Lecture 18: Training a Neural Network

COMP 411, Fall 2021
Victoria Manfredi

WESLEYAN
UNIVERSITY

Acknowledgements: These slides are based primarily on slides created by Vivek Srikumar (Utah), Dan Roth (Penn), Sergey Levine (UC Berkeley), content from the book "Machine Learning" by Tom Mitchell and content from the book "Neural Networks and Deep Learning" by Michael Nielsen

Today's Topics

Training a neural network

- Backpropagation

vumanfredi@wesleyan.edu

Training a Neural Network

MOTIVATION

Training a neural network

Given

- **A network architecture**
  - layout of neurons, neuron connectivity, and neuron activations
- **A dataset of labeled examples**
  - $S = \{(\mathbf{x_i}, y_i)\}$

Goal

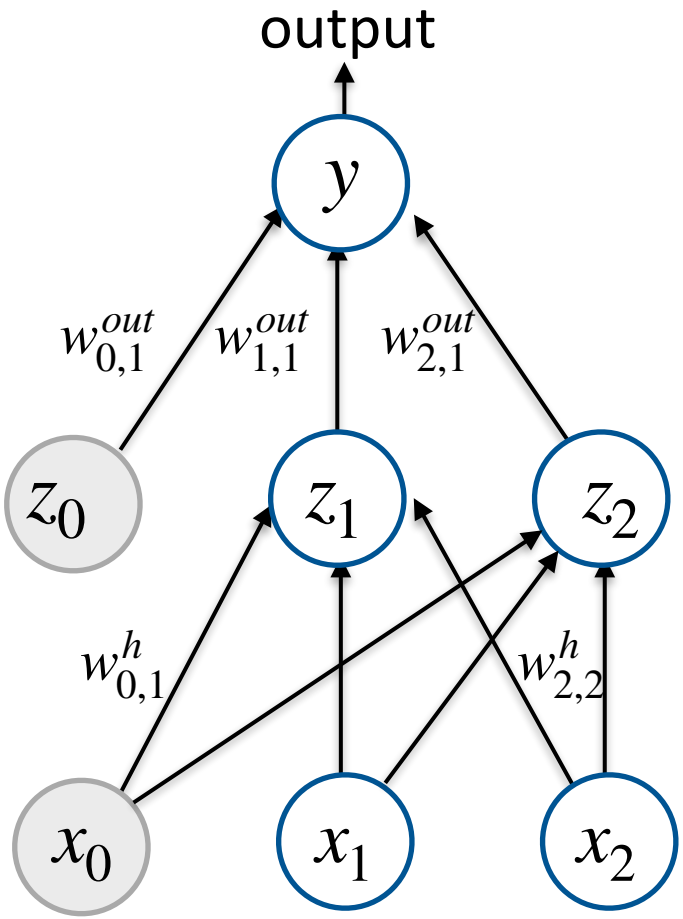
- Learn the weights of the neural network

Remember

- For a fixed architecture, a neural network is a function parameterized by its weights
- Prediction: $y = NN(\mathbf{x}, \mathbf{w})$

Back to our running example

Given an input $\mathbf{x}$, how is the output predicted?



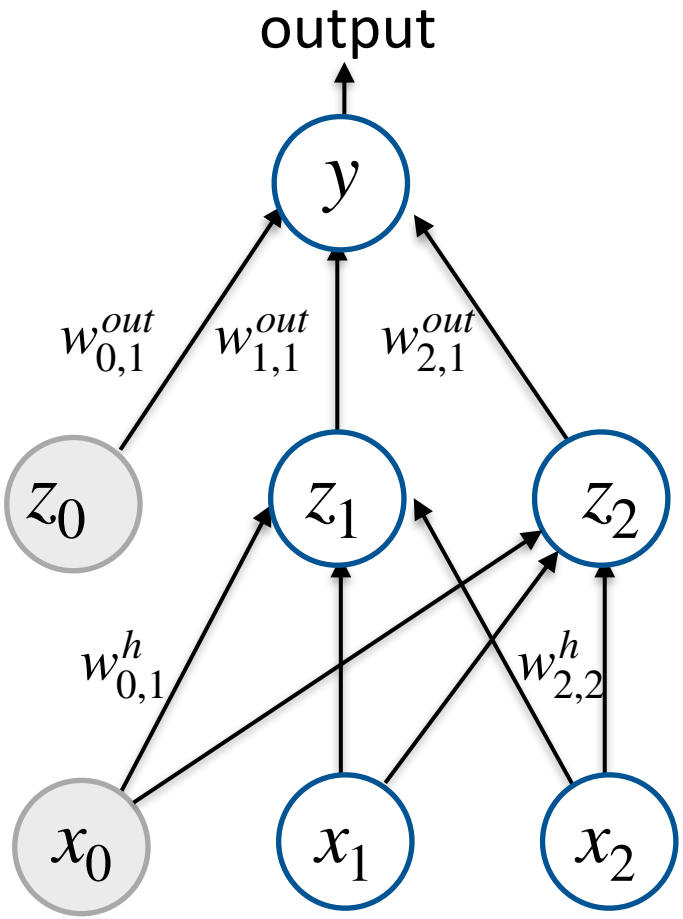
$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

Back to our running example

Given an input $\mathbf{x}$, how is the output predicted?



$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

Suppose the true label for this example is a number y_i

We can write the square loss for this example as:

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

The derivative of the loss function?

$$\nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$$

If the neural network is a differentiable function, we can find the gradient

- Or maybe its sub-gradient (to minimize non-differentiable function)
- This is decided by the activation functions and the loss function

Easy if only one layer. But how do find the sub-gradient of a more complex function?

- E.g., 150 layer neural network for image classification!

We need an efficient algorithm:
Backpropagation

Checkpoint

If we have neural network (structure, activations, and weights), we can make a prediction for an input

If we had the true label of the input, then we can define the loss for that example

If we can take the derivative of the loss with respect to each of the weights, we can take a gradient step in SGD

Some simple expressions

$f(x, y) = x + y$

$$\frac{\partial f}{\partial x} = 1$$

$$\frac{\partial f}{\partial y} = 1$$

$f(x, y) = xy$

$$\frac{\partial f}{\partial x} = y$$

$$\frac{\partial f}{\partial x} = x$$

$f(x, y) = \max(x, y)$

$$\frac{\partial f}{\partial x} = 1, \text{ if } x \geq y, 0 \text{ otherwise}$$

$$\frac{\partial f}{\partial y} = 1, \text{ if } y \geq x, 0 \text{ otherwise}$$

Useful to keep in mind what these derivatives represent in these (and all other) cases:

$$\frac{\partial f}{\partial x}$$

Represents the rate of change of the function f with respect to a small change in x

More complicated cases?

$$f(x, y, z) = x(y^2 + z)$$

This is still simple enough to manually take derivatives, but let us work through this in a slightly different way

Break down the function in terms of simple forms

$$g = y^2 + z$$
$$f = xg$$

Each of these is a simple form. We know how to compute $\frac{\partial g}{\partial y}, \frac{\partial g}{\partial z}, \frac{\partial f}{\partial x}, \frac{\partial f}{\partial g}$

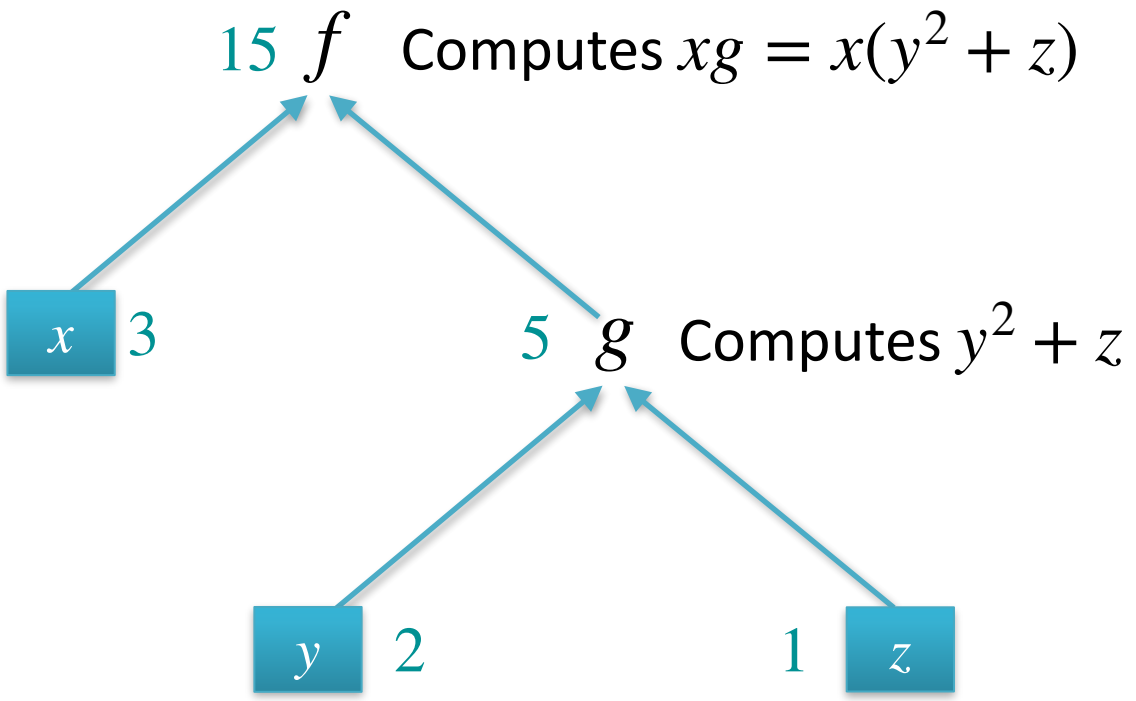
Key idea: build up derivatives of compound expressions by breaking it down into simpler pieces, and applying the chain rule

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial g}, \frac{\partial g}{\partial y} = x \cdot 2y = 2xy$$

