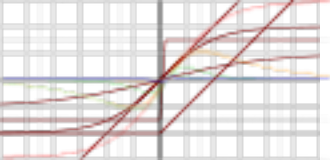
Neural Network Architectures

Neil Gong

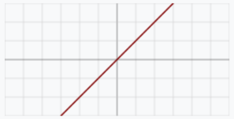
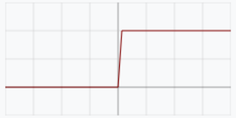
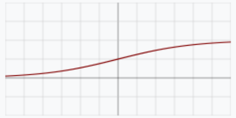
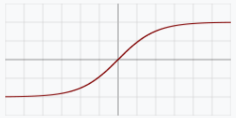
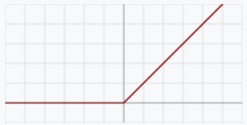
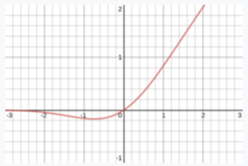
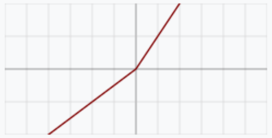
Slides adapted from Kaiming He and Sergey Levine

Artificial Neural Networks

- Input/output
- Weight
- Activation function
- Connection pattern



Activation function

Name	Plot	Function, $g(x)$
Identity		x
Binary step		$\begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$
Logistic, sigmoid, or soft step		$\sigma(x) \doteq \frac{1}{1+e^{-x}}$
Hyperbolic tangent (tanh)		$\tanh(x) \doteq \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Rectified linear unit (ReLU)		$(x)^+ \doteq \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} = \max(0, x) = x\mathbf{1}_{x>0}$
Gaussian Error Linear Unit (GELU)		$\frac{1}{2}x\left(1+\text{erf}\left(\frac{x}{\sqrt{2}}\right)\right) = x\Phi(x)$
Leaky rectified linear unit (Leaky ReLU)		$\begin{cases} 0.01x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$

Source: Wikipedia

Connection patterns

- Fully connected
- Softmax
- Convolution
- Residual
- Transformer

Convolution: a 2-D example

input

0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0

filter

1	**2**	**1**
0	**0**	**0**
-1	**-2**	**-1**

output

Convolution: a 2-D example

input

0 **1**	0 **2**	0 **1**	0	0	0	0	0
0 **0**	0 **0**	0 **0**	0	0	1	1	0
0 **-1**	1 **-2**	1 **-1**	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0

filter

1	2	1
0	0	0
-1	-2	-1

- sliding window
- dot product

output

-3					

Convolution: a 2-D example

input

0	0 **1**	0 **2**	0 **1**	0	0	0	0
0	0 **0**	0 **0**	0 **0**	0	1	1	0
0	1 **-1**	1 **-2**	1 **-1**	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0

filter

1	**2**	**1**
0	**0**	**0**
-1	**-2**	**-1**

output

-3	-4				

- sliding window
- dot product

Convolution: a 2-D example

input

0	0	0 **1**	0 **2**	0 **1**	0	0	0
0	0	0 **0**	0 **0**	0 **0**	1	1	0
0	1	1 **-1**	1 **-2**	1 **-1**	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	1	1	1	0	0	0
0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	0

filter

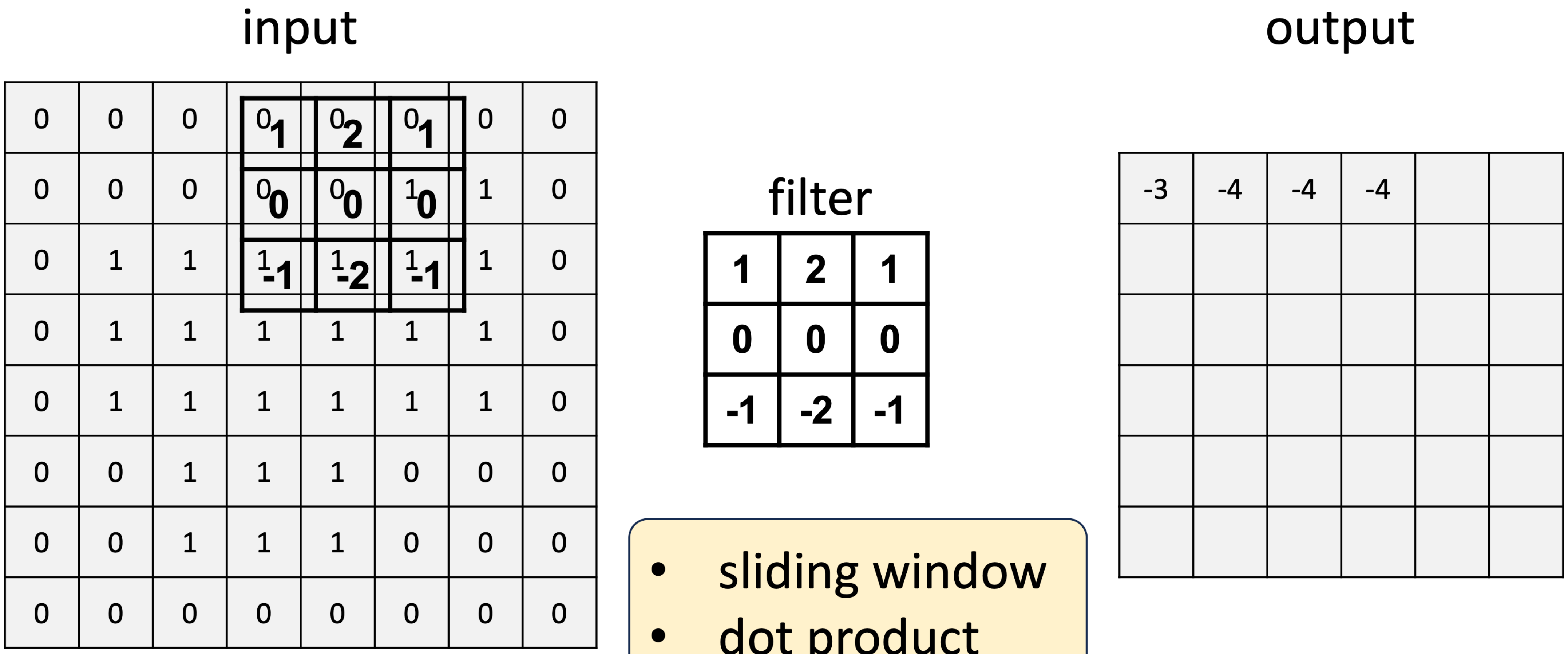
1	**2**	**1**
0	**0**	**0**
-1	**-2**	**-1**

- sliding window
- dot product

output

-3	-4	-4			

Convolution: a 2-D example



- sliding window
- dot product

Convolution: a 2-D example

input

0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	1	1	1	1	1	1	0
0	0	1	1	1	0 1	0 2	0 1
0	0	1	1	1	0 0	0 0	0 0
0	0	0	0	0	0 -1	0 -2	0 -1

filter

1	**2**	**1**
0	**0**	**0**
-1	**-2**	**-1**

- sliding window
- dot product

output

-3	-4	-4	-4	-4	-3
-3	-4	-4	-3	-1	0
0	0	0	0	0	0
2	1	0	1	3	3
2	1	0	1	3	3
1	3	4	3	1	0

