# Foundations for Machine Learning

L. Y. Stefanus

TU Dresden, June-July 2018

Slides 11

# Reference

- Ian Goodfellow and Yoshua Bengio and Aaron Courville. DEEP LEARNING. MIT Press, 2016.

# Convolutional Neural Networks (CNN)

# Challenges for AI

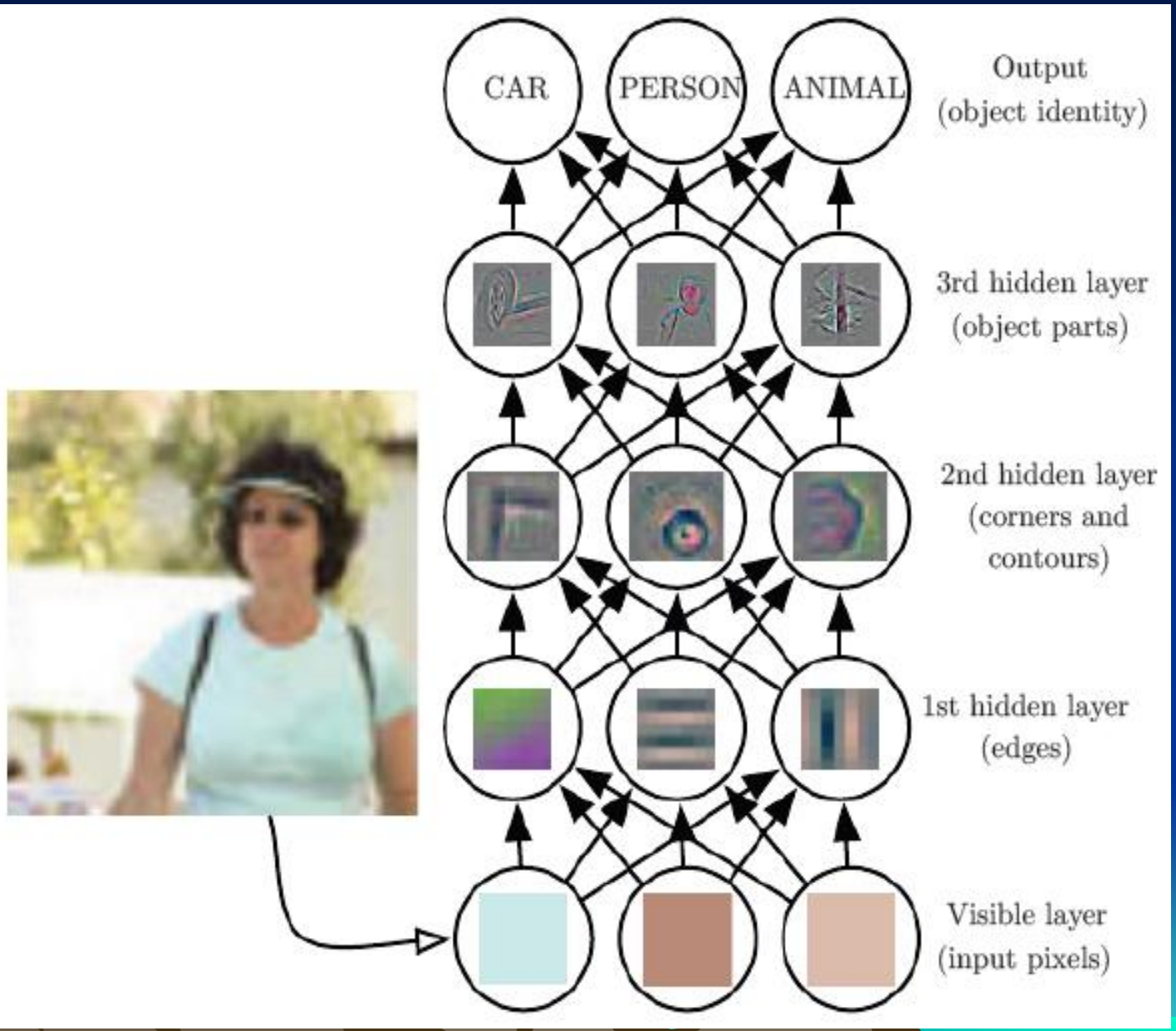
- In the early days of AI, it tackled and solved problems that are intellectually difficult for human beings but relatively straightforward for computers—problems that can be described by a list of formal, mathematical rules.
- The true challenge to AI proved to be solving the tasks that are easy for people to perform but hard for people to describe formally—problems that we solve intuitively, that feel automatic, like recognizing spoken words or faces in images.

# Deep Learning

- Deep Learning is about finding a solution to these more intuitive problems. This solution is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts.
- By gathering knowledge from experience, this approach avoids the need for human operators to formally specify all the knowledge that the computer needs.
- The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones.

