

Neural Networks I

Instructor: Siting Liu

School of Information Science and Technology (SIST), ShanghaiTech University

Shanghai Institute of Semiconductor Innovation (SISI)

<https://faculty.sist.shanghaitech.edu.cn/liust/EE219/EE219.html>



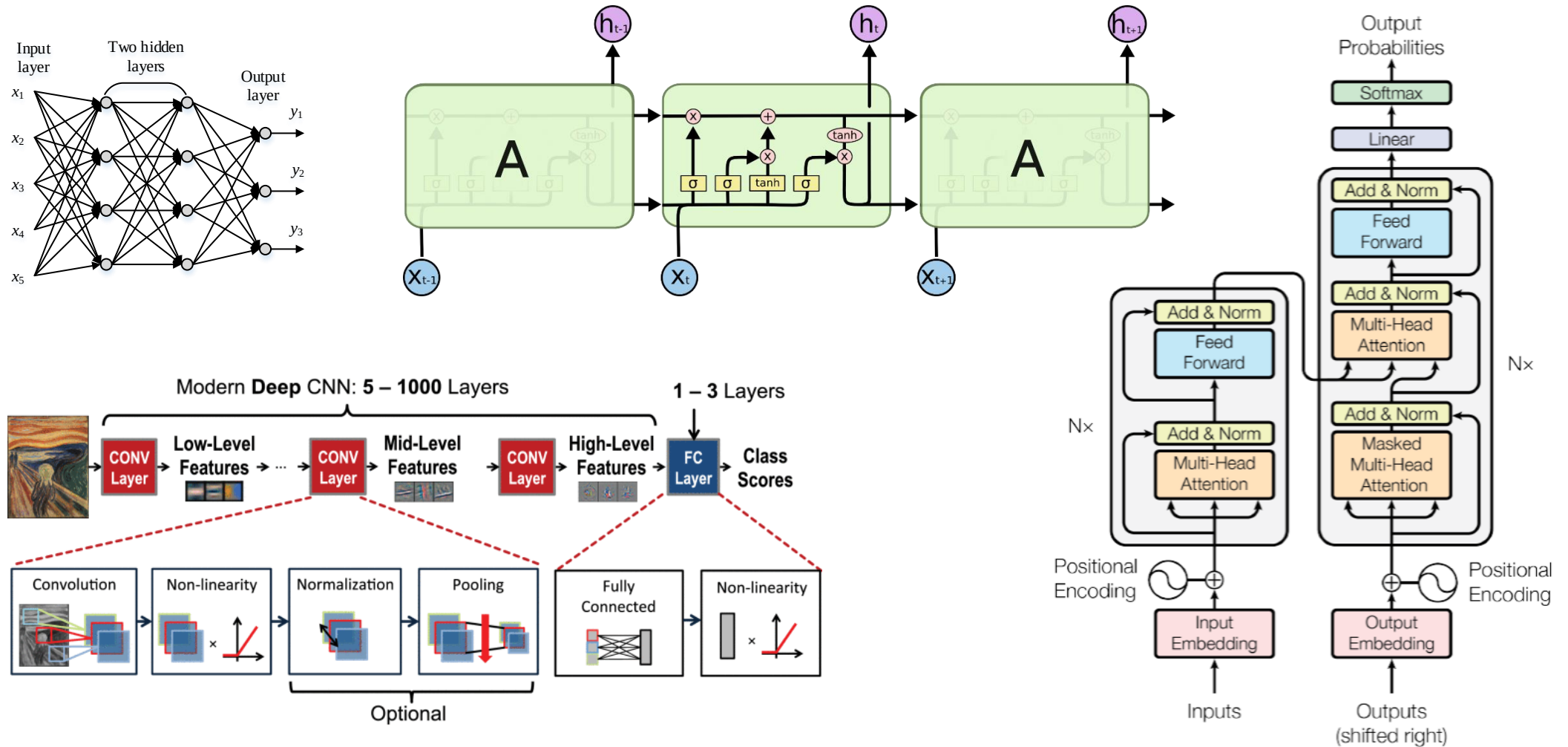
Computer Architecture - EE219
faculty.sist.shanghaitech.edu.cn

立志成才 报国裕民

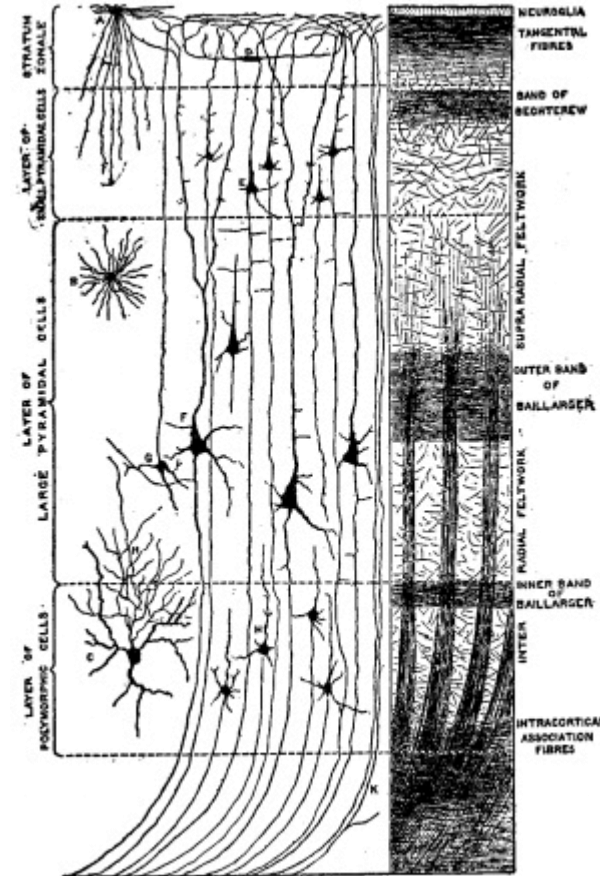
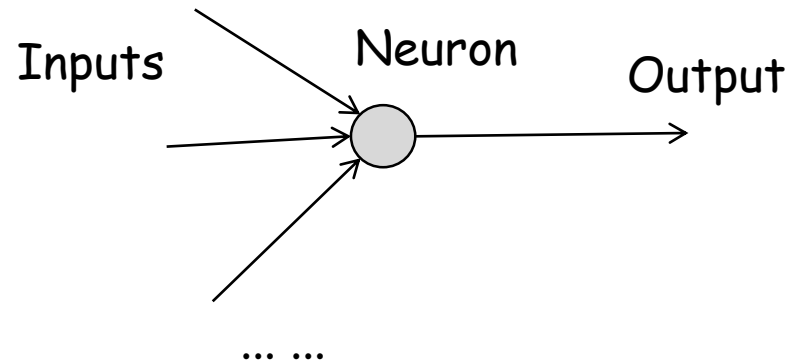
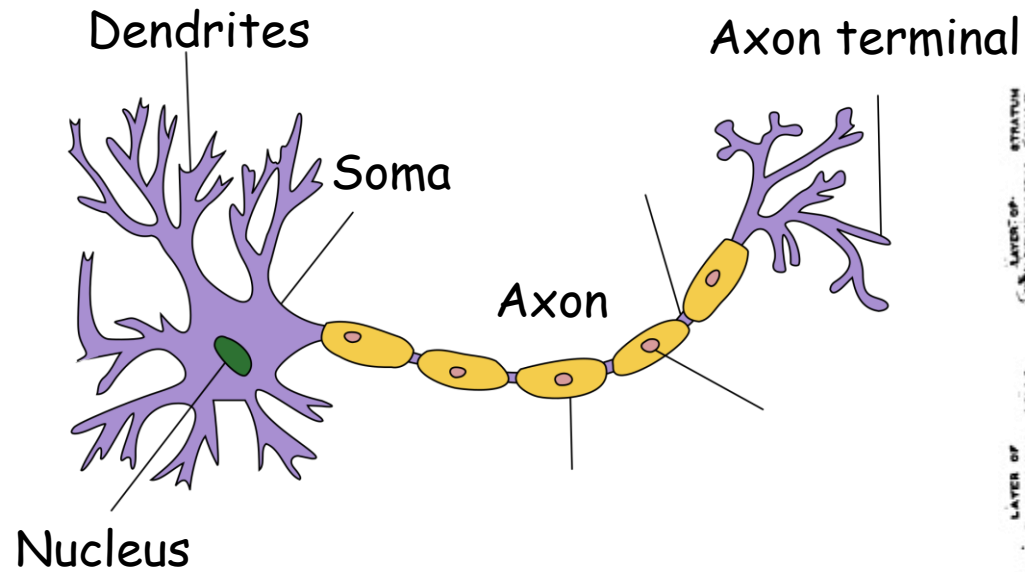


Neural Network Architectures

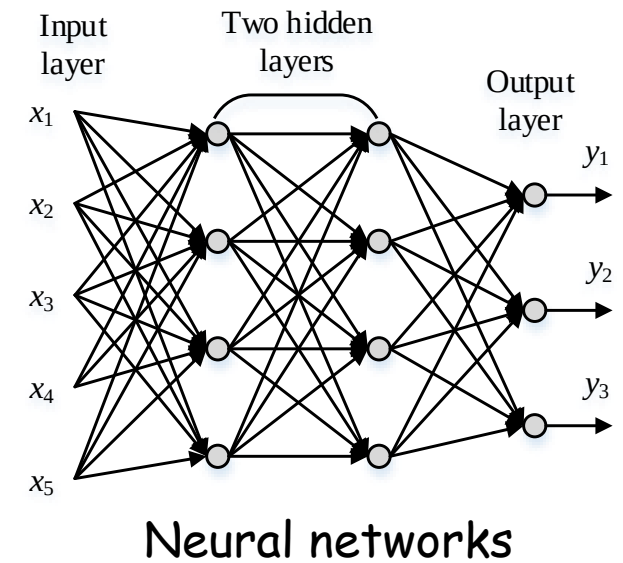
- Model (T)



Origins

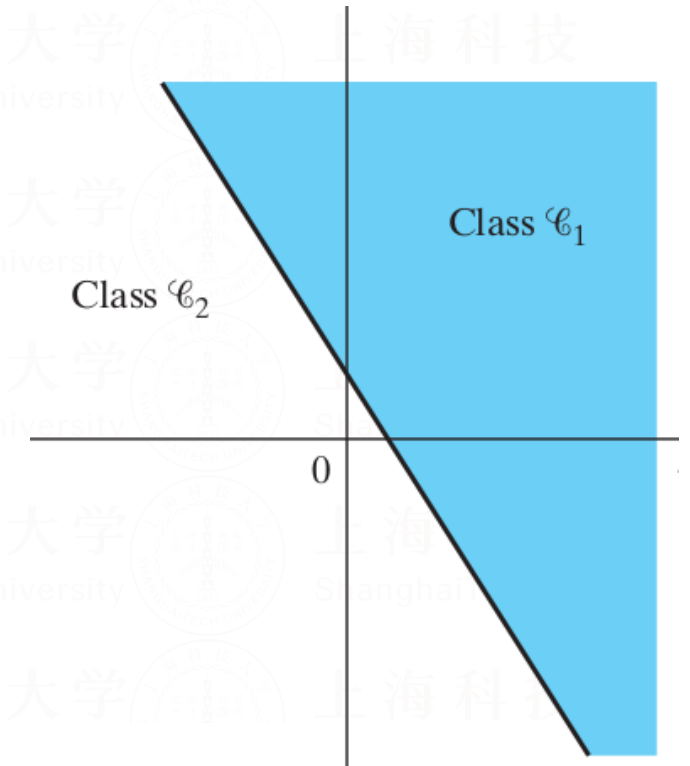
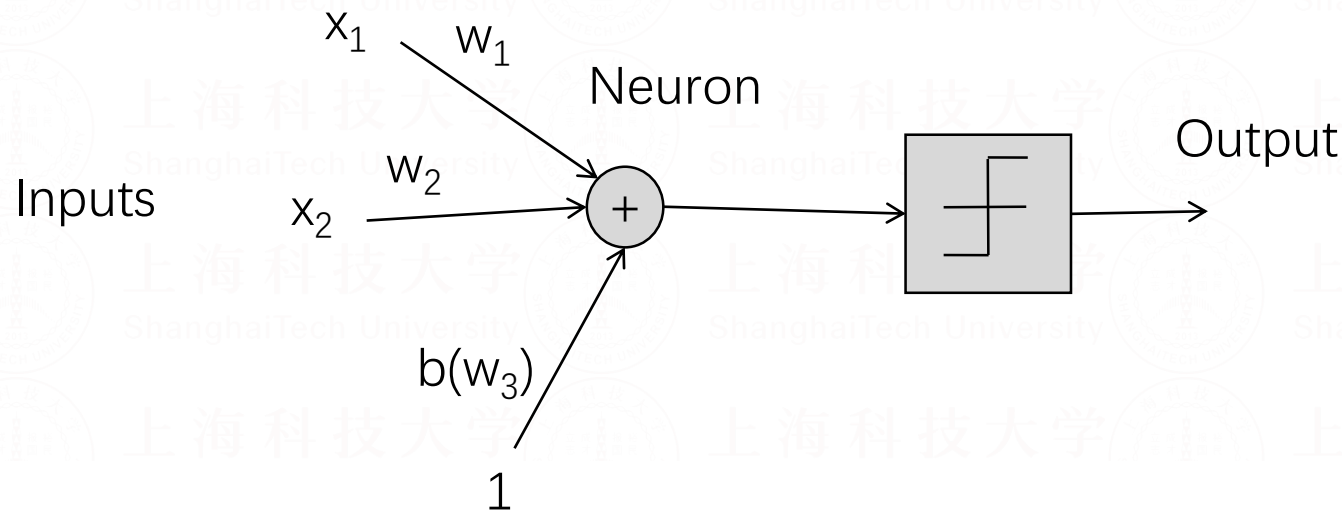


Layered structure in the brain, from Wikipedia



Perceptron (for Linear Separable Problem)