Convolution: a 2-D example

input map

filter weights

coordinates in a local window

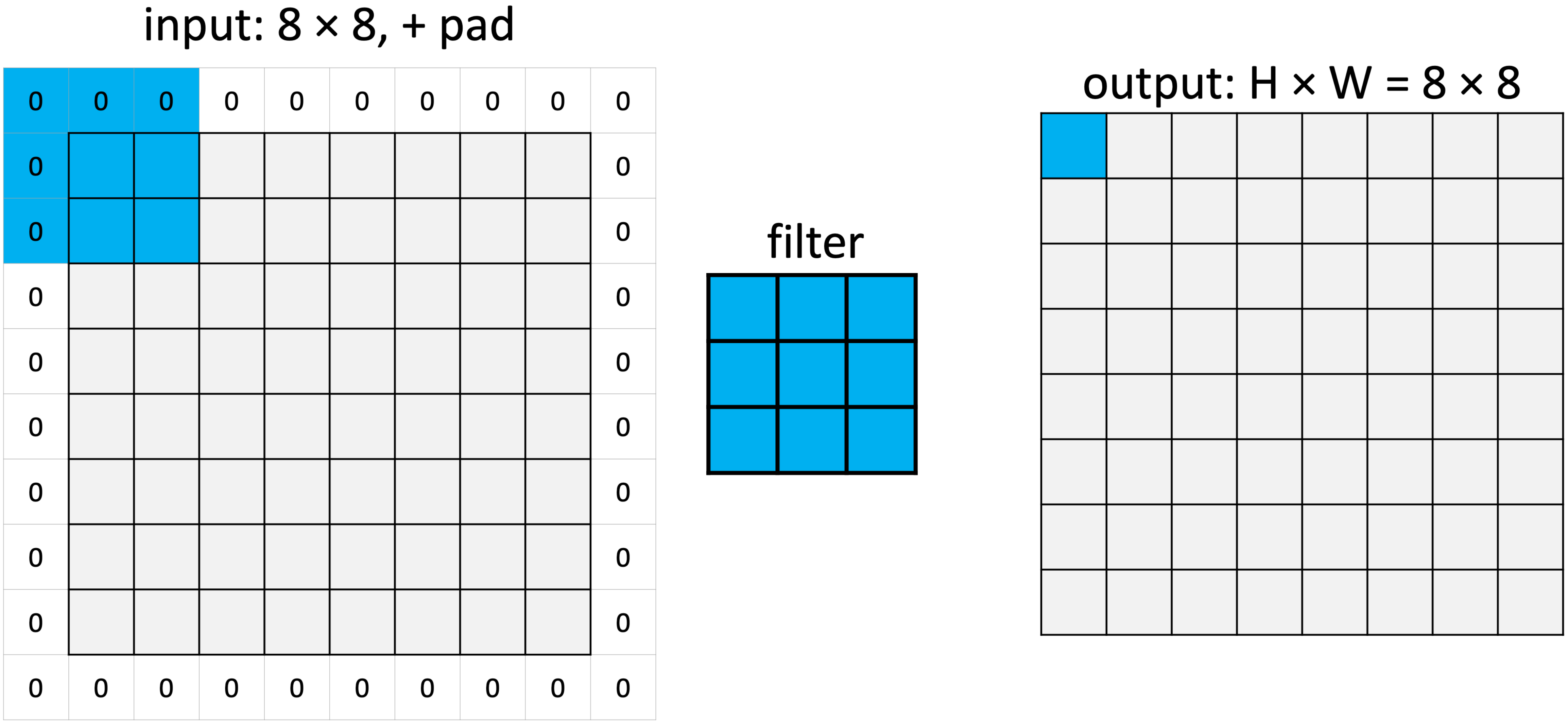
$$\boldsymbol{y}[n,m] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} \boldsymbol{w}[i,j]\,\boldsymbol{x}[n+i, m+j]$$

