Lecture 8: Optimization

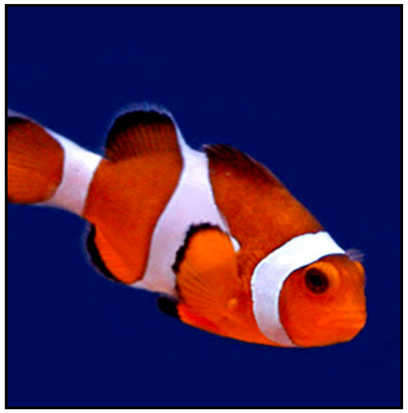
- Pset3 due tonight
- **Important**: Name your .ipynb file as `<uniquename>_<umid>.ipynb`, for example `hzwang_20200201.ipynb`.

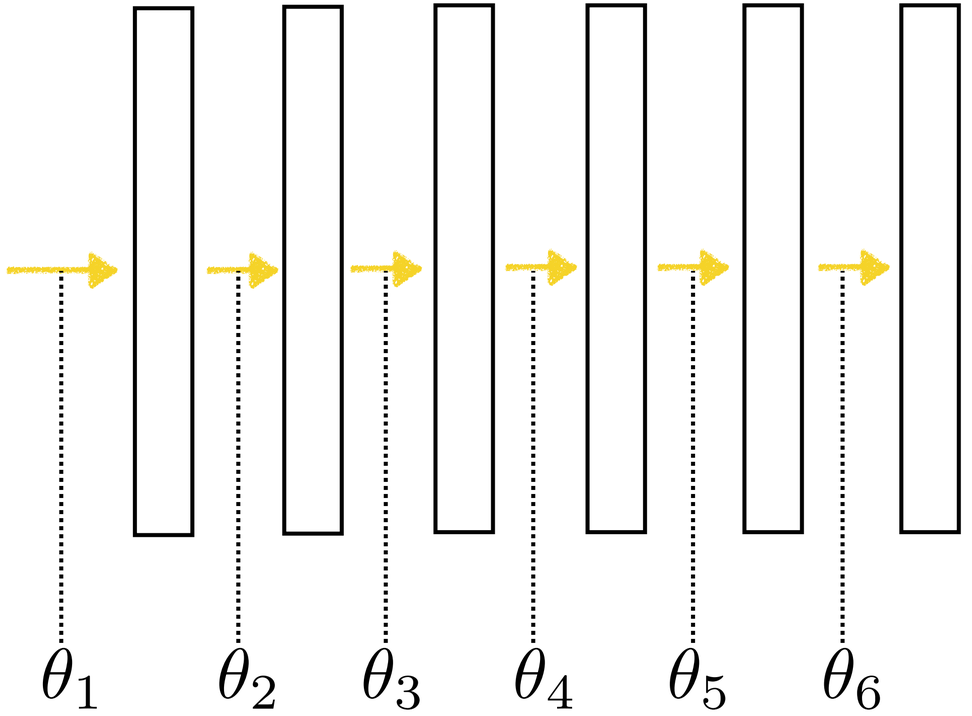
Deep Neural Networks

$\mathbf{y}_i$

"clown fish"

$\mathbf{x}_i$





$\mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$

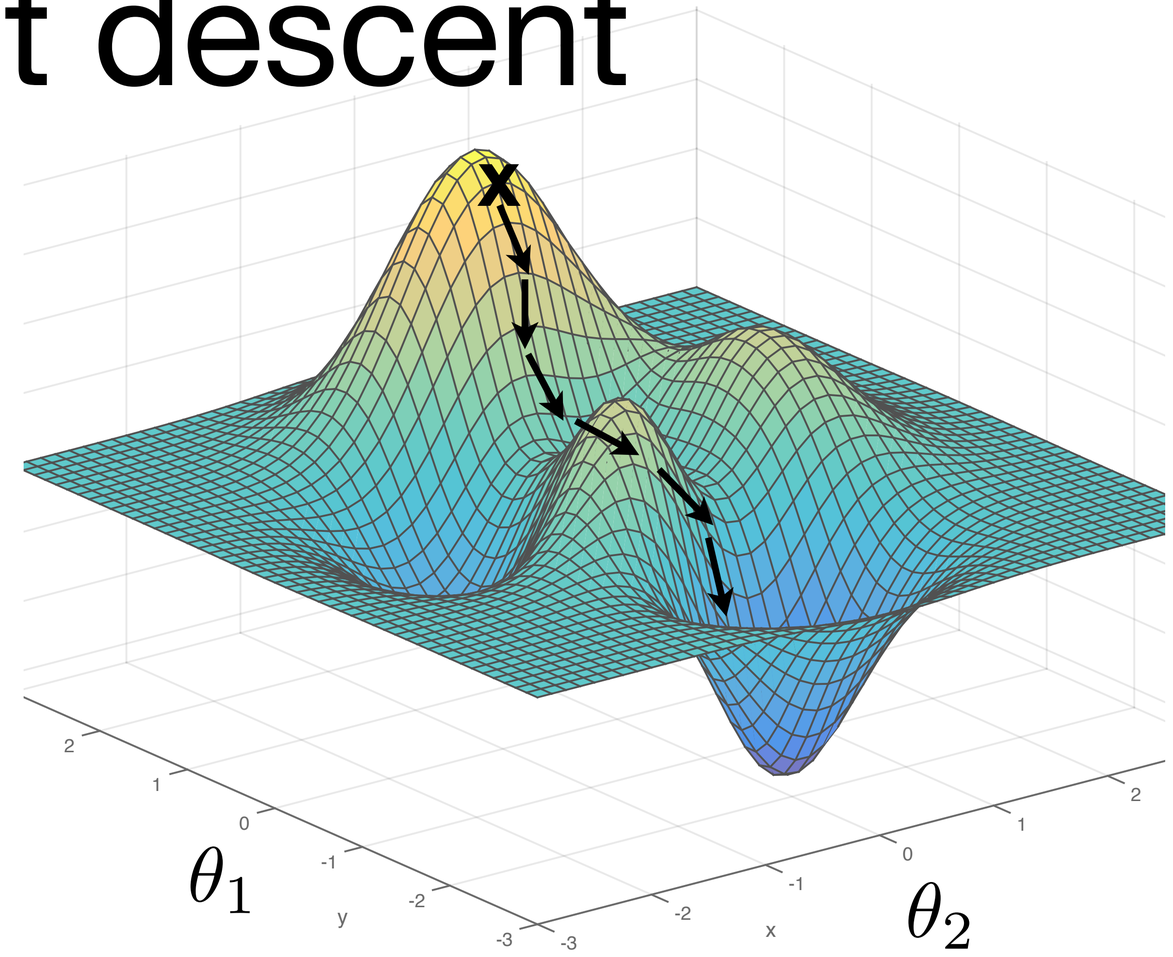
Examples:
MSE, cross-entropy, etc

$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$$

Source: Freeman, Torralba, Isola

Gradient descent

$J(\theta)$

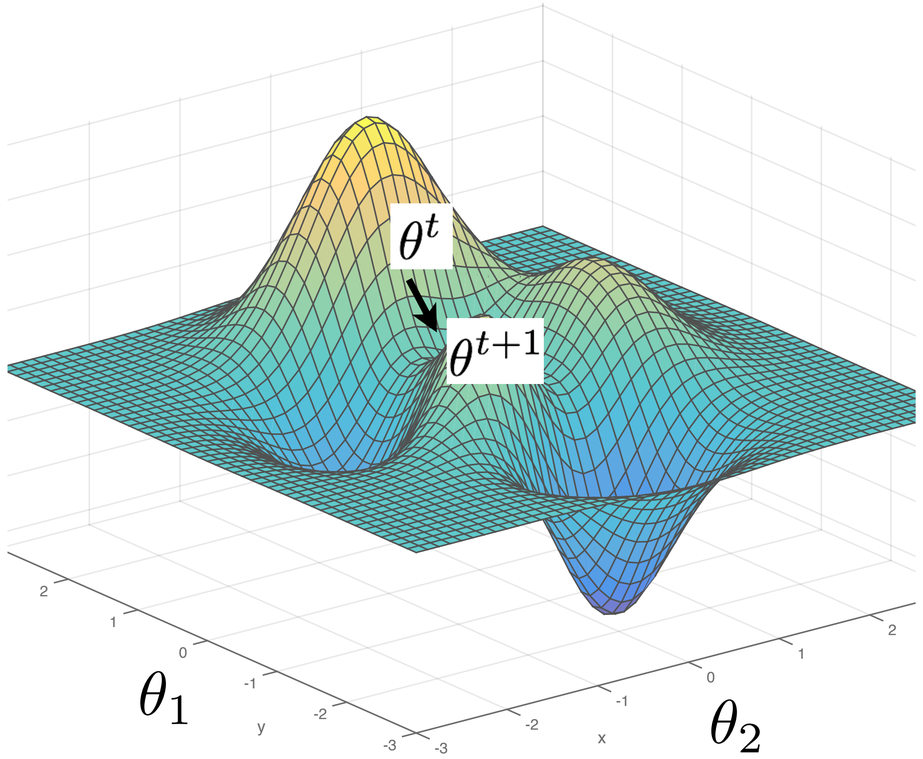


$$\theta^* = \arg\min_{\theta} J(\theta)$$

Source: Freeman, Torralba, Isola

Gradient descent

$$\theta^* = \arg\min_\theta \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{J(\theta)}$$



One iteration of gradient descent:

$$\theta^{t+1} = \theta^t - \eta_t \left. \frac{\partial J(\theta)}{\partial \theta} \right|_{\theta=\theta^t}$$

learning rate

Source: Freeman, Torralba, Isola

Optimization

Params

θ → J → $J(\theta)$
$\nabla_\theta J(\theta)$
$H_\theta(J(\theta))$

$$\theta^* = \arg\min_\theta J(\theta)$$

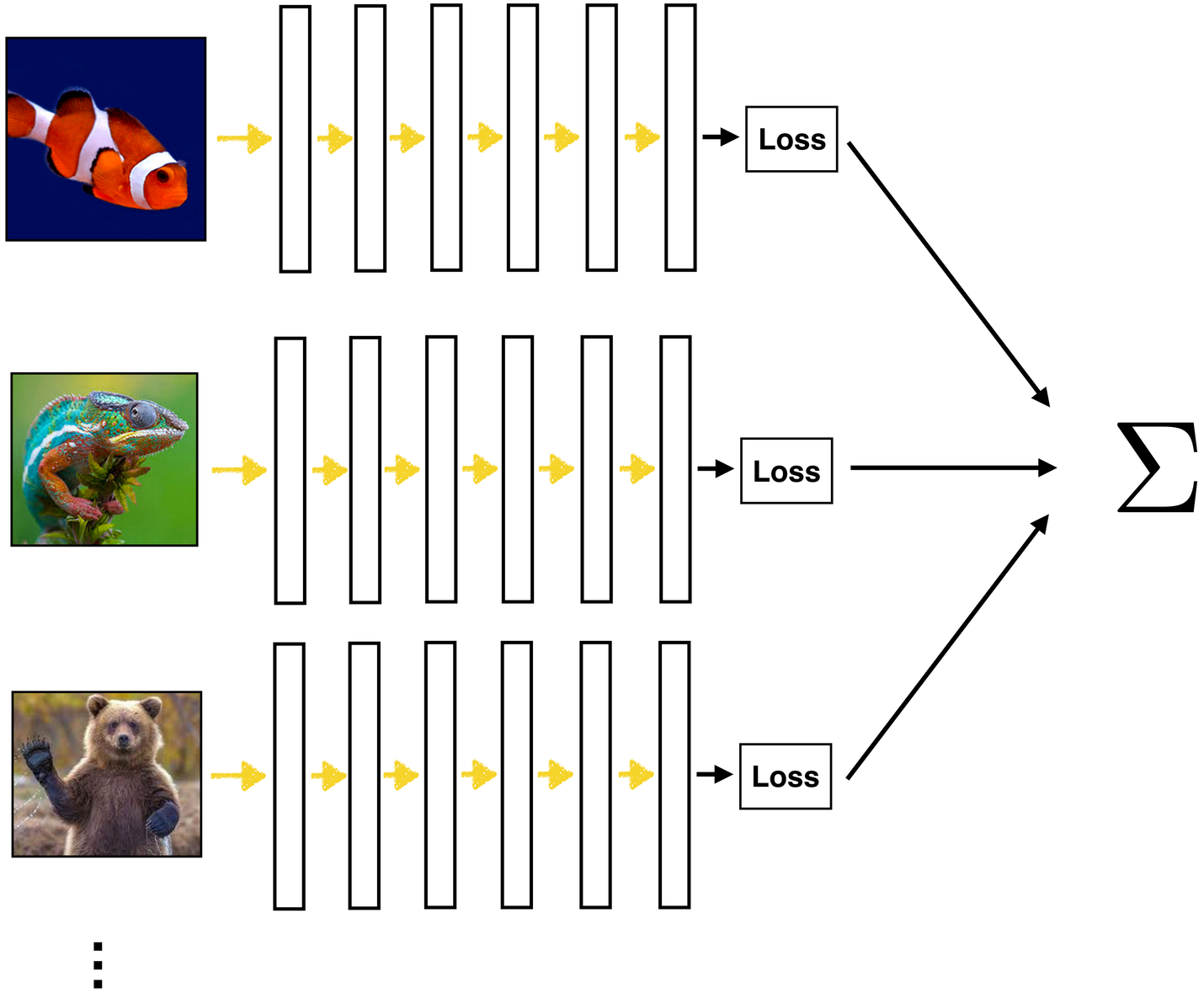
- What's the knowledge we have about J?
  - We can evaluate $J(\theta)$ ← Black box optimization
  - We can evaluate $J(\theta)$ and $\nabla_\theta J(\theta)$ ← First order optimization
  - We can evaluate $J(\theta)$, $\nabla_\theta J(\theta)$, and $H_\theta(J(\theta))$ ← Second order optimization

Gradient

Hessian

Source: Freeman, Torralba, Isola

Batch (parallel) processing



Source: Freeman, Torralba, Isola

Stochastic gradient descent (SGD)

- Want to minimize overall loss function **J**, which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.

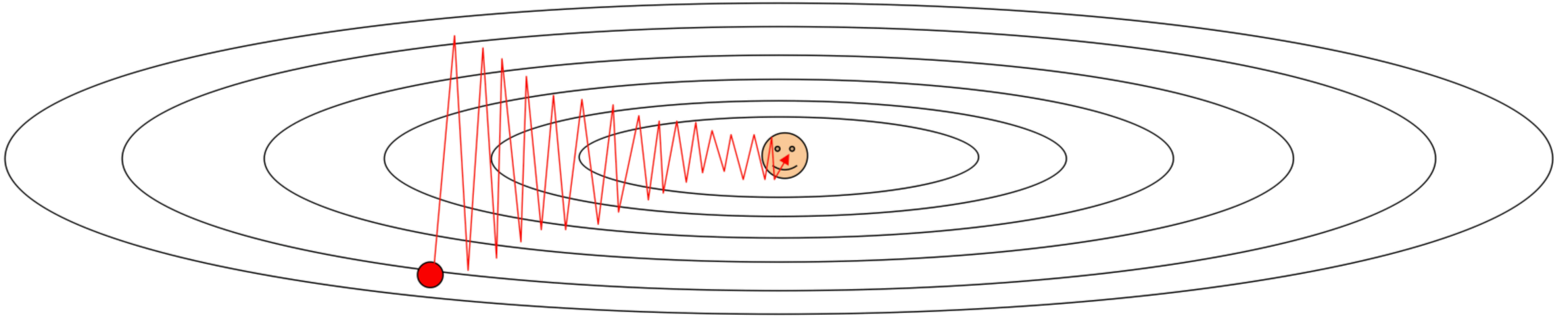
  If batchsize=1 then θ is updated after each example.

