

Convolutional networks



10-701 Introduction to Machine Learning
Geoff Gordon and Pradeep Ravikumar

Nonlinearity

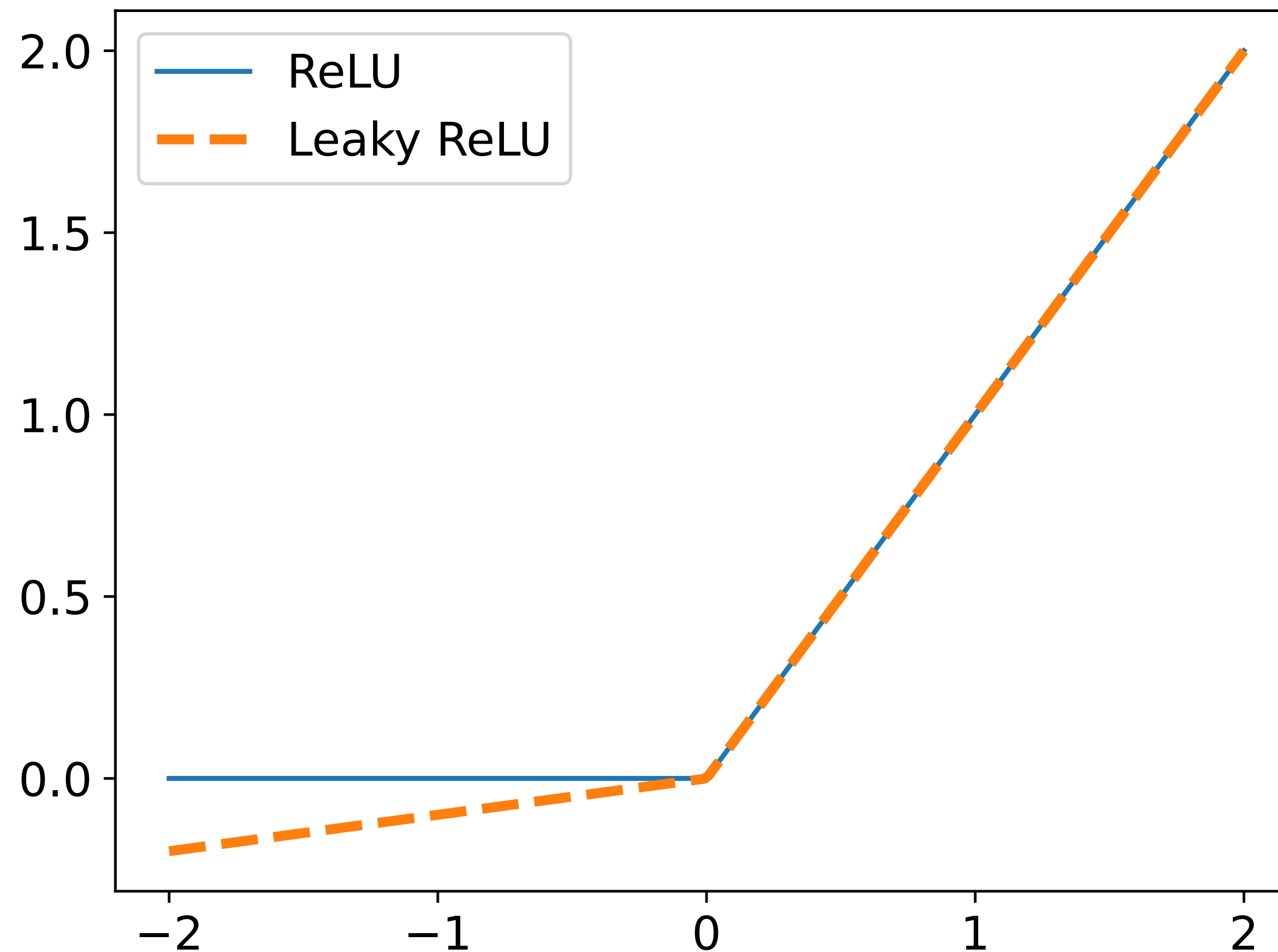
$$\text{softmax}(z_1, z_2) = \begin{pmatrix} \sigma(z_2 - z_1) \\ \sigma(z_1 - z_2) \end{pmatrix}$$

- So far we've seen $\sigma(\cdot)$, $[\cdot]_+$
 - ▶ variant of σ : softmax, $z \mapsto e^z / \sum e^z$, $\mathbb{R}^d \rightarrow \Delta$
 - ▶ should really be “soft argmax,” but who am I to argue
- Many more choices:
 - ▶ leaky ReLU, SiLU, GELU, gating, normalization, ...

Dead zones

- Gradient of ReLU $[z]_+$ is 0 for $z < 0$
- OK if this is true for some examples
- If it's true for all examples, we're stuck: SGD has no gradient signal to help fix it
 - ▶ unit contributes nothing to learned function
 - ▶ might get lucky and SGD noise makes it wander back into life via random changes to input
 - ▶ else, a waste of GPU time and memory

Leaky ReLU



- ReLU can be written $\max(0, z)$
- Leaky ReLU: $\max(\alpha z, z)$ for parameter $\alpha > 0$
 - ▶ now we have nonzero gradient even when $z < 0$
- Can fix α or learn it by SGD

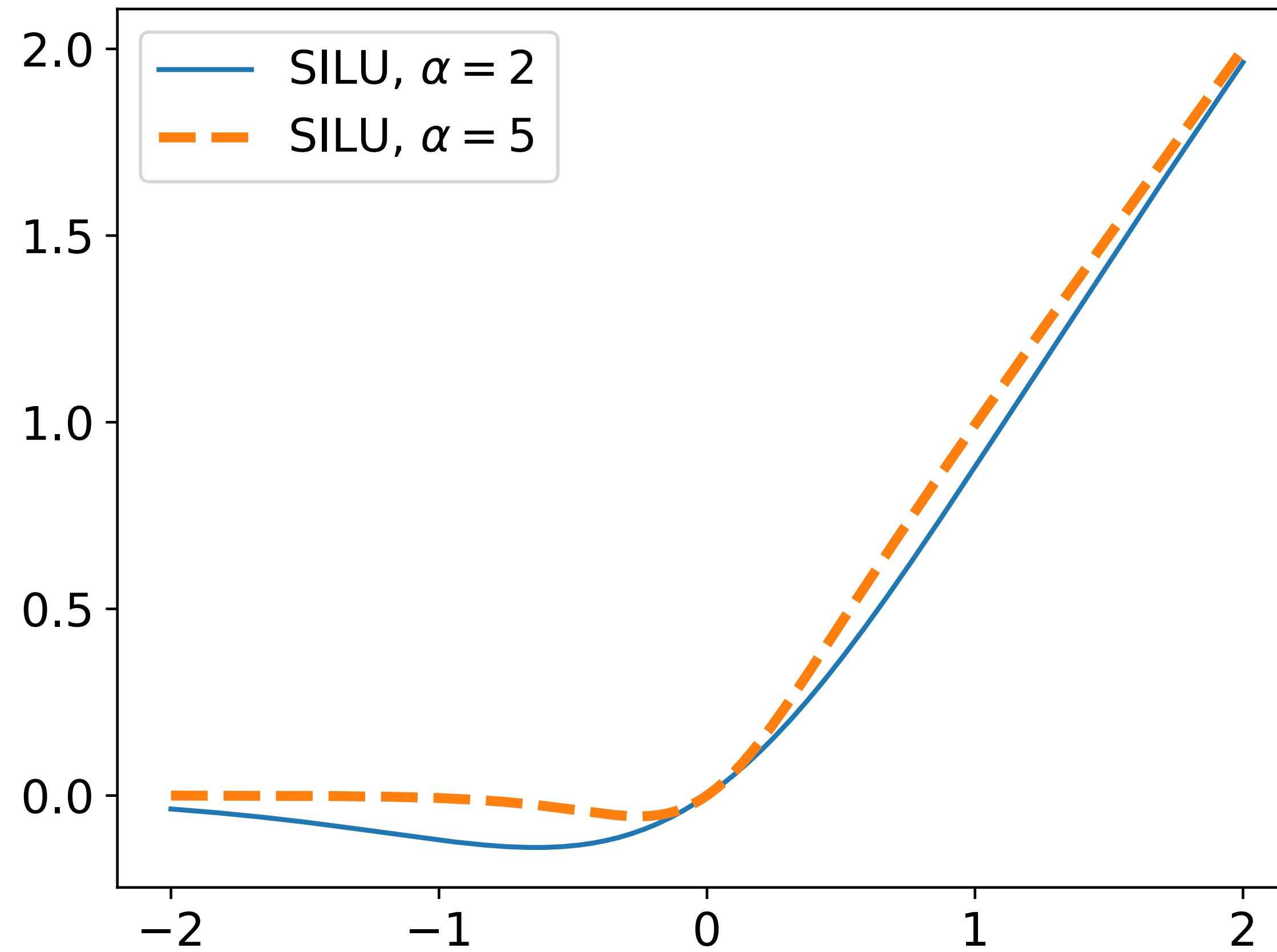
$\alpha = 0$ yields original ReLU

$\alpha < 1$

Large negative values

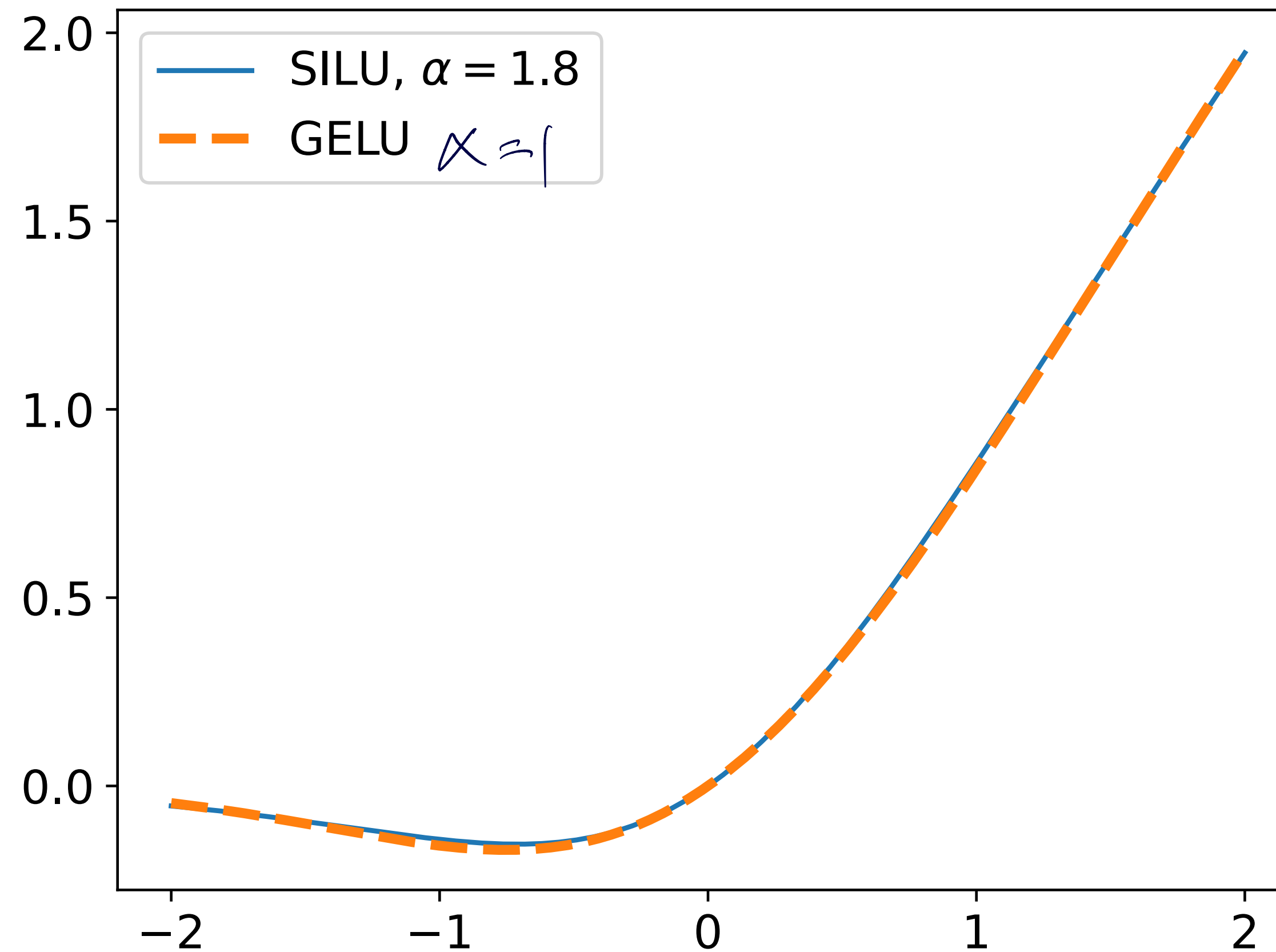
- Part of ReLU's benefit is that it zeros out large negative values of activation z
- Leaky ReLU doesn't do this: if $z \ll 0$, output can still have large absolute value
- Fixes: SILU, GELU
 - ▶ give up monotonicity to recover suppression of $z \ll 0$
 - ▶ can't have both

SILU



- Sigmoid linear unit = $z \sigma(\alpha z)$ for $\alpha > 0$
 - ▶ also called Swish
 - ▶ approaches ReLU for $\alpha \rightarrow \infty$

GELU



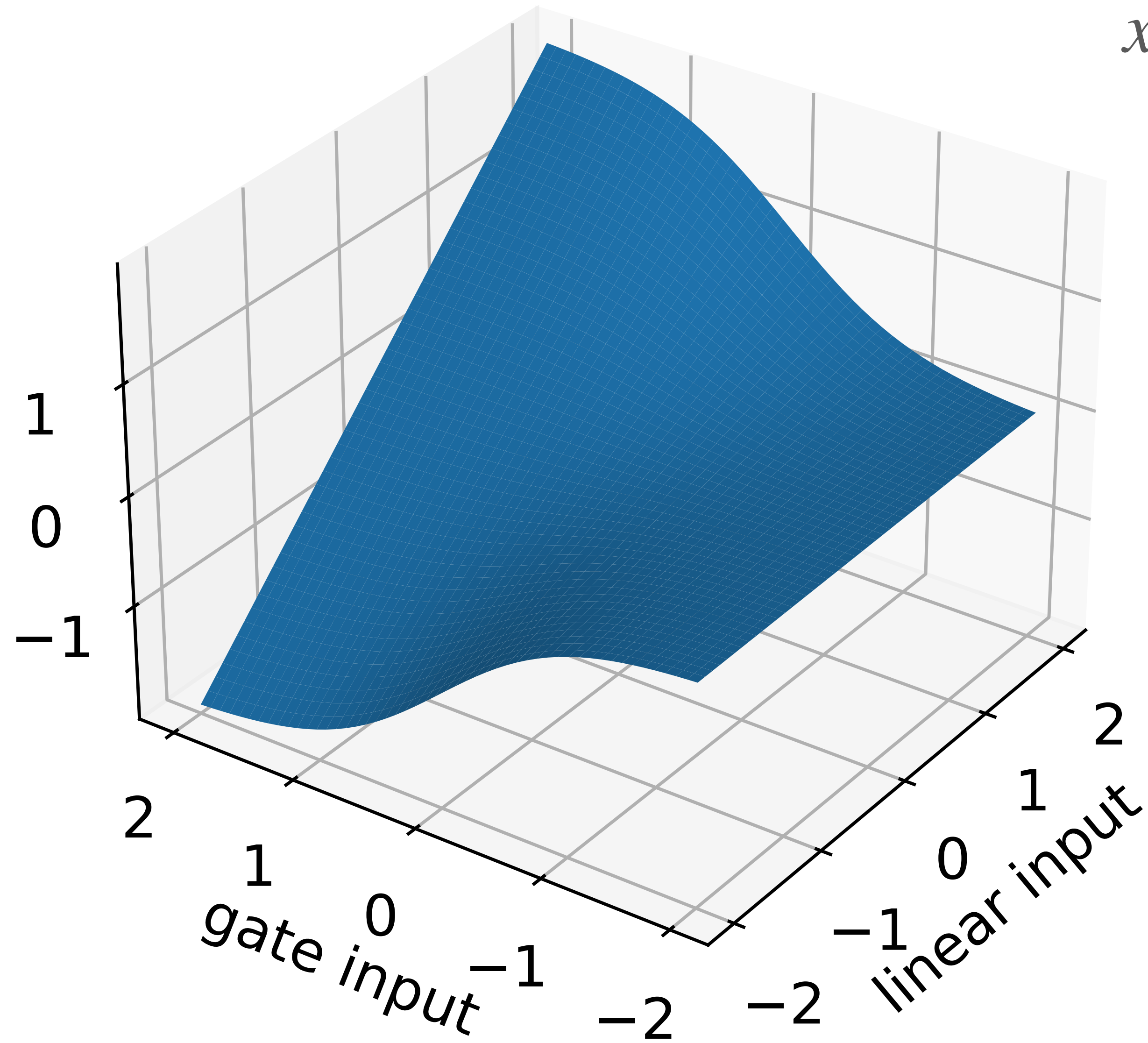
- Can do something like SILU with any CDF
- Popular: Gaussian CDF \rightarrow GELU
 - ▶ Gaussian error linear unit, since the Gaussian CDF is related to the *error function* $\text{erf}(z)$

