SDU Summer School

Deep Learning

Summer 2018

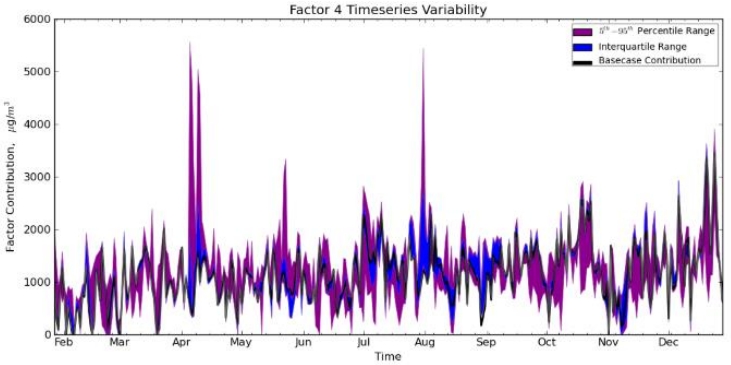
Convolutional Neural Networks

Convolutional Neural Networks

- Convolution
- Pooling
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

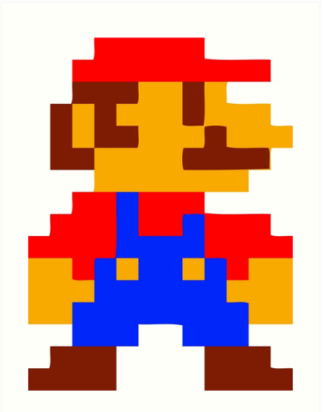
Convolutional Networks

- Convolutional neural networks (CNNs) are a specialized kind of neural network
  - Neural networks that use **convolution** in place of general matrix multiplication in at least one of their layers



E.g. Ex: time-series data, which is a 1-D grid, taking samples at intervals

- Well suited for data with a grid-like topology



More obviously: Image data, which are 2-D grid of pixels

- But: Convolution can be viewed as multiplication by a matrix

What is Convolution?

- Convolution is an operation on two functions of a real-valued argument
- Examples of the two functions
  - Tracking location of a spaceship by a laser sensor
    - A laser sensor provides a single output $x(t)$, the position of spaceship at time t
  - w a function of a real-valued argument
    - If laser sensor is noisy, we want a weighted average that gives more weight to recent observations
    - Weighting function is $w(a)$ where a is age of measurement
- Convolution is the smoothed estimate of the position of the spaceship

$$s(t) = \int x(a)w(t-a)da$$

What is Convolution?

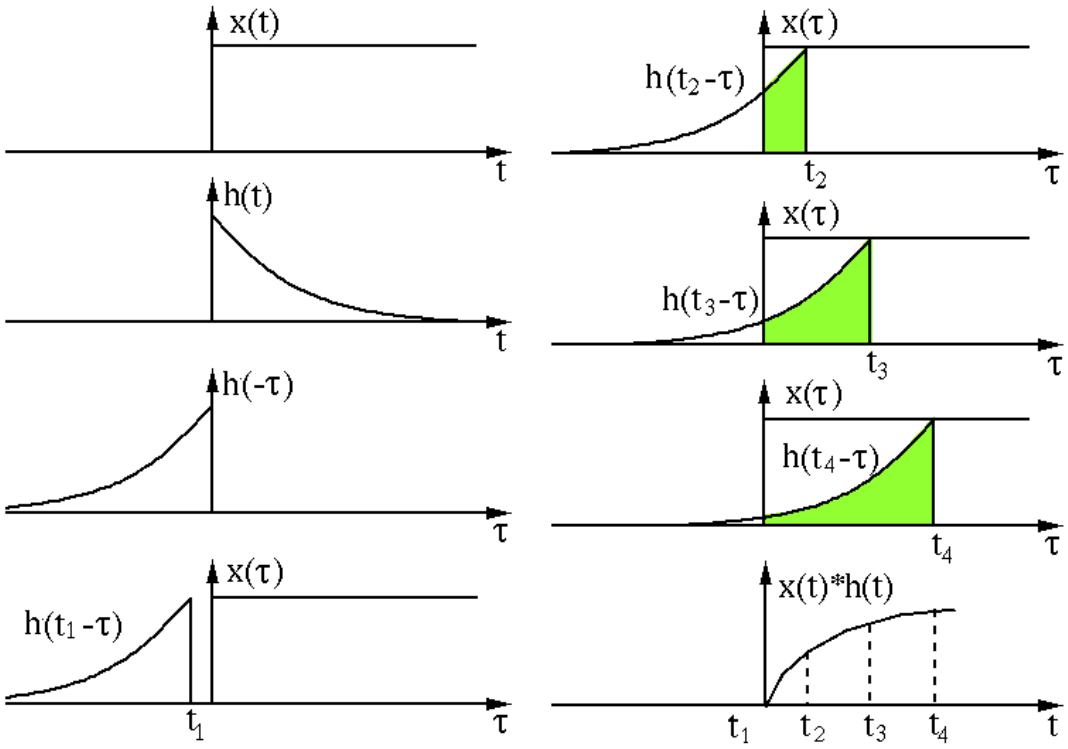
- One-dimensional continuous case
  - Input $f(t)$ is convolved with a kernel $g(t)$

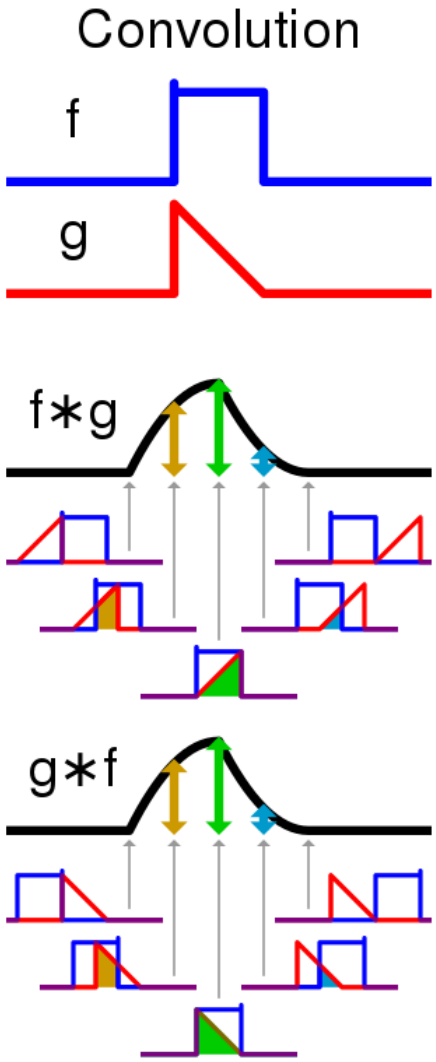
$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

- Note that $(f * g)(t) = (g * f)(t)$

- Terminology:
  - The first argument (here the function f) to the convolution is often referred to as the **input** and
  - The second argument (in this example, the function g) as the **kernel**.
  - The output is sometimes referred to as the **feature map**.

Visual Explanations of Convolution!





- http://fourier.eng.hmc.edu/e161/lectures/convolution/index.html
- www.wikipedia.org

Convolution with Discrete Variables

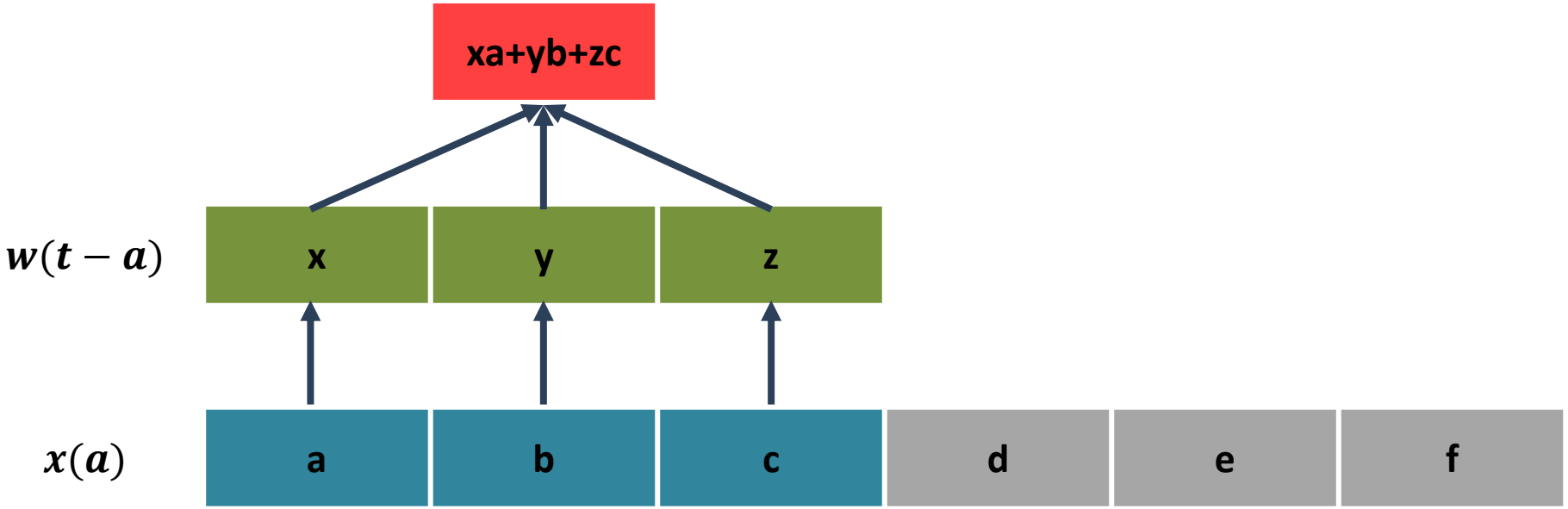
- Laser sensor may only provide data at regular intervals
- Time index t can take on only integer values
  - x and w are defined only on integer t

$$s(t) = (x * w)(t) \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

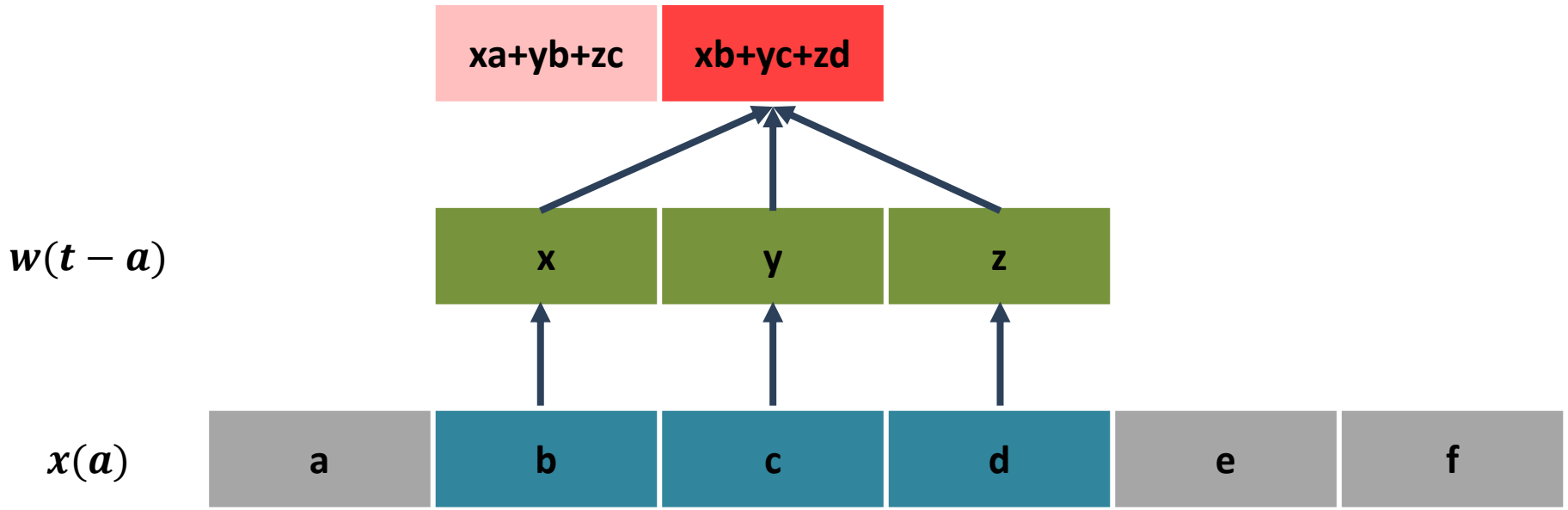
- In ML applications, input is a multidimensional array of data and the kernel is a multidimensional array of parameters that are adapted by the learning algorithm