- Linear model followed by hard threshold



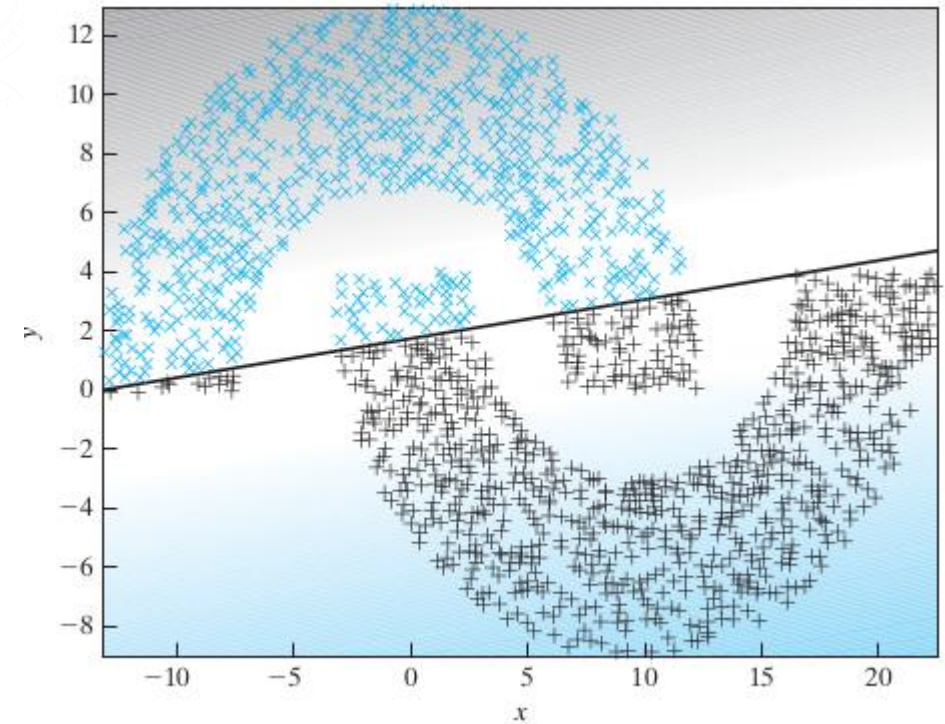
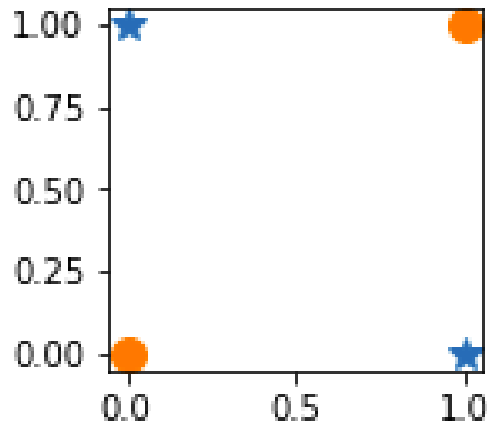
$$w_1x_1 + w_2x_2 + b = 0$$

Decision boundary

Simon O. Haykin, Neural Networks and Learning Machines, third edition.

Perceptron- Limited Capacity

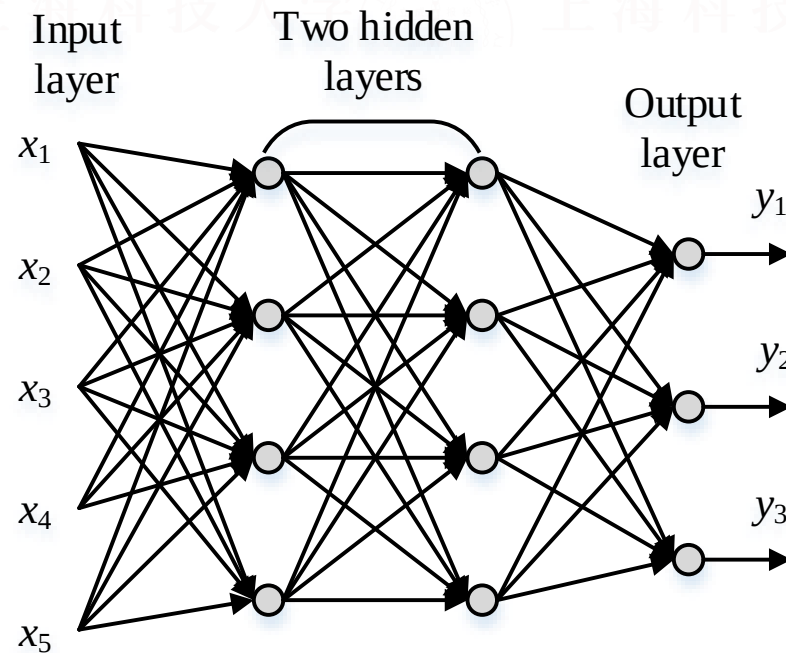
- Not Linear separable



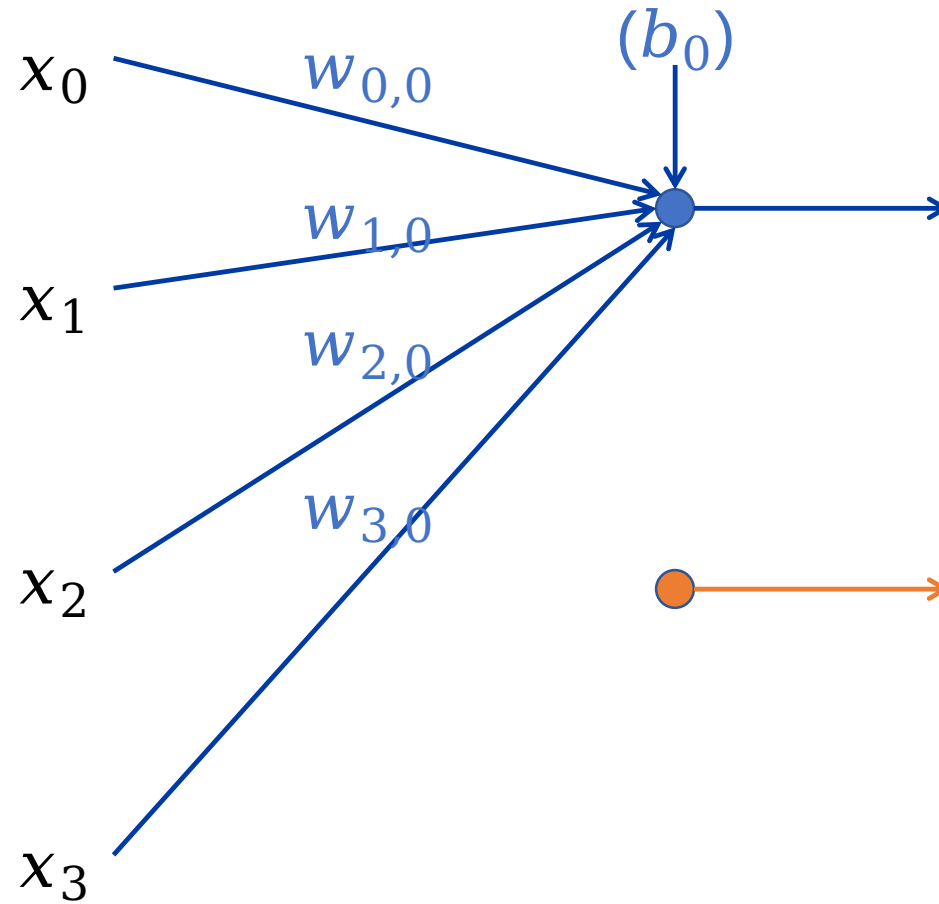
Simon O. Haykin, Neural Networks and Learning Machines, third edition.

Multi-Layer Perceptron (MLP)

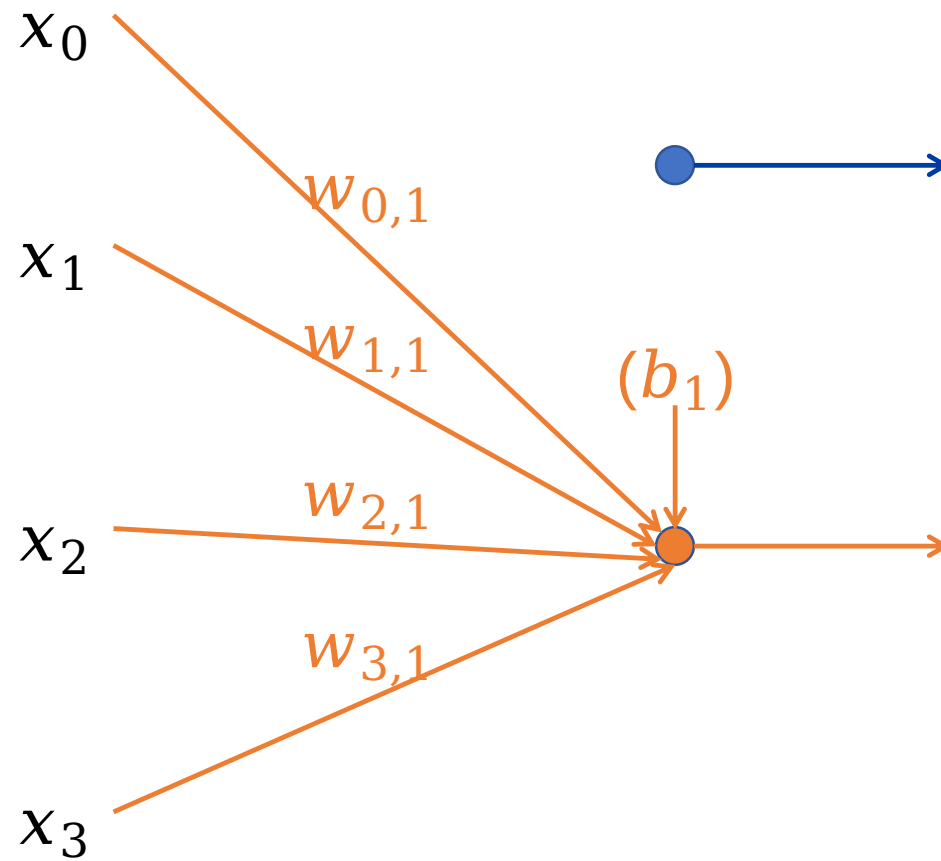
- Introduce a **nonlinear** activation function that is **differentiable**
- Contains **one or more layers** that are hidden from both the input and output nodes
- Exhibits a **high degree of connectivity**



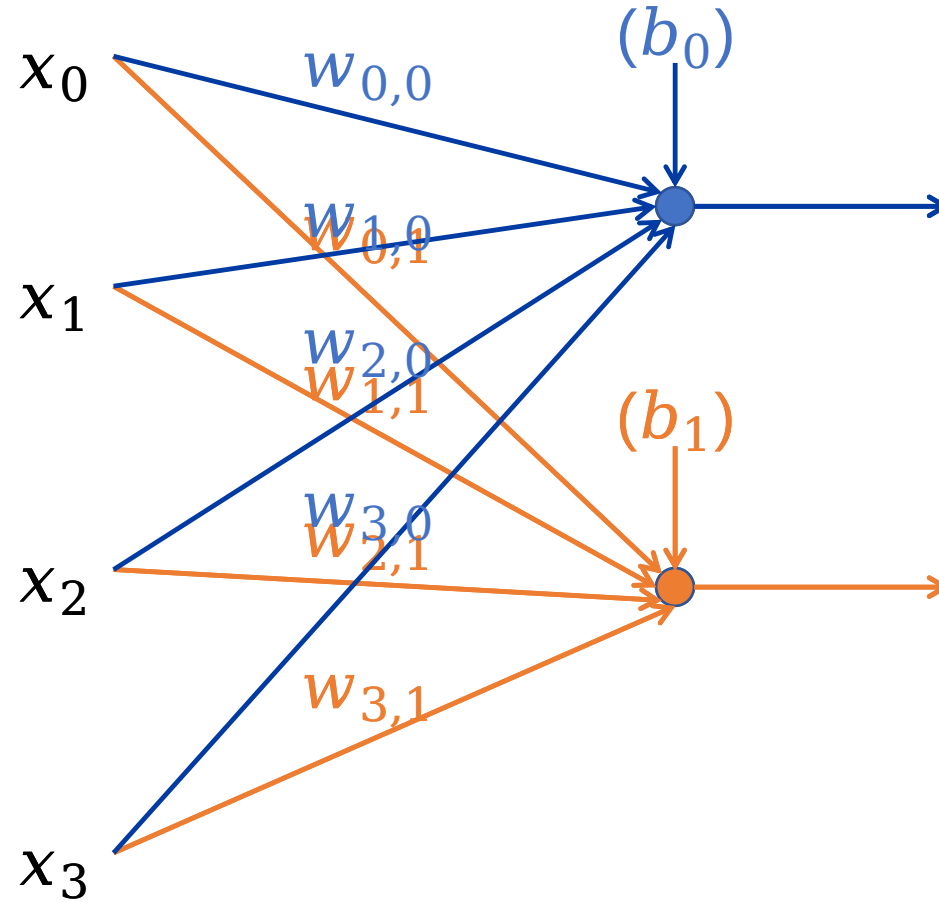
MLP Inference



MLP Inference (Cont'd)



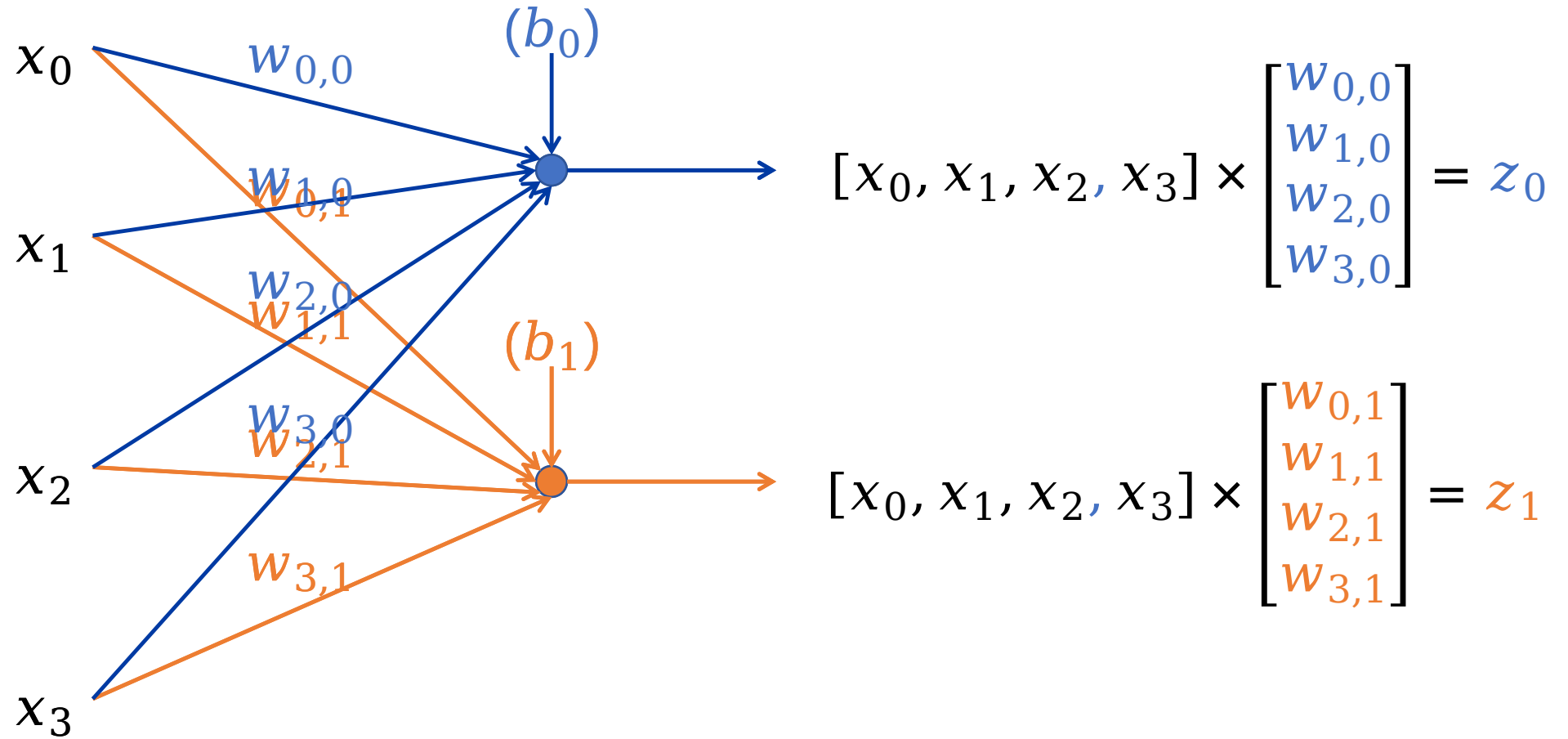
MLP Inference (Cont'd)



