

Lecture 4

Image representations

IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

Limits of “shallow” classifiers



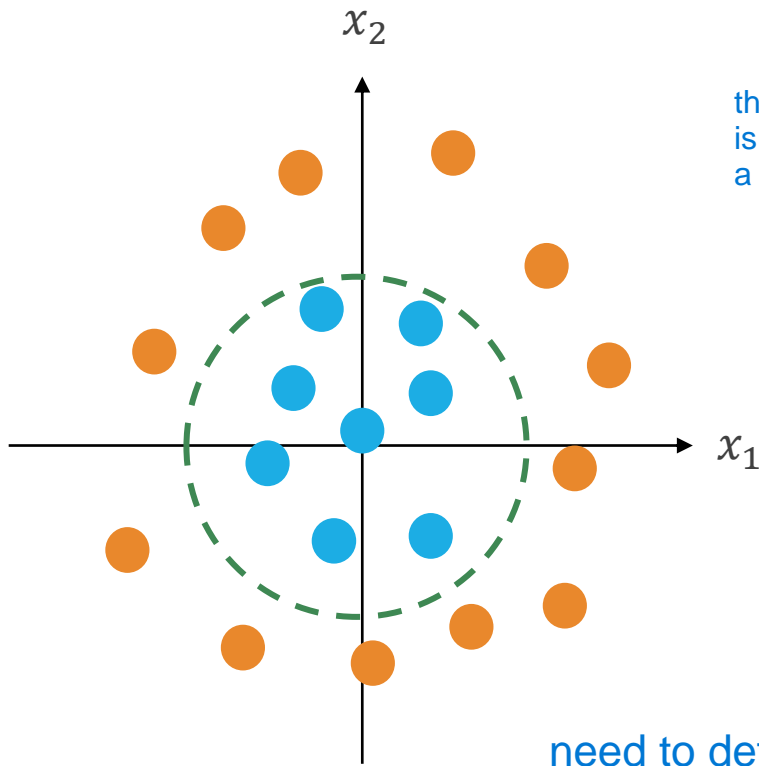
k -NN and linear classifiers are limited by the low effectiveness of input pixels as data features.

pixel space is not effective to represent features

Representation is important

new task: learn new image representation
- in this way, we can build a better model

we don't need fancier classifiers, we need
better representations



this is the reason why representation
is important: linear classifier only provide
a linear boundary

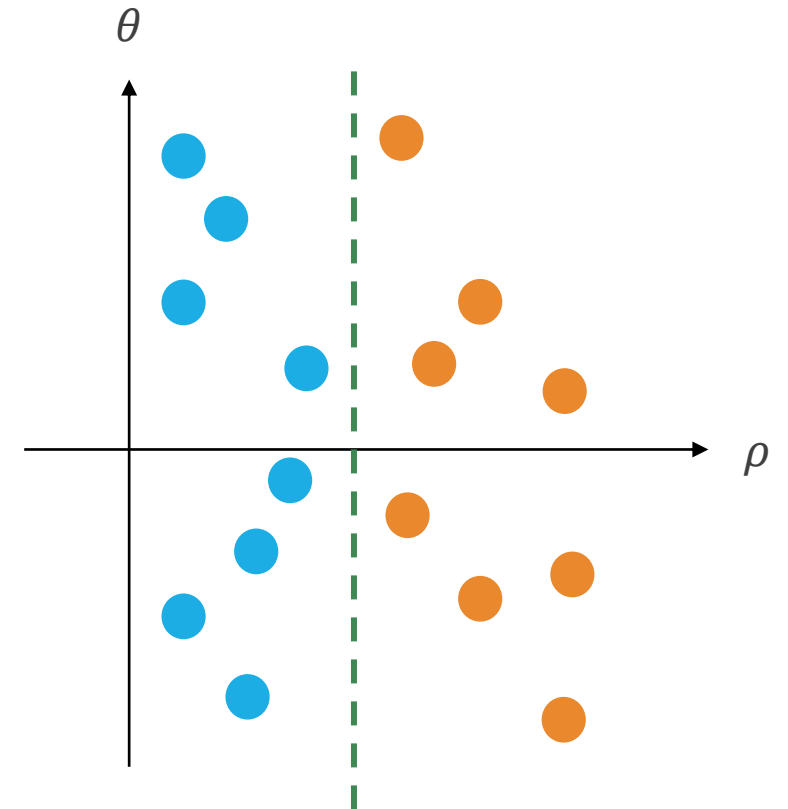
Switch to polar
coordinates

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

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

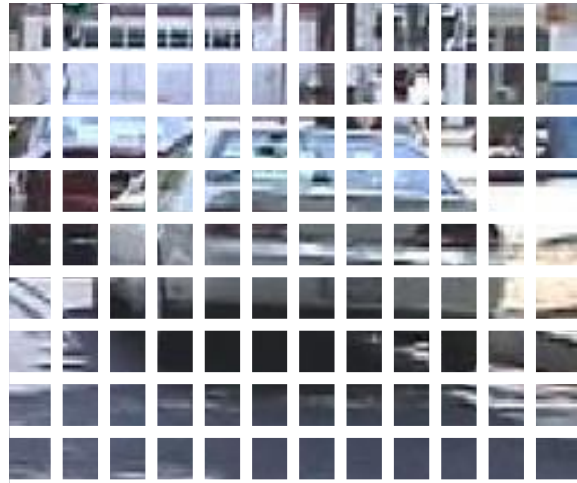
need to define a new feature representation
space: pixel (input) space is not enough

Non-linear decision
boundary in input space

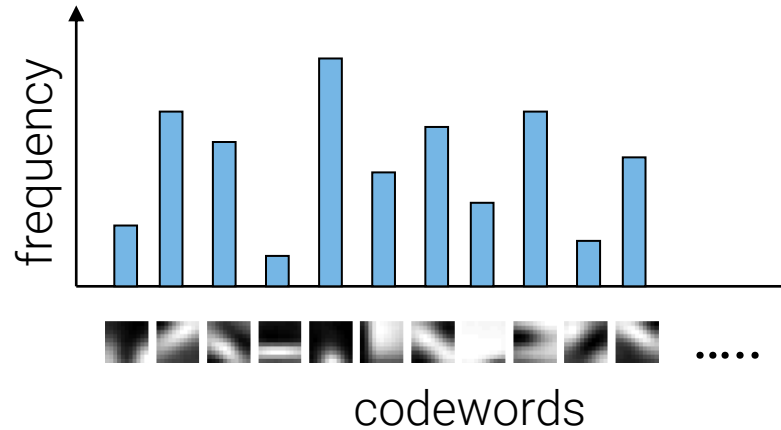


Linear decision boundary
in feature space

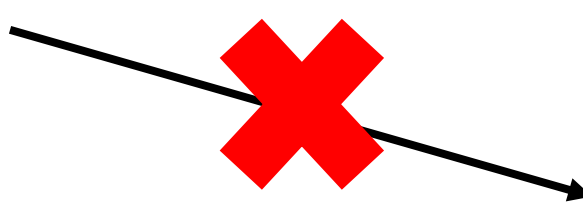
Bag of Visual Words (BoVW)



each of these patches
is a visual "word"



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**.


$$f(x; \theta) = Wx + b = \textit{scores}$$

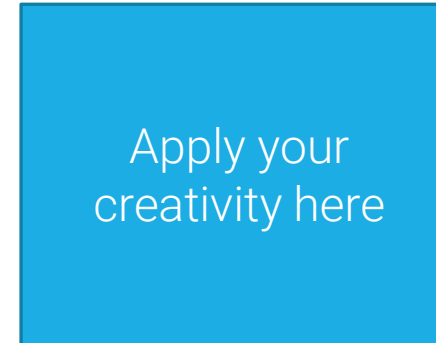
Csurka et al., Visual Categorization with Bags of Keypoints, ECCV 04
Fei-Fei et Perona, A Bayesian Hierarchical Model for Learning Natural Scene Categories, CVPR 05
Sivic et al, Discovering objects and their location in images, ICCV05

BoVW was the dominant paradigm until 2012

each patch is represented" by it own descriptor, computed with SIFT descriptor



2) Assign each SIFT descriptor in an image to the nearest visual word. Create a histogram of visual word occurrences.

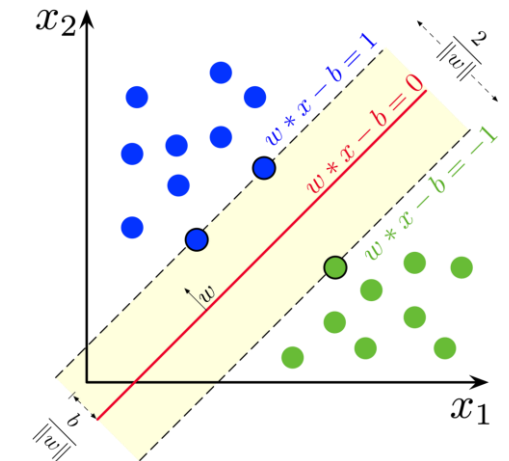


Precomputed dense SIFT descriptors at 3 scales

1) Collect all SIFT descriptors from training images. Use k-means clustering to cluster the descriptors into k visual words.

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

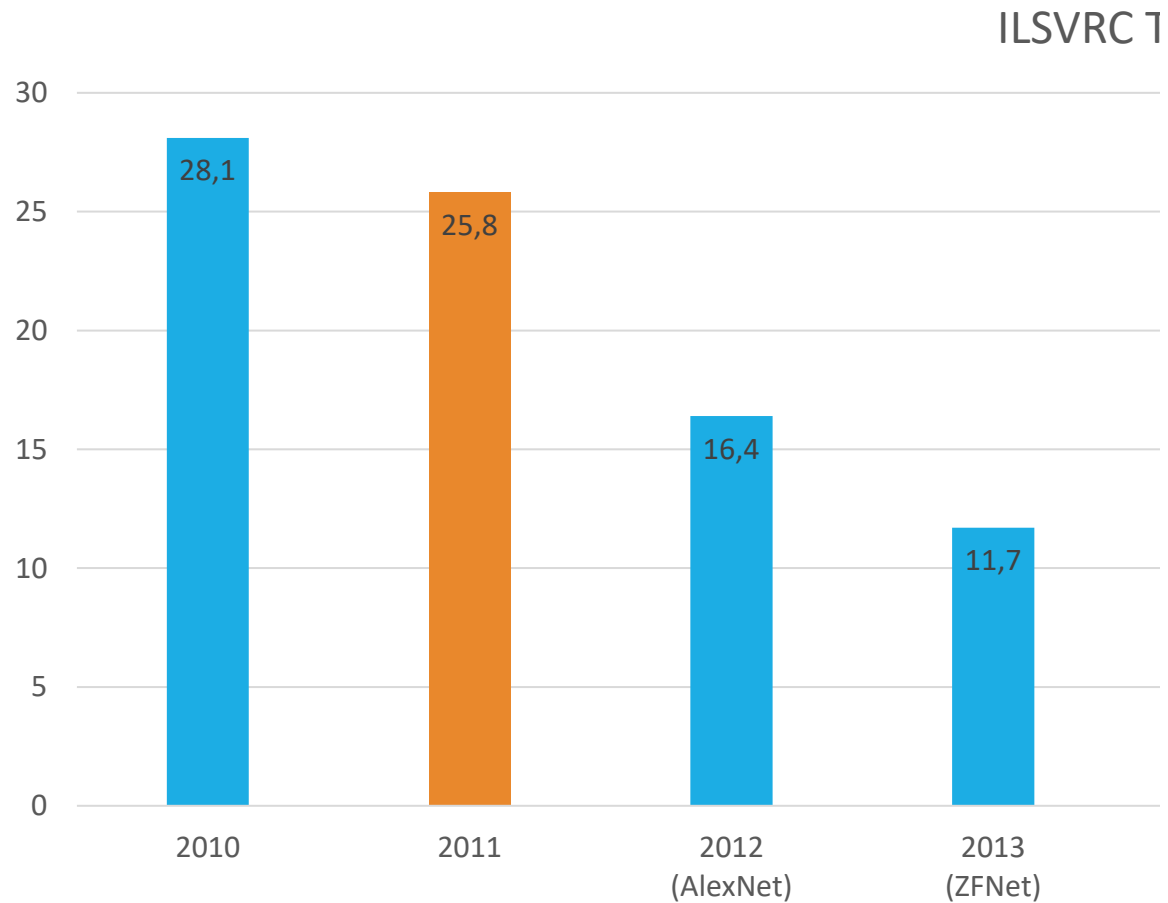
3) Use the histograms to train a classifier.



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 \approx 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 \Rightarrow 520K dim x 2
- compression: $G=8$, $b=1$ bit per dimension

One-vs-all SVM learning with SGD

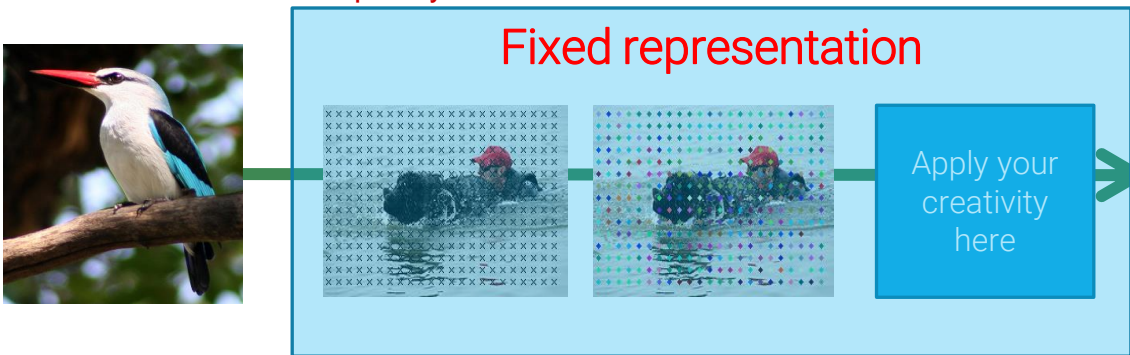
Late fusion of SIFT and color systems

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

Deep networks have sufficient expressive power to transform data into a space where a linear decision boundary is effective. The network learns the necessary transformations to make the data linearly separable.

