

# Deep learning

## Backward propagation through convolutions

Siniša Šegvić

Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

## CONTENTS

- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size  $k$
- gradients of 2D convolution parameters
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- 2D convolution over 3rd order tensors
- gradients of strided convolutions
- efficient implementations

## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass



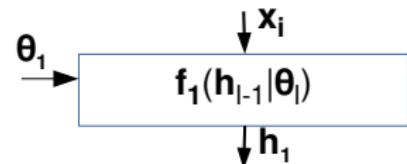
- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

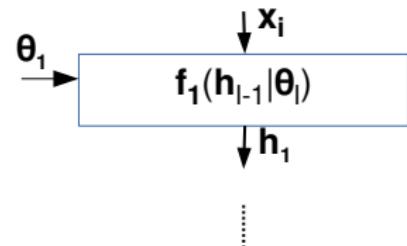


## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

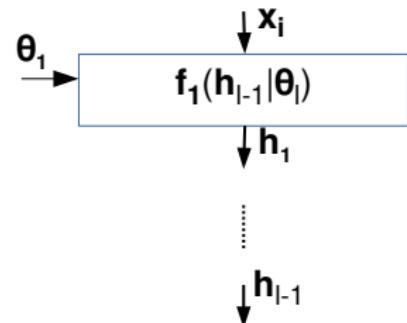


## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

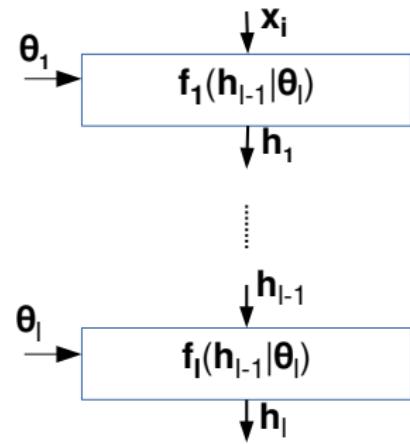


## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(h_L)$

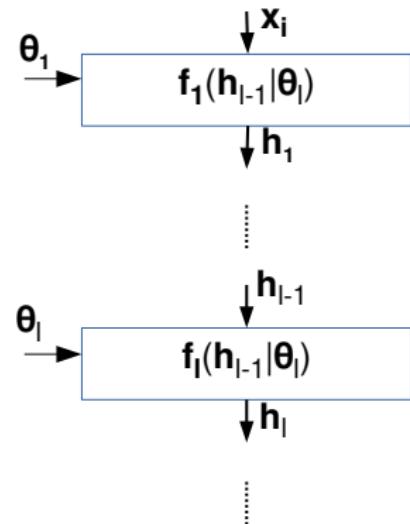


## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

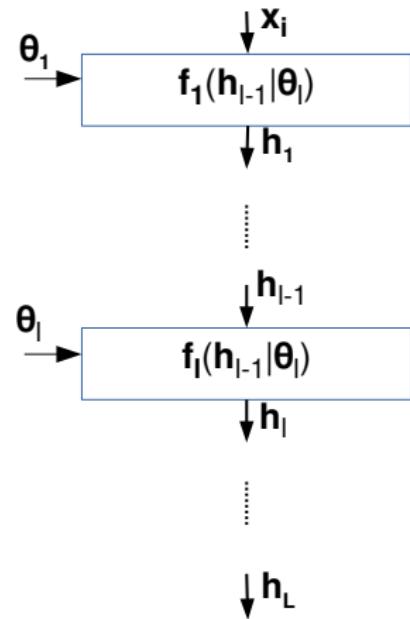


## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$



## BACKWARD PASS: DEEP MODELS

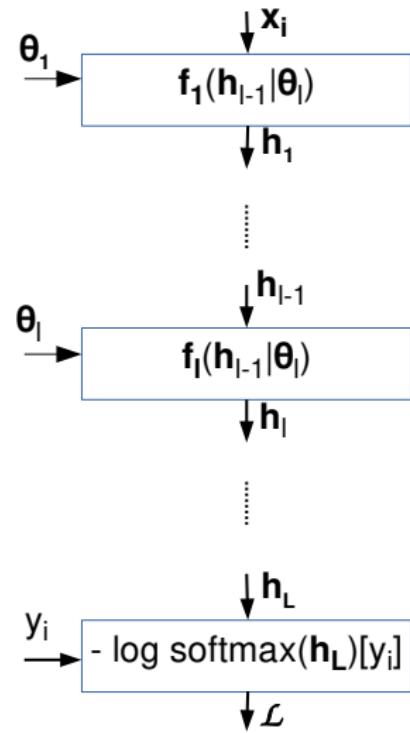
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



## BACKWARD PASS: DEEP MODELS

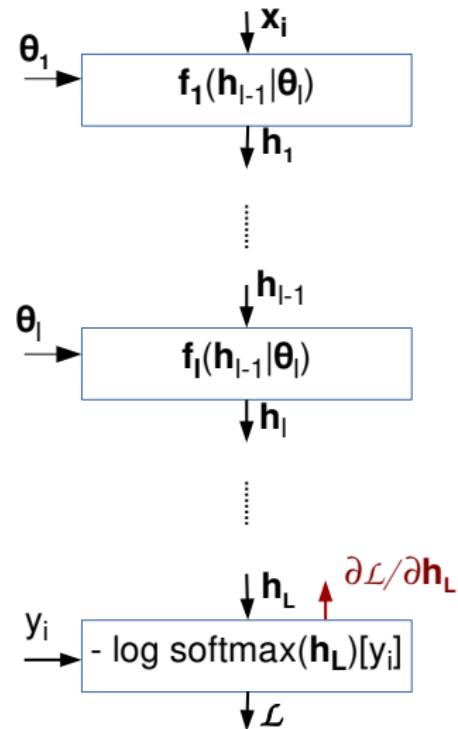
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



## BACKWARD PASS: DEEP MODELS

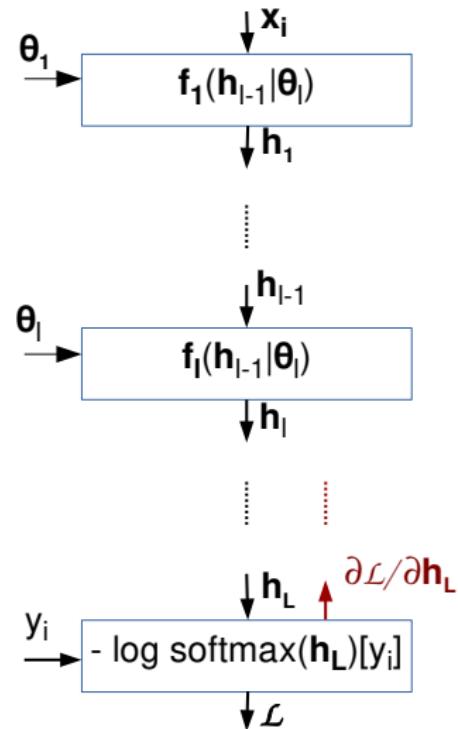
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



## BACKWARD PASS: DEEP MODELS

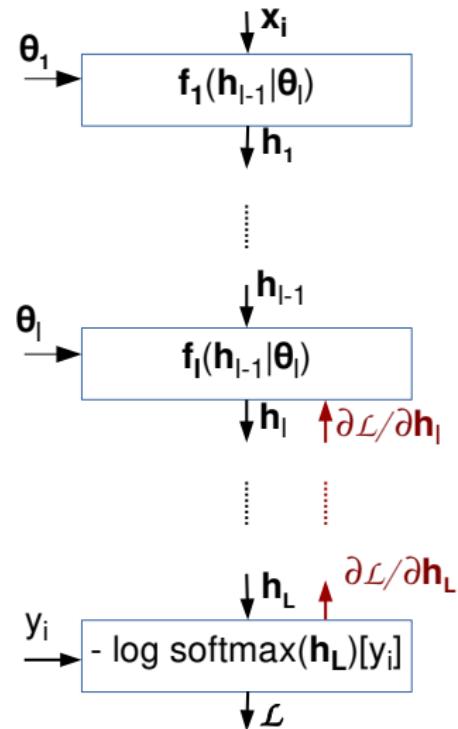
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



## BACKWARD PASS: DEEP MODELS

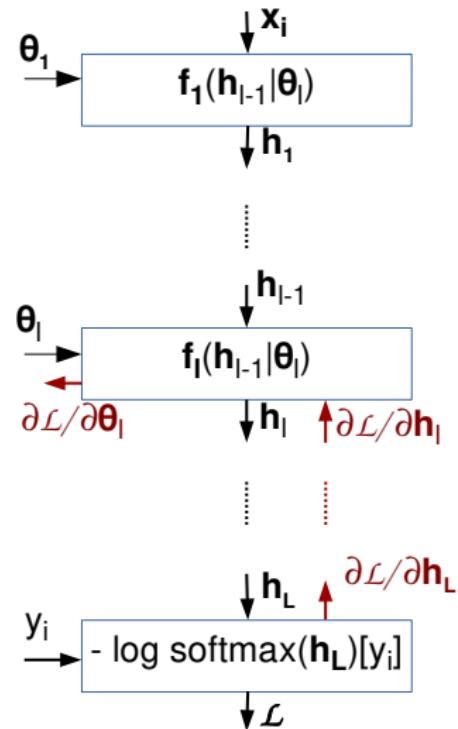
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



