IN5400 – Convolutional Neural networks

Alex Binder

February 3, 2021

[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.]

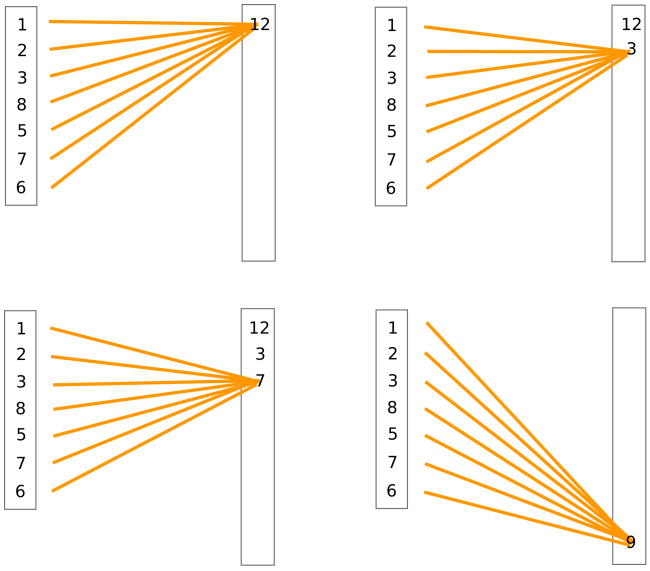
Key takeaways:

- Convolution in deep learning boils down to a sliding window of inner products
- Convolutions can represent a battery of the same pattern detector applied over 1d-,2d-,nd-spatial dimensions
- the pattern detectors in convolutions are locally applied
- convolutions have less parameters than naive fully connected networks – easier to learn
- stacking conv layers means to stack pattern detectors – higher layers detect more complex patterns built from simple ones
- formulas: convolution with C input and D output channels
- formulas: the effect of kernel size, stride and padding on the output shape
- formulas: be able to compute the size of a feature map (spatial dimensions and the number of output channels), when convolution is applied
- formulas: the theoretical receptive field size
- be able to explain how sum-pooling, max-pooling works
- be able to explain how dilated convolution works

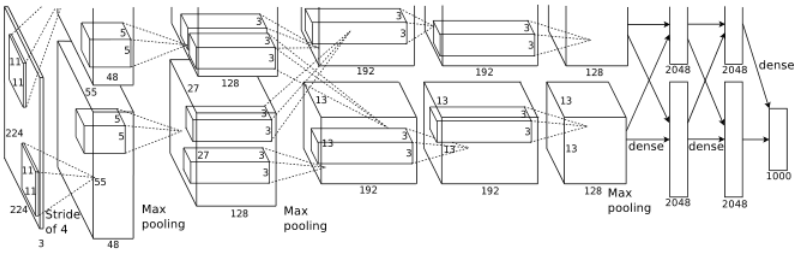
1 Convolutional Neural networks

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

1.1 not a convolution: fully connected layer, 1 input channel



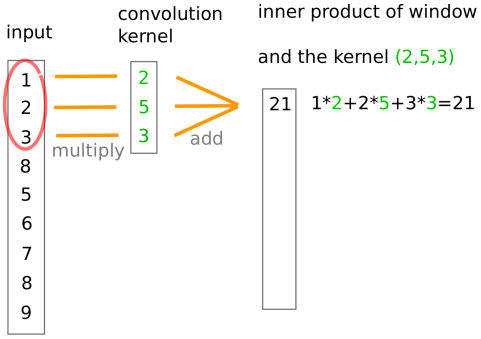
Observation for fully connected nets: number of weights grows with the number of elements in the input and the output

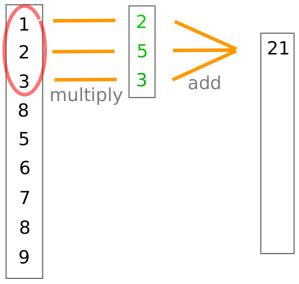


taken from: Alex Krizhevsky et al., NIPS2012 `http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf`

