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

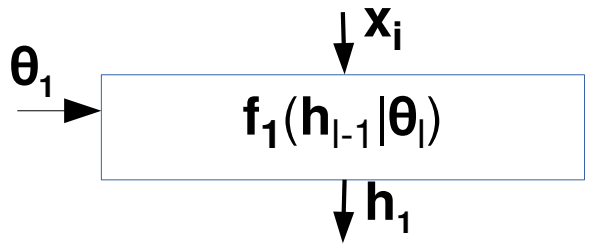
- input: datum $\mathbf{x}$, output: $\mathrm{softmax}(\mathbf{h}_L)$

BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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

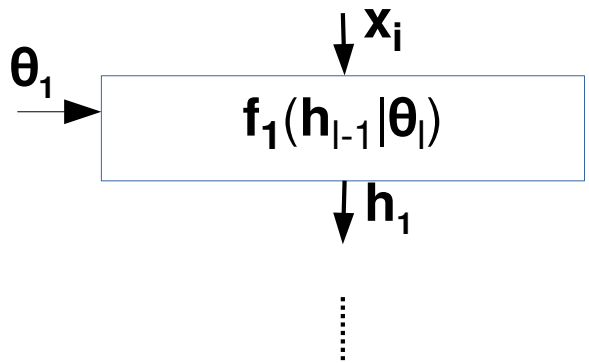


BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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

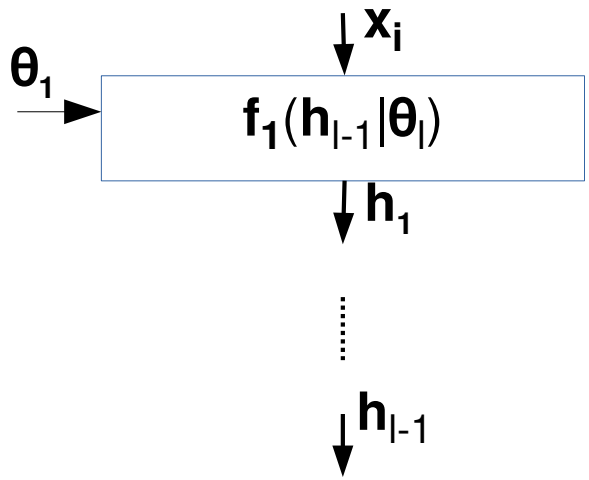


BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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

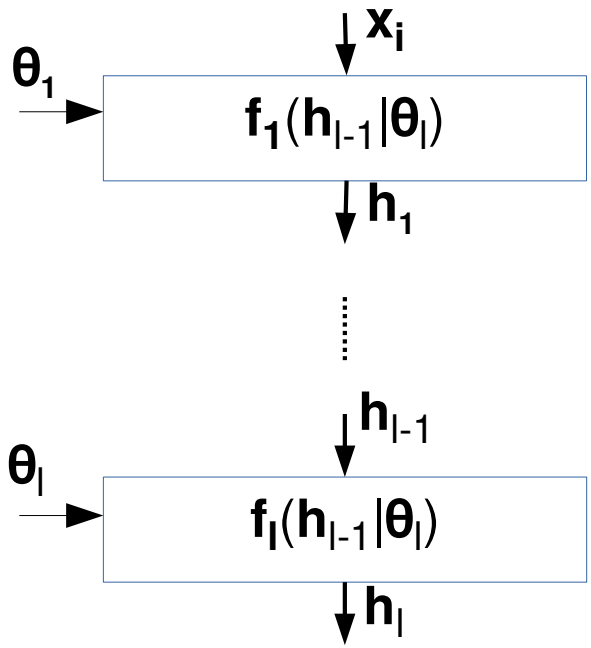


BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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

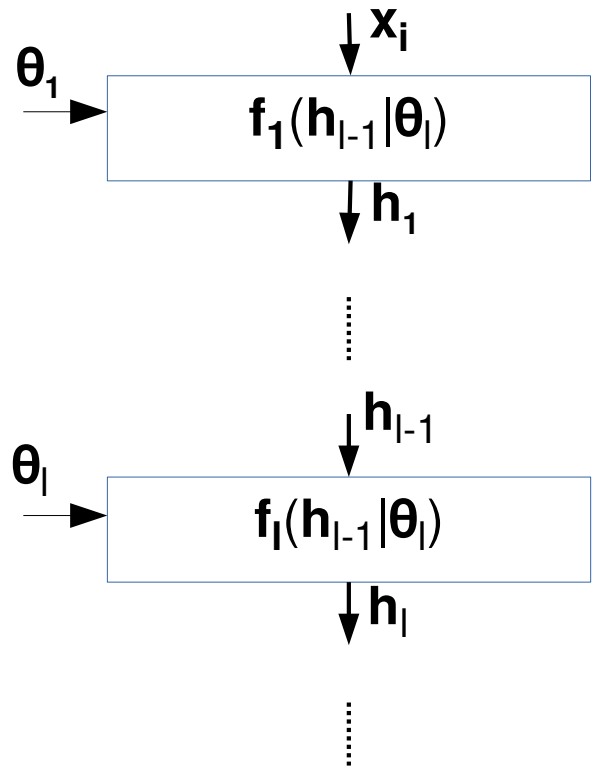


BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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

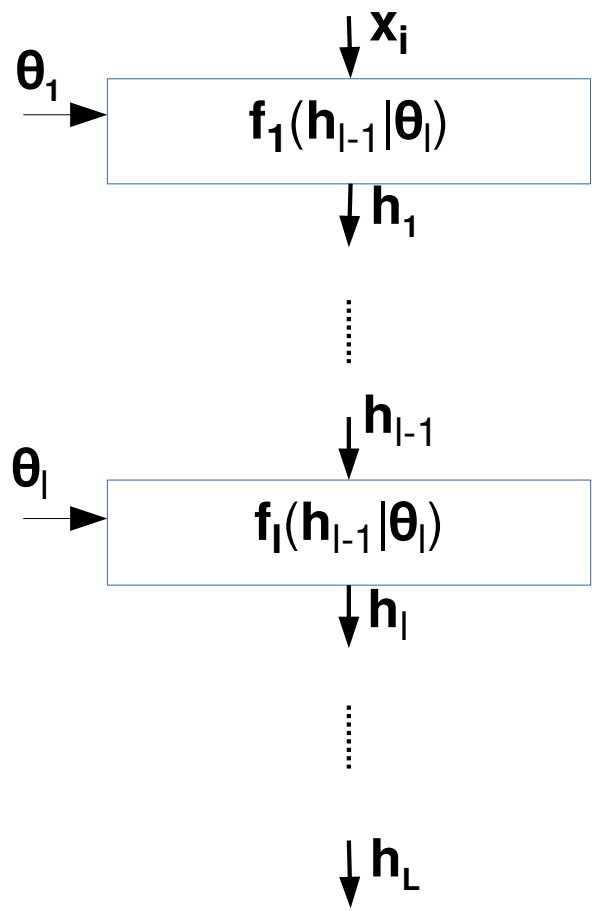


BACKWARD PASS: DEEP MODELS

Deep model: composition of parametric nonlinear transformations

forward pass

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



BACKWARD PASS: DEEP MODELS

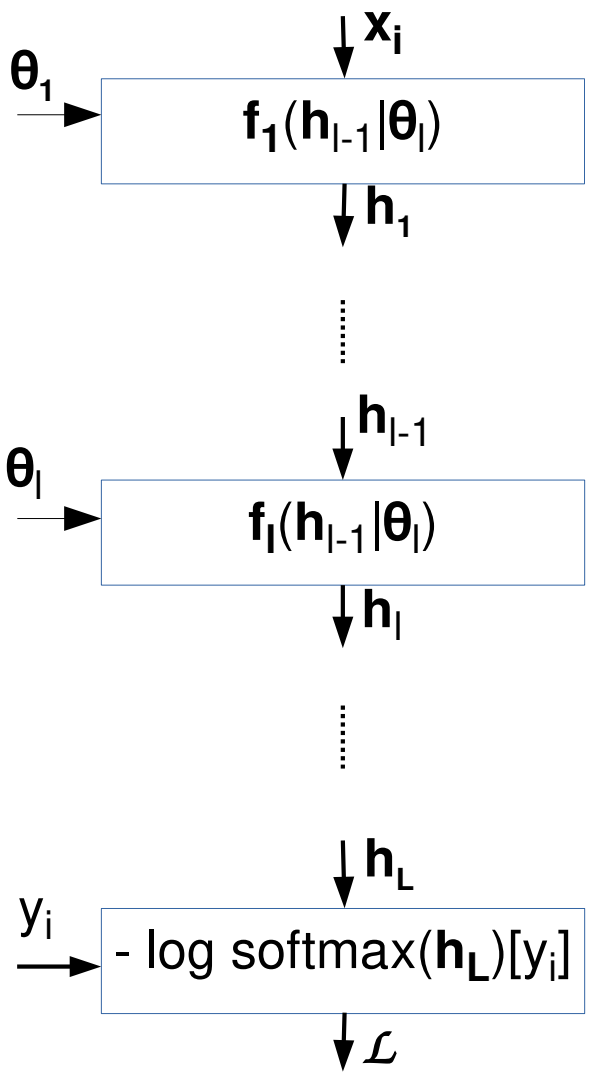
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

