Convolutional Neural Networks (Ch 14)

- Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, and in the last few years they achieved better than human performance on some complex visual tasks

- CNN power image search services, self-driving cars, automatic video classification systems

- But CNNs are not restricted to visual perception: they are also successful at many other tasks, such as voice recognition and natural language processing.

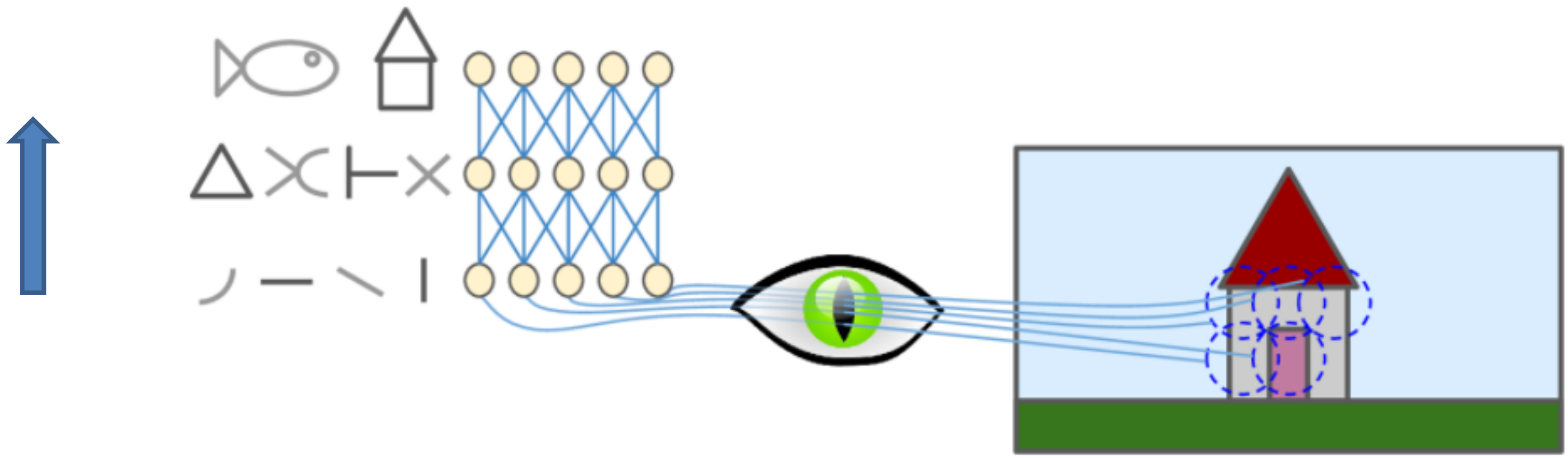
- We will focus on visual applications

- Is object detection hard?
  - Computers couldn't recognize images for a long time (think about captchas)
  - Recognizing objects on images feels easy for humans, but is it really easy?
  - In the human brain object recognition is unconscious, which is why it feels so easy
  - Major parts of the human brain is dedicated for the visual input
  - Because it is unconscious, we can't explain how we do it.

- How we (humans) do it?
  - In 1958-1959 studies were done on cats and monkeys to understand the visual cortex (Nobel Prize)
  - They showed that many neurons in the visual cortex have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field
  - The receptive fields of different neurons may overlap, and together they tile the whole visual field
  - some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations)
  - some neurons have larger receptive fields, and they react to more complex patterns that are combinations of the lower-level patterns

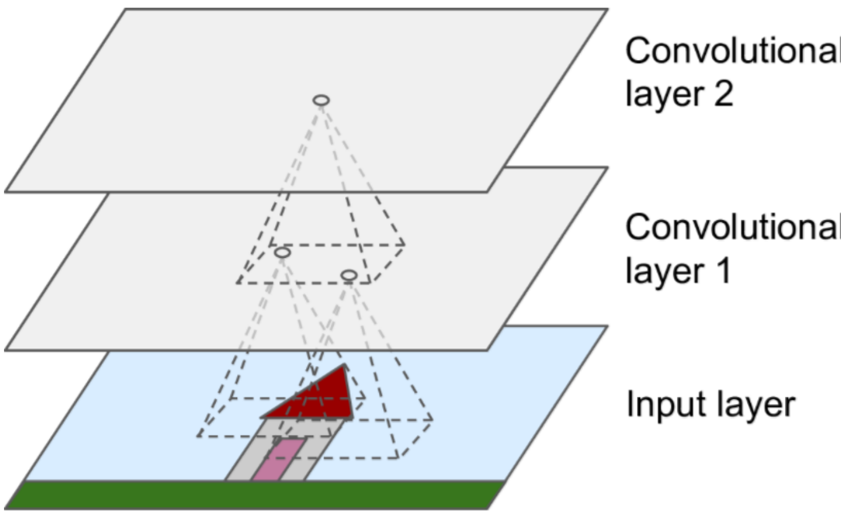
– higher-level neurons are based on the outputs of neighboring lower-level neurons

