Convolutional Architectures For Image Classification

Advanced Computer Vision

Niels Landwehr

Overview

- Introduction: Computer Vision
- Data, Models, Optimization
- Neural Networks and Automatic Differentiation
- Convolutional Architectures For Image Classification

Recap: Image Data as 3D-Tensors

- Reminder: in image processing, images are 3D-tensors

3D-tensor, $\mathbb{R}^{3\times3\times3}$

$$\begin{pmatrix} \begin{pmatrix} 0 & 0.3 & 1 \\ 0 & 0.3 & 1 \\ 0 & 0.3 & 1 \end{pmatrix} & \begin{matrix} 5 \\ 5 \\ 1 \end{matrix} \\ \begin{pmatrix} 1 & 0.7 & 0.5 \\ 0.2 & 0.5 & 0.1 \end{pmatrix} & \begin{matrix} 1 \end{matrix} \end{pmatrix}$$

$$\mathbf{x} \in \mathcal{X} = \mathbb{R}^{m\times l\times d}$$

m = image height

l = image width

d = number of channels

number of channels can be 1 (greyscale images), 3 (RGB images), or >3 (hyperspectral images)

- So far, we ignored the structure and flattened the 3D-tensor into a 1D vector
- Example: linear model for Cifar-10 image classification

$$\mathcal{X} = \mathbb{R}^{32\times32\times3}$$

$$\begin{pmatrix} 0 \\ 0.3 \\ 0.1 \\ \dots \\ 0.7 \end{pmatrix}$$

$$\mathcal{X} = \mathbb{R}^{3072}$$

Linear model (scores for 10 classes):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^{10}$$

$$\mathbf{W} \in \mathbb{R}^{10\times3072}, \quad \mathbf{b} \in \mathbb{R}^{10}$$

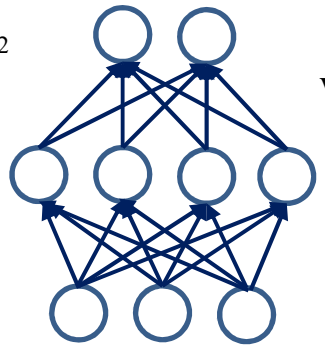
Recap: Multilayer Perceptrons

- Reminder: multilayer perceptrons (fully connected feedforward neural networks) work on flat vectors

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_2\mathbf{z}_1 + \mathbf{b}_2$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{x} \in \mathbb{R}^d$$



$$\mathbf{W}_2 \in \mathbb{R}^{k \times k_1}, \quad \mathbf{b}_2 \in \mathbb{R}^k$$

$$\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \quad \mathbf{b}_1 \in \mathbb{R}^{k_1}$$

Input is vector

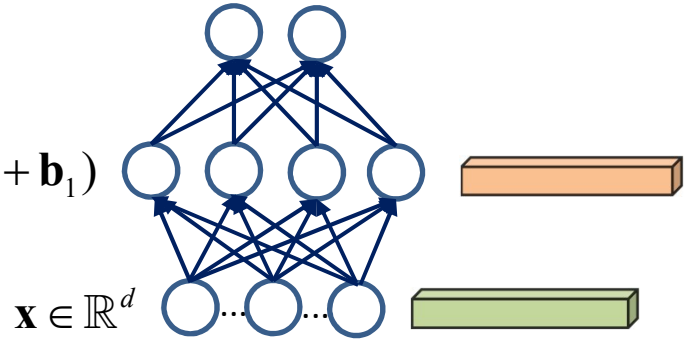
($\in \mathbb{R}^3$ in example)

- Non-linear model, more expressive than simple linear model
- Not particularly good for image data: we should take the spatial image structure into account
- **Convolutional Neural Networks**: neural networks that reflect the 3D structure of image data directly in the architecture and the computational operations

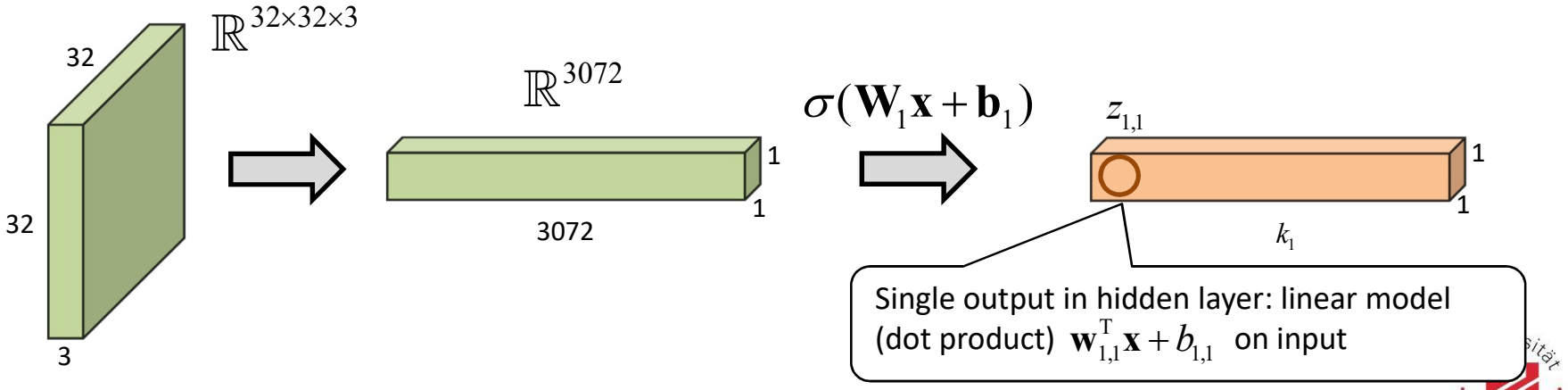
Multilayer Perceptron on Images

- First hidden layer within a multilayer perceptron: compute several linear models on the input (and pass through nonlinear activation)

$$\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \quad \mathbf{b}_1 \in \mathbb{R}^{k_1} \qquad \mathbf{z}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$



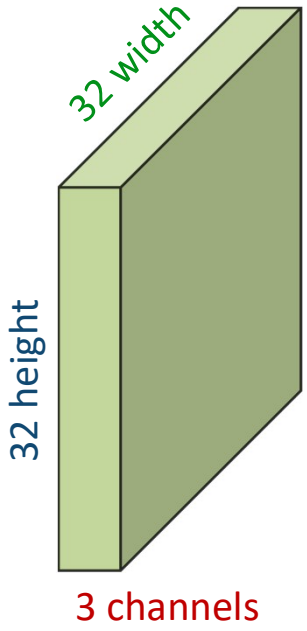
- With flattened image data:



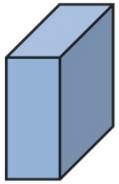
Convolution Operation

- **Convolution layer**: preserve spatial structure in image, apply linear models locally (so-called filters)

32 x 32 x 3 image



5 x 5 x 3 filter: 75 learnable weights

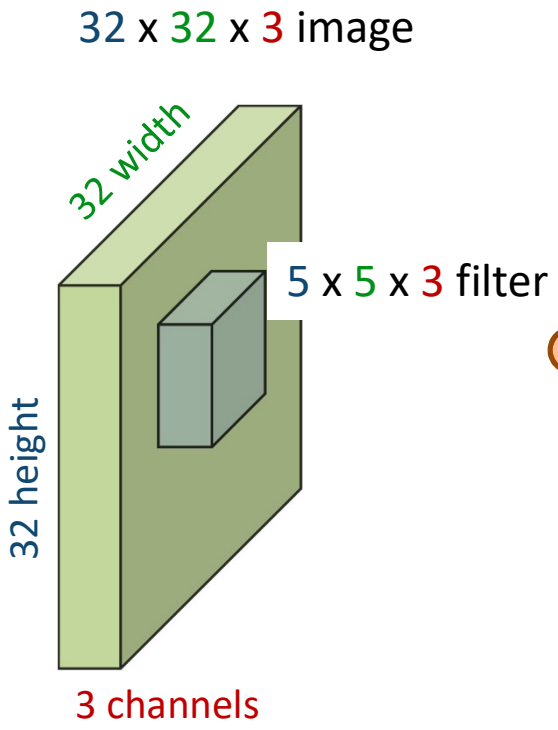


Compute the so-called **convolution** of the filter with the image: slide filter over image and locally compute dot product of filter and image

Filter is usually of squared shape, that is, *k* x *k* x *#channels*, where *k* is small (e.g. 3,5,7,11)

Convolution Layer

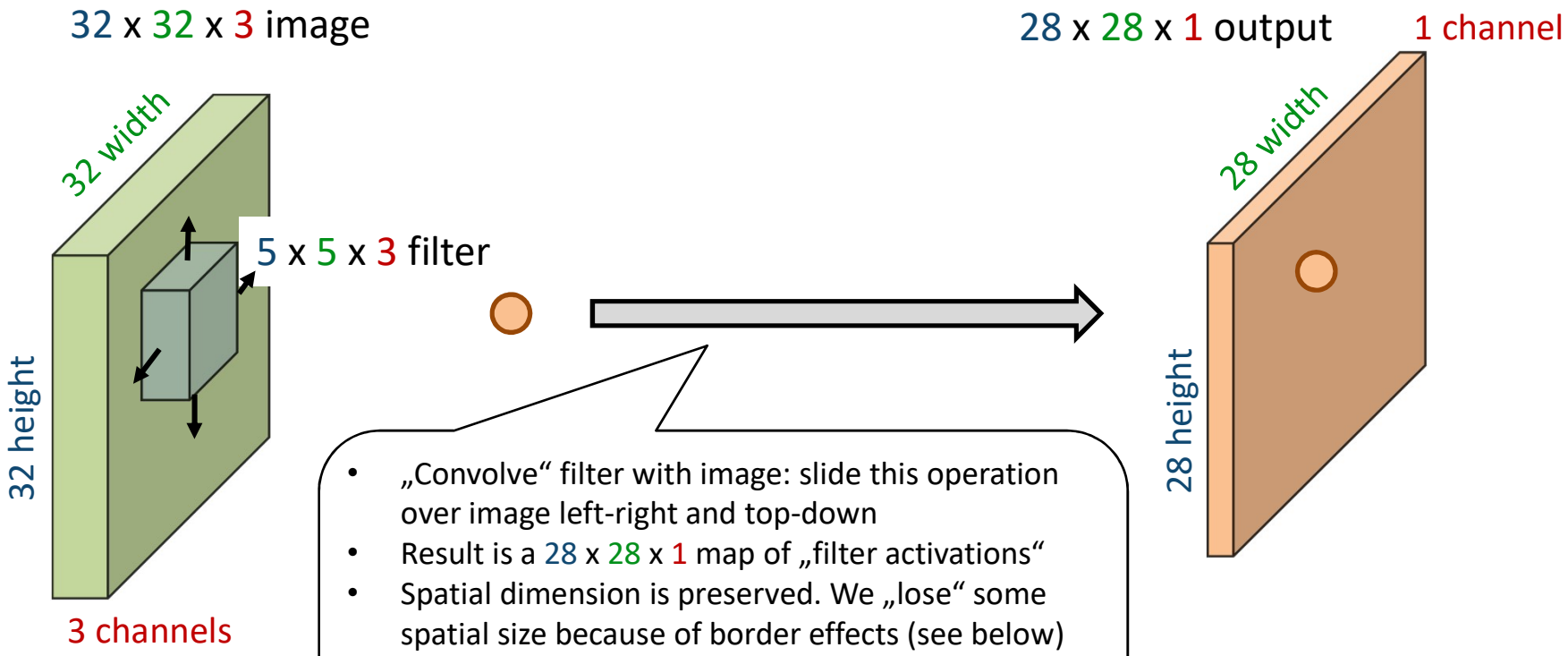
- **Convolution layer**: preserve spatial structure in image, apply linear models locally (so-called filters)



- Single output in convolution layer: dot product between filter and image.
- Filter consists of 5 x 5 x 3 = 75 numbers, each of these numbers is multiplied with the corresponding pixel value in a 5 x 5 x 3 3D-crop from the image, and result is added up
- The channel dimension (3 in example) must be identical for filter and image
- After forming dot product, we also add a scalar bias to result
- Output is $\mathbf{w}^{\mathrm{T}}\mathbf{x}+b$, $\mathbf{w}\in\mathbb{R}^{75}, b\in\mathbb{R}$, where $\mathbf{x}\in\mathbb{R}^{75}$ is the flattened 5 x 5 x 3 3D-crop from the image and $\mathbf{w}$ are the (flattened) trainable weights in the filter.

Convolution Layer

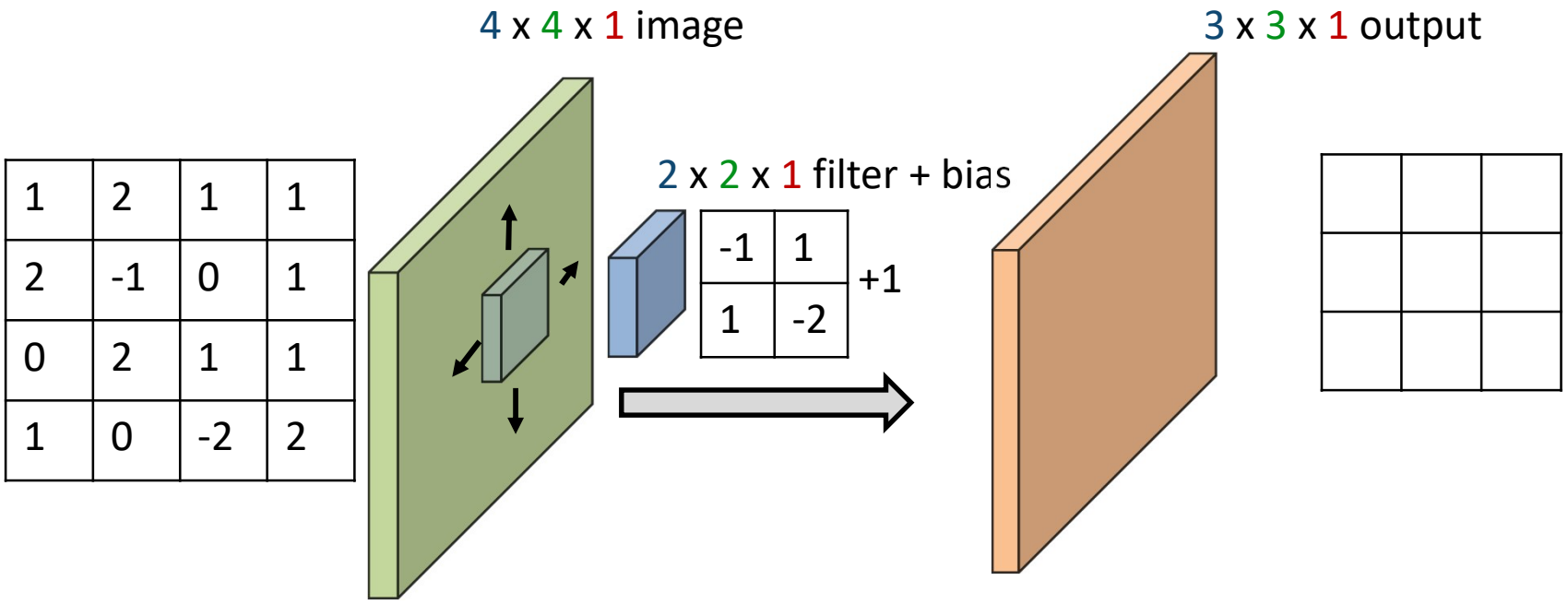
- **Convolution layer**: preserve spatial structure in image, apply linear models locally (so-called filters)



- „Convolve“ filter with image: slide this operation over image left-right and top-down
- Result is a 28 x 28 x 1 map of „filter activations“
- Spatial dimension is preserved. We „lose“ some spatial size because of border effects (see below)
- Channel dimension is not preserved
- The linear model $\mathbf{w}^{\mathrm{T}}\mathbf{x}+b$, $\mathbf{w}\in\mathbb{R}^{75}, b\in\mathbb{R}$ is the same for each output in the 28 x 28 x 1 map.

