Training FeedForward Networks

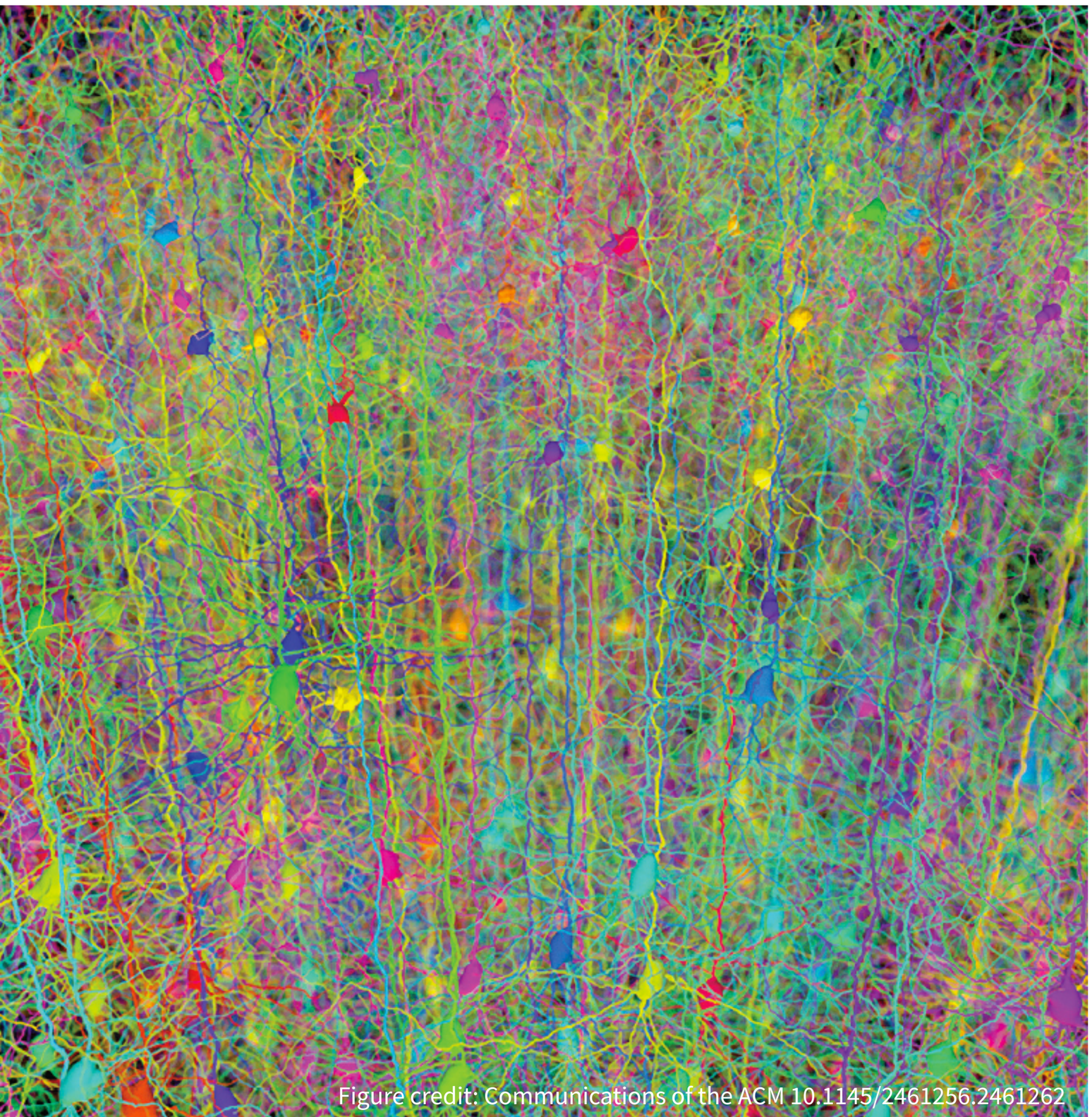


Figure credit: Communications of the ACM 10.1145/2461256.2461262

GRAHAM TAYLOR

VECTOR INSTITUTE

SCHOOL OF ENGINEERING
UNIVERSITY OF GUELPH

CANADIAN INSTITUTE
FOR ADVANCED RESEARCH

UNIVERSITY of GUELPH
CHANGING LIVES
IMPROVING LIFE

CIFAR
CANADIAN
INSTITUTE
FOR
ADVANCED
RESEARCH

Empirical Risk Minimization

A framework to design learning algorithms:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

regularized objective objective function regularizer (penalizes certain values of $\boldsymbol{\theta}$)

Learning cast as optimization:

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{m} \sum_{i=1}^{m} L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right) + \alpha\Omega(\boldsymbol{\theta})$$

Note: loss function is typically
a surrogate for what we truly should optimize

Stochastic Gradient Descent

Stochastic Gradient Descent

An algorithm that performs parameter updates after each example:

Stochastic Gradient Descent

An algorithm that performs parameter updates after each example:

- initialize: $\boldsymbol{\theta}$ $\left(\boldsymbol{\theta} \equiv \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}\right)$
- while stopping criterion not met, do:
  - for each minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with targets $y^{(i)}$
    - Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right) + \alpha \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
    - Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$

Stochastic Gradient Descent

An algorithm that performs parameter updates after each example:

- initialize: $\boldsymbol{\theta}$ $\left(\boldsymbol{\theta} \equiv \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}\right)$
- while stopping criterion not met, do:
  - for each minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with targets $y^{(i)}$
    - Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right) + \alpha \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
    - Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$

