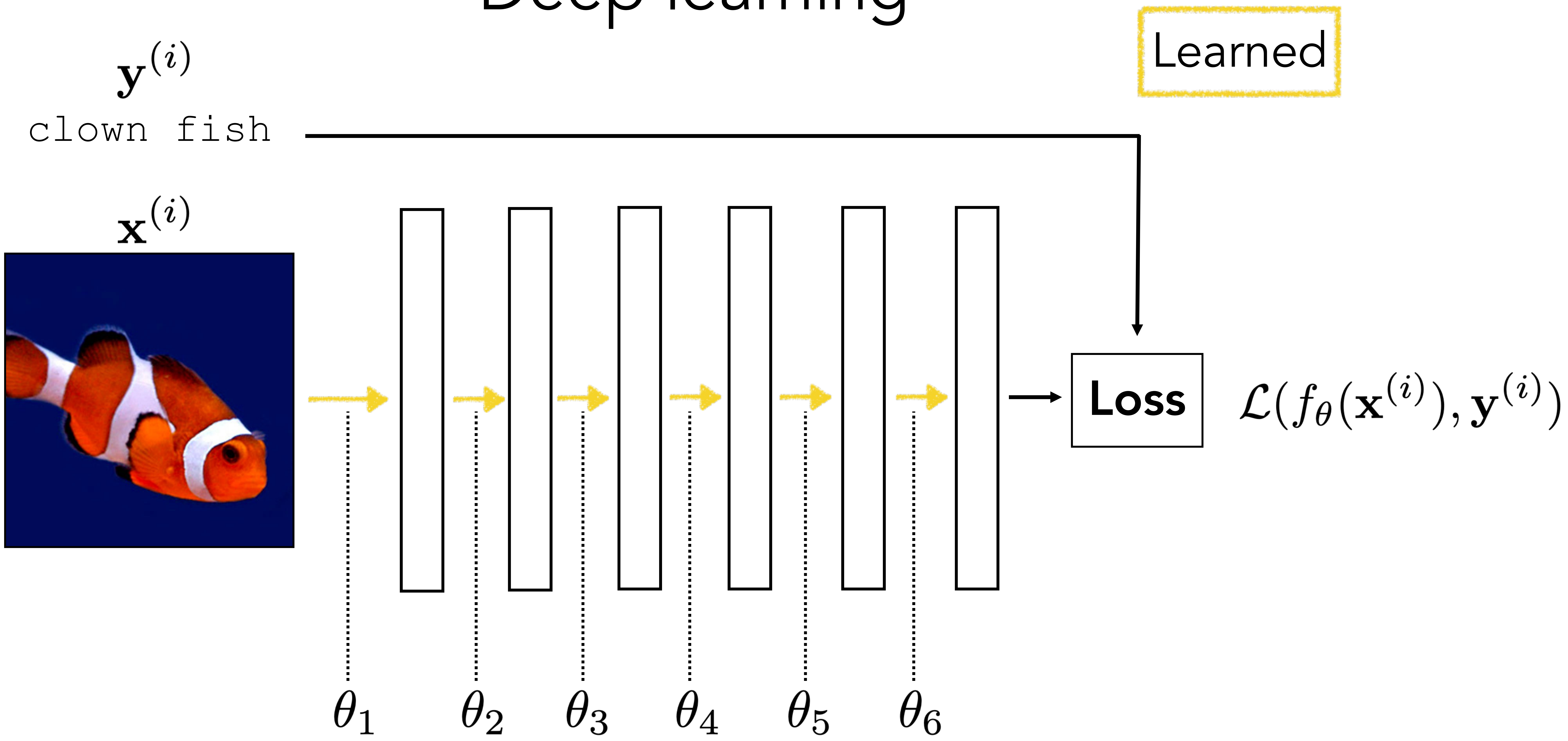
Lecture 8

Backpropagation

Backprop

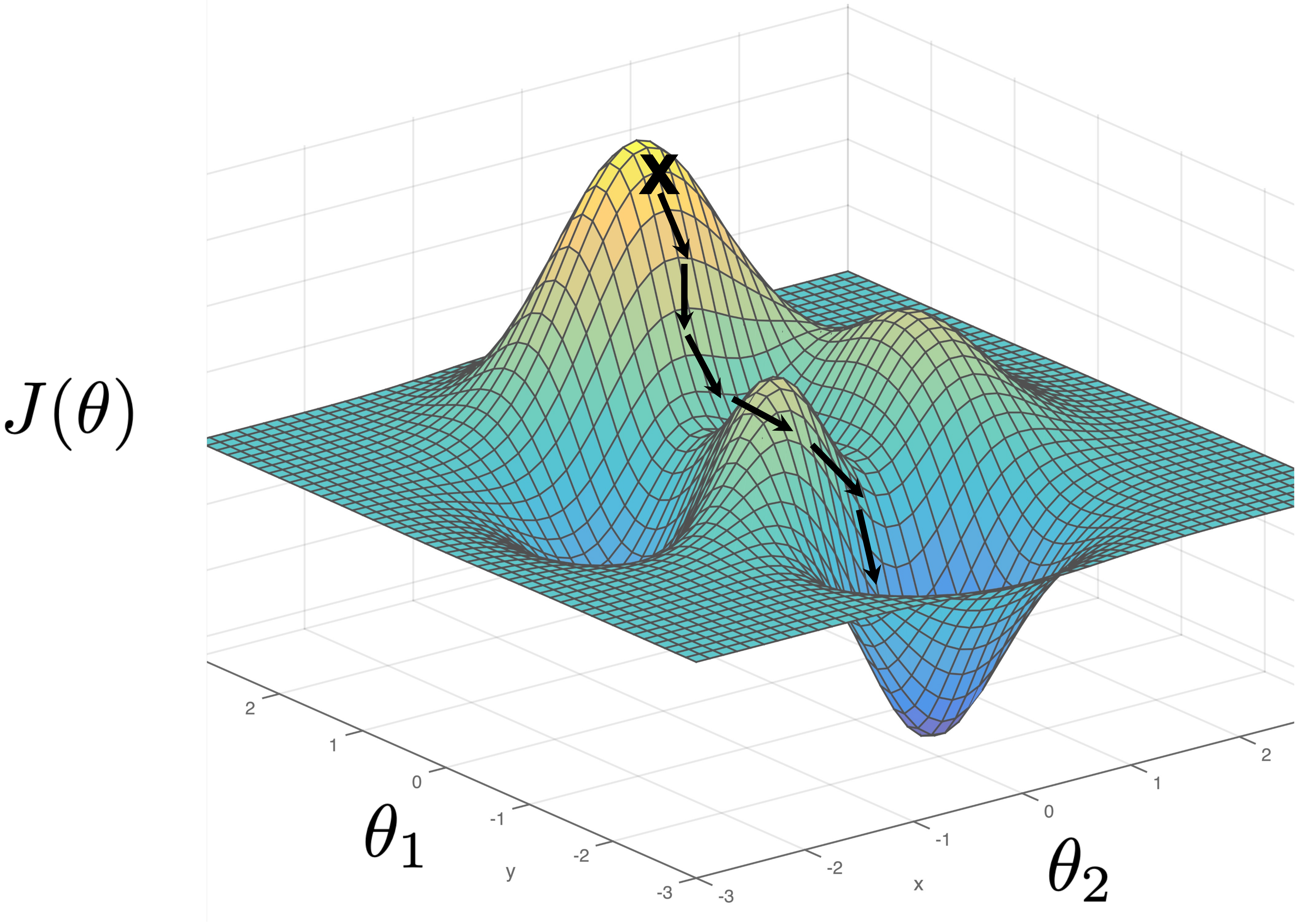
- Review of gradient descent, SGD
- Computation graphs
- Backprop through chains
- Backprop through MLPs
- Backprop through DAGs
- Optimization tricks
- Differentiable programming

Deep learning



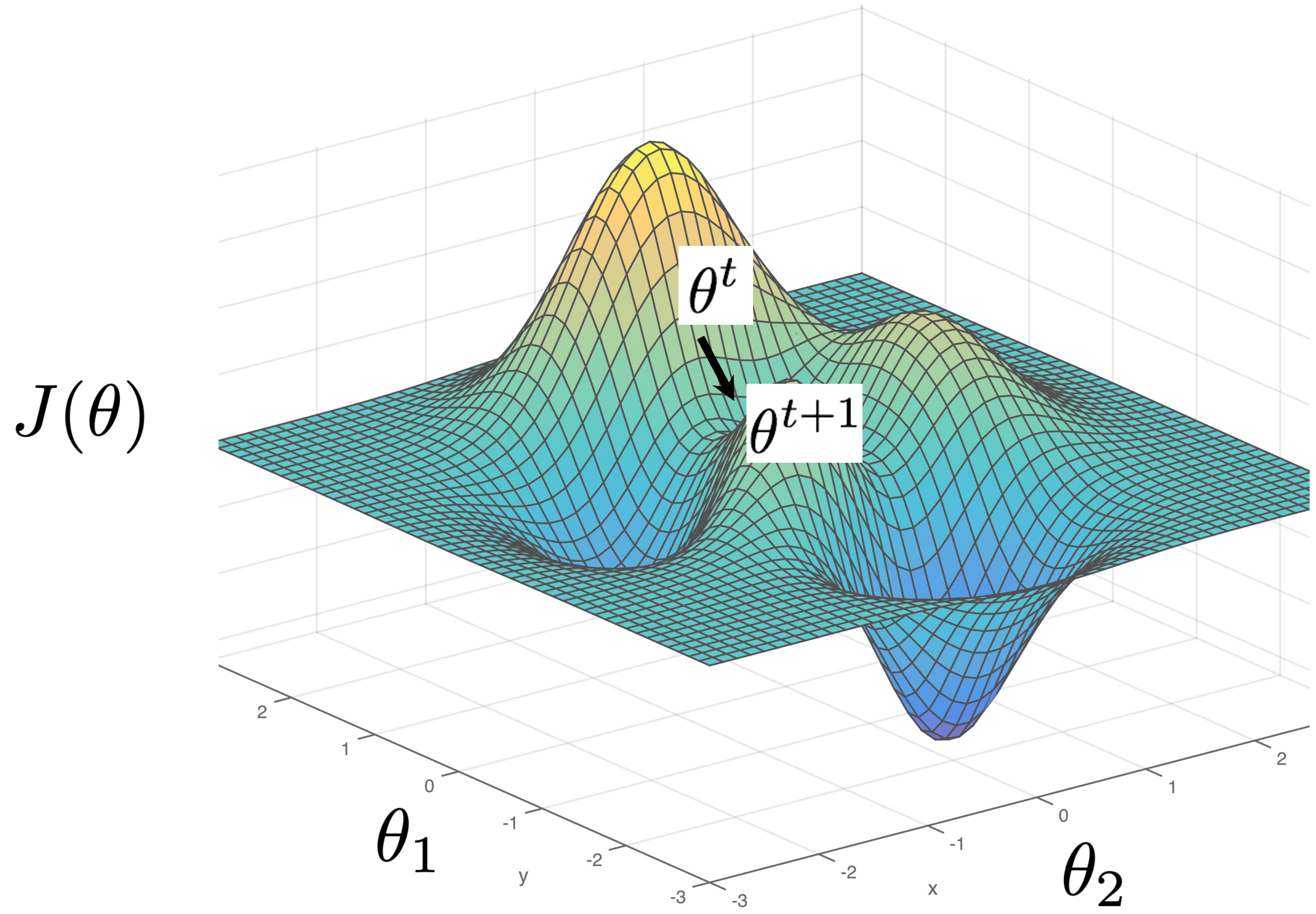
$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})$$

Gradient descent



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

Gradient descent



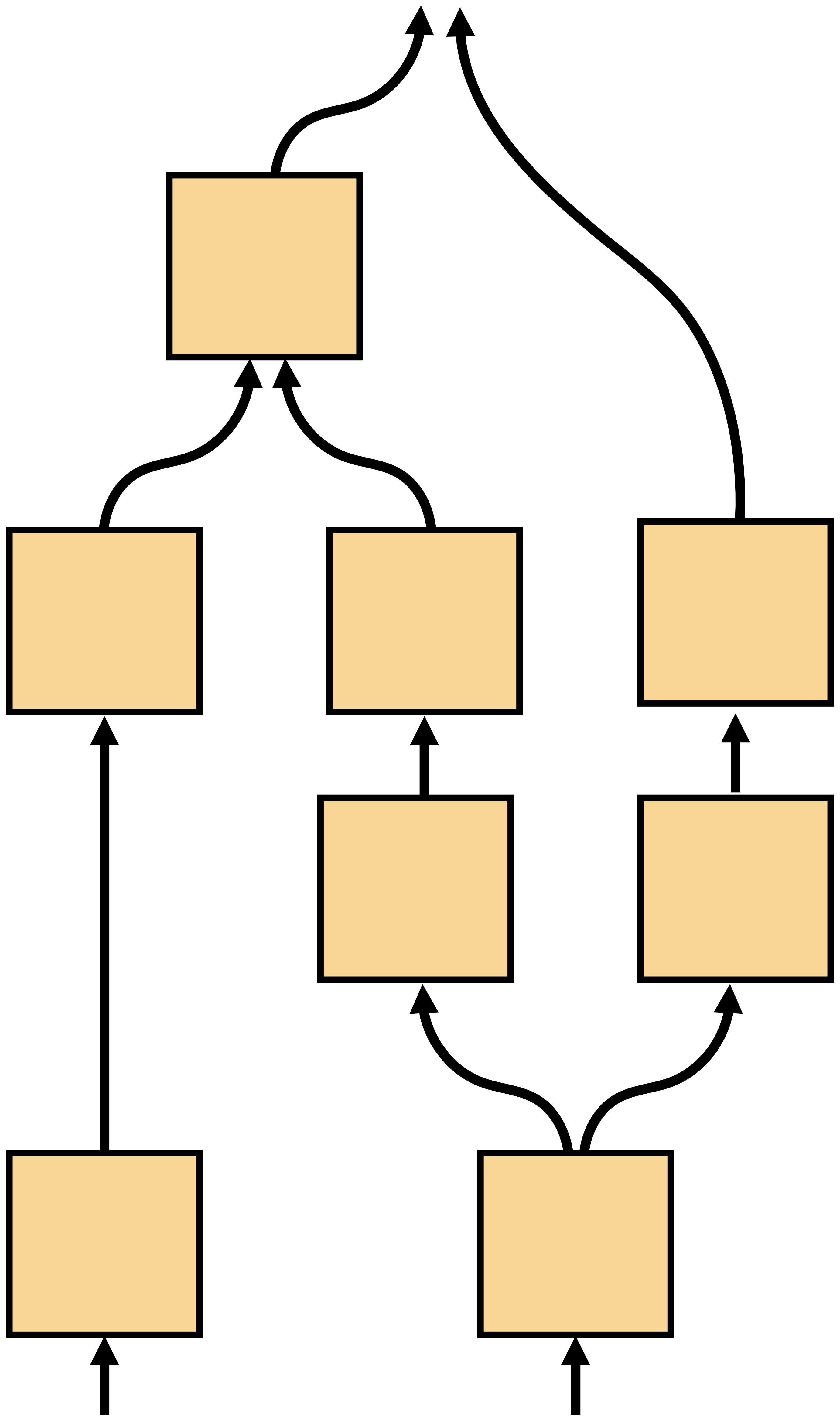
$$\boxed{\theta^* = \arg\min_{\theta} \underbrace{\sum_{i=1}^{N} \mathcal{L}(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)})}_{J(\theta)}}$$

One iteration of gradient descent:

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

learning rate

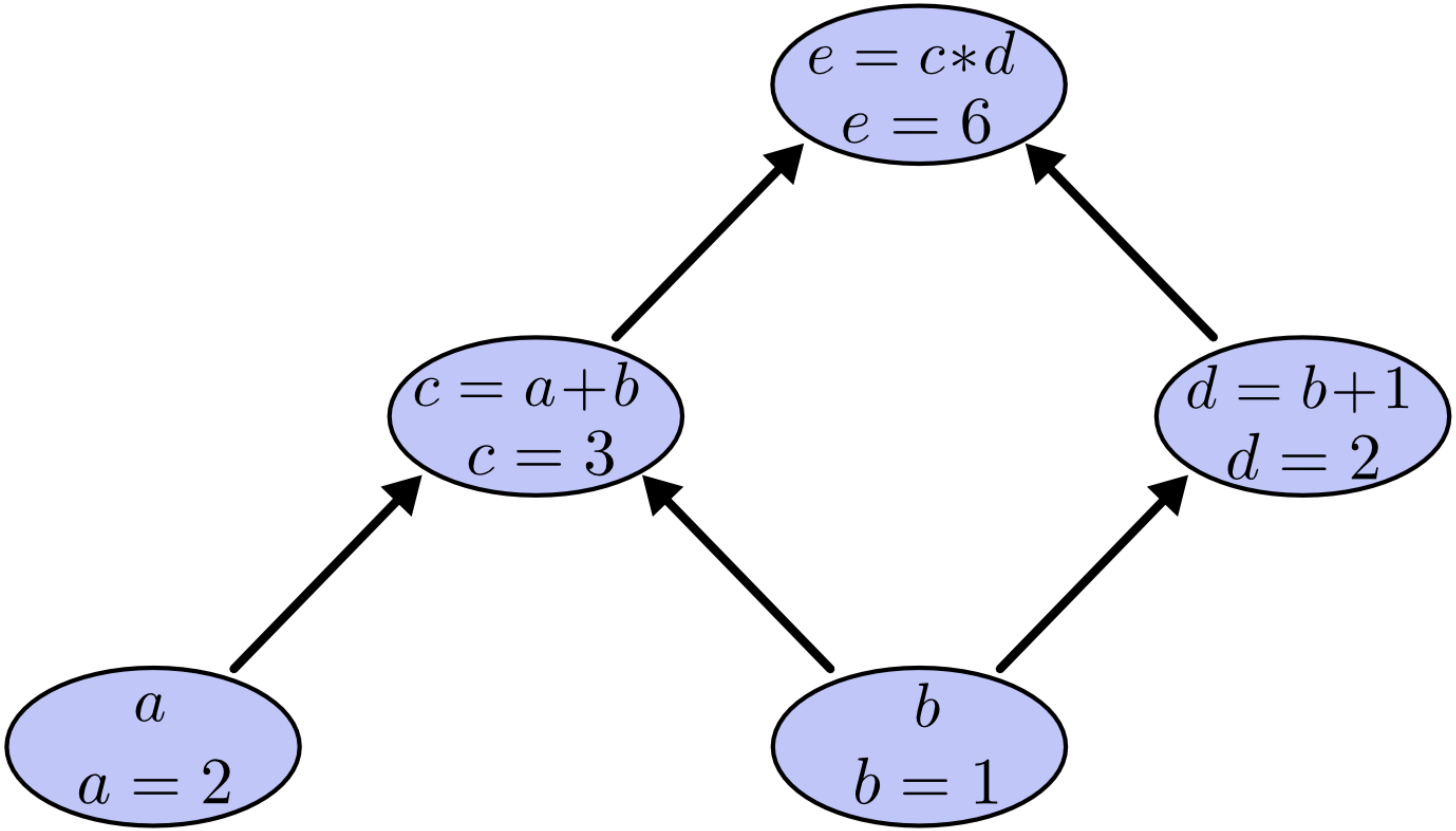
Computation Graphs



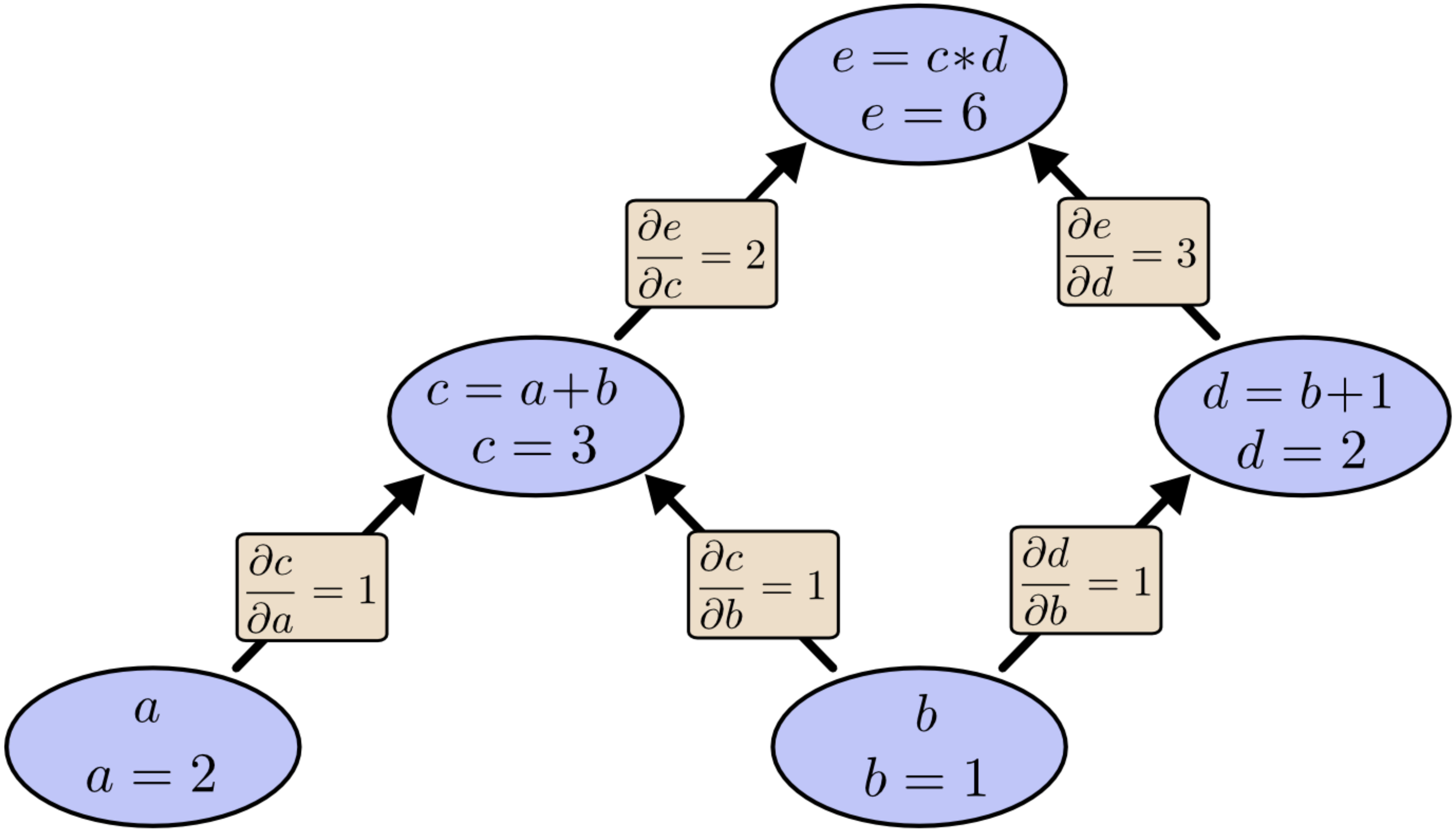
A graph of functional transformations, nodes ( ), that when strung together perform some useful computation.

Deep learning deals (primarily) with computation graphs that take the form of **directed acyclic graphs** (DAGs), and for which each node is differentiable.

