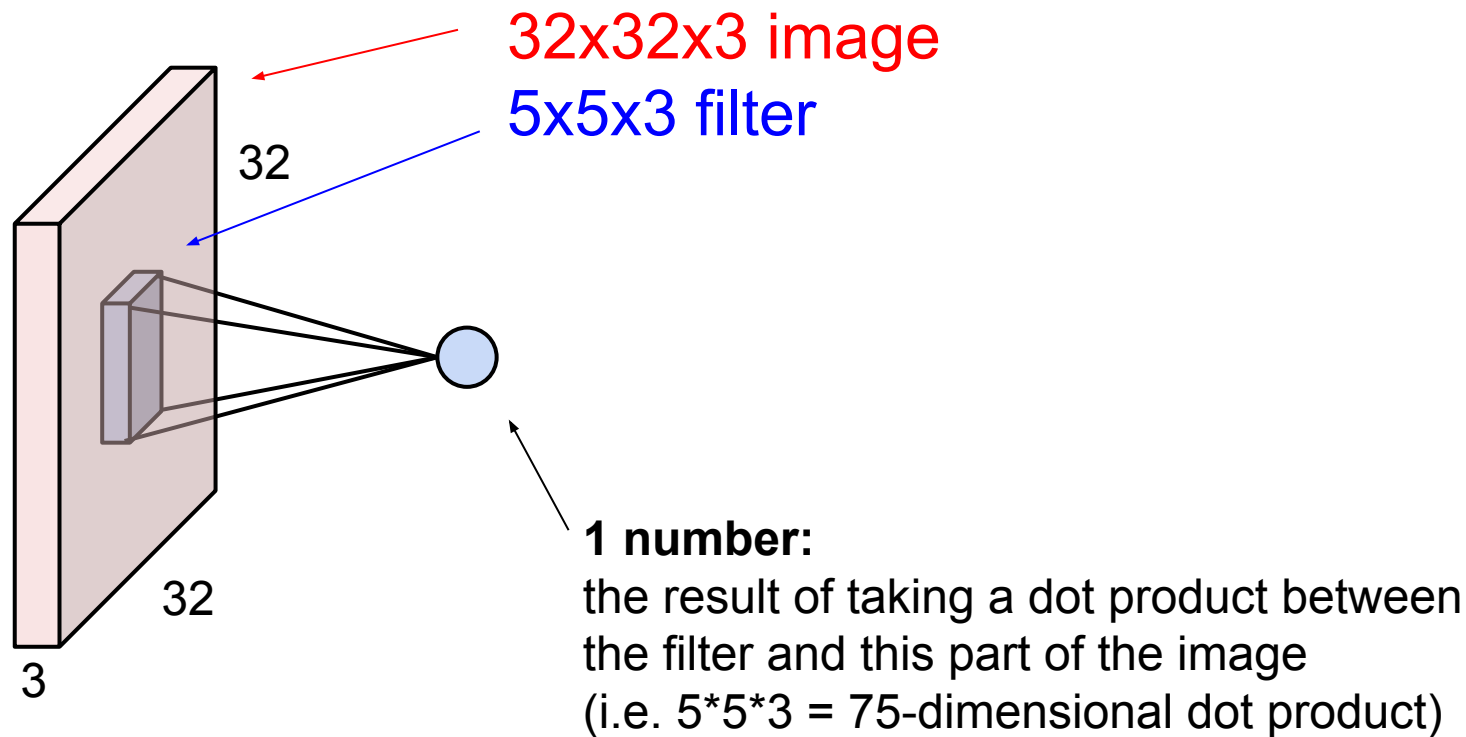
When using this layer as the first layer in a model, provide the keyword argument `input_shape` (tuple of integers, does not include the batch axis), e.g. `input_shape=(128, 128, 3)` for 128x128 RGB pictures in `data_format="channels_last"`.

