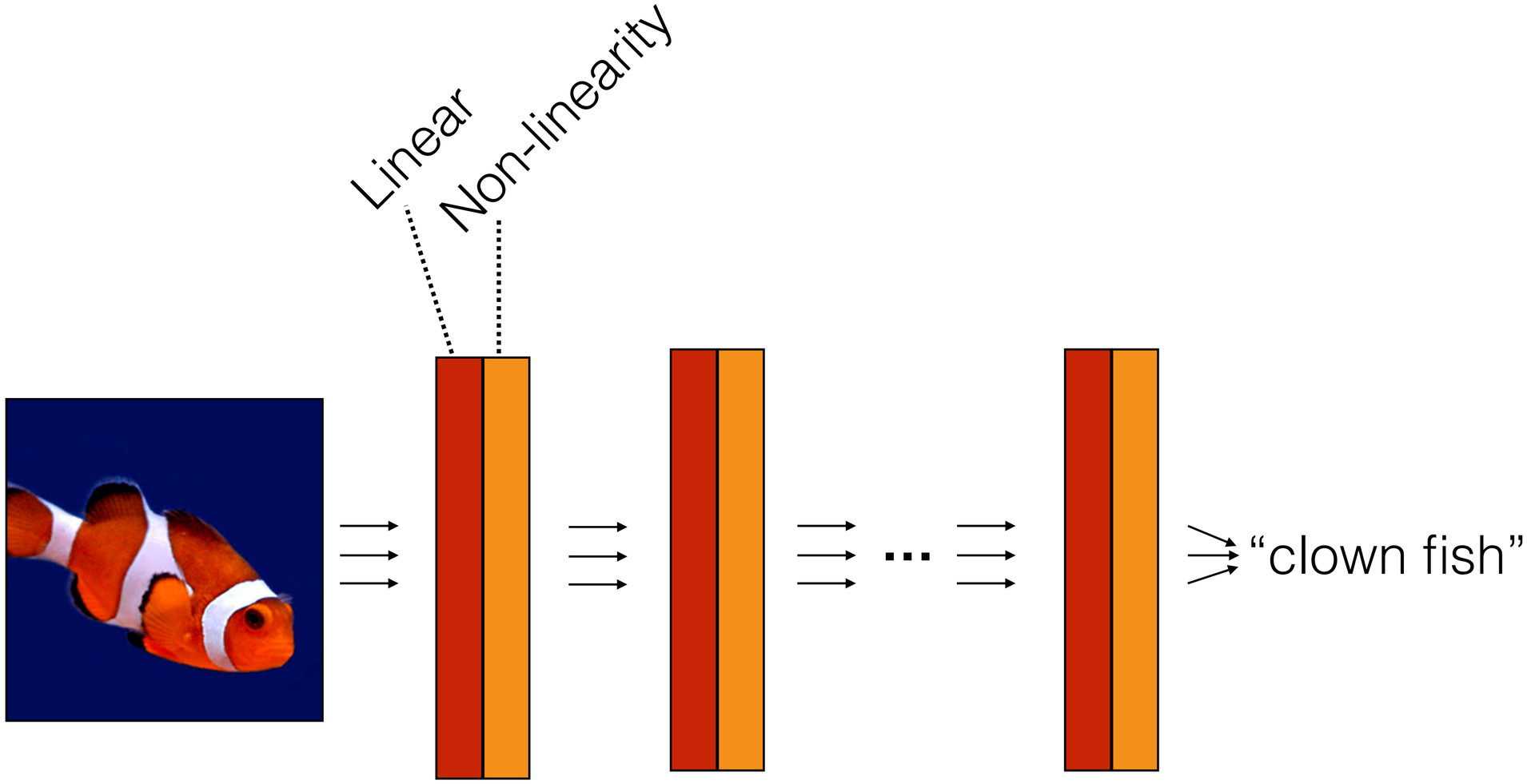
Lecture 8: Optimization

- Section this week: neural nets and optimization
- Reading:
  - Szeliski 5.3.5
  - Lecture notes by Roger Grosse (basis for much of this lecture)

Recall: neural nets and gradient descent

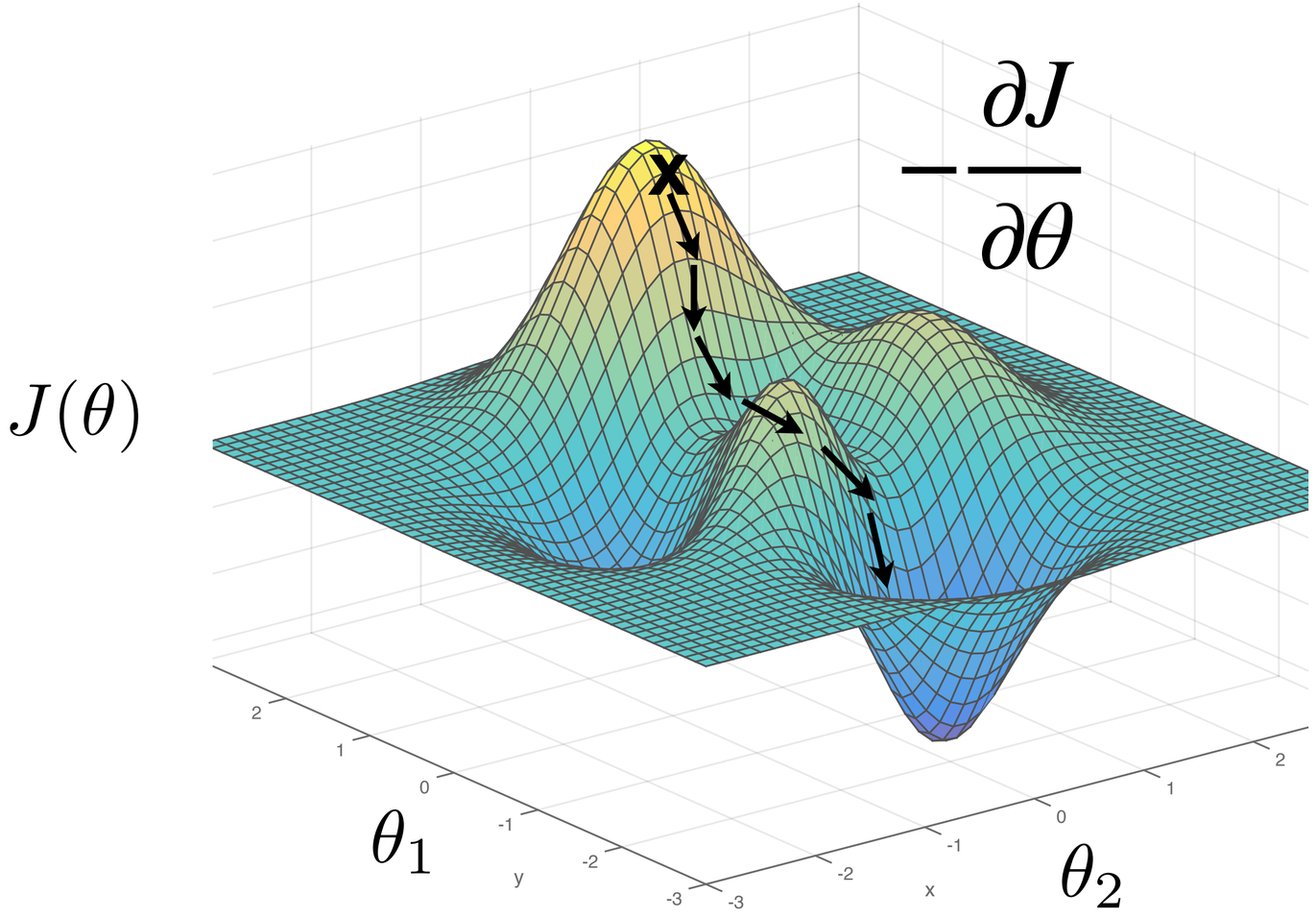
Recall: deep nets



$$f(\mathbf{x}) = f_L(\dots f_2(f_1(\mathbf{x})))$$

Source: Isola, Torralba, Freeman

Recall: Learn parameters using gradient descent



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

Source: Isola, Torralba, Freeman

Recall: learning parameters

Squared loss with single-variable network:

$$L = \frac{1}{2}(f(x) - y)^2$$

$$L = \frac{1}{2}(y - \sigma(wx + b))^2$$

Example source: Roger Grosse

Recall: computing derivatives with the chain rule

$$\frac{\partial L}{\partial w} = \frac{\partial}{\partial w}\left[\frac{1}{2}(y - \sigma(wx + b))^2\right]$$

$$= (y - \sigma(wx + b))\frac{\partial}{\partial w}(y - \sigma(wx + b))$$

$$= -(y - \sigma(wx + b))\sigma'(wx + b)\frac{\partial}{\partial w}(wx + b)$$

$$= -(y - \sigma(wx + b))\sigma'(wx + b)x$$

$$\frac{\partial L}{\partial b} = \frac{\partial}{\partial b}\left[\frac{1}{2}(y - \sigma(wx + b))^2\right]$$

$$= (y - \sigma(wx + b))\frac{\partial}{\partial b}(y - \sigma(wx + b))$$

$$= -(y - \sigma(wx + b))\sigma'(wx + b)\frac{\partial}{\partial b}(wx + b)$$

$$= -(y - \sigma(wx + b))\sigma'(wx + b)$$

Inefficient and time-consuming (for us to write out)

Example source: Roger Grosse

Learning parameters

Writing out the layers:

$z = wx + b$ linear (affine) layer

$t = \sigma(z)$ nonlinearity

$L = \frac{1}{2}(y - t)^2$ loss

Example source: Roger Grosse

Learning parameters

Writing out the layers:

$$z = wx + b$$

$$t = \sigma(z)$$

$$L = \frac{1}{2}(y - t)^2$$

Another way to write derivatives:

$$\frac{\partial L}{\partial t} = t - y$$

Each step is easy to compute, and we can reuse computation.

