



**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Convolutional Neural Networks

K. Breininger, S. Vesal, B. Geissler, N. Maul, L. Reeb, M. Vornehm, Z. Yang, A. Popp, M. Nau,  
S. Gündel, F. Denzinger, F. Thamm, C. Bergler, S. Jaganathan, F. Meister, C. Liu, T. Würfl

Pattern Recognition Lab, Friedrich-Alexander University of Erlangen-Nürnberg

May 17, 2020





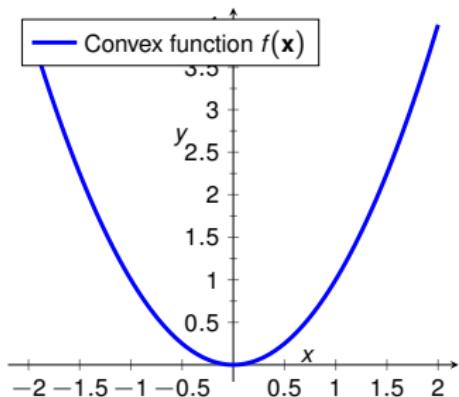
**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

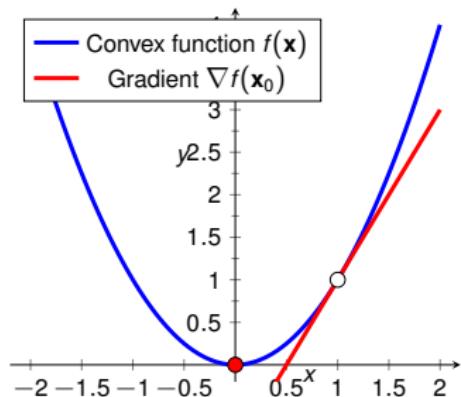
# Initializers



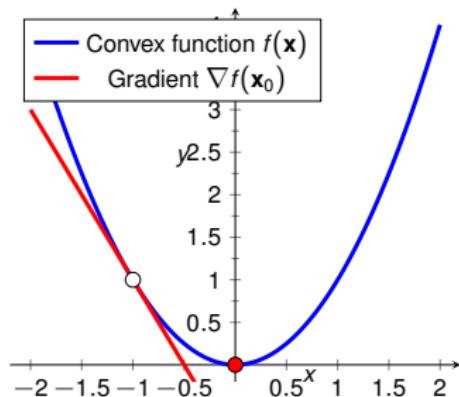
# Does initialization matter?



# Does initialization matter?

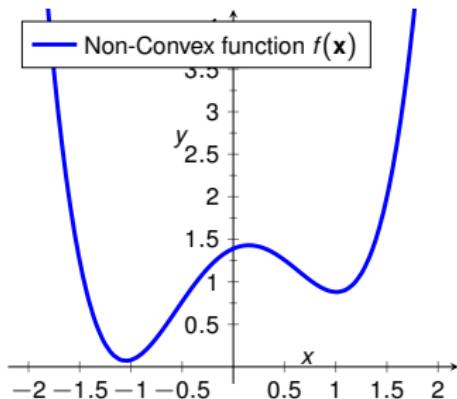


## Does initialization matter?



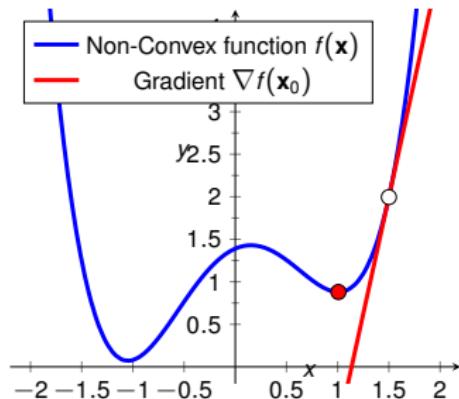
- No it doesn't for convex optimization problems

## Does initialization matter?



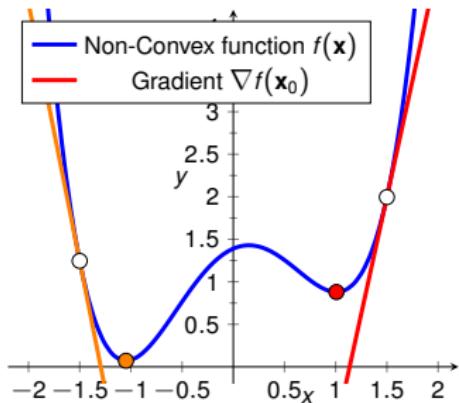
- No it doesn't for **convex** optimization problems