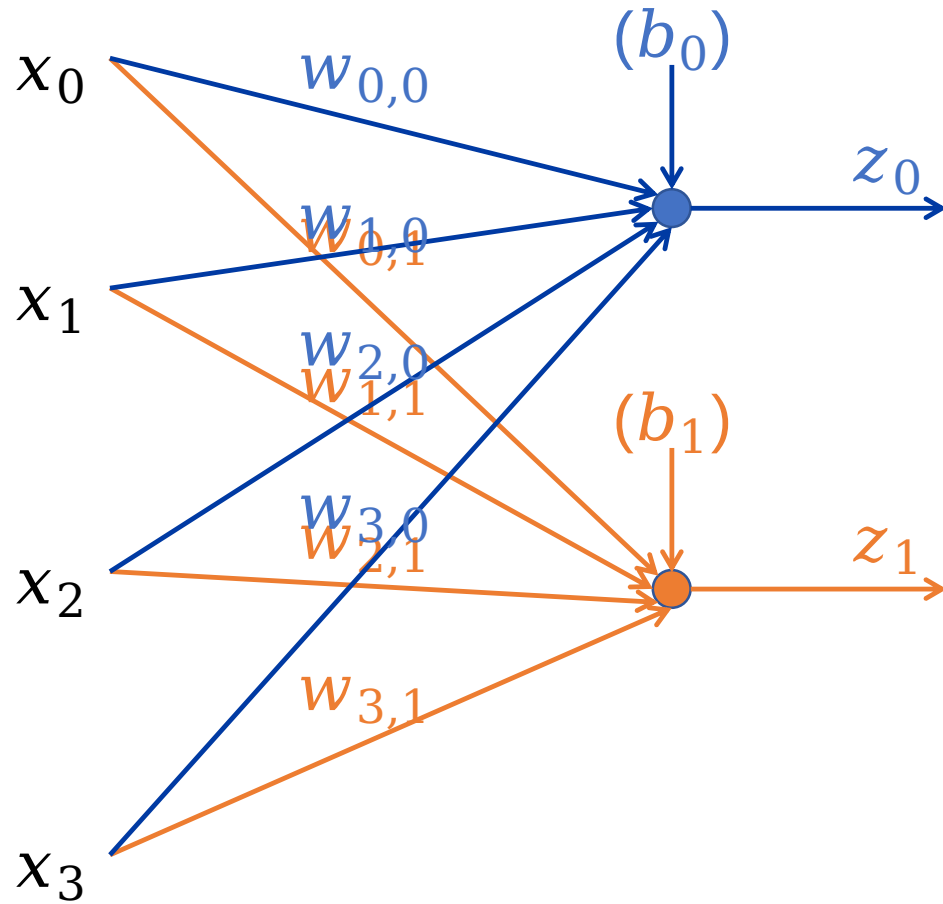
$$x_0 w_{0,0} + x_1 w_{1,0} + x_2 w_{2,0} + x_3 w_{3,0}$$

$$x_0 w_{0,1} + x_1 w_{1,1} + x_2 w_{2,1} + x_3 w_{3,1}$$

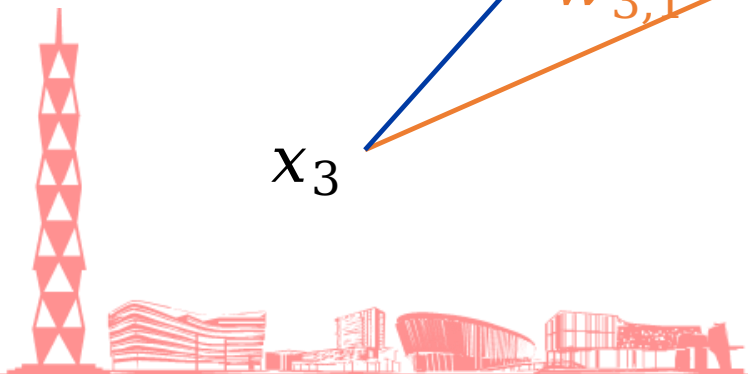
MLP Inference (Cont'd)



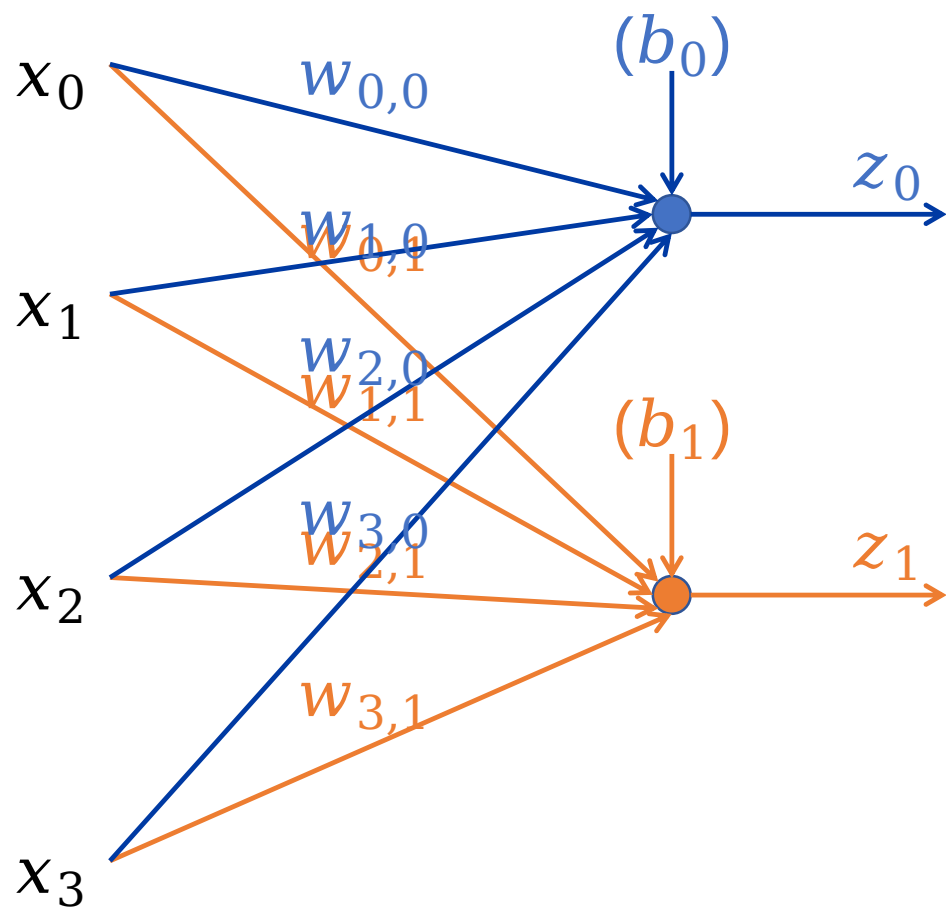
MLP Inference (Cont'd)



$$[x_0, x_1, x_2, x_3] \times \begin{bmatrix} w_{0,0} & w_{0,1} \\ w_{1,0} & w_{0,1} \\ w_{2,0} & w_{0,1} \\ w_{3,0} & w_{0,1} \end{bmatrix} = [z_0 \quad z_1]$$



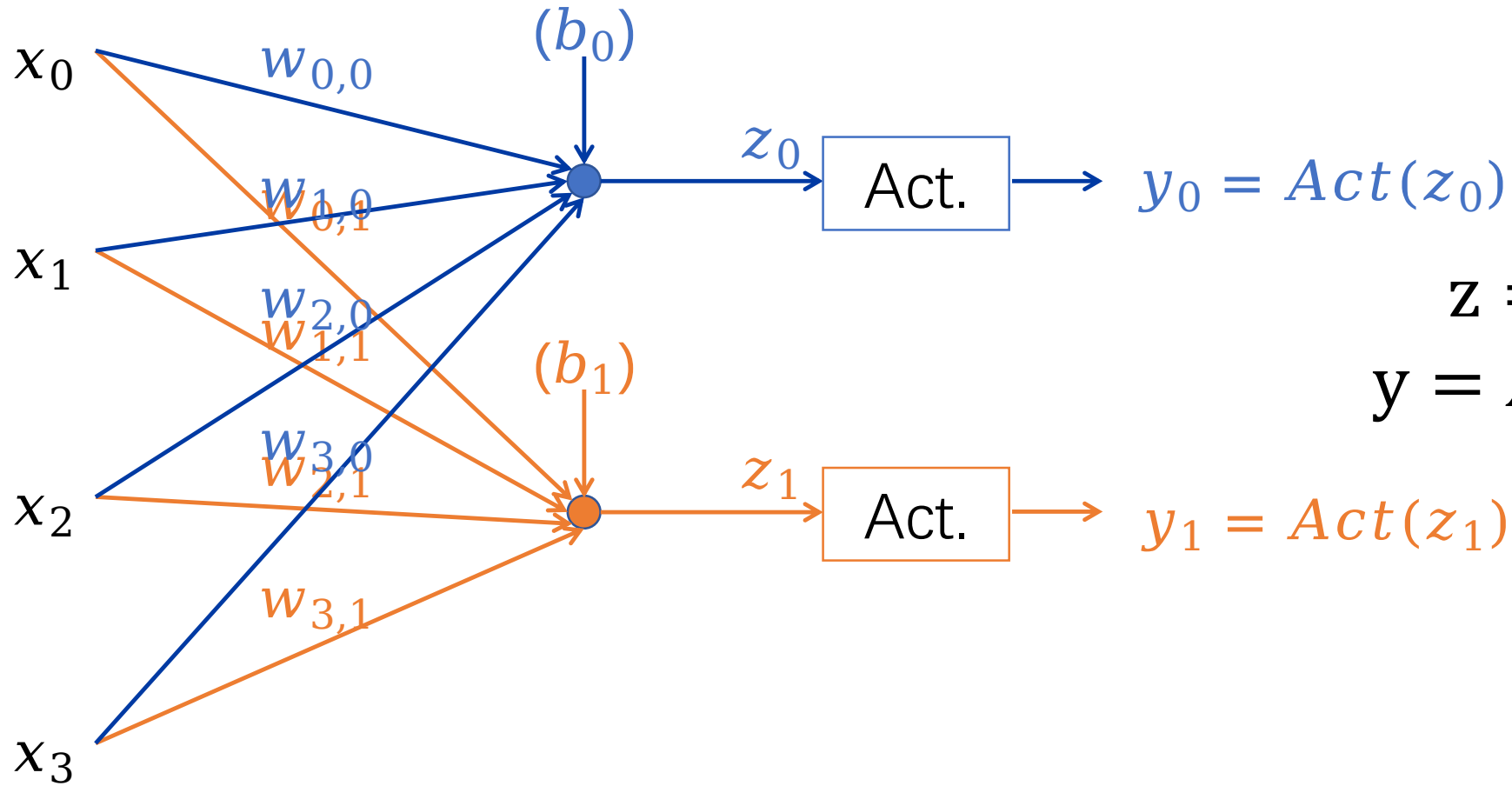
MLP Inference (Cont'd)



$$\mathbf{z} = \mathbf{x} \times \mathbf{w}$$

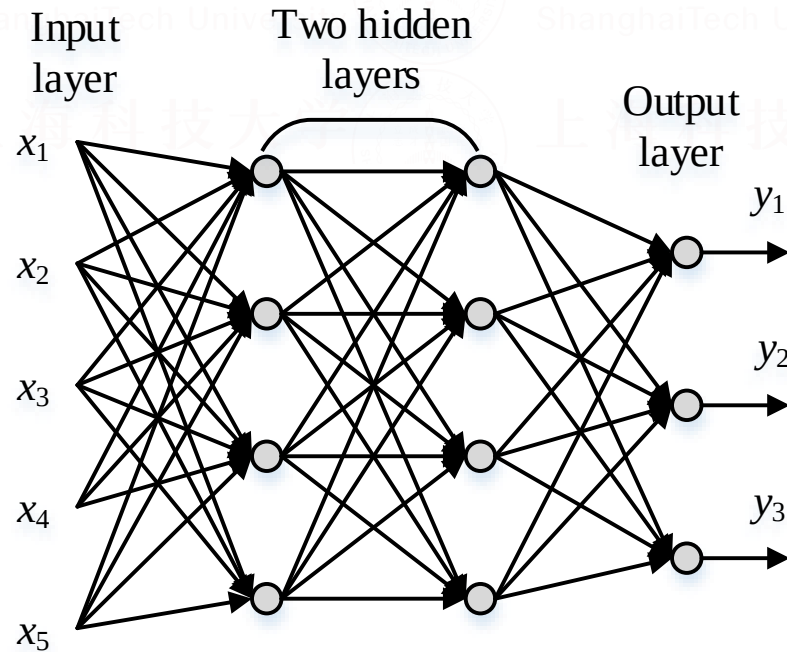


MLP Inference (Cont'd)



MLP Inference (Cont'd)

- Forward pass, single neuron (inference, T)



