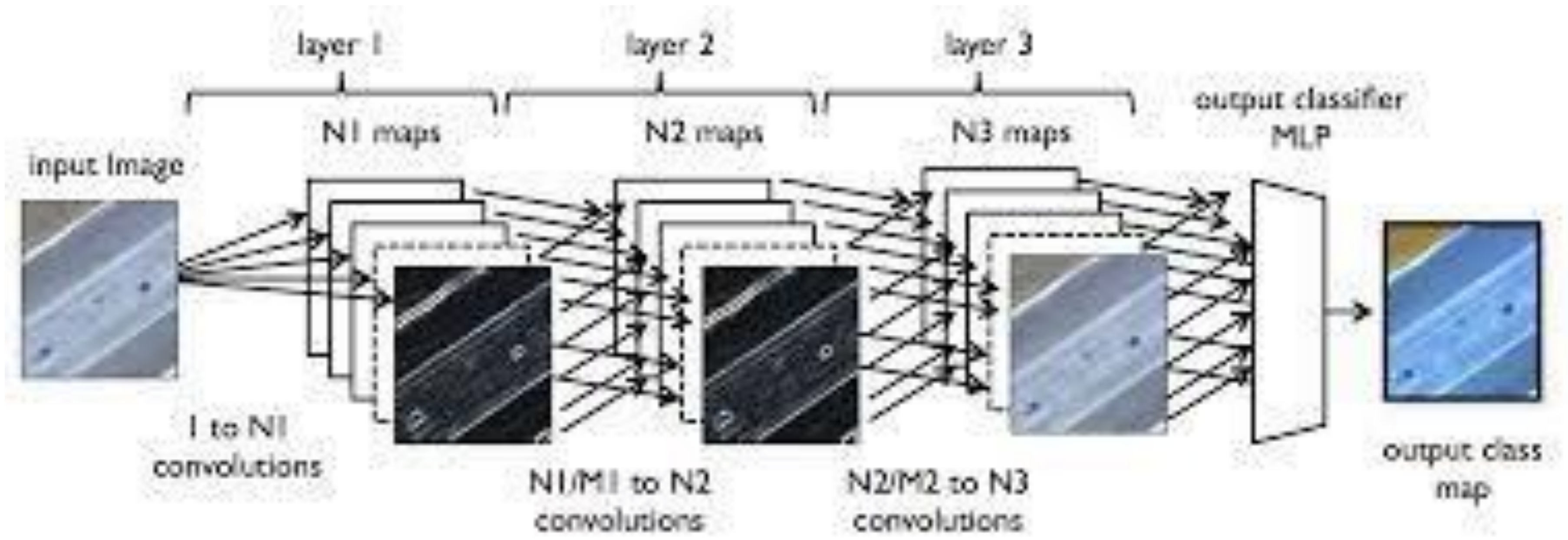
Deep Learning

a.k.a. Neural Network



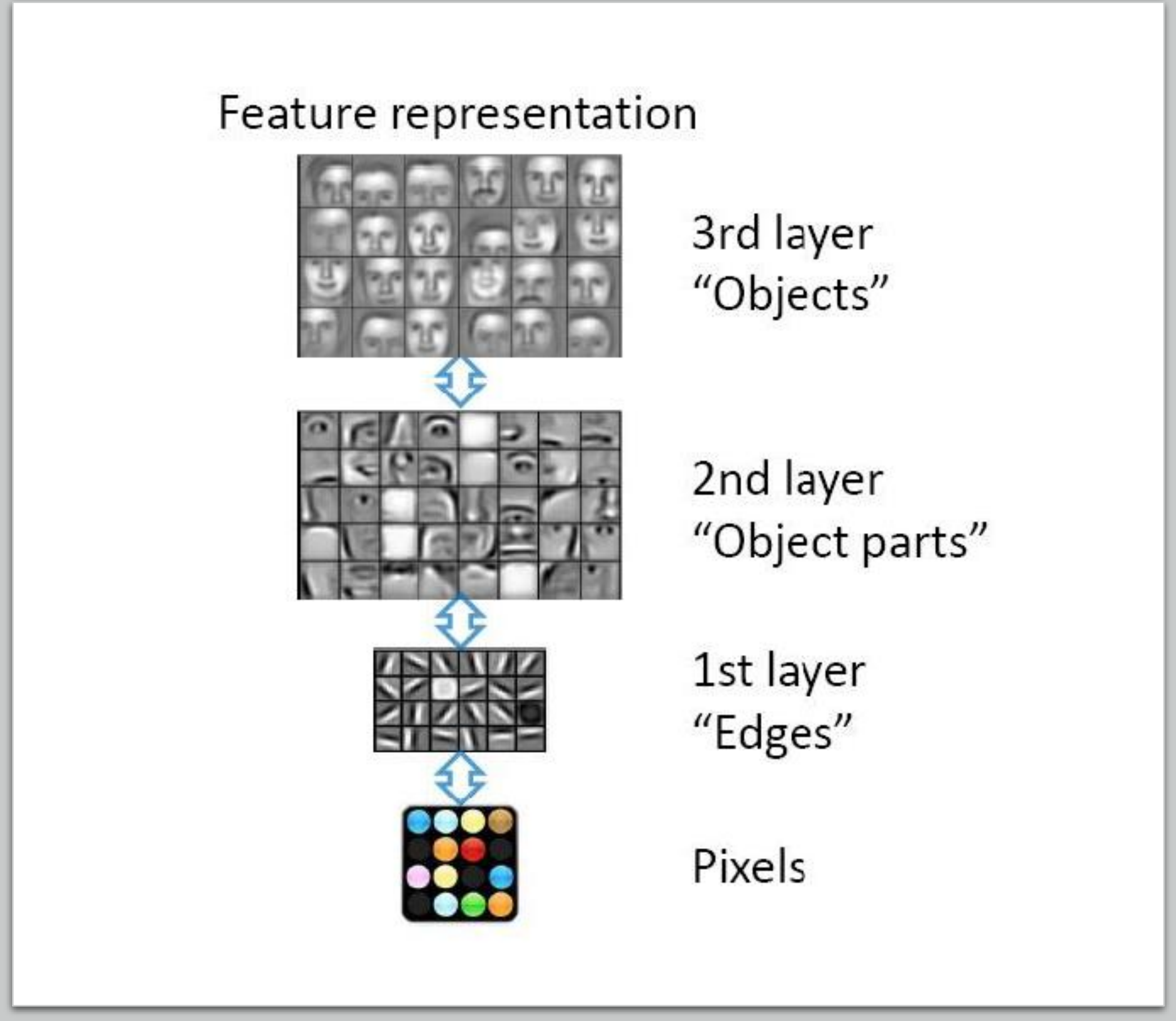
Deep neural network

- Deep NN are NN where the number of layers is much higher than in original NN (with just one or two hidden layers)
- The basic idea of deep NN is gradually «compressing» the input to obtain a (hierarchically) higher level representations of the input
- The last layer performs the prediction based on the highest level representation

Example: Image Processing

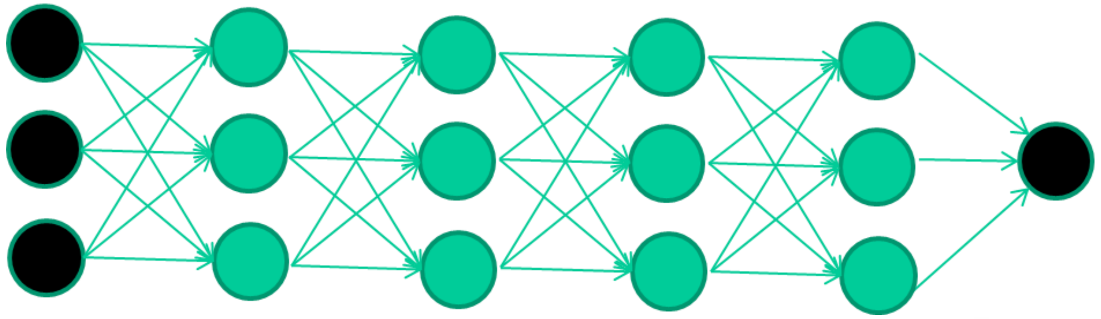
Hierarchical Learning:

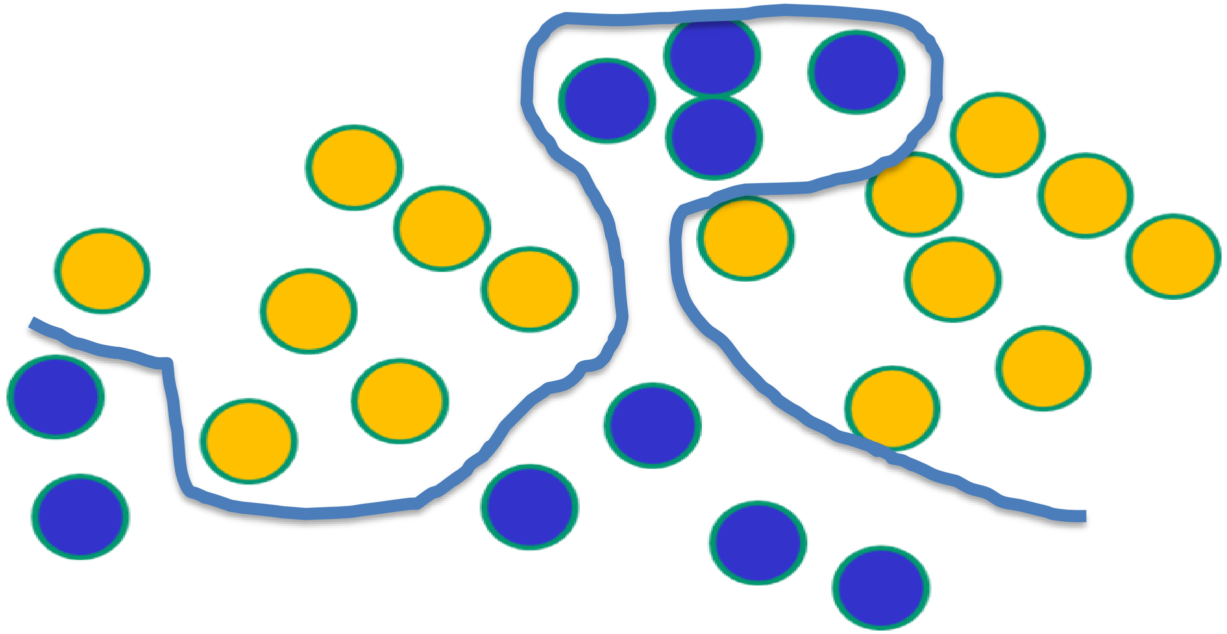
- ➢The natural progression from a low level to a higher level features as seen in many real problems



Deep learning

But, until recently, available weight-updating algorithms (e.g., backpropagation) simply did not work on multi-layer architectures. Why?





Backpropagation Drawbacks

- Based on iterative weight-updating
- NN work by making thousands and thousands of tiny adjustments to edge weights, each making the network do better at the most recent pattern, but perhaps a little worse on many others
- **Gets stuck in local minima, especially since it starts with random initialization**

Complexity in NNs

- The number of levels (depth) is only one of the sources of complexity. Others are the number of neurons, connections, and weights
- These impact both on the cost of forwarding and backpropagation
- The main problem as complexity grows is the **vanishing/exploding gradient**
- What is this?

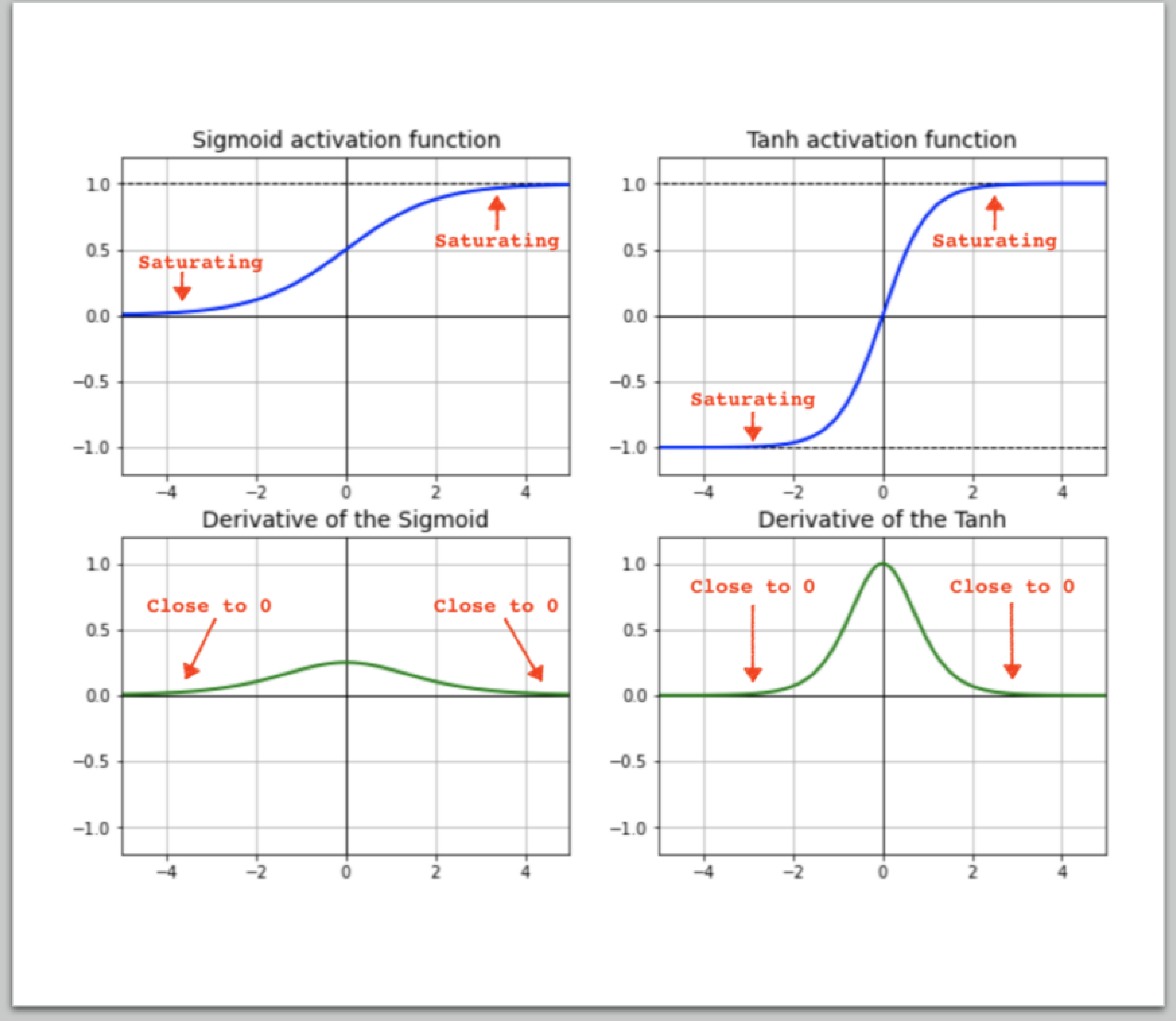
The Vanishing Gradient Problem

As more layers using certain activation functions (e.g. sigmoid) are added to neural networks, the gradients of the loss function **approaches zero**, making the network hard to train.

- In short, it is related to the fact that weights updating at each layer depends on the gradient (the *delta* function)
- Using the backpropagation, the gradient at the earlier layers is the **product of many terms from all the subsequent layers**.
- When there are **many layers**, there is an intrinsically **unstable** situation (it turns out that earlier weights gets updates much slower than the later weights).
- The mathematical motivation for instability lies in the used activation functions, like the sigmoid function, that squishes a large input space into a small input space between 0 and 1. For example, the **sigmoid function,** whose derivative has the shape of a **gaussian distribution**. (link, link)

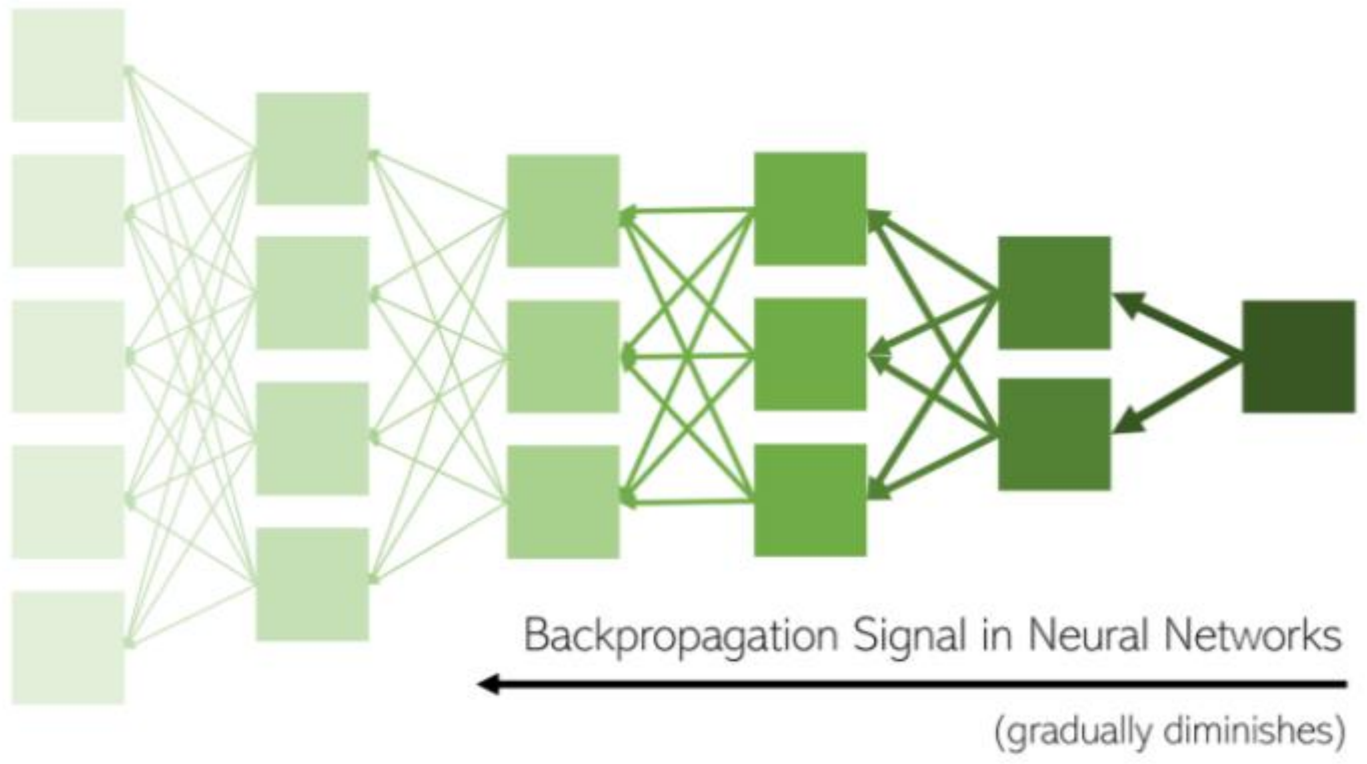
Vanishing gradient: why?

- Vanishing gradients usually happen when using the Sigmoid (or Tanh) activation functions in the hidden layer units.
- When inputs become very small or very large, the sigmoid function saturates at 0 and 1 and the tanh function saturates at -1 and 1. In both these cases, **their derivatives are extremely *close* to 0** (*saturating* regions).
- Thus, if the input (**net**$_j$) to the activation function lies in any of the saturating regions, then it has almost no gradient to propagate back through the network.



Vanishing gradient visualized

- The “useful” gradient information (the “deltas”) from the end of the network fails to reach the beginning of the network



Vanishing gradient (math)

$$\delta_j^1 = o_j^1(1-o_j^1)\sum_{j\to k} \boldsymbol{\delta_k^2}\, w_{jk}$$

$$\delta_j^1 = o_j^1(1-o_j^1)\sum_{j\to k} \boldsymbol{\delta_k^2}\, w_{jk} = o_j^1(1-o_j^1)\sum_{j\to k}\left(o_k^2(1-o_k^2)\sum_{k\to i} \boldsymbol{\delta_i^3}\, w_{ki}\right) w_{jk}$$

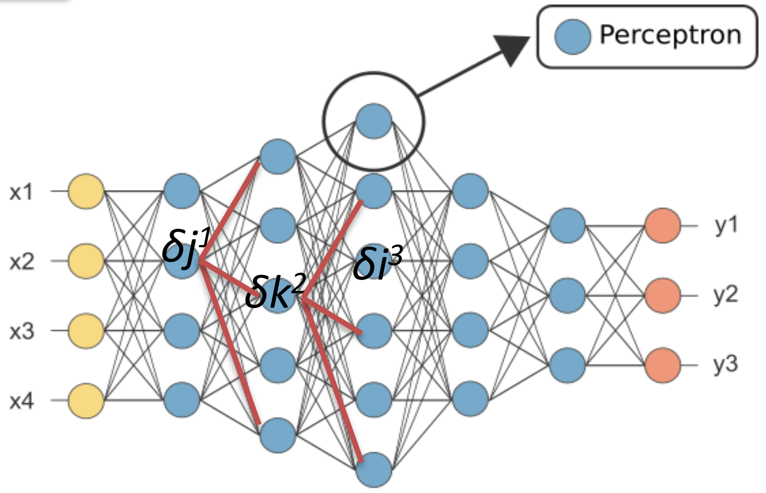
$$= o_j^1(1-o_j^1)\sum_{j\to k}\left(o_k^2(1-o_k^2)\sum_{k\to i} \boldsymbol{\delta_i^3}\, w_{ki}\right) w_{jk}$$

$$= \underbrace{o_j^1(1-o_j^1)}\sum_{j\to k}\left(o_k^2(1-o_k^2)\sum_{k\to i}\left(o_i^3(1-o_i^3)\sum_{i\to n} \boldsymbol{\delta_n^4}\, w_{in}\right) w_{ki}\right) w_{jk} =$$

Remember, the superscript here denotes the layer and [ ] are omitted for simplicity

$$\boxed{= \frac{\partial\sigma(net_{1,j})}{\partial net_{1,j}}\sum_{j\to k}\frac{\partial\sigma(net_{2,k})}{\partial net_{2,k}}\sum_{k\to i}\frac{\partial\sigma(net_{3,i})}{\partial net_{3,i}}}\sum_{i\to n}\boxed{(\boldsymbol{\delta_n^4}\, w_{in})\, w_{ki})\, w_{jk}} = ..$$

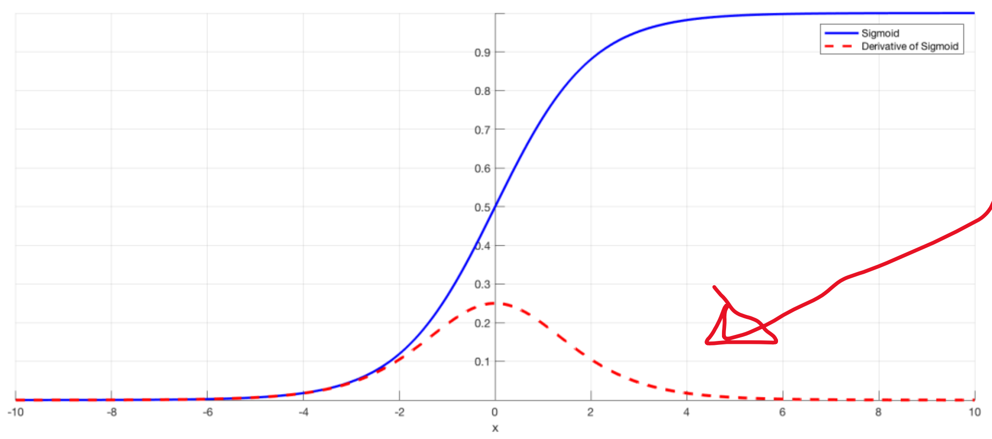
- Where σ(x) is the sigmoid function
- So, deltas at the initial layers are the results of **repetitive multiplications of derivatives of the sigmoid function** and synaptic weights, which are all between -1 and 1



Vanishing Gradient

$$\delta_j^1 = \frac{\partial\sigma(net_{1,j})}{\partial net_{1,j}} (\sum_{i\to k} \frac{\partial\sigma(net_{2,k})}{\partial net_{2,k}} (\sum_{k\to i} \frac{\partial\sigma(net_{3,i})}{\partial net_{3,i}} \left(\sum_{i\to n} \boldsymbol{\delta_n^4} w_{in}\right) w_{ki}) \, w_{jk}) = ..$$

Since weights are usually -1<w<1, and the derivative of a sigmoid σ'(x) has the following gaussian-like shape:



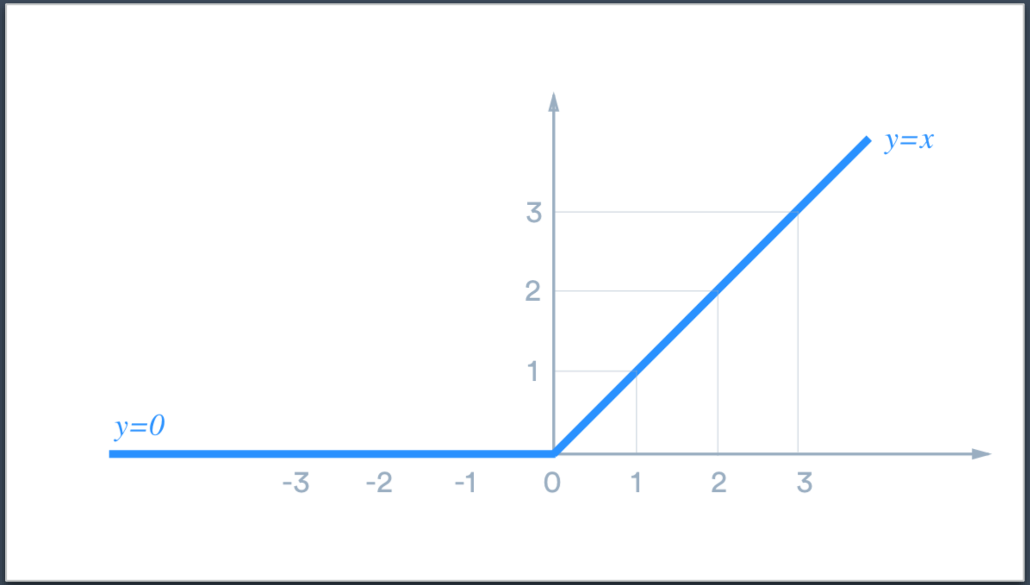
..the "deltas" at the initial layers become quickly **very small** as the number of layers increase (the product of many derivatives all much lower than one) – causing very slow convergence of weights on stable values

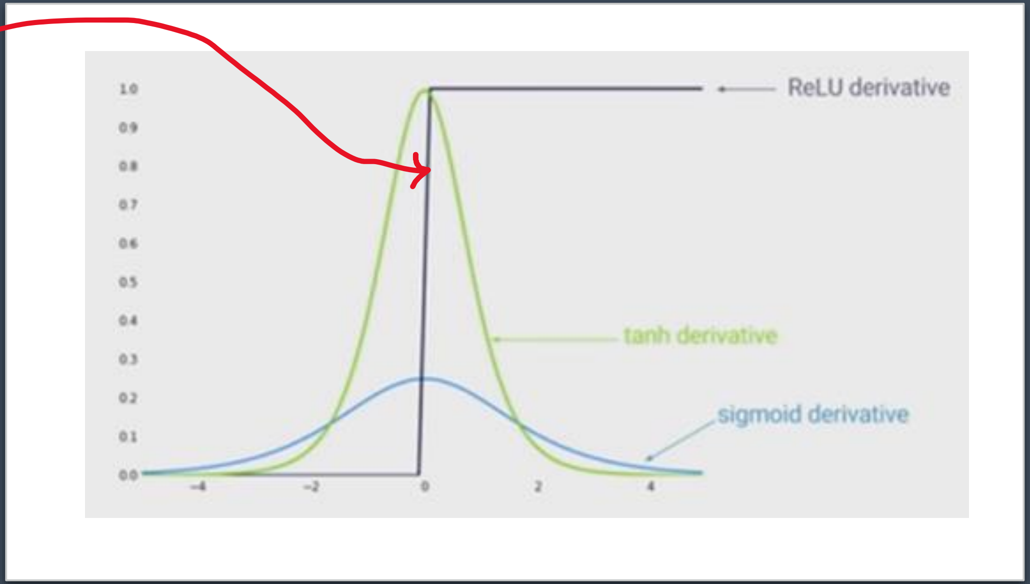
How to avoid the vanishing gradient problem

- Solutions:
  - **Simplest method:** use other activation functions which doesn’t cause a small derivative (e.g. **ReLU**).
  - **Batch Normalization layers** (link) –will see later in this course (Data preprocessing and model fitting)
  - **Residual networks**: they provide residual connections straight to earlier layers (skip connections). (link, link) – not discussed in this course

Why ReLU?