  If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).
- Advantages
  - Faster: approximate total gradient with small sample
  - Implicit regularizer
- Disadvantages
  - High variance, unstable updates

Source: Freeman, Torralba, Isola

Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Source: Johnson

Momentum

- Basic idea: like a ball rolling down a hill, we should build up speed so as to make faster progress when “on a roll”

- Can dampen oscillations in SGD updates

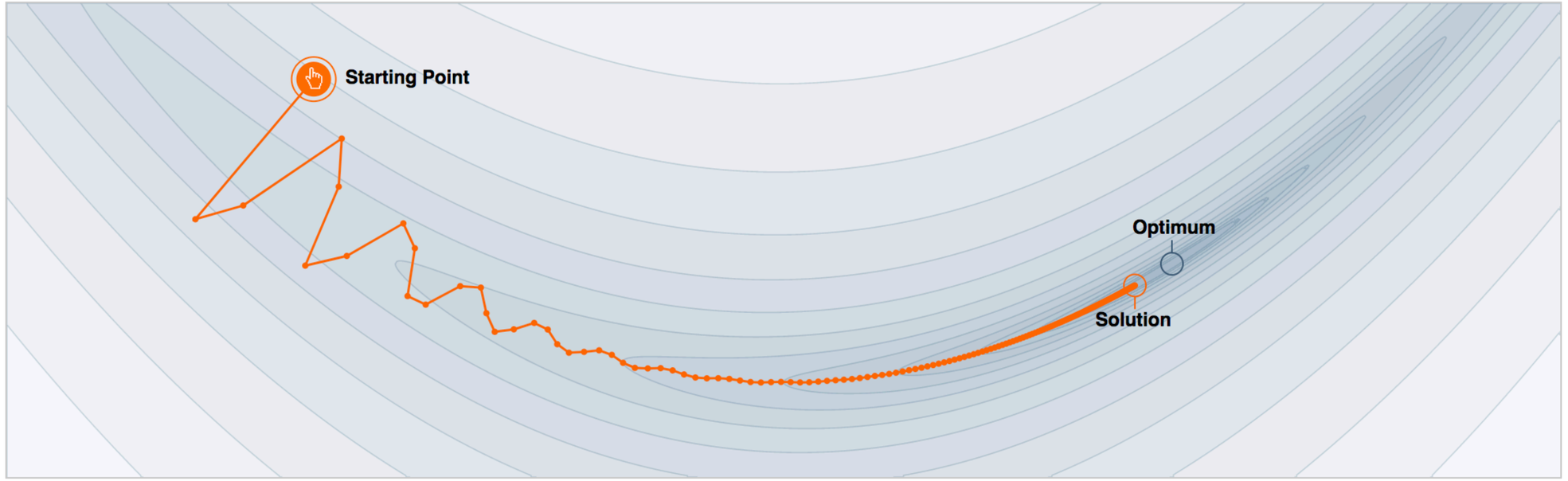
Momentum update rule

$$z^{t+1} = \beta\theta^t + (1-\beta)\nabla J\left(\theta^t\right)$$
$$\theta^{t+1} = \theta^t - \eta z^{t+1}$$

Used in popular methods including Adam

Source: Freeman, Torralba, Isola

Why Momentum Really Works



Step-size α = 0.02

0 0.003 0.006

Momentum β = 0.99

0.00 0.500 0.990

We often think of Momentum as a means of dampening oscillations and speeding up the iterations, leading to faster convergence. But it has other interesting behavior. It allows a larger range of step-sizes to be used, and creates its own oscillations. What is going on?

GABRIEL GOH
UC Davis

April. 4
2017

Citation:
Goh, 2017

[https://distill.pub/2017/momentum/]

Source: Freeman, Torralba, Isola

RMSProp

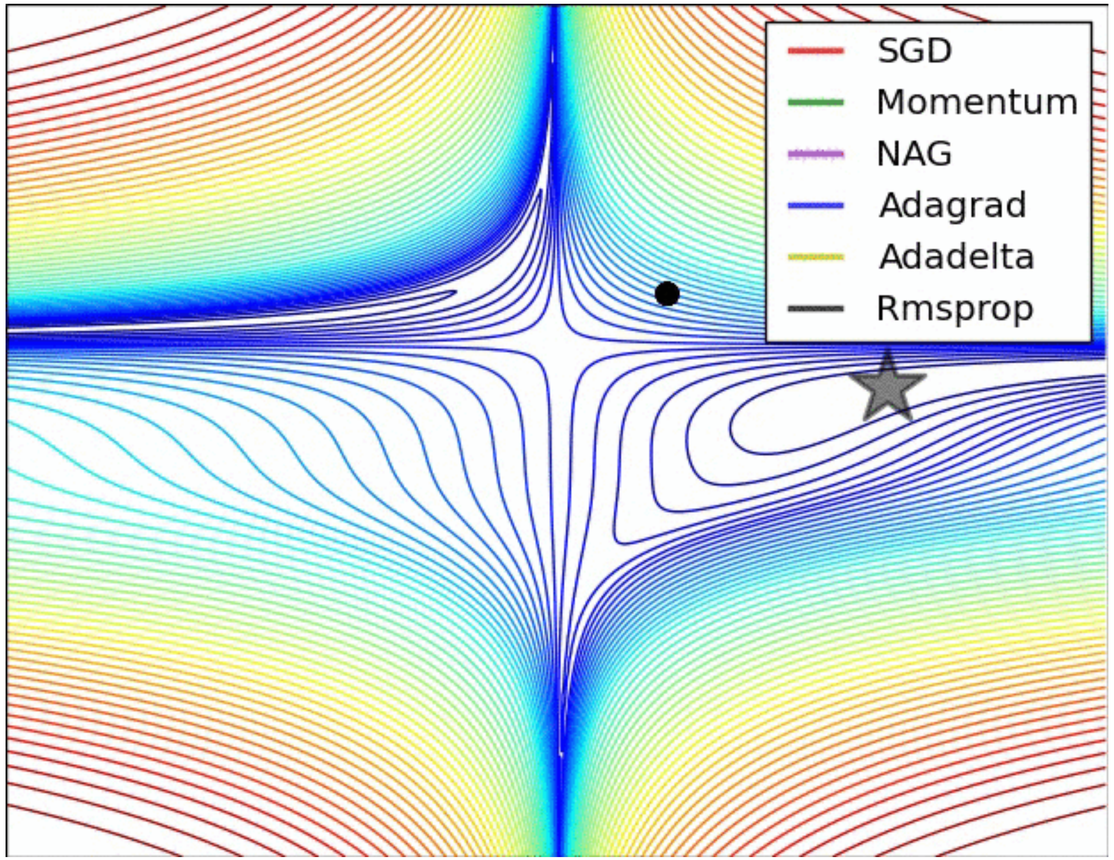
- Elementwise learning rate
- Damp learning rate for “steep” directions
- Speed up learning rate for “flat” directions

RMSProp update rule

$$g_t^2 = \beta g_{t-1}^2 + (1-\beta)(\nabla J(\theta))^2$$
$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{g_t^2}} \nabla J(\theta)$$

Adam is essentially the combination of momentum and RMSProp

Comparison of gradient descent variants



[http://ruder.io/optimizing-gradient-descent/]

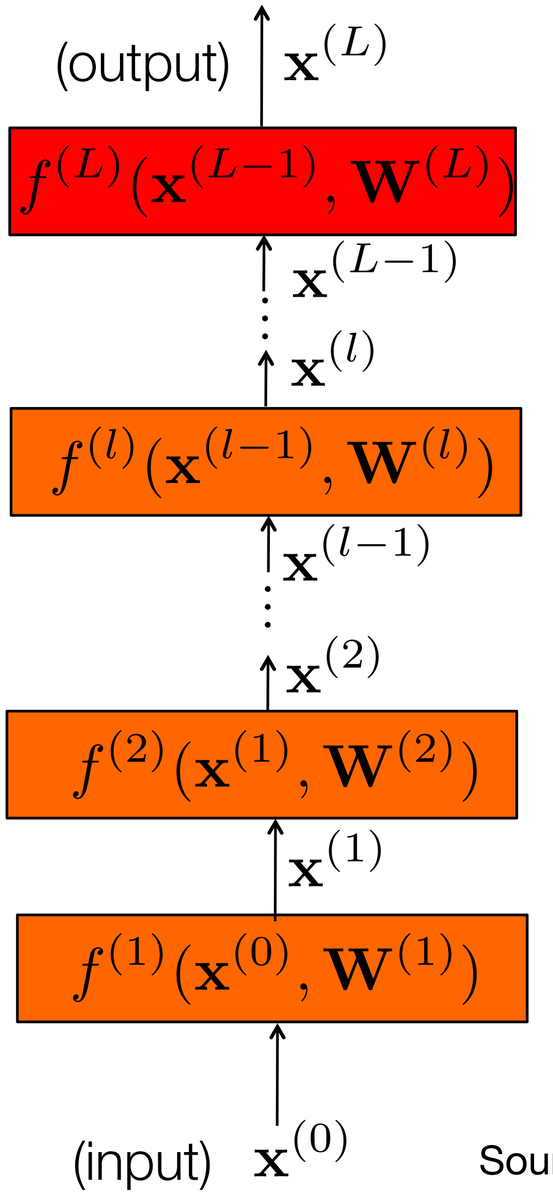
Source: Freeman, Torralba, Isola

Forward pass

- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass**: takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:

$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.



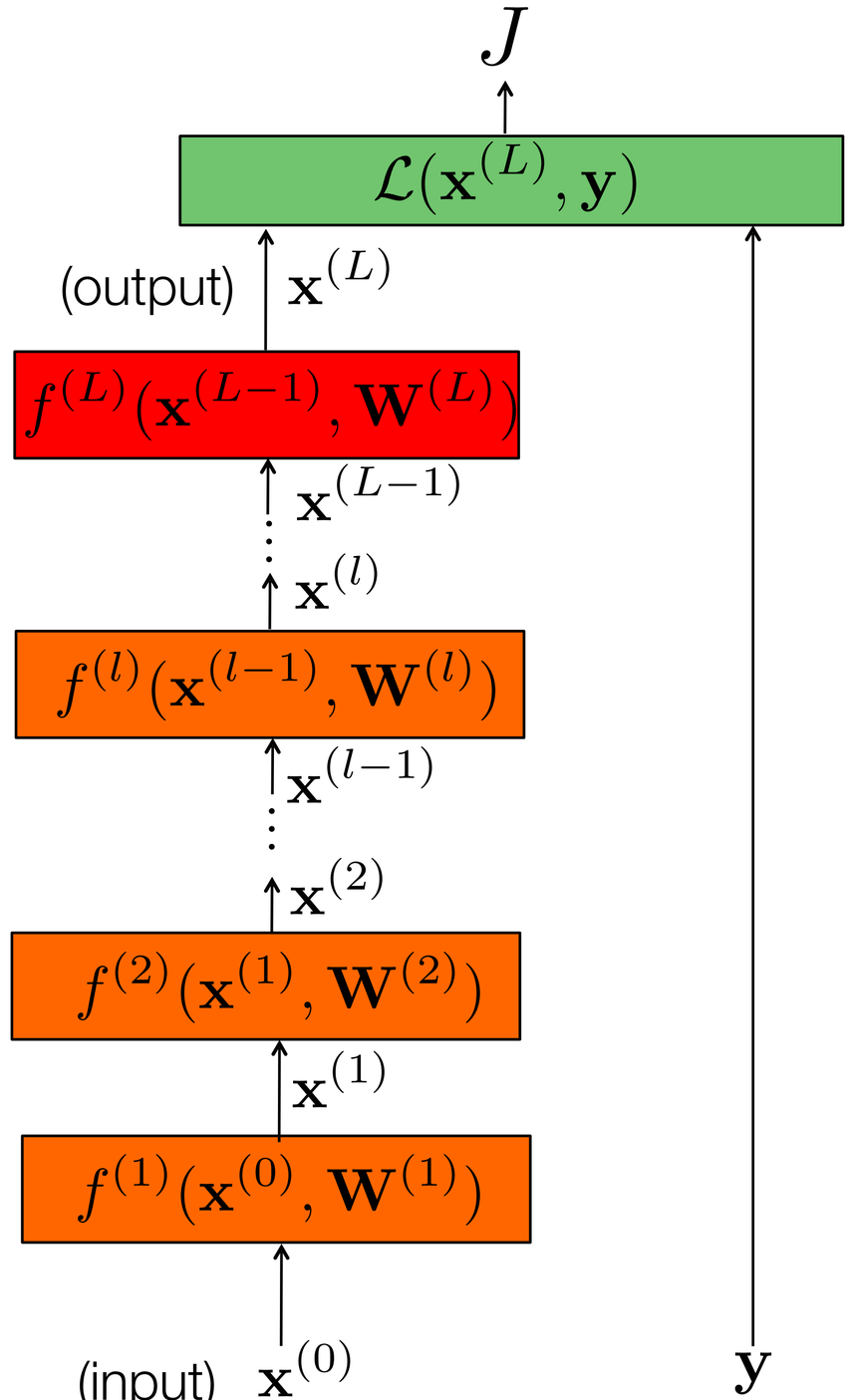
Source: Freeman, Torralba, Isola

Forward pass

- Consider model with L layers. Layer l has vector of weights $\mathbf{W}^{(l)}$
- **Forward pass**: takes input $\mathbf{x}^{(l-1)}$ and passes it through each layer $f^{(l)}$:

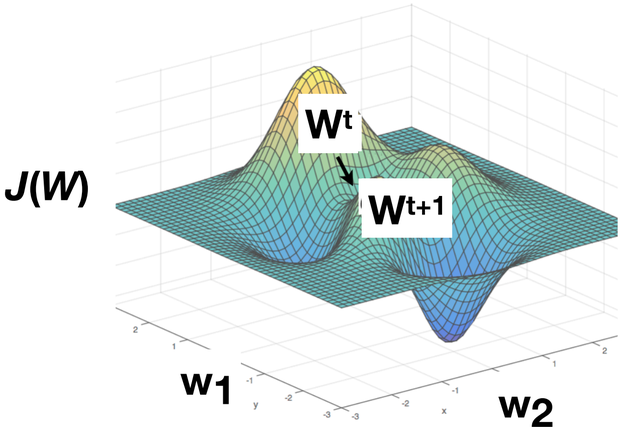
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

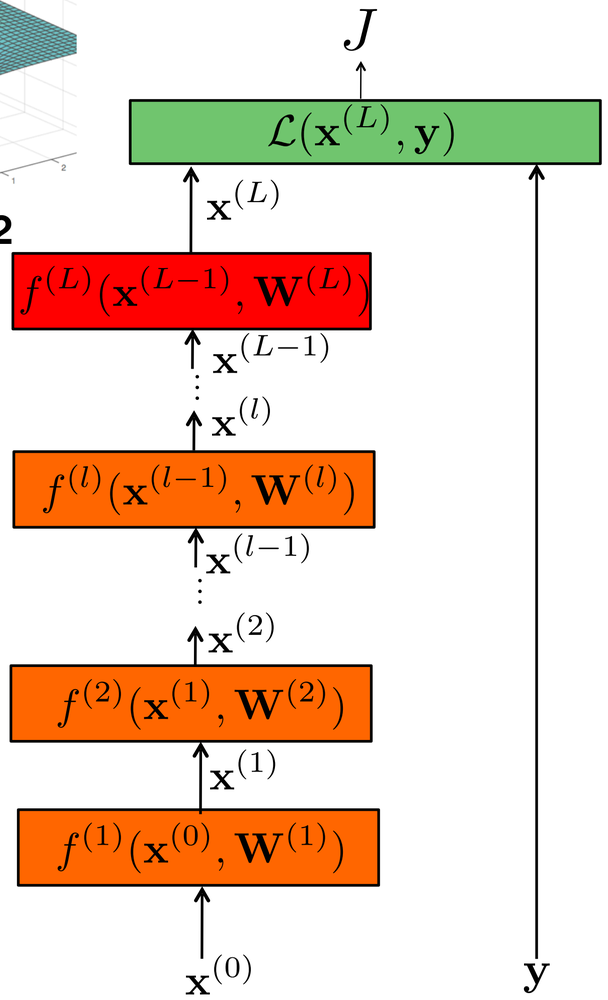
- Output of layer l is $\mathbf{x}^{(l)}$.
- Network output (top layer) is $\mathbf{x}^{(L)}$.
- **Loss function** $\mathcal{L}$ compares $\mathbf{x}^{(L)}$ to $\mathbf{y}$.
- Overall loss is the sum of the cost over all training examples: $J = \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i)$



Source: Freeman, Torralba, Isola

Gradient descent





- We need to compute gradients of the cost with respect to model parameters $\mathbf{W}^{(l)}$.

- By design, each layer is differentiable with respect to its parameters and input.

