Lecture 2: How to train a neural net

Speaker: Sara Beery

Announcements

- Pset 1 out — due 9/24
- OH starting this week, see webpage for locations and times
- Pytorch tutorials this week

2. How to train a neural net

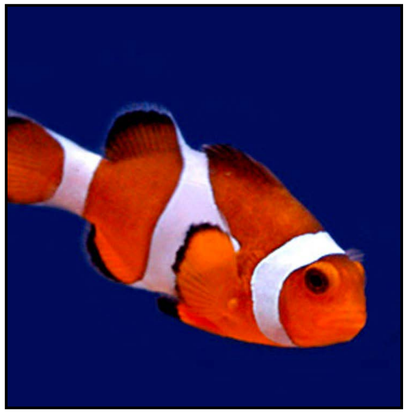
- Review of gradient descent, SGD
- Computation graphs
- Backprop through chains
- Backprop through MLPs
- Backprop through DAGs
- Differentiable programming

Deep learning

$\mathbf{y}^{(i)}$

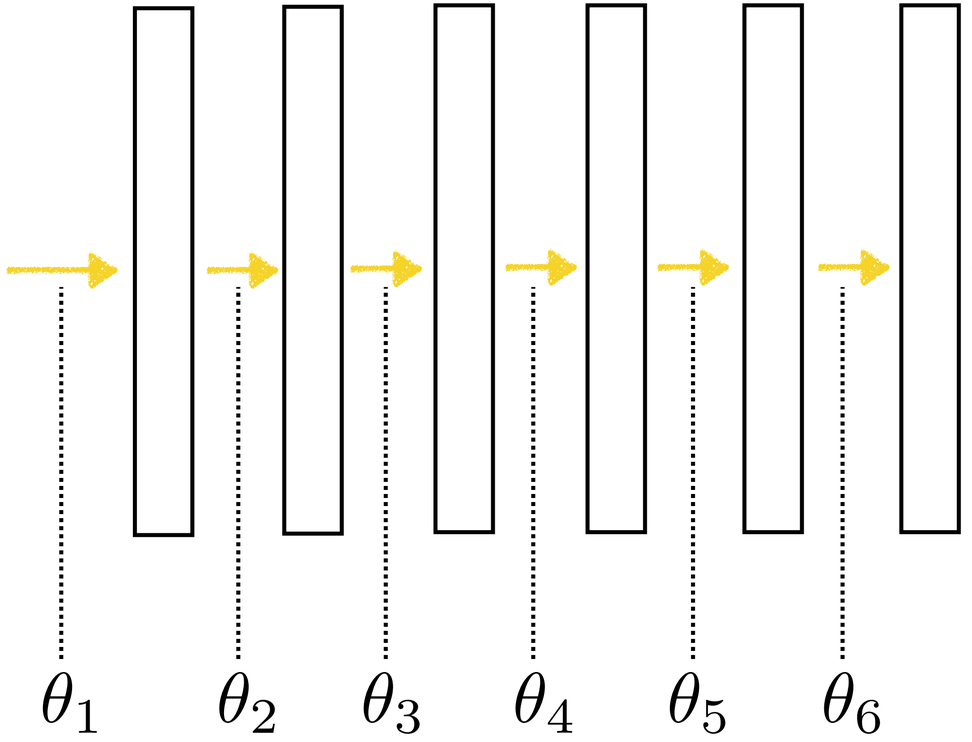
`clown fish`

$\mathbf{x}^{(i)}$



Clown fish © source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/

Learned



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

Gradient Descent

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

Optimization

Params

θ → J → $J(\theta)$

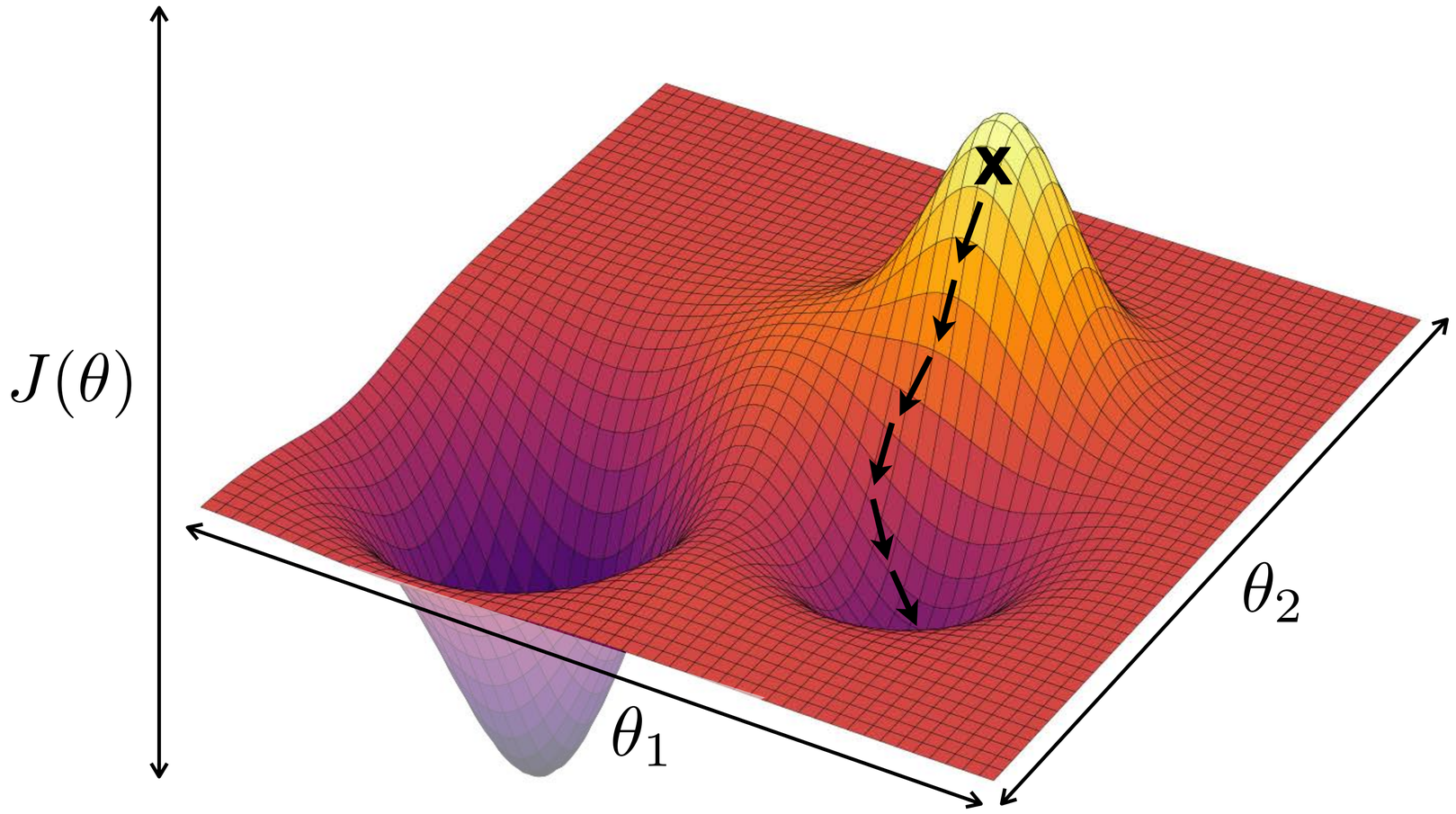
$\nabla_\theta J(\theta)$

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

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

- 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 (**Gradient**)
  - We can evaluate $J(\theta)$, $\nabla_\theta J(\theta)$, and $H_\theta(J(\theta))$ ← Second order optimization (**Hessian**)

Gradient Descent



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

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:

$$\theta^{k+1} \quad \theta^k - \eta \nabla_\theta J(\theta^k)$$

learning rate

Stochastic Gradient Descent (SGD)

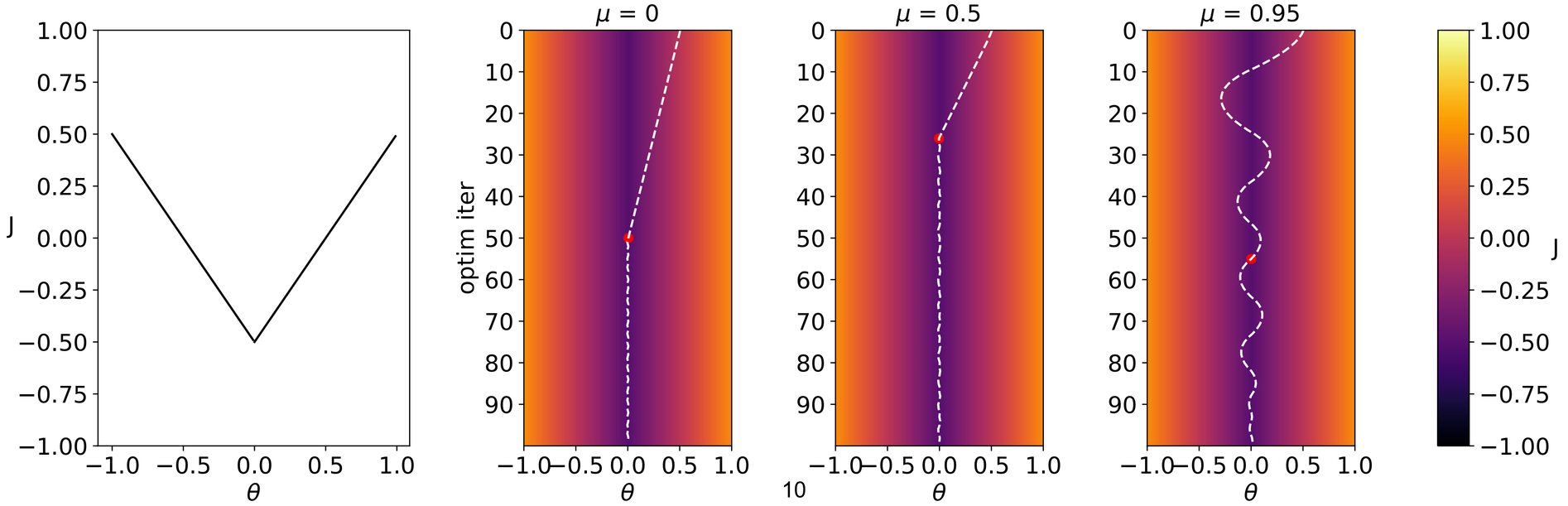
- Want to minimize overall loss function **J**, which is sum of individual losses over each example.
- In Stochastic gradient descent, compute gradient on sub-set (batch) of data.
  - If batchsize=1 then θ is updated after each example.
  - If batchsize=N (full set) then this is standard gradient descent.
- Gradient direction is noisy, relative to average over all examples (standard gradient descent).
- Advantages
  - Faster: approximates total gradient with small sample
  - Implicit regularizer
- Disadvantages
  - High variance, unstable updates

Momentum

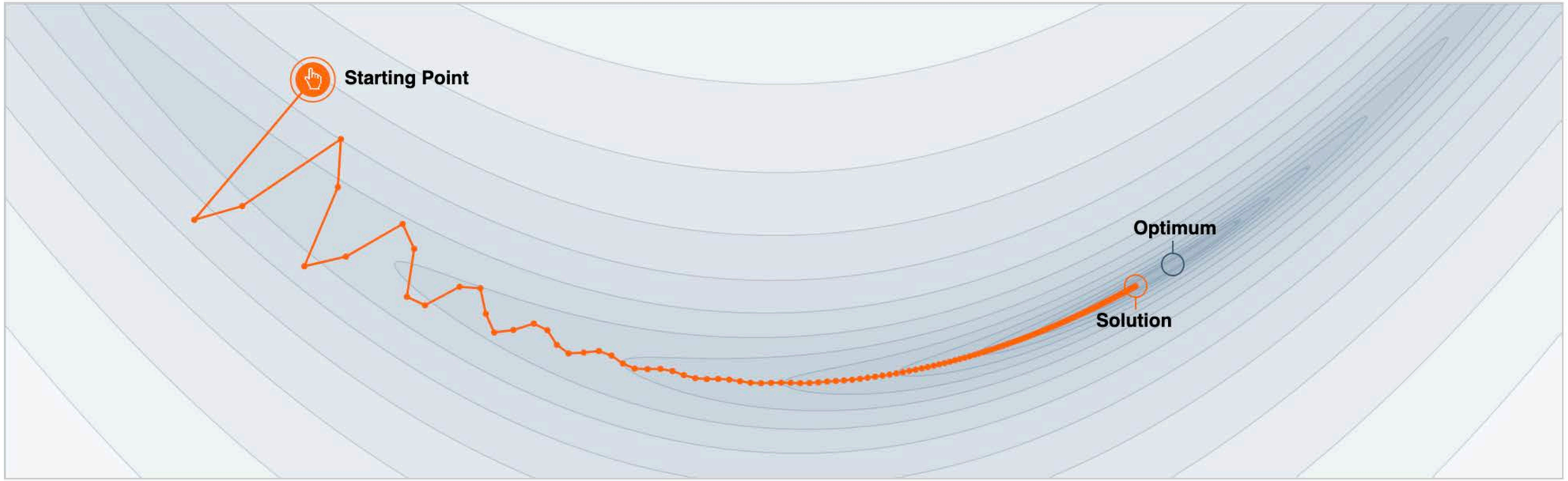
- A heavy ball rolling down a hill, gains speed.
- Gradient steps biased to continue in direction of previous update:

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

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



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?

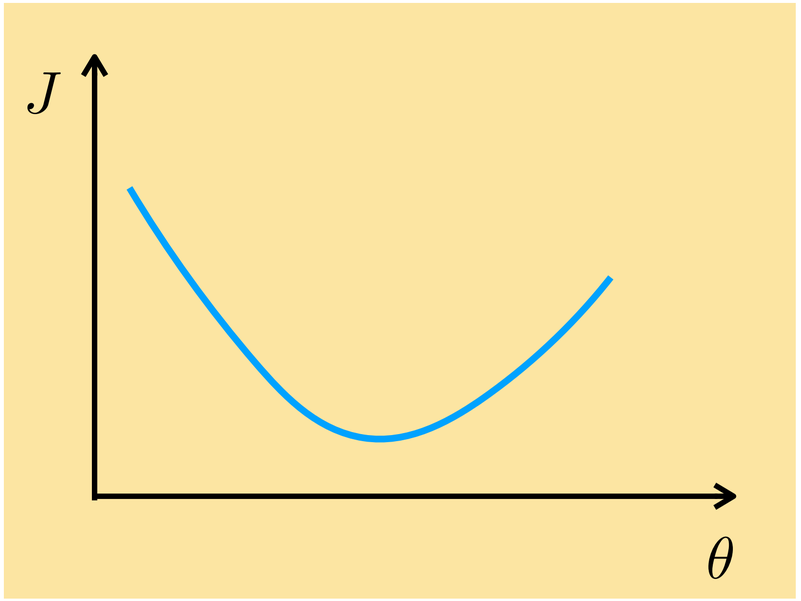
GABRIEL GOH
UC Davis

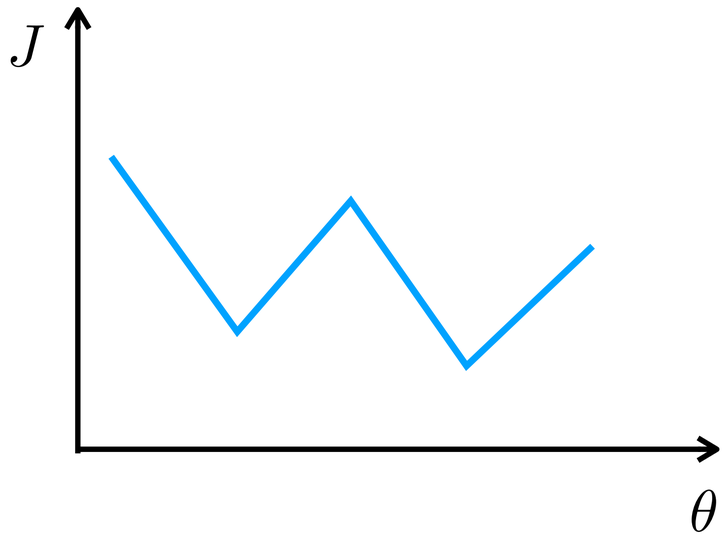
April. 4
2017

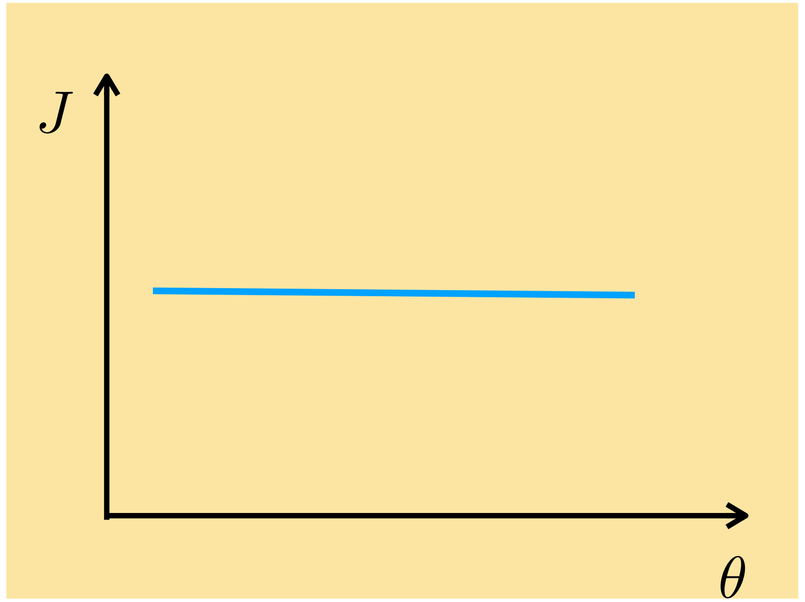
Citation:
Goh, 2017

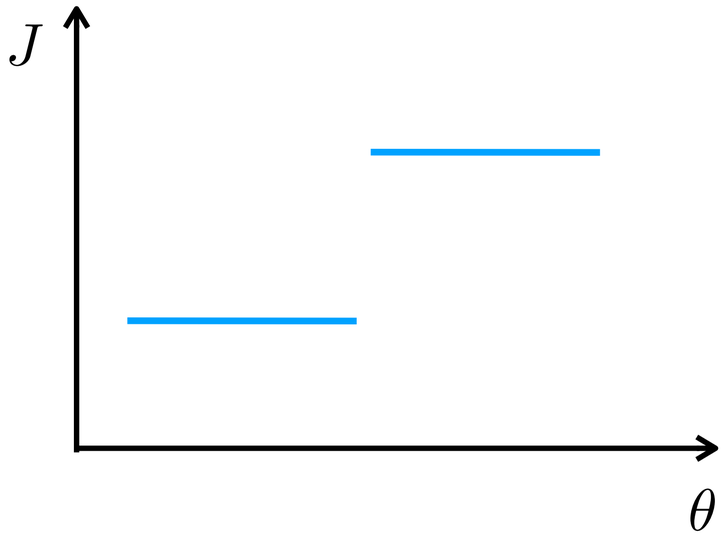
https://distill.pub/2017/momentum/

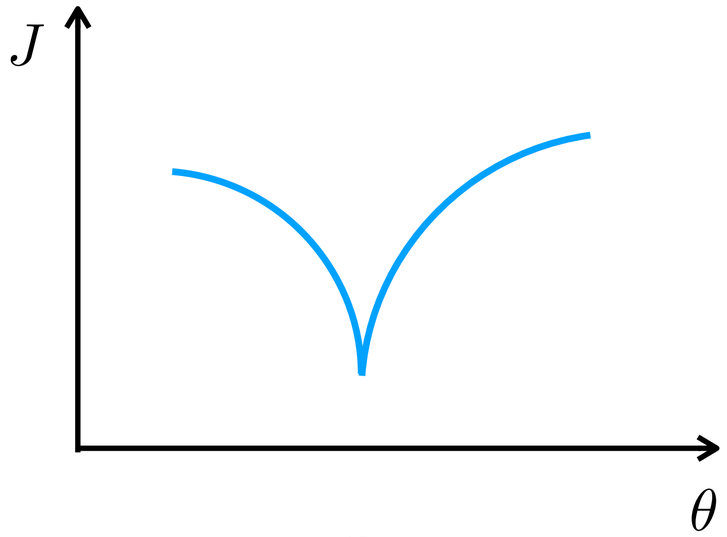
Which are differentiable?