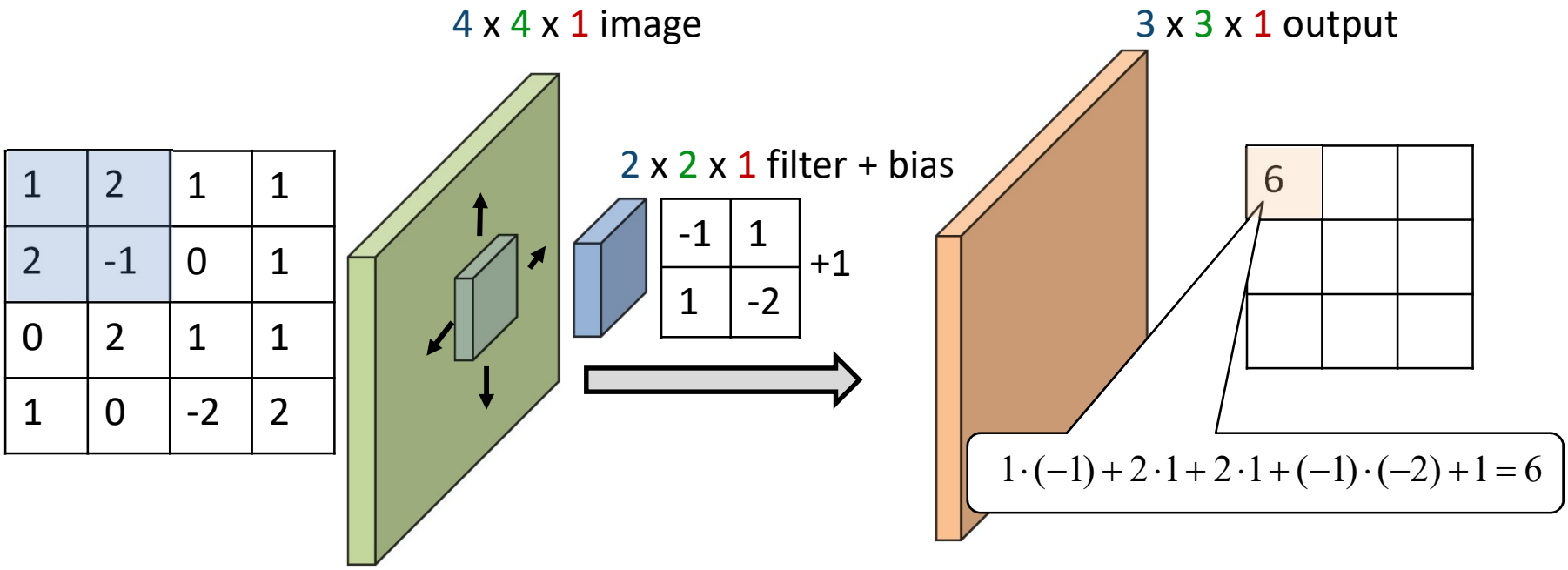
Convolution: Example

- Example for convolution operation (one input channel):



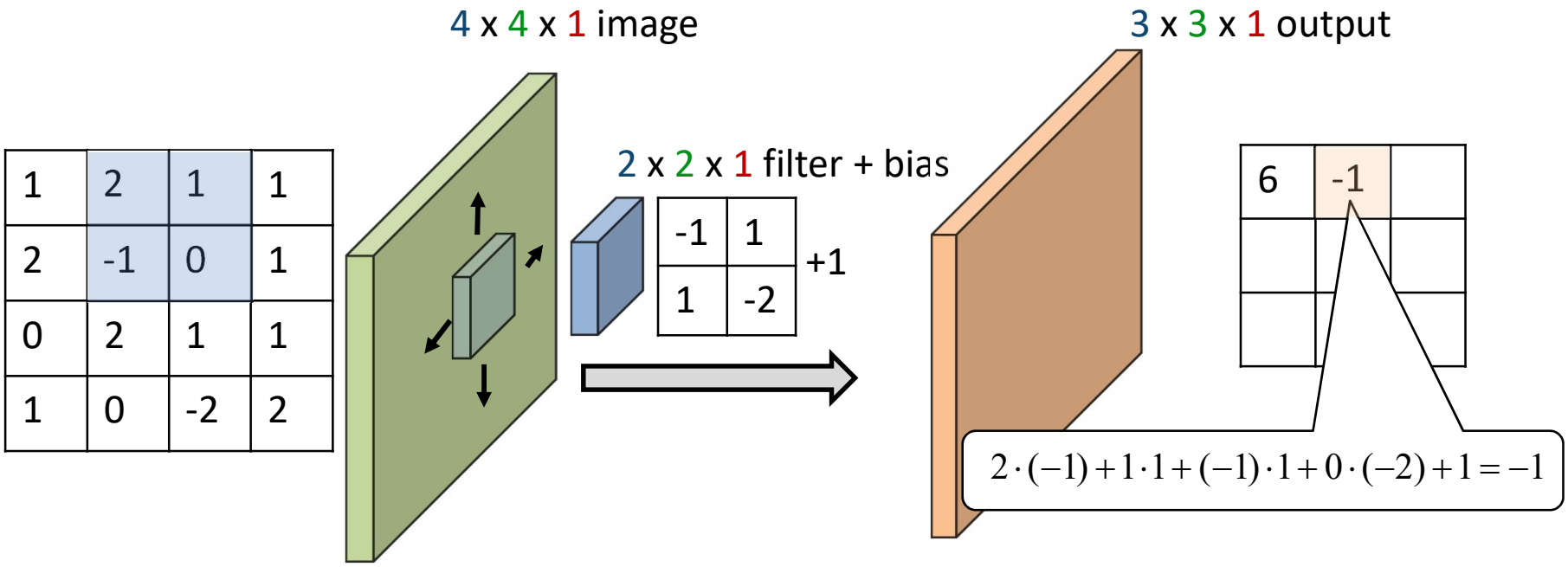
Convolution: Example

- Example for convolution operation (one input channel):



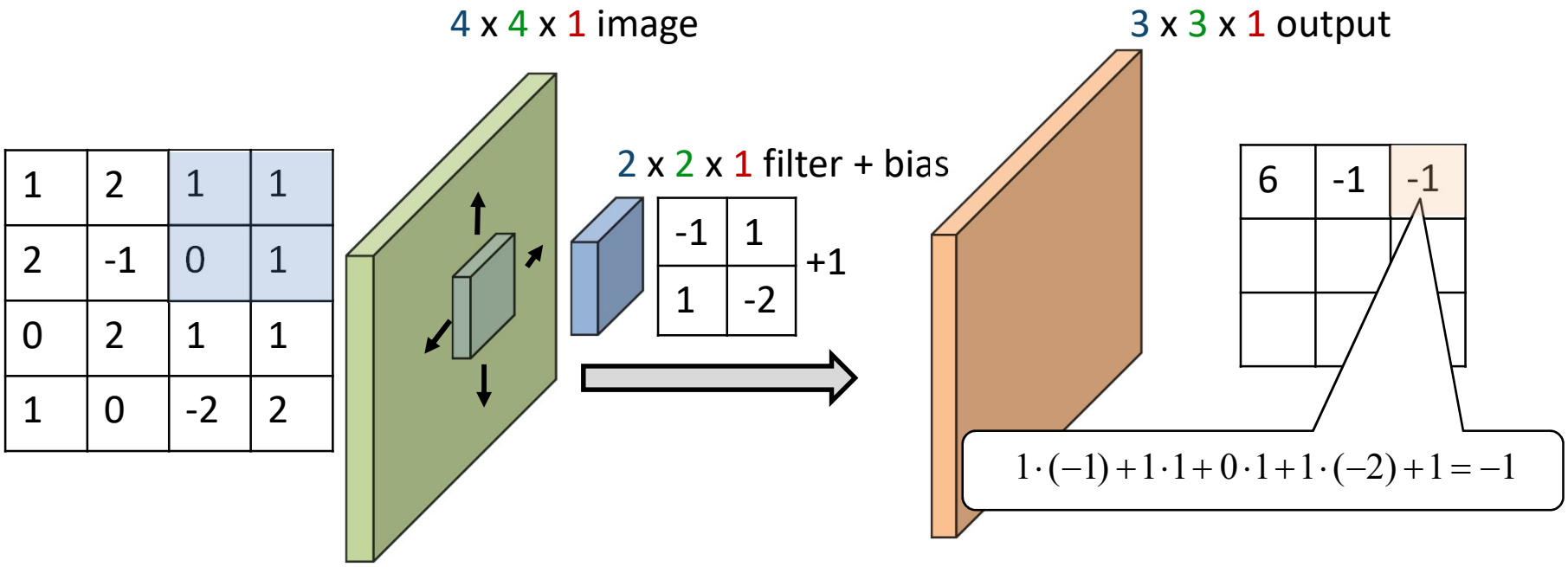
Convolution: Example

- Example for convolution operation (one input channel):



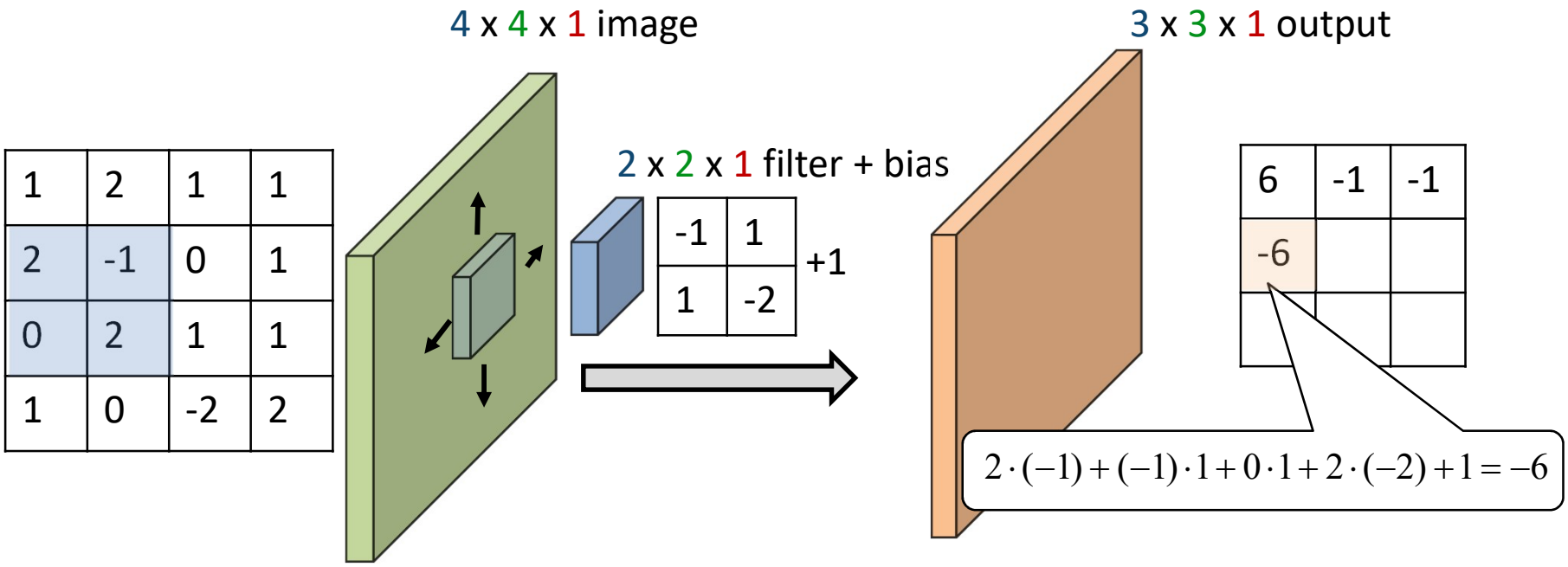
Convolution: Example

- Example for convolution operation (one input channel):



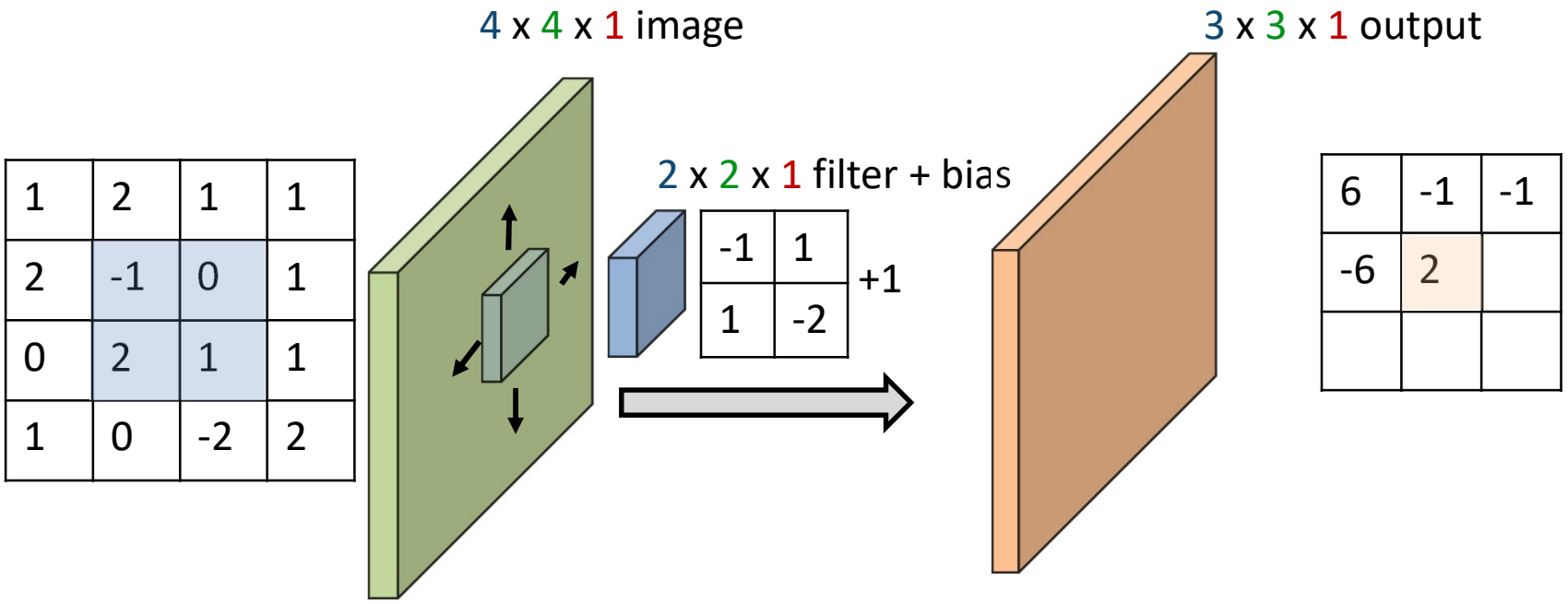
Convolution: Example

- Example for convolution operation (one input channel):



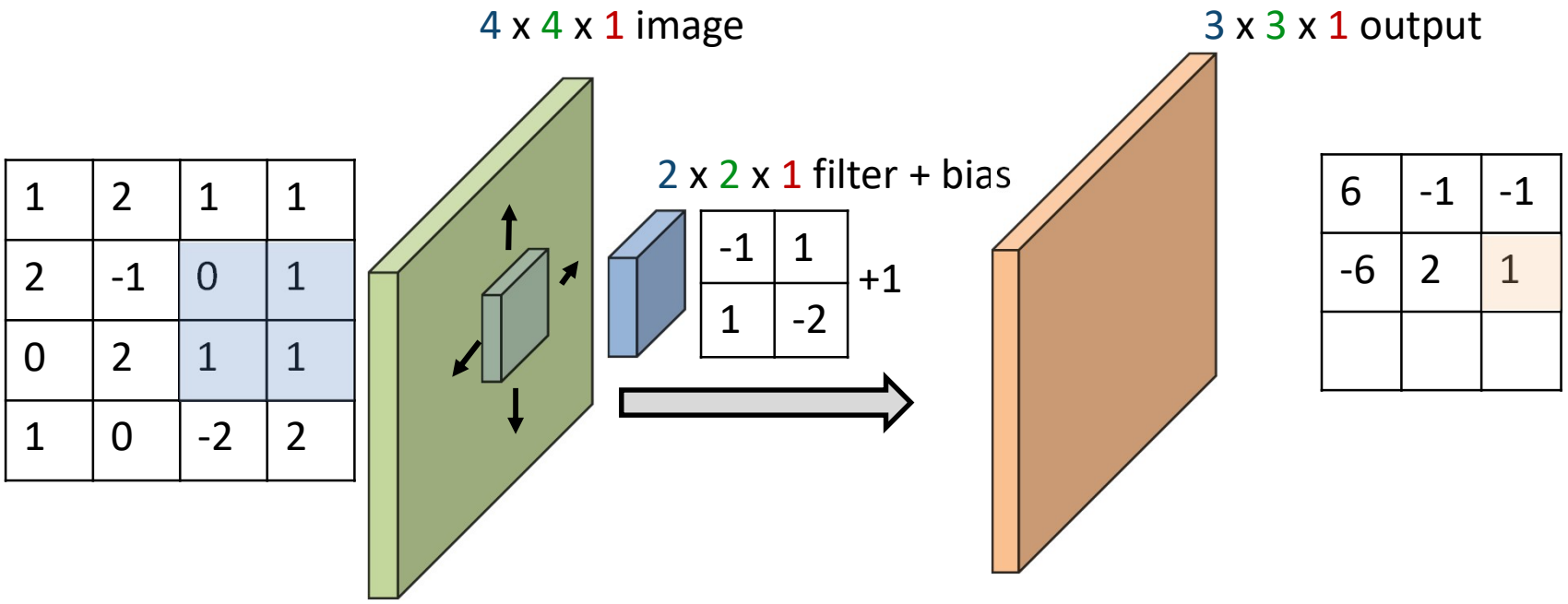
Convolution: Example

- Example for convolution operation (one input channel):



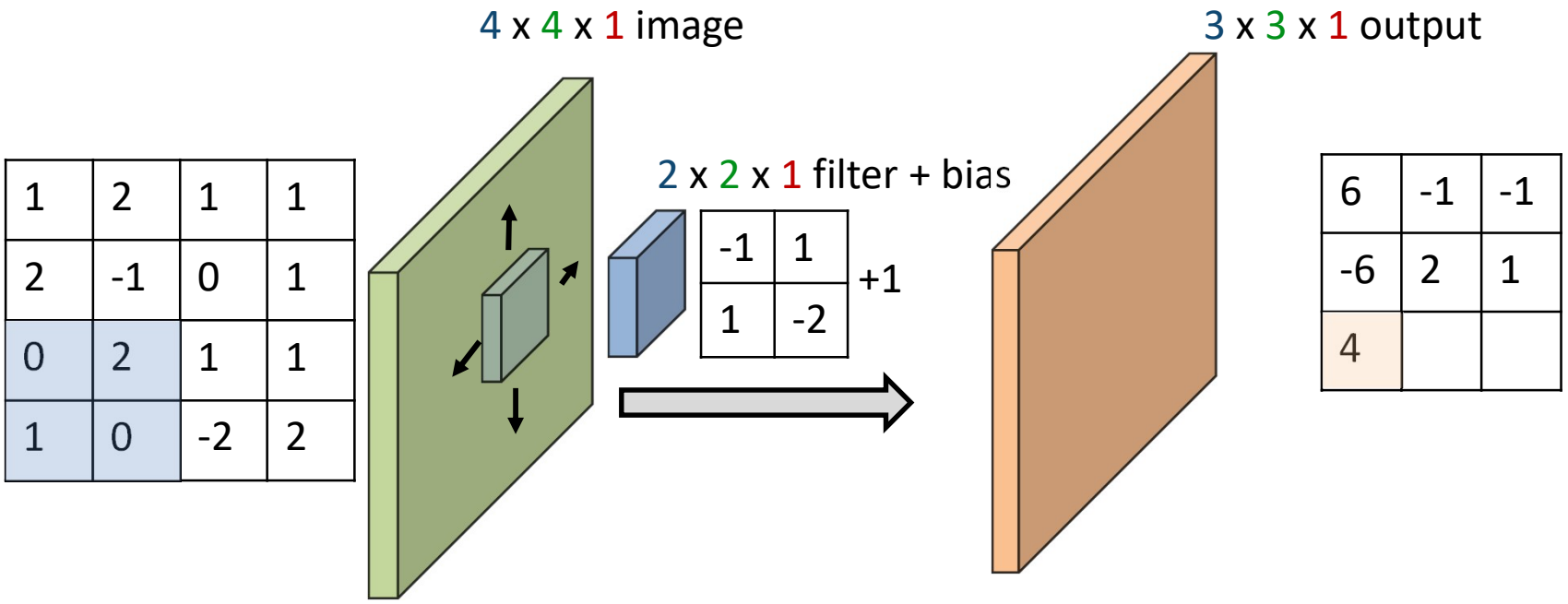
Convolution: Example

- Example for convolution operation (one input channel):