– this powerful architecture is able to detect all sorts of complex patterns in any area of the visual field



- This resulted in what we call "Convolutional Neural Networks (CNN)" today
- Why not use a regular fully connected deep neural network?
  - Because the number of parameters (connections weights) would be way too high

- The Convolutional Neural Networks have two new types of layer:
  - Convolutional layer:
    - neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields
    - each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer

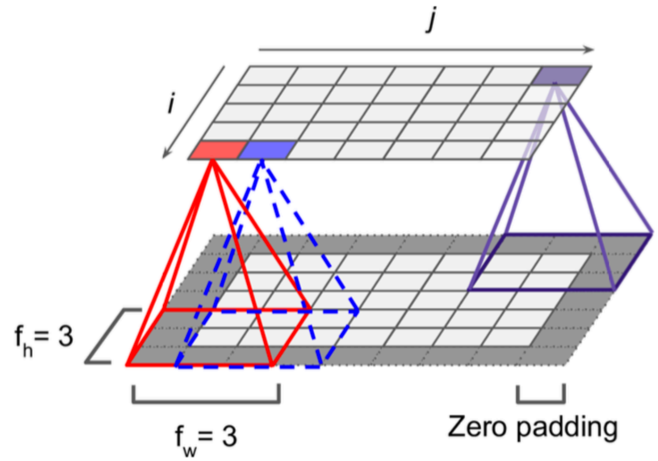


=> the network can concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer

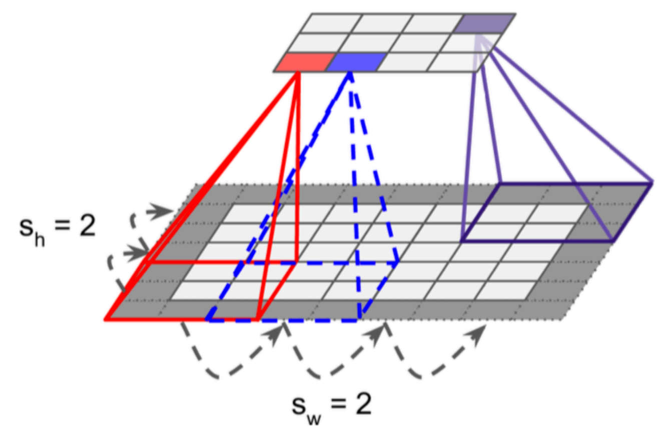
- Connections between convolutional layers:

- With zero padding (layer size is preserved)

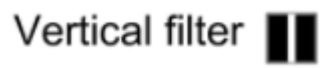
  The neuron at position (i, j) is connected to the outputs of neurons in previous layer in
  rows: i *to* $i + f_h - 1$
  columns: j *to* $j + f_w - 1$