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

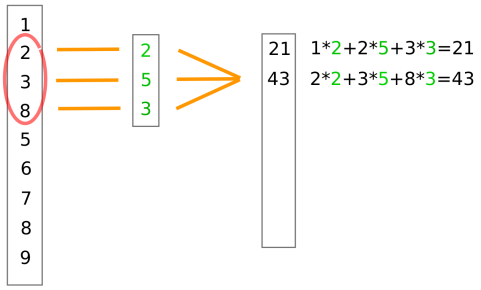
1.2 1-d convolutions, 1 input channel

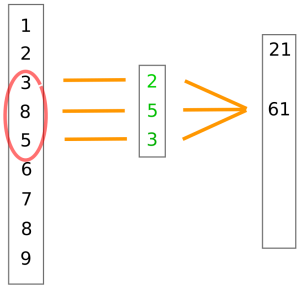




stride 1 = move kernel by 1 element

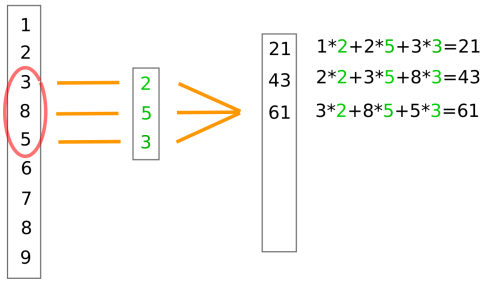
stride 2 = move kernel by 2 elements

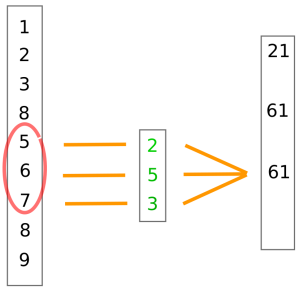




stride 1 = move kernel by 1 element

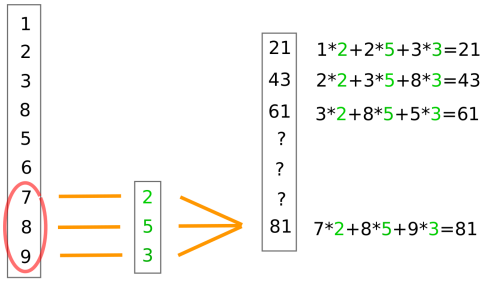
stride 2 = move kernel by 2 elements

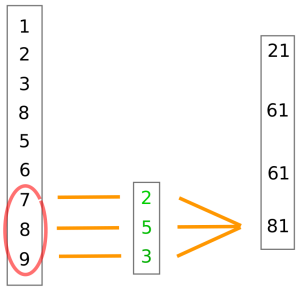




stride 1 = move kernel by 1 element

stride 2 = move kernel by 2 elements

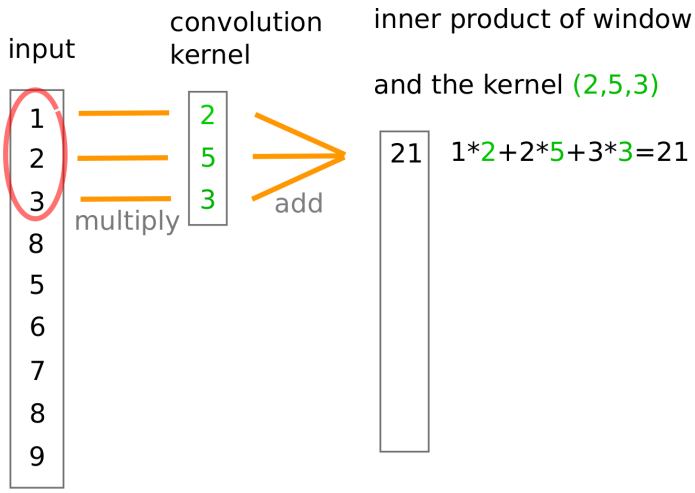




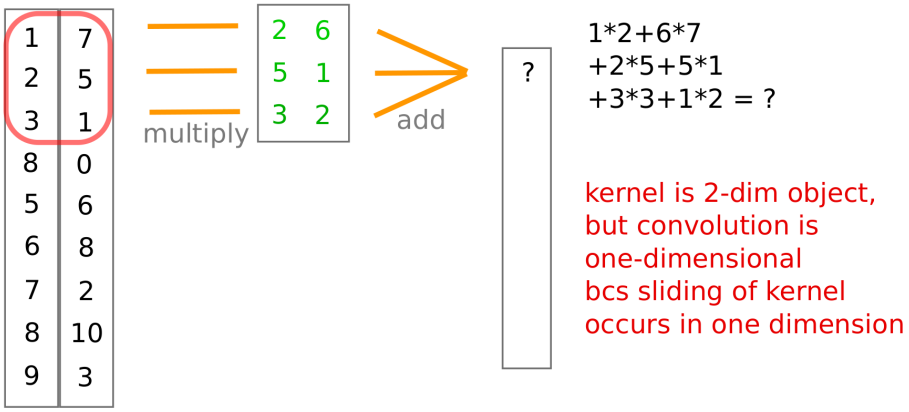
no padding of input:
outputsize = inputsize -2 = inputsize -(kernelsize-1)