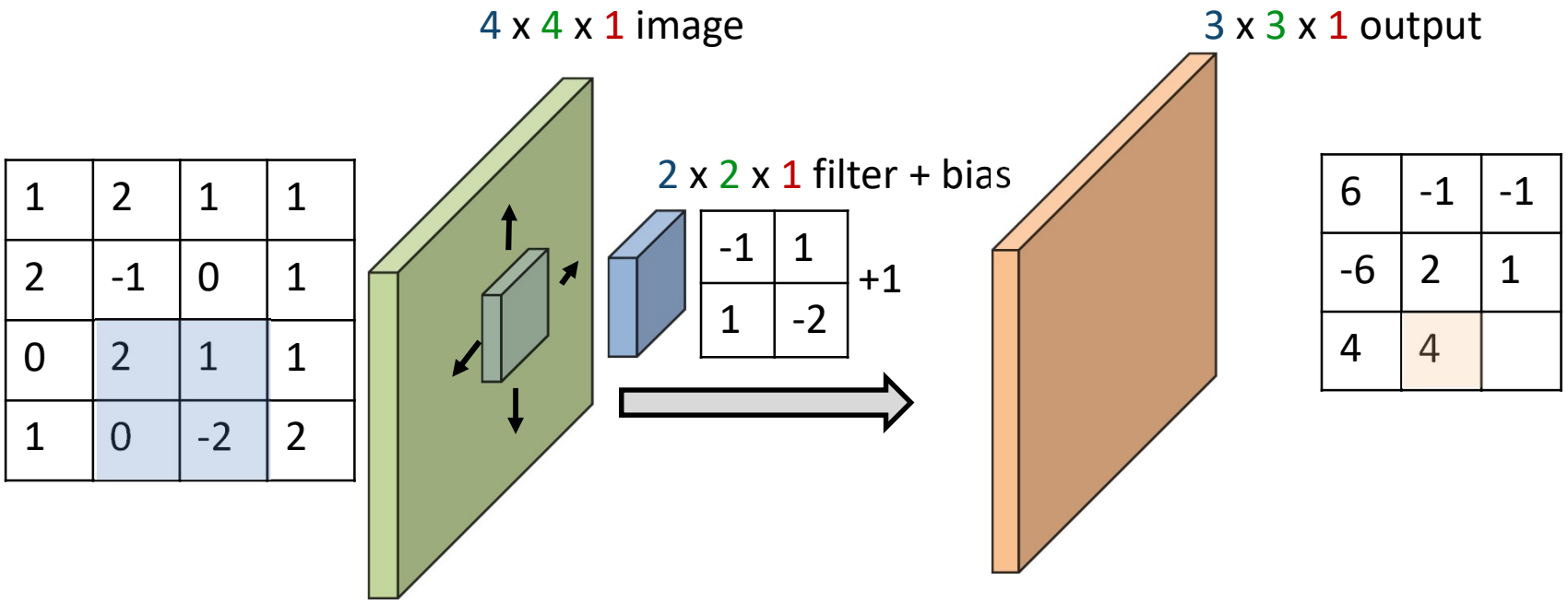
Convolution: Example

- Example for convolution operation (one input channel):



Convolution: Example

- Example for convolution operation (one input channel):



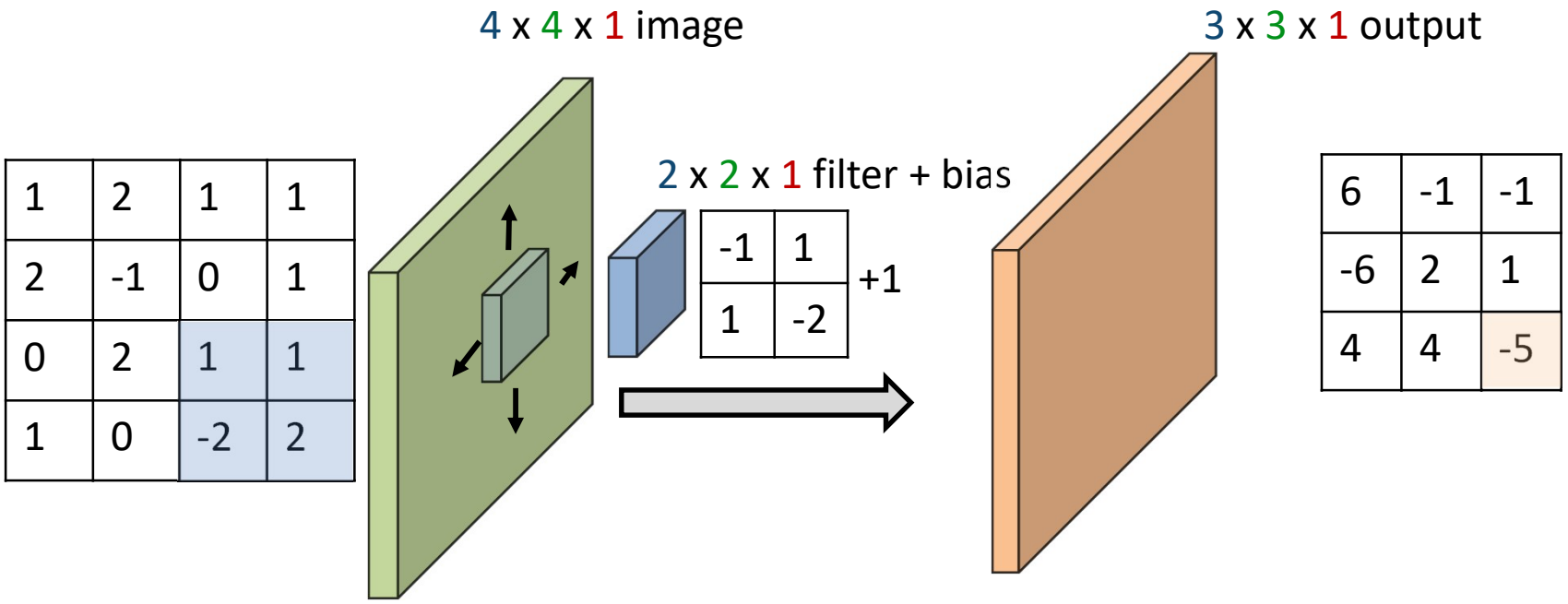
1	2	1	1
2	-1	0	1
0	2	1	1
1	0	-2	2

-1	1
1	-2

6	-1	-1
-6	2	1
4	4	

Convolution: Example

- Example for convolution operation (one input channel):



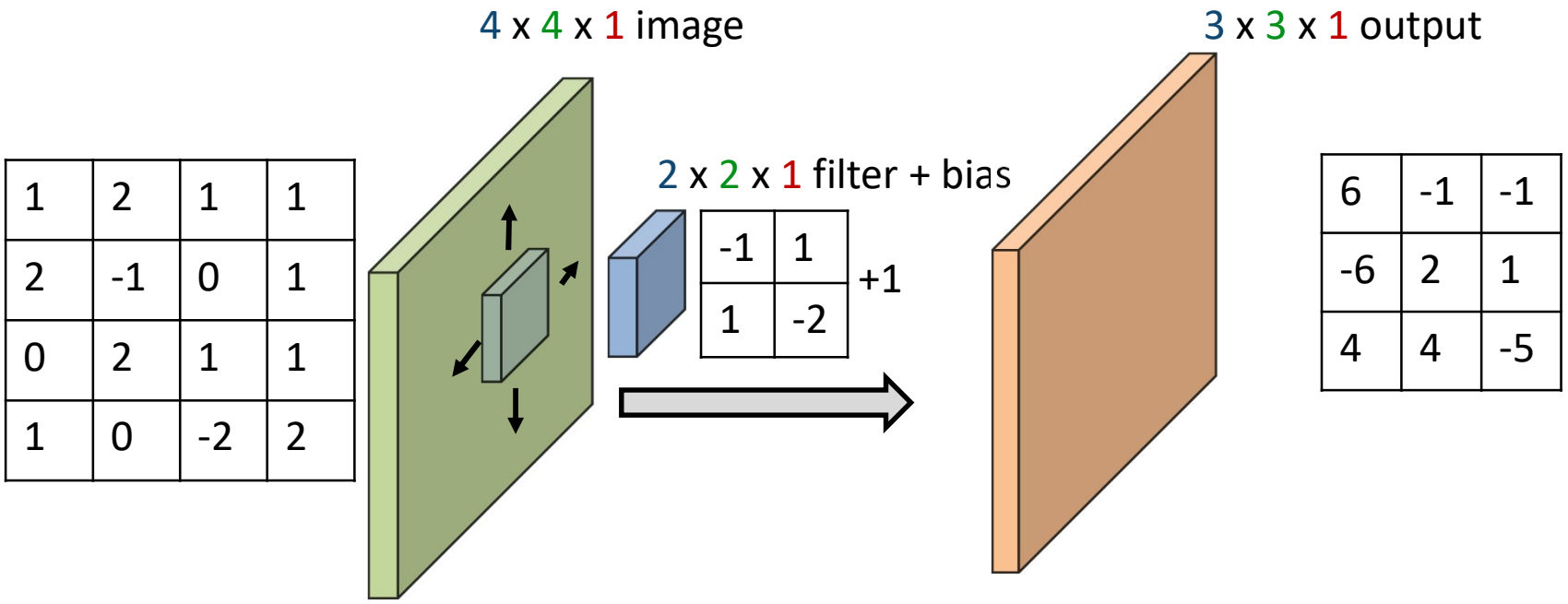
1	2	1	1
2	-1	0	1
0	2	1	1
1	0	-2	2

-1	1
1	-2

6	-1	-1
-6	2	1
4	4	-5

Convolution: Example

- Example for convolution operation (one input channel):



1	2	1	1
2	-1	0	1
0	2	1	1
1	0	-2	2

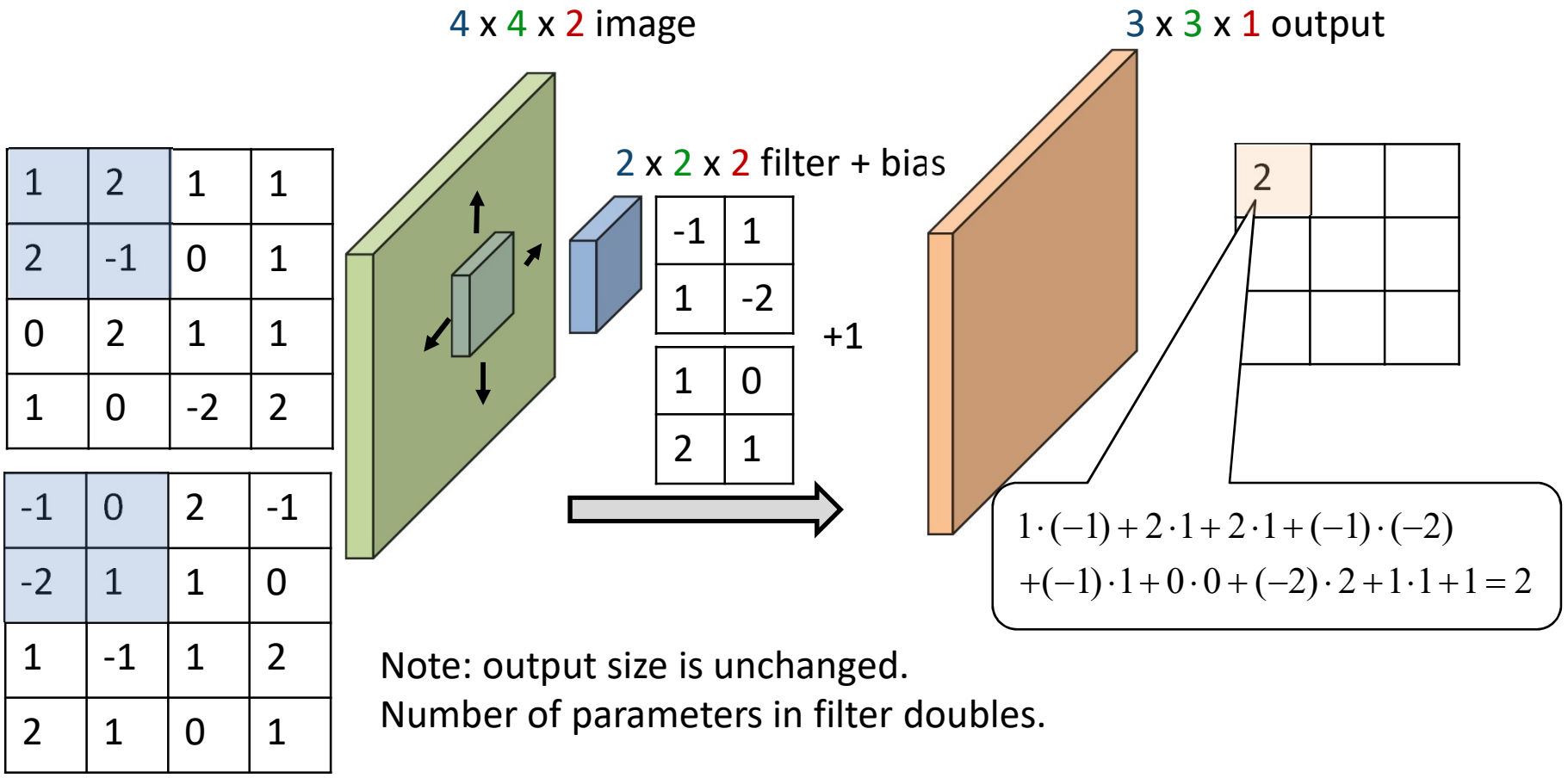
-1	1
1	-2

6	-1	-1
-6	2	1
4	4	-5

- Observation: due to border effects, the output width and height is going to be smaller by *k*-1 than the input size, for a *k* x *k* x *#channels* filter
- Channel dimension of output is one (so far, extension see below)

Convolution: Example

- Example for convolution operation with 2 input channels



Note: output size is unchanged.
Number of parameters in filter doubles.

Convolution: Example

- Observation: due to border effects, the output width and height is going to be smaller by *k*-1 than the input size, for a *k* x *k* x *#channels* filter
- This introduces the complication that when stacking many convolution layers (see below), output size can shrink a lot and depend on network depth
- Often avoided by **padding** input image with (*k*-1)/2 zeros (*k* usually uneven)

4 x 4 x 1 image padded with zeros

0	0	0	0	0	0
0	1	2	1	1	0
0	2	-1	0	1	0
0	0	2	1	1	0
0	1	0	-2	2	0
0	0	0	0	0	0

3 x 3 x 1 filter + bias

1	-1	1
2	0	-1
-2	2	1

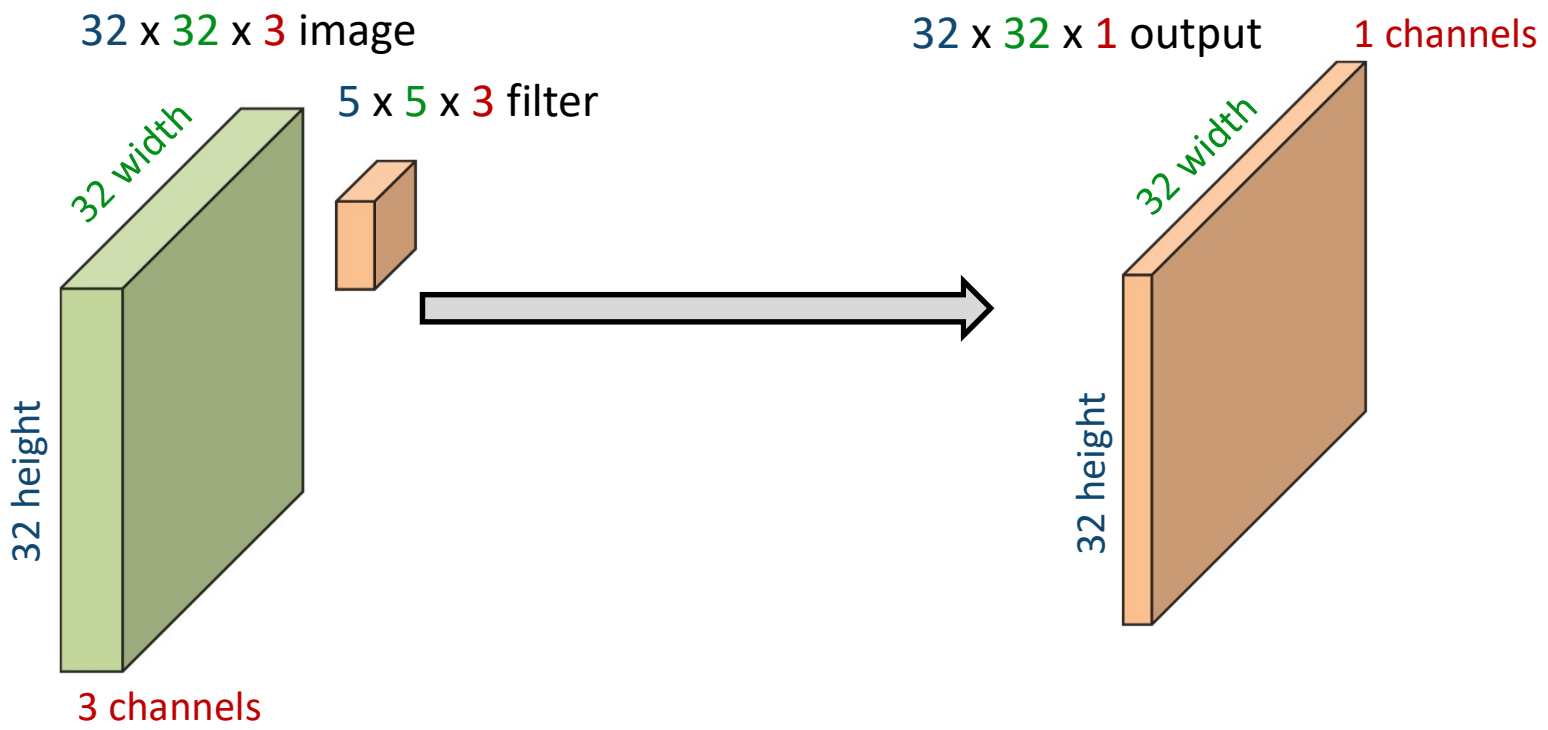
+1

4 x 4 x 1 output

2	...		
...	...		

