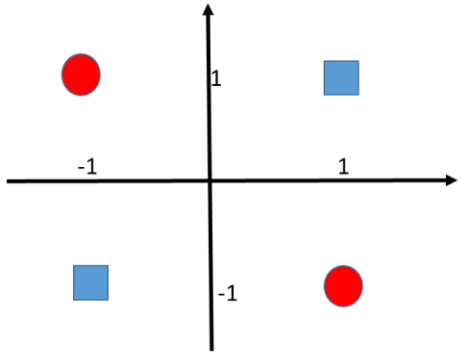
Deep Neural Network (DNN)
- computational graph and automatic differentiation

Liang Liang

We have learned those Models for classification and regression

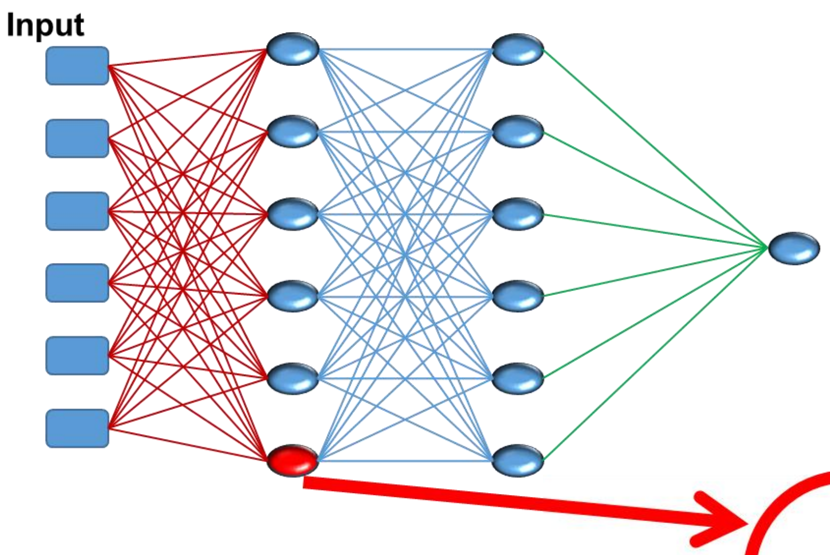
- Linear Models can only describe linear relationship between input x and output y
  - Classifiers: Logistic regression classifier, Linear SVM
  - Regressors: Linear regression model, Linear SVR
- Bayes Models to describe nonlinear relationship
  - not easy to get class PDF
  - GMM may not work for your data
- Trees and Forests to describe nonlinear relationship
  - good for tabular data
  - not so good for image data, voice data, and text data



The four points are Not linearly separable

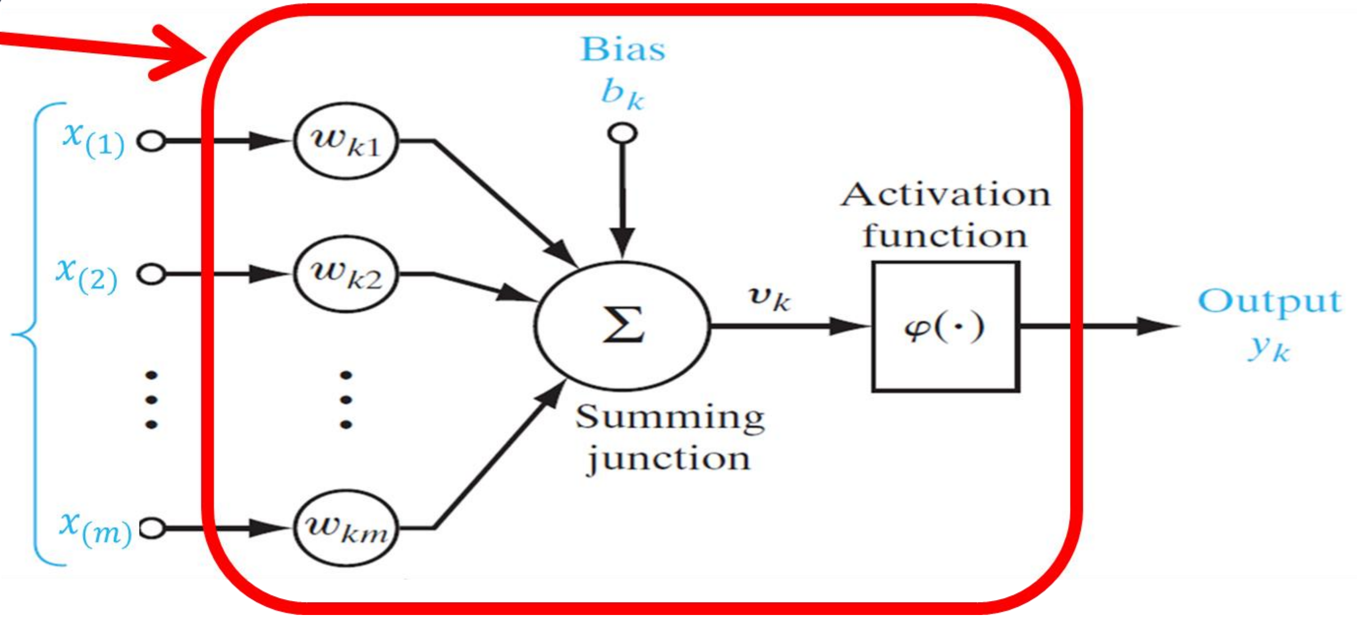
- We could use feature transform $\phi(x)$ to model nonlinear relationship.
  - Polynomial features: $x \Rightarrow [x, x^2, x^3, \ldots]$
    Not easy to find a good feature transform
  - Kernel trick in SVM/SVR (implicit feature transform)
    Not easy to find a good kernel function

A Neural Network is a nonlinear model if the activation functions are nonlinear



This network is also called **M**ulti-**L**ayer **P**erceptron (**MLP**)

One Unit (Perceptron)



A Multi-layer (≥1 hidden layer) Neural Network is a Universal Function Approximator

- Universal Approximation Theorem
  - Every bounded continuous function can be approximated with arbitrary small error, by a network with one hidden layer
  - Activation functions need to be locally bounded and piecewise continuous

- Deep network vs shallow network

A continuous function can be approximated by a deep network or a shallow network. The deep network usually uses less number of units.

Training a Neural Network using Gradient Descent: Forward Pass, Backward Pass, Parameter Update

- Forward Pass to perform inference

  compute the output of each unit/layer

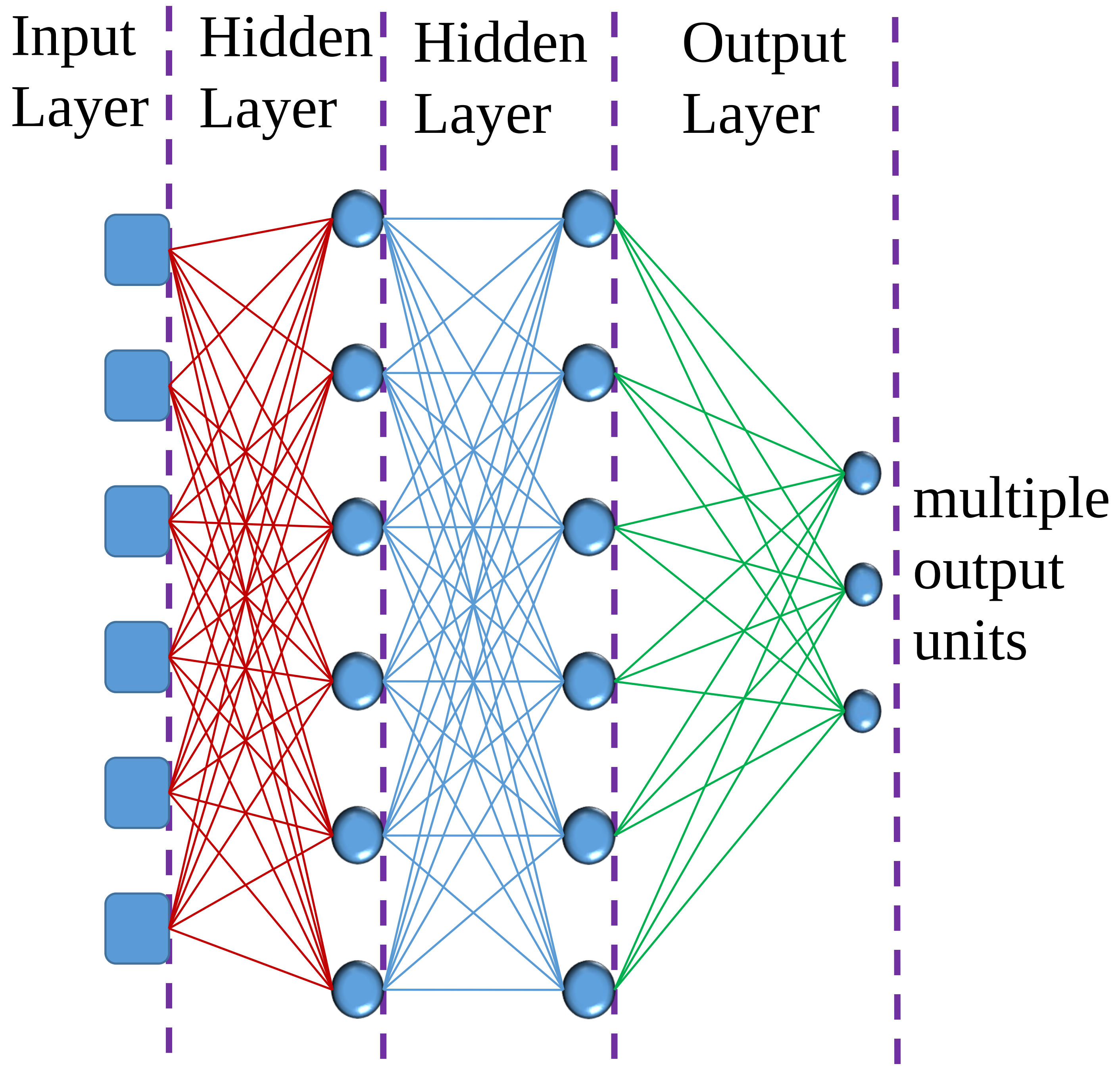
- Backward Pass to perform learning

  compute the derivatives of the loss

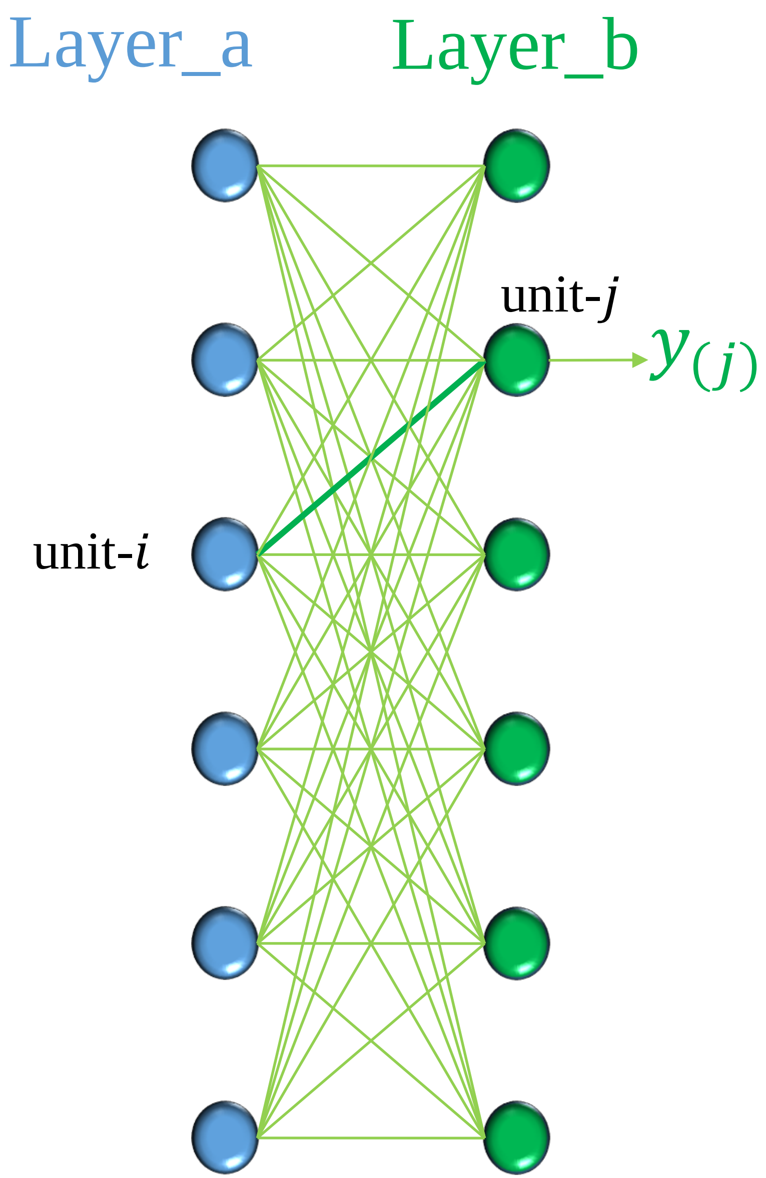
- Parameter Update to adjust the parameters using derivatives

  Update: $w \leftarrow w - \eta \frac{\partial L}{\partial w}$

  η is called learning rate



In general,
a neural network (MLP) could have many hidden layers, and the output could be a vector.



