

Lecture 11

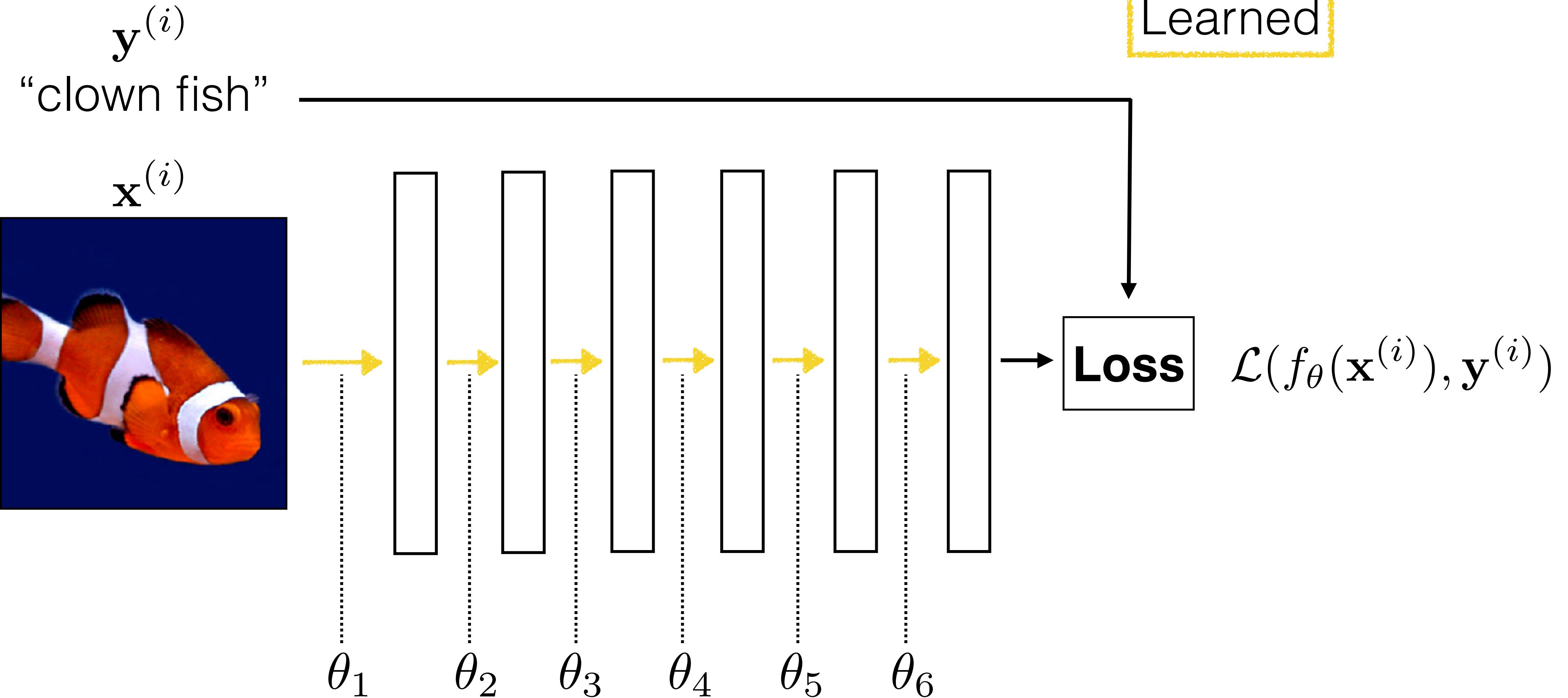
Backpropagation



11. Backpropagation

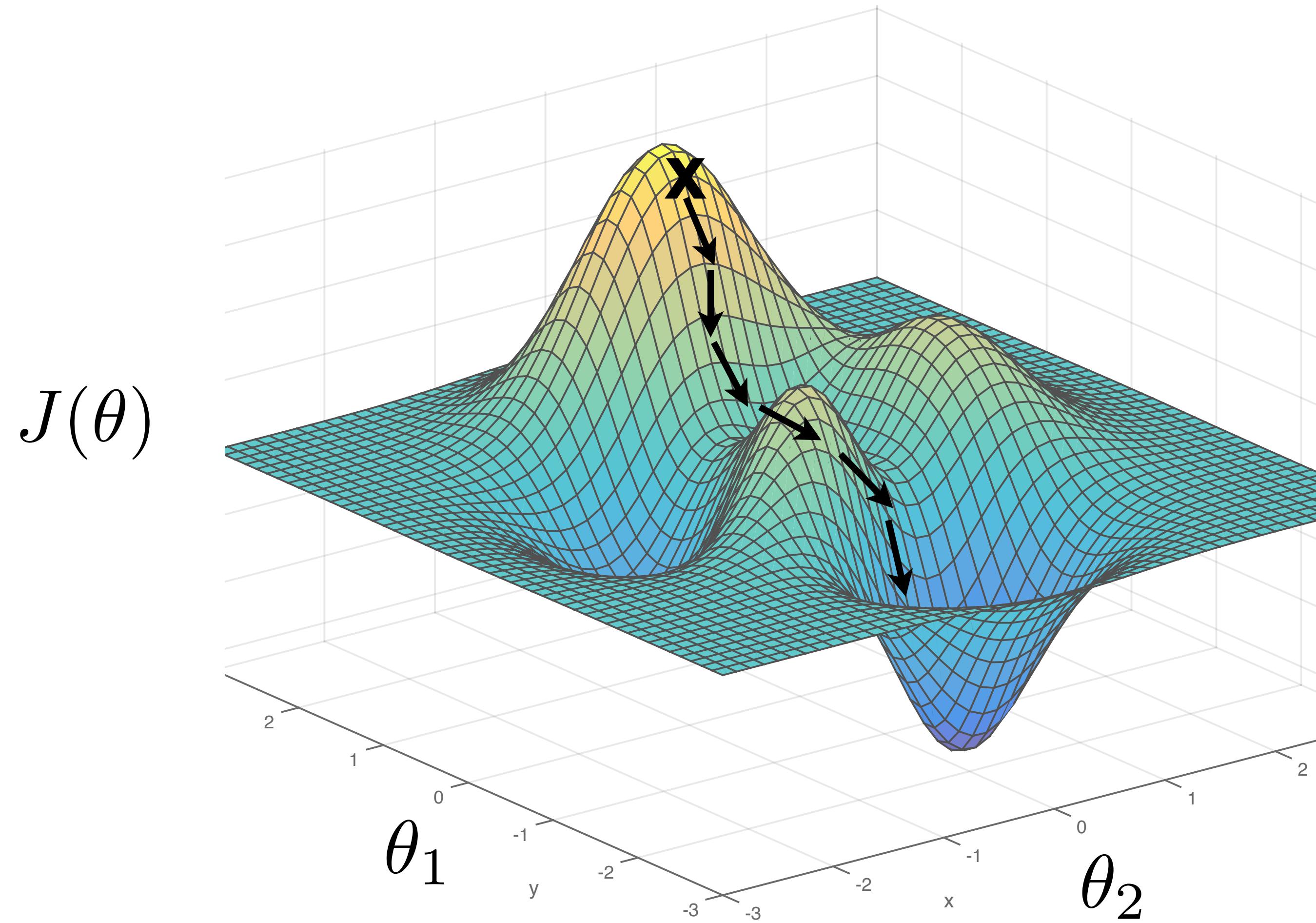
- Review of gradient descent
- Forward propagation
- Computing gradients
- Backward propagation
- Computation graphs and differentiable programming

Deep learning



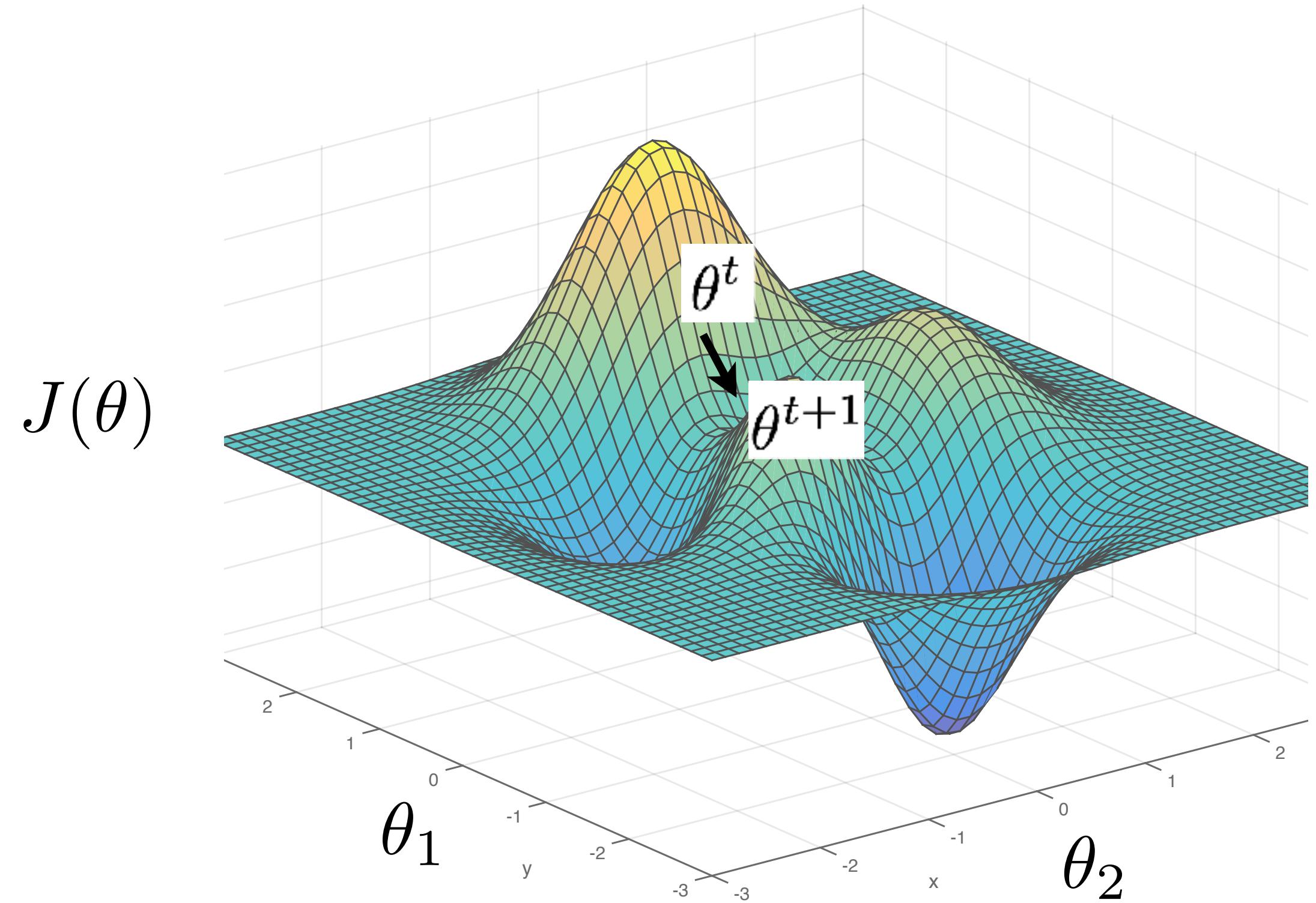
$$\theta^* = \arg \min_{\theta} \sum_{i=1}^N \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

Gradient descent



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

Gradient descent



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

$\underbrace{\hspace{10em}}_{J(\theta)}$

One iteration of gradient descent:

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

↓
learning rate

Stochastic gradient descent (SGD)

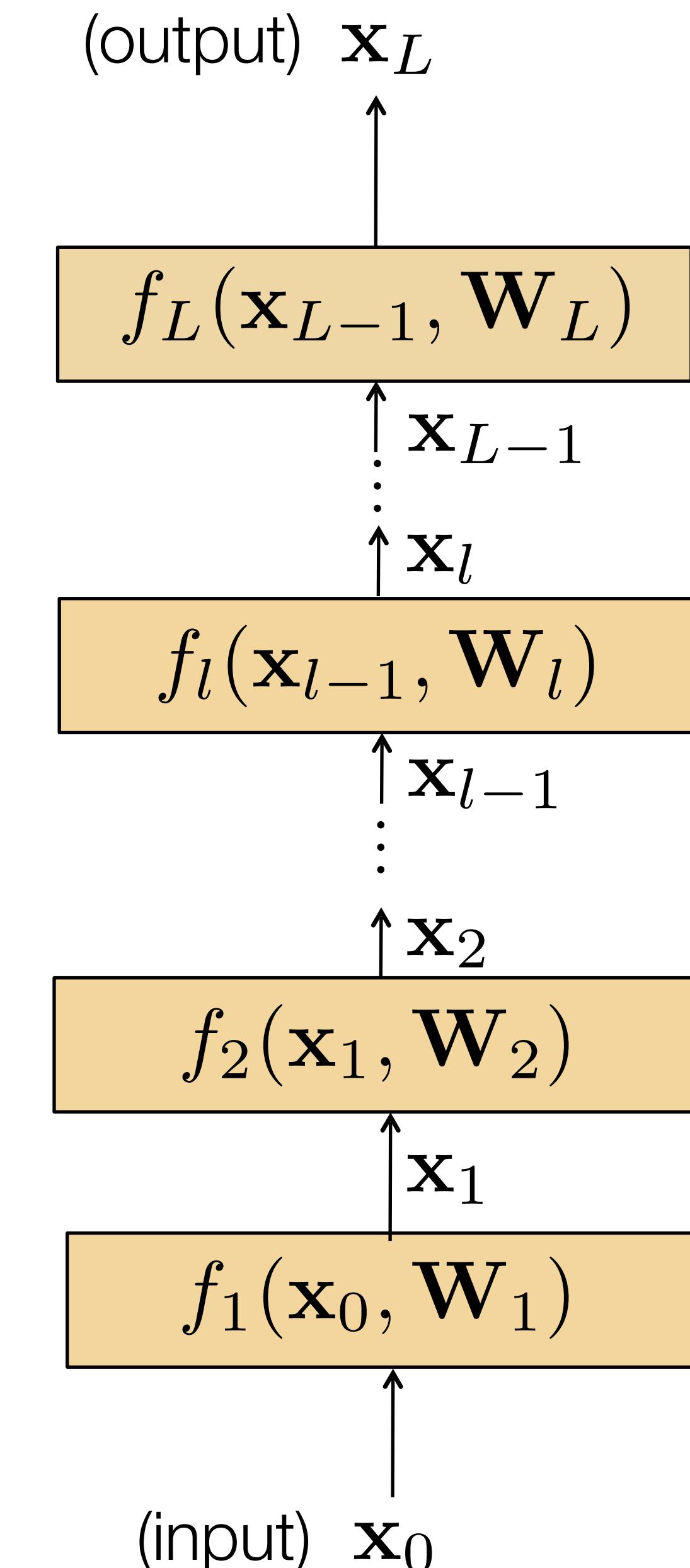
- Want to minimize overall loss function \mathbf{J} , which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.
 - If batchsize=1 then θ is updated after each example.
 - If batchsize=N (full set) then this is standard gradient 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

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 :

$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)$$

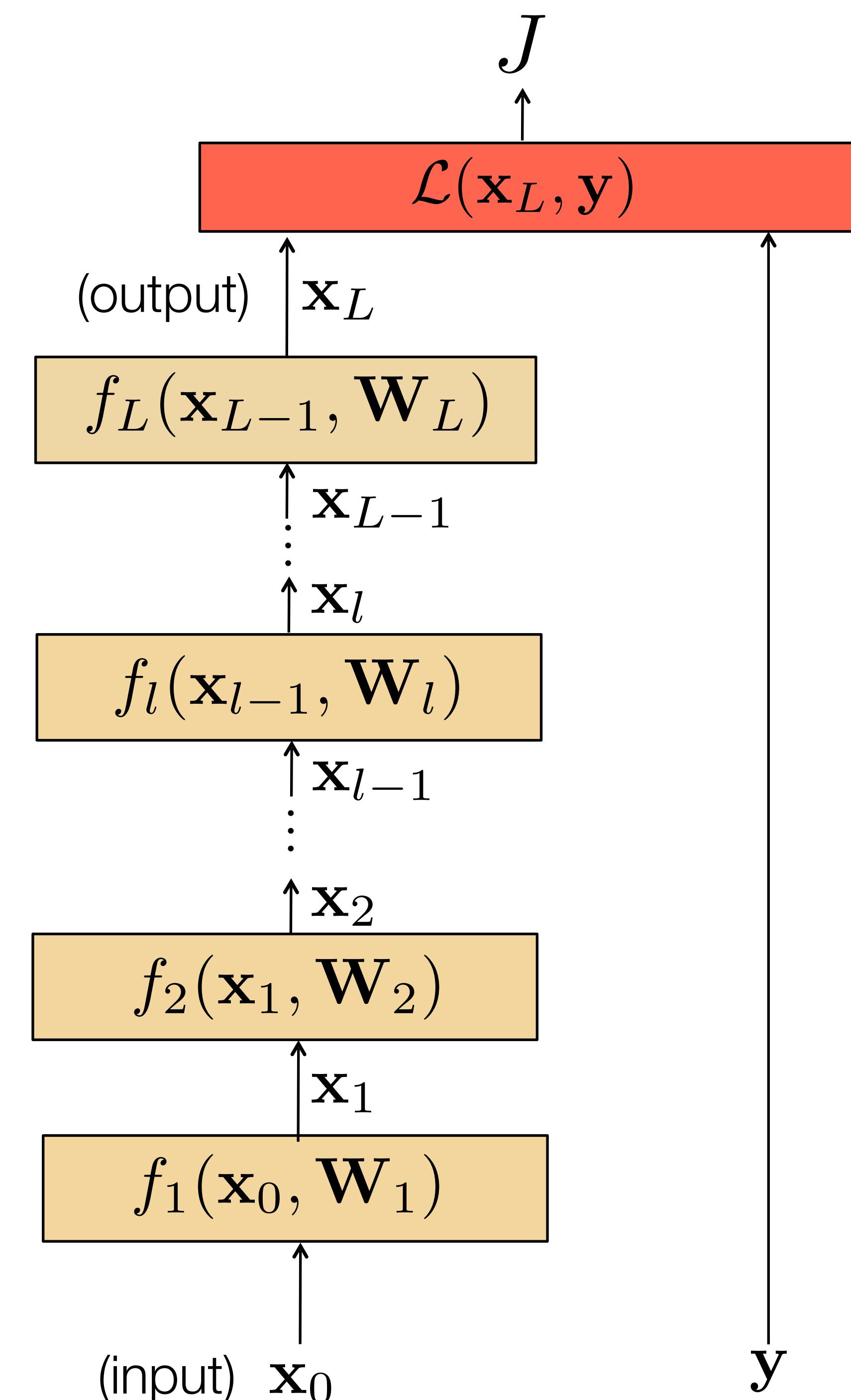
- Output of layer l is \mathbf{x}_l .
- Network output (top layer) is \mathbf{x}_L .