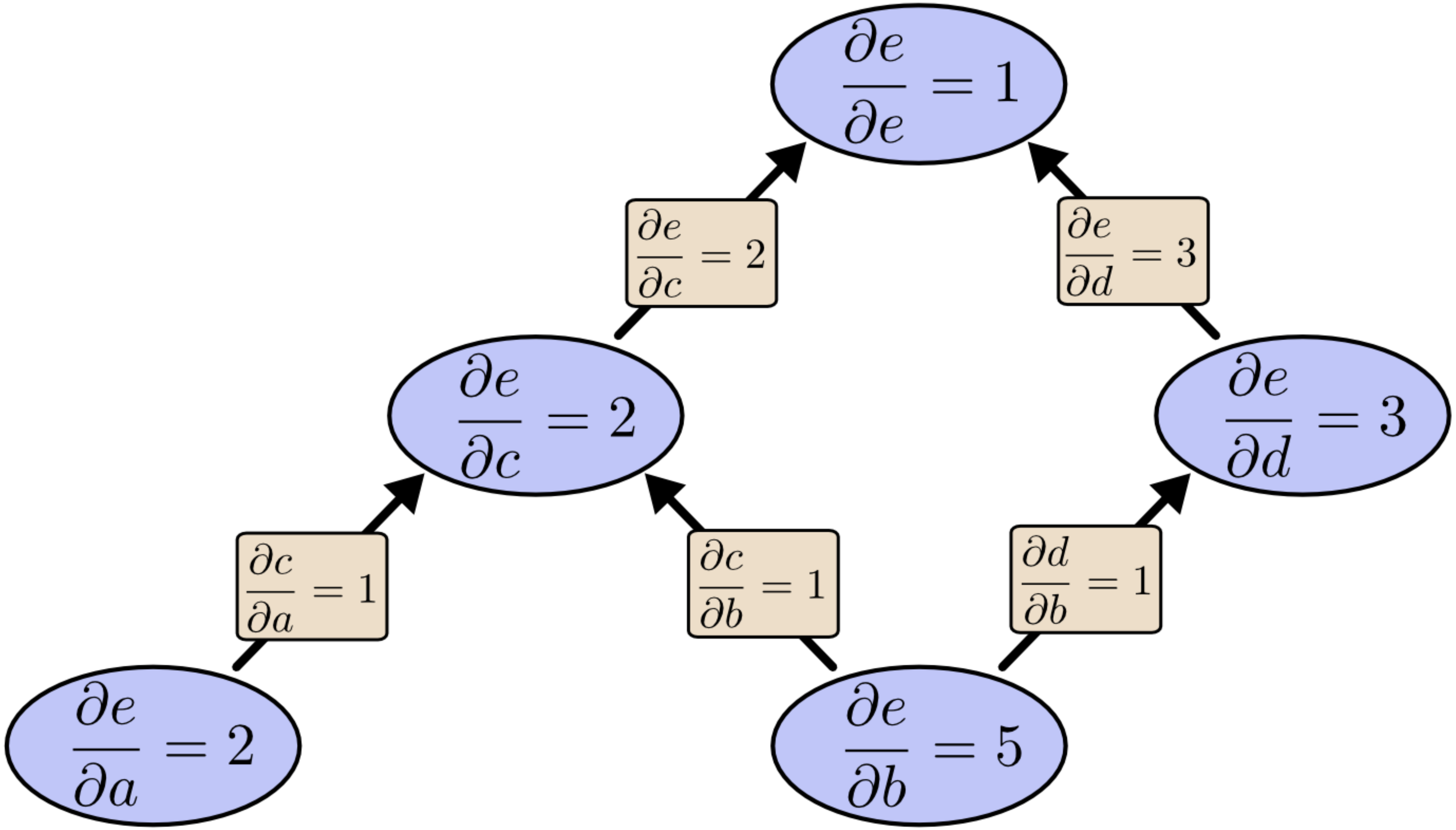
A Simple Example



A Simple Example



A Simple Example



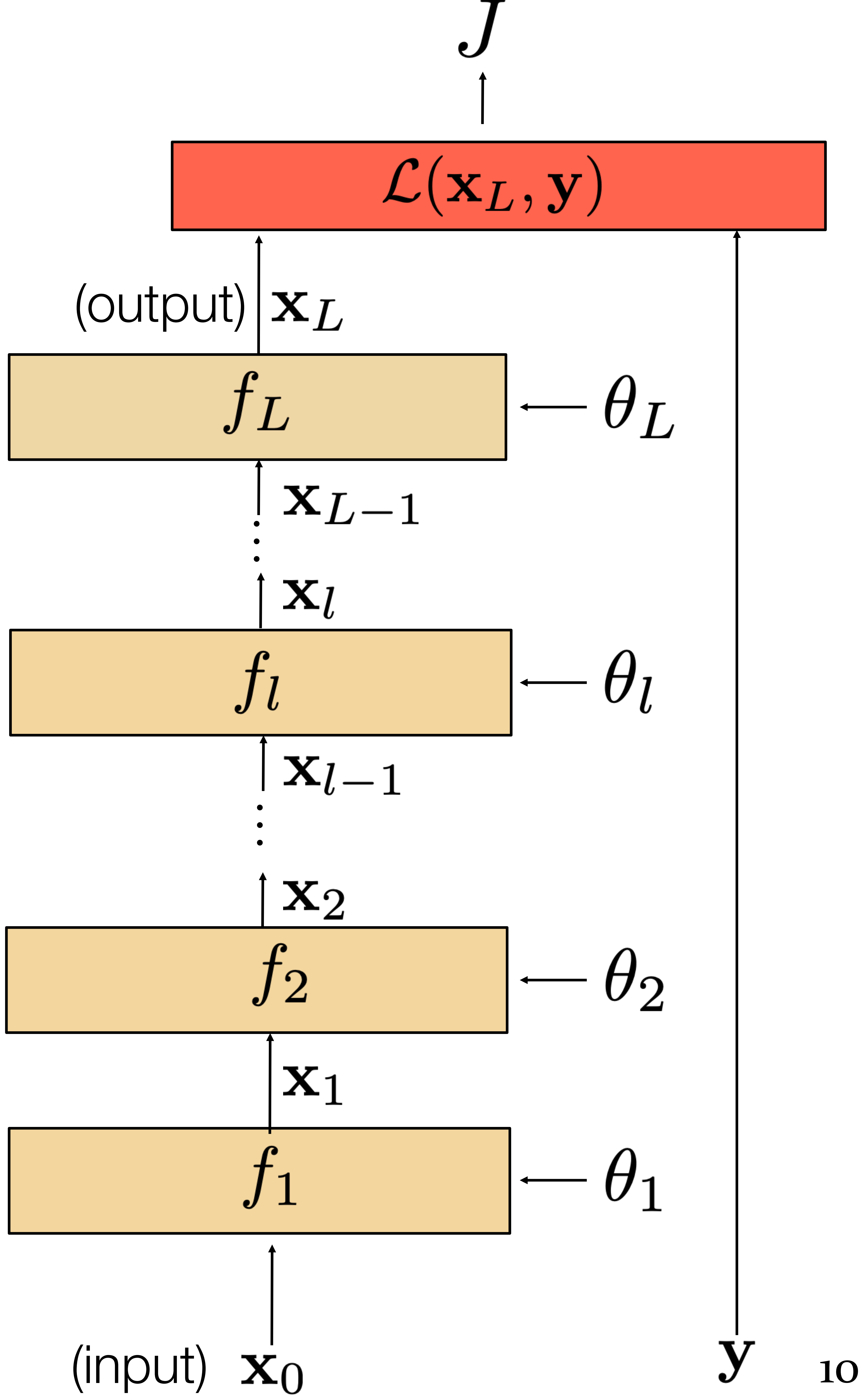
Chains

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

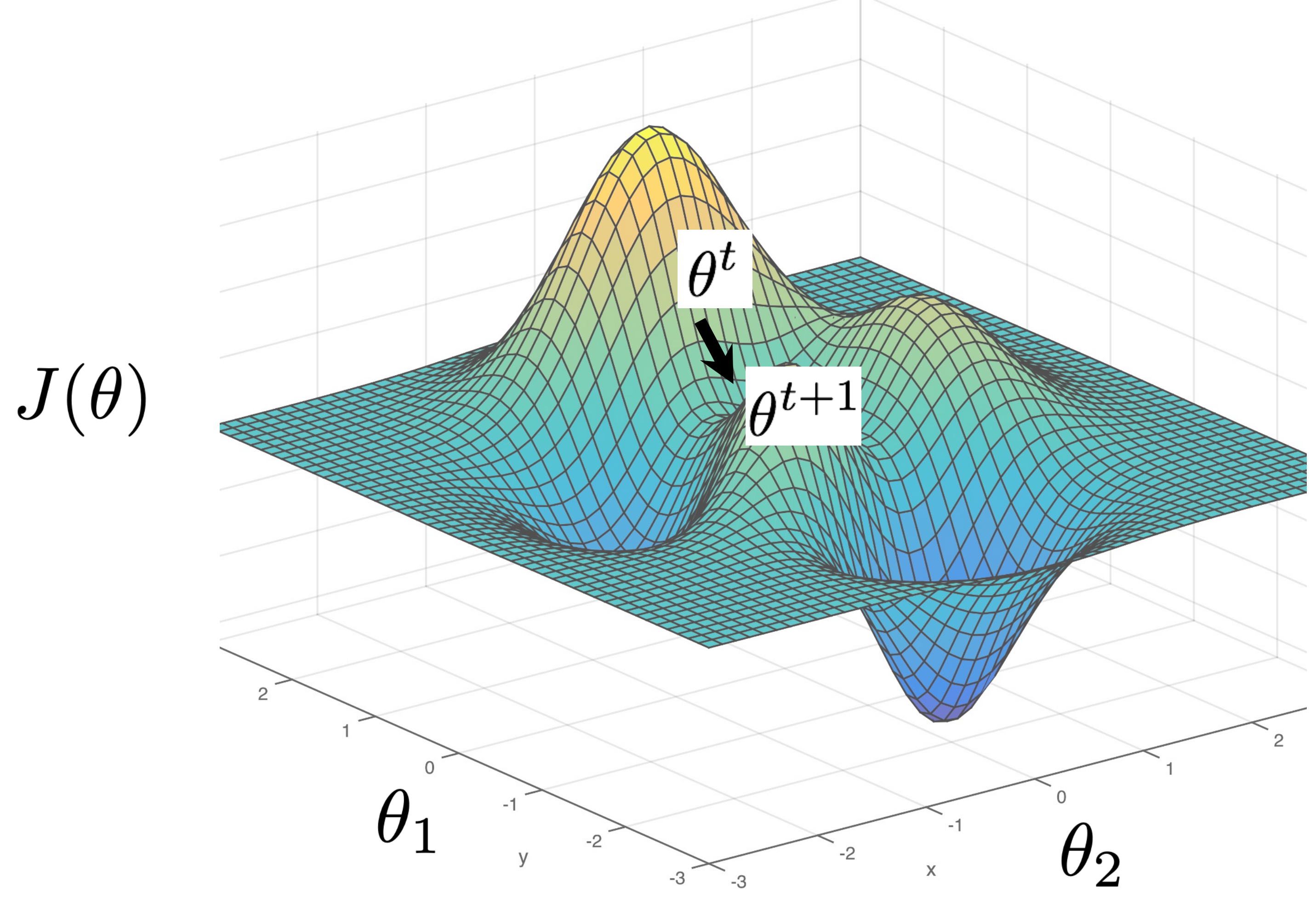
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \theta_l)$$

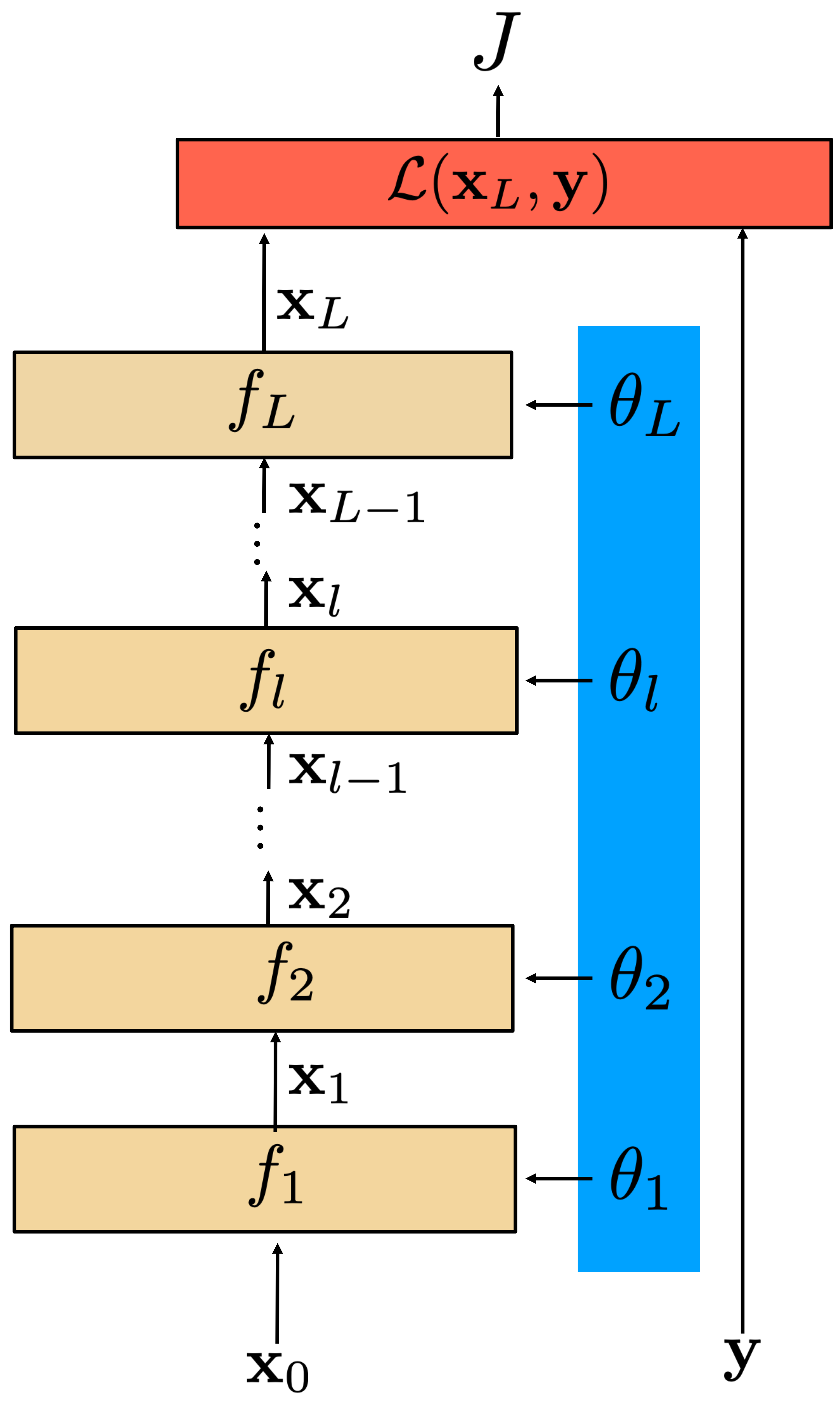
- An example of such a computation graph is an MLP
- **Loss function** $\mathcal{L}$ compares $\mathbf{x}_L$ to $\mathbf{y}$
- Overall cost is the sum of the losses over all training examples:

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



Gradient descent





- We need to compute gradients of the cost with respect to model parameters.
- By design, each layer will be differentiable with respect to its inputs (the inputs are the data and parameters)

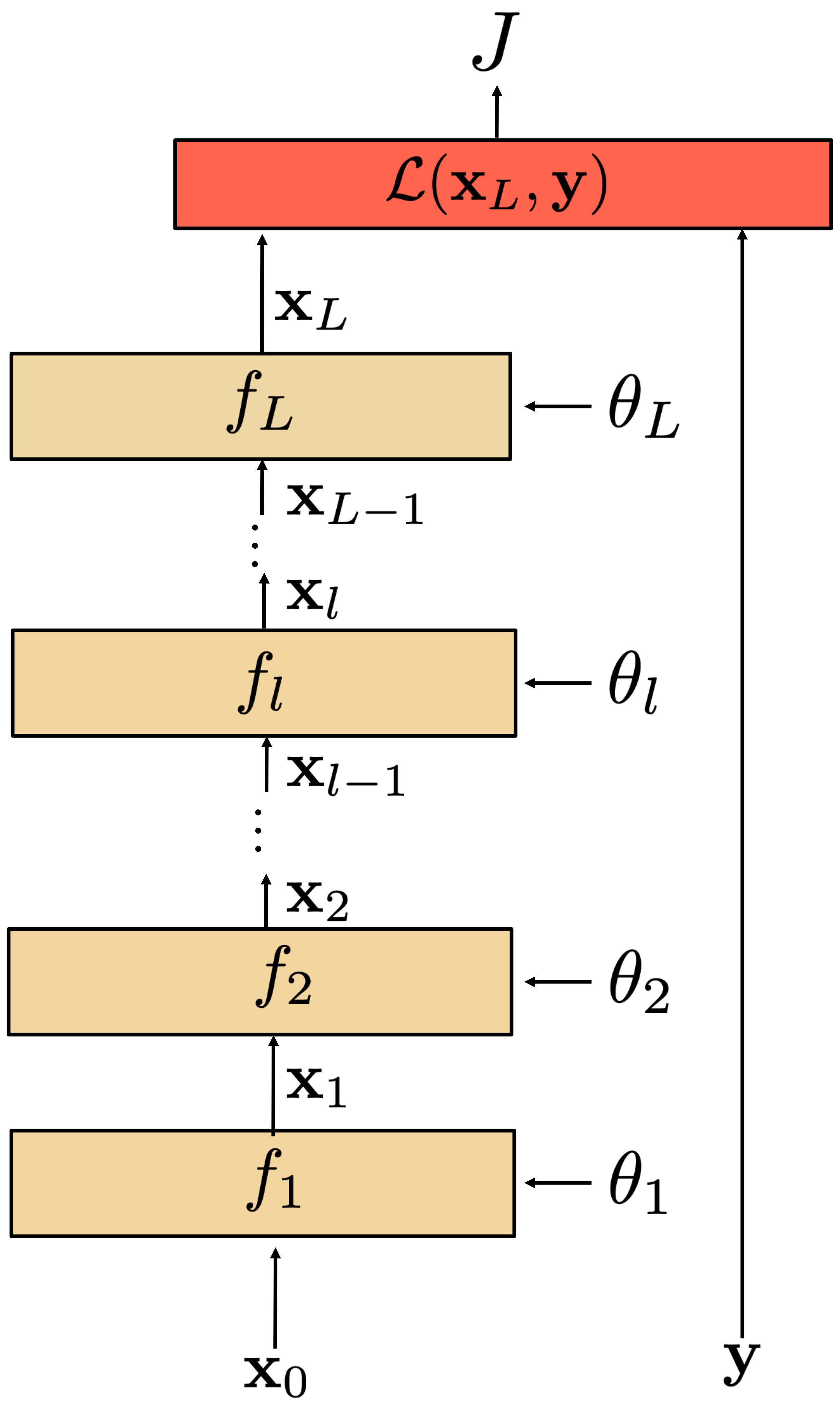
Computing gradients

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

$$J(\theta) = \sum_{i=1} \mathcal{L}(f_L(\ldots f_2(f_1(\mathbf{x}_0^{(i)}, \theta_1), \theta_2), \ldots \theta_L), \mathbf{y}^{(i)})$$

And then evaluate each partial derivatives separately…

$$\frac{\partial J}{\partial \theta_l}$$



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

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

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)

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

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

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

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

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

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

)

Computing gradients

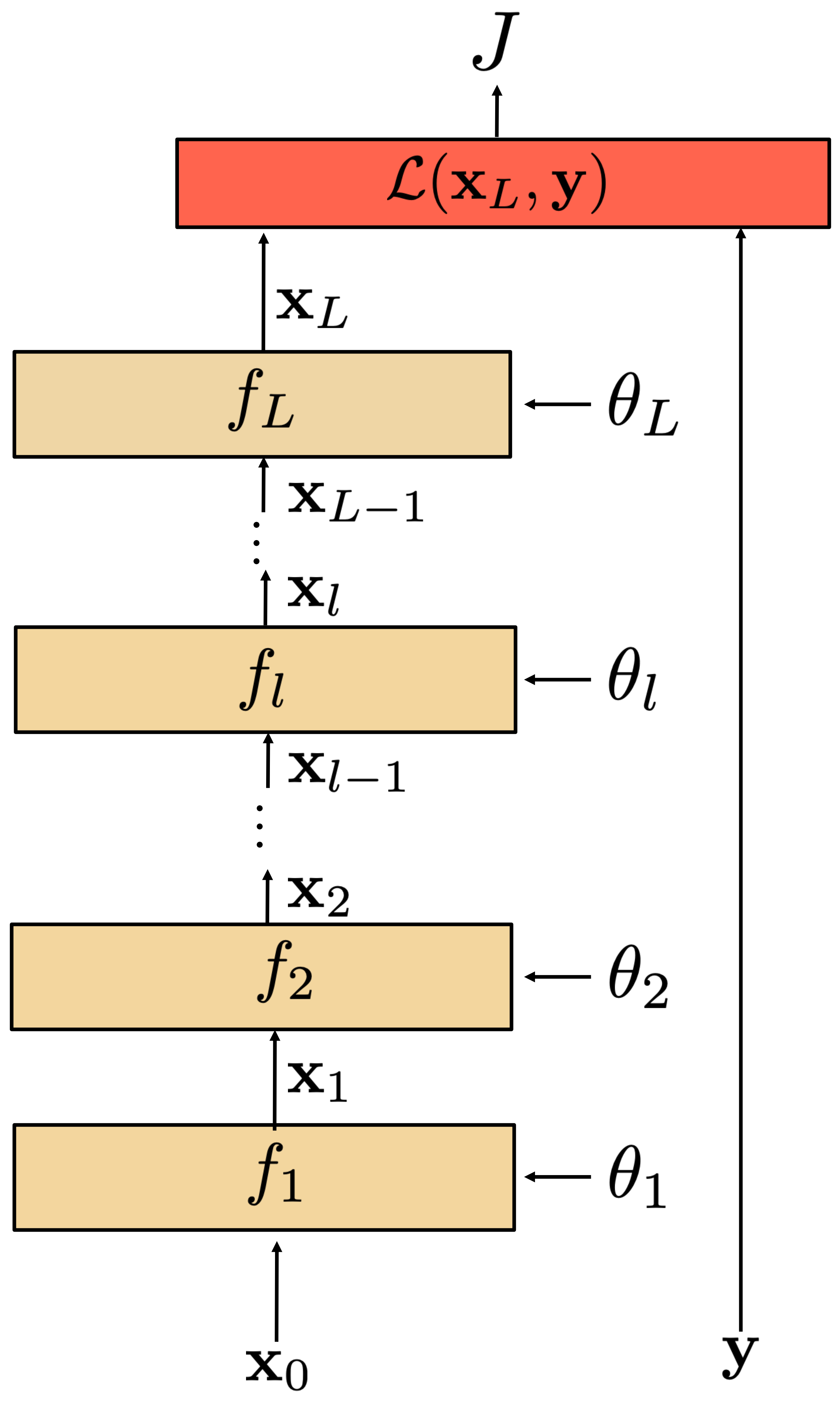
The loss J is the sum of the losses associated with each training example

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

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

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

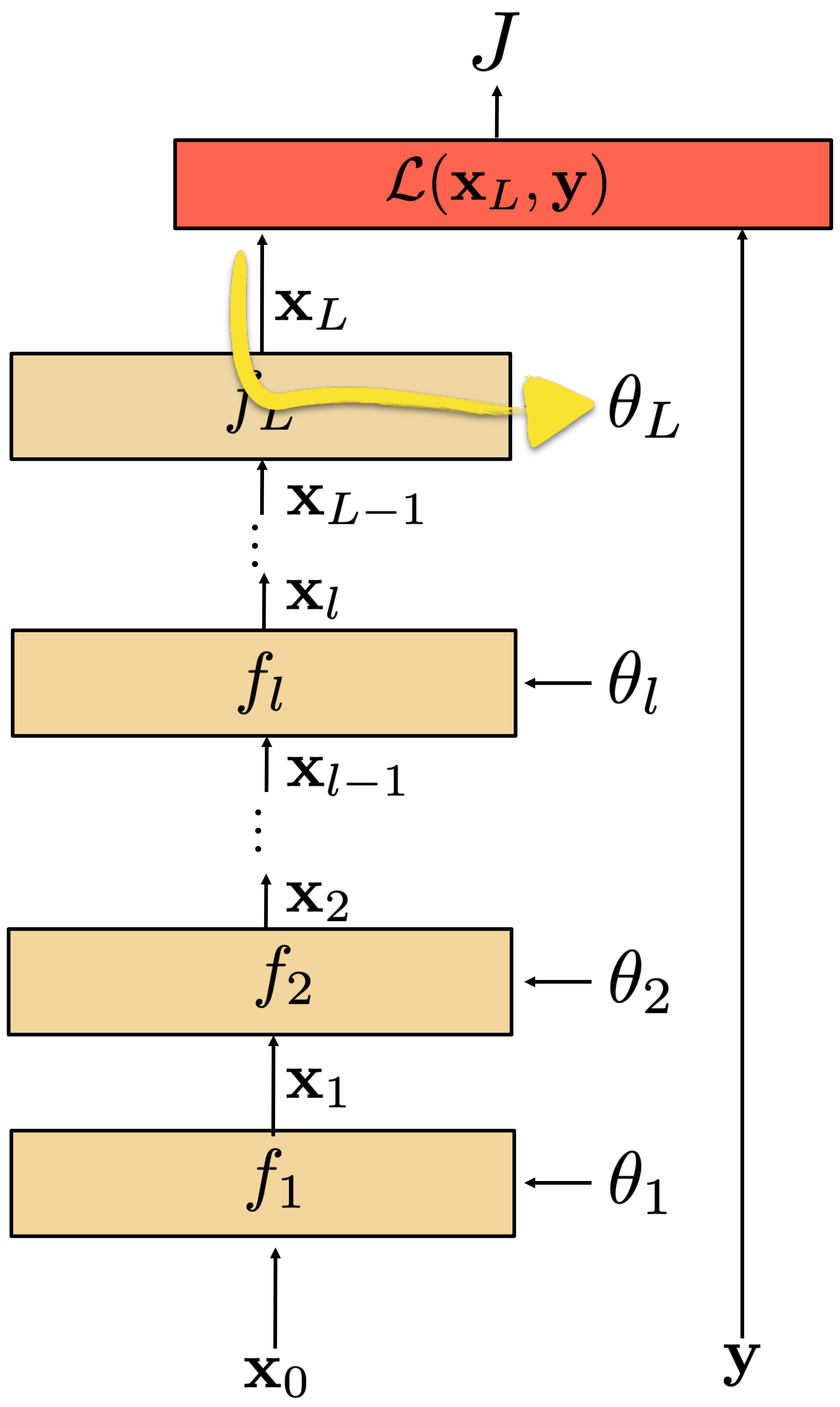
Aka how much J varies when the parameter θ_i is varied.



Computing gradients

To compute the parameter update for the last layer, we can use the **chain rule**:

$$\boxed{\frac{\partial J}{\partial \theta_L} = \frac{\partial J}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \theta_L}}$$



How much the loss changes when we change θ_i?

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.