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

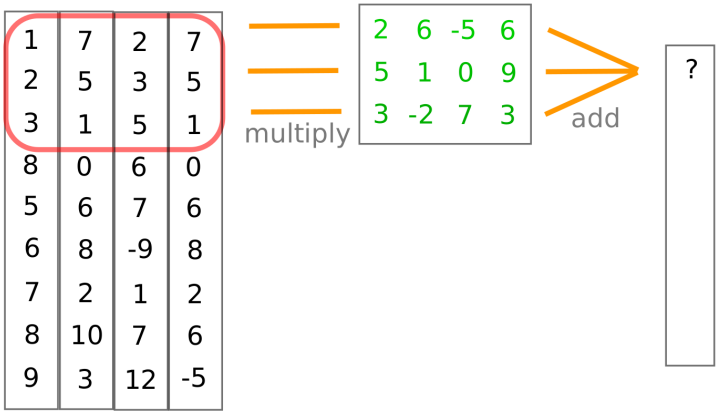
1.3 1-d convolutions, 2+ input channels



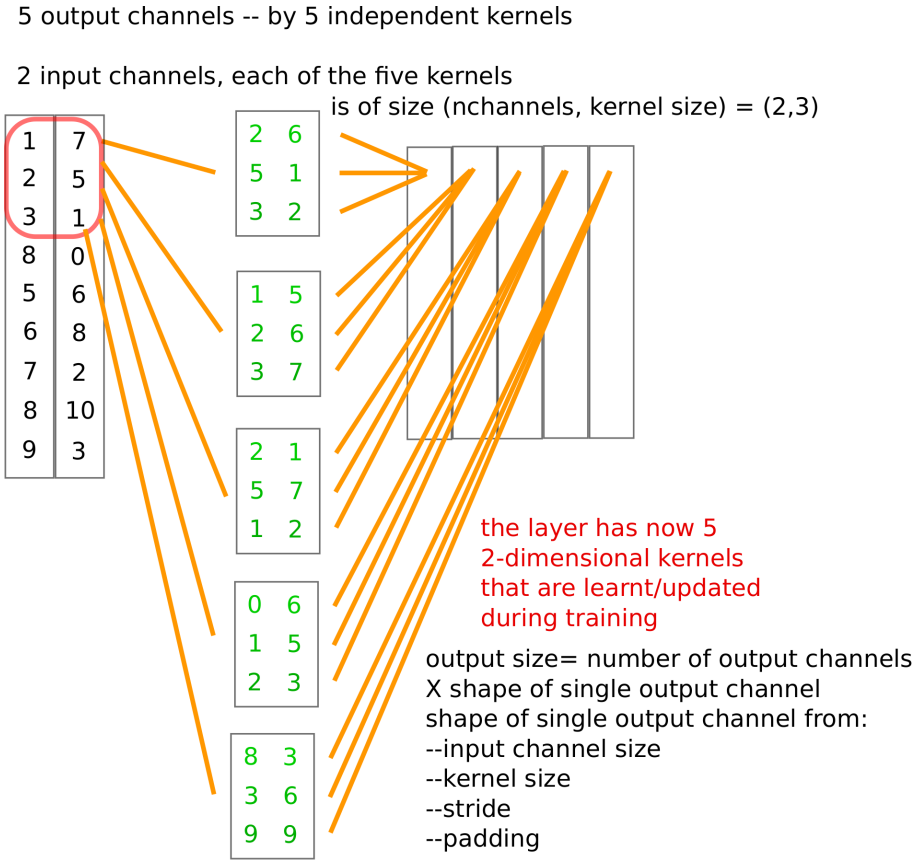
2 input channels, kernel is of size (nchannels, kernel size) = (2,3)