Arguments

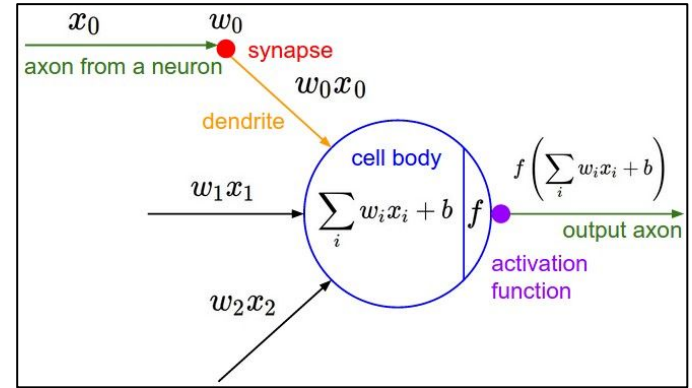
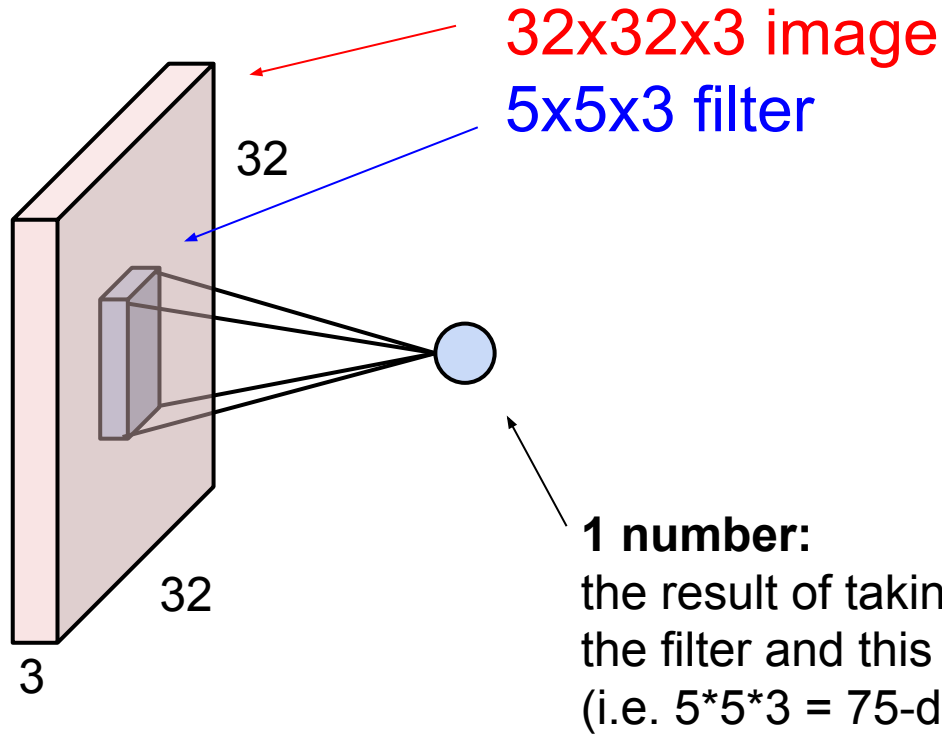
- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
- **kernel_size**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding**: one of "valid" or "same" (case-insensitive). Note that "same" is slightly inconsistent across backends with `strides != 1`, as described [here](#)
- **data_format**: A string, one of "channels_last" or "channels_first". The ordering of the dimensions in the inputs. "channels_last" corresponds to inputs with shape (batch, height, width, channels) while "channels_first" corresponds to inputs with shape (batch, channels, height, width). It defaults to the `image_data_format` value found in your Keras config file at `~/keras/keras.json`. If you never set it, then it will be "channels_last".

[Keras](#) is licensed under the [MIT license](#).