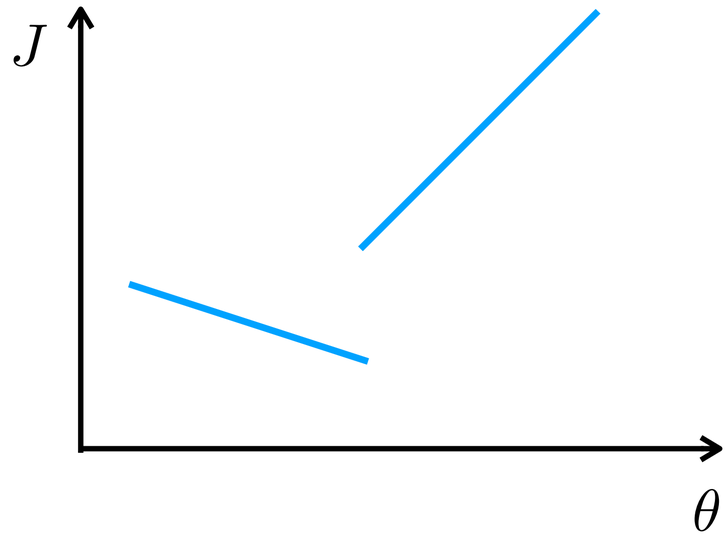




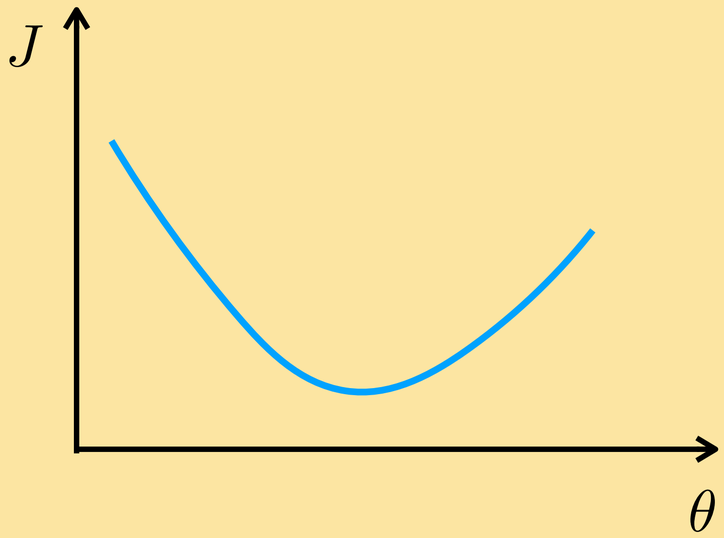


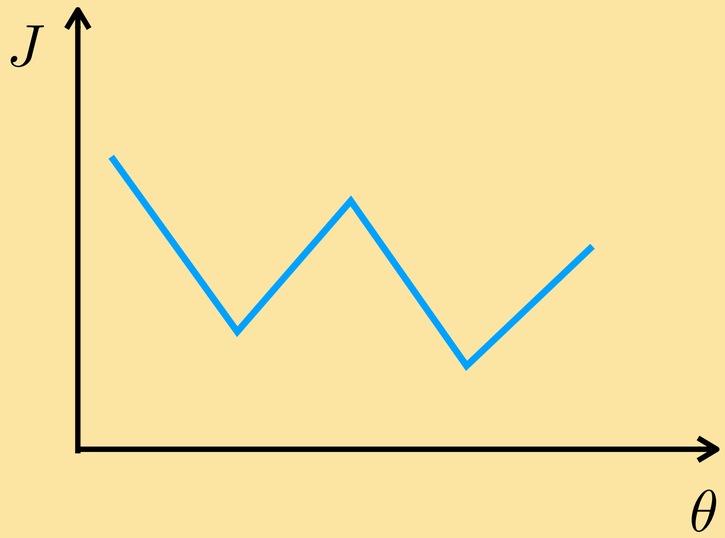


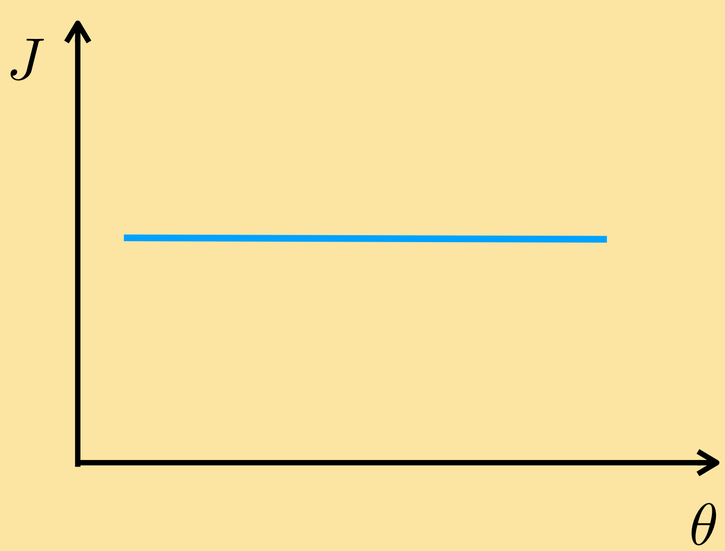


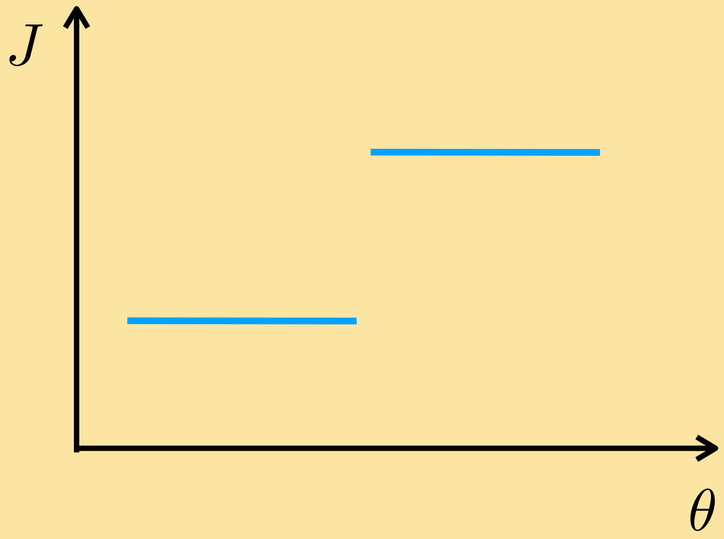


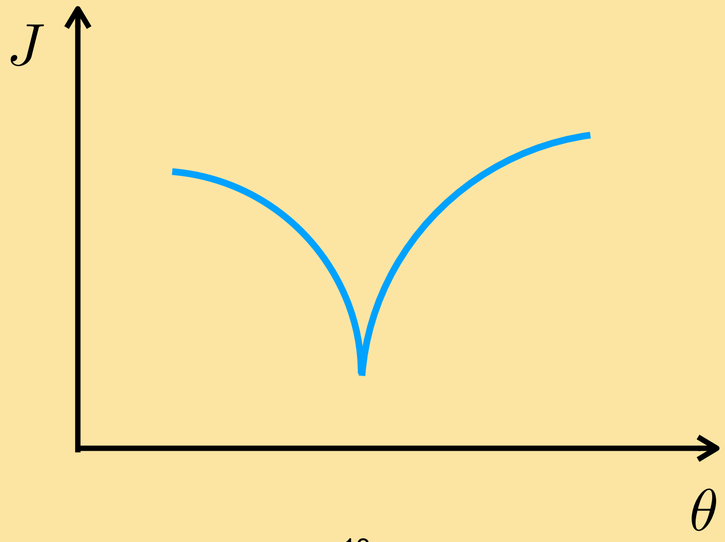
Which are have defined gradients in pytorch?

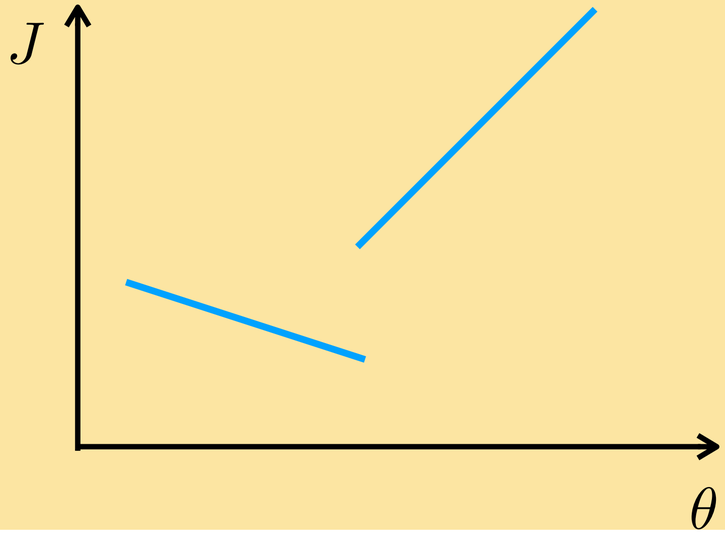




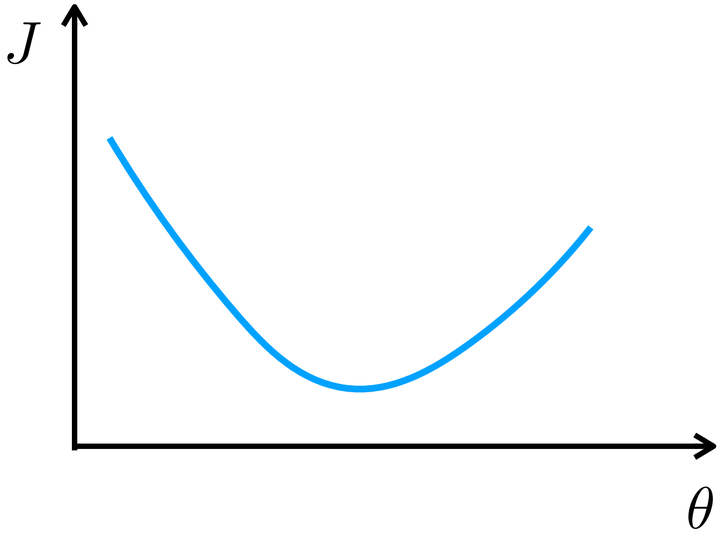


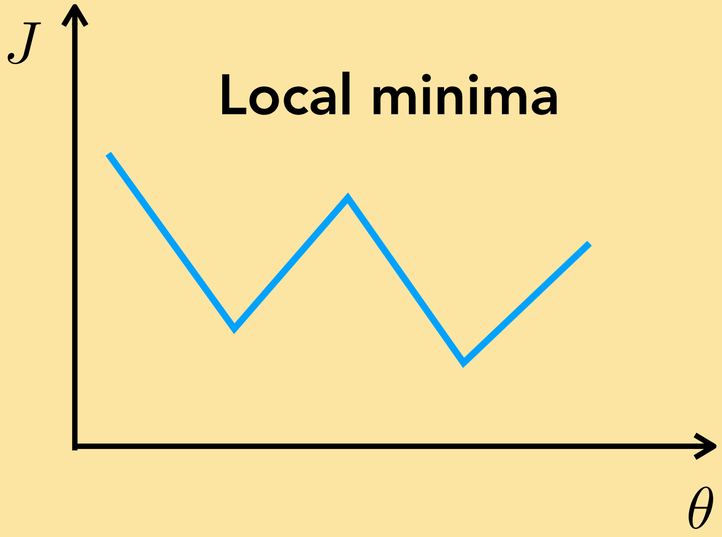


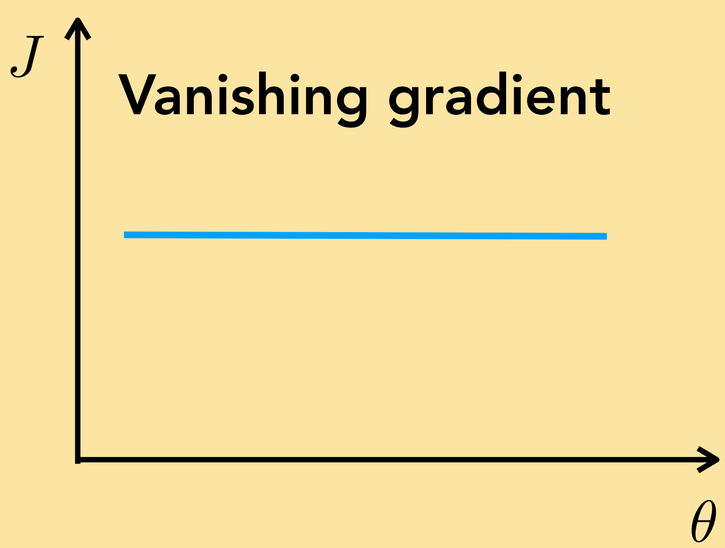


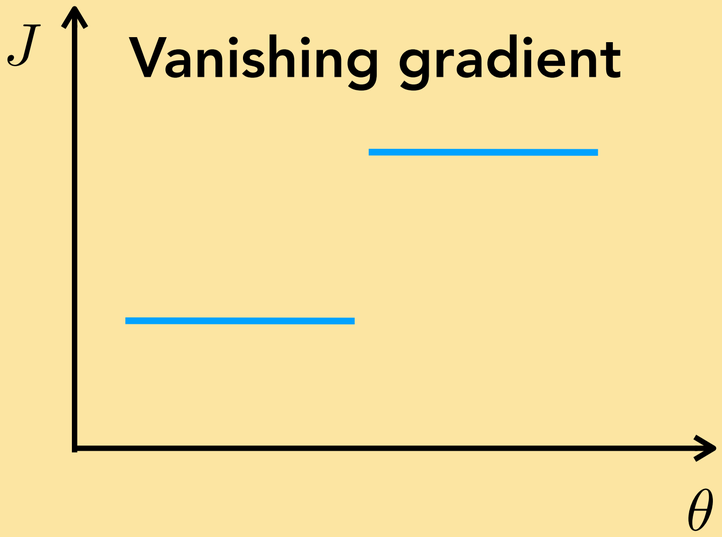


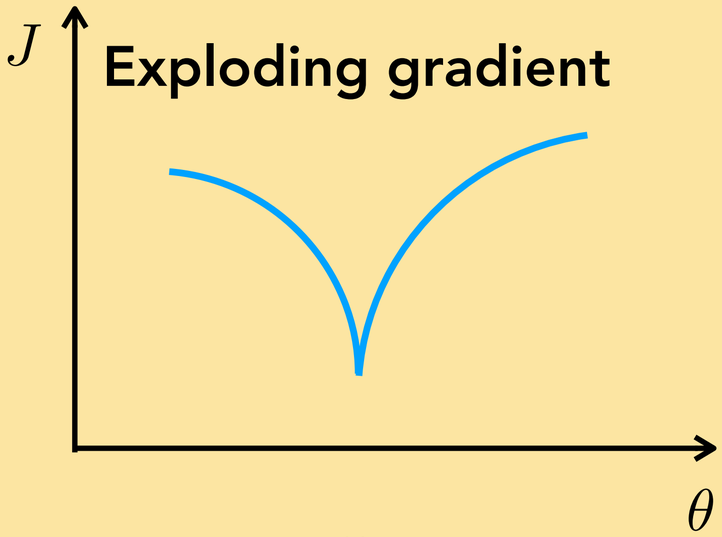
Which will be hard to optimize?

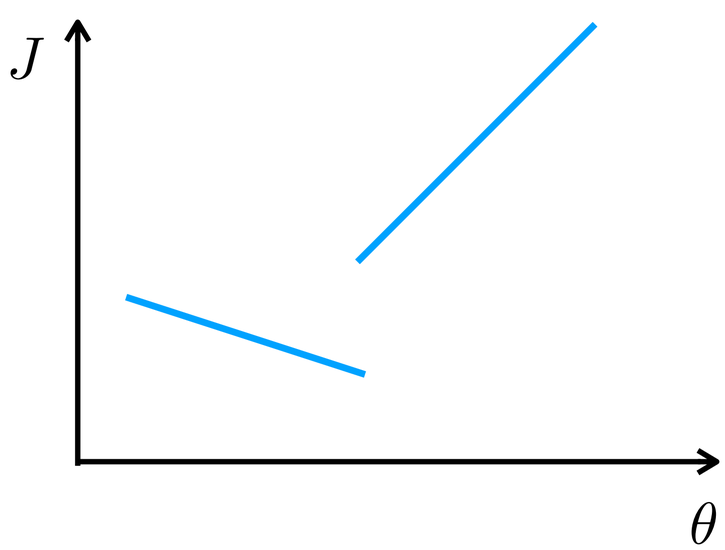






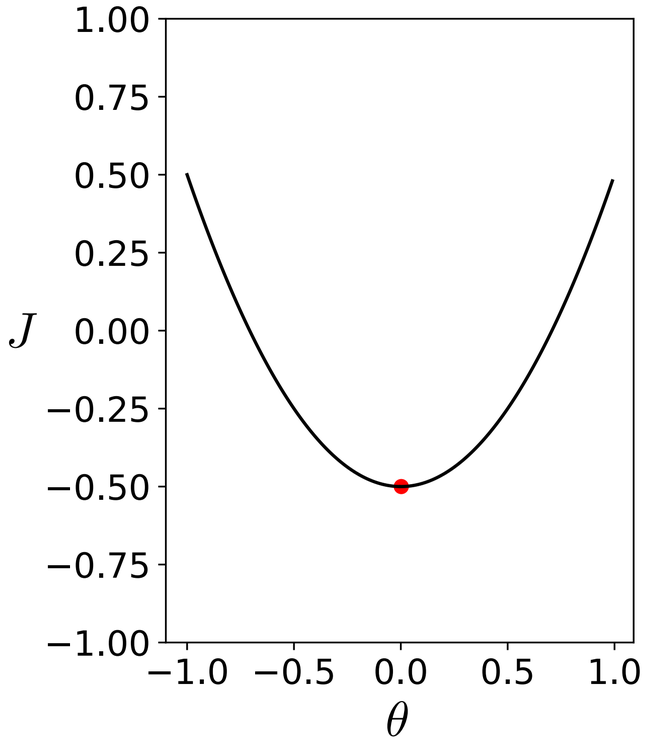


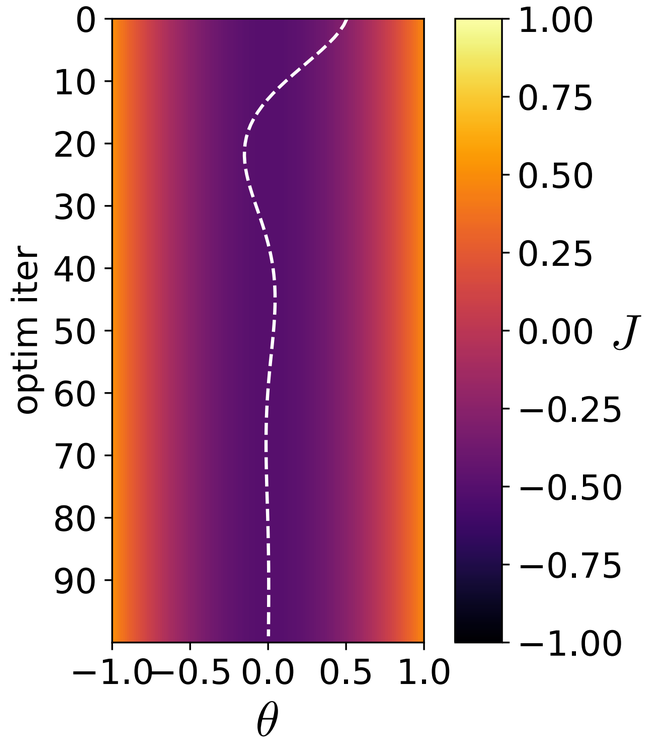




Simple case:

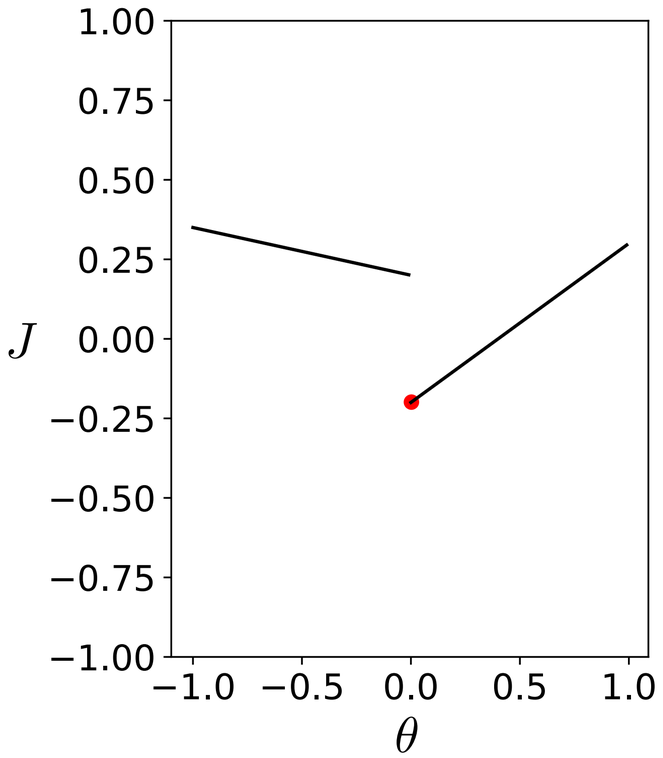
- Convex
- Single minimum
- Gradients point toward it everywhere
- Gradient gracefully goes to zero as minimum is approached

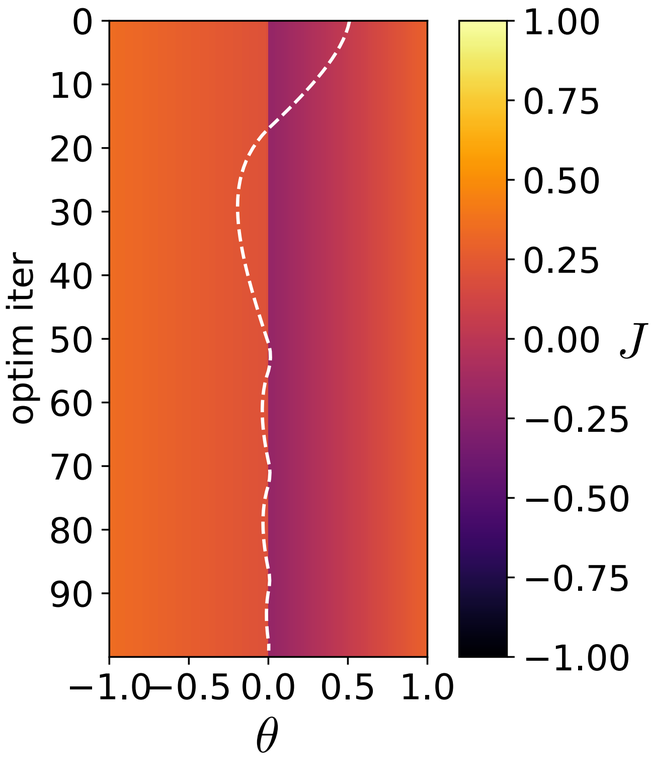




Discontinuous:

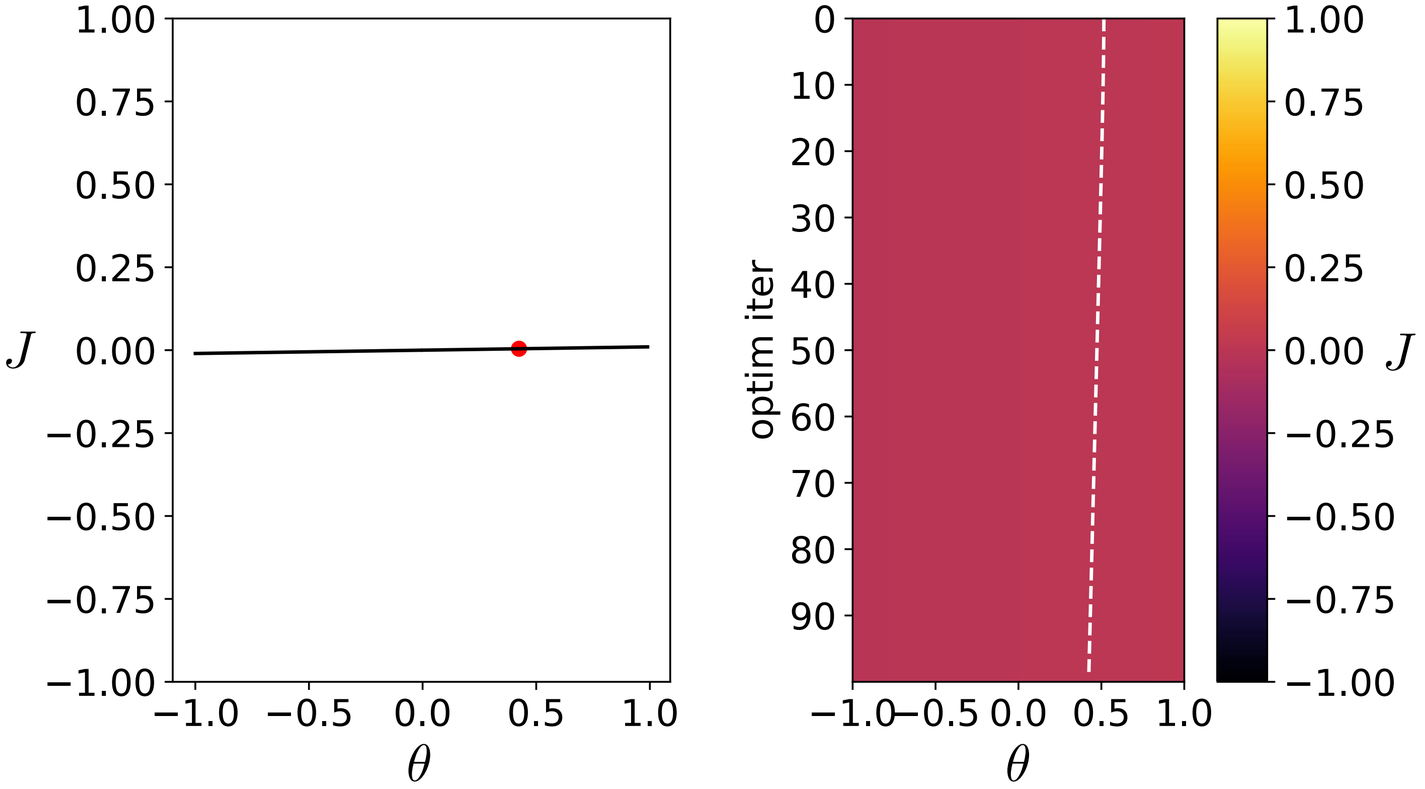
- But well-defined one-sided derivatives
- Not a problem for Pytorch





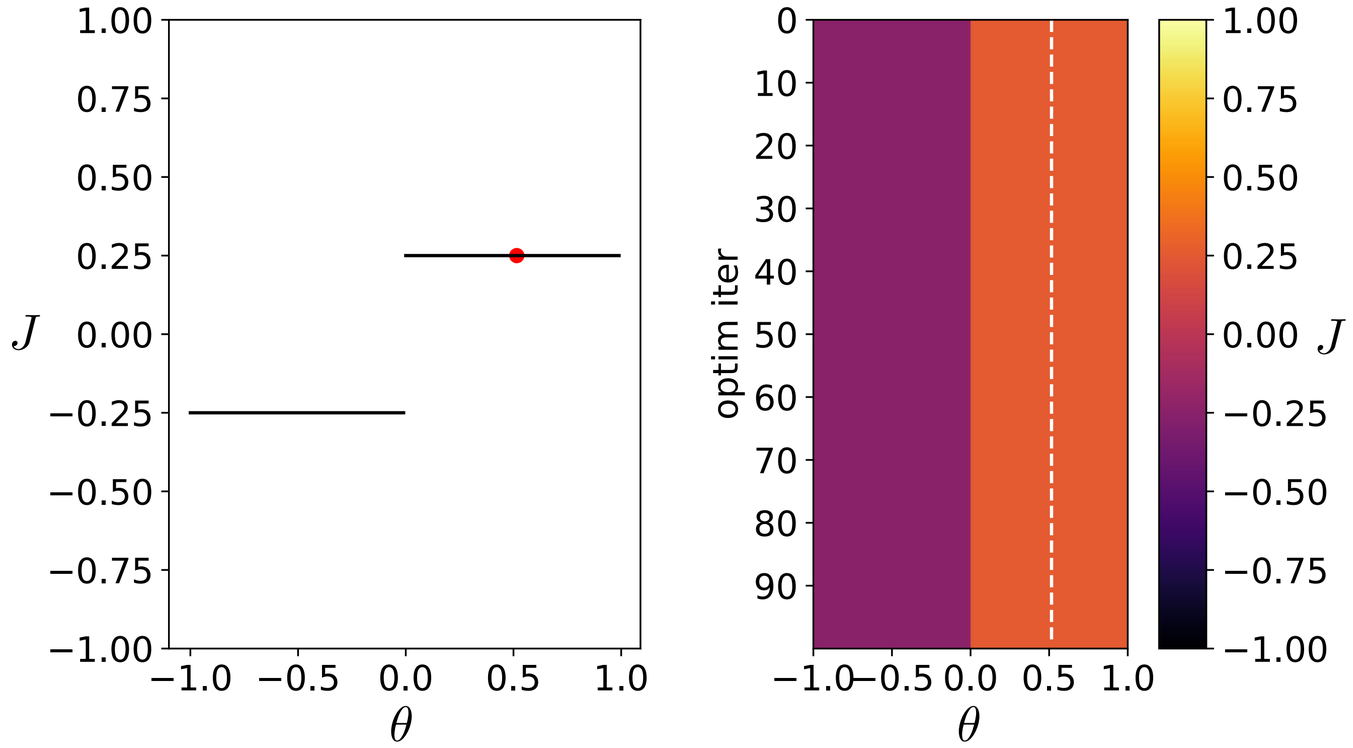
Vanishing gradient:

- progress is slow, noise may dominate



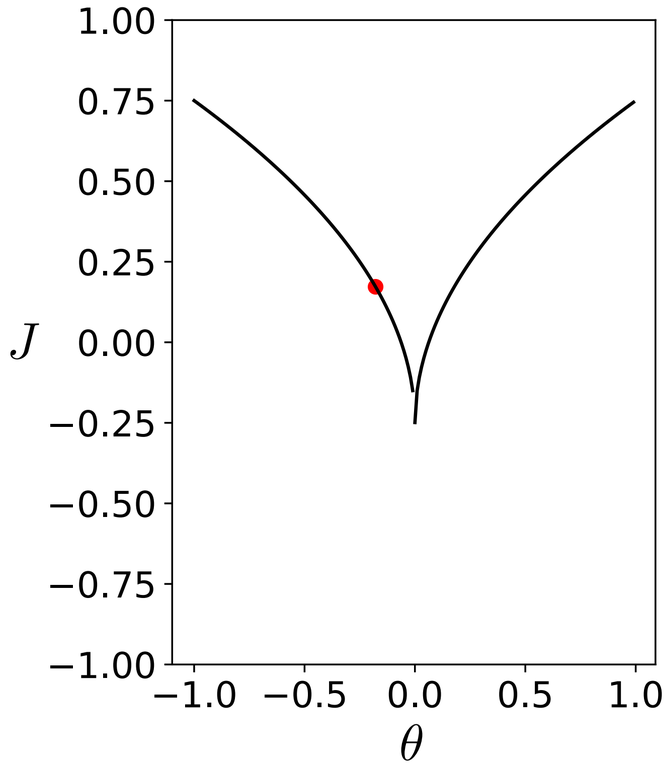
Zero gradient:

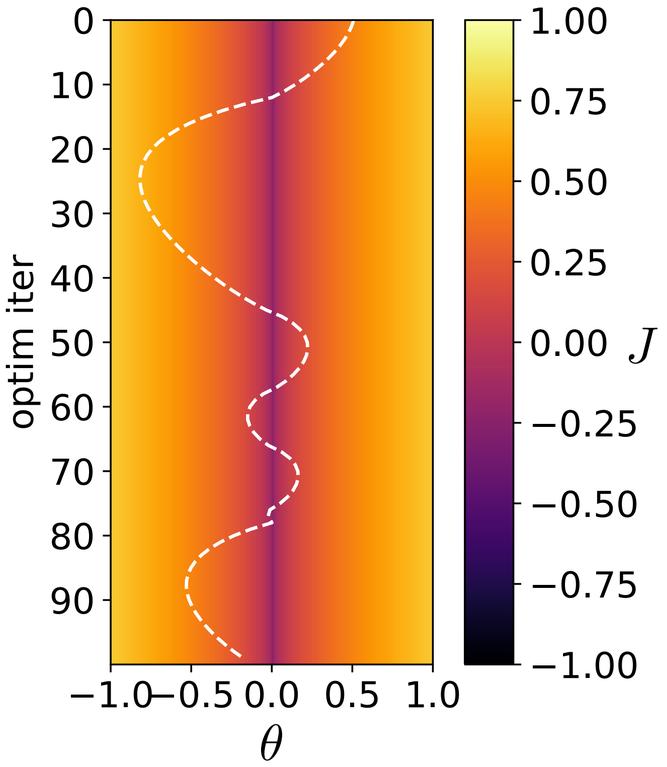
- Gradient is completely uninformative as to how to make progress
- Low loss region is never reached



Exploding gradient:

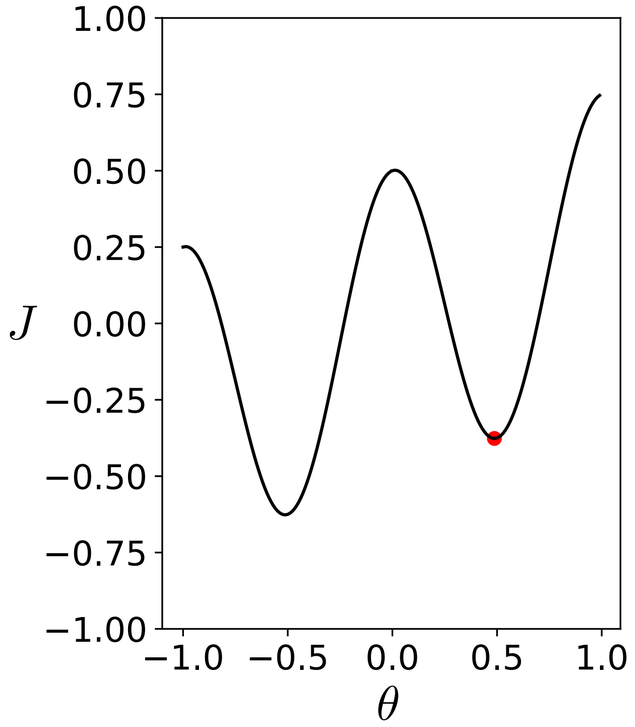
- Gradient goes to infinity as minimizer is approached
- Unstable updates, overshoots

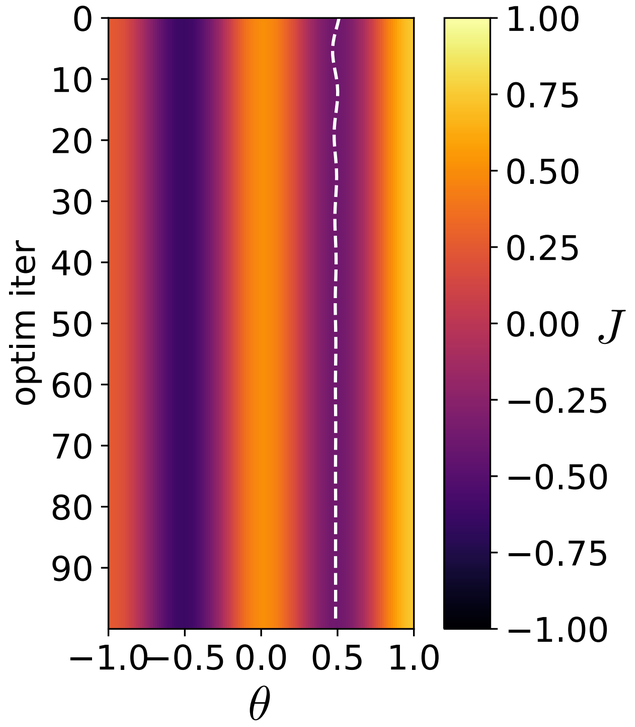




Multiple local minima:

- Where you initialize matters
- Gradient descent does not guarantee reaching global minimizer
- Reaches a local minimizer





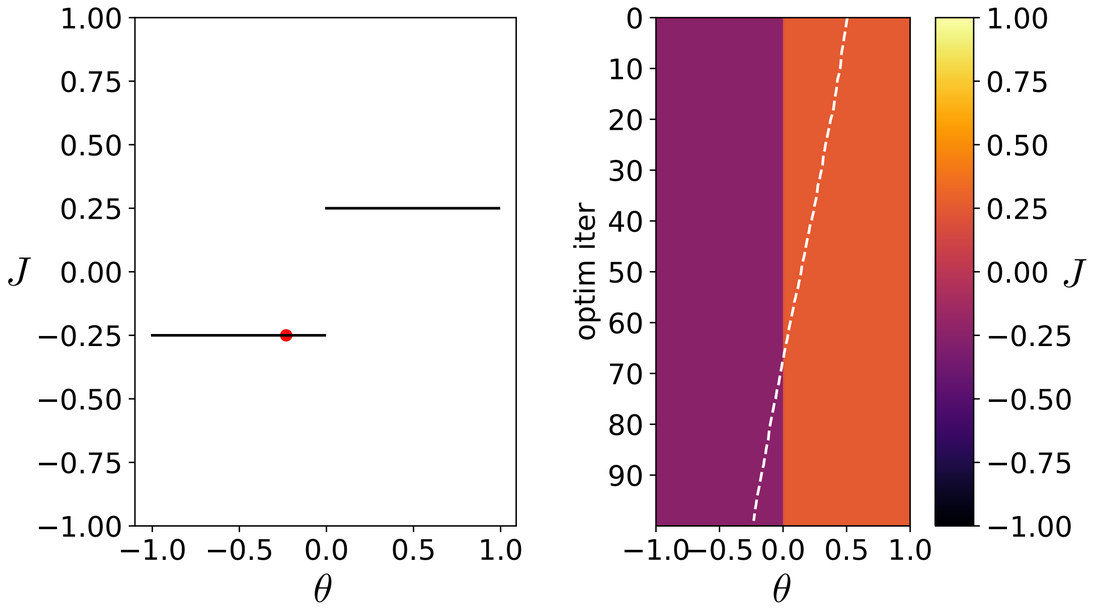
Evolution Strategies

- Gradient-like: finds a locally loss-minimizing direction in parameter space
- Sample small perturbations of θ and move toward perturbations that achieved lower loss

$$\epsilon_i \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$
$$s_i = J(\theta + \sigma\epsilon_i)$$

$$\theta^{k+1} \quad \theta^k - \eta \frac{1}{\sigma M} \sum_{i=1}^{M} s_i \epsilon_i$$

Successfully minimizes this function:

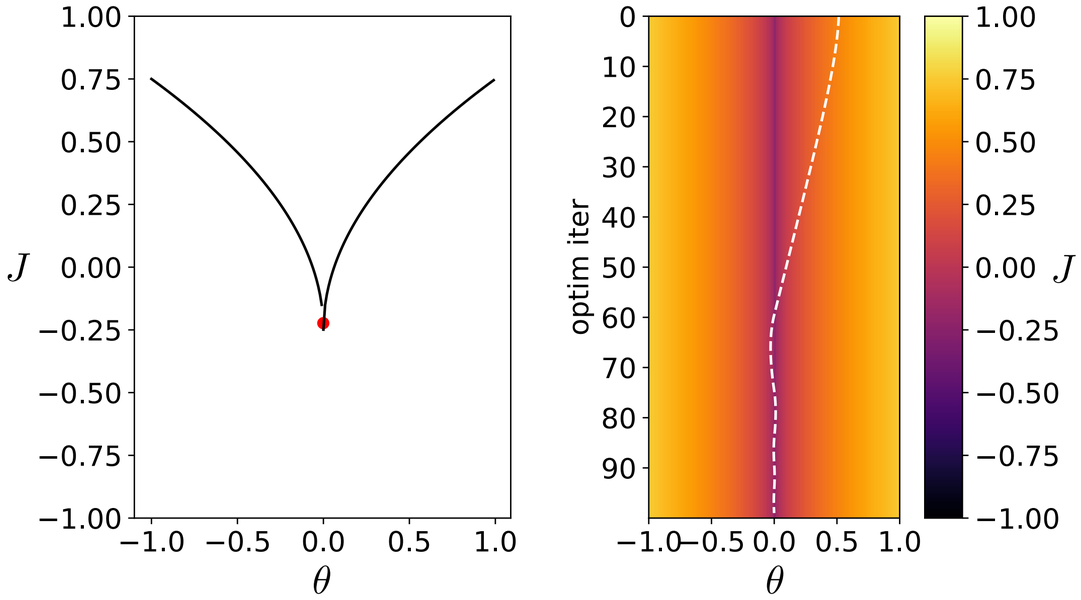


Gradient clipping

- If gradients exceed a magnitude m, scale them to magnitude m
- Useful, and commonly used hack

$$\mathbf{v} = \nabla_\theta J(\theta^k)$$
$$\theta^{k+1} \quad \theta^k - \eta[\texttt{clip}(v_1, -m, m), \ldots, \texttt{clip}(v_M, -m, m)]^\mathsf{T}$$

Successfully minimizes this function:



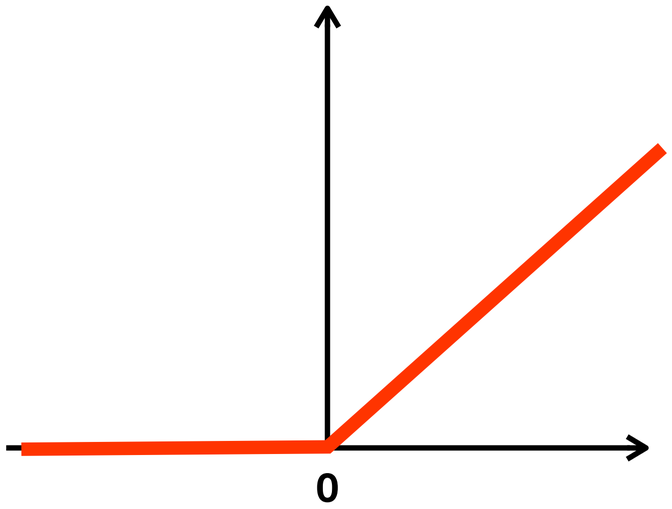
What is important in a loss function?

- Everywhere continuous
- Everywhere differentiable
- Everywhere smooth

What is important in a loss function?

- Everywhere continuous ✓
- Everywhere differentiable (Almost!)
- Everywhere smooth ✗

ReLU

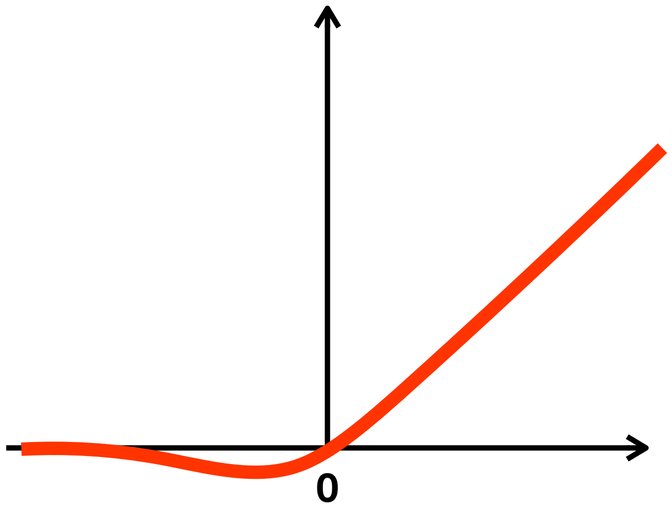


$$\mathrm{ReLU}(z) = \max(0, z)$$

What is important in a loss function?