## Does initialization matter?



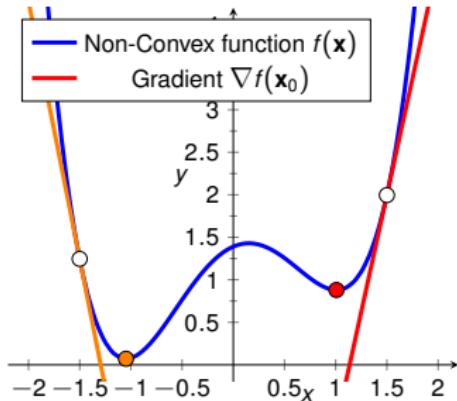
- No it doesn't for **convex** optimization problems

## Does initialization matter?



- No it doesn't for **convex** optimization problems
- But it **does** for every **non-convex** one

## Does initialization matter?



- No it doesn't for **convex** optimization problems
- But it **does** for every **non-convex** one
- Neural Networks with a non-linearity are in general **non-convex**

## Initializer objects

- **Goal:** Be flexible and allow different initialization strategies
- **Solution:** Every layer with weights will get initializer objects:  
One object for the **bias** and one for the other **weights**
- We have to refactor the code:
  - The **FullyConnected** layer to accept initializers
  - And the **NeuralNetwork** class to distribute them

# Simple initialization schemes

## Uniform

- Usually in the range  $[0, 1]$
- Same as before

## Constant

- With a given value
- Default to 0.1
- **Very bad** for weights
- Typically for biases
- . . . in conjunction with **ReLUs**

## Initializers: Nomenclature

The **number** of **inputs** and **outputs** to a layer are often used for initializing weights

- For **fully connected** layers:
  - “fan\_in”: **input** dimension of the weights
  - “fan\_out”: **output** dimension of the weights
- For **convolutional** layers:
  - “fan\_in”: **[# input channels × kernel height × kernel width]**
  - “fan\_out”: **[# output channels × kernel height × kernel width]**

128 input  
 11x11x3 Vector input  
 3?

	Batch	Height	Width	In Depth	Out Depth
Input	256	27	27	3	
kernel		11	11	3	96
Output	256	55	55		96

$(27 - 11 + 2 \times 1) / 4 + 1$

## Xavier/Glorot

- Typically for **weights**
- Normalizes weights with respect to number of units
- Zero-mean Gaussian:  $\mathcal{N}(0, \sigma)$
- $$\sigma = \sqrt{\frac{2}{\text{fan\_out} + \text{fan\_in}}}$$
  
*“fan\_in” and “fan\_out” as defined previously*

*avoid gradient vanishing*

## He

- Derived from Xavier initialization
- He initialization: Standard deviation of weights determined by size of previous layer only
- $\sigma = \sqrt{\frac{2}{\text{fan\_in}}}$
- Weights initialized by zero-mean Gaussian:  $\mathcal{N}(0, \sigma)$



**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Advanced Optimizers



## Momentum

- Parameter update based on current and past gradients:

$$\mathbf{v}^{(k)} = \underbrace{\mu}_{\text{momentum}} \mathbf{v}^{(k-1)} - \eta \underbrace{\nabla L(\mathbf{w}^{(k)})}_{\text{Gradient}}$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \mathbf{v}^{(k)}$$

- commonly:  $\mu = \{0.9, 0.95, 0.99\}$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \nabla L(\mathbf{w}^{(k)})$$

gradient descent !

$$\mathbf{v}^0 = 0$$

$$\mathbf{v}' = -\eta \nabla L(\mathbf{w}')$$

⋮

## Where to save intermediate values?

- We need to store optimization related intermediates
- They must be associated to some layer

## Where to save intermediate values?

- We need to store optimization related intermediates
- They must be associated to some layer
- Possibilities:
  - Make a cache and every layer identifiable
  - Enable every layer to have a customizable cache

## Where to save intermediate values?

- We need to store optimization related intermediates
- They must be associated to some layer
- Possibilities:
  - Make a cache and every layer identifiable
  - Enable every layer to have a customizable cache
- Our solution: Make the bias and weights of every layer have a copy of the optimizer  
*deep copy*

## Where to save intermediate values?

- We need to store optimization related intermediates
- They must be associated to some layer
- Possibilities:
  - Make a cache and every layer identifiable
  - Enable every layer to have a customizable cache
- Our solution: Make the bias and weights of every layer have a copy of the optimizer
- This means each set of weights could have a different optimizer

