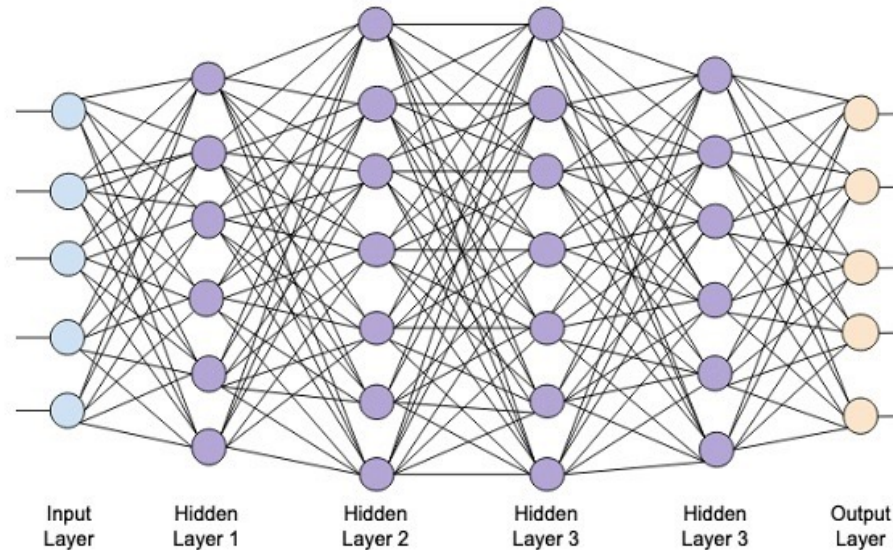


Artificial Neural Network - Algorithm

-Automatic differentiation/algorithmic differentiation

- Backpropagation

□ Neural Network Model:



Machine Learning Framework/Structure:

1. Data: $\mathcal{D} = (\vec{x}^{(i)}, y^{(i)})$ for $i = 1 \dots n$.
2. Model $h_{\Theta}(\vec{x})$
3. Cost Function $J(\Theta)$
4. Optimization $\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}} J(\Theta)$
5. Prediction $h_{\hat{\Theta}}(\vec{x})$

Neural Network Model:

$$h_{\Theta}(\vec{x}) := F^{[m]} \circ \Theta^{[m]} \circ \dots \circ F^{[2]} \circ \Theta^{[2]} \circ F^{[1]} \circ \Theta^{[1]}$$

Cost Functions :

$$J(\Theta) := L(h_{\Theta}(X), \vec{y}), \text{ where } L(-, -) \text{ is a } \mathbf{metric}.$$

For example:

1. Mean Square Error for regression

$$J(\Theta) = \frac{1}{n} \|h_{\Theta}(X) - \vec{y}\|^2 = \frac{1}{n} \sum_{i=1}^n (h_{\Theta}(\vec{x}^{(i)}) - y^{(i)})^2$$

2. Cross-Entropy cost for classification

$$J(\Theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K \mathbb{I}(y^{(i)} = k) \ln \left(h_{\Theta}(\vec{x}^{(i)})_k \right)$$

3. Hinge loss, 0–1 loss, ...

Cost Function Example:

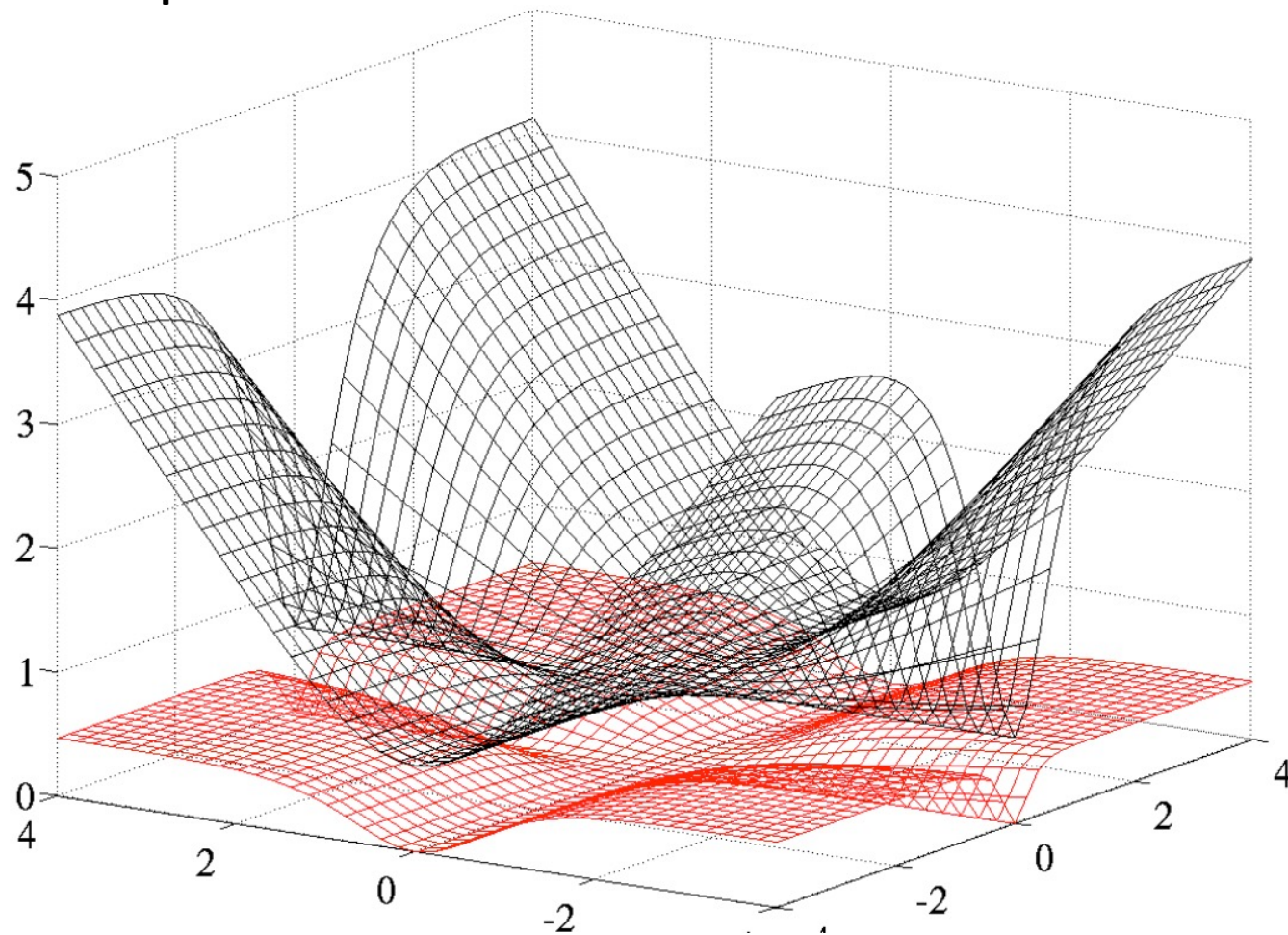


Figure: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, θ_1 respectively on the first layer and θ_2 on the second, output layer. (Glorot & Bentio (2010))

Optimization-Gradient Descent:

$$\Theta^{j+1} = \Theta^j - \alpha \nabla J(\Theta^j)$$

The key calculation is the **gradient** $\nabla J(\Theta)$

Derivative from Calculus:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Computation Methods:

- **Numerical** differentiation using finite difference approximations;
- Manually working out **analytical** derivatives and coding them directly;
- Derive analytic gradient, check your implementation with numerical gradient to avoid redundant computations; (called automatic differentiation, also called **algorithmic differentiation**), e.g., Back-propagation method.

➤ **Back-propagation** (Reverse auto-differentiation)

Goal: Minimize the loss function $J(\Theta)$

We need to calculate $\nabla J(\Theta^j)$, but not write the whole formula $\nabla J(\Theta)$.

Difficulty: Too much calculation/memory in formula $\nabla J(\Theta)$

Solution: Back-propagation

In 1986, (*Learning representations by back-propagating errors, Nature, 323(9): 533-536*) D. E. Rumelhart popularized the idea of **back propagation** to compute gradients. It is not a learning method, but a computational trick. It is actually a simple implementation of chain rule of derivatives.

BP algorithms as stochastic gradient descent algorithms (Robbins–Monro 1950; Kiefer- Wolfowitz 1951) with Chain rules of Gradient maps. Implemented to run on computers as early as 1970 by Seppo Linnainmaa.

➤ The Chain Rules:

- Univariate Chain Rule

$$\mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$$

$$\frac{df(x(t))}{dt} = \frac{df}{dx} \frac{dx}{dt}$$

- Multivariate Chain Rule

$$\mathbb{R} \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\frac{df(\vec{x}(t))}{dt} = \frac{\partial f}{\partial x_1} \frac{dx_1}{dt} + \dots + \frac{\partial f}{\partial x_n} \frac{dx_n}{dt}$$

$$\mathbb{R}^s \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\frac{\partial f(\vec{x}(\vec{t}))}{\partial t_i} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t_i} + \dots + \frac{\partial f}{\partial x_n} \frac{\partial x_n}{\partial t_i}$$

Chain Rule in Matrix Calculus Notation:

$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m$$

Use the **numerator layout** convention for the derivative of $F(\vec{x})$

$$\frac{\partial F}{\partial \vec{x}} := \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

called Jacobian matrix.

The Chain Rule for $F = F(\vec{x}(\vec{t}))$:

$$\mathbb{R}^s \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$\frac{\partial F}{\partial \vec{t}} = \frac{\partial F}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{t}}$$

More compositions $F = F(\vec{z}(\vec{x}(\vec{t})))$:

$$\mathbb{R}^a \rightarrow \mathbb{R}^s \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$\frac{\partial F}{\partial \vec{t}} = \frac{\partial F}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{t}}$$

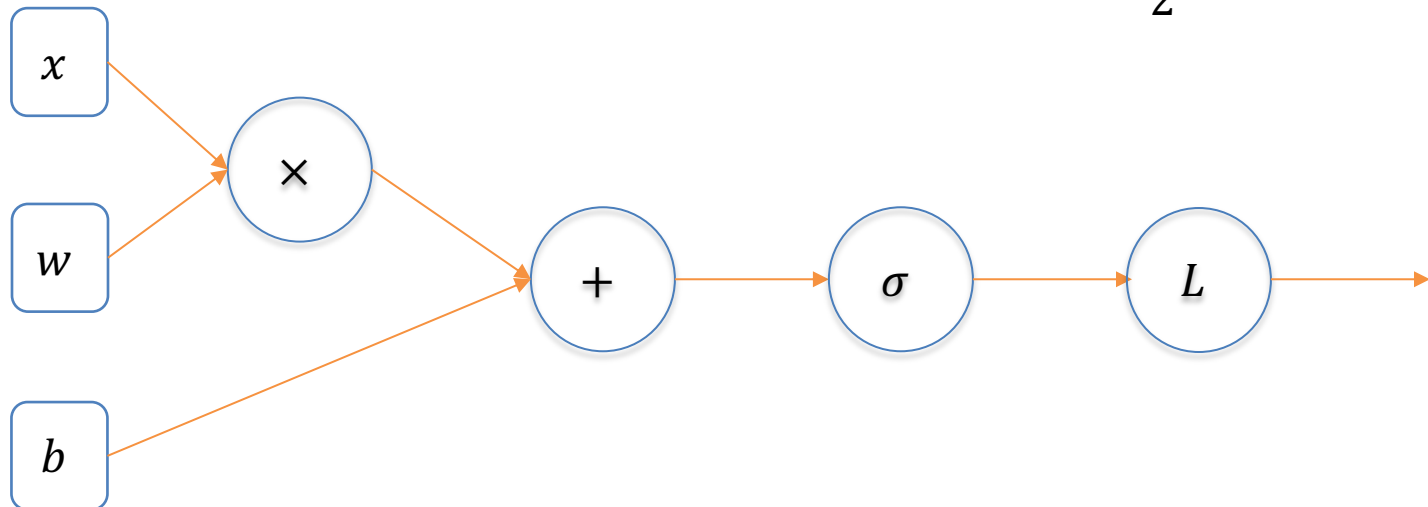
Example: One layer neural network:

$$z = wx + b$$

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

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

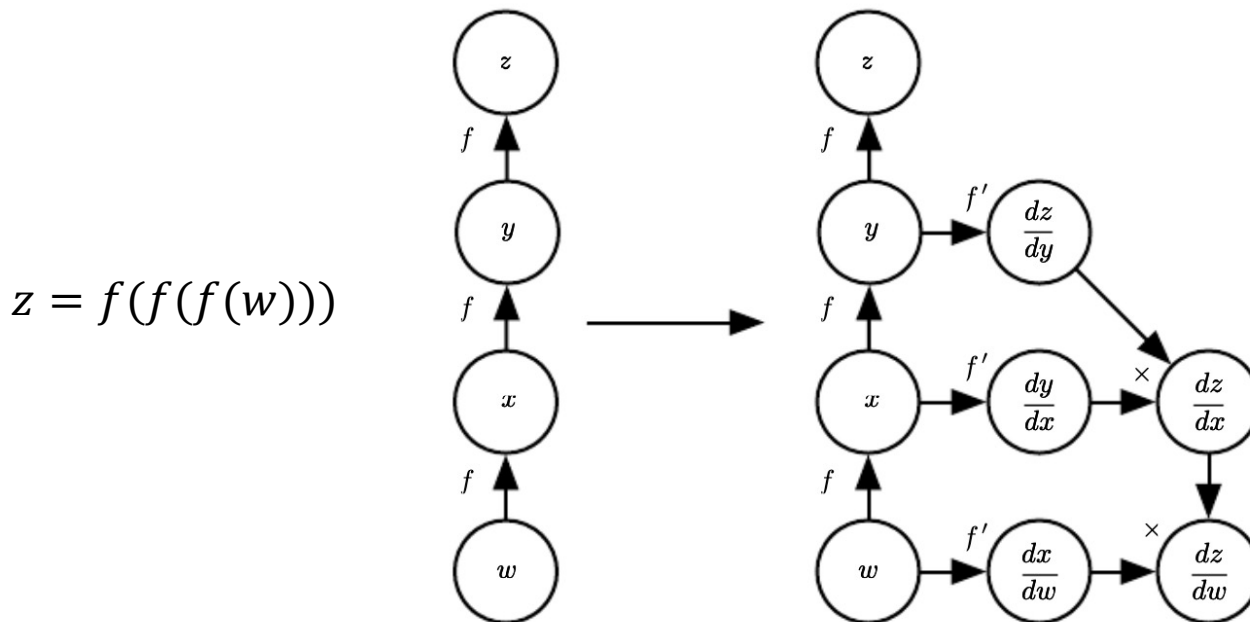
Computational graphs



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

Computational Graphs

- Formalize computation as graphs
- Nodes indicate variables (scalar, vector, tensor or another variable)
- Operations are simple functions of one or more variables
- Our graph language comes with a set of allowable operations



Analytical Derivatives:

$$\begin{aligned}\frac{\partial L}{\partial w} &= \frac{\partial}{\partial w} \left(\frac{1}{2} (\sigma(wx + b) - t)^2 \right) = \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial w} (wx + b)\end{aligned}$$

$$\frac{\partial L}{\partial w} = \frac{dL}{dy} \frac{dy}{dz} \frac{\partial z}{\partial w} = (\sigma(wx + b) - t) \sigma'(wx + b) x$$

$$\begin{aligned}\frac{\partial L}{\partial b} &= \frac{\partial}{\partial b} \left(\frac{1}{2} (\sigma(wx + b) - t)^2 \right) = \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \\ &= (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \\ &= (\sigma(wx + b) - t) \sigma'(wx + b) \frac{\partial}{\partial b} (wx + b)\end{aligned}$$

$$\frac{\partial L}{\partial b} = \frac{dL}{dy} \frac{dy}{dz} \frac{\partial z}{\partial b} = (\sigma(wx + b) - t) \sigma'(wx + b)$$

Disadvantages?

Efficient algorithmic differentiation :

- Computing the loss functions:

$$z = wx + b$$

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

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

- Computing the derivatives:

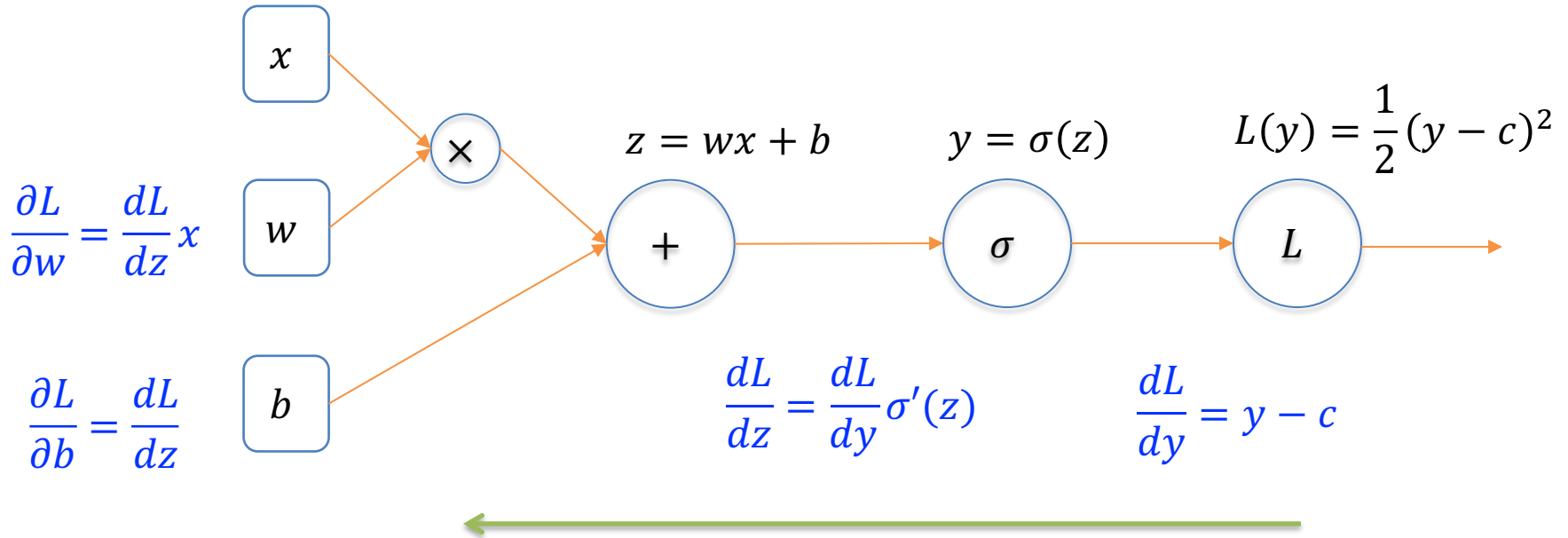
$$\frac{dL}{dy} = y - c$$

$$\frac{dL}{dz} = \frac{dL}{dy} \sigma'(z) = \frac{dL}{dy} \sigma(z)(1 - \sigma(z))$$

$$\frac{\partial L}{\partial w} = \frac{dL}{dz} x \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{dL}{dz}$$

➤ Computation Graphs

Computing the loss functions:



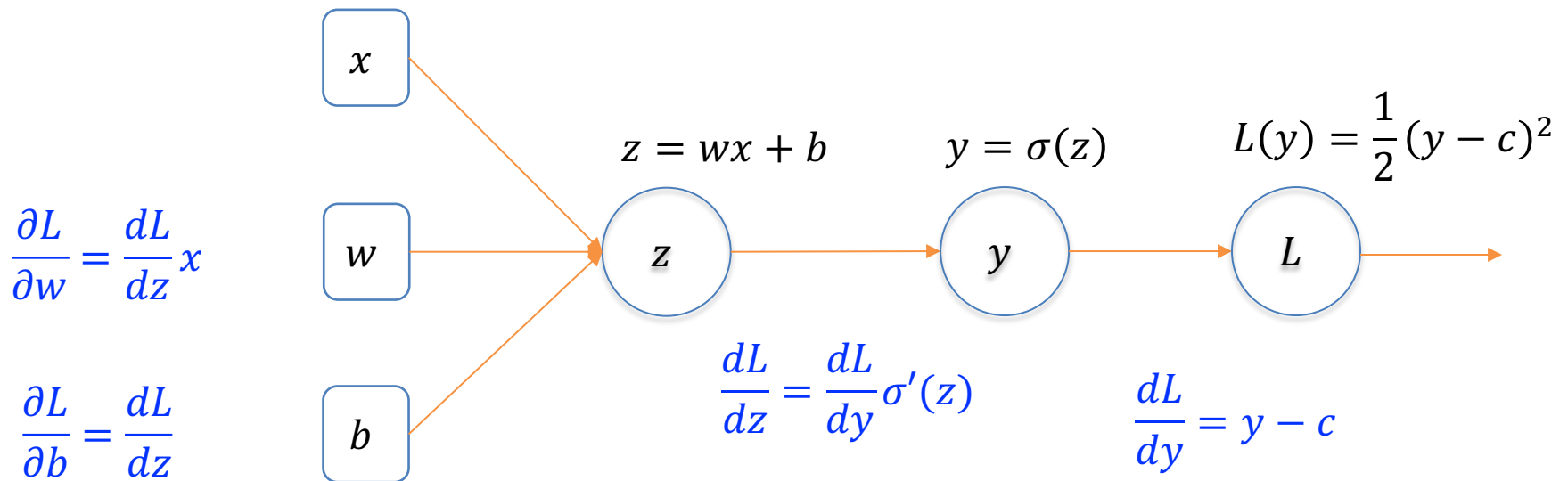
Computing the derivatives:

The goal isn't to obtain closed-form solutions for derivative.

The goal is to write a program that efficiently computes the derivatives.