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? – less parameters

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

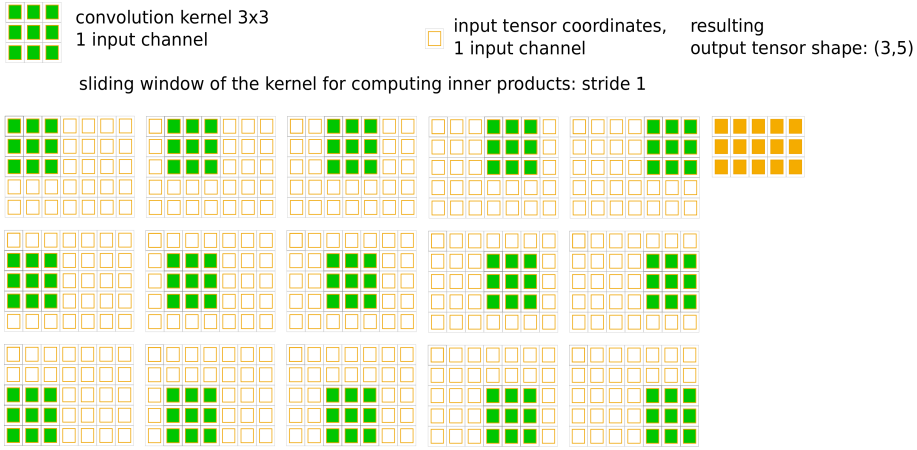
- Fully connected: Assume we have N_l neurons in layer l and N_{l+1} neurons in layer $l+1$: parameters have dimensionality =?
- Locally connected: Assume we have N_l neurons in layer l and N_{l+1} neurons in layer $l+1$, each output neuron takes input from a patch of 5 neighbors: parameters have dimensionality ?
- 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:

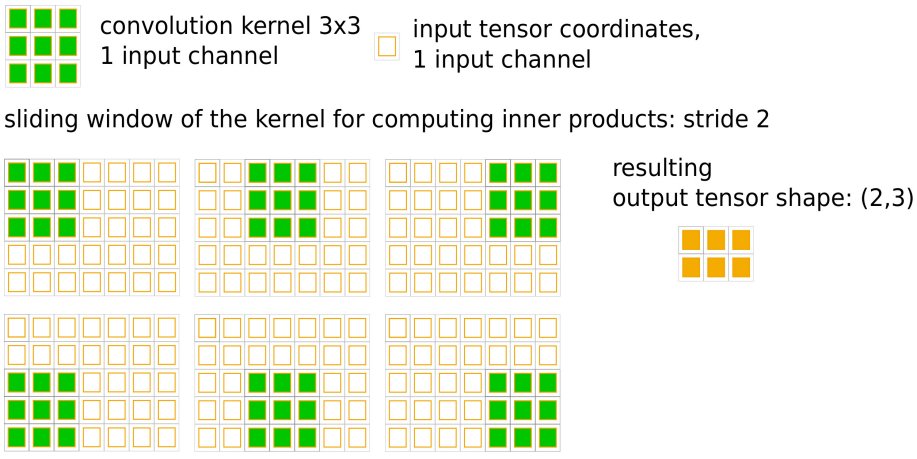
- machine learning in general: keep number of parameters limited relative to training set size
- deep learning in particular: better stack simple functions in deeply in many layers than learning in one layer something very complex.

1.6 2-d convolutions, 1 input channel

A simplified convolution (one input channel only) can be drawn like this for stride 1



A simplified convolution (one input channel only) can be drawn like this for stride 2



In terms of math language a 2d-convolution in deep learning **for one input channel**, with stride 1, no padding (and dilation 1) can be expressed like this:

$$z[h,w] = \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} u[i_h, i_w] x[h+i_h, w+i_w]$$

In image processing this is known as cross-correlation, however.

Image processing has its own definition of convolution, which would be:

$$\begin{aligned} z[h,w] &= \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} u[i_h, i_w] x[h + k_h - i_h, w + k_w - i_w] \\ &= \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} u[k_h - i_h, k_w - i_w] x[h + i_h, w + i_w] \end{aligned}$$

There in each axis – the filter w is traversed in the opposite direction, as if it was rotated by 180 degrees!

> Take note of this point of confusion: deep learning convolution is cross-correlation in image processing!!
> 2D-Convolution in image processing amounts to using a 180 degrees rotated filter

Technically, convolution as in deep learning is an inner product (no matter the kernel is traversed forward or inverted). Below shows one output element $z[h, w]$ at position (h, w) in the output feature map:

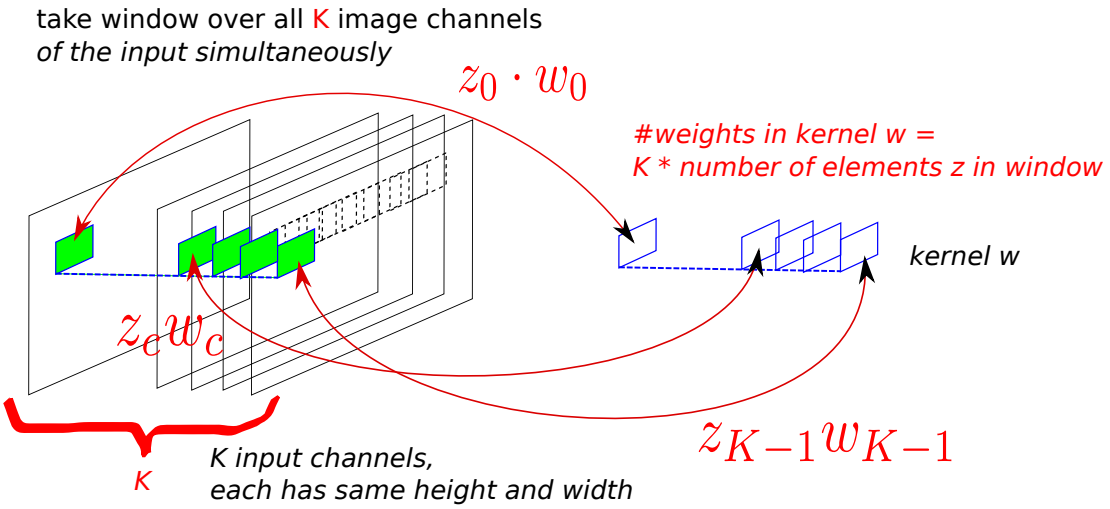
$$z[h,w] = \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} u[i_h, i_w] x[h + i_h, w + i_w]$$

between u and the slice $x[h : h + k_h, w : w + k_w]$ of tensor x

> Take note of the fact that convolution is a series of inner products between the filter u and a window-view of the tensor.
> The window is sliding, for 1D-convolutions along 1 dimensions, for 2D-convolutions along 2 dimensions.

1.7 dealing with multiple input channels

A 2D-convolution with multiple input channels and one output channel can be drawn like this:



$$output(oneposition) = z_0 w_0 + \ldots + z_c w_c + \ldots + z_{K-1} w_{K-1}$$

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

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

In math, one output element of a 2D-convolution with C input channels and 1 output channel can be written as:

$$z[h,w] = \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} \sum_{c=1}^{C} u[c,i_h,i_w] x[c,h+i_h,w+i_w]$$

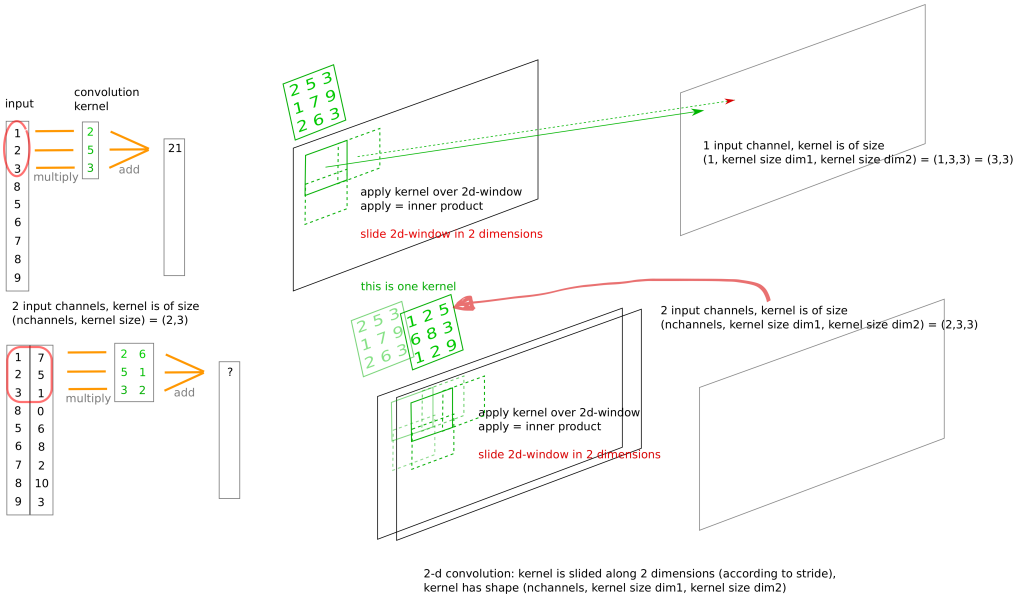
This is still an inner product between
$u[:,:,:]$ and the slice $x[:,h:h+k_h,w:w+k_w]$ of tensor x.

The convolution kernel has one additional dimension, besides the number of dimension used to run the sliding window.
This is to process multiple input channels. In this additional it has as many entries as the number of input channels in the feature map to be processed.