Source: Freeman, Torralba, Isola

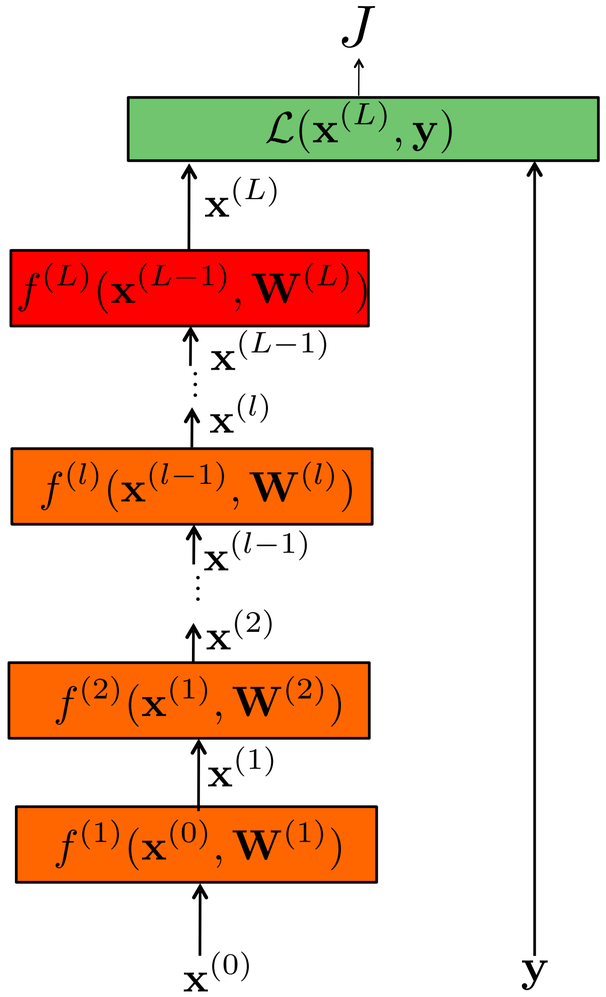
Computing gradients

To compute the gradients, we could start by writing the full energy J as a function of the network parameters.

$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f^{(L)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}_i^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \dots \mathbf{W}^{(L)}), \mathbf{y}_i)$$

And then compute the partial derivatives…

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$$



instead, we can use the chain rule to derive a compact algorithm: **backpropagation**

Source: Freeman, Torralba, Isola

Manually calculating gradient

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$
$$\mathcal{R} = \frac{1}{2}w^2$$
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda\mathcal{R}.$$

$$\begin{aligned}
\mathcal{L}_{\text{reg}} &= \frac{1}{2}(\sigma(wx+b) - t)^2 + \frac{\lambda}{2}w^2 \\
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b) - t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b) - t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx+b) - t)\frac{\partial}{\partial w}(\sigma(wx+b) - t) + \lambda w \\
&= (\sigma(wx+b) - t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\
&= (\sigma(wx+b) - t)\sigma'(wx+b)x + \lambda w
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}_{\text{reg}}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b) - t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b) - t)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx+b) - t)\frac{\partial}{\partial b}(\sigma(wx+b) - t) + 0 \\
&= (\sigma(wx+b) - t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b) - t)\sigma'(wx+b)
\end{aligned}$$

Manually calculating gradient is cumbersome:

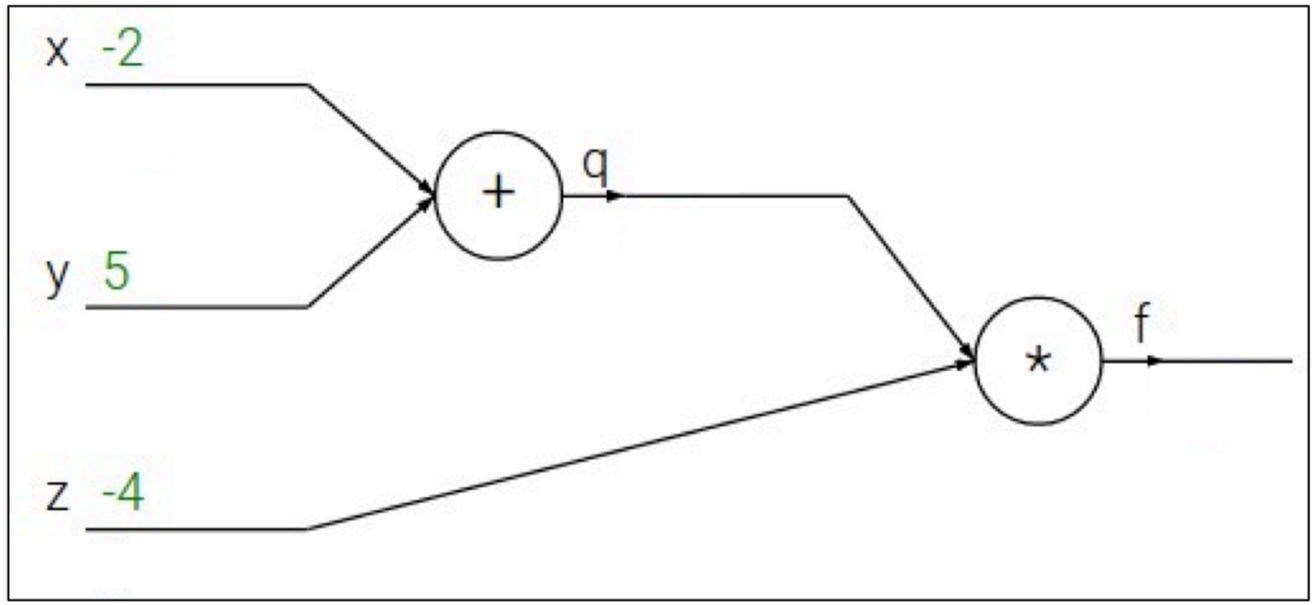
1. a lot of repeated computation
2. error-prone
3. not scalable

Source: Grosse, Ba

Backpropagation: Simple Example

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

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

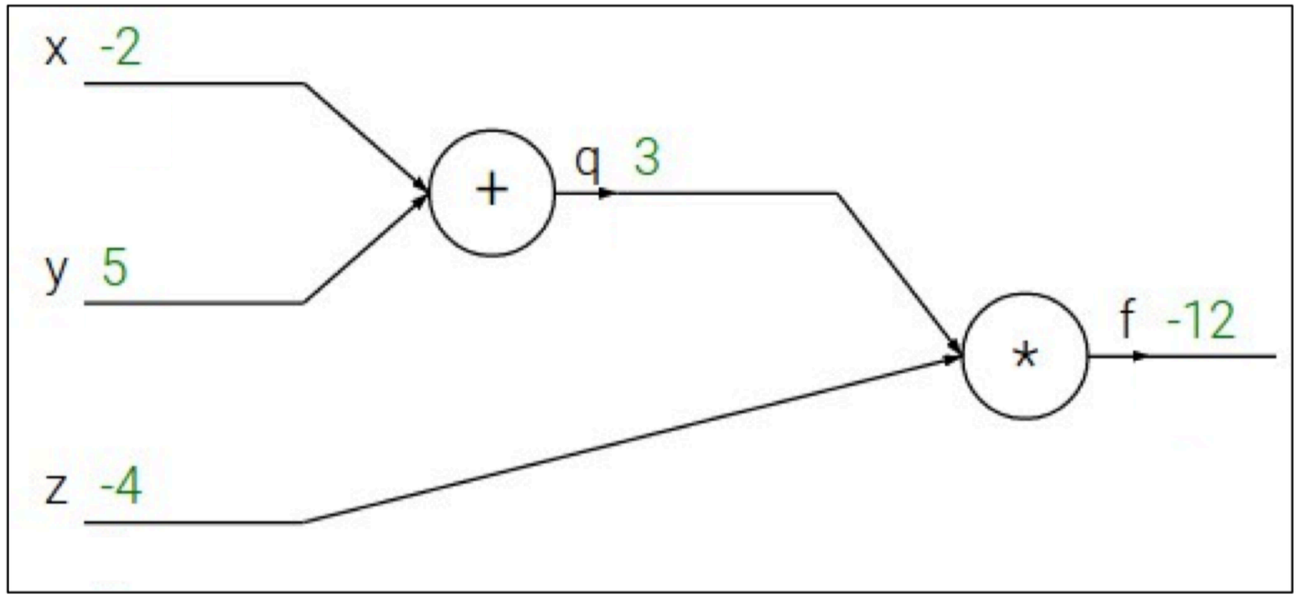


Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

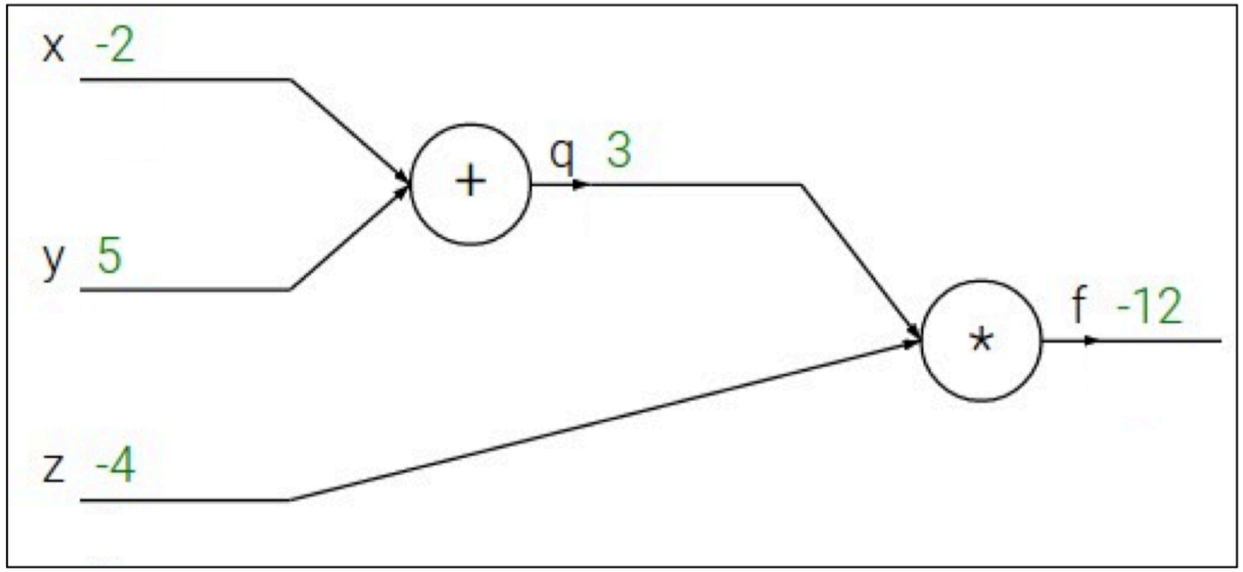
$$q = x + y \qquad f = qz$$

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

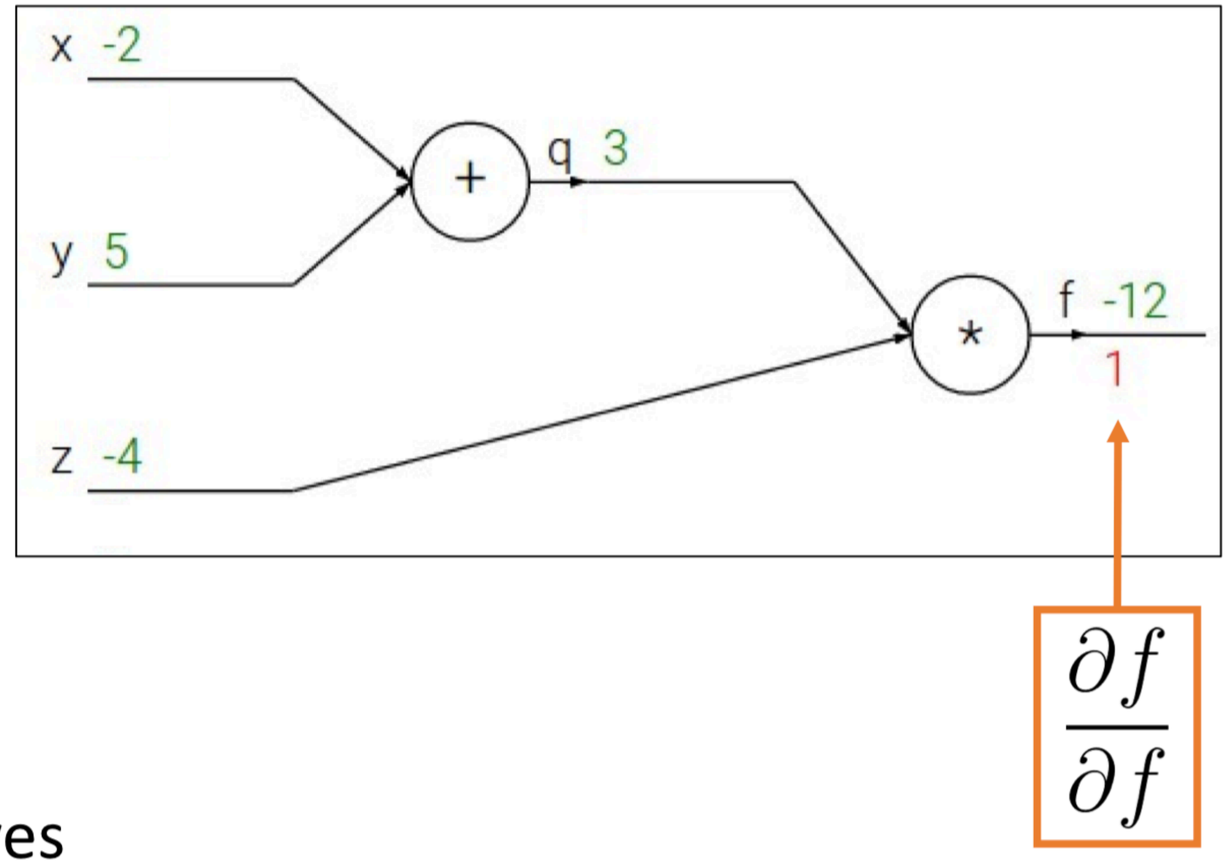
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

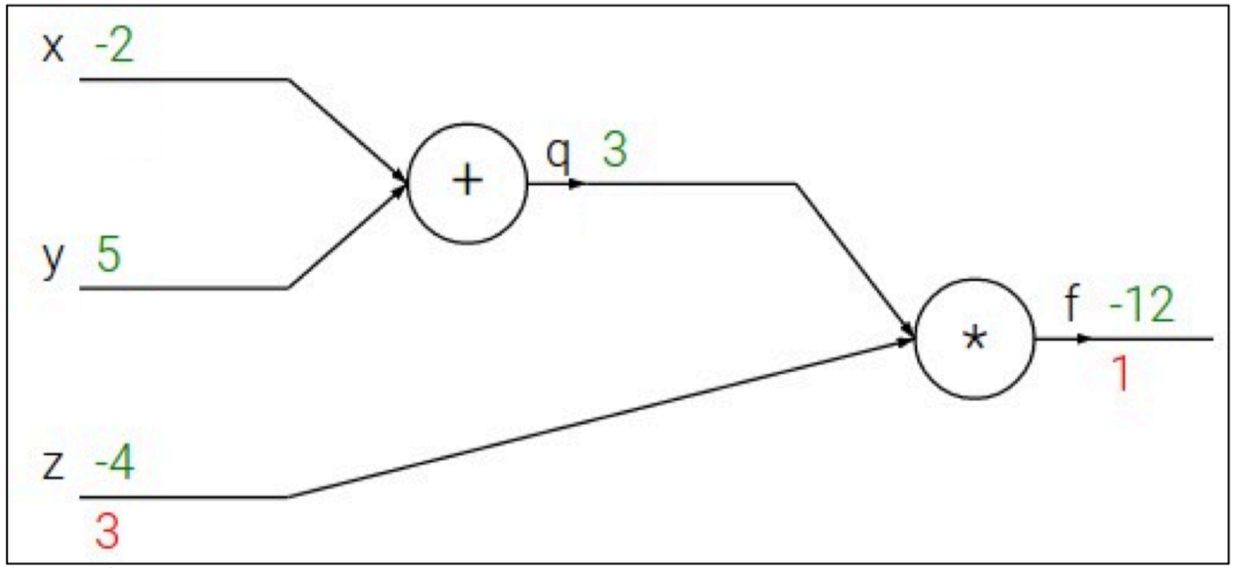
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

