50.021 -AI

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.]

Convolutional Neural networks

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

Convolutional Neural Networks are besides Pooling Operators an essential components of Neural networks for any vision tasks.

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 $l+1$.

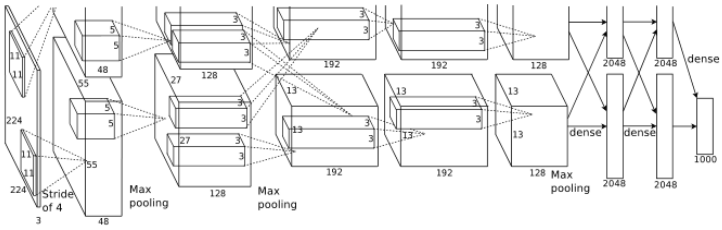
Assume we have N_l neurons in layer l and N_{l+1} neurons in layer $l+1$: then the weight vector has dimensionality $N_l * N_{l+1}$, $1000 \times 1000 =$ a million.

Why is that a problem, an old paper of Yann Lecun formulated a rule of thumb: a neural networks needs 4 times as many samples as weights. so for simple $1000x1000$ need 4 million data points only for two layers.

Otherwis we are in the situation of too many parameters, too little data: overfitting.

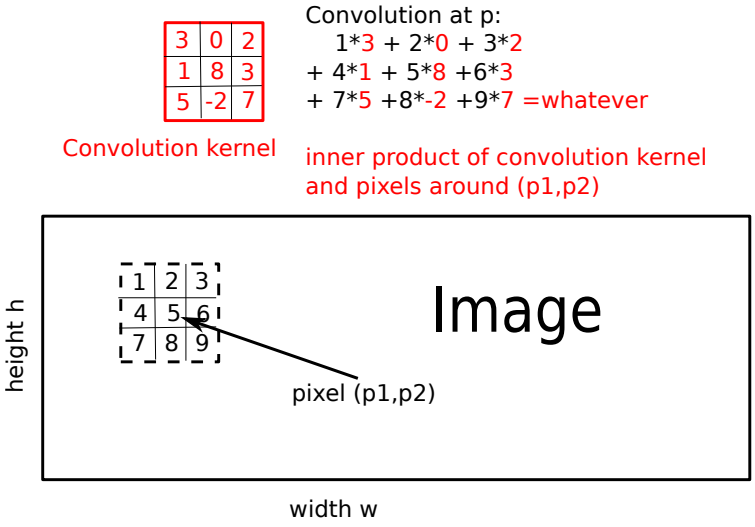
In NN for vision there is one way to deal with it: convolutional layers. Simple example: 1000 and 1000 neurons but each neuron in the second layer takes only 25 connections from neighboring neurons of the layer before ... only 25 000 weights = factor 40 less.

now consider that in 2 dims: neuron layer l is a 2-dim layer, layer $l+1$, too. one output neuron is computed by taking connections from e.g. $5x5$ neurons in the layer below.



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.



- First thought: 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? Every neuron gives a high signal for some inputs, and a low signal for other inputs. So it can be seen as a detector for some kind of structure.

  Then a pixel in above graphic is an **input neuron**, and this neuron is a detector for some typical structure, e.g. a car wheel, a car window. So, above weighted sum 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 structures that are *neighboring*.

- Semantically meaningfully Parts in an image (eye, fur, leg, whole dog) form usually a connected, neighboring region in an image [1].

[1] a fence is a counterexample!

So above windowed-sum: **a convolution at one fixed 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, then the first layer looks at 3×3, but the next layer looks at 9×9 regions (ad the third 27×27). So each layer looks at regions in the input image with a larger size.