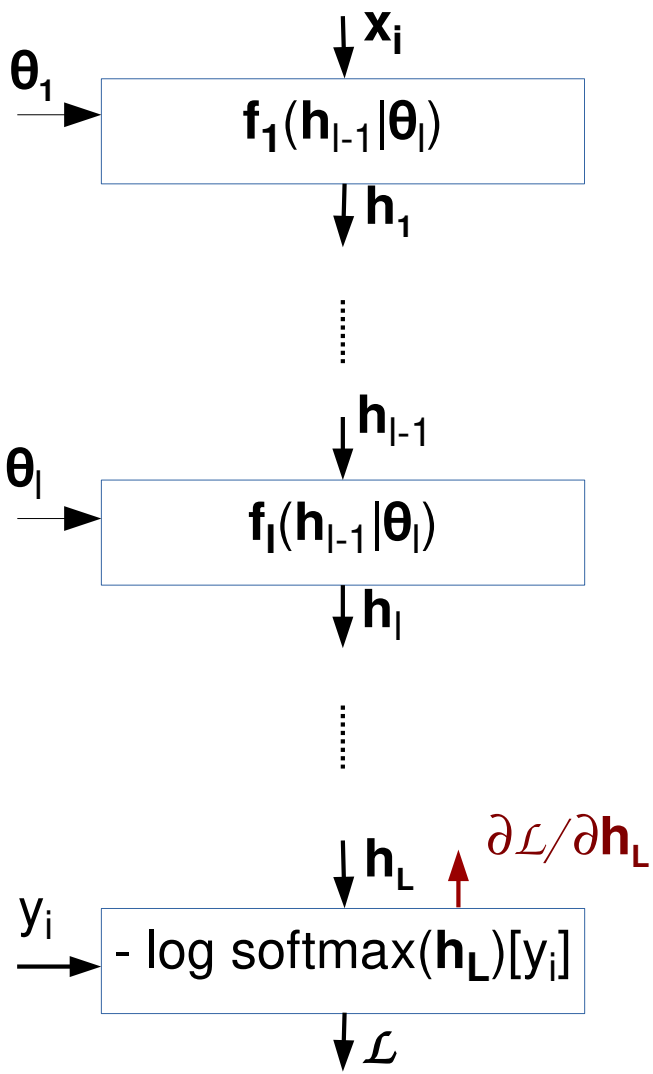
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

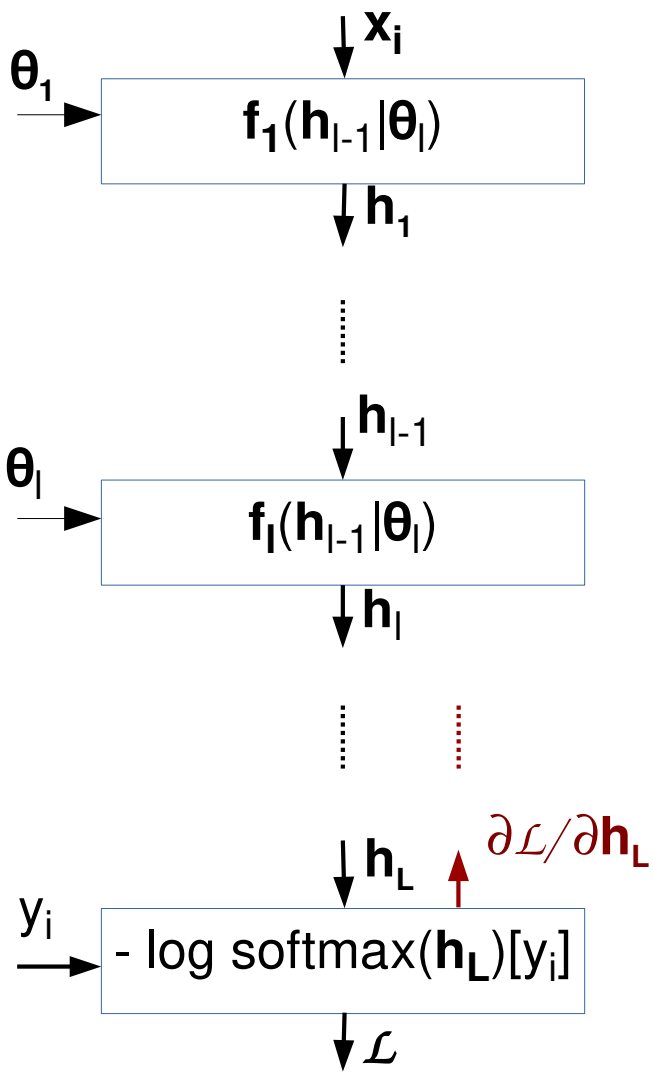
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

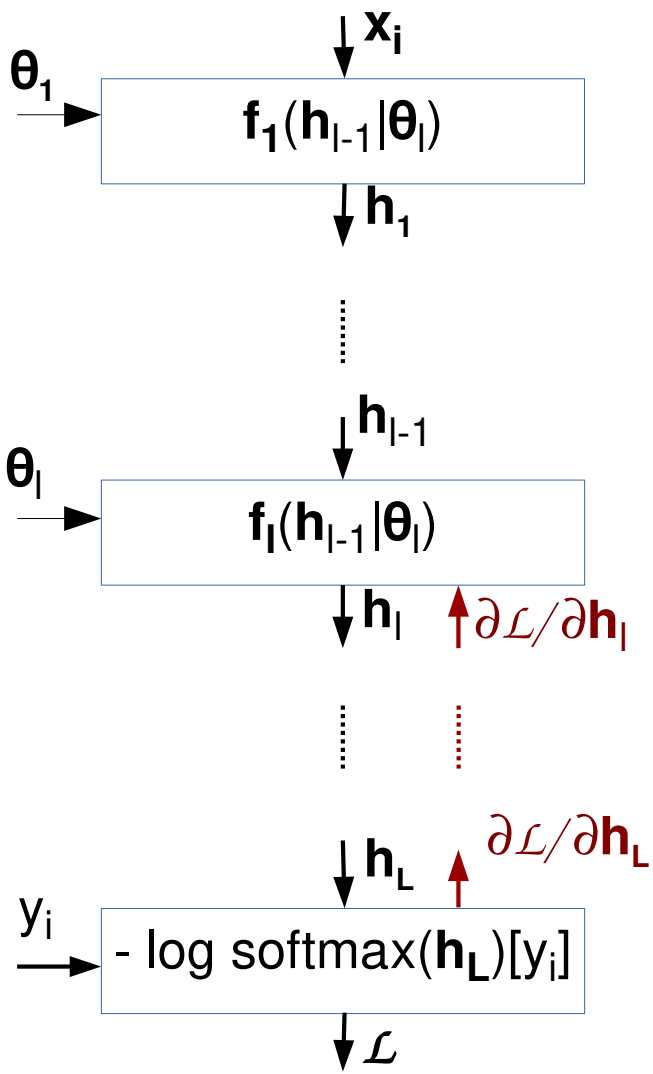
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