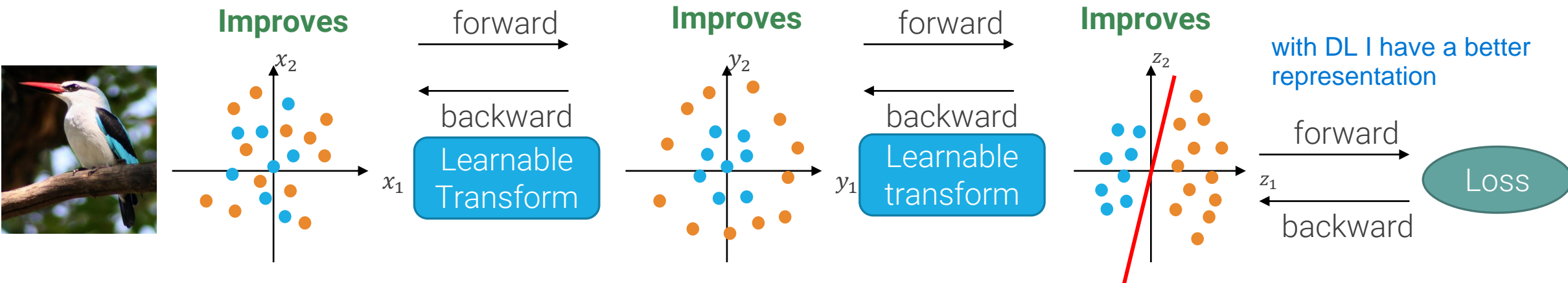
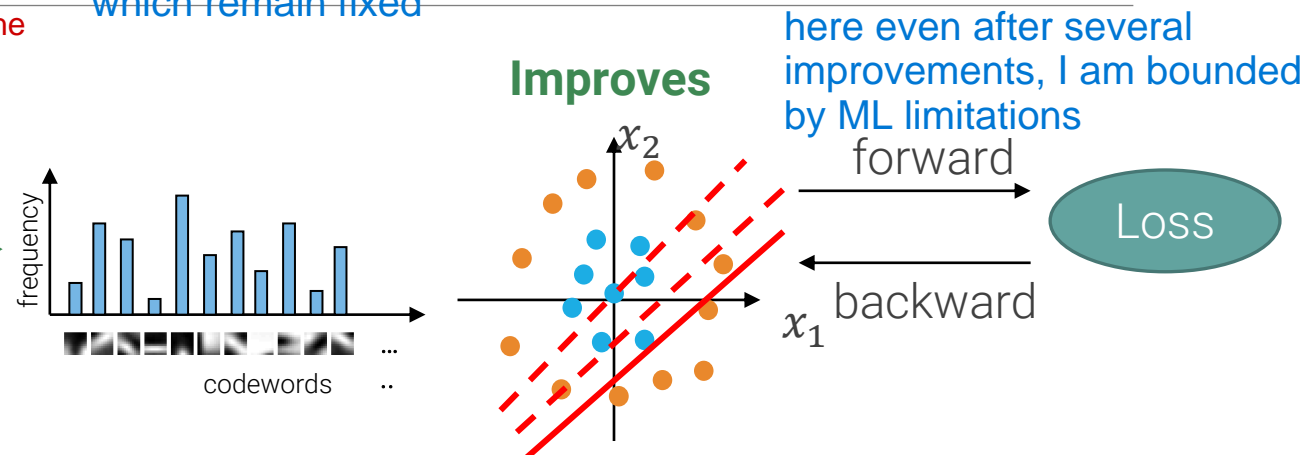
Representation learning

Using a linear classifier at the end simplifies the optimization problem because linear classifiers are easier to train and converge faster. Adding a non-linear classifier at the end would increase the complexity even further.



DL solves the problem of representation: it is able to LEARN REPRESENTATION, unlike shallow ML

with shallow ML I can only improve classifier, not the representation, which remain fixed



Deep learning \approx Representation learning

Neural networks

instead of a linear classifier working directly on x , I have a linear classifier that works on an inner representation h , which is the output of a module which has parameters: W_1 and b_1 are parameters of the learnable inner representation; W_2 and b_2 are the params of the linear classifier

Linear classifier

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

the linear representation is the same as before, but h is the inner representation

Neural Network

3072x1

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

10xC

Cx1

10x1

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

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

the set of weights is larger than before

10xC

Cx3072

3072x1

Cx1

10x1

C hyperparam

New hyper-parameters

- Dimension of inner representation C
- Activation function ϕ

Why do we insert activation functions?

$$\begin{aligned} f(\mathbf{x}; \theta) &= \mathbf{W}_2 \mathbf{r} + \mathbf{b}_2 \\ &= \mathbf{W}_2 (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 \\ &= (\mathbf{W}_2 \mathbf{W}_1) \mathbf{x} + (\mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2) \\ &= \mathbf{W}_{21} \mathbf{x} + \mathbf{b}_{21} \end{aligned}$$

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

we need activations function because we need NON-LINEARITY in our model, otherwise we will have a linear classifier another time

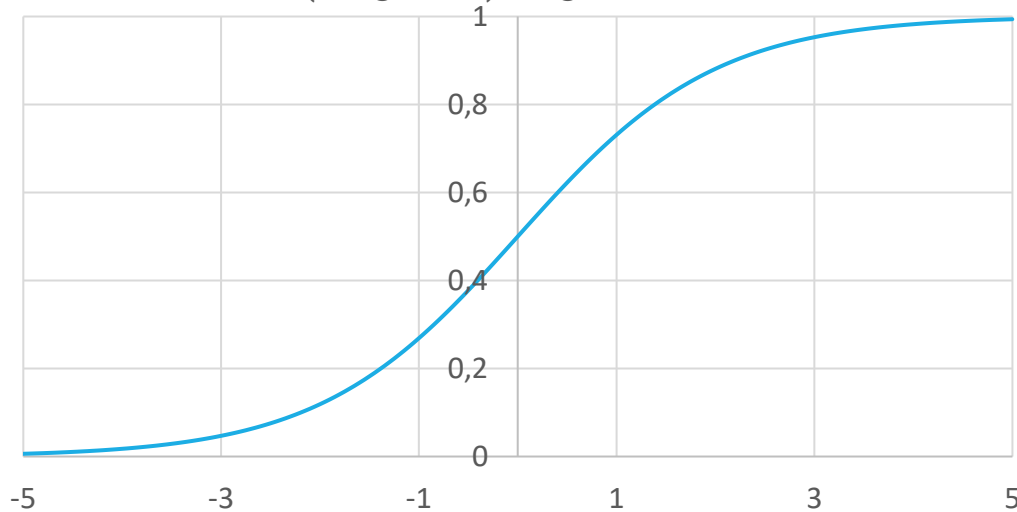
Activation functions

ReLU is the best choice: is linear in the positive side
is 0 in the negative one

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

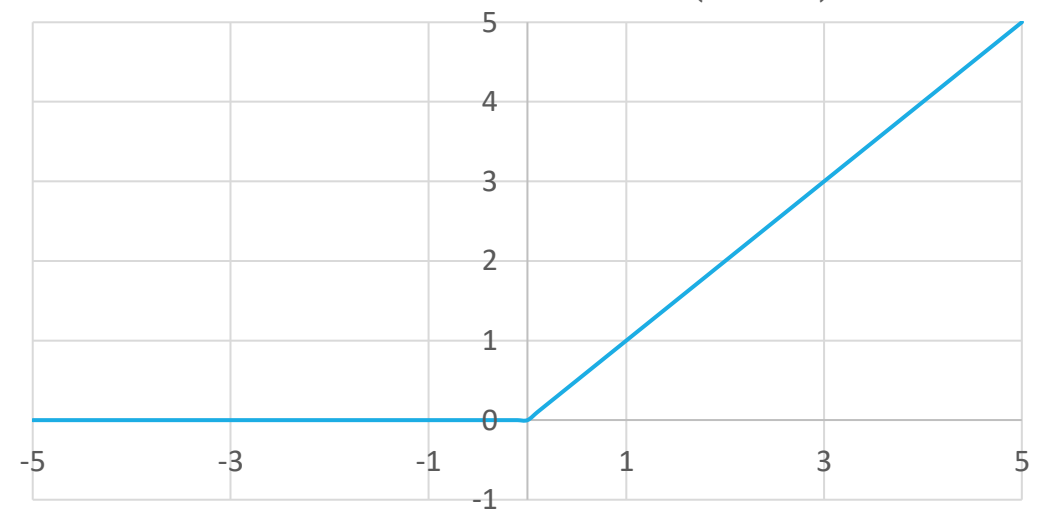
$$f(\mathbf{x}; \theta) = \mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

(Logistic) Sigmoid



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

Rectified Linear Unit (ReLU)



$$\phi(a) = \max(0, a) = \text{ReLU}(a)$$

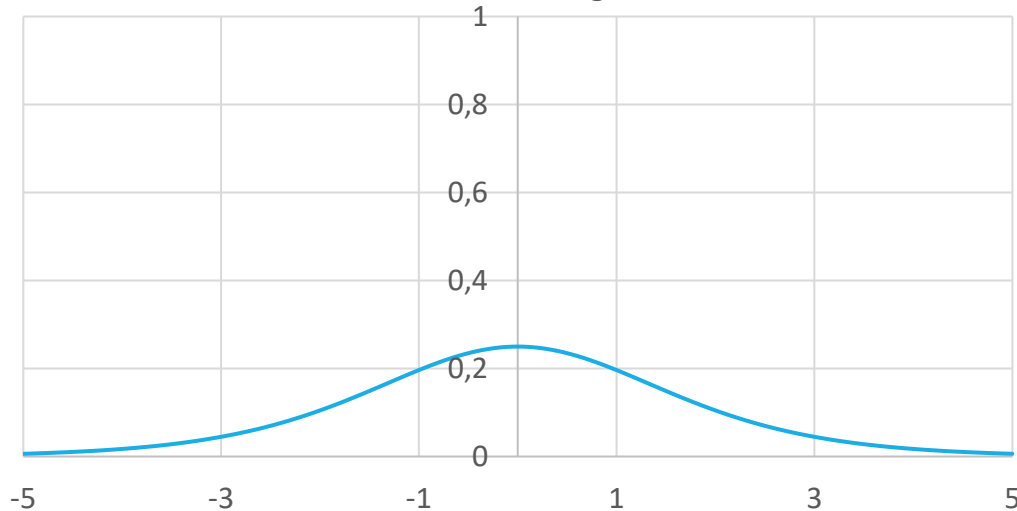
Nair, V. and Hinton, G. E. "Rectified linear units improve Restricted Boltzmann Machines". ICML 2010.
Xavier Glorot; Antoine Bordes; Yoshua Bengio "Deep sparse rectifier neural networks", AISTATS 2011

Activation functions - gradients

A "dead" neuron refers to a situation where the neuron consistently outputs zero (or remains inactive) for all inputs during training and inference. This occurs when the input to the ReLU activation function is always negative.

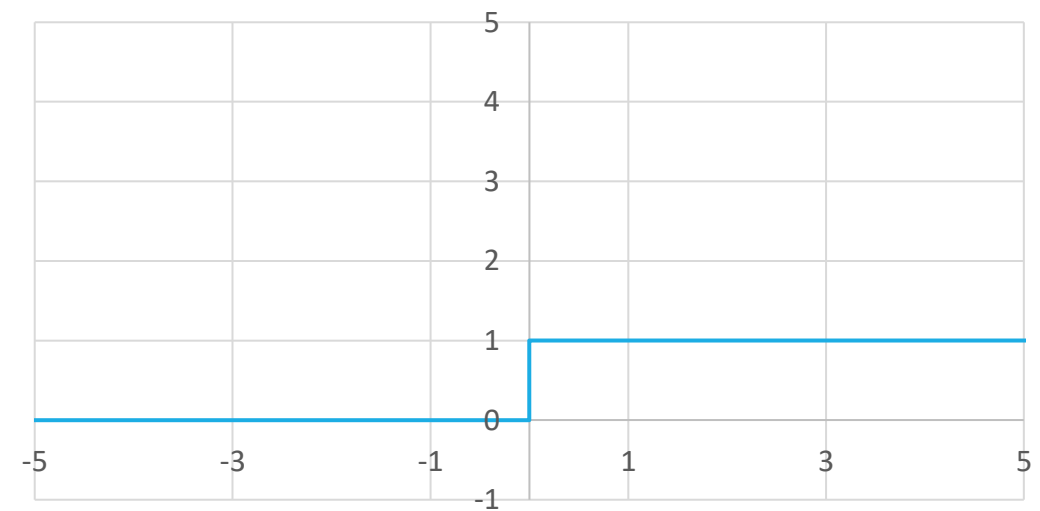
Gradient of the output of activation functions with respect to input is very different between sigmoid and ReLU. The gradient of the sigmoid function saturates when the input is large in absolute value, while ReLU has always gradient "1" for positive values: it is easier to train (deep) networks when using ReLU. Yet, ReLU can give rise to "dead" neurons, if for all inputs the output is negative. This phenomenon hampers the network's ability to learn and can lead to reduced performance.

Gradient of sigmoid



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

Gradient of ReLU



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

in general, ReLU is preferred over all the other activation functions because it makes the model to perform better at training time: avoids vanishing gradient problem, less costly, simpler derivative, etc.