- Input and kernel are explicitly stored separately

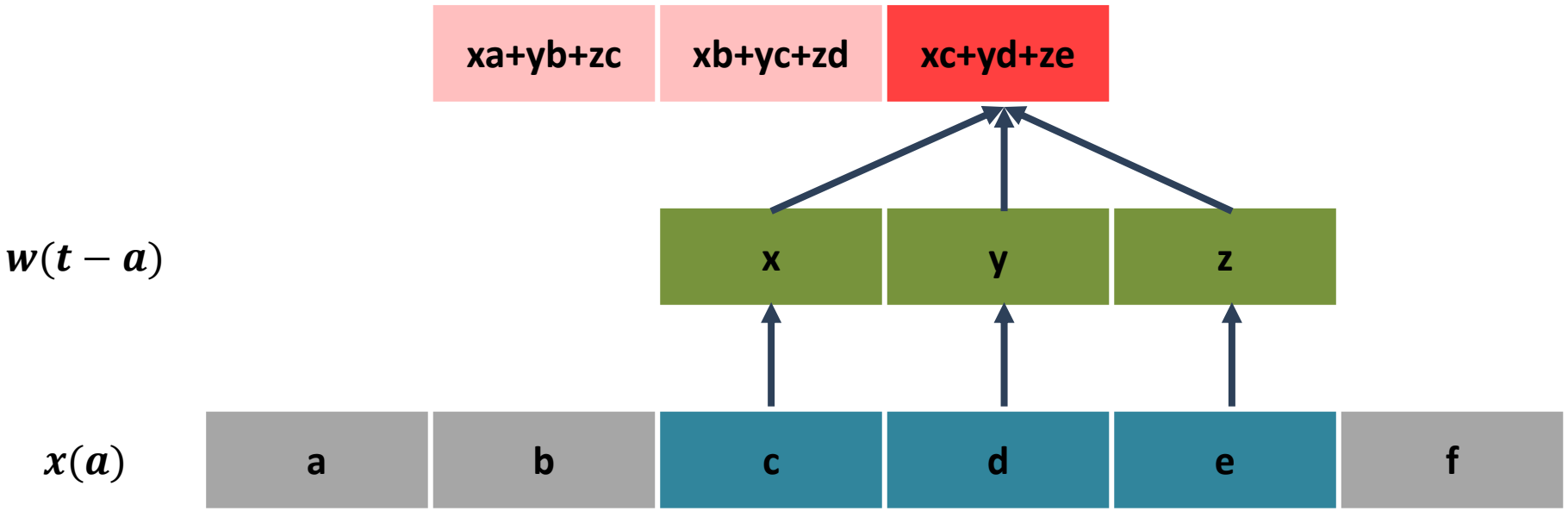
Discrete 1-Dimensional Case



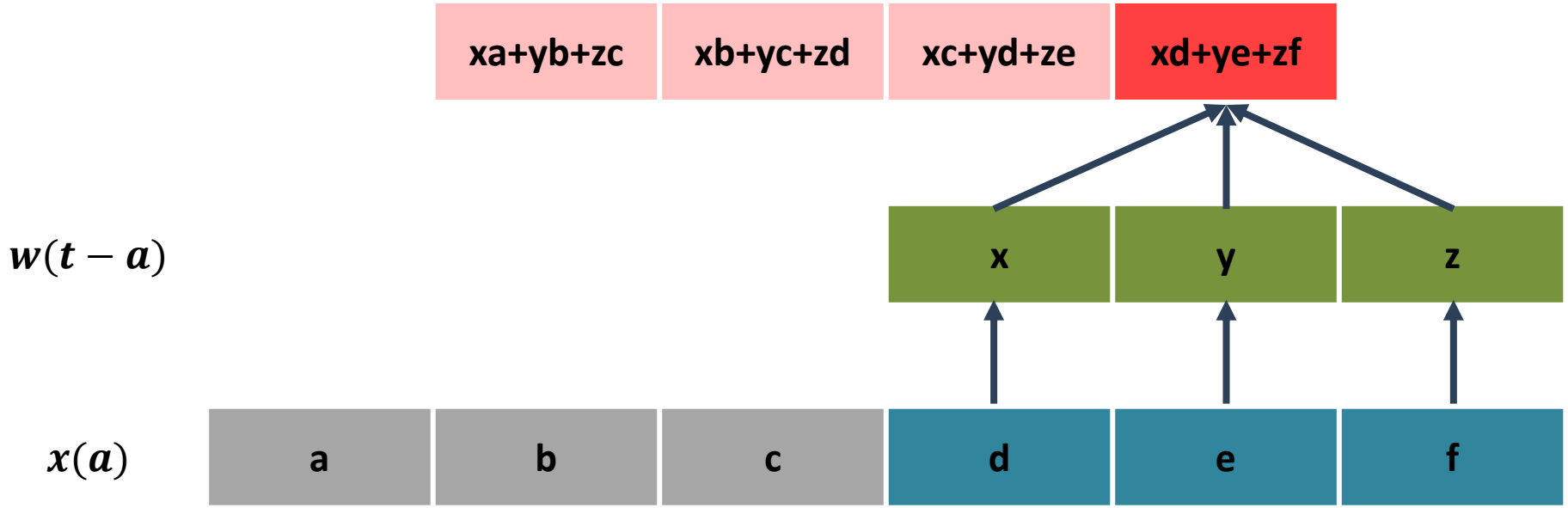
Discrete 1-Dimensional Case



Discrete 1-Dimensional Case

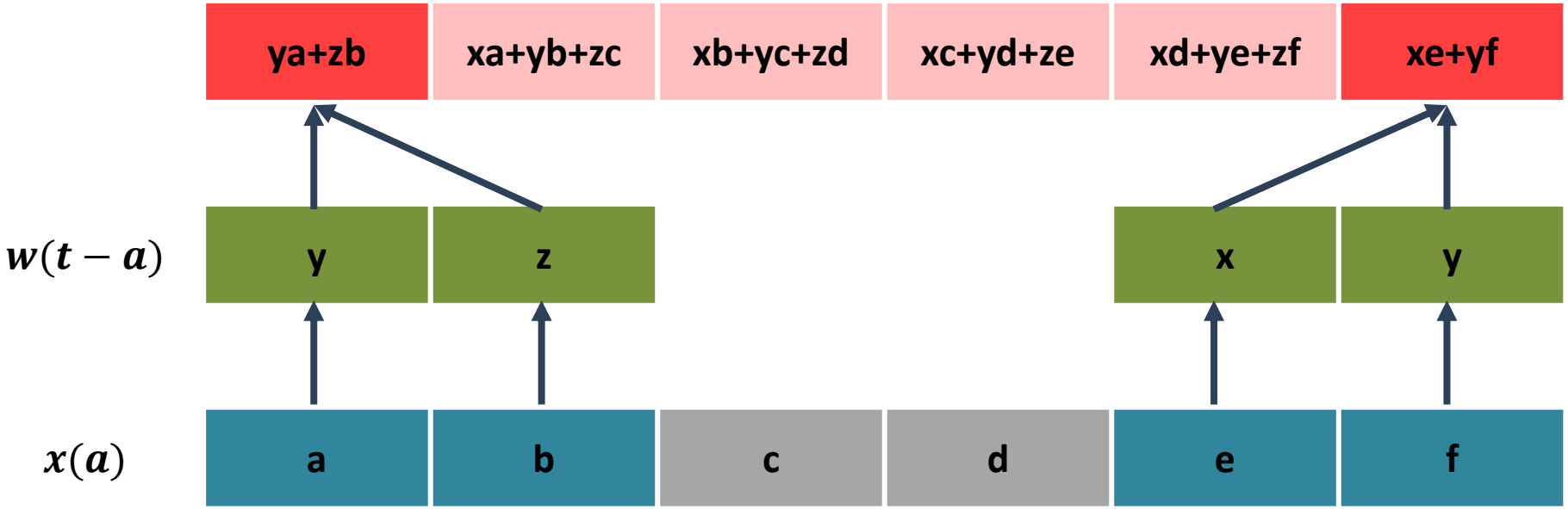


Discrete 1-Dimensional Case

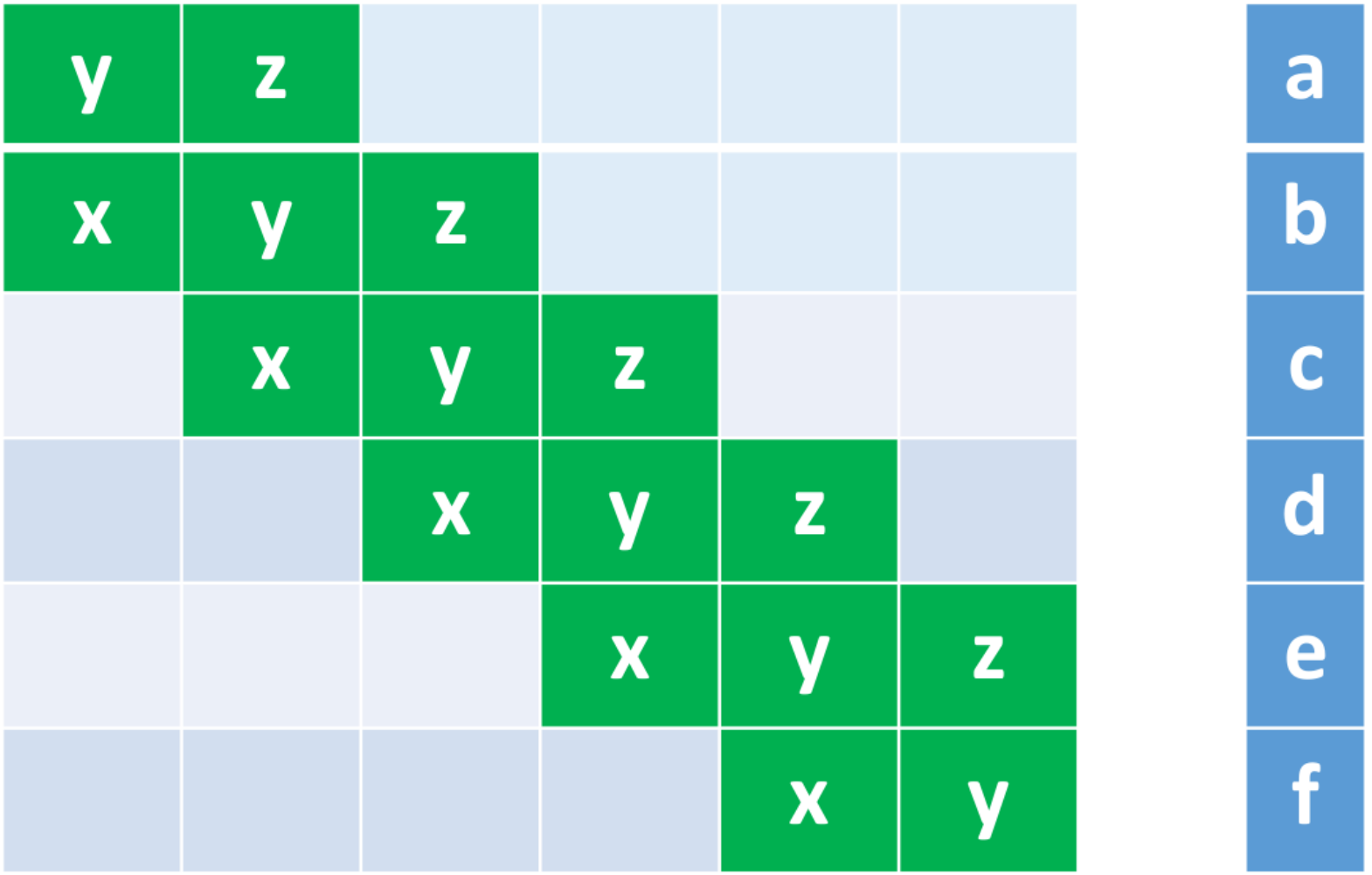


Discrete 1-Dimensional Case: Border Cases

- Border require special treatment
- Most commonly they are ignored
- Sometimes a smaller kernel is used then