To apply this algorithm to neural network training, we need:

Stochastic Gradient Descent

An algorithm that performs parameter updates after each example:

- initialize: $\boldsymbol{\theta} \qquad \left(\boldsymbol{\theta} \equiv \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \ldots, \boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}\right)$
- while stopping criterion not met, do:
  - for each minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with targets $y^{(i)}$
    - Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right) + \alpha \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
    - Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$

To apply this algorithm to neural network training, we need:

- the loss function $L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$
- a procedure to compute parameter gradients $\nabla_{\boldsymbol{\theta}} L\left(f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$
- the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)
- an initialization method

Loss Function

Regression (Mean-squared error):

$$L_{MSE} = \frac{1}{2}\left(f(\boldsymbol{x}) - y\right)^2$$

Classification:

- Let neural network estimate $f(\boldsymbol{x})_c = p(y = c|\boldsymbol{x})$
  - We could maximize the probabilities of $y^{(i)}$ given $x^{(i)}$ in the training set
- To frame as minimization, minimize the negative log-likelihood

$$L_{CE} = -\sum_c 1_{(y=c)} \log f(\boldsymbol{x})_c = -\log f(\mathbf{x})_y$$

"cross-entropy"

  - take the log to simplify numerical stability and mathematical simplicity

Forward and Back-propagation

- The process of propagating $\boldsymbol{x}$ through a feedforward neural network to produce an output $\hat{\boldsymbol{y}}$ is called forward propagation

- During training, forward propagation continues to produce a scalar cost $J(\boldsymbol{\theta})$

- The back-propagation algorithm (backprop) allows the cost to flow backward through the net to compute the gradient

Misunderstanding?

Misunderstanding?

- Backprop is not the whole learning algorithm for neural networks
- Back-propagation only refers to the method for computing the gradient
- Another algorithm, such as stochastic gradient descent (SGD) is used to perform learning, using this gradient

Misunderstanding?

- Backprop is not the whole learning algorithm for neural networks
- Back-propagation only refers to the method for computing the gradient
- Another algorithm, such as stochastic gradient descent (SGD) is used to perform learning, using this gradient

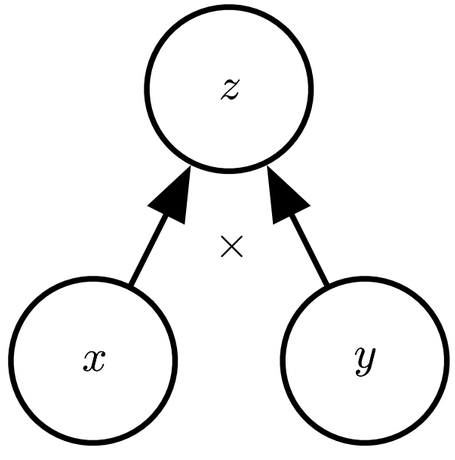
Terminology: backprop computes gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$

- for an arbitrary function f
- where $\boldsymbol{x}$ is a set of variables whose derivatives are desired
- and $\boldsymbol{y}$ are inputs to the function whose derivatives are not desired

Operations

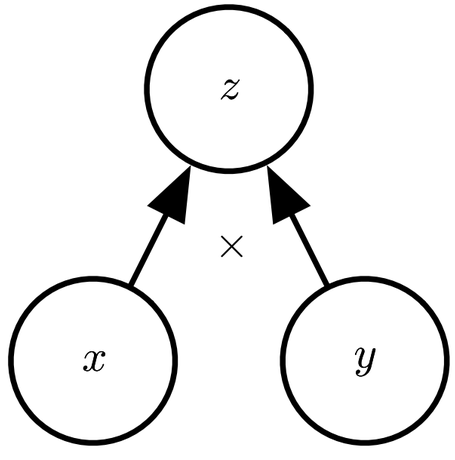
- To introduce the computational graph language, we need to introduce the idea of an operation
- An operation is a simple function of one or more variables
- A graph language is accompanied by a set of allowable operations
- More complicated functions are made by composing many operations
- Without loss of generality, we define an operation to return only a single variable (the output can have multiple entries)

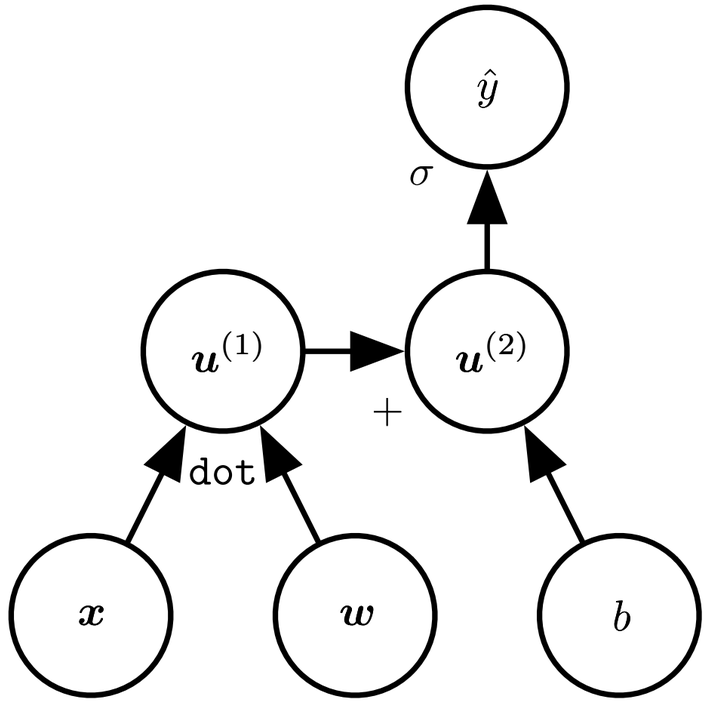
Example: Computational Graphs



Reproduced from Goodfellow et al.

