Chapter 06 – CNNs and ResNets

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Invariants

We can make Deep Feed-Forward Networks arbitrarily complex and very difficult to optimize. If we could take an advantage for finding structural similarities in images, that would help us in exploiting these problems. This is done through various steps of Computer Vision, such as:

- Self-similarity: Data tends to be similar across the domain (equal patches in an image)
- Translation invariant: Translations do not change the image content
- Deformation invariance: We perform some perturbation on the image
- Hierarchy, compositionality & invariance to partially isometric deformations

2. Convolutional Neural Networks

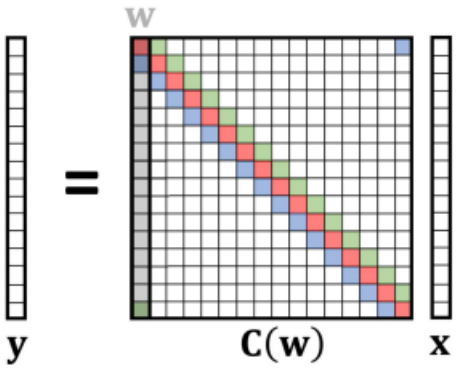
In CNN, we define the convolutional operator between a function f and g.

$$(f \star g)(x) = \int_{-\pi}^{\pi} f(t)\, g(x-t)\, dt$$

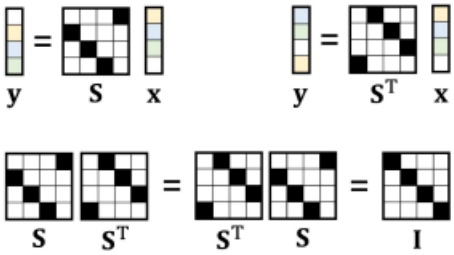
We also introduce the Circulant Matrix $C(\boldsymbol{w})$ which basically is a linear operator between two vectors x and w.

This matrix is formed by stacking shifted (modulo n) versions of $\boldsymbol{w}$.

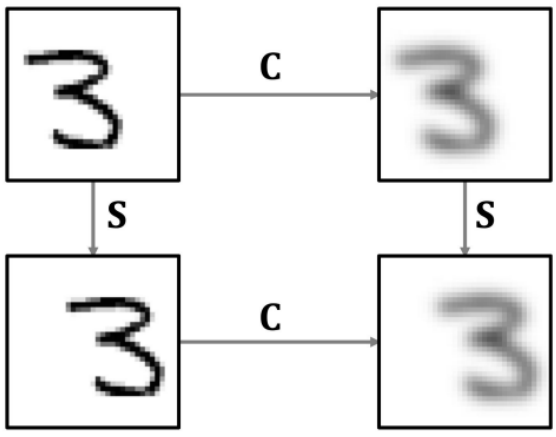
In general, $C(\boldsymbol{u})C(\boldsymbol{w}) = C(\boldsymbol{w})C(\boldsymbol{u})$ (Commutative)



A particular choice of **w** = [0,1,0…,0] yields a special circulant matrix that shifts vectors to the right by one position.



Also, we can say that it is shift equivariant:



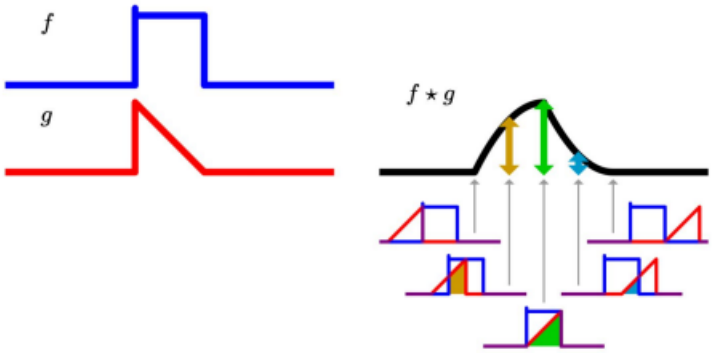
The convolution $x \star w$ can be computed in two ways:

1. As a circulant matrix $C(\boldsymbol{w})$ applied to $\boldsymbol{x}$ in the original spatial domain
2. In the Fourier basis ("Spectral domain") by first computing the Fourier transform of $\Phi^* x$, multiplying it by the Fourier transform of $\boldsymbol{w}$ and then computing the inverse Fourier Transform Φx.

Talking now about Feature Maps and kernels, we defined them in this way:

Given two functions f, g : [−π, π] → R their convolution is a function:

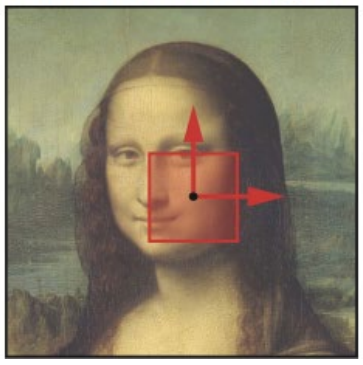
$$\underbrace{(f \star g)(x)}_{\text{feature map}} = \int_{-\pi}^{\pi} f(t)\, \underbrace{g(x-t)}_{\text{kernel}}\, dt$$



- On 2D domains (e.g. RGB images f : R2 → R3), for each channel:

$$(\mathbf{f} \star \mathbf{g})[m,n] = \sum_k \sum_\ell \mathbf{f}[k,\ell]\, \mathbf{g}[m-k, n-\ell]$$

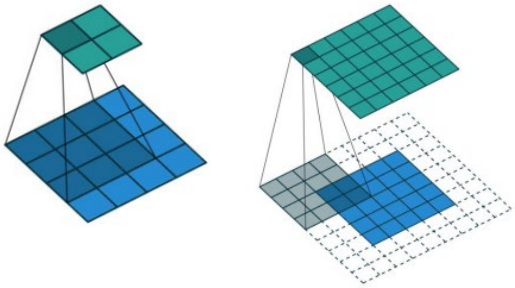
- We can interpret this as a sort of moving windows **f** over the vector **g**.