In math, one output element of a 2D-convolution with C input channels and D output channels can be written as:

$$z[d, h, w] = \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} \sum_{c=1}^{C} u[c, i_h, i_w, d] x[c, h + i_h, w + i_w]$$
$$d \in 0, \ldots, D-1$$

compare 1d vs 2d convolutions:



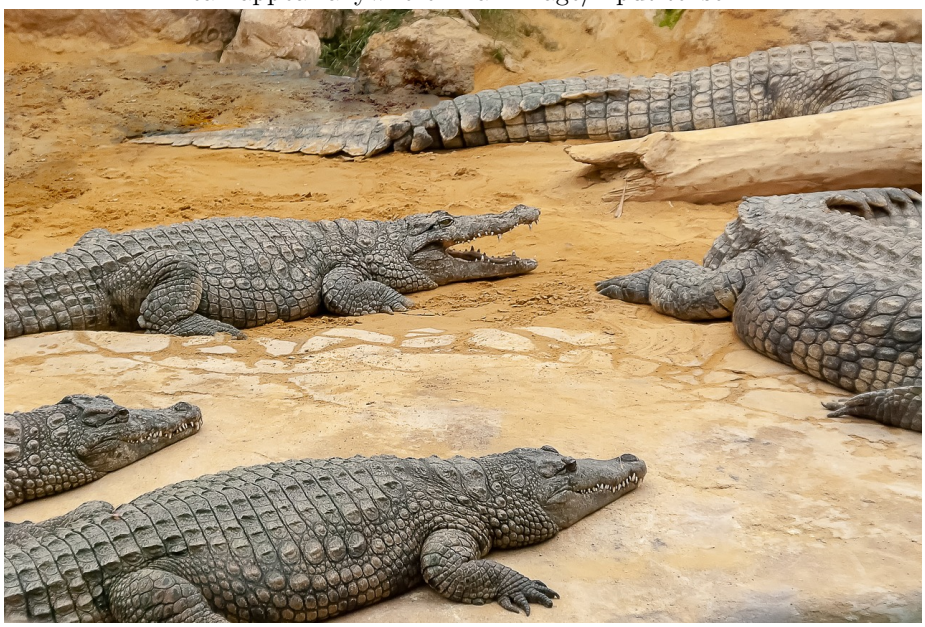
- a convolution at one fixed region (region means here: rectangular window) x of an image is an inner product $w \cdot x$ between kernel weights w and this region x - this is a single real number.
- convolution: we slide the convolutional kernel w over the image/input tensor and apply the inner product over many windows. **In convolution we apply the same kernel w across all locations** for computing inner products.
- the parameters to be learned during training are the values of the kernel w!

1.8 why convolutions II? – one layer as battery of pattern detectors

Convolution implements a battery of localized pattern detectors all over the input tensor.
Why is that useful?

an object ...

can appear anywhere in an image/input tensor



Therefore one wants to be able to perform a detection over all positions in an input tensor.

In order to see why:

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

  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)
- compare to fully connected layer: detection is global for fully connected, localized for convolutions

> Convolution allows to perform a pattern detection in a translation-invariant manner!

1.9 Kernel parameters and their influence on the output shape

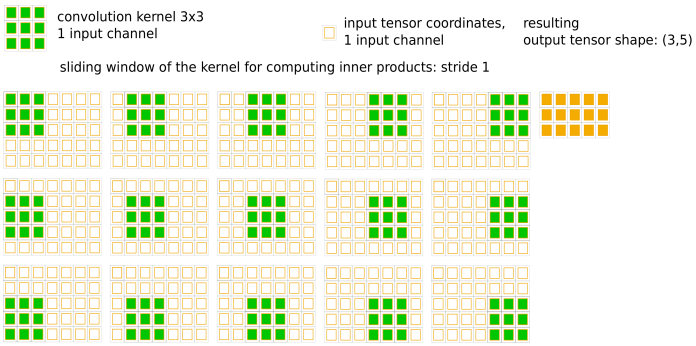
What is the size of the output matrix ? Suppose we use 2d convolutions and inputs are $M \times N$. First observation: analysis can be done separately for every sliding dimension.

> 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.

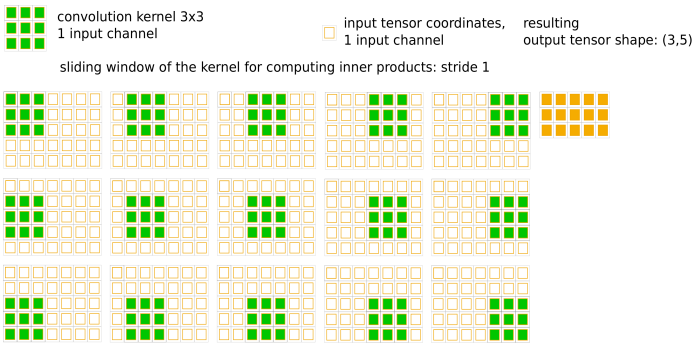
shape without padding:

Problem: if we apply a kernel to a window, it must fit into it. An array of length M starts at 0 and ends at $M-1$.