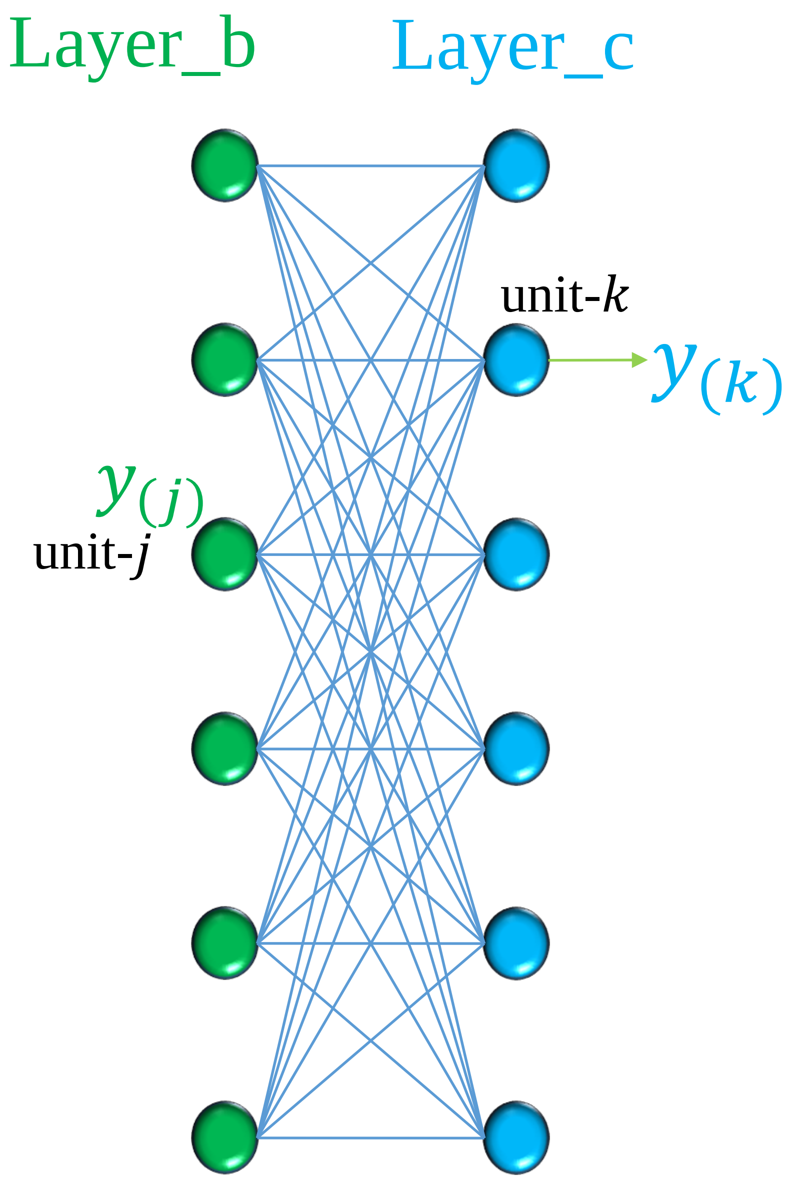
- w_{ij} is the weight of the link connecting unit-i in layer_a and unit-j in layer_b
- $x_{(i)}$ is the output of the unit-i in layer_a
- $y_{(j)}$ is the output of the unit-j in layer_b
- f_j is the activation function of the unit-j
- $y_{(j)} = f_j(v_j)$ and $v_j = \sum_i w_{ij} x_{(i)} + b_j$
- L is the loss

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial y_{(j)}} \frac{\partial y_{(j)}}{\partial w_{ij}}$$

$$\frac{\partial y_{(j)}}{\partial w_{ij}} = \frac{\partial f_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ij}}$$

How to get $\frac{\partial L}{\partial y_{(j)}}$?

$$\frac{\partial v_j}{\partial w_{ij}} = x_{(i)}$$



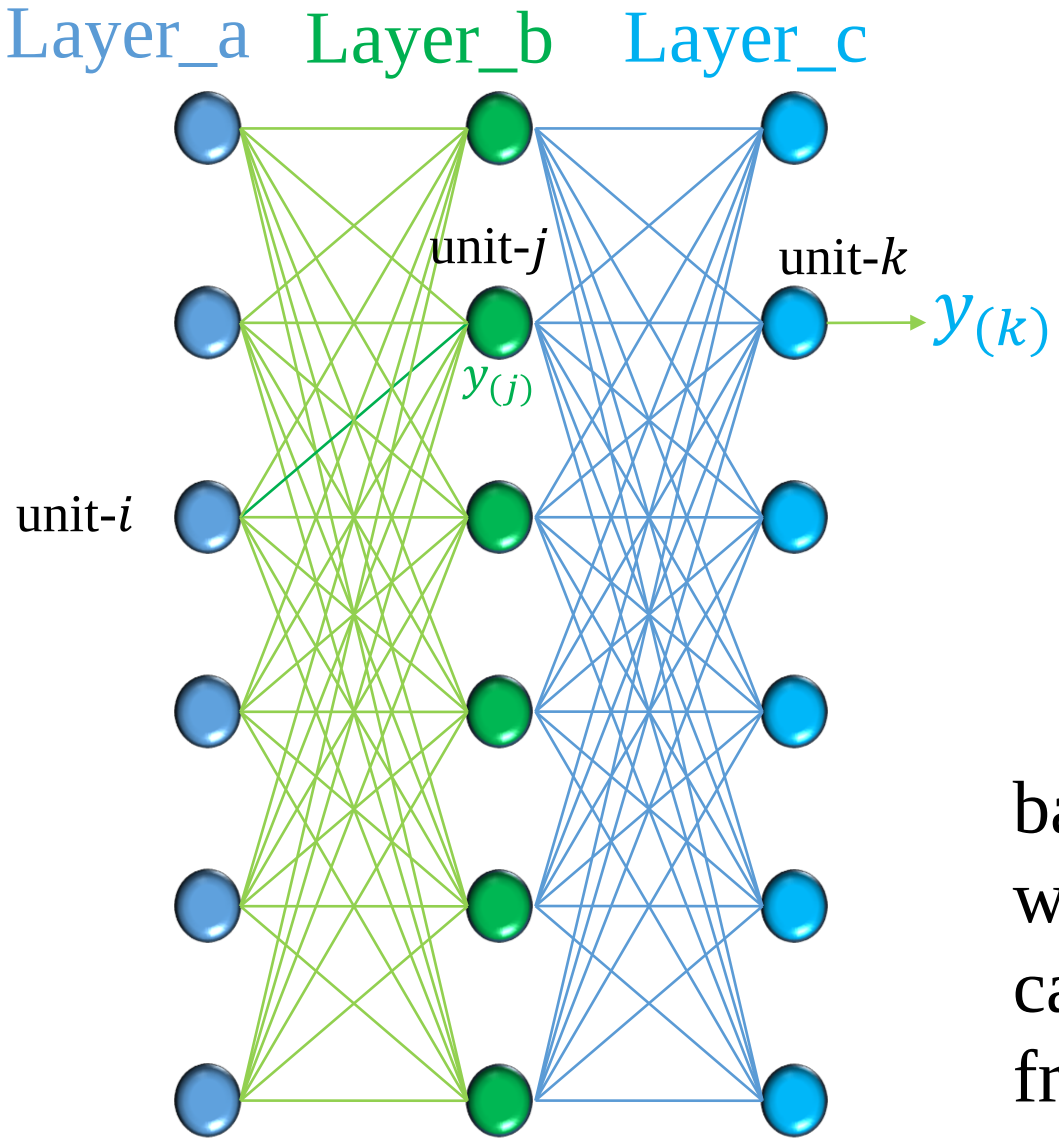
- w_{jk} is the weight of the link connecting unit-j in layer_b and unit-k in layer_c
- $y_{(j)}$ is the output of the unit-j in layer_b
- $y_{(k)}$ is the output of the unit-k in layer_c
- f_k is the activation function of the unit-k
- $y_{(k)} = f_k(u_k)$ and $u_k = \sum_j w_{jk} y_{(j)} + b_k$

$$\frac{\partial L}{\partial y_{(j)}} = \sum_{k=1}^{K} \frac{\partial L}{\partial y_{(k)}} \frac{\partial y_{(k)}}{\partial y_{(j)}}$$

$$\frac{\partial y_{(k)}}{\partial y_{(j)}} = \frac{\partial f_k}{\partial u_k} \frac{\partial u_k}{\partial y_{(j)}}$$

To get $\frac{\partial L}{\partial y_{(k)}}$ we check the next layer_d

$$\frac{\partial u_k}{\partial y_{(j)}} = w_{jk}$$

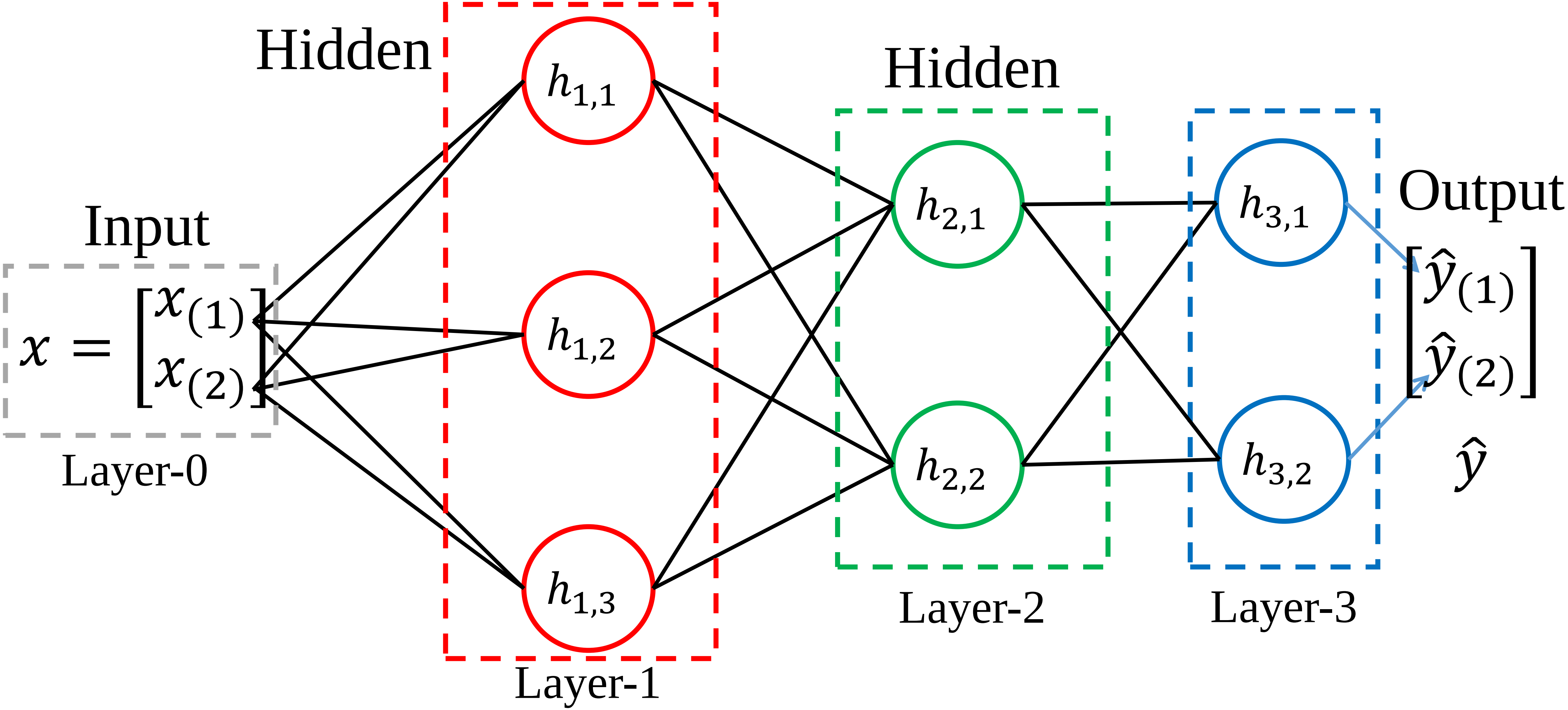


back- propagation:

we can write a program to calculate the derivatives, starting from the last layer

$$\frac{\partial L}{\partial w_{ij}} \Leftarrow \frac{\partial L}{\partial y_{(j)}} \Leftarrow \frac{\partial L}{\partial y_{(k)}} \text{ (every k)}$$