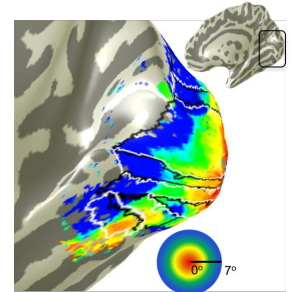
The brain/neuron view of CONV Layer



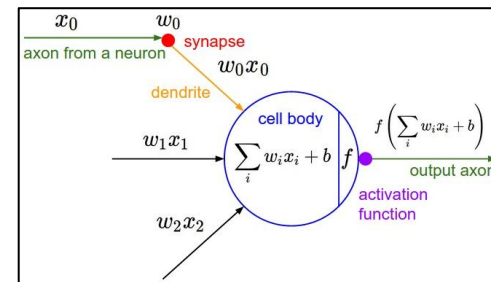
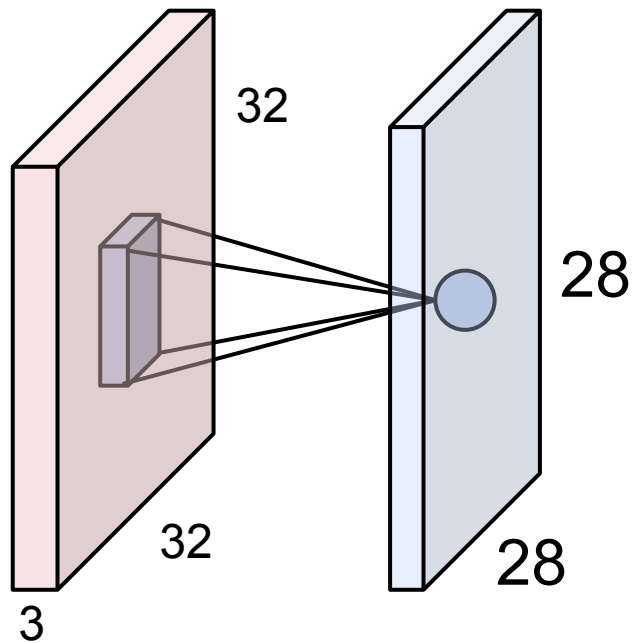
The brain/neuron view of CONV Layer



It's just a neuron with local connectivity...



Receptive field

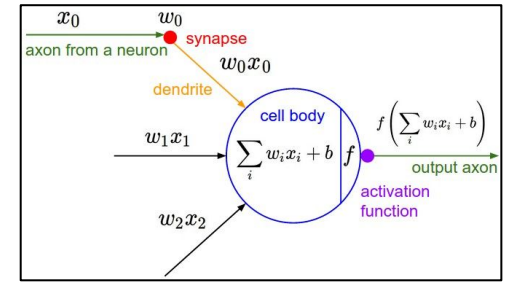
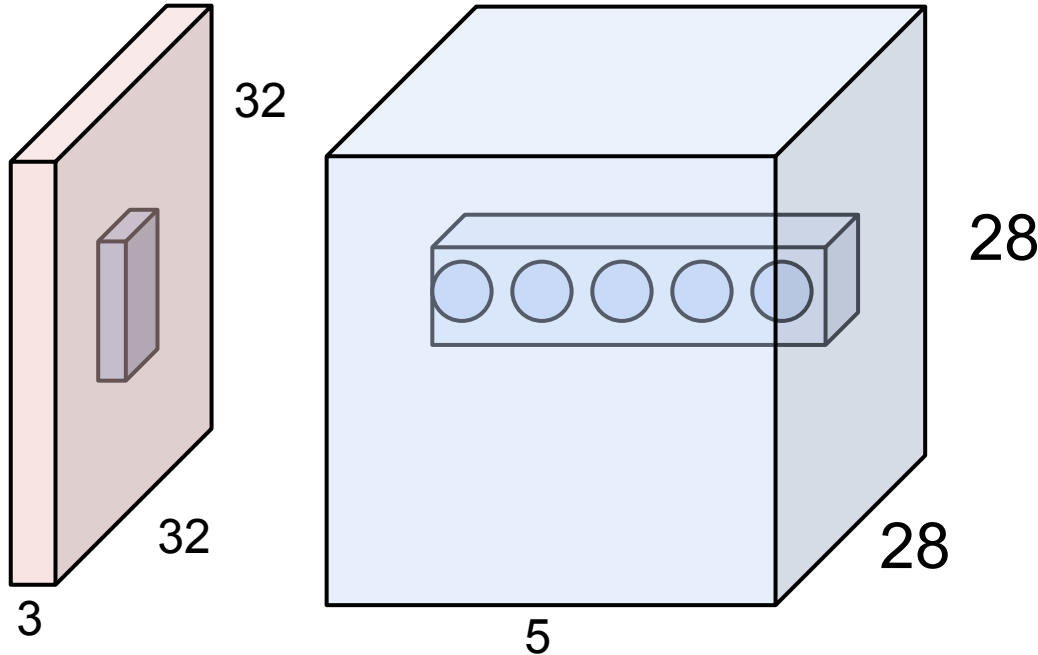


