Convolutional models

Josip Krapac and Siniša Šegvić

Motivation: distinguish bison from oxen

A deep model has a chance to learn features that respond to parts, e.g.: [hump?, small horns?, wilderness?, ...]

- bison: [YES, YES, YES, ...], cattle: [NO, NO, NO, ...]

Fully connected models are at risk of learning noise because:

- the translated image is completely different from the original
- key features determined by local neighborhoods
- the model needs to learn each translation separately

Overview

- What are convolutional models?
- What is convolution?
- Why convolution?
- Pooling and padding

What are convolutional models?

Models specialized for compound data with lattice topology

- topology: the structure of the neighborhood relation

Typical examples:

- time sequences (1 axis), images (2 axes), volumes (3 axes)

A simple definition: a convolutional model has at least one convolutional layer

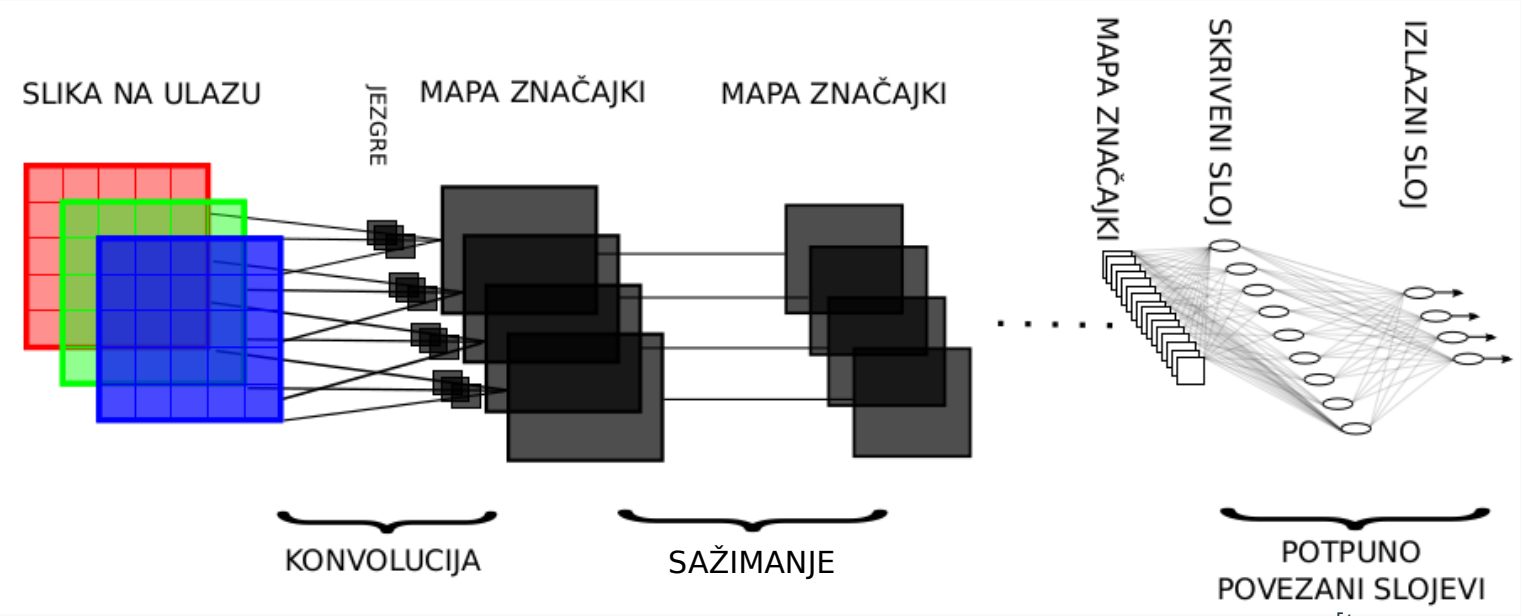
- convolutional layers are often accompanied by pooling layers, non-linear activations (ReLU) and fully connected layers

Let's look at the meaning of *convoluted* (nomen est omen):

- extremely complex and difficult to follow
- intricately folded, twisted, or coiled

What are convolutional models?

Classic convolutional architecture (LeNet-5):



[lecun89nc,vukotic14ms]

- convolutional layers transform tensors of the third order:
  - two spatial, one "semantic" axis
  - we will first assume that we have only one semantic dimension
- all mapings are local: output activations ("pixels") depend on the local neighborhood of the input tensor

What is convolution?

We define convolution as a scalar product of a function w with a shifted and reflected function x:

$$h(t) = (w * x)(t) = \int_{\mathcal{D}(w)} w(\tau)x(t-\tau)d\tau$$

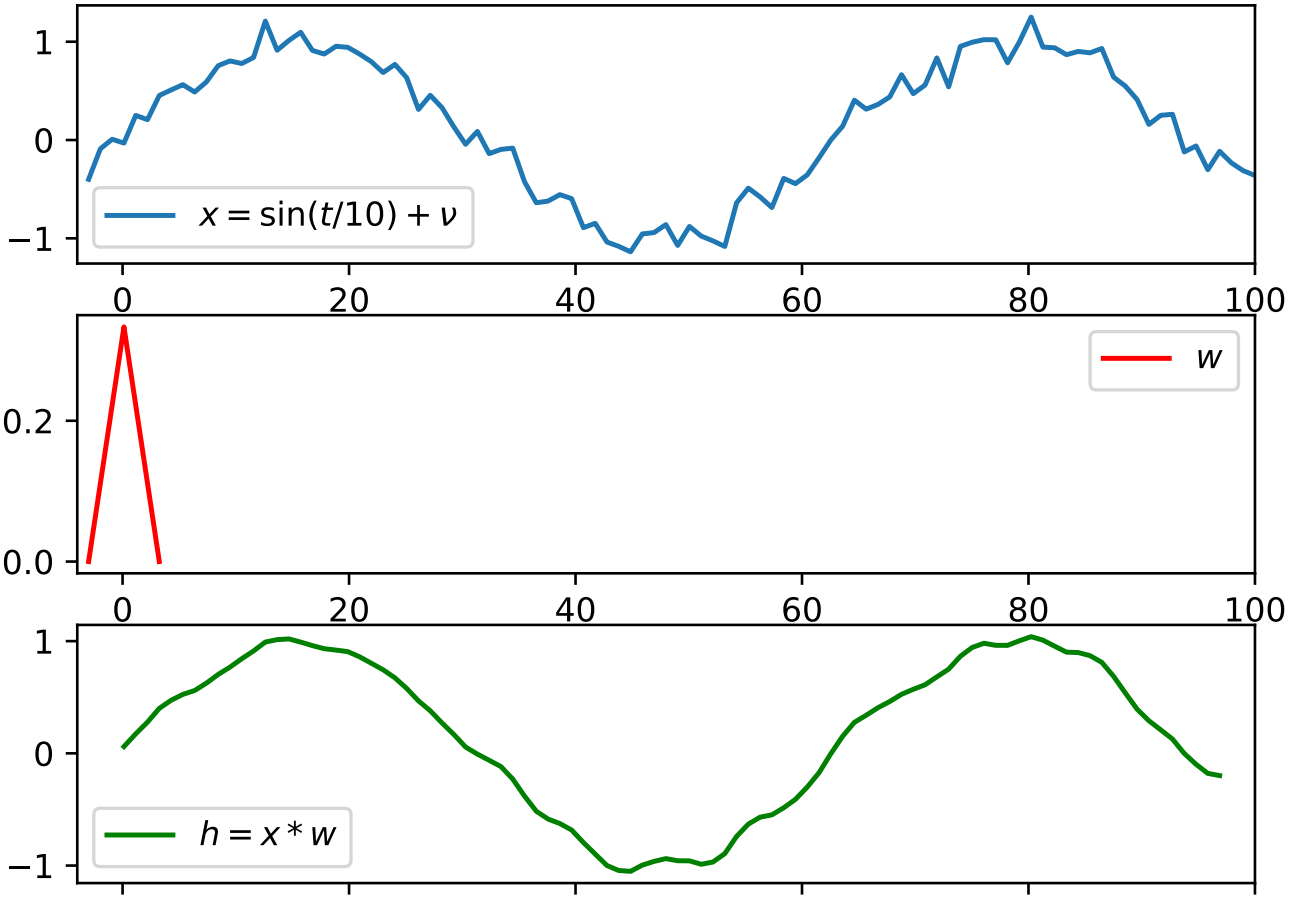
In machine learning, convolutions are often implemented as cross-correlation:

$$h(t) = (w \star x)(t) = \int_{\mathcal{D}(w)} w(\tau)x(t+\tau)d\tau$$

Convolutions and cross-correlations are interesting as differentiable operations with free parameters

The equations suggest that kernel w extracts local translation-equivariant features from the input signal x

What is convolution?



$$h(t) = (w \star x)(t) = \int_{\mathcal{D}(w)} w(\tau)x(t+\tau)d\tau$$

What is convolution?

We have smoothed the input signal $x(t)$ by cross-correlating it with a suitable function $w(t)$:

$$h(t) = w(t) \star x(t)$$

In the context of a "convolutional" layer:

- function x (argument) is input,
- function w (argument) is kernel (free parameters),
- function h (result) is called a feature map.

What is convolution?

In the discrete case, we replace the integral with the sum:

$$h(t) = (w \star x)(t) = \sum_{\tau=-\infty}^{\infty} w(\tau)x(t+\tau)$$

We assume that the kernel domain is finite, i.e. that the function $W(\tau)$ equals zero for $\tau < \tau_{\text{min}}$ and $\tau > \tau_{\text{max}}$:

$$h(t) = (w \star x)(t) = \sum_{\tau=\tau_{\text{min}}}^{\tau_{\text{max}}} w(\tau)x(t+\tau)$$

In most applications, x and w are multivariate functions: $\boldsymbol{x}(t), \boldsymbol{w}(t) \in \mathbb{R}^d$.

What is convolution?

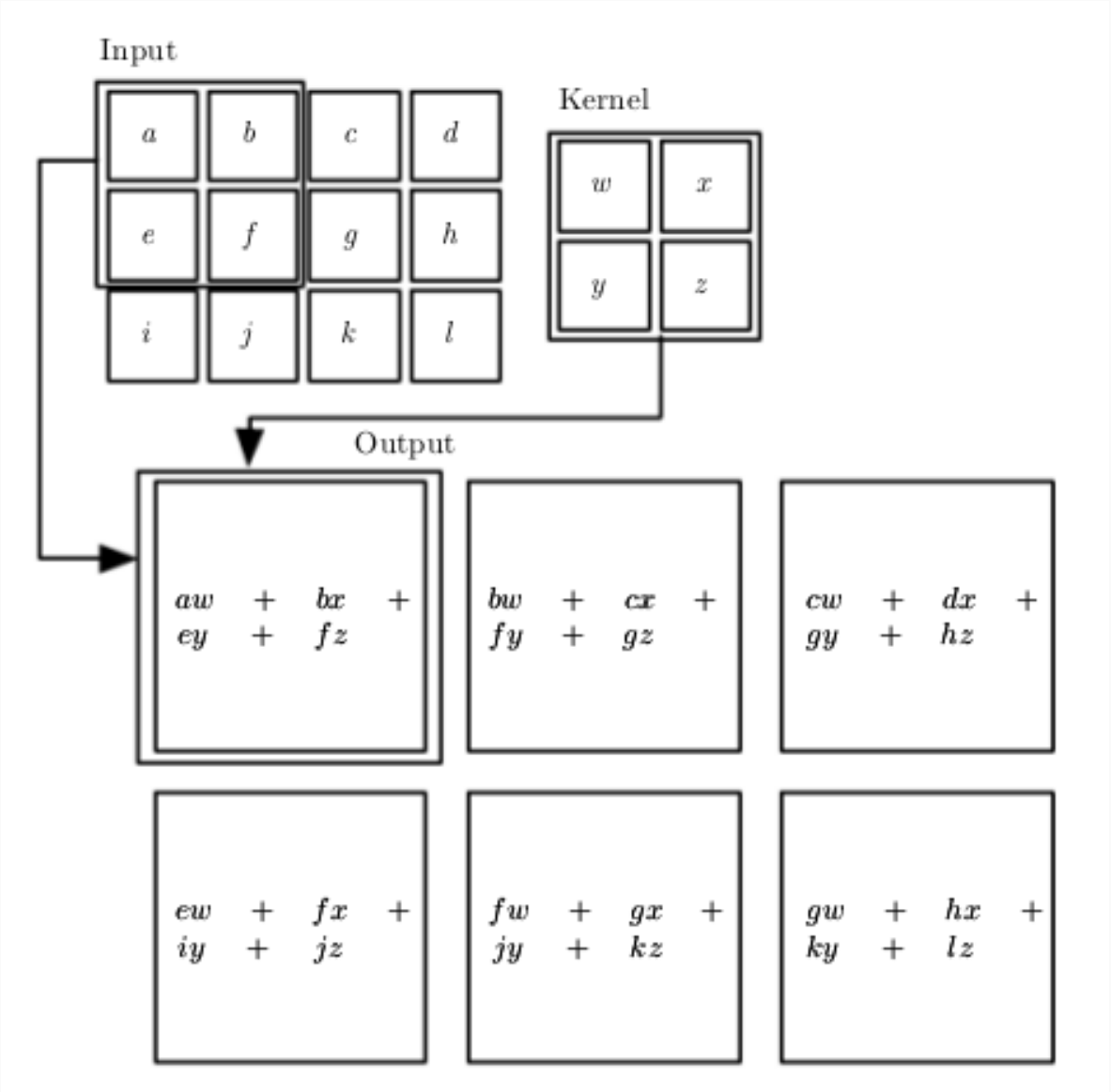
Correlation can be applied across multiple axes. In case of images, we typicly apply 2D convolution:

$$S(i,j) = (K \star I)(i,j) = \sum_{m=m_{\min}}^{m_{\max}} \sum_{n=n_{\min}}^{n_{\max}} K(m,n) \cdot I(i+m, j+n)$$

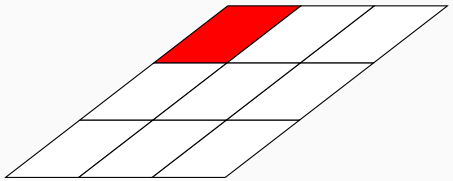
.