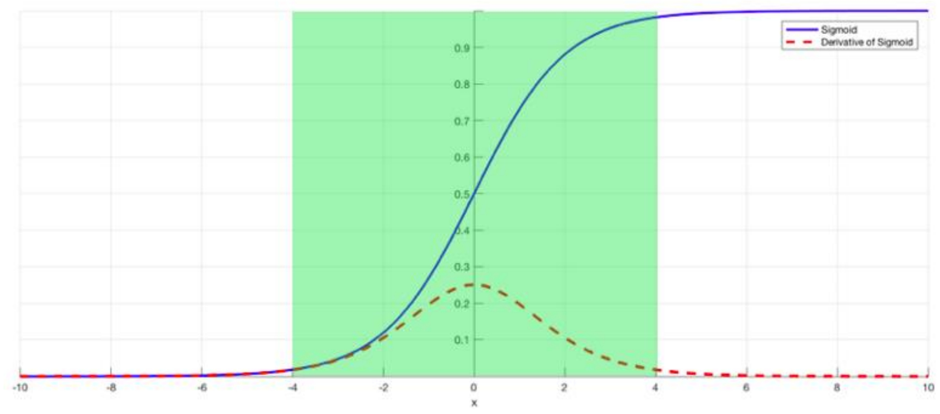
It has a “bigger” and constant derivative





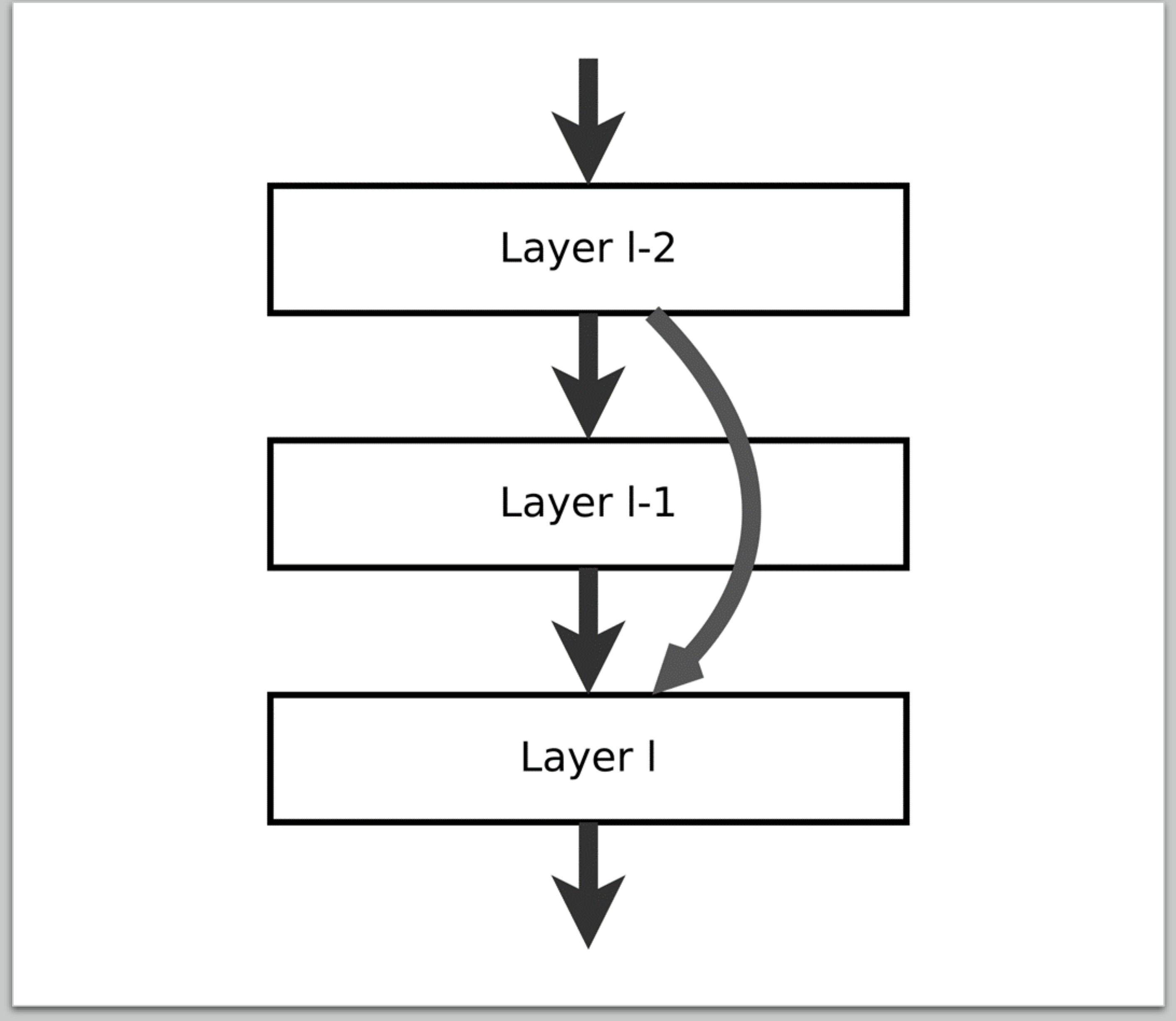
Batch normalization (in short)

- Batch normalization normalizes the input and ensures that |x| lies within the "*good range*" (the green region) and doesn't reach the outer edges of the sigmoid function.
- **If the input is in the "good" range, then the activation does not saturate, and thus the derivative also stays in the good range**, i.e- the derivative value isn't too small.
- Thus, batch normalization prevents the gradients from becoming too small and makes sure that the gradient signal is heard.
- (details later in this course)



Residual networks (ResNet) in short

- **Residual networks**: they provide connections straight to earlier layers (skip connections).
- link



Deep Neural Network Advantages

- **High-performance computing** (such as GPU, graphics processing units): we can train models with many layers, nodes and connections (often millions of parameters) thanks to high parallel computing like Hadoop, NoSQL, Spark, etc.
- **Vanishing Gradient**: This is greatly simplified by the use of **ReLU function** (rectifier linear unit, see later) and other activation functions with "good" properties, or other mechanisms such as batch norm. and ResNets.
- **Versatile method**: Deep Neural Network approaches can be used for supervised, semi-supervised, unsupervised problems and for reinforcement learning.

Many Deep Learning models

1. Feedforward "discriminative" models (semi-supervised/supervised training, used especially for image *processing):*

- ***CNN Convolutional Neural Networks***

2. Unsupervised training ("generative" models to re-build the input, select essential features, etc: used for a large set of applications and input data)

- ***Stacked Denoising Auto-Encoders***
- Generative Adversarial Networks (GAN)

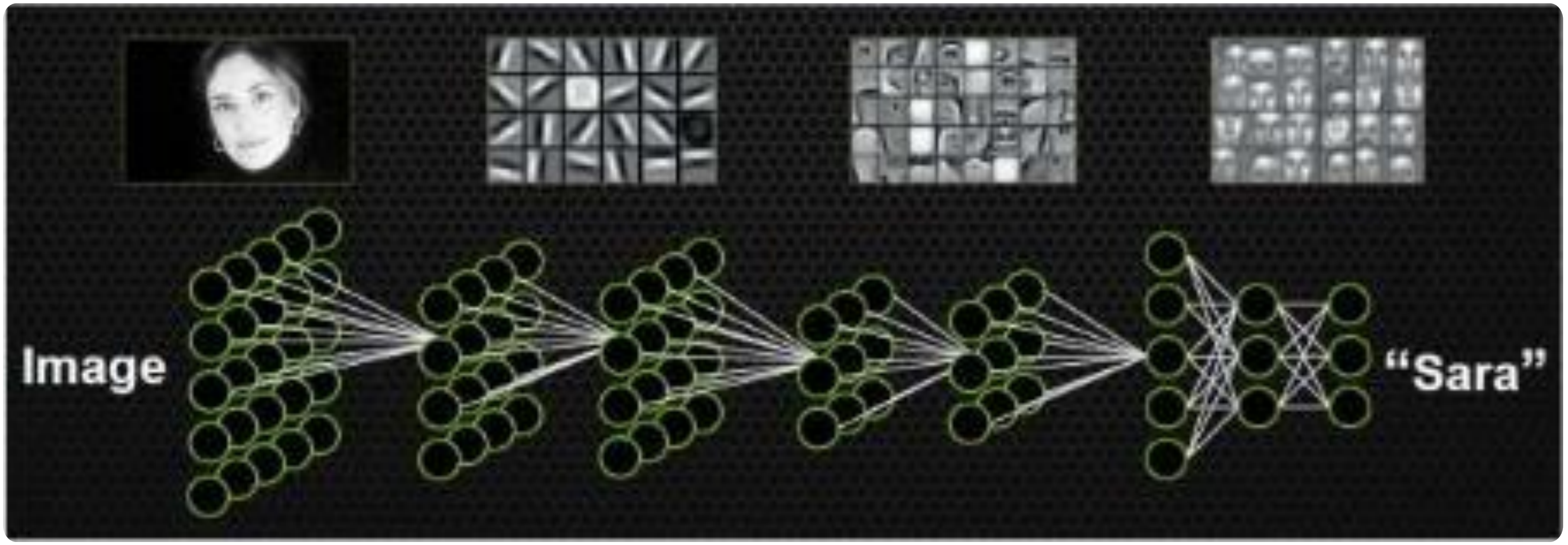
3. Recurrent models (used for sequential data, such as speech, natural language, patients trajectories..)

- ***RNN Recurrent neural Networks***
- ***LSTM Long Short-Term memory***

4. Reinforcement learning (to learn behaviors and strategies)

- ***Deep Q-Learning***

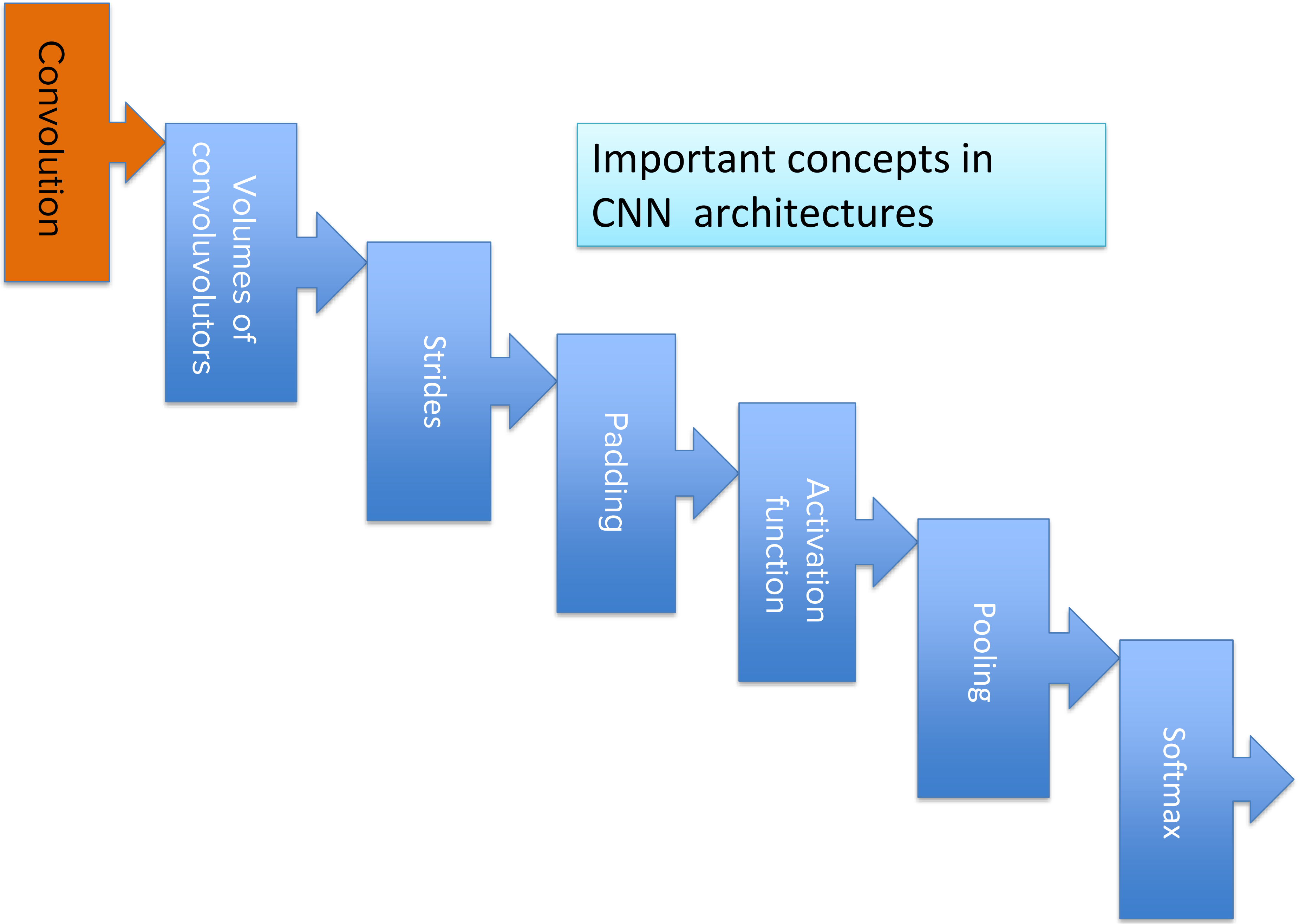
CONVOLUTIONAL NEURAL NETWORKS



CNN

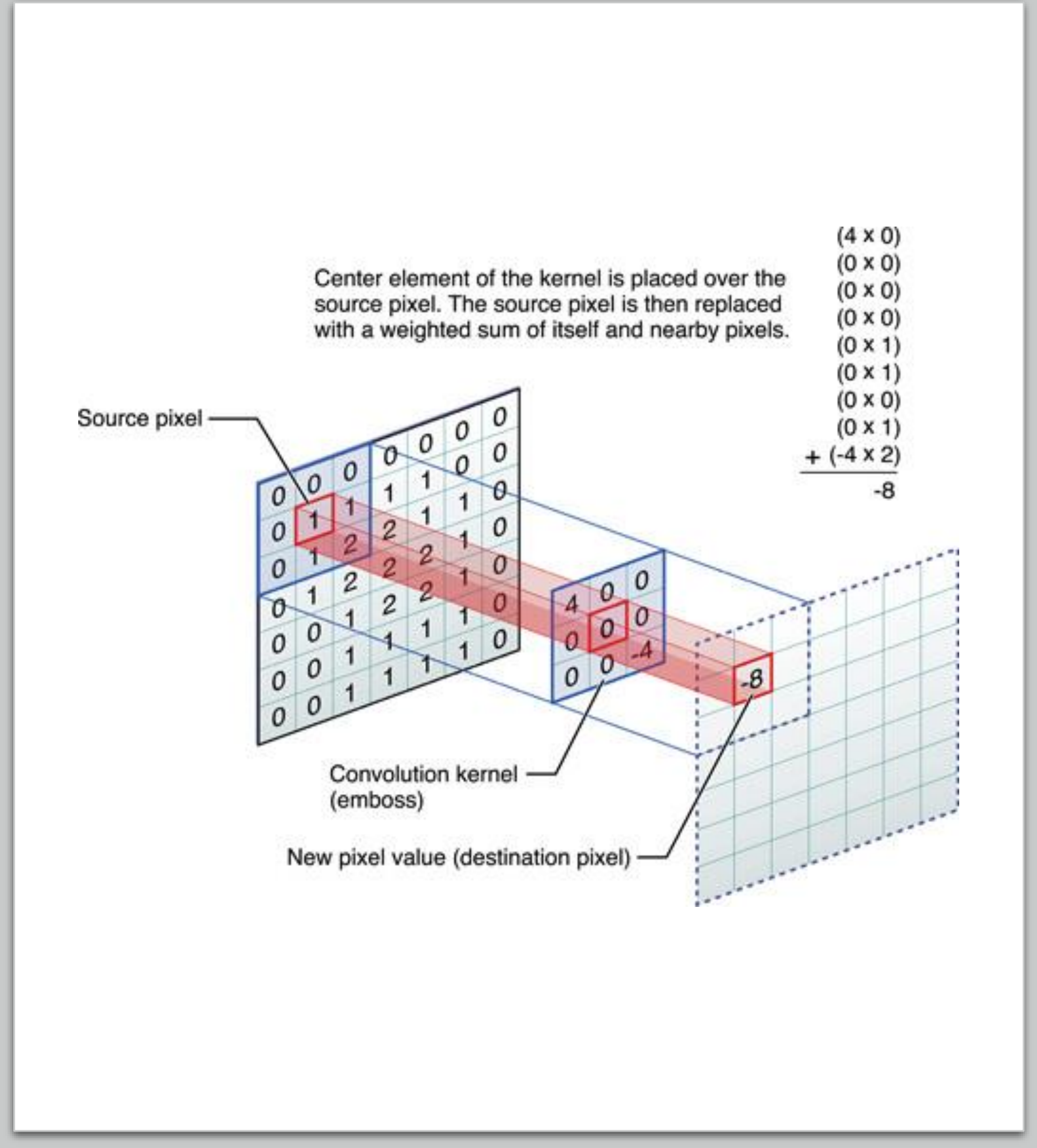
- Explicitly designed for image processing (although extensible to data that can be represented in matrix/tensor form)
- **Architecture**: a hierarchy of "loosely" connected layers, except for the last layers which are usually fully connected
- The first layer is directly connected to "pixels"

Important concepts in CNN architectures



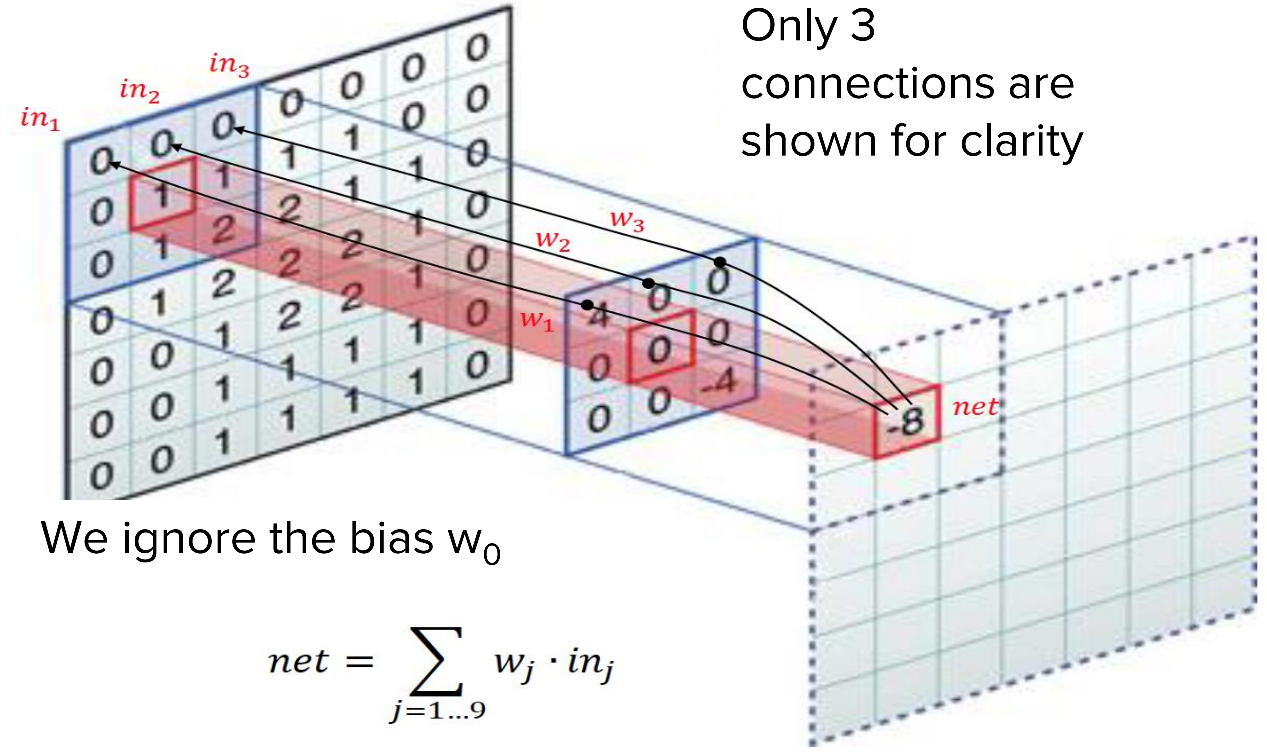
Convolution: what is ?

- A digital filter (a small 2D weight mask, also called **kernel**) sliding **over** the different input positions;
- For each position, an output value is generated by replacing the source pixel with a weighted sum of itself and it nearby pixels (convolution).
- **This is the equivalent of the net_j computation in MPN neurons**, but only applies to a subset of signals (depending upon the dimension of the filter)
- So here each neuron performs a convolution on a **subset of input** signals
- Note: input in CNN is a matrix (or a multi-layer **volume**) rather than a vector


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Neurons like convolutors

- Given an *n*x*n* filter (also called kernel), if we connect the n^2 inputs that it "covers" at the previous level, and we use weights as connection weights, we notice that a convolution computes a weighted sum of input (**net**) like a “classic” neuron.
- However, neurons here are less connected, each only takes a subset of inputs from previous layer



Only 3 connections are shown for clarity

We ignore the bias w_0

$$net = \sum_{j=1\ldots9} w_j \cdot in_j$$

1	0	1
0	1	0
1	0	1

kernel

2-D convolution (3x3 kernel)

input

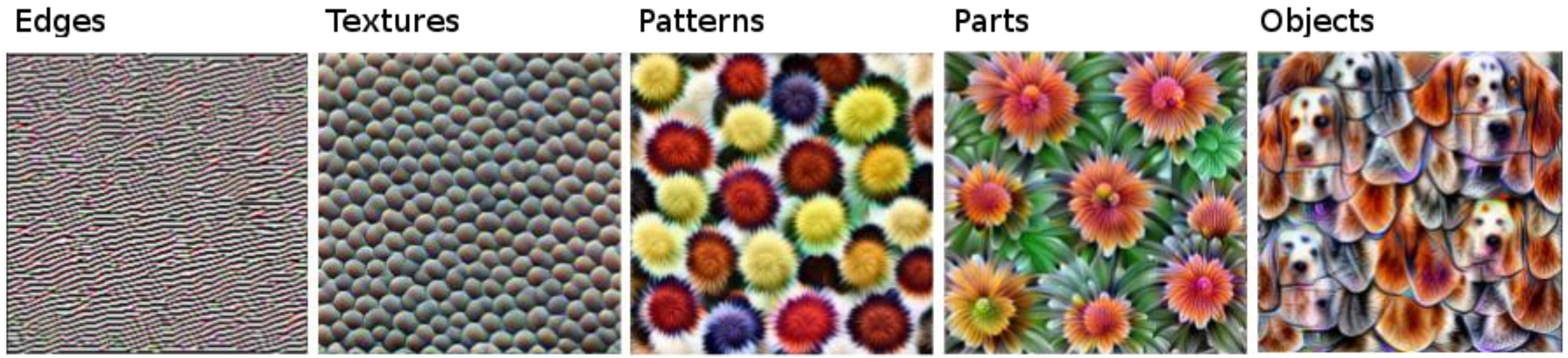
1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

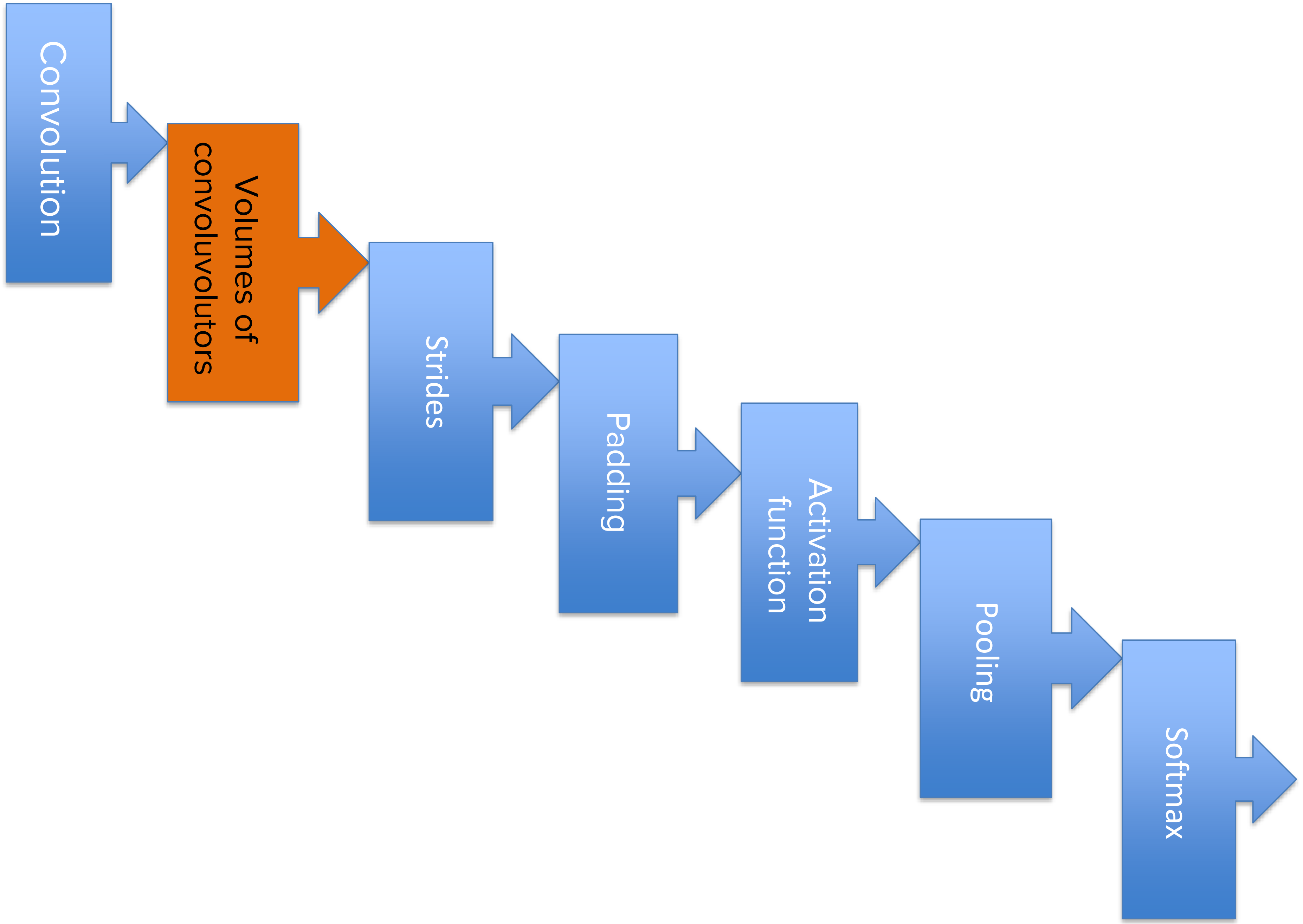
Output («feature map»)

4		

Further note that the output is a COMPRESSED representation of input, called **feature map** (it refers to the fact that convolutions learn automatically **latent higer-level features** of the image)

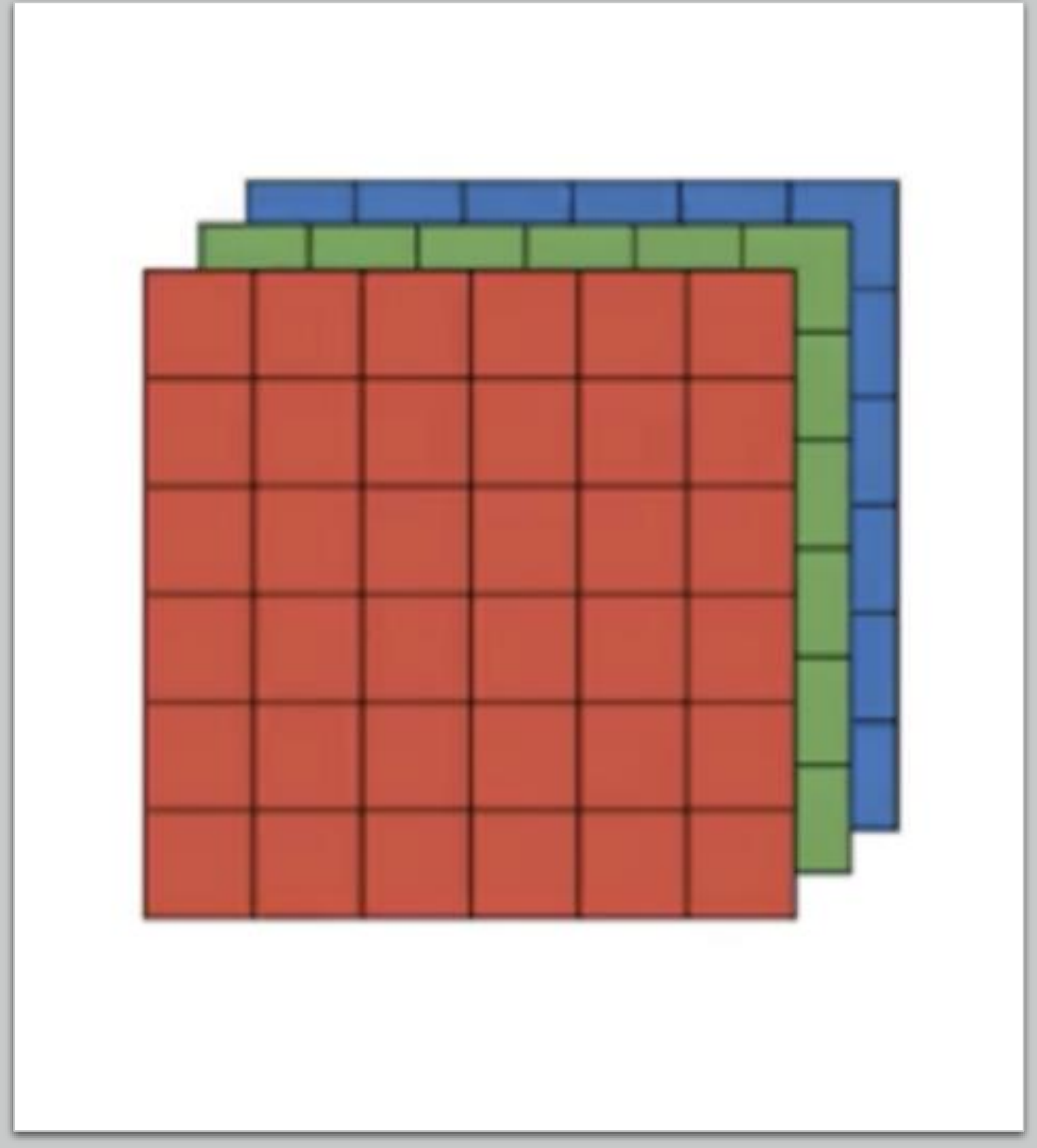
Convolutional layers, once trained, detect progressively more complex features of an image



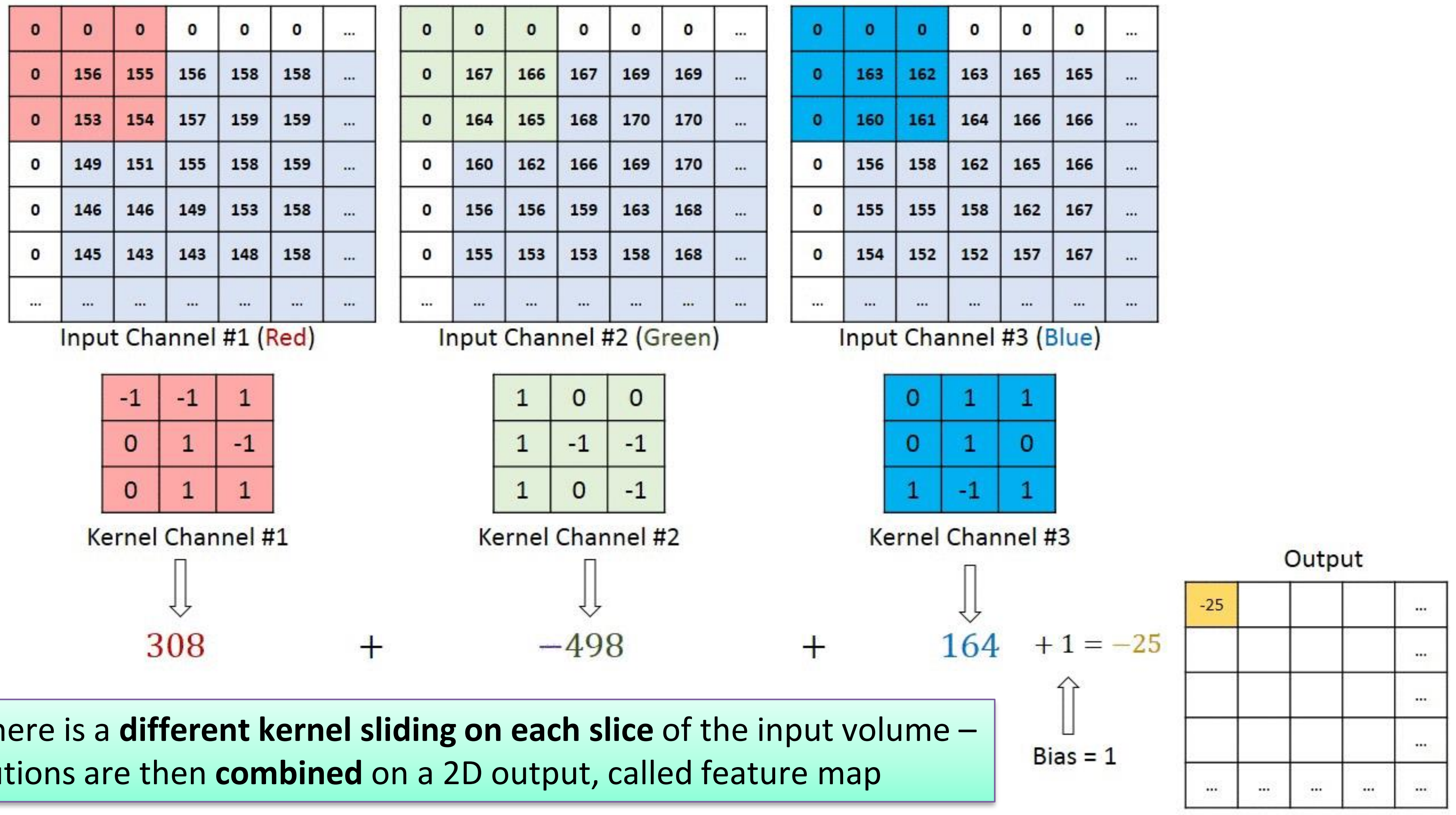

Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

3D Volumes

- Input data to CNN are 3-dimensional: *height, width, depth. The depth is, e.g.,* the number of color channels. For example, an RGB image would have a depth of 3, and
  the greyscale image would have a depth of 1.
- They can also be different representations of an image, like rotations, so the “meaning” of the third dimension really depends on applications
- A fourth dimension is the *batch size*, see later.
- In summary, *input and output data of a convolutional layers have 3D (or 4D with batch) dimensions, called VOLUMES*.
- Filters (kernels) on each slice of a volume can be the same but in general are different.