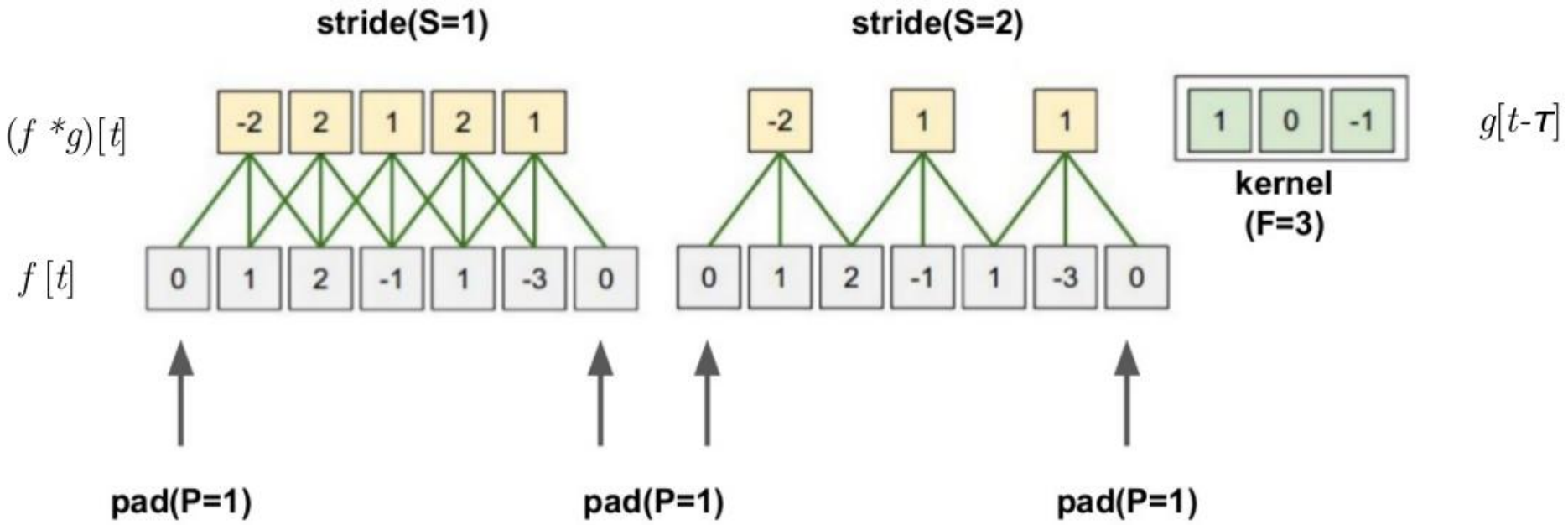


Seen as a Matrix Multiplication



Parameters of Convolution

- Kernel Size (F)
- Padding (P)
- Stride (S)



Two-dimensional Convolution

- If we use a 2D image I as input and use a 2D kernel K we have

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

- Since convolution is commutative, we can also write:

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

- Commutativity arises because we have flipped the kernel relative to the input
  - As m increases, index to the input increases, but index to the kernel decreases

Cross-Correlation

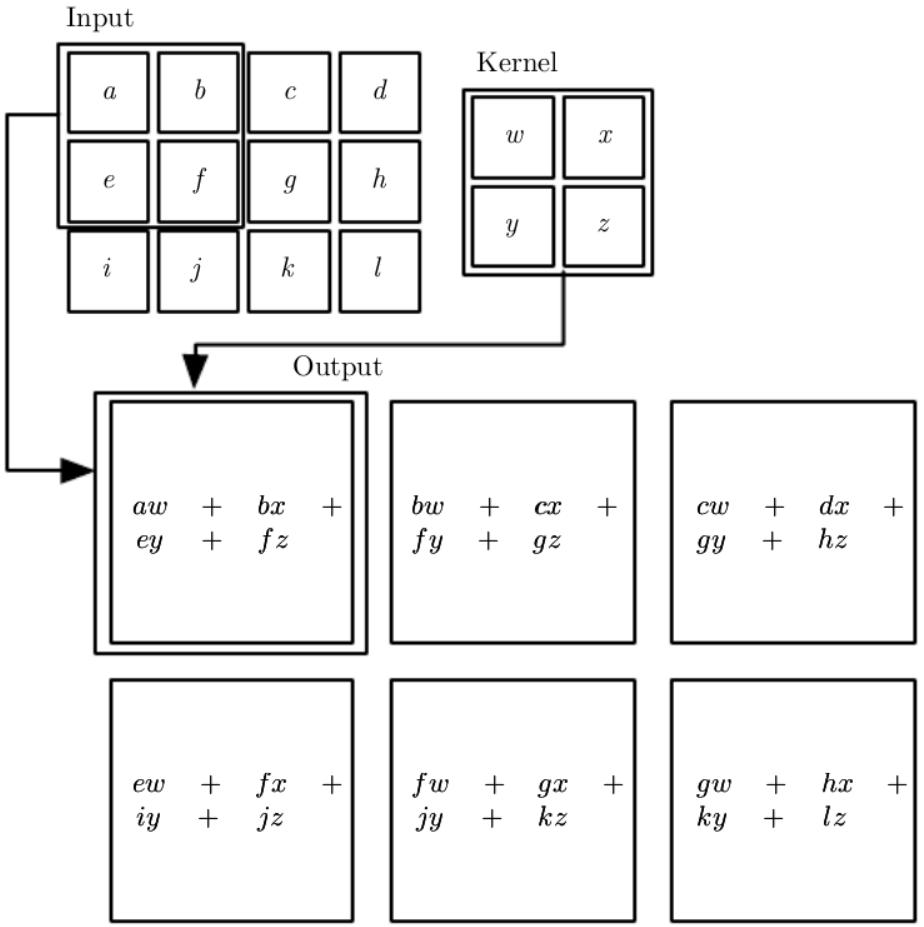
- Same as convolution, but without flipping the kernel

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

- Both referred to as convolution, and whether kernel is flipped or not
- The learning algorithm will learn appropriate values of the kernel in the appropriate place

Example of 2D convolution

- Convolution without kernel flipping applied to a 2D tensor
- Output is restricted to case where kernel is situated entirely within the image
- Arrows show how upper-left of input tensor is used to form upper-left of output tensor



The 2D Case in Matrix Interpretation

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

- Most parameters are shared
- Additionally, the matrix is very sparse

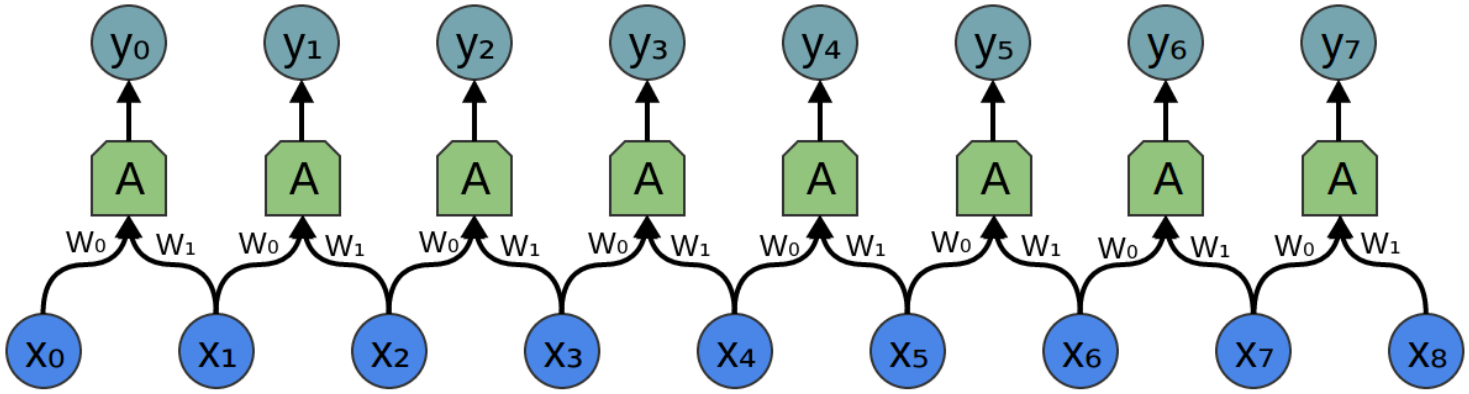
Motivation for using convolution networks

- Convolution leverages three important ideas to improve ML systems:

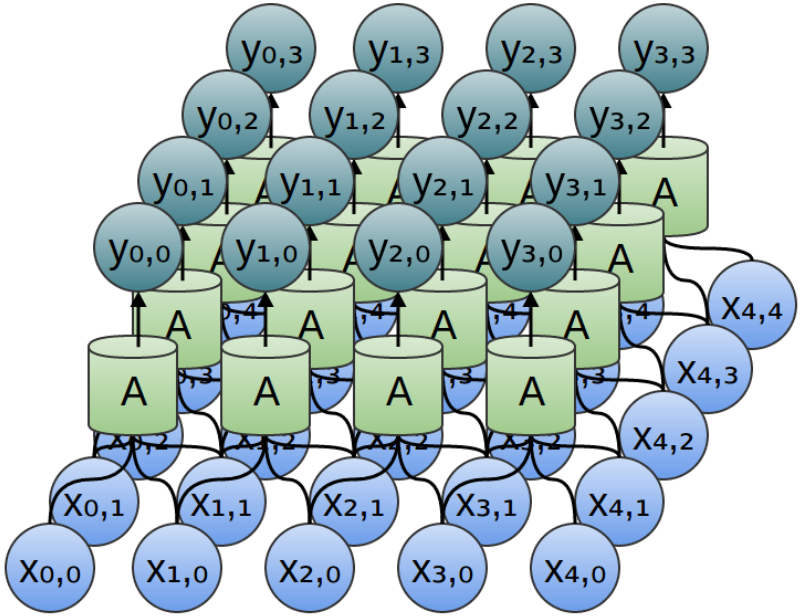
1. Sparse interactions
2. Parameter sharing
3. Equivariant representations