For example, data $x = 1, c = 1$

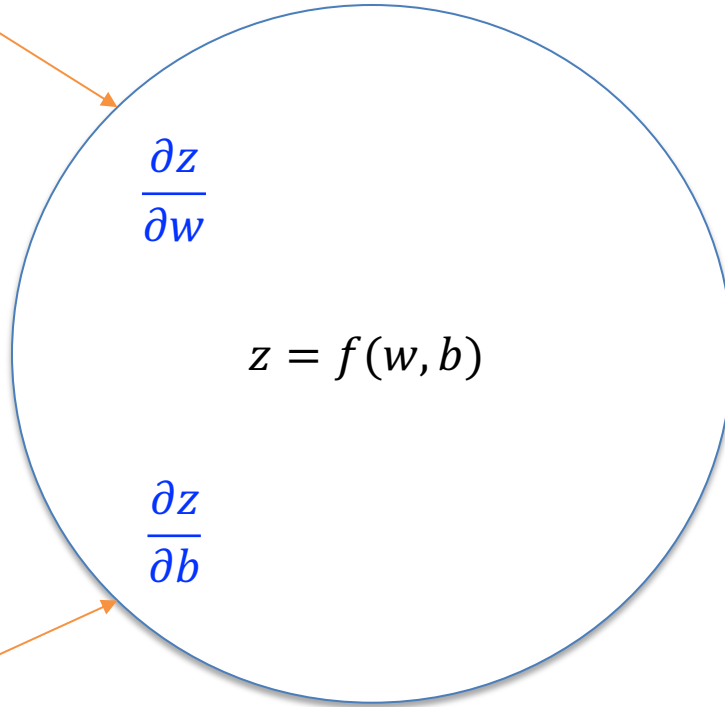
In initial step $w = 0, b = 0$



Chain Rule Again:

w

$$\frac{\partial L}{\partial w} = \frac{dL}{dz} \frac{\partial z}{\partial w}$$



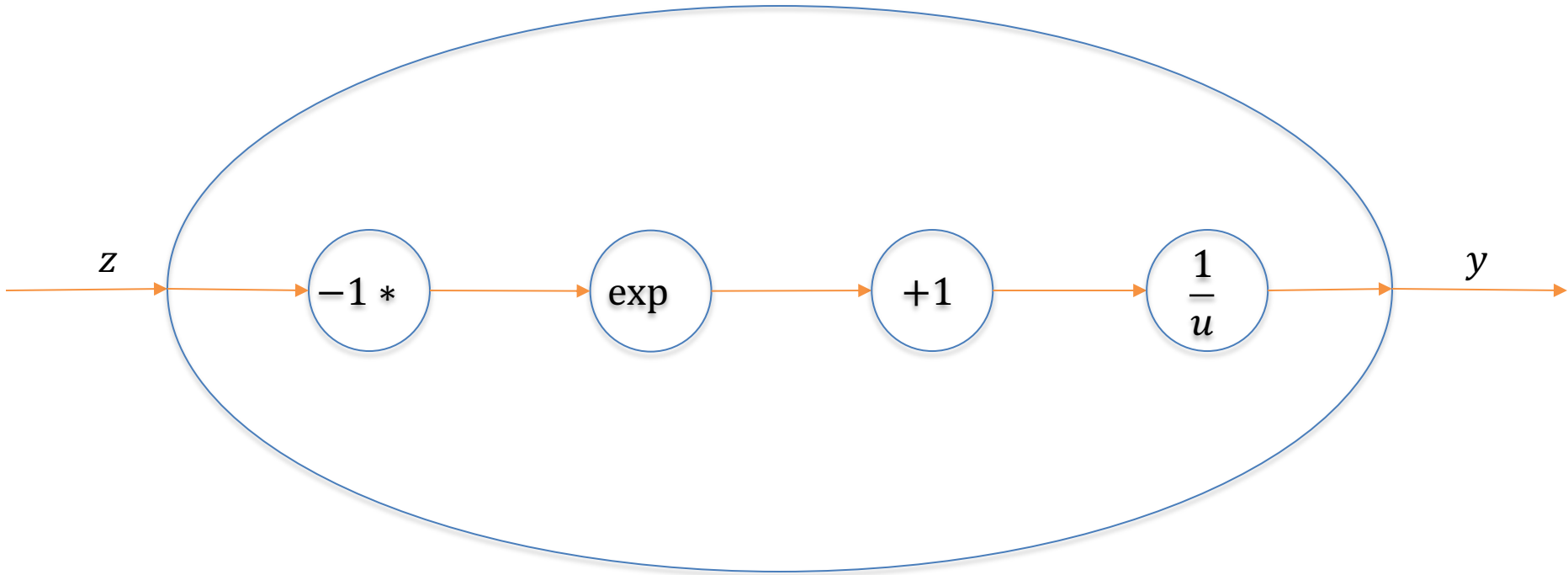
$$\frac{\partial L}{\partial b} = \frac{dL}{dz} \frac{\partial z}{\partial b}$$

b

$$\frac{dL}{dz}$$

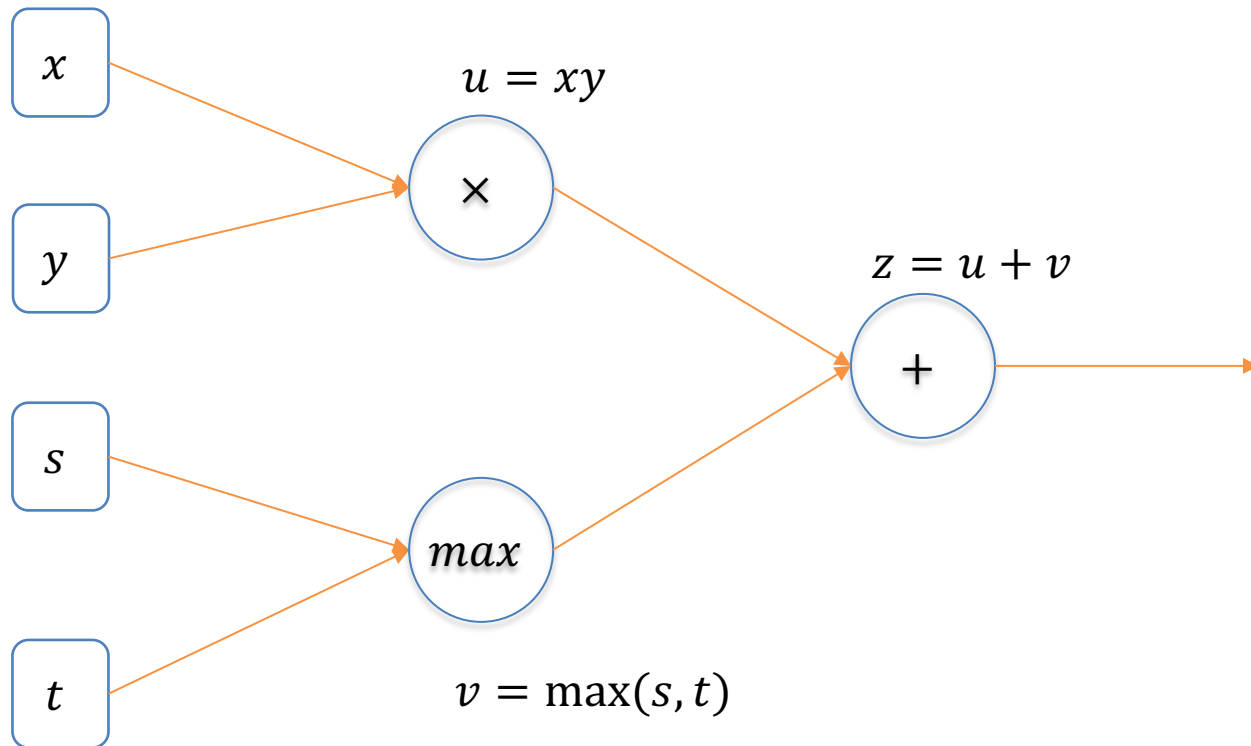
Sigmoid Gate:

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

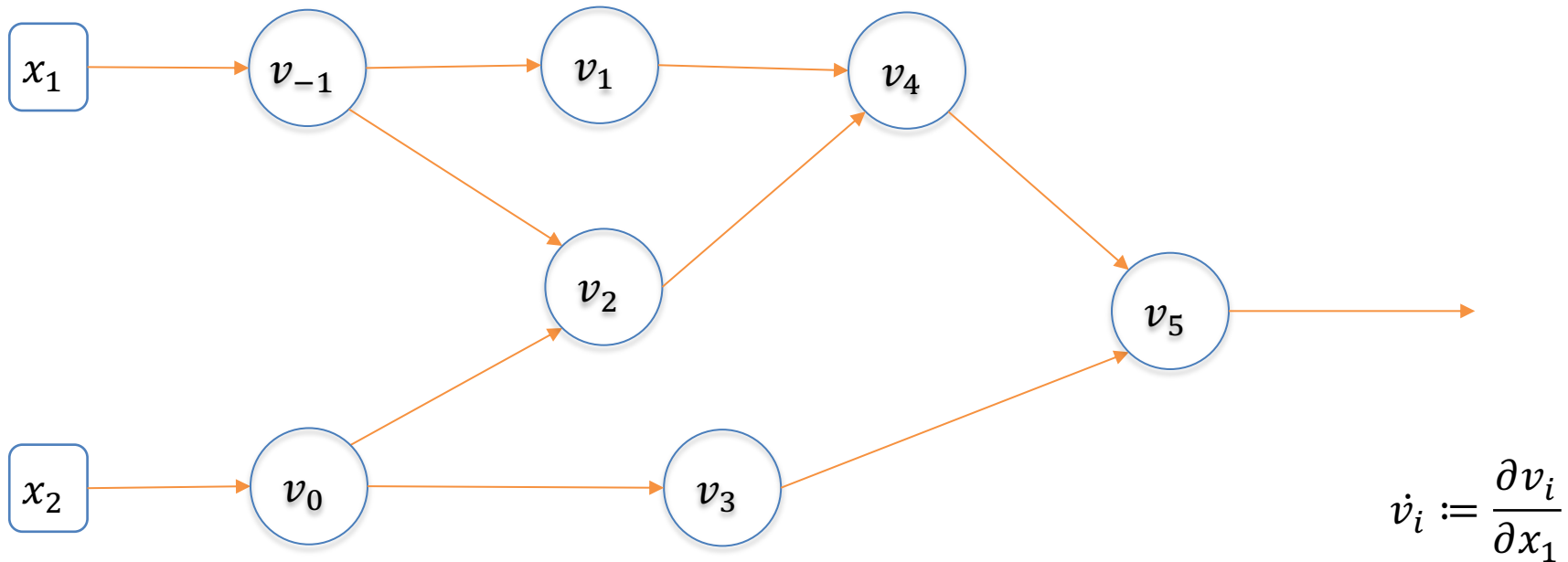


Other gates:

- **add gate:** gradient distributor
- **max gate:** gradient router
- **multiplication gate:** gradient switcher (swap multiplier)
- **copy gate:** gradient adder



Another Example: $f(x_1, x_2) = \ln(x_1) + x_1x_2 - \sin(x_2)$

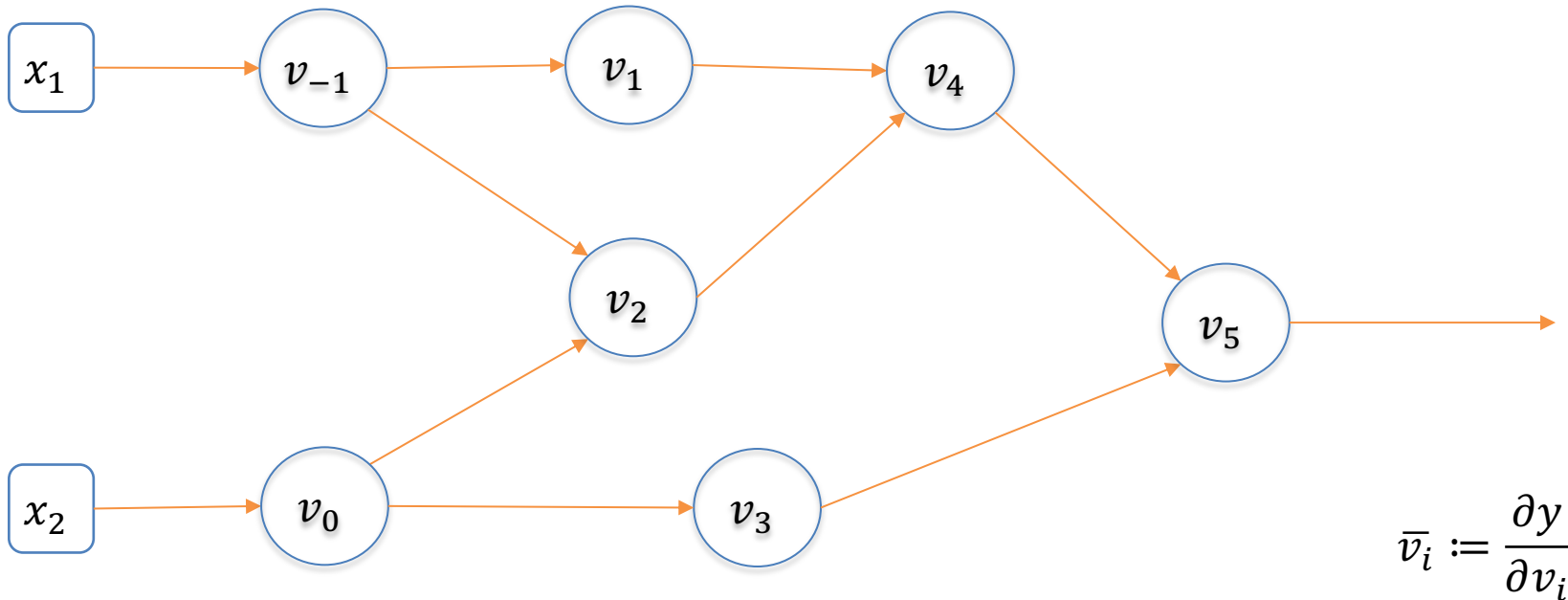


Forward Primal Trace

$v_{-1} = x_1$	$= 2$
$v_0 = x_2$	$= 5$
<hr/>	
$v_1 = \ln v_{-1}$	$= \ln 2$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$
$v_3 = \sin v_0$	$= \sin 5$
$v_4 = v_1 + v_2$	$= 0.693 + 10$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$
<hr/>	
$y = v_5$	$= 11.652$