Example:
Convolution on 3-D input volumes with 3 different filters



0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

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

Kernel Channel #1

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

Kernel Channel #2

0	1	1
0	1	0
1	-1	1

Kernel Channel #3

$$308 + -498 + 164 + 1 = -25$$

Bias = 1

Output

-25				...
				...
				...
				...
...	...	...	...	...

Here, there is a **different kernel sliding on each slice** of the input volume – convolutions are then **combined** on a 2D output, called feature map

Dimension of input/output volumes

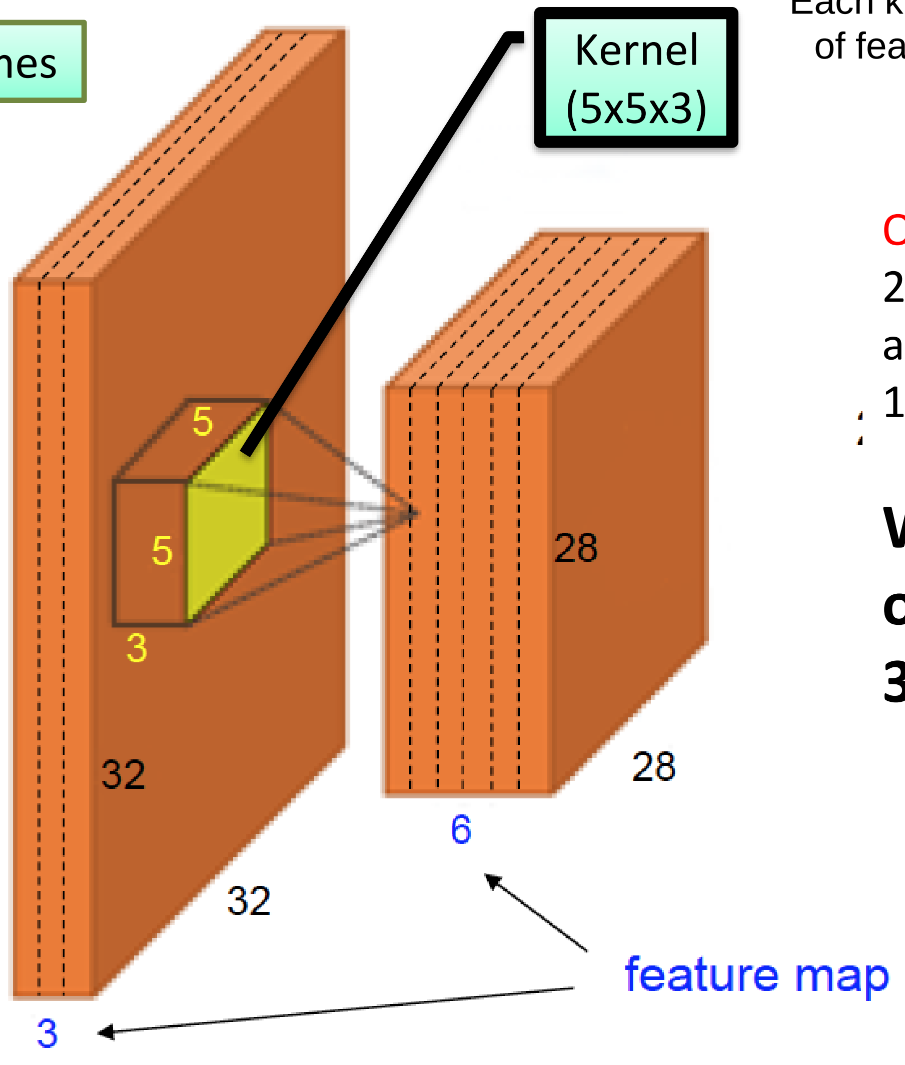
- Input has a volume (*HxWxD*)
- Each kernel/filter k_j has a volume $H_j x W_j x D_j$ spanning either on a single slice (D_j=1) of the input volume, or over the entire input volume *D*
- And you can have multiple kernels *n* over the same 3D volume
- The dimension of the output volume depends on the input volume,on the dimension and number of filters, and on how they slide over each slice of a volume (see later for exact computation)

Convolutions on 3D volumes compress the data

Example of **two-layers 3D-volumes**

Each kernel is applied on the input 3D volume of feature maps and creates a compressed feature map

Input volume :
32x32x3 neurons



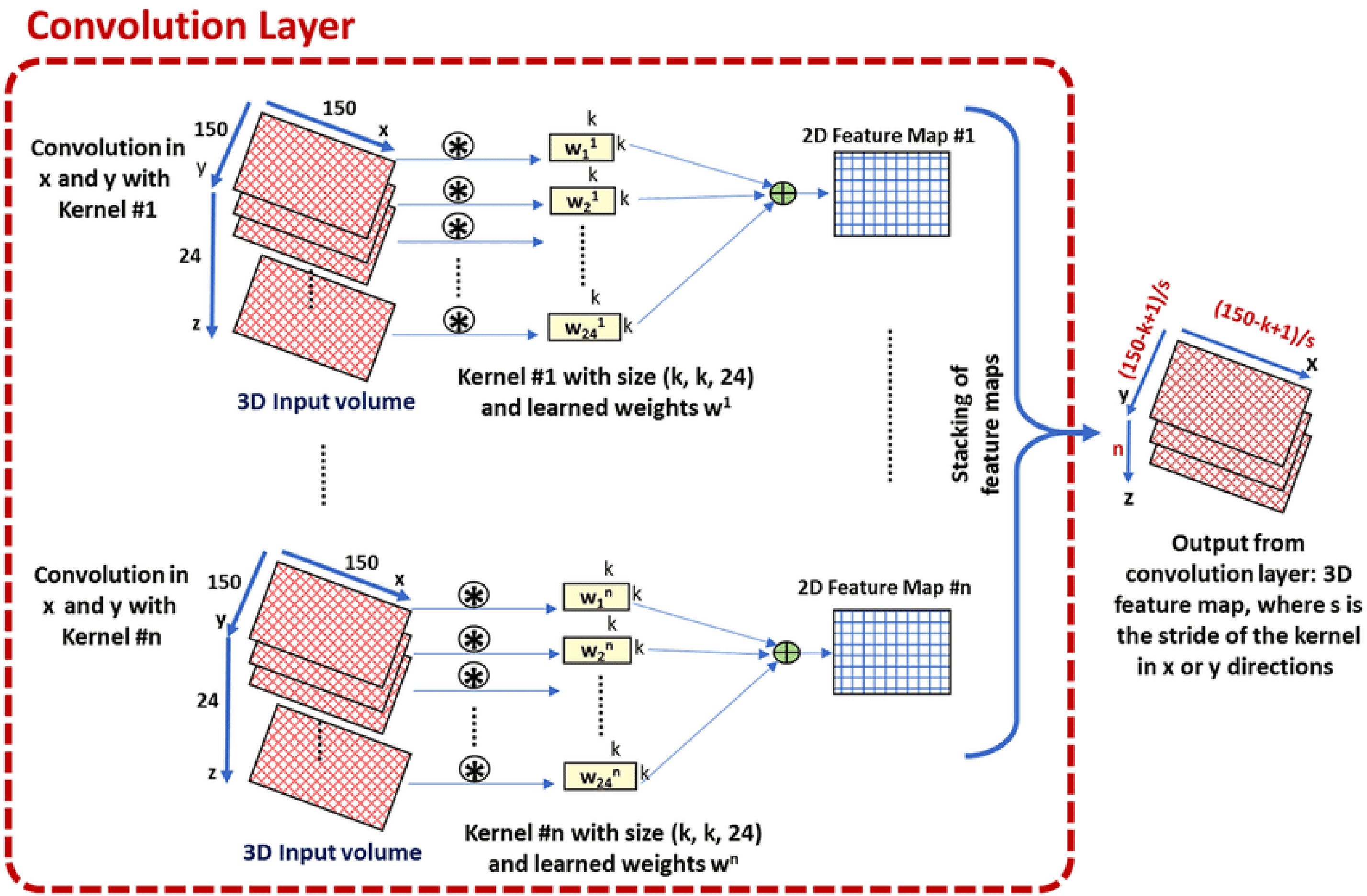
Output volume i:
28x28x6 neurons (obtained by applying **6** 5x5x3 kernels to the i-1 volume) => D_{out}=6

$W_{out}=W_{in}-F+1$ (F is size of kernel)

32-5+1=28

V_{in}=**150x150x24** (W_{in} H_{in} D_{in})
$Kernel_j$=*k*x*k*x24 but the 24 output are summed on a single slice (feature map)
And we have *n* kernels, so D_{out}=n
Concerning W_{out} and H_{out}, see later how to compute

Here, each kernel slides on all slices, output is a compressed 3D volume

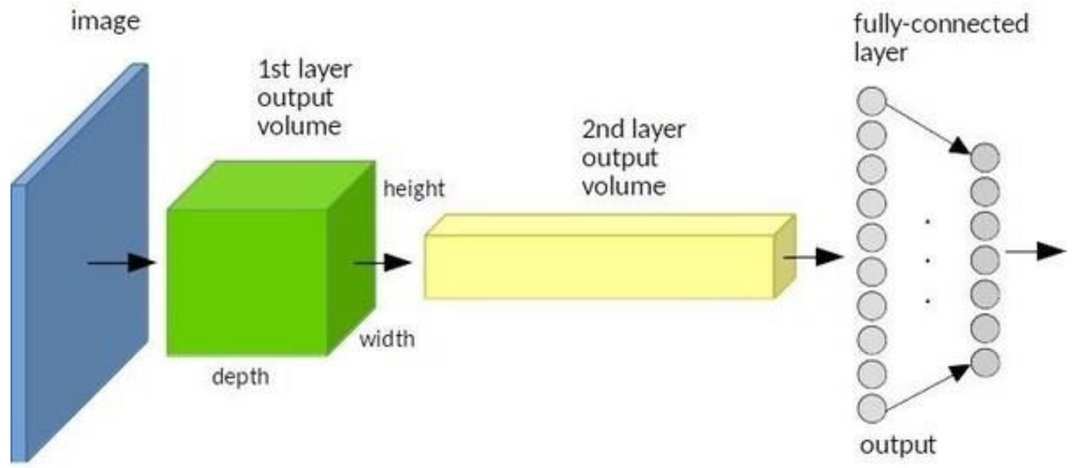


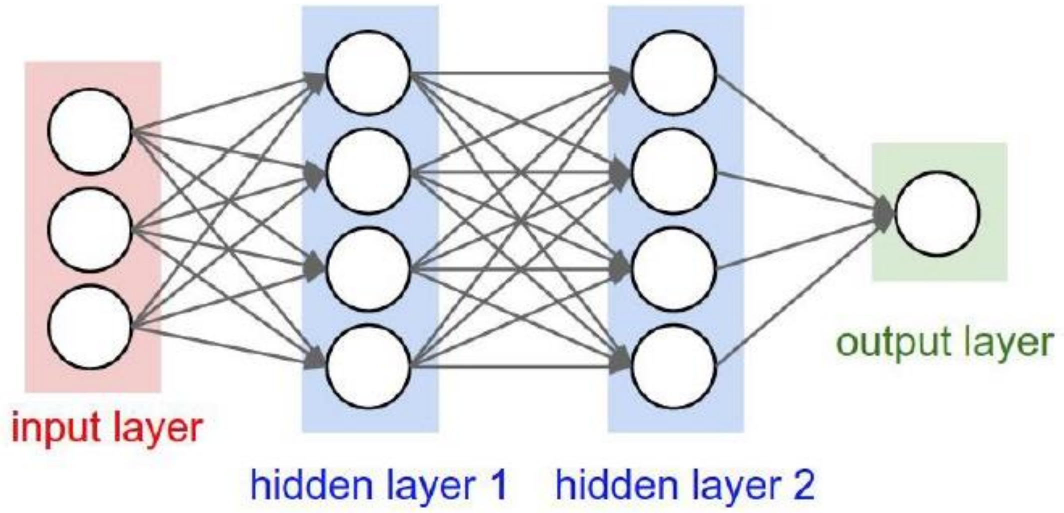
Convolution Layer
Convolution in x and y with Kernel #1
150
150
x
y
24
z
3D Input volume
k
w_1^1
w_2^1
w_{24}^1
Kernel #1 with size (k, k, 24) and learned weights w^1
2D Feature Map #1
Stacking of feature maps
Convolution in x and y with Kernel #n
3D Input volume
w_1^n
w_2^n
w_{24}^n
Kernel #n with size (k, k, 24) and learned weights w^n
2D Feature Map #n
(150-k+1)/s
(150-k+1)/s
n
Output from convolution layer: 3D feature map, where s is the stride of the kernel in x or y directions

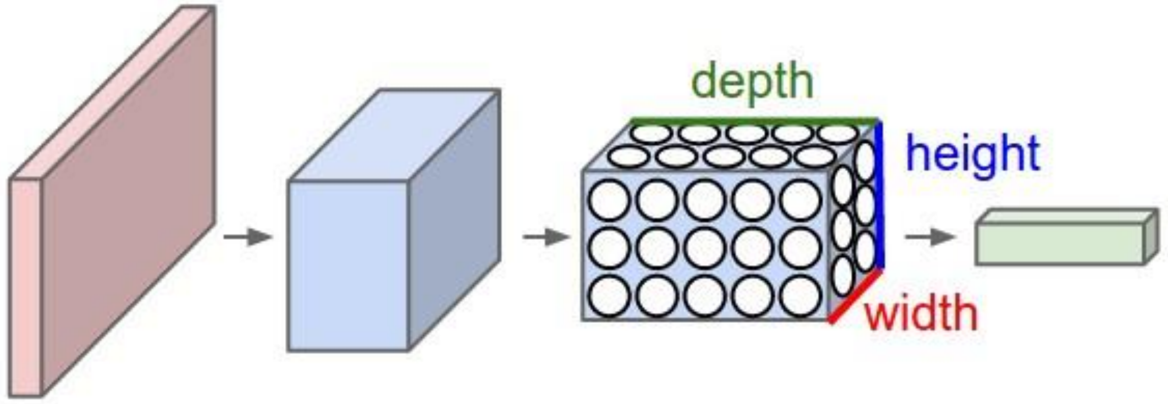
Illustration of the processing of a single volume using CNN. To clarify how a single convolution layer of the CNN processes a 3D input volume of size 150 × 150 × 24, each 3D volume has 24 images (depth is 24) and a 2D convolution is applied to each image using a kernel of size K × K. The output of an individual kernel is then summed to create a 2D feature map (150-k+1 X 150-k+1). Usually, multiple kernels are used, and the above step is applied again for those kernels, and different feature maps are produced for respective kernels. The final result is a volume of feature maps. (https://www.researchgate.net/figure/Illustration-of-the-processing-of-a-single-volume-using-CNN-To-clarify-how-a-single_fig1_332349176)

The network is now a sequence of “3D-Volumes” layers

- Each layer of the NN receive and output a 3D volume (except the last)
- **Width** and **height** surfaces preserve the spatial organization of the original input (**although progressively compressed in highr-level features**), the depth identifies the “salient features” captured from the image by each kernel, arranged in **slices** (every slice is the result of the application of a kernel). Note that # of slices may vary in any layer







MLP vs 3D volumes

Ok but..

Q1: What exactly are these kernels/filters?

Q2: How do we compute the kernel/filters weight?

Q3: How do we select the size of a kernel/filter?

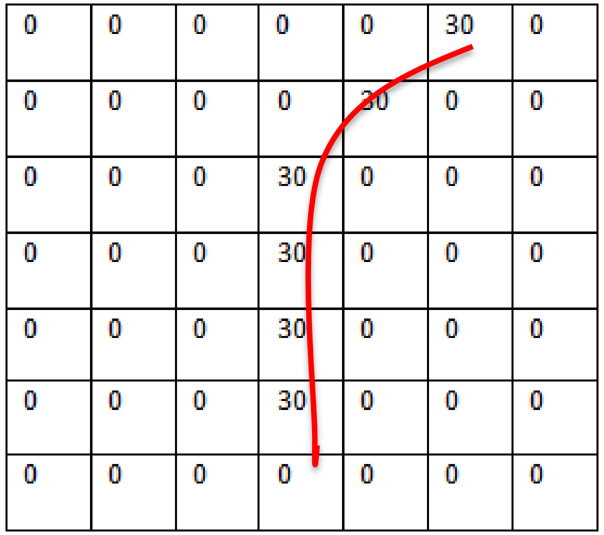
Q1: Kernels and Features

- Each filter/kernel can be thought of as a **feature (pattern, trait) identifier**.
- Captured **features** in the first layer can be (for images) straight ***edges***, simple ***colors***, and ***curves***. The simplest characteristics that all images have in common with each other.
- In the subsequent layers, patterns can be more complex (e.g., eyes, nose, for a face classifier). Note however that there is no explicit semantics here.
- We can therefore see kernels as a "**set of learnable weights to capture relevant features in data**". (parameters)
- For this reason, the slices of output volumes after each convolutional+activation layer are called "**feature maps**"

Example:

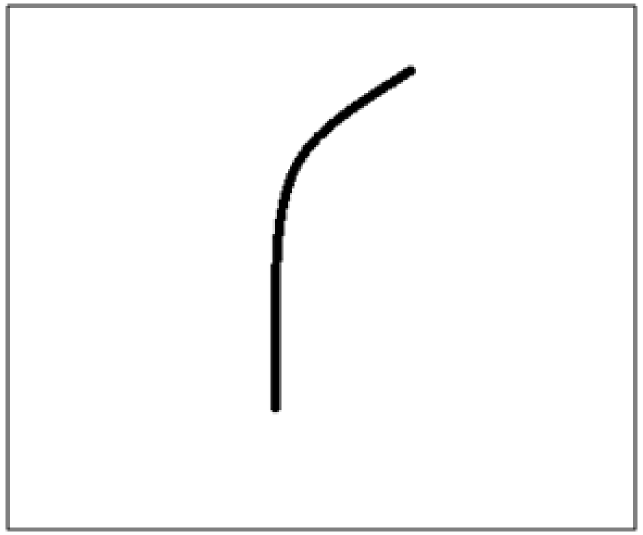
line detector

kernel

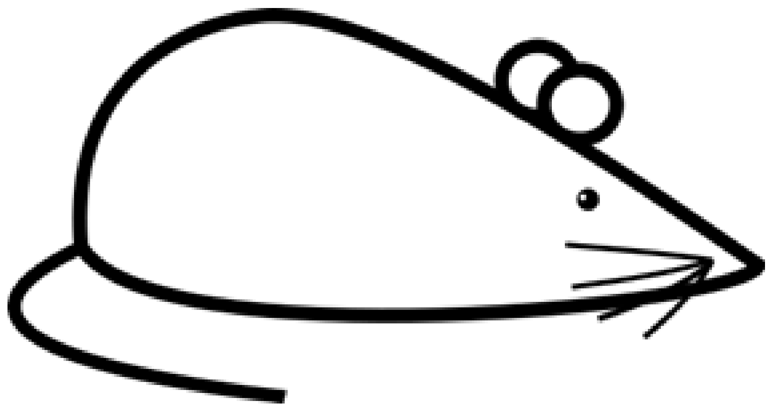


0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

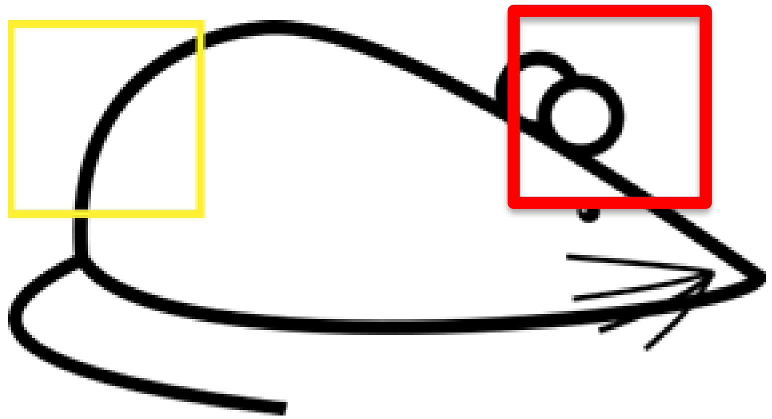
Pixel representation of filter



Visualization of a curve detector filter

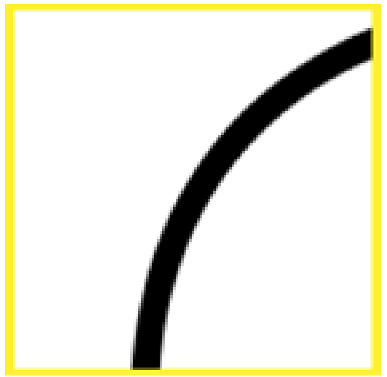


Original image



Visualization of the filter on the image

Application of a kernel to a pixel map



Visualization of an image detail

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

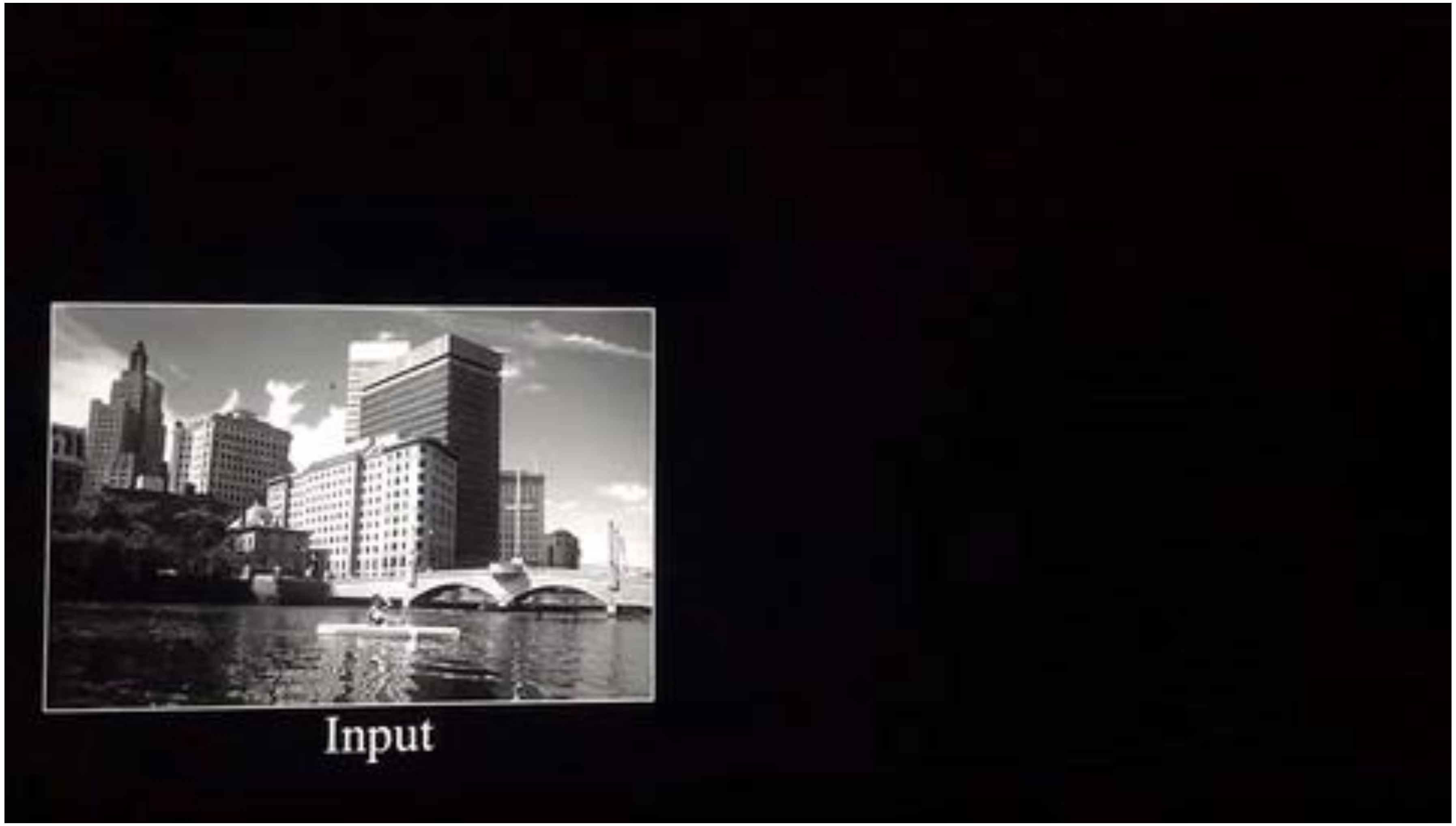
Pixel representation of filer

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

Examples from: link

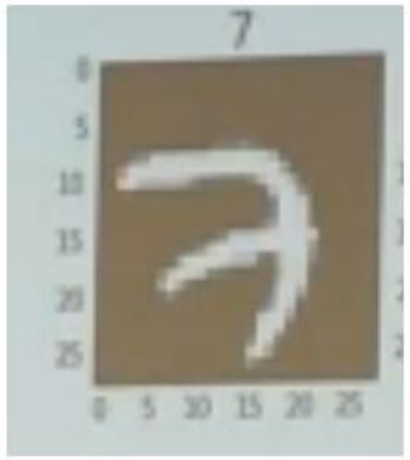
Another example

source link



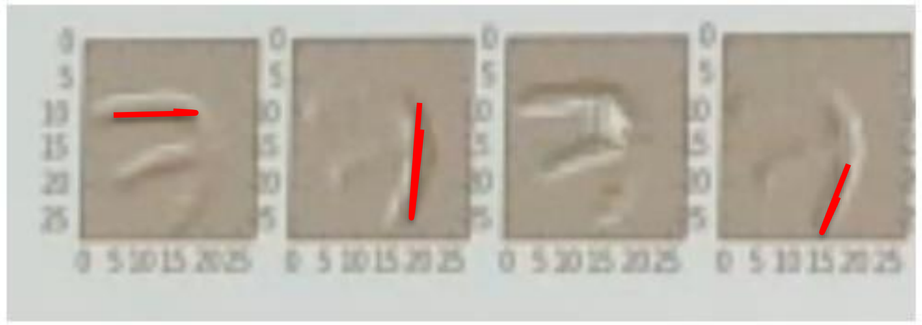
Example 2:

Effect of different «edge» filters/kernels



- The upper image is the initial one. The subsequent 3 images are the result of applying 3 filters/kernels (in 3 slices).
- Note that each kernel is capturing a different edge of the original image

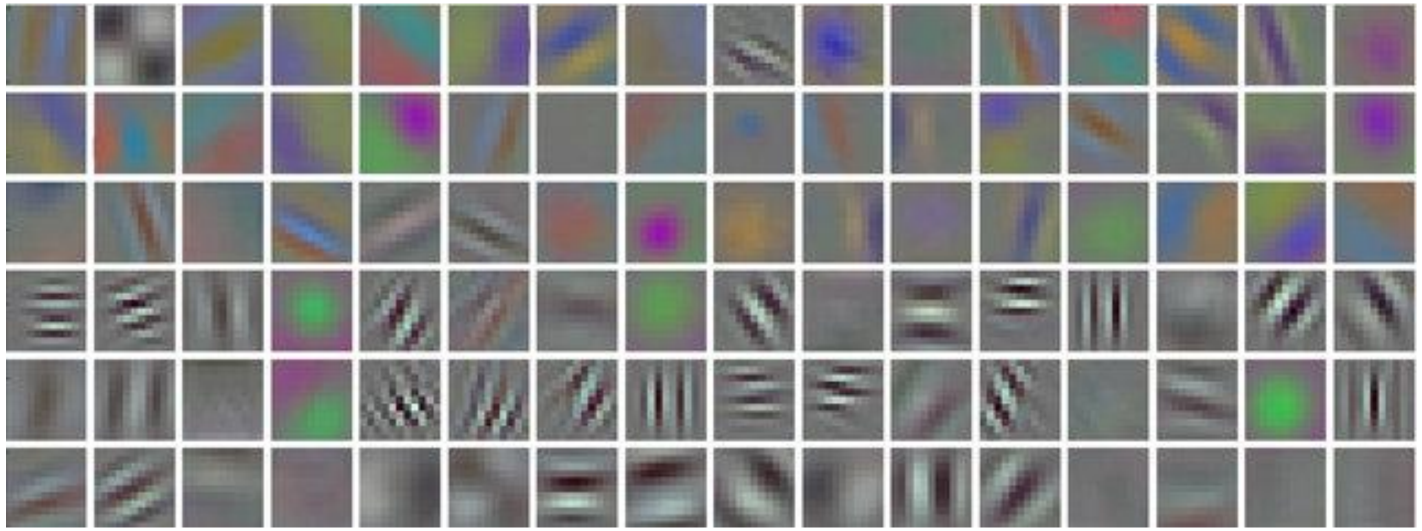
can be experienced by the network as:



- In this example: kernels are edge detectors, each slice results from applying a specific type of edge filter to the input image

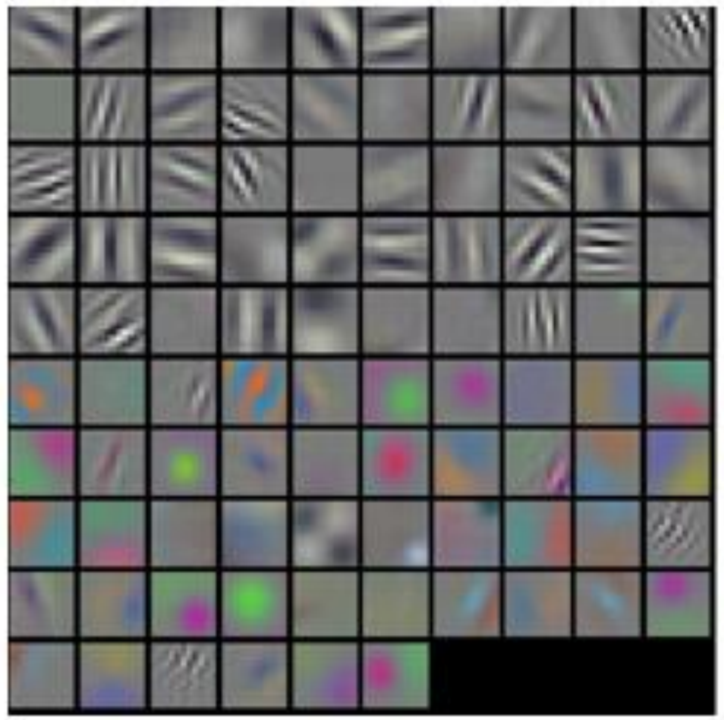
How many filters in each layer?