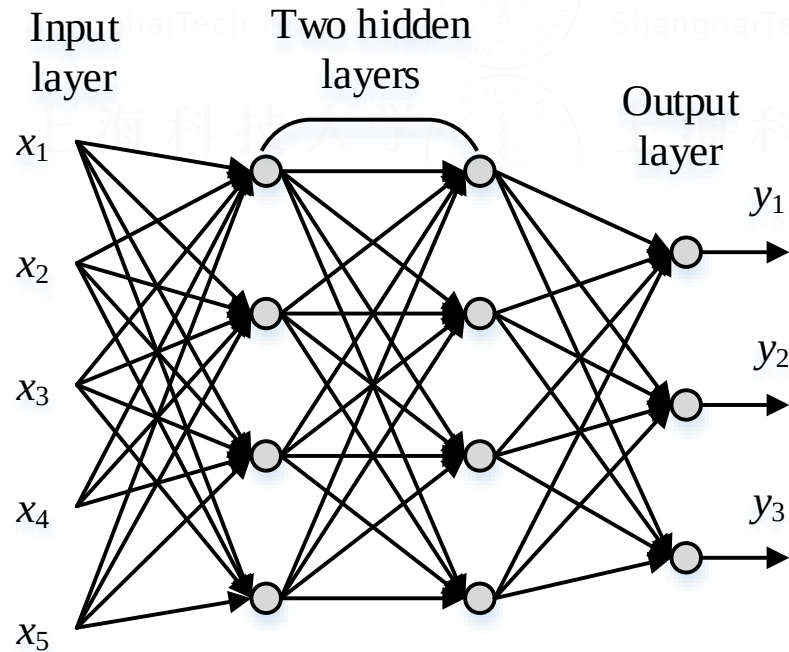
$$\mathbf{y} = f(\mathbf{x}, \mathbf{w})$$

Computation complexity

Activations	Multiplications	Additions
11	$20+16+12=48$	$20+16+12=48$

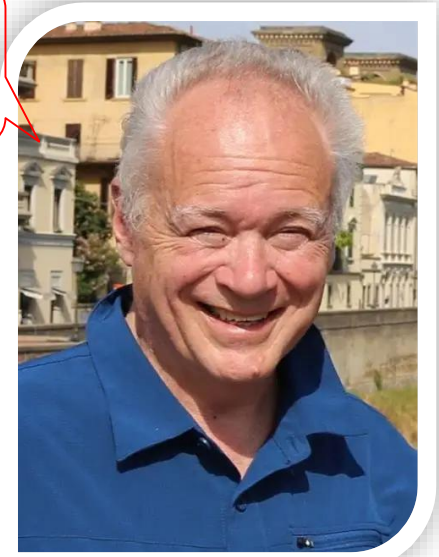
Training of an MLP

- Forward and backward pass



Why need forward and backward pass?

We need the gradients for optimization!!!

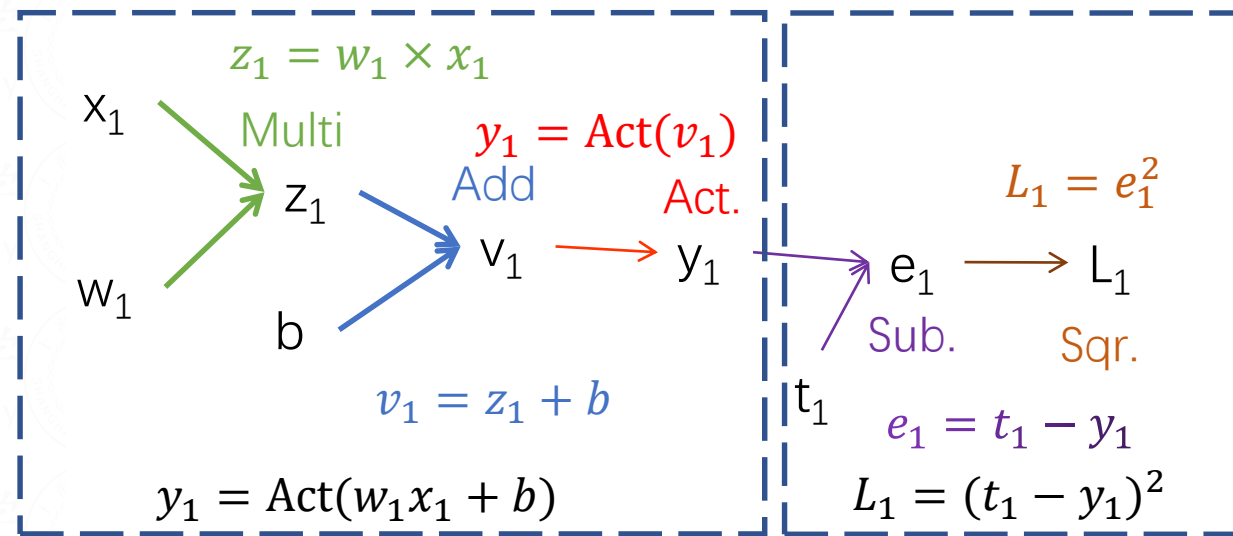


Training of an MLP-a Single Neuron

- Computational graph
- Variables as nodes
- Operation attached to the output
- Apply the chain rule

$$\begin{aligned}\frac{\partial L_1}{\partial w_1} &= \frac{\partial L_1}{\partial y_1} \frac{\partial y_1}{\partial w_1} \\ &= \frac{\partial L_1}{\partial e_1} \frac{\partial e_1}{\partial y_1} \frac{\partial y_1}{\partial v_1} \frac{\partial v_1}{\partial z_1} \frac{\partial z_1}{\partial w_1}\end{aligned}$$

$$= 2e_1(-1) \frac{d \text{Act}(v_1)}{dv_1} (+1)x_1$$



Training of an MLP-a Single Layer, Mini-batch

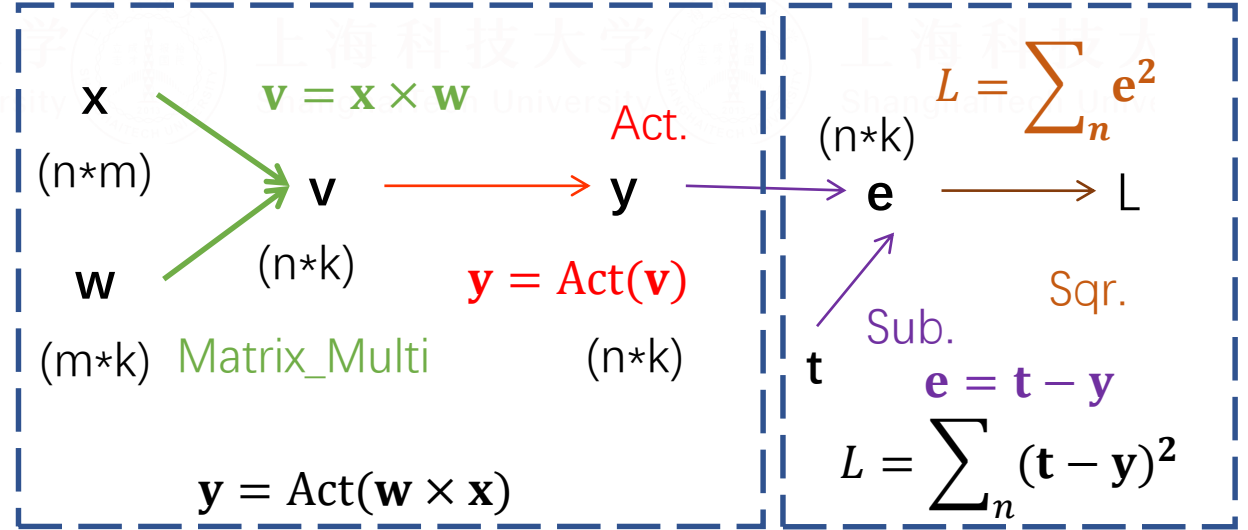
- A mini-batch

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{y}} \otimes \frac{\partial \mathbf{y}}{\partial \mathbf{v}} \left(\frac{\partial \mathbf{v}}{\partial \mathbf{w}} \right)^T$$

$$\frac{\partial L}{\partial \mathbf{y}} =$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{v}} =$$

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}} =$$



Training of an MLP-a Single Layer, Mini-batch

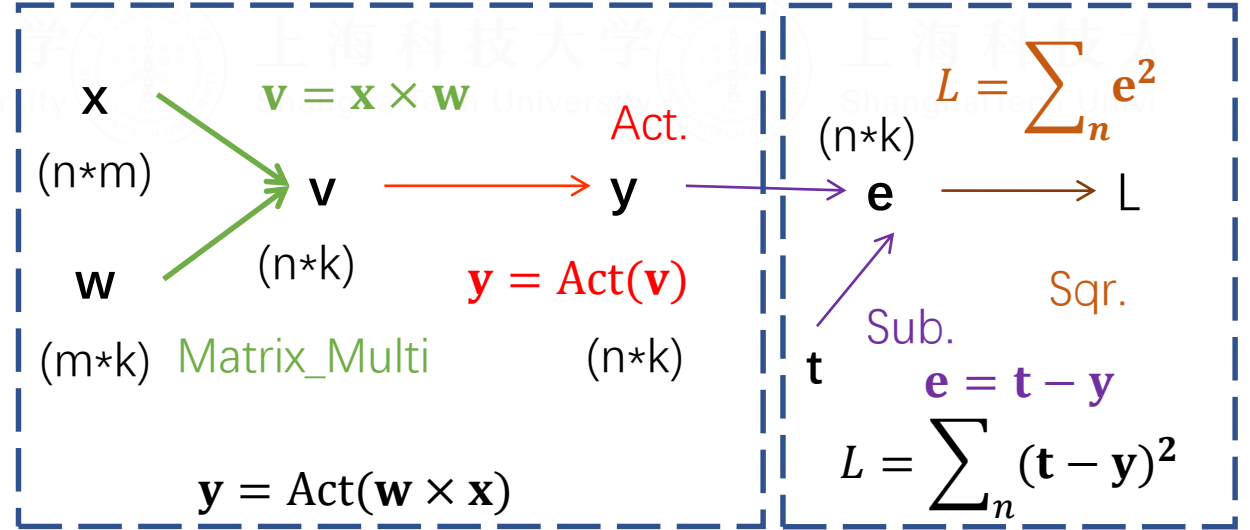
- A mini-batch

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{y}} \otimes \frac{\partial \mathbf{y}}{\partial \mathbf{v}} \left(\frac{\partial \mathbf{v}}{\partial \mathbf{w}} \right)^T$$

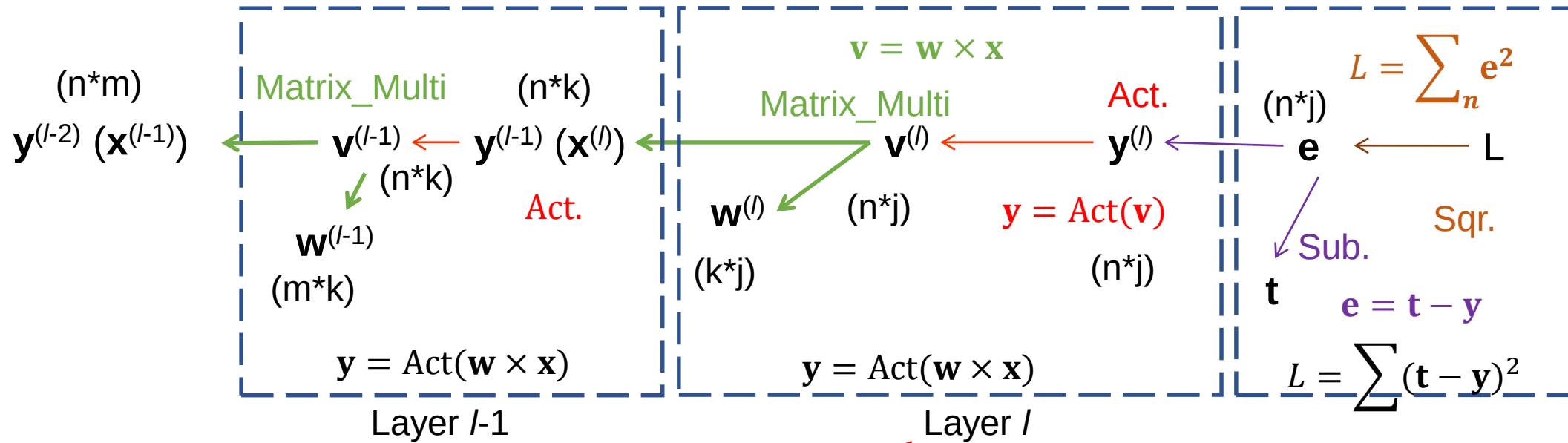
$$\frac{\partial L}{\partial \mathbf{y}} =$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{v}} =$$

$$\frac{\partial \mathbf{v}}{\partial \mathbf{w}} =$$



Training of an MLP-Multi-layer



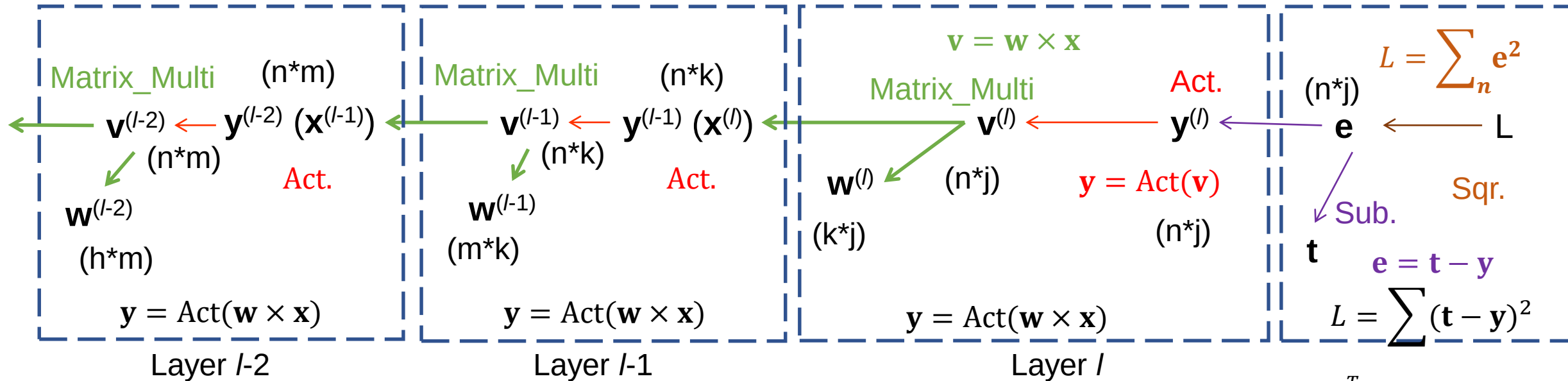
?
$$\frac{\partial L}{\partial \mathbf{w}^{(l-1)}} = \left(\frac{\partial \mathbf{v}^{(l-1)}}{\partial \mathbf{w}^{(l-1)}} \right)^T \left(\frac{\partial L}{\partial \mathbf{y}^{(l)}} \otimes \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{v}^{(l)}} \left(\frac{\partial \mathbf{v}^{(l)}}{\partial \mathbf{y}^{(l-1)}} \right)^T \otimes \frac{\partial \mathbf{y}^{(l-1)}}{\partial \mathbf{v}^{(l-1)}} \right)$$

$$= (\mathbf{x}^{(l-1)})^T \left(\frac{\partial L}{\partial \mathbf{y}^{(l)}} \otimes \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{v}^{(l)}} (\mathbf{w}^{(l)})^T \otimes \frac{\partial \mathbf{y}^{(l-1)}}{\partial \mathbf{v}^{(l-1)}} \right)$$

$$\frac{\partial L}{\partial \mathbf{w}^{(l)}} = \left(\frac{\partial \mathbf{v}}{\partial \mathbf{w}} \right)^T \left(\frac{\partial L}{\partial \mathbf{y}} \otimes \frac{\partial \mathbf{y}}{\partial \mathbf{v}} \right)$$

$$= (\mathbf{x}^{(l)})^T \left(\frac{\partial L}{\partial \mathbf{y}^{(l)}} \otimes \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{v}^{(l)}} \right)$$

