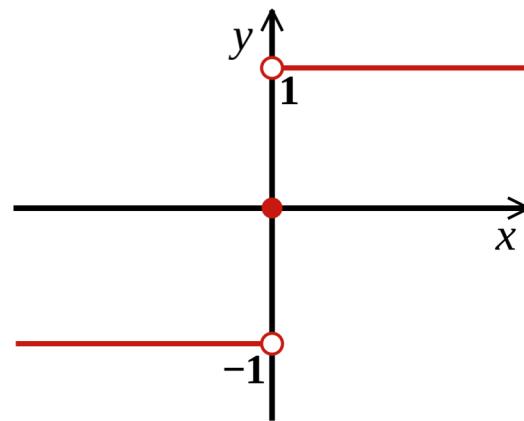
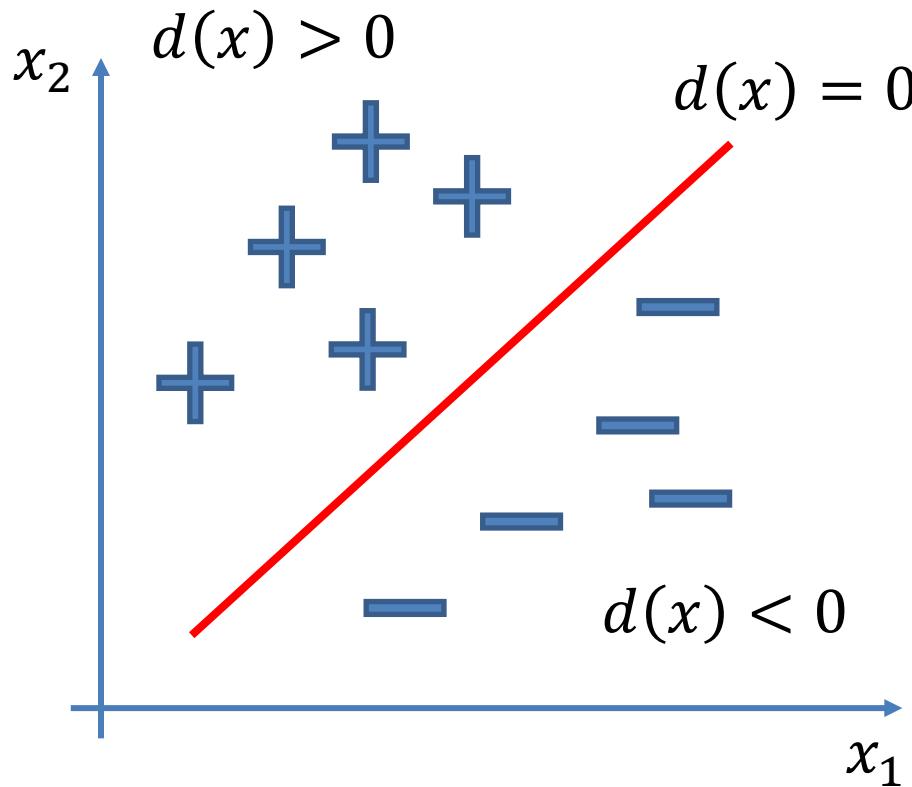


Intro

- In this video we will talk about the simplest neural network – multi-layer perceptron (MLP)

Let's recall linear binary classification

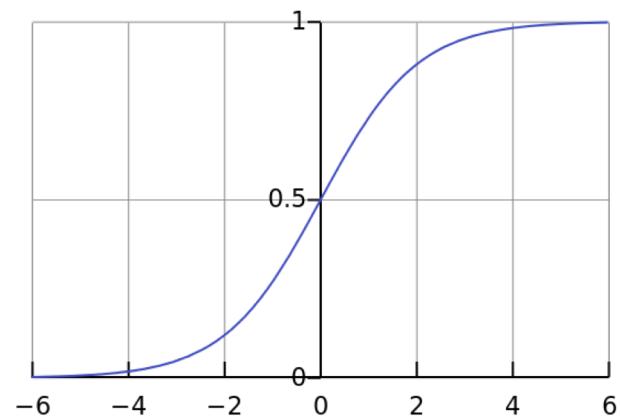
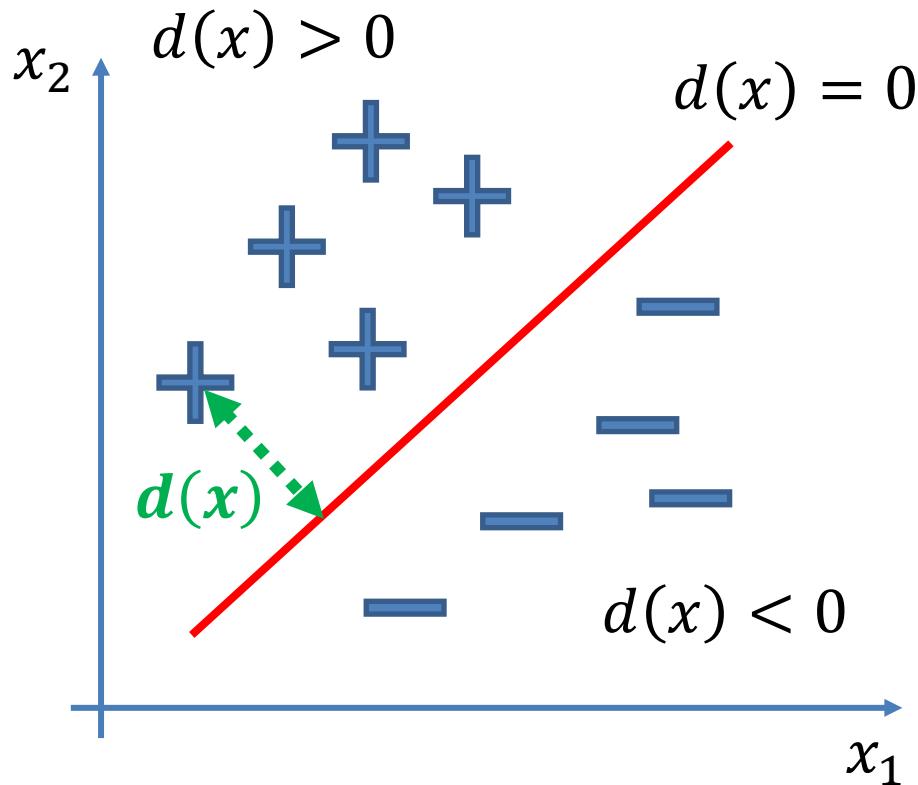
- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$
- Decision function: $d(x) = w_0 + w_1 x_1 + w_2 x_2$
- Algorithm: $a(x) = \text{sign}(d(x))$



$\text{sign}(x)$

Logistic regression

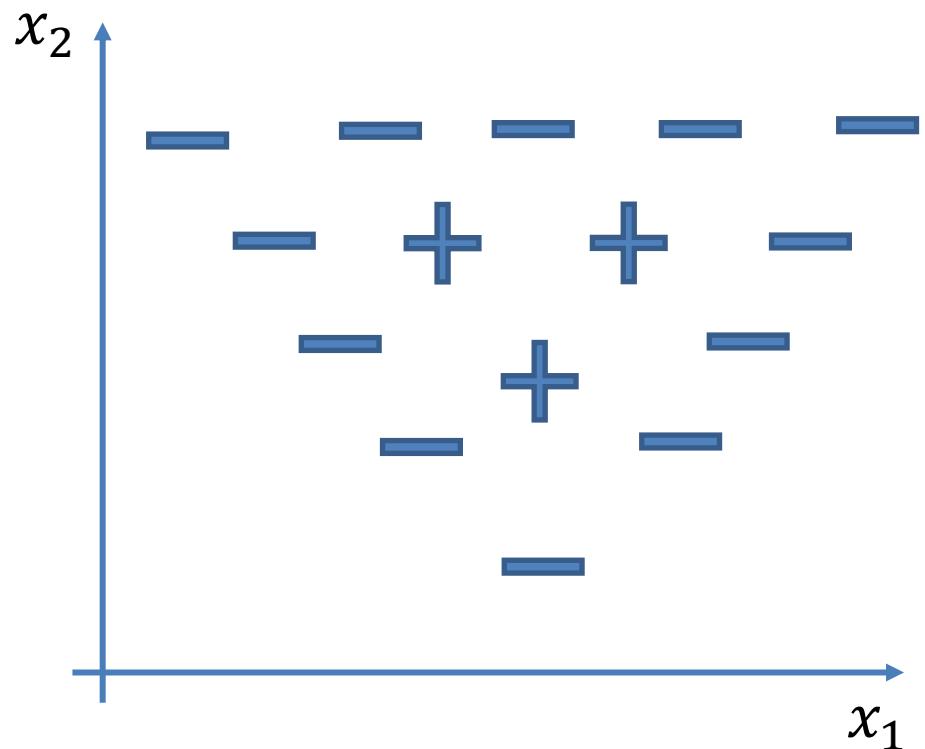
- Predicts probability of the positive class (+1)
- Decision function: $d(x) = w_0 + w_1x_1 + w_2x_2$
- Algorithm: $a(x) = \sigma(d(x))$



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

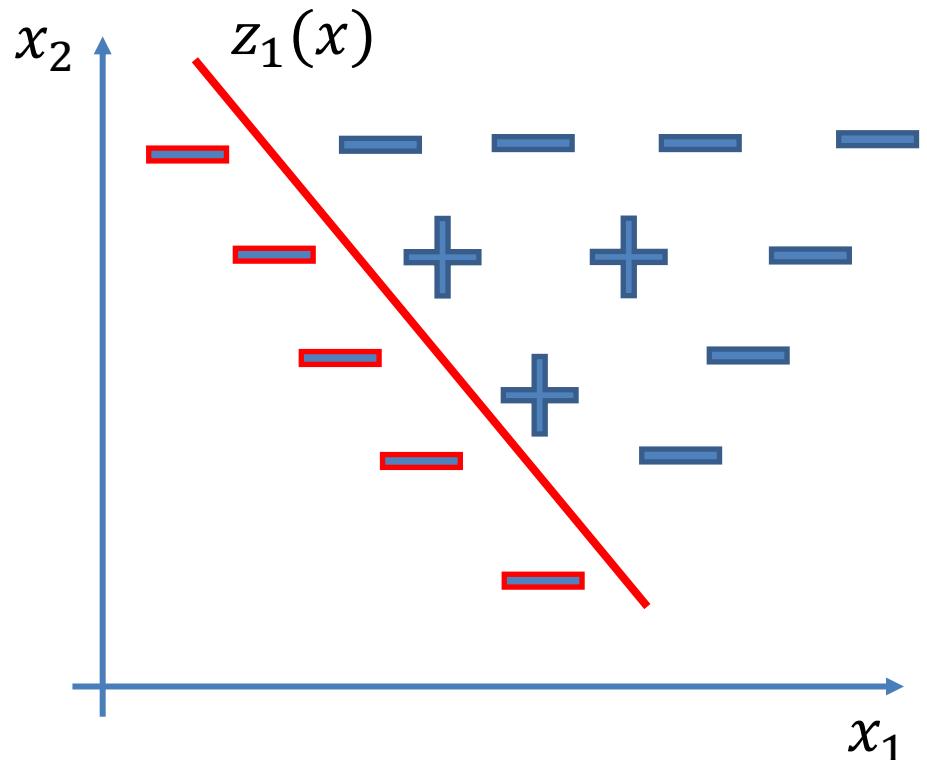
Triangle problem

- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$



Triangle problem: solving a subproblem

- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$

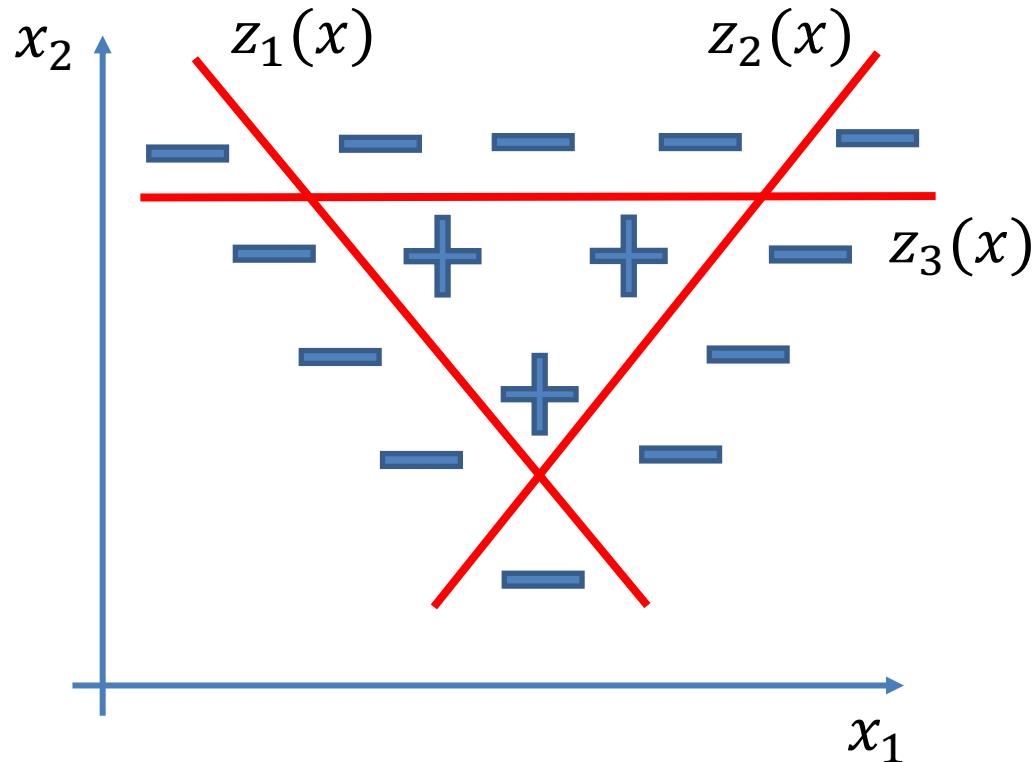


This line helps us to separate minuses on the left

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

A logistic regression per a subproblem

- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$

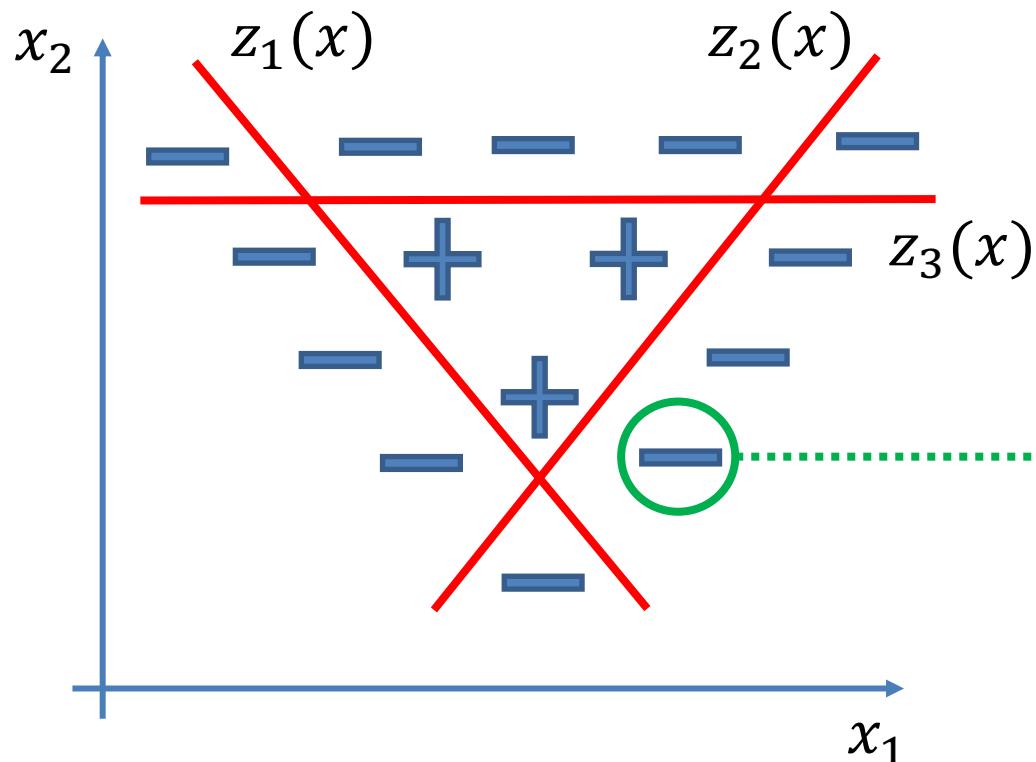


Imagine that we've somehow found these 3 lines

$$z_i = \sigma(w_{0,i} + w_{1,i}x_1 + w_{2,i}x_2)$$

These lines give us new features

- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$



New features:

$z_1(x)$	$z_2(x)$	$z_3(x)$	y
0.6	0.3	0.8	-1
0.7	0.7	0.7	+1

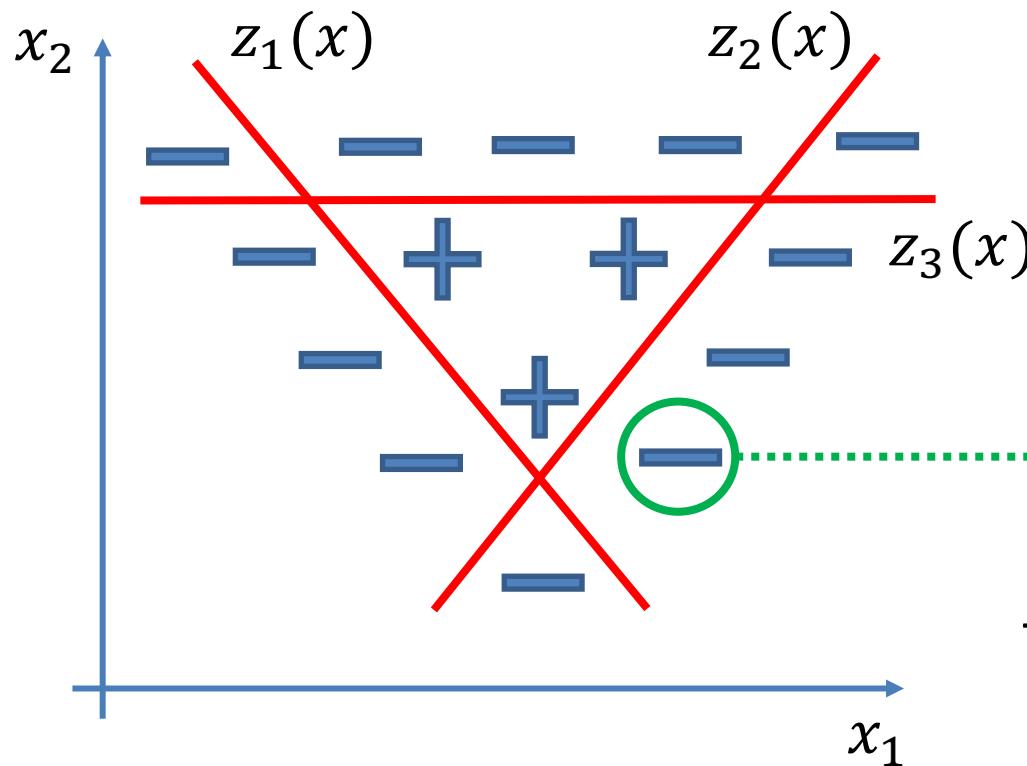
$$(x_1, x_2) \rightarrow (z_1, z_2, z_3)$$

What to do next?

$$z_i = \sigma(w_{0,i} + w_{1,i}x_1 + w_{2,i}x_2)$$

Final algorithm

- Features: $x = (x_1, x_2)$
- Target: $y \in \{+1, -1\}$



$$z_i = \sigma(w_{0,i} + w_{1,i}x_1 + w_{2,i}x_2)$$

New features:

$z_1(x)$	$z_2(x)$	$z_3(x)$	y
0.6	0.3	0.8	-1
0.7	0.7	0.7	+1

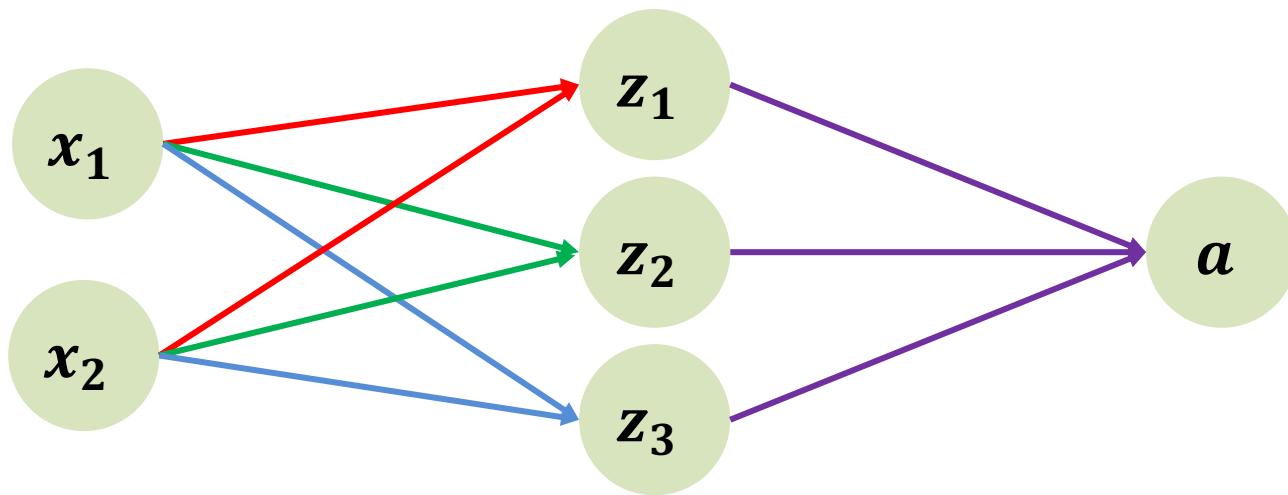
$$(x_1, x_2) \rightarrow (z_1, z_2, z_3)$$

Let's build a linear model on top of these new features:

$$a(x) = \sigma(w_0 + w_1 z_1(x) + w_2 z_2(x) + w_3 z_3(x))$$

We still don't know how to find these 4 lines

- But we know how our algorithm will predict once we find them:
 - $z_i = \sigma(w_{0,i} + w_{1,i}x_1 + w_{2,i}x_2)$
 - $a(x) = \sigma(w_0 + w_1z_1(x) + w_2z_2(x) + w_3z_3(x))$
- Let's rewrite our algorithm in terms of a **computation graph**:

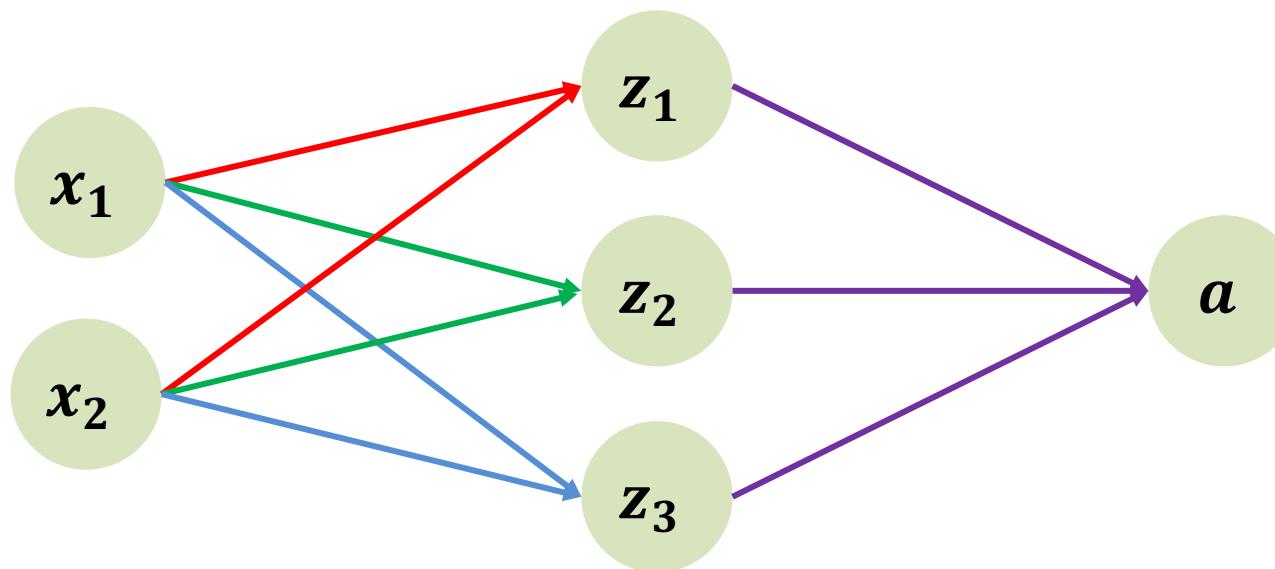


Nodes: computed variables ($x_1, x_2, z_1, z_2, z_3, a$)

Edges: dependencies (we need x_1 and x_2 to compute z_1)

Our computation graph has a name

- Multi-layer perceptron (MLP):



Input layer

Features

Hidden layer

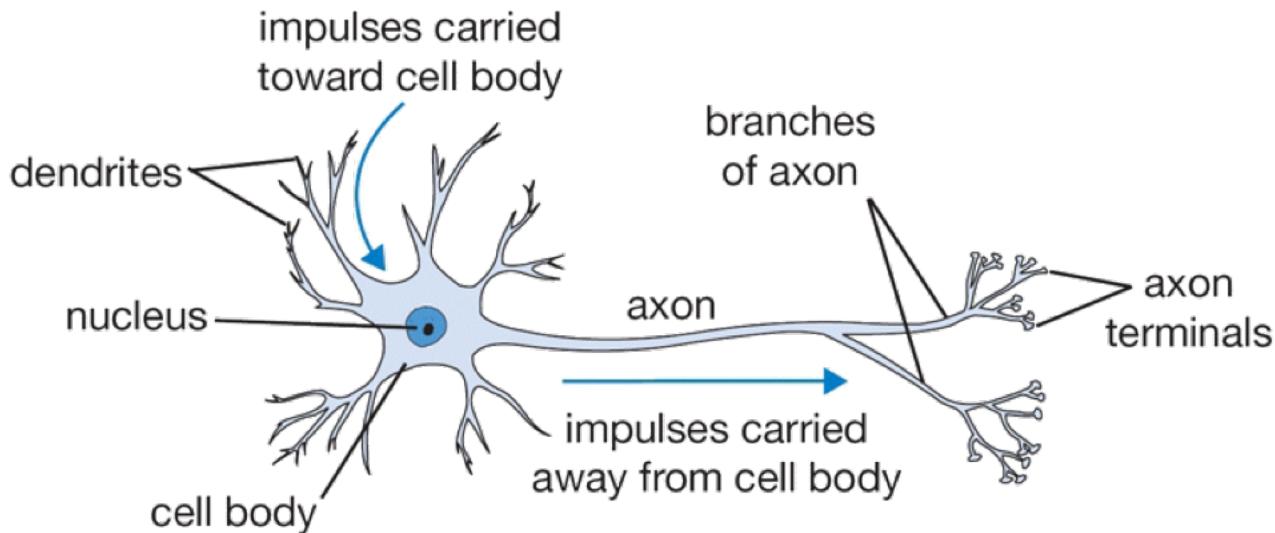
Here each node is a **neuron**:

1. Take a linear combination of inputs
2. Apply **activation** function (e.g. $\sigma(x)$)

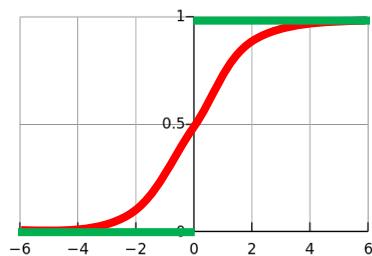
Output layer

Why is it called a neuron?

- Neuron in a human brain:

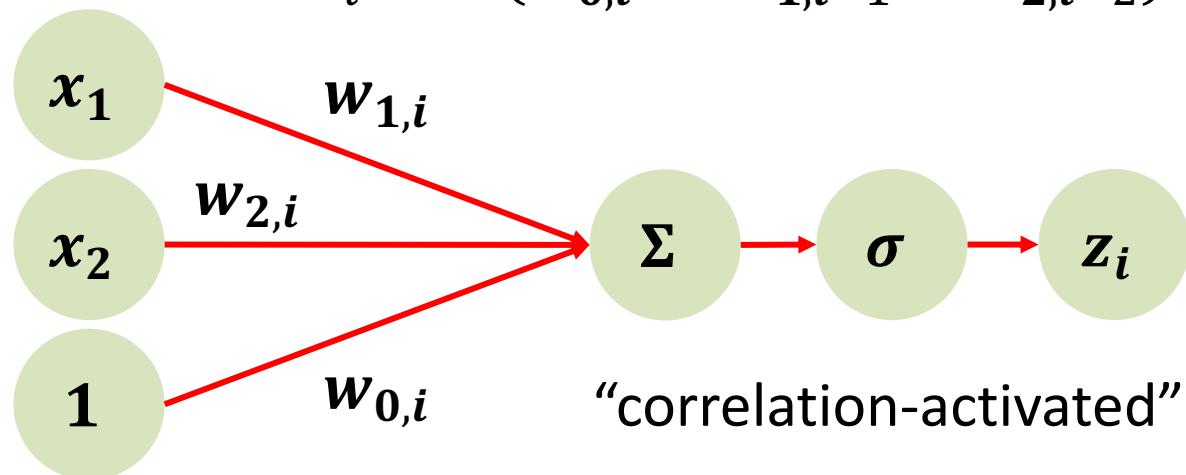


- Artificial neuron:



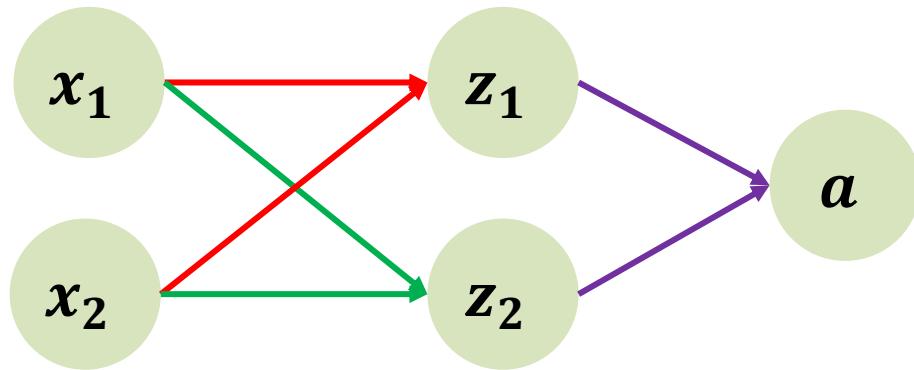
“smooth
indicator”

$$z_i = \sigma(w_{0,i} + w_{1,i}x_1 + w_{2,i}x_2)$$



We need a non-linear activation function!

- Let's see what happens if we throw away $\sigma(x)$:



$$z_1 = w_{1,1}x_1 + w_{2,1}x_2$$

$$z_2 = w_{1,2}x_1 + w_{2,2}x_2$$

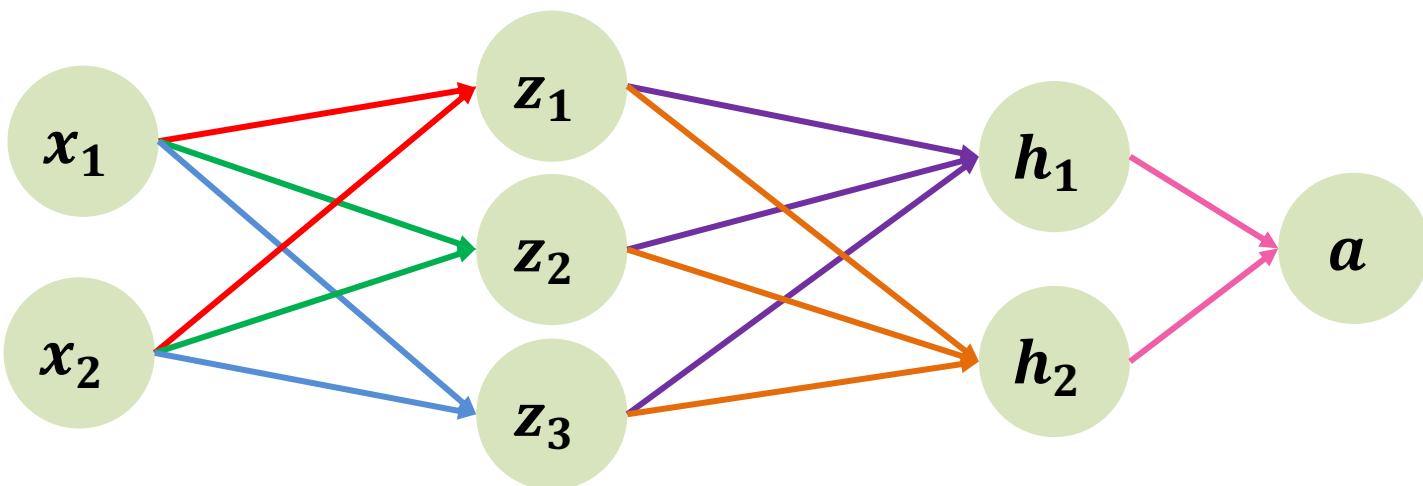
$$a = w_1z_1 + w_2z_2$$

- Our algorithm turns into a fancy **linear function!**

- $$a = (w_1w_{1,1} + w_2w_{1,2})x_1 + (w_1w_{2,1} + w_2w_{2,2})x_2$$

MLP overview

- MLP is an example of **artificial neural network**
- MLP can have many hidden layers:



- **Architecture** of an MLP:
 - Number of layers
 - Number of neurons in each layer
 - Activation function
- Hidden layer in MLP:
 - Dense layer
 - Fully-connected layer

How to train your MLP?

- We know how to train one neuron (e.g. logistic regression): SGD
- Let's do the same for the whole MLP!

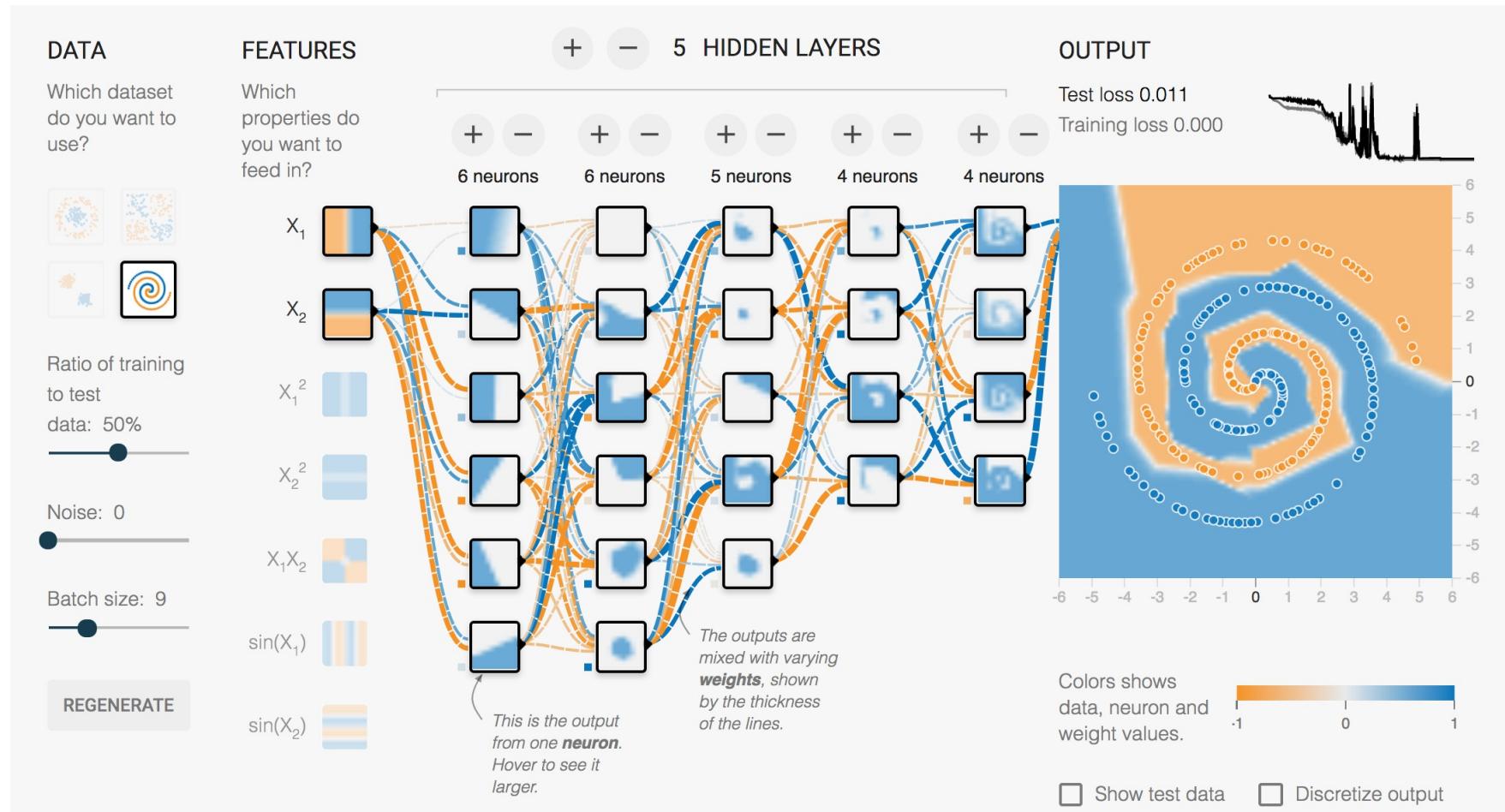


Bob chilling at a local optima

<https://hackernoon.com/life-is-gradient-descent-880c60ac1be8>

Check out MLP training demo

- <http://playground.tensorflow.org> (change activation to ReLU)



How to train your MLP?

- Other problems:
 - We can have many hidden layers (hyper-parameter) → we need to calculate gradients **automatically!**
 - We can have many neurons → we need to calculate gradients **fast!**
- In the next video we will solve these problems!

Intro

- In this video we will learn how to compute the gradients for MLP **automatically**.