The same thing happens in 3D where we slide the kernel in a 3D space.

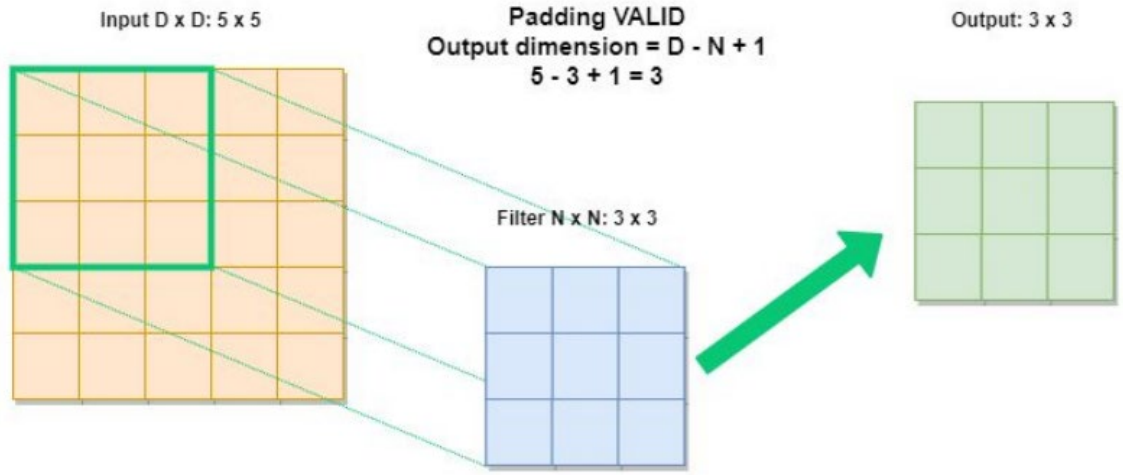
Also padding takes place in case we do not respect boundary conditions (kernel slips out of the image).

- No padding = Kernel directly applied on top of the image.
- Full padding = The domain (image) is enlarged and padded with zeros and the convolution kernel is applied within the larger boundaries.

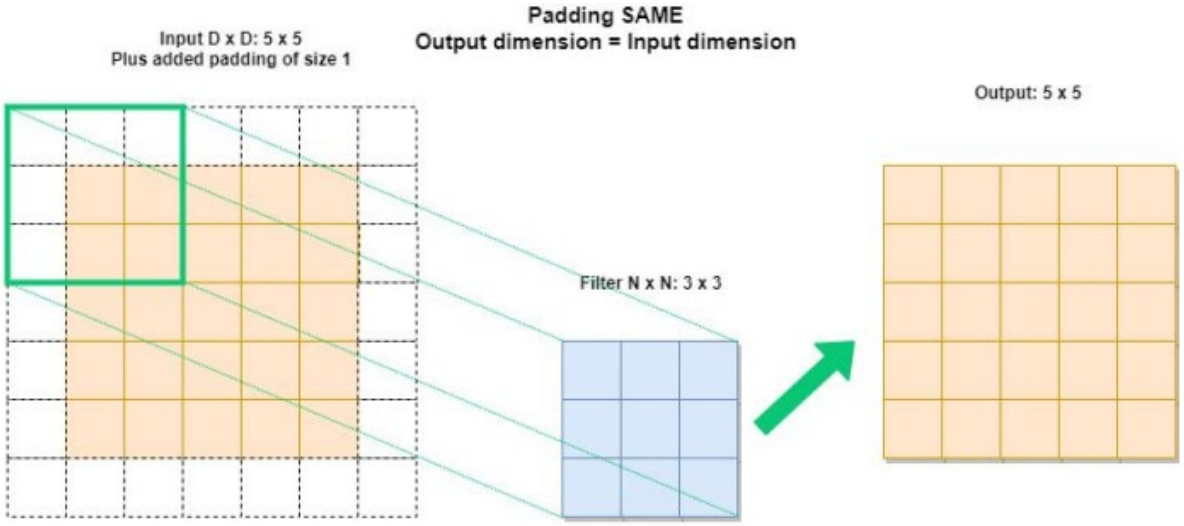


We can distinguish also between Valid Padding and Same Padding:

- **VALID:**



- **SAME:**



3. 1D Convolutional Networks

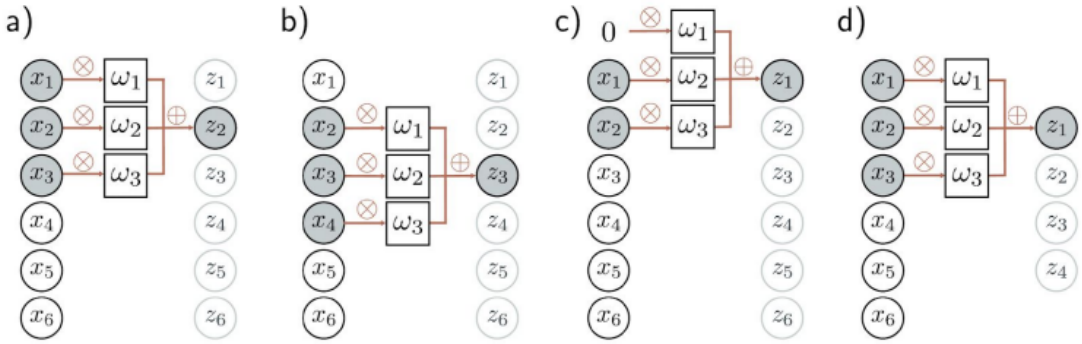


Figure 10.2 1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]$. a) Output z_2 is computed as $z_2 = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$. b) Output z_3 is computed as $z_3 = \omega_1 x_2 + \omega_2 x_3 + \omega_3 x_4$. c) At position z_1, the kernel extends beyond the first input x_1. This can be handled by zero padding, in which we assume values outside the input are zero. The final output is treated similarly. d) Alternatively, we could only compute outputs where the kernel fits within the input range ("valid" convolution); now, the output will be smaller than the input.

