Computational Systems Biology
Deep Learning in the Life Sciences

6.802 6.874 20.390 20.490 HST.506

David Gifford
Lecture 2
February 7, 2017

Feedforward Networks

http://mit6874.github.io

Overall goal for today

- Teach you how to optimize an arbitrary feedforward network function that includes linear and non-linear operations using gradient descent

Linear regression can be solved by gradient descent

```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.mul(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)

# Gradient descent
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Today's lecture

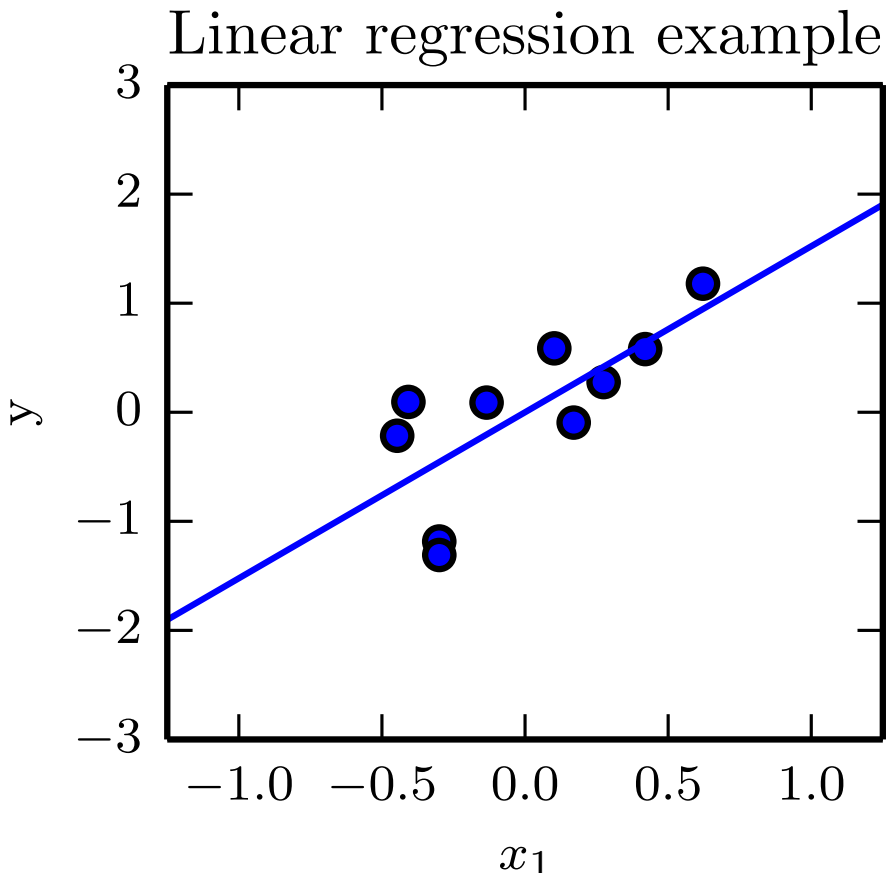
- Optimizing linear regression with gradient descent
  - Closed form solutions exist for linear models
  - Gradient descent can solve for unknowns
- Optimizing logistic regression with gradient descent
  - Non-linear models can solve harder problems
  - Backpropogation computes gradients for large non-linear networks

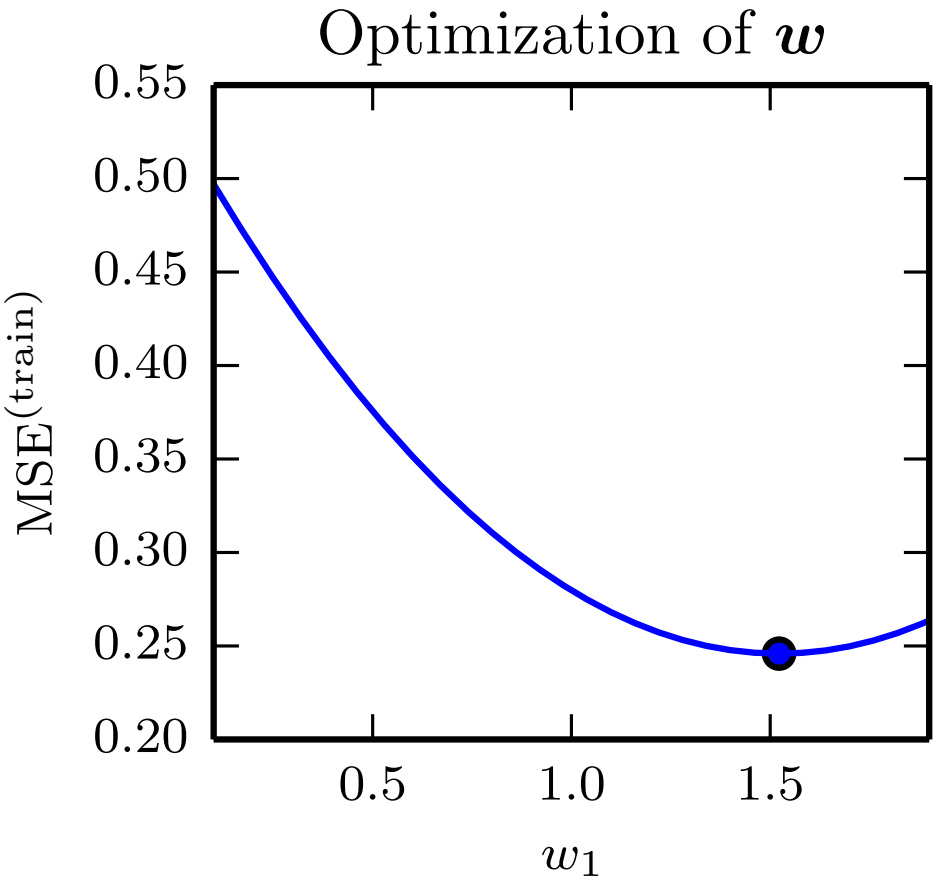
Solving linear feedforward networks (Part I)

Linear regression predicts predicts unobserved Y from X

$$\underbrace{\mathbf{X} = \begin{pmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \cdots & \cdots \\ 1 & x^{(n)} \end{pmatrix}}_{n \times (D+1)} \quad \underbrace{W = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_D \end{pmatrix}}_{(D+1) \times 1} \quad \underbrace{Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \cdots \\ y^{(n)} \end{pmatrix}}_{n \times 1} \Bigg\} \; n \text{ input points}$$

What cost function should we use to pick w?





$$E = -\sum_{i=1}^{n} log\ p(y^{(i)}|\vec{x}^{(i)})$$

Gaussian maximum-likelihood estimate (MLE) is achieved by minimizing mean squared error (MSE)

Parameters to learn

Weights: $\vec{w}$

Bias: b

Model

$$\widehat{y}^{(i)} \sim \mathcal{N}(\vec{w}^T \vec{x}^{(i)} + b, \sigma)$$

Gaussian maximum-likelihood estimate (MLE) is achieved by minimizing mean squared error (MSE)

$$E^{(i)} = -log\ p(y^{(i)}|\vec{x}^{(i)})$$

$$E^{(i)} = -log\ \mathcal{N}(y^{(i)} \mid \vec{w}^T\vec{x}^{(i)} + b, \sigma)$$

$$\vec{w}, b = \arg\min_{\vec{w},b} -\sum_{i=1}^{n} log\ \mathcal{N}(y^{(i)} \mid \vec{w}^T\vec{x}^{(i)} + b, \sigma)$$

$$\vec{w}, b = \arg\min_{\vec{w},b} -\sum_{i=1}^{n} log\ \frac{1}{\sqrt{2\sigma^2\pi}} \exp -\frac{(y^{(i)} - \vec{w}^T\vec{x}^{(i)} - b)^2}{\sigma^2}$$

$$\vec{w}, b = \arg\min_{\vec{w},b} \sum_{i=1}^{n} (y^{(i)} - \vec{w}^T\vec{x}^{(i)} - b)^2$$

Solving linear regression

$$\underbrace{\mathbf{X} = \begin{pmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \cdots & \cdots \\ 1 & x^{(n)} \end{pmatrix}}_{n \times (D+1)} \quad \underbrace{W = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_D \end{pmatrix}}_{(D+1) \times 1} \quad \underbrace{Y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \cdots \\ y^{(n)} \end{pmatrix}}_{n \times 1} \Bigg\} \; n \text{ input points}$$

$$\begin{aligned} \mathrm{Err}(W) &= (\mathbf{X}W - Y)^T (\mathbf{X}W - Y) \\ \nabla_{\mathbf{w}} \mathrm{Err}(W) &= \mathbf{X}^T (\mathbf{X}W - Y) + \mathbf{X}^T (\mathbf{X}W - Y) \\ 0 &= 2\mathbf{X}^T (\mathbf{X}W - Y) \\ \mathbf{X}^T Y &= \mathbf{X}^T \mathbf{X} W \\ W &= (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T Y \end{aligned}$$

Gradient Descent for linear regression

$$\vec{w}, b = \arg\min_{\vec{w},b} \sum_{i=1}^{n} (y^{(i)} - \vec{w}^T \vec{x}^{(i)} - b)^2$$

$$\frac{\partial E}{\partial \vec{w}} = \frac{2}{n} \sum_{i=1}^{n} (\vec{w}^T \vec{x}^{(i)} + b - y^{(i)}) \vec{x}^{(i)}$$

$$\frac{\partial E}{\partial \vec{w}} = \frac{2}{n} \sum_{i=1}^{n} (\widehat{y}^{(i)} - y^{(i)}) \vec{x}^{(i)}$$

$$\frac{\partial E}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\widehat{y}^{(i)} - y^{(i)})$$

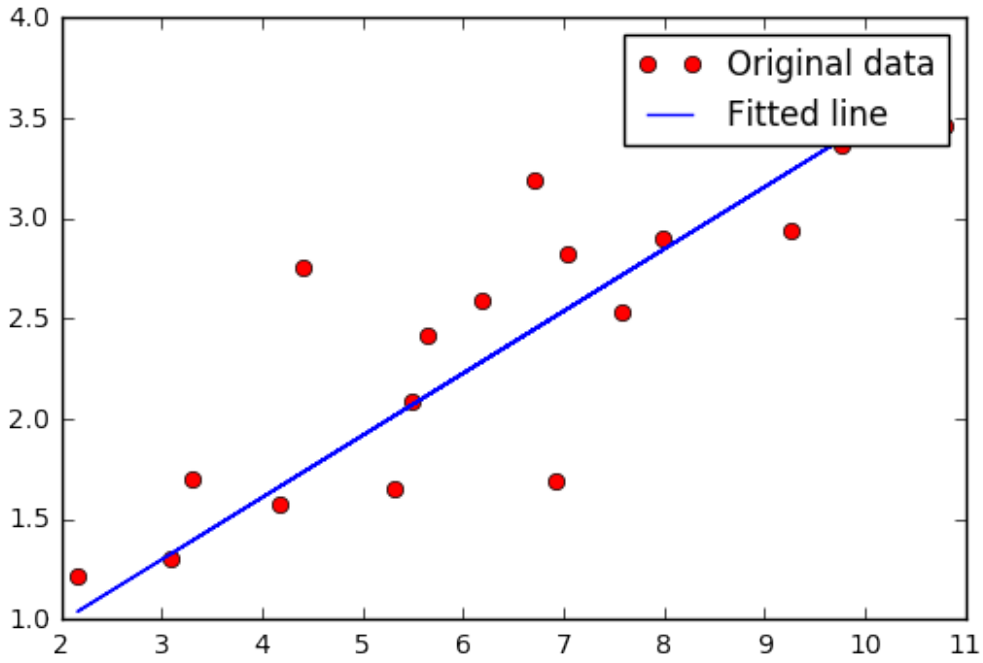
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

Linear regression can be solved by gradient descent