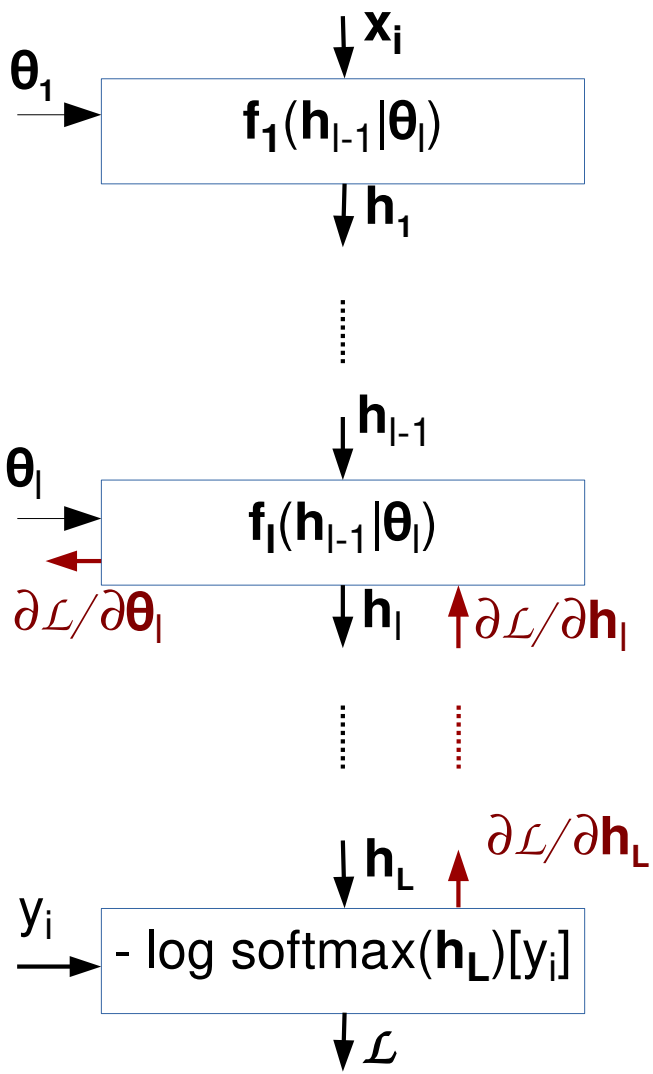
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

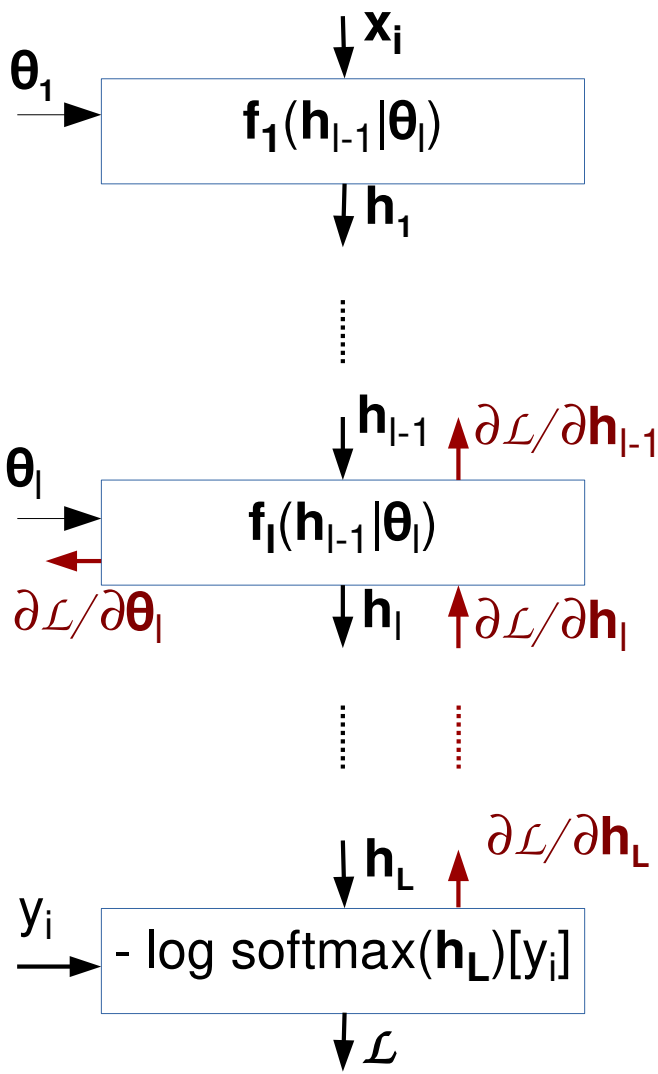
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



BACKWARD PASS: DEEP MODELS

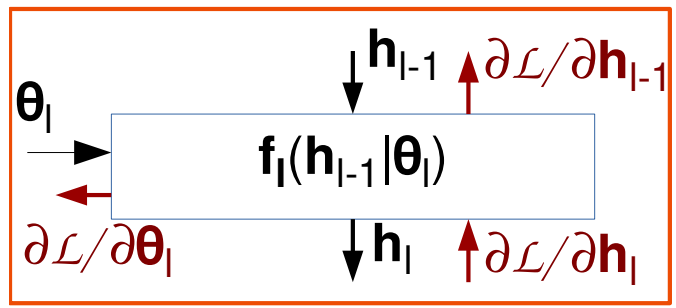
Deep model: composition of parametric nonlinear transformations

forward pass

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

Backward pass

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



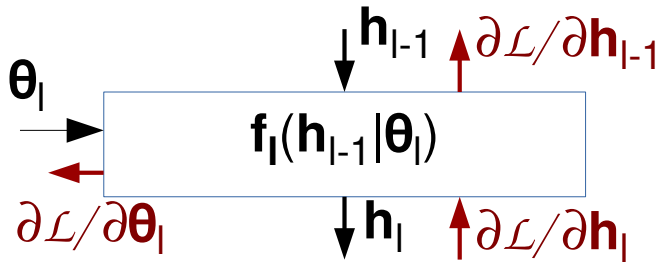
Definition of the l-th layer:

- forward pass: $\mathbf{h}_l = \mathbf{f}_l(\mathbf{h}_{l-1} \mid \boldsymbol{\theta}_l)$
- 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): $[n, c, h, w]$



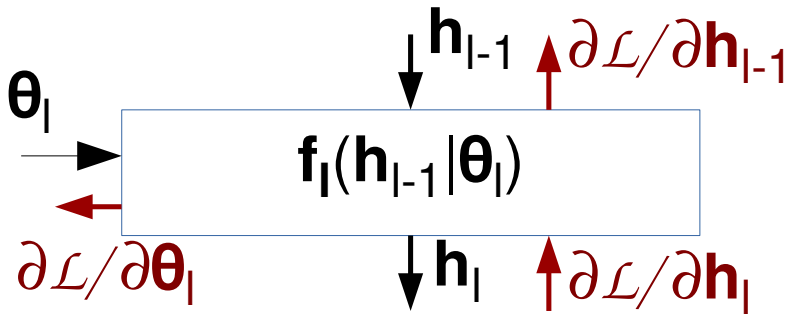
Dimensions of the Jacobian $\partial \mathcal{L} / \partial \boldsymbol{W}_l = (\partial \mathcal{L} / \partial \boldsymbol{W}_{luv})$:

- $\partial \mathcal{L} / \partial \boldsymbol{W}_l$: 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: DUBOKI MODEL, DETALJI (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!