- For example, the following 96 filters are used in the first layer of a CNN to measure left-right mirror symmetry in images (see paper)



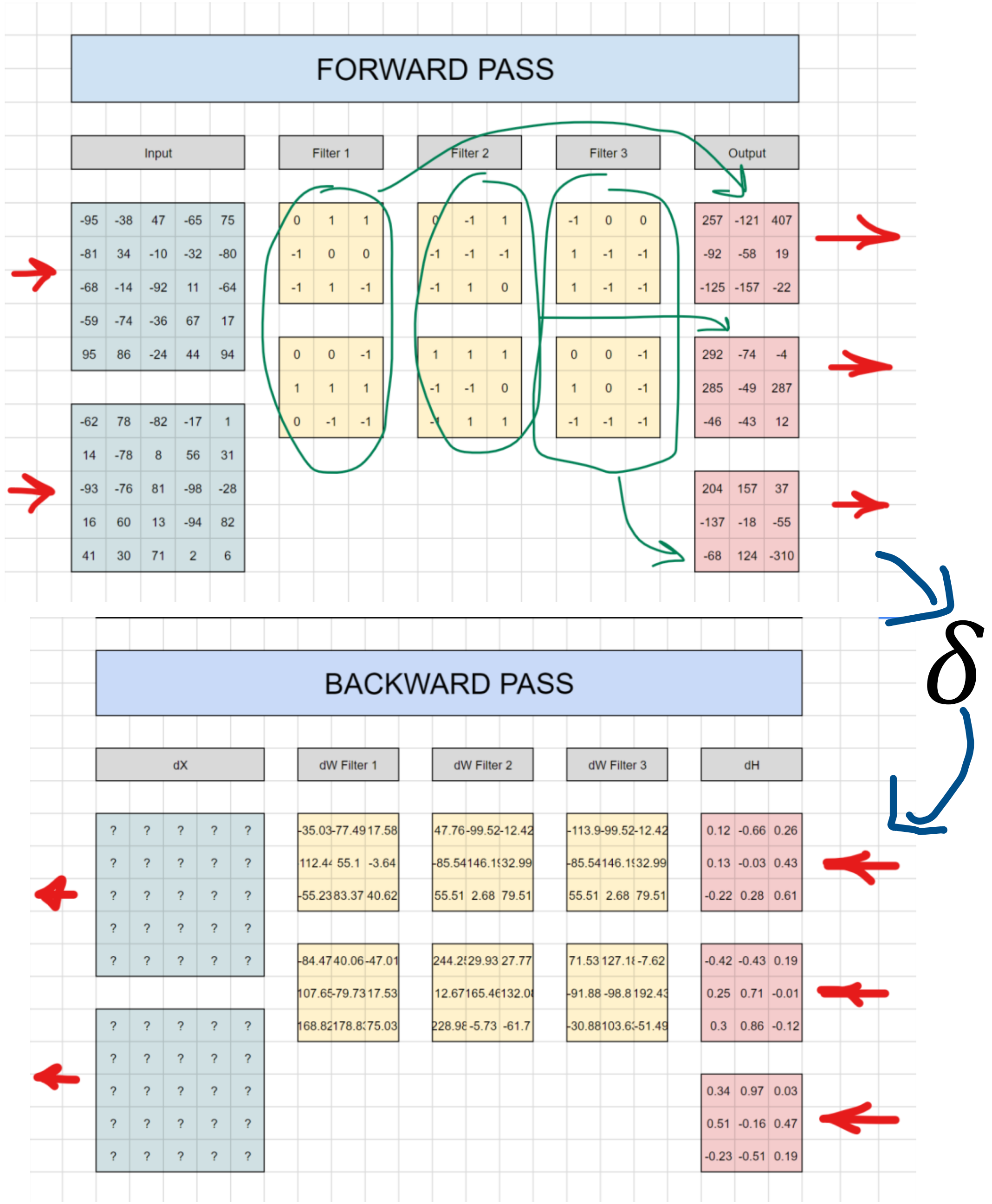
Remember: kernel weights are the PARAMETERS of a CNN

- During the training phase, kernels (=the weights of each kernel) must be learned using a process similar to backpropagation
- **Number and dimension of kernels are hyperparameters**
- During the **prediction phase**, kernels layers convolve around the input image and “activate” (like neurons) when the specific learned patterns they are looking for (a line, a texture) is found in the input.


Visualizations of filters

Q2: How to learn filter weights?

- This is easy: start with random selection of weights and use backpropagation with gradient descent, like for multi-layer perceptron
- **The only difference is with the neuron activation function** (**ReLu**, see later)
- In the **final fully connected layer** uses a different approach (**Softmax**, see later)
- Since neurons are only locally connected, a **lower number of weights** must be computed in each slice (but we have many layers and many filters).
- Note: especially in the first layer, we can also use pre-defined kernels (e.g., Gabor filters in the first layer)



Input

-95	-38	47	-65	75
-81	34	-10	-32	-80
-68	-14	-92	11	-64
-59	-74	-36	67	17
95	86	-24	44	94

-62	78	-82	-17	1
14	-78	8	56	31
-93	-76	81	-98	-28
16	60	13	-94	82
41	30	71	2	6

Filter 1			Filter 2			Filter 3		
0	1	1	0	-1	1	-1	0	0
-1	0	0	-1	-1	-1	1	-1	-1
-1	1	-1	-1	1	0	1	-1	-1
0	0	-1	1	1	1	0	0	-1
1	1	1	-1	-1	0	1	0	-1
0	-1	-1	-1	1	1	-1	-1	-1

Output

257	-121	407
-92	-58	19
-125	-157	-22
292	-74	-4
285	-49	287
-46	-43	12
204	157	37
-137	-18	-55
-68	124	-310

dX

?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?

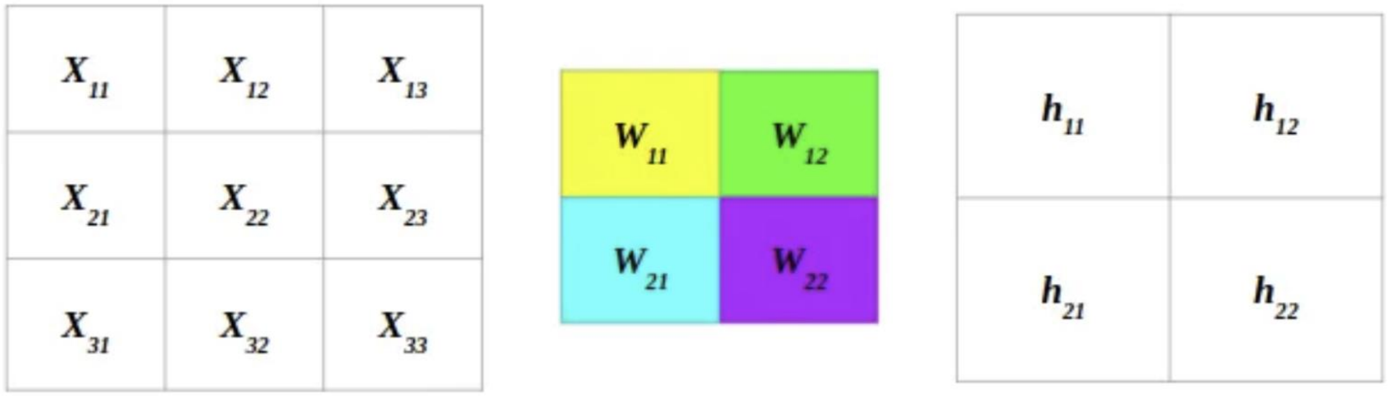
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?

dW Filter 1			dW Filter 2			dW Filter 3		
-35.03	-77.49	17.58	47.76	-99.52	-12.42	-113.9	-99.52	-12.42
112.4	55.1	-3.64	-85.54	146.1	32.99	-85.54	146.1	32.99
-55.23	83.37	40.62	55.51	2.68	79.51	55.51	2.68	79.51
-84.47	40.06	-47.01	244.2	29.93	27.77	71.53	127.1	-7.62
107.6	-79.73	17.53	12.67	165.4	132.0	-91.88	-98.8	192.4
168.8	178.8	75.03	228.9	-5.73	-61.7	-30.88	103.6	-51.49

dH

0.12	-0.66	0.26
0.13	-0.03	0.43
-0.22	0.28	0.61
-0.42	-0.43	0.19
0.25	0.71	-0.01
0.3	0.86	-0.12
0.34	0.97	0.03
0.51	-0.16	0.47
-0.23	-0.51	0.19

Let’s consider the backpropagation step only on filters: this is the forward step to cpmpute convolutions



Input Size : 3×3, **Filter Size** : 2×2, **Output Size** : 2×2

$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

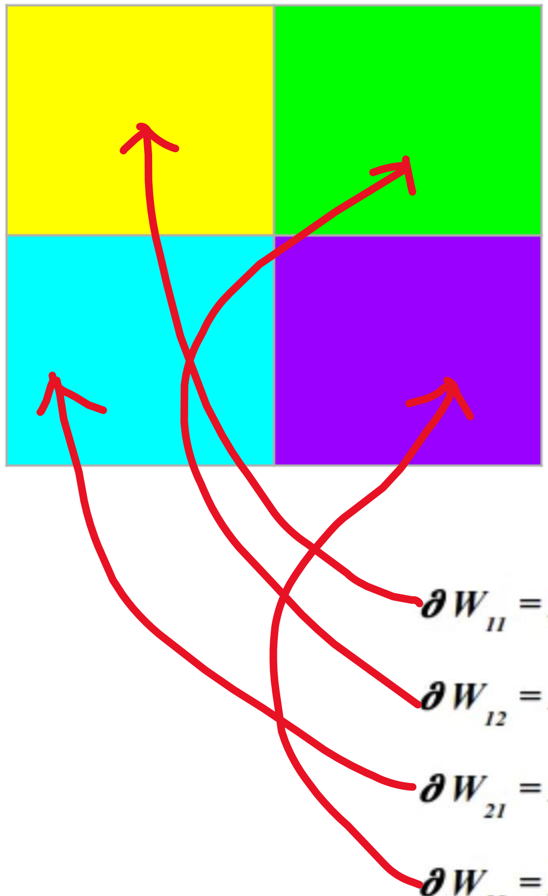
$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

Backword step

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}



∂h_{11}	∂h_{12}
∂h_{21}	∂h_{22}

$$\partial W_{11} = X_{11}\partial h_{11} + X_{12}\partial h_{12} + X_{21}\partial h_{21} + X_{22}\partial h_{22}$$

$$\partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22}$$

$$\partial W_{21} = X_{21}\partial h_{11} + X_{22}\partial h_{12} + X_{31}\partial h_{21} + X_{32}\partial h_{22}$$

$$\partial W_{22} = X_{22}\partial h_{11} + X_{23}\partial h_{12} + X_{32}\partial h_{21} + X_{33}\partial h_{22}$$

∂h_{ij} represents $\dfrac{\partial L}{\partial h_{ij}}$

∂w_{ij} represents $\dfrac{\partial L}{\partial w_{ij}}$

Credits: https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199

Q3: How do we establish the size of kernels?

Size of the kernels is an **hyperparameter** and plays an important role in finding the key patterns in data:

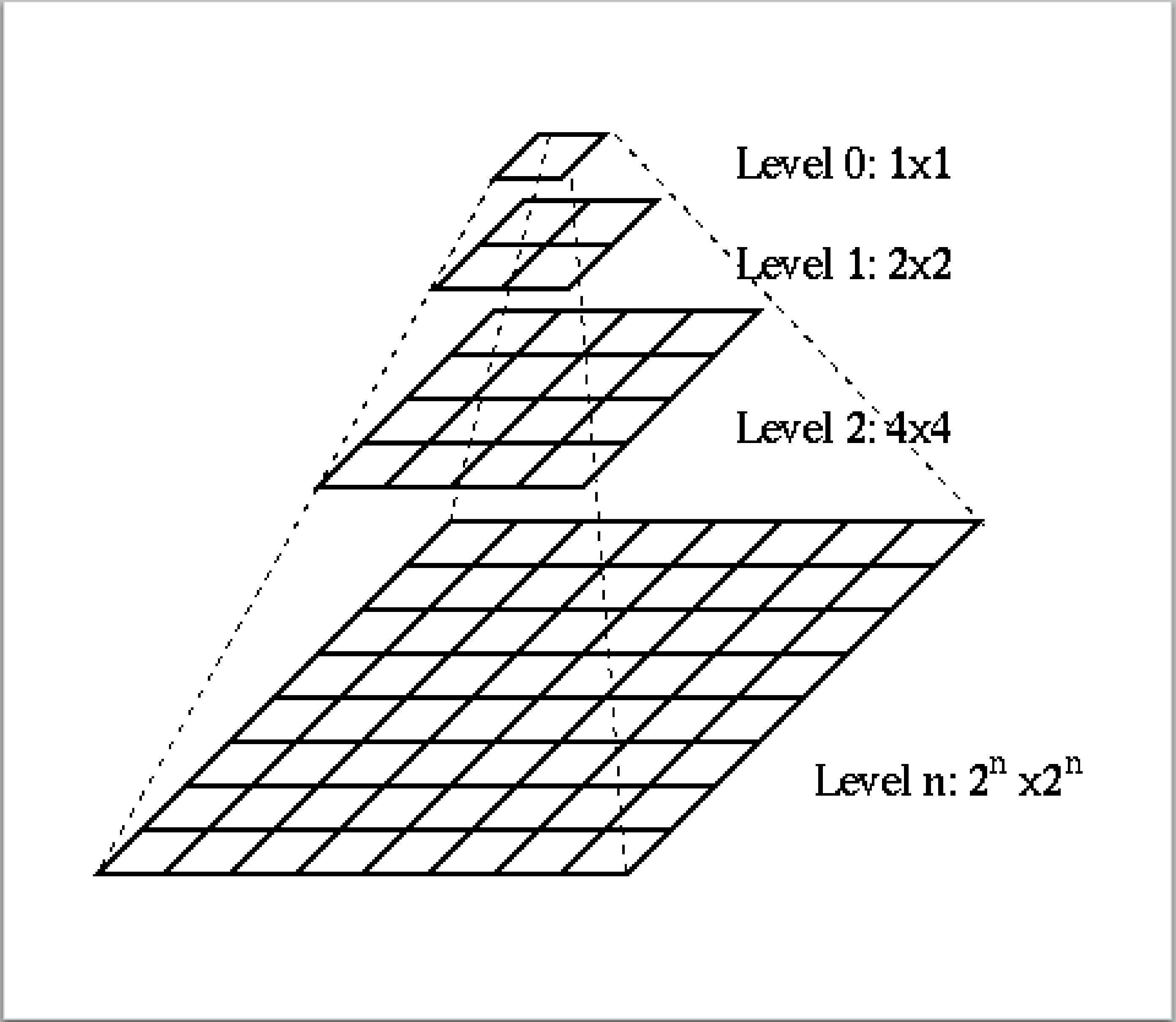
- ➢ A larger size kernel can overlook at the features and could skip the essential details in the input
- ➢ A smaller size kernel could provide more information leading to more confusion. Thus there is a need to determine **the most suitable size of the kernel** (given the type of images, and the purpose of computation, e.g., face recognition, symmetry detection, semantic labeling of images ....)

How do we establish the size of kernels?

- **Local kernels**: In one extreme case where we have 1x1 kernels, we are essentially saying low-level features are **per-pixel**, and they don't affect neighboring pixels at all, and that we should apply the same operation to all pixels.
- **Global kernel**: In the other extreme, we have kernels with the size of the entire image. In this case, the CNN essentially becomes **fully connected** and stops being a CNN, and we are no longer making any assumption **on low-level feature locality**.

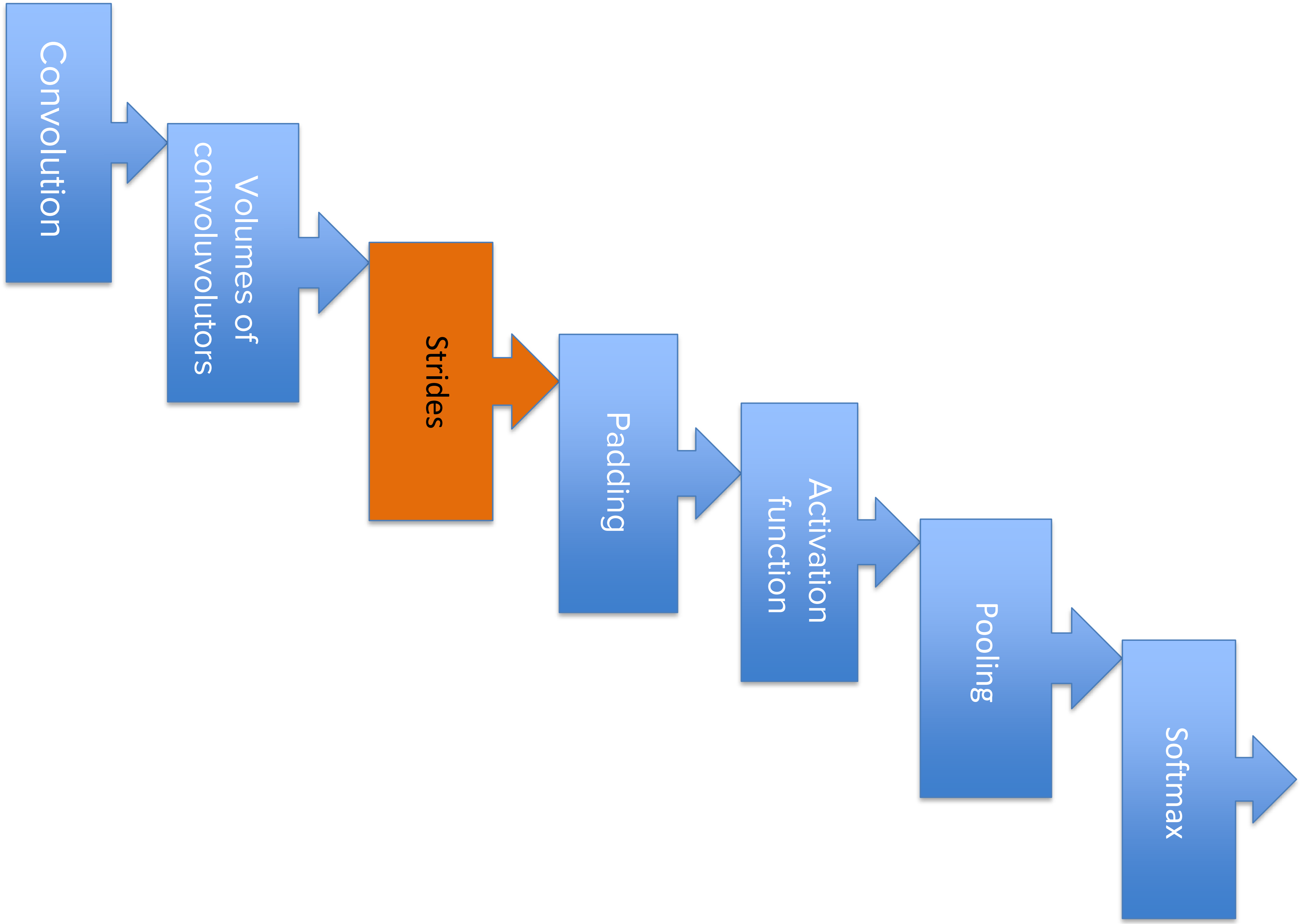
Gaussian Pyramid

- Approaches like **gaussian pyramids** (set of different sized kernels) are generally used to test the efficiency of the feature extraction and appropriate size of the filter is determined.



Summary so far

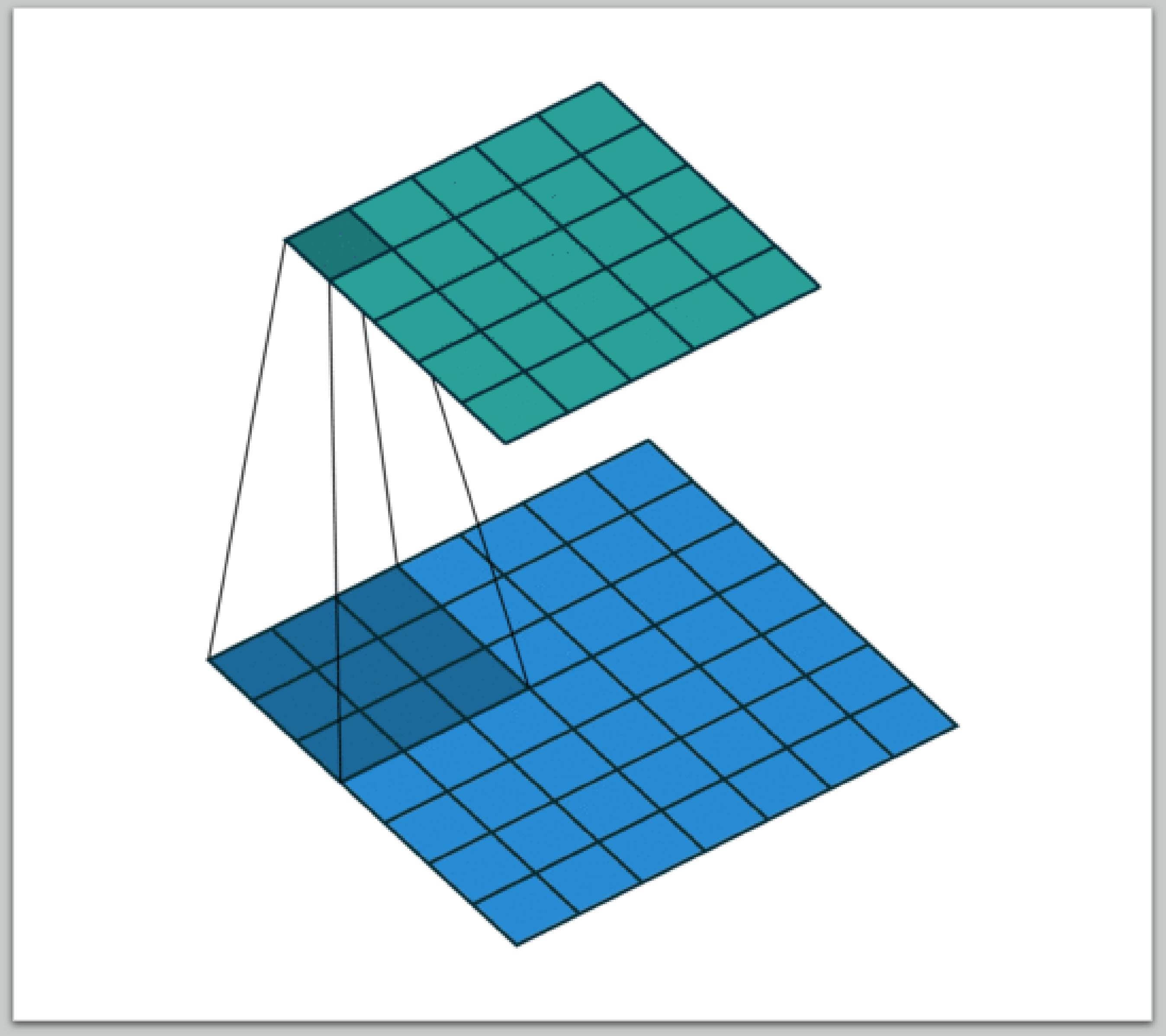
- CNNs are multi-layered loosely connected neural networks
- At each layer, a (2)-3D convolution operation is applied on the current layer using a (2) 3D kernels
- Key elements of the convolution are the kernel **values** and **size**
- **Values** (connection weights) are the CNN parameters, learned using **backpropagation**
- **Size** is a hyperparameter to be tuned
- Kernel values are learned through backpropagation


Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

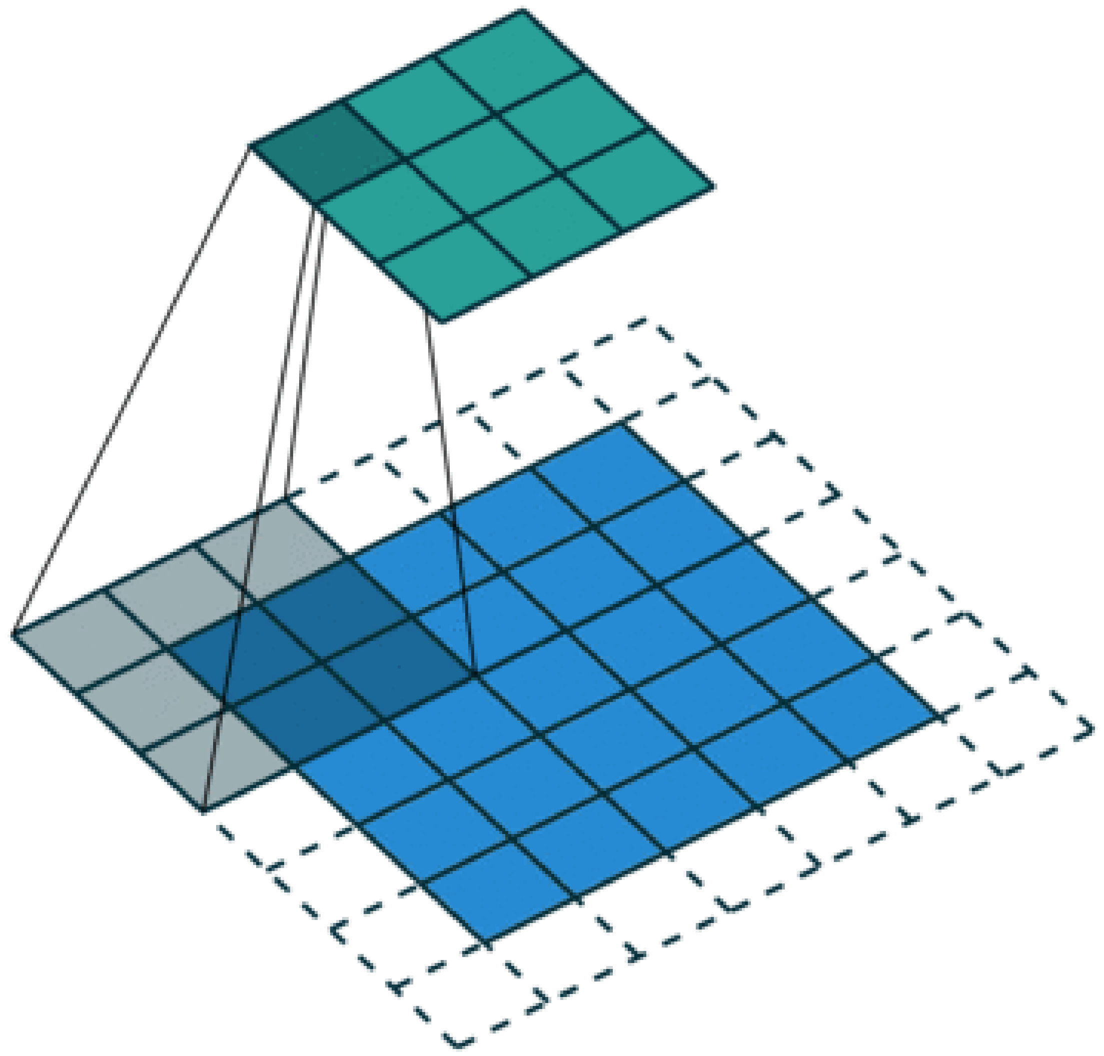
Strides:

- kernels slide on input with a 1-unit step (moves 1 neuron to the left). We can use **higher increment**, or **stride**. This further reduces the number of connections to be learned.
- A stride of 2 or 4 may imply a high-efficiency gain on first layers
- The increment of strides **s** are another *hyperparameter*

Example of Stride=1

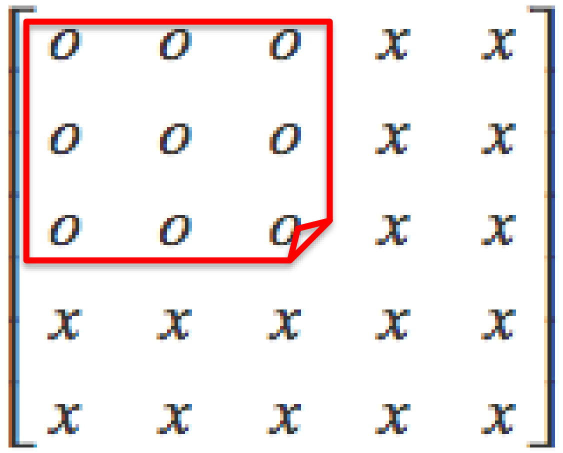


Example of Stride=2



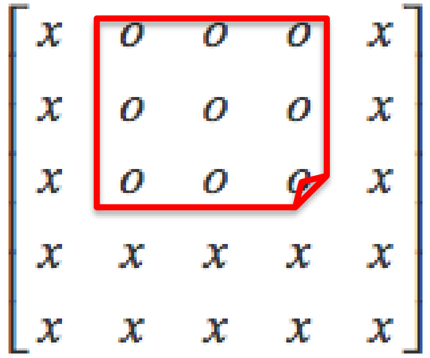
Why strides?