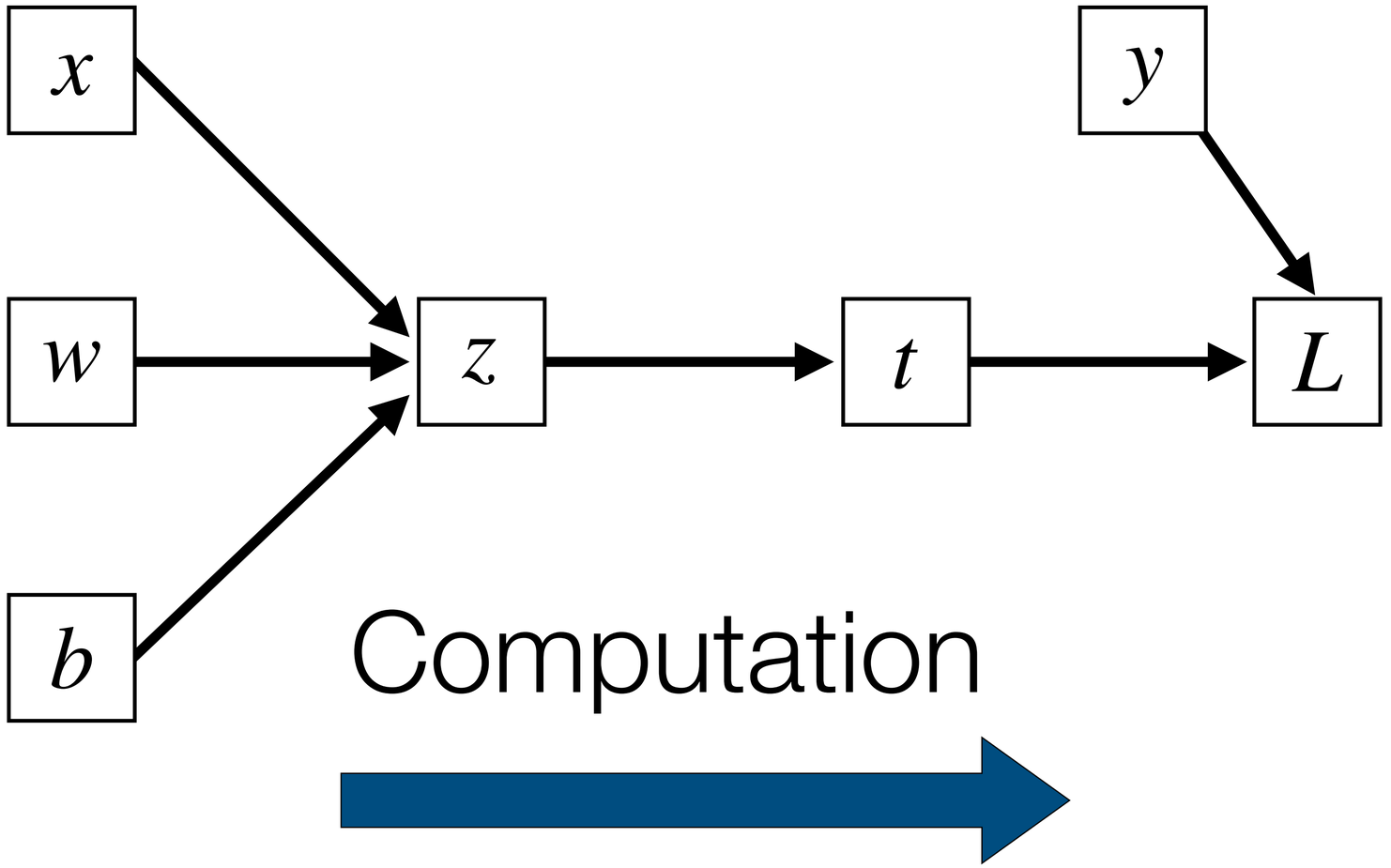
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial t}\frac{\partial t}{\partial z} = \frac{\partial L}{\partial t}\sigma'(z)$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z}x$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$$

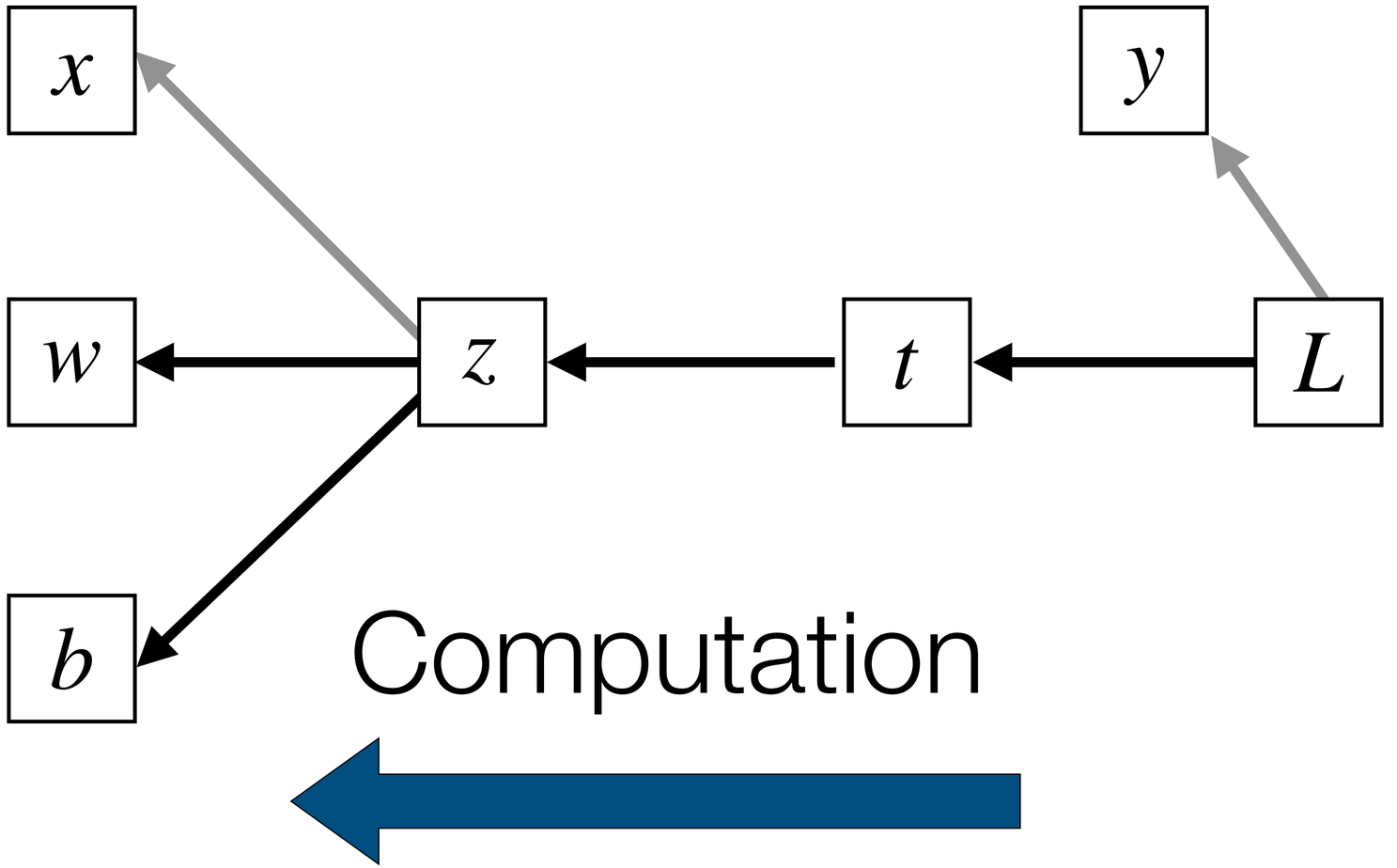
Example source: Roger Grosse

Computation graph: loss



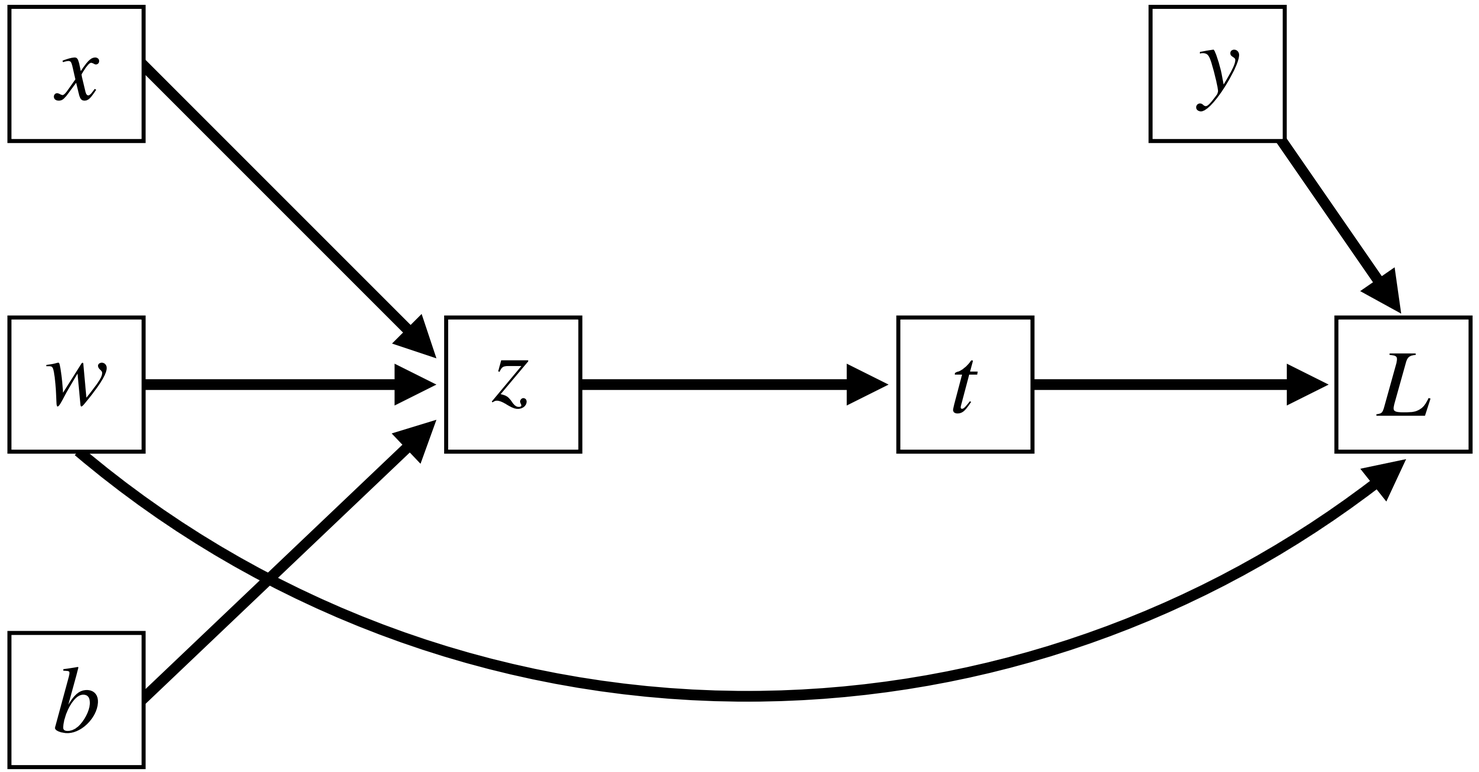
Adapted from: Roger Grosse

Computation graph: derivatives



Adapted from: Roger Grosse

One subtlety: nodes with multiple outputs



$$L = (y - t)^2 + \|w\|^2$$

Adapted from: Roger Grosse

Chain rule

Single-variable chain rule:

$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$$

Multivariable chain rule:

$$\frac{\partial}{\partial x} f(g(x), h(x)) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} + \frac{\partial f}{\partial h} \frac{\partial h}{\partial x}$$

Multivariable chain rule: example

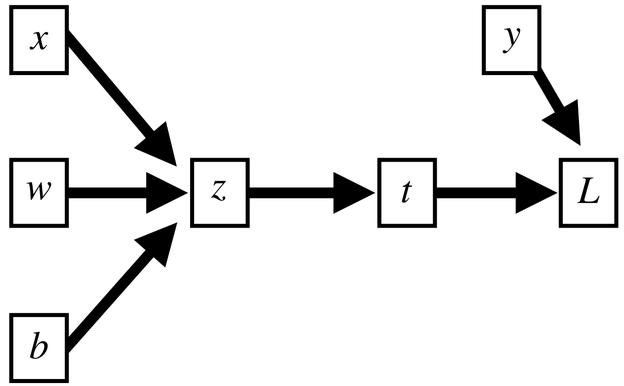
$$f(g(x), h(x)) = g(x) + g(x)h(x)$$
$$g(x) = x^2$$
$$h(x) = \exp(x)$$

$$\boxed{\frac{\partial}{\partial x} f(g(x), h(x)) = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} + \frac{\partial f}{\partial h}\frac{\partial h}{\partial x}}$$

$$\begin{aligned}
\frac{\partial}{\partial x} f(g(x), h(x)) &= \frac{\partial}{\partial x}[g(x) + g(x)h(x)] \\
&= \underbrace{[1 + h(x)]g'(x)}_{\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}} + \underbrace{g(x)h'(x)}_{\frac{\partial f}{\partial h}\frac{\partial h}{\partial x}} \\
&= (1 + \exp(x))(2x) + x^2 \exp(x) \\
&= 2x + 2x\exp(x) + x^2 \exp(x)
\end{aligned}$$

Backpropagation

Step #1: Compute loss and every value in computation graph.



Forward pass:

for i = 0 … N:

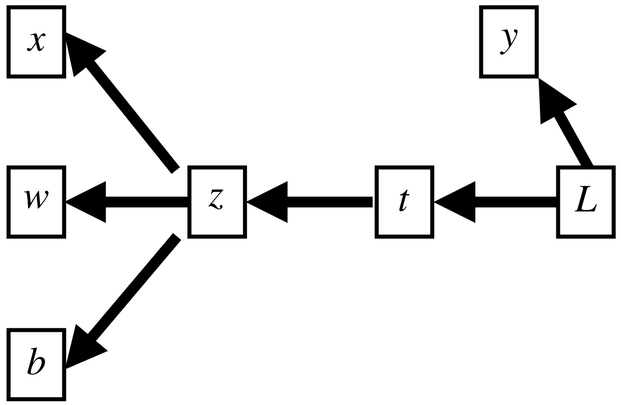
compute each node value v_i using values from previous nodes

where N nodes $v_0, v_1, \ldots v_n$ are ordered topographically (i.e. inputs always come before outputs).

Adapted from: Roger Grosse

Backpropagation

Step #2: Compute derivatives.



Backward pass:

$v_N = 1$

for i = N-1 .. 0:

$$\frac{\partial L}{\partial v_i} = \sum_{j \in \text{Outputs}(v_i)} \frac{\partial L}{\partial v_j} \frac{\partial v_j}{\partial v_i}$$

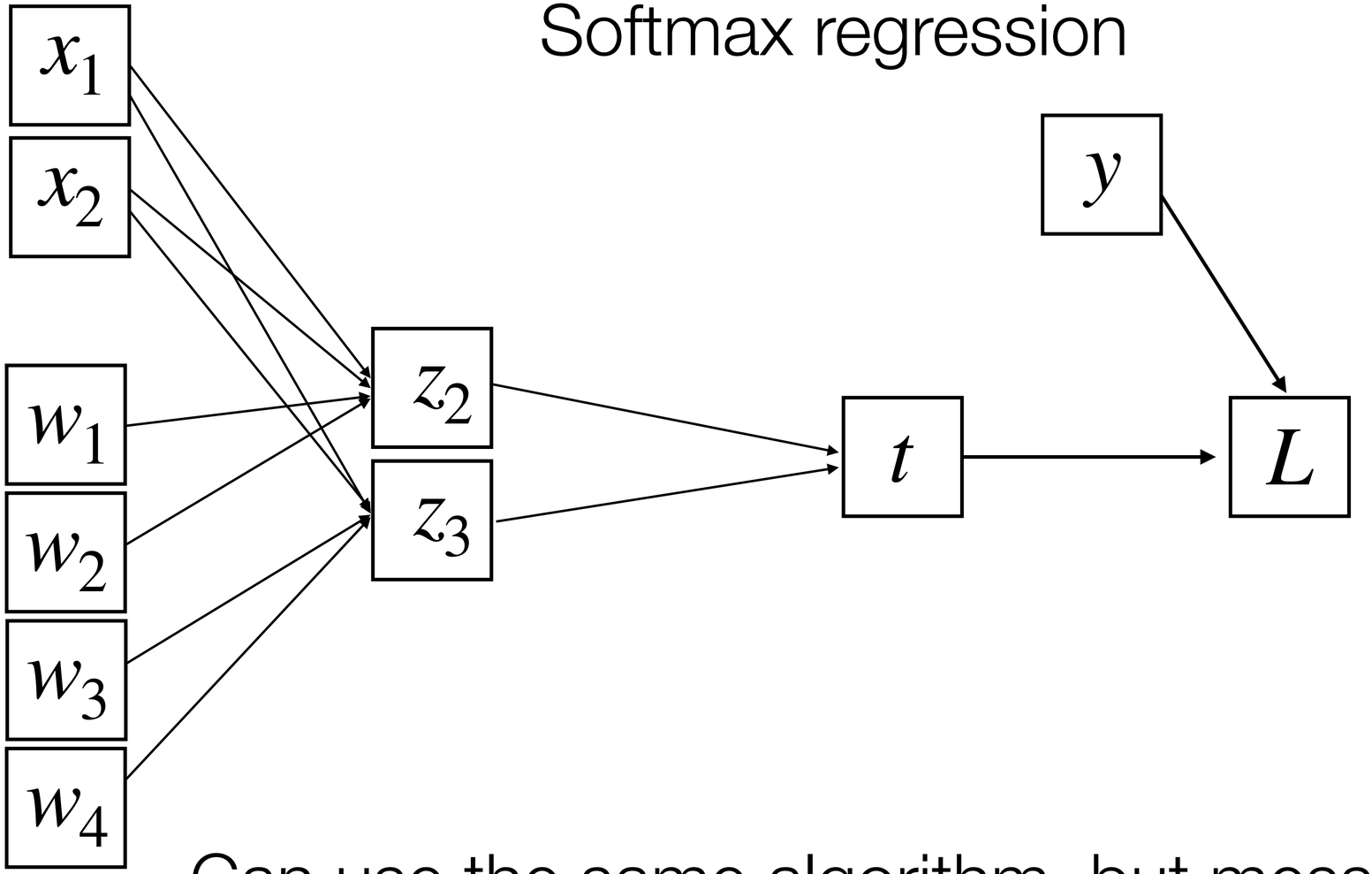
Starting from the loss, work your way to the inputs. Compute the derivative of the loss w.r.t. each value.

Summary: compute derivatives efficiently using the chain rule.

Adapted from: Roger Grosse

What about vector-valued functions?

Softmax regression



Can use the same algorithm, but messy…

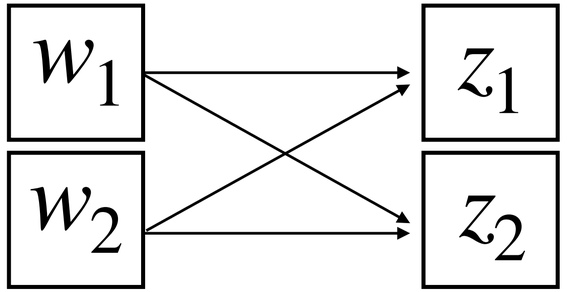
Adapted from: Roger Grosse

Handling vectors

Vector-valued functions:

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 2w_1 + 3w_2 \\ 5w_1 - 2w_2 \end{bmatrix}$$

Computation graph:



Multivariable chain rule:

$$\frac{\partial L}{\partial w_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_i}$$

In matrix form:

$$\frac{\partial L}{\partial \mathbf{w}} = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{w}}\right)^\top \frac{\partial L}{\partial \mathbf{z}}$$

Jacobian matrix:

$$\begin{bmatrix} \frac{\partial z_1}{\partial w_1} & \frac{\partial z_1}{\partial w_2} \\ \frac{\partial z_2}{\partial w_1} & \frac{\partial z_2}{\partial w_2} \end{bmatrix}$$

Adapted from: Roger Grosse

Backward step (with vectors)

Recall step #2: Computing derivatives.

Backward pass:

$$\mathbf{v_N} = 1$$

for i = N-1 .. 0:

$$\frac{\partial L}{\partial \mathbf{v_i}} = \sum_{j \in \text{Outputs}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^{\mathsf{T}} \frac{\partial L}{\partial \mathbf{v_j}}$$

Forward pass is the same as it was before.