```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")

# Construct a linear model
pred = tf.add(tf.mul(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)

# Gradient descent
optimizer =
tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Cost is minimized after 1000 epochs

```
Epoch: 0050 cost= 0.195095107 W= 0.441748 b=
-0.580876
Epoch: 0100 cost= 0.181448311 W= 0.430319 b=
-0.498661
Epoch: 0150 cost= 0.169377610 W= 0.419571 b=
-0.421336
Epoch: 0200 cost= 0.158700854 W= 0.409461 b=
-0.348611
Epoch: 0250 cost= 0.149257123 W= 0.399953 b= -0.28021
Epoch: 0300 cost= 0.140904188 W= 0.391011 b=
-0.215878
Epoch: 0350 cost= 0.133515999 W= 0.3826 b= -0.155372
Epoch: 0400 cost= 0.126981199 W= 0.374689 b=
-0.0984639
Epoch: 0450 cost= 0.121201262 W= 0.367249 b=
-0.0449408
Epoch: 0500 cost= 0.116088994 W= 0.360252 b=
0.00539905
Epoch: 0550 cost= 0.111567356 W= 0.35367 b= 0.052745
Epoch: 0600 cost= 0.107568085 W= 0.34748 b= 0.0972751
Epoch: 0650 cost= 0.104030922 W= 0.341659 b= 0.139157
Epoch: 0700 cost= 0.100902475 W= 0.336183 b= 0.178547
Epoch: 0750 cost= 0.098135538 W= 0.331033 b= 0.215595
Epoch: 0800 cost= 0.095688373 W= 0.32619 b= 0.25044
Epoch: 0850 cost= 0.093524046 W= 0.321634 b= 0.283212
Epoch: 0900 cost= 0.091609895 W= 0.317349 b= 0.314035
Epoch: 0950 cost= 0.089917004 W= 0.31332 b= 0.343025
Epoch: 1000 cost= 0.088419855 W= 0.30953 b= 0.370291
Optimization Finished!
Training cost= 0.0884199 W= 0.30953 b= 0.370291
```



We will always approximate the best solution when the cost is convex

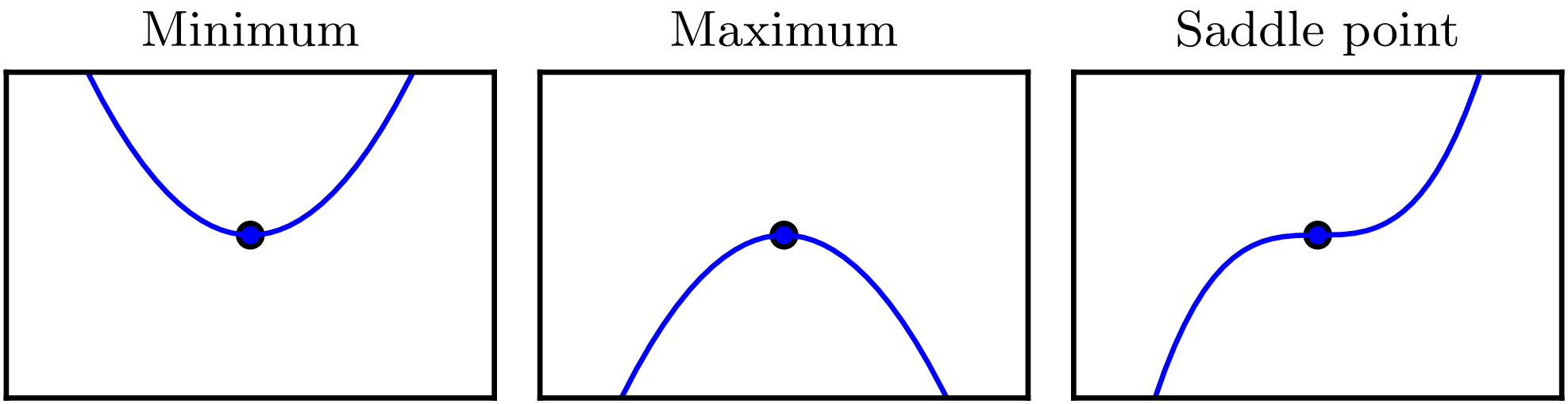


Figure 4.2

Example convex functions

- $c(x) = Mx + b$
- $c(x) = e^{c1(x)}$
- $c(x) = c1(x) + c2(x)$
- $c(x) = x^p$
- $c(x) = |x|$
- $c(x) = x \log x$
- $c(x) = \max(c1(x), c2(x))$
- Sigmoid functions are not convex
  - But we can still use gradient descent to solve them

