Convolutional Neural Networks

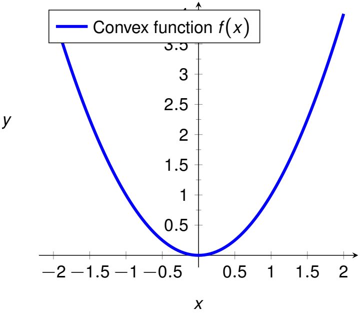
K. Breininger, V. Christlein, Z. Yang, L. Rist, M. Nau, S. Jaganathan, C. Liu, N. Maul, L. Folle, M. Zinnen, K. Packhäuser

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

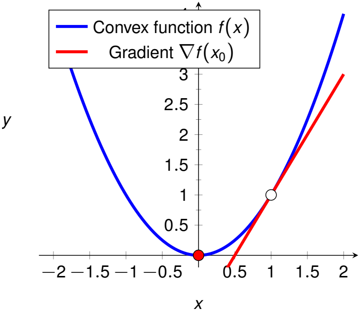
April 20, 2024

Initializers

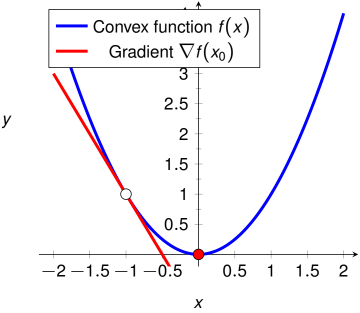
Does initialization matter?



Does initialization matter?

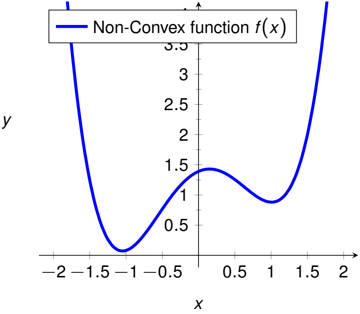


Does initialization matter?



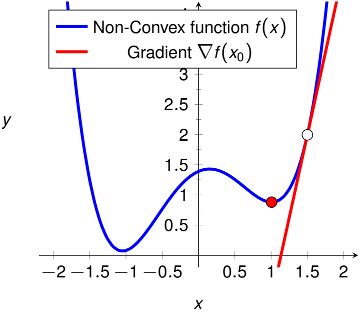
- No it doesn't for **convex** optimization problems

Does initialization matter?



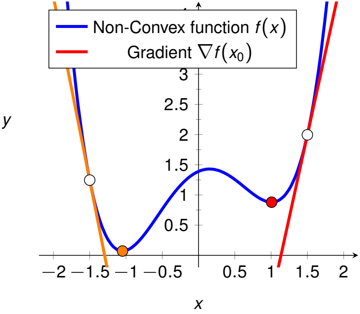
- No it doesn't for **convex** optimization problems

Does initialization matter?



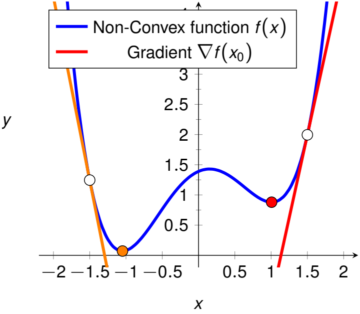
- No it doesn't for **convex** optimization problems

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**
- $\cdots$ in conjunction with **ReLU**s

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** $\times$ **kernel height** $\times$ **kernel width**]
  - “fan_out”: [**# output channels** $\times$ **kernel height** $\times$ **kernel width**]

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

He

- Derived from Xavier initialization

$\rightarrow$ 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: $\mathcal{N}(0, \sigma)$

Advanced Optimizers

Momentum

- Parameter update based on current and past gradients:

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

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

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

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

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

- Parameter update based on current and past gradients:

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

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

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

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

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

ADAM

- Parameter update based on current and past gradients:

$$\mathbf{g}^{(k)} = \nabla L(\mathbf{w}^{(k)})$$
$$\mathbf{v}^{(k)} = \mu \mathbf{v}^{(k-1)} + (1-\mu)\mathbf{g}^{(k)}$$
$$\mathbf{r}^{(k)} = \rho \mathbf{r}^{(k-1)} + (1-\rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}$$
$$\text{Bias correction: } \hat{\mathbf{v}}^{(k)} = \frac{\mathbf{v}^{(k)}}{1-\mu^{k}} \quad \hat{\mathbf{r}}^{(k)} = \frac{\mathbf{r}^{(k)}}{1-\rho^{k}}$$
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \frac{\hat{\mathbf{v}}^{(k)}}{\sqrt{\hat{\mathbf{r}}^{(k)}} + \epsilon}$$