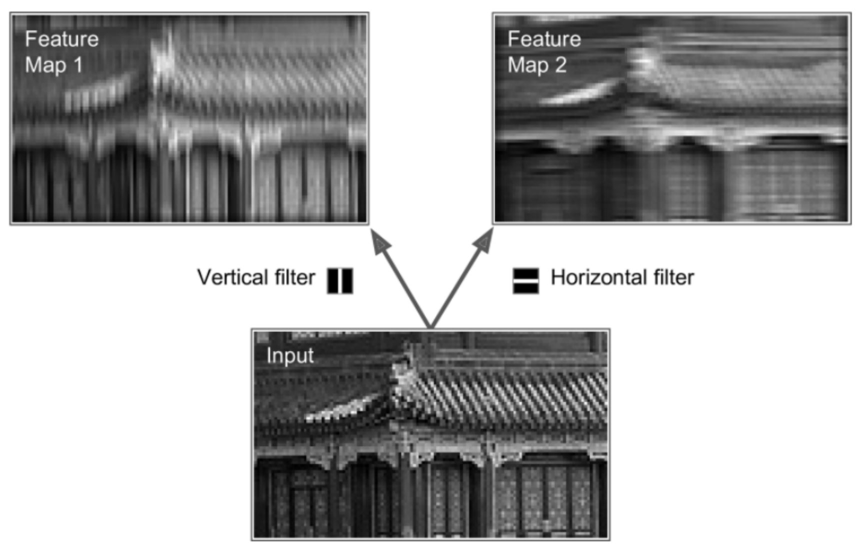
- With stride to reduce dimensionality (stride=2):



- Filters / convolutional kernels:
  - A filter is a given set of neuron weights for example a 7x7 matrix full with 0s except the central column which is full with 1s



  - Neurons using these weights will ignore everything except vertical lines
  - A whole layer of neurons has these same weights - so they can detect vertical lines anywhere in the image
  - Applying two different filters to get two feature maps:

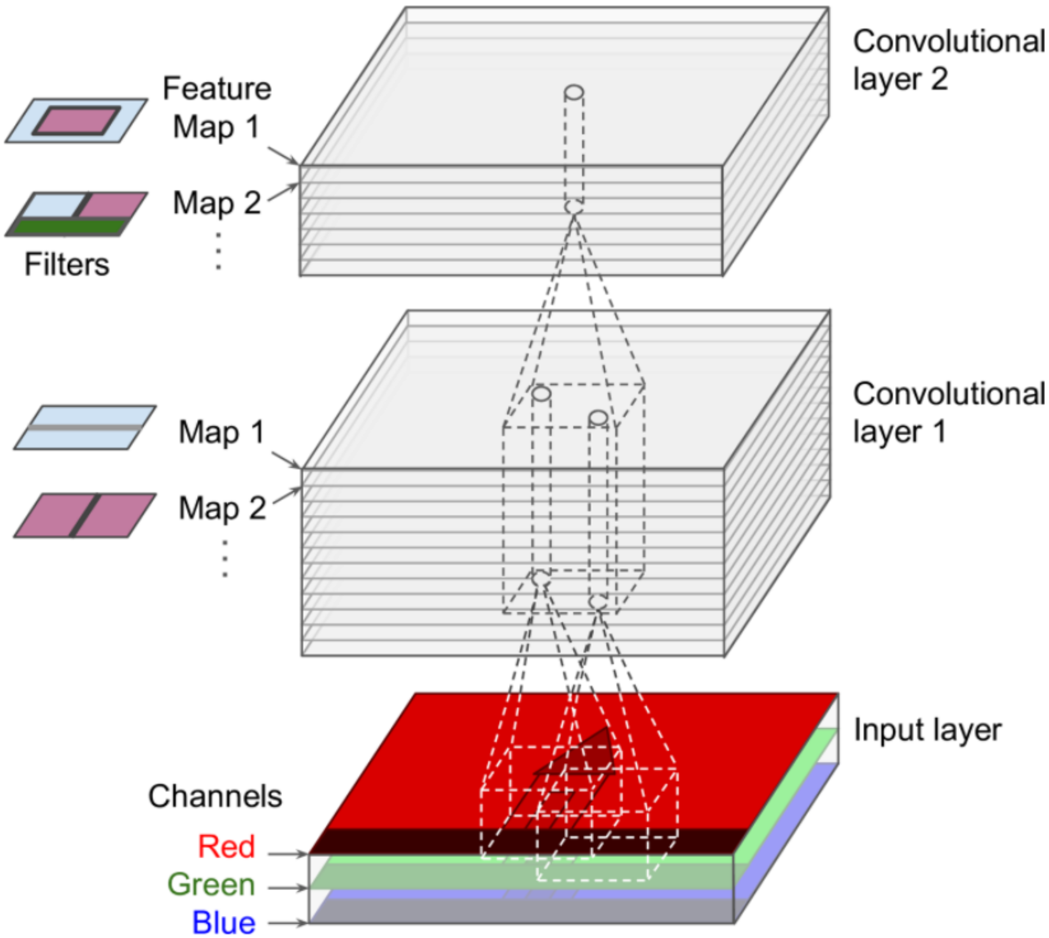


- Stacking multiple feature maps:

  - Convolutional layers are really 3 dimensional, as each of these layers consists of multiple feature maps

  - Within one feature map all neurons share the same parameters (weights and bias)

  - The neurons' receptive field is the same in all feature maps of the given layer



Computing the output of a neuron in a convolutional layer

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ is the output of the neuron located in row i, column j in **feature map *k*** of the convolutional layer (layer l).
- s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer $l-1$).
- $x_{i',j',k'}$ is the output of the neuron located in layer $l-1$, row i', column j', feature map k' (or channel k' if the previous layer is the input layer).
- b_k is the bias term for feature map k (in layer l). You can think of it as a knob that tweaks the overall brightness of the feature map k.
- $w_{u,v,k',k}$ - connection weight between any neuron in feature map k of the layer l and its input located at row u, column v (relative to the neuron's receptive field), and feature map k'.

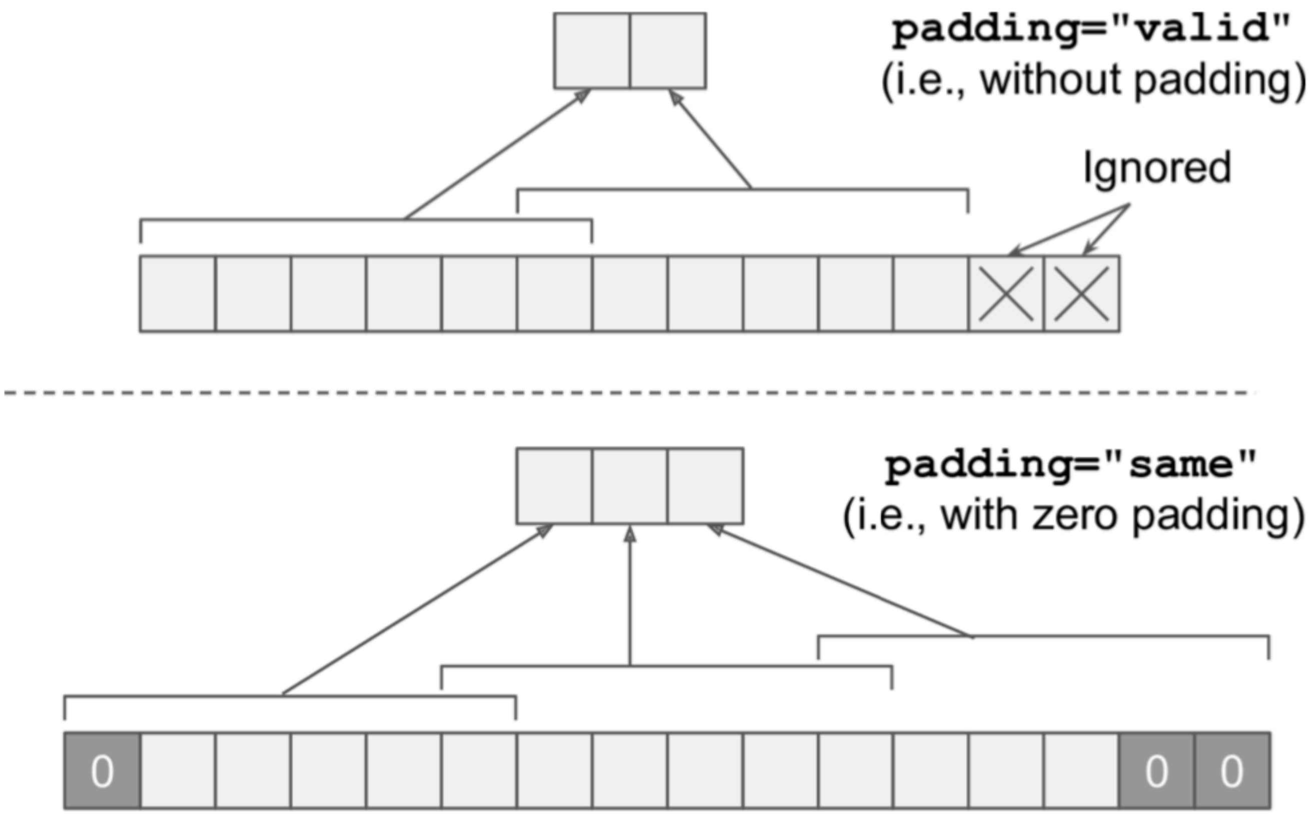
TensorFlow Implementation

- Each input image is typically represented as a 3D tensor of shape [*height, width, channels*].
- A mini-batch is represented as a 4D tensor of shape [*mini-batch size, height, width, channels*].
