

DDA4220 · Fall 2026

Lecture 6

Deep Learning and Applications

Convolutional networks and object recognition

Zhen Liu

CUHK-Shenzhen

Lecture outline

| 01 Images and convolution

02 Convolution, spatial structure and classification

03 CNN architectures

Image classification



Bird



Chameleon



Bear

$$x \mapsto (z_1, \dots, z_C)$$

One score for each category

Visual variation

Dog



Different views,
lighting and
backgrounds



Hand-designed image features

Input image



Histogram of Oriented Gradients



Feature rule

Chosen by hand

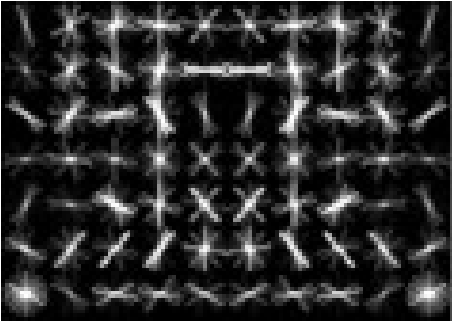
Classifier

Learned from
labeled examples

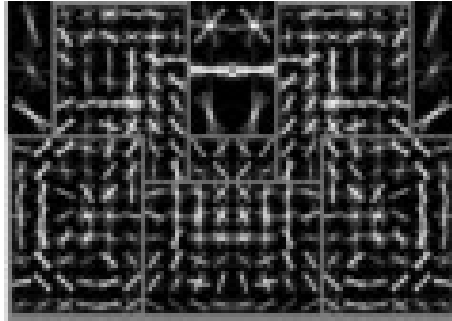
Traditional object detection

Deformable part model (DPM)

Whole object



Parts



A misplaced part can affect the refined object box.

HOG features

Fixed descriptor



Object + part scores

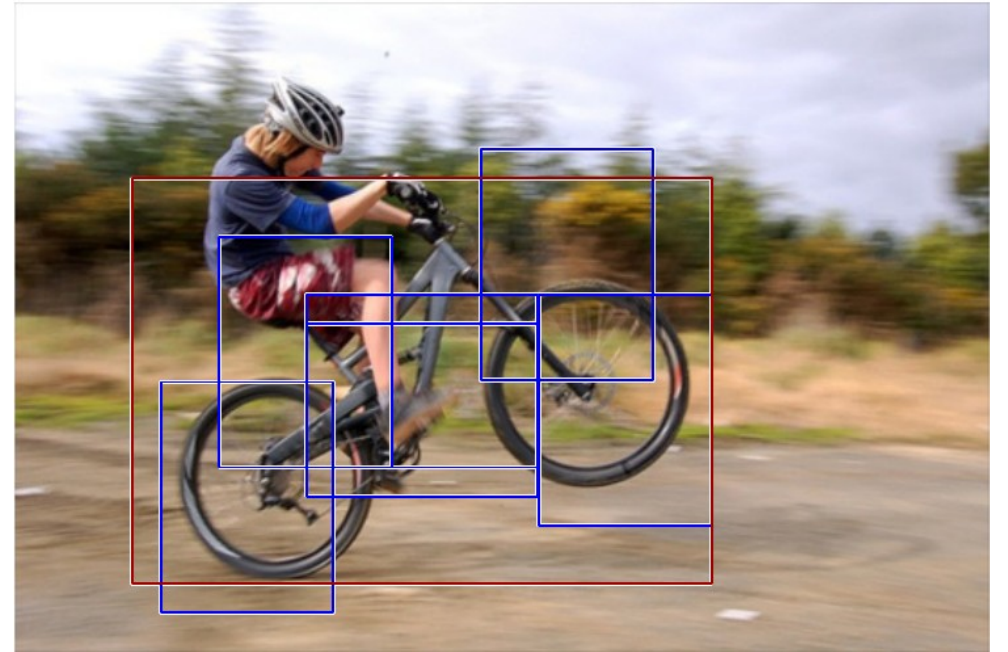
Jointly learned model



Refine / select boxes

Separate processing

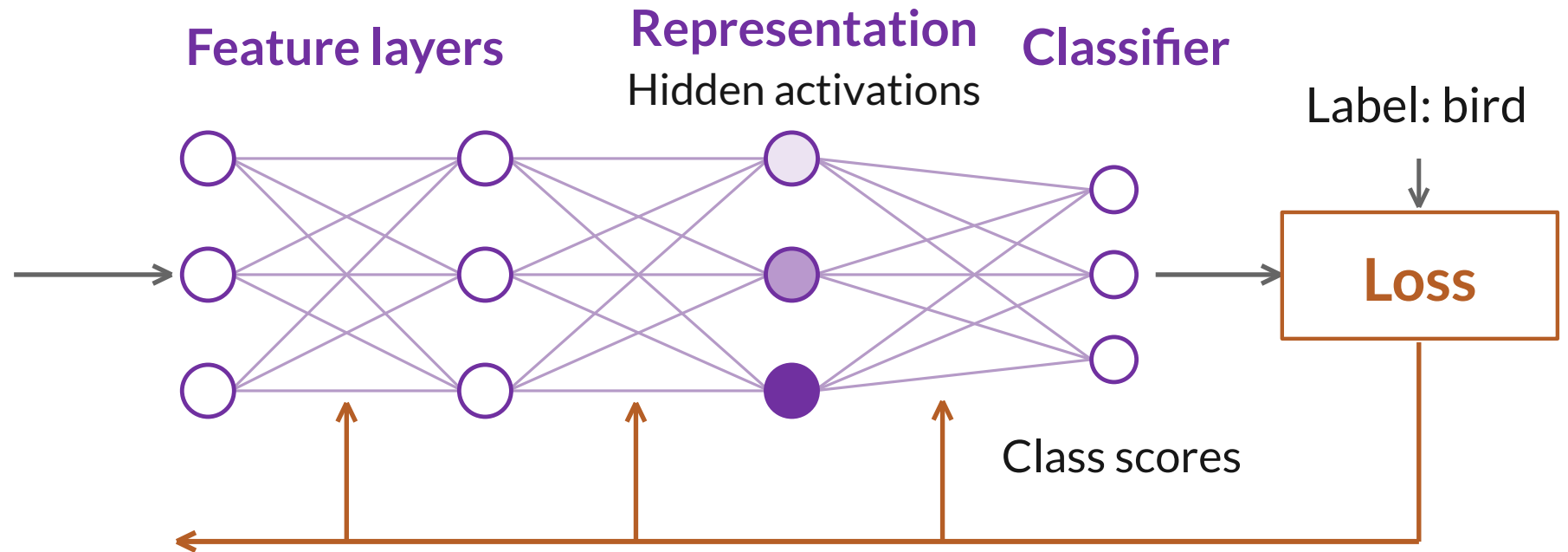
Matched object and parts



Learning representations

A representation describes an input using numerical features.

Input image



Training learns the weights that compute these features.

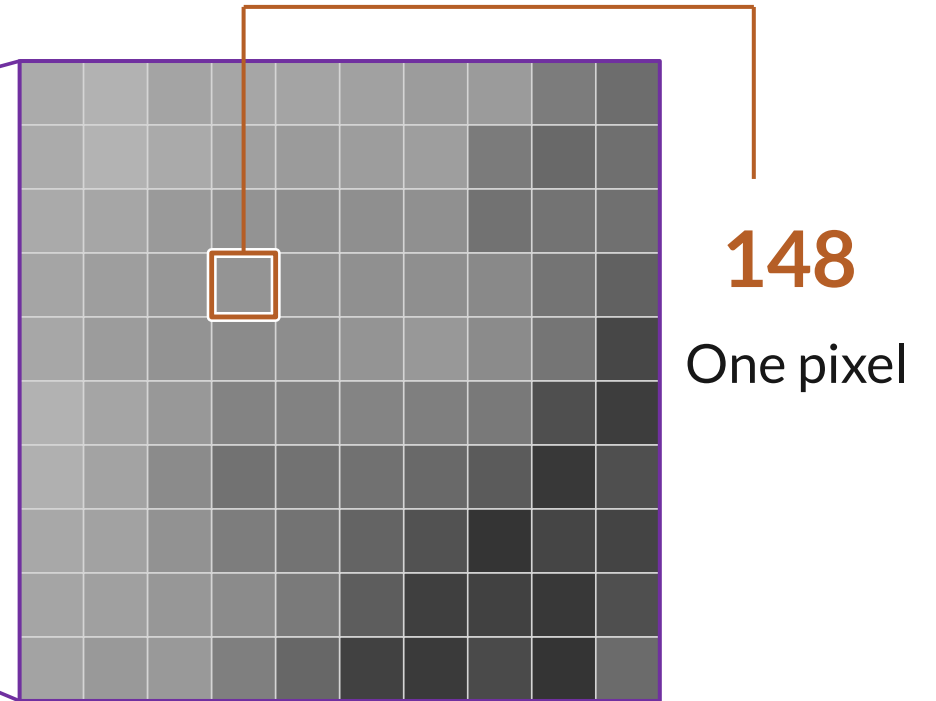
End-to-end learning: one loss trains the feature layers and classifier.

Pixels

Photograph



Same region in grayscale



Grayscale: one intensity per pixel

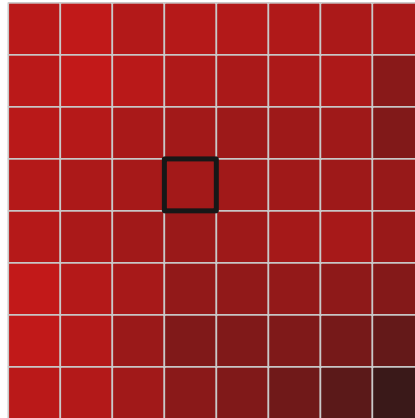
10 × 10 pixels

Color channels

RGB image

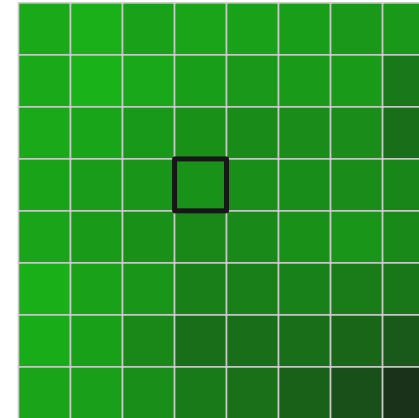


Red
Same local region



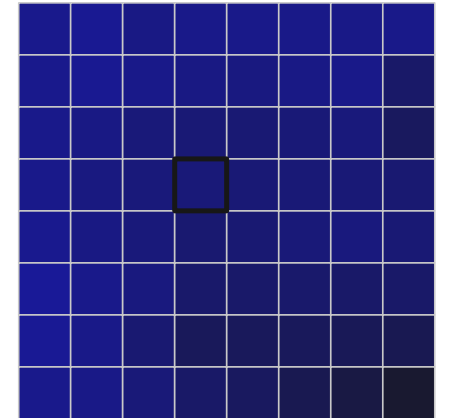
163

Green



146

Blue



120

Three components at one position

A batch of images



Image 1



Image 2



Image 3

Tensor axes

N images

C channels

H height

W width

$$N \times C \times H \times W$$

Tensor reshaping

One image: two channels

1	2	3
4	5	6

7	8	9
10	11	12



One vector

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

Same values, in the same order

Keep each image separate in the batch

$$N \times C \times H \times W \longrightarrow N \times (CHW)$$

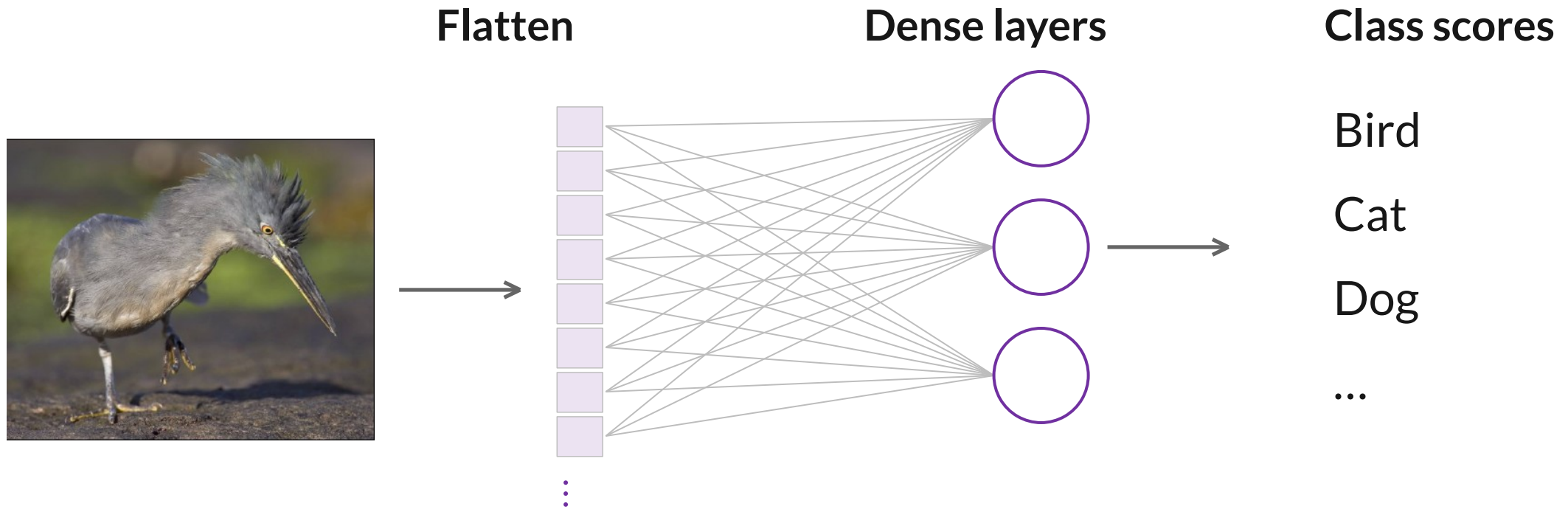
Flattening in PyTorch

```
flat = x.flatten(1)           # keep the batch axis
hidden = nn.Linear(3*32*32, 128) # 128 hidden units
h = torch.relu(hidden(flat))   # hidden activations
head = nn.Linear(128, 10)      # ten output classes
logits = head(h)               # class scores
```

Prepared input batch

$$x : N \times 3 \times 32 \times 32$$

An MLP for images

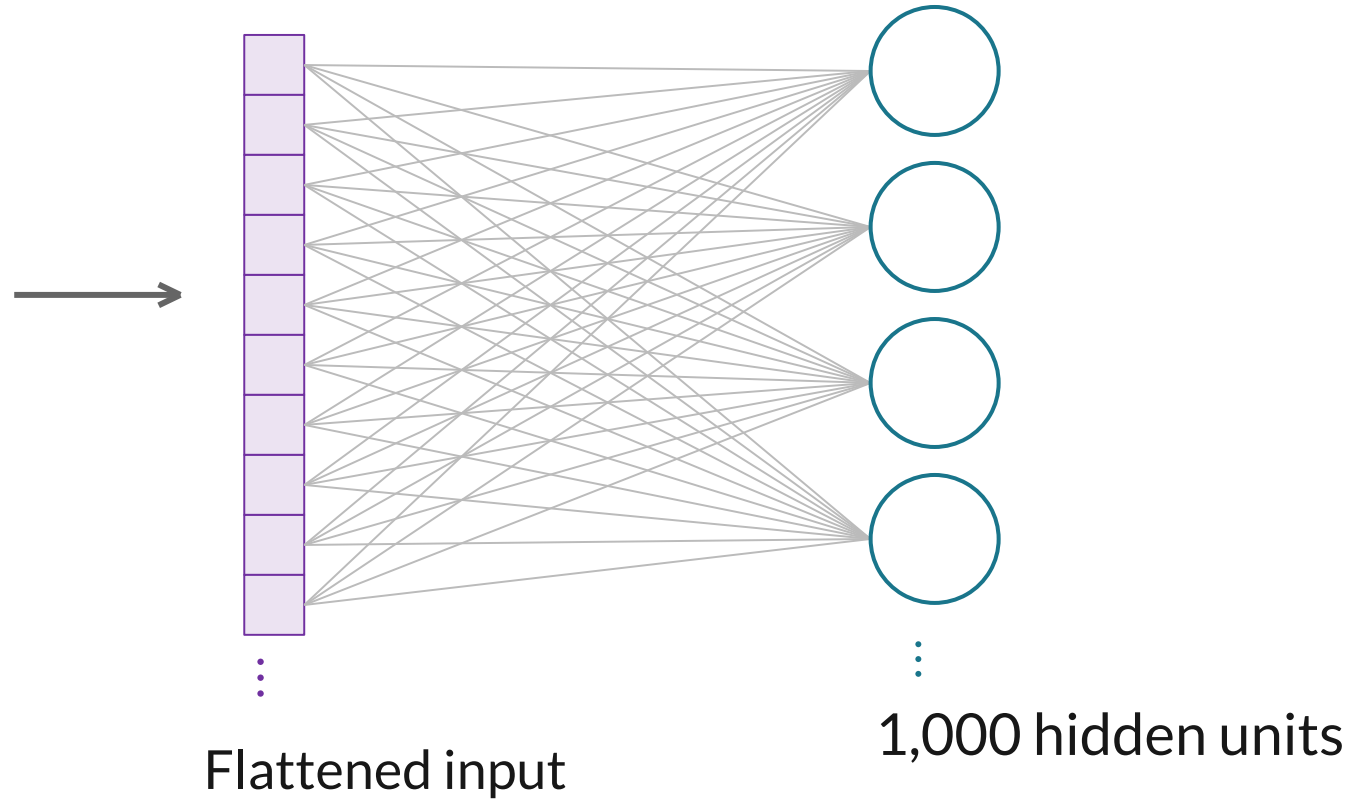


Every hidden unit can use every input value

Dense connections at image scale

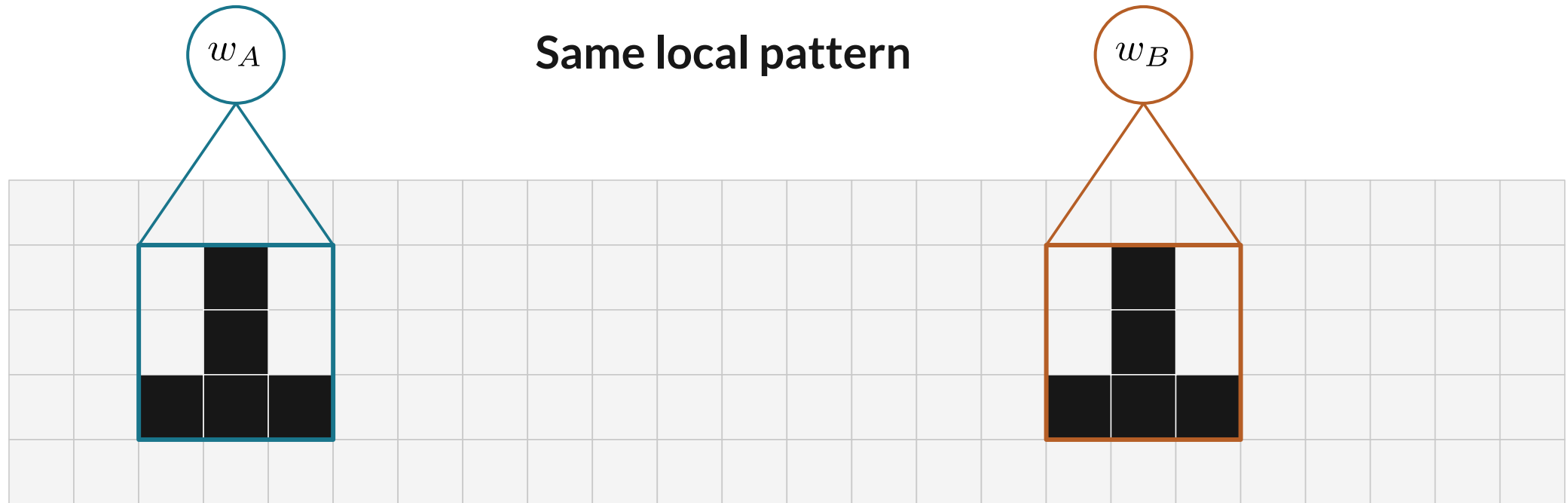


224 × 224 RGB image



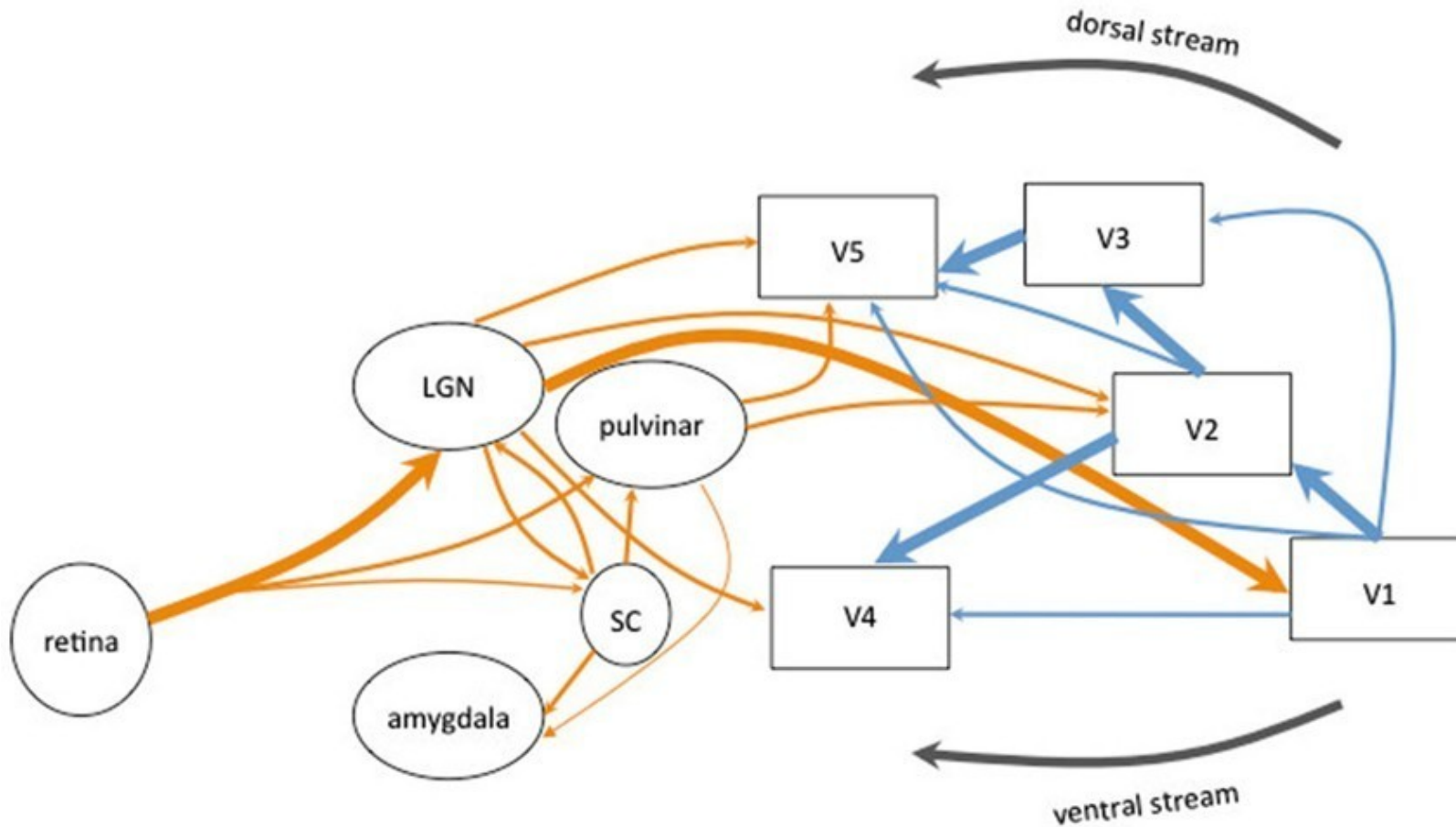
$$(224 \cdot 224 \cdot 3) \cdot 1000 \approx 150 \text{ million weights}$$

The same pattern at different positions



Independent dense weights at different positions

Visual processing

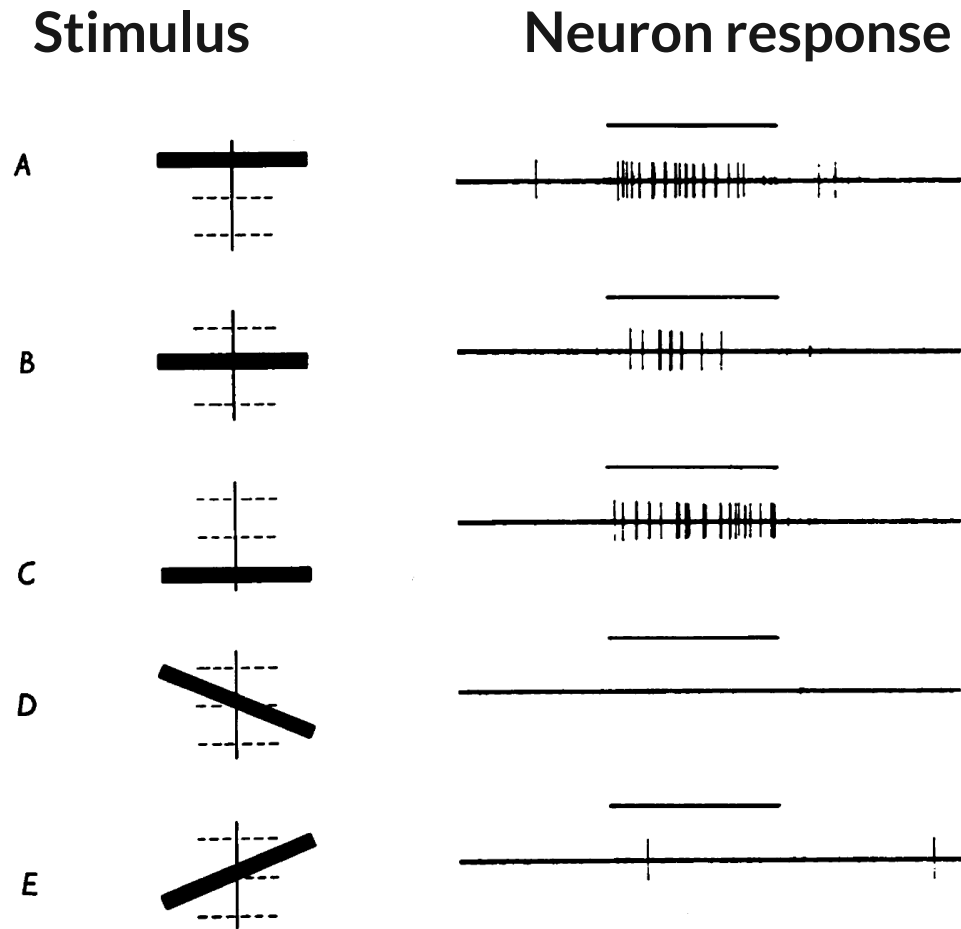


V1

An early cortical stage of vision

Visual processing continues through connected areas.