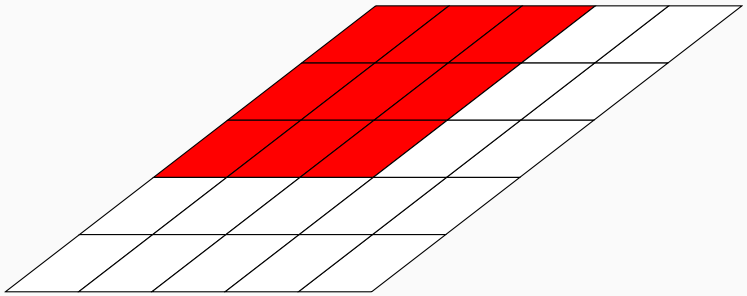
- the summation ranges are defined by the kernel domain (i.e. where $K \neq 0$).
- kernels are usually smaller than images (3x3 - 7x7 vs MPx)

What is convolution?

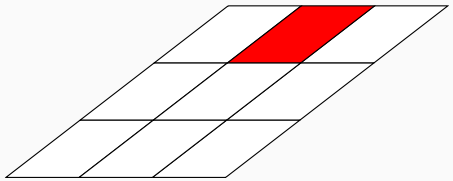


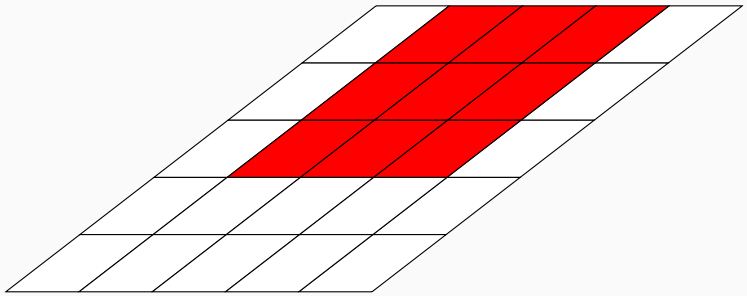
What is convolution?



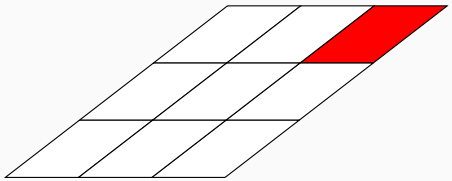


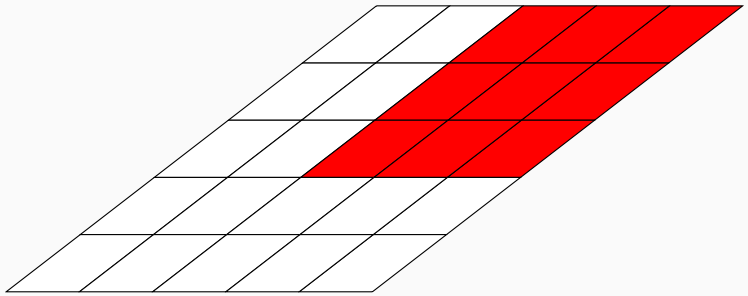
What is convolution?



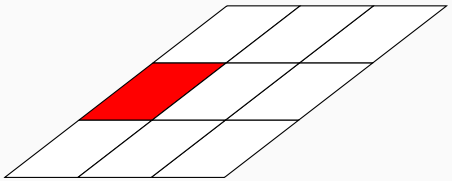


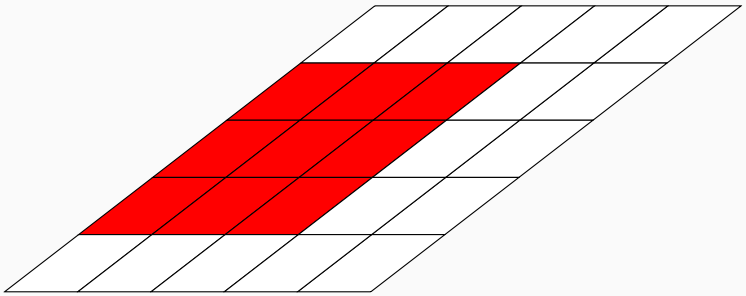
What is convolution?



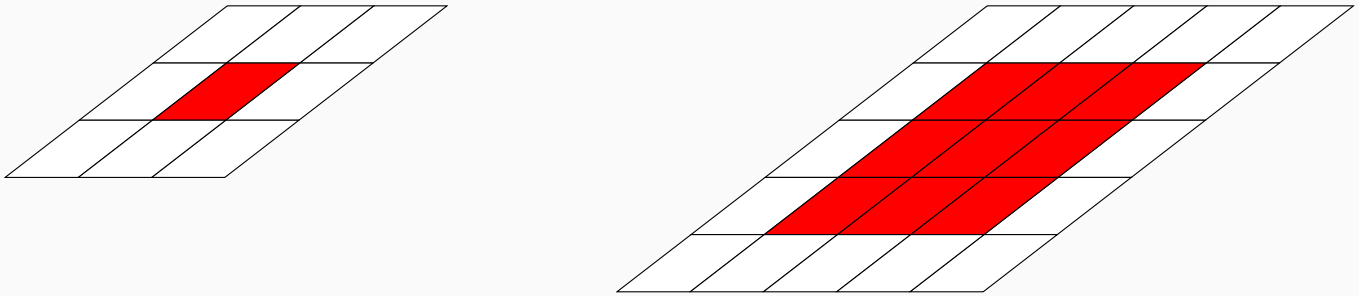


What is convolution?

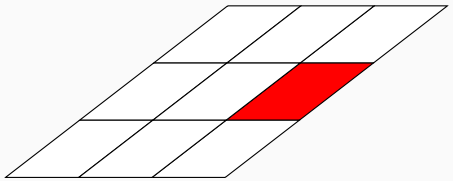


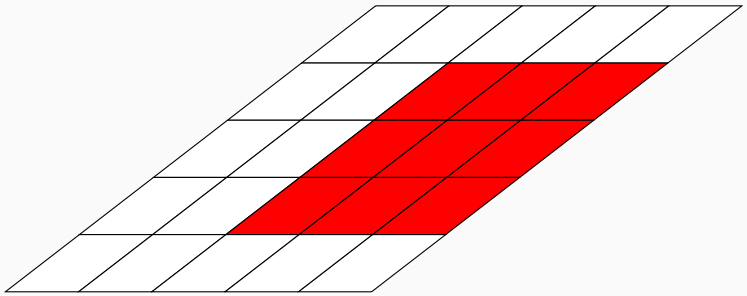


What is convolution?

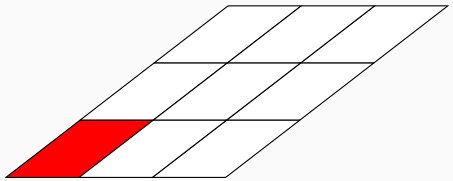


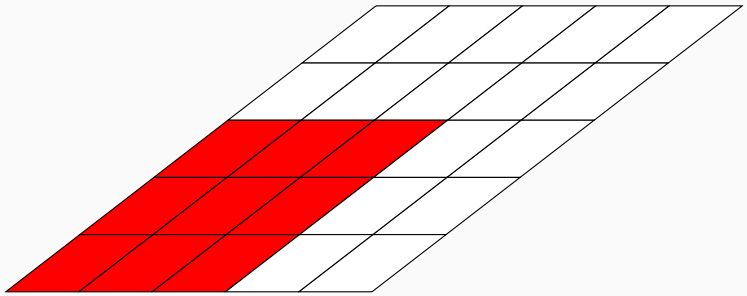
What is convolution?



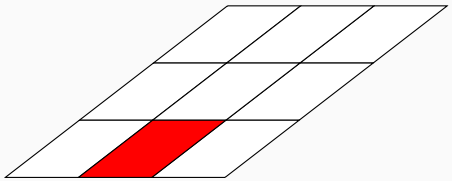


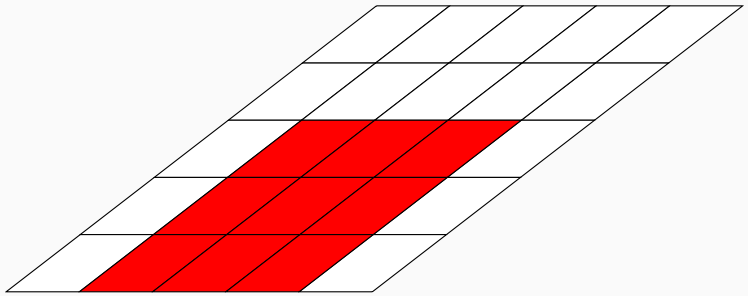
What is convolution?



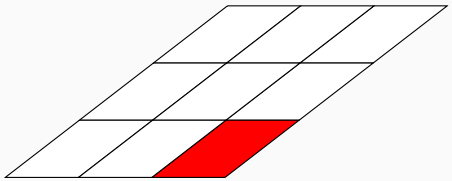


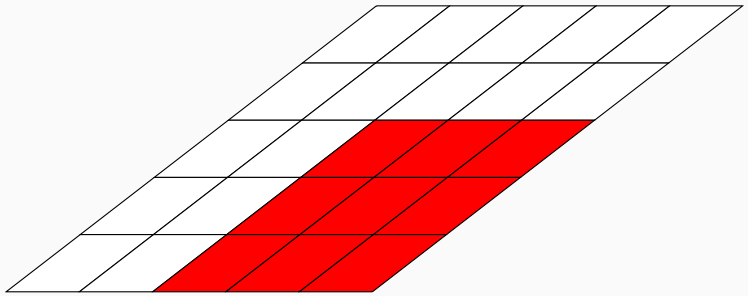
What is convolution?





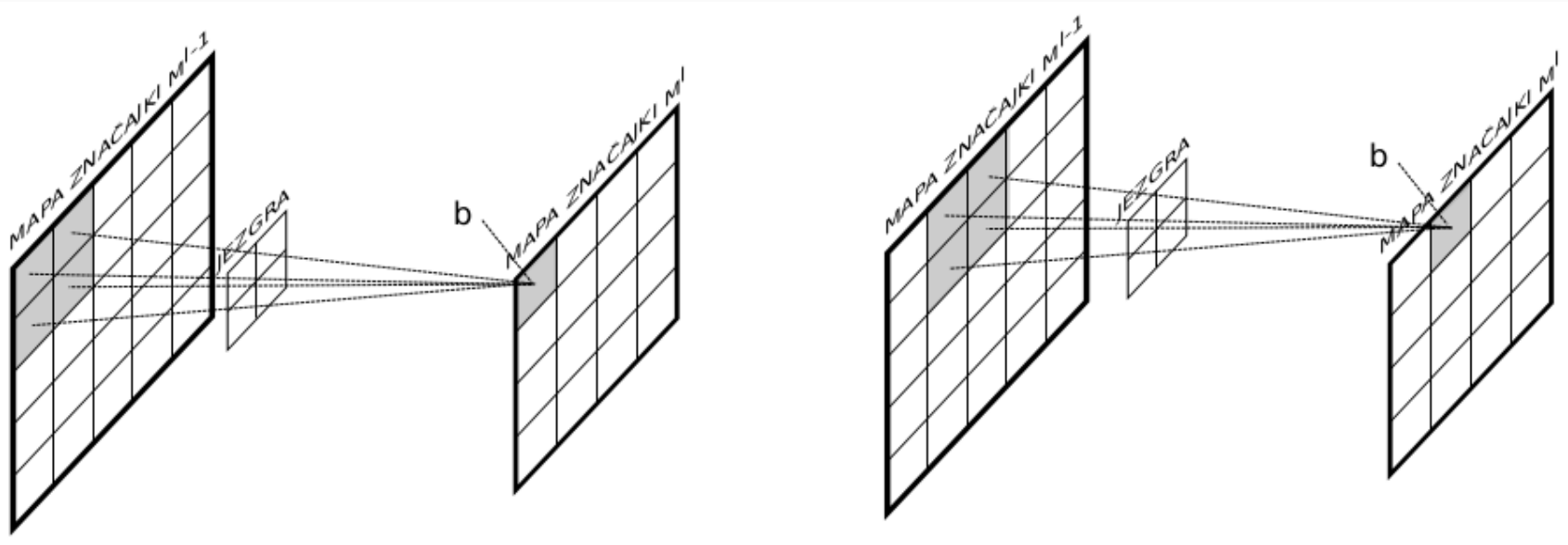
What is convolution?





What is convolution?