# Deep Learning

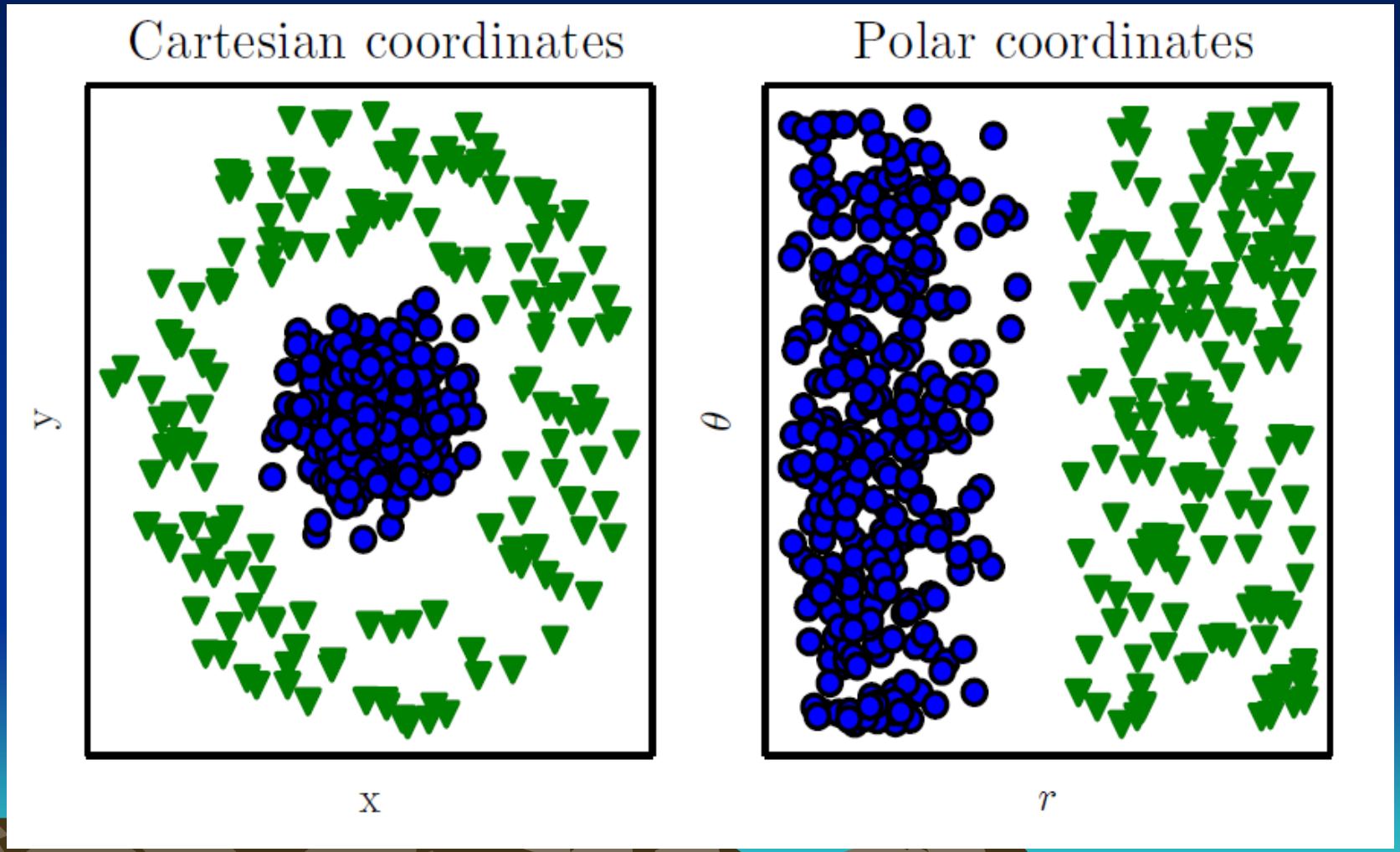
- If we draw a graph showing how these concepts are built on top of each other, the graph is deep, with many layers. Therefore, we call this approach deep learning.


CAR
PERSON
ANIMAL
Output
(object identity)
3rd hidden layer
(object parts)
2nd hidden layer
(corners and
contours)
1st hidden layer
(edges)
Visible layer
(input pixels)

# Representations are very important

- Suppose we want to separate two categories of data by drawing a line between them in a scatterplot.
- If we represent the data using Cartesian coordinates, the task is impossible. But, if we represent the data with polar coordinates, the task becomes simple to solve with a vertical line.

# Representations are very important



# Representations Learning

- The task of machine learning is to discover not only the mapping from representation to output but also the representation itself.
- This approach is known as representation learning.
- Learned representations often result in much better performance than can be obtained with hand-designed representations. They also enable AI systems to rapidly adapt to new tasks, with minimal human intervention.
- The common example of a representation learning algorithm is the autoencoder.

# Representations Learning

- An autoencoder is the combination of an encoder function, which converts the input data into a different representation, and a decoder function, which converts the new representation back into the original format.
- Autoencoders are trained to preserve as much information as possible when an input is run through the encoder and then the decoder, but they are also trained to make the new representation have various nice properties.
- Different kinds of autoencoders aim to achieve different kinds of properties.

# factors of variation

- One common goal of learning is to separate the factors of variation that explain the observed data.
- For example: When analyzing an image of a car, the factors of variation include the position of the car, its color, and the angle and brightness of the sun.
- A major source of difficulty in many real-world AI applications is that many of the factors of variation influence every single piece of data we are able to observe. For example: The individual pixels in an image of a red car might be very close to black at night. The shape of the car's silhouette depends on the viewing angle.

# factors of variation

- Most applications require us to disentangle the factors of variation and discard the ones that we do not care about.
- Deep Learning can solve this problem.
- The idea of learning the right representation for the data provides one perspective on deep learning. Another perspective on deep learning is that depth enables the computer to learn a multistep computer program.

# Applications of Deep Learning

- Deep learning has already proved useful in many software disciplines, including computer vision, speech and audio processing, natural language processing, robotics, bioinformatics, video games, search engines, online advertising and finance.

# Intro to Convolutional Networks

- In 1989, LeCun introduced the convolutional networks, also known as convolutional neural networks (CNN). They are a specialized kind of neural network for processing data that has a known grid-like topology.
- Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels.
- Convolutional networks have been very successful in practical applications.

# Intro to Convolutional Networks

- The name “convolutional neural network” indicates that the network employs the convolution operation.
- Convolutional neural networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

# The Convolution Operation

- As explained in the previous lectures, convolution is an operation on two functions of a real-valued argument in the continuous case or an operation on two sequences of real values in the discrete case.
- Continuous case: Given functions x(u) and h(u), the convolution of x(u) and h(u) is:

$$c(t) = (x*h)(t) = \int_{-\infty}^{+\infty} x(u)h(t-u)du$$

- Discrete case: Given sequences $a = (a_0, a_1, \ldots, a_{m-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, the convolution of a and b is sequence $c = (c_0, c_1, \ldots, c_{n+m-2})$, where

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}$$

# The Convolution Operation

- In convolutional neural network terminology, the first argument (the function $x$ or the sequence $a$) to the convolution is often referred to as the input, and the second argument (the function h or the sequence $b$) as the kernel. The output is sometimes referred to as the feature map.