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

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

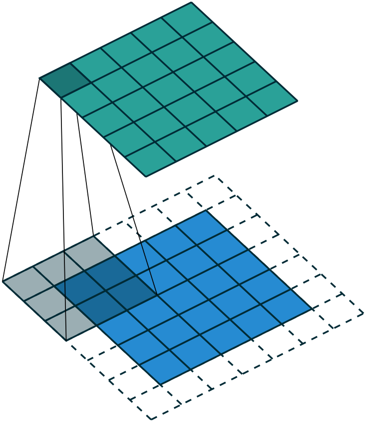
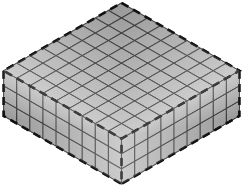


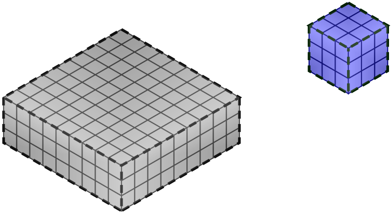
Figure: Convolution

Source: https://github.com/vdumoulin/conv_arithmetic

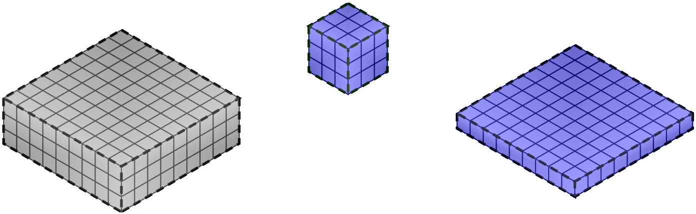
Forward pass, Multi channel, Multi output maps



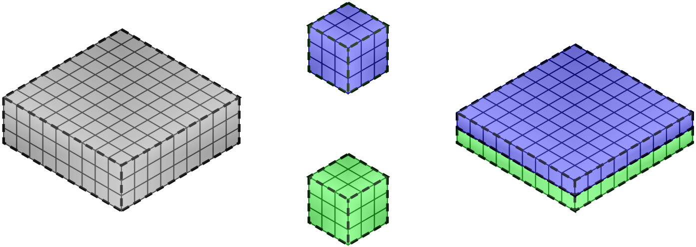
Forward pass, Multi channel, Multi output maps



Forward pass, Multi channel, Multi output maps



Forward pass, Multi channel, Multi output maps



Forward pass

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”
- **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 to result in the correct dimension.

Forward pass

Matrix implementation

- Convolution is a linear operator $\rightarrow$ it has a matrix representation
- Reshape the kernel to the correct matrix before performing the convolution

Backward pass

Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n$
- $\nabla\mathbf{W} = \mathbf{E_n}\mathbf{X}^T$
- **Needs a lot of rearranging to create the right weights and error matrices!**

Convolution implementation

Backward pass

Matrix implementation

- We can use the same formulas as in a fully connected layer!
- $\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n$
- $\nabla\mathbf{W} = \mathbf{E}_{\mathbf{n}}\mathbf{X}^T$
- **Needs a lot of rearranging to create the right weights and error matrices!**

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!
- $\mathbf{E}_{n-1} = \mathbf{W}^{\mathbf{T}}\mathbf{E}_n$
- $\nabla \mathbf{W} = \mathbf{E}_{\mathbf{n}}\mathbf{X}^T$
- **Needs a lot of rearranging to create the right weights and error matrices!**

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

Backward pass

Why to flip the filters in the backward pass?

- Lets consider the 1D case:
- We have an input $[a, b, c]$ and a filter $[x, y]$. Including padding this will lead to an output $[ay\ ax + by\ bx + cy\ cx]$.
- For the backward pass we want to calculate the derivative:

$$\frac{\partial \mathrm{E}}{\partial \mathrm{I}} = \frac{\partial \mathrm{E}}{\partial \mathrm{O}} \frac{\partial \mathrm{O}}{\partial \mathrm{I}} \tag{1}$$

- The important part now is $\frac{\partial \mathrm{O}}{\partial \mathrm{I}}$.
- E.g. if we calculate $\frac{\partial \mathrm{O}}{\partial \mathrm{b}} \rightarrow [0\ y\ x\ 0] \rightarrow$ the kernel is flipped.
- Also for a and c we will get a flipped kernel.

Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau) g(x - \tau) \mathrm{d}\tau \tag{2}$$

Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau)g(x-\tau)\mathrm{d}\tau \tag{2}$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau)\, g(x+\tau)\, \mathrm{d}\tau \tag{3}$$

Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau) g(x - \tau) \mathrm{d}\tau \tag{2}$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau) \underbrace{g(x + \tau)}_{\text{Flipped kernel}} \mathrm{d}\tau \tag{3}$$

- Cross correlation = convolution with flipped kernel and vice versa. Therefore, using cross correlation flips our kernel automatically!

Convolution versus cross correlation

- Convolution:

$$(f * g)(x) := \int_{-\infty}^{\infty} f(\tau) g(x - \tau) \mathrm{d}\tau \tag{2}$$

- Cross correlation:

$$(f \star g)(x) := \int_{-\infty}^{\infty} f(\tau) \underbrace{g(x + \tau)}_{\text{Flipped kernel}} \mathrm{d}\tau \tag{3}$$

- Cross correlation = convolution with flipped kernel and vice versa. Therefore, using cross correlation flips our kernel automatically!
- Often cross correlation is used in the **forward pass**, because the weights are random anyway. This means convolution is then used if you want to flip the kernel

Backward pass

How to handle the bias on backward pass

- The bias in the backward pass can be handled by:

$$\frac{\partial \mathrm{L}}{\partial \mathrm{b}} = \sum_{b,w,h}^{B,W,H} \mathrm{E}_{b,w,h} \tag{4}$$

Backward pass

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

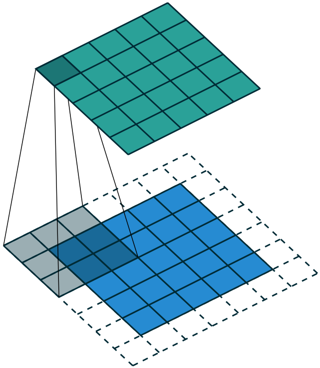


Figure: Convolution

Backward pass

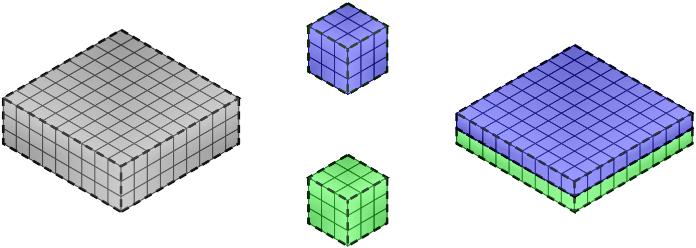
Matrix implementation

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

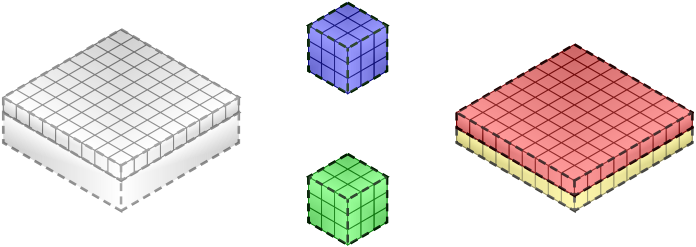
Convolution implementation

- The gradient with respect to the bias is simply sums over $\mathbf{E}_n$
- 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 $\rightarrow$ rearrange weights

Backward pass - Gradient with respect to lower layers

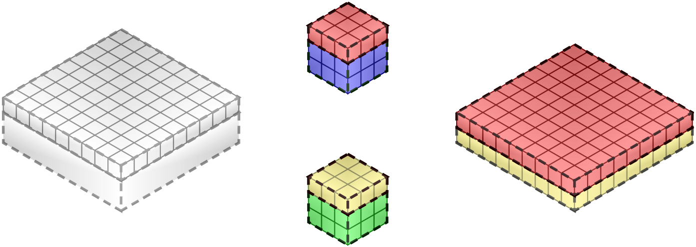


Backward pass - Gradient with respect to lower layers



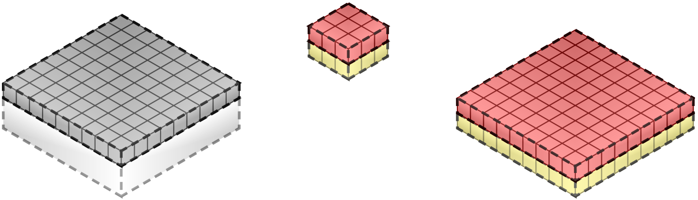
- Channel h of $\mathbf{E}_{n-1}$ **depends only** on the H kernels $\mathbf{K}_{\underline{s},N,M}$, where $h = s$