An activation map is a 28x28 sheet of neuron outputs:

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”

The brain/neuron view of CONV Layer



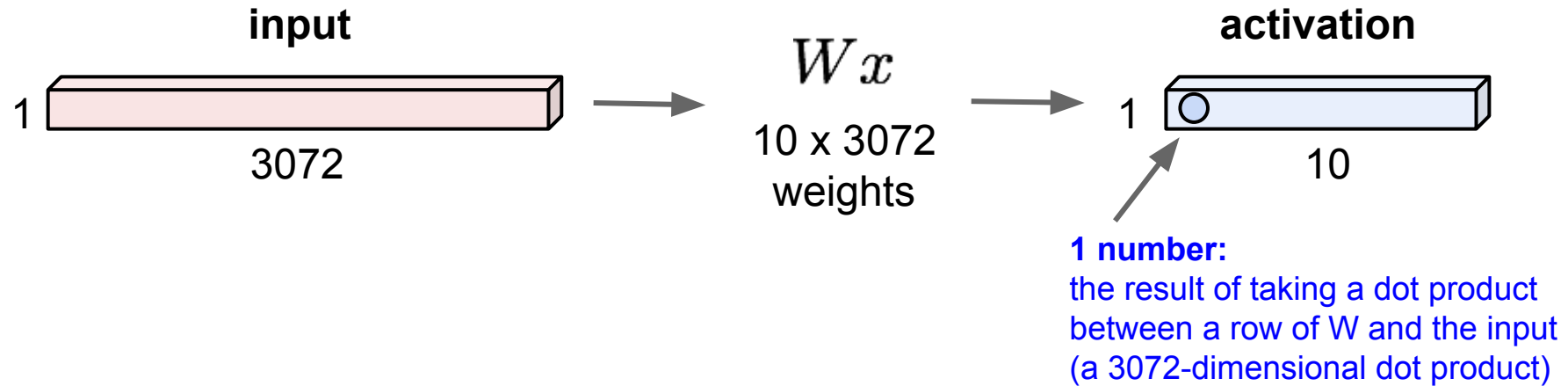
E.g. with 5 filters,
CONV layer consists of
neurons arranged in a 3D grid
(28x28x5)