Visual receptive fields



Cat visual cortex

Receptive field

Region that influences a neuron's response

Horizontal bars

Strong responses

Tilted bars

Weak responses

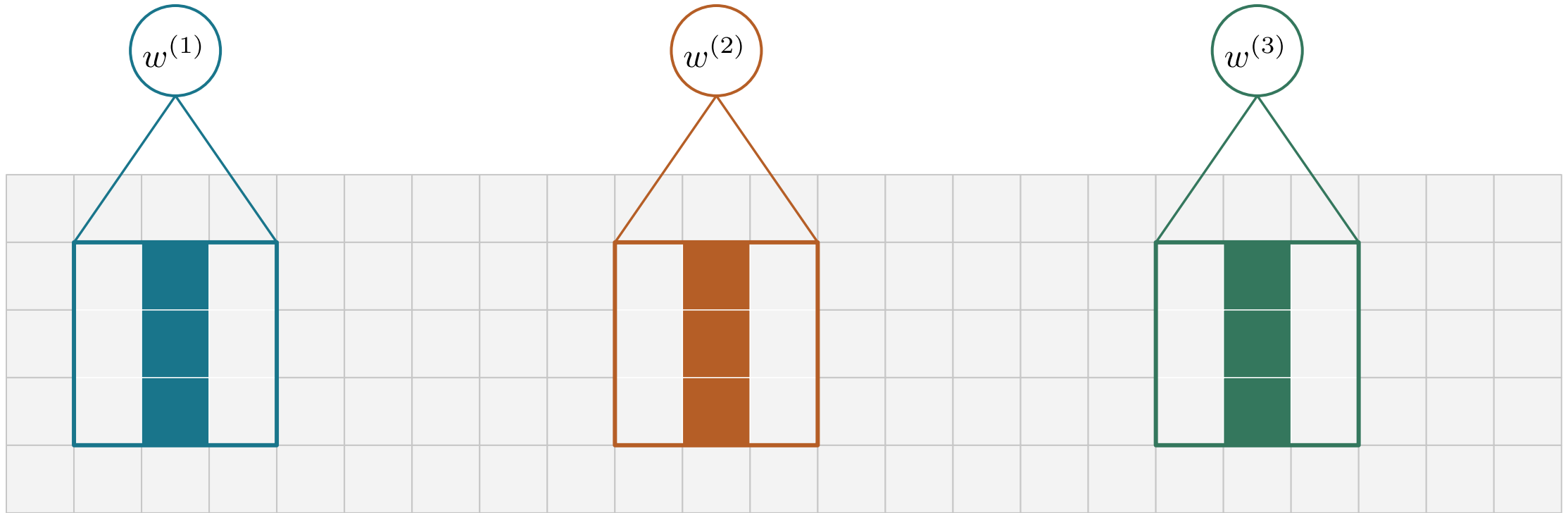
Lecture outline

01 Images and convolution

| 02 **Convolution, spatial structure and classification**

03 CNN architectures

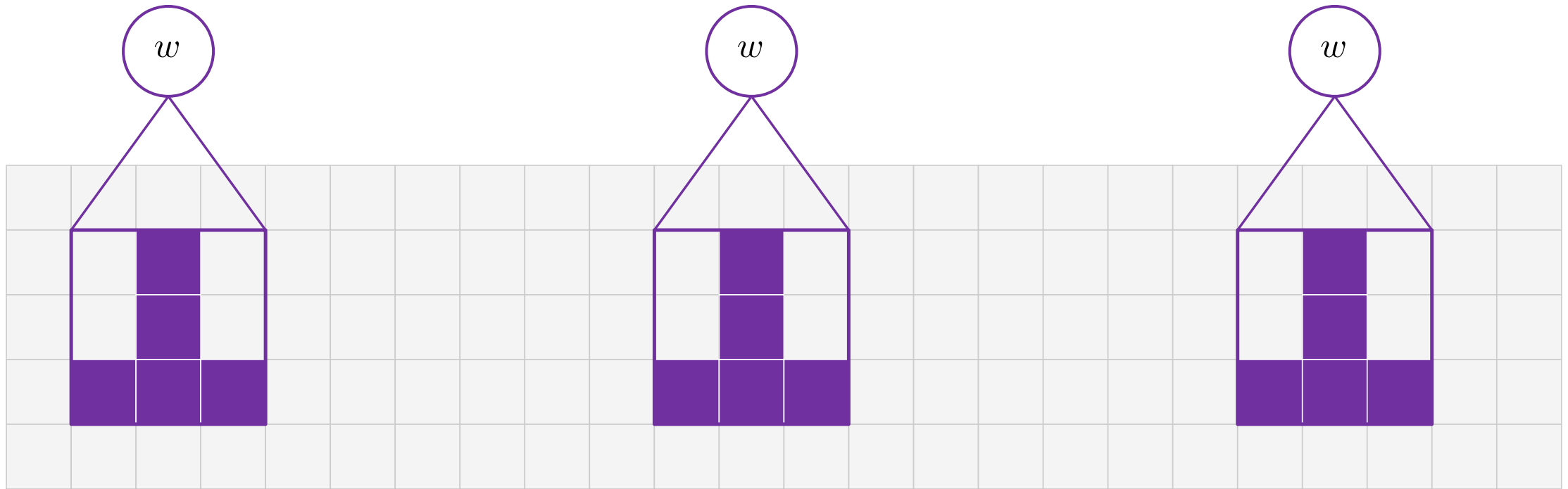
Locally connected layer



Independent weights at different locations

Weight sharing in CNNs

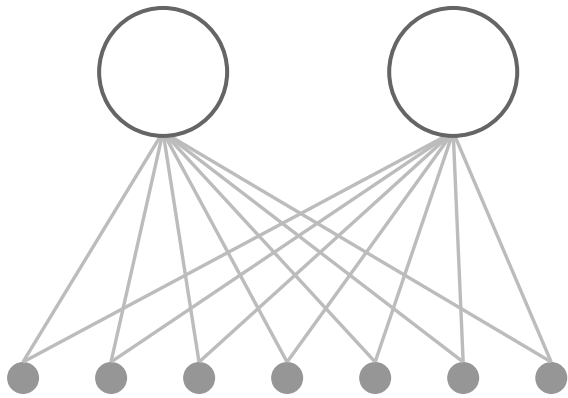
Shifting a local pattern should not require a different detector.



One set of weights, reused across positions

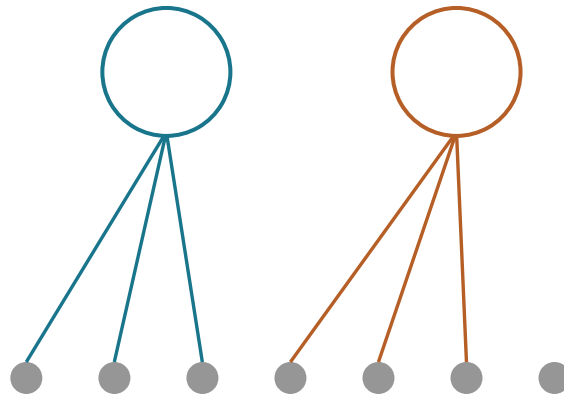
Dense, local and shared connections

Dense



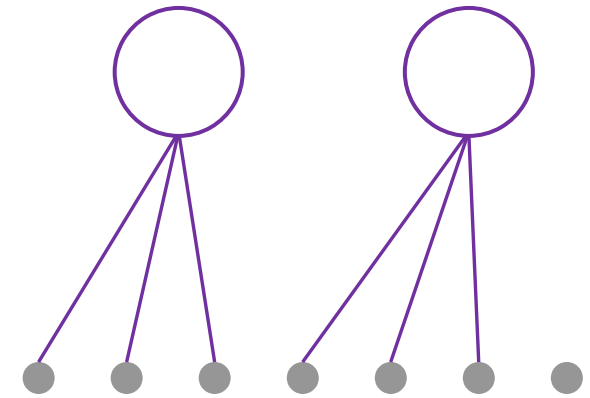
All input positions

Local



Nearby inputs

Local + shared



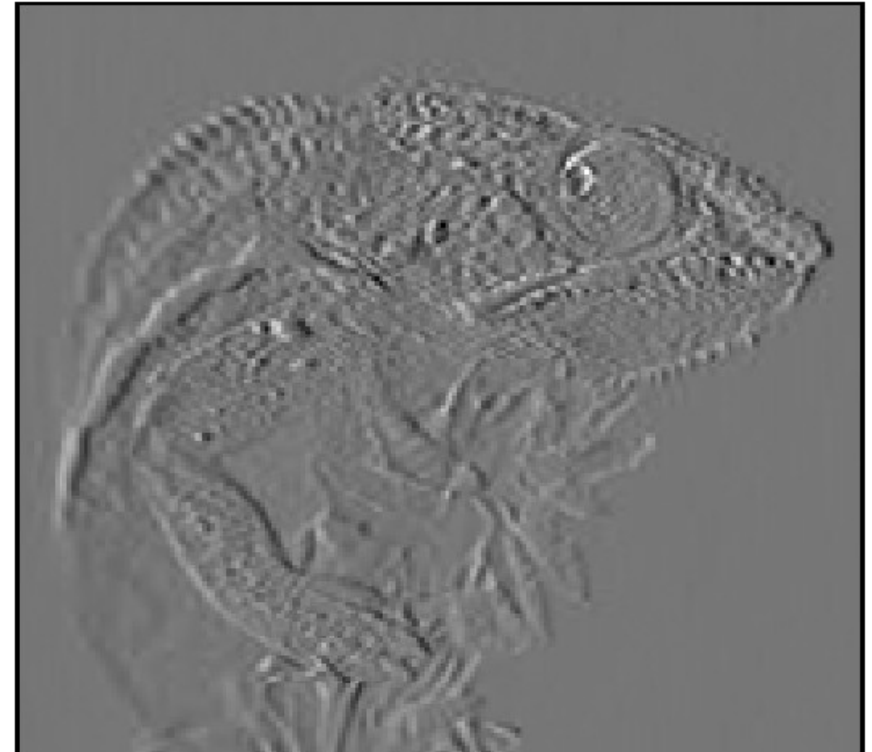
Same local detector

Filters and image responses

Input photograph

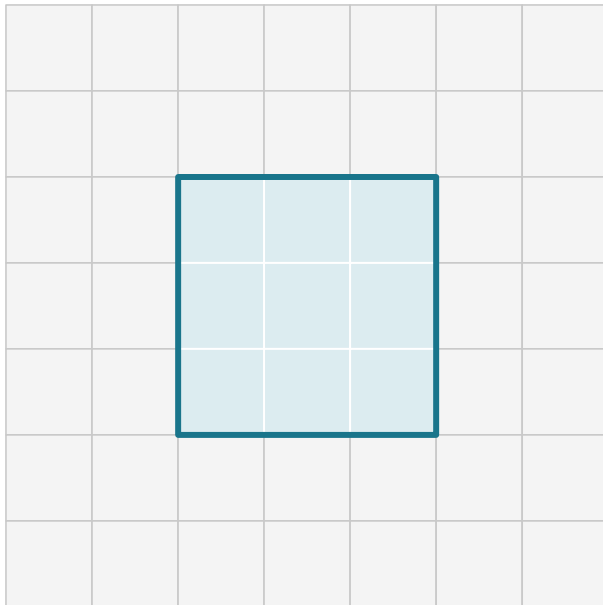


Filter response



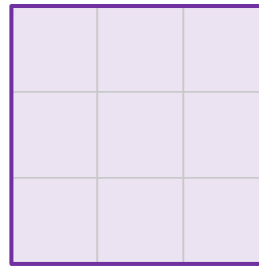
One output location

Input image



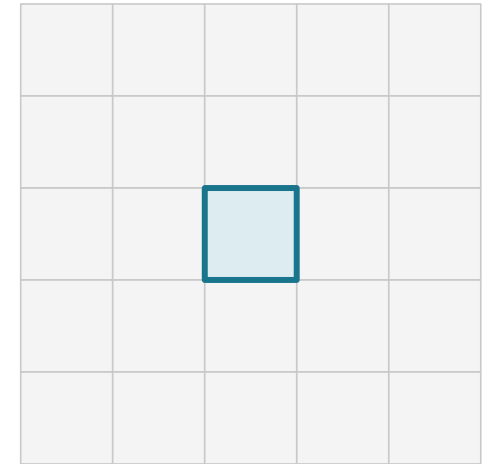
Only this patch contributes

Shared kernel



Same kernel weights

Output map



One response

Local weighted sum

Input patch

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}$$

Kernel coefficients

$$\begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix}$$

$$\odot$$

$$z = k_1x_1 + k_2x_2 + \cdots + k_9x_9 + b$$

Multiply matching entries, sum, then add the bias

Convolution: the weighted sum

One 2D image and one 2D kernel

$$Z_{i,j} = \sum_{u,v} K_{u,v} X_{i+u,j+v} + b$$

$Z_{i,j}$

$K_{u,v} X_{i+u,j+v}$

Output at (i, j)

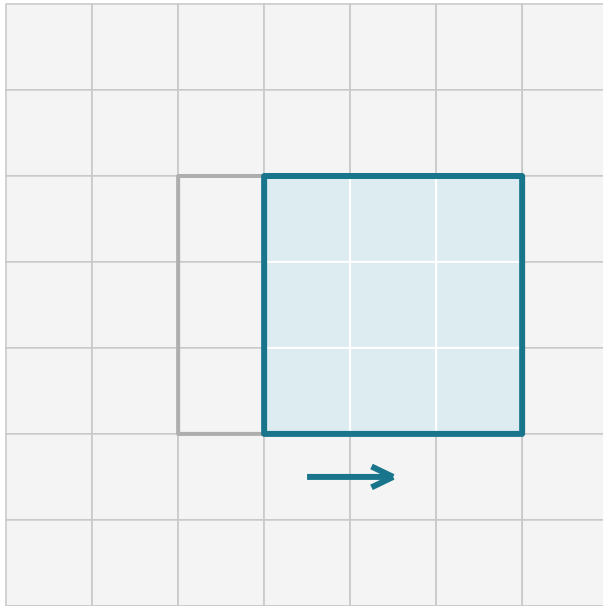
Kernel weight × input value

Sum over all positions (u, v) in the kernel.

X: input image K: kernel b: bias

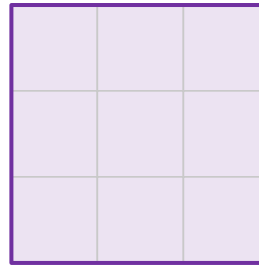
Moving the filter

Input image



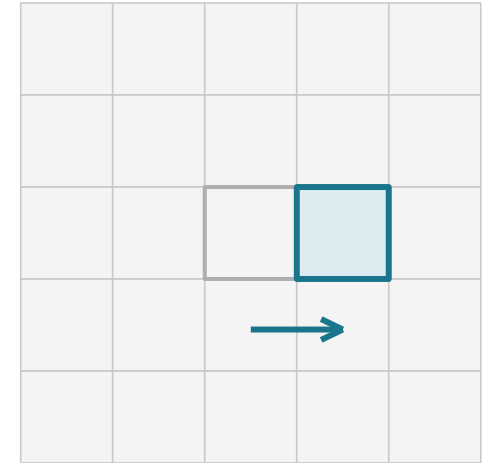
Patch moves one position

Shared kernel



Same kernel weights

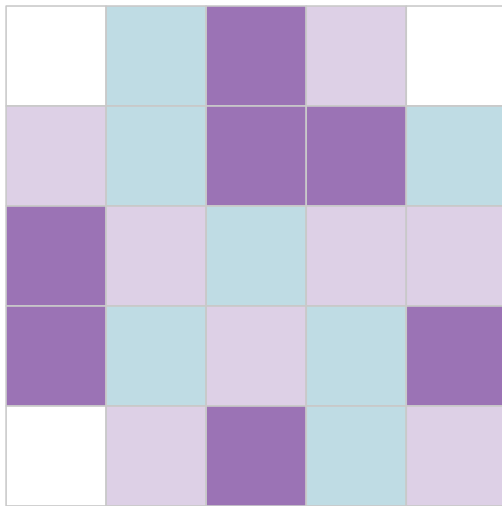
Output map



One response

Feature maps

Weighted responses



Light Blue Negative

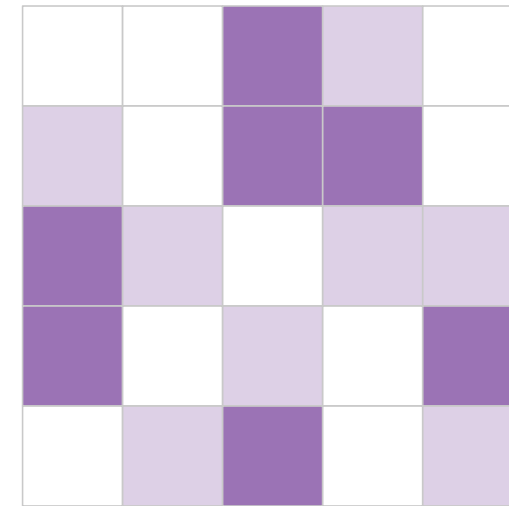
Purple Positive

Elementwise



$$a = \max(0, z)$$

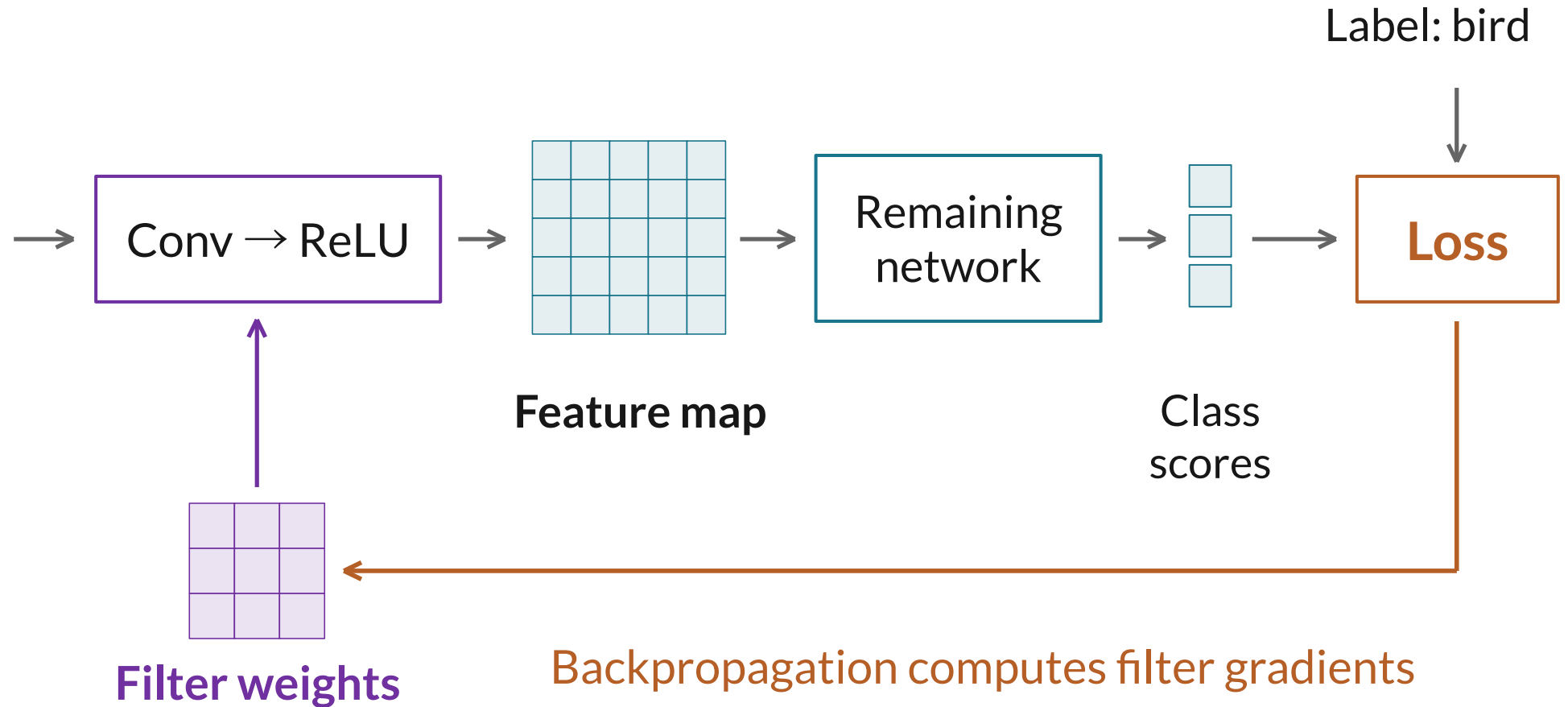
After ReLU



Negative responses become zero

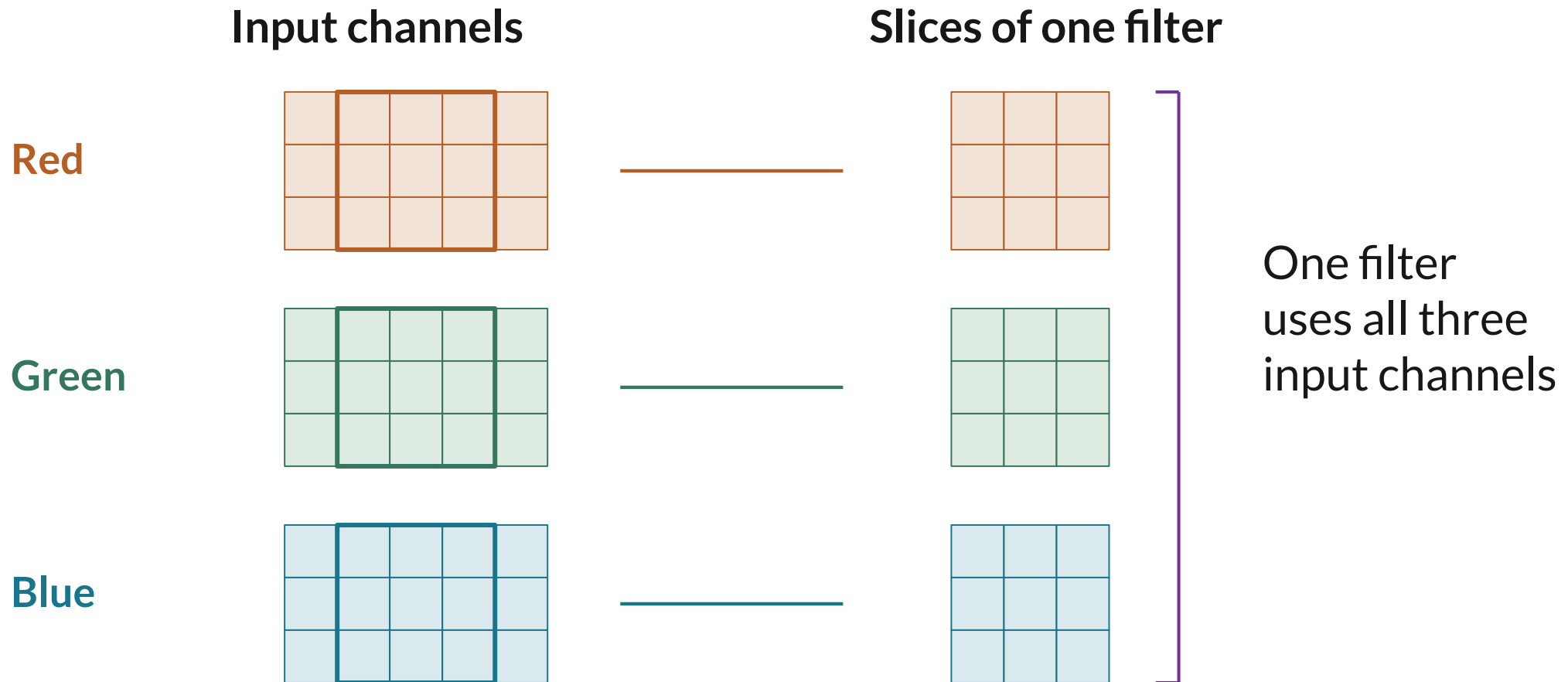
Learning the filters

Image



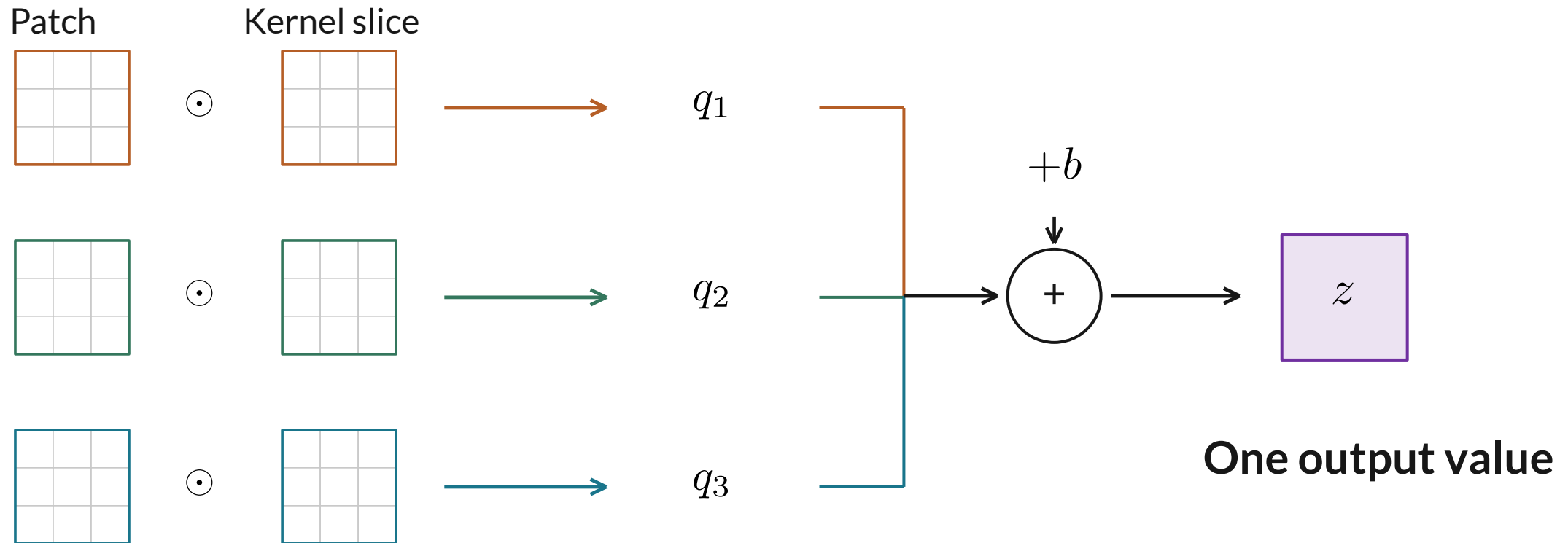
The optimizer uses these gradients to update the filter weights.