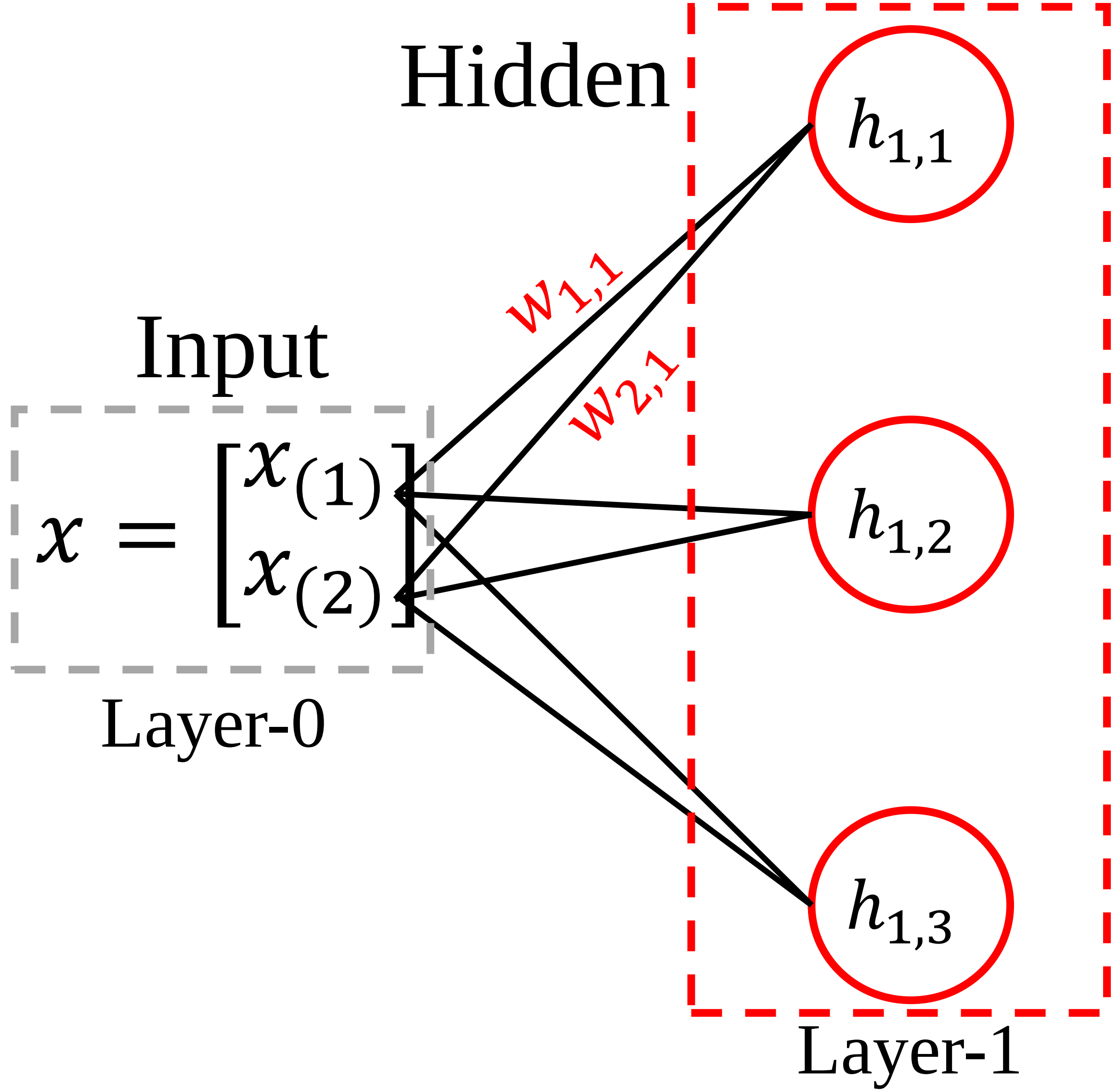
The "Modern" View of A Deep Neural Network
- A Computational Graph



$w_{i,j}$ is the weight of the link connecting unit-i in layer-0 and unit-j in layer-1

$w_{i,j}$ is the weight of the link connecting unit-i in layer-1 and unit-j in layer-2

$w_{i,j}$ is the weight of the link connecting unit-i in layer-2 and unit-j in layer-3



$h_{1,1}$ is the output of the first unit
$f_{1,1}$ is the activation function
b_1 is the bias

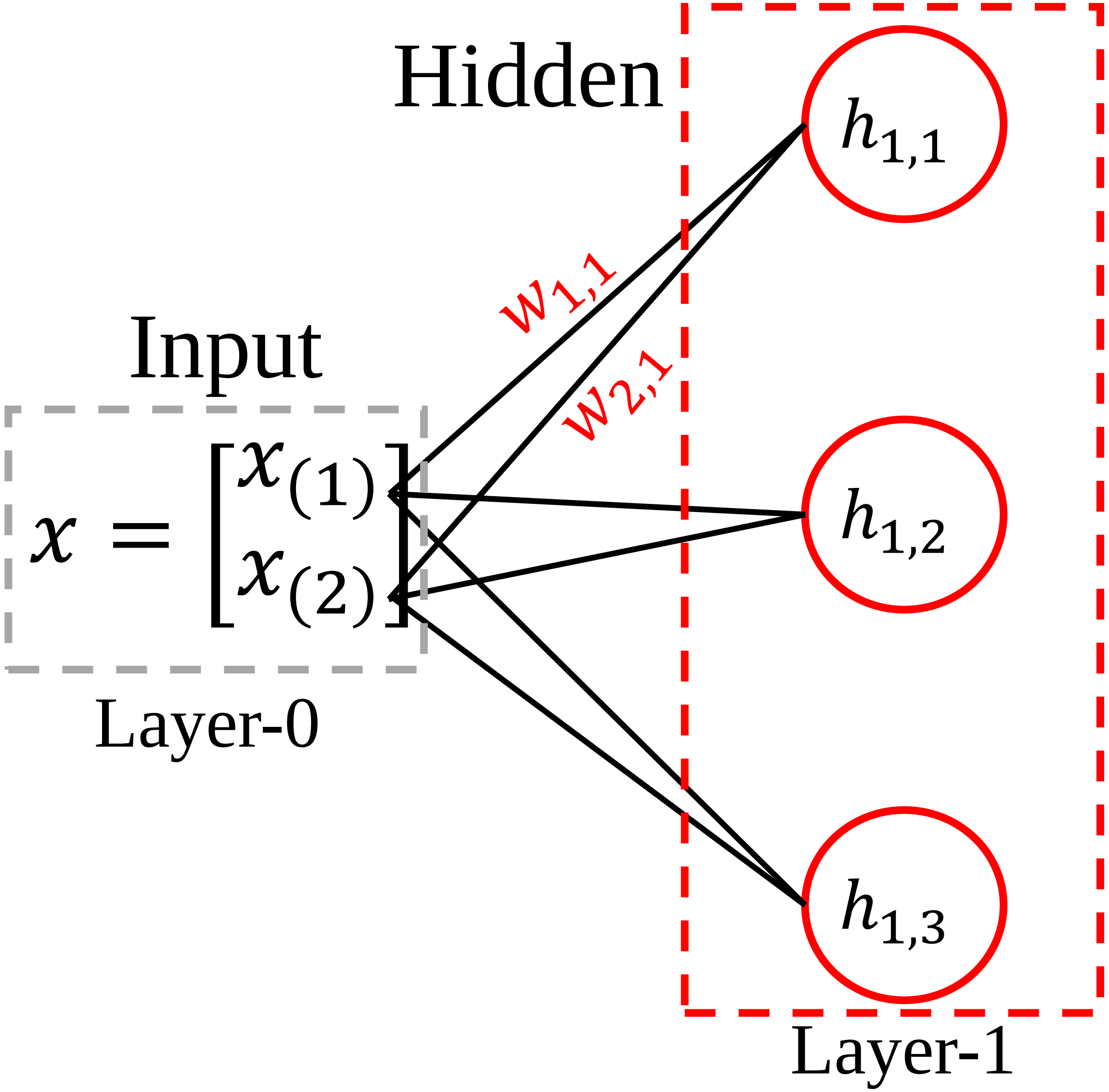
$$h_{1,1} = f_{1,1}(w_{1,1}x_{(1)} + w_{2,1}x_{(2)} + b_1)$$

$h_{1,2}$ is the output of the second unit
$f_{1,2}$ is the activation function
b_2 is the bias

$$h_{1,2} = f_{1,2}(w_{1,2}x_{(1)} + w_{2,2}x_{(2)} + b_2)$$

$h_{1,3}$ is the output of the third unit
$f_{1,3}$ is the activation function
b_3 is the bias

$$h_{1,3} = f_{1,3}(w_{1,3}x_{(1)} + w_{2,3}x_{(2)} + b_3)$$



$$h_{1,1} = f_{1,1}(w_{1,1}x_{(1)} + w_{2,1}x_{(2)} + b_1)$$
$$h_{1,2} = f_{1,2}(w_{1,2}x_{(1)} + w_{2,2}x_{(2)} + b_2)$$
$$h_{1,3} = f_{1,3}(w_{1,3}x_{(1)} + w_{2,3}x_{(2)} + b_3)$$

usually, $f_{1,1} = f_{1,2} = f_{1,3}$

Let's use matrix notation, define:

$$h_1 = \begin{bmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \end{bmatrix}, \quad f_1 = \begin{bmatrix} f_{1,1} \\ f_{1,2} \\ f_{1,3} \end{bmatrix}, \ b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

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

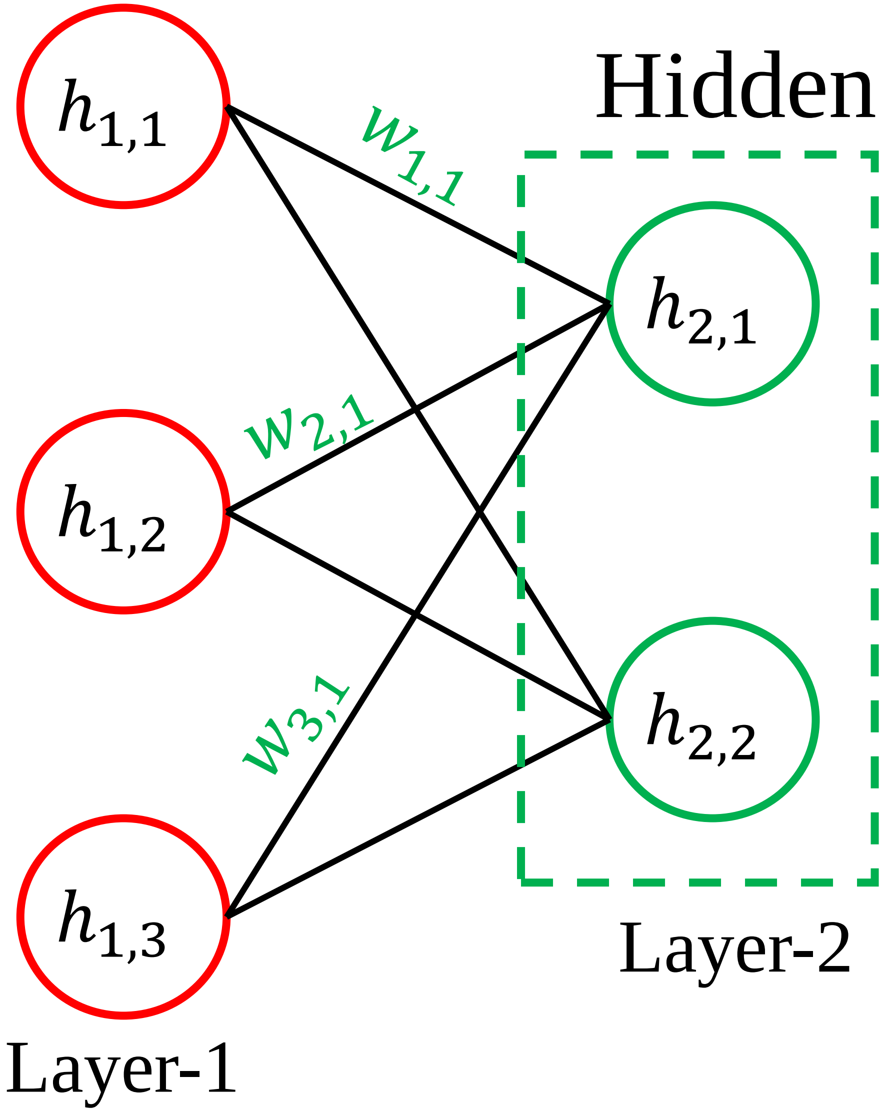
then:

$$h_1 = f_1(W^T x + b)$$

h_1 is the output of layer-1

W is the weight matrix of layer-1

b is the bias vector of layer-1



$$h_{2,1} = f_{2,1}\left(w_{1,1}h_{1,1} + w_{2,1}h_{1,2} + w_{3,1}h_{1,3} + b_1\right)$$
$$h_{2,2} = f_{2,2}\left(w_{1,2}h_{1,1} + w_{2,2}h_{1,2} + w_{3,2}h_{1,3} + b_2\right)$$

usually, $f_{2,1} = f_{2,2}$

Let's use matrix notation, define:

$$h_2 = \begin{bmatrix} h_{2,1} \\ h_{2,2} \end{bmatrix}, \quad f_2 = \begin{bmatrix} f_{2,1} \\ f_{2,2} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