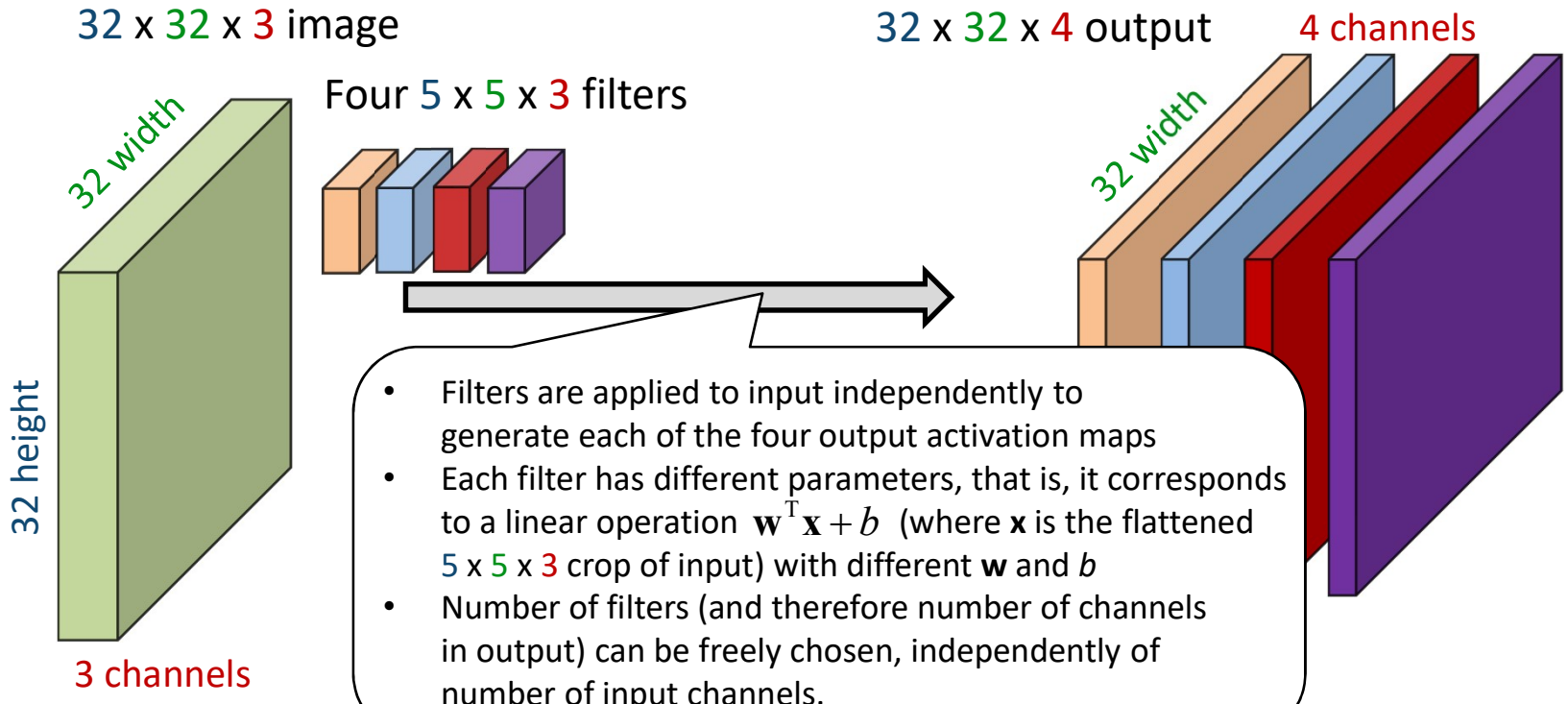
Convolution Layer

- Convolution layer so far: map *m* x *l* x *d* image (e.g. 32 x 32 x 3) onto *m* x *l* x *1* activation map (using appropriate padding)



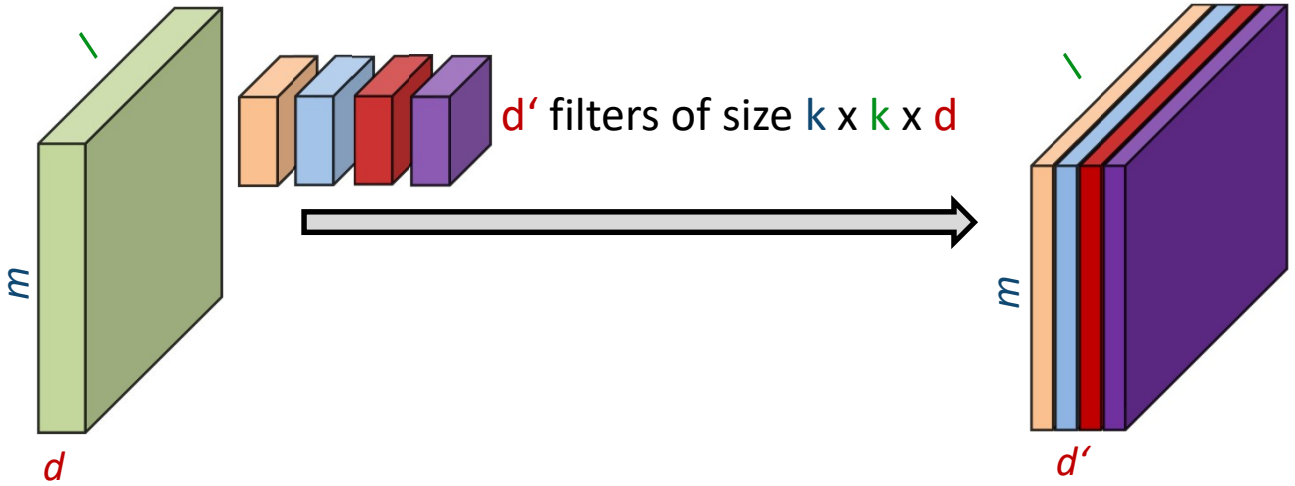
Convolution Layer

- Convolution layer so far: map *m* x *l* x *d* image (e.g. 32 x 32 x 3) onto *m* x *l* x *1* activation map (using appropriate padding)
- **Idea**: use more than one filter, different filters correspond to different output channels



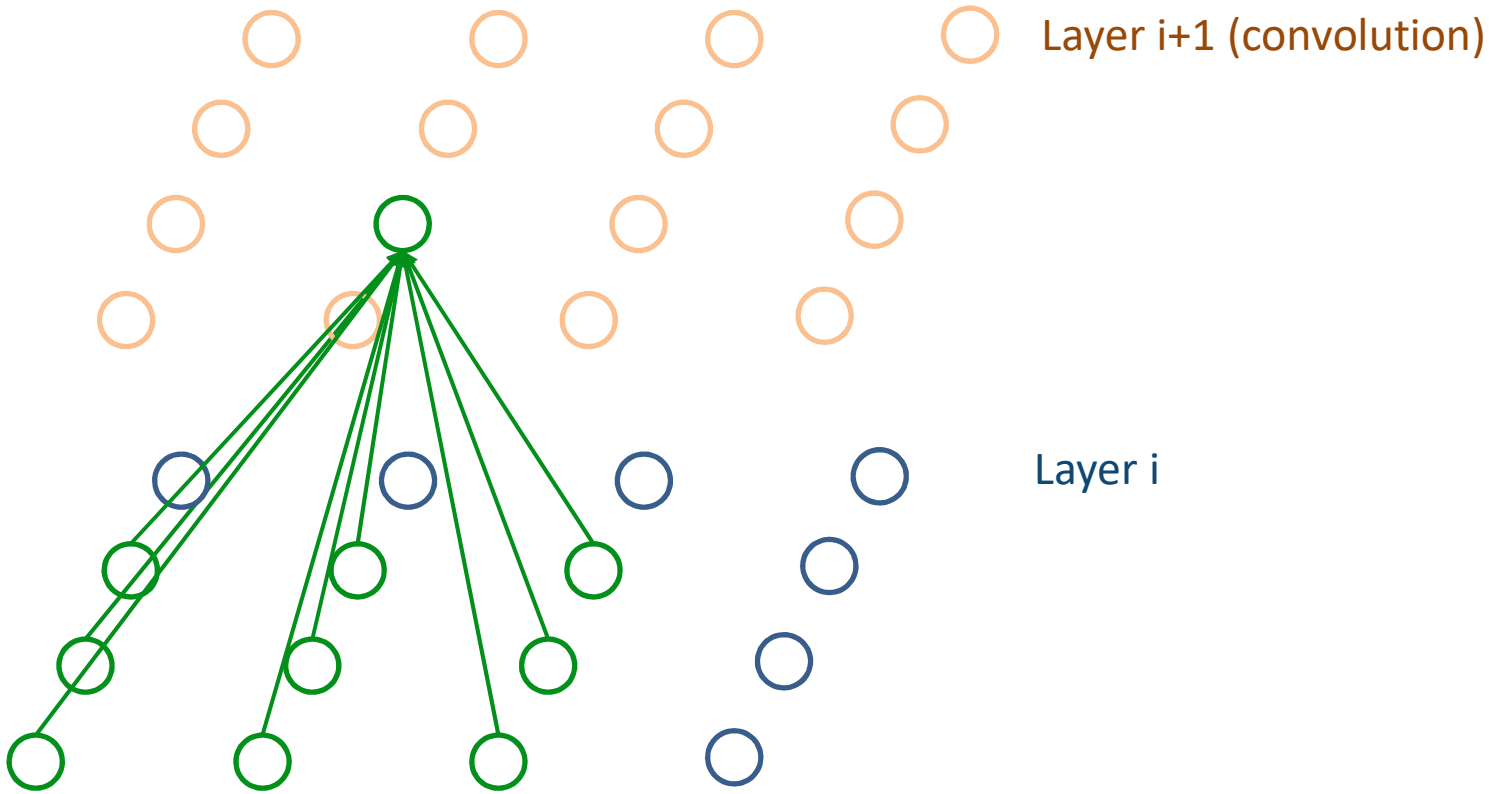
Convolution Layers Input and Output

- **Summary**: Input to a convolution layer is *m* x *l* x *d* 3D-tensor, e.g. input RGB-image of height *m*, width *l*, and with *d*=3 color channels.
- To define the convolution layer, we have to choose
  - the size of the filter: typically squared, that is, *k* x *k* x *d*. Note that *d* is determined by input. The parameter *k* can be chosen, is called kernel size
  - the number of filters *d'*, which will become number of output channels
  - whether to use padding or not (usually yes)
- Output of convolution layer will be *m* x *l* x *d'* 3D-tensor (if padding is used)



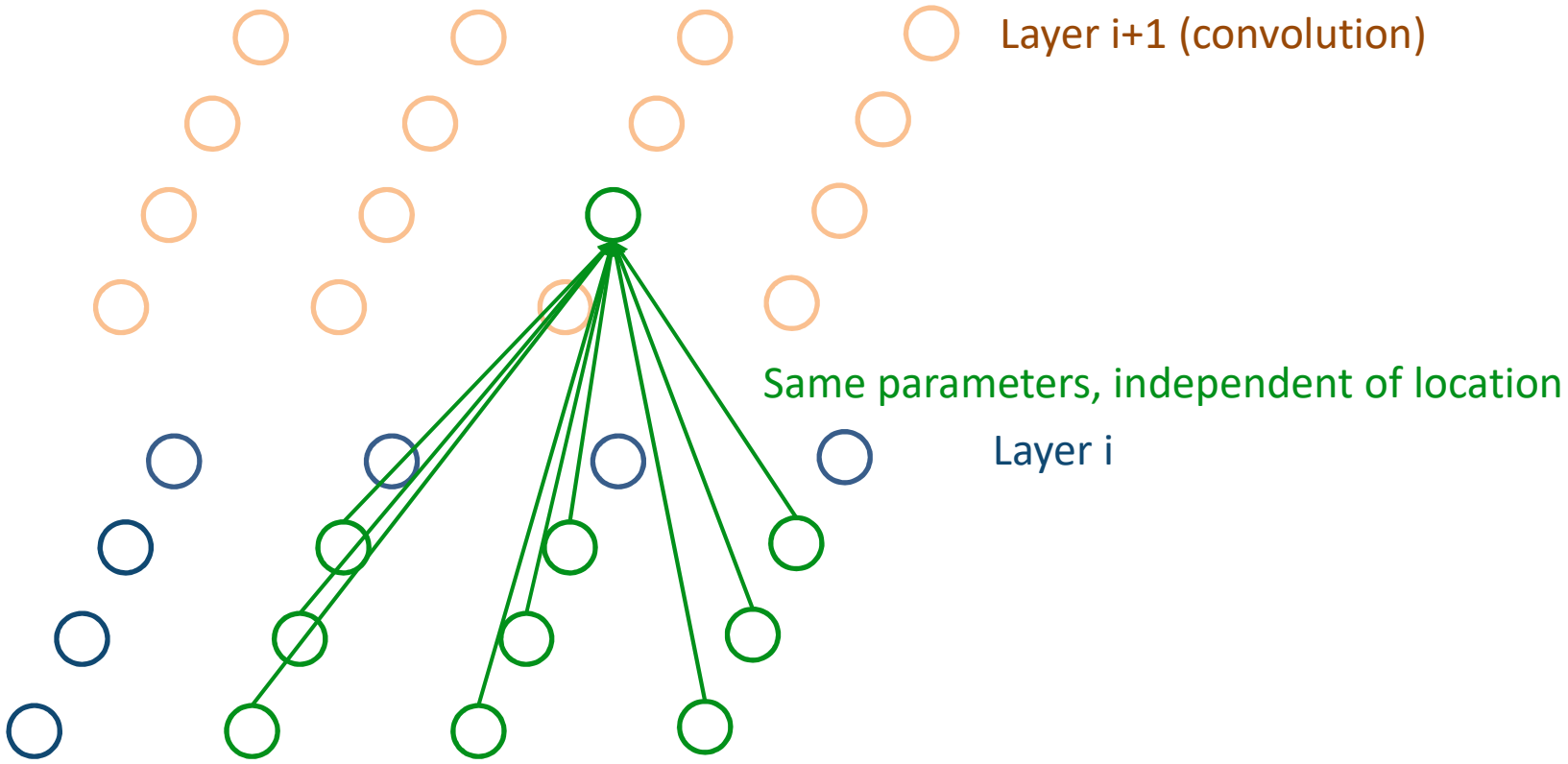
Convolution Layers: Graph Representation

- Graph representation (3 x 3 x 1 filter): local connectivity and parameter sharing



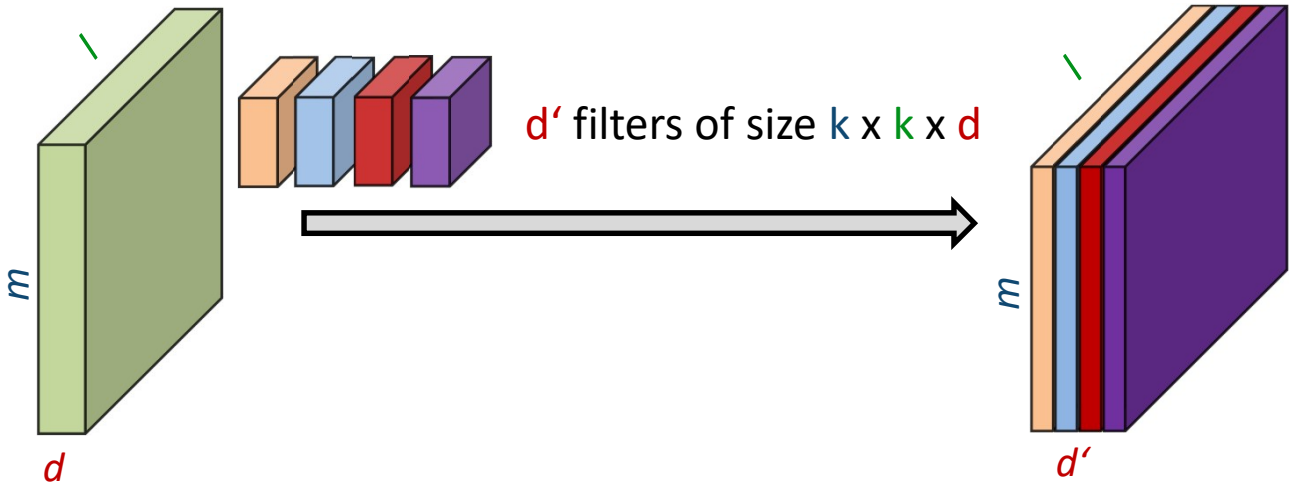
Convolution Layers: Graph Representation

- Graph representation (3 x 3 x 1 filter): local connectivity and parameter sharing



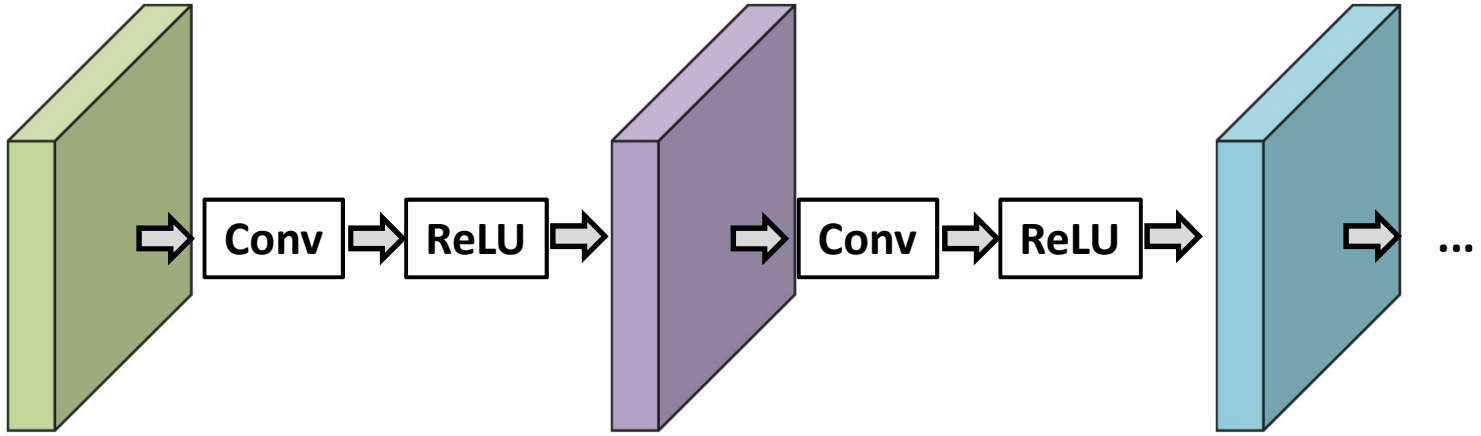
Convolution Layers: Parameters and Compute Operations

- How many model parameters are in a convolution layer?
  - Single filter of size *k* x *k* x *d*: $k^2 d + 1$ parameters
  - Complete convolution layer: $d' \cdot (k^2 d + 1)$
- How many compute operations, assuming *m* x *l* x *d* input layer?
  - Each output is $k^2 d + 1$ multiply-add operations
  - There are *m* x *l* x *d'* outputs, therefore overall $m \cdot l \cdot d' (k^2 d + 1)$ multiply-add operations