The convolutional layer computes its output by convolving the input, adding a bias term β and passing each result through an activation function $a[\cdot]$. With:

- Kernel size: 3x3
- Stride: 1
- Dilation rate: 0

The i-th hidden unit h_i would be computed as:

$$h_i = a[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}] =$$

$$= a\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-2}\right]$$

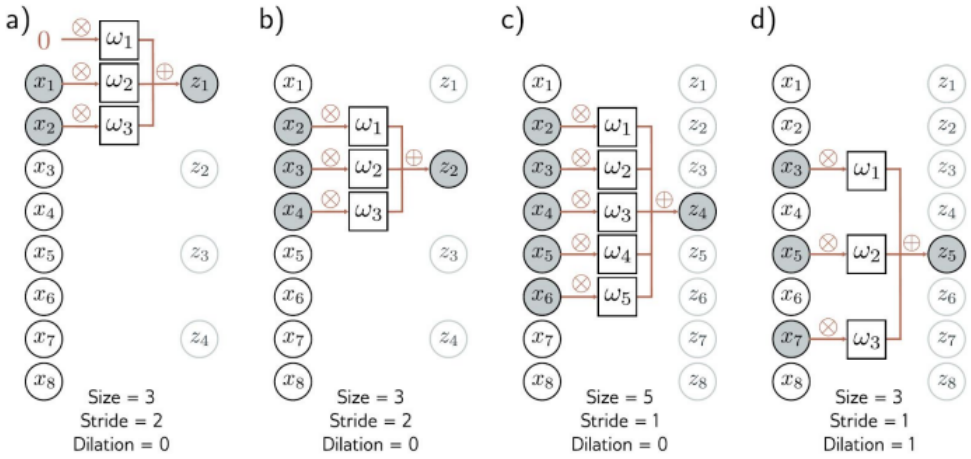


Figure 10.3 Stride, kernel size, and dilation. a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can also be changed. With a kernel size of five, we take a weighted sum of the nearest five inputs. d) In dilated or atrous convolution, we intersperse zeros in the weight vector to allow us to combine information over a large area using fewer weights.

Each convolution produces a new set of hidden variables, called feature maps or channels. In general, the input and the hidden layers all have multiple channels.

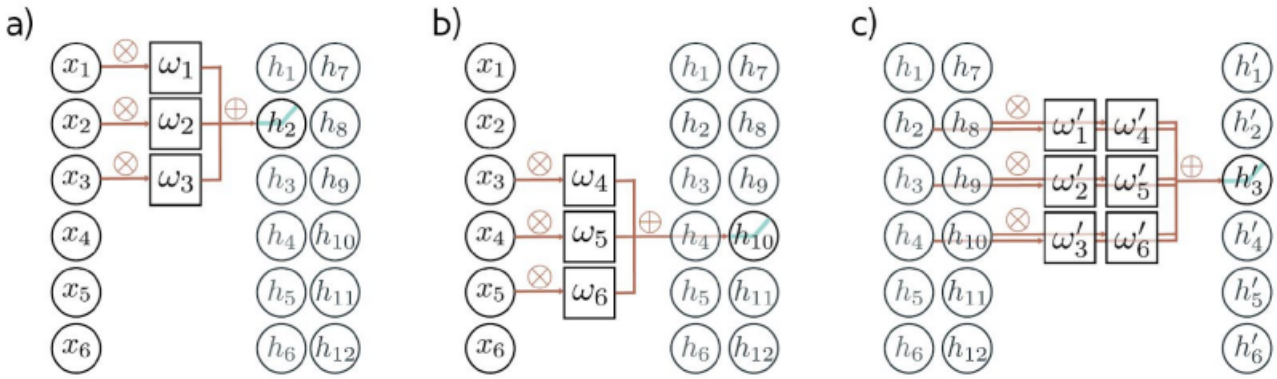
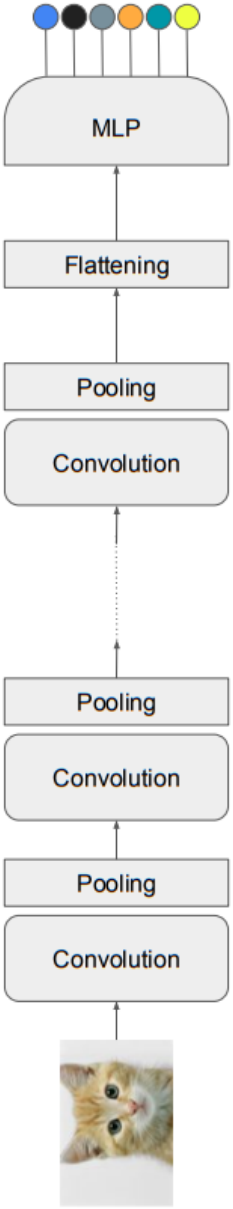
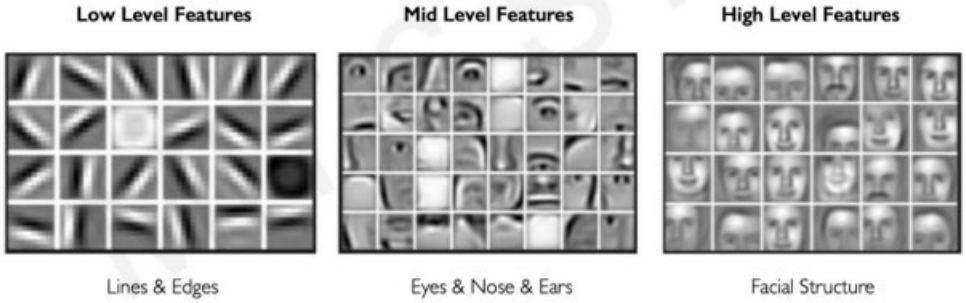
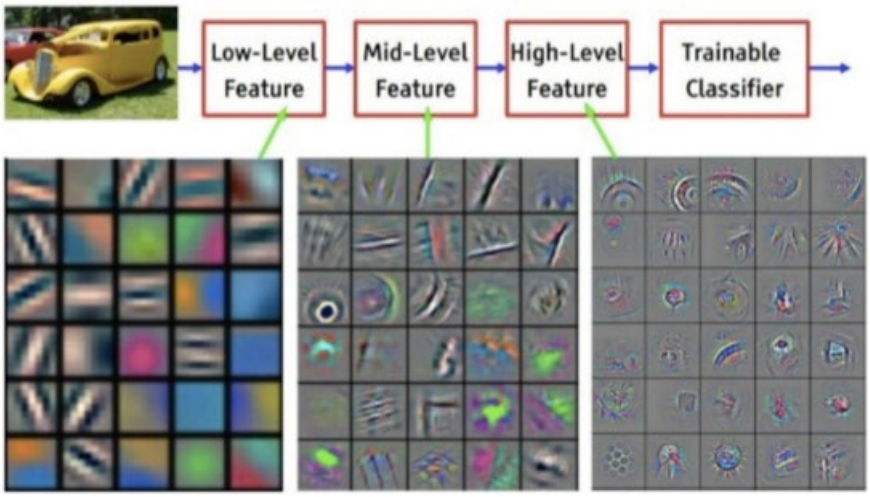