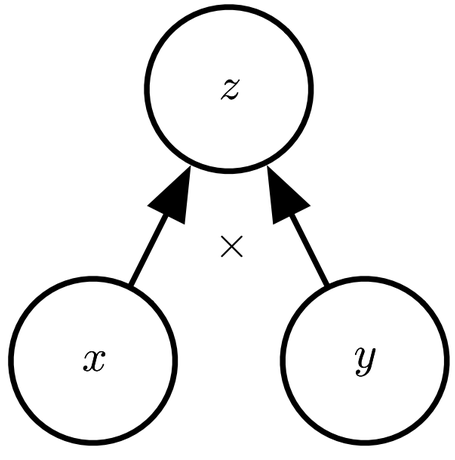
Example: Computational Graphs

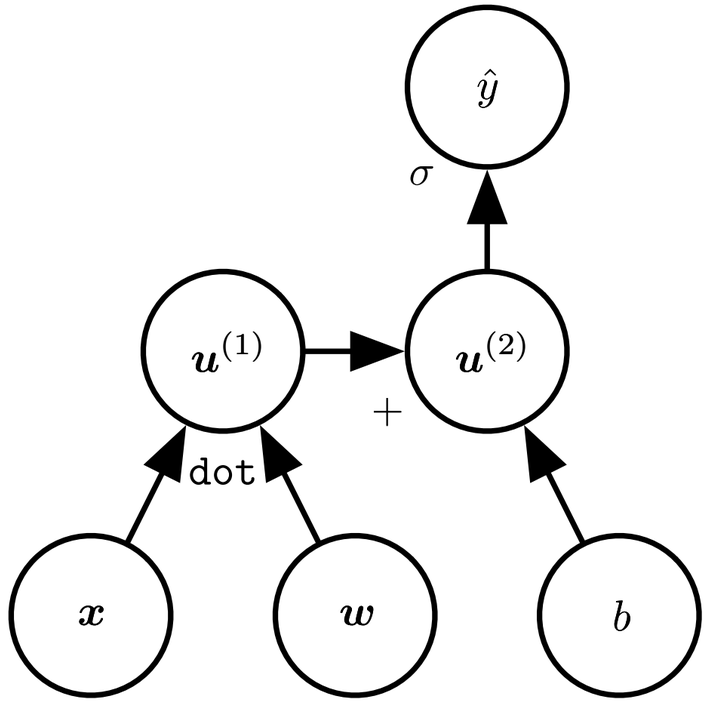


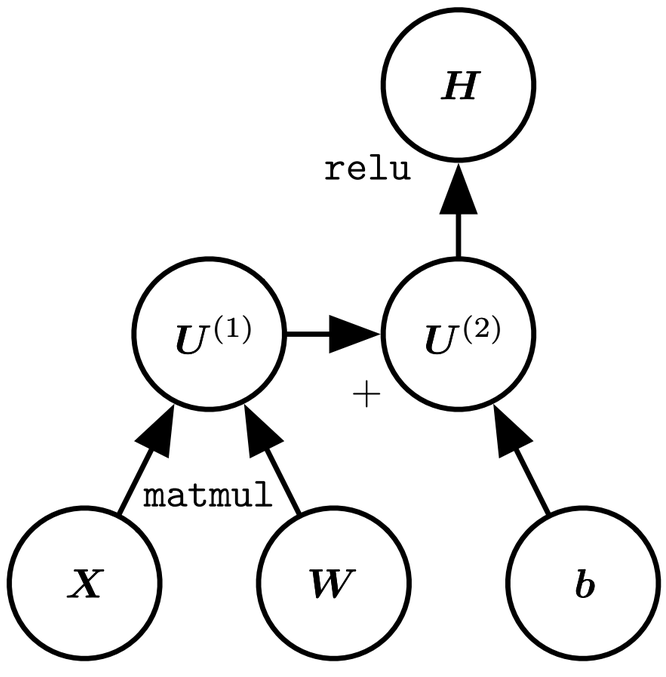


Reproduced from Goodfellow et al.