# ADAM Adaptive Momentum Estimation

- Parameter update based on current and past gradients:

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$

Momentum : first order

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} + (1 - \mu) \mathbf{g}^{(k)}$$

Adaptive : second order

$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1 - \rho) \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$

element-wise

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\hat{\mathbf{v}}^{(k)}}{\sqrt{\hat{\mathbf{r}}^{(k)}} + \epsilon}$$

normalization:  
standard deviation

- commonly:  $\mu = 0.9$ ,  $\rho = 0.999$ ,  $\eta = 0.001$

# ADAM

- Parameter update based on current and past gradients:

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$

$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} + (1 - \mu) \mathbf{g}^{(k)}$$

$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1 - \rho) \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$

Bias correction:  $\hat{\mathbf{v}}^{(k)} = \frac{\mathbf{v}^{(k)}}{1 - \mu^k}$     $\hat{\mathbf{r}}^{(k)} = \frac{\mathbf{r}^{(k)}}{1 - \rho^k}$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\hat{\mathbf{v}}^{(k)}}{\sqrt{\hat{\mathbf{r}}^{(k)}} + \epsilon}$$

- commonly:  $\mu = 0.9, \rho = 0.999, \eta = 0.001$
- The k is actually an exponent, not an iteration-index!



**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Convolution layer



## Vectors versus Images

- So far we only considered **batches** of abstract **input vectors**
- This has been intuitive when Neural Networks were considered classifiers

## Vectors versus Images

- So far we only considered **batches** of abstract **input vectors**
- This has been intuitive when Neural Networks were considered classifiers
- For feature learning, we have to consider **spatial** layout again

## Vectors versus Images

- So far we only considered **batches** of abstract **input vectors**
- This has been intuitive when Neural Networks were considered classifiers
- For feature learning, we have to consider **spatial** layout again
- Convolution layers therefore have to consider the spatial dimensions

## Vectors versus Images

- So far we only considered **batches** of abstract **input vectors**
- This has been intuitive when Neural Networks were considered classifiers
- For feature learning, we have to consider spatial layout again
- Convolution layers therefore have to consider the spatial dimensions
- Keep in mind: We can also convolve 1-D signals!

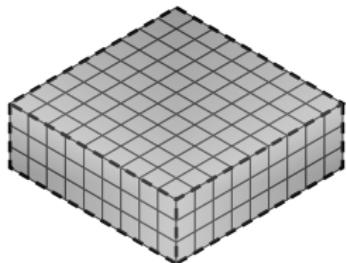
## Forward pass



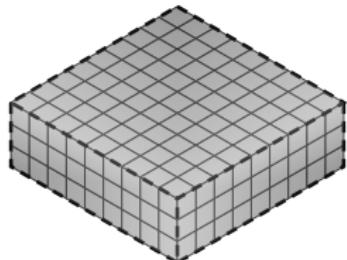
Figure: Convolution

Source: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

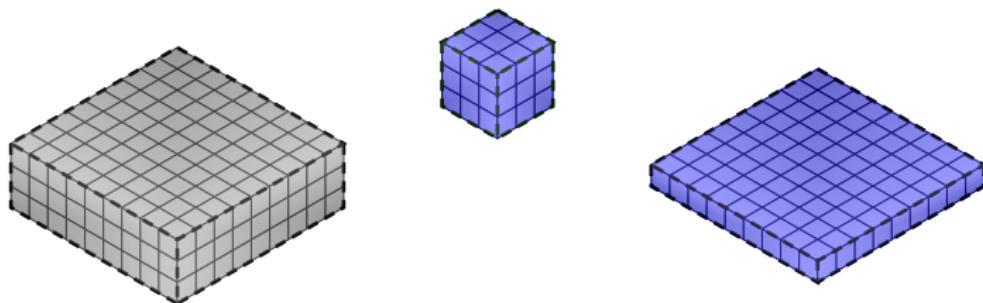
## Forward pass, Multi channel, Multi output maps