# The Convolution Operation

- In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm.
- These multidimensional arrays are usually called tensors.
- Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but in the finite set of points for which we store the values. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements.

# The Convolution Operation

- Furthermore, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image *I* as our input, we also want to use a two-dimensional kernel *K*:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \quad (9.4)$$

- Convolution is commutative, so we can equivalently write:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \quad (9.5)$$

# The Convolution Operation

- As we have seen in the previous lectures, the commutative property of convolution arises because we have flipped the kernel relative to the input.
- While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation.
- Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

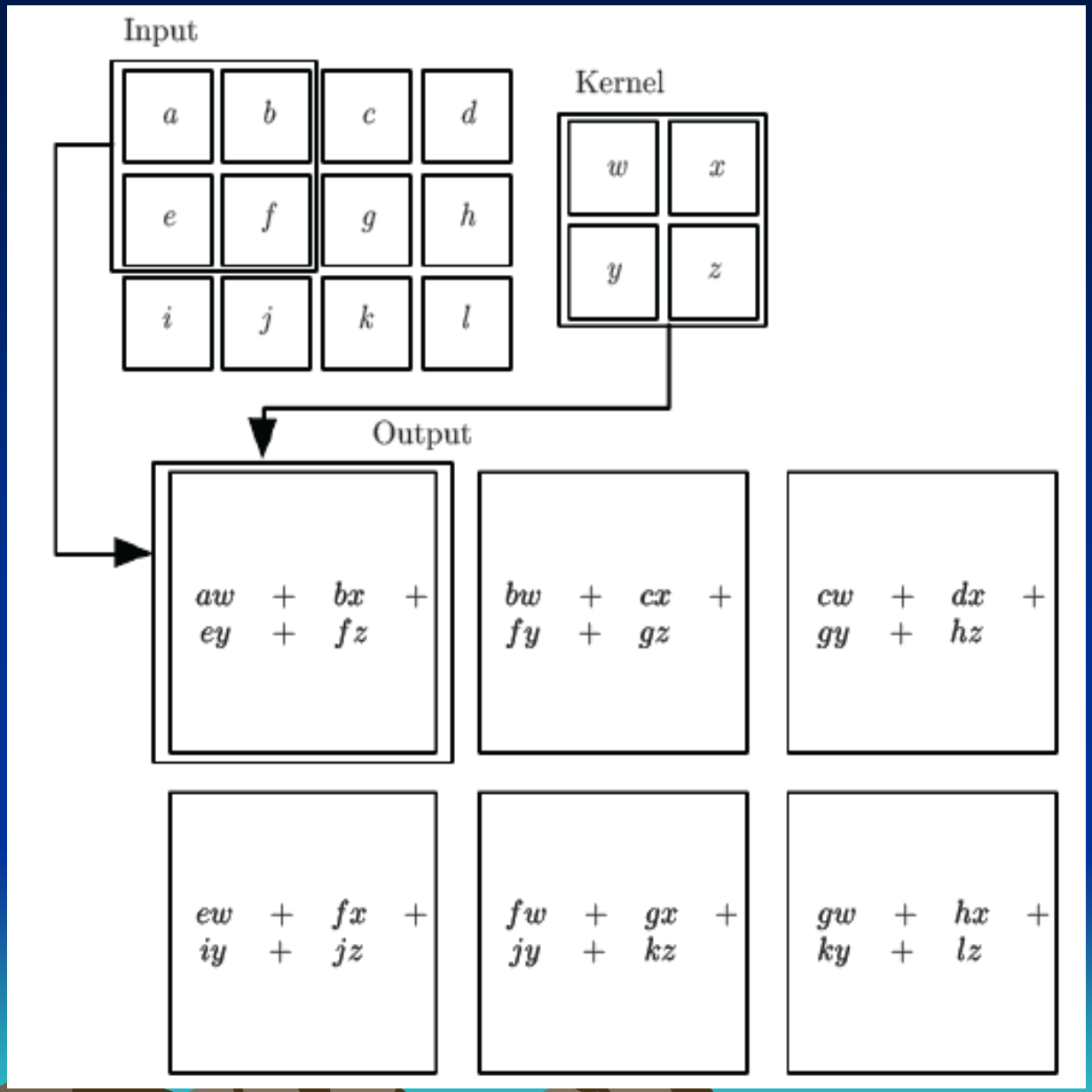
# The Convolution Operation

Cross-Correlation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n). \quad (9.6)$$

- Many machine learning libraries implement cross-correlation but call it convolution.

Convolution (without kernel flipping) applied to a 2-D tensor. The output is restricted to only positions where the kernel lies entirely within the input tensor. This is called “valid” convolution in some contexts.



# Block matrices

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \vdots & \vdots \\ A_{M1} & A_{M2} & \cdots & A_{MN} \end{bmatrix}$$

- $A_{ij}$ are matrices.
- If the structure of $A$, with respect to its sub-matrices, is Toeplitz then matrix $A$ is called **block-Toeplitz.**
- If each individual $A_{ij}$ is also a Toeplitz matrix then $A$ is called **doubly block-Toeplitz**.
- Consider an example:

# 2D convolution using doubly block Toeplitz matrices

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix}$$

$f[m,n]$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$h[m,n]$

$M_1 \text{ x } N_1 = 2\text{x}3$

$M_2 \text{ x } N_2 = 2\text{x}2$

The result will be of size $(M_1+M_2\text{-}1) \text{ x } (N_1+N_2\text{-}1) = 3 \text{ x } 4$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix}$$

$f[m,n]$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$h[m,n]$

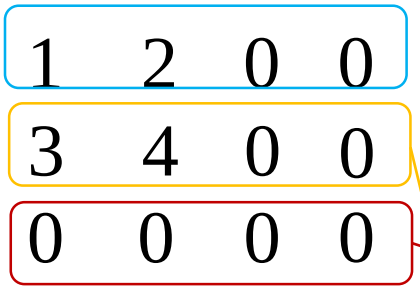
- Firstly, $h[m,n]$ is zero-padded to $3 \times 4$ (the size of the result).
- Then, for each row of $h[m,n]$, a Toeplitz matrix with 3 columns (the number of **columns** of $f[m,n]$) is constructed.