Example: Computational Graphs

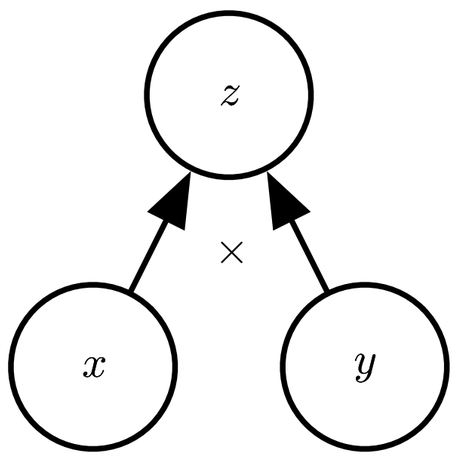


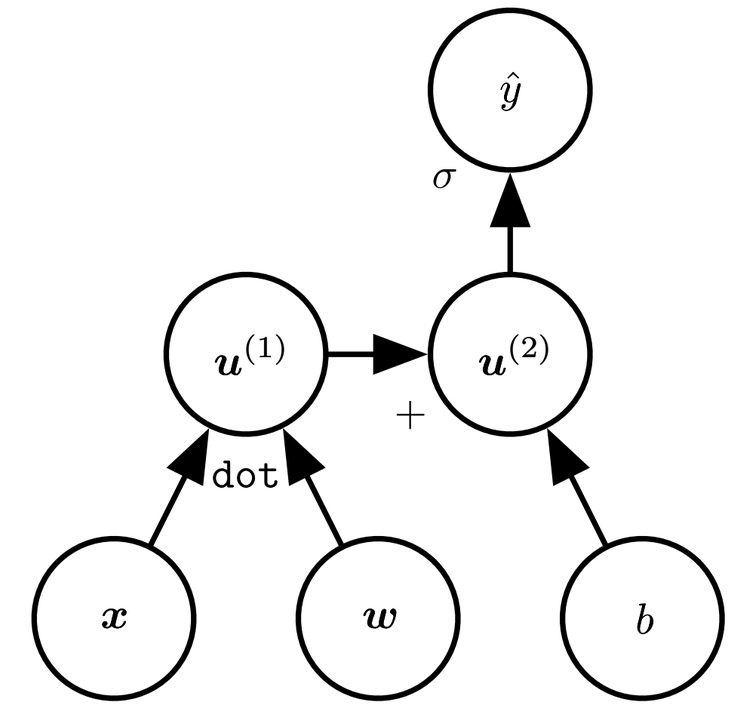


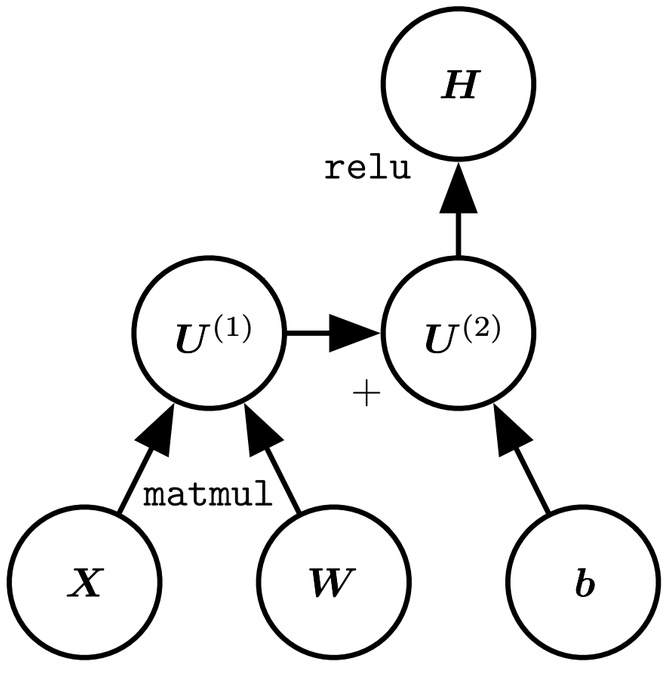


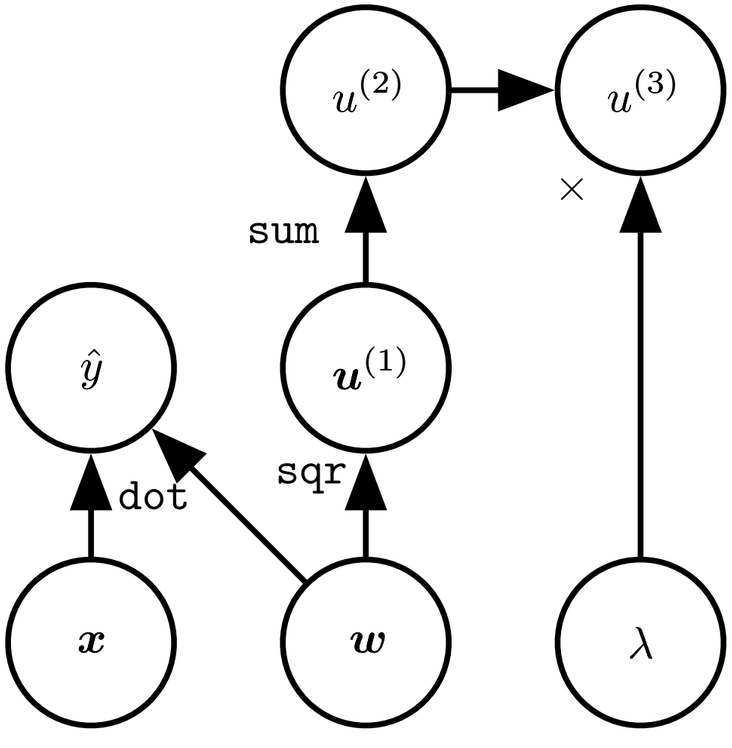
Reproduced from Goodfellow et al.

Example: Computational Graphs









Reproduced from Goodfellow et al.

