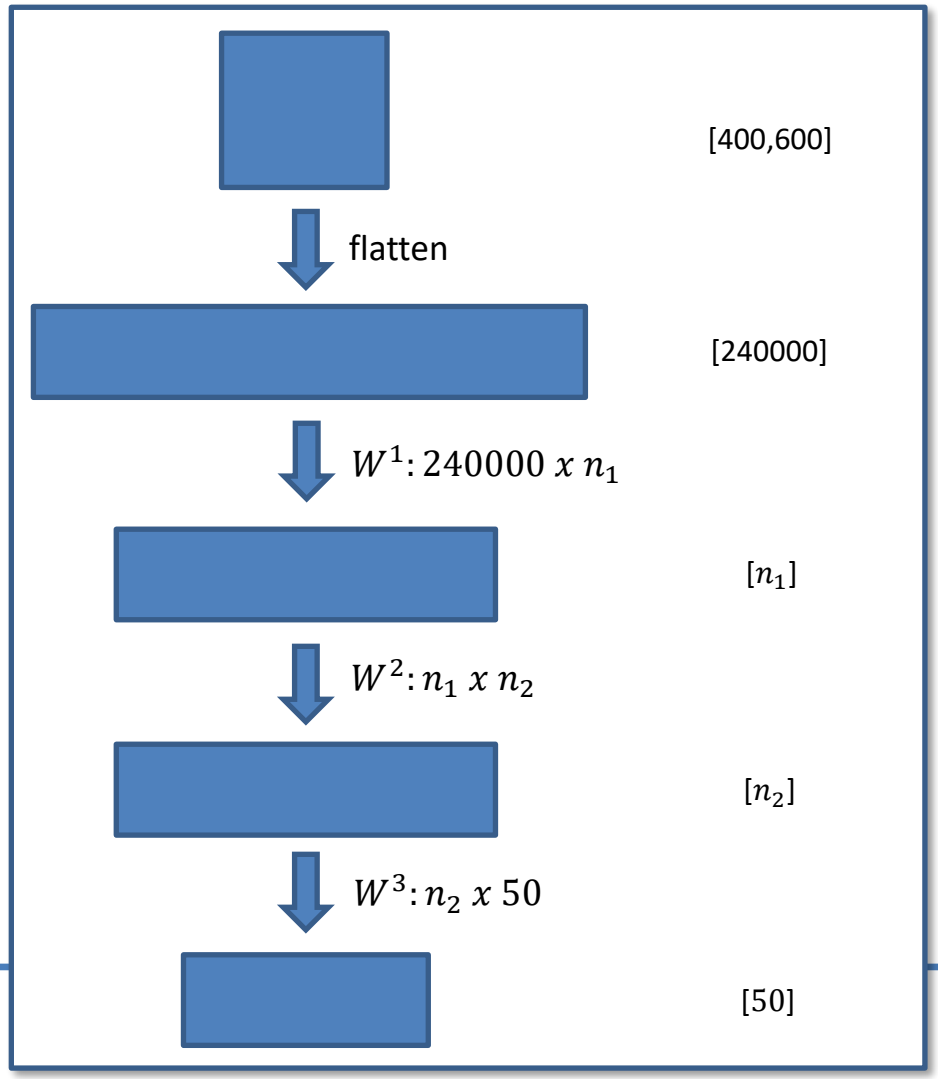
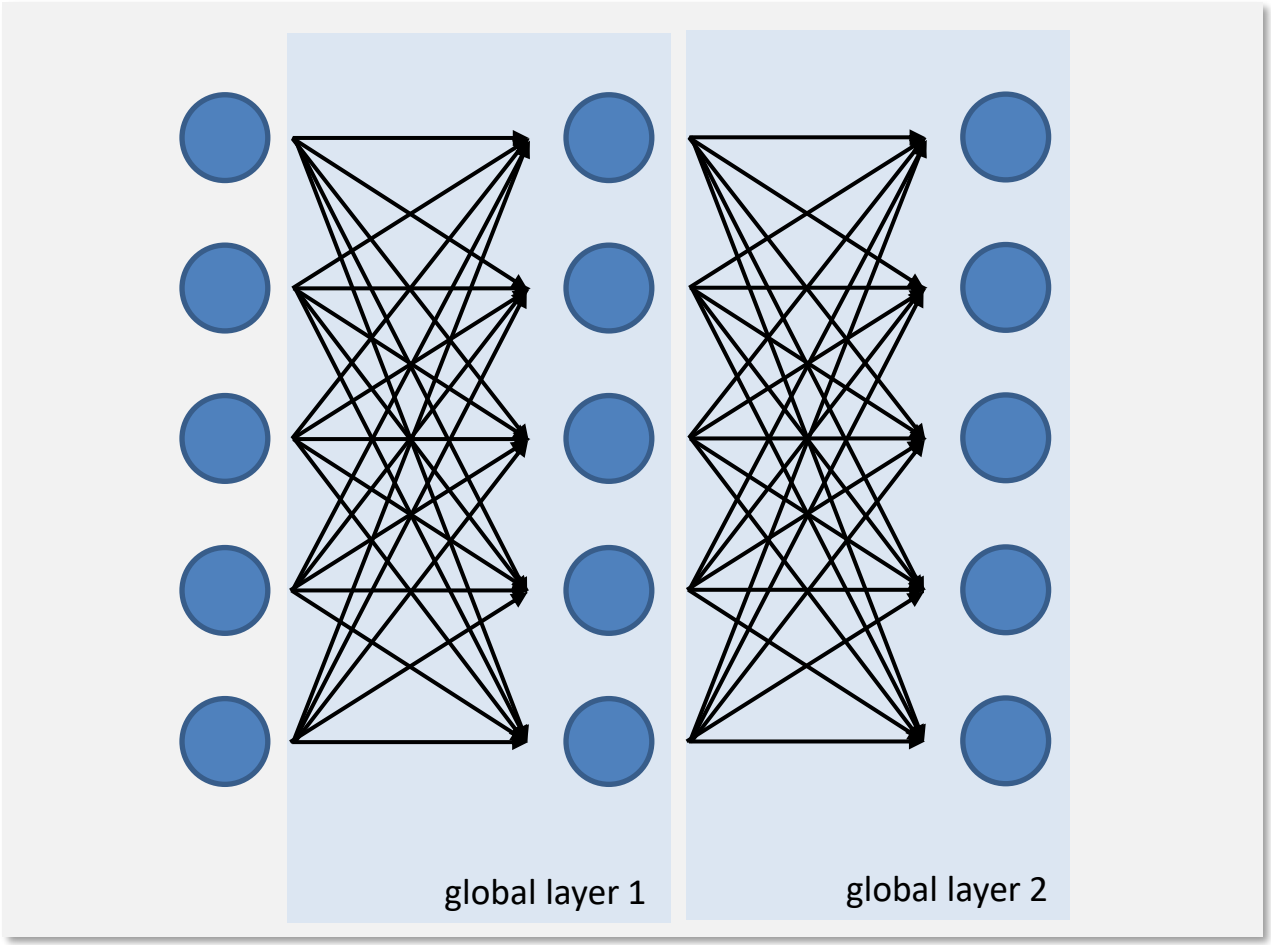
Modern image classification nets take the image pixels as input. Consider a classification task with 50 classes and a fully connected net with e.g. 2 hidden layers without bias, assume a gray-valued image (1 byte/pxl) with a moderate resolution of 400x600 pxl:



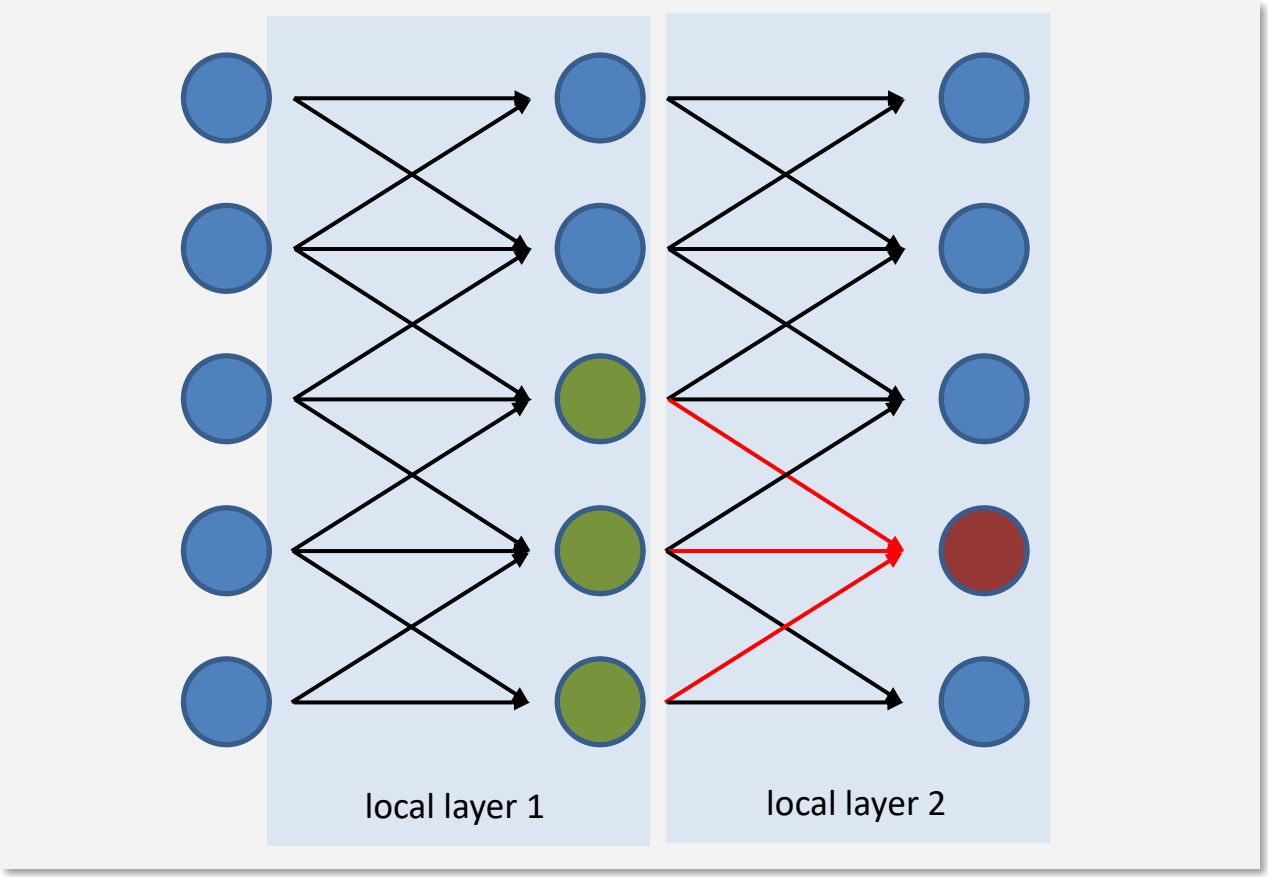
That makes 2405025000 weights for a typical setup ($n_1 = 10000, n_2 = 500$).

Billions of parameters are feasible and not that uncommon in professional systems nowadays, but is it really necessary for this task?

Layers in which the activation of every unit/neuron depends on every output of the previous layer are called ***global***. We have already encountered some global layers such as fully connected layers. Also softmax layers or a layer which outputs the sum of all inputs are global layers.

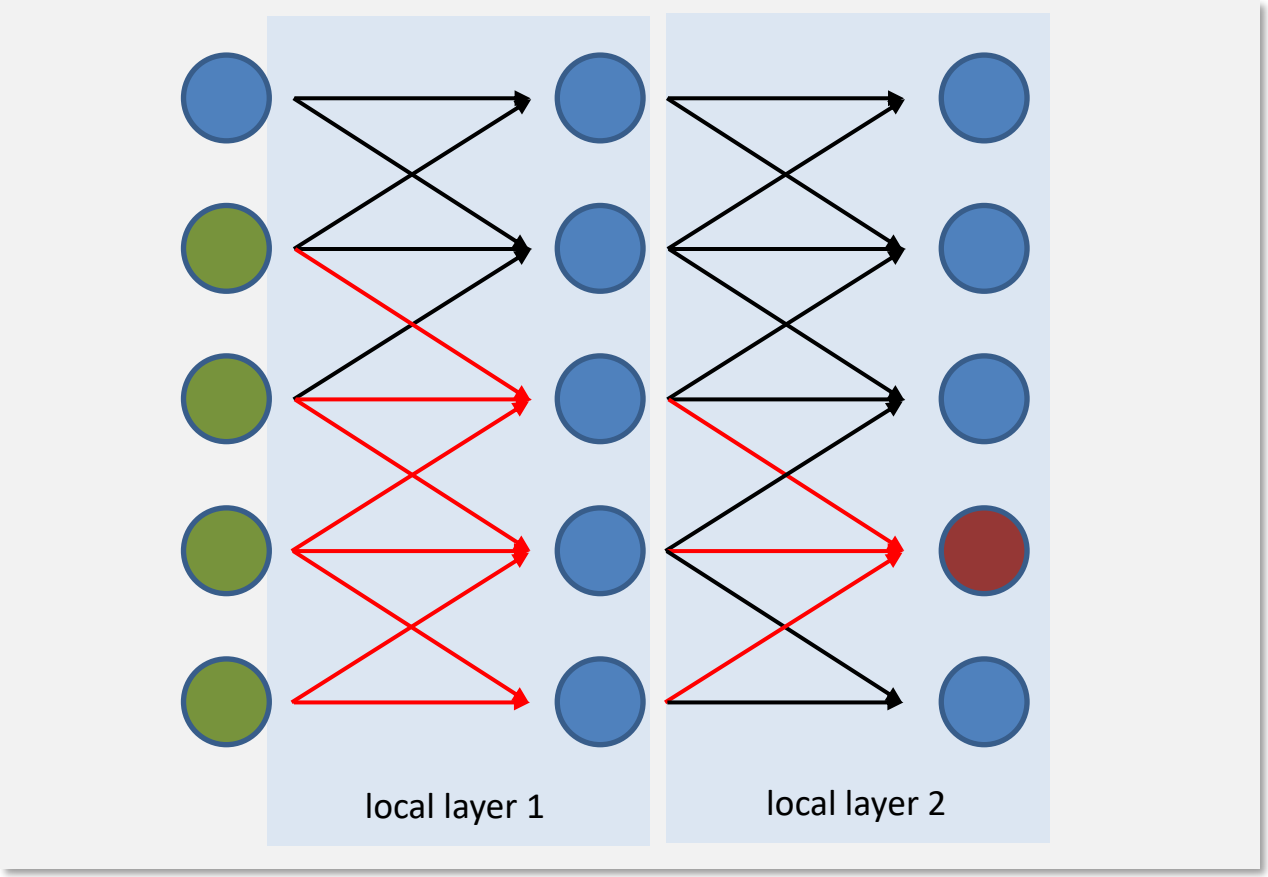


Layers in which the activation of every unit/neuron depends only on **some units/neurons of the previous layer which lie in the "vicinity"**[1] of the **target unit/neuron** are called ***local***.



[1]The term "vicinity" can be defined as appropriate.