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$h[m,n]$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

- For each row of $h[m,n]$, a Toeplitz matrix with 3 columns (the number of columns of $f[m,n]$) is constructed.




$h[m,n]$

$$H_1 = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad H_2 = \begin{bmatrix} 3 & 0 & 0 \\ 4 & 3 & 0 \\ 0 & 4 & 3 \\ 0 & 0 & 4 \end{bmatrix} \quad H_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix}$$

$f[m,n]$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$h[m,n]$

- Using matrices $\mathbf{H}_1$, $\mathbf{H}_2$ and $\mathbf{H}_3$ as elements, a doubly block Toeplitz matrix $\mathbf{H}$ is then constructed with 2 columns (the number of **rows** of $f[m,n]$).

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 & \mathbf{H}_3 \\ \mathbf{H}_2 & \mathbf{H}_1 \\ \mathbf{H}_3 & \mathbf{H}_2 \end{bmatrix}_{12\times 6}$$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix}$$

$f[m,n]$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$h[m,n]$

- We now construct a vector from the elements of $f[m,n]$.

$$\mathbf{f} = \begin{bmatrix} 2 \\ 5 \\ 3 \\ \hline 1 \\ 4 \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|ccc} (2 & 5 & 3)^T \\ \hline (1 & 4 & 1)^T \end{array}\right]$$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$f[m,n]$ $\qquad$ $h[m,n]$

$$\mathbf{g} = \mathbf{Hf} = \begin{bmatrix} \mathbf{H}_1 & \mathbf{H}_3 \\ \mathbf{H}_2 & \mathbf{H}_1 \\ \mathbf{H}_3 & \mathbf{H}_2 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \\ 3 \\ 1 \\ 4 \\ 1 \end{bmatrix}$$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\mathbf{g} = \mathbf{H}\mathbf{f} = \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ \hline 3 & 0 & 0 & 1 & 0 & 0 \\ 4 & 3 & 0 & 2 & 1 & 0 \\ 0 & 4 & 3 & 0 & 2 & 1 \\ 0 & 0 & 4 & 0 & 0 & 2 \\ \hline 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 4 & 3 \\ 0 & 0 & 0 & 0 & 0 & 4 \end{array}\right] \left[\begin{array}{c} 2 \\ 5 \\ 3 \\ \hline 1 \\ 4 \\ 1 \end{array}\right] = \left[\begin{array}{c} 2 \\ 9 \\ 13 \\ 6 \\ \hline 7 \\ 29 \\ 38 \\ 14 \\ \hline 3 \\ 16 \\ 19 \\ 4 \end{array}\right]$$

# 2D linear convolution using doubly block Toeplitz matrices (cont.)

$$\begin{bmatrix} 2 & 5 & 3 \\ 1 & 4 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$f[m,n]$ $\qquad$ $h[m,n]$

$$= \begin{bmatrix} 2 & 9 & 13 & 6 \\ 7 & 29 & 38 & 14 \\ 3 & 16 & 19 & 4 \end{bmatrix}$$

$g[m,n]$

# 2D Convolution in Python

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jul 13 18:34:45 2018
@author: L Y Stefanus
"""
import numpy as np
from scipy import signal
a = np.array([[2,5,3],[1,4,1]])
print("a:\n",a)
b = np.array([[1,2],[3,4]])
print("b:\n",b)
c = signal.convolve2d(a,b)
print("a*b:\n",c)
```

# 2D Convolution in Python