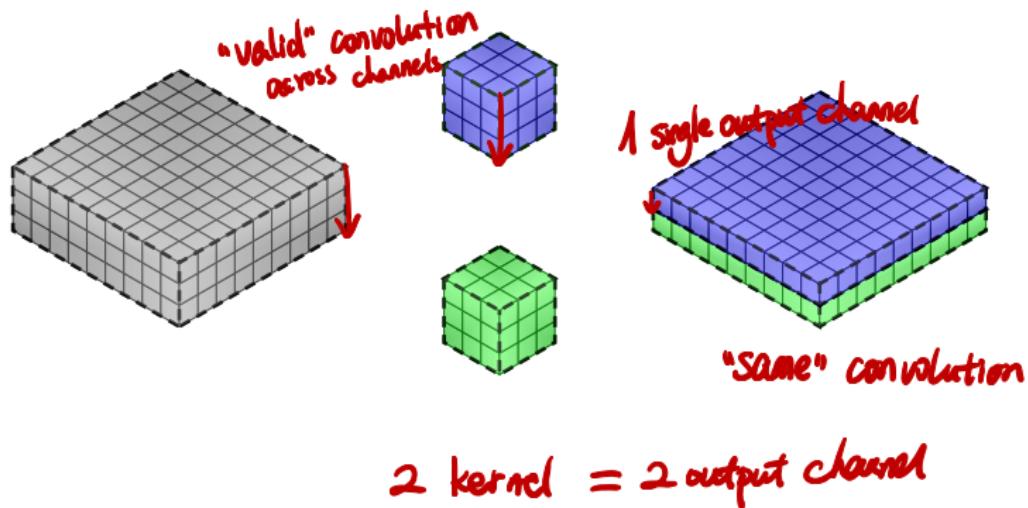
## Forward pass, Multi channel, Multi output maps



## Forward pass, Multi channel, Multi output maps



## Forward pass, Multi channel, Multi output maps



avoid "info loosing" | valid: no padding, kernel ~~size~~ ~~ima~~

## Forward pass

padding | same: with padding, kernel ~~size~~ ~~ima~~  
~~size~~  $\frac{k-1}{2}$  output size = ima size

## Convolution implementation

- Run a loop for every element of the **batch**
- The "depth" dimension  $S$  is **identical** for **kernel** and **image**
  - fully connected across channels
  - 3D convolution with no padding across channels
- The number of kernels  $H$  determines the output "depth"  
ONE
- Bias is an element-wise addition of a scalar value for every kernel 
- Important! We have a 'same' convolution across the **image plane axes** and a 'valid' convolution across the **channel axis** 
- Even kernel sizes are allowed
  - This requires asymmetric padding at the boundaries.

## Forward pass

### Matrix implementation

- Convolution is a linear operator → it has a matrix representation
- Reshape the kernel to the correct matrix performing the convolution

96'7 kernel , stride = 4 , padding = 0

	Batch	Height	Width	In Depth	Out Depth
Input	256	227	227	3	
kernel		11	11	3	96
Output	256	55	55		96

$(227 - 11 + 2 * 0) / 4 + 1$

## Backward pass

### Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^T \mathbf{E}_n$
- $\nabla \mathbf{W} = \mathbf{E}_n \mathbf{X}^T$

### Convolution implementation

## Backward pass

### Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^T \mathbf{E}_n$
- $\nabla \mathbf{W} = \mathbf{E}_n \mathbf{X}^T$

### Convolution implementation

- Backward pass is also a convolution but with spatially-flipped filters

## Backward pass

### Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^T \mathbf{E}_n$
- $\nabla \mathbf{W} = \mathbf{E}_n \mathbf{X}^T$

### Convolution implementation

- Backward pass is also a convolution but with spatially-flipped filters
- Instead of flipping filters, cross correlation (CC) can be used...
- ... and vice versa, we can use CC in the forward and convolution in the backward pass

## Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau \quad (1)$$

## Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau \quad (1)$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau)g(x + \tau)d\tau \quad (2)$$

## Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau)g(x - \tau)d\tau \quad (1)$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau) \underbrace{g(x + \tau)}_{\text{Flipped kernel}} d\tau \quad (2)$$

- This means cross correlation flips our kernel automatically!

## Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau) g(x - \tau) d\tau \quad (1)$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau) \underbrace{g(x + \tau)}_{\text{Flipped kernel}} d\tau \quad (2)$$

- This means cross correlation flips our kernel automatically!
- Often cross correlation is used in the forward pass, because the weights are random anyway

## Backward pass

How does a pixel of the input contribute to the pixels of the output?

how a single pixel  $x \rightarrow y$  ?

$$y_1 = w_6 \cdot x$$

$$y_2 = w_5 \cdot x$$

$$y_3 = w_4 \cdot x$$



Figure: Convolution

## Backward pass

### Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^T \mathbf{E}_n$
- $\nabla \mathbf{W} = \mathbf{E}_n \mathbf{X}^T$