In terms of "computation graphs"

$$f(x, y, z) = x(y^2 + z)$$

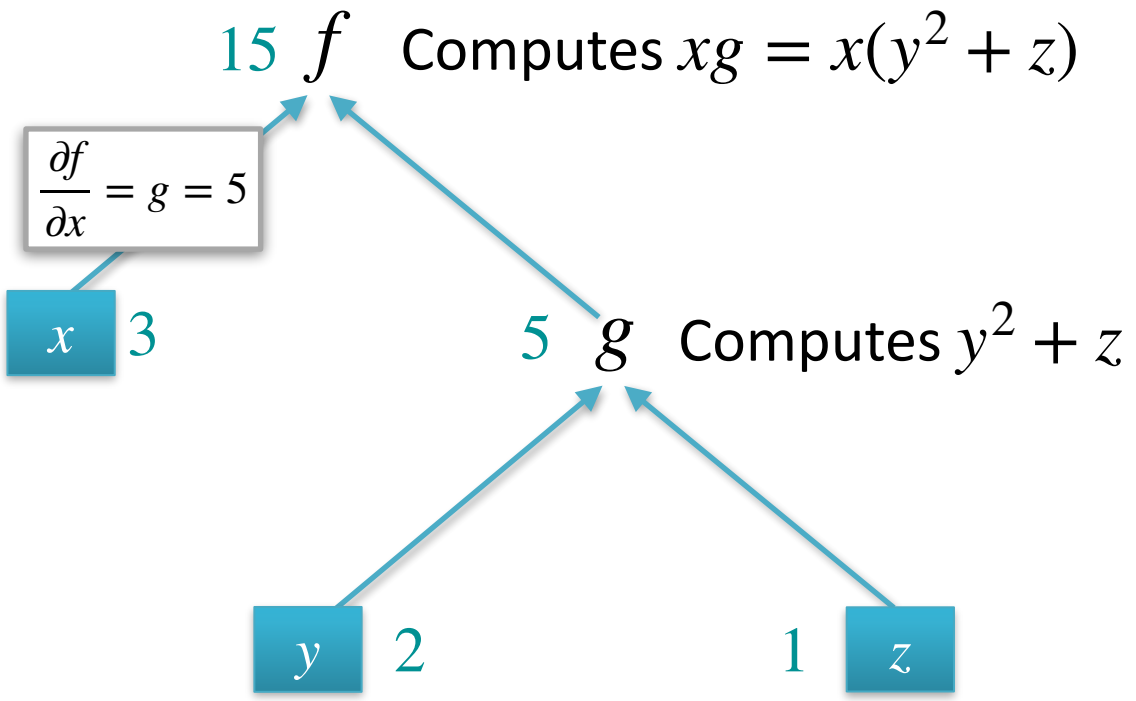
The forward pass:
computes function
values for specific inputs



In terms of "computation graphs"

$$f(x, y, z) = x(y^2 + z)$$

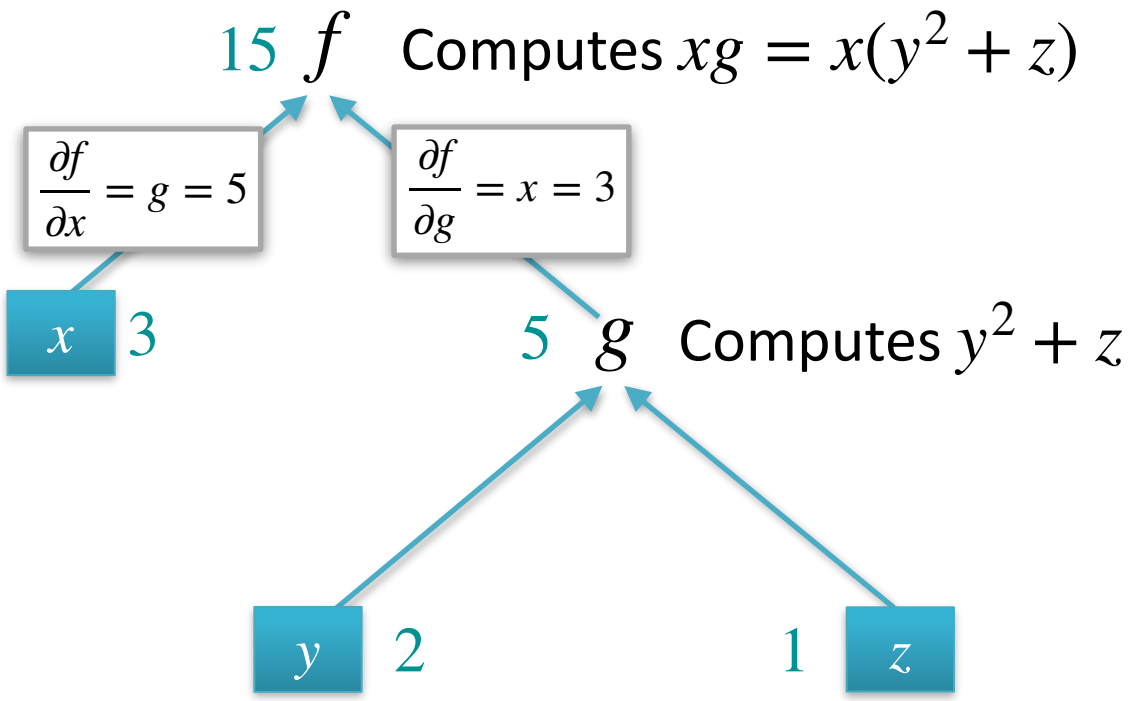
The backward pass:
computes derivatives of each intermediate node



In terms of "computation graphs"

$$f(x, y, z) = x(y^2 + z)$$

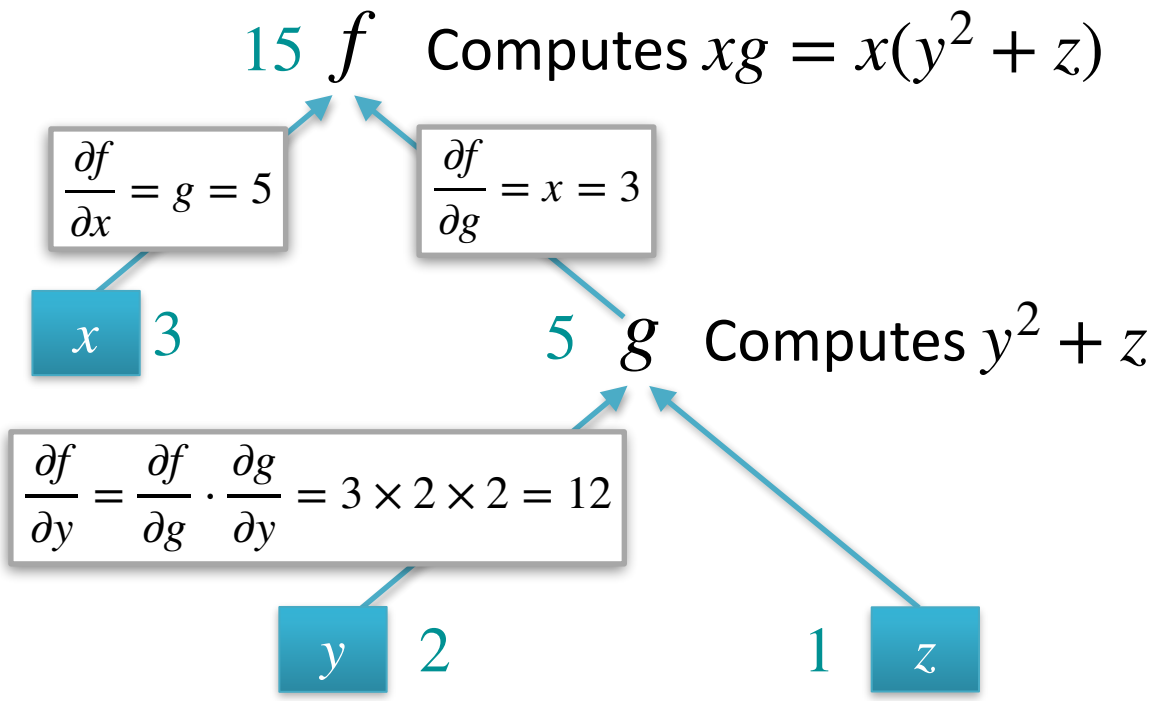
The backward pass:
computes derivatives of each intermediate node



In terms of "computation graphs"

$$f(x, y, z) = x(y^2 + z)$$

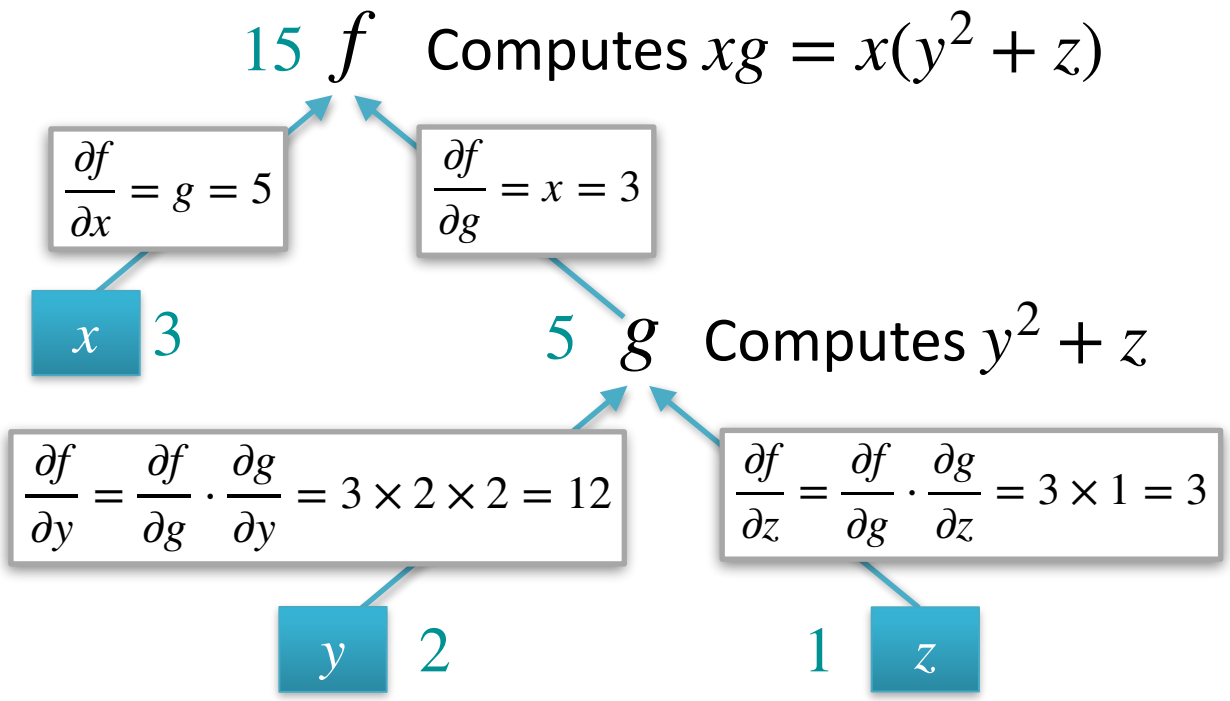
The backward pass:
computes derivatives of each intermediate node



