50.039 – Theory and Practice of Deep Learning

Alex

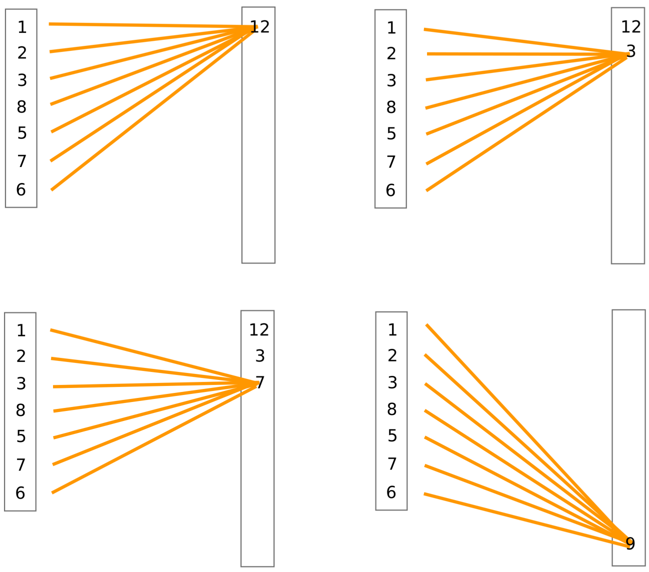
Week 03: Convolutional Neural networks

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

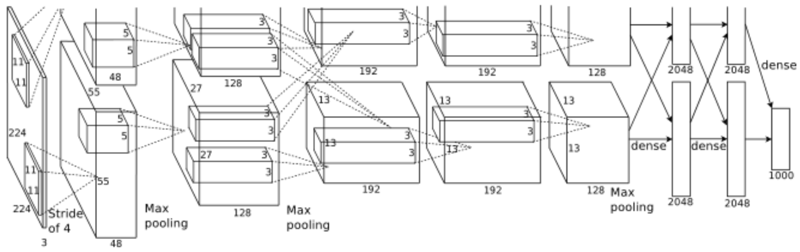
1 Convolutional Neural networks

See also for example: `http://neuralnetworksanddeeplearning.com/chap6.html`.

1.1 a fully connected layer, 1 input channel



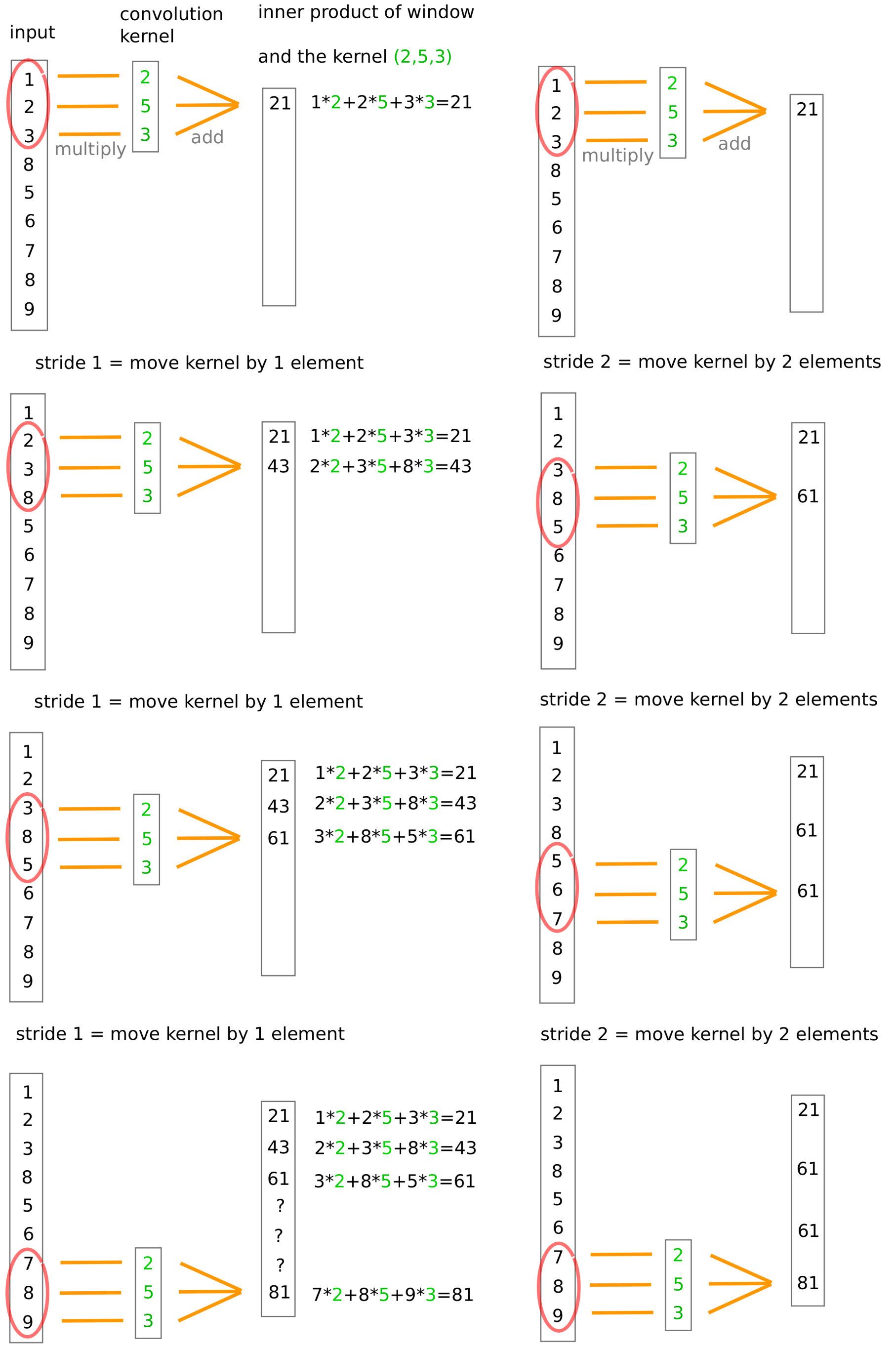
number of weights grows with the number of elements in the input and the output