Input channels



One response from all channels

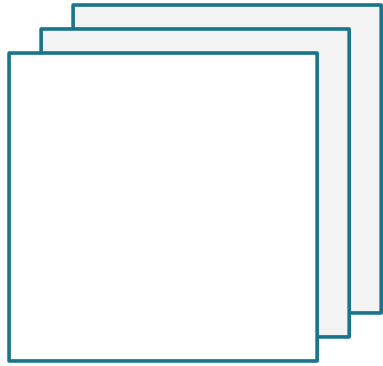
A weighted sum in every input channel



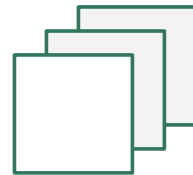
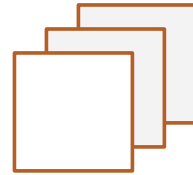
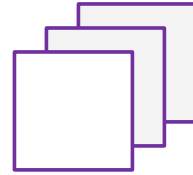
$$z = q_1 + q_2 + q_3 + b$$

Output channels

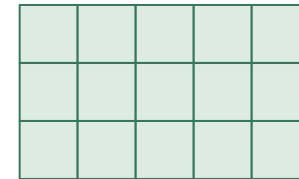
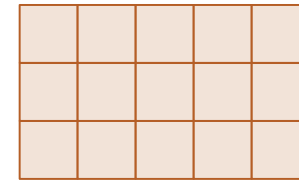
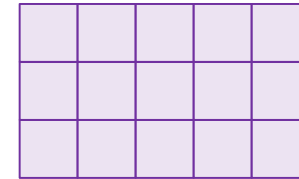
Input channels



Different filters



Output feature maps

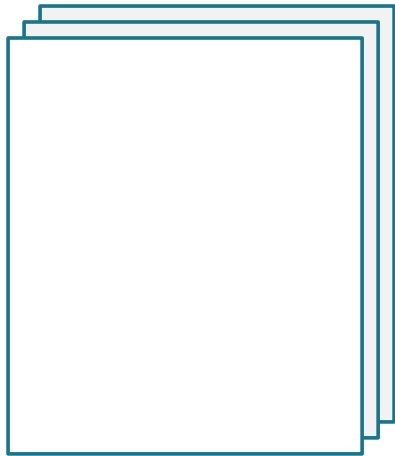


Every filter spans all input channels

One map per filter

Convolutional layer

Input channels



3 channels

Filter bank



4 complete filters

Each filter spans all input channels

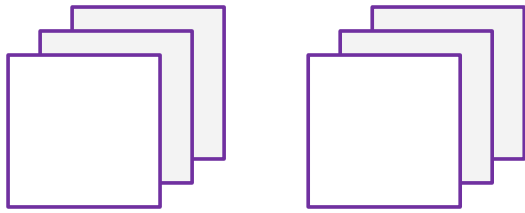
Output channels



4 feature maps

Parameters and activations

Parameters



Learned kernel banks

$$\theta = \{K^{(1)}, b^{(1)}, K^{(2)}, b^{(2)}, \dots\}$$

Updated during training

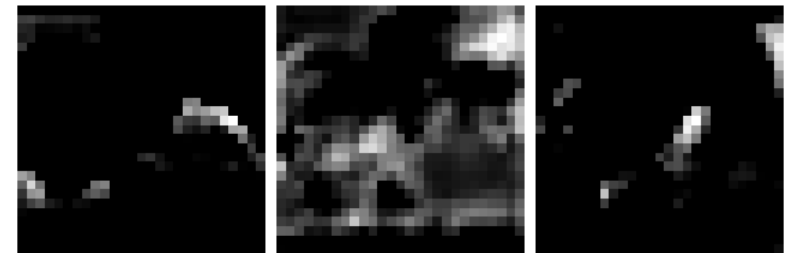
Activations for this image



conv1 (after first conv layer)



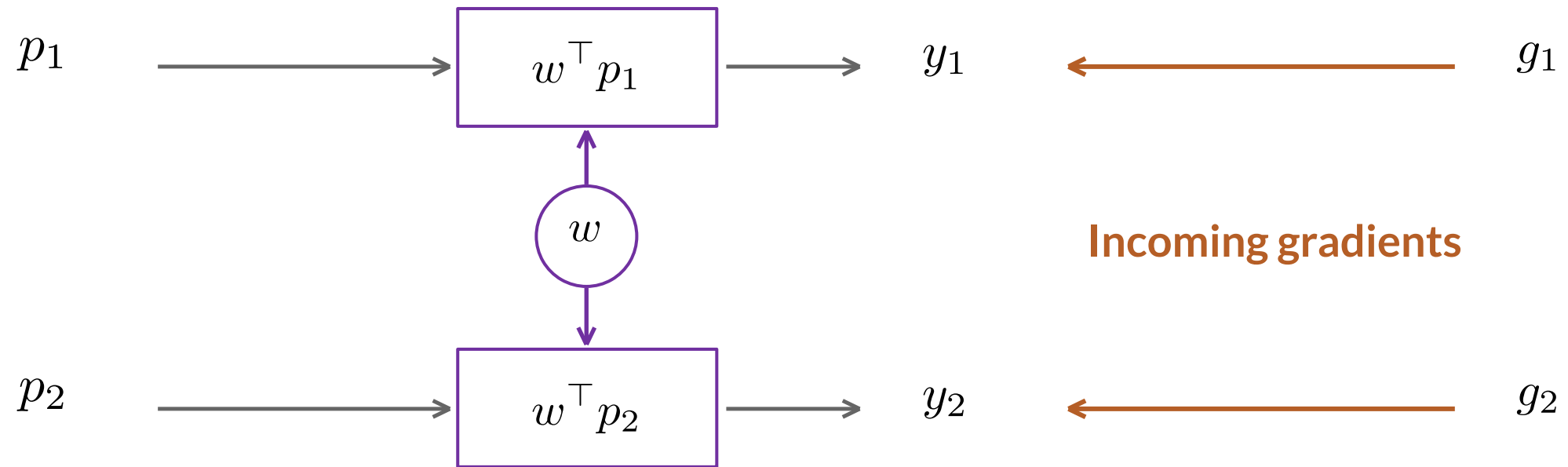
conv2 (after second conv layer)



Recomputed for each input

Gradients of shared weights

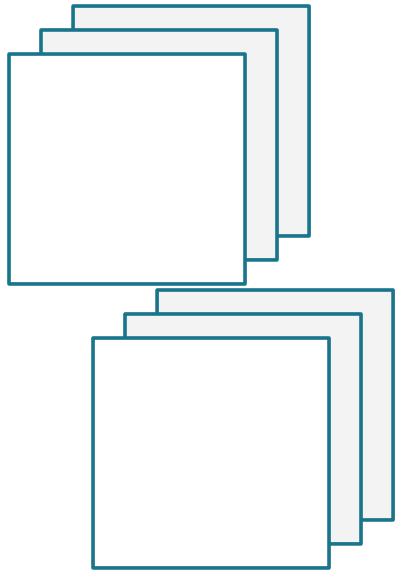
Two patches, one shared weight vector



$$\nabla_w \mathcal{L} = g_1 p_1 + g_2 p_2$$

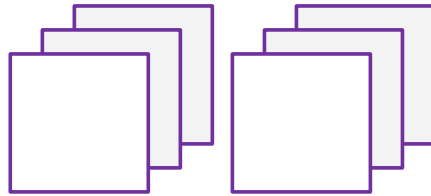
Input, weight and output shapes

Input batch



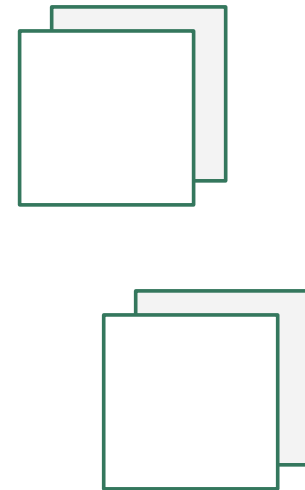
$$N \times C_{\text{in}} \times H \times W$$

Shared weights



$$C_{\text{out}} \times C_{\text{in}} \times k_h \times k_w$$

Output batch



$$N \times C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}$$

Convolution in PyTorch

```
conv = nn.Conv2d(3, 16, kernel_size=3)
z = conv(x)          # local sums plus bias
h = torch.relu(z)    # separate activation
```

Input

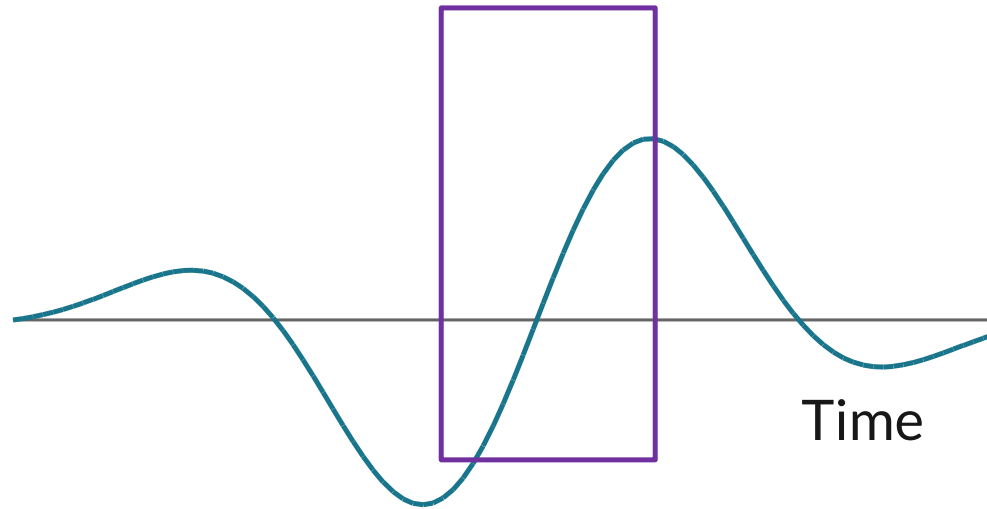
$$N \times 3 \times 32 \times 32$$

Output

$$N \times 16 \times 30 \times 30$$

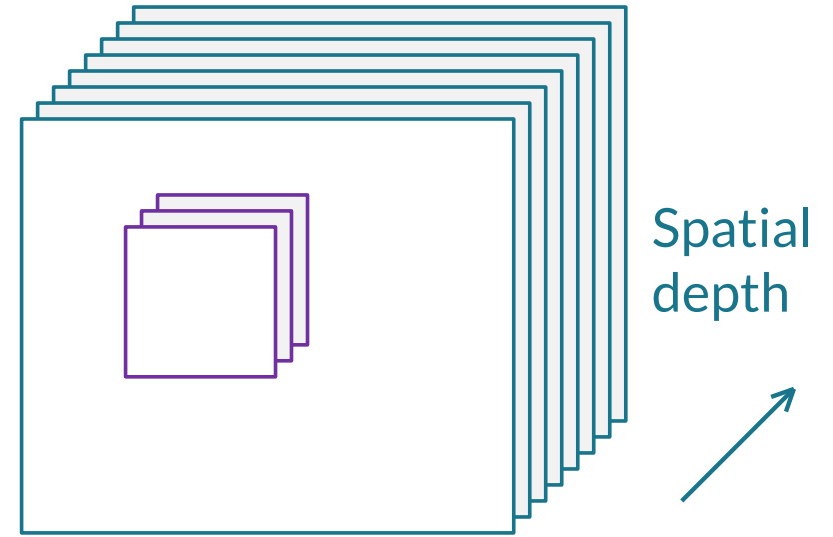
Convolution beyond images

1D convolution



A local window along time

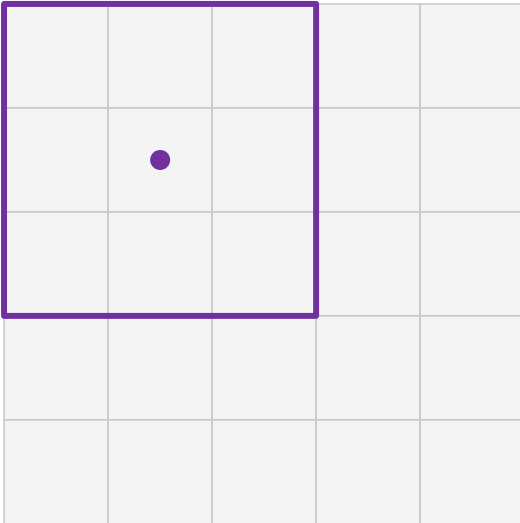
3D convolution



Neighborhood in x, y and z

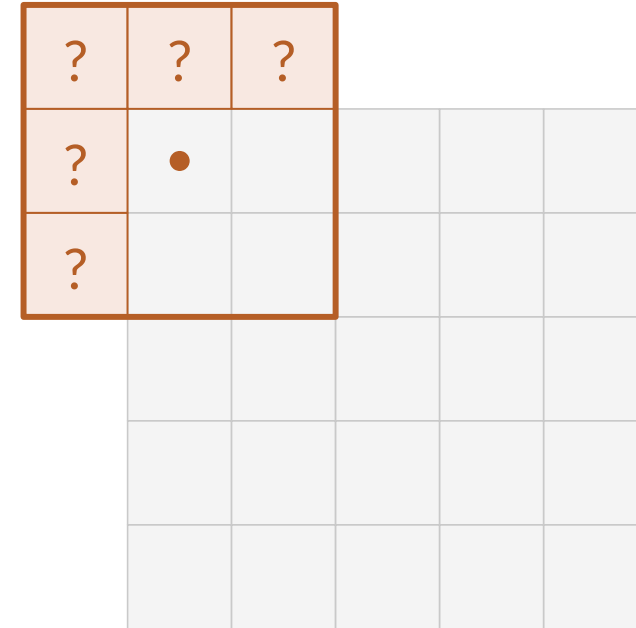
Boundary positions

A complete patch



Fully valid windows shrink the output

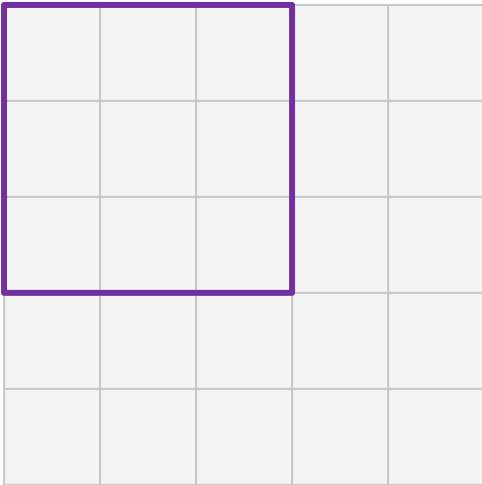
A boundary-centered patch



Some required values are missing

Padding

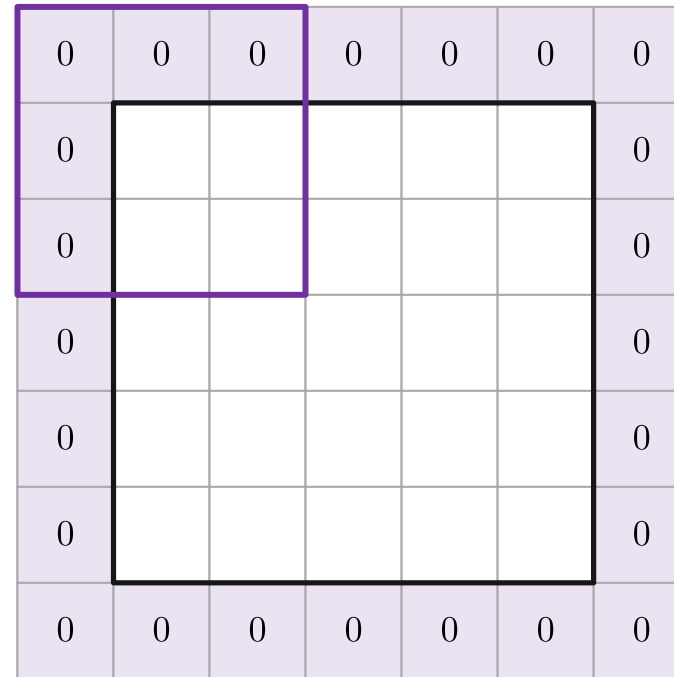
Valid windows



3×3 output



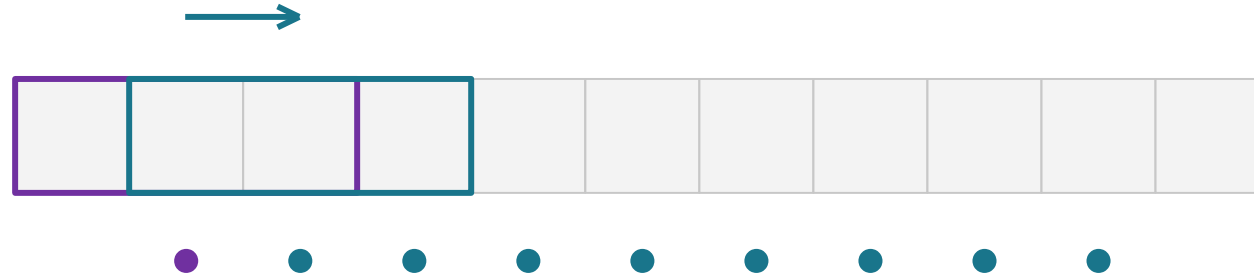
One-cell zero padding



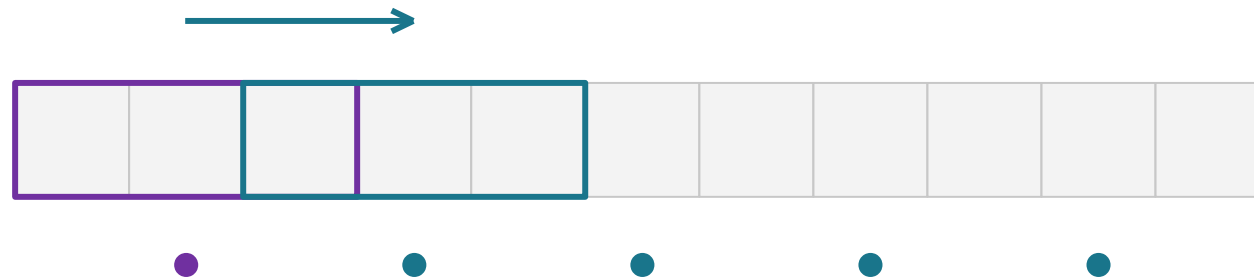
5×5 output

Stride

Stride 1



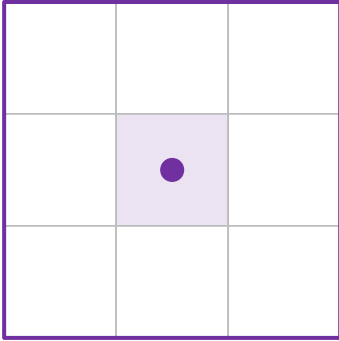
Stride 2



Evaluated window centers

Odd and even kernels

3 × 3: a central pixel

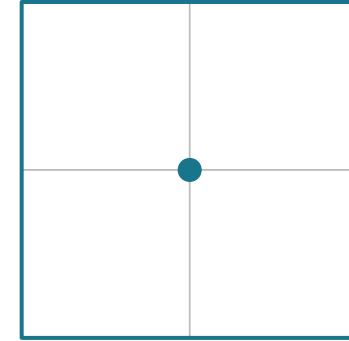


Equal reach on both sides

Stride 1: keep spatial size
with symmetric padding.

$$p = \frac{k - 1}{2} \quad \text{for odd } k$$

2 × 2: center between pixels



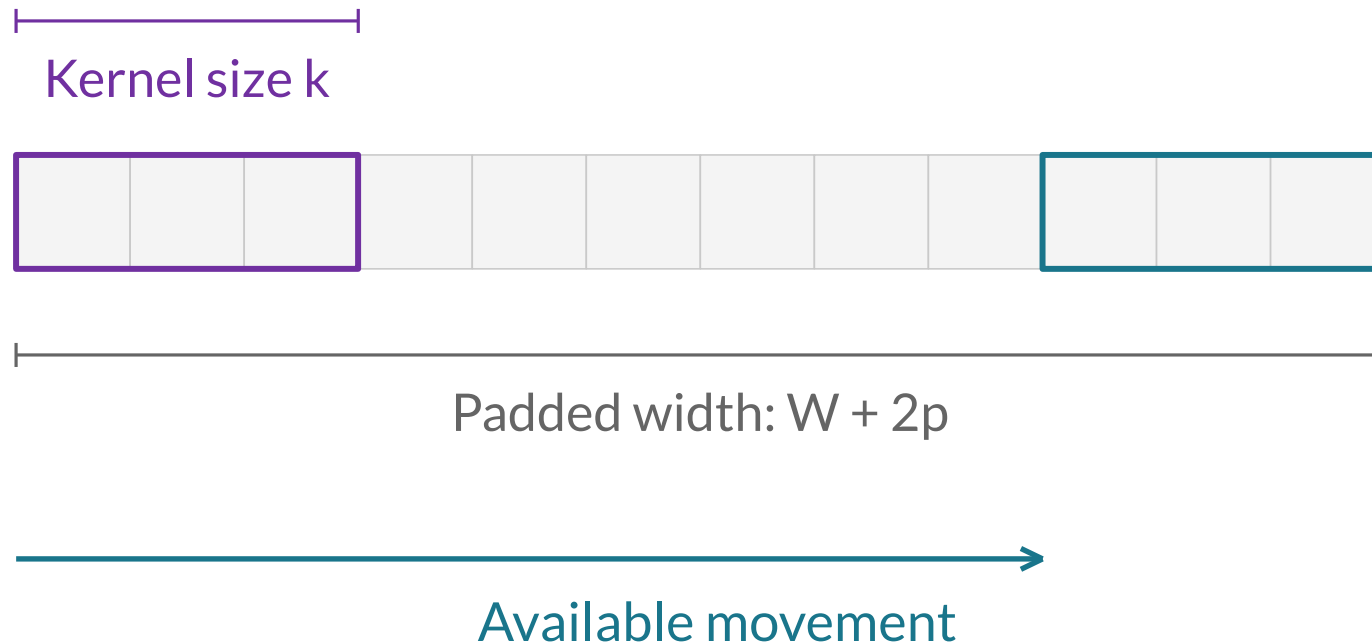
No unique central pixel

2 × 2, stride 2: useful for downsampling.

Even kernels are valid too.

