Lecture 4
Image representations

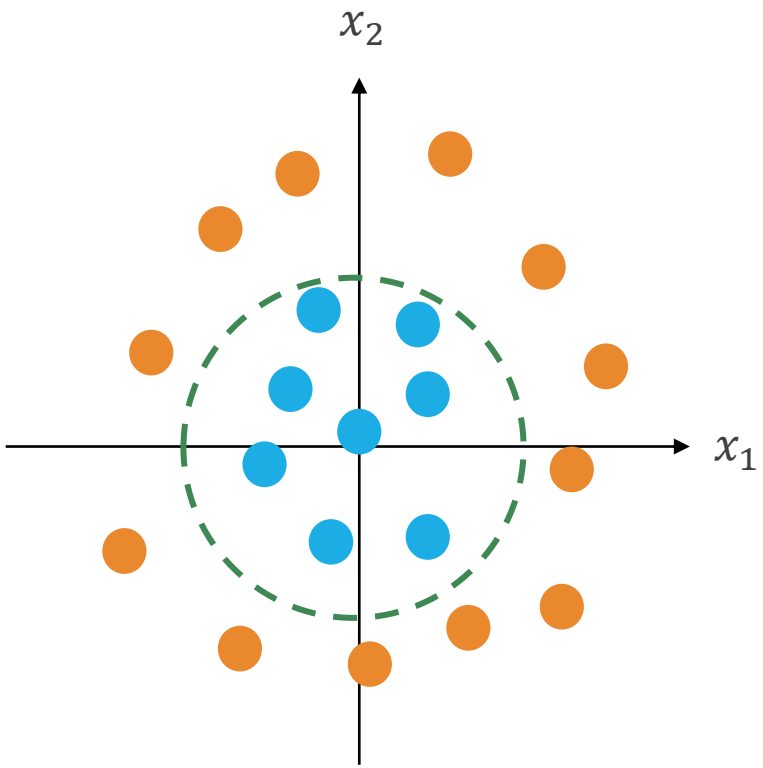
IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

Limits of “shallow” classifiers

k-NN and linear classifiers are limited by the low effectiveness of input pixels as data features.

Representation is important

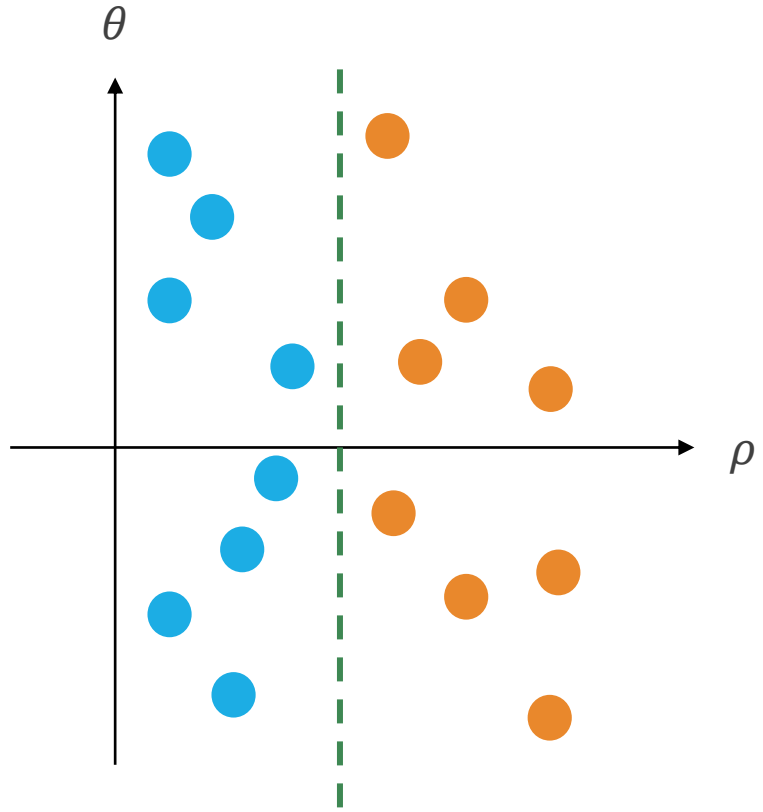


Non-linear decision boundary in input space

Switch to polar coordinates

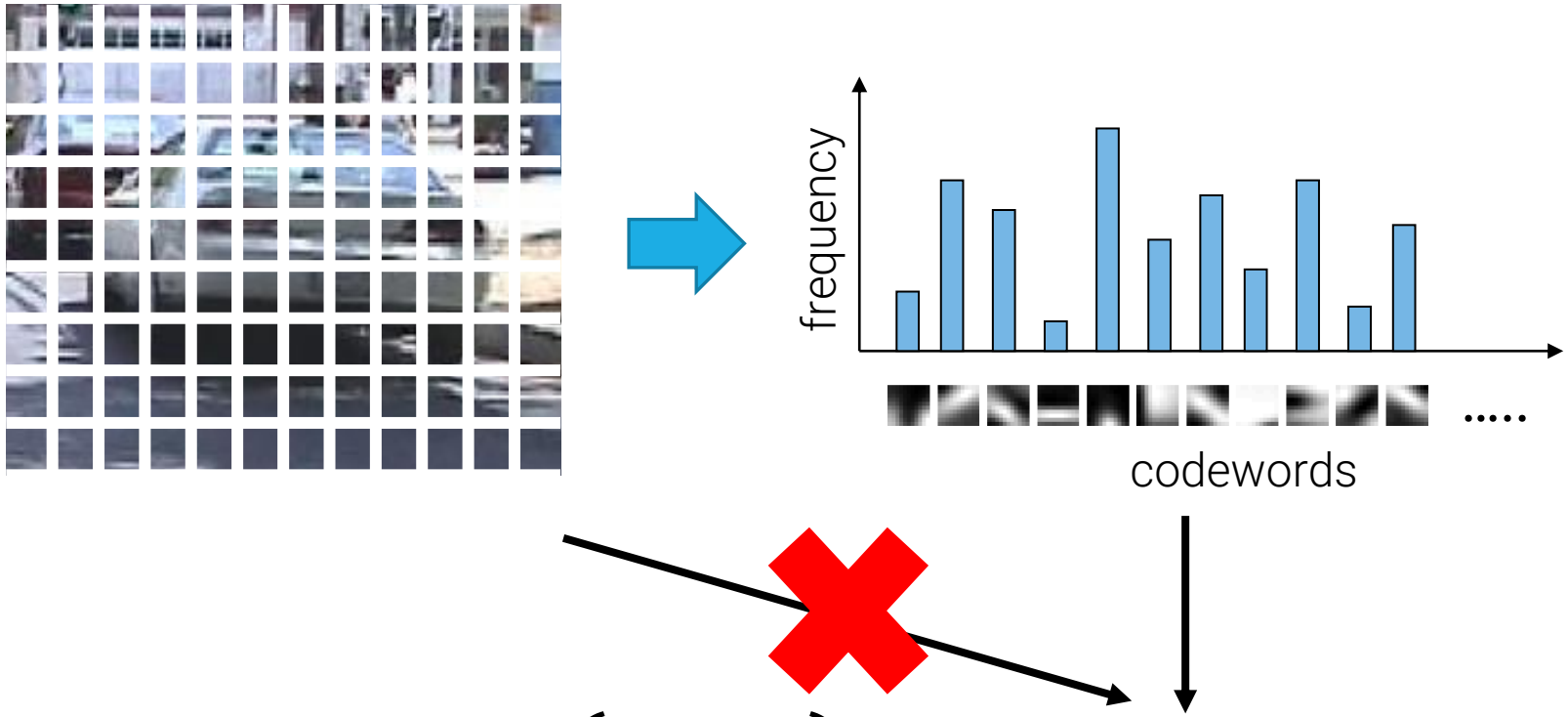
$$\rho = \sqrt{x_1^2 + x_2^2}$$

$$\theta = \tan^{-1}\frac{x_2}{x_1}$$



Linear decision boundary in feature space

Bag of Visual Words (BoVW)



The classifier does not work anymore with the image pixels directly, but with a histogram of codeword frequencies, known as Bag of (Visual) Words (BoVW), inspired by similar representations that were popular at the time in Natural Language Processing.

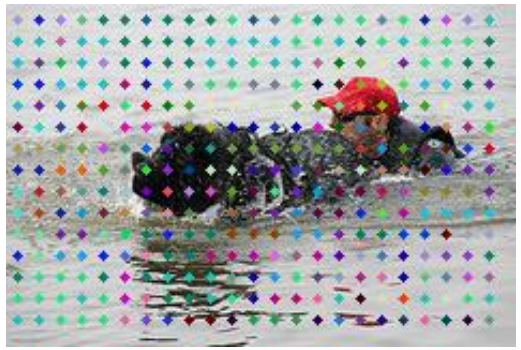
$$f(\boldsymbol{x};\theta) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{scores}$$

Csurka et al., Visual Categorization with Bags of Keypoints, ECCV 04
Fei-Fei et Perona, A Bayesian Hierarchical Model for Learning Natural Scene Categories, CVPR 05
Sivic et al, Discovering objects and their location in images, ICCV05

BoVW was the dominant paradigm until 2012

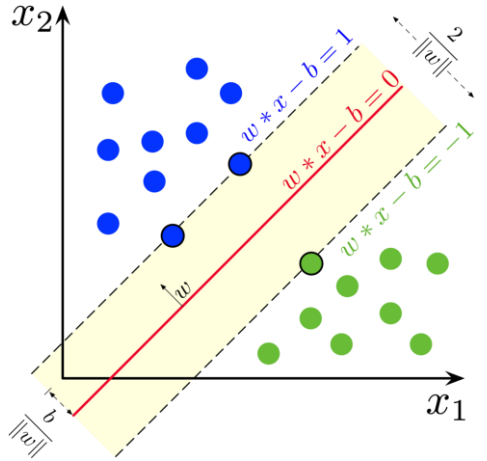
IMAGENET

Precomputed dense SIFT descriptors at 3 scales



Precomputed 1000 codewords running k −means on 1 million randomly selected SIFT descriptors

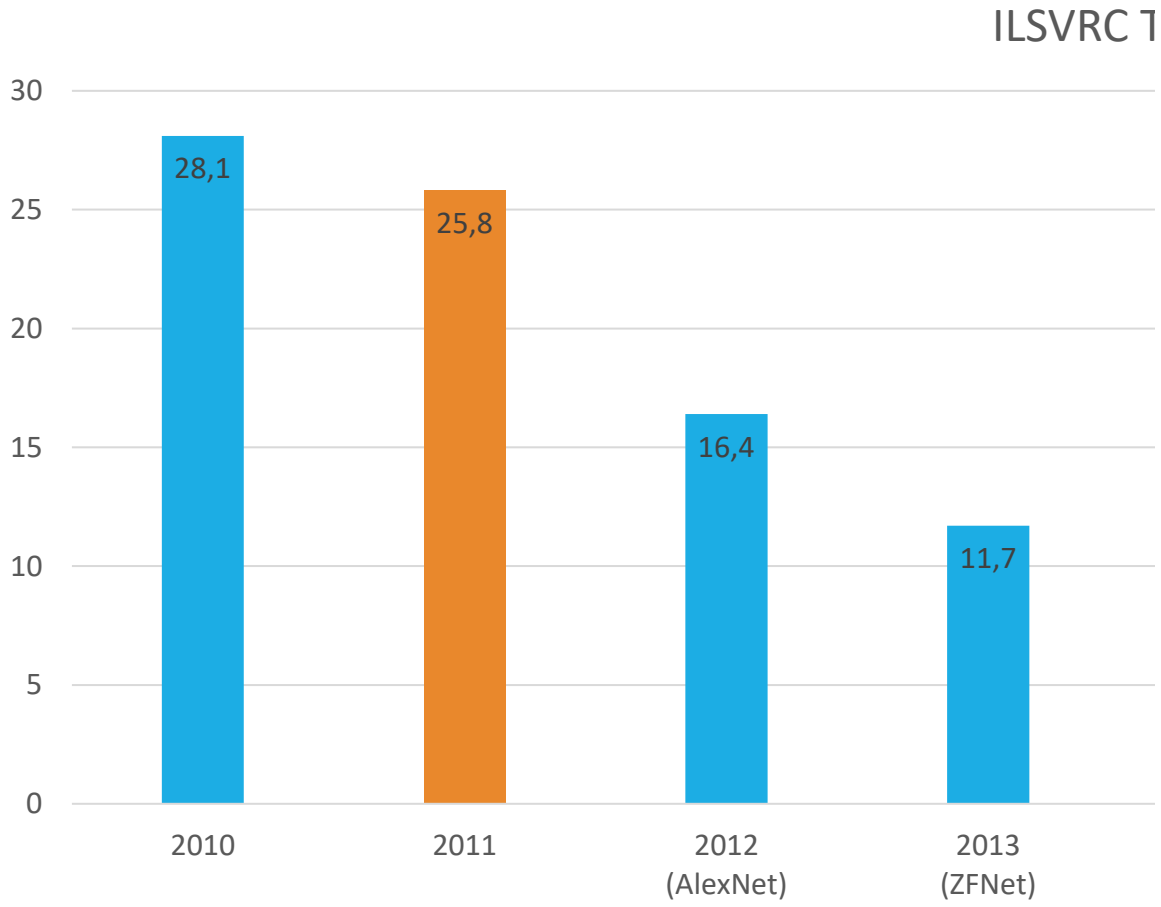
Apply your creativity here



Train your favorite classifier, usually a linear SVM trained with SGD due to problems in scaling up other classifiers

http://www.image-net.org/download-features

ILSVRC11 winning entry



Summary

Low-level feature extraction ≈ 10k patches per image

- SIFT: 128-dim
- color: 96-dim

} reduced to 64-dim with PCA

FV extraction and compression:

- N=1,024 Gaussians, R=4 regions ⇨ 520K dim x 2
- compression: G=8, b=1 bit per dimension

One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

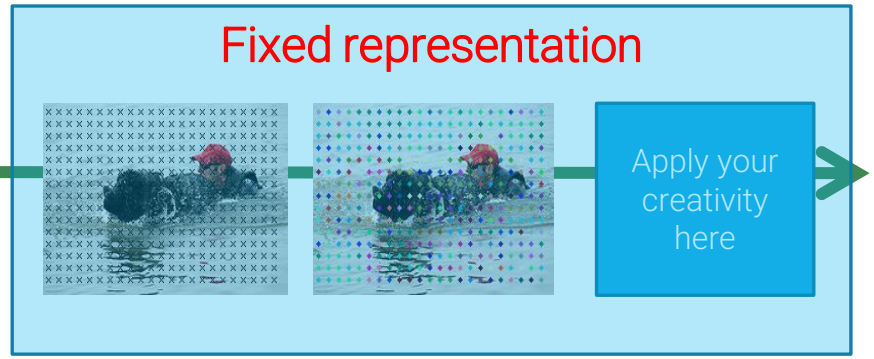
For details, see: Sánchez and Perronnin, "High-dimensional signature compression for large-scale image classification", CVPR'11.

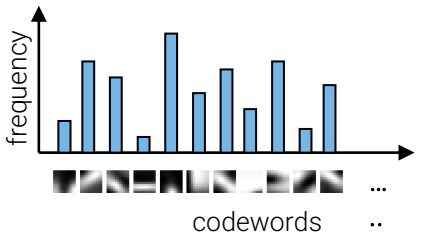
Page 13 F. Perronnin, J. Sánchez, "Compressed Fisher vectors for LSVRC", PASCAL VOC / ImageNet workshop, ICCV, 2011.