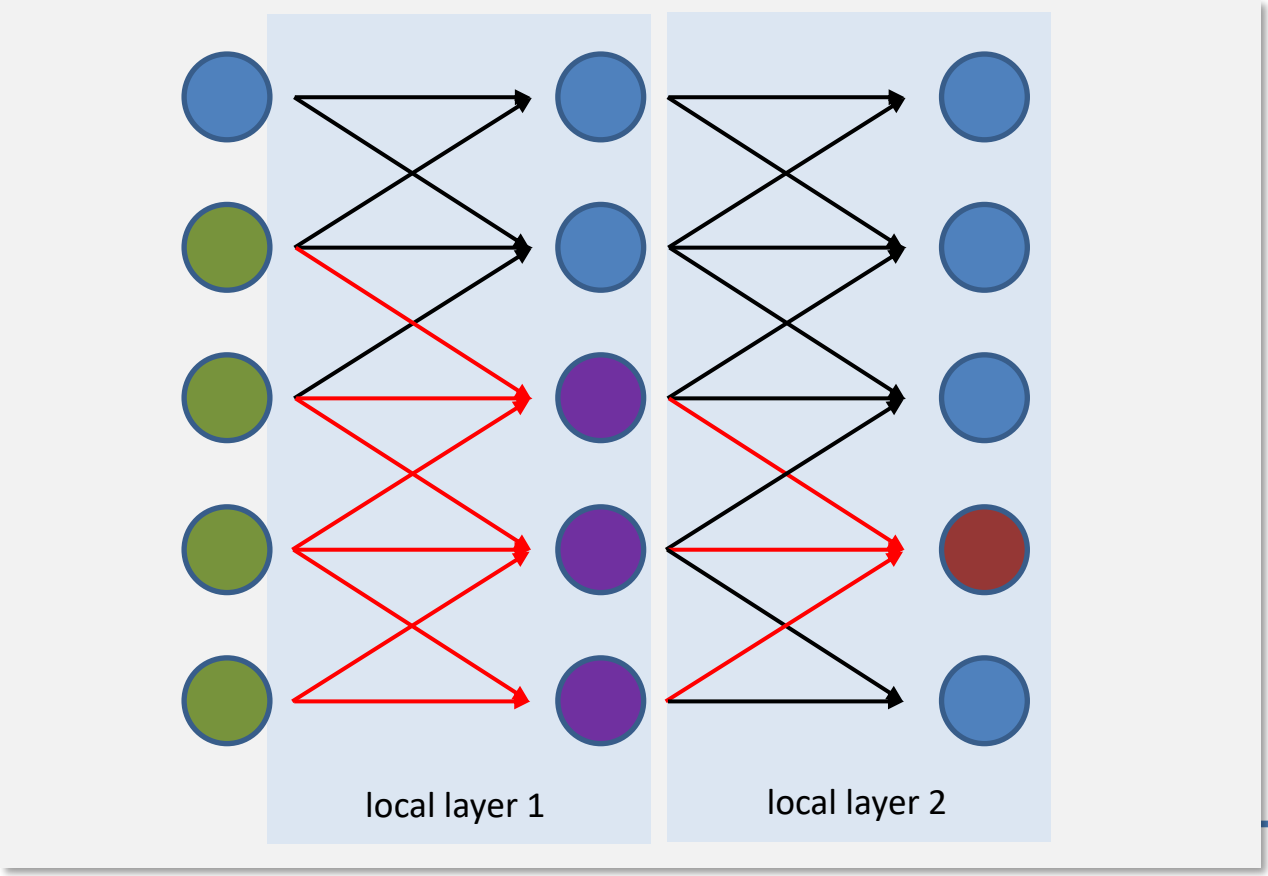
All units in some previous layer (here: in layer 0, the input layer) which influence the **target unit** are called its ***receptive field***.



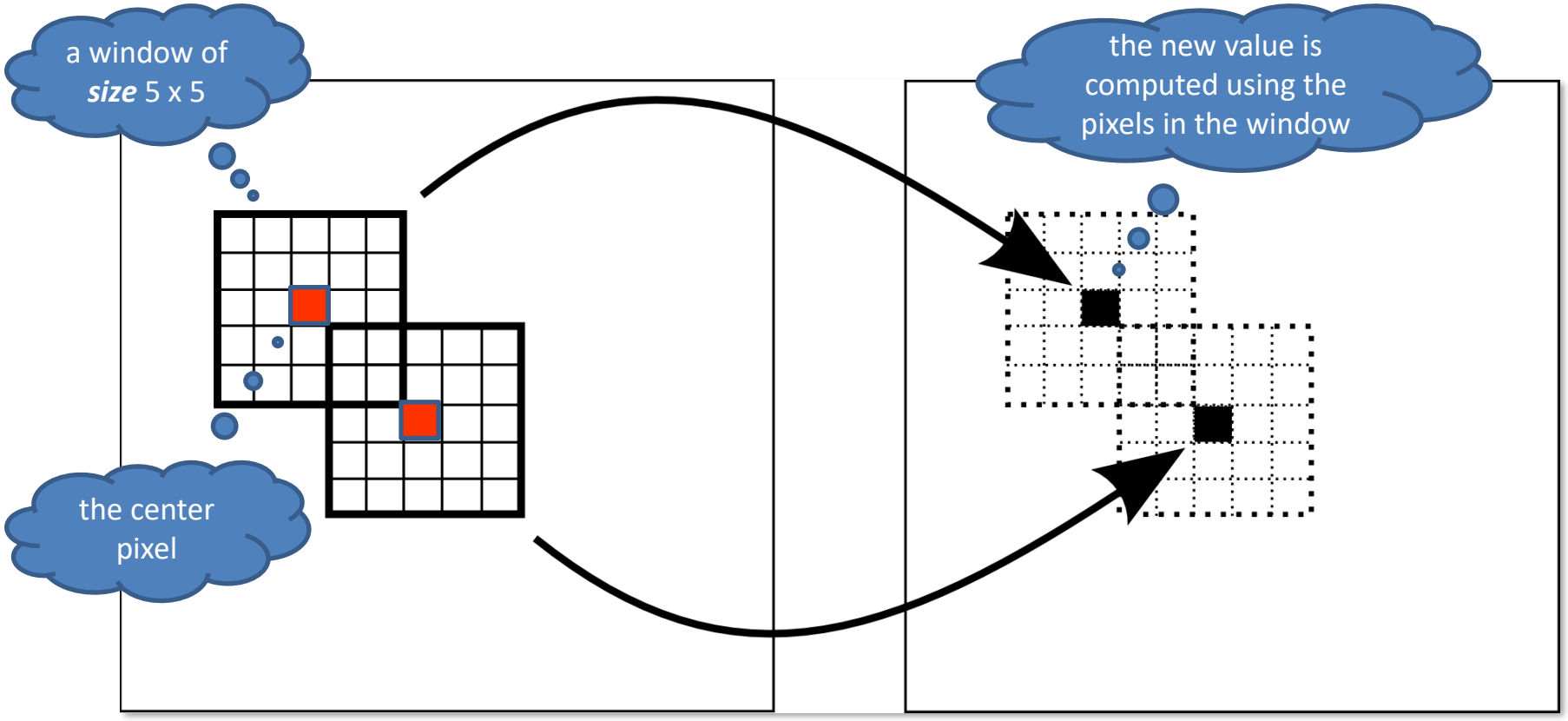
[1]The term "vicinity" can be defined as appropriate.

Local Layers and Receptive Fields

Assume a net with local layers. Observe that the receptive fields generally consist only of a portion of the layer's units. The **receptive field** grows as the distance from **target unit** to the considered layer increases. In our example, the **receptive field in layer 1** consists of 3 units (unless we reach the border). The receptive field two layers ahead is 5 units wide (unless we reach the border).



In image processing, ***window-based*** operations consider all neighbored pixels in a ***window*** around the ***center pixel*** to compute the resulting pixel. Even though a window with the same and uneven ***window size*** in x- and y-direction is most easily to draw, the window sizes may be chosen arbitrarily. (The "center" position needs to be defined somehow for even window sizes.)