Convolutions share parameters and model local interactions:



The reationship between input and output is linear, but:

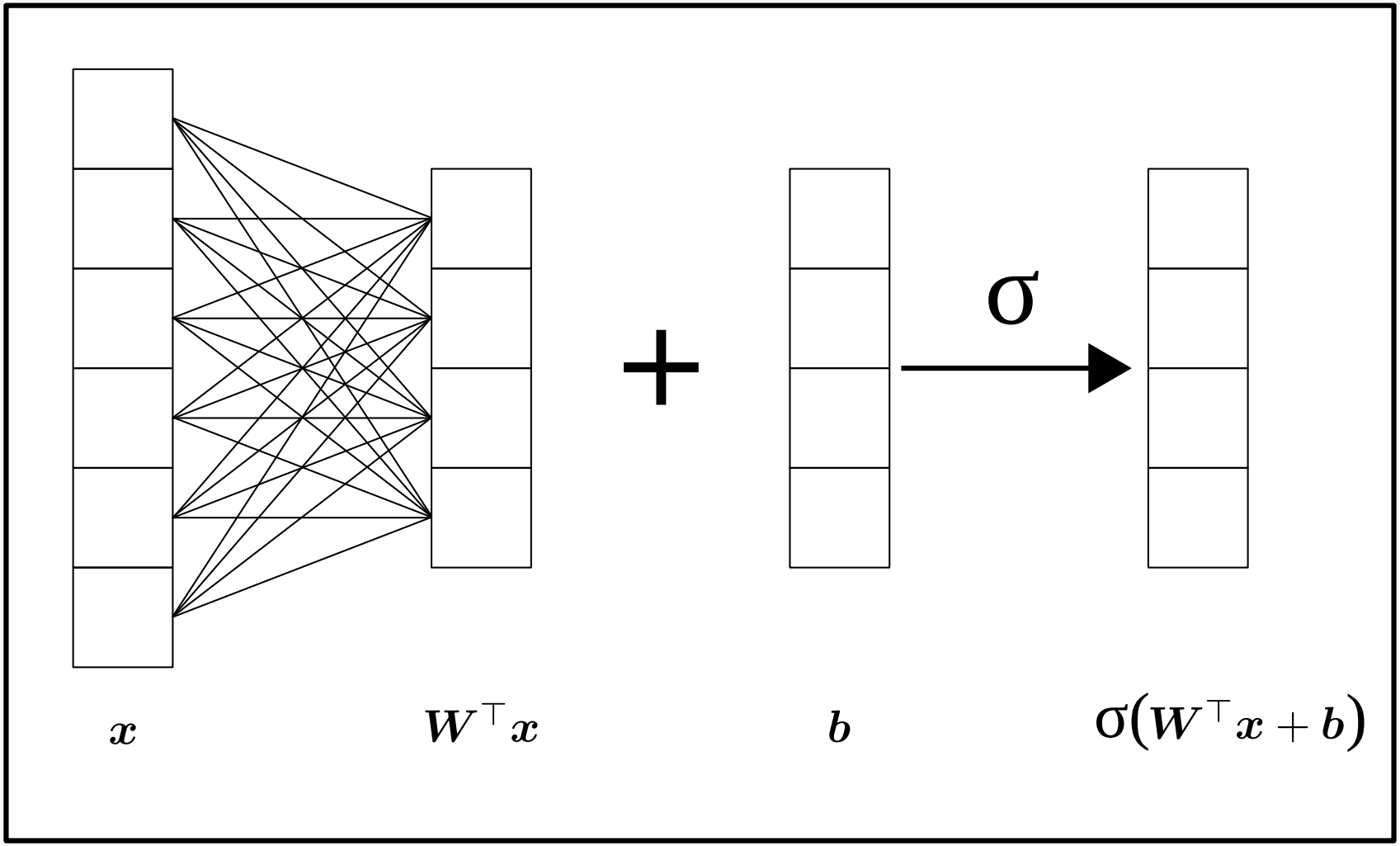
- elements of the output M^l depend on the corresponding **local neighbourhood** of the input M^{l-1}
- all outputs use the **same** (shared) set of parameters

Why convolution?

Convolution is similar to a fully connected layer, but there are important differences:

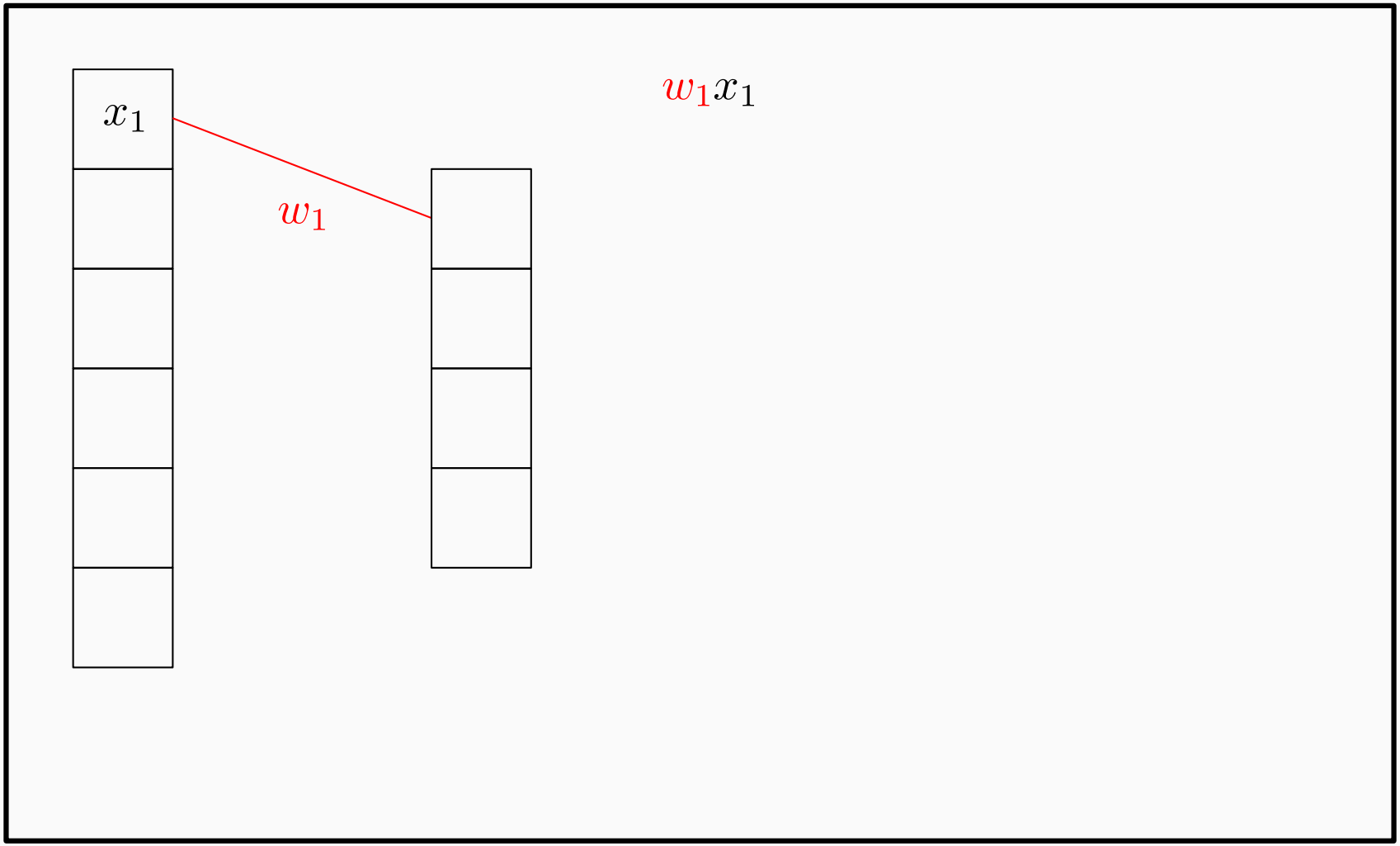
- they can model only local interactions
- parameter sharing $\Rightarrow$ output representation is **equivariant** with respect to translation.

Fully connected layer (linear part)

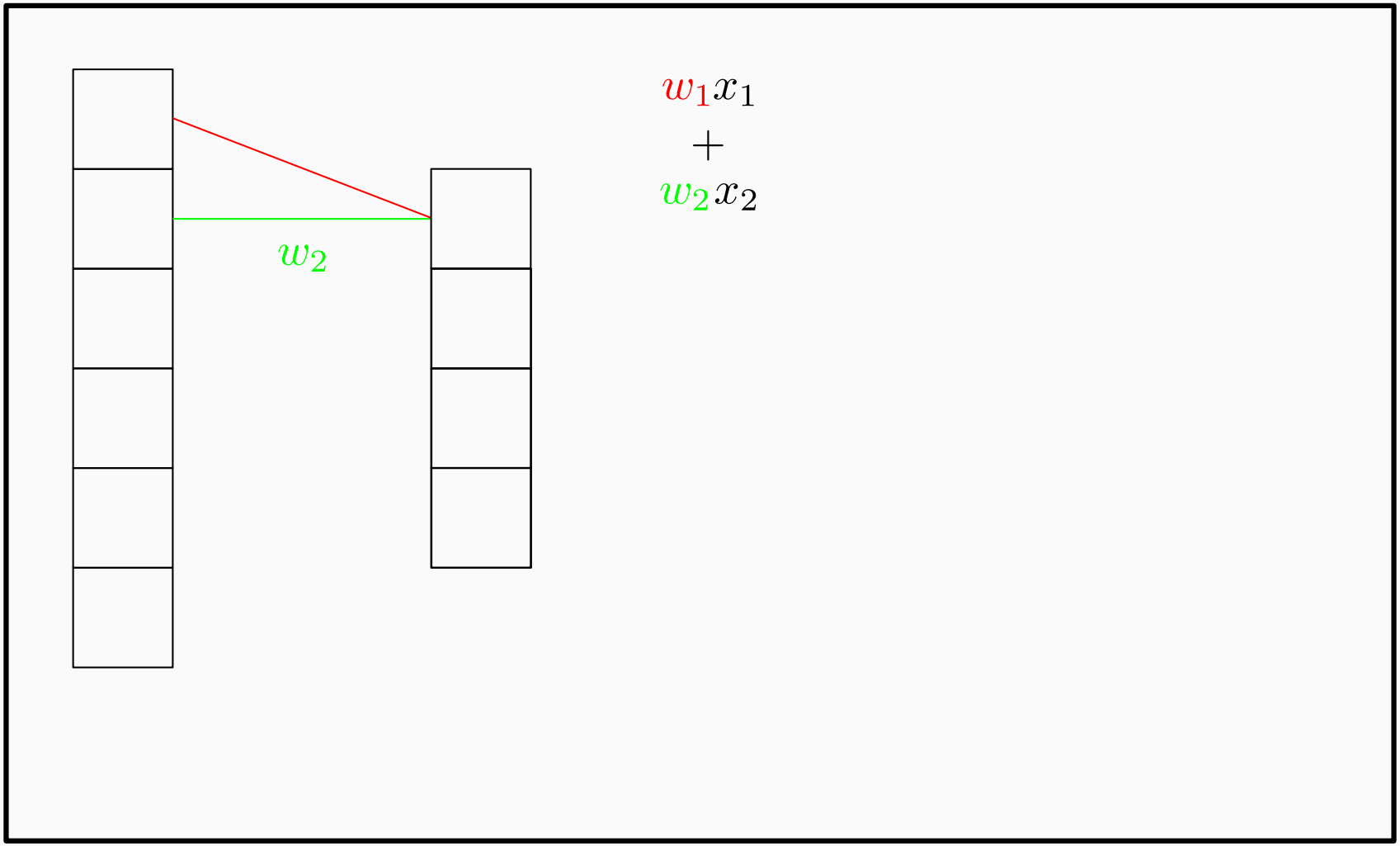


$$f(x, \Theta = (W, b)) = Wx + b$$

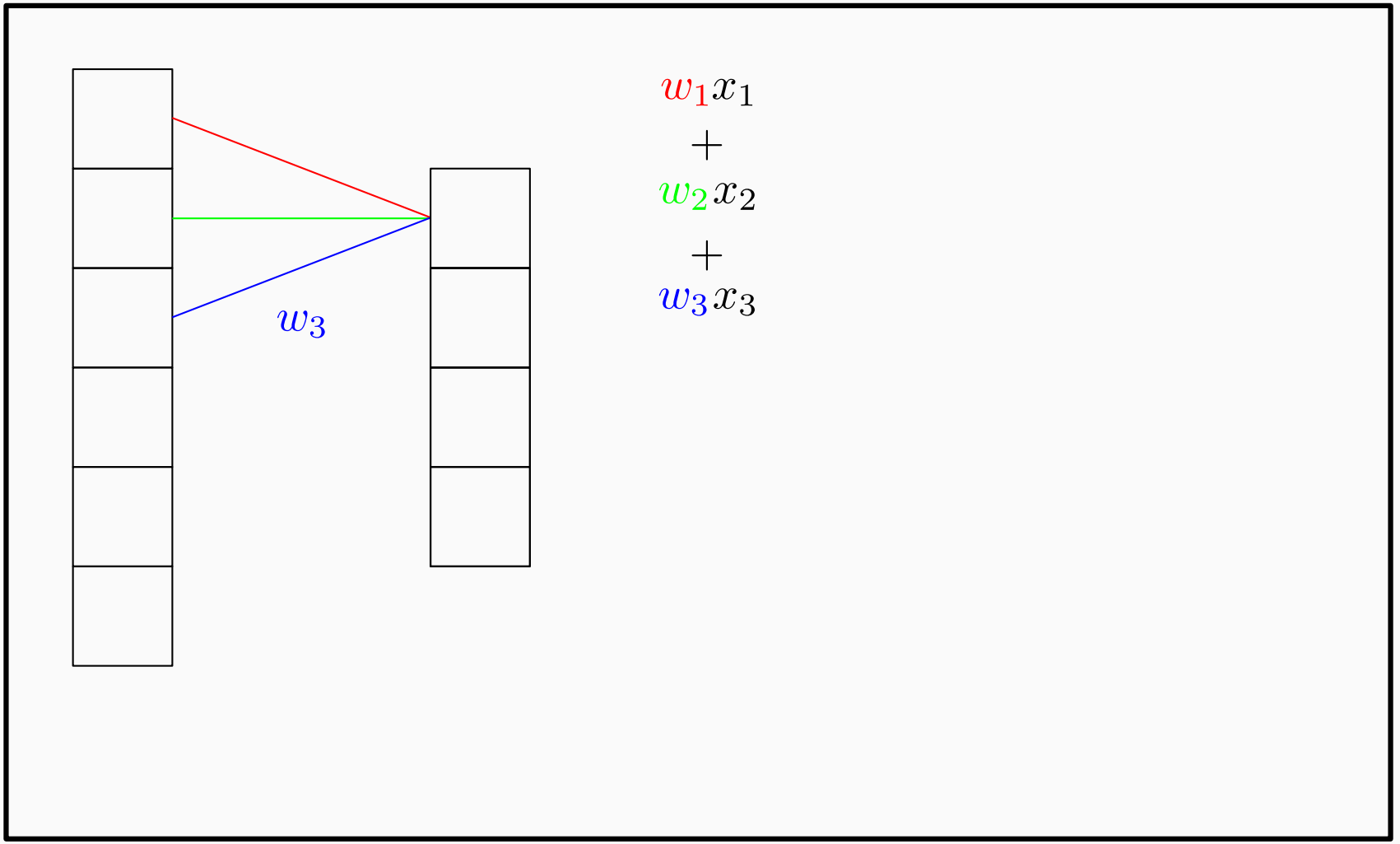
Convolutional layer (1D, linear part)