Backward pass - Gradient with respect to lower layers



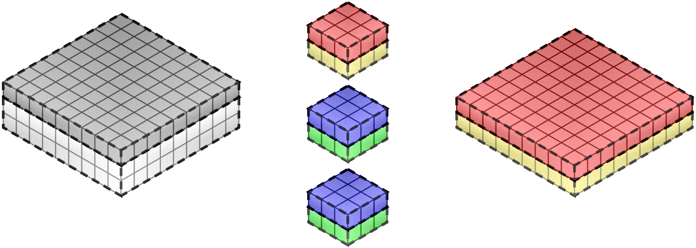
- Channel h of $\mathbf{E}_{n-1}$ **depends only** on the H kernels $\mathbf{K}_{\underline{s},N,M}$, where $h = s$
- The channels of the H, $\mathbf{K}_{\underline{S},N,M}$ kernels can be combined to a $\hat{\mathbf{K}}_{\underline{H},N,M}$ one

Backward pass - Gradient with respect to lower layers



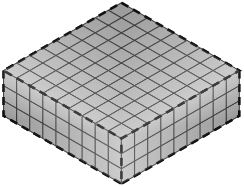
- Channel h of $\mathbf{E}_{n-1}$ **depends only** on the H kernels $\mathbf{K}_{\underline{s},N,M}$, where $h = s$
- The channels of the H, $\mathbf{K}_{\underline{S},N,M}$ kernels can be combined to a $\hat{\mathbf{K}}_{\underline{H},N,M}$ one
- We have to **combine the channels** of the H kernels to S new kernels

Backward pass - Gradient with respect to lower layers

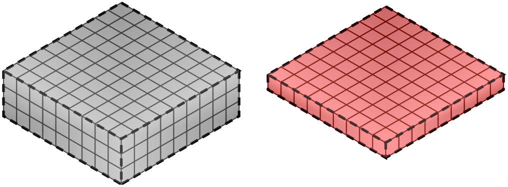


- Channel h of $\mathbf{E}_{n-1}$ **depends only** on the H kernels $\mathbf{K}_{\underline{s},N,M}$, where $h = s$
- The channels of the H, $\mathbf{K}_{\underline{S},N,M}$ kernels can be combined to a $\hat{\mathbf{K}}_{\underline{H},N,M}$ one
- We have to **combine the channels** of the H kernels to S new kernels
- Using 3D operations is possible if you include the channel dimension
- If a 3D-cross-correlation was used in the forward pass and 3D-convolution in the backward, the channel dimension needs to be flipped once more!
- If cross-correlation and convolution were 2D, e.g. you looped over the channels, no additional channel flipping is needed.

Backward pass - Gradient with respect to the weights

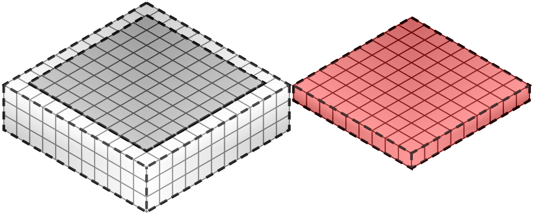


Backward pass - Gradient with respect to the weights



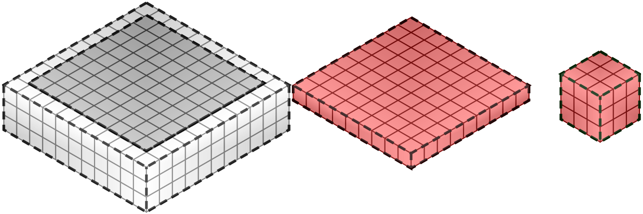
- We have to **correlate** the channels H of $\mathbf{E}_{\underline{h},n}$ with those of $\mathbf{X}$

Backward pass - Gradient with respect to the weights



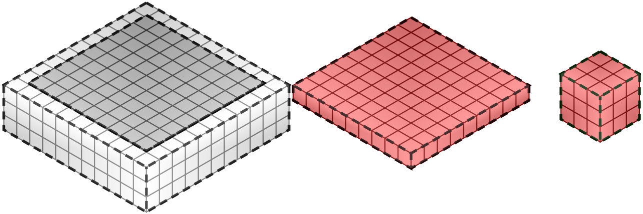
- We have to **correlate** the channels H of $\mathbf{E}_{\underline{h},n}$ with those of $\mathbf{X}$
- Pad $\mathbf{X}$ with half the kernels' width