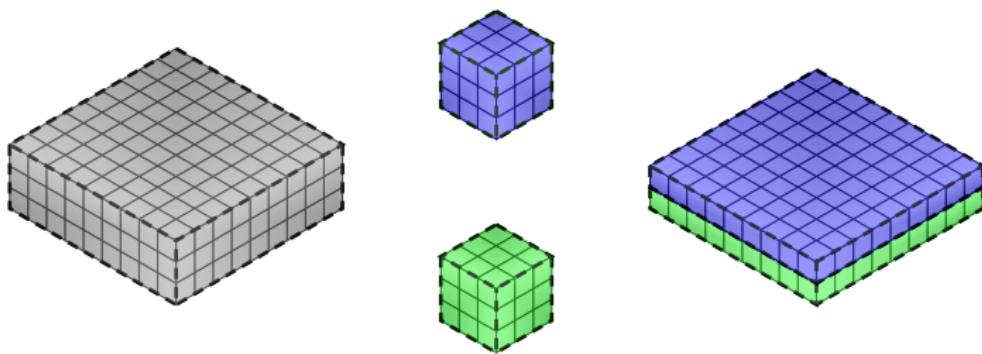
### Convolution implementation



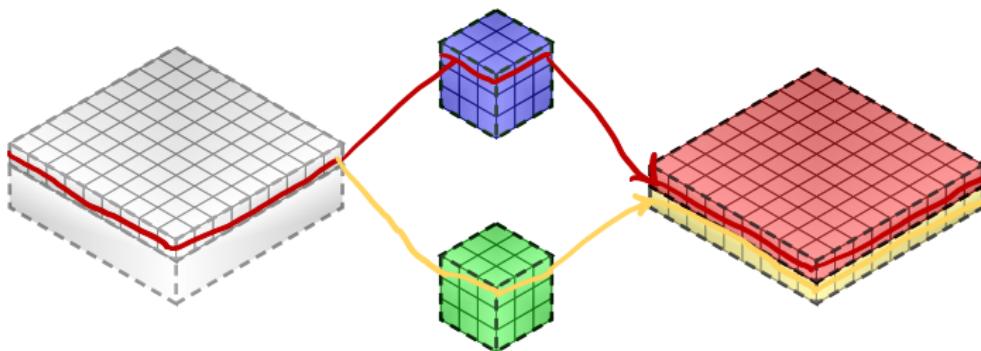
- The gradient with respect to the bias is simply sums over  $\mathbf{E}_n$  (attention: w)
- Filters need to be **flipped** (rotated 180°)
- What about the channels?
  - If we had  $H$  kernels with  $S$  channels
  - We obviously need  $S$  kernels in the backward pass → rearrange weights



## Backward pass - Gradient with respect to lower layers

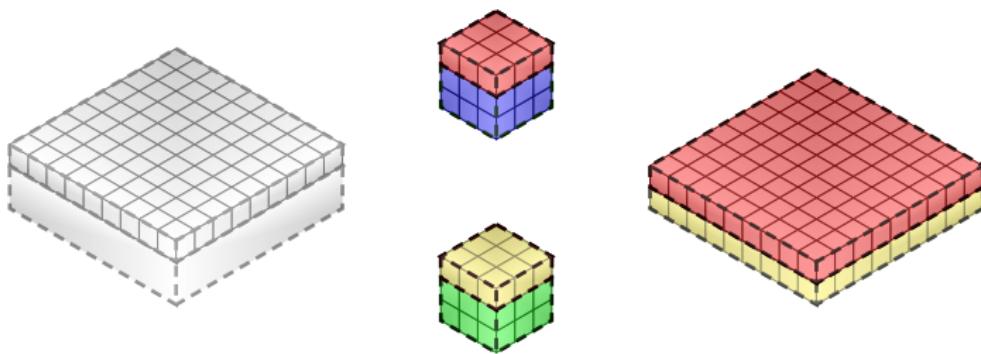


## Backward pass - Gradient with respect to lower layers



- The error is of the same dimension as the output

## Backward pass - Gradient with respect to lower layers



- The error is of the same dimension as the output
- Channel  $h$  of  $\mathbf{E}_{n-1}$  **depends only** on the  $H$  kernels  $\mathbf{K}_{s,N,M}$ , where  $h = s$