Chain rule

- We know derivatives for simple functions:

$$\frac{dx^2}{dx} = 2x \quad \frac{de^x}{dx} = e^x \quad \frac{d\ln(x)}{dx} = \frac{1}{x}$$

- Let's take a composite function:

$$z_1 = z_1(\textcolor{red}{x}_1, x_2)$$

$$z_2 = z_2(\textcolor{green}{x}_1, x_2) \quad \text{where } z_1, z_2, p \text{ are differentiable}$$

$$p = p(\textcolor{violet}{z}_1, \textcolor{brown}{z}_2)$$

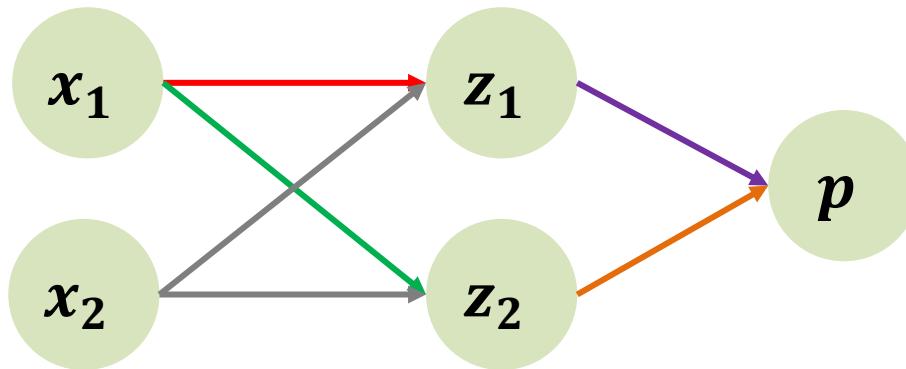
Chain rule: $\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2} \frac{\partial z_2}{\partial x_1}$

Example for $h(x) = f(x)g(x)$:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial f} \frac{\partial f}{\partial x} + \frac{\partial h}{\partial g} \frac{\partial g}{\partial x} = \textcolor{blue}{g} \frac{\partial f}{\partial x} + \textcolor{pink}{f} \frac{\partial g}{\partial x}$$

Derivatives computation graph

- Let's take our simple computation graph:

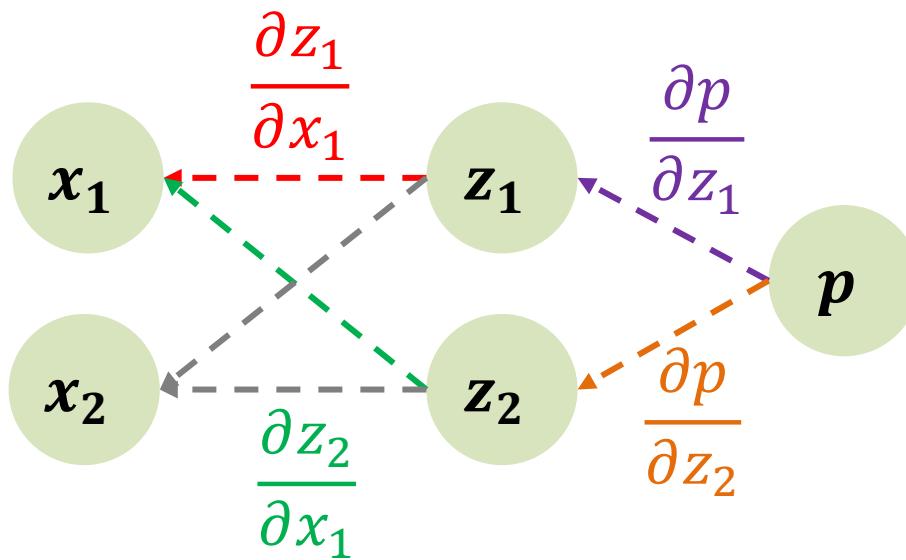


$$z_1 = z_1(x_1, x_2)$$

$$z_2 = z_2(x_1, x_2)$$

$$p = p(z_1, z_2)$$

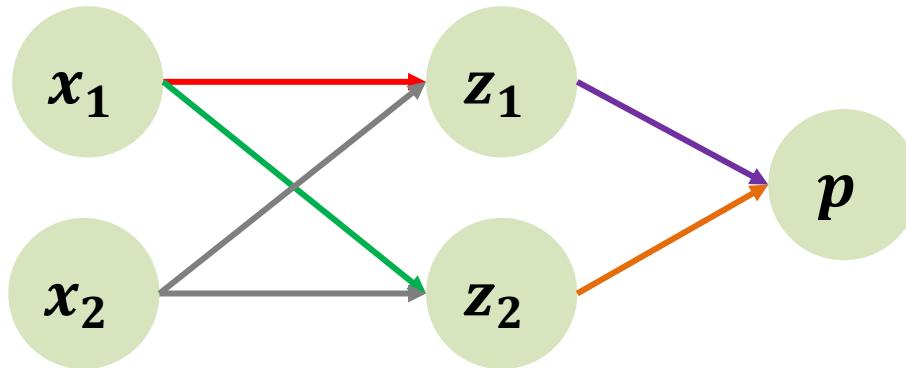
- And construct a new graph of derivatives:



Each edge is assigned
to derivative of origin
w.r.t. destination

Derivatives computation graph

- Let's take our simple computation graph:

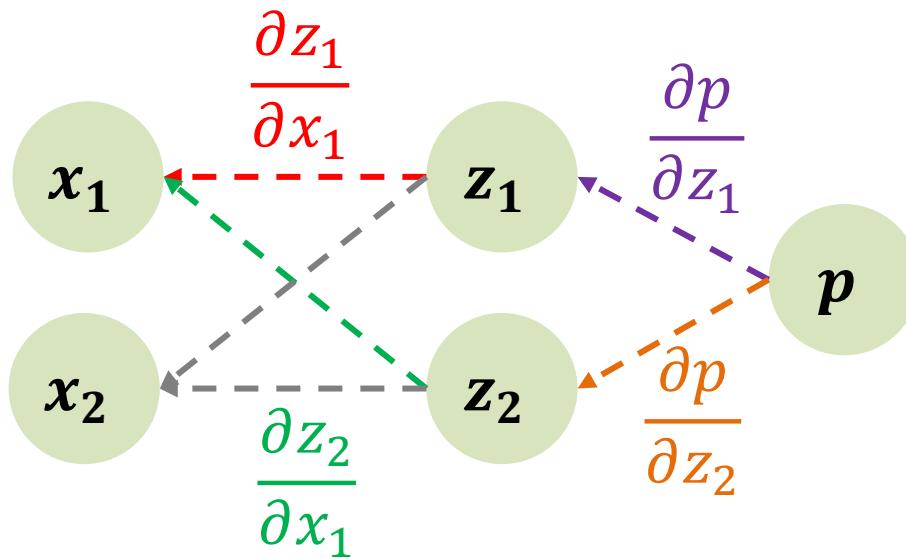


$$z_1 = z_1(x_1, x_2)$$

$$z_2 = z_2(x_1, x_2)$$

$$p = p(z_1, z_2)$$

- And construct a new graph of derivatives:

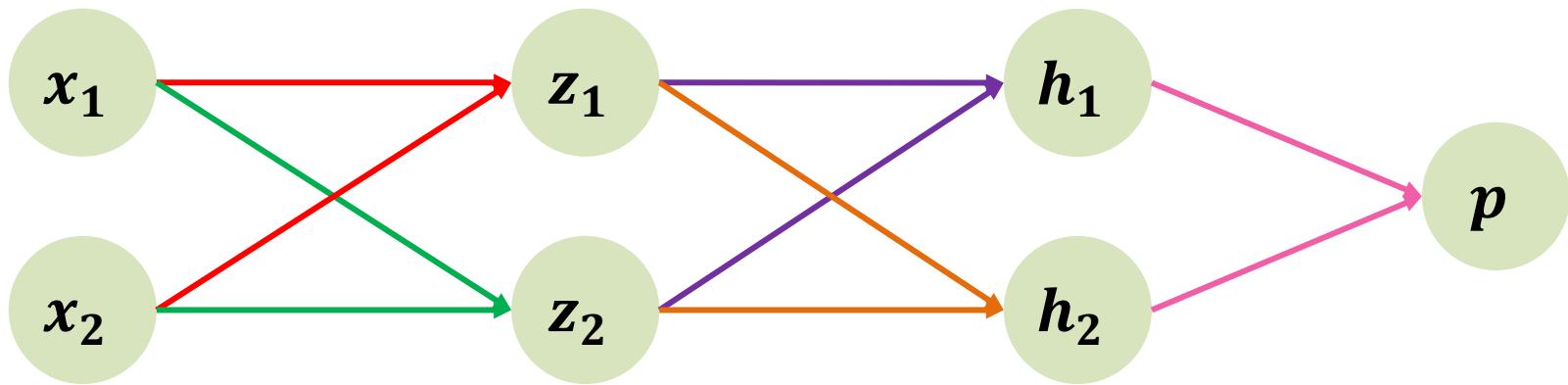


$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

You can see how
a **chain rule** works

Let's go deeper

- A little bit more composite function:



$$z_1 = z_1(x_1, x_2) \quad h_1 = h_1(z_1, z_2)$$

$$z_2 = z_2(x_1, x_2) \quad h_2 = h_2(z_1, z_2)$$

$$p = p(h_1, h_2)$$

Let's go deeper

Chain rule: $\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial x_1}$

Let's go deeper

Chain rule:

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial h_1}{\partial x_1} = \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial h_2}{\partial x_1} = \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \boxed{\frac{\partial h_1}{\partial x_1}} + \frac{\partial p}{\partial h_2} \boxed{\frac{\partial h_2}{\partial x_1}}$$

Let's go deeper

Chain rule:

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial x_1}$$

$$\frac{\partial h_1}{\partial x_1} = \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

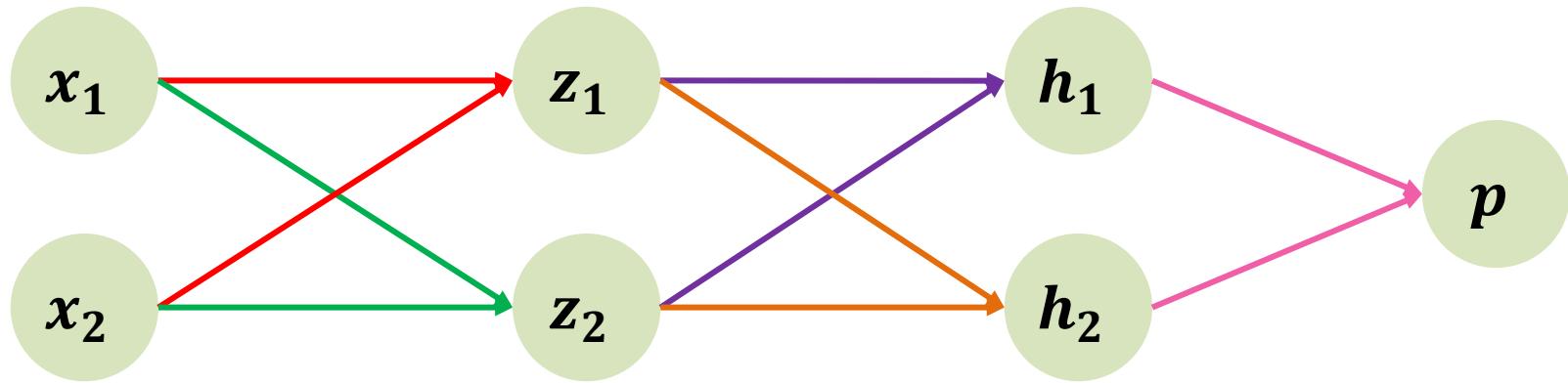
$$\frac{\partial h_2}{\partial x_1} = \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \left(\frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} \right) + \frac{\partial p}{\partial h_2} \left(\frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1} \right)$$

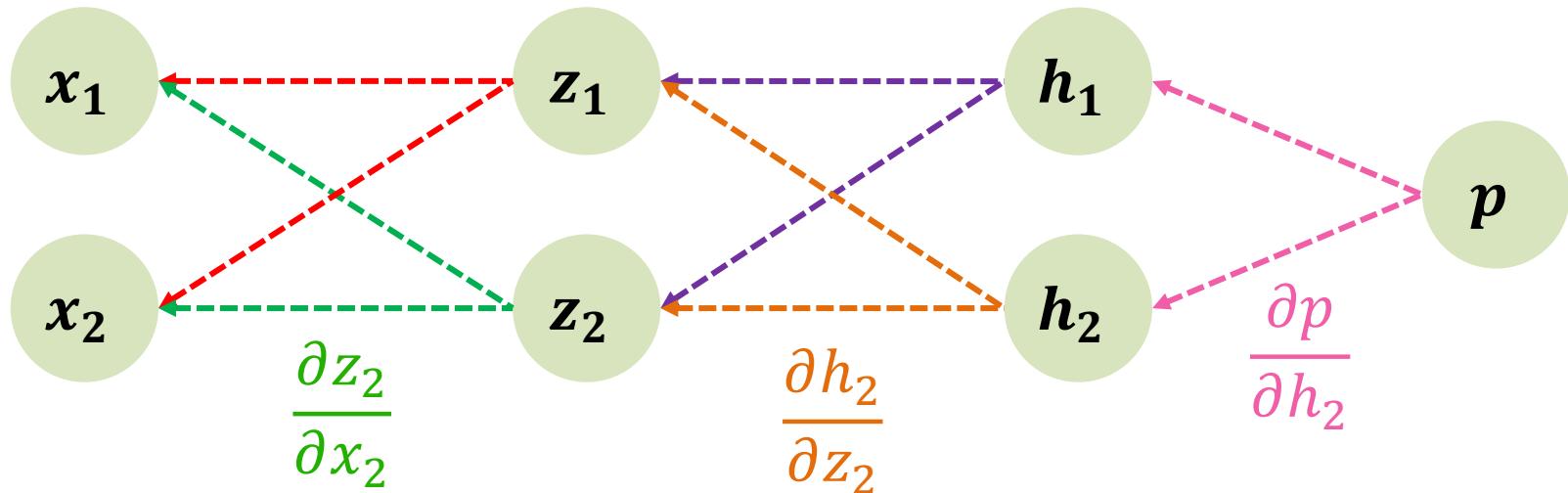
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

Let's check out the derivatives graph!

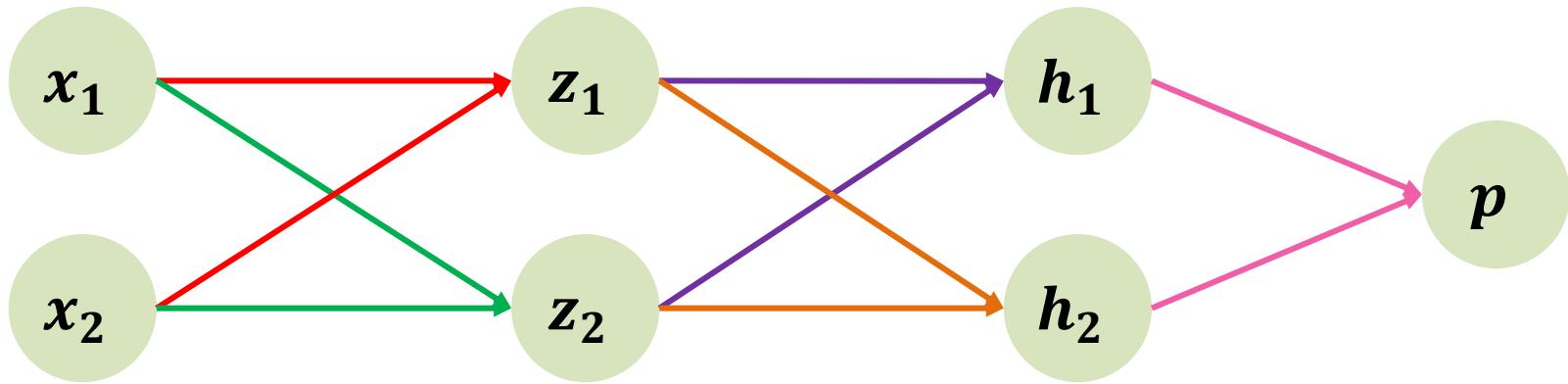
Let's go deeper



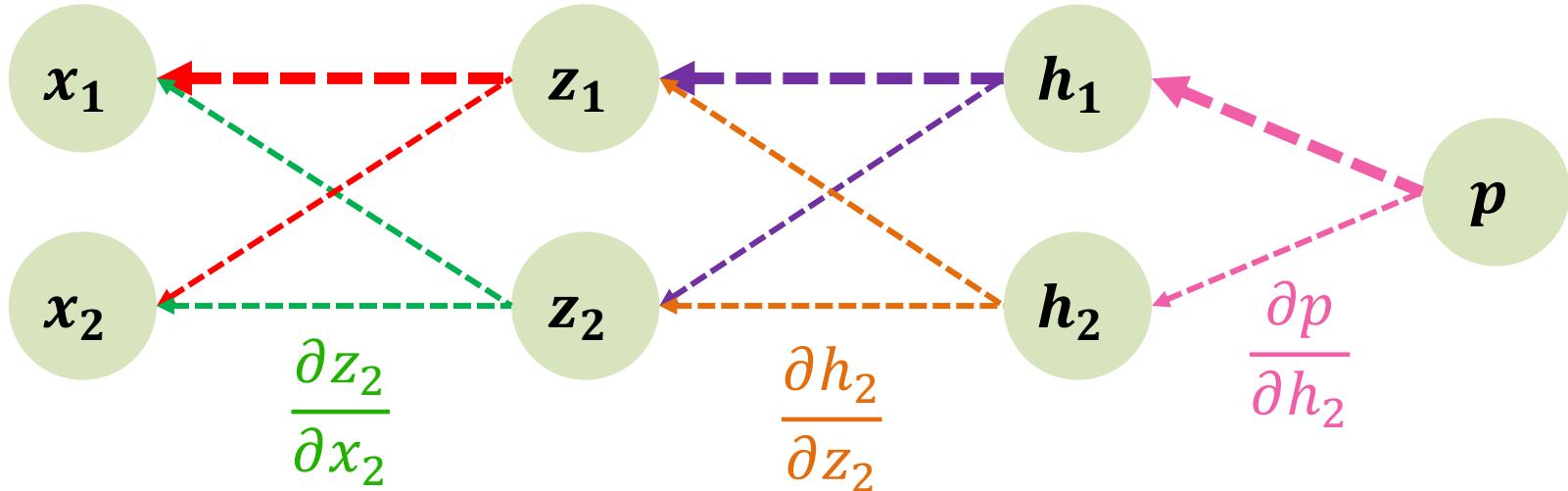
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$



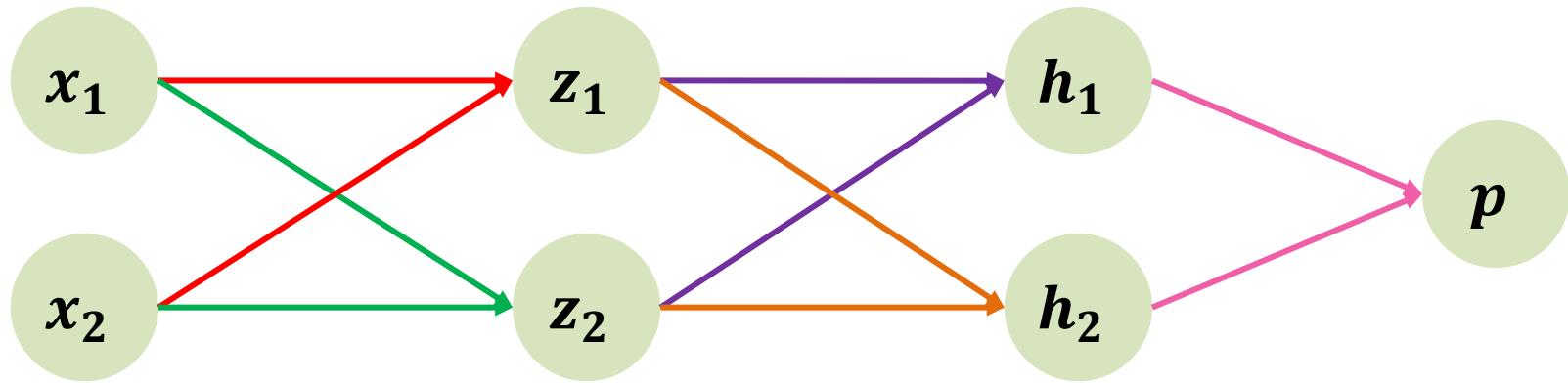
Let's go deeper



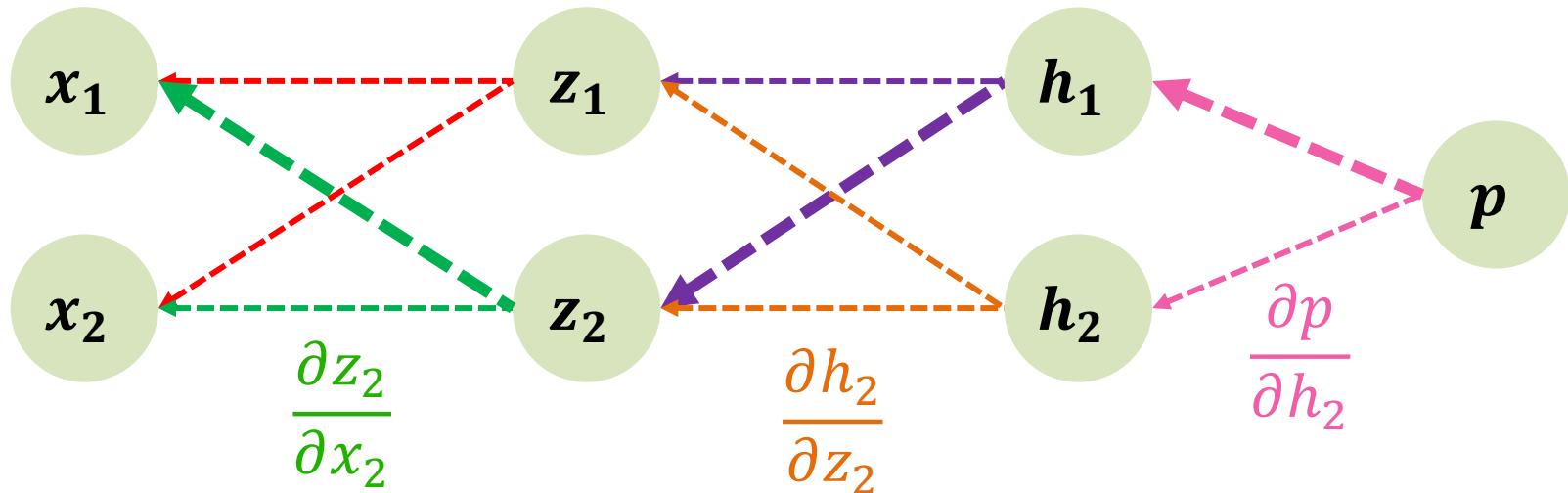
$$\frac{\partial p}{\partial x_1} = \boxed{\frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1}} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$