If we use a 3×3 convolution kernel w, if its left border start at index 0, then its right border ends at index 2. Thus the last start can be only at index $M-3$, because then its right border would be at $M-3+2 = M-1$. $\{0, 1, \ldots, M-3\}$ are $M-2$ elements.

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



For a 5×5 kernel: if its left border start at index 0, then its right border ends at index 4. Thus the last start can be only at index $M-5$, totalling $M-4$ elements

> In general without padding, if we apply a kernel of size $k \times k$, and if 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

draw a stride fig / ksize 3,5

For a $k \times k$ kernel we start with the left boundary at 0,
move always $0, 0+s, 0+2s, 0+3s$
and end at the largest index c such that $cs + k - 1 \leq M - 1$ holds.

How many sliding windows do we have as a function of input size M
($M = k$ means image is as large as the kernel size k)?

$$\begin{aligned} M &= k && \mapsto 1,\ c = 0 \\ M &= k + s && \mapsto 2,\ c = 1 \\ M &= k + 2s && \mapsto 3,\ c = 2 \\ M &= k + 3s && \mapsto 4,\ c = 3 \\ M &= k + 4s && \mapsto 5,\ c = 4 \end{aligned}$$

Therefore: we seek the largest c such that

$$cs + k - 1 \leq M - 1$$

this means:

$$c \leq \frac{M-k}{s}$$

However c is an integer, $\frac{M-k}{s}$ might be a fractional number. consider examples

- $\frac{M-k}{s} = 2.4$ – then the largest $c = 2$
- $\frac{M-k}{s} = 3.3$ – then the largest $c = 3$
- $\frac{M-k}{s} = 5.9$ – then the largest $c = 5$
- $\frac{M-k}{s} = 5.0$ – then the largest $c = 5$
- $\frac{M-k}{s} = 3.0$ – then the largest $c = 3$

in all cases $c = floor(\frac{M-k}{s})$ ($floor(v)$ is the largest integer $\leq v$), therefore

$$c \leq floor(\frac{M-k}{s})$$

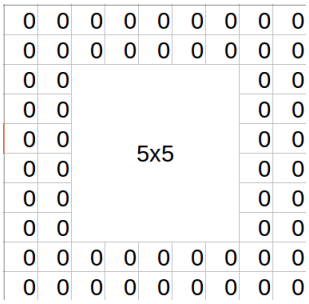
As seen above, we have always $c+1$ outputs, therefore the output size is

$$floor(\frac{M-k}{s}) + 1 = floor(\frac{M-k}{s} + 1)$$

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: $floor((M - ksize)/s + 1)$

Example: 5×7, ksize= (3×1), $s = 2$, then: output is $floor(\frac{5-3}{2} + 1) \times floor(\frac{7-1}{2} + 1) = 2 \times 4$

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:



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: $floor((M+2r-ksize)/s+1)$

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

Observations:

- stride s shrinks an output shape much more than kernel size $ksize$ does ... see $floor((M-ksize)/s+1)$
- if we use standard padding (which is adaptive), then kernel size has no influence on the output shape
- Qualitative impact of stride: 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: sliding window of inner products, combines **neighboring/localized 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

1.10 The theoretical receptive field

- 3×3,stride= 1 kernel has a field of view of 3
- if we stack two kernels 3×3,stride= 1 – field of view ? 5!
- if we stack three kernels 3×3,stride= 1 – field of view ? 7!
- 3×3,stride= 1 kernel has a field of view of 3
- if we stack two kernels 3×3,stride= 2 and 3×3, any stride – field of view ? 7!

Important: the stride of the first kernel plays a role for the second feature map, the stride of the second kernel would be of importance only for the third feature map!

Let R be the receptive field size at layer $k \geq 1$. S_k the stride in layer k, F_k the kernel size in layer k.
Then the theoretical receptive field will be:

$$R_k = R_{k-1} + (F_k - 1) \prod_{i=1}^{k-1} S_i$$

What is preferable to use?

- one 5×5,stride= 1 or a stack of two 3×3 , stride= 1 ??
- both have the same receptive size

1.11 why convolutions III? – stacking convolutions

- Think of the image in the above graphic not as an image, but as a grid showing signals of detectors.