- **They reduce the size** (width, length) of the next layer, and lets you decide how much overlap you want between two output values in a layer.
- Let’s say you have a 5x5 input to a layer, and the kernel is 3x3. This is the first thing it sees:

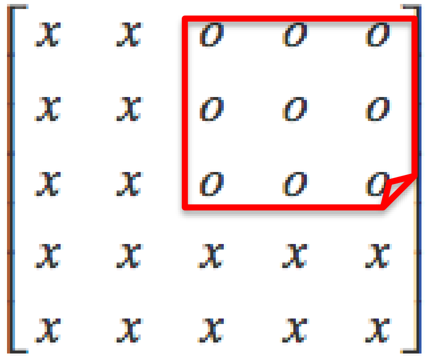


Why strides?

- With a stride of 1, this will be the second thing:



- With a stride of two, instead:

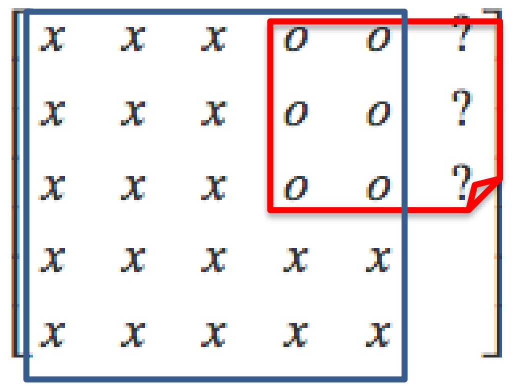


Formula to compute the dimension W' of the output (same applies to H)

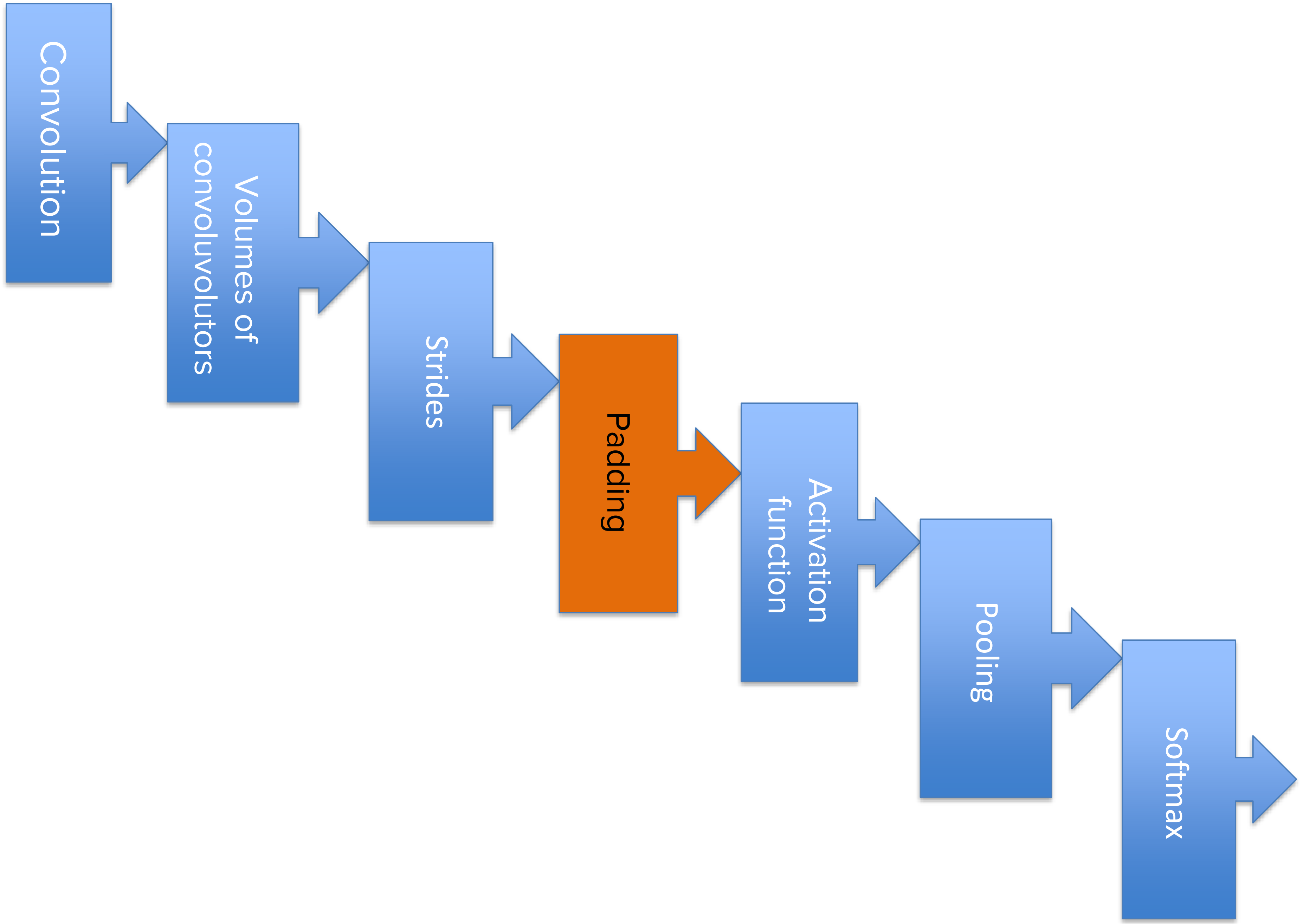
W'=(W-F)/S +1
=(5-3)/2 +1=2
F dim of filter, S dim of slide

Problem with strides

- With a stride of 3:

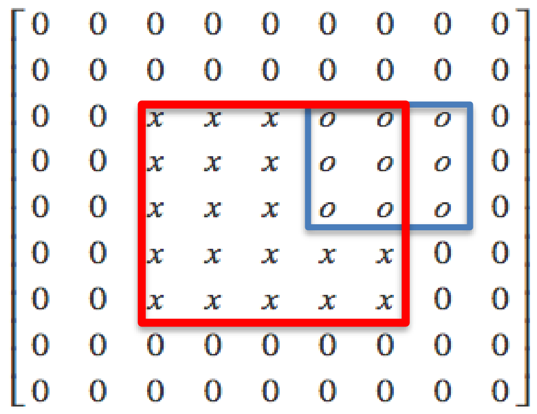


- We exceed the boundary of the input! So, how can we avoid this problem?


Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

Padding

- To solve the problem of «extremes edges» (kernels that either don’t reach an edge of the input or exceed its boundaries) we use 0-valued **borders** around the input volume. This is called **padding**
- Around the entire input, we add padding data with a width equal to the kernel/filter width minus one (or height equal to kernel height minus one) **P=F-1** (larger paddings can be used)
- The hyperparameter “padding” P defines the thickness of such borders.

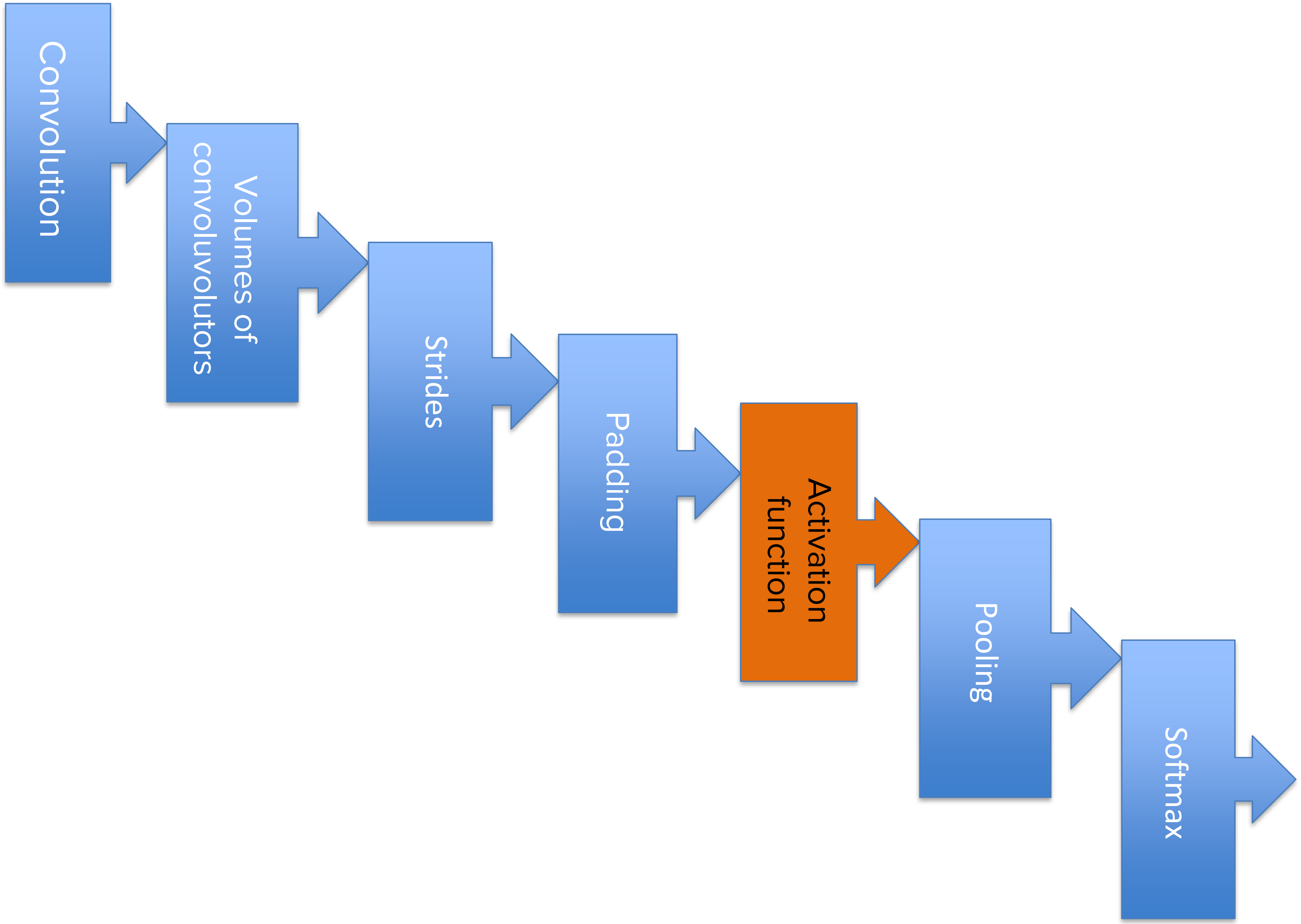


Computing output feature-map dimensions with strides and padding

- Let W_{out} be the (horizontal) size of the output of a layer *i* and W_{in} the corresponding input dimension. Also, let F be the horizontal size of the kernel/filter. The following relationship applies:

$$W_{out} = ((W_{in} - F + 2 \cdot Padding)/Stride) + 1$$

- A similar relationship links the vertical parameters.
- In previous example, padding=2, F=3, W_{in}=5; stride=3, W_{out} =((5-3+2x2)/3)+1)= 3
- Larger strides, larger compression
- The depth of the output volume instead depends on the depth of each kernel d and on the number of kernels k D_{out}= (kxd)

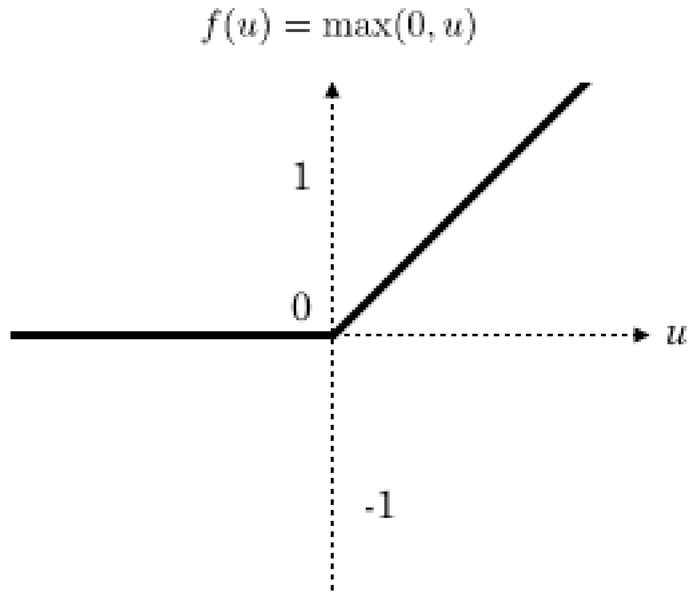

Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

Activation function

- So far we introduced the steps to compute a convolution of input data (the analogous of $net_j(\mathbf{x})$ function in standard neural networks). The convolutions are then fed into neurons n_j that compute an activation function, to finally produce an output o_j.
- In the Multilayer Perceptron (MLP) networks the (historically) most popular activation function is the **sigmoid function**.
- In **deep networks**, the use of sigmoid is problematic for the **vanishing gradient problem**, as we have seen.

ReLU

To solve this problem, the Rectified Linear (ReLU) activation function has been introduced

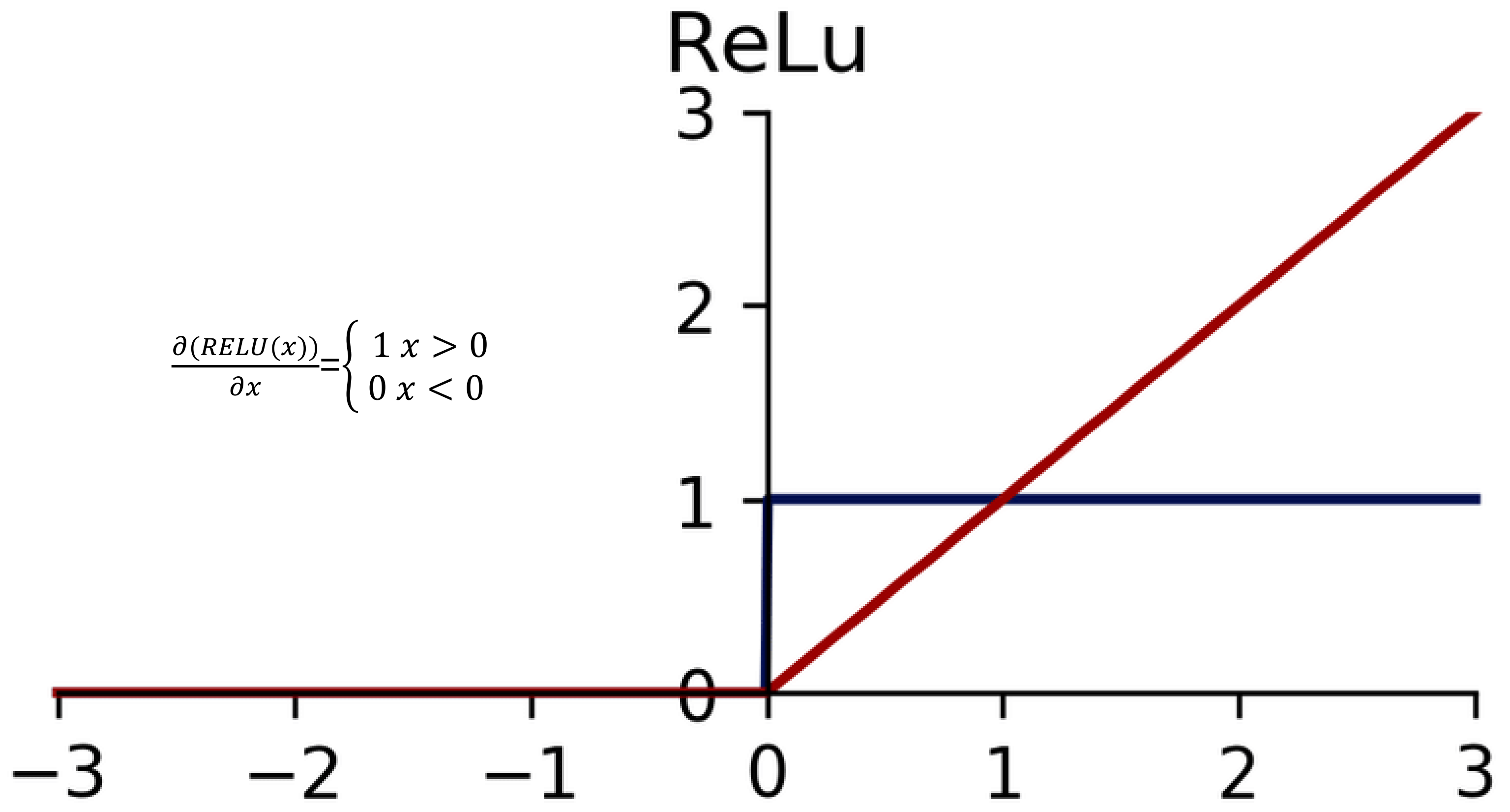


Note that 2 and 3 are also drawback, please read more on ReLU

1. The **derivative is 0** for negative or null values of the net(x)
2. No **saturation** for positive values: the range of activations is [0, ∞)
3. It causes sparse activation of neurons which are shown to add robustness (all the activations of negative input are zero). Note that ReLU is **NOT linear:** it is linear in [-∞ 0)[0 + ∞) but not in the full interval
4. Issue: output of layers is no longer normalized in [0 1] (➔ need batch norm. for inner layes, not just input)

Relu derivative

$$\frac{\partial(RELU(x))}{\partial x} = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$



Types of Activation Functions

(many are used in literature)

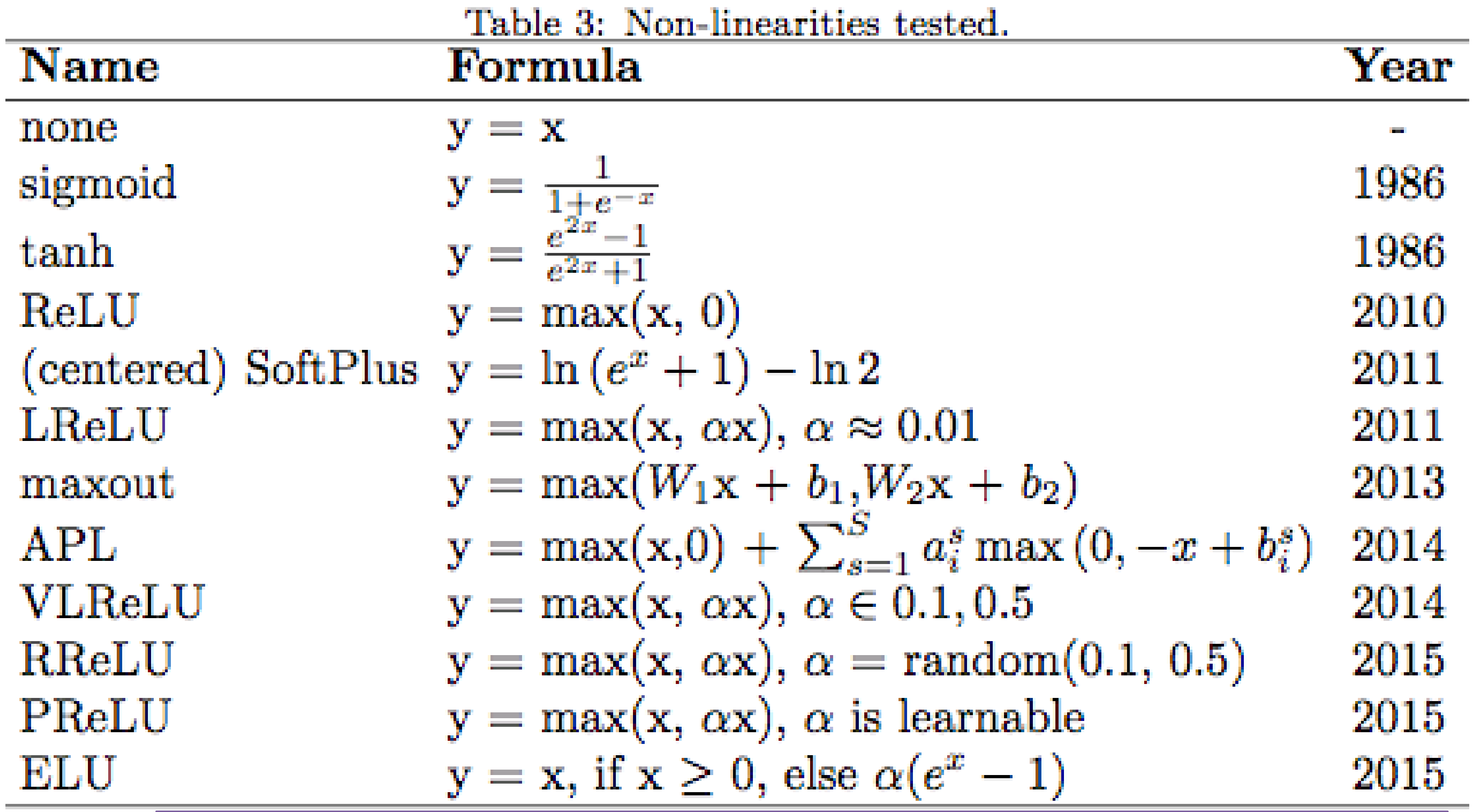
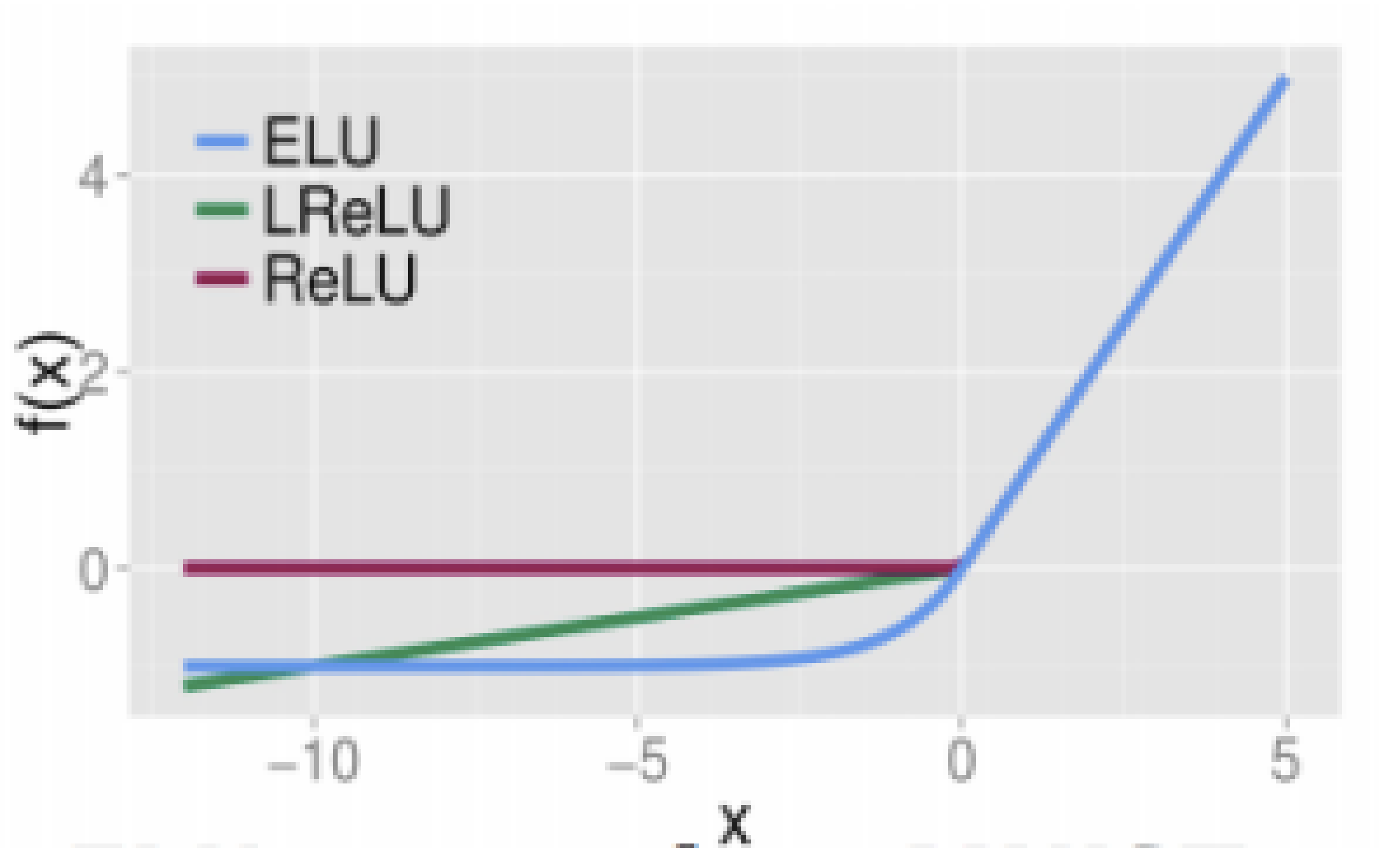


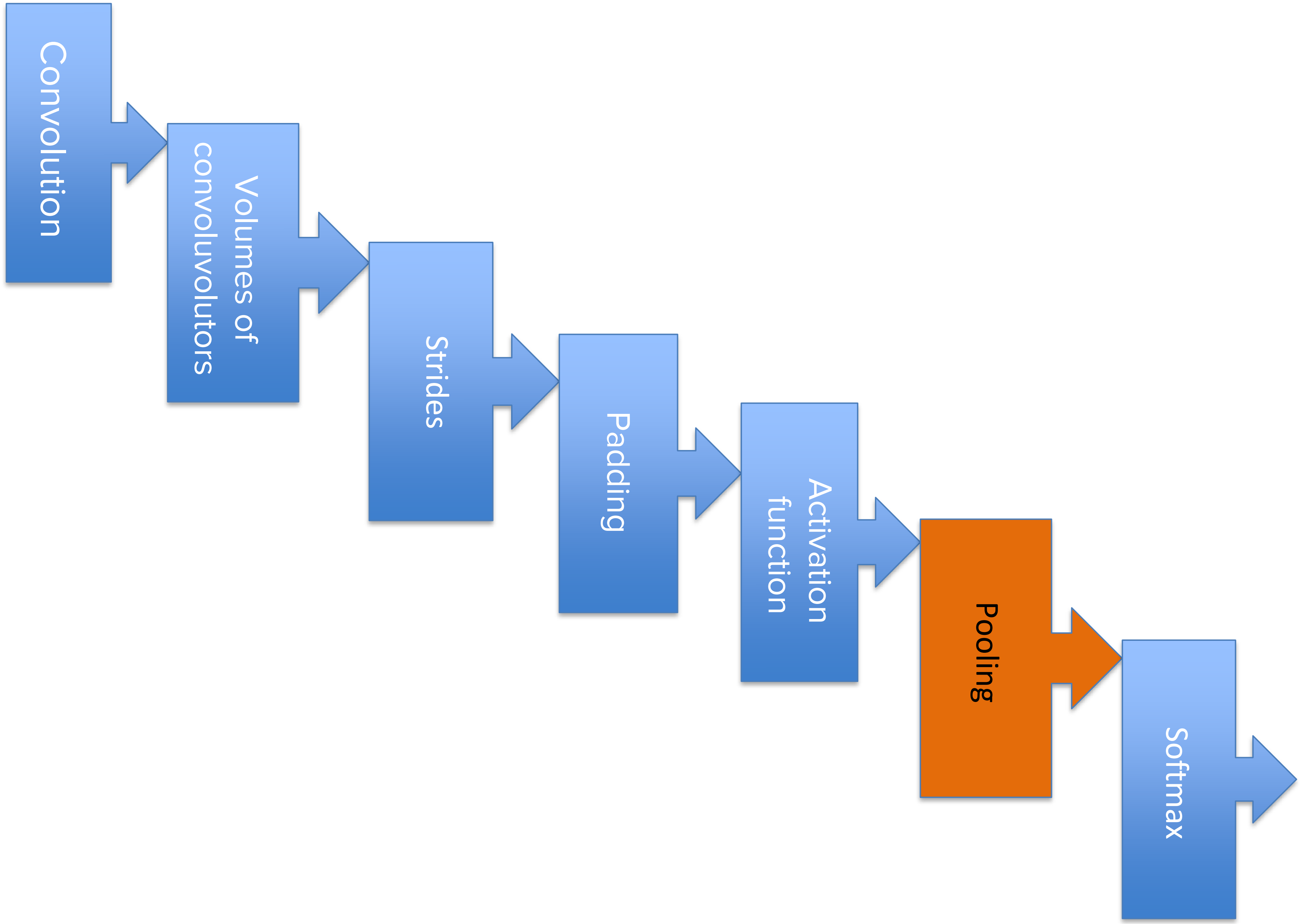
Table 3: Non-linearities tested.

