Deep Learning

06 - Convolutional Neural Networks and Resnets

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri

Recapping

We know what to do to…

- Use an MLP
- Design a Loss
- Train a Neural Network
- Measure Performance
- Try to improve performance
  - Regularization
- We have seen that… MLP are universal approximators but…
  - Too many parameters when dealing with real-world objects, e.g., images

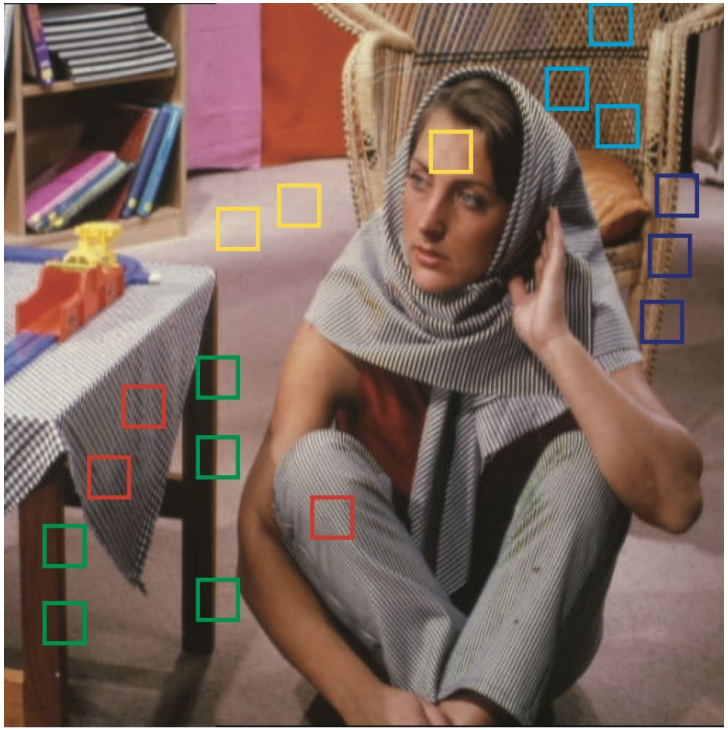
Invariants

The General Idea

- Deep feed-forward networks are provably universal. However:
  - We can make them arbitrarily complex.
  - The number of parameters can be huge.
  - Very difficult to optimize.
  - Very difficult to achieve generalization.
- We can take advantage of structural similarities in images:
  - Self-similarity
  - Translation invariance
  - Deformation invariance
  - Hierarchy and compositionality

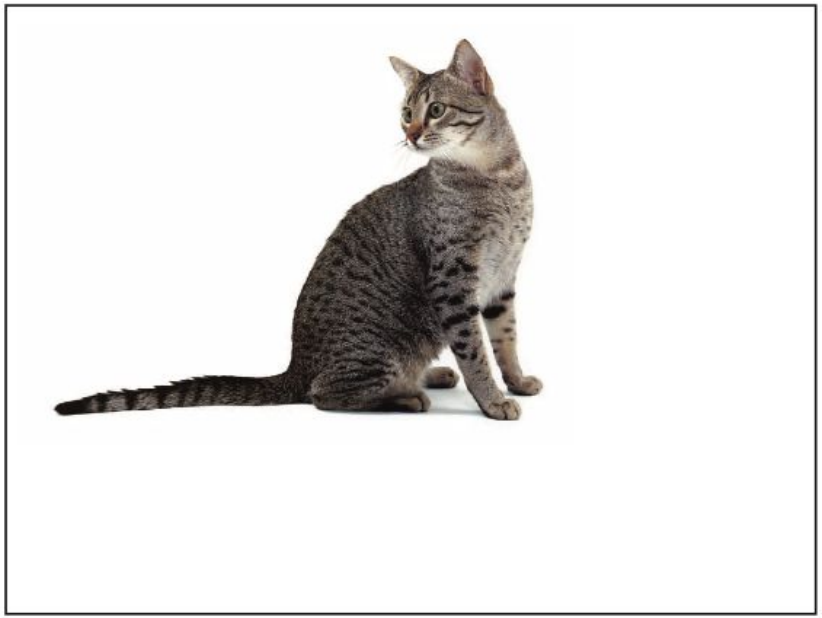
Self-Similarity

Data tends to be self-similar across the domain:

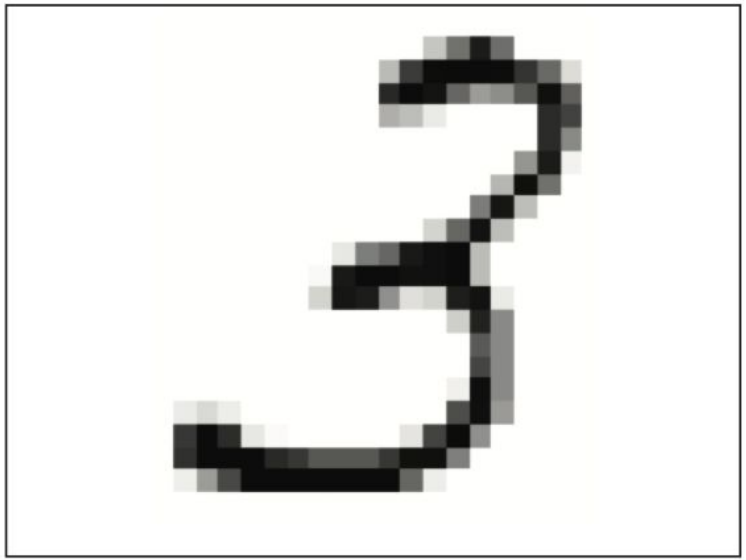


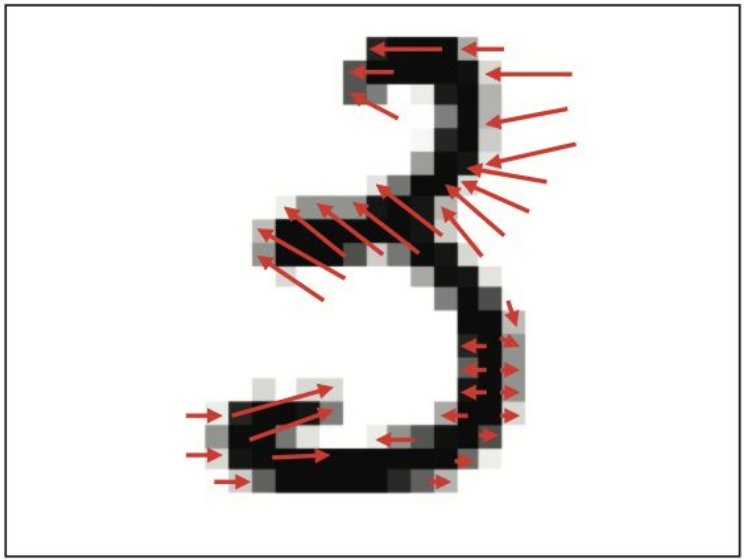
Translation Invariant

Translations do not change the image content.

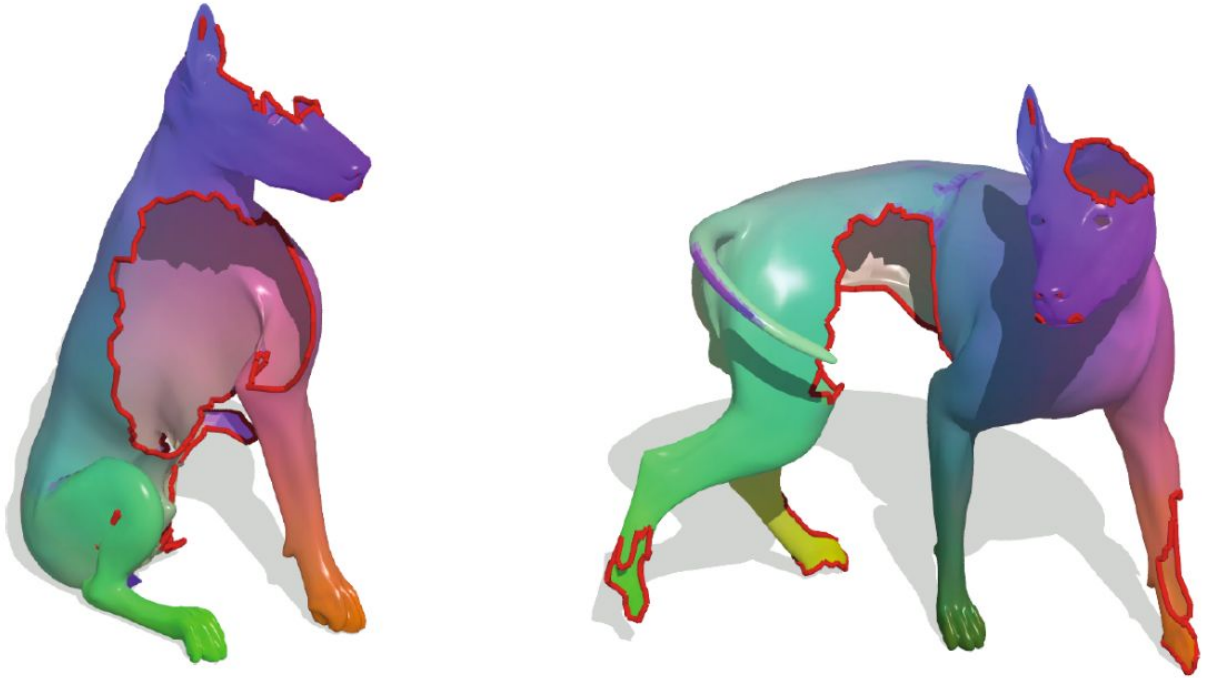


Deformation Invariance

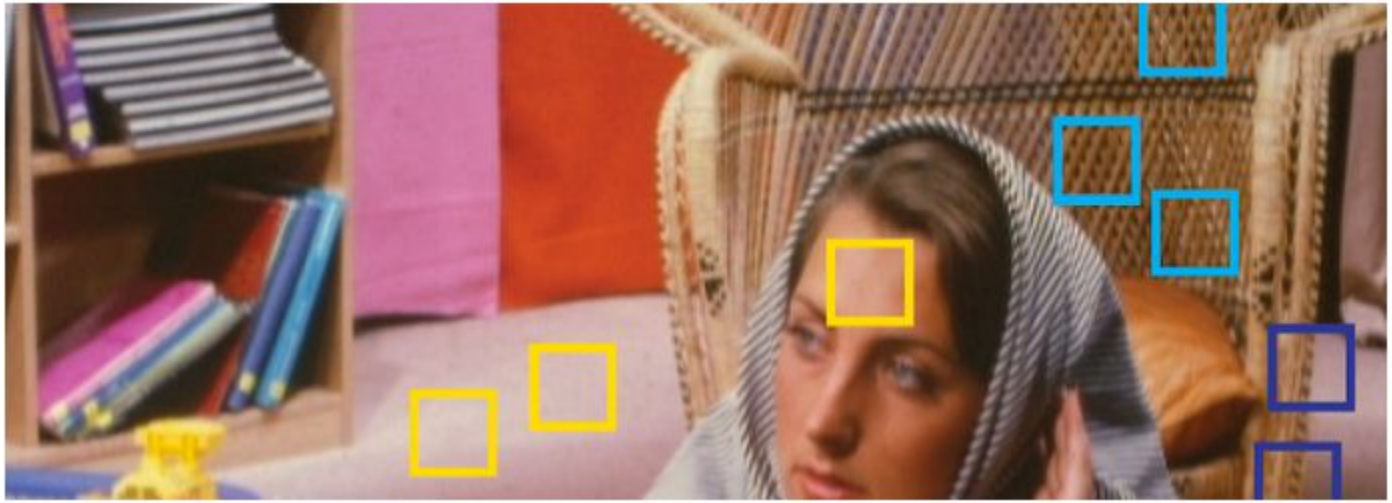


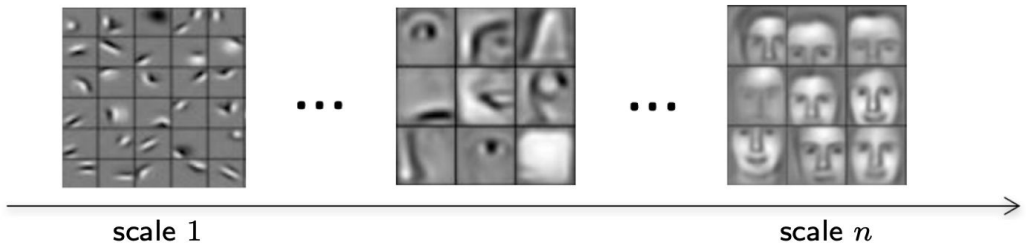


Invariance to partiality and isometric deformations



Hierarchy and compositionality





Convolutional Neural Networks

SAPIENZA
UNIVERSITÀ DI ROMA

Convolution from First Principles

Why Convolutional Neural Networks?

- Data is often composed of hierarchical, local, shift-invariant patterns.
- CNNs directly exploit this fact using a specific inductive bias.

Convolution Operation

- Classical signal theory definition. Given to function **f** and **g**:

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

- (Maybe) lesser known definition → Discrete Time Convolution. Given two vectors **x** and **w**.

$$(\mathbf{x} * \mathbf{w})_i = \sum_{k=0}^{n-1} w_{i-k \bmod n}\, x_k$$

A Geometric View

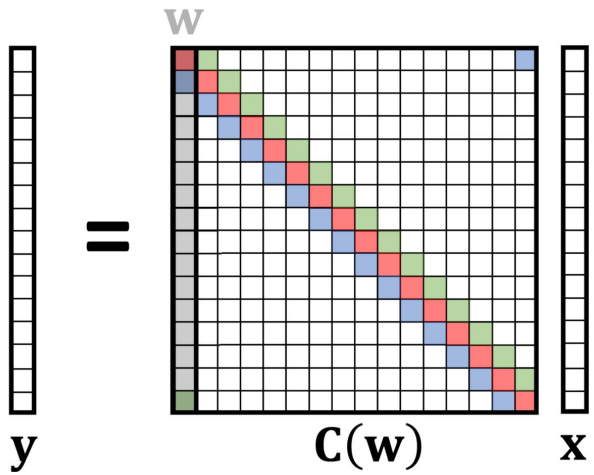
$$(\mathbf{x} * \mathbf{w})_i = \sum_{k=0}^{n-1} w_{i-k \bmod n}\, x_k$$

Can be also viewed as the result of a linear operator C(**w**) applied to vector **x** → y = C(**w**)**x**

What's the structure of C(**w**)?

The structure of C(**w**)

- C(**w**) carries a special structure
  - It is a **circulant** matrix.
  - Circulant matrices have very nice properties…
- C(w) is formed by stacking shifted (modulo n) versions of **w**



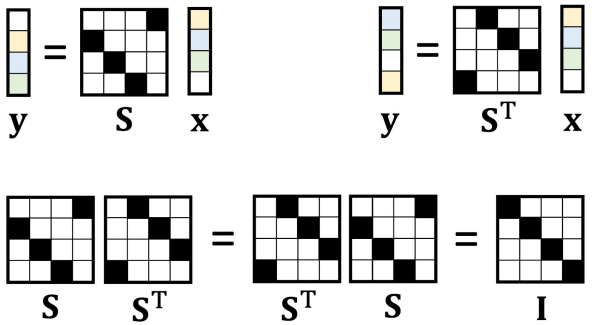
$$(\mathbf{x} * \mathbf{w})_i = \sum_{k=0}^{n-1} w_{i-k \bmod n}\, x_k$$

Circulant Matrices Commute

- In general $AB \neq BA$, but…
- … given two vectors $\mathbf{u}$, and $\mathbf{w}$
- $C(\mathbf{u})C(\mathbf{w}) = C(\mathbf{w})C(\mathbf{u}) \rightarrow$ Convolution is commutative $\mathbf{u} * \mathbf{w} = \mathbf{w} * \mathbf{u}$.
  - Proof left as exercise (spoiler here)

Circulant Matrices and Shift Matrices

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

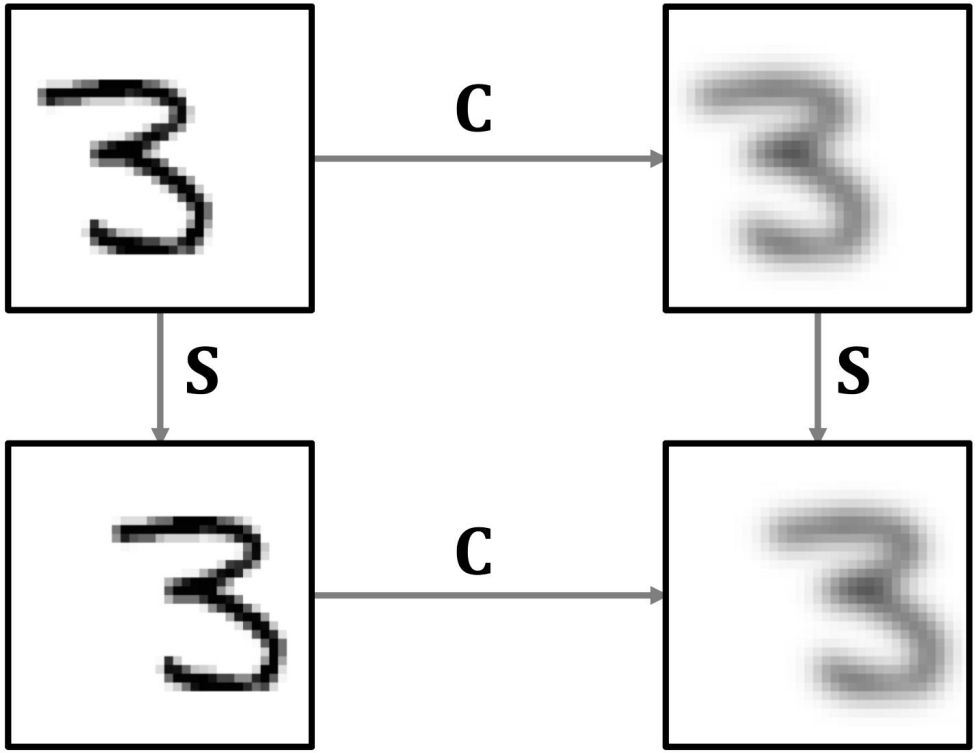


- Shift Matrices are orthogonal
- A matrix is circulant if and only if it commutes with shift.

A matrix is circulant if and only if it commutes with shift

- (⇒) Translation or shift equivariance
- (⇐) convolution as the shift-equivariant linear operation:
  - In order to commute with shift, a matrix must have the circulant structure
- Starting from the requirement that we want an operator with the “shift equivariance” property and we get to convolution
  - Yay!!!

An Illustration of Shift Equivariance

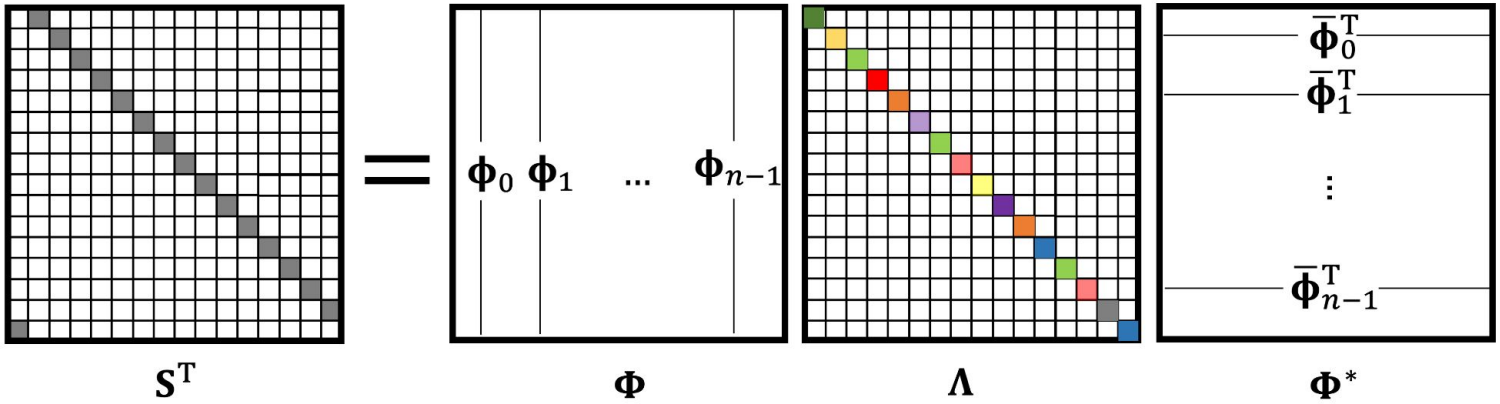


Commuting Matrices are Jointly Diagonalizable

- Two n x n matrices A,B are said to be simultaneously diagonalizable if there is a nonsingular matrix S such that both $S^{-1}AS$ and $S^{-1}BS$ are diagonal matrices.
- Two matrices satisfying AB = BA will be jointly diagonalizable thus, will have the same eigenvectors
  - Possibly different eigenvalues.
- Since all circulant matrices commute, we pick one and we compute the corresponding eigenvectors

Eigenvectors of the Shift Operator

- The shift operator is diagonalized by the Fourier transform.