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

  If the input tensor is the output of a previous convolution layer with kernel v, then the input is a a detector for the signal encoded by v , because it measures similarity to v in every spatial location.

  Then every input "pixel" of the input feature map is a signal of some detector for a kind of part/structure, e.g. v may encode an eye, a furry ear or a car wheel.

  A convolution is a weighted sum of inputs

$$z[d,h,w] = \sum_{i_h=0}^{k_h} \sum_{i_w=0}^{k_w} \sum_{c=1}^{C} u[c,i_h,i_w,d]x[c,h+i_h,w+i_w]$$

  $\Rightarrow$ It is a weighted sum of signals of different detectors (by it sum over different channels c) over neighboring spatial positions (sum over i,j).

  A convolution allows to learn a combination of part detectors that are spatially *neighboring*.

- Why it is ok to learn a neighboring combination of parts? Semantically meaningfully parts in an image (eye, fur, leg, whole cat) 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

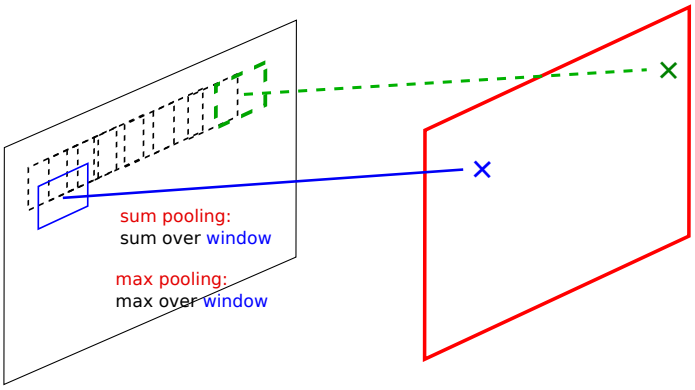
- convolutions can be applied as 1d-convolution to any sequence data, like time series, language sentences, DNA sequences.

[1] What would be a counterexample?

1.12 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.



- **Average pooling:** does not sum, but computes the average over the number of elements
- 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

1.13 Old-style (2014 ... before-batchnorm and before 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.

1.14 Dilated convolutions / a-trous convolutions / atrous convolutions

atrous - its French

- dilation factor $d = 1$ - usual convolution
- dilation factor $d = 2$ - put between each two elements in u 1 zero, apply convolution with zeros-expanded u_*, effectively uses every second value of the input feature map
- dilation factor $d = 4$ - put between each two elements in u 3 zeros , apply convolution with zeros-expanded u_*, effectively uses every fourth value of the input feature map
- putting $d-1$ zeros into the kernel – can be seen as skipping $d-1$ elements in the input each time, and using only each d-th element in the input:

$$z[h] = \sum_{i_h=0}^{k} w[i_h]x[h + i_h * d]$$

`https://github.com/vdumoulin/conv_arithmetic`

- idea: obtain larger receptive fields, looks at larger contexts
- used in segmentation methods e.g. deeplab v2/v3 `https://arxiv.org/abs/1606.00915`, `https://arxiv.org/abs/1706.05587`

1.15 fractionally strided convolutions

Will be treated in the GAN class. Will be class-relevant topic then.

1.16 other topics

How does a learned w look like?

Can answer: what structures does it fire mostly on. How ? Compute convolution responses over a large set of real images and sort! Later lecture on explainable AI: how to compute scores for feature map channels for images which maximize the activation of a channel.

See "Visualizing and Understanding Convolutional Networks",Matthew D. Zeiler and Rob Fergus, ECCV 2014

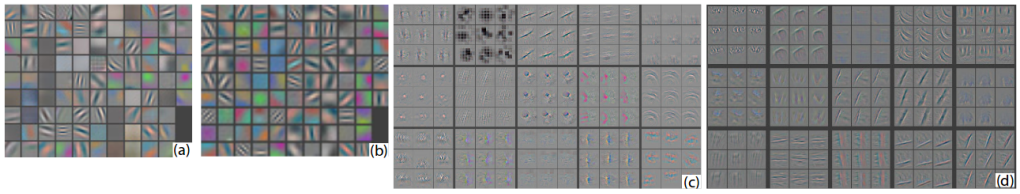


Fig. 5. (a): 1st layer features without feature scale clipping. Note that one feature dominates. (b): 1st layer features from Krizhevsky *et al.* [18]. (c): Our 1st layer features. The smaller stride (2 vs 4) and filter size (7x7 vs 11x11) results in more distinctive features and fewer "dead" features. (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