Gating

- Can think of ReLU, SiLU, GELU as implementing a logical test (a *gate*) on activation $z = Wx + b$
 - ▶ if $z < 0$, suppress output (possibly differentiably)
- What if we want the gate to depend on something other than sign of z ?
- Gated linear unit (GLU, aka multiplicative integration):
 - ▶ $(Ux + b) \circ \sigma(Wx + c)$
 - ▶ two separate linear functions of previous layer, second one gates the first
 - ▶ can replace σ with Gaussian CDF or anything else (ReLU, identity, ...)
- Twice as many parameters (matters for speed, memory)

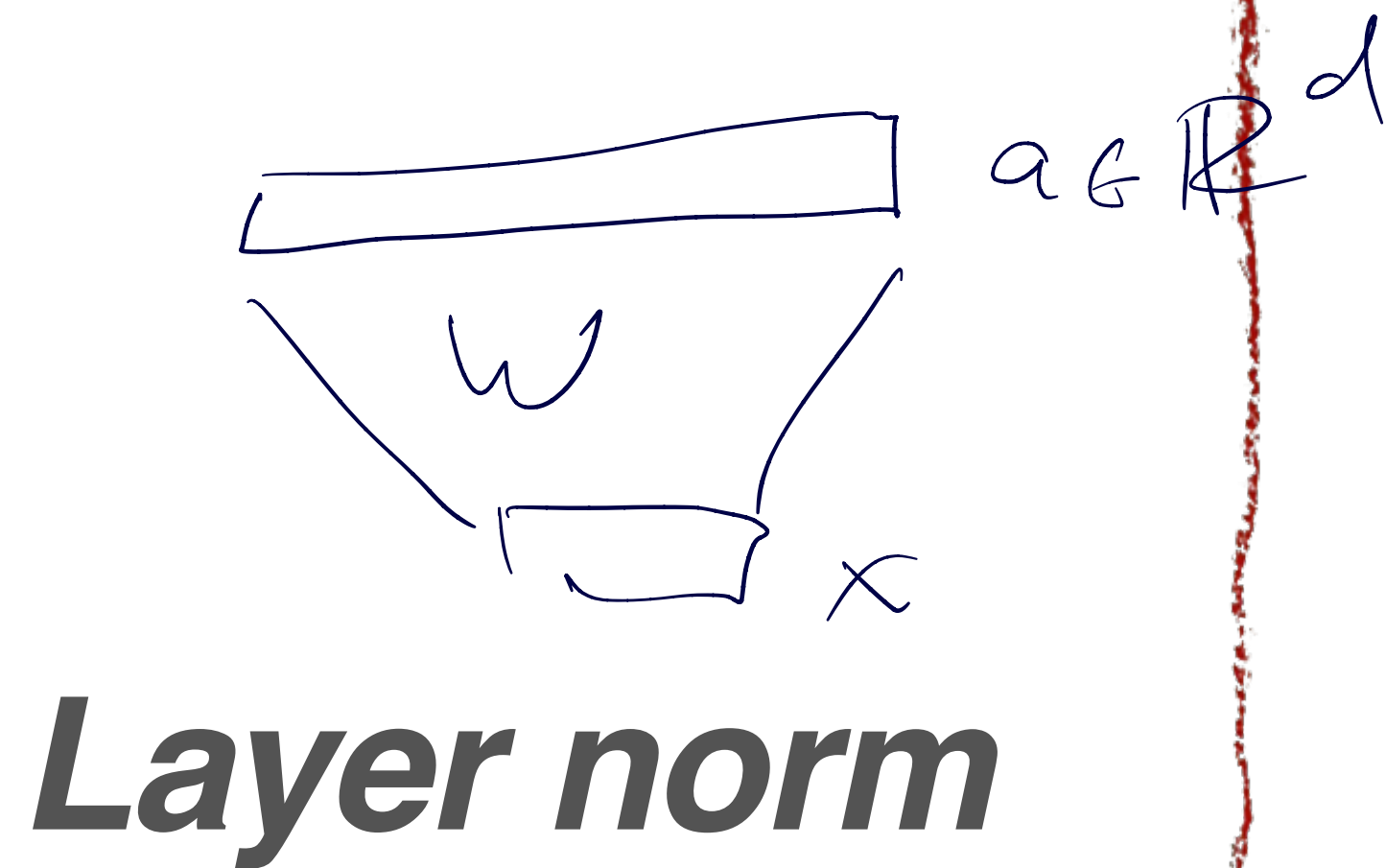
GLU

$$x_2 \circ \sigma(2x_1)$$



Normalize

- Typical example layer: $f(Wx + b)$: matrix W , vector x , b , componentwise nonlinear activation f
- Many activations are most interesting near zero
 - ▶ sigmoid's non-constant region
 - ▶ ReLU's derivative discontinuity
 - ▶ ...
- Idea: auto-shift and auto-scale inputs to be in this neighborhood by default
 - ▶ but let the network learn to move away if needed
- Hope: normalization helps generalization
 - ▶ in practice: effects poorly understood, but can be positive



no need for bias weight
or 1s column

tiny fixed number,
to prevent divide-
by-zero

- Given $a = Wx \in \mathbb{R}^d$
 - ▶ define $\bar{a} = \frac{1}{d} \sum_{i=1}^d a_i$
 - ▶ define $\sigma^2 = \frac{1}{d} \sum_{i=1}^d (a_i - \bar{a})^2 + \epsilon$
 - ▶ define $a' = \frac{a - \bar{a}}{\sigma}$
- New version of layer, with layer norm: $f(g \circ a' + b)$
 - ▶ new componentwise *gain* parameter $g \in \mathbb{R}^d$ picks scale
 - ▶ existing bias parameter b lets us shift away from zero
 - ▶ but by default (if $g = (1, 1, \dots)^\top$ and $b = 0$), inputs to f have mean zero and variance 1

bias often still
dropped

RMS norm

- Given $a = Wx \in \mathbb{R}^d$
 - ▶ define $\bar{a} = 0$
 - ▶ define $\sigma^2 = \frac{1}{d} \sum_{i=1}^d (a_i - \bar{a})^2 + \epsilon = \frac{1}{d} \sum_{i=1}^d a_i^2 + \epsilon$
 - ▶ define $a' = \frac{a - \bar{a}}{\sigma} = \frac{a}{\sigma}$
- New version of layer, with RMS norm: $f(g \circ a' + b)$
 - ▶ only difference is not to subtract mean as first step
 - ▶ somewhat faster, similar performance

What to normalize?

- Layer norm says the bag of hidden activations for any *single* example should have mean 0, variance 1
- Alternate goal: *each* hidden activation should have mean 0, variance 1 *across* examples
 - ▶ enforcing this property is called *batch norm*
 - ▶ can't measure it with just one example
 - ▶ makes batch norm more complicated to implement

Batch norm

- To the rescue: *minibatch* SGD
- Measure mean and variance in each minibatch
- Given $a_i = Wx_i \in \mathbb{R}^d$ for each example in minibatch B
 - ▶ define $\bar{a} = \frac{1}{|B|} \sum_{i \in B} a_i \quad \bar{a} \in \mathbb{R}^d$
 - ▶ define $\sigma^2 = \frac{1}{|B|} \sum_{i \in B} (a_i - \bar{a})^2 + \epsilon \quad \sigma \in \mathbb{R}^d$
 - ▶ define $a' = \frac{a - \bar{a}}{\sigma}$ ← *componentwise*
- New version of layer, with batch norm: $f(g \circ a' + b)$

contrast w/ layer norm where
mean, variance are scalars

Batch norm: test time

- We don't necessarily have a minibatch available at test time — so we can't compute \bar{a} and σ !
- So, after we're done with SGD:
 - ▶ learn \bar{a} and σ from entire training set
 - ▶ these parameters become part of our model
- Tempting: test in minibatches, compute $\bar{\alpha}$, σ then
 - ▶ why is this wrong?

Exploding gradients

$$z = f\left(g \circ \frac{Wx - \bar{a}}{\sigma} + b\right)$$

$$dL = \langle r, dz \rangle \quad dz = \underbrace{f' \left(g \circ \frac{Wx - \bar{a}}{\sigma} + b \right)}_{\text{chain rule}} \circ d \left(g \circ \frac{Wx - \bar{a}}{\sigma} + b \right)$$

$$= g' \circ g \circ \left(\frac{W dx}{\sigma} \right)$$
$$dL = \langle r, g' \circ g \circ \left(\frac{W dx}{\sigma} \right) \rangle$$

- All these normalization operations are differentiable
- When activations are small or large, normalization scales them
 - ▶ → possibility of vanishing or exploding gradients
 - ▶ must use a mitigation if we have normalization layers
 - ▶ most popular: residual connections

Poll

<https://forms.gle/QQWzcMY36bNMKb6c7>



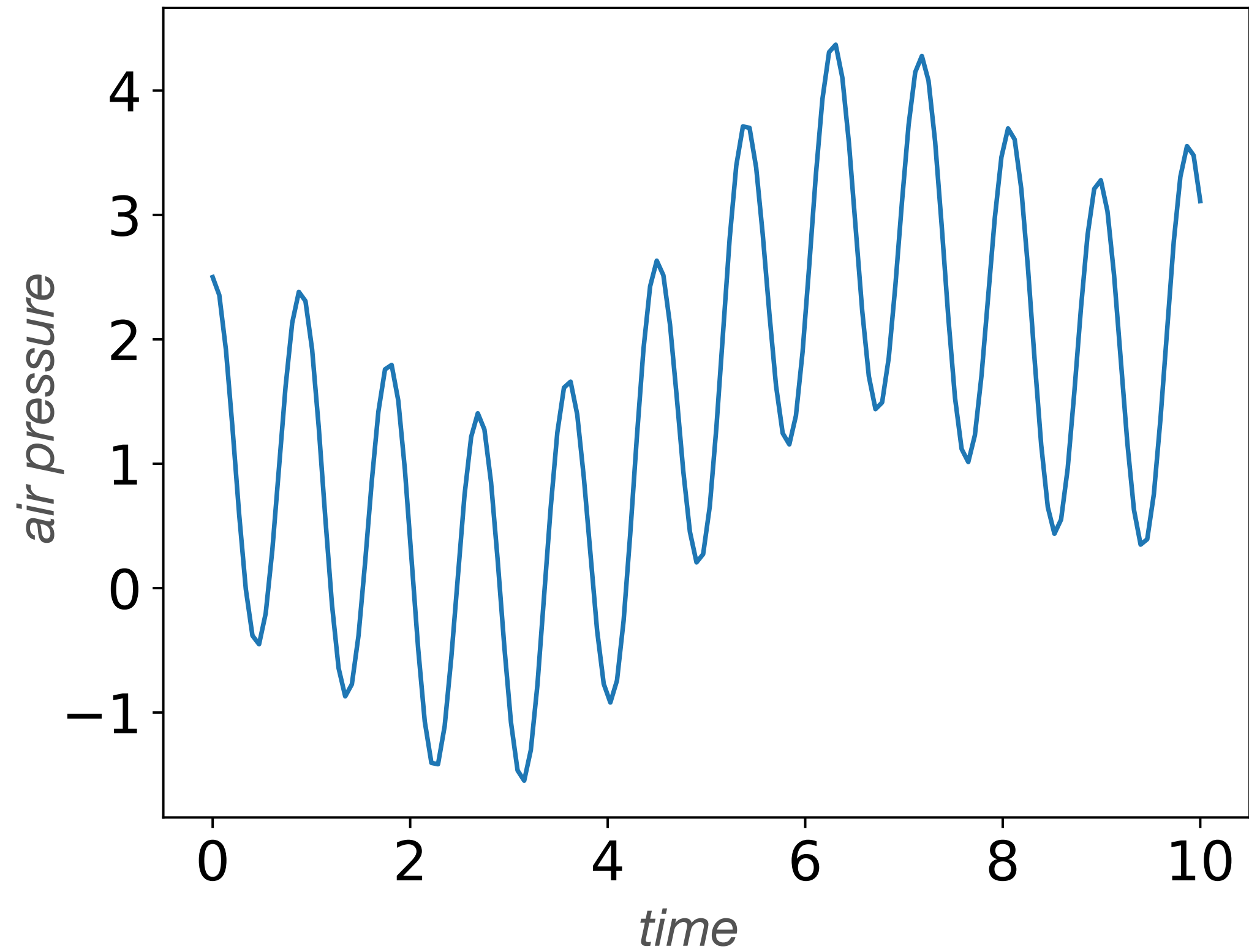
- If we want to avoid vanishing/exploding gradients, which of the following network features can help?
 - A. Careful initialization at the proper weight scale
 - B. Residual connections
 - C. Layer normalization blocks
 - D. Both A and B
 - E. Both B and C



High-dimensional learning

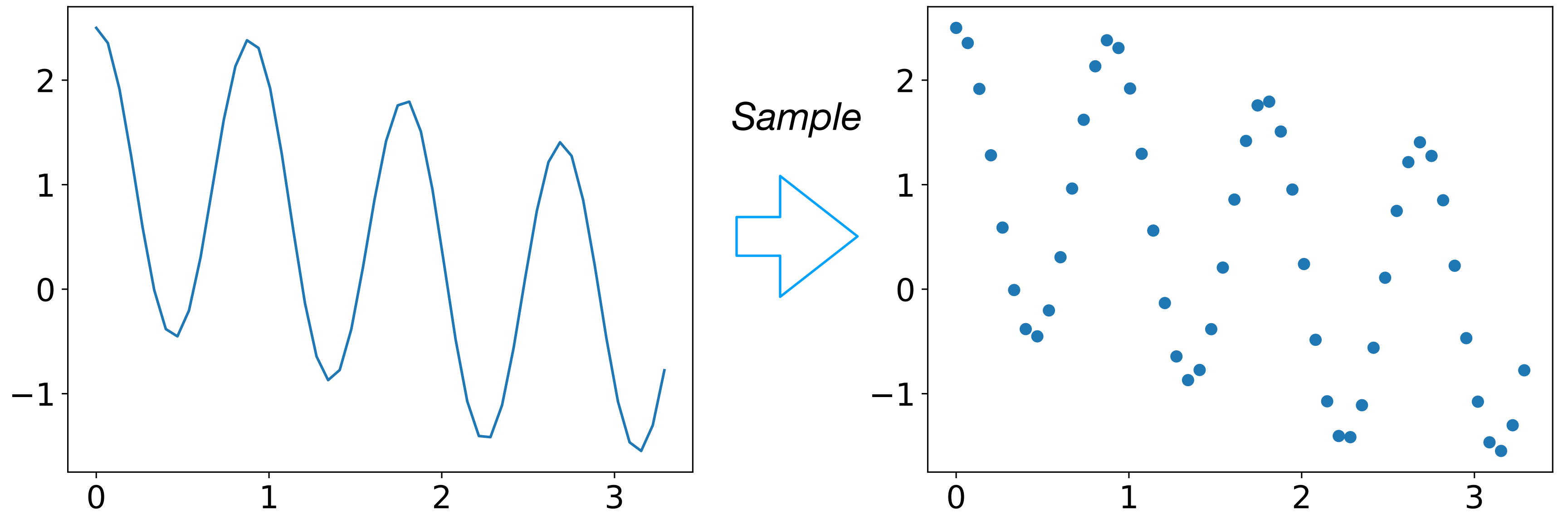
- In ML we often need to learn predictors for high-dimensional inputs and outputs
 - ▶ means we need to work with high-dimensional tensors (algebra, differentiation)
- Example: ***signal processing*** or prediction from audio, images, video, ...