```
>>>
a:
 [[2 5 3]
 [1 4 1]]
b:
 [[1 2]
 [3 4]]
a*b:
 [[ 2  9 13  6]
 [ 7 29 38 14]
 [ 3 16 19  4]]
```

# Circulant Matrices

- An n x n circulant matrix C takes the form

$$C = \begin{bmatrix} c_0 & c_{n-1} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ & & \ddots & \ddots & \\ c_{n-2} & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \end{bmatrix}.$$

# Circulant Matrices

- A **circulant matrix** is a special kind of Toeplitz matrix where each column is rotated downward one element relative to the preceding column. Circulant matrices are important because they can be diagonalized by a DFT (discrete Fourier transform), hence their eigenvalues, eigenvectors, and inverses can be computed efficiently.

# circular convolution

- Circular convolution uses circulant matrices instead of general Toeplitz matrices.

# The Convolution Operation

- Convolution supports three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations.
- Convolution also provides a means for working with inputs of variable size.

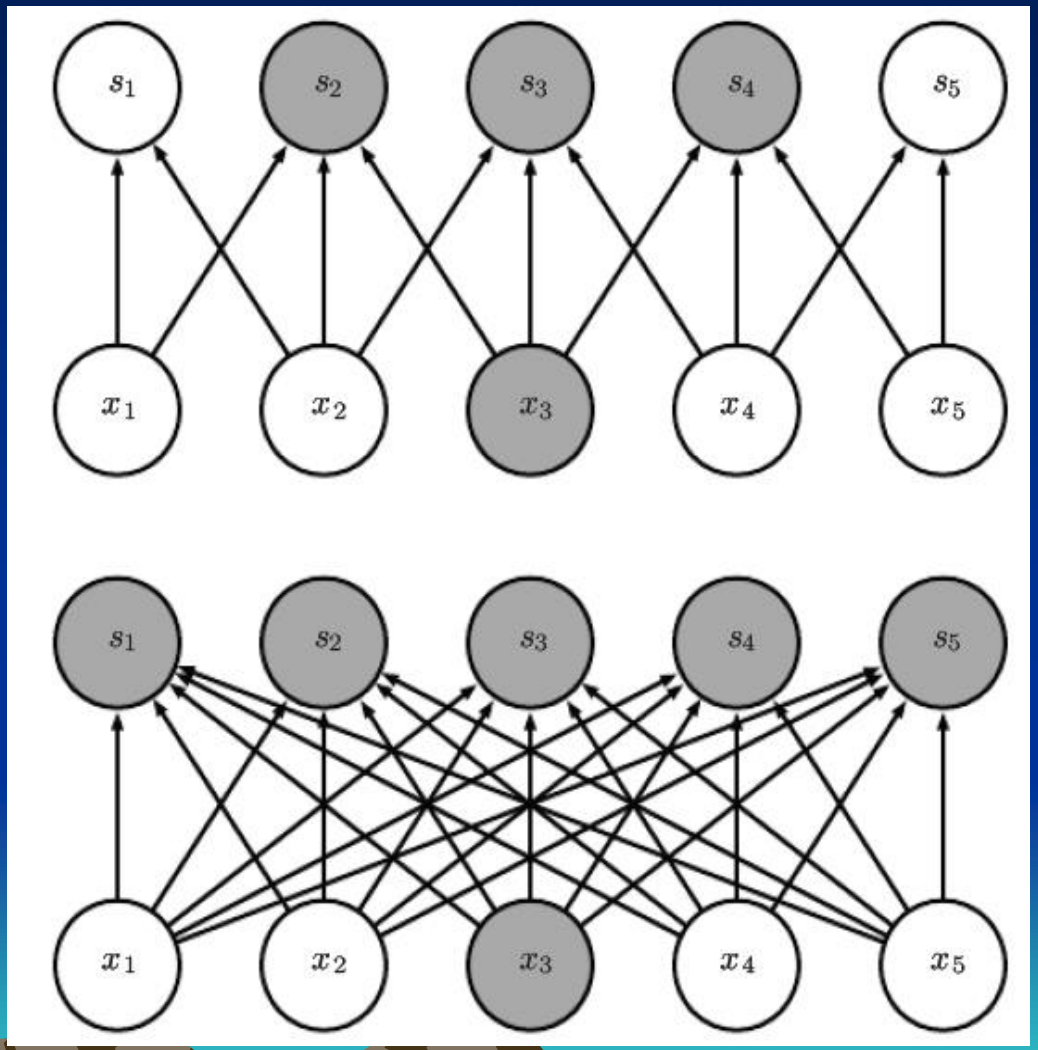
# Sparse Interactions

- Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit.
- Convolutional neural networks, however, typically have sparse interactions (sparse connectivity or sparse weights).
- This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only hundreds of pixels.

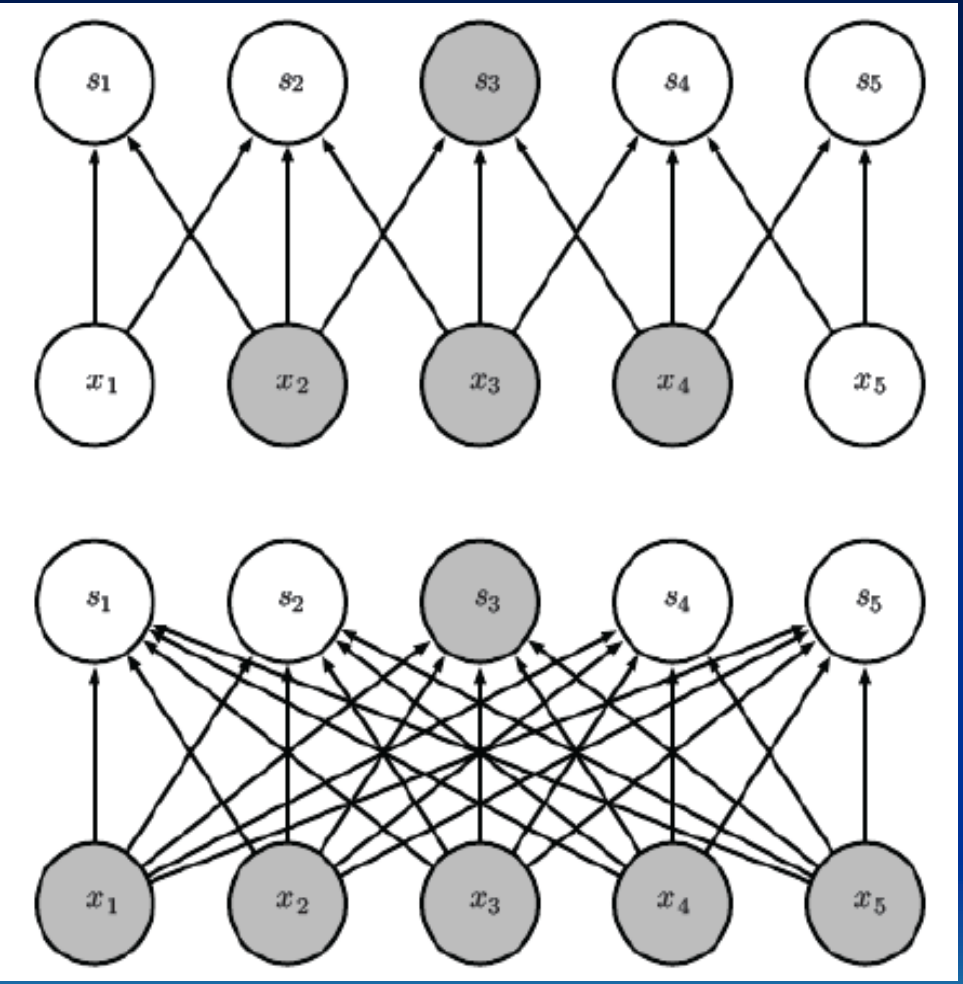
# Sparse Interactions

- This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.
- In a deep convolutional neural network, units in the deeper layers may *indirectly* interact with a larger portion of the input.
- This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

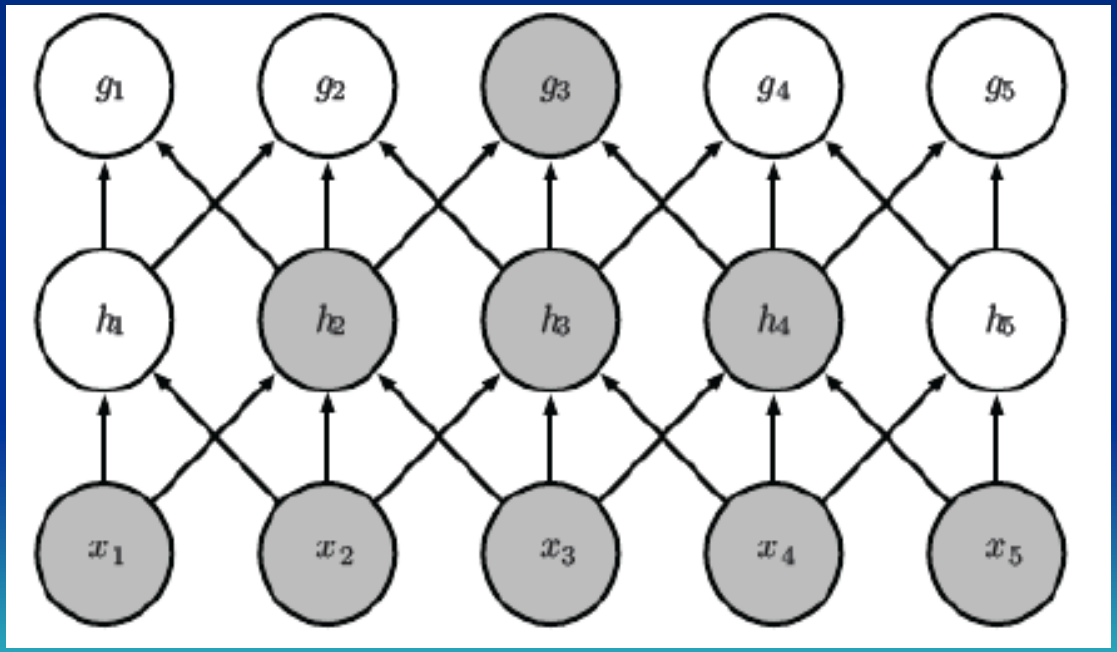
Sparse connectivity, viewed from below.
Consider one input unit, *x*3, and some output units in layer *s* that are affected by this unit.
Top part: When *s* is formed by convolution with a kernel of width 3, only three output units are affected by *x*3.