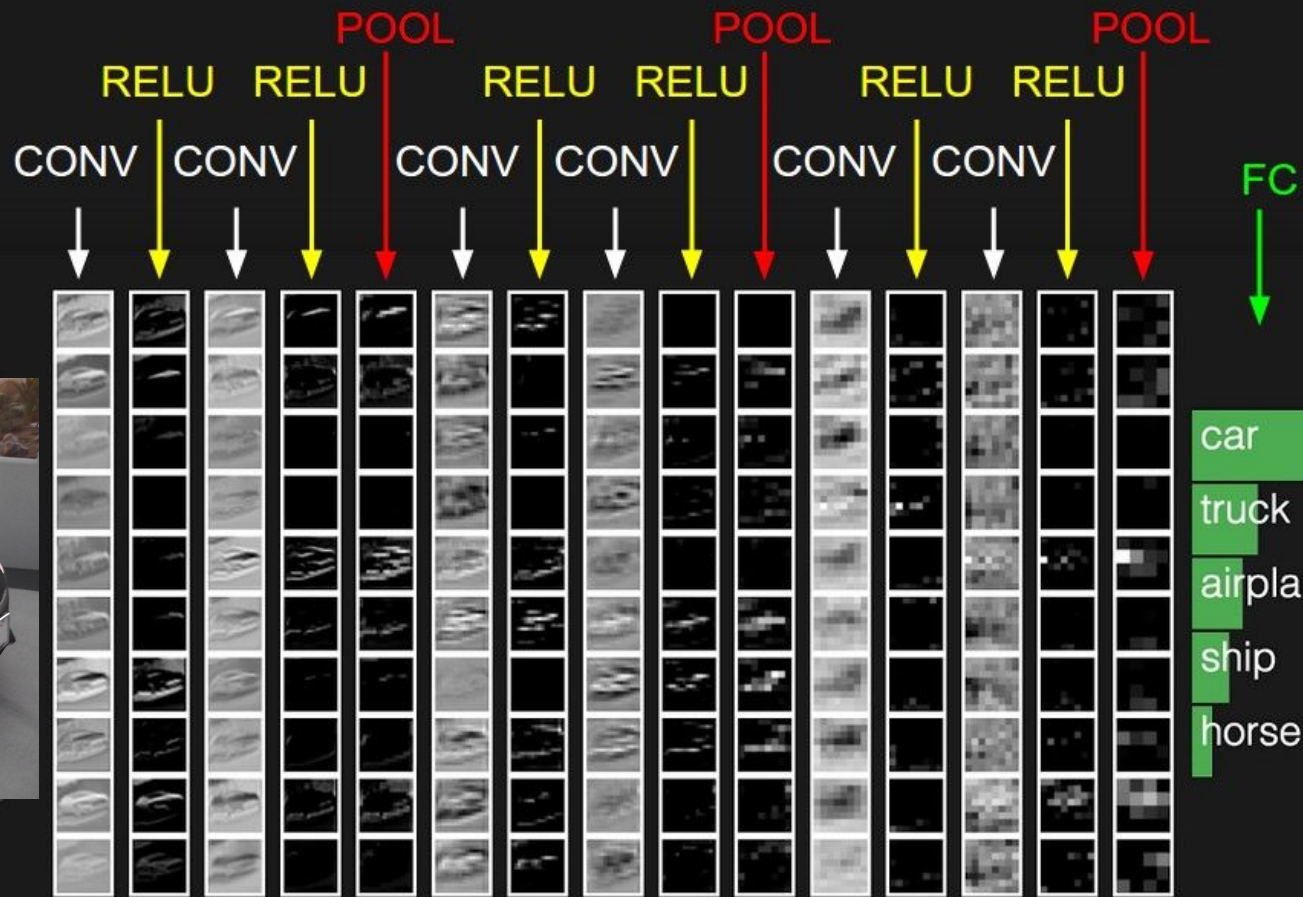
There will be 5 different
neurons all looking at the same
region in the input volume

Fully connected layer is a special convolution

32x32x3 image $\rightarrow$ stretch to 3072 x 1

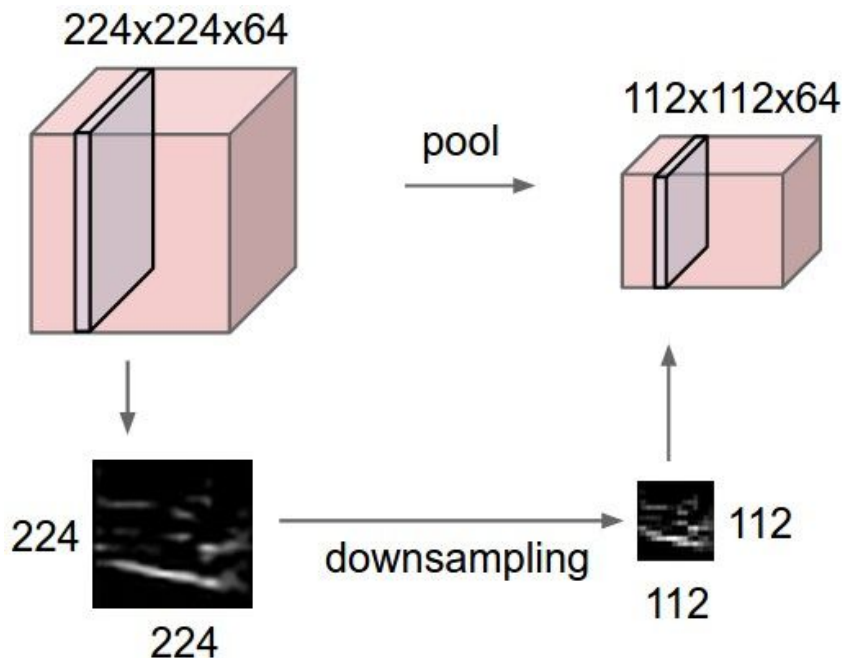
Each neuron
looks at the full
input volume