Name	Formula	Year
none	$y = x$	-
sigmoid	$y = \frac{1}{1+e^{-x}}$	1986
tanh	$y = \frac{e^{2x}-1}{e^{2x}+1}$	1986
ReLU	$y = \max(x, 0)$	2010
(centered) SoftPlus	$y = \ln\left(e^{x}+1\right) - \ln 2$	2011
LReLU	$y = \max(x, \alpha x), \alpha \approx 0.01$	2011
maxout	$y = \max(W_1 x + b_1, W_2 x + b_2)$	2013
APL	$y = \max(x,0) + \sum_{s=1}^{S} a_i^s \max\left(0, -x + b_i^s\right)$	2014
VLReLU	$y = \max(x, \alpha x), \alpha \in 0.1, 0.5$	2014
RReLU	$y = \max(x, \alpha x), \alpha = \text{random}(0.1, 0.5)$	2015
PReLU	$y = \max(x, \alpha x), \alpha$ is learnable	2015
ELU	$y = x$, if $x \geq 0$, else $\alpha(e^{x} - 1)$	2015

To learn more on CNN activation functions: link

ELU, ReLU, LReLU

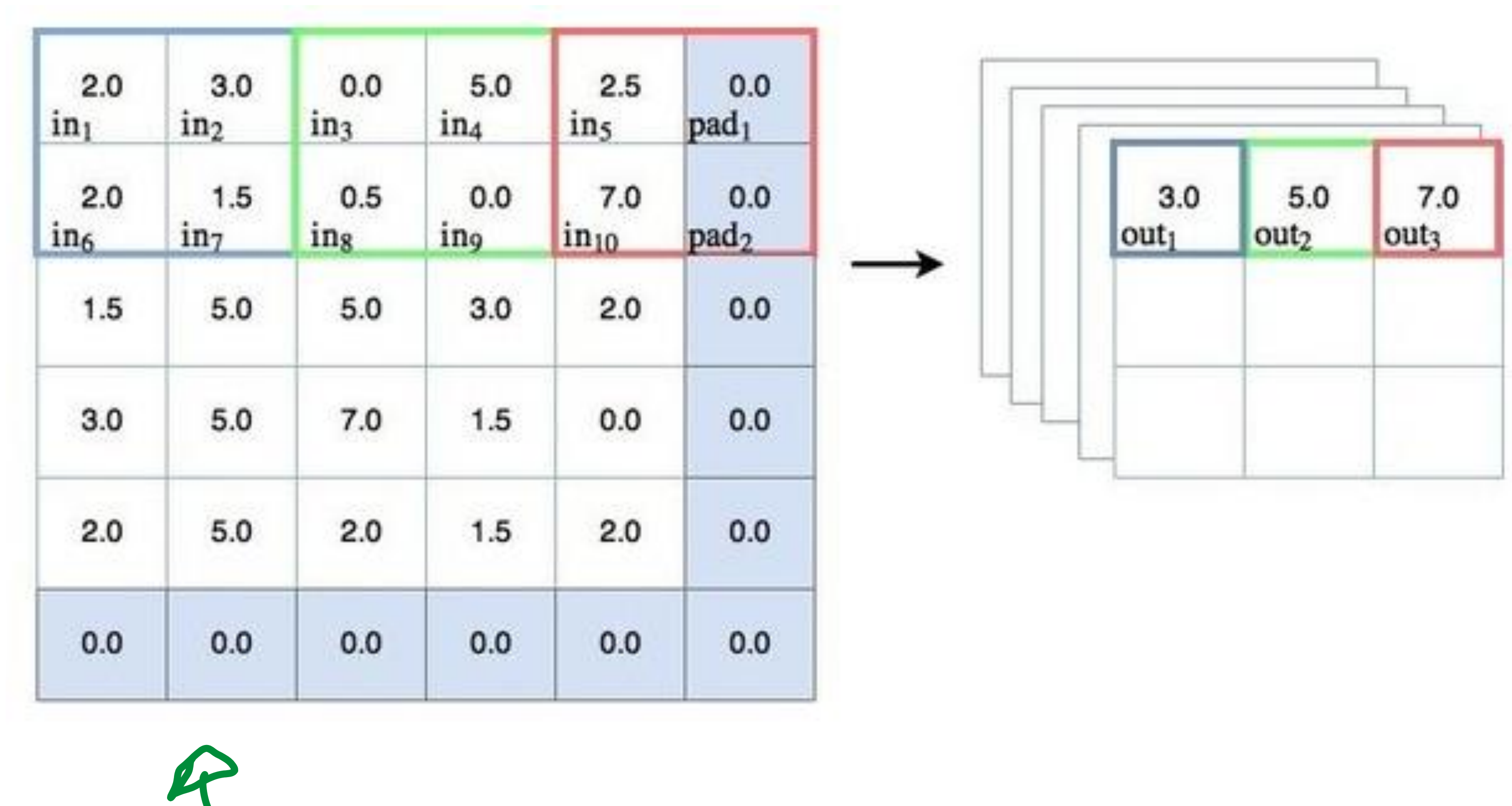



Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

Pooling Layer

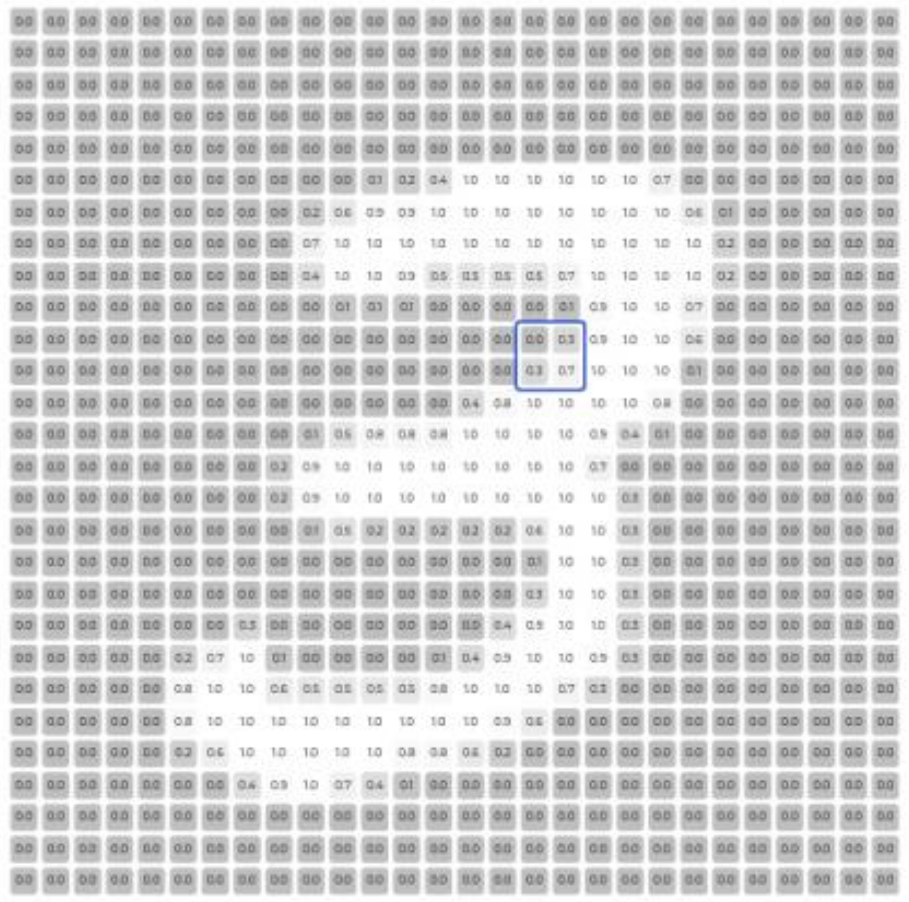
- Pooling is to aggregate data in input volume, to generate **lower dimension feature maps** (it is an additional way to compress data).
- A **pooling layer** is a new **layer** added **after** the convolutional **layer** and after nonlinearity (e.g. ReLU) has been applied;
- A Pooling layer is frequently used in convolutional neural networks, with the purpose to progressively reduce the spatial size of the representation, to reduce the number of features and the computational complexity of the network.
- The main reason for the pooling layer is to prevent the model from overfitting. The idea is to reduce the number of unnecessary details, keeping the "invariants".
- "Reasonably" invariant operators for small translations are the *average* and *max* pooling
- The pooling layer operates upon **each feature map** separately, to create a new set of the same number of pooled feature maps
- It makes sense for images (not always reasonable for other types of input) to reduce variability due to rotation and scaling

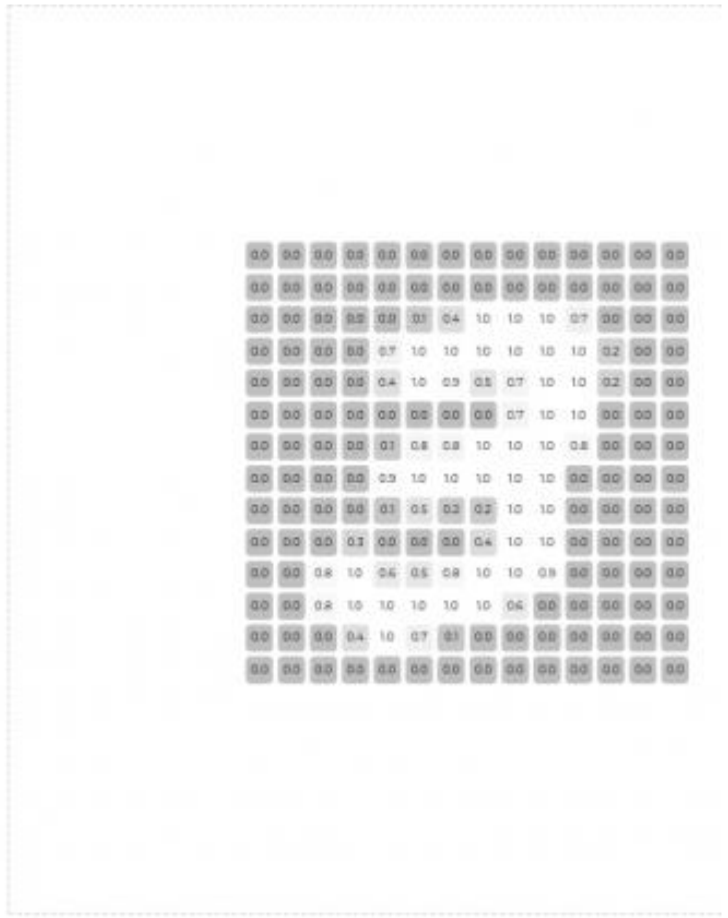
Pooling operations: Max pooling

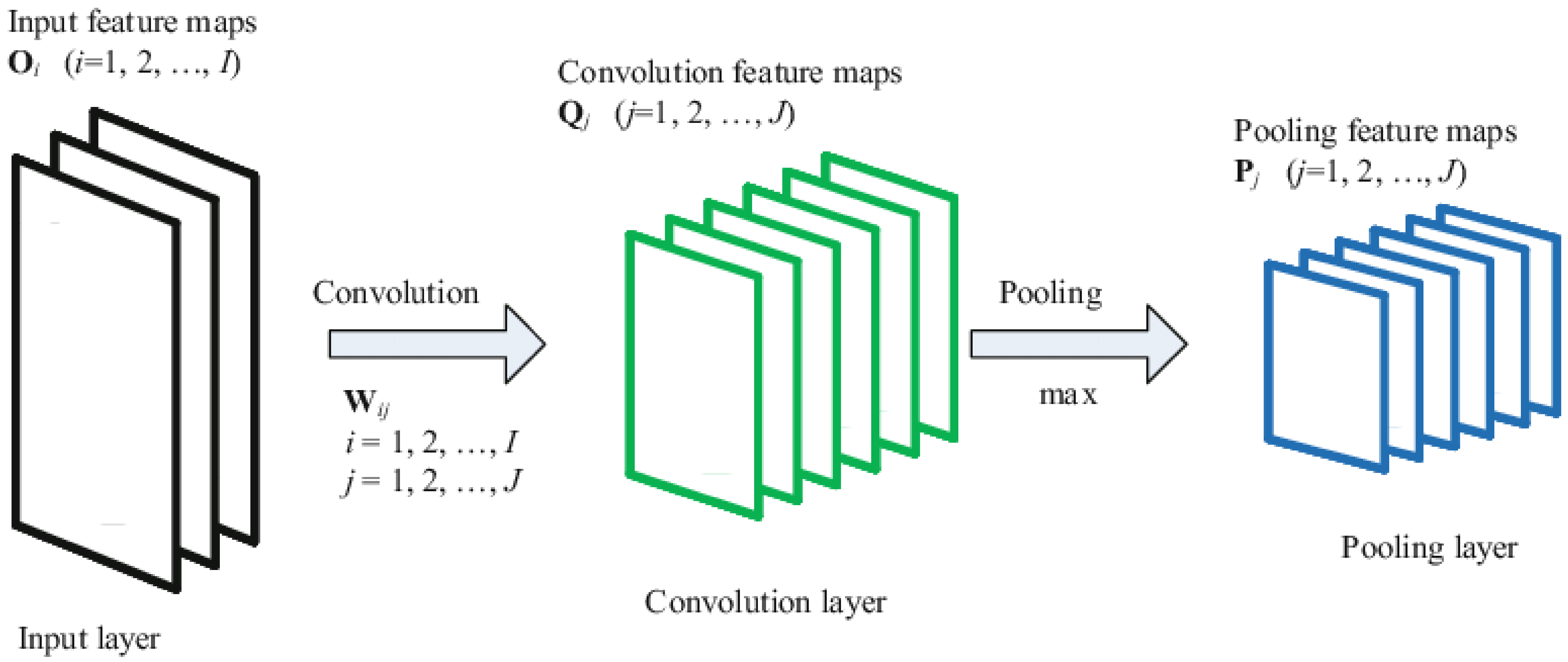


Output slice j of layer *i*

Example of effect of max pooling

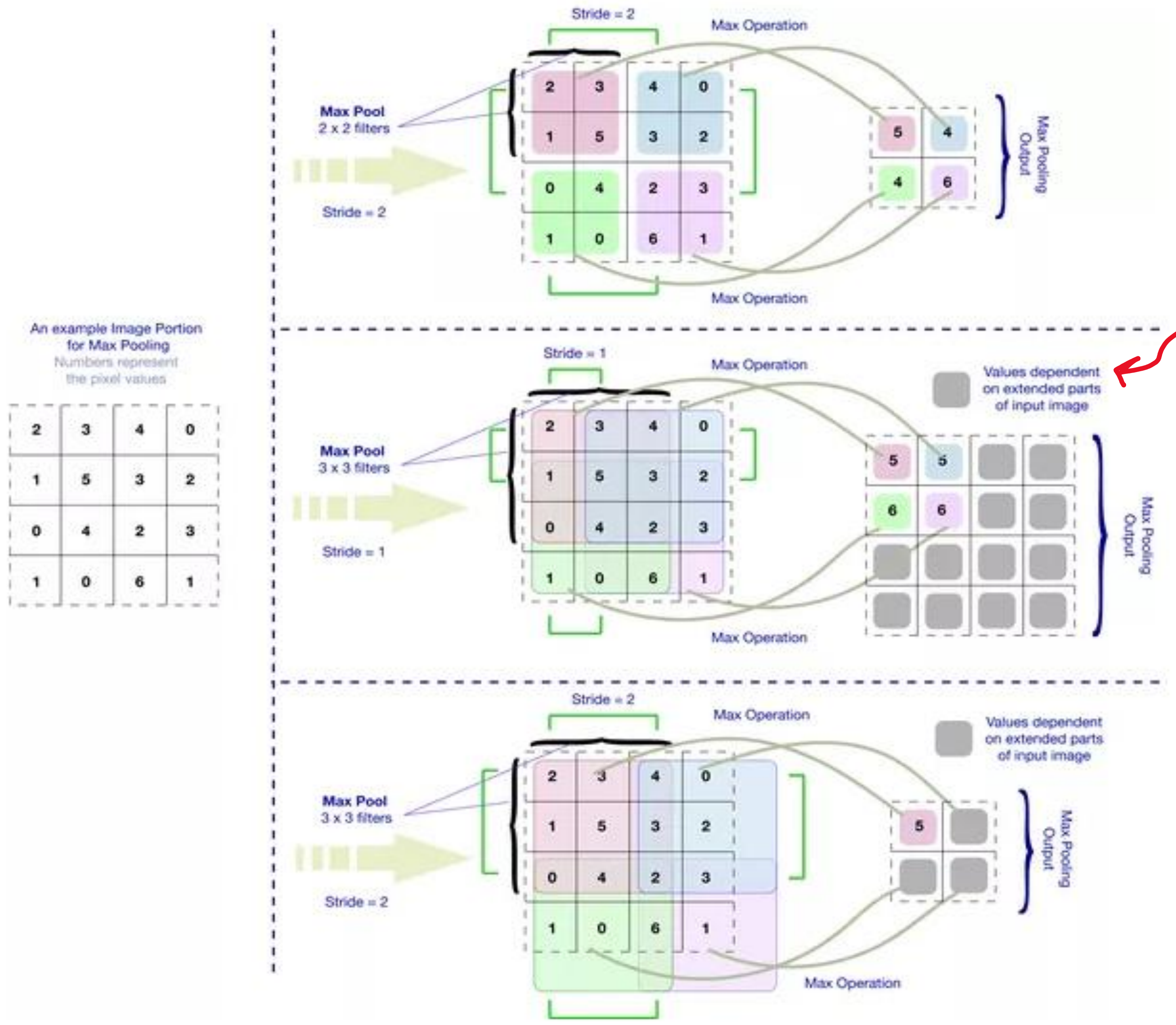






Input feature maps
$\mathbf{O}_i$ (i=1, 2, …, I)
Convolution feature maps
$\mathbf{Q}_j$ (j=1, 2, …, J)
Pooling feature maps
$\mathbf{P}_j$ (j=1, 2, …, J)
Convolution
$\mathbf{W}_{ij}$
i = 1, 2, …, I
j = 1, 2, …, J
Pooling
max
Input layer
Convolution layer
Pooling layer

Pooling with different kernels and stride dimensions



What numbers do you have on paddig layers?

Pooling methods

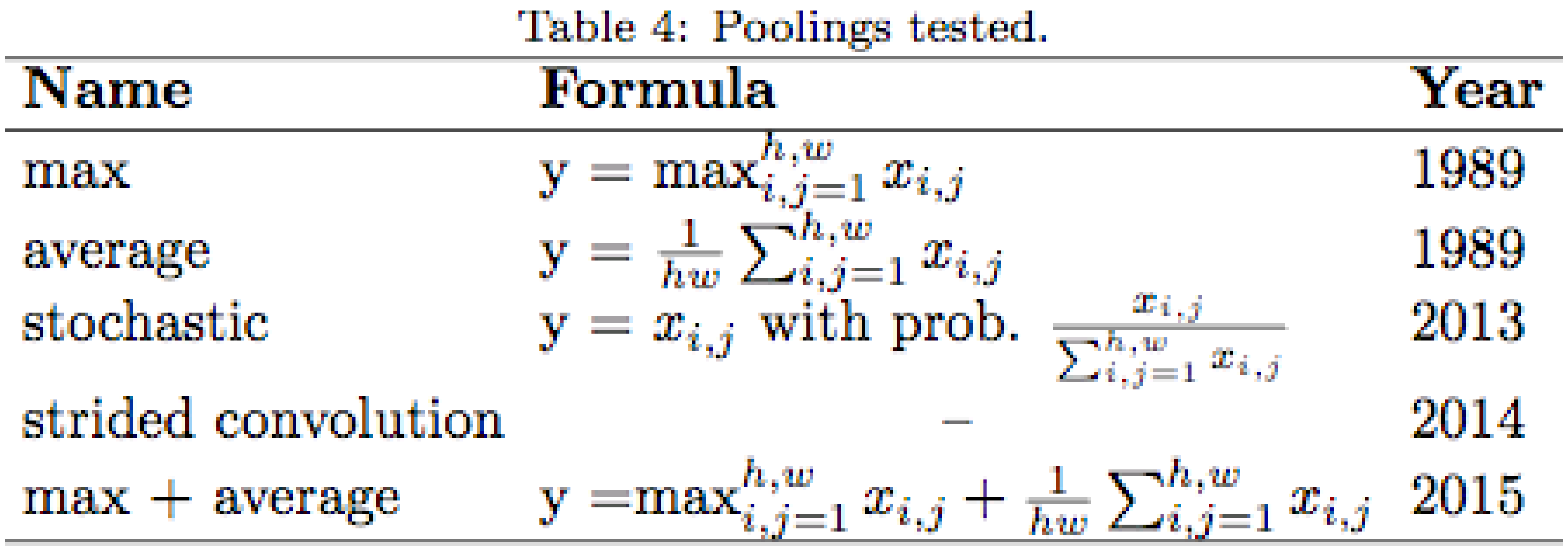


Table 4: Poolings tested.

Name	Formula	Year
max	$y = \max_{i,j=1}^{h,w} x_{i,j}$	1989
average	$y = \frac{1}{hw} \sum_{i,j=1}^{h,w} x_{i,j}$	1989
stochastic	$y = x_{i,j}$ with prob. $\frac{x_{i,j}}{\sum_{i,j=1}^{h,w} x_{i,j}}$	2013
strided convolution	–	2014
max + average	$y = \max_{i,j=1}^{h,w} x_{i,j} + \frac{1}{hw} \sum_{i,j=1}^{h,w} x_{i,j}$	2015

More on: link

Advantages of pooling

- The idea is that pooling creates "summaries" of each sub-region.
- **Dimension Reduction:** In deep learning when we train a model, because of excessive data size the model can take a huge amount of time for training.
- Consider e.g., the use of max-pooling of size 5x5 with stride 1, applied after the convolution operation. It reduces each successive region of size 5x5 of a slice to a 1x1 region with a max value of the 5x5 region.

Advantages of pooling

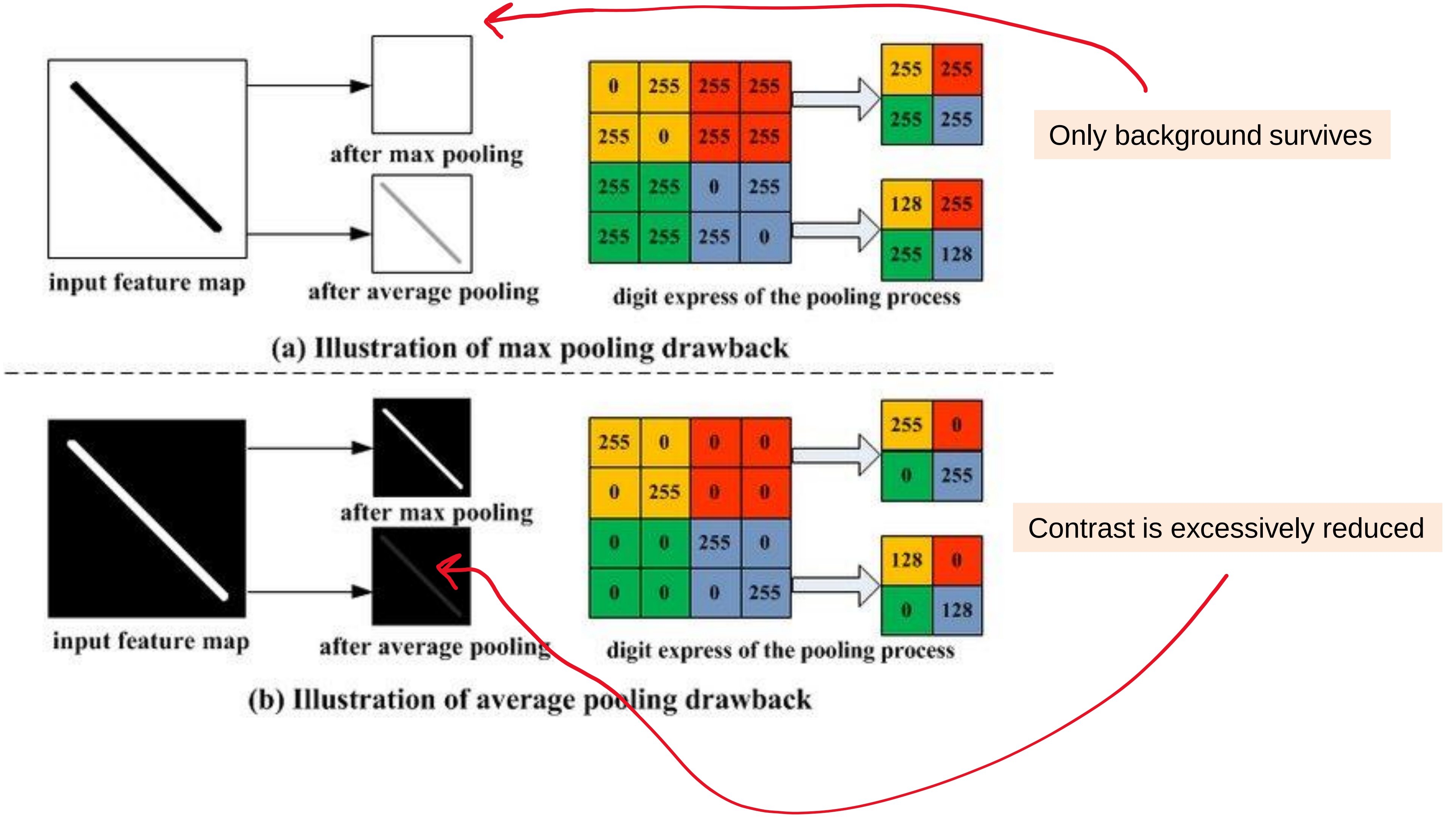
- **Rotational/Position Invariance Feature Extraction:** Pooling can also be used for **extracting rotational and position invariant feature**.
- Consider the same example of using pooling of size 5x5. Pooling extracts the max value from the given 5x5 region. Extract **the dominant feature value** (max value) from the given region, irrespective of the position of the feature value.
- The max value would be the same **from any position** inside the region. Pooling thus provides rotational/positional **invariant** feature extraction.