Figure 10.5 Channels. Typically, multiple convolutions are applied to the input $\mathbf{x}$ and stored in channels. a) A convolution is applied to create hidden units h_1 to h_6, which form the first channel. b) A second convolution operation is applied to create hidden units h_7 to h_{12}, which form the second channel. The channels are stored in a 2D array $\mathbf{H}_1$ that contains all the hidden units in the first hidden layer. c) If we add a further convolutional layer, there are now two channels at each input position. Here, the 1D convolution defines a weighted sum over both input channels at the three closest positions to create each new output channel.

4. Convolutional Networks Structure



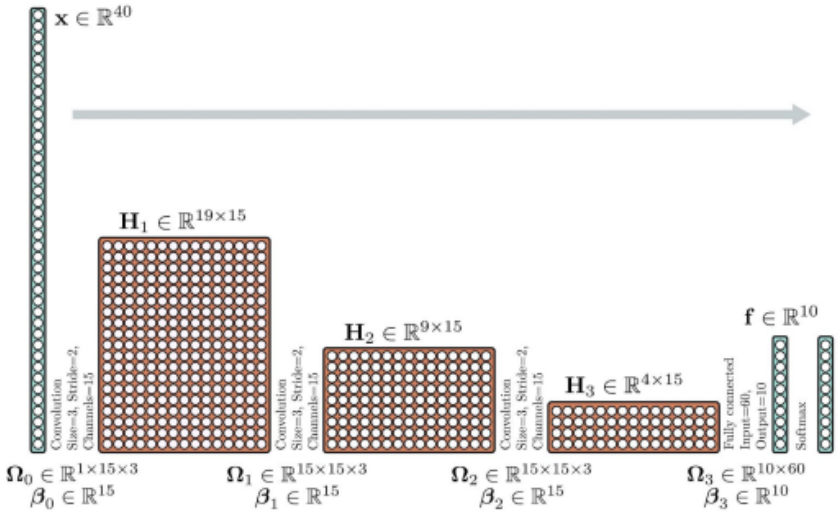




4.1 CNN on MNIST-1D

If we consider applying CNN on MNIST-1D Data, we know that the input x is a 40D vector (40x1) and the output f is a 10D vector (10x1) that is passed through a SoftMax Layer to produce class probabilities.

The network used in this example on the right uses three hidden layers. In $\boldsymbol{H}_1$, the fifteen channels are each computed using a kernel size of (*3x3*) and a stride of *2* with padding "Valid", giving *19* spatial positions.



4.1.1 Results

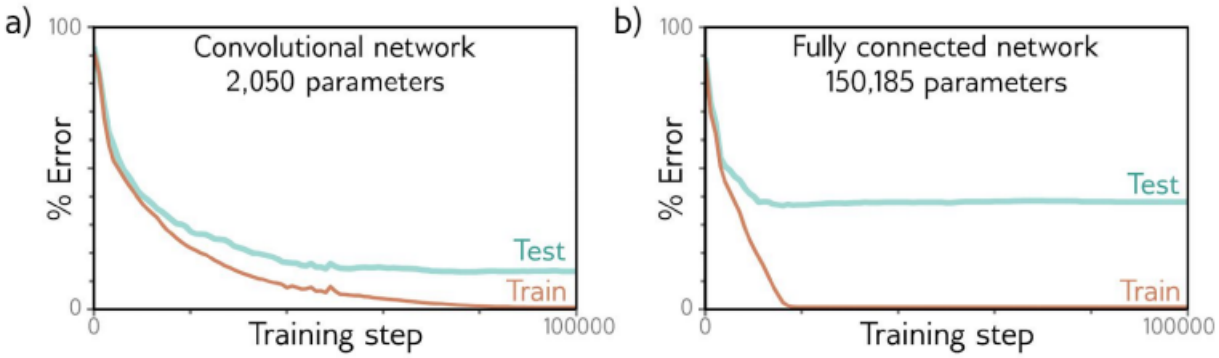


Figure 10.8 MNIST-1D results. a) The convolutional network from figure 10.7 eventually fits the training data perfectly and has ∼17% test error. b) A fully connected network with the same number of hidden layers and the number of hidden units in each learns the training data faster but fails to generalize well with ∼40% test error. The latter model can reproduce the convolutional model but fails to do so. The convolutional structure restricts the possible mappings to those that process every position similarly, and this restriction improves performance.