Bottom part: When *s* is formed by ordinary matrix multiplication, connectivity is no longer sparse, so all the output units are affected by *x*3.



Sparse connectivity, viewed from above.
Consider one output unit, *s*3, and some input units that affect this unit. These units are known as the receptive field of *s*3.
Top part: When layer *s* is formed by convolution with a kernel of width 3, only three input units affect *s*3.
Bottom part: When *s* is formed by ordinary matrix multiplication, connectivity is no longer sparse, so all the input units affect *s*3.



- The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers.
- This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.



# parameter sharing

- Parameter sharing refers to using the same parameter for more than one function in a model.
- In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never reused.
- In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary).
- The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set.

# parameter sharing

- As an example, let's consider how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

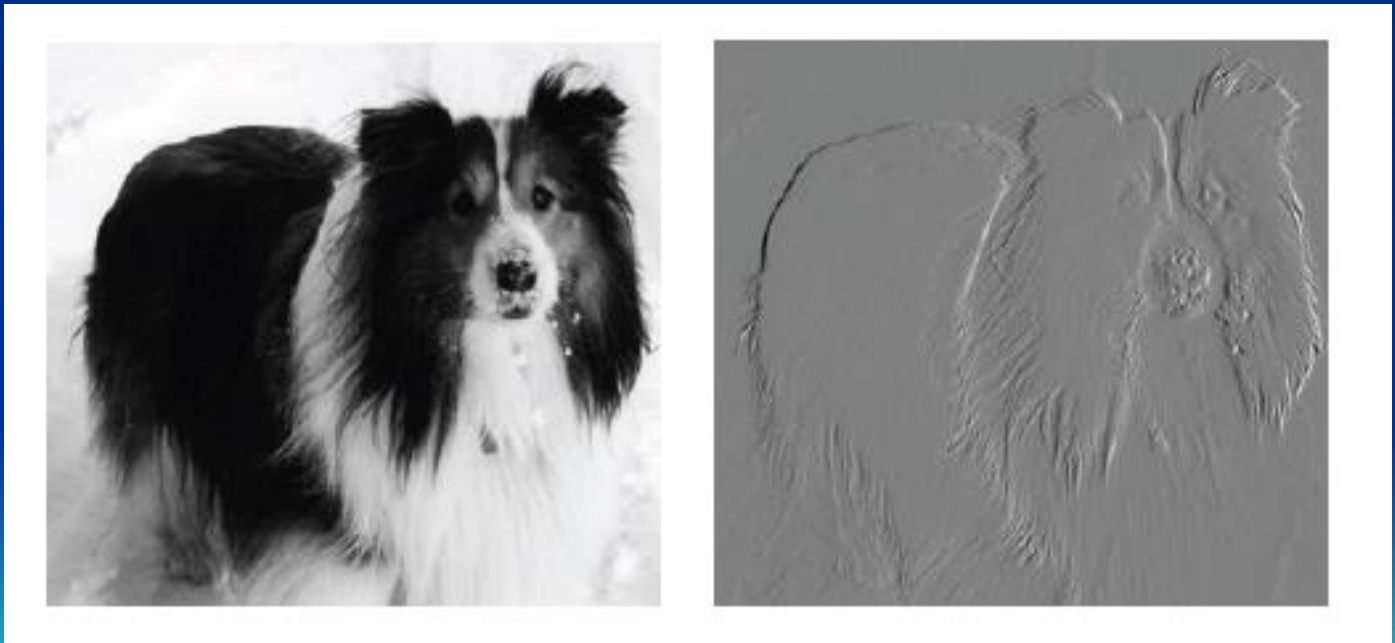


Photo credit: Paula Goodfellow.

- The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all the vertically oriented edges in the input image, which can be a useful operation for object detection.
- Both images are 280 pixels tall. The input image is 320 pixels wide, while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267\,960$ floating-point operations (two multiplications and one addition per output pixel) to compute using convolution.
- The same transformation with ordinary matrix multiplication would take $320 \times 280 \times 319 \times 280 = 8\,003\,072\,000$, or over eight billion, entries in the weight matrix, making convolution four billion times more efficient for representing this transformation.