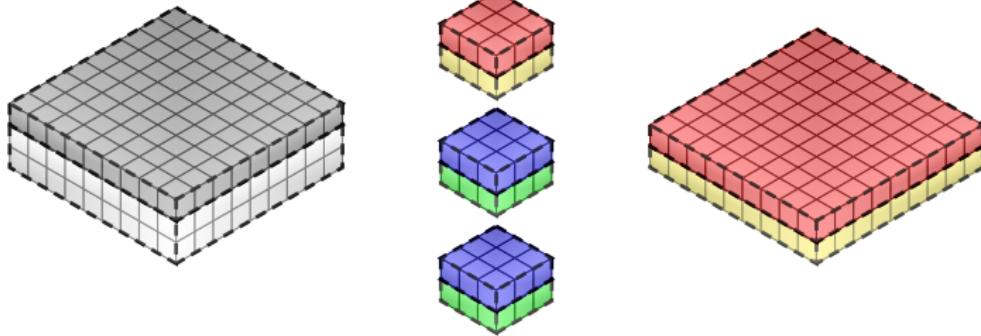
## Backward pass - Gradient with respect to lower layers



- The error is of the same dimension as the output
- Channel  $h$  of  $\mathbf{E}_{n-1}$  **depends only** on the  $H$  kernels  $\mathbf{K}_{s,N,M}$ , where  $h = s$
- The channels of the  $H$ ,  $\mathbf{K}_{S,N,M}$  kernels can be combined to a  $\hat{\mathbf{K}}_{H,N,M}$  one

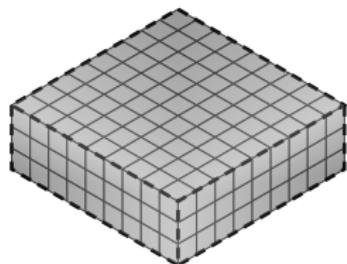


## Backward pass - Gradient with respect to lower layers

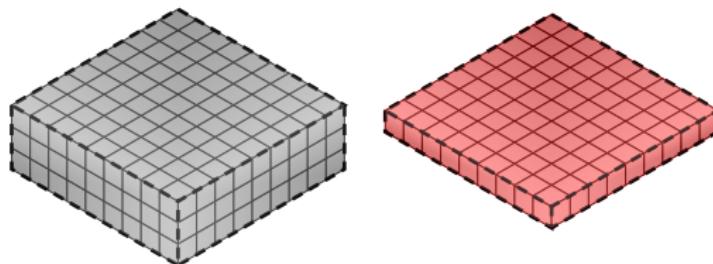


- The error is of the same dimension as the output
- Channel  $h$  of  $\mathbf{E}_{n-1}$  **depends only** on the  $H$  kernels  $\mathbf{K}_{s,N,M}$ , where  $h = s$
- The channels of the  $H$ ,  $\mathbf{K}_{S,N,M}$  kernels can be combined to a  $\hat{\mathbf{K}}_{H,N,M}$  one
- We have to **combine the channels** of the  $H$  kernels to  $S$  new kernels
- If a 3D-cross-correlation was used in the forward pass and 3D-convolution in the backward, only the channel dimension needs to be flipped!
- If cross-correlation and convolution were 2D, no channel flipping is needed.

## Backward pass - Gradient with respect to the weights

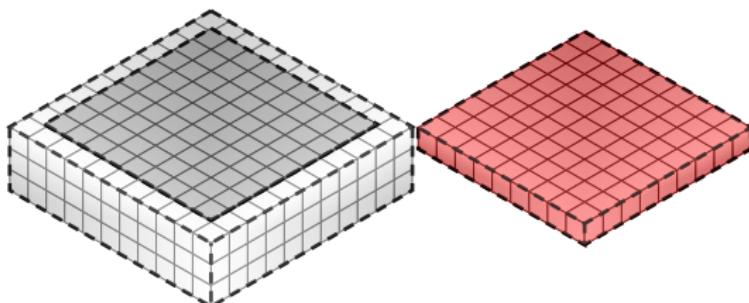


## Backward pass - Gradient with respect to the weights



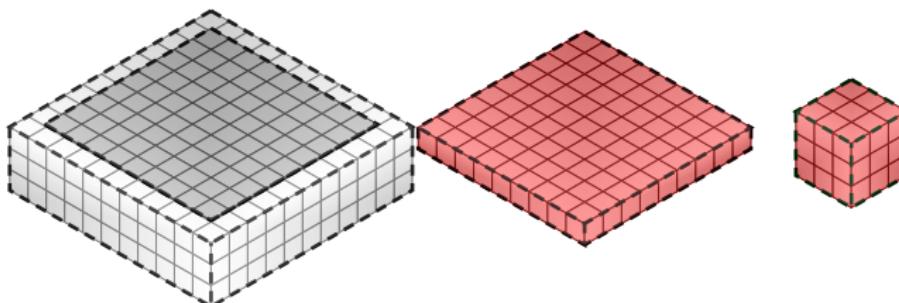
- We have to **correlate** the channels  $H$  of  $\mathbf{E}_{h,n}$  with those of  $\mathbf{X}$