Adapted from: Roger Grosse

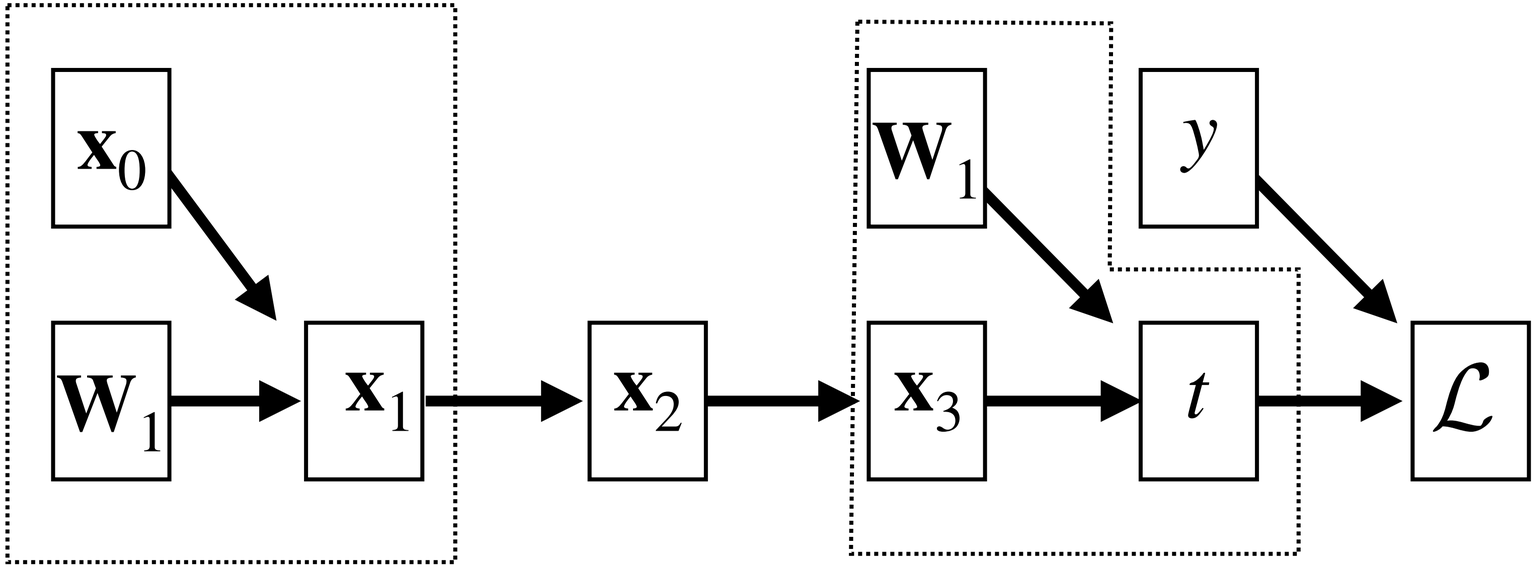
Example: multi-layer regression

Linear transformation

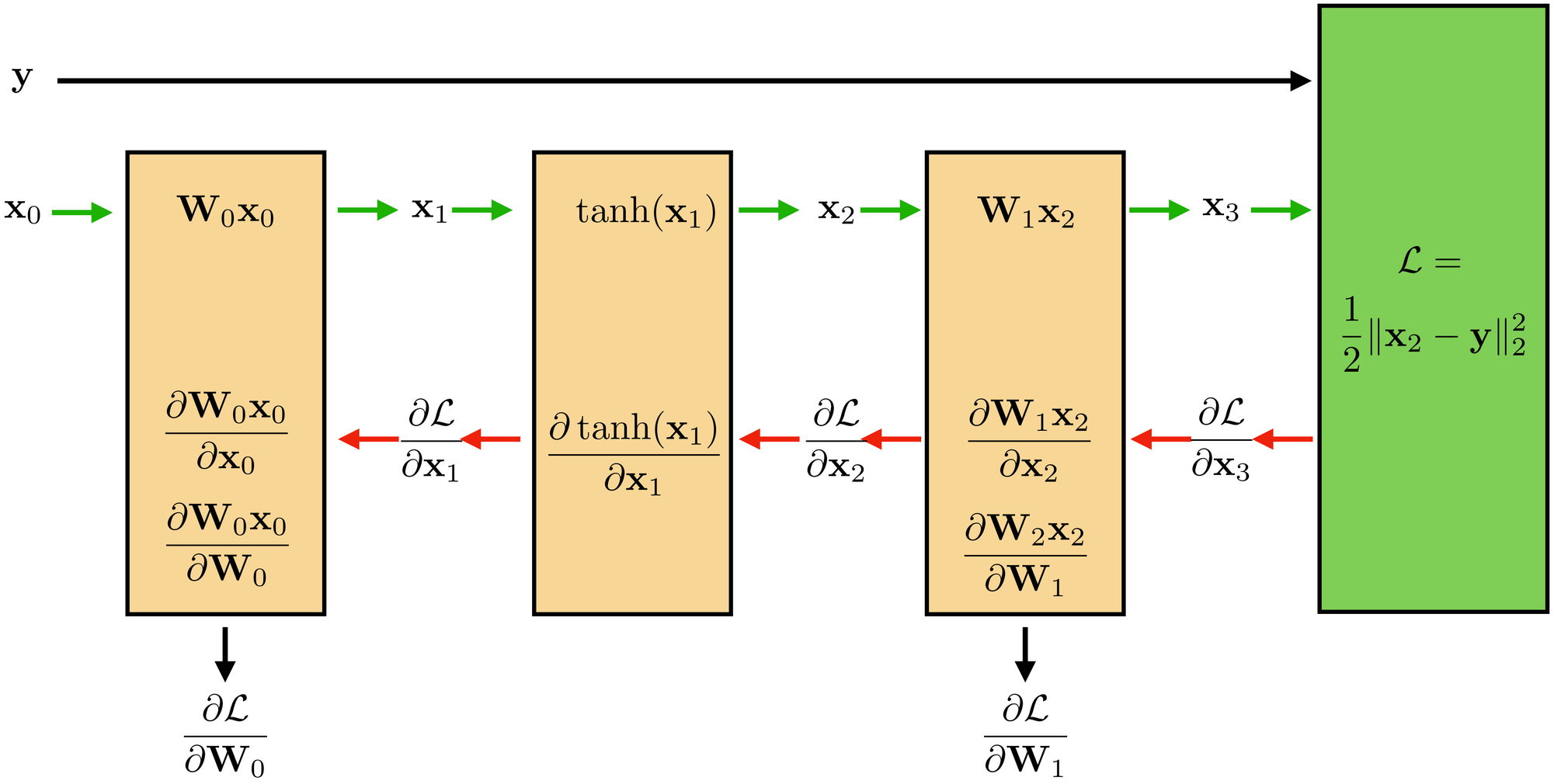
Elementwise nonlinearity

Squared L_2 norm

$$\mathcal{L} = \frac{1}{2} \| \mathbf{W}_1 \tanh(\mathbf{W}_0 \mathbf{x}_0) - y \|^2$$



Rewrite as layers to reuse common operations

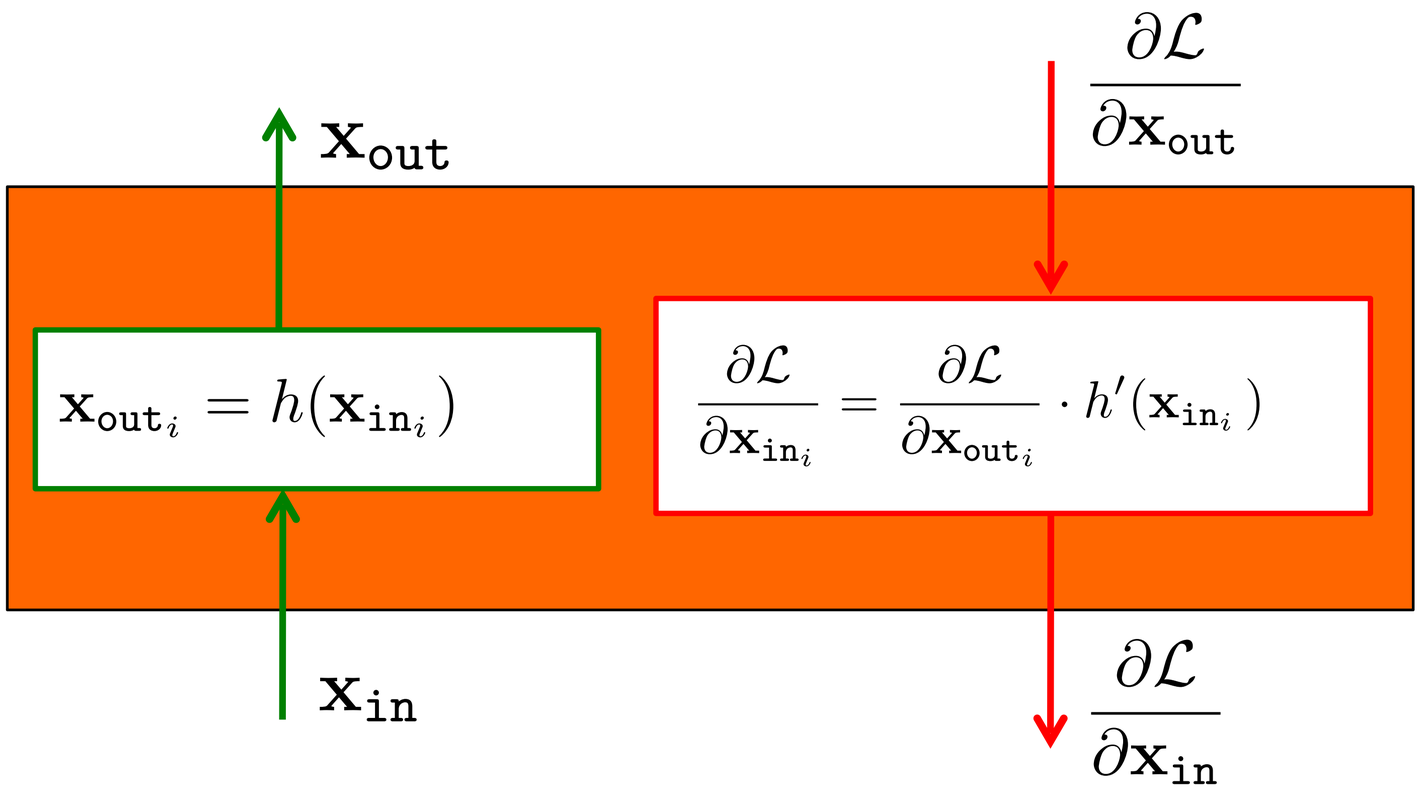


linear layer

linear layer

Source: Torralba, Isola, Freeman

Pointwise function



Example #1:

$$\mathbf{x}_{\text{out}_i} = \tanh(\mathbf{x}_{in_i})$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{in_i}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{out_i}}(1 - \tanh^2(\mathbf{x}_{in_i}))$$