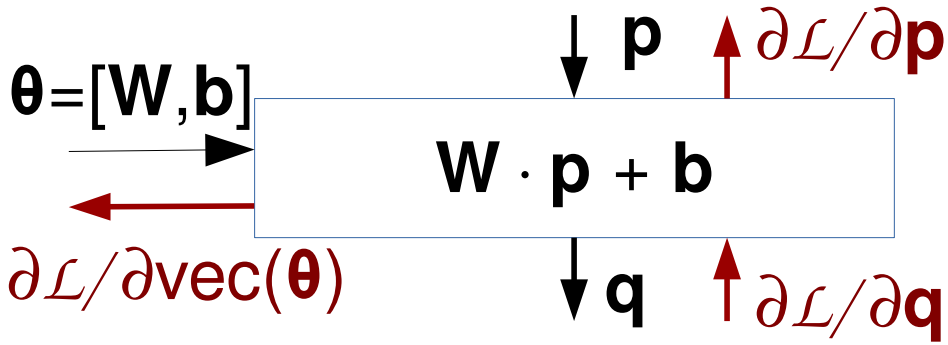
Dimensions of the Jacobian $\partial \mathbf{h}_l / \partial \mathbf{W}_l$:

- affine transform:
  - $\partial \mathbf{h}_l / \partial \mathbf{W}_l$: sparse tensor of the 3rd order
  - $\partial \mathbf{h}_l / \partial \mathrm{vec}(\mathbf{W}_l)$: 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 $\mathbf{h}_{l-1}$

$$\partial \mathbf{h}_l / \partial \mathbf{W}_{luv} = \mathrm{shift}(\mathbf{h}_{l-1}, u - o_k, v - o_k),\ o_k = \lfloor k/2 \rfloor + 1$$

BACKWARD PASS: FULLY CONNECTED LAYER

Forward pass: matrix multiplication + bias addition



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

Gradients wrt vectorized parameters:

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

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

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

BACKWARD PASS: OPTMIZED 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: OPTMIZED FC LAYER, BATCH

Per-batch 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

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

$$\Rightarrow$$

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

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

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

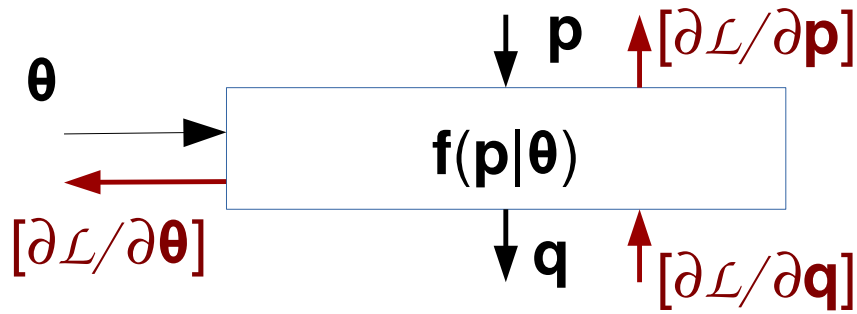
This implies the following procedure:

- forward pass caches input activations of all data $\mathbf{P} = \left\{ \mathbf{p}_i^\top \right\}$
- 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 INERFACE

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: ALGORITAM

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:

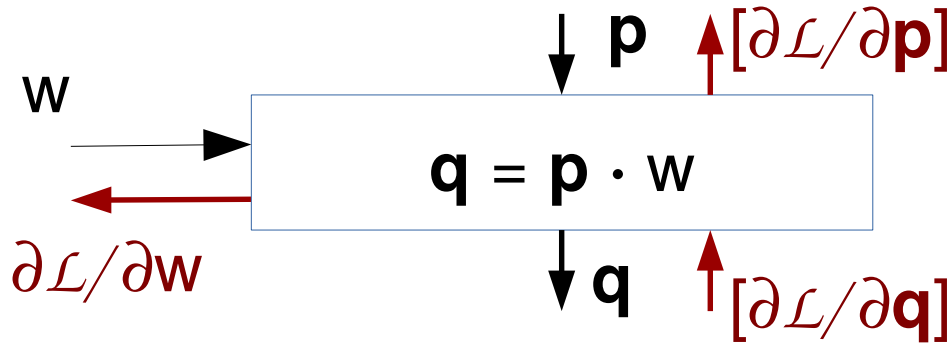
- comply with the interface `Layer`
- are compatible with neighbours wrt input/output dimensions

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



Gradients wrt input:

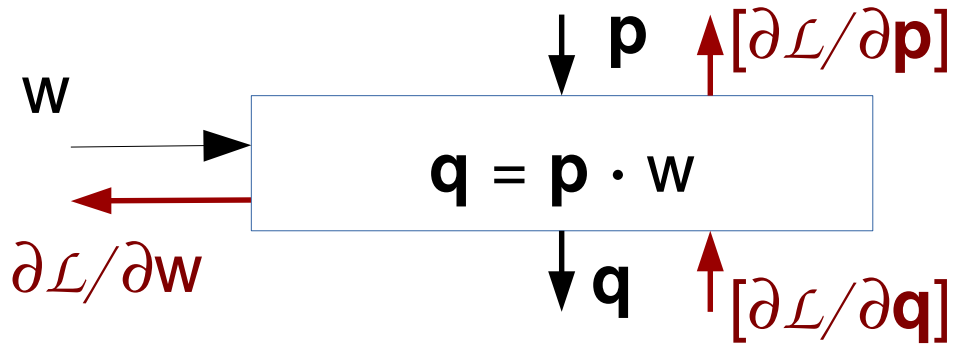
$$\begin{aligned}\partial\mathcal{L}/\partial\mathbf{p} &= \partial\mathcal{L}/\partial\mathbf{q} \cdot \partial\mathbf{q}/\partial\mathbf{p} \\ &= \partial\mathcal{L}/\partial\mathbf{q} \cdot (w \cdot \mathbf{I}) \\ &= w \cdot \partial\mathcal{L}/\partial\mathbf{q}\end{aligned}$$

Gradijents 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: $\mathbf{q} = w \cdot \mathbf{p}$



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

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:

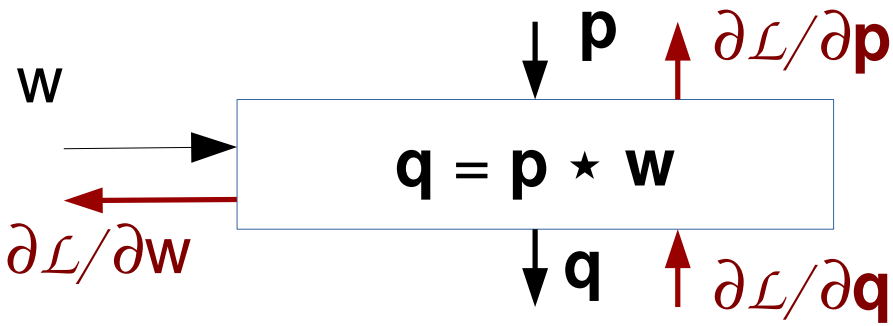
```
class LayerConv1x1: #...
  def fwd_pass(self, p):
    self.p=p
    return self.w*self.p
  def bwd_pass(self,dq):
    dp = self.w * dq
    self.dw = np.dot(dq,self.p)
    return dp
```

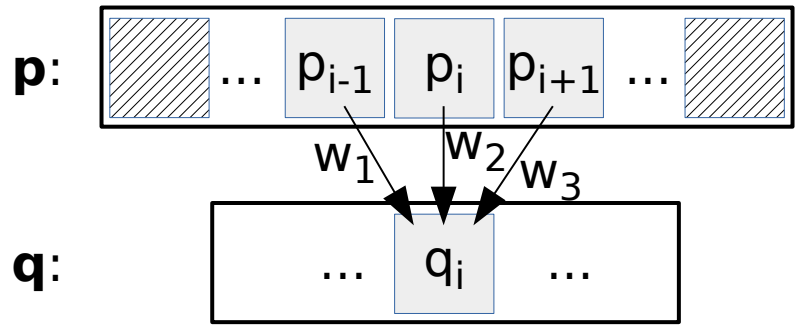
As before, the layer object has to aggregate activations, parameters and gradijents wrt parameters

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:

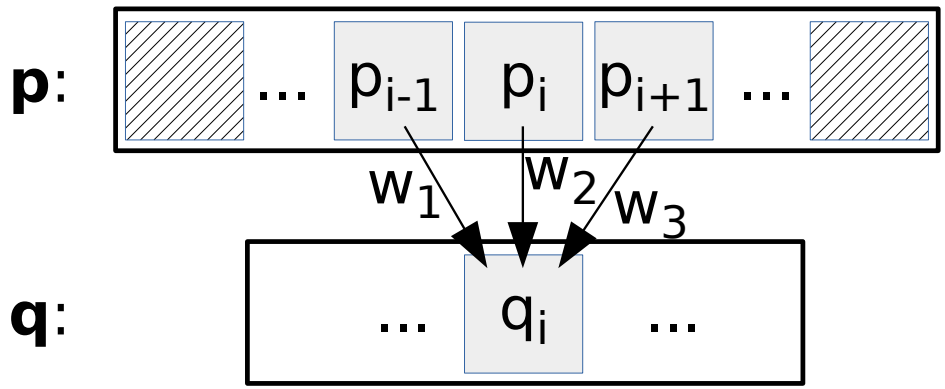
$$q_i = \mathrm{crop}_k(\mathbf{p}, i)^\top \cdot \mathbf{w}$$