Forward pass

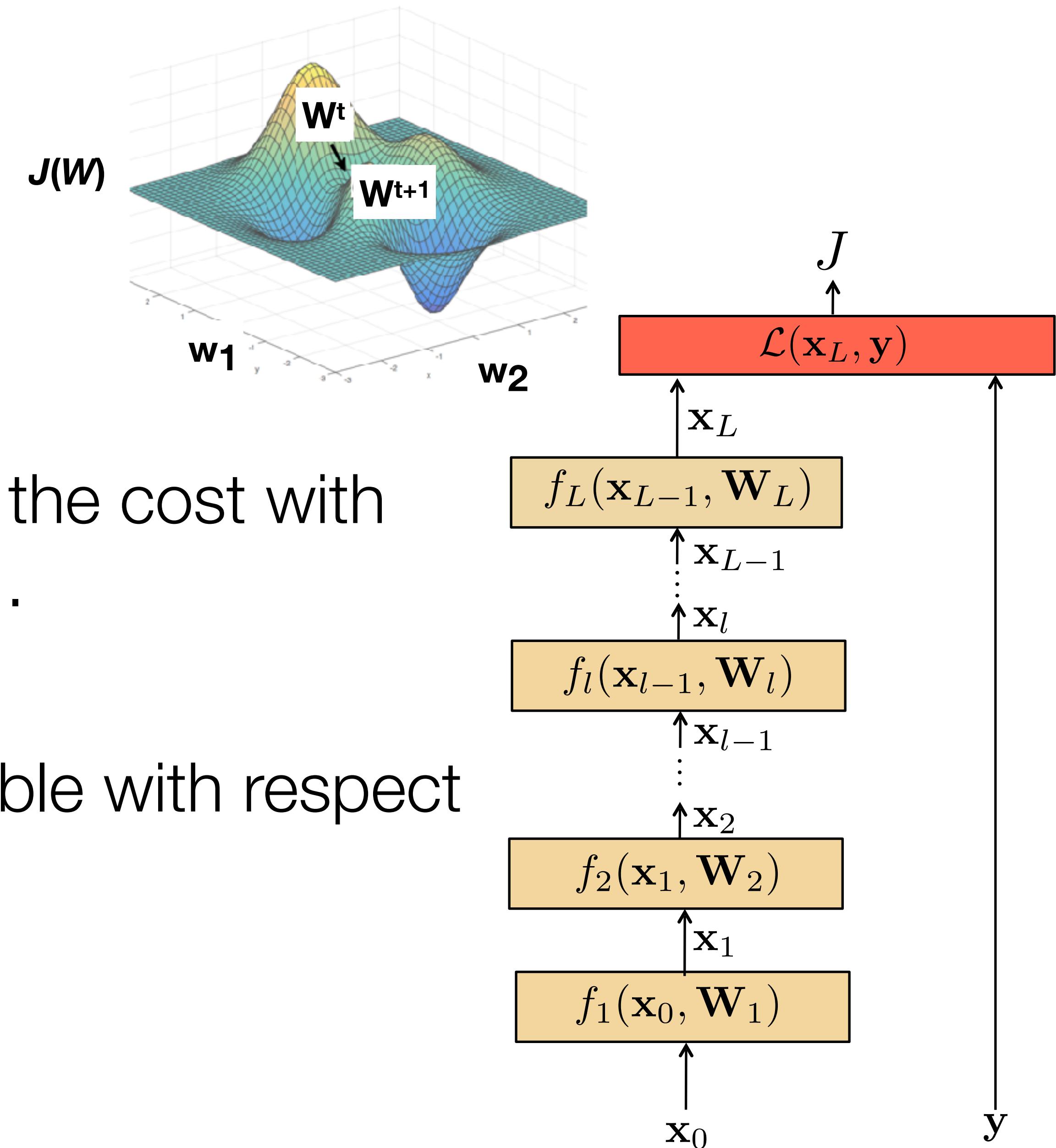
- Consider model with L layers. Layer l has vector of weights \mathbf{W}_l
- **Forward pass:** takes input \mathbf{x}_{l-1} and passes it through each layer f_l :
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)$$
- Output of layer l is \mathbf{x}_l .
- Network output (top layer) is \mathbf{x}_L .
- **Loss function** \mathcal{L} compares \mathbf{x}_L to \mathbf{y} .
- Overall cost is the sum of the losses over all training examples:

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



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.



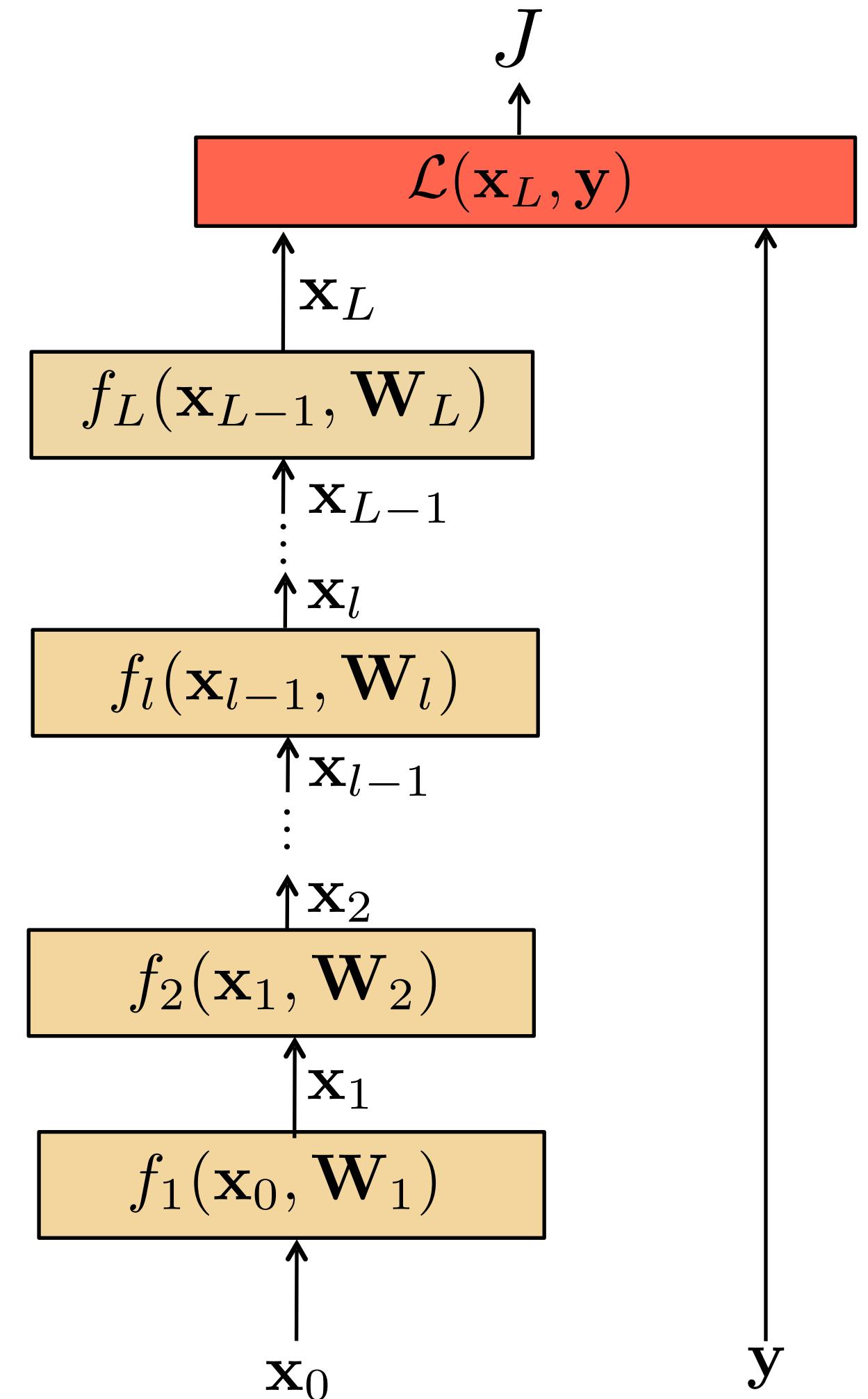
Computing gradients

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

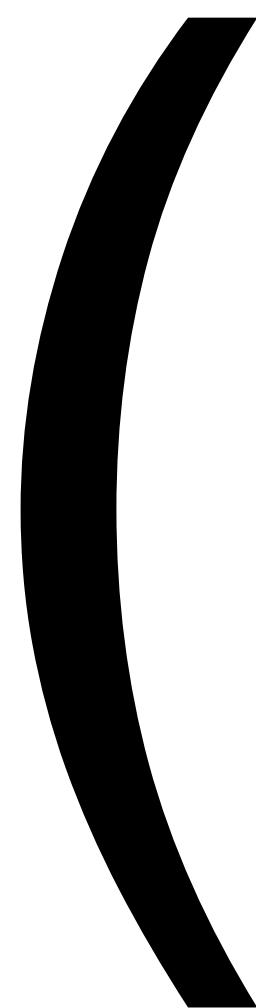
$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f_L(\dots f_2(f_1(\mathbf{x}_0^{(i)}, \mathbf{W}_1), \mathbf{W}_2), \dots \mathbf{W}_L), \mathbf{y}^{(i)})$$

And then compute the partial derivatives...

$$\frac{\partial J}{\partial \mathbf{W}_l}$$



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



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

$$\frac{\partial y}{\partial \mathbf{x}} = \left(\frac{\partial y}{\partial x_1} \quad \frac{\partial y}{\partial x_2} \quad \dots \quad \frac{\partial y}{\partial x_n} \right)$$

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

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

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

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

(m rows and n columns)

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 $[m \times n]$

Wikipedia: The three types of derivatives that have not been considered are those involving vectors-by-matrices, matrices-by-vectors, and matrices-by-matrices. These are not as widely considered and a notation is not widely agreed upon.

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 $\mathbf{u} = g(\mathbf{x})$:

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

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

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

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

$$h'(\mathbf{x}) = \begin{array}{|c|c|c|c|} \hline \textcolor{blue}{\square} & \textcolor{blue}{\square} & \textcolor{blue}{\square} & \textcolor{blue}{\square} \\ \hline \end{array} = \begin{array}{|c|c|} \hline \textcolor{blue}{\square} & \textcolor{blue}{\square} \\ \hline \end{array} \quad \begin{array}{|c|c|c|c|} \hline \textcolor{red}{\square} & \textcolor{red}{\square} & \textcolor{red}{\square} & \textcolor{red}{\square} \\ \hline \end{array}$$

\uparrow \uparrow \uparrow
 $[1 \times 4]$ $[2 \times 2]$ $[2 \times 4]$

Matrix calculus

- Chain rule:

For the function: $h(\mathbf{x}) = f_L(f_{L-1}(\dots f_1(\mathbf{x})))$

With $\mathbf{u}_1 = f_1(\mathbf{x})$

$\mathbf{u}_l = f_l(\mathbf{u}_{l-1})$

$\mathbf{z} = \mathbf{u}_n = f_L(\mathbf{u}_{L-1})$

The derivative becomes a product of matrices:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{a}} = \frac{\partial \mathbf{z}}{\partial \mathbf{u}_{L-1}} \Big|_{\mathbf{u}_{L-1}=f_{L-1}(\mathbf{u}_{L-2})} \cdot \frac{\partial \mathbf{u}^{L-1}}{\partial \mathbf{u}_{L-2}} \Big|_{\mathbf{u}_{L-2}=f_{L-2}(\mathbf{u}_{L-3})} \cdots \frac{\partial \mathbf{u}_2}{\partial \mathbf{u}_1} \Big|_{\mathbf{u}_1=f_1(\mathbf{a})} \cdot \frac{\partial \mathbf{u}_1}{\partial \mathbf{x}} \Big|_{\mathbf{x}=\mathbf{a}}$$

(exercise: check that all the matrix dimensions work out fine)

)

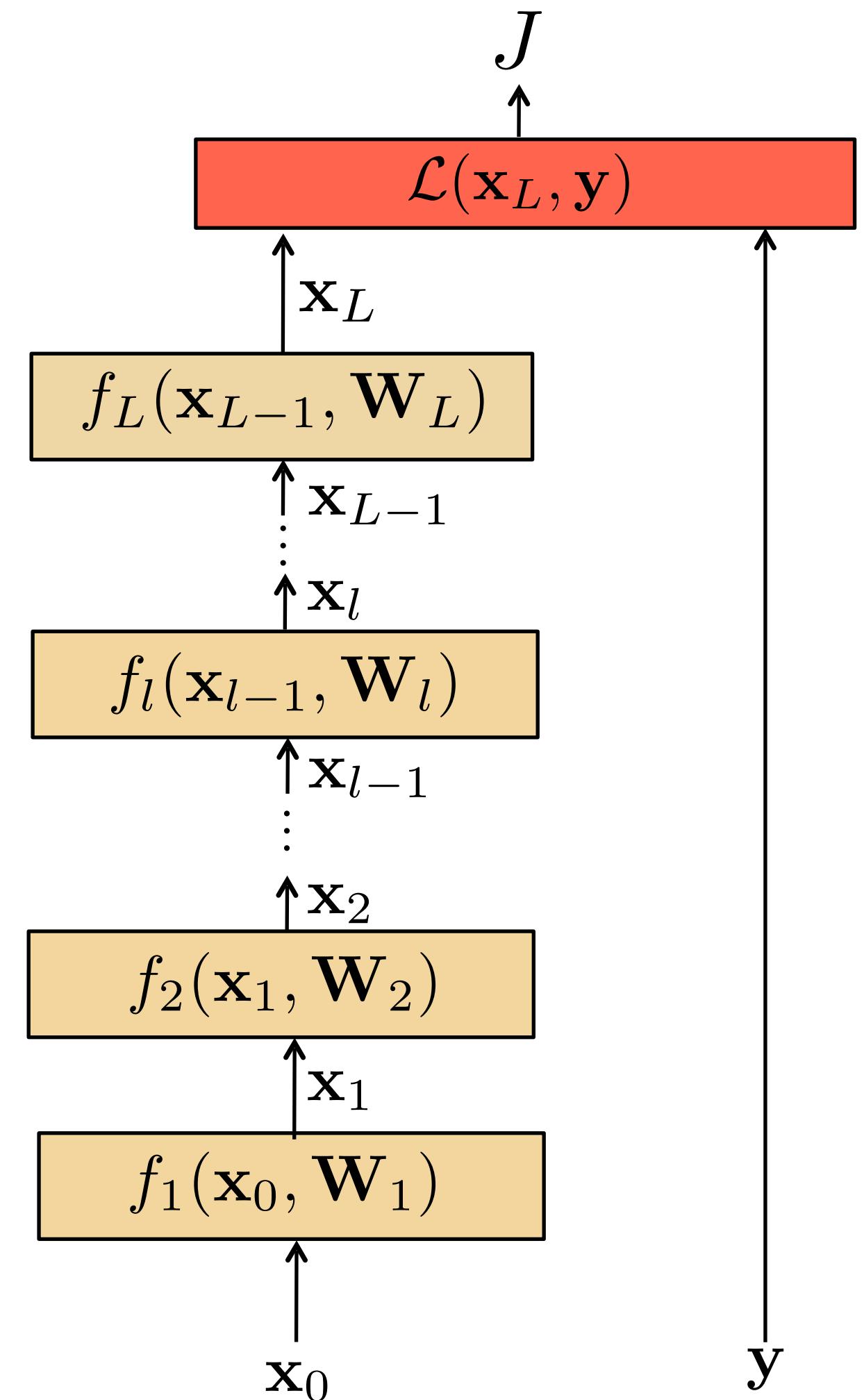
Computing gradients

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

$$J(\mathbf{W}) = \sum_{i=1} \mathcal{L}(f_L(\dots f_2(f_1(\mathbf{x}_0^{(i)}, \mathbf{W}_1), \mathbf{W}_2), \dots \mathbf{W}_L), \mathbf{y}^{(i)})$$