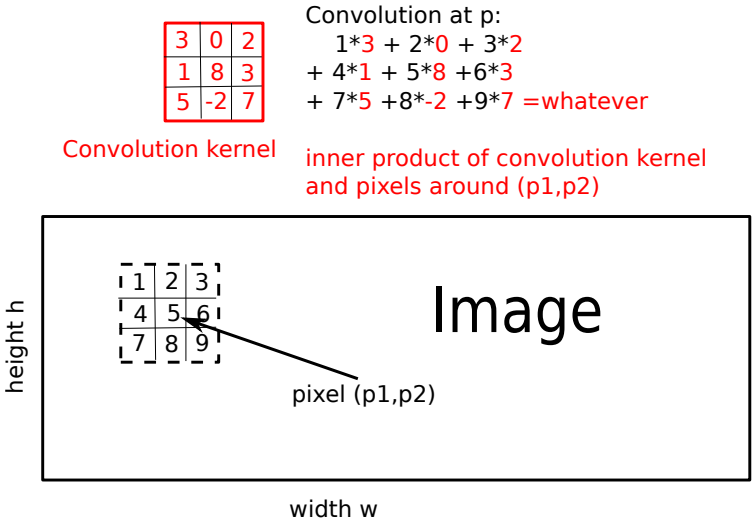
  Neurons at high layers look at very large regions. at the output they cover the whole image

- We dont know right size of neighbors to look at – no problem: higher layers cover / "look at" bigger regions.

The neighbor aspect can be described also as follows:

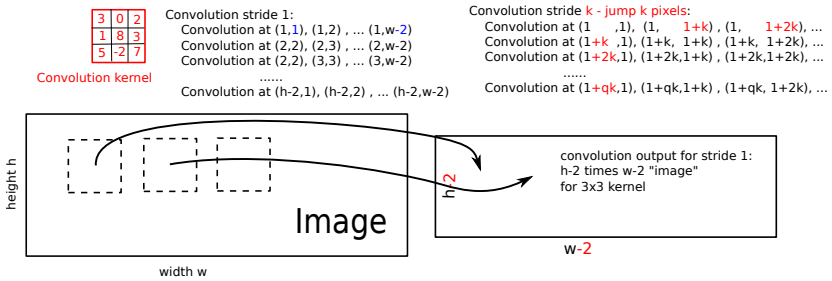
- it makes sense to learn to combine the car wheel and the car window nearby - this belongs likely to the same car.
- Do NOT learn to combine a car wheel and another car window at the opposite end of an image - likely one learns something wrong

Lets go into technical details of a convolution



- 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.
- the parameters to be learned are the values of the kernel w!

- 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.

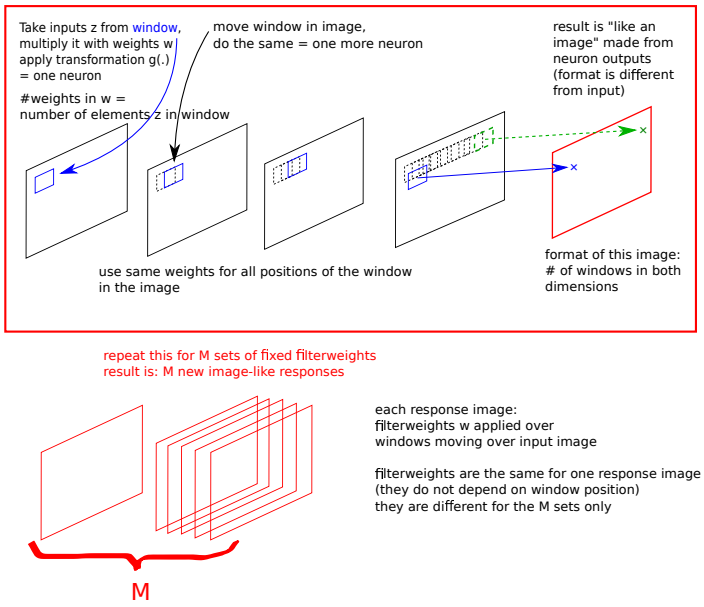


What is the size of the output matrix ? Suppose inputs are $M \times N$. Problem: if we apply w to a window, it must fit. If we use a 3×3 convolution kernel w, then at the borders could start only at element #2, and must end at $M-1$ ($N-1$).