In terms of "computation graphs"

$$f(x, y, z) = x(y^2 + z)$$

The backward pass:
computes derivatives of each intermediate node



The abstraction

Each node in the graph knows 2 things:

1. How to compute the value of a function with respect to its inputs (forward)
2. How to compute the partial derivative of the output with respects to its inputs (backward)

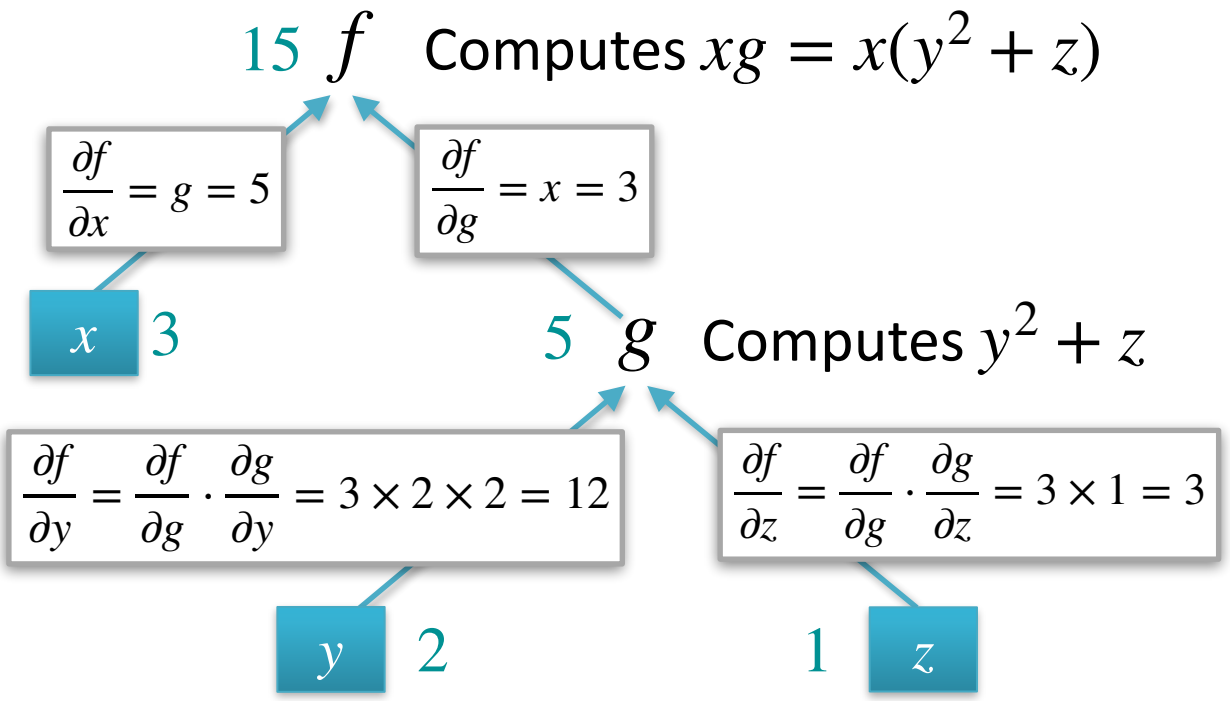
These can be defined independently of what happens in the rest of the graph

We can build up complicated functions using simple nodes and compute values and partial derivatives of these as well

In terms of "computation graphs"

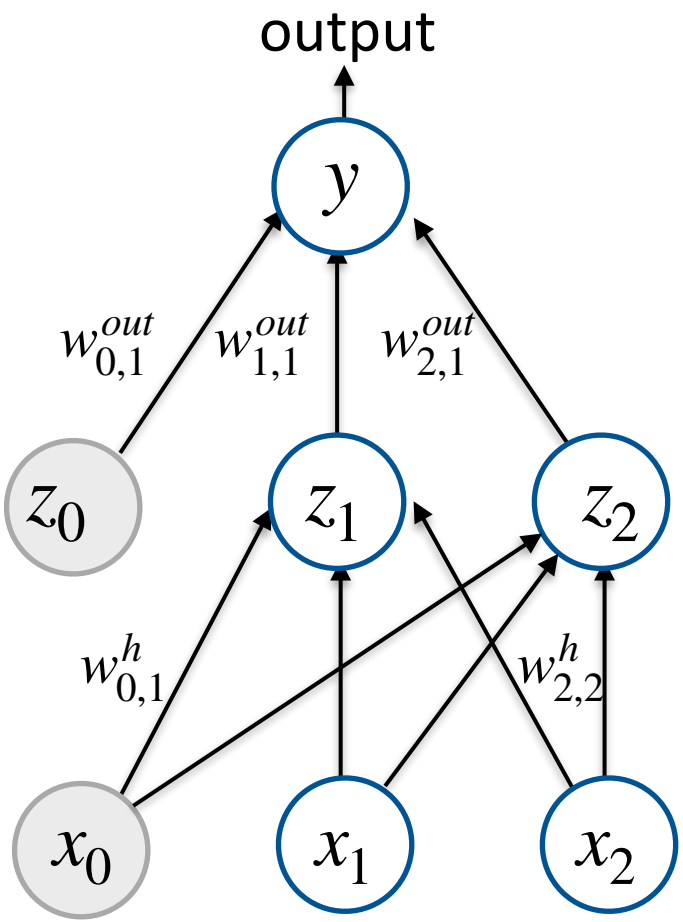
$$f(x, y, z) = x(y^2 + z)$$

Meaning of the partial derivatives: how sensitive is the value of f to the value of each variable



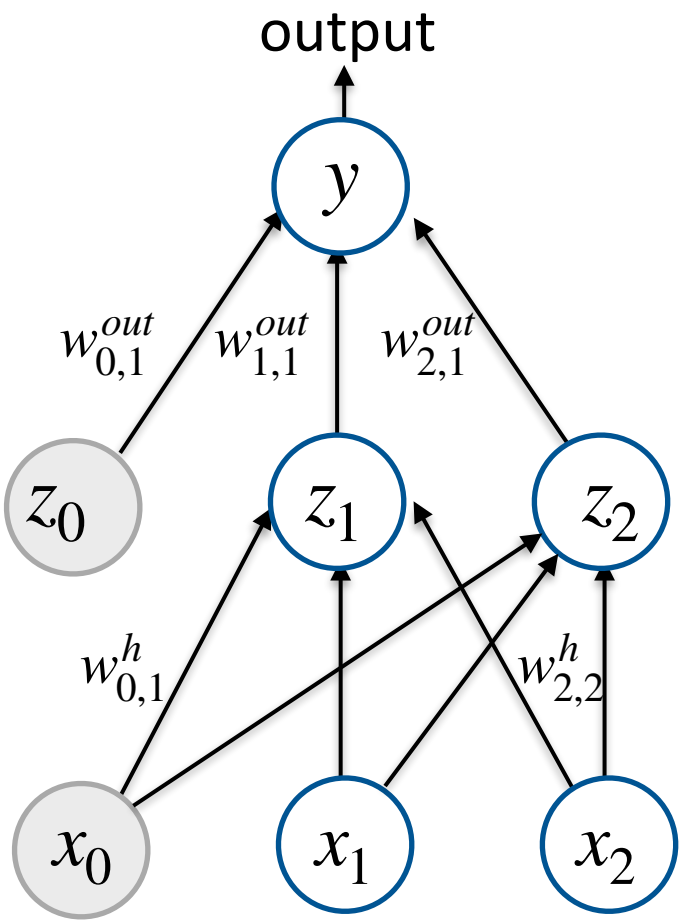
A notational convenience

Commonly nodes in the network represent not only single numbers (e.g., features, outputs) but also ***vectors*** (an array of numbers), ***matrices*** (a 2d array of numbers) or ***tensors*** (an n-dimensional array of numbers)



A notational convenience

Commonly nodes in the network represent not only single numbers (e.g., features, outputs) but also ***vectors*** (an array of numbers), ***matrices*** (a 2d array of numbers) or ***tensors*** (an n-dimensional array of numbers)



Each element of $\mathbf{z}$ is z_i and is generated by the sigmoid activation to each element of $\mathbf{W}^h\mathbf{x}$