Backward pass - Gradient with respect to the weights



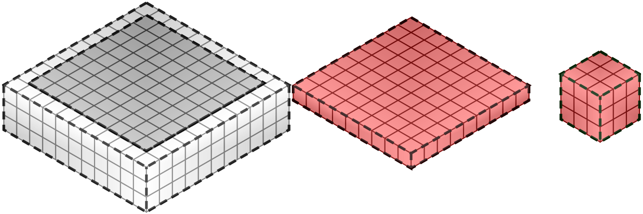
- We have to **correlate** the channels H of $\mathbf{E}_{\underline{h},n}$ with those of $\mathbf{X}$
- Pad $\mathbf{X}$ with half the kernels' width
- Compute for $s \in [1 \cdots S]$: $\mathbf{X}_s \star \mathbf{E}_{\underline{h},n}$ to receive the kernel $\nabla K_{\underline{h},S,N,M}$

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° of the kernels

Stride

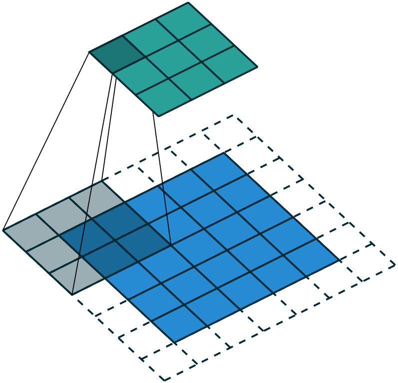


Figure: Strided convolution

Source: https://github.com/vdumoulin/conv_arithmetic

Stride

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

Stride

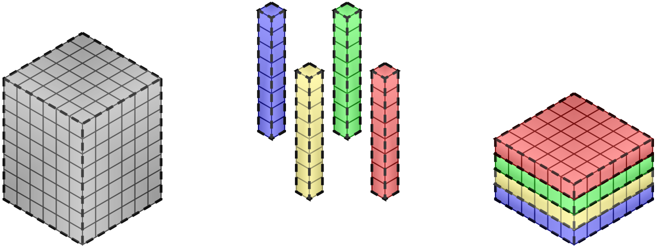
- Stride is often used to **reduce the dimension** of the input
- More mathematically stride can be seen as **correlation/convolution followed by subsampling**
- Similarly the backward pass can be calculated by **upsampling followed by convolution/correlation**

Stride

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

1x1 Convolutions

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



Overview tensor shapes

The following table shows examplatory tensor shapes for forward and backward pass for an input image of size (S, X, Y). The batch size is neglected. Be aware that the tensor of the input column still needs to be padded to achieve the desired output shape.

	Input tensor	Convolve/correlate with	Output tensor
Forward pass	(S, X, Y)	$H \times (S, N, M)$	(H, X, Y)
Gradient w.r.t weights	(S, X, Y)	$H \times (X, Y)$	(H, S, N, M)
Gradient w.r.t. lower layers	(H, X, Y)	$S \times (H, N, M)$	(S, X, Y)

Pooling layer

Forward pass max-pooling

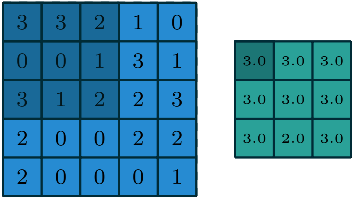


Figure: Max-pooling

Source: https://github.com/vdumoulin/conv_arithmetic

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
SCHOOL OF ENGINEERING

Forward pass max-pooling

- **Stride** is crucial now and controls amount of downsampling
- $\cdots$ and **typically as big** as the kernel size
- We need to **store the locations** of the maxima

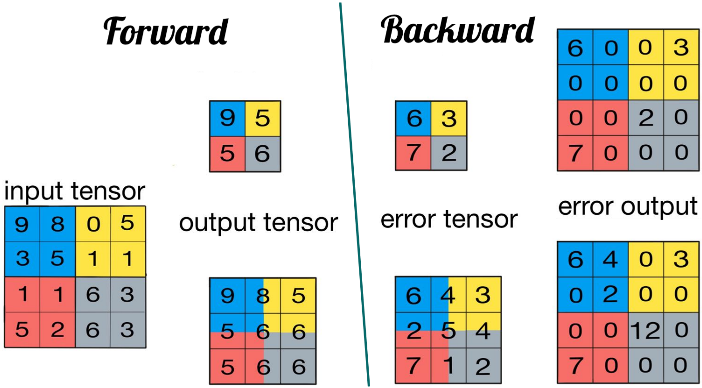
Backward pass max-pooling

Backward pass max-pooling

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
- In cases where the stride is smaller than the kernel size the error might be routed multiple times to the same location and therefore has to be summed up

Pooling with/without stride



Flatten layer

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

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?