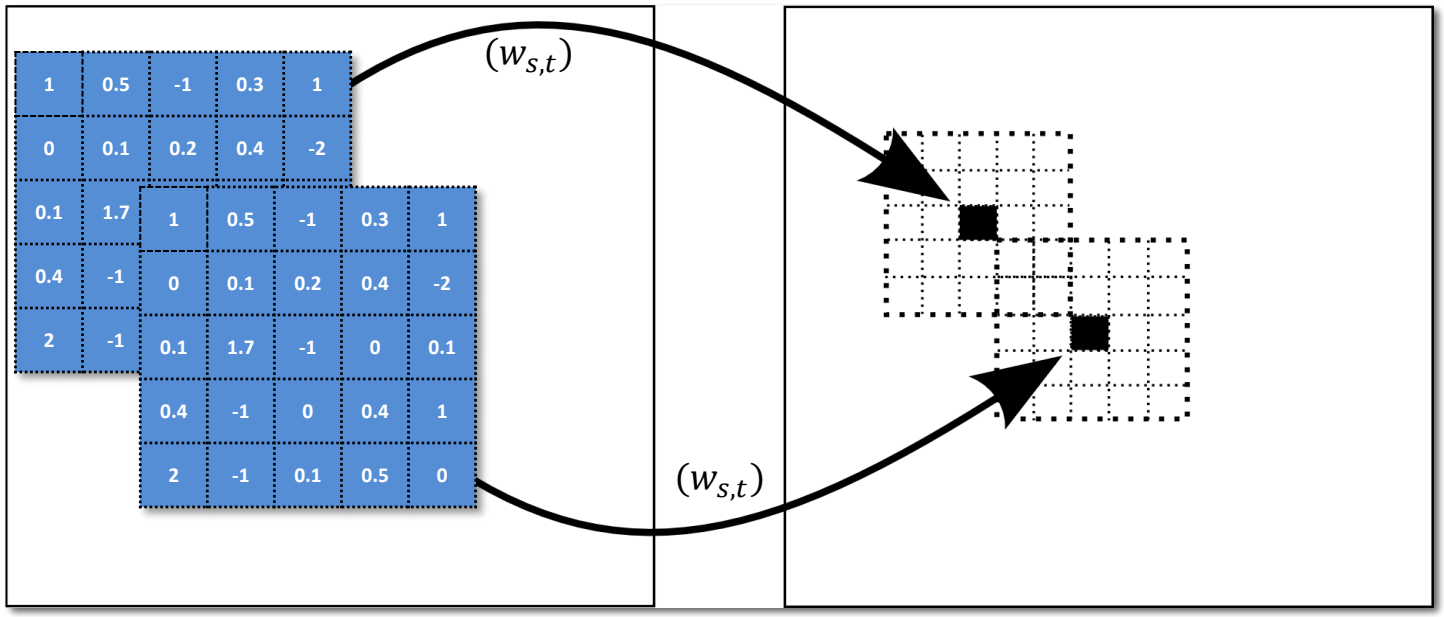
Convolutions

The simplest window-based operation is the ***two-dimensional convolution*** (more precisely: ***correlation***, also called ***linear filter***). It uses a fixed weight matrix (a ***kernel***) of the size of the window

$$W = (w_{s,t})_{\substack{s=-R_y,..,R_y,\\ t=-R_x,...,R_x}}$$

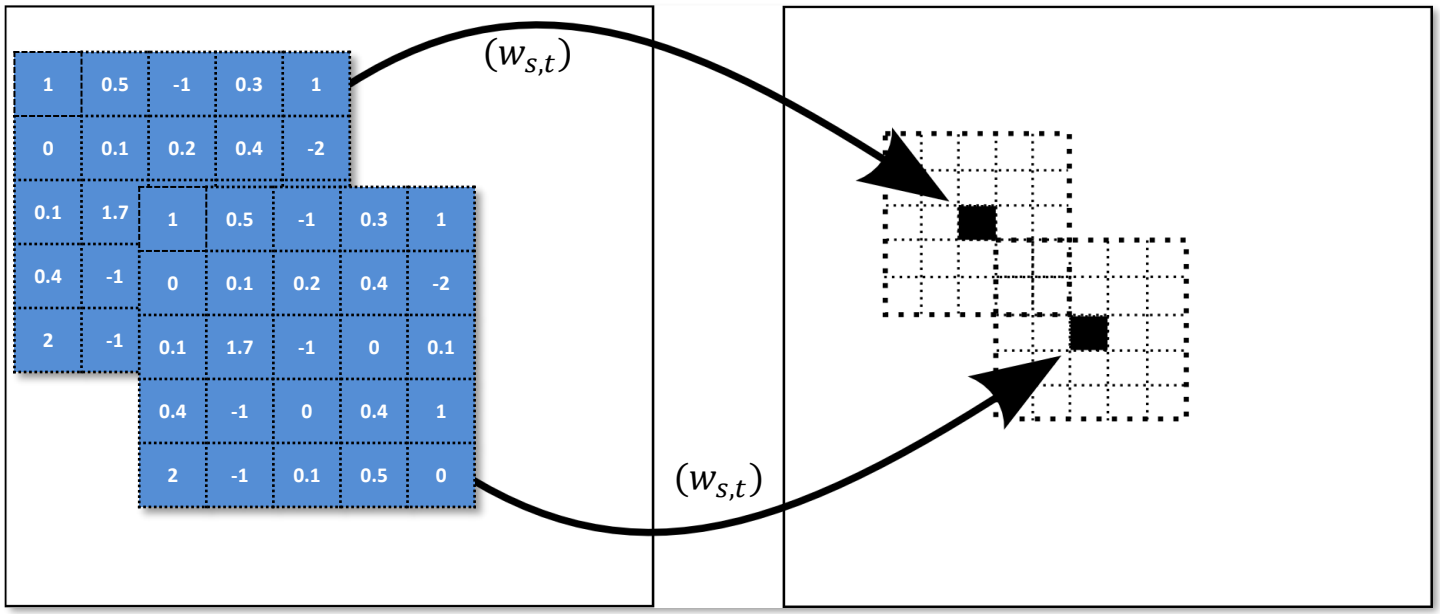
It is most natural to define centered windows with odd kernel sizes, but other configurations are possible (with other ways to index the weights)

We therefore often speak of the ***kernel size ks*** $= (2R_y + 1, 2R_x + 1)$ instead of window size.



We can visualize the convolution by filling the window with the kernel. We then slide the window over the image and compute the weighted sum of all image pixels in the current window range. The result is noted in the window center position in the feature map.

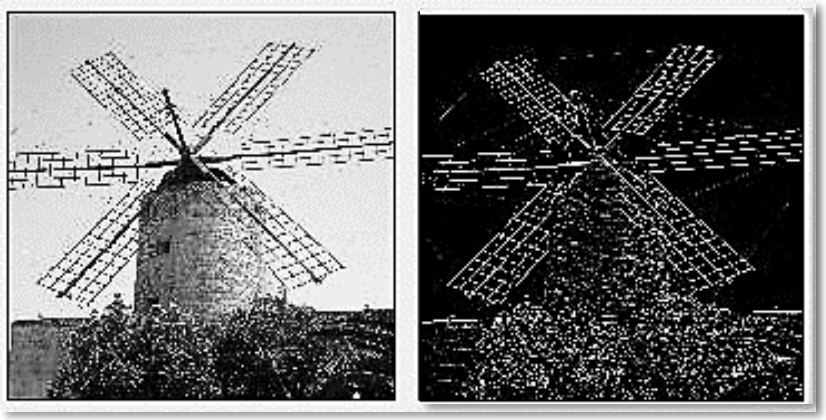
Observe that the kernel remains constant while sliding over the image!



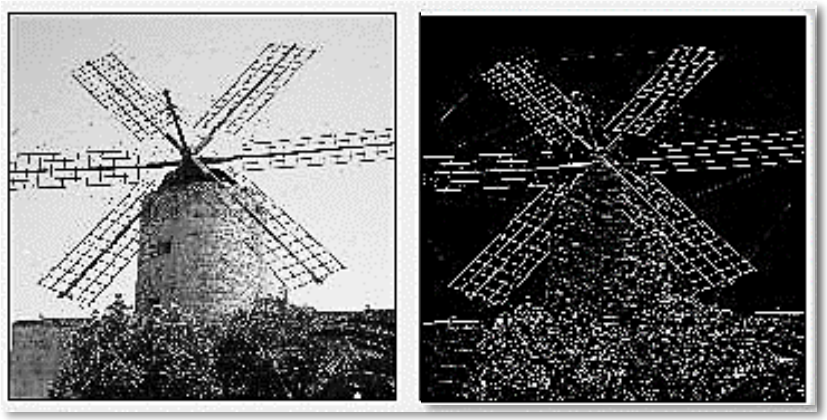
2D-Convolutions can be used to emphasize features in images. Here, the weights

$$W = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix},$$

which defines the ***Sobel-Operator,*** are convolved with the left image *(check it!).* The right image gives the absolute value of the result.