Chain Rule of Calculus

Chain Rule of Calculus

Used to compute the derivatives of functions formed by composing other functions whose derivatives are known

- Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$
- Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Chain Rule of Calculus

Used to compute the derivatives of functions formed by composing other functions whose derivatives are known

- Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$
- Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

We can generalize this beyond the scalar case

- Suppose that $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

- In vector notation, this can be equivalently rewritten as:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

Chain Rule of Calculus

Used to compute the derivatives of functions formed by composing other functions whose derivatives are known

- Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$
- Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

We can generalize this beyond the scalar case

- Suppose that $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$

- In vector notation, this can be equivalently rewritten as:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

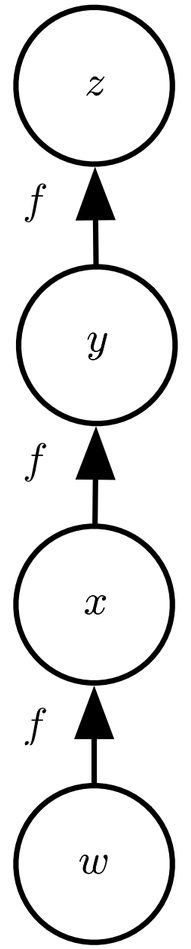
Back-propagation consists of performing one Jacobian-gradient product for each operation in the graph

Back-propagation as Recursive Chain Rule

- Using the chain rule, it is straightforward to write down an expression for the gradient of a scalar w.r.t. any node in the graph

- Actually evaluating that expression involves some extra considerations

- Specifically, many subexpressions may be repeated several times within overall expression for gradient



$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f'(y)f'(x)f'(w) = f'(f(f(w)))f'(f(w))f'(w)$$

Reproduced from Goodfellow et al.

Back-propagation in a Computational Graph

- Consider a computational graph describing how to compute a single scalar $u^{(n)}$ (e.g. the loss on a single example)

- This scalar is the quantity whose gradient we want to obtain, w.r.t. the n_i input nodes, $\{u^{(1)}, \ldots, u^{(n_i)}\}$ (e.g. the parameters of a model)

  - In other words, we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}} \forall i \in \{1, 2, \ldots, n_i\}$

- Assume that the nodes of the graph are ordered in such a way that we can compute their output one after another, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$

- Each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating:

$$u^{(i)} = f^{(i)}(\mathbb{A}^{(i)})$$

set of all nodes that are parents of $u^{(i)}$

- Back-propagation computes each derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the respective forward graph node $u^{(i)}$, via the chain rule:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

Simplified Forward Pass

(See Algorithm 6.1 in Goodfellow et al.)

Simplified Backward Pass

(See Algorithm 6.2 in Goodfellow et al.)

Forward Propagation in MLP

Example of a 2 hidden layer (or 4 layer) network …

$\boldsymbol{x}$	input
$\boldsymbol{h}^{(1)}$	1st layer hidden units
$\boldsymbol{h}^{(2)}$	2nd layer hidden units
$\hat{\boldsymbol{y}}$	output (prediction)

Forward Propagation (L1)

Forward propagation is the process of computing the output of the network given its input

$$\boldsymbol{x} \in \mathbb{R}^d \qquad \boldsymbol{W}^{(1)} \in \mathbb{R}^{n_1 \times d} \qquad \boldsymbol{b}^{(1)} \in \mathbb{R}^{n_1} \qquad \boldsymbol{h}^{(1)} \in \mathbb{R}^{n_1}$$

$$\boldsymbol{a}^{(1)} = \boldsymbol{b}^{(1)} + \boldsymbol{W}^{(1)}\boldsymbol{x}$$

$$\boldsymbol{h}^{(1)} = g(\boldsymbol{a}^{(1)})$$

$\boldsymbol{W}^{(1)}$ 1st layer weights

$\boldsymbol{b}^{(1)}$ 1st layer biases

$g(\cdot)$ activation function (e.g. ReLU)

Forward Propagation (L2)

$$\boldsymbol{h}^{(1)} \in \mathbb{R}^{n_1} \qquad \boldsymbol{W}^{(2)} \in \mathbb{R}^{n_2 \times n_1} \qquad \boldsymbol{b}^{(2)} \in \mathbb{R}^{n_2} \qquad \boldsymbol{h}^{(2)} \in \mathbb{R}^{n_2}$$

$$\boldsymbol{a}^{(2)} = \boldsymbol{b}^{(2)} + \boldsymbol{W}^{(2)} \boldsymbol{h}^{(1)}$$

$$\boldsymbol{h}^{(2)} = g(\boldsymbol{a}^{(2)})$$

$\boldsymbol{W}^{(2)}$ 2nd layer weights

$\boldsymbol{b}^{(2)}$ 2nd layer biases

Forward Propagation (Out)

$\boldsymbol{h}^{(2)} \in \mathbb{R}^{n_2}$ $\boldsymbol{W}^{(3)} \in \mathbb{R}^{n_3 \times n_2}$ $\boldsymbol{b}^{(3)} \in \mathbb{R}^{n_3}$ $\hat{\boldsymbol{y}} \in \mathbb{R}^{n_3}$

$$\boldsymbol{a}^{(3)} = \boldsymbol{b}^{(3)} + \boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)}$$

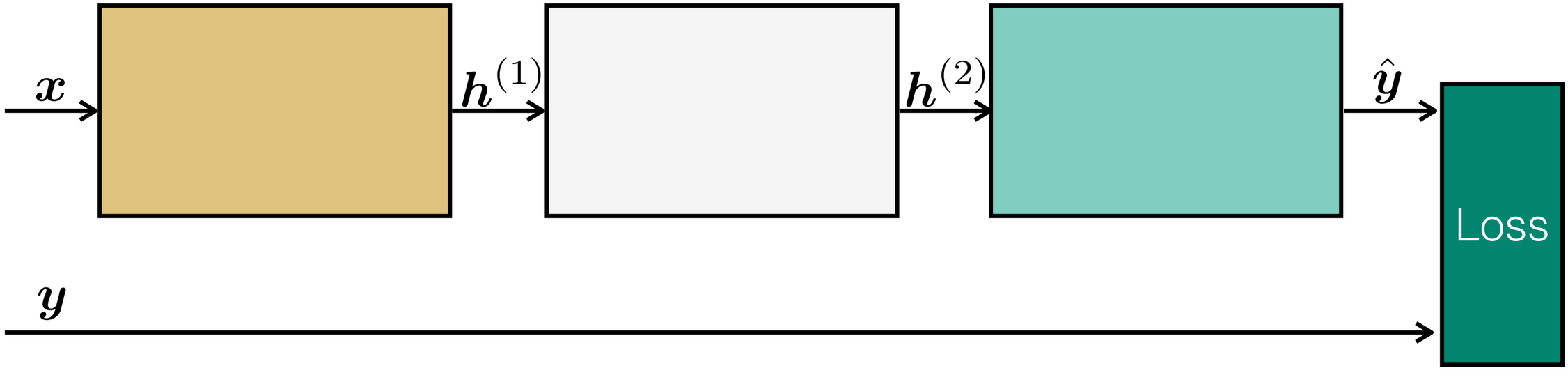
$$\hat{\boldsymbol{y}} = g(\boldsymbol{a}^{(3)})$$