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

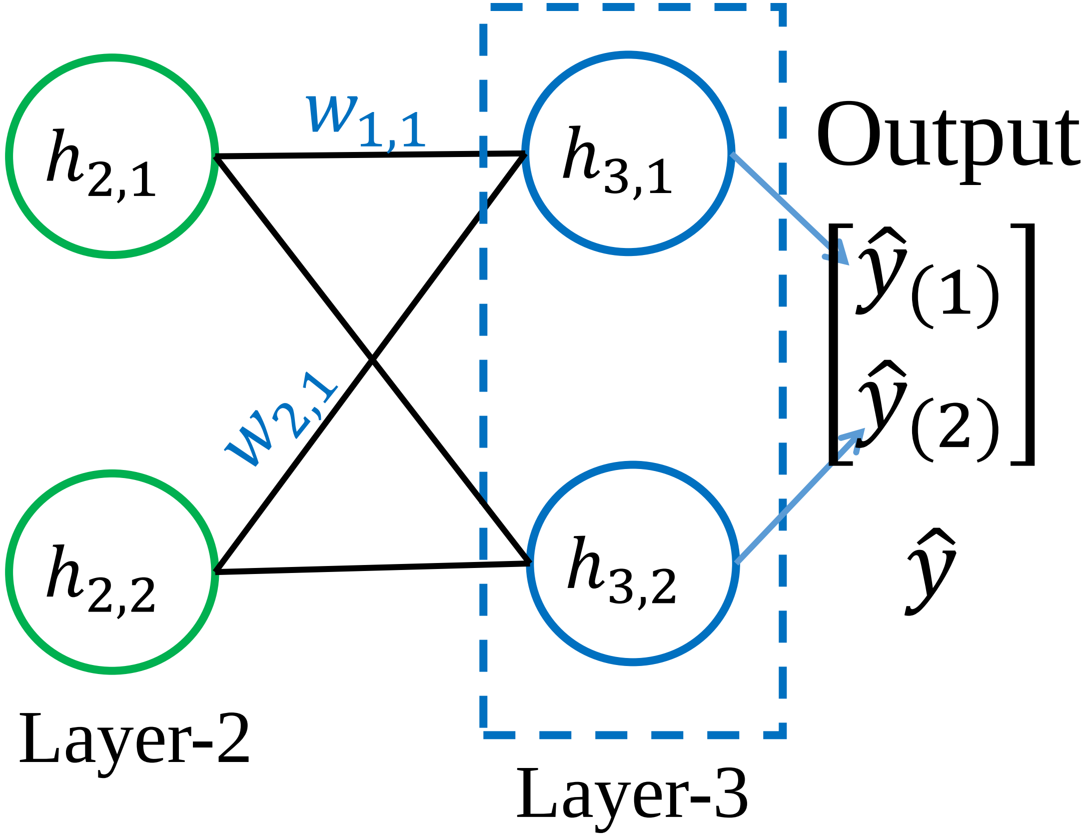
then:

$$h_2 = f_2(W^T h_1 + b)$$

h_2 is the output of layer-2

W is the weight matrix of layer-2

b is the bias vector of layer-2



$$\hat{y}_{(1)} = h_{3,1} = f_{3,1}(w_{1,1}h_{2,1} + w_{2,1}h_{2,2} + b_1)$$
$$\hat{y}_{(2)} = h_{3,2} = f_{3,2}(w_{1,2}h_{2,1} + w_{2,2}h_{2,2} + b_2)$$

usually, $f_{3,1} = f_{3,2}$

Let's use matrix notation, define:

$$h_3 = \begin{bmatrix} h_{3,1} \\ h_{3,2} \end{bmatrix}, \quad f_3 = \begin{bmatrix} f_{3,1} \\ f_{3,2} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

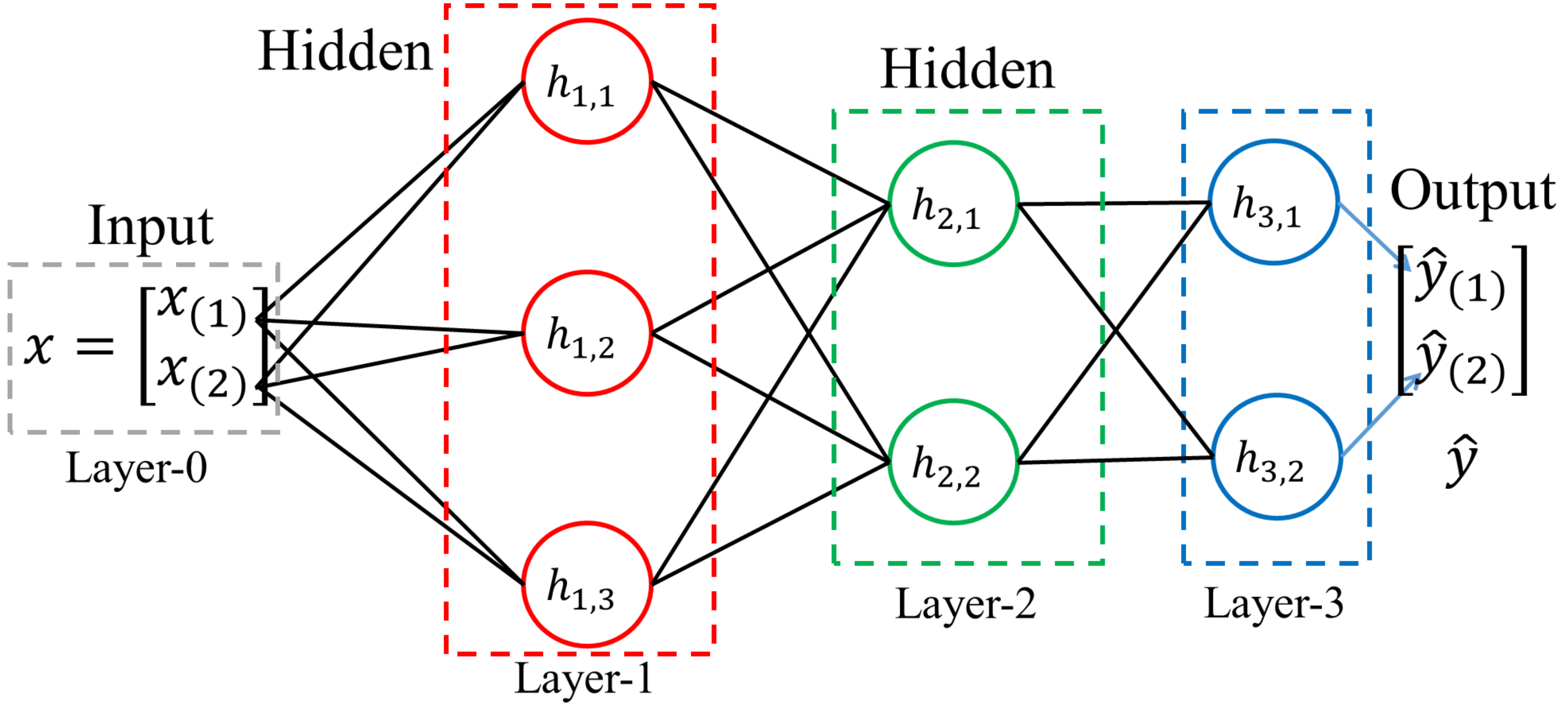
then:

$$\hat{y} = h_3 = f_3(W^T h_2 + b)$$

h_3 is the output of layer-3

W is the weight matrix of layer-3

b is the bias vector of layer-3



Instead of using circles connected by arrows, we can use a computational graph to represent a neural network:

$$x \rightarrow \boxed{h_1 = f_1(W^T x + b)} \rightarrow \boxed{h_2 = f_2(W^T h_1 + b)} \rightarrow \boxed{\hat{y} = h_3 = f_3(W^T h_2 + b)}$$

Input | Hidden Layer-1 | Hidden Layer-2 | (Output) Layer-3

Loss Function

- Regression (MSE, MAE, MAPE, robust regression loss, etc)
  - The output layer usually does not have nonlinear activation functions.
  - You may use softplus if the output should be nonnegative
- Binary Classification (BCE: binary cross entropy)
  - The output layer only has one unit
  - The output unit has a sigmoid activation function
- Multi-class Classification (CE: cross entropy)
  - The output layer has K output units for K classes
  - The output layer has a softmax function to convert 'raw' outputs to probability/confidence distribution across the K classes

Derivative of binary cross entropy loss when the last layer has sigmoid activation function

- A set of training data points $\{(x_n, y_n), n = 1, \ldots, N\}$, true label $y_n = 1 \; or \; 0$

- $\hat{y}_n$ is the output from the neural network, given the input x_n

- BCE loss $L = -\frac{1}{N}\sum_{n=1}^{N}\left(y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n)\right)$

- The derivative $\frac{\partial L}{\partial z}$ is needed to compute $\frac{\partial L}{\partial w}$ during backpropagation

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

Derivative of binary cross entropy loss when the last layer has a sigmoid activation function

- The loss for a single data point (x, y), $y = 1\ or\ 0$

$$L = -\left(ylog(\hat{y}) + (1 - y)log(1 - \hat{y})\right),\ \hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$$

- Compute $\frac{\partial L}{\partial z}$ using the chain rule

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

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

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

- Then we get $\frac{\partial L}{\partial z} = \sigma - y$

BCE loss in PyTorch

https://pytorch.org/docs/stable/nn.html#torch.nn.BCELoss

BCELoss $L(\hat{y}_n, y_n)$

CLASS `torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

The loss can be described as:

$$= L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \underbrace{x_n}_{\boxed{\hat{y}_n}} + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If reduce is `True`, then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if size\_average} = \text{True}, \\ \operatorname{sum}(L), & \text{if size\_average} = \text{False}. \end{cases}$$

BCEWithLogitsLoss $L(z_n, y_n)$ and $\hat{y}_n = \sigma(z_n) = \frac{1}{1+e^{-z_n}}$

CLASS `torch.nn.BCEWithLogitsLoss(weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None)` [SOURCE]

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The loss can be described as:

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

$$= \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If reduce is `True`, then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if size\_average} = \text{True}, \\ \text{sum}(L), & \text{if size\_average} = \text{False}. \end{cases}$$

Two loss functions: BCELoss and BCEWithLogitsLoss

$$L = -\left(y log(\hat{y}) + (1-y) log(1-\hat{y})\right)$$

- BCEWithLogitsLoss: the derivative $\frac{\partial L}{\partial z}$ is directly computed by

$$\frac{\partial L}{\partial z} = \sigma - y$$

- BCELoss: chain rule will be applied to compute $\frac{\partial L}{\partial z}$

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial z} = \frac{\sigma - y}{\sigma(1-\sigma)} \times \frac{\partial \sigma}{\partial z}$$

if σ is close to 1 or 0, the result is not numerically stable/accurate

https://keras.io/backend/#binary_crossentropy

binary_crossentropy

$$L(y_n, \hat{y}_n)$$

```
keras.losses.binary_crossentropy(y_true, y_pred, from_logits=False, label_smoothing=0)
```

binary_crossentropy

$$L(y_n, z_n) \text{ and } \hat{y}_n = \sigma(z_n) = \frac{1}{1 + e^{-z_n}}$$

```
keras.losses.binary_crossentropy(y_true, y_pred, from_logits=True, label_smoothing=0)
```

label_smoothing: smooth the ground-truth label
if $y_n = 0$ => $y_n = 0.1$; if $y_n = 1$ => $y_n = 0.9$
read the paper: https://arxiv.org/pdf/1906.02629.pdf

The "Modern" View of A Deep Neural Network - A Computational Graph

- To understand DNN and its implementation, it is necessary to understand the concepts of Computational Graph and Automatic Differentiation.
- The following slides are math-heavy (linear algebra and calculus).
- If you want to do research in machine learning (e.g. developing new algorithms), then you need to understand every equation in the following slides.
- If you only use machine learning methods to make applications, then you need to get a rough understanding of the concepts.

A computation process

$$\boxed{\begin{aligned} h &= 2x \\ y &= 3h \end{aligned}}$$

The computation graph

$$x \rightarrow \boxed{h = 2x} \rightarrow \boxed{y = 3h} \rightarrow y$$

It has two computing nodes

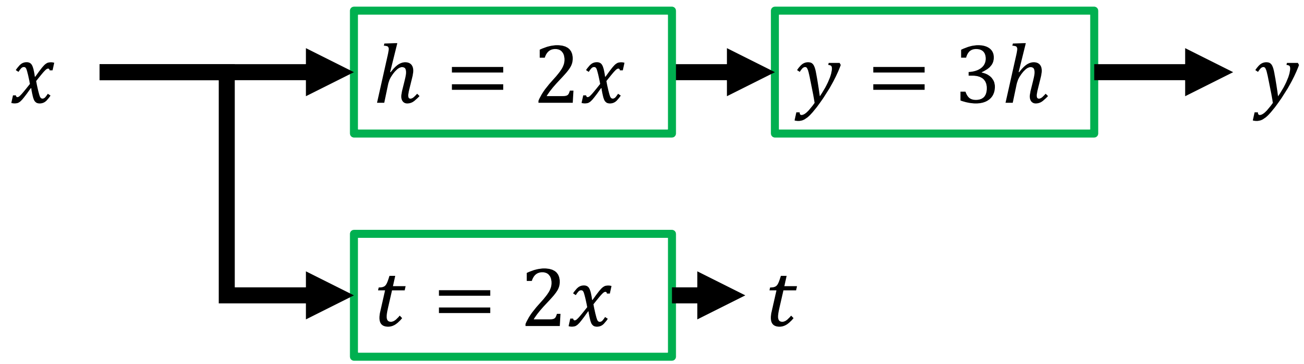
We can compute the derivative:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial h}\frac{\partial h}{\partial x} = 3 \times 2 = 6$$

A computation process

$$t = 2x$$
$$h = 2x$$
$$y = 3h$$

The computation graph (3 nodes)



In 'pure' Math:

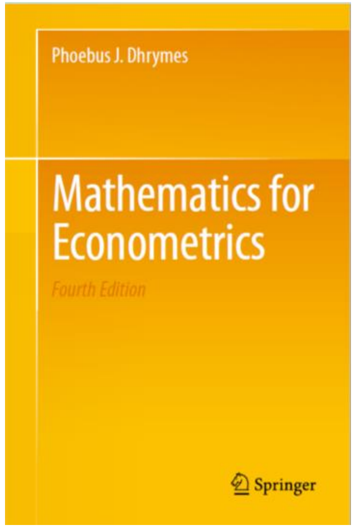
$$y = 3t$$

Thus:

$$\frac{\partial y}{\partial t} = 3$$

From the computation graph,
y is not a function of t, thus
$\frac{\partial y}{\partial t}$ is not defined: it does not exist.
$\frac{\partial y}{\partial t}$ can bet set to None or 0.