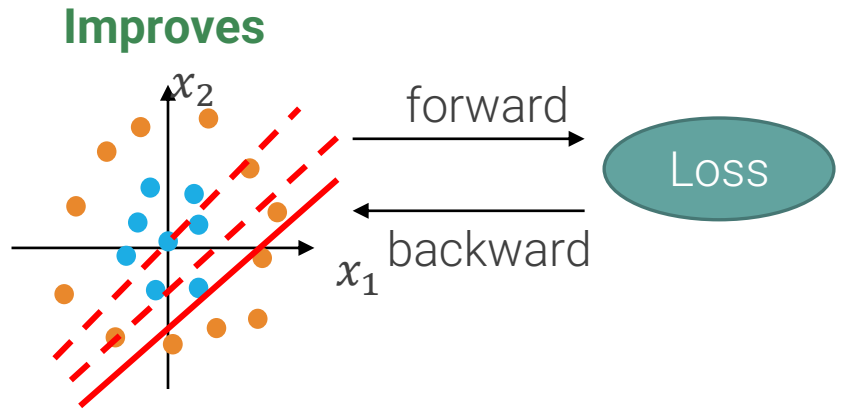
xerox

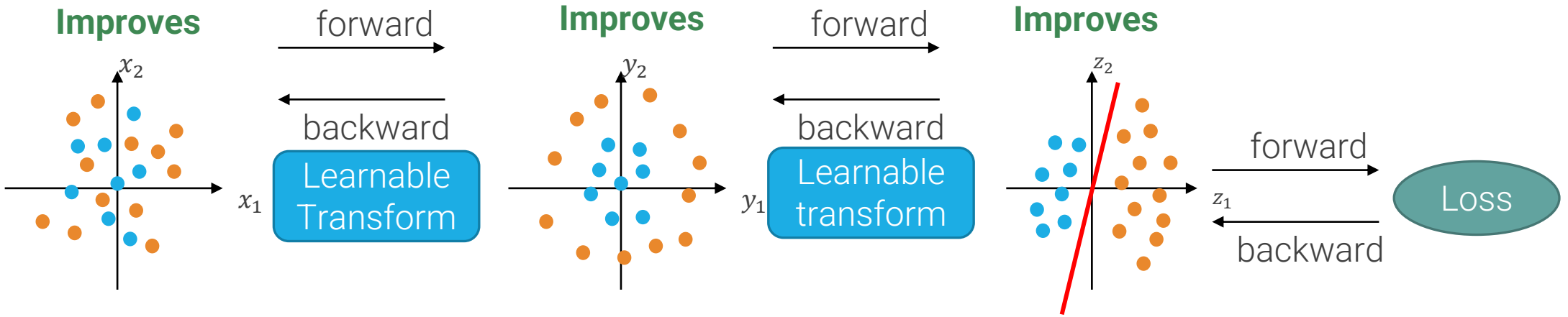
Florent Perronnin, et al., Improving the Fisher kernel for large-scale image classification. In Proc. ECCV, 2010.
http://image-net.org/challenges/LSVRC/2011/ilsvrc11.pdf

Representation learning









Deep learning ≈ Representation learning

Neural networks

Linear classifier

$$f(x;\theta) = Wx + b$$

Neural Network

$$f(x;\theta) = W_2 h + b_2$$

$$= W_2 \phi(W_1 x + b_1) + b_2$$

$$(W_2, b_2, W_1, b_1)$$

3072x1

10xC Cx1 10x1

10xC Cx3072 3072x1 Cx1 10x1

New hyper-parameters

- Dimension of inner representation C
- Activation function ϕ

Why do we insert activation functions?

$$
\begin{aligned}
f(\boldsymbol{x};\theta) &= W_2\boldsymbol{r} + \boldsymbol{b_2} \\
&= W_2\,(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2} \\
&= (W_2\,W_1)\boldsymbol{x} + (W_2\boldsymbol{b_1} + \boldsymbol{b_2}) \\
&= W_{21}\boldsymbol{x} + \boldsymbol{b_{21}}
\end{aligned}
$$

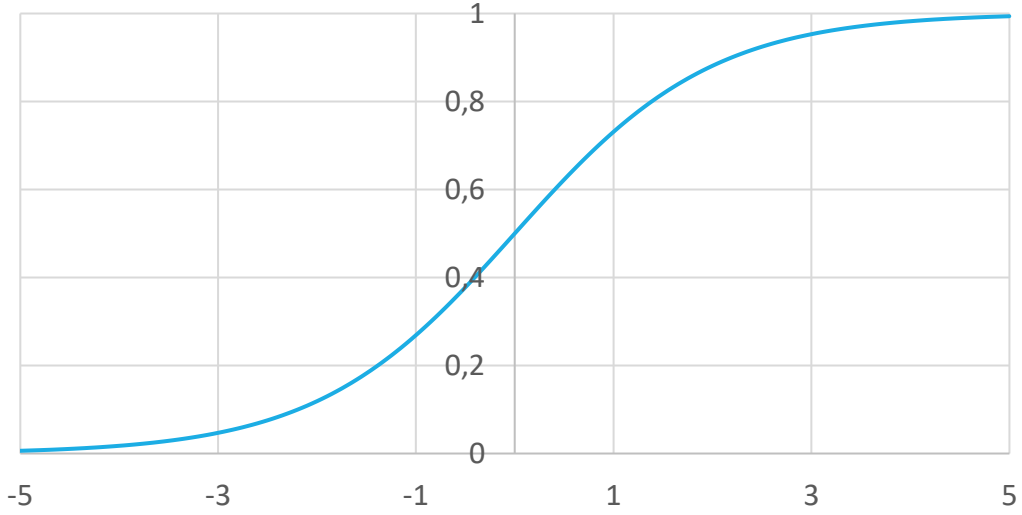
Without activation functions, we end-up again with a linear classifier

Activation functions

A non-linear function, which is applied to every element of the input tensor

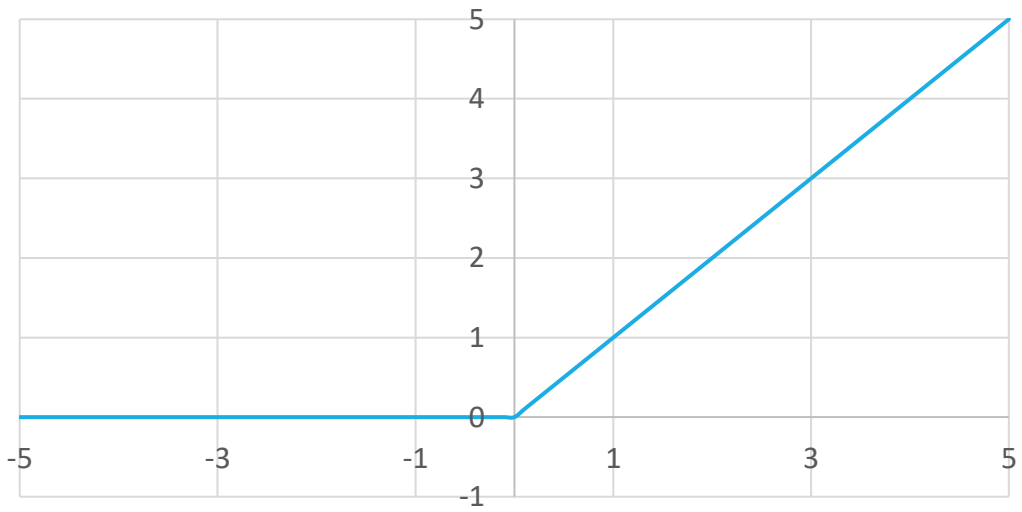
$$f(\boldsymbol{x};\theta) = W_2\phi(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$

(Logistic) Sigmoid



$$\phi(a) = \frac{1}{1+\exp(-a)} = \sigma(a)$$

Rectified Linear Unit (ReLU)



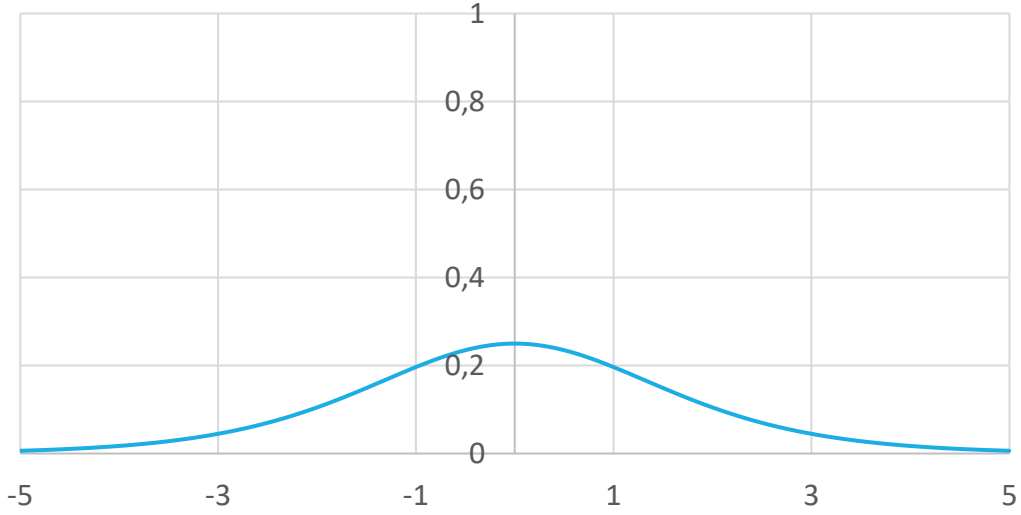
$$\phi(a) = \max(0, a) = ReLU(a)$$

Nair, V. and Hinton, G. E. "Rectified linear units improve Restricted Boltzmann Machines". ICML 2010.
Xavier Glorot; Antoine Bordes; Yoshua Bengio "Deep sparse rectifier neural networks", AISTATS 2011

Activation functions - gradients

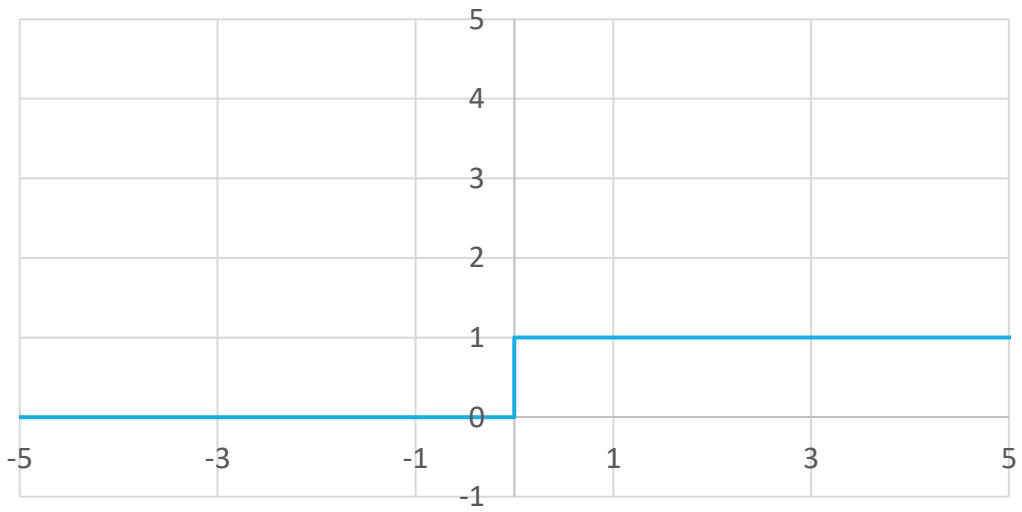
Gradient of the output of activation functions with respect to input is very different between sigmoid and ReLU. The gradient of the sigmoid function saturates when the input is large in absolute value, while ReLU has always gradient “1” for positive values: it is easier to train (deep) networks when using ReLU. Yet, ReLU can give rise to “dead” neurons, if for all inputs the output is negative.

Gradient of sigmoid



$$\frac{d\,\sigma(a)}{da} = \sigma(a)\big(1-\sigma(a)\big)$$

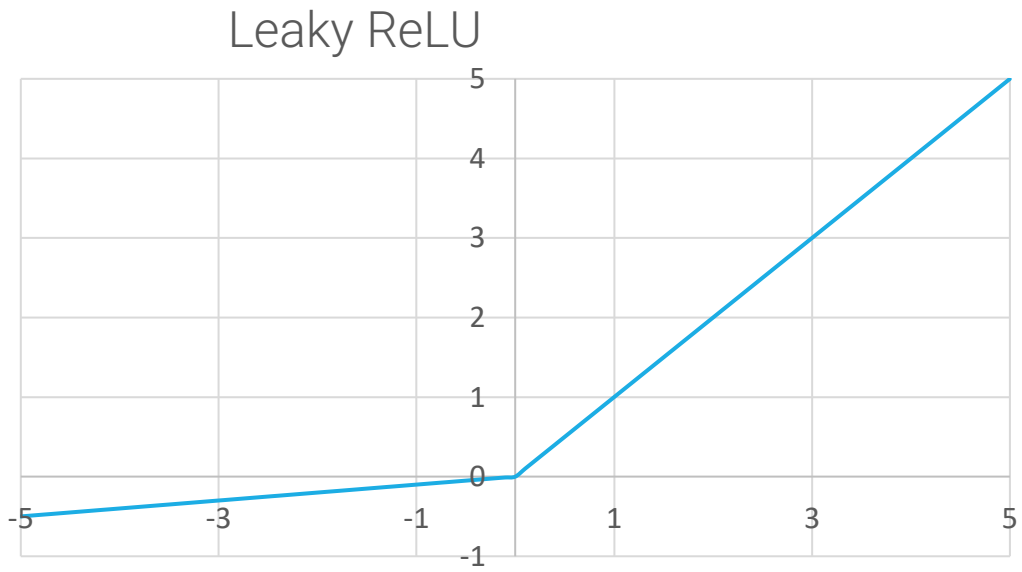
Gradient of ReLU



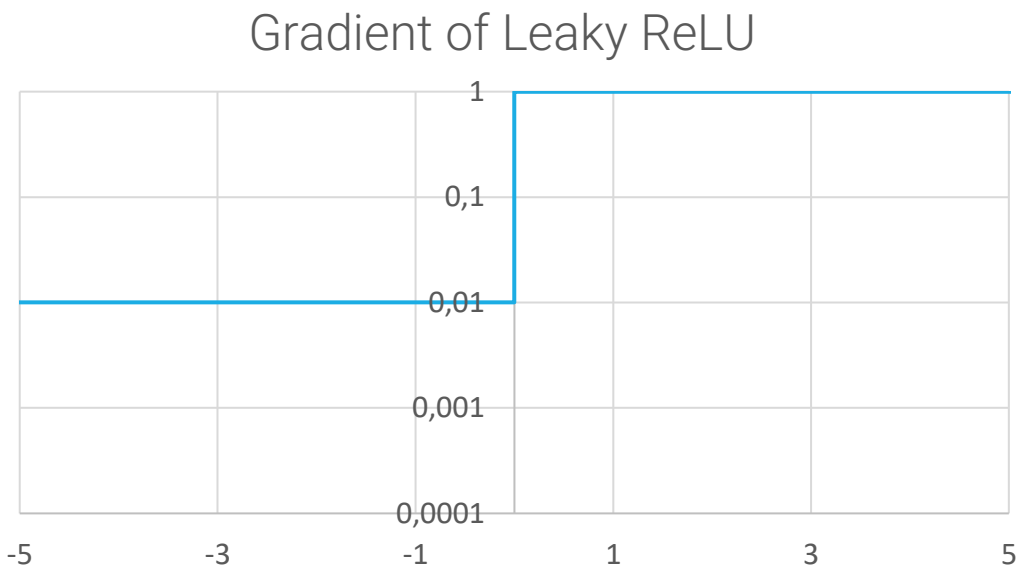
$$\frac{dReLU(a)}{da} = \begin{cases} 1 & if\ a \geq 0 \\ 0 & ow \end{cases}$$

Activation functions – Leaky ReLU

To avoid dead neurons, a small response can be produced also for negative inputs. The simplest variant is called Leaky ReLU, which has a small but non-zero constant gradient also for negative inputs.



$$Leaky\ ReLU(a) = \begin{cases} a & if\ a \geq 0 \\ 0.01\ a & ow \end{cases}$$

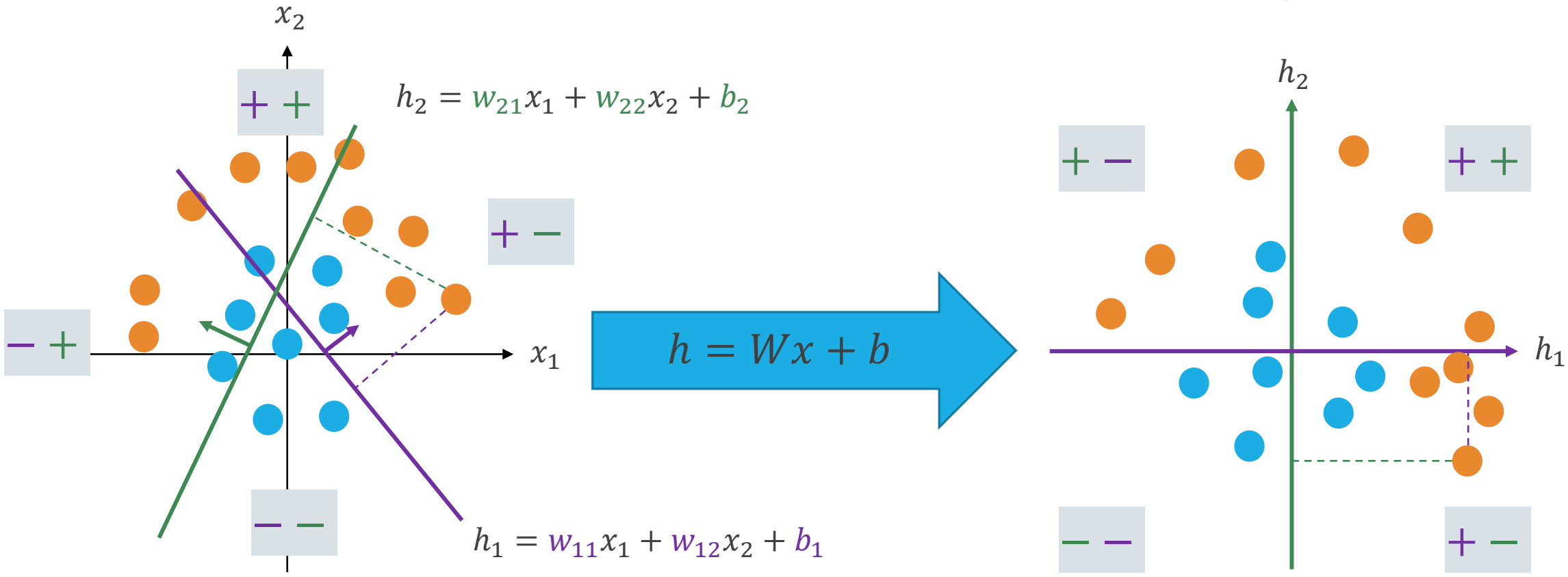


$$\frac{d\ Leaky\ ReLU(a)}{da} = \begin{cases} 1 & if\ a \geq 0 \\ 0.01 & ow \end{cases}$$

Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng. "Rectifier Nonlinearities Improve Neural Network Acoustic Models". ICML 2013

Is ReLU enough?