## Backward pass - Gradient with respect to the weights



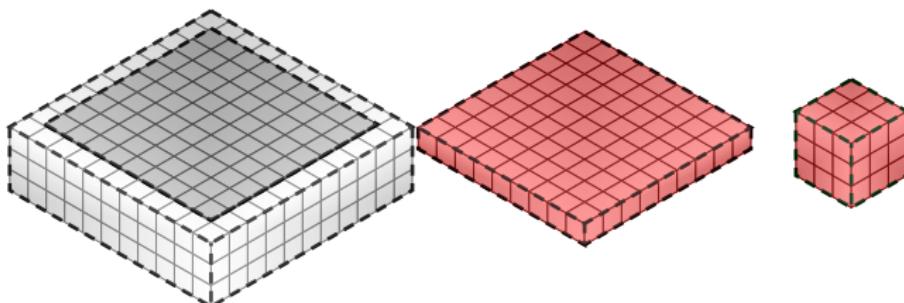
- We have to **correlate** the channels  $H$  of  $\mathbf{E}_{h,n}$  with those of  $\mathbf{X}$
- Pad  $\mathbf{X}$  with half the kernels' width

## Backward pass - Gradient with respect to the weights



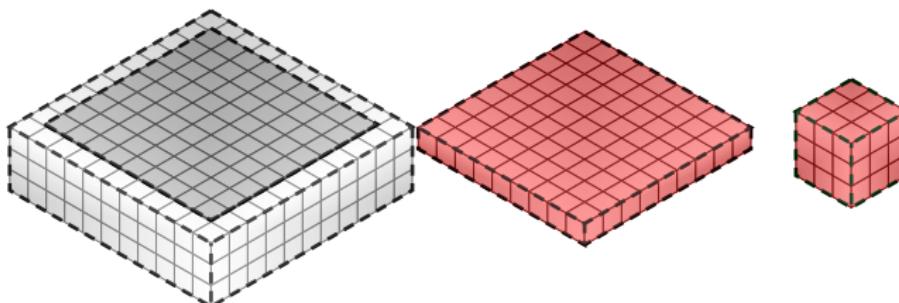
- We have to **correlate** the channels  $H$  of  $\mathbf{E}_{\underline{h},n}$  with those of  $\mathbf{X}$
- Pad  $\mathbf{X}$  with half the kernels' width
- Compute for  $s \in [1 \cdots S]$ :  $\mathbf{X}_s \star \mathbf{E}_{\underline{h},n}$  to receive the kernel  $\nabla K_{\underline{h},S,N,M}$

## Backward pass - Gradient with respect to the weights



- We have to **correlate** the channels  $H$  of  $\mathbf{E}_{\underline{h},n}$  with those of  $\mathbf{X}$
- Pad  $\mathbf{X}$  with half the kernels' width
- Compute for  $s \in [1 \cdots S]$ :  $\mathbf{X}_s \star \mathbf{E}_{\underline{h},n}$  to receive the kernel  $\nabla K_{\underline{h},S,N,M}$
- If correlation is used in the forward pass we directly receive the correct gradient in the backward pass

## Backward pass - Gradient with respect to the weights



- We have to **correlate** the channels  $H$  of  $\mathbf{E}_{\underline{h},n}$  with those of  $\mathbf{X}$
- Pad  $\mathbf{X}$  with half the kernels' width
- Compute for  $s \in [1 \cdots S]$ :  $\mathbf{X}_s \star \mathbf{E}_{\underline{h},n}$  to receive the kernel  $\nabla K_{\underline{h},s,N,M}$
- If correlation is used in the forward pass we directly receive the correct gradient in the backward pass
- If convolution is used in the forward pass, we have to manually rotate the x,y-plane by  $180^\circ$  of the kernels