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

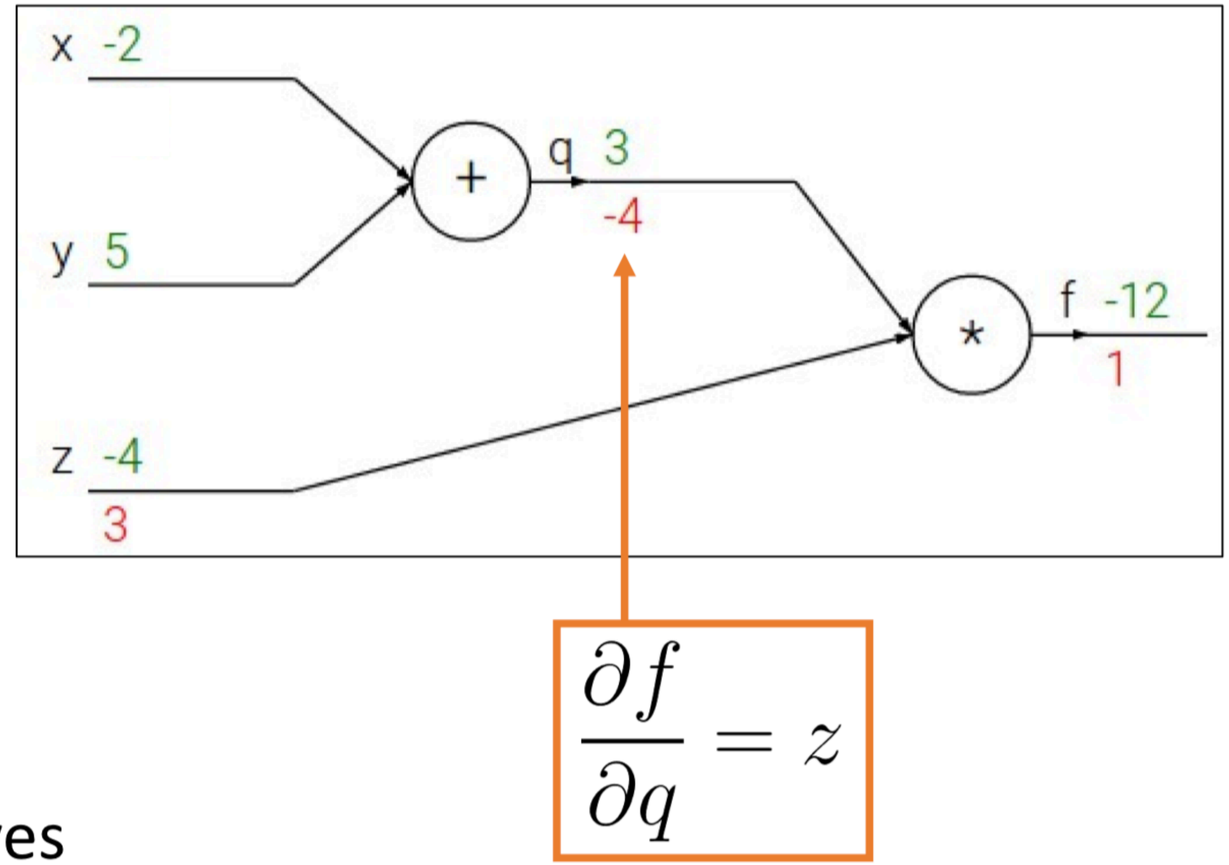
$$\frac{\partial f}{\partial z} = q$$

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

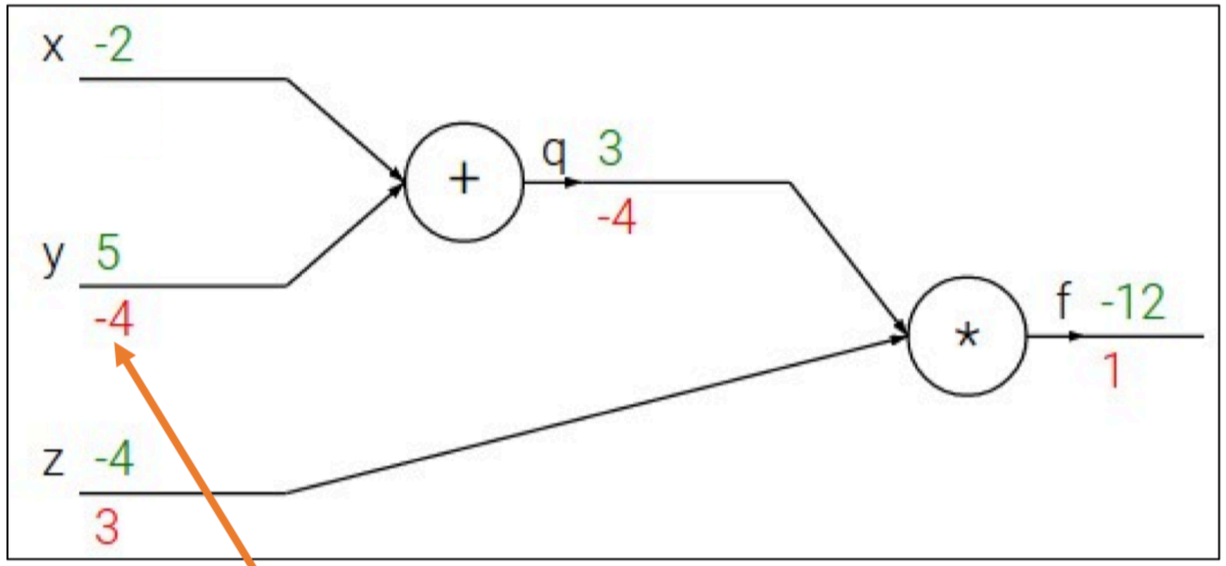
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

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

Chain Rule

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

$$\frac{\partial q}{\partial y} = 1$$

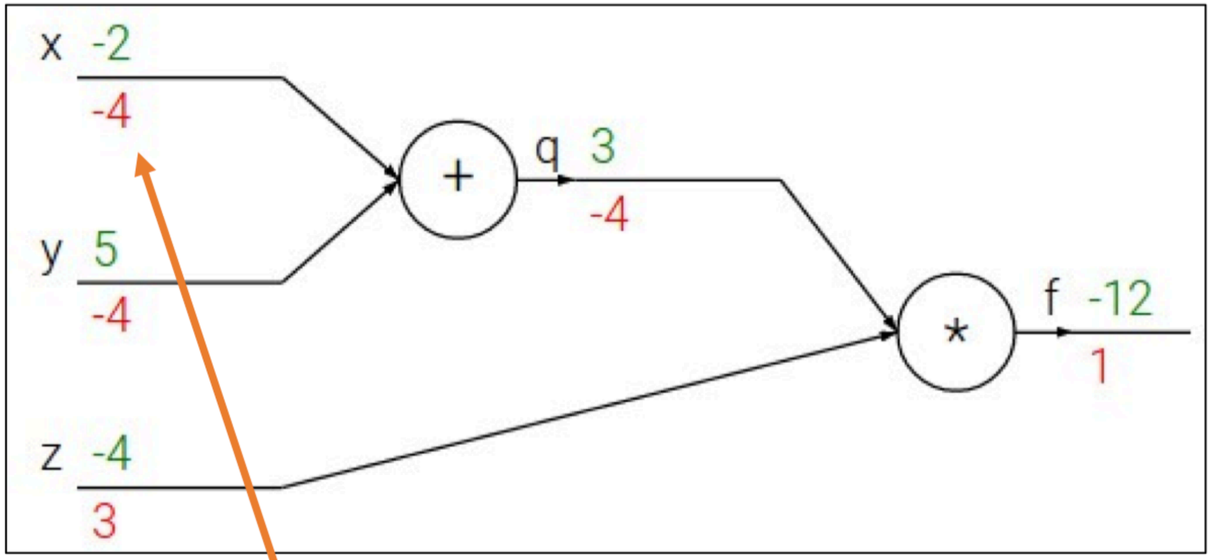
Downstream Gradient **Local Gradient** **Upstream Gradient**

Source: Johnson

Backpropagation: Simple Example

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

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



1. Forward pass: Compute outputs

$$q = x + y \qquad f = qz$$

2. Backward pass: Compute derivatives

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

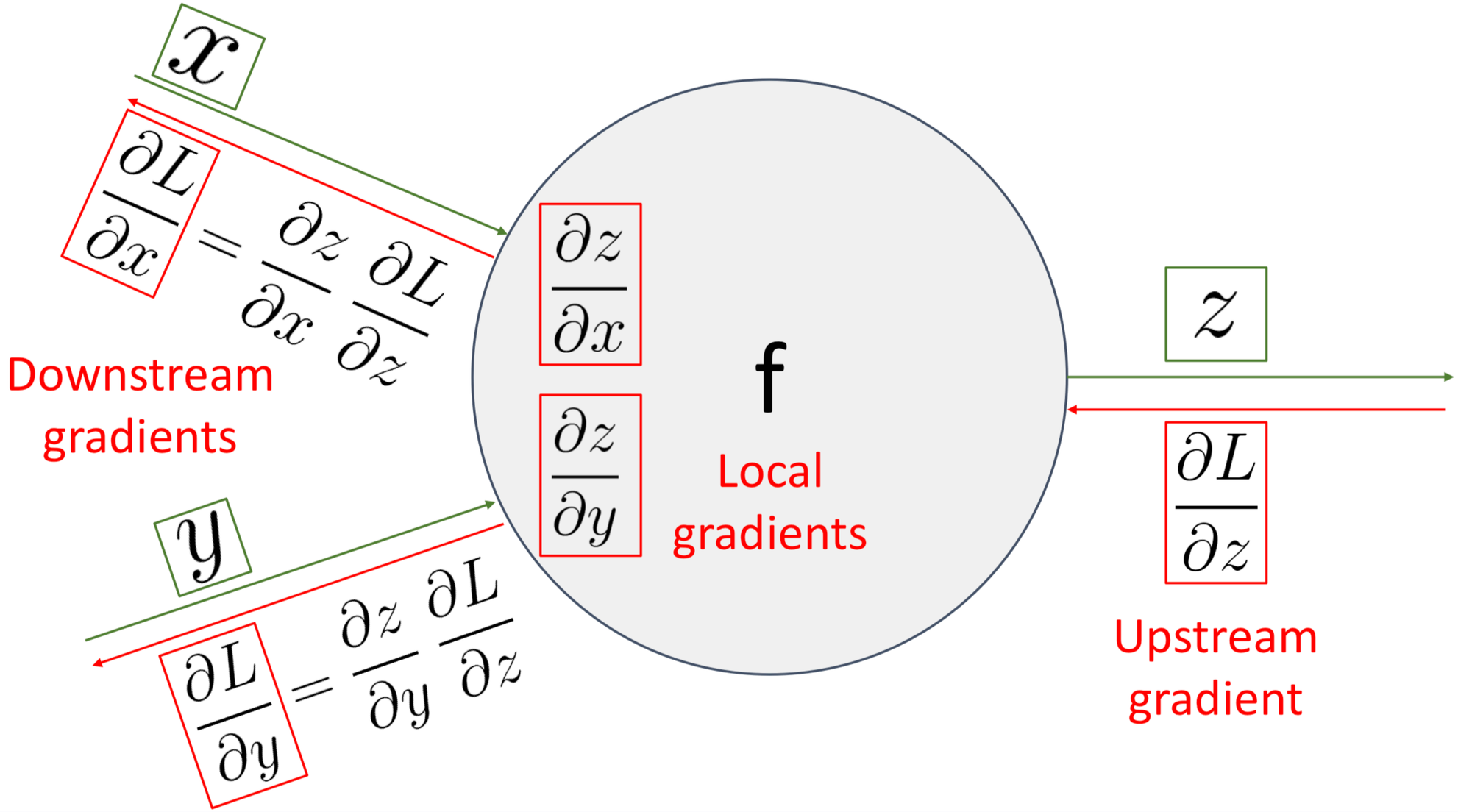
Chain Rule

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

$$\frac{\partial q}{\partial x} = 1$$

Downstream Gradient **Local Gradient** **Upstream Gradient**

Source: Johnson



Source: Johnson

Matrix calculus

- $\mathbf{x}$ column vector of size $[n \times 1]$:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

- We now define a function on vector $\mathbf{x}$: $\mathbf{y} = f(\mathbf{x})$
- If y is a scalar, then

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{pmatrix}$$

The derivative of y is a row vector of size $[1 \times n]$

- If $\mathbf{y}$ is a vector $[m \times 1]$, then (*Jacobian formulation*):

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

The derivative of y is a matrix of size $[m \times n]$

(m rows and n columns)

Source: Freeman, Torralba, Isola

Matrix calculus

- If y is a scalar and $\mathbf{X}$ is a matrix of size $[n \times m]$, then

$$\frac{\partial y}{\partial \mathbf{X}} = \begin{pmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{21}} & \cdots & \frac{\partial y}{\partial x_{n1}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y}{\partial x_{1m}} & \frac{\partial y}{\partial x_{2m}} & \cdots & \frac{\partial y}{\partial x_{nm}} \end{pmatrix}$$

The output is a matrix of size $[m \times n]$

Source: Freeman, Torralba, Isola

Matrix calculus

- Chain rule:

For the function: $h(\mathbf{x}) = f(g(\mathbf{x}))$

Its derivative is: $h'(\mathbf{x}) = f'(g(\mathbf{x}))g'(\mathbf{x})$

and writing $\mathbf{z} = f(\mathbf{u})$, and $\mathbf{u} = g(\mathbf{x})$:

$$\left.\frac{\partial \mathbf{z}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}} = \left.\frac{\partial \mathbf{z}}{\partial \mathbf{u}}\right|_{\mathbf{u}=g(\mathbf{a})} \cdot \left.\frac{\partial \mathbf{u}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}}$$

$[m \times n] \qquad [m \times p] \qquad [p \times n]$

with $p =$ length of vector $\mathbf{u} = |\mathbf{u}|$, $m = |\mathbf{z}|$, and $n = |\mathbf{x}|$

Example, if $|\mathbf{z}| = 1$, $|\mathbf{u}| = 2$, $|\mathbf{x}| = 4$

$h'(\mathbf{x}) =$  $=$

Source: Freeman, Torralba, Isola

Matrix calculus

- Chain rule:

For the function: $h(\mathbf{x}) = f^{(n)}(f^{(n-1)}(\dots f^{(1)}(\mathbf{x})))$

With $\mathbf{u}^{(1)} = f^{(1)}(\mathbf{x})$

$\mathbf{u}^{(i)} = f^{(i)}(\mathbf{u}^{(i-1)})$

$\mathbf{z} = \mathbf{u}^{(n)} = f^{(n)}(\mathbf{u}^{(n-1)})$

The derivative becomes a product of matrices:

$$\left.\frac{\partial \mathbf{z}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}} = \left.\frac{\partial \mathbf{z}}{\partial \mathbf{u}^{(n-1)}}\right|_{\mathbf{u}^{(n-1)}=f^{(n-1)}(\mathbf{u}^{n-2})} \cdot \left.\frac{\partial \mathbf{u}^{(n-1)}}{\partial \mathbf{u}^{(n-2)}}\right|_{\mathbf{u}^{(n-2)}=f^{(n-2)}(\mathbf{u}^{n-3})} \cdots \left.\frac{\partial \mathbf{u}^{(2)}}{\partial \mathbf{u}^{(1)}}\right|_{\mathbf{u}^{(1)}=f^{(1)}(\mathbf{a})} \cdot \left.\frac{\partial \mathbf{u}^{(1)}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}}$$

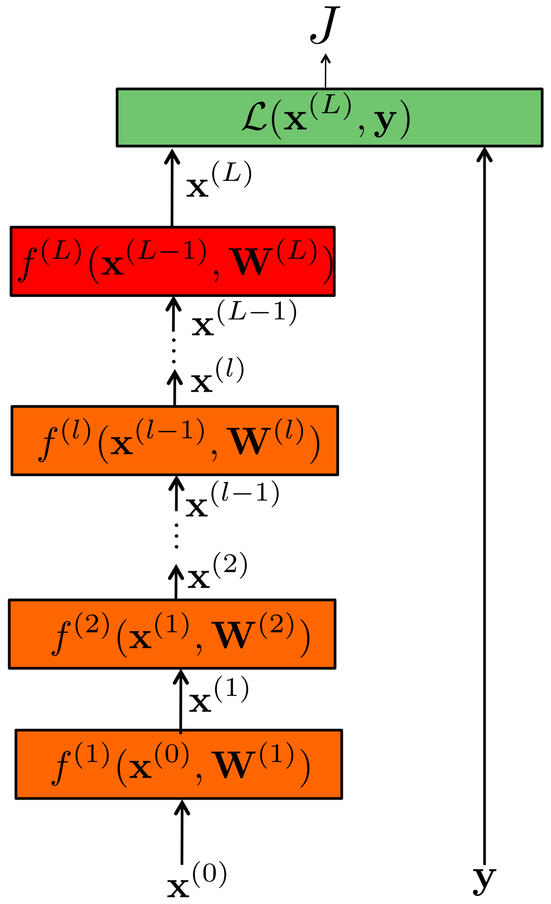
Source: Freeman, Torralba, Isola

Computing gradients

To compute the gradients, we could start by writing the full energy J as a function of the network parameters.