This kernel emphasizes horizontal lines and can be used for edge detection.



What is the main difference in generating a processed image by convolution between image processing algorithms and machine learning algorithms?

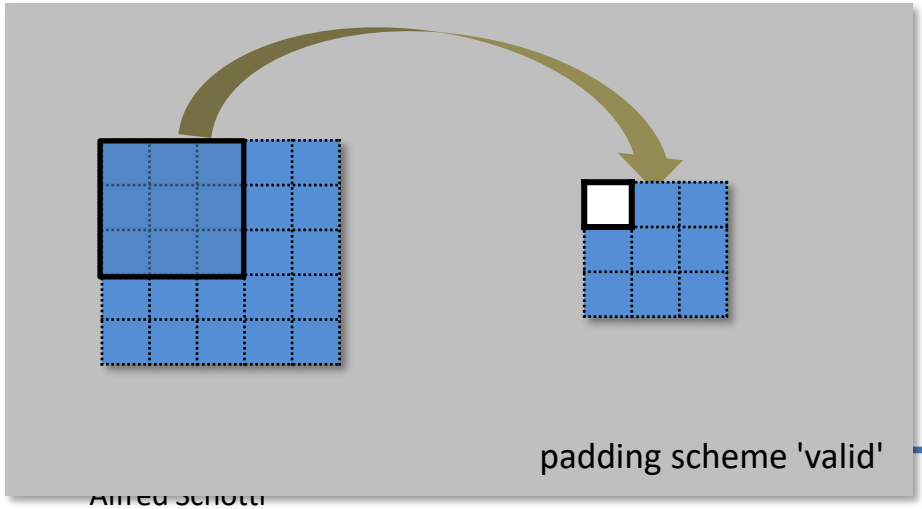
?

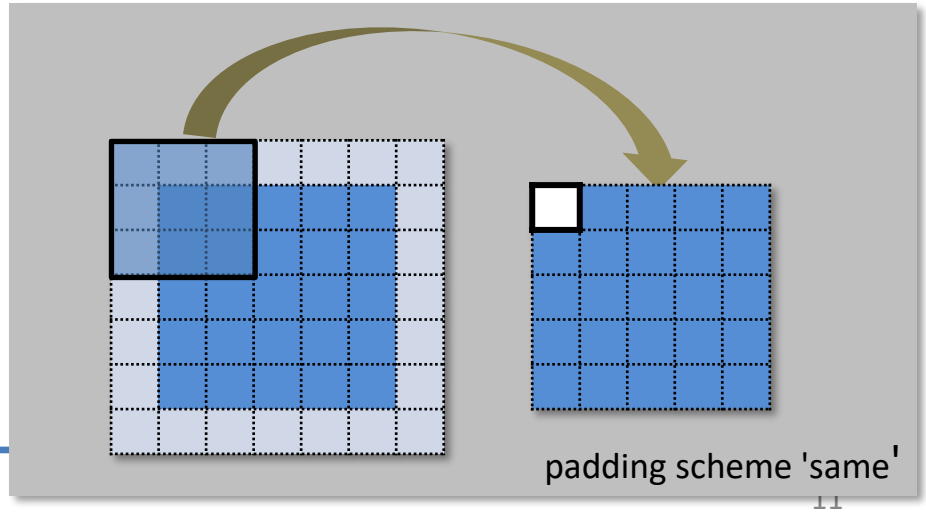
Borders:

Let $\boldsymbol{W}$ be a kernel with kernel size (ks_y, ks_x).

There are various ways to handle border pixels. Most methods ***pad*** the image (adding a "frame" of artificial pixels around the image) before convolving. Then, all *completely covered* windows are processed.

- If there is no previous expansion, the resulting image is reduced in size by $ks_x - 1$ horizontally and $ks_y - 1$ vertically. This ***padding scheme is often called 'valid'***.
- If we extend (***pad***) the original image by $ks_x - 1$ pixels horizontally and $ks_y - 1$ pixels vertically before applying the convolution, we can maintain the original image shape. This ***padding scheme is often called 'same'.***





Remarks:

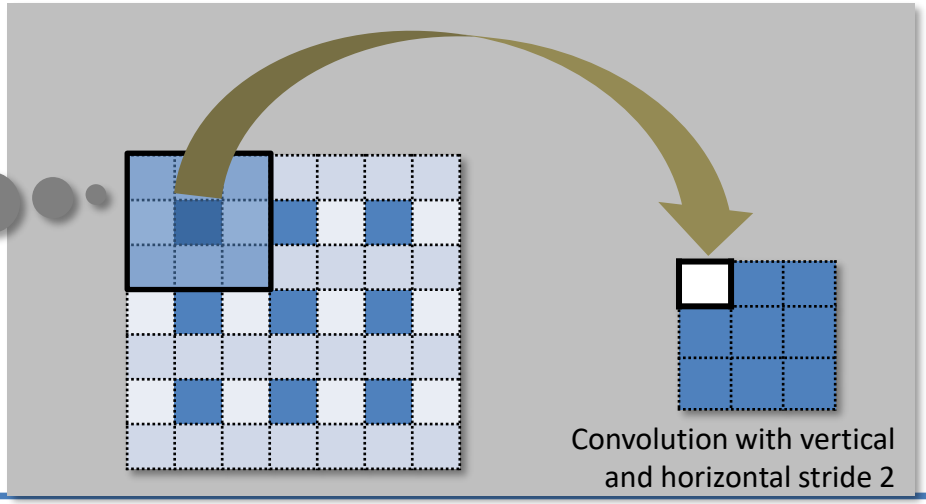
- There are numerous ways to pad the image with the padding scheme 'same'. We could add the padded pixels on the right/bottom or frame the image (which is PyTorch's behavior). We may pad with zeros or copy image pixels (this is configurable in both tools).
- Window-based operators always suffer from the problem of *how* to handle the border pixels. There is no preferred solution (easier to understand in models is padding scheme 'same').
- It is sometimes *a desired feature* in convolutional nets that the image size reduces by the convolution.

Convolution V

Remarks II:

- Things which are possible with convolutions are edge detection, smoothing, low/high-pass filtering, gradients in one specific direction etc.
- An elegant method to reduce the size of the resulting image is to shift the center pixel by more than one pixel. This shift is often called ***stride****.
- In PyTorch, you cannot specify the padding schemes 'valid' and 'same' if the stride is different from 1. In this case, you have to explicitly define the number of padded pixels on each side (which is $\frac{ks_x-1}{2}, \frac{ks_y-1}{2}$ for odd kernel sizes and padding scheme 'same').