## BACKWARD PASS: DEEP MODELS

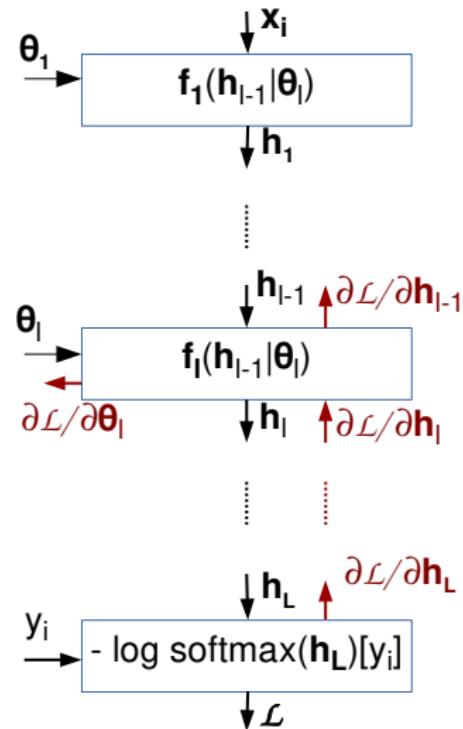
**Deep model:** composition of parametric nonlinear transformations

forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



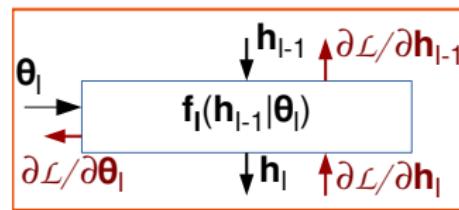
## BACKWARD PASS: DEEP MODELS

**Deep model:** composition of parametric nonlinear transformations forward pass

- input: datum  $x$ , output:  $\text{softmax}(\mathbf{h}_L)$

Backward pass

- input: prediction  $\hat{y}$ , label  $y$ , activations  $\{\mathbf{h}_l, l \in [1, L]\}$
- output: gradients of the parameters  $\partial \mathcal{L} / \partial \theta_l$



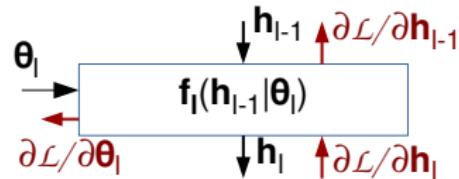
Definition of the  $l$ -th layer:

- forward pass:  $\mathbf{h}_l = f_l(\mathbf{h}_{l-1} | \theta_l)$
- gradients of the input:  
$$\partial \mathcal{L} / \partial \mathbf{h}_{l-1} = \partial \mathcal{L} / \partial \mathbf{h}_l \cdot \partial \mathbf{h}_l / \partial \mathbf{h}_{l-1}$$

## BACKWARD PASS: DEEP MODEL, DETAILS

Dimensions of the Jacobian  $\partial \mathcal{L} / \partial \mathbf{h}_l$ :

- $\partial \mathcal{L} / \partial \mathbf{h}_l$ : same as  $\mathbf{h}_l^\top$ 
  - affine transform:  $1 \times D_l$
  - convolution (Torch):  $[n, c, h, w]$



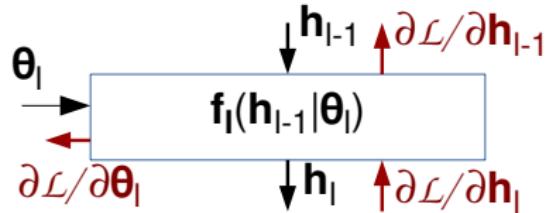
Dimensions of the Jacobian  $\partial \mathcal{L} / \partial \mathbf{W}_l = (\partial \mathcal{L} / \partial \mathbf{W}_{luv})_{u,v=1,1}^{H,W}$ :

- $\partial \mathcal{L} / \partial \mathbf{W}_l$ : same as  $\mathbf{W}_l$ 
  - affine transform:  $D_l \times D_{l-1}$
  - simple convolution:  $k \times k$
  - full convolution (Torch):  $[c_{\text{out}}, c_{\text{in}}, k, k]$
- $\partial \mathcal{L} / \partial \text{vec}(\mathbf{W}_l)$ : same as  $\text{vec}(\mathbf{W}_l)^\top$ 
  - affine transform:  $1 \times D_l \cdot D_{l-1}$
  - simple convolution:  $1 \times k \cdot k$

## BACKWARD PASS: DEEP MODEL, DETAILS (2)

Dimensions of Jacobian  $\partial \mathbf{h}_l / \partial \mathbf{h}_{l-1}$ :

- affine transform:
  - dense matrix  $D_l \times D_{l-1}$
- simple convolution:
  - sparse tensor, 4th (6th!) order
  - direct recovery impractical!

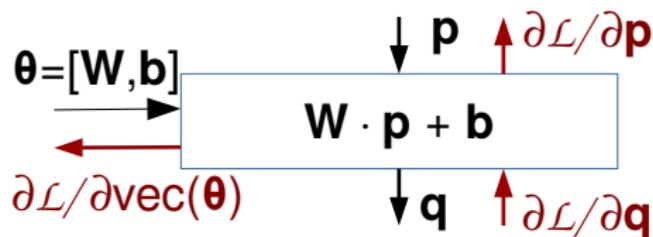


Dimensions of the Jacobian  $\partial \mathbf{h}_l / \partial \mathbf{W}_l$ :

- affine transform:
  - $\partial \mathbf{h}_l / \partial \mathbf{W}_l$ : sparse tensor of the 3rd order
  - $\partial \mathbf{h}_l / \partial \text{vec}(\mathbf{W}_l)$ : sparse matrix  $D_l \times D_l \cdot D_{l-1}$
  - practical solutions take advantage of the sparsity of  $\partial \mathbf{h}_l / \partial \mathbf{W}_l$
- simple convolution: dense tensor of the 4th (7th!) order!
  - components of this tensor correspond to the shifted inputs  $\mathbf{h}_{l-1}$
  - $$\partial \mathbf{h}_l / \partial \mathbf{W}_{luv} = \text{shift}(\mathbf{h}_{l-1}, u - o_k, v - o_k), o_k = \lfloor k/2 \rfloor + 1$$

## BACKWARD PASS: FULLY CONNECTED LAYER

Forward pass: matrix multiplication + bias addition



Gradients wrt input:  $\partial \mathcal{L} / \partial p = \partial \mathcal{L} / \partial q \cdot \partial q / \partial p = \partial \mathcal{L} / \partial q \cdot W$

Gradients wrt vectorized parameters:

$$\partial \mathcal{L} / \partial \text{vec}(W) = \partial \mathcal{L} / \partial q \cdot \partial q / \partial \text{vec}(W)$$

Problem: dimensions of the Jacobian  $\partial q / \partial \text{vec}(W)$

- $\dim(q) \times \dim(q) \cdot \dim(p)$
- eg. for MNIST:  $784 \cdot 10^2$  multiplications per datum

## BACKWARD PASS: OPTIMIZED FC LAYER, SINGLE DATUM

The Jacobian  $\partial \mathbf{q} / \partial \mathbf{W}$  is sparse due to specific pattern of dependency:

- each output  $q_j$  depends only on one row of the weight matrix -  $\mathbf{W}_{j,:}$
- on the other hand,  $\partial \mathbf{q} / \partial \mathbf{p}$  is dense: each  $q_j$  depends on each  $p_i$

Optimization 1: exploit the sparsity of  $\partial \mathbf{q} / \partial \mathbf{W}$

$$\partial \mathcal{L} / \partial \mathbf{W}_{j,:} = \partial \mathcal{L} / \partial q_j \cdot \partial q_j / \partial \mathbf{W}_{j,:} = \partial \mathcal{L} / \partial q_j \cdot \mathbf{p}^\top$$

Optimization 2: solve all rows through outer product

$$\partial \mathcal{L} / \partial \mathbf{W} = [\partial \mathcal{L} / \partial \mathbf{q}]^\top \cdot \mathbf{p}^\top$$

Textbook expression (connection with the chain rule?):

$$\partial \mathcal{L} / \partial \mathbf{W}^\top = \mathbf{p} \cdot \partial \mathcal{L} / \partial \mathbf{q}$$

Eg. for MNIST:  $784 \cdot 10$  multiplications per datum

- $10 \times$  faster than in the vectorized case
- more than 10 output dimensions  $\Rightarrow$  better improvement

## BACKWARD PASS: OPTIMIZED FC LAYER, BATCH

Batch-level loss is the expected per-datum loss:

$$\partial \mathcal{L} / \partial \mathbf{W} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}_i}{\partial \mathbf{W}}$$

Optimization 3: aggregate batch gradients through matrix multiplication

$$\partial \mathcal{L}_i / \partial \mathbf{W} = [\partial \mathcal{L}_i / \partial \mathbf{q}]^\top \cdot \mathbf{p}_i^\top$$

$\Rightarrow$

$$\partial \mathcal{L} / \partial \mathbf{W} = \frac{1}{N} \cdot \mathbf{G} \cdot \mathbf{P}, \text{ where:}$$

$$\mathbf{G}_{:,i} = [\partial \mathcal{L}_i / \partial \mathbf{q}]^\top$$

$$\mathbf{P}_{i,:} = \mathbf{p}_i^\top$$

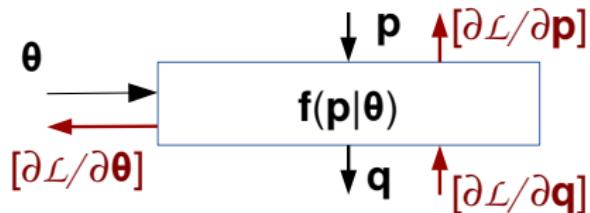
This implies the following procedure:

- forward pass caches input activations of all data  $\mathbf{P} = \{\mathbf{p}_i^\top\}$
- assume we have batch-level gradients wrt output:  $\mathbf{G} = \{\partial \mathcal{L}_i / \partial \mathbf{q}\}^\top$
- then, batch-level gradients wrt parameters are:  $\frac{1}{N} \cdot \mathbf{G} \cdot \mathbf{P}$