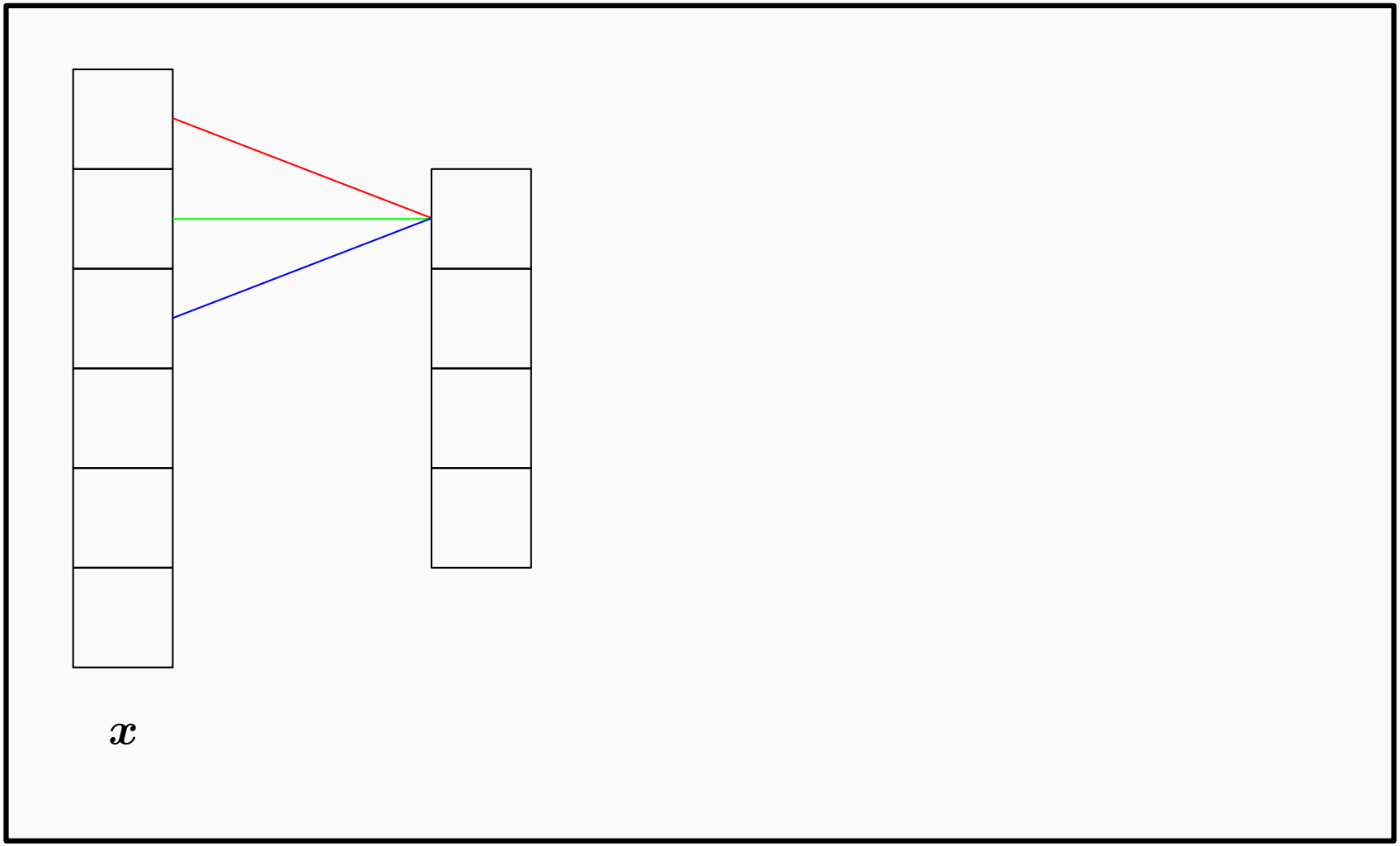
Convolutional layer (1D, linear part)



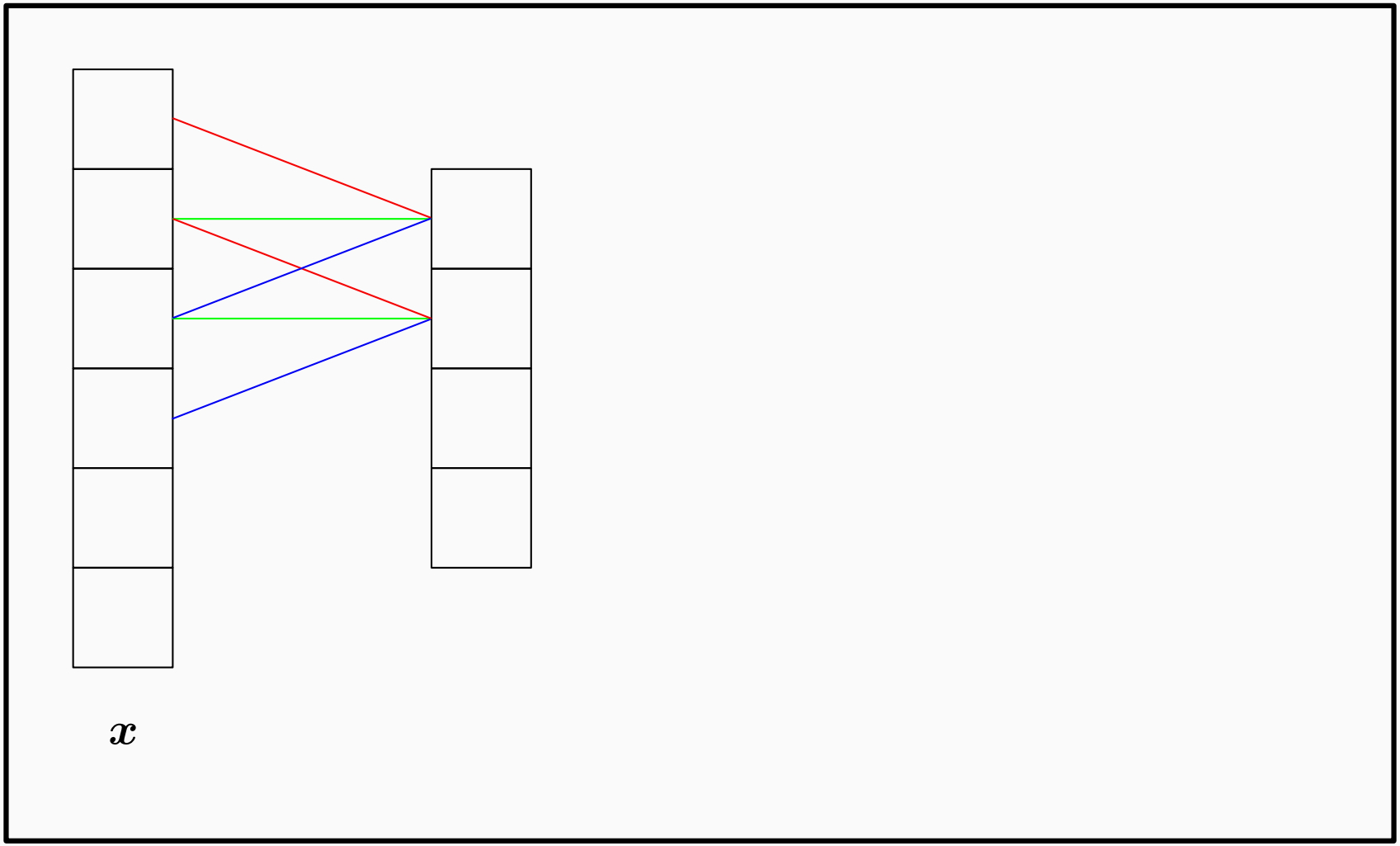
Convolutional layer (1D, linear part)



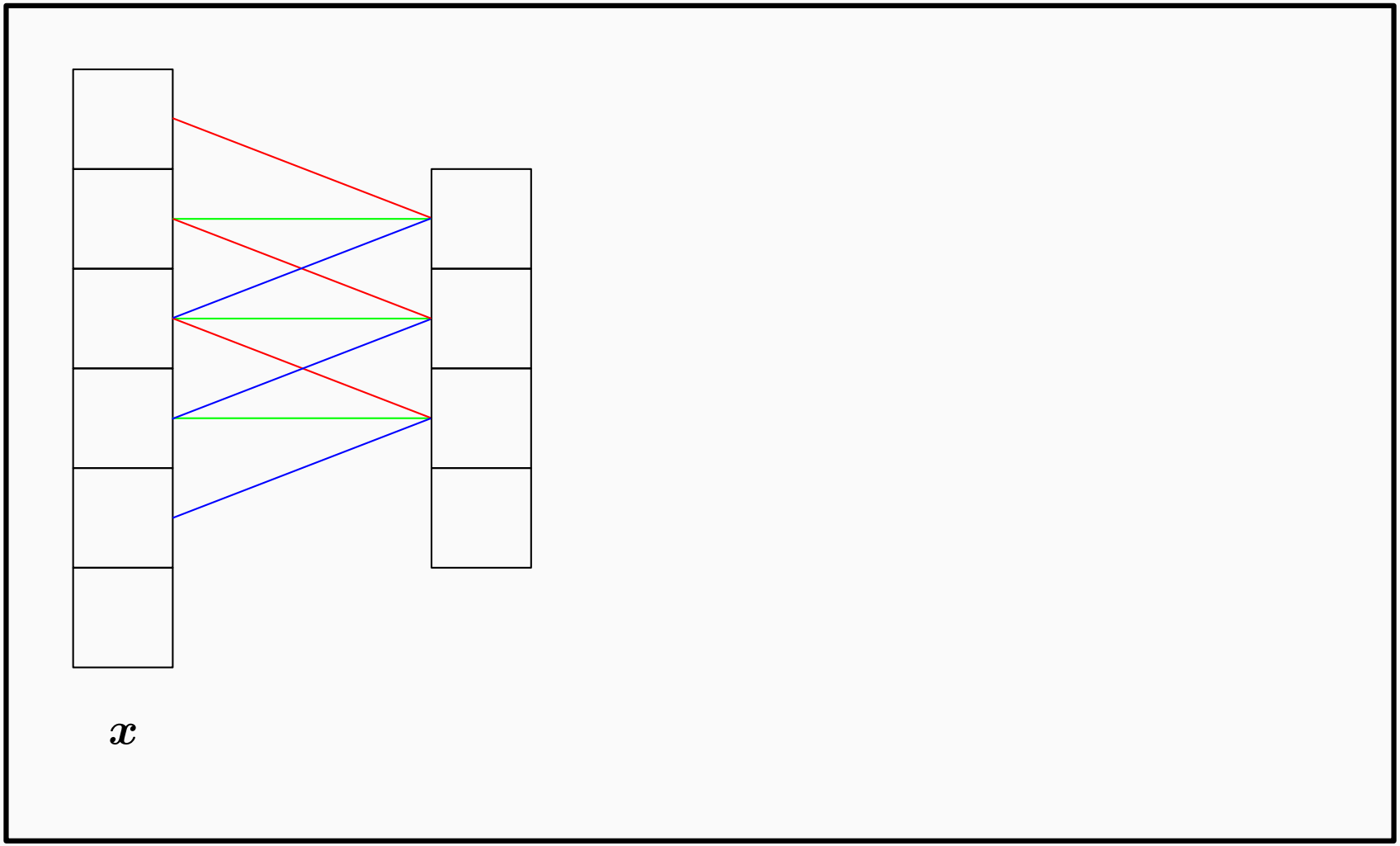
Convolutional layer (1D, linear part)