Signal



- Signal: function from $\mathbb{R}^d \rightarrow \mathbb{R}$
- E.g., sound waves
 - ▶ here $d = 1$ represents time

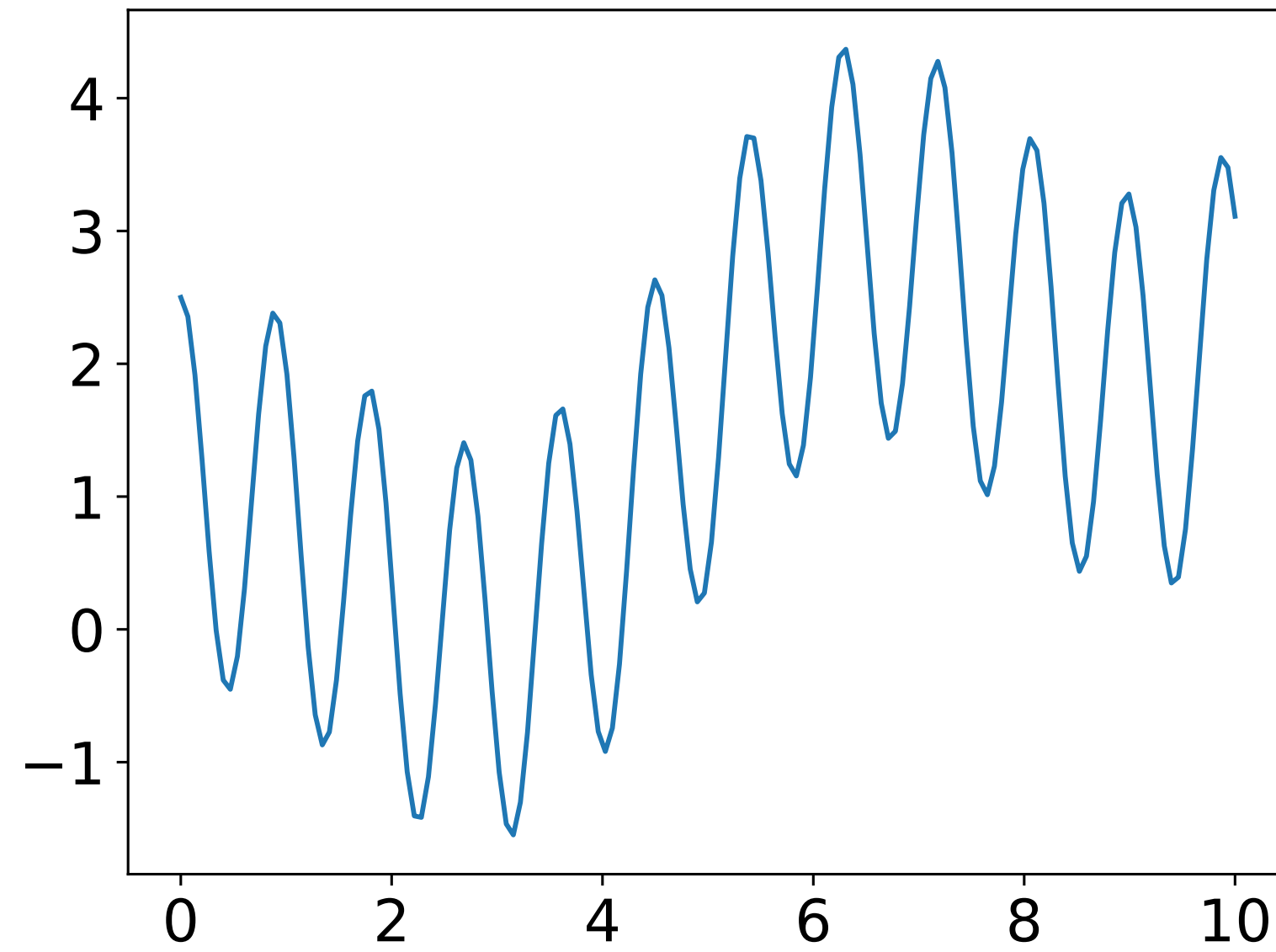
Sampled signal



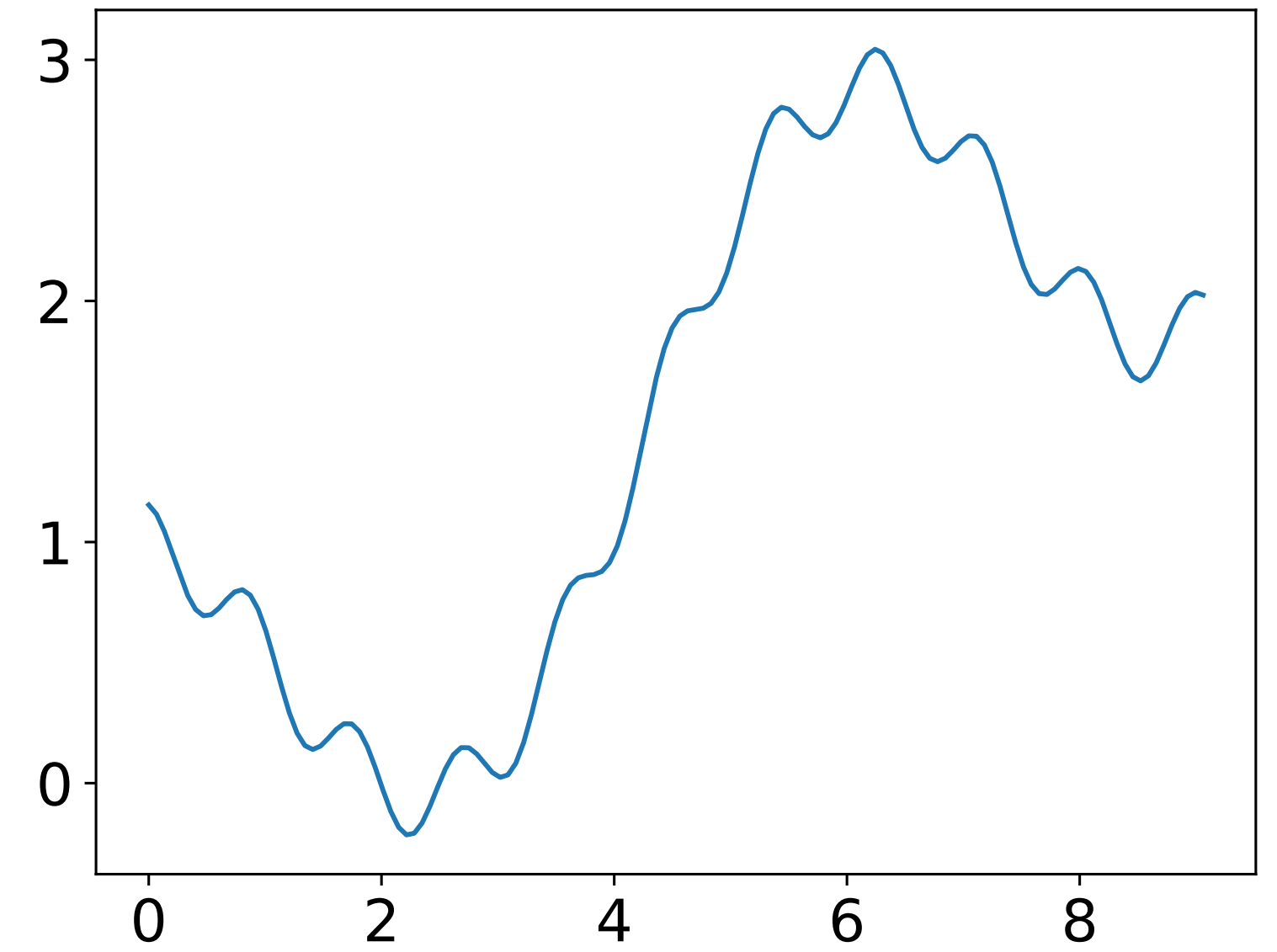
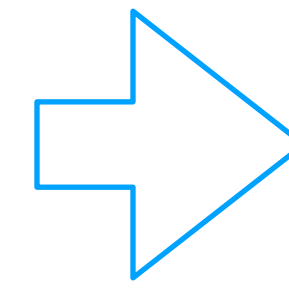
- Continuous-time signal $\xrightarrow{\text{sampling}}$ vector in \mathbb{R}^k
 - ▶ instead of $f(0.134) = 1.92$, we have $x_3 = 1.92$
- We do this every time we make a graph of a function
 - ▶ think of f as an infinite-d vector, x as its finite-d version
 - ▶ argument t in $f(t) \sim$ index i in x_i

Signal processing

low-pass filter



air pressure v. time



air pressure v. time



image credit: ChatGPT

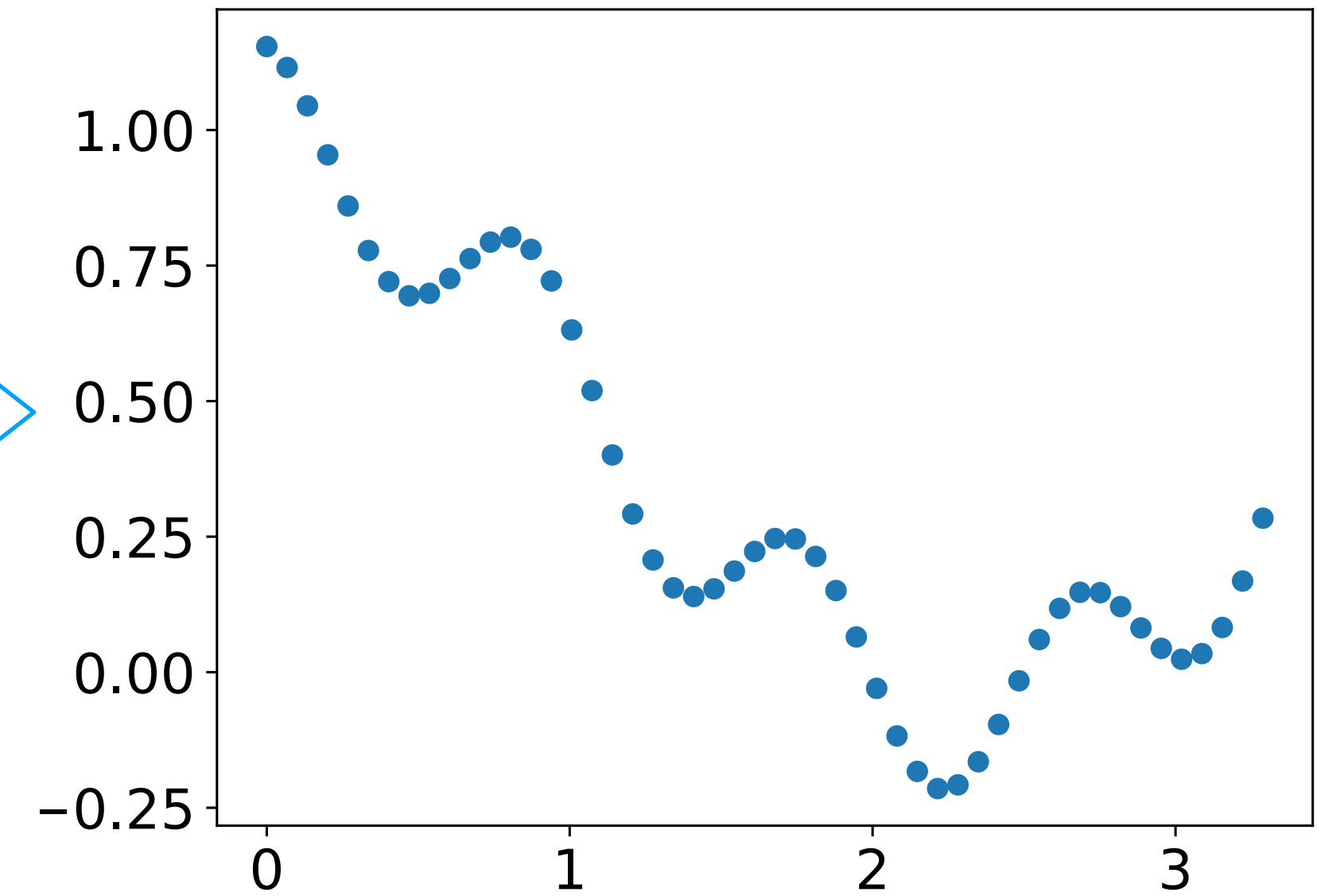
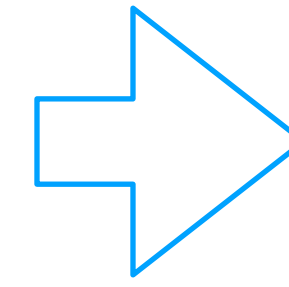
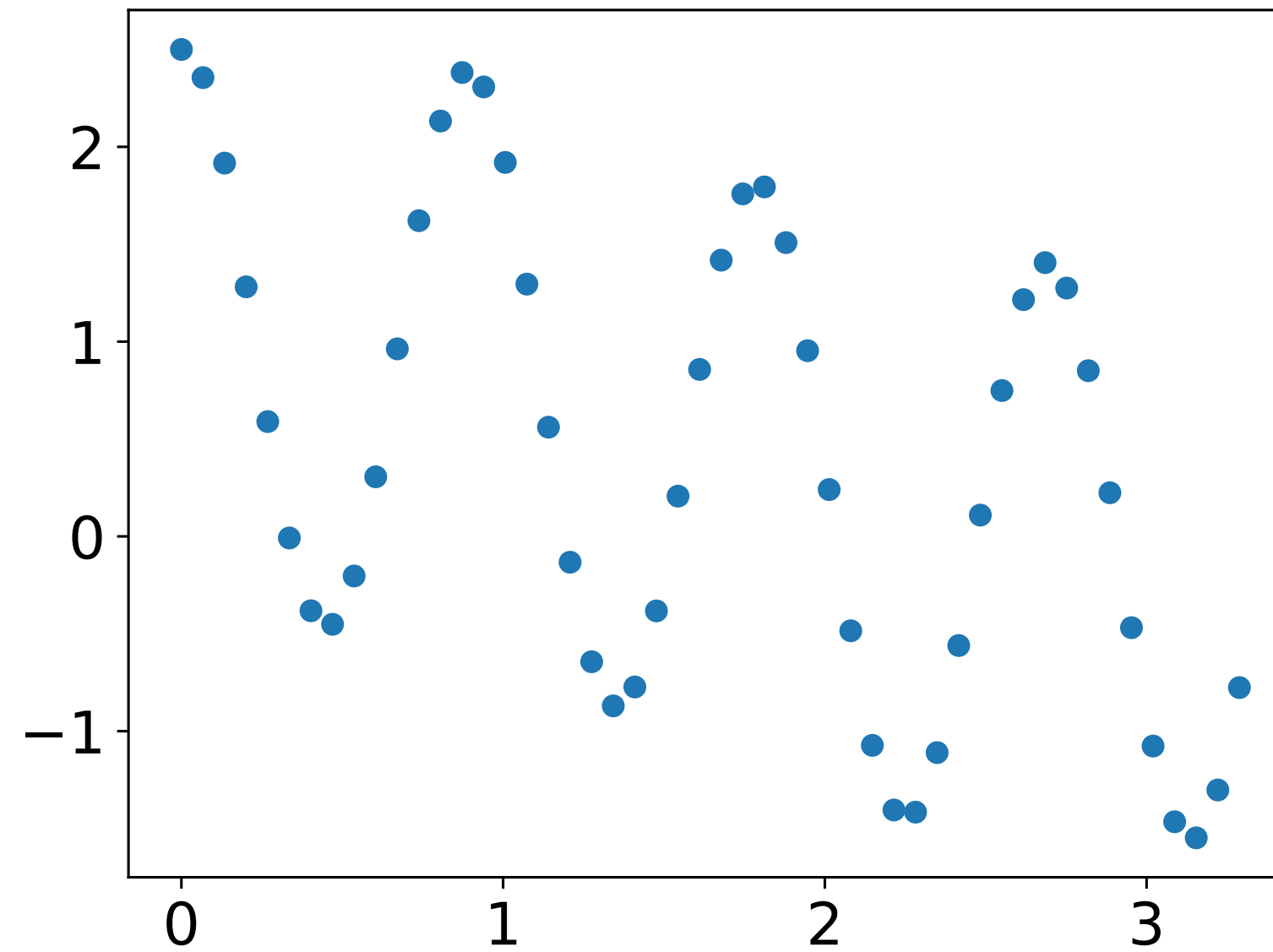
Fewer knobs