$\boldsymbol{W}^{(3)}$ 3rd layer weights

$\boldsymbol{b}^{(3)}$ 3rd layer biases

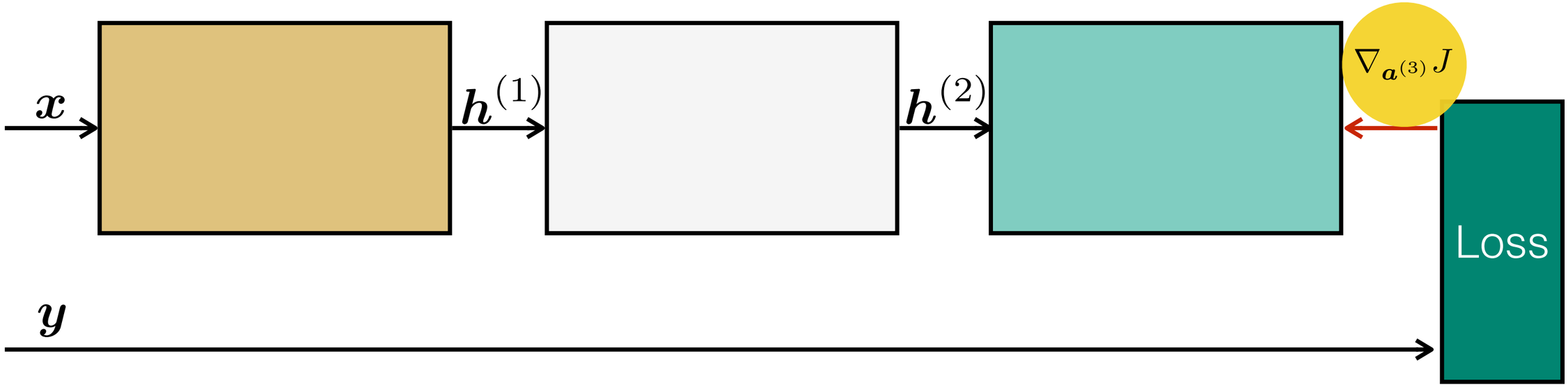
Note: $g(\cdot)$ is usually different at the output layer

How Good is a Network?



y	target (ground truth)
$\theta \equiv \{W^{(1)}, b^{(1)}, \ldots, W^{(l)}, b^{(l)}\}$	parameters
$L(\hat{y}, y)$	loss function
$J = L(\hat{y}, y) + \alpha\Omega(\theta)$	regularized cost

Backward Propagation (Out)



After the forward computation, compute the gradient on the output layer

$$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$$

Then, convert the gradient on the layer's output into a gradient on the pre-activation:

$$g \leftarrow \nabla_{a^{(3)}} J$$

Derivative w.r.t. Input of Softmax

First convert the discrepancy between each output and its target value into an error derivative

$$\hat{y}_k = p(c_k = 1|\boldsymbol{x}) = \frac{e^{a_k}}{\sum_{j=1}^{C} e^{a_j}}$$

Note: layer superscript has been left off for simplicity, i.e. $a_j = (\boldsymbol{a}^{(3)})_j$

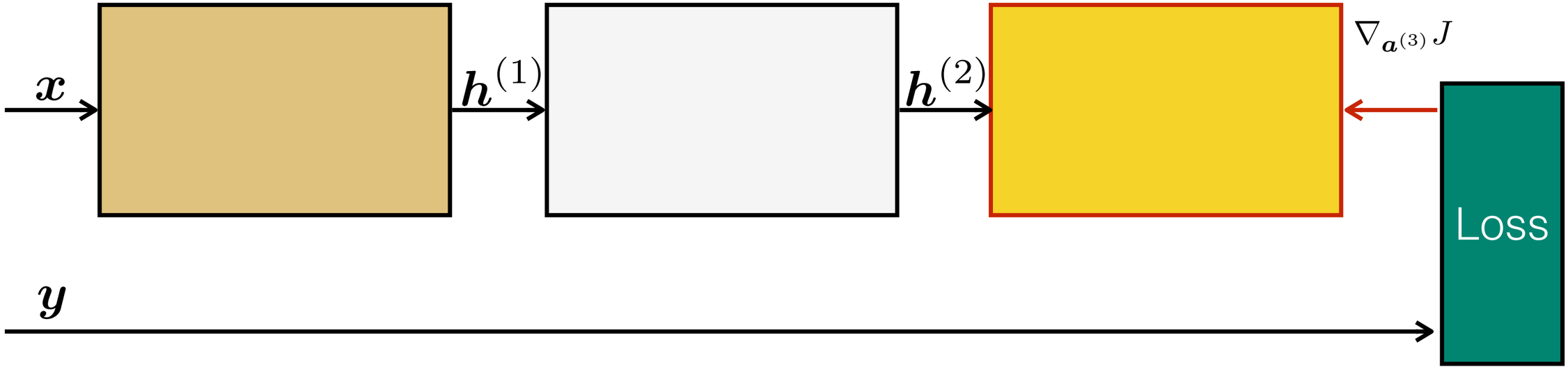
$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_j y_j \log p(c_j|\boldsymbol{x})$$