Forward Tangent (Derivative) Trace

$\dot{v}_{-1} = \dot{x}_1$	$= 1$
$\dot{v}_0 = \dot{x}_2$	$= 0$
<hr/>	
$\dot{v}_1 = \dot{v}_{-1}/v_{-1}$	$= 1/2$
$\dot{v}_2 = \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
$\dot{v}_3 = \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
<hr/>	
$\dot{y} = \dot{v}_5$	$= 5.5$



Forward Primal Trace

$v_{-1} = x_1$	= 2
$v_0 = x_2$	= 5
$v_1 = \ln v_{-1}$	= $\ln 2$
$v_2 = v_{-1} \times v_0$	= 2×5
$v_3 = \sin v_0$	= $\sin 5$
$v_4 = v_1 + v_2$	= $0.693 + 10$
$v_5 = v_4 - v_3$	= $10.693 + 0.959$
$y = v_5$	= 11.652

Reverse Adjoint (Derivative) Trace

$\bar{x}_1 = \bar{v}_{-1}$	= 5.5
$\bar{x}_2 = \bar{v}_0$	= 1.716
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	= $\bar{v}_{-1} + \bar{v}_1 / v_{-1} = 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	= $\bar{v}_0 + \bar{v}_2 \times v_{-1} = 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	= $\bar{v}_2 \times v_0 = 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	= $\bar{v}_3 \times \cos v_0 = -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	= $\bar{v}_4 \times 1 = 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	= $\bar{v}_4 \times 1 = 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	= $\bar{v}_5 \times (-1) = -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	= $\bar{v}_5 \times 1 = 1$
$\bar{v}_5 = \bar{y}$	= 1

Backpropagation

- **Backpropagation:** recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates.
- **Forward:** compute result of an operation and save any intermediates needed for gradient computation in memory.
- **Backward:** apply the chain rule to compute the gradient of the loss function with respect to the inputs.

A non-standard (error signal) notation

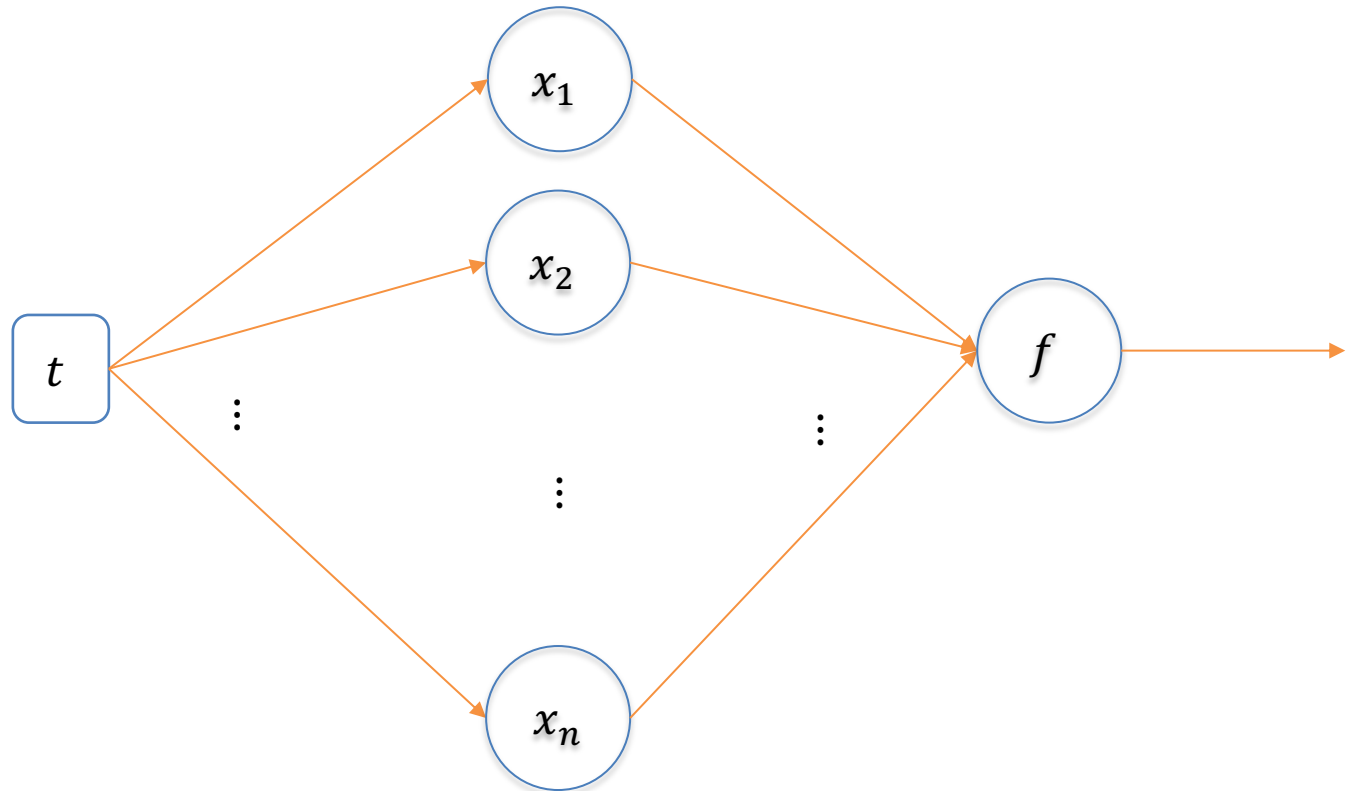
- Use \bar{v} to denote the derivative $\frac{dL}{dv}$, sometimes called the error signal.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).

Backpropagation algorithm:

Let v_1, \dots, v_N be a topological ordering of the computation graph

Multivariate Chain Rule

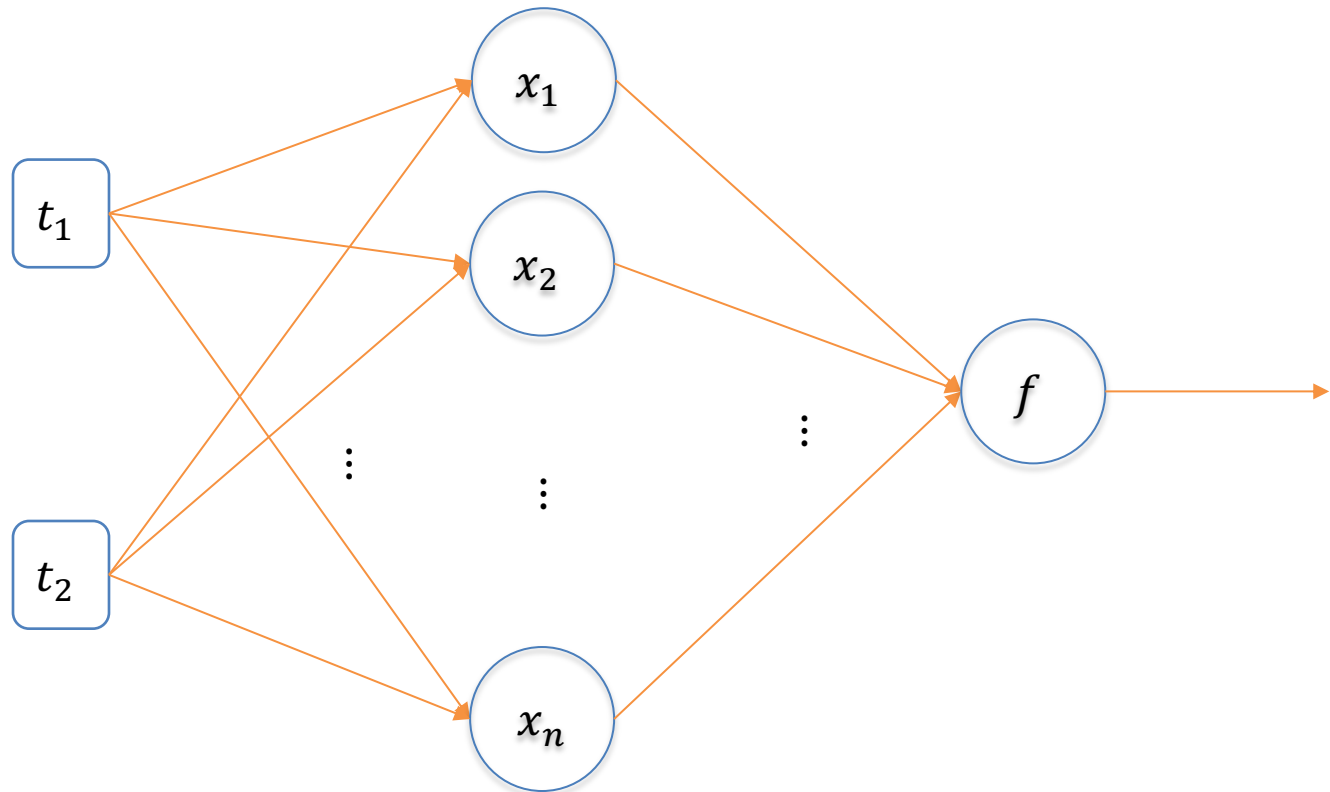
$$\mathbb{R} \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$$



Chain Rule:

$$\frac{df(\vec{x}(t))}{dt} = \frac{\partial f}{\partial x_1} \frac{dx_1}{dt} + \dots + \frac{\partial f}{\partial x_n} \frac{dx_n}{dt}$$