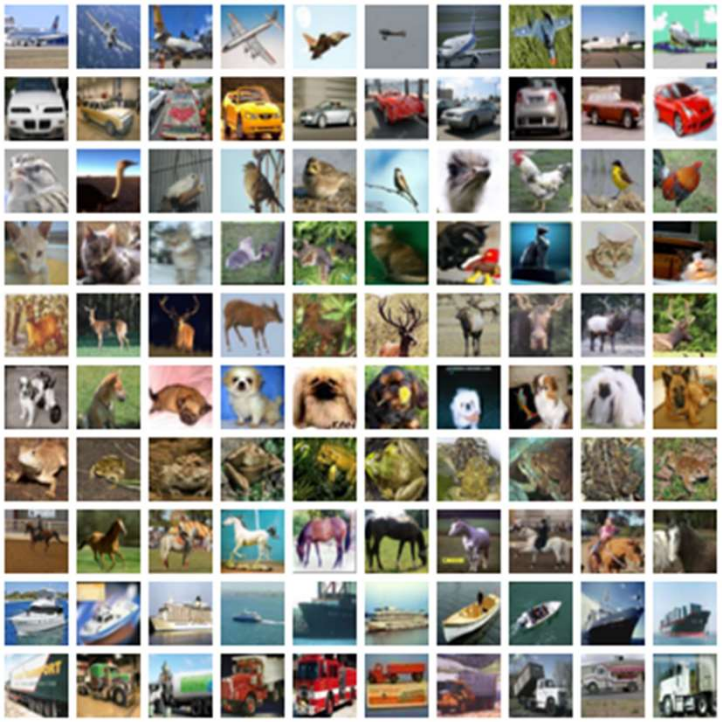
Stacking Convolution Layers

- Convolutional layers are typically stacked similarly as fully connected layers are stacked in a multilayer perceptron, with nonlinear activations (ReLU, tanH) in between



Example: Cifar-10 Data Set

- Example: Cifar-10 data set



airplane
car
bird
cat
deer
dog
frog
horse
ship
truck

10-class image classification

32 x 32 x 3 color images

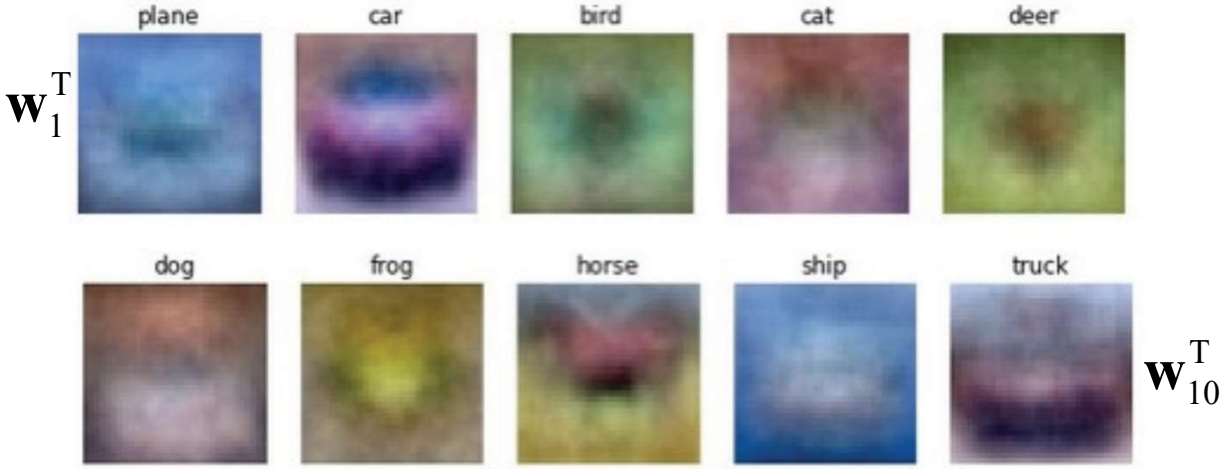
6000 images per class

What Do Linear Models Learn?

- What would a linear classifier learn on Cifar-10?
- Flatten 32 x 32 x 3 images to 3072-dim vector, linear model on this vector

class scores $f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^{10} \quad \mathbf{W} \in \mathbb{R}^{10\times 3072}, \quad \mathbf{b} \in \mathbb{R}^{10}$

- Score for class *i*: $\mathbf{w}_i^{\mathrm{T}}\mathbf{x} + b_i$, with $\mathbf{w}_i^{\mathrm{T}}$ i-th row of $\mathbf{W}$ and b_i i-th element of $\mathbf{b}$
- Learns one „template“ per class: visualize by reshaping $\mathbf{w}_i^{\mathrm{T}}$ back to 32 x 32 x 3 image.



Class score will be high if input image matches template (dot product is a similarity measure)

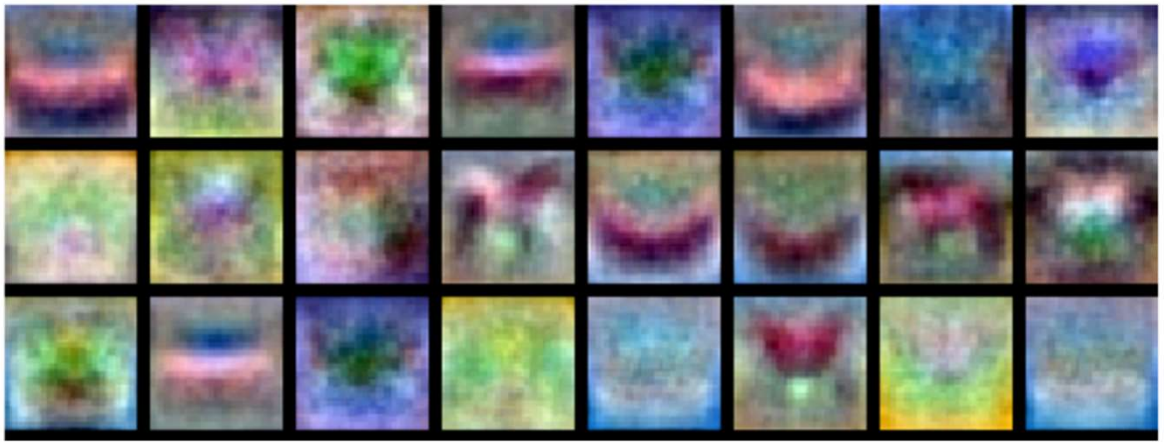
What Do Multilayer Perceptrons Learn?

- What would the first fully connected layer in a multilayer perceptron learn?
- Flatten 32 x 32 x 3 images to 3072-dim vector, each hidden unit in first layer learns a linear model on this vector (and passes through activation function)

activations of hidden units

$$\sigma(\mathbf{W}\mathbf{x}+\mathbf{b}) \in \mathbb{R}^{k_1} \qquad \mathbf{W} \in \mathbb{R}^{k_1 \times 3072}, \quad \mathbf{b} \in \mathbb{R}^{k_1}$$

- Each hidden unit learns a full-image template that serves as feature for subsequent layers

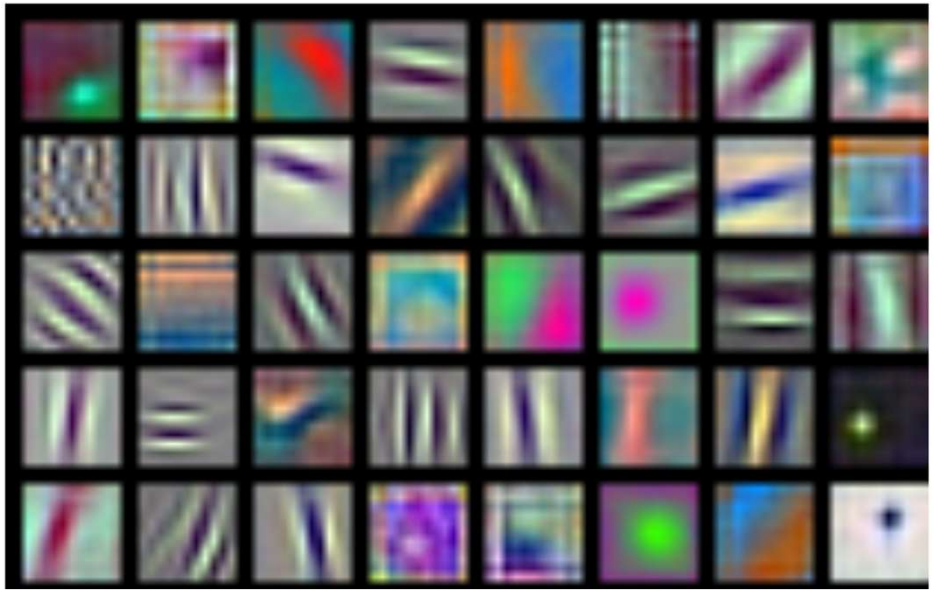


Activation will be high if input image matches template (dot product is a similarity measure)

What Do Convolutional Networks Learn?

- What would the first convolution layer in a convolutional network learn?
- Example: 11 x 11 x 3 filter in „AlexNet“ architecture
  - each filter learns a local template or pattern that can match a 11 x 11 x 3 crop from the input image
  - the 11*11*3 = 363 learnable parameters in the filter can be visualized as 11 x 11 x 3 images

$\mathbf{w}_1$



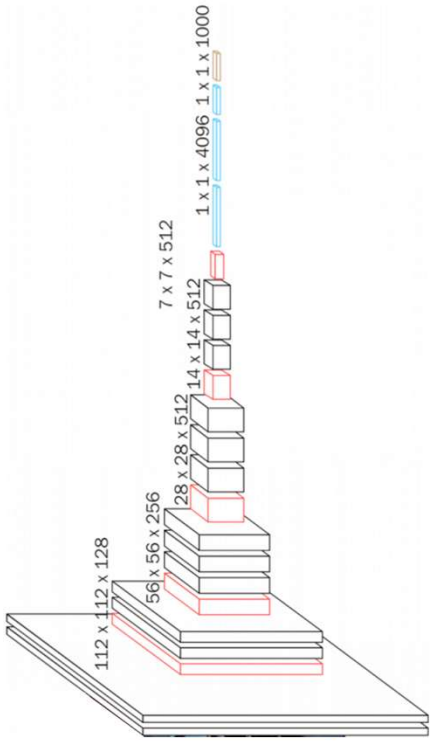
$\mathbf{w}_{d'}$

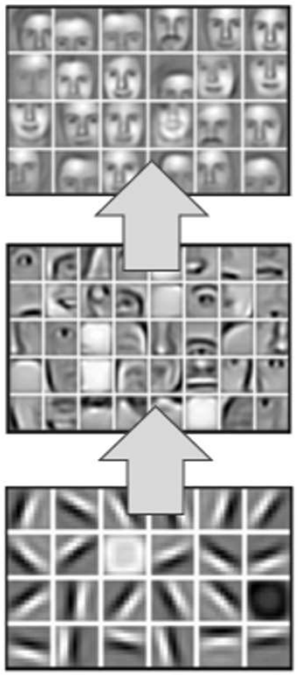
- Activation at location will be high if crop from input image matches
- Often learns oriented edges or other simple shapes
- E.g. filter will activate if horizontal edges appear in image
- Different channels in activation map encode different such simple shapes/features and where they occur

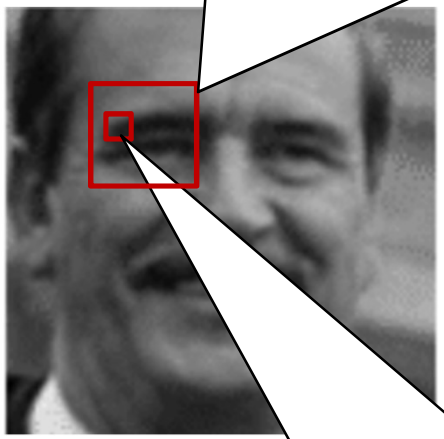
Pooling Layers

- Recap: with neural networks, we want to learn hierarchies of increasingly abstract features

Example: Face recognition with deep convolutional neural network





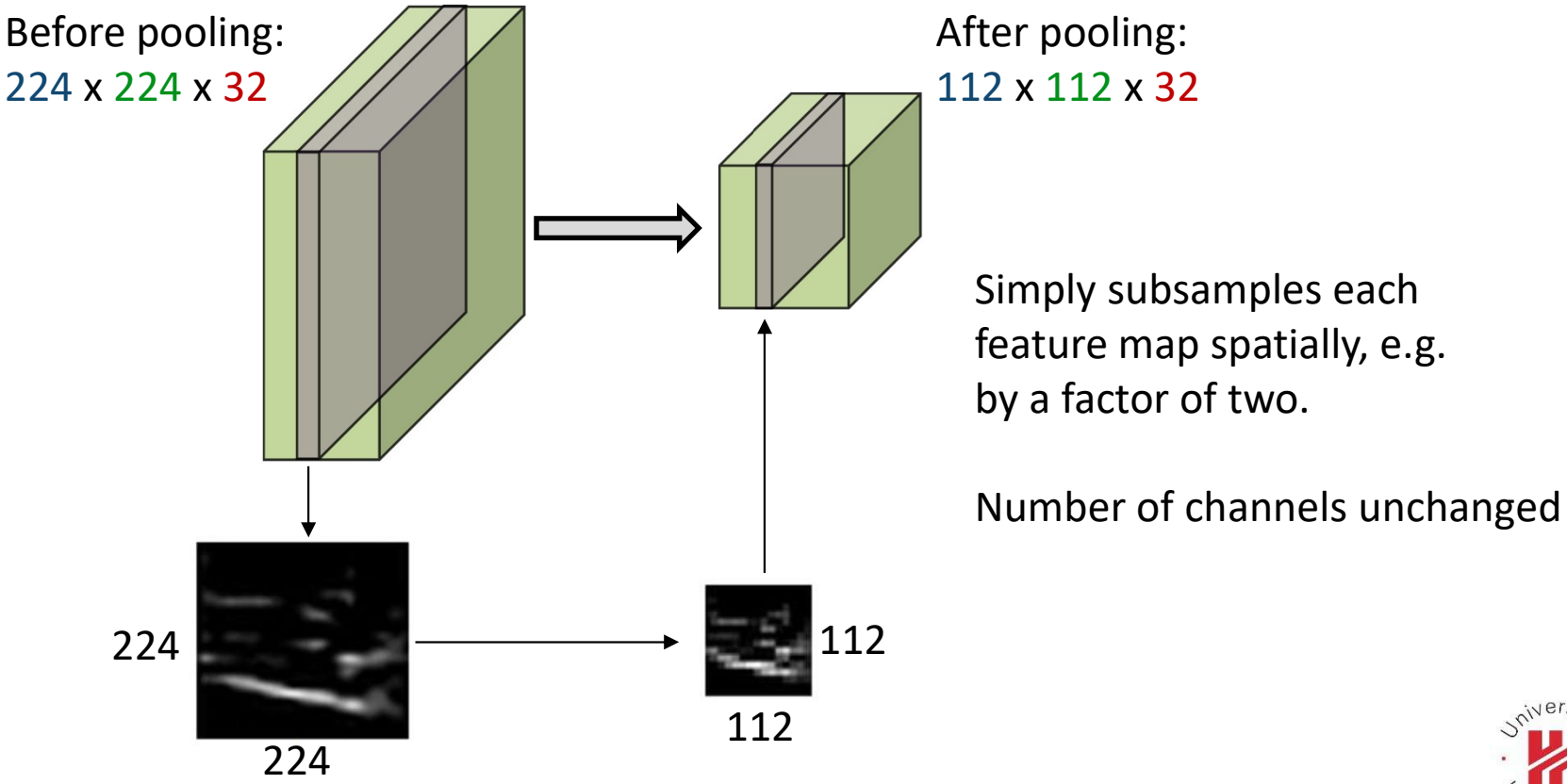


High-level layers should extract more complex and abstract features. Spatially, these need to cover larger parts of the image, and need to be build up from the simpler features.

Low level layers extract simple features such as edges with convolution filters from input

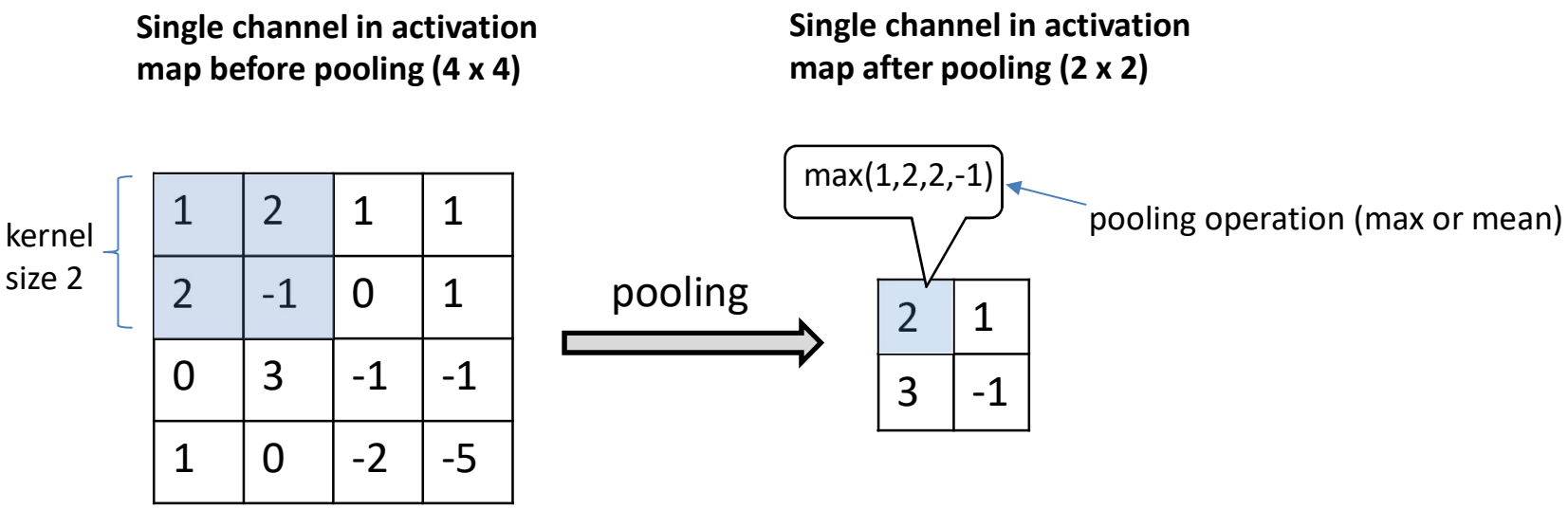
Pooling Layers

- Pooling layers: layer type within convolutional neural networks that spatially subsamples the current activation map, in order to enable learning more abstract features at higher layers