$\mathbf{z}$

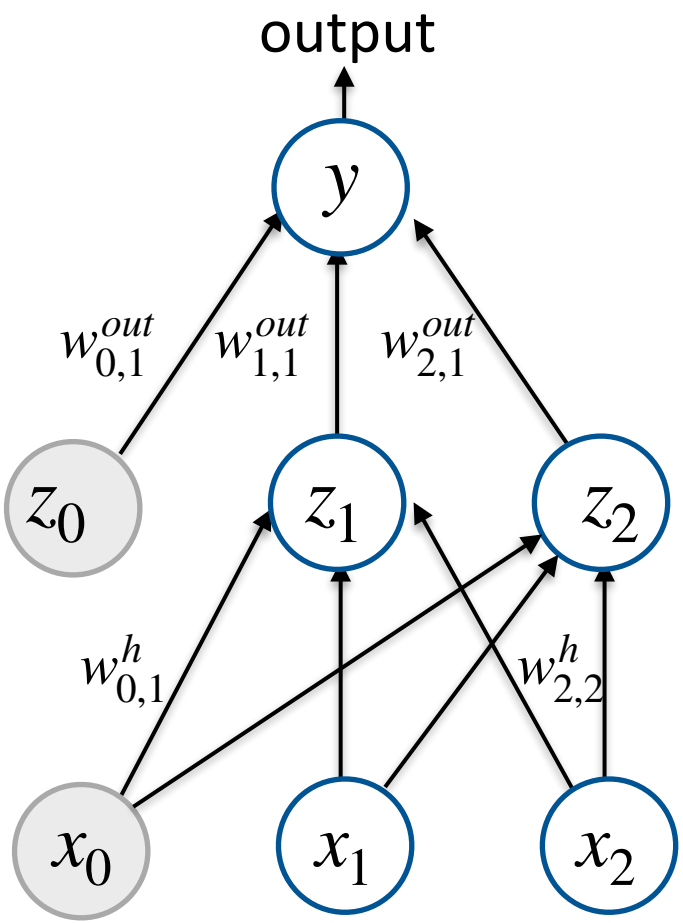
$\mathbf{z} = \sigma(\mathbf{W}^h\mathbf{x})$

$$\mathbf{W}^h = \begin{bmatrix} w^h_{0,1} & w^h_{1,1} & w^h_{2,1} \\ w^h_{0,2} & w^h_{1,2} & w^h_{2,2} \end{bmatrix}$$

$\mathbf{x}$ Represents $[x_0, x_1, x_2]$

A notational convenience

Commonly nodes in the network represent not only single numbers (e.g., features, outputs) but also ***vectors*** (an array of numbers), ***matrices*** (a 2d array of numbers) or ***tensors*** (an n-dimensional array of numbers)



No activation because the output is defined to be linear

$y = \mathbf{w}^o\mathbf{x}$

$\mathbf{W}^o = \begin{bmatrix} w^o_{0,1} & w^o_{1,1} & w^o_{2,1} \end{bmatrix}$

$\mathbf{z} = \sigma(\mathbf{W}^h\mathbf{x})$

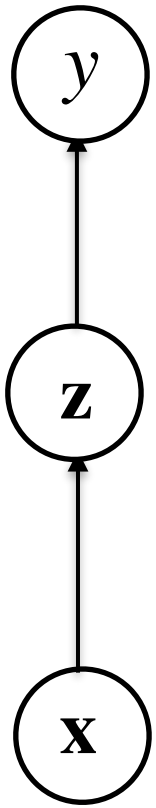
$\mathbf{W}^h = \begin{bmatrix} w^h_{0,1} & w^h_{1,1} & w^h_{2,1} \\ w^h_{0,2} & w^h_{1,2} & w^h_{2,2} \end{bmatrix}$

$\mathbf{x}$ Represents $[x_0, x_1, x_2]$

Reminder: chain rule of derivatives

If y is a function of $\mathbf{z}$ and $\mathbf{z}$ is a function of $\mathbf{x}$ then y is a function of $\mathbf{x}$ as well

How to find $\frac{\partial y}{\partial \mathbf{x}}$?

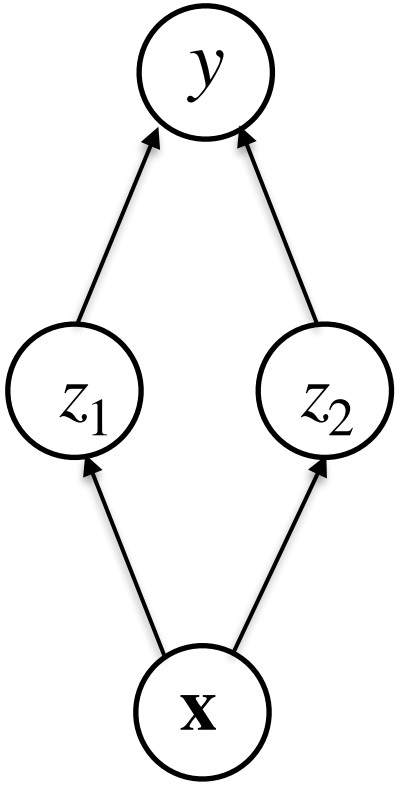


$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

Reminder: chain rule of derivatives

If y is a function of $\mathbf{z}$ and $\mathbf{z}$ is a function of $\mathbf{x}$ then y is a function of $\mathbf{x}$ as well

How to find $\frac{\partial y}{\partial \mathbf{x}}$?

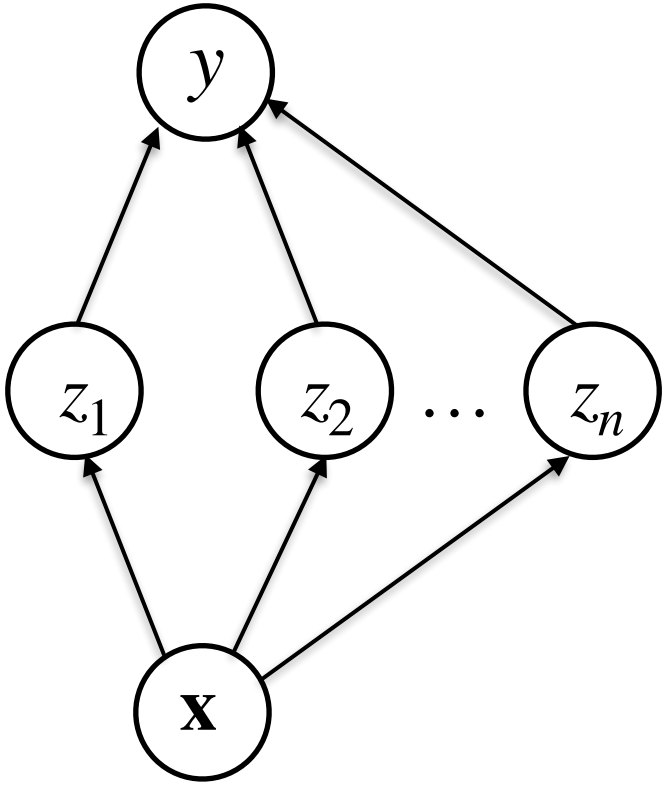


$$\frac{\partial y}{\partial \mathbf{x}} = \frac{\partial y}{\partial z_1} \cdot \frac{\partial z_1}{\partial \mathbf{x}} + \frac{\partial y}{\partial z_2} \cdot \frac{\partial z_2}{\partial \mathbf{x}}$$

Reminder: chain rule of derivatives

If y is a function of $\mathbf{z}$ and $\mathbf{z}$ is a function of $\mathbf{x}$ then y is a function of $\mathbf{x}$ as well

How to find $\frac{\partial y}{\partial \mathbf{x}}$?



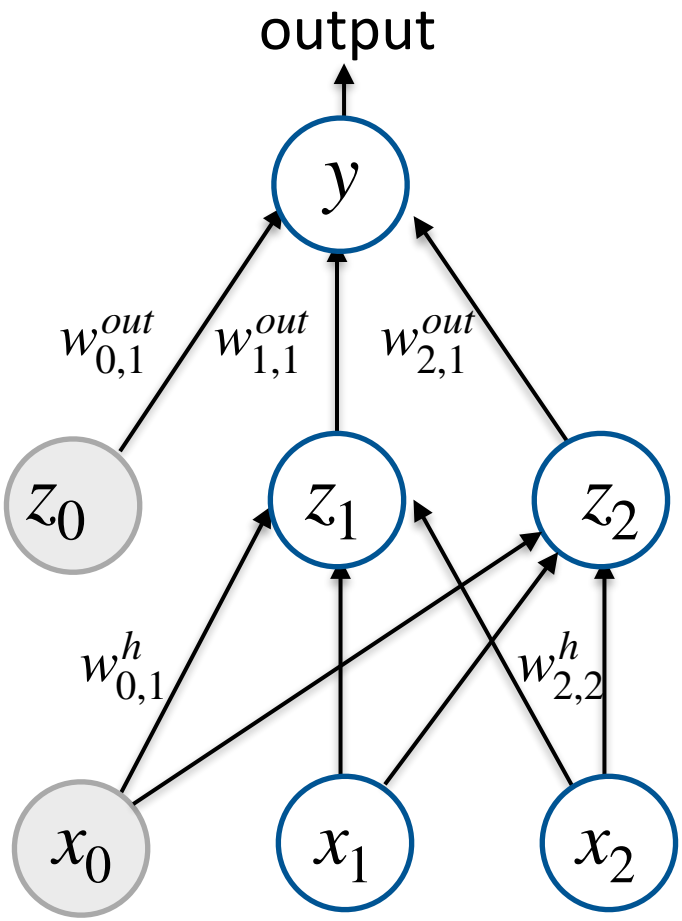
$$\frac{\partial y}{\partial \mathbf{x}} = \sum_{i=1}^{n} \frac{\partial y}{\partial z_i} \cdot \frac{\partial z_i}{\partial \mathbf{x}}$$

Training a neural network

BACKPROPAGATION

Backpropagation (of errors)

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



$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

We want to compute $\frac{\partial L}{\partial w^o_{ij}}$ and $\frac{\partial L}{\partial w^h_{ij}}$