There is one alternative to this: **padding**. We pad the input matrix by adding two times 1 columns and 1 row (we add 1 column on top, 1 column below , ... same with rows)

then:

- if we move matrix w 1 row and 1 column always, then result is $M \times N$ output
- if we move matrix w 2 rows and 2 columns, then result is $M/2 \times N/2$ output approximately, actually $ceil(M/2) \times ceil(N/2)$



- if we move matrix w k rows and k columns, then ?

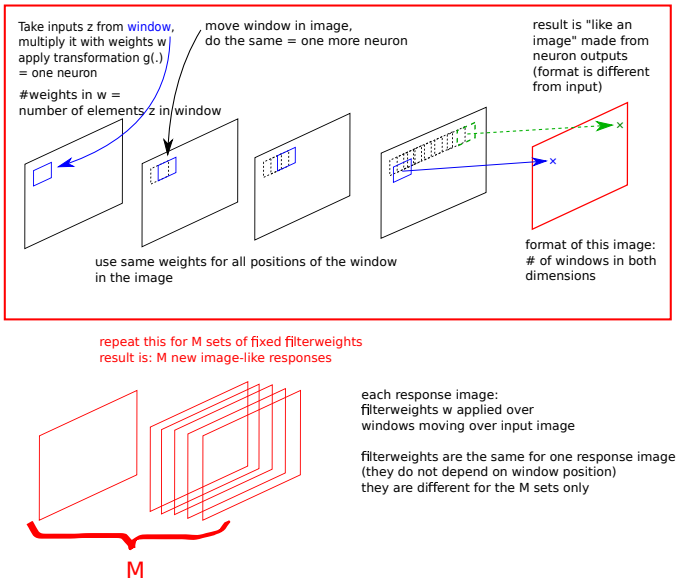
This move/sliding size k is called **stride**.

A stride k results in approximately $ceil(N/k) \times ceil(M/k)$ output if padding is used. Sidelength of w does not matter (padding adapts to this). [2]

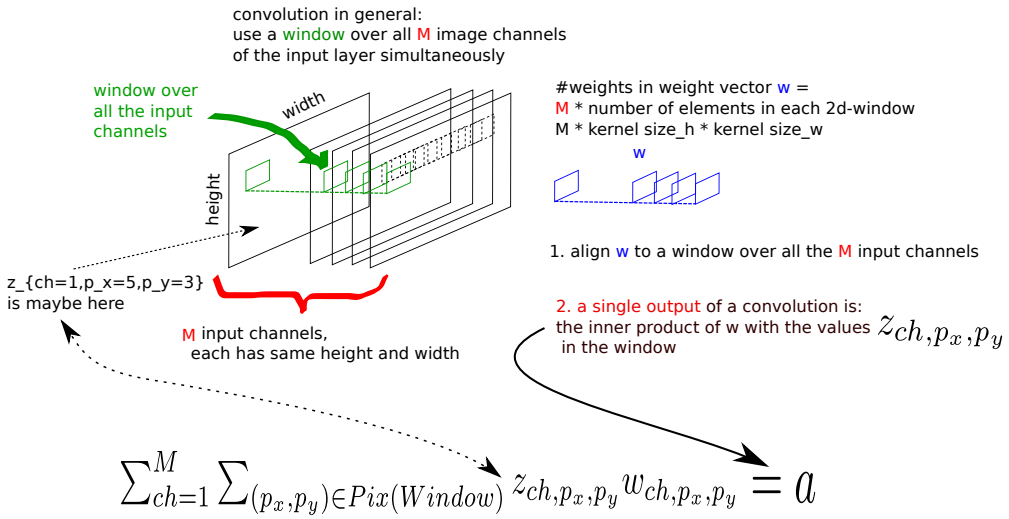
Choosing a strides controls how to reduce the size of output.