Review on Vector and Matrix Differentiation



- $y = f(x)$ where $x \in R^N$ and $y \in R^M$
- x and y are column vectors

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_{(1)}}{\partial x_{(1)}} & \frac{\partial y_{(1)}}{\partial x_{(2)}} & \cdots & \frac{\partial y_{(1)}}{\partial x_{(N)}} \\ \frac{\partial y_{(2)}}{\partial x_{(1)}} & \frac{\partial y_{(2)}}{\partial x_{(2)}} & \cdots & \frac{\partial y_{(2)}}{\partial x_{(N)}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y_{(M)}}{\partial x_{(1)}} & \frac{\partial y_{(M)}}{\partial x_{(2)}} & \cdots & \frac{\partial y_{(M)}}{\partial x_{(N)}} \end{bmatrix} \text{ a } M\text{-by-}N \text{ Matrix}$$

- The element of $\frac{\partial y}{\partial x}$ in i-th row and j-th col is $\left[\frac{\partial y}{\partial x}\right]_{i,j} = \frac{\partial y_{(i)}}{\partial x_{(j)}}$

Review on Vector and Matrix Differentiation

- $y = f(x)$ where $x \in R^N$ and $y \in R^M$

- The element of $\frac{\partial y}{\partial x}$ in i-th row and j-th col is $\left[\frac{\partial y}{\partial x}\right]_{i,j} = \frac{\partial y_{(i)}}{\partial x_{(j)}}$

- If y is a scalar (i.e. $M = 1$), then $\frac{\partial y}{\partial x}$ is a row vector

$$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x_{(1)}} \quad \frac{\partial y}{\partial x_{(2)}} \quad \cdots \quad \frac{\partial y}{\partial x_{(N)}}\right]$$

Review on Vector and Matrix Differentiation

- y is a scalar (i.e. $M = 1$), then $\frac{\partial y}{\partial x}$ is a row vector

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_{(1)}} & \frac{\partial y}{\partial x_{(2)}} & \cdots & \frac{\partial y}{\partial x_{(N)}} \end{bmatrix}$$

- Notation to get a column vector

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

Derivatives of Functions

- $y = Ax$ where $x \in R^N$ and $y \in R^M$
- $A \in R^{M \times N}$
- A does not depend on x
- $\frac{\partial y}{\partial x} = A$
- If x is a function of a vector z, then we have the chain rule:

$$\frac{\partial y}{\partial z} = \frac{\partial y}{\partial x}\frac{\partial x}{\partial z} = A\frac{\partial x}{\partial z}$$

Derivatives of Functions

- $\alpha = y^T A x$ where $x \in R^N$ and $y \in R^M$
- α is a scalar, A is independent of x and y
- $\frac{\partial \alpha}{\partial x} = y^T A$
- $\frac{\partial \alpha}{\partial y} = \frac{\partial}{\partial y}(y^T A x) = \frac{\partial}{\partial y}(x^T A^T y) = x^T A^T$

Derivatives of Functions

- $\alpha = x^T A x$ where $x \in R^N$ α is a scalar, A is independent of x, and a_{ij} is an element of A
- $\frac{\partial \alpha}{\partial x} = x^T(A^T + A)$, a row vector
- proof:

$$\alpha = \sum_j \sum_i a_{ij} x_i x_j$$

$$\frac{\partial \alpha}{\partial x_k} = \sum_j a_{kj} x_j + \sum_i a_{ik} x_i$$

$$\sum_j a_{kj} x_j = x^T \times row_k[A] = x^T \times col_k[A^T]$$

$$\sum_i a_{ik} x_i = x^T \times col_k[A]$$

Thus: $\frac{\partial \alpha}{\partial x} = x^T(A^T + A)$

Chain Rule

- $\alpha = y^T x$ where $x, y \in R^N$ and α is a scalar

  x and y are functions of a vector z , then

$$\frac{\partial \alpha}{\partial z} = \frac{\partial \alpha}{\partial y}\frac{\partial y}{\partial z} + \frac{\partial \alpha}{\partial x}\frac{\partial x}{\partial z} = x^T\frac{\partial y}{\partial z} + y^T\frac{\partial x}{\partial z}$$

- $\alpha = x^T x$ and x is function of z, then

$$\frac{\partial \alpha}{\partial z} = 2x^T\frac{\partial x}{\partial z}$$

Chain Rule

- $\alpha = y^T A x$ wherewhere $x \in R^N$, $y \in R^M$ and α is a scalar

  x and y are functions of a vector z , then

$$\frac{\partial \alpha}{\partial z} = \frac{\partial \alpha}{\partial y}\frac{\partial y}{\partial z} + \frac{\partial \alpha}{\partial x}\frac{\partial x}{\partial z} = x^T A^T \frac{\partial y}{\partial z} + y^T A \frac{\partial x}{\partial z}$$

- $\alpha = x^T A x$ wherewhere $x \in R^N$ and α is a scalar

  x is function of z , then

$$\frac{\partial \alpha}{\partial z} = x^T (A + A^T) \frac{\partial x}{\partial z}$$

Chain Rule: a note

- x is a function of z
- y is a function of z
- α is a function of x and y
- You may be familiar with this "total derivative":

$$\frac{d\alpha}{dz} = \frac{\partial \alpha}{\partial y}\frac{dy}{dz} + \frac{\partial \alpha}{\partial x}\frac{dx}{dz}$$

- $\alpha(z)$, $x(z)$ and $y(z)$ are single variable functions, thus we can write

$$\frac{\partial \alpha}{\partial z} = \frac{d\alpha}{dz} \qquad \frac{\partial x}{\partial z} = \frac{dx}{dz} \qquad \frac{\partial y}{\partial z} = \frac{dy}{dz}$$

- Example: $\alpha = x + y$, $x = z$ and $\mathrm{y} = 2z$, thus $\alpha = 3z$, $\frac{\partial \alpha}{\partial z} = \frac{d\alpha}{dz} = 3$

Second order derivative

- $y = f(x)$ where $x \in R^N$ and $y \in R^M$, then

$$\frac{\partial^2 y}{\partial x \partial x} = \frac{\partial}{\partial x} vec\left[\left(\frac{\partial y}{\partial x}\right)^T\right]$$

It is a matrix of size $MN \times N$

Vectorization of a Matrix: convert a matrix to a vector

- Matrix $A \in R^{M \times N}$
- column vectorization: to convert a matrix to a column vector

$$cvec(A) = \begin{bmatrix} col_1 \\ col_2 \\ \vdots \\ col_N \end{bmatrix}$$

- row vectorization: to convert a matrix to a row vector

$$rvec(A) = [row_1 \quad row_2 \quad \dots \quad row_M]$$

- $vec(A)$ denotes $cvec(A)$

Vectorization of a Matrix: convert a matrix to a vector

- $W = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}$
- column vectorization: to convert a matrix to a column vector

$$cvec(W) = \begin{bmatrix} w_{1,1} \\ w_{2,1} \\ w_{1,2} \\ w_{2,2} \\ w_{1,3} \\ w_{2,3} \end{bmatrix}$$

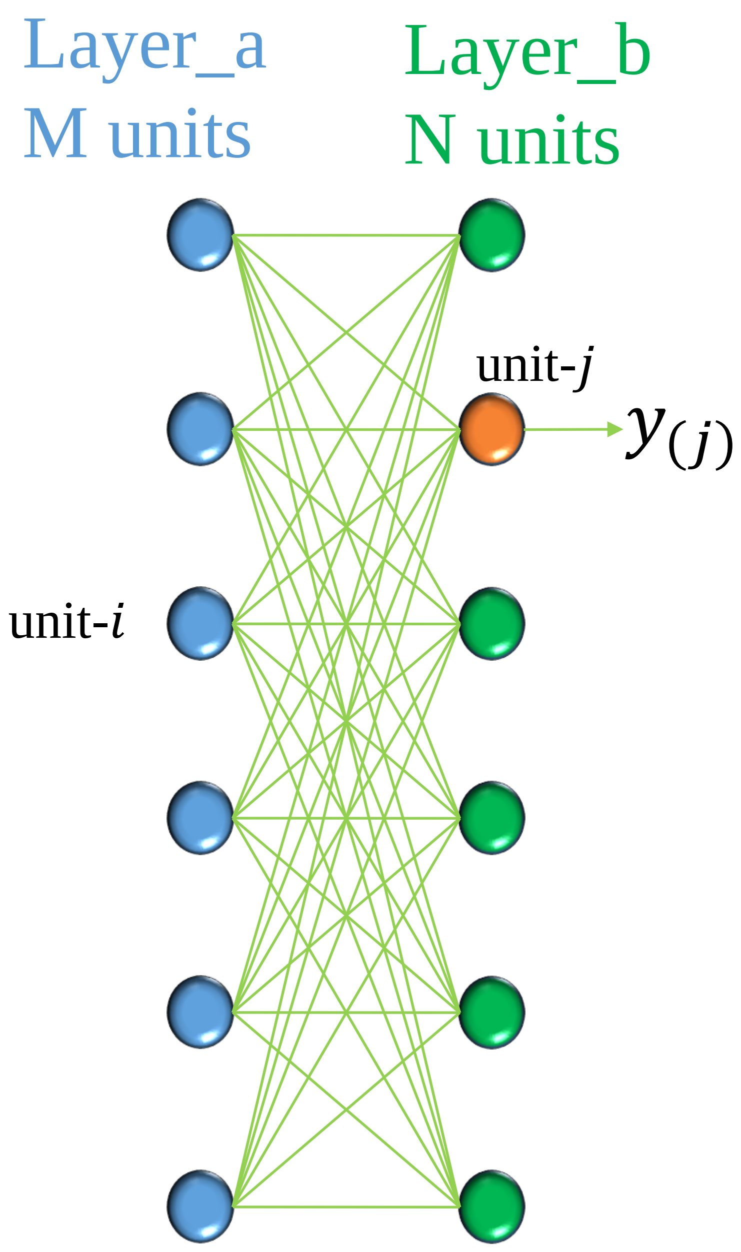
- row vectorization: to convert a matrix to a row vector

$$rvec(W) = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{2,1} & w_{2,2} & w_{2,3} \end{bmatrix}$$

- $vec(W)$ denotes $cvec(W)$

Vector-valued Function: the output is a vector

$$y = f(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_M(x) \end{bmatrix} \quad \text{where } x \in R^N \text{ and } y \in R^M$$



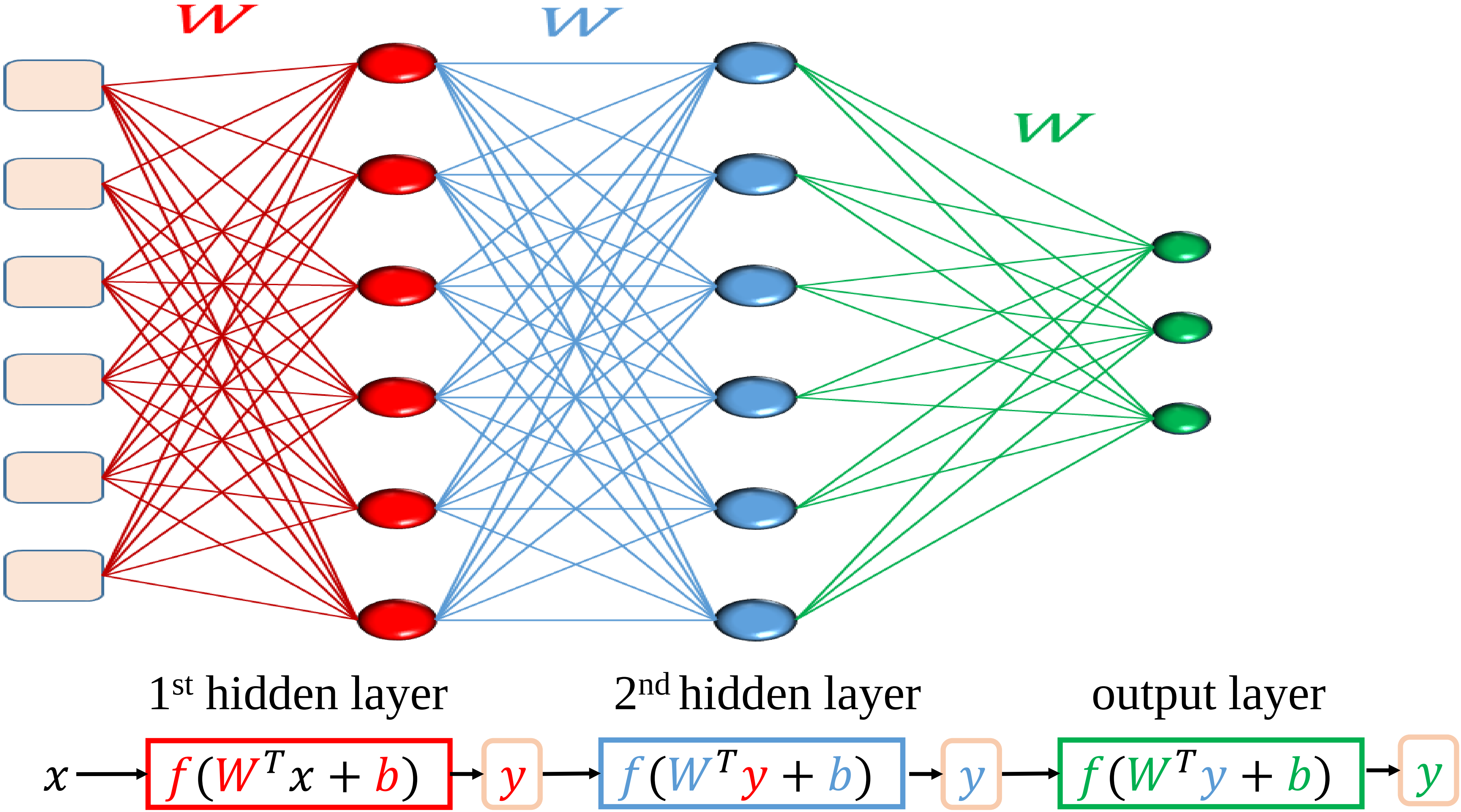
- w_{ij} is the weight of the link connecting unit-i in Layer_a and unit-j in Layer_b