Training of an MLP-Multi-layer (Cont'd)



$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}^{(l-2)}} &= \left(\frac{\partial \mathbf{v}^{(l-2)}}{\partial \mathbf{w}^{(l-2)}} \right)^T \left(\frac{\partial L}{\partial \mathbf{v}^{(l-1)}} \right) \left(\frac{\partial \mathbf{v}^{(l-1)}}{\partial \mathbf{y}^{(l-2)}} \right)^T \otimes \frac{\partial \mathbf{y}^{(l-2)}}{\partial \mathbf{v}^{(l-2)}} \\ &= (\mathbf{x}^{(l-2)})^T \left(\frac{\partial L}{\partial \mathbf{v}^{(l-1)}} \right) (\mathbf{w}^{(l-1)})^T \otimes \frac{\partial \mathbf{y}^{(l-2)}}{\partial \mathbf{v}^{(l-2)}} \end{aligned}$$

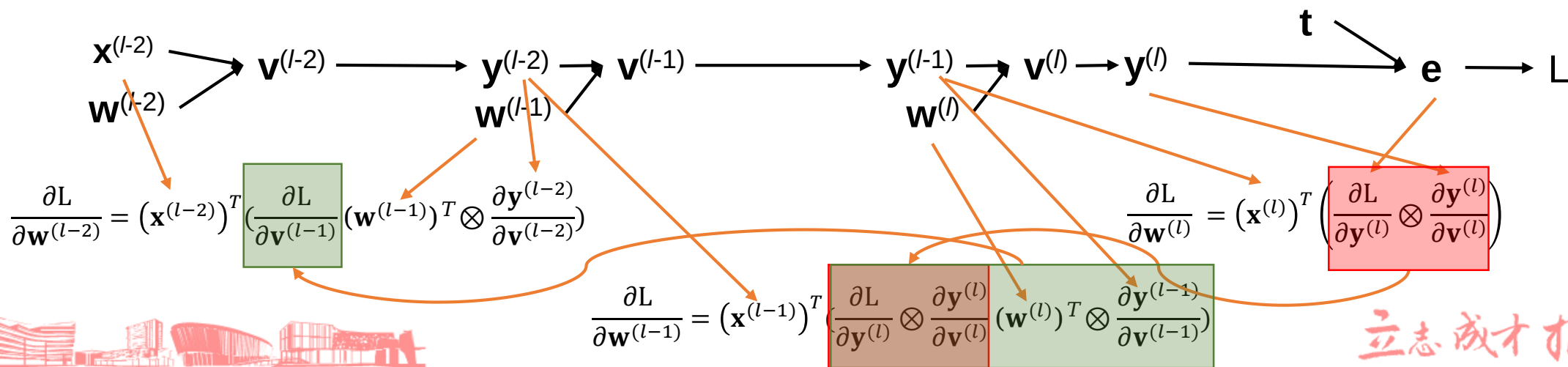
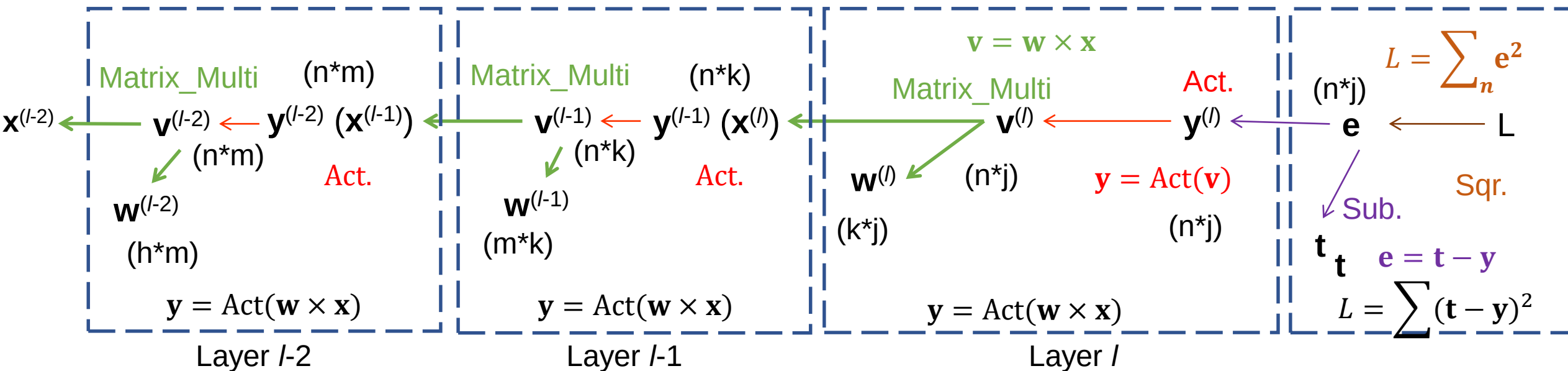
Layer l :

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}^{(l)}} &= \left(\frac{\partial \mathbf{v}}{\partial \mathbf{w}} \right)^T \left(\frac{\partial L}{\partial \mathbf{y}} \otimes \frac{\partial \mathbf{y}}{\partial \mathbf{v}} \right) \\ &= (\mathbf{x}^{(l)})^T \left(\frac{\partial L}{\partial \mathbf{y}^{(l)}} \otimes \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{v}^{(l)}} \right) \end{aligned}$$

Layer $l-1$:

$$\frac{\partial L}{\partial \mathbf{w}^{(l-1)}} = (\mathbf{x}^{(l-1)})^T \left(\frac{\partial L}{\partial \mathbf{y}^{(l)}} \otimes \frac{\partial \mathbf{y}^{(l)}}{\partial \mathbf{v}^{(l)}} \right) (\mathbf{w}^{(l)})^T \otimes \frac{\partial \mathbf{y}^{(l-1)}}{\partial \mathbf{v}^{(l-1)}}$$

Memory Requirement during Training



The Activation Functions

- Rectified linear unit (ReLU)

$$\text{Act}(x) = \text{Max}(x, 0)$$

$$\frac{d \text{Act}(x)}{dx} = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

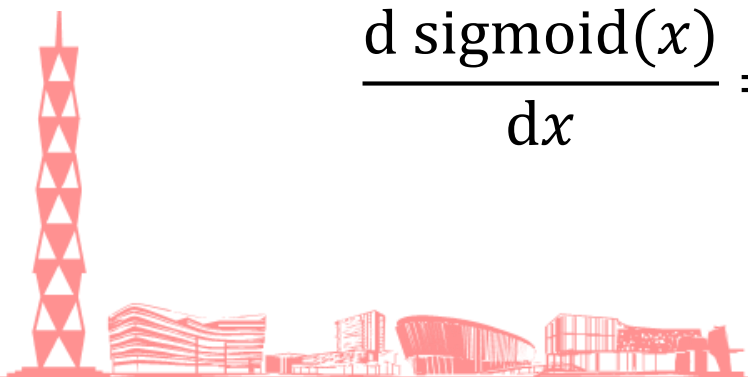
- Sigmoidal functions (sigmoid, tanh)

$$y = \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

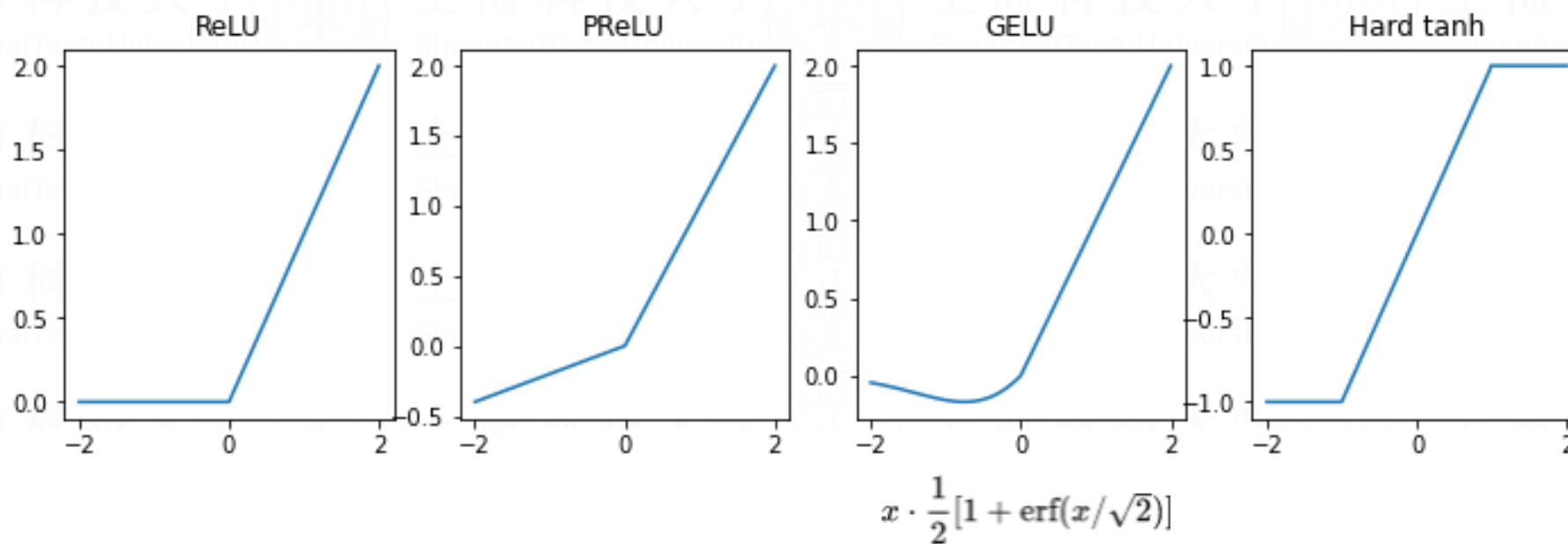
$$\frac{d \text{sigmoid}(x)}{dx} = y(1 - y)$$

$$\frac{d \tanh(x)}{dx} = 1 - y^2$$



The Activation Functions (Cont'd)

- Leaky/parametric ReLU (PReLU), exponential linear unit (ELU), Gaussian error linear unit (GELU), etc.



See more at <https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity>.

Softmax

- Convert real vector into probability distribution. For a K-dimensional vector $\mathbf{z}$ (logits)

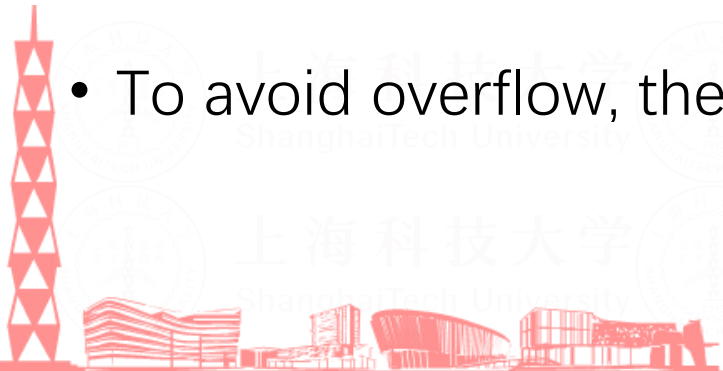
$$y_i = \text{softmax}(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

- When cross entropy is used as the cost function, it leads to easy differentiation.

$$L = - \sum_i^K y_i \log \hat{y}_i$$

$$\frac{\partial L}{\partial y_i} = \hat{y}_i - y_i \longrightarrow \text{Label}$$

- To avoid overflow, the equation is adjusted as $\text{softmax}(\mathbf{z}) = \frac{e^{z_i - \max_j z_j}}{\sum_{j=1}^K e^{z_j - \max_j z_j}}$.



Softmax (Cont'd)

- Can be used as the classification layer of a neural network. The labels are usually one-hot coded.

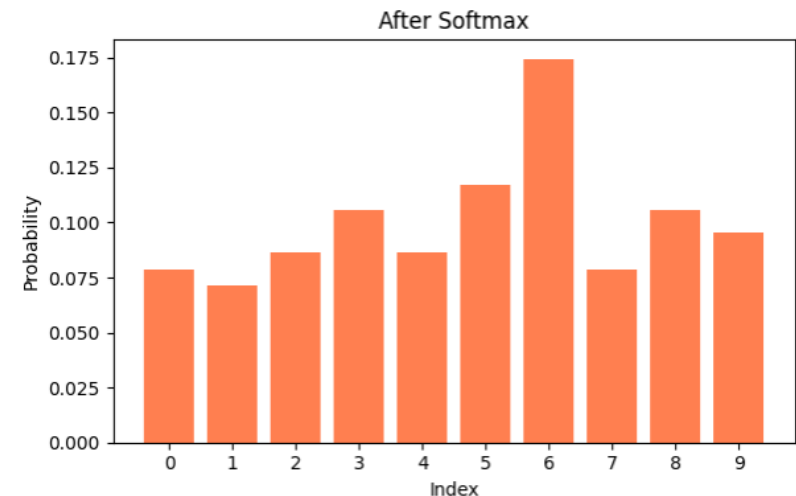
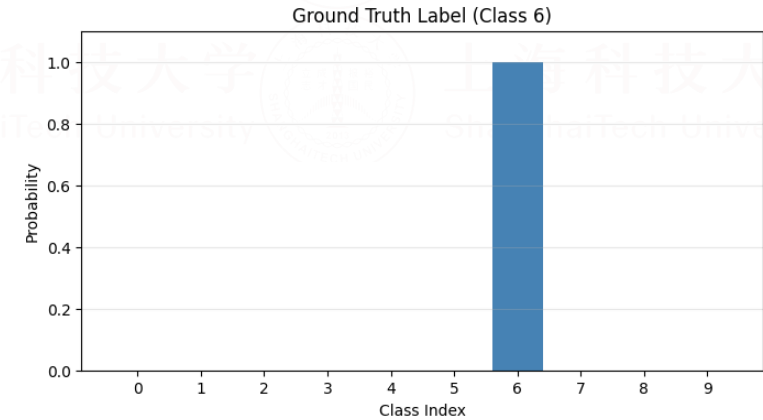
A '6' from MNIST

6 $\longrightarrow$ [0 0 0 0 0 0 1 0 0 0] Label

The logits could be

[0.1 0.01 0.2 0.4 0.2 0.5 0.9 0.1 0.4 0.3]

Top-k accuracy



Softmax (Cont'd)

- Can be used as the classification layer of a neural network. The labels are usually one-hot coded.

