Lecture 12-13: Basic Neural Nets
Deep Feedforward Networks

CS 109B, STAT 121B, AC 209B, CSE 109B

Mark Glickman and Pavlos Protopapas

Beyond Linear Models

- Linear models
  - Can be fit efficiently (via convex optimization)
  - Limited model capacity
- Alternative:

$$f(x) = w^T \phi(x)$$

where ϕ is a *non-linear transform*

Traditional ML

- Manually engineer ϕ
  - Domain specific, enormous human effort
- Generic transform
  - Maps to a higher-dimensional space
  - Kernel methods: e.g. RBF kernels
  - Over fitting: does not generalize well to test set
  - Cannot encode enough prior information

Deep Learning

- Directly learn ϕ

$$f(x;\theta) = w^T \phi(x;\theta)$$

where θ are parameters of the transform

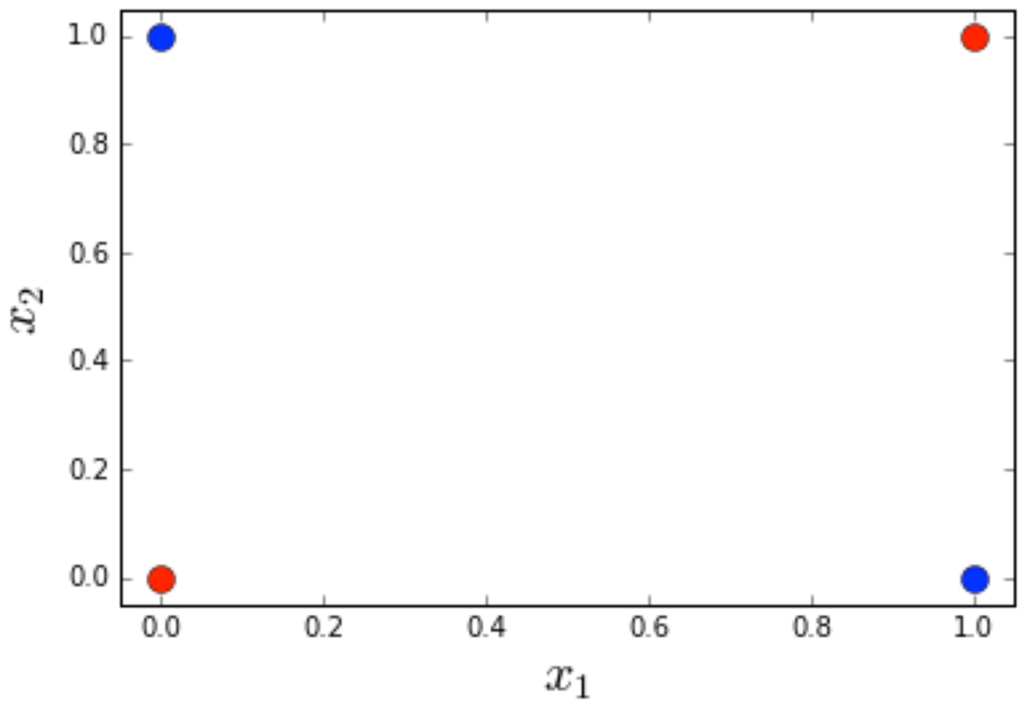
- ϕ defines hidden layers
- Non-convex optimization
- Can encode prior beliefs, generalizes well

SVM vs Neural Networks

- Hand-written digit recognition: MNIST data

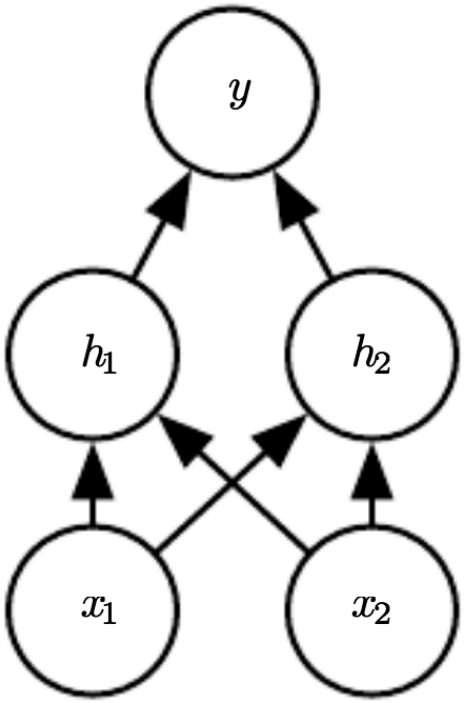
See illustration in notebook

Example: Learning XOR



- Optimal linear model (sq. loss)
  - Predicts 0.5 on all points

Example: Learning XOR



$$h_1 = \sigma(w_1^T x + c_1)$$

$$h_2 = \sigma(w_2^T x + c_2)$$

$$y = \sigma(w^T h + b)$$

where,

$$\sigma(z) = \max\{0, z\}$$

See illustration in notebook

Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

Cost Function

- Cross-entropy between training data and model distribution (i.e. negative log-likelihood)

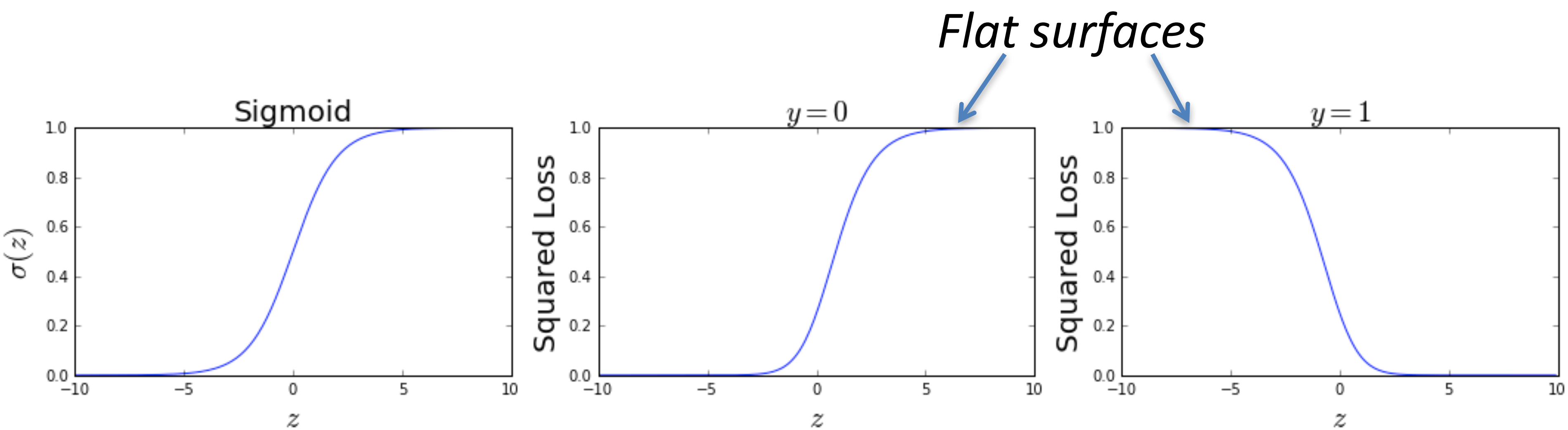
$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

- Do not need to design separate cost functions
- Gradient of cost function must be large enough

Cost Function

- Example: sigmoid output + squared loss

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad L_{sq}(y,z) = (y - \sigma(z))^2$$

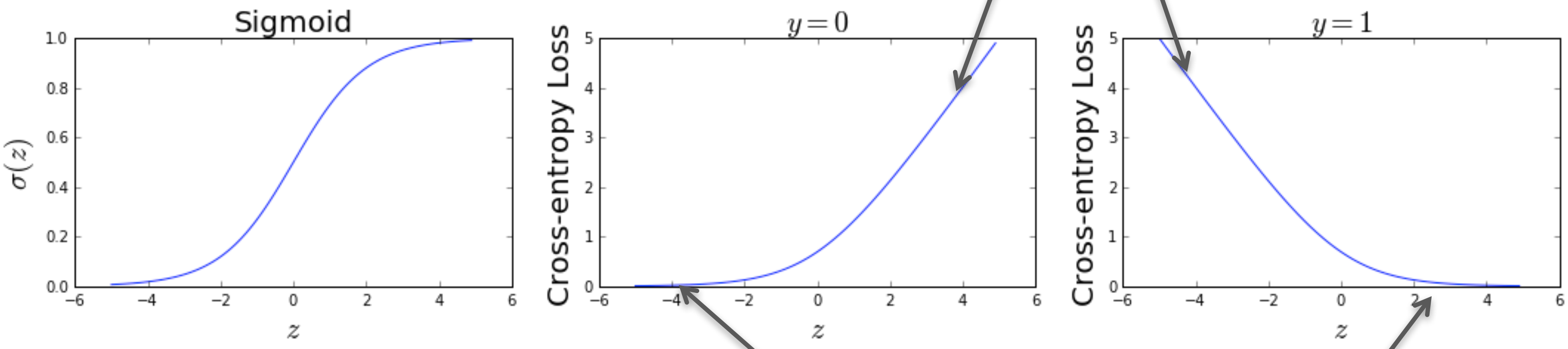


Cost Function

- Example: sigmoid output + cross-entropy loss

$$L_{ce}(y,z) = -(y\log(z) + (1-y)\log(1-z))$$

log undoes exp



Saturates only when the model makes correct predictions

Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

Output Units

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

(Goodfellow 2017)

Softmax Output

- Discrete / Multinoulli output distribution
- For output scores $z_1, \ldots, z_n$