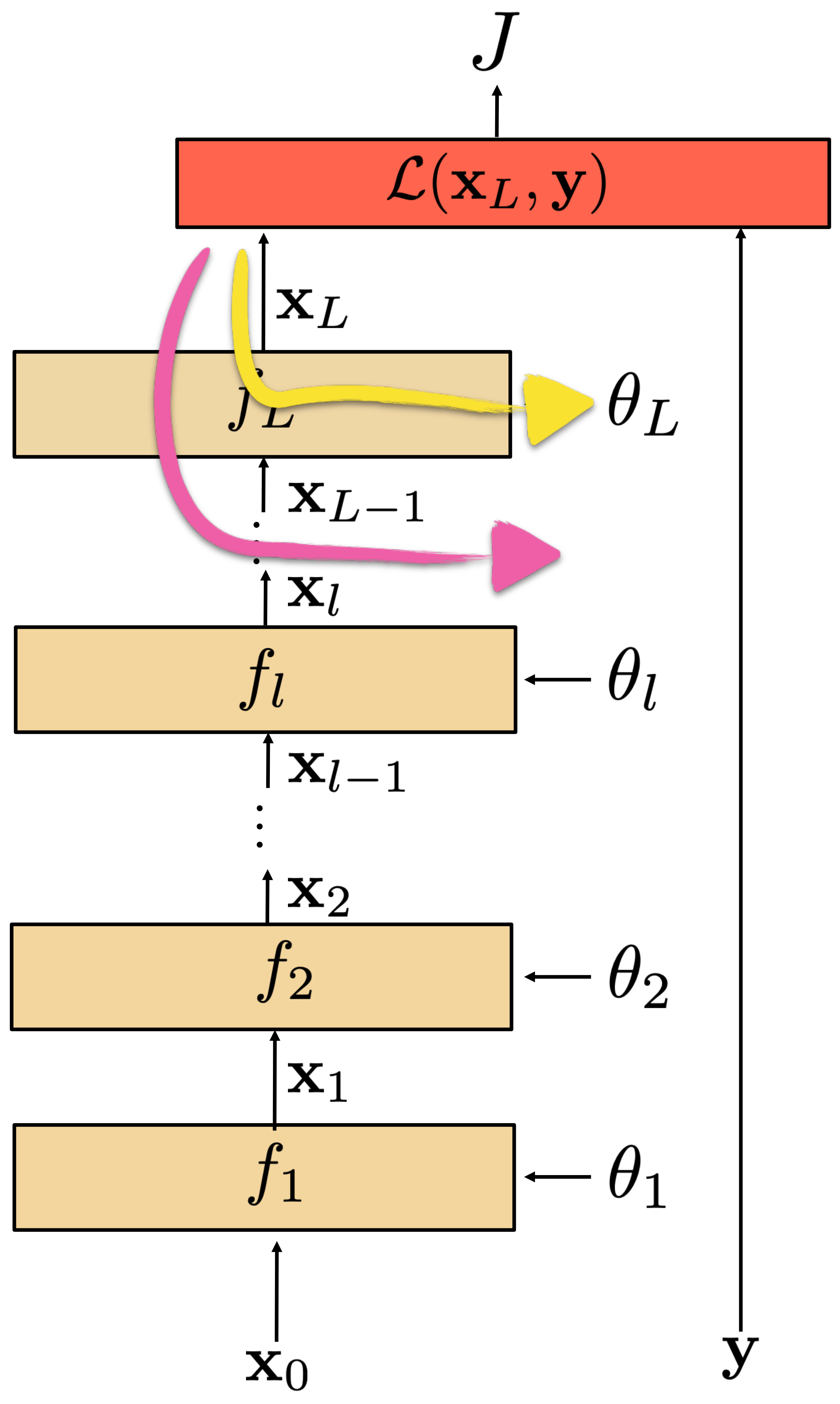
Computing gradients

To compute the parameter update for the last layer, we can use the **chain rule**:

$$\boxed{\frac{\partial J}{\partial \theta_L} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \theta_L}}$$

To compute the parameter update for the second-to-last layer:

$$\boxed{\frac{\partial J}{\partial \theta_{L-1}} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \frac{\partial \mathbf{x}_{L-1}}{\partial \theta_{L-1}}}$$

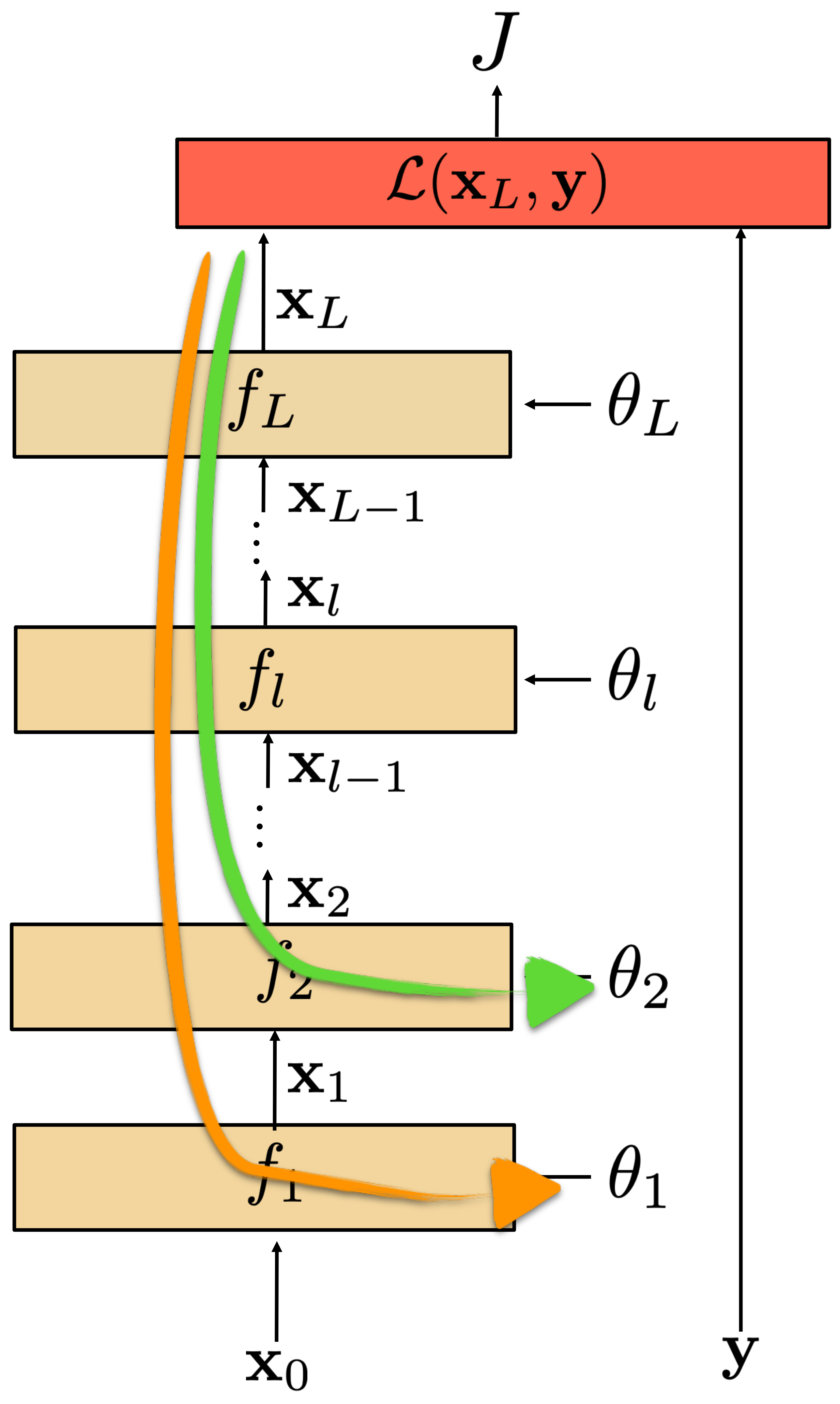


Computing gradients

To compute the parameter update for the 2nd and 1st layers:

$$\boxed{\frac{\partial J}{\partial \theta_2} = \frac{\partial J}{\partial \mathbf{x}_L} \cdots \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \theta_2}}$$

$$\boxed{\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L} \cdots \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \theta_1}}$$



Blue terms are all shared! Can compute that product once and share it between these two equations.

The trick of backpropagation — reuse of computation
(aka dynamic programming)

Gradient w.r.t. loss at layer L-1

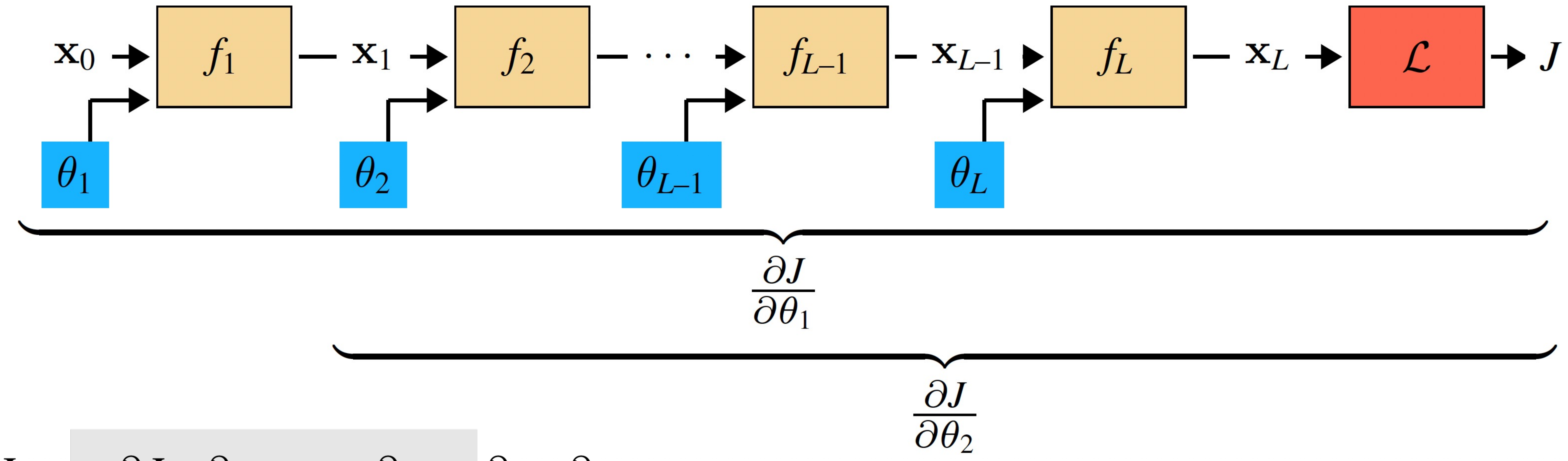
$$\frac{\partial J}{\partial \mathbf{x}_{L-1}} = \frac{\partial J}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}}$$

Layer L's gradient

Gradient w.r.t. loss at layer L

The trick of backpropagation — reuse of computation

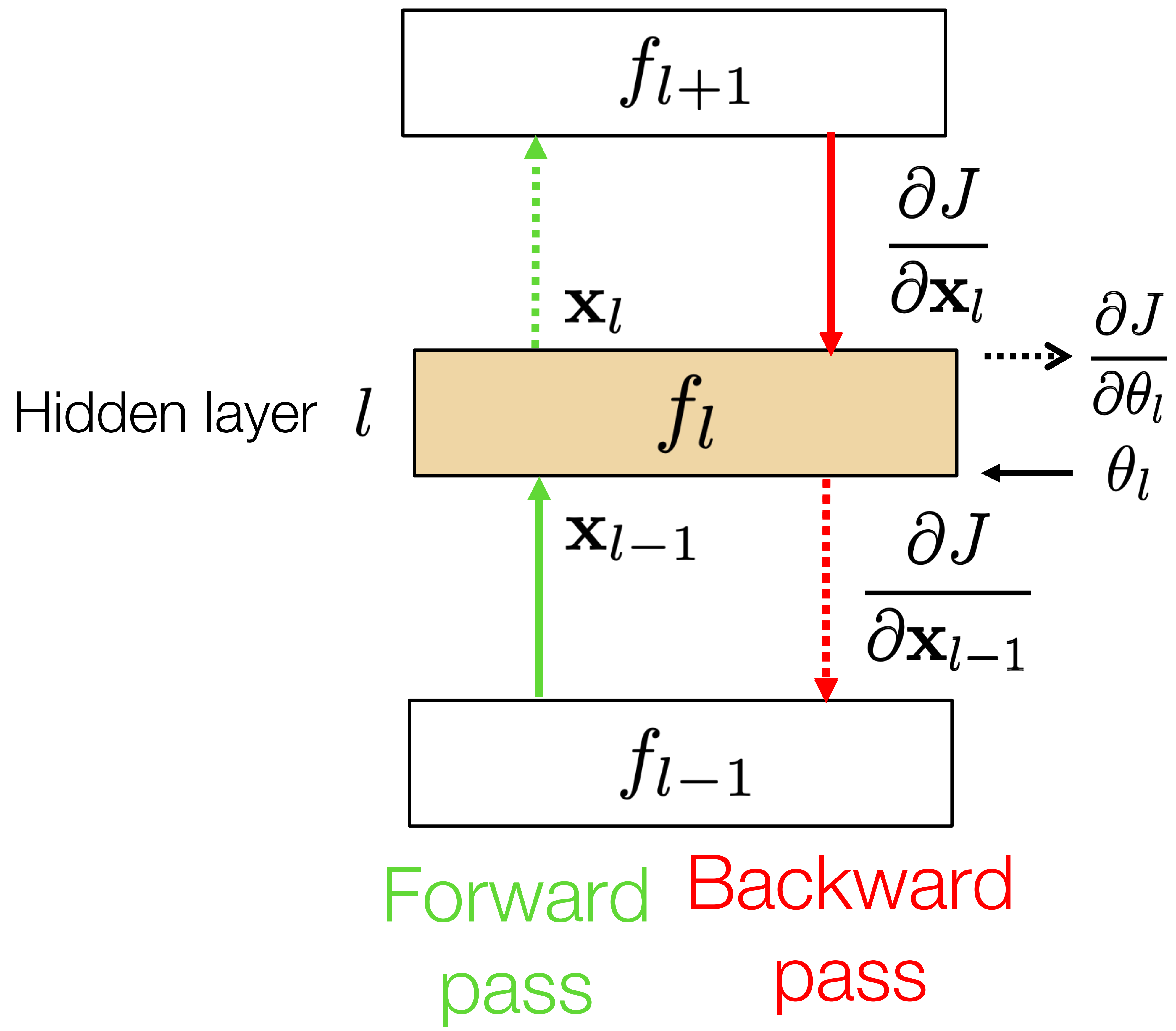
(aka dynamic programming)



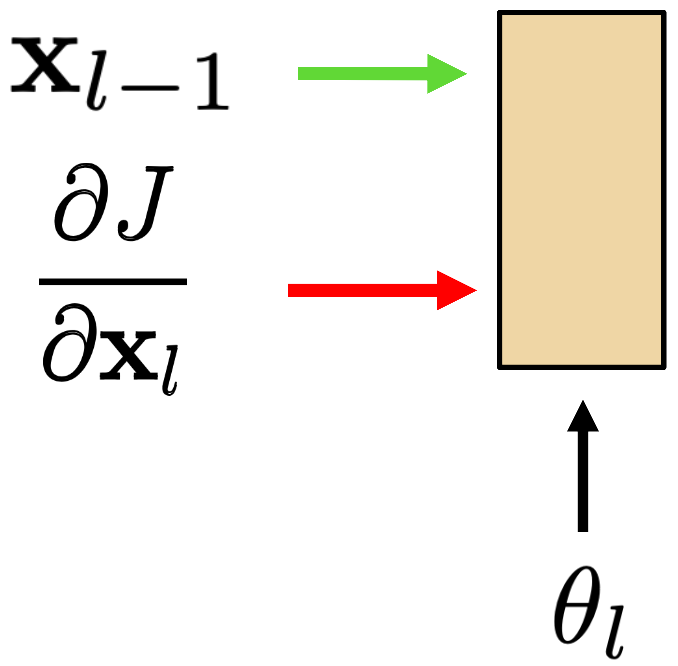
$$\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \theta_1}$$

$$\frac{\partial J}{\partial \theta_2} = \frac{\partial J}{\partial \mathbf{x}_L} \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdots \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \theta_2}$$

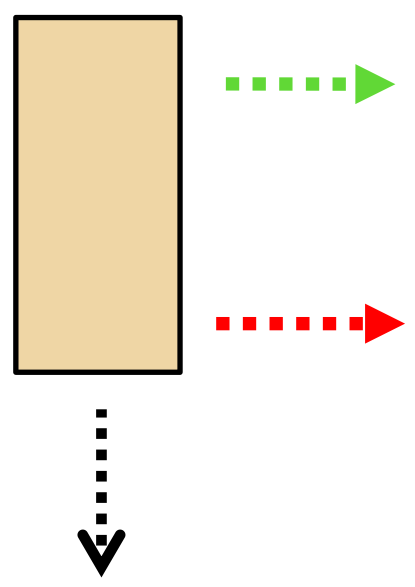
Backpropagation — Goal: to update parameters of layer l



- Layer l has three inputs (during training)



- And three outputs



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

$$\frac{\partial J}{\partial \mathbf{x}_{l-1}} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \mathbf{x}_{l-1}}$$

$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \theta_l}$$

- Given the inputs, we just need to evaluate:

$$f_l \qquad \frac{\partial f_l}{\partial \mathbf{x}_{l-1}} \qquad \frac{\partial f_l}{\partial \theta_l}$$

Backpropagation Summary

1. **Forward pass:** for each training example, compute the outputs for all layers:

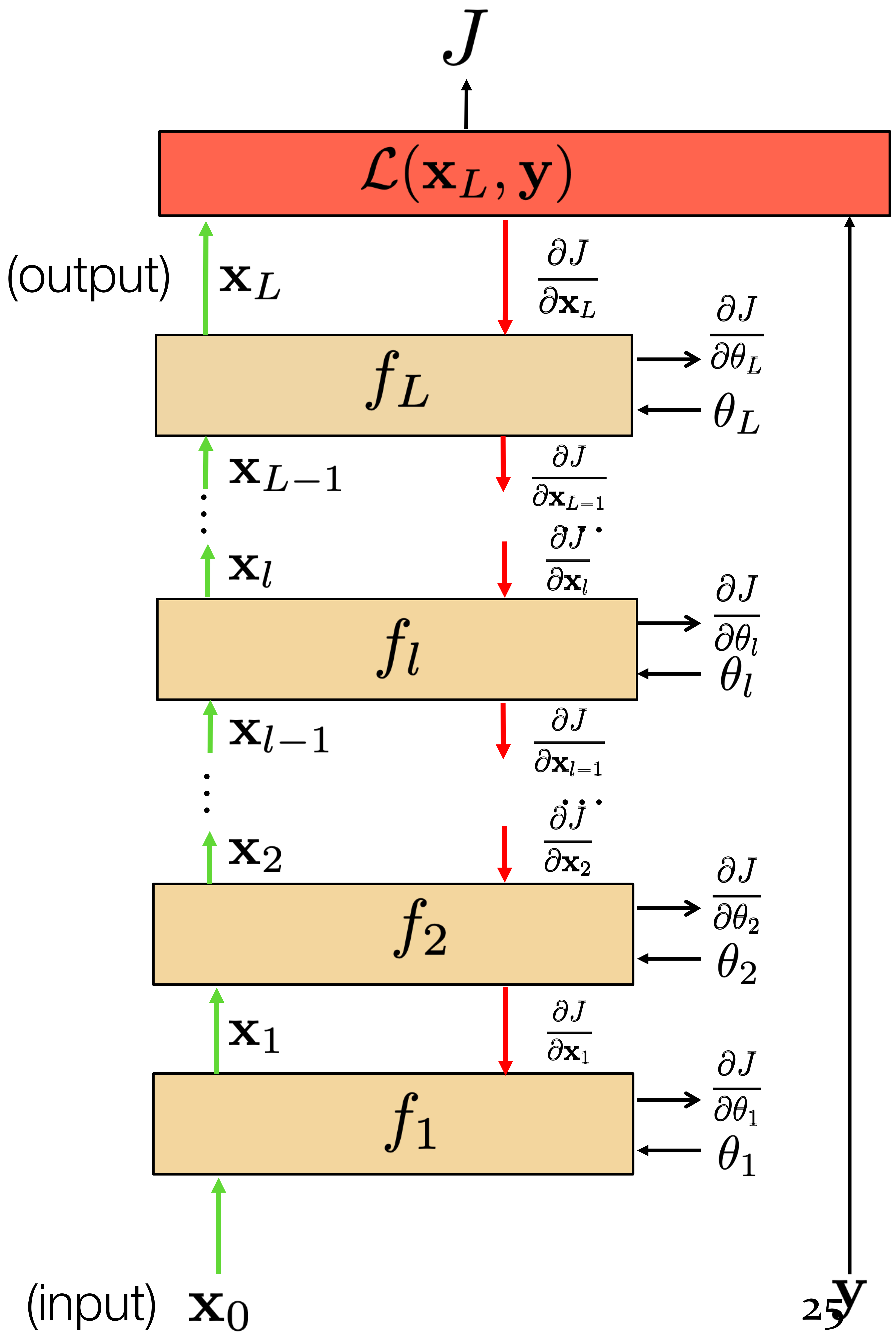
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \theta_l)$$

2. **Backwards pass:** compute loss derivatives iteratively from top to bottom:

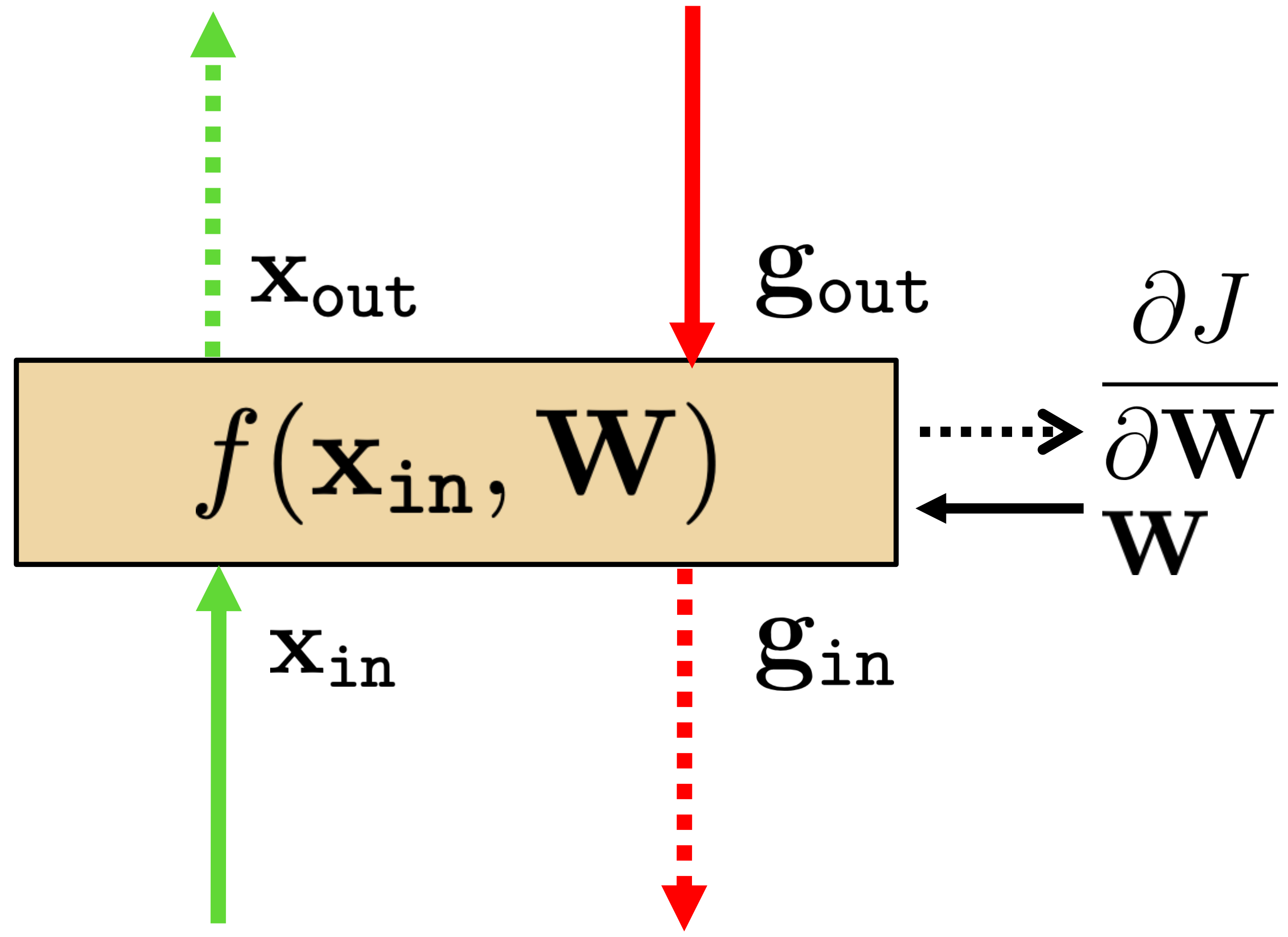
$$\frac{\partial J}{\partial \mathbf{x}_{l-1}} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \mathbf{x}_{l-1}}$$

3. **Parameter update:** Compute gradients w.r.t. weights, and update weights:

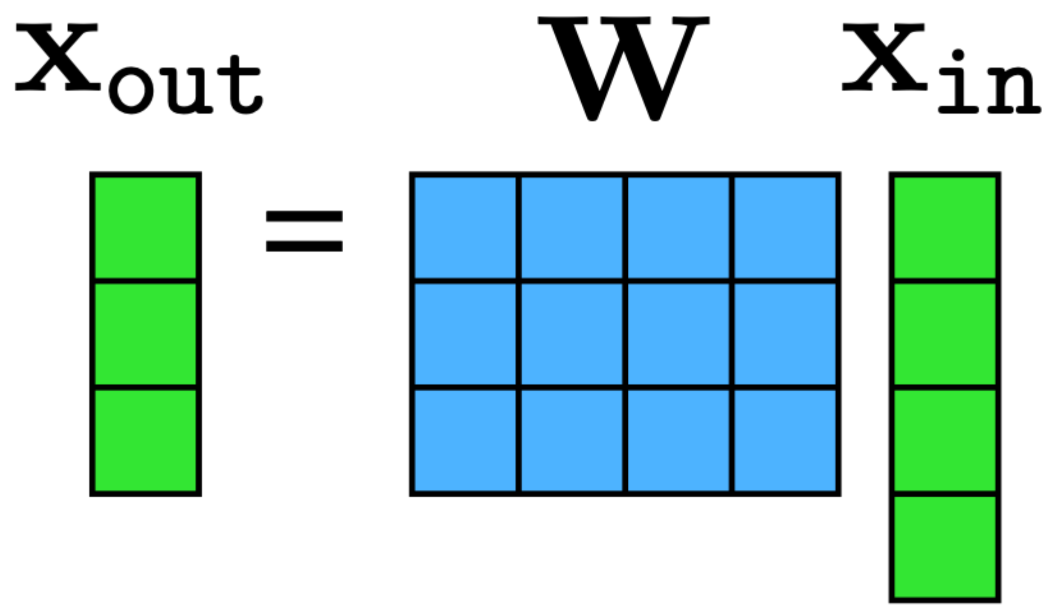
$$\frac{\partial J}{\partial \theta_l} = \frac{\partial J}{\partial \mathbf{x}_l} \cdot \frac{\partial f_l}{\partial \theta_l}$$



Linear layer



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



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

- Backprop to input:

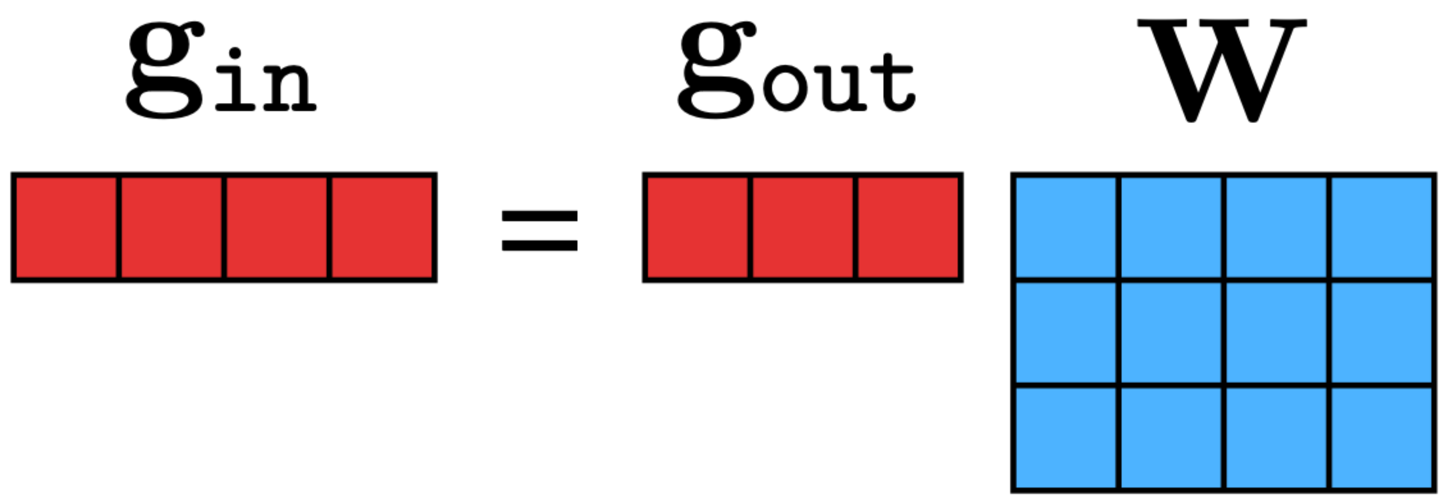
$$\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}} \cdot \frac{\partial f(\mathbf{x}_{\texttt{in}}, \mathbf{W})}{\partial \mathbf{x}_{\texttt{in}}} = \mathbf{g}_{\texttt{out}} \cdot \frac{\partial \mathbf{x}_{\texttt{out}}}{\partial \mathbf{x}_{\texttt{in}}}$$

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

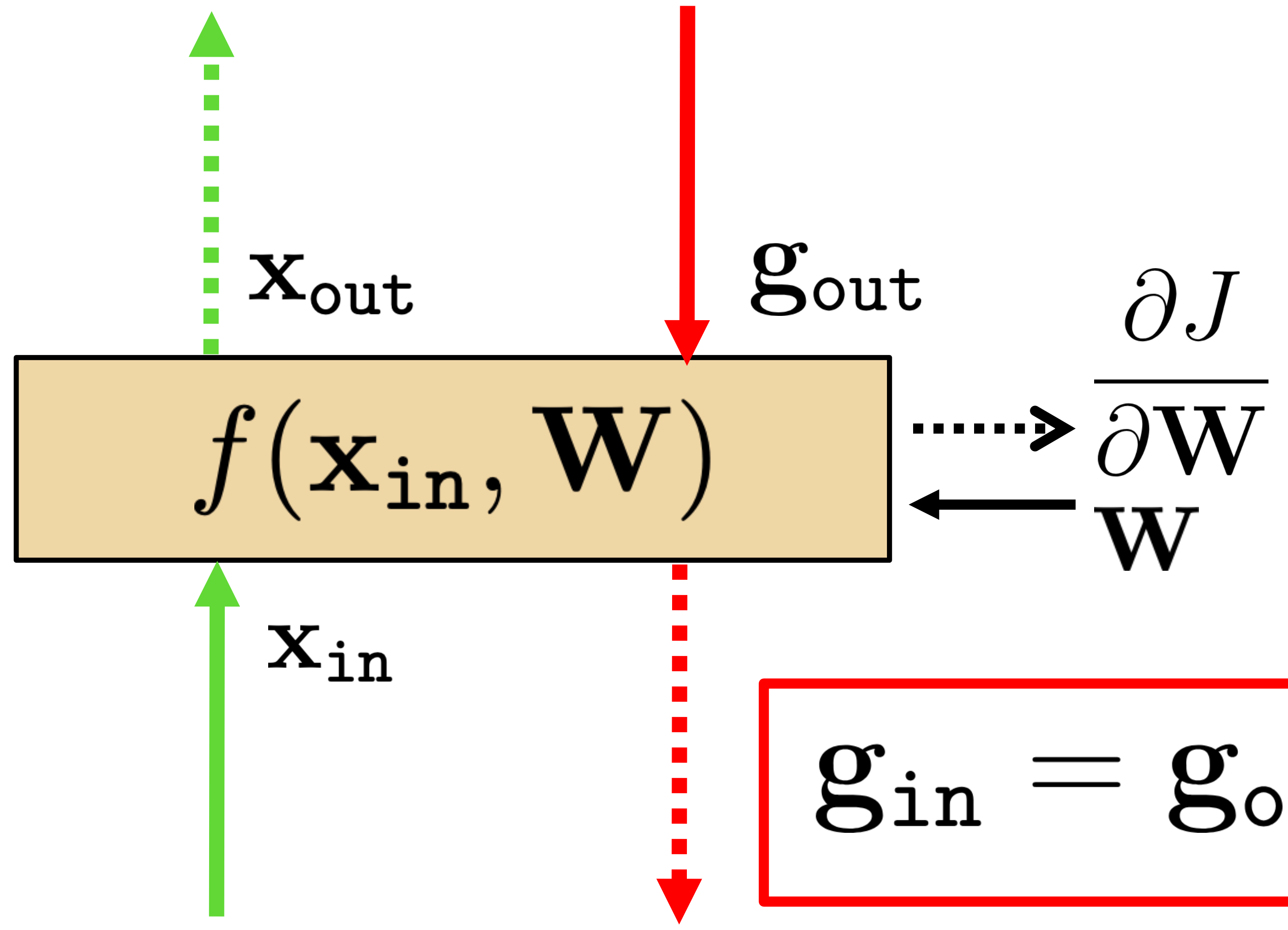
$$\frac{\partial \mathbf{x}_{\texttt{out}_i}}{\partial \mathbf{x}_{\texttt{in}_j}} = \mathbf{W}_{ij} \longrightarrow \frac{\partial f(\mathbf{x}_{\texttt{in}}, \mathbf{W})}{\partial \mathbf{x}_{\texttt{in}}} = \mathbf{W}$$

Therefore:

$$\boxed{\mathbf{g}_{\texttt{in}} = \mathbf{g}_{\texttt{out}} \cdot \mathbf{W}}$$

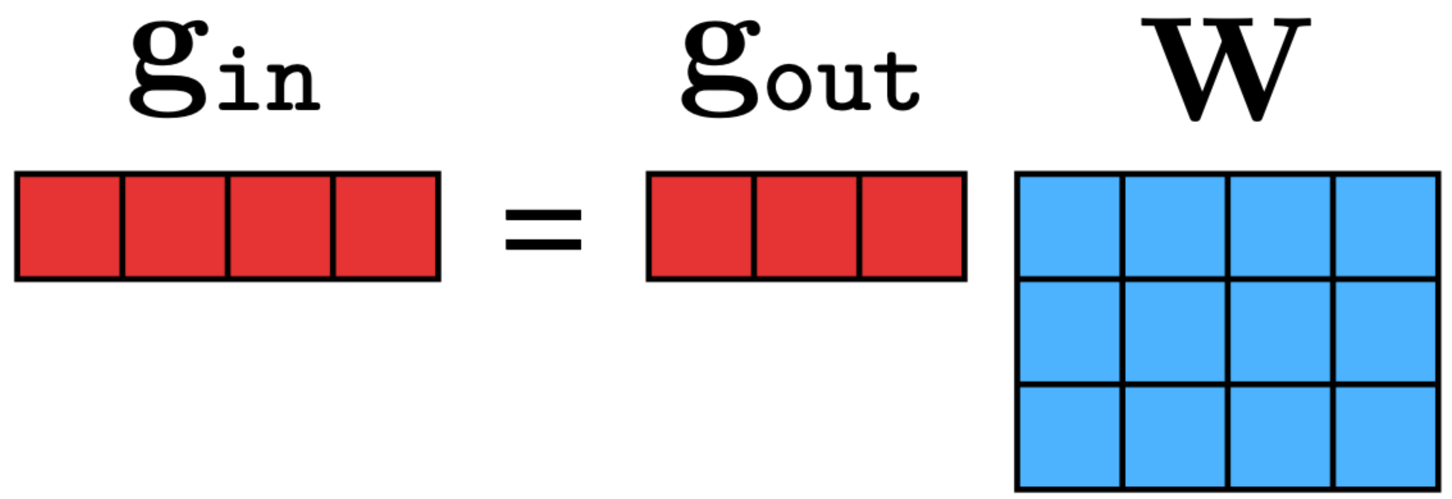


Linear layer



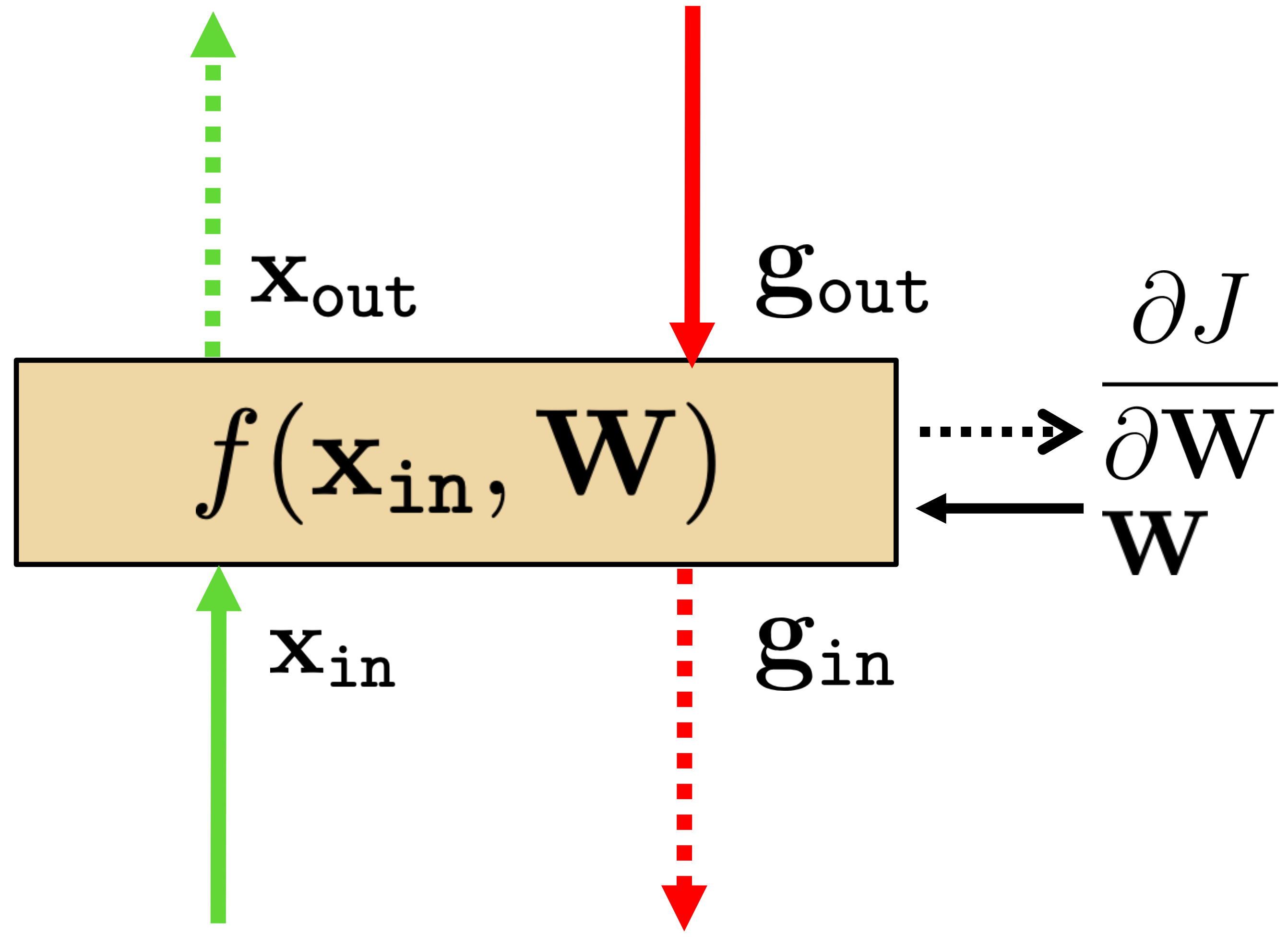
$$\boxed{\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \cdot \mathbf{W}}$$

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



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

Linear layer



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

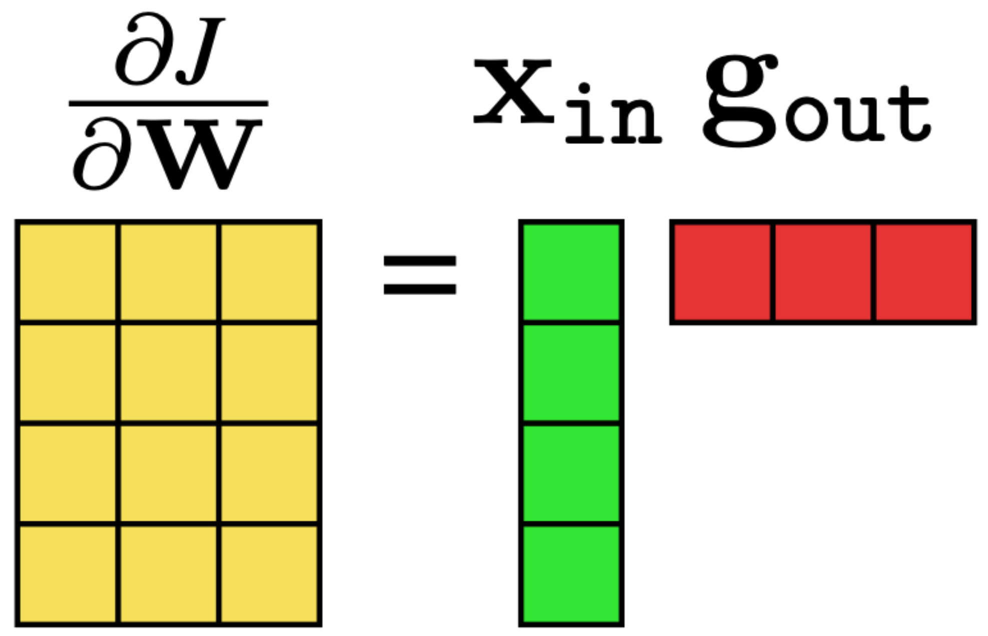
$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{W}}$$

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

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

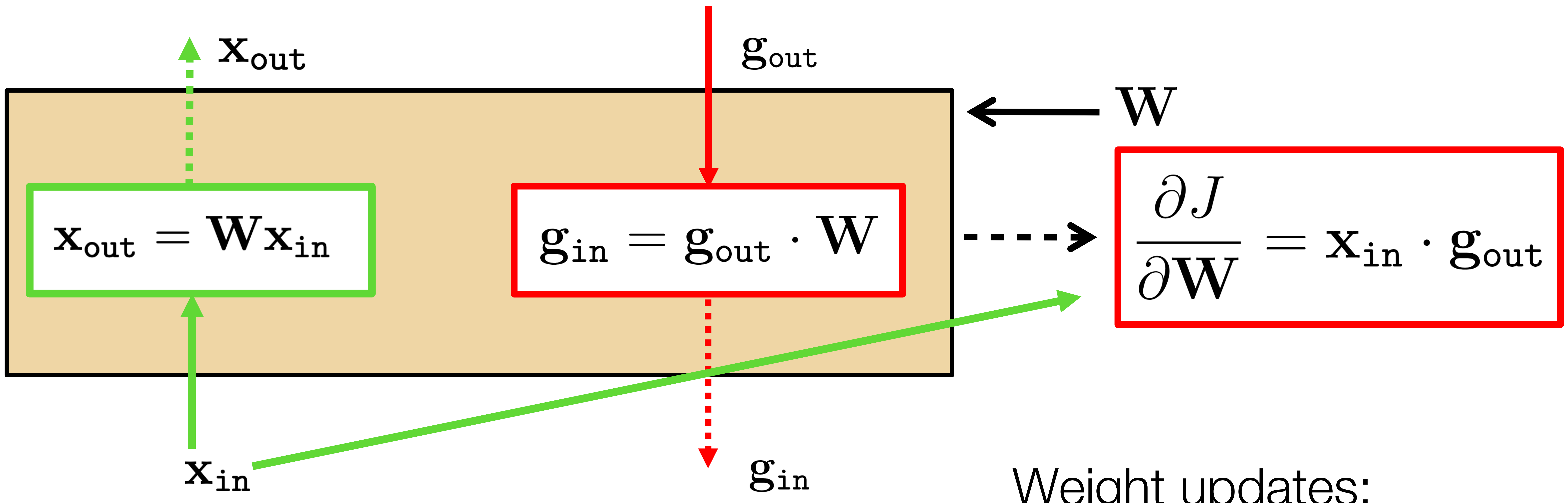
$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \mathbf{x}_{\text{in}_j}$$

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



And now we can update the weights: $\boxed{\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} + \eta \left(\frac{\partial J}{\partial \mathbf{W}}\right)^{T}}$

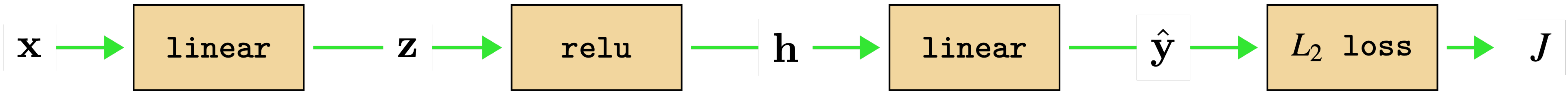
Linear layer

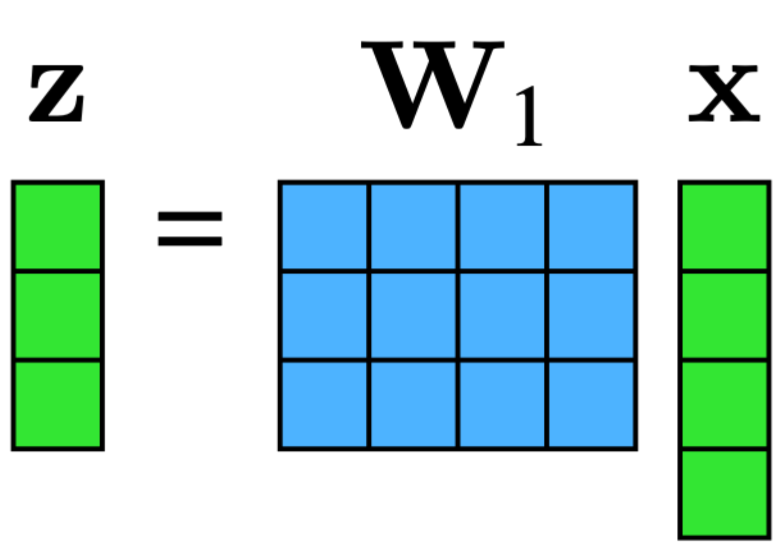


Weight updates:

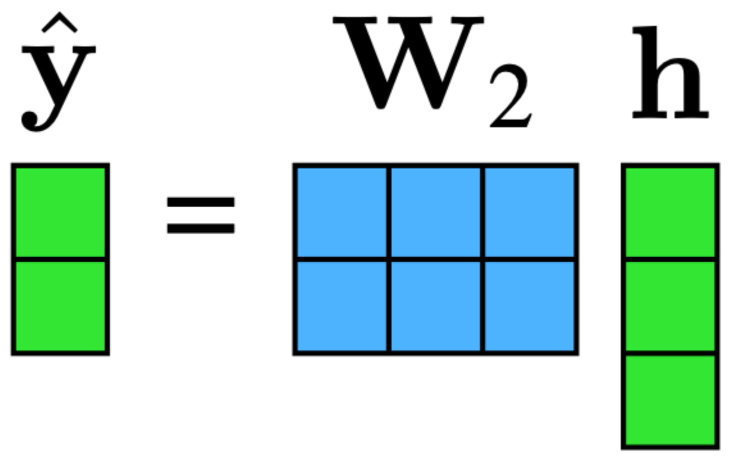
$$\mathbf{W}^{k+1} \leftarrow \mathbf{W}^{k} + \eta \left(\frac{\partial J}{\partial \mathbf{W}} \right)^{T}$$

Now lets look at a whole MLP: Forward





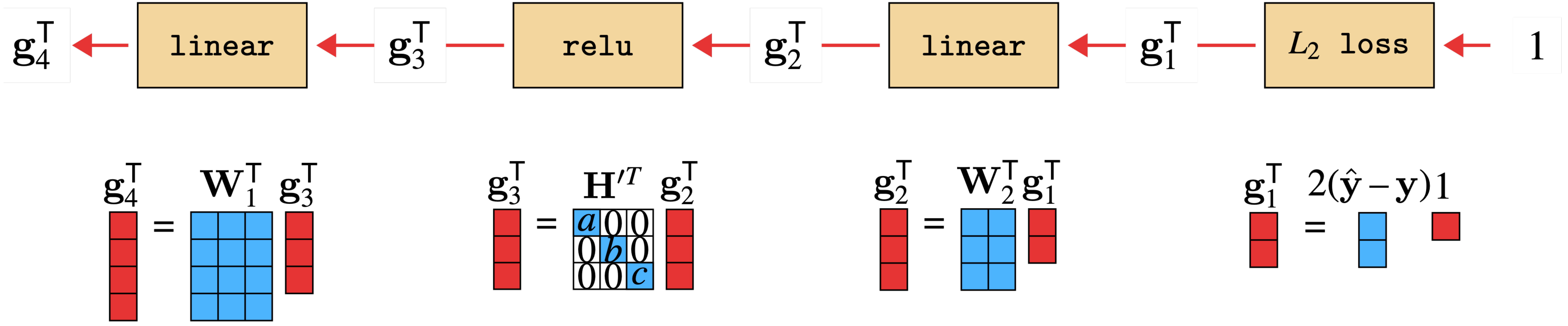
$$\mathbf{h} = \texttt{relu}(\mathbf{z})$$