- The straightforward matrix multiplication algorithm performs over sixteen billion floating point operations, making convolution roughly 60,000 times more efficient computationally.
- Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small local region across the entire input.

# Equivariance

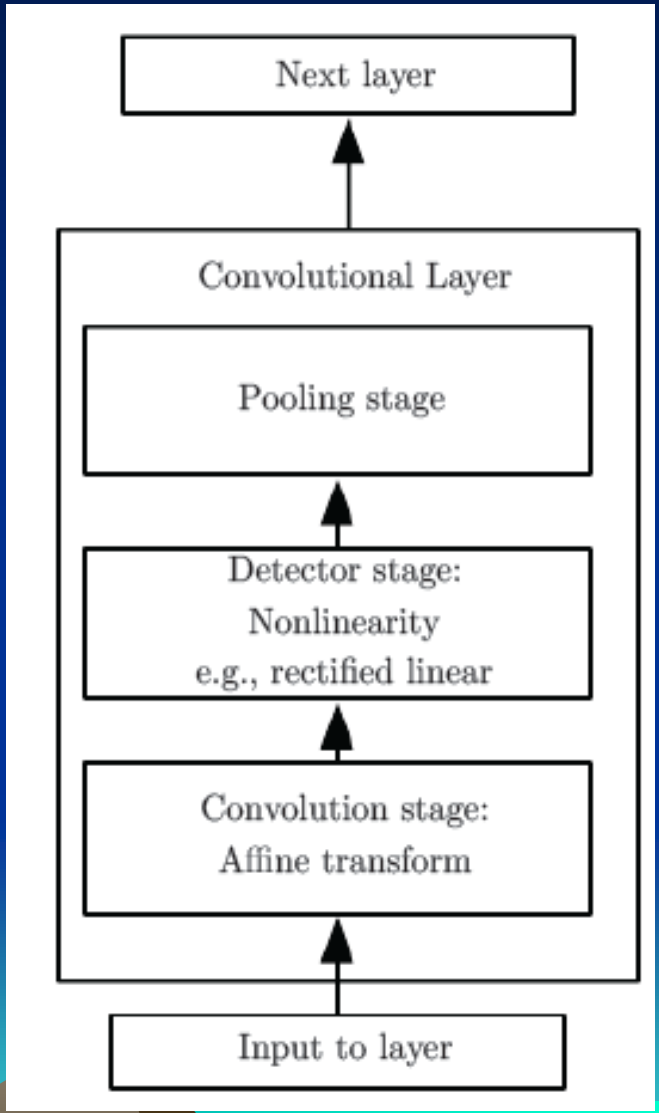
- In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation.
- To say a function is equivariant means that if the input changes, the output changes in the same way.
- Specifically, a function $f(x)$ is equivariant to a function $g$ if $f(g(x)) = g(f(x))$.
- In the case of convolution, if we let $g$ be any function that translates (shifts) the input, then the convolution function is equivariant to $g$.

# Equivariance

- When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later.
- Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

# Pooling

- A typical layer of a convolutional neural network consists of three stages:
- In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations.
- In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function (ReLU).
- In the third stage, a pooling function is used to modify the output of the layer further.



# Pooling

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
- For example, the max pooling operation reports the maximum output within a rectangular neighborhood.
- Other popular pooling functions include the average of a rectangular neighborhood, the $L^2$ norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

# Pooling

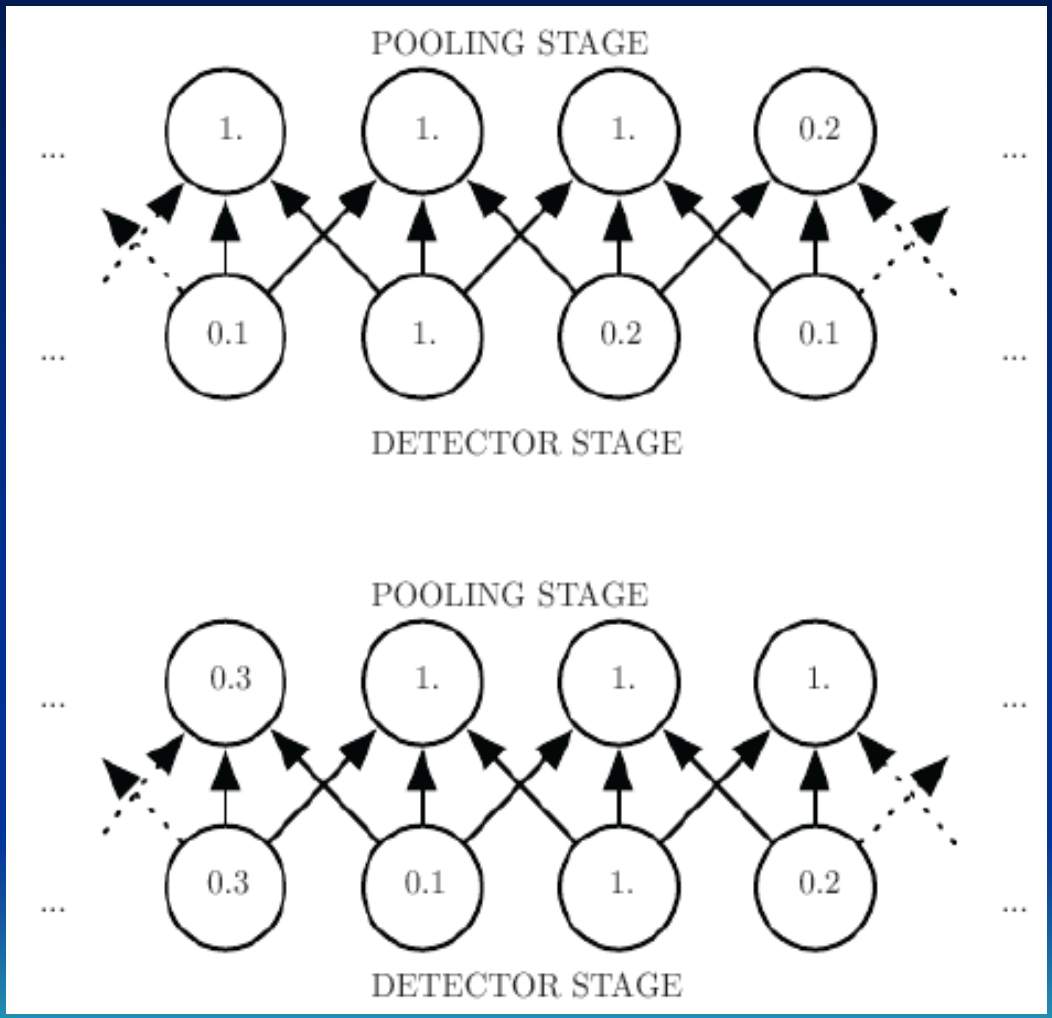
- In all cases, pooling helps to make the representation approximately invariant to small translations of the input.
- Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.
- Invariance to local translation can be a useful property if we care more about whether some feature is present than exactly where it is.
- For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy.