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

backpropagation



Computing gradients

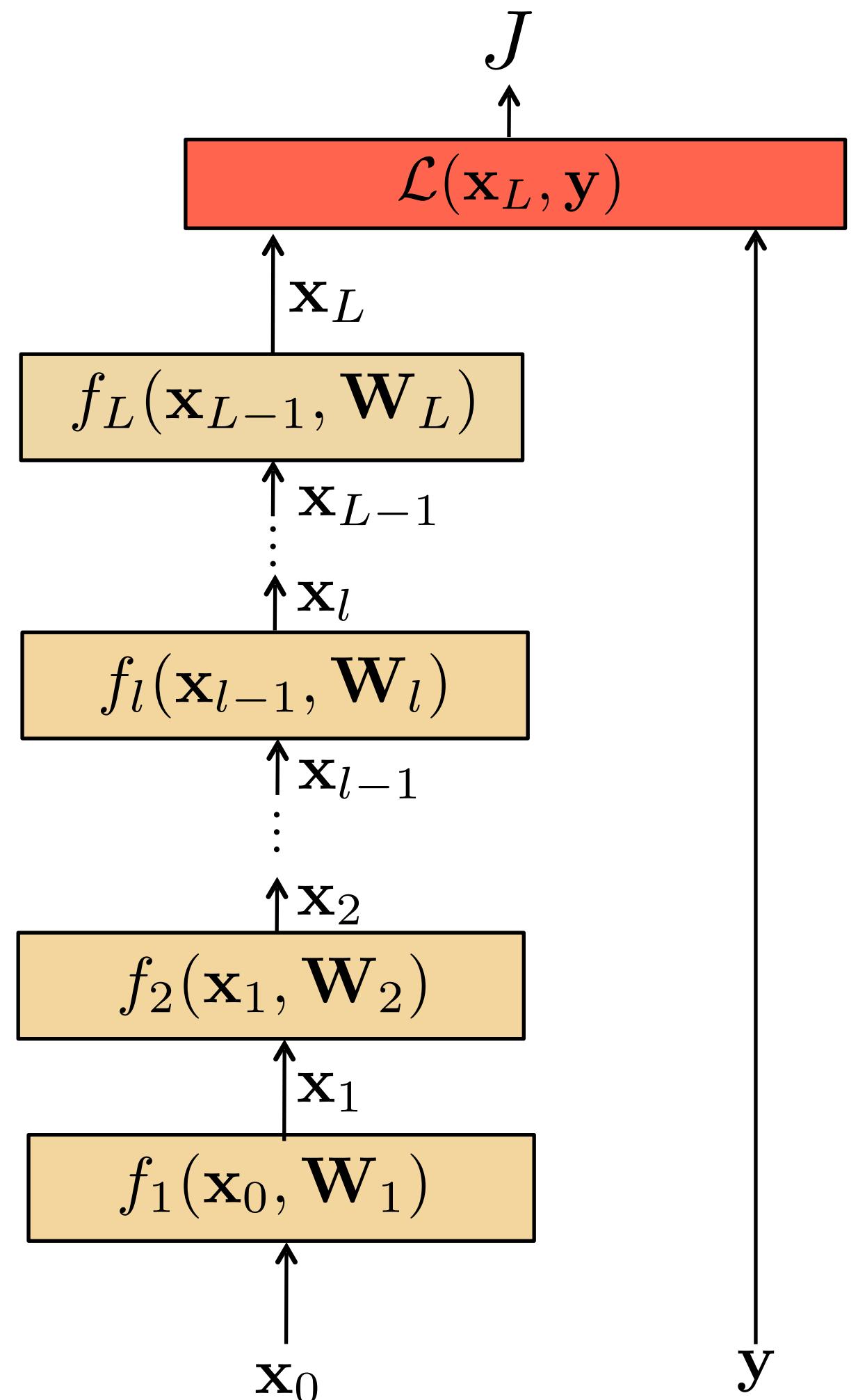
The loss J is the sum of the losses associated with each training example $\{\mathbf{x}_0^{(i)}, \mathbf{y}^{(i)}\}$

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

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

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

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



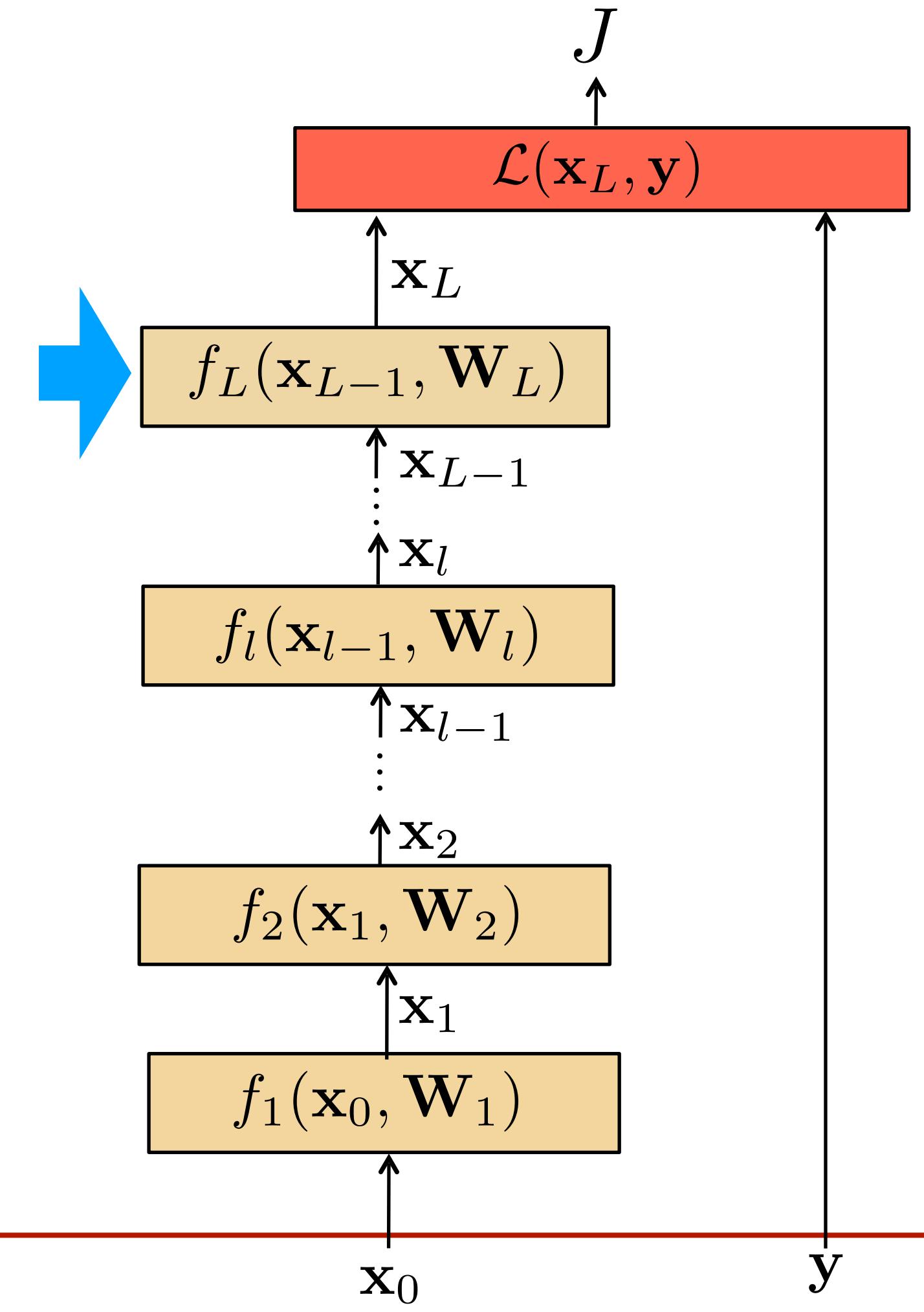
Computing gradients

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) \mathbf{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 \mathbf{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.

Computing gradients: loss layer

If we compute the gradient with respect to the parameters of the last layer (output layer) \mathbf{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:

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

Will depend on the layer structure and non-linearity.

The gradient is:

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

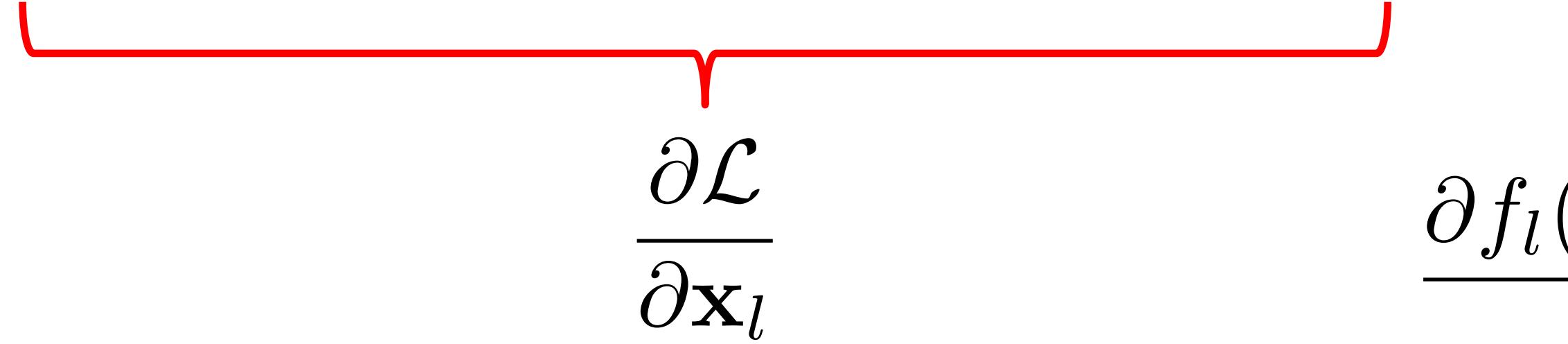
Computing gradients: layer l

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:

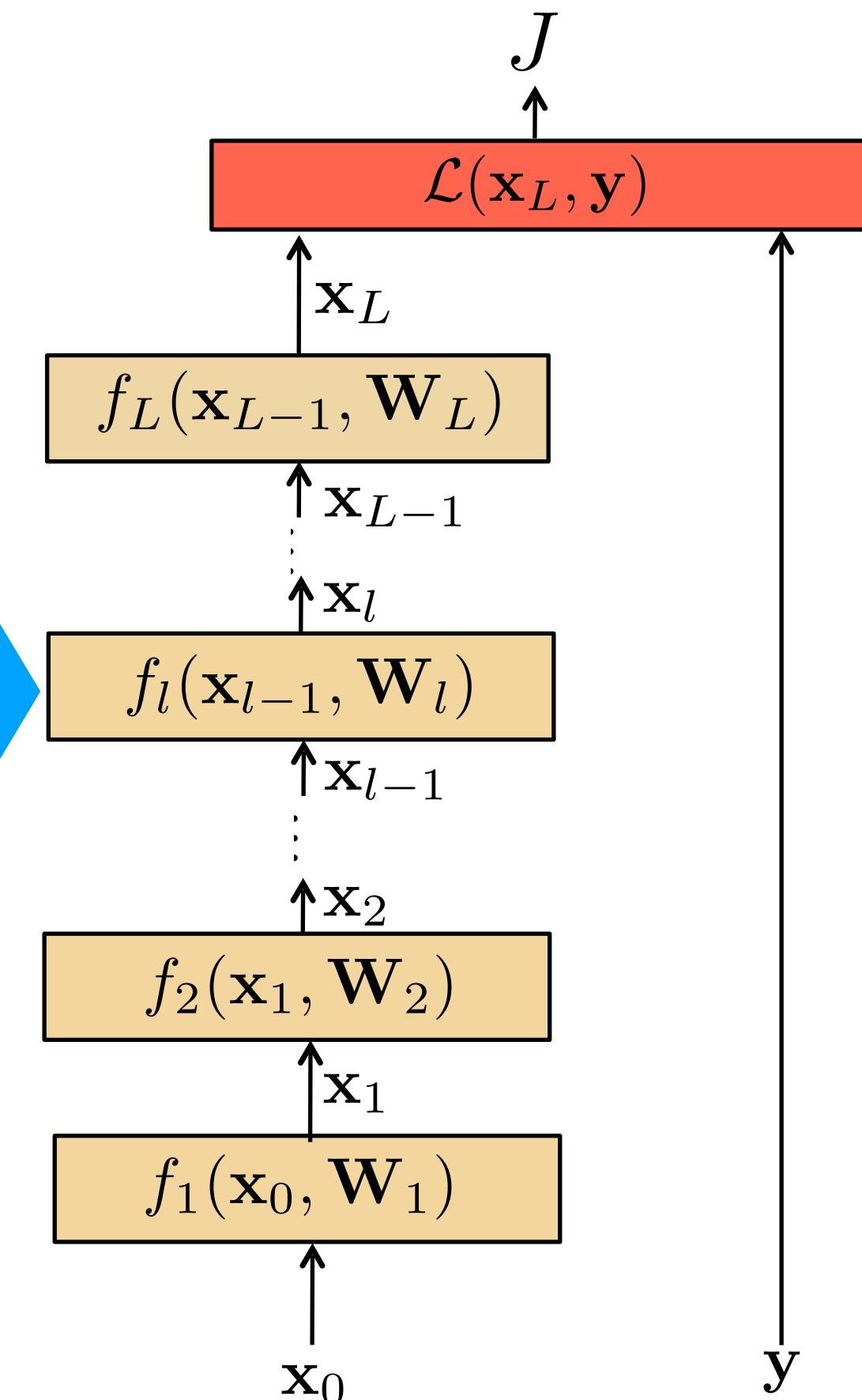
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_L} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdot \frac{\partial \mathbf{x}_{L-1}}{\partial \mathbf{x}_{L-2}} \cdots \frac{\partial \mathbf{x}_{l+1}}{\partial \mathbf{x}_l} \cdot \frac{\partial \mathbf{x}_l}{\partial \mathbf{W}_l}$$



$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_l}$$
$$\frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{W}_l}$$

And this can be
computed iteratively!

This is easy.



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

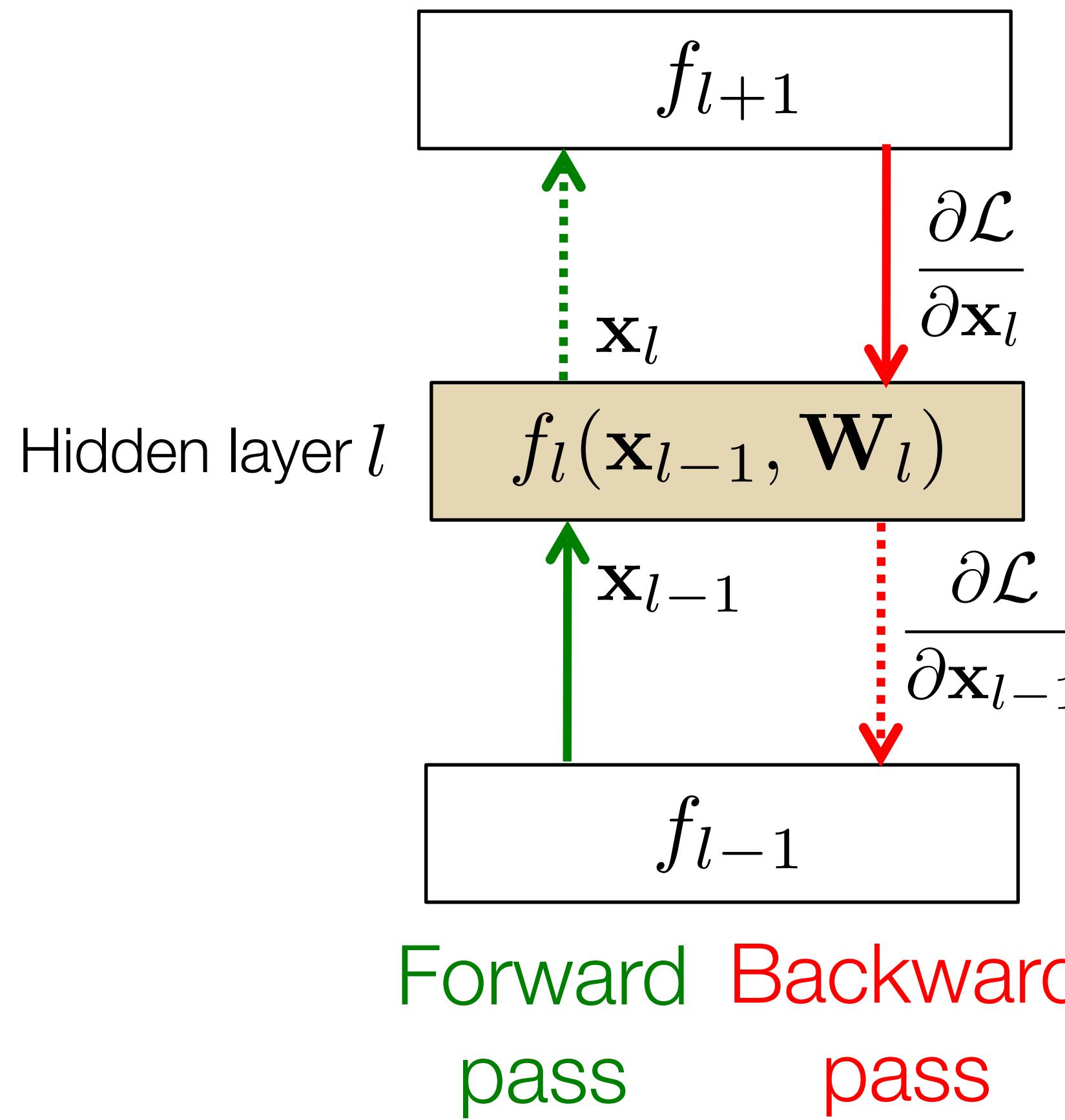
If we have the value of $\frac{\partial \mathcal{L}}{\partial \mathbf{x}_l}$ we can compute the gradient at the
layer below as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{l-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} \cdot \frac{\partial \mathbf{x}_l}{\partial \mathbf{x}_{l-1}}$$