$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f^{(L)}(\ldots f^{(2)}(f^{(1)}(\mathbf{x}_i^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \ldots \mathbf{W}^{(L)}), \mathbf{y}_i)$$

And then compute the partial derivatives… instead, we can use the chain rule to derive a compact algorithm: **backpropagation**



Source: Freeman, Torralba, Isola

Computing gradients

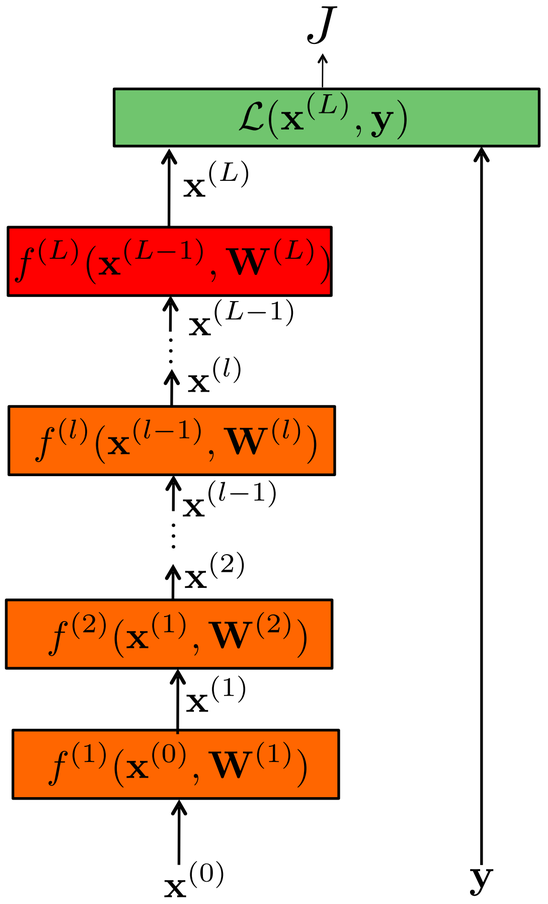
The loss J is the sum of the losses associated to each training example $\{\mathbf{x}_i^{(0)}, \mathbf{y}_i\}$

$$J(\mathbf{W}) = \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i; \mathbf{W})$$

Its gradient with respect to each of the network's parameters w is:

$$\frac{\partial J(\mathbf{W})}{\partial w} = \sum_{i=1}^{N} \frac{\partial \mathcal{L}(\mathbf{x}_i^{(L)}, \mathbf{y}_i; \mathbf{W})}{\partial w}$$

is how much J varies when the parameter w is varied.



Source: Freeman, Torralba, Isola

Computing gradients

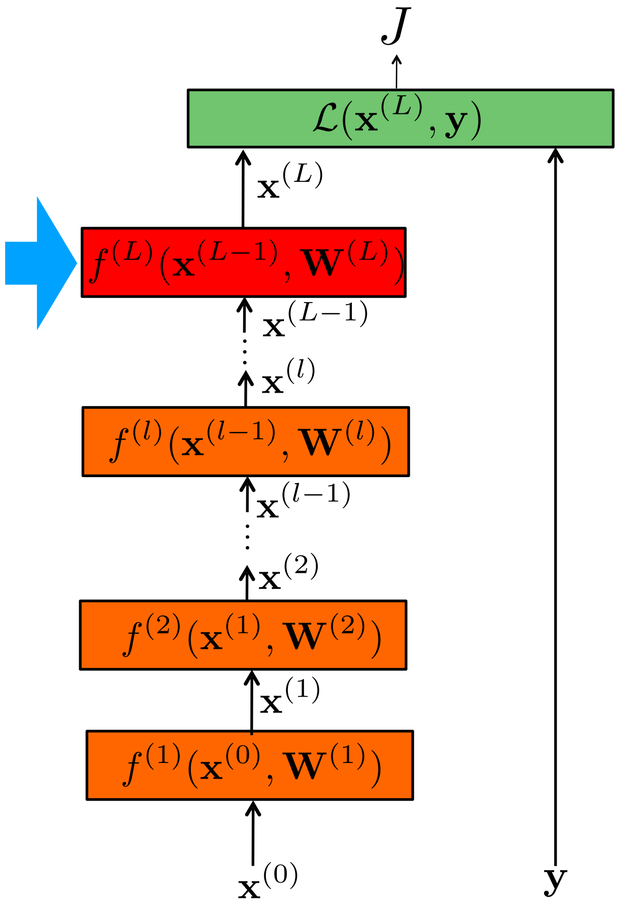
We could write the loss function to get the gradients as:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}; \mathbf{W}) = \mathcal{L}(f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)}), \mathbf{y})$$

If we compute the gradient with respect to the parameters of the last layer (output layer) $W^{(L)}$, using the **chain rule**:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)})}{\partial \mathbf{W}^{(L)}}$$

How much the loss changes when we change $W^{(L)}$?
The change is the product between how much the loss changes when we change the output of the last layer and how much the output changes when we change the layer parameters.



Source: Freeman, Torralba, Isola

Computing gradients: cost layer

If we compute the gradient with respect to the parameters of the last layer (output layer) $W^{(L)}$, using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial f^{(L)}(\mathbf{x}^{(L-1)}, \mathbf{W}^{(L)})}{\partial \mathbf{W}^{(L)}}$$

For example, for an Euclidean loss:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}) = \frac{1}{2} \left\| \mathbf{x}^{(L)} - \mathbf{y} \right\|_2^2$$

The gradient is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} = \mathbf{x}^{(L)} - \mathbf{y}$$

Will depend on the layer structure and non-linearity.

Source: Freeman, Torralba, Isola

Computing gradients: layer l

We could write the full loss function to get the gradients:

$$\mathcal{L}(\mathbf{x}^{(L)}, \mathbf{y}; \mathbf{W}) = \mathcal{L}(f^{(L)}(\dots f^{(2)}(f^{(1)}(\mathbf{x}^{(0)}, \mathbf{W}^{(1)}), \mathbf{W}^{(2)}), \dots \mathbf{W}^{(L)}), \mathbf{y})$$

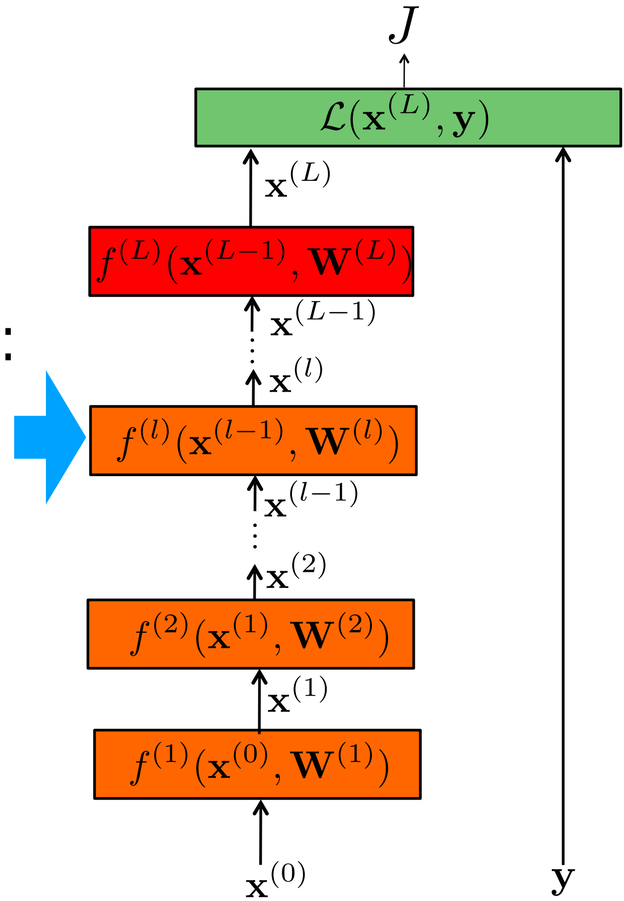
If we compute the gradient with respect to $\mathbf{W}^{(l)}$, using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \cdot \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \cdot \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{x}^{(L-2)}} \cdots \frac{\partial \mathbf{x}^{(l+1)}}{\partial \mathbf{x}^{(l)}}}_{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{W}^{(l)}}$$

$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

And this can be computed iteratively!

This is easy.



Source: Freeman, Torralba, Isola

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l+1)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}}$$

And this can be computed iteratively. We start at the top (L) and we can compute the gradient at layer l-1

If we have the value of $\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}}$ we can compute the gradient at the layer below as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbf{x}^{(l-1)}}$$

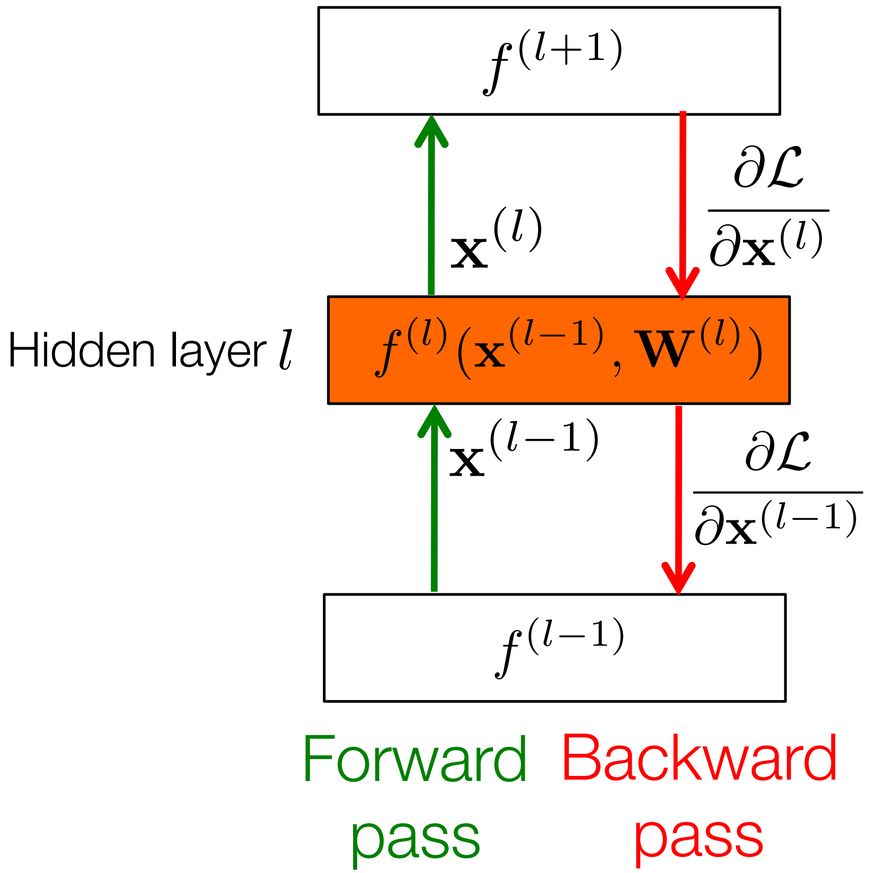
Gradient layer l-1

Gradient layer l

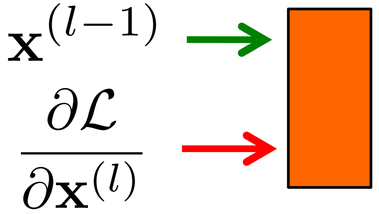
$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

Source: Freeman, Torralba, Isola

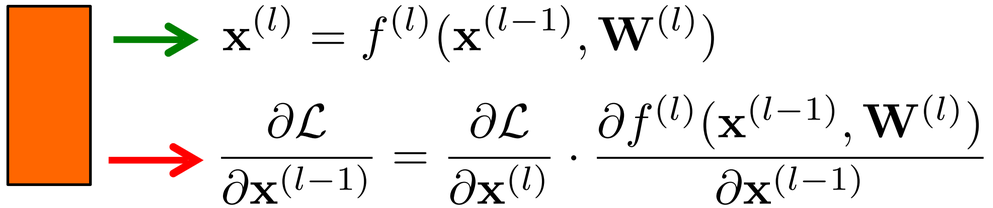
Backpropagation — Goal: to update parameters of layer l



- Layer l has two inputs (during training)



- We compute the outputs



- To compute the output, we need:

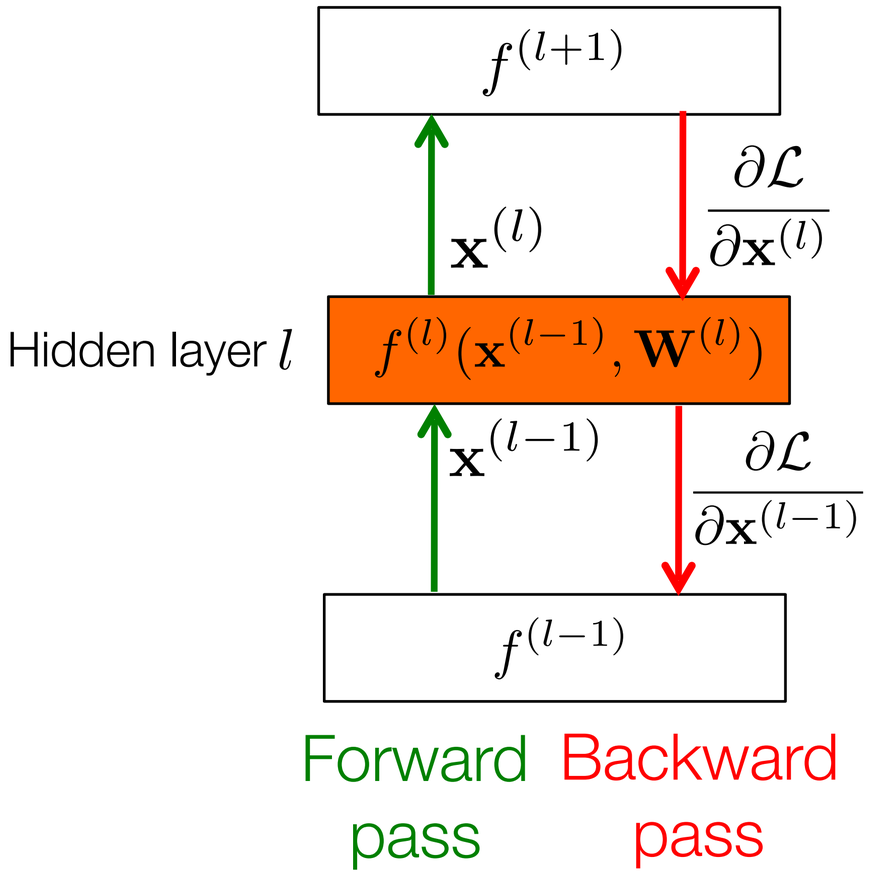
$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- To compute the weight update, we need:

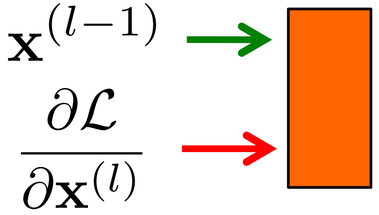
$$\frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

Source: Freeman, Torralba, Isola

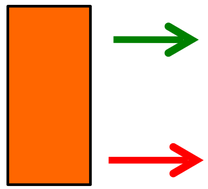
Backpropagation — Goal: to update parameters of layer l



- Layer l has two inputs (during training)



- We compute the outputs



$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

- The weight update equation is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \left(\frac{\partial J}{\partial \mathbf{W}^{(l)}} \right)^T$$

(sum over all training examples to get J)

Source: Freeman, Torralba, Isola

Backpropagation Summary

- Forward pass: for each training example, compute the outputs for all layers:

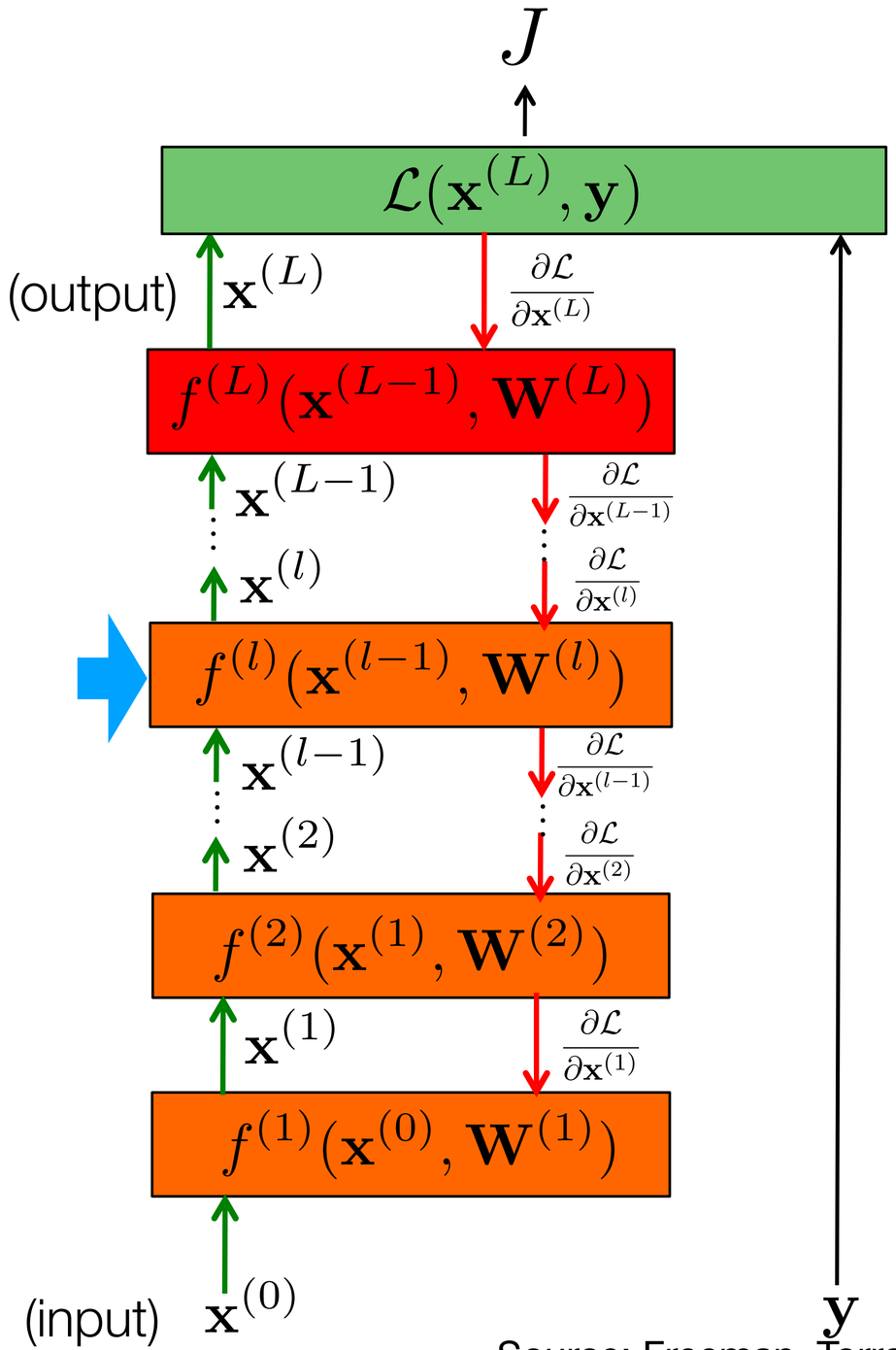
$$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})$$

- Backwards pass: compute loss derivatives iteratively from top to bottom:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{x}^{(l-1)}}$$

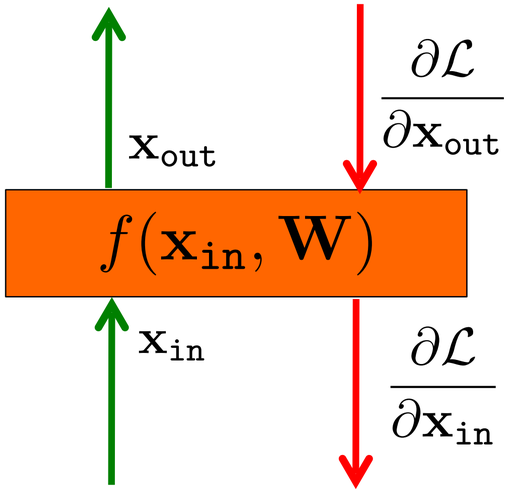
- Compute gradients w.r.t. weights, and update weights:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(l)}} \cdot \frac{\partial f^{(l)}(\mathbf{x}^{(l-1)}, \mathbf{W}^{(l)})}{\partial \mathbf{W}^{(l)}}$$



Source: Freeman, Torralba, Isola

Linear Module



- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W}\mathbf{x}_{\text{in}}$

With W being a matrix of size $|x_{out}| \times |x_{in}|$

- Backprop to input:

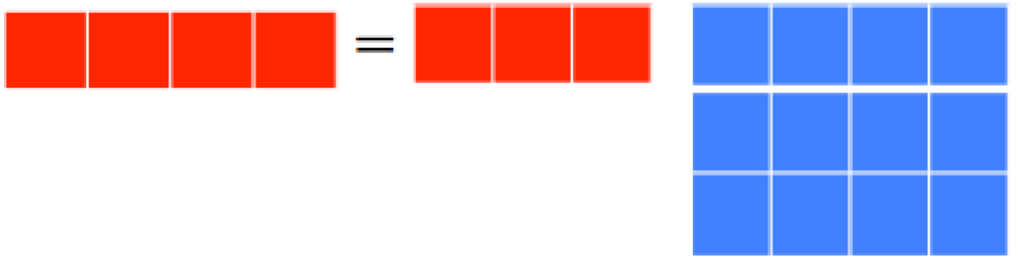
$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{x}_{\text{in}}}$$

If we look at the j component of output x_{out}, with respect to the i component of the input, x_{in}:

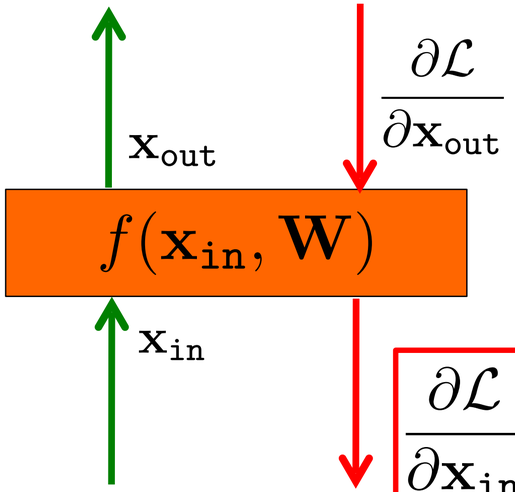
$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{x}_{\text{in}_j}} = \mathbf{W}_{ij} \longrightarrow \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{W}$$

Therefore:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \mathbf{W}}$$



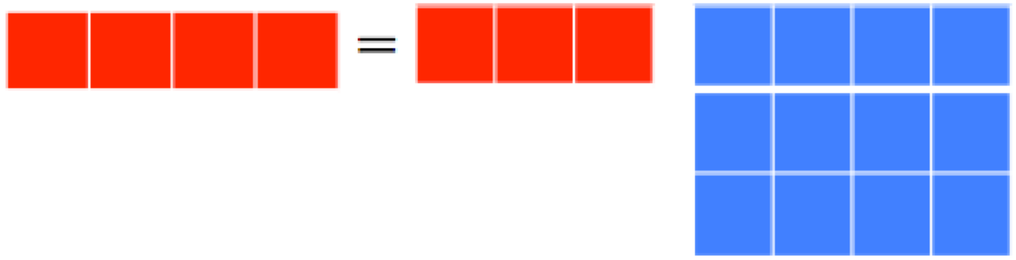
Source: Freeman, Torralba, Isola



Linear Module

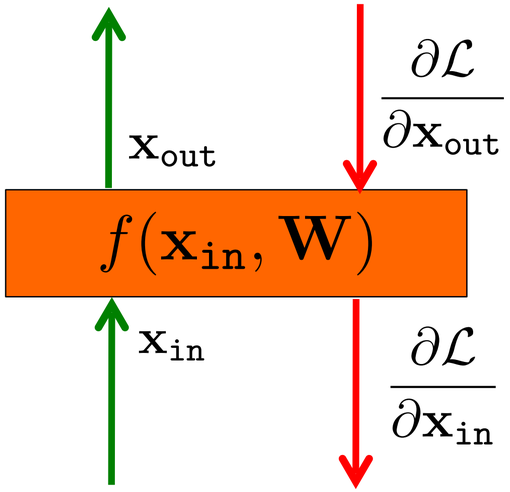
- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W}\mathbf{x}_{\text{in}}$
- Backprop to input:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \mathbf{W}$$



Now let's see how we use the set of outputs to compute the weights update equation (backprop to the weights).

Source: Freeman, Torralba, Isola



Linear Module

- Forward propagation: $\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \mathbf{W}) = \mathbf{W}\mathbf{x}_{\text{in}}$
- Backprop to weights:

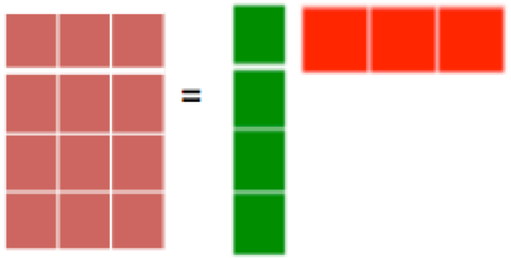
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{W}}$$

If we look at how the parameter W_{ij} changes the cost, only the i component of the output will change, therefore:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ij}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \mathbf{x}_{\text{in}_j}$$

$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \mathbf{x}_{\text{in}_j}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}}}$$



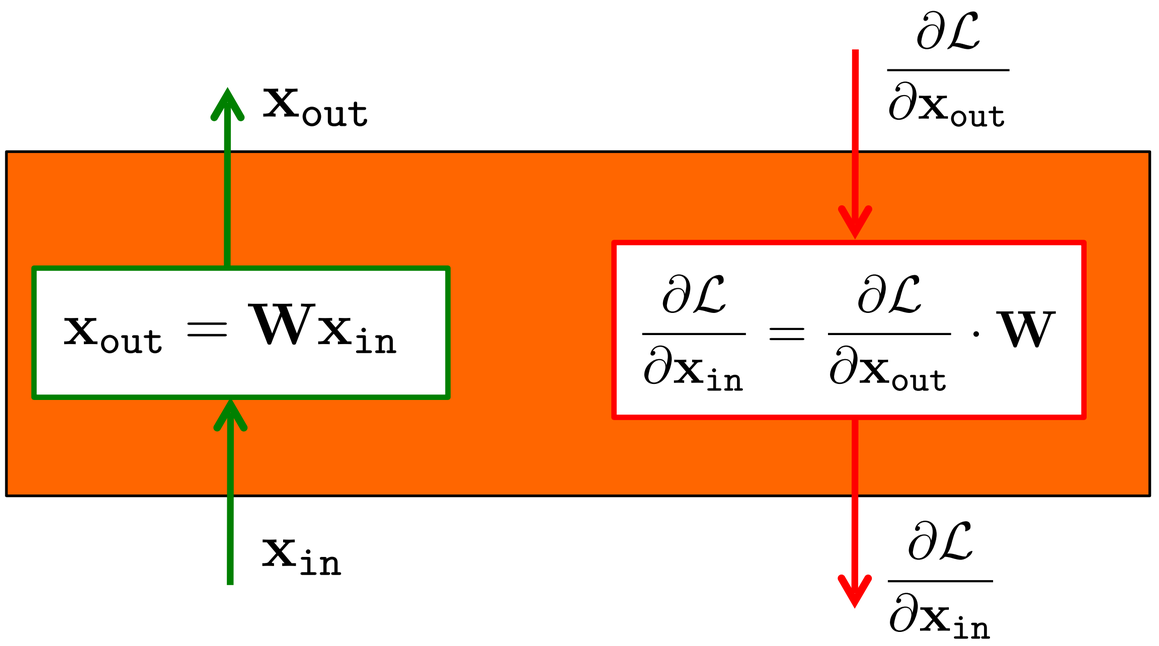
And now we can update the weights (by summing over all the training examples):

$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} + \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^{T}$$

(sum over all training examples to get J)

Source: Freeman, Torralba, Isola

Linear Module



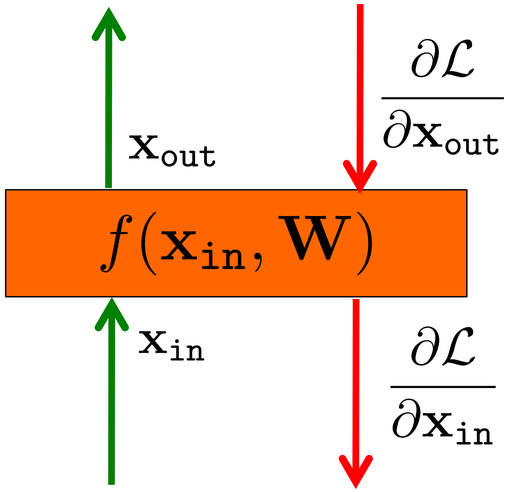
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}}$$

$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} + \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^{T}$$

$$J = \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i, \mathbf{y}_i) \longrightarrow \frac{\partial J}{\partial \mathbf{W}} = \sum_{i=1}^{N} \mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \Big|_{\mathbf{x}_i, \mathbf{y}_i}$$

Sum over N training pairs

Source: Freeman, Torralba, Isola



Pointwise function

- Forward propagation:

$$\mathbf{x}_{\text{out}_i} = h(\mathbf{x}_{\text{in}_i} + b_i)$$

h = an arbitrary function, b_i is a bias term.

- Backprop to input: $\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}_i}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{x}_{\text{in}_i}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot h'(\mathbf{x}_{\text{in}_i} + b_i)$

- Backprop to bias: $\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial \mathbf{x}_{\text{out}_i}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot h'(\mathbf{x}_{\text{in}_i} + b_i)$

We use this last expression to update the bias.

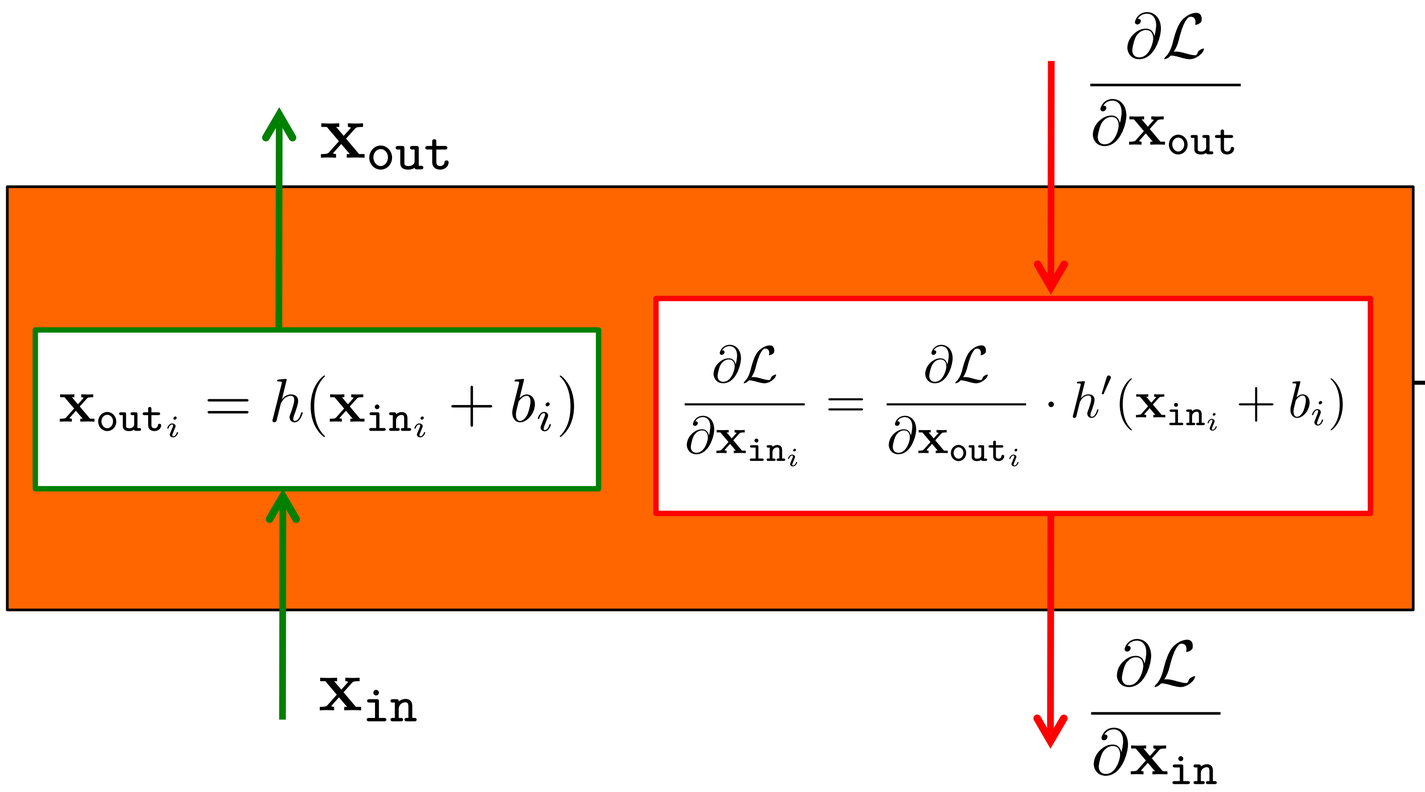
Some useful derivatives:

For hyperbolic tangent: $\tanh'(x) = 1 - \tanh^2(x)$

For ReLU: $h(x) = \max(0, x), \quad h'(x) = \mathbb{1}(x \geq 0)$

Source: Freeman, Torralba, Isola

Pointwise function



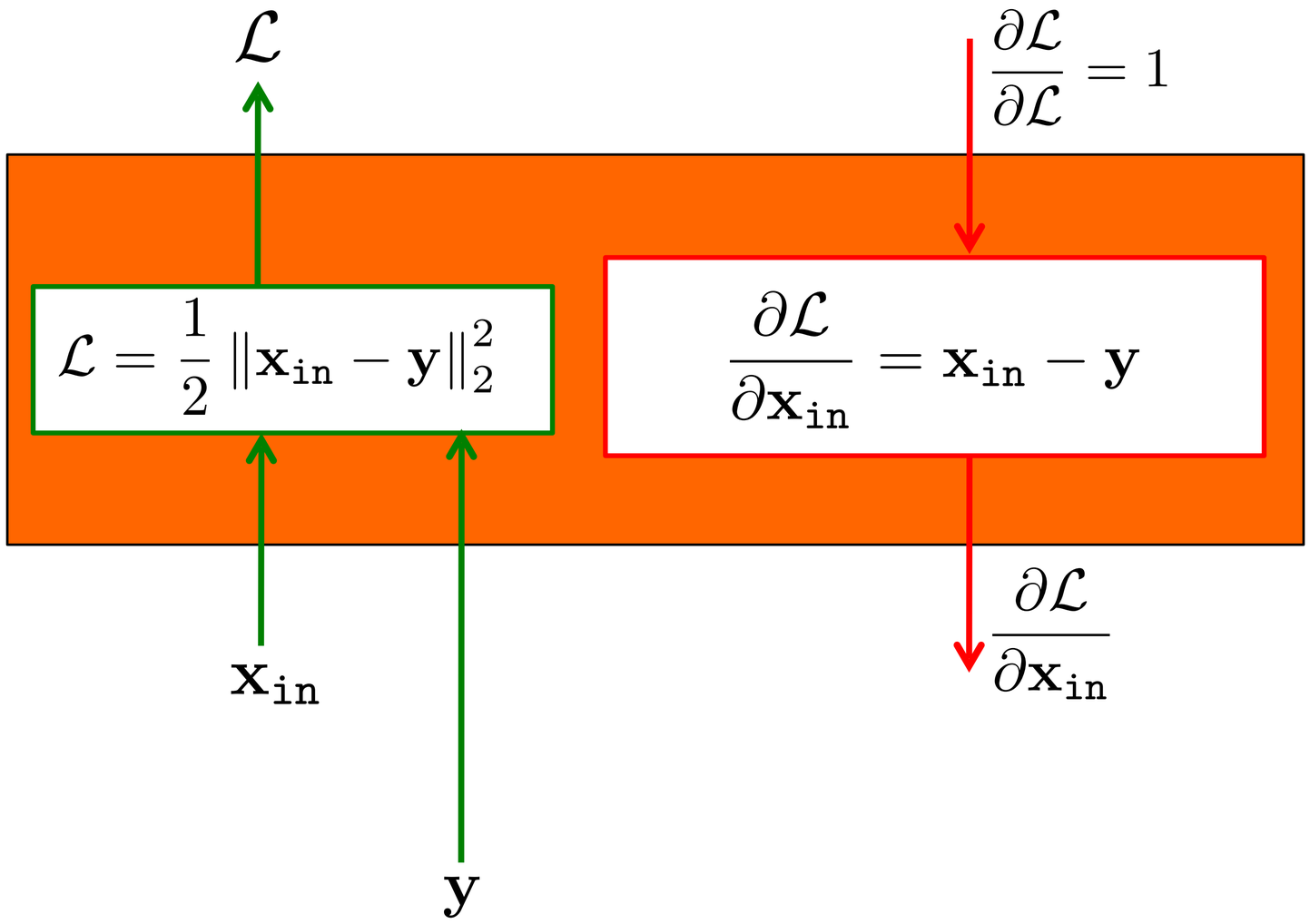
$$\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\mathtt{out}_i}} \cdot h'(\mathbf{x}_{\mathtt{in}_i} + b_i)$$

Bias updates:

$$b_i^{k+1} \leftarrow b_i^k + \eta \frac{\partial J}{\partial b_i}$$

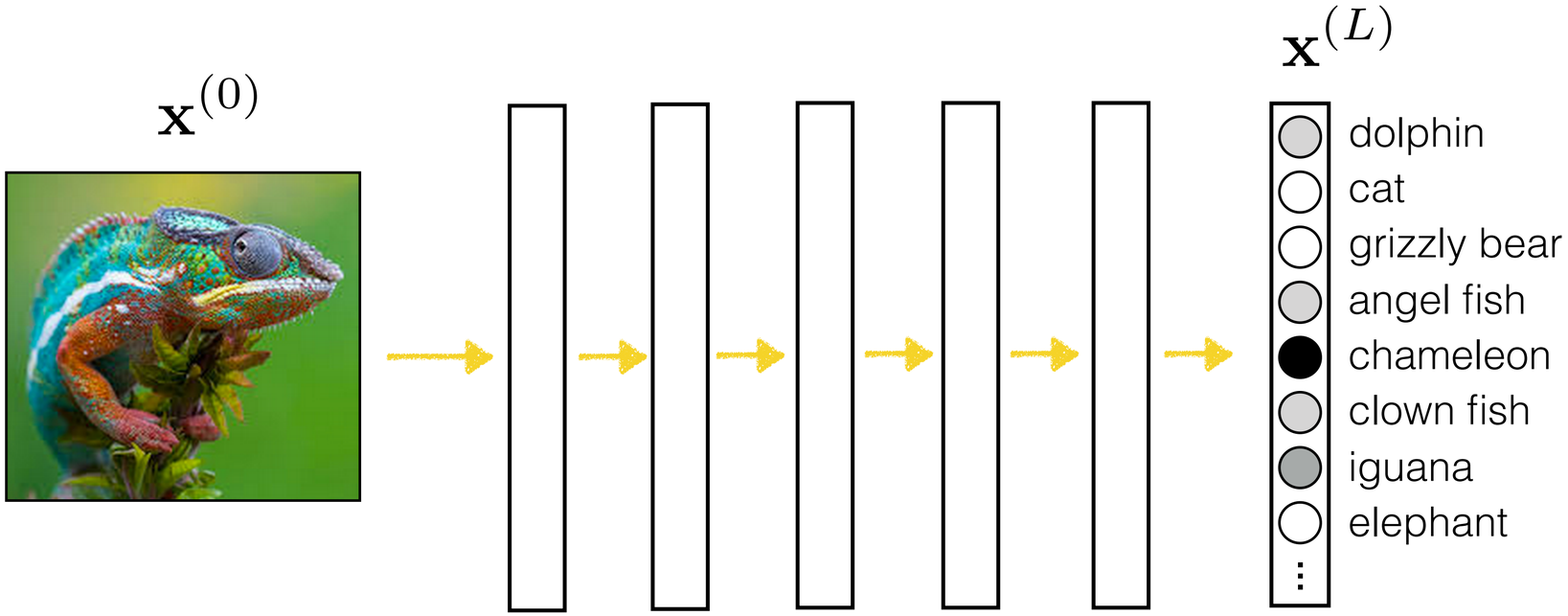
Source: Freeman, Torralba, Isola

Euclidean cost module



Source: Freeman, Torralba, Isola

Unit visualization via backprop



$$\frac{\partial x_j^{(L)}}{\partial \mathbf{x}^{(0)}} = \frac{\partial x_j^{(L)}}{\partial \mathbf{x}^{(L-1)}} \cdot \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{x}^{(L-2)}} \cdots \frac{\partial \mathbf{x}^{(2)}}{\partial \mathbf{x}^{(1)}} \cdot \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{x}^{(0)}}$$

How much the “chameleon” score is increased or decreased by changing the image pixels.

Source: Freeman, Torralba, Isola

Unit visualization via backprop

$$\arg\max_{\mathbf{x}^{(0)}} x_j^{(L)} + \lambda R(\mathbf{x}^{(0)})$$

$$\mathbf{x}^{(0)^{k+1}} \leftarrow \mathbf{x}^{(0)^k} + \eta \frac{\partial(x_j^{(L)} + \lambda R(\mathbf{x}^{(0)}))}{\partial \mathbf{x}^{(0)}}$$

Make an image that maximizes the “cat” output neuron:

[https://distill.pub/2017/feature-visualization/]

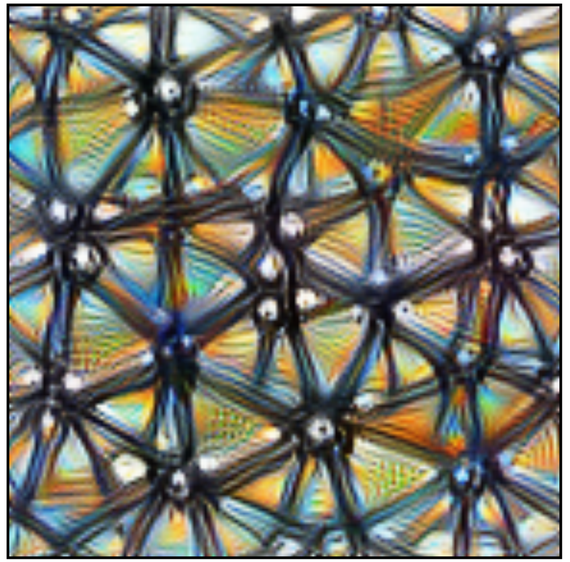
Source: Freeman, Torralba, Isola

Unit visualization via backprop

$$\arg\max_{\mathbf{x}^{(0)}} x_j^{(L)} + \lambda R(\mathbf{x}^{(0)})$$

$$\mathbf{x}^{(0)^{k+1}} \leftarrow \mathbf{x}^{(0)^k} + \eta \frac{\partial(x_j^{(L)} + \lambda R(\mathbf{x}^{(0)}))}{\partial \mathbf{x}^{(0)}}$$

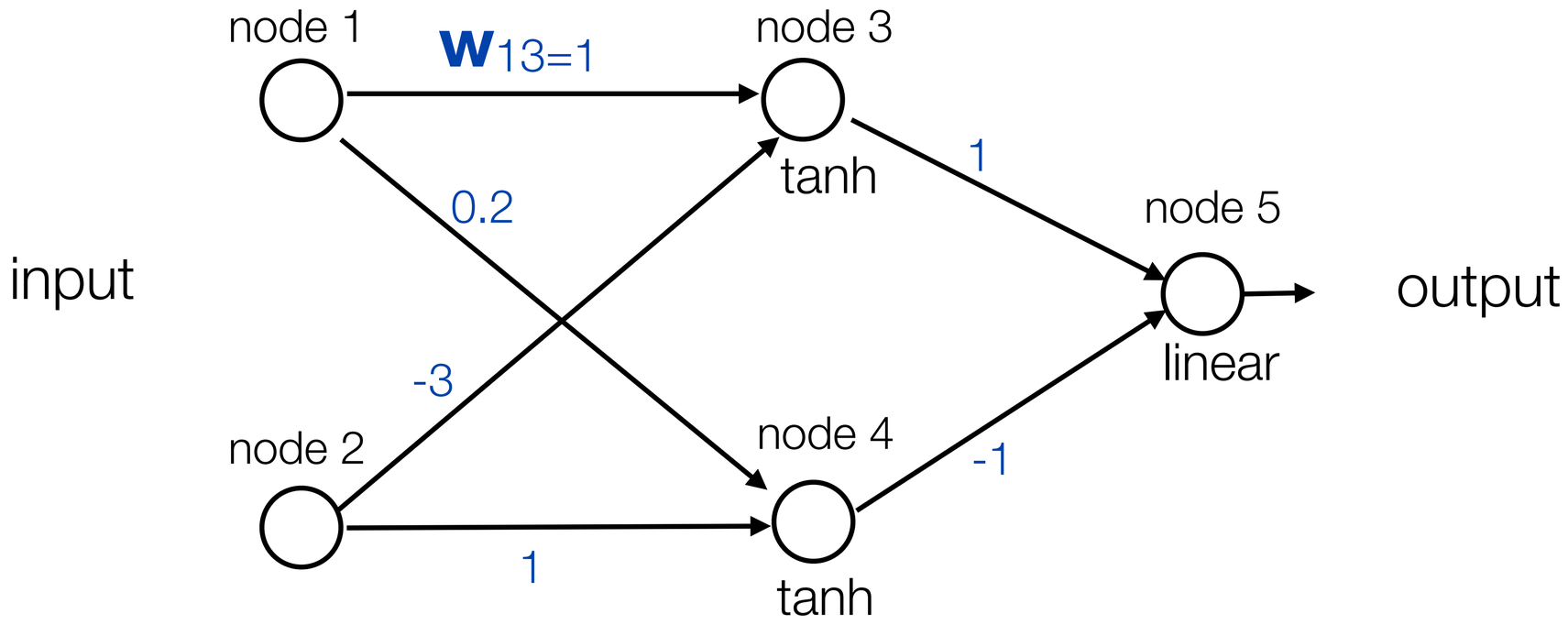
Make an image that maximizes the value of a random neuron in the middle of the network:



[https://distill.pub/2017/feature-visualization/]

Source: Freeman, Torralba, Isola

Backpropagation example



Learning rate η = 0.2 (because we used positive increments)