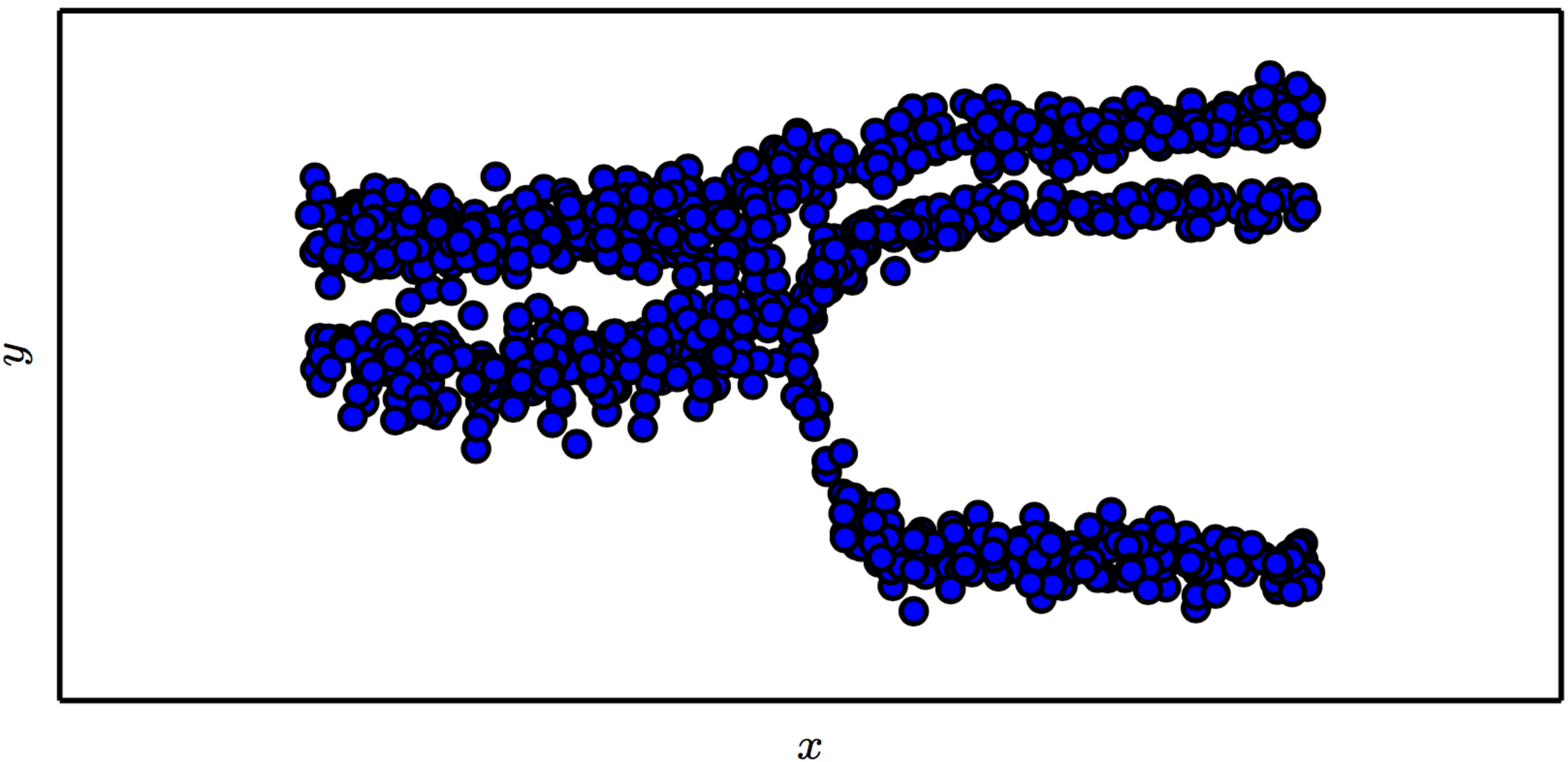
$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Log-likelihood undoes exp

$$\log \text{softmax}(z)_i = z_i - \log \sum_j \exp(z_j)$$
$$\approx z_i - \max_j z_j$$

(Score to target label – Maximum score)

Mixture Density Output



(Goodfellow 2017)

Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

Hidden Units

$$\mathbf{h} = g(\mathbf{W}^T x + \mathbf{b})$$

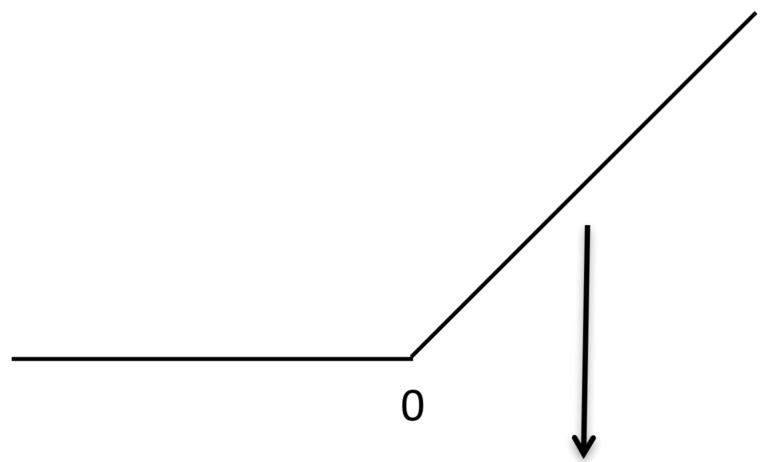
with activation function g

- Ensure gradients remain large through hidden unit
- Preferred: piece-wise linear activation
- Avoid sigmoid/tanh activation
  - Do not provide useful gradient info when they saturate

ReLU

- Rectified Linear Units

$$g(z) = \max\{0, z\}$$



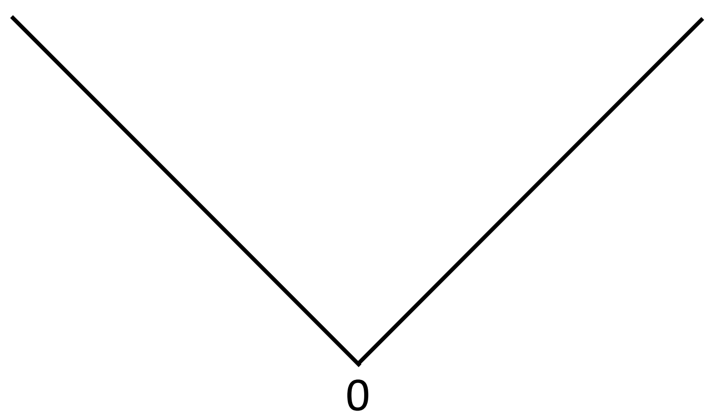
- Gradient is 1 whenever unit is active
  - More useful for learning compared to sigmoid
  - No useful gradient information when z<0

Generalized ReLU

- Generalization: For $\alpha_i > 0$,

$$g(z;\alpha)_i = \max\{0, z_i\} + \alpha_i \min\{0, z_i\}$$

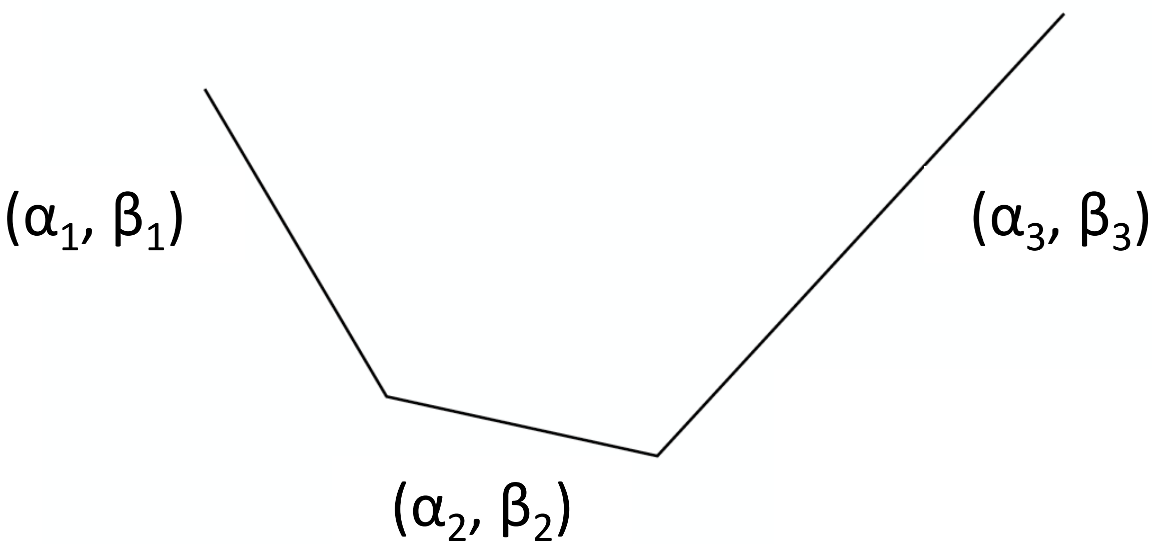
- E.g. Absolute value ReLU: $\alpha_i = -1 \;\Rightarrow\; g(z) = |z|$