taken from: Alex Krizhevsky et al., NIPS2012 `http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`

- how to connect neurons sparsely?
- **key idea**: in images neighbor pixels tend to be related! So we connect only neighboring neurons in the input.

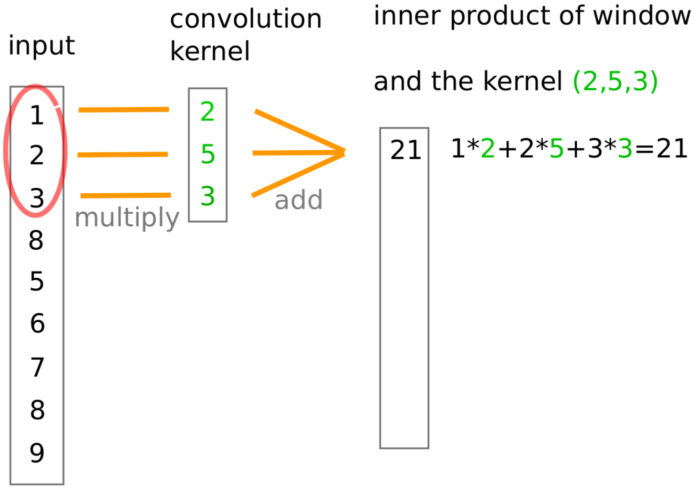
1.2 1-d convolutions, 1 input channel



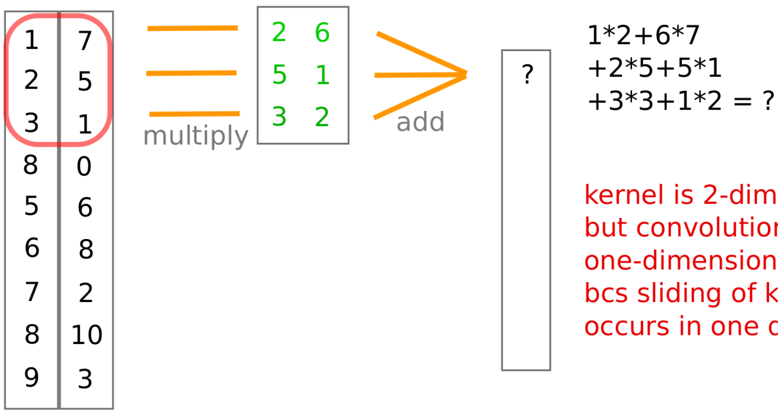
no padding of input:
outputsize = inputsize -2 = inputsize -(kernelsize-1)

no padding of input:
outputsize = ceil ((inputsize -2)/2)
= ceil((inputsize -(kernelsize-1))/stride)