Pooling Caveat

- It is important to be careful in the use of the pooling layer since you are actually reducing the "context" of elements within a given input (e.g., pixels surrounding a pixel; words to the left and right of a word; etc)
- While it would help at significantly reducing the complexity of the model, it might cause to lose the «location sensitivity» in the model.
- This is particularly problematic e.g., for videos, where **context** might be very important (e.g., detecting trajectories)

Pooling Drawbacks

example



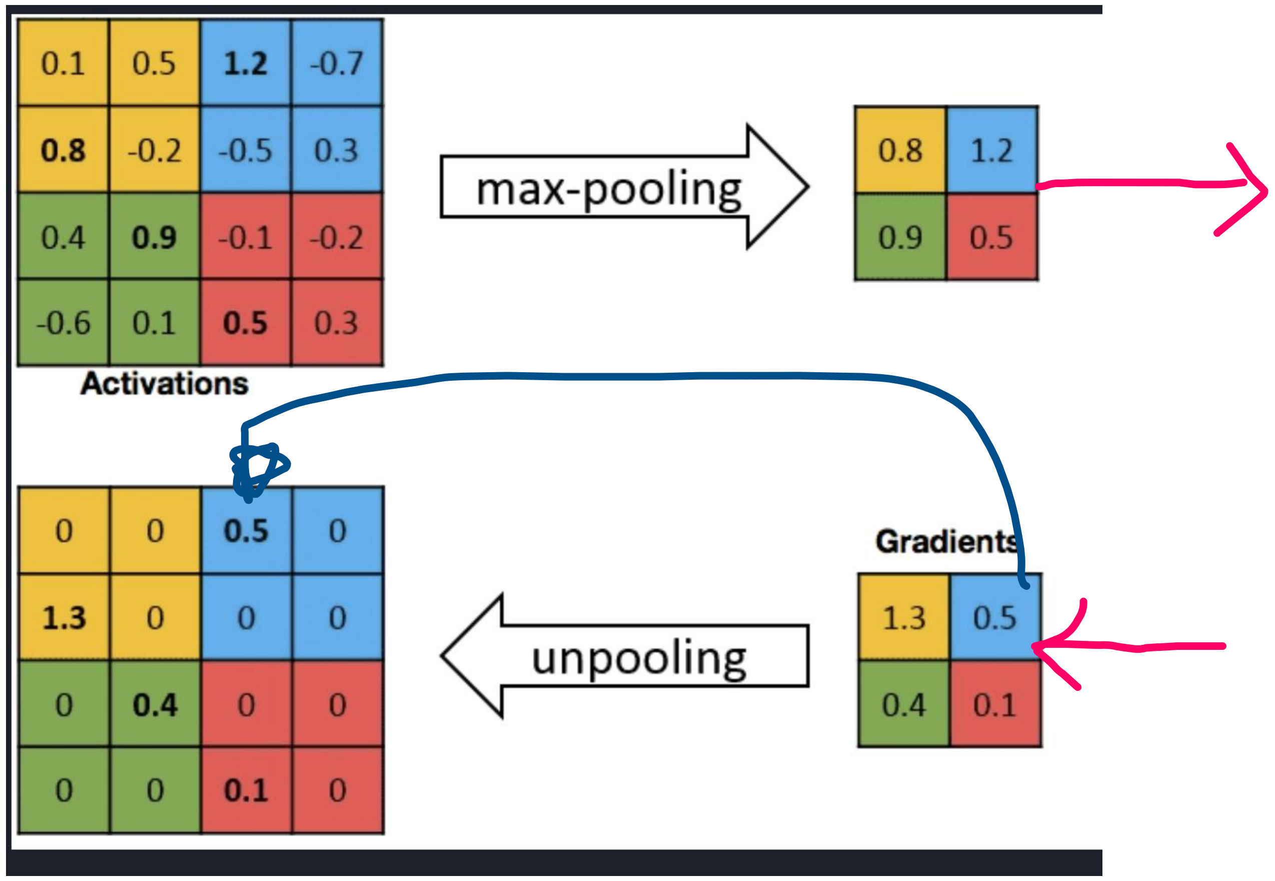
(a) Illustration of max pooling drawback

(b) Illustration of average pooling drawback

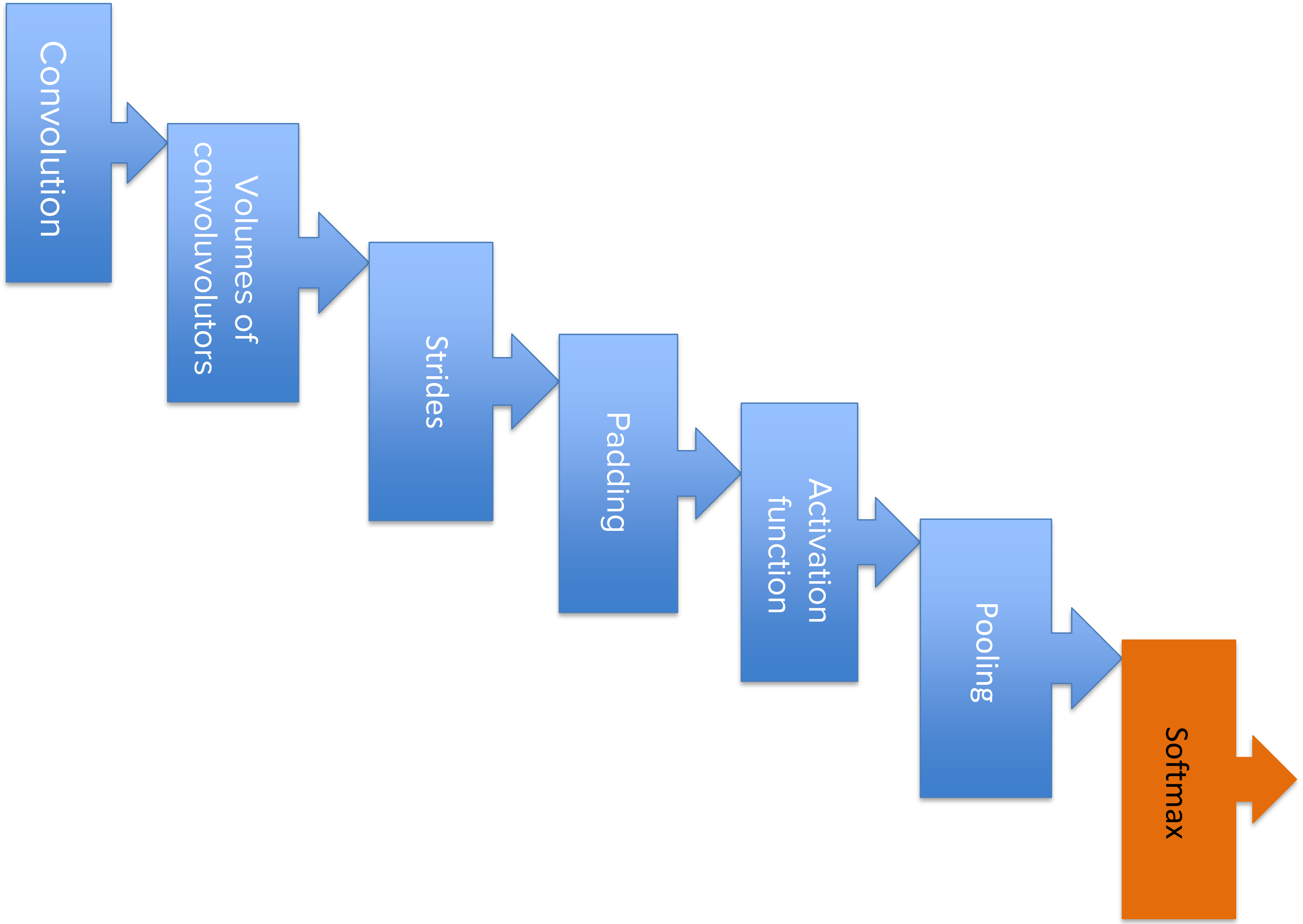
Only background survives

Contrast is excessively reduced

What about backpropagation of the pooling layers?

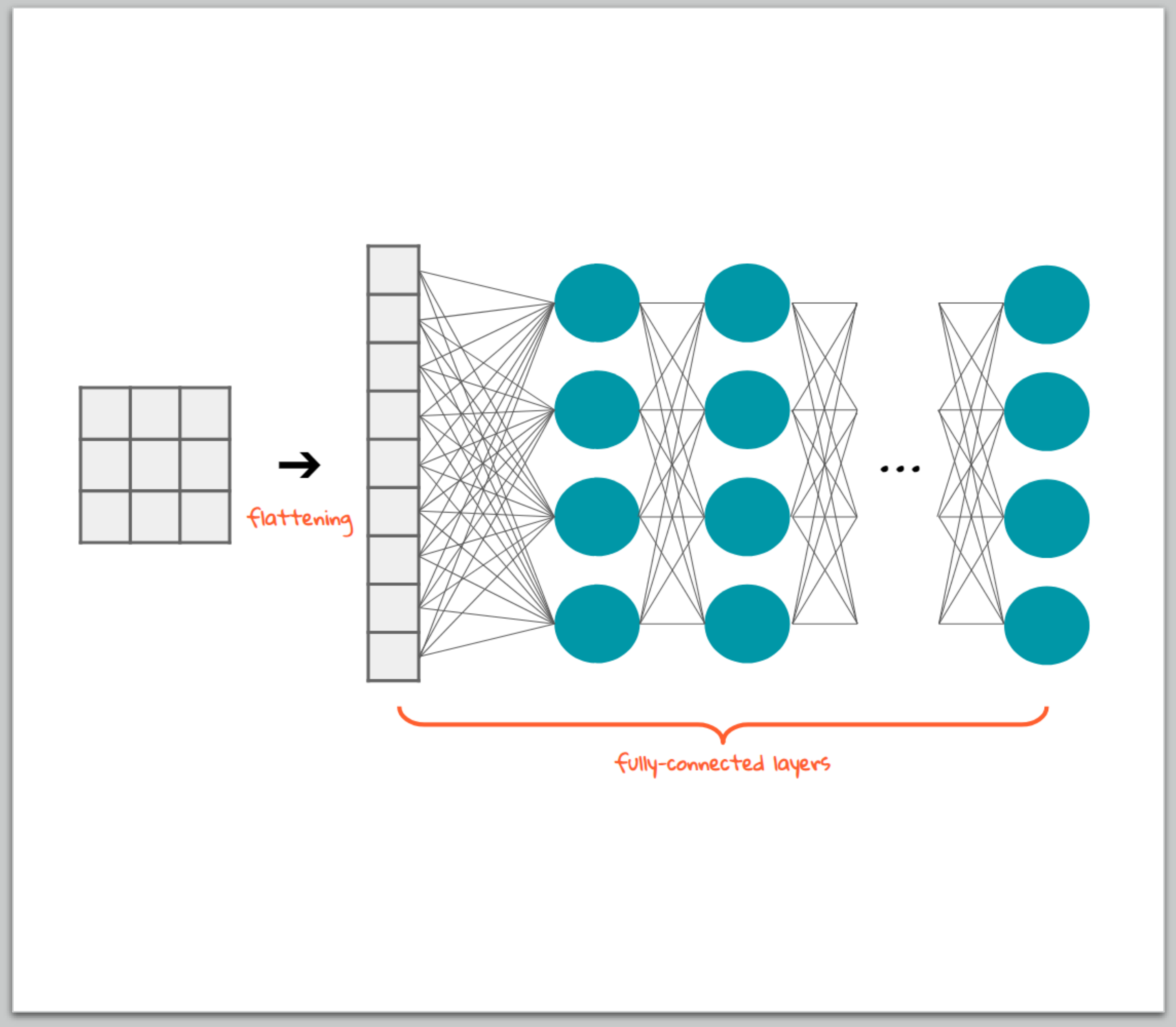


- Intuitively a non-max value will not affect the output, since the output is only concerned about the max value in the filter. Therefore the non-max values have a gradient of 0.
- (note the gradients flowing from subsequent layers are set here to "some" value just to provide an example)


Convolution
Volumes of convoluvolutors
Strides
Padding
Activation function
Pooling
Softmax

The last layer is a fully connected layer

- Therfore the first operation to connect the output volume of the last CNN layer before the FC layers is FLATTENING (maps are converted into vectors)



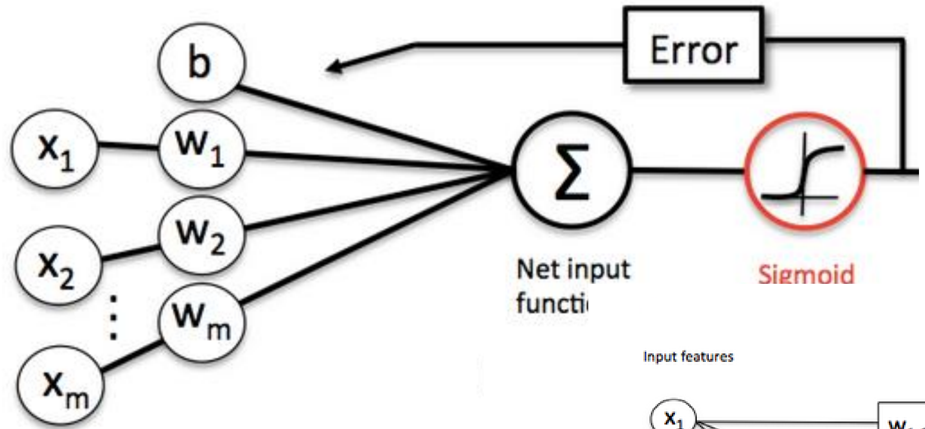
Softmax Function

- In the FC layer, the activation level net_k of individual neurons is calculated in the usual way , but the activation function for the last output layer is (instead of Sigmoid or ReLU) the Softmax:

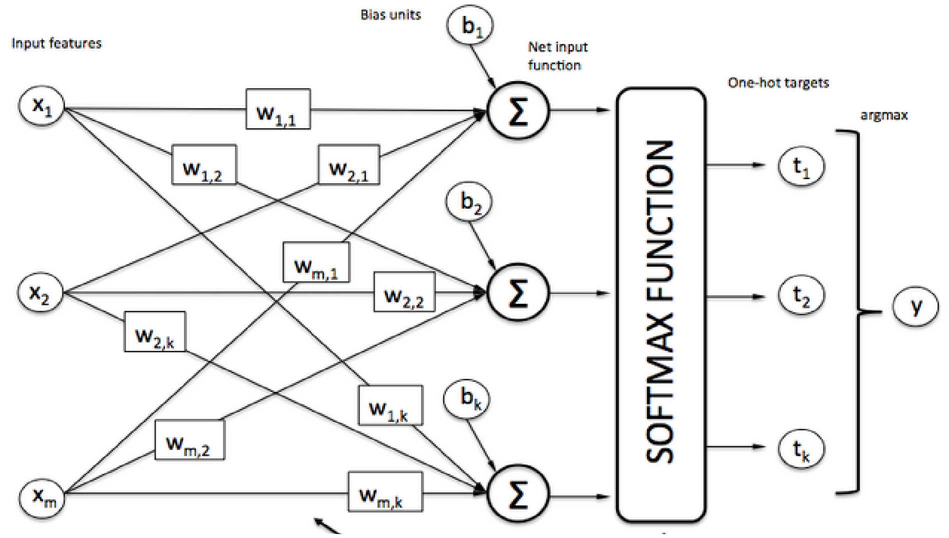
$$y_k = \mathrm{f}(net_k) = \frac{e^{net_k}}{\sum_j e^{net_j}}$$

where the y_k can be interpreted **as probabilities** since they are in the [0,1] range

Softmax VS Sigmoid

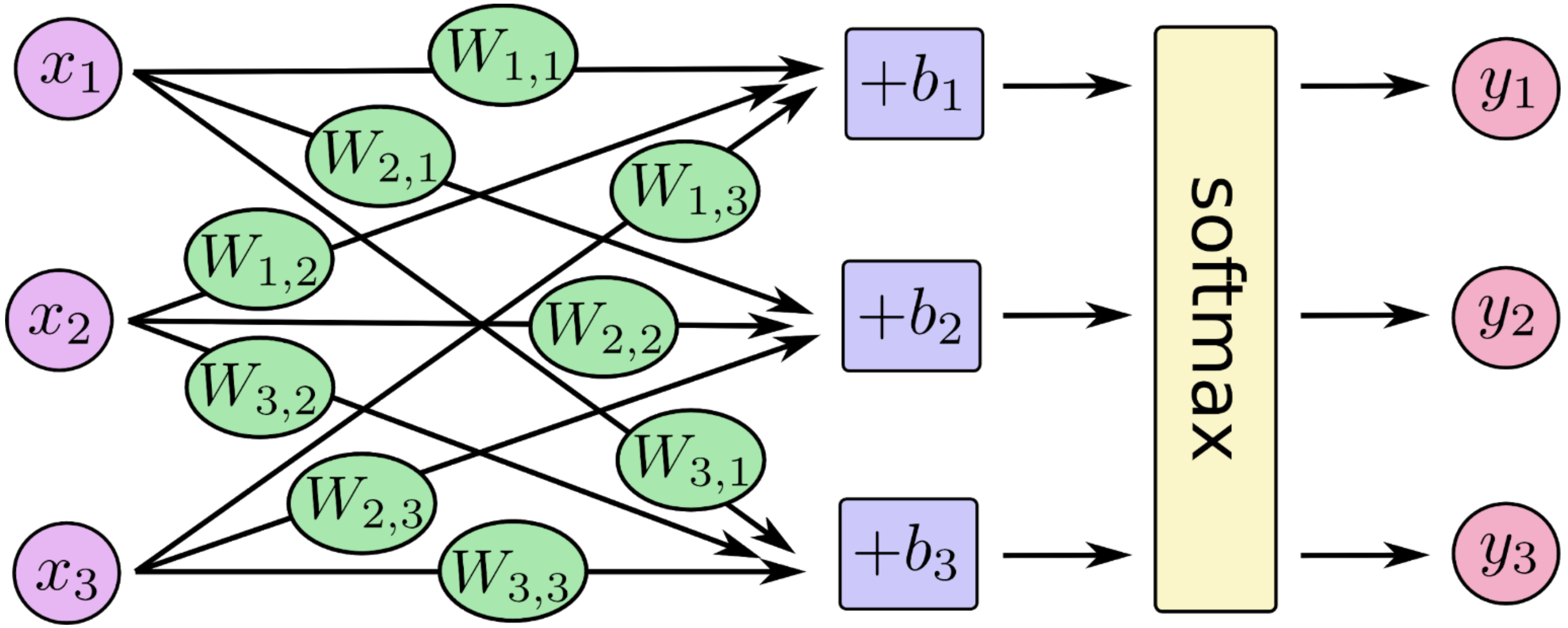


Note: **b** here is the bias/threshold, as usual



Fully connected NN layer with softmax

Example (1)



x_1
x_2
x_3
$W_{1,1}$
$W_{2,1}$
$W_{1,3}$
$W_{1,2}$
$W_{2,2}$
$W_{3,2}$
$W_{3,1}$
$W_{2,3}$
$W_{3,3}$
$+b_1$
$+b_2$
$+b_3$
softmax
y_1
y_2
y_3

Example (2)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

Example (3)

in matrix form

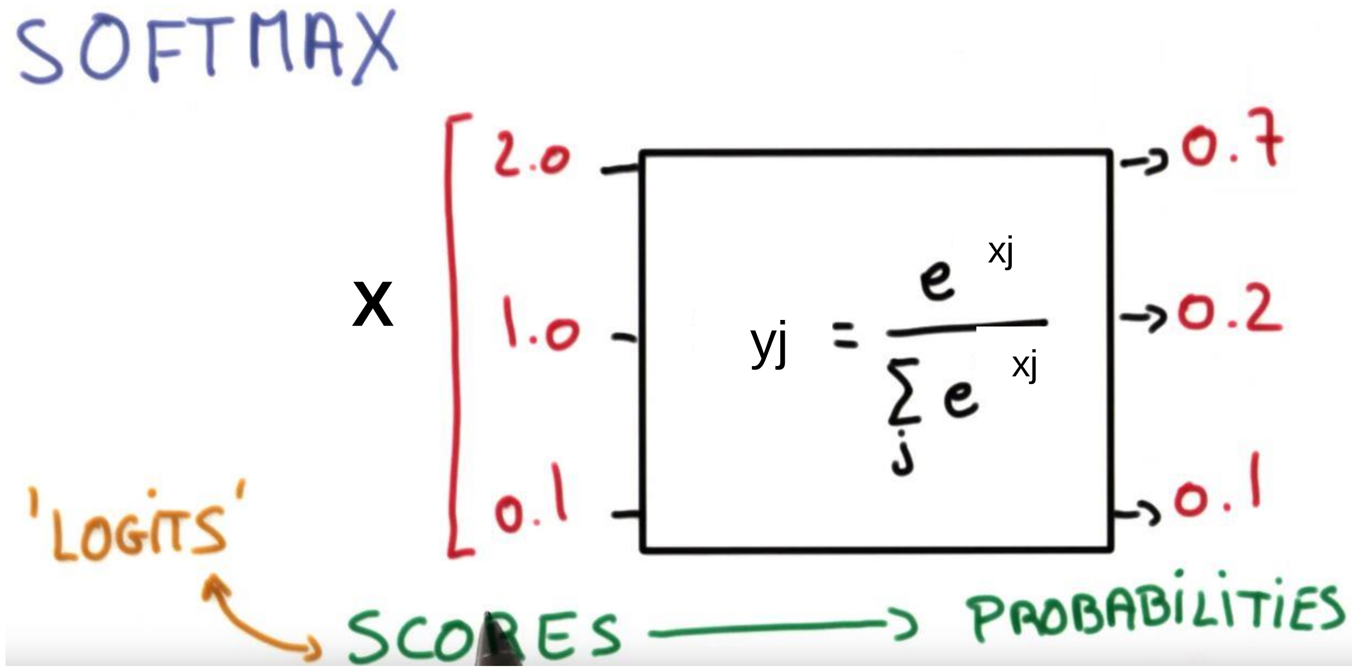
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

$$y = \text{softmax}(Wx + b)$$

The effect of the Softmax function

From scores to probabilities. Highest probabilities are

assigned to highest values



Computing the Loss in the softmax layer

Gradient descent of a loss function, as for NN:

- However:
  - ➢ Commonly, we use **Cross-Entropy** in the **final layer** of a convolutional net
  - ➢ The cross-entropy between two discrete distributions p and q (measuring how far q differs from p for fixed p) is defined by:

$$H(p,q) = -\sum_{v} p(v) \cdot log(q(v))$$

Computing the error (e.g., for multi-class classifiers)

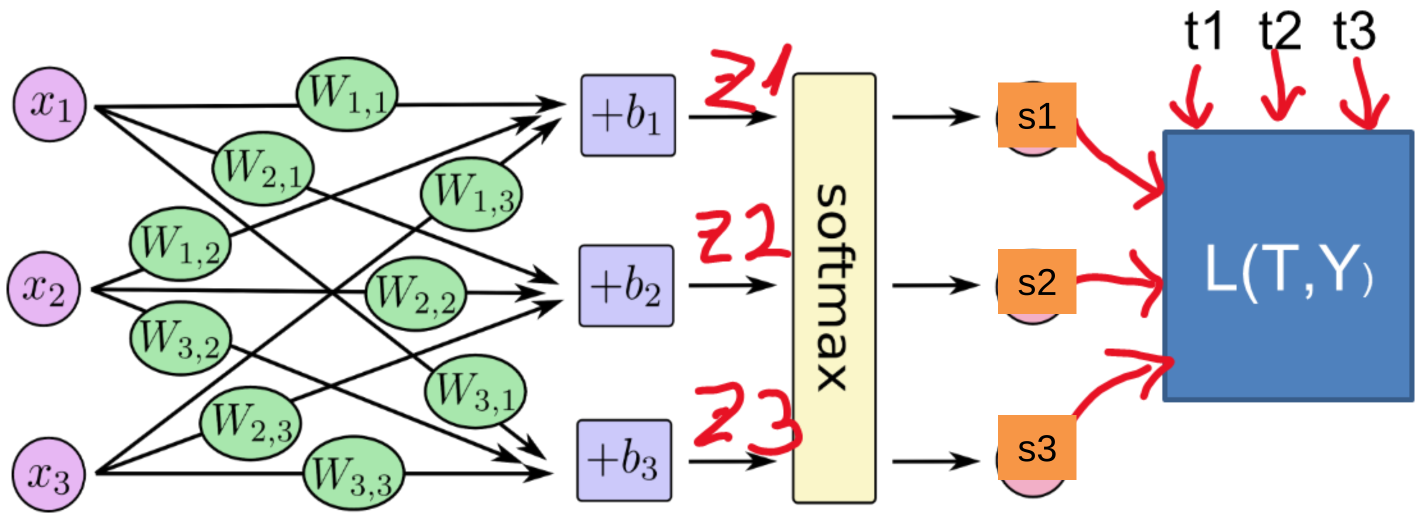
- Let **T**^{h}=[0,0..**1**,0..0] be the ground truth multi-class classification (n classes) for input instance ***in***^{h} $\in D$ $h=1..|D|$ (a 1 value in position *k* of **T**^{h} indicates that the correct classification for ***in***^{h} is the *k-th* class label),
- let **Y**^{h} be the output (softmax) vector , assigning probabilities to each possible class value;
- The cross-entropy H(**T**^{h}, **Y**^{h}) is given by (for n classes):

$$H(\boldsymbol{T^h},\boldsymbol{Y}^h)=\sum_{j=1..n} t_j^h \log\left(y_j^h\right) = \sum_{j=1..n} t_j^h \log\left(s_j^h\left(x^h{}_j\right)\right)$$

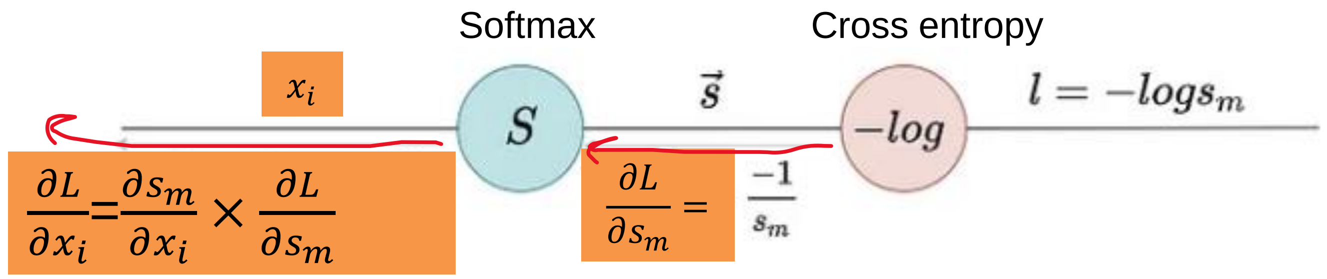
$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right)$$

Backpropagation on softmax layer

- First of all, note that the ground truth vector T: [t1,t2,…tk] is a «one-hot» vector, only one element (say, t_m) is =1, and all the others are zero (the softmax is used for classifiers!)
- Therefore, $\frac{\partial L}{\partial y_i} = \frac{\partial L}{\partial s_i}$=0 $if\ i \neq m\ and\ -\frac{1}{s_m}\ if\ i = m$
- $where\ s_i$ is the softmax of z_i



Backpropagation of softmax layer


Softmax
Cross entropy
x_i
S
$\vec{s}$
$-log$
$l = -log s_m$
$\frac{\partial L}{\partial x_i} = \frac{\partial s_m}{\partial x_i} \times \frac{\partial L}{\partial s_m}$
$\frac{\partial L}{\partial s_m} = \frac{-1}{s_m}$

We can write $\frac{\partial s_m}{\partial x_i} = s_m(1 - s_m)$ if i=m else $-s_i s_m$ (see al the derivation steps here)

This means that the backward gradient is $-s_m(1 - s_m)\frac{1}{s_m} = (s_m - 1)$ at position m, and $-s_i s_m(-\frac{1}{s_m}) = s_i$ for $i \neq m$

Backpropagation of the softmax layer (general formulation)

$$\partial L/\partial w_{ij} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

$$L(H(T,Y) = -\sum_j t_j log y_j$$

$$y_j = s_j = \frac{e^{z_j}}{\sum_i e^{z_i}}$$

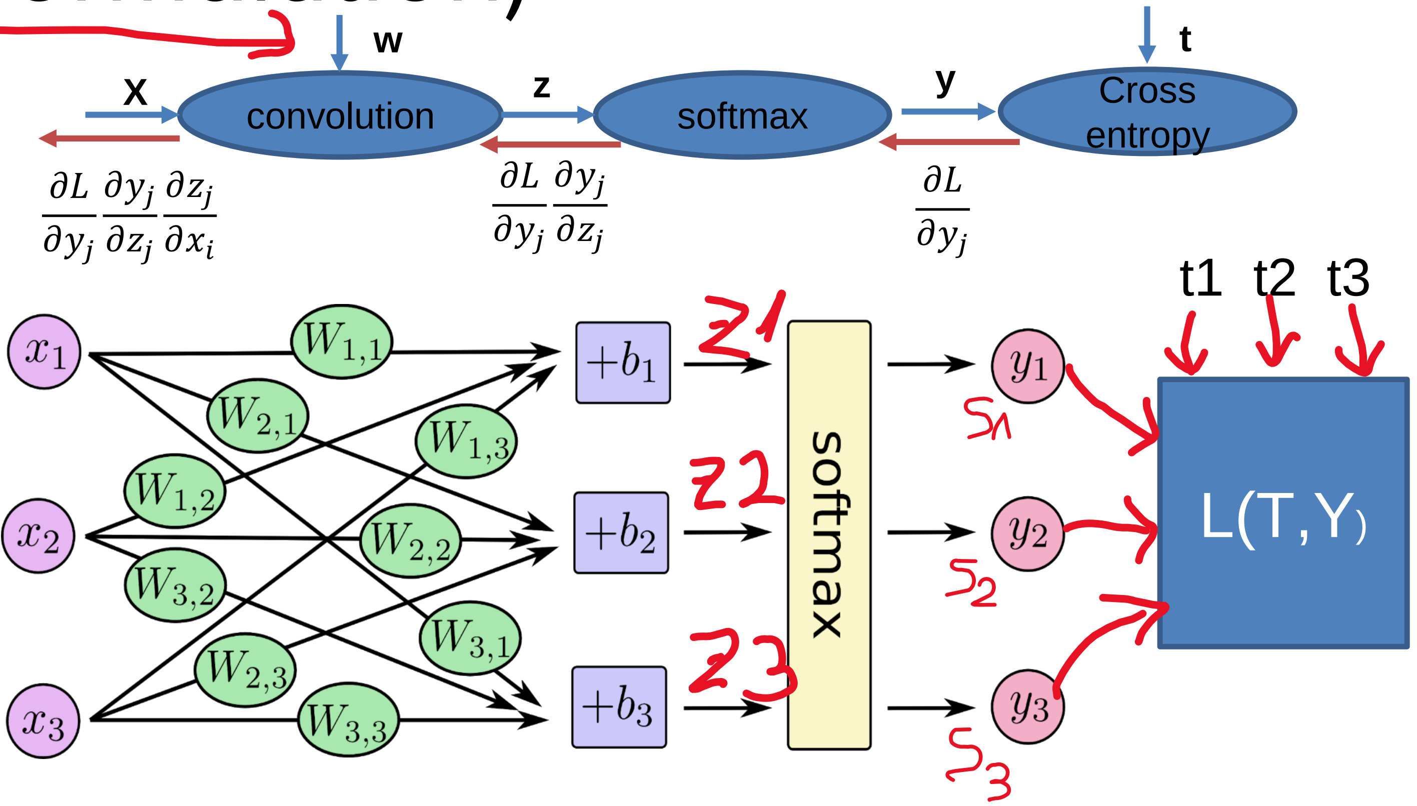
$$z_j = \sum_i w_{ij}x_i + bj$$

$$\frac{\partial L}{\partial s_j} = -\frac{t_j}{y_j}$$

$$\frac{\partial y_j}{\partial z_j} = s_j(1-s_j)$$

$$\frac{\partial z_j}{\partial w_{ij}} = x_i$$

$$\boxed{\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = -t_j(1-s_j)x_i}$$


w
t
x
convolution
z
softmax
y
Cross entropy
$\frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial x_i}$
$\frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial z_j}$
$\frac{\partial L}{\partial y_j}$
t1 t2 t3
x_1 x_2 x_3
$W_{1,1}$ $W_{2,1}$ $W_{1,2}$ $W_{1,3}$ $W_{2,2}$ $W_{3,2}$ $W_{3,1}$ $W_{2,3}$ $W_{3,3}$
$+b_1$ $+b_2$ $+b_3$
z1 z2 z3
softmax
y_1 y_2 y_3
s_1 s_2 s_3
L(T,Y)

Superscript h is omitted here

But, remember that $t_j = 0\ for\ all\ j \neq m\ and\ t_m = 1$
Therefore, only w_{im} (i=1…K) are updated!