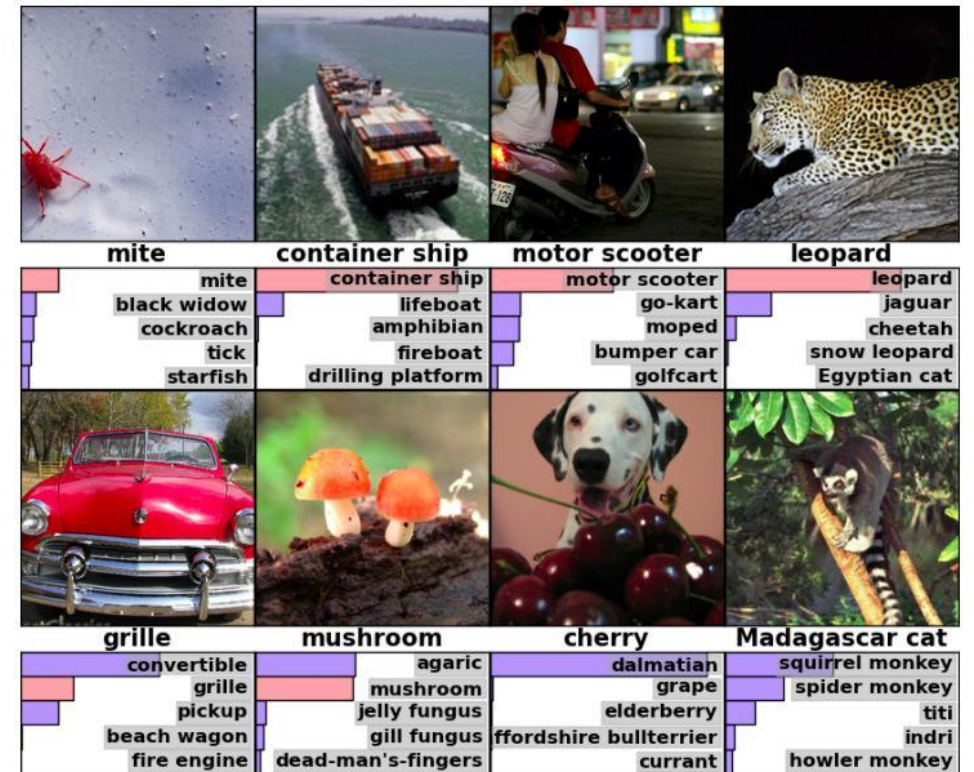
A '6' from MNIST

6 $\longrightarrow$ [0 0 0 0 0 0 1 0 0 0] Label

The logits could be

[0.1 0.01 0.2 0.4 0.2 0.5 0.9 0.1 0.4 0.3]

Top-k accuracy



Miscellaneous in Training

- Weight initialization
 - Weight initialization affects the outcome of training greatly;
 - Historically, initialized with random small values (uniform or normal distributed)
 - He (Kaiming) initialization (ReLU) & Xavier initialization (tanh/sigmoid)

$$\text{weight} = U \left[-\frac{\sqrt{6}}{\sqrt{n}}, \frac{\sqrt{6}}{\sqrt{n}} \right] \quad \text{weight} = U \left[-\frac{\sqrt{6}}{\sqrt{n+m}}, \frac{\sqrt{6}}{\sqrt{n+m}} \right], \quad n: \text{fan in}, \quad m: \text{fan out}$$

See more at <https://pytorch.org/docs/stable/nn.init.html#torch-nn-init>

Understanding the difficulty of training deep feedforward neural networks. **Xavier Glorot, Yoshua Bengio** *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, PMLR 9:249-256, 2010.

Kaiming He, et al. titled "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification."

Miscellaneous in Training (Cont'd)

- Learning rate scheduling
- `torch.optim.lr_scheduler` in pytorch
 - StepLR: multiplicative factor of *gamma* for every *step_size* epochs
 - CosineAnnealingLR: the learning rate follows a sine wave
 - ReduceLROnPlateau: Reduce learning rate when a metric has stopped improving
- See `torch.optim` for more details: <https://pytorch.org/docs/stable/optim.html>

$$w[n] = w[n - 1] - \overset{\text{learning rate}}{\mu} \frac{\partial L}{\partial w[n - 1]}$$



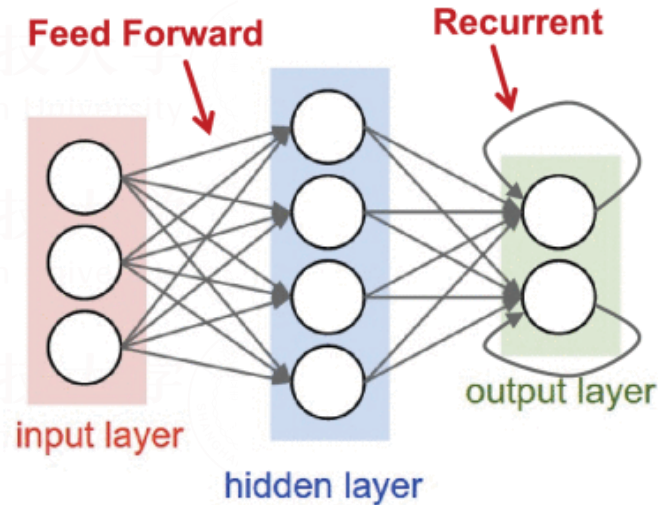
Implications on Hardware

- Inference
 - Forward pass only (with optimized weights)
 - Require less memory & computation
 - Mostly can be deployed on edge devices (could be accomplished by AI inference chips)
 - As the emergence of LLM, some applications, e.g., text/image generation need to be done on servers/cloud (could be with AI inference chips)
 - Usually one inference at a time (application dependent)
 - Usually no transpose required
- Training
 - Forward & backward propagation
 - Require more memory & computation
 - Mostly deployed on servers/cloud (could be equipped with AI training chips), or PCs with high-end GPUs
 - Trained at batch (more parallelism can be exploited)
 - Transpose is complex on hardware

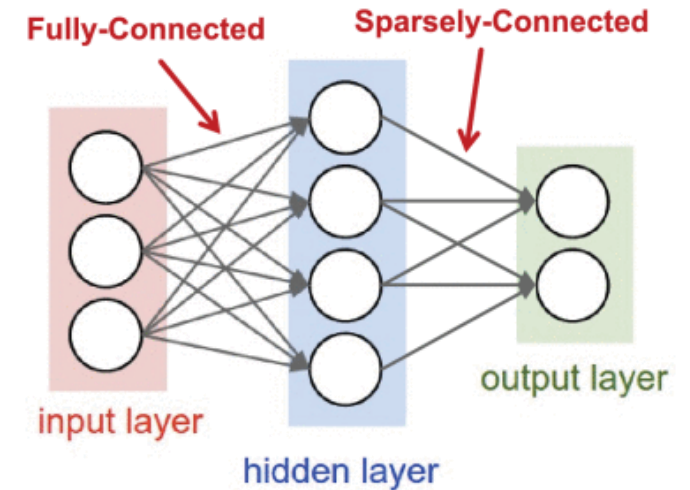


Deep Neural Networks

- Fully connected layers
 - Usually used as the last few layers
- Recurrent/feedback
- Sparsely-connected
 - E.g. Conv. layer
- Fewer parameters

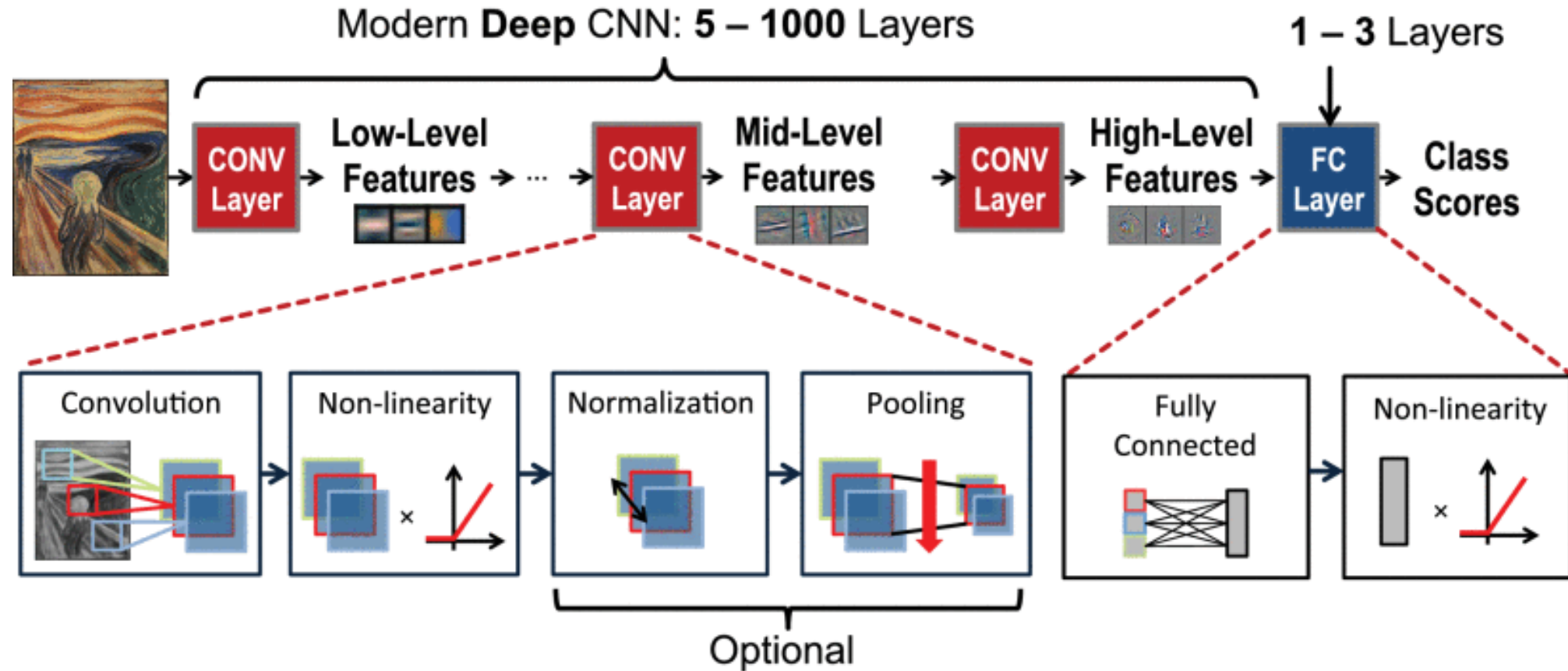


(a) Feedforward versus feedback (re-current) networks



(b) Fully connected versus sparse

Deep Neural Network Example

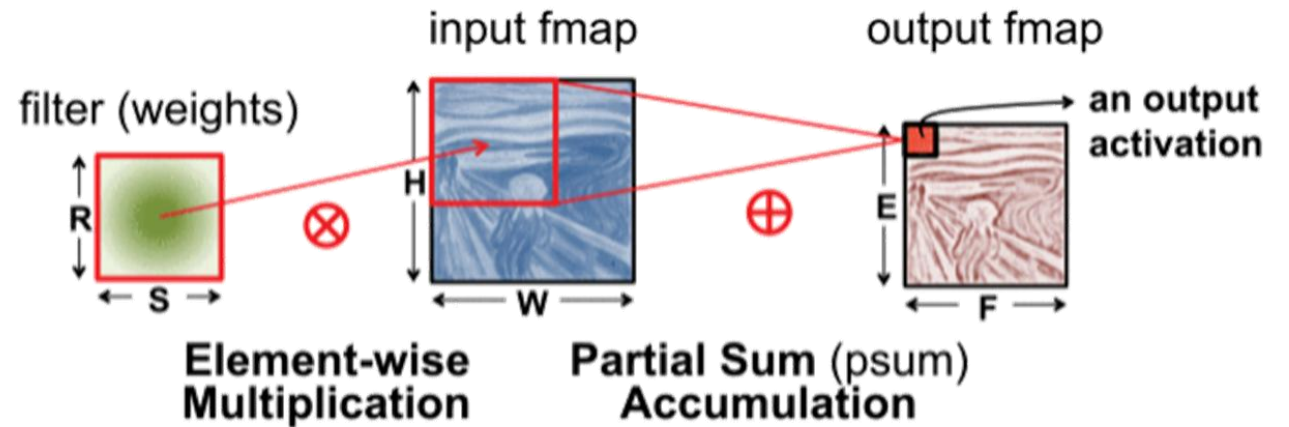


V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017.