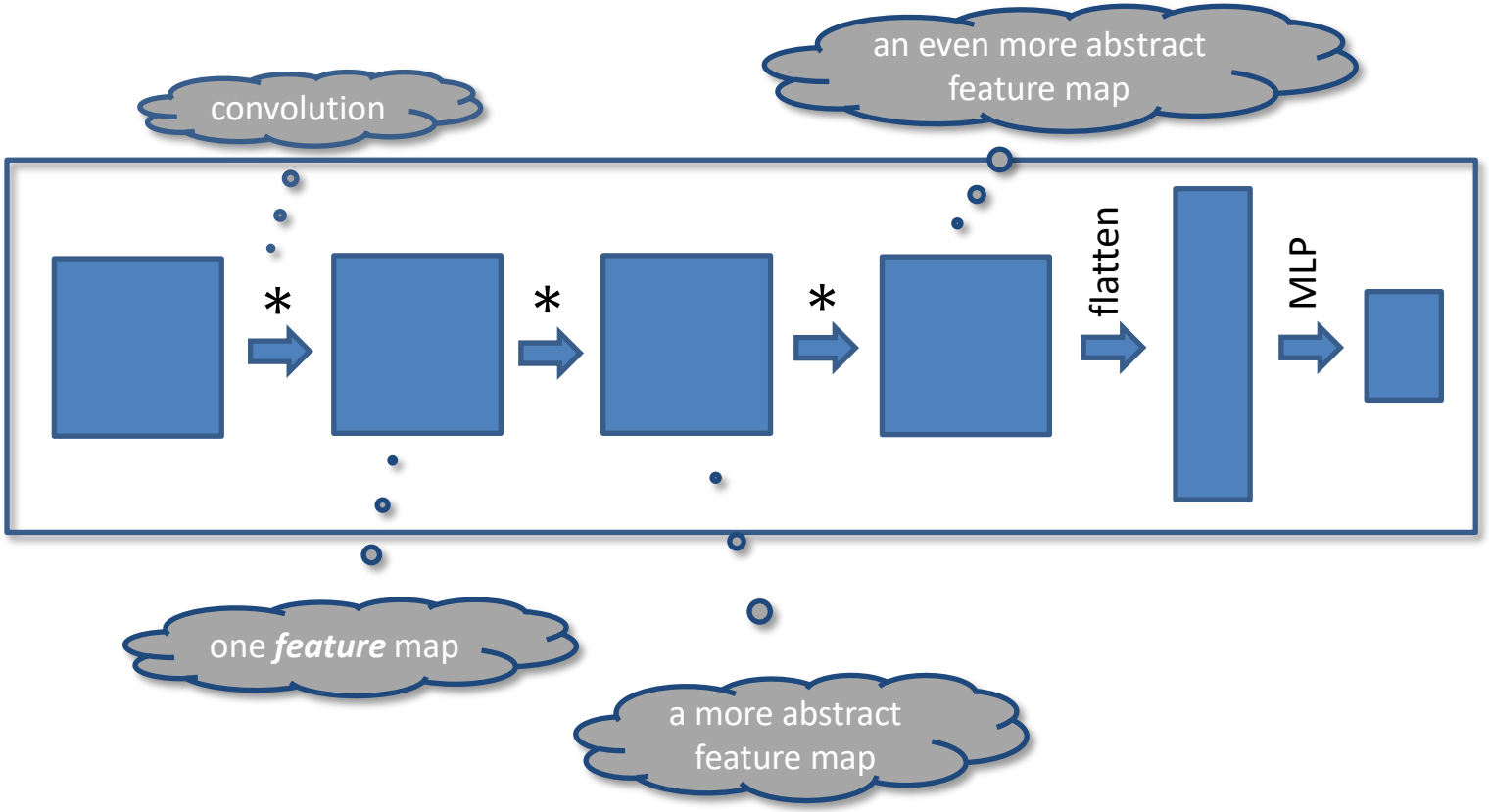
Roughly, a stride>1 reduces the map size by this factor. The exact new extension depends on the padding mode and the tool. We will simply calculate with this factor.



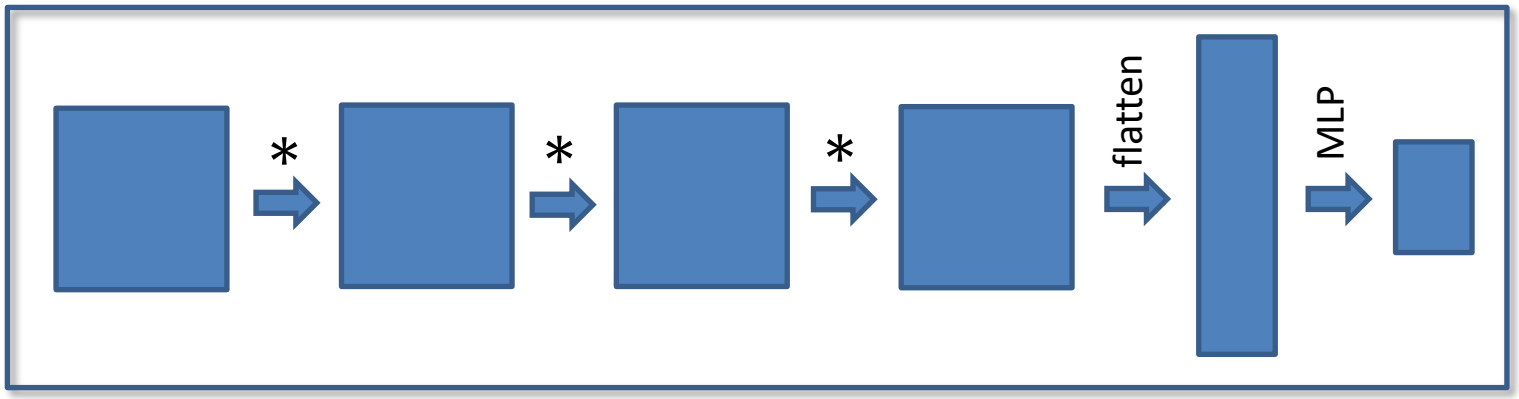
Convolution with vertical and horizontal stride 2

*In image processing, stride is defined differently.

Classical convolutional nets stack several convolutions. This gives us gradually more abstract feature maps of the original image. The resulting feature map is then flattened and used as input to a small classification net (MLP), e. g. a fully connected net with 1-2 layers.

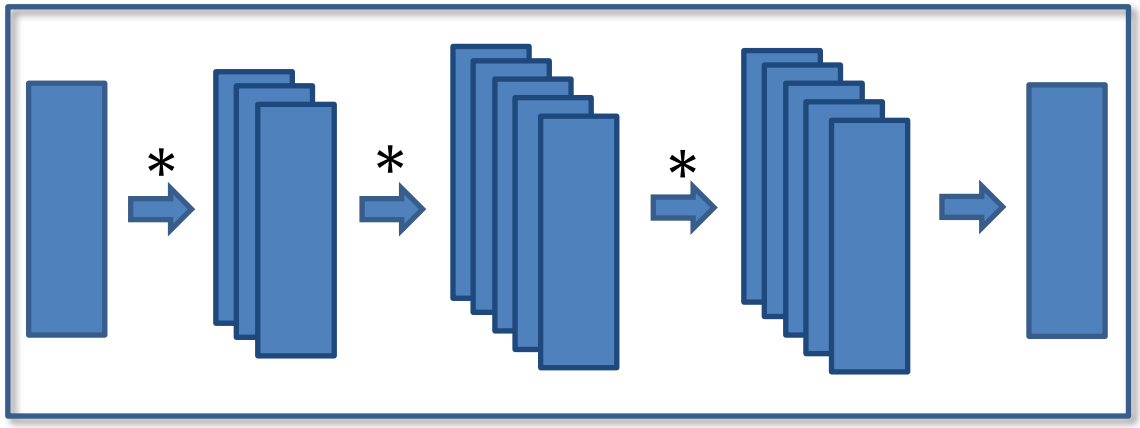


Consider a classification task with 50 classes and a fully connected net with 1-2 layers without bias, assume a gray-valued image (1 byte/pxl) with a moderate resolution of 400x600 pxl:



That makes ridiculous 75 weights for the first 3 layers in a typical setup with 5x5 kernels!
It however needs almost 240000 x 50 weights for one fully connected classification layer.

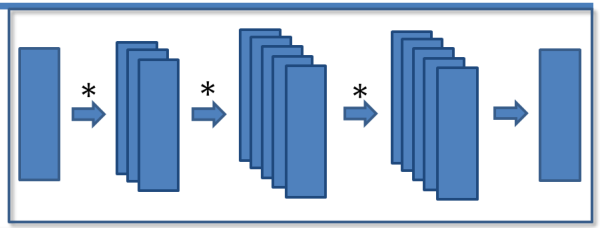
One feature map per layer only stores the result of one filter. We need, however a ***filter bank*** for a good performance. It is e. g. typical that edge detectors for different directions are needed. A filter bank leads to multiple ***feature maps*** or ***channels*** per layer.