- Convolution also allows for working with inputs of variable size

Sparse connectivity due to Convolution



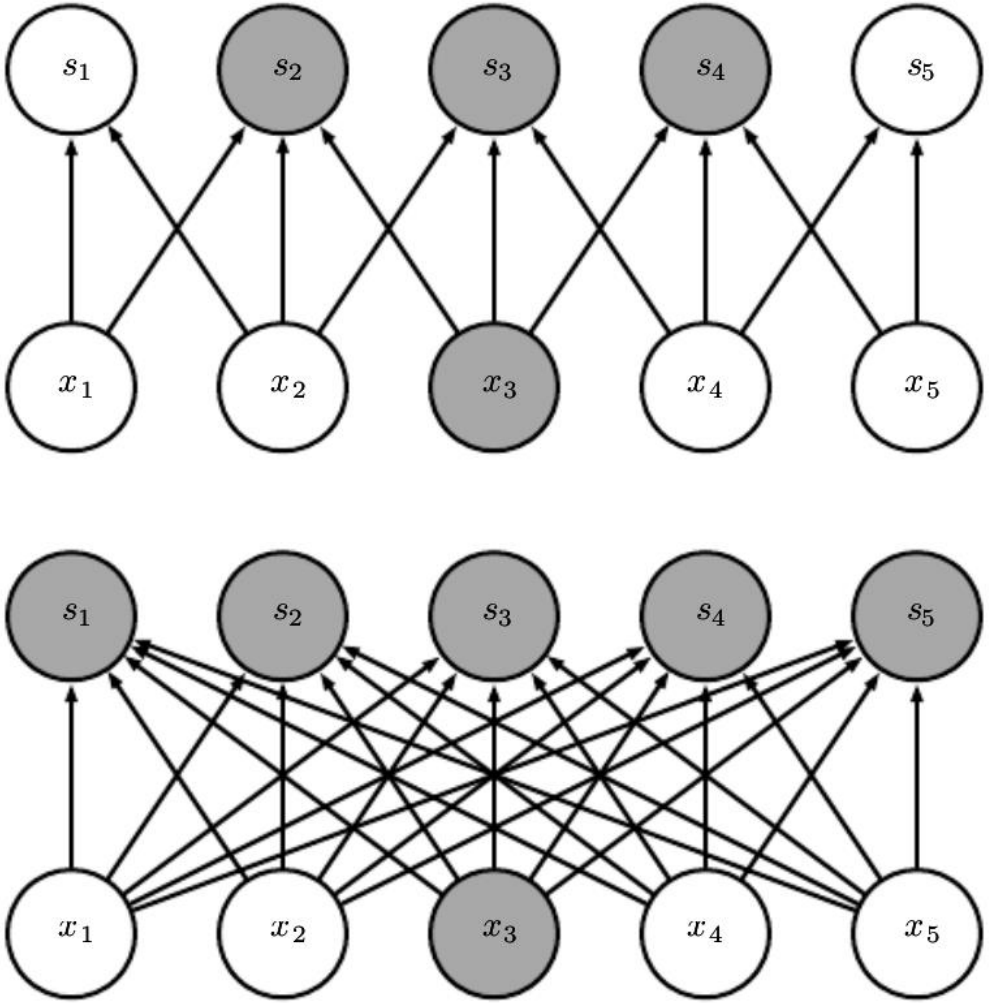
- Instead of learning a full matrix, only a couple of weights of the kernel need to be learned



Sparse Connectivity: Input Perspective

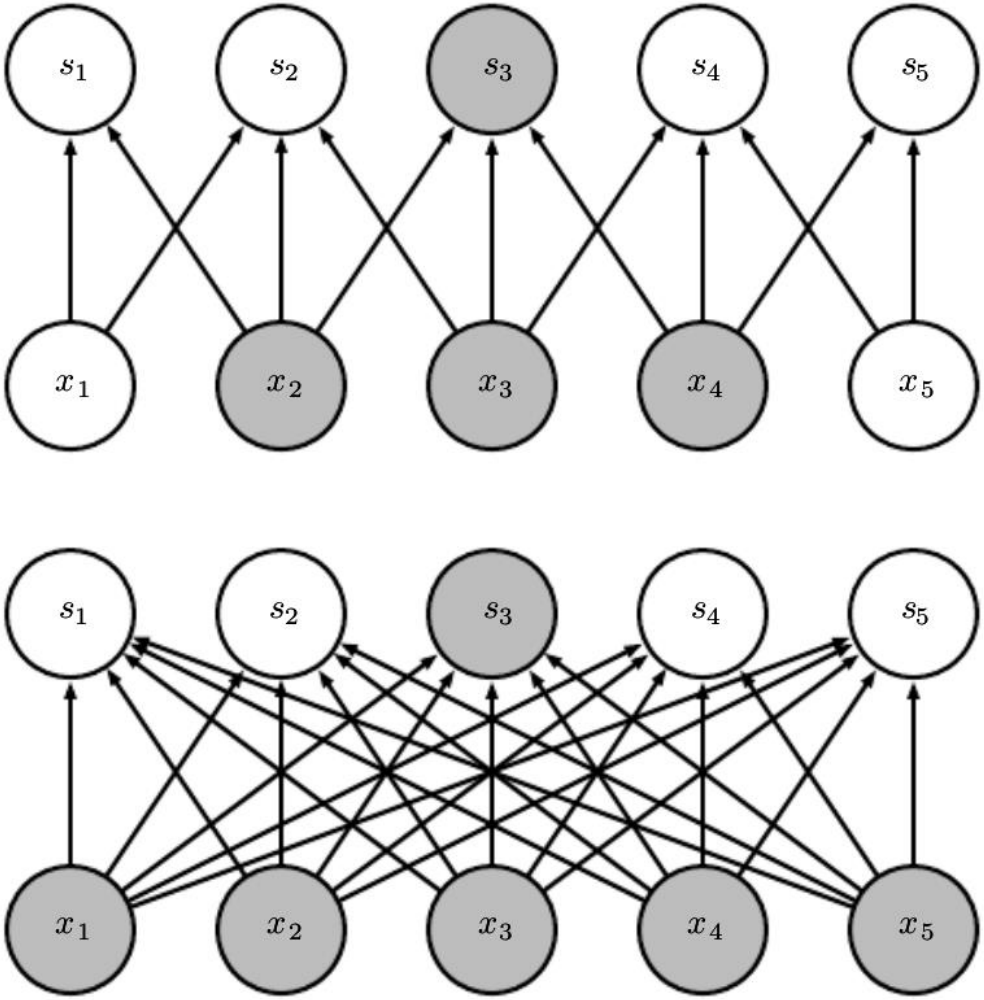
Highlight one input x_3 and output units s affected by it

- Top: when s is formed by convolution with a kernel of width 3, only three outputs are affected by x_3
- Bottom: when s is formed by matrix multiplication connectivity is no longer sparse => All outputs are affected by x_3



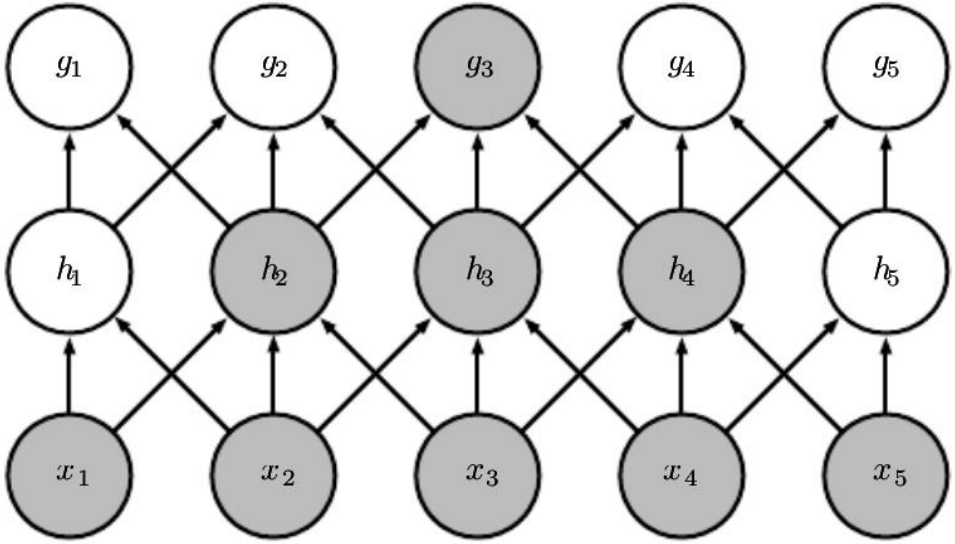
Sparse Connectivity: Output Perspective

Highlight one input s_3 and output units x affected by it



Performance with Reduced Connections

- It is possible to obtain good performance while keeping k several magnitudes lower than m
- In a deep neural network, units in deeper layers may indirectly interact with a larger portion of the input: Depth is the key
- This allows the network to efficiently describe complicated interactions between many variables from simple building blocks that only describe sparse interactions



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 never revisited
- Parameter sharing is synonymous with tied weights
  - Value of the weight applied to one input is tied to a weight applied elsewhere
- In a Convolutional net, each member of the kernel is used in every position of the input (boundaries might be different)

Efficiency of Parameter Sharing

- Parameter sharing by convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set

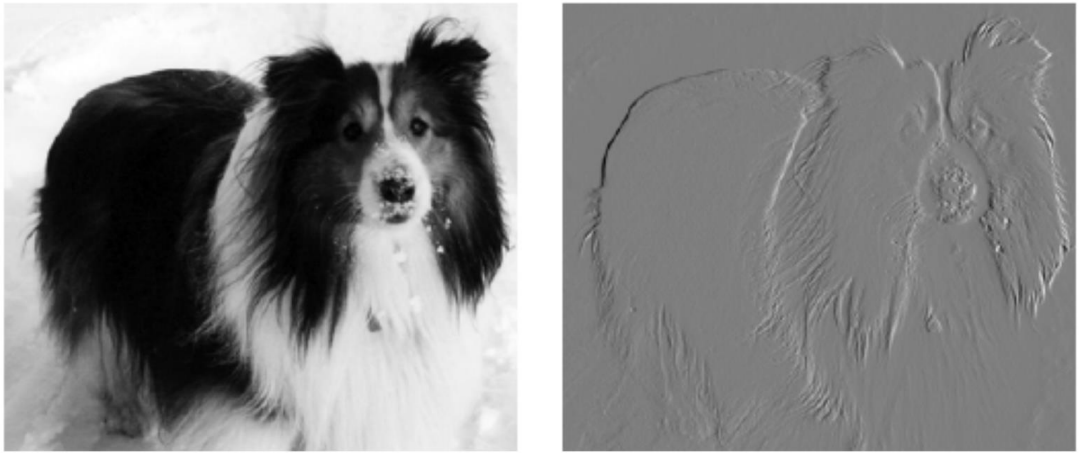
- This does not affect runtime of forward propagation which is still $O(k \cdot n)$
  - Since m and n are roughly the same size, $k \cdot n$ is much smaller than $m \cdot n$

- But further reduces the storage requirements to k parameters
  - k is orders of magnitude less than m

- Convolution is way better in terms of memory and calculation

➢ m are the number of inputs, n the number of outputs, and k the size of the kernel

Example: Efficiency of Convolution for Edge Detection

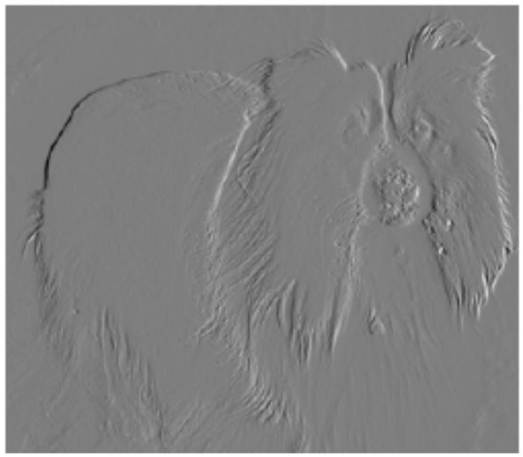


- Image on right formed by taking each pixel of input image and subtracting the value of its neighboring pixel on the left
- Input image is 320x280, output is 319x280
- Computational effort:

$$319 \cdot 320 \cdot 3 = 267{,}960 \text{ flops}$$

  - Each output pixel is calculated by a convolutional kernel with 2 entries, i.e. 2 multiplications, one add