↑ ↑ ↗
Gradient Gradient $\frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{x}_{l-1}}$
layer l-1 layer l

Backpropagation — Goal: to update parameters of layer l

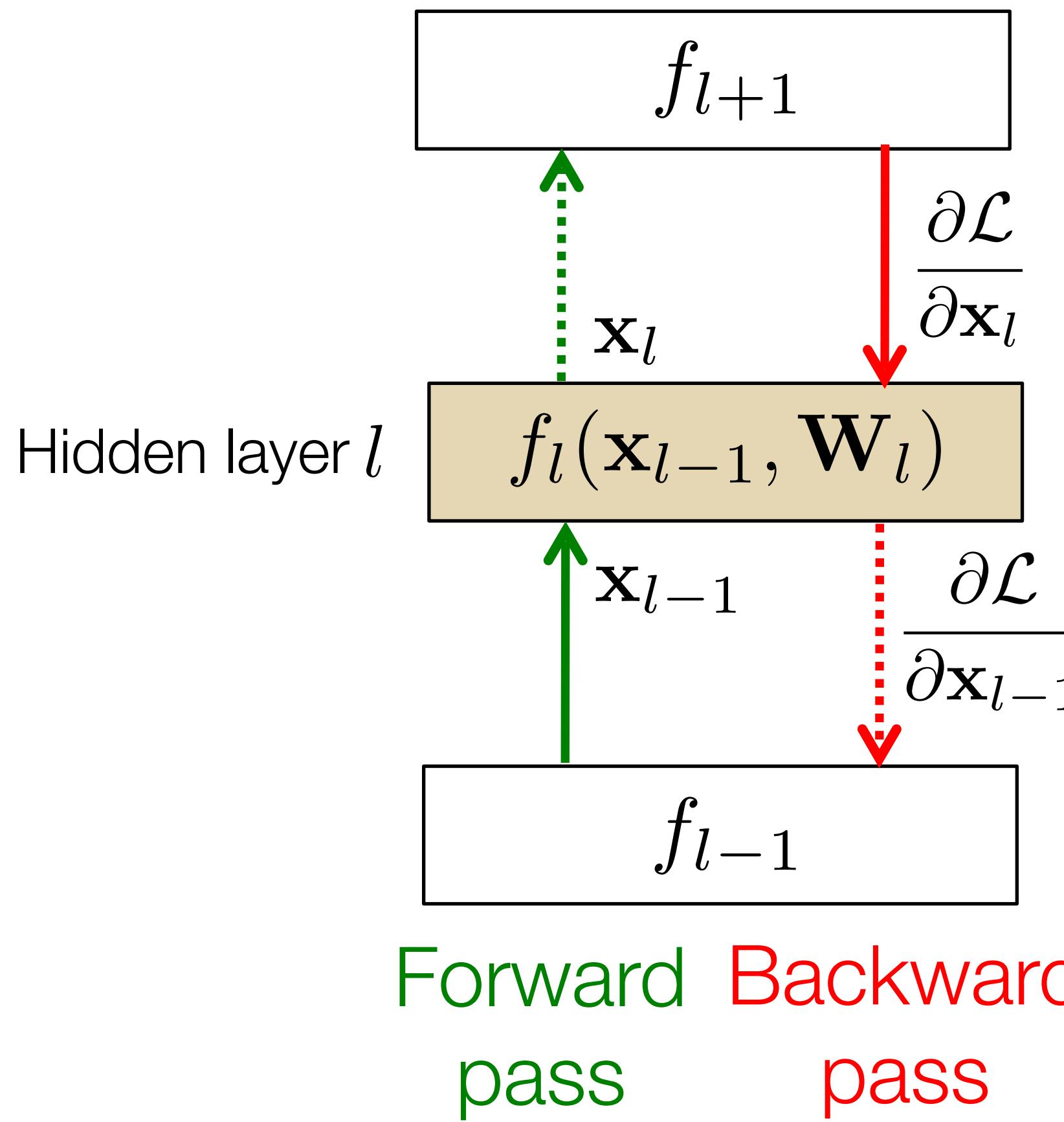
- Layer l has two inputs (during training)



- 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:
$$\frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{W}_l}$$

Backpropagation — Goal: to update parameters of layer l

- Layer l has two inputs (during training)



- Layer l has two inputs (during training)
- We compute the outputs

$$\begin{aligned} \text{brown box} &\rightarrow \mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \mathbf{W}_l) \\ \text{brown box} &\rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{l-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{x}_{l-1}} \end{aligned}$$

- The weight update equation is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{W}_l}$$

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

(sum over all training examples to get J)

Backpropagation Summary

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

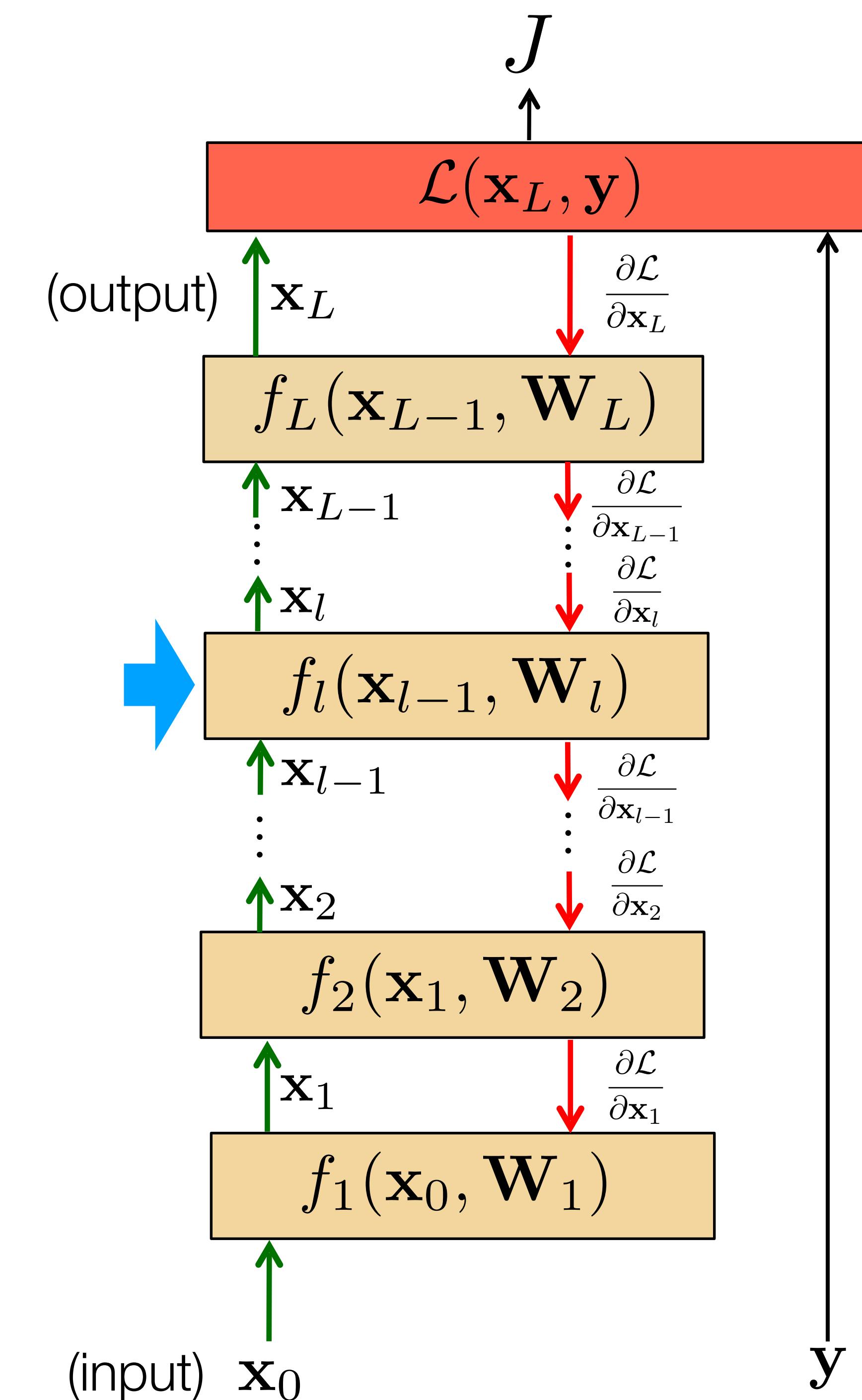
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)$$

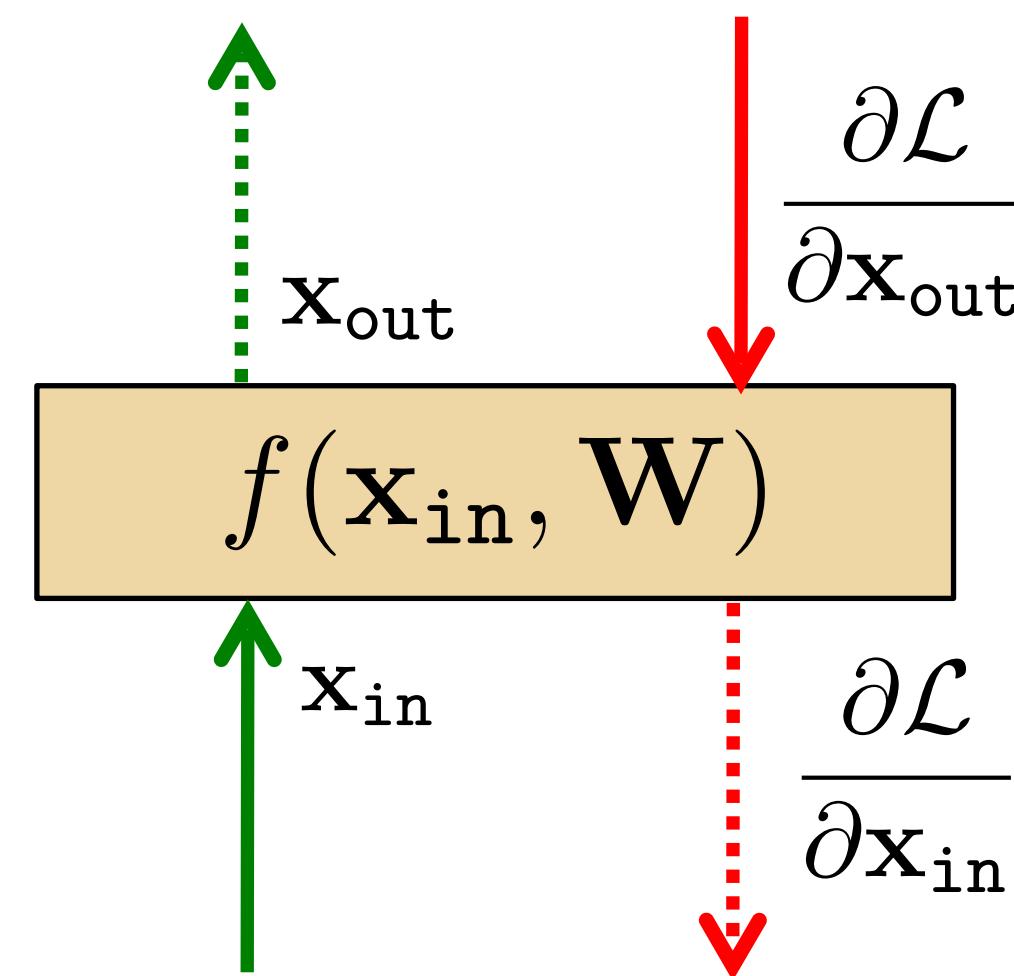
- Backwards pass: compute loss derivatives iteratively from top to bottom:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{l-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{x}_{l-1}}$$

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

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l(\mathbf{x}_{l-1}, \mathbf{W}_l)}{\partial \mathbf{W}_l}$$





Linear Module

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

$$\begin{array}{c} \text{green} \\ \text{green} \\ \text{blue} \\ \text{blue} \\ \text{blue} \\ \text{blue} \end{array} = \begin{array}{|c|c|c|c|} \hline \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ \hline \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ \hline \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ \hline \text{blue} & \text{blue} & \text{blue} & \text{blue} \\ \hline \end{array} \begin{array}{c} \text{green} \\ \text{green} \\ \text{green} \\ \text{green} \end{array}$$

With \mathbf{W} being a matrix of size $|\mathbf{x}_{\text{out}}| \times |\mathbf{x}_{\text{in}}|$

- Backprop to input:

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

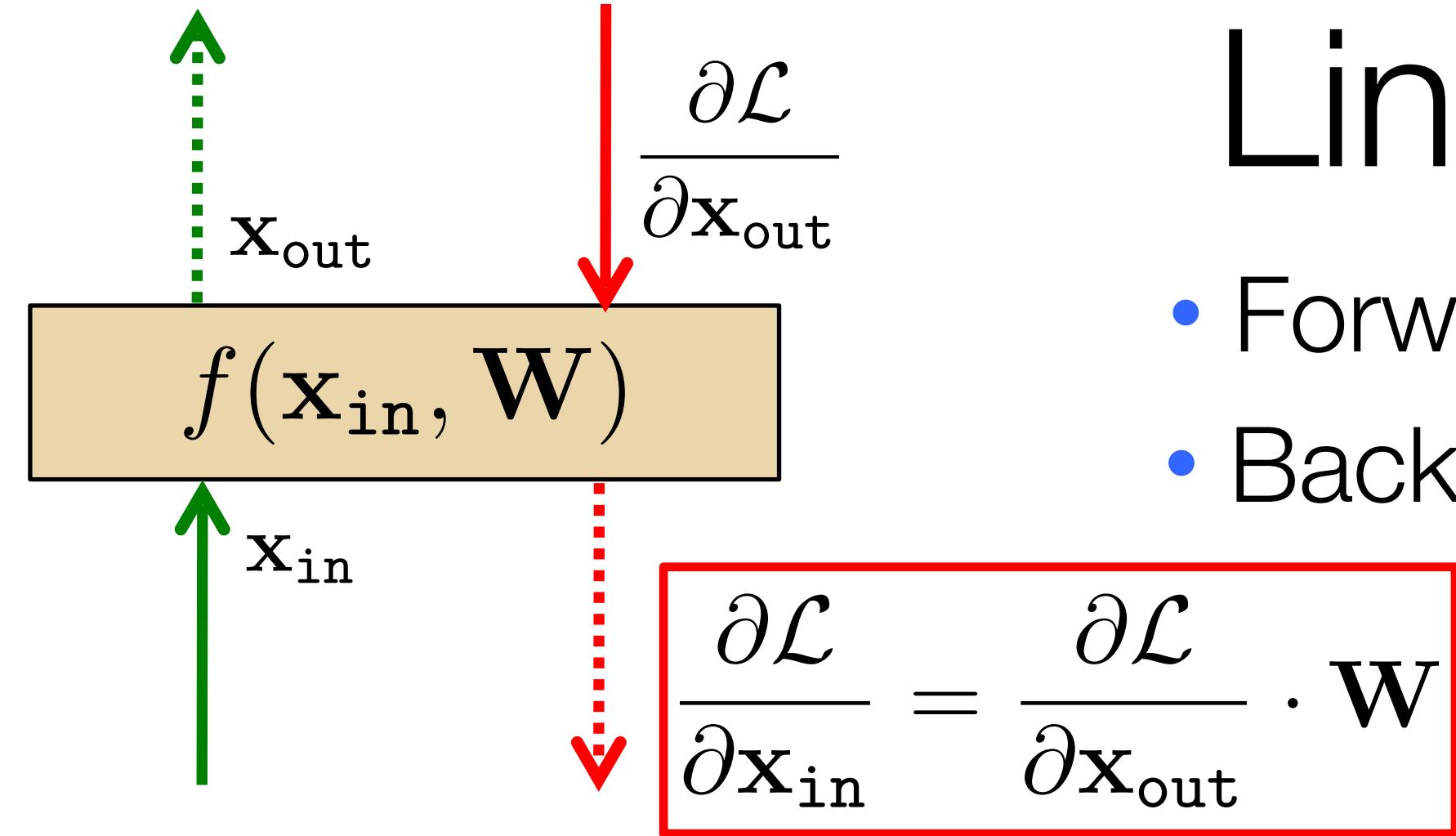
If we look at the j component of output \mathbf{x}_{out} , with respect to the i component of the input, \mathbf{x}_{in} :

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

Therefore:

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

$$\begin{array}{c} \text{red} \\ \text{red} \\ \text{red} \\ \text{red} \end{array} = \begin{array}{|c|c|c|} \hline \text{red} & \text{red} & \text{red} \\ \hline \end{array} \begin{array}{c} \text{blue} \\ \text{blue} \\ \text{blue} \\ \text{blue} \end{array}$$