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:

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

Vectors: kernel size k, backward weights (1)



$$q_i = \sum_{u=1}^{k} p_{i-o_k+u} \cdot w_u$$

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!):

$$\partial \mathcal{L} / \partial w_u = \sum_i \partial \mathcal{L} / \partial q_i \cdot \partial q_i / \partial w_u = \sum_i \partial \mathcal{L} / \partial q_i \cdot p_{i-o_k+u}$$

$$= \partial \mathcal{L} / \partial \mathbf{q} \cdot \mathrm{shift}(\mathbf{p}, -o_k + u)$$

Vectors: kernel size k, backward weights (2)

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

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

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

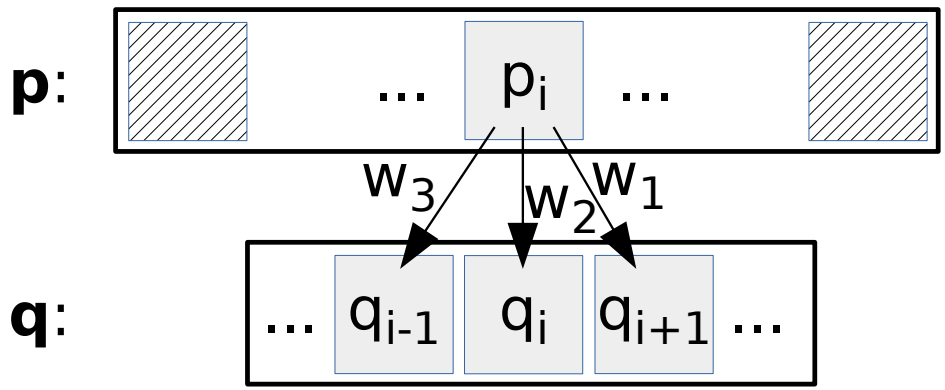
$$\partial\mathcal{L}/\partial w_3 = \sum_i \partial\mathcal{L}/\partial q_i \cdot p_{i+1} = \partial\mathcal{L}/\partial\mathbf{q} \cdot \mathrm{shift}(\mathbf{p}, 1)$$

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

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

Vectors: kernel size k, backward inputs (1)

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



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

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

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

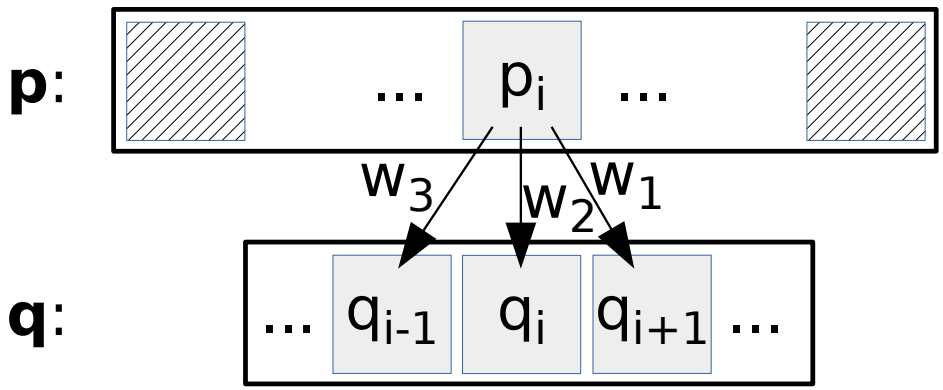
Let us focus on p_i as in the figure:

$$\partial \mathcal{L}/\partial p_i = \sum_r \partial \mathcal{L}/\partial q_r \cdot \partial q_r/\partial p_i$$

$$= \sum_{u=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \partial \mathcal{L}/\partial q_{i+u} \cdot \partial q_{i+u}/\partial p_i$$

$$= \sum_{u=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \partial \mathcal{L}/\partial q_{i+u} \cdot w_{o_k - u}$$

Vectors: kernel size k, backward p (2)



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

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

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

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

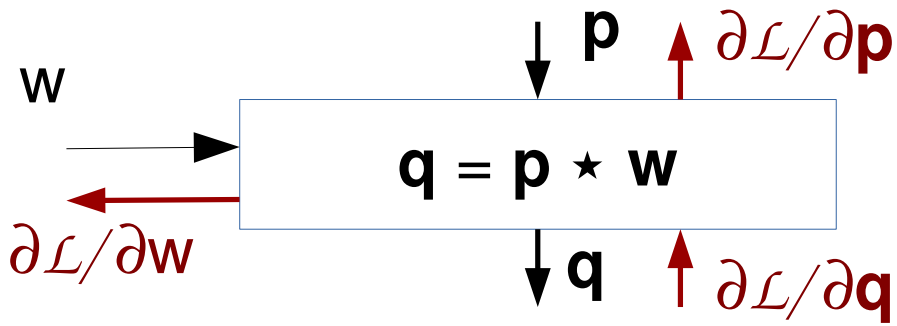
We observe that gradient wrt i-th input corresponds to the **scalar product** of a crop of gradient wrt output and flipped kernel:

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

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

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

Vectors: kernel size k, summary



Forward pass:

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

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

Backward pass over parameters:

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

Backward pass over inputs:

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

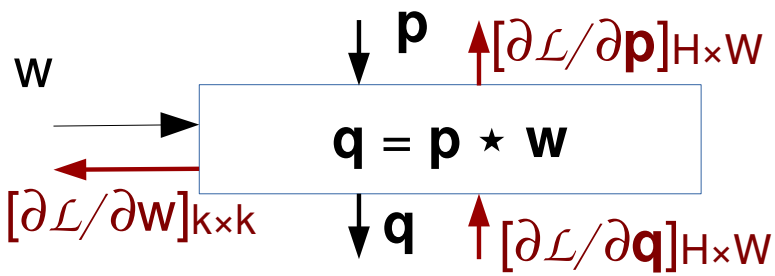
Vectors: code

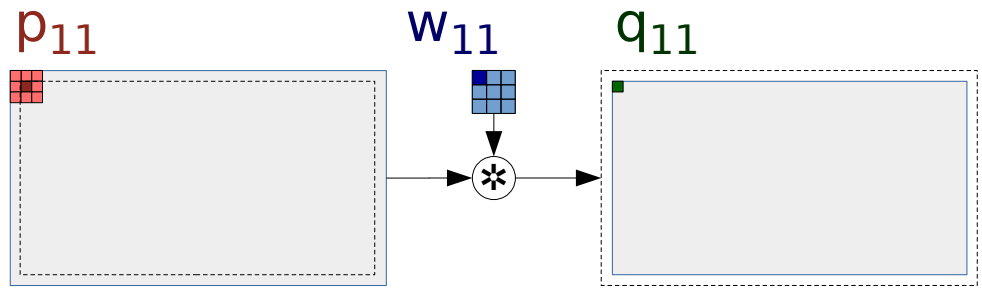
```
def my_conv1d(data, kernel):
  data_pad = torch.nn.functional.pad(data,(1,1),'constant',0)
  return torch.nn.functional.conv1d(
    data_pad.reshape([1,1,-1]),
    kernel.reshape([1,1,-1])).squeeze()

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)
dLdp = my_conv1d(dLdq, torch.flip(w, (0,)))
dLdw = my_conv1d(p, dLdq)

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

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

Matrices: kernel kxk, forward





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

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

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

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

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 kxk, backward

Dimensions of tensors that participate in the backward pass:

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

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

General formulation of gradients wrt one input pixel:

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

Key questions for recovering gradients wrt input:

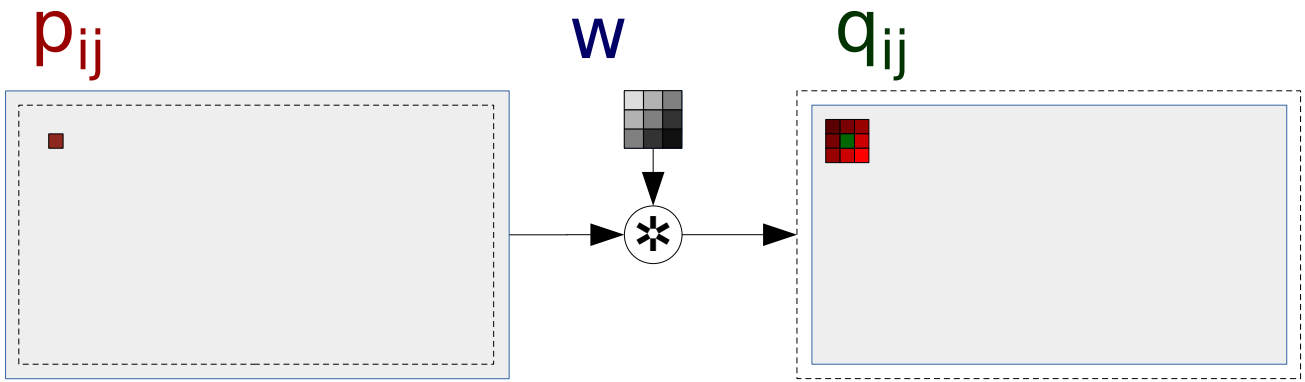
- which q_r depend on p_{ij}?
- in which r we have $\partial q_r/\partial p_{ij} \neq 0$?

Matrices: kernel kxk, backward, inputs

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

Then, the output $\mathbf{q}$ will have the following non-zero pixels:

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



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

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

Matrices: kernel kxk, backward, inputs (2)

What happens when we consider all activations of $\mathbf{p}$?

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

We are ready to formulate the gradients wrt input:

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

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):

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

Matrices: kernel kxk, backward, inputs (3)

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

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

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

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:

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

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 will denote $\mathrm{pad}(\mathbf{x}, \lfloor k/2 \rfloor) \star \mathrm{flip2d}(\mathbf{w})$ 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: $\mathbf{x} \star^{\top} \mathbf{w} \triangleq \mathrm{pad}(\mathbf{x}, \lfloor k/2 \rfloor) \star \mathrm{flip2d}(\mathbf{w})$

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: kernel kxk, 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 $\rightarrow$ 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:

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

Matrices: kernel kxk, backward, parameters (2)

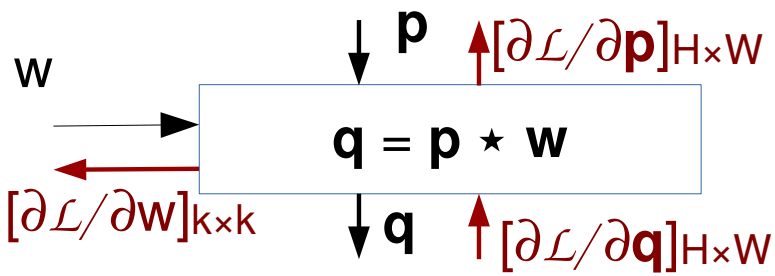
When we consider all kernel elements $\mathbf{w}$:

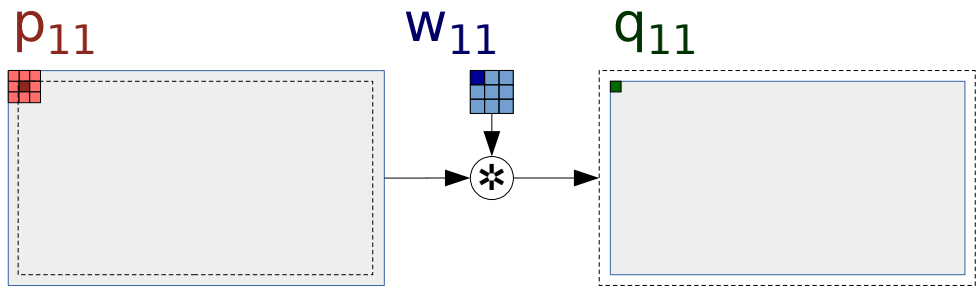
- activations $\mathbf{q}$ correspond to a sum of contributions from all w_{uv}
- the gradijents $\partial q_{ij}/\partial w_{uv}$ remain the same as on the last slide

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

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

Matrices: kernel kxk, backward, summary





Forward pass:

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

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

Backward pass over parameters:

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

Backward pass over inputs:

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

Matrices: problem

We consider a convlutional model for classifying greyscale images:

- convolution 3x3 with no bias and padding ($\mathbf{w}_1$, $\mathbf{w}_2$), ReLU
- global max-pooling
- fully connected layer with no bias ($\mathbf{W}$)
- softmax

Initialization:

- w_{121}=-1, w_{122}=1, all other elements of $\mathbf{w}_1$ are 0
- w_{232}=-1, w_{222}=1, all other elements of $\mathbf{w}_2$ are 0
- W_{11}= W_{22}=1, all other elements of $\mathbf{W}$ are 0

The input is a 4×4 image $\mathbf{x}$, x_{22}=1, x_{33}=1, all other elements of 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

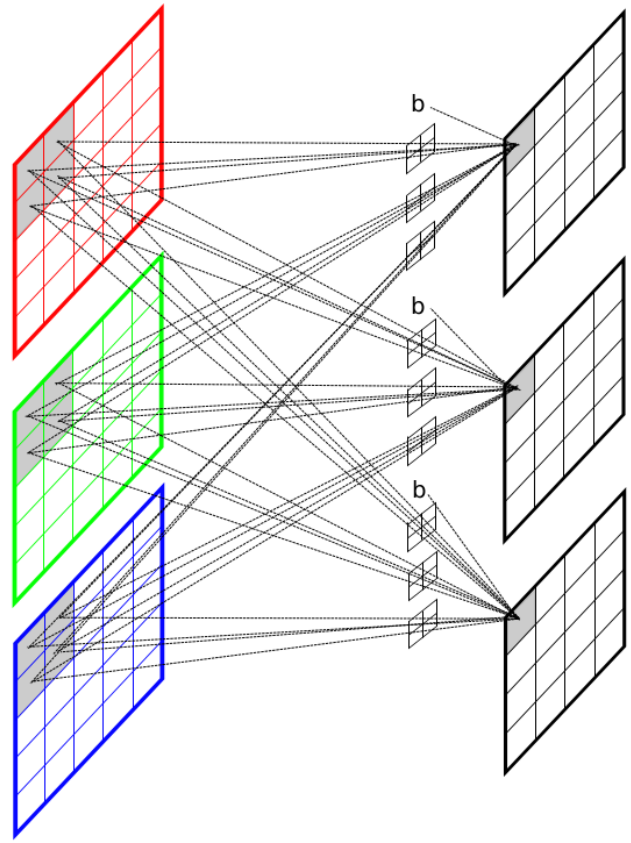
- recap: affine layers, backward propagation
- gradients of 1D convolution parameters
  - kernel size 1
  - kernel size k
- gradients of 2D convolution parameters
  - kernel size k$\times$k
  - problem: backward pass through a convolutional problem
- **2D convolution over 3rd order tensors**
- gradients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

Multi-channel 2d conv: forward

We overload the $\star$ operator so we can keep the same syntax as before: $\mathbf{q} = \mathbf{p} \star \mathbf{w}$

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

$$\mathbf{q}^{(r)} = \sum_s \mathbf{p}^{(s)} \star \mathbf{w}^{(r,s)}$$



The same formulation in scalar algebra:

$$q_{ij}^{(r)} = \sum_{suv} p_{i-o_k+u,j-o_k+v}^{(s)} \cdot w_{uv}^{(rs)}$$

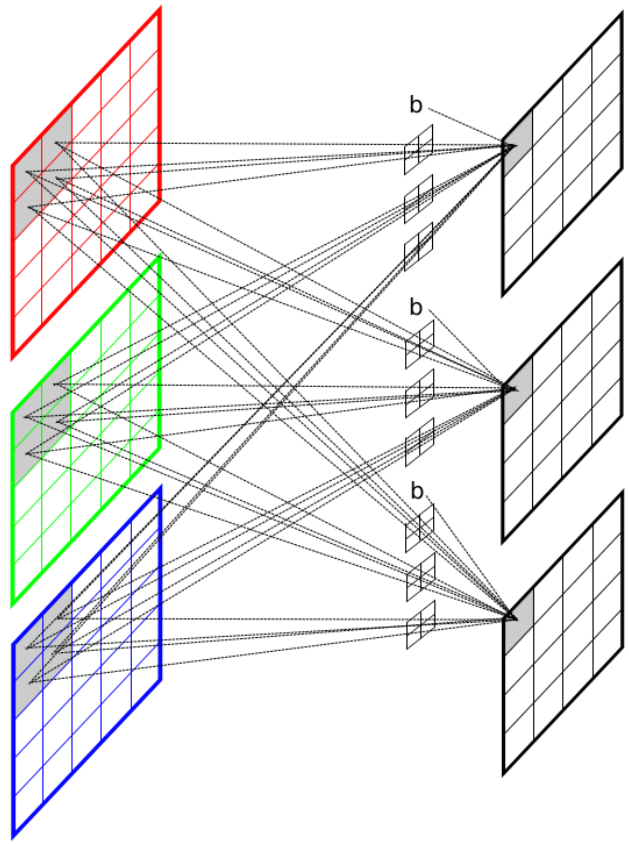
Convolutional layers often include the bias $\mathbf{b}$: $\mathbf{q} = \mathbf{p} \star \mathbf{w} + \mathbf{b}$

- we have one component for each output feature map: $\mathbf{b} = [b_r]$
- gradients wrt bias correspond to aggregated gradients wrt output:

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

Multi-channel 2d conv: backward, inputs

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



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

$$\partial\mathcal{L}/\partial\mathbf{p}^{(s)}_{ij} = \sum_r \sum_{u,v} \partial\mathcal{L}/\partial\mathbf{q}^{(r)}_{i+u,j+v} \cdot w^{(r,s)}_{o_k-u,o_k-v}$$

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

$$\partial\mathcal{L}/\partial\mathbf{p}^{(s)} = \sum_r \mathrm{pad}(\partial\mathcal{L}/\partial\mathbf{q}^{(r)}, \lfloor k/2 \rfloor) \star \mathrm{flip2d}(\mathbf{w}^{(rs)})$$

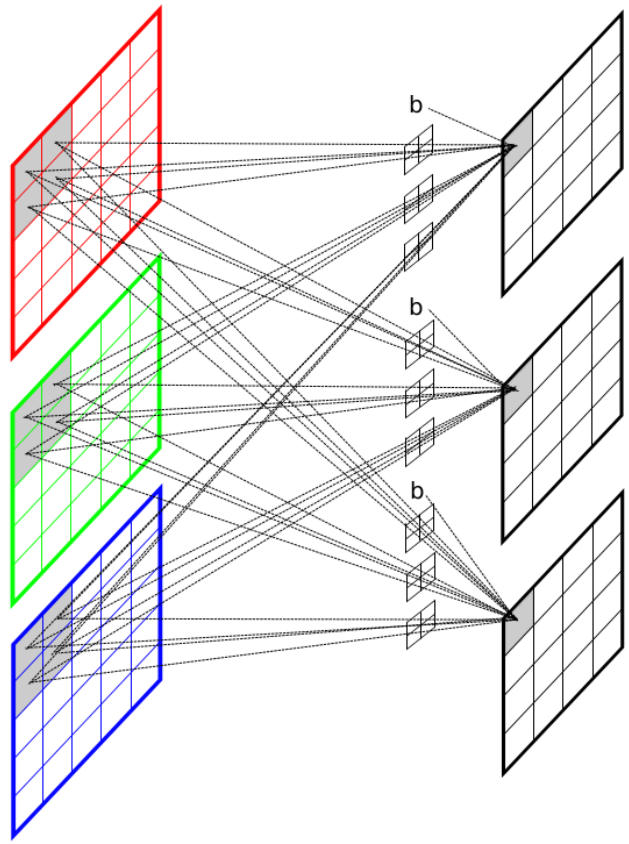
This corresponds to multi-channel transposed convolution:

$$\mathbf{x} \star^{\top} \mathbf{w} \triangleq \sum_r \mathrm{pad}(\mathbf{x}^{(r)}, \lfloor k/2 \rfloor) \star \mathrm{flip2d}(\mathbf{w}^{(rs)})$$

Multi-channel 2d conv: backward, parameters

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

- the sth slice of the rth kernel influences the rth output slice through convolution with the sth input slice



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

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

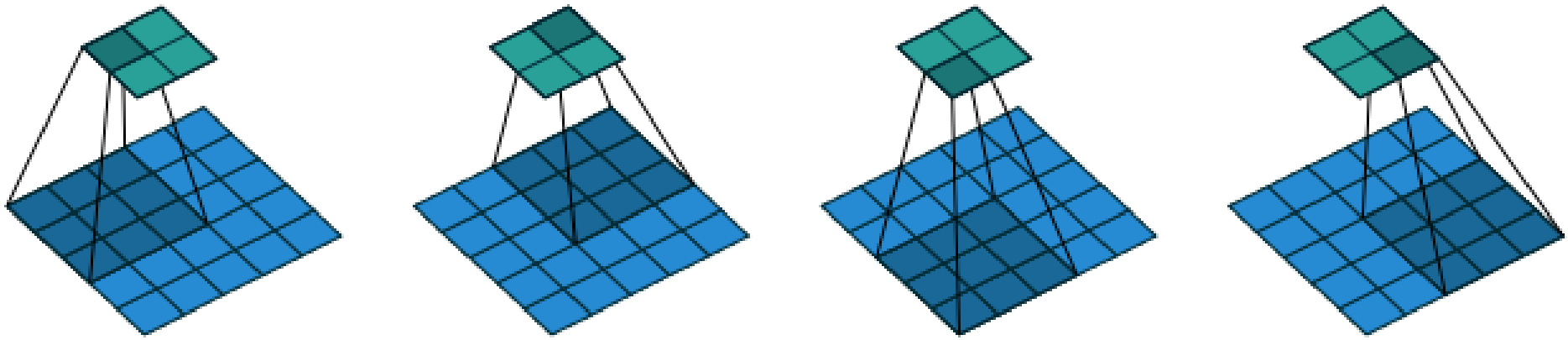
$$\partial \mathcal{L}/\partial \mathbf{w}^{(rs)} = \mathrm{pad}(\mathbf{p}^{(s)}, \lfloor k/2 \rfloor) \star \partial \mathcal{L}/\partial \mathbf{q}^{(r)}$$

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: naprijed

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



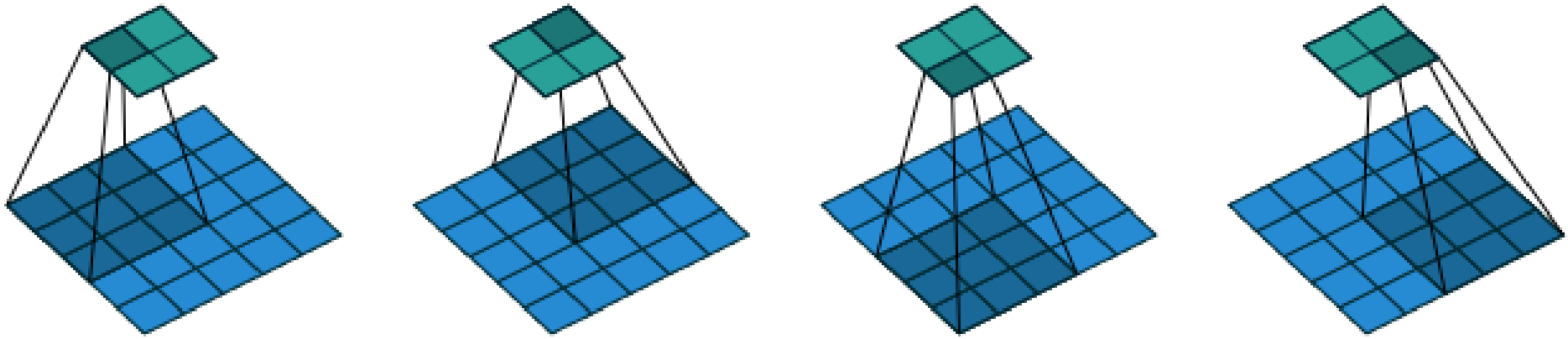