- The weights of a convolutional layer are repre- sented as a 4D tensor of shape $[f_h,\ f_w,\ f_{n'}, f_n]$. The bias terms of a convolutional layer are simply represented as a 1D tensor of shape $[f_n]$.

Example – applying filters to an image

There are many hyper-parameters to choose - # filters, their height & width, the stride, padding type. Cross-validation is too time consuming…

The meaning of option “padding” in the main call tf.nn.conv2d():



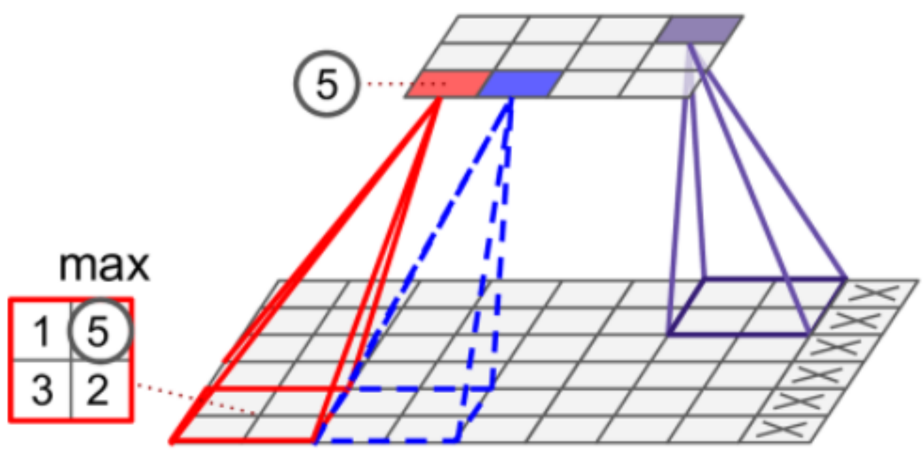
For “SAME”, the output size = round-up (“# of input neurons/ stride”)

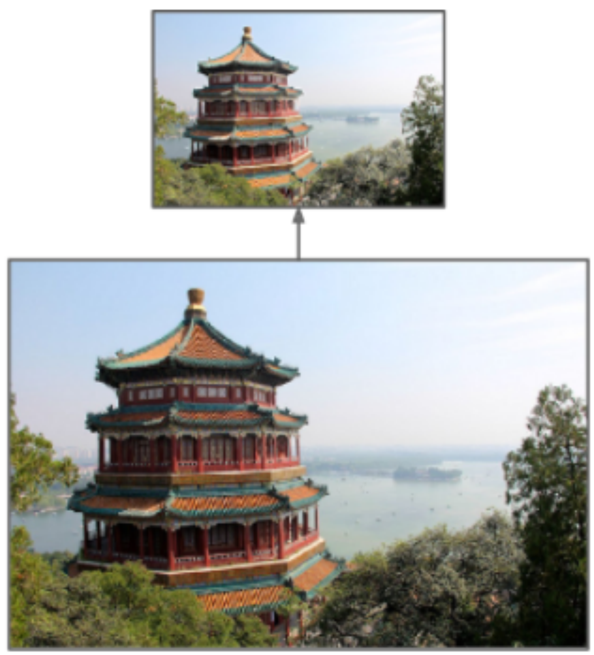
Memory Requirements

- Consider a conv layer - 5×5 filters, outputting 200 feature maps of size 150×100, with stride 1 and "same" padding.
- If the input is a 150×100 RGB image (3 channels) =>
  # parameters = $(5 \times 5 \times 3 \ + \ 1) \times 200 \ = \ 15{,}200$ which is fairly small compared to a fully connected layer ($675M$ parameters)
- Each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 \ = \ 75$ inputs, total of 225M float multiplications. Still computationally intensive.
- Moreover, if the feature maps are represented using 32-bit floats, then the convolutional layer's output will occupy 200 × 150 × 100 × 32 = 96 million bits (12 MB) of RAM.
- And that's just for one instance—if a training batch contains 100 instances, then this layer will use up 1.2 GB of RAM!
- And this volume leads to another way to reduce computations