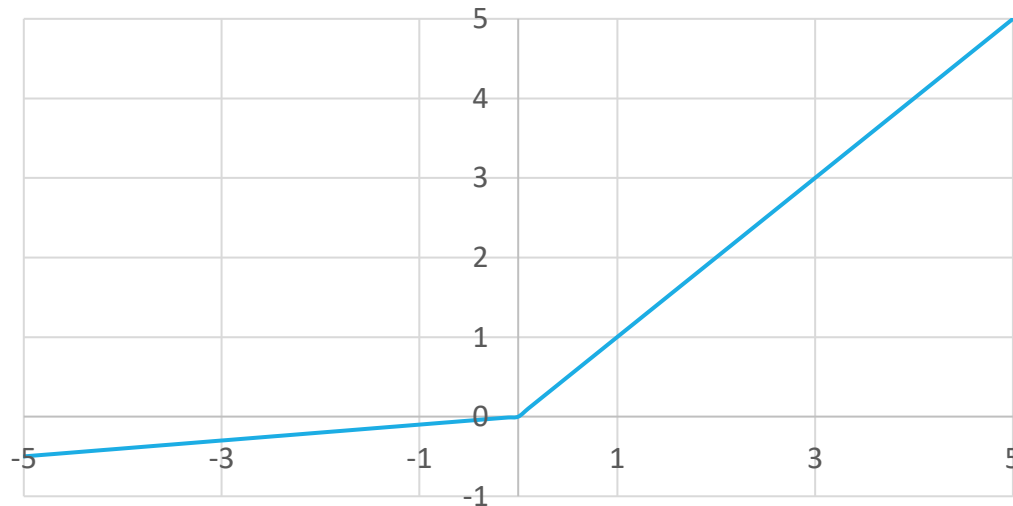
Activation functions – Leaky ReLU

in this way, all neurons, even those with negative inputs, can learn something

neurons that receives neg. inputs and it outputs 0, then we'll have 0 gradient

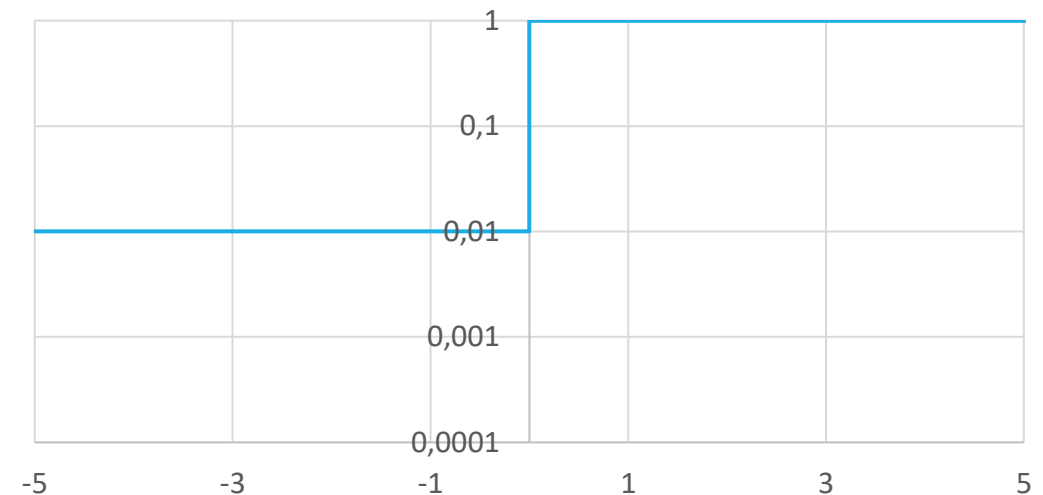
To avoid dead neurons, a small response can be produced also for negative inputs. The simplest variant is called Leaky ReLU, which has a small but non-zero constant gradient also for negative inputs.

Leaky ReLU



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

Gradient of Leaky ReLU



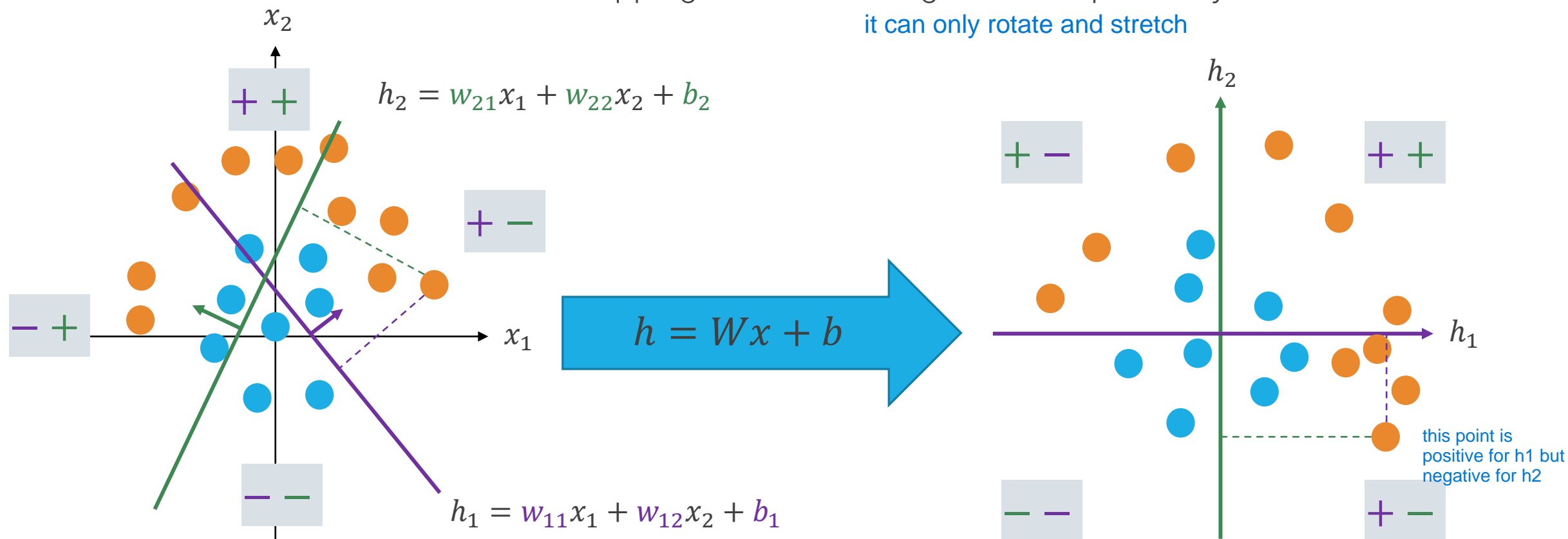
$$\frac{d \text{ Leaky ReLU}(a)}{da} = \begin{cases} 1 & \text{if } a \geq 0 \\ 0.01 & \text{ow} \end{cases}$$

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

Is ReLU enough?

h is a linear transformation of x_1 and x_2 , thus hyperplanes

A linear or affine mapping does not change linear separability
it can only rotate and stretch

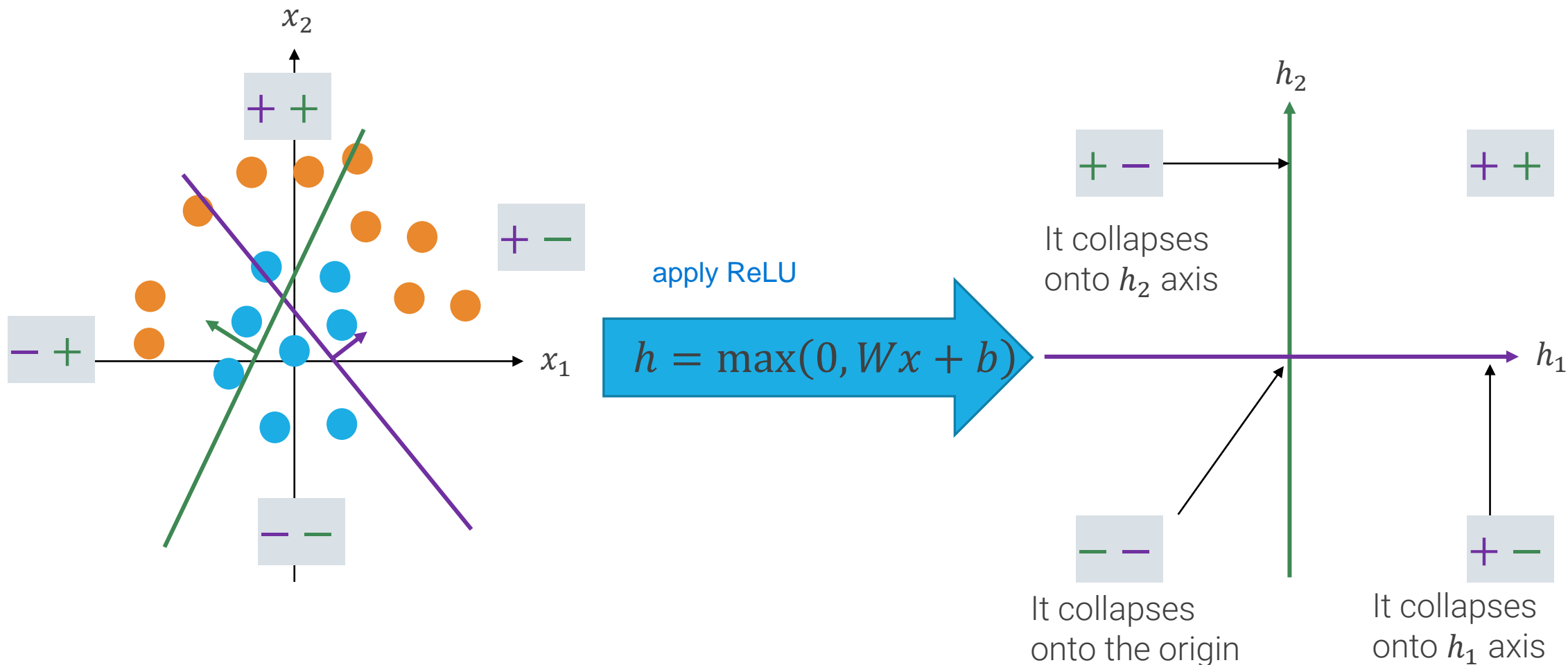


Points not linearly separable in input space

Points not linearly separable in hidden space

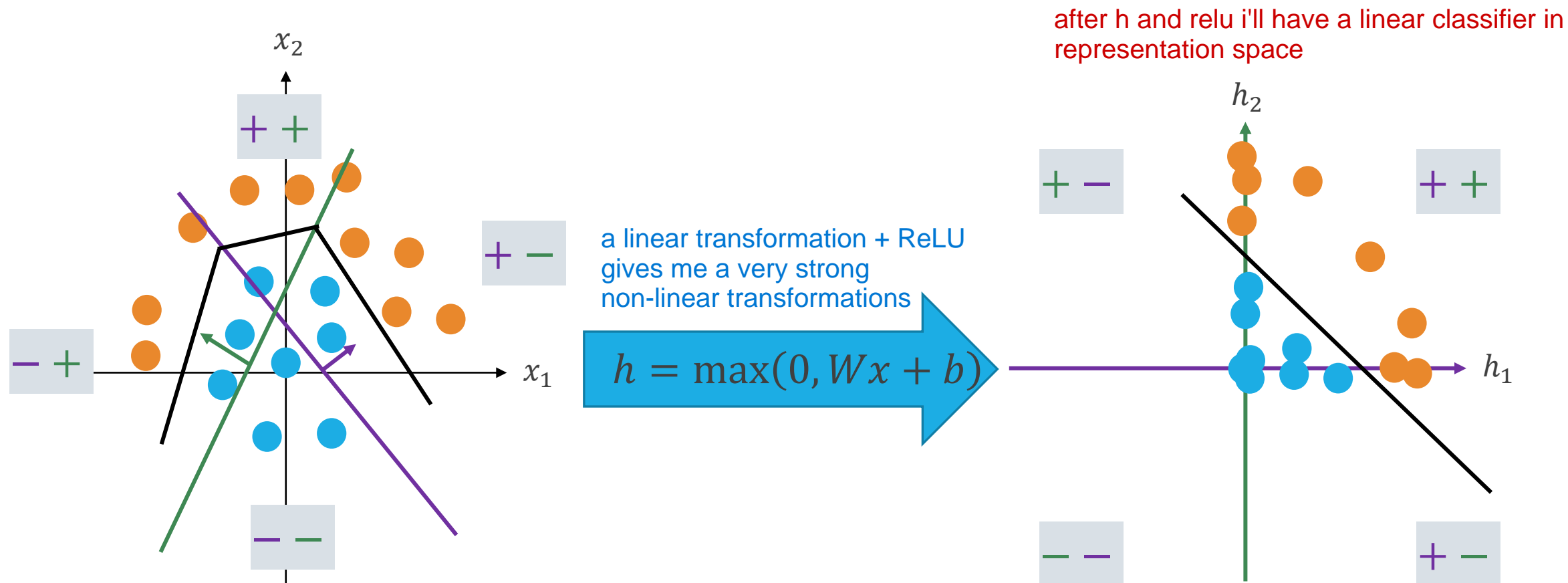
Is ReLU enough?

While the inner representations (feature vectors) learned by intermediate layers of a DNN are non-linearly wrt the input, they are structured in such a way that classes may become linearly separable in this higher-dimensional feature space.