Output dimensions

Count valid window placements



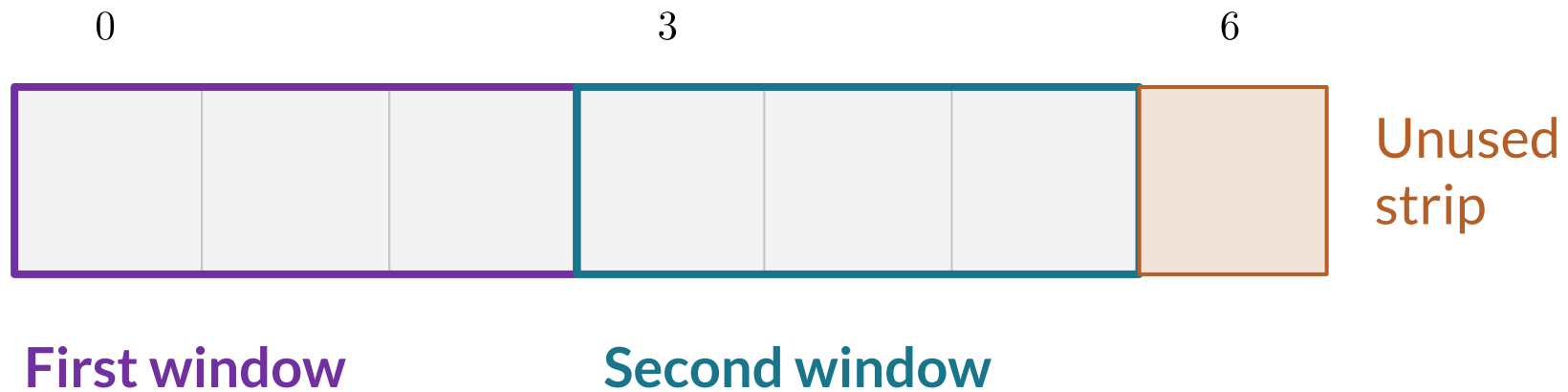
$$W_{\text{out}} = \left\lfloor \frac{W + 2p - k}{s} \right\rfloor + 1$$

p: padding

s: stride

The last valid window

$$W = 7, \quad k = 3, \quad s = 3, \quad p = 0$$

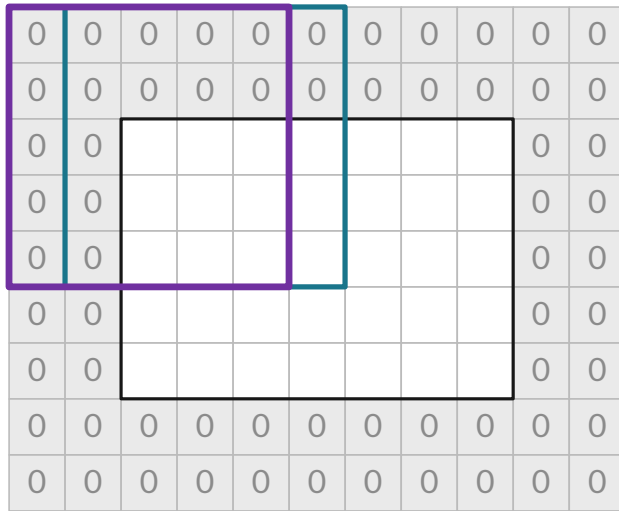


2 outputs

2D convolution: preserving spatial size

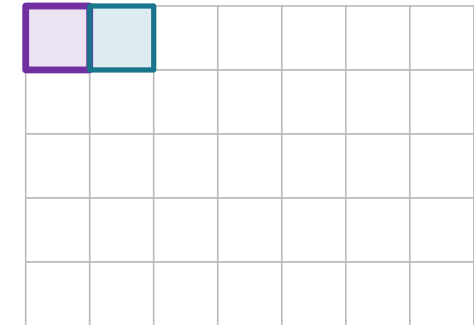
Input: 5×7

5×5 kernel · stride 1 · padding 2



One value per window

Move 1 pixel



Output: 5×7

Gray border: two zeros on each side

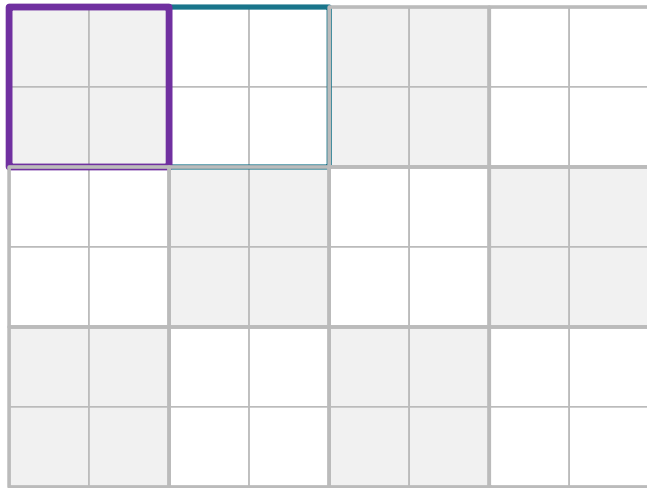
Height $\left\lfloor \frac{5 + 2 \cdot 2 - 5}{1} \right\rfloor + 1 = 5$

Width $\left\lfloor \frac{7 + 2 \cdot 2 - 5}{1} \right\rfloor + 1 = 7$

2D convolution: downsampling

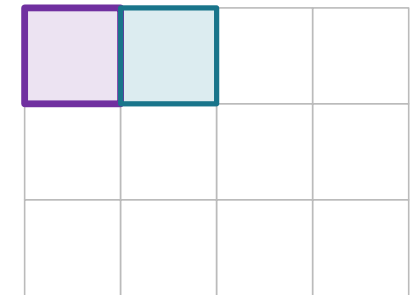
Input: 6×8

2×2 kernel · stride 2 · no padding



One value per window

Move 2 pixels



Output: 3×4

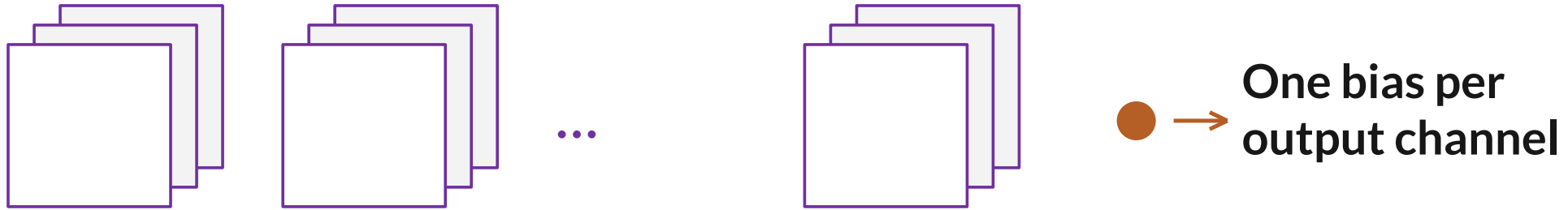
Non-overlapping 2×2 windows

Height $\left\lfloor \frac{6 + 2 \cdot 0 - 2}{2} \right\rfloor + 1 = 3$

Width $\left\lfloor \frac{8 + 2 \cdot 0 - 2}{2} \right\rfloor + 1 = 4$

Parameters in a convolution

Kernel bank



$$C_{\text{out}}(C_{\text{in}}k_hk_w + 1)$$

$$16(3 \cdot 3 \cdot 3 + 1) = 432 + 16 = 448$$

Convolution as a linear layer

One channel in 1D; stride 1; no padding; bias omitted.

Outputs

Shared weight matrix

Flattened input

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} k_1 & k_2 & k_3 & 0 & 0 & 0 \\ 0 & k_1 & k_2 & k_3 & 0 & 0 \\ 0 & 0 & k_1 & k_2 & k_3 & 0 \\ 0 & 0 & 0 & k_1 & k_2 & k_3 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

Toeplitz structure: each diagonal repeats the same weight.

Images use the corresponding block-Toeplitz structure.

Convolution with matrix multiplication

im2col: put each overlapping input patch into a column.

Filter weights

$$\begin{bmatrix} k_1 & k_2 & k_3 \end{bmatrix}$$

$\times$

Input patches

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ x_2 & x_3 & x_4 & x_5 \\ x_3 & x_4 & x_5 & x_6 \end{bmatrix}$$

$=$

Output values

$$\begin{bmatrix} z_1 & z_2 & z_3 & z_4 \end{bmatrix}$$

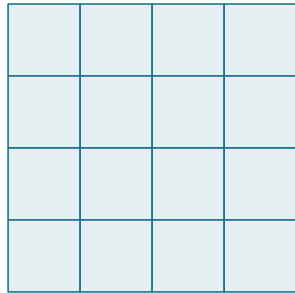
Many filters: stack their weights as rows.

Optimized libraries can avoid storing the full patch matrix.

Image size and convolution cost

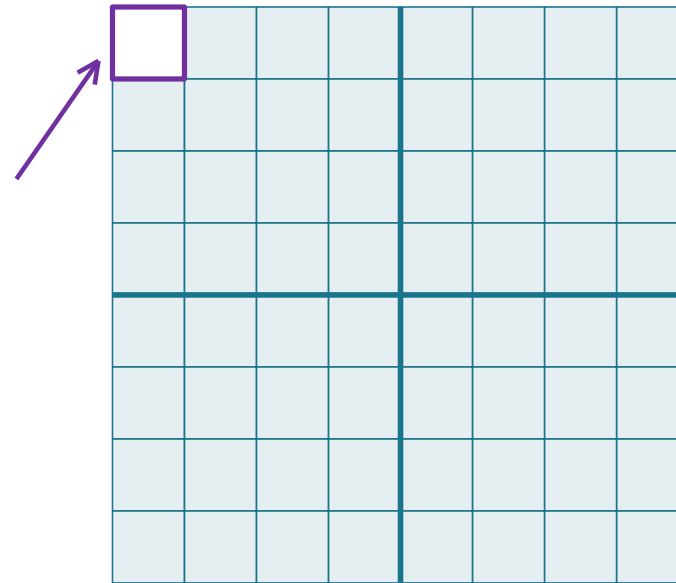
Fixed filters and channels; stride 1; padding preserves spatial size.

Output positions



$$H \times W$$

Each cell requires one application of the same filter.



$$2H \times 2W = 4HW$$

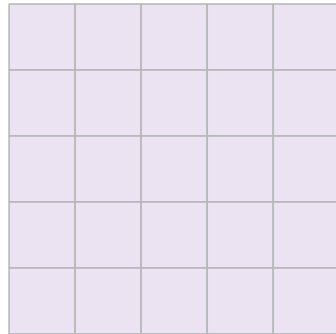
Both use the same weights

4× output values and multiply-add work

Stacking convolutional layers

One linear filter

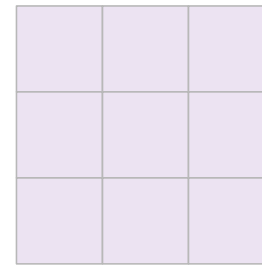
5×5 Conv



$$z = w^{\top} x + b$$

Composition with ReLU

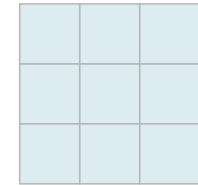
3×3 Conv



ReLU



3×3 Conv



$$z = v^{\top} \text{ReLU}(Wx + b) + c$$

Nonlinear combinations of local responses

Receptive field

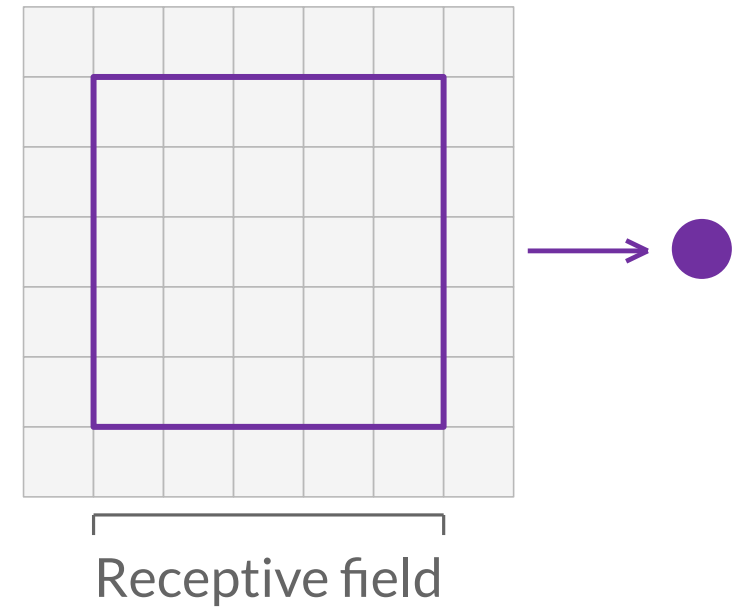
Local crop



More context

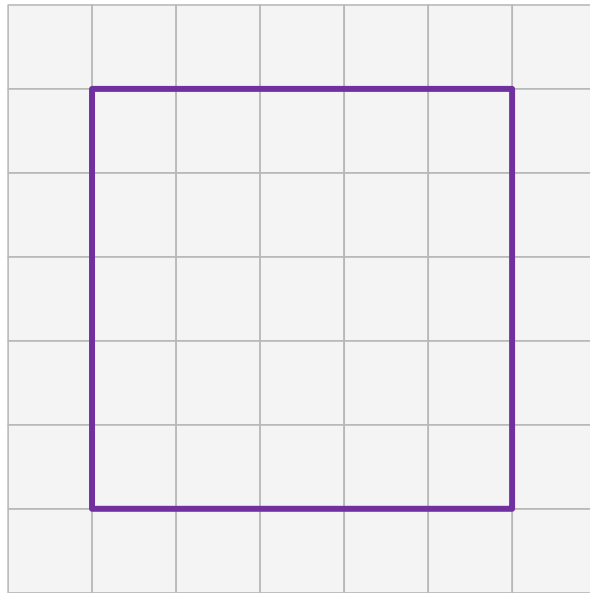


One output activation



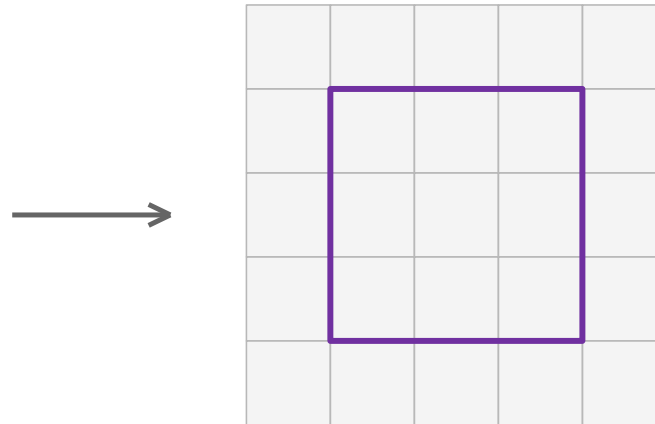
Receptive fields through layers

Input



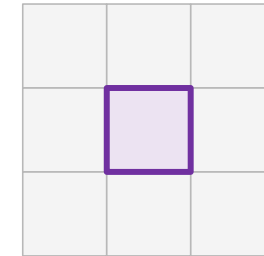
5×5 input region

After 3×3 convolution



3×3 neighbors

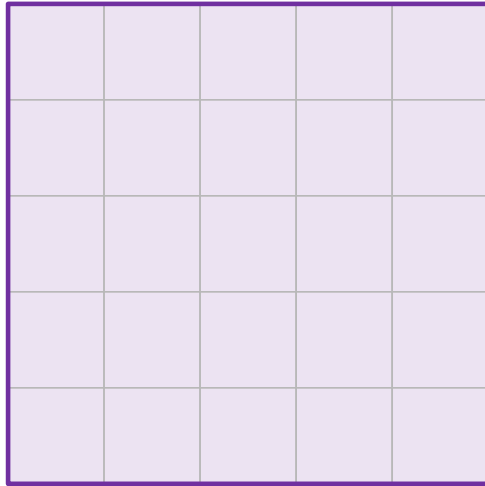
After another 3×3



Selected output

Stacking small kernels

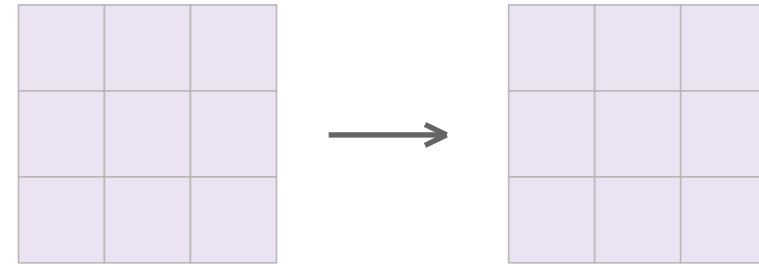
One 5×5 layer



$$25C^2$$

Equal channel width C throughout; biases omitted.

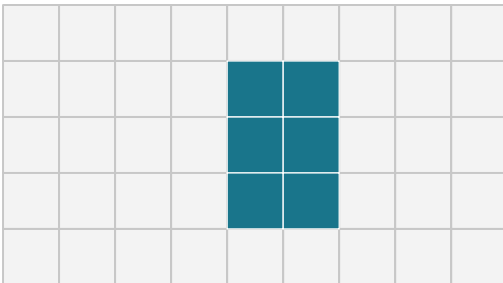
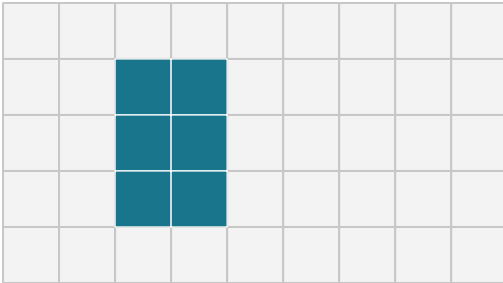
Two 3×3 layers



$$9C^2 + 9C^2 = 18C^2$$

Shifted inputs and feature maps

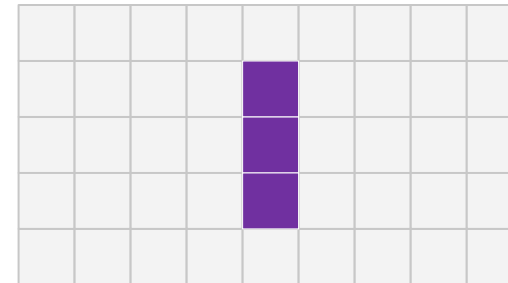
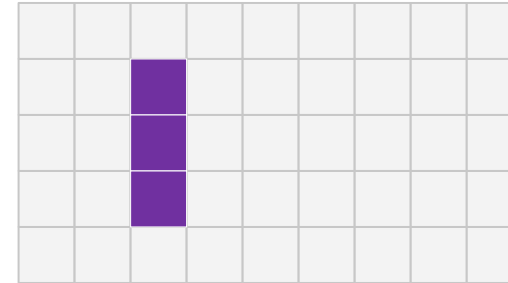
Input



same filter



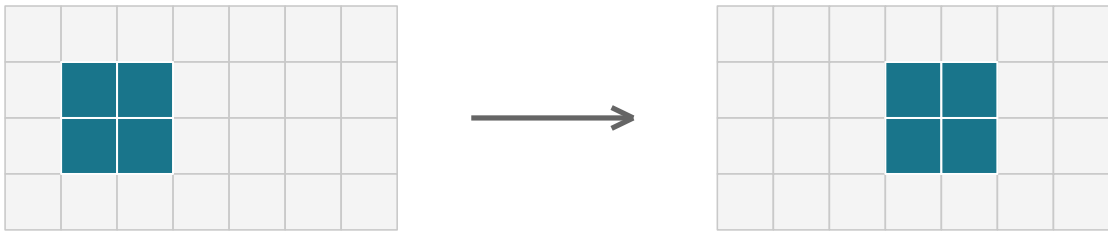
Response map



Schematic: stride 1, away from image boundaries

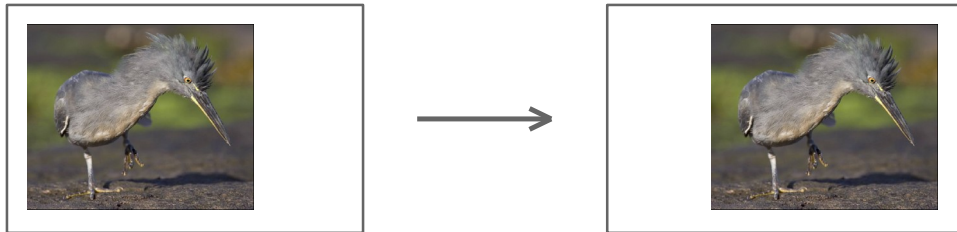
Shifts in features and class labels

Feature maps: equivariance



Feature responses move with the input pattern

Class label: desired invariance



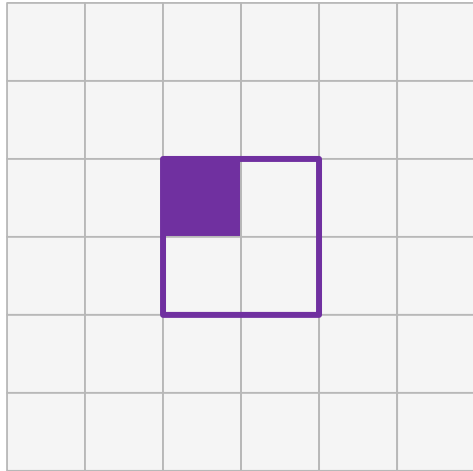
Bird

The class label stays the same

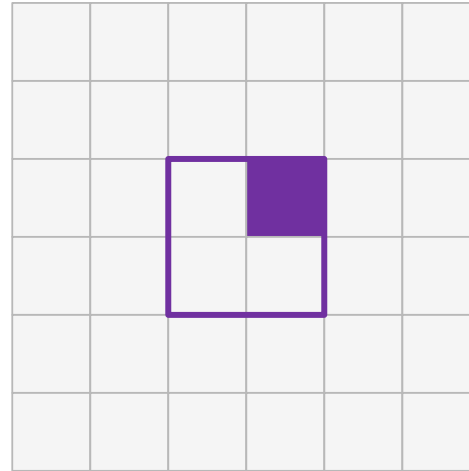
The classifier must combine responses across locations.

Spatial resolution in a classifier

Nearby responses



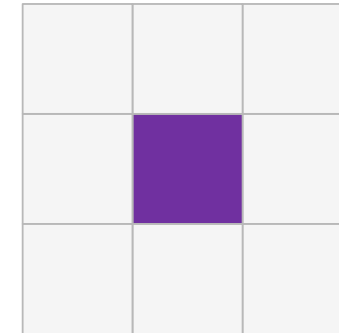
Original response



Slight shift



Coarser spatial map



Preserve the response
within a neighborhood

Less spatial detail

Less computation in later layers

Max pooling

One input channel

1	3	2	0
2	6	1	4
0	1	3	2
5	2	4	0

$$\max(1, 3, 2, 6) = 6$$



2×2 window · stride 2

Pooled output

6	4
5	4

Apply separately to every channel.

```
nn.MaxPool2d(kernel_size=2, stride=2)
```

Average pooling

One strong response

0	0
0	8

Max: 8

$$\text{Average} = \frac{0 + 0 + 0 + 8}{4} = 2$$

Max keeps a peak.

Several moderate responses

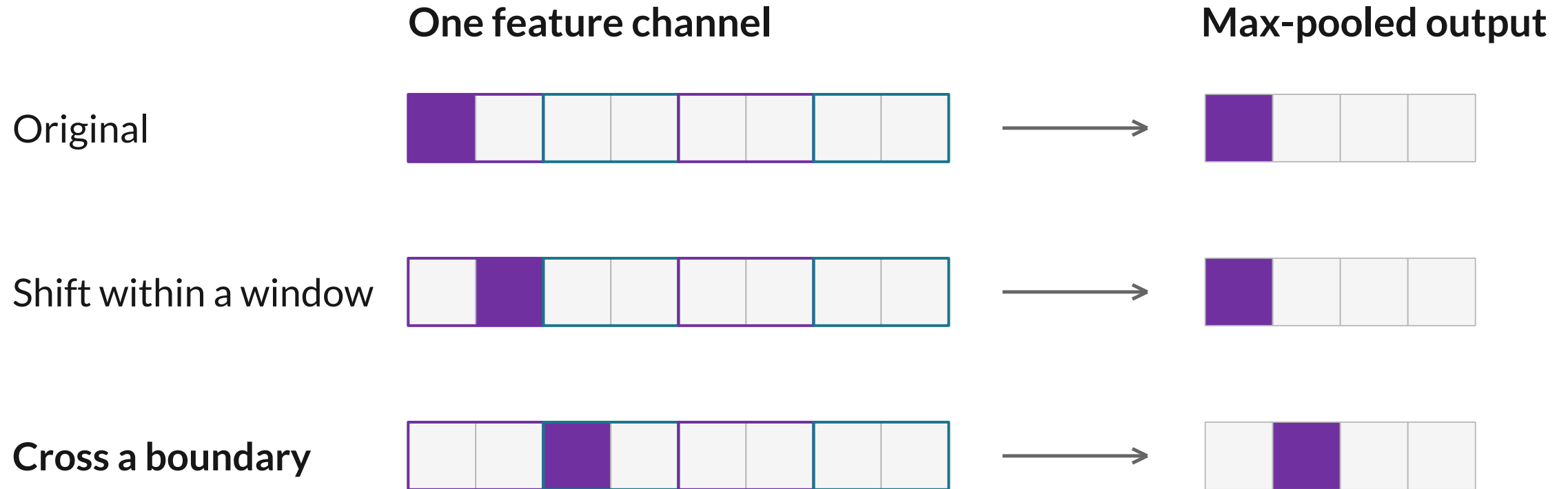
3	3
3	3

Max: 3

$$\text{Average} = \frac{3 + 3 + 3 + 3}{4} = 3$$

Average reflects all responses.

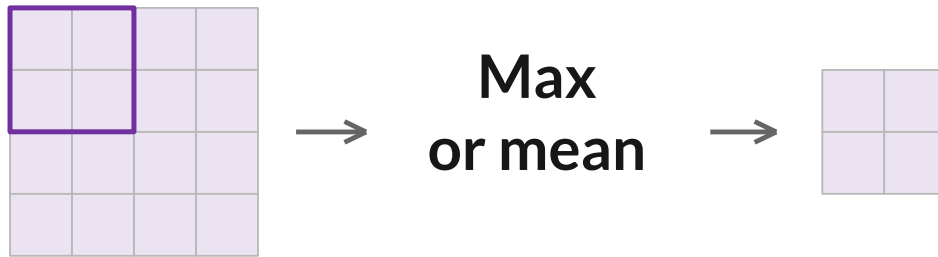
Small shifts and pooling



Pooling can reduce sensitivity to small shifts; it does not guarantee invariance.