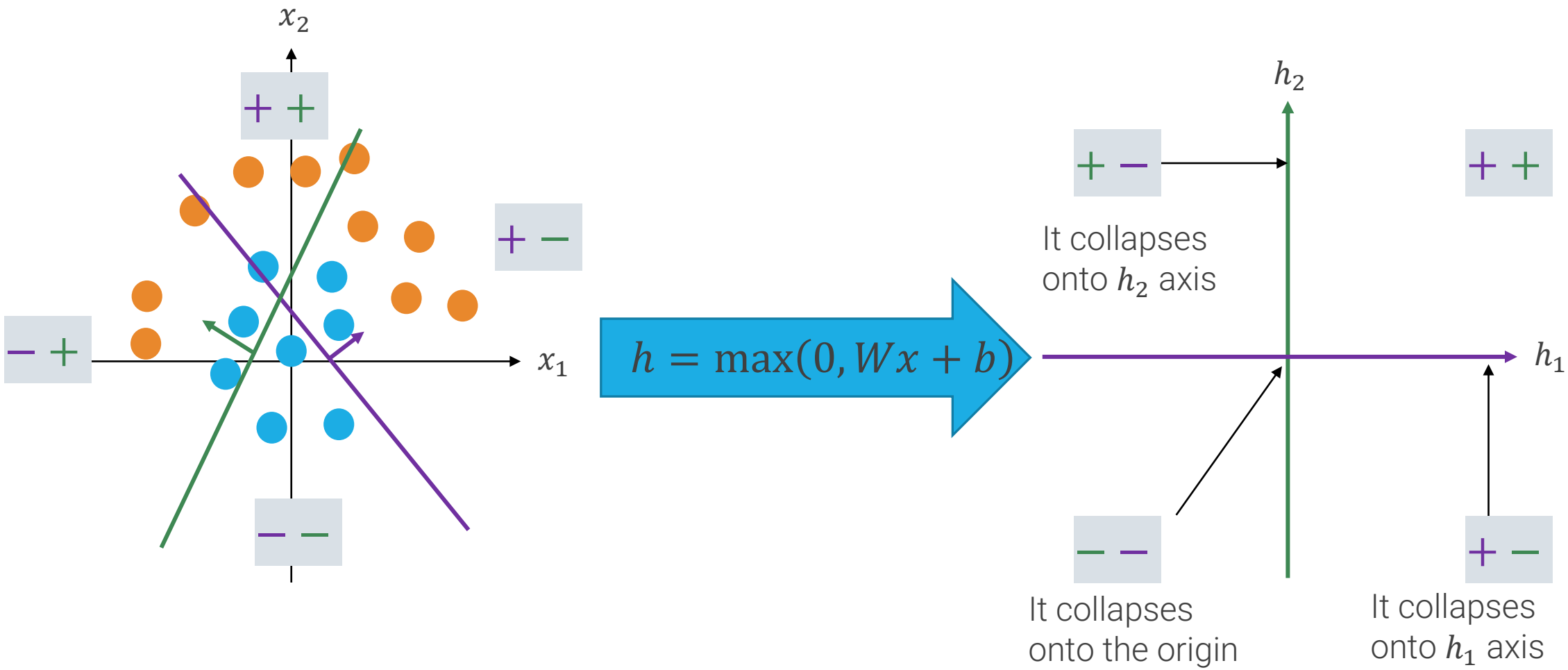
A linear or affine mapping does not change linear separability



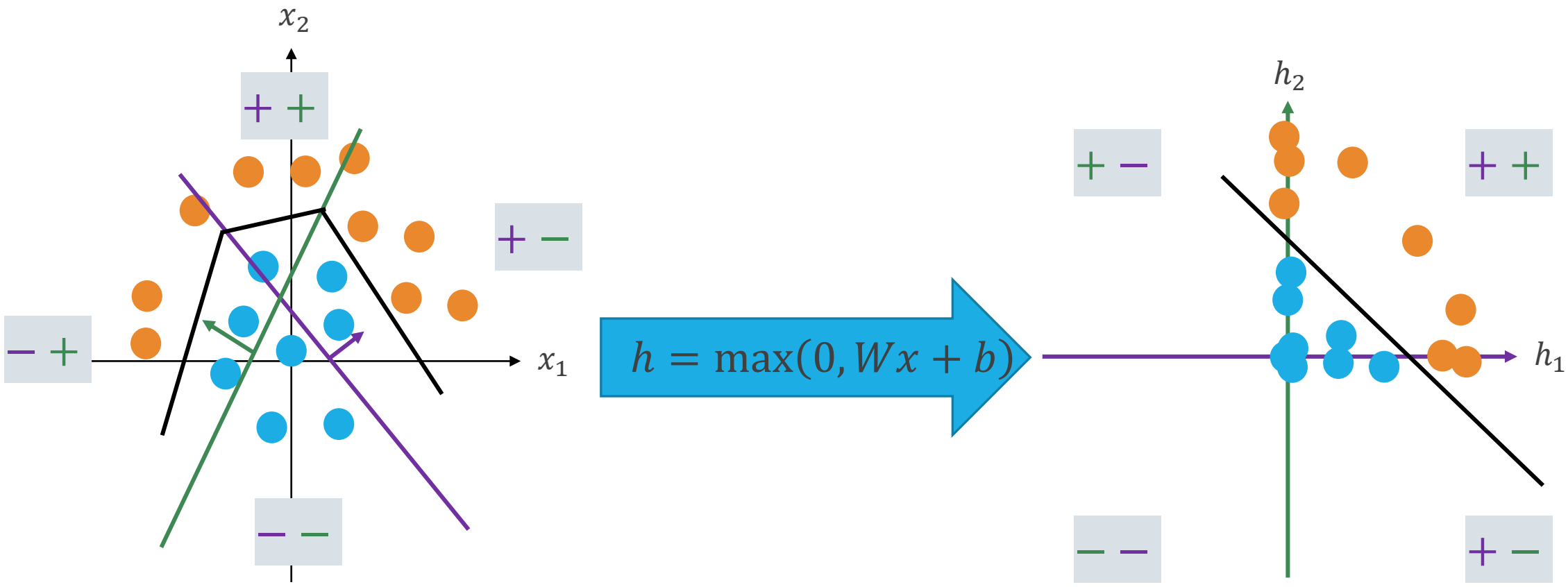
Points not linearly separable in input space

Points not linearly separable in hidden space

Is ReLU enough?



Is ReLU enough?

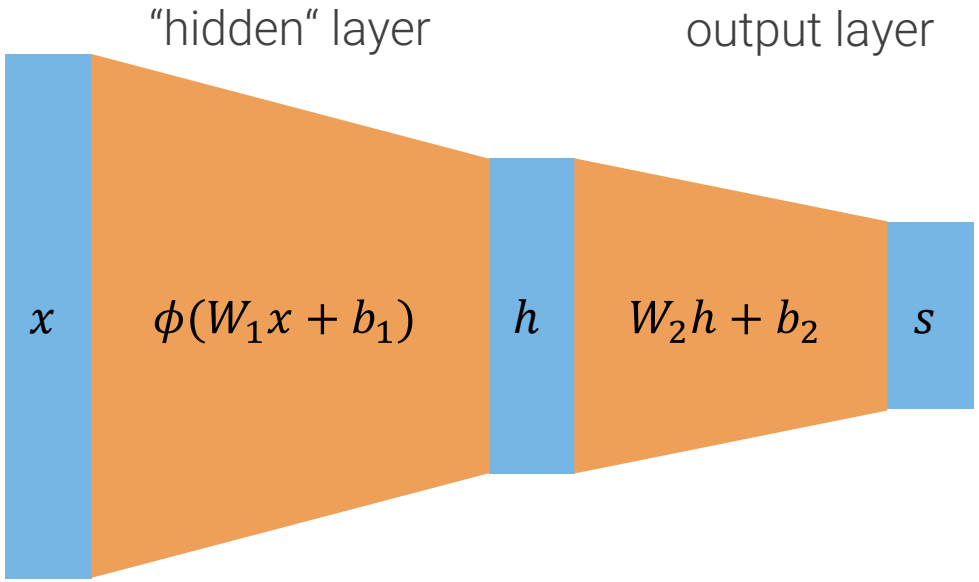


Linear classifier in representation space creates nonlinear classifier in input space

Points linearly separable in hidden space

Terminology

$$f(\boldsymbol{x};\theta) = W_2\boldsymbol{h} + \boldsymbol{b_2}$$
$$= W_2\phi(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2} = \boldsymbol{s}$$



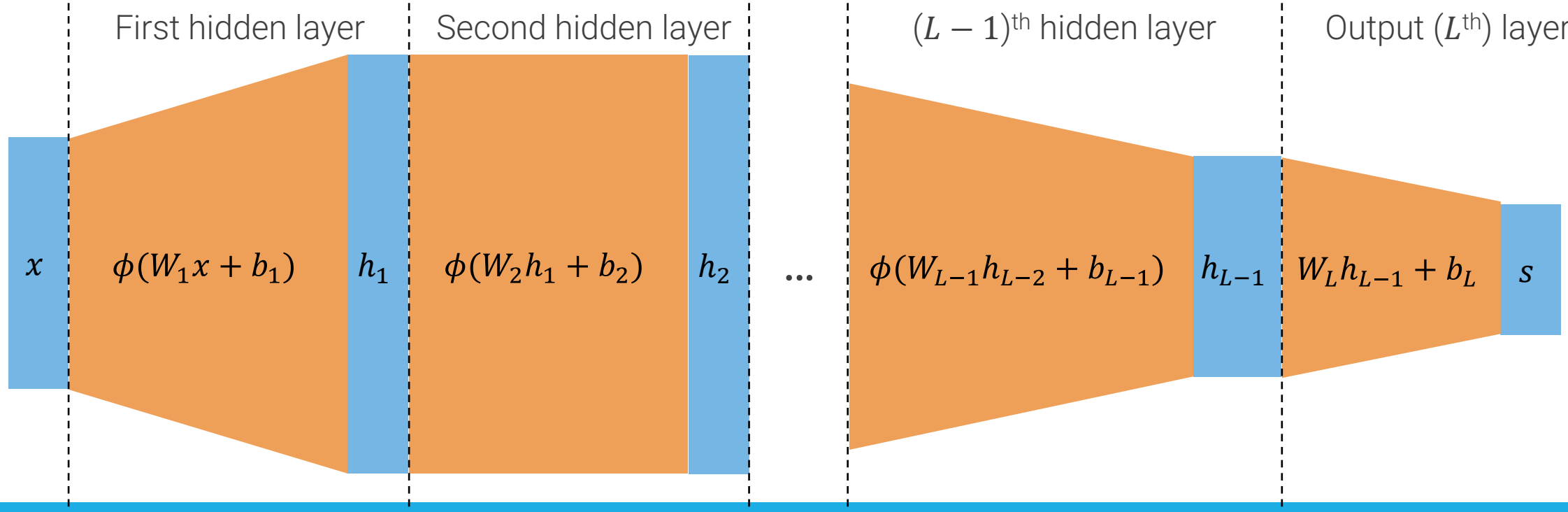
Terminology
x is the input (tensor)
h, s are activations
W_i, b_i (and other numbers we may use to go from one activation to another one) are parameters

Every layer is often called a fully connected (FC) layer, as every element of the input influences every element of the output
A neural network with 2 or more layers is also called a Multi-Layer Perceptron (MLP)

“Deep” neural networks

$$f(\boldsymbol{x};\theta) = W_L \boldsymbol{h_{L-1}} + \boldsymbol{b_L}$$
$$= W_L \phi(W_{L-1} \boldsymbol{h_{L-2}} + \boldsymbol{b_{L-1}}) + \boldsymbol{b_L}$$
$$= W_L \phi(W_{L-1} \phi(\dots \phi(W_1 \boldsymbol{x} + \boldsymbol{b_1}) \dots) + \boldsymbol{b_{L-1}}) + \boldsymbol{b_L} = \boldsymbol{s}$$

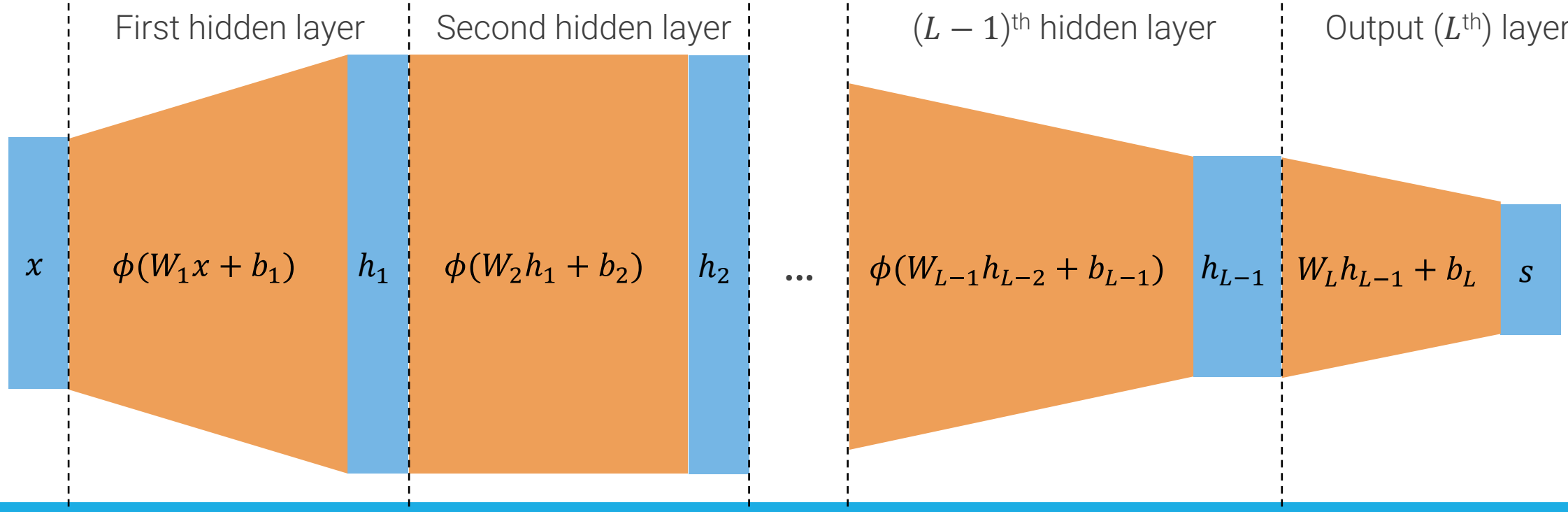
Number of layers is the depth of the network
If $L > 2$, we consider the network to be “deep”



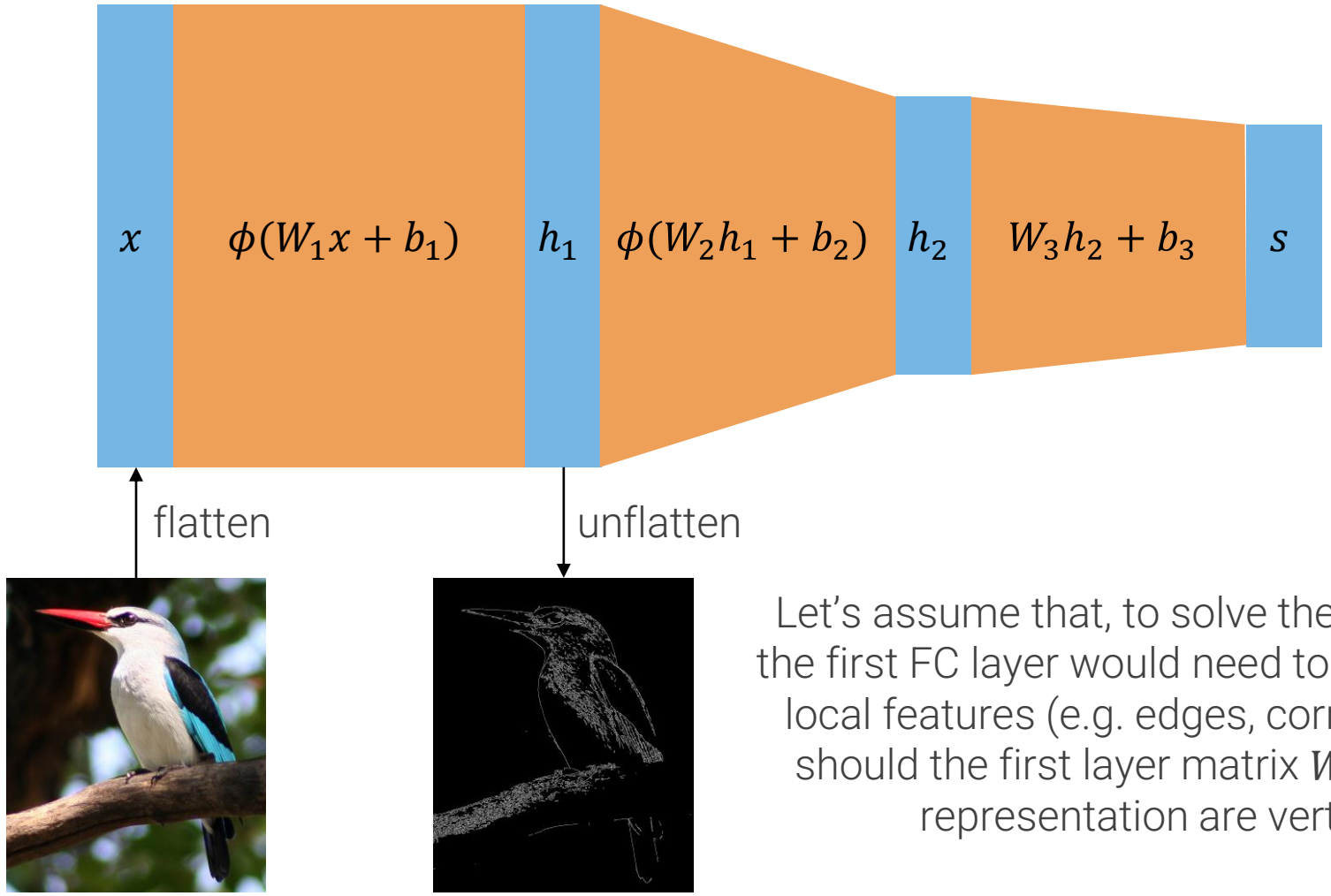
“Width” of neural networks

$$f(\boldsymbol{x};\theta) = W_L \boldsymbol{h_{L-1}} + \boldsymbol{b_L}$$
$$= W_L \phi(W_{L-1} \boldsymbol{h_{L-2}} + \boldsymbol{b_{L-1}}) + \boldsymbol{b_L}$$
$$= W_L \phi(W_{L-1} \phi(\dots \phi(W_1 \boldsymbol{x} + \boldsymbol{b_1}) \dots) + \boldsymbol{b_{L-1}}) + \boldsymbol{b_L} = \boldsymbol{s}$$