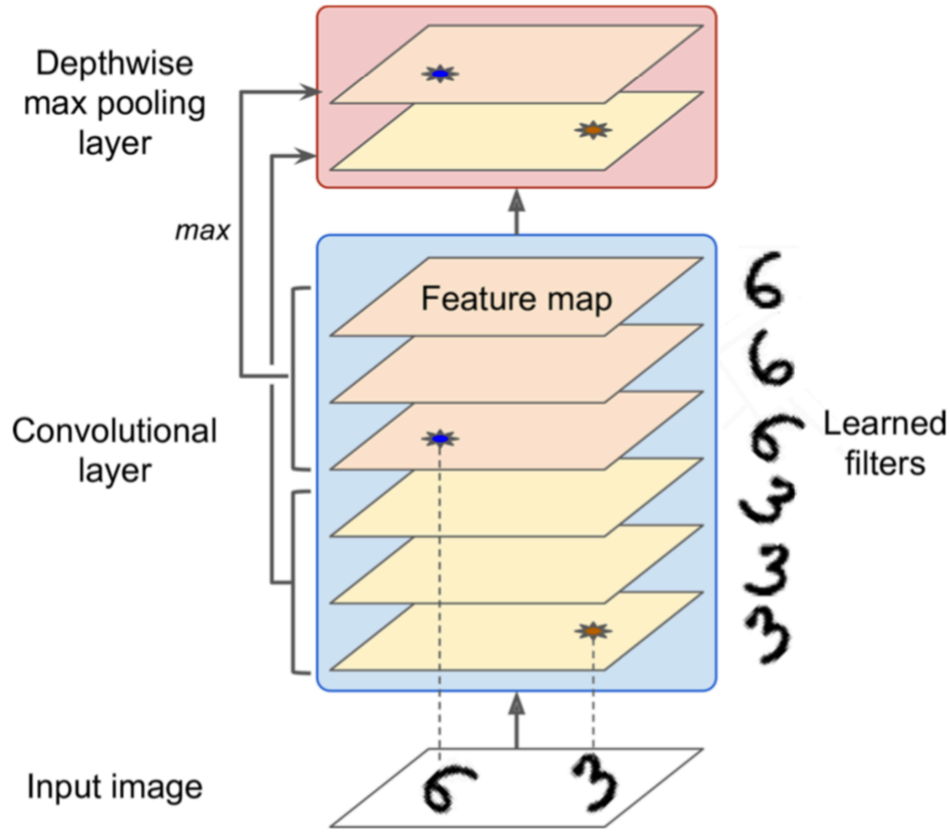
Pooling/sub-sampling layer

- They subsample (shrink) the input image
- It applies some kind of aggregation (max / average) to the connected neurons (no weights!)
- Pooling layers typically operate on each input channel independently, so the depth of the layer doesn't change
- A max pooling layer with a 2x2 kernel and stride of 2 and no padding:



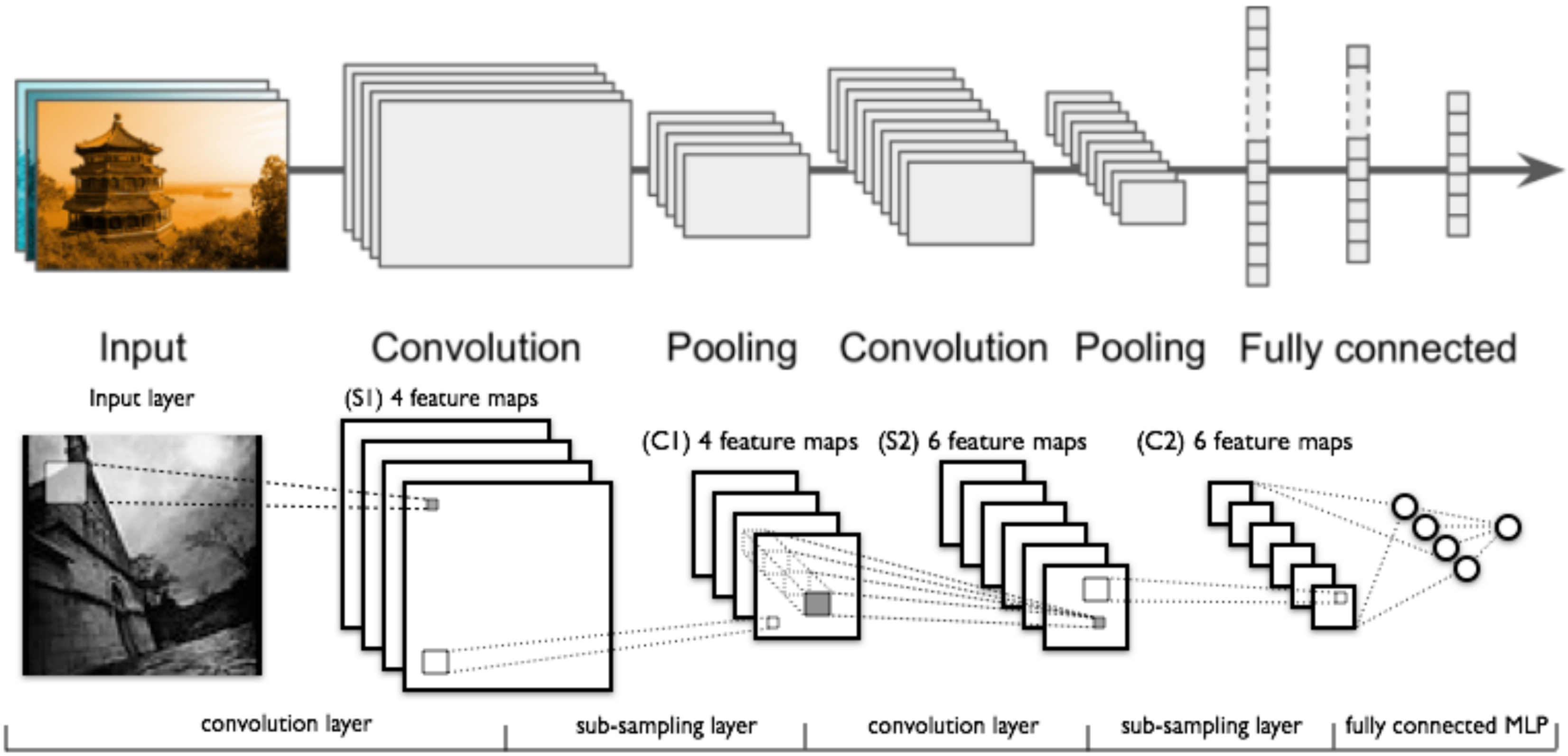


- Depth-wise pooling is not common but can allow the CNN to become invariant to various features, e.g. to rotation



- CNN architectures stack multiple convolutional layers (with ReLU) and pooling layers on top of each other

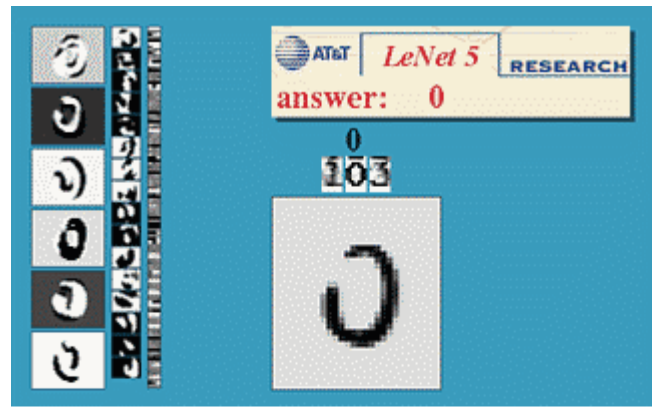
- At the end, you usually have a fully connected network



CNN architectures

- LeNet-5 architecture (1998):