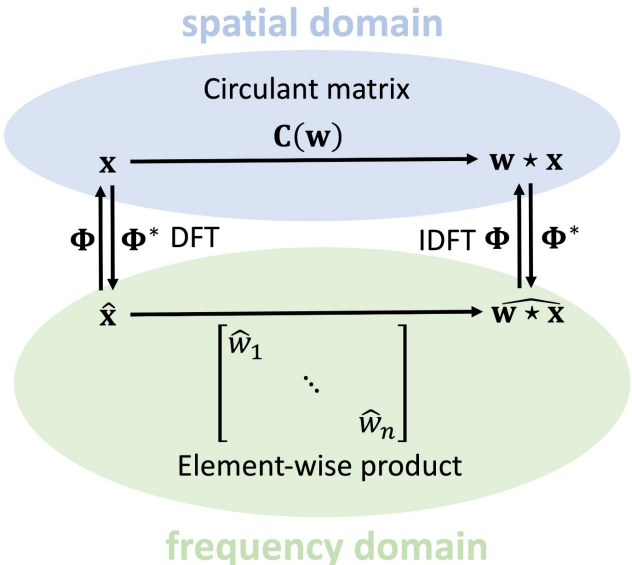


- Therefore the eigenvalues of C(w) is the Fourier transform of w

The Convolution Theorem

- The convolution $\mathbf{x} * \mathbf{w}$ can be computed in two ways:
  a. As a circulant matrix C(**w**) applied to **x** in the original s
  b. In the Fourier basis ("spectral domain") by first compu
     multiplying it by the Fourier transform of **w**, and then c
     $\Phi\mathbf{x}$.

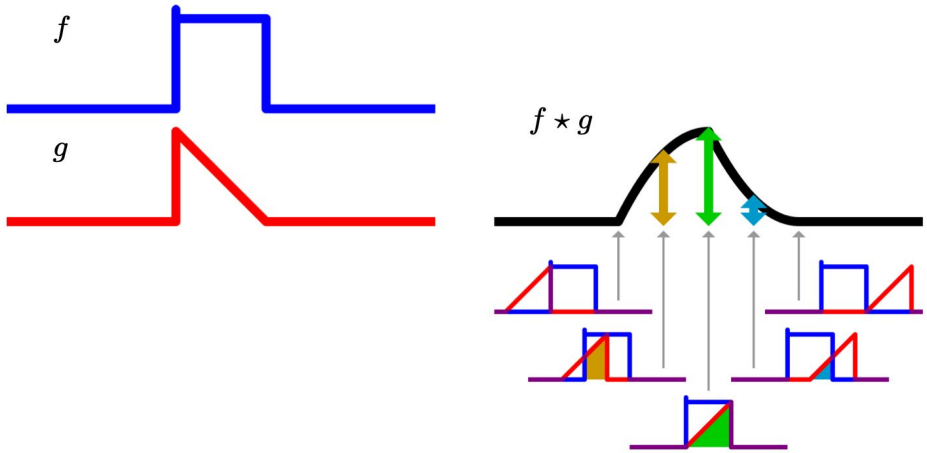
Because Φ has a special redundant structure, the products $\Phi^*\mathbf{x}$ and $\Phi\mathbf{x}$ can be computed with $\mathcal{O}(n \log n)$ complexity using a Fast Fourier Transform (FFT) algorithm.



Feature Map and Kernel

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

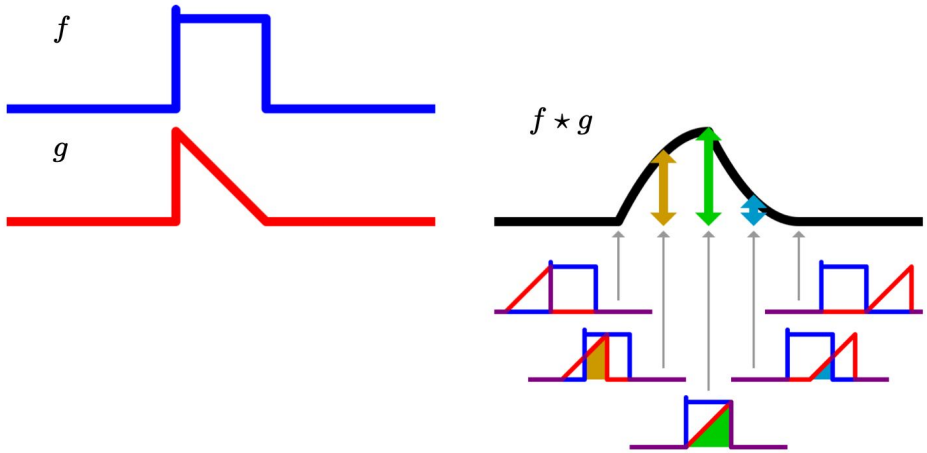
$$(f \star g)(x) = \int_{-\pi}^{\pi} f(t)\, g(x-t)\, dt$$



Feature Map and Kernel

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

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

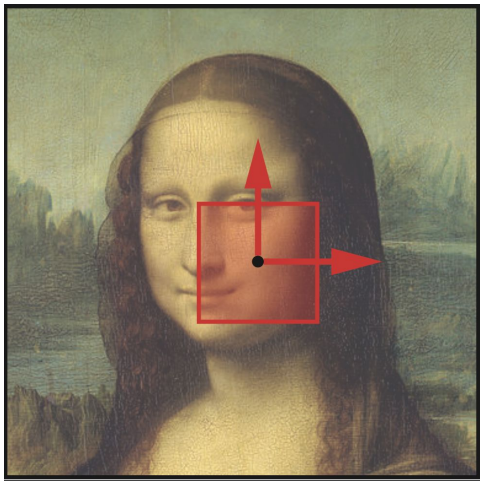


The Discrete Convolution in 2D Domains

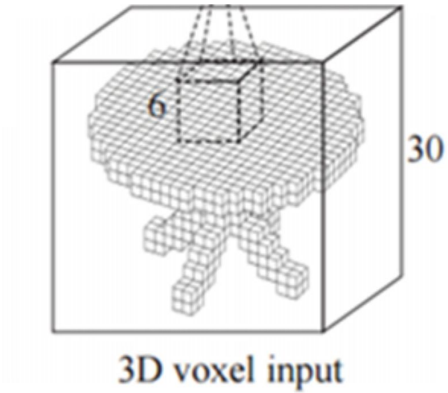
- On 2D domains (e.g. RGB images f : R2 → R3), for each channel:

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

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



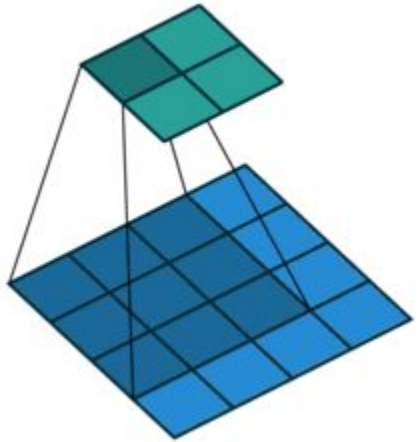
Similarly in 3D



In general, for functions f : $\mathbf{R}^k \rightarrow \mathbf{R}^m$ defined on Euclidean domains, convolution is well-defined up to appropriate boundary conditions.

Boundary Conditions and Stride

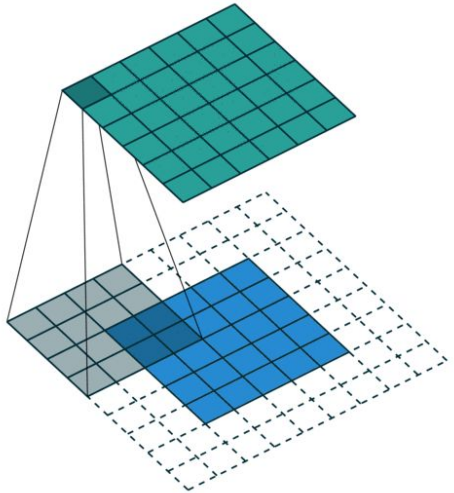
- No padding: The convolution kernel is directly applied within the boundaries of the underlying function (an image in this example).



- The result of the convolution is a smaller image.

Boundary Conditions and Stride

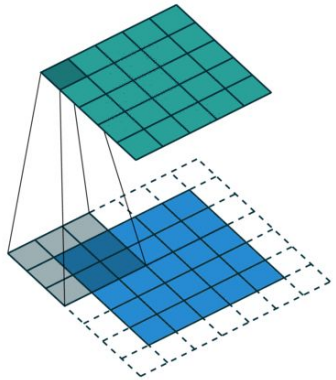
- Full padding: The domain is enlarged and padded with zeros. The convolution kernel is applied within the (now larger) boundaries.



- The result of the convolution is a larger image.

Boundary Conditions and Stride

- Arbitrary zero-padding, with stride: The domain is enlarged and padded with zeros, but not enough to capture the boundary pixels. Further, each discrete step skips one pixel.



- The result is the same as no stride followed by downsampling.

1D ConvNets

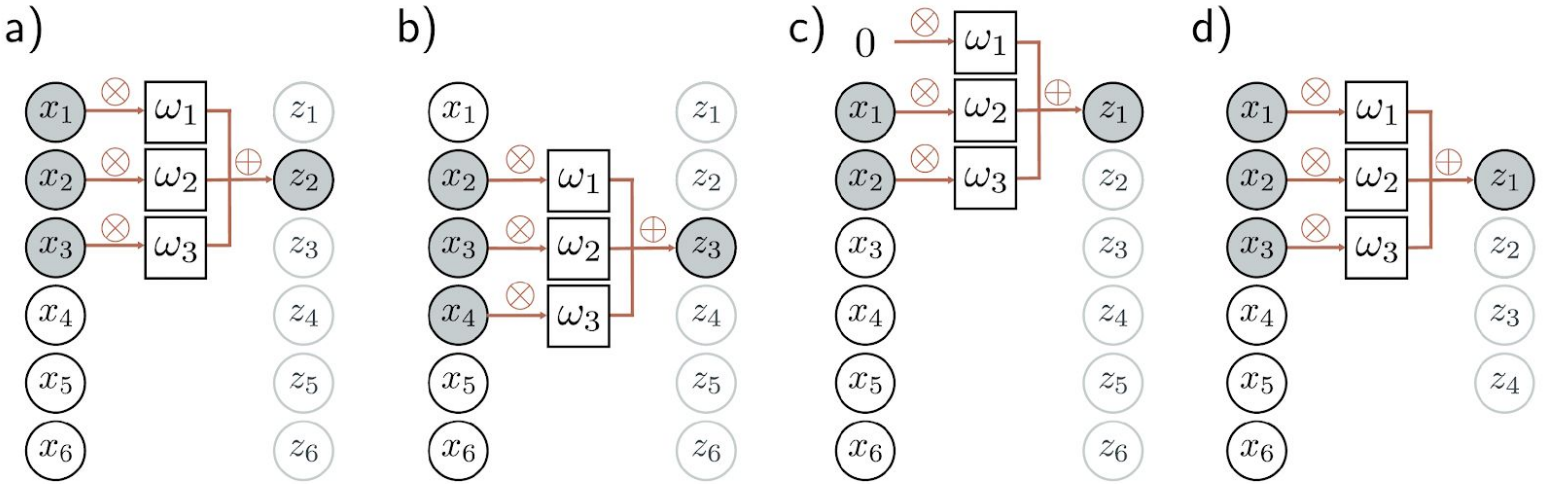


Figure 10.2 1D convolution with kernel size three. Each output z_i is a weighted sum of the nearest three inputs x_{i-1}, x_i, and x_{i+1}, where the weights are $\boldsymbol{\omega} = [\omega_1, \omega_2, \omega_3]$. a) Output z_2 is computed as $z_2 = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3$. b) 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.

Convolution Layer

- A convolutional layer computes its output by convolving the input, adding a bias β, and passing each result through an activation function a[•]. With kernel size three, stride one, and dilation rate zero, the i-th hidden unit h_i would be computed as:

$$\begin{aligned} h_i &= \mathrm{a}\left[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}\right] \\ &= \mathrm{a}\left[\beta + \sum_{j=1}^{3} \omega_j x_{i+j-2}\right], \end{aligned}$$

Convolution Layer

- Where the bias β and kernel weights ω1, ω2, ω3 are trainable parameters, and (with zero padding) we treat the input x as zero when it is out of the valid range. This is a special case of a fully connected layer that computes the i-th hidden unit as:

$$h_i = \mathrm{a}\left[\beta_i + \sum_{j=1}^{D} \omega_{ij} x_j\right]$$

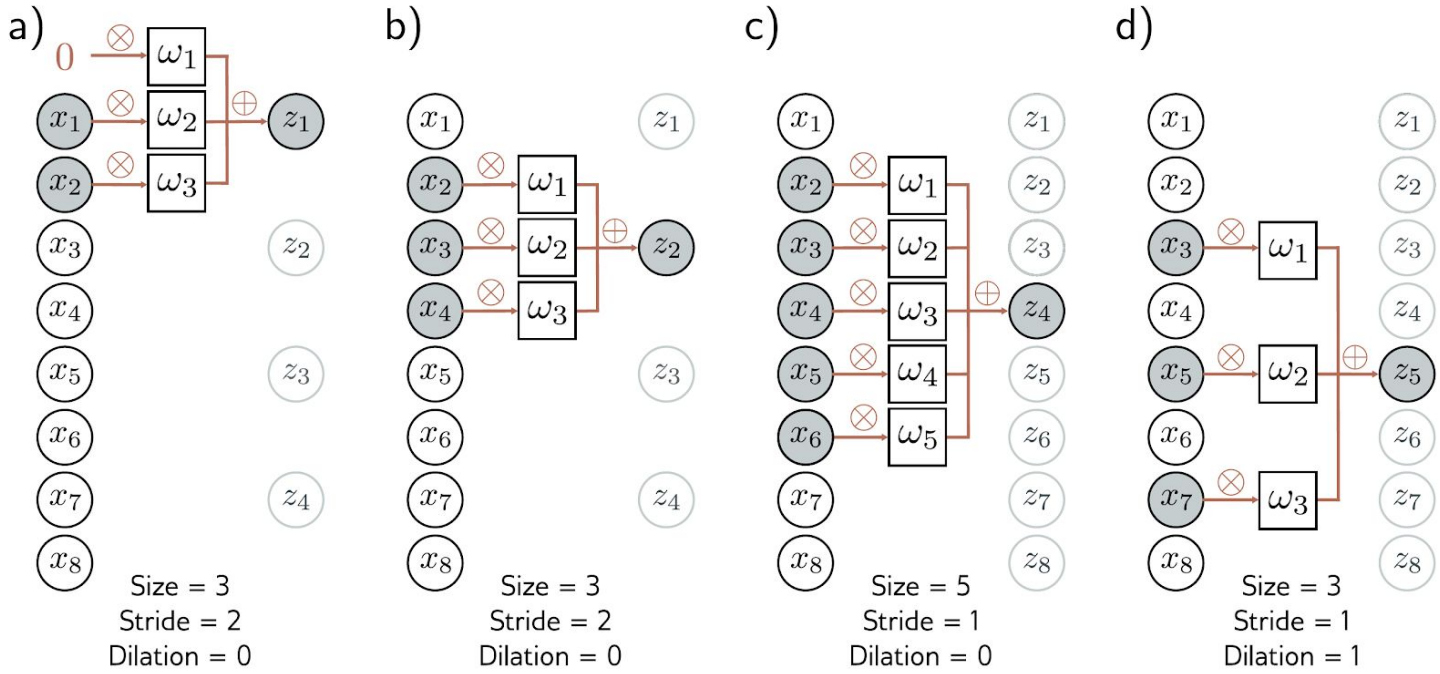


Figure 10.3 Stride, kernel size, and dilation. a) With a stride of two, we evaluate the kernel at every other position, so the first output z_1 is computed from a weighted sum centered at x_1, and b) the second output z_2 is computed from a weighted sum centered at x_3 and so on. c) The kernel size can 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.

Convolution Layer

- If there are D inputs $x_•$ and D hidden units $h_•$, this fully connected layer would have D^2 weights $\omega_{••}$ and D biases $\beta_•$. The convolutional layer only uses three weights and one bias.
- A fully connected layer can reproduce this exactly if most weights are set to zero and others are constrained to be identical

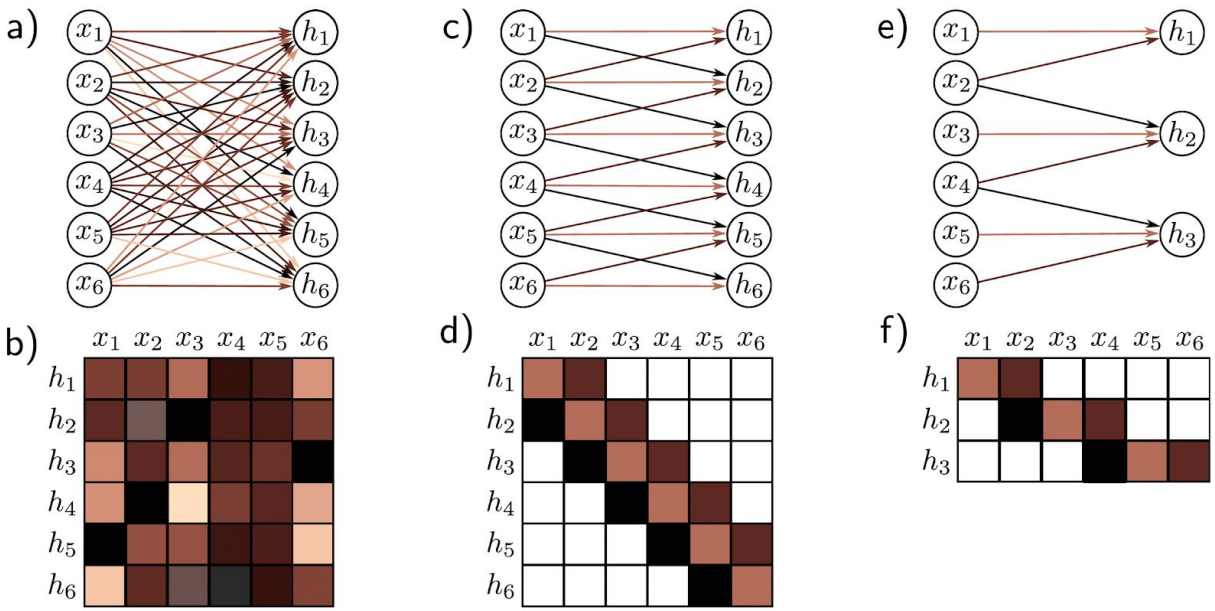


Figure 10.4 Fully connected vs. convolutional layers. a) A fully connected layer has a weight connecting each input x to each hidden unit h (colored arrows) and a bias for each hidden unit (not shown). b) Hence, the associated weight matrix $\mathbf{\Omega}$ contains 36 weights relating the six inputs to the six hidden units. c) A convolutional layer with kernel size three computes each hidden unit as the same weighted sum of the three neighboring inputs (arrows) plus a bias (not shown). d) The weight matrix is a special case of the fully connected matrix where many weights are zero and others are repeated (same colors indicate same value, white indicates zero weight). e) A convolutional layer with kernel size three and stride two computes a weighted sum at every other position. f) This is also a special case of a fully connected network with a different sparse weight structure.

Channels

- It is usual to compute several convolutions in parallel. Each convolution produces a new set of hidden variables, termed a feature map or channel.
- In general, the input and the hidden layers all have multiple channels. If the incoming layer has C_i channels and kernel size K, the hidden units in each output channel are computed as a weighted sum over all Ci channels and K kernel positions using a weight matrix $\Omega \in R^{Ci \times K}$ and one bias.
    - If there are C_o channels in the next layer, then we need $\Omega \in R^{Ci \times Co \times K}$ weights and $\beta \in R^{Co}$ biases.