This picture does not specify how the feature maps in the next layer are exactly generated from the maps in the former layer. See next slide.

We will now have a lot of convolutions with different kernels. Let $W^{p,q,k}$ be the kernel which is applied to channel p of layer $k-1$. The convolution result contributes to channel q of layer k.

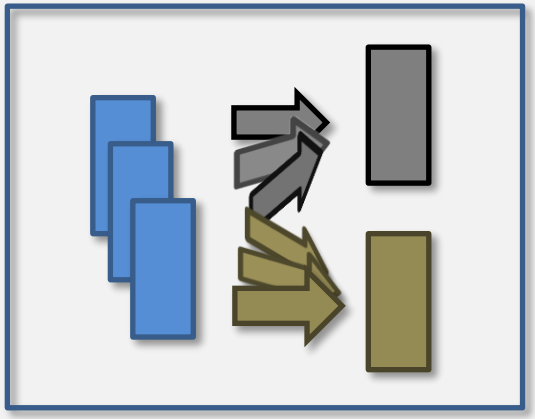
Let $\boldsymbol{z}^{p,k}$ be the pth channel in layer k.



The most important convolution operation with several channels is the ***mixing convolution***:

$$\boldsymbol{z}^{q,k} = \sum_{p} W^{p,q,k} \star \boldsymbol{z}^{p,k-1}$$

Convolve each channel of the previous layer with a different set of weights and combine all of them (by summing up). Repeat this for all desired output channels with separate sets of weights.



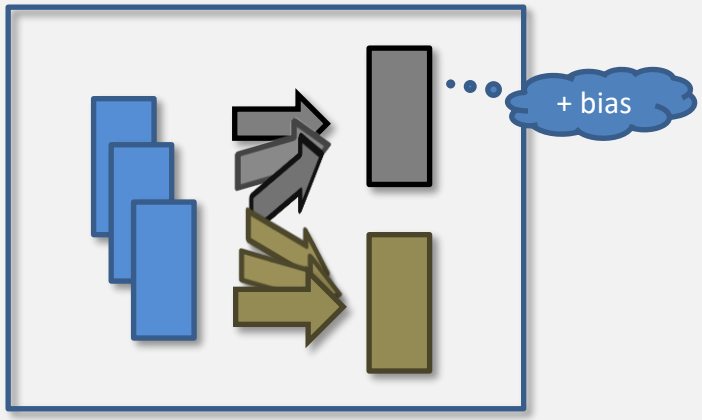
We need a 4-dimensional weight tensor $\boldsymbol{W^k}$ to specify the operation (if we fix the layer k)! This means that $W_{i,j}^{p,q,k}$ is the weight belonging to the output channel q of layer k, associated with the kernel position i, j used for the convolution of channel p.

Let $\boldsymbol{z}^{p,k}$ be the pth channel in layer k.

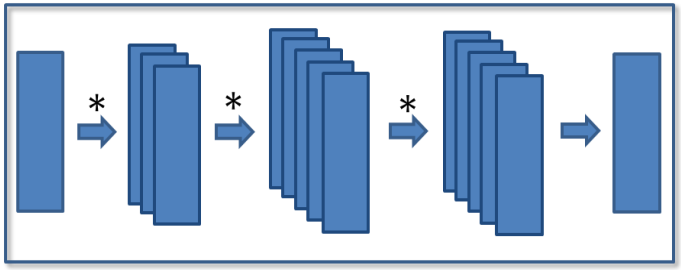
The convolutions are often complemented with biases (which are often not explicitly mentioned):

$$\boldsymbol{z}^{q,k} = \sum_{p} W^{p,q,k} \star \boldsymbol{z}^{p,k-1} + b^{q,k}$$

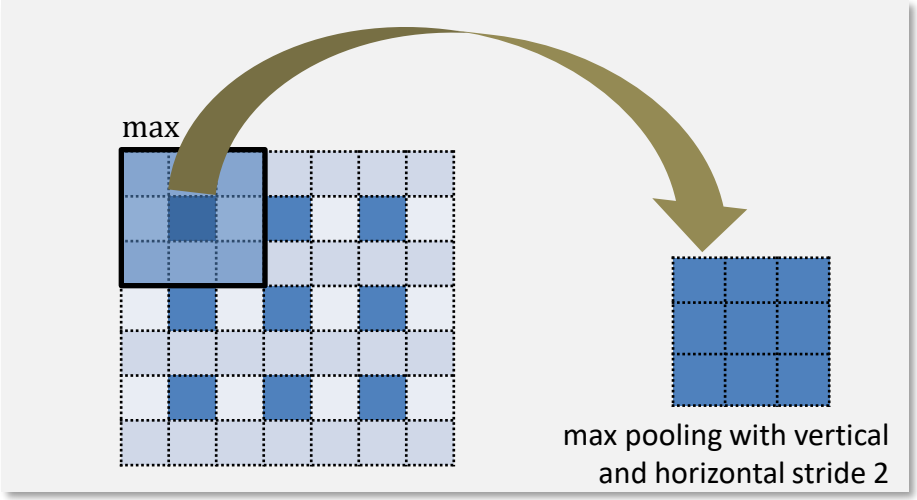
Observe that there is only one bias $b^{q,k}$ (one number) per output feature map.



A second operation which is important for convolutional nets is the gradual reduction of the map size. The reduction not only serves as means of saving parameters, it also forces the net to find more global representations from layer to layer. We already introduced size reduction by strides during convolution.



An alternative is ***pooling***. Pooling is also a window-based operation typically using strides greater than 1. It takes the maximum (***max-pooling***) or the average (***average-pooling***) of all elements of the window. It is typically used per channel but can also be extended to a pooling over channel barriers. Standard is max-pooling.

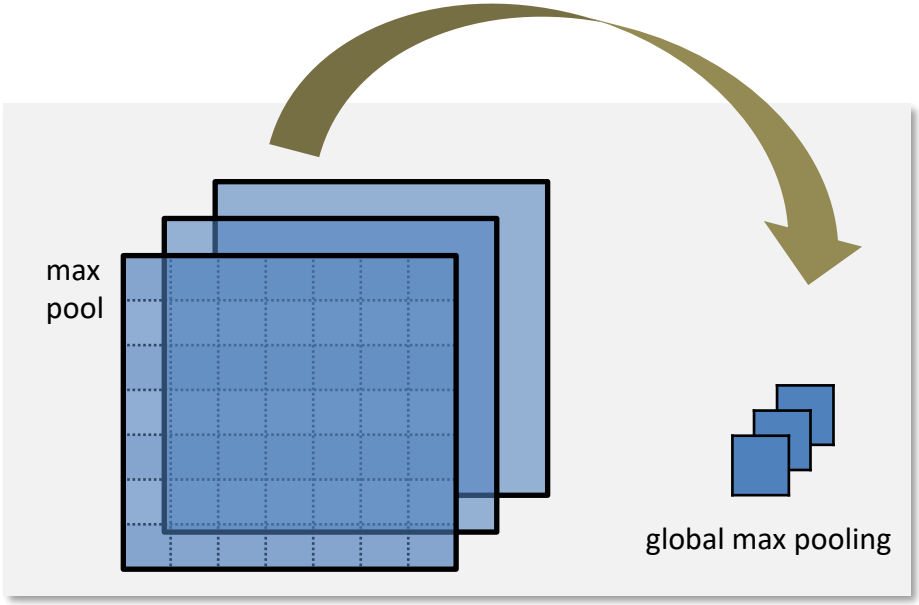


Max pooling not only reduces the size, it also introduces a slight ***translation invariance***. The invariance can be extended by stacking over several layers.

Pooling may be interpreted as feature extraction step.