- W is the weight matrix of Layer_b: $M \times N$

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & \boxed{w_{1j}} & \cdots & w_{1N} \\ w_{21} & w_{21} & \cdots & w_{2j} & \cdots & w_{21} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{M1} & w_{M2} & \cdots & w_{Mj} & \cdots & w_{MN} \end{bmatrix}$$

- Column-j of W belongs to unit-j
- $b = [b_1, b_2, \dots, b_j, \dots, b_N]^T$

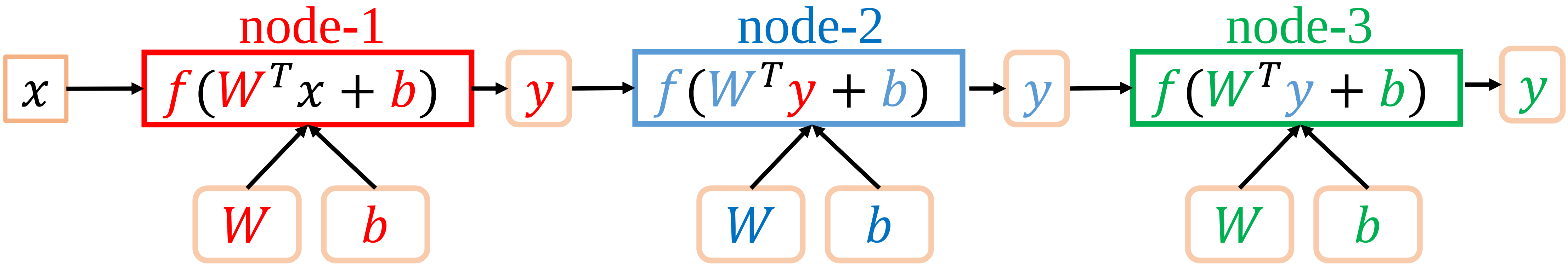
b_j is the bias of unit-j



W
W
W
1st hidden layer
2nd hidden layer
output layer
$x \rightarrow f(W^T x + b) \rightarrow y \rightarrow f(W^T y + b) \rightarrow y \rightarrow f(W^T y + b) \rightarrow y$

A neural network is a computational graph

- A layer is a computation node in the graph
- A computation node is a function
- A tensor (vector/matrix) is input or output of a function



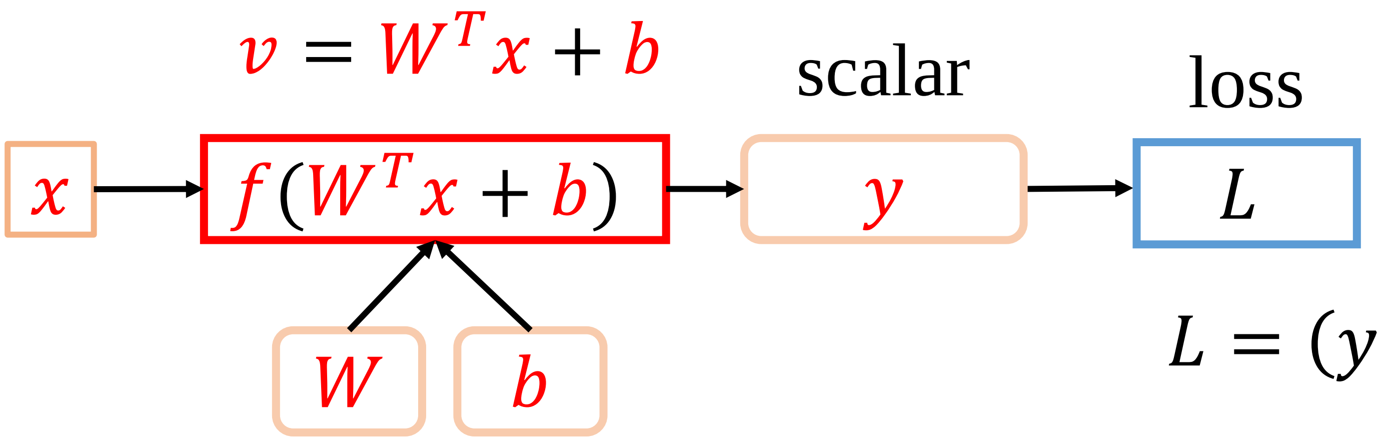
$\boxed{W}$ is a leaf of the graph (it is not the output of a function)

A computation node is a function - PyTorch

```
                 Functions in the Autograd Graph
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When viewing the autograd system as a graph, `Function`s are the vertices or
nodes, connected to each other via (directed) `Edge`s, which themselves are
represented via (`Function`, input_nr) pairs. `Variable`s are the outputs to
and inputs of `Function`s, and travel between these edges during execution
of the graph. When two or more `Edge`s (from different sources) point at the
same input to a `Function`, the values produced along all of these edges are
implicitly summed prior to being forwarded to the target `Function`.
```

https://github.com/pytorch/pytorch/blob/master/torch/csrc/autograd/function.h

Backpropagation in a graph

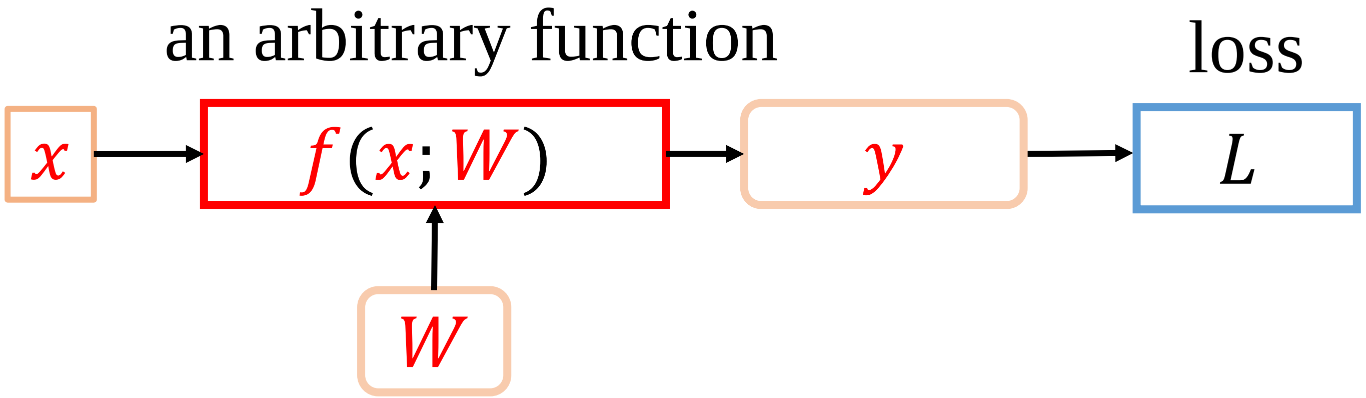


$L = (y - t)^2$

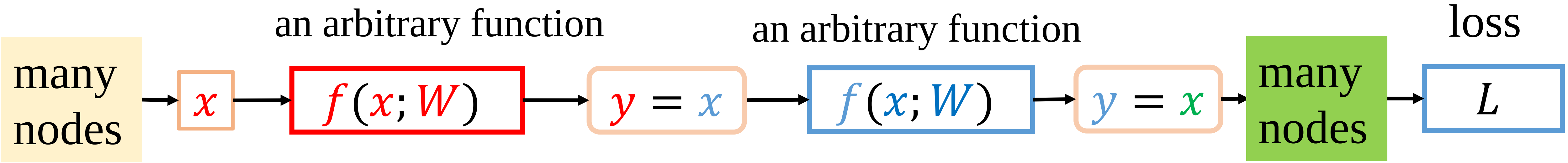
t is true label (target value)

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial v}\frac{\partial v}{\partial W} = 2(y - t)f'(v)x^T$$

Backpropagation in a graph



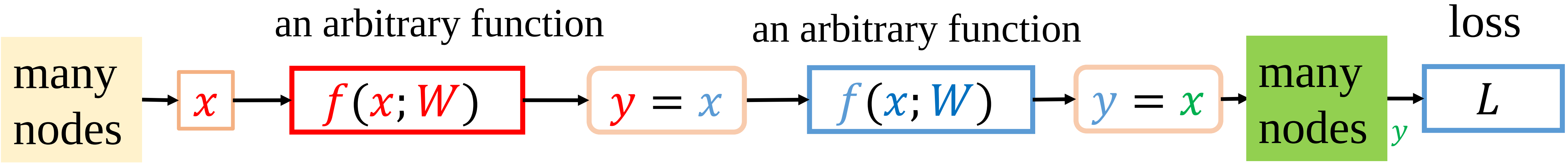
$$\frac{\partial L}{\partial vec(W)} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial vec(W)} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial vec(W)}$$



$$\frac{\partial L}{\partial vec(W)} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial vec(W)} = \frac{\partial L}{\partial x}\frac{\partial f}{\partial vec(W)}$$

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

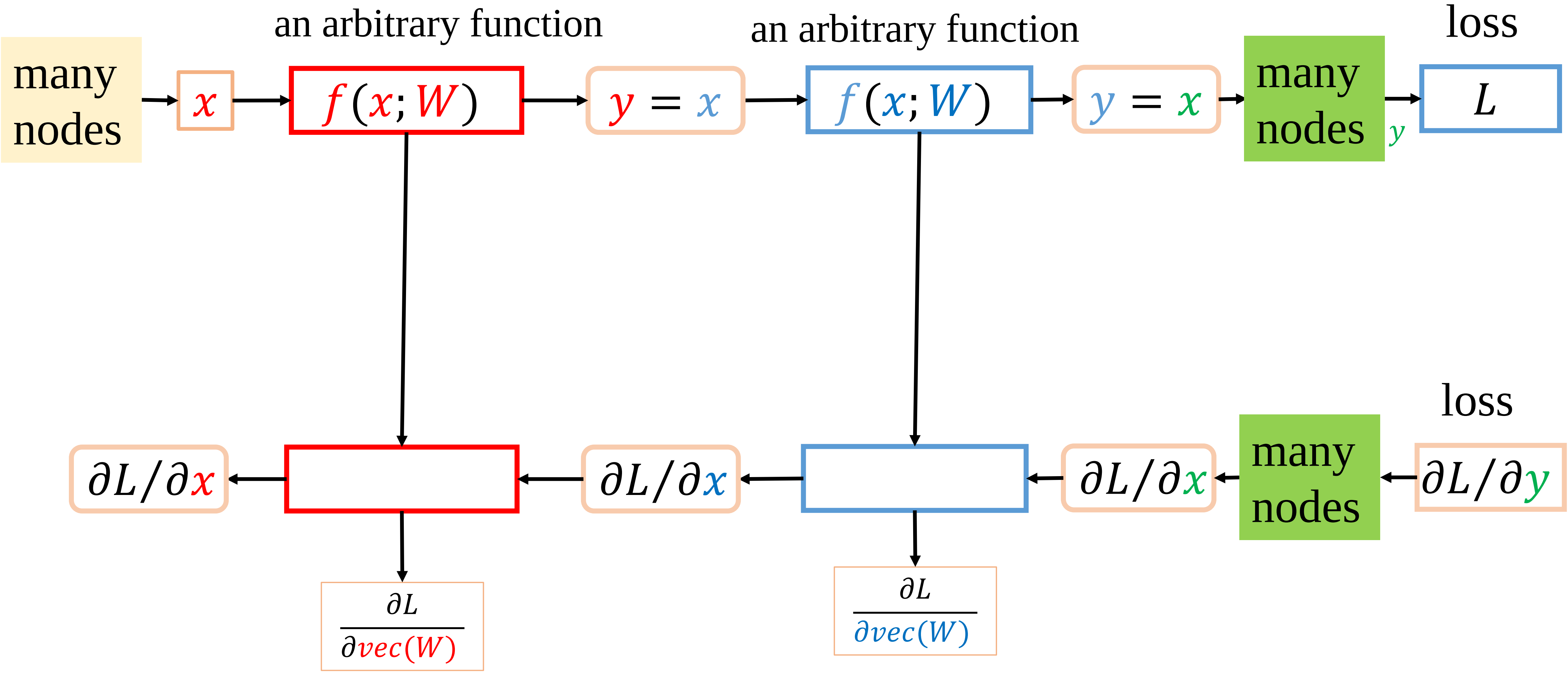
$$\left(\frac{\partial L}{\partial vec(W)}\right)^T = \left(\frac{\partial f}{\partial vec(W)}\right)^T \left(\frac{\partial L}{\partial x}\right)^T$$