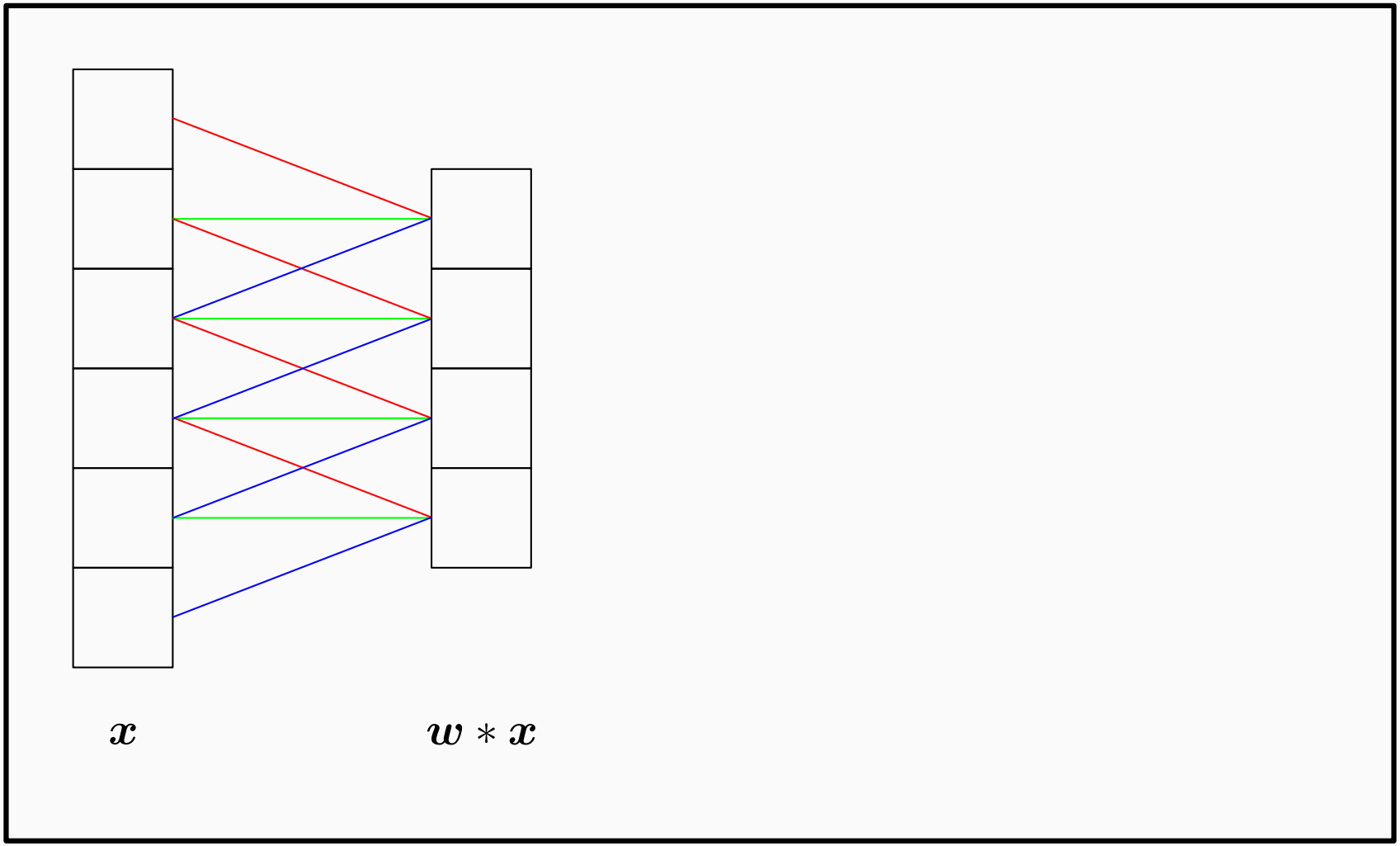
Convolutional layer (1D, linear part)



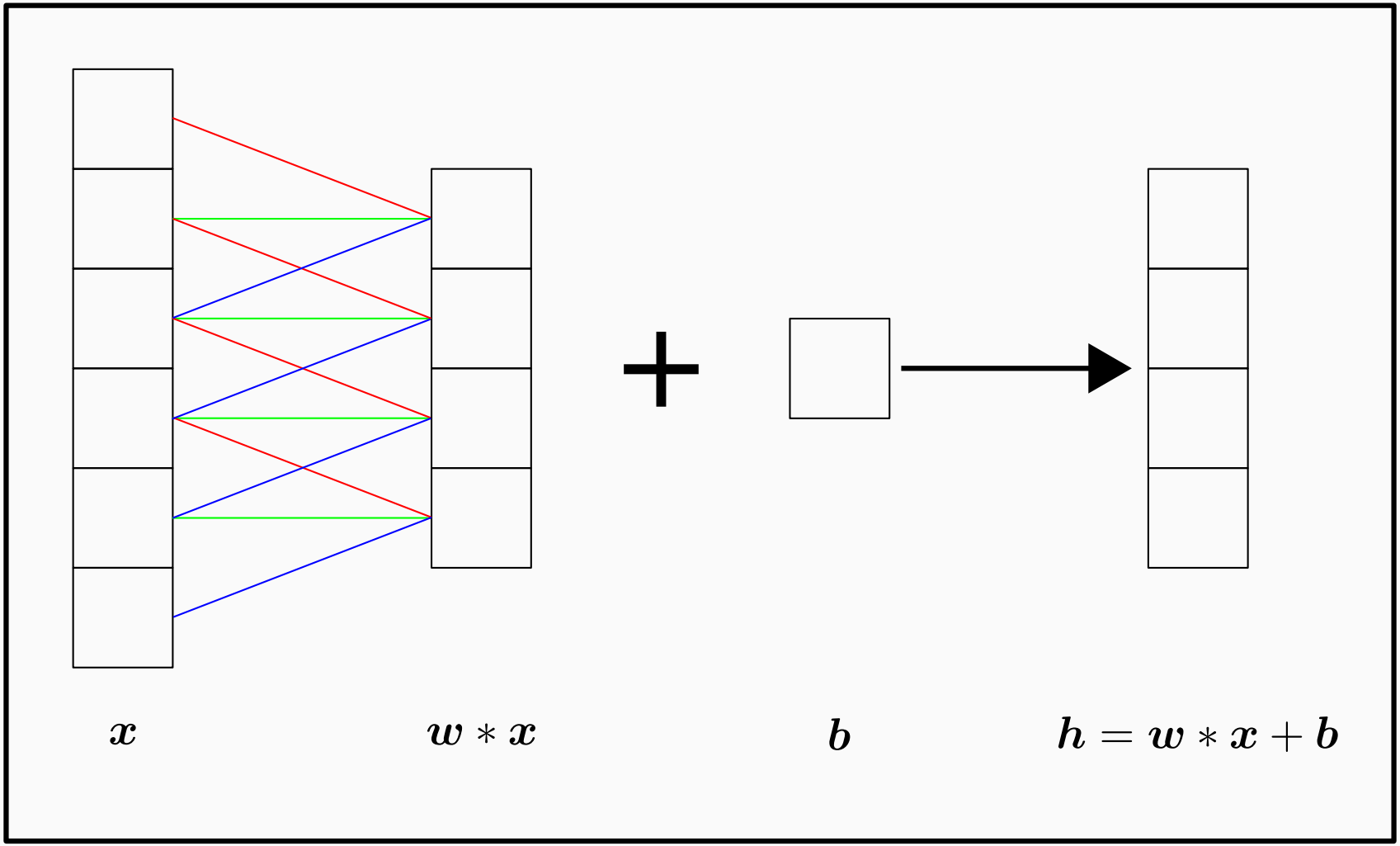
Convolutional layer (1D, linear part)



Convolutional layer (1D, linear part)



Convolutional layer (1D, linear part)

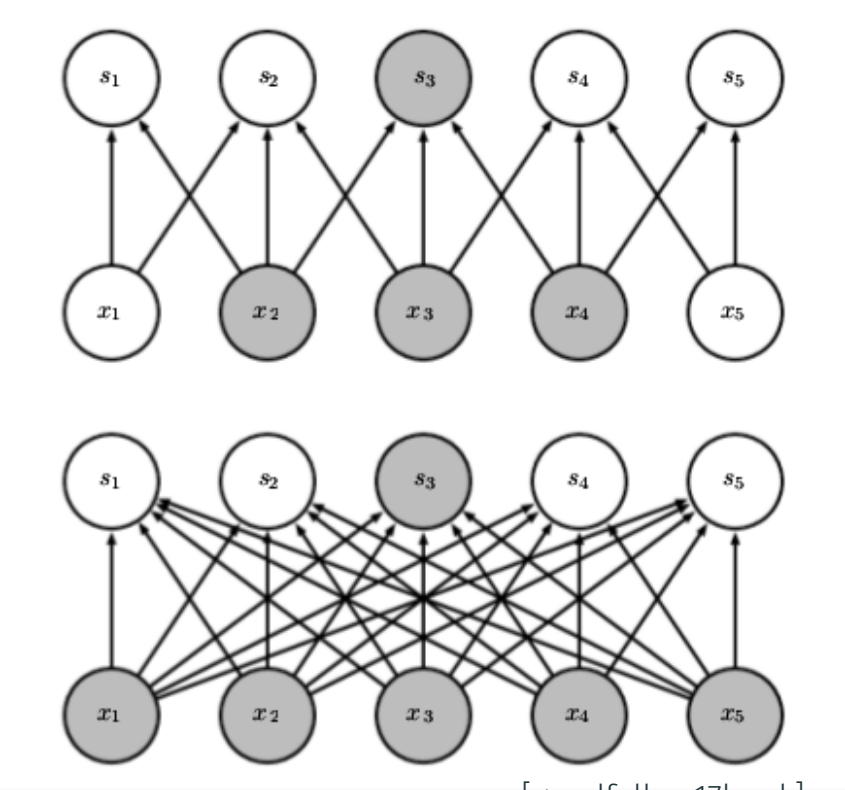


$$f(x, \Theta = (w, b)) = w * x + b$$

Local interactions

Comparison of convolution with affine transformation:

- convolutional activations (above) "see" only a few inputs
- affine activations (below) "see" all inputs
- convolutions induce fewer connections and parameters.

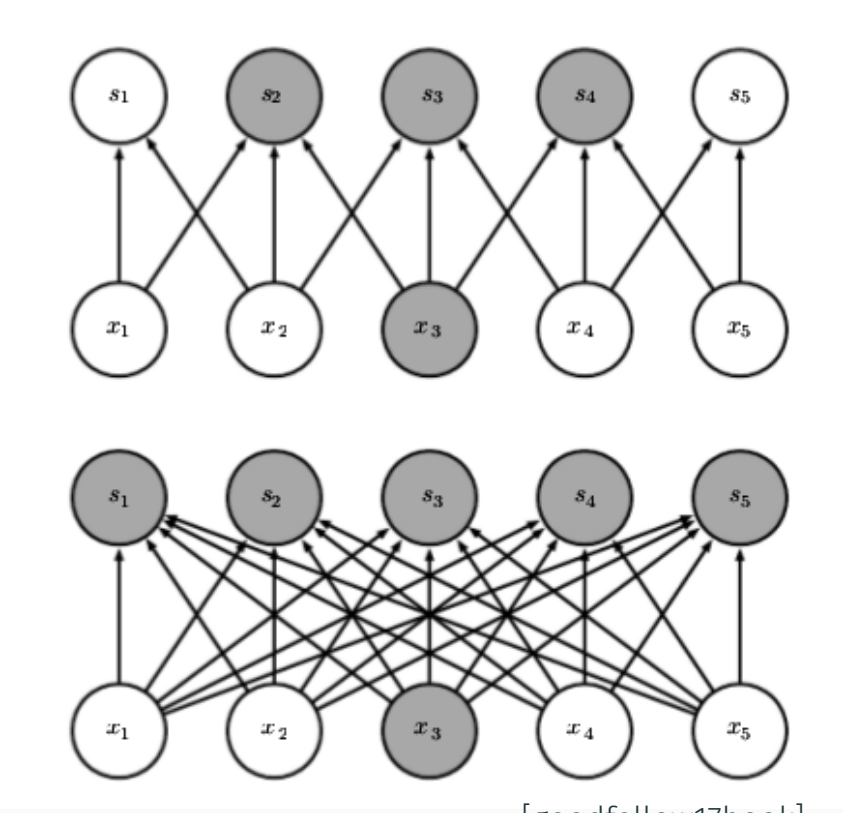


[goodfellow17book]

Local interactions (2)

Comparison of convolution with affine transformation (2):

- the convolution inputs (above) affect only few outputs
- this suggests that the backward pass (back-propagation of gradients) will also be expressed by convolution



[goodfellow17book]

Local interactions (3)

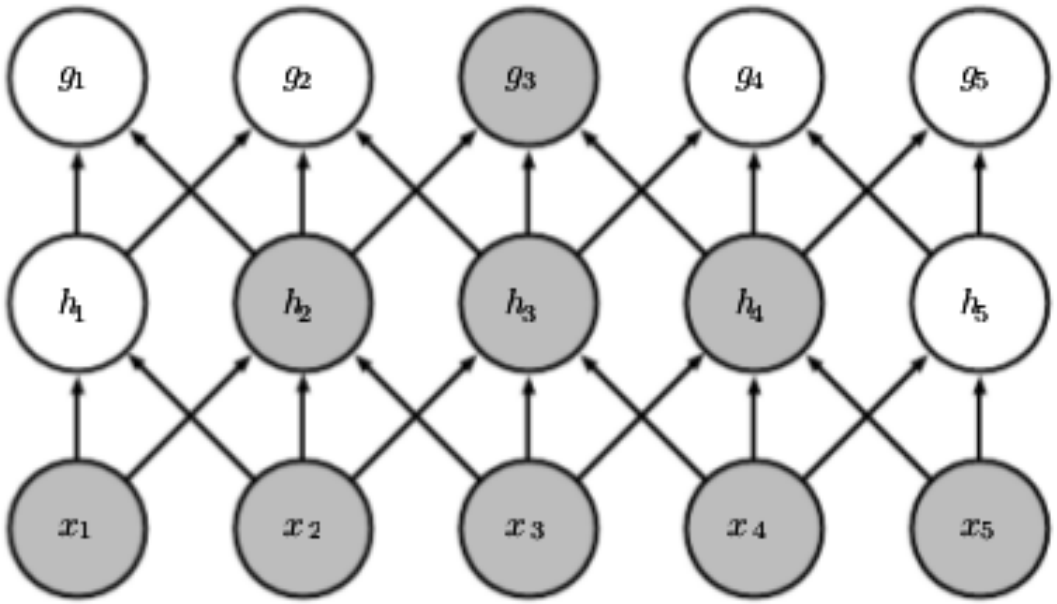
Advantages of convolution over affine transformation:

- faster inference $O(m \cdot n)$ *vs* $O(k \cdot n)$
  - k kernel size
  - m input dimension
  - $k \ll m$.
- less capacity: $k \cdot n$ *vs* $m \cdot n$ parametars
  - fewer parameters to learn with the same amount of labels.
  - here we only consider locality and ignore parameter sharing!.

Local interactions (4)

Features of a deep convolutional model can indirectly interact with a large input region:

- receptive field: the set of all **input** elements that affect the particular feature
- receptive field of convolutional activations increases with depth.



[goodfellow17book]

Parameter sharing (or weight-tying)

All outputs are computed with respect to the same set of weights:

- the weights for computing the output s_{00} are the same as the weights for $s_{kl} \quad \forall k, l$

Instead of learning separate sets of parameters, each output activation uses the same parameters:

- more learning signal
- less susceptible to overfitting

Parameter sharing (2)

Advantages over affine transformation (2):

- even fewer parameters: $m \cdot n$ vs k (superior statistical efficiency)
- computational complexity: the same as for the model that has only local interactions but does not share weights: $O(n \cdot k)$.