- Everywhere continuous ✓
- Everywhere differentiable ✓
- Everywhere smooth ✓

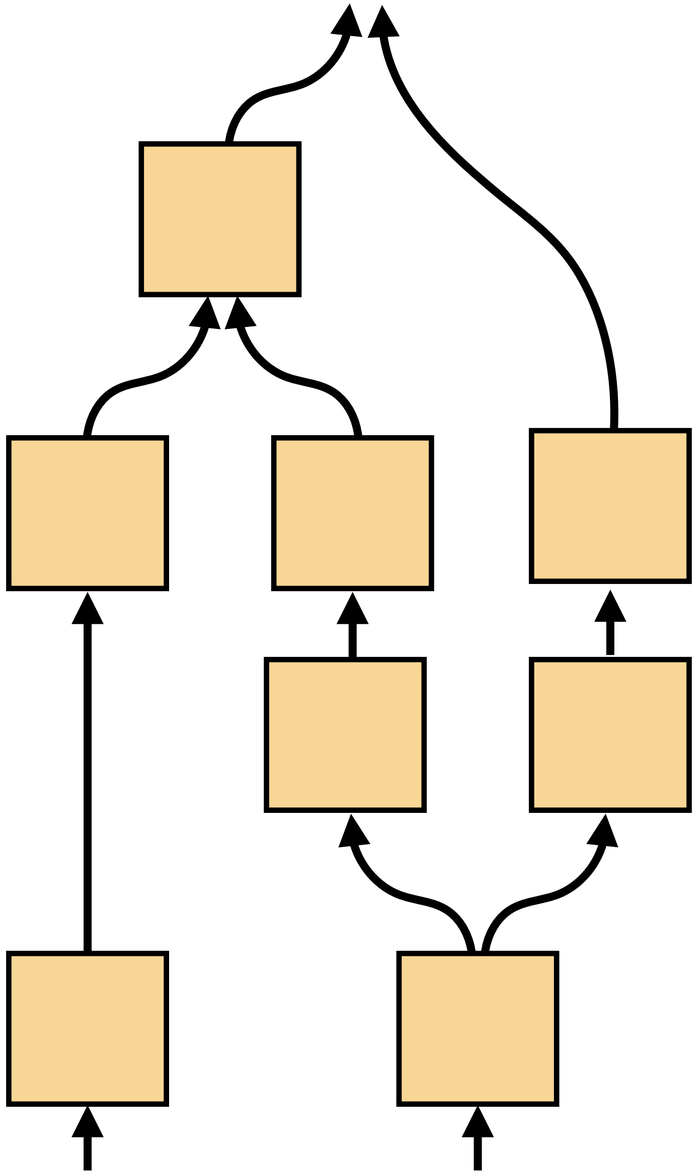
GeLU



$$\mathrm{GELU}(z) = z * \Phi(z)$$

https://arxiv.org/abs/1606.08415

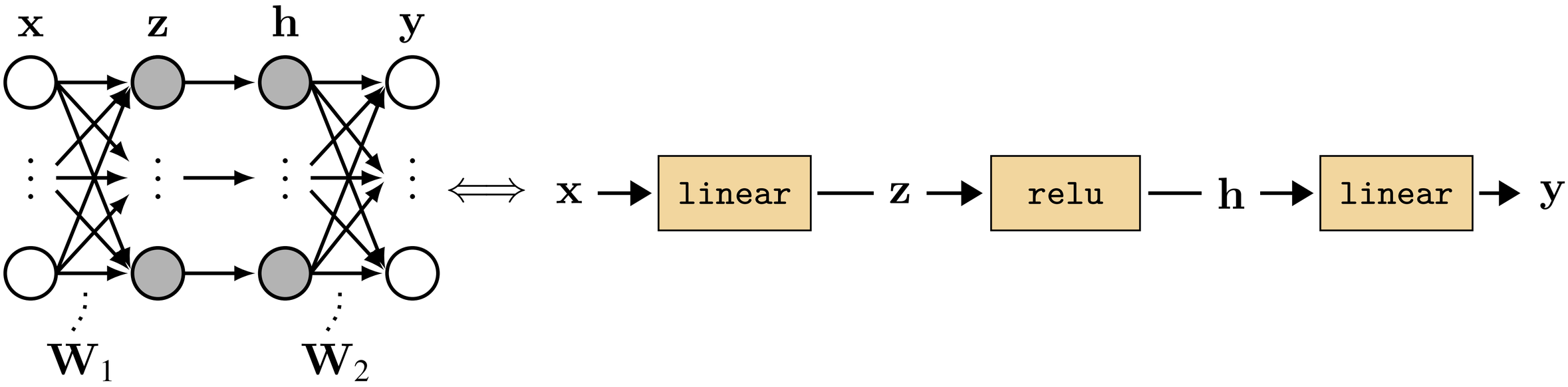
Computation Graphs



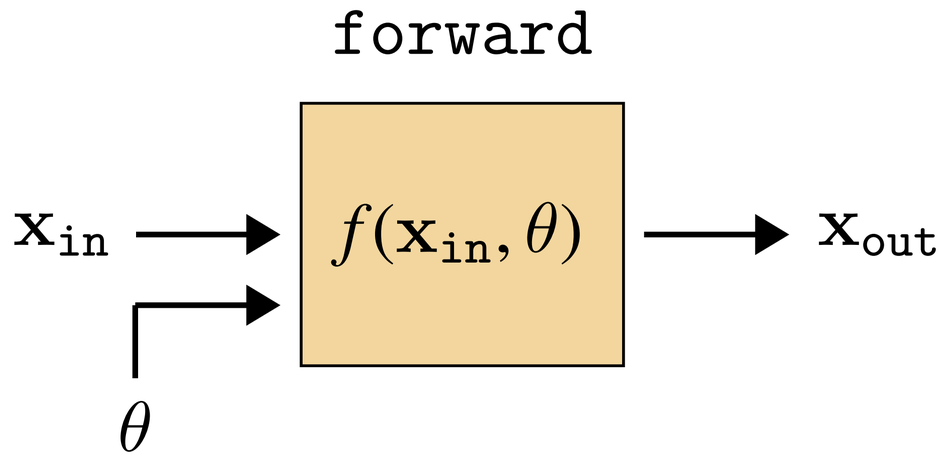
A graph of functional transformations, nodes ( ), that when strung together perform some useful computation.

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

Computation Graphs

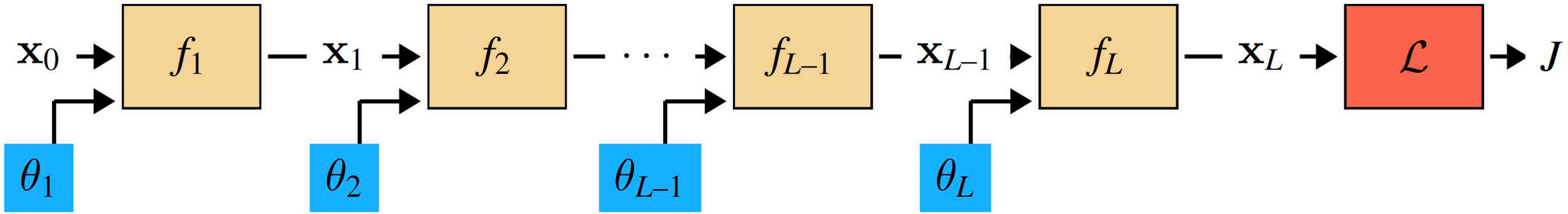


Forward pass



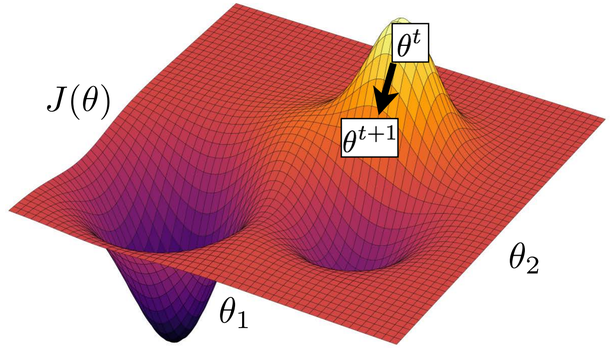
$$\mathbf{x}_{\text{out}} = f(\mathbf{x}_{\text{in}}, \theta)$$

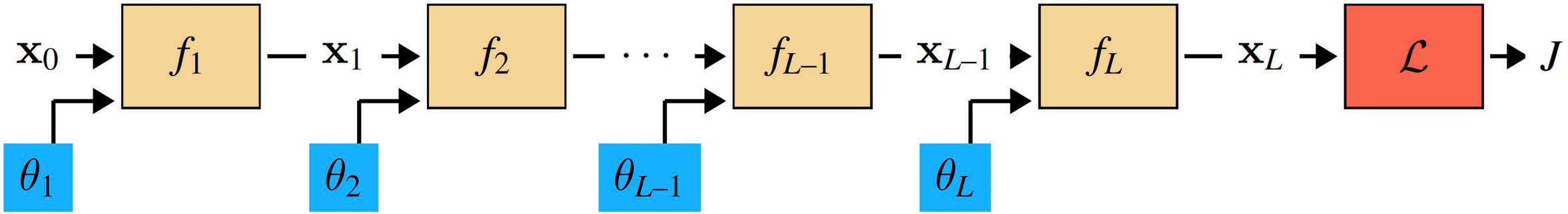
Forward pass — multiple layers



- This computation graph could represent an MLP, for example

Learning





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

(

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 $n = |\mathbf{x}|$

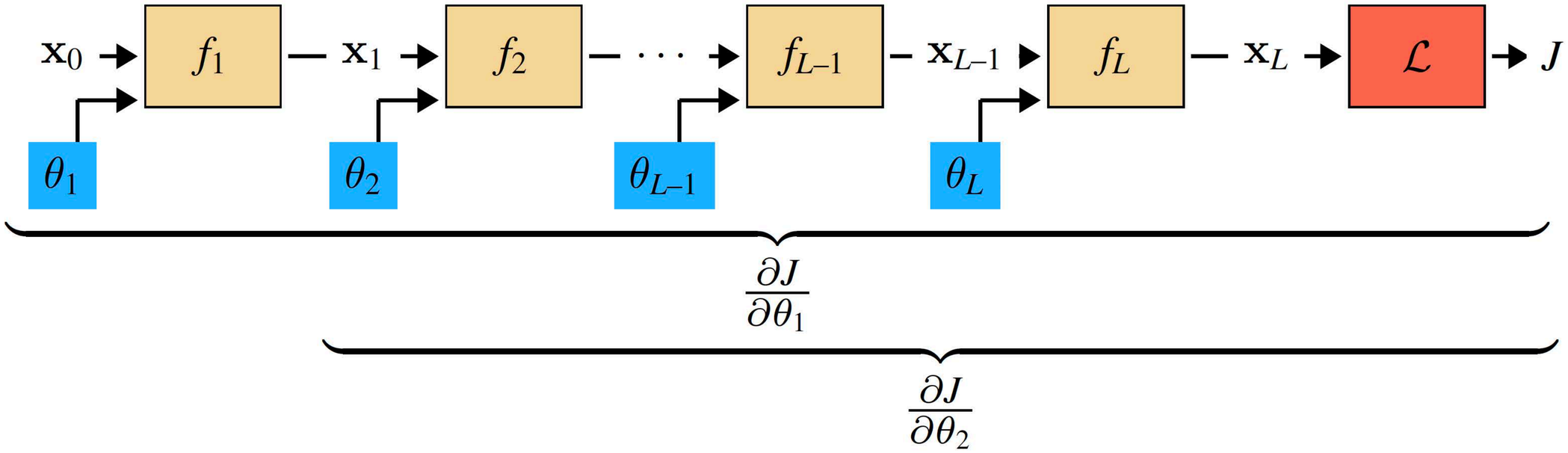
Example, if $|\mathbf{z}| = 1$, $|\mathbf{u}| = 2$, $|\mathbf{x}| = 4$

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

)

The Trick of Backpropagation — Reuse of Computation

(aka dynamic programming)

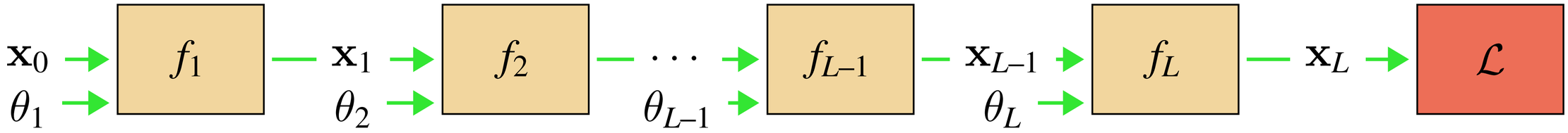


$$\frac{\partial J}{\partial \theta_1} = \frac{\partial J}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}}\cdots\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \theta_1}$$

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

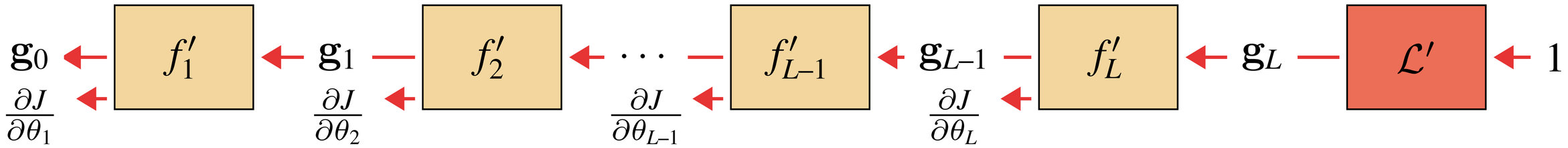
- We could separately compute all the derivatives using the chain rule.
- But the terms in the gray box are shared. So we should only compute this value once.
- **Backpropagation** is an algorithm for propagating shared terms throughout the computation graph

Forward pass



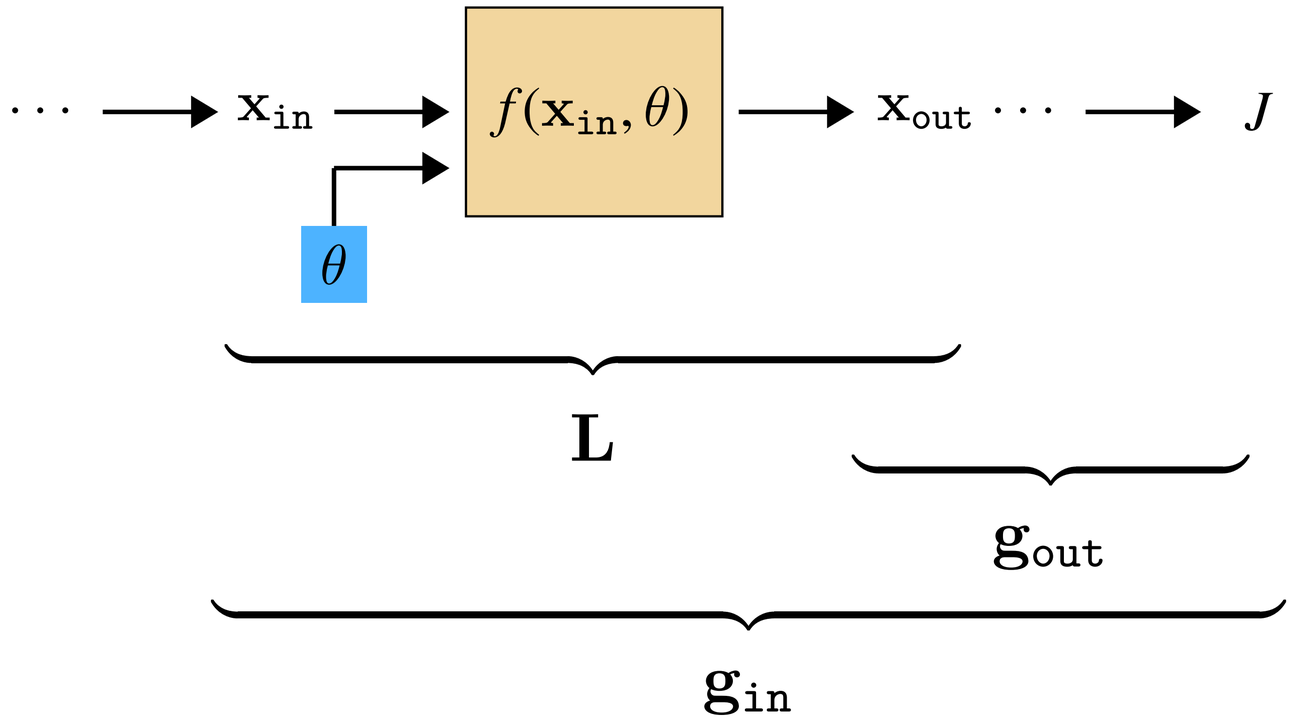
Send data forward through the network, computing outputs and calculating loss.

Backward pass



Send error signals (gradients) backwards through the network, from outputs and loss back to inputs and parameters.

Backward for a Generic Layer



We will keep track of two kinds of arrays of partial derivatives:

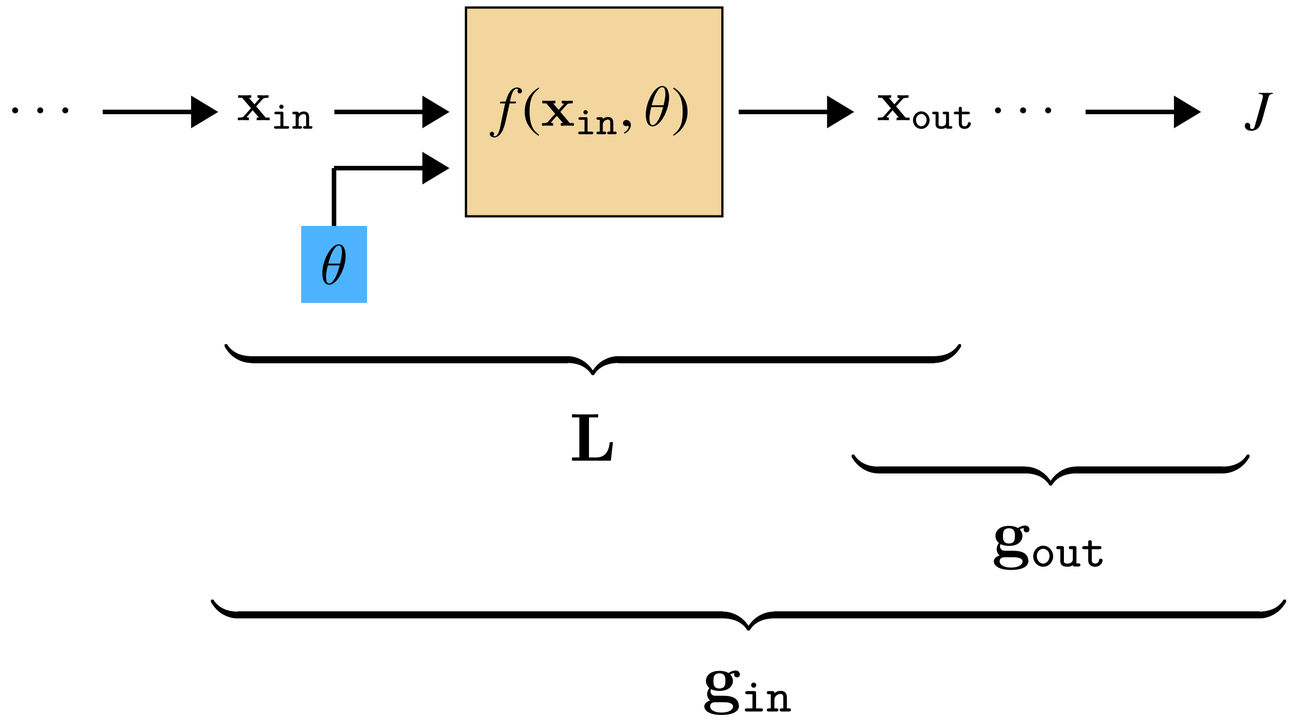
- **L**: gradient of layer outputs w.r.t. layer inputs (a matrix)
- **g**: gradient of cost w.r.t. activations (a row vector)

$$\mathbf{L} \triangleq \frac{\partial \mathbf{x}_{\text{out}}}{\partial [\mathbf{x}_{\text{in}}, \theta]} \qquad \mathbf{g} \triangleq \frac{\partial J}{\partial \mathbf{x}}$$

$$\mathbf{L}^{\mathbf{x}} \triangleq \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{x}_{\text{in}}} \qquad \mathbf{g}_{\text{out}} \triangleq \frac{\partial J}{\partial \mathbf{x}_{\text{out}}}$$

$$\mathbf{L}^{\theta} \triangleq \frac{\partial \mathbf{x}_{\text{out}}}{\partial \theta} \qquad \mathbf{g}_{\text{in}} \triangleq \frac{\partial J}{\partial \mathbf{x}_{\text{in}}}$$

Backward for a Generic Layer

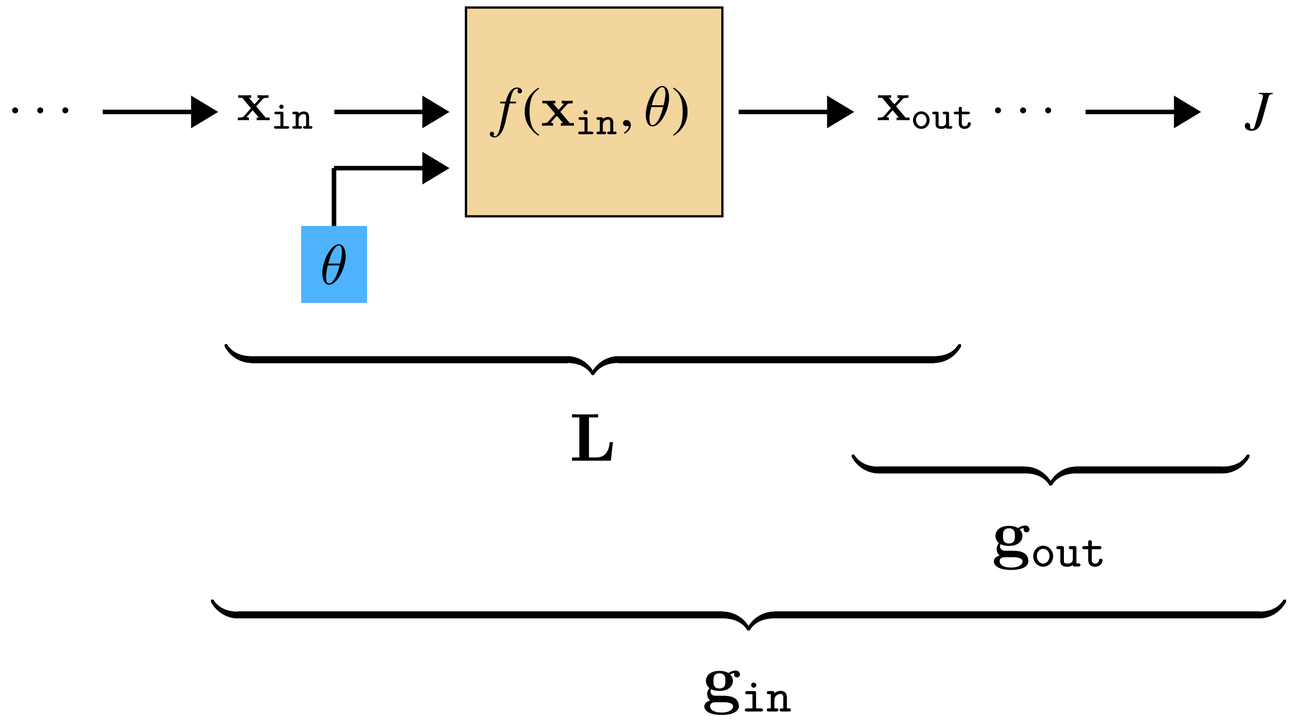


The parameter update is easy if we know **L** and **g** for a layer:

$$\frac{\partial J}{\partial \theta} = \underbrace{\frac{\partial J}{\partial \mathbf{x}_{\text{out}}}}_{\mathbf{g}_{\text{out}}} \underbrace{\frac{\partial \mathbf{x}_{\text{out}}}{\partial \theta}}_{\mathbf{L}^{\theta}} = \mathbf{g}_{\text{out}} \mathbf{L}^{\theta}$$

$$\theta^{i+1} \quad \theta^{i} - \eta \left(\frac{\partial J}{\partial \theta} \right)^{\mathsf{T}}$$

Backward for a Generic Layer



But how do we get **L** and **g** for each layer?