Convolution

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`

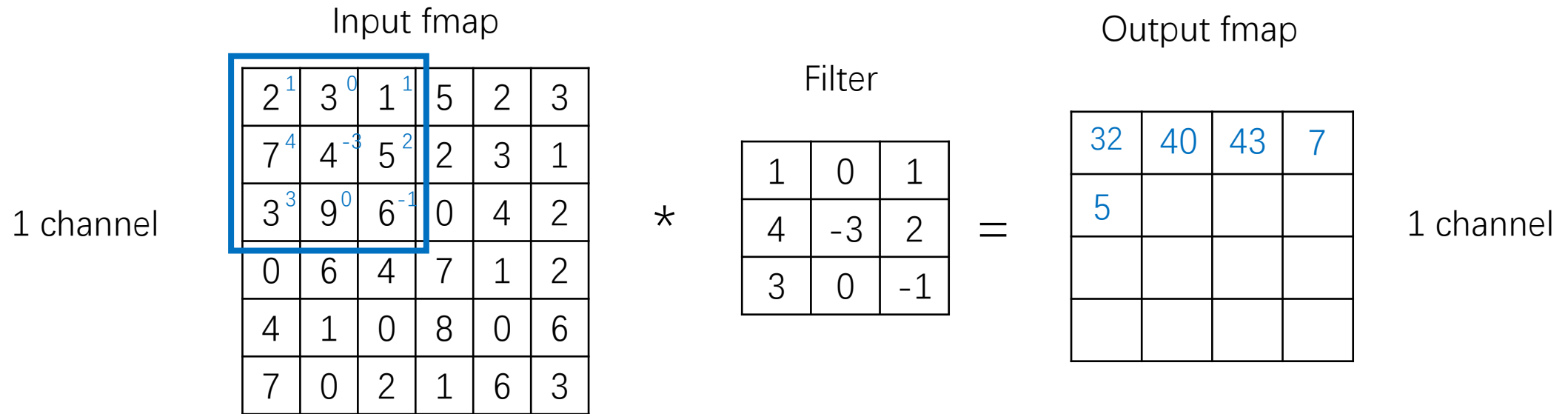
- Conv. (sparsely-connected layers with shared weights)
 - **Feature maps**
 - Channels
 - **Filters/kernels (weights)**



(a) 2-D convolution in traditional image processing

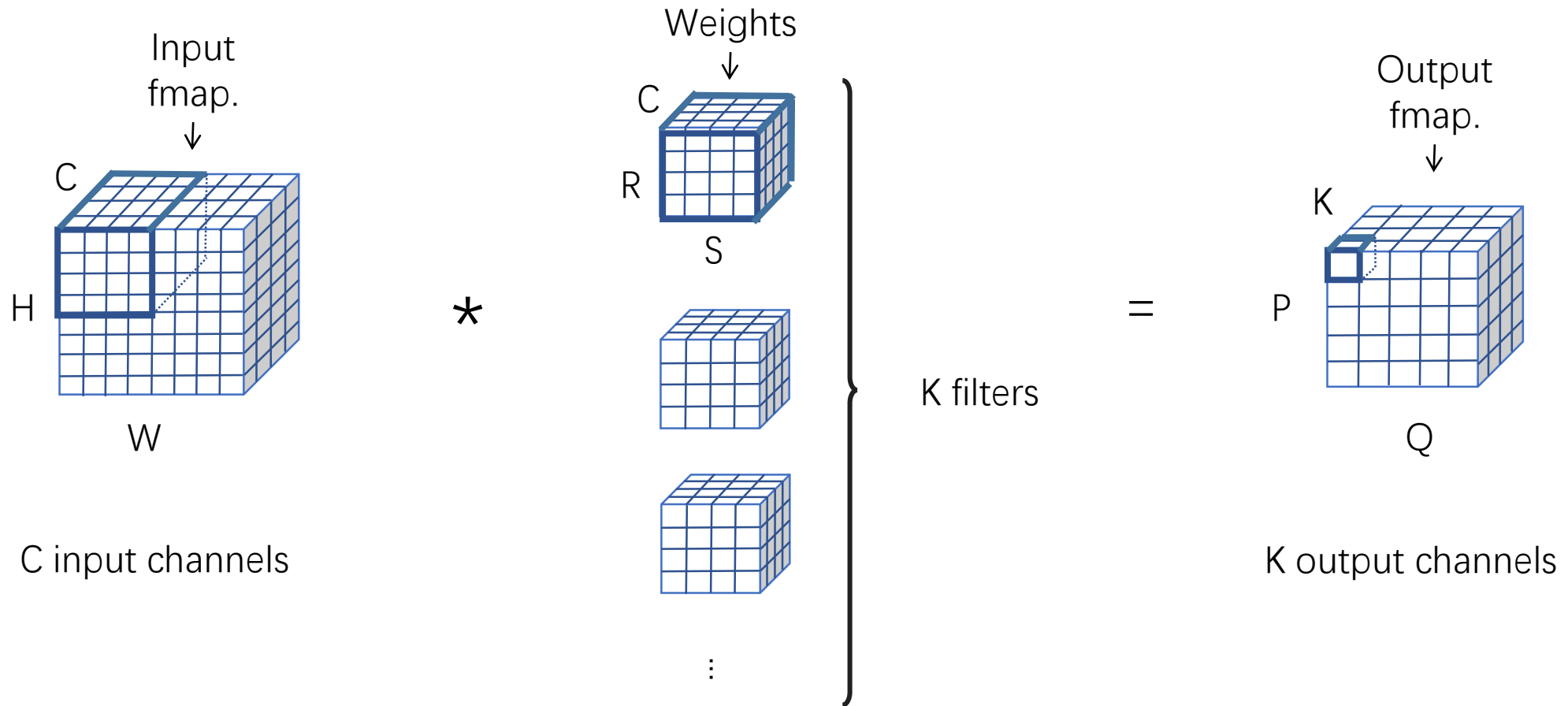
Basic 2D Convolution

- Recognize objects by computing the similarity between the feature map and the filter