image credit: ChatGPT

***Filter =
linear
function***

low-pass filter



- Each coordinate of filtered signal is a linear function of input signal: $y = Wx$
 - ▶ $x, y \in \mathbb{R}^{50}$
 - ▶ $W \in \mathbb{R}^{50 \times 50}$

Similar behavior over time

- Audio filters are special matrices: behavior of filter stays fixed over time
 - ▶ i.e., the linear function that gives us y_{23} as a function of x is “the same as” the linear function that gives us y_{42}
 - ▶ “same” means the coefficients are the same but shifted in time

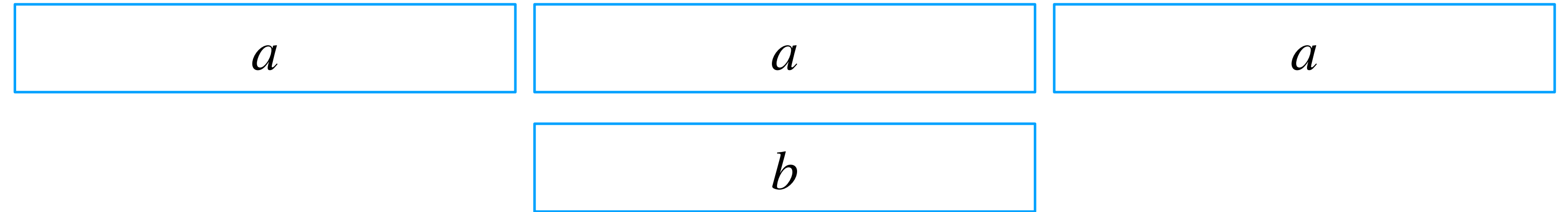
$$\begin{pmatrix} 0.33 & 0.33 & 0.33 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0.33 & 0.33 & 0.33 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0.33 & 0.33 & 0.33 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

- This operation is called *convolution*, and we can learn its weights (the filter) from examples

i.e., each output (say y_{17}) is a target (label); corresponding features are a window of nearby inputs (say $(x_{15}, x_{16}, \dots, x_{19})$)

Convolution

shift b along a , take dot product at each position



$$a = (1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 3 \ 2 \ 1)$$

$$b = (1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

$$a * b =$$

- Given two vectors $a, b \in \mathbb{R}^d$, their *convolution* is

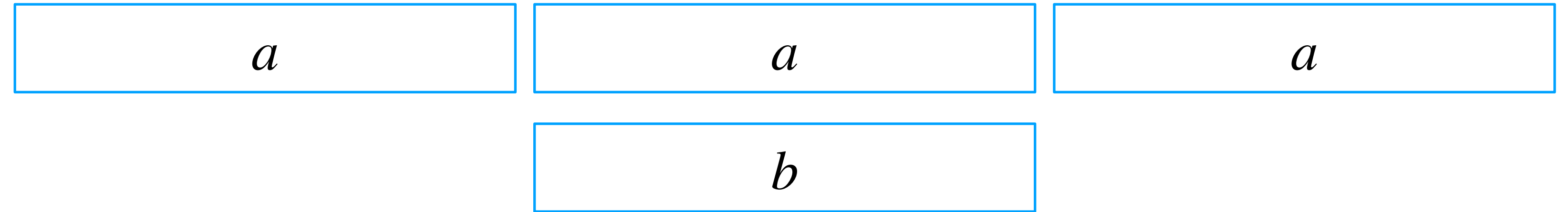
- ▶ $[a * b]_i = \sum_j a_{i+j \bmod d} b_j$

convention: 0-based indexing

note: deep net version

Convolution

shift b along a , take dot product at each position



$$a = (1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 3 \ 2 \ 1)$$

$$b = (1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

$$a * b = ((1 - 2) = -1$$

- Given two vectors $a, b \in \mathbb{R}^d$, their *convolution* is

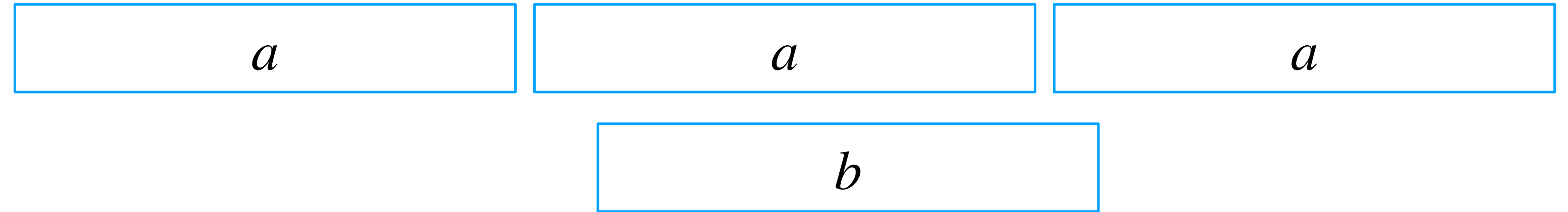
- ▶ $[a * b]_i = \sum_j a_{i+j \bmod d} b_j$

convention: 0-based indexing

note: deep net version

Convolution

shift b along a , take dot product at each position



$$a = (1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 3 \ 2 \ 1)$$

$$b = (1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

$$a * b = (-1 \ (2 - 3) = -1)$$

- Given two vectors $a, b \in \mathbb{R}^d$, their *convolution* is

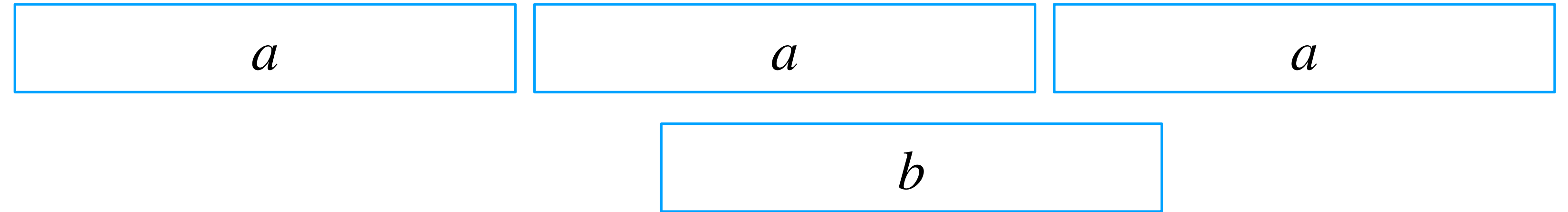
- ▶ $[a * b]_i = \sum_j a_{i+j \bmod d} b_j$

convention: 0-based indexing

note: deep net version

Convolution

shift b along a , take dot product at each position



$$a = (1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 3 \ 2 \ 1)$$

$$b = (1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

$$a * b = (-1 \ -1 \ (3 - 4) = -1)$$

- Given two vectors $a, b \in \mathbb{R}^d$, their *convolution* is

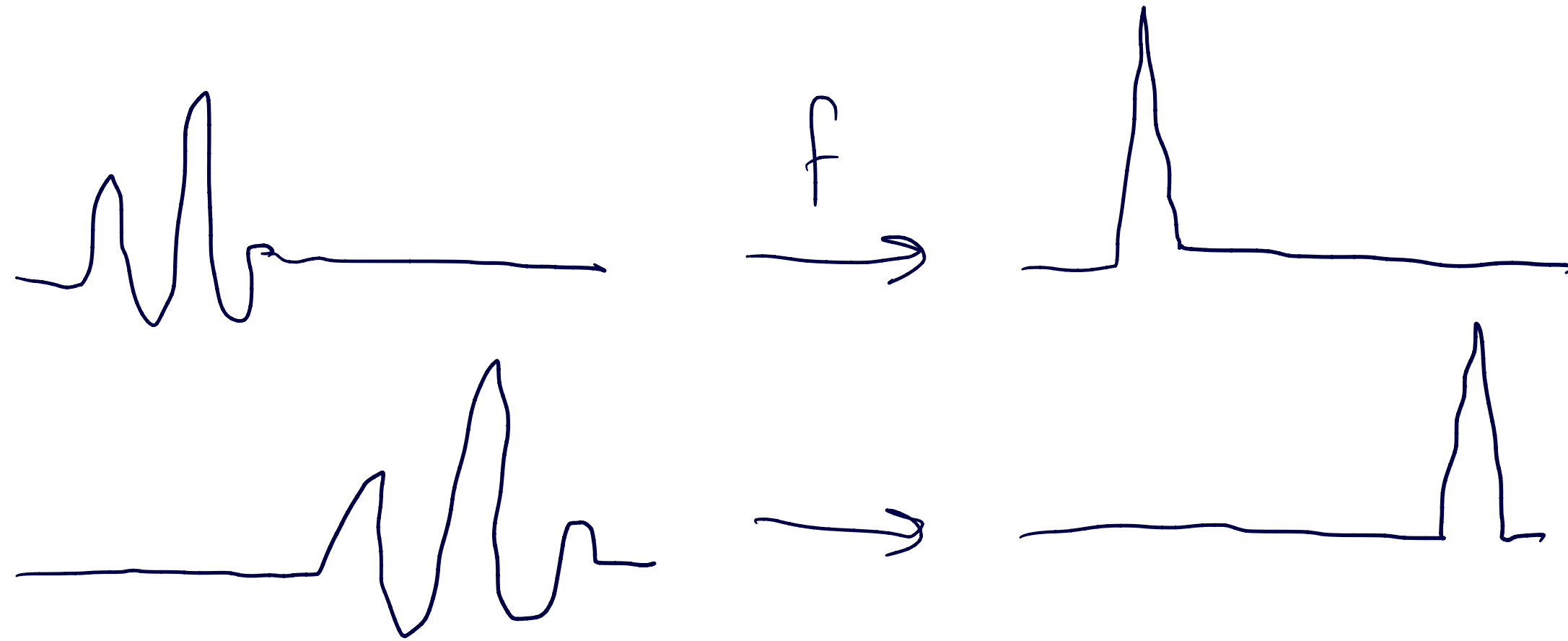
- ▶ $[a * b]_i = \sum_j a_{i+j \bmod d} b_j$

convention: 0-based indexing

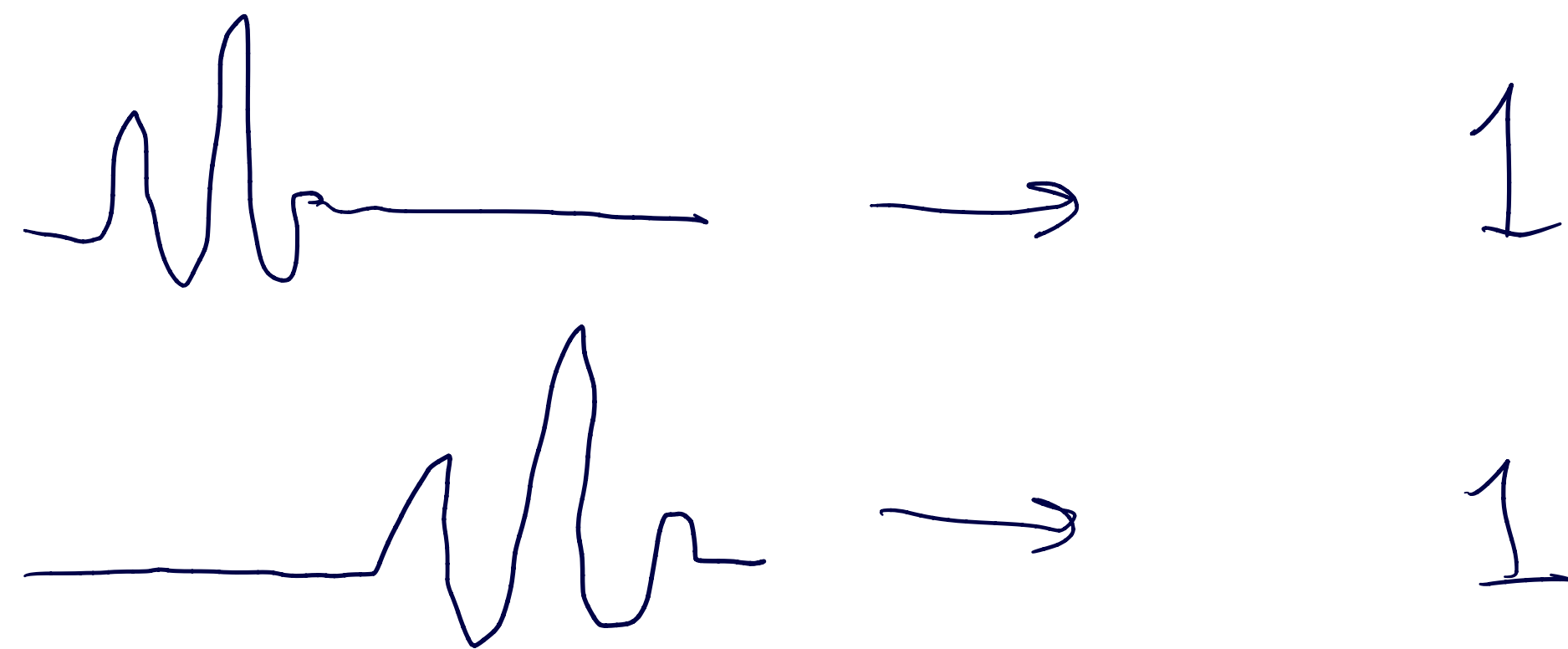
note: deep net version

Invariance and equivariance

- Equivariance



- Invariance



Convolution is

- Linear
 - ▶ $(a + 2a') * b = a * b + 2(a' * b)$
- Equivariant
 - ▶ shifting a or $b \rightarrow$ will shift the result the same amount
($a \rightarrow, b \leftarrow$)
- Parsimonious
 - ▶ filter has $O(d)$ parameters (fewer if sparse)
 - ▶ vs. $O(d^2)$ for ordinary linear layer
- Fast, from either
 - ▶ sparsity: $O(ds)$ when b has s nonzeros
 - ▶ FFT: $O(d \ln d)$ via multiplication-convolution theorem