Maxout

- Directly learn the activation function
  - Max of k linear functions

$$g(z) = \max_{i \in \{1,\ldots,k\}} \alpha_i z_i + \beta_i$$

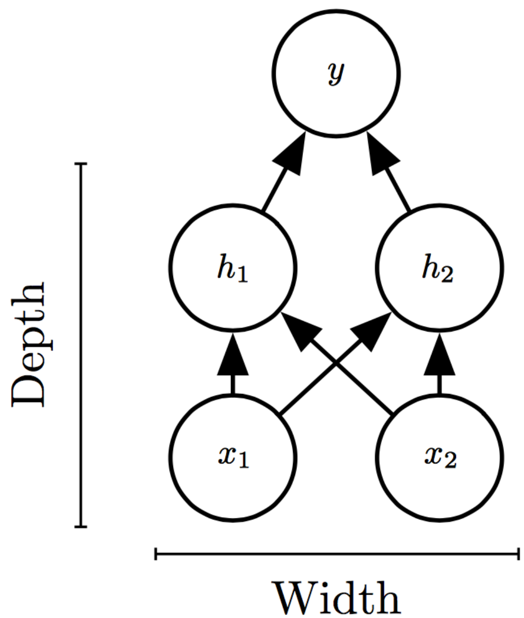


Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

Universal Approximation Theorem

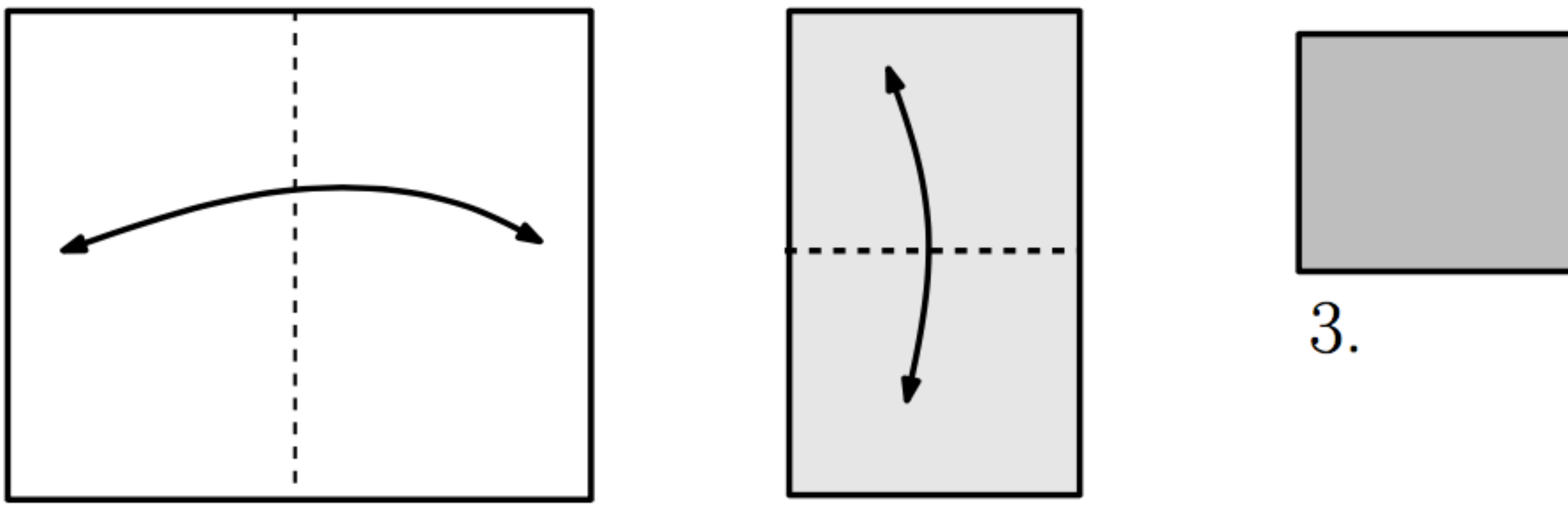
- One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy
- So why deeper?
  - Shallow net may need (exponentially) more width
  - Shallow net may overfit more



Exponential Gain with Depth

- Each hidden layer folds the space of activations of the previous layer. E.g. abs activation $g(z) = |z|$

Montúfar (2014)

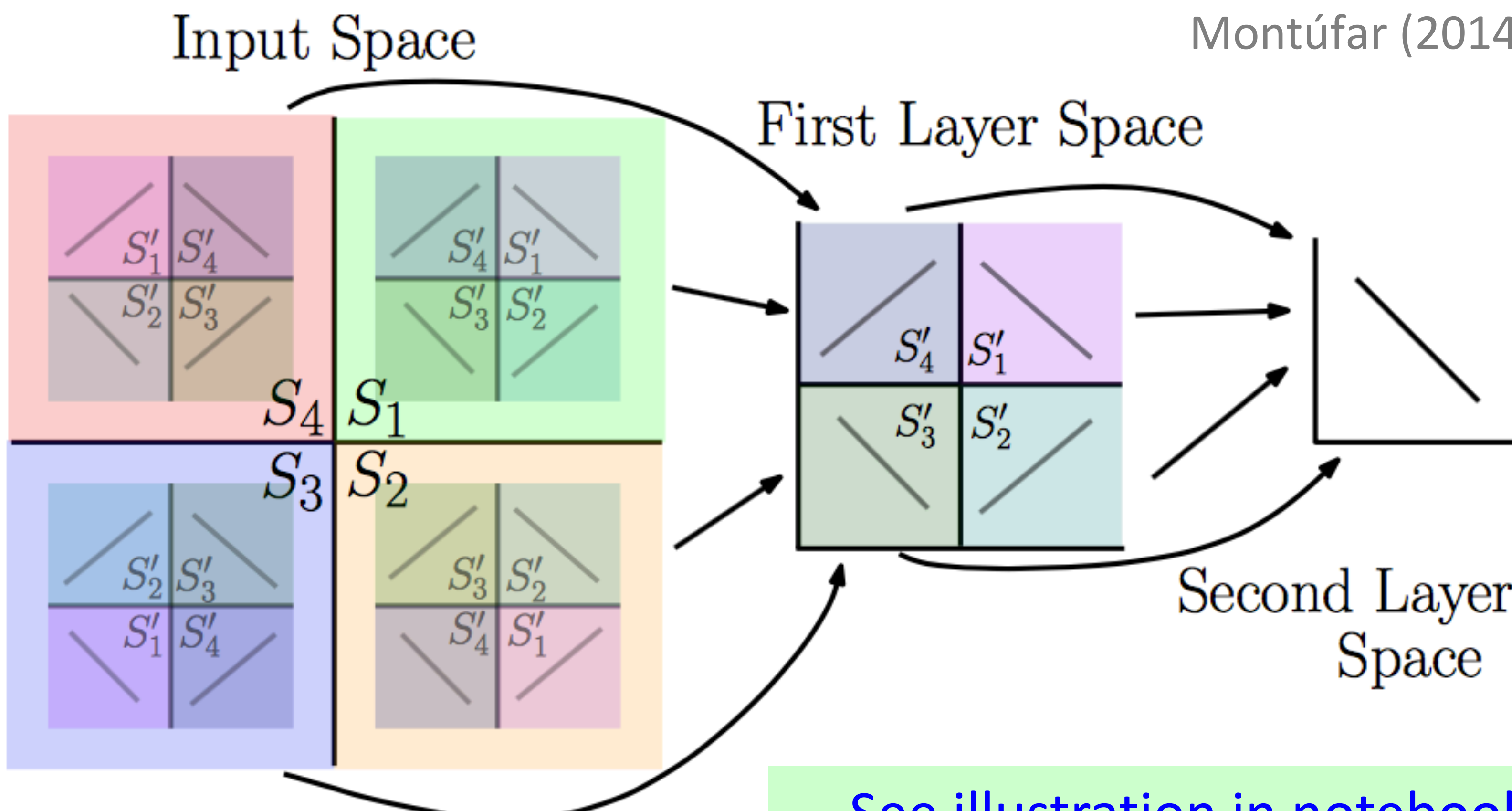


1. Fold along the vertical axis
2. Fold along the horizontal axis
3.

Exponential Gain with Depth

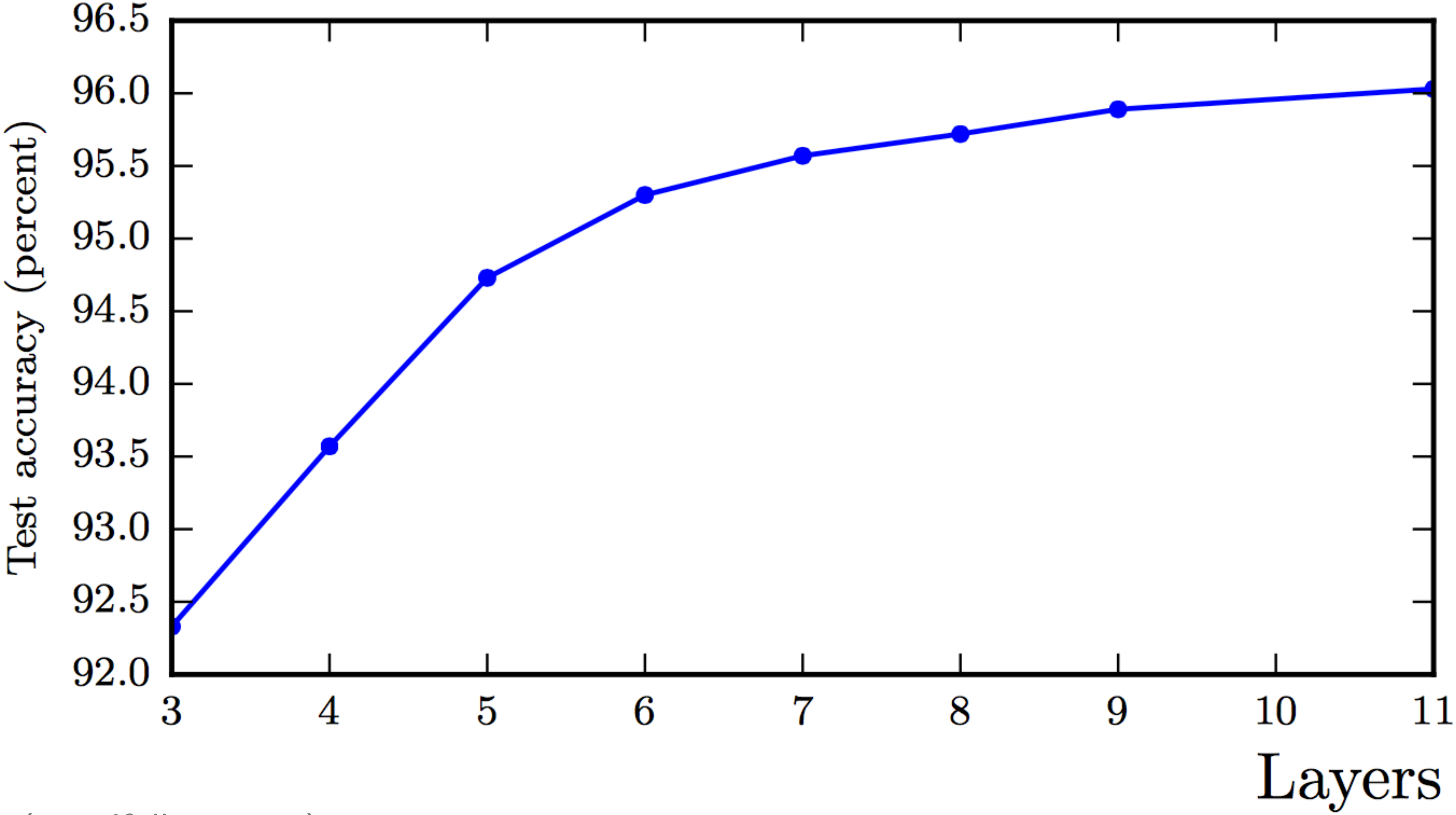
- With N hidden layers, there are $O(4^N)$ piecewise linear regions