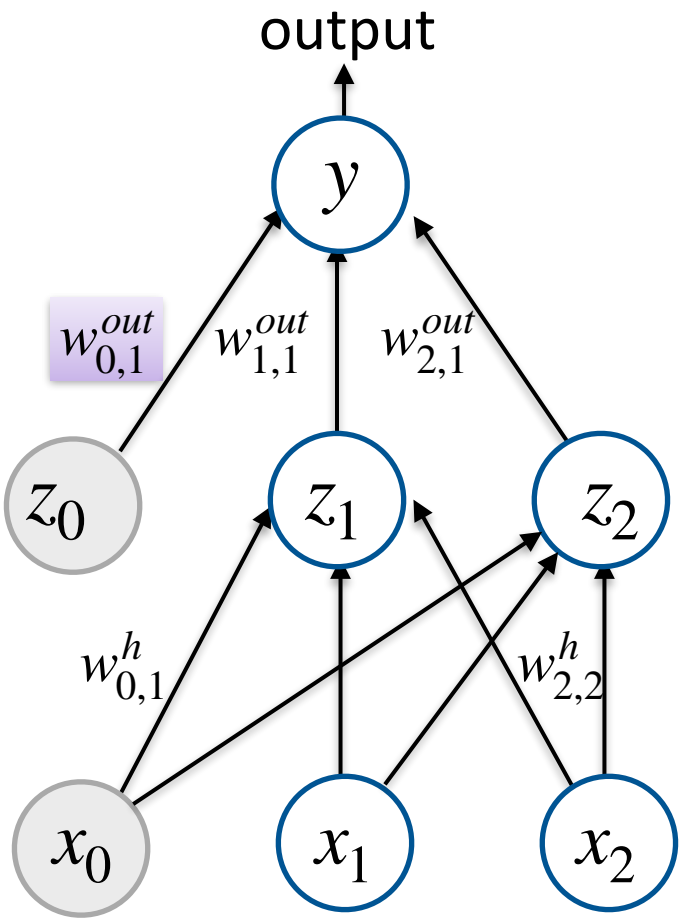
Important: L is a differentiable function of all of the weights

Applying the chain rule to compete the gradient (and remembering partial computations along the way to speed up learning)

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

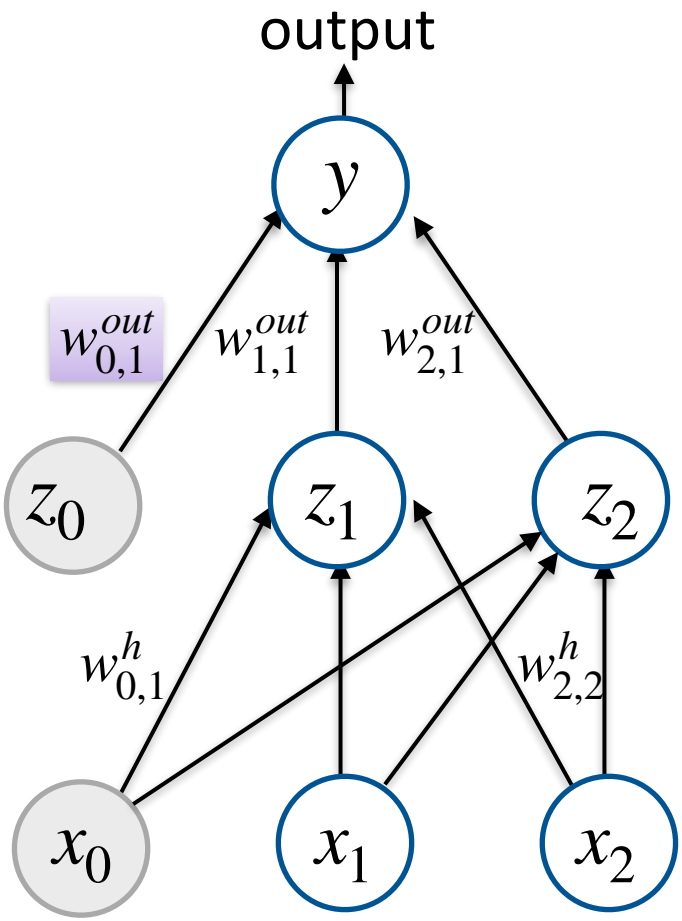


$$\frac{\partial L}{\partial w^o_{0,1}}$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

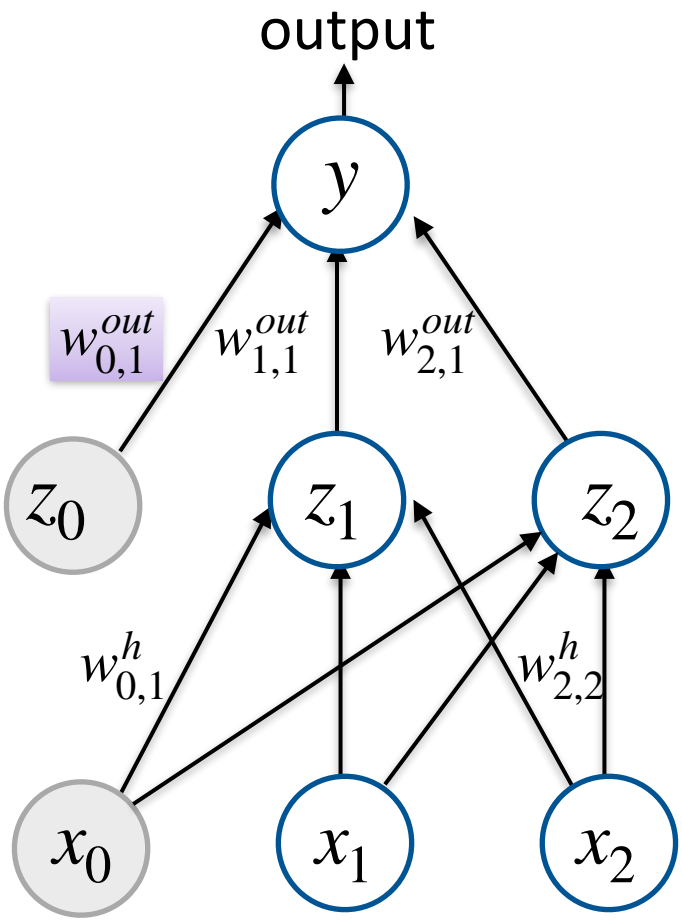


$$\frac{\partial L}{\partial w^o_{0,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{0,1}}$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



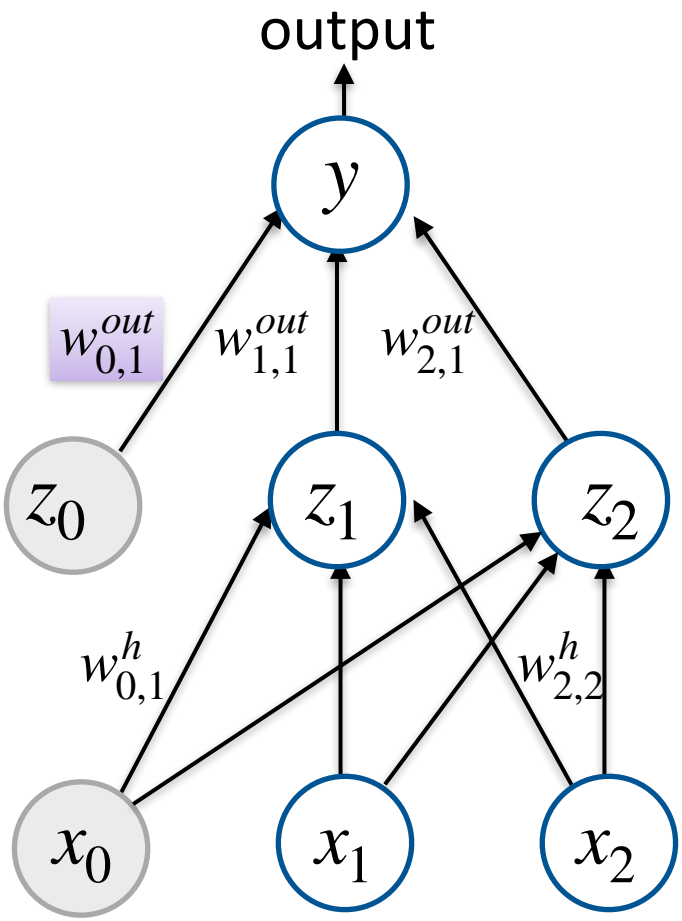
$$\frac{\partial L}{\partial w^o_{0,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{0,1}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^o_{0,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{0,1}}$$

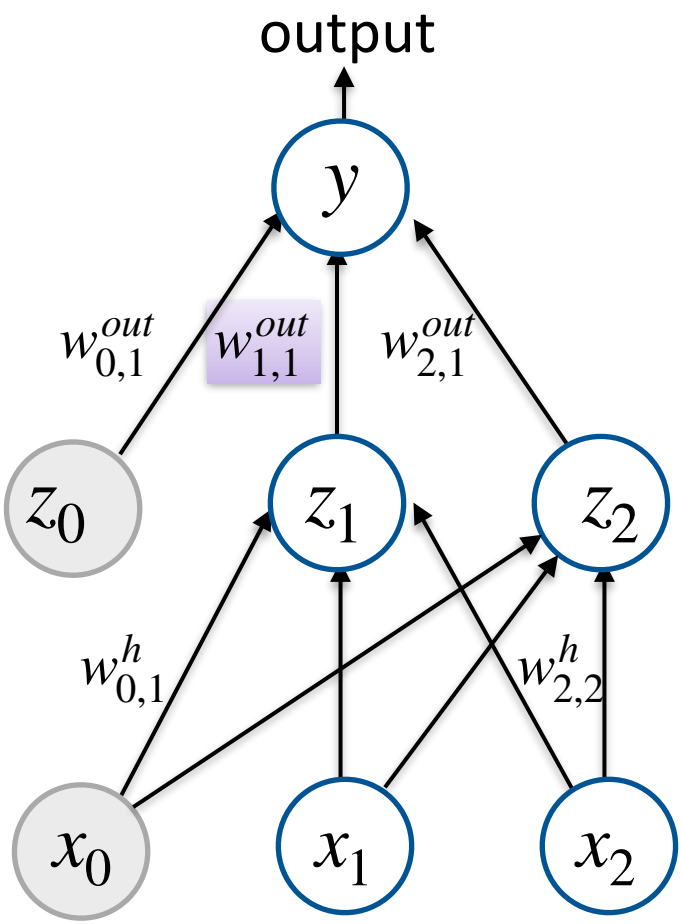
$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial y}{\partial w^o_{0,1}} = 1$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