Example: Efficiency of Convolution for Edge Detection

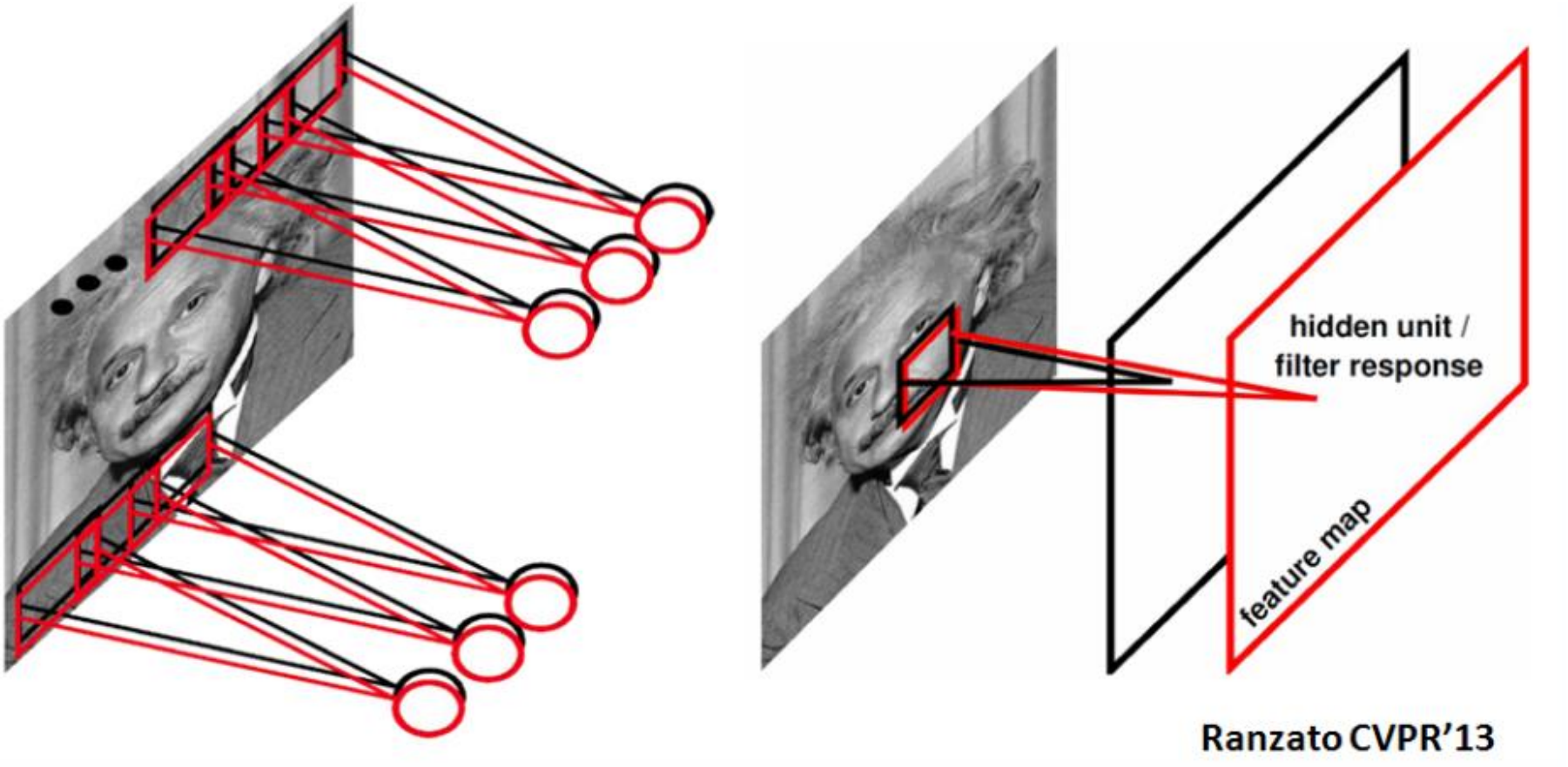


- Same operation with a full matrix with

$$320 \cdot 280 \cdot 319 \cdot 280 = 8 \text{ billion entries}$$

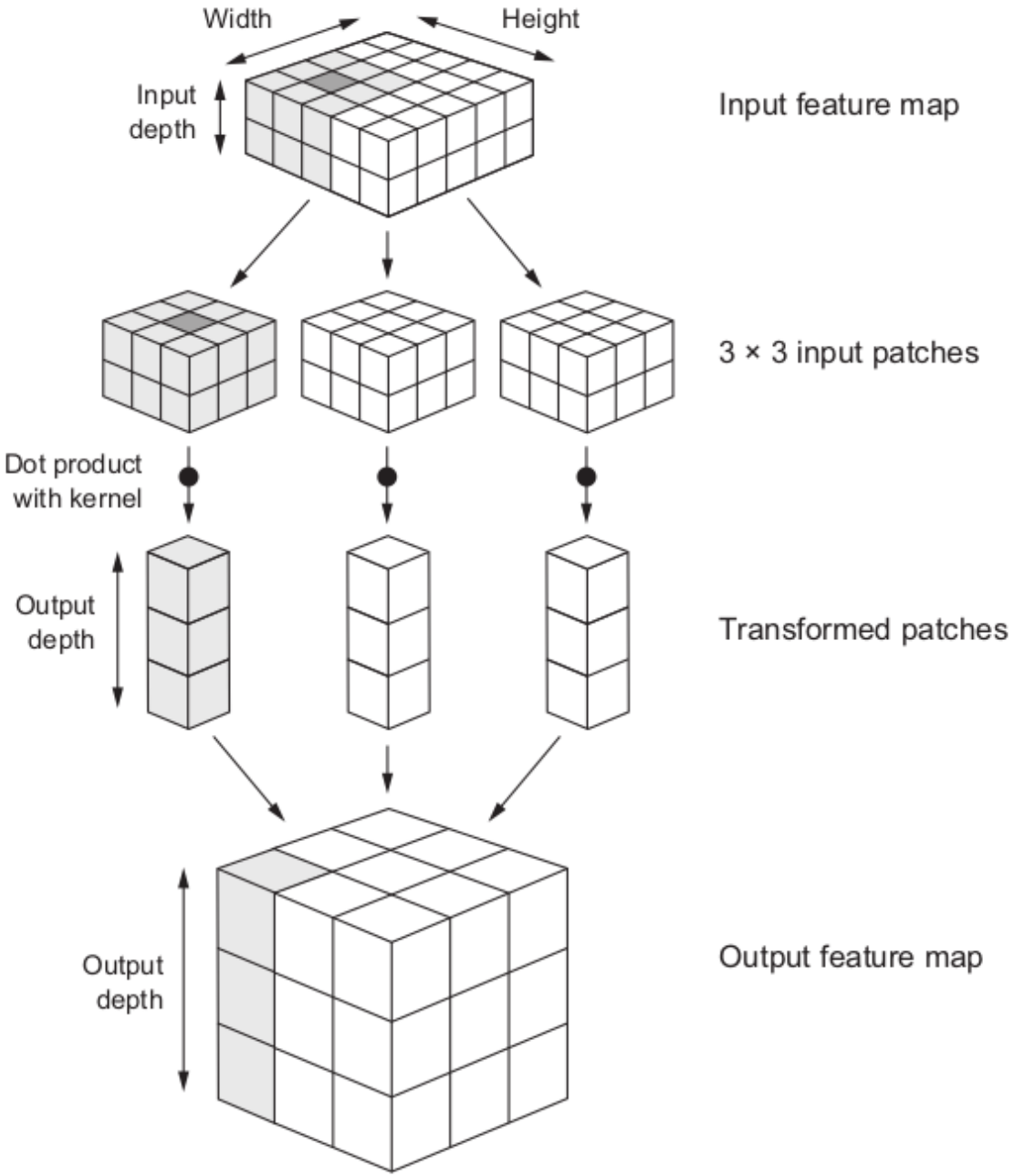
- Storing the matrix (double vals): 8 billion * 8byte ≈ 60 Gbyte
- Performing the calculation: roughly 60,000 times less efficient computationally

Depth of the Output Volume



Depth of the Output Volume

- The depth of the output volume is a hyperparameter
- It corresponds to the number of filters we would like to use, each learning to look for something different in the input.
- For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color.
- We will refer to a set of neurons that are all looking at the same region of the input as a depth column (some people also prefer the term fiber).


Width
Height
Input depth
Input feature map
3 × 3 input patches
Dot product with kernel
Output depth
Transformed patches
Output depth
Output feature map

Equivariance of Convolution to Translation

- The particular form of parameter sharing leads to equivariance to translation
  - A function $f(x)$ is equivariant to a function g if $f\big(g(x)\big) = g\left(f(x)\right)$
  - Equivariant means that if the input changes, the output changes in the same way

- If g is a function that translates the input, i.e., that shifts it, then the convolution function is equivariant to g
  - $I(x, y)$ is image brightness at point (x, y)
  - $I' = g(I)$ is image function with $I'(x, y) = I(x - 1, y)$
  - If we apply g to I and then apply convolution, the output will be the same as if we applied convolution to I, then applied transformation g to the output

Example of equivariance

- With 2D images convolution creates a map where certain features appear in the input
- If we move the object in the input, the representation will move the same amount in the output
- Example:
  - It is useful to detect edges in first layer of convolutional network
  - Edges are roughly the same every where in the image
  - So it is practical to share parameters across entire image

Absence of equivariance

- In some cases, we may not wish to share parameters across entire image
  - If image is cropped to be centered on a face, we may want different features from different parts of the face
  - Part of the network processing the top of the face looks for eyebrows
  - Part of the network processing the bottom of the face looks for the chin
- Certain image operations such as scale and rotation are not equivariant to convolution
  - Other mechanisms are needed for such transformations

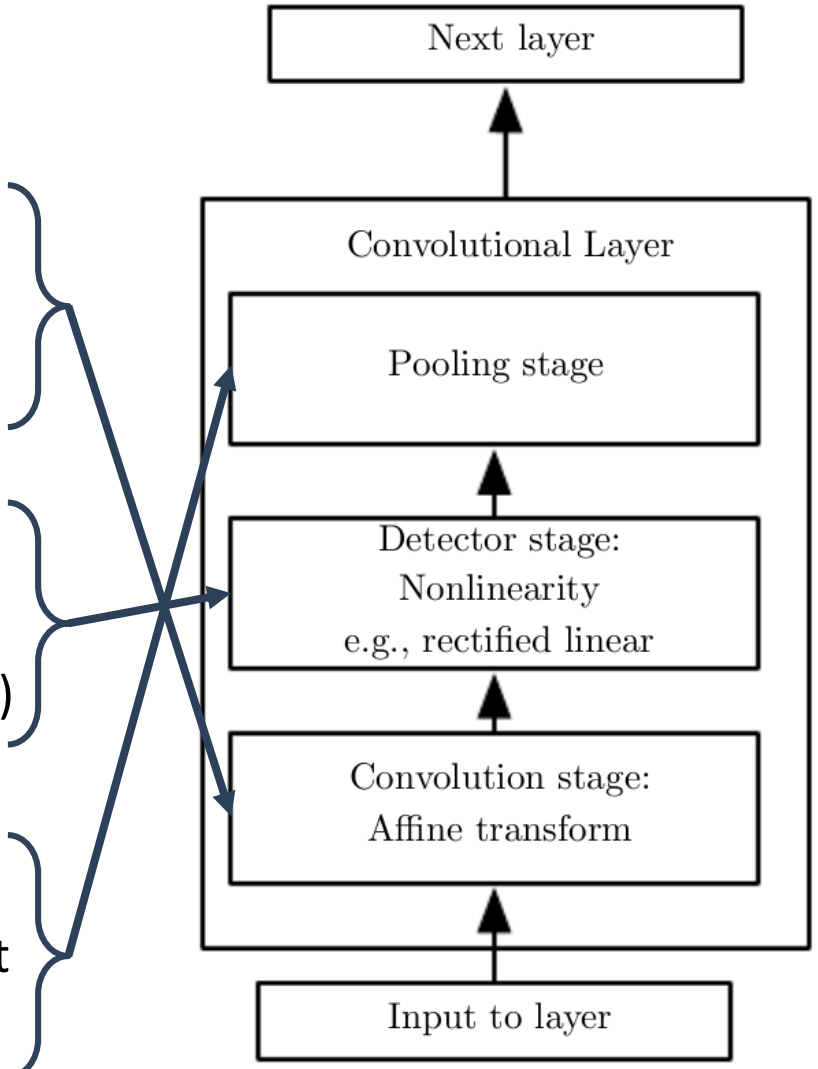
Convolutional Neural Networks

- Convolution
- **Pooling**
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