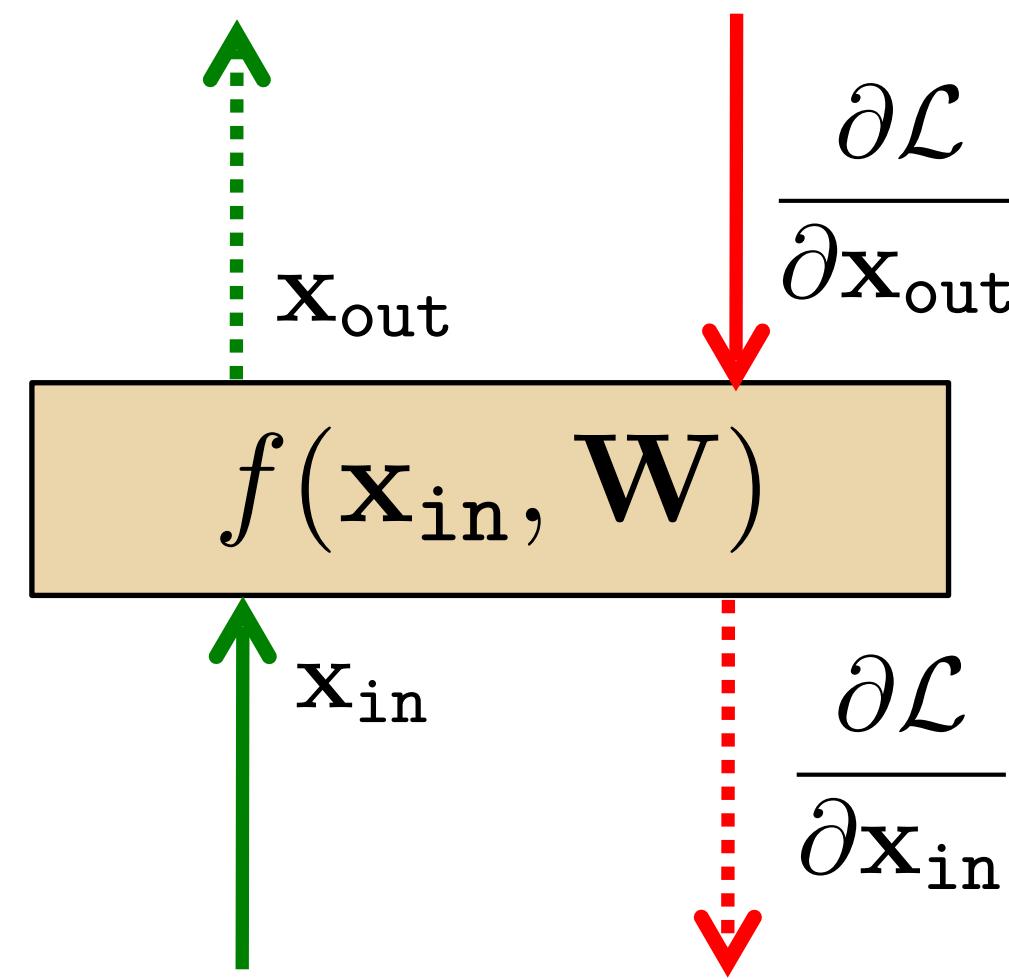


Linear Module

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

$$\begin{matrix} \textcolor{red}{\boxed{\textcolor{red}{\square}} \quad \textcolor{red}{\boxed{\textcolor{red}{\square}} \quad \textcolor{red}{\boxed{\textcolor{red}{\square}} \quad \textcolor{red}{\boxed{\textcolor{red}{\square}}}}}} & = & \textcolor{red}{\boxed{\textcolor{red}{\square}} \quad \textcolor{red}{\boxed{\textcolor{red}{\square}} \quad \textcolor{red}{\boxed{\textcolor{red}{\square}}}}}} & \textcolor{blue}{\boxed{\textcolor{blue}{\square} \quad \textcolor{blue}{\boxed{\textcolor{blue}{\square}} \quad \textcolor{blue}{\boxed{\textcolor{blue}{\square}} \quad \textcolor{blue}{\boxed{\textcolor{blue}{\square}}}}}} \\ \text{Input} & & \text{Output} & \text{Weights} \end{matrix}$$

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



Linear Module

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

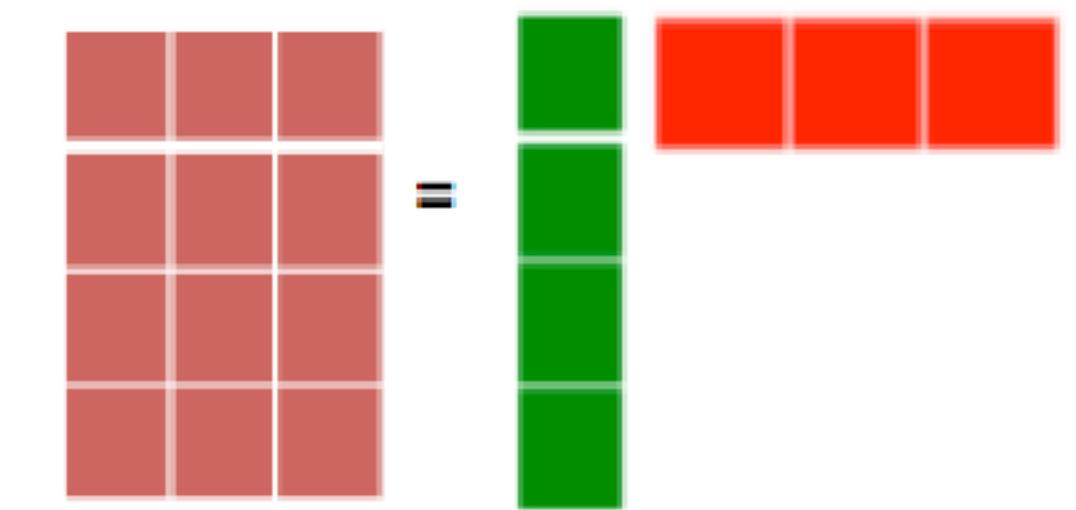
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial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

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

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

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

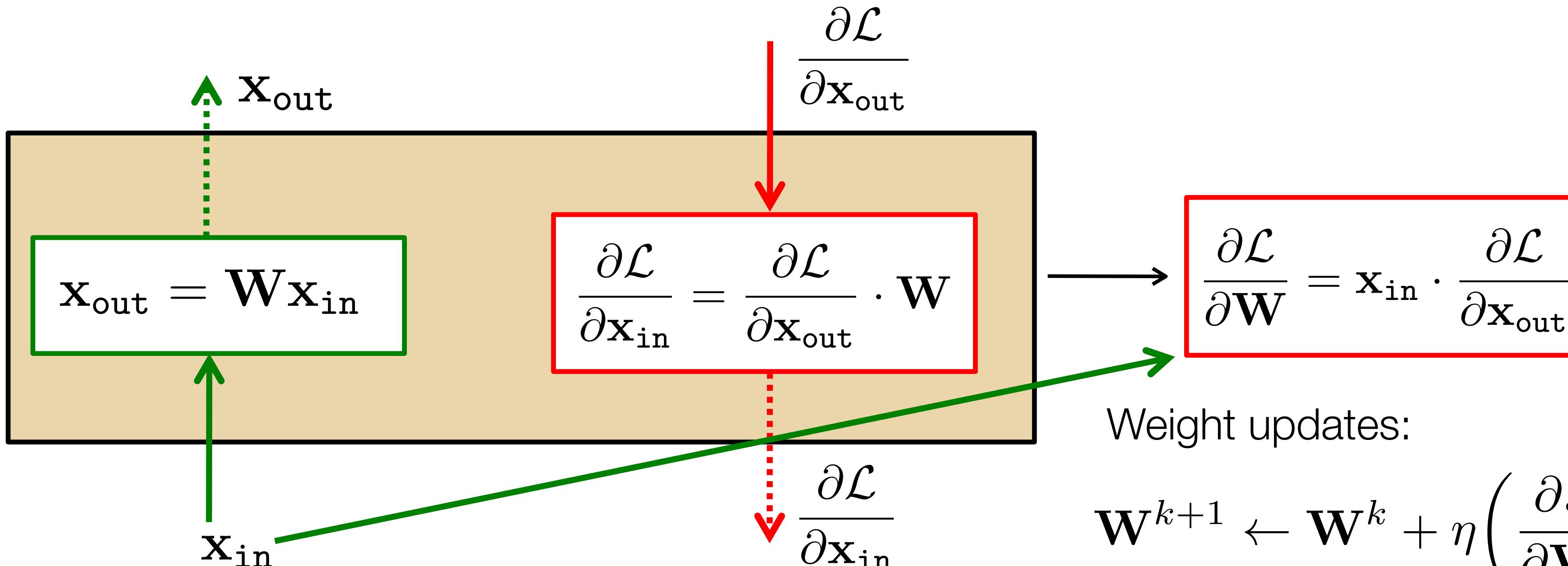
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}}$$



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

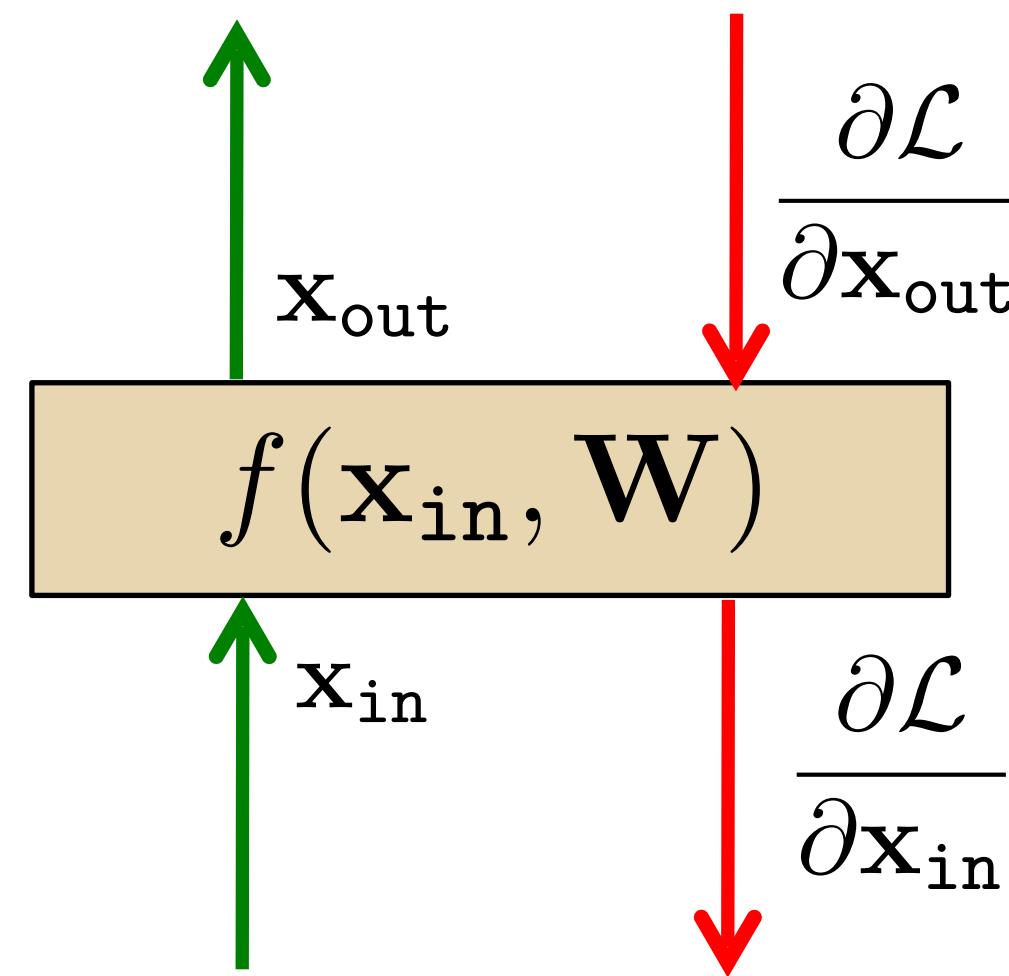
$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^k + \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^T \quad (\text{sum over all training examples to get J})$$

Linear Module



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

Sum over N training pairs



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

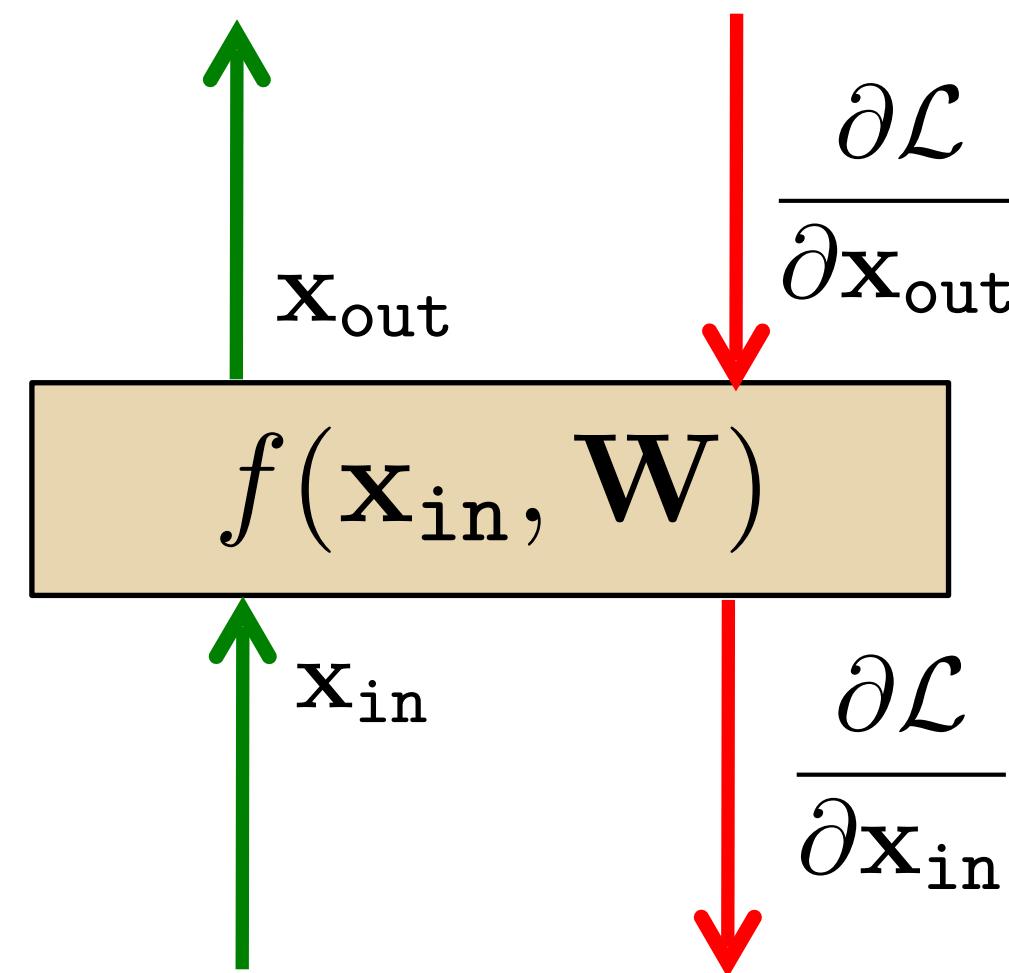
- Backprop to bias: $\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial \mathbf{x}_{\text{out}_i}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}_i}} \cdot \frac{\partial h(\mathbf{x}_{\text{in}_i}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

We use this last expression to update the bias.

Some useful derivatives:

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

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



Pointwise function

- Forward propagation:

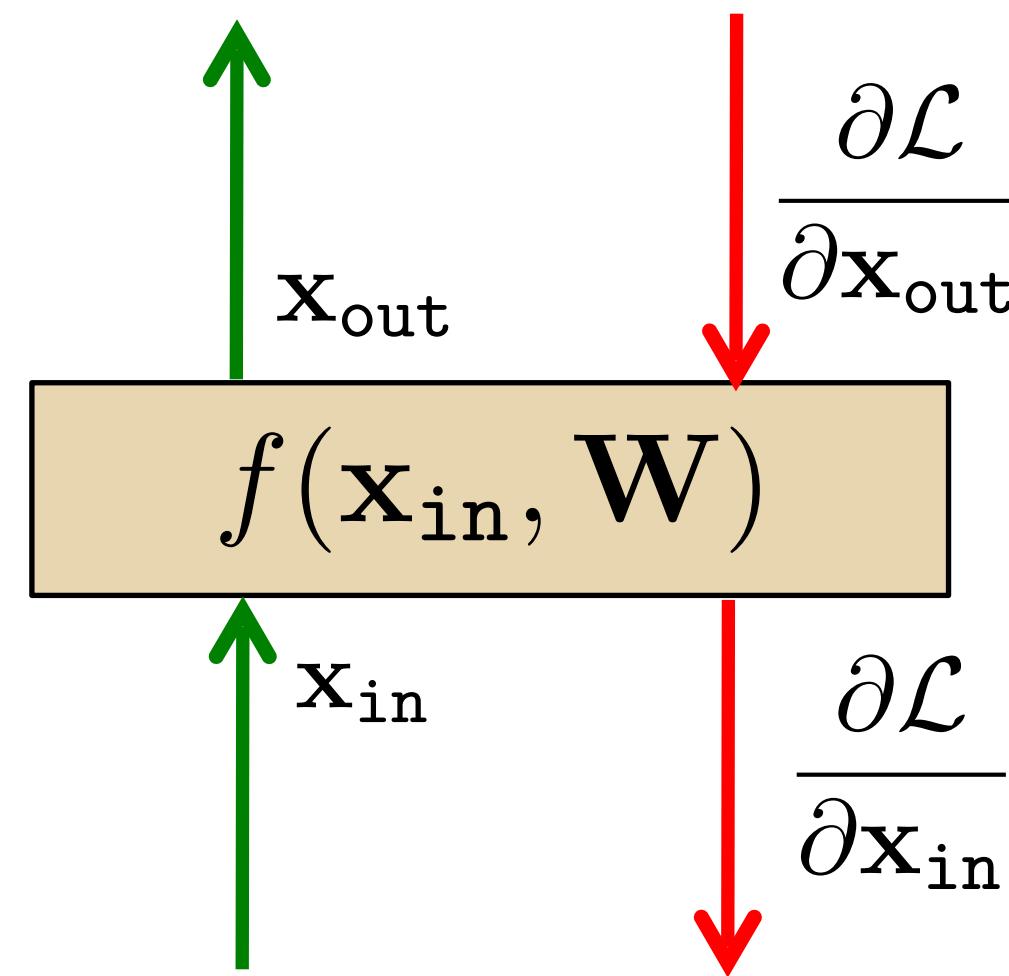
$$\mathbf{x}_{\text{out}_i} = h(\mathbf{x}_{\text{in}_i} + b_i)$$

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

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

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

We use this last expression to update the bias.