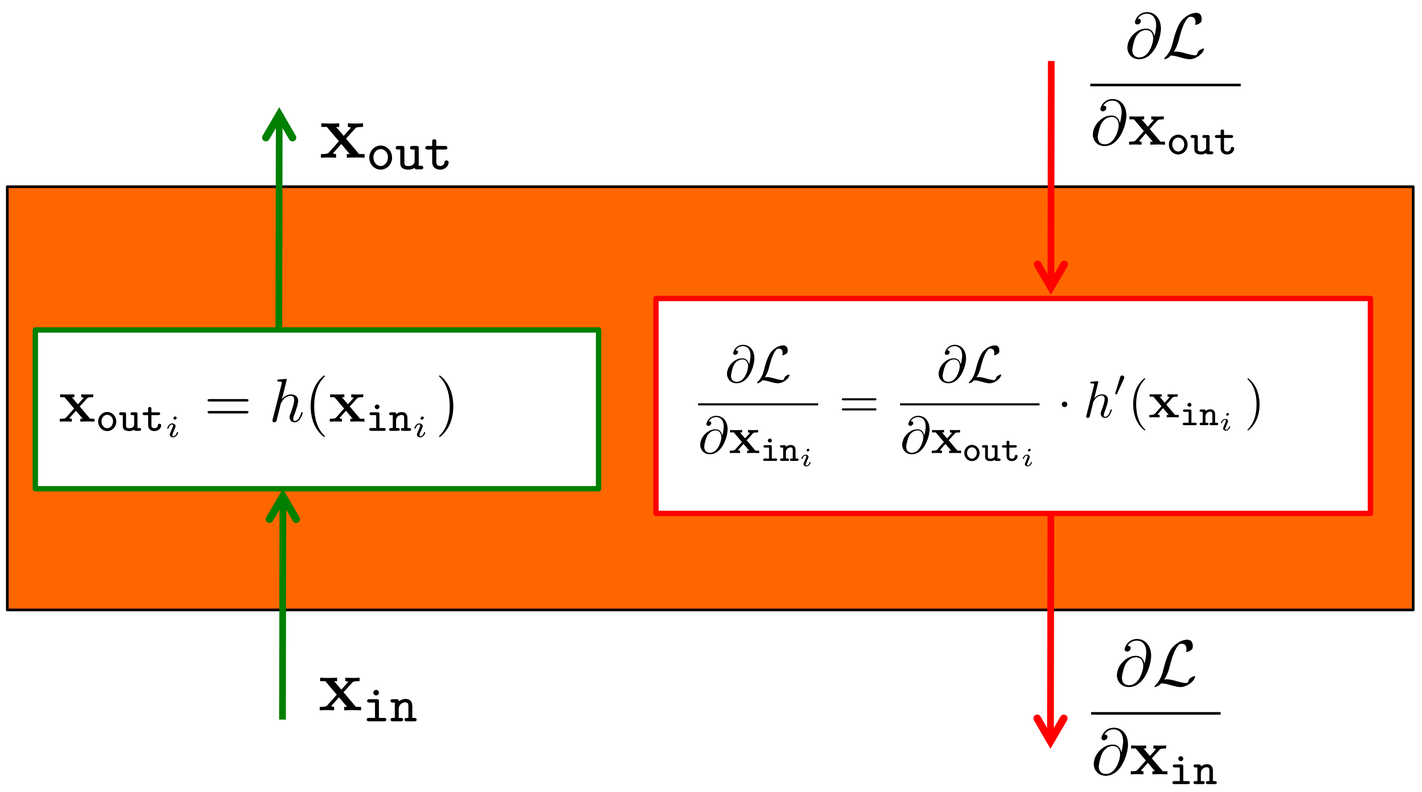
Example #2:

$$\mathbf{x}_{\text{out}_i} = \max(0, \mathbf{x}_{in_i})$$

What's the backward pass?

Source: Torralba, Isola, Freeman

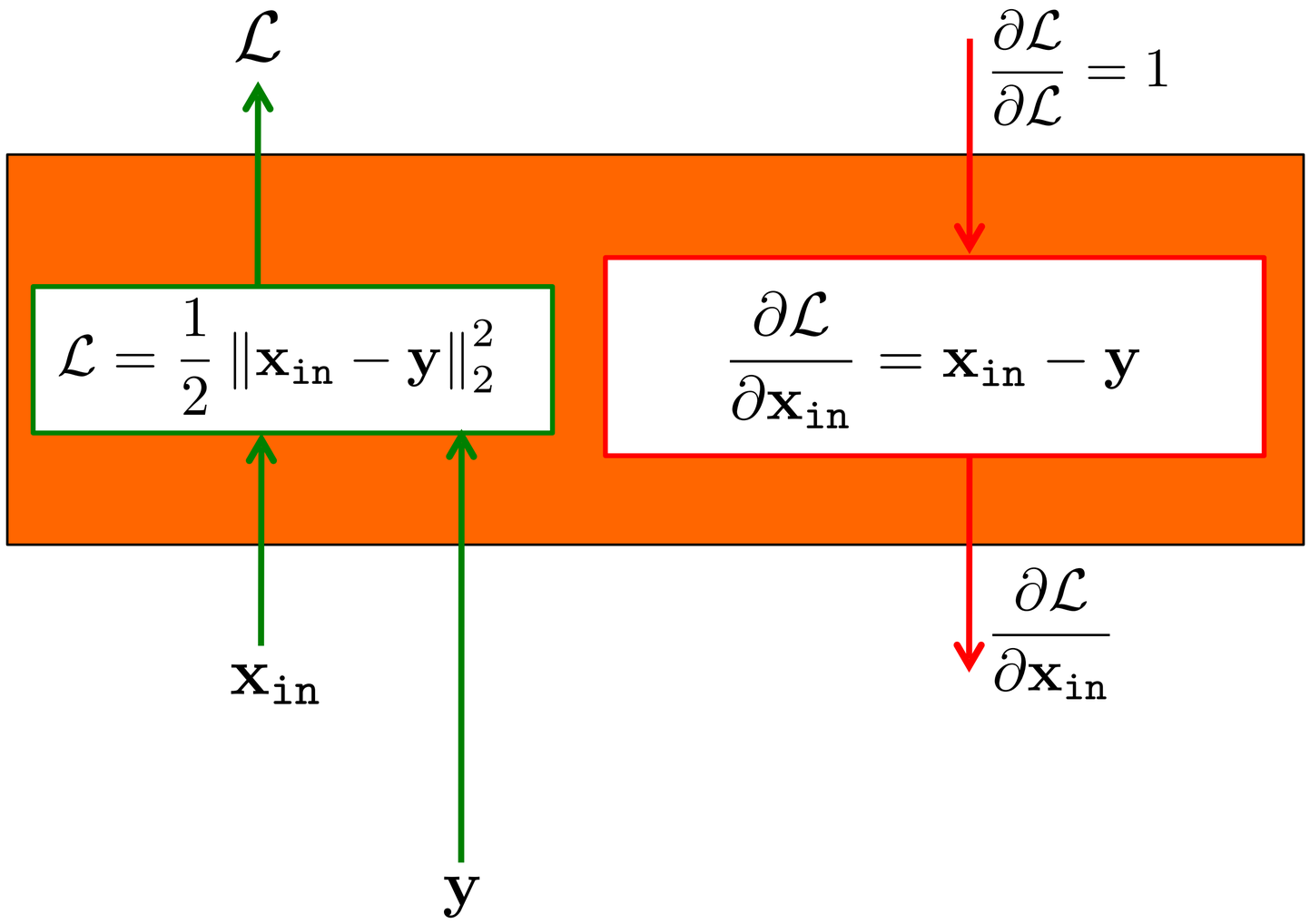
Implementation



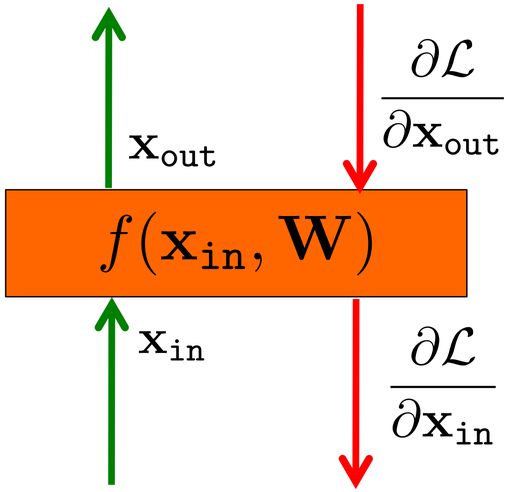

```python
class ReLULayer:
  def forward(self, x):
    # save for backward pass
    self.x = x
    return np.maximum(x, 0)

  def backward(self, grad_output):
    grad_input = grad_output.clone()
    grad_input[self.x < 0] = 0
    return grad_input
```

See https://pytorch.org/tutorials/beginner/pytorch_with_examples.html for functioning code.

Squared cost layer



Source: Torralba, Isola, Freeman



Linear layer

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

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

- Backprop to input:

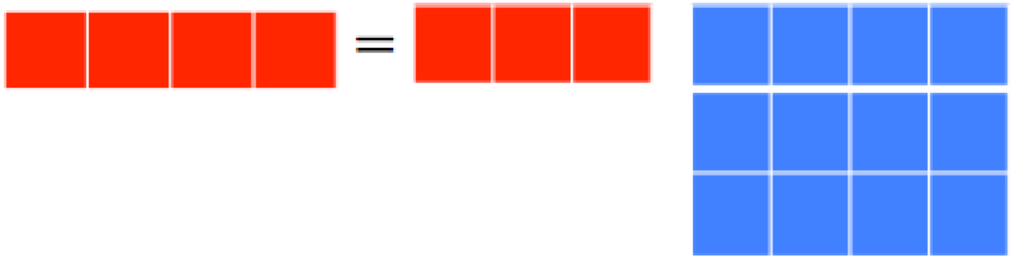
$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial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

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

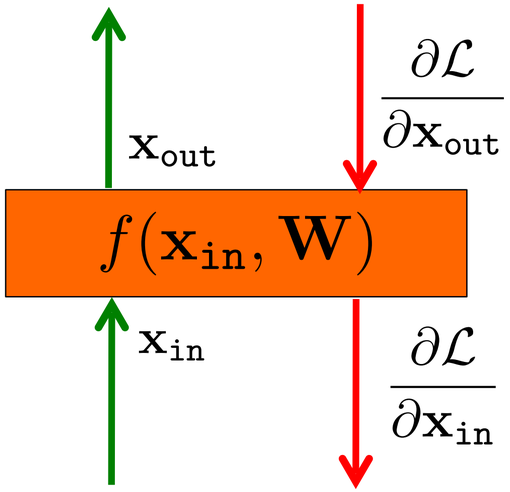
$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{x}_{\text{in}_j}} = \mathbf{W}_{ij} \longrightarrow \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{x}_{\text{in}}} = \mathbf{W}$$

Therefore:

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



Source: Torralba, Isola, Freeman



Linear layer

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

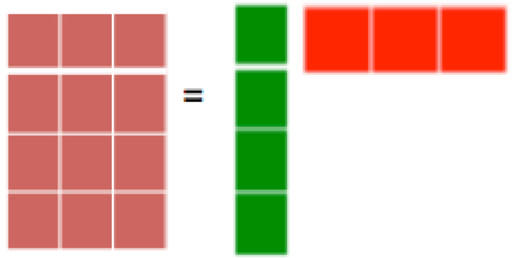
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial f(\mathbf{x}_{\text{in}}, \mathbf{W})}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}} \cdot \frac{\partial \mathbf{x}_{\text{out}}}{\partial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

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

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

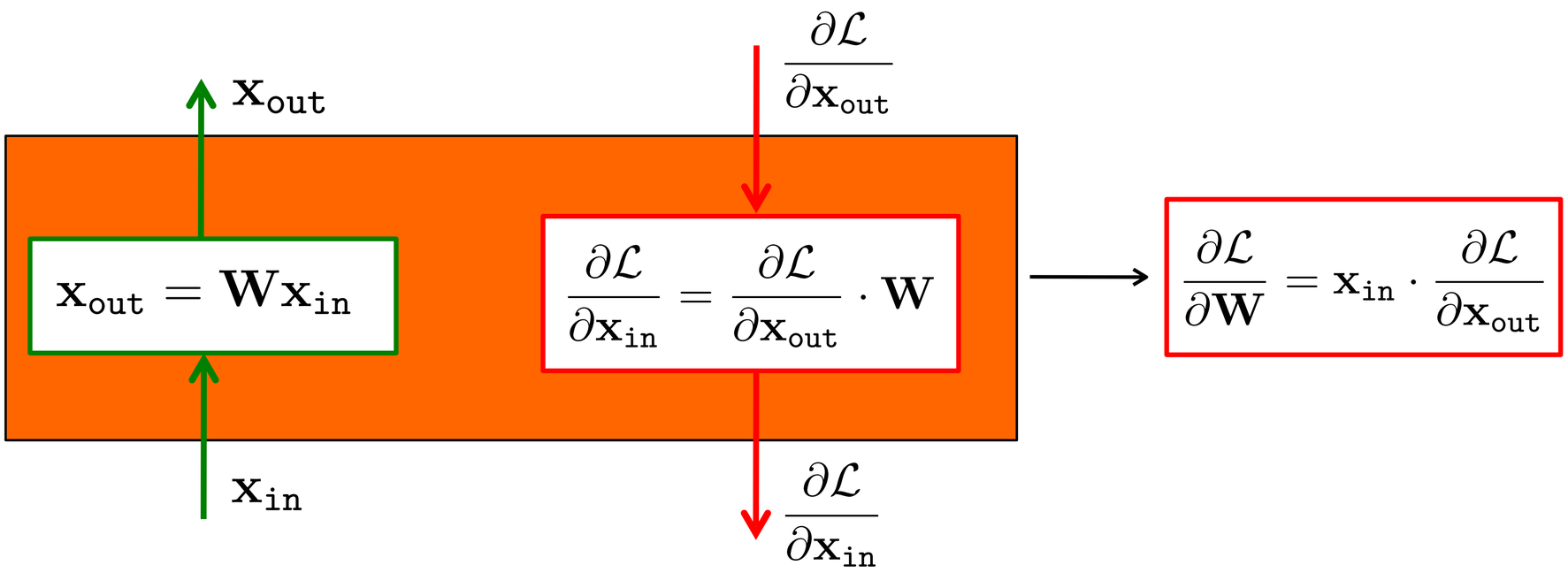
$$\frac{\partial \mathbf{x}_{\text{out}_i}}{\partial \mathbf{W}_{ij}} = \mathbf{x}_{\text{in}_j}$$

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



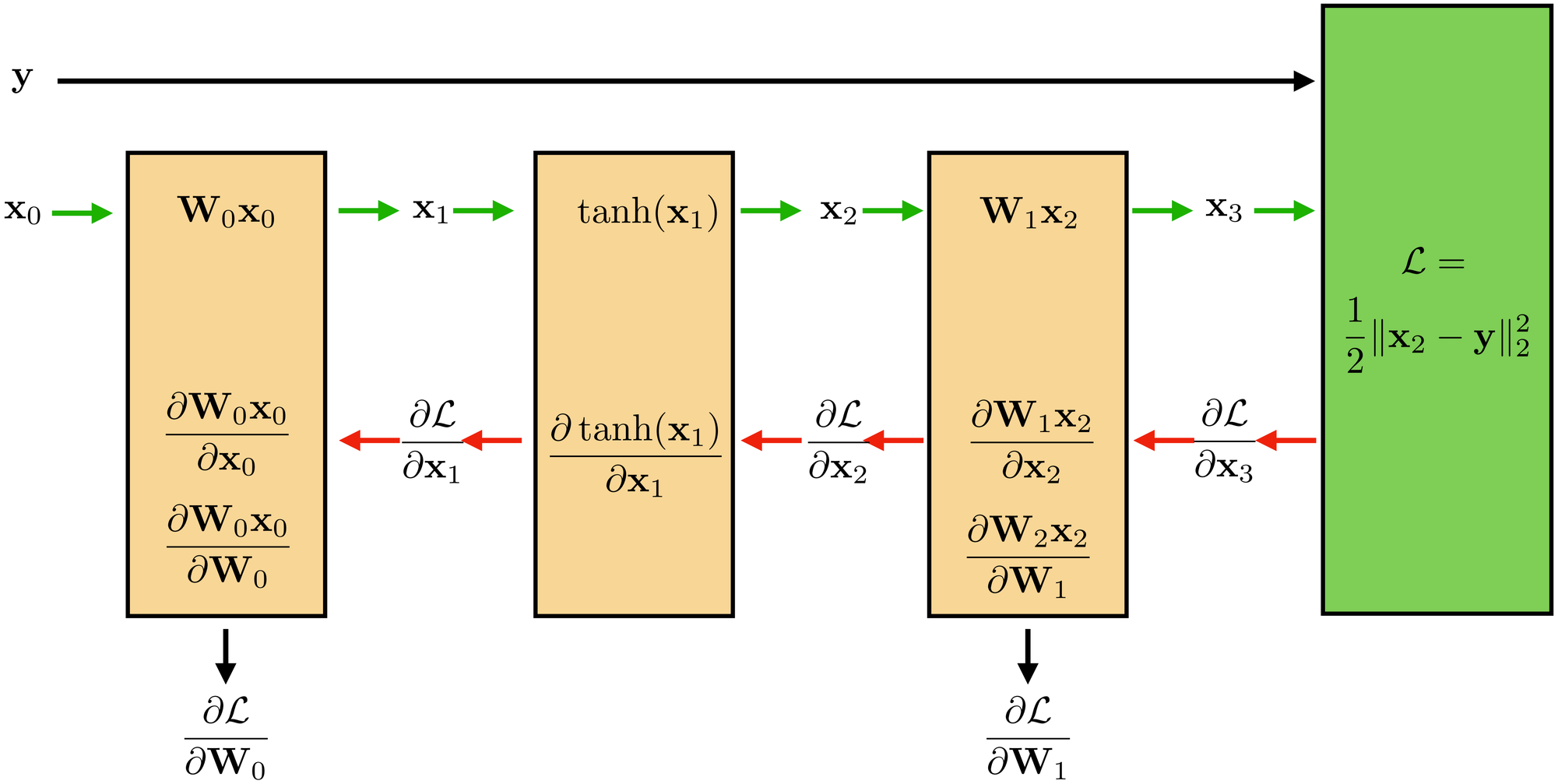
Source: Torralba, Isola, Freeman

Linear layer



Source: Torralba, Isola, Freeman

Recall: multilayer model



We need to compute all these terms so that we can find the gradients at the bottom.

Source: Torralba, Isola, Freeman

Automatic differentiation

- **Backpropagation**: algorithm for computing derivatives
- Also known as **reverse-mode autodifferentiation**
- Usually implement backprop using autodifferentiation (“autodiff”) software package.
  - Build your program out of primitive operations, similar to `numpy` operations
  - Behind the scenes, the autodiff package builds the computation graph for you!
  - It’s *not* finite difference. Computes exact gradients using backprop!

Automatic differentiation

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

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

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

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

very sneaky!

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

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

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

Autograd constructs a function for computing derivatives

Figure source: Roger Grosse

Deep learning libraries

- In this class, we'll use a popular library called PyTorch.
- Common alternatives: JAX and TensorFlow
- Designed to use GPUs (graphics processing unit).
- Requires writing vectorized code. Each vectorized op runs efficiently on GPU (e.g., matrix multiplication may be 100x faster).
- They also provide other utilities, like data loading.
- More on these in next week's section.

Optimization

Loss:

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

Sum over N training pairs

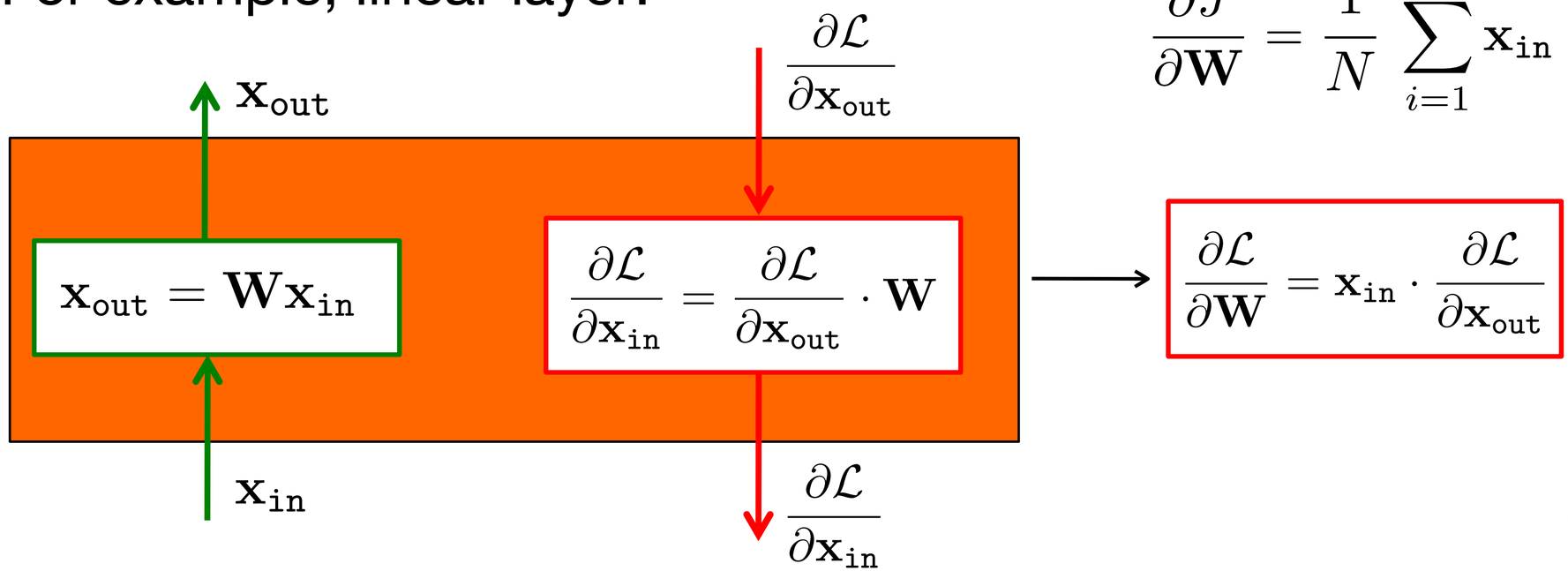
Gradient descent:

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

Average over examples:

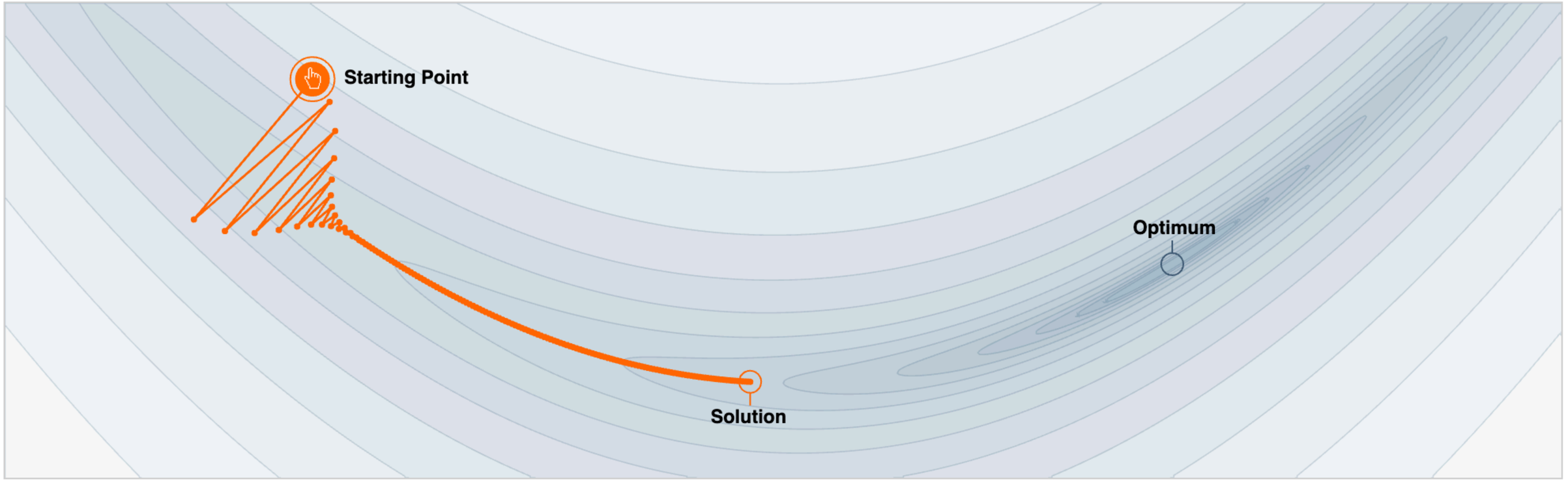
$$\frac{\partial J}{\partial \mathbf{W}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_{\text{in}} \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{x}_{\text{out}}}\bigg|_{\mathbf{x}_i, \mathbf{y}_i}$$

For example, linear layer:



SGD extensions

Dealing with oscillations



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

Momentum

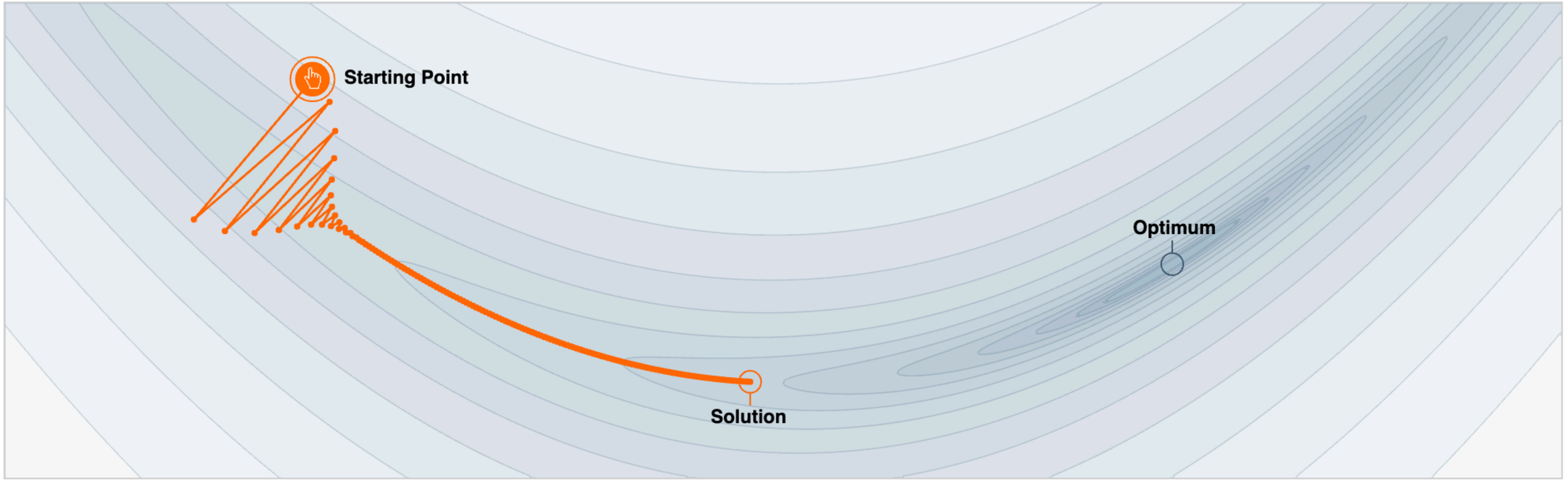
- Also known as the “heavy ball momentum”

$$\mathbf{v}^{(k+1)} \leftarrow \beta \mathbf{v}^{(k)} - \eta \frac{\partial J}{\partial \mathbf{W}}(\mathbf{W}^{(k)})$$

$$\mathbf{W}^{(k+1)} \leftarrow \mathbf{W}^{(k)} + \mathbf{v}^{(k+1)}$$

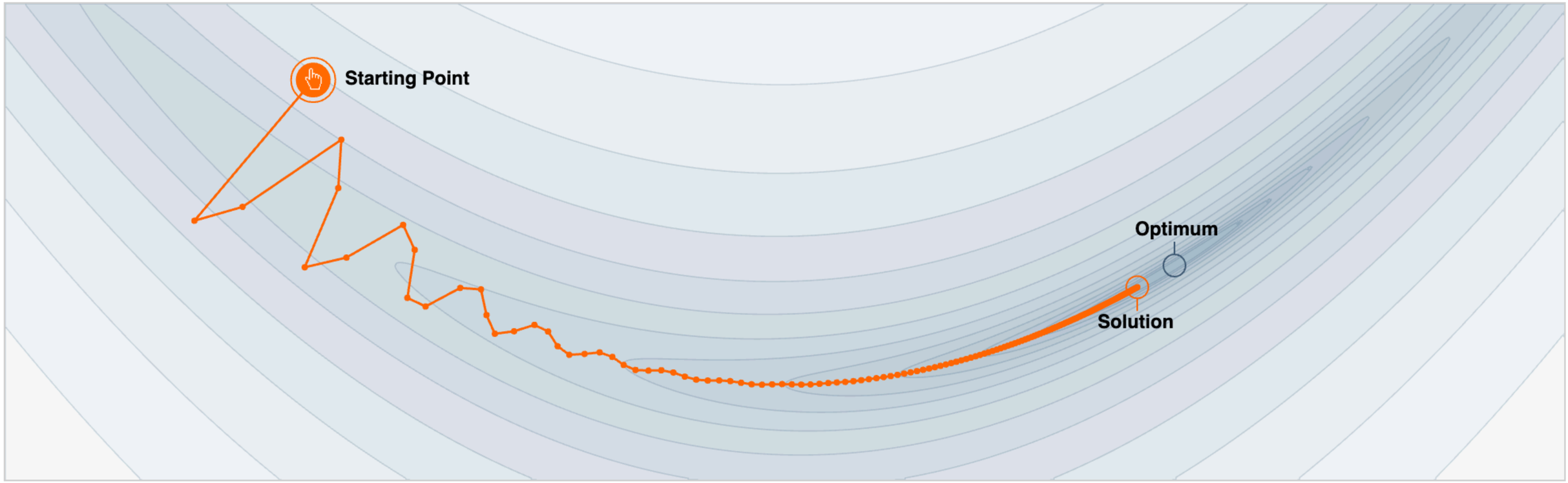
- Remember previous gradients and “build up speed”, setting $0 \leq \beta < 1$
  - If $\beta = 0$, equivalent to vanilla gradient descent
  - Why not $\beta = 1$?
- Reduces oscillations in high curvature regions, and picks up speed in when the loss surface is nearly flat.

Dealing with oscillations (no momentum)



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

Dealing with oscillations (with momentum)



$\beta = 0.99$

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

Adaptive gradient methods

- Scale each gradient component using a running average of its magnitude.