1.3 1-d convolutions, 2+ input channels

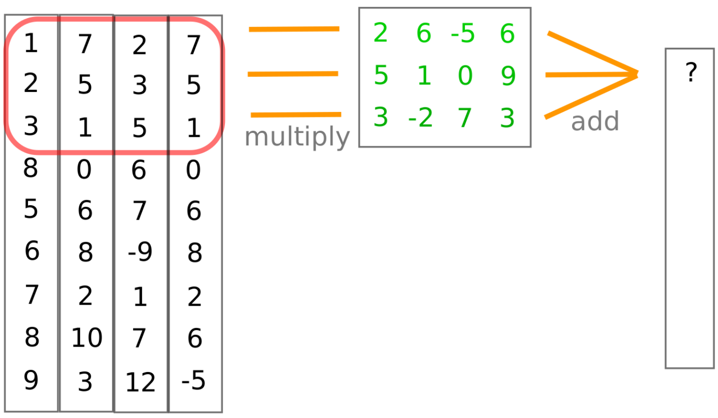


2 input channels, kernel is of size (nchannels, kernel size) = (2,3)

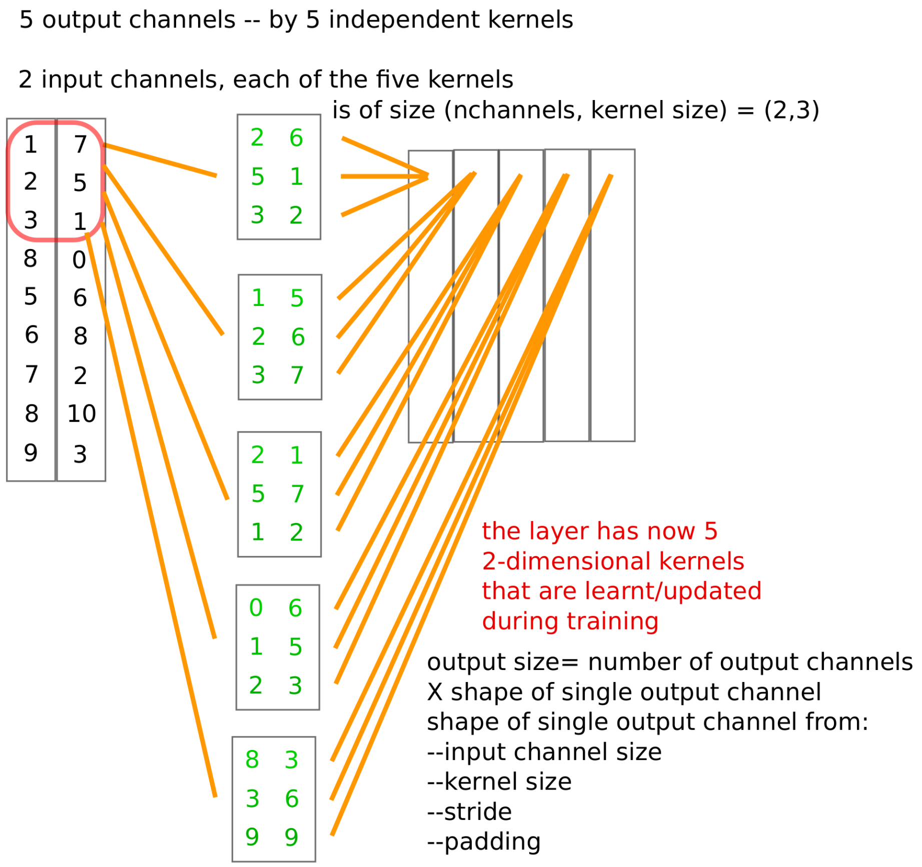


kernel is 2-dim object,
but convolution is
one-dimensional
bcs sliding of kernel
occurs in one dimension

4 input channels, kernel is of size (nchannels, kernel size) = (4,3)



1.4 1-d convolutions, one whole convolution layer (multiple output channels)



1.5 why convolutions I?