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

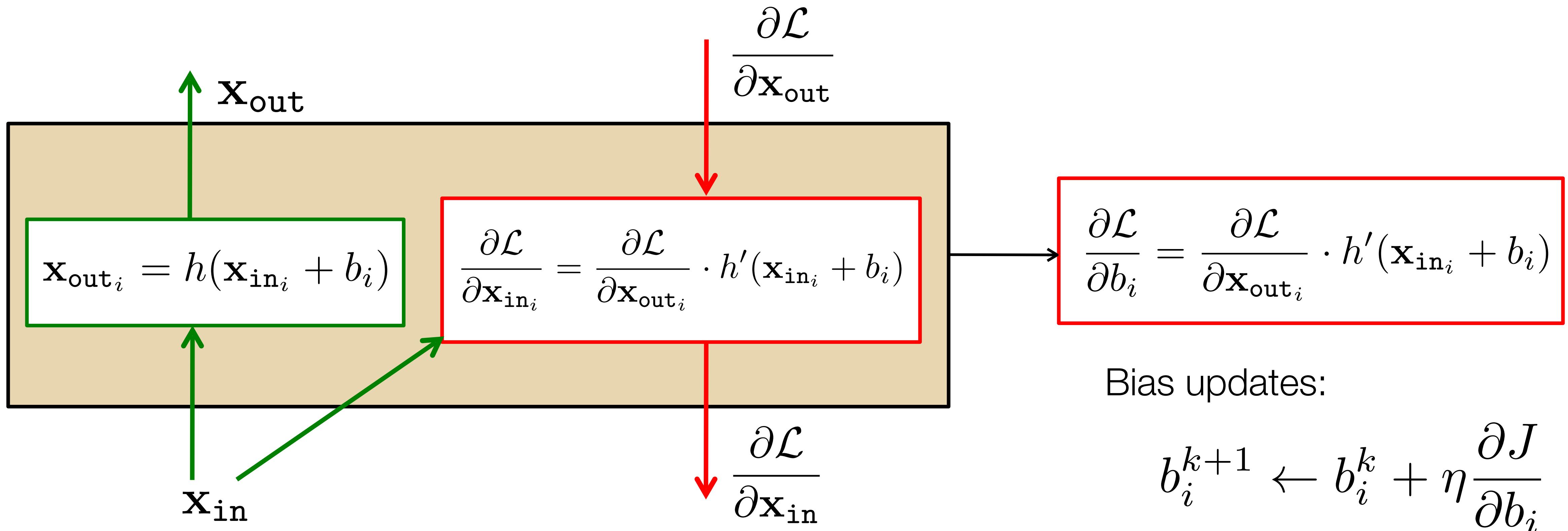
We use this last expression to update the bias.

Some useful derivatives:

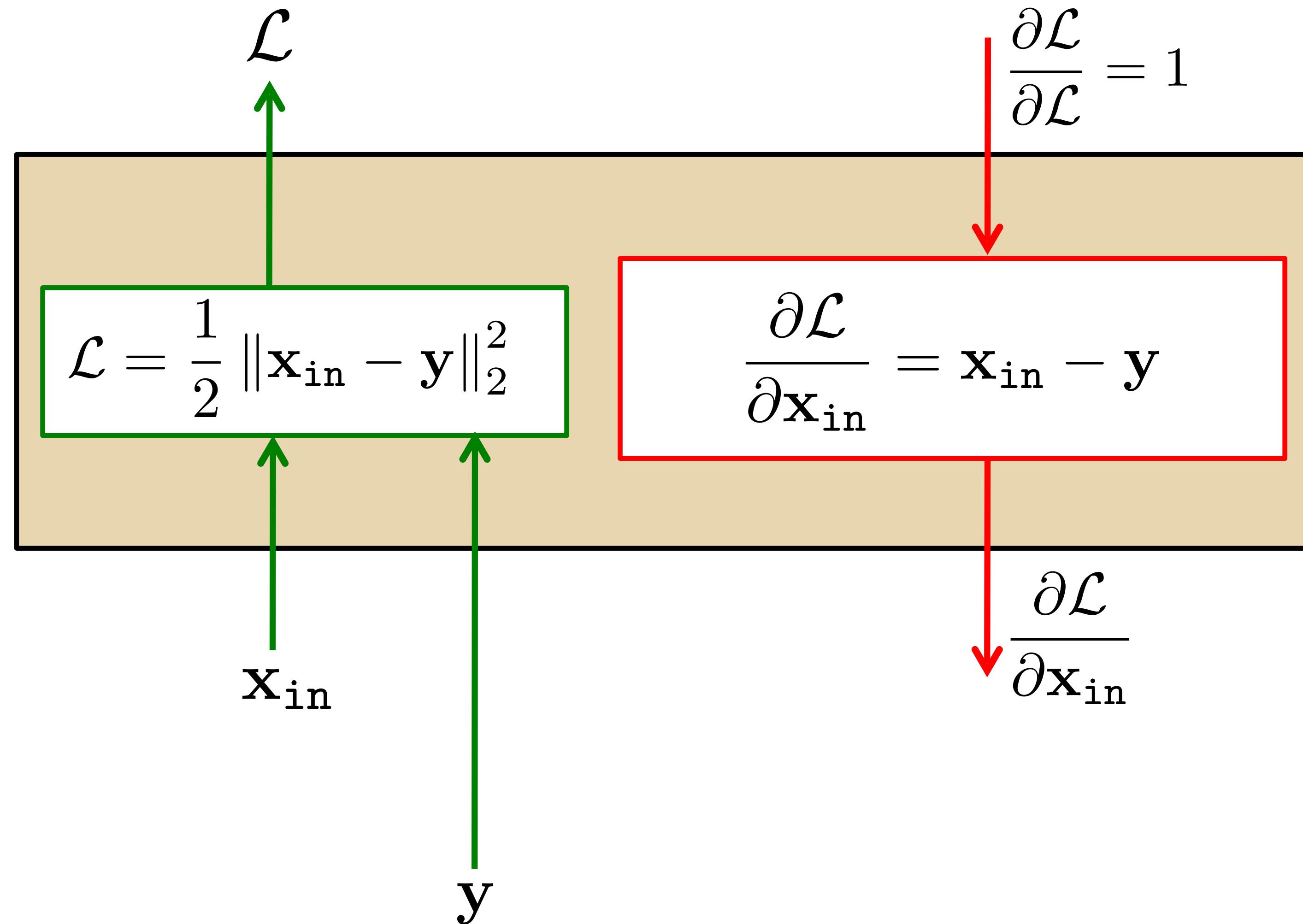
For hyperbolic tangent: $\tanh'(x) = 1 - \tanh^2(x)$

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

Pointwise function

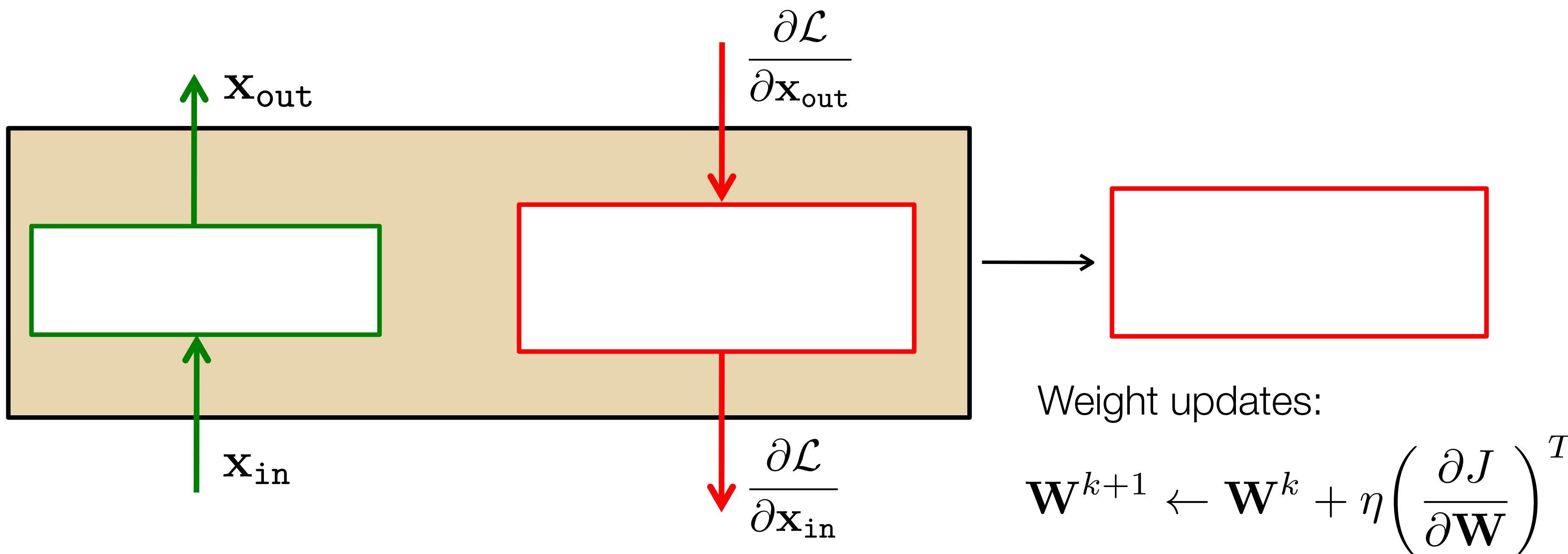


Euclidean loss module



Homework: Convolution Module

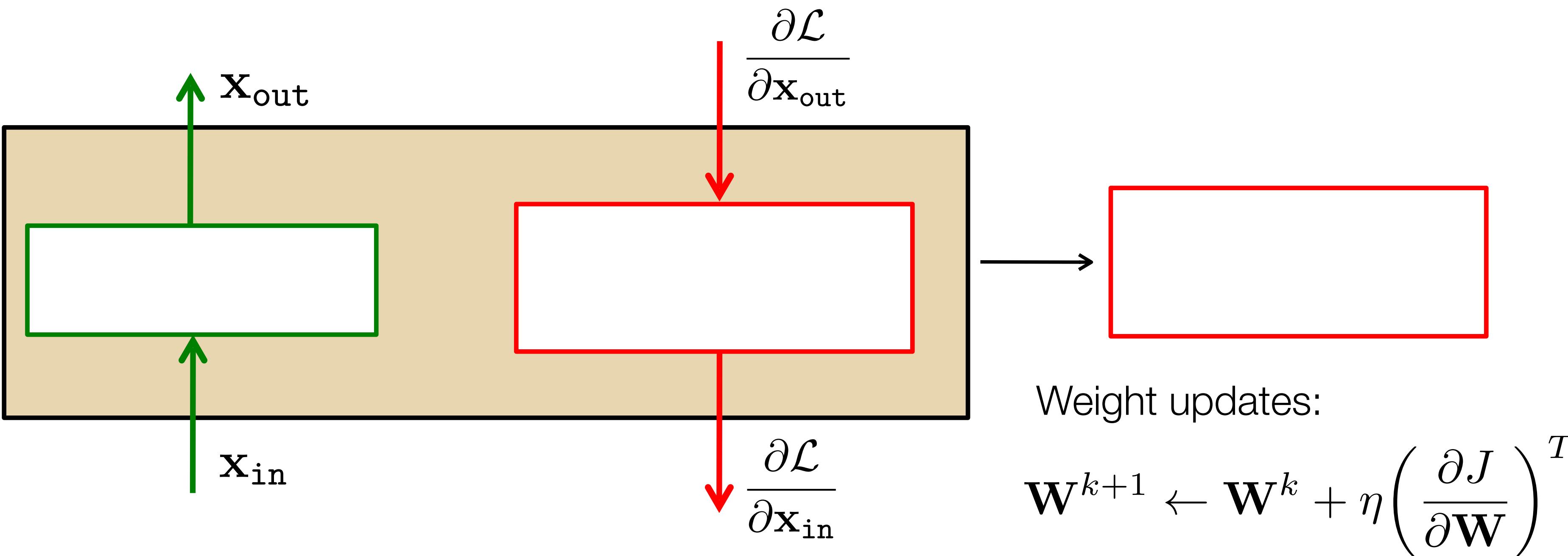
Assume the input x_{in} and output x_{out} are 1D signals of the same length N .
The convolution kernel is W , and has length $k < N$



Derive the equations that go inside each box.
Discuss how you handle the boundaries.

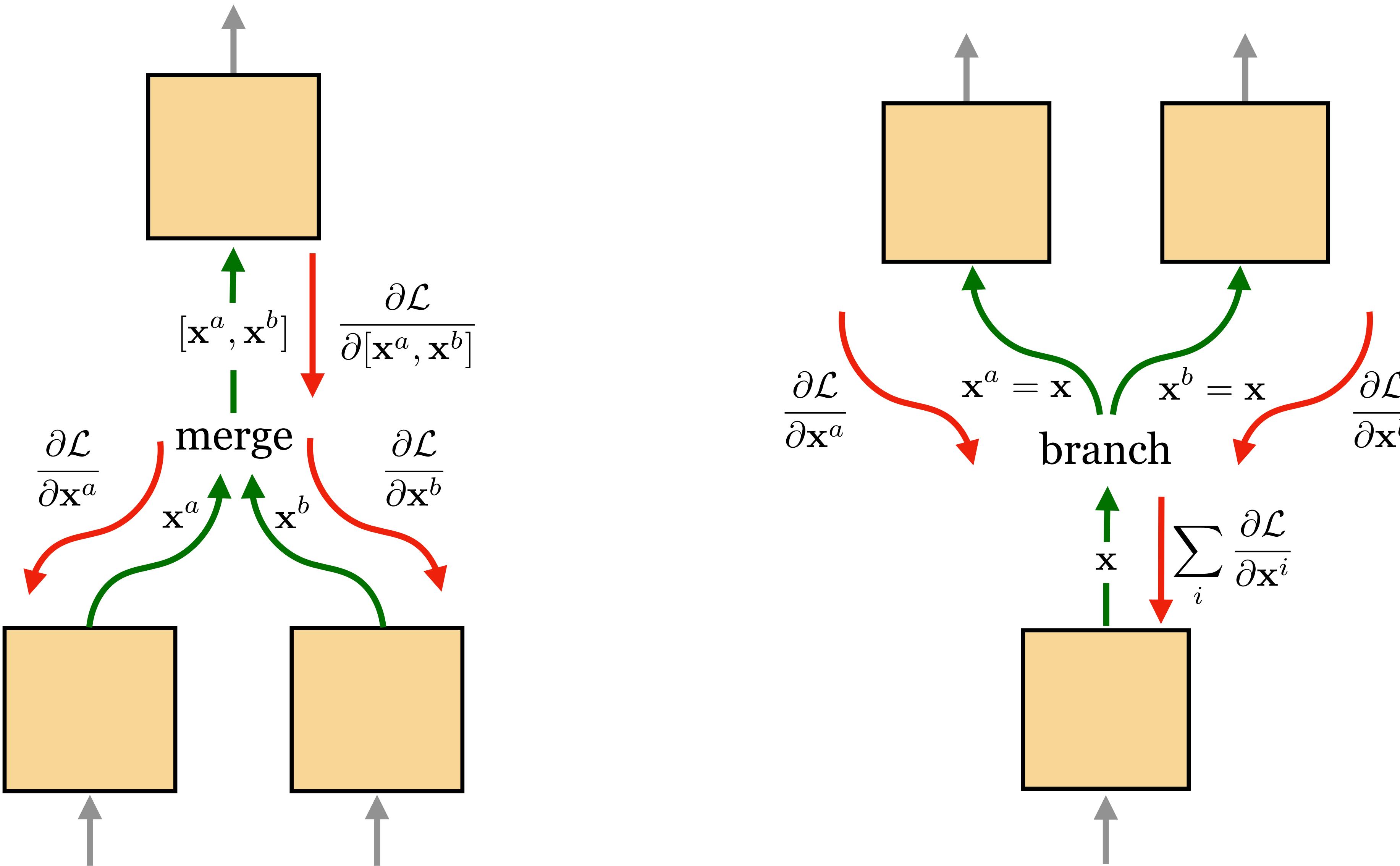
Homework: max pooling module

Assume the input x_{in} and output x_{out} are 1D signals of different lengths.