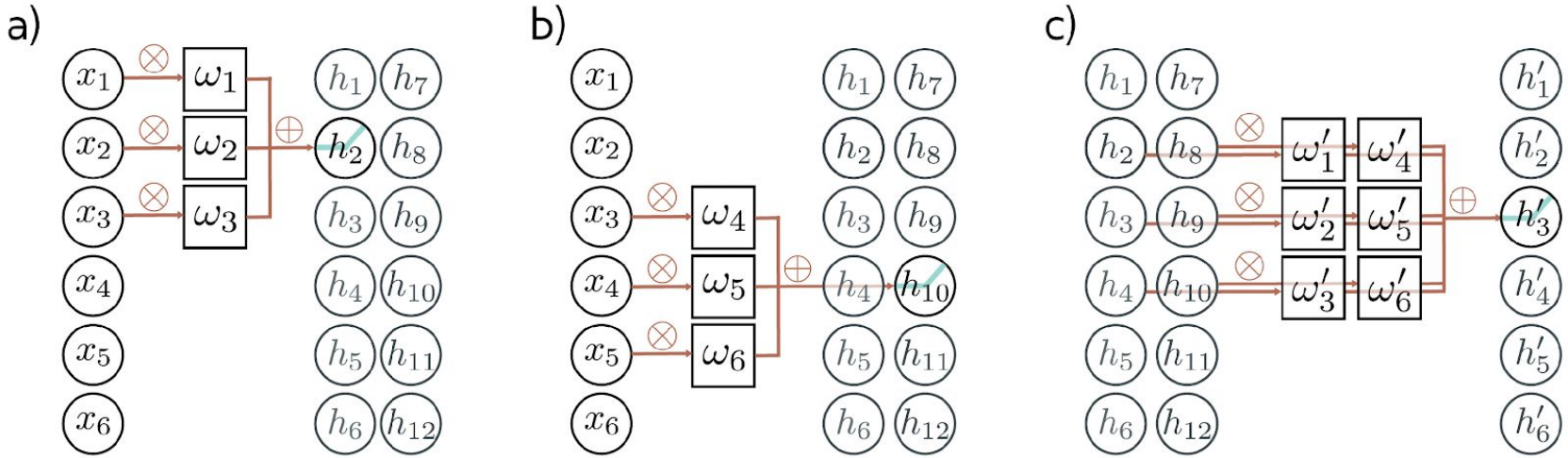


Figure 10.5 Channels. Typically, multiple convolutions are applied to the input $\mathbf{x}$ and stored in channels. a) A convolution is applied to create hidden units h_1 to h_6, which form the first channel. b) A second convolution operation is applied to create hidden units h_7 to h_{12}, which form 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.

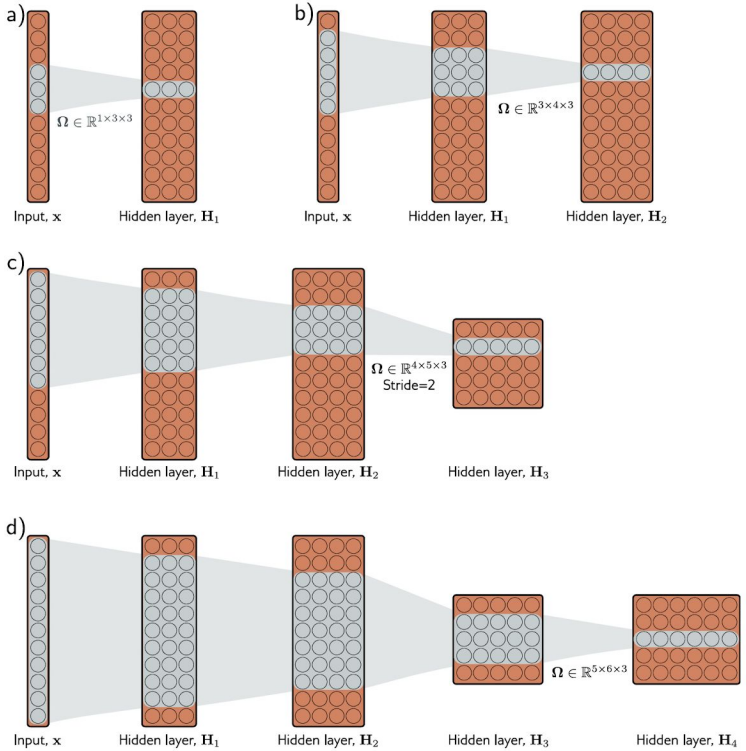
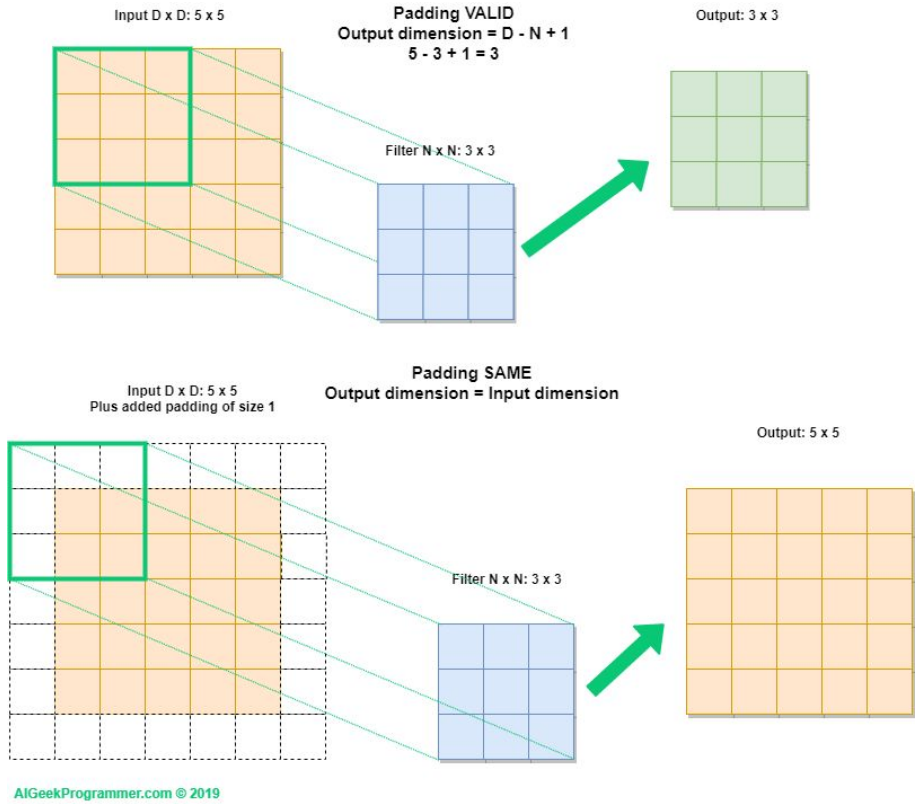
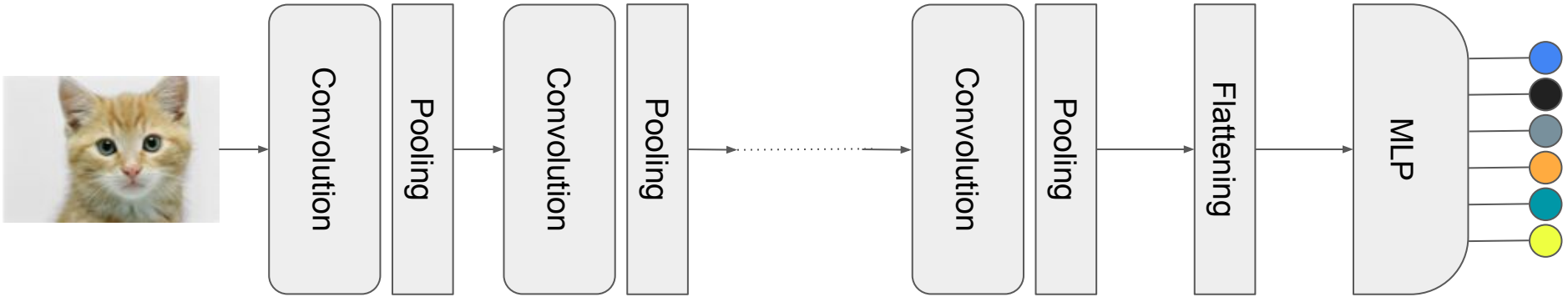


Figure 10.6 Receptive fields for network with kernel width of three. a) An input with eleven dimensions feeds into a hidden layer with three channels and convolution kernel of size three. The pre-activations of the three highlighted hidden units in the first hidden layer $\mathbf{H}_1$ are different weighted sums of the nearest three inputs, so the receptive field in $\mathbf{H}_1$ has size three. b) The pre-activations of the four highlighted hidden units in layer $\mathbf{H}_2$ each take a weighted sum of the three channels in layer $\mathbf{H}_1$ at each of the three nearest positions. Each hidden unit in layer $\mathbf{H}_1$ weights the nearest three input positions. Hence, hidden units in $\mathbf{H}_2$ have a receptive field size of five. c) The hidden units in the third layer (kernel size three, stride two) increases the receptive field size to seven. d) By the time we add a fourth layer, the receptive field of the hidden units at position three have a receptive field that covers the entire input.

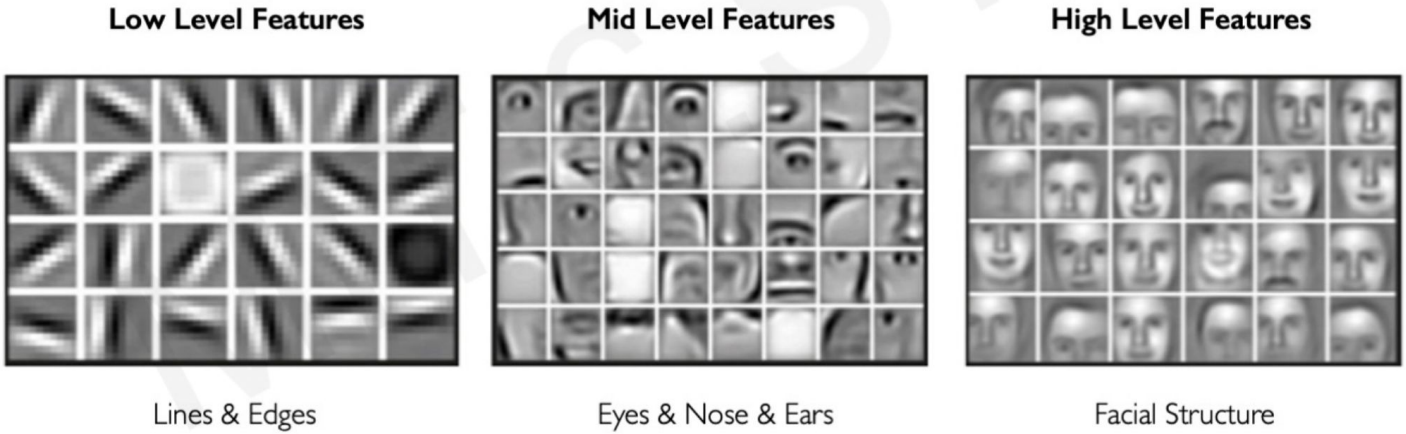
Padding: Valid vs. Same



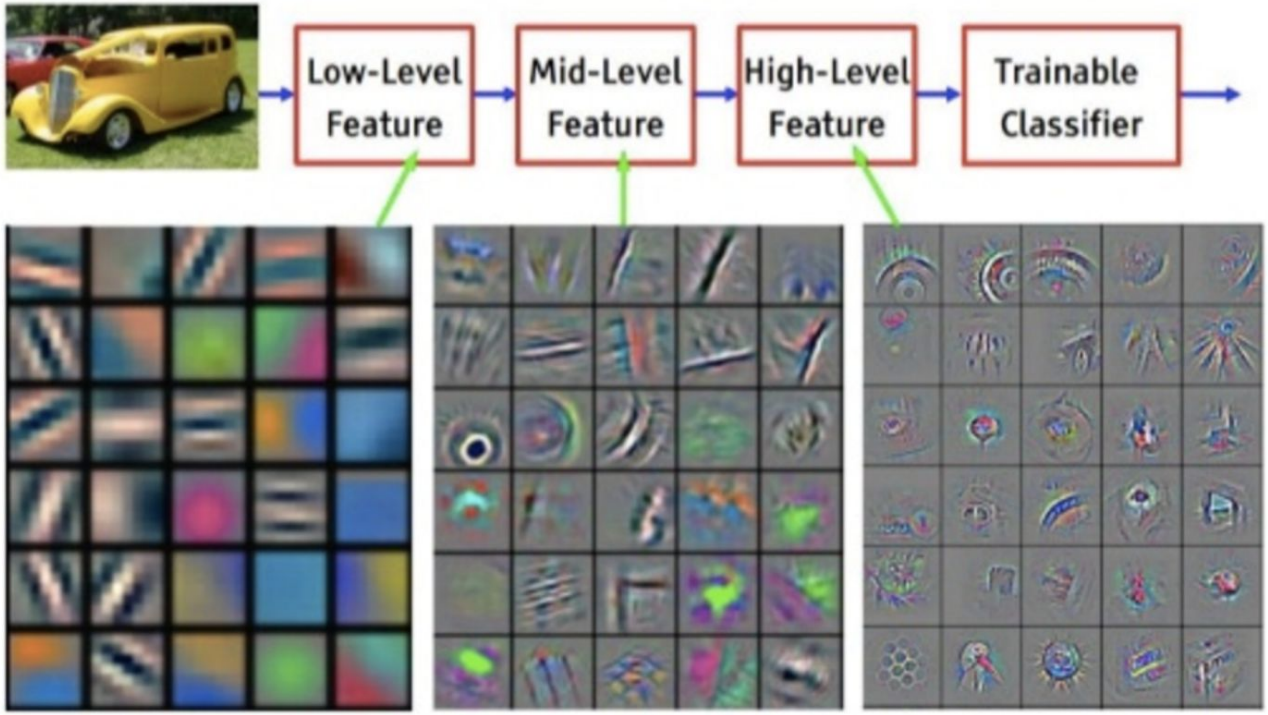
Typical ConvNet Structure



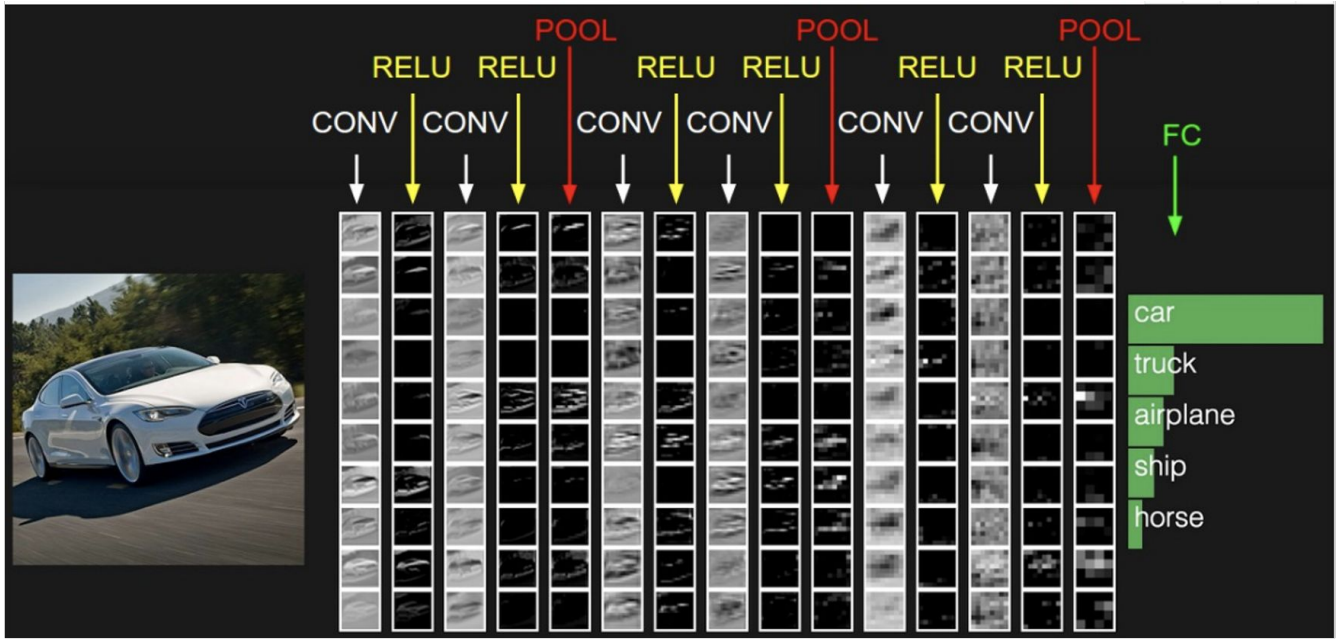
Receptive Fields and Features



Automatic Feature Extraction



Automatic Feature Extraction

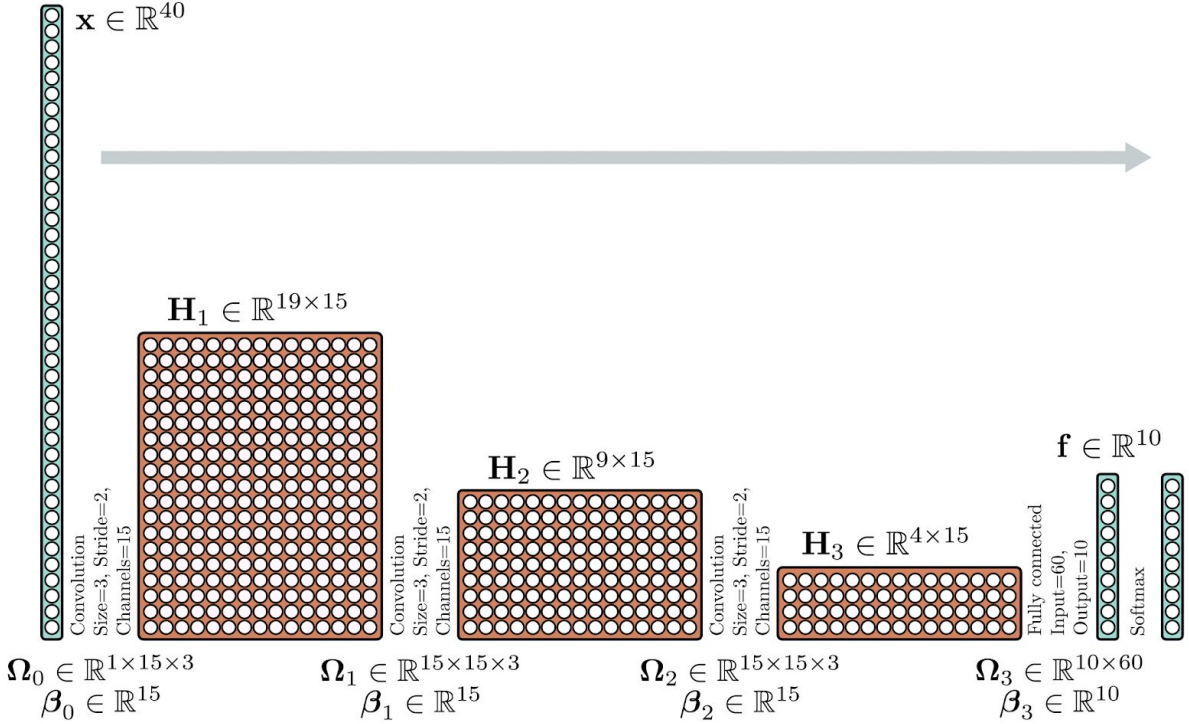


CONV: Convolutional kernel layer
RELU: Activation function
POOL: Dimension reduction layer
FC: Fully connection layer

CNN on MNIST-1D Data

- The input **x** is a 40D vector, and the output **f** is a 10D vector that is passed through a softmax layer to produce class probabilities
- We use a network with three hidden layers
- The fifteen channels of the first hidden layer H1 are each computed using a kernel size of three and a stride of two with “valid” padding, giving nineteen spatial positions.
- The second hidden layer H2 is also computed using a kernel size of three, a stride of two, and “valid” padding. The third hidden layer is computed similarly.