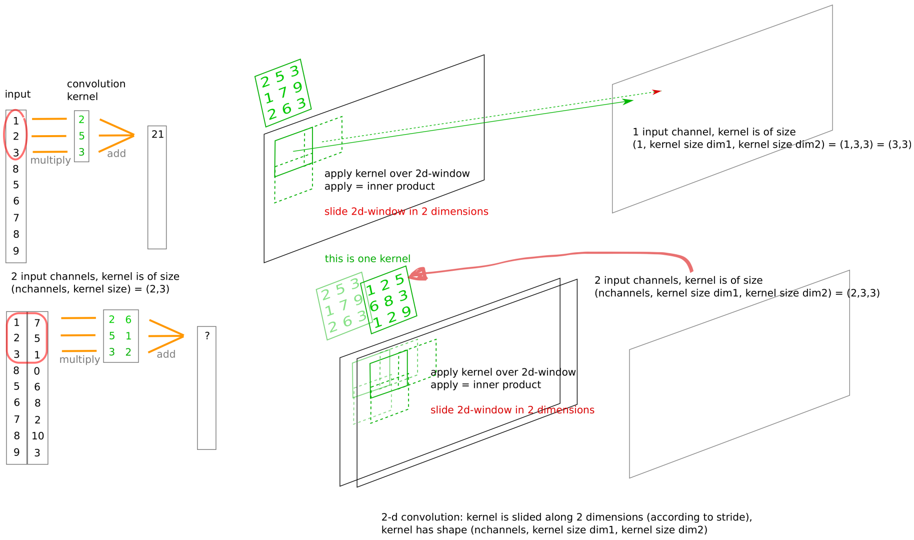
The neural net example code for mnist has linear (fully connected) layers. In it: each neuron of layer l is connected to each neuron of layer $l+1$.

- Fully connected: Assume we have N_l neurons in layer l and N_{l+1} neurons in layer $l+1$: parameters have dimensionality =?
- Locally connected: Assume we have N_l neurons in layer l and N_{l+1} neurons in layer $l+1$, each output neuron takes input from a patch of 5 neighbors: parameters have dimensionality ?
- 1-d convolution with kernel size 5: parameters have dimensionality 5, no matter how many inputs or outputs in the sliding dimension
- convolutions have a small parameter dimensionality, independent of input and output size in the sliding dimension (number of output channels matters though)

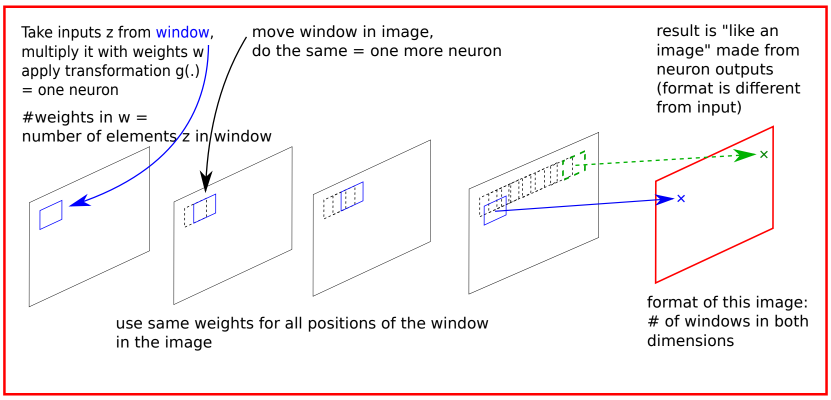
The philosophy:

- machine learning in general: keep number of parameters limited relative to training set size
- deep learning: better stack simple functions in deeply in many layers than learning in one layer something very complex.

1.6 2-d convolutions, 2+ input channels



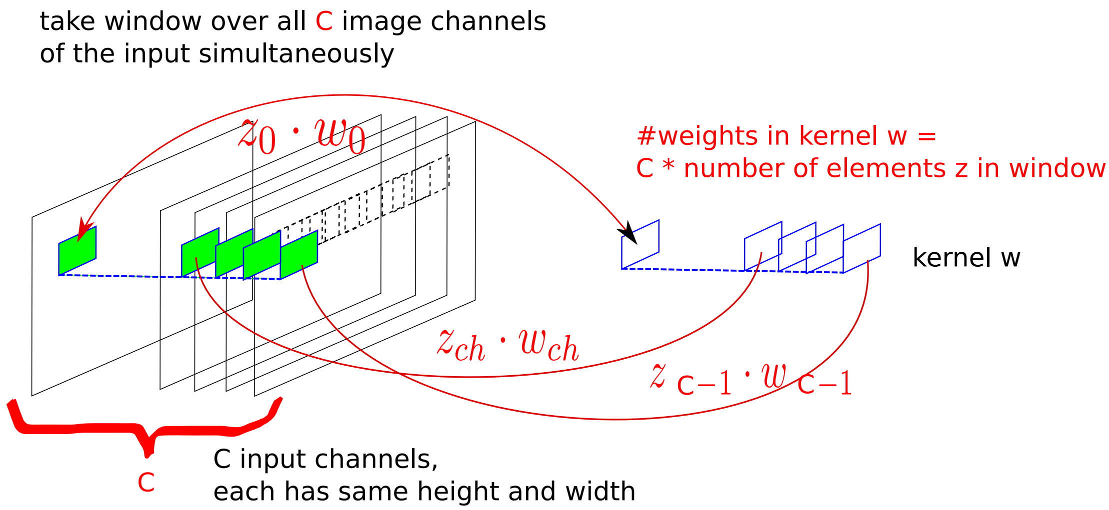
A simplified convolution (one input channel only) can be drawn like this:



- a convolution at one fixed region (region means here: rectangular window) x of an image is an inner product $w \cdot x$ between kernel weights w and this region x - this is a single real number.
- we have applied the matrix in one point, resulting it in one real number as output. now can slide the kernel w along both axes (height and widths) - results in a matrix of outputs

- This means: we slide the convolutional kernel w over the image and apply the inner product over many windows. **In convolution we apply the same w across all locations** for computing inner products.
- the parameters to be learned during training are the values of the kernel w!

can apply to images with multiple input channels - conv kernel has one depth dimension more than sliding dimensions, result still one channel image. A less simplified convolution (multiple input channels, one output channel) can be drawn like this:



$$output(h, w) = z_0 w_0 + \ldots + z_{ch} w_{ch} + \ldots + z_{C-1} w_{C-1}$$

sum of C inner products -- one for each input channel

Above shows convolution for a convolution kernel with 1 output channels. Finally, one convolution layer usually uses O independent convolutional kernels, resulting in O channels as output.

1.7 why convolutions II?

Convolution implements a battery of localized pattern detectors.

- kernel matrix is some pattern (Krizhevsky paper, Zeiler paper)
- apply convolution, get $y = kernel \cdot inputwindow + b$

  inner product is a similarity measure, high positive for inputs parallel to kernel, zero for inputs orthogonal, high negative for inputs antiparallel to kernel
- apply convolution with activation function $g(\cdot)$ in the next layer – result for one window has formula $g(kernel \cdot inputwindow + b)$, detector for patterns in input channels similar to the kernel

- detection is performed all over across the sliding space (1-d,2-d,3-d,n-d)

1.8 why convolutions III?

- Think of the image in the above graphic not as an image, but as a grid of signals of detectors. Why does that makes sense?

  An RGB-input image itself is a signal of detectors (camera sensors)

  $w \cdot inputwindow$ is a similarity between these two. A neuron output $g(w \cdot inputwindow + b)$ can be seen as a detector for some kind of structure encoded by w. It gives a high signal for some inputs, and a low signal for other inputs.

  Then every input pixel in above graphic is a signal of some detector for a kind of part/structure, e.g. a car wheel, a car window.

  A convolution can be seen as weighted sum of inputs $conv(pos1, pos2) = \sum_{i,j} w_{i,j} z_{pos1+i,pos2+j}$. So it is a weighted sum of signals of detectors over different positions.

  A localized $k \times k$ combination of neurons allows to learn a combination of parts that are *neighboring*. Why it is ok to learn a neighboring combination of parts?

- Semantically meaningfully Parts in an image (eye, fur, leg, whole dog) form usually a connected, neighboring region in an image [1].So **a convolution at one fixed output point learns to combine neighboring parts.**

- another thought: by stacking convolution layers one can look at regions that get larger with every layer:

- If one stacks two such layers by $3x3$ kernels, then the first layer looks at 3×3, but the next layer looks at 5×5 regions (and the third 7×7). So each layer looks at regions in the input image with a larger size.

  Neurons at high layers look at very large regions of the input image

- convolutions can be applied as 1d-convolution to any sequence data, like time series, language sentences, DNA sequences.

[1] What would be a counterexample?

1.9 Kernel parameters and their influence on the output shape

What is the size of the output matrix ? Suppose we use 2d convolutions and inputs are $M \times N$. First observation: analysis can be done separately for every sliding dimension.

> Takeaway: there are three parameters influencing the output size: padding, kernel size, and stride.

It is clear what kernel size is – the shape of w.

shape without padding:

Problem: if we apply a kernel to a window, it must fit into it. If we use a 3×3 convolution kernel w, then at the borders could start only at element 1 **(index starts at 0)**, and must end at $M - 2$ $(N - 2)$.