Pooling and strided convolution

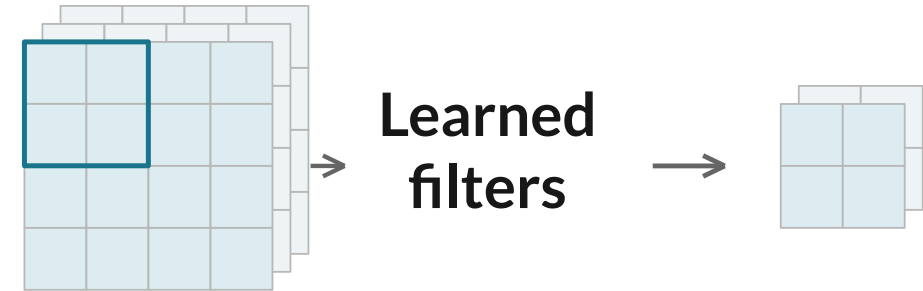
Pooling



One summary per channel

No learned weights

Strided convolution



Can mix and change channels

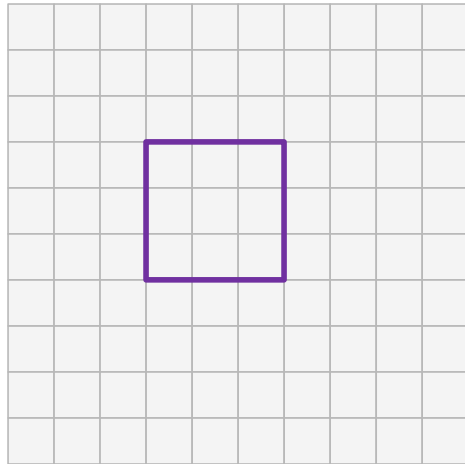
Learned weights

Both can reduce spatial resolution.

Receptive fields with downsampling

Theoretical receptive field in the input image

Conv 3×3

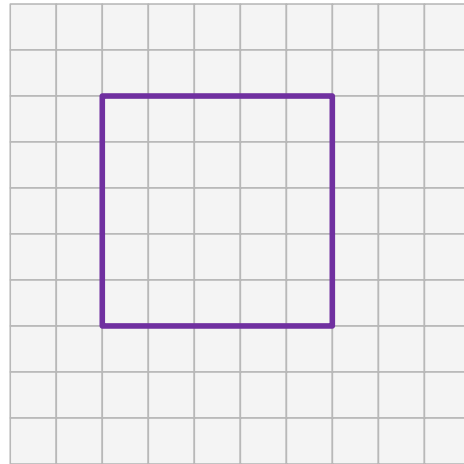


Input span: 3

Center spacing: 1



Conv 3×3

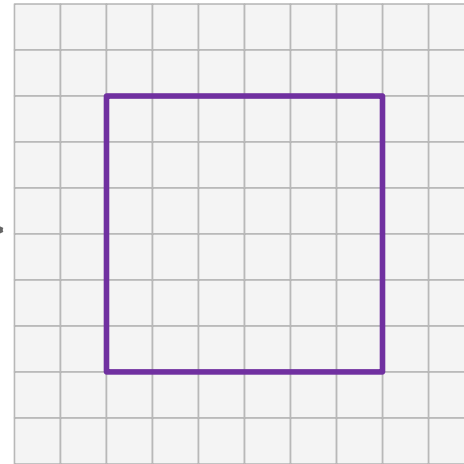


Input span: 5

Center spacing: 1



Pool 2×2, stride 2

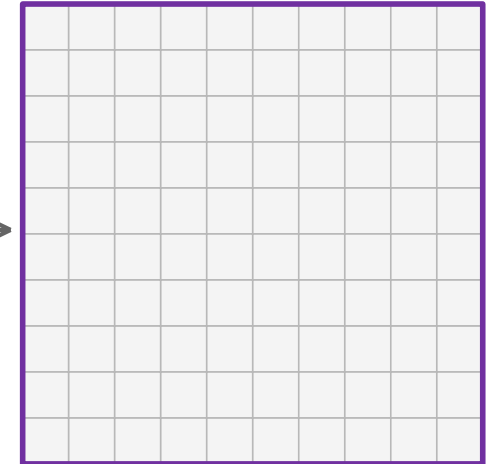


Input span: 6

Center spacing: 2



Conv 3×3



Input span: 10

Center spacing: 2

Spatial detail after downsampling

Fine spatial detail

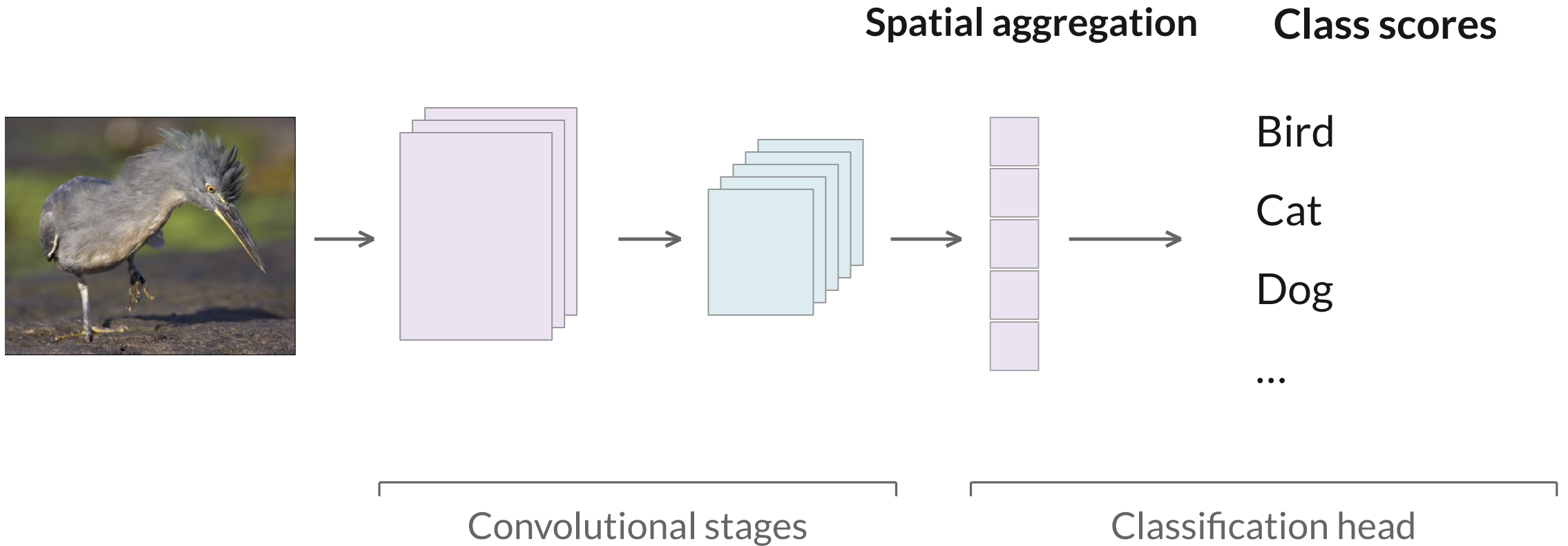


After downsampling



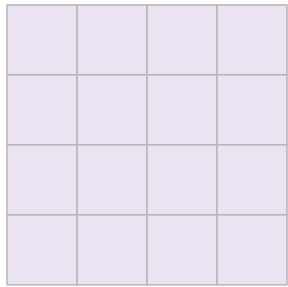
Local averaging and a coarser grid

Convolutional classifier



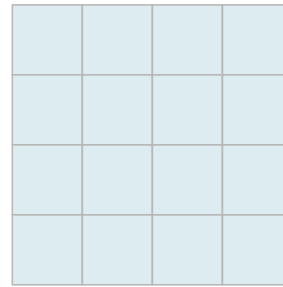
Global average pooling

Channel 1

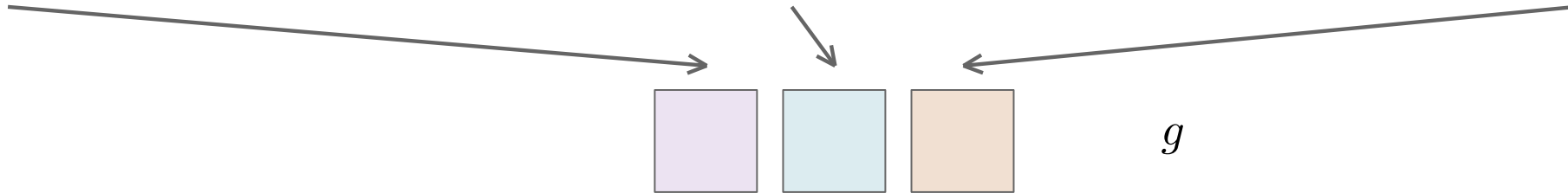
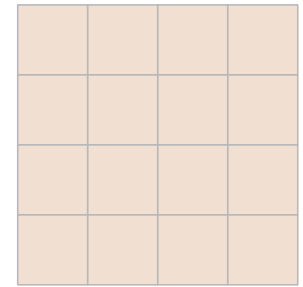


h

Channel 2



Channel 3



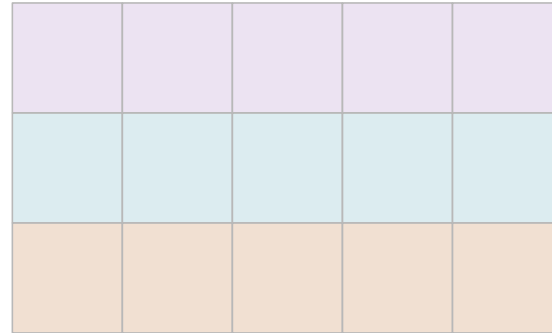
$$g_c = \frac{1}{HW} \sum_{i=1}^H \sum_{j=1}^W h_{c,i,j}$$

Classification head

Pooled features



Learned linear map



Class logits



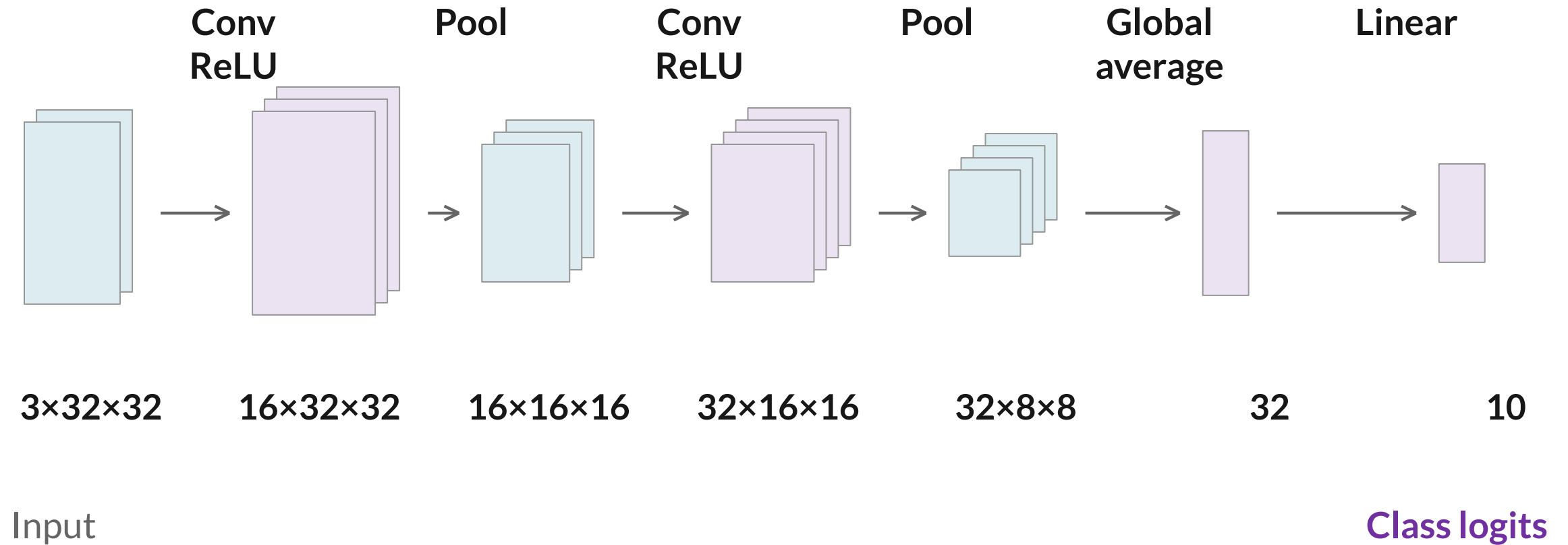
Bird

Cat

Dog

$$z = Wg + b$$

Complete CNN



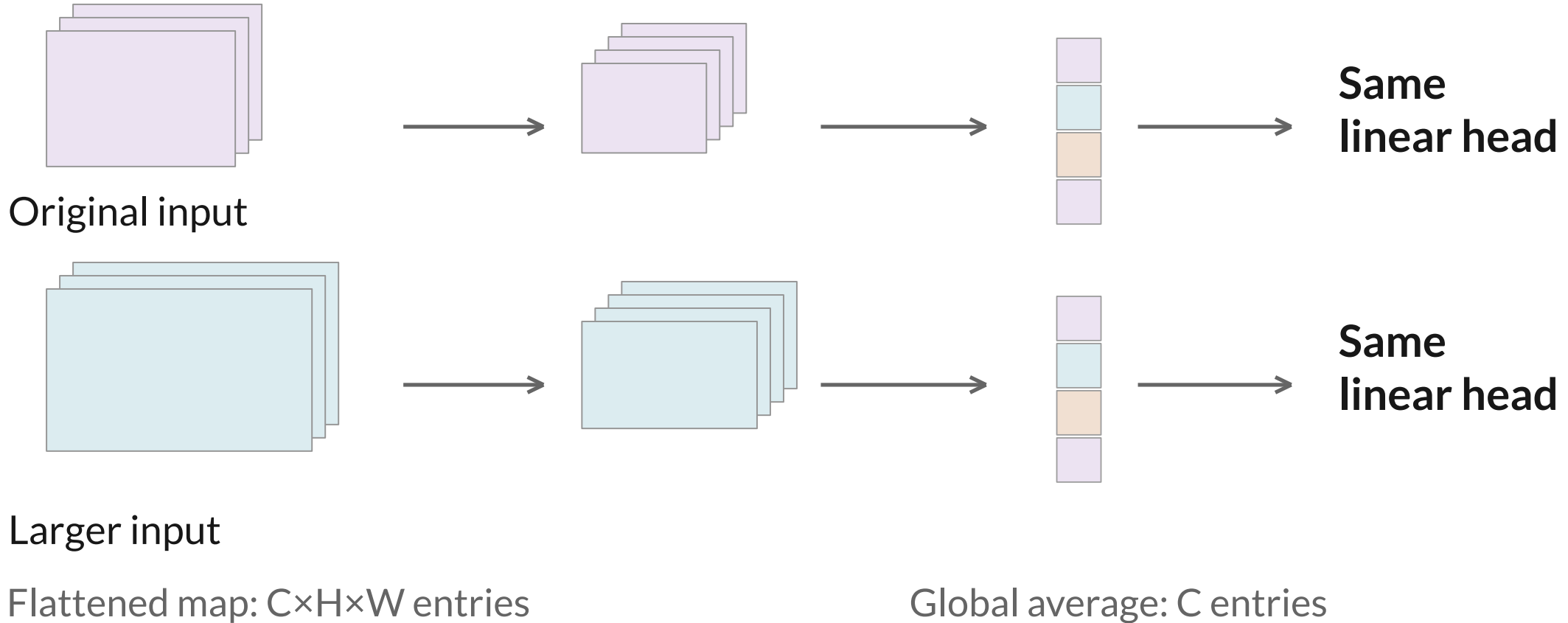
CNN in PyTorch

```
model = nn.Sequential(  
    nn.Conv2d(3, 16, 3, padding=1),    # 16 feature maps  
    nn.ReLU(),                        # elementwise  
    nn.MaxPool2d(2),                  # halve spatial size  
    nn.Conv2d(16, 32, 3, padding=1),  # mix input channels  
    nn.ReLU(),                        # elementwise  
    nn.MaxPool2d(2),                  # halve spatial size  
    nn.AdaptiveAvgPool2d(1),          # one mean per channel  
    nn.Flatten(1),                    # keep the batch axis  
    nn.Linear(32, 10),                # class logits  
)
```

Input size and the head

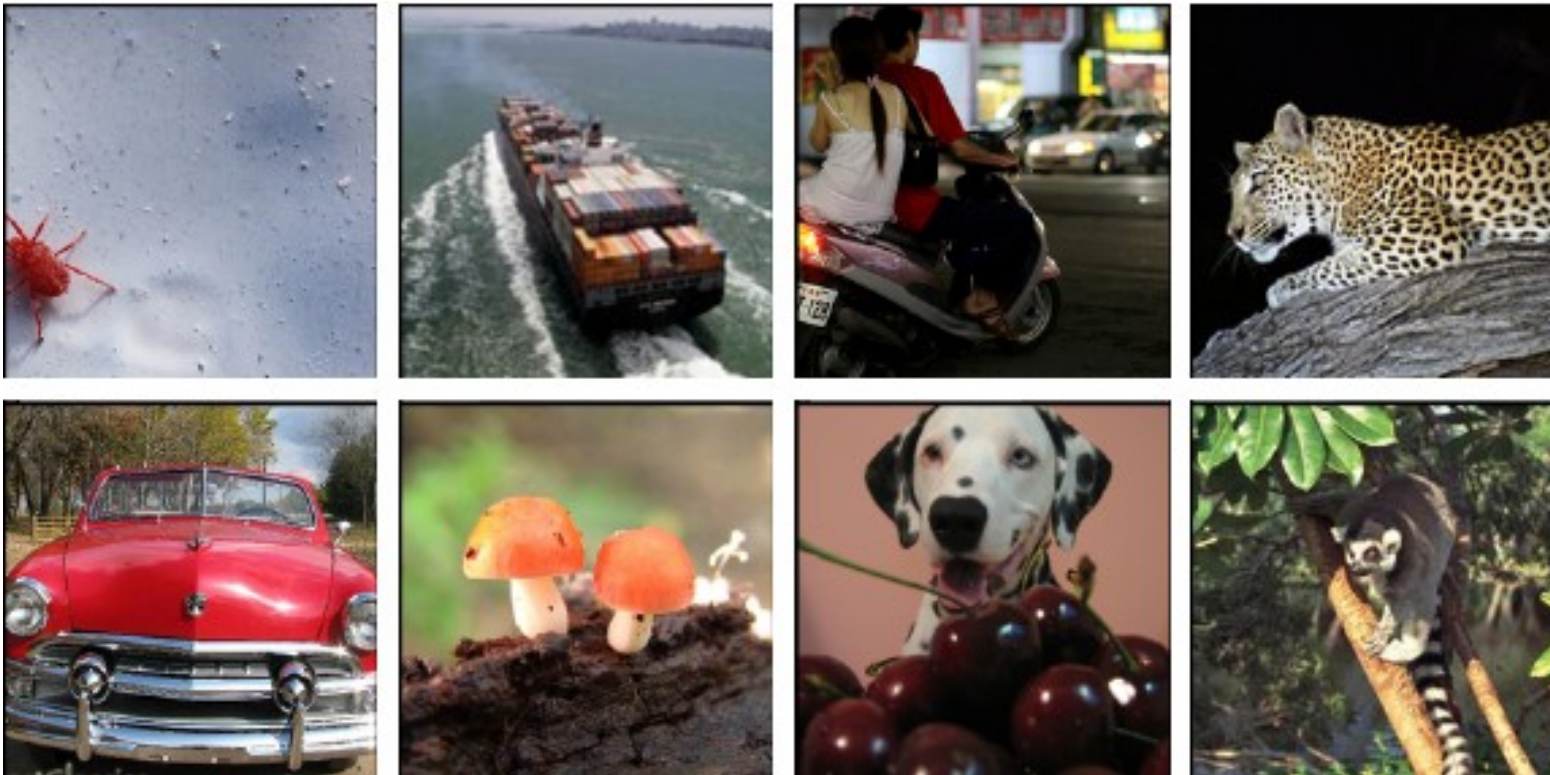
Spatial maps can change size

The feature vector keeps C entries



ImageNet classification

ImageNet-1k / ILSVRC classification



≈1.2 million

training images

1,000

categories

Top-1 and top-5 accuracy



Target: Bird

Ranked classes

1	Dog	Top-1
2	Cat	
3	Bird	Top-5 includes Bird
4	Plane	
5	Ship	

Training and evaluation images

Images arrive with different sizes and shapes



Training

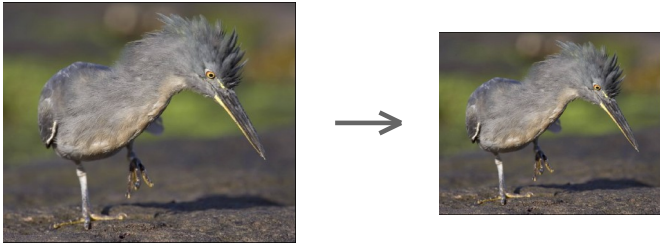
Compatible inputs and varied views

Evaluation

A fixed procedure for comparison

Resizing and cropping

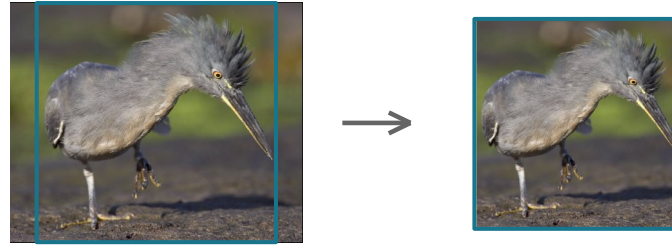
Resize



New pixel grid

Whole field of view

Crop



Select a region

Smaller field of view

Reshape

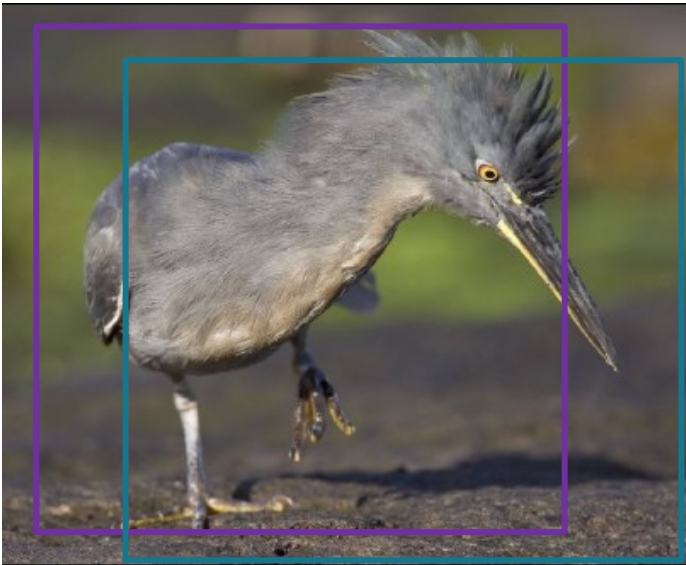


Same values

A different tensor shape

Random training crops

Sample a crop region



Random crop: vary location.



Different sampled views



Random resized crop: also vary area and shape, then resize to the required input size.

Center-crop evaluation

Original image



Resize the shorter edge



Preserve aspect ratio



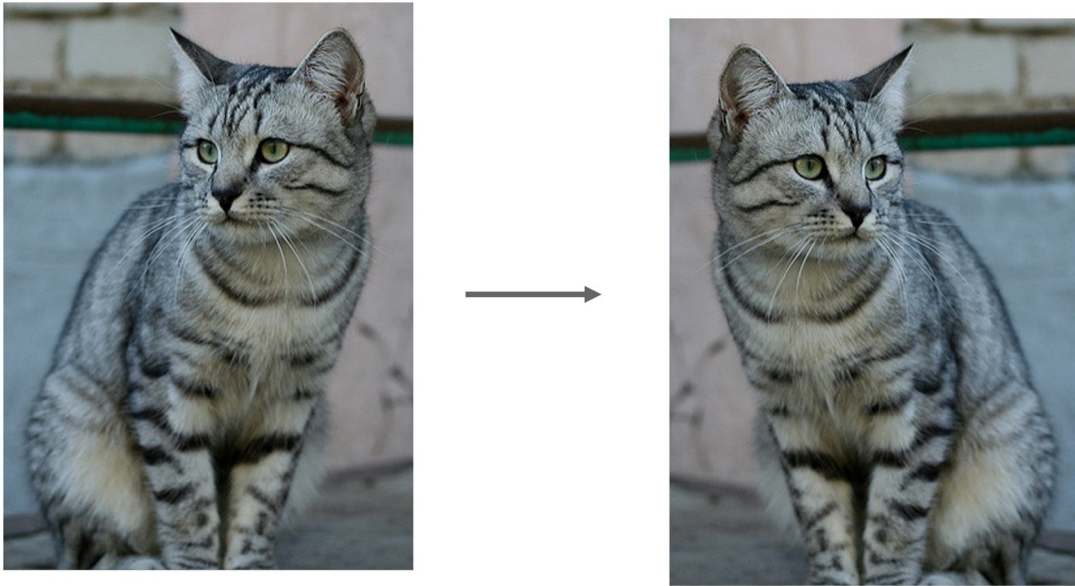
Center crop



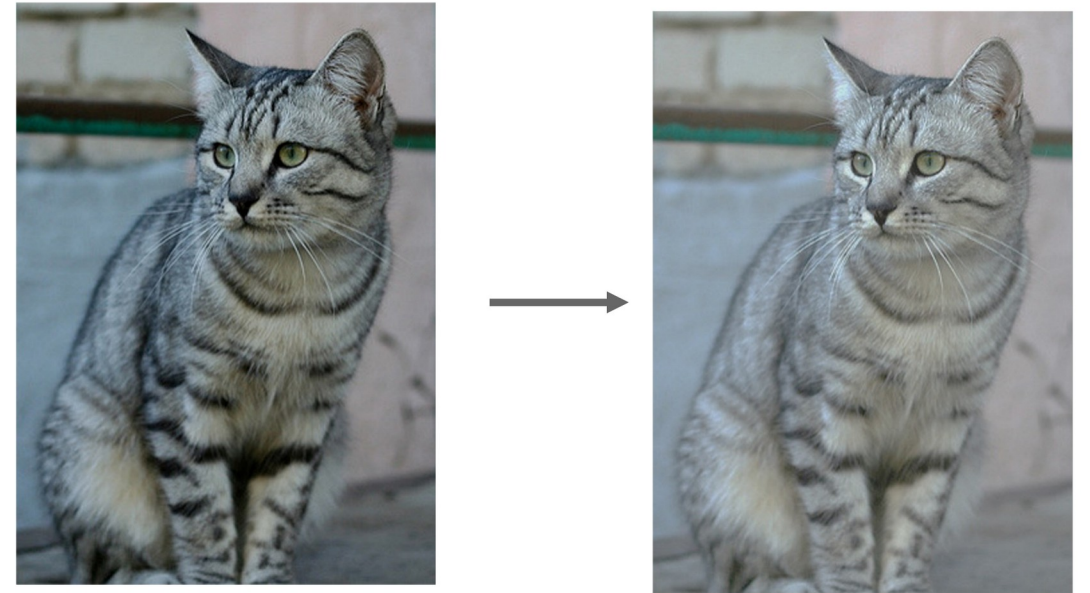
Same view each time

Image augmentation

Horizontal flip

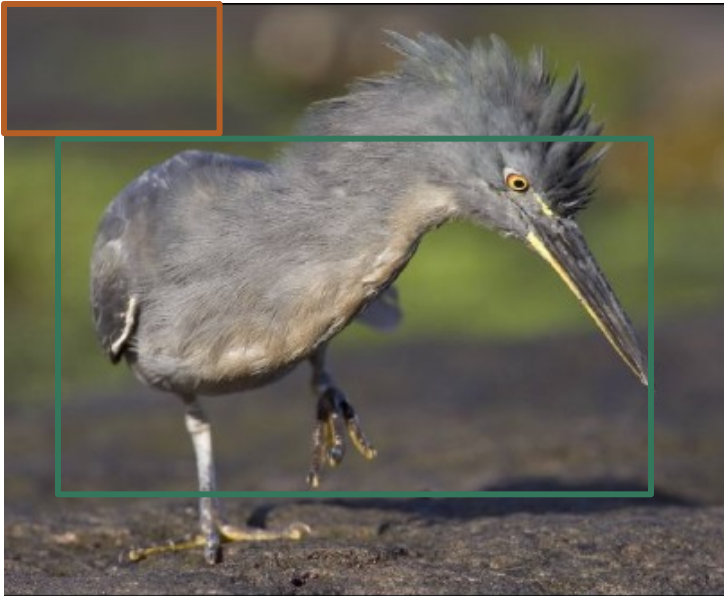


Lighting variation



Cropping and class labels

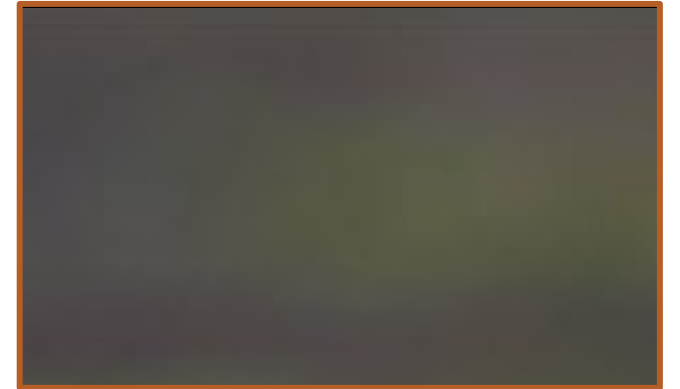
Original image



Bird remains visible



Bird cropped out



Training label: bird

Preprocessing and model mode

```
from torchvision import transforms as T
```

Training

```
train_tf = T.Compose([
    T.RandomResizedCrop(224),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    normalize,
])
model.train()
```

Evaluation

```
eval_tf = T.Compose([
    T.Resize(256),
    T.CenterCrop(224),
    T.ToTensor(),
    normalize,
])
model.eval()
```

Transforms prepare images. Model mode controls layer behavior.