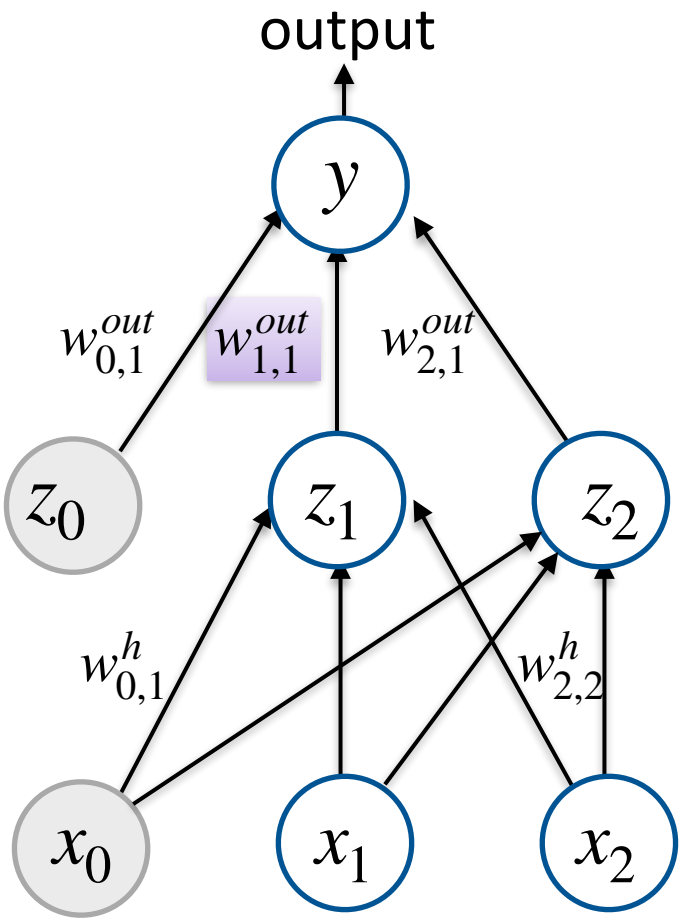


$$\frac{\partial L}{\partial w^o_{1,1}}$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

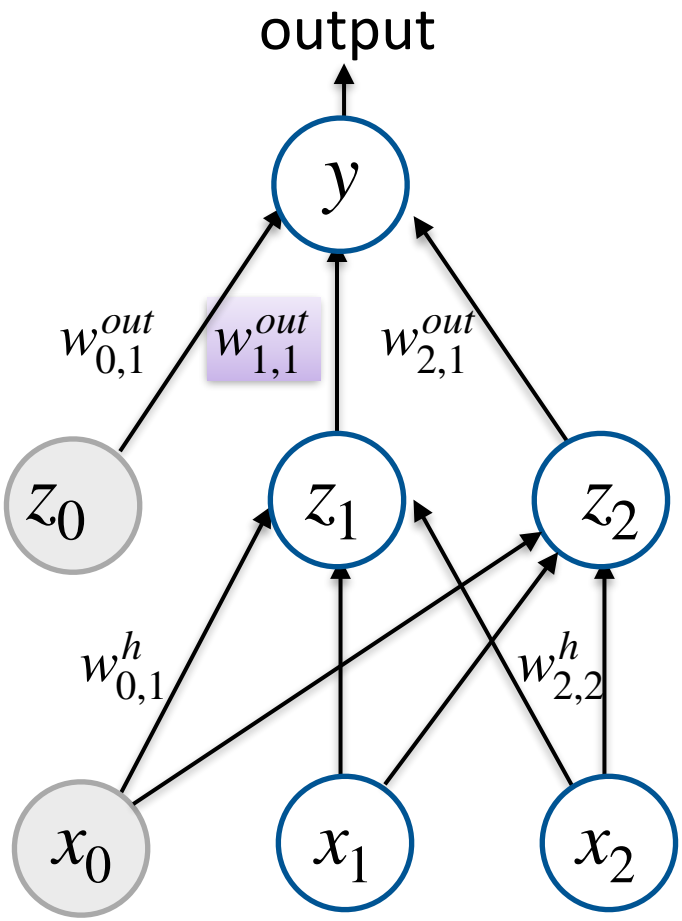


$$\frac{\partial L}{\partial w^o_{1,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{1,1}}$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



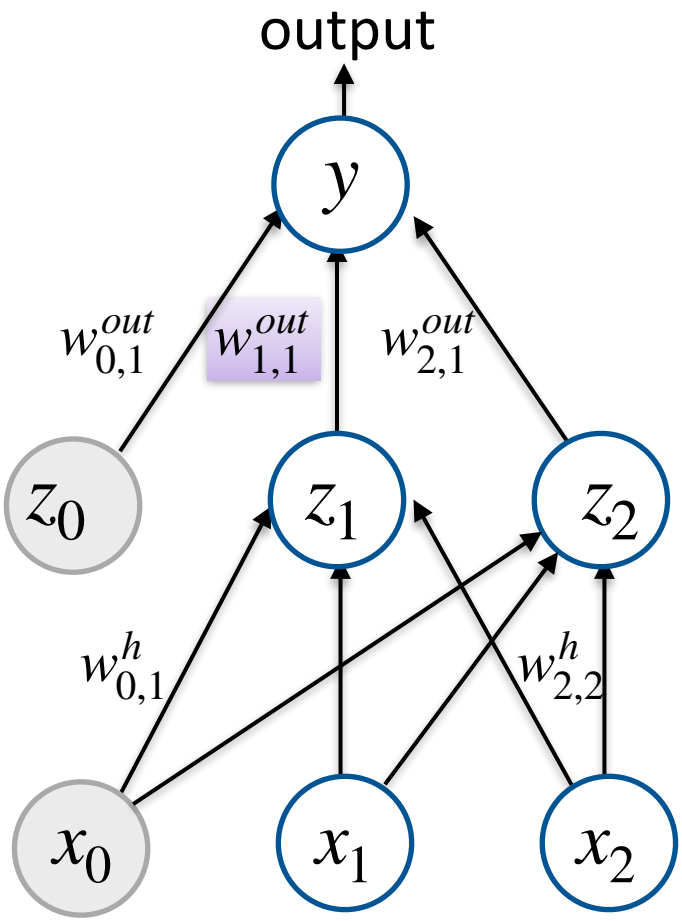
$$\frac{\partial L}{\partial w^o_{1,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{1,1}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

Backpropagation: output layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



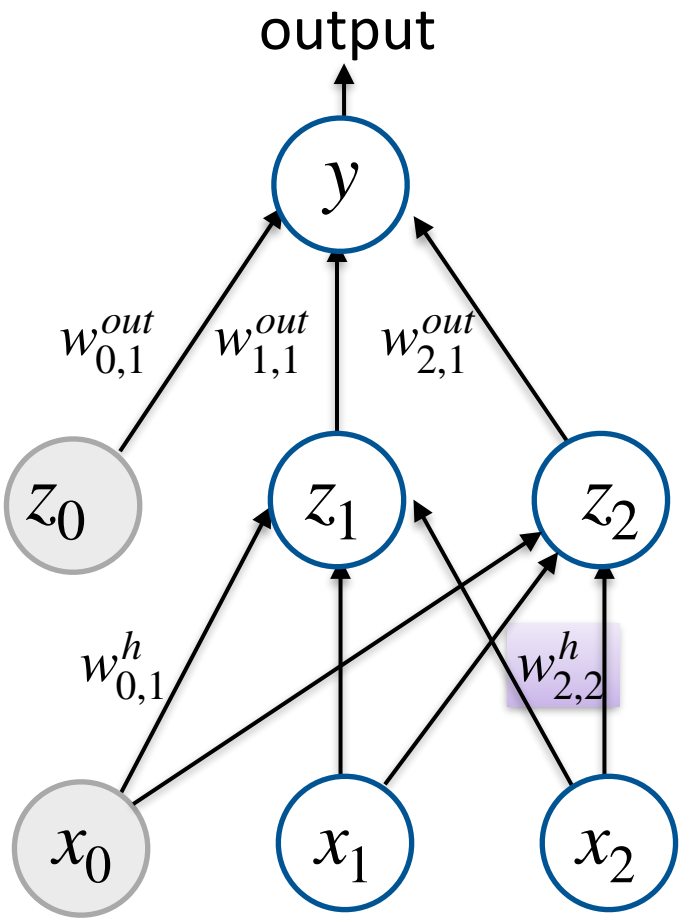
$$\frac{\partial L}{\partial w^o_{1,1}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^o_{1,1}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial y}{\partial w^o_{1,1}} = z_1$$

We have already computed this partial derivative for the previous case. Cache to speed up!