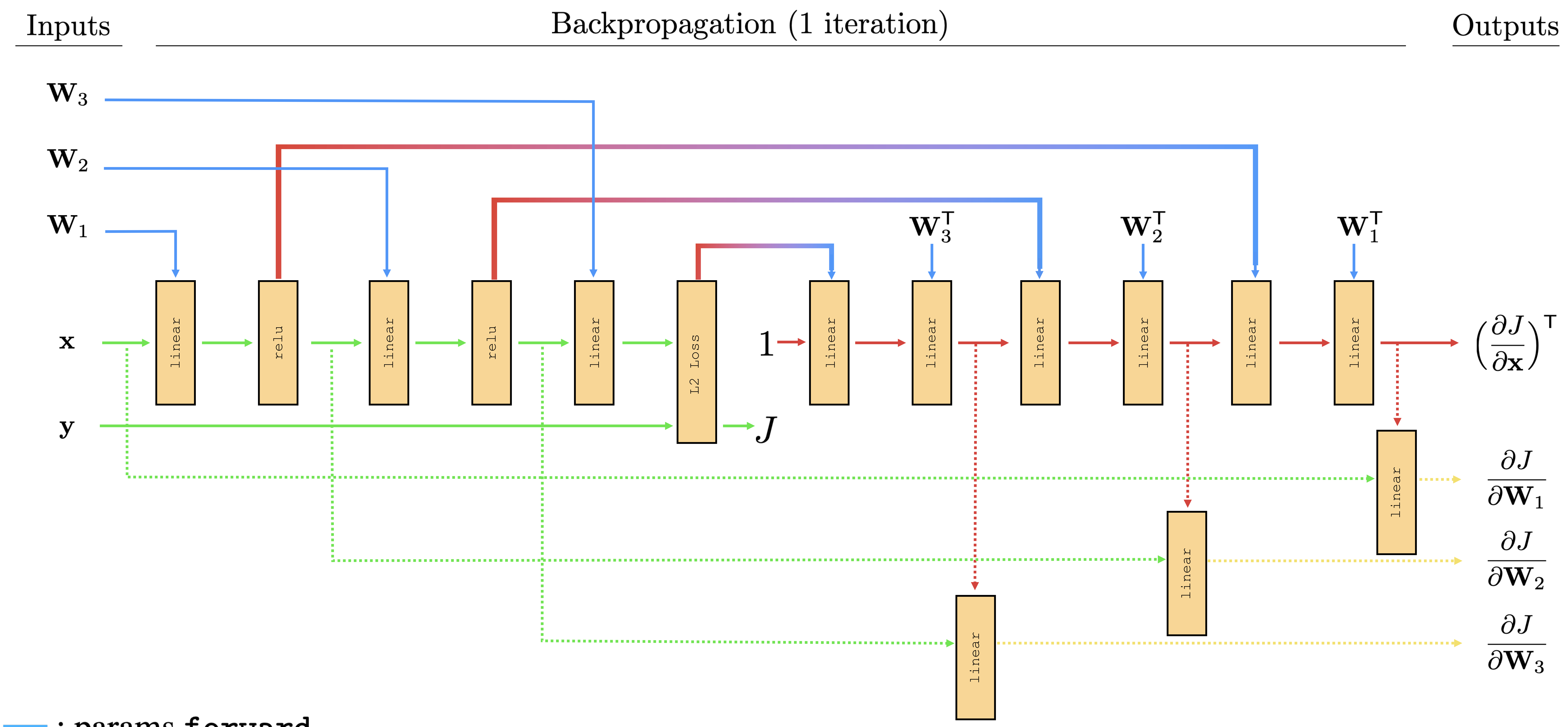


$$J = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$$

Now lets look at a whole MLP: Backward

$$\mathbf{g}_{\texttt{in}}^{\mathsf{T}} = (\mathbf{g}_{\texttt{out}}\mathbf{W})^{\mathsf{T}} = \mathbf{W}^{\mathsf{T}}\mathbf{g}_{\texttt{out}}^{\mathsf{T}}$$





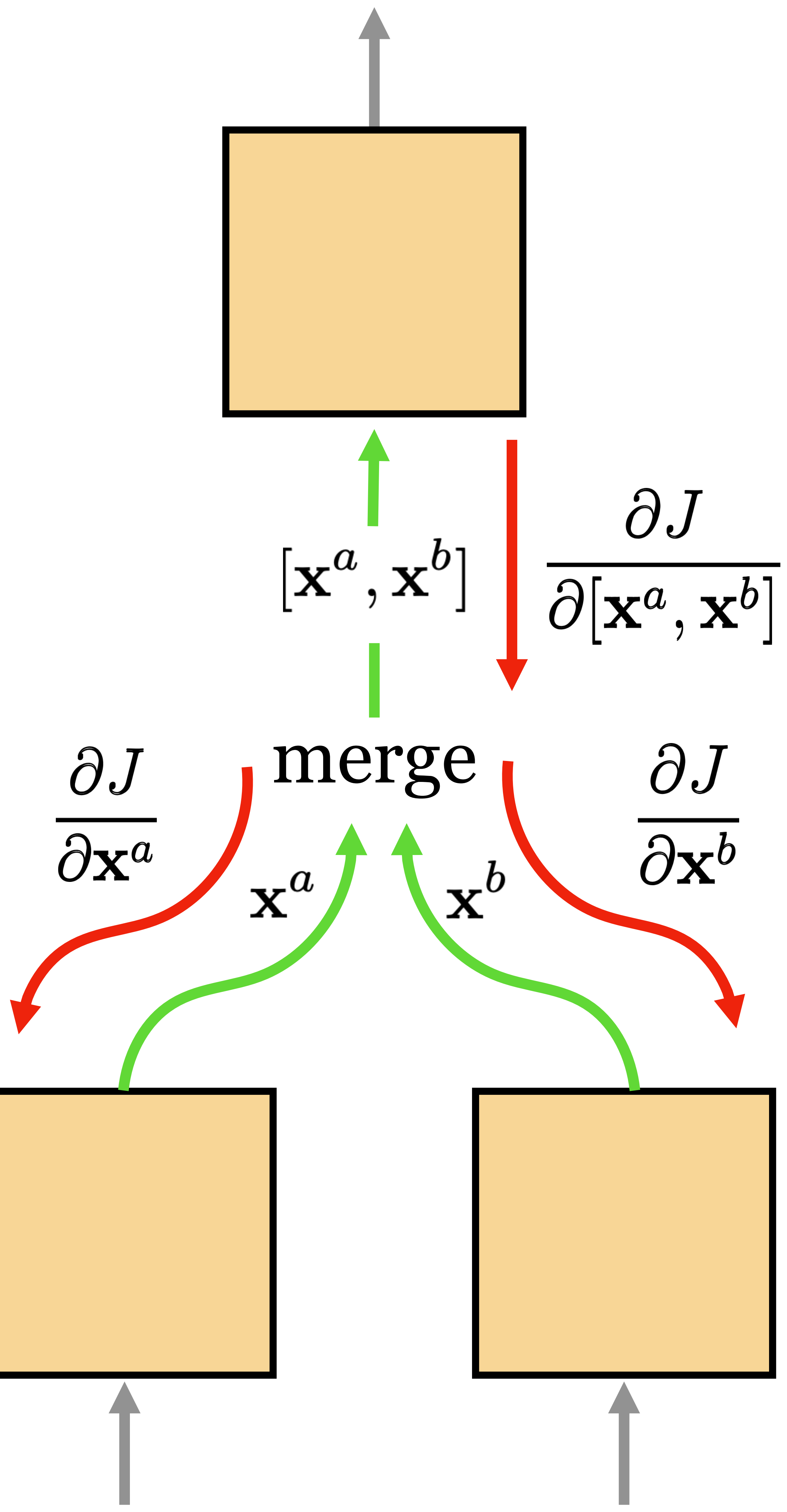
: params `forward`

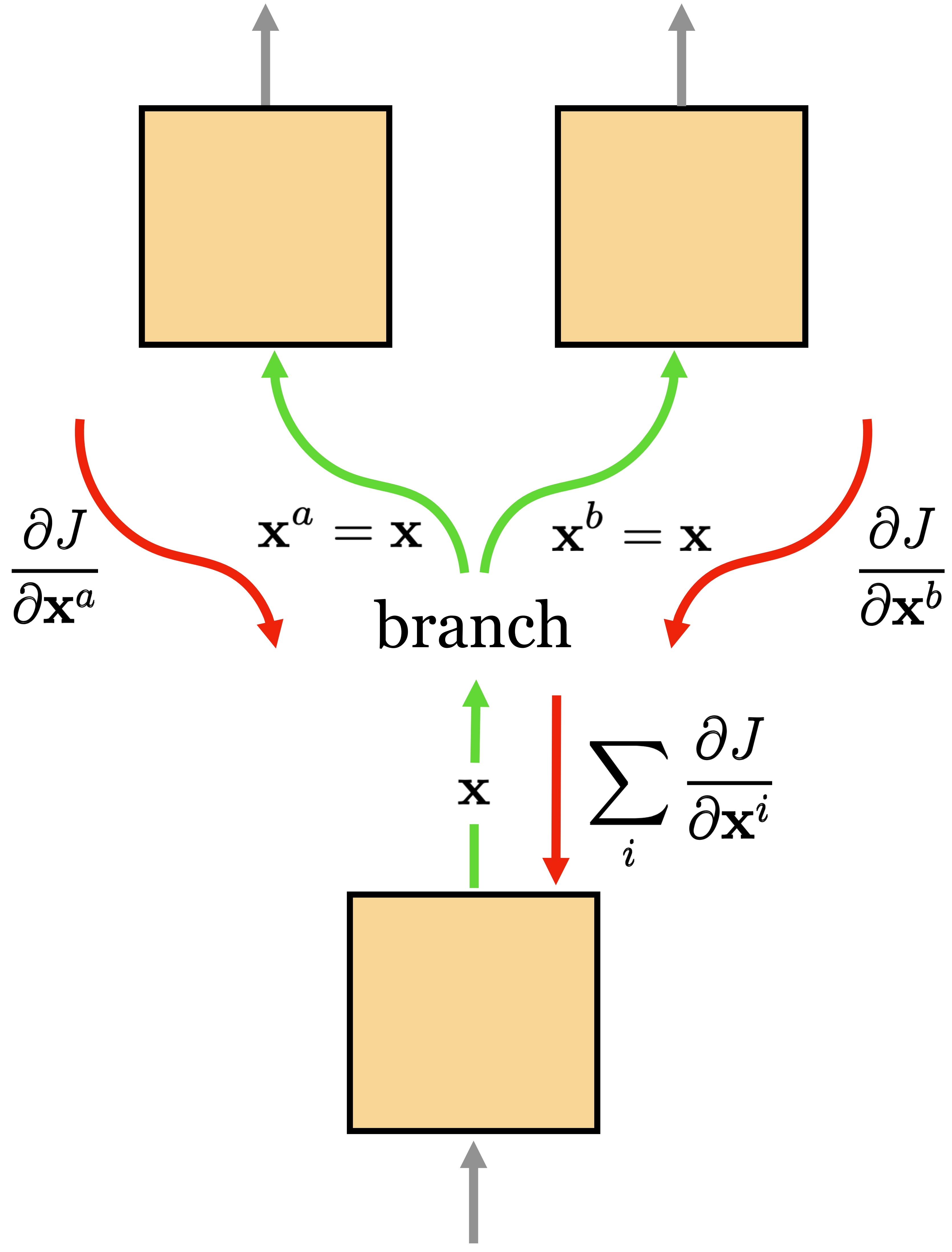
: params `backward`

: data `forward`

: data `backward`

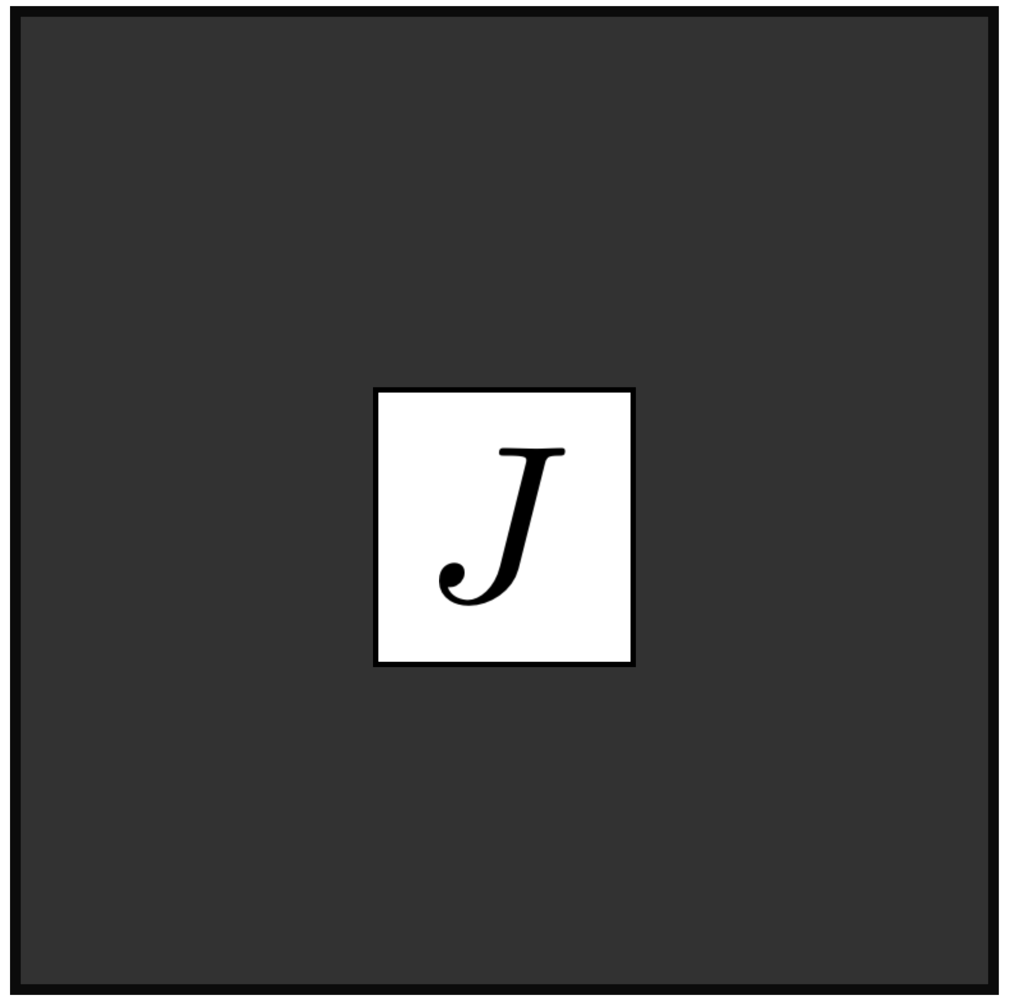
DAGs


$[\mathbf{x}^a, \mathbf{x}^b]$
$\frac{\partial J}{\partial [\mathbf{x}^a, \mathbf{x}^b]}$
merge
$\frac{\partial J}{\partial \mathbf{x}^a}$
$\frac{\partial J}{\partial \mathbf{x}^b}$
$\mathbf{x}^a$
$\mathbf{x}^b$


$\frac{\partial J}{\partial \mathbf{x}^a}$
$\mathbf{x}^a = \mathbf{x}$
$\mathbf{x}^b = \mathbf{x}$
$\frac{\partial J}{\partial \mathbf{x}^b}$
branch
$\mathbf{x}$
$\sum_i \frac{\partial J}{\partial \mathbf{x}^i}$

Optimization

Params

$\theta \longrightarrow$  $\longrightarrow$

$J(\theta)$

$\nabla_\theta J(\theta)$

$H_\theta(J(\theta))$

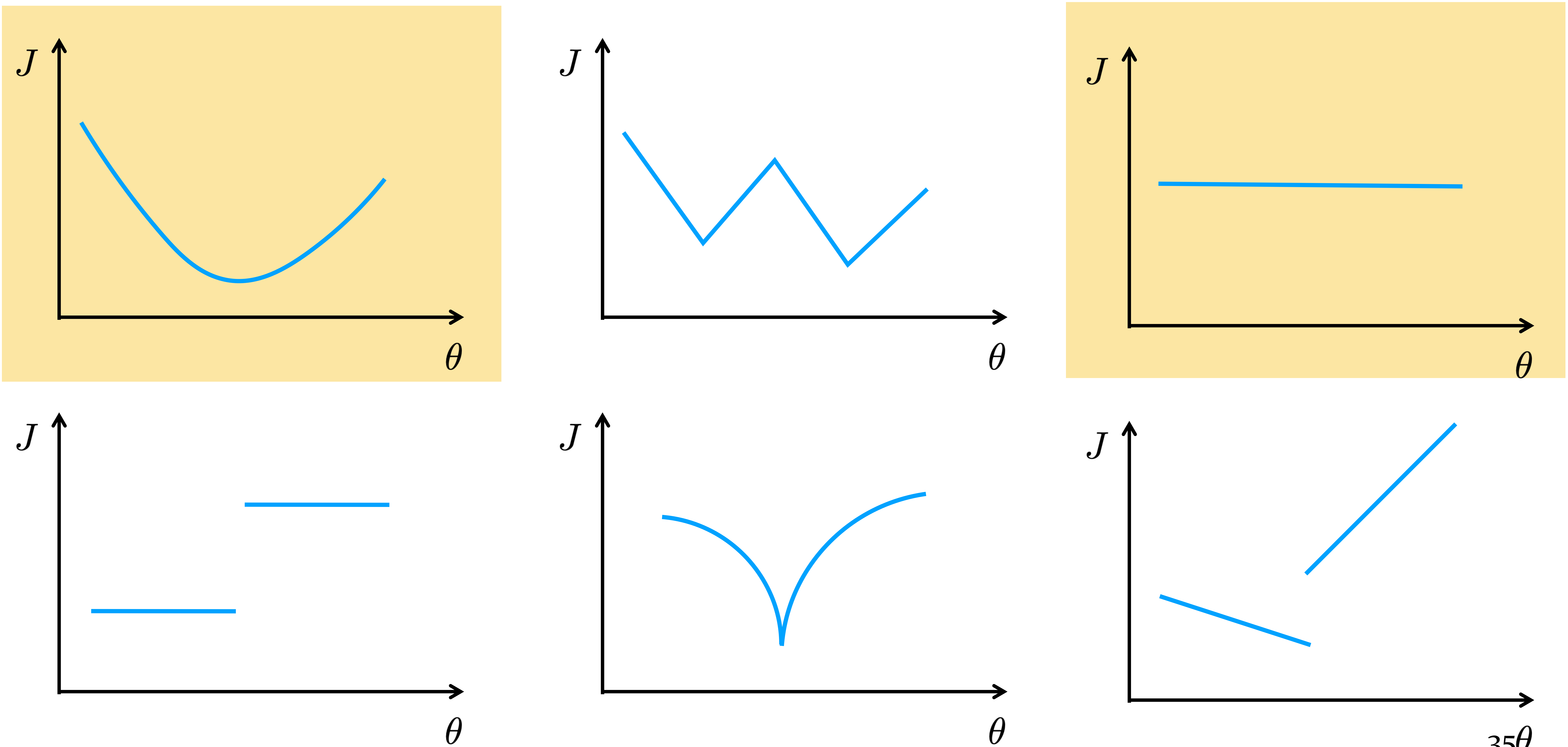
$$\boxed{\theta^* = \arg\min_\theta J(\theta)}$$

- 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

Which are differentiable?



Which will be hard to optimize?

Stochastic Gradient Descent (SGD)

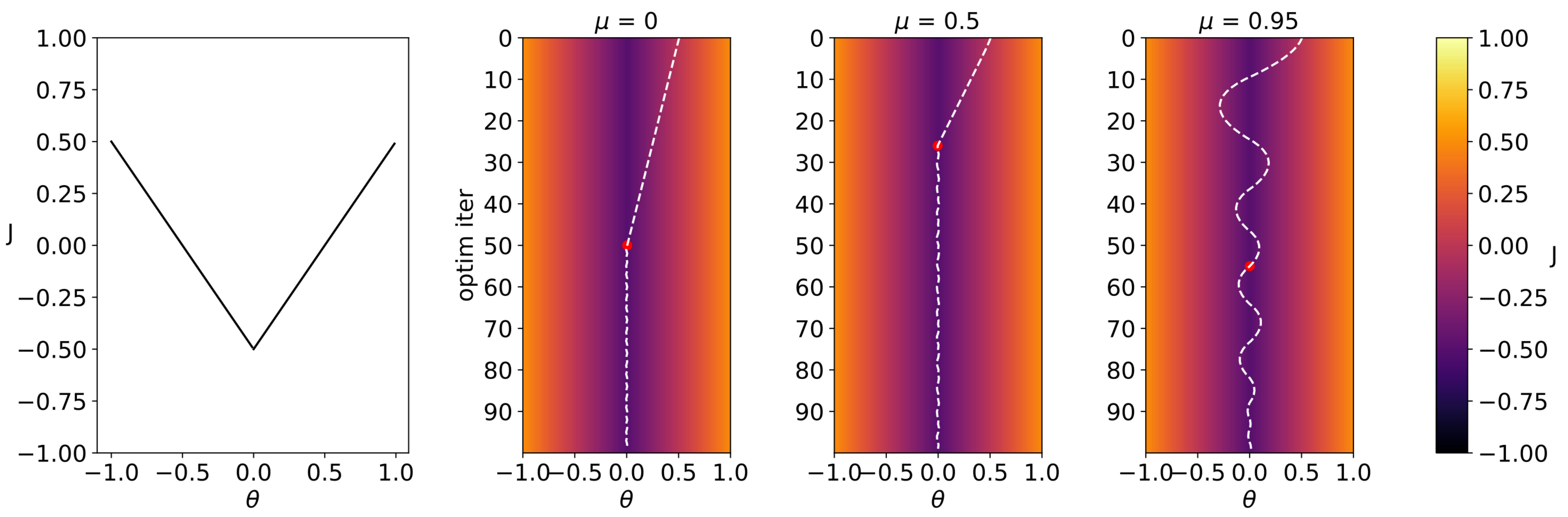
- Want to minimize overall loss function **J**, which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.
  - If batchsize=1 then **θ** is updated after each example.
  - If batchsize=N (full set) then this is standard gradient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

Momentum

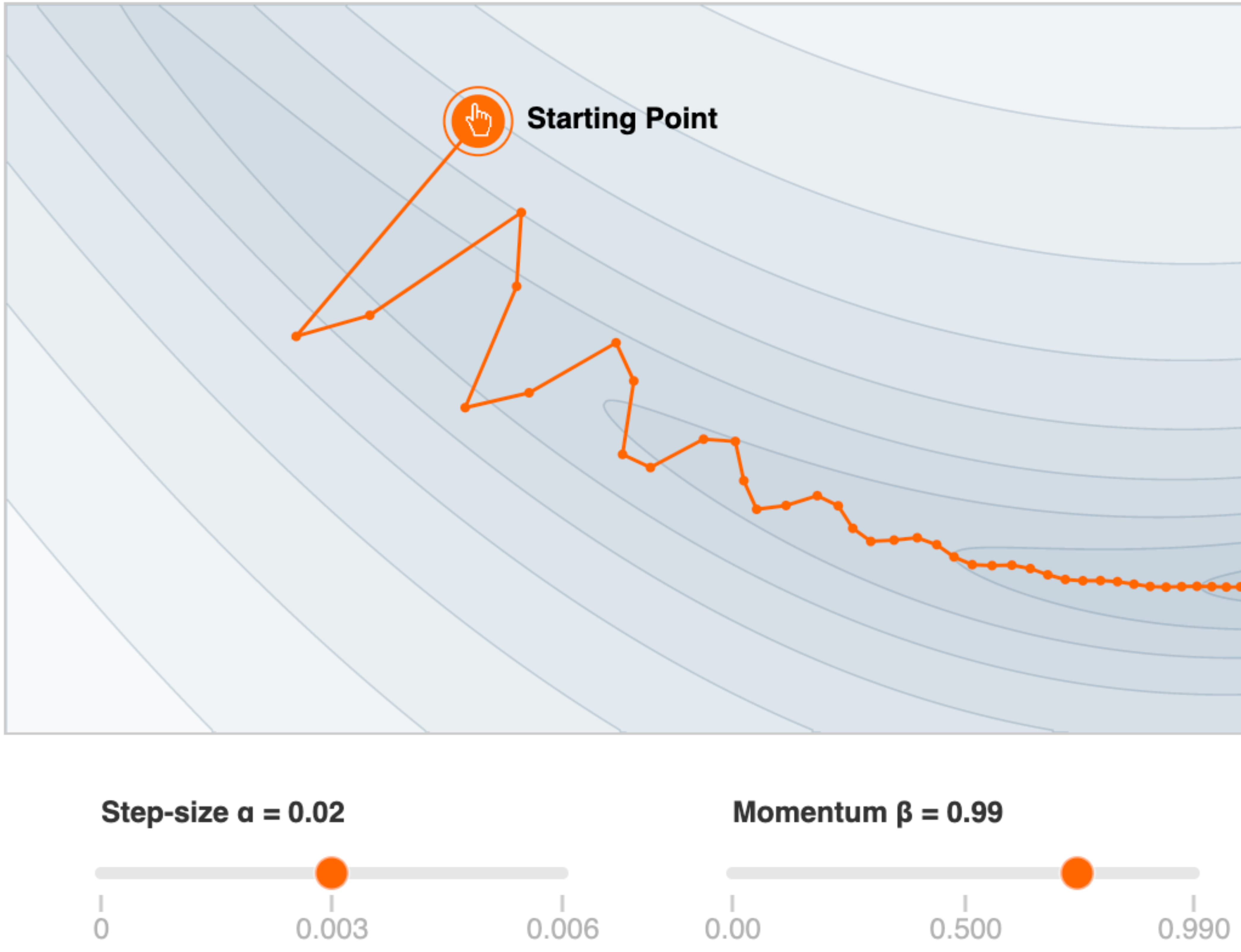
- A heavy ball rolling down a hill, gains speed.
- Gradient steps biased to continue in direction of previous update:

$$\theta^{t+1} \leftarrow \theta^t - \eta \nabla f(\theta^t) - \alpha m^t$$

- Can help or hurt. Strength of momentum is a hyperparam.