Pooling

Typical layer of a CNN consists of three stages:

- Stage 1 (**Convolution**):
  - Perform several convolutions in parallel to produce a set of linear activations
- Stage 2 (**Detector**):
  - Each linear activation is run through a nonlinear activation function (e.g. ReLU)
- Stage 3 (**Pooling**):
  - Use a pooling function to modify output of the layer further



Types of Pooling functions

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby inputs

Popular pooling functions:

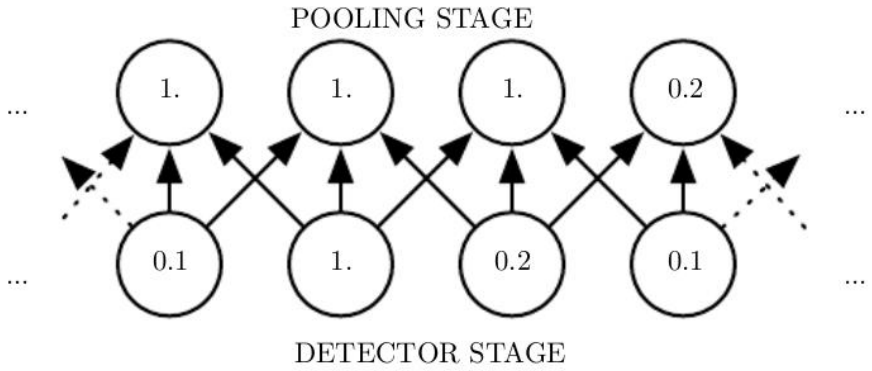
1. Max pooling operation reports the maximum output within a rectangular neighborhood
2. Average of a rectangular neighborhood
3. L^2 norm of a rectangular neighborhood
4. Weighted average based on the distance from the central pixel

Why Pooling?

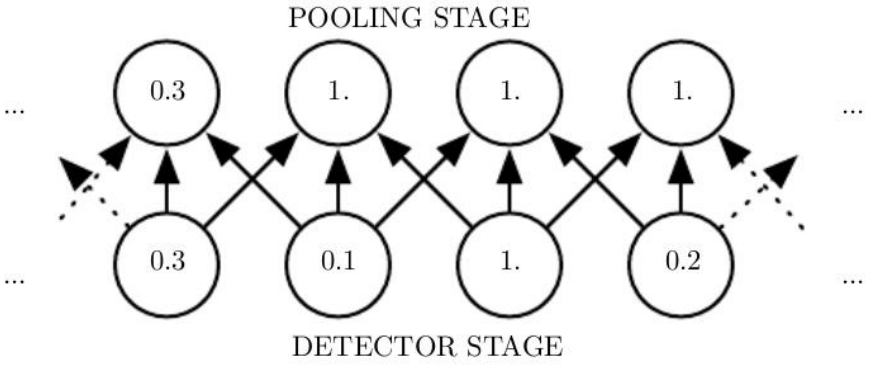
- In all cases, pooling helps make the representation become approximately invariant to small translations of the input
- If we translate the input by a small amount values of most of the outputs does not change
- **Pooling can be viewed as adding a strong prior that the function the layer learns must be invariant to small translations**

Max pooling introduces invariance to translation

- View of middle of output of a convolutional layer



- Same network after the input has been shifted by one pixel



- Every input value has changed, but only half the values of output have changed

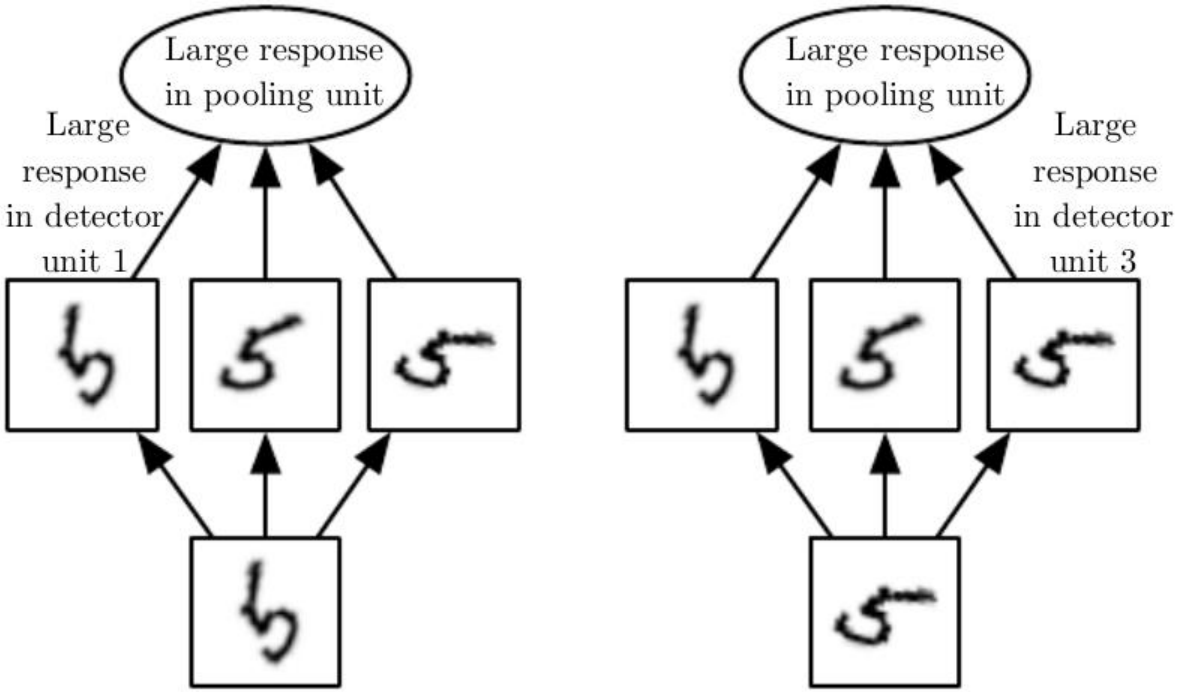
Importance of Translation Invariance

- Invariance to translation is important if we care about whether a feature is present rather than exactly where it is
  - For detecting a face we just need to know that an eye is present in a region, not its exact location

- In other contexts it is more important to preserve location of a feature
  - E.g., to determine a corner we need to know whether two edges are present and test whether they meet

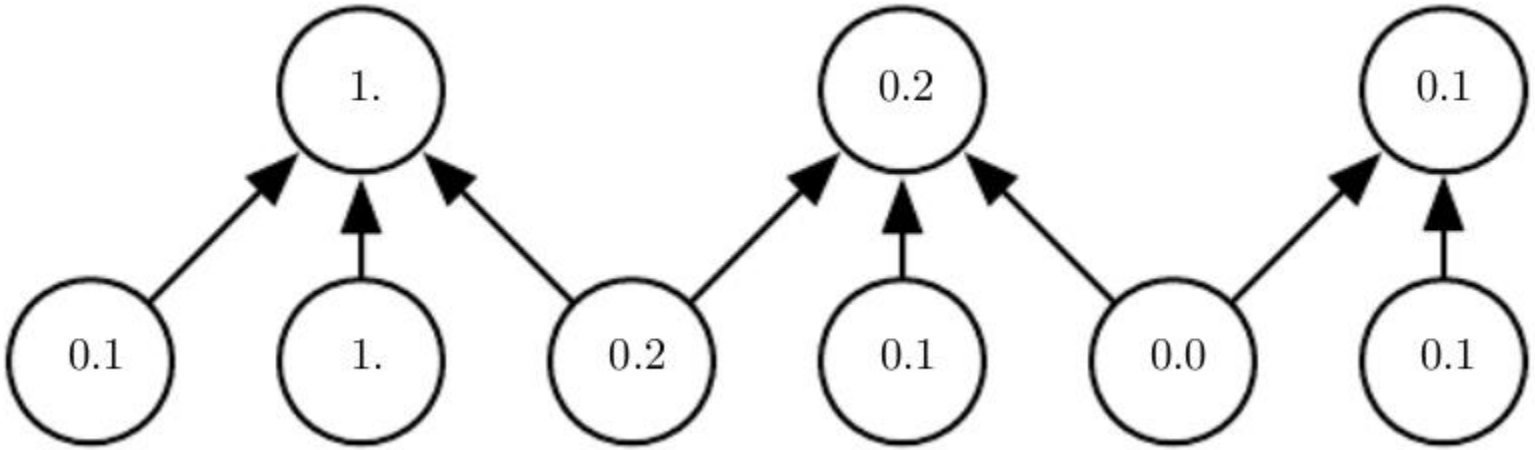
Learning other invariances

- Pooling over spatial regions produces invariance to translation
- But if we pool over the results of separately parameterized convolutions, the features can learn which transformations to become invariant to

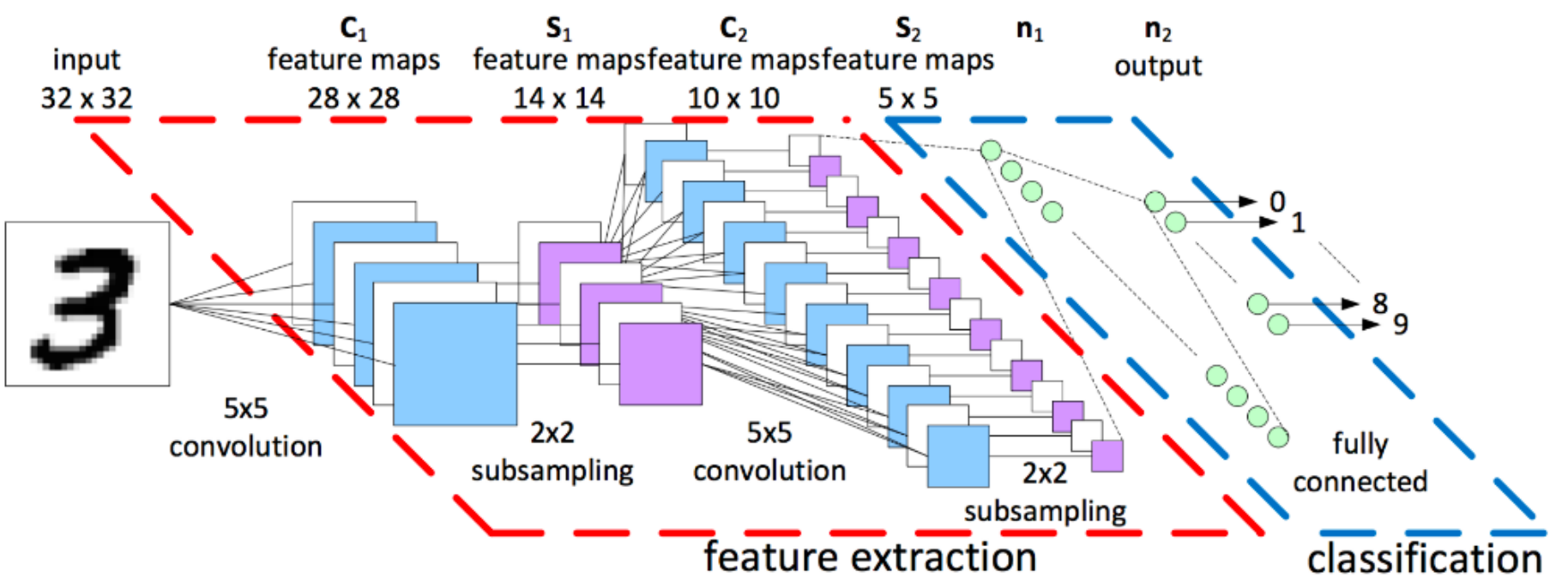


Pooling with Down-Sampling

- 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 one pixel apart
  - Next layer has k times fewer inputs to process