Backpropagation: hidden layer



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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2} x_1 + w^h_{2,2} x_2)$$

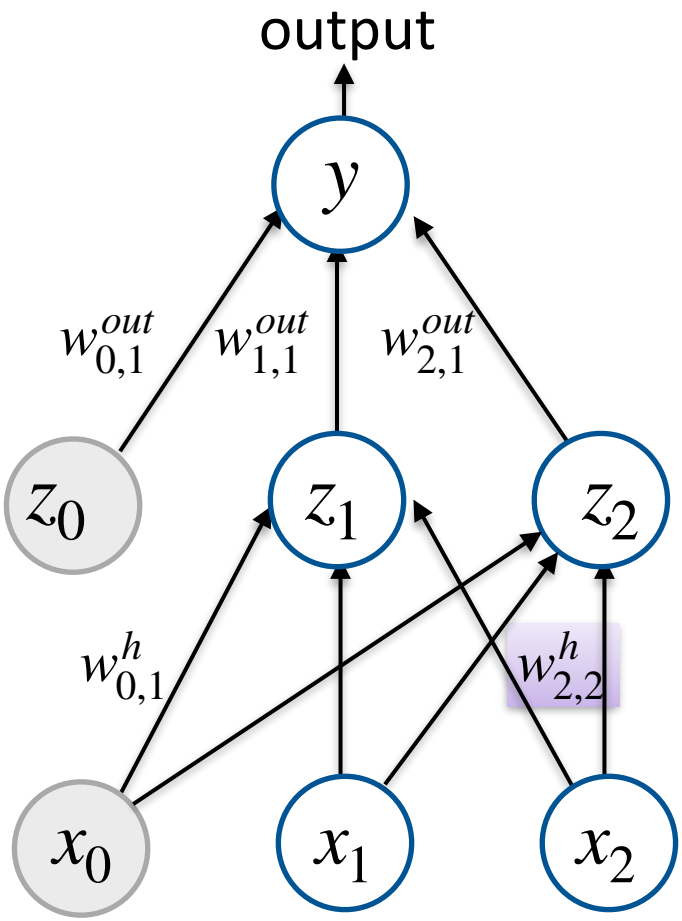
$$z_1 = \sigma(w^h_{0,1} + w^h_{1,1} x_1 + w^h_{2,1} x_2)$$

We want to compute $\frac{\partial L}{\partial w^h_{22}}$

Backpropagation: hidden layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^h_{2,2}}$$

$$= \frac{\partial L}{\partial y} \cdot \frac{\partial}{\partial w^h_{2,2}}(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$= \frac{\partial L}{\partial y} \cdot (w^o_{1,1} \frac{\partial}{\partial w^h_{2,2}} z_1 + w^o_{2,1} \frac{\partial}{\partial w^h_{2,2}} z_2)$$

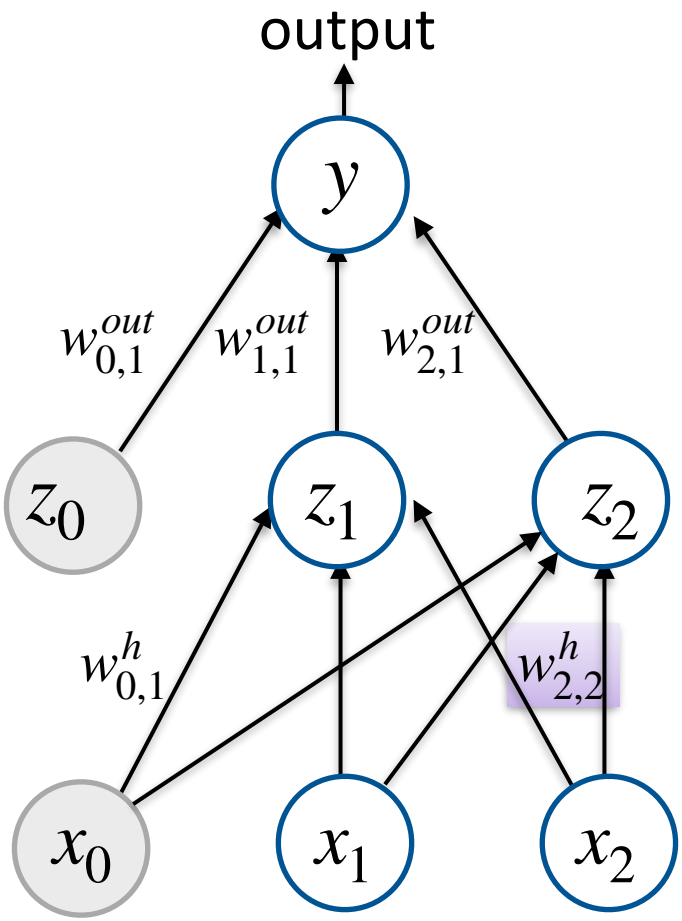
$\frac{\partial}{\partial w^h_{2,2}} z_1 \rightarrow 0$

z_1 is not a function of $w^h_{2,2}$

Backpropagation: hidden layer

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

$$y = \sigma(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^h_{2,2}}$$

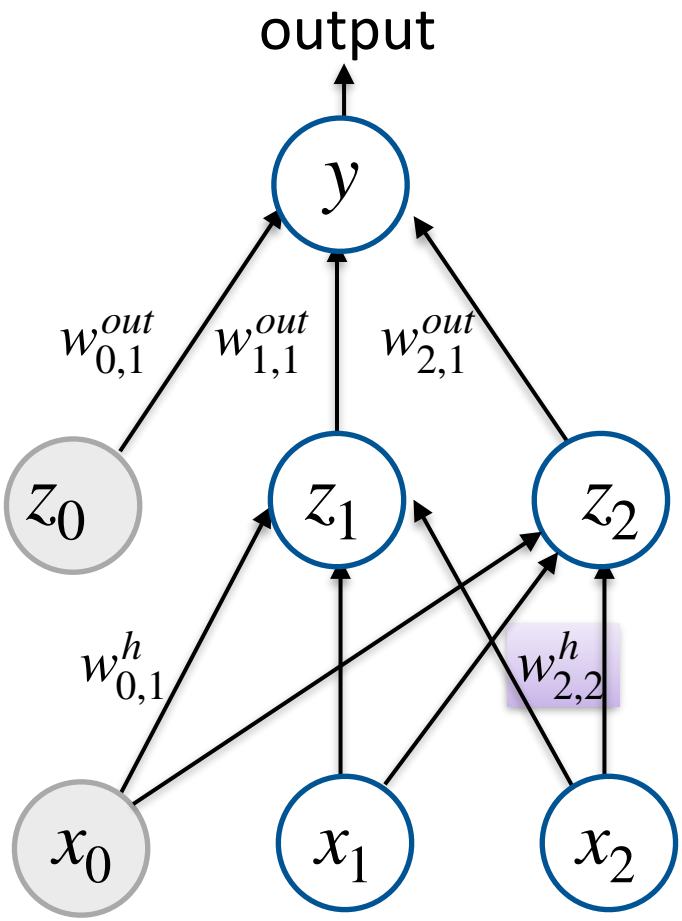
$$= \frac{\partial L}{\partial y} \cdot \frac{\partial}{\partial w^h_{2,2}}(w^o_{0,1} + w^o_{1,1} z_1 + w^o_{2,1} z_2)$$

$$= \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial w^h_{2,2}}$$

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^h_{2,2}}$$

$$= \frac{\partial L}{\partial y} \cdot \frac{\partial}{\partial w^h_{2,2}}(w^o_{0,1} + w^o_{1,1}z_1 + w^o_{2,1}z_2)$$

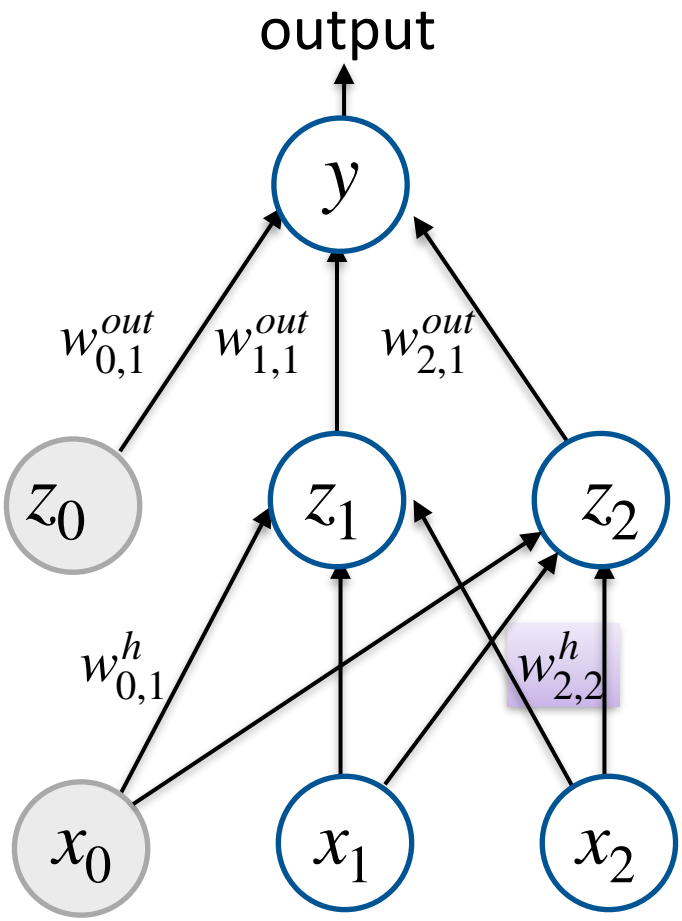
$$= \frac{\partial L}{\partial y} \cdot w^o_{2,1}\frac{\partial z_2}{\partial w^h_{2,2}}$$

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$

s



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w^h_{2,2}}$$

$$= \frac{\partial L}{\partial y} \cdot \frac{\partial}{\partial w^h_{2,2}}(w^o_{0,1} + w^o_{1,1}z_1 + w^o_{2,1}z_2)$$

$$= \frac{\partial L}{\partial y} \cdot w^o_{2,1}\frac{\partial z_2}{\partial w^h_{2,2}}$$

$$= \frac{\partial L}{\partial y} \cdot w^o_{2,1}\frac{\partial z_2}{\partial s}\frac{\partial s}{\partial w^h_{2,2}}$$

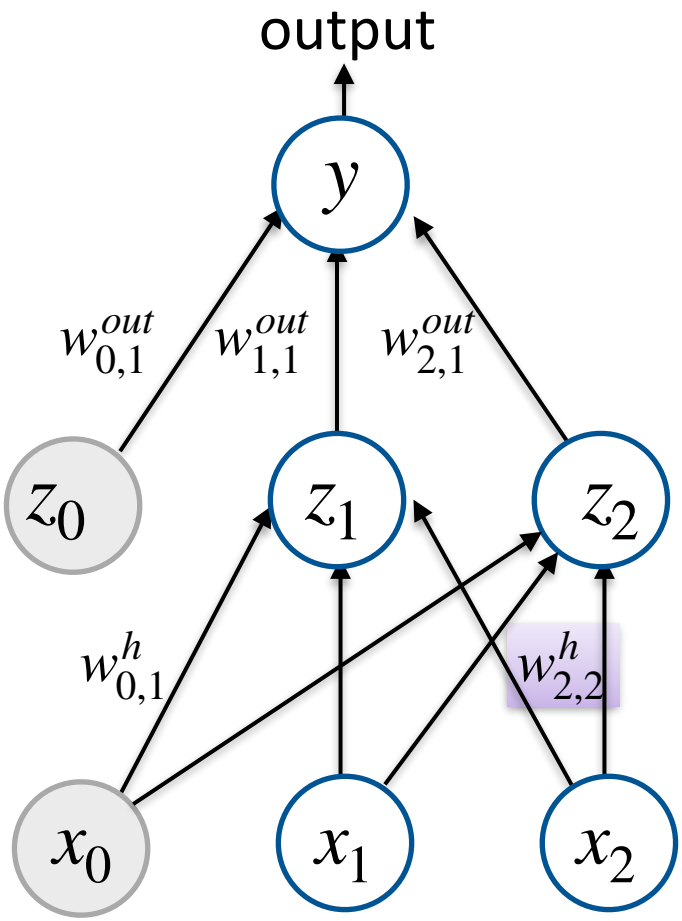
Each of these partial derivatives is easy!