[2] without padding the output size would be $ceil([M - l(w) + 1]/k) \times ceil([N - l(w) + 1]/k)$

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



- can apply to images with multiple input channels - conv kernel has one depth dimension more, result still one channel image.
- can apply multiple conv kernels - outputs an image with multiple channels again

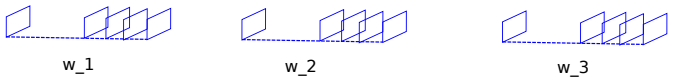


3. actual convolution:
from a single output (2.) to a matrix of outputs by sliding w along height and width

compute a 2-d matrix of outputs by sliding w with stride k along height and width dimension

the dimension of convolutions is not the dimension of w,
it is the number of dimensions along which the sliding with a stride is happening (here 2 - hei,wid)

4. convolution as usually implemented:
have C vectors w, results in C (not just one) 2d-matrices as output



here: C=3 ... typical is 32, 64 or 128

height and width of output 2-d matrices depend on: stride, and whether padding is used
if no padding is used, then kernel size (size of w along moving dimensions) plays also a rule

one example for the output dimensions with standard padding is $ceil(inputdim/stride)$

one example for the output dimensions without padding is $ceil((inputdim - kernelsize + 1)/stride)$

the C weight vectors w_i is what you learn during training in a conv layer.
the output of the inner product of weight vector w times the values in the input window is the score of a detector for some kind of learned (!!) structure.

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

Convolution as definition:

A 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. 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 can be 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 every where in the image
- for these reasons w is shared across the image.
- convolutional neural nets: combine **only neighbor 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

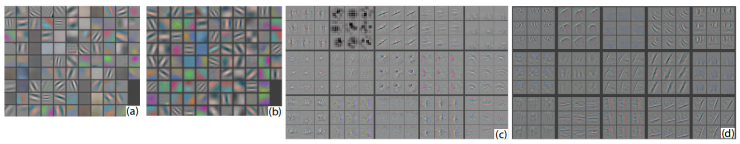


Fig. 5. (a): 1st layer features without feature scale clipping. Note that one feature dominates. (b): 1st layer features from Krizhevsky *et al.* [18]. (c): Our 1st layer features. The smaller stride (2 vs 4) and filter size (7x7 vs 11x11) results in more distinctive features and fewer "dead" features.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

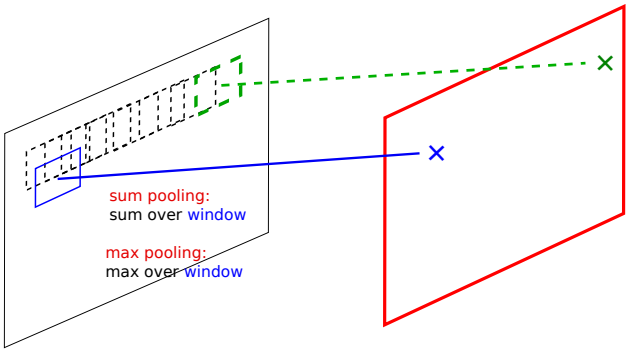
- each filter kernel w works like a detector for some structure by computing an inner product.

- 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

Pooling

Its related to convolution, but has no parameters to be learned.

- **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 $w = const$ $\sum_i z_i \cdot 1$
- same as convolution: works with M input channels. See 1×1 pooling (or convolution).
- 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

Typical Neural Network structure for Computer vision tasks

- **Convolution-Relu-Pooling** Repeated. Last $1-2$ layers are a **fully connected** layer.
- Relu: Rectified linear $g(x) = max(x, 0)$
- Sometimes right after convolution follows a **batch normalization** layer. *Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. https://arxiv.org/abs/1502.03167, ICML 2015*

  Note: Batch normalization uses the parallel copies per batch structure of neural networks. Idea:

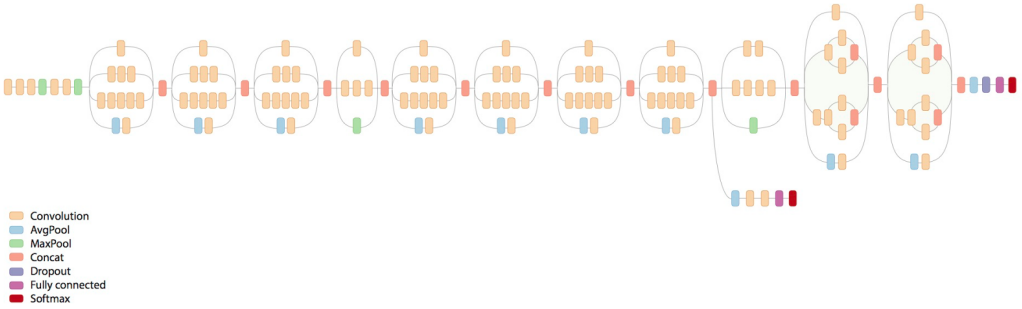
  Normalize the outputs of one layer and all elements in a minibatch to have unit standard deviation and mean zero, after that apply an affine transformation:
  Training time:

$$
\begin{aligned}
x_{nor} &= \frac{x-\mu}{\sigma} \text{ for } x \in minibatch \\
\mu &= mean_{x \in minibatch} \\
\sigma &= std_{x \in minibatch} \\
y &= \alpha x_{nor} + \beta
\end{aligned}
$$

  This is however a bit different at test time, see the paper, where μ and σ are computed over the whole training set instead.

- another component **shortcut connections across network layers**: "**ResNets**". `http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf`

- "inception": takes an input and lets multiple convolution and pooling operators work on them in parallel. It is Multi-scale processing.



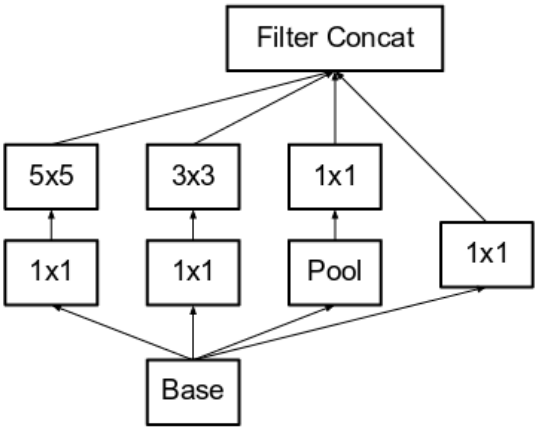


Figure 4. Original Inception module as described in [20].

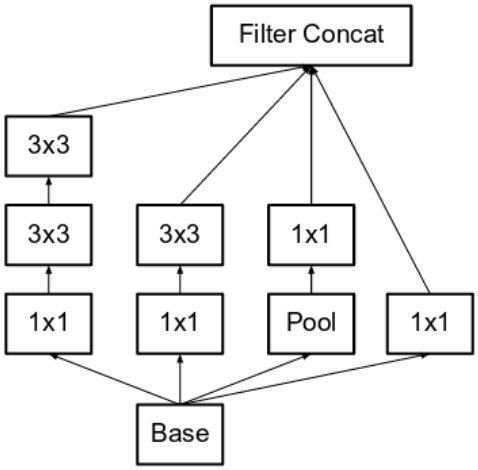


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

from: Rethinking the Inception Architecture for Computer Vision
`https://www.arxiv.org/pdf/1512.00567v1.pdf`

- ResNets with BatchNormalization are still the state of the art.
- train ensembles of models – bcs every model find just a local optimum - ILSVRC 2016 challenge was won by them.

Average out
(similar to CLT. randomness converge to mean)

- later in this class: **no one trains neural nets from scratch!!** (when data is less than $n \cdot 100000$) – transfer learning. Reuse weights from another model, except for the very last prediction layer.