$$a * b = \mathcal{F}^{-1}(\mathcal{F}a \circ \mathcal{F}b)$$

Variations

Boundary

$$a = (1 \ 2 \ 3)$$

(1 2 3 1 2 3 1 2 3) *cyclic*

(0 0 0 1 2 3 0 0 0) *zero*

(3 2 1 1 2 3 3 2 1) *reflect*

(1 1 1 1 2 3 3 3 3) *hold*

Different size

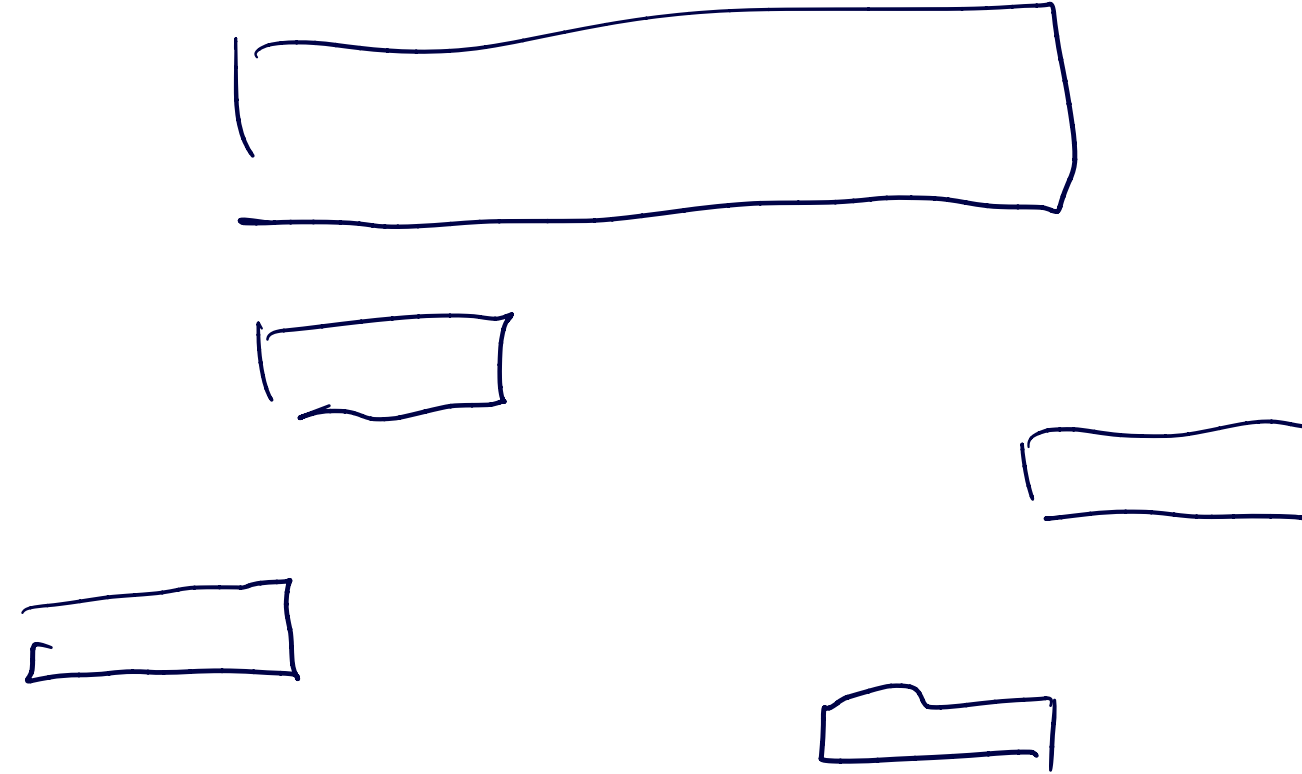
$$a = (1 \ 2 \ 3 \ 4 \ 5 \ 4 \ 3 \ 2 \ 1)$$

$$b = (1 \ -1)$$

$$\rightarrow b = (1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0)$$

- Boundary conditions
 - ▶ *cyclic* boundary condition (a repeats, as above)
 - ▶ *zero-pad* (add zeros next to a)
 - ▶ *reflect* (add flipped copies of a next to a)
 - ▶ *hold* (repeat closest element of a)
- Different-size vectors: e.g., b smaller than a
 - ▶ handled by boundary conditions — e.g., zero-pad b

$$[a * b]_i = \sum_j a_{i+j} \text{mod } d b_j$$



Variations

- Size of output
 - ▶ as above: output same size as input
 - ▶ big: keep going as long as b overlaps a
 - ▶ small (“valid”): only create outputs where boundary conditions don’t matter
 - ▶ PyTorch: default small, options control padding/mode

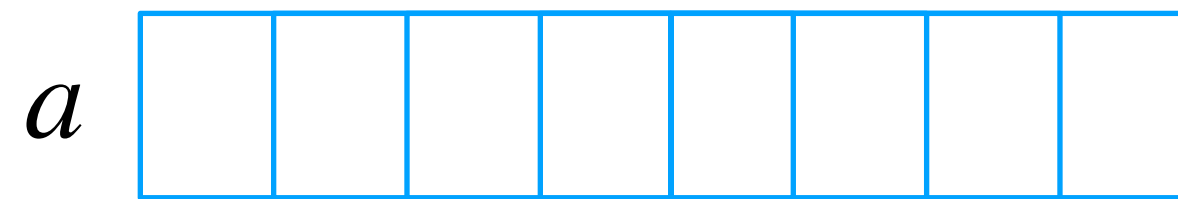
$$[a * b]_i = \sum_j a_{i+j \bmod d} b_j$$

Variations

- Dilation (cover larger area with few parameters)

- ▶ parameter k tells us to spread out b

- ▶ $[a * b]_i = \sum_j a_{i+kj \bmod d} b_j$

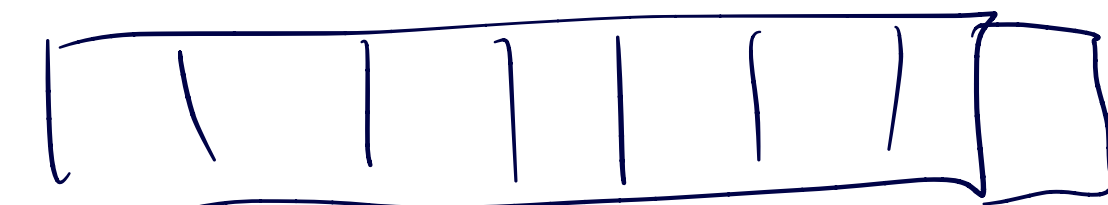
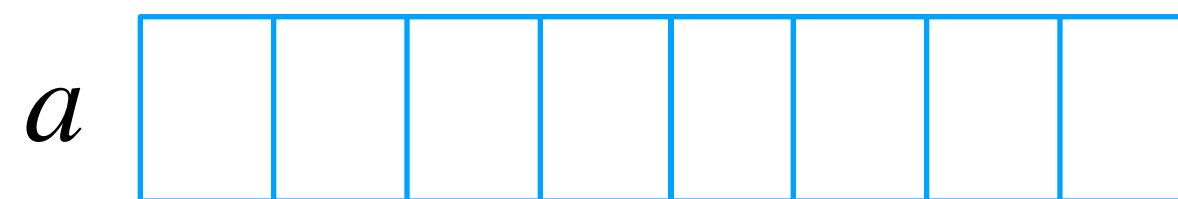


$$k = 3$$

- Stride (downsample to get smaller result)

- ▶ parameter ℓ tells us to skip starting positions in a

- ▶ $[a * b]_i = \sum_j a_{\ell i+j \bmod d} b_j$



8

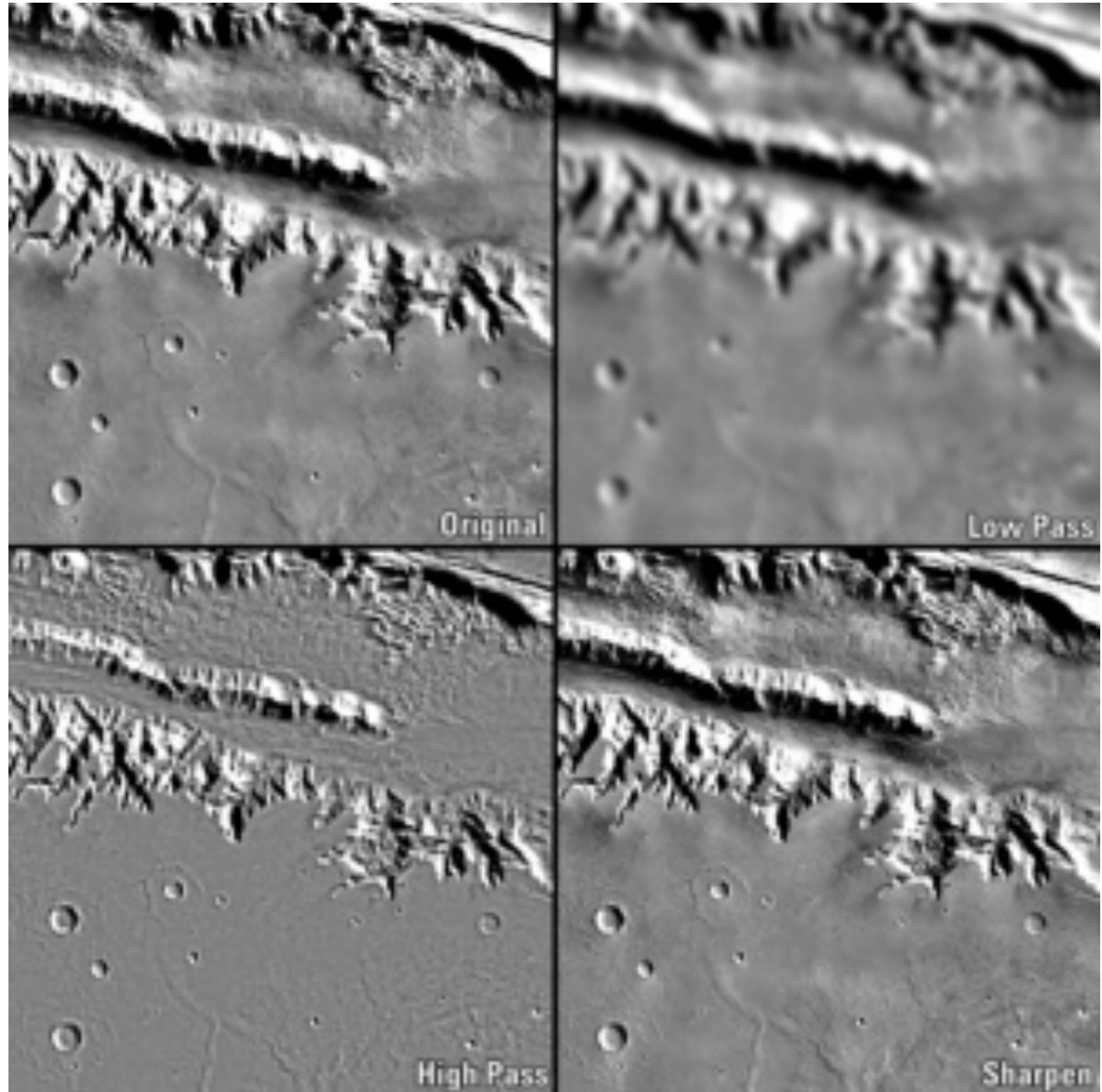
Convolution, invariance, equivariance

- Because convolution is equivariant, it's a great tool for developing equivariant ML models
 - ▶ we'll see some soon
- Also extremely useful for invariant models
 - ▶ e.g., suppose we have an *equivariant* model that outputs a spike when it detects a chirp
 - ▶ if we sum its outputs across time, we get # of chirps
 - ▶ which is *invariant* to shifts

n-d signals

- Beyond having graphs (1d signals) as inputs, common to have 2d, 3d, or higher
 - ▶ e.g., temperature vs. location across US (2d)
 - ▶ e.g., picture of fluffy kitten downloaded from internet (2d if grayscale, 3d if color)
 - ▶ e.g., gas pressure throughout a reaction vessel (3d)
 - ▶ e.g., value function for a control problem where state is the position of a rigid body (6d)

2-d convolution



credit: USGS

2-d convolution

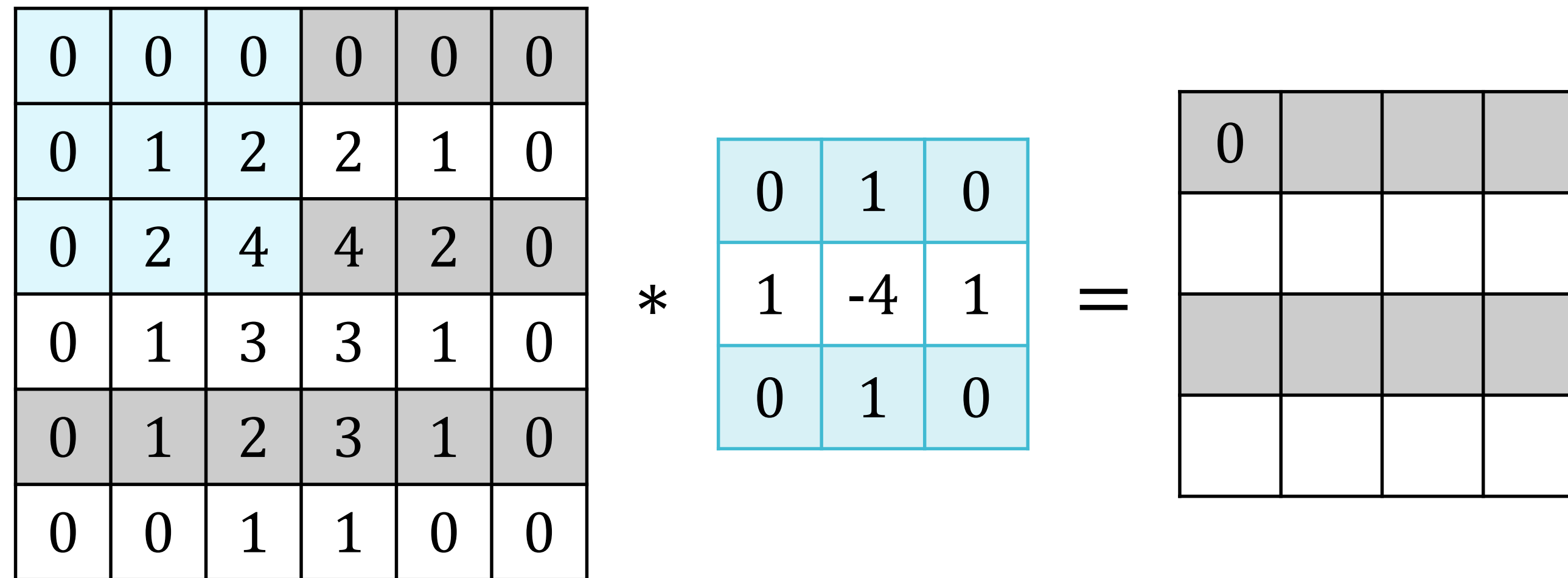
0	0	0	0	0	0
0	1	2	2	1	0
0	2	4	4	2	0
0	1	3	3	1	0
0	1	2	3	1	0
0	0	1	1	0	0