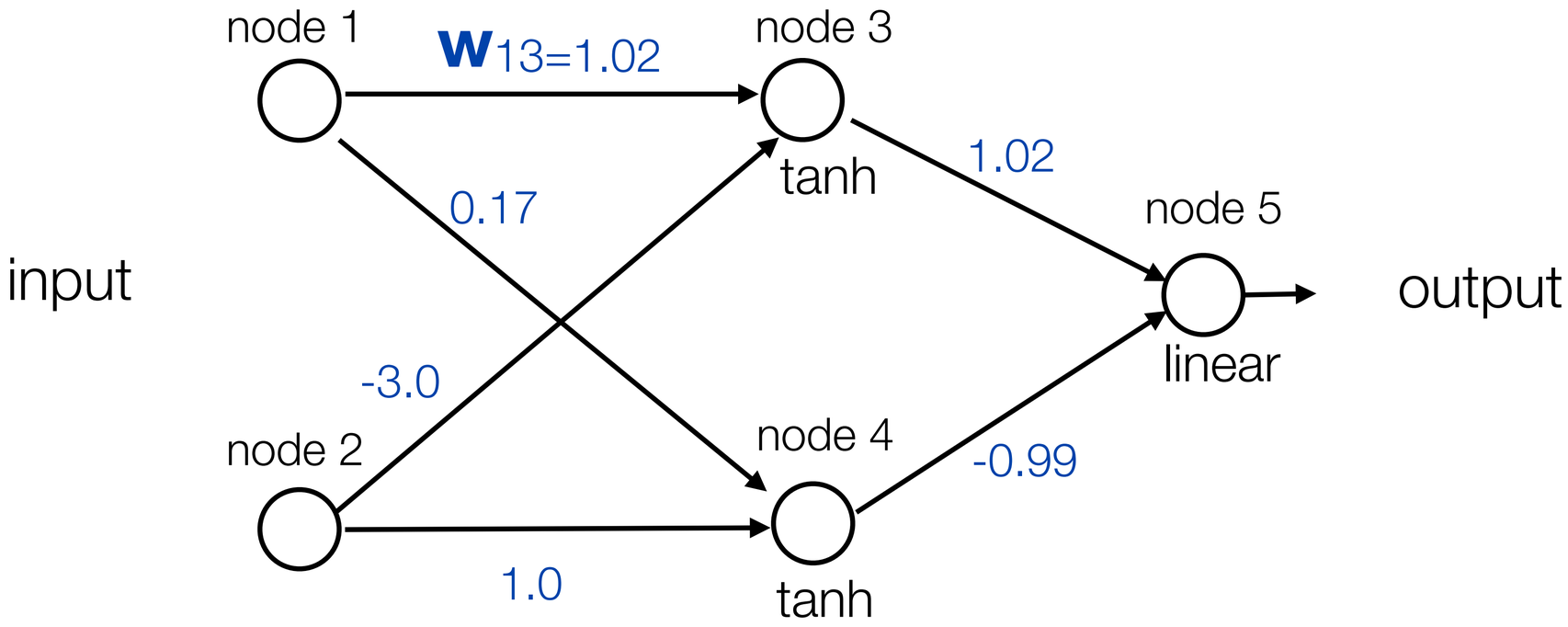
Euclidean loss

Training data:	input		desired output
	node 1	node 2	node 5
	1.0	0.1	0.5

Exercise: run one iteration of back propagation

Source: Freeman, Torralba, Isola

Backpropagation example



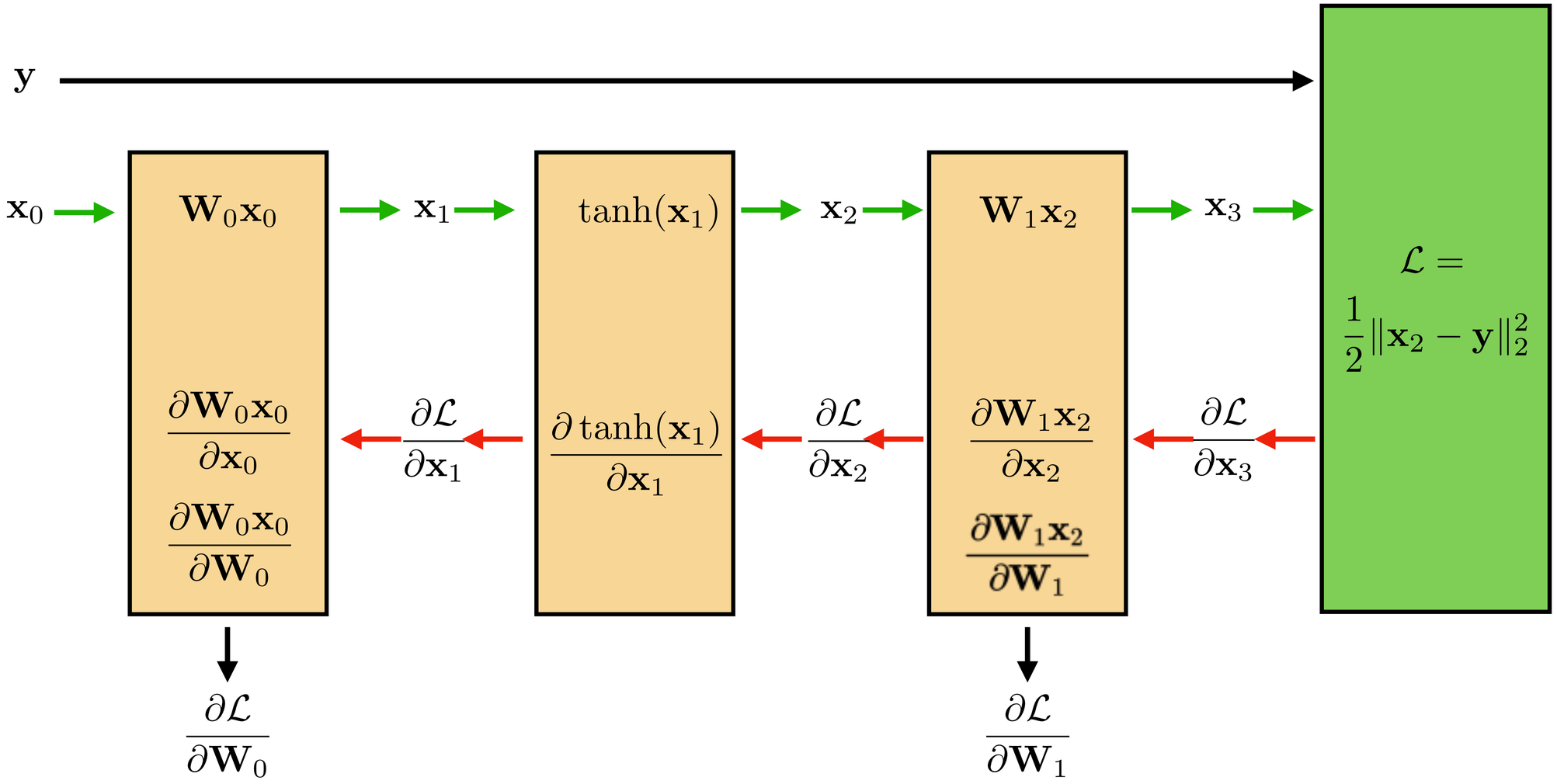
After one iteration (rounding to two digits)

Source: Freeman, Torralba, Isola

Step by step solution

Source: Freeman, Torralba, Isola

First, let's rewrite the network using the modular block notation:



We need to compute all these terms simply so we can find the weight updates at the bottom.

Source: Freeman, Torralba, Isola

Our goal is to perform the following two updates:

$$\mathbf{W}_0^{k+1} = \mathbf{W}_0^k + \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_0} \right)^T$$

$$\mathbf{W}_1^{k+1} = \mathbf{W}_1^k + \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} \right)^T$$

where W^k are the weights at some iteration k of gradient descent given by the first slide:

$$\mathbf{W}_0^k = \begin{pmatrix} 1 & -3 \\ 0.2 & 1 \end{pmatrix} \qquad \mathbf{W}_1^k = \begin{pmatrix} 1 & -1 \end{pmatrix}$$

Source: Freeman, Torralba, Isola

First we compute the derivative of the loss with respect to the output:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_3} = \mathbf{x}_3 - \mathbf{y}$$

Now, by the chain rule, we can derive equations, working *backwards*, for each remaining term we need:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_3}\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_3}\mathbf{W}_1$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_2}\frac{\partial \tanh(\mathbf{x}_1)}{\partial \mathbf{x}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_2}(1 - \tanh^2(\mathbf{x}_1))$$

ending up with our two gradients needed for the weight update:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_0} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \mathbf{W}_0} = \mathbf{x}_0\frac{\partial \mathcal{L}}{\partial \mathbf{x}_1}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_3}\frac{\partial \mathbf{x}_3}{\partial \mathbf{W}_1} = \mathbf{x}_2\frac{\partial \mathcal{L}}{\partial \mathbf{x}_3}$$

Source: Freeman, Torralba, Isola

The values for input vector x_0 and target y are also given by the first slide:

$$\mathbf{x}_0 = \begin{pmatrix} 1.0 \\ 0.1 \end{pmatrix} \qquad \mathbf{y} = 0.5$$

Finally, we simply plug these values into our equations and compute the numerical updates:

Forward pass:

$$\mathbf{x}_1 = \mathbf{W}_0\mathbf{x}_0 = \begin{pmatrix} 1 & -3 \\ 0.2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.3 \end{pmatrix}$$

$$\mathbf{x}_2 = \tanh(\mathbf{x}_1) = \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix}$$

$$\mathbf{x}_3 = \mathbf{W}_1\mathbf{x}_2 = \begin{pmatrix} 1 & -1 \end{pmatrix} \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix} = 0.313$$

$$\mathcal{L} = \frac{1}{2}(\mathbf{x}_3 - \mathbf{y})^2 = 0.017$$

Source: Freeman, Torralba, Isola

Backward pass:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_3} = \mathbf{x}_3 - \mathbf{y} = -0.1869$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_2} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_3}\mathbf{W}_1 = -0.1869 \begin{pmatrix} 1 & -1 \end{pmatrix} = \begin{pmatrix} -0.1869 & 0.1869 \end{pmatrix}$$

diagonal matrix because tanh is a pointwise operation

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_2}(1 - \tanh^2(\mathbf{x}_1)) = \begin{pmatrix} -0.1869 & 0.1869 \end{pmatrix} \begin{pmatrix} 1 - \tanh^2(0.7) & 0 \\ 0 & 1 - \tanh^2(0.3) \end{pmatrix} = \begin{pmatrix} -0.1186 & 0.171 \end{pmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_0} = \mathbf{x}_0 \frac{\partial \mathcal{L}}{\partial \mathbf{x}_1} = \begin{pmatrix} 1.0 \\ 0.1 \end{pmatrix} \begin{pmatrix} -0.1186 & 0.171 \end{pmatrix} = \begin{pmatrix} -0.1186 & 0.171 \\ -0.01186 & 0.0171 \end{pmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} = \mathbf{x}_2 \frac{\partial \mathcal{L}}{\partial \mathbf{x}_3} = \begin{pmatrix} 0.604 \\ 0.291 \end{pmatrix} \begin{pmatrix} -0.1186 \end{pmatrix} = \begin{pmatrix} -0.113 \\ -0.054 \end{pmatrix}$$

Source: Freeman, Torralba, Isola

Gradient updates:

$$\mathbf{W}_0^{k+1} = \mathbf{W}_0^k + \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_0} \right)^T$$
$$= \begin{pmatrix} 1 & -3 \\ 0.2 & 1 \end{pmatrix} - 0.2 \begin{pmatrix} -0.1186 & 0.171 \\ -0.01186 & 0.0171 \end{pmatrix}$$
$$= \begin{pmatrix} 1.02 & -3.0 \\ 0.17 & 1.0 \end{pmatrix}$$

$$\mathbf{W}_1^{k+1} = \mathbf{W}_1^k + \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1} \right)^T$$
$$= \begin{pmatrix} 1 & -1 \end{pmatrix} - 0.2 \begin{pmatrix} -0.113 & -0.054 \end{pmatrix}$$
$$= \begin{pmatrix} 1.02 & -0.989 \end{pmatrix}$$

Source: Freeman, Torralba, Isola

Automatic Differentiation

- An autodiff engine automatically builds the computation graph and applies the backpropagation
- Examples: TensorFlow, PyTorch, Theano, Autograd, etc

- Backpropagation: math operation
- Autodiff: software implementation

Source: Grosse, Ba

Autodiff is not finite difference

- Finite difference

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}$$

Useful for checking correctness of analytical expressions.

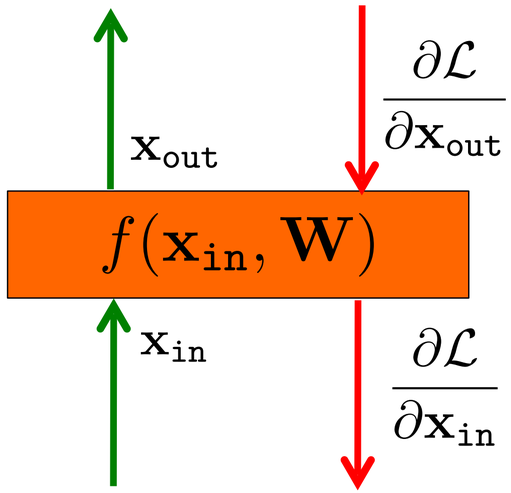
Issues:

- not efficient
- numerical error

Source: Grosse, Ba

What Autodiff Is

An autodiff system will convert the program into a sequence of *primitive operations (ops)* which have specified routines for computing derivatives.



$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \boxed{\frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \boxed{\frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{W}}}$$

Implemented by Autodiff package

Example: Autograd

```python
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

very sneaky!

```python
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print  "Trained loss:", training_loss(weights)
```

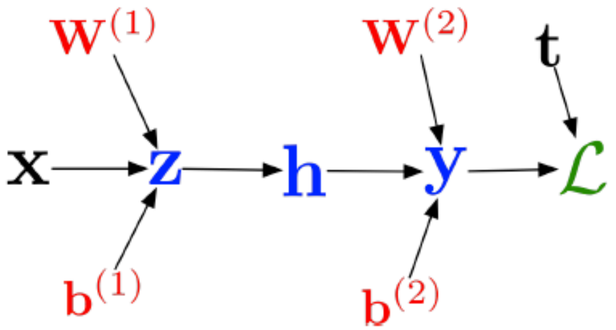
Autograd constructs a function for computing derivatives

Auuograd: https://github.com/HIPS/autograd

Source: Grosse, Ba

Building the Computation Graph

Autograd builds the computation graph by tracing the forward pass computation, allowing for an interface nearly indistinguishable from NumPy.

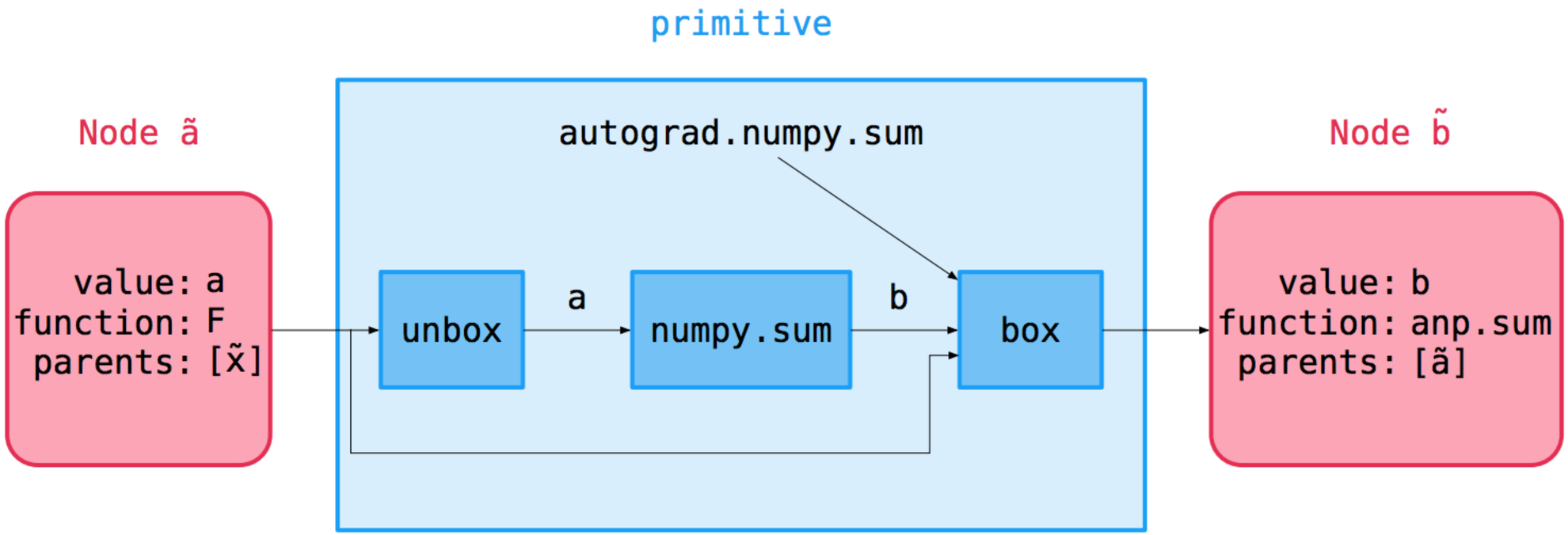


The Node class (defined in tracer.py) represents a node of the computation graph. It has attributes:

- `value`, the actual value computed on a particular set of inputs
- `fun`, the primitive operation defining the node
- `args, kwargs`, the arguments the op was called with
- `parents`, the parent Nodes

Source: Grosse, Ba

Building the Computation Graph



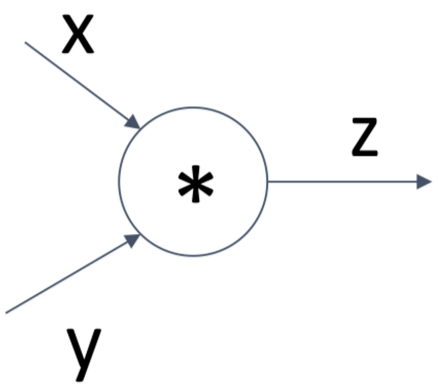
Source: Grosse, Ba

Backprop Implementation: Modular API

```
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
```

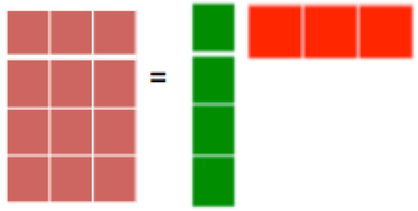
Source: Johnson

Example: PyTorch Autograd Functions



(x,y,z are scalars)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}}$$




```
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

Source: Johnson

Current State-of-the-Arts

- JAX
- PyTorch
- TensorFlow

https://github.com/google/jax

https://github.com/pytorch/pytorch

https://github.com/tensorflow/tensorflow