5. 2D Inputs

Usually, CNNs are applied to 2D image data.

The convolutional kernel is now a 2D object, which is substantially a 3x3 kernel $\Omega \in \mathbb{R}^{3x3}$ applied to a 2D input. Applying this kernel for elements x_{ij} results in a single layer hidden unit h_{ij} computed as:

$$h_{ij} = a\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3} \omega_{mn} x_{i+m-2,j+n-2}\right]$$

Where ω_{mn} are the entries of the convolutional kernel. Zero padding is also present in this 2D version.

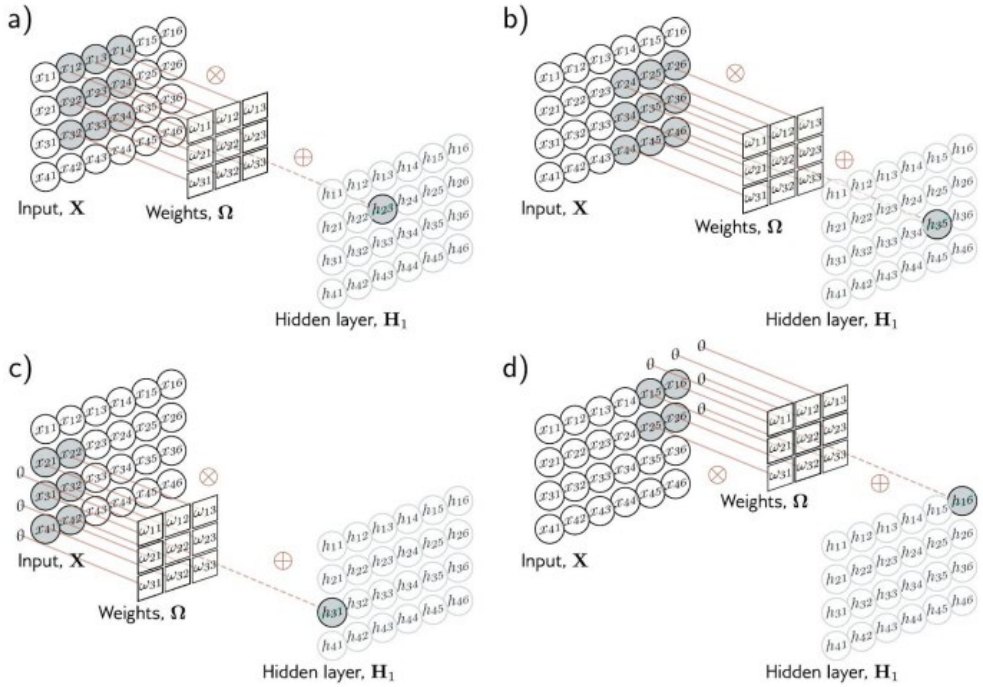


Figure 10.9 2D convolutional layer. Each output h_{ij} computes a weighted sum of the 3×3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here, the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different outputs are computed by translating the kernel across the image grid in two dimensions. c–d) With zero padding, positions beyond the image's edge are considered to be zero.

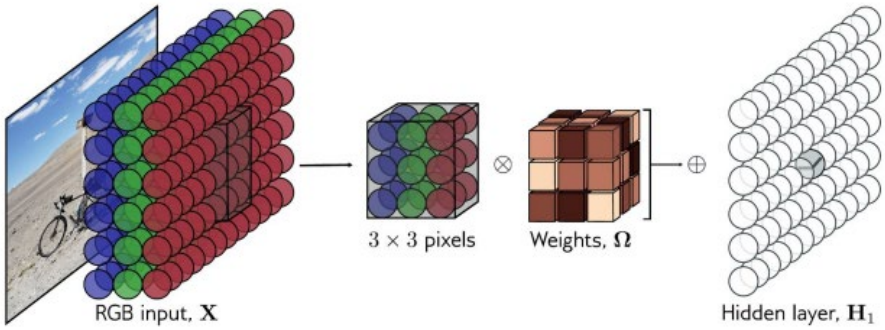


Figure 10.10 2D convolution applied to an image. The image is treated as a 2D input with three channels corresponding to the red, green, and blue components. With a 3×3 kernel, each pre-activation in the first hidden layer is computed by pointwise multiplying the 3×3×3 kernel weights with the 3×3 RGB image patch centered at the same position, summing, and adding the bias. To calculate all the pre-activations in the hidden layer, we "slide" the kernel over the image in both horizontal and vertical directions. The output is a 2D layer of hidden units. To create multiple output channels, we would repeat this process with multiple kernels, resulting in a 3D tensor of hidden units at hidden layer $\mathbf{H}_1$.

The calculations for the number of parameters, we have the following scheme:

- Assuming an RGB image (2D with 3 channels)
- Kernel: 3x3

Then we would have 3x3x3 weights and biases applied to the three different channels at each of the 3x3 position to create a 2D output.

To generate multiple output channels, we repeat this process with different kernel weights and append the results to form a 3D tensor.