Approximate Optimization

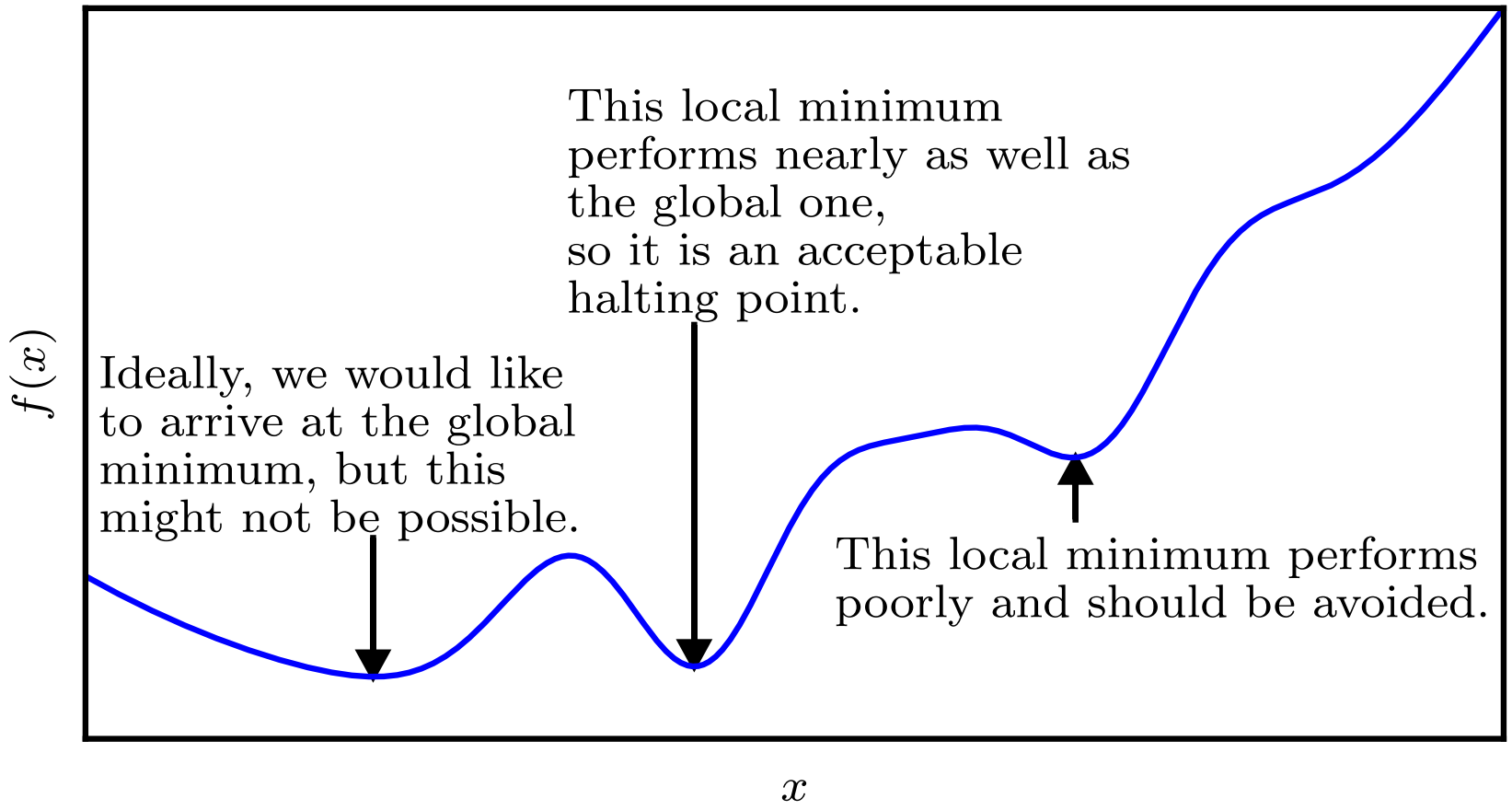


Figure 4.3

Curvature

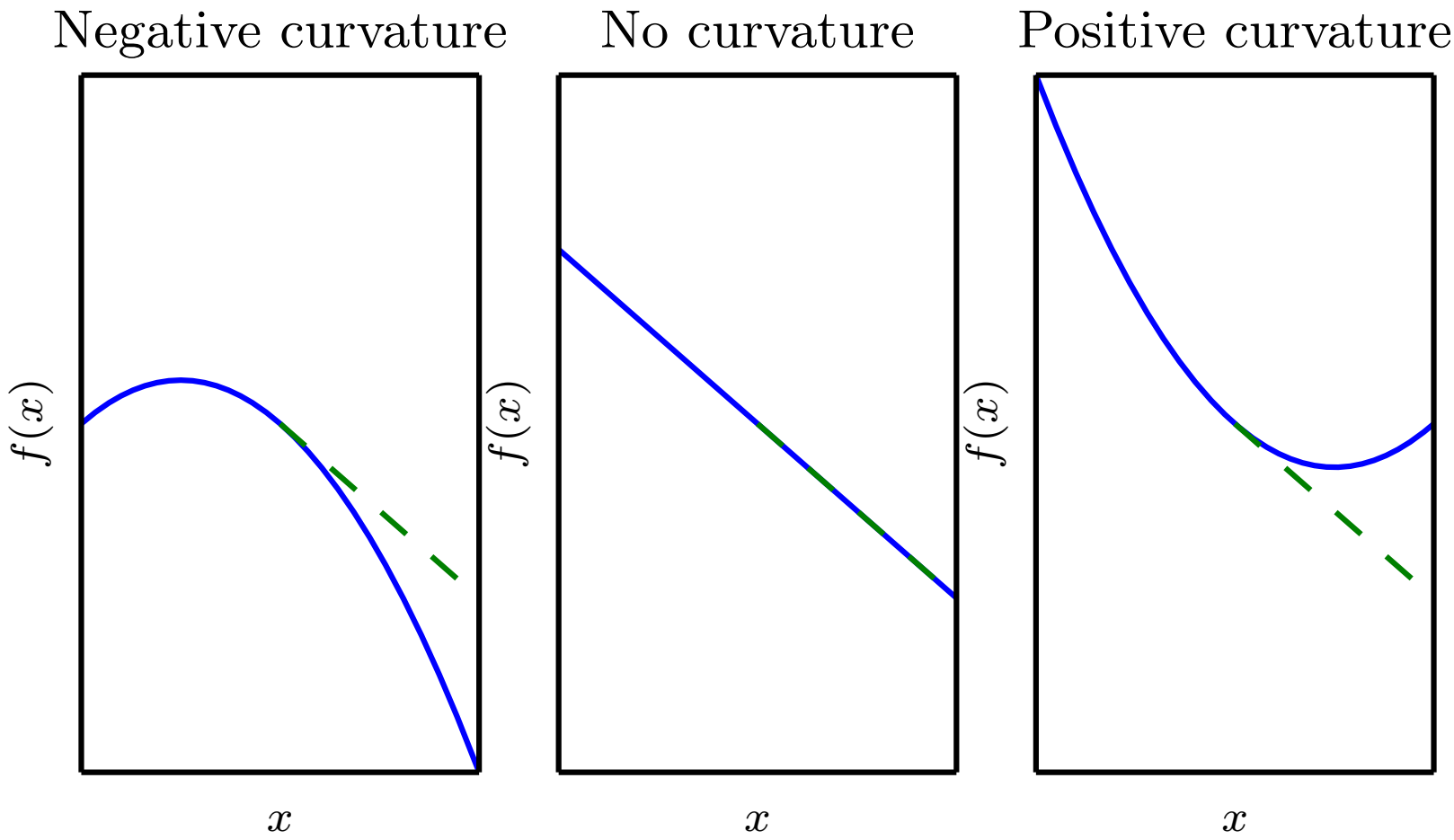


Figure 4.4

Gradient Descent and Poor Conditioning

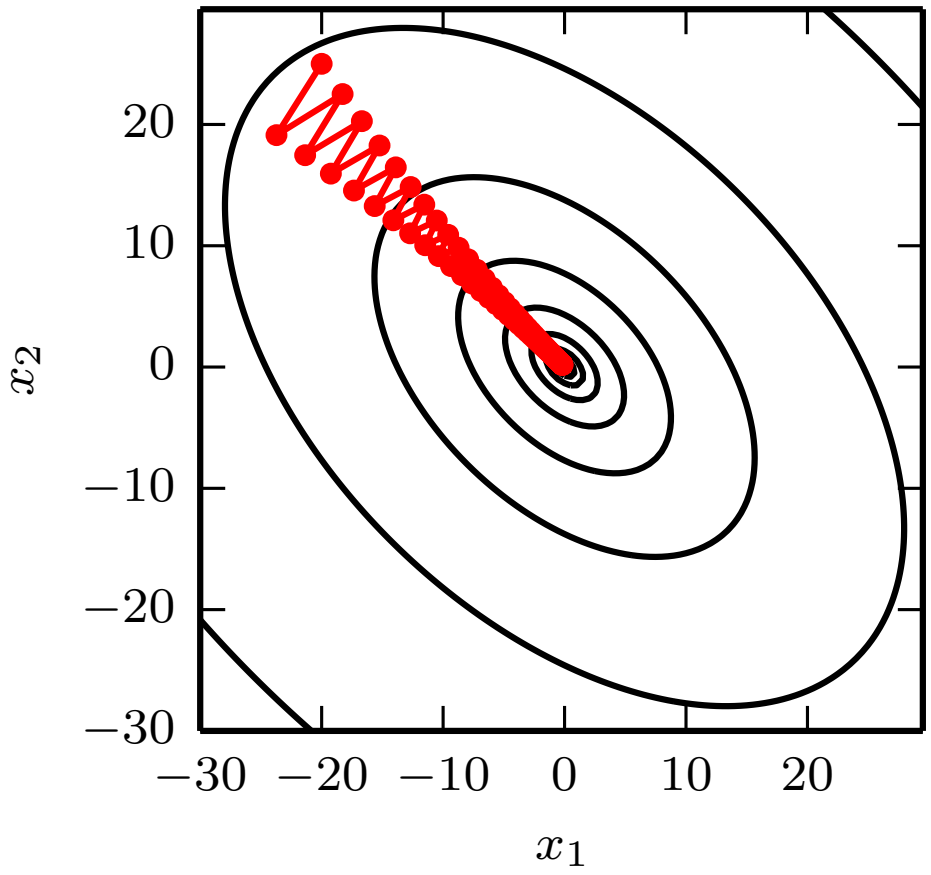


Figure 4.6

Solving non linear feedforward networks (Part II)

Why not just stacks of linear layers to juice up our function space?

- $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$
- Stacks of linear layers are identical to a single layer with their weight matrices combined

XOR can be solved by a non-linear network (Section 6.1)

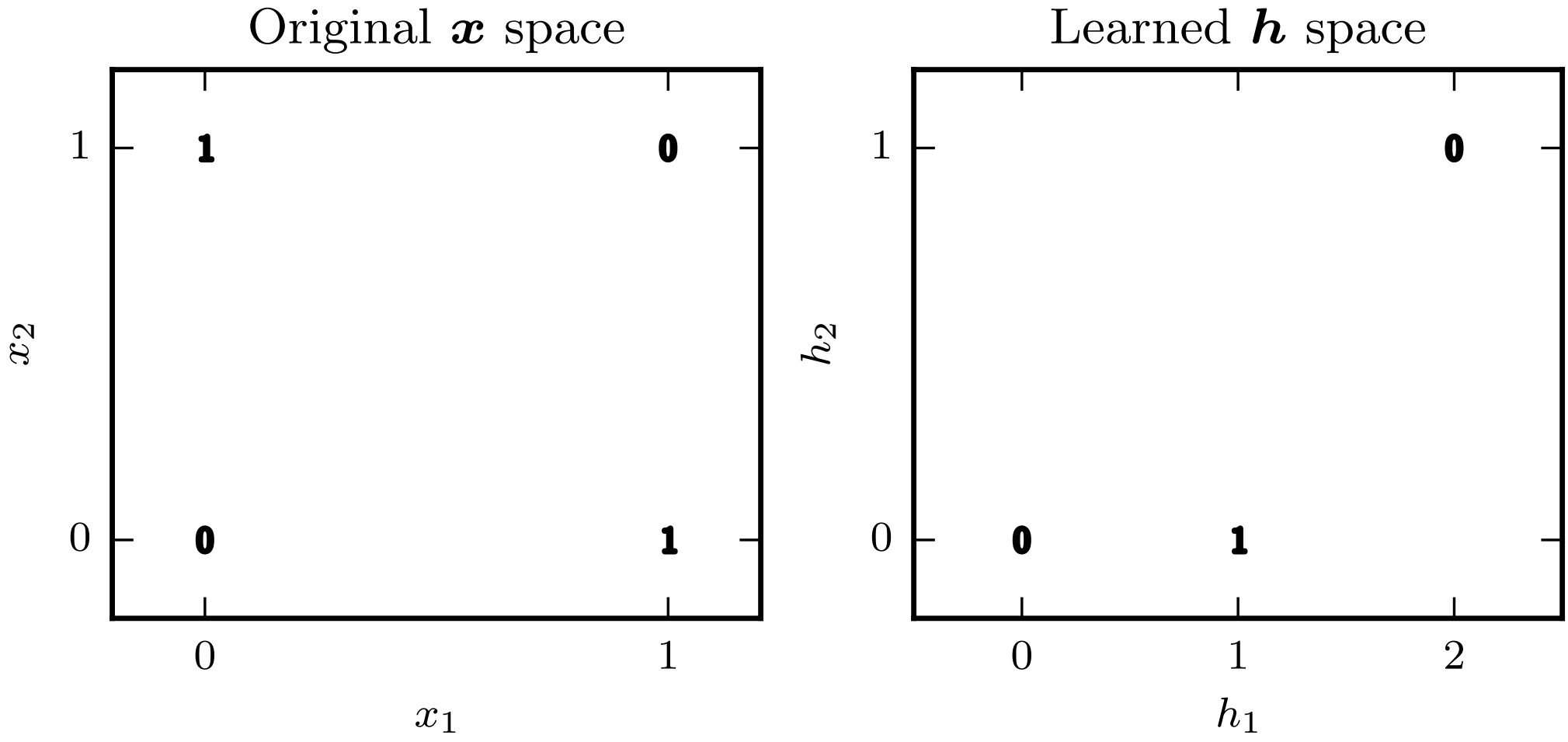


Figure 6.1

Logistic regression is a non-linear network

Features: $\vec{x}^{(i)}$

Labels: $y^{(i)} \in \{1, 0\}$

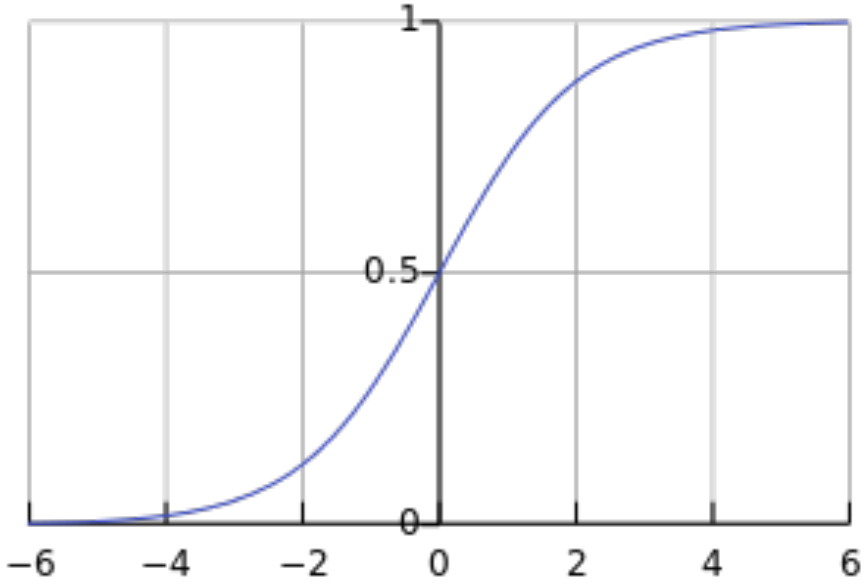