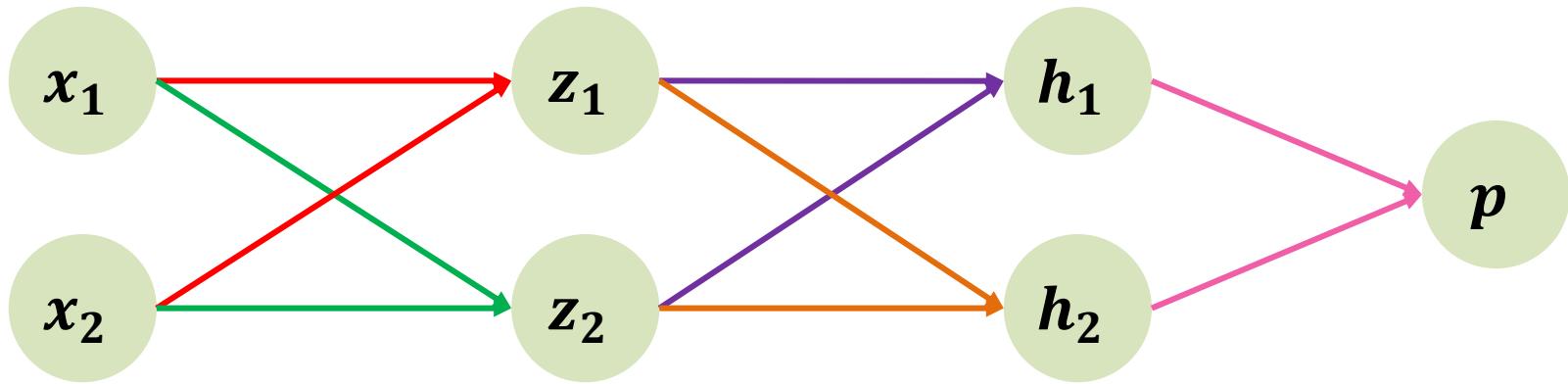
Let's go deeper



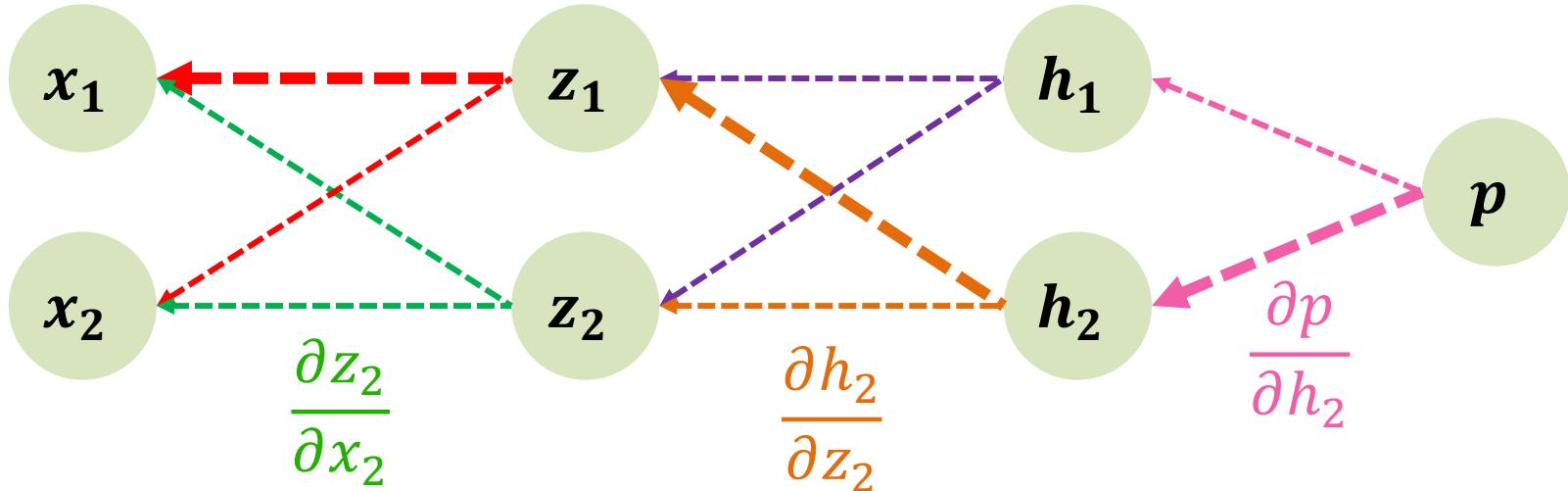
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \boxed{\frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1}} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$



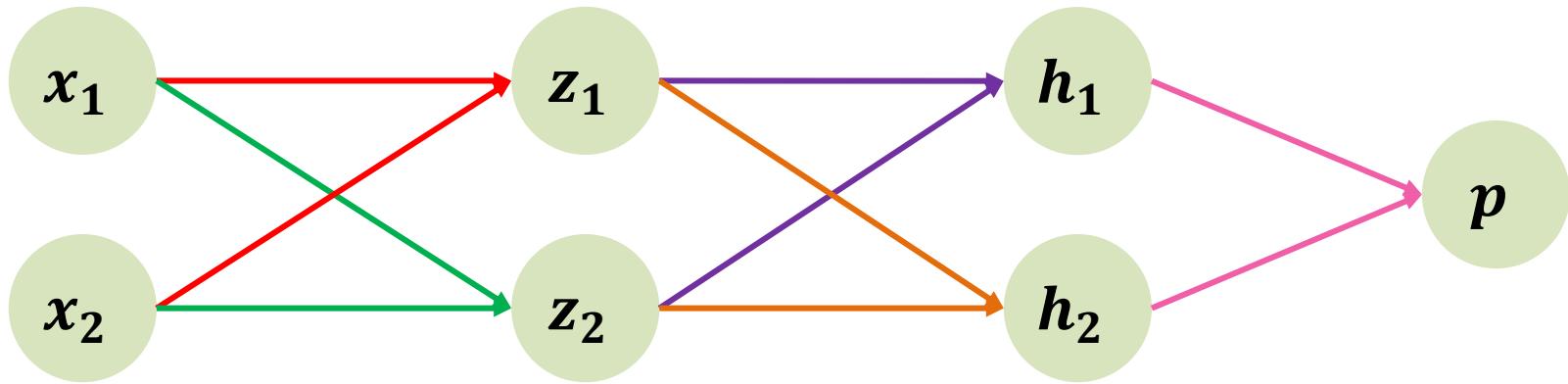
Let's go deeper



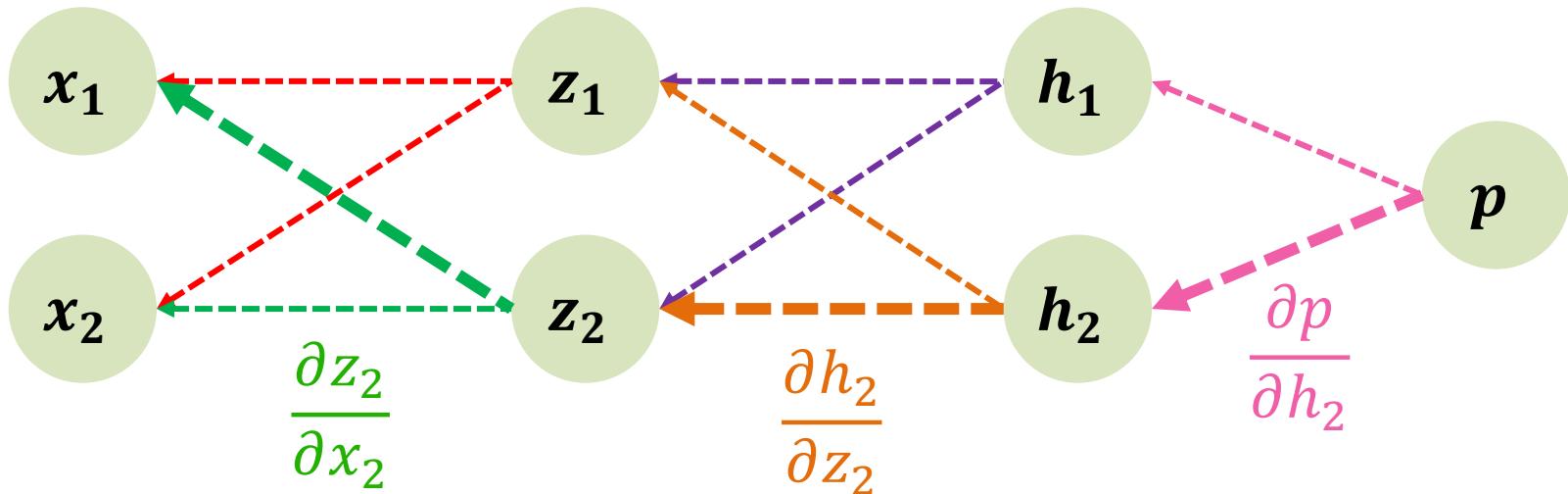
$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \boxed{\frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1}} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$



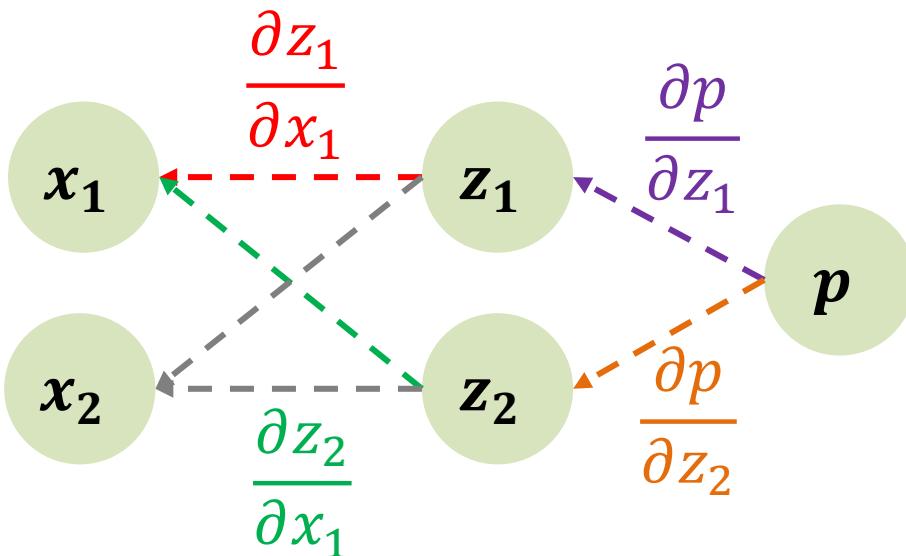
Let's go deeper



$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \boxed{\frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}}$$



How this graph of derivatives helps

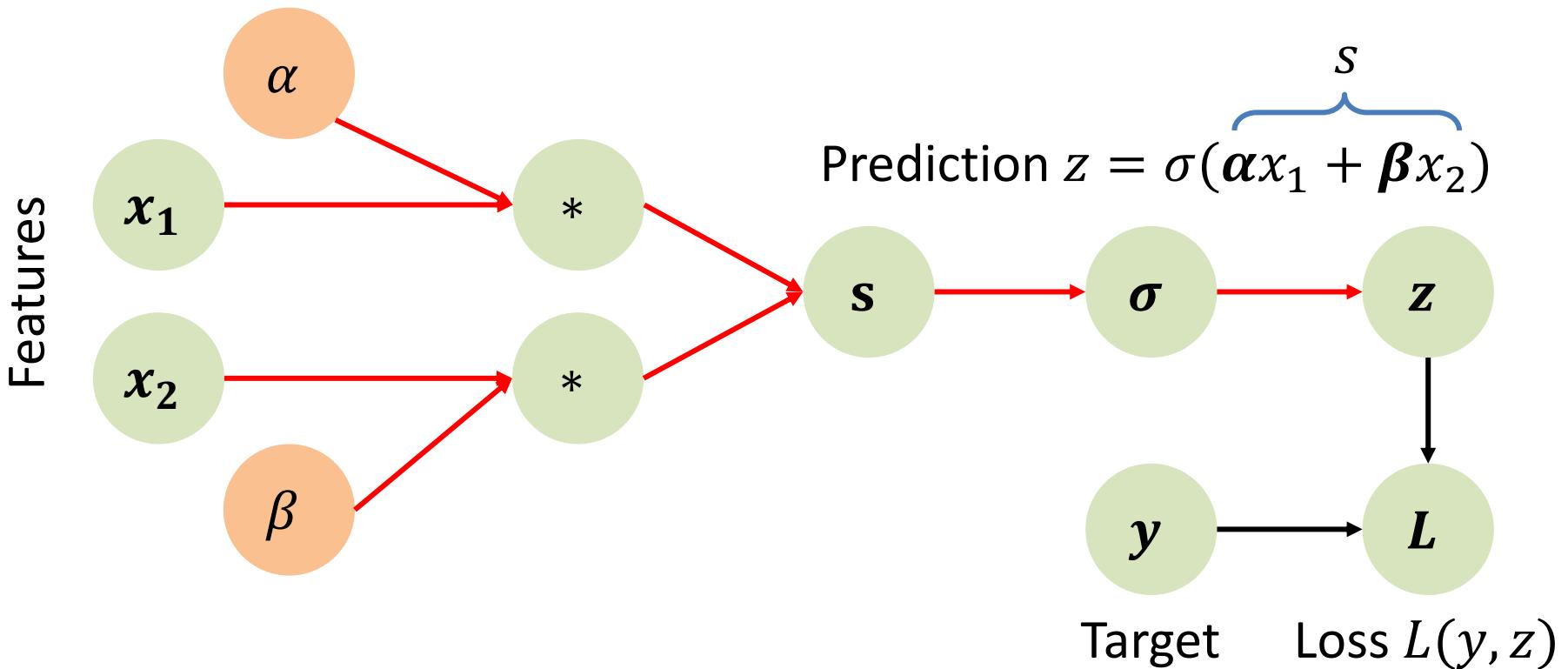


$$\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

How to calculate a derivative of node a w.r.t. node b :

- Find an unvisited path from a to b
- Multiply all edge values along this path
- Add to the resulting derivative