## BACKWARD PASS: LAYER INTERFACE

Each layer can be implemented behind the following interface:

```
class Layer: # ...
    def fwd_pass(self, p):
        self.p=p
        q=f(p, self.theta)
        return q
    def bwd_pass(self, dq):
        dp = g(dq, self.theta)
        self.dtheta = h(dq, self.p)
        return dp
```



In order to support backprop, each layer has to incorporate:

- parameters (required for gradients wrt input)
- cached input activations (required for gradients wrt parameters)
- gradients of the parameters (required for learning)

## BACKWARD PASS: ALGORITHM

We are now ready to expose a general backprop formulation

- this algorithm is invoked by `loss.backward()` in torch
- more details on reverse mode autodiff: [gavranovic18sem]

```
def backprop(L, x,y):  
    # forward pass  
    q=x # input  
    for l in L:  
        q=l.fwd_pass(q)  
    # backward pass, q=logits  
    dq=grad_loss(q, y)  
    for l in reversed(L):  
        dq=l.bwd_pass(dq)
```

This formulation accepts all layers that:

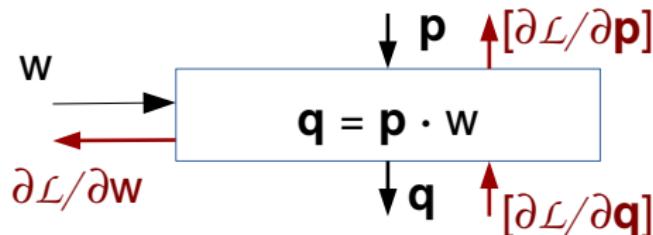
- comply with the interface Layer
- are compatible with neighbours wrt input/output dimensions

## CONTENTS

- recap: affine layers, backward propagation
- **gradients of 1D convolution parameters**
  - kernel size 1
  - kernel size  $k$
- gradients of 2D convolution parameters
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- 2D convolution over 3rd order tensors
- gradients of strided convolutions
- efficient implementations

## VECTORS: SCALAR KERNEL, EQUATIONS

Forward pass:  $\mathbf{q} = \mathbf{w} \cdot \mathbf{p}$



Gradients wrt input:

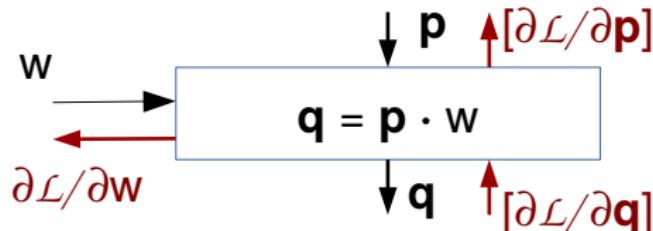
$$\begin{aligned}\partial \mathcal{L}/\partial \mathbf{p} &= \partial \mathcal{L}/\partial \mathbf{q} \cdot \partial \mathbf{q}/\partial \mathbf{p} \\ &= \partial \mathcal{L}/\partial \mathbf{q} \cdot (\mathbf{w} \cdot \mathbf{I}) \\ &= \mathbf{w} \cdot \partial \mathcal{L}/\partial \mathbf{q}\end{aligned}$$

Gradients wrt parameters:

$$\begin{aligned}\partial \mathcal{L}/\partial w &= \partial \mathcal{L}/\partial \mathbf{q} \cdot \partial \mathbf{q}/\partial w \\ &= \partial \mathcal{L}/\partial \mathbf{q} \cdot \mathbf{p}\end{aligned}$$

## VECTORS: SCALAR KERNEL, EQUATIONS (2)

Forward pass:  $\mathbf{q} = \mathbf{w} \cdot \mathbf{p}$



Gradients wrt input:  $\partial \mathcal{L}/\partial \mathbf{p} = \mathbf{w} \cdot \partial \mathcal{L}/\partial \mathbf{q}$

Gradients wrt parameters:  $\partial \mathcal{L}/\partial w = \partial \mathcal{L}/\partial \mathbf{q} \cdot \mathbf{p}$

What differences can we expect for non-trivial kernels?

- more complex gradients wrt input:  $\mathbf{q}_i$  depends on the neighbourhood  $\mathcal{N}(\mathbf{p}_i)$
- more gradients wrt parameters, but similar to what we have here

## VECTORS: SCALAR KERNEL, CODE

Our layer can be described with the following code:

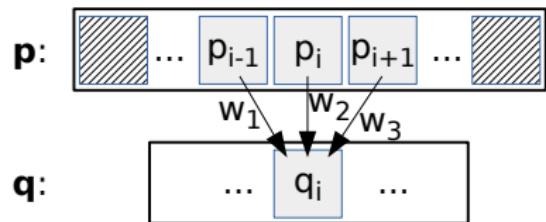
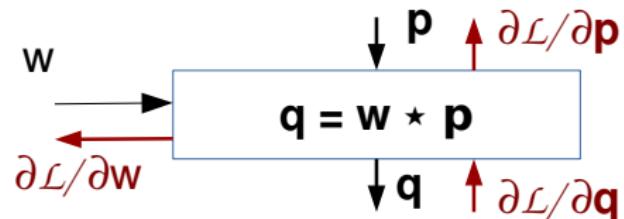
```
class LayerConv1x1: #...
    def fwd_pass(self, p):
        self.p=p
        return self.w*self.p
    def bwd_pass(self,dq):
        dp = self.w * dq
        self.dw = np.dot(dq,self.p)
        return dp
```

As before, the layer object has to aggregate activations, parameters and gradients wrt parameters

Missing pieces for all layers:

- access to parameters and gradients
- parameter initialization and update

## VECTORS: KERNEL SIZE K, FORWARD



The output  $q_i$  is a **dot product** of the crop at index  $i$  and the kernel:

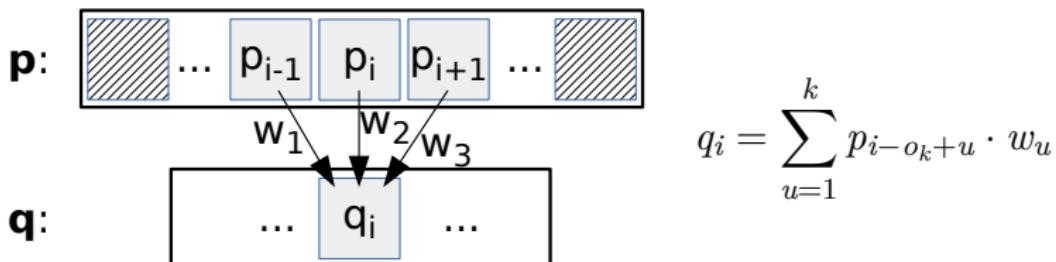
$$q_i = \text{crop}_k(\mathbf{p}, i)^\top \cdot \mathbf{w}$$

Eg. for  $k=3$ ,  $i=5$  we would have:  $q_5 = p_4 \cdot w_1 + p_5 \cdot w_2 + p_6 \cdot w_3$

Denote  $o_k = \lfloor k/2 \rfloor + 1$  (eg.  $o_3=2$ ); then the general formulation is:

$$q_i = \text{crop}_k(\mathbf{p}, i)^\top \cdot \mathbf{w} = \sum_{u=1}^k p_{i-o_k+u} \cdot w_u$$

## VECTORS: KERNEL SIZE K, BACKWARD WEIGHTS (1)



We recover the gradient wrt  $w_u$  by asking: how does  $q_i$  depend on  $w_u$ ?

$$\partial q_i / \partial w_u = p_{i-o_k+u}$$

Our gradient corresponds to the scalar product between the gradient wrt output and shifted input ("illegal access" is solved by padding!):

$$\begin{aligned}\partial \mathcal{L} / \partial w_u &= \sum_i \partial \mathcal{L} / \partial q_i \cdot \partial q_i / \partial w_u = \sum_i \partial \mathcal{L} / \partial q_i \cdot p_{i-o_k+u} \\ &= \partial \mathcal{L} / \partial \mathbf{q} \cdot \text{shift}(\mathbf{p}, -o_k + u)\end{aligned}$$

## VECTORS: KERNEL SIZE K, BACKWARD WEIGHTS (2)

Let us analyze the resulting gradients wrt parameters for k=3:

$$\partial \mathcal{L} / \partial w_1 = \sum_i \partial \mathcal{L} / \partial q_i \cdot p_{i-1} = \partial \mathcal{L} / \partial \mathbf{q} \cdot \text{shift}(\mathbf{p}, -1)$$

$$\partial \mathcal{L} / \partial w_2 = \sum_i \partial \mathcal{L} / \partial q_i \cdot p_i = \partial \mathcal{L} / \partial \mathbf{q} \cdot \text{shift}(\mathbf{p}, 0)$$

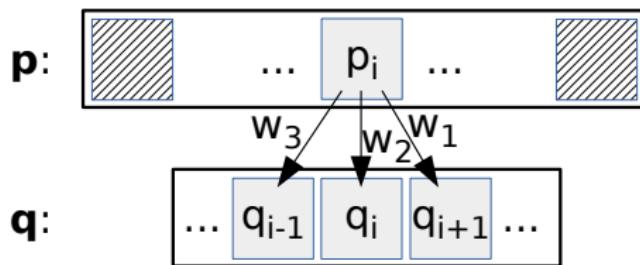
$$\partial \mathcal{L} / \partial w_3 = \sum_i \partial \mathcal{L} / \partial q_i \cdot p_{i+1} = \partial \mathcal{L} / \partial \mathbf{q} \cdot \text{shift}(\mathbf{p}, 1)$$

We observe that gradients wrt all parameters are obtained through cross correlation of the padded input with gradients wrt output:

$$\partial \mathcal{L} / \partial \mathbf{w} = \partial \mathcal{L} / \partial \mathbf{q} \star \text{pad}(\mathbf{p}, \lfloor k/2 \rfloor)$$

## VECTORS: KERNEL SIZE K, BACKWARD INPUTS (1)

We start by asking: which  $q_r$  depend on  $p_i$  and how?