 *

0	1	0
1	-4	1
0	1	0

Just like 1-d convolution, we place the filter at each possible position over the input, then take “dot product”

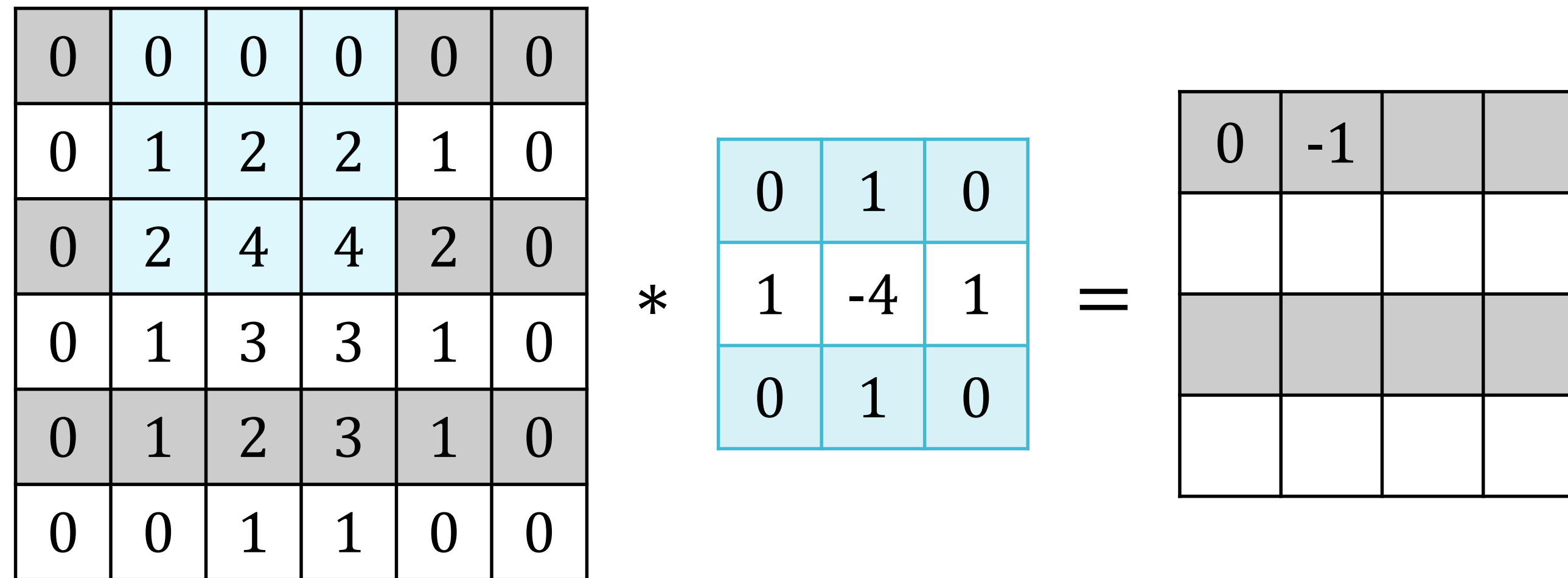
2-d convolution



$$(0 * 0) + (0 * 1) + (0 * 0) + (0 * 1) + (1 * -4) + (2 * 1) + (0 * 0) + (2 * 1) + (4 * 0) = 0$$

Just like 1-d convolution, we place the filter at each possible position over the input, then take “dot product”

2-d convolution



$$(0 * 0) + (0 * 1) + (0 * 0) + (1 * 1) + (2 * -4) \\ + (2 * 1) + (2 * 0) + (4 * 1) + (4 * 0) = -1$$

Just like 1-d convolution, we place the filter at each possible position over the input, then take “dot product”

2-d convolution

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

 *

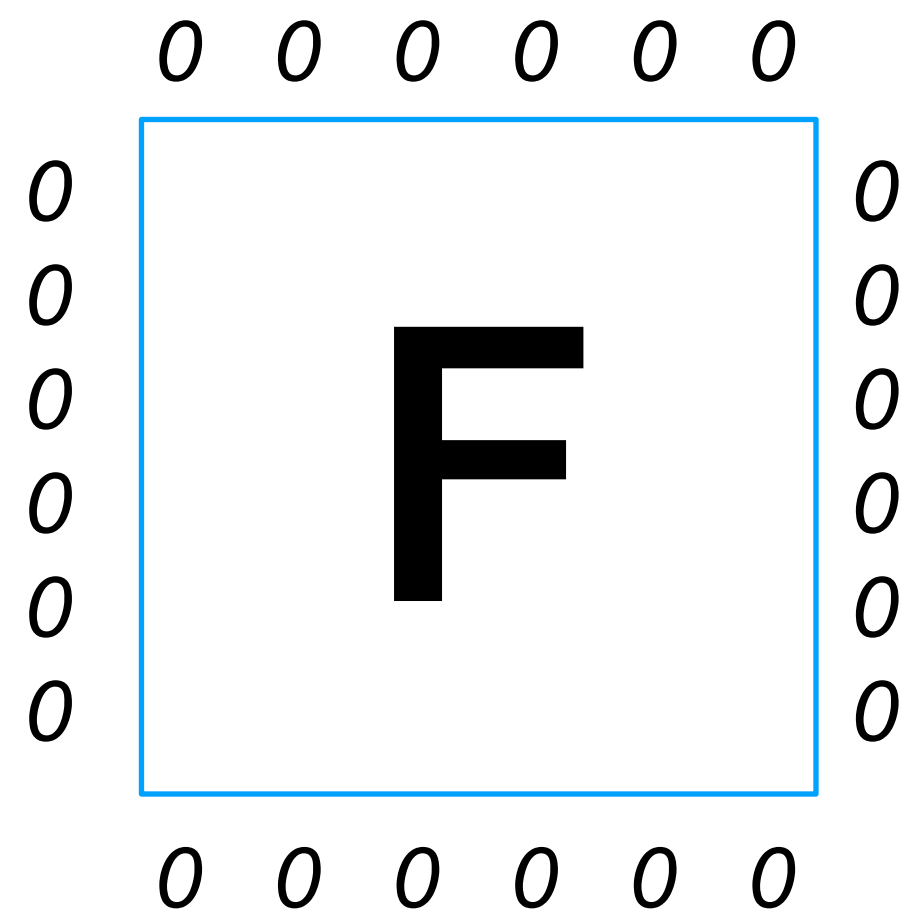
0	1	0
1	-4	1
0	1	0

 =

0	-1	-1	0
-2	-5	-5	-2
2	-2	-1	3
-1	0	-5	0

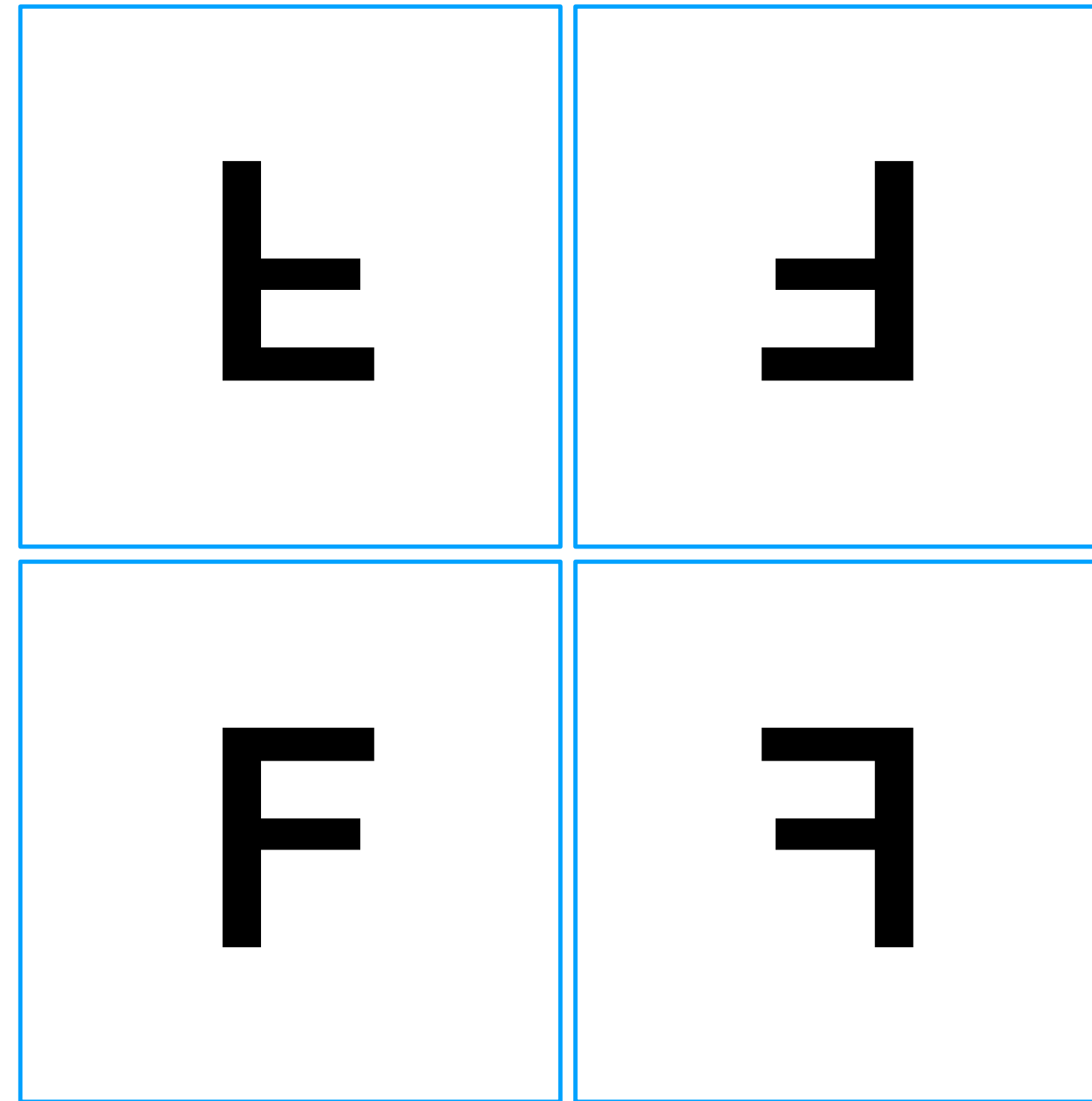
Just like 1-d convolution, we place the filter at each possible position over the input, then take “dot product”

Boundary conditions



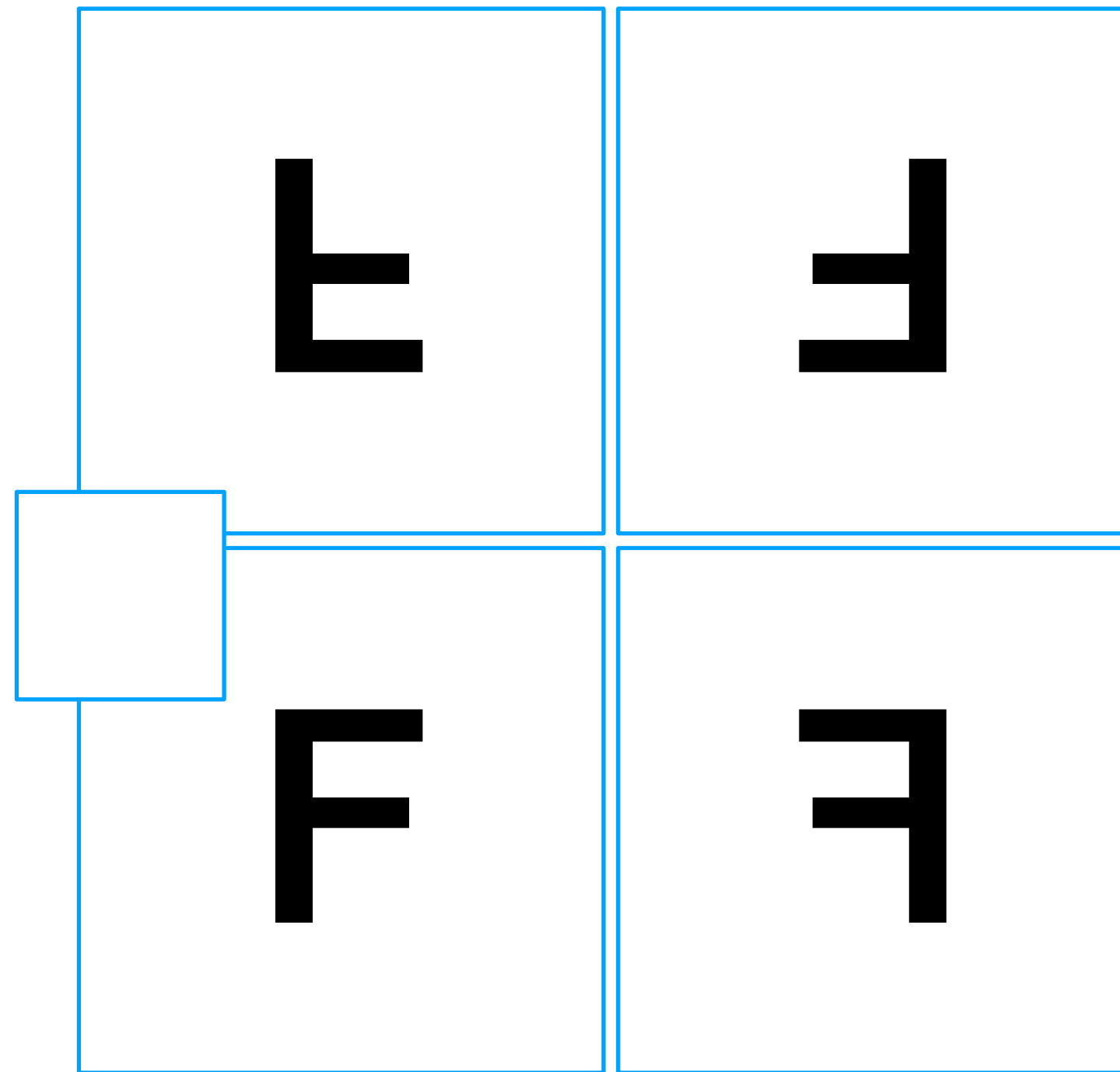
*Similar choices for boundary conditions and size: e.g., **valid** (previous slide), **zero**, or **reflect***