[dumoulin16arxiv]

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
- we see that integrated backward step through strided convolution decreases the effective receptive field by s times
  - due to this effect, Theano denotes strided transposed convolution as fractional convolution

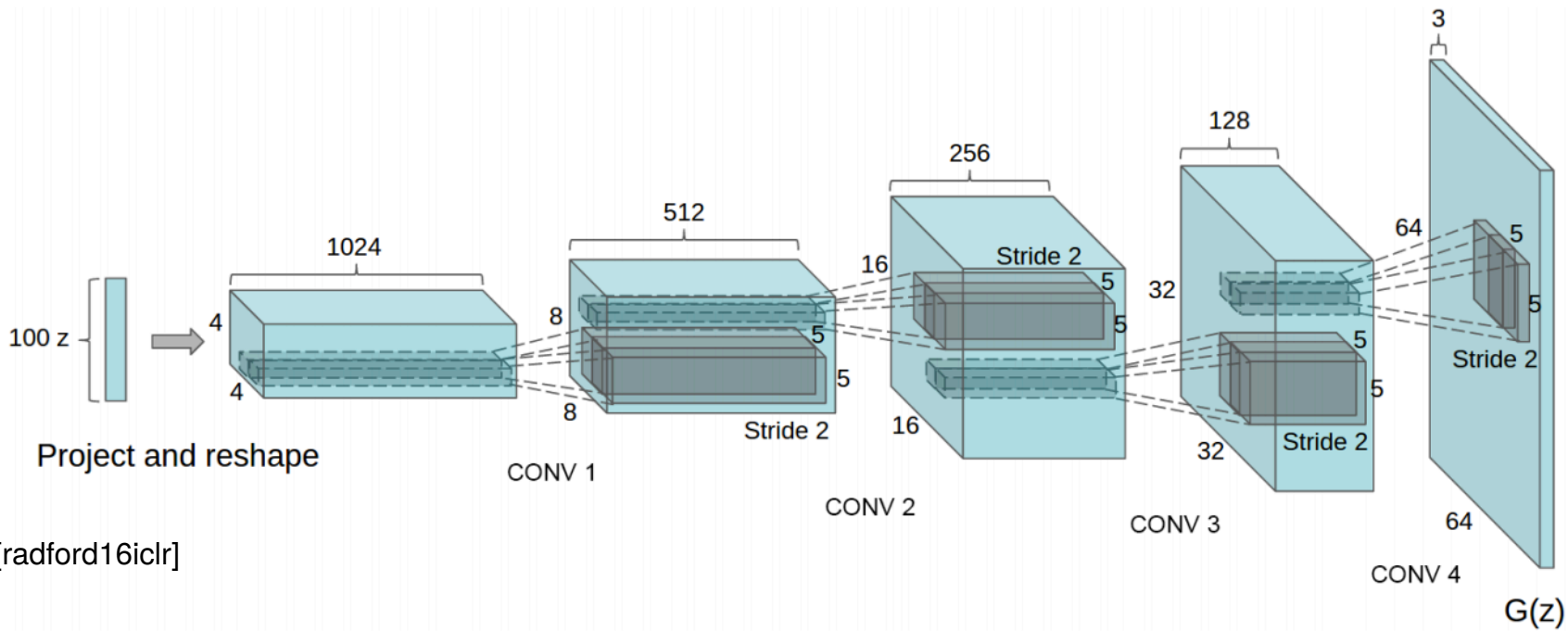


Stride: integrated backward

Idea: accumulate contributions from each output:

$$\partial \mathcal{L}/\partial \mathbf{p}^{(s)} = \sum_{ij} \sum_{r} \partial \mathcal{L}/\partial q_{ij}^{(r)} \cdot \partial q_{ij}^{(r)}/\partial \mathbf{p}^{(s)}$$

$$= \sum_{ij} \sum_{r} \partial \mathcal{L}/\partial q_{ij}^{(r)} \cdot \mathrm{embed}_{HW}(\mathrm{flip2d}(\mathbf{w}^{(rs)}), i^{(*)}, j^{(*)})$$



[radford16iclr]

Such operation can not be expressed with convolution

- $\Rightarrow$ requires special implementation

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 kernal (good idea)

Recent implementations use the Winograd algorithm

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

Implementation: flattening activations

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

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

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

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

Implementation: flattening the kernel

Rearrange the input so convolution corresponds to matrix multiplication:

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

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

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_{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:

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

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

$\boldsymbol{X}$

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}

$\boldsymbol{X}^{\mathrm{ROWS}}$

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:

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

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

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_{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:

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

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

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_{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: backward, parameters

We have expressed convolution as matrix multiplication:

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

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

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

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

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

Implementation: backward, inputs

Recall the expression to recover gradients wrt input:

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

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

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

Now the convolution can be implemented as matrix multiplication:

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

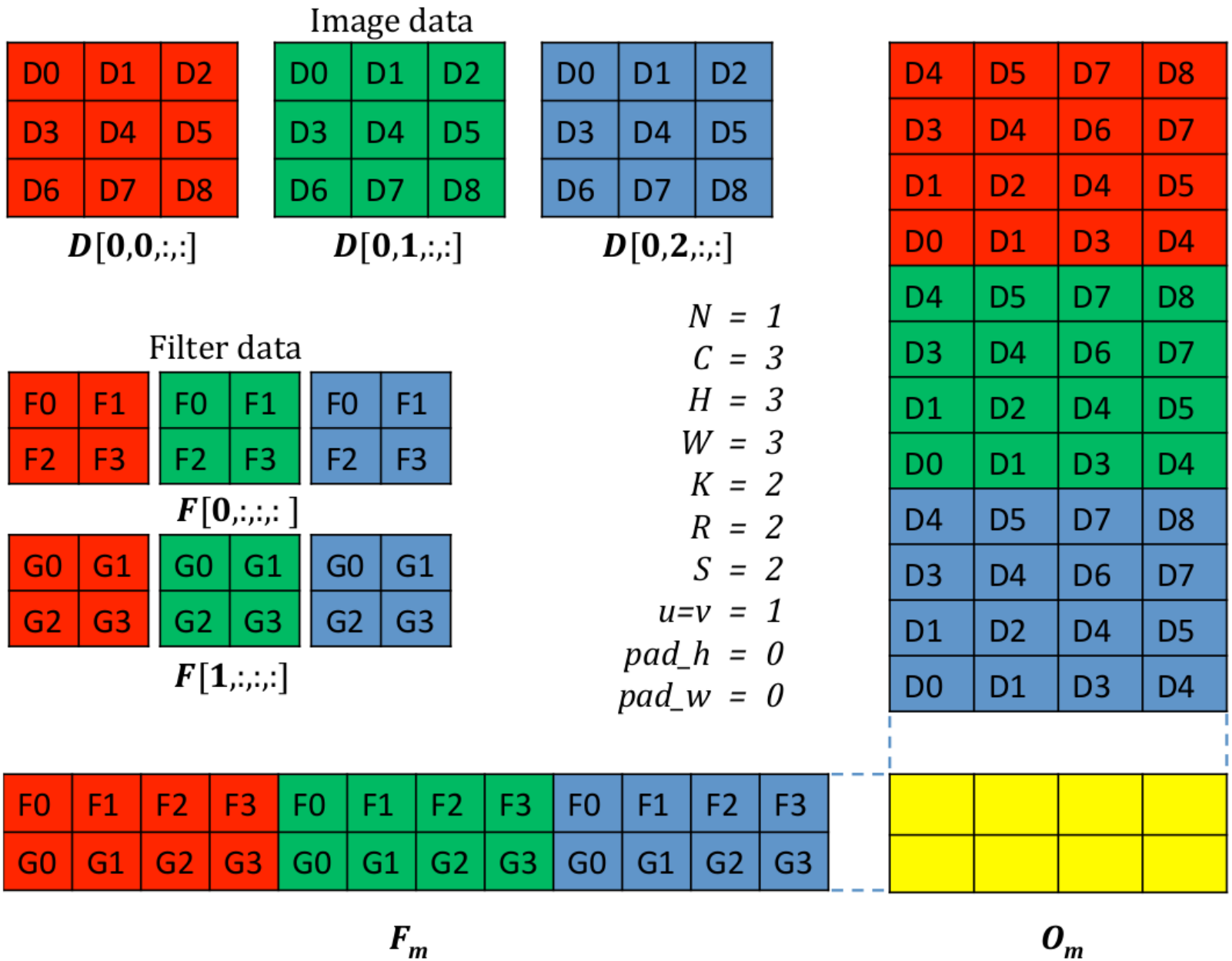
Finally, we recover the result by reshaping $\mathbf{G}_{\mathbf{p}}^{\mathrm{VEC}}$:

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

If we consider a strided convolution, then:

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

Implementation: multi-channel [chetlur14arxiv]



ZAHVALA

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

http://multiclod.zemris.fer.hr