The number of activations computed by each layer, i.e. the length of h_i, is the width of the network



Limits of fully connected layers



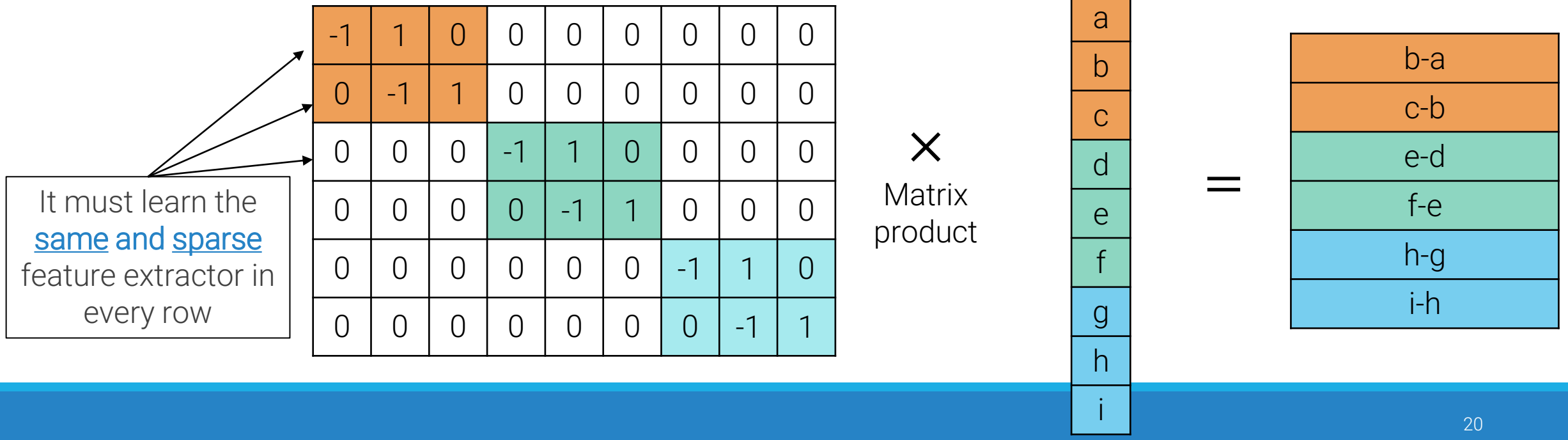
Let's assume that, to solve the classification task, the first FC layer would need to detect some kind of local features (e.g. edges, corners, blobs..). What should the first layer matrix W_1 be if a "good" h_1 representation are vertical edges?

Vertical edge detection with FC layer

If the input image has size $H \times \mathrm{W}$, the layer requires

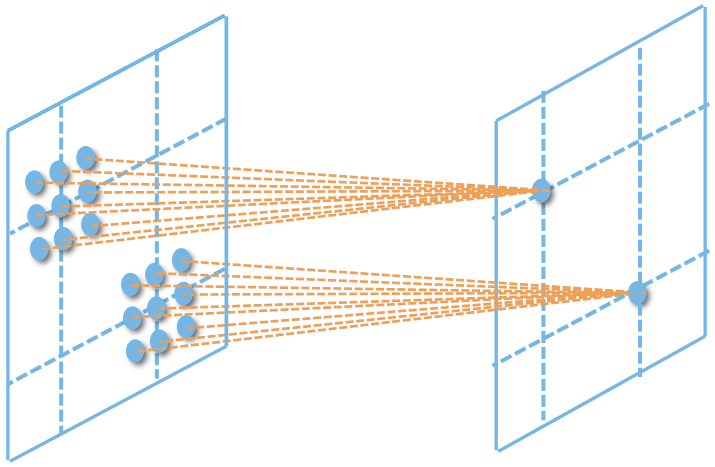
- $(H \times W) \times (H \times (W-1)) \approx H^2W^2$ parameters/memory
- $2(H \times W) \times (H \times (W-1)) \approx 2H^2W^2$ multiply-add ops (FLOPS)

e.g. for a 224×224 image, more than 2.5×10^9 params and more than 5×10^9 floating point operations, i.e. **5 Giga FLOPs**



Convolutions/correlations

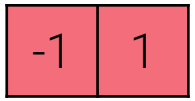
In traditional image processing and computer vision, we usually rely on convolution/correlation with hand-crafted filters (kernels) to process images (e.g. denoise or detect local features).



- Unlike linear layers, in a convolution, the input and output are not flattened, i.e. convolution preserves the spatial structure of images.
- Unlike linear layers, a convolution processes only a – small – set of neighboring pixels at each location. In other words, each output unit is connected only to local input units. This realizes a so called local receptive field.
- Unlike linear layers, the parameters associated with the connections between an output unit and its input neighbors are the same for all output units. Thus, parameters are said to be shared and the convolution detect structures regardless of the input position.

Convolutions embody inductive biases dealing with the structure of images: images exhibit informative local patterns that may appear everywhere across an image.

Vertical edge detection with correlation/convolution



★

correlation

If the input image has size $H \times \mathbf{W}$, it requires

- 2 parameters
- $3 \times (H \times (W-1)) \approx 3HW$ multiply-add ops, i.e. 150 K flops for a 224×224 image

-1	1	0	0	0	0	0	0	0
0	-1	1	0	0	0	0	0	0
0	0	0	-1	1	0	0	0	0
0	0	0	0	-1	1	0	0	0
0	0	0	0	0	0	-1	1	0
0	0	0	0	0	0	0	-1	1

Input image

a	b	c
d	e	f
g	h	i

Flatten image

a
b
c
d
e
f
g
h
i

×

matrix product

=

b-a	c-b
e-d	f-e
h-g	i-h

Flatten | unflatten

=

b-a
c-b
e-d
f-e
h-g
i-h

Correlation or convolution?

Convolution between an image I and a kernel K would actually use a flipped kernel

$$[I * K](i,j) = \sum_l \sum_m I(l,m)K(i-l, j-m)$$

and, in this case, it is commutative

$$[I * K](i,j) = [K * I](i,j) = \sum_l \sum_m K(l,m)I(i-l, j-m)$$

The proper name for what we use in neural networks is (cross-)correlation

$$[K \star I](i,j) = \sum_l \sum_m K(l,m)I(i+l, j+m)$$

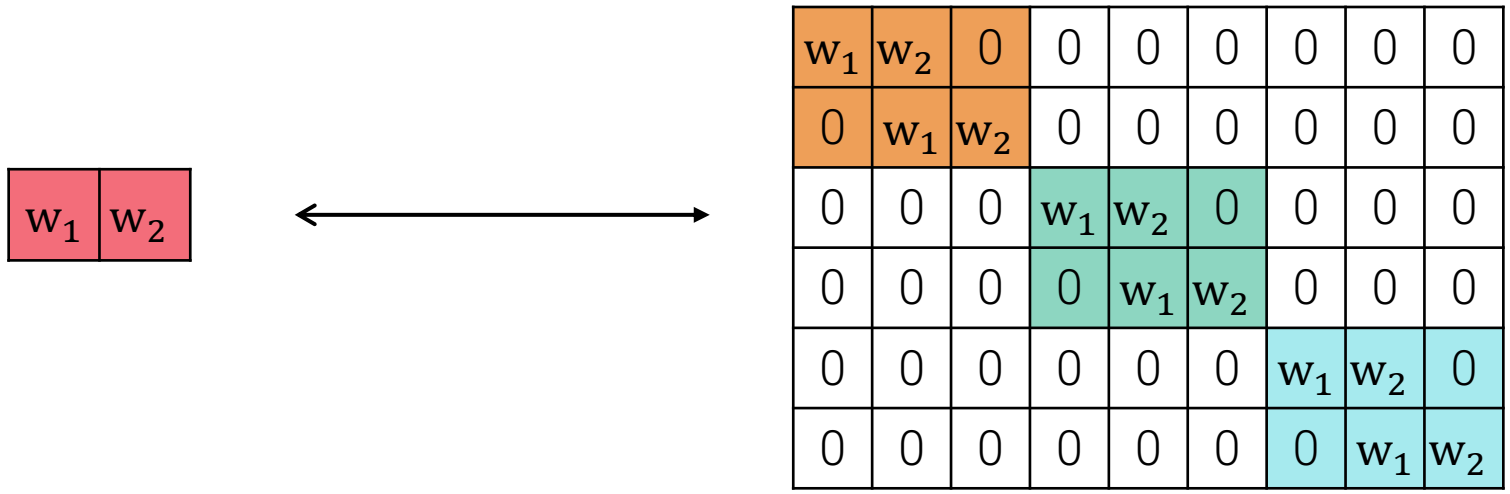
Notice that, at every location, it can be seen as the dot product between the kernel and an image patch.

Convolution as matrix multiplication

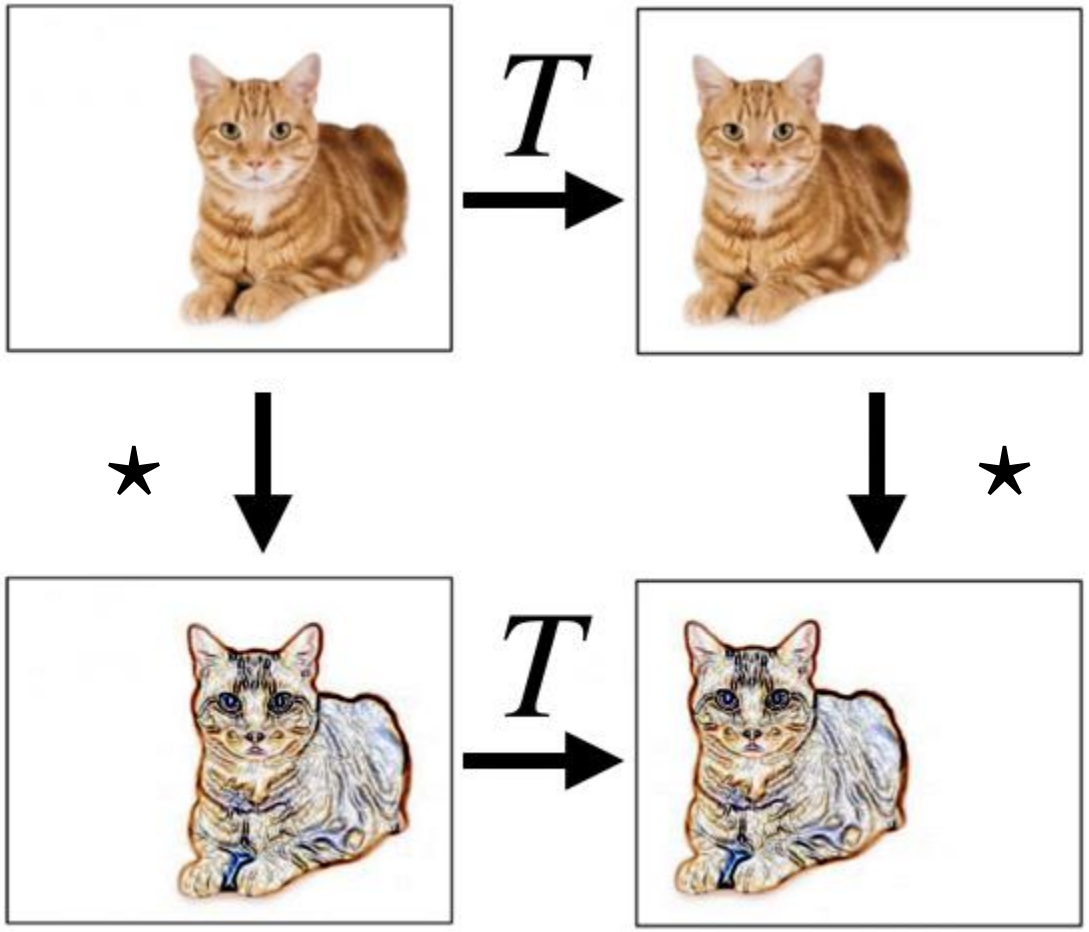
Convolution/correlation can be interpreted as matrix multiplication, if we reshape inputs and outputs.

The resulting matrix is still a linear operator, which:

1. shares parameters across its rows
2. is sparse, i.e. each output unit is connected only to a small set of neighboring input entries
3. seamlessly adapts to varying input sizes
4. is equivariant to translations of the input, i.e. translation of the output of the convolution is equivalent to computation of the convolution on the translated input



Equivariance



Equivariance with respect to translation means that we can swap translation and correlation and get the same result

$$T(x) \star K = T(x \star K)$$

It is another form of inductive bias that improves data efficiency with respect to linear layers thanks to parameter sharing: we do not have to see features (e.g. edges) at all locations in the training dataset to be able to learn to detect them effectively.

Note that correlation is not equivariant with respect to rotation or scale.

http://irregulardeep.org/An-introduction-to-Invariant-Graph-Networks-(1-2)/

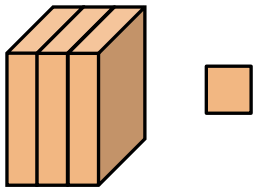
Multiple input channels

Images have 3 channels, so convolution kernels will be 3-dimensional tensors of size $3 \times H_K \times W_K$ and

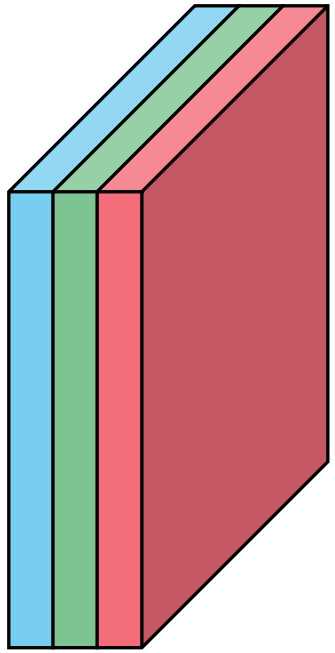
$$[K * I](j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} K_n(m,l) I_n(j-m, i-l) + \boldsymbol{b}$$

This is still a 2D convolution, but over vector-valued functions, not a 3D convolution (notice we do not slide over channels)

As usual, we compute an affine function, so we also have a bias term



Filter or kernel,
e.g. $\mathbf{3 \times 5 \times 5}$

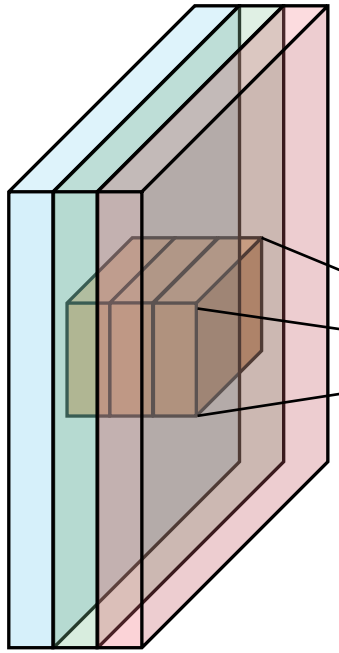


Input image,
e.g. $\mathbf{3 \times 32 \times 32}$

Filters and input depth always match, and the third dimension of a filter is usually implicit, i.e. we refer to this convolution as a "5 by 5 convolution", but it has $5 \times 5 \times 3 = 75$ parameters (76 with the bias), not 25

Output activation

$$[K * I](j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} K_n(m,l) I_n(j-m, i-l) + b$$



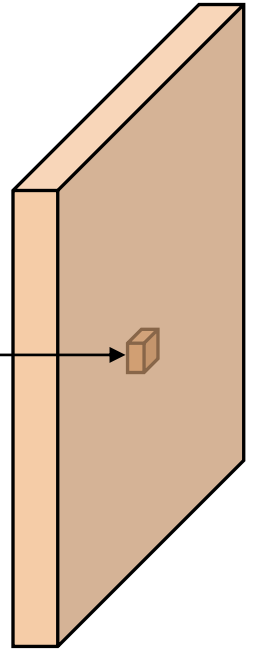
Input image,
e.g. $\mathbf{3 \times 32 \times 32}$

Filter or kernel,
e.g. $\mathbf{3 \times 5 \times 5}$

When the filter is at position (j,i) on the input image, we compute one number, i.e. the dot product between the kernel and the patch of the input image plus the bias

By sliding the filter over the image, we get a single-valued output image, which is the output activation