output map

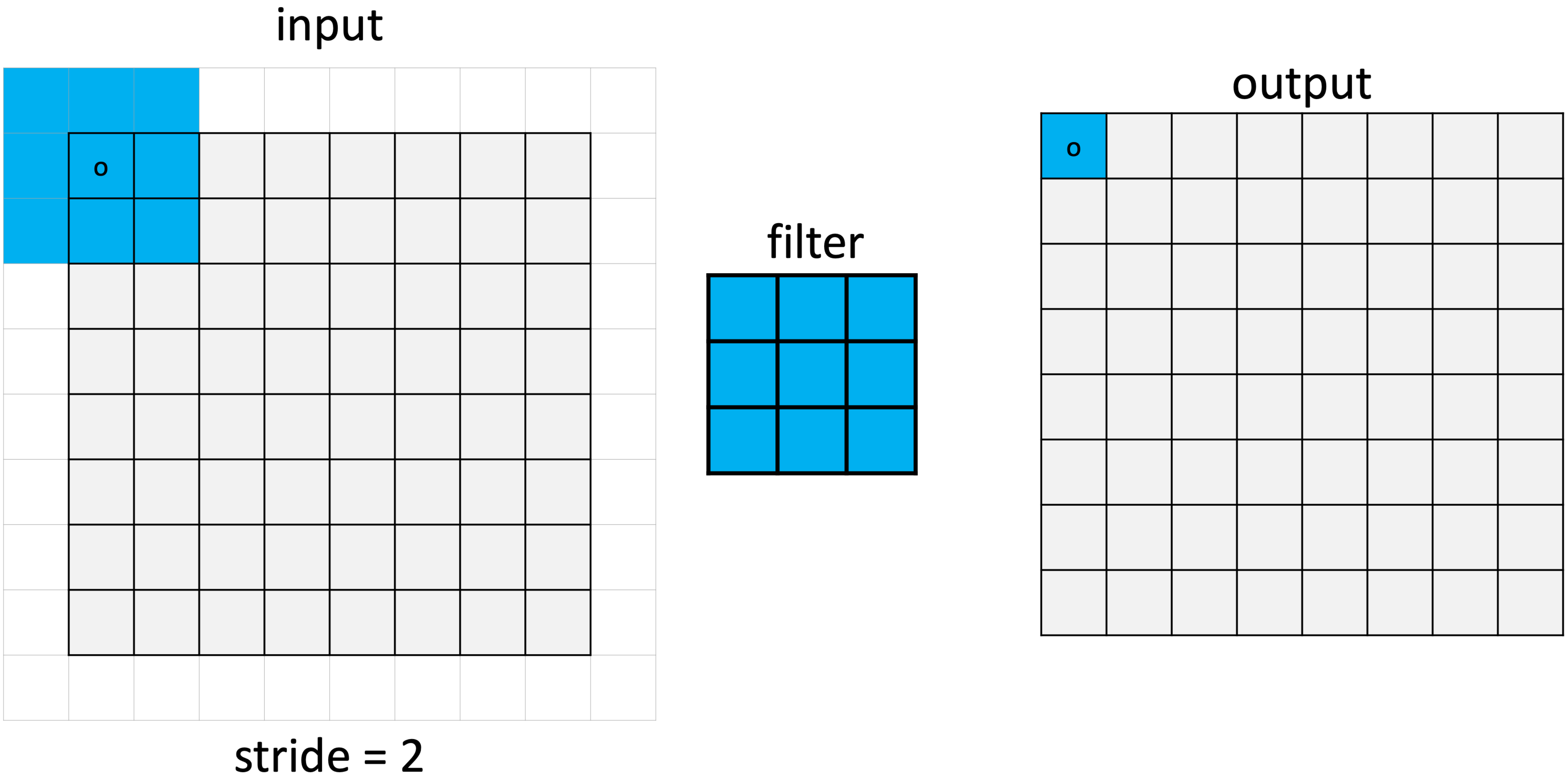
r: kernel radius

kernel size = $2r + 1$

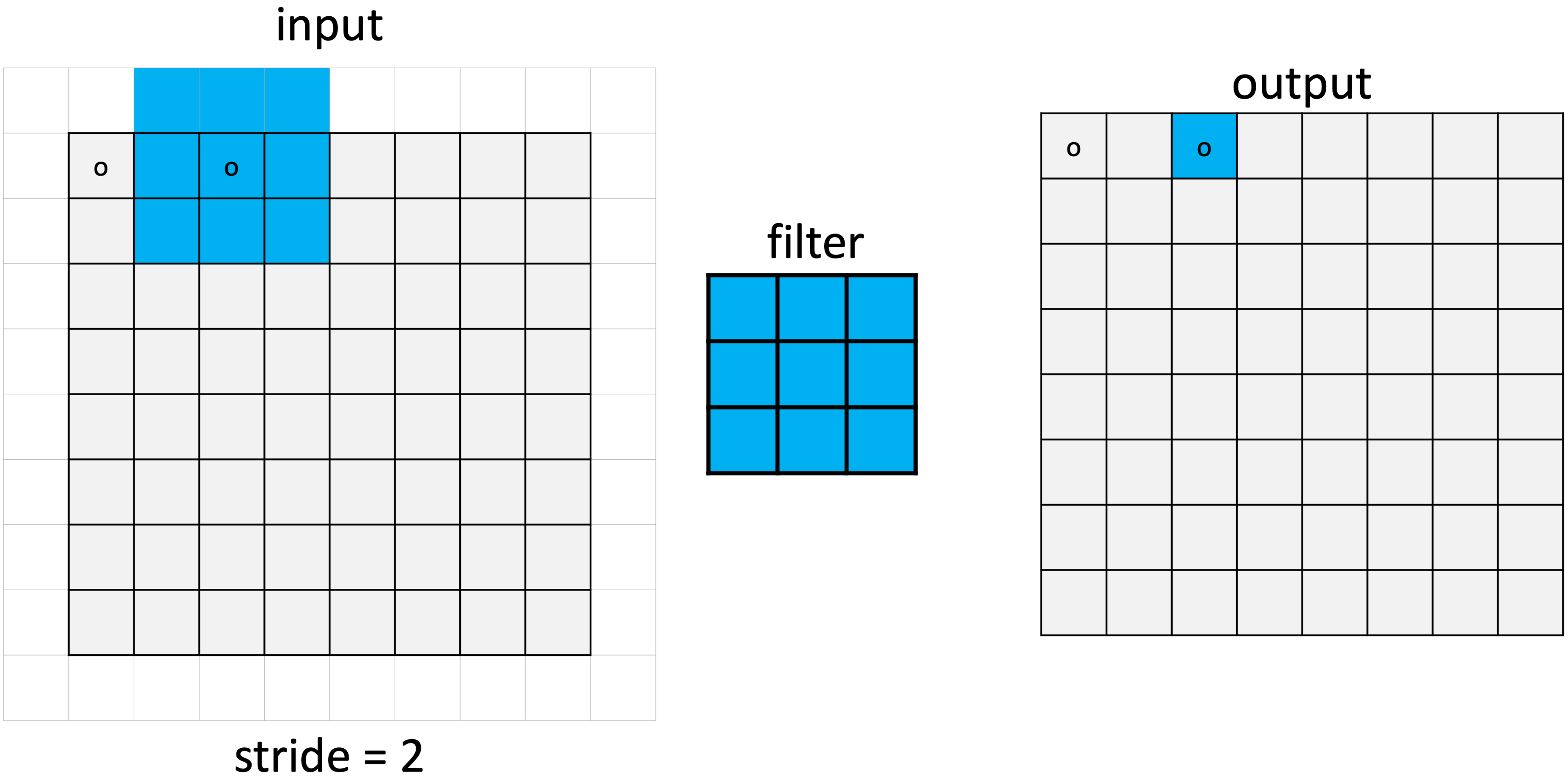
Convolution: padding



Convolution: stride

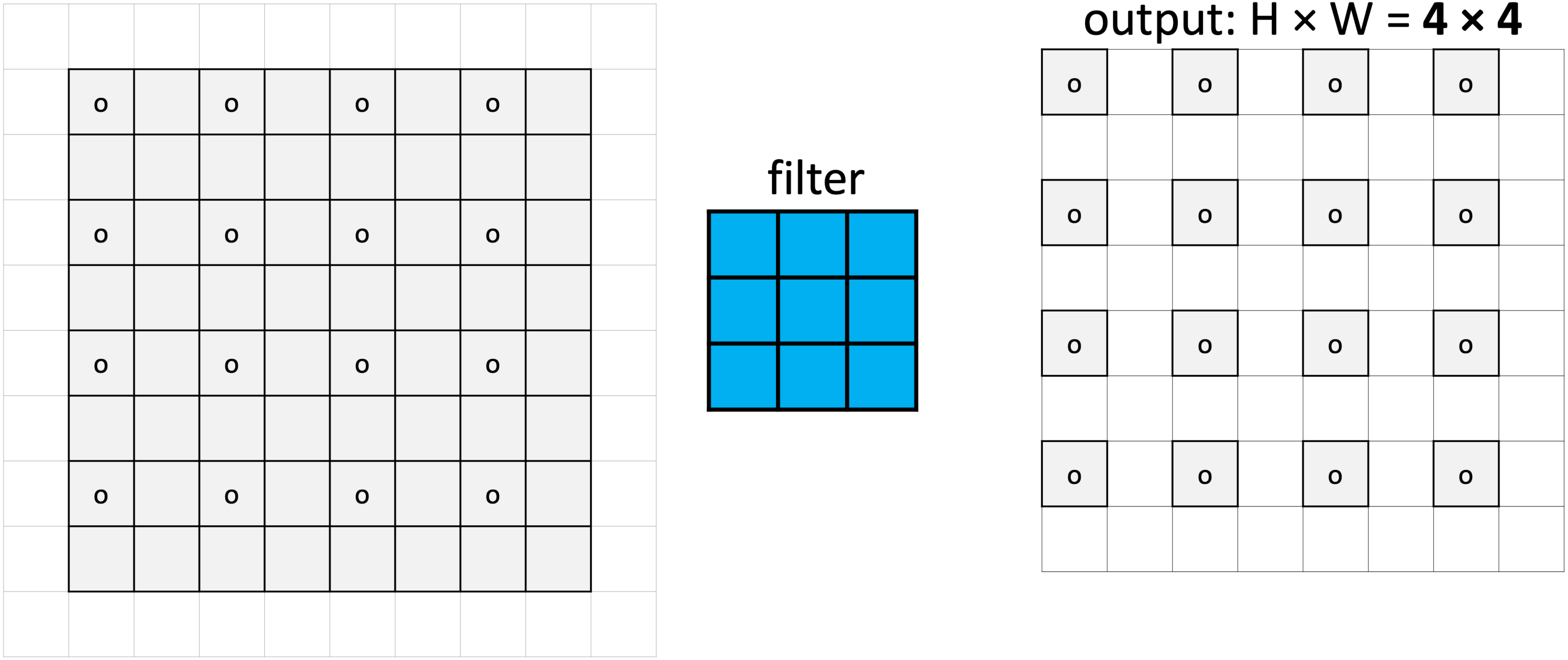


Convolution: stride



Convolution: stride

input: H × W = 8 × 8

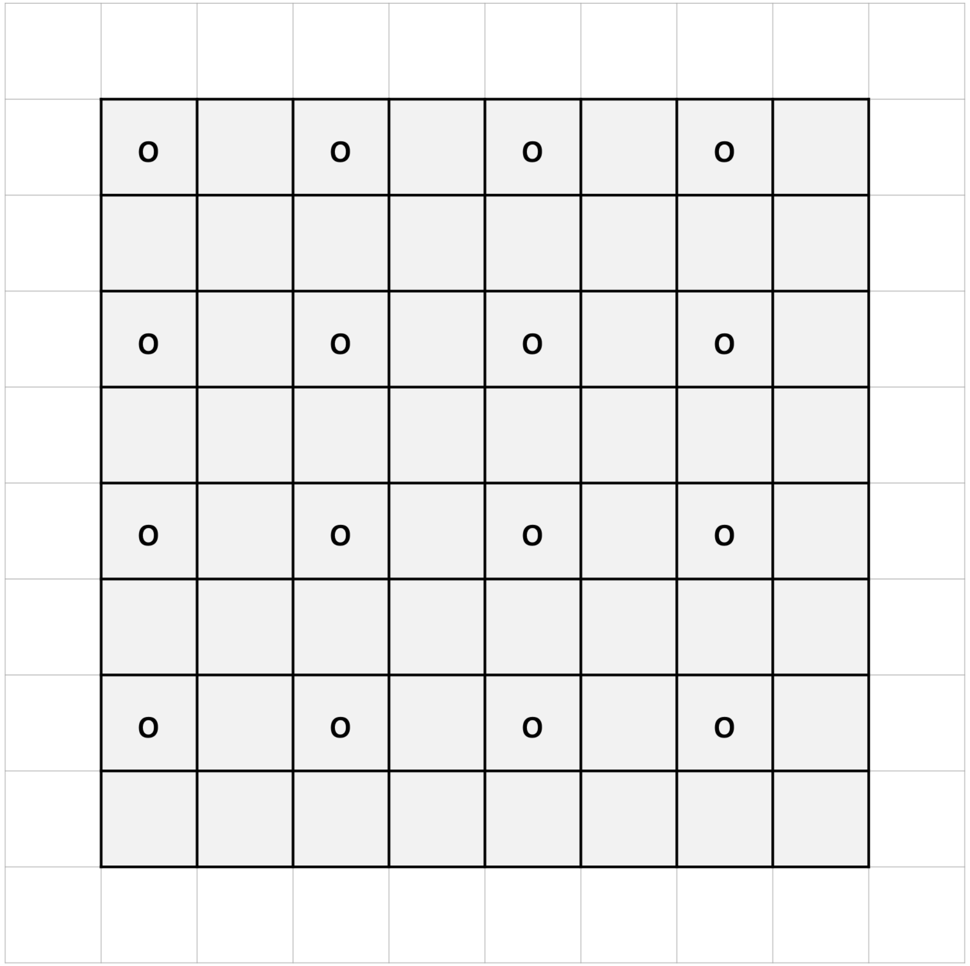


stride = 2

Convolution: stride