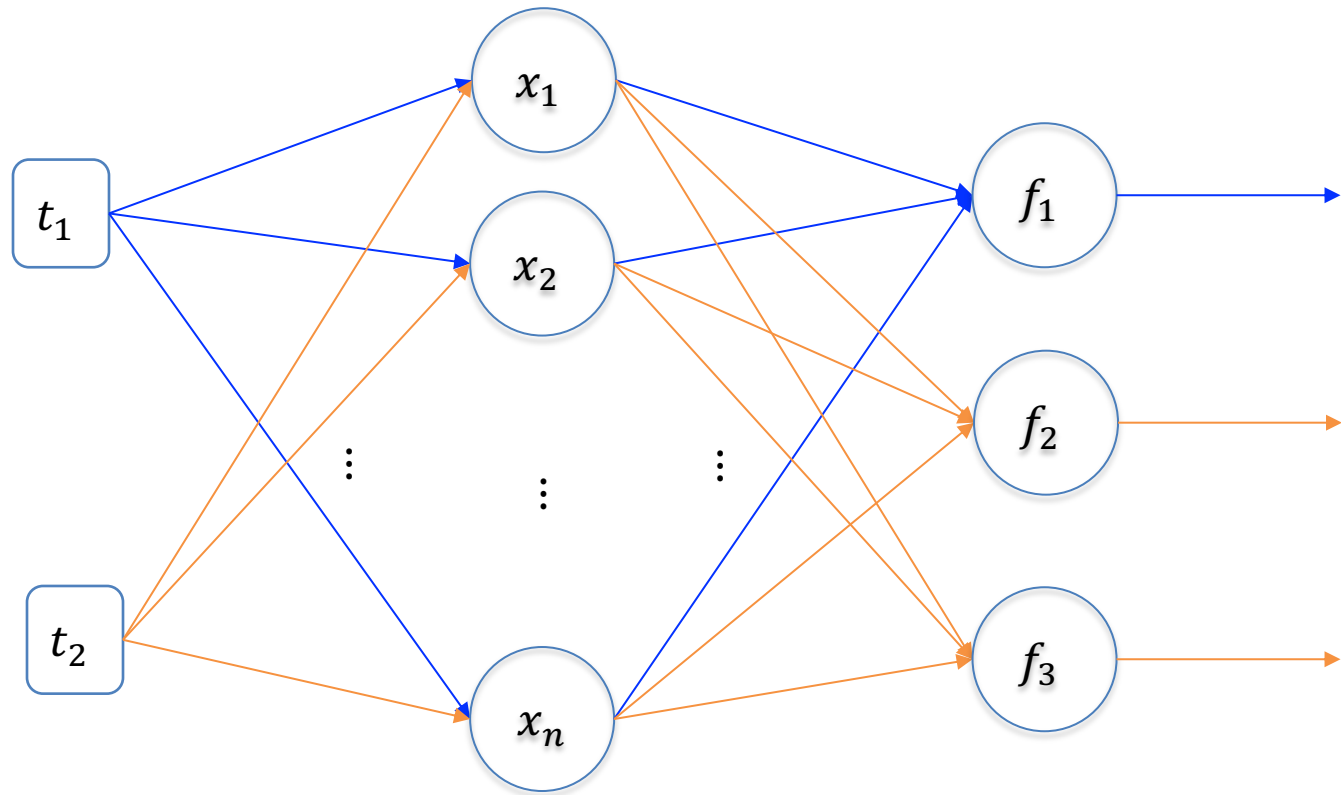
$$\mathbb{R}^2 \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$$



Chain Rule:

$$\frac{\partial f(\vec{x}(\vec{t}))}{\partial t_i} = \frac{\partial f}{\partial x_1} \frac{\partial x_1}{\partial t_i} + \dots + \frac{\partial f}{\partial x_n} \frac{\partial x_n}{\partial t_i}$$

$$\mathbb{R}^2 \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^3$$



Chain Rule:

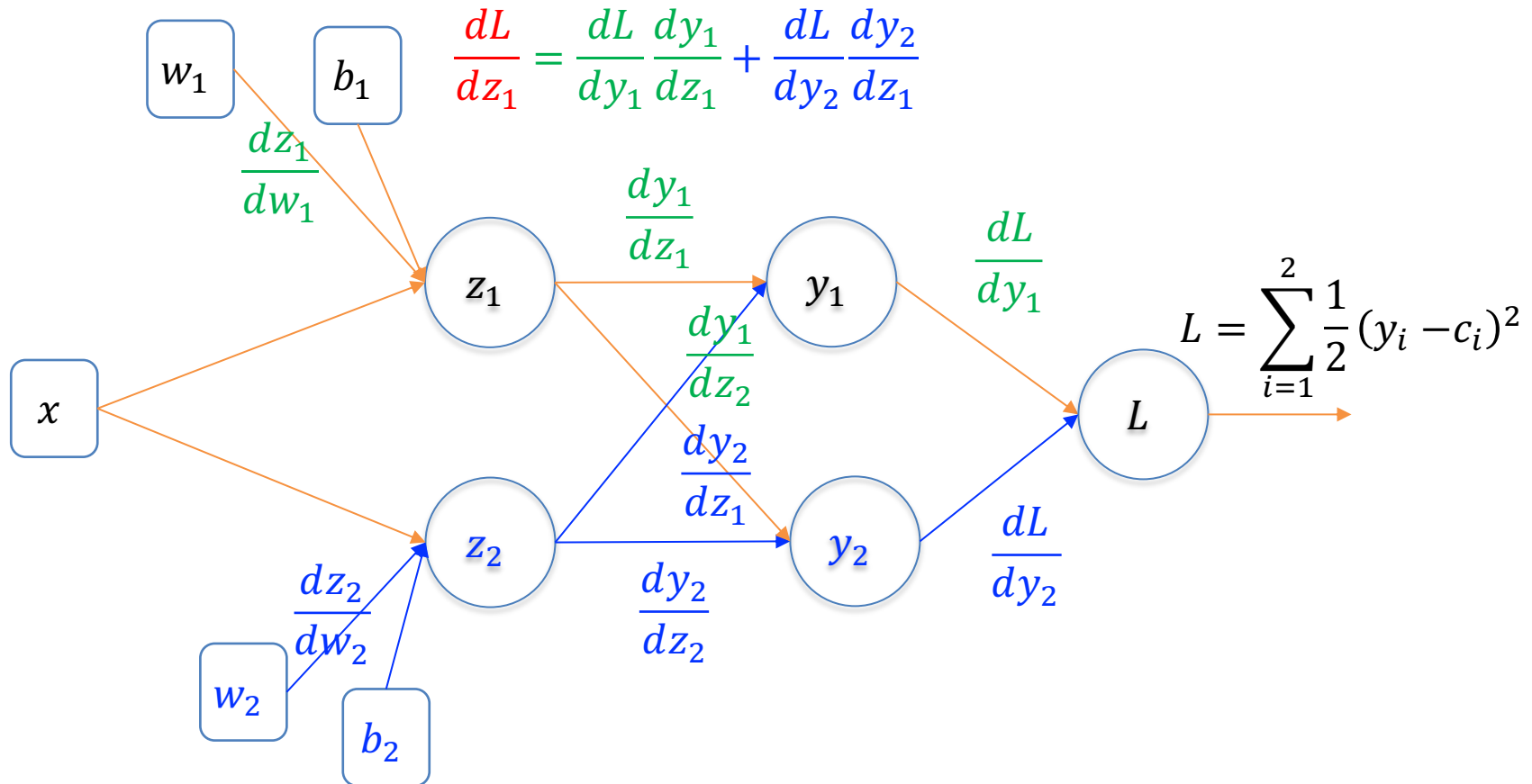
$$\frac{\partial F}{\partial \vec{t}} = \frac{\partial F}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{t}}$$

Or

$$\frac{\partial f_j(\vec{x}(\vec{t}))}{\partial t_i} = \frac{\partial f_j}{\partial x_1} \frac{\partial x_1}{\partial t_i} + \dots + \frac{\partial f_j}{\partial x_n} \frac{\partial x_n}{\partial t_i}$$

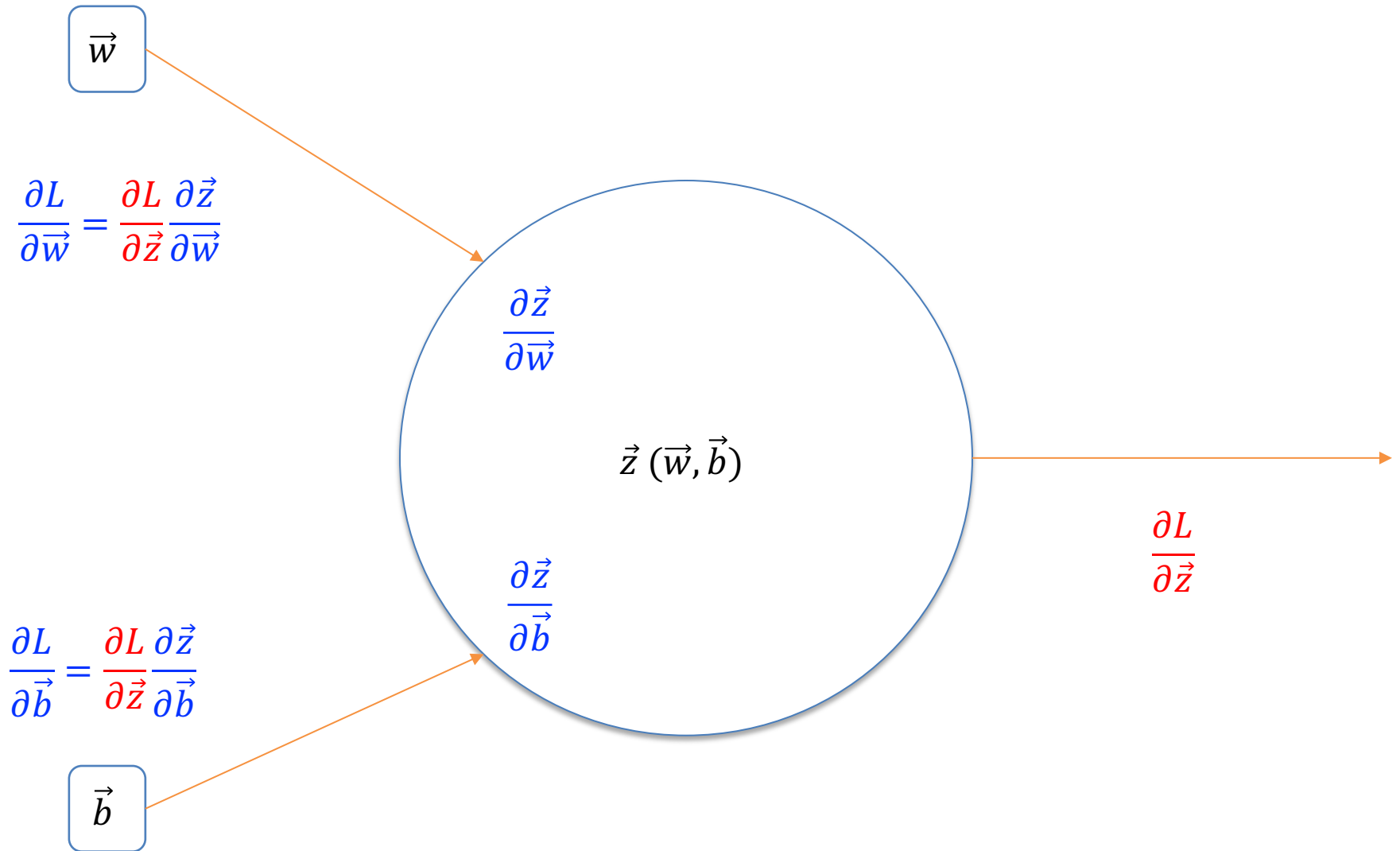
Backprop as message passing:

$$\frac{\partial L}{\partial w_1} = \frac{dL}{dz_1} \frac{dz_1}{dw_1}$$

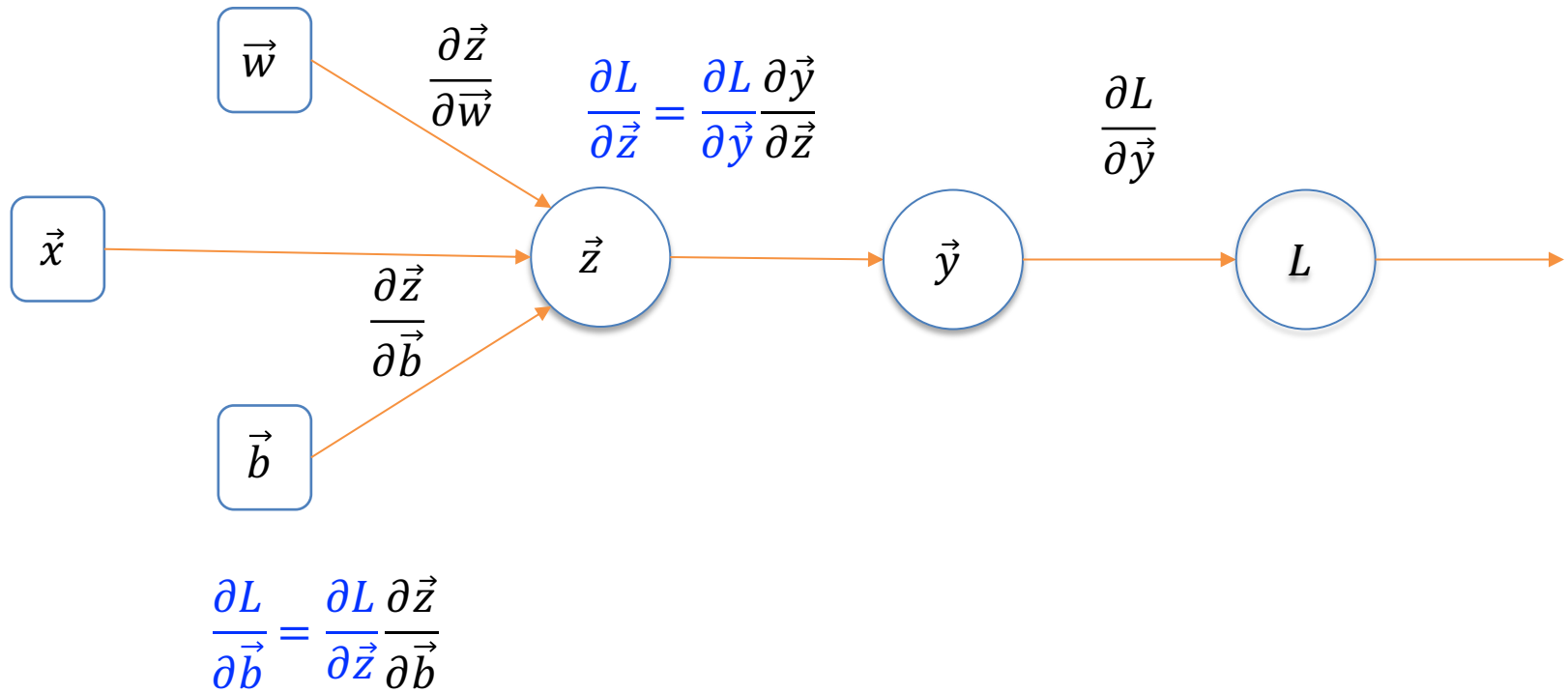


Modularity: each node only has to know how to compute derivatives with respect to its arguments, and doesn't have to know anything about the rest of the graph.

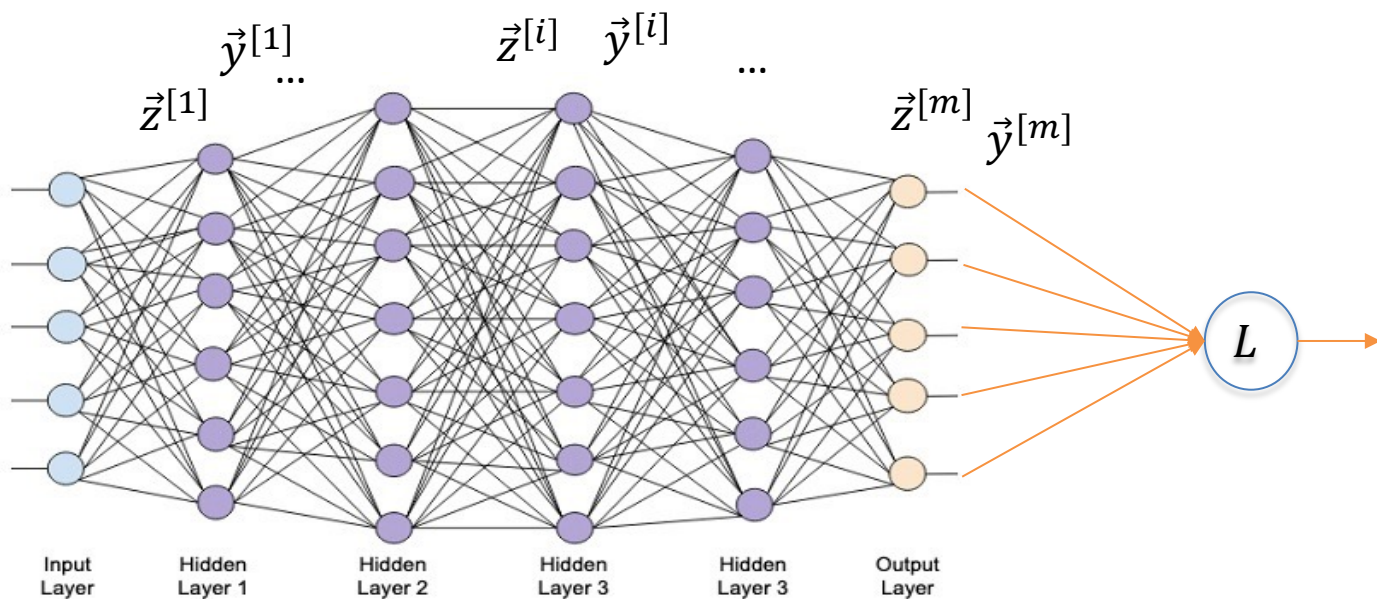
Chain Rule Again (matrix notation)



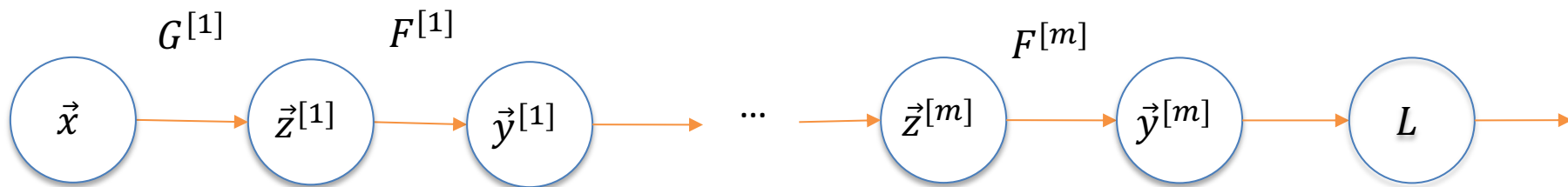
Matrix notation



Go back to Neural Network model:



Matrix Notation:



$$\vec{z}^{[l+1]} = G^{[l+1]}(\vec{y}^{[l]}) = W^{[l]}\vec{y}^{[l]} + \vec{b}^{[l]}$$

$$\vec{y}^{[l]} = F^{[l]}(\vec{z}^{[l]})$$

Model: $h_{\Theta}(\vec{x}) := F^{[m]} \circ G^{[m]} \circ \dots \circ F^{[2]} \circ G^{[2]} \circ F^{[1]} \circ G^{[1]}$

Cost: $J(\Theta) := L(h_{\Theta}(X), \vec{y})$, where $L(-, -)$ is a metric.

We need to calculate three type of derivatives along the computational graph, then use the backpropagation method:

(1) Derivative of the metric function $\frac{\partial L}{\partial \vec{z}^{[m]}}$

(2) Derivative of the activation function $\frac{\partial F^{[i]}}{\partial \vec{z}^{[i]}}$

(3) Derivative of the linear function $\frac{\partial G^{[i]}}{\partial \vec{y}^{[i]}}$

(1) Derivative for the **metric function** $\frac{\partial L}{\partial \vec{z}^{[m]}}$

1. Mean Square Error for regression

$$J(\vec{z}) = \frac{1}{n} \|\vec{z} - \vec{y}\|^2$$

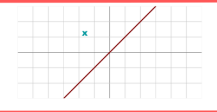


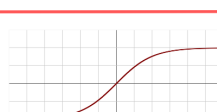




2. Cross-Entropy cost for classification

$$J(\vec{z}) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K \mathbb{I}(y^{(i)} = k) \ln(z_i)$$

3. Hinge loss, 0–1 loss, ...

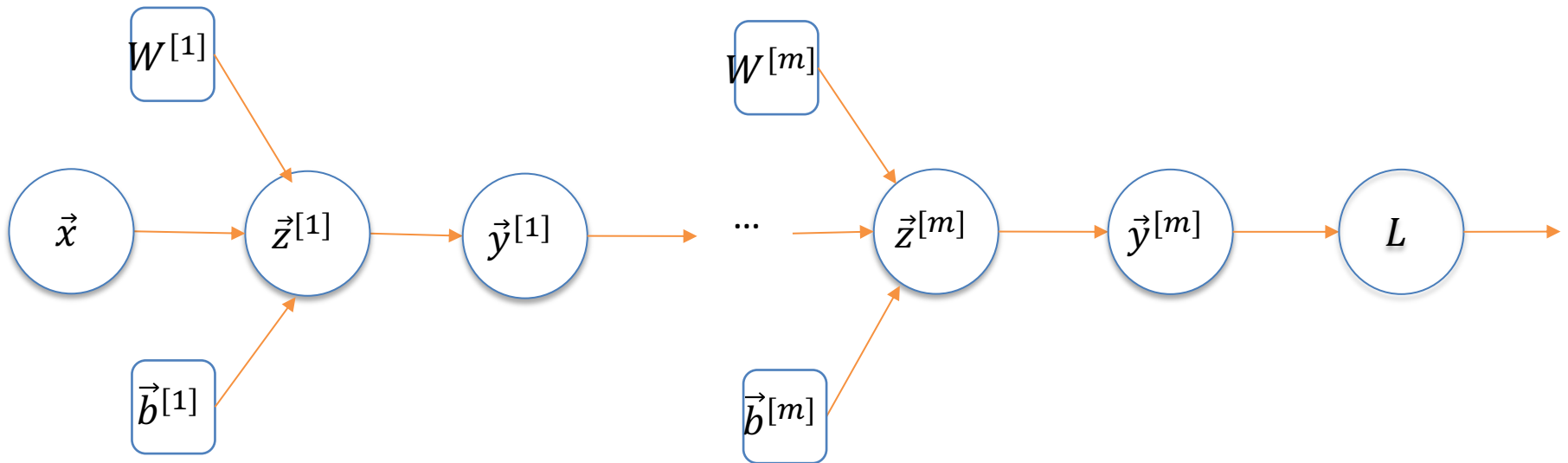
(2) Derivative of the activation function

$$\frac{\partial F^{[i]}}{\partial \vec{z}^{[i]}}$$

ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$ *
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$ *
Softplus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$	$(0, 1)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$(-1, 1)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$[0, \infty)$

$$\vec{z}^{[l+1]} = W^{[l]}\vec{y}^{[l]} + \vec{b}^{[l]}$$

$$\vec{y}^{[l]} = F^{[l]}(\vec{z}^{[l]})$$



Notation: $\vec{y}^{[0]} := \vec{x}$ and $\vec{z}^{[m+1]} = L$

(3) Derivative of the linear function $\frac{\partial \vec{z}^{[l+1]}}{\partial W^{[l]}}$