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

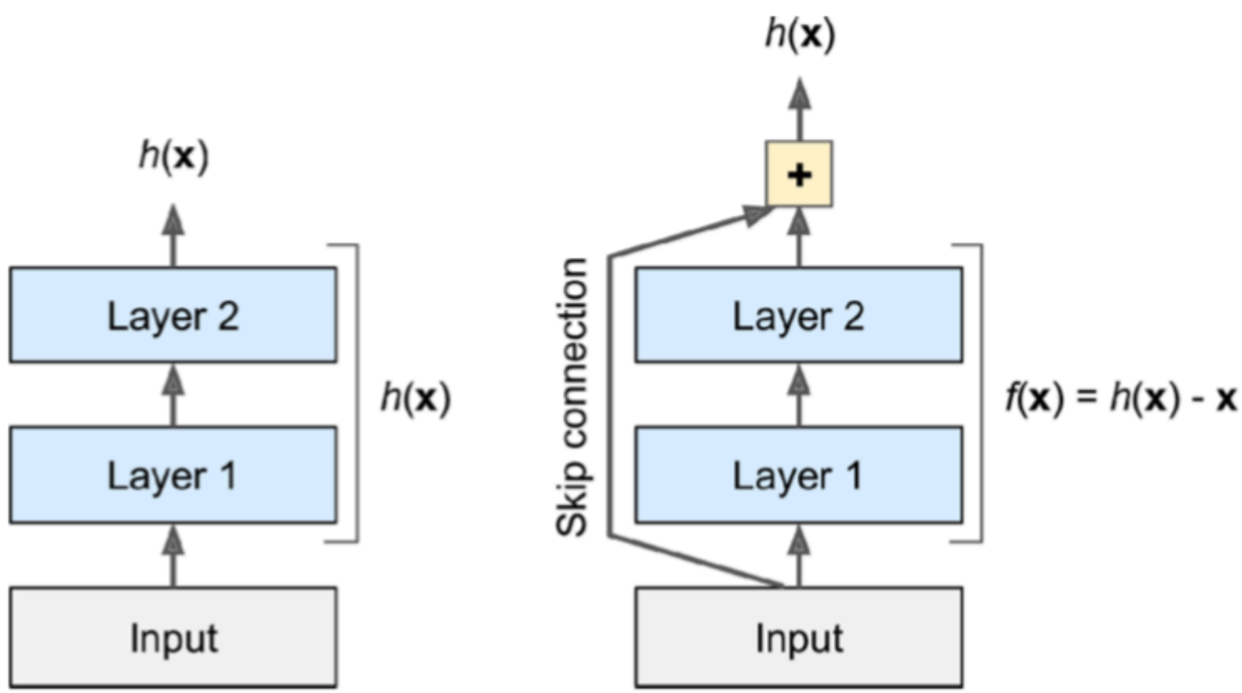


- Input is 0-padded (from 28x28 to 32x32), no padding afterwards
- Avg. pooling: multiplies the result by a learnable coefficient (one per map) and adds a learnable bias term (again, one per map), then applies the activation function

- AlexNet architecture (2012) - ImageNet challenge 17% top-5 error rate (read)
- ResNet – 2015 winner in ILSVRC challenge, only 3.6% top-5 error rate
- The winning variant used an extremely deep CNN composed of 152 layers
- It confirmed the general trend: models are getting deeper and deeper, with fewer and fewer parameters.
- The key to being able to train such a deep network is to use skip connections (also called shortcut connections): the signal feeding into a layer is also added to the output of a layer located a bit higher up the stack.

- When training a neural network, the goal is to make it model a target function $h(\boldsymbol{x})$. If you add the input $\boldsymbol{x}$ to the output of the network (i.e., you add a skip connection), then the network will be forced to model

$$f(x) = h(x) - x \text{ rather than } h(x).$$



- When you initialize a regular neural network, its weights are close to 0, so the network just outputs values close to zero.
- If you add a skip connection, the resulting network just outputs a copy of its inputs; in other words, it initially models the identity function. If the target function is fairly close to the identity function (which is often the case), this will speed up training considerably.

Pretrained Models for Transfer Learning

Classification and Localization

Object Detection

Semantic Segmentation