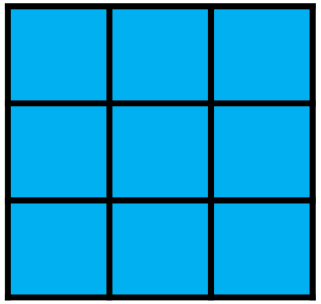
- reduces feature map size
- compress and abstract

input: H × W = 8 × 8

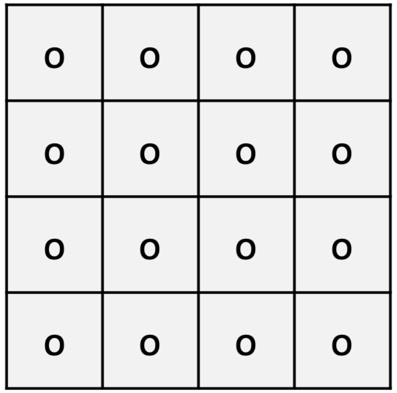


stride = 2

filter



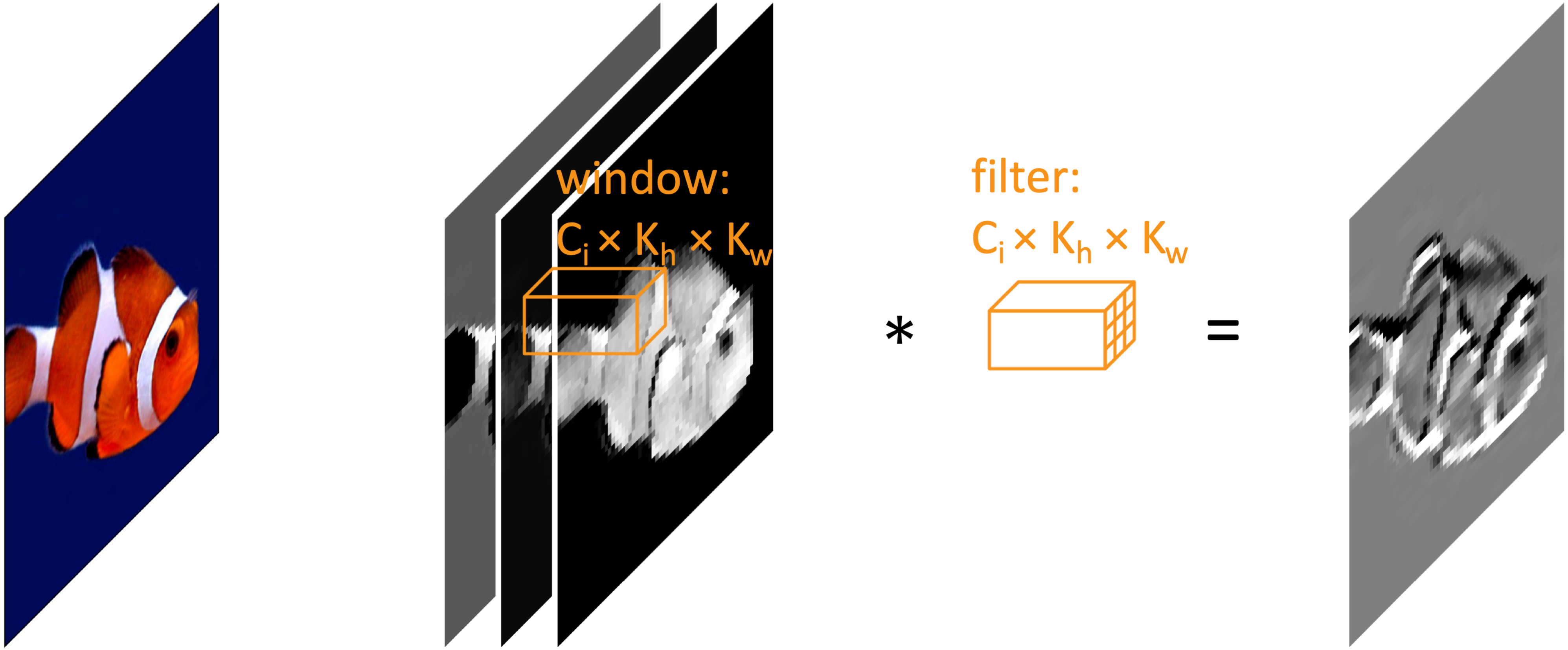
output: H × W = **4 × 4**



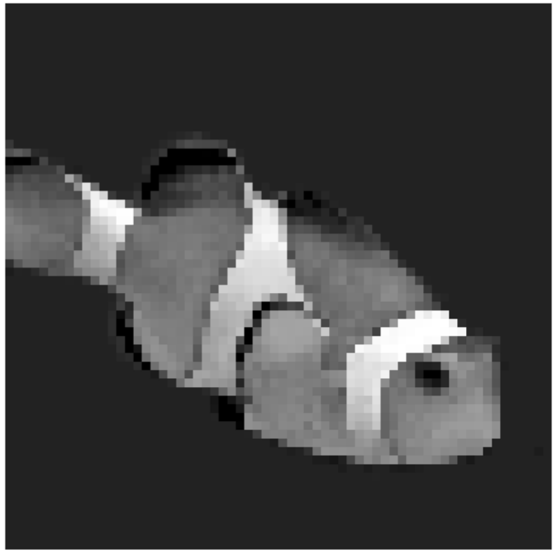
$$H_{out} = \lfloor (H_{in} + 2pad_h - K_h) / str \rfloor + 1$$