CNN on MNIST-1D Data



Colab time: CNN on MNIST 1D

https://colab.research.google.com/drive/1-MMnCa0dSFHvW_E-eWMFOuBqR94qyX03

Solution

https://colab.research.google.com/drive/1rUvQexD82X5_39SYI1NX3C0DlIJAO2pZ

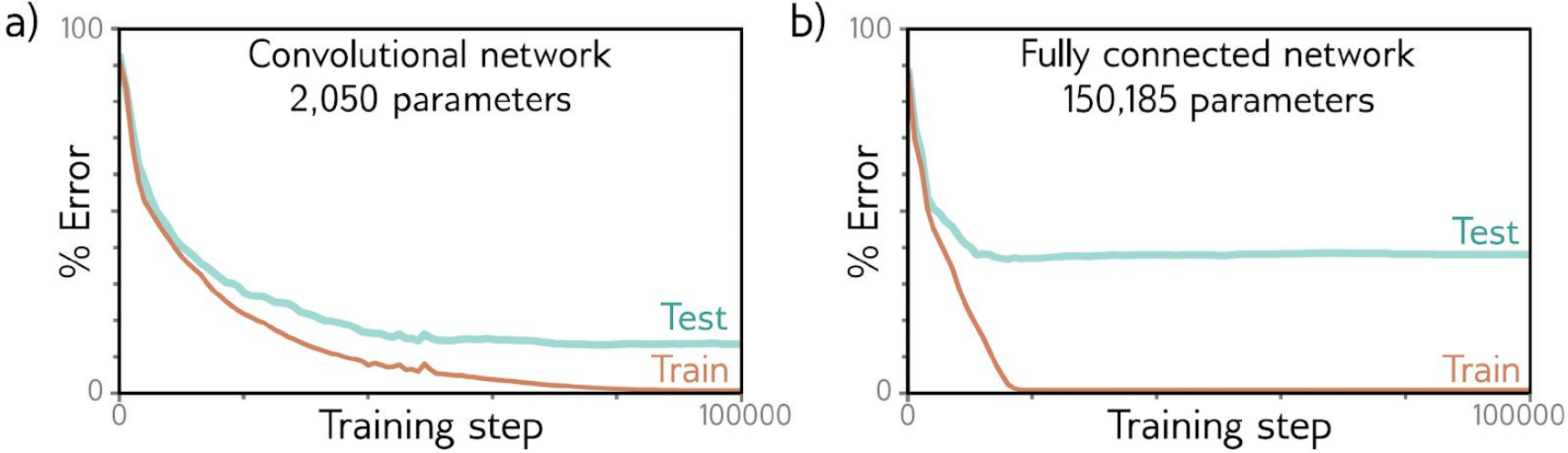


Figure 10.8 MNIST-1D results. a) The convolutional network from figure 10.7 eventually fits the training data perfectly and has ∼17% test error. b) A fully connected network with the same number of hidden layers and the number of hidden units in each learns the training data faster but fails to generalize 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.

How about 2D input?

- CNNs are more usually applied to 2D image data.
  - The convolutional kernel is now a 2D object. A 3×3 kernel $\Omega \in R^{3\times3}$ applied to a 2D input comprising of elements x_{ij} computes a single layer of hidden units h_{ij} 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.
- This is simply a weighted sum over a square 3×3 input region.

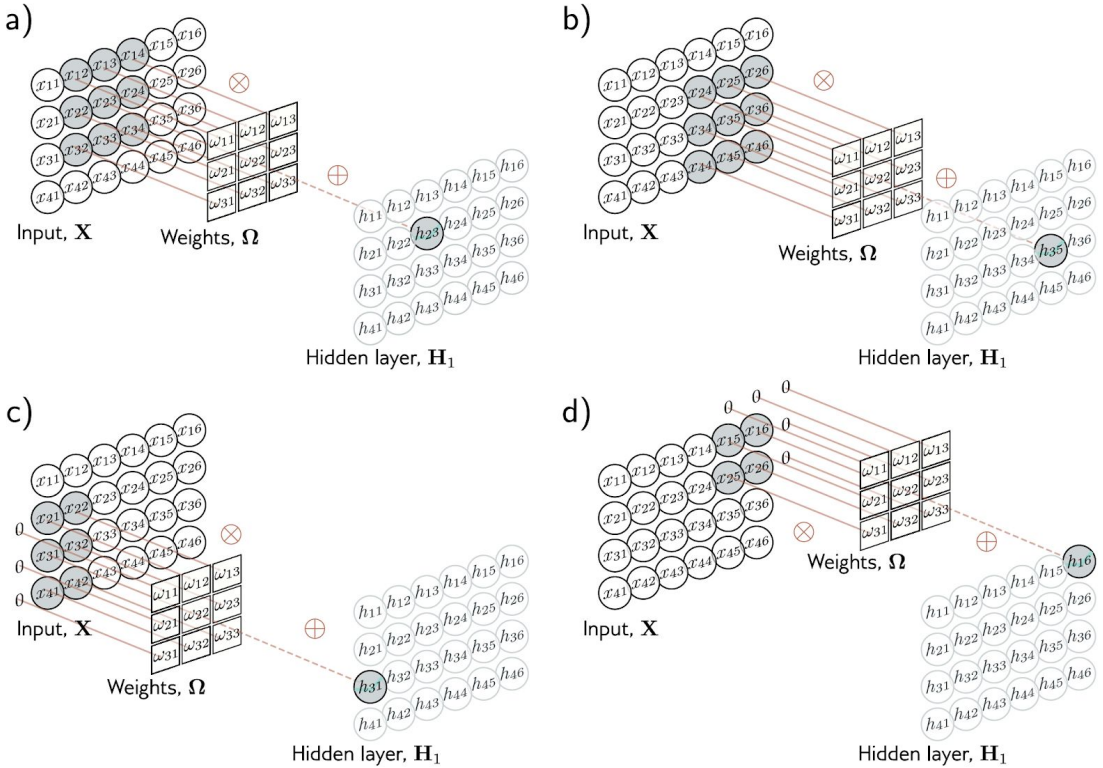


Figure 10.9 2D convolutional layer. Each output h_{ij} computes a weighted sum of the 3×3 nearest inputs, adds a bias, and passes the result through an activation function. a) Here, the output h_{23} (shaded output) is a weighted sum of the nine positions from x_{12} to x_{34} (shaded inputs). b) Different 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.

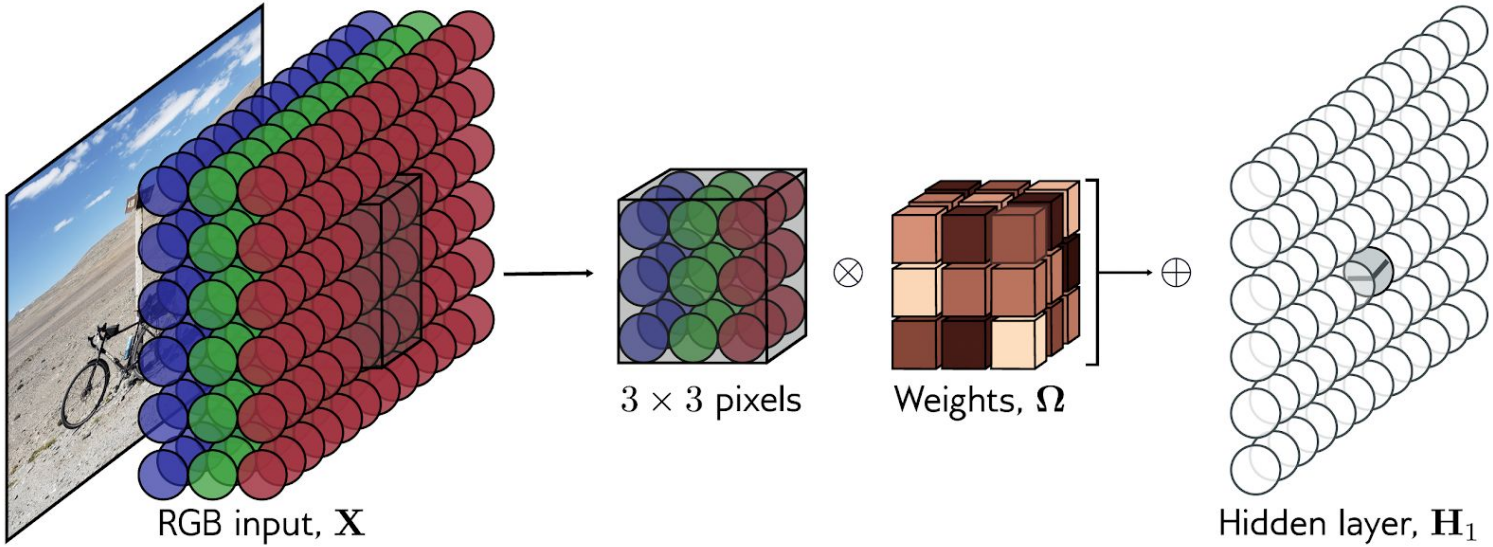
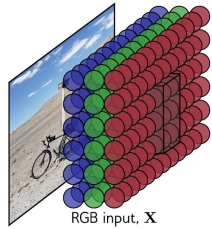


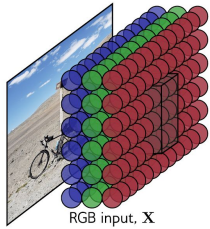
Figure 10.10 2D convolution applied to an image. The image is treated as a 2D input with three channels corresponding to the red, green, and blue components. With a 3×3 kernel, each pre-activation in the first hidden layer is computed by pointwise multiplying the 3×3×3 kernel weights with the 3×3 RGB 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 $\mathbf{H}_1$.

How Many Parameters?



- Let's assume the input is an RGB image, which is treated as a 2D signal with three channels (RGB)
- Here, a 3×3 kernel would have 3×3×3 weights and be applied to the three input channels at each of the 3×3 positions to create a 2D output that is the same height and width as the input image (assuming zero padding).
- To generate multiple output channels, we repeat this process with different kernel weights and append the results to form a 3D tensor.
- If the kernel is size K × K, and there are C_i input channels, each output channel is a weighted sum of $C_i \times K \times K$ quantities plus one bias.
- It follows that to compute Co output channels, we need $C_i \times C_o \times K \times K$ weights and C_o biases.

Downsampling



- **Simple**. When we use a stride of S, effectively sampling positions simultaneously with the convolution operation.
- **Max pooling** retains the maximum of the K×K input values.
  - This induces some invariance to translation; if the input is shifted by K-1 pixel, many of these maximum values remain the same.
- **Average** (**Avg**) / **Mean pooling** averages the inputs.
- In all the cases the number of channels stays the same but the dimensions are reduced

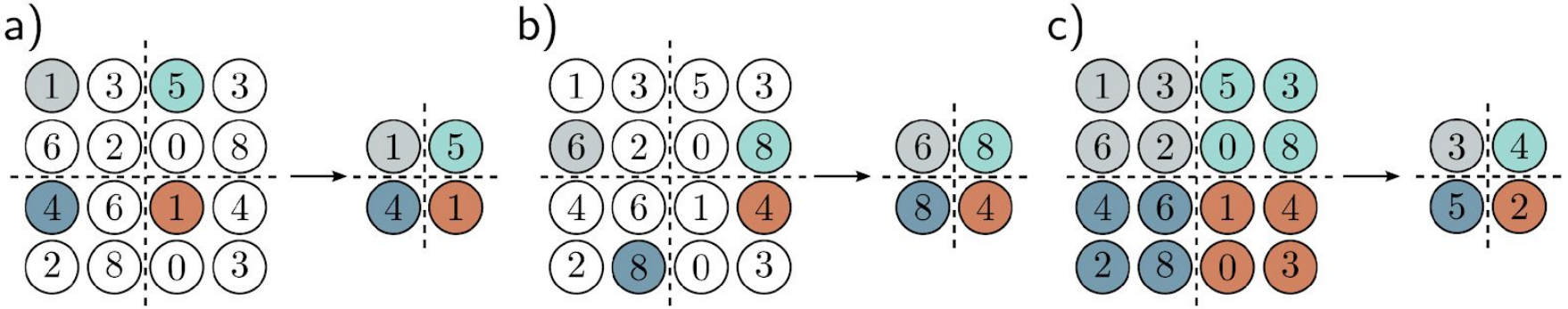
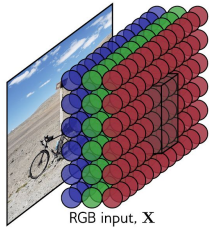


Figure 10.11 Methods for scaling down representation size (downsampling). a) Sub-sampling. The original 4×4 representation (left) is reduced to size 2×2 (right) by retaining every other input. Colors on the left indicate which inputs contribute to the outputs on the right. This is effectively what happens 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



- **Simple**. Duplicate all the channels at each spatial position M times
- **Max unpooling**. This is used where we have previously used a max pooling operation for downsampling
  - we distribute the values to the positions they originated from.
- **Bilinear interpolation** to fill in the missing values between the points where we have samples.

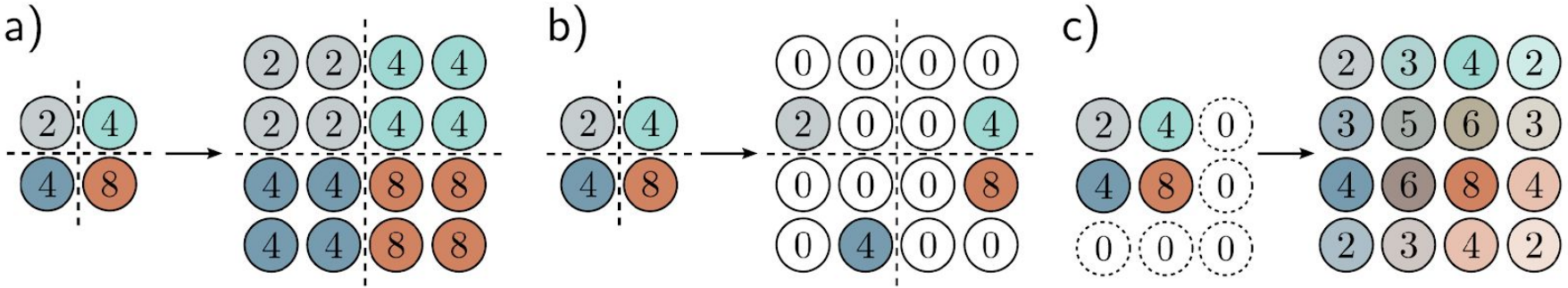
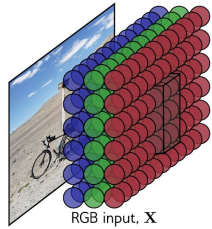


Figure 10.12 Methods for scaling up representation size (upsampling). a) The simplest way to double the size of a 2D layer is to duplicate each input four times. b) In networks where we have previously used a max pooling operation (figure 10.11b), we can redistribute the values to the same positions 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.

Transposed Convolution



- analogous to downsampling using a stride of S.
  - In that method, there were half as many outputs as inputs, and for kernel size three, each output was a weighted sum of the three closest inputs.
- In transposed convolution, this picture is reversed.
  - There are twice as many outputs.

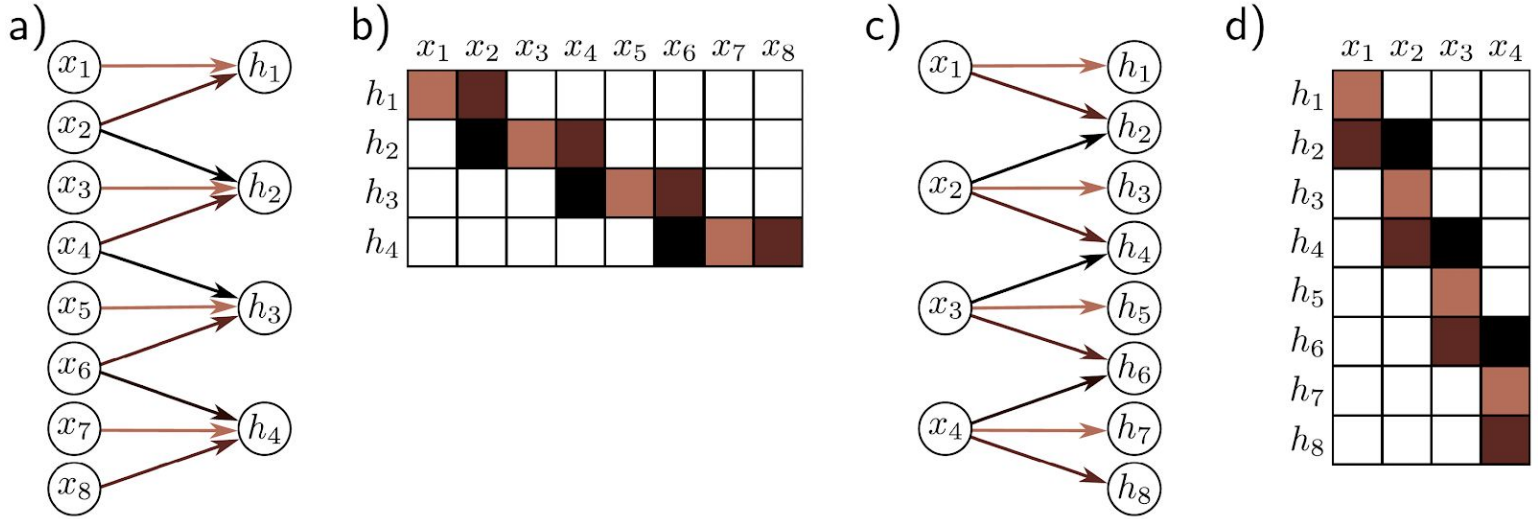


Figure 10.13 Transposed convolution in 1D. a) Downsampling with kernel size three, stride two, and zero padding. Each output is a weighted sum of three inputs (arrows indicate weights). b) This can be expressed by a weight matrix (same color indicates shared weight). c) In transposed convolution, each input contributes three values to the output layer, which has twice as many outputs as inputs. d) The associated weight matrix is the transpose of that in panel (b).

1x1 Convolution

- To change the number of channels we can apply a 1×1 kernel.
- Each output channel is computed by taking a weighted sum of all of the channels at the same position, adding a bias, and passing through an activation function.
- Multiple output channels are created by repeating this operation with different weights and biases.

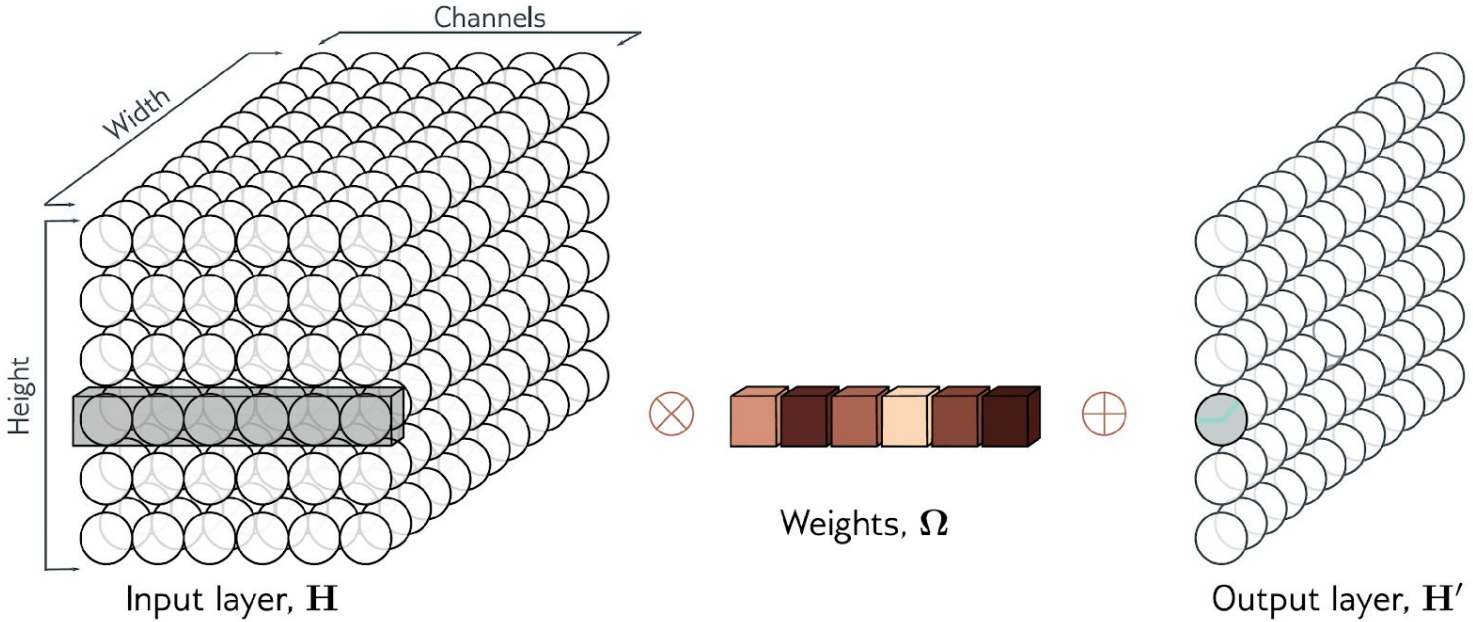


Figure 10.14 1×1 convolution. To change the number of channels without spatial pooling, we apply a 1×1 kernel. Each output channel is computed by taking a weighted sum of all of the channels at the same position, adding a bias, and passing through an activation function. Multiple output channels are created by repeating this operation with different weights and biases.