Is ReLU enough?

this is what ReLU basically does, it is a massive folding function



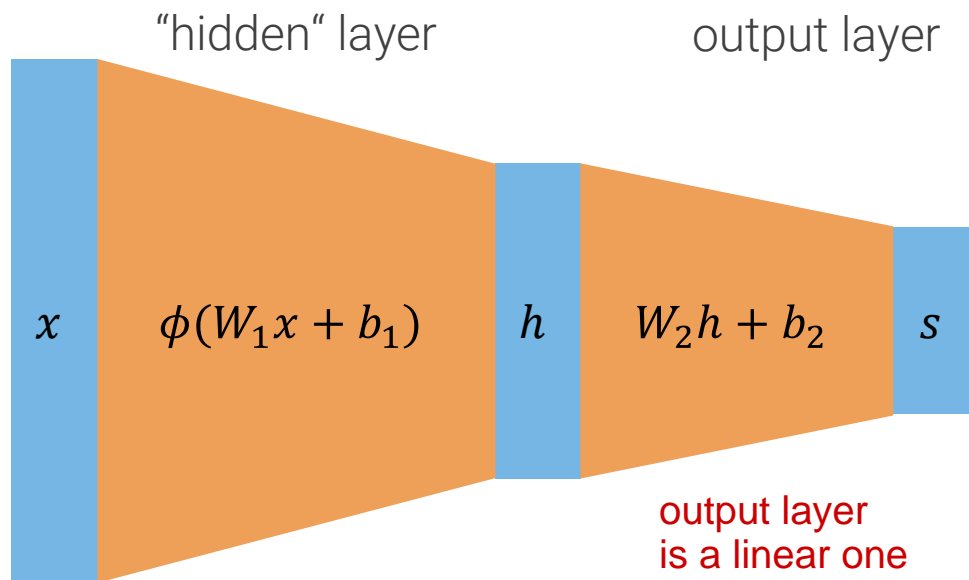
Linear classifier in representation space
creates nonlinear classifier in input space

Points linearly separable in hidden space

Graphical representation & terminology

$$\begin{aligned} f(x; \theta) &= W_2 h + b_2 \\ &= W_2 \phi(W_1 x + b_1) + b_2 = s \end{aligned}$$

is the output of a linear trans + a non-linearity (activation function) and it is called ACTIVATION



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**

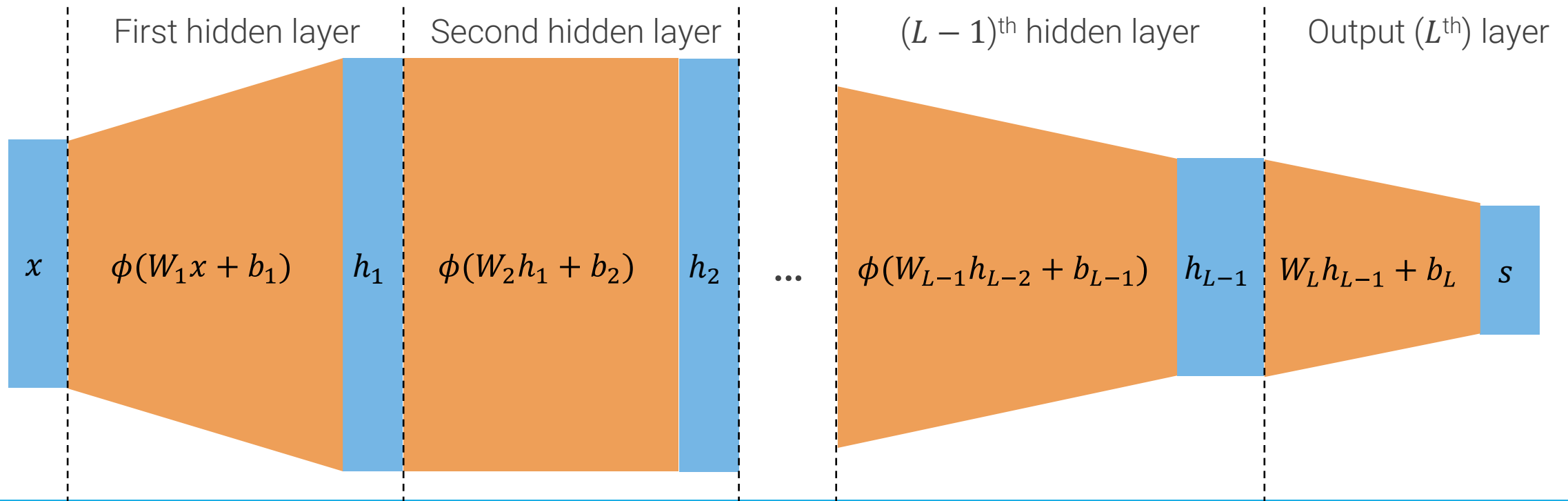
Every layer is often called a **fully connected (FC) layer**, as every element of the input influences every element of the output

A neural network with 2 or more layers is also called a **Multi-Layer Perceptron (MLP)**

“Deep” neural networks

$$\begin{aligned} f(\mathbf{x}; \theta) &= \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L \\ &= \mathbf{W}_L \phi(\mathbf{W}_{L-1} \mathbf{h}_{L-2} + \mathbf{b}_{L-1}) + \mathbf{b}_L \\ &= \mathbf{W}_L \phi(\mathbf{W}_{L-1} \phi(\dots \phi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \dots) + \mathbf{b}_{L-1}) + \mathbf{b}_L = \mathbf{s} \end{aligned}$$

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

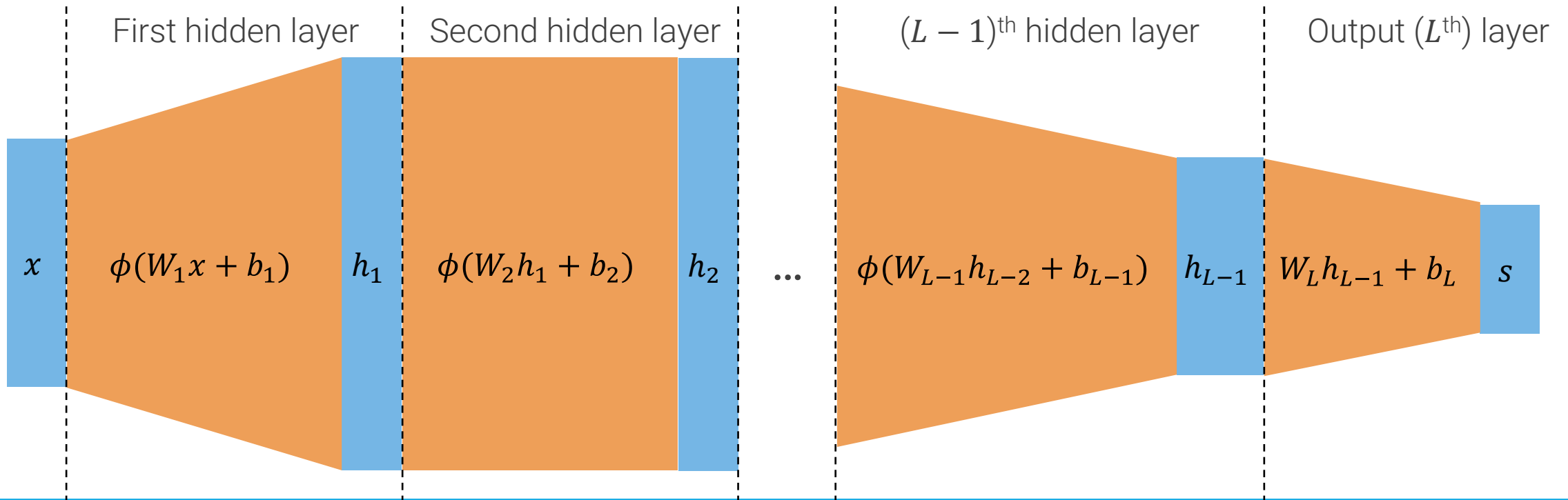


“Width” of neural networks

the WIDTH is the number of neurons in that layer. Each neuron produces one activation value, so the total number of activations in a layer equals the number of neurons in that layer.

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

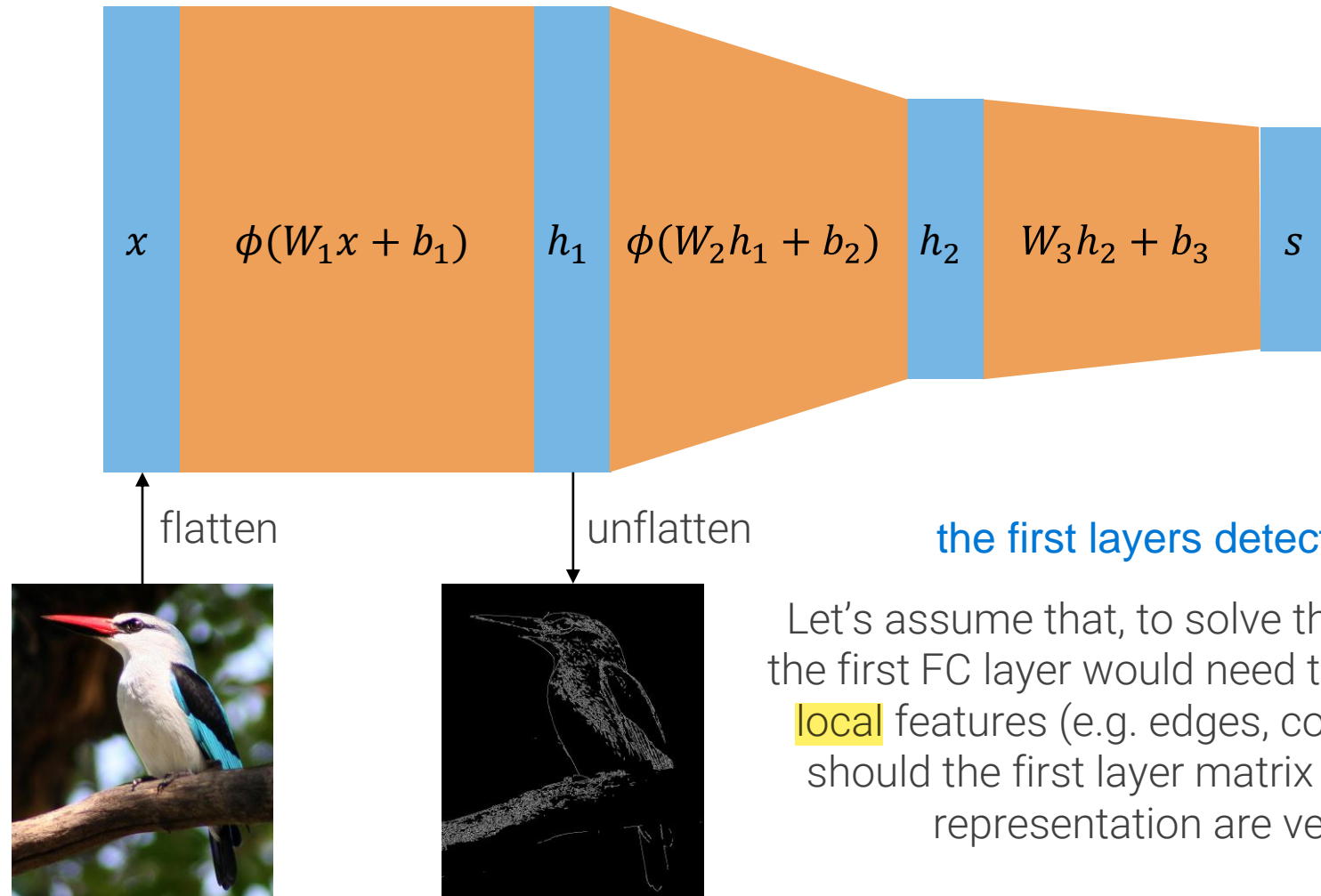
The number of activations computed by each layer, i.e. the length of h_i , is the **width** of the network



Limits of fully connected layers

a NN like this can learn any function

approximations



the first layers detect high-level features

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 $H \times W$, the layer requires

- $(H \times W) \times (H \times (W - 1)) \approx H^2 W^2$ parameters/memory
- $2(H \times W) \times (H \times (W - 1)) \approx 2H^2 W^2$ 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

2 billion params
just for vertical edges

-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

It must learn the same and sparse feature extractor in every row

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

fully connection is a problem, it forces me to look at all pixels in the image in order to detect vertical edges

my model contains general rules about the data, and it is learning from the data, but this also comes with the inductive biases that we design

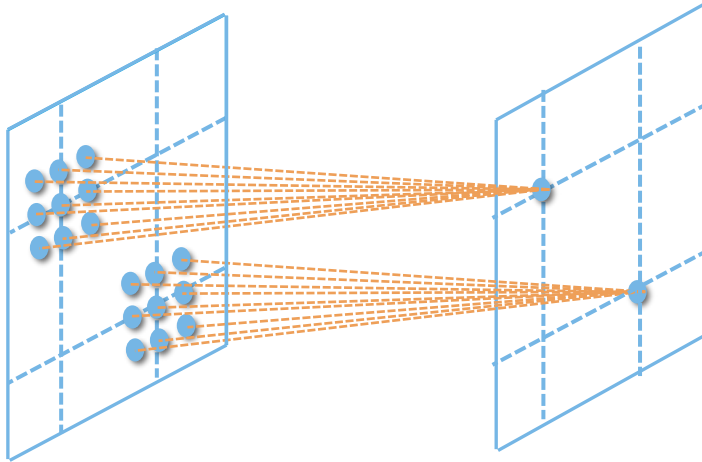
inductive bias about images, that reflects our belief: information between images tend to be local

Convolutions/correlations

inductive bias: whatever forces our model to induct something modelling our model in relation to out belief

we use INDUCTION 'coz we want to GENERALIZE

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.

we perform the same operation in each part of the image -> same params -> shared params

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

it spares me GPU time to also detect horizontal edges

Vertical edge detection with correlation/convolution

it is a more efficient and practical way to detect edges, for example

-1	1
----	---

★

correlation

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

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

RGB

-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

=

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

Flatten image

×
matrix
product

a
b
c
d
e
f
g
h
i

=

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

Flatten unflatten

Correlation or convolution?