# Pooling

- The use of pooling can be viewed as adding an infinitely strong prior knowledge that the function the layer learns must be invariant to small translations.
- When this assumption is correct, it can greatly improve the statistical efficiency of the network.
- Example:

# Max pooling causes invariance.

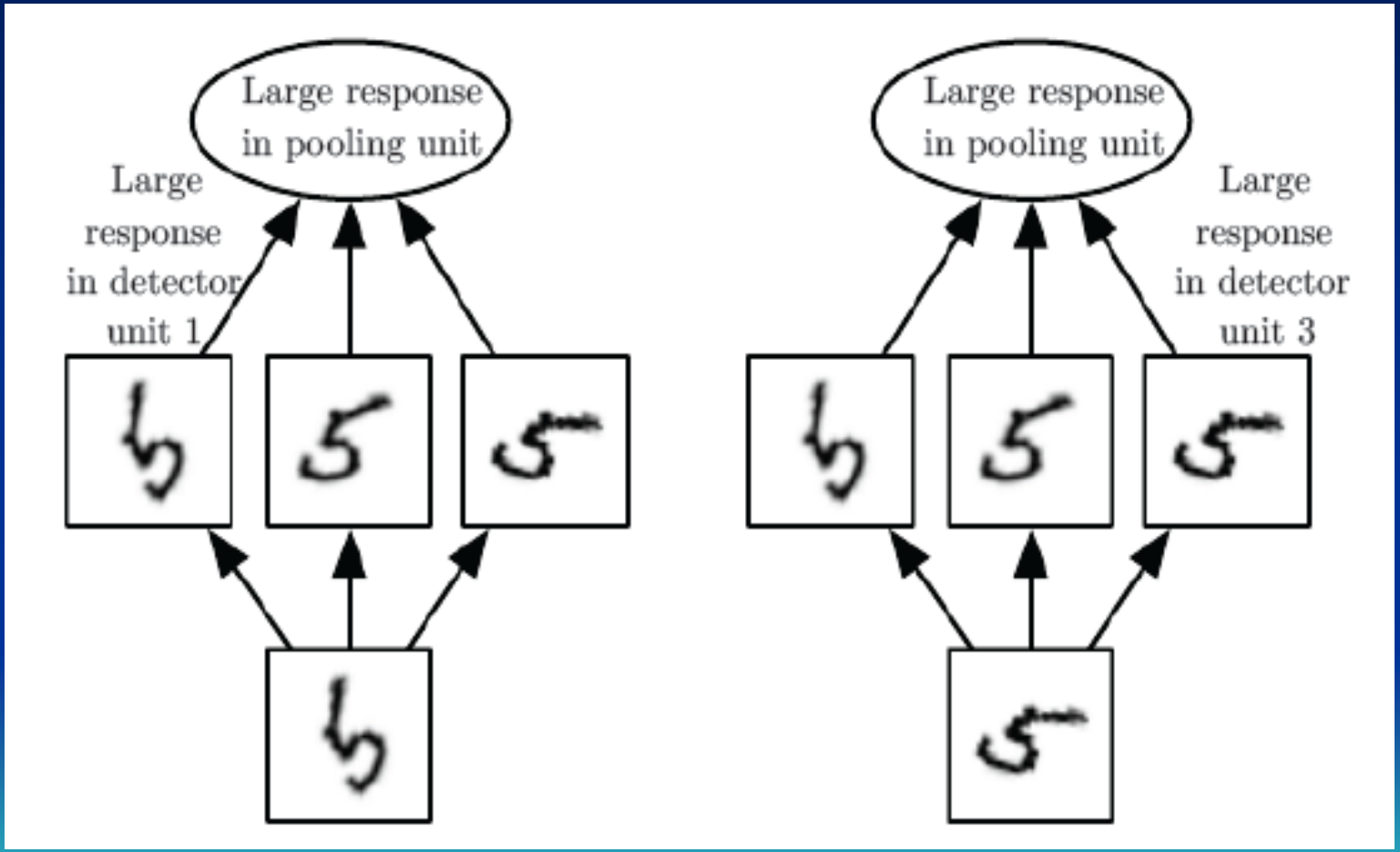
Top part: A view of the middle of the output of a convolu-tional layer. The bottom row shows outputs of the non-linearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions
and a pooling region width of three pixels.
Bottom part: A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location.



# learned invariances

- Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to.
- Example:

# Example of learned invariances



- A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input.
- Example of how a set of three learned filters and a max pooling unit can learn to become invariant to rotation.
- All three filters are intended to detect a hand written 5. Each filter attempts to match a slightly different orientation of the 5.
- When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which detector unit was activated.
- This shows how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way.
- Max pooling over spatial positions is naturally invariant to translation; this multichannel approach is only necessary for learning other transformations.

# pooling with downsampling

- Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced *k* pixels apart rather than 1 pixel apart.
- This improves the computational efficiency of the network because the next layer has roughly *k* times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication), this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.

- Here we use max pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer.
- Note that the rightmost pooling region has a smaller size but must be included if we do not want to ignore some of the detector units.