$$\frac{\partial q_{i-1}}{\partial p_i} = w_3$$

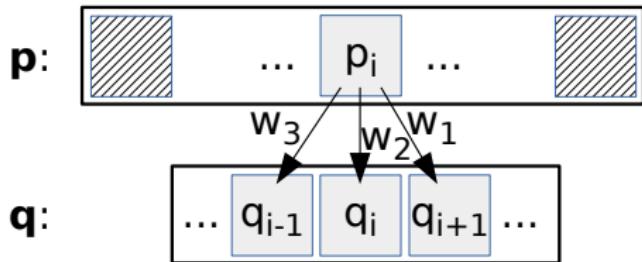
$$\frac{\partial q_{i+1}}{\partial p_i} = w_1$$

$$\frac{\partial q_{i+s}}{\partial p_i} = w_{o_k-s} = w_{2-s}$$

Let us focus on  $p_i$  as in the figure ( $o_k = \lfloor k/2 \rfloor + 1$ ,  $o_3=2$ ):

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial p_i} &= \sum_r \frac{\partial \mathcal{L}}{\partial q_r} \cdot \frac{\partial q_r}{\partial p_i} \\&= \sum_{s=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \frac{\partial \mathcal{L}}{\partial q_{i+s}} \cdot \frac{\partial q_{i+s}}{\partial p_i} \\&= \sum_{s=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \frac{\partial \mathcal{L}}{\partial q_{i+s}} \cdot w_{o_k-s}\end{aligned}$$

## VECTORS: KERNEL SIZE K, BACKWARD P (2)



$$\frac{\partial \mathcal{L}}{\partial p_i} = \sum_{s=-1}^1 \frac{\partial \mathcal{L}}{\partial q_{i+s}} \cdot w_{o_k-s}$$

We observe that gradient wrt  $i$ -th input corresponds to the **scalar product** of a crop of gradient wrt output and flipped kernel:

$$\frac{\partial \mathcal{L}}{\partial p_i} = \text{sum}(\text{crop}_k(\frac{\partial \mathcal{L}}{\partial \mathbf{q}}, i) \odot \text{flip}(\mathbf{w}))$$

Gradient wrt all inputs can be recovered by **cross correlation** of padded gradients wrt output and flipped kernel:

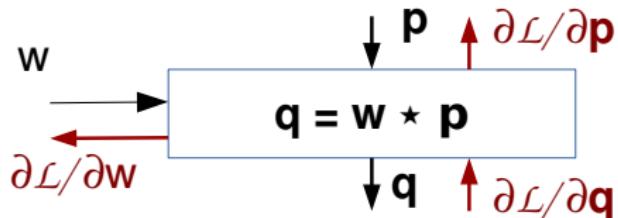
$$\frac{\partial \mathcal{L}}{\partial \mathbf{p}} = \text{flip}(\mathbf{w}) \star \text{pad}(\frac{\partial \mathcal{L}}{\partial \mathbf{q}}, \lfloor k/2 \rfloor)$$

$$\frac{\partial q_{i-1}}{\partial p_i} = w_3$$

$$\frac{\partial q_{i+1}}{\partial p_i} = w_1$$

$$\frac{\partial q_{i+s}}{\partial p_i} = w_{2-s}$$

## VECTORS: KERNEL SIZE K, SUMMARY



Forward pass:

$$q_i = \text{crop}_k(\mathbf{p}, i)^\top \cdot \mathbf{w}$$

$$\mathbf{q} = \mathbf{w} \star \mathbf{p}$$

Backward pass over parameters:

$$\partial \mathcal{L}/\partial \mathbf{w} = \partial \mathcal{L}/\partial \mathbf{q} \star \text{pad}(\mathbf{p}, \lfloor k/2 \rfloor)$$

Backward pass over inputs:

$$\partial \mathcal{L}/\partial \mathbf{p} = \text{flip}(\mathbf{w}) \star \text{pad}(\partial \mathcal{L}/\partial \mathbf{q}, \lfloor k/2 \rfloor)$$

## VECTORS: CODE

```
def my_conv1d(data, kernel):
    data_pad = torch.nn.functional.pad(data,(1,1), 'constant',0)
    return torch.nn.functional.conv1d(
        data_pad.reshape([1,1,-1]),
        kernel.reshape([1,1,-1])).squeeze()

D,K = 7,3
w = torch.tensor(np.random.randn(K), requires_grad=True)
p = torch.tensor(np.random.randn(D), requires_grad=True)
q = my_conv1d(p, w)
torch.sum(q).backward()

dLdq = torch.ones(D, dtype=torch.float64) # dL/dq_j=1 !
dLdp = my_conv1d(dLdq, torch.flip(w, (0,)))
dLdw = my_conv1d(p, dLdq)

print(dLdw, w.grad)
print(dLdp, p.grad)
```

## VECTORS: RELU

The forward pass applies the nonlinearity independently in each input:

$$q_i = \max(0, p_i)$$

Backward pass (each  $q_i$  depends only on  $p_i$ ):

$$\partial \mathcal{L} / \partial \mathbf{p} = \partial \mathcal{L} / \partial \mathbf{q} \cdot \partial \mathbf{q} / \partial \mathbf{p}$$

$$\begin{aligned} &= \partial \mathcal{L} / \partial \mathbf{q} \cdot \begin{bmatrix} \llbracket p_1 > 0 \rrbracket \\ \llbracket p_2 > 0 \rrbracket \\ \ddots \\ \llbracket p_n > 0 \rrbracket \end{bmatrix} \\ &= \partial \mathcal{L} / \partial \mathbf{q} \odot \llbracket \mathbf{p} > 0 \rrbracket \end{aligned}$$

The last expression is applicable to higher order tensors as well.

## VECTORS: GLOBAL MAX POOL

Global max pool reduces the input to scalar:

$$q = \text{maxpool}(\mathbf{p})$$

We express the backward pass in terms of the function onehot:

- example:  $\text{onehot}^4(2) = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$

$$\begin{aligned}\partial \mathcal{L} / \partial \mathbf{p} &= \partial \mathcal{L} / \partial q \cdot \partial q / \partial \mathbf{p} \\ &= \partial \mathcal{L} / \partial q \cdot [\llbracket \arg \max(\mathbf{p}) = 1 \rrbracket, \llbracket \arg \max(\mathbf{p}) = 2 \rrbracket, \dots \\ &\quad \dots, \llbracket \arg \max(\mathbf{p}) = \dim(\mathbf{p}) \rrbracket] \\ &= \partial \mathcal{L} / \partial q \cdot \text{onehot}^n(\arg \max \mathbf{p})\end{aligned}$$

The last expression is applicable to higher order tensors as well, but then instead of  $n$  we have  $\text{shape}(\mathbf{p})$

## VECTORS: MAX POOL, KERNEL K STRIDE K

The operation  $\text{maxpool}_k$  independently pools non-overlapping input **regions** of size  $k$ :

$$\mathbf{q} = \text{maxpool}_k(\mathbf{p}) .$$

The input is subsampled  $k \times$ :  $\dim(\mathbf{p}) = N \Rightarrow \dim(\mathbf{q}) = \lceil N/k \rceil$ .

Example:

$$\text{maxpool}_2 \left( \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \right) = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$$

## VECTORS: MAX POOL, KERNEL K, STRIDE K (BACK)

We express the backward pass in terms of the function  $\text{embed}_k^n(\mathbf{x}, p)$ :

- input:  $\mathbf{x}$ ,  $\dim(\mathbf{x}) = k$ ,
- output:  $\mathbf{x}'$ ,  $\dim(\mathbf{x}') = n$ , such that  $\mathbf{x}'_{[p:p+k]} = \mathbf{x}$
- property:  $\text{onehot}_1^n(p) = \text{embed}_1^n(1, p)$

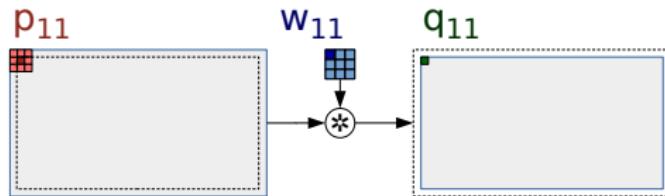
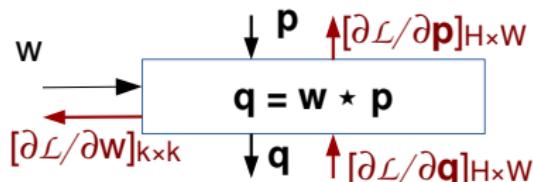
The backward pass propagates  $\partial \mathcal{L} / \partial q_i$  as in global pooling ( $\text{onehot}_1^n$ ) and **embeds** the result into  $\partial \mathcal{L} / \partial \mathbf{p}$  by means of  $\text{embed}_k^n$ :

$$\begin{aligned}\partial \mathcal{L} / \partial \mathbf{p} &= \sum_{i=0}^{\lfloor N/k \rfloor - 1} \partial \mathcal{L} / \partial q_i \cdot \text{embed}_k^N(\partial q_i / \partial \mathbf{p}_{[k \cdot i : k \cdot i + k]}, k \cdot i) \\ &= \sum_{i=0}^{\lfloor N/k \rfloor - 1} \partial \mathcal{L} / \partial q_i \cdot \text{embed}_k^N(\text{onehot}_1^k(\arg \max \mathbf{p}_{[k \cdot i : k \cdot i + k]}), k \cdot i)\end{aligned}$$

# SADRŽAJ

- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size  $k$
- **gradients of 2D convolution parameters**
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- 2D convolution over 3rd order tensors
- gradients of strided convolutions
- efficient implementations