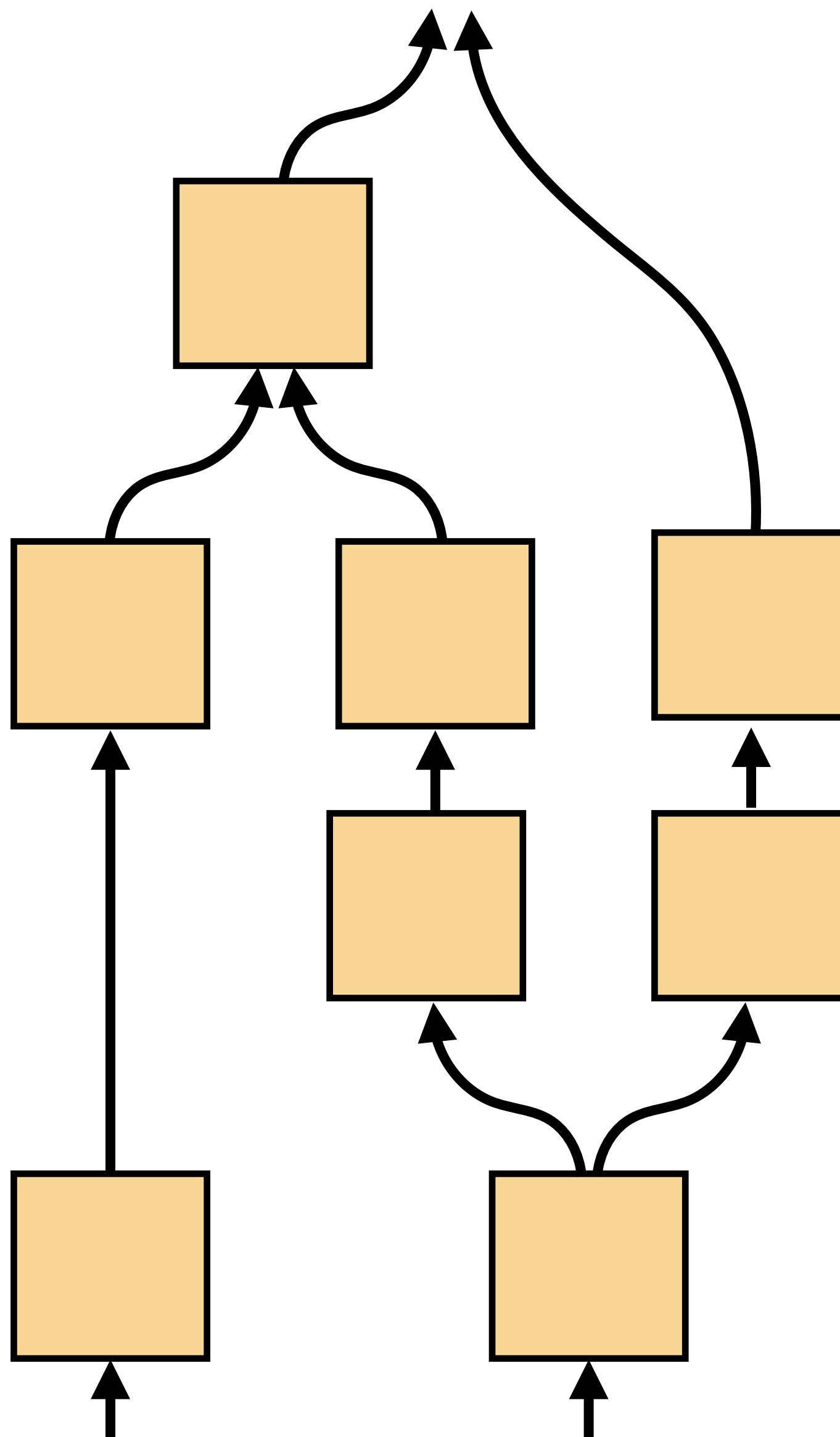


Derive the equations that go inside each box.
Discuss how you handle the boundaries.

Branching and Merging



Computation Graphs



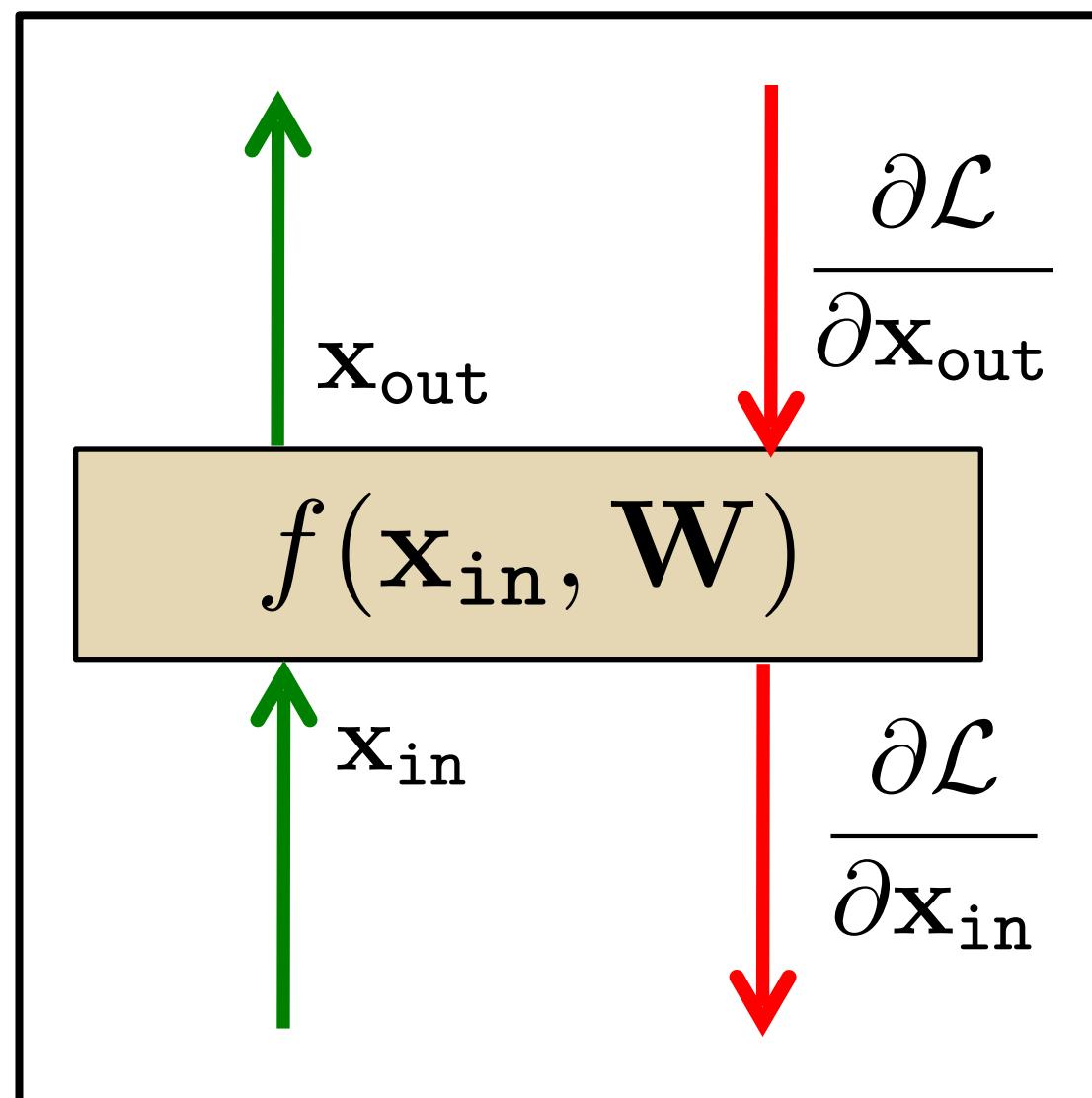
If all the modules (□) are differentiable, and they are connected to form a Directed Acyclic Graph (**DAG**), then you can use backprop to train this whole system!

This is an example of a **computation graph**.

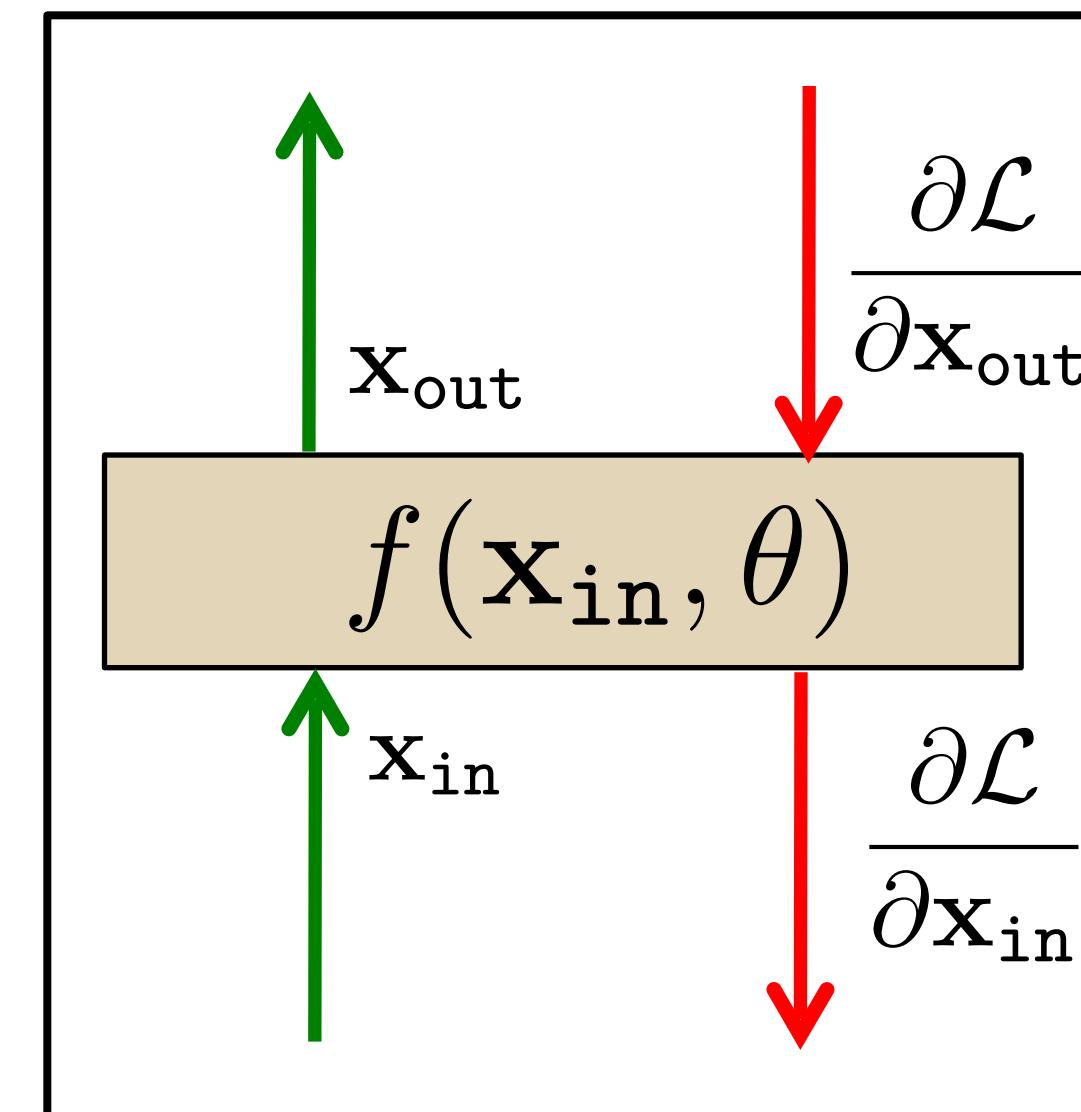
Many programs can be represented with differentiable DAGs, not just “neural nets”.

Differentiable programming

Deep learning

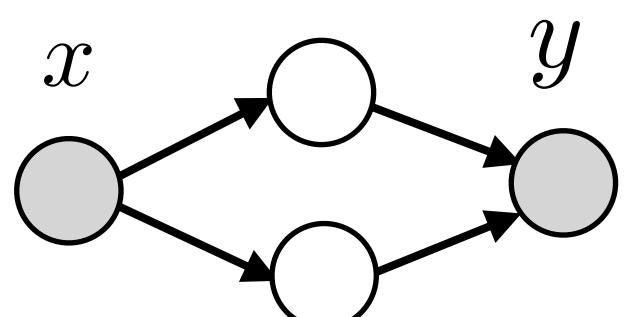


Differentiable programming



PyTorch

TensorFlow™



```
1 for i, data in enumerate(dataset):
2     iter_start_time = time.time()
3     if total_steps % opt.print_freq == 0:
4         t_data = iter_start_time - iter_data_time
5         visualizer.reset()
6     total_steps += opt.batch_size
7     epoch_iter += opt.batch_size
8     model.set_input(data)
9     model.optimize_parameters()
```

Differentiable programming

Deep nets are popular for a few reasons:

1. Easy to optimize (differentiable)
2. Compositional “block based programming”

An emerging term for general models with these properties is **differentiable programming**.



Yann LeCun

January 5 ·

OK, Deep Learning has outlived its usefulness as a buzz-phrase.
Deep Learning est mort. Vive Differentiable Programming!



Thomas G. Dietterich

@tdietterich

Following

DL is essentially a new style of programming--"differentiable programming"--and the field is trying to work out the reusable constructs in this style. We have some: convolution, pooling, LSTM, GAN, VAE, memory units, routing units, etc. 8/