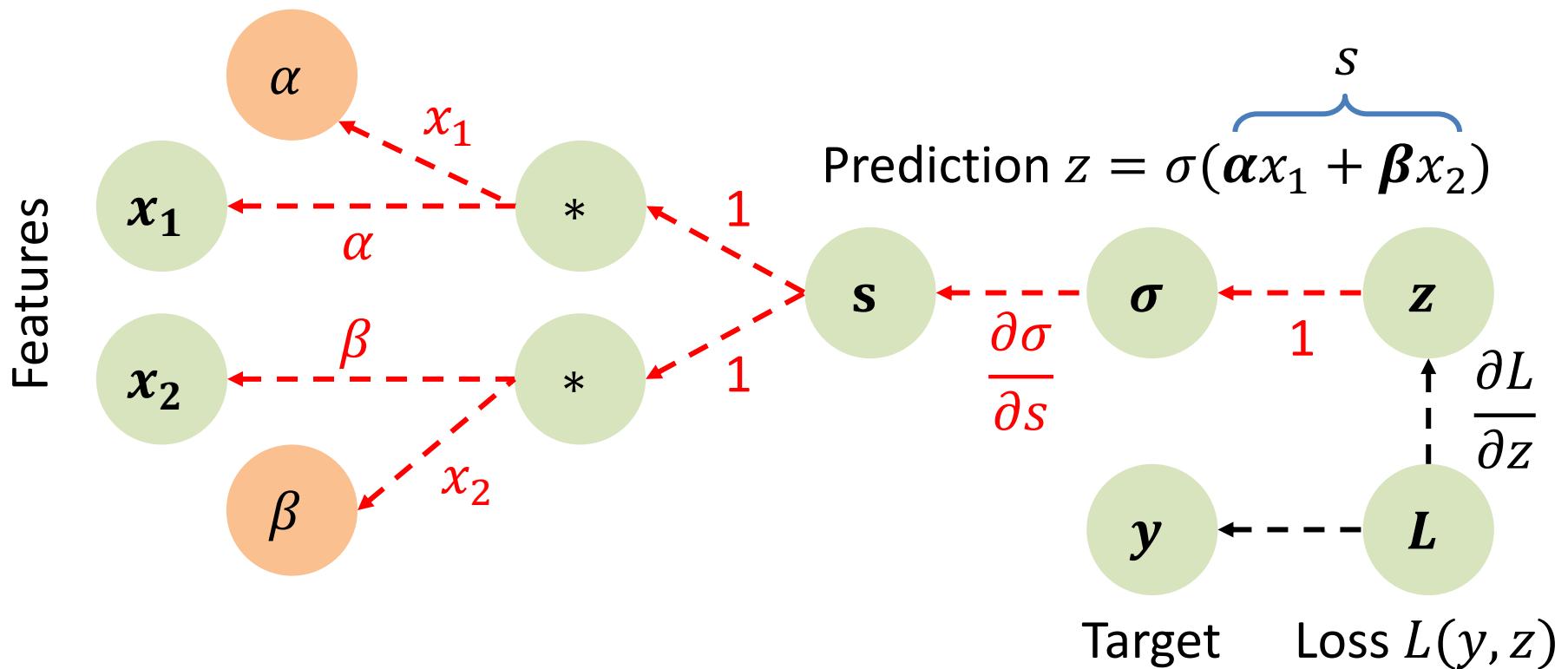
How chain rule helps to train a neuron



For SGD to work we need

$$\frac{\partial L}{\partial \alpha}, \frac{\partial L}{\partial \beta}$$

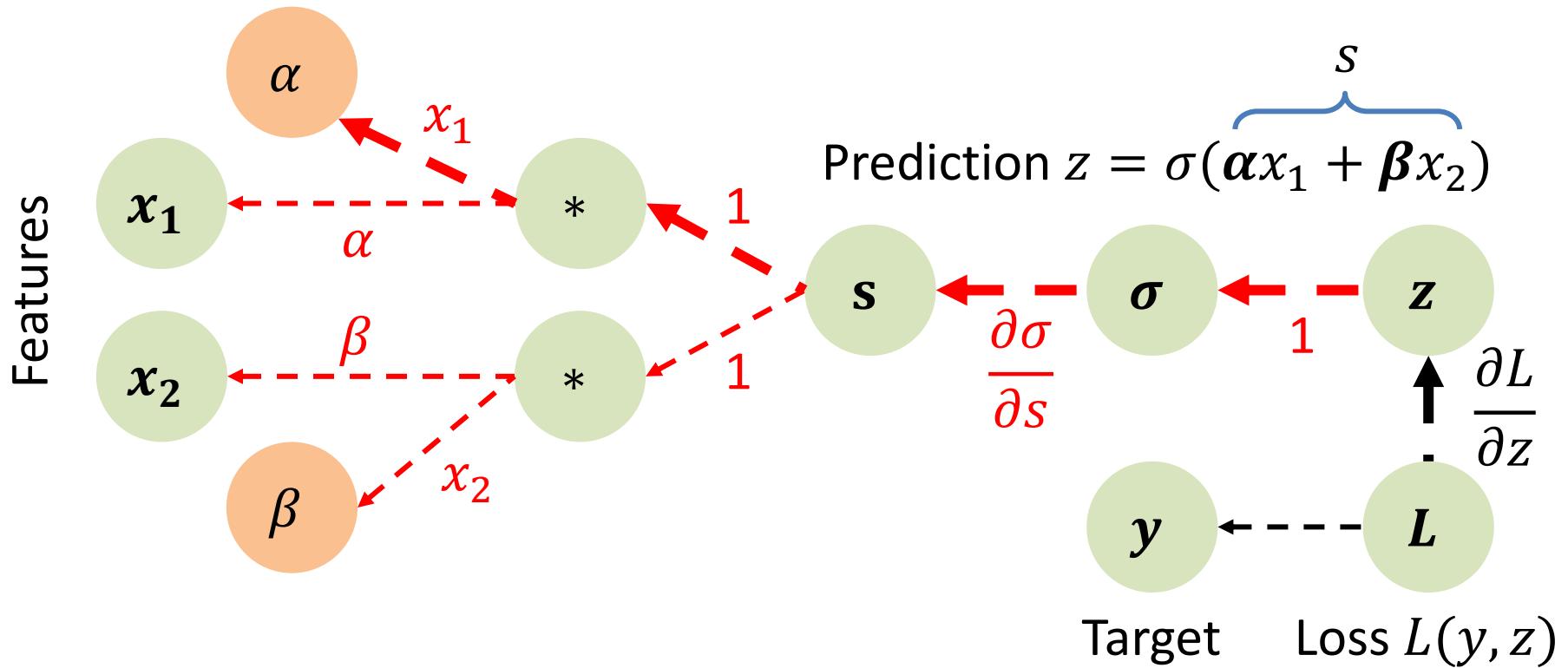
Derivatives computation graph



For SGD to work we need

$$\frac{\partial L}{\partial \alpha}, \frac{\partial L}{\partial \beta}$$

Derivatives computation graph

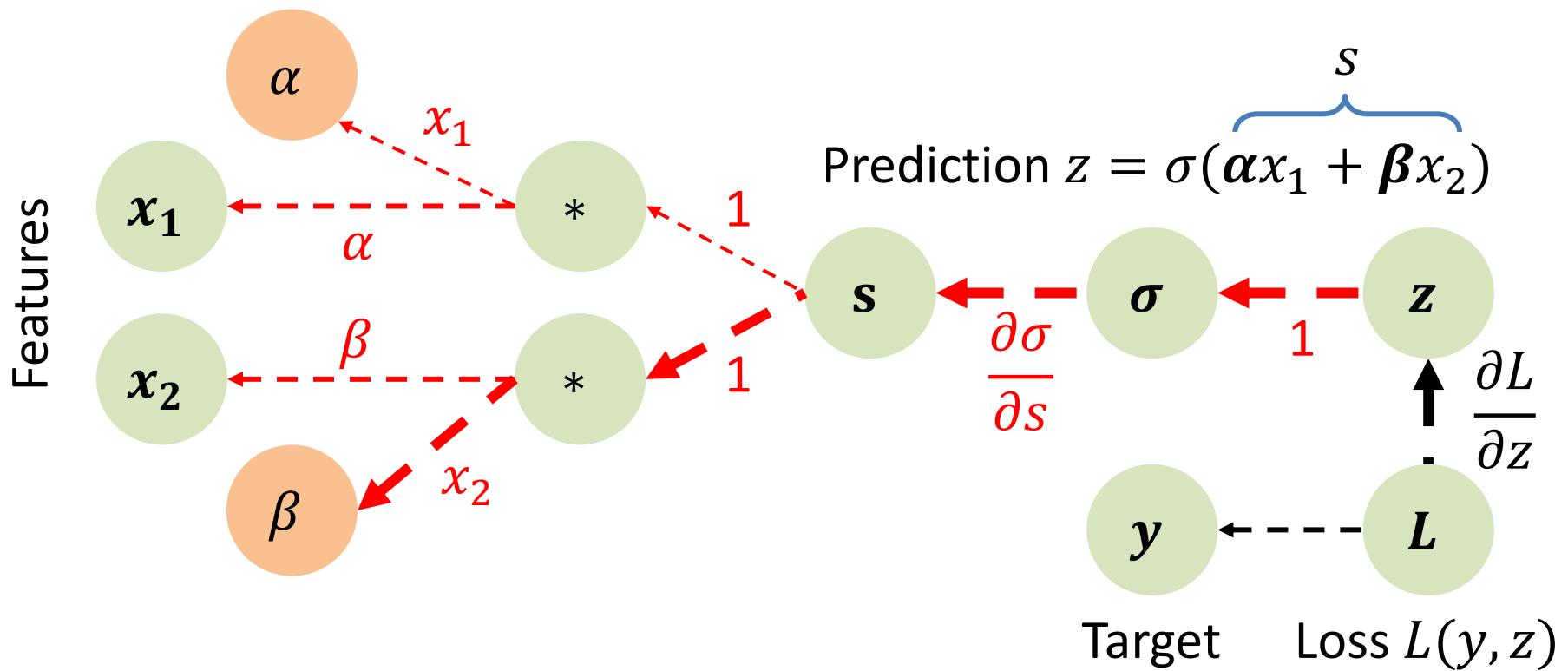


For SGD to work we need

$$\frac{\partial L}{\partial \alpha}, \frac{\partial L}{\partial \beta}$$

$$\frac{\partial L}{\partial \alpha} = \frac{\partial L}{\partial z} \frac{\partial \sigma}{\partial s} x_1$$

Derivatives computation graph



For SGD to work we need

$$\frac{\partial L}{\partial \alpha}, \frac{\partial L}{\partial \beta}$$

$$\frac{\partial L}{\partial \alpha} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial s} x_1$$

$$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial s} x_2$$

Summary

- We can use chain rule to compute derivatives of composite functions
- We can use a computation graph of derivatives to compute them **automatically**
- In the next video we will find out how to do this **fast!**

Intro

- In this video we will find out how to apply a chain rule **efficiently!**

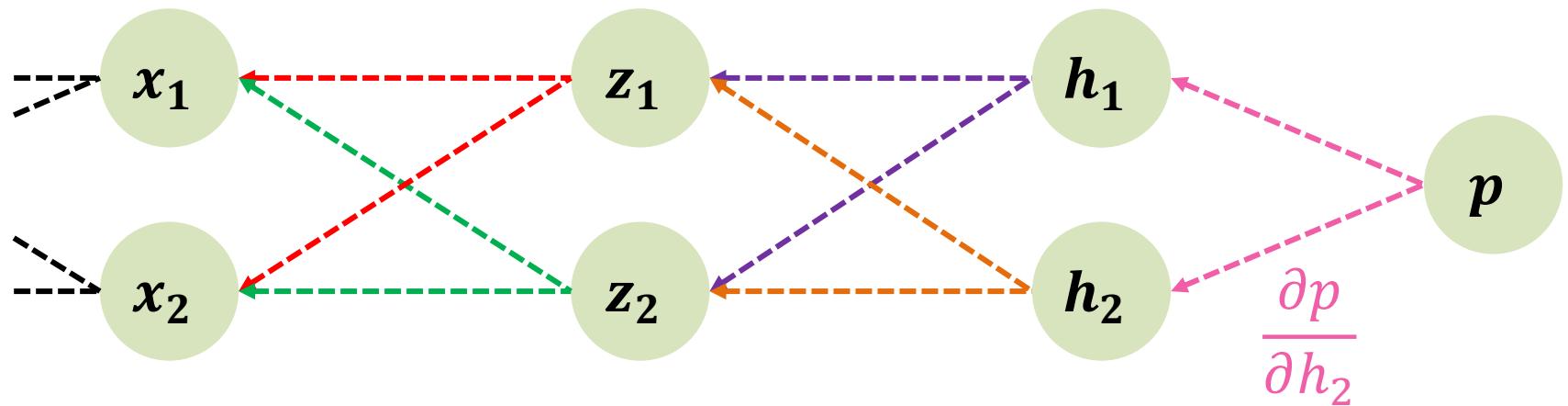
Let's look at MLP with 3 hidden layers

Let's drill down to an actual parameter of MLP:

$$h_2 = \sigma(w_0 + w_1 z_1 + w_2 z_2)$$

Gradient
Descent:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial w_1} = \frac{\partial L}{\partial p} \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial w_1}$$



We need to do this efficiently!

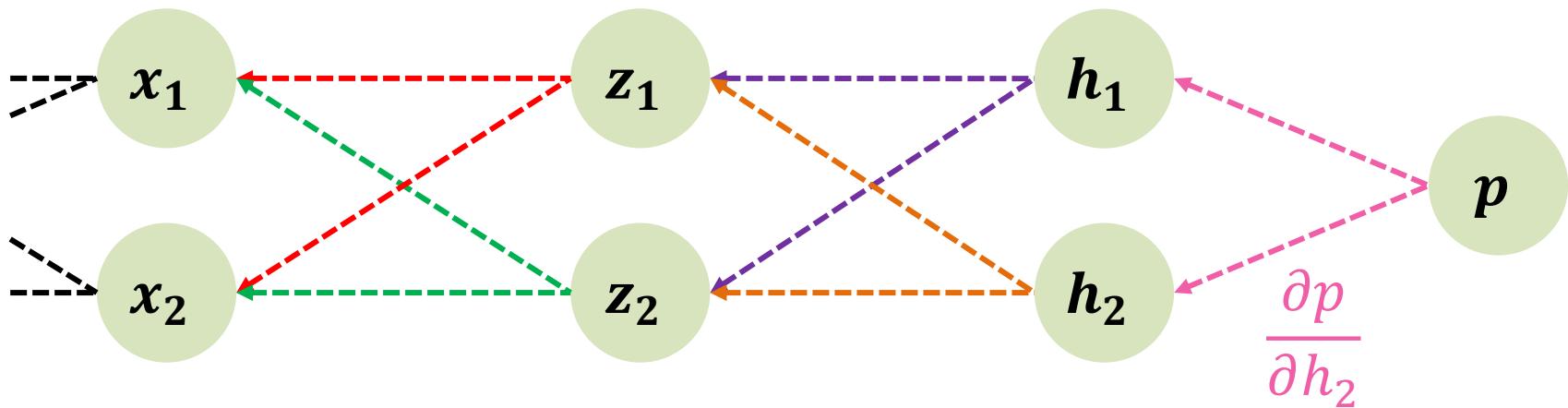
$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

We will need these for GD

$$2: \frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \quad \frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$1: \frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$



We can reuse previous computations

3:

$$\boxed{\frac{\partial p}{\partial h_1}}$$

$$\boxed{\frac{\partial p}{\partial h_2}}$$

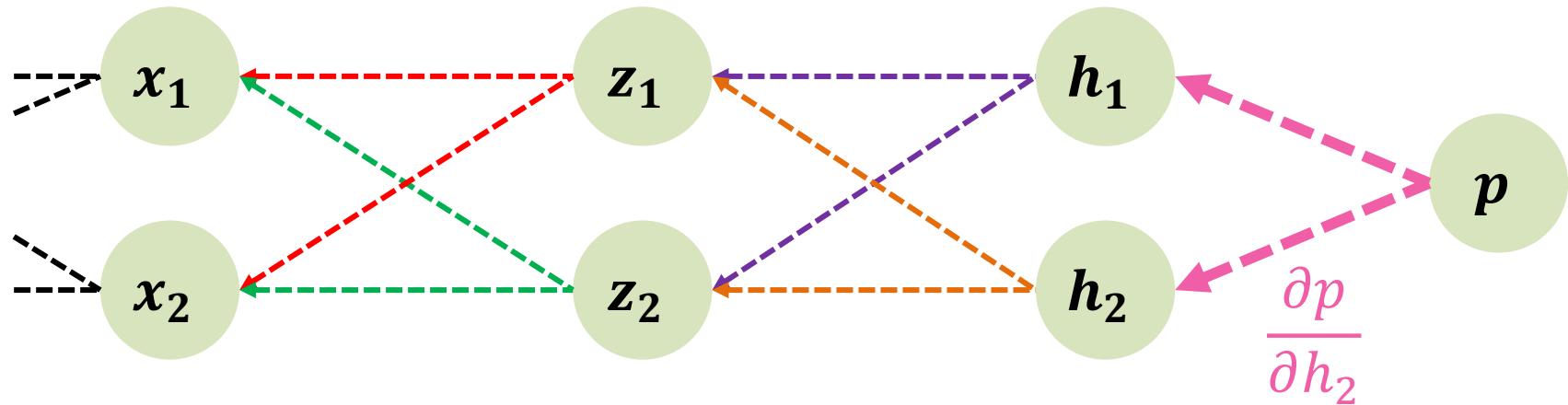
We will need these for GD

2: $\frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1}$

$$\frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

1: $\frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$

$\frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$



We can reuse previous computations

$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

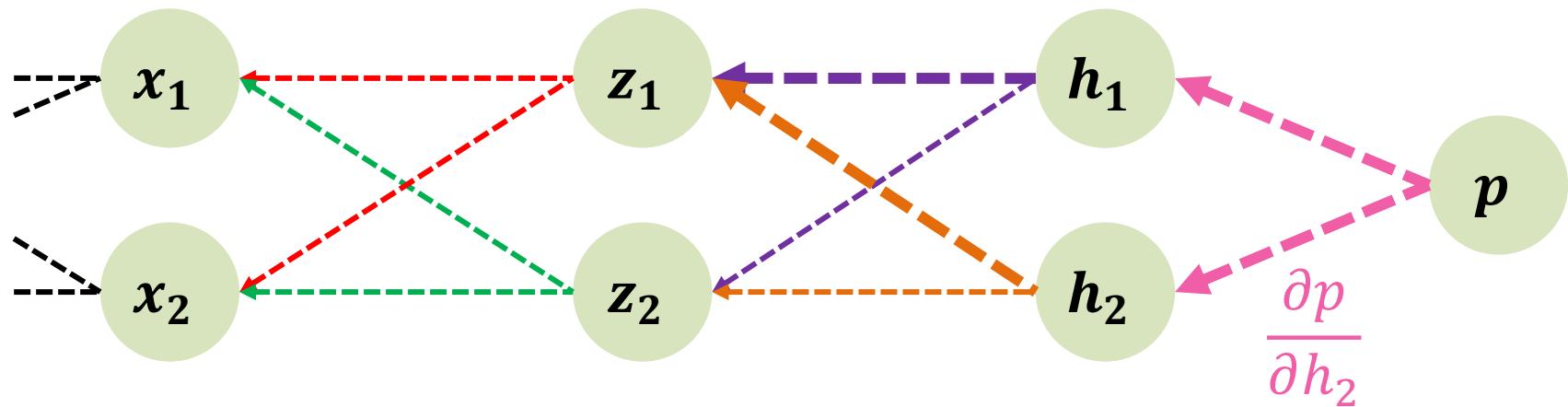
We will need these for GD

$$2: \boxed{\frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1}}$$

$$\frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$1: \frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$



We can reuse previous computations

$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

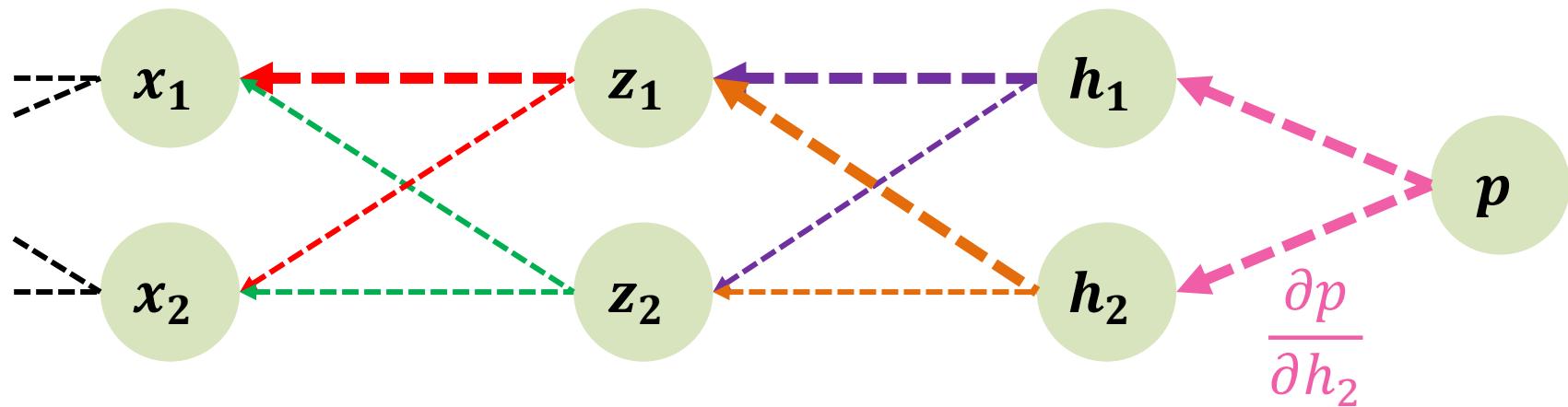
We will need these for GD

$$2: \frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1}$$

$$\frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \boxed{\frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_1}} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$



We can reuse previous computations

$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

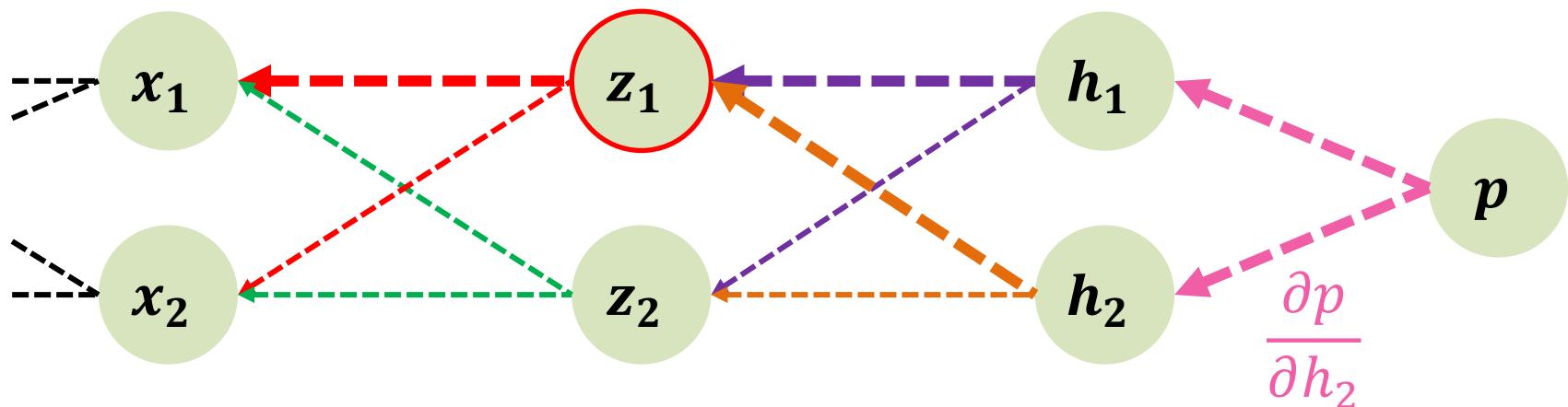
We will need these for GD

$$2: \frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1}$$

$$\frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \left(\frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \right) \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$



We can reuse previous computations

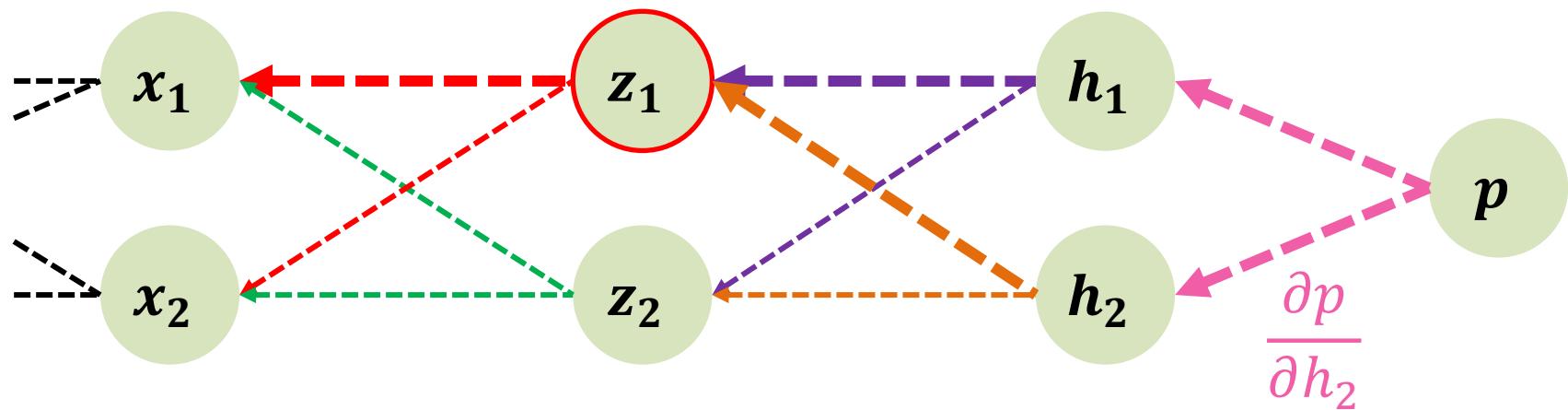
$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