*rounding operation depends on libraries

Convolution: Multi-channel inputs



Convolution: Multi-channel outputs



*

1	2	1
0	0	0
-1	-2	-1

=

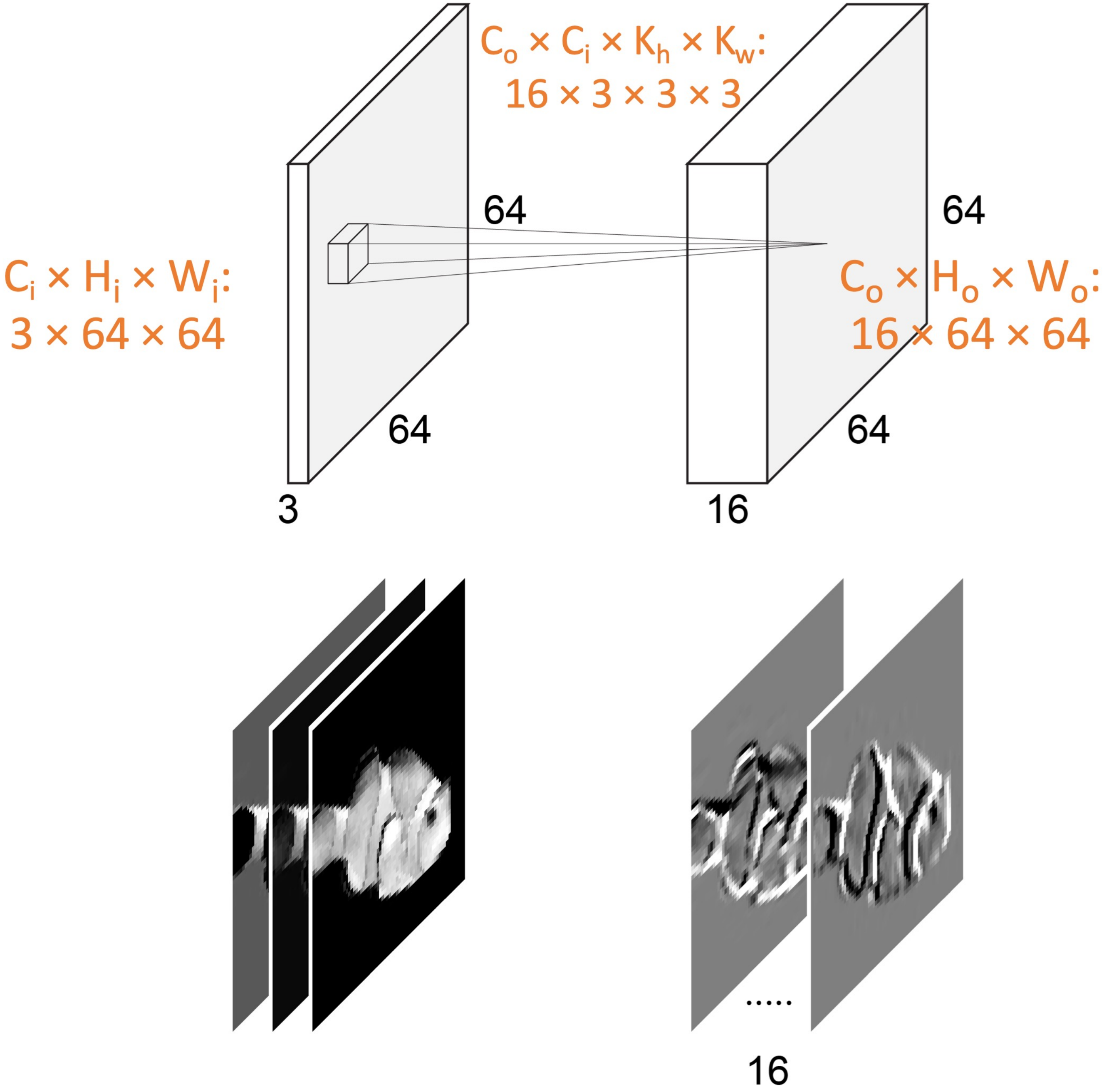
one filter, one feature

*

1	0	-1
2	0	-2
1	0	-1

=

Convolution: tensor views

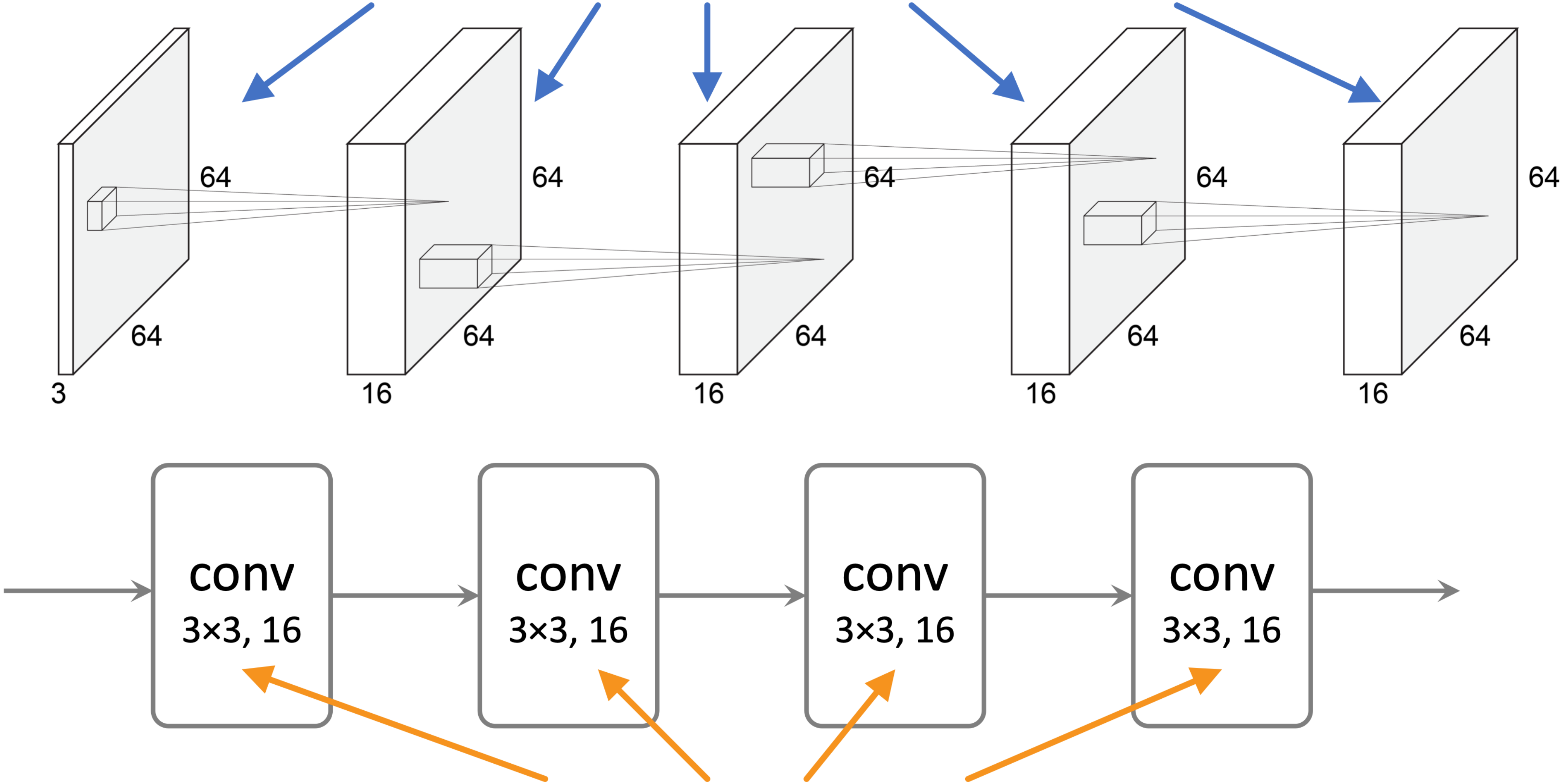


- Tensor: high-dimension array
- feature maps
  - 3-D tensor: $C \times H \times W$
  - C: channels
  - H: height
  - W: width
- filters
  - 4-D tensor: $C_o \times C_i \times K_h \times K_w$
  - C_o: output channels
  - C_i: input channels
  - K_h, K_w: filter height, width

Composing basic operations

two ways of showing neural nets

these are activations (features, embeddings, tensors …)



these are operations (functions, transforms, layers …)

Deep Residual Learning

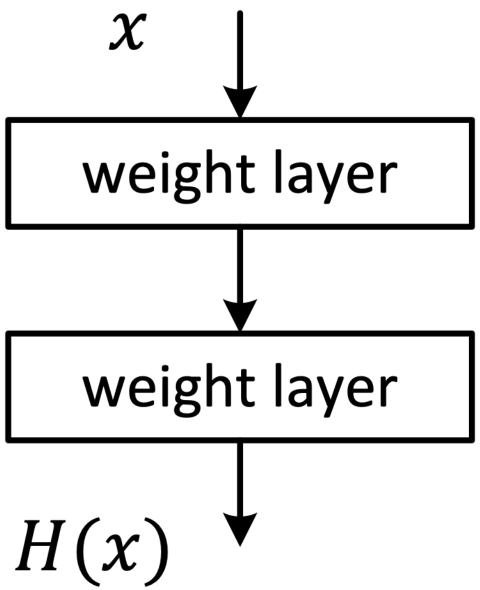
- Deep Learning gets way deeper
- simple component: identity shortcut
- enable networks w/ hundreds of layers

Compose simple modules into complex functions

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015, CVPR 2016.