Applications

Image Classification

- ImageNet
  - Contains 1,281,167 training images, 50,000 validation images, and 100,000 test images, and every image is labeled as belonging to one of 1,000 possible categories.
- input **x** to the network is a 224×224 RGB image, and the output is a probability distribution over the 1,000 classes.
- In 2016, *the best deep learning models eclipsed human performance*.

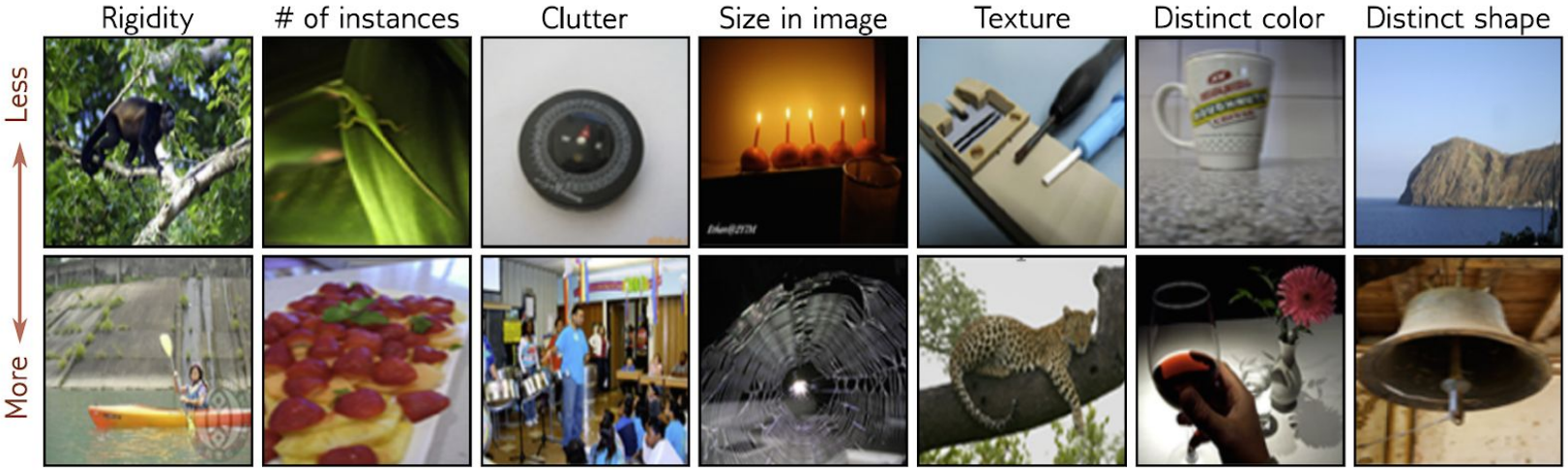
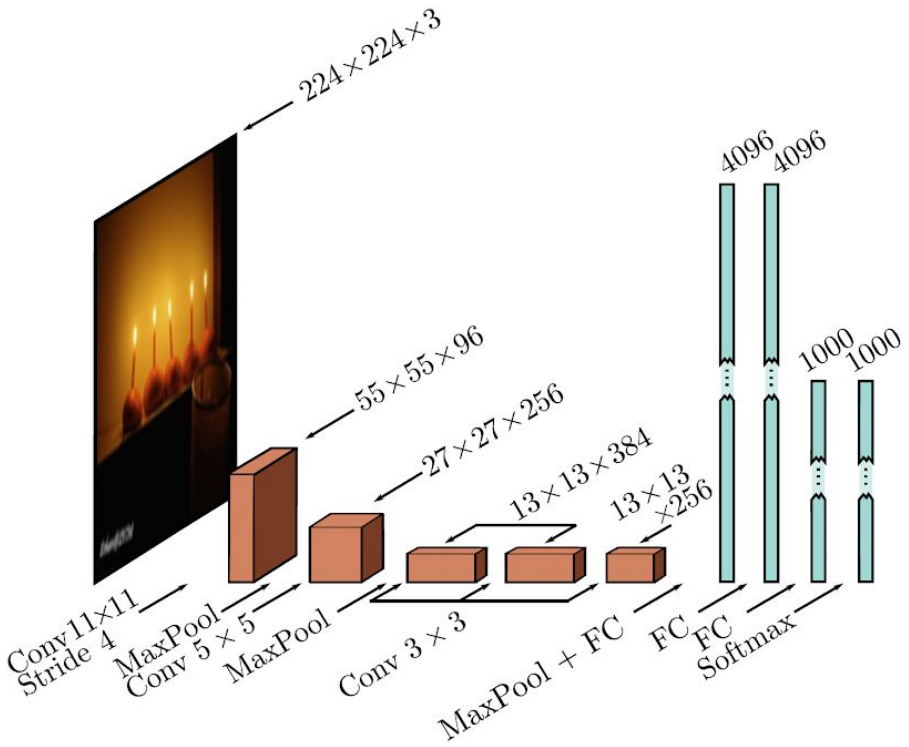


Figure 10.15 Example ImageNet classification images. The model aims to assign an input image to one of 1000 classes. This task is challenging because the images vary widely along different attributes (columns). These include rigidity (monkey < canoe), number of instances in image (lizard < strawberry), clutter (compass < steel drum), size (candle < spiderweb), texture (screwdriver < leopard), distinctiveness of color (mug < red wine), and distinctiveness of shape (headland < bell). Adapted from Russakovsky et al. (2015).

Figure 10.16 AlexNet (Krizhevsky et al., 2012). The network maps a 224×224 color image to a 1000-dimensional vector representing class probabilities. The network first convolves with 11×11 kernels and stride 4 to create 96 channels. It decreases the resolution again using a max pool operation and applies a 5×5 convolutional layer. Another max pooling layer follows, and three 3×3 convolutional layers are applied. After a final max pooling operation, the result is vectorized and passed through three fully connected (FC) layers and finally the softmax layer.



AlexNet

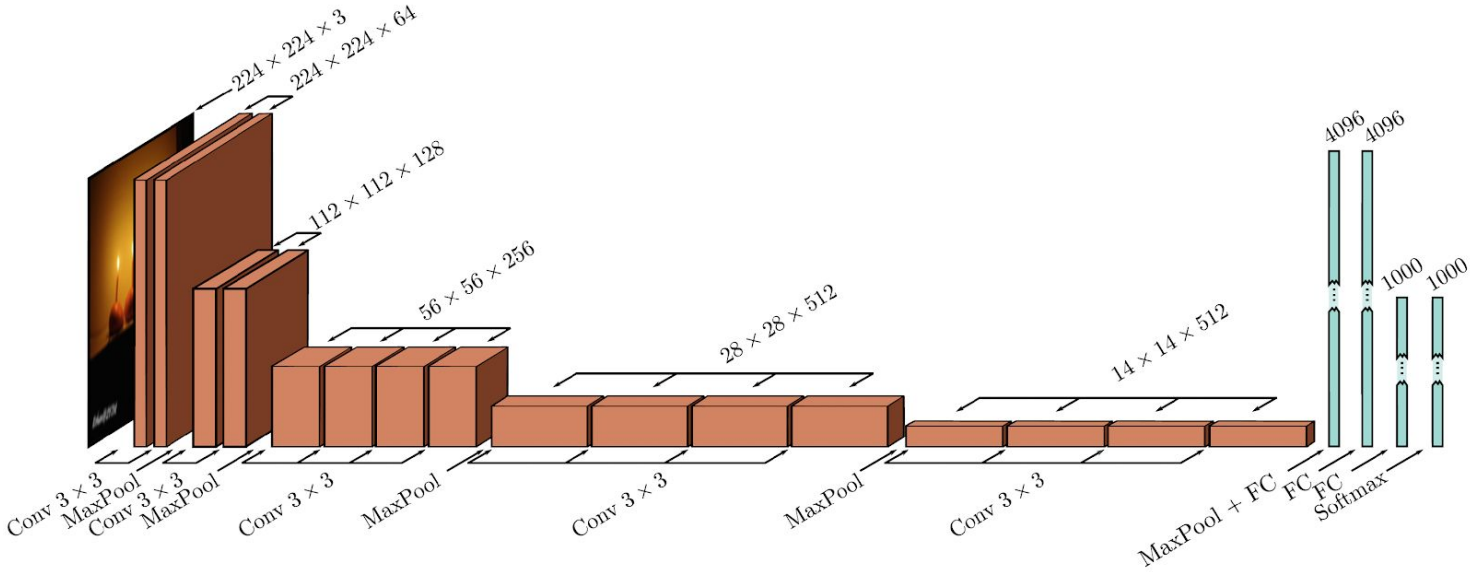


Figure 10.17 VGG network (Simonyan & Zisserman, 2014) depicted at the same scale as AlexNet (see figure 10.16). This network consists of a series of convolutional layers and max pooling operations, in which the spatial scale of the representation gradually decreases, but the number of channels gradually increases. The hidden layer after the last convolutional operation is resized to a 1D vector and three fully connected layers follow. The network outputs 1000 activations corresponding to the class labels that are passed through a softmax function to create class probabilities.

VGG

Object Detection

- The goal is to identify and localize multiple objects within the image.
- An early method based on convolutional networks was You Only Look Once, or YOLO for short.
  - The input to the YOLO network is a 448×448 RGB image.
  - This is passed through 24 convolutional layers that gradually decrease the representation size using max pooling operations while concurrently increasing the number of channels
    - Similarly to the VGG network.
  - The final convolutional layer is of size 7×7 and has 1024 channels.
  - This is reshaped to a vector, and a fully connected layer maps it to 4096 values.
  - One further fully connected layer maps this representation to the output.

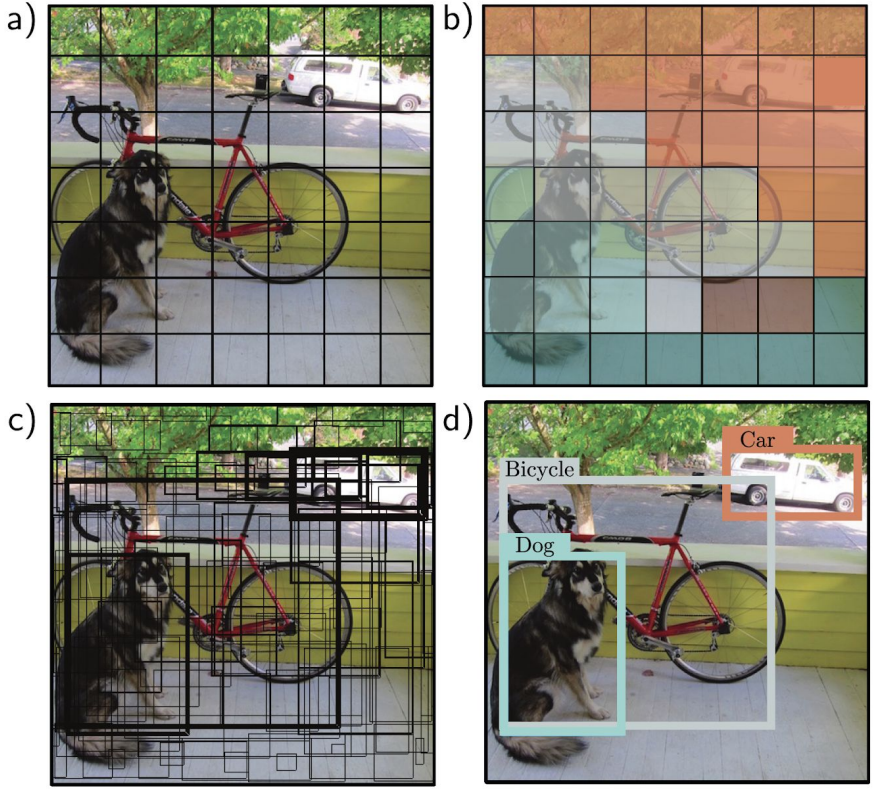
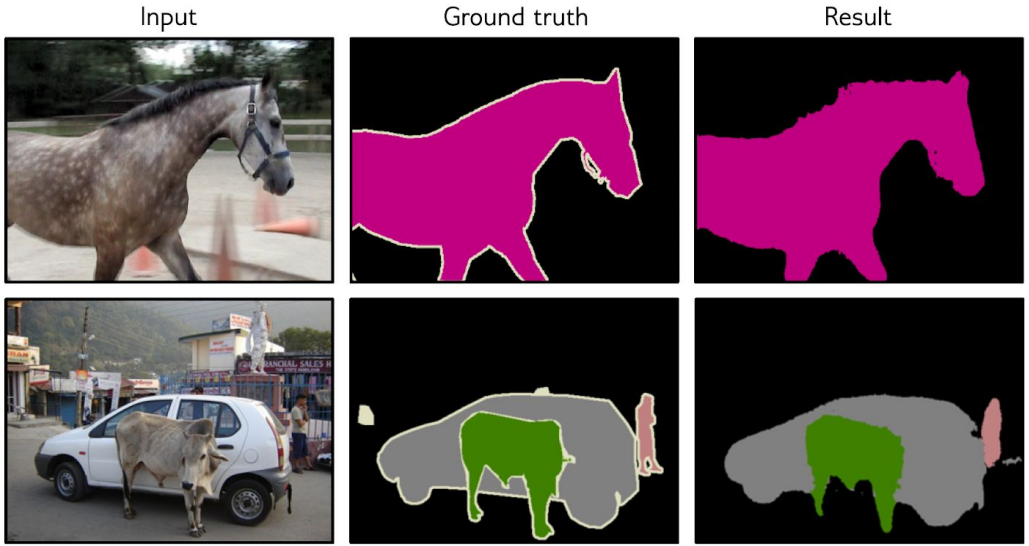


Figure 10.18 YOLO object detection. a) The input image is reshaped to 448×448 and divided into a regular 7×7 grid. b) The system predicts the most likely class at each grid cell. c) It also predicts two bounding boxes per cell, and a confidence value (represented by thickness of line). d) During inference, 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).

Semantic Segmentation

- The goal of semantic segmentation is to assign a label to each pixel according to the object that it belongs to or no label if that pixel does not correspond to anything in the training database.



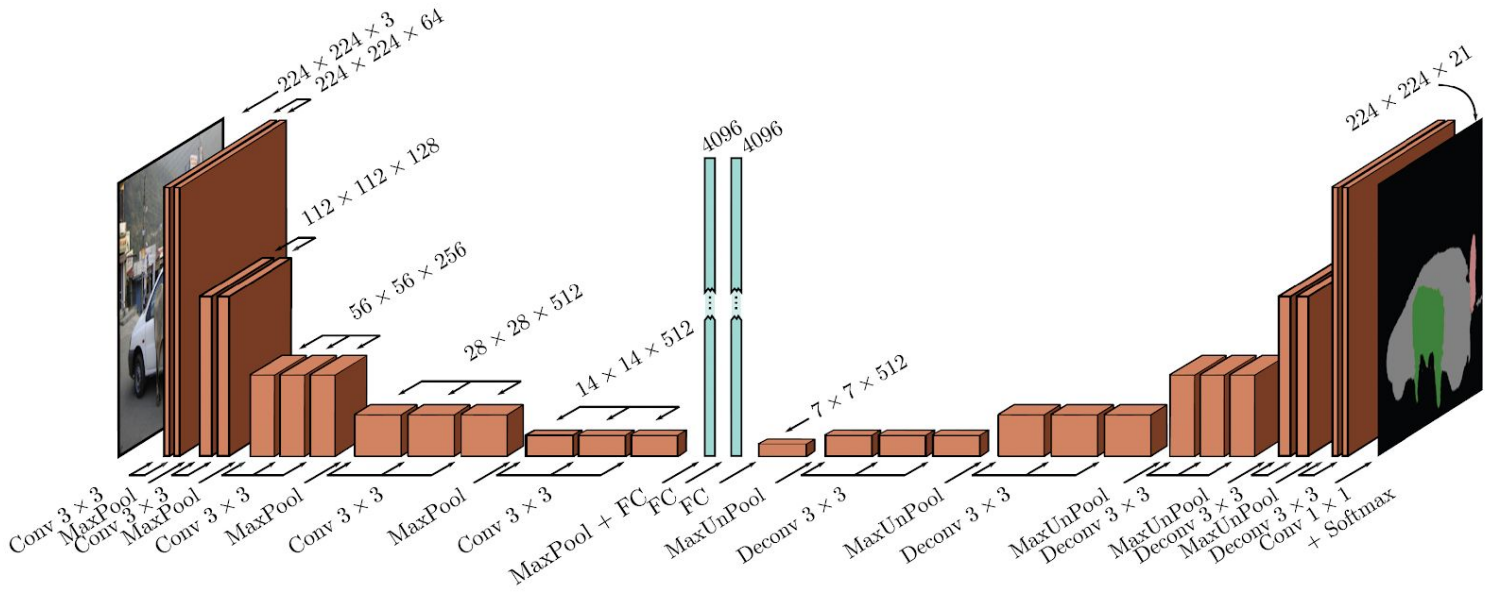


Figure 10.19 Semantic segmentation network of Noh et al. (2015). The input is a 224×224 image, which is passed through a version of the VGG network and eventually transformed into a representation of size 4096 using a fully connected layer. This contains information about the entire image. This is then reformed into a representation of size 7×7 using another fully connected layer, and the image is upsampled and deconvolved (transposed convolutions without upsampling) in a mirror image of the VGG network. The output is a 224×224×21 representation that gives the output probabilities for the 21 classes at each position.

In Summary

- In convolutional layers, each hidden unit is computed by taking a ***weighted sum of the nearby inputs***, adding a bias, and applying an activation function.
- The weights and the bias are the same at every spatial position, so there are far ***fewer parameters than in a fully connected network***, and the parameters don't increase with the input image size.
- To ensure that information is not lost, this operation is repeated with different weights and biases to ***create multiple channels*** at each spatial position.

In Summary

- Typical convolutional networks consist of convolutional layers interspersed with layers that downsample by a factor of two.
- As the network progresses, the spatial dimensions usually decrease by factors of two, and the number of channels increases by factors of two.
- At the end of the network, there are typically one or more fully connected layers that integrate information from across the entire input and create the desired output.
- **If the output is an image, a mirrored “decoder” upsamples back to the original size**.

In Summary

- The translational equivariance of convolutional layers imposes a useful inductive bias that increases performance for image-based tasks relative to fully connected networks.
- We described image classification, object detection, and semantic segmentation networks.
- Image classification performance was shown to improve as the network became deeper.
- However, subsequent experiments showed that ***increasing the network depth indefinitely doesn't continue to help; after a certain depth, the system becomes difficult to train***.