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

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

Flipping the kernel makes the operation symmetrical and ensures proper alignment of the kernel with the input

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**

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

correlation is like convolution but without flipping the kernel

Flipping isn't necessary in CNNs because the primary goal is feature extraction, where the direction of the filter is not as important as in signal processing

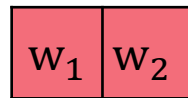
Convolution as matrix multiplication

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

The resulting matrix is still a **linear operator**, which: **so they are linear transformations**

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

I don't use a huge linear operator because it's too powerful:
it has more degrees of freedom
- this could lead to more chances of overfitting



w_1	w_2	0	0	0	0	0	0	0
0	w_1	w_2	0	0	0	0	0	0
0	0	0	w_1	w_2	0	0	0	0
0	0	0	0	w_1	w_2	0	0	0
0	0	0	0	0	0	w_1	w_2	0
0	0	0	0	0	0	0	w_1	w_2

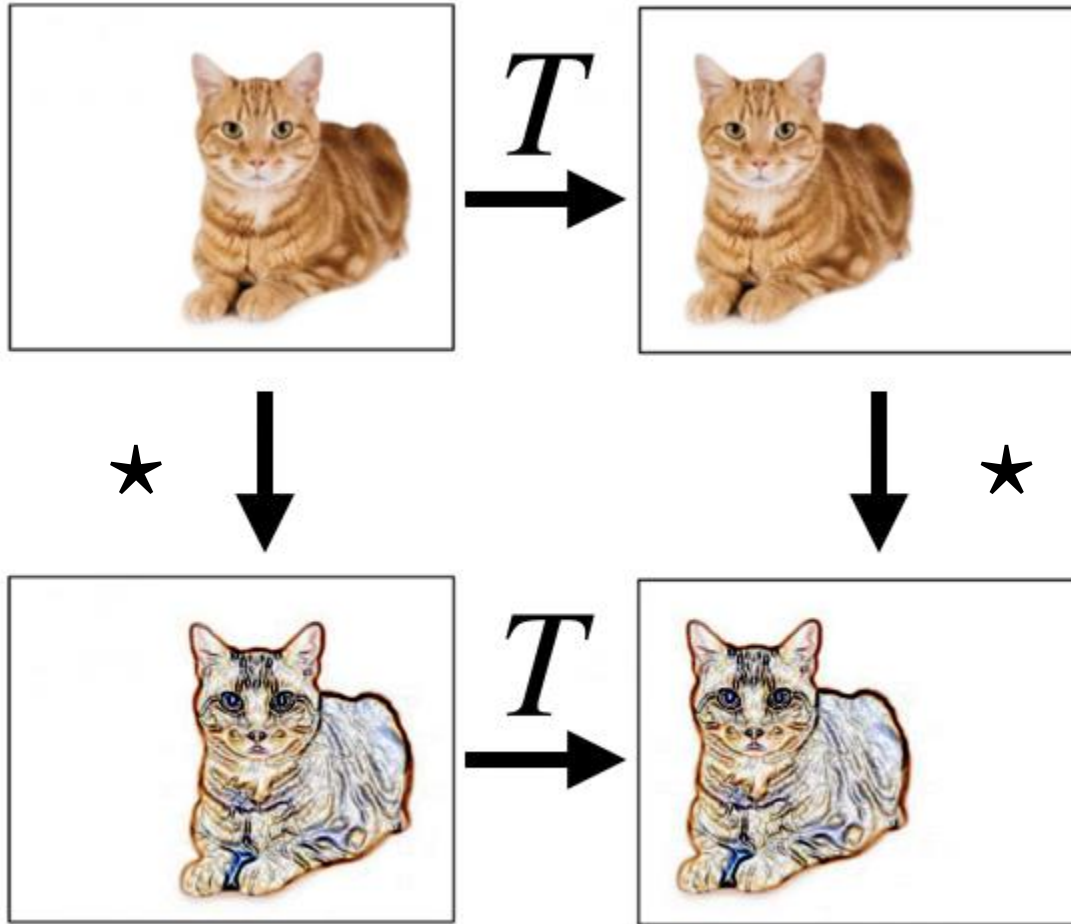
share weights since they are always the same

use inductive bias that **CONSTRAINS** the model without learning spurious correlations within training data

this makes learning easier

Equivariance

i dont care about the position of the cat



bruteforce variance:

data augmentation in training set to let the model learn the variance of data, since correlation is not rotation and scaling invariant, unlike translation

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.

because of the parameter sharing in convolutional layers, a feature learned at one location in the image can be detected at any other location without needing to learn new parameters for each location.

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

Multiple input channels

inputs would not be images, but tensors

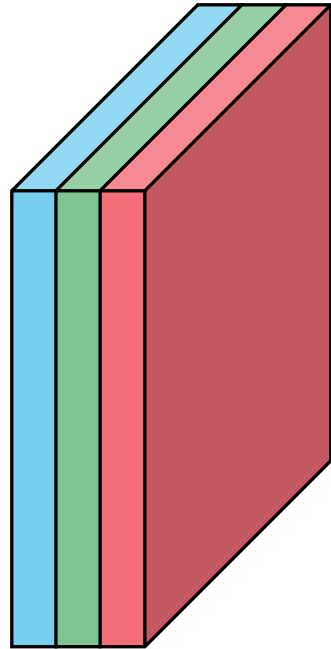
Images have 3 channels, so convolution kernels will be 3-dimensional tensors of size $3 \times H_K \times W_K$ and

3 loops: 2 spatial dim and one over channels

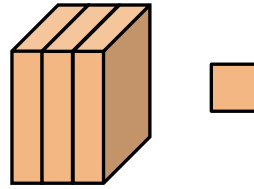
$$[K * I](j, i) = \sum_{n=1}^3 \sum_m \sum_l K_n(m, l) I_n(j - m, i - l) + b$$

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**



Input image,
e.g. $3 \times 32 \times 32$



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

kernels have a hidden dimension which express depth, channels NN will decide kernel thorough learning , we dont manually decide it like in part1

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

inductive bias: locality expressed by 5x5 kernel

Output activation

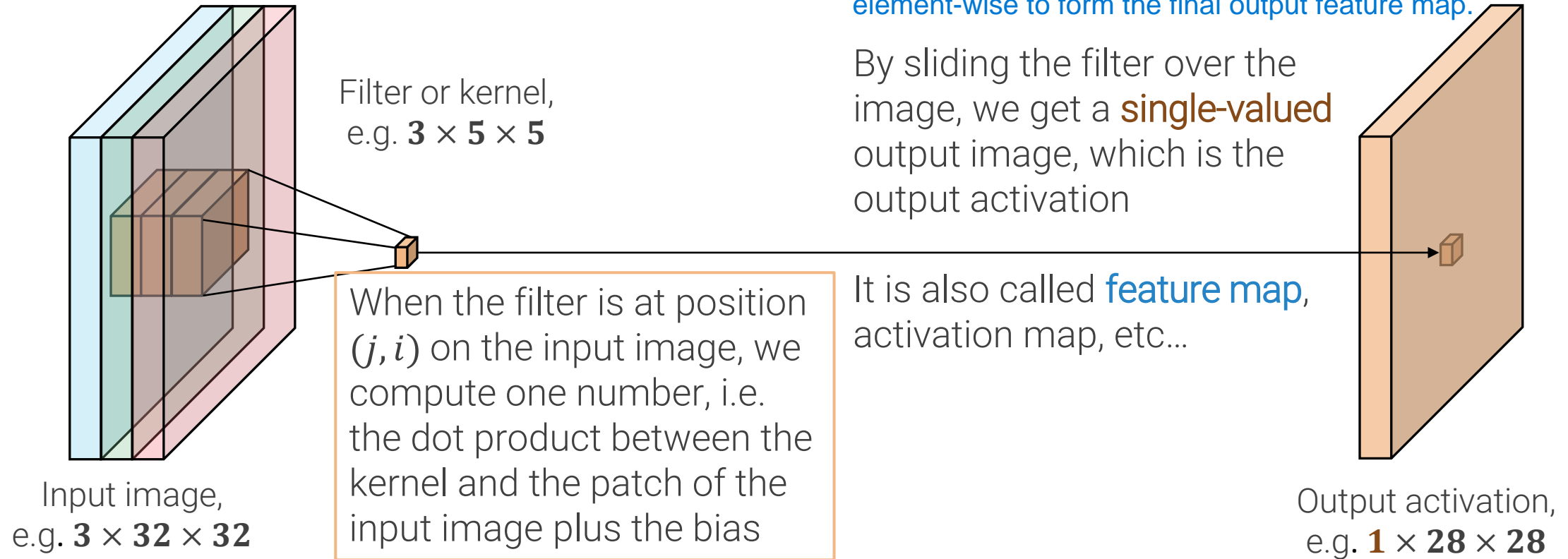
I compute conv with kernel in every channel
how? - sliding over spatial dimensions

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

the activation maps for each channel are summed together element-wise to form the final output feature map.

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...

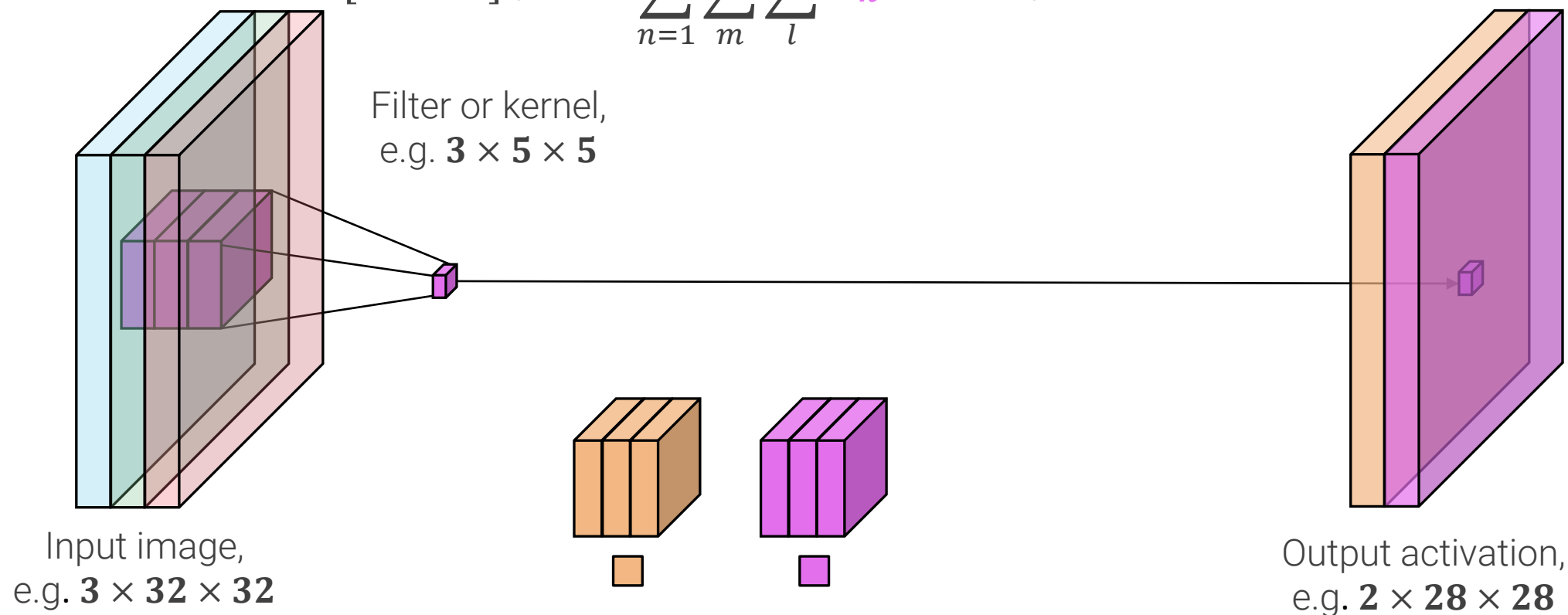


Multiple output channels

a 2nd filter which produces a different feature map

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

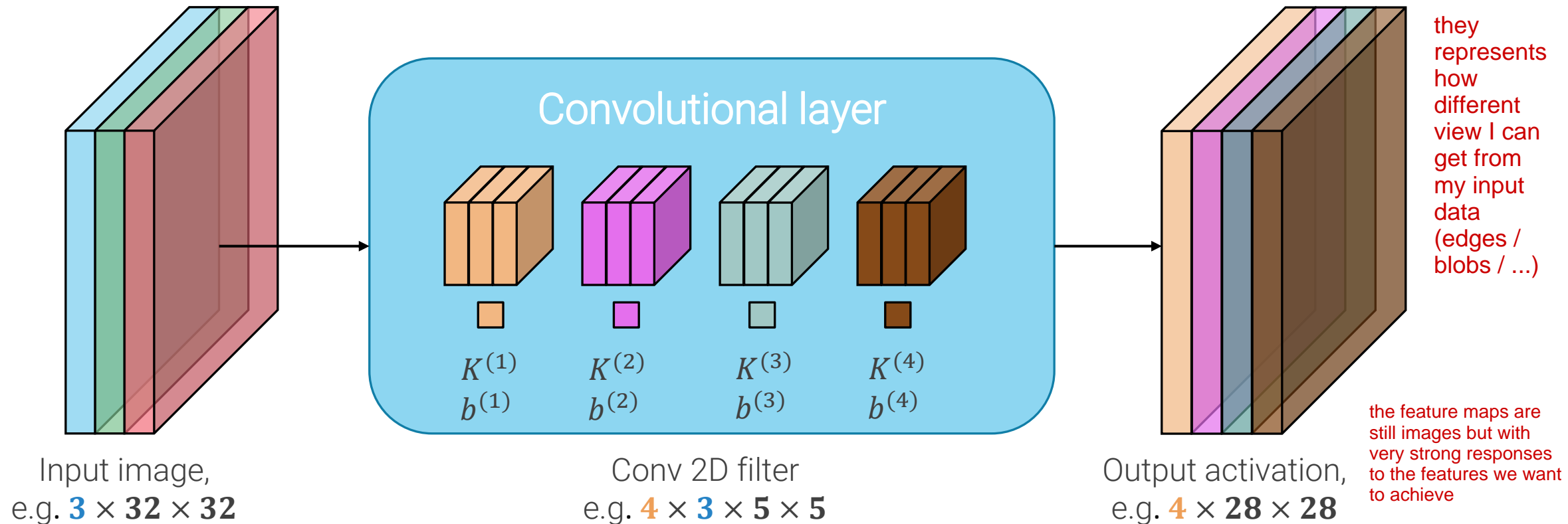
$$[K^{(2)} * I](j, i) = \sum_{n=1}^3 \sum_m \sum_l K_n^{(2)}(m, l) I(j - m, i - l) + b^{(2)}$$



Convolutional layer

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

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

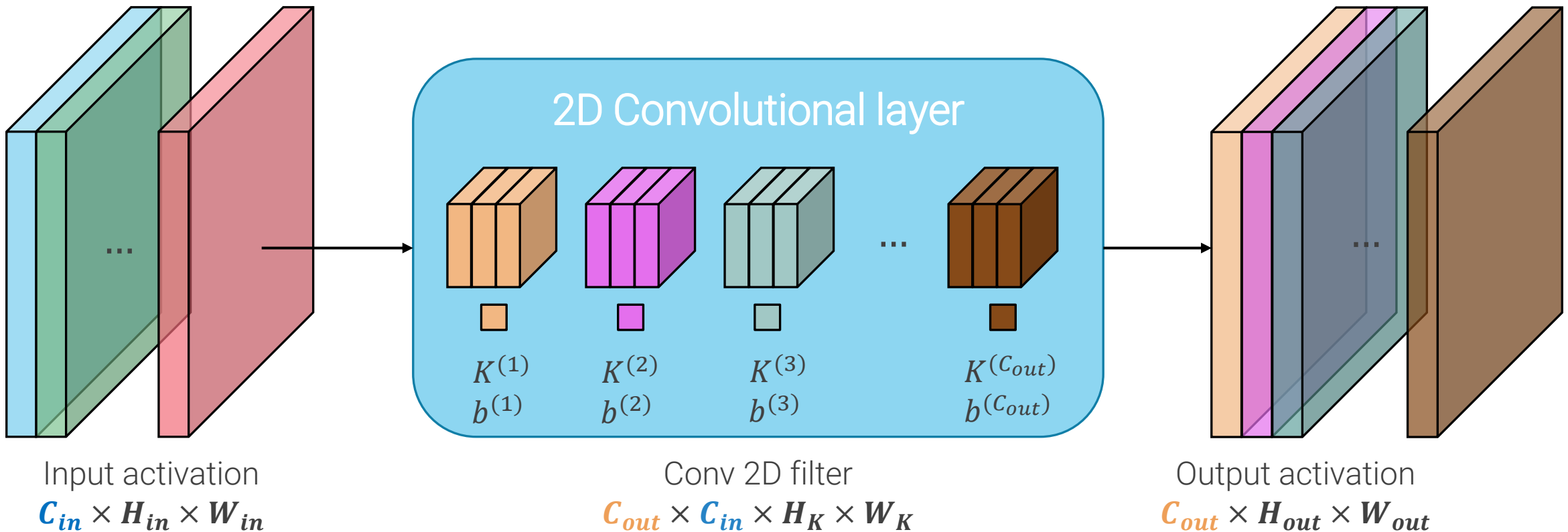


Convolutional layer

- C_{out} is the number of different kernels used
- C_{in} is the number of channels of the input image/activation, namely also the depth of each kernel