# Stride

Figure: Strided convolution

Source: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

## Stride

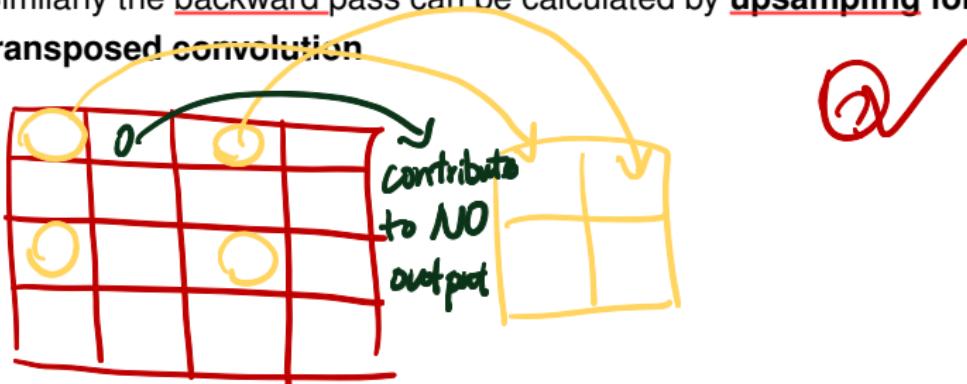
- Stride is often used to **reduce the dimension** of the input

e.g. stride = 2, input =  $\frac{1}{2}$  output

## Stride

**Forward pass: downsampling** less info,  $\frac{1}{2}$  output size < input size

- Stride is often used to **reduce the dimension** of the input
- More mathematically stride can be seen as **convolution followed by subsampling**
- Similarly the backward pass can be calculated by upsampling followed by transposed convolution

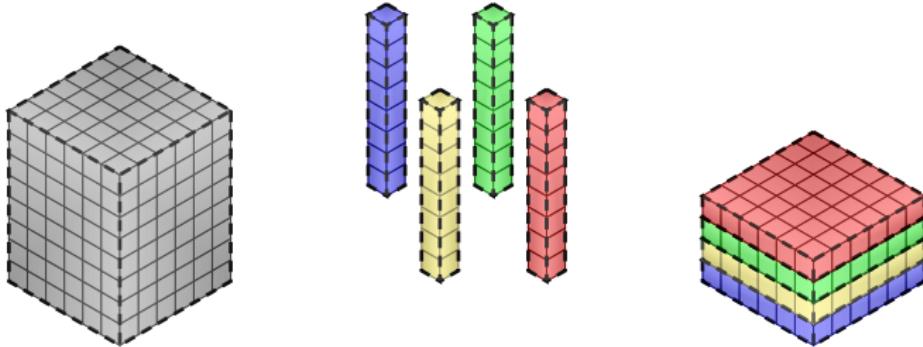


## Stride

- Stride is often used to **reduce the dimension** of the input
- More mathematically stride can be seen as **convolution followed by subsampling**
- Similarly the backward pass can be calculated by **upsampling followed by transposed convolution**
- Stride is not provided by any scipy/numpy convolution

## 1x1 Convolutions

- Important special case
- Equal to applying a **fully connected layer along the channels**





**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Pooling layer



# Forward pass max-pooling

Figure: Max-pooling

Source: [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)

## Forward pass max-pooling

- **Stride** is crucial now and controls amount of downsampling
- . . . and typically as big as the kernel size      *kernel w/o overlapping*
- We need to **store the locations** of the maxima

## Backward pass max-pooling



## Backward pass max-pooling



"THE WINNER  
TAKES IT ALL"  
- En hyllning till ABBA

## Backward pass max-pooling

- A **subgradient** is given by the colloquial rule “**Winner takes it all**”
- Layer has no trainable parameters, hence only gradient with respect to input required
- We need the stored maxima locations
- The error is routed towards these locations and is zero for all other pixels
- And summed if the stride is smaller than the kernel size



**FAU**

FRIEDRICH-ALEXANDER-  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG  
SCHOOL OF ENGINEERING

# Flatten layer



## Flatten layer

What does it do?

Forward:

- **Input:** batch of multi-dimensional arrays (spatial + channels)
- **Output:** batch of one dimensional feature vectors
- “Linearizes” each element in a batch

Backward:  $1-D \rightarrow \text{Multi-D}$

# Flatten layer

## What does it do?

- **Input:** batch of multi-dimensional arrays (spatial + channels)
- **Output:** batch of one dimensional feature vectors
- “Linearizes” each element in a batch

## Why flatten?

- Enables connecting convolution/pooling and fully connected layers
- Modularity - flatten as a separate layer provides flexibility
- Alternatives include global pooling layers

Thanks for listening.  
**Any questions?**