Weights: $\vec{w}$

Bias: b

$$z^{(i)} = \vec{w}^T \vec{x}^{(i)} + b$$

$$p^{(i)} = p(y^{(i)} = 1 | \vec{x}^{(i)}) = \sigma(z^{(i)})$$

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$



Our cost function for logistic regression

$$p(y^{(i)}|\vec{x}^{(i)}) = (p^{(i)})^{y^{(i)}}(1-p^{(i)})^{(1-y^{(i)})}$$

$$p(y|\vec{x}) = \prod_{i=1}^{n} p(y^{(i)}|\vec{x}^{(i)})$$

$$E^{(i)} = -(y^i log\ p^{(i)} + (1-y^i) log(1-p^{(i)}))$$

E = .314

$$E^{(i)} = -(y^i log\ p^{(i)} + (1 - y^i) log(1 - p^{(i)}))$$

p = .73 y = 1

$$p^{(i)} = \sigma(z^{(i)})$$

z = 1

$$z^{(i)} = \vec{w}^T \vec{x}^{(i)} + b$$

w = [.3]
x = [1]
b = .7

$$\frac{\partial E^{(i)}}{\partial p^{(i)}} = -\frac{y^{(i)}}{p^{(i)}} + \frac{(1 - y^{(i)})}{(1 - p^{(i)})}$$