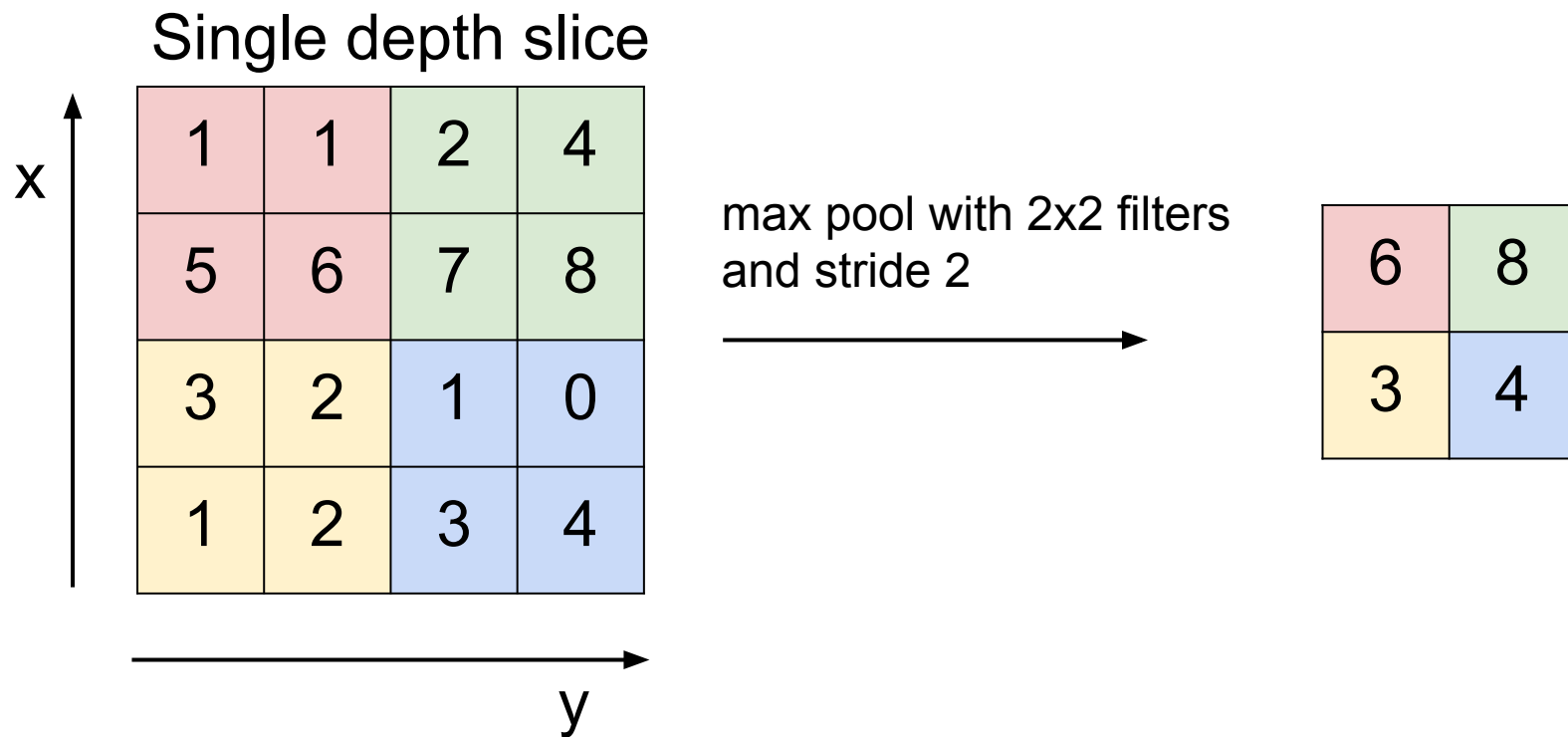


Pooling layer

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



MAX POOLING



Pooling layer: summary

Let's assume input is $W_1 \times H_1 \times C$

Conv layer needs 2 hyperparameters:

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

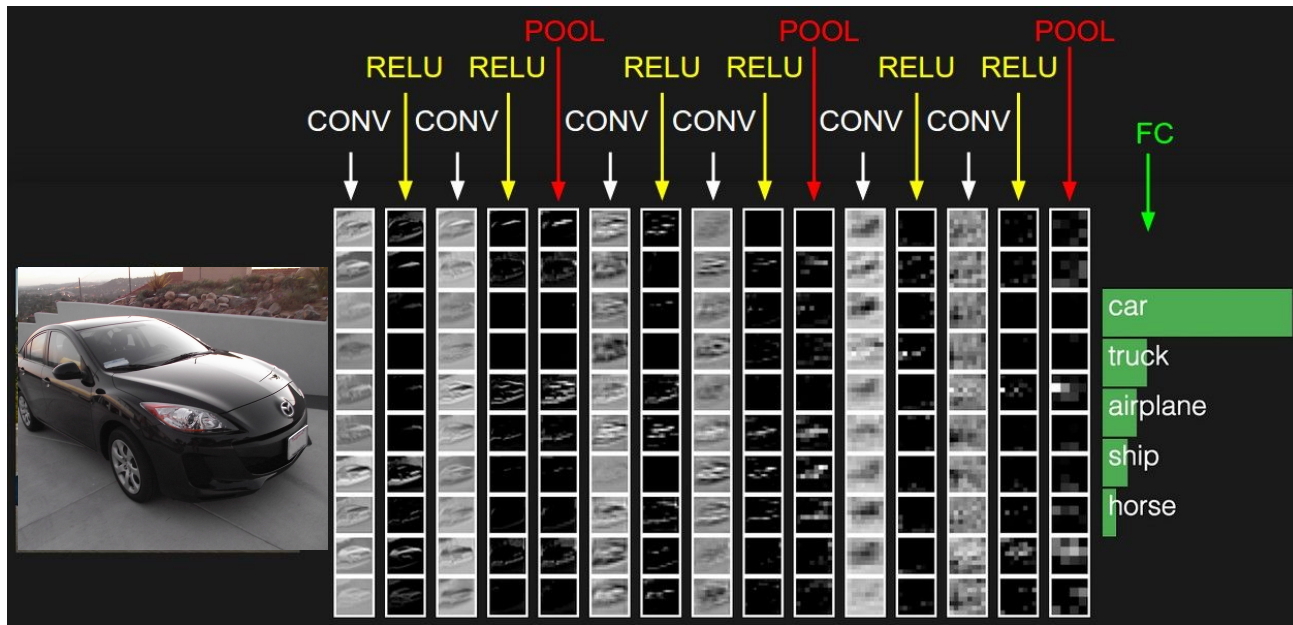
This will produce an output of $W_2 \times H_2 \times C$ where:

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

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
 $[(\text{CONV-RELU})^N \text{- POOL?}]^M \text{- (FC-RELU)}^K, \text{SOFTMAX}$
where N is usually up to ~5, M is large, $0 \leq K \leq 2$.
 - but recent advances such as ResNet/GoogLeNet have challenged this paradigm