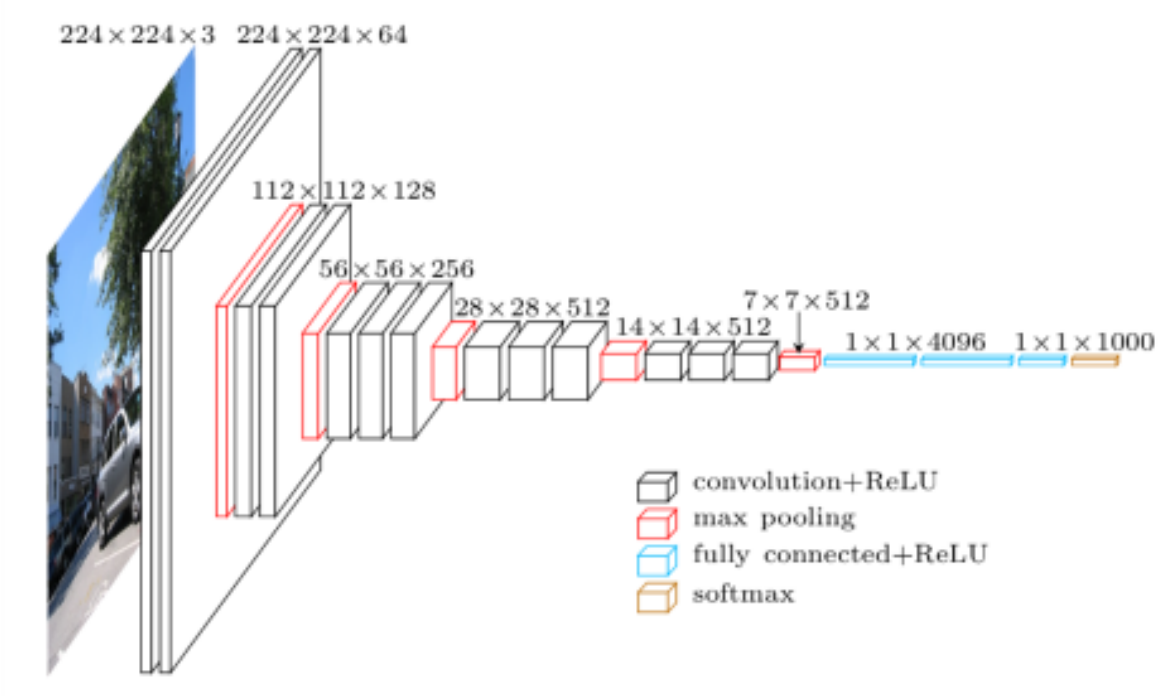


Example for a Convolutional Network



VGG Net

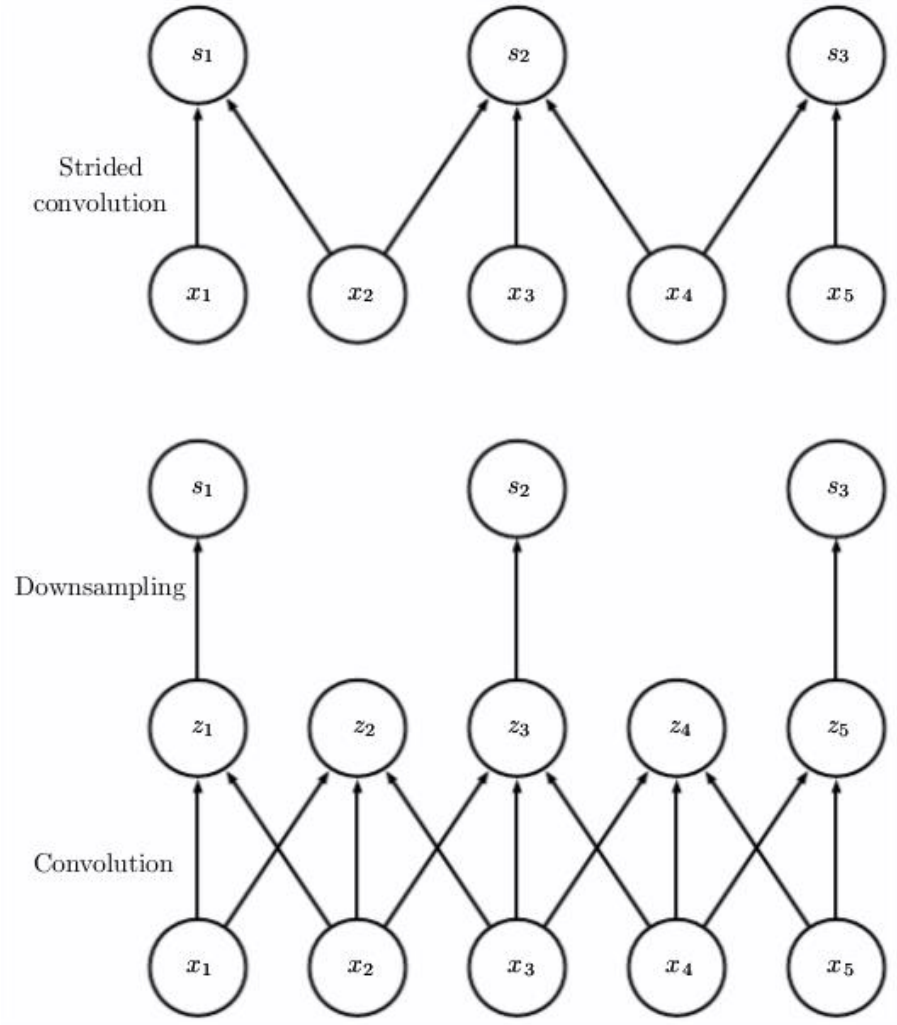
- VGG is a convolutional neural network model
  - "Very Deep Convolutional Networks for Large-Scale Image Recognition"
- The model achieves 93% top-5 test accuracy in ImageNet which is a dataset of over 14 million images belonging to 1000 classes.



Convolutional Neural Networks

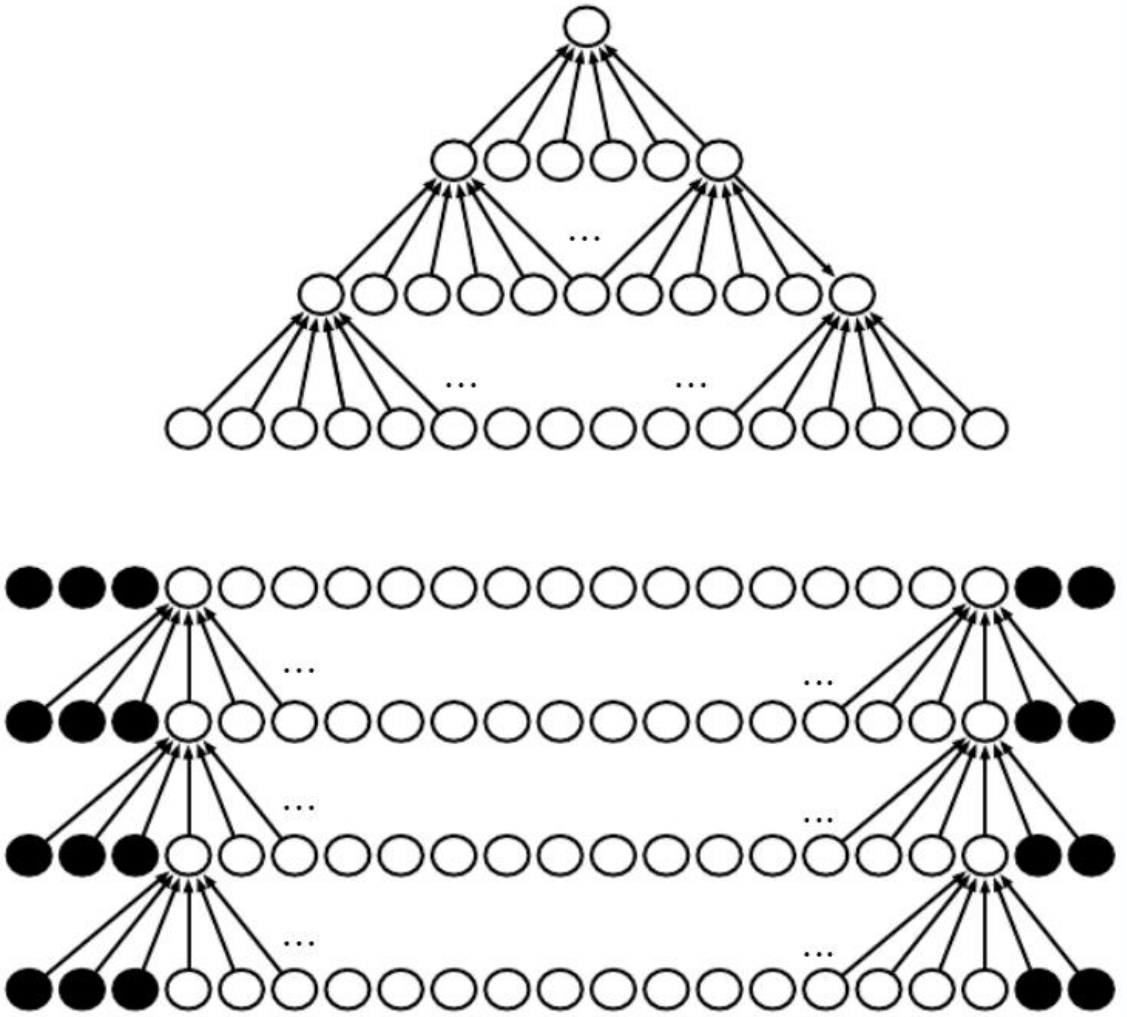
- Convolution
- Pooling
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

Convolution with a stride: Implementation



- Strided convolution is equivalent to normal convolution followed by a downsampling
- Second approach is computationally wasteful