It is also called feature map, activation map, etc…

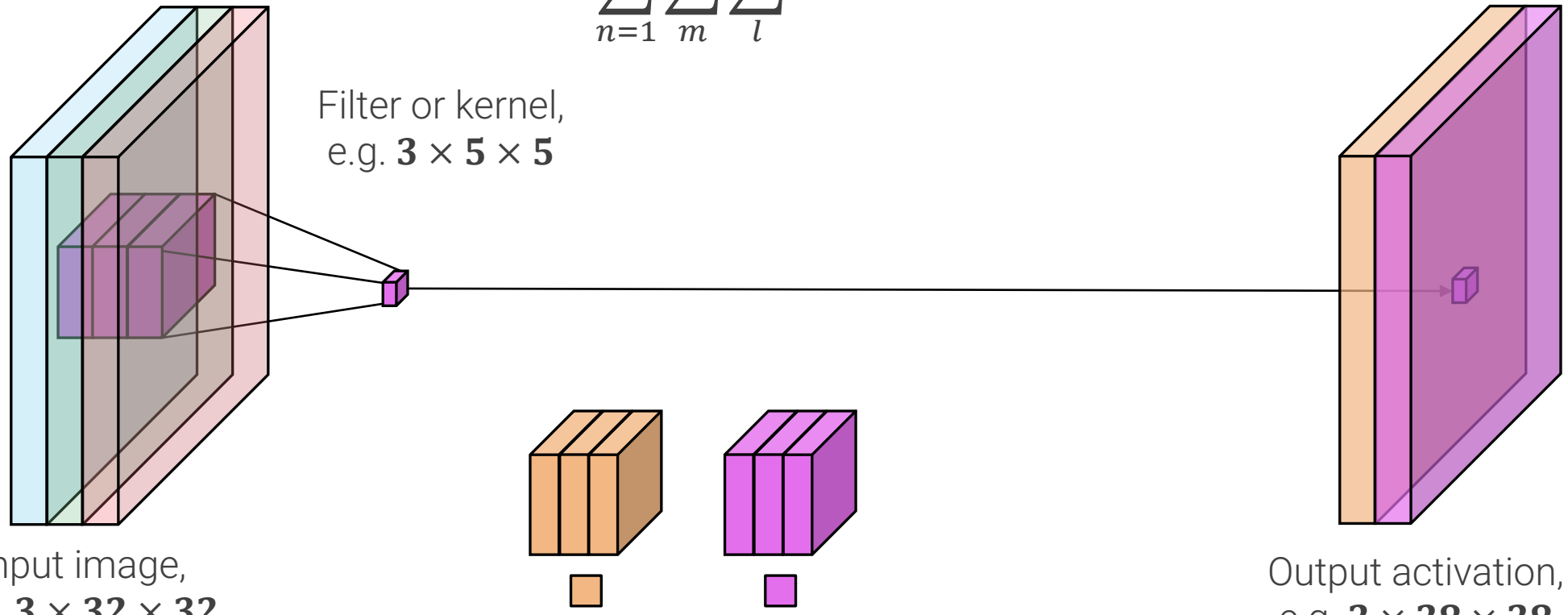


Output activation,
e.g. $\mathbf{1 \times 28 \times 28}$

Multiple output channels

We can repeat the same operation with a second filter, with different weights, e.g. a filter that detects horizontal edges instead of vertical ones

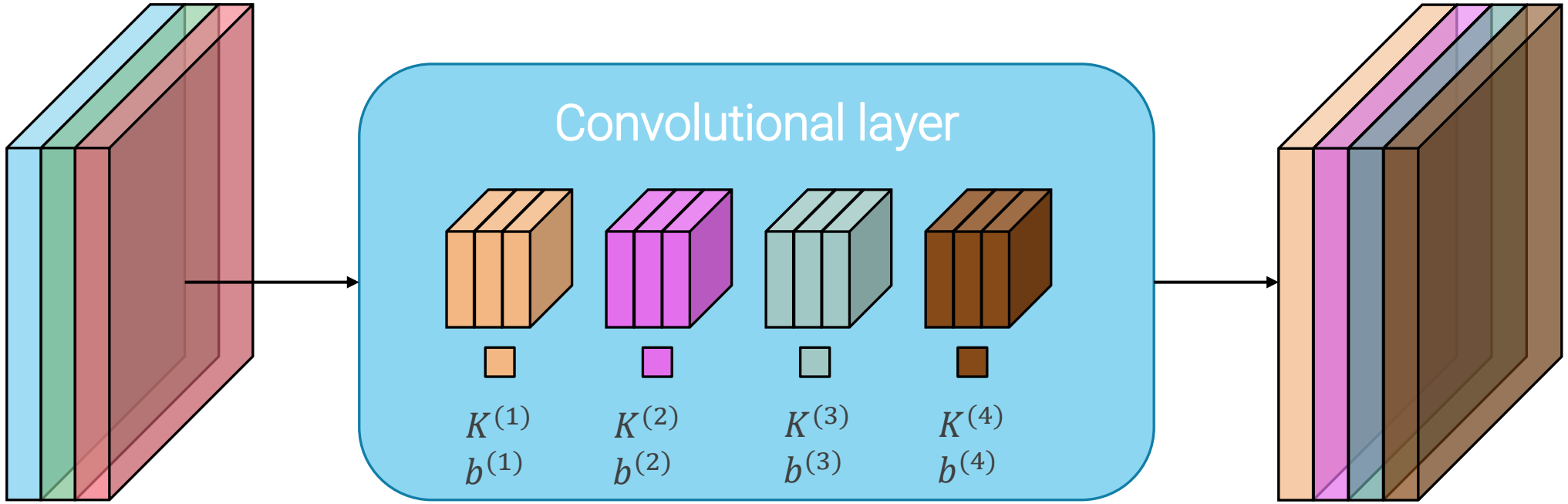
$$[\boldsymbol{K}^{(2)} * I](j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} \boldsymbol{K}_{\boldsymbol{n}}^{(2)}(m,l) I(j-m, i-l) + \boldsymbol{b}^{(2)}$$



Convolutional layer

If we have **4** filters, each of size $\mathbf{3 \times 5 \times 5}$, we can describe the overall operation realized by the layer as

$$[K * I]_k(j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} \boldsymbol{K_n^{(k)}}(m,l) I(j-m, i-l) + b^{(k)} \qquad k = 1, \dots, 4$$



Input image,
e.g. $\mathbf{3 \times 32 \times 32}$

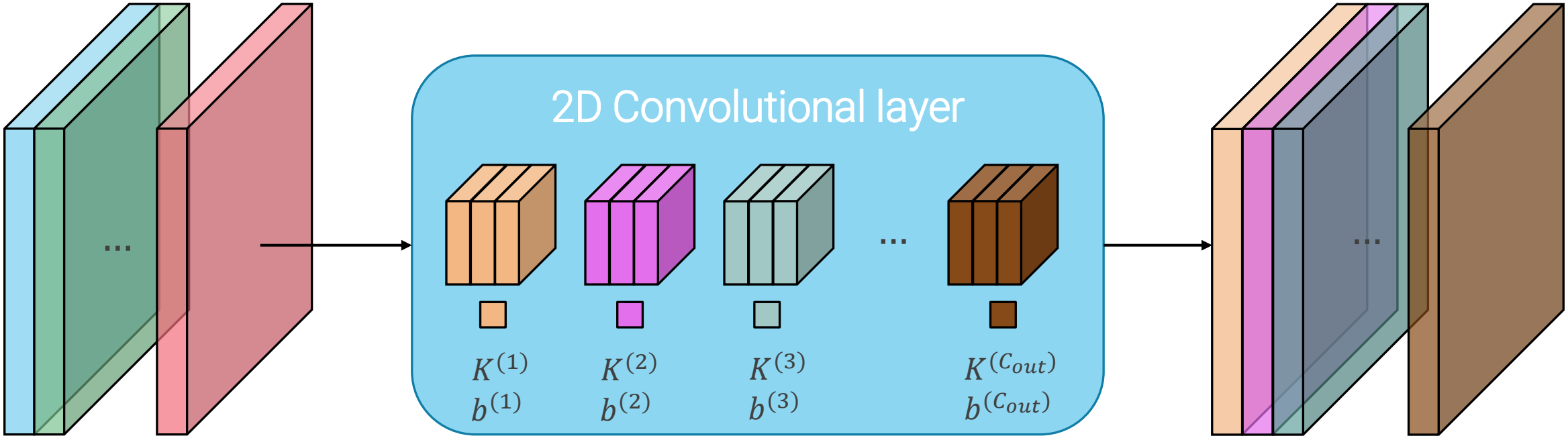
Conv 2D filter
e.g. $\mathbf{4 \times 3 \times 5 \times 5}$

Output activation,
e.g. $\mathbf{4 \times 28 \times 28}$

Convolutional layer

In the general case, we compute C_{out} convolutions between vector-valued kernels and input activations

$$[K * I]_k(j,i) = \sum_{n=1}^{C_{in}} \sum_{m} \sum_{l} K_n^{(k)}(m,l) I_n(j-m, i-l) + b^{(k)} \qquad k = 1, \ldots, C_{out}$$

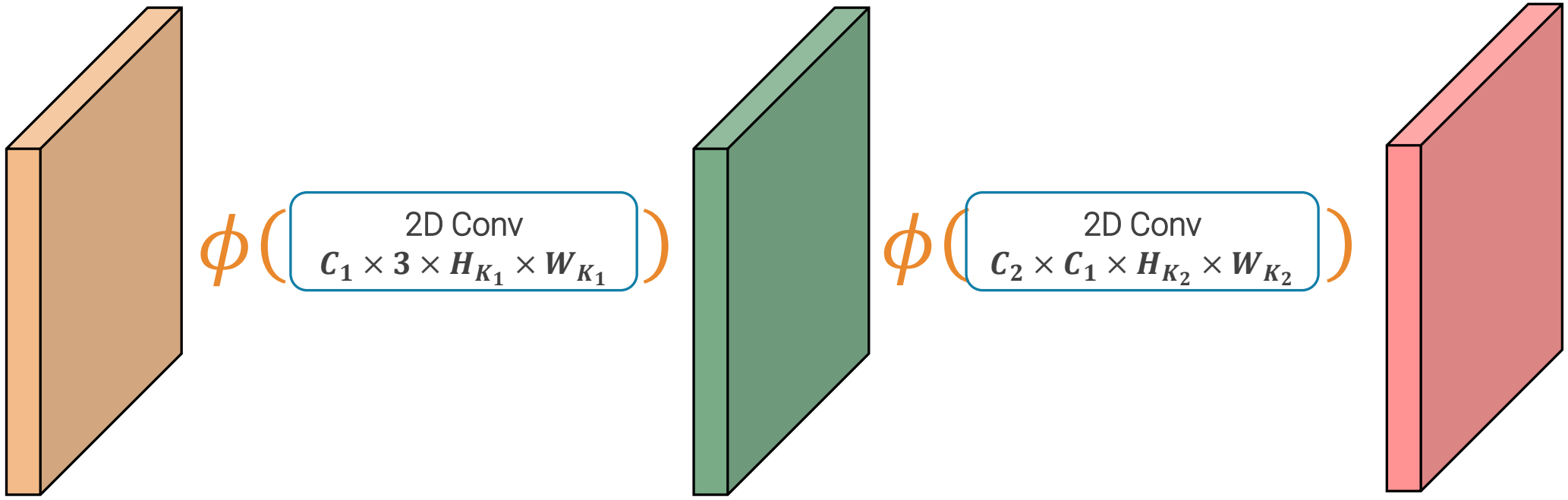


Input activation
$C_{in} \times H_{in} \times W_{in}$

Conv 2D filter
$C_{out} \times C_{in} \times H_K \times W_K$

Output activation
$C_{out} \times H_{out} \times W_{out}$

Multiple convolutional layers



Input image
$3 \times H_{in} \times W_{in}$

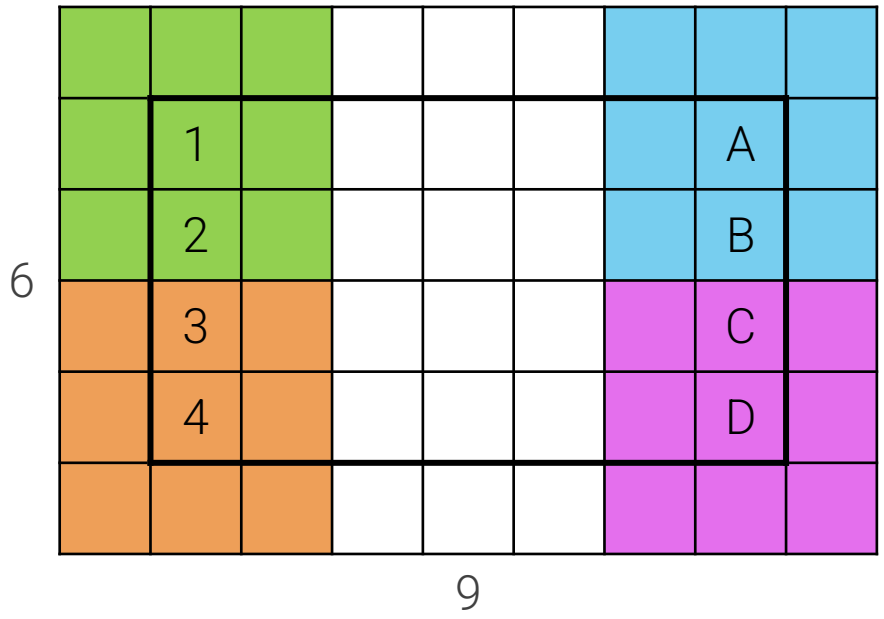
First activation
$C_1 \times H_1 \times W_1$

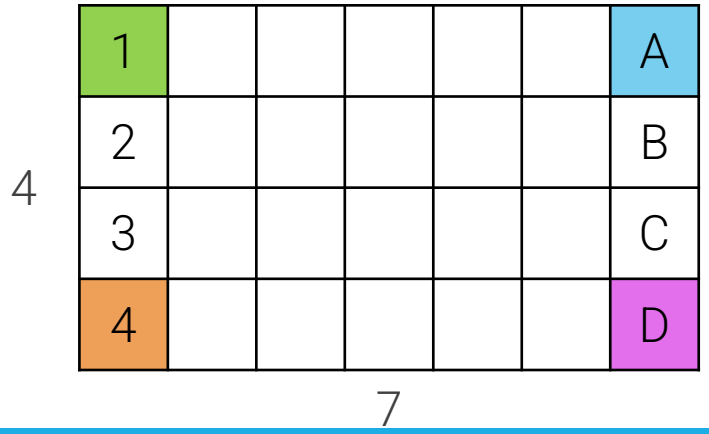
Second activation
$C_2 \times H_2 \times W_2$

We have seen that convolutional layers can be interpreted as a constrained form of linear layers.

Hence, they follow the same rule: to meaningfully compose them, we need to insert non-linear activation functions between them.

Relationship between spatial dimensions





$$H_{in} \times W_{in} = 6 \times 9$$

$$H_k \times W_k = 3 \times 3$$

$$H_{out} \times W_{out} = 4 \times 7$$

In general

$$H_{out} = H_{in} - H_k + 1$$

$$W_{out} = W_{in} - W_k + 1$$

If we stack several convolution layers, feature maps will shrink after each layer. The absence of padding is also referred to as padding="valid".

Zero Padding

0	0	0	0	0	0	0	0	0	0	0
0	1									0
0	2									0
0	3									0
0	4									0
0	5									0
0	6									0
0	0	0	0	0	0	0	0	0	0	0

6 + 2

9+2

Common "solution" is to add zero padding around the original image

$H_{in} \times W_{in} = 6 \times 9$

$H_K \times W_K = 3 \times 3, \boldsymbol{P = 1}$

$H_{out} \times W_{out} = 6 \times 9$

In general

$$H_{out} = H_{in} - H_K + 1 + 2P$$

$$W_{out} = W_{in} - W_K + 1 + 2P$$

Usually $P = \frac{(H_K - 1)}{2}$ to have output with the same size of input (referred to also as padding="same").

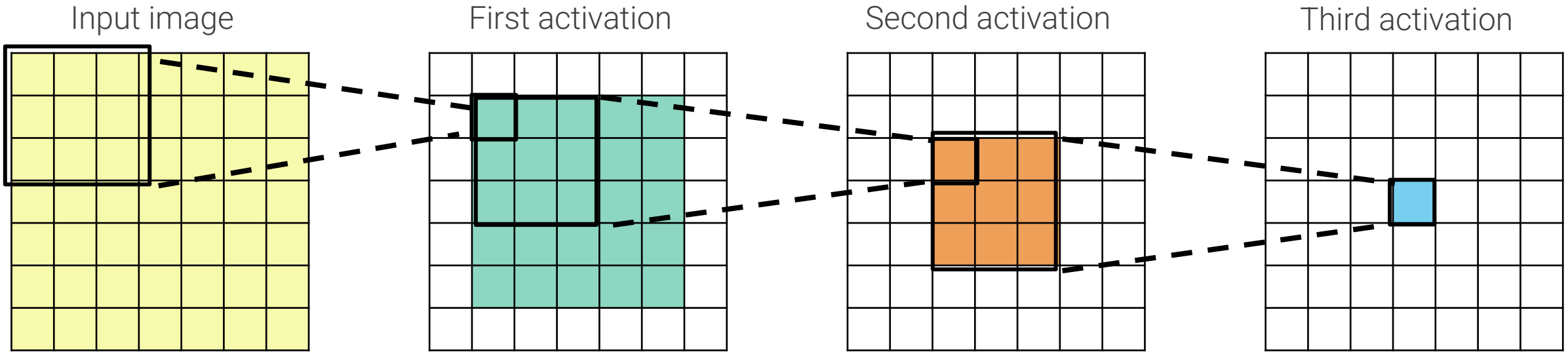
Receptive fields

The input pixels affecting a hidden unit are called its receptive field.