## MATRICES: KERNEL KxK, FORWARD



Outputs of 2D convolution correspond to **scalar products** between input windows and the kernel:

$$q_{ij} = \text{sum}(\text{crop}_{k \times k}(\mathbf{p}, i, j) \odot \mathbf{w})$$

Denote  $o_k = \lfloor k/2 \rfloor + 1$  (eg.  $o_3=2$ ); then the general formulation is:

$$q_{ij} = \text{sum}(\text{crop}_{k \times k}(\mathbf{p}, i, j) \odot \mathbf{w}) = \sum_{uv} \mathbf{p}_{i-o_k+u, j-o_k+v} \cdot \mathbf{w}_{uv} .$$

Eg. for  $i=j=1$  and  $k=3$  ( $o_k=2$ ) we have:

$$\mathbf{q}_{11} = \sum_{uv=1}^3 \mathbf{p}_{u-1, v-1} \cdot \mathbf{w}_{uv}$$

## MATRICES: KERNEL KXX, BACKWARD

Dimensions of tensors that participate in the backward pass:

- input (gradient wrt output):  $\partial \mathcal{L} / \partial \mathbf{q}$  -  $H \times W$
- output (gradient wrt input):  $\partial \mathcal{L} / \partial \mathbf{p}$  -  $H \times W$ 
  - we assume padding in the forward pass
- output (gradient wrt parameters):  $[\partial \mathcal{L} / \partial \mathbf{w}]$  -  $(k \times k)$

Important: the outputs  $\mathbf{q}$  do not depend on all inputs  $\mathbf{p}$

General formulation of gradients wrt one input pixel:

$$\partial \mathcal{L} / \partial p_{ij} = \partial \mathcal{L} / \partial \text{vec}(\mathbf{q}) \cdot \partial \text{vec}(\mathbf{q}) / \partial p_{ij} = \sum_r \partial \mathcal{L} / \partial q_r \cdot \partial q_r / \partial p_{ij}$$

Key questions for recovering gradients wrt input:

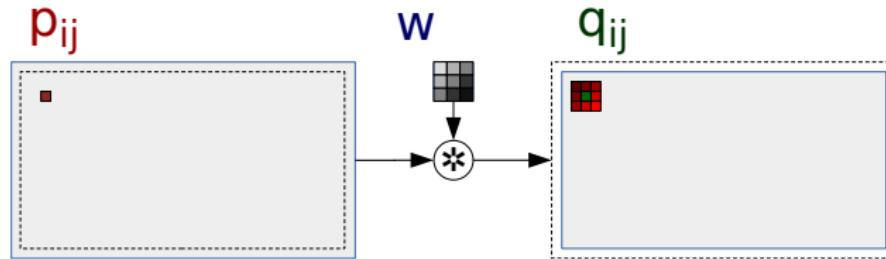
- which  $q_r$  depend on  $p_{ij}$ ?
- in which  $r$  we have  $\partial q_r / \partial p_{ij} \neq 0$ ?

## MATRICES: KERNEL K×K, BACKWARD, INPUTS

Assume that  $p$  has only one pixel  $p_{ij} \neq 0$

Then, the output  $q$  will have the following non-zero pixels:

$$q_{i+s,j+t} = p_{ij} \cdot w_{o_k-s,o_k-t}, \text{ where } s, t \in -\lfloor k/2 \rfloor .. \lfloor k/2 \rfloor$$



Red pixels in  $q$ : sphere of influence of  $p_{ij}$

- $\mathbf{N}_{q_{ij}}^{k \times k}$ :  $k \times k$  neighbourhood at  $q_{ij}$
- in these pixels  $\partial q_r / \partial p_{ij} \neq 0!$
- more precisely:  $\partial q_{i+s,j+t} / \partial p_{ij} = \mathbf{w}_{o_k-s,o_k-t}$
- lower-right pixel in  $\mathbf{N}_{q_{ij}}^{k \times k}$  interacts with the upper-left element of  $w$ :
  - $\partial q_{i+1,j+1} / \partial p_{ij} = \mathbf{w}_{1,1}$ , for  $k=3$ .

## MATRICES: KERNEL KXK, BACKWARD, INPUTS (2)

What happens when we consider all activations of  $p$ ?

- each element of  $q$  sums contributions from  $p_{ij}$
- consequently, the gradients  $\partial q_{i+s,j+t} / \partial p_{ij}$  stay the same.

We are ready to formulate the gradients wrt input:

$$\begin{aligned}\partial \mathcal{L} / \partial p_{ij} &= \sum_{s,t} \partial \mathcal{L} / \partial q_{i+s,j+t} \cdot \partial q_{i+s,j+t} / \partial p_{ij} \\ &= \sum_{s,t} \partial \mathcal{L} / \partial q_{i+s,j+t} \cdot w_{o_k-s, o_k-t}\end{aligned}$$

They are a **scalar product** between a square crop of gradients wrt output and the 2D flipped kernel (this is analogous to the 1D case):

$$\partial \mathcal{L} / \partial p_{ij} = \text{sum}(\text{crop}_{k \times k}(\partial \mathcal{L} / \partial \mathbf{q}, i, j) \odot \text{flip2d}(\mathbf{w}))$$

## MATRICES: KERNEL KXK, BACKWARD, INPUTS (3)

We have shown that we obtain gradients wrt input by sliding a flipped kernel over gradients wrt output:

$$\partial \mathcal{L} / \partial p_{ij} = \text{sum}(\text{crop}_{k \times k}(\partial \mathcal{L} / \partial \mathbf{q}, i, j) \odot \text{flip2d}(\mathbf{w}))$$

This can be reformulated as cross correlation of the gradient wrt output with the flipped kernel  $\text{flip2d}(\mathbf{w})$ .

in order to recover the gradient in all input pixels, we can pad the gradients wrt output with  $\lfloor k/2 \rfloor$  zeros from each side:

$$\partial \mathcal{L} / \partial \mathbf{p} = \text{flip2d}(\mathbf{w}) \star \text{pad}(\partial \mathcal{L} / \partial \mathbf{q}, \lfloor k/2 \rfloor)$$

If no padding were used during forward pass, then we would have to pad  $k-1$  zeros (from each side)

## MATRICES: KERNEL KxK, TRANSPOSED CONVOLUTION

We shall denote the last operation as [transposed convolution](#):

- the input is padded with  $\lfloor k/2 \rfloor$  zeros
- the kernel is flipped over both axes
- deprecated terms: deconvolution, backward convolution

We introduce a new operator:  $\mathbf{w} \star^T \mathbf{x} \triangleq \text{flip2d}(\mathbf{w}) \star \text{pad}(\mathbf{x}, \lfloor k/2 \rfloor)$

Later we shall get to know more details about transposed convolution:

- when applying 2D convolution to 3rd order tensors, the kernel must be flipped only over spatial axes
- transposed convolution wrt strided forward pass can be used to upsample the input.

## MATRICES: K×K, BACKWARD, PARAMETERS

If only one  $w_{uv} \neq 0$ , then each output depends on a single input:

$$q_{ij} = p_{i-o_k+u, j-o_k+v} \cdot w_{uv}, \forall i, j$$

Each weight affects all outputs → we aggregate all output locations:

$$\begin{aligned}\partial \mathcal{L} / \partial w_{uv} &= \sum_{ij} \partial \mathcal{L} / \partial q_{ij} \cdot \partial q_{ij} / \partial w_{uv} \\ &= \sum_{ij} \partial \mathcal{L} / \partial q_{ij} \cdot p_{i-o_k+u, j-o_k+v}\end{aligned}$$

This corresponds to a reduction of a Hadamard (elementwise) product between the gradient wrt output and shifted inputs:

$$\begin{aligned}\partial \mathcal{L} / \partial w_{uv} &= \sum_{ij} \partial \mathcal{L} / \partial q_{ij} \cdot p_{i-o_k+u, j-o_k+v} \\ &= \text{sum}(\text{shift}(\mathbf{p}, -o_k + u, -o_k + v) \odot \partial \mathcal{L} / \partial \mathbf{q})\end{aligned}$$

## MATRICES: K×K, BACKWARD, PARAMETERS (2)

When we consider all kernel elements  $w$ :

- activations  $q$  correspond to a sum of contributions from all  $w_{uv}$
- the gradients  $\partial q_{ij} / \partial w_{uv}$  remain the same as on the last slide

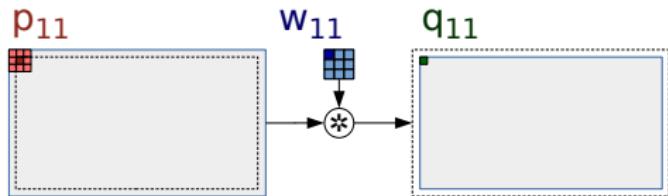
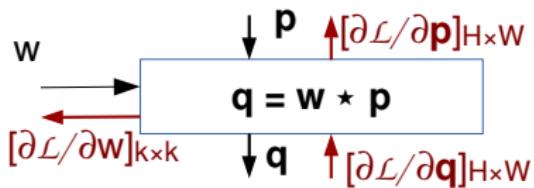
We recover the gradient wrt all weights by cross correlating the gradient wrt output with the padded input:

$$\partial \mathcal{L} / \partial \mathbf{w} = \partial \mathcal{L} / \partial \mathbf{q} \star \text{pad}(\mathbf{p}, [k/2])$$

This equation holds when the input and output have the same resolution due to padding in the forward pass.

If forward pass does not use padding than the gradients wrt parameters do not require padding.