Montúfar (2014)



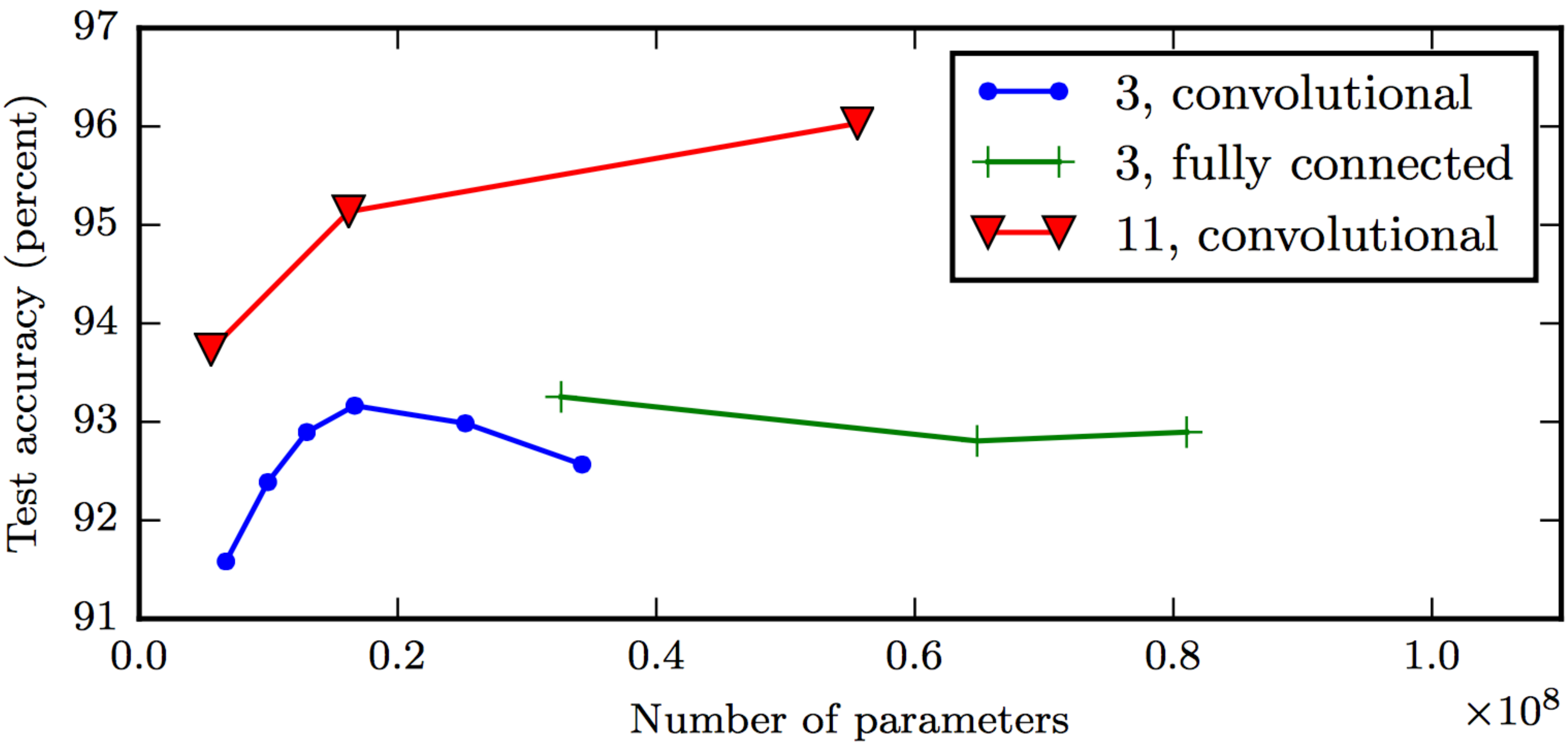
See illustration in notebook

Better Generalization with Depth



(Goodfellow 2017)

Large, Shallow Nets Overfit More



(Goodfellow 2017)

Design Choices

- Cost function
- Output units
- Hidden units
- Architecture
- Optimizer

Gradient-based Optimizer
(e.g. stochastic gradient descent)

- "Chain rule" for computing gradients:

$$\mathbf{y} = g(\mathbf{x}) \qquad z = f(\mathbf{y})$$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

- For deeper networks

Naïve computation takes exponential time

$$\frac{\partial z}{\partial x_i} = \sum_{j_1} \cdots \sum_{j_m} \frac{\partial z}{\partial y_{j_1}} \cdots \frac{\partial y_{j_m}}{\partial x_i}$$

Backpropagation

- Avoids repeated sub-expressions
- Uses dynamic programming (table filling)
- Trades-off memory for speed

Backprop: Arithmetic

- Jacobian-gradient products

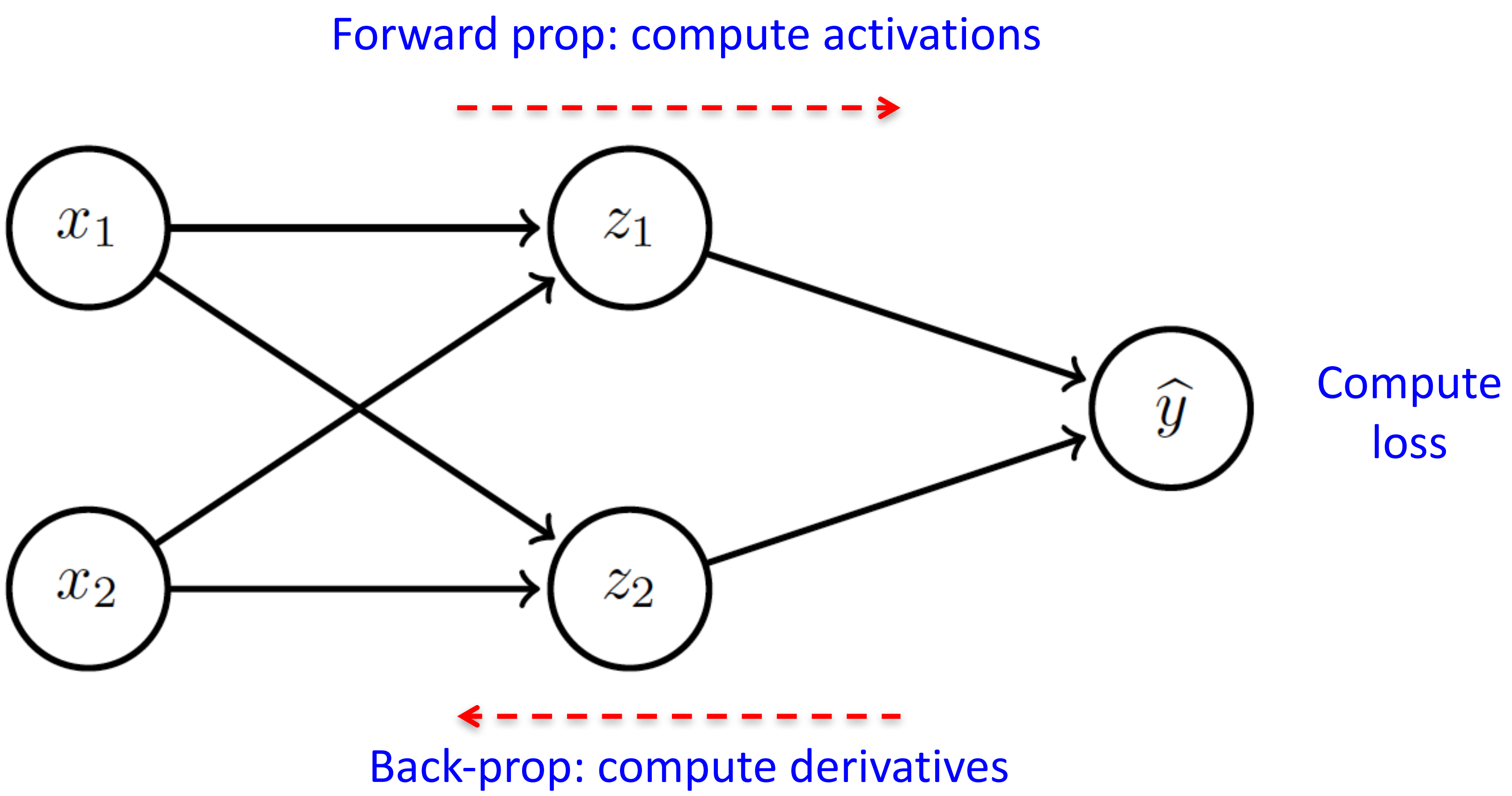
$$\mathbf{z} = g(\mathbf{x})$$
$$y = f(\mathbf{z})$$

$$\underbrace{\begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix}}_{\text{grad w.r.t. } \mathbf{x}} = \underbrace{\begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial x_n} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix}}_{\text{Jacobian of 'g'}} \times \underbrace{\begin{bmatrix} \frac{\partial y}{\partial z_1} \\ \vdots \\ \frac{\partial y}{\partial z_m} \end{bmatrix}}_{\text{grad w.r.t. } \mathbf{z}}$$