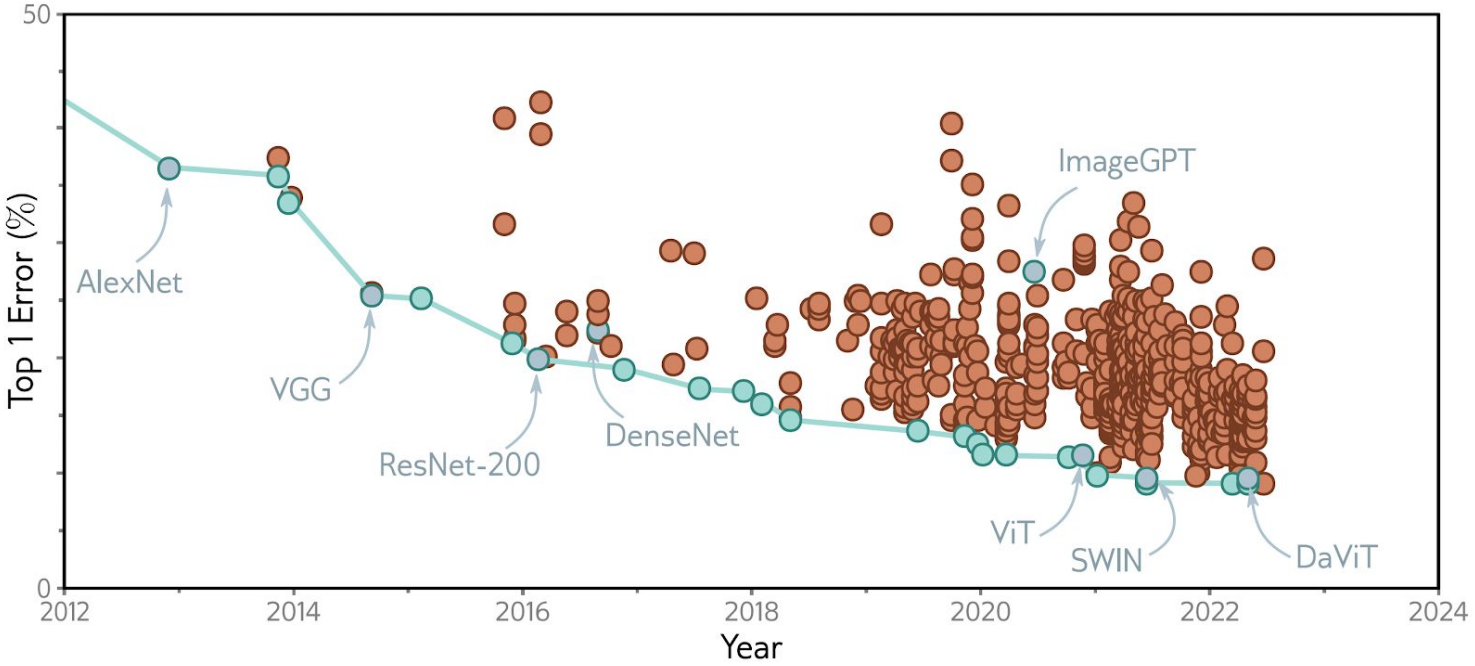


Figure 10.21 ImageNet performance. Each circle represents a different published model. Blue circles represent models that were state-of-the-art. Models discussed in this book are also highlighted. The AlexNet and VGG networks were remarkable for their time but are now far from state of the art. ResNet-200 and DenseNet are discussed in chapter 11. ImageGPT, ViT, SWIN, and DaViT are discussed in chapter 12. Adapted from https://paperswithcode.com/sota/image-classification-on-imagenet.

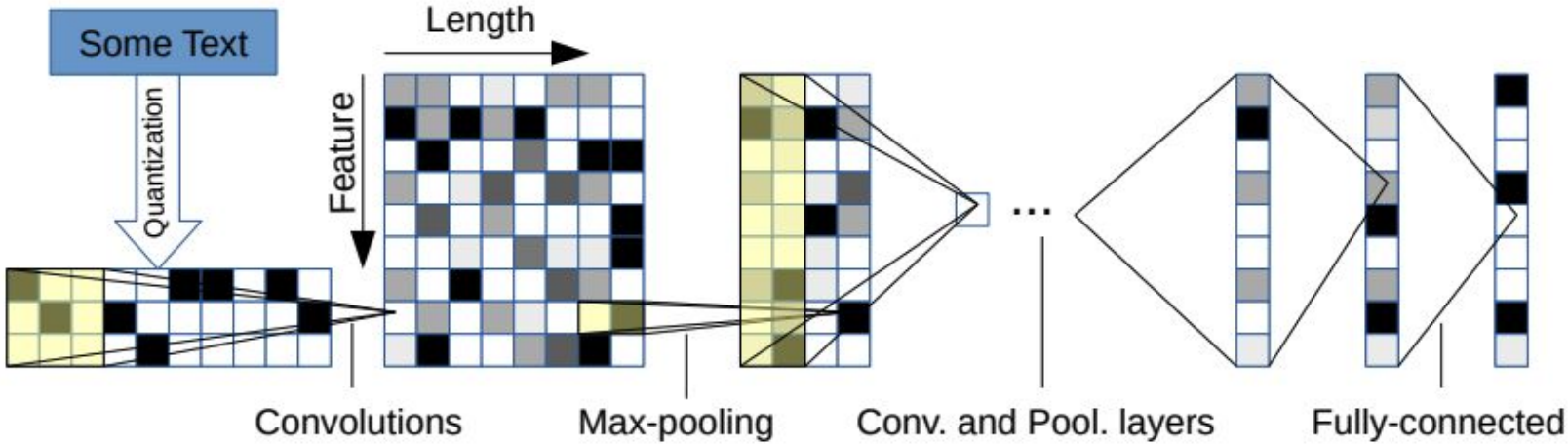
Colab time: CNN on MNIST

https://colab.research.google.com/drive/1bVSmZpGbNz_n0U2qDLZO8hSqSoxVR1Uk

Solution: CNN on MNIST

https://colab.research.google.com/drive/1WsTnG8DVAD4wVY9hj9IlpmvzLEDIdMR2

CNN for Text



ResNets

(Chapter 11)

SAPIENZA
UNIVERSITÀ DI ROMA

Sequential Processing

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

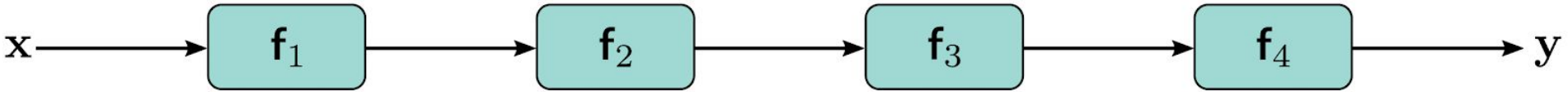
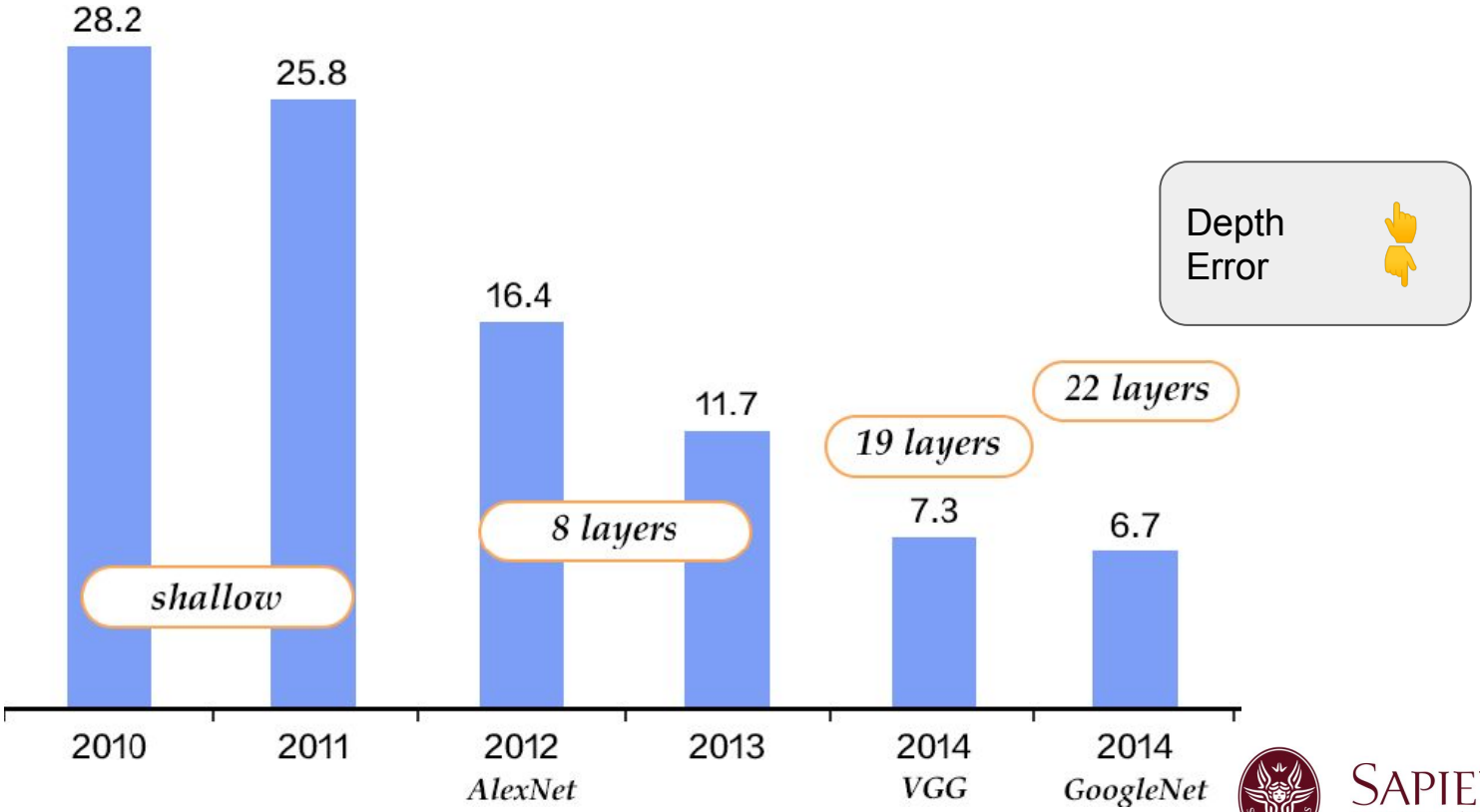


Figure 11.1 Sequential processing. Standard neural networks pass the output of each layer directly into the next layer.

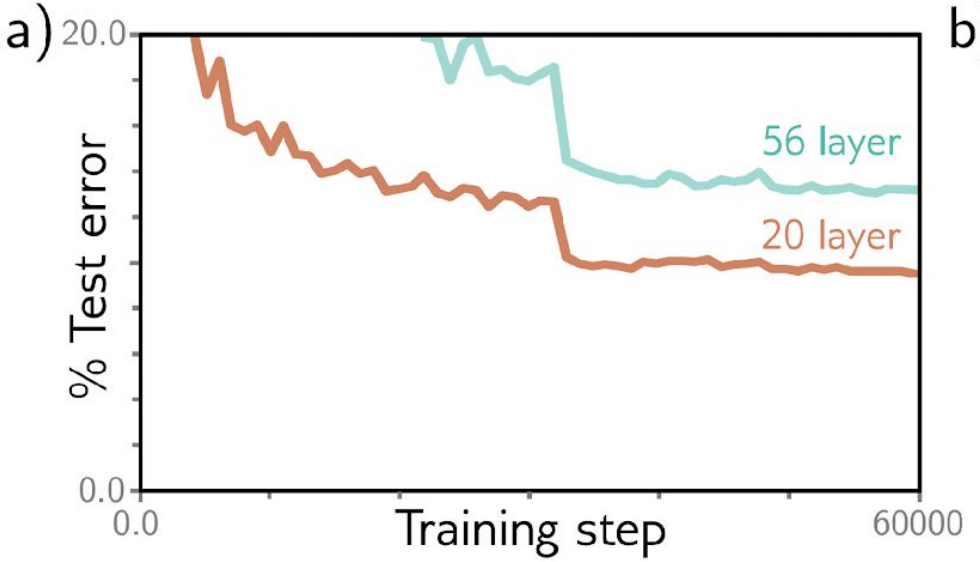
- Since the processing is sequential, we can equivalently think of this network as a series of nested functions:

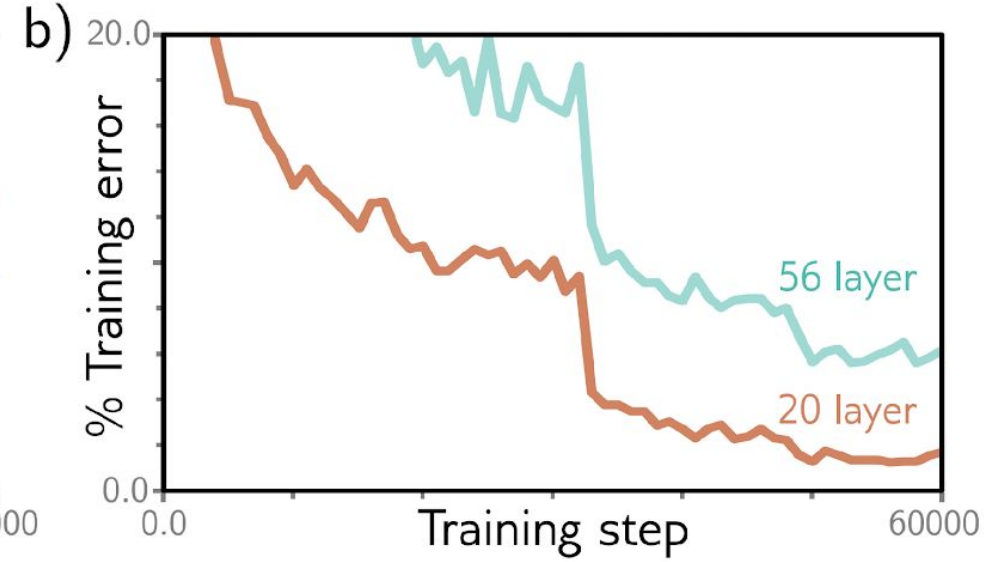
$$\mathbf{y} = \mathbf{f}_4\Big[\mathbf{f}_3\Big[\mathbf{f}_2\big[\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\big], \boldsymbol{\phi}_3\Big], \boldsymbol{\phi}_4\Big]$$

The Role of Depth



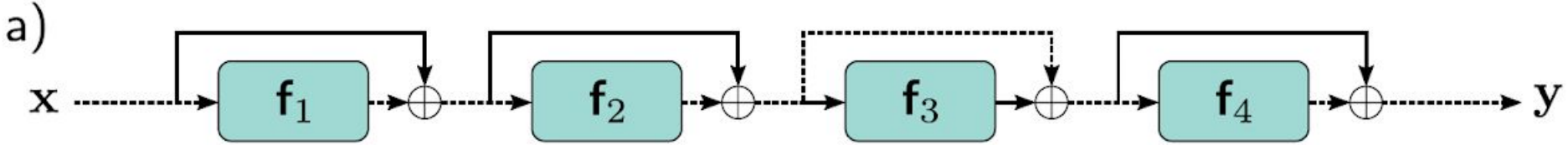
But…





Residual Connections

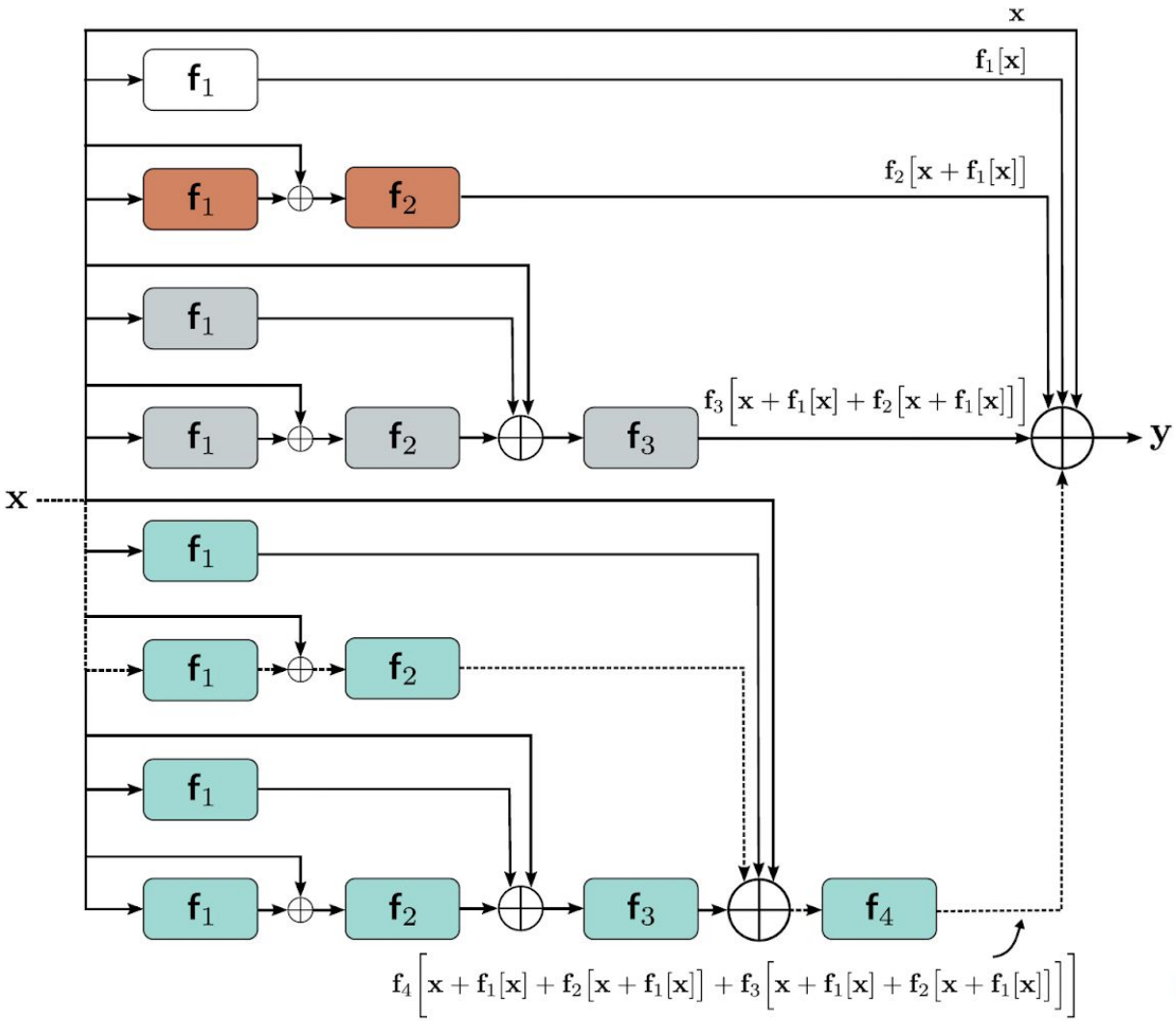
$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3,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



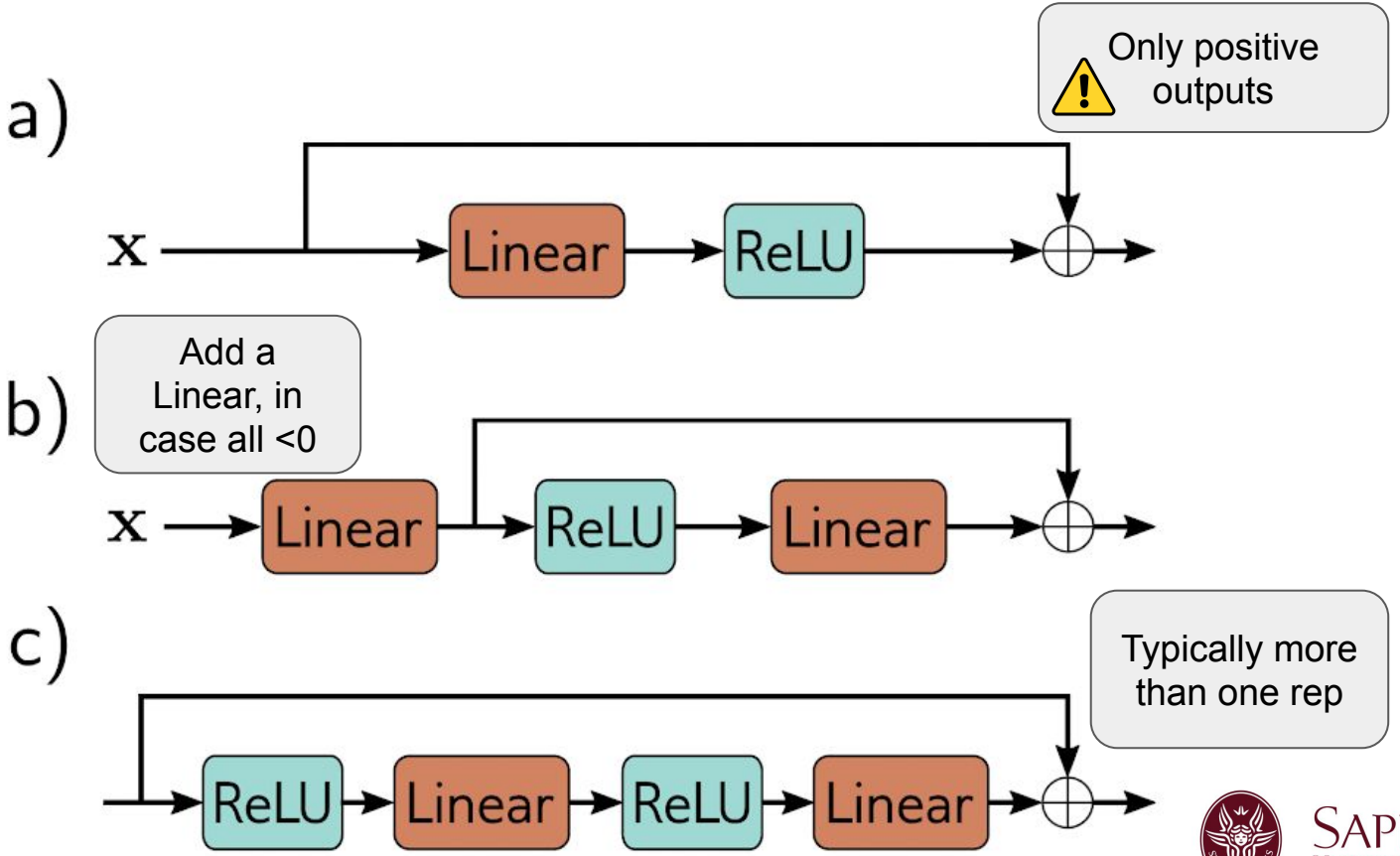
Expand

$$
\begin{aligned}
\mathbf{y} = \mathbf{x} &+ \mathbf{f}_1[\mathbf{x}] \\
&+ \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right] \\
&+ \mathbf{f}_3\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right]\right] \\
&+ \mathbf{f}_4\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right] + \mathbf{f}_3\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\left[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\right]\right]\right]
\end{aligned}
$$

Expand



Typical Residual Block



Interpretations

1. Ensemble of models
2. K paths (in the example 16) from Input to output
3. Let's have a look at one of the partial derivatives:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \mathbf{I} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \left(\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} \right) + \left(\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2} \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} \right)$$

   - The identity term on the right hand side shows that changes in the parameters ϕ_1 in the first layer $\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1]$ contribute directly to changes in the network output y.
   - They also contribute indirectly through the other chains of derivatives of varying lengths. In general, gradients through shorter paths will be better behaved.