$$\vec{z}^{[l+1]} = W^{[l]} \vec{y}^{[l]} + \vec{b}^{[l]}$$

$$\frac{\partial \vec{z}^{[l+1]}}{\partial \vec{y}^{[l]}} = W^{[l]}$$

$$z_i^{[l+1]} = \sum_j w_{ij}^{[l]} y_j^{[l]} + b_i^{[l]}$$

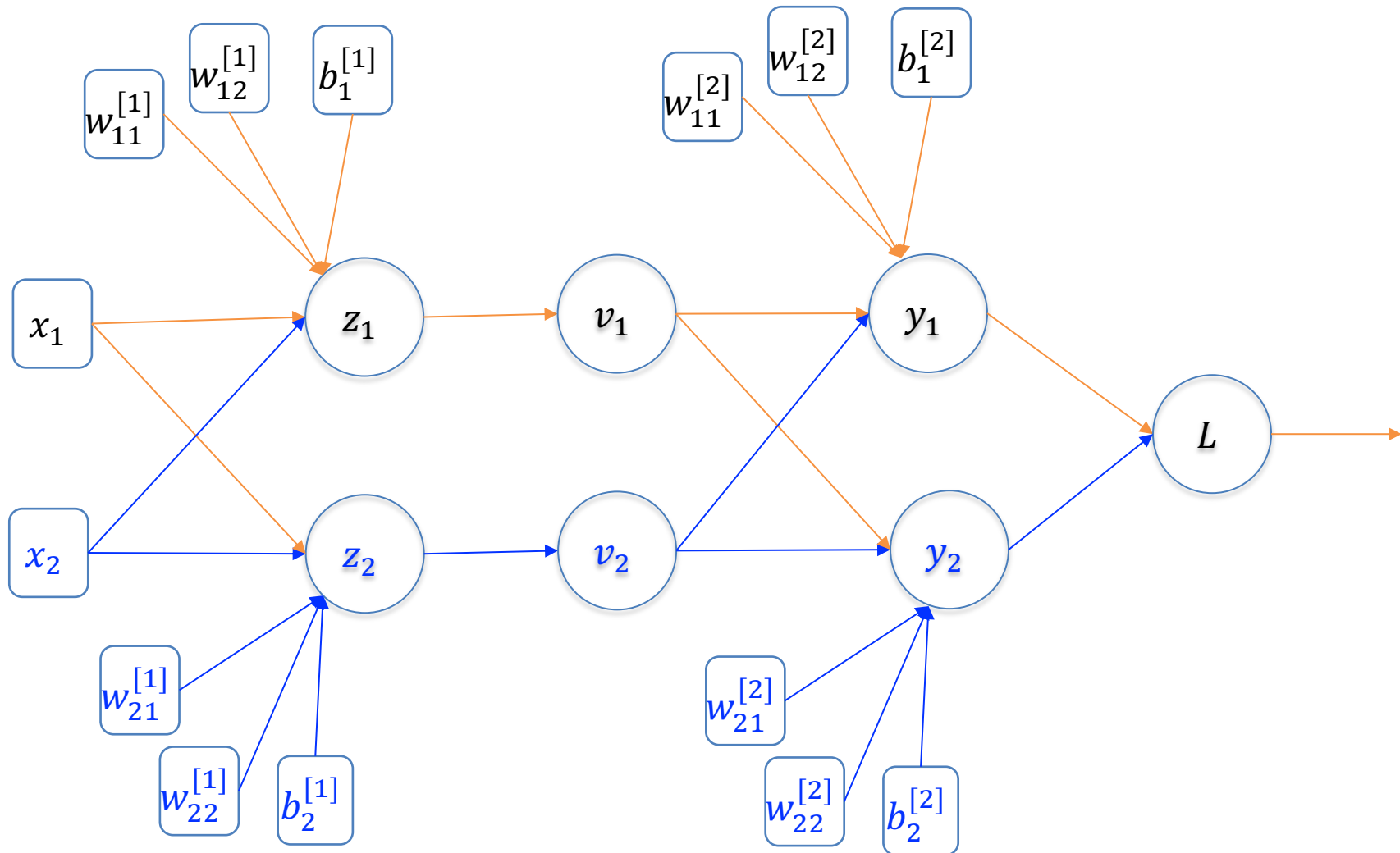
$\frac{\partial \vec{z}^{[l+1]}}{\partial W^{[l]}}$ is an $m \times (m \times d)$ matrix with $\frac{\partial z^{[l+1]}_k}{\partial w_{ij}^{[l]}} = \begin{cases} y_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$

Proposition:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{\partial L}{\partial \vec{z}^{[l+1]}} \frac{\partial \vec{z}^{[l+1]}}{\partial W^{[l]}} = \vec{y} \frac{\partial L}{\partial \vec{z}^{[l+1]}}$$

$$\frac{\partial L}{\partial \vec{b}^{[l]}} = \frac{\partial L}{\partial \vec{z}^{[l+1]}} \frac{\partial \vec{z}^{[l+1]}}{\partial \vec{b}^{[l]}} = \frac{\partial L}{\partial \vec{z}^{[l+1]}}$$

Multilayer Perceptron (multiple outputs)



Forward pass:

$$z_i = \sum_j w_{ij}^{[1]} x_j + b_i^{[1]}$$

$$h_i = \sigma(z_i)$$

$$y_i = \sum_j w_{ij}^{[2]} v_j + b_i^{[2]}$$

$$L = \sum_k \frac{1}{2} (y_k - c_k)^2$$

Backward pass:

$$\frac{\partial L}{\partial y_i}$$
$$\frac{\partial L}{\partial w_{ij}^{[2]}} = \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial w_{ij}^{[2]}} = \frac{\partial L}{\partial y_i} v_j$$

$$\frac{\partial L}{\partial b_i^{[2]}} = \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial b_i^{[2]}} = \frac{\partial L}{\partial y_i}$$

$$\frac{\partial L}{\partial v_i} = \sum_k \frac{\partial L}{\partial y_k} \frac{\partial y_k}{\partial v_i} = \sum_k \frac{\partial L}{\partial y_k} w_{ki}^{[2]}$$

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial v_i} \frac{\partial v_i}{\partial z_i} = \frac{\partial L}{\partial v_i} \sigma'(z_i)$$

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial w_{ij}^{[1]}} = \frac{\partial L}{\partial z_i} x_j$$

$$\frac{\partial L}{\partial b_i^{[1]}} = \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial b_i^{[1]}} = \frac{\partial L}{\partial z_i}$$

Notations:

- Recall the **numerator layout** convention for the derivatives

$$f: \mathbb{R}^n \rightarrow \mathbb{R}$$

$$\frac{\partial f}{\partial \vec{x}} := \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \end{bmatrix}$$

$$F: \mathbb{R}^n \rightarrow \mathbb{R}^m$$

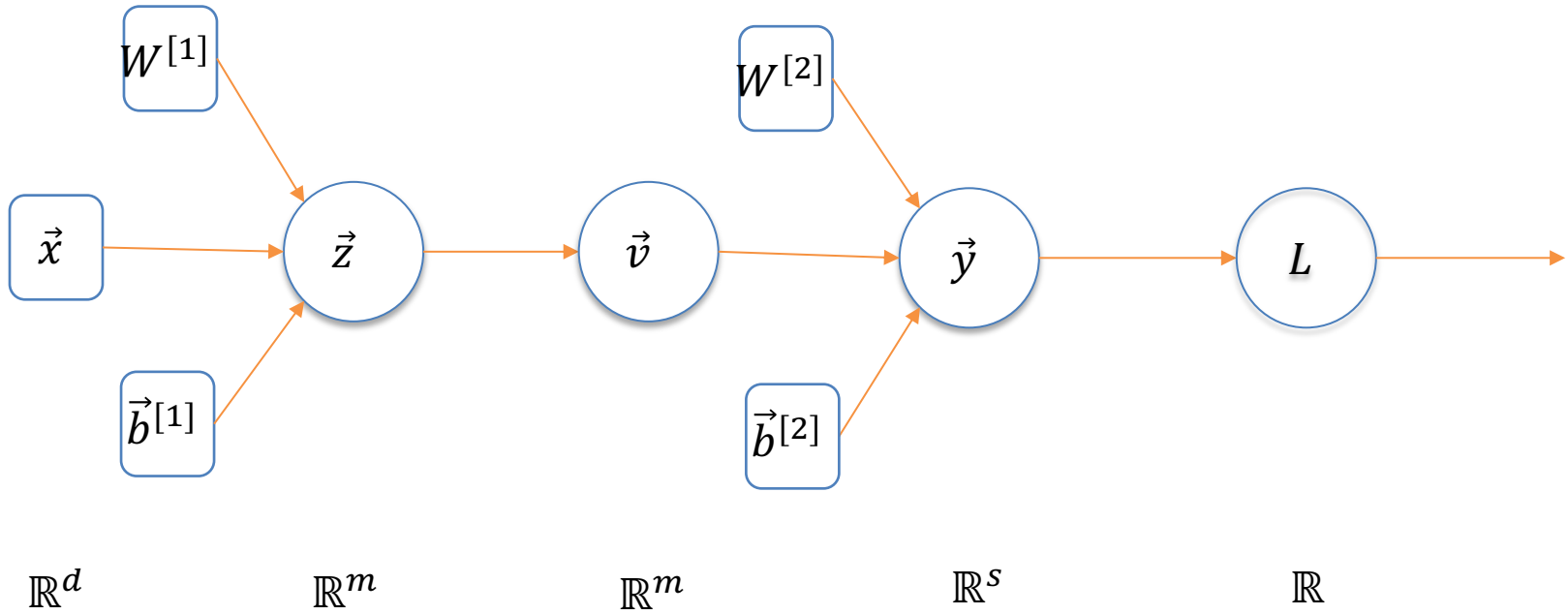
$$\frac{\partial F}{\partial \vec{x}} := \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- Using numerator layout:** If $F(\vec{x}) = A\vec{x}$, then $\frac{\partial F}{\partial \vec{x}} = A$

- If $f: \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$, the derivative of f is still defined to be

$$\frac{\partial f}{\partial X} := \begin{bmatrix} \frac{\partial f}{\partial x_{11}} & \dots & \frac{\partial f}{\partial x_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_{m1}} & \dots & \frac{\partial f}{\partial x_{mn}} \end{bmatrix}$$

Matrix Notation (numerator layout)