L comes from the derivative function, f′, of the layer (which we assume is provided):

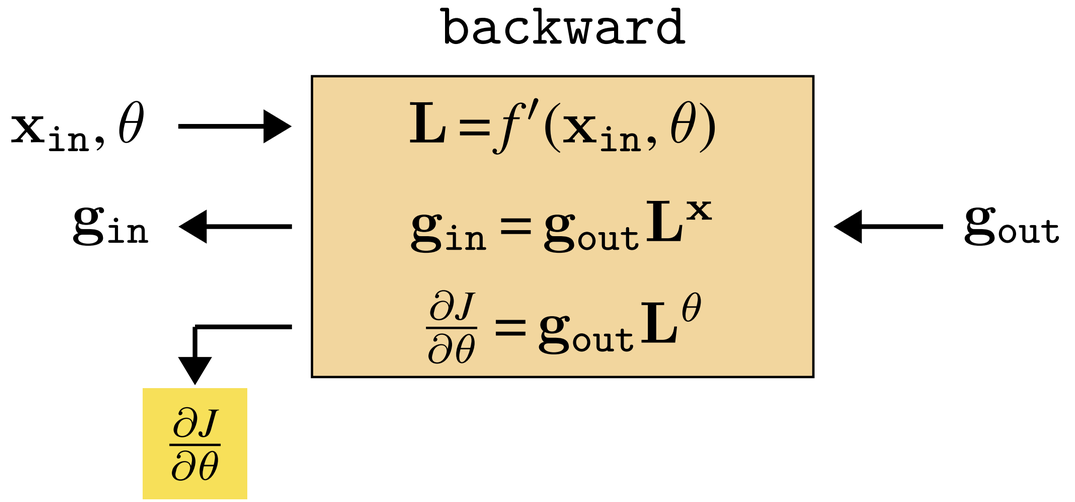
$$\mathbf{L} = f'(\mathbf{x}_{\text{in}}, \theta)$$

g can be computed iteratively via the following recurrence:

$$\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \mathbf{L}^{\mathbf{x}}$$

backpropagation of error signals

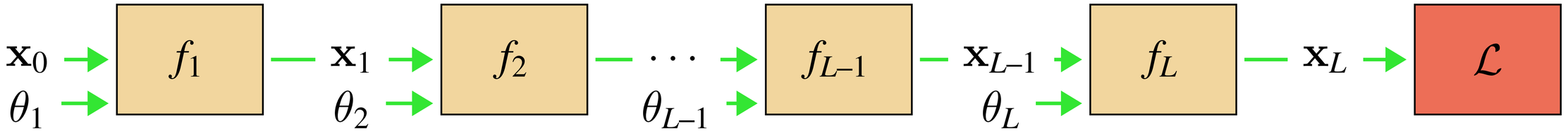
Backward for a Generic Layer



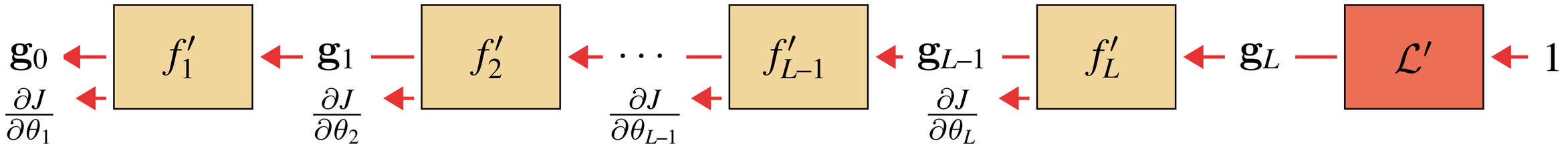
- All this machinery is to compute parameter update directions

The Full Algorithm: Forward, Then Backward

Forward:



Backward:

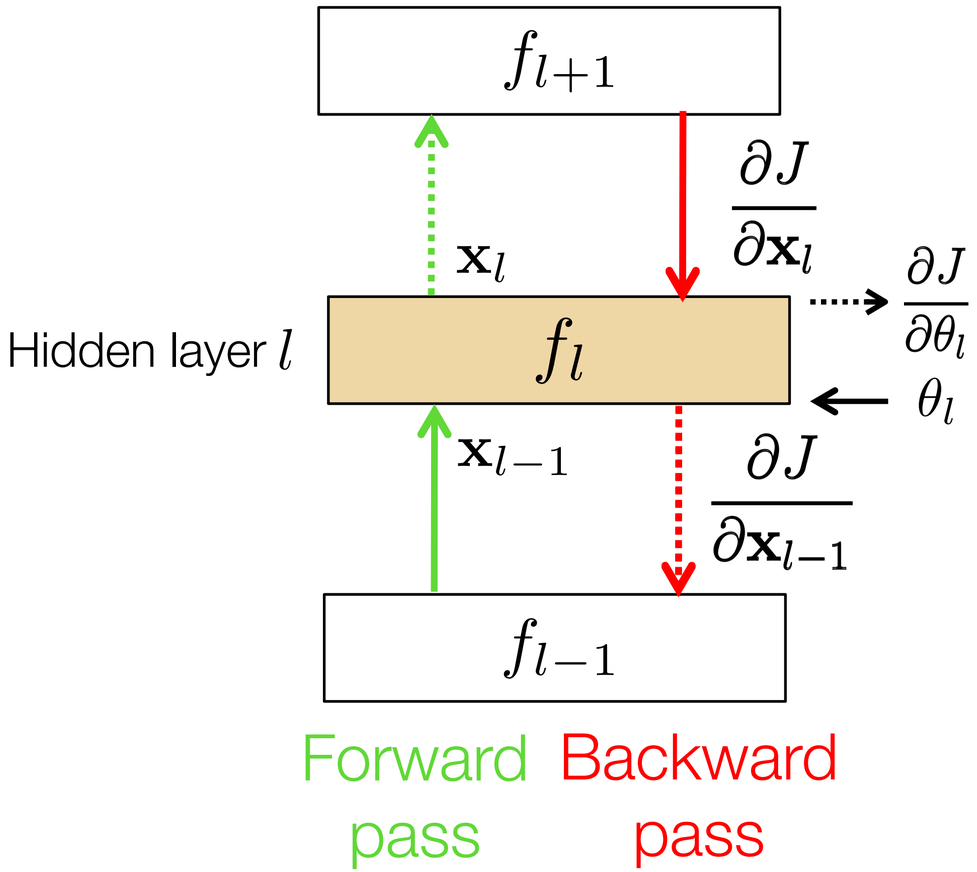


Update:

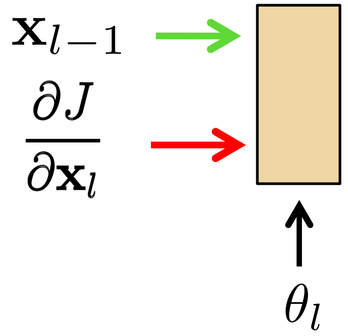
$$\theta^{i+1} \leftarrow \theta^i - \eta \left(\frac{\partial J}{\partial \theta} \right)^\mathsf{T}$$

… and repeat

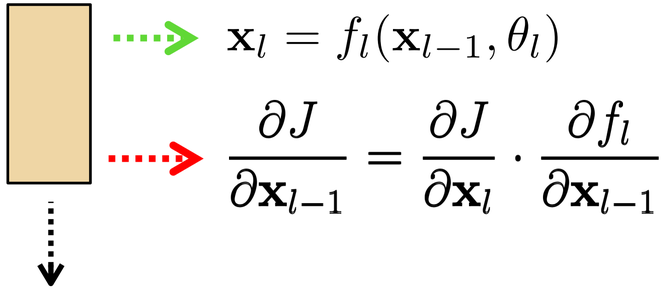
Backpropagation — Goal: to update parameters of layer l



- Layer l has three inputs (during training)



- And three outputs



- 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:

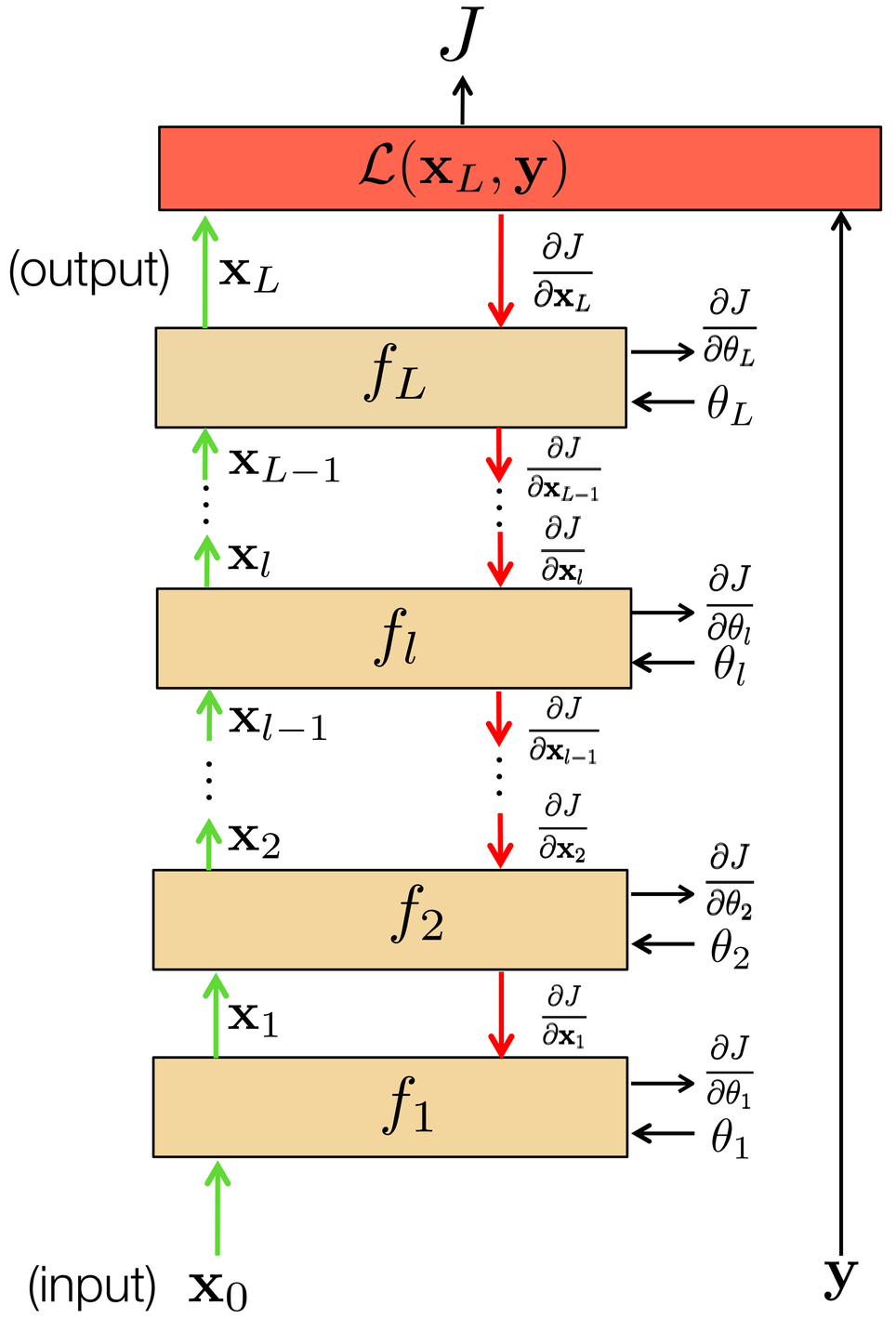
$$\mathbf{x}_l = f_l(\mathbf{x}_{l-1}, \theta_l)$$

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:

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



Backpropagation Over Data Batches

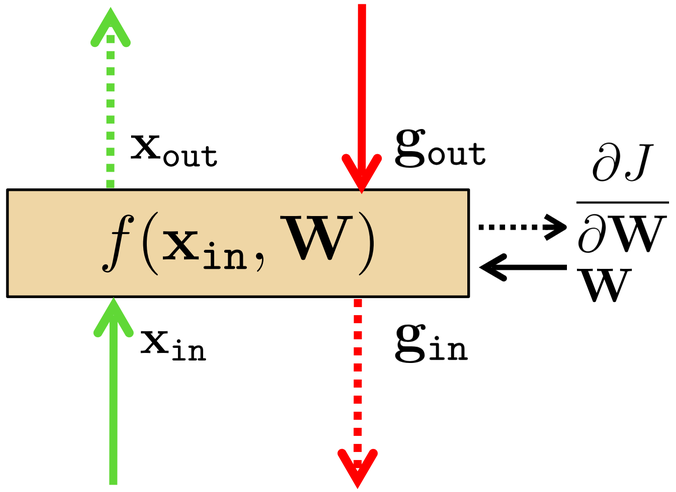
Typically we want to minimize the average cost over lots of datapoints:

$$J = \frac{1}{N} \sum_{i=1}^{N} J_i(\mathbf{x}^i, \theta)$$

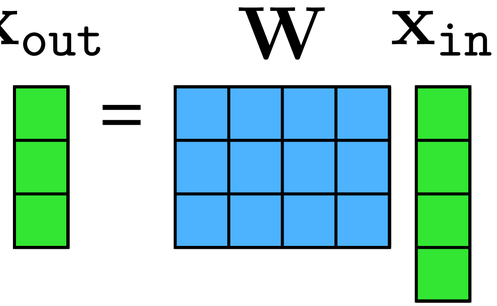
Then the gradient of the total cost is just the average of all the gradients of all the per-datapoint costs:

$$\frac{\partial J}{\partial \theta} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial J_i(\mathbf{x}^i, \theta)}{\partial \theta}$$

Linear layer



- 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:

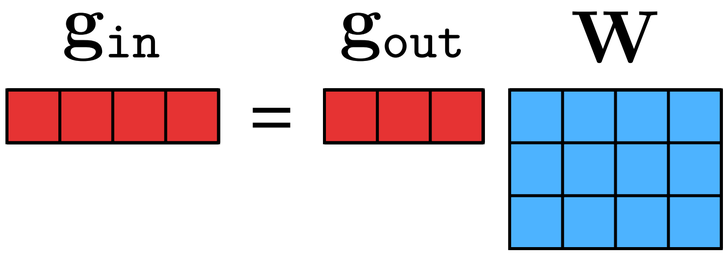
$$\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{g}_{\text{out}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial \mathbf{x}_{\text{in}}} \triangleq \mathbf{g}_{\text{out}} \cdot \mathbf{L}^{\mathbf{x}}$$

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

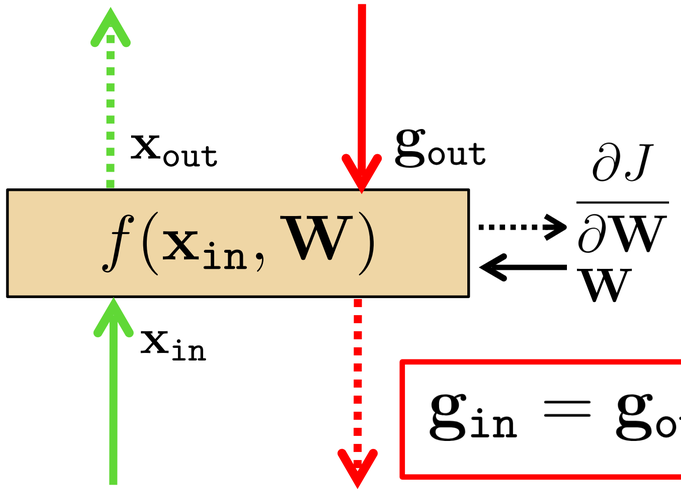
$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{x}_{\text{in}_j}} = \mathbf{W}_{ij} \longrightarrow \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{W}$$

Therefore:

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

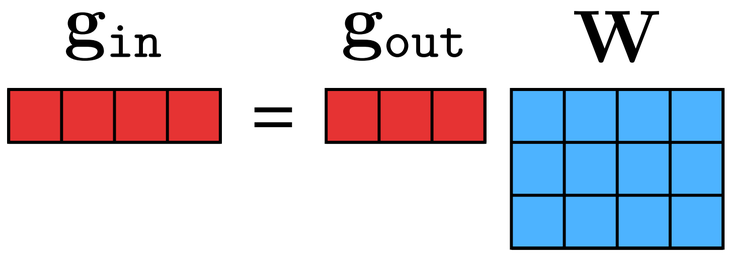


Linear layer



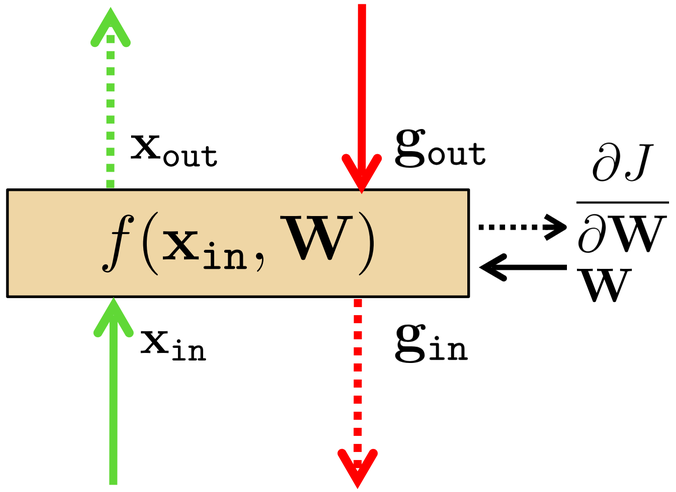
$$\mathbf{g}_{\text{in}} = \mathbf{g}_{\text{out}} \cdot \mathbf{W}$$

- 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:

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

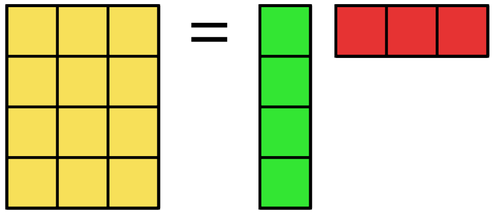
If we look at how the parameter W_{ij} changes the cost, only the i component of the output will change, therefore:

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

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

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

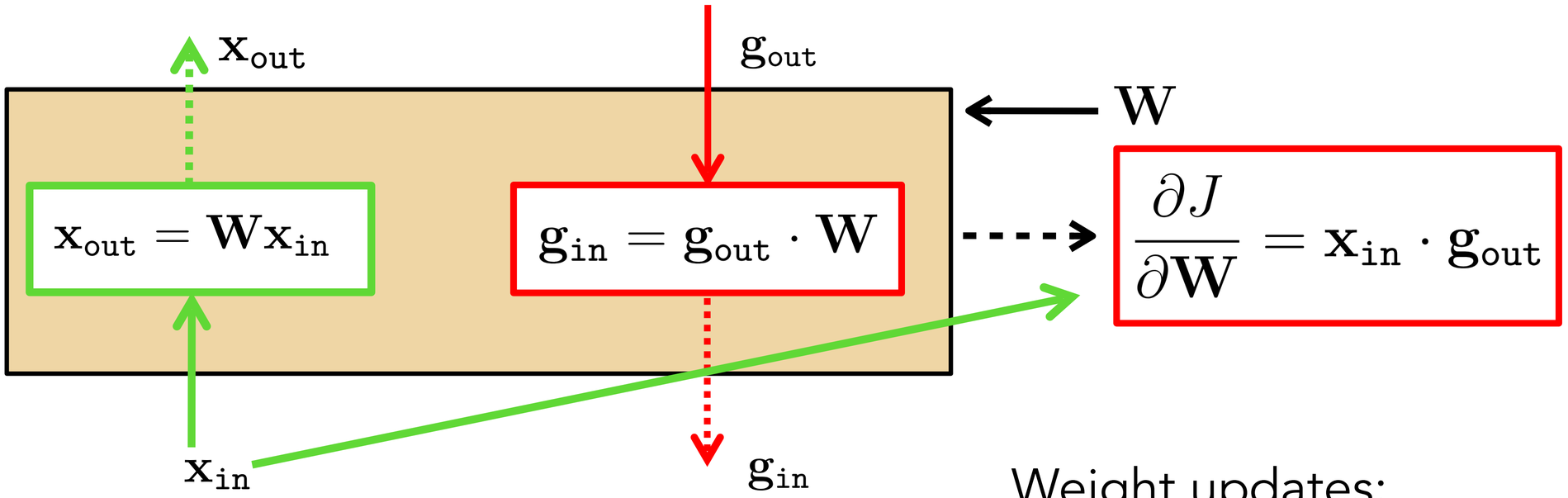
$\frac{\partial J}{\partial \mathbf{W}}$ = $\mathbf{x}_{\text{in}}$ $\mathbf{g}_{\text{out}}$



And now we can update the weights:

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

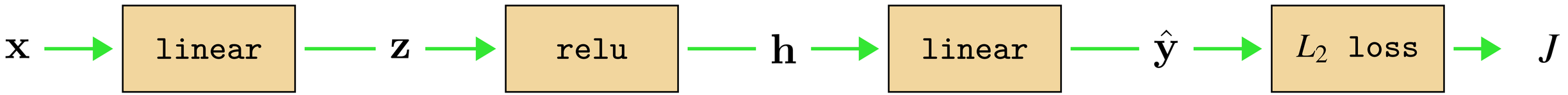
Linear layer



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



$$\mathbf{z} = \mathbf{W}_1 \mathbf{x}$$

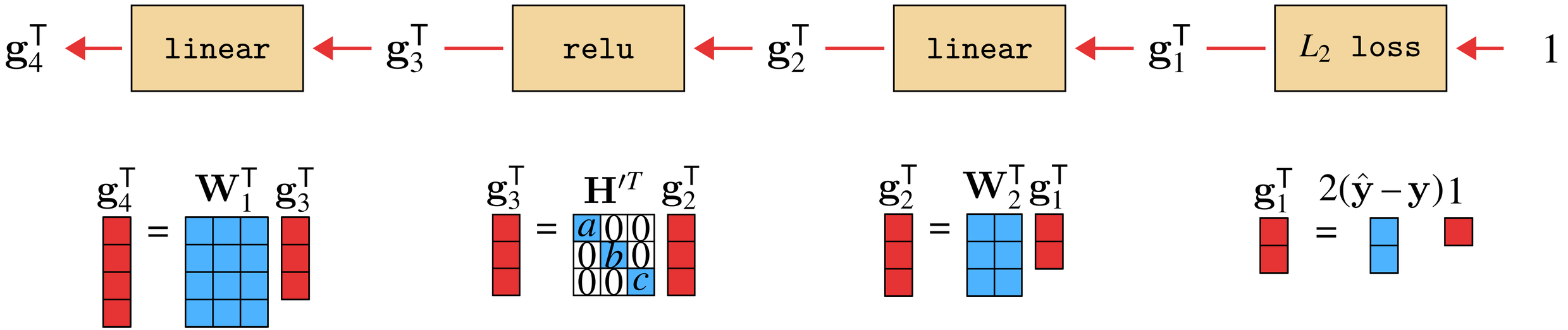
$$\mathbf{h} = \texttt{relu}(\mathbf{z})$$

$$\hat{\mathbf{y}} = \mathbf{W}_2 \mathbf{h}$$

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

Now lets look at a whole MLP: Backward

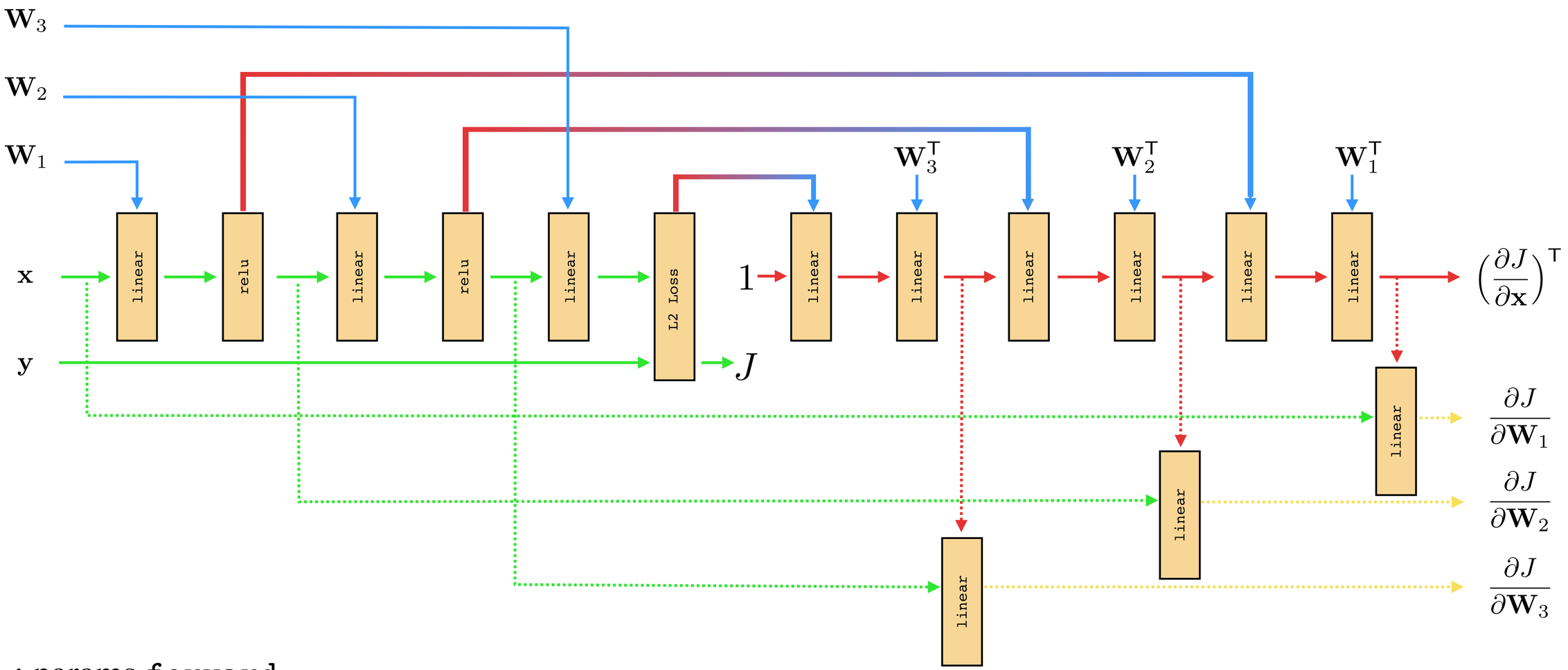
$$\mathbf{g}_{\text{in}}^{\mathsf{T}} = (\mathbf{g}_{\text{out}}\mathbf{W})^{\mathsf{T}} = \mathbf{W}^{\mathsf{T}}\mathbf{g}_{\text{out}}^{\mathsf{T}}$$



Inputs

Backpropagation (1 iteration)

Outputs



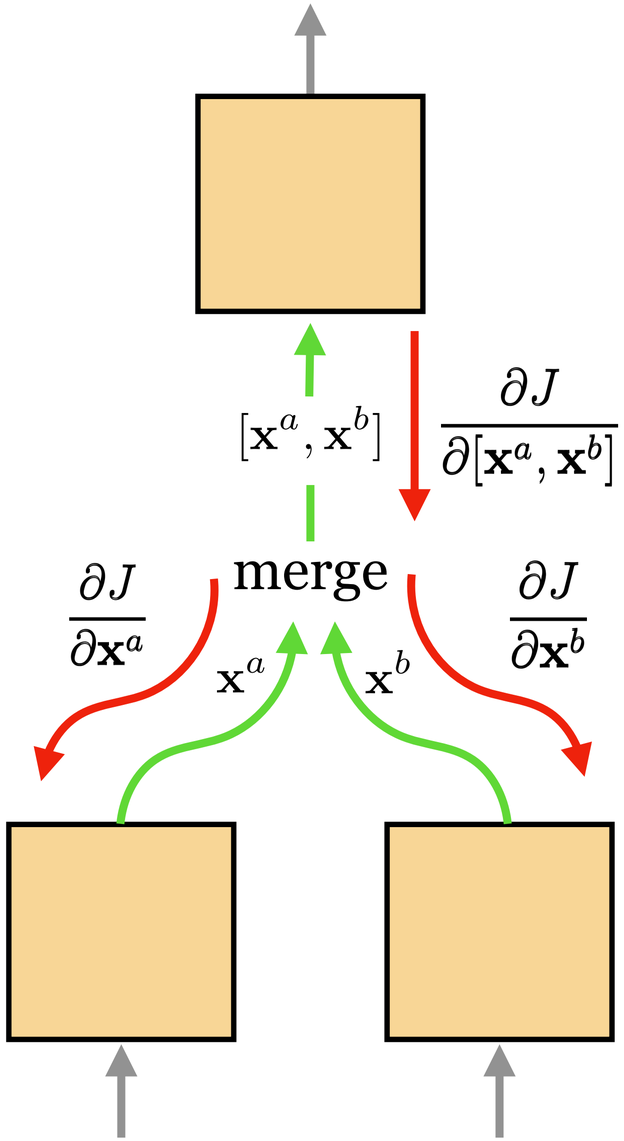
: params `forward`

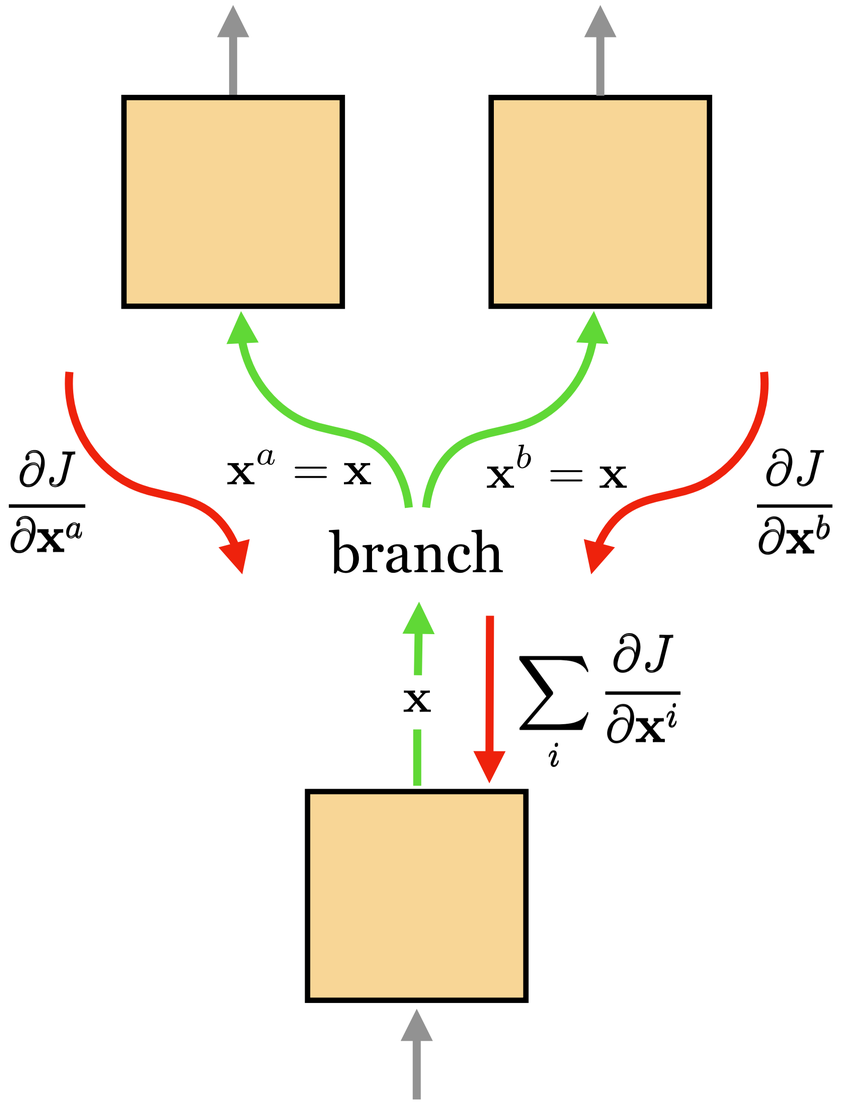
: params `backward`

: data `forward`

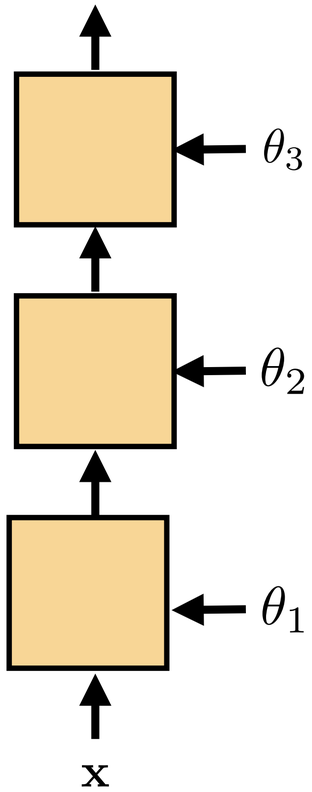
: data `backward`

DAGs

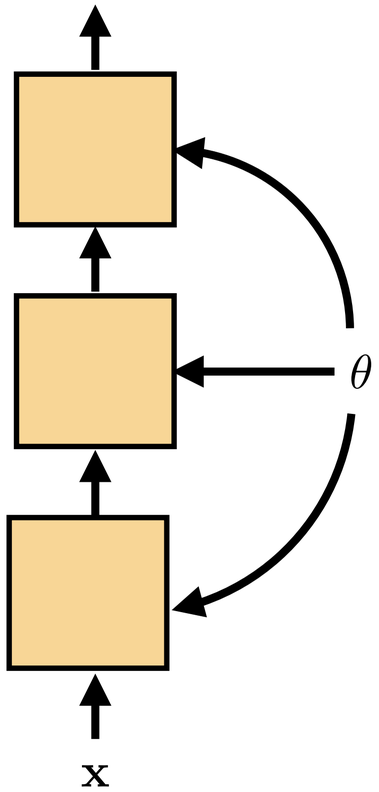

$[\mathbf{x}^a, \mathbf{x}^b]$
$\frac{\partial J}{\partial [\mathbf{x}^a, \mathbf{x}^b]}$
merge
$\frac{\partial J}{\partial \mathbf{x}^a}$
$\frac{\partial J}{\partial \mathbf{x}^b}$
$\mathbf{x}^a$
$\mathbf{x}^b$

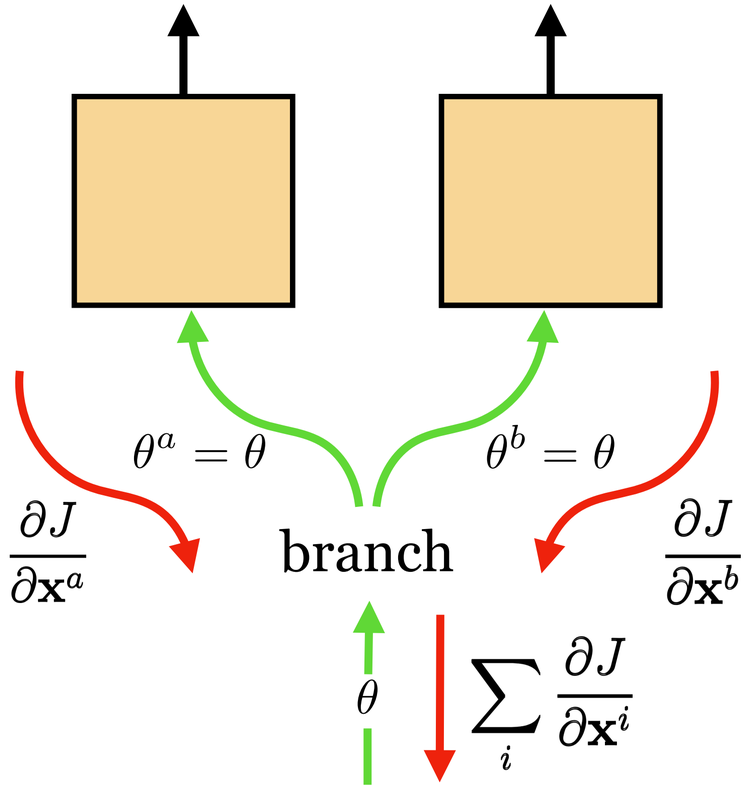

$\frac{\partial J}{\partial \mathbf{x}^a}$
$\mathbf{x}^a = \mathbf{x}$
$\mathbf{x}^b = \mathbf{x}$
$\frac{\partial J}{\partial \mathbf{x}^b}$
branch
$\mathbf{x}$
$\sum_i \frac{\partial J}{\partial \mathbf{x}^i}$

Parameter sharing



Parameter sharing

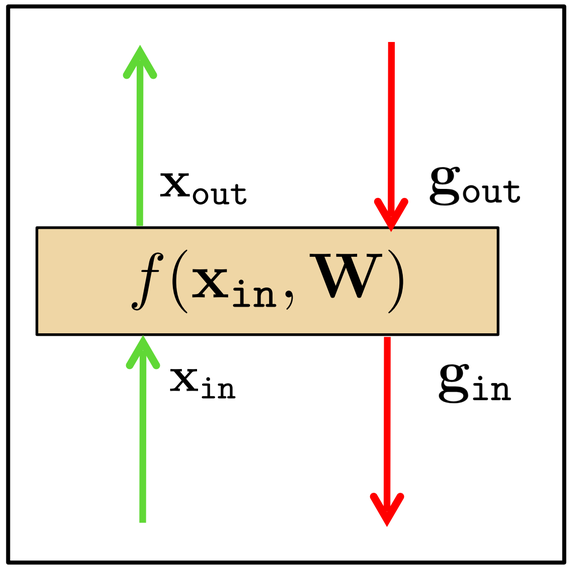




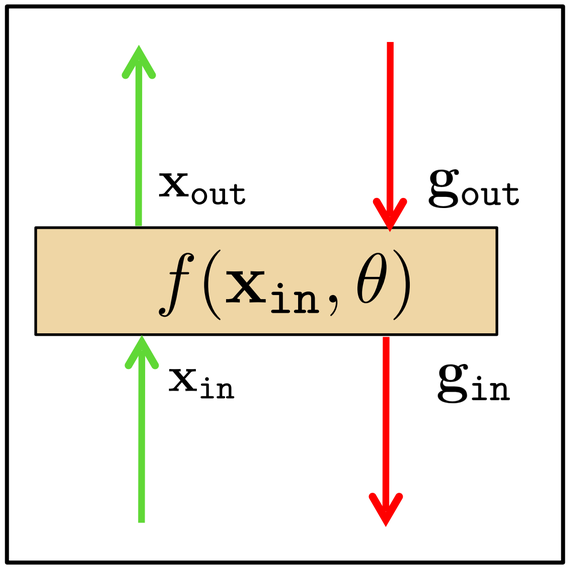
Parameter sharing —> sum gradients

Differentiable programming

Deep learning

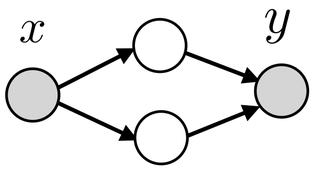


Differentiable programming



PyTorch

TensorFlow ™



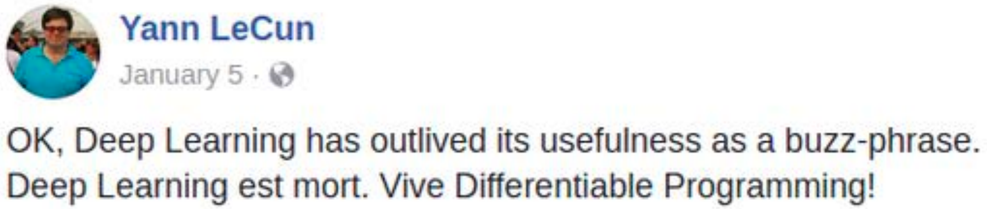

```
for i, data in enumerate(dataset):
  iter_start_time = time.time()
  if total_steps % opt.print_freq == 0:
      t_data = iter_start_time - iter_data_time
  visualizer.reset()
  total_steps += opt.batch_size
  epoch_iter += opt.batch_size
  model.set_input(data)
  model.optimize_parameters()
```