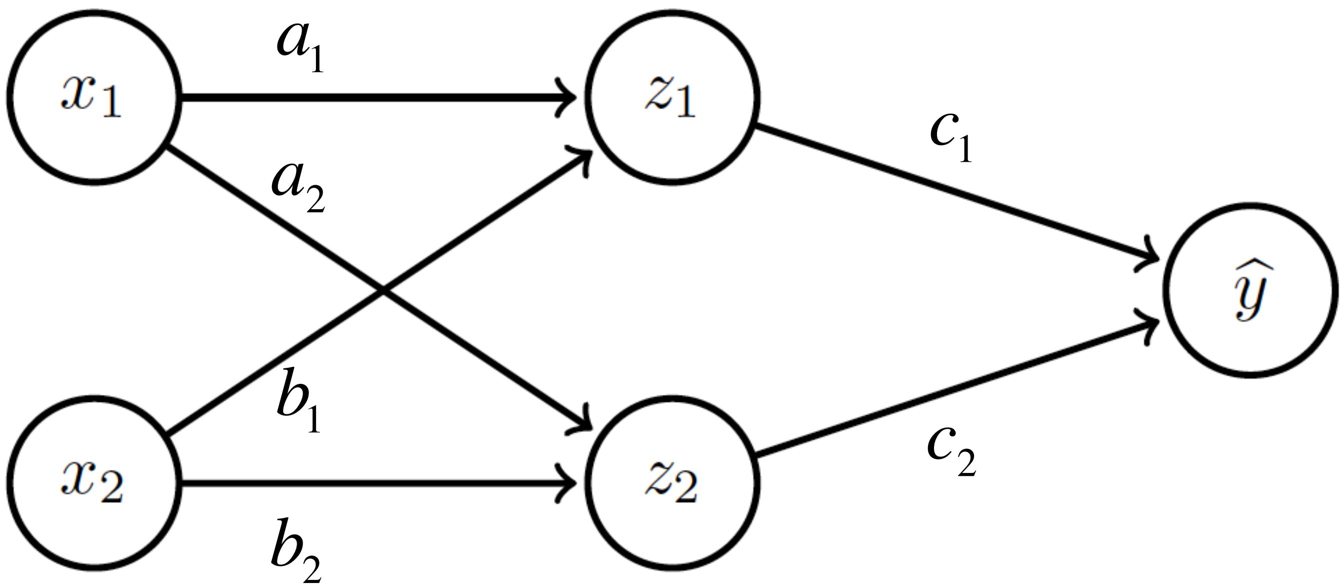
$$\nabla_{\mathbf{x}} y = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \right)^T \nabla_{\mathbf{z}} y$$

Apply recursively!