Pooling Layers

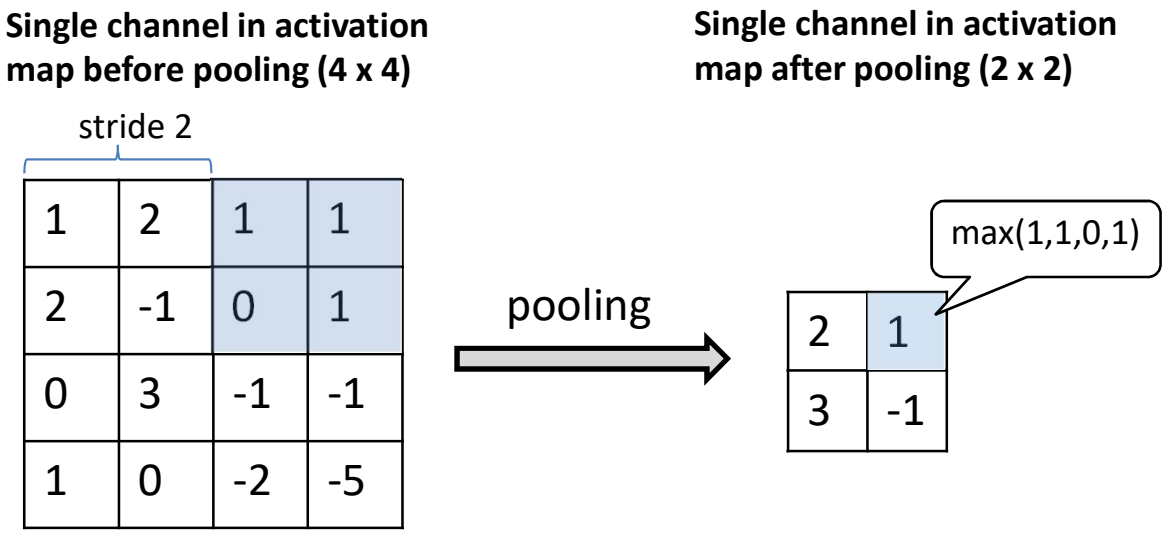
- Example: max-pooling with kernel size 2x2, stride 2



- Pooling algorithm in general (perform on all channels independently):
  - Look at a crop of size <kernel size> x <kernel size> in input channel
  - Apply pooling operation to values (usually, max or mean) to get output
  - Move crop area by <stride> in input, repeat to get further outputs

Pooling Layers

- Example: max-pooling with kernel size 2x2, stride 2



- Pooling algorithm in general (perform on all channels independently):
  - Look at a crop of size <kernel size> x <kernel size> in input channel
  - Apply pooling operation to values (usually, max or mean) to get output
  - Move crop area by <stride> in input, repeat to get further outputs

Pooling Layers

- Example: max-pooling with kernel size 2x2, stride 2

Single channel in activation map before pooling (4 x 4)

1	2	1	1
2	-1	0	1
0	3	-1	-1
1	0	-2	-5

pooling

Single channel in activation map after pooling (2 x 2)

2	1
3	-1

- Pooling algorithm in general (perform on all channels independently):
  - Look at a crop of size <kernel size> x <kernel size> in input channel
  - Apply pooling operation to values (usually, max or mean) to get output
  - Move crop area by <stride> in input, repeat to get further outputs

Pooling Layers

- Example: max-pooling with kernel size 2x2, stride 2

Single channel in activation map before pooling (4 x 4)

1	2	1	1
2	-1	0	1
0	3	-1	-1
1	0	-2	-5

pooling

Single channel in activation map after pooling (2 x 2)

2	1
3	-1

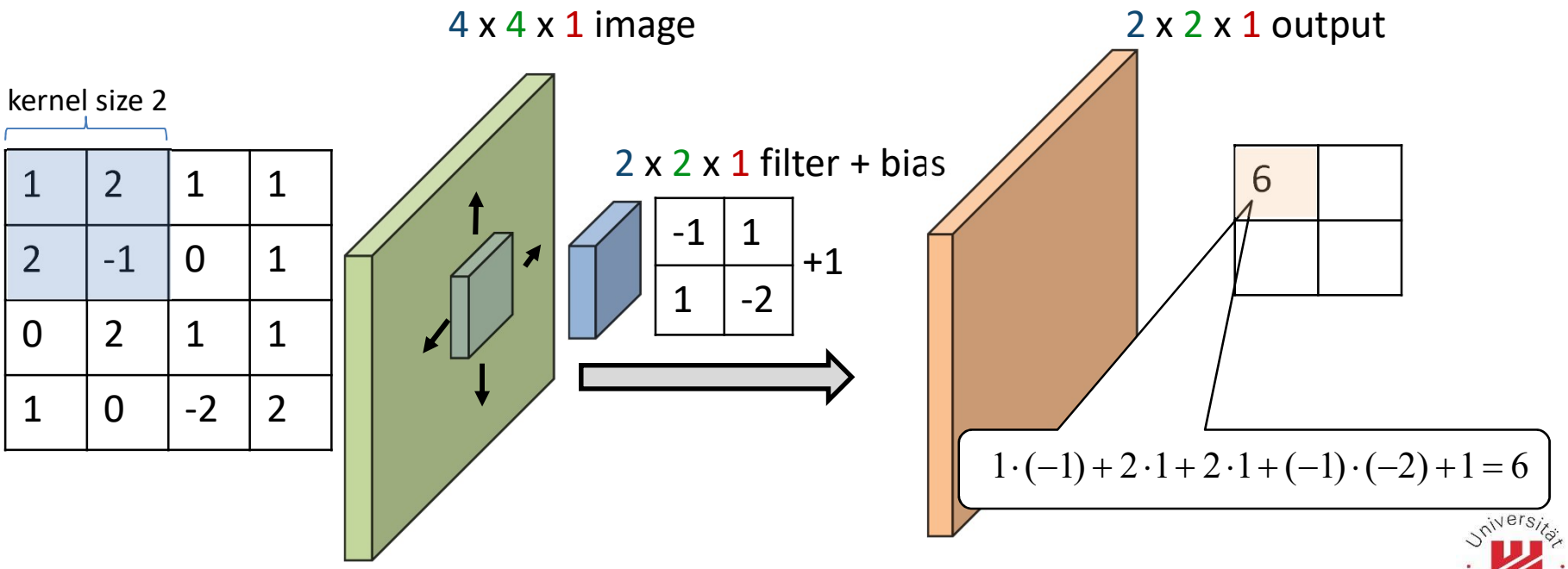
For kernel size 2x2, stride 2, halves the spatial size of the output

- Pooling algorithm in general (perform on all channels independently):
  - Look at a crop of size <kernel size> x <kernel size> in input channel
  - Apply pooling operation to values (usually, max or mean) to get output
  - Move crop area by <stride> in input, repeat to get further outputs

Alternative to Pooling: Strided Convolutions

- Instead of pooling layer, we can also use so-called strided convolution layers
- **Idea**: Move the filter by more than one unit in x/y direction in convolution
- Also performs spatial subsampling

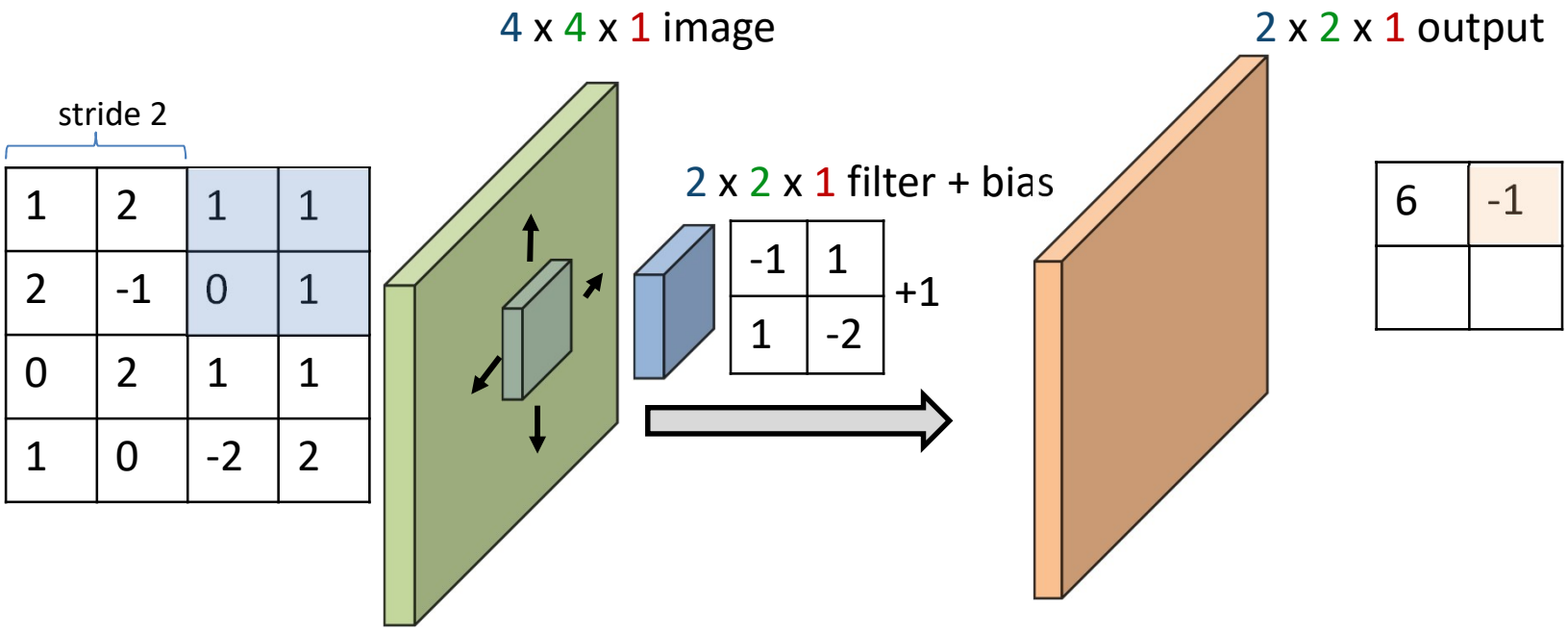
Example: convolution with 2 x 2 kernel and stride 2



Alternative to Pooling: Strided Convolutions

- Instead of pooling layer, we can also use so-called strided convolution layers
- **Idea**: Move the filter by more than one unit in x/y direction
- Also performs spatial subsampling

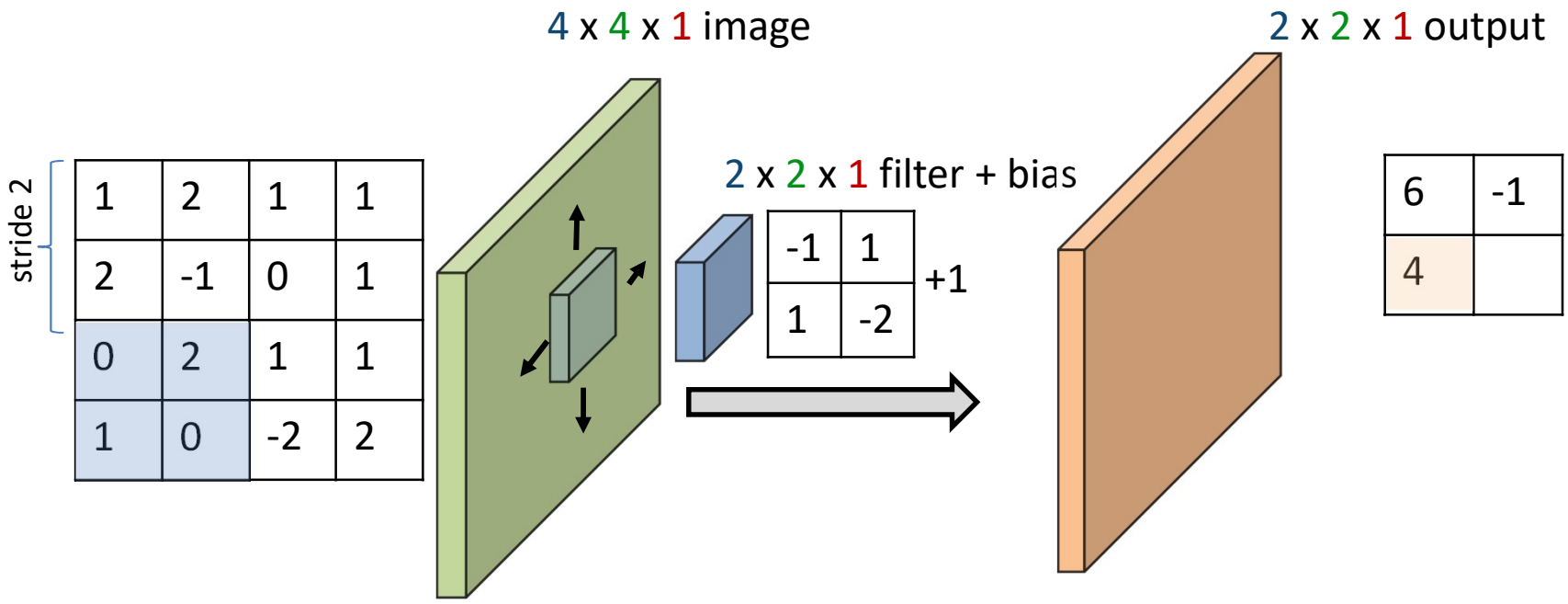
Example: convolution with 2 x 2 kernel and stride 2



Alternative to Pooling: Strided Convolutions

- Instead of pooling layer, we can also use so-called strided convolution layers
- **Idea**: Move the filter by more than one unit in x/y direction
- Also performs spatial subsampling

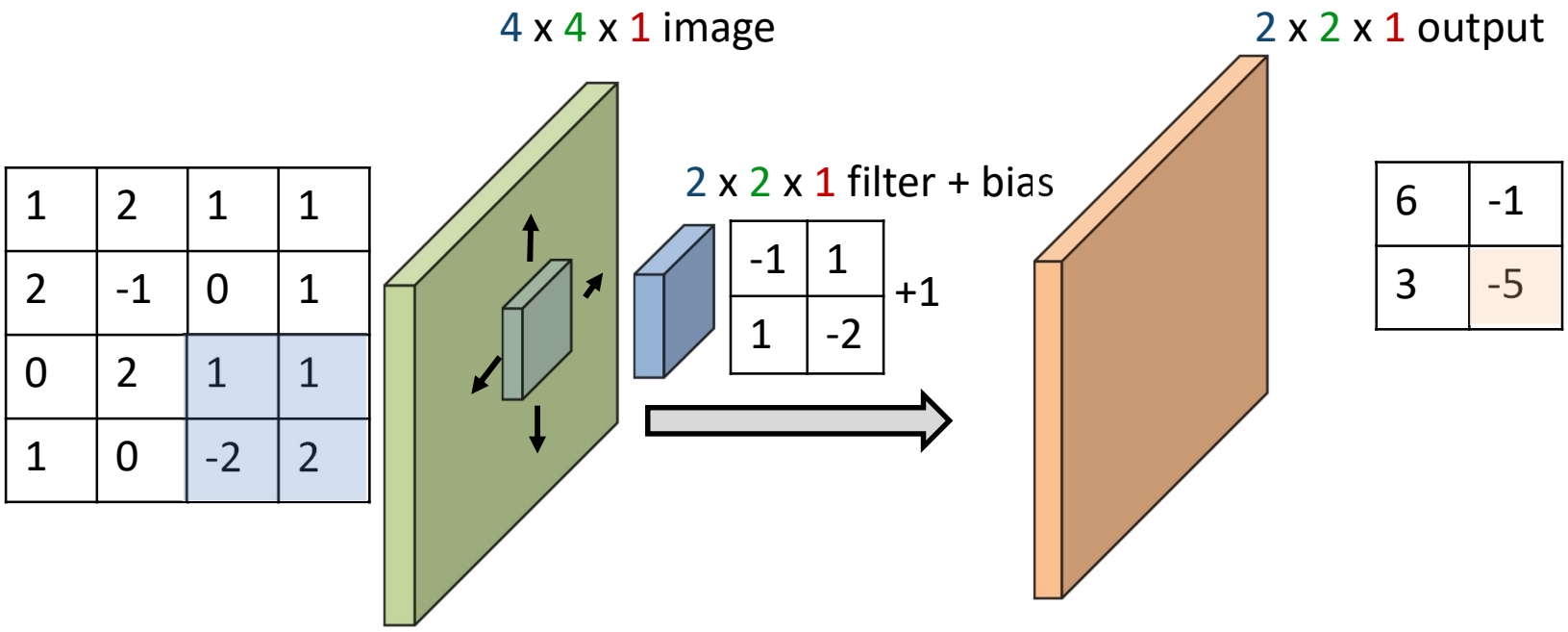
Example: convolution with 2 x 2 kernel and stride 2



Alternative to Pooling: Strided Convolutions

- Instead of pooling layer, we can also use so-called strided convolution layers
- **Idea**: Move the filter by more than one unit in x/y direction
- Also performs spatial subsampling

Example: convolution with 2 x 2 kernel and stride 2

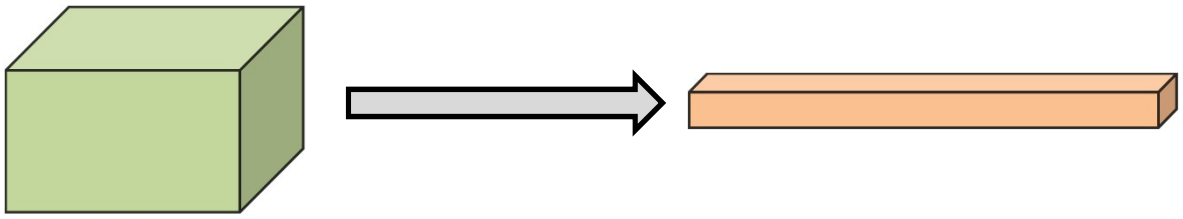


Flattening

- Convolution and pooling layers consume and produce 3D-tensors that represent spatially arranged feature activations
- The final layer of the neural network has to produce a 1D-vector of class scores from these feature activations
- Sometimes additional fully connected layers (1D) are also included before final output layer
- How is the final 3D-tensor transformed into a 1D representation?
- **Simplest approach: flattening (often represented as a layer)**

Before flattening layer:
m x *l* x *d*

After flattening layer:
1 x 1 x *mld*



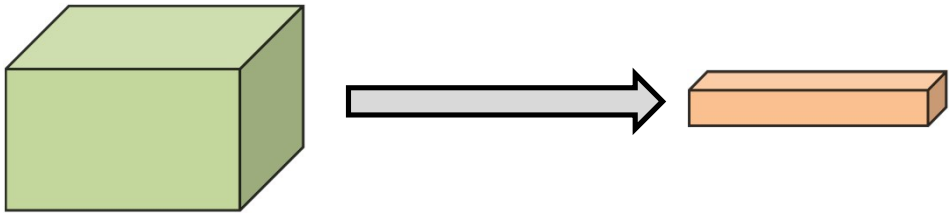
Simply rearrange the $m \cdot l \cdot d$ numbers in the $m \times l \times d$ tensor into a single vector