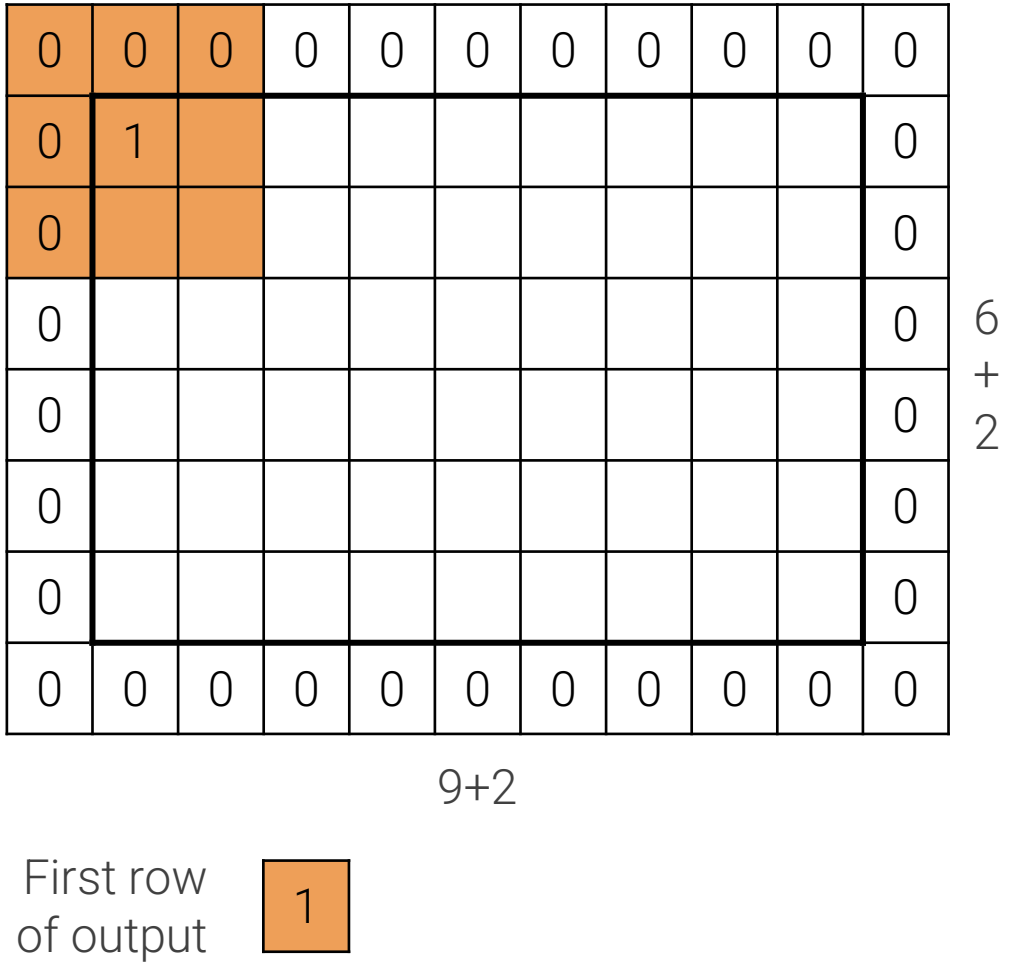
For instance, if we apply a $H_K \times W_K$ kernel size at each layer, the receptive field of an element in the L-th activation has size

$$r_L = [1 + L(H_K - 1)] \times [1 + L(W_K - 1)]$$

Each side grows linearly with the number of layers $L \rightarrow$ to compute features corresponding to a large portion of the input, we would need too many layers. To obtain larger receptive fields with a limited number of layers, we down-sample the activations inside the network.



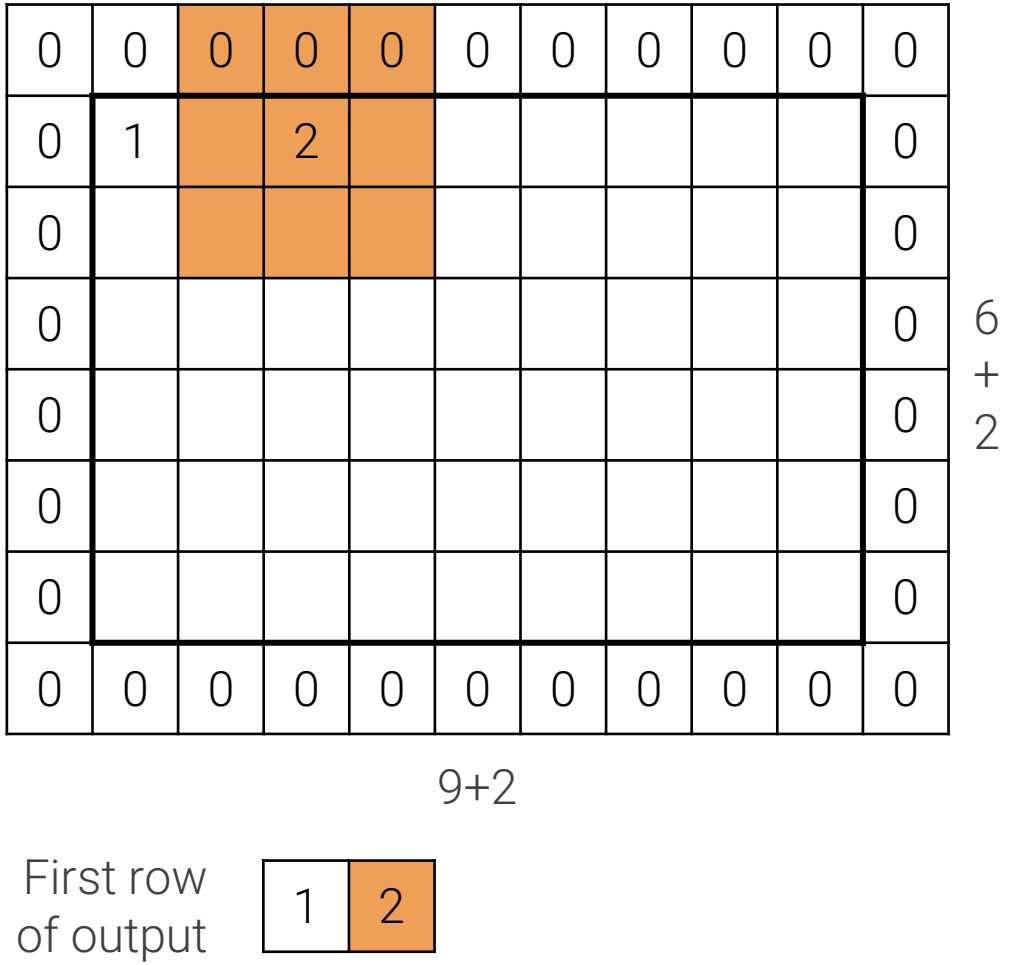
Strided convolutions



$H_{in} \times W_{in} = 6 \times 9$

$H_k \times W_k = 3 \times 3,\ P = 1,\ S = 2$

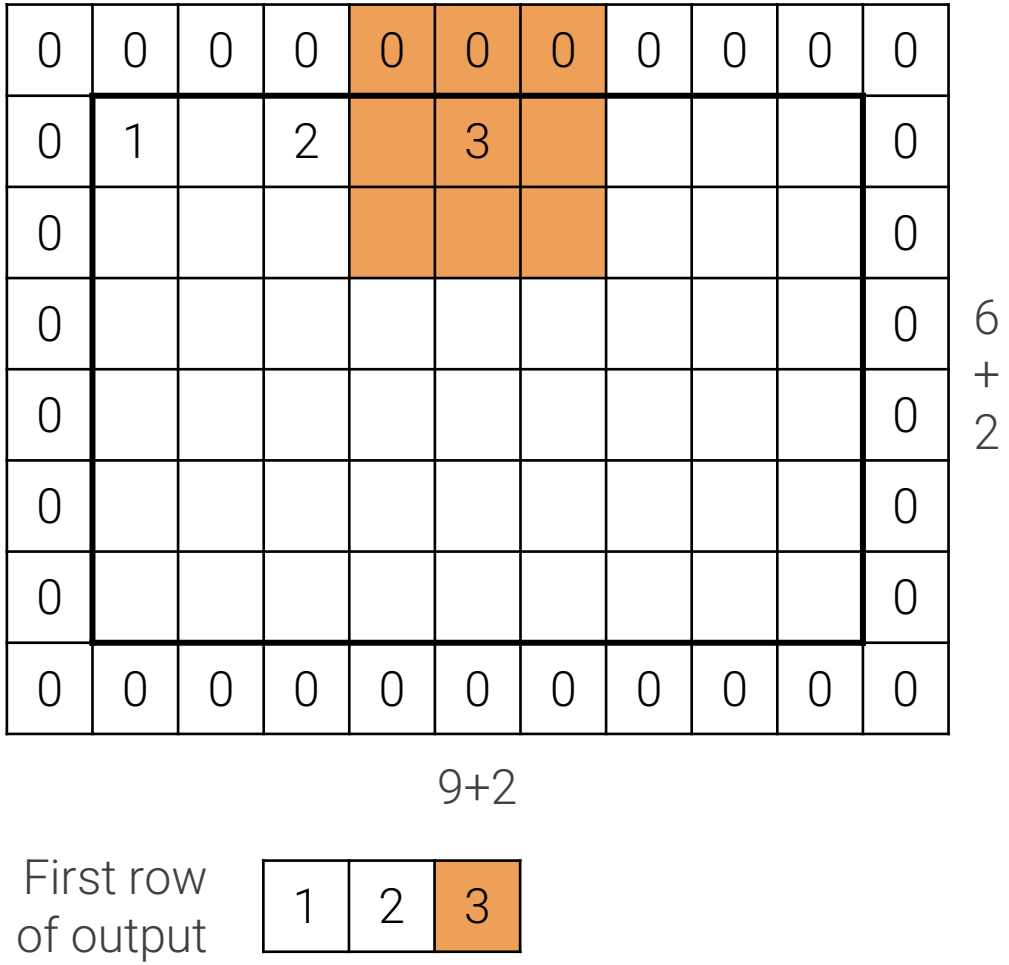
Strided convolutions



$H_{in} \times W_{in} = 6 \times 9$

$H_k \times W_k = 3 \times 3,\ P = 1,\ S = 2$

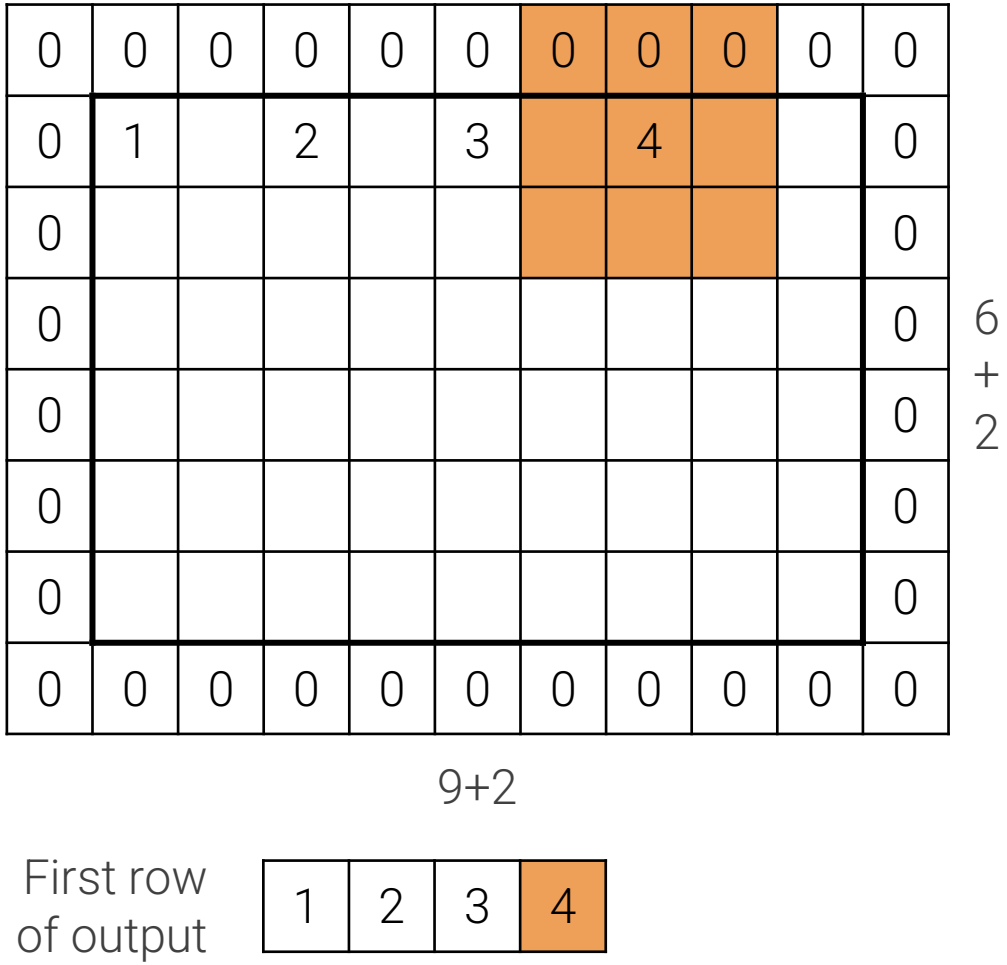
Strided convolutions



$H_{in} \times W_{in} = 6 \times 9$

$H_k \times W_k = 3 \times 3,\ P = 1,\ S = 2$

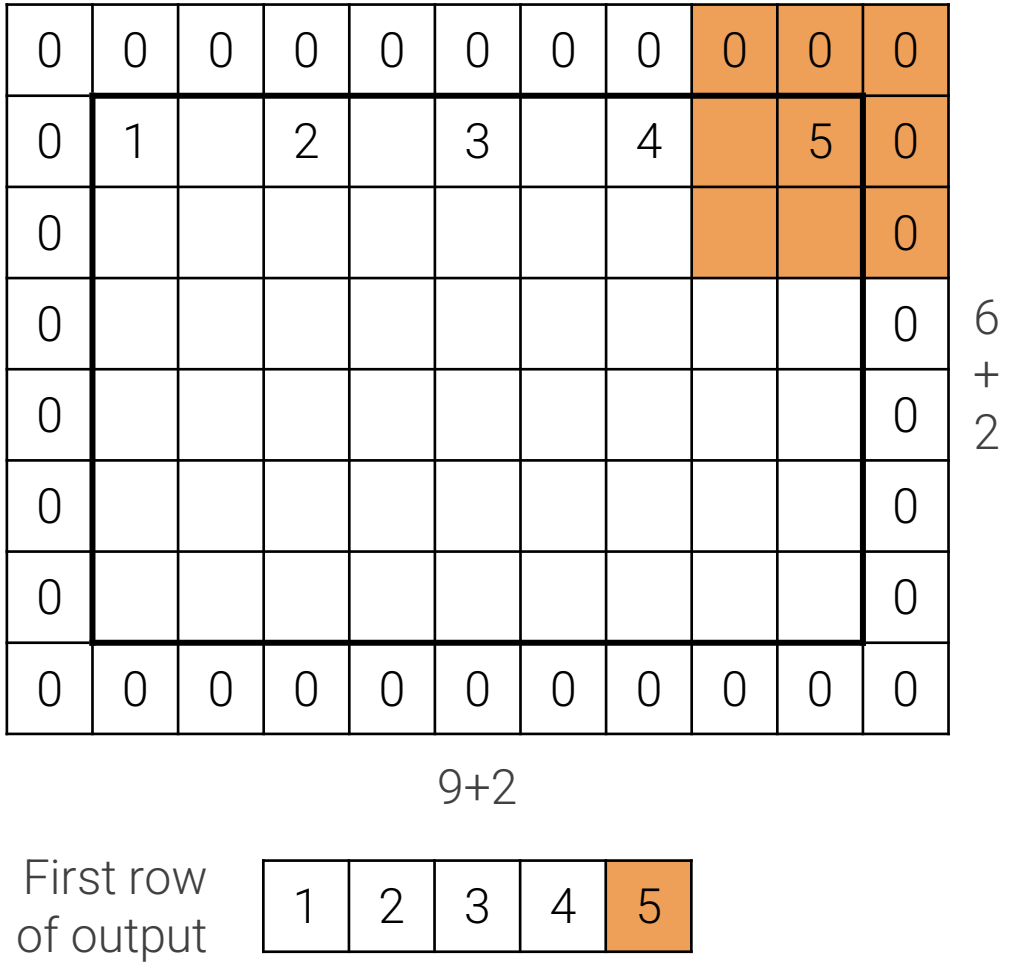
Strided convolutions



$H_{in} \times W_{in} = 6 \times 9$

$H_k \times W_k = 3 \times 3,\ P = 1,\ S = 2$

Strided convolutions



$$H_{in} \times W_{in} = 6 \times 9$$

$$H_k \times W_k = 3 \times 3, \; P = 1, \; S = 2$$

$$H_{out} \times W_{out} = 3 \times 5$$

In general

$$H_{out} = \left\lfloor \frac{(H_{in} - H_K + 2P)}{S} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{(W_{in} - W_K + 2P)}{S} \right\rfloor + 1$$

Animations describing the effects of all these parameters can be found at https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning, arXiv, 2016.