So, each output channel is a weighted sum of $C_i \; x \; K \; x \; K + one \; bias$ where C_i is the input channel, $K \; x \; K$ is the dimension of the kernel.

It follows that to compute C_o output channels, we need $C_i \; x \; C_o \; x \; K \; x \; K$ weights and C_o biases.

Talking about downsampling, **max_pooling** retains the maximum of the 2x2 input values, while **avg_pooling** averages the inputs.

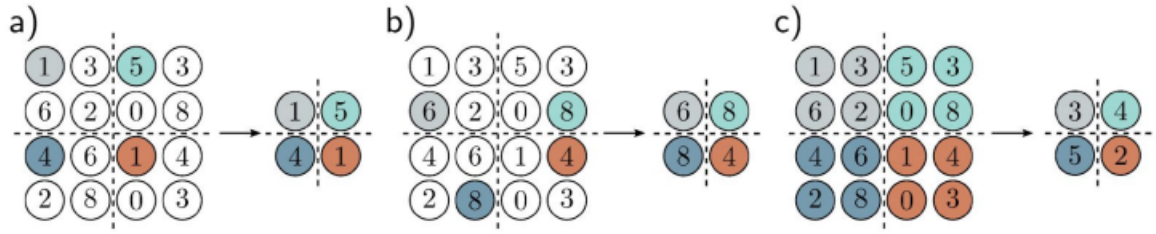


Figure 10.11 Methods for scaling down representation size (downsampling). a) Sub-sampling. The original 4×4 representation (left) is reduced to size 2×2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens with a kernel of stride two, except that the intermediate values are never computed. b) Max pooling. Each output comprises the maximum value of the corresponding 2×2 block. c) Mean pooling. Each output is the mean of the values in the 2×2 block.

Upsampling instead, duplicates all the channels at each spatial position M times, or we perform **unpooling** (distribute the values to the position they originated from) or **bilinear interpolation** (filling missing values between points).

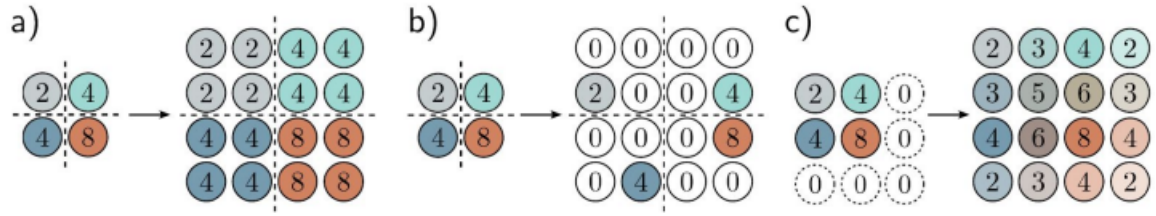


Figure 10.12 Methods for scaling up representation size (upsampling). a) The simplest way to double the size of a 2D layer is to duplicate each input four times. b) In networks where we have previously used a max pooling operation (figure 10.11b), we can redistribute the values to the same positions they originally came from (i.e., where the maxima were). This is known as max unpooling. c) A third option is bilinear interpolation between the input values.

6. Applications

As for applications, we say that Convolutions are needed for Image Classification (Think about ImageNet through AlexNet, VGG), Object Detection, Semantic Segmentation.

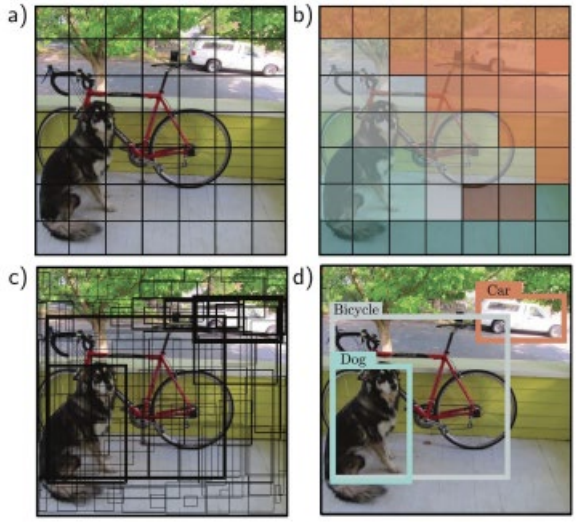
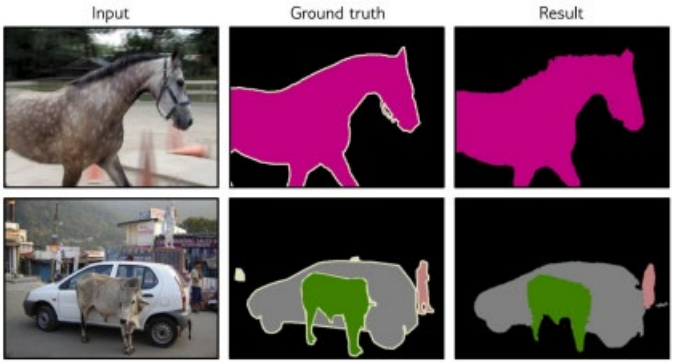


Figure 10.18 YOLO object detection. a) The input image is reshaped to 448×448 and divided into a regular 7×7 grid. b) The system predicts the most likely class at each grid cell. c) It also predicts two bounding boxes per cell, and a confidence value (represented by thickness of line). d) During inference, the most likely bounding boxes are retained, and boxes with lower confidence values that belong to the same object are suppressed. Adapted from Redmon et al. (2016).



In summary, we have fewer parameters than in a fully connected network and they do not increase with input image size. Typical convolutions consist in 2 Convolutional Layers where we downsample by a factor of two (Max/Avg Pooling).

More depth -> More efficiency.