Differentiable programming

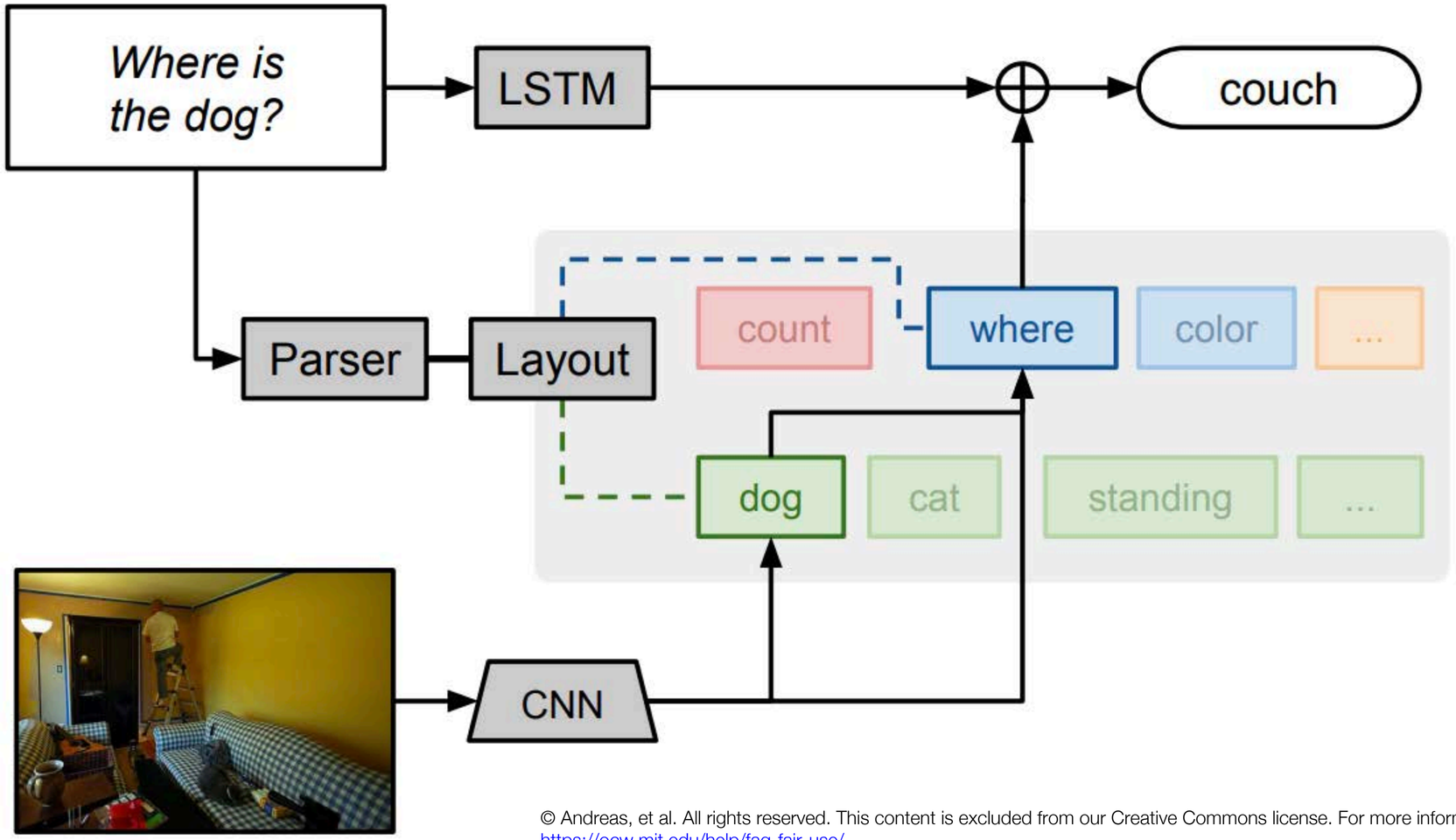
Deep nets are popular for a few reasons:

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

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



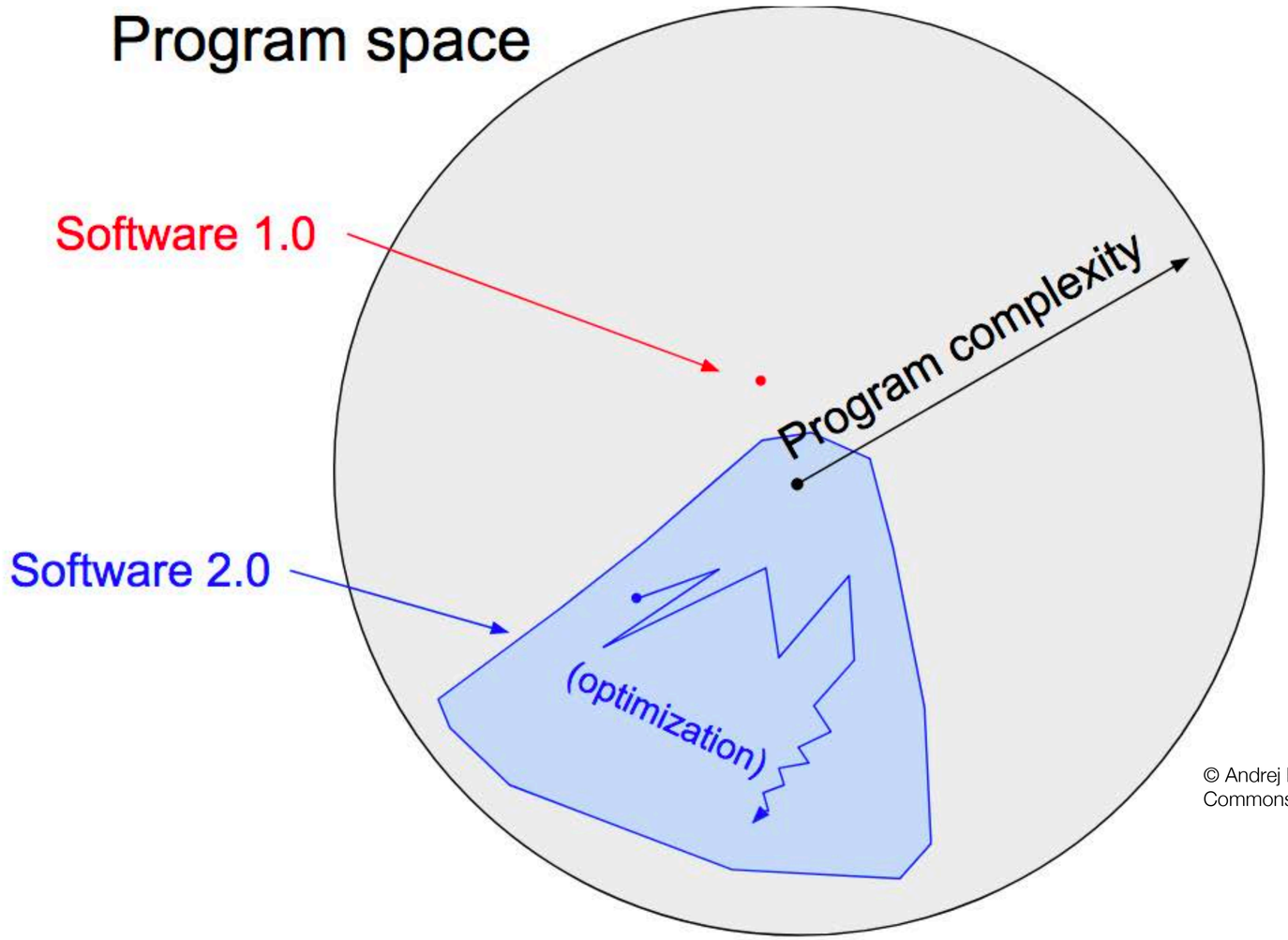
Differentiable programming

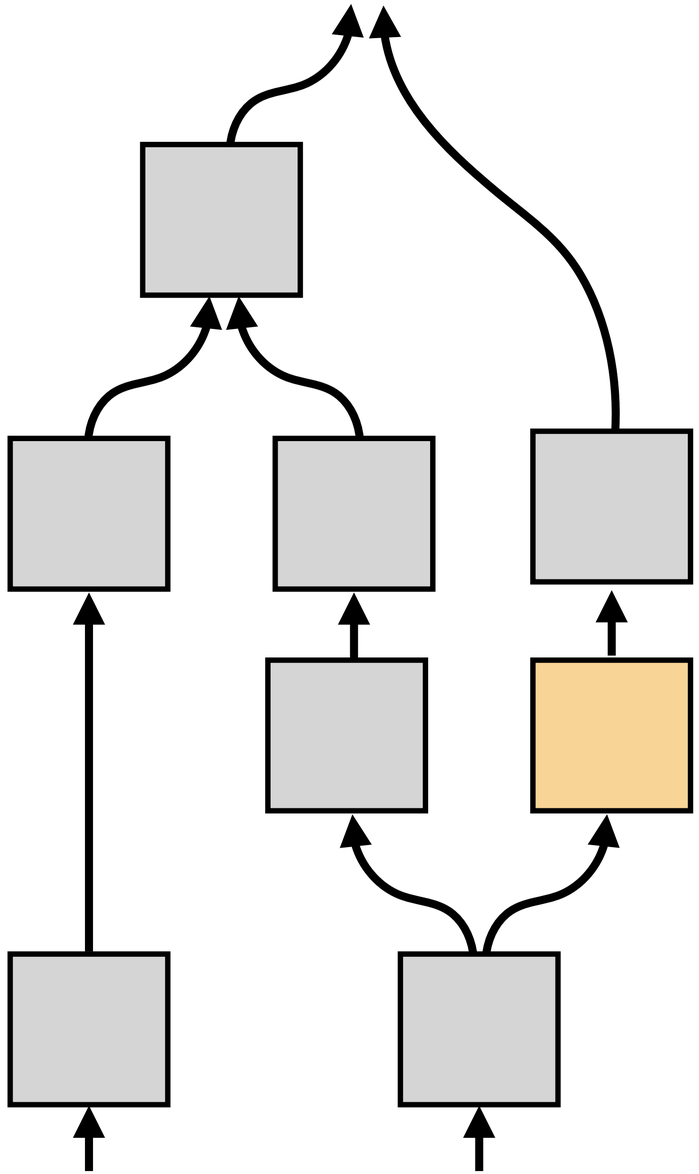


[Figure from “Neural Module Networks”, Andreas et al. 2017]

Software 2.0

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



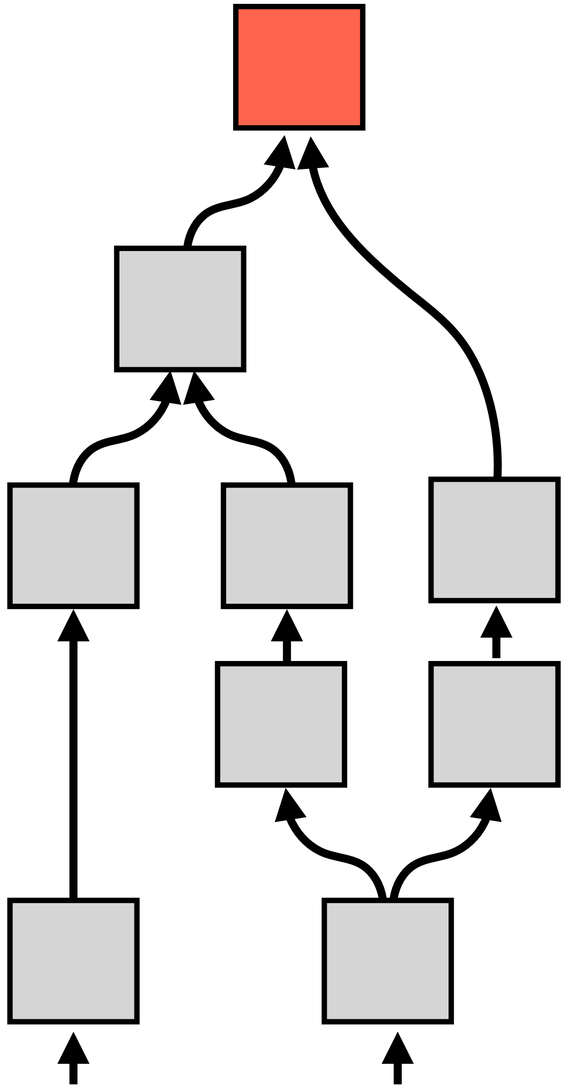


Programmed by a human

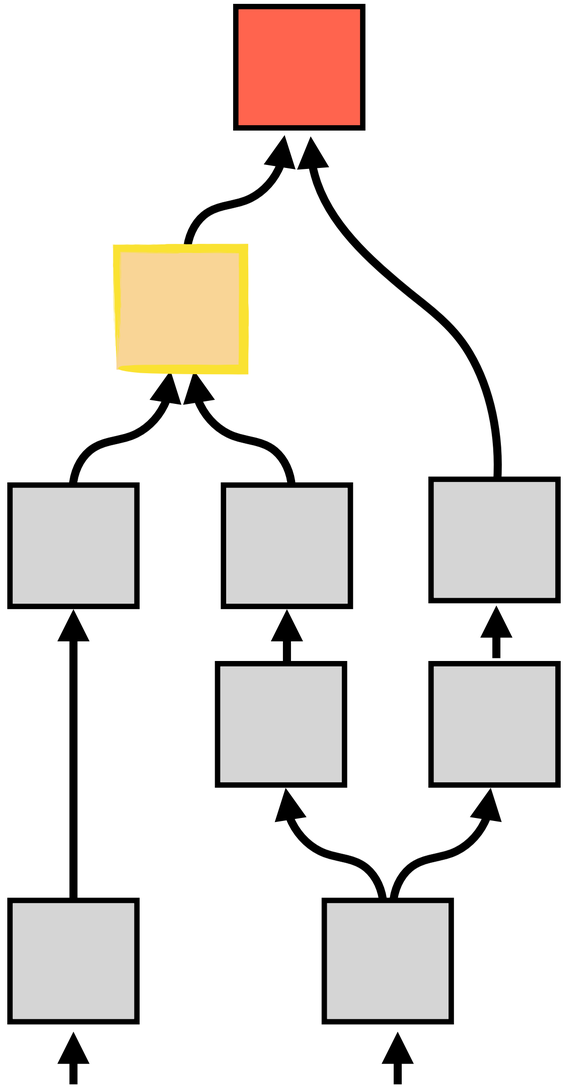
Programmed by backprop

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. to any scalar cost

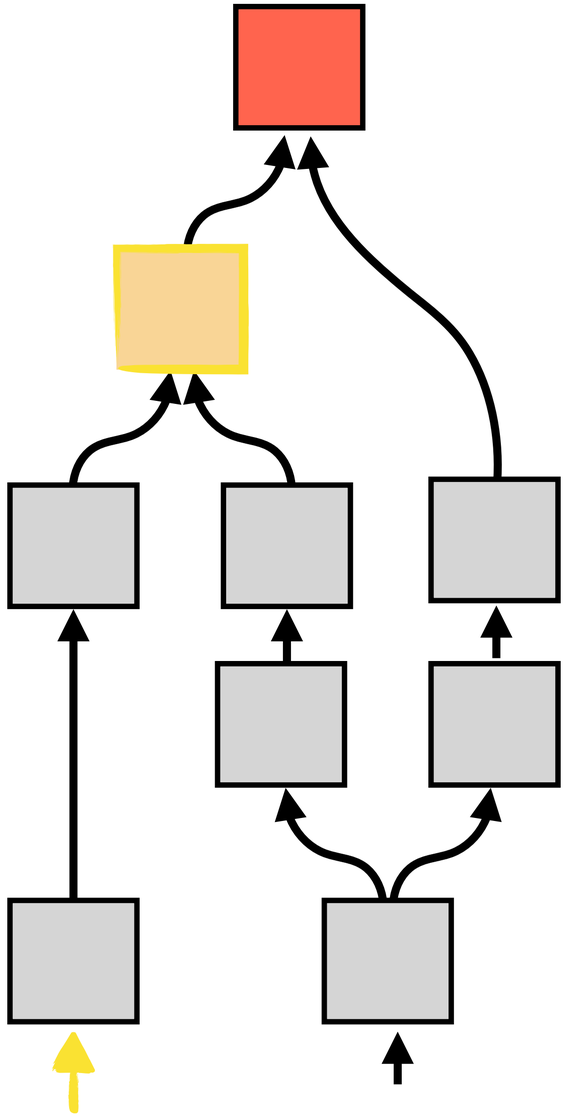


Backprop lets you optimize any node (function) or edge (variable) in your computation graph w.r.t. to any scalar cost



$\frac{\partial \blacksquare}{\partial \blacksquare}$ How the cost changes when the weights of that function (yellow) change

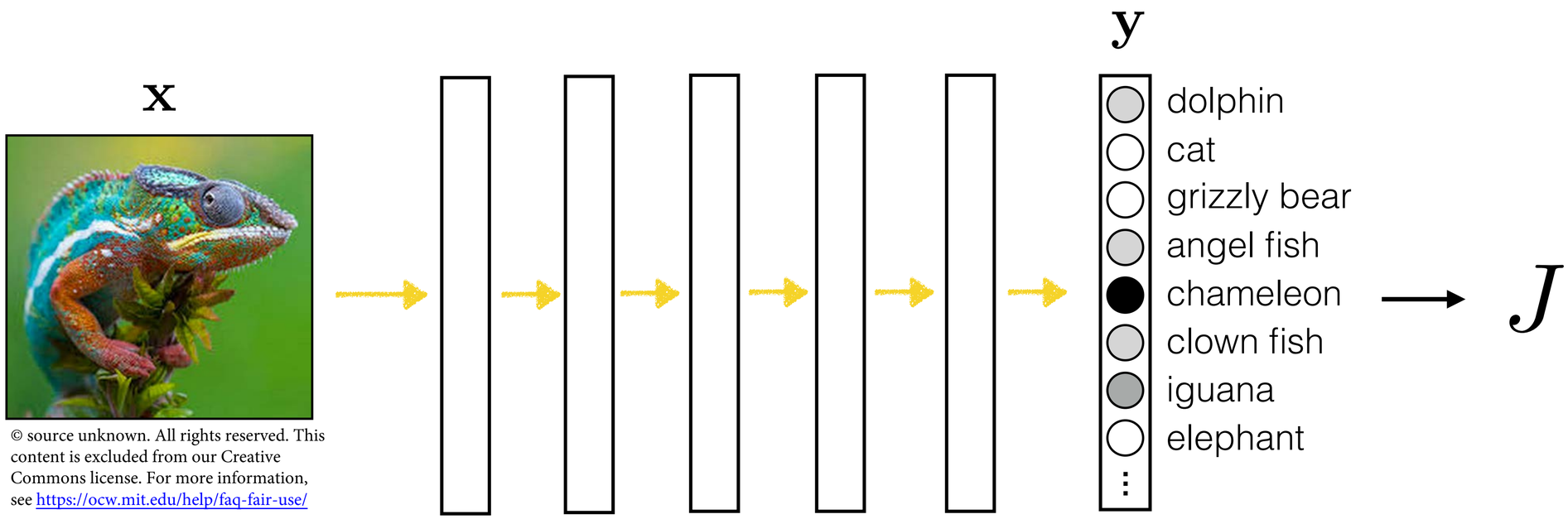
Backprop lets you optimize any node (function) or edge (variable) in your computation graph w.r.t. to any scalar cost



$\frac{\partial \blacksquare}{\partial \blacksquare}$ How the cost changes when the functional node highlighted changes

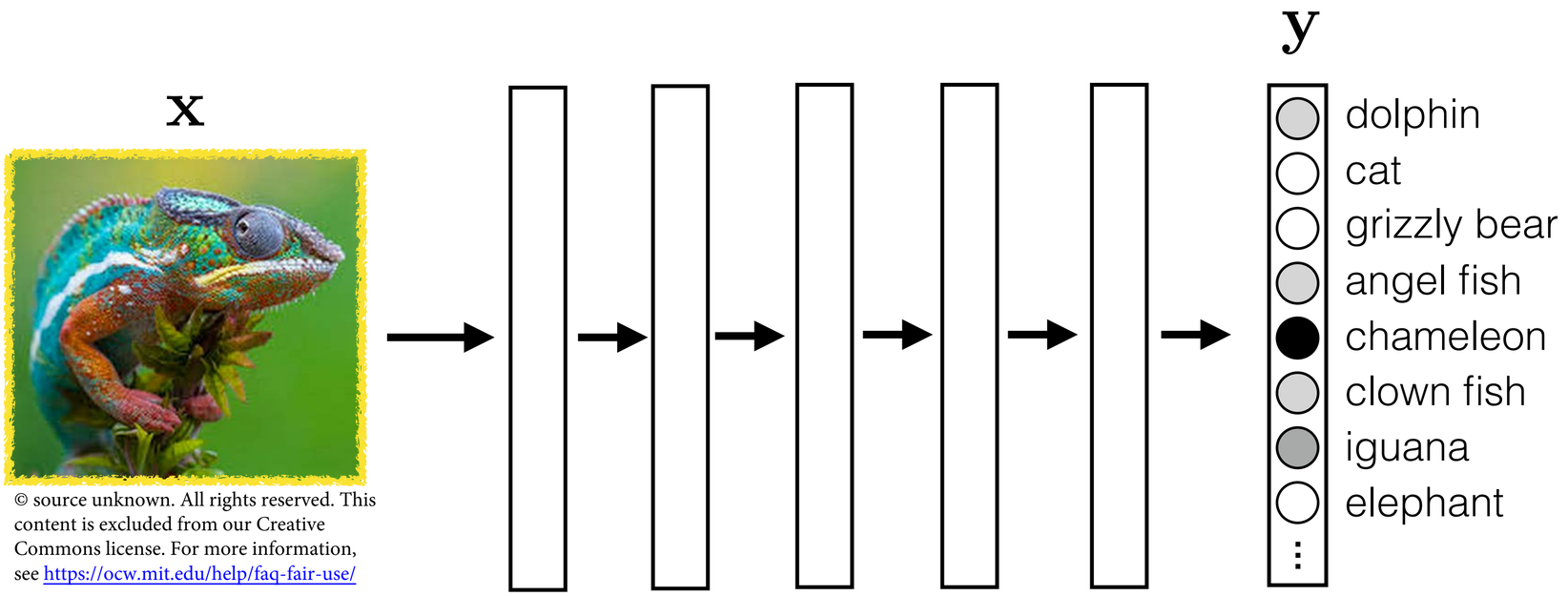
$\frac{\partial \blacksquare}{\partial \uparrow}$ How the cost changes when the input data changes

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:

Courtesy of Olah, et al. Used under CC BY.