Why Momentum Really Works



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

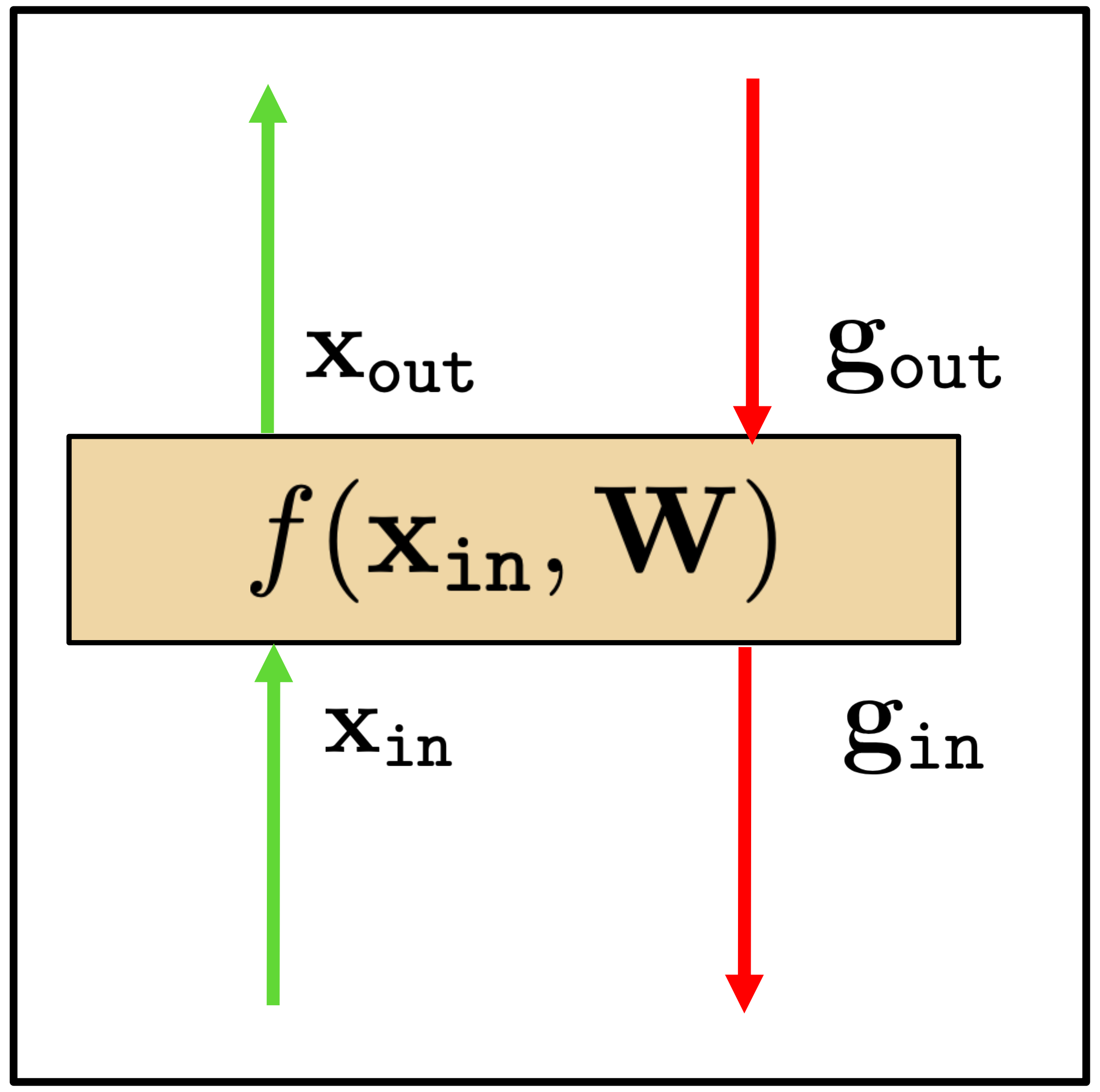
April. 4
2017

Citation:
Goh, 2017

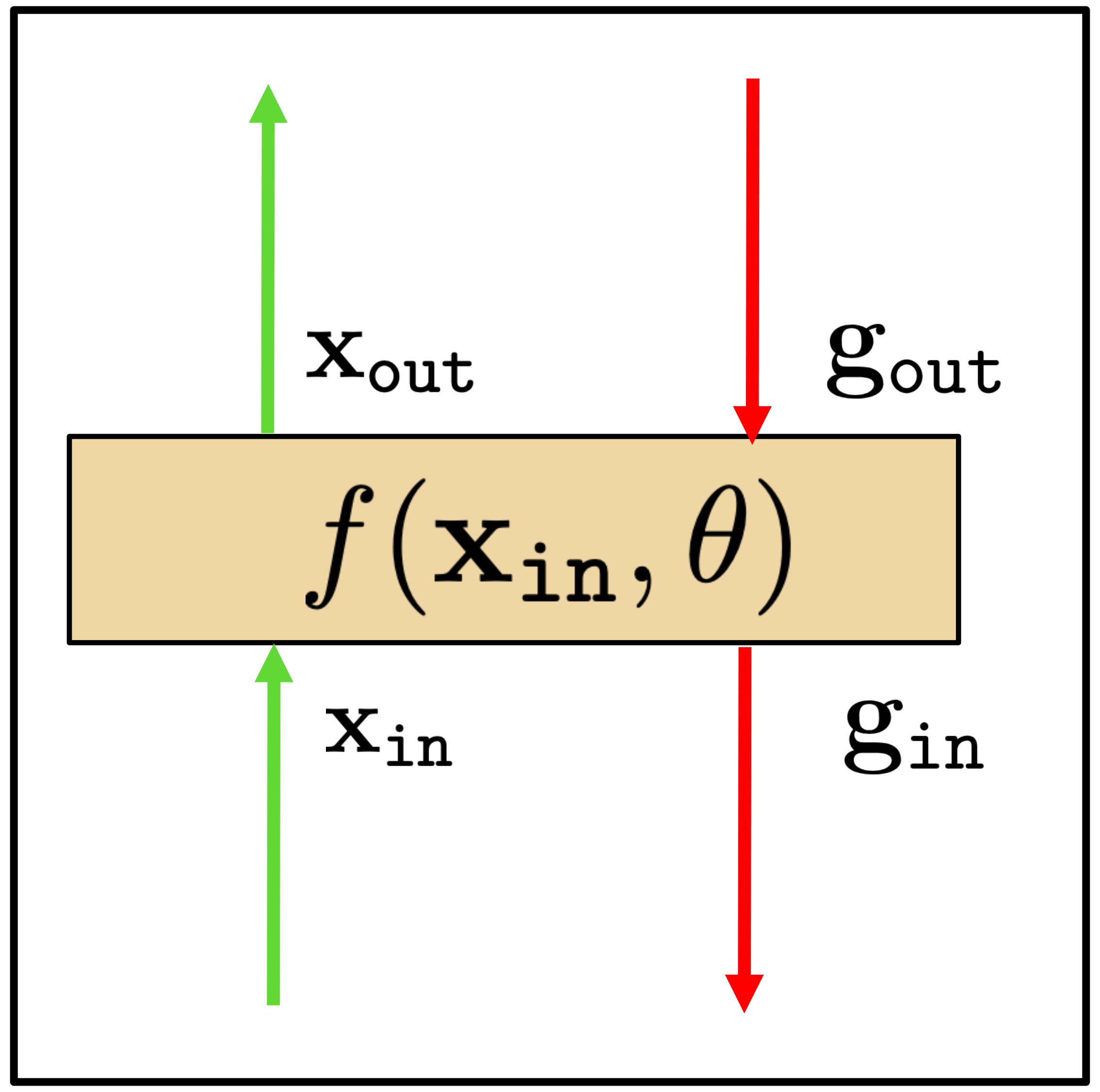
https://distill.pub/2017/momentum/

Differentiable programming

Deep learning

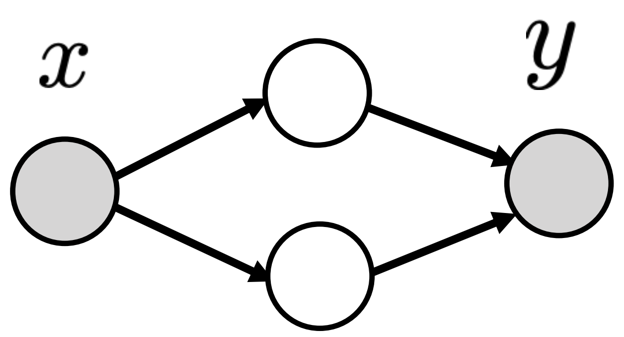


Differentiable programming



PyTorch

TensorFlow ™



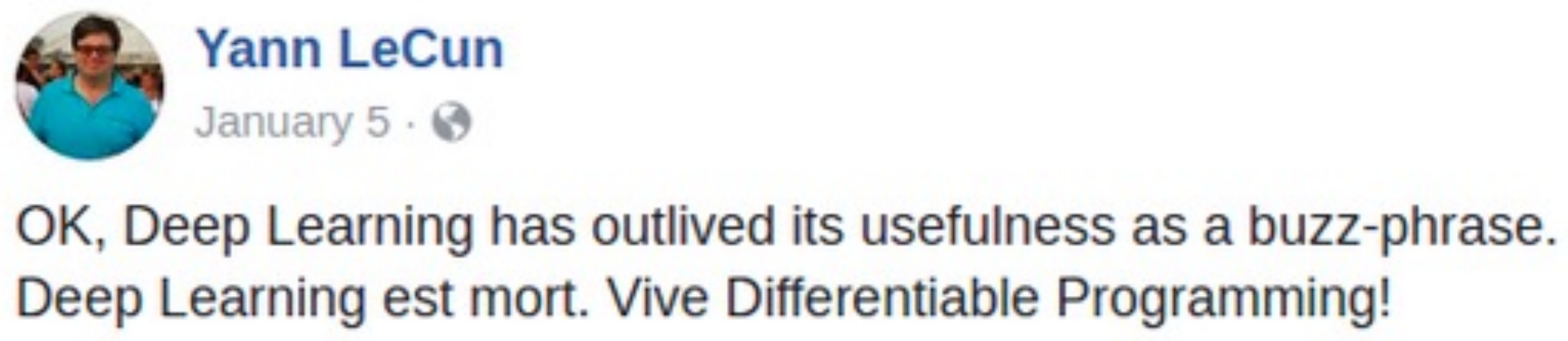

```
for i, data in enumerate(dataset):
  iter_start_time = time.time()
  if total_steps % opt.print_freq == 0:
      t_data = iter_start_time - iter_data_time
  visualizer.reset()
  total_steps += opt.batch_size
  epoch_iter += opt.batch_size
  model.set_input(data)
  model.optimize_parameters()
```

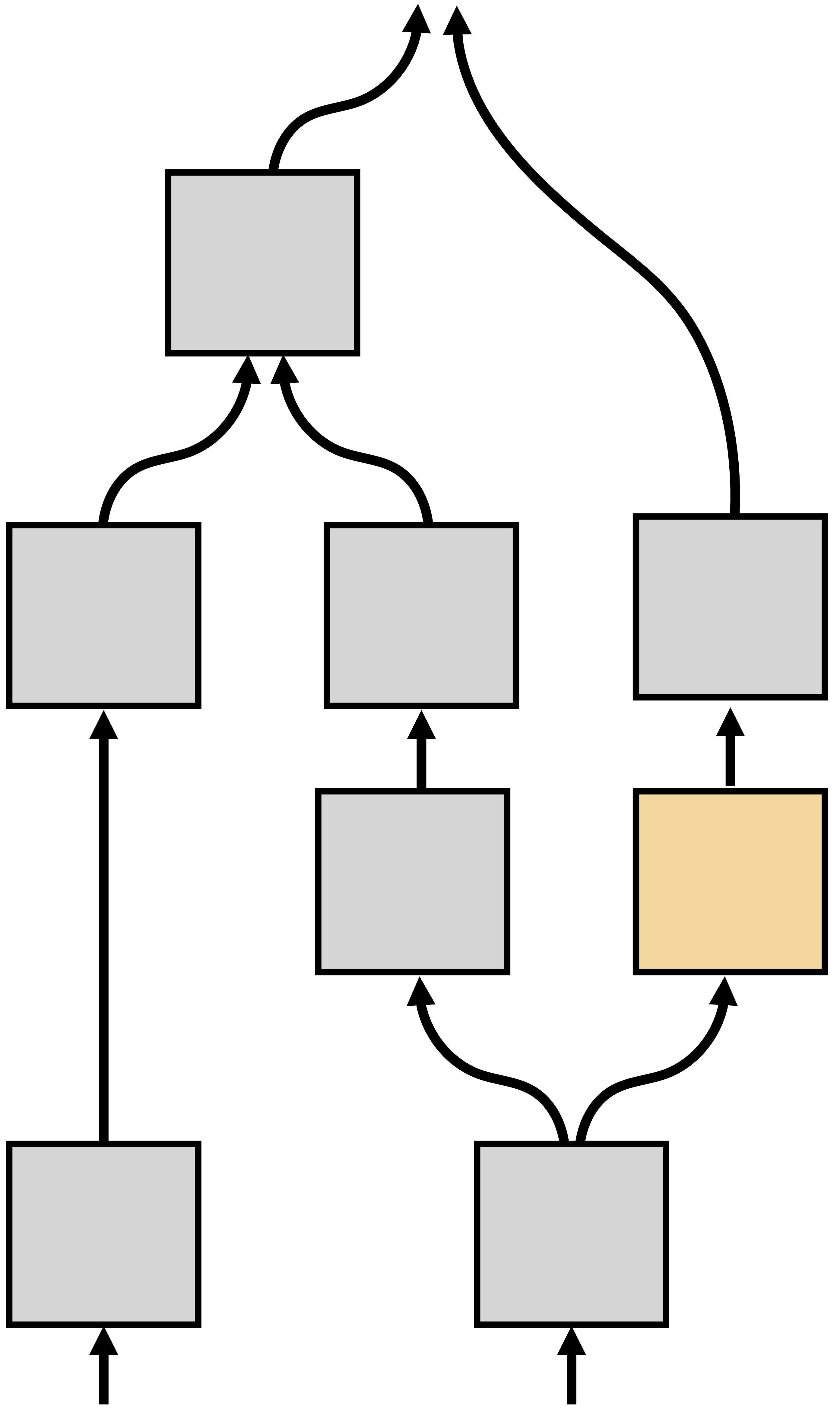
Differentiable programming

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**.



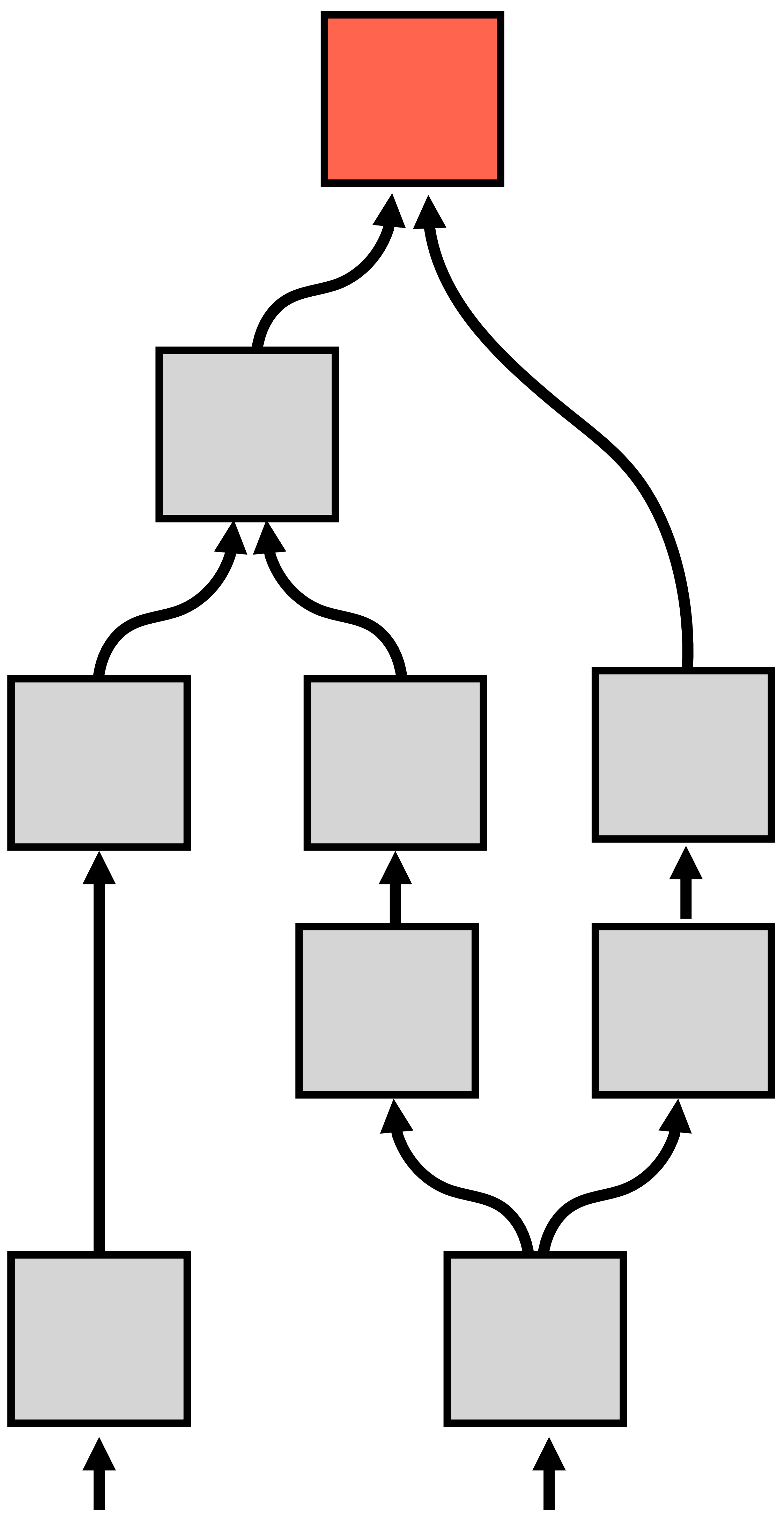


Programmed by a human

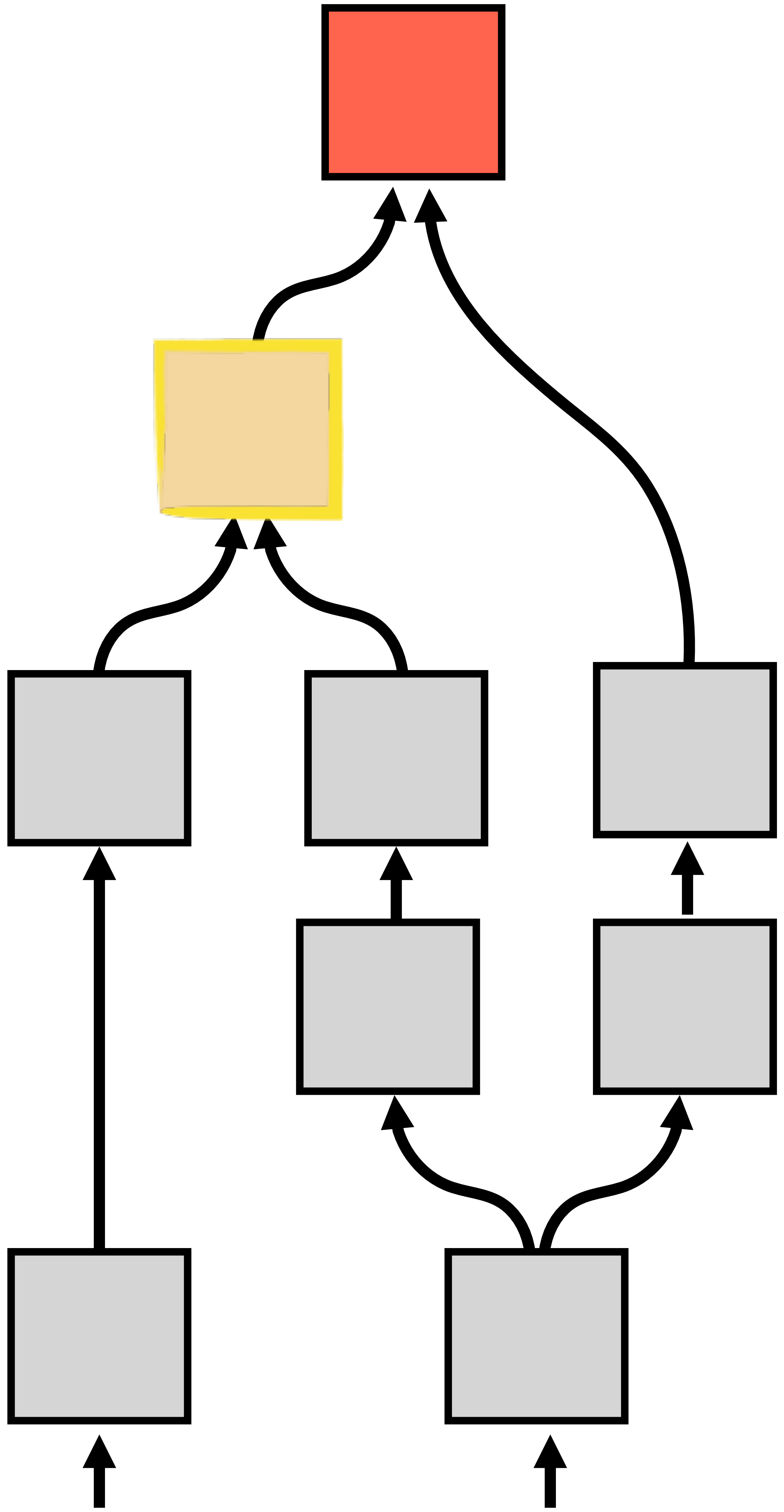
Programmed by backprop

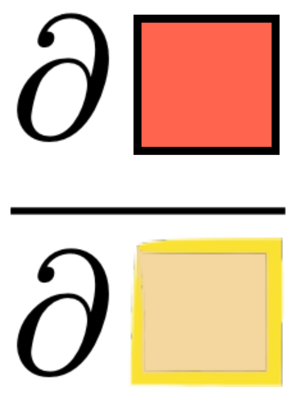
e.g., programmed by tuning behavior to match training examples

Backprop lets you optimize any node (function) or edge (variable) in your computation graph w.r.t. any scalar cost



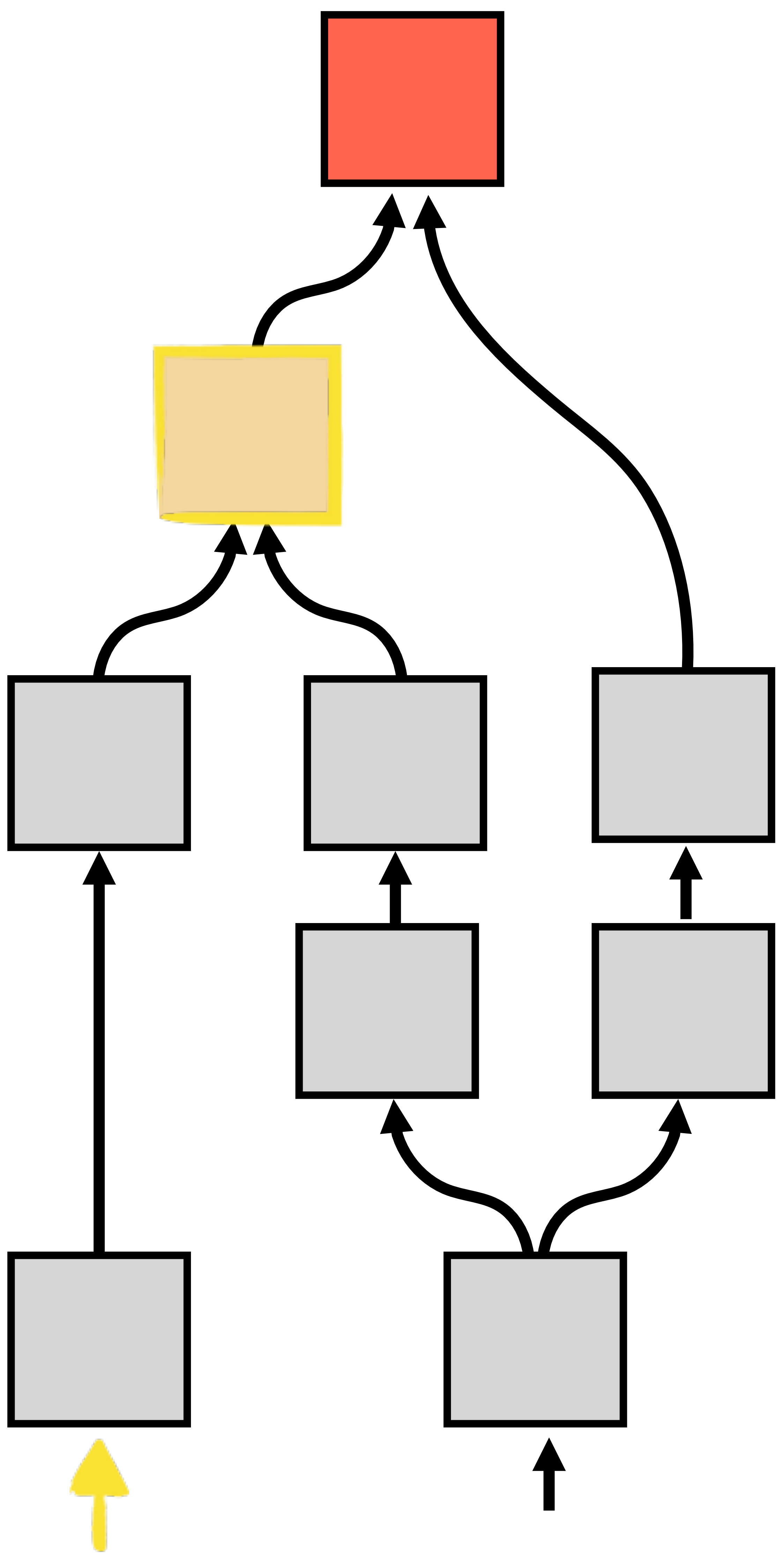
Backprop lets you optimize any node (function) or edge (variable) in your computation graph w.r.t. any scalar cost