We will need these for GD

$$2: \frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \quad \frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \left(\frac{\partial p}{\partial z_1} \right) \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$\frac{\partial p}{\partial x_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$



We can reuse previous computations

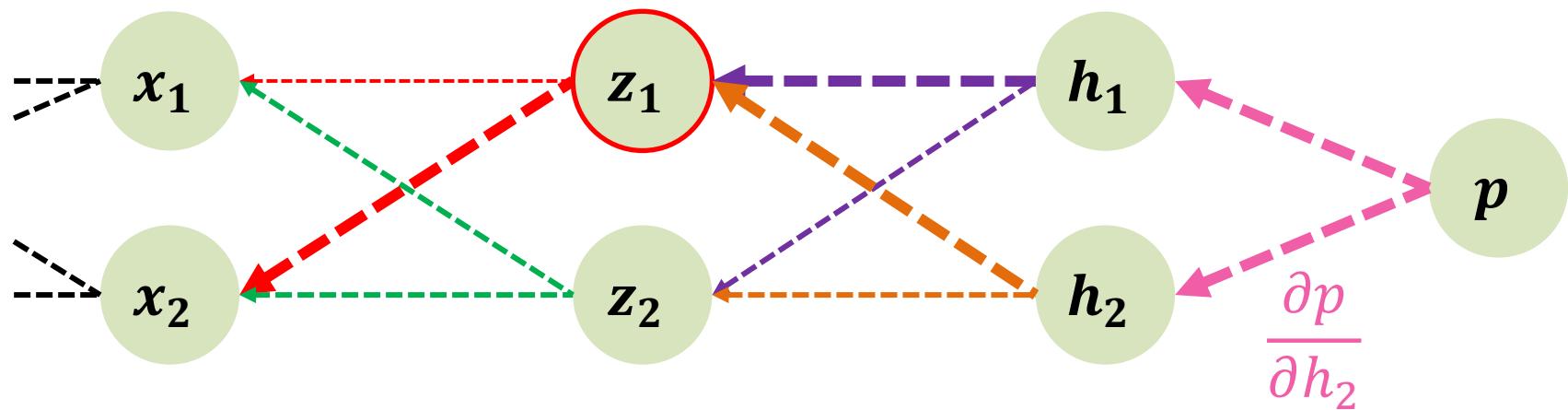
$$3: \frac{\partial p}{\partial h_1} \quad \frac{\partial p}{\partial h_2}$$

We will need these for GD

$$2: \frac{\partial p}{\partial z_1} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_1} \quad \frac{\partial p}{\partial z_2} = \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2}$$

$$1: \frac{\partial p}{\partial x_1} = \left(\frac{\partial p}{\partial z_1} \right) \frac{\partial z_1}{\partial x_1} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_1} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_1}$$

$$1: \frac{\partial p}{\partial x_2} = \left(\frac{\partial p}{\partial z_1} \right) \frac{\partial z_1}{\partial x_2} + \frac{\partial p}{\partial h_1} \frac{\partial h_1}{\partial z_2} \frac{\partial z_2}{\partial x_2} + \frac{\partial p}{\partial h_2} \frac{\partial h_2}{\partial z_2} \frac{\partial z_2}{\partial x_2}$$

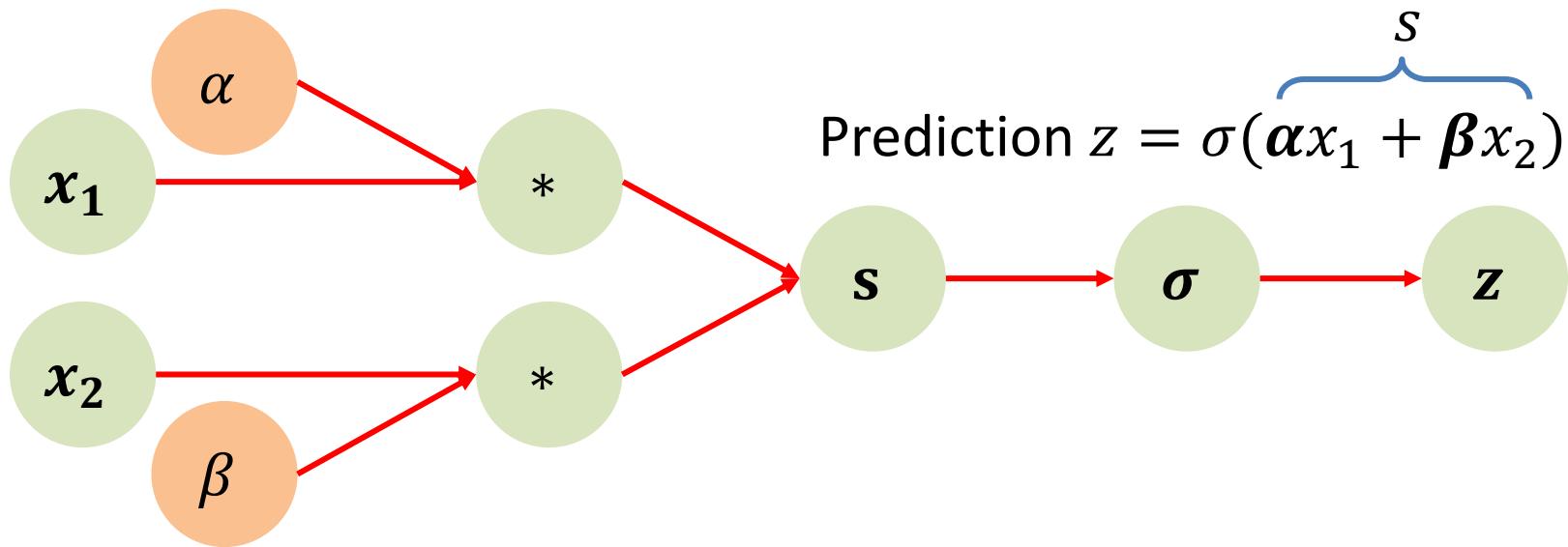


This is called reverse-mode differentiation

- In application to neural networks it has one more name: **back-propagation**.
- It works **fast**, because we reuse computations from previous steps.
- In fact, for each edge we compute its value only once. And multiply by its value exactly once.

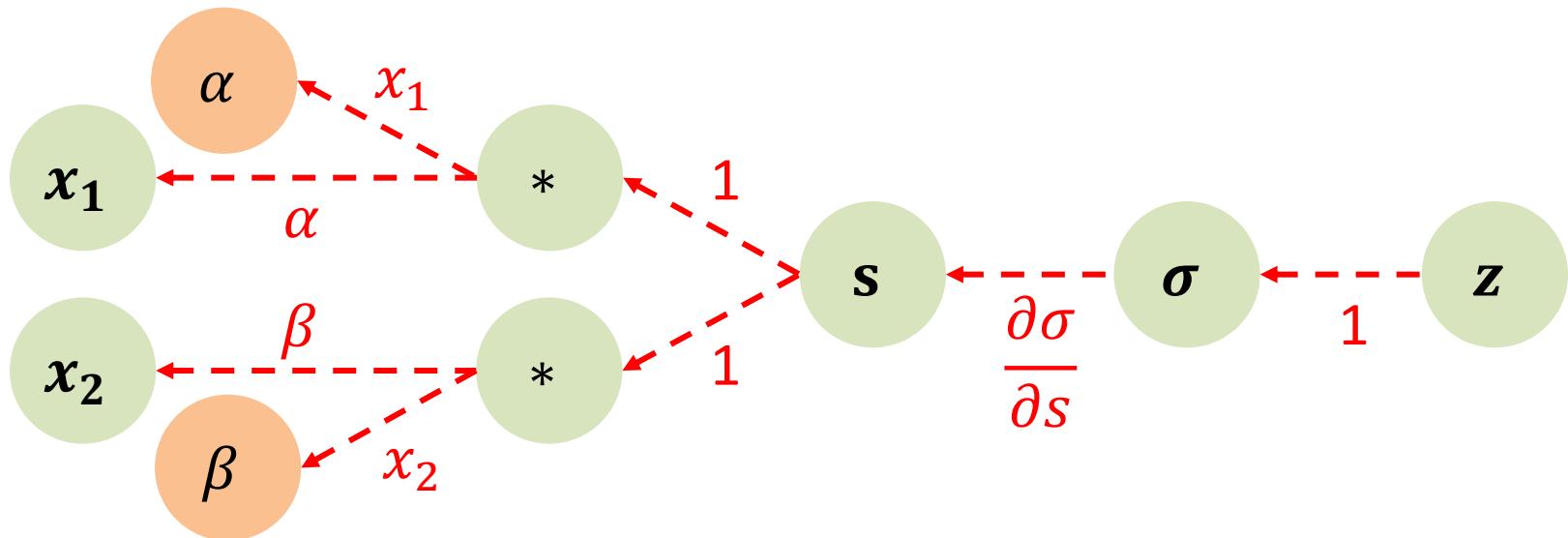
Back-propagation (Back-prop)

Forward pass



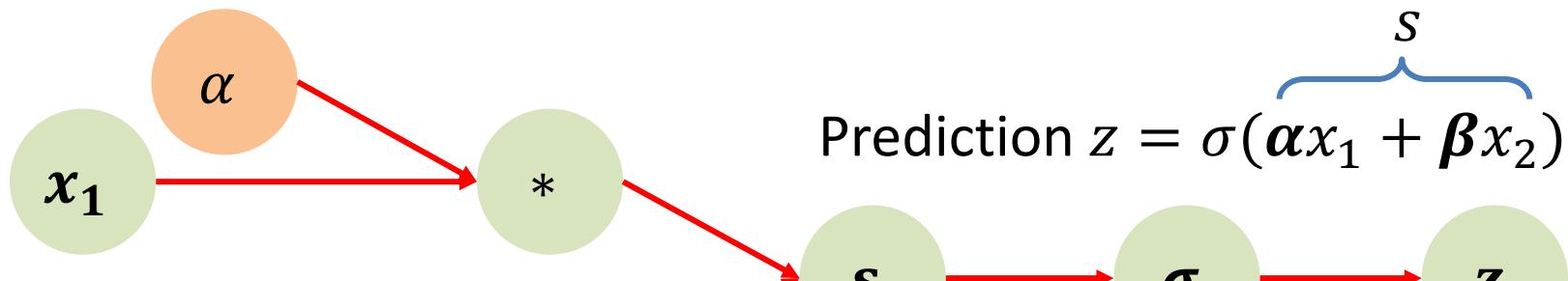
$$\text{Prediction } z = \sigma(\alpha x_1 + \beta x_2)$$

Backward pass



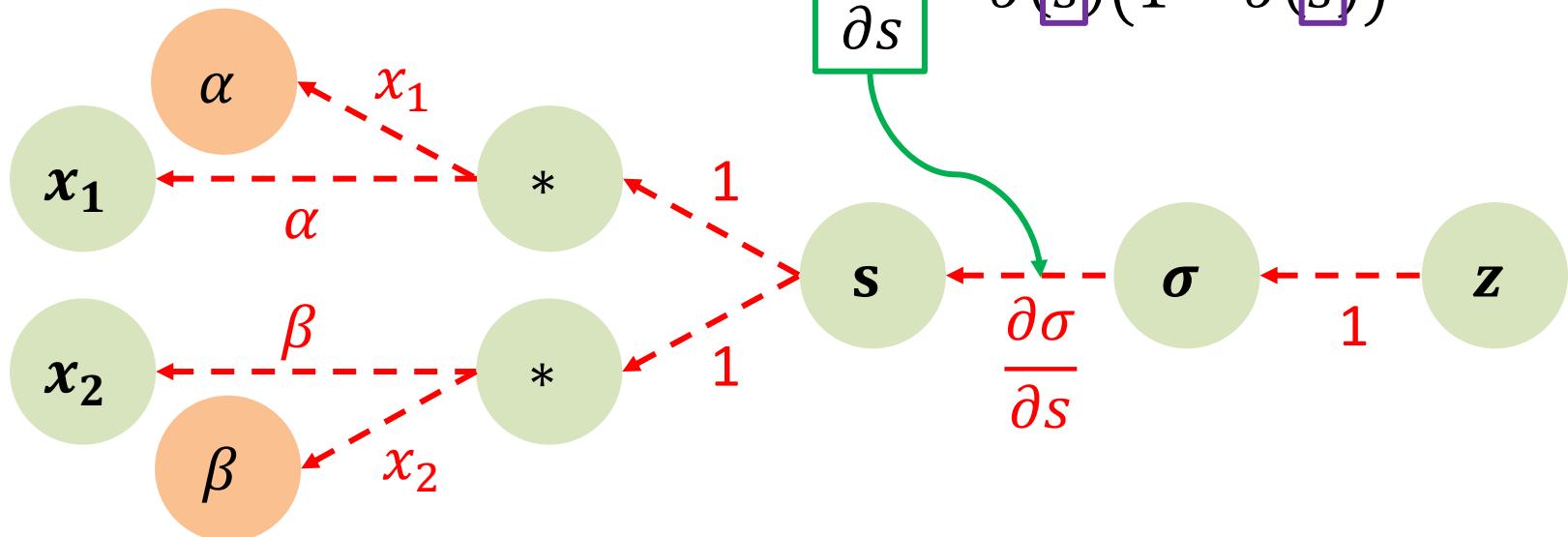
Back-propagation (Back-prop)

Forward pass



$$\text{Prediction } z = \sigma(\alpha x_1 + \beta x_2)$$

Backward pass

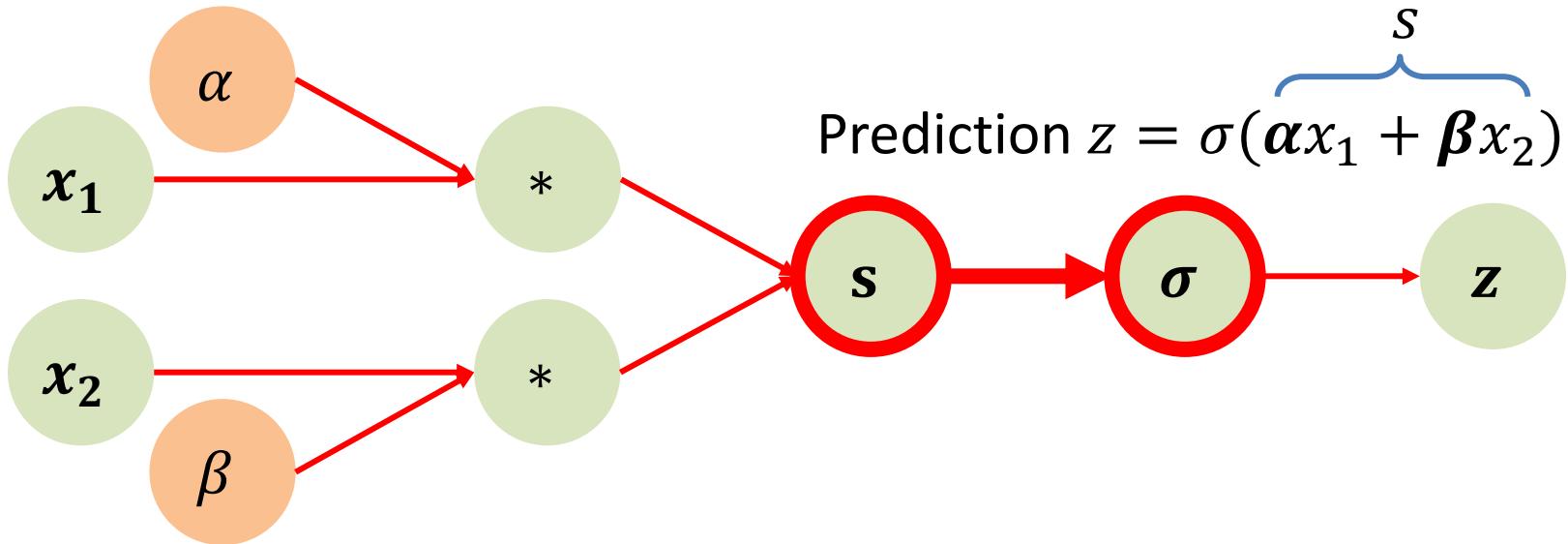


$$\frac{\partial \sigma}{\partial s} = \sigma(s)(1 - \sigma(s))$$

Forward pass interface

Let's implement a sigmoid activation node!

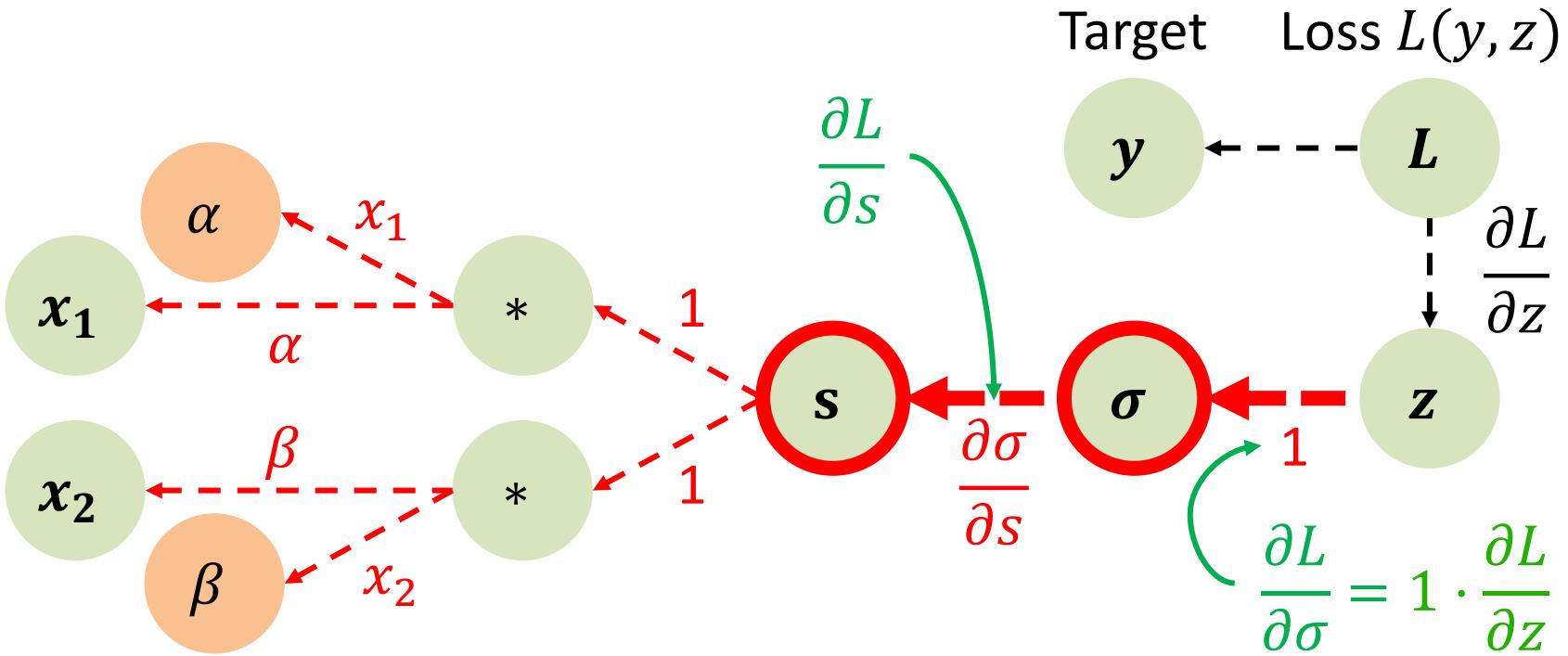
Forward pass



```
def forward_pass(inputs):
    return 1. / (1 + np.exp(-inputs))
```