Lecture outline

01 Images and convolution

02 Convolution, spatial structure and classification

| 03 CNN architectures

AlexNet and large-scale recognition

ImageNet classification, 2012

Top-5 classification error



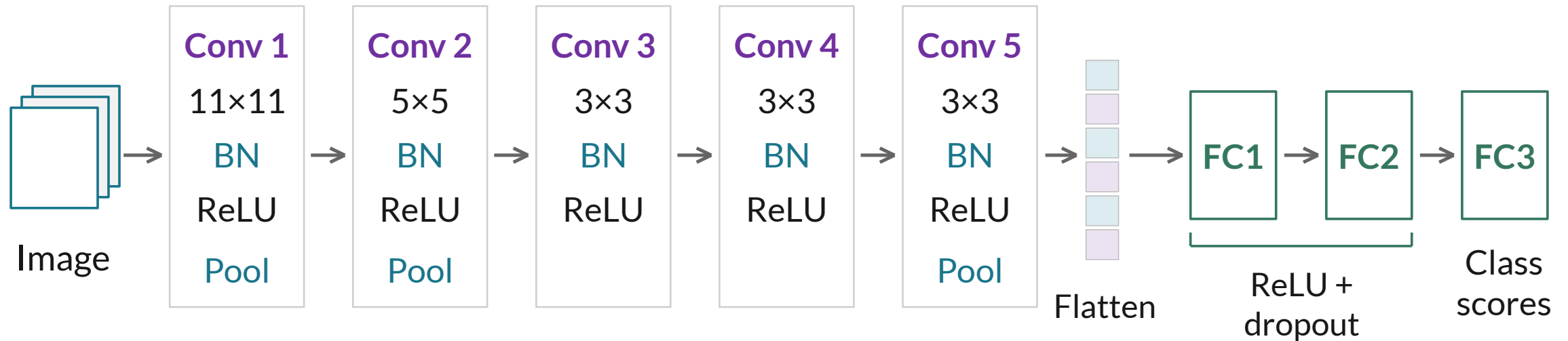
A deep convolutional model

Large labeled data
GPU computation
ReLU-based training

Entries using the supplied training data

AlexNet

AlexNet-style network with BatchNorm



BN: BatchNorm

FC: fully connected

Pool: max pooling after Conv 1, 2 and 5

Original AlexNet: LRN (local response normalization) after the first two ReLUs.

Learned first-layer filters

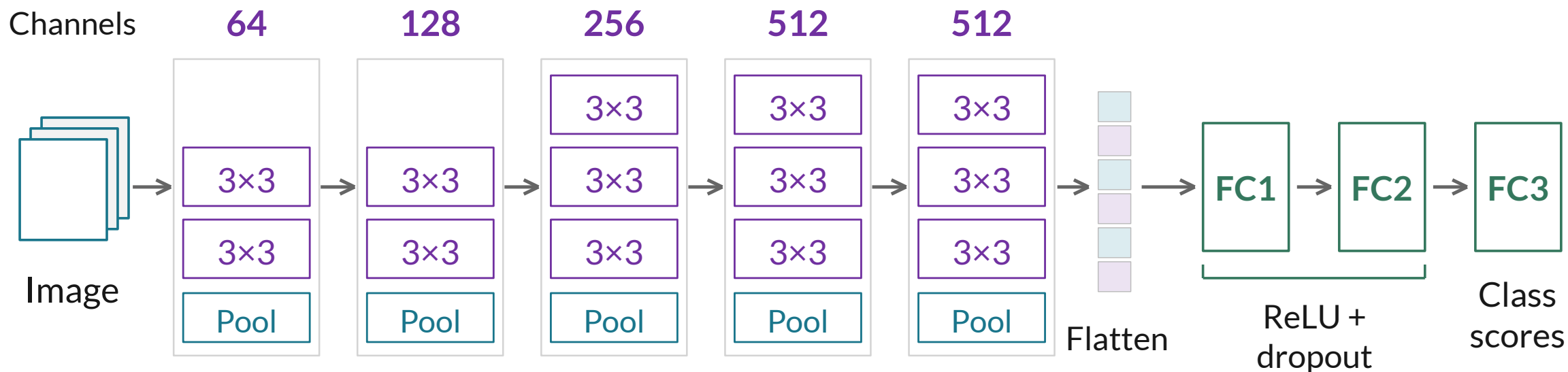
Learned first-layer RGB kernels



VGGNet

VGG-16-BN: 13 convolutional + 3 fully connected layers

Each convolution: $3 \times 3 \rightarrow \text{BN} \rightarrow \text{ReLU}$



Convolution: stride 1, padding 1

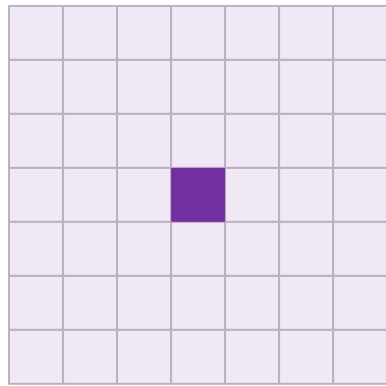
Max pooling: 2×2 , stride 2

The original VGGNet did not use BatchNorm.

Small kernels in VGGNet

Same receptive-field span, different computation

One 7×7 convolution

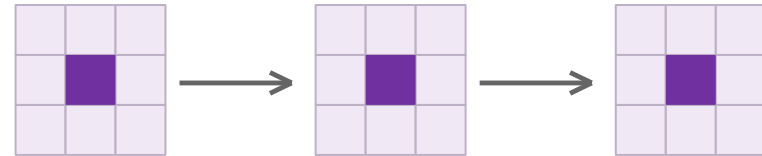


$49C^2$
weights

One nonlinear stage

C input and output channels in every layer; stride 1.

Three 3×3 convolutions



$$3 \times 9C^2 = 27C^2$$

Three nonlinear stages

Inception: capacity and computation

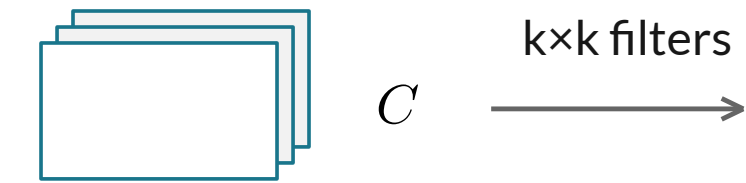
2014: more capacity within a computation budget

Widening every layer becomes expensive quickly.

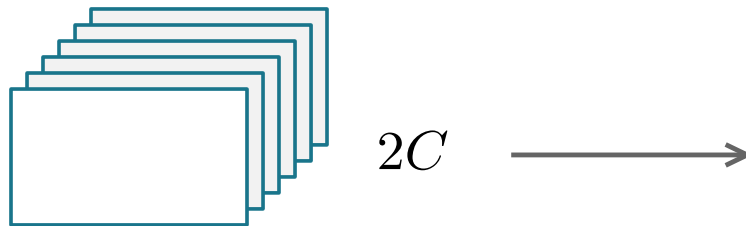
Input channels

Output channels

Work at each output position



$$k^2 C^2$$

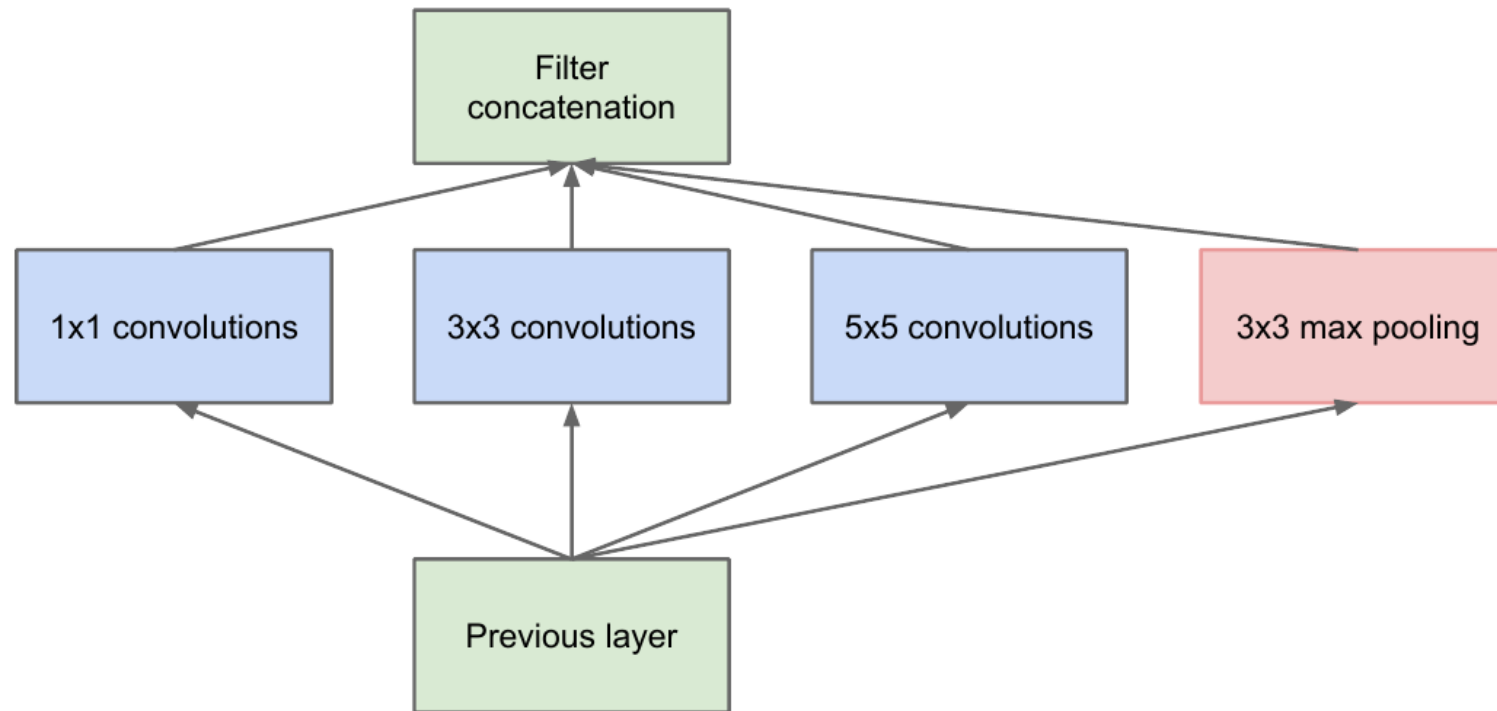


$$k^2 (2C)(2C) = 4k^2 C^2$$

Inception uses several smaller branches and learned channel reductions.

Inception

Fine patterns and wider context need different spatial scales.



Combining parallel branches

Keep each branch's feature maps as separate channels.

1×1 branch



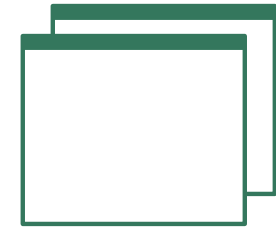
3×3 branch



5×5 branch



Pooling branch



Same height and width in every branch

$$C_{\text{out}} = C_1 + C_2 + C_3 + C_{\text{pool}}$$

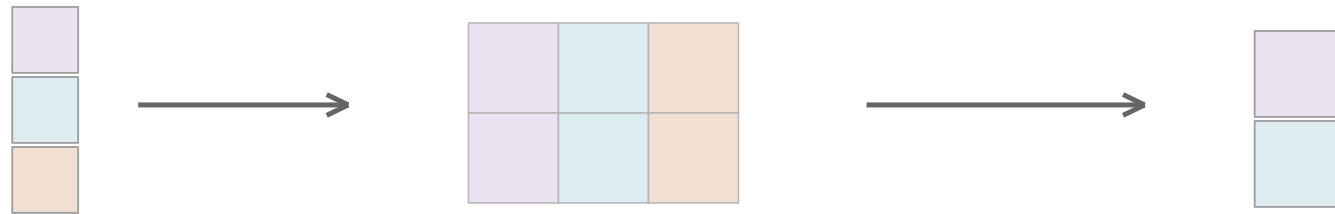


1×1 convolution

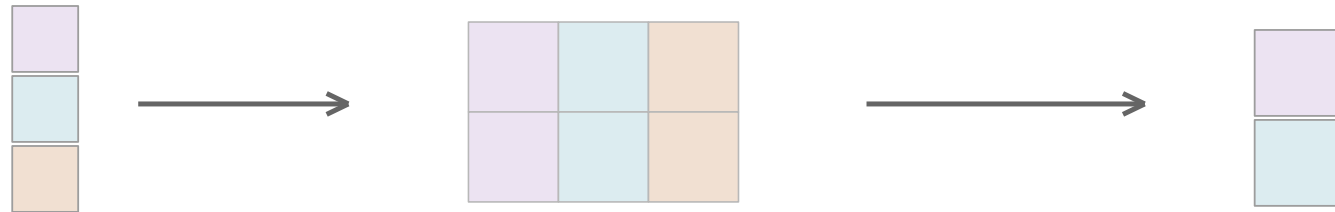
$$y_{:,i,j} = W x_{:,i,j} + b$$

Same learned matrix W

Position A



Position B



Input x

Output y

(i, j) : image position b : bias

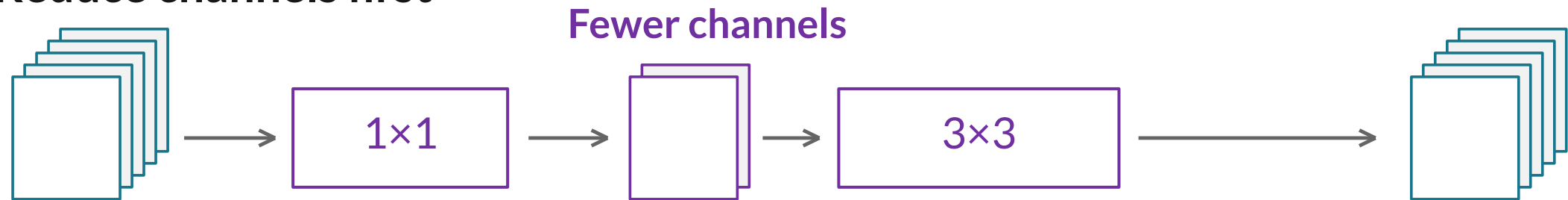
Channel bottlenecks

Full channel width



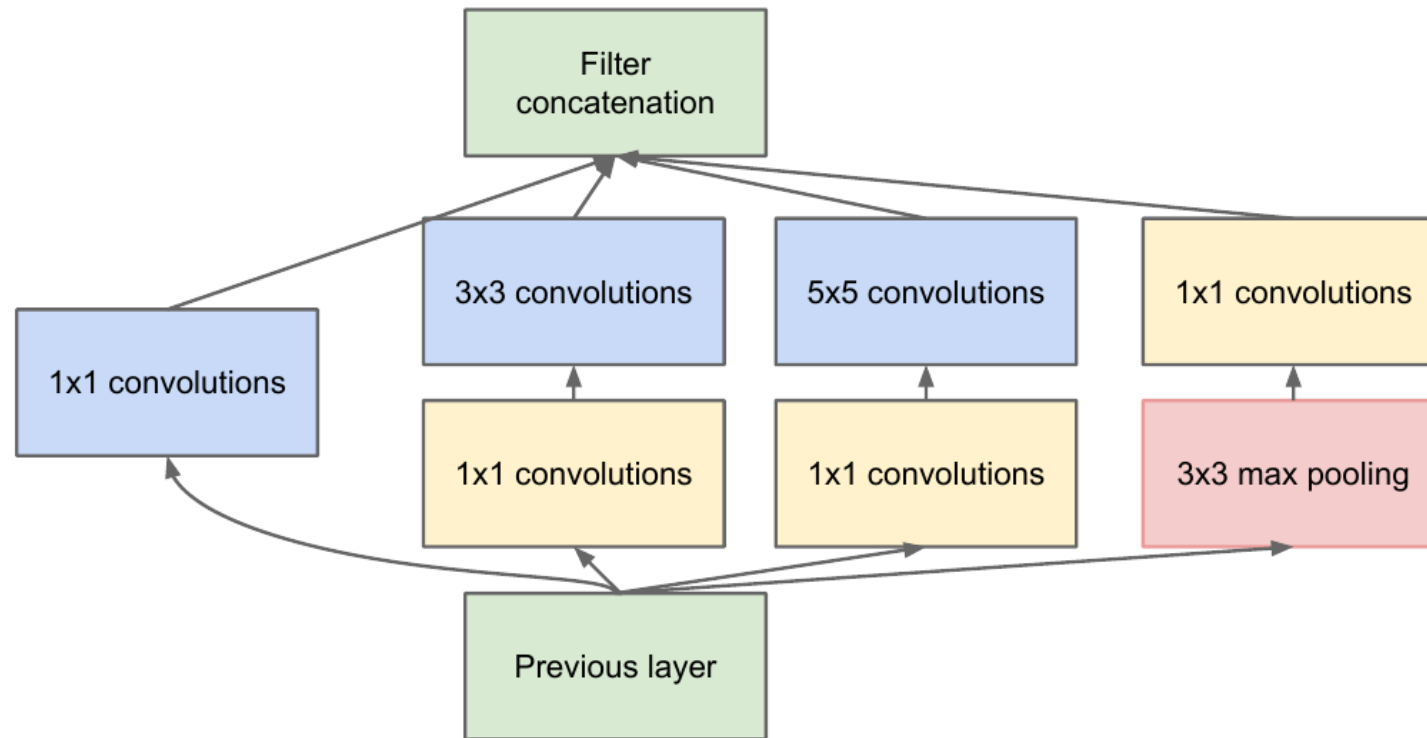
Many input channels enter every spatial filter

Reduce channels first



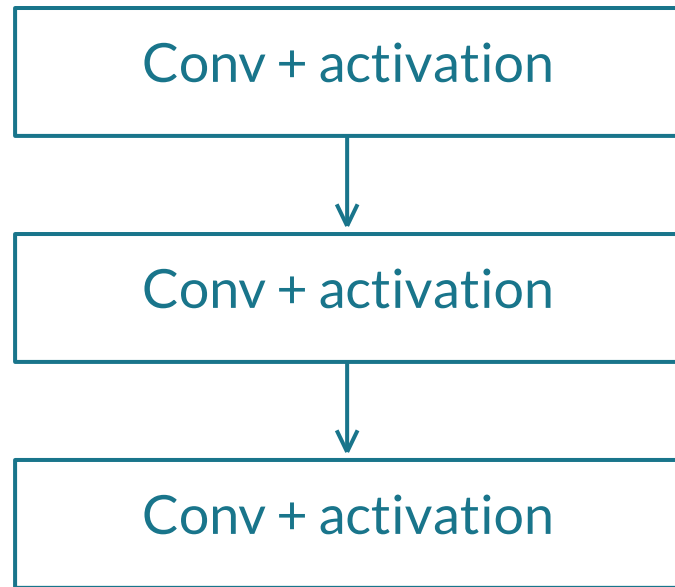
Inception block

1×1 + ReLU reduces channels before the expensive filters.

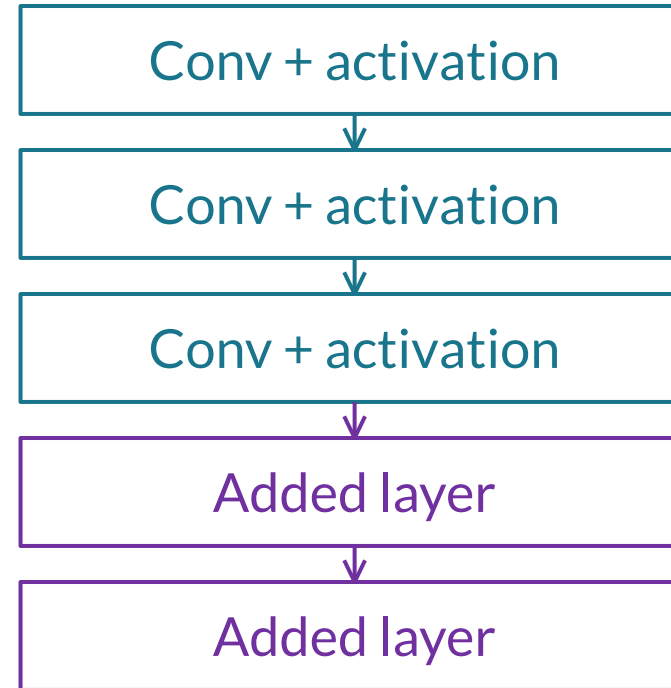


Increasing depth

Plain network

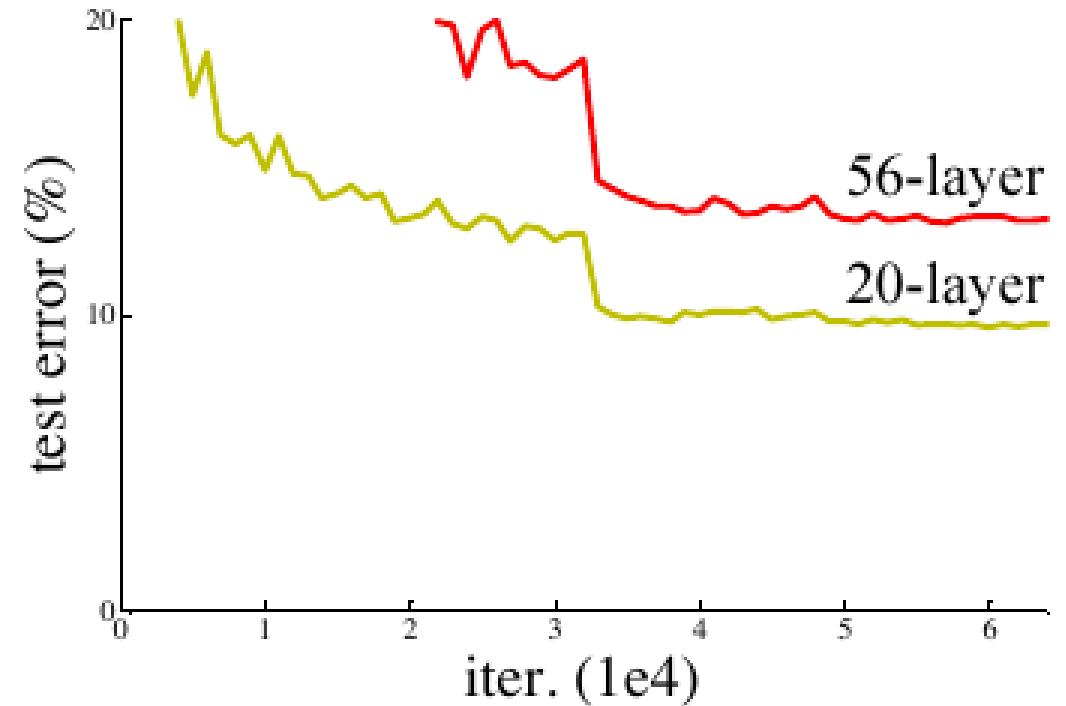
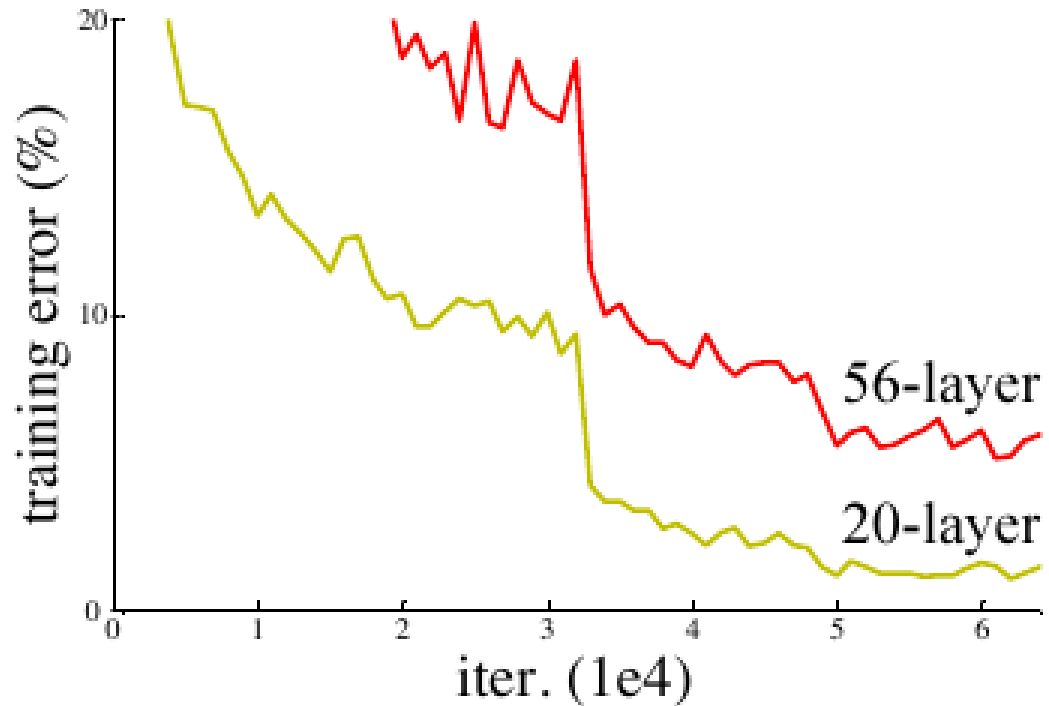


Deeper plain network



More composition can increase representational capacity.