Boundary conditions



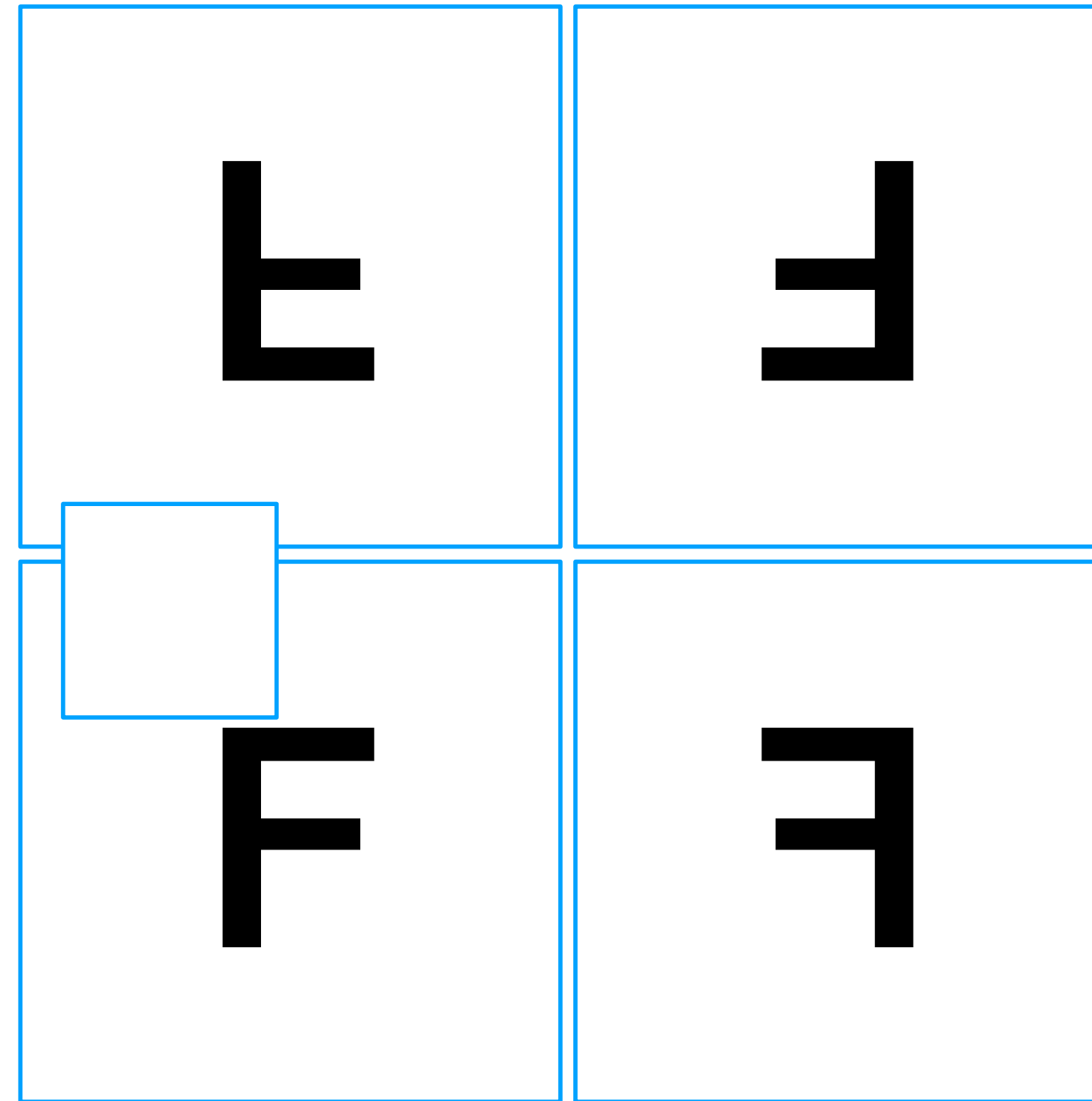
*Similar choices for boundary conditions and size: e.g.,
valid (previous slide), **zero**, or **reflect***

Boundary conditions



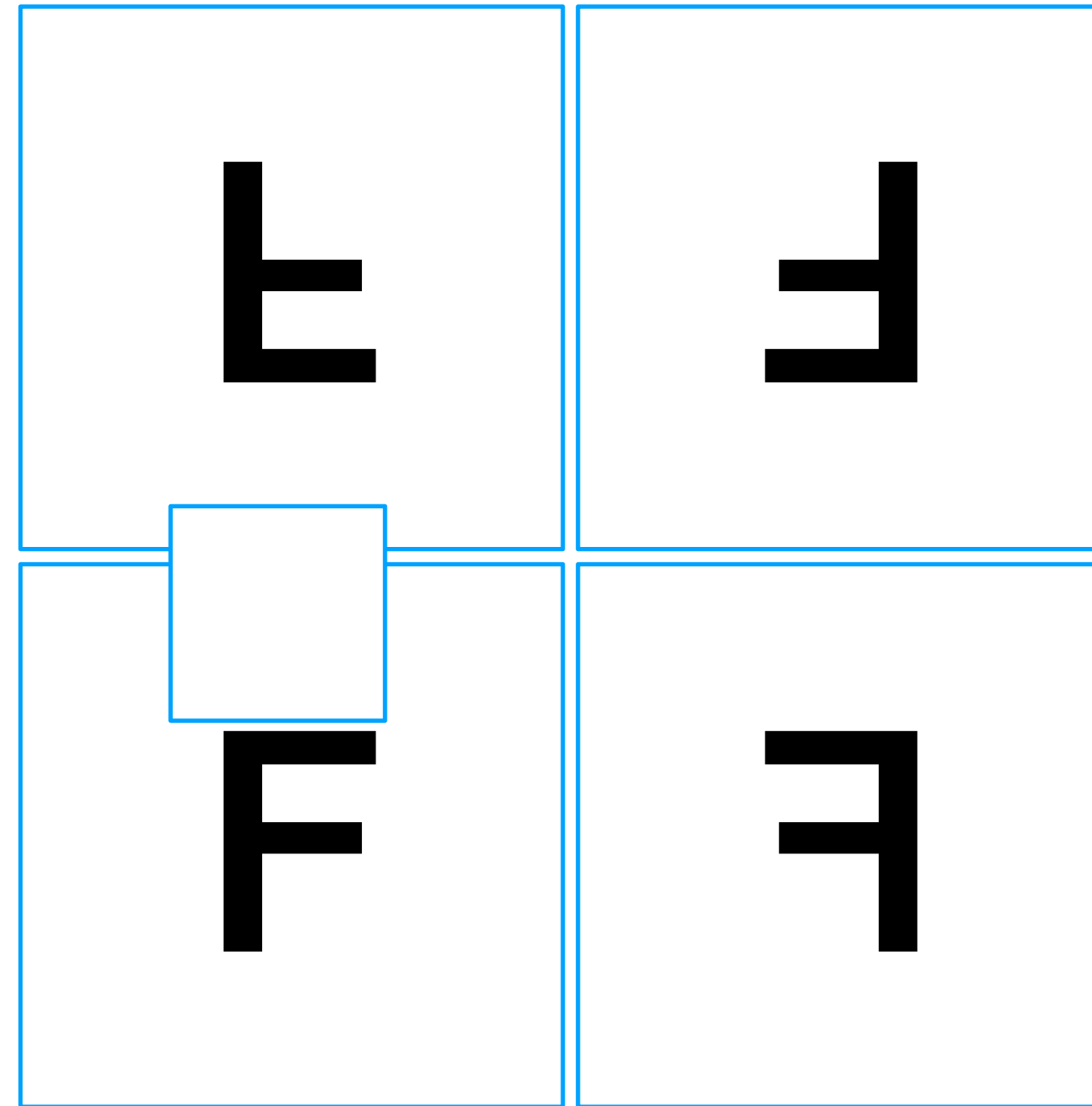
*Similar choices for boundary conditions and size: e.g., **valid** (previous slide), **zero**, or **reflect***

Boundary conditions



*Similar choices for boundary conditions and size: e.g.,
valid (previous slide), **zero**, or **reflect***

Boundary conditions



*Similar choices for boundary conditions and size: e.g., **valid** (previous slide), **zero**, or **reflect***

Equivariance

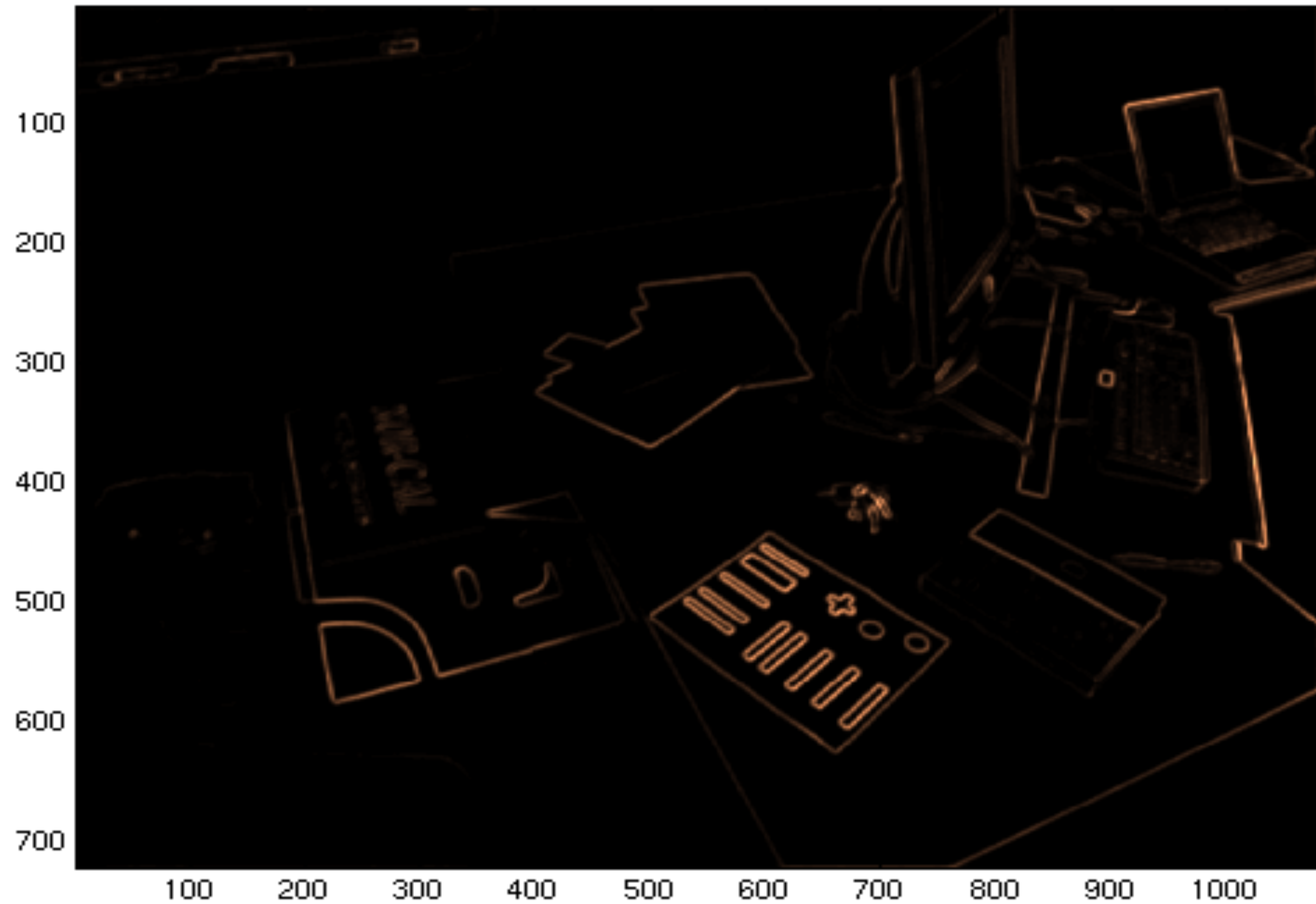
- Convolution remains equivariant in 2-d
 - ▶ in fact, in any number of dimensions
- Shift input any amount in any direction → output shifts same amount in same direction

What can convolution do?



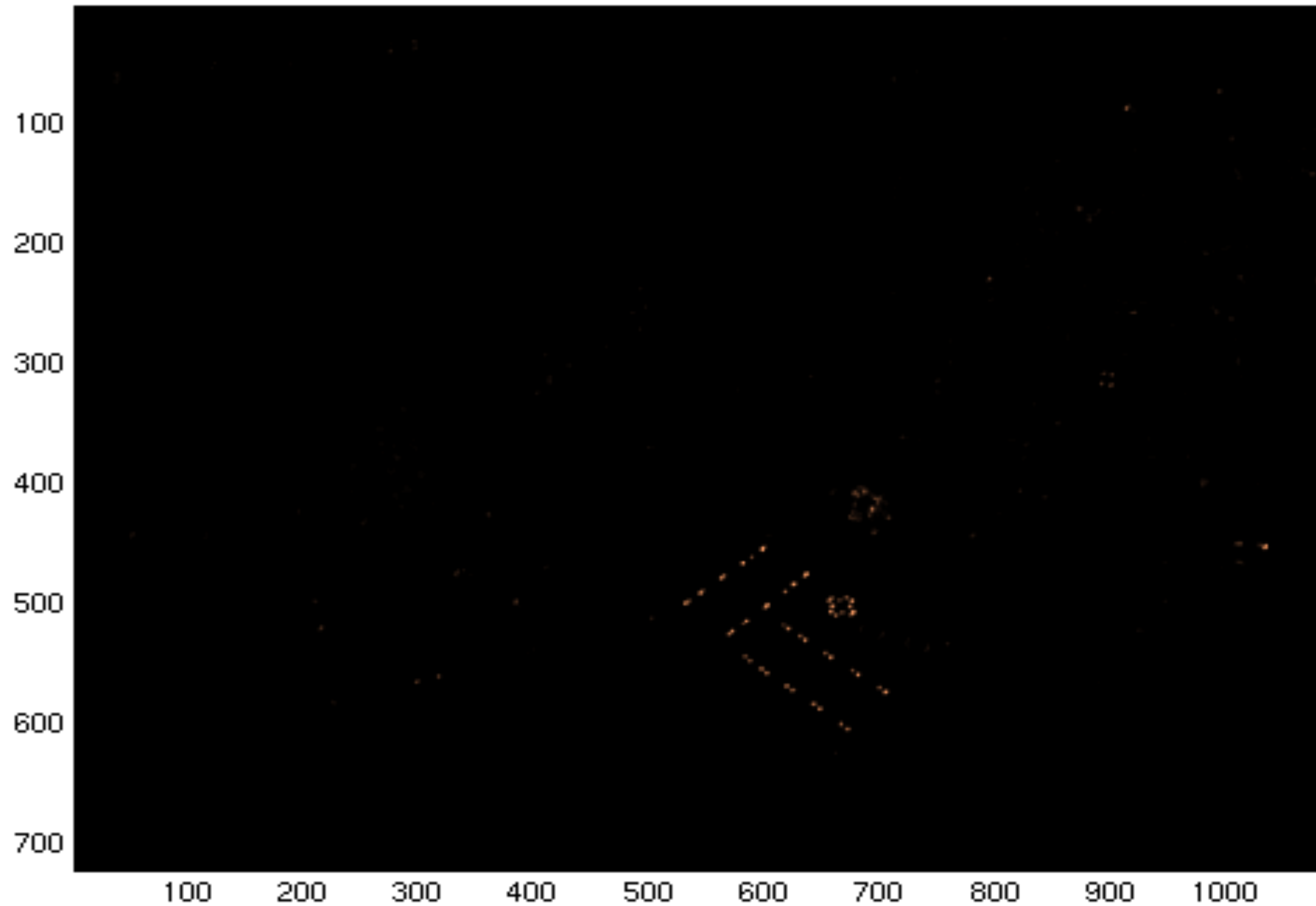
- We've seen: low-pass filter (smoothing), high-pass filter, sharpen (high-pass + residual), edge detection
- Other common uses: detrending, equalization of brightness and contrast, corner detection
 - ▶ here: fixed filter and nonlinearity

What can convolution do?