RMSProp:

$$\mathbf{g}^{(k+1)} \leftarrow \frac{\partial J}{\partial \mathbf{W}}(\mathbf{W}^{(k)})$$

$$\mathbf{s}^{(k+1)} \leftarrow \beta \mathbf{s}^{(k)} + (1-\beta)(\mathbf{g}^{(k+1)})^2$$

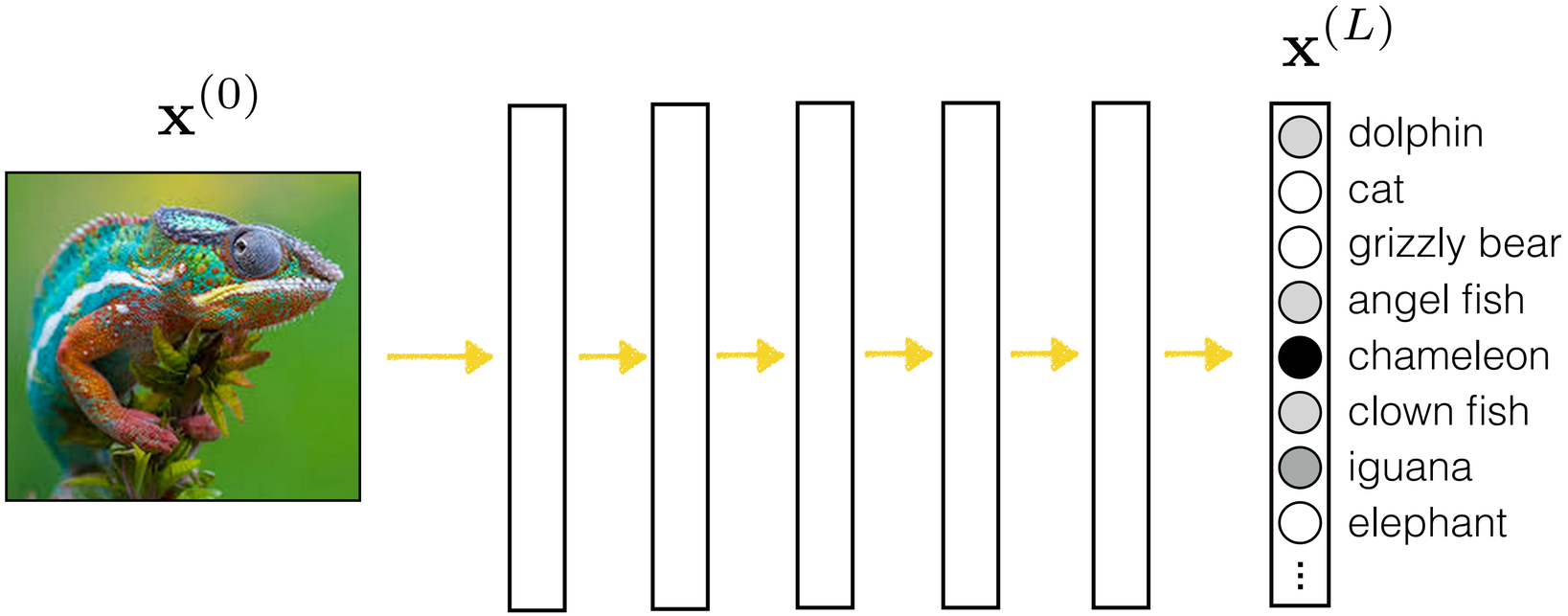
$$\mathbf{W}^{(k+1)} \leftarrow \mathbf{W}^{(k)} - \eta \frac{1}{\sqrt{\mathbf{s}^{(k+1)} + \epsilon}} \odot \mathbf{g}^{(k+1)}$$

where ϵ is a small constant

- After scaling, each update to the network parameters has approximately a magnitude of η.
- Can be viewed as a very rough approximation to 2nd-order optimization.
- A very common method is **Adam** (also AdamW), which is essentially RMSProp + momentum.

Other uses for backprop

Unit visualization via backprop



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

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

Source: Torralba, Isola, Freeman

Unit visualization via backprop

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

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

Source: Torralba, Isola, Freeman

Unit visualization via backprop

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

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

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

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

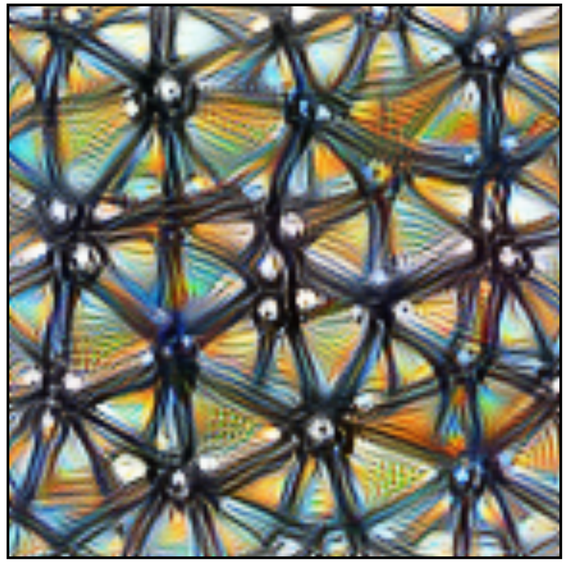
Source: Torralba, Isola, Freeman

Unit visualization via backprop

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

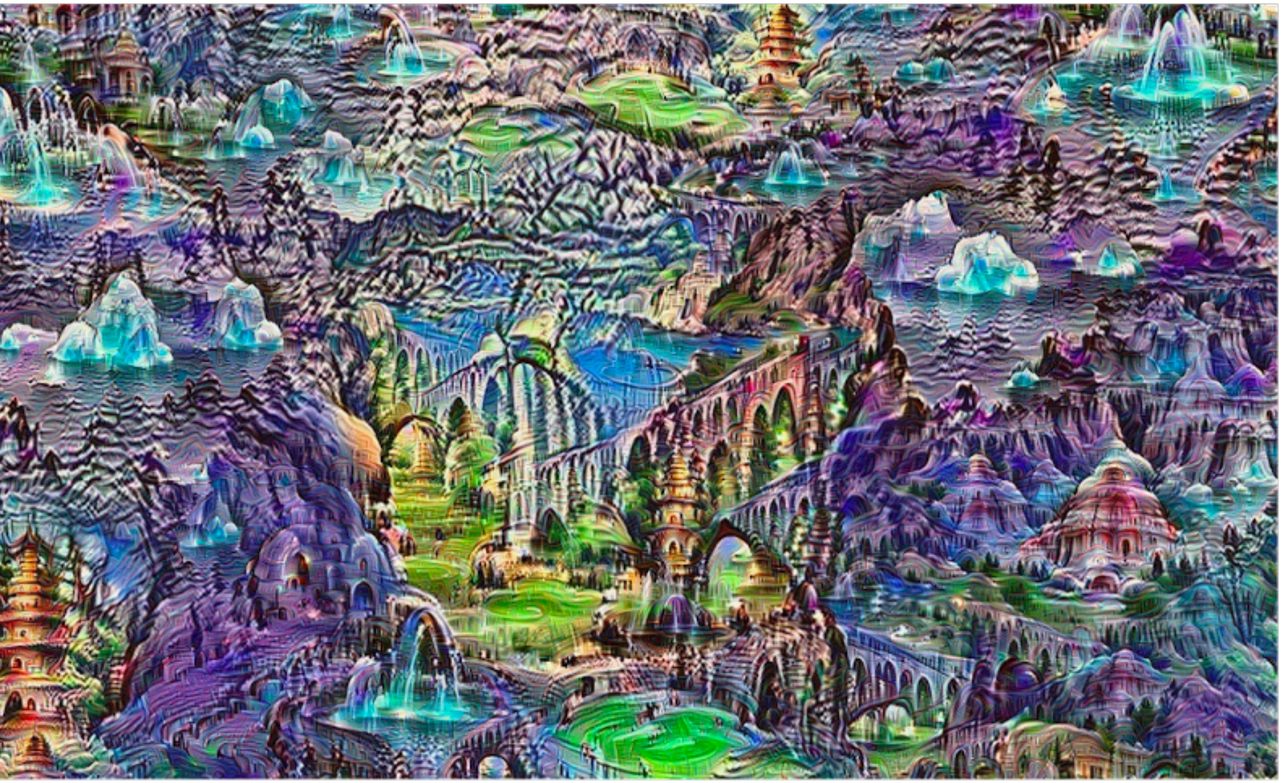
$$\mathbf{x}^{(0)^{k+1}} \leftarrow \mathbf{x}^{(0)^k} + \eta \frac{\partial(x_j^{(L)} + \lambda R(\mathbf{x}^{(0)}))}{\partial \mathbf{x}^{(0)}}$$

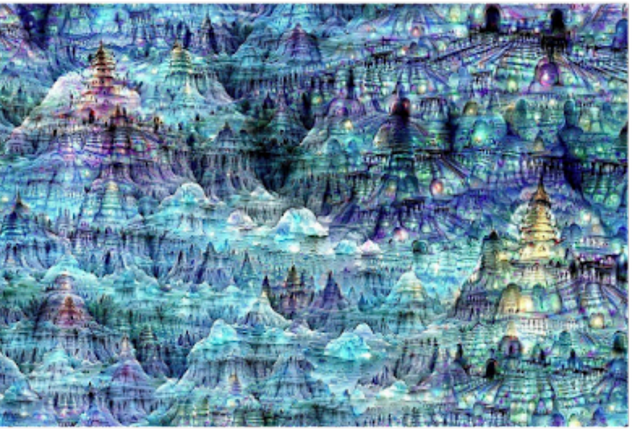
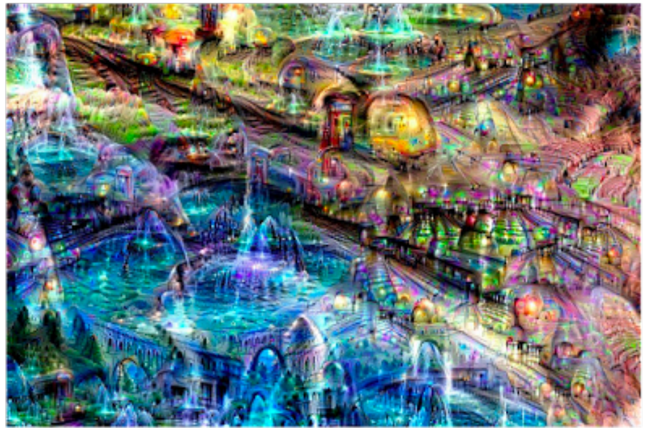
Make an image that maximizes the value of a random neuron in the middle of the network:

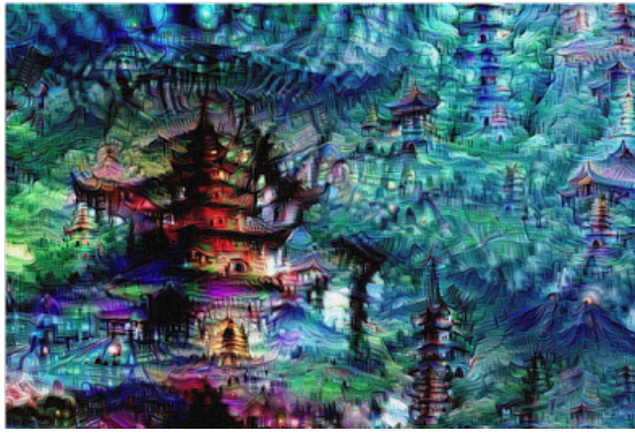


[https://distill.pub/2017/feature-visualization/]

Source: Torralba, Isola, Freeman





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

Source: Torralba, Isola, Freeman

Next class: convolutional neural networks