Receptive field with stride

If we apply a $H_K \times W_K$ kernel size at each layer, but with stride S_l at layer l, the receptive field of an element in the L-th activation has size

$$r_L = \left[1 + \sum_{l=1}^{L} \left((H_K - 1) \prod_{i=1}^{l-1} S_i\right)\right] \times \left[1 + \sum_{l=1}^{L} \left((W_K - 1) \prod_{i=1}^{l-1} S_i\right)\right]$$

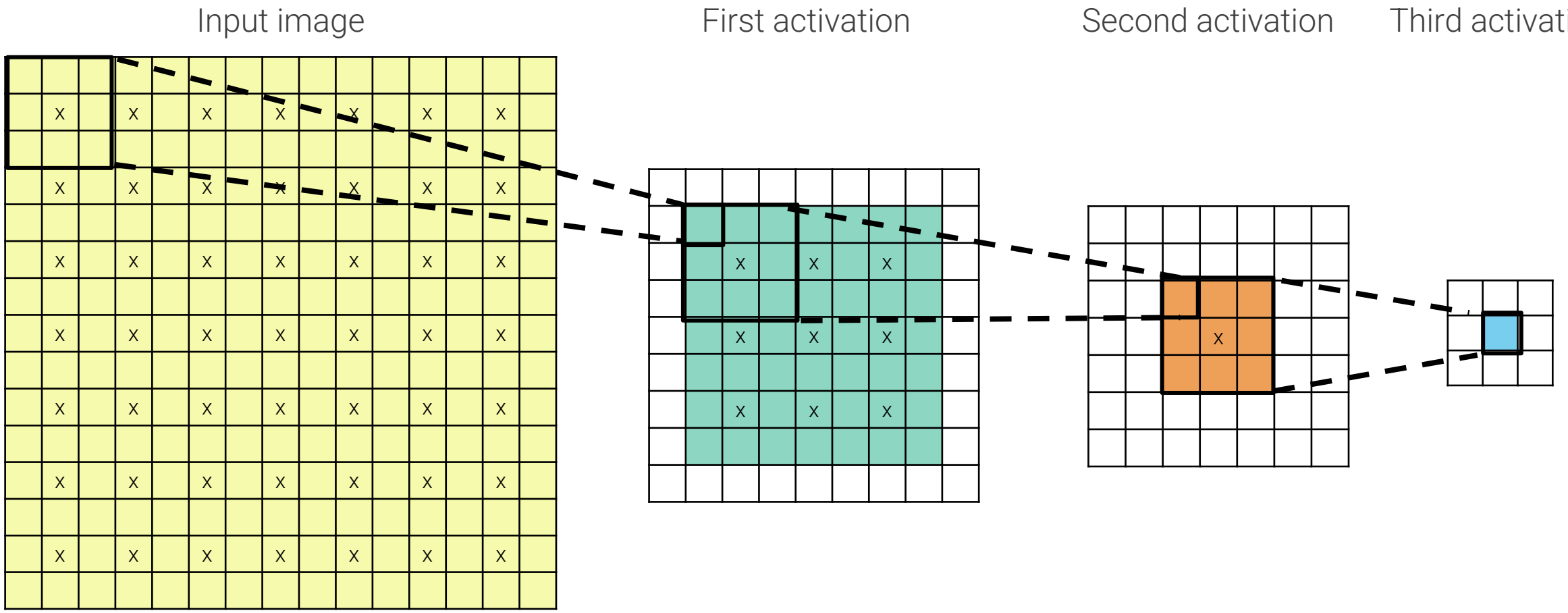
If for all layers we have $S_l = S$, then $\prod_{i=1}^{l-1} S_i = S^{l-1}$ and

$$r_L = \left[1 + \sum_{l=1}^{L} (H_K - 1)\left(S^{l-1}\right)\right] \times \left[1 + \sum_{l=1}^{L} \left((W_K - 1) S^{l-1}\right)\right]$$

i.e., the size of the receptive field grows exponentially with respect to the number of layers with stride > 1.

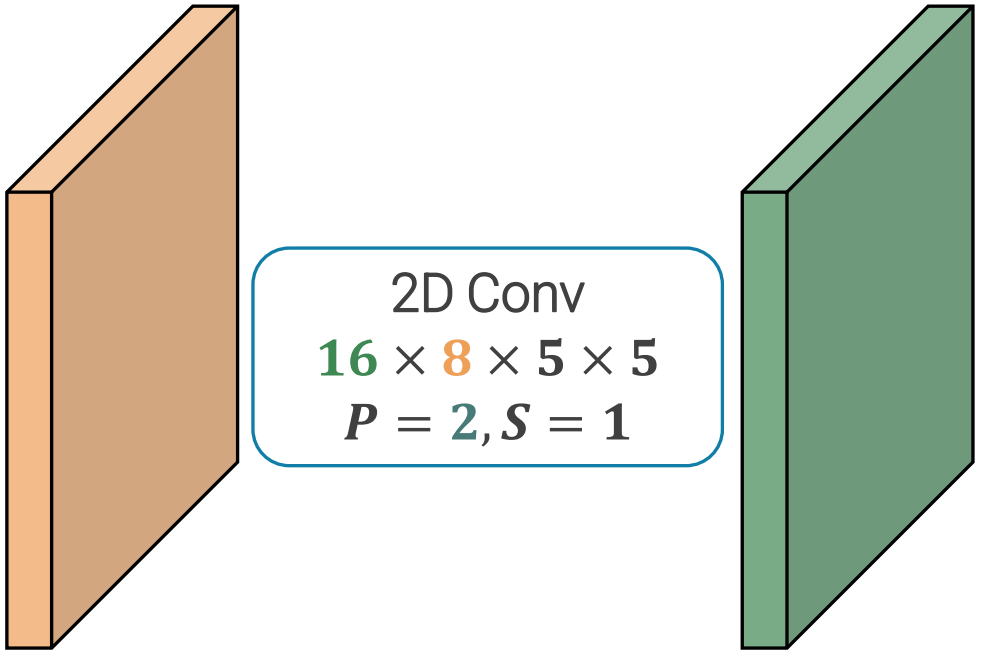
https://distill.pub/2019/computing-receptive-fields

Receptive field with stride S=2



https://distill.pub/2019/computing-receptive-fields

Convolution parameters: example



Input activation
$8 \times 32 \times 32$

Output activation
$16 \times 32 \times 32$

The number of learnable parameters for the convolution layer on the left is (+1 is for biases)

$$16 \times (8 \times 5 \times 5 + 1) = 16 \times 201 = 3,216$$

In general, a generic convolution layer whose shape is $C_{out} \times C_{in} \times H_K \times W_K$ has

$$C_{out}\ (C_{in}\ H_K\ W_K + 1)$$

learnable parameters

The size of the output for this example is

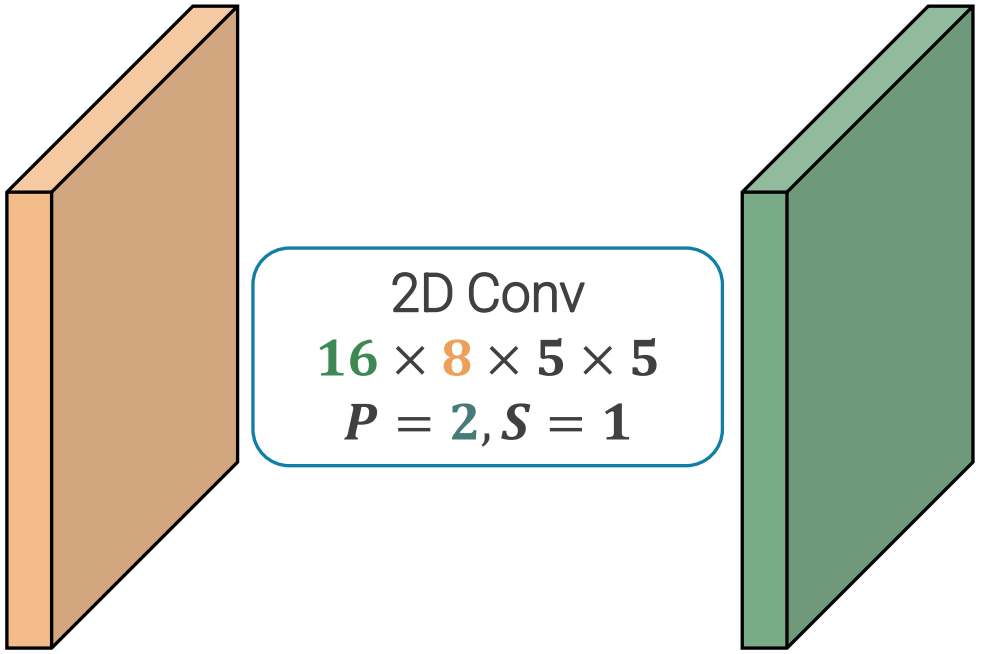
$$H_{out} = W_{out} = 32 - 5 + 2 * 2 + 1 = 32$$

Hence, there are

$$16 \times 32 \times 32 = 16,384$$

values in the output activation, i.e. about 64 KB. In general, there are $C_{out}\ H_{out}\ W_{out}$ values in the output activation.

Convolution flops: example



Input activation
$8 \times 32 \times 32$

Output activation
$16 \times 32 \times 32$

To compute each of the output values we take the dot product between $200 = 8 \times 5 \times 5$ weights and the corresponding input values, which requires **200 multiplications** and **200 additions** for inputs of size n, i.e., $2 * 200$ floating-point operations (**flops**). If the hardware supports Multiply-accumulate operations (**MACs**) that can be performed in one clock cycle, it is common to express computational complexity in terms of them, by dropping the factor 2.

Hence, the total number of flops for this convolution is

$$16,384 \times (8 \times 5 \times 5) \times 2 \cong 6.5\text{M},$$

while the total number of MACs is $\cong 3.25\text{M}$.

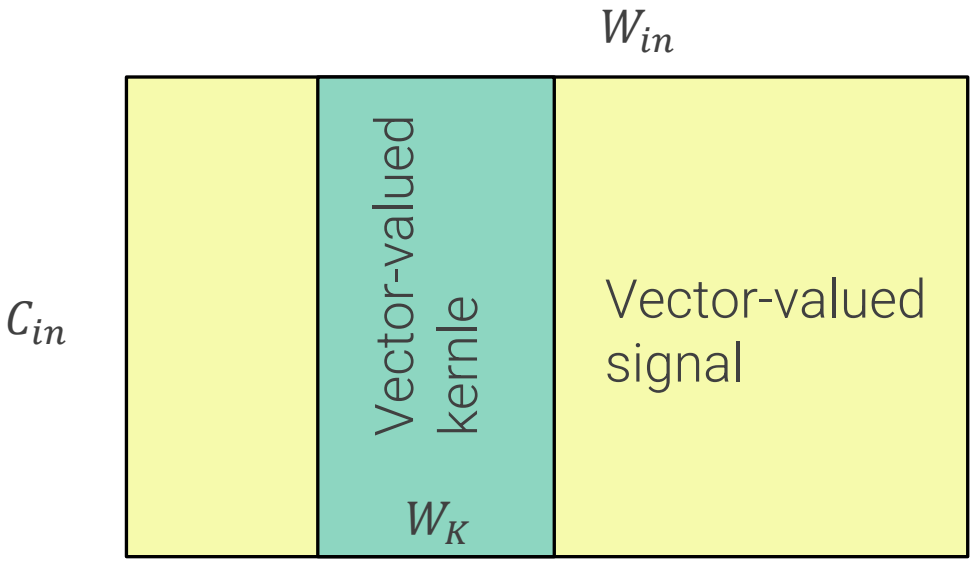
In general, the number of flops is

$$2(C_{out}\, H_{out}\, W_{out})(C_{in} H_K W_K)$$

Other common convolutions

1D convolutions

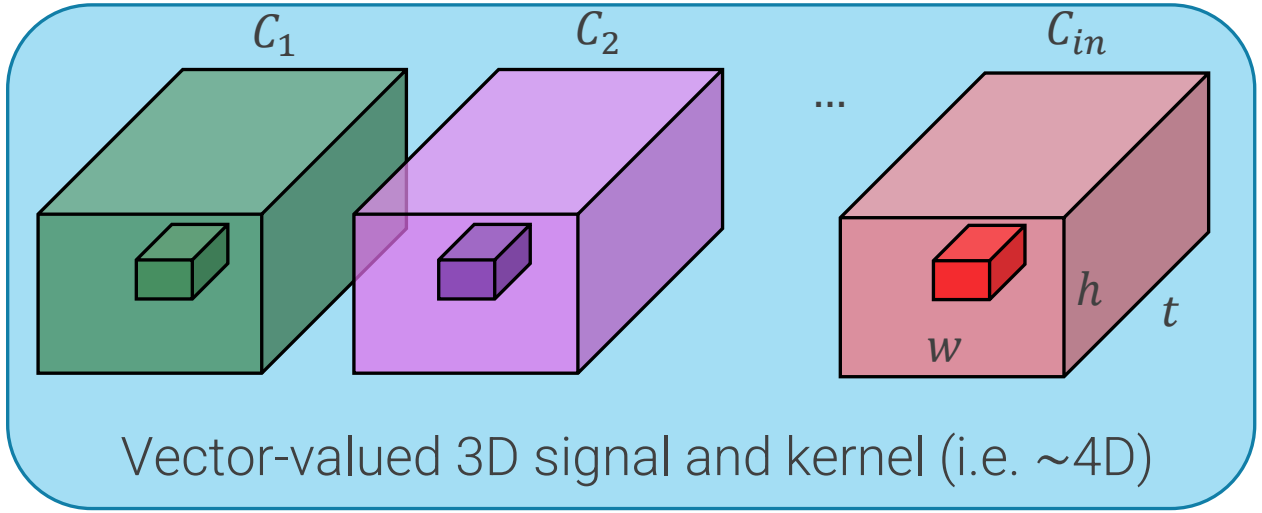
$$[K * S]_k(i) = \sum_{n=1}^{C_{in}} \sum_l K_n^{(k)}(l) S_n(i-l) + b^{(k)}$$



3D convolutions

$$[K * V]_k(h,j,i) =$$

$$\sum_{n=1}^{C_{in}} \sum_p \sum_m \sum_l K_n^{(k)}(p,m,l) V_n(h-p, j-m, i-l) + b^{(k)}$$

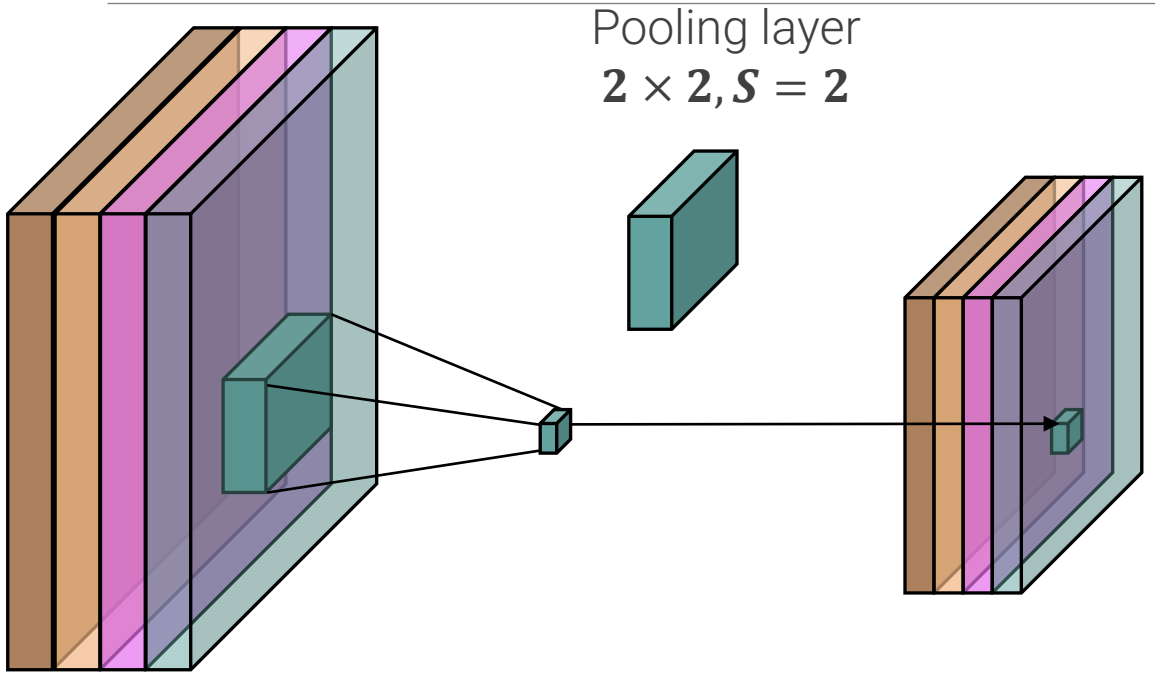


Common layers in CNNs

Neural networks based on convolutional layers are called Convolutional Neural Networks (short form is either CNNs or convnets). Beside convolutional layers, they are a composition of

- non-linear activation functions
- fully connected (a.k.a. linear) layers
- pooling layers and
- (batch-)normalization layers

Pooling layers



Hyper-parameters

Kernel width and height $W_K \times H_K$

Kernel function (pre-specified): $\max, \text{avg}, \ldots$

Stride S (usually >1)

Common choice:

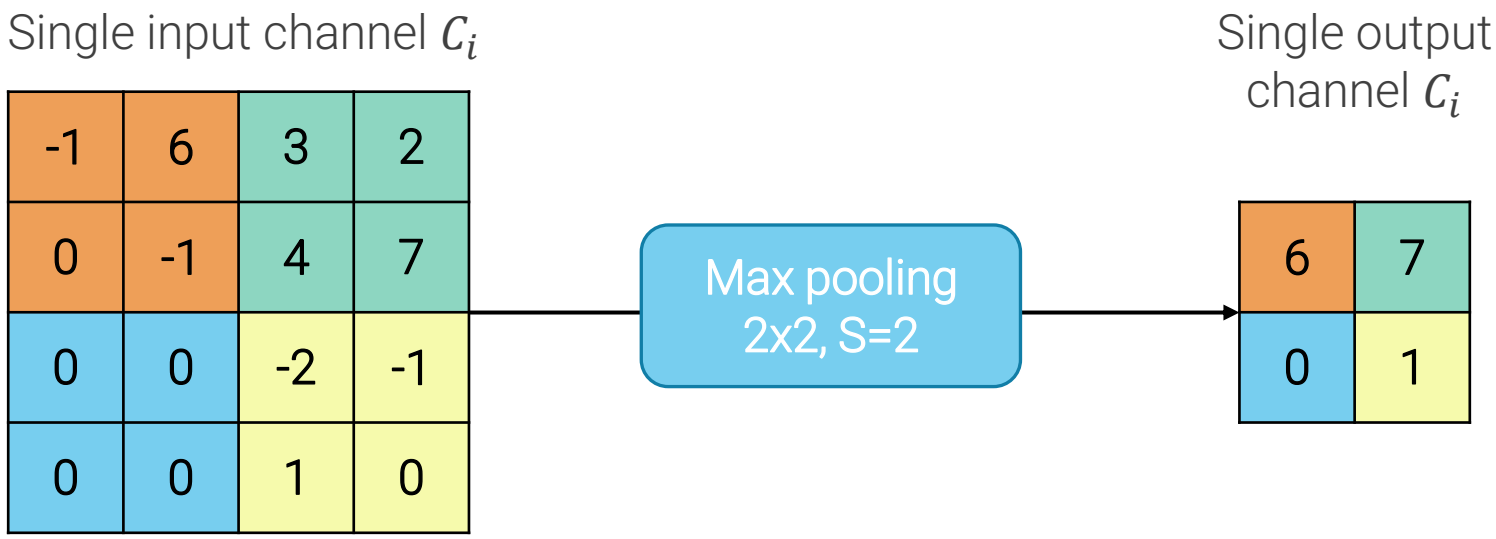
2×2, $S = 2$, kernel func= $\max$

$$[I_{pool}]_k(j,i) = \max_{l \in [0, W_k - 1]} I_k(S\,j + l, S\,i + l)$$
$$k = 1, \ldots, C_{out}$$

It aggregates several values into one output value with a pre-specified (i.e. not learned) kernel.

Key difference with convolution: each input channel is aggregated independently, i.e. the kernel works only along the spatial dimensions, and $C_{out} = C_{in}$

Max pooling



"The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." G. Hinton

Downsampling comes from stride, not from pooling: we can achieve it with convolutions as well.

Yet, compared to strided convolutions, max pooling

- has no learnable parameters (pro and con)
- provides invariance to small spatial shifts.

https://www.reddit.com/r/MachineLearning/comments/2lmo0l/ama_geoffrey_hinton/

Batch Norm

$$f(\boldsymbol{x};\theta) = W_2\boldsymbol{h} + \boldsymbol{b_2}$$
$$= W_2\phi(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$

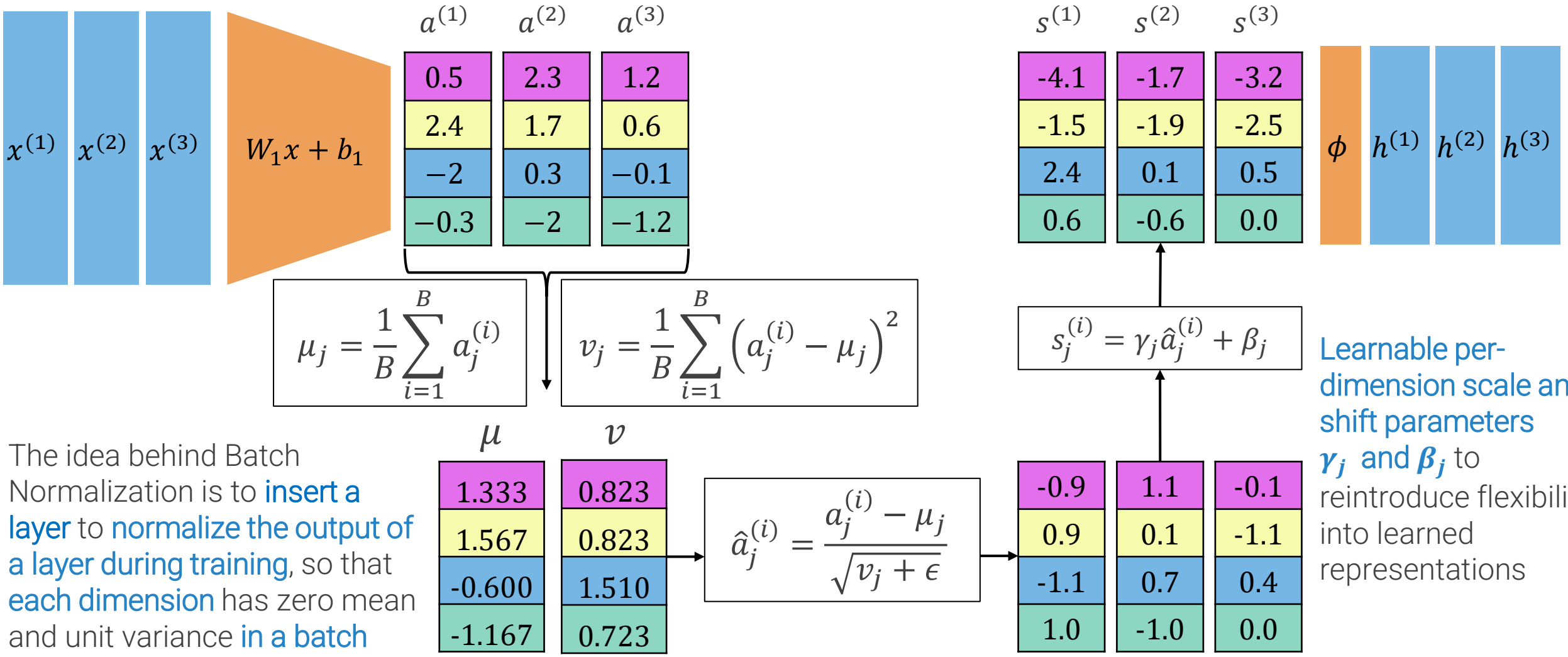
Problem: we know that it is in practice beneficial to normalize input data to shallow classifiers like SVMs, i.e. make them live in the range [-1,1] or make the data follow a Gaussian distribution by standardizing them. It makes training more stable and produces more performant classifiers.

This is normally done by computing statistics, like mean and variance or max and min, across the entire training set and by using them to normalize the input data.

However, when the input to the last linear classifier is the representation h computed by other layers before it, it is not clear how to realize normalization. Even if we compute a mean and variance at a certain step, the distribution of h will change after any mini-batch when W_1 and b_1 are updated by SGD. Moreover, in this hypothetical approach, it is not clear how to have normalization and optimization not "undo" each other, i.e. how to make SGD aware of the standardization constants.

Sergey Ioffe, Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch norm layer – training time



The idea behind Batch Normalization is to insert a layer to normalize the output of a layer during training, so that each dimension has zero mean and unit variance in a batch

Learnable per-dimension scale and shift parameters γ_j and β_j to reintroduce flexibility into learned representations

Batch normalization at training time

Input: a mini-batch of activations $\{a^{(i)} | i = 1, \dots, B\}$ where each sample has dimension D

$2D$ learnable parameters: γ_j and β_j, $j = 1, \dots, D$

Four steps: all of them are differentiable and can be back-propagated through: it guaranties that optimization does not "undo" normalization

Learning $\gamma_j = \sqrt{v_j}$ and $\beta_j = \mu_j$ may recover the identity function, if it were the optimal thing to do.

At training time, we also keep a running average for each mean and variance ($m = 0.1$ is usually called BN momentum)

$$\mu_j^{(t)} = (1 - m)\, \mu_j^{(t-1)} + m\, \mu_j,$$
$$v_j^{(t)} = (1 - m)\, v_j^{(t-1)} + m\, v_j$$

Shape: $1 \times D$

$$\mu_j = \frac{1}{B} \sum_{i=1}^{B} a_j^{(i)}$$

Shape: $1 \times D$

$$v_j = \frac{1}{B} \sum_{i=1}^{B} \left(a_j^{(i)} - \mu_j\right)^2$$

Shape: $1 \times D$

$$\hat{a}_j^{(i)} = \frac{a_j^{(i)} - \mu_j}{\sqrt{v_j + \epsilon}}$$

Shape: $B \times D$

$$s_j^{(i)} = \gamma_j\, \hat{a}_j^{(i)} + \beta_j$$

Shape: $B \times D$

Batch normalization at test time

Input: a mini-batch of activations $\{a^{(i)} | i = 1, \dots, B\}$ where each sample has dimension D

$2D$ learned parameters: γ_j and β_j, $j = 1, \dots, D$

At test time, we do not want to stochastically depend on the other items in the mini-batch: we want the output to depend only on the input, deterministically. Mean and variance become two constant values: the final values of the running averages computed at training time.

Hence, BN becomes a deterministic linear transformation, which can also be fused with the previous fully connected or convolutional operation

$\mu_j^{(T)}$ = final value of the running average computed at training time . It's constant at test time.

Shape: $1 \times D$

$v_j^{(T)}$ = final value of the running average computed at training time . It's constant at test time.

Shape: $1 \times D$

$$s_j^{(i)} = \frac{\gamma_j}{\sqrt{v_j^{(T)} + \epsilon}} a_j^{(i)} + \left(\beta_j - \frac{\gamma_j \mu_j^{(T)}}{\sqrt{v_j^{(T)} + \epsilon}} \right)$$

Shape: $B \times D$

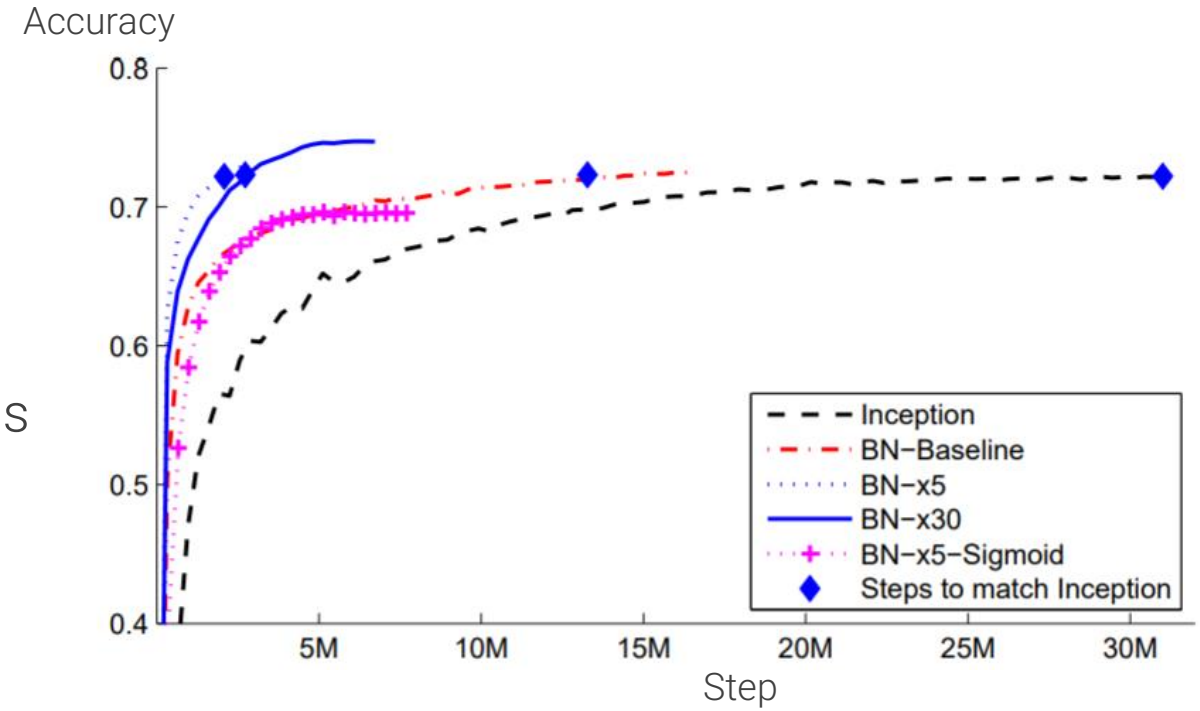
Batch norm: pros and cons

Pros

- allows the use of higher learning rates
- careful initialization is less important
- Training is not deterministic, acts as regularization
- No overhead at test-time: can be fused with previous layer

Cons

- Not clear why it is so beneficial
- Need to distinguish between training and testing time makes implementations more complex, source of many bugs
- Does not scale down to “micro-batches”



Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Sergey Ioffe, Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch norm for convolutional layers

Batch norm for fully connected layers

Normalize along mini-batch dimension

$$a: B \times D$$

$$\mu, v: 1 \times D$$

$$\gamma, \beta: 1 \times D$$

$$s = \gamma \frac{a-\mu}{\sqrt{v+\epsilon}} + \beta$$

Batch norm for convolutional layers

Normalize along mini-batch and spatial dimensions

$$a: B \times C_{out} \times H \times W$$

$$\mu, v: 1 \times C_{out} \times 1 \times 1$$

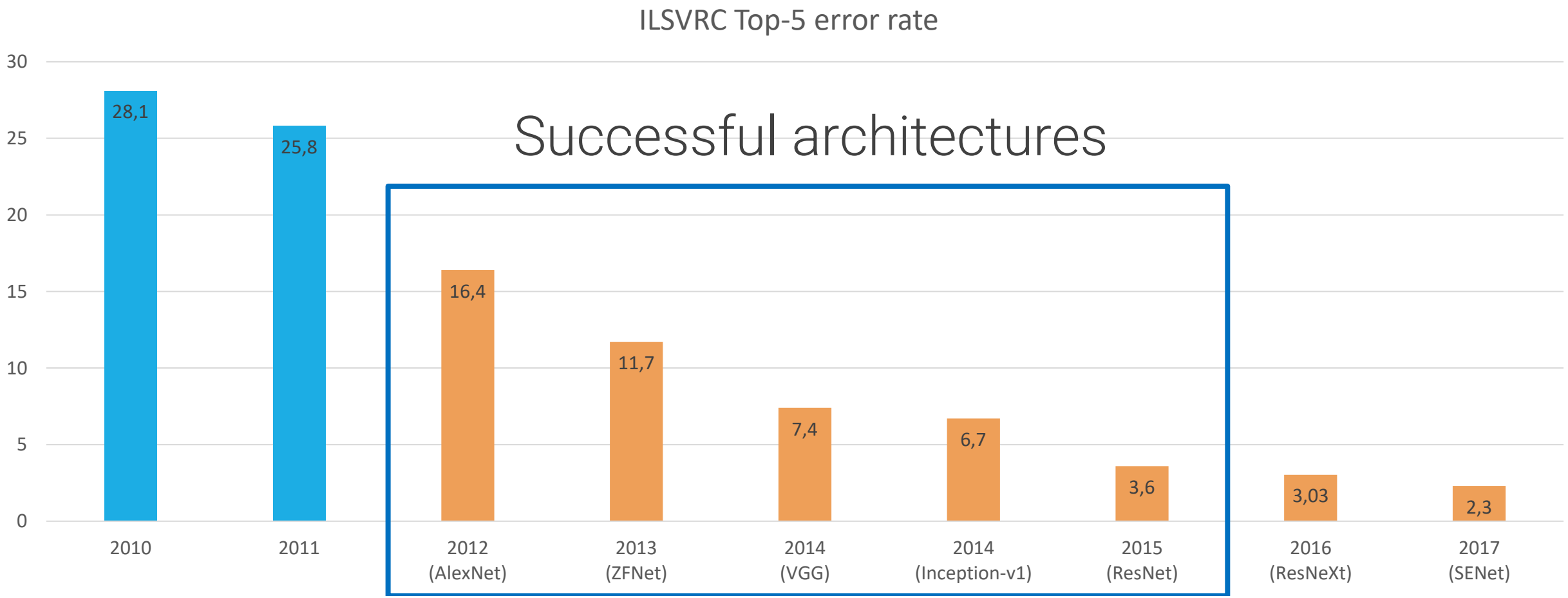
$$\gamma, \beta: 1 \times C_{out} \times 1 \times 1$$

$$s = \gamma \frac{a-\mu}{\sqrt{v+\epsilon}} + \beta$$

Also known as Spatial Batch Norm, BatchNorm2D

Why? The idea is to respect how convolutional layers work: elements of the same feature map are processed by the same kernel are normalized in the same way

What's next?


ILSVRC Top-5 error rate
Successful architectures
30
25
20
15
10
5
0
28,1
25,8
16,4
11,7
7,4
6,7
3,6
3,03
2,3
2010
2011
2012 (AlexNet)
2013 (ZFNet)
2014 (VGG)
2014 (Inception-v1)
2015 (ResNet)
2016 (ResNeXt)
2017 (SENet)