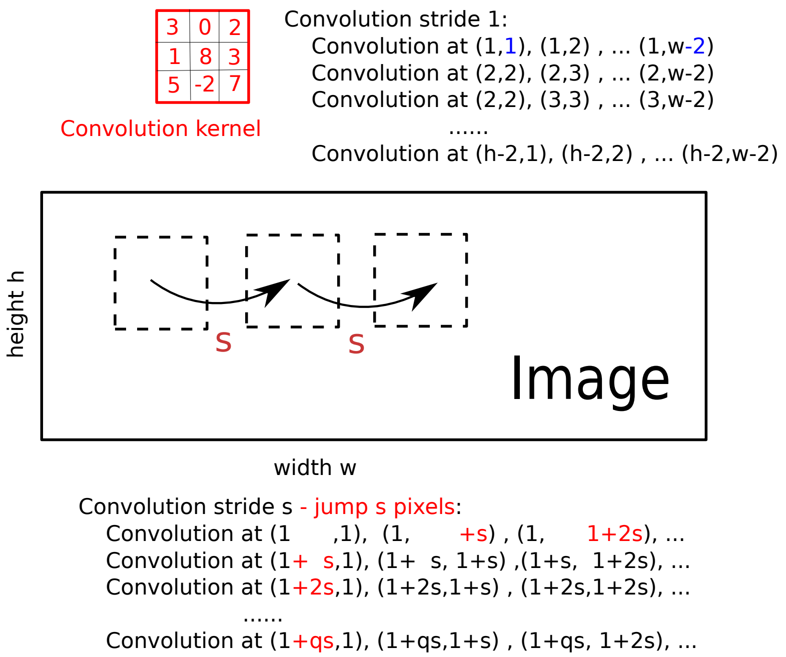
Result is: if we move w always by one pixel (=stride 1), then the output is $(M - 2) \times (N - 2)$. **That can be drawn**

For a 5×5 kernel: then at the borders could start only at element #2, and must end at $M - 3$ $(N - 3)$, so output shape will be $(M - 4) \times (N - 4)$.

> In general without padding, if we apply a kernel of size $k \times k$, and we move w always by a stride of 1, then the output shape will be $(M - k + 1) \times (N - k + 1)$

What is if we use a stride s larger than one ?

> stride is the number of input elements/pixels which w is moved in every step



For a $k \times k$ kernel we start with the first convolution window placed at $(k-1)/2$, move always $(k-1)/2, (k-1)/2+s, (k-1)/2+2s, (k-1)/2+3s$ and end at the largest index c such that $c + (k-1)/2 \leq M-1$ holds.

How many sliding windows do we have?

$$
\begin{aligned}
M &= k, \ldots, k+s-1 && \mapsto 1,\ M = k \text{ image as large as kernel} \\
M &= k+s, \ldots, k+2s-1 && \mapsto 2 \\
M &= k+2s, \ldots, k+3s-1 && \mapsto 3 \\
M &= k+3s, \ldots, k+4s-1 && \mapsto 4 \\
M &= k+4s, \ldots, k+5s-1 && \mapsto 5
\end{aligned}
$$

lets look at an example

$$
\begin{aligned}
M = k+2s, \ldots, k+3s-1 &\mapsto 3 \\
M-k+1 = 2s+1, \ldots, 3s &\mapsto 3 \\
(M-k+1)/s \in [2.x, 3] &\mapsto 3
\end{aligned}
$$

Therefore: this function can be given as:

$$ceil((M - ksize + 1)/s)$$

> If we have an input of length M in one dimension, then the output size in that dimension for a kernel of size $ksize$ and stride s without padding is given as: $ceil((M - ksize + 1)/s)$

Example: $5 \times 7, ksize = (3 \times 1), s = 2$, then: output is $ceil(5-3+1)/2 \times ceil(7-1+1)/2 = 2 \times 4$

shape with padding:
Padding means to add for every dimension at both ends of an input a layer of zeros. Example: 2d-input, pad 2 means:

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0						0	0
0	0						0	0
0	0			5x5			0	0
0	0						0	0
0	0						0	0
0	0						0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Padding of 2

- add two columns at the beginning
- add two columns at the end
- add two rows at the beginning
- add two rows at the end
- $M \times N \to (M+4) \times (N+4)$

In general: padding by r changes the input shape $M \times N \to (M+2r) \times (N+2r)$

If we pad by r, then the image dimension increases from M to $M+2r$, therefore:

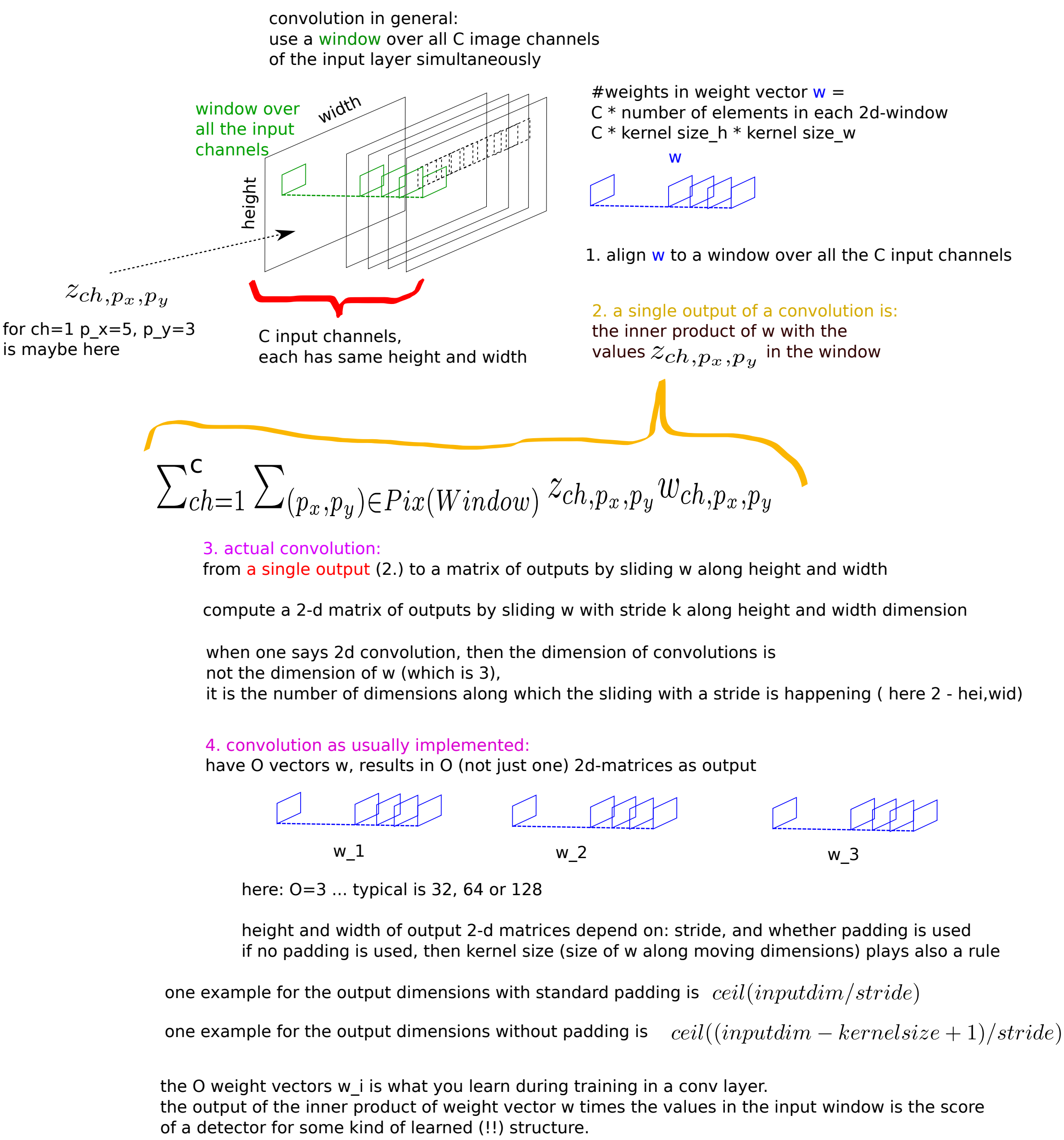
the output kernel size for a kernel of size $ksize$ and stride s with padding of r is given as: $ceil((M - ksize + 1 + 2r)/s)$

Standard padding: Standard padding is used if we pad for a kernel of size $ksize$ by a pad value r such that $r = (ksize - 1)/2$.
In such a case the output shape is $ceil(M/s)$.

Observations:

- stride s shrinks an output shape much more than kernel size $ksize$ does ... see $ceil((M - ksize + 1)/s)$
- if we use standard padding (which is adaptive), then kernel size has no influence on the output shape

- Convolution with stride s takes an image with height (h, w) and creates a downsampled image with dimensions being approximately $(h/s, w/s)$



https://github.com/vdumoulin/conv_arithmetic

2d-Convolution as recapitulation:
A 2d-convolutional layer applies a convolutional kernel w over a multi-channel

image (that is an 3-dimensional array having format $C \times width \times height$). Application means here: the kernel is slided along 2-dimensions (height,width) of the multi-channel image according to a stride. Everytime it stops over a rectangular window, an inner product between that kernel and the rectangular window is computed. This inner product is a real number. By sliding the kernel, one obtains as output one matrix per kernel. Using multiple kernels results in a multi-channel image with format $\#kernels \times newwidth \times newheight$. The weights w for all the kernels are learnt during neural network training.