$$\boldsymbol{y} = [00 \ldots 010 \ldots 0]$$

By substituting the first formula into the second, and taking the derivative w.r.t. a we get:

$$\frac{\partial L}{\partial a_j} = p(c_j|\boldsymbol{x}) - y_j$$

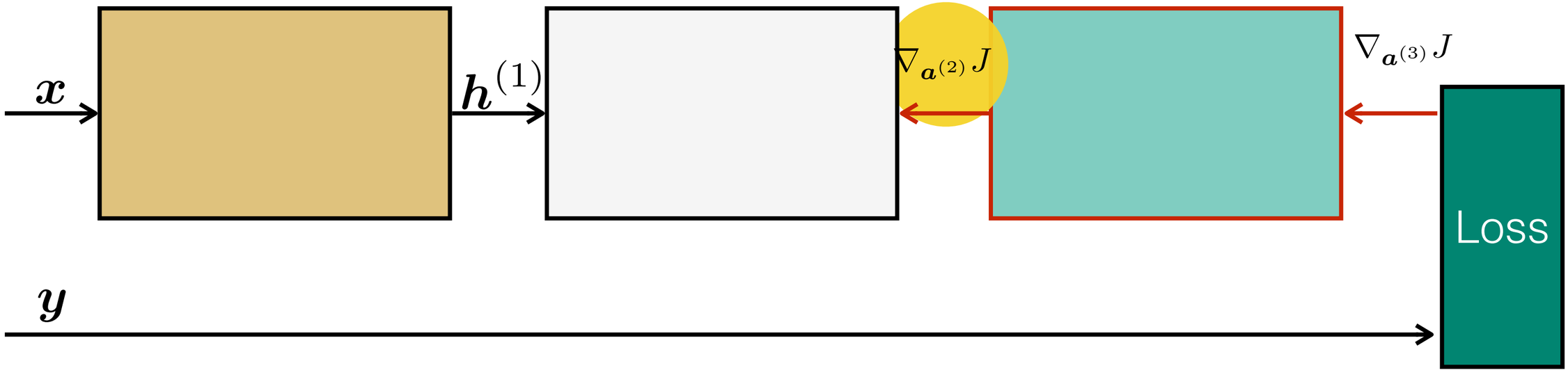
Backward Propagation (Update)



Compute gradients on weights and biases:

$$\nabla_{\boldsymbol{b}^{(3)}} J = \boldsymbol{g} + \alpha \nabla_{\boldsymbol{b}^{(3)}} \Omega(\boldsymbol{\theta}) \qquad \nabla_{\boldsymbol{W}^{(3)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(2)\top} + \alpha \nabla_{\boldsymbol{W}^{(3)}} \Omega(\boldsymbol{\theta})$$

Backward Propagation (L2)



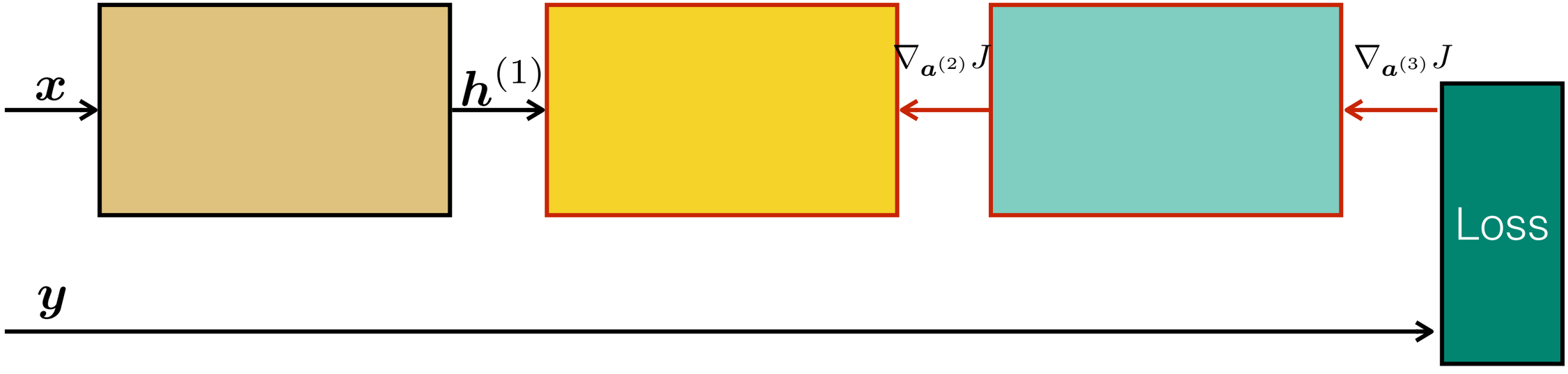
Propagate the gradients w.r.t. the next lower-level hidden layer's activation

$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(2)}} J = \boldsymbol{W}^{(3)\top} \boldsymbol{g}$$