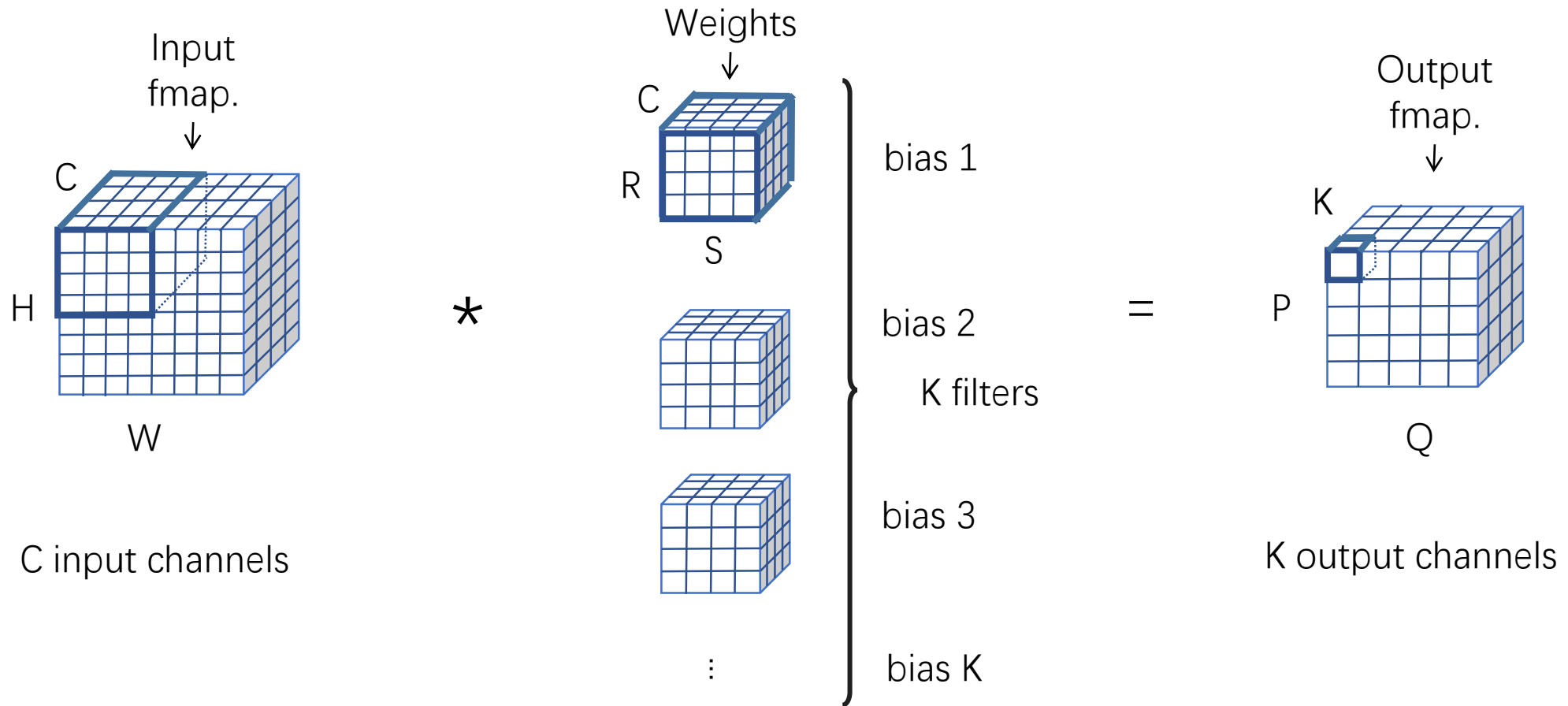
Convolution with 3D Inputs

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`



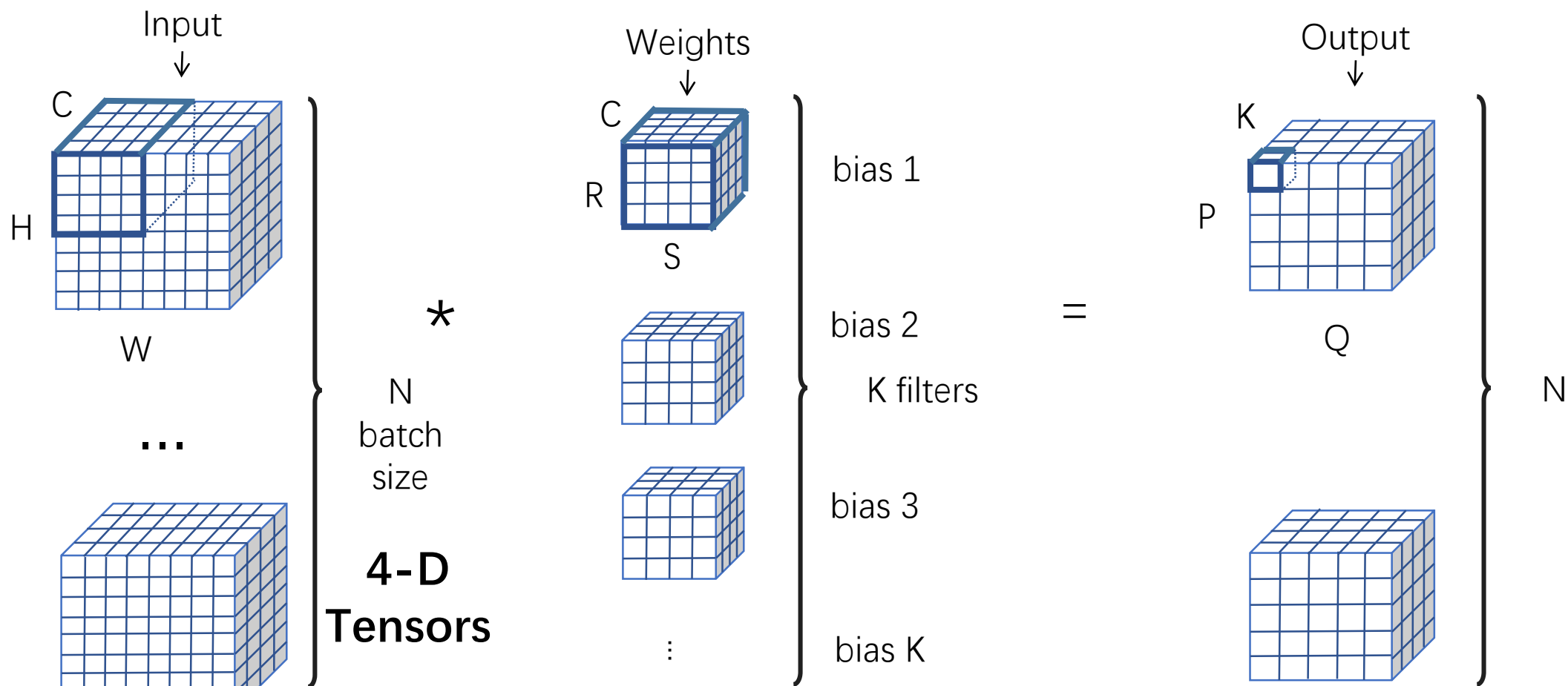
Convolution with Biases

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`



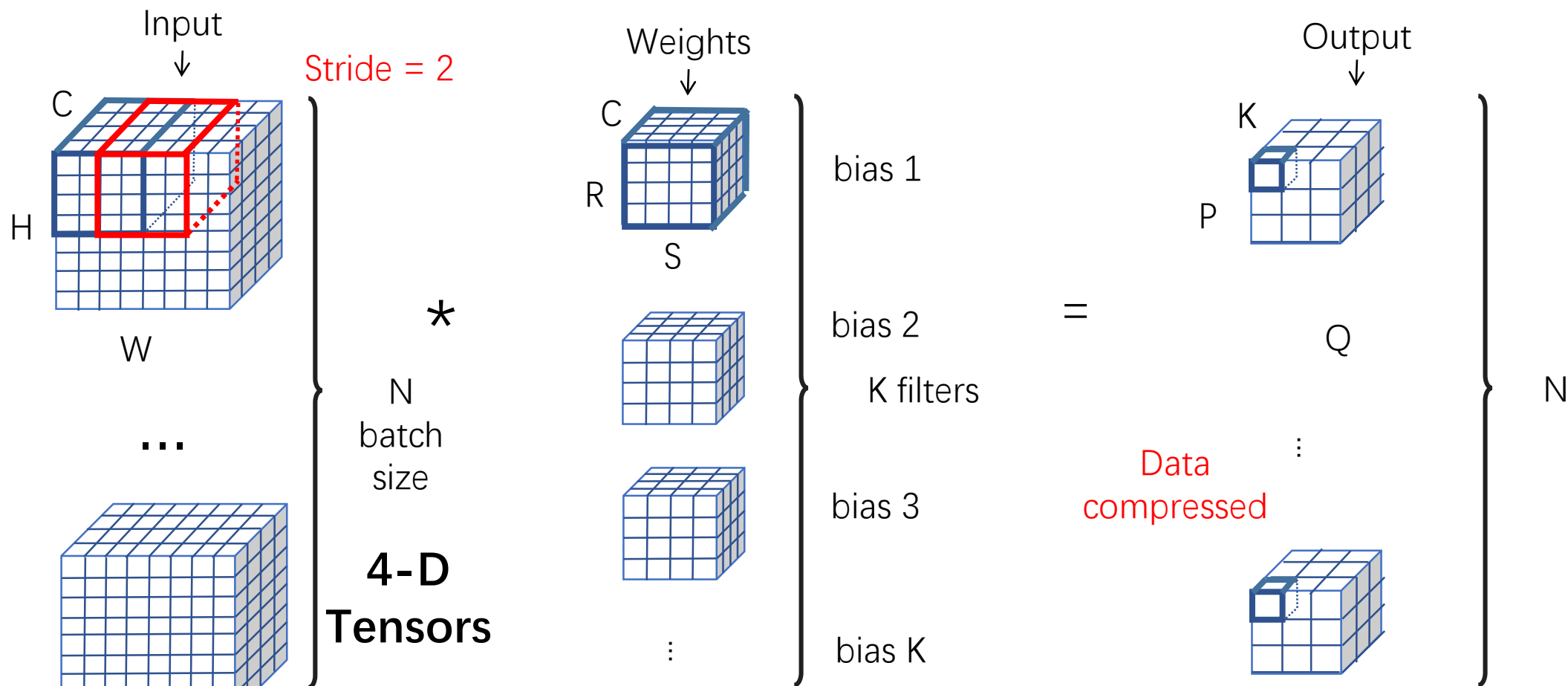
Batched Convolution

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`



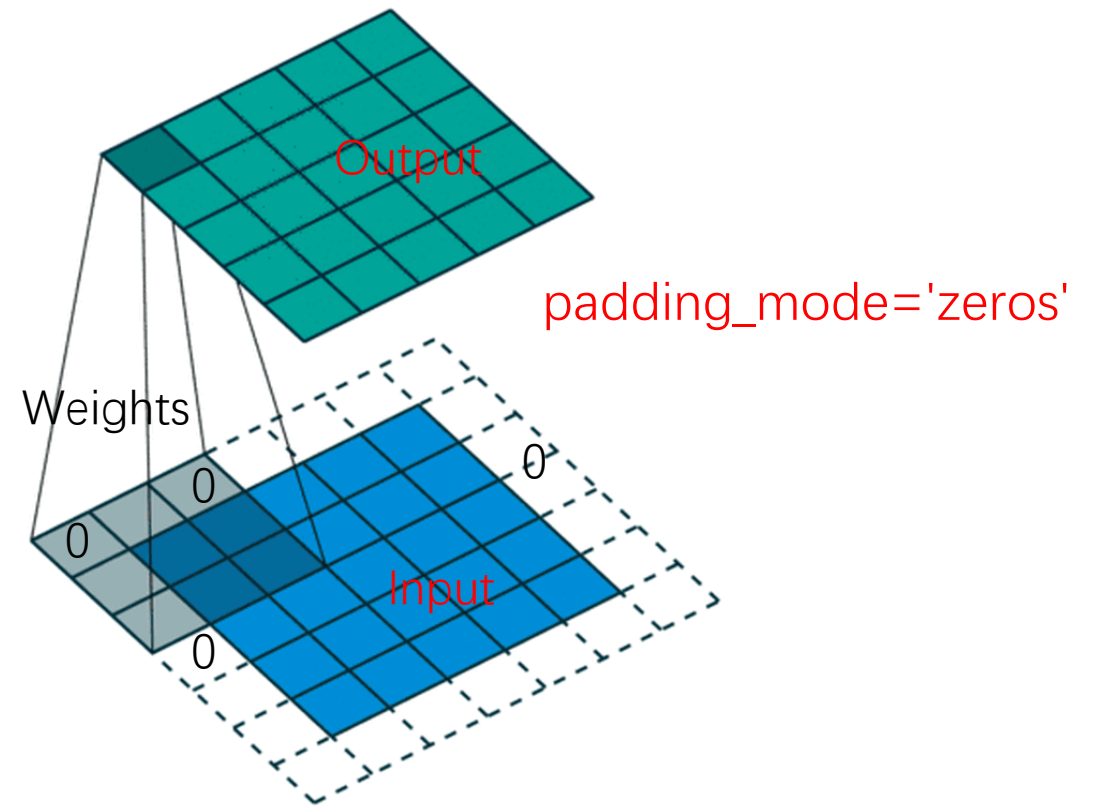
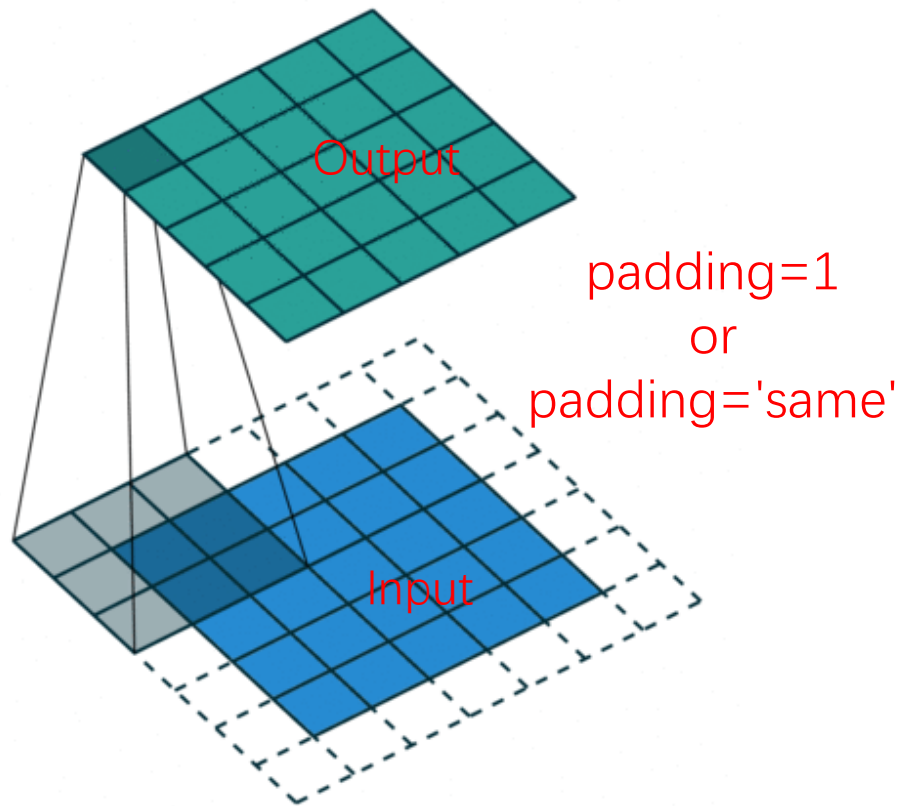
Convolution with Strides

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`



Convolution with Padding

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=1('same'), dilation=1, groups=1,  
bias=True, padding_mode='zeros', device=None, dtype=None)
```



Convolution with Padding (Cont'd)

`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=1('same'), dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`

padding_mode='zeros'

0	0	0	0	0	0
0	9	6	0	4	0
0	6	4	7	1	0
0	1	0	8	0	0
0	0	2	1	6	0
0	0	0	0	0	0

padding_mode='reflect'

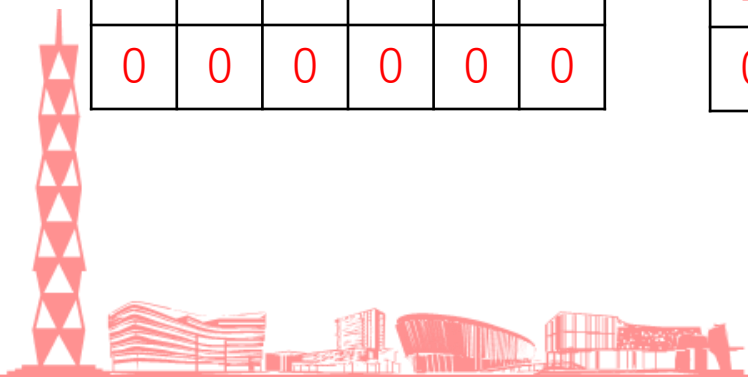
4	6	4	7	1	7
6	9	6	0	4	0
4	6	4	7	1	7
0	1	0	8	0	8
2	0	2	1	6	1
0	1	0	8	0	8

padding_mode='replicate'

9	9	6	0	4	4
9	9	6	0	4	4
6	6	4	7	1	1
1	1	0	8	0	0
0	0	2	1	6	6
0	0	2	1	6	6

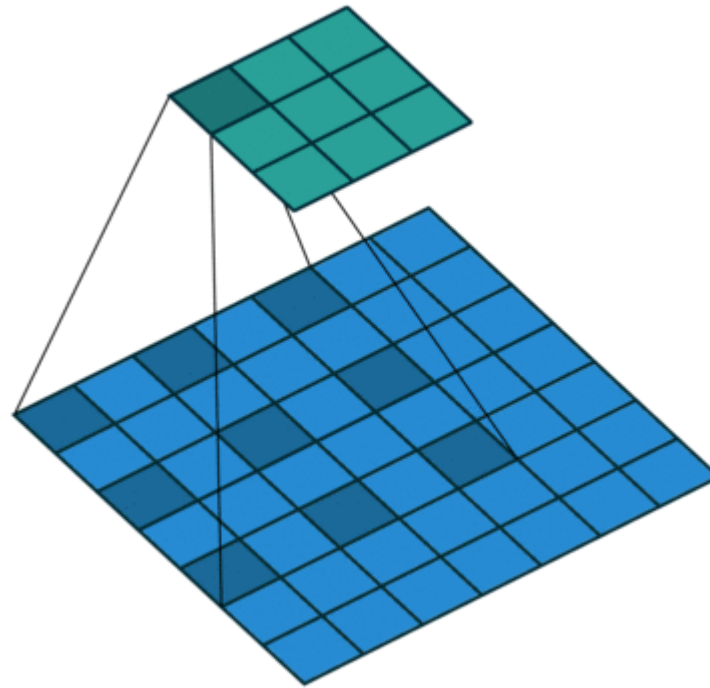
padding_mode='circular'

6	0	2	1	6	0
4	9	6	0	4	4
1	6	4	7	1	1
0	1	0	8	0	0
6	0	2	1	6	6
4	9	6	0	4	9



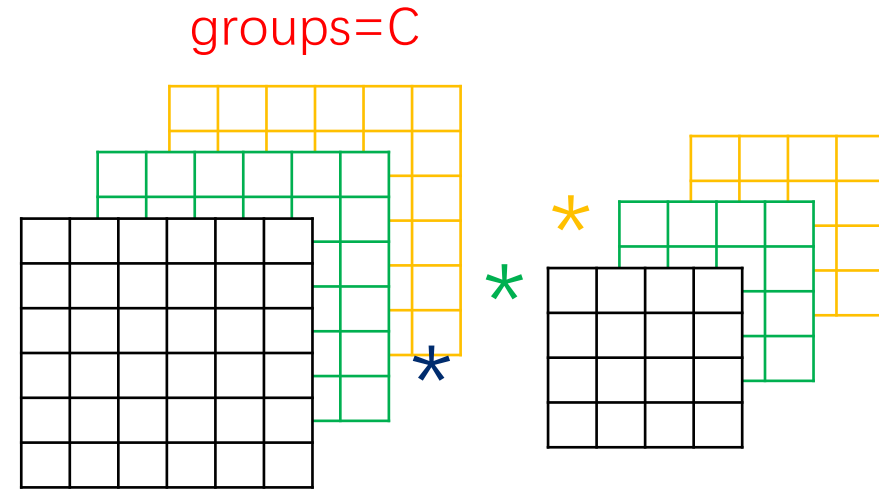
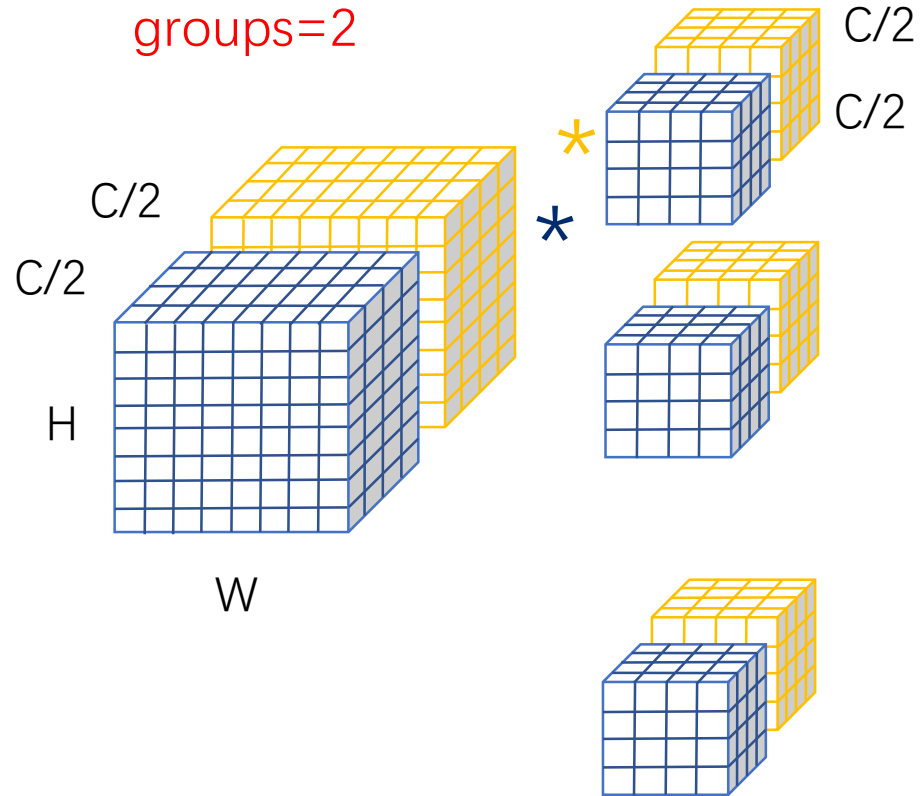
Convolution with Dilation

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=0, dilation=2, groups=1,  
bias=True, padding_mode='zeros', device=None, dtype=None)
```



Grouped Convolution

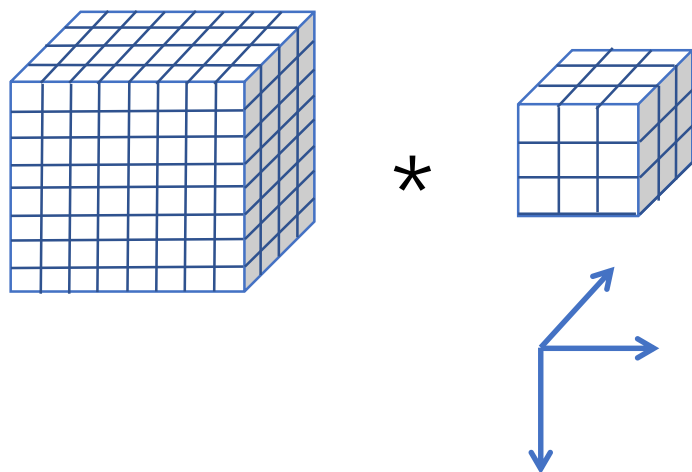
`torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=0, dilation=2, groups=N, bias=True, padding_mode='zeros', device=None, dtype=None)`



Depthwise separable layer,
widely used in light/efficient CNNs

3D Convolution

- Conv3d



The sizes of channel do not have to be the same for 3D convolution.



The Other Components in Convolution Layers

- Pooling
 - Usually used to reduce data
 - `torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)`
 - `torch.nn.quantized.functional.avg_pool2d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)`

9	6	0	4
6	4	7	1
1	0	8	0
0	2	1	6

2*2 maxpool
stride = 2

9	7
2	8

8	6	0	4
6	4	7	1
2	0	8	0
0	2	2	6

2*2 avgpool
stride = 2

7	3
1	4