Backpropagation:

$$\left(\frac{\partial L}{\partial vec(W)}\right)^T = \left(\frac{\partial f}{\partial vec(W)}\right)^T \left(\frac{\partial L}{\partial x}\right)^T$$

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



$$\boxed{\frac{\partial L}{\partial vec(W)} = \frac{\partial L}{\partial x}\frac{\partial f}{\partial vec(W)}, \frac{\partial L}{\partial x} = \frac{\partial L}{\partial x}\frac{\partial f}{\partial x}}$$

The computational graph for backpropagation

In general, x and y could be high-dimensional tensors.
We can apply the same analysis by vectorizing every tensor

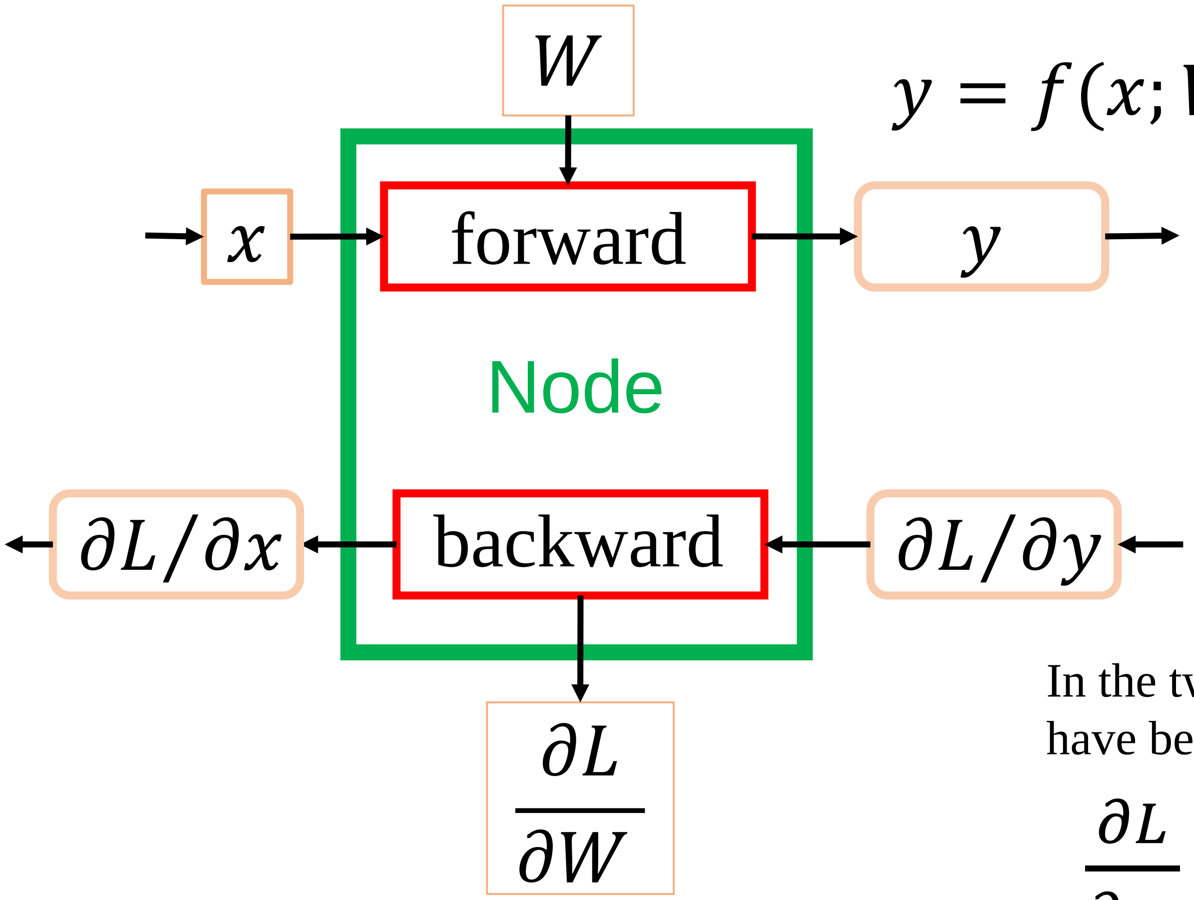
an arbitrary function

$$\rightarrow \boxed{x} \rightarrow \boxed{f(x;W)} \rightarrow \boxed{y} \rightarrow$$

$$\frac{\partial L}{\partial vec(W)} = \frac{\partial L}{\partial vec(y)} \times \frac{\partial vec(f)}{\partial vec(W)}$$

$$\frac{\partial L}{\partial vec(x)} = \frac{\partial L}{\partial vec(y)} \times \frac{\partial vec(f)}{\partial vec(x)}$$

Forward and Backward inside a Node (Function)



$$y = f(x; W)$$

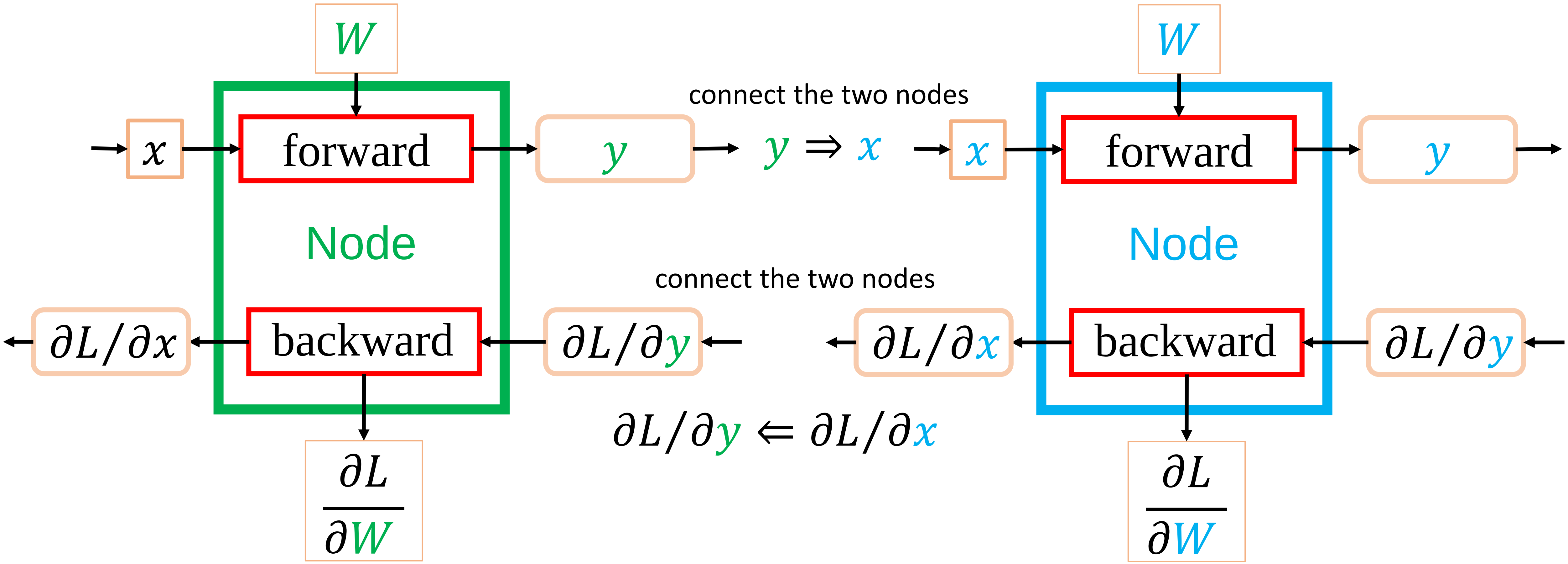
In the two equations, we assume x, y and w have been vectorized.

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

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial L}{\partial y}\frac{\partial f}{\partial W}$$

https://pytorch.org/docs/master/notes/extending.html

Forward and Backward of two connected nodes



https://pytorch.org/docs/master/notes/extending.html

PyTorch has implemented the Forward and Backward methods for many functions

- Every 'normal' function that you can think of, has been implemented.
- We can define a complex function using many simple functions
- We define a chain of functions as a layer of a network
- We need to design network structure and/or loss function
- Automatic differentiation: we do not need to calculate the derivatives (backward pass) by hand

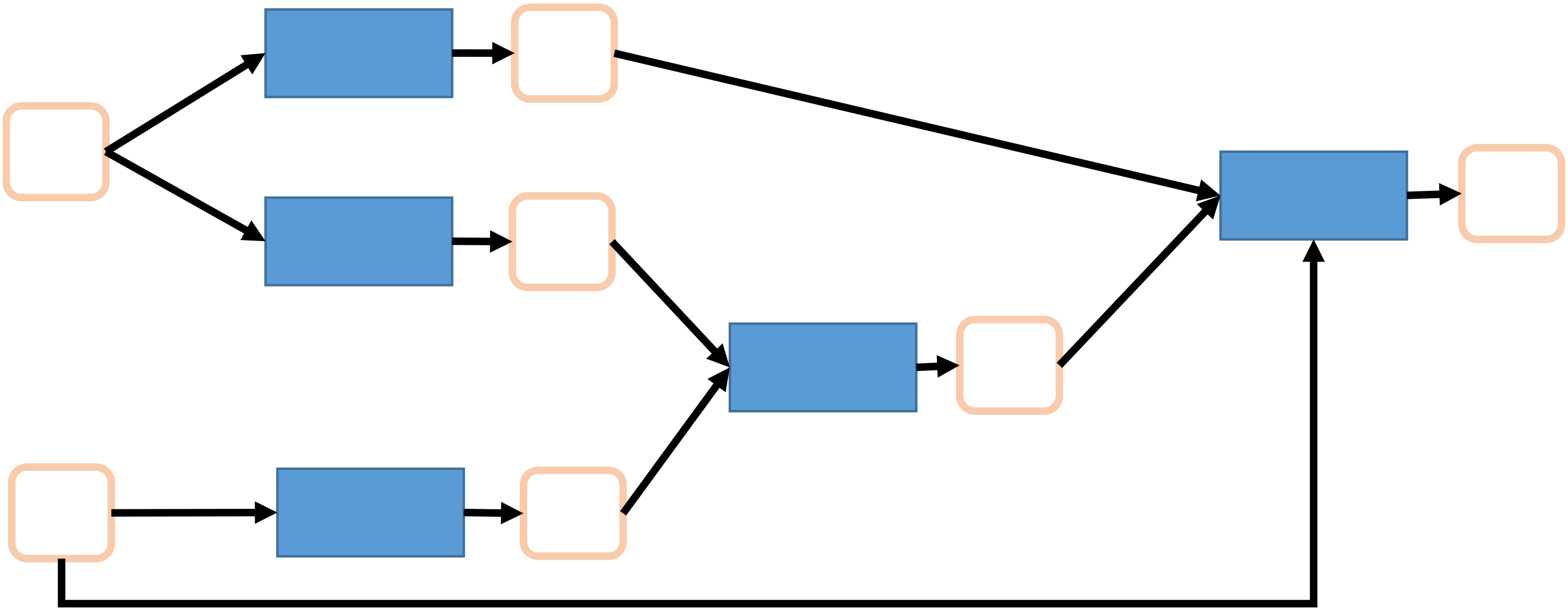
Tensorflow also has implemented the Forward and Backward methods for many functions

- Every 'normal' function that you can think of, has been implemented.
- We can define a complex function using many simple functions
- We define a chain of functions as a layer of a network
- Automatic differentiation
- Tensorflow API is NOT as user-friendly as Pytorch

Keras is built on top of Tensorflow

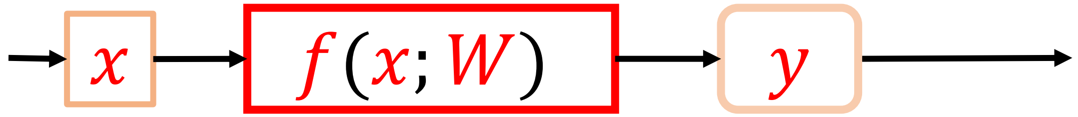
- Tensorflow API is NOT as user-friendly as Pytorch
- Keras is built on top of Tensorflow.
- We use Tensorflow via Keras
- It is very difficult to implement new algorithms/losses in Keras.
- If you want to use/design new algorithms, I recommend Pytorch.