Backprop: Overview



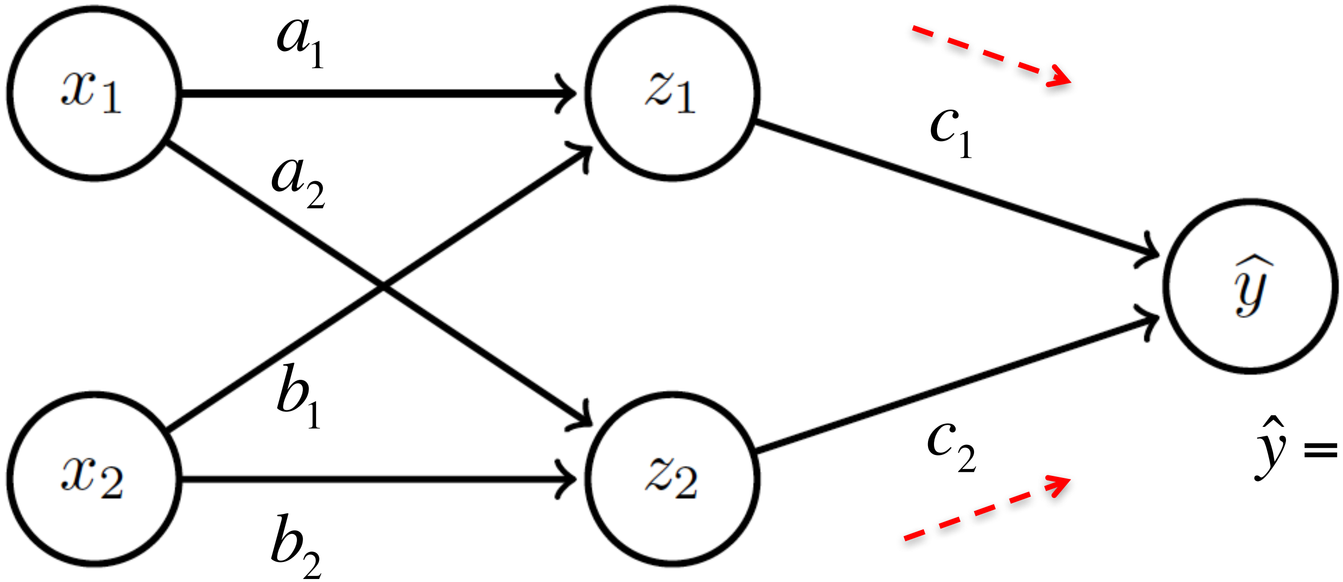
Backprop: Example



Linear activation functions
No bias
Squared loss

Backprop: Example

$$z_1 = a_1 x_1 + a_2 x_2$$

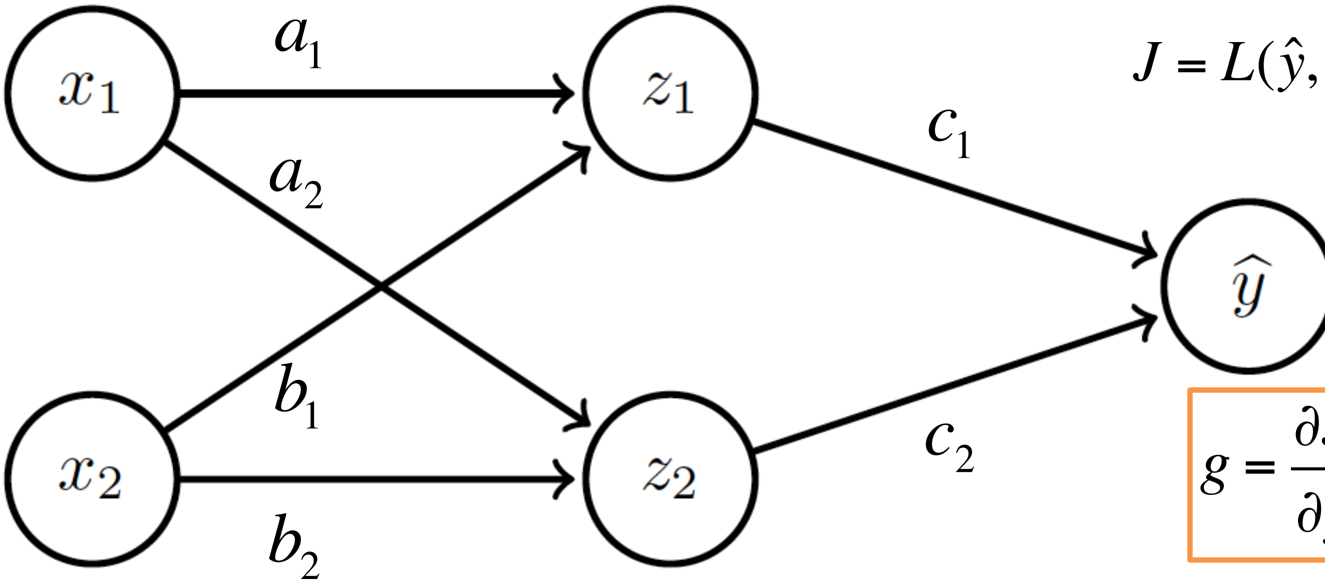


$$\hat{y} = c_1 z_1 + c_2 z_2$$

$$z_2 = b_1 x_1 + b_2 x_2$$

Forward prop: Propagate activations to output layer

Backprop: Example



$J = L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

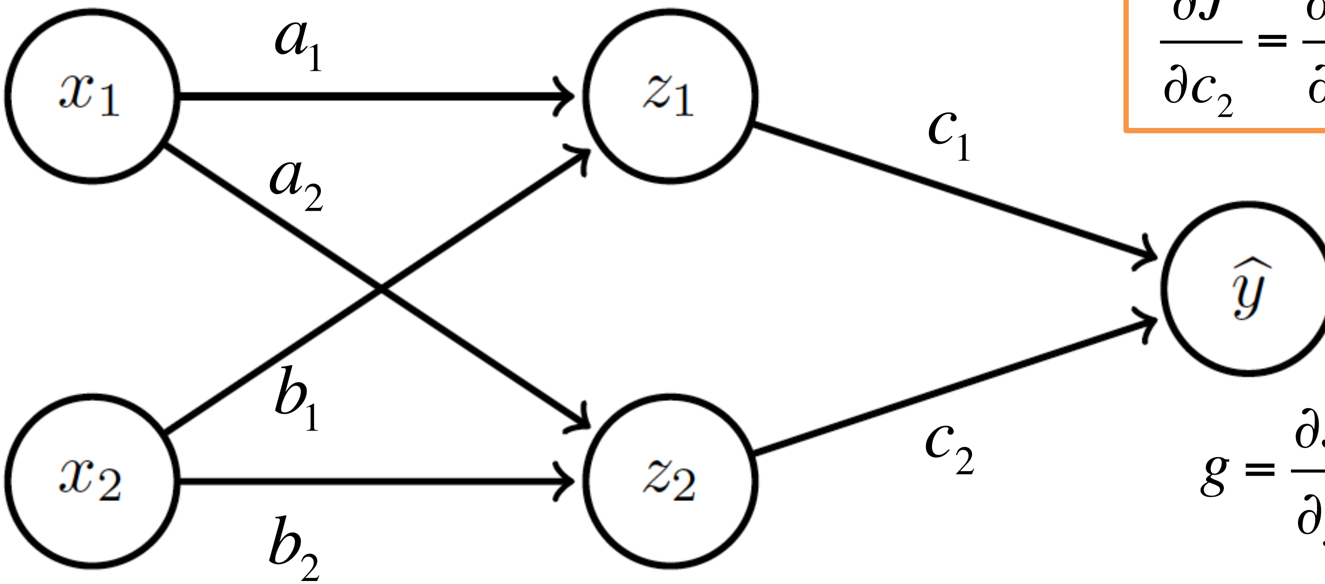
$$\boxed{g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y}$$

Backward prop: Compute loss and its derivative

Backprop: Example

$$\frac{\partial J}{\partial c_1} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial c_1}$$

$$\frac{\partial J}{\partial c_2} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial c_2}$$



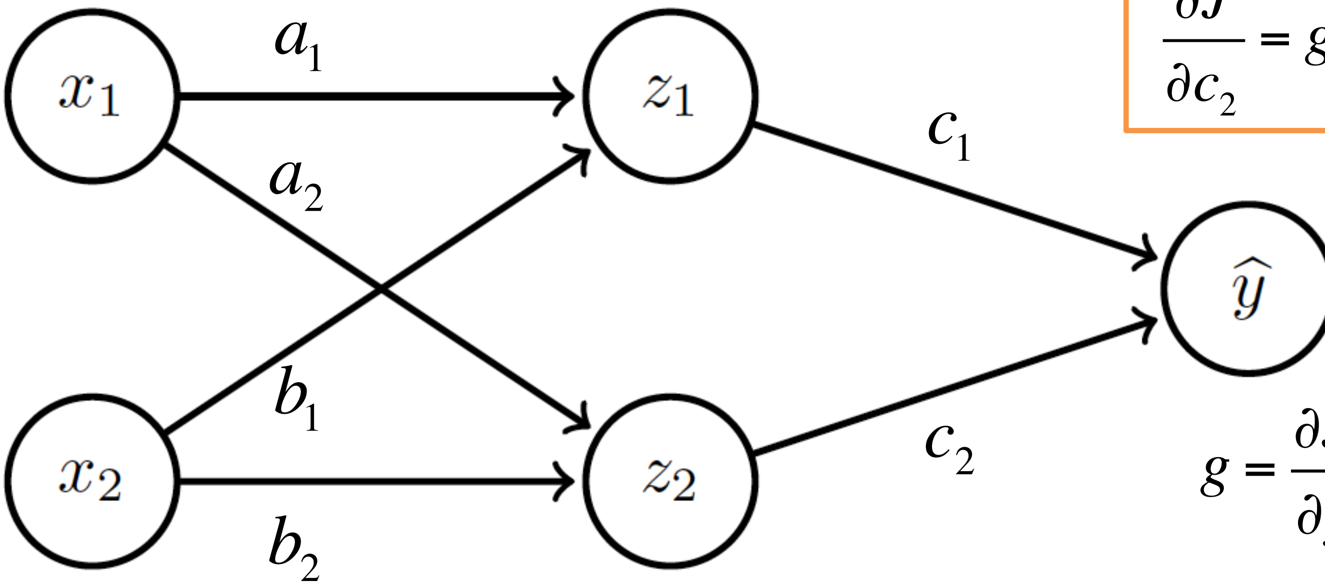
$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights c_1 and c_2

Backprop: Example

$$\frac{\partial J}{\partial c_1} = g \cdot z_1$$

$$\frac{\partial J}{\partial c_2} = g \cdot z_2$$

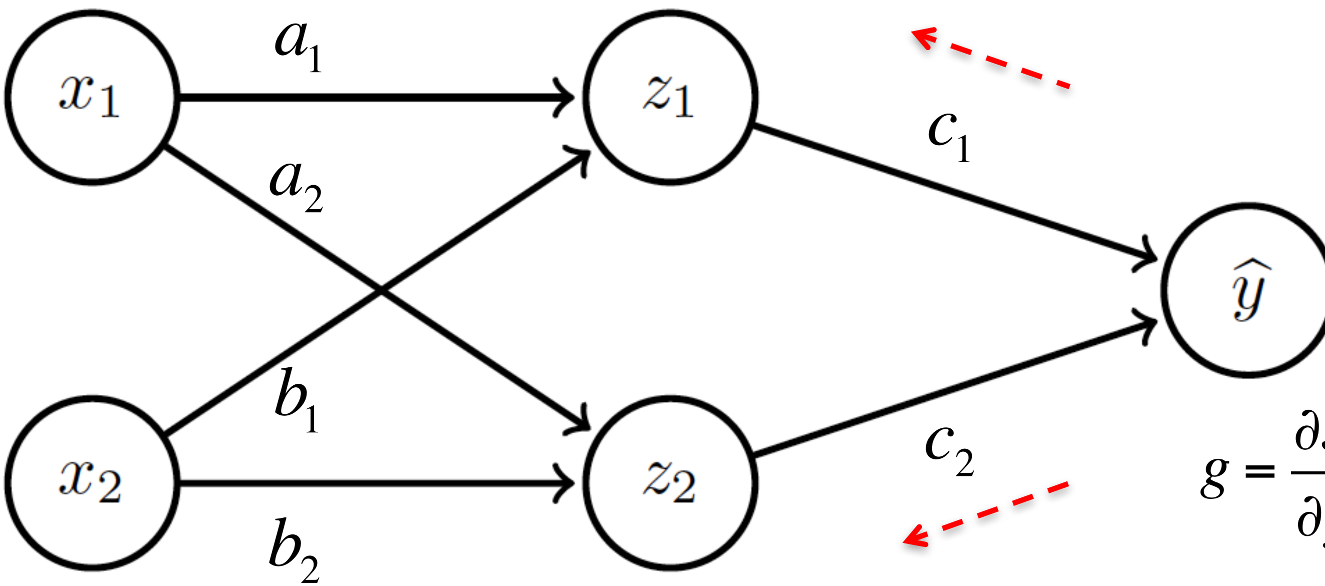


$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

Backward prop: Compute derivatives w.r.t. weights c_1 and c_2

Backprop: Example

$$\boxed{g_1' = \frac{\partial J}{\partial z_1} = g c_1}$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$\boxed{g_2' = \frac{\partial J}{\partial z_2} = g c_2}$$

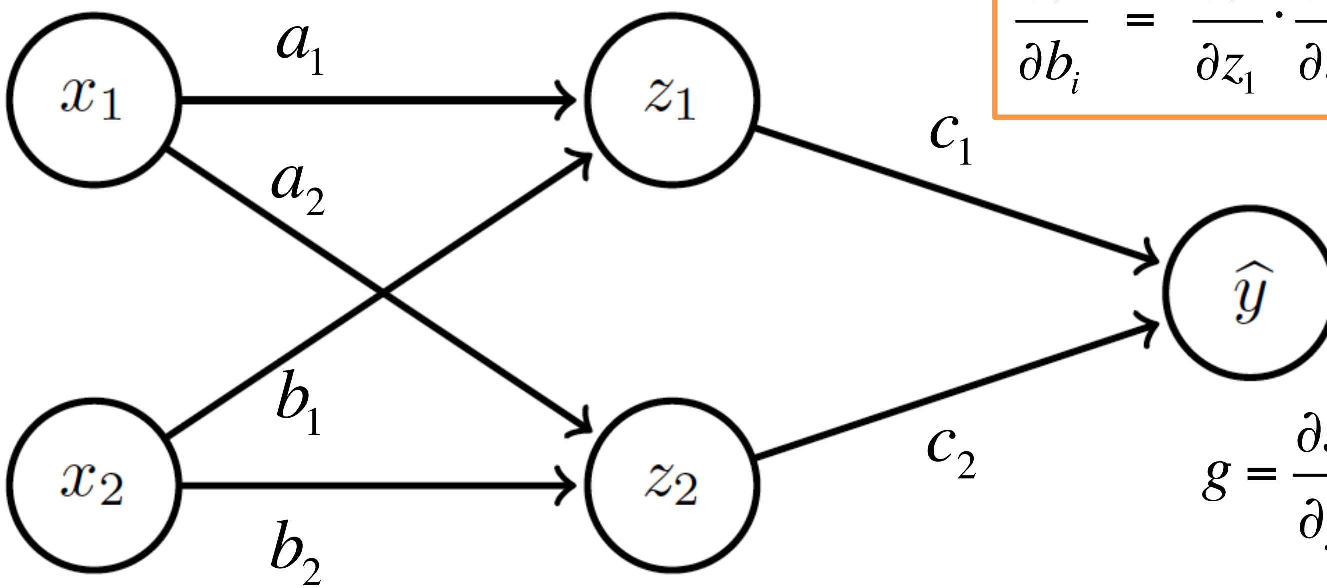
Backward prop: Propagate derivative to hidden layer