Deep Residual Learning

a subnet in
a deep net



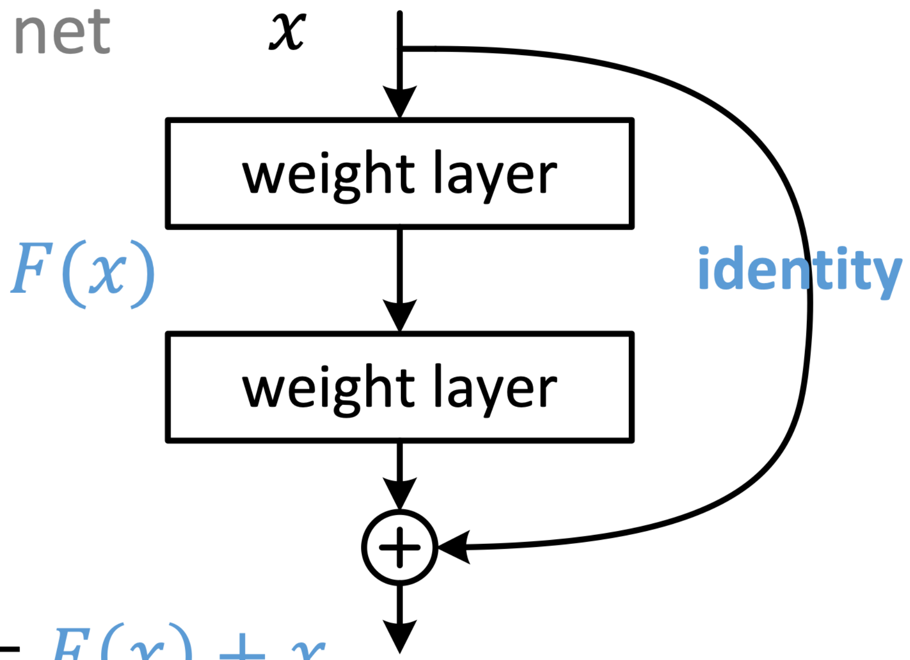
classical network

- $H(x)$: desired function to be fit by a subnet
- let weight layers fit $H(x)$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015, CVPR 2016.

Deep Residual Learning

a subnet in
a deep net



$H(x) = F(x) + x$

residual block

- $H(x)$: desired function to be fit by a subnet

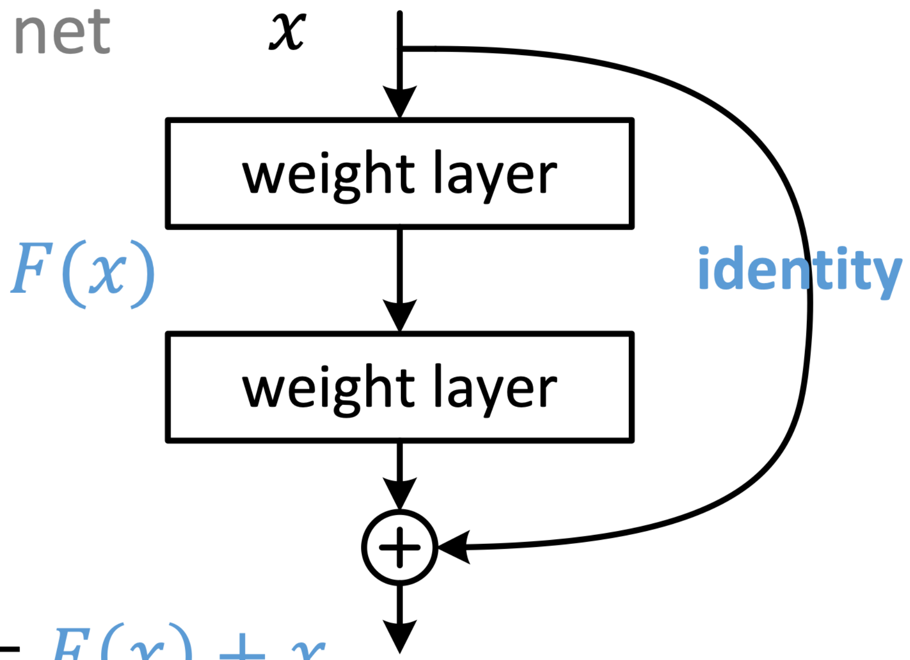
- ~~let weight layers fit $H(x)$~~
- let weight layers fit $F(x)$

- set $H(x) = F(x) + x$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015, CVPR 2016.

Deep Residual Learning

a subnet in
a deep net



$$H(x) = F(x) + x$$

residual block

- $F(x)$: residual function
- if $H(x)$ = identity is near-optimal
  - push weights to small
  - encourage small changes
- initialization
  - small or zero weights

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015, CVPR 2016.

Residual Networks (ResNet)

Building very deep nets:

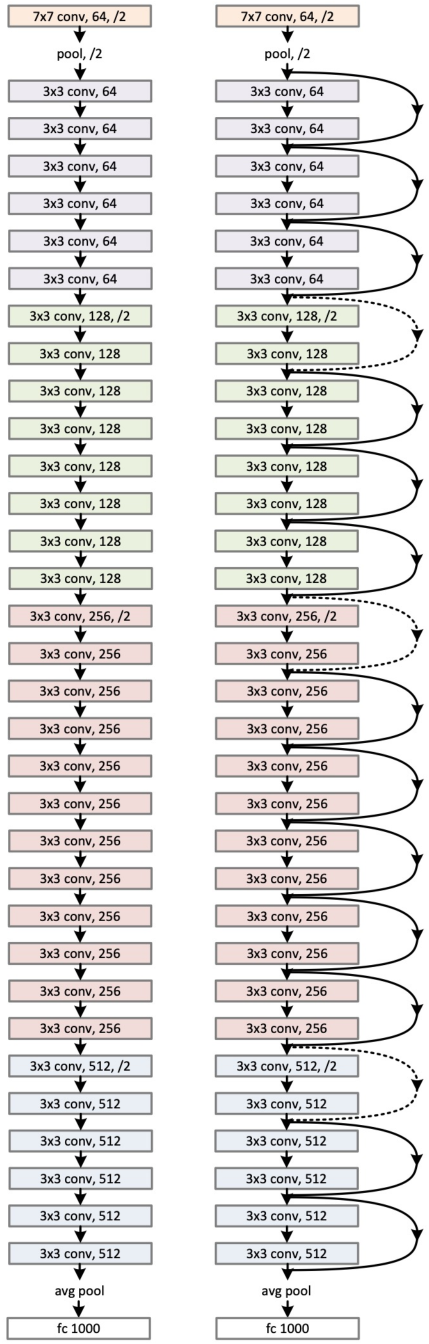
- add **identity connections** to vanilla nets
  (a.k.a. skip/shortcut/residual connections)

or:

- stack many **residual blocks**

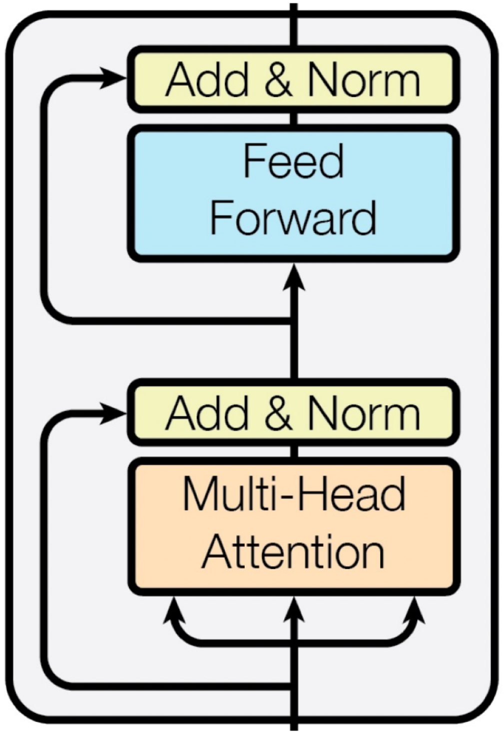
Residual Blocks:

- new generic modules for neural nets
- design blocks and compose them



Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015, CVPR 2016.