## MATRICES: K×K, BACKWARD, SUMMARY



Forward pass:

$$q_{ij} = \text{sum}(\text{crop}_{k \times k}(\mathbf{p}, i, j) \odot \mathbf{w})$$

$$\mathbf{q} = \mathbf{w} * \mathbf{p}$$

Backward pass over parameters:

$$\partial \mathcal{L} / \partial \mathbf{w} = \partial \mathcal{L} / \partial \mathbf{q} * \text{pad}(\mathbf{p}, \lfloor k/2 \rfloor)$$

Backward pass over inputs:

$$\partial \mathcal{L} / \partial \mathbf{p} = \text{flip2d}(\mathbf{w}) * \text{pad}(\partial \mathcal{L} / \partial \mathbf{q}, \lfloor k/2 \rfloor)$$

## MATRICES: PROBLEM

We consider a convolutional model for classifying greyscale images:

- convolution  $3 \times 3$  with no bias, no padding, and ReLU activation
  - two output feature maps determined with kernels  $w_1$  and  $w_2$
- global max-pooling
- fully connected layer with no bias ( $W$ ), softmax

Initialization:

- $w_{121} = -1$ ,  $w_{122} = 1$ , all other elements of  $w_1$  are 0
- $w_{232} = -1$ ,  $w_{222} = 1$ , all other elements of  $w_2$  are 0
- $W_{11} = W_{22} = 1$ , all other elements of  $W$  are 0

The input is a  $4 \times 4$  image  $x$ ,  $x_{22} = 1$ ,  $x_{33} = 1$ , all other elements are 0

Determine the gradients of negative log-likelihood with  $y=2$ .

## MATRICES: SOLUTION

```
import torch
import numpy as np

x = np.zeros([4,4])
x[1,1] = x[2,2] = 1
x = x.reshape([1,*x.shape])
x = torch.tensor(x, requires_grad=True)

w1 = np.zeros([3,3])
w1[1,0], w1[1,1] = -1,1
w1 = w1.reshape([1,1,*w1.shape])
w1 = torch.tensor(w1, requires_grad=True)

w2 = np.zeros([3,3])
w2[2,1], w2[1,1] = -1,1
w2 = w2.reshape([1,1,*w2.shape])
w2 = torch.tensor(w2, requires_grad=True)

W = torch.tensor(np.eye(2), requires_grad=True)
```

## MATRICES: SOLUTION (2)

```
f1 = torch.nn.functional.conv2d(x, w1)
f1r = torch.nn.functional.relu(f1)
f1m = torch.nn.functional.max_pool2d(f1r, [2,2])

f2 = torch.nn.functional.conv2d(x, w2)
f2r = torch.nn.functional.relu(f2)
f2m = torch.nn.functional.max_pool2d(f2r, [2,2])

h = torch.concat([f1m, f2m]).squeeze()
s = W @ h

for t in [f1, f2, h, s]: t.retain_grad()
L = torch.nn.functional.cross_entropy(s, torch.tensor(1))
L.backward()

for t in [s, h, W, f1, f2, w1, w2]:
    print(t.data, t.grad.data, sep='\n', end='\n\n')
```

## MATRICES: SOLUTION (3)

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, y = 2$$

$$\frac{\partial L}{\partial \mathbf{w}_1} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$$

$$\frac{\partial L}{\partial \mathbf{w}_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -0.5 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

Interpretation: after the update the logit of the branch 1 will decrease while the logit of the branch 2 will increase.

## CONTENTS

- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size  $k$
- gradients of 2D convolution parameters
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- **2D convolution over 3rd order tensors**
- gradients of strided convolutions
- efficient implementations

## MULTI-CHANNEL 2D CONV: FORWARD

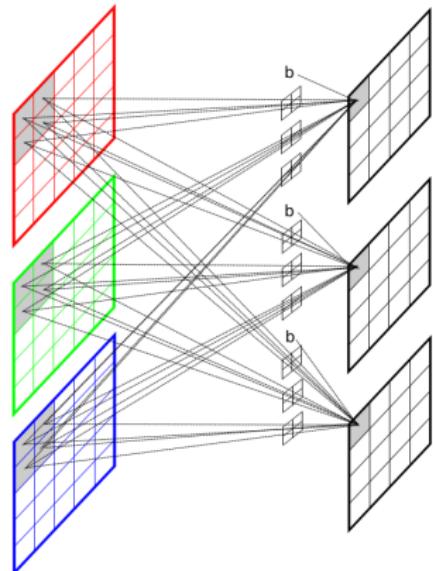
We keep the standard syntax:

$$\mathbf{q} = \mathbf{w} \star \mathbf{p} + \text{broadcast}(\mathbf{b})$$

The output  $\mathbf{q}^{(g)}$  convolves the corresponding slices of the input and the  $g$ -th kernel, and aggregates the results:

$$\mathbf{q}^{(g)} = \sum_f \mathbf{w}^{(g,f)} \star \mathbf{p}^{(f)}$$

$$q_{ij}^{(g)} = \sum_{fuv} p_{i-o_k+u, j-o_k+v}^{(f)} \cdot w_{uv}^{(g,f)}$$



Gradients wrt bias correspond to aggregated gradients wrt output:

$$\partial \mathcal{L} / \partial b_g = \sum_{ij} \partial \mathcal{L} / \partial q_{ij}^{(g)} \cdot \partial q_{ij}^{(g)} / \partial b_g = \sum_{ij} \partial \mathcal{L} / \partial q_{ij}^{(g)}$$

## MULTI-CHANNEL 2D CONV: BACKWARD, INPUTS

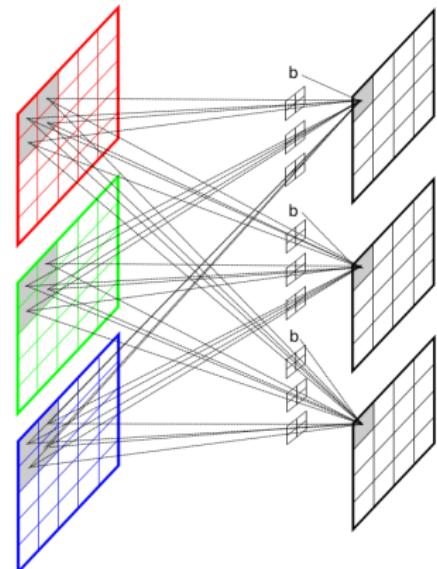
The input  $\mathbf{p}^{(f)}$  influences each slice  $\mathbf{q}^{(g)}$  through convolution with  $\mathbf{w}^{(g,f)}$

Hence, gradients wrt the  $s$ th slice of input aggregate gradients from all  $\mathbf{q}^{(g)}$ :

$$\begin{aligned}\partial \mathcal{L} / \partial \mathbf{p}_{ij}^{(f)} &= \\ \sum_g \sum_{s,t} \partial \mathcal{L} / \partial \mathbf{q}_{i+s,j+t}^{(g)} &\cdot w_{o_k-s, o_k-t}^{(g,f)}\end{aligned}$$

As before, these gradients are recovered through convolution with flipped kernel:

$$\partial \mathcal{L} / \partial \mathbf{p}^{(f)} = \sum_g \text{flip2d}(\mathbf{w}^{(g,f)}) \star \text{pad}(\partial \mathcal{L} / \partial \mathbf{q}^{(g)}, \lfloor k/2 \rfloor)$$



This corresponds to multi-channel transposed convolution:

$$\mathbf{w} \star^\top \mathbf{x} \triangleq \sum_g \text{flip2d}(\mathbf{w}^{(g,f)}) \star \text{pad}(\mathbf{x}^{(g)}, \lfloor k/2 \rfloor)$$

## MULTI-CHANNEL 2D CONV: BACKWARD, PARAMETERS

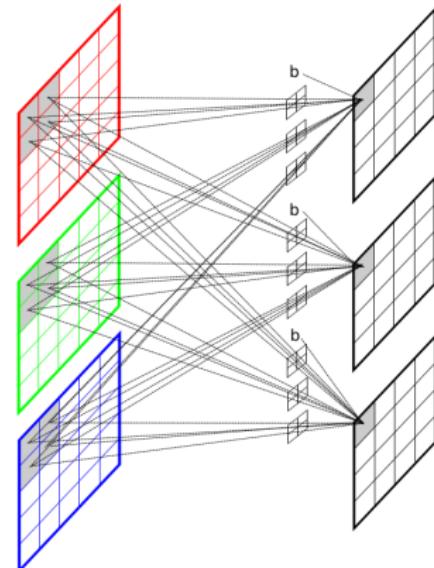
The output tensor aggregates contributions from the corresponding slices of the kernel and the input:

- the  $f$ -th slice of the  $g$ -th kernel influences the  $g$ -th output slice through convolution with the  $s$ th input slice

Thus we recover the gradients wrt parameters separately for each kernel slice.