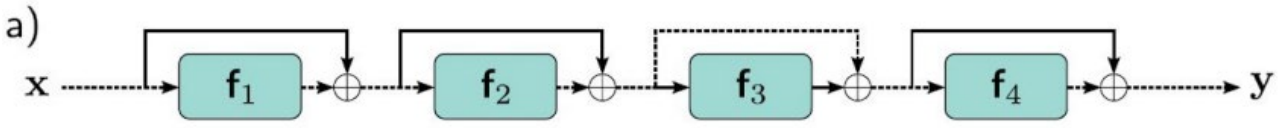
7. Residual Blocks

We know that standard Neural Networks pass the output of each layer directly into the next layer. We can think of this network as a series of nested functions:

$$\mathbf{y} = \mathbf{f}_4\Big[\mathbf{f}_3\Big[\mathbf{f}_2\big[\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\big], \boldsymbol{\phi}_3\Big], \boldsymbol{\phi}_4\Big]$$

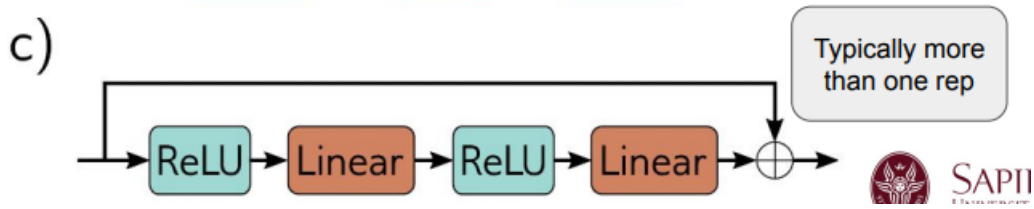
Residual blocks basically are self-explanative:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4]
\end{aligned}$$



In fact, between f2 and f3 for example, we would add up the output of f1.

In terms of a proper non-linear NN:



In general residual blocks are helpful for networks, helping in achieving better results. Usually, we do use Batch Normalization in order to stop the exponentially increase of the gradients (Adaptive reparameterization), motivated as well by the difficulty of training very deep models.

The reparameterization significantly reduces the problem of coordinates updates cross many layers. It can be applied to any input or hidden layer in a network.

Let H be a minibatch of activations of the layer to normalize, arranged as a design matrix with activations for each example appearing in a row of the matrix, which could be in this form:

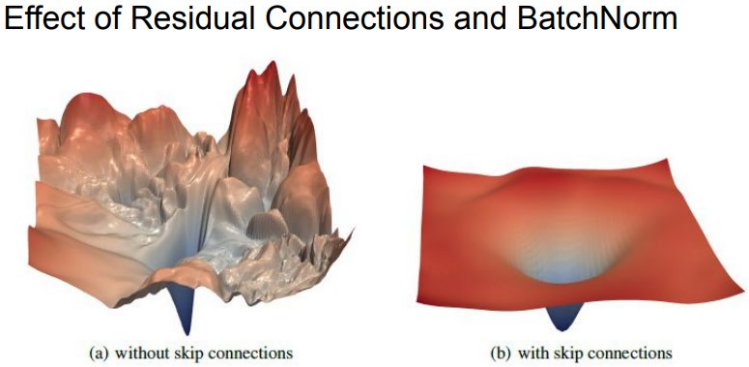
$$X = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 2 \\ 1 & 0 & 3 \\ 1 & 0 & 4 \\ 1 & 0 & 5 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 1 & 4 \\ 1 & 1 & 5 \end{pmatrix}$$

To normalize it, we replace it with:

$$H' = \frac{H - \mu}{\sigma}$$

Where: μ is a vector containing the mean of each unit and σ is a vector containing the standard deviation of each unit.

We compute the mean and the standard deviation through training (via backpropagation) in order to apply them to H (normalizing it).



8. Architectures

8.1 ResNet

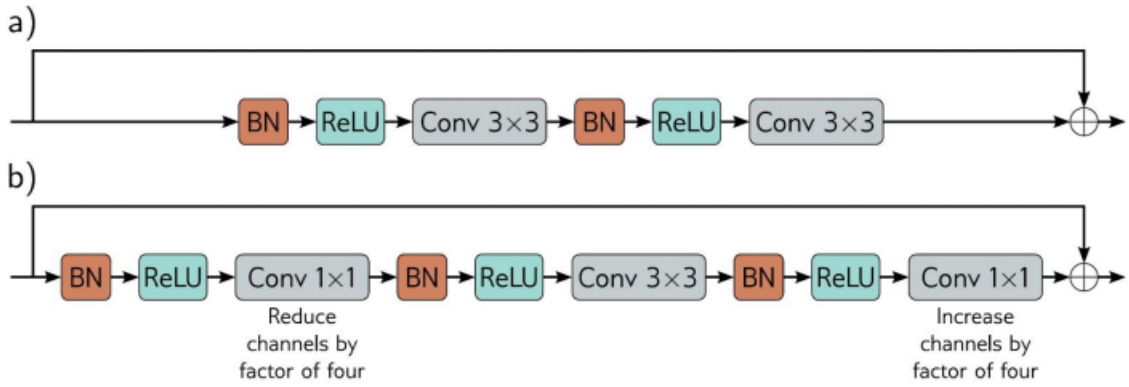


Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses fewer parameters. It contains three convolutions. The first 1×1 convolution reduces the number of channels. The second 3×3 convolution is applied to the smaller representation. A final 1×1 convolution increases the number of channels again so that it can be added back to the input.