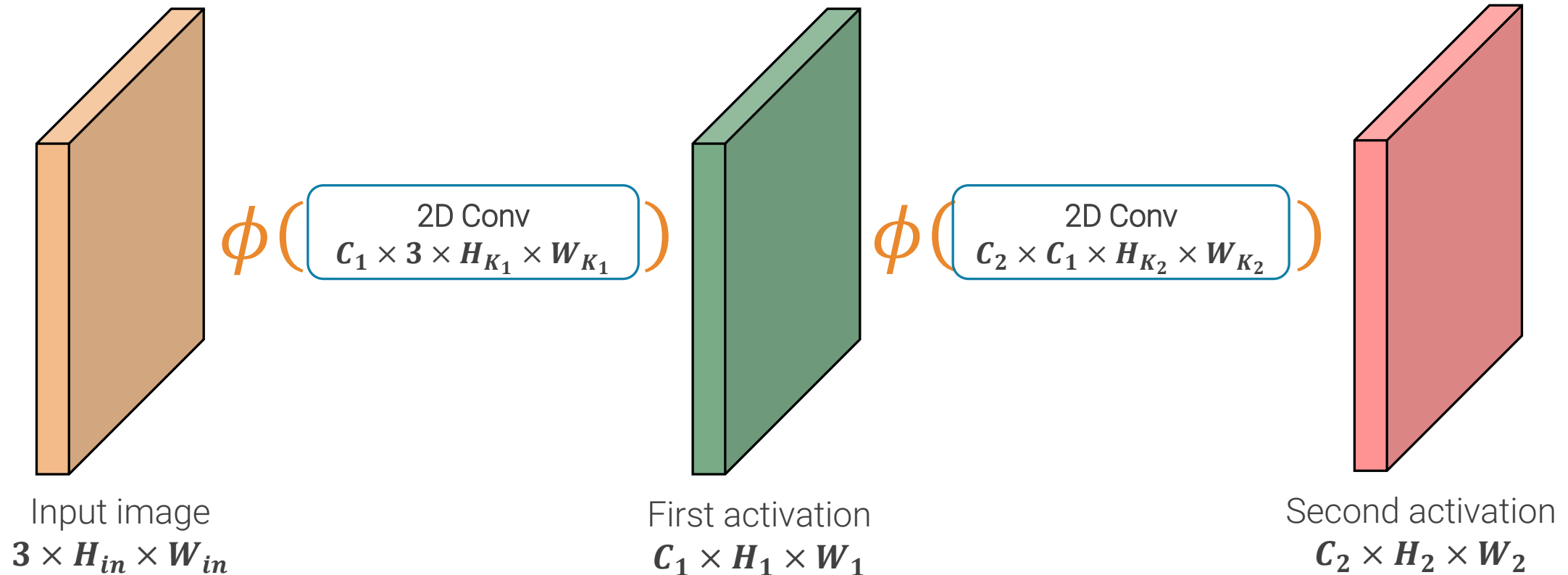
In the general case, we compute C_{out} convolutions between **vector-valued** kernels and input activations

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



Multiple convolutional layers

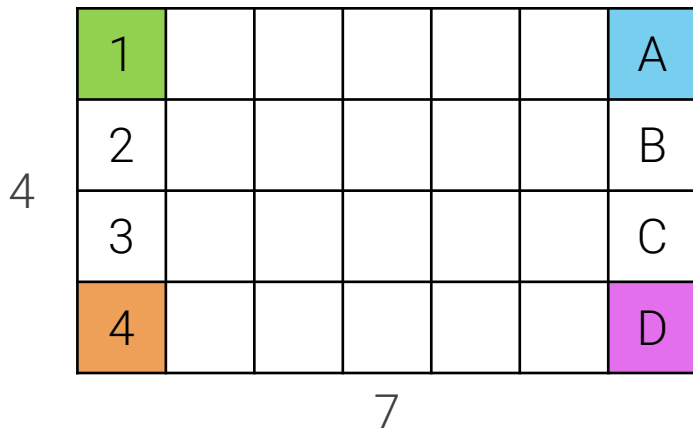
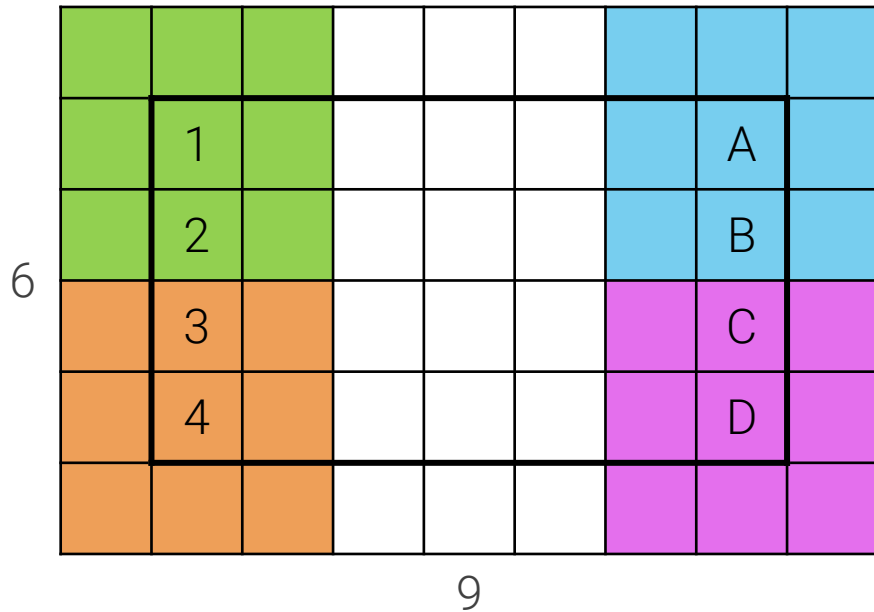
they are still linear transf, we use non linear activation functions between them



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



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

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

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

here no padding

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".

I am applying kernel only in valid position of input image

Zero Padding

add padding in order to avoid to shrink the image

changing type of padding does not change too much to classification tasks -> those pixels are not that much important

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

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

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

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

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

stacking convolutions

I augment the number of stackings because receptive field grows too slowly, not because I want to recombine features a lot of times

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

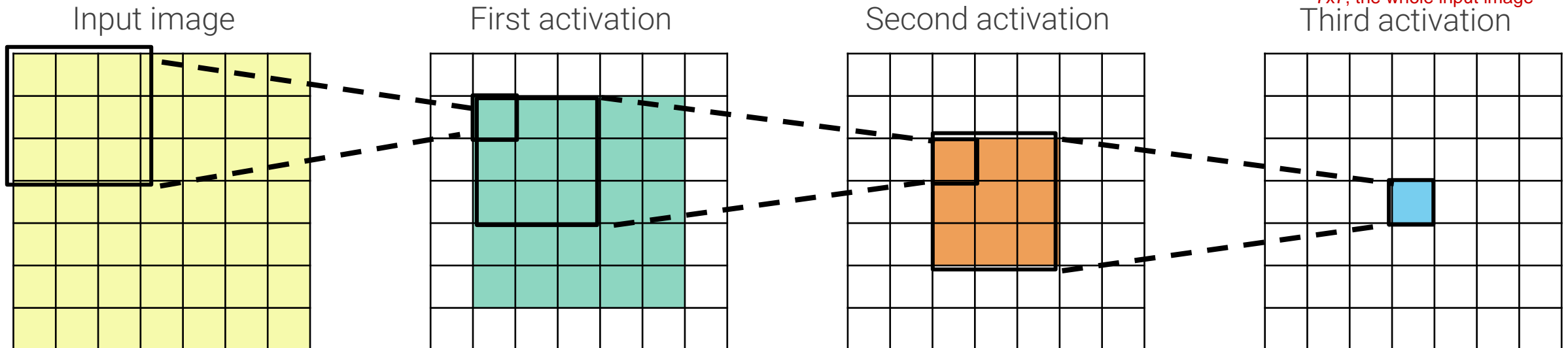
now we refer to receptive fields to the pixels that influences a neuron

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

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

this grows linearly with the number of layers

Each side grows linearly with the number of layers $L \rightarrow$ to compute attributes 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

they downsample the input as pooling does

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

9+2

6
+
2

First row
of output

1

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

$$H_k \times W_k = 3 \times 3, P = 1, \text{stride } 2, S = 2$$

STRIDING: it refers to the step size with which the convolutional filter moves across the input image or feature map. Striding affects how the filter traverses the image and how much the output is downsampled

Strided convolutions

0	0	0	0	0	0	0	0	0	0	0
0	1		2							0
0										0
0										0
0										0
0										0
0										0
0	0	0	0	0	0	0	0	0	0	0

9+2

6
+
2

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

stride 2

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

First row
of output

1	2
---	---

Strided convolutions

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

9+2

6
+
2

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

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

First row
of output

1	2	3
---	---	---

Strided convolutions

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

9+2

6
+
2

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

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

First row
of output

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

Strided convolutions

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

6
+
2

9+2

First row
of output

1	2	3	4	5
---	---	---	---	---

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

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

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

In general

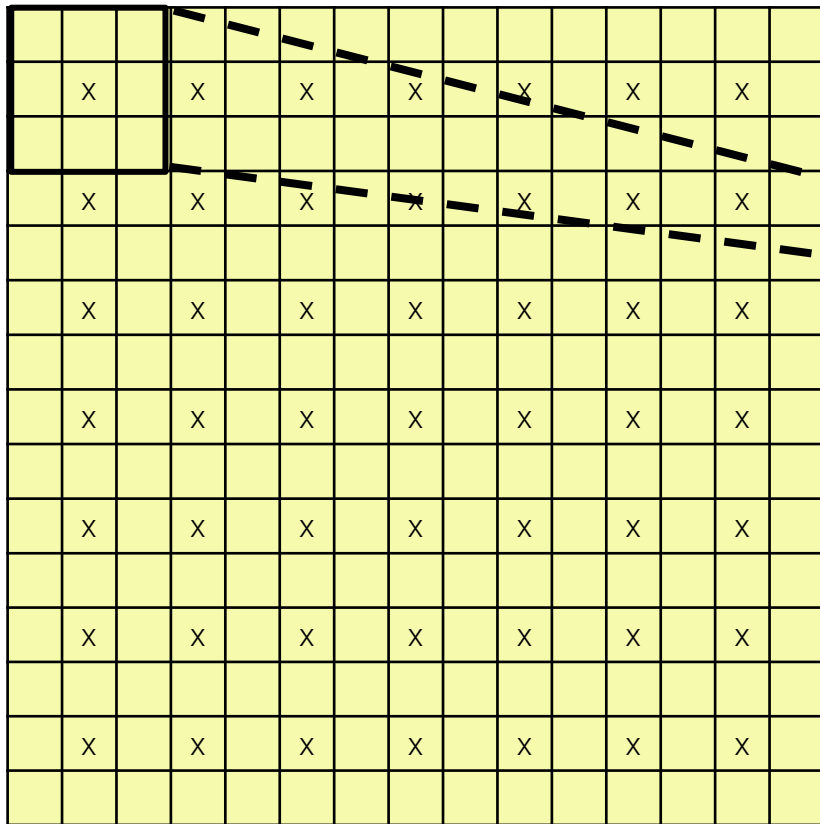
$$H_{out} = \left\lfloor \frac{(H_{in} - H_K + 2P)}{S} \right\rfloor + 1$$

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

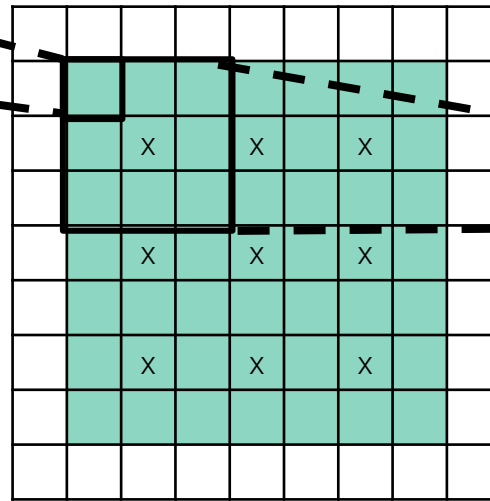
Animations describing the effects of all these parameters can be found at https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md

Receptive field with stride $S=2$

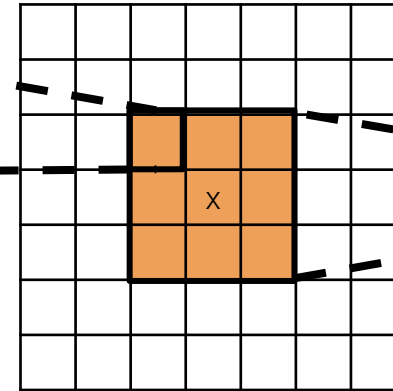
Input image



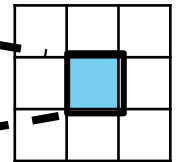
First activation



Second activation



Third activation



With a stride greater than 1, each neuron skips some of the input pixels, covering a larger region of the input space in fewer steps. This skipping causes the receptive field to grow more quickly.