The final expression is exactly the same as in the single-channel case:

$$\partial \mathcal{L} / \partial \mathbf{w}^{(g,f)} = \partial \mathcal{L} / \partial \mathbf{q}^{(g)} \star \text{pad}(\mathbf{p}^{(f)}, \lfloor k/2 \rfloor)$$

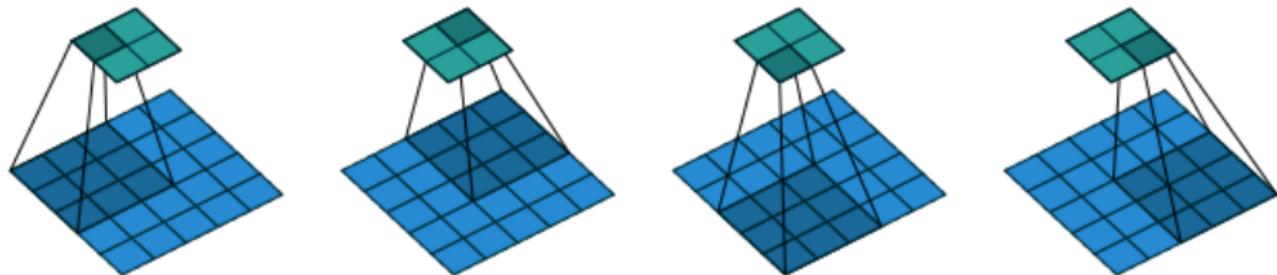


## CONTENTS

- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size  $k$
- gradients of 2D convolution parameters
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- 2D convolution over 3rd order tensors
- **gradients of strided convolutions**
- efficient implementations

## STRIDE: FORWARD

Convolution with stride  $s$  retains each  $s$ th activation of the default convolution:



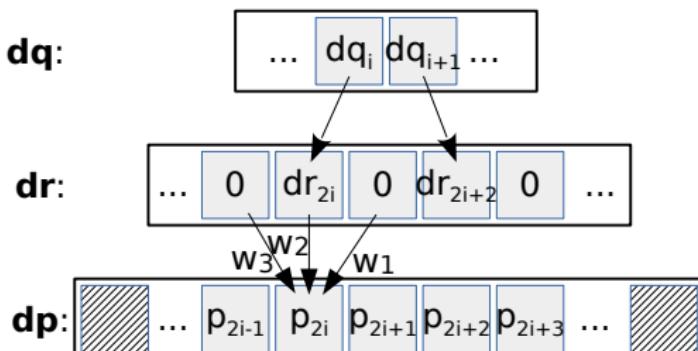
[dumoulin16arxiv]

This results in the same effect as if we applied a default convolution followed by subsampling  $\times s$

## STRIDE: BACKWARD

Idea: strided convolution = default convolution  $\circ$  subsampling  $\times s$ :

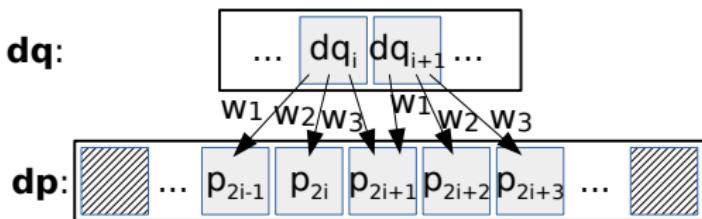
- when backprop passes from subsampling to the convolution, the gradient of each neglected convolution output becomes zero
- subsequently, the backprop would proceed towards convolution inputs through transposed convolution
- thus, backprop through strided convolution decreases the receptive field  $s$  times wrt convolution with the same kernel
  - this operation is also known as fractional convolution (Theano)



## STRIDE: INTEGRATED BACKWARD

Idea: accumulate contributions from each output ( $t$  denotes stride!):

- concurrent writing to the same location is bad for efficient parallelism, but here it may be a method of choice



Such operation can not be expressed with convolution

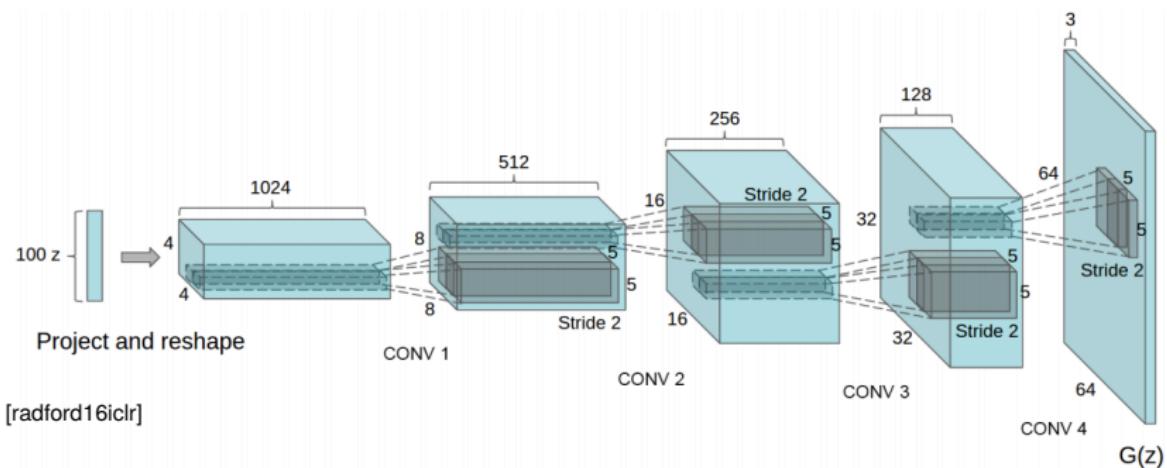
- ⇒ requires special implementation

$$\begin{aligned}\partial \mathcal{L} / \partial \mathbf{p}^{(f)} &= \sum_{ij} \sum_g \partial \mathcal{L} / \partial q_{ij}^{(g)} \cdot \partial q_{ij}^{(g)} / \partial \mathbf{p}^{(f)} \\ &= \sum_{ij=0,0}^{\text{shape}(\mathbf{q})} \sum_g \text{embed}_{k \times k}^{H \times W} (\partial \mathcal{L} / \partial q_{ij}^{(g)} \cdot \mathbf{w}^{(g,f)}, [s \cdot i, s \cdot j])\end{aligned}$$

## STRIDE: INTEGRATED BACKWARD (2)

Transposed strided convolution in the forward pass:

- standard trick for upsampling a convolutional representation
- may be useful for recognizing small images (eg. CIFAR)
- may be useful in generative models with low-dimensional latent representation (eg. DCGAN, VAE)



## STRIDE: BACKWARD SUMMARY

Strided transposed convolution used for forward pass:

- can be used to upsample a representation
  - useful for generative models (eg. CVAE, DCGAN) and decoders for dense recognition (eg. FPN, SwiftNet)
- can be used for recognizing small images (eg. CIFAR)

Alternative: bilinear upsampling followed by default convolution

- difference: holes in upsampled representation filled with interpolated features (instead with zeros)
- larger complexity but similar speed in practice
- this worked very well in our experiments [kreso17iccvw]

## CONTENTS

- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size  $k$
- gradients of 2D convolution parameters
  - kernel size  $k \times k$
  - problem: backward pass through a convolutional problem
- 2D convolution over 3rd order tensors
- gradients of strided convolutions
- **efficient implementations**

## IMPLEMENTATION: OVERVIEW

Idea: implement convolution as matrix multiplication

- capitalize GEMM for efficient evaluation

Two implementation approaches:

- flatten input and output activations (bad idea)
- flatten the weights of the convolutional kernel (good idea)

Recent implementations use the Winograd algorithm

- total number of multiplications reduced for 50%
- Nervana [lavin15arxiv], cudnn v5.1+

# IMPLEMENTATION: FLATTENING ACTIVATIONS

$$\mathbf{q}_{3 \times 3} = \mathbf{w}_{3 \times 3} * \mathbf{p}_{5 \times 5}$$

$$\text{vec}(\mathbf{q}_{3 \times 3}) = \text{doubly\_circ}(\mathbf{w}_{3 \times 3}) \cdot \text{vec}(\mathbf{p}_{5 \times 5})$$

$$\begin{bmatrix} q_{11} \\ q_{12} \\ q_{13} \\ q_{21} \\ q_{22} \\ q_{23} \\ q_{31} \\ q_{32} \\ q_{33} \end{bmatrix} = \left[ \begin{array}{cccccccccccccccccccccccc} w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . & . & . & . & . \\ . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . & . & . & . & . \\ . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & w_{11} & w_{12} & w_{13} & . & . & w_{21} & w_{22} & w_{23} & . & . & w_{31} & w_{32} & w_{33} & . & . & . & . & . & . & . \end{array} \right] \begin{bmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{14} \\ p_{15} \\ p_{21} \\ p_{22} \\ p_{23} \\ p_{24} \\ p_{25} \\ p_{31} \\ p_{32} \\ p_{33} \\ p_{34} \\ p_{35} \\ p_{41} \\ p_{42} \\ p_{43} \\ p_{44} \\ p_{51} \\ p_{52} \\ p_{53} \\ p_{54} \\ p_{55} \end{bmatrix}$$

Temporal complexity:  $O(W^2 H^2)$  vs  $O(WHk^2)$

## IMPLEMENTATION: FLATTENING THE KERNEL

Rearrange the input so convolution corresponds to matrix multiplication:

$$\text{vec}(\mathbf{q}) = \text{im2row}(\mathbf{p}) \cdot \text{vec}(\mathbf{w})$$

We show the input  $\mathbf{X} = \mathbf{p}$ , and the rearrangement  $\mathbf{X}^{\text{ROWS}} = \text{im2row}(\mathbf{p})$ :

x

$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$
$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{25}$
$x_{31}$	$x_{32}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{41}$	$x_{42}$	$x_{43}$	$x_{44}$	$x_{45}$
$x_{51}$	$x_{52}$	$x_{53}$	$x_{54}$	$x_{55}$

X ROWS

$x_{11}$	$x_{12}$	$x_{15}$	$x_{21}$	$x_{22}$	$x_{23}$	$x_{31}$	$x_{32}$	$x_{33}$
$x_{12}$	$x_{13}$	$x_{14}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{32}$	$x_{33}$	$x_{34}$
$x_{13}$	$x_{14}$	$x_{15}$	$x_{23}$	$x_{24}$	$x_{25}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{33}$	$x_{34}$	$x_{35}$	$x_{43}$	$x_{44}$	$x_{45}$	$x_{53}$	$x_{54}$	$x_{55}$

Temporal complexity:  $O(WHk^2)$  vs  $O(WHk^2)$

Spatial complexity:  $O(WHk^2)$  vs  $O(WH)$

## IMPLEMENTATION: FLATTENING THE KERNEL

Rearrange the input so convolution corresponds to matrix multiplication:

$$\text{vec}(\mathbf{q}) = \text{im2row}(\mathbf{p}) \cdot \text{vec}(\mathbf{w})$$

We show the input  $\mathbf{X} = \mathbf{p}$ , and the rearrangement  $\mathbf{X}^{\text{ROWS}} = \text{im2row}(\mathbf{p})$ :