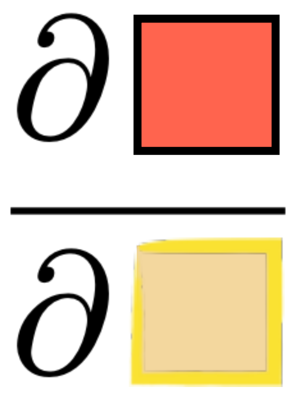




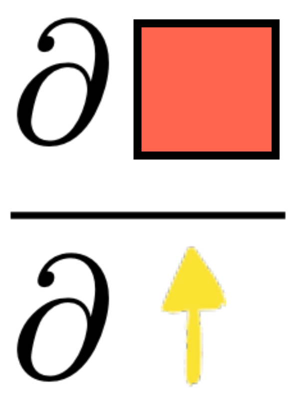
How the loss changes when the weights of that function (yellow) change

Backprop lets you optimize any node (function) or edge (variable) in your computation graph w.r.t. any scalar cost



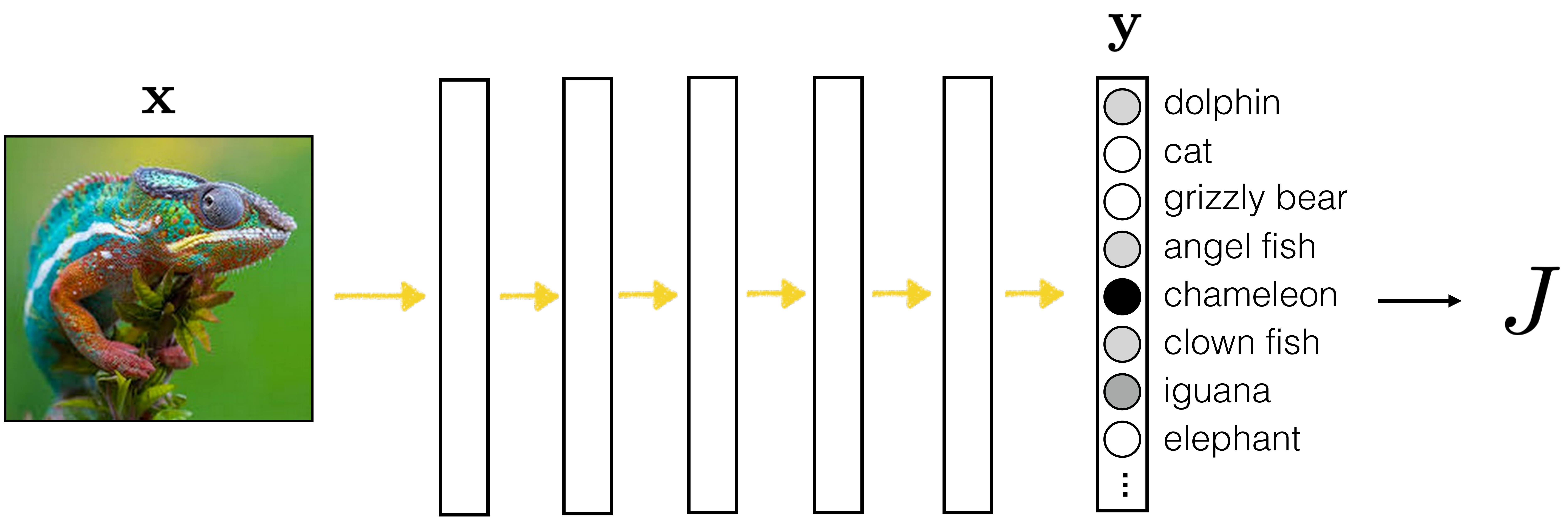


How the loss changes when the functional node highlighted changes



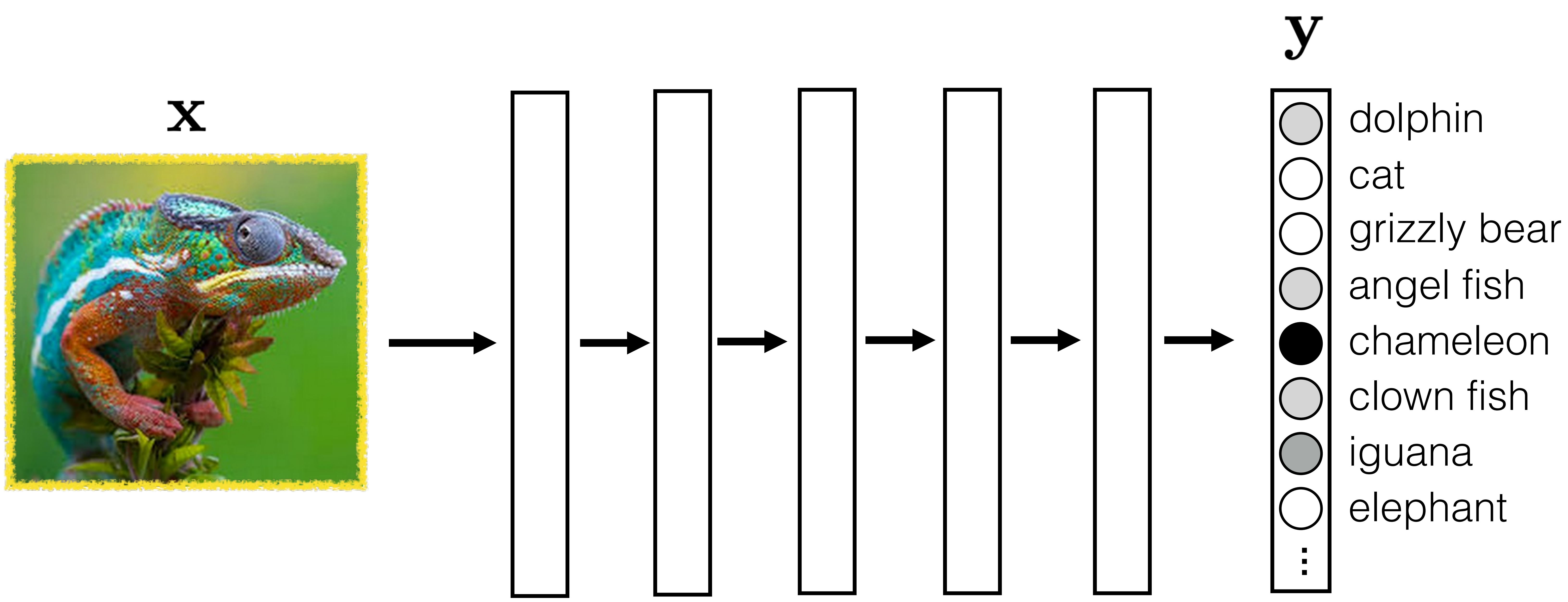
How the cost changes when the input data changes

Optimizing parameters versus optimizing inputs



$$\frac{\partial J}{\partial \theta} \leftarrow$$ How much the total cost is increased or decreased by changing the parameters.

Optimizing parameters versus optimizing inputs



$$\frac{\partial y_j}{\partial \mathbf{x}} \leftarrow$$ How much the “chameleon” score is increased or decreased by changing the image pixels.

Unit visualization

$$\arg\max_{\mathbf{x}} y_j + \lambda R(\mathbf{x})$$

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \eta \frac{\partial(y_j(\mathbf{x}) + \lambda R(\mathbf{x}))}{\partial \mathbf{x}}\Big|_{\mathbf{x}=\mathbf{x}^k}$$

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

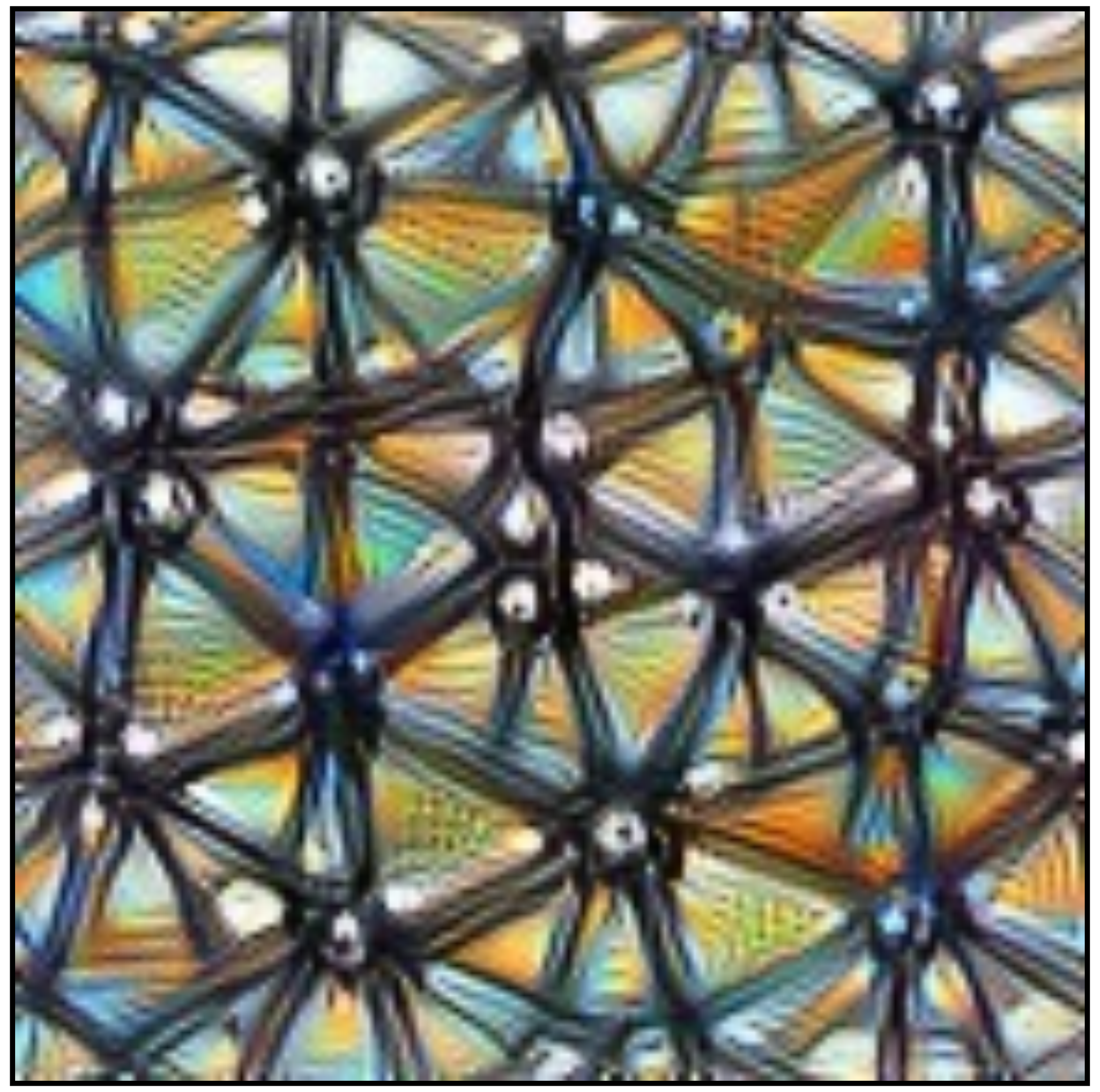
[https://distill.pub/2017/feature-visualization/]

Unit visualization

$$\arg\max_{\mathbf{x}} h_{l_j} + \lambda R(\mathbf{x})$$

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \eta \frac{\partial(h_{l_j}(\mathbf{x}) + \lambda R(\mathbf{x}))}{\partial \mathbf{x}}\Bigg|_{\mathbf{x}=\mathbf{x}^k}$$

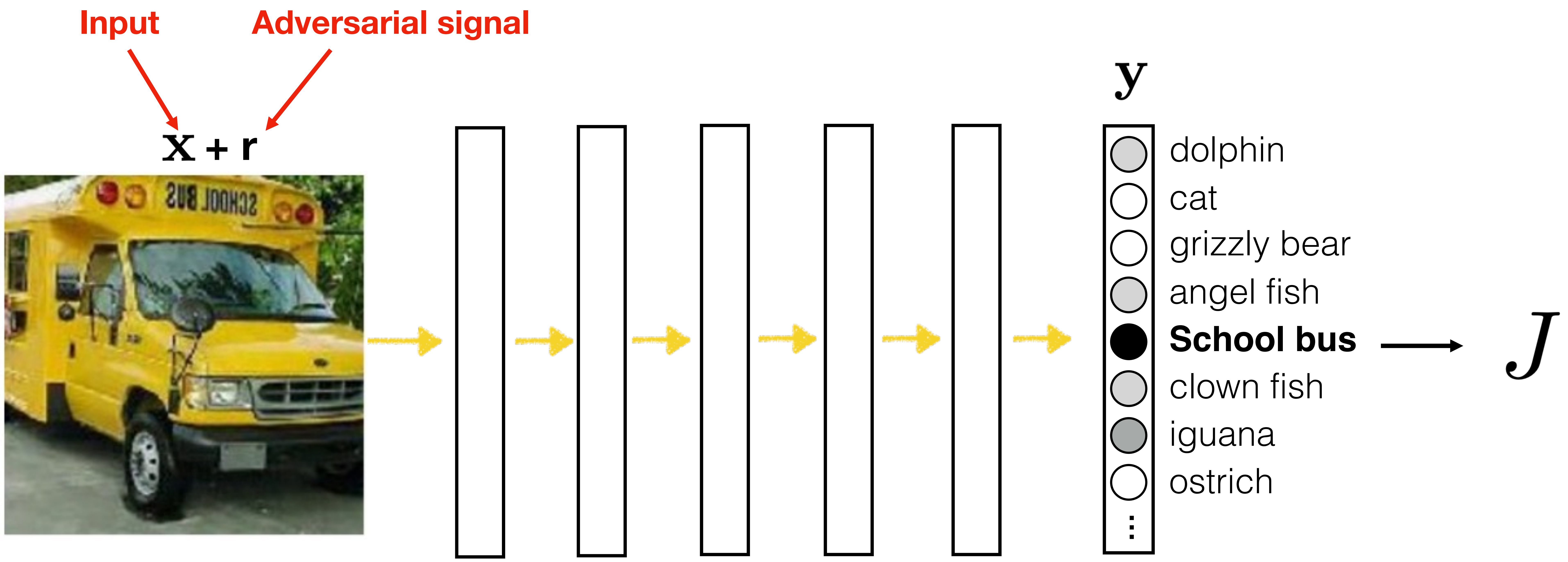
Make an image that maximizes the value of neuron j on layer l of the network:



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

“Deep dream” [https://ai.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html]

Adversarial attacks



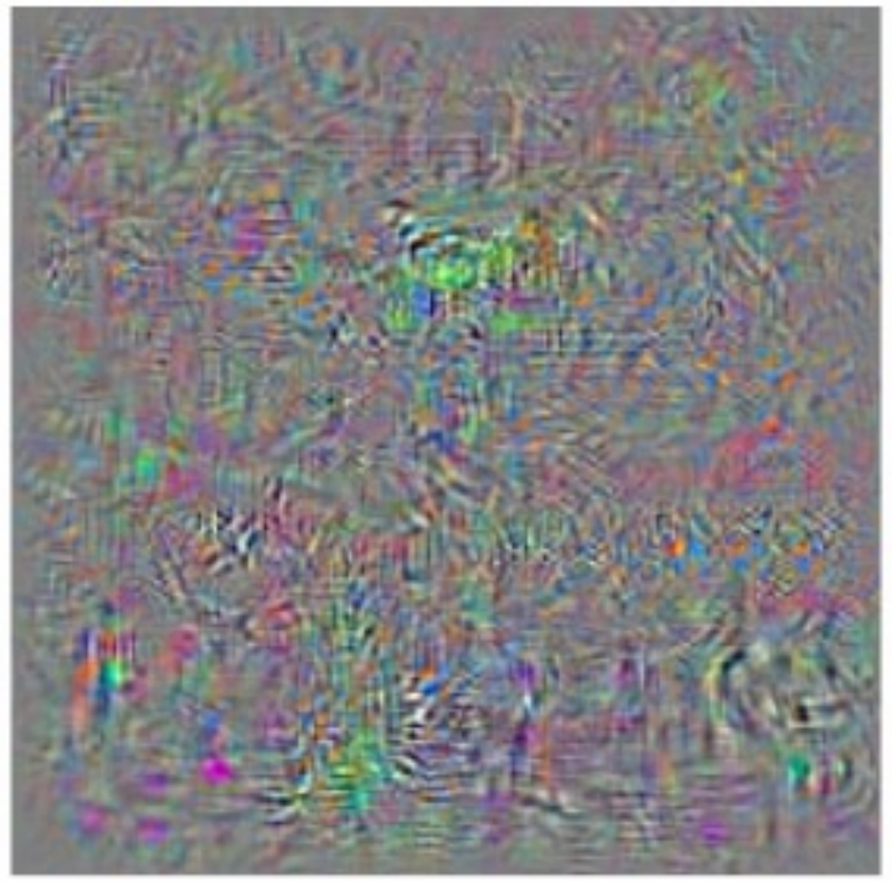
$$\frac{\partial y_j}{\partial r} \leftarrow$$ What adversarial signal r should we add to change the output label?

["Intriguing properties of neural networks", Szegedy et al. 2014]

Adversarial attacks

$\mathbf{x}$ $\quad$ $\mathbf{r}$ $\quad$ $\mathbf{x} + \mathbf{r}$

+



=

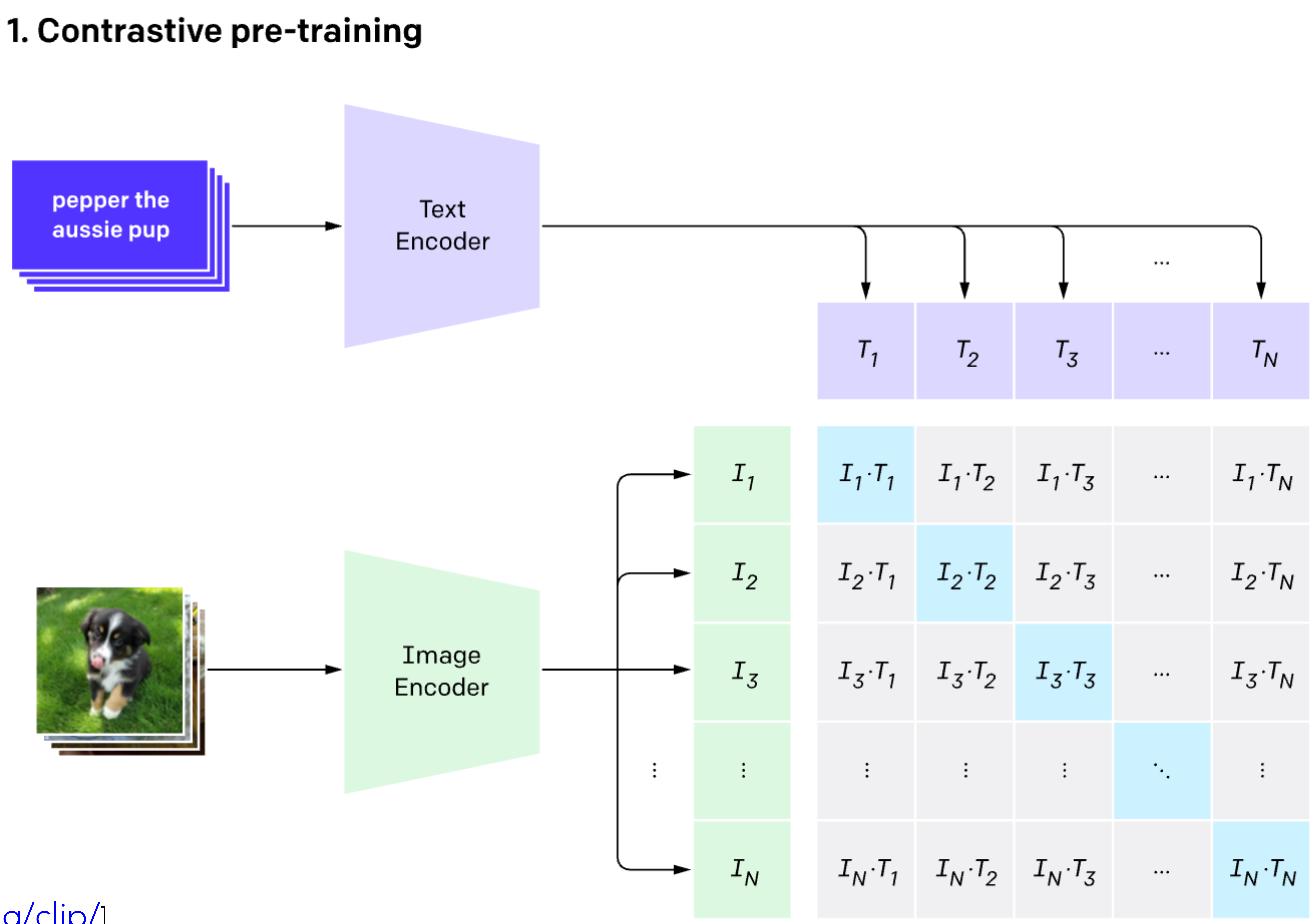
f $\quad$ f

y $\quad$ “School bus” $\quad$ “Ostrich”

$$\arg\max_{\mathbf{r}} p(y = \texttt{ostrich} | \mathbf{x} + \mathbf{r}) \quad \text{subject to} \quad ||\mathbf{r}|| < \epsilon$$

[“Intriguing properties of neural networks”, Szegedy et al. 2014]

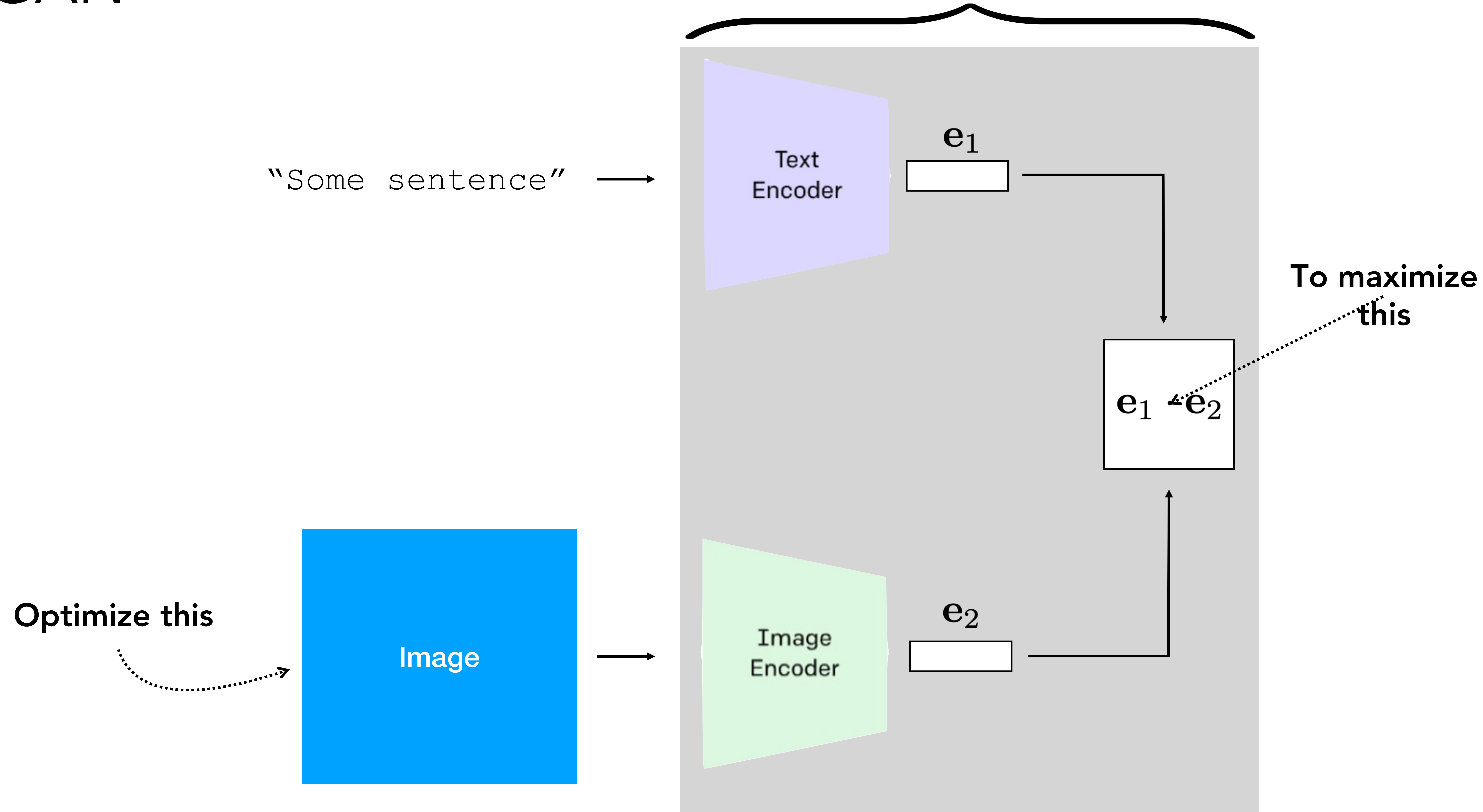
CLIP



[https://openai.com/blog/clip/]