- 1/.73 = -1.36

$$\frac{\partial p^{(i)}}{\partial z^{(i)}} = p^{(i)}(1 - p^{(i)})$$ = .197

-1.36 * .197 = -0.268

$$\frac{\partial z^{(i)}}{\partial \vec{w}} = \vec{x}^{(i)}$$ = 1

-0.268 * 1 = -0.268

Logistic regression gradients simplify

$$\frac{\partial E^{(i)}}{\partial \vec{w}} = \frac{\partial E^{(i)}}{\partial p^{(i)}} \frac{\partial p^{(i)}}{\partial z^{(i)}} \frac{\partial z^{(i)}}{\partial \vec{w}}$$

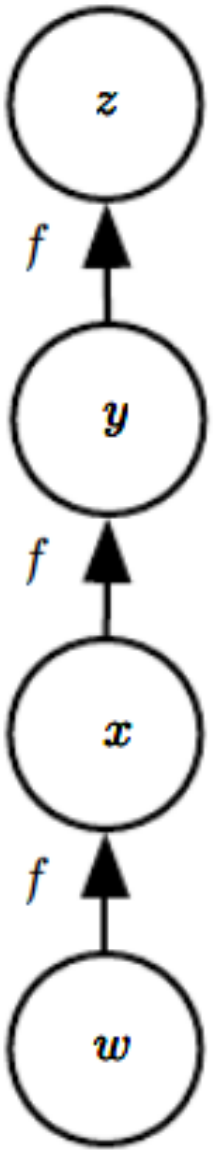
$$\frac{\partial E^{(i)}}{\partial p^{(i)}} = -\frac{y^{(i)}}{p^{(i)}} + \frac{(1 - y^{(i)})}{(1 - p^{(i)})}$$

$$\frac{\partial p^{(i)}}{\partial z^{(i)}} = p^{(i)}(1 - p^{(i)}) = \sigma(z^{(i)})(1 - \sigma(z^{(i)}))$$

$$\frac{\partial z^{(i)}}{\partial \vec{w}} = \vec{x}^{(i)}$$

$$\frac{\partial E^{(i)}}{\partial \vec{w}} = (p^{(i)} - y^{(i)})\vec{x}^{(i)}$$

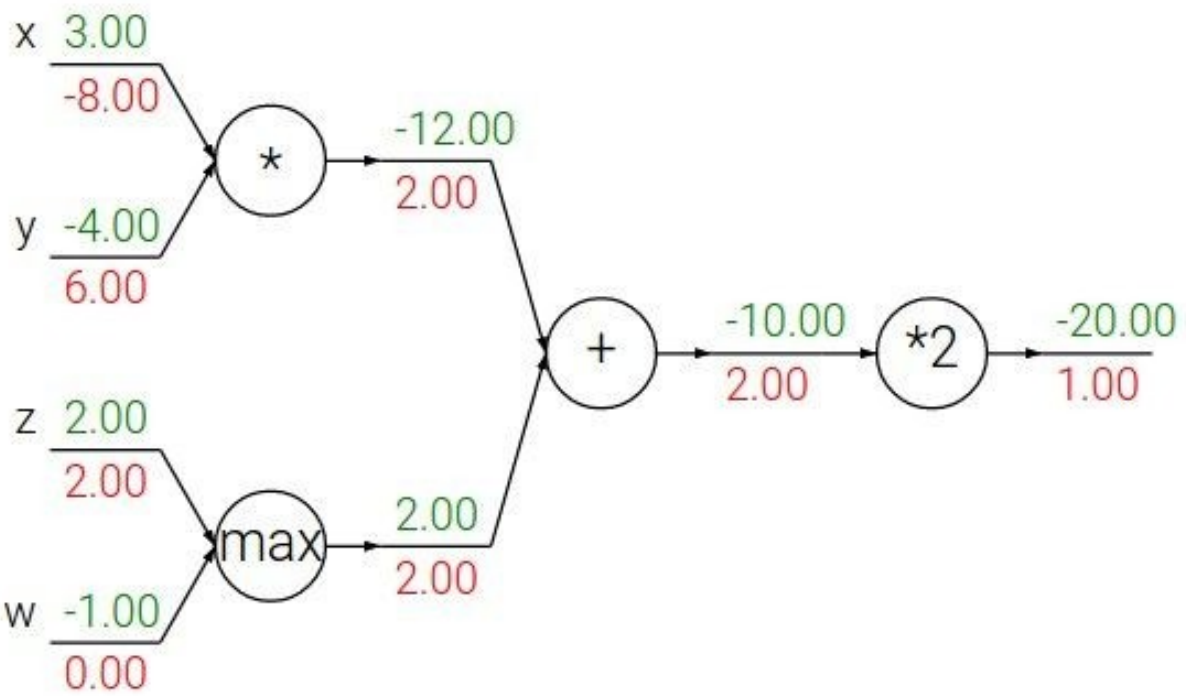
Backpropogation computes gradients via the chain rule



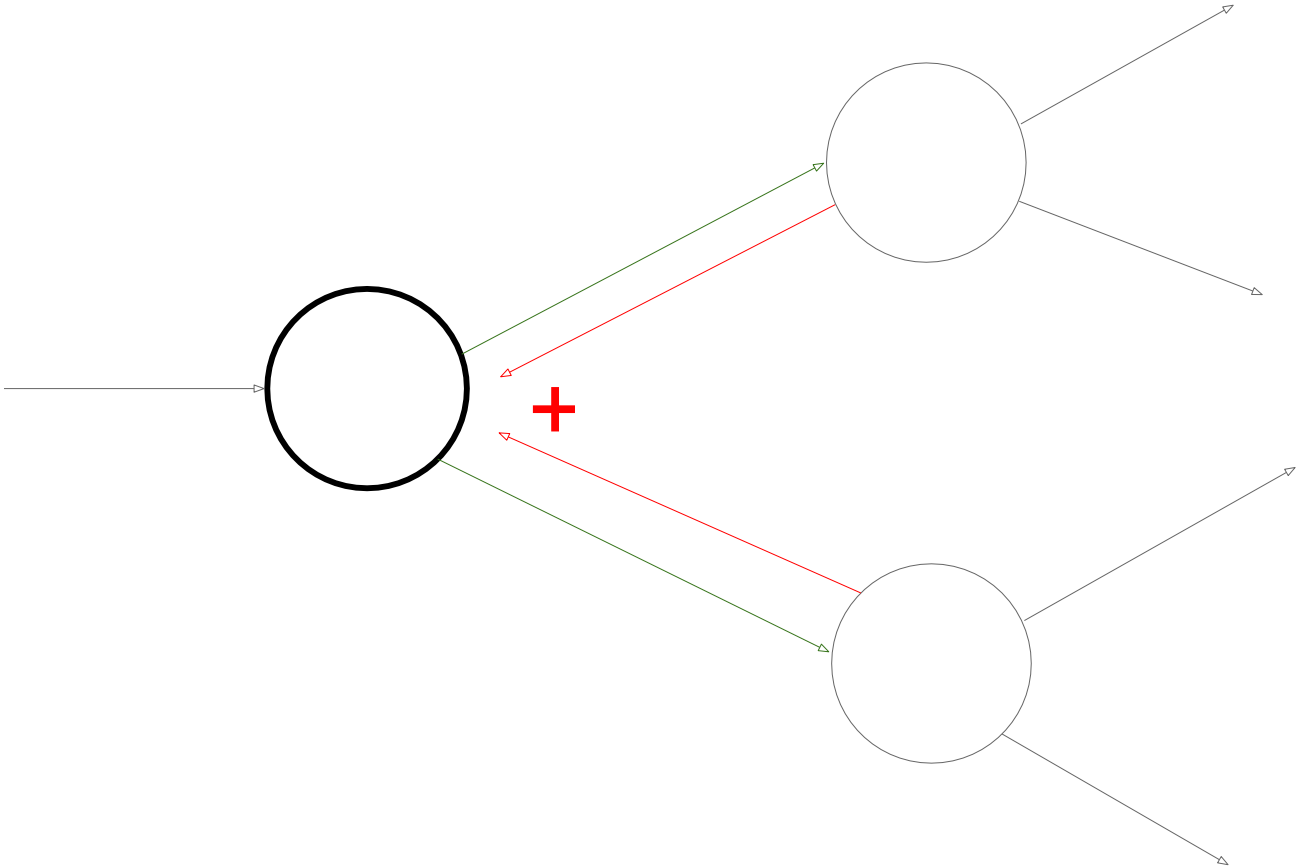
$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