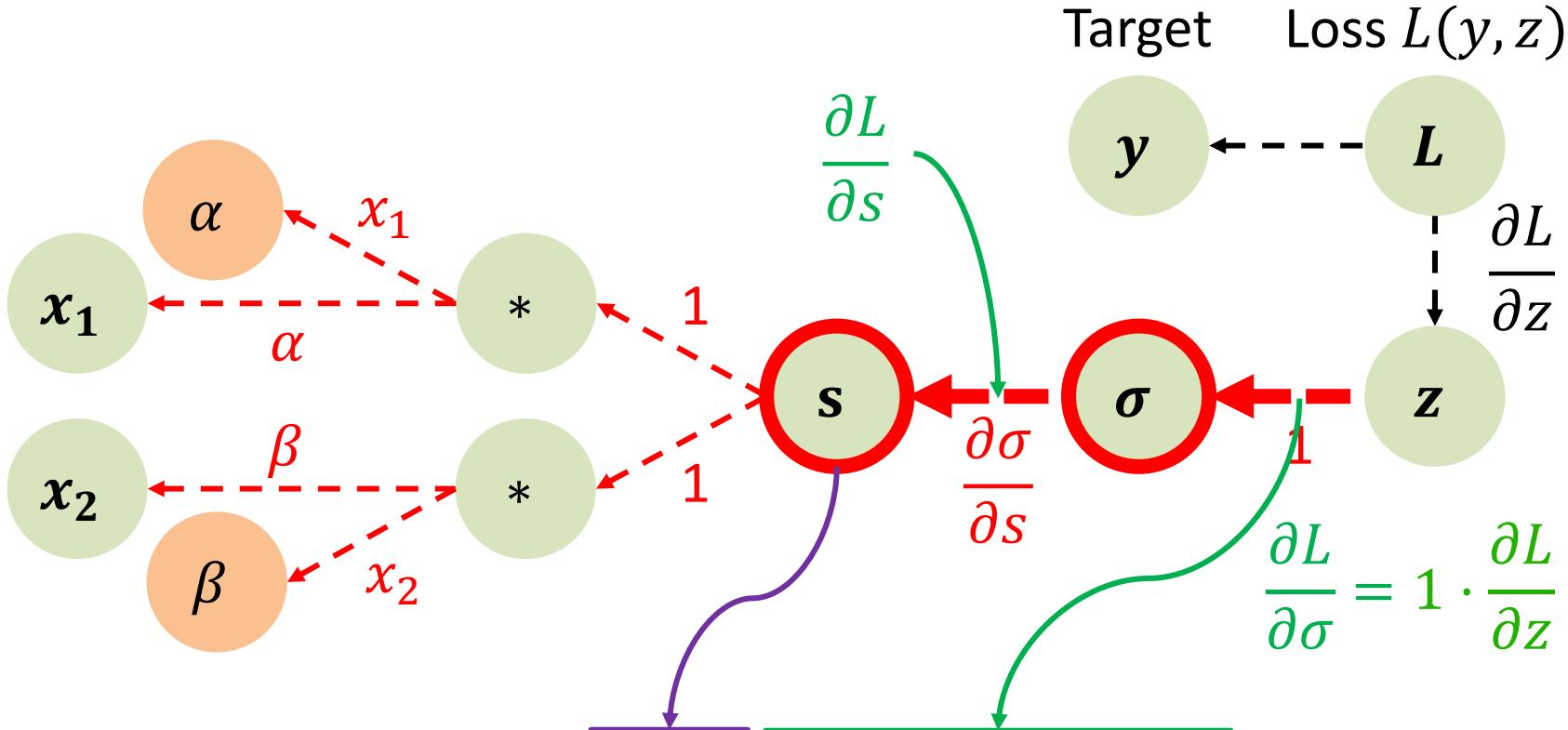
Backward pass interface

Backward pass



Backward pass interface

Backward pass



```
def backward_pass(inputs, incoming_gradient):
    sigmoid = 1. / (1 + np.exp(-inputs))
    return sigmoid * (1 - sigmoid) * incoming_gradient
```

$$\frac{\partial L}{\partial s} =$$

$$\frac{\partial \sigma}{\partial s} \cdot$$

$$\frac{\partial L}{\partial \sigma}$$

Summary

- Back-propagation is a name for an **automatic** reverse-mode differentiation
- Back-propagation is **fast**
- And this is how neural nets are trained today
- In the next video we will look at MLP implementation details

Intro

- In this video we will talk about MLP implementation details

Dense layer as a matrix multiplication

- We can compute 2 neurons with linear activation, 3 inputs and no bias term as a matrix multiplication:

$$(x_1 \quad x_2 \quad x_3) \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = (z_1 \quad z_2)$$

which is equivalent to:

$$\begin{aligned} z_1 &= x_1 w_{1,1} + x_2 w_{2,1} + x_3 w_{3,1} \\ z_2 &= x_1 w_{1,2} + x_2 w_{2,2} + x_3 w_{3,2} \end{aligned}$$

and is written as: $xW = z$

- Matrix multiplication can be done faster on both CPU (e.g. **BLAS**) and GPU (e.g. **cuBLAS**).
- Matrix multiplication with **numpy** is much faster than **Python** loops.

Backward pass for a dense layer

- Forward pass:

$$xW = z \quad (x_1 \quad x_2 \quad x_3) \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = (z_1 \quad z_2)$$

- For backward pass we need $\frac{\partial L}{\partial W}$, where $L(z_1, z_2)$ is our scalar loss.

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} \\ \frac{\partial L}{\partial w_{3,1}} & \frac{\partial L}{\partial w_{3,2}} \end{bmatrix}$$

which is convenient for SGD:

$$W_{new} = W - \gamma \frac{\partial L}{\partial W}$$

Backward pass for a dense layer

- Forward pass:

$$xW = z \quad (x_1 \quad x_2 \quad x_3) \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = (z_1 \quad z_2)$$

- For backward pass we need $\frac{\partial L}{\partial W}$, where $L(z_1, z_2)$ is our scalar loss.

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} \\ \frac{\partial L}{\partial w_{3,1}} & \frac{\partial L}{\partial w_{3,2}} \end{bmatrix} \quad \text{let's apply a chain rule}$$
$$\frac{\partial L}{\partial w_{i,j}} = \sum_k \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} x_i$$
$$z_j = x_1 w_{1,j} + x_2 w_{2,j} + x_3 w_{3,j}$$

Backward pass for a dense layer

- Forward pass:

$$xW = z \quad (x_1 \quad x_2 \quad x_3) \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = (z_1 \quad z_2)$$

- For backward pass we need $\frac{\partial L}{\partial W}$, where $L(z_1, z_2)$ is our scalar loss.

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial w_{1,1}} & \frac{\partial L}{\partial w_{1,2}} \\ \frac{\partial L}{\partial w_{2,1}} & \frac{\partial L}{\partial w_{2,2}} \\ \frac{\partial L}{\partial w_{3,1}} & \frac{\partial L}{\partial w_{3,2}} \end{bmatrix} \quad \frac{\partial L}{\partial w_{i,j}} = \sum_k \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} x_i$$
$$\frac{\partial L}{\partial z} = \left(\frac{\partial L}{\partial z_1} \quad \frac{\partial L}{\partial z_2} \right) \quad \text{gradient vector}$$
$$\frac{\partial L}{\partial W} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \cdot \left(\frac{\partial L}{\partial z_1} \quad \frac{\partial L}{\partial z_2} \right) = x^T \frac{\partial L}{\partial z}$$

Forward pass for mini-batches

- Usually we do forward pass in mini-batches.

Batch of 2:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$$

Matrix notation: $XW = Z$

1st neuron for 2nd sample: $z_{2,1} = x_{2,1}w_{1,1} + x_{2,2}w_{2,1} + x_{2,3}w_{3,1}$

Backward pass for mini-batches

- SGD with mini-batches takes a sum of losses for each sample.

Batch of 2:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$$

SGD step:

$$\frac{\partial L_b}{\partial W} = \frac{\partial L(z_{1,1}, z_{1,2})}{\partial W} + \frac{\partial L(z_{2,1}, z_{2,2})}{\partial W}$$

For one sample:

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{i,j}} = \frac{\partial L}{\partial z_j} x_i \quad \text{we know this}$$

For 2 samples:

$$\frac{\partial L_b}{\partial w_{i,j}} = \frac{\partial L}{\partial z_{1,j}} x_{1,i} + \frac{\partial L}{\partial z_{2,j}} x_{2,i}$$

Backward pass for mini-batches

- SGD with mini-batches takes a sum of losses for each sample.

Batch of 2:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$$

SGD step:

$$\frac{\partial L_b}{\partial W} = \frac{\partial L(z_{1,1}, z_{1,2})}{\partial W} + \frac{\partial L(z_{2,1}, z_{2,2})}{\partial W}$$

For 2 samples:

$$\frac{\partial L_b}{\partial w_{i,j}} = \frac{\partial L}{\partial z_{1,j}} x_{1,i} + \frac{\partial L}{\partial z_{2,j}} x_{2,i}$$

$$\frac{\partial L_b}{\partial W} = X^T \frac{\partial L}{\partial Z} \quad \left| \quad X^T = \begin{pmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \\ x_{1,3} & x_{2,3} \end{pmatrix} \quad \right| \quad \frac{\partial L}{\partial Z} = \begin{pmatrix} \frac{\partial L}{\partial z_{1,1}} & \frac{\partial L}{\partial z_{1,2}} \\ \frac{\partial L}{\partial z_{2,1}} & \frac{\partial L}{\partial z_{2,2}} \end{pmatrix}$$

Backward pass for mini-batches

- SGD with mini-batches takes a sum of losses for each sample.

Batch of 2:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$$

SGD step:

$$\frac{\partial L_b}{\partial W} = \frac{\partial L(z_{1,1}, z_{1,2})}{\partial W} + \frac{\partial L(z_{2,1}, z_{2,2})}{\partial W}$$

For 2 samples:

$$\frac{\partial L_b}{\partial w_{3,2}} = \frac{\partial L}{\partial z_{1,2}} x_{1,3} + \frac{\partial L}{\partial z_{2,2}} x_{2,3} \quad \text{let's check}$$

$$\frac{\partial L_b}{\partial W} = X^T \frac{\partial L}{\partial Z} \quad \left| \begin{array}{l} X^T = \begin{pmatrix} x_{1,1} & x_{2,1} \\ x_{1,2} & x_{2,2} \\ x_{1,3} & x_{2,3} \end{pmatrix} \\ \frac{\partial L}{\partial Z} = \begin{pmatrix} \frac{\partial L}{\partial z_{1,1}} & \frac{\partial L}{\partial z_{1,2}} \\ \frac{\partial L}{\partial z_{2,1}} & \frac{\partial L}{\partial z_{2,2}} \end{pmatrix} \end{array} \right.$$

Backward pass for X (used in MLP)

- X could contain outputs of a previous hidden layer:

Batch of 2: $\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \end{pmatrix} \cdot \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ w_{3,1} & w_{3,2} \end{bmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$

SGD step: $\frac{\partial L_b}{\partial X} = \frac{\partial L(z_{1,1}, z_{1,2})}{\partial X} + \frac{\partial L(z_{2,1}, z_{2,2})}{\partial X}$

For 1 sample: *let's apply a chain rule*

$$\frac{\partial L(z_{i,1}, z_{i,2})}{\partial x_{i,j}} = \sum_k \frac{\partial L}{\partial z_{i,k}} \frac{\partial z_{i,k}}{\partial x_{i,j}} = \sum_k \frac{\partial L}{\partial z_{i,k}} w_{j,k} = \sum_k \frac{\partial L}{\partial z_{i,k}} w_{k,j}^T$$

For 2 samples: $\frac{\partial L_b}{\partial X} = \frac{\partial L}{\partial Z} W^T$

← contributes 1 non-zero row

Fast Python implementation

- Forward pass for a **dense layer** with **numpy**:

```
def forward_pass(X, W):  
    return X.dot(W) 
$$XW = Z$$

```

- Backward pass with **numpy**:

```
def backward_pass(X, W, dZ):  
    dX = dZ.dot(W.T)  
    dW = X.T.dot(dZ)  
    return dX, dW 
$$\frac{\partial L_b}{\partial X} = \frac{\partial L}{\partial Z} W^T$$
 
$$\frac{\partial L_b}{\partial W} = X^T \frac{\partial L}{\partial Z}$$

```

- One more reason to have $\frac{\partial L}{\partial Z}$ in backward pass **interface**:
 - Otherwise, we would need $\frac{\partial Z}{\partial X}$ and $\frac{\partial Z}{\partial W}$, which is scary!

Summary

- Forward pass for a **dense layer** is a matrix multiplication
- Backward pass is a matrix multiplication as well
- Efficient for mini-batches on both CPU and GPU
- Easy to code it with **numpy**
- In the next video we will take a quick look at other matrix derivatives

Intro

- In this video we will take a look at other cases of matrix derivatives

Jacobian

- Let's take a composition of two vector functions:

$$g(x): (x_1, \dots, x_n) \rightarrow (g_1, \dots, g_m)$$

$$f(g): (g_1, \dots, g_m) \rightarrow (f_1, \dots, f_k) \quad \text{this will be useful for RNN}$$

$$h(x) = f(g(x)): (x_1, \dots, x_n) \rightarrow (h_1, \dots, h_k) = (f_1, \dots, f_k)$$

- The matrix of partial derivatives $\frac{\partial h_i}{\partial x_j}$ is called the Jacobian:

$$J^h = \begin{pmatrix} \frac{\partial h_1}{\partial x_1} & \dots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_1} & \dots & \frac{\partial h_k}{\partial x_n} \end{pmatrix}$$

Jacobian

- Let's take a composition of two vector functions:

$$g(x): (x_1, \dots, x_n) \rightarrow (g_1, \dots, g_m)$$

$$f(g): (g_1, \dots, g_m) \rightarrow (f_1, \dots, f_k) \quad \text{this will be useful for RNN}$$

$$h(x) = f(g(x)): (x_1, \dots, x_n) \rightarrow (h_1, \dots, h_k) = (f_1, \dots, f_k)$$

- The matrix of partial derivatives $\frac{\partial h_i}{\partial x_j}$ is called the Jacobian:

$$J^h = \begin{pmatrix} \frac{\partial h_1}{\partial x_1} & \dots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_k}{\partial x_1} & \dots & \frac{\partial h_k}{\partial x_n} \end{pmatrix}$$

$$J_{i,j}^h = \frac{\partial h_i}{\partial x_j} =$$

$$\sum_l \frac{\partial f_i}{\partial g_l} \frac{\partial g_l}{\partial x_j} = \sum_l J_{i,l}^f \cdot J_{l,j}^g$$

Chain rule: $J^h = J^f \cdot J^g$

Matrix by matrix derivative

- Matrix product:

$$C = AB = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} =$$
$$\begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}$$

- How to calculate the derivative $\frac{\partial C}{\partial A}$?

$$\frac{\partial C}{\partial A} = \left\{ \frac{\partial c_{i,j}}{\partial a_{k,l}} \right\}_{i,j,k,l}$$

This is a **tensor** (high-dimensional array)!

Chain rule

$$C = AB = \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}$$

$$\frac{\partial C}{\partial A} = \begin{pmatrix} \frac{\partial c_{1,1}}{\partial A} & \frac{\partial c_{1,2}}{\partial A} \\ \frac{\partial c_{2,1}}{\partial A} & \frac{\partial c_{2,2}}{\partial A} \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} b_{1,1} & b_{2,1} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} b_{1,2} & b_{2,2} \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} b_{1,1} & b_{2,1} \end{bmatrix} & \begin{bmatrix} b_{1,2} & b_{2,2} \end{bmatrix} \end{pmatrix}$$

Possible notation

- Let's say we have $\frac{\partial L}{\partial C} = \begin{bmatrix} \alpha & \beta \\ \gamma & \eta \end{bmatrix}$

$$\frac{\partial L}{\partial A} = \sum_{i,j} \frac{\partial L}{\partial c_{i,j}} \frac{\partial c_{i,j}}{\partial A} = \alpha \frac{\partial c_{1,1}}{\partial A} + \beta \frac{\partial c_{1,2}}{\partial A} + \gamma \frac{\partial c_{2,1}}{\partial A} + \eta \frac{\partial c_{2,2}}{\partial A} =$$
$$\begin{bmatrix} \alpha b_{1,1} + \beta b_{1,2} & \alpha b_{2,1} + \beta b_{2,2} \\ \gamma b_{1,1} + \eta b_{1,2} & \gamma b_{2,1} + \eta b_{2,2} \end{bmatrix}$$

What's bad about it

- Our chain rule is a **linear combination** of matrices:

$$\frac{\partial L}{\partial A} = \sum_{i,j} \frac{\partial L}{\partial c_{i,j}} \frac{\partial c_{i,j}}{\partial A} = \alpha \frac{\partial c_{1,1}}{\partial A} + \beta \frac{\partial c_{1,2}}{\partial A} + \gamma \frac{\partial c_{2,1}}{\partial A} + \eta \frac{\partial c_{2,2}}{\partial A} =$$
$$\begin{bmatrix} \alpha b_{1,1} + \beta b_{1,2} & \alpha b_{2,1} + \beta b_{2,2} \\ \gamma b_{1,1} + \eta b_{1,2} & \gamma b_{2,1} + \eta b_{2,2} \end{bmatrix}$$

- **Crunching a lot of zeros:**

$$\frac{\partial C}{\partial A} = \begin{pmatrix} \frac{\partial c_{1,1}}{\partial A} & \frac{\partial c_{1,2}}{\partial A} \\ \frac{\partial c_{2,1}}{\partial A} & \frac{\partial c_{2,2}}{\partial A} \end{pmatrix} = \begin{pmatrix} \begin{bmatrix} b_{1,1} & b_{2,1} \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} b_{1,2} & b_{2,2} \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ b_{1,1} & b_{2,1} \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ b_{1,2} & b_{2,2} \end{bmatrix} \end{pmatrix}$$

What's bad about it

- Our chain rule is a **linear combination** of matrices:

$$\frac{\partial L}{\partial A} = \sum_{i,j} \frac{\partial L}{\partial c_{i,j}} \frac{\partial c_{i,j}}{\partial A} = \alpha \frac{\partial c_{1,1}}{\partial A} + \beta \frac{\partial c_{1,2}}{\partial A} + \gamma \frac{\partial c_{2,1}}{\partial A} + \eta \frac{\partial c_{2,2}}{\partial A} =$$
$$\begin{bmatrix} \alpha b_{1,1} + \beta b_{1,2} & \alpha b_{2,1} + \beta b_{2,2} \\ \gamma b_{1,1} + \eta b_{1,2} & \gamma b_{2,1} + \eta b_{2,2} \end{bmatrix}$$

- **Performance issues**
 - e.g. no such operation in BLAS

What's bad about it

- Our chain rule is a **linear combination** of matrices:

$$\frac{\partial L}{\partial A} = \sum_{i,j} \frac{\partial L}{\partial c_{i,j}} \frac{\partial c_{i,j}}{\partial A} = \alpha \frac{\partial c_{1,1}}{\partial A} + \beta \frac{\partial c_{1,2}}{\partial A} + \gamma \frac{\partial c_{2,1}}{\partial A} + \eta \frac{\partial c_{2,2}}{\partial A} =$$

$$\begin{bmatrix} \alpha b_{1,1} + \beta b_{1,2} & \alpha b_{2,1} + \beta b_{2,2} \\ \gamma b_{1,1} + \eta b_{1,2} & \gamma b_{2,1} + \eta b_{2,2} \end{bmatrix}$$



- We know how to compute $\frac{\partial L}{\partial A}$ skipping $\frac{\partial C}{\partial A}$ tensor computation:

$$\frac{\partial L}{\partial C} = \begin{bmatrix} \alpha & \beta \\ \gamma & \eta \end{bmatrix}$$

$$B^T = \begin{pmatrix} b_{1,1} & b_{2,1} \\ b_{1,2} & b_{2,2} \end{pmatrix}$$

From MLP lecture: $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial C} B^T = \begin{bmatrix} \alpha & \beta \\ \gamma & \eta \end{bmatrix} \begin{pmatrix} b_{1,1} & b_{2,1} \\ b_{1,2} & b_{2,2} \end{pmatrix}$

We're still fine

- Thankfully, in deep learning frameworks we need to calculate gradients of a **scalar loss** with respect to another **scalar, vector, matrix or tensor**.
- This's how `tf.gradients()` in TensorFlow works.
- Deep learning frameworks have **optimized** versions of backward pass for standard layers.

Summary

- For vector functions chain rule says that you need to multiply respective **Jacobians**.
- Matrix by matrix derivative is a **tensor**
- Chain rule for such tensors is **not very useful** in practice
- Thankfully, in deep learning frameworks we usually need to track gradients of a **scalar loss** with respect to all other parameters.

Intro

- In this video we will overview basic principles of TensorFlow

TensorFlow DL framework

- We will use it in Jupyter Notebook with Python 3 kernel

```
import numpy as np
import tensorflow as tf
```
- We will overview Python API for TensorFlow 1.2+
- APIs in other languages exist: Java, C++, Go
 - Python API is at present the most complete and the easiest to use
 - https://www.tensorflow.org/api_docs/



What is TensorFlow?