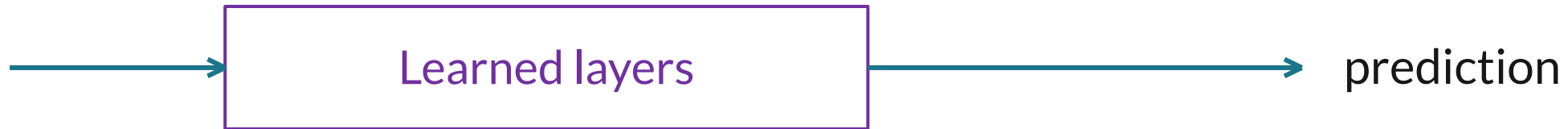
Training deeper plain networks



The deeper plain network has higher training error.

Depth and optimization

Shallower model

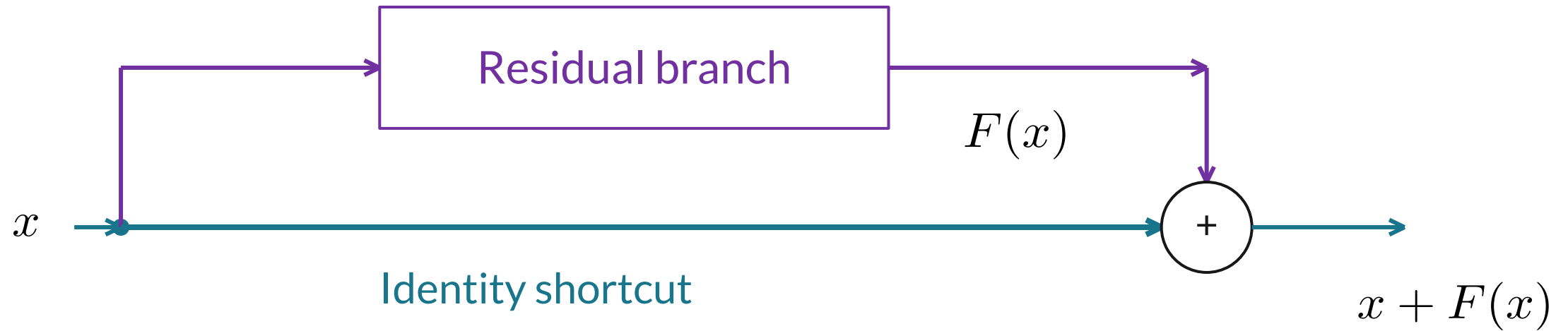


Deeper model



The best attainable training loss cannot increase.
Finding those parameters is an optimization problem.

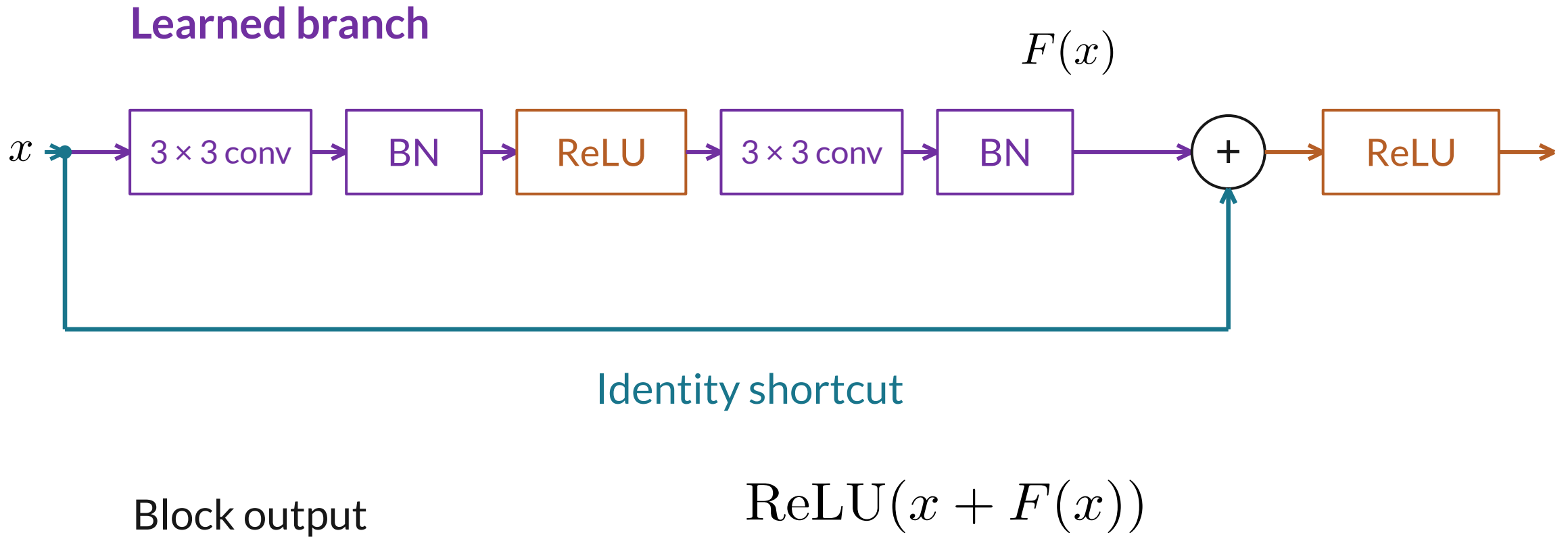
Residual learning



$$F(x) = 0 \implies x + F(x) = x$$

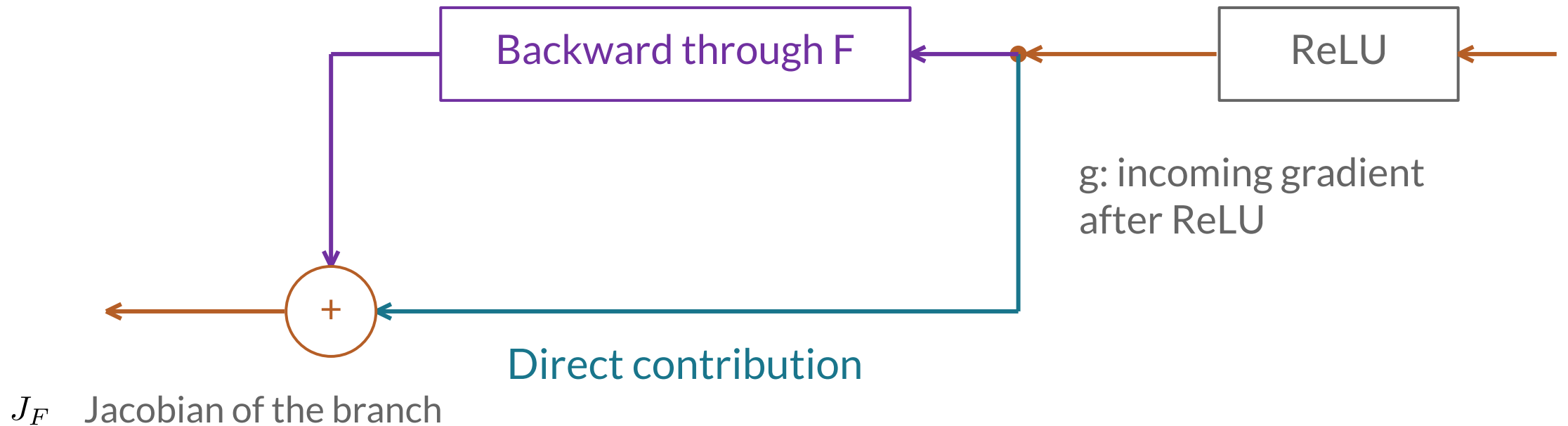
Keep the representation; learn a change when it helps.

ResNet v1 basic block



Gradients through a shortcut

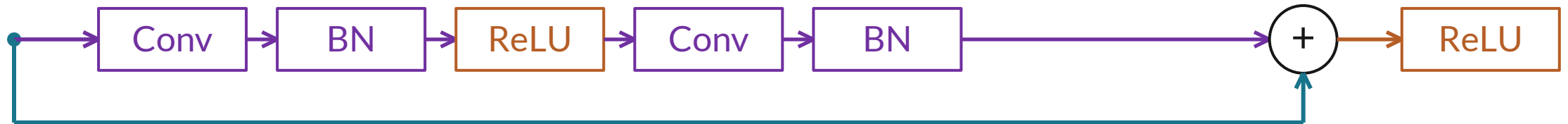
Backward through the residual block



$$\frac{\partial L}{\partial x} = g + J_F(x)^T g$$

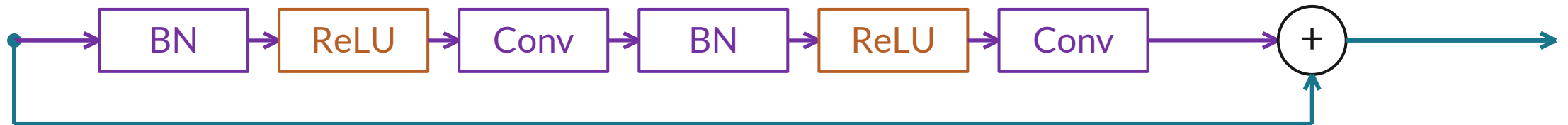
ResNet v1 and v2

ResNet v1 • activation after addition



The final ReLU can clip values and gradients.

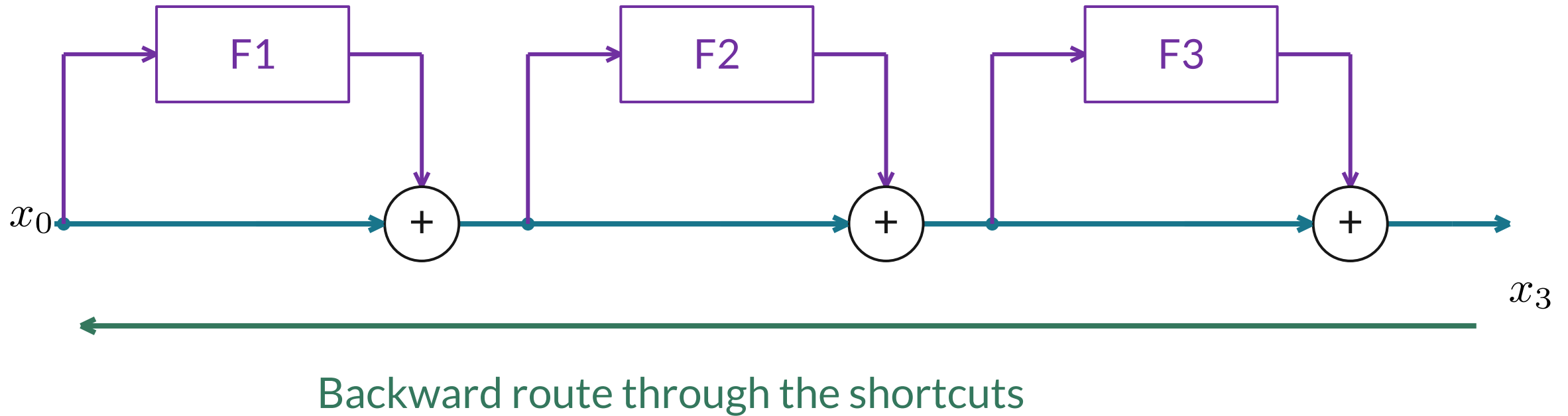
ResNet v2 • pre-activation



Direct route through the addition

Identity shortcuts across depth

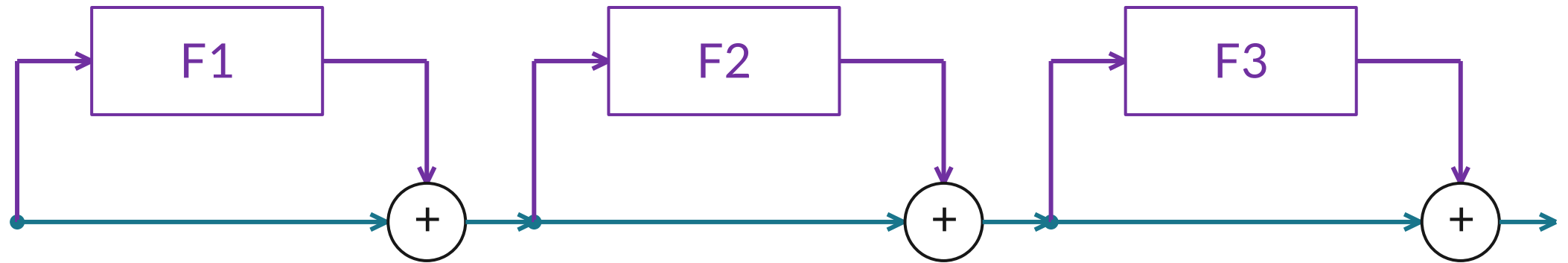
Within a stage: identity shortcuts, no activation after addition



It bypasses the weight layers and activation derivatives.

ResNet as an ensemble

Shortcuts create routes of different lengths.



All shortcuts



One branch



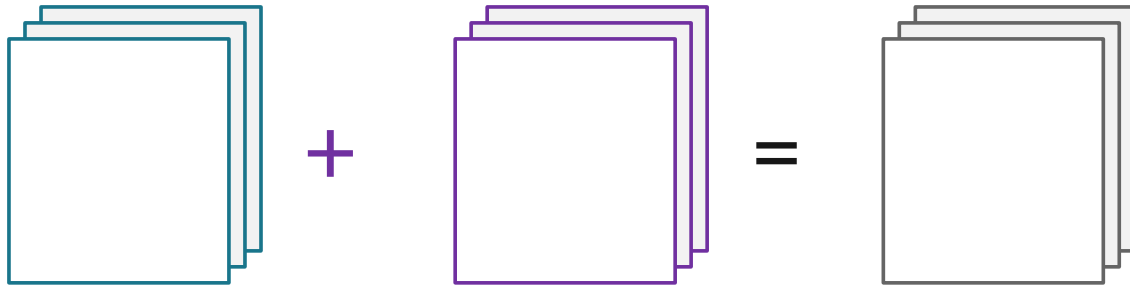
All branches



Shared blocks, trained together

Adding feature tensors

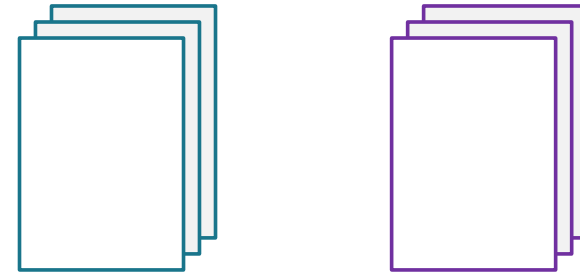
Elementwise addition



$$C \times H \times W$$

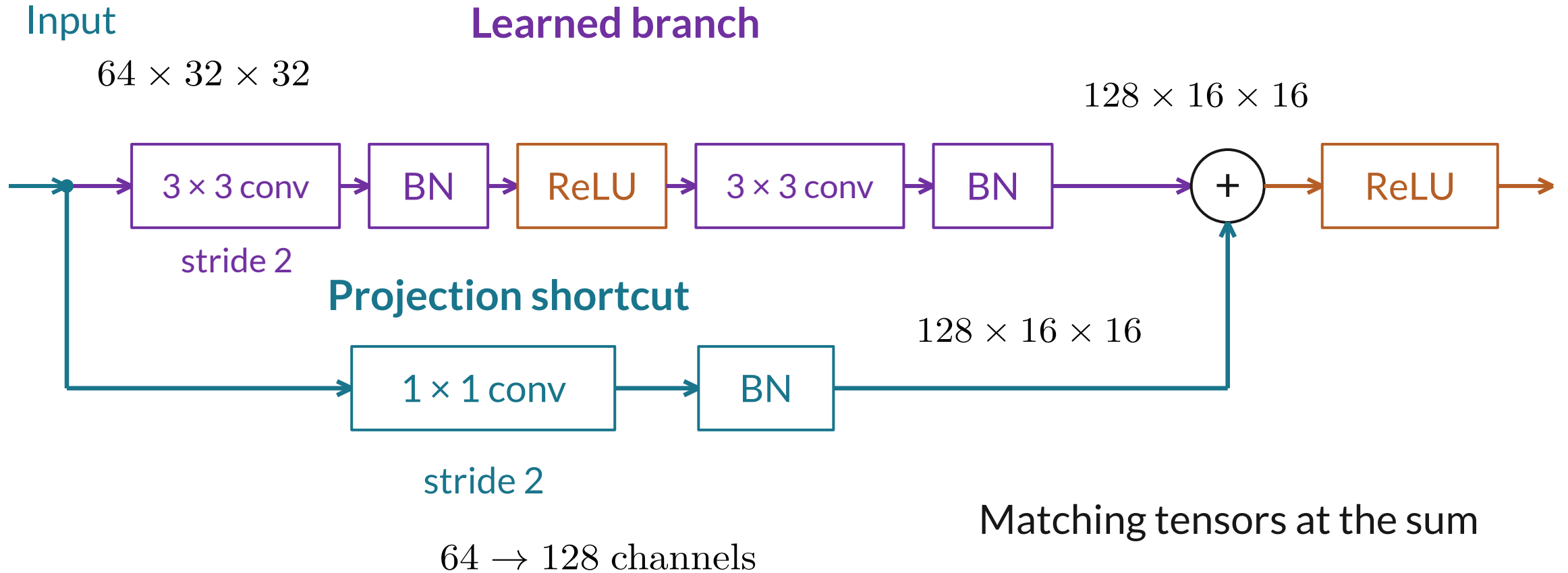
Same shape in and out

Channel concatenation



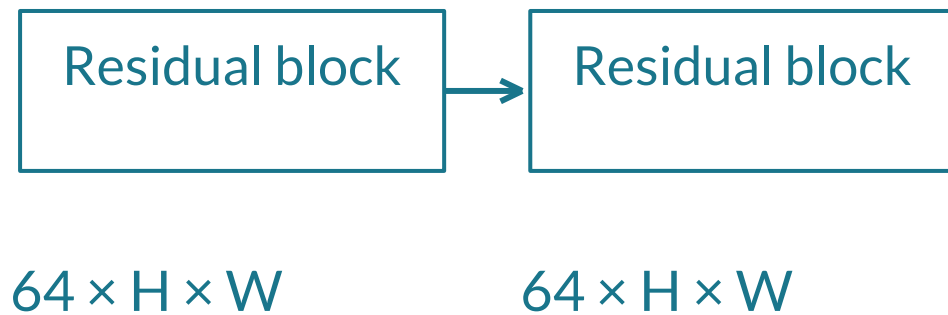
$$2C \times H \times W$$

Projection shortcuts

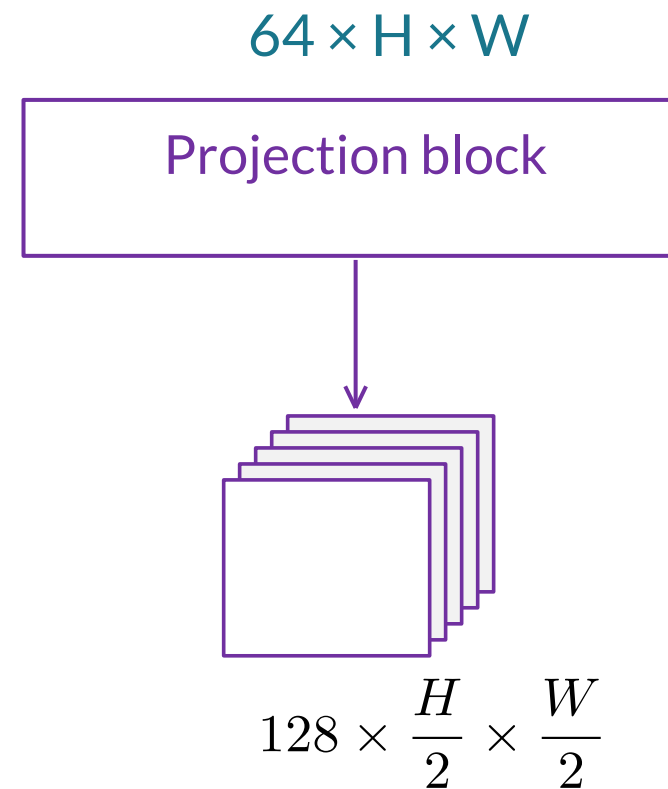


Residual stages

Within a stage

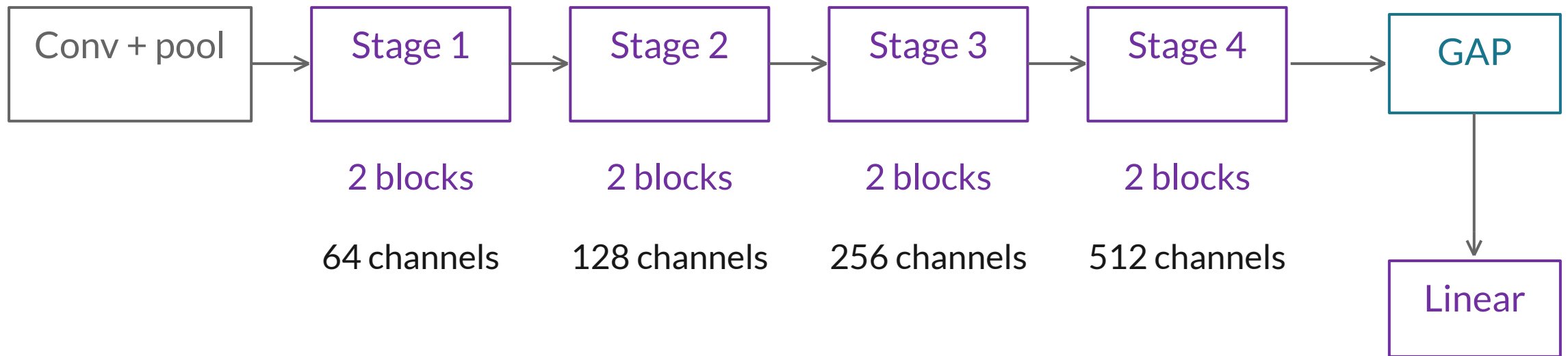


Stage transition



ResNet-18

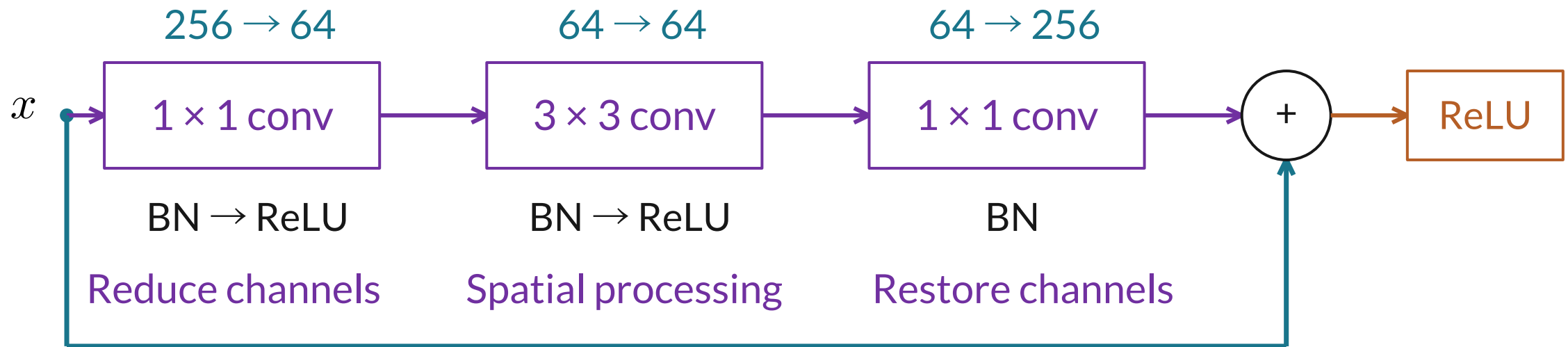
Four residual stages and a classification head



Stage boundaries reduce resolution and increase channels.