Backprop: Example

$$\frac{\partial J}{\partial a_i} = \frac{\partial J}{\partial z_1} \cdot \frac{\partial z_1}{\partial a_i} + \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_i}$$

$$\frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_i} + \frac{\partial J}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_i}$$

$$g_1' = \frac{\partial J}{\partial z_1} = g c_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2' = \frac{\partial J}{\partial z_2} = g c_2$$

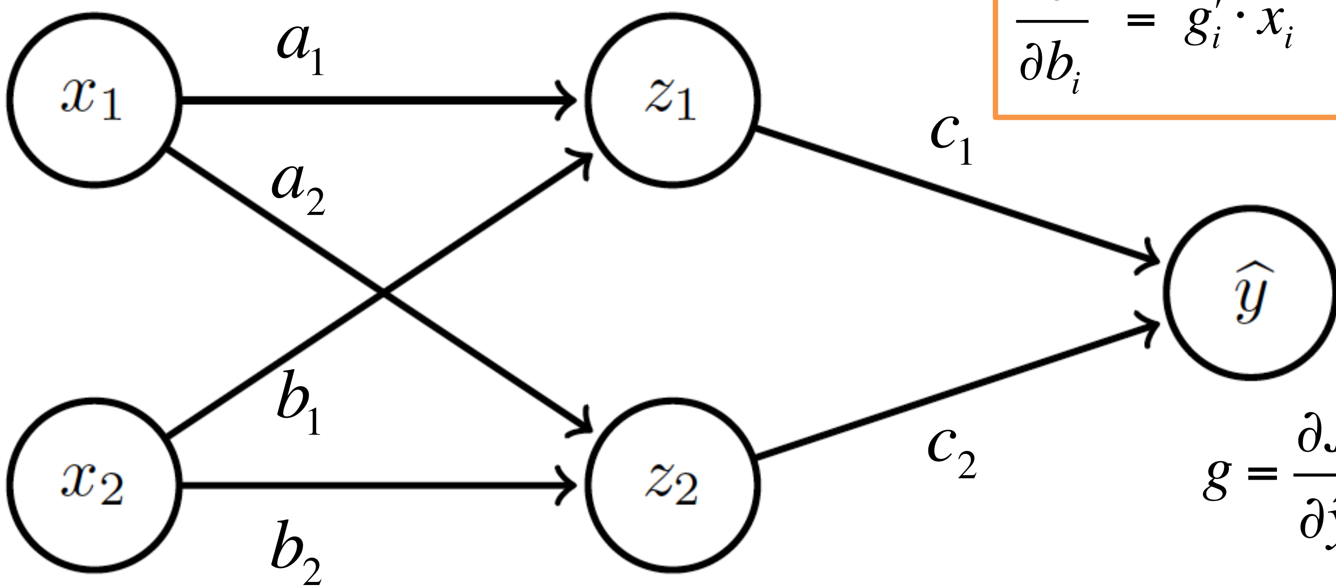
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example

$$\frac{\partial J}{\partial a_i} = g_i' \cdot x_i$$

$$\frac{\partial J}{\partial b_i} = g_i' \cdot x_i$$

$$g_1' = \frac{\partial J}{\partial z_1} = gc_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

$$g_2' = \frac{\partial J}{\partial z_2} = gc_2$$

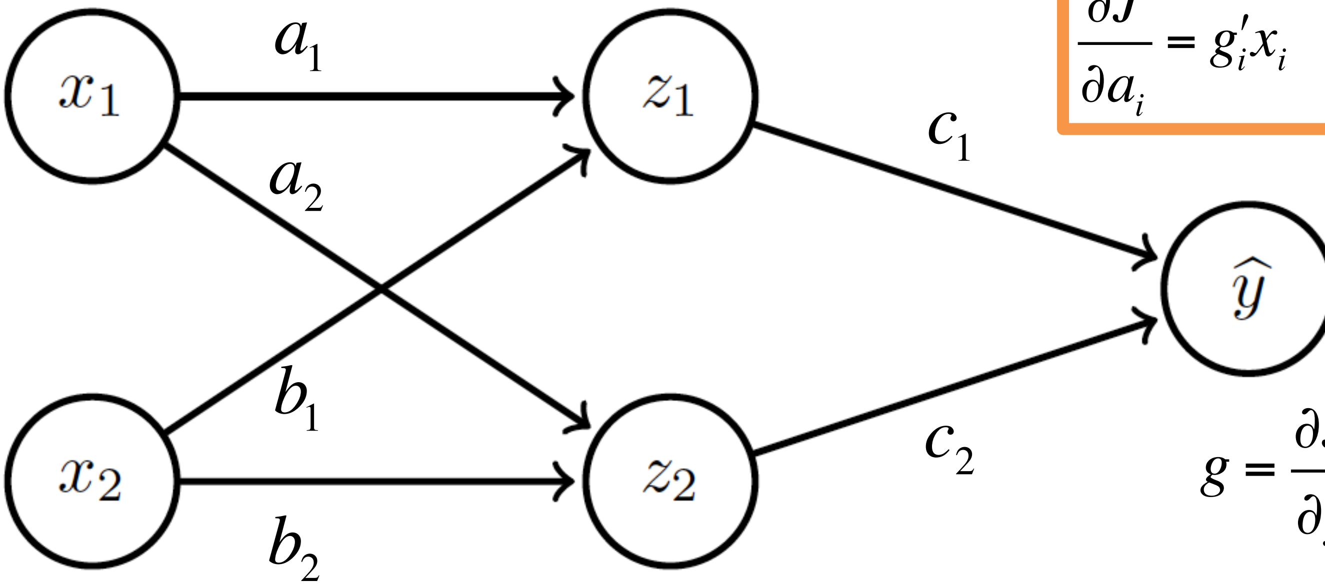
Backward prop: Compute derivatives w.r.t. weights a_1, a_2, b_1 and b_2

Backprop: Example

$$\frac{\partial J}{\partial c_1} = g z_1 \qquad \frac{\partial J}{\partial c_2} = g z_2$$

$$\frac{\partial J}{\partial a_i} = g_i' x_i \qquad \frac{\partial J}{\partial b_i} = g_i' x_i$$

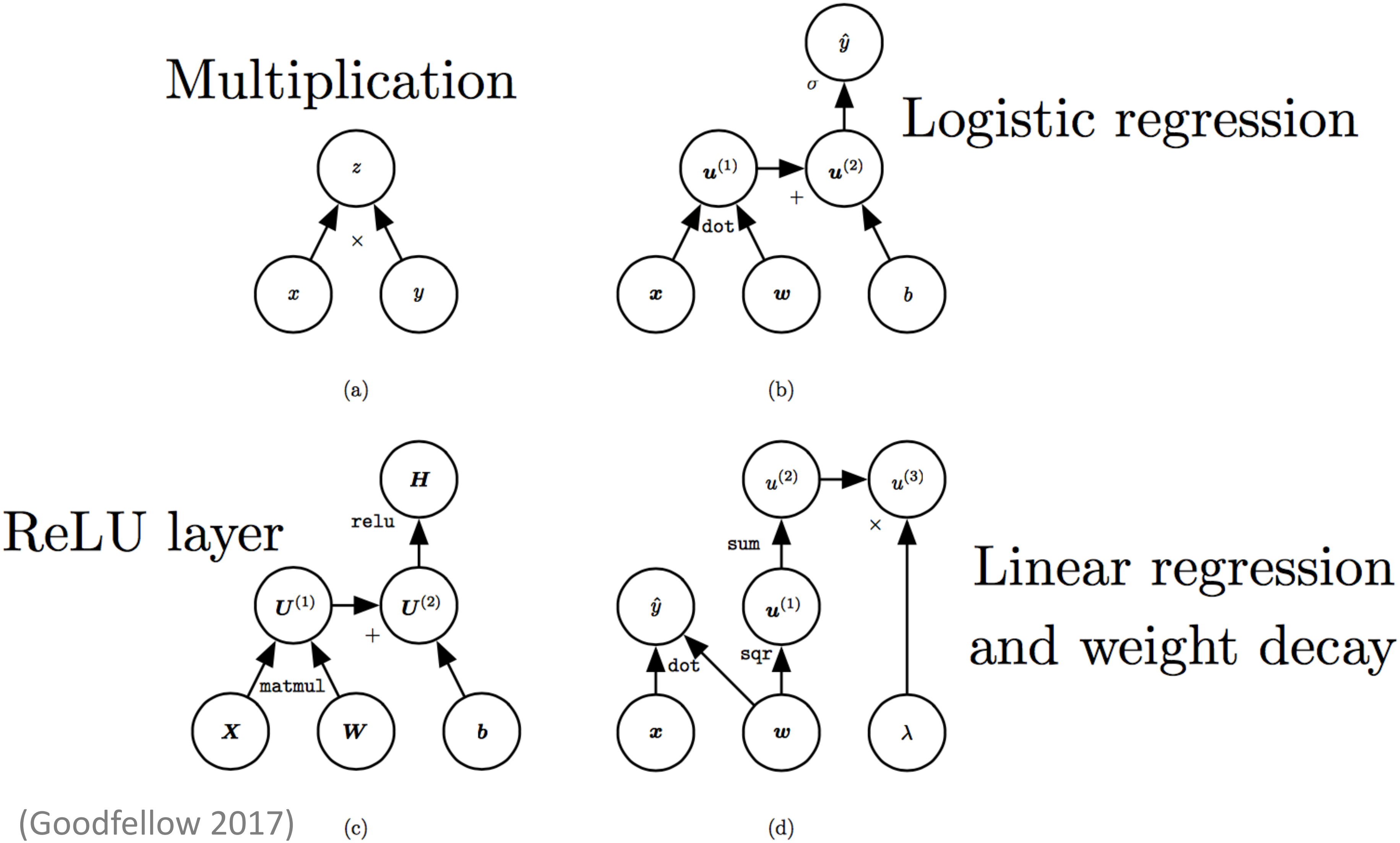
$$g_1' = \frac{\partial J}{\partial z_1} = g c_1$$



$$g = \frac{\partial J}{\partial \hat{y}} = \hat{y} - y$$

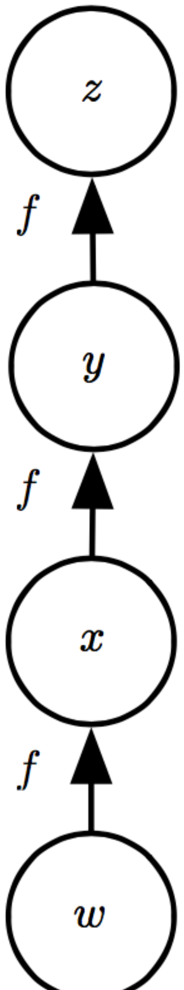
$$g_2' = \frac{\partial J}{\partial z_2} = g c_2$$