Global Average Pooling

- As an alternative to flattening the 3D-tensor into a 1D-vector, we can also average out the spatial dimension
- **Global average pooling layer**:

Before global average pooling:
m x *l* x *d*

After global average pooling:
1 x 1 x *d*



For each channel in input, average feature activation over all spatial locations:

$$z[c] = \frac{1}{m \cdot l} \sum_{a=1}^{m} \sum_{b=1}^{l} x[a,b,c] \qquad 1 \leq c \leq d$$

1D-output

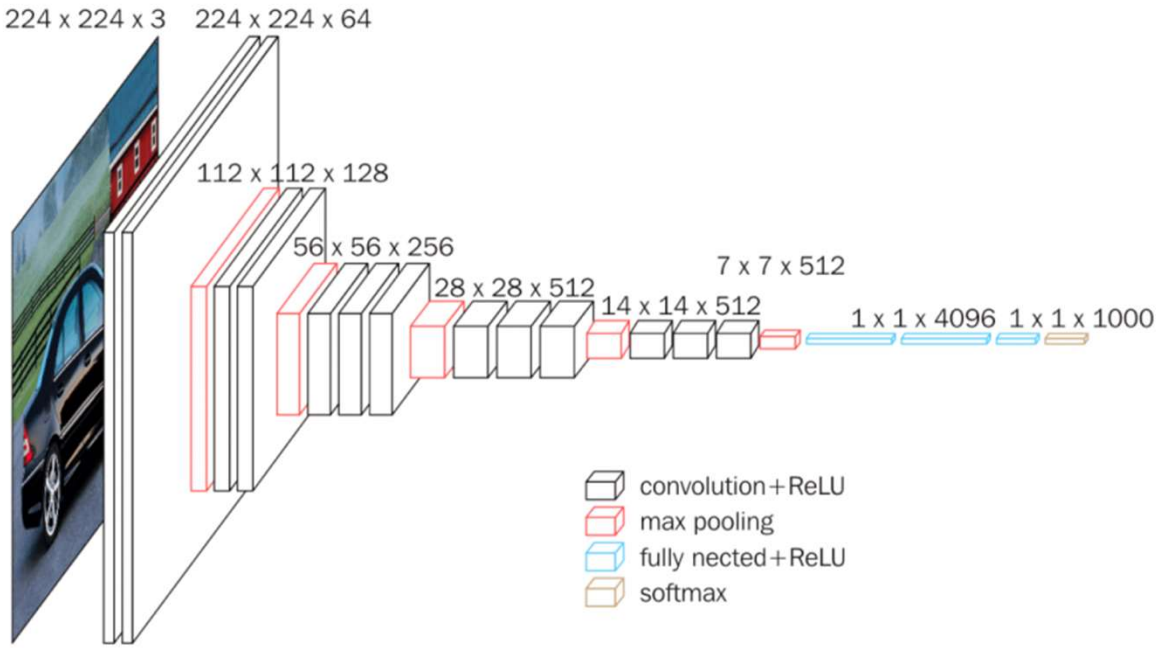
3D-input

Convolutional Neural Networks: Architectures

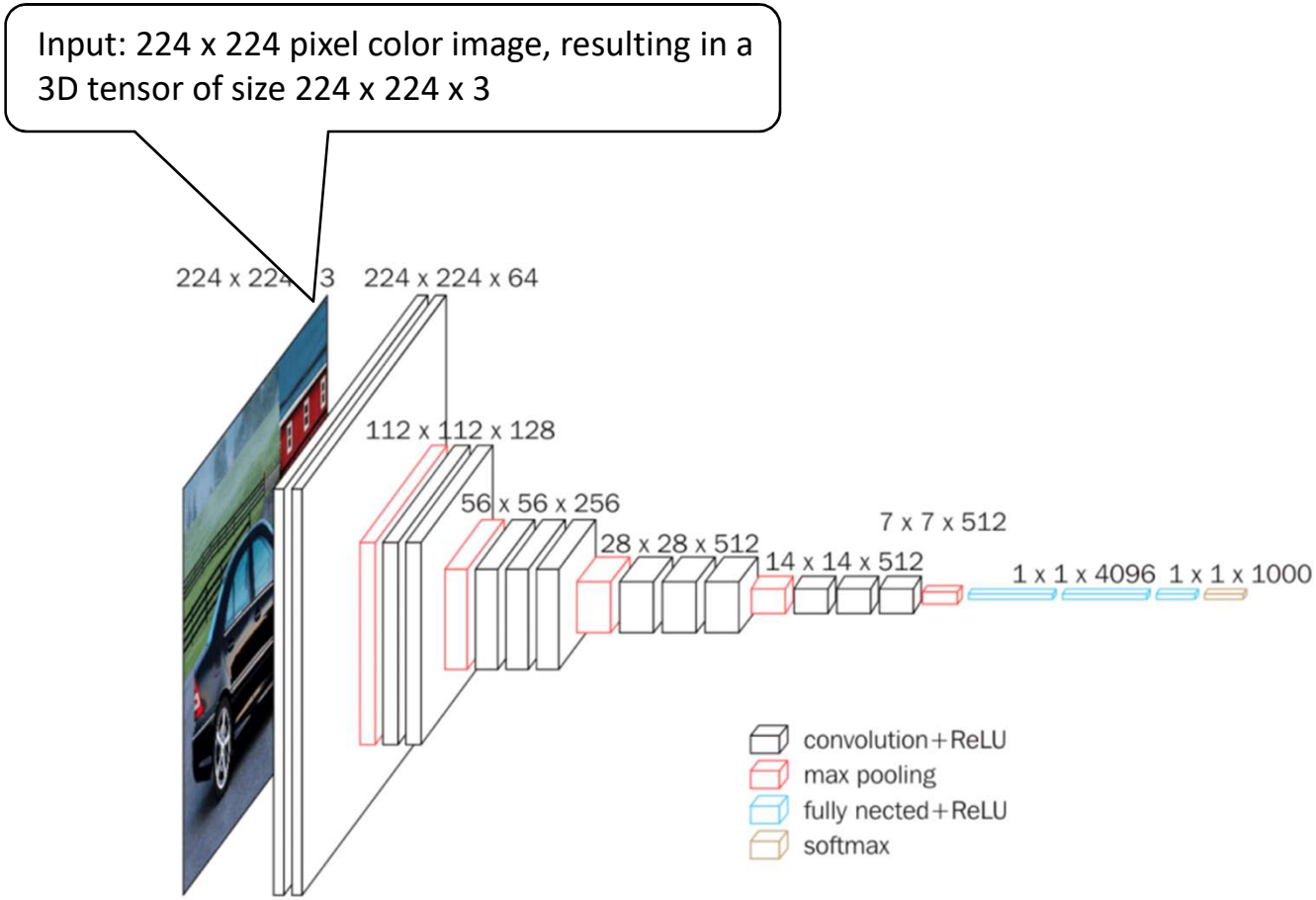
- Convolution layers are central building blocks in most deep neural networks for computer vision
- How do we build up a complete neural network model, e.g. for classification?
- General principle:
  - Start with convolution layers to learn low-level features directly on input
  - repeat:
    - Follow up with spatial subsampling (pooling or strided convolutions)
    - Follow up with more convolution layers to learn more abstract features
  - At the end, employ fully connected layers after flattening or global average pooling to learn a classification from the high-level level features in the final convolution layer
- **„Representation learning“**: most of the network can be interpreted as extracting more and more complex features from the data, on which final classification layer is then based
- **„End-to-end training“**: the whole network is trained directly from labeled data (unlike approaches that decouple feature extraction from training)

Example: VGG Architecture

- Example: VGG-16 architecture [Simonyan & Zisserman, 2015]
- Simple, widely used architecture for image classification (somewhat outdated)
- Depth: 16 parameter-carrying layers (excluding pooling layers)
- 13 convolution layers (with interleaved pooling) followed by 3 fully connected layers

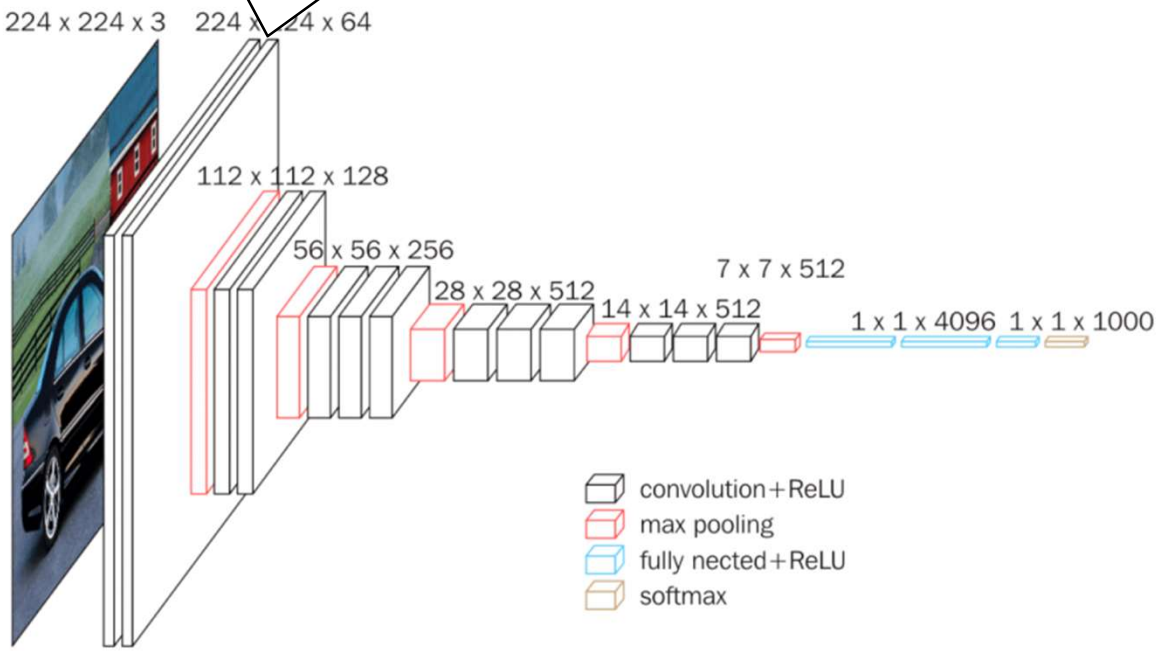


Example: VGG Architecture



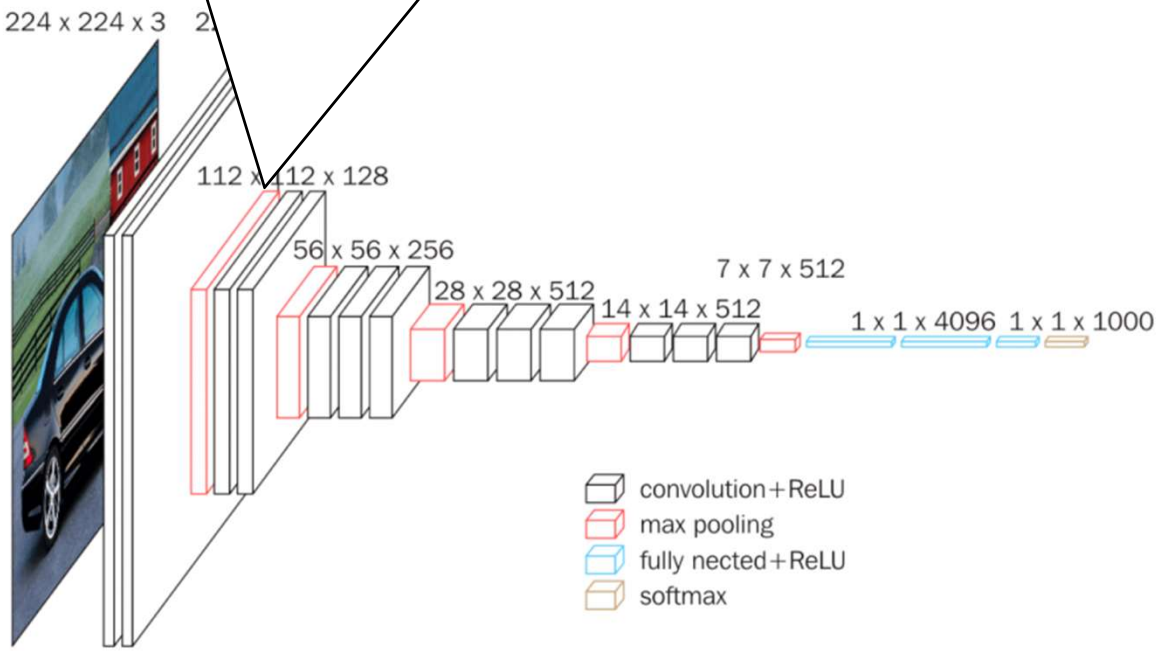
Example: VGG Architecture

- Each black box is a convolution layer followed by a ReLU nonlinear activation.
- Every convolution layer in the model uses 3 x 3 kernels and stride 1 and ReLU activations
- First convolution: input is 224 x 224 x 3, output is 224 x 224 x 64 (layer has 64 filters, uses padding)
- Second convolution: input is 224 x 224 x 64, output is again 224 x 224 x 64



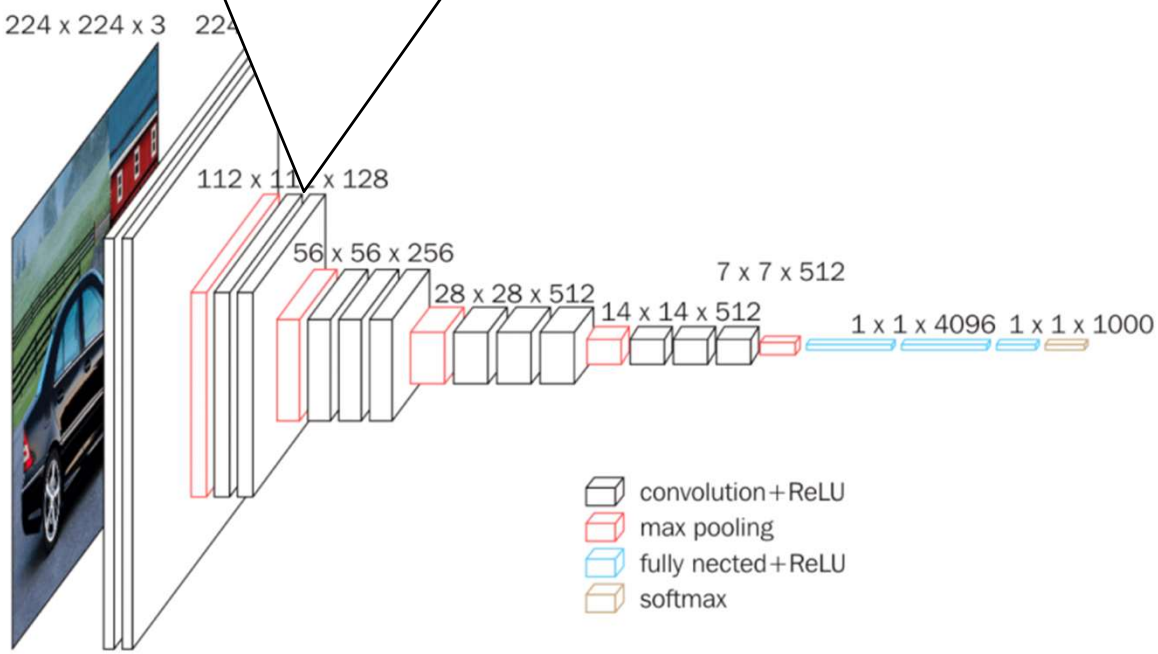
Example: VGG Architecture

- Each red box is a pooling layer, with kernel size 2, stride 2. This halves the spatial resolution.
- First pooling layer: input is output from second convolution, so 224 x 224 x 64, output is 112 x 112 x 64



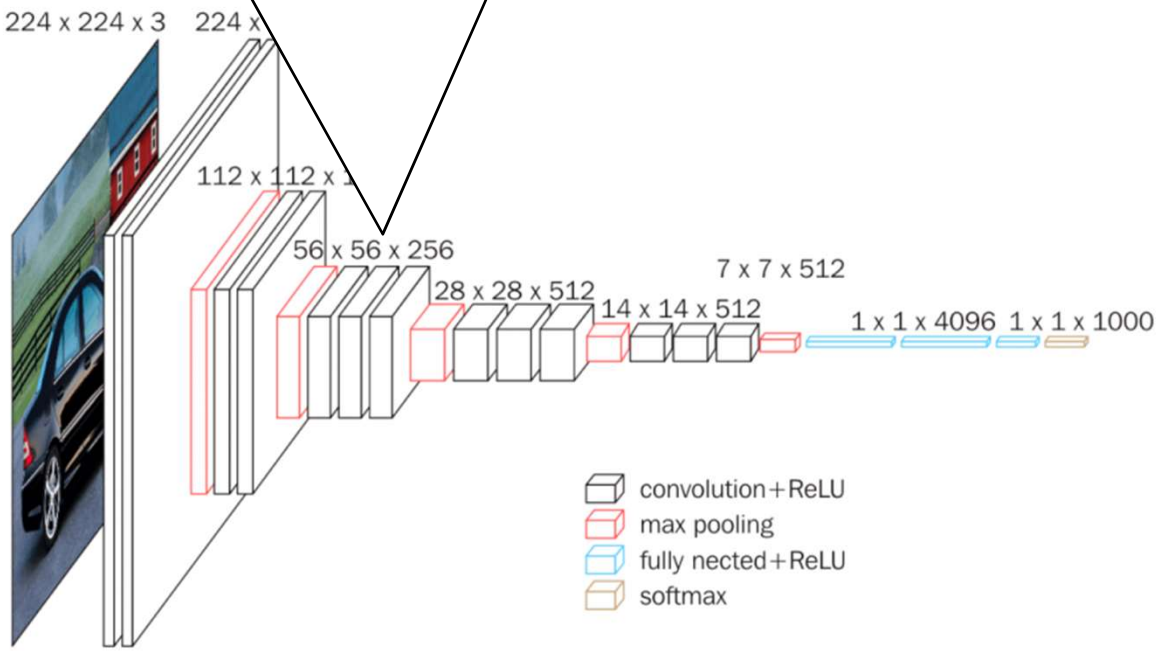
Example: VGG Architecture

- First pooling layer is followed by two more convolution layers
- First convolution layer: input is 112 x 112 x 64, output is 112 x 112 x 128 (layer has 128 filters)
- Second convolution layer: input is 112 x 112 x 128, output is 112 x 112 x 128