Equivariance to translation

$f(x)$ is equivariant with respect to g if:

$$f(g(x)) = g(f(x))$$

Convolution (f) is equivariant with respect to translation (g):

- the convolution output is a spatial map of dense latent features of the input tensor (time sequence, matrix, volume)
- if we translate the input, the feature map will translate accordingly.
- convolutional models are suitable for images, speech, language, bioinformatics, ...

Convolution is not equivariant with respect to some other transformations such as scaling or rotation.

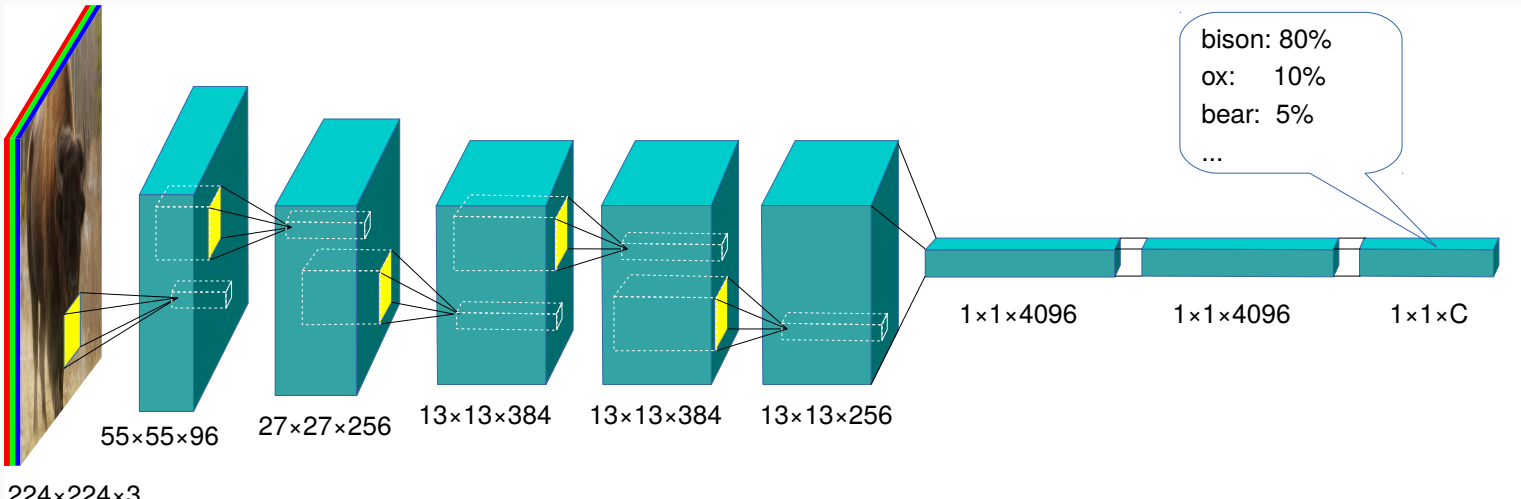
Pooling layers

Pooling layers compress input neighbourhoods into a single vector.

Usually, the output is a statistical indicator of the input neighbourhood, e.g. mean or maximum.

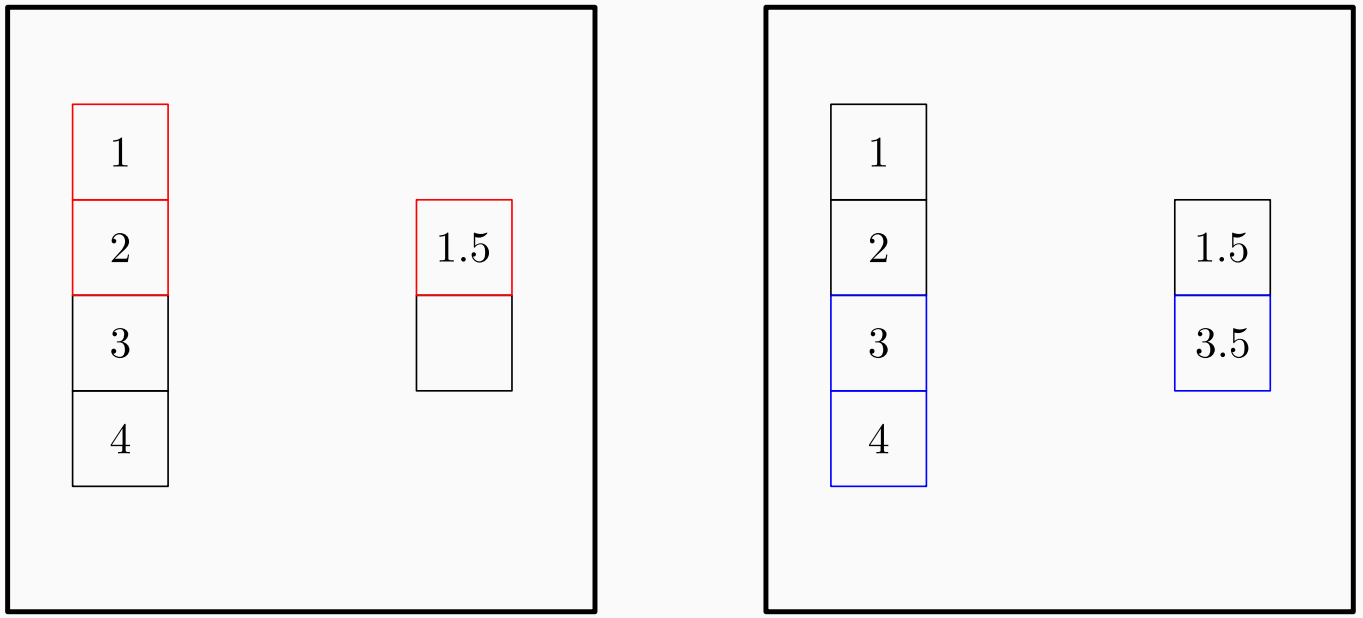
We apply pooling when the model converts the input tensor (image, sentence) into a scalar prediction:

- we need to re-knead a pizza dough into a baguette shape
- this is not the only notivation for pooling layers



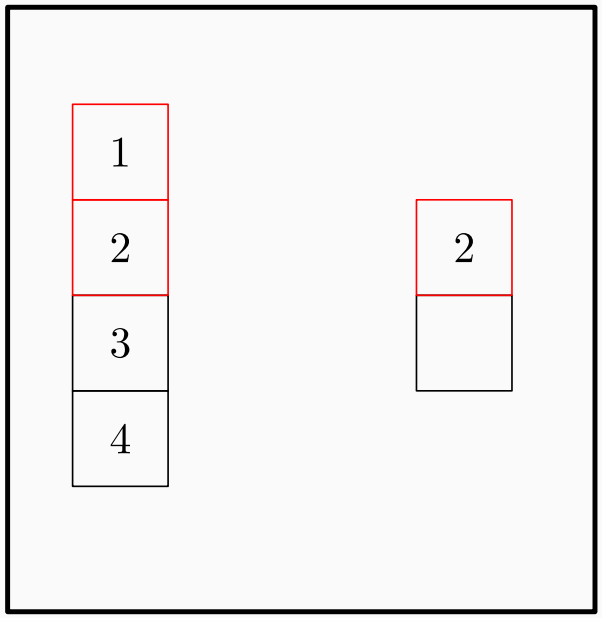
Pooling functions

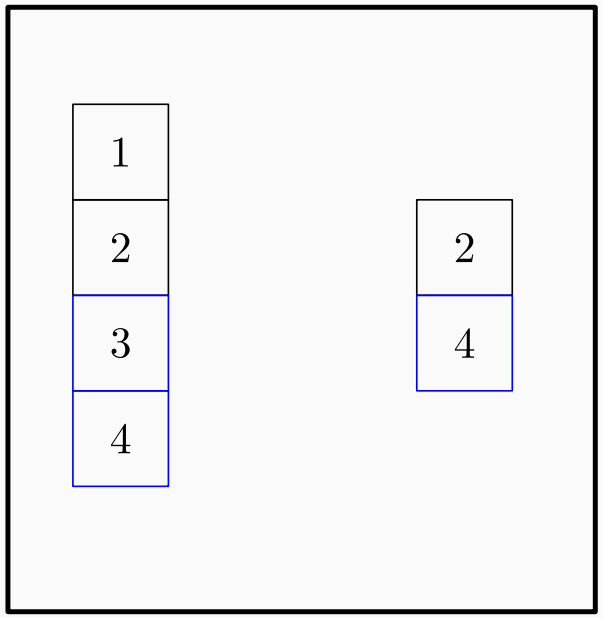
Mean-pooling with kernel size 2



Pooling functions

Max-pooling with kernel size 2

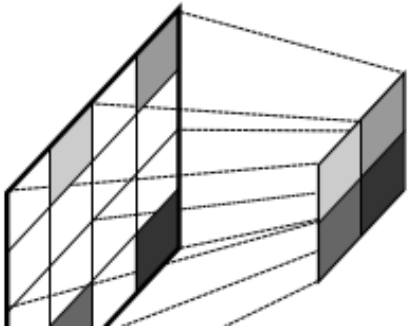


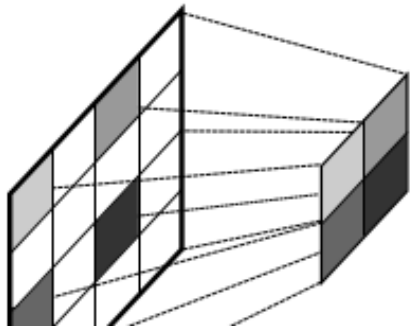


Pooling motivation

Increase translation invariance:

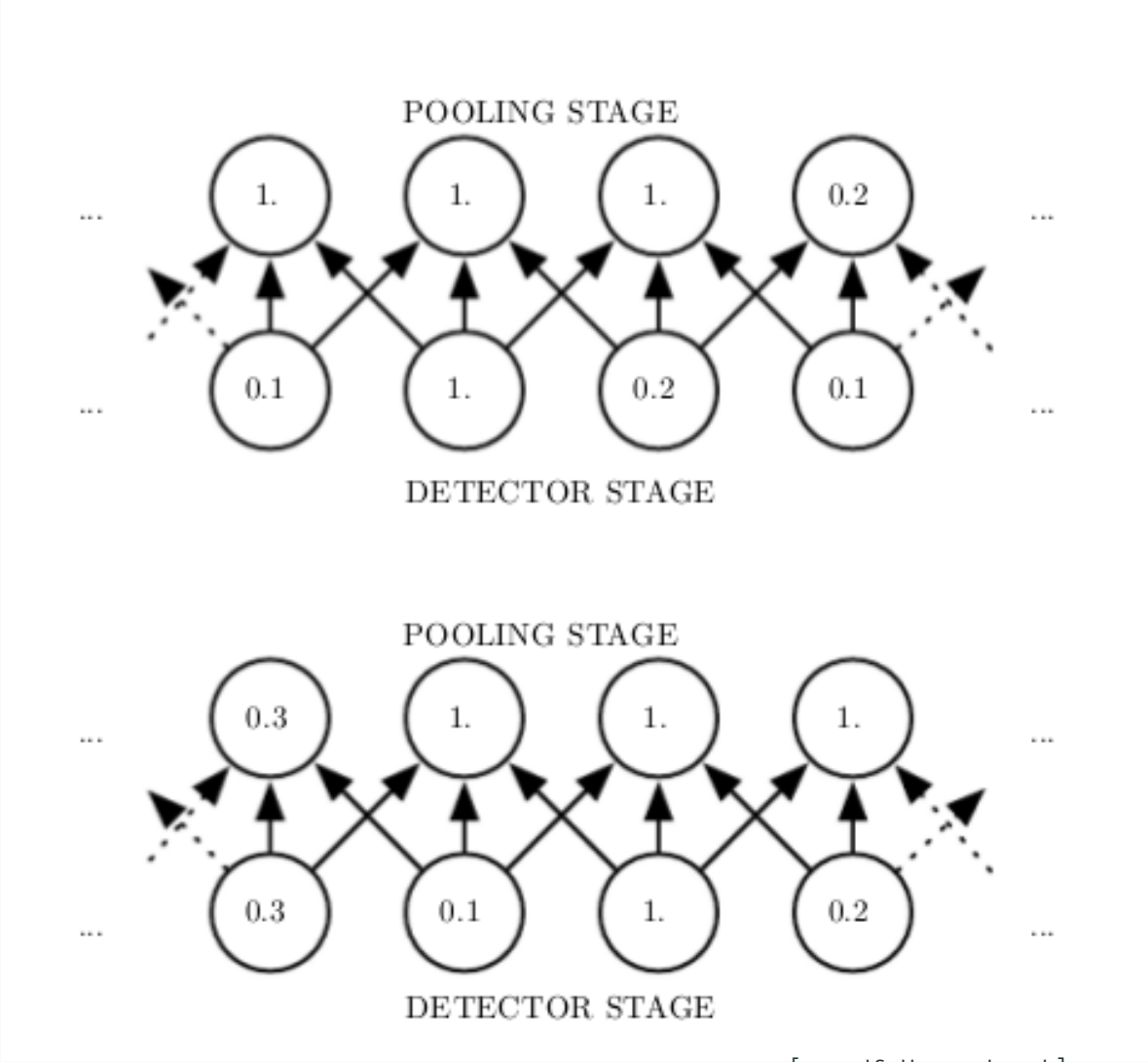
- $f(x)$ is invariant with respect to g if: $f(g(x)) = f(x)$
- especially useful if the model has to detect *presence* of the concept rather than the location:
  - eg. in face detection, displacement of the eyes in relation to the nose varies from person to person
- size of the pooling region regulates the amount of invariance:
  larger region $\rightarrow$ invariance to larger displacements
  - eg. in image categorization, the object that defines the class can be anywhere in the image





Pooling motivation (2)

Pooling layers allow to aggregate evidence across input regions:

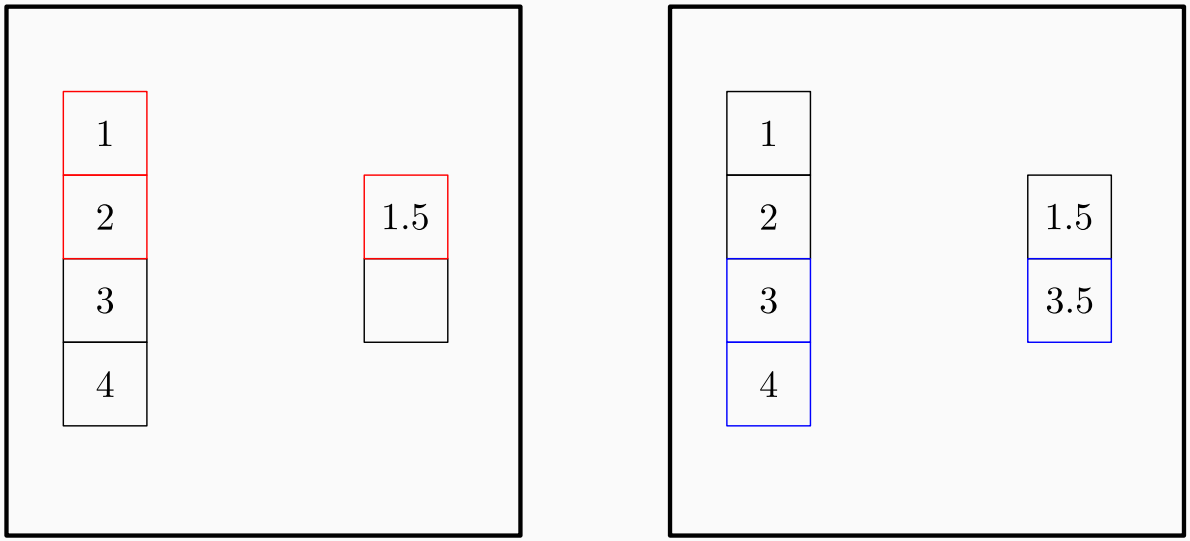


[goodfellow17book]

Pooling motivation (3)

Decrease computational complexity:

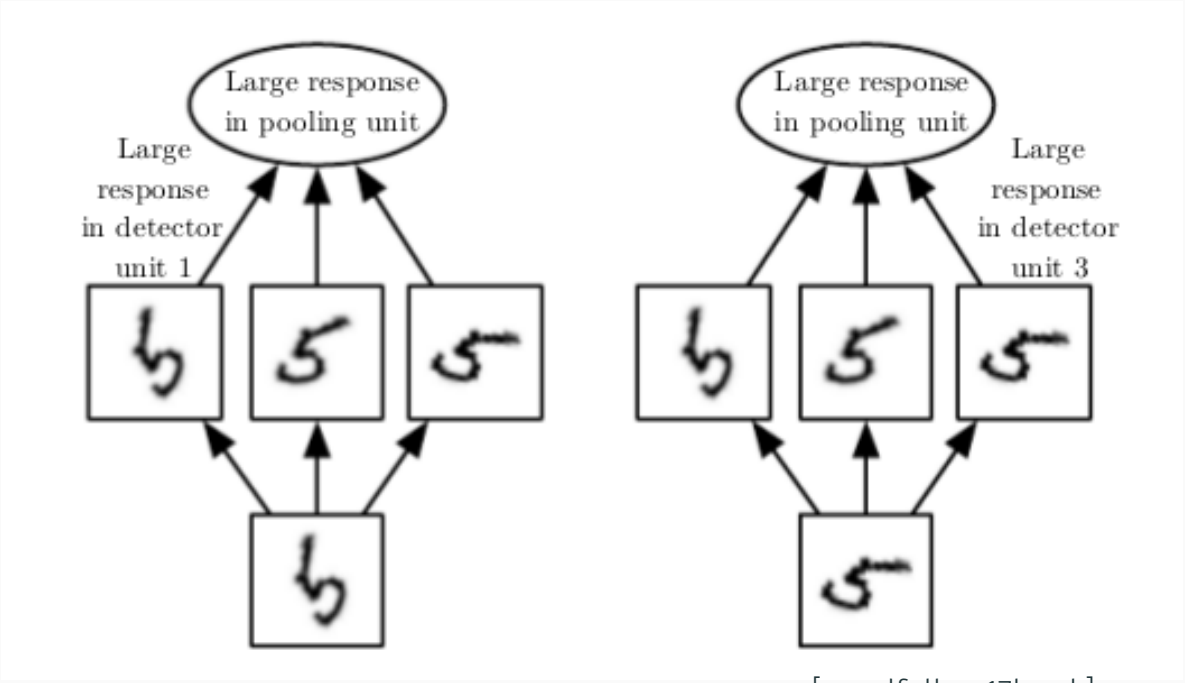
- most contemporaneous archiectures reduce the input resolution to $H/4 \times W/4$ after the first convolutional layer
- this reduces the latency both during training and inference
- furthermore, this decreases the memory footprint of the model (this is often more important than speed)



Pooling motivation (4)

Pooling can be carried out not only over adjacent features, but also across different feature maps.

In this case the model can learn invariance with respect to different transformations.

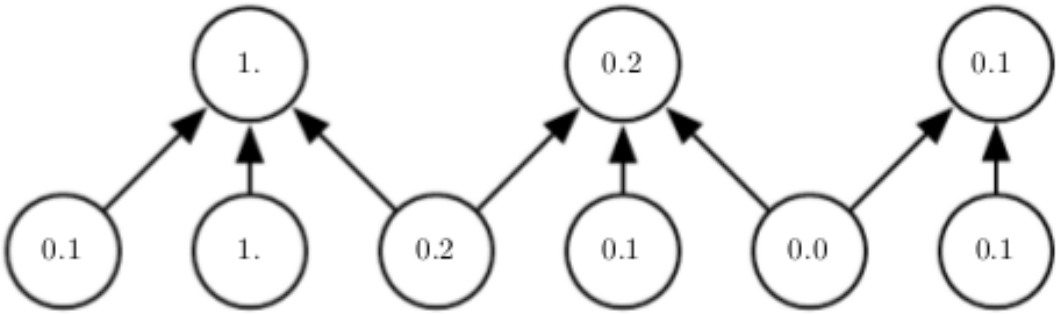


[goodfellow17book]

Pooling usage

Pooling layers typically involve an output stride $k > 1$:

- the feature map is divided into regular spatial regions
- each region pools into one feature with unchanged semantic dimensionality
- the output feature map is $k\times$ subsampled
  - most often: $k = 2$, (k_h, k_w)=$(H/q, W/q)$ or (k_h, k_w)=(H, W).
- regions may overlap (cf. figure below) although this is seldom used

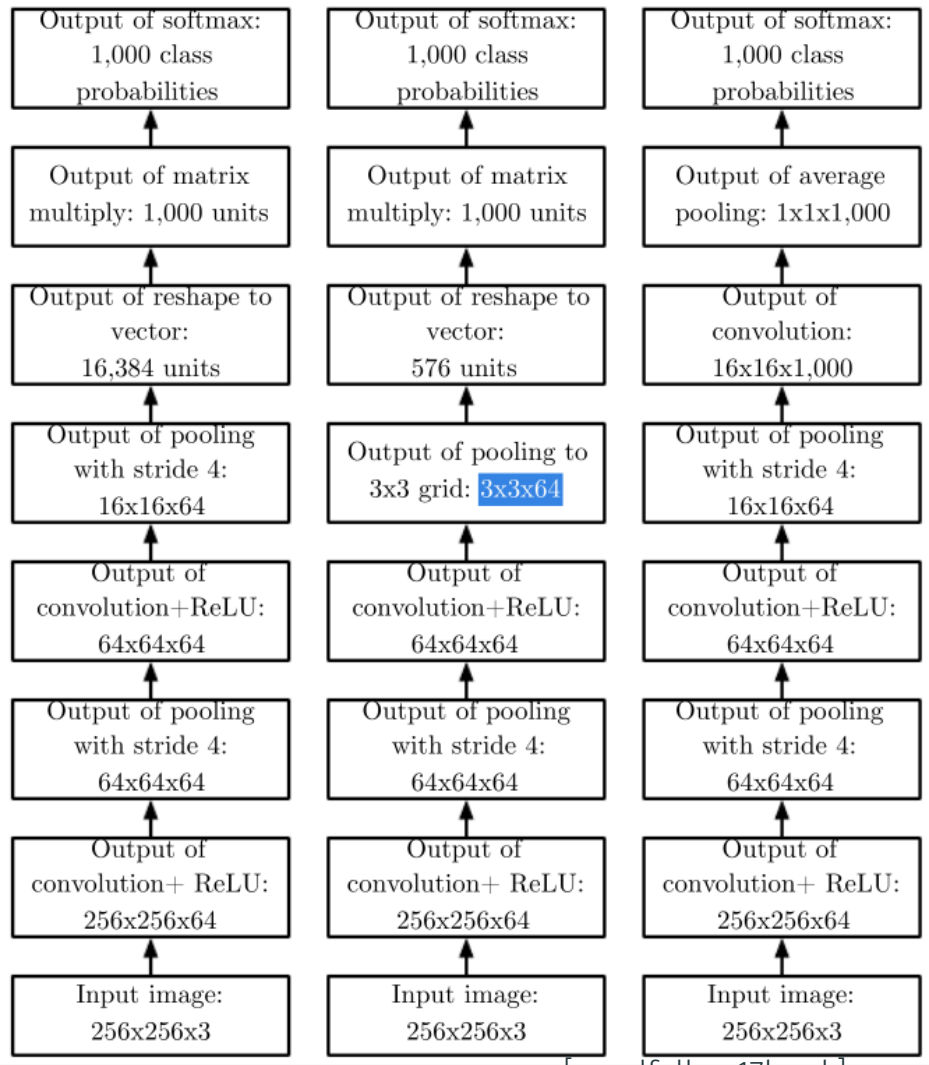


[goodfellow17book]

Pooling usage (2)

The size of pooling regions may be adaptive

- this allows to process inputs of different sizes

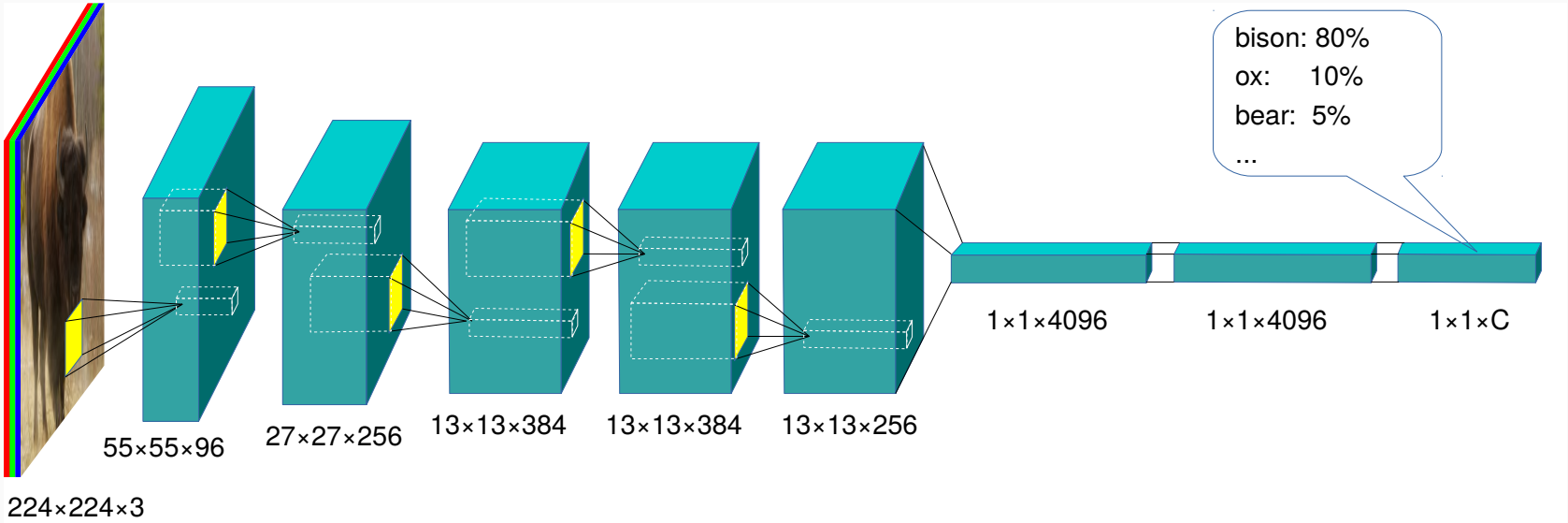


[goodfellow17book]

Pooling usage (3)

Questions:

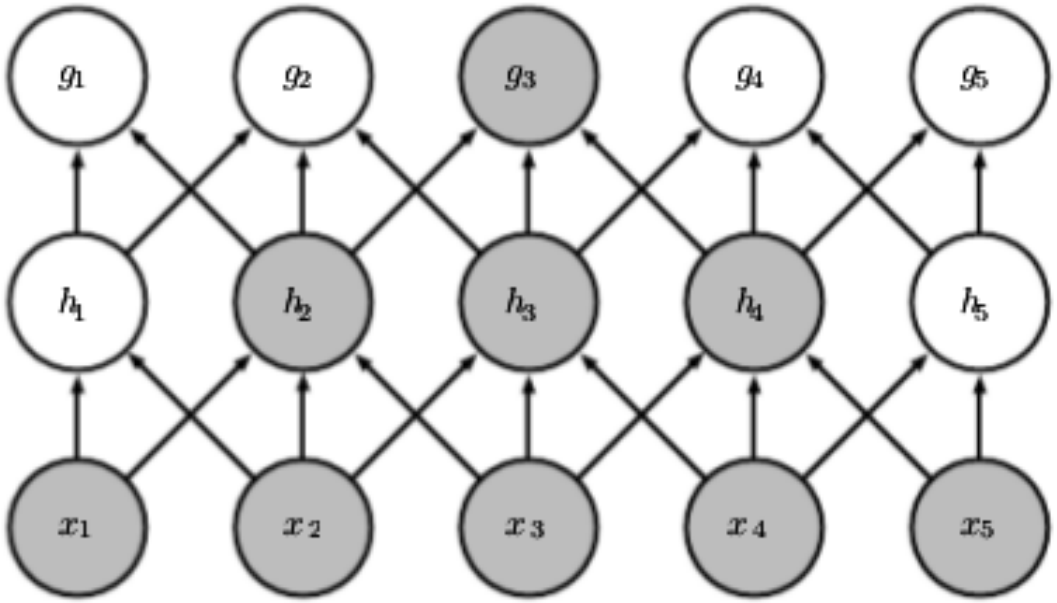
- what happens with the receptive field of the convolutional features that operate on pooled features (assume output stride k)?
- how does the pooling affect the number of model parameters?