Colab time: ResNet

https://colab.research.google.com/drive/1hCNtryRPctrXORtBCDnfCMNAnk013EIn

Solution: ResNet

https://colab.research.google.com/drive/1vAwRqbkGCIe24bGYwy8SJ71_jrczVCZb

Exploding Gradients

1. Adding residual connections does not help too much to increase depth
2. It can be shown that gradients exponentially increase even with He init.
3. Solution:
    - **Batch Normalization** (Batch Norm)

Motivations

- Batch normalization is one of the most exciting innovations in optimizing deep neural networks
  - It is, basically, always used
- It is a method of adaptive reparametrization, motivated by the difficulty of training very deep models
- Very deep models involve the composition of several functions, or layers.
  - The gradient tells how to update each parameter, under the (wrong) assumption that the other layers do not change.

Reparametrization through Batch Normalization

- The reparametrization significantly reduces the problem of coordinating updates across many layers
- Batch normalization 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 the activations for each example appearing in a row of the matrix.
- To normalize H, we replace it with

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

where **μ** is a vector containing the mean of each unit and **σ** is a vector containing the standard deviation of each unit.

Reparametrization through Batch Normalization

- Within each row, the arithmetic is element-wise, so $H_{i,j}$ is normalized by subtracting μ_j and dividing by σ_j
- At training time

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \boldsymbol{H}_{i,:}$$

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2},$$

- Where δ is a small positive value such as 10^{-8}, imposed to avoid encountering the undefined gradient of $\sqrt{z}$ at $z = 0$.
- Crucially, we back-propagate through these operations for computing the mean and the standard deviation, and for applying them to normalize **H**.

Reparametrization through Batch Normalization

- At inference time, **μ** and **σ** may be replaced by running averages that were collected during training time.
  - This allows the model to be evaluated on a single example, without needing to use definitions of **μ** and **σ** that depend on an entire minibatch.

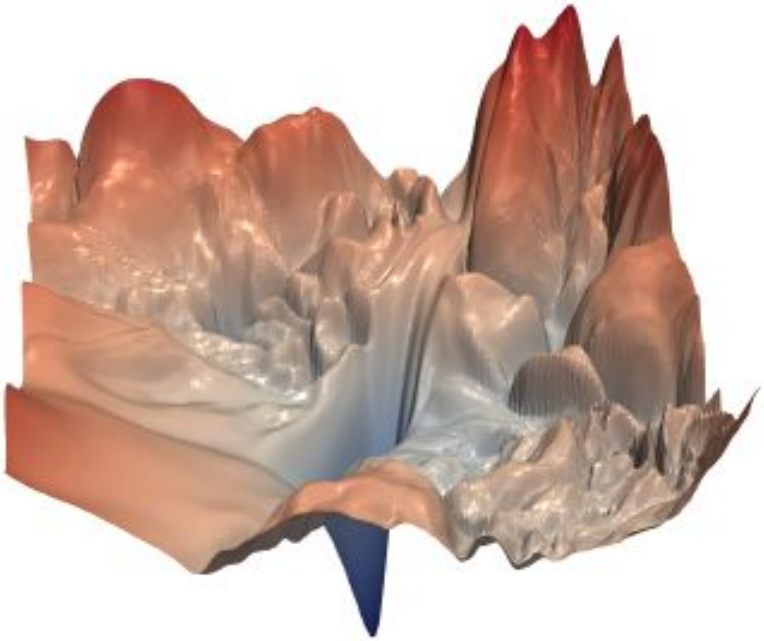
Colab time: BatchNorm

https://colab.research.google.com/drive/1NwGa4NwAuUn12B6xcZEg6dNtm1rjxllG

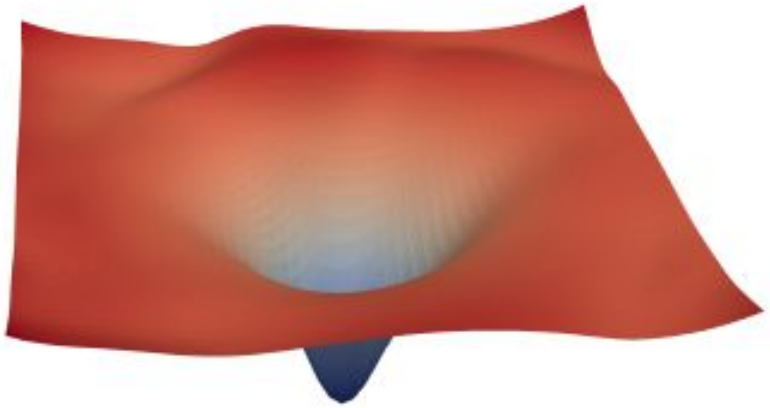
Solution: BatchNorm

https://colab.research.google.com/drive/1P_6JUS6dGMjMtYuHXHlVhr7g6Urumc5i

Effect of Residual Connections and BatchNorm



(a) without skip connections



(b) with skip connections

Some Residual Architectures

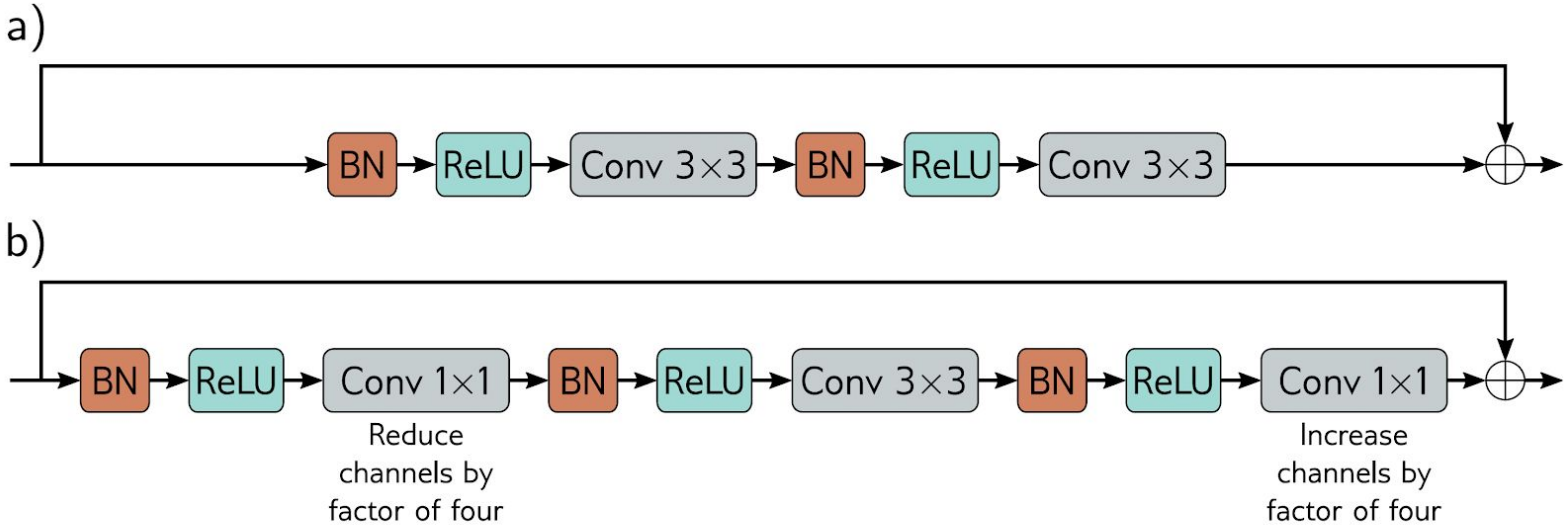


Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses 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.

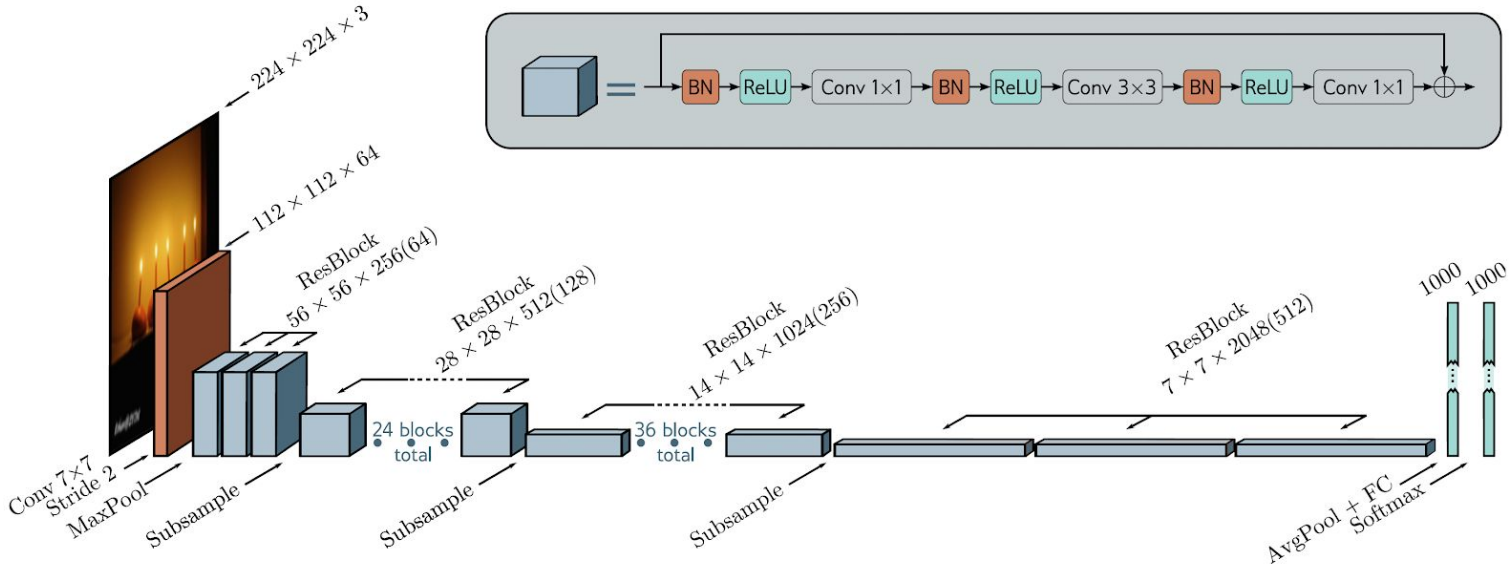


Figure 11.8 ResNet-200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling and accompanying increases in the number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

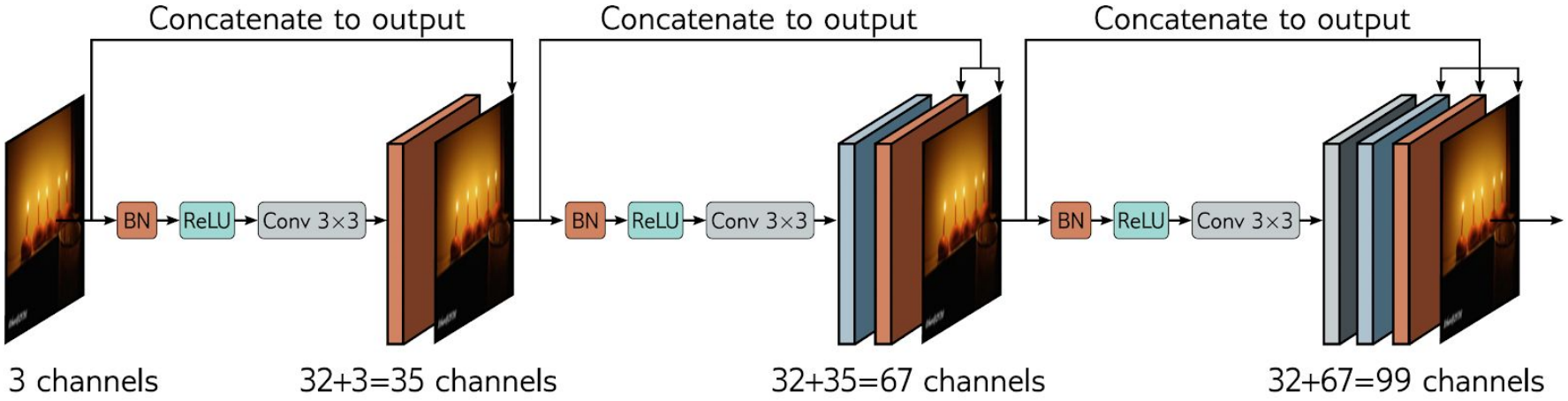


Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation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

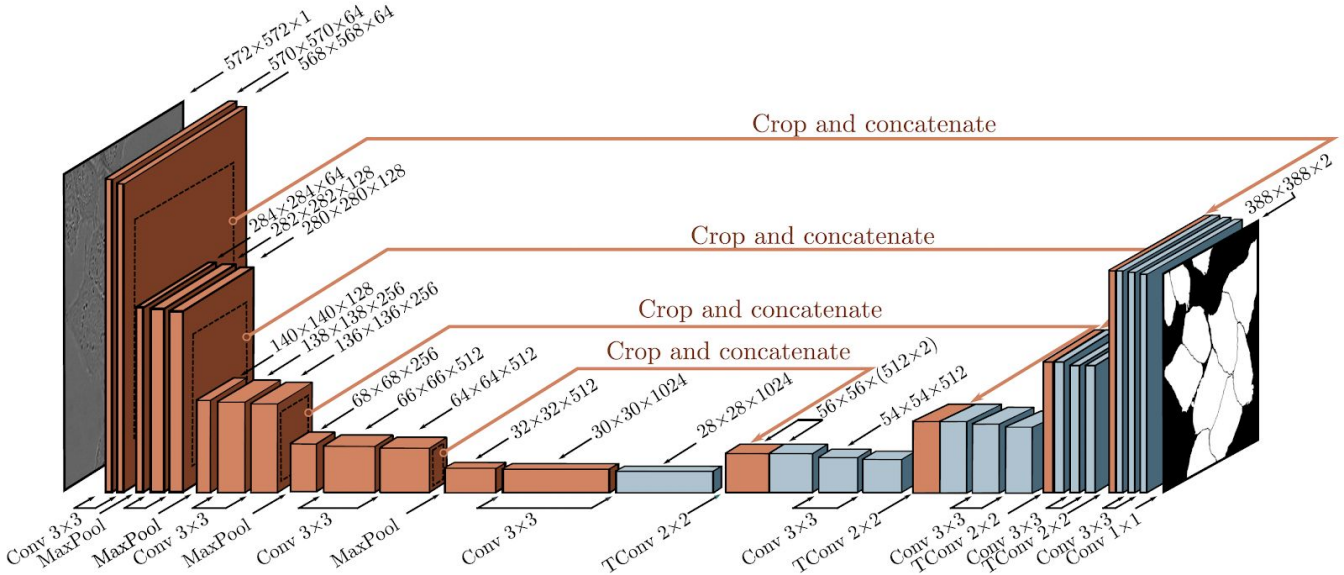


Figure 11.10 U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the 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).

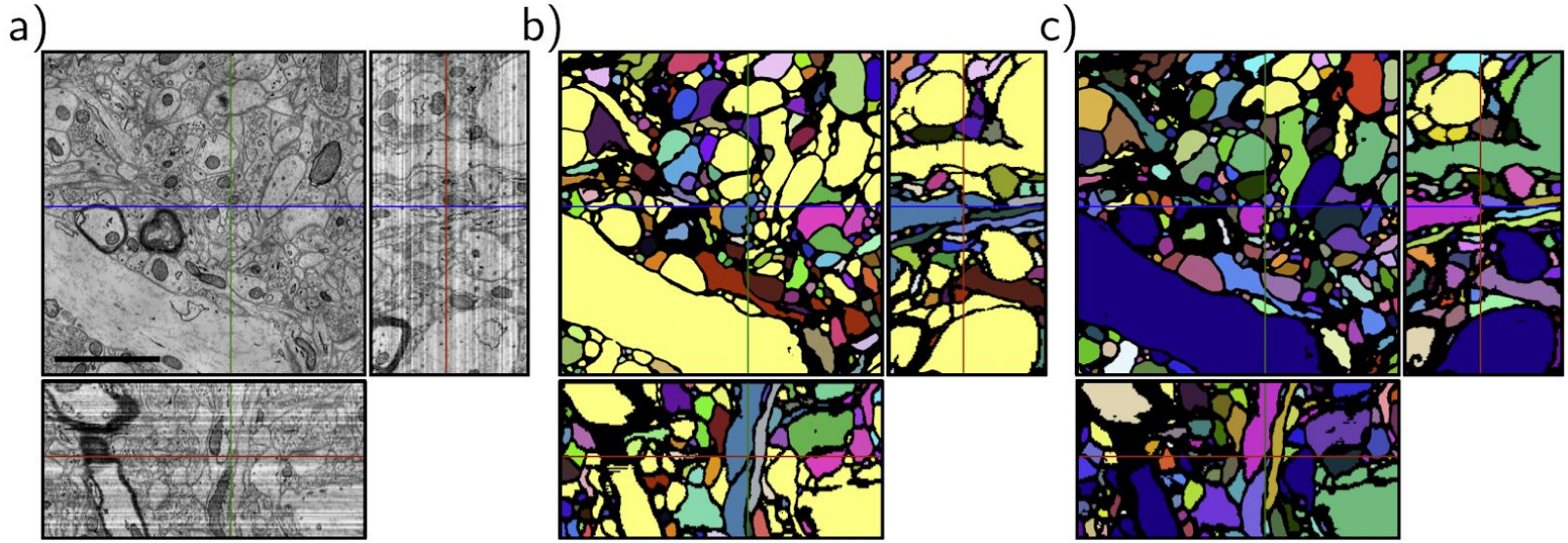


Figure 11.11 Segmentation using U-Net in 3D. a) Three slices through a 3D volume of mouse cortex taken by scanning electron microscope. b) A single U-Net is used to classify voxels as being inside or outside neurites. Connected regions are identified with different colors. c) For a better result, an ensemble of five U-Nets is trained, and a voxel is only classified as belonging to the cell if all five networks agree. Adapted from Falk et al. (2019).