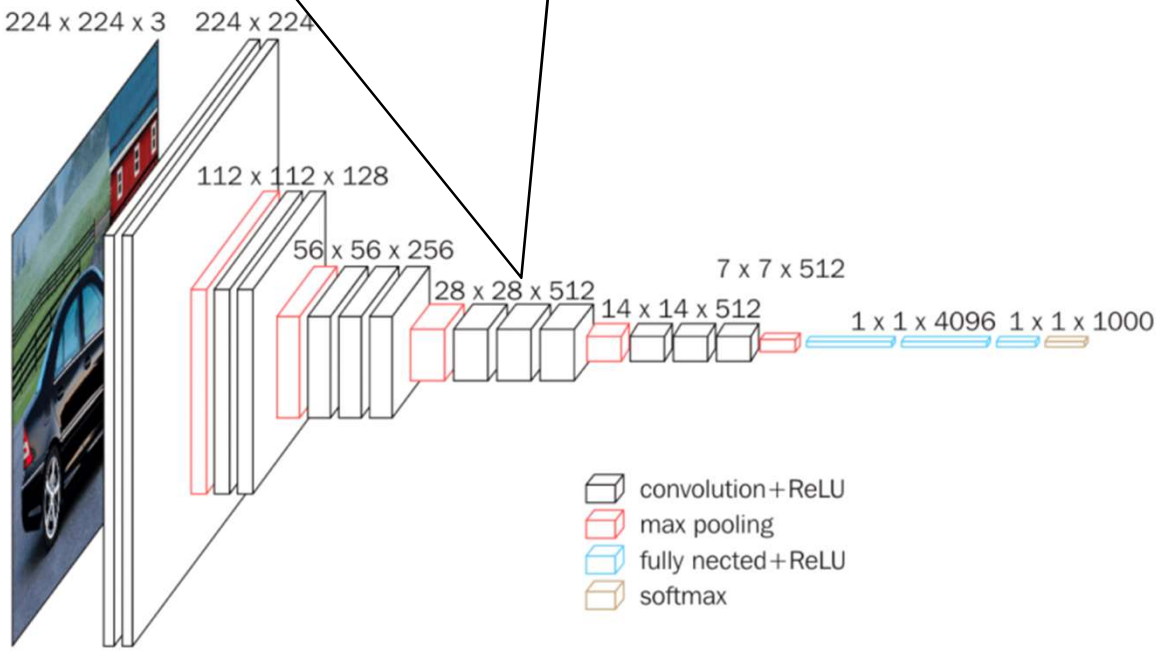
Example: VGG Architecture

- Again a block of pooling and convolution, with three convolution layers this time
- Spatial resolution (after the pooling operation) is now 56 x 56
- Number of filters is increased to 256



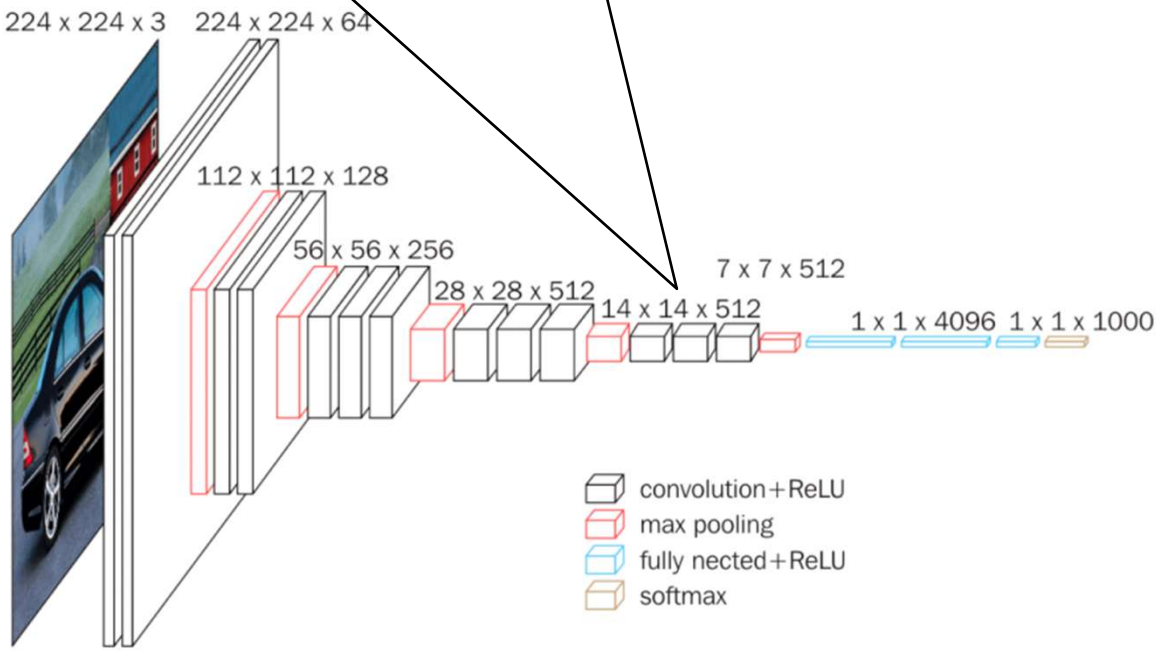
Example: VGG Architecture

- Again a block of pooling and convolution, with three convolution layers this time
- Spatial resolution (after the pooling operation) is now 28 x 28
- Number of filters is increased to 512



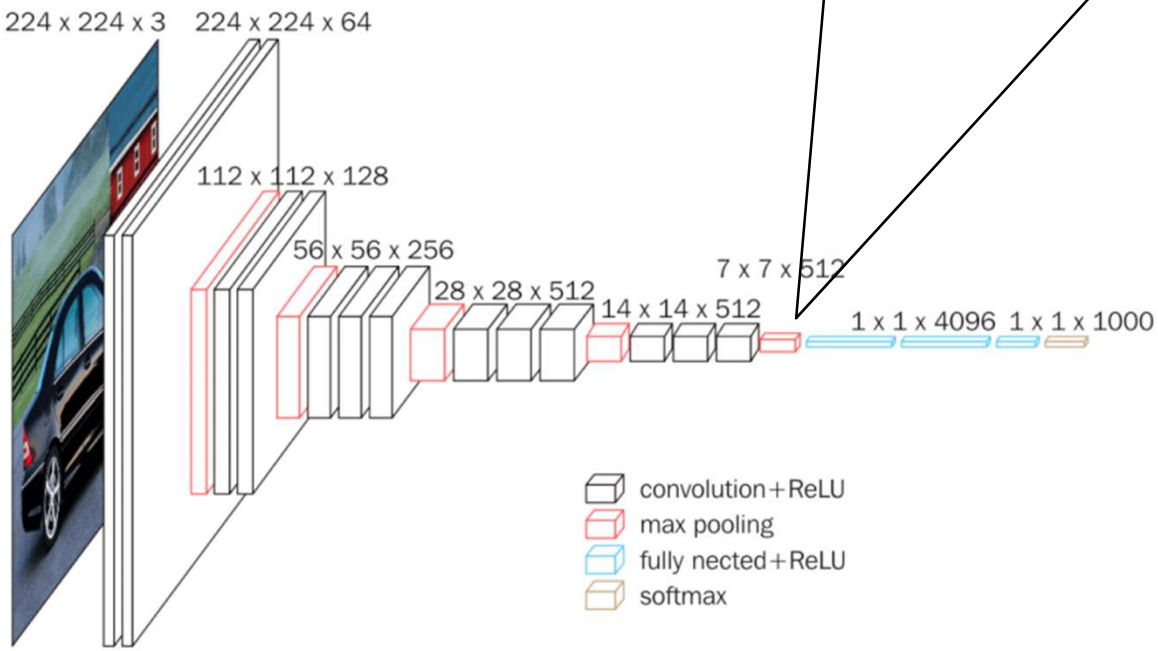
Example: VGG Architecture

- Again a block of pooling and convolution, with three convolution layers this time
- Spatial resolution (after the pooling operation) is now 14 x 14
- Number of filters stays at 512



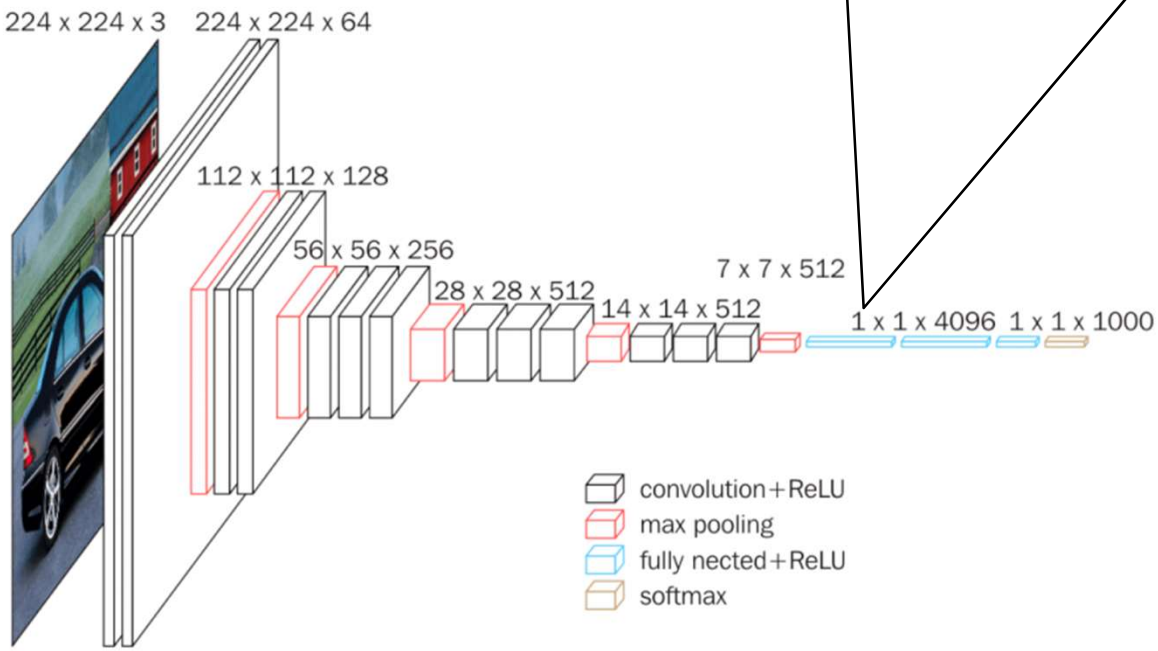
Example: VGG Architecture

- Final pooling layer reduces dimension to 7 x 7 x 512
- This layer represents high-level image content features, with a relatively low spatial resolution



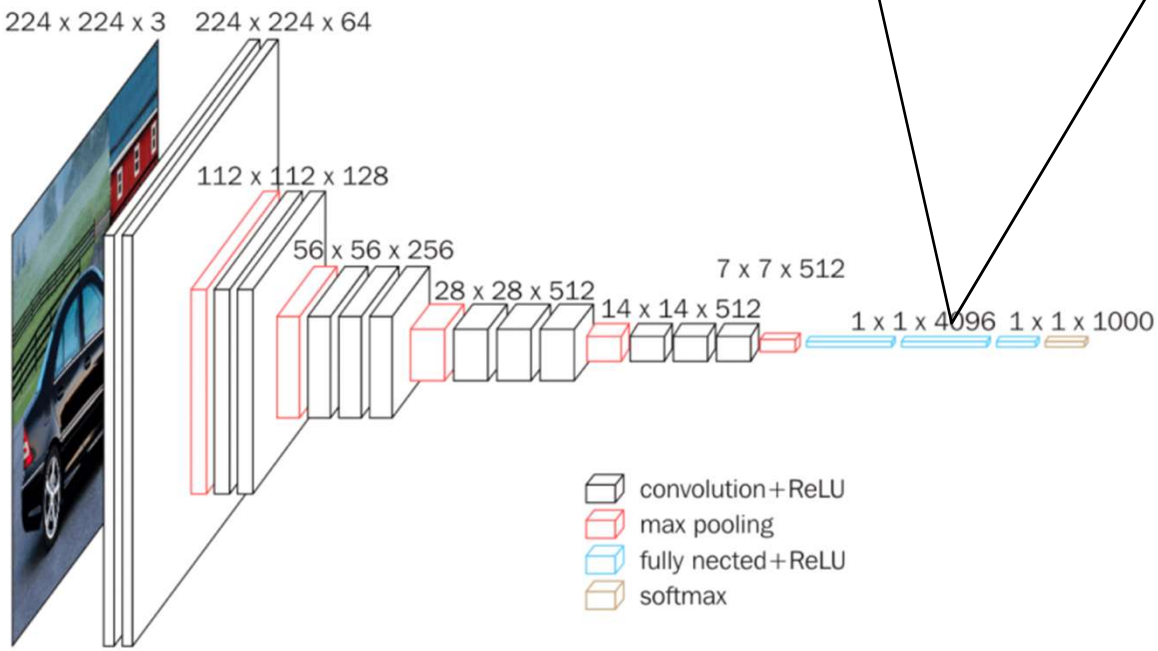
Example: VGG Architecture

- The final pooling layer output (7 x 7 x 512) is used as input to a fully connected layer.
- As the fully connected layer has no spatial structure, the 7 x 7 x 512 3D-tensor is flattened into a 25088-dimensional vector in the input to the fully connected layer
- The fully connected layer has 4096 neurons
- All fully connected layers also use ReLU activations



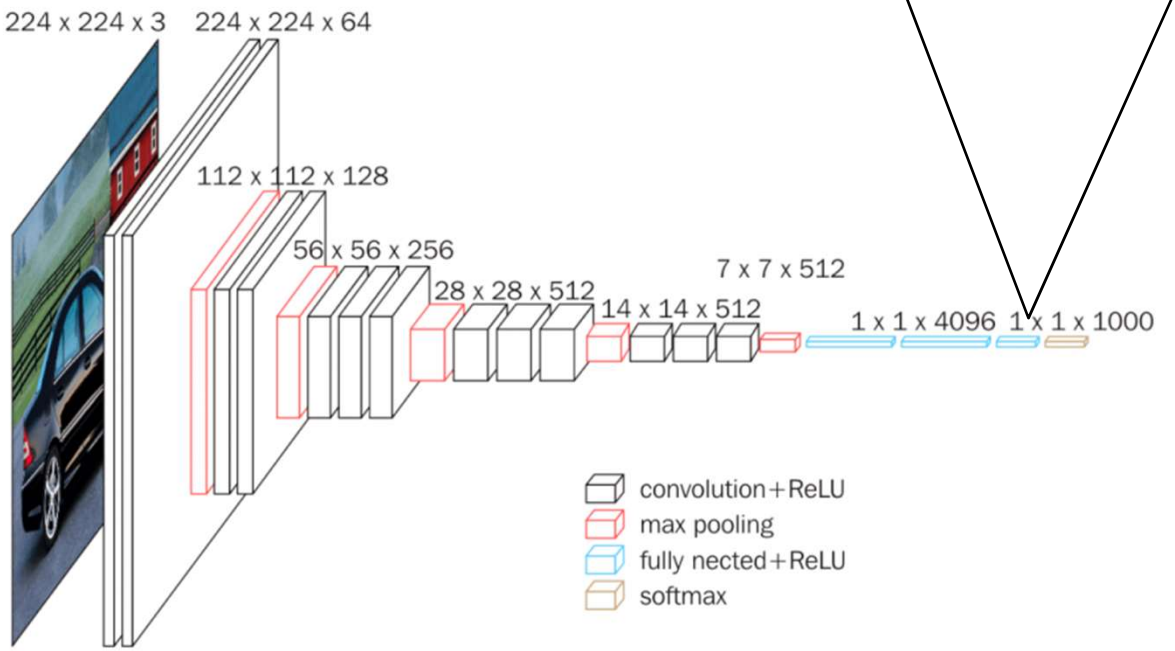
Example: VGG Architecture

- The 1 x 1 x 4096 vector output of the first fully connected layer is followed by another fully connected layer with 4096 neurons



Example: VGG Architecture

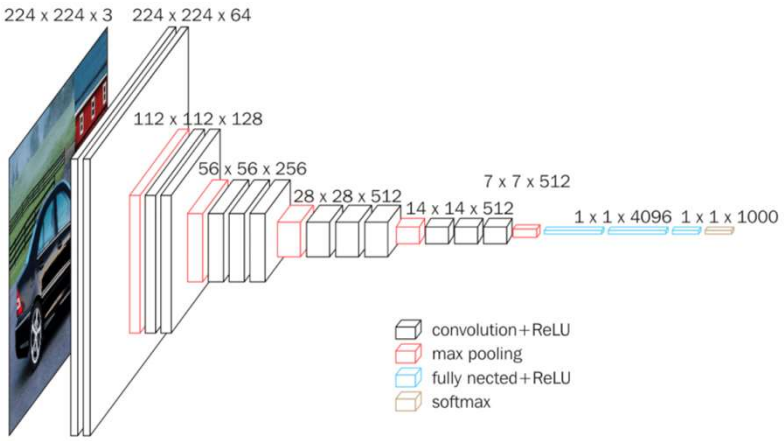
- The 1 x 1 x 4096 vector output of the second fully connected layer is followed by a final fully connected layer with 1000 neurons that returns class scores for 1000 classes (ImageNet model)
- The brown „softmax“ box refers to normalizing these class scores to probabilities by passing them through the exponential function and dividing by the sum over the 1000 exponentiated class scores.



CNN as Nested Function

- The complete convolutional neural network model is a function $f_{\theta} : \mathbb{R}^{224\times224\times3} \rightarrow \mathbb{R}^{1000}$ that maps images to scores for the *k*=1000 classes
- The function is mostly built up from stacked/nested (local) linear models and maximum operations (for max-pooling and ReLU activations)
- The parameter vector θ consists of all filter parameters in the convolution layers, all parameters in the fully connected layers, and all biases.
- VGG-16 example: approximately 132.000.000 parameters, 16.000.000.000 multiply-add operations

Input:
224*224*3=
150528 numbers



Output:
1000 class scores
(normalized to
probabilities)

Training by Stochastic Gradient Descent

- Models are trained by stochastic gradient descent as discussed above
  - Define loss function L, typically cross-entropy for classification
  - Put model + loss on data into (big) computation graph
  - Use automatic differentiation to compute $\nabla L(\boldsymbol{\theta}) \in \mathbb{R}^{132000000}$
  - Perform gradient descent on small batches of training data
  - Training will take a while...

Use deep learning framework such as Tensorflow, Pytorch, ...