In convolution we apply one weight vector w over many positions in one set of input channels – why sharing a w across the image makes sense ?
the inner product between a window of the input layer and the weight w can be seen as using w as a "detector".

- One wants to learn the detector over all regions in the image.
- when one has found a good detector, one wants to apply the same detector over all regions in the image
- for these reasons w is shared across the image.
- convolutional neural nets: combine **neighboring neurons** into a neuron in the next layer
- original paper: *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.* K. Fukushima, Biological Cybernetics, 1980
- one further idea: related to summing up $\sum_i w_{ij} z_i$, see next slides

How does a learned w look like?

One can consider the filters in the paper "Visualizing and Understanding Convolutional Networks",Matthew D. Zeiler and Rob Fergus, ECCV 2014

826 M.D. Zeiler and R. Fergus

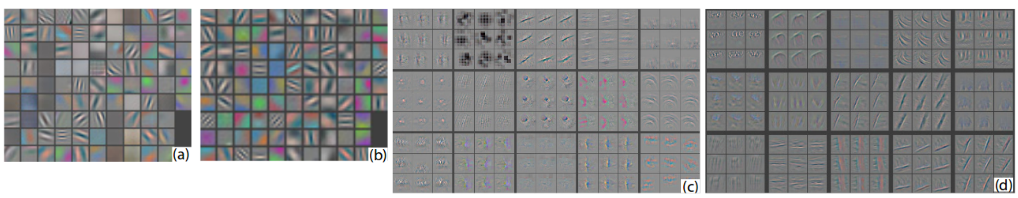


Fig. 5. (a): 1st layer features without feature scale clipping. Note that one feature dominates. (b): 1st layer features from Krizhevsky *et al.* [18]. (c): Our 1st layer features. The smaller stride (2 vs 4) and filter size (7x7 vs 11x11) results in more distinctive features and fewer "dead" features. (d): Visualizations of 2nd layer features from Krizhevsky *et al.* [18]. (e): Visualizations of our 2nd layer features. These are cleaner, with no aliasing artifacts that are visible in (d).

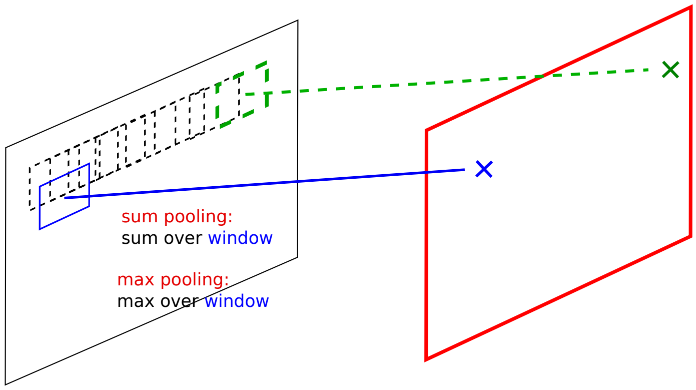
See also works by Anh Mai Nguyen, Jeff Clune, U Wyoming.

- each filter kernel w works like a detector for some structure by computing an inner product.
- the detector is applied over a window. the window gets slided. everytime the detector is applied to an array of $2-d$ images (at the input: RGB image are 3 2-d images, in higher layers one can have many more channels than 3).
- learn the values of w by backpropagation

1.10 Pooling

Its related to convolution, but has no parameters to be learned. in case of multi channels: each channel separately treated, usually not combined.

- **Sum pooling:** Sums up all the elements over a window and returns a real number. Same sliding window approach with kernel size (= window size) and stride – yields then a matrix as output.



- Equivalent to a Filter kernel $w = const$
- same as convolution: works with M input channels – but usually each channel is pooled separately!
- Idea: Often after convolution+activation ... average of detector outputs
- **Max pooling:** Computes a max up all the elements over a window and returns a real number. Same sliding window approach with kernel size (= window size) and stride – yields then a matrix as output.
- Equivalent to a Filter kernel $w = const$ and replace aggregation: sum gets replaced by a max (see that you could plug in other aggregation operations).
- Idea: Often after convolution+activation ... highest detector output over a field of view

1.11 Old-style (pre-batchnorm and residual connections) Neural Network structure for Computer vision tasks

- **Convolution-Relu-Pooling** Repeated. Last $1-2$ layers are a **fully connected** layer.
- ReLU: Rectified linear $g(x) = max(x, 0)$. Nowadays more alternatives: leaky ReLU, eLU, SeLU. Sigmoids are used for bounded regression outputs, not common for vision.