Putting it all together

- The network is a sequence of convolution, activation and pooling layers with a **final fully connected layer** with **Softmax**
- Hyper-Parameters are: pooling and strides, padding, the number and type of kernels/filters and their dimension, activation function, number of hidden layers;
- Parameters to be estimated: **all weights** (kernels and final MLP connection weights)
- Clearly also the shape of input and input features is relevant: this is part of data pre-processing. For images, RGB pixels are often sufficient (no pre-processing)
- Method to estimate weights: backpropagation

More mathematical details

- Backpropagation in convolutional layers: link
- Softmax: link
- Much more in next semester (Prof. Rodolà)

STACKED DENOISING AUTOENCODERS (untrained deep NN)

Stacked Denoising Auto-encoders

- They are basically a «minimally trained» (or untrained) generalization of Backpropagation, with multiple layers, such as for CNN
- Purpose is to learn in an untrained manner the "latent" (essential) features of an input (images or any other type of non-sequential input)
- Main application is anomaly detection

Basic idea of autoencoders

- Learn a «compressed» representation of input data, one that preserves its essential features
- To learn this representation,
  - first compress the input (encoder),
  - then try to reconstruct it (decoder),
  - Finally, measure the reconstruction error (loss) and update the parameters to minimize it
- Once trained, the model is able to identify instances that subtantially deviate from normality (➔ high reconstruction error)

Auto-Encoders: Encoder+Decoder

- **Encoder**: The deterministic (= non-stochastic) mapping f_θ that transforms an input **x** into a «hidden» compressed representation **y**=f_θ(**x**)

$$f_\theta(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + \mathbf{b}).$$

  - ➢ **θ**= (**W**,b) are the *parameters* of the encoder (convolutional layers in CNN are encoders)

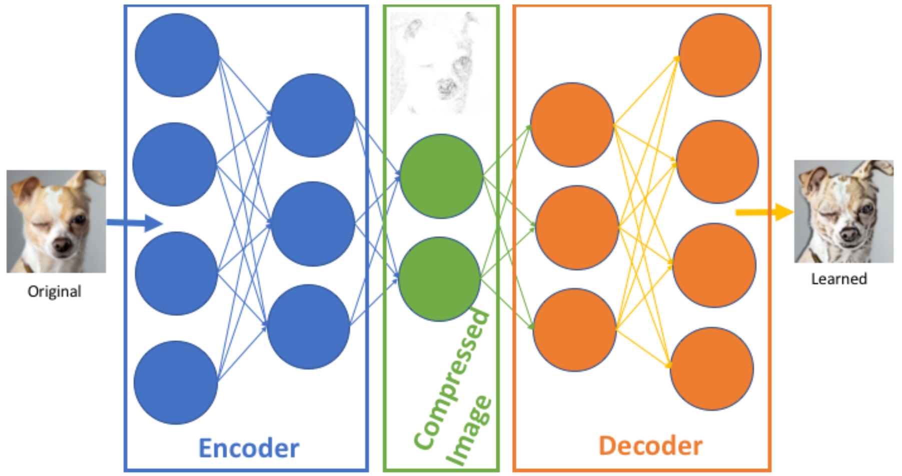
- **Decoder**: The resulting hidden representation **y**=f(**x**) is then mapped back to a «reconstructed» **d** dimensional vector **z** via a «reverse» decoding function g(**y**)

$$\mathbf{z} = g_{\theta'}(\mathbf{y}) \qquad g_{\theta'}(\mathbf{y}) = s(\mathbf{W}'\mathbf{y} + \mathbf{b}')$$

Auto-Encoders

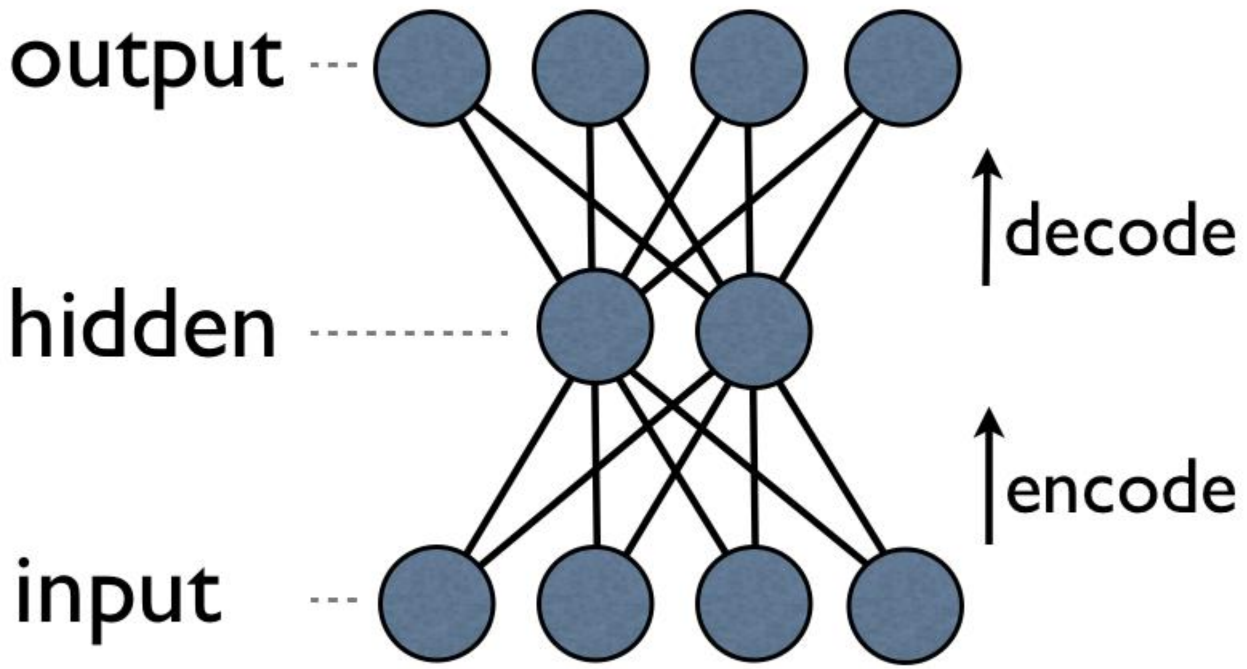
$$\mathbf{z} = g_{\theta'}(\mathbf{y}) \qquad g_{\theta'}(\mathbf{y}) = s(\mathbf{W}'\mathbf{y} + \mathbf{b}')$$

- In general, **z** is not to be interpreted as an **exact reconstruction of x**, but rather, in probabilistic terms, as the parameters ***θ*** (typically the mean) of a distribution **p(X=x|Z = z)** that «may generate **x** with high probability» where p(**x**|**z**) denotes the conditional probability of **x** given **z**.
- **Autoencoder training** objective consists of minimizing the **reconstruction error** (the difference between **x** and z=**$g_{\theta'}$(y)**)
- Intuitively, if a representation *f(**x**)* allows a good reconstruction of its original input **x**, it means that it has retained much of the information that was present in that input (=**identifying the «essential» features**).



Auto-encoders Implementation
NN with backpropagation

- They can be implemented as a neural network with one (or more) hidden layer(s) and an **equal number of nodes** in input and output
- Can be trained **without** supervision, since we don't need to know the class of an instance: the task is simply **to learn the weights** in order for the output to be similar to the input (or better: to learn the "essential" features of the input instances)
- Often used an as **anomaly detector**: if an instance is very different from "normality", the learnd network is unable to reconstruct it (reconstruction error is high).



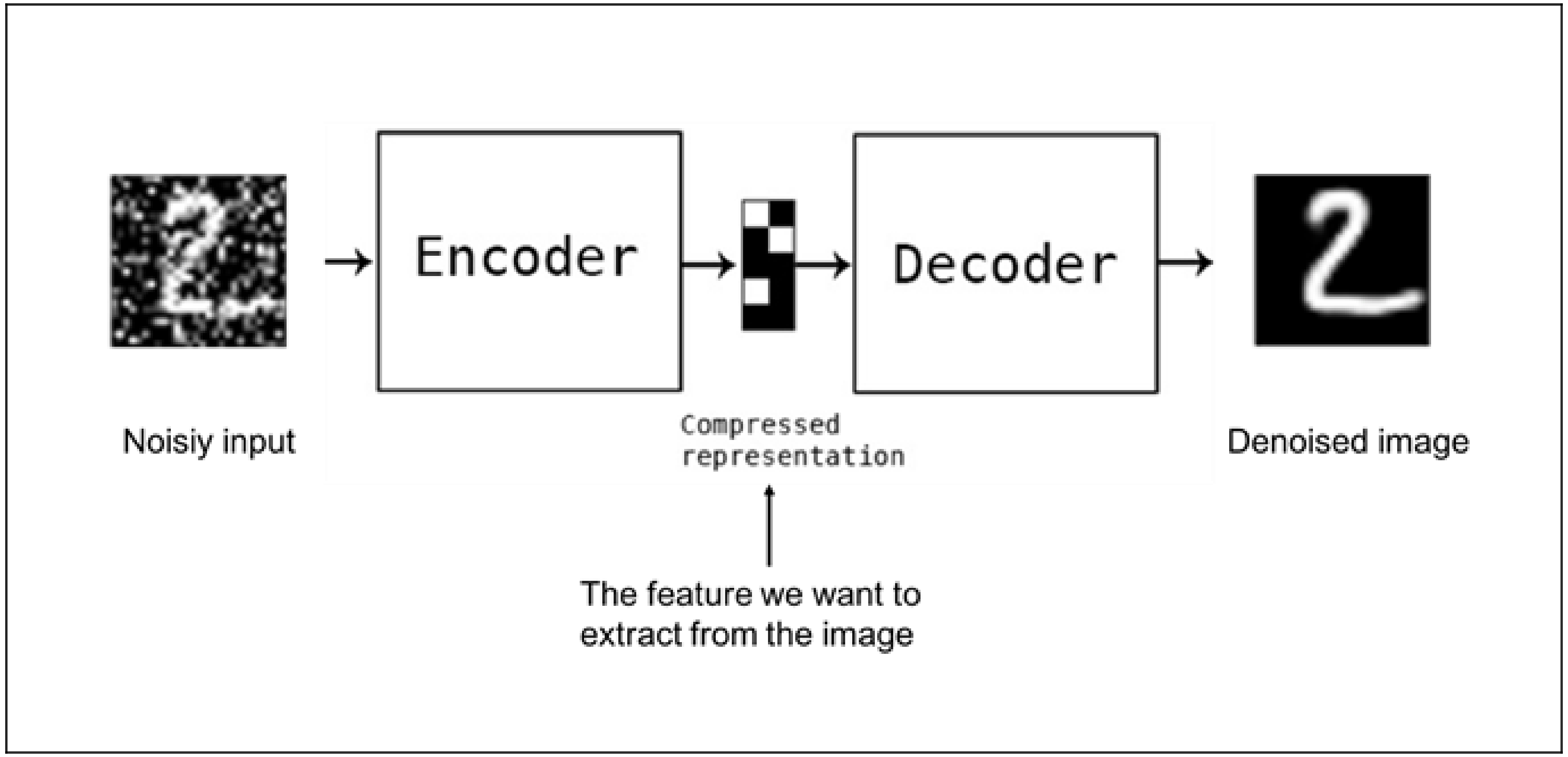
Denoising Auto-encoders a better method

- The reconstruction criterion alone is unable to guarantee the extraction of useful features as it can lead to the obvious («trivial») solution “simply copy the input on output”
- Denoising criterion:

  “A good representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input”

  - This means that we add some **random noise** to (=we corrupt) the input when training, to avoid trivial learning and detect the relevant features of the input in a “robust” way

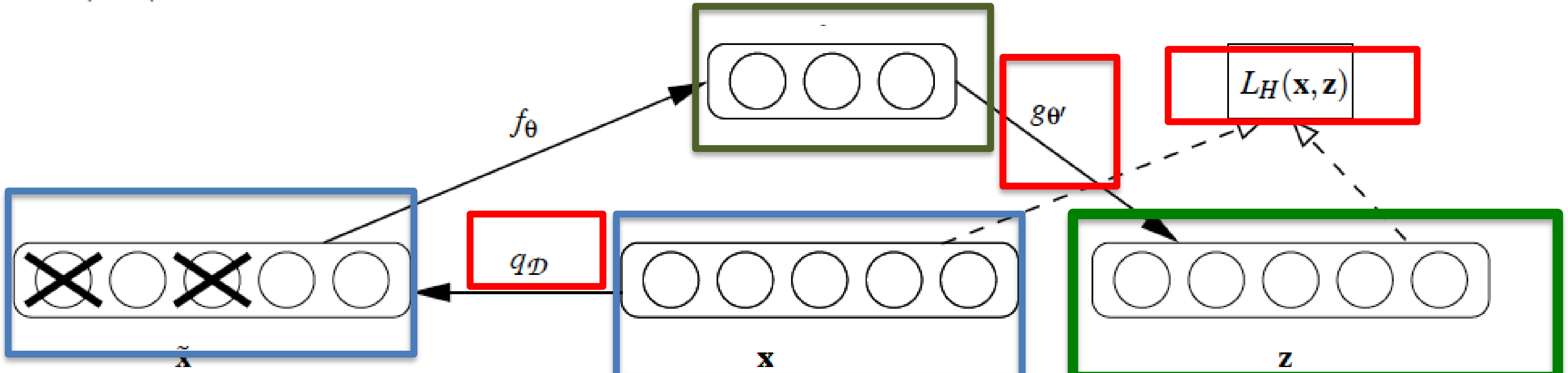
Denoising autoencoders


Encoder
Decoder
Noisiy input
Compressed representation
Denoised image
The feature we want to extract from the image

Denoising Auto-encoders Architecture

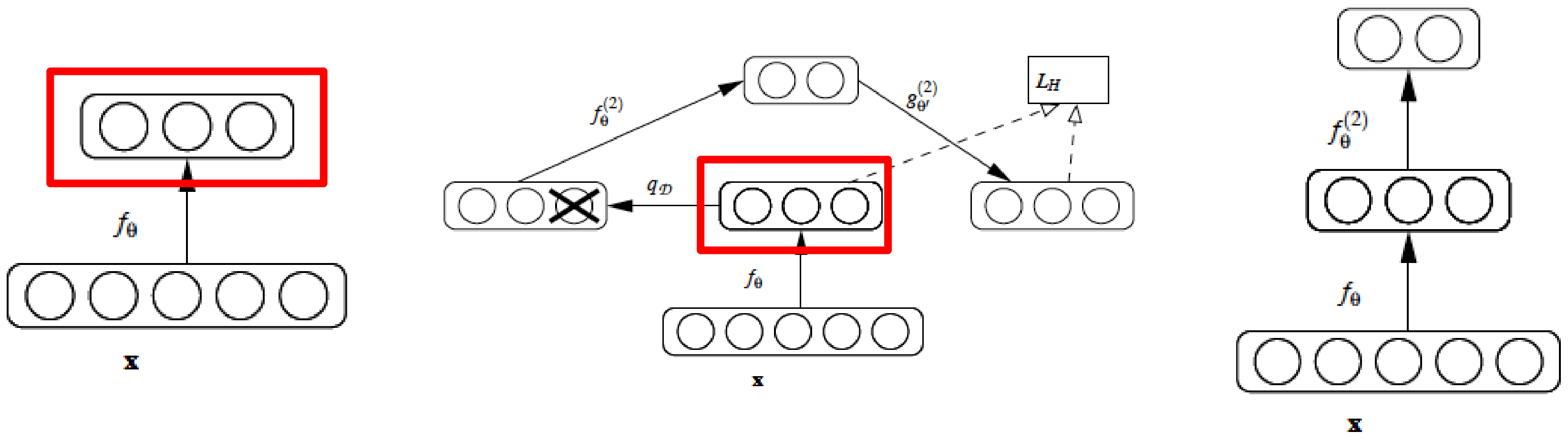
Denoising autoencoders can still be learned using backpropagation with gradient descent. What is the advantage? As in the figure, **y** is a "compact" representation of **x**, one that only retains only its essential features.

The denoising autoencoder architecture. An example $\mathbf{x}$ is stochastically corrupted (via $q_{\mathcal{D}}$) to $\tilde{\mathbf{x}}$. The autoencoder then maps it to $\mathbf{y}$ (via encoder f_θ) and attempts to reconstruct $\mathbf{x}$ via decoder $g_{\theta'}$, producing reconstruction $\mathbf{z}$. Reconstruction error is measured by loss $L_H(\mathbf{x}, \mathbf{z})$.



Stacked Denoising Auto-encoder Architecture

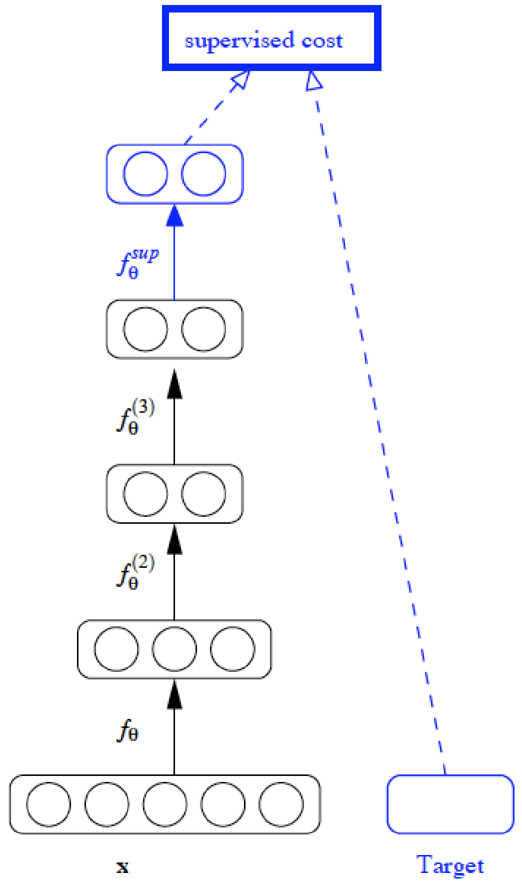
- After training a first level denoising autoencoder, **its learned encoding function $\mathbf{f}_\theta$** is used on “clean” input (no noise)
- The resulting representation is used to train a second level denoising autoencoder (middle) to learn a second level encoding function **f**.
- From there, the procedure can be repeated with more layers (for details, see this paper)



Staked vrs Deep autoencoders.. What's the difference?

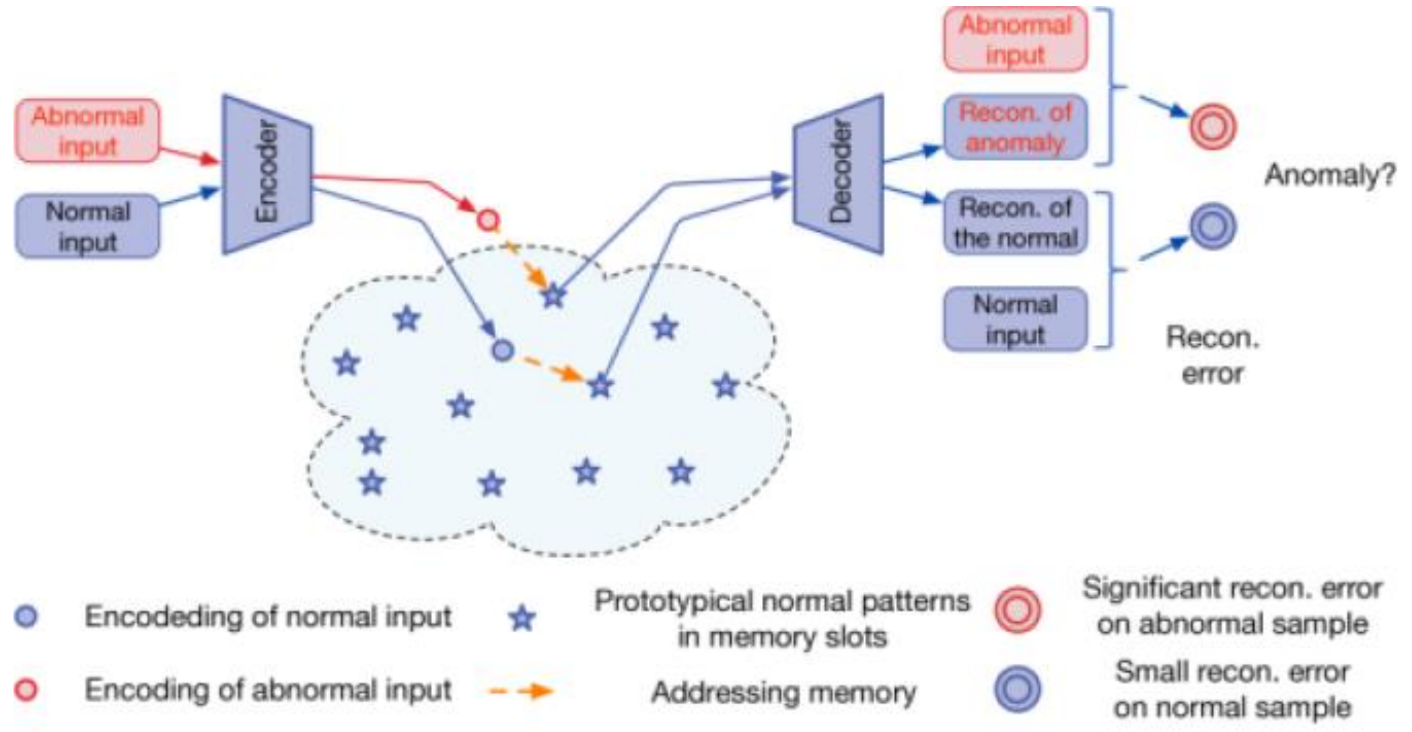
- The difference is in the way they are trained
- Deep autoencoders are trained with standard backpropagation, staked autoencoders are trained «layer by layer» as explained in the already indicated paper

SAE can also be used to learn a classifier with minimal supervision, the final layer training can be supervised



- After training a stack of **encoders** , an output layer is added on top of the stack (with softmax, as for convnets).
- The parameters of the whole system are fine-tuned to minimize the error in predicting the supervised target (e.g., class), by performing **gradient descent**.
- Everything here just like convnets

Anomaly detection is untrained



Abnormal input
Normal input
Encoder
Decoder
Abnormal input
Recon. of anomaly
Recon. of the normal
Normal input
Anomaly?
Recon. error
Encodeding of normal input
Encoding of abnormal input
Prototypical normal patterns in memory slots
Addressing memory
Significant recon. error on abnormal sample
Small recon. error on normal sample

Large reconstruction errors imply anomaly

Advantages of deep methods over «traditional» ML algorithms

- Traditionally input features to a machine learning algorithm have to be **hand-crafted**, and the result often depends on practitioners' expertise and domain knowledge to determine patterns of prior interest.
- Conversely, Deep Learning techniques learn **optimal features** (the slices, or feature maps) **directly from the data** without any guidance, allowing for the automatic discovery of latent relationships that might otherwise be not found
- **Caveat**: this is true for images, not for many other types of data, especially human-enterd that still require a lot of effor for feature extraction and engineering
- The most essential idea of Deep Learning is that of **representation**. Algorithms rely on complex, «dense» data representations that are often expressed as **compositions of other, simpler representations**

Other deep Learning issues

1. **Highly parametric**: like (and perhaps more than) for “surface” learners, parameter and feature tuning is a complex task (can easily have more than 100 million parameters (weights)!!) GTP-4 has 1.76 trillion parameters!!!
2. **Poor interpretability**: machine learning algorithms like decision trees (or itemset mining) give us crisp rules as to why it chose what it chose, while deep learners produce scores and not reveal why they have given that score. In many industrial applications, this is an issue.
3. **Need lots of training data**. For simpler and sparse input data «off the shelf» ML methods can still be better (e.g. regression/decision forests and matrix factorization methods)

How to cope with high number of parameters

- **Dropout**: consists of «turning off» neurons with a predetermined probability (e.g. 40% in the fully connected layers).
- Turning off a neuron means temporarily removing it from the network, along with all its incoming and outgoing connections
- This means that every iteration uses a different sample of the model's parameters, which forces each neuron to have more robust features that can be used with other random neurons.
- It is also a way to prevent **overfitting**
- However, dropout also increases the training time needed for the model's convergence.
- More here

How to cope with high number of hyper-parameters

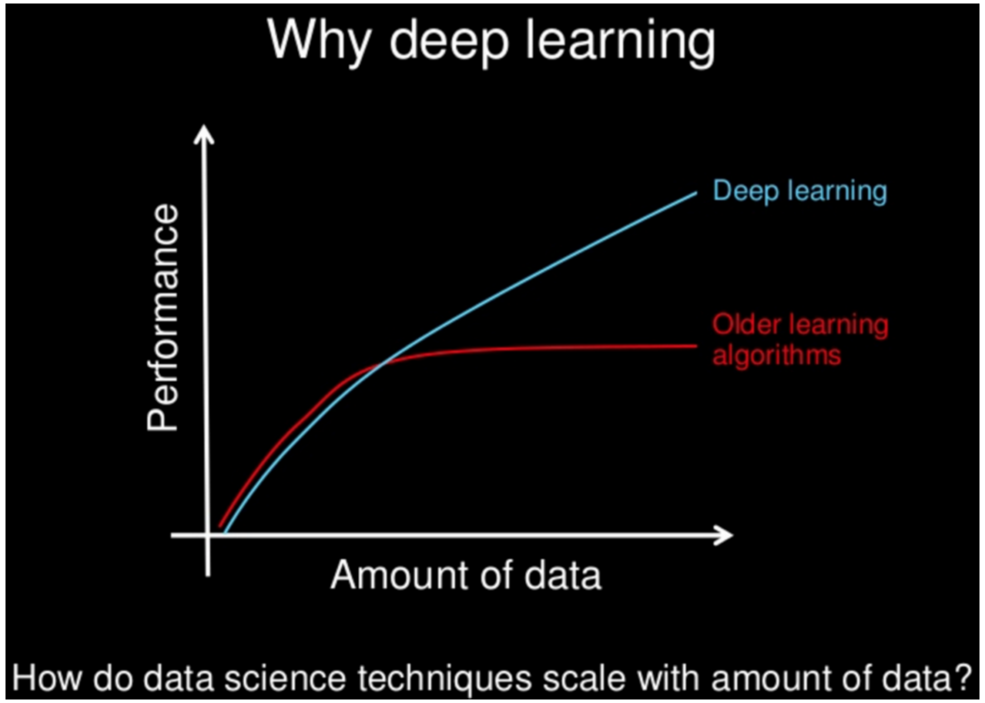
- **Random Search** uses random combinations of hyperparameters. This means that not all of the h-parameter values are tried, and instead, they are sampled with fixed (k) numbers of iterations
- **Grid Search** looks through each combination of hyperparameters. This means that every combination of specified hyperparameter values will be tried (often unfeasible).
- Both implemented in *sckit learn* and other ML platforms. Use Random Search when number of h-parameters is very high.

How to cope with limited data

3. Needs big datasets

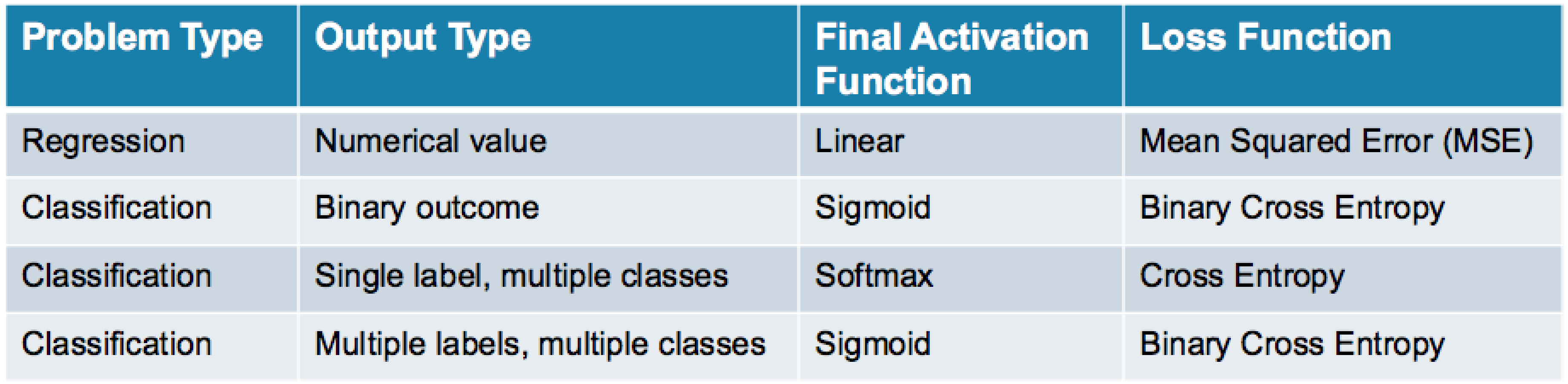
In general, with so many parameters to learn, it may perform poorly **with small datasets**

- However, in some case, it is possible to "expand" the representation of instances with data extracted from larger datasets
- See «few shot learning» as a battery of methods to «make the best» out of few examples
- Transfer learning (train on a large dataset, fine-tune on the one for which we have few examples) is also used



How to select the right
Loss and Final Activation Functions

Usually, the selection of the Loss and Activation function depends on the problem type (LINK):



Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Additional and «HOT TOPICS» in Deep Learning

- NN Architectures LINK, LINK, LINK
- How to choose the right activation and loss functions: LINK
- About the optimizers: LINK
- Overfitting: Dropout and Regularization LINK, LINK, LINK
- NN weight initialization LINK, LINK
- Curse of dimensionality LINK
- Interpretability/Explainability LINK, LINK, LINK
- Novel LOSS functions LINK
- Meta-learning LINK
- Transfer Learning LINK
- Few-shot learning LINK
- Attention Mechanisms LINK
- Other useful links LINK , LINK

Some will be expanded during lessons on ML workfolw (data engineering and model fitting)