CLIP+GAN

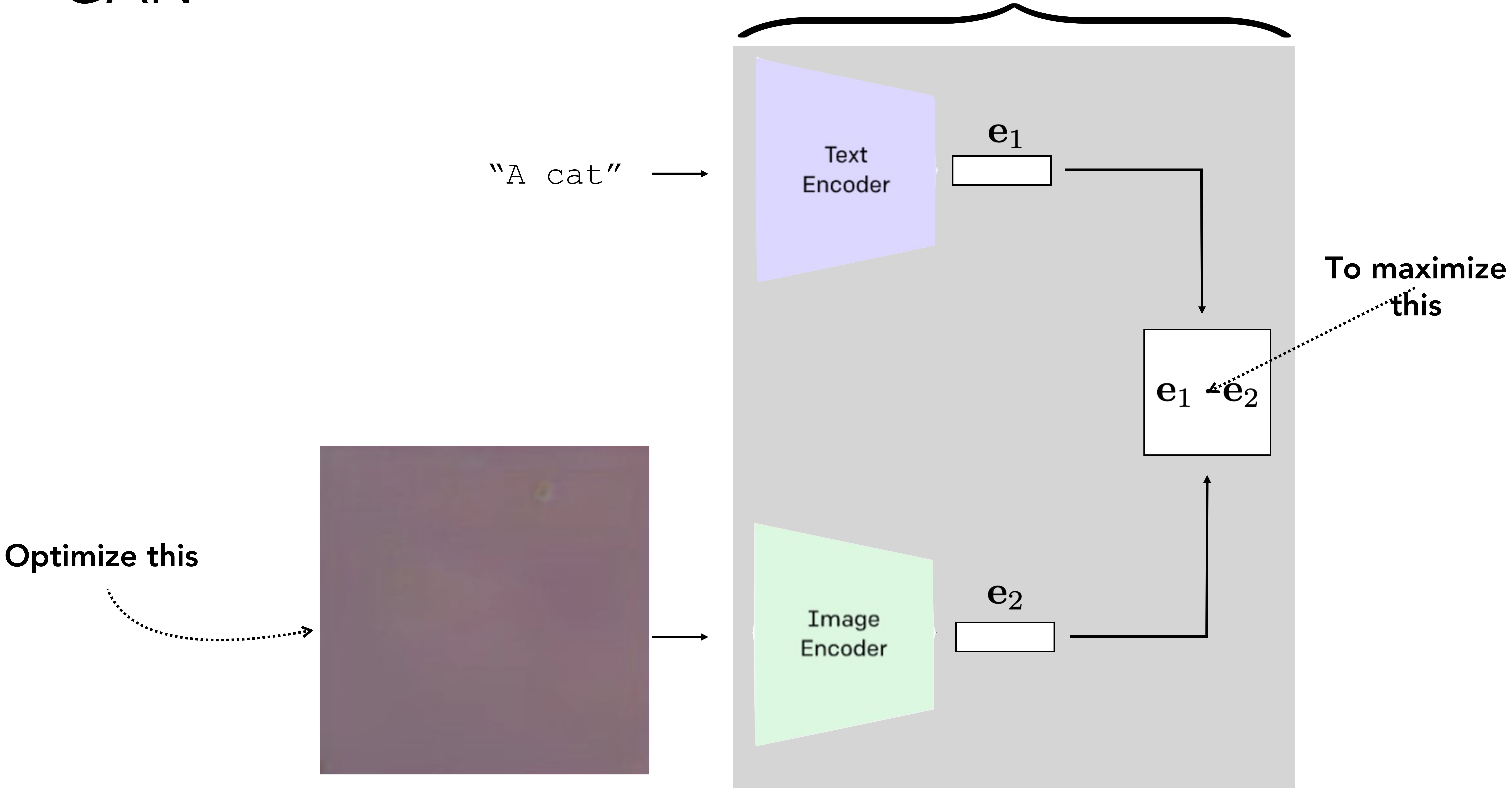
Differentiable program that measures the similarity between text and images



Code: https://colab.research.google.com/drive/1_4PQqzM_0KKytCzWtn-ZPi4cCa5bwK2F?usp=sharing

CLIP+GAN

Differentiable program that measures the similarity between text and images

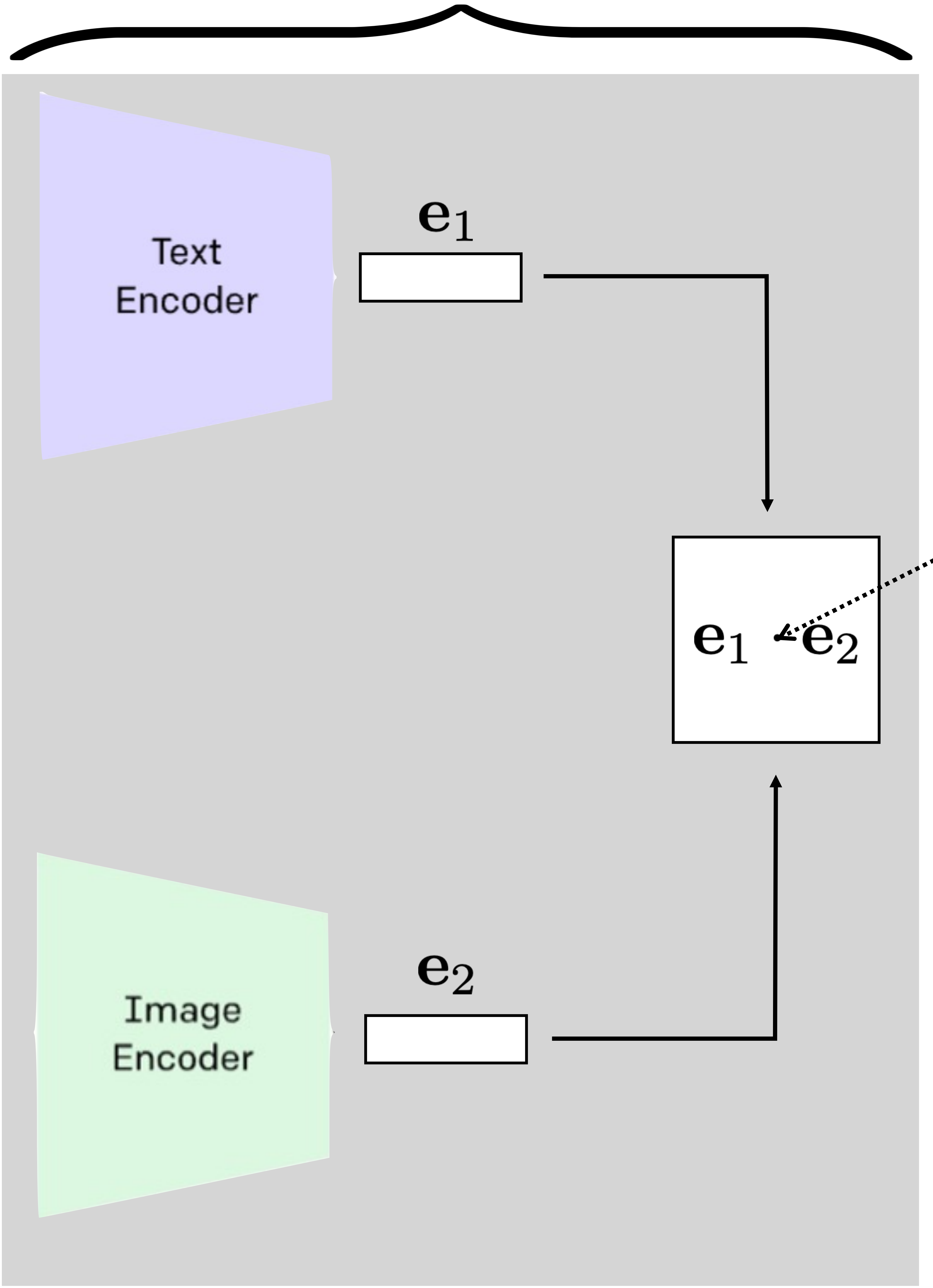


Code: https://colab.research.google.com/drive/1_4PQqzM_0KKytCzWtn-ZPi4cCa5bwK2F?usp=sharing

CLIP+GAN

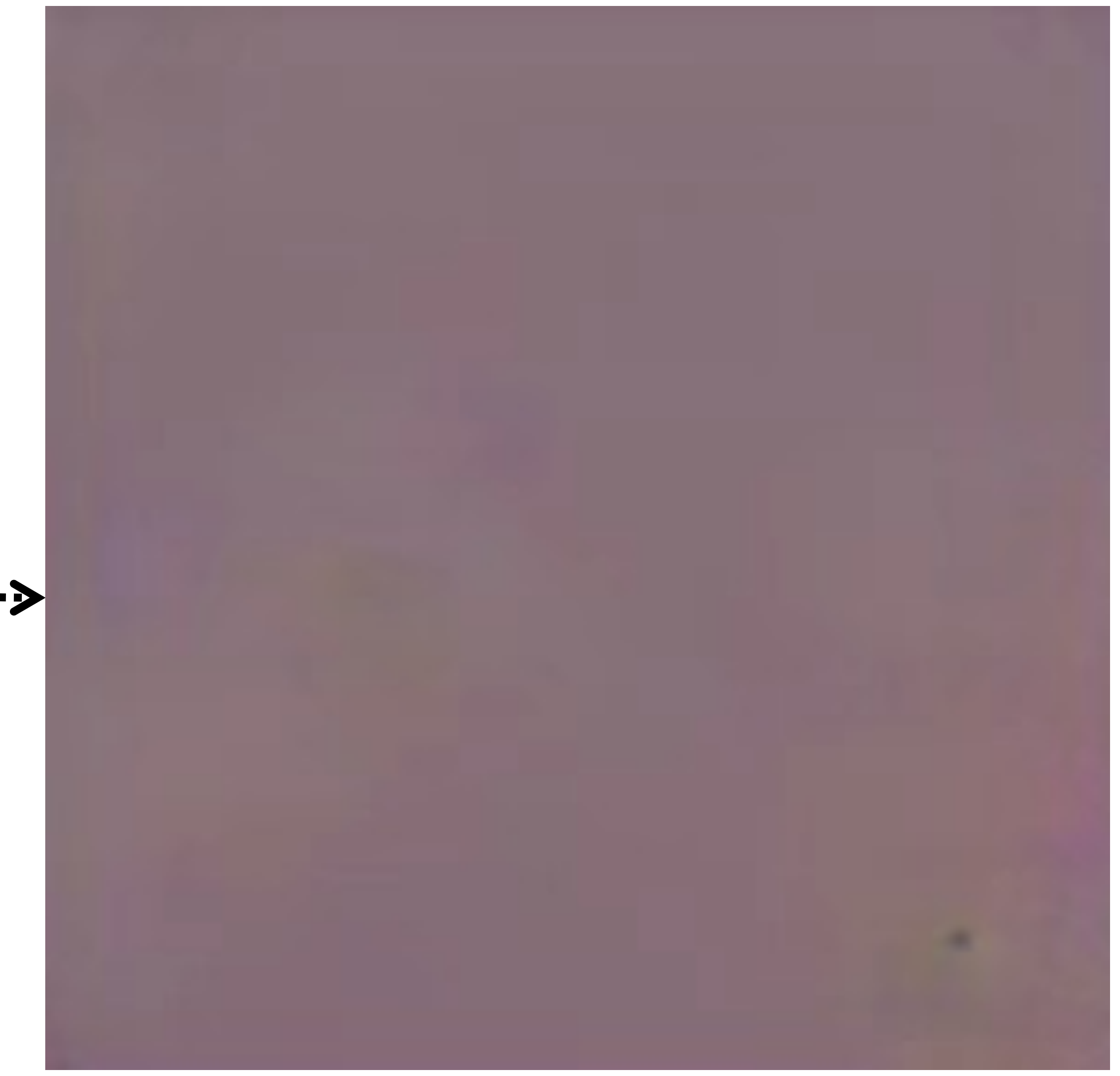
Differentiable program that measures the similarity between text and images

```
"What is the answer to the
ultimate question of life,
the universe, and everything?"
```



To maximize this

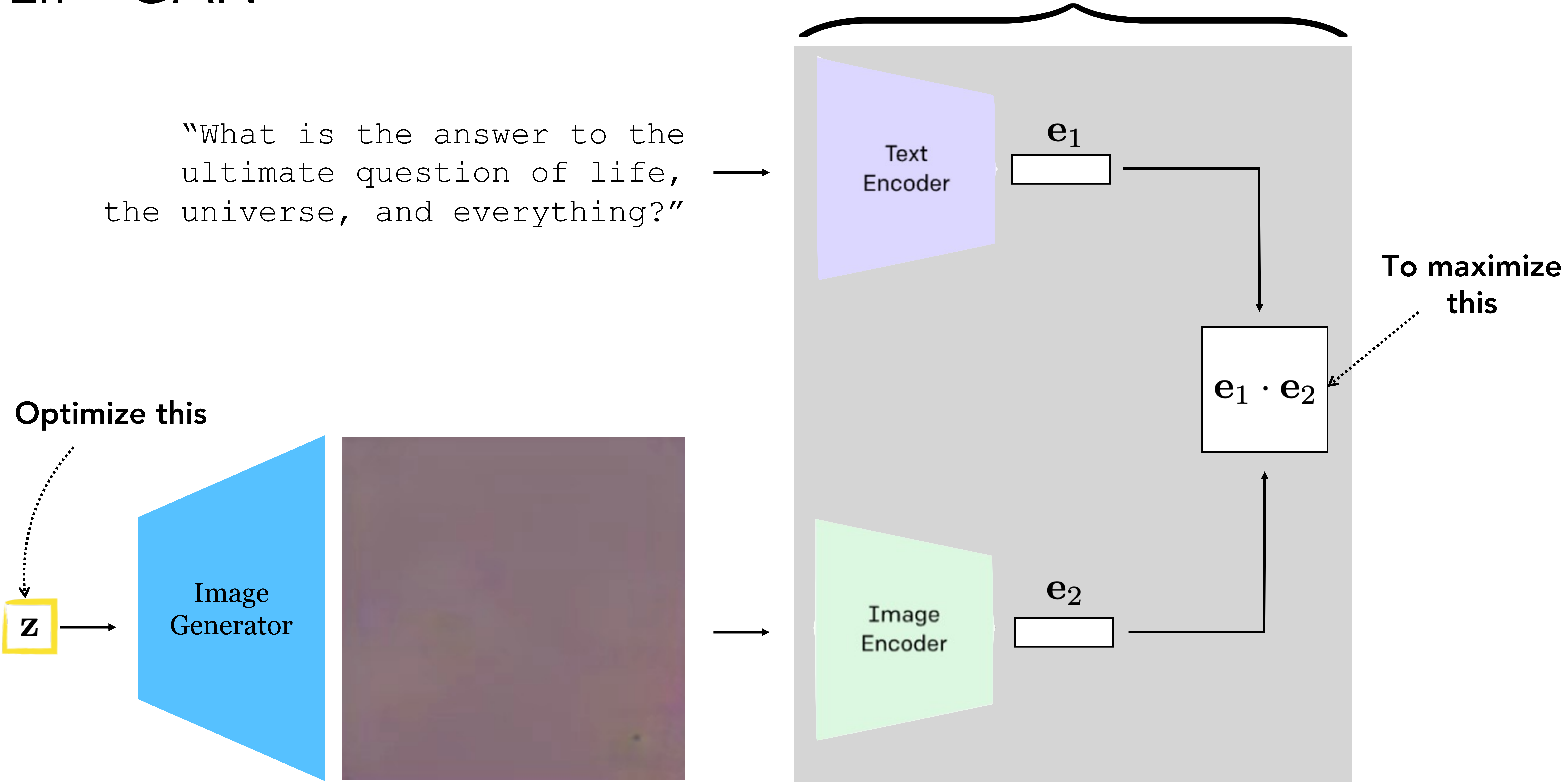
Optimize this



Code: https://colab.research.google.com/drive/1_4PQqzM_0KKytCzWtn-ZPi4cCa5bwK2F?usp=sharing

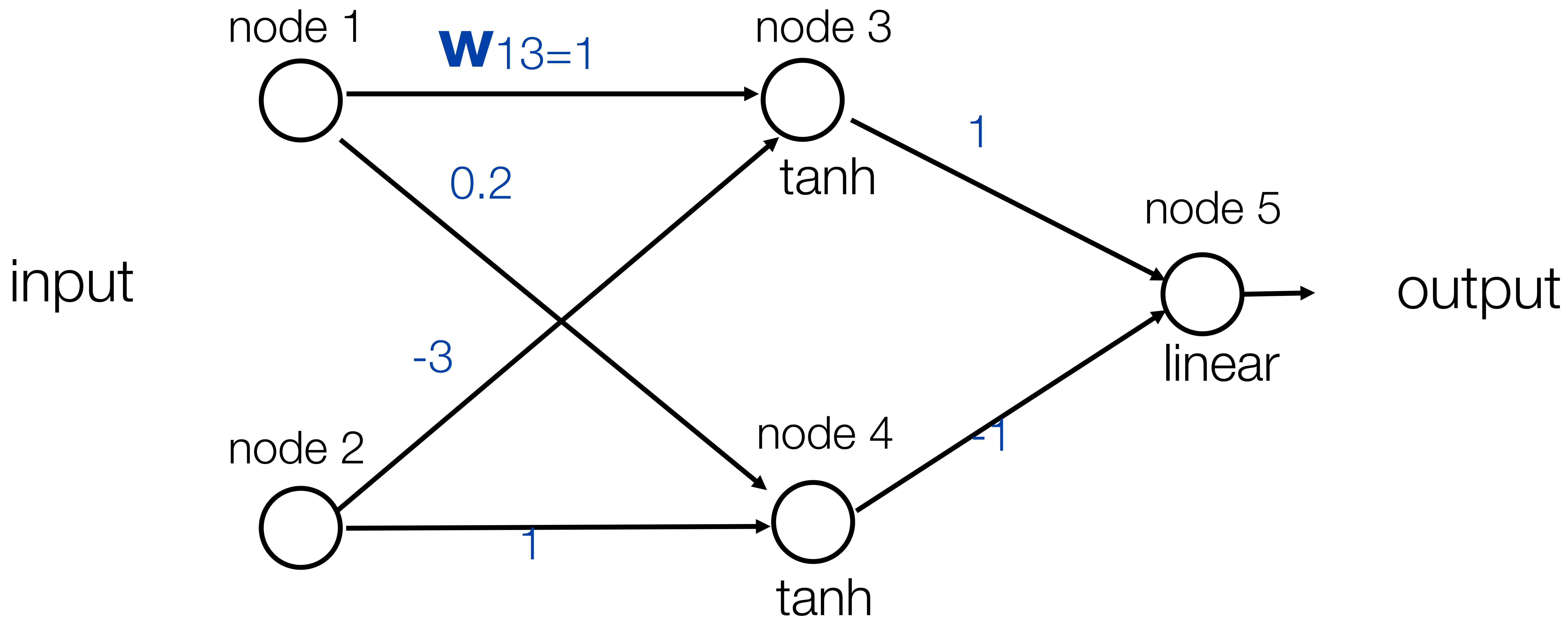
CLIP+GAN

Differentiable program that measures the similarity between text and images



Code: https://colab.research.google.com/drive/1_4PQqzM_0KKytCzWtn-ZPi4cCa5bwK2F?usp=sharing

Backpropagation example



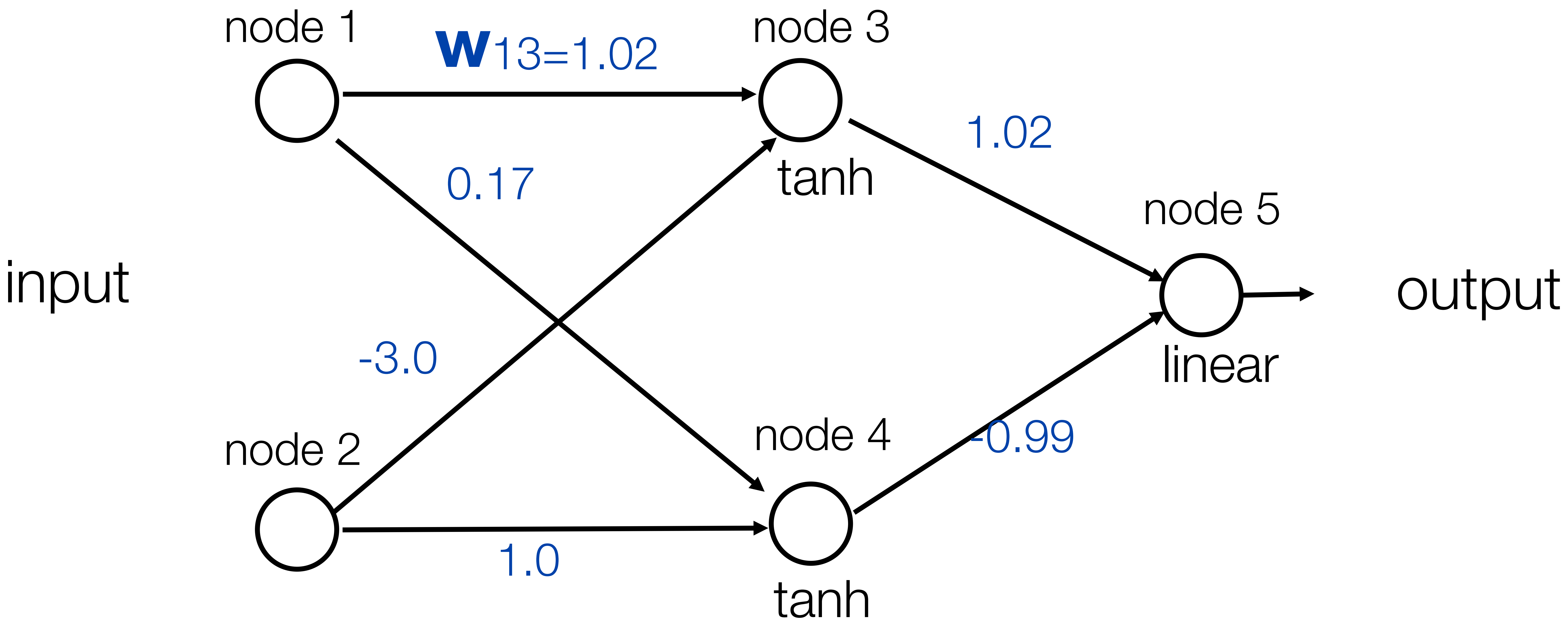
Learning rate η = -0.2 (because we used positive increments)

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

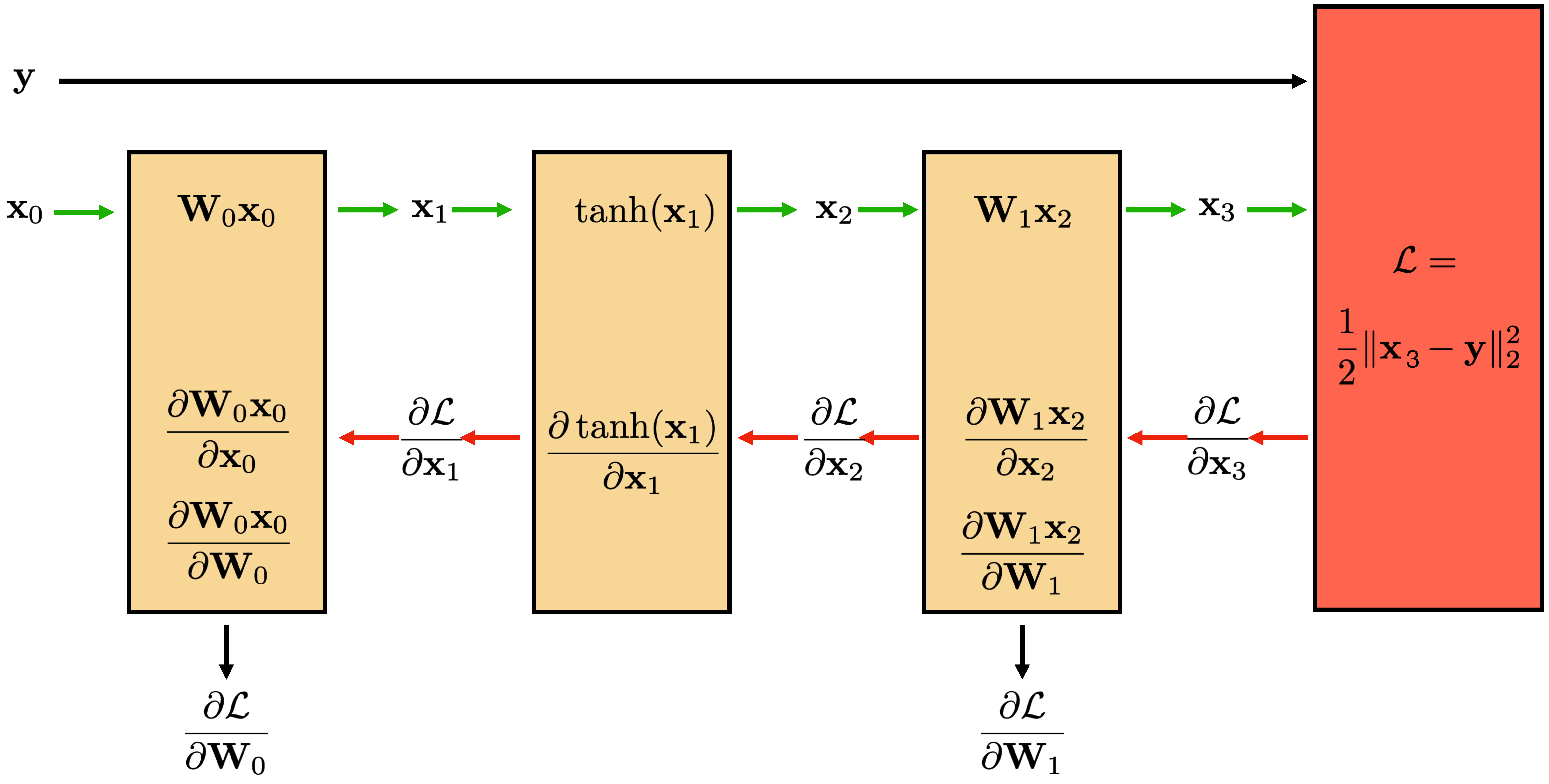
Backpropagation example



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

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

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 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:

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

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

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

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

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

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

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

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

Gradient updates:

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

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