Bottleneck residual blocks

ResNet-50, 101 and 152 use three-layer bottleneck blocks.

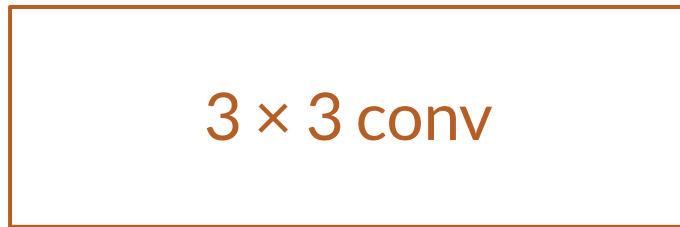


The spatial grid stays the same inside this block.
The wide shortcut preserves the input representation.

The cost of a wide convolution

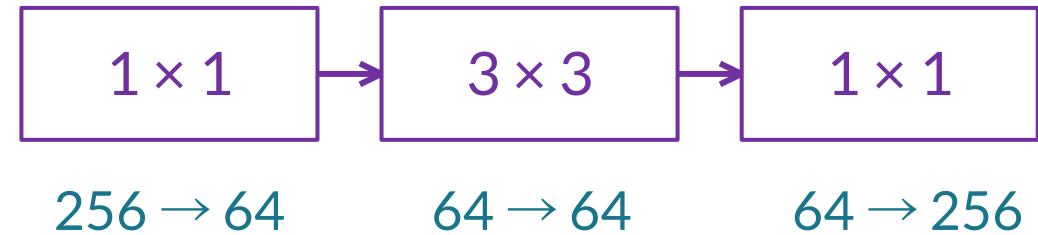
A wide 3×3 convolution is expensive in both channel dimensions.

256 input $\rightarrow$ 256 output channels



$$9 \times 256 \times 256$$

Reduce $\rightarrow$ process $\rightarrow$ restore



$$256 \times 64 + 9 \times 64 \times 64 + 64 \times 256$$

Multiply-adds per output location

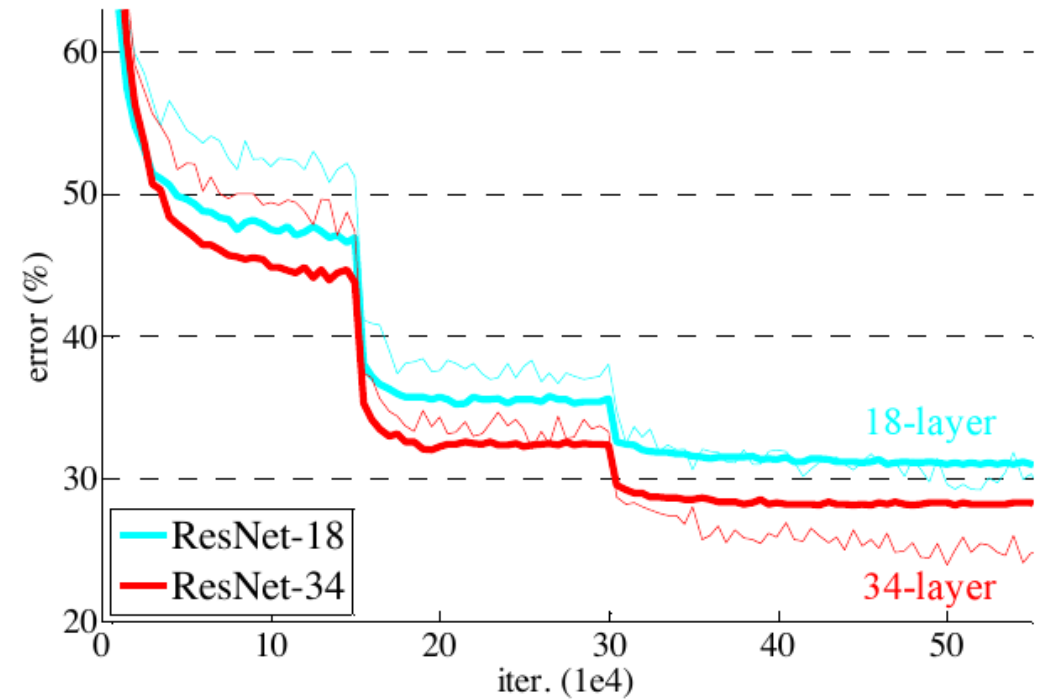
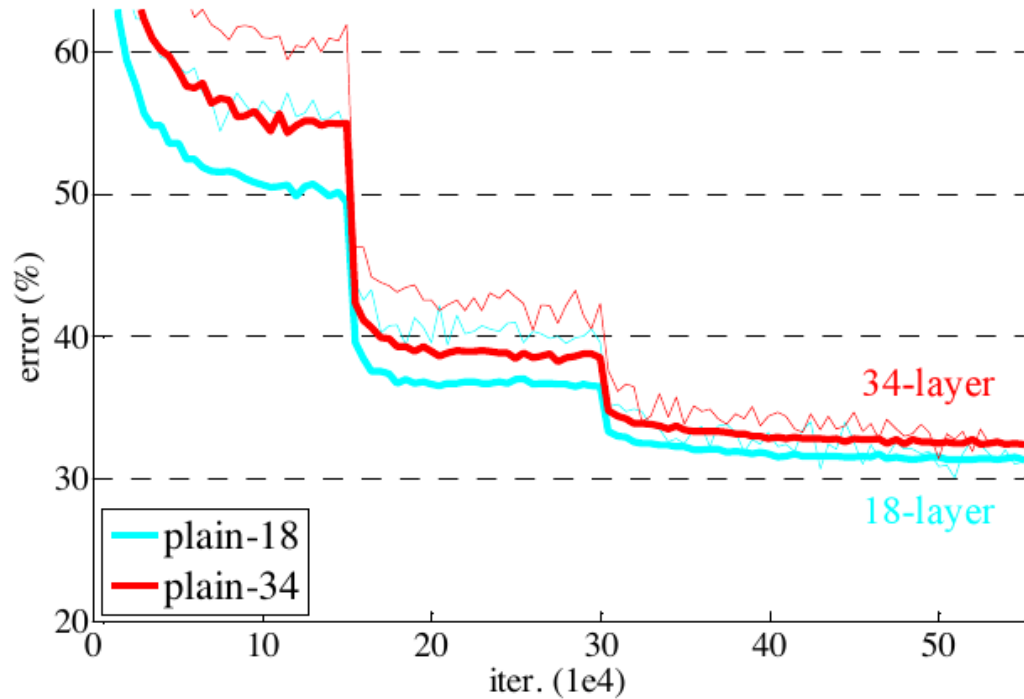
Much less computation at the same spatial resolution.

The channel bottleneck makes much deeper networks affordable.

Residual block in PyTorch

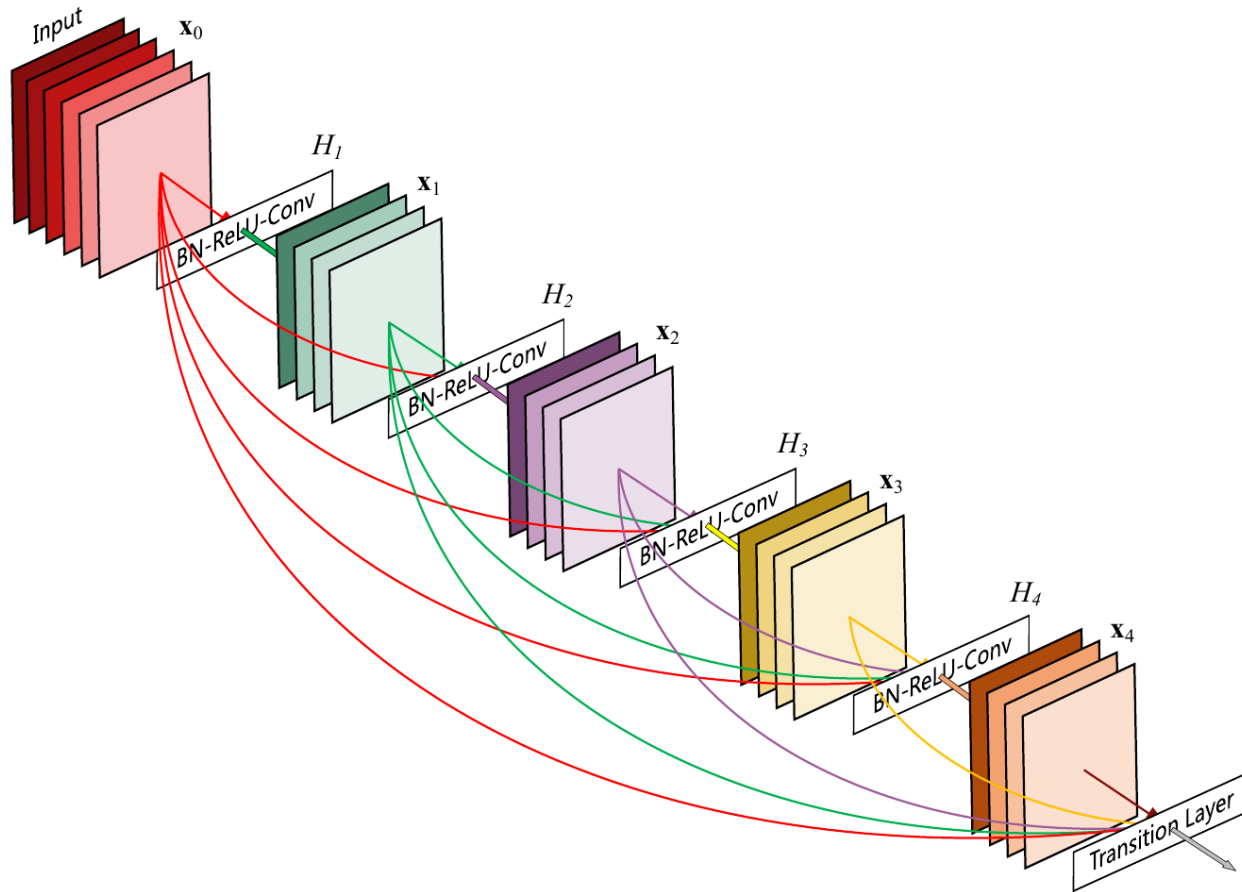
```
def forward(self, x):  
    residual = self.branch(x)          # learned branch  
    shortcut = self.shortcut(x)        # identity or projection  
    summed = shortcut + residual        # elementwise sum  
    return self.activation(summed)     # ReLU after the sum
```

Plain and residual networks



Thin curves: training error Bold curves: validation error

DenseNet



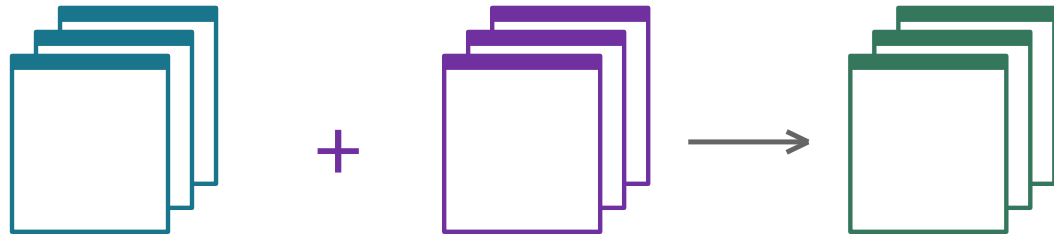
Reuse earlier feature maps

Within a dense block, each layer receives all earlier feature maps.

H: BN $\rightarrow$ ReLU $\rightarrow$ convolution
x: newly produced feature maps

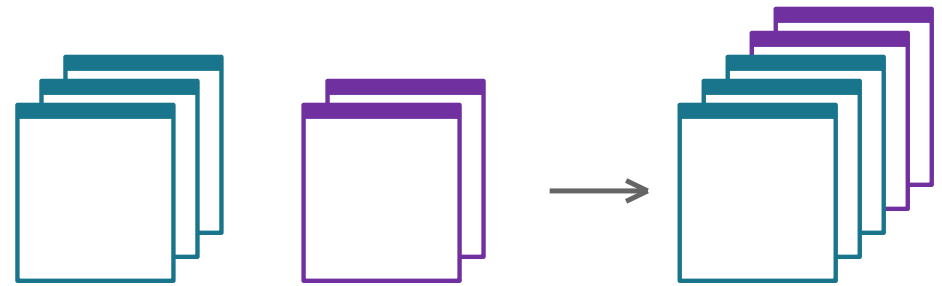
Feature reuse in a dense block

Residual addition



Same channel count

Dense concatenation



Retained channels + new channels

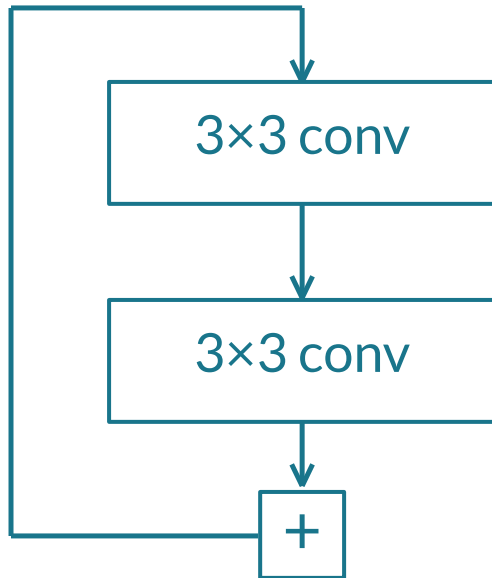
Updates existing feature maps

Direct reuse of earlier maps
More stored activations

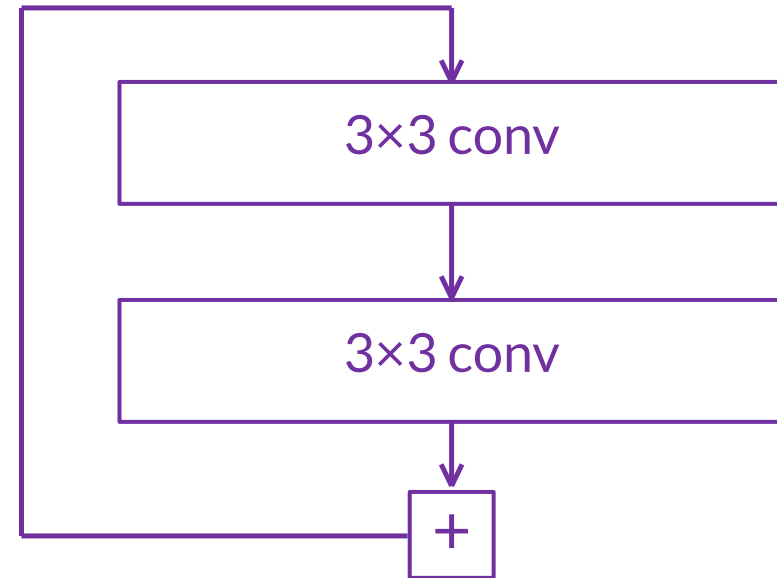
Wide ResNet

Give each block more representational capacity by increasing channels.

Both blocks also use BN and ReLU.



Narrower representation

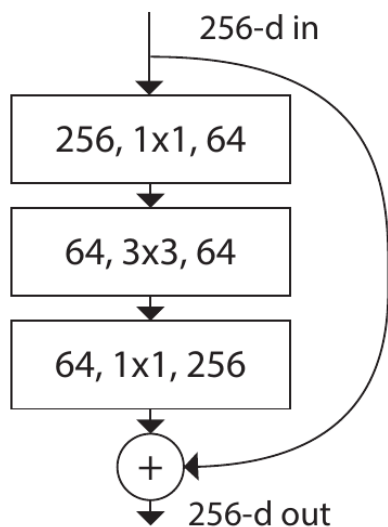


Wider representation
More work per block

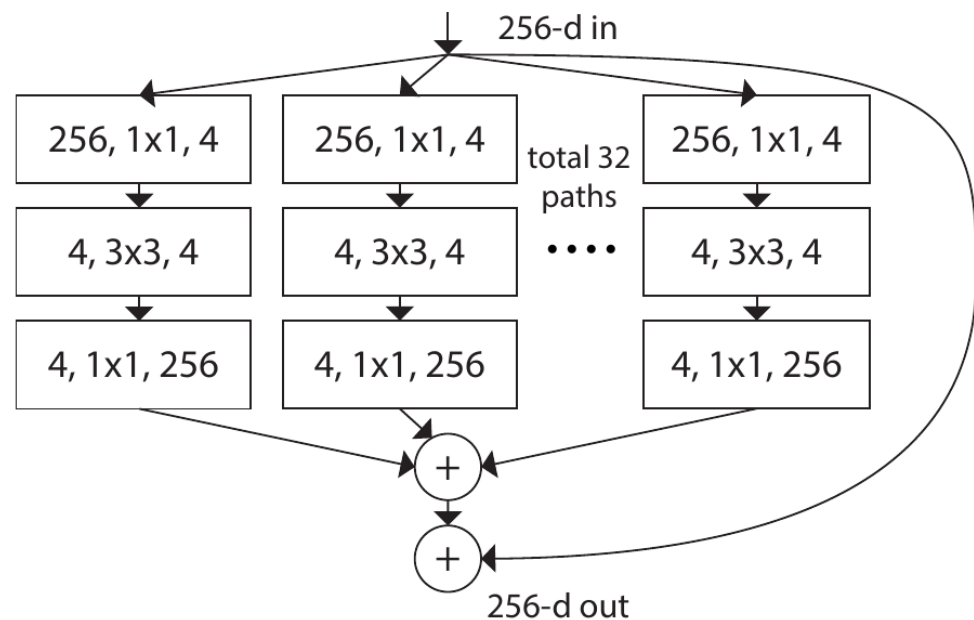
ResNeXt

Increase a block's capacity with parallel, narrow transformations.

ResNet bottleneck



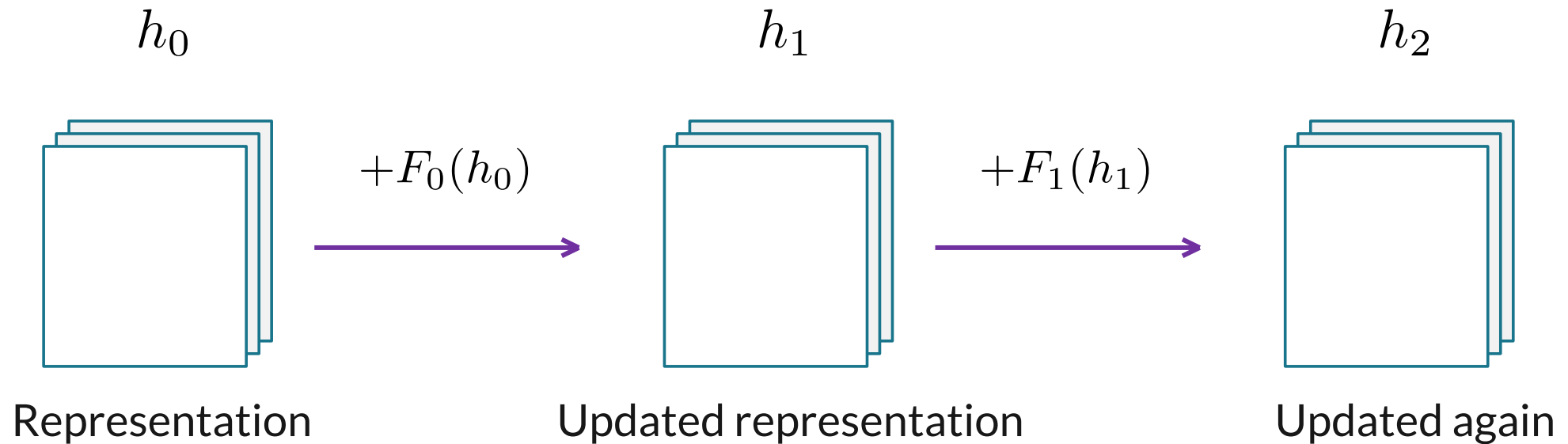
ResNeXt



Same branch structure, different weights; outputs are summed.

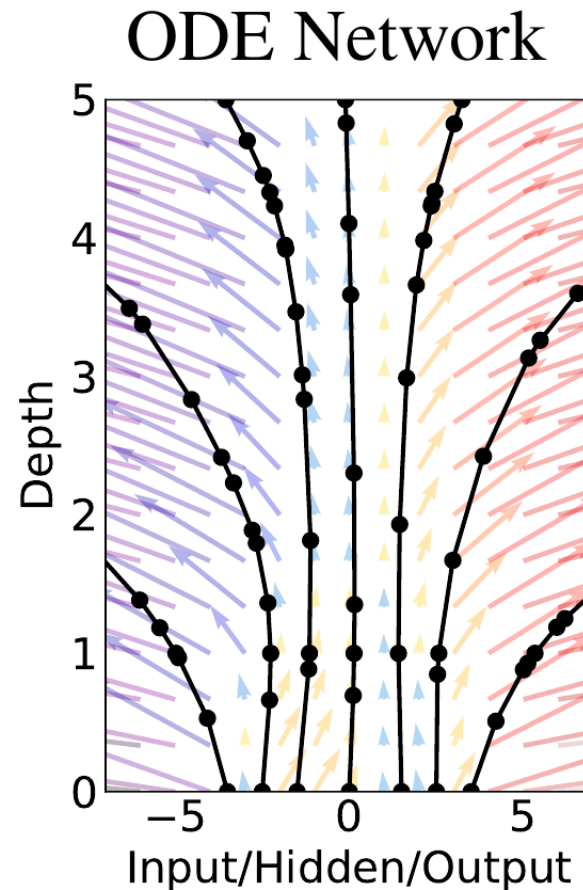
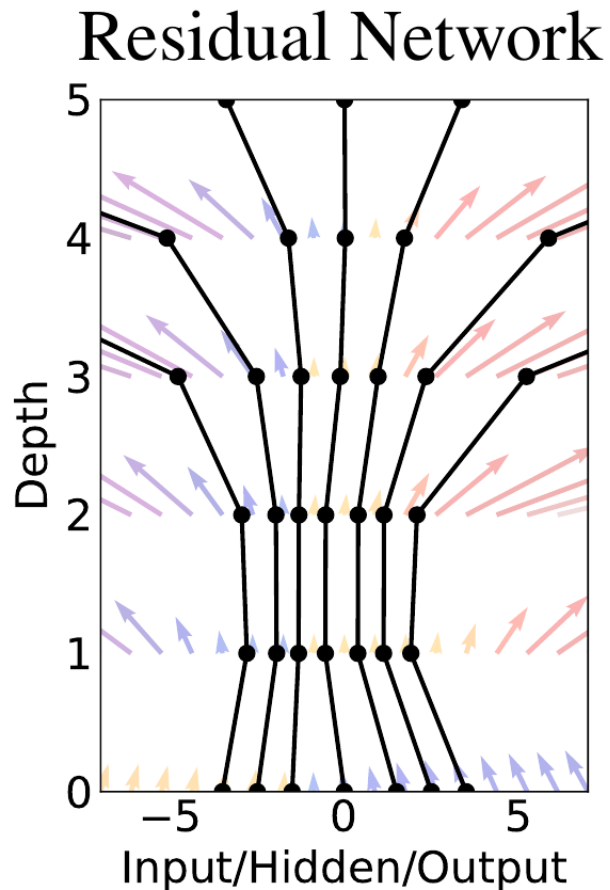
Residual updates

Each block adds a learned change.



$$h_{k+1} = h_k + F_k(h_k)$$

Residual networks and differential equations



Rate of change

$$\frac{dh(t)}{dt} = f(h(t), t)$$

Small step (Euler)

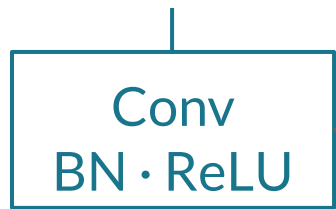
$$h_{k+1} \approx h_k + \Delta t f(h_k, t_k)$$

Δt Step along depth

Here, t measures network depth.

Reading a CNN

Local, shared filters



Residual stage

Composition and
residual sums

Lower resolution,
more channels

Downsample

GAP

Spatial means

Class scores

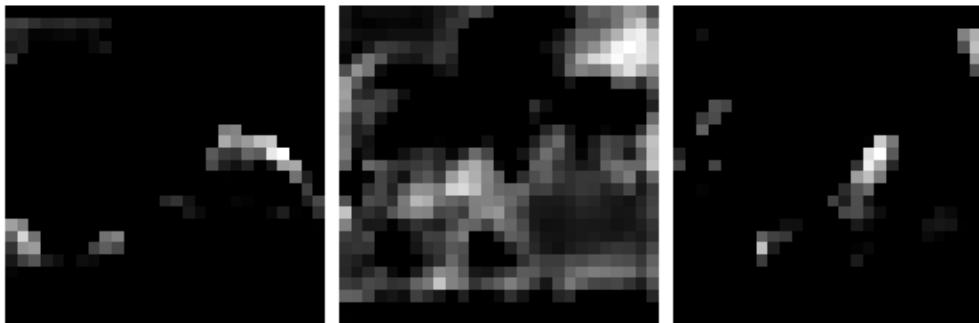
Linear

Next: representations and transfer learning

conv1 (after first conv layer)



conv2 (after second conv layer)



- Visualize learned features
- Adversarial examples
- Reuse and fine-tune pretrained networks