Receptive field

Effect of convolution with the kernel size k:

- increase the receptive field of the output for $k - 1$ (if there were no pooling layers...)



[goodfellow17book]

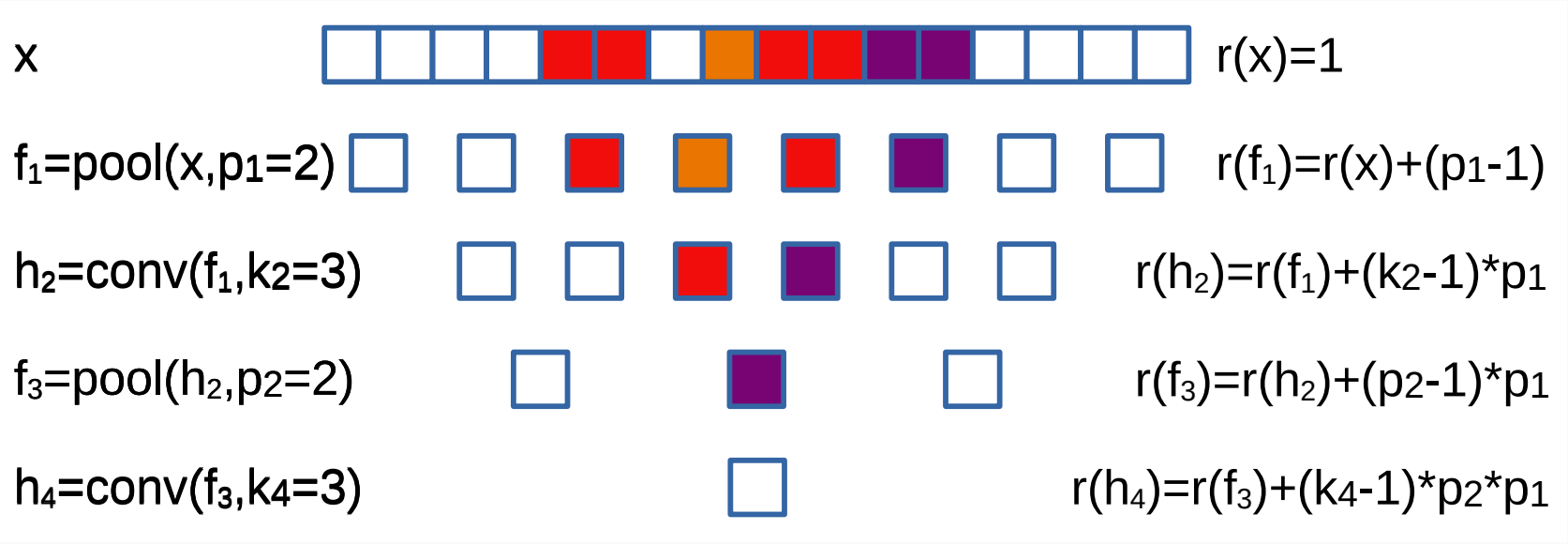
Effect of subsampling with output stride k:

- increase the receptive field of the output for $k - 1$

Receptive field (2)

We can measure the receptive field by forward analysis:

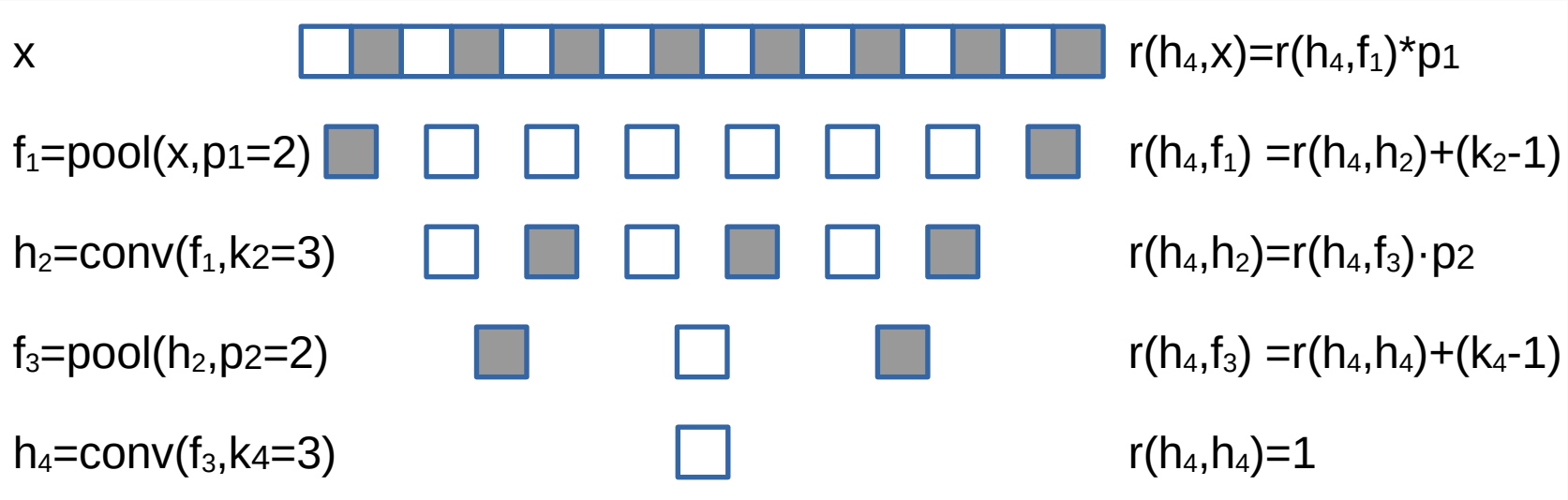
- the red color indicates the receptive field of the bottom-most red activation (same for orange, violet and white)
- we must keep count of the total subsampling factor (eg. $p_2 \cdot p_1$) that multiplies the receptive contribution (eg. $k_4 - 1$)



Receptive field (3)

We can also measure the receptive field by backward analysis:

- we assume that the mother grows towards early layers
- grey activtions denote the increase of the receptive field with respect to the earlier layer
- advantage: we do not have to keep count of the total subsampling factor (harder to make a mistake)



Bias of convolution and pooling

Convolutions and poolings introduce the following pieces of bias:

- convolution: all interactions are local $\rightarrow$ the model will gneralize well on data with lattice topology
- convolution: predictions are translation-equivariant
- pooling $\times k$: predictions are invariant to small translations
- global pooling: predictions are translation-invariant

These assumptions increase bias and decrease variance:

- theory: this can lead both to good generalization and under-fitting
- practice: no under-fitting, convolutional models generalize better than fully connected ones

Bias of convolution and pooling (2)

Convolution can be viewed as a fully connected layer which zeros all weights outside the kernel domain:

- this intervention will enlarge the loss on training data

If a convolutional layer leads to underfitting:

- the local interactions may be insufficient $\rightarrow$ increase the receptive field

If a pooling layer leads to underfitting:

- the desired functionality requires accurate feature locations $\rightarrow$ reduce the pooling region

Convolutions with padding

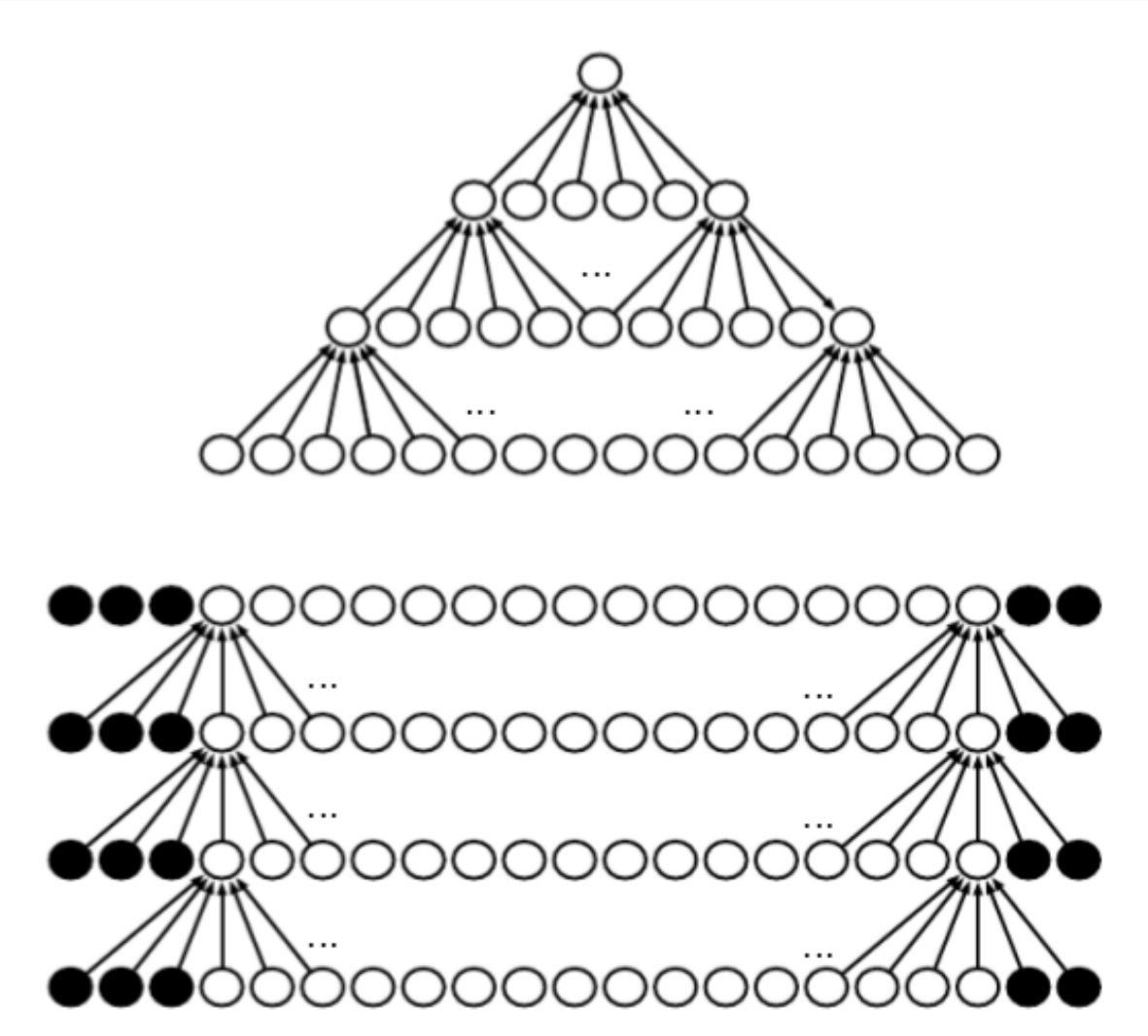
Default convolution: latent representations shrink with depth

- if input size is m, than the output is $m - k + 1$ (k denotes the kernel size)
- shortcoming: borders have less effect to the predictions
- shortcoming: the depth is limited by m and k
- some software frameworks denote such convolution as `"VALID"`.

Convolution with zero-padded input: model depth is unlimited

- if we pad the input with $k - 1$ zeros the output will have the same shape as the input
- example for $k = 5$: we pad two zeros to all input borders
- frameworks denote such convolutions as `"SAME"`

Convolutions with padding (2)



Convolution: problem 1

We consider a convolutional model for classifying grayscale images of shape 28×28. The model consists of the following components:

- two processing blocks: 5×5 convolution with bias and no padding; ReLU activation; max-pooling 2×2 stride 2;
  - conv1: 16 channels, conv2: 32 channels;
- flatten
- fully connected layer with bias, 512D output, ReLU activation;
- fully connected layer with bias, 10D output, softmax.

Tasks:

1. determine dimensions of latent tensors, the number of parameters and the size of the receptive field in all layers;
2. propose a PyTorch implementation with methods

Convolution: problem 2

Propose your own implementation of 1D convolution under Numpy (one for-loop, only forward pass).

```
import numpy as np

def conv1d_my(vector, kernel):
  n = vector.shape[0]
  k = kernel.shape[0]
  out = np.zeros((n-k+1,), dtype=np.float32)
  kernel = np.flip(kernel)
  for i in range(n-k+1):
    out[i] = np.sum(vector[i:i+k] * kernel)
  return out
```

Convolution: problem 3

Compare the functionality of your implementation of convolution with the corresponding implementations from torch and scipy.

```
import torch
from torch.nn.functional import conv1d as conv1d_torch
from scipy.ndimage import convolve1d as conv1d_scipy

x = np.array([2.0]*3 + [6.0]*4)
w = np.array([-1.0, 1.0])
print(conv1d_my(x,w))
print(conv1d_scipy(x, w, mode='nearest'))
print(conv1d_torch(torch.tensor(x).reshape([1,1,-1]),
  torch.tensor(w).reshape([1,1,-1])).numpy().squeeze())

# [ 0.  0. -4.  0.  0.  0.]
# [ 0.  0. -4.  0.  0.  0.  0.]
# [ 0.  0. -4.  0.  0.  0.]
```