$\mathbf{X}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$
$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{25}$
$x_{31}$	$x_{32}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{41}$	$x_{42}$	$x_{43}$	$x_{44}$	$x_{45}$
$x_{51}$	$x_{52}$	$x_{53}$	$x_{54}$	$x_{55}$

$\mathbf{X}^{\text{ROWS}}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{21}$	$x_{22}$	$x_{23}$	$x_{31}$	$x_{32}$	$x_{33}$
$x_{12}$	$x_{13}$	$x_{14}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{32}$	$x_{33}$	$x_{34}$
$x_{13}$	$x_{14}$	$x_{15}$	$x_{23}$	$x_{24}$	$x_{25}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{33}$	$x_{34}$	$x_{35}$	$x_{43}$	$x_{44}$	$x_{45}$	$x_{53}$	$x_{54}$	$x_{55}$

Temporal complexity:  $O(WHk^2)$  vs  $O(WHk^2)$

Spatial complexity:  $O(WHk^2)$  vs  $O(WH)$

## IMPLEMENTATION: FLATTENING THE KERNEL

Rearrange the input so convolution corresponds to matrix multiplication:

$$\text{vec}(\mathbf{q}) = \text{im2row}(\mathbf{p}) \cdot \text{vec}(\mathbf{w})$$

We show the input  $\mathbf{X} = \mathbf{p}$ , and the rearrangement  $\mathbf{X}^{\text{ROWS}} = \text{im2row}(\mathbf{p})$ :

$\mathbf{X}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$
$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{25}$
$x_{31}$	$x_{32}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{41}$	$x_{42}$	$x_{43}$	$x_{44}$	$x_{45}$
$x_{51}$	$x_{52}$	$x_{53}$	$x_{54}$	$x_{55}$

$\mathbf{X}^{\text{ROWS}}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{21}$	$x_{22}$	$x_{23}$	$x_{31}$	$x_{32}$	$x_{33}$
$x_{12}$	$x_{13}$	$x_{14}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{32}$	$x_{33}$	$x_{34}$
$x_{13}$	$x_{14}$	$x_{15}$	$x_{23}$	$x_{24}$	$x_{25}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{33}$	$x_{34}$	$x_{35}$	$x_{43}$	$x_{44}$	$x_{45}$	$x_{53}$	$x_{54}$	$x_{55}$

Temporal complexity:  $O(WHk^2)$  vs  $O(WHk^2)$

Spatial complexity:  $O(WHk^2)$  vs  $O(WH)$

## IMPLEMENTATION: FLATTENING THE KERNEL

Rearrange the input so convolution corresponds to matrix multiplication:

$$\text{vec}(\mathbf{q}) = \text{im2row}(\mathbf{p}) \cdot \text{vec}(\mathbf{w})$$

We show the input  $\mathbf{X} = \mathbf{p}$ , and the rearrangement  $\mathbf{X}^{\text{ROWS}} = \text{im2row}(\mathbf{p})$ :

$\mathbf{X}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{14}$	$x_{15}$
$x_{21}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{25}$
$x_{31}$	$x_{32}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{41}$	$x_{42}$	$x_{43}$	$x_{44}$	$x_{45}$
$x_{51}$	$x_{52}$	$x_{53}$	$x_{54}$	$x_{55}$

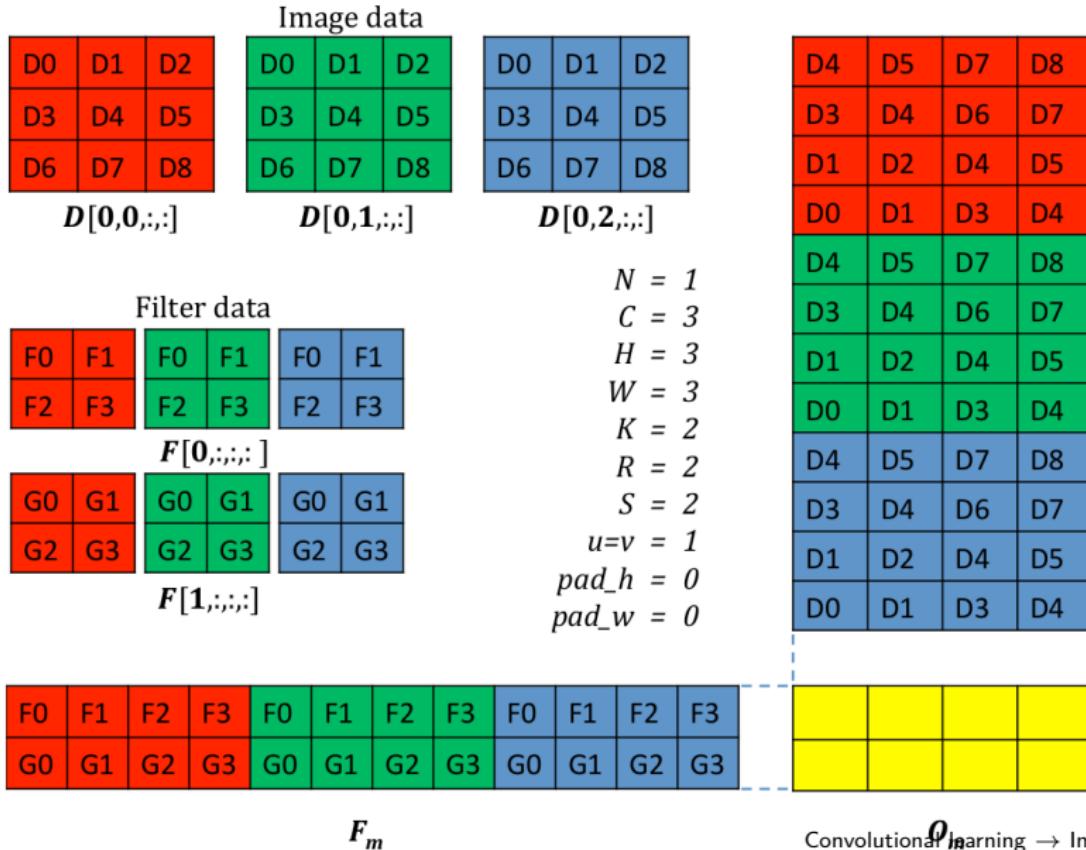
$\mathbf{X}^{\text{ROWS}}$

$x_{11}$	$x_{12}$	$x_{13}$	$x_{21}$	$x_{22}$	$x_{23}$	$x_{31}$	$x_{32}$	$x_{33}$
$x_{12}$	$x_{13}$	$x_{14}$	$x_{22}$	$x_{23}$	$x_{24}$	$x_{32}$	$x_{33}$	$x_{34}$
$x_{13}$	$x_{14}$	$x_{15}$	$x_{23}$	$x_{24}$	$x_{25}$	$x_{33}$	$x_{34}$	$x_{35}$
$x_{33}$	$x_{34}$	$x_{35}$	$x_{43}$	$x_{44}$	$x_{45}$	$x_{53}$	$x_{54}$	$x_{55}$

Temporal complexity:  $O(WHk^2)$  vs  $O(WHk^2)$

Spatial complexity:  $O(WHk^2)$  vs  $O(WH)$

# IMPLEMENTATION: FLATTENING THE KERNEL (3RD ORDER INPUT)



## IMPLEMENTATION: BACKWARD, PARAMETERS

We have expressed convolution as matrix multiplication:

$$\text{vec}(\mathbf{q}) = \text{im2row}(\mathbf{p}) \cdot \text{vec}(\mathbf{w})$$

The above equation clearly shows that the rearranged input corresponds to the partial derivative of the output wrt parameters:

$$\frac{\partial \text{vec}(\mathbf{q})}{\partial \text{vec}(\mathbf{w})} = \text{im2row}(\mathbf{p})$$

Finally, the gradients wrt parameters can be recovered by multiplying the gradients wrt output with the rearranged input:

$$\begin{aligned}\partial \mathcal{L} / \partial \text{vec}(\mathbf{w}) &= \partial \mathcal{L} / \partial \text{vec}(\mathbf{q}) \cdot \partial \text{vec}(\mathbf{q}) / \partial \text{vec}(\mathbf{w}) \\ &= \partial \mathcal{L} / \partial \text{vec}(\mathbf{q}) \cdot \text{im2row}(\mathbf{p})\end{aligned}$$

## IMPLEMENTATION: BACKWARD, INPUTS

Recall the expression to recover gradients wrt input:

$$\partial \mathcal{L} / \partial \mathbf{p} = \text{flip2d}(\mathbf{w}) \star \text{pad}(\partial \mathcal{L} / \partial \mathbf{q}, \lfloor k/2 \rfloor)$$

We first rearrange gradients wrt output into  $\mathbf{G}_{\mathbf{q}}^{\text{ROWS}}$ :

$$\mathbf{G}_{\mathbf{q}}^{\text{ROWS}} = \text{im2row}(\text{pad}(\partial \mathcal{L} / \partial \mathbf{q}, \lfloor k/2 \rfloor))$$

Now the convolution can be implemented as matrix multiplication:

$$\mathbf{G}_{\mathbf{p}}^{\text{VEC}} = \mathbf{G}_{\mathbf{q}}^{\text{ROWS}} \cdot \text{vec}(\text{flip2d}(\mathbf{w}))$$

Finally, we recover the result by reshaping  $\mathbf{G}_{\mathbf{p}}^{\text{VEC}}$ :

$$\partial \mathcal{L} / \partial \mathbf{p} = \text{reshape}(\mathbf{G}_{\mathbf{p}}^{\text{VEC}}, [H, W])$$

If we consider a strided convolution, then:

- $\mathbf{G}_{\mathbf{q}}^{\text{ROWS}}$  corresponds to a sparse matrix
- better performance through integrated transposed convolution

## ACKNOWLEDGEMENT

These lectures came out from research funded by Croatian Science Foundation through project I-2433-2014 MultiCLoD.



<http://multiclod.zemris.fer.hr>