8.2 ResNet-200

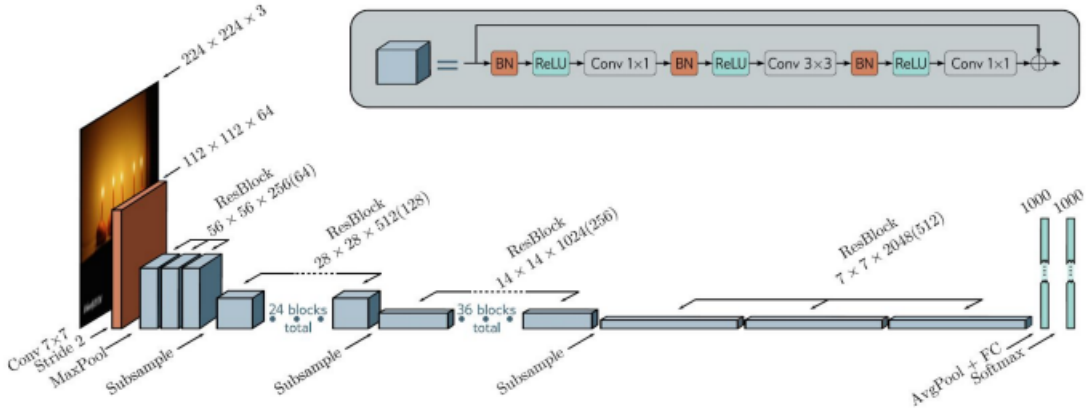


Figure 11.8 ResNet-200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling and accompanying increases in the number of channels. The network concludes with average pooling across all spatial positions and a fully connected layer that maps to pre-softmax activations.

8.3 DenseNet

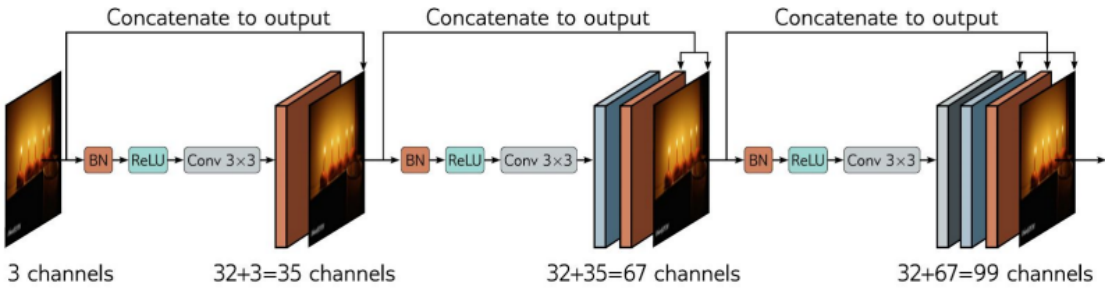


Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation, and both earlier representations are concatenated to this to create a total of 67 channels and so on.

8.4 U-Net

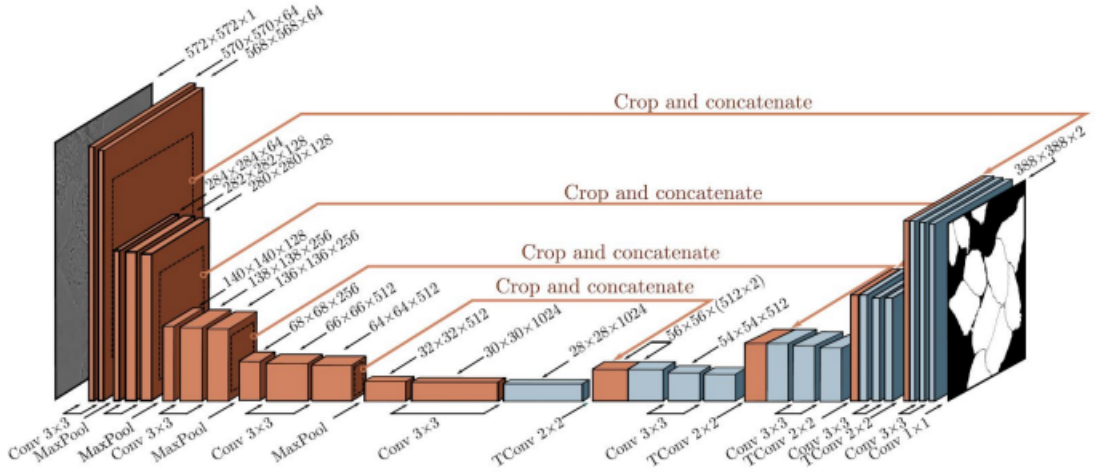
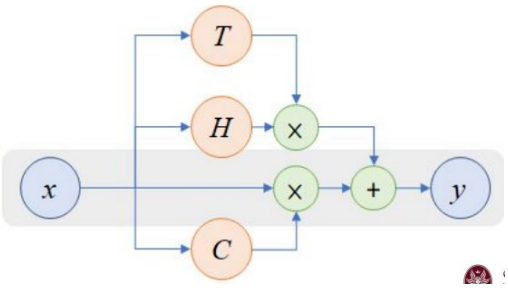


Figure 11.10 U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the last representation at each scale in the encoder to the first representation at the same scale in the decoder (orange arrows). The original U-Net used "valid" convolutions, so the size decreased slightly with each layer, even without downsampling. Hence, the representations from the encoder were cropped (dashed squares) before appending to the decoder. Adapted from Ronneberger et al. (2015).

8.5 Highway Networks

Highway Networks are a generalization of Residual connections, where we have a plain feedforward Neural Network $y = H(x, W_h)$



For a Highway Network, we additionally define two non-linear transform T and C such that:

$$y = H(x, W_h) \cdot T(x, W_t) + x \cdot C(x, W_c)$$

Where T is the transform gate and C is the carry gate, since they express how much of the output is produced by transforming the input and carrying it, respectively.