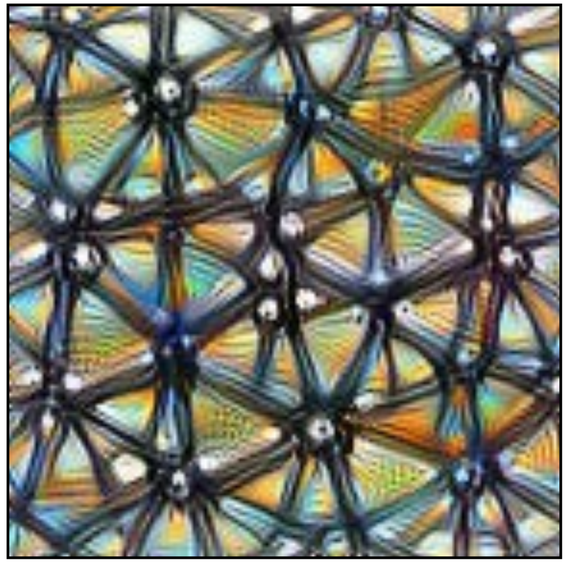
[https://distill.pub/2017/feature-visualization/]

Unit visualization

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

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

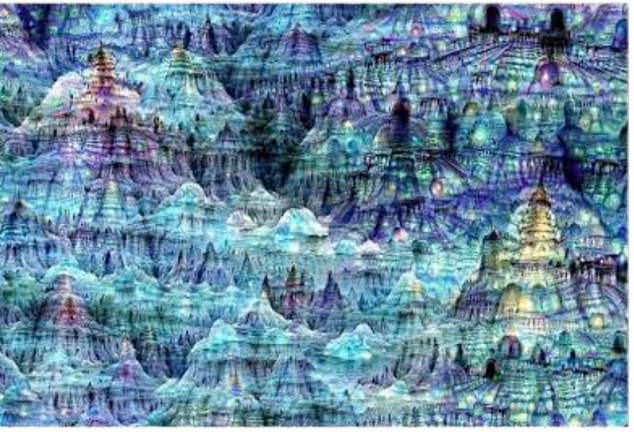
Make an image that maximizes the value of neuron j on layer l of the network:

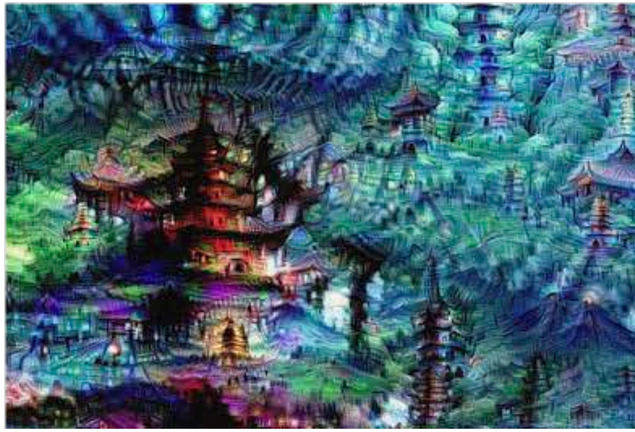


Courtesy of Olah, et al. Used under CC BY.

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



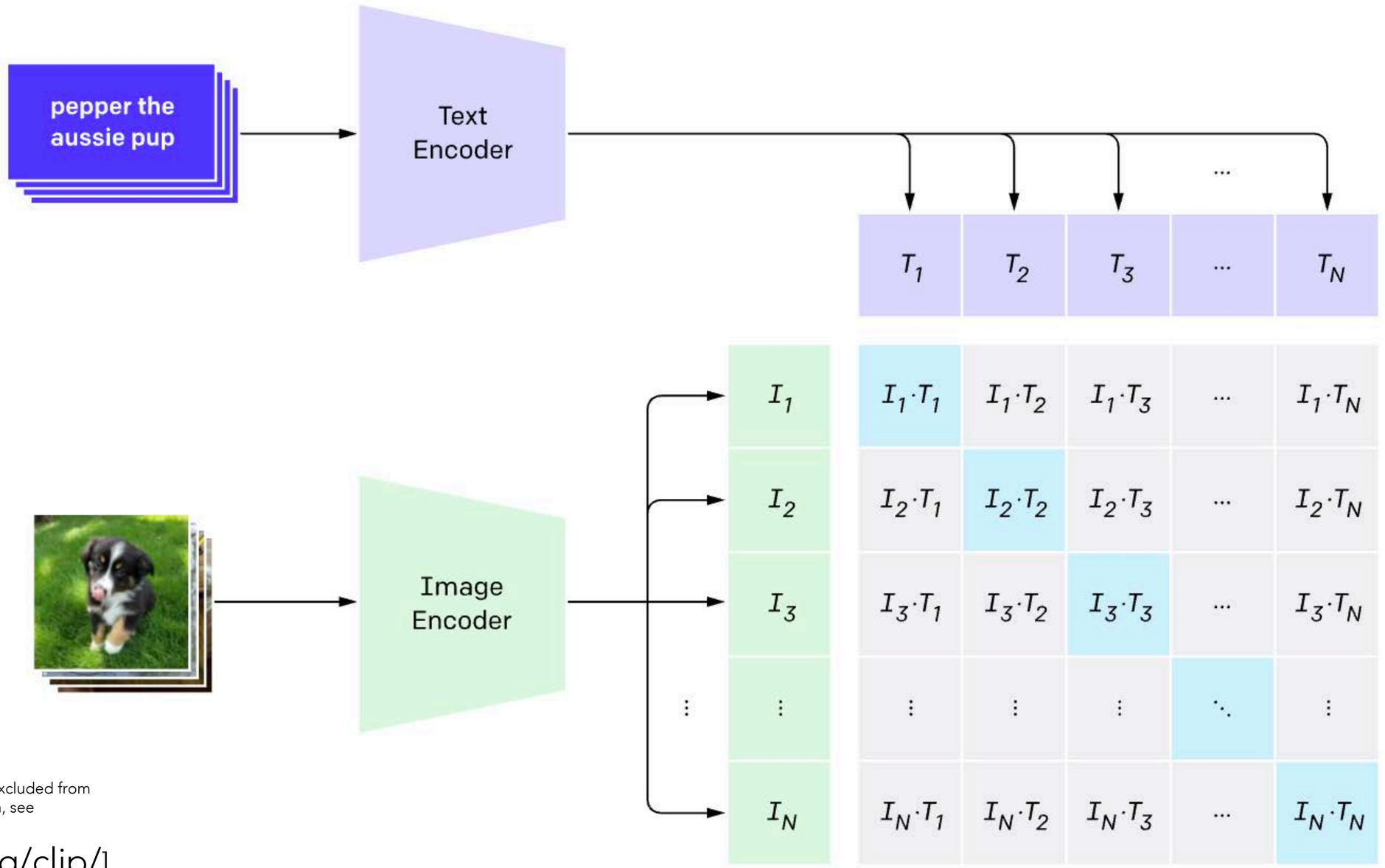


Images created using a network trained on places by MIT Computer Science and AI Laboratory.

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

CLIP

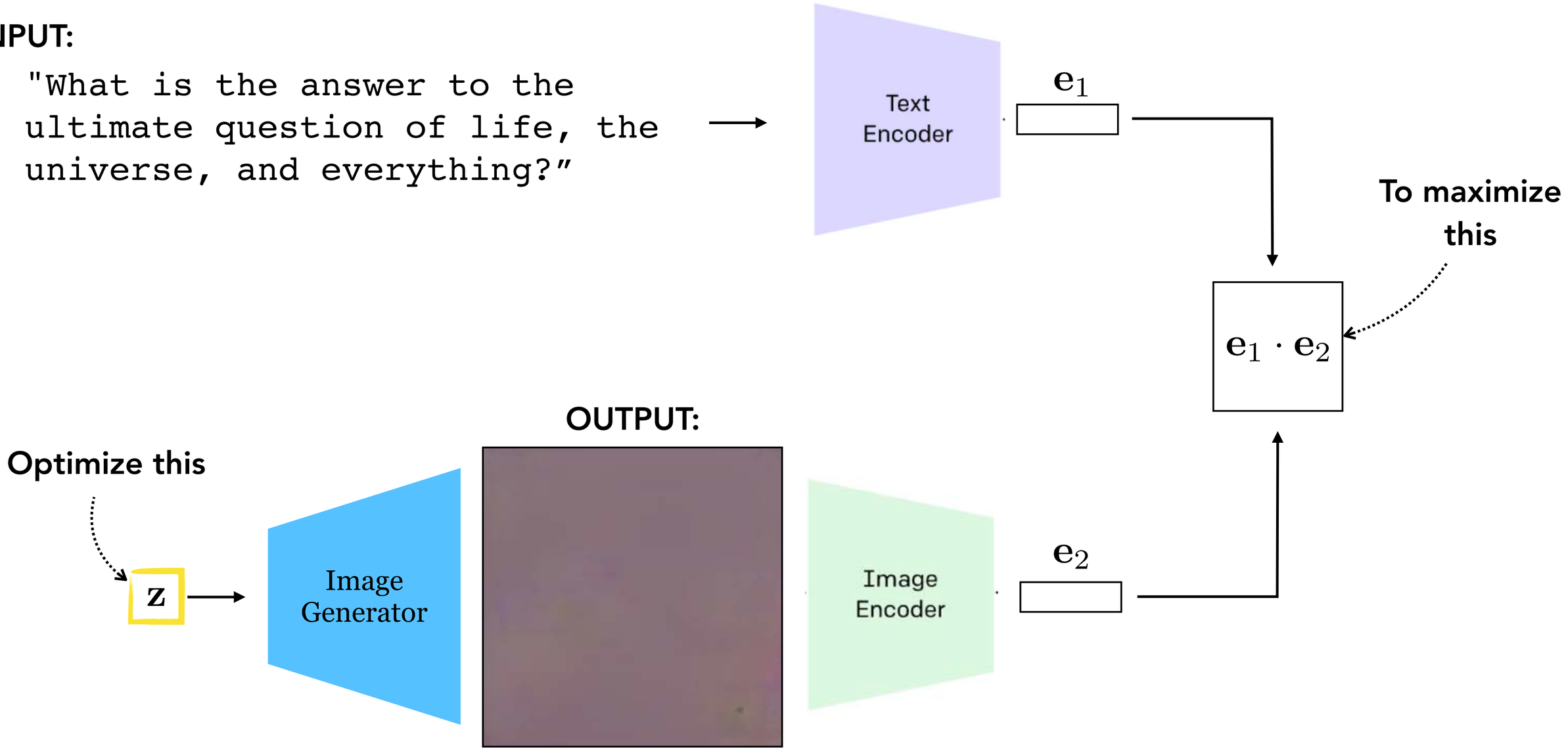


[https://openai.com/blog/clip/]

CLIP+GAN

INPUT:

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

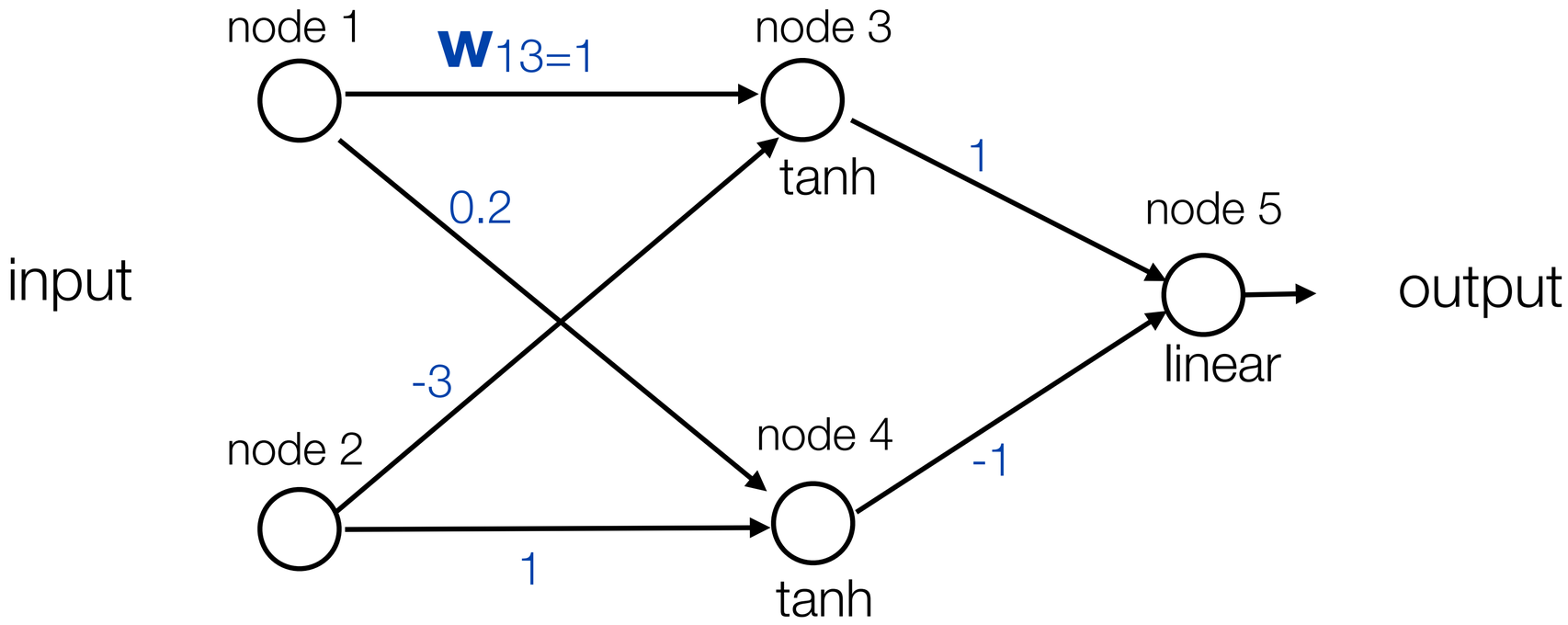


Code: https://colab.research.google.com/drive/1_4PQqzM_0KKytCzWtn-ZPi4cCa5bwK2F?usp=sharing

2. How to train a neural net

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

Backpropagation example



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

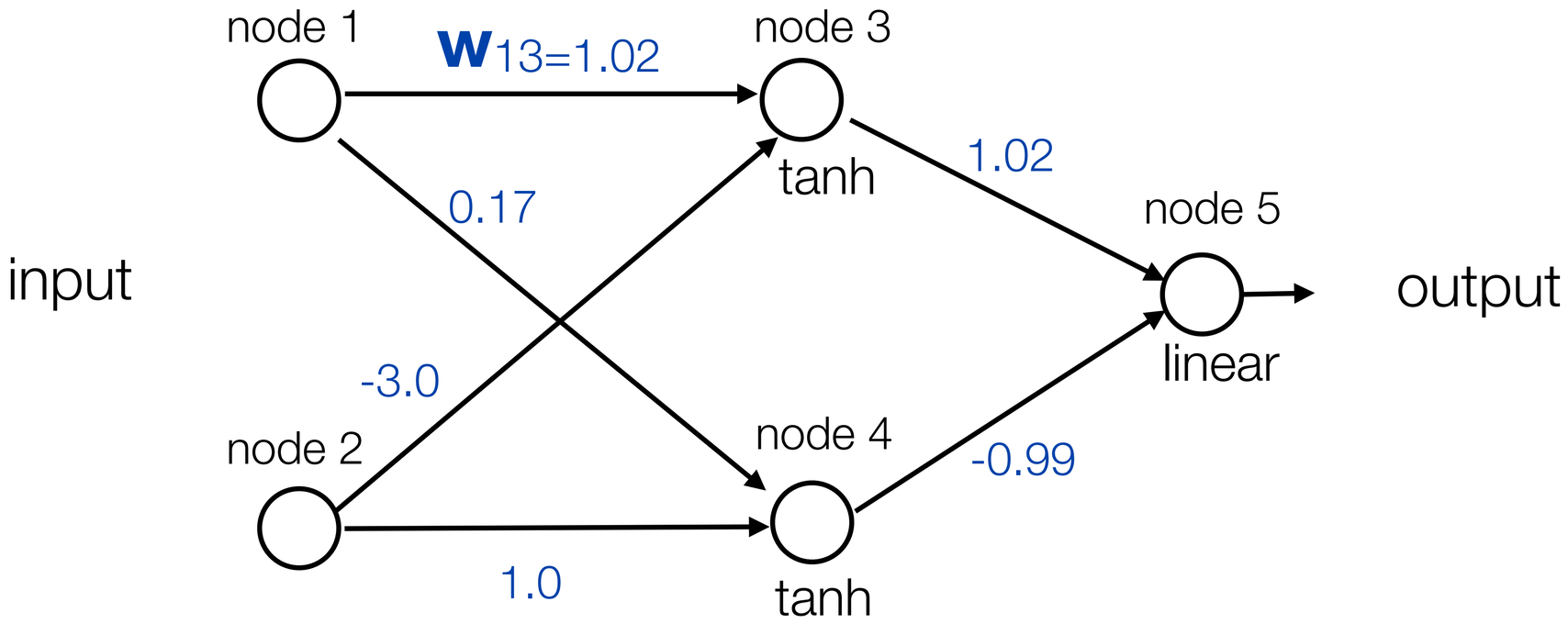
Euclidean loss

Training data: input desired output

node 1	node 2	node 5
1.0	0.1	0.5

Exercise: run one iteration of back propagation

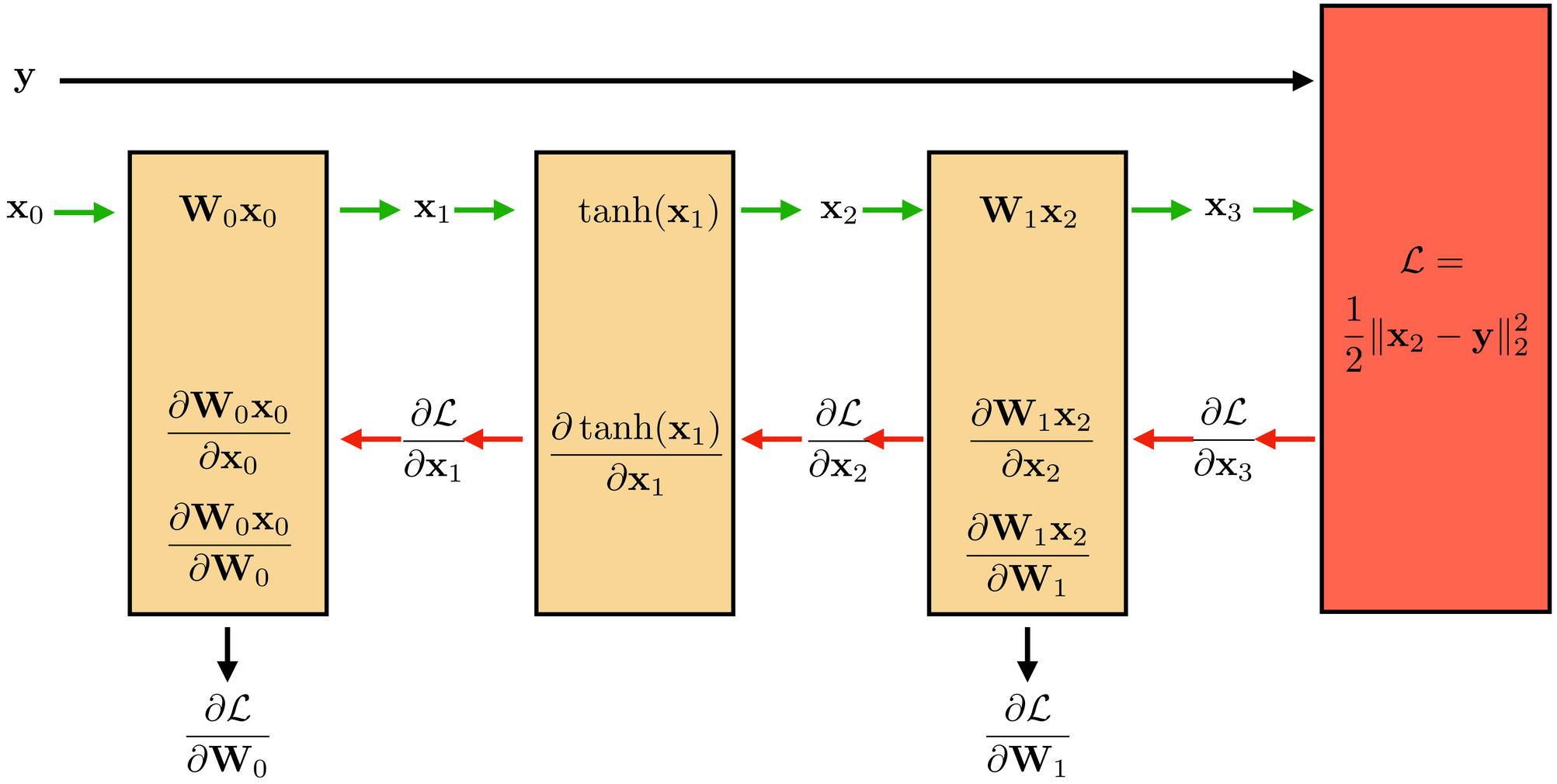
Backpropagation example



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:

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

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:

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

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

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:

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

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

6.7960 Deep Learning
Fall 2024