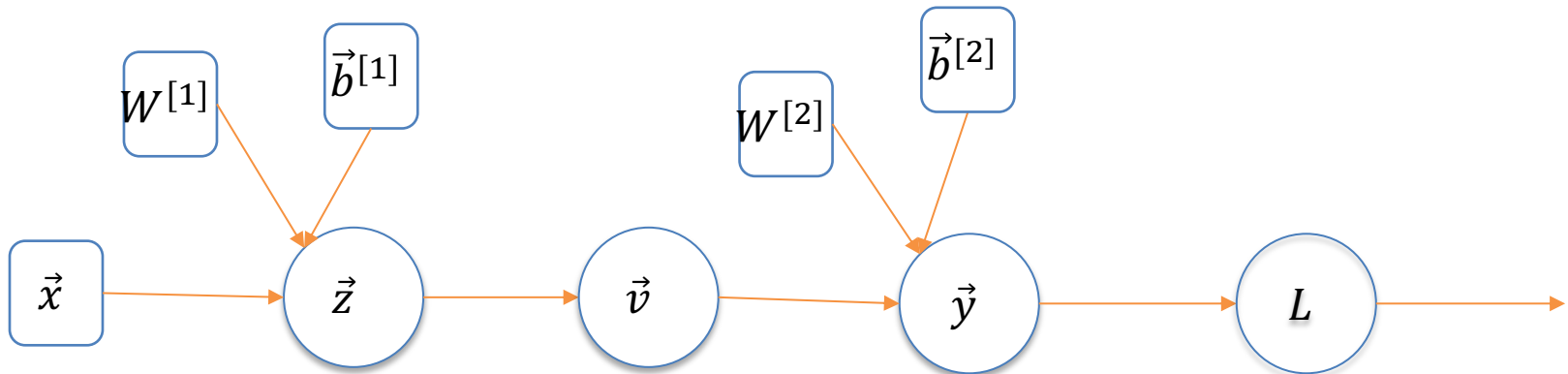


Forward pass: $\vec{z} = W^{[1]}\vec{x} + \vec{b}^{[1]}$

$$\vec{v} = \sigma(\vec{z})$$

$$\vec{y} = W^{[2]}\vec{v} + \vec{b}^{[2]}$$

$$L = \frac{1}{2} \|\vec{y} - \vec{c}\|^2$$



Backward pass:

$$\frac{\partial L}{\partial \vec{y}} = \vec{y} - \vec{c}$$

$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial W^{[2]}} = \vec{v} \frac{\partial L}{\partial \vec{y}}$$

$$\frac{\partial L}{\partial \vec{b}^{[2]}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{b}^{[2]}} = \frac{\partial L}{\partial \vec{y}}$$

$$\frac{\partial L}{\partial \vec{v}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{v}} = \frac{\partial L}{\partial \vec{y}} W^{[2]}$$

$$\frac{\partial L}{\partial \vec{z}} = \frac{\partial L}{\partial \vec{v}} \frac{\partial \vec{v}}{\partial \vec{z}} = \frac{\partial L}{\partial \vec{v}} \sigma'(\vec{z})$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial W^{[1]}} = \vec{x} \frac{\partial L}{\partial \vec{z}}$$

$$\frac{\partial L}{\partial \vec{b}^{[1]}} = \frac{\partial L}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial \vec{b}^{[1]}} = \frac{\partial L}{\partial \vec{z}}$$

Remark:

$$\vec{z} = W^{[1]}\vec{x} + \vec{b}^{[1]}$$

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial W^{[1]}} \quad 1 \times m \times d \text{ matrix}$$

$$\frac{\partial L}{\partial \vec{z}} := \left[\frac{\partial L}{\partial z_1} \quad \dots \quad \frac{\partial L}{\partial z_m} \right] \quad 1 \times m \text{ matrix}$$

$$\frac{\partial \vec{z}}{\partial W^{[1]}} \text{ is an } m \times (m \times d) \text{ matrix with } \frac{\partial z_k}{\partial w_{ij}^{[1]}} = \begin{cases} x_j & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$

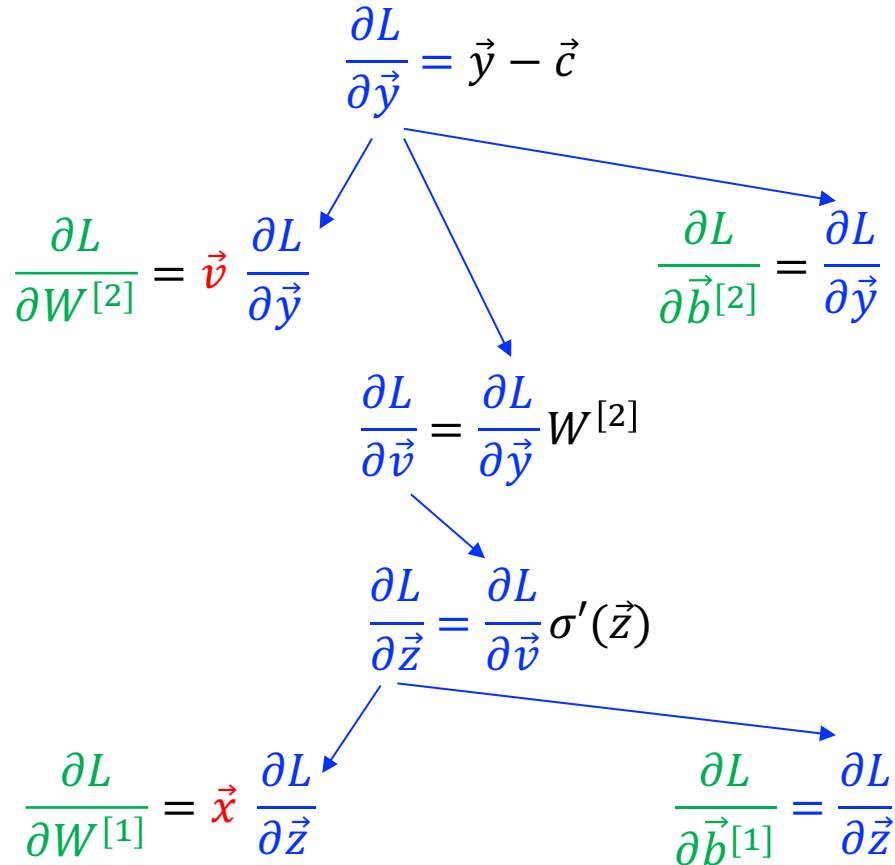
$$\frac{\partial L}{\partial W^{[i]}} = \frac{\partial L}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial W^{[i]}} = \dots = \vec{v} \frac{\partial L}{\partial \vec{y}}$$

Here, we avoid huge matrix(tensor) multiplications. When backpropagating through a linear layer, using minibatches, we need to similar trick:

<http://cs231n.stanford.edu/handouts/linear-backprop.pdf>

Back-Propagation Algorithm for two-layer neural network:

1. Compute the function values $\vec{z}, \vec{v}, \vec{y}$ (in **forward** pass)
2. Compute the derivative values (in **backward** pass)

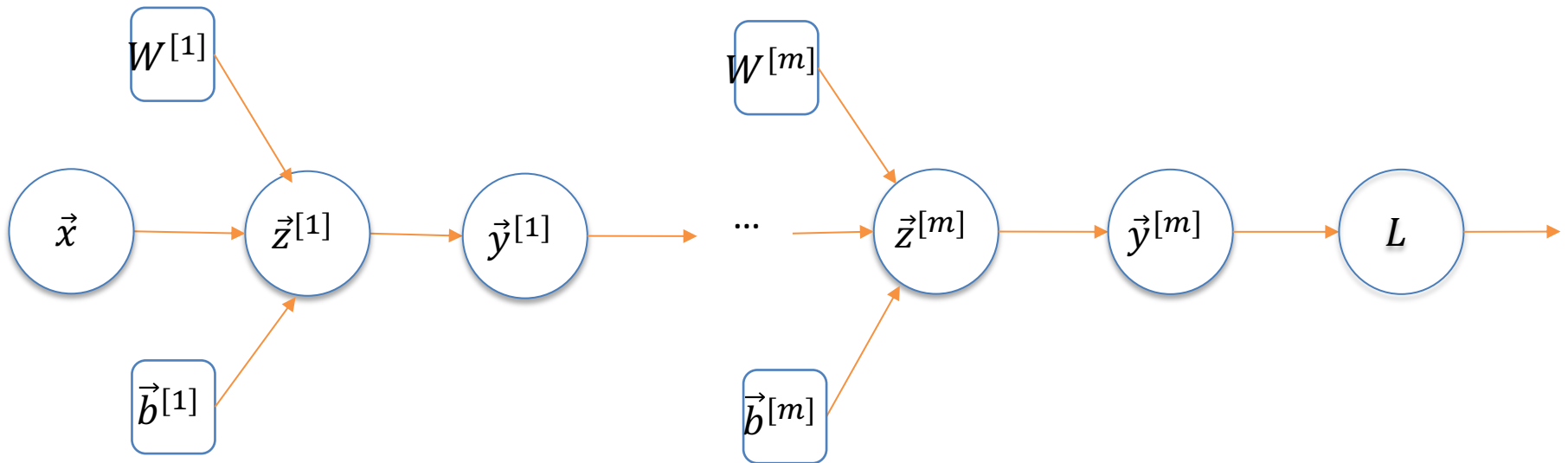


Here, σ can be replaced by any activation function.

Back-Propagation for multi-layer neural network:

$$\vec{z}^{[l+1]} = W^{[l]}\vec{y}^{[l]} + \vec{b}^{[l]}$$

$$\vec{y}^{[l]} = F^{[l]}(\vec{z}^{[l]})$$



Notation: $\vec{y}^{[0]} := \vec{x}$ and $\vec{z}^{[m+1]} = L$

Back-Propagation Algorithm for multi-layer neural network:

1. Compute the values $\vec{z}^{[k]}$ and $\vec{y}^{[k]}$ (in **forward** pass, for $k = 1, \dots, m$)
2. Compute the derivative values (in **backward** pass)

For $k = m, \dots, 1$, **do**

$$\frac{\partial L}{\partial \vec{y}^{[m]}} = \text{depending on the metric}$$

$$\frac{\partial L}{\partial \vec{y}^{[k]}} = \frac{\partial L}{\partial \vec{z}^{[k+1]}} W^{[k]}$$

$$\frac{\partial L}{\partial \vec{z}^{[k]}} = \frac{\partial L}{\partial \vec{y}^{[k]}} \sigma'(\vec{z}^{[k]})$$

$$\frac{\partial L}{\partial W^{[1]}} = \vec{y}^{[k]} \frac{\partial L}{\partial \vec{z}^k}$$

$$\frac{\partial L}{\partial \vec{b}^{[k]}} = \frac{\partial L}{\partial \vec{z}^{[k]}}$$

Here, σ can be replaced by any activation function.

Machine learning frameworks

Machine learning frameworks like MATLAB, TensorFlow, PyTorch, and MxNet combine

- (1) automatic differentiation via backprop (Backpropagation)
- (2) automatic compilation of matrix multiplies to GPUs for fast compute,
- (3) built-in functions and learning examples that make it easy to write and train neural networks. Mostly use Python as the front-end interface.

These frameworks make it easy to train deep neural networks and get good performance and scalability, even for people who do not understand the principles behind their operation. This is a major driving force behind the deep learning revolution!

Question: “Why do we have to write the backward pass when frameworks in the real world, such as TensorFlow, compute them for you automatically?”

Answer: Problems might surface related to underlying gradients when debugging your model (e.g. vanishing or exploding gradients)

<https://karpathy.medium.com/yes-you-should-understand-backprop-e2f06eab496b>

Similar questions: Why do we learn math computations of models like ridge, lasso, logistics, LDA/QDA, SVM, etc. as they already built in sk-learn, statsmodels, MATLAB, R, etc.?

Backpropagation is used to train the overwhelming majority of neural nets today. Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.

The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

ABCD

References:

Books: [Murphy 1] Sec 13.3
[Bishop]Section 5.3
[G. Strang] Linear Algebra and learning from data. SecVII.3

Deep Learning: <https://www.deeplearningbook.org/contents/mlp.html>

Lecture Notes: Google Search: “Backpropagation lecture notes”

- <https://www.cs.princeton.edu/~rlivni/cos511/lectures/lect16.pdf>
- <https://www.cs.cornell.edu/courses/cs4787/2019sp/notes/lecture12.pdf>
- <https://www.cs.cmu.edu/~mgormley/courses/10601-s17/slides/lecture20-backprop.pdf>
- https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec06.pdf
- http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture04.pdf
- <http://6.869.csail.mit.edu/fa17/lecture/lecture6deepnets.pdf>

More notes:

- <http://cs231n.stanford.edu/handouts/linear-backprop.pdf>
- https://cs229.stanford.edu/notes2020spring/cs229-notes-deep_learning.pdf
- https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec10.pdf

A Survey Paper: (Including 200 references.)

- **Automatic Differentiation in Machine Learning: a Survey**
<https://www.jmlr.org/papers/volume18/17-468/17-468.pdf>