1. A tool to describe computational graphs
 - The foundation of computation in TensorFlow is the **Graph** object. This holds a network of nodes, each representing one **operation**, connected to each other as inputs and outputs.
2. A runtime for execution of these graphs
 - On CPU, GPU, TPU, ...
 - On one node or in distributed mode

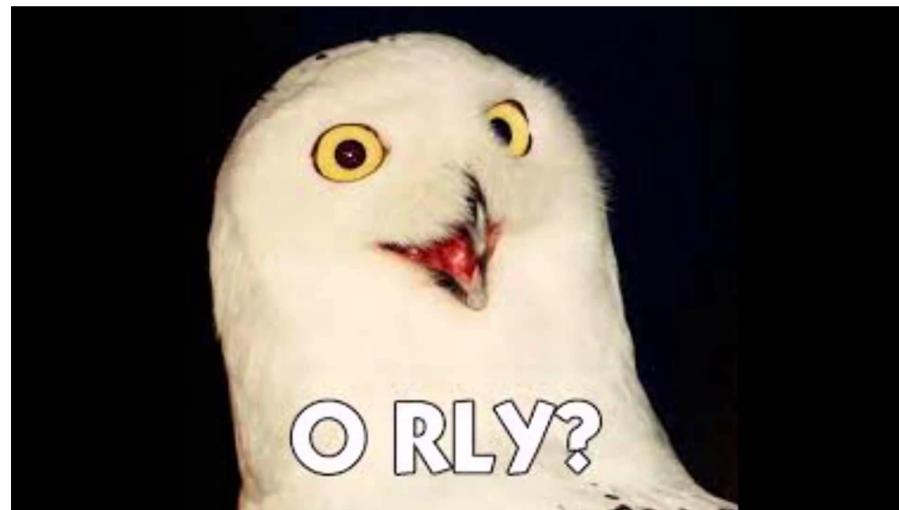


Why this name

- **Input** to any operation will be a collection of **tensors** (multi-dimensional arrays)
- **Output** will be a collection of tensors as well.
- We will have a graph of operations, each of which transforms tensors into another tensors, so it's a kind of a flow of tensors

Why this name

- **Input** to any operation will be a collection of **tensors** (multi-dimensional arrays)
- **Output** will be a collection of tensors as well.
- We will have a graph of operations, each of which transforms tensors into another tensors, so it's a kind of a flow of tensors



How the input looks like

- **Placeholder**
 - This is placeholder for a tensor, which will be fed during graph execution (e.g. input features)
 - `x = tf.placeholder(tf.float32, (None, 10))`
- **Variable**
 - This is a tensor with some value that is updated during execution (e.g. weights matrix in MLP)
 - `w = tf.get_variable("w", shape=(10, 20), dtype=tf.float32)`
 - `w = tf.Variable(tf.random_uniform((10, 20)), name="w")`
- **Constant**
 - This is a tensor with constant value, that cannot be changed
 - `c = tf.constant(np.ones((4, 4)))`

Operation example

- Matrix product:

```
x = tf.placeholder(tf.float32, (None, 10))
w = tf.Variable(tf.random_uniform((10, 20)), name="w")
z = x @ w
# z = tf.matmul(x, w)
print(z)
```

- Output:

```
Tensor("matmul:0", shape=(?, 20), dtype=float32)
```

- We don't do any computations here, we just **define** the graph!

Computational graph

- TensorFlow creates a **default graph** after importing
 - All the operations will go there by default
 - You can get it with `tf.get_default_graph()`, which returns an instance of `tf.Graph`.
- You can **create your own** graph variable and define operations there:

```
g = tf.Graph()
with g.as_default():
    pass
```
- You can **clear** the default graph like this:

```
tf.reset_default_graph()
```

Jupyter Notebook cells

- If you run this cell **3 times**:

```
x = tf.placeholder(tf.float32, (None, 10))
```

- This is what you get in your **default graph**:

- Using `tf.get_default_graph().get_operations()`

```
[<tf.Operation 'Placeholder' type=Placeholder>,
 <tf.Operation 'Placeholder_1' type=Placeholder>,
 <tf.Operation 'Placeholder_2' type=Placeholder>]
```

- **Your graph is cluttered!**

- Clear your graph with `tf.reset_default_graph()`

Operations and tensors

- Every **node** in our graph is an **operation**:

```
x = tf.placeholder(tf.float32, (None, 10), name="x")
```

- Listing nodes with `tf.get_default_graph().get_operations()`:

```
[<tf.Operation 'x' type=Placeholder>]
```

- How to get **output tensors** of operation:

- `tf.get_default_graph().get_operations()[0].outputs`
- Output: `[<tf.Tensor 'x:0' shape=(?, 10) dtype=float32>]`

Running a graph

- A **tf.Session** object encapsulates the environment in which tf.Operation objects are executed, and tf.Tensor objects are evaluated.
- Create a session: `s = tf.InteractiveSession()`
- Defining a graph:
`a = tf.constant(5.0)`
`b = tf.constant(6.0)`
`c = a * b`
- Running a graph:
`print(c) # here just looking at the type`
`print(s.run(c)) # that's how you run the graph`
- Output:
`Tensor("mul:0", shape=(), dtype=float32)`
`30.0`

Running a graph

- Operations are written in C++ and executed on CPU or GPU.
- `tf.Session` owns necessary resources to execute your graph, such as `tf.Variable`, that occupy RAM.
- It is important to release these resources when they are no longer required with `tf.Session.close()`

Initialization of variables

- A variable has an initial value:
 - Tensor: `tf.Variable(tf.random_uniform((10, 20)), name="w")`
 - Initializer: `tf.get_variable("w", shape=(10, 20), dtype=tf.float32)`
- You need to run some code to **compute that initial value** in graph execution environment
- This is done with a call in your session s:

```
s.run(tf.global_variables_initializer())
```
- Without it you will get “Attempting to use uninitialized value” errors

Example

- Definition:

```
tf.reset_default_graph()  
a = tf.constant(np.ones((2, 2), dtype=np.float32))  
b = tf.Variable(tf.ones((2, 2)))  
c = a @ b
```

- Running attempt:

```
s = tf.InteractiveSession()  
s.run(c)
```

Output: “**Attempting to use uninitialized value**” error

- Running properly:

```
s.run(tf.global_variables_initializer())  
s.run(c)
```

Output: array([[2., 2.], [2., 2.]], dtype=float32)

Feeding placeholder values

- Definition:

```
tf.reset_default_graph()  
a = tf.placeholder(np.float32, (2, 2))  
b = tf.Variable(tf.ones((2, 2)))  
c = a @ b
```

- Running attempt:

```
s = tf.InteractiveSession()  
s.run(tf.global_variables_initializer())  
s.run(c)
```

Output: “**You must feed a value for placeholder tensor**” error

- Running properly:

```
s.run(tf.global_variables_initializer())  
s.run(c, feed_dict={a: np.ones((2, 2))})
```

Output: array([[2., 2.], [2., 2.]], dtype=float32)

Summary

- TensorFlow: defining and running computational graphs
- Nodes of a graph are operations, that convert a collection of tensors into another collection of tensors
- In Python API you **define** the graph, you **don't execute** it along the way
 - In 1.5+ the latter mode is supported: eager execution
- You create a **session** to execute your graph (fast C++ code on CPU or GPU)
- Session **owns** all the resources (tensors eat RAM)

Intro

- In this video we will train our first model in TensorFlow

Optimizers in TensorFlow

- Let's define **f** as a square of variable **x**:

```
import numpy as np
import tensorflow as tf

tf.reset_default_graph()
x = tf.get_variable("x", shape=(), dtype=tf.float32)
f = x ** 2
```

- Let's say we want to *minimize* the value of **f** w.r.t **x**:

```
optimizer = tf.train.GradientDescentOptimizer(0.1)
step = optimizer.minimize(f, var_list=[x])
```

Trainable variables

- You don't have to specify all the optimized variables:

```
step = optimizer.minimize(f, var_list=[x])  
step = optimizer.minimize(f)
```

- Because all variables are **trainable** by default:

```
x = tf.get_variable("x", shape=(), dtype=tf.float32)  
x = tf.get_variable("x", shape=(), dtype=tf.float32, trainable=True)
```

- You can get all of them:

```
tf.trainable_variables()
```

- Output:

```
[<tf.Variable 'x:0' shape=() dtype=float32_ref>]
```

Making gradient descent steps

- Now we need to create a **session** and **initialize** variables:

```
s = tf.InteractiveSession()  
s.run(tf.global_variables_initializer())
```

- We are ready to make 10 **gradient descent** steps:

```
for i in range(10):  
    _, curr_x, curr_f = s.run([step, x, f])  
    print(curr_x, curr_f)
```

- Output:

```
0.448929 0.314901  
0.359143 0.201537  
...  
0.0753177 0.00886368  
0.0602542 0.00567276
```

← *GD step is already applied to x*

Logging with tf.Print

- We can **evaluate** tensors and print them like this:

```
for i in range(10):
    _, curr_x, curr_f = s.run([step, x, f])
    print(curr_x, curr_f)
```

- Or we can pass our tensor of interest through **tf.Print**:

```
...
f = x ** 2
f = tf.Print(f, [x, f], "x, f:")
...
for i in range(10):
    s.run([step, f])
```

x, f:[1.5879565][2.521606]
x, f:[1.2703652][1.6138278]
...
x, f:[0.26641488][0.070976891]
x, f:[0.2131319][0.04542521]

*This is Jupyter Server stdout.
Not visible in the Notebook!*

Logging with TensorBoard

- We can add so-called **summaries**:

```
tf.summary.scalar('curr_x', x)
tf.summary.scalar('curr_f', f)
summaries = tf.summary.merge_all()
```

- This is how we log these summaries:

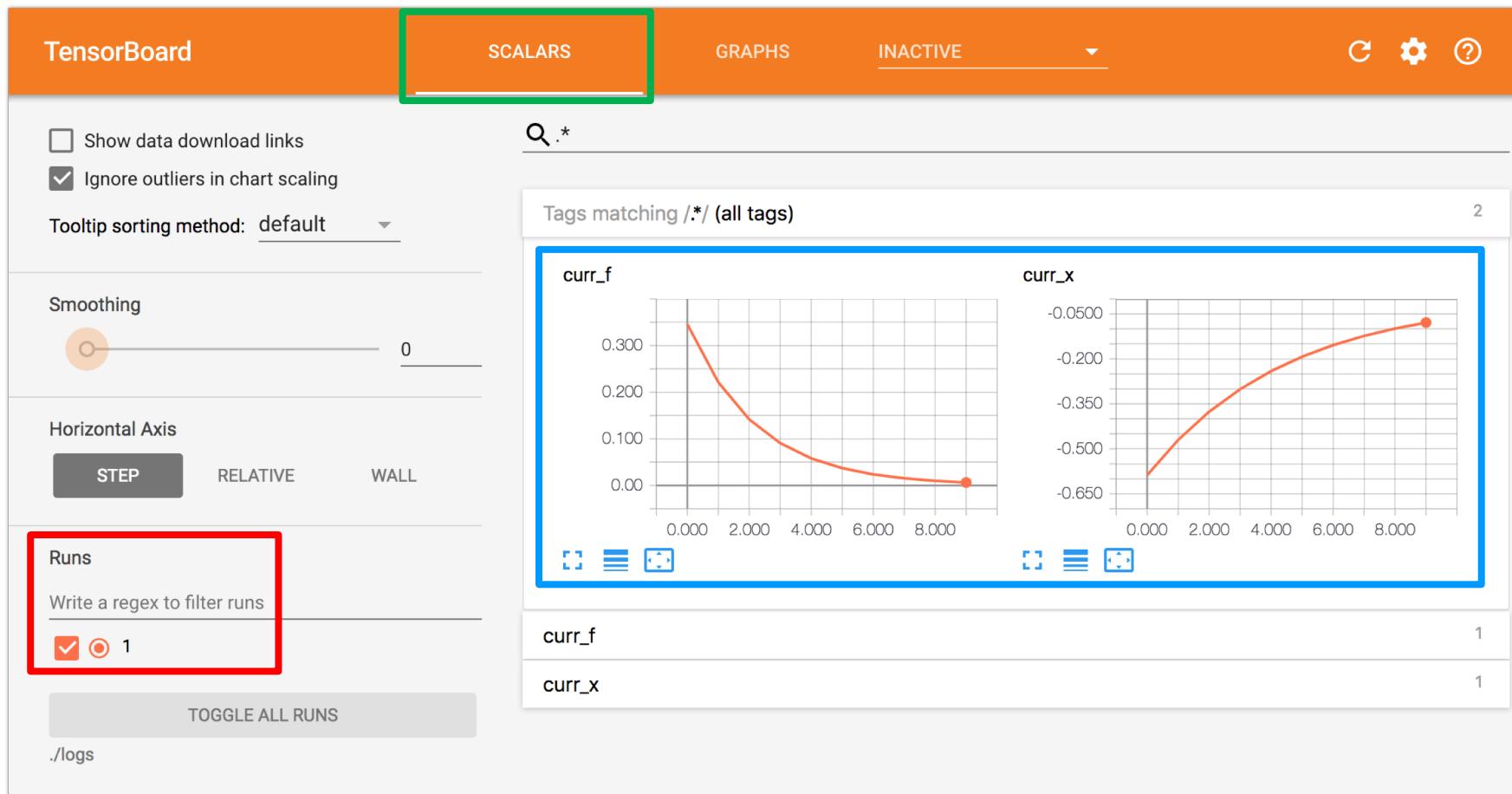
```
s = tf.InteractiveSession()
summary_writer = tf.summary.FileWriter("logs/1", s.graph)
s.run(tf.global_variables_initializer())
for i in range(10):
    _, curr_summaries = s.run([step, summaries])
    summary_writer.add_summary(curr_summaries, i)
    summary_writer.flush()
```

Run number



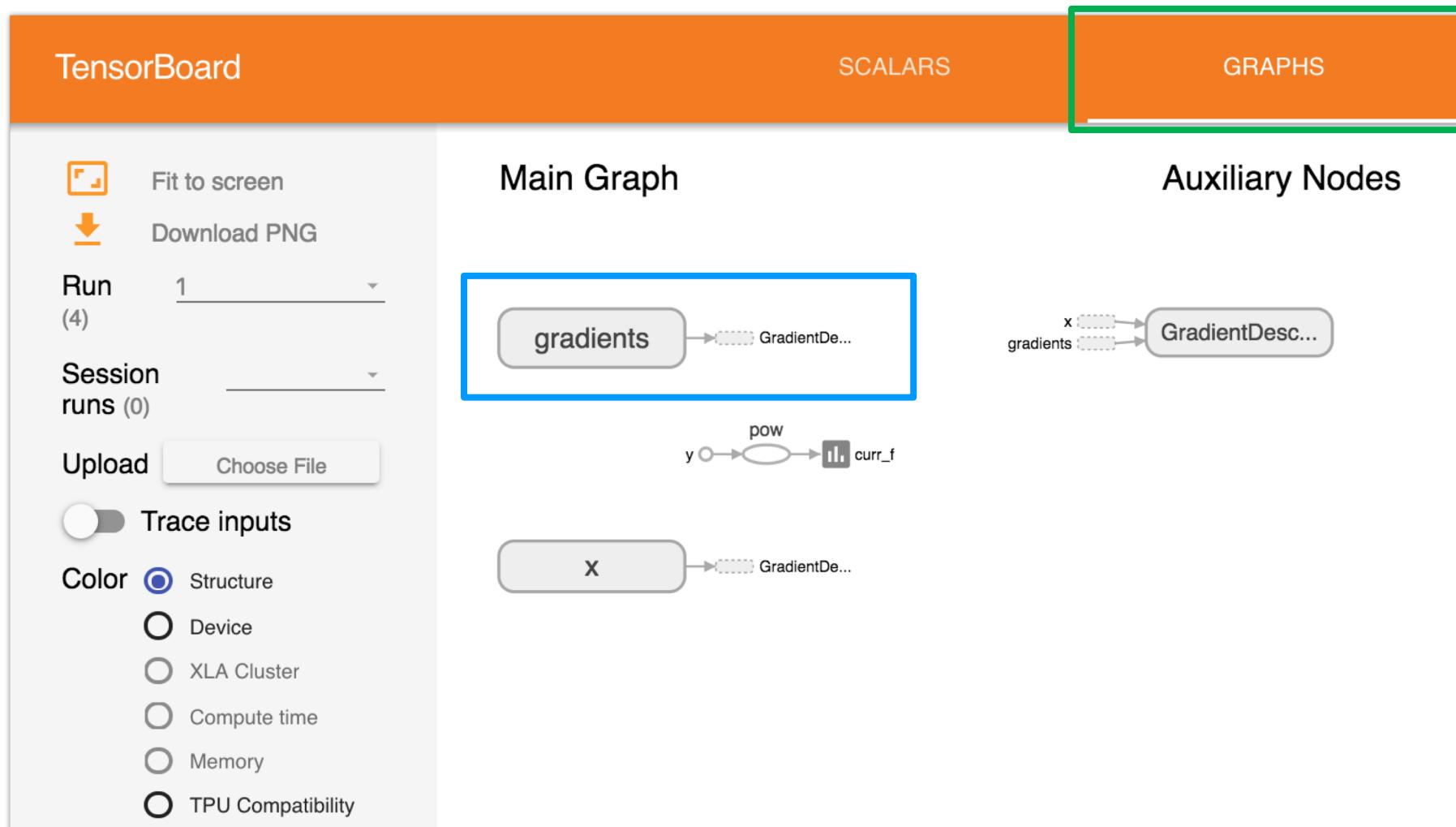
Launching TensorBoard

- Now you can launch TensorBoard via bash:
tensorboard --logdir=./logs
- And open <http://127.0.0.1:6006> in your browser.



Visualizing graph in TensorBoard

- You can see that **gradients computation** is a part of our graph:



Solving a linear regression

- Let's generate a model dataset:

$N = 1000$

$D = 3$

$x = np.random.random((N, D))$

$w = np.random.random((D, 1))$

$y = x @ w + np.random.randn(N, 1) * 0.20$

Solving a linear regression

- We will need **placeholders** for input data:

```
tf.reset_default_graph()  
features = tf.placeholder(tf.float32, shape=(None, D))  
target = tf.placeholder(tf.float32, shape=(None, 1))
```

- This is how we make **predictions**:

```
weights = tf.get_variable("w", shape=(D, 1), dtype=tf.float32)  
predictions = features @ weights
```

- And define our **loss**:

```
loss = tf.reduce_mean((target - predictions) ** 2)
```

- And **optimizer**:

```
optimizer = tf.train.GradientDescentOptimizer(0.1)  
step = optimizer.minimize(loss)
```

Solving a linear regression

- Gradient descent:

```
s = tf.InteractiveSession()  
s.run(tf.global_variables_initializer())  
for i in range(300):  
    _, curr_loss, curr_weights = s.run(  
        [step, loss, weights], feed_dict={features: x, target: y})  
if i % 50 == 0:  
    print(curr_loss)
```

Filling placeholders



- Ground truth weights:

[0.11649134, 0.82753164, 0.46924019]

- Found weights:

[0.13715988, 0.79555332, 0.47024861]

Model checkpoints

- We can save variables' state with `tf.train.Saver`:

```
s = tf.InteractiveSession()
saver = tf.train.Saver(tf.trainable_variables())
s.run(tf.global_variables_initializer())
for i in range(300):
    _, curr_loss, curr_weights = s.run(
        [step, loss, weights], feed_dict={features: x, target: y})
    if i % 50 == 0:
        saver.save(s, "logs/2/model.ckpt", global_step=i)
        print(curr_loss)
```

Model checkpoints

- We can **list** last checkpoints:

```
saver.last_checkpoints
```

```
['logs/2/model.ckpt-50', 'logs/2/model.ckpt-100',
 'logs/2/model.ckpt-150', 'logs/2/model.ckpt-200',
 'logs/2/model.ckpt-250']
```

- We can **restore** a previous checkpoint like this:

```
saver.restore(s, "logs/2/model.ckpt-50")
```

- Only variables' values are restored, which means that you need to define a graph in *the same way* **before restoring** a checkpoint.

Summary

- TensorFlow has **built-in optimizers** that do back-propagation automatically.
- TensorBoard provides **tools for visualizing** your training progress.
- TensorFlow allows you to **checkpoint** your graph to restore its state later (*you need to define it in exactly the same way though*)