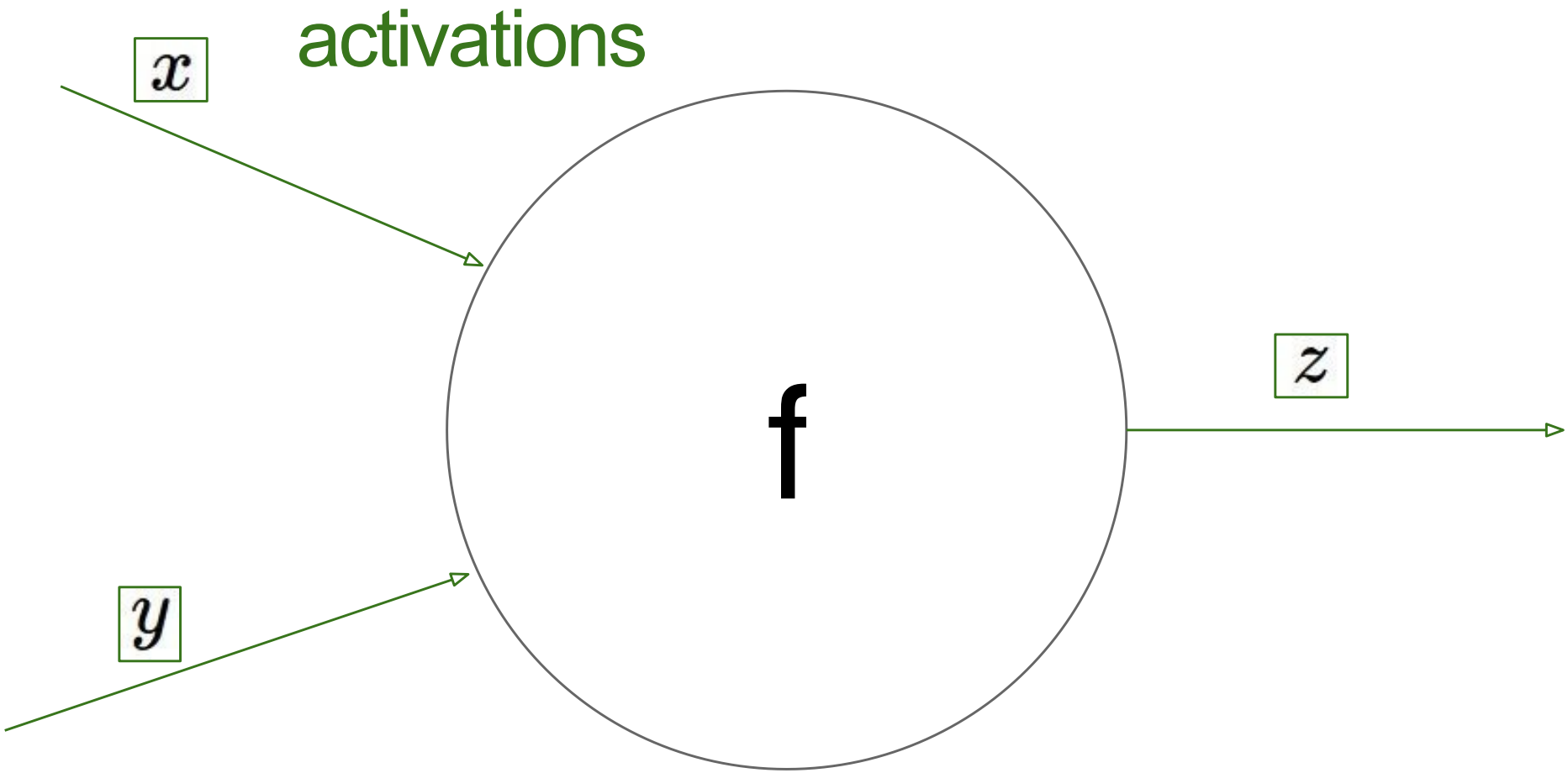
Patterns in backward flow

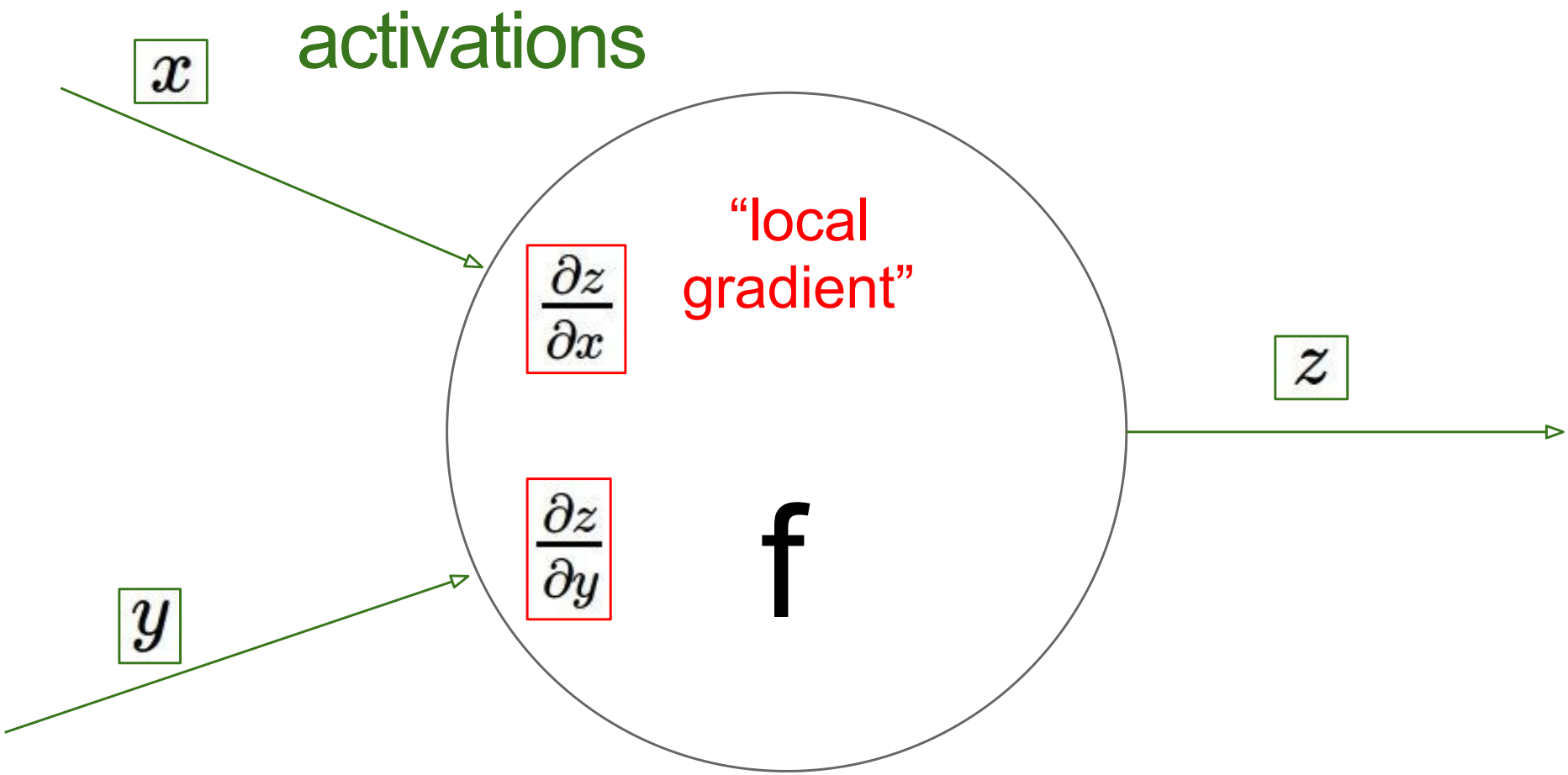
add gate: gradient distributor
max gate: gradient router
mul gate: gradient… "switcher"?