Effect of Zero-padding on network size



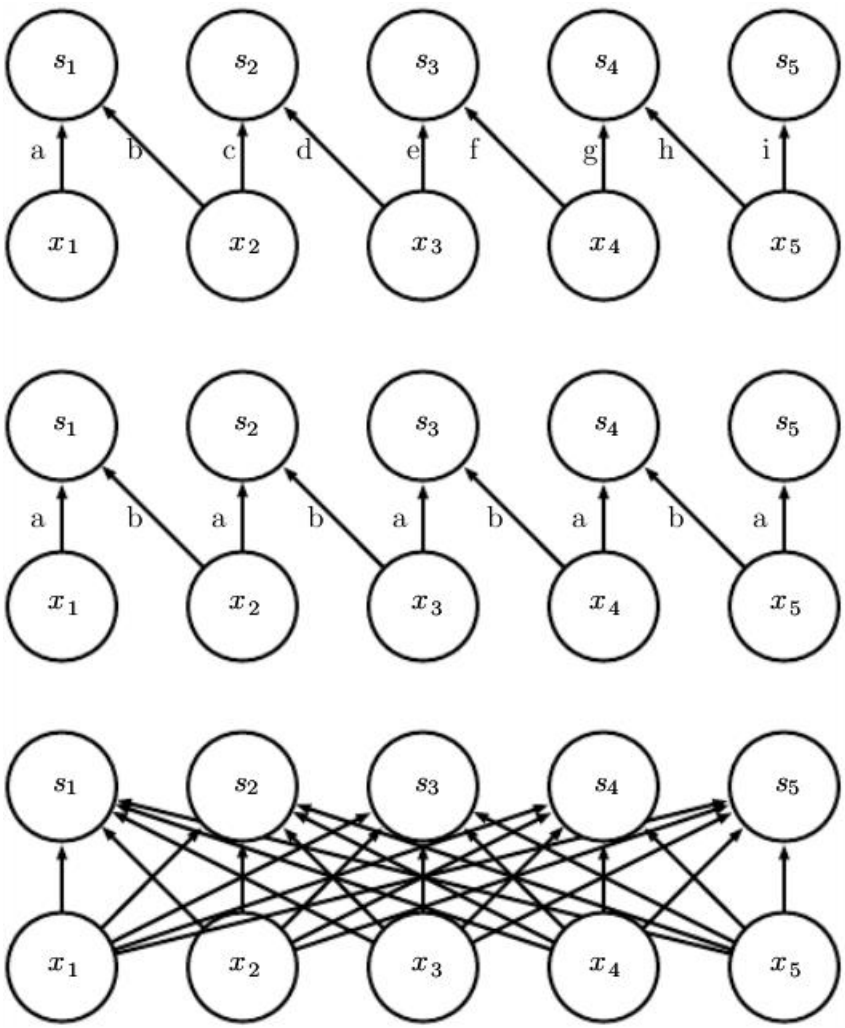
Locally connected layer

- In some cases, we do not actually want to use convolution, but rather sparsely connected layers
  - adjacency matrix in the graph of the network is the same, but every connection has its own weight, specified by a 6-D tensor $\mathbf{W}$:
    - i, the output channel
    - j, the output row,
    - k, the output column,
    - l, the input channel,
    - m, the row offset within the input, and
    - n, the column offset within the input.

$$Z_{i,j,k} = \sum_{l,m,n} \left[V_{l,j+m-1,k+n-1} W_{i,j,k,l,m,n} \right]$$

- Also called unshared convolution

Local Connections, Convolution, Full connections



Use of Locally Connected Layers

- Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space

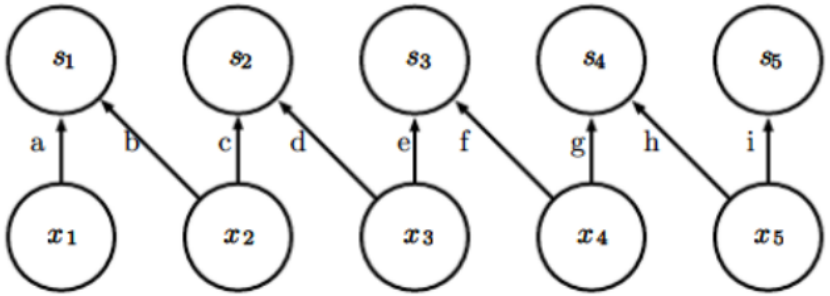
- Ex: if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image

Tiled Convolution

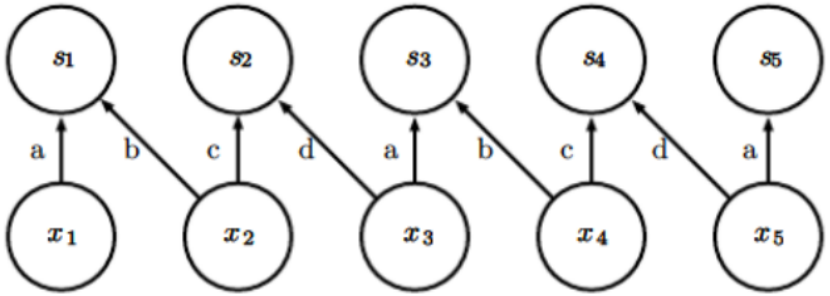
- Compromise between a convolutional layer and a locally connected layer.
  - Rather than learning a separate set of weights at every spatial location, we learn a set of kernels that we rotate through as we move through space.

- This means that immediately neighboring locations will have different filters, like in a locally connected layer,
  - but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels
  - rather than the size of the entire output feature map.

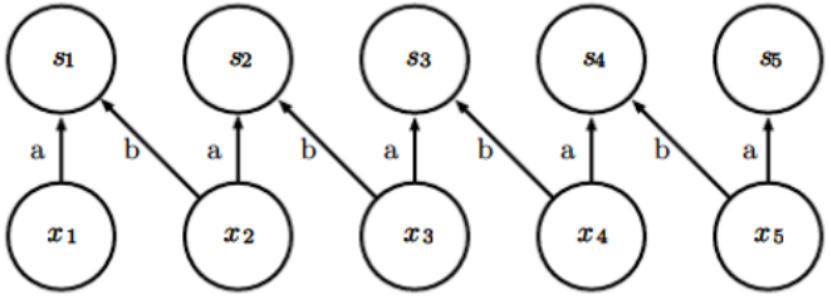
Locally Connected vs. Tiled Convolution vs. Convolution



A locally connected layer
Has no sharing at all
Each connection has its own weight



Tiled convolution
Has a set of different kernels
With $t=2$



Traditional convolution
Equivalent to tiled convolution
with $t=1$
There is only one kernel and it is applied everywhere

Convolutional Neural Networks

- Convolution
- Pooling
- Variants of Convolution
- **Efficient Convolution**
- Convolution in Keras

Speed-Up Convolution

- Typically, the most expensive part of convolutional network training is learning the features
- The output layer is usually relatively inexpensive (small number of features after passing through pooling)
- When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network.
- One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

Obtaining Kernels without Training

There are three basic strategies for obtaining convolution kernels without supervised training.

1. Simply initialize them randomly
2. Design them by hand, for example by setting each kernel to detect edges at a certain orientation or scale
3. Learn the kernels with an unsupervised criterion

Random Initialization

- Random filters often work surprisingly well in convolutional networks

- It was shown that layers consisting of convolution following by pooling naturally become frequency selective and translation invariant when assigned random weights

- Strategy:
  - first evaluate the performance of several convolutional network architectures by training only the last layer
  - take the best of these architectures and train the entire architecture using a more expensive approach.

Blurring Filter

$$K_{box} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad \hat{K}_{box} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

the same image without any filtering, after a 3x3 box blur and after a 9x9 box blur

- https://www.taylorpetrick.com/blog/post/convolution-part3

Edge Detection

- Simple

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

- Sobel

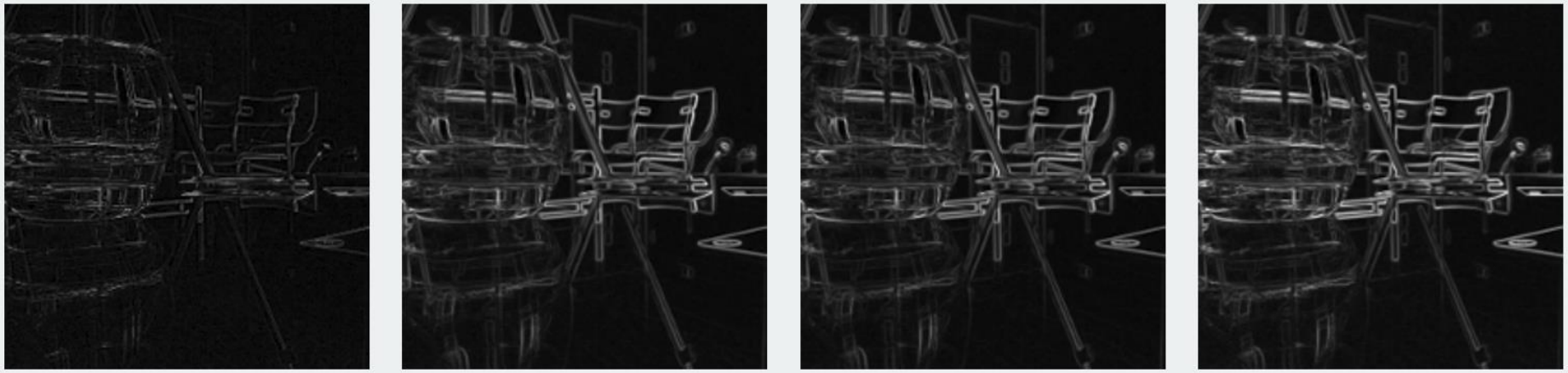
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

- Prewitt

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * \mathbf{A}$$

- Kirsch (8 directions)

$$\mathbf{g}^{(1)} = \begin{bmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(2)} = \begin{bmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}, \mathbf{g}^{(3)} = \begin{bmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{bmatrix}$$



comparison of simple, sobel, prewitt and kirsch edge detection filters

- https://www.taylorpetrick.com/blog/post/convolution-part3

Sharpen

$$K_{sharp} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} * amount$$



comparison of unfiltered image and sharpened images with amount=2 and amount=8

Learn Kernels Unsupervised

- For example, k-means clustering to small image patches
- Use each learned centroid as a convolution kernel
- Learning the features with an unsupervised criterion allows them to be determined separately from the classifier layer at the top of the architecture.
- One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer

Convolutional Neural Networks

- Convolution
- Pooling
- Variants of Convolution
- Efficient Convolution
- Convolution in Keras

Defining Convolutions in Keras

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

The Convolutional Layer in Keras

```
keras.layers.Conv2D(filters, kernel_size, #Number of filters, kernel
                                            dimensions
    strides=(1, 1),  # Using a stride?
    padding='valid’, # ‘valid’ means only valid positions are used.
                ‘same’ means output has same dimension as input
    data_format=None, # ‘channels_last’ or ‘channels_first’
    dilation_rate=(1, 1), # Increase the field of “vision”
    activation=None, # Rest of the parameters same as dense layer
    use_bias=True,
    kernel_initializer='glorot_uniform’,
    bias_initializer='zeros’,
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None
)
```

The Pooling Layer in Keras

```
keras.layers.MaxPooling2D(
    pool_size=(2, 2), # Size of the pooling
    strides=None, # Integer, tuple of 2 integers, or None. Strides
                  values. If None, it will default to pool_size.
    padding='valid', # One of "valid" or "same"
    data_format=None # Same as with Conv2D
)
```

- There are other pooling layers available
- Function principle remains the same

What if the dataset is too large?

- Especially when working with images, the datasets can become easily too large
- We cannot hold them all in memory
- We also want to distort images to create new examples

ImageDataGenerator

- Keras provides a helper class doing most of the leg work for us
- It provides functionality to automatically read images from a directory in batches
- Can additionally perform modifications to the images
- Instead of a dataset, the model now receives a generator as input.

Example

```
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_dir, # Where are our images stored
    target_size=(150, 150) # Resizes all images to 150 × 150
    batch_size=20,
    class_mode='binary’)

validation_generator = test_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=20,
    class_mode='binary’)
```

- The directories should contain one subdirectory per class
- Returns a directory iterator
- Never ends, keeps looping through the directory

Training your Model

```
history = model.fit_generator(
    train_generator,
    steps_per_epoch=100,
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50
)
```

- Have to use the fit_generator method
- Have to specify after how many steps the epoch ends

Automatically Distort Images

```
datagen = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest’
)
```

- This can be done by feeding additional information to the data generator

Example

```
from keras.preprocessing import image

img = image.load_img(img_path, target_size=(150, 150)) # Read and
                                                        resize
x = image.img_to_array(img) # numpy array (150,150,3)
x = x.reshape((1,) + x.shape) # Reshapes it to (1, 150, 150, 3)

i = 0
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break

plt.show()
```

Example