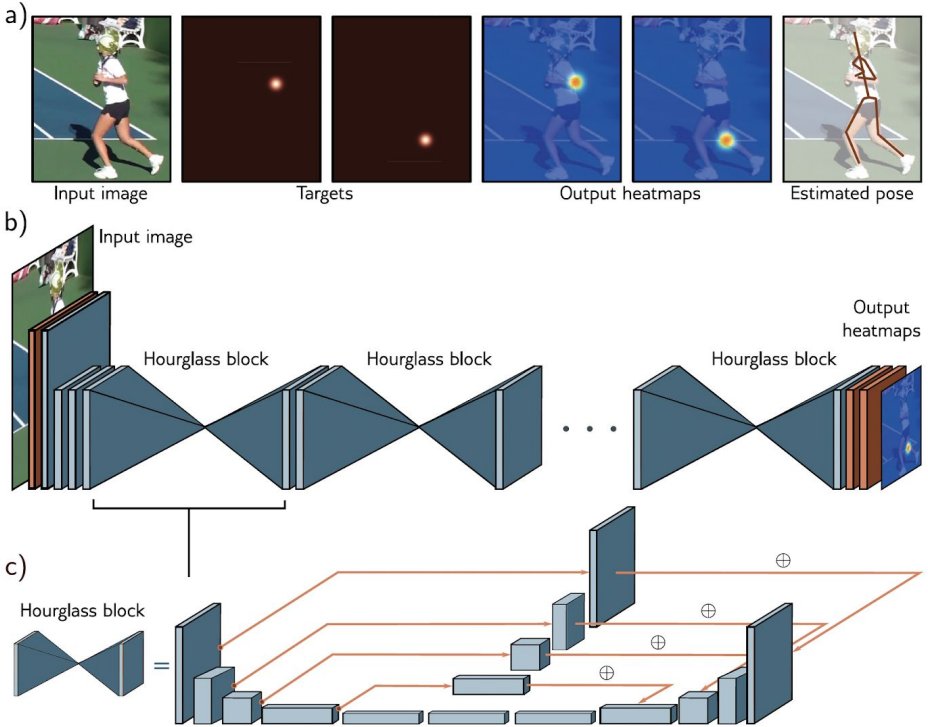


Figure 11.12 Stacked hourglass networks for pose estimation. a) The network input is an image containing a person, and the output is a set of heatmaps, with one heatmap for each joint. This is formulated as a regression problem where the targets are heatmap images with small, highlighted regions at the ground-truth joint positions. The peak of the estimated heatmap is used to establish each final joint position. b) The architecture consists of initial convolutional and residual layers followed by a series of hourglass blocks. c) Each hourglass block consists of an encoder-decoder network similar to the U-Net except that the convolutions use zero padding, some further processing is done in the residual links, and these links add this processed representation rather than concatenate it. Each blue cuboid is itself a bottleneck residual block (figure 11.7b). Adapted from Newell et al. (2016).

U-Net Networks

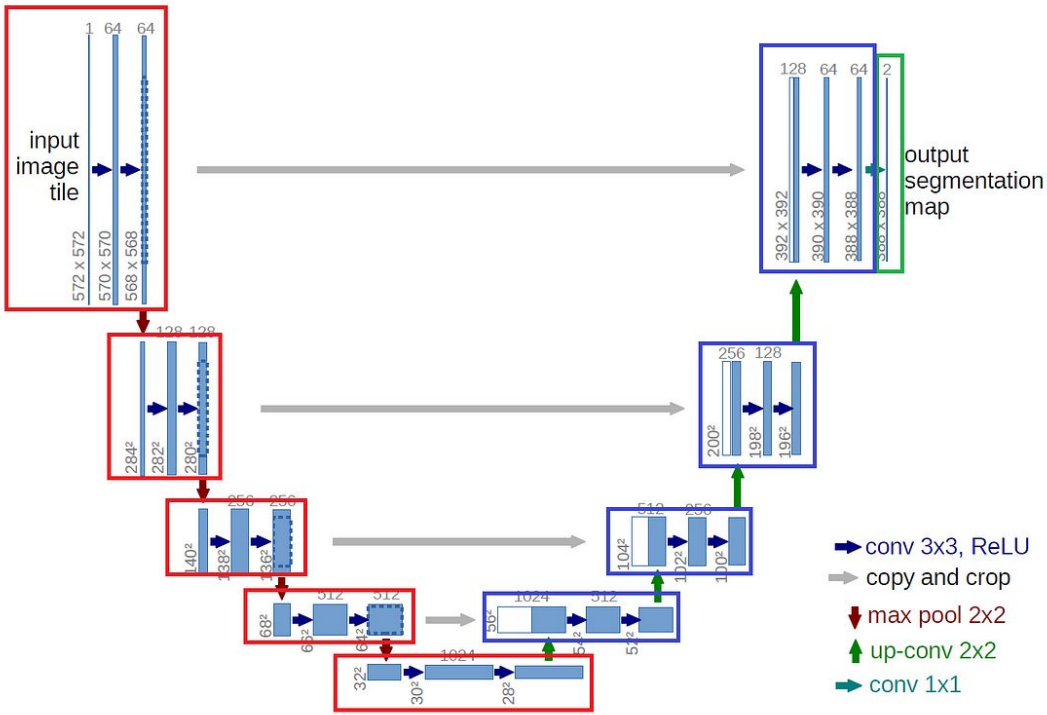
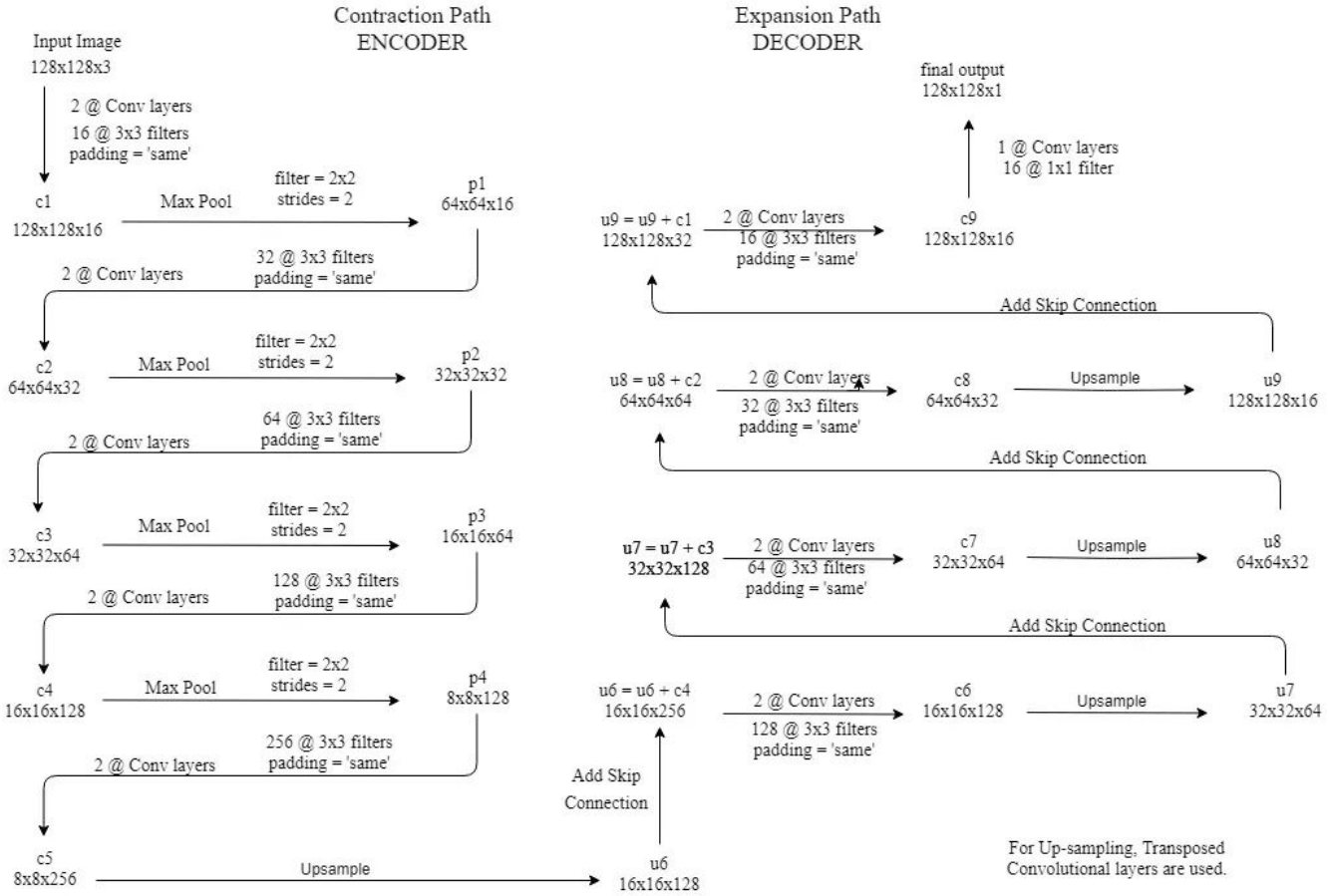


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.



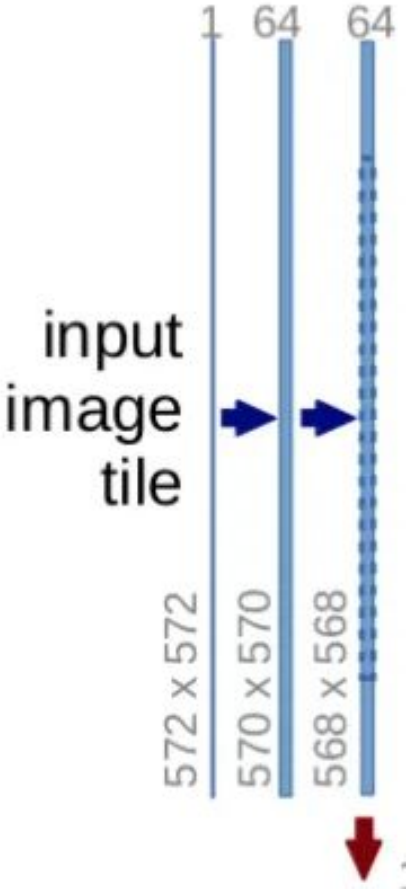
For Up-sampling, Transposed Convolutional layers are used.

The parameters for each Transpose Convolution are such that, the height and width of the image are doubled while the depth (no. of channels) is halved

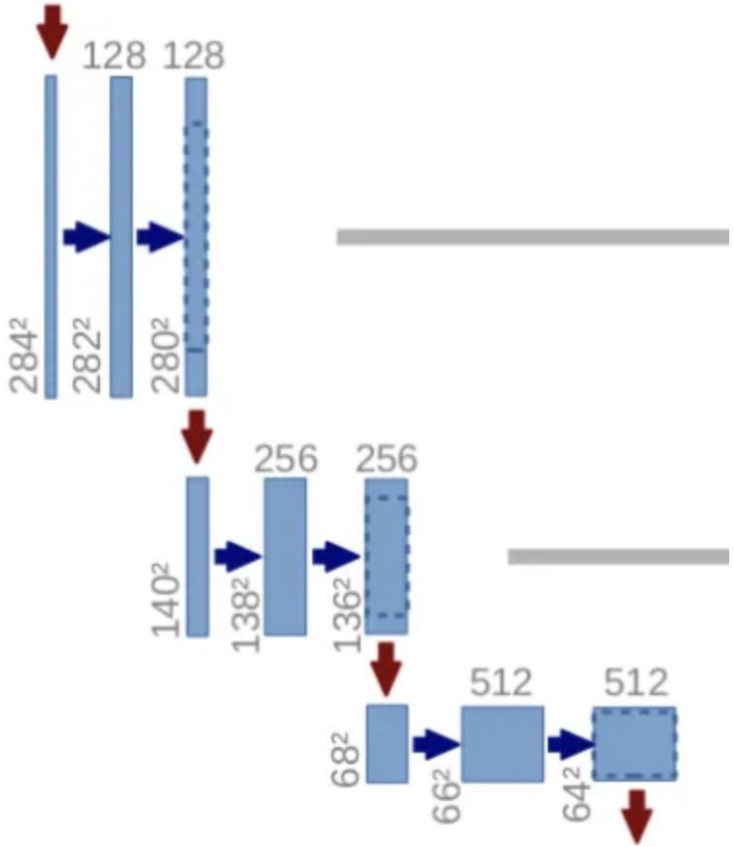
Contracting Path

The contracting path follows the formula:

```
conv_layer1 -> conv_layer2 -> max_pooling -> dropout(optional)
```



The process is repeated 3 more times:



and now we reaches at the bottommost:

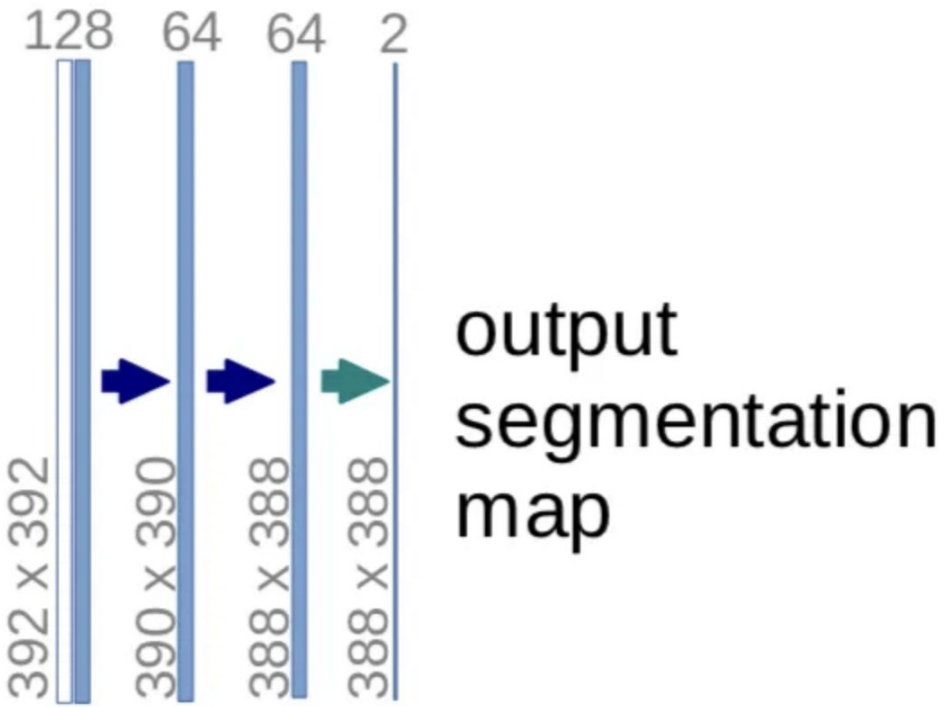
still **2 convolutional layers are built, but with no max pooling**

Expansive Path

In the expansive path, the image is going to be upsized to its original size. The formula follows:

```
conv_2d_transpose -> concatenate -> conv_layer1 -> conv_layer2
```

Now we've reached the uppermost of the architecture, the last step is to reshape the image to satisfy our prediction requirements.



```
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=False):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = (DoubleConv(n_channels, 64))
        self.down1 = (Down(64, 128))
        self.down2 = (Down(128, 256))
        self.down3 = (Down(256, 512))
        factor = 2 if bilinear else 1
        self.down4 = (Down(512, 1024 // factor))
        self.up1 = (Up(1024, 512 // factor, bilinear))
        self.up2 = (Up(512, 256 // factor, bilinear))
        self.up3 = (Up(256, 128 // factor, bilinear))
        self.up4 = (Up(128, 64, bilinear))
        self.outc = (OutConv(64, n_classes))

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

```python
class DoubleConv(nn.Module):
    """(convolution => [BN] => ReLU) * 2"""

    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)
```

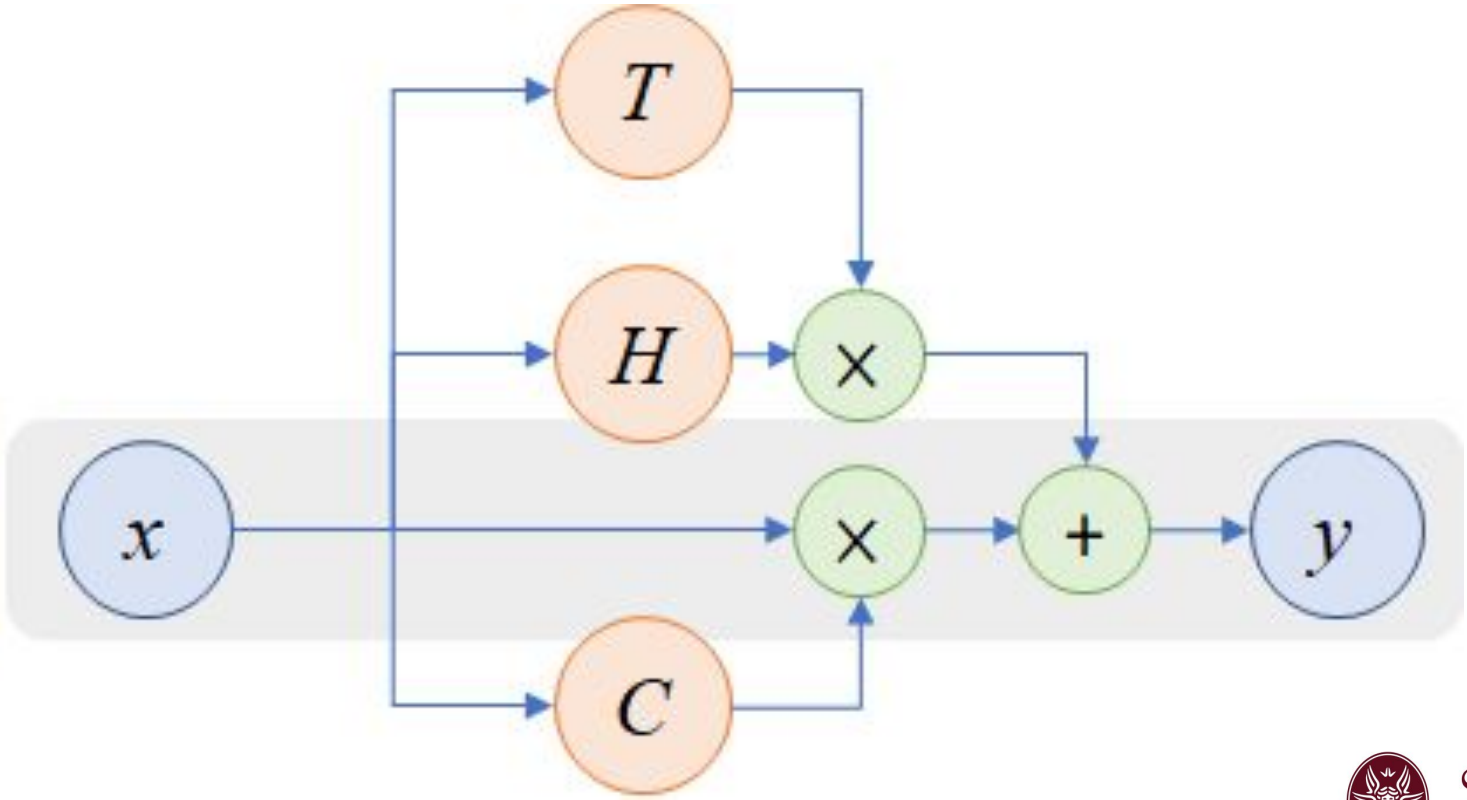
```
class Down(nn.Module):
    """Downscaling with maxpool then double conv"""

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)
```

```python
class Up(nn.Module):
    """Upscaling then double conv"""

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels // 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        # if you have padding issues, see
        # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
        # https://github.com/xiaopeng-liao/Pytorch-UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

```
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

    def forward(self, x):
        return self.conv(x)
```

Highway Networks

Generalization of Residual Connections



Generalization of Residual Connections

- A plain feedforward neural network

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H})$$

- For a highway network, we additionally define two non-linear transforms T($\mathbf{x}$,$\mathbf{W}_T$) and C($\mathbf{x}$,$\mathbf{W}_C$) such that

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W_C})$$

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

- Usually, we set C = 1 − T, giving

$$\mathbf{y} = H(\mathbf{x}, \mathbf{W_H}) \cdot T(\mathbf{x}, \mathbf{W_T}) + \mathbf{x} \cdot (1 - T(\mathbf{x}, \mathbf{W_T}))$$

Deep Learning

End of Lecture

06 - Convolutional Neural Networks and Resnets

SAPIENZA
UNIVERSITÀ DI ROMA

Fabrizio Silvestri