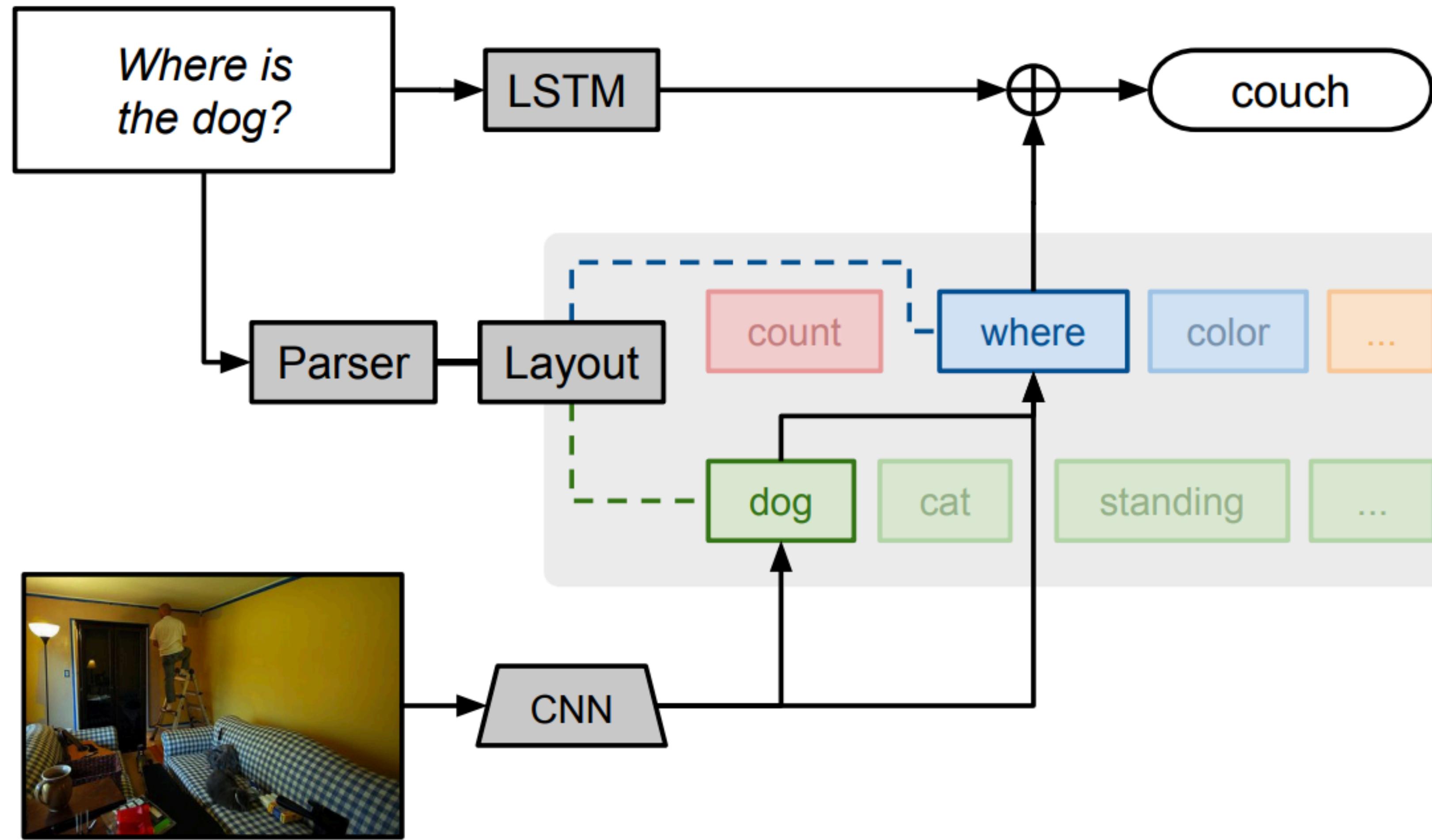
8:02 AM - 4 Jan 2018

65 Retweets 194 Likes



6 65 194

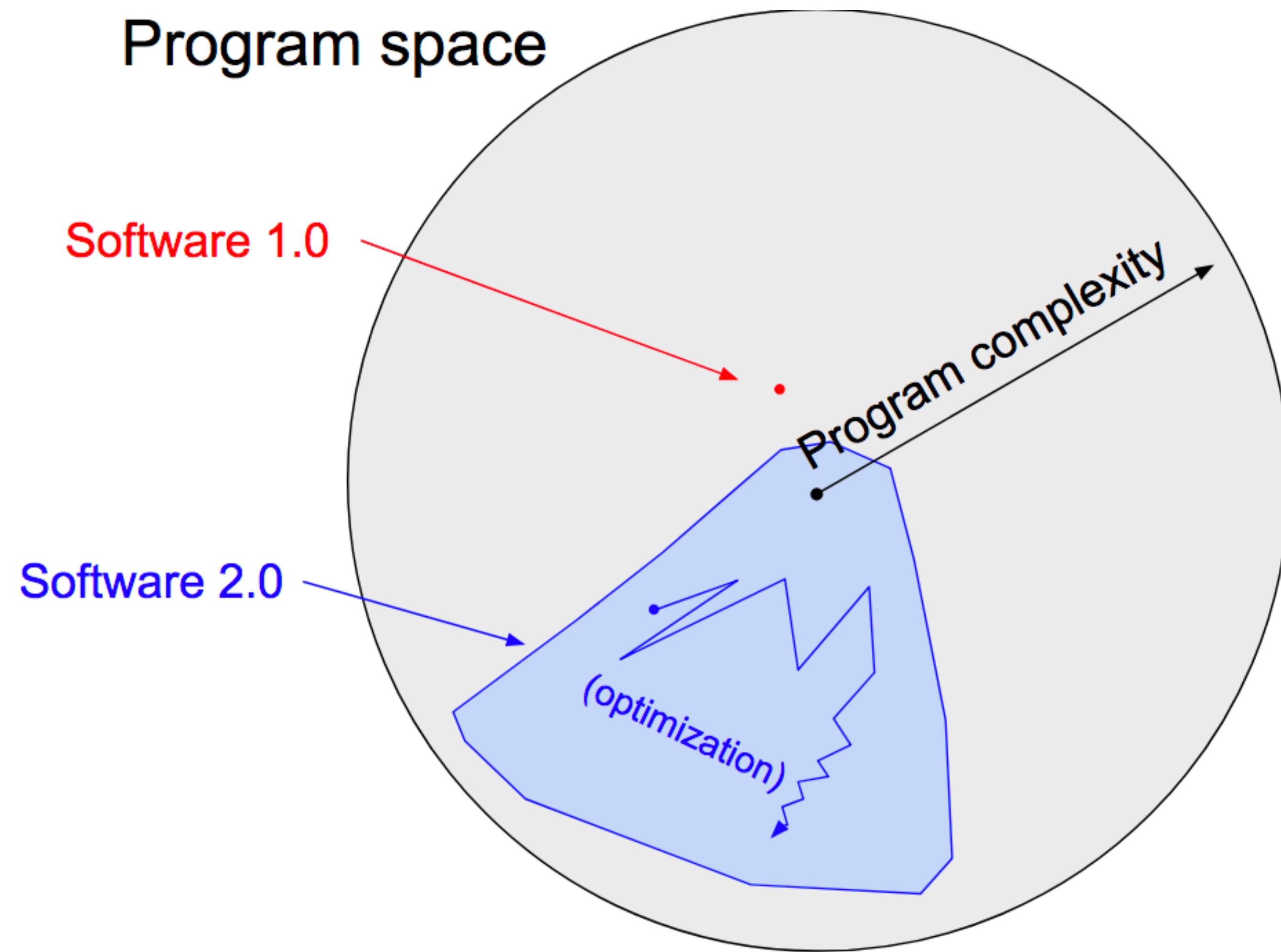
Differentiable programming

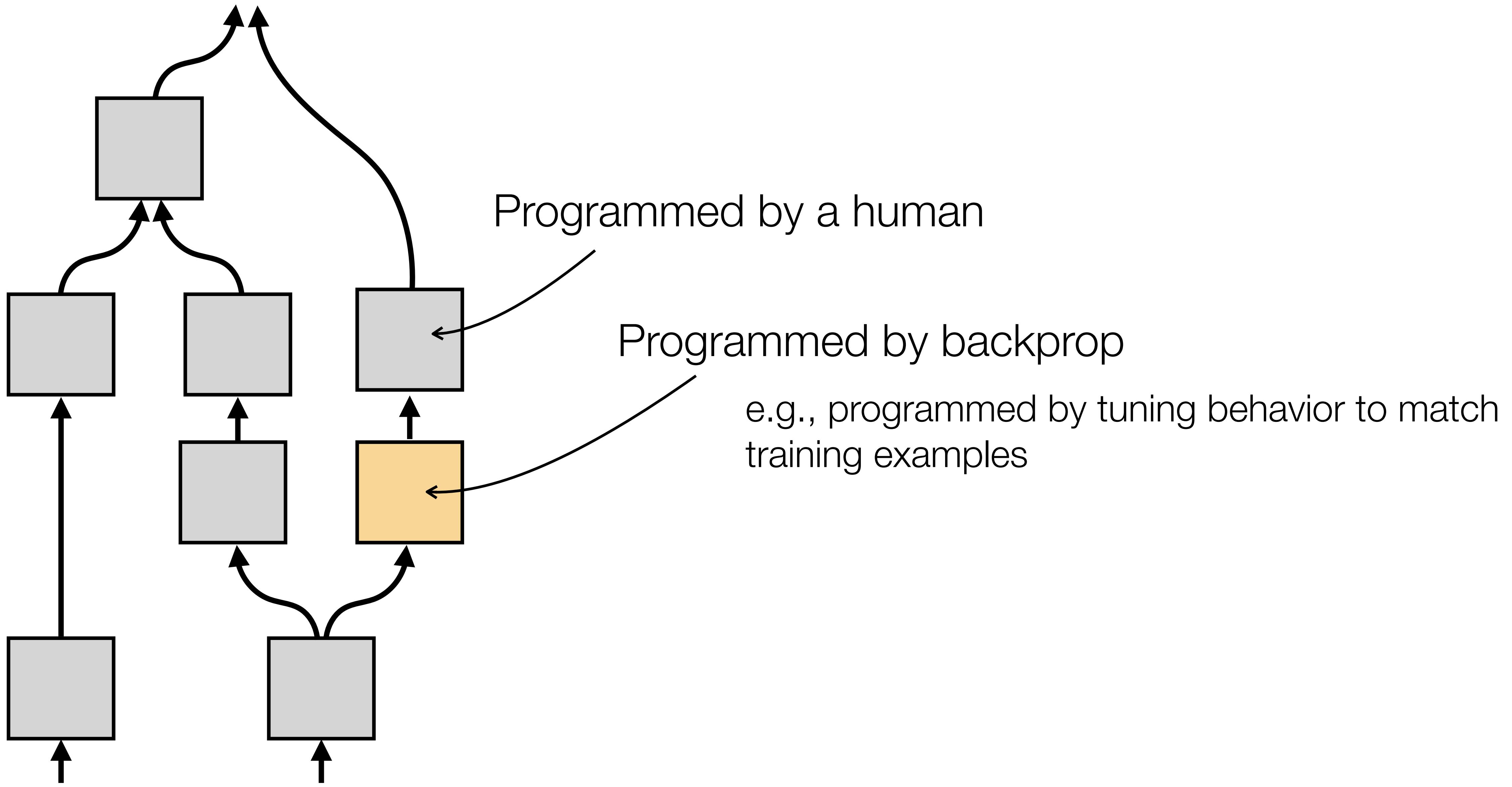


[Figure from “Neural Module Networks”, Andreas et al. 2017]

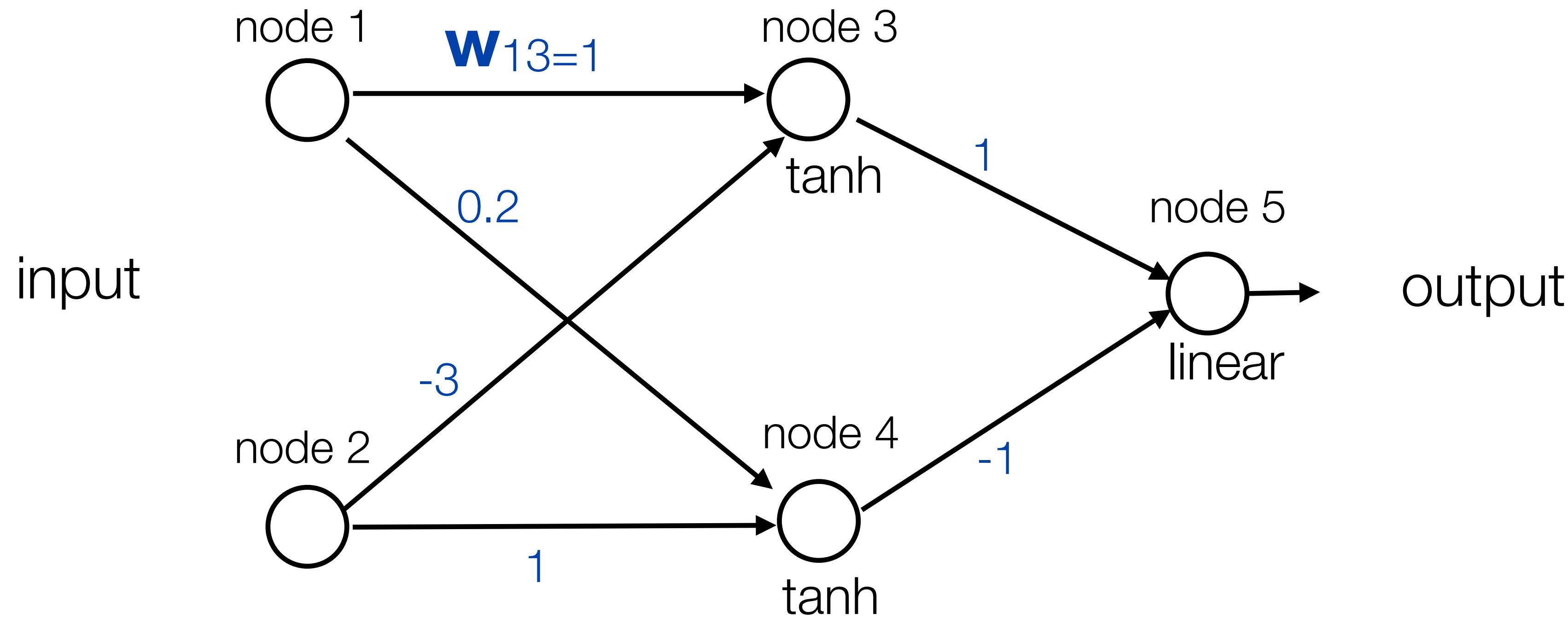
Software 2.0

[Andrej Karpathy: <https://karpathy.medium.com/software-2-0-a64152b37c35>]





Backpropagation example



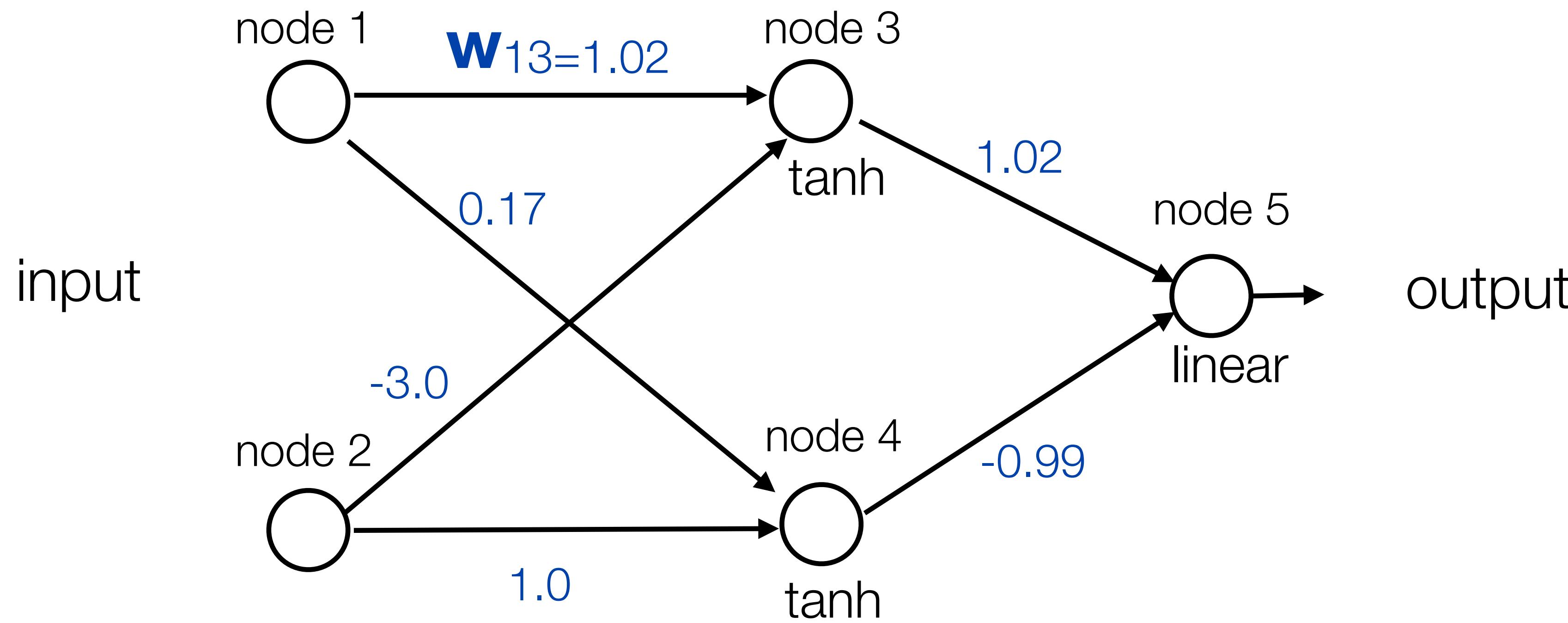
Learning rate $\eta = -0.2$ (because we used positive increments)

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

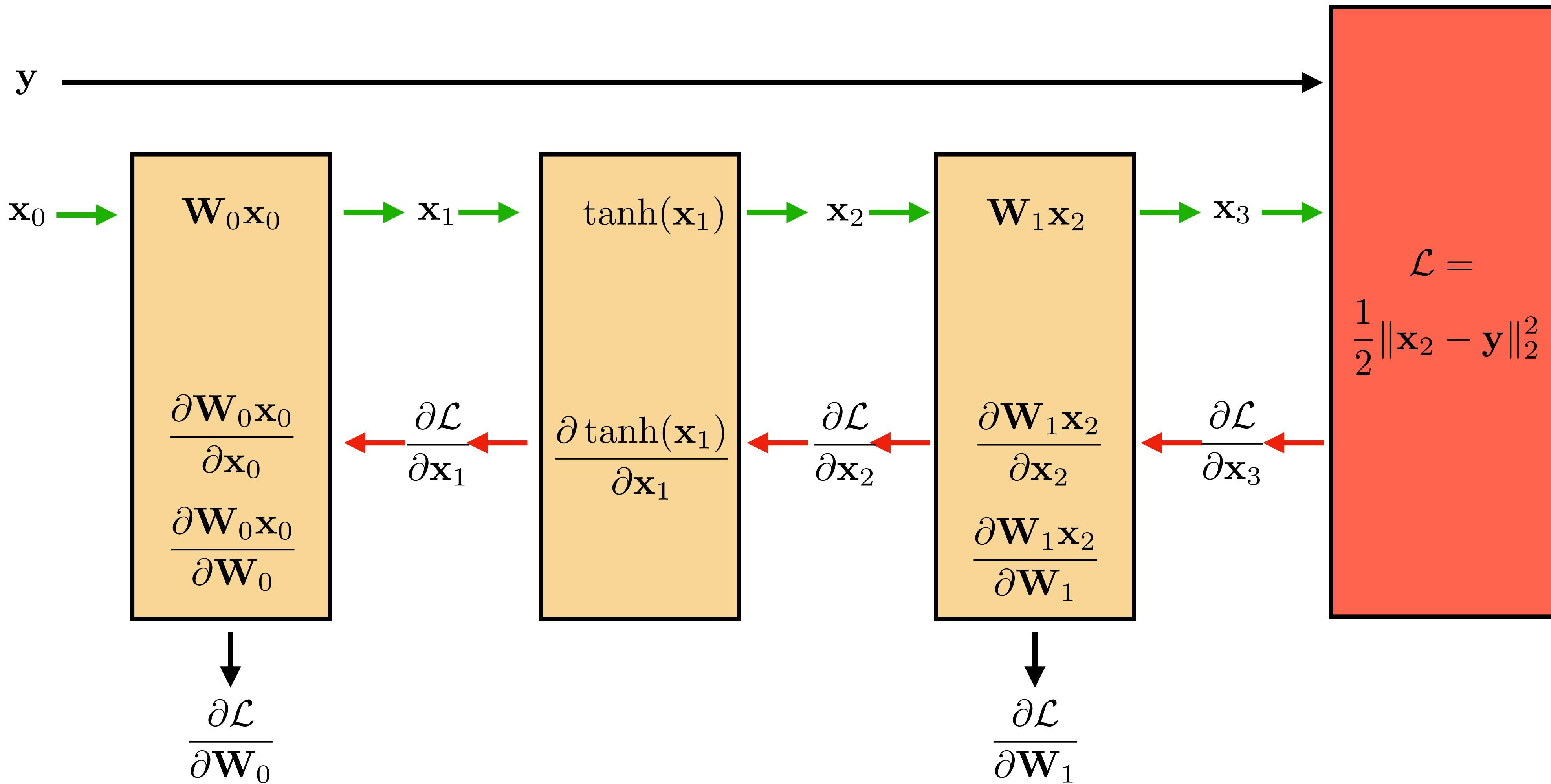
Backpropagation example



After one iteration (rounding to two digits)

Step by step solution

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.

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 \mathbf{W}^k are the weights at some iteration k of gradient descent given by the first slide:

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

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

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

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

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

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

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

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

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

Notice the ordering of the two terms being multiplied here. The notation hides the details but you can write out all the indices to see that this is the correct ordering — or just check that the dimensions work out.

The values for input vector \mathbf{x}_0 and target y are also given by the first slide:

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

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

Forward pass:

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

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

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

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

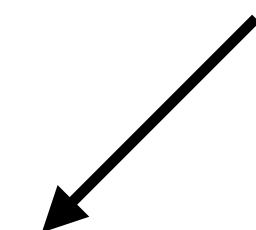
Backward pass:

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

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

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

diagonal matrix because tanh is a pointwise operation



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

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

Gradient updates:

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

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

11. Backpropagation

- Review of gradient descent
- Forward propagation
- Computing gradients
- Backward propagation
- Computation graphs and differentiable programming