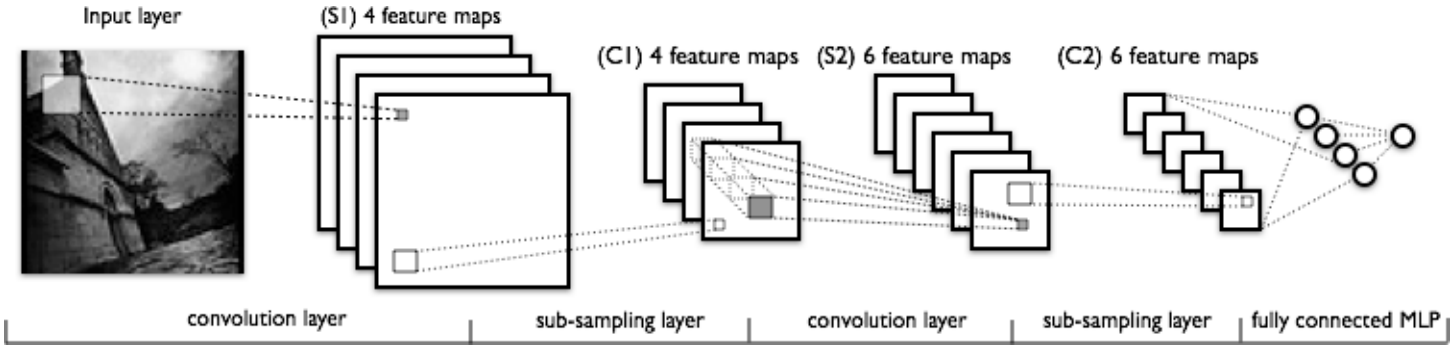
The pooling operations can be made global by using a window size which is big enough. The result of a global pooling operation is just one number per channel. Global average pooling with a subsequent softmax layer is used as an alternative for the classification part of the convolutional net. Note that the number of channels in the last convolutional layer has to correspond to the number of classes in this case.

Observe that these nets do no longer need any dense/fully connected layer!



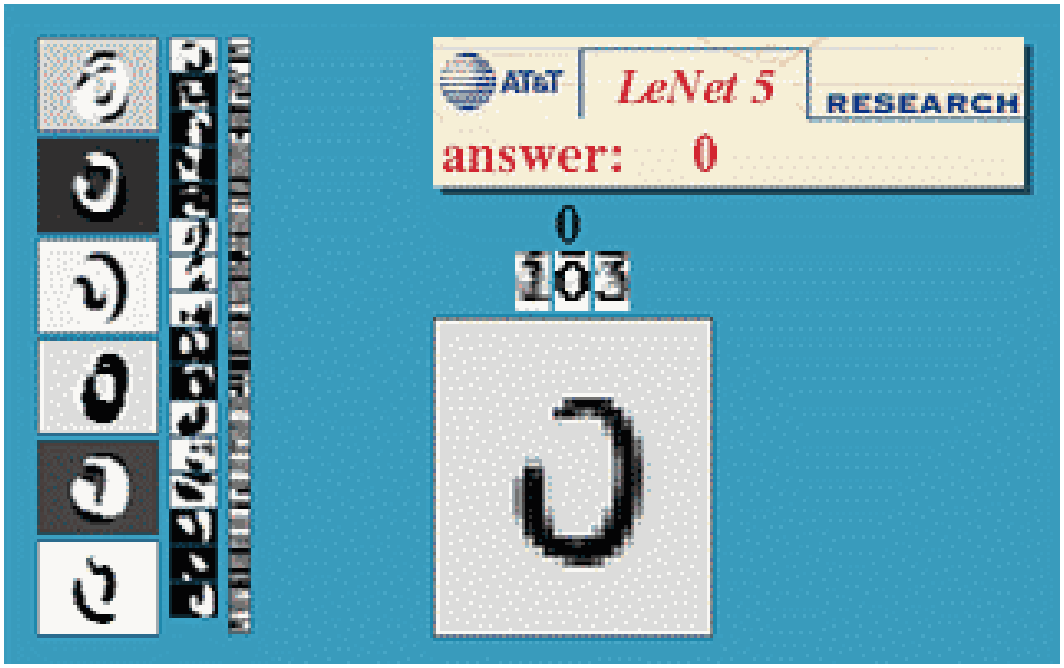
Putting all together

A typical (simple and of historical importance) convolutional net for image classification has the following structure. This family of nets was introduced by Yann LeCun and is therefore dubbed ***LeNet***.



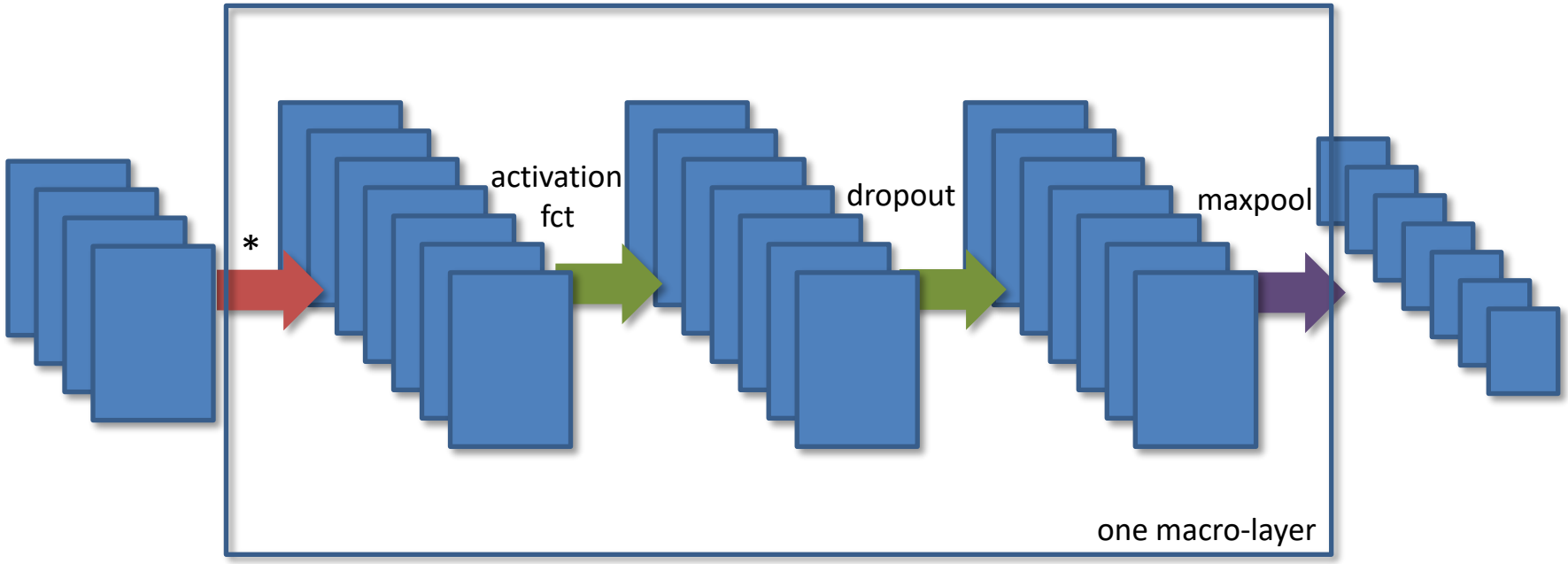
The nets consists of an alternating sequence of convolution and sub-sampling (most often ***max-pooling***) layers. The map size is significantly reduced over the layers, no activation functions are applied in this part. The last two layers are fully connected layers (with sigmoid and softmax activation) bringing together all features detected in the last feature maps.

A typical (simple) convolutional net for image classification has the following structure. This family of nets was introduced by Yann LeCun 1989 for reading ZIP codes and is therefore dubbed ***LeNet***.



©Yann LeCun, *yann.lecun.com/exdb/lenet*

Modern (big) CNNs include additional layers per convolution to perform non-linearities (activation functions) and regularization operations such as ***dropout*** (see later). Such a combined structure of basic layers can be seen as a "macro layer".



A good rule of thumb is to increase the number of channels by a similar rate as the feature sizes reduce. For classification task, we typically flatten the convolutional result and append one or two dense layers.