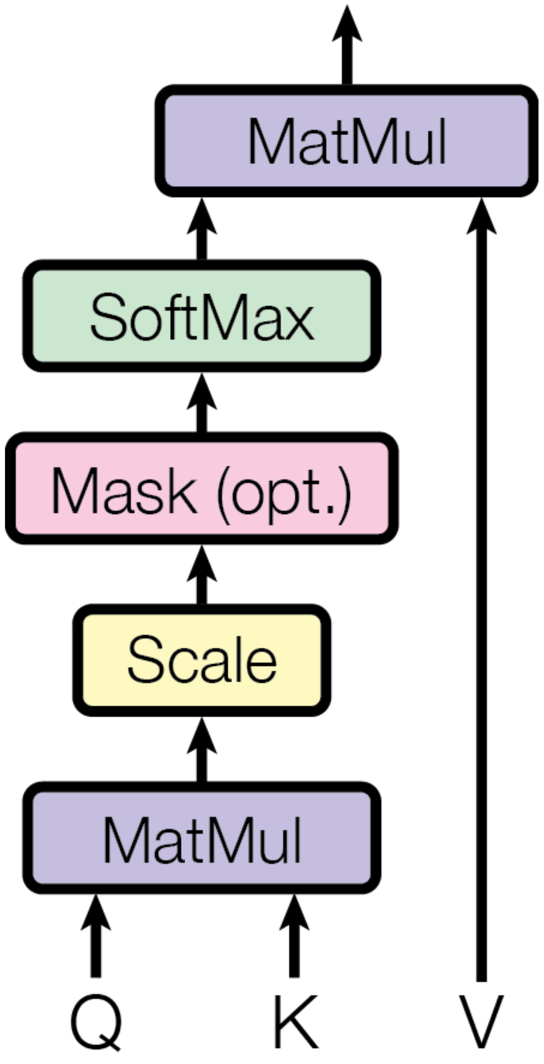
Residual Block: Transformer



A Transformer Block has two Residual Blocks.

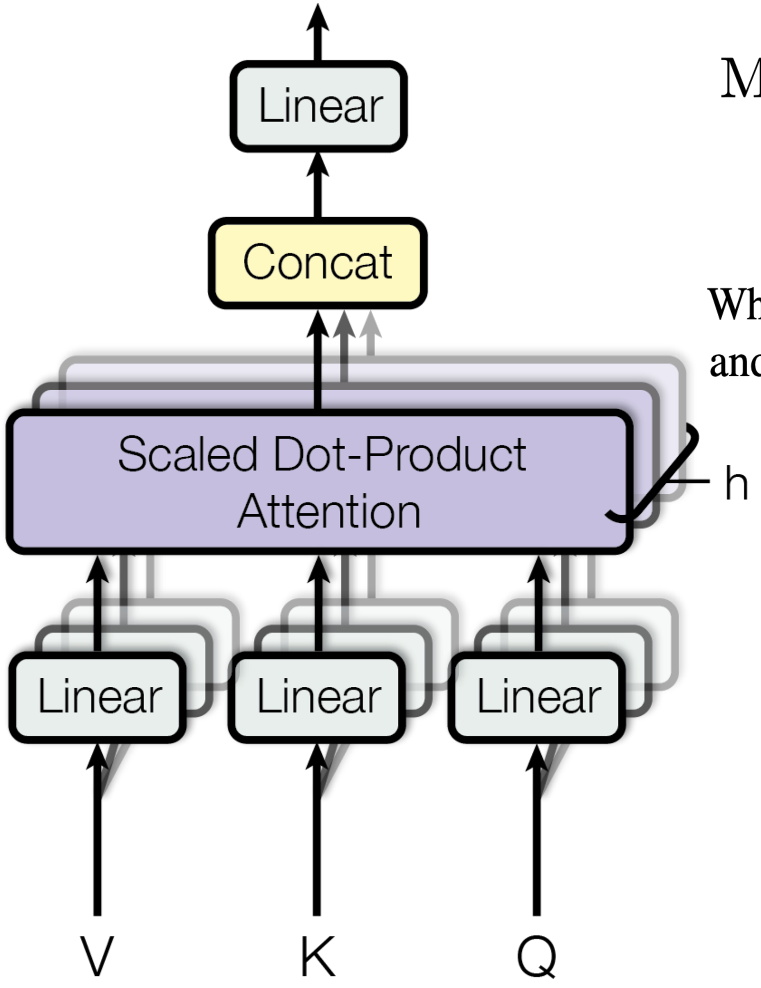
"Attention is all you need", Vaswani, et al., 2017

Scaled Dot-Product Attention



$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention



$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(\mathrm{head_1},...,\mathrm{head_h})W^O$$
$$\text{where } \mathrm{head_i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$.

Position-wise feed-forward network

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

One last detail: layer normalization

Main idea: batch normalization is very helpful, but hard to use with sequence models

Sequences are different lengths, makes normalizing across the batch hard

Sequences can be very long, so we sometimes have small batches

Simple solution: "layer normalization" – like batch norm, but not across the batch

Batch norm

$a_1, a_2, \ldots, a_B$ ← d-dimensional vectors for each sample in batch

d-dim → μ

$$\mu = \frac{1}{B}\sum_{i=1}^{B} a_i \qquad \sigma = \sqrt{\frac{1}{B}\sum_{i=1}^{B}(a_i - \mu)^2}$$

$$\bar{a}_i = \frac{a_i - \mu}{\sigma}\gamma + \beta$$

Layer norm

a

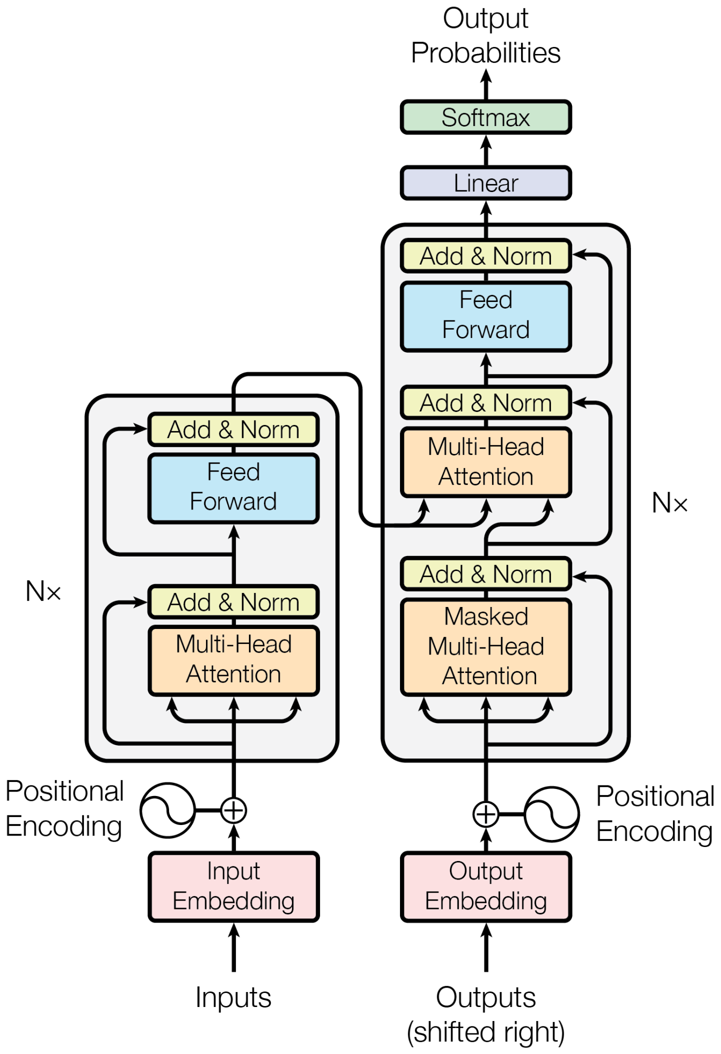
different *dimensions* of a → a_j

1-dim → μ

$$\mu = \frac{1}{d}\sum_{i=1}^{d} a_j \qquad \sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(a_j - \mu)^2}$$

$$\bar{a} = \frac{a - \mu}{\sigma}\gamma + \beta$$

Transformer architecture



Positional encoding: sin/cos

Naïve positional encoding: just append t to the input $\bar{x}_t = \begin{bmatrix} x_t \\ t \end{bmatrix}$