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

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 stride constant

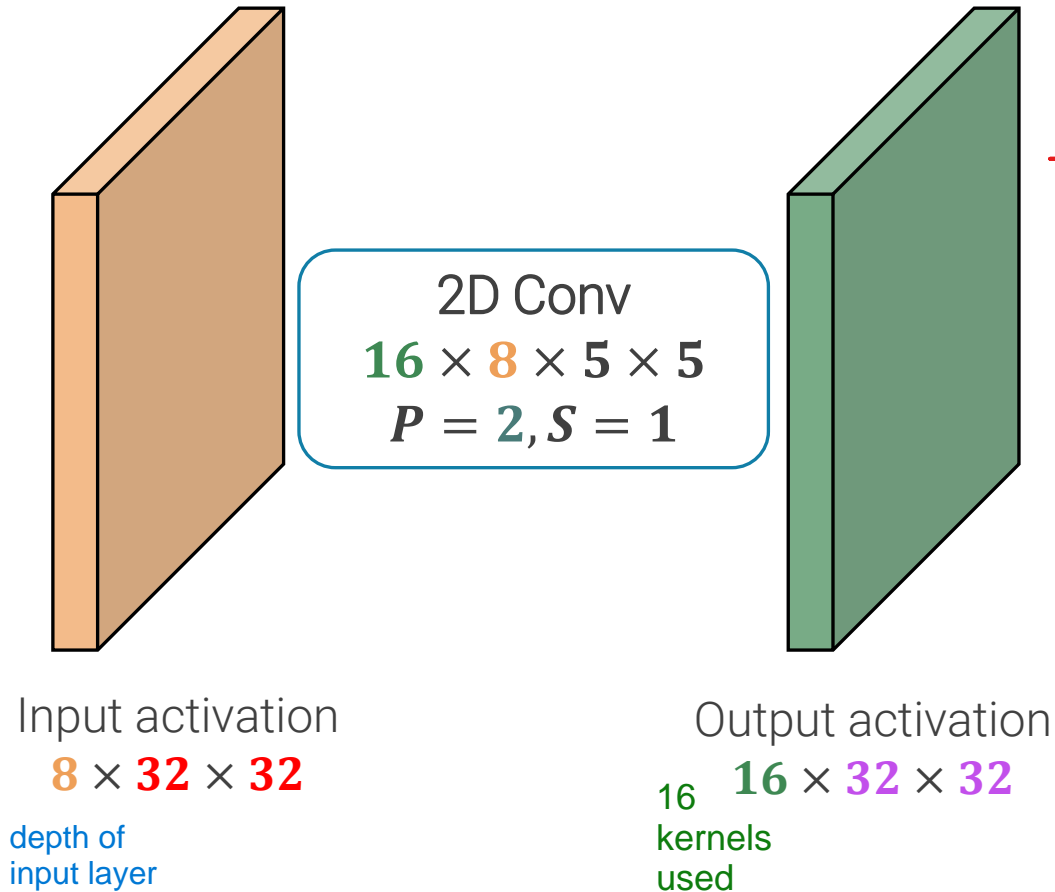
If for all layers we have $S_l = S$, then $\prod_{i=1}^{l-1} S_i = S^{l-1}$ and

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

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

Convolution parameters: example

padding and stride enlarge memory occupation



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

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

all the conv layer dims are multiplied together

learnable parameters

→ The size of the output for this example is

all the output activation dims are multiplied together

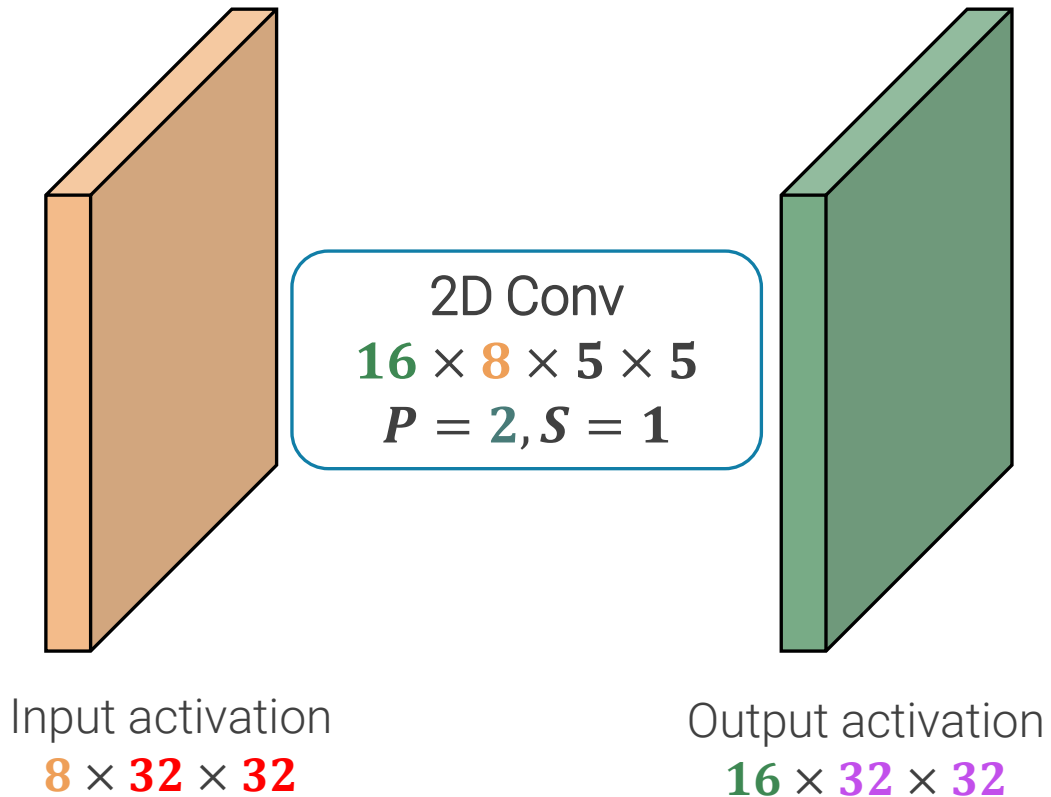
$$H_{out} = W_{out} = \frac{32 - 5 + 2 * 2 + 1}{1} = 32$$

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



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

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

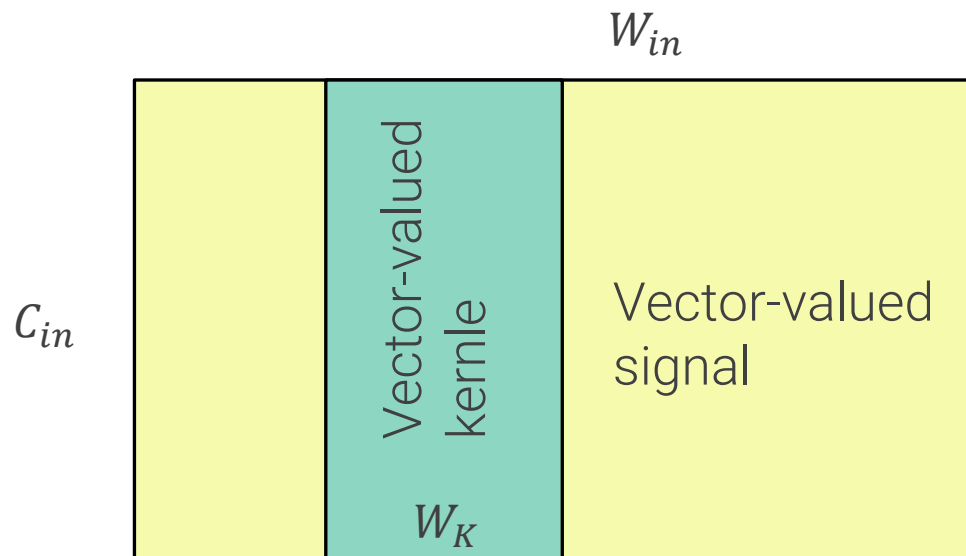
Other common convolutions

conv can be also 3d conv

video for example

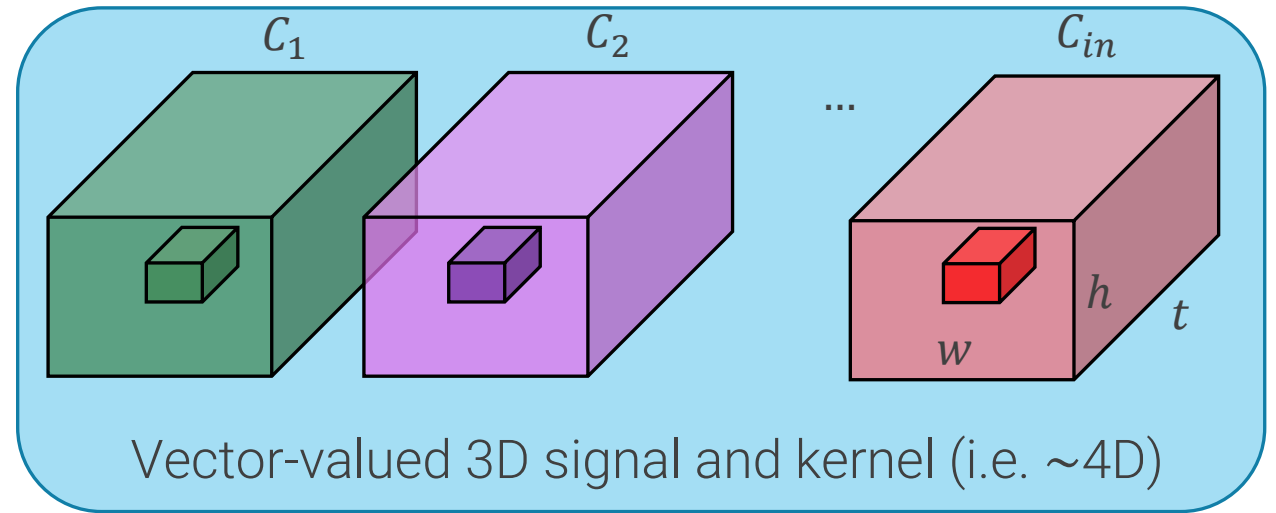
1D convolutions

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



3D convolutions

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



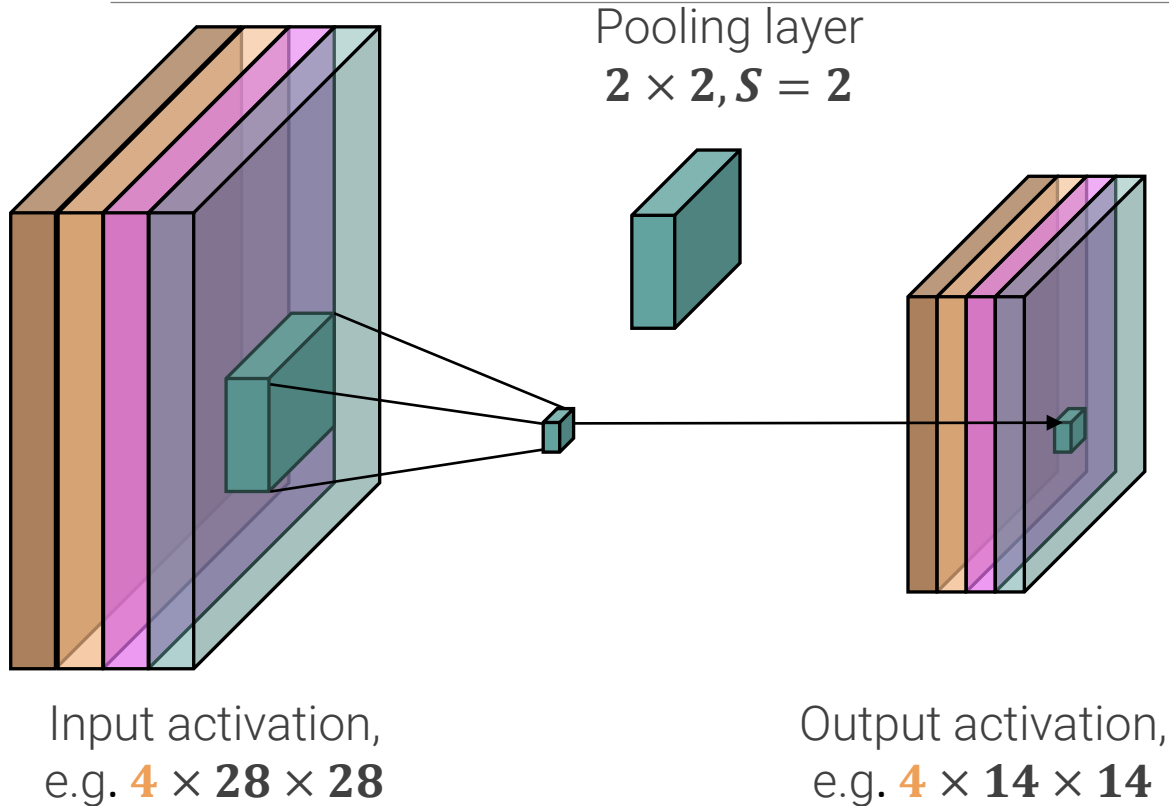
Common layers in CNNs

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

Pooling is used primarily for downsampling the spatial dimensions (height and width) of feature maps while preserving important information. Pooling helps reduce the number of parameters and the computational complexity of the network, to control overfitting, and achieve spatial translation invariance (as we'll see in the next slide)



Hyper-parameters

Kernel width and height $W_K \times H_K$

Kernel function (pre-specified): max, avg, ...

Stride S (usually >1)

Common choice:

$2 \times 2, S = 2$, kernel func= max

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

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

- * E.G. detecting eyes: it doesn't matter if eyes are shifted, the output actually summarizes strong response wrt eyes:
 - the 6 can be either at top left of the orange box or at its bottom right, the response would be always 6

→ Emphasizes the presence of the most dominant feature in the pooling window

Single input channel C_i

-1	6	3	2
0	-1	4	7
0	0	-2	-1
0	0	1	0

Max pooling
2x2, S=2

Single output channel C_i

6	7
0	1

“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

is this empirical, even if theoretically max pooling loses information etc.