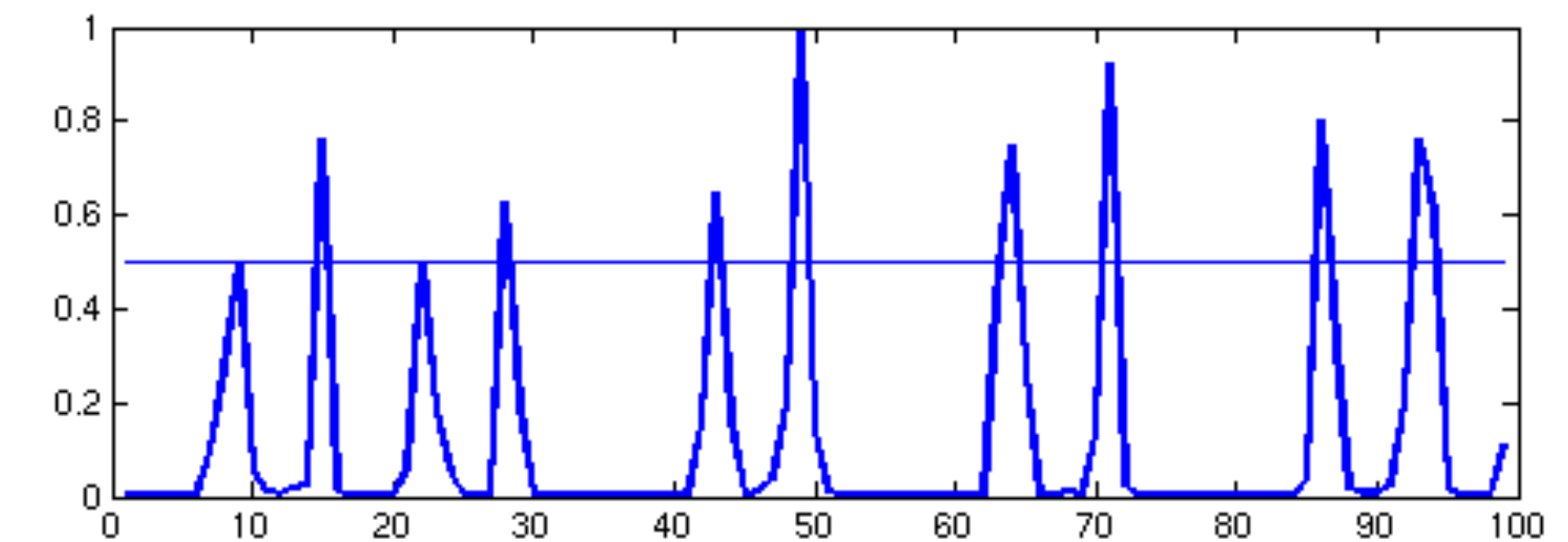
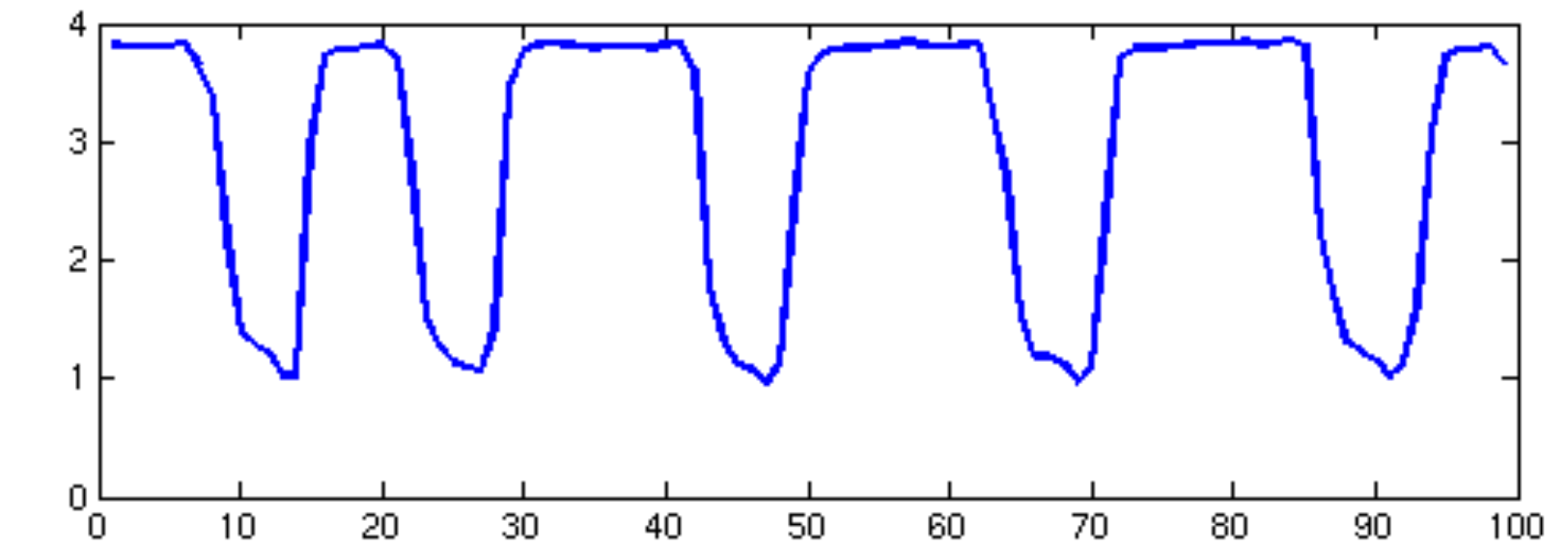
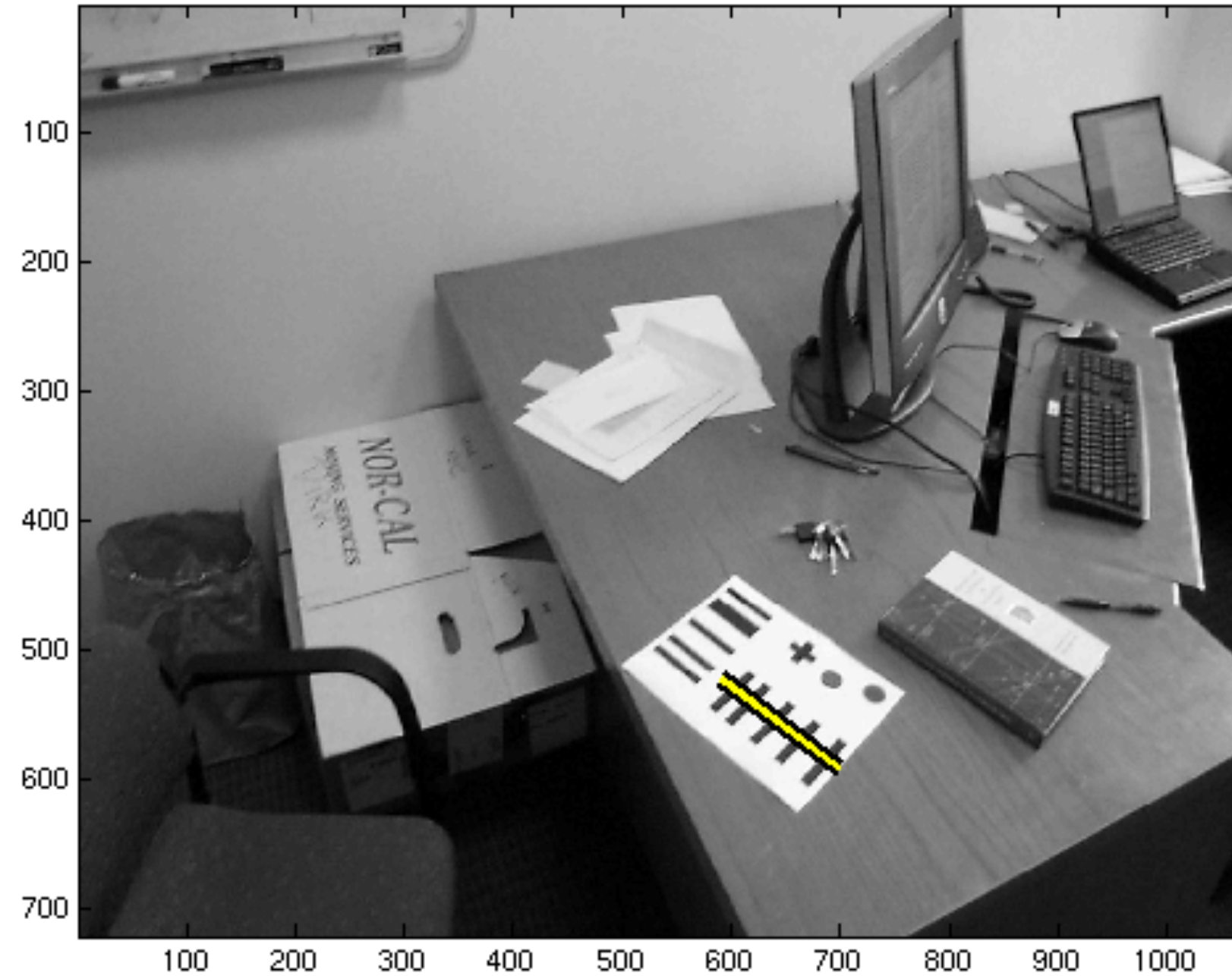
- We've seen: low-pass filter (smoothing), high-pass filter, sharpen (high-pass + residual), edge detection
- Other common uses: detrending, equalization of brightness and contrast, corner detection
 - ▶ here: fixed filter and nonlinearity

What can convolution do?



- We've seen: low-pass filter (smoothing), high-pass filter, sharpen (high-pass + residual), edge detection
- Other common uses: detrending, equalization of brightness and contrast, corner detection
 - ▶ here: fixed filter and nonlinearity

What can convolution do?



- Simple application: detect and read bar codes in an image (what your phone is doing for the polls)

Signals as tensors

- An n -dimensional signal is represented as a tensor with n indices
 - ▶ e.g., grayscale image: 2 indices (pixel i, j)
 - ▶ e.g., color image: 3 indices (last is channel: R, G, or B)
 - ▶ e.g., pressure in a volume: 3 indices (voxel i, j, k)
- Each index also called a **mode**
 - ▶ helps with overloading of the word “dimension”
- List of index ranges: the **shape** of the tensor
 - ▶ e.g., RGB image at VGA resolution: (640, 480, 3)
 - ▶ shape (3, 4, 2, 5) tensor has 120 total elements
 - ▶ first index can be 0, 1, 2; second 0, 1, 2, 3; ...

Tensor operations

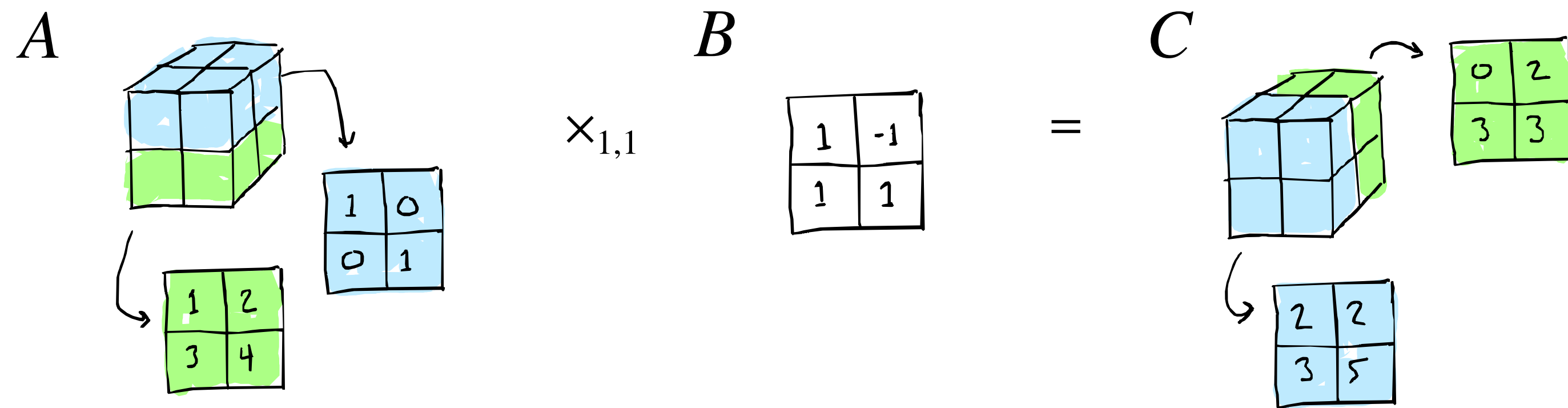
$$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix}$$

- Componentwise $+$, \times , \cos , ...
 - ▶ with *broadcasting*:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 9 \\ 4 & 10 & 18 \end{pmatrix}$$

Tensor operations

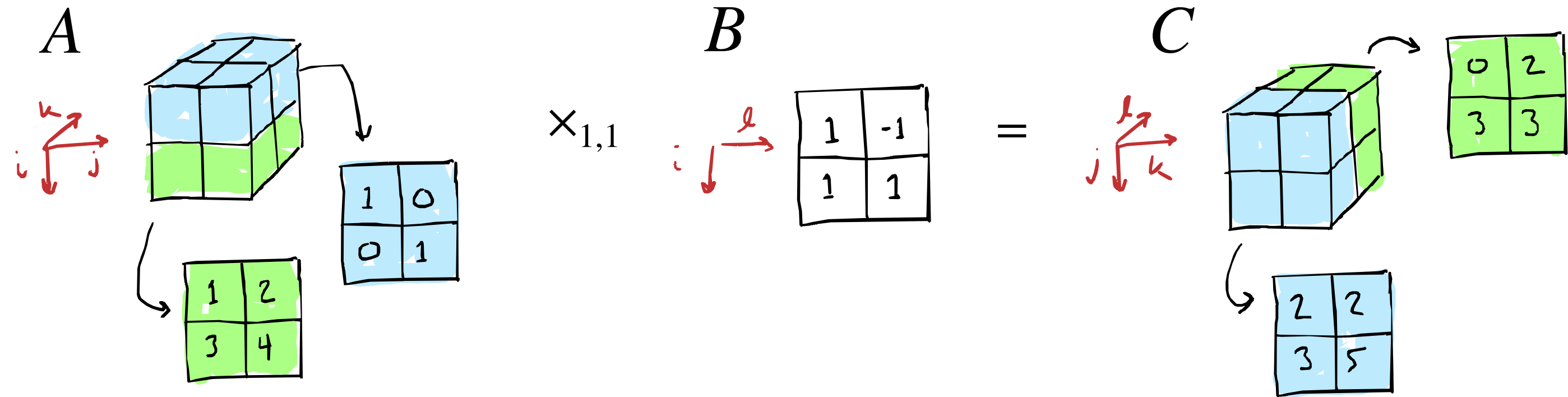


- Contraction: e.g., $A \times_{3,2} B$ or $A \times_{1,1} B$
 - ▶ like matrix multiplication (precise definition below), but using mode 3 of A and mode 2 of B — regular matrix multiplication is $\times_{2,1}$
 - ▶ modes of $A \times_{3,2} B$: modes of A (except 3) then modes of B (except 2)
 - ▶ by convention, all of A 's modes first (in order) then B 's
- Represents a linear function of B with parameters A
 - ▶ or a linear function of A with parameters B

Contraction is a special case of Einstein summation

- Convention: a_{ij} means a matrix whose i, j element is a_{ij}
 - ▶ range of i, j can be left implicit or specified separately:
e.g., $i \in \{1 \dots 5\}, j \in \{1 \dots 7\}$
- Similarly, a_{ijkl} means a 4-mode tensor
- So does $a_{ijk}b_{jkl}$ — the i, j, k, l element is $a_{ijk}b_{jkl}$
- To represent contraction, we raise one instance of index
 - ▶ e.g., $a_{ijk}b_{jl}^k$ is a 3-mode tensor
 - ▶ element i, j, l is $\sum_k a_{ijk}b_{jkl}$
- In general (Einstein summation), any index that appears both raised and lowered gets summed out
 - ▶ $a_{ij}^k b_{jk}^l c_{lm}$ has indices i, j, m , elements $\sum_k \sum_l a_{ijk} b_{jkl} c_{lm}$

Tensor contraction as Einstein summation



- Can write the tensor contraction from above as an Einstein summation: $[A_{ijk}B_l^i] = [C_{jkl}]$
- Notation difference: instead of numbering the modes, we name them by their indices