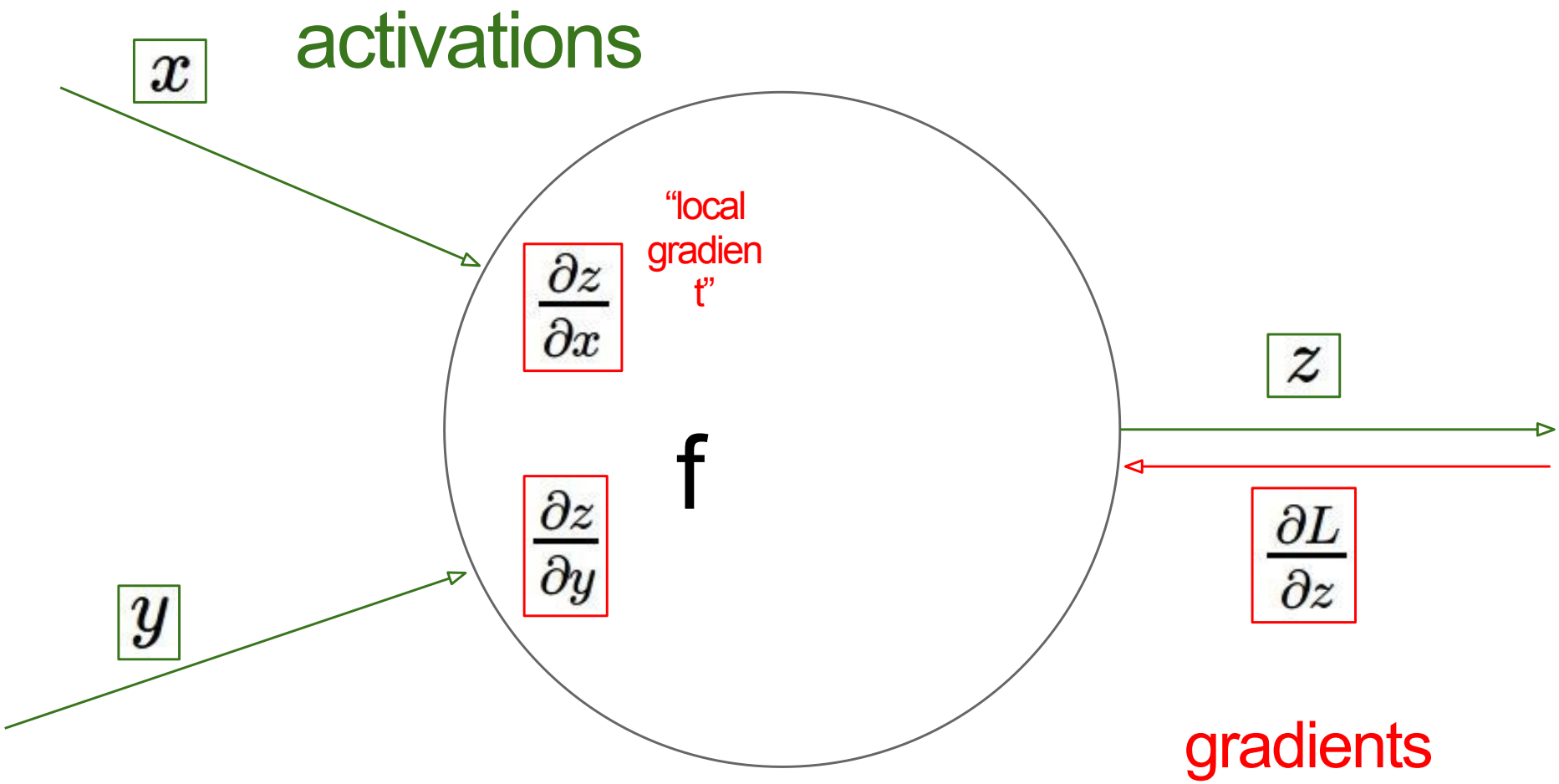


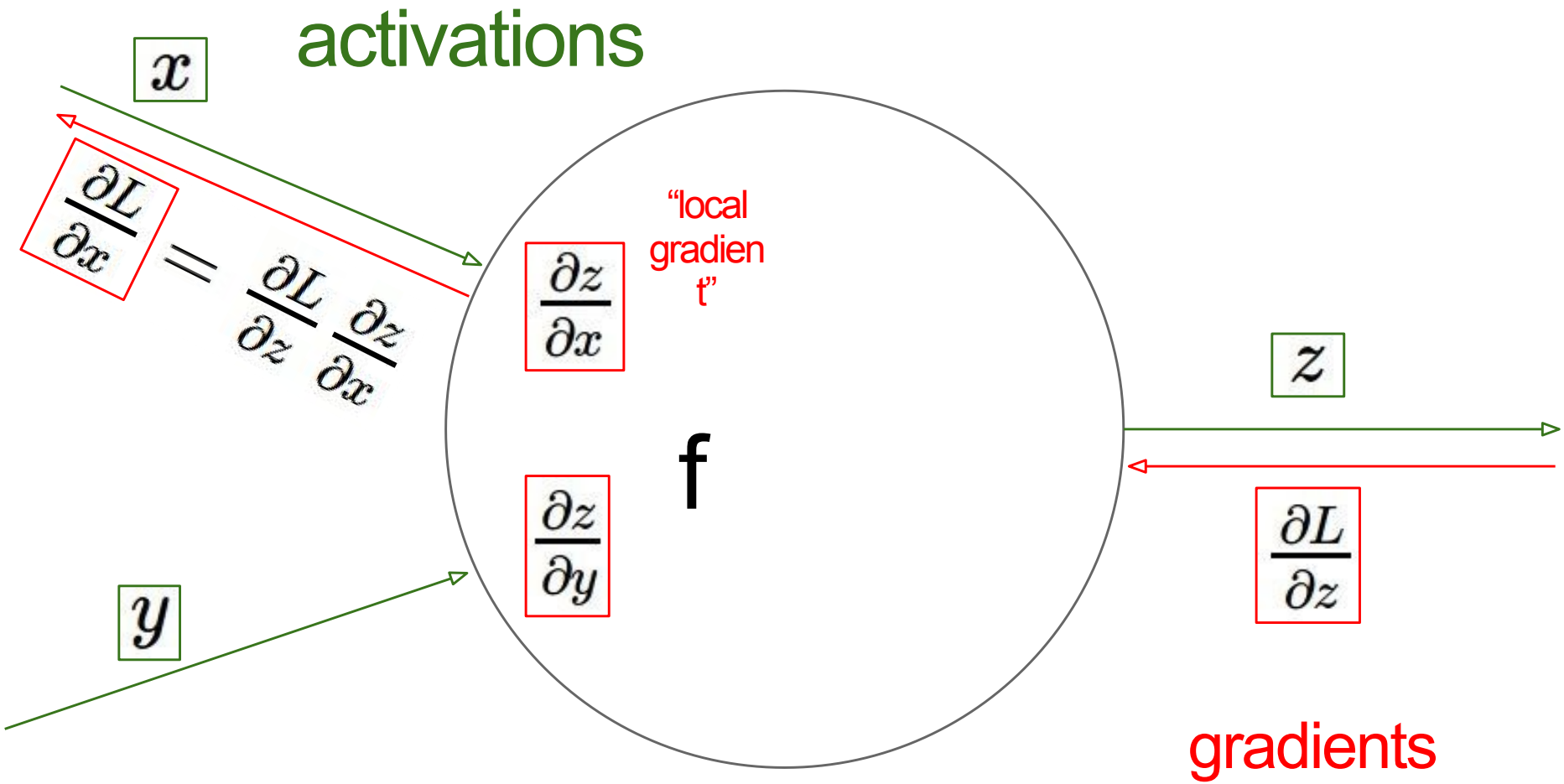
Gradients add at branches

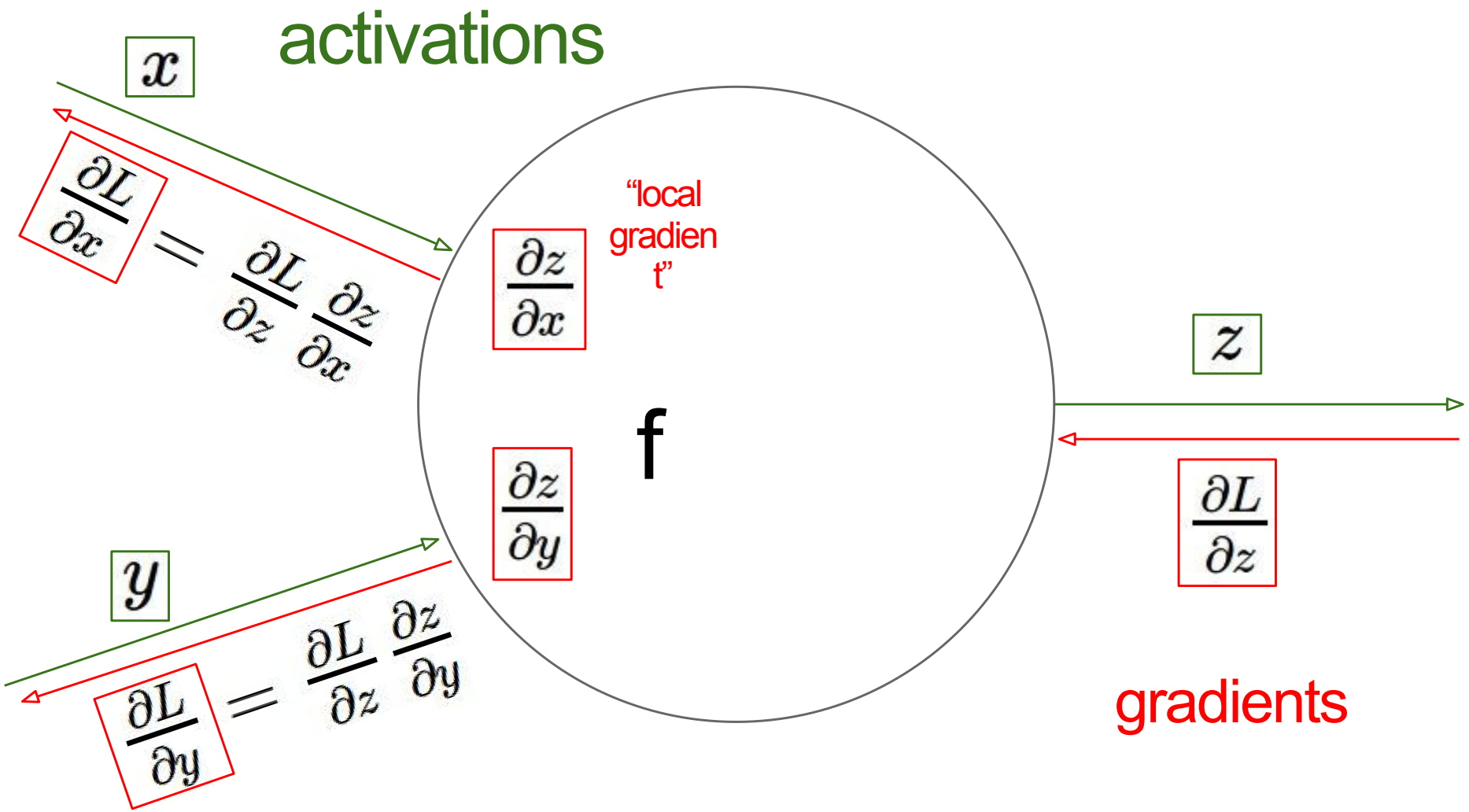


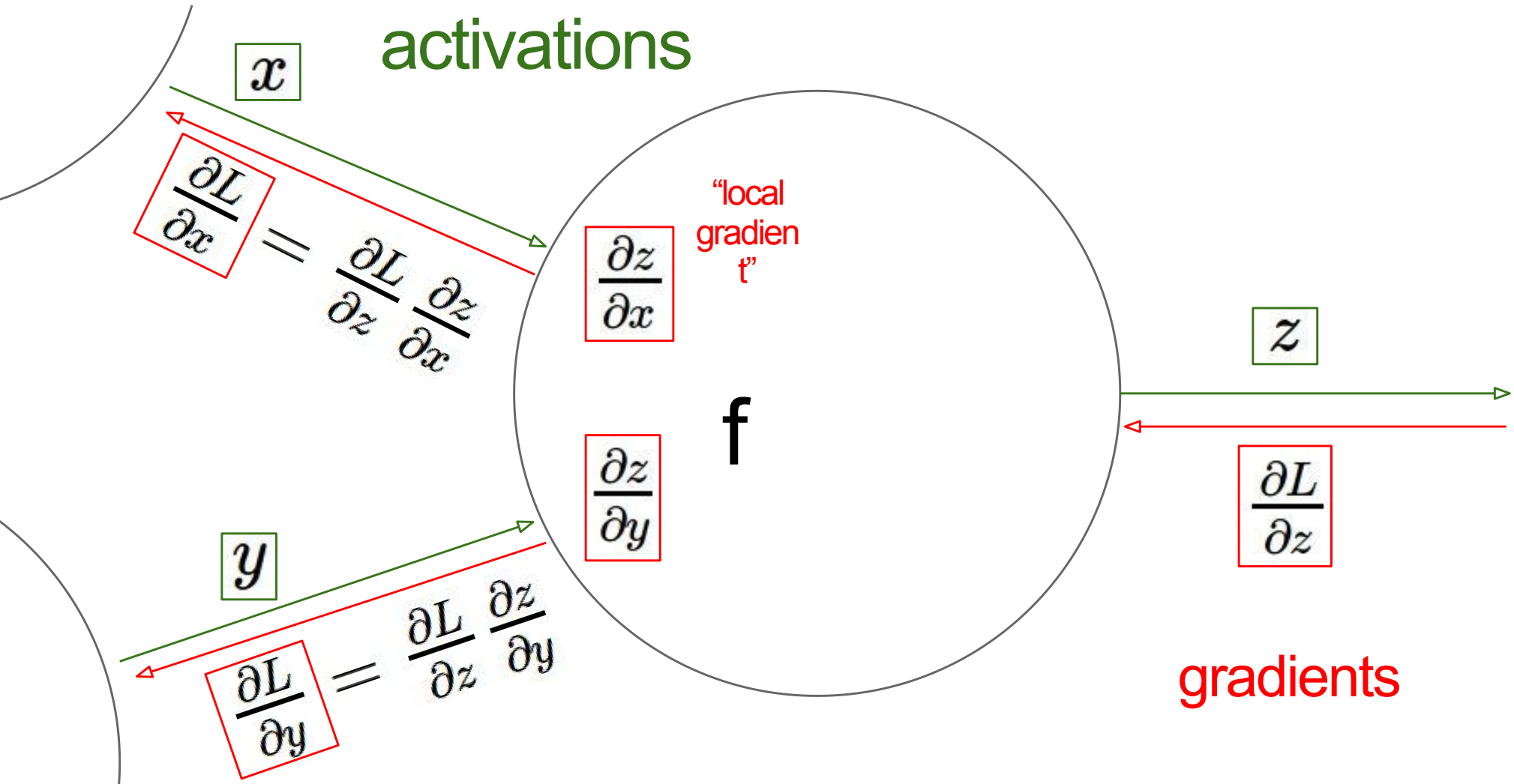

activations
x
y
f
z


activations
x
y
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
"local gradient"
f
z


activations
x
y
$\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial y}$
f
z
$\frac{\partial L}{\partial z}$
gradients


activations
x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
y
z
$\frac{\partial L}{\partial z}$
gradients


activations
x
y
z
f
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients


activations
x
y
z
f
"local gradient"
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

FIN - Thank You