Computation Graphs



(Goodfellow 2017)

Repeated Sub-expressions



$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
&= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
&= f'(y)f'(x)f'(w) \\
&= f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

Back-prop avoids computing this twice

(Goodfellow 2017)

Backprop on Computation Graph

Maintain grad table

1: Initialize $\mathbf{g} \in R^n$ where g_i denotes $\dfrac{\partial u^n}{\partial u^i}$

2: for $j = n-1$ to 1 do:

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

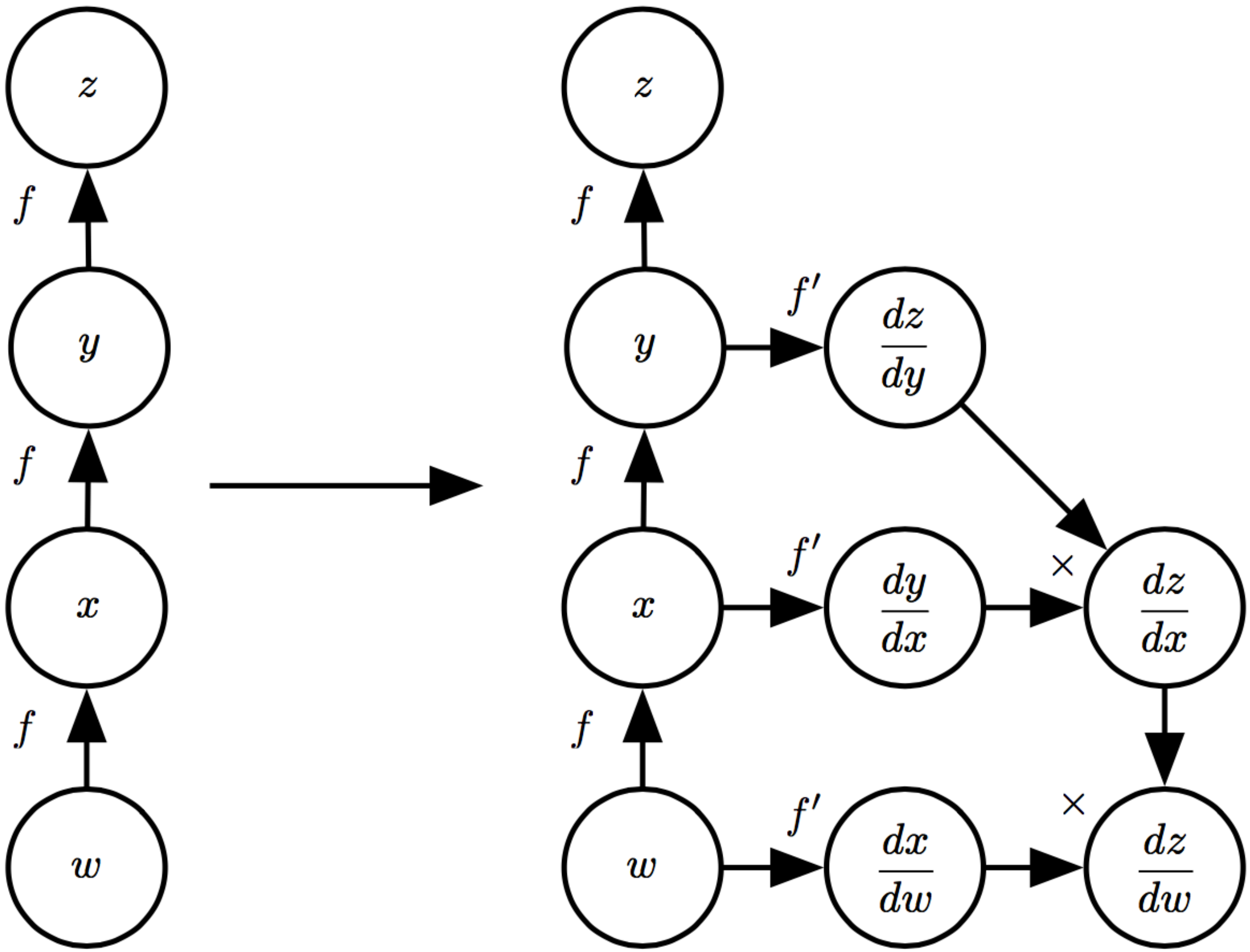
4: return $\mathbf{g}$

Parents of u^i

Symbol-to-symbol Differentiation

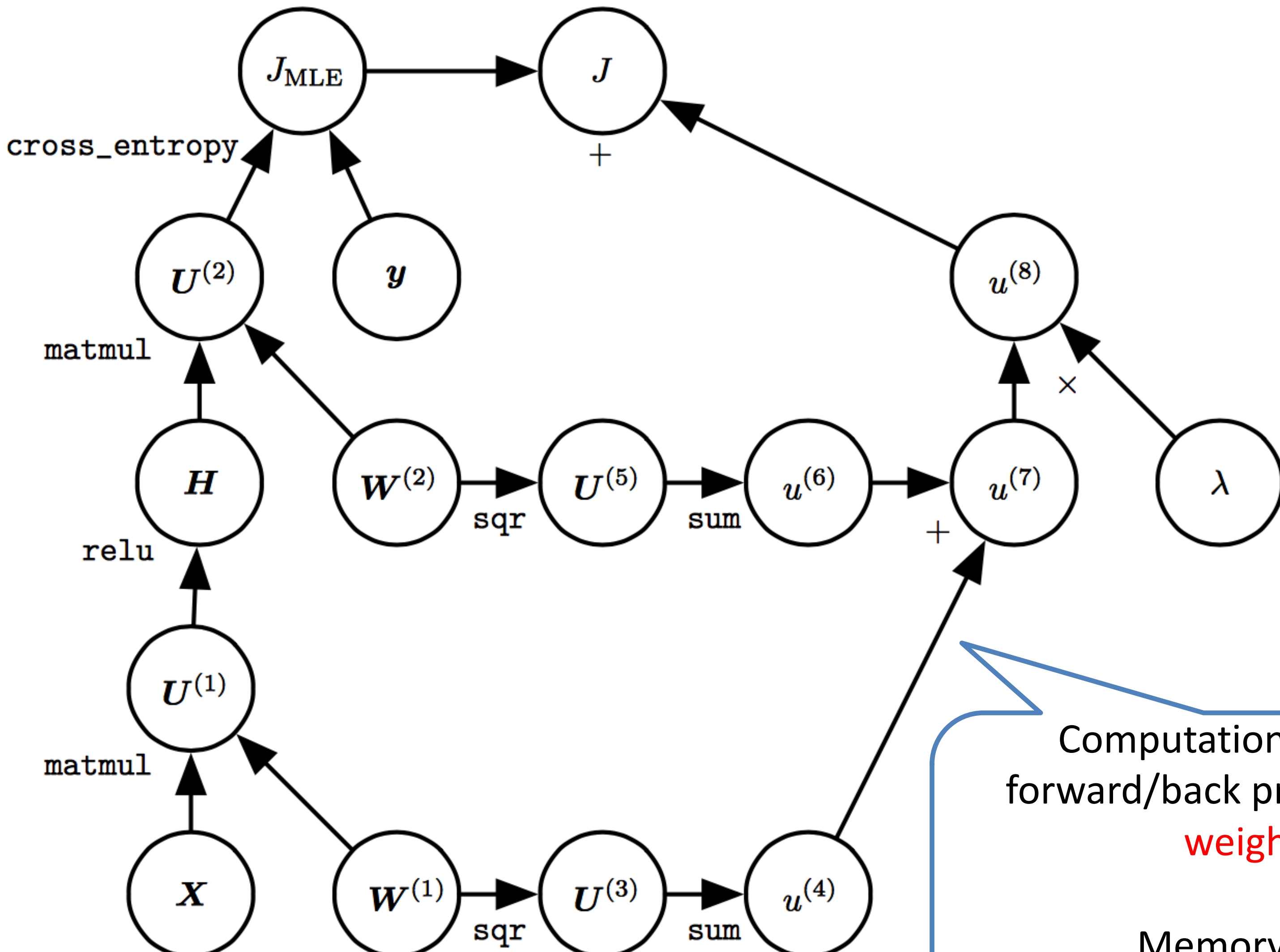
- Derivatives as computation graphs
  - Same language for both forward and back-propagation
- During execution, replace symbolic inputs with numeric value
- Used by Theano and TensorFlow
- Symbol-to-number differentiation: e.g. Torch and Caffe

Symbol-to-symbol Differentiation



(Goodfellow et al. 2017)

Training Feed-forward Nets



Computational cost for forward/back prop: O(#num-weights)

Memory cost: O(#num-layers x minibatch-size)

(Goodfellow et al. 2017)