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)

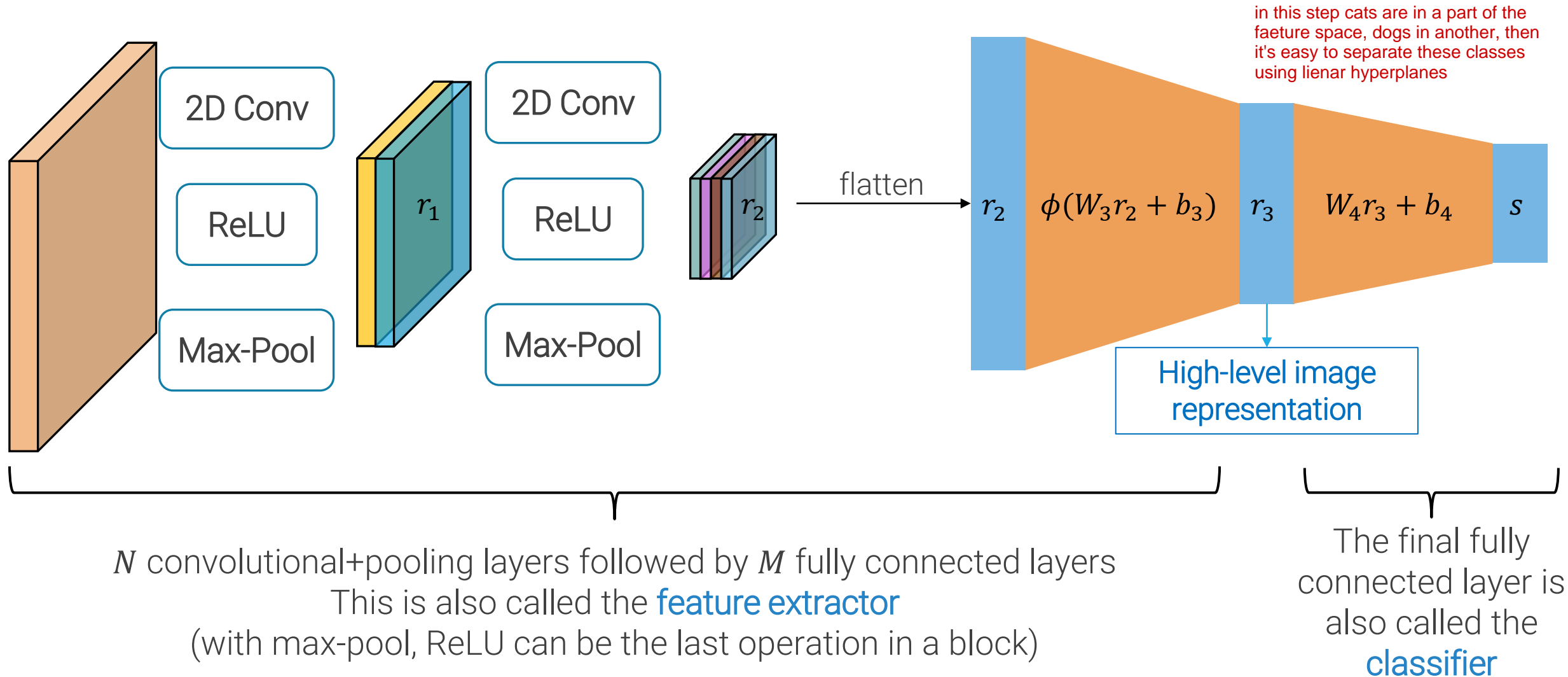
○ provides **invariance** to small spatial shifts. * we get the same responses no matter where we are in the image

- we mostly do maxpooling at the beginning of the network
- we used avg pooling to speed up training when people had not so powerful machine to run NNs

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

Convolutional Neural Networks

VANILLA ARCHITECTURE



If the inputs to each layer (activations from the previous layer) have very different scales, the internal covariate shift becomes more pronounced. This is because changes in the weights during training can cause large fluctuations in activations, leading to instability

Internal Covariate Shift

This term refers to the phenomenon where the distribution of activations (outputs of neurons) changes during training as the network parameters are updated. This shift in distributions occurs not just because of varying input feature scales, but more broadly due to parameter updates across all layers, which cause the input distributions of deeper layers to change over time.

→ the distribution of the activations become unstable

$$f(\mathbf{x}; \theta) = \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2$$

\mathbf{h} was computed by the previous layer, where \mathbf{W}_1 and \mathbf{b}_1 "are"

$$= \mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

internal covariate shift:

since \mathbf{h} is changing due to updates in \mathbf{W}_1 and \mathbf{b}_1 , the distribution of \mathbf{h} that \mathbf{W}_2 and \mathbf{b}_2 see is not stable

Problem: even when using ReLUs instead of sigmoids, and when properly initializing all the layers, **deep architectures** following the pattern presented in the previous slide **were very hard and/or slow to train**.

→ One (maybe not correct...) intuition: if we consider the $\mathbf{W}_2, \mathbf{b}_2$ linear classifier in the neural network above, the (distribution of the) representation \mathbf{h} it tries to classify changes at each training iteration, because \mathbf{W}_1 and \mathbf{b}_1 are updated. Moreover, the gradient \mathbf{W}_2 and \mathbf{b}_2 receive was computed to improve performance for the "old" distribution of \mathbf{h} . This effect becomes worse when multiple layers influence the changes of \mathbf{h} .

This phenomenon is referred to as **internal covariate shift**: the change in the distribution of network activations due to the change in network parameters during training.

When the activations change significantly due to internal covariate shift, the gradients computed based on these activations may not accurately guide parameter updates. This discrepancy can lead to ineffective optimization and slower convergence during training.

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

BatchNorm aims to address int. cov. shift by normalizing the activations within each mini-batch.

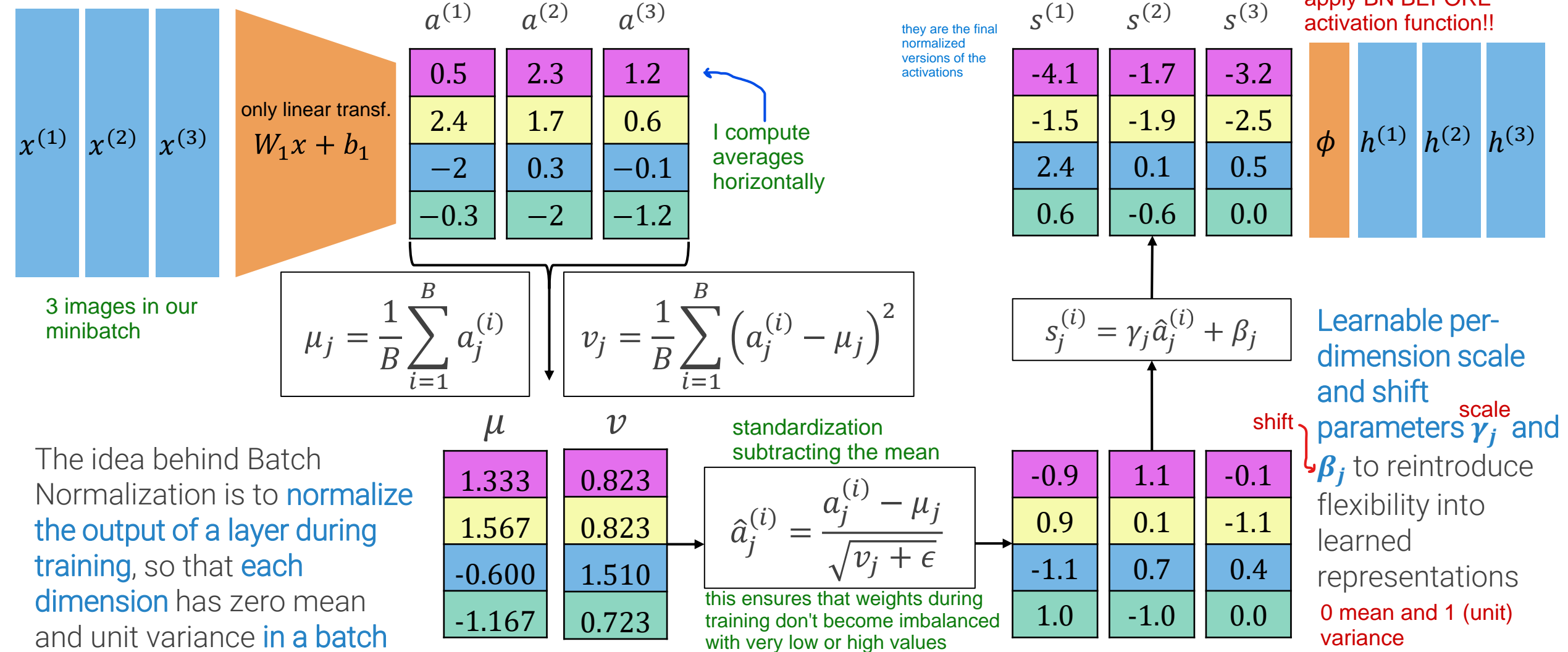
the batch used is the current one use by SGD!!

for each channel, BatchNorm computes the mean and variance across the entire batch, considering all instances (images).

Batch norm layer – training time

there is nothing to learn, no params

keeps under control distribution of activations



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

By normalizing activations, BatchNorm ensures that the scale of the inputs to each layer is consistent. This helps in reducing the impact of varying feature scales, as the normalized activations no longer vary widely in scale. This leads to more balanced gradients and weight updates.

Batch normalization at training time

batch normalization isn't neither well theoretically supported, nor empirically

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

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

Four steps: **all of them are differentiable** and can be back-propagated through: **it guaranties that optimization does not “undo” normalization** not only normalizes the activations but also includes mechanisms to ensure that the network can still learn effectively despite the 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.

explained in the next slide

At training time, we also **keep a running average of mean and variance** ($\beta = 0.1$ is usually called **BN momentum**)

$$\begin{aligned}\mu_j^{(t)} &= (1 - \beta) \mu_j^{(t-1)} + \beta \mu_j, \\ v_j^{(t)} &= (1 - \beta) v_j^{(t-1)} + \beta v_j\end{aligned}$$

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: $B \times D$

these op. are all differentiable and they must be within the NN

"deterministic" means that the output of the normalization process is predictable and fixed given a specific input

Batch normalization at test time

we don't have to optimize anything 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 $\beta_j, j = 1, \dots, D$

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

then I don't pay the cost of having batch norm

μ_j = final value of the running average computed at training time .
It's constant at test time.

Shape: $1 \times D$

v_j = final value of the running average computed at training time .
It's constant at test time.

Shape: $1 \times D$

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

Shape: $B \times D$

i have to keep a running mean and variance across all the batches because at test time I want an overall result, a constant mean and variance

Batch norm: pros and cons

on the x-axis we have steps, namely how many times we update parameters

because as we said before, epochs are not that much informative wrt steps

Pros training is faster and more stable

- 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

training inject noise, not determ: the same image produce different activations during training process

it's more important to have batch norm not early in the network, since it's not that much to normalize

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"

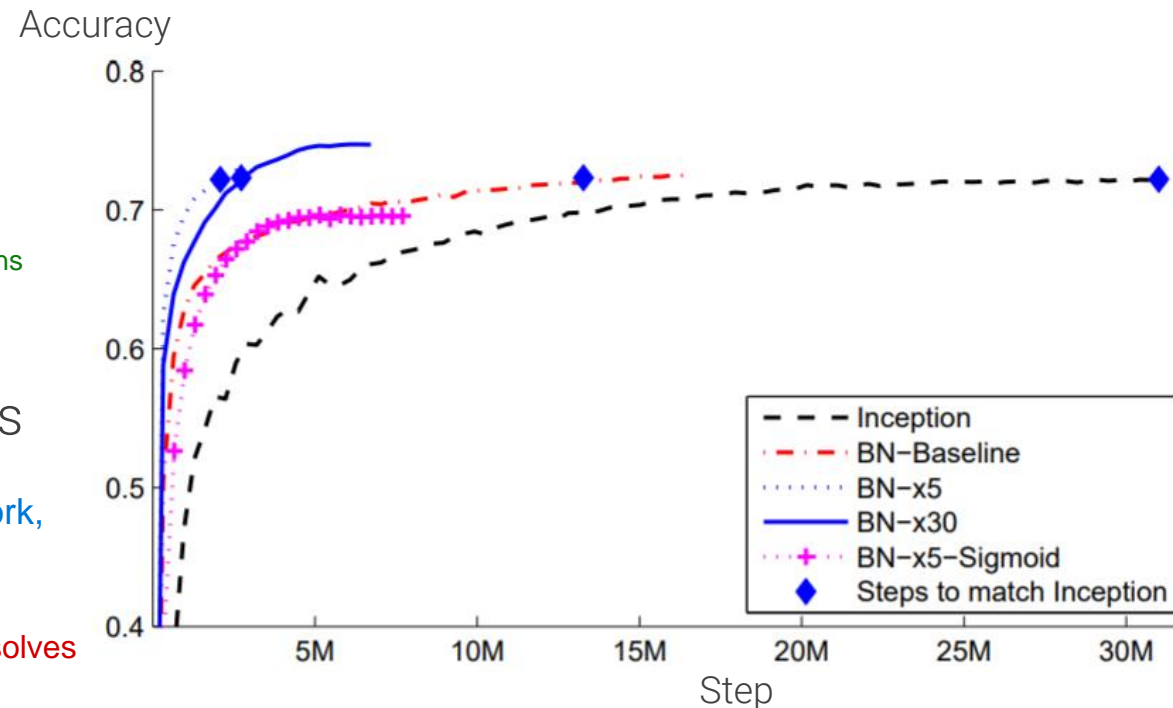
we don't actually know if it solves internal covariant shift

it isn't statistically reliable computing a mean across 2,3,4 samples

not so democratic: if low GPU power -> micro-batches -> doesn't work so well

try hybrid approach: don't do whitening at each batch

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



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%

Batch norm for convolutional layers

Batch norm for fully connected layers

Normalize along mini-batch dimension

$$a: B \times D$$

D variances, D means, D gammas and D betas

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

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

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

this is an element-wise operation, since each entry of a has its own mean, variance etc.

C_{out} is the number of kernels used, namely the number of feature map computed by each kernel after a convolution

Batch norm for convolutional layers

Normalize along mini-batch and spatial dimensions

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

apply this to all the pixels in each channel, averaging not only on the batch dimension as usual, but also along the spatial dimension

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

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

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

I can apply batch norm also along the spatial dim because conv is translation invariant

Also known as Spatial Batch Norm, BatchNorm2D

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

What's next?