This is not a great idea, because **absolute** position is less important than **relative** position

I walk my dog every day

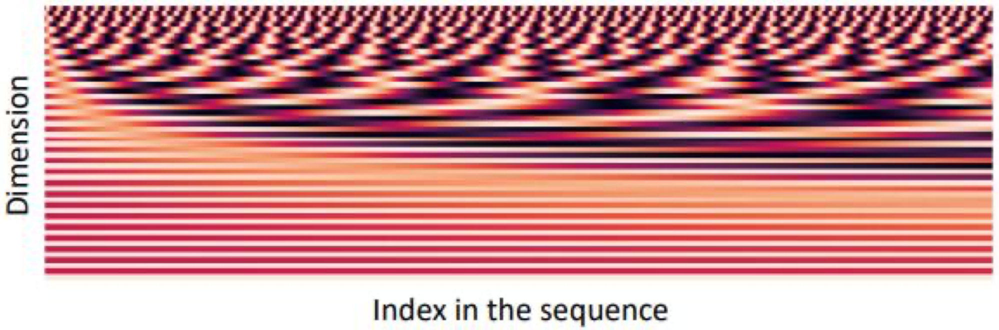
every single day I walk my dog

The fact that "my dog" is right after "I walk" is the important part, not its absolute position

we want to represent **position** in a way that tokens with similar **relative** position have similar **positional encoding**

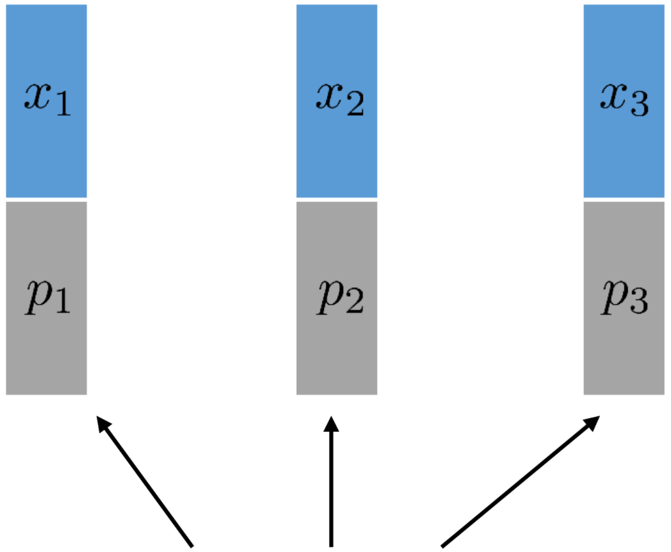
$$p_t = \begin{bmatrix} \sin(t/10000^{2*1/d}) \\ \cos(t/10000^{2*1/d}) \\ \sin(t/10000^{2*2/d}) \\ \cos(t/10000^{2*2/d}) \\ \dots \\ \sin(t/10000^{2*\frac{d}{2}/d}) \\ \cos(t/10000^{2*\frac{d}{2}/d}) \end{bmatrix}$$

dimensionality of positional encoding



Positional encoding: learned

Another idea: just learn a positional encoding



Different for every input sequence

The same learned values for every sequence

but different for different time steps

dimensionality

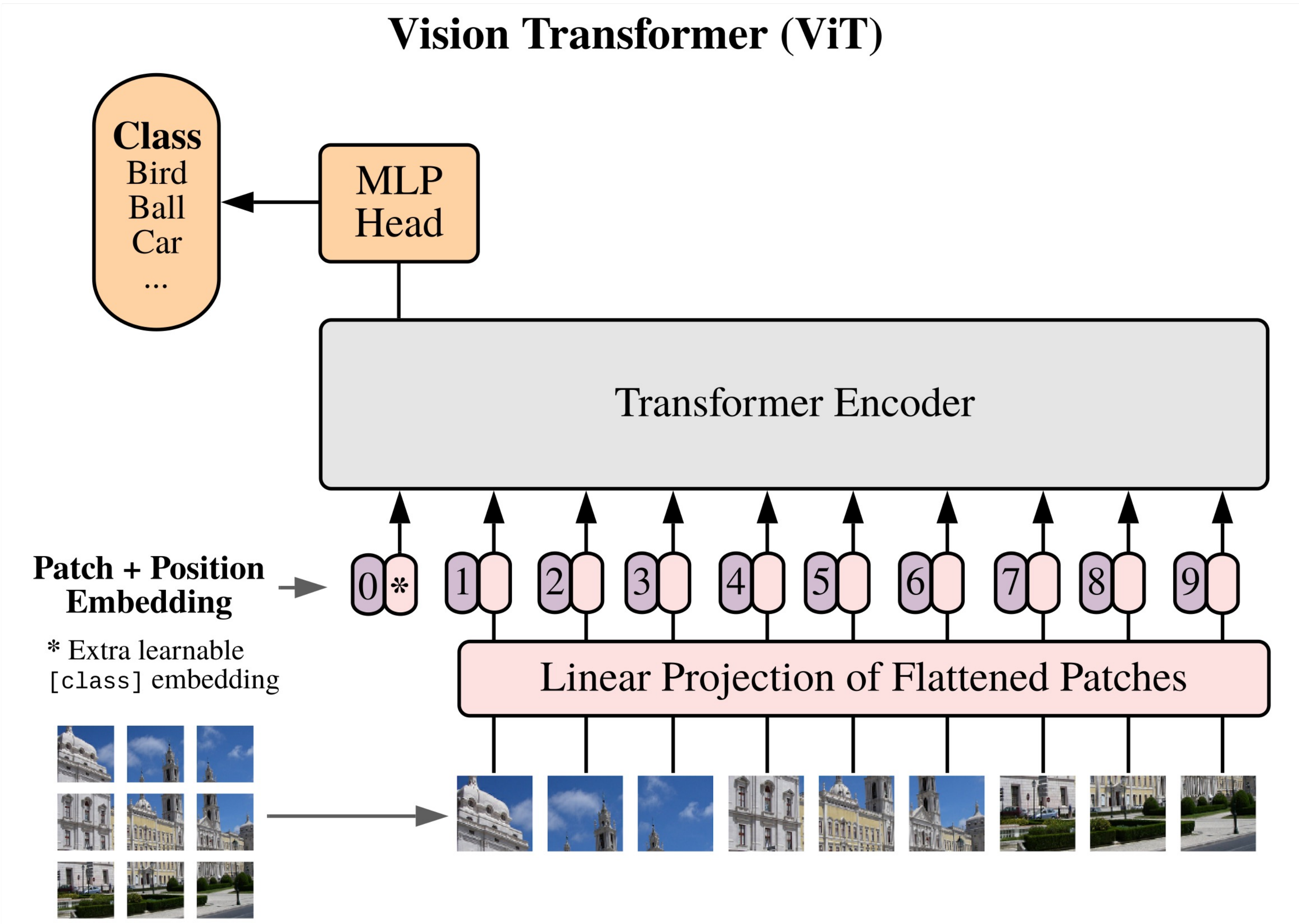
max sequence length

How many values do we need to learn?

$$P = [p_1, p_2, \ldots, p_T] \in R^{d \times T}$$

+ more flexible (and perhaps more optimal) than sin/cos encoding

+ a bit more complex, need to pick a max sequence length (and can’t generalize beyond it)



Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.” ICLR 2021.