Then, convert the gradient on the layer's output into a gradient on the pre-activation (element-wise nonlinearity):

$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(2)}} J = \boldsymbol{g} \odot g'(\boldsymbol{a}^{(2)})$$

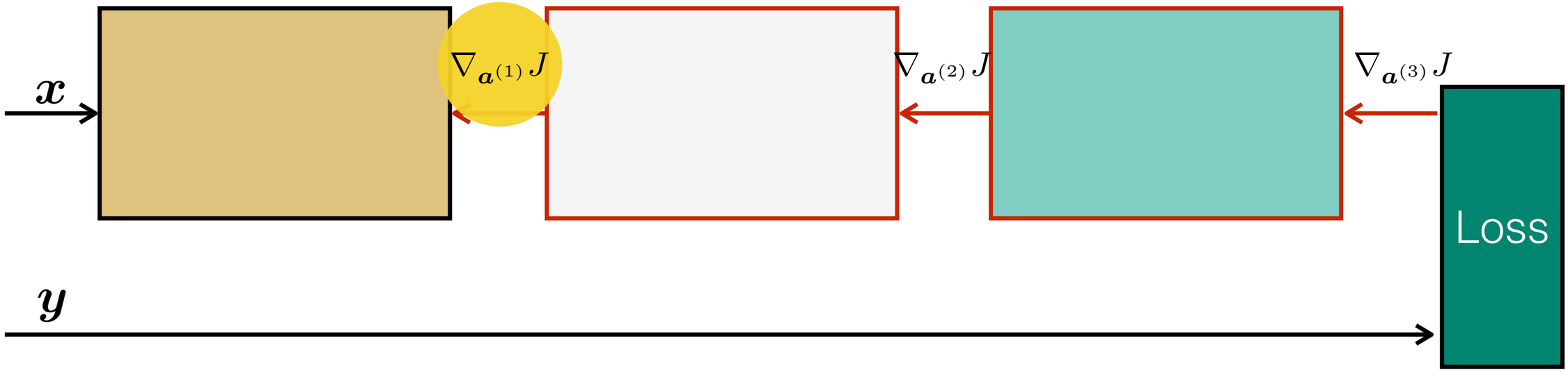
Backward Propagation (Update)



Compute gradients on weights and biases:

$$\nabla_{\boldsymbol{b}^{(2)}} J = \boldsymbol{g} + \alpha \nabla_{\boldsymbol{b}^{(2)}} \Omega(\boldsymbol{\theta}) \qquad \nabla_{\boldsymbol{W}^{(2)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(1)\top} + \alpha \nabla_{\boldsymbol{W}^{(2)}} \Omega(\boldsymbol{\theta})$$

Backward Propagation (L1)



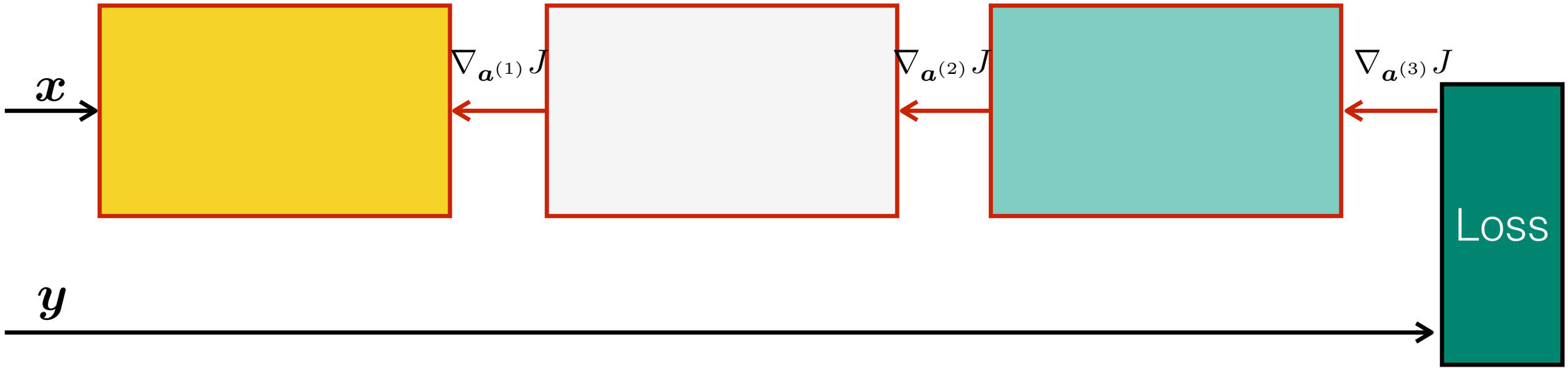
Propagate the gradients w.r.t. the next lower-level hidden layer's activation

$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(1)}} J = \boldsymbol{W}^{(2)\top} \boldsymbol{g}$$

Then, convert the gradient on the layer's output into a gradient on the pre-activation (element-wise nonlinearity):

$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(1)}} J = \boldsymbol{g} \odot g'(\boldsymbol{a}^{(1)})$$

Backward Propagation (Update)



Compute gradients on weights and biases:

$$\nabla_{\boldsymbol{b}^{(1)}} J = \boldsymbol{g} + \alpha \nabla_{\boldsymbol{b}^{(1)}} \Omega(\boldsymbol{\theta}) \qquad \nabla_{\boldsymbol{W}^{(1)}} J = \boldsymbol{g}\, \boldsymbol{x}^{\top} + \alpha \nabla_{\boldsymbol{W}^{(1)}} \Omega(\boldsymbol{\theta})$$