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$

s



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

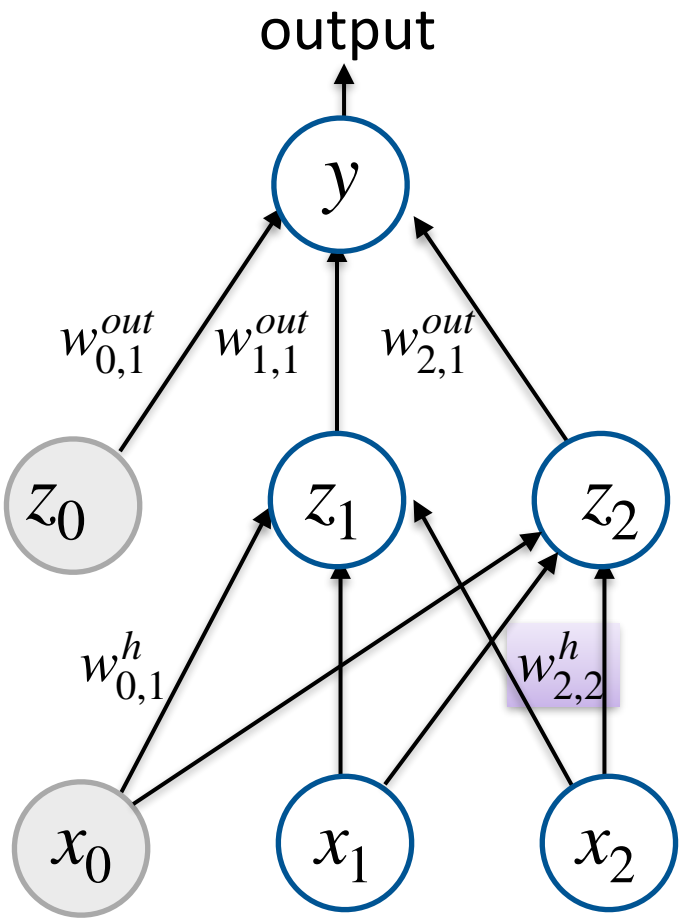
$$\frac{\partial L}{\partial y} = y - y^*$$

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$

s



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial z_2}{\partial s} = z_2(1 - z_2)$$

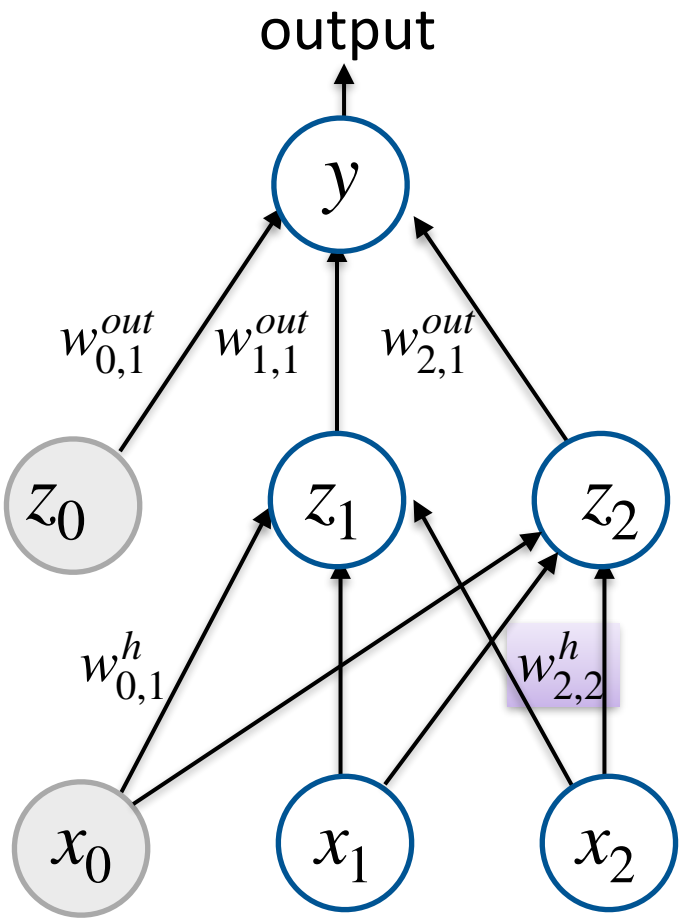
Because $z_2(s)$ is the logistic function we have already seen

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$

s



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial z_2}{\partial s} = z_2(1 - z_2)$$

$$\frac{\partial s}{\partial w^h_{2,2}} = x_2$$

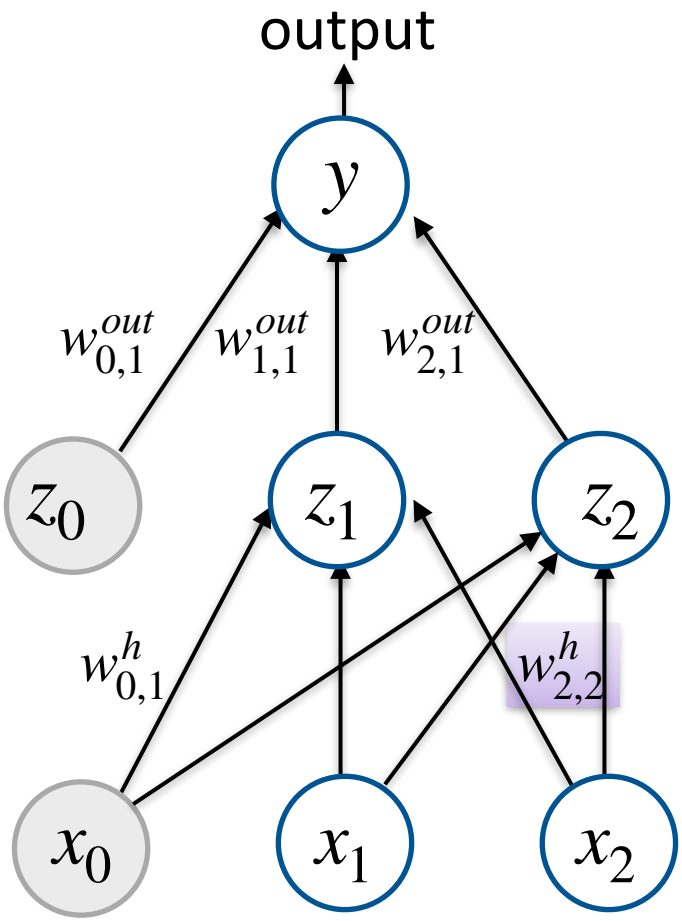
Because $z_2(s)$ is the logistic function we have already seen

Backpropagation: hidden layer

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

$$z_2 = \sigma(w^h_{0,2} + w^h_{1,2}x_1 + w^h_{2,2}x_2)$$

s



$$\frac{\partial L}{\partial w^h_{2,2}} = \frac{\partial L}{\partial y} \cdot w^o_{2,1} \frac{\partial z_2}{\partial s} \frac{\partial s}{\partial w^h_{2,2}}$$

$$\frac{\partial L}{\partial y} = y - y^*$$

$$\frac{\partial z_2}{\partial s} = z_2(1 - z_2)$$

$$\frac{\partial s}{\partial w^h_{2,2}} = x_2$$

Because $z_2(s)$ is the logistic function we have already seen

Importantly: we have already computed many of these partial derivatives because we are proceeding from top to bottom (i.e., backwards)

Backpropagation algorithm

The same algorithm works for multiple layers and more complicated architectures

Repeated application of the chain rule for partial derivatives

- First perform forward pass from inputs to the output
- Compute loss
- From loss, proceed backwards to compute partial derivatives using chain rule
- Cache partial derivatives as you compute them to use for lower layers

Mechanizing learning

Backpropagation gives you the gradient that will be used for gradient descent

- SGD gives us a generic learning algorithm
- Backpropagation is a generic method for computing partial derivatives

A recursive algorithm that proceeds from top of neural network to bottom

Modern neural network libraries implement automatic differentiation using back propagation

- Allows easy exploration of network architectures
- Don’t have to keep deriving the gradients by hand each time

Stochastic gradient descent

$$\min_{w} \sum_{i} L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$$

The objective is not convex
Initialization can be important

Given a training set $S = \{(\mathbf{x_i}, y_i)\}, \mathbf{x} \in \Re^{\mathbf{d}}$

1. Initialize parameters $\mathbf{w}$
2. For epoch $= 1 \ldots T$:
   - Shuffle the training set
   - For each training example $(\mathbf{x_i}, y_i) \in S$:
     - ➡ Treat this example as the entire dataset

       Compute the gradient of the loss $\nabla L(NN(\mathbf{x_i}, \mathbf{w}), y_i)$ using backpropagation

     - ➡ Update $\mathbf{w} \leftarrow \mathbf{w} - \gamma_t \nabla L(NN(\mathbf{x}_i, \mathbf{w}), y_i)$

γ_t: learning rate, many tweaks possible

Return $\mathbf{w}$