A computational graph is a Directed Acyclic Graph (DAG)
directed: each edge has a direction
acyclic: no cycles



a computation node could be a neural network

a network



W denotes a list of parameters (tensors) of the network

In other words, we can build a network of networks

PyTorch can automatically calculate the derivatives

```
import torch
```

```
x1 = torch.rand((2,3), requires_grad=True)
x2 = torch.rand((2,3), requires_grad=True)
y = torch.sum(x1*x2)
```

```
x1
```

```
tensor([[0.1161, 0.1989, 0.2589],
        [0.3380, 0.1101, 0.2910]], requires_grad=True)
```

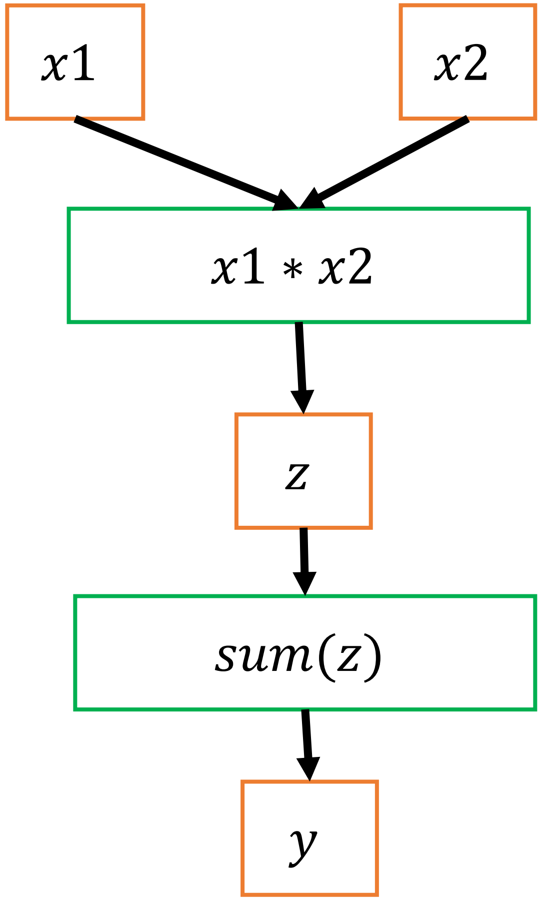
```
x2
```

```
tensor([[0.3296, 0.7152, 0.5111],
        [0.4479, 0.1195, 0.6649]], requires_grad=True)
```

```
y
```

```
tensor(0.6708, grad_fn=<SumBackward0>)
```

the graph with two computation nodes



PyTorch can automatically calculate the derivatives

y must be a scalar

```
1 y.backward()
```

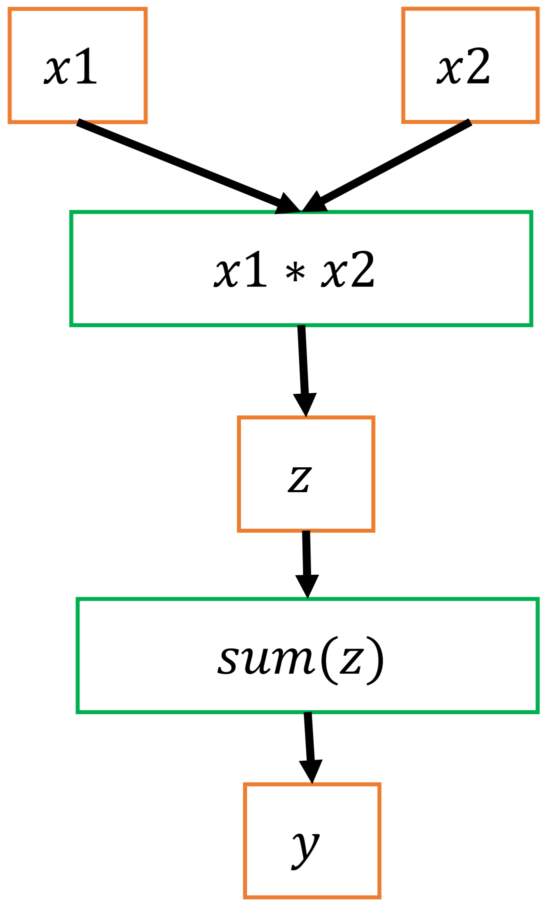
```
1 x1.grad # dy/dx1
```

```
tensor([[0.3296, 0.7152, 0.5111],
        [0.4479, 0.1195, 0.6649]])
```

```
1 x2.grad # dy/dx2
```

```
tensor([[0.1161, 0.1989, 0.2589],
        [0.3380, 0.1101, 0.2910]])
```

the graph with two computation nodes



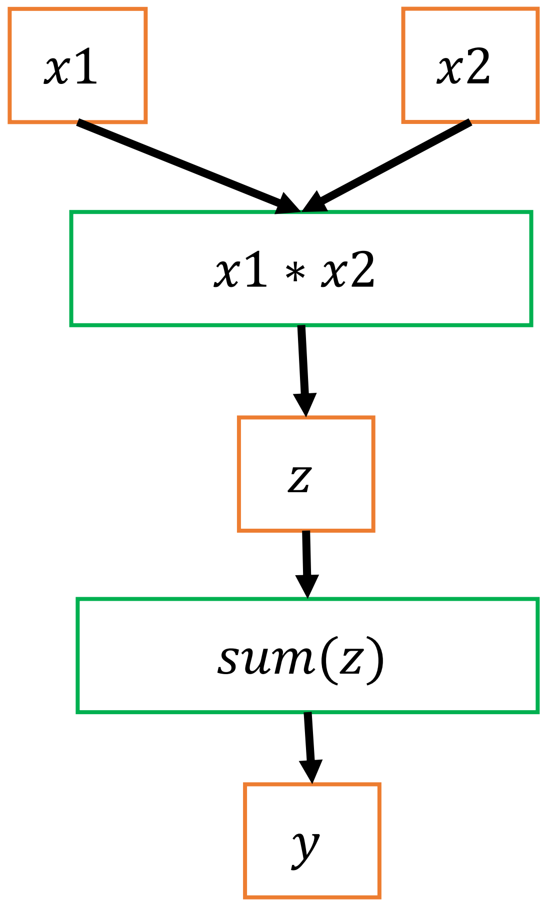
We can not run this line multiple times

```
y.backward()
```

set retain_graph=True
if you want to run this line multiple times

```
y.backward(retain_graph=True)
```

the graph with two computation nodes

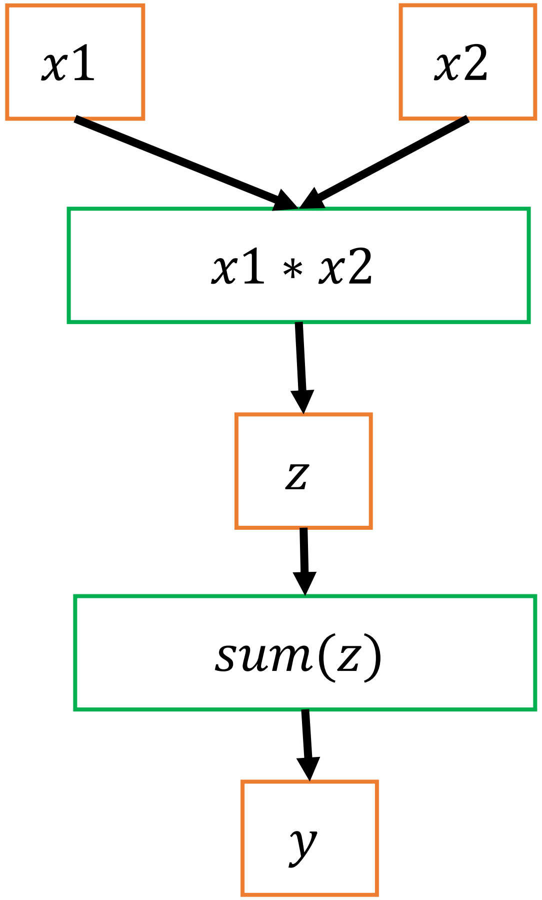


```python
import torch
```

```python
x1 = torch.rand((2,3), requires_grad=True)
x2 = torch.rand((2,3), requires_grad=True)
z = x1*x2
y = torch.sum(z)
```

```python
z.backward() # error
```

the graph with two computation nodes



```
1  dy_dz=torch.ones((2,3))
```

```
1  z.backward(gradient=dy_dz)
```

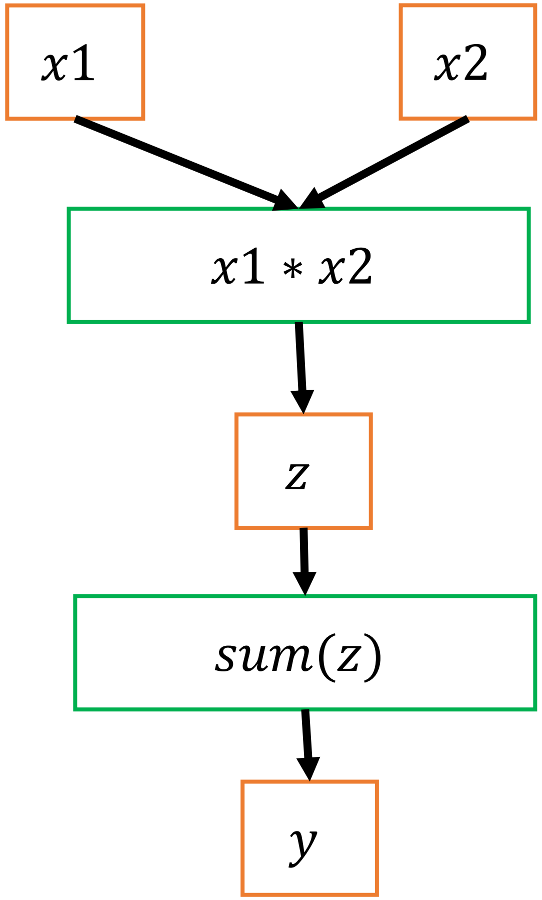
```
1  x2
```

```
tensor([[0.5142, 0.8137, 0.7438],
        [0.1581, 0.6234, 0.1528]], requires_grad=True)
```

```
1  x1.grad    # dy/dx1  or dz/dx1 ?
```

```
tensor([[0.5142, 0.8137, 0.7438],
        [0.1581, 0.6234, 0.1528]])
```

the graph with two computation nodes



$$\frac{dy}{dx_1} = \frac{dy}{dz}\frac{dz}{dx_1}$$

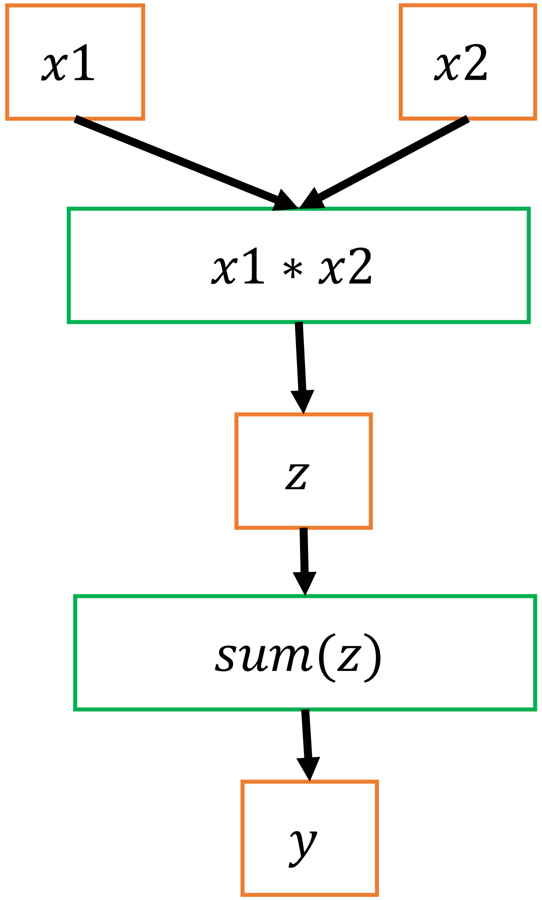
```
z.backward(gradient=dy_dz)
```

Then, we get $\frac{dy}{dx_1}$

```
x1.grad
```

```
tensor([[0.5142, 0.8137, 0.7438],
        [0.1581, 0.6234, 0.1528]])
```

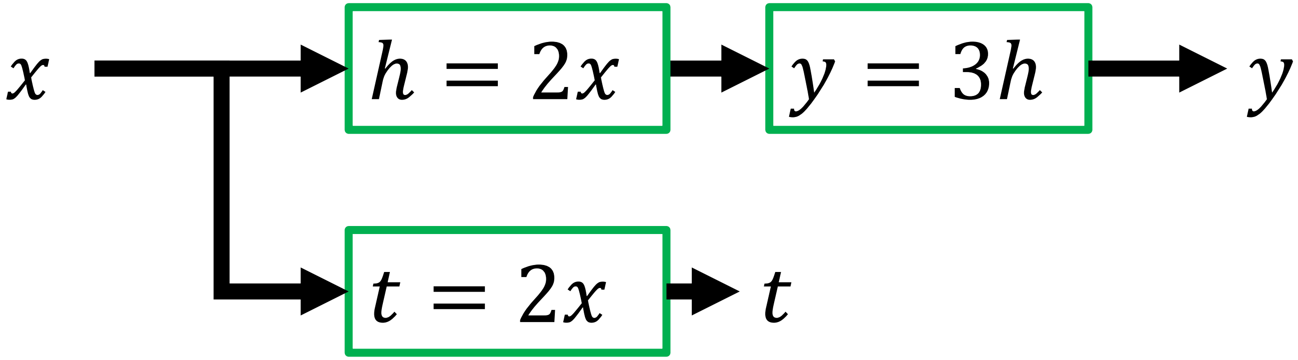
the graph with two computation nodes



A computation process

$$t = 2x$$
$$h = 2x$$
$$y = 3h$$

The computation graph (3 nodes)



In 'pure' Math:

$$y = 3t$$

Thus:

$$\frac{\partial y}{\partial t} = 3$$

From the computation graph,
y is not affected by t, thus
$\frac{\partial y}{\partial t}$ does not exist
$\frac{\partial y}{\partial t}$ is None in Pytorch

```python
import torch
```

```python
x = torch.rand((1,), requires_grad=True)
h=2*x
t=2*x
y=3*h
```

```python
dy_dx=torch.autograd.grad(y,x, retain_graph=True)
dy_dx
```

```
(tensor([6.]),)
```

```python
dy_dt=torch.autograd.grad(y,t, retain_graph=True, allow_unused=True)
dy_dt
```

```
(None,)
```