

# Deep Learning

A journey from feature extraction and engineering to end-to-end pipelines

Part 3: Convolutional Neural Networks

Andrei Bursuc

*With slides from A. Karpathy, F. Fleuret, J. Johnson, S. Yeung, G. Louppe, Y. Avrithis ...*

# Convolutional layers

# Why would we need them?

If they were handled as normal "unstructured" vectors, large-dimension signals such as sound samples or images would require models of intractable size.

For instance a linear layer taking a  $256 \times 256$  RGB image as input, and producing an image of same size would require:

$$(256 \times 256 \times 3)^2 \simeq 3.87e + 10$$

parameters, with the corresponding memory footprint ( $\simeq 150\text{Gb}$  !), and excess of capacity.

# Why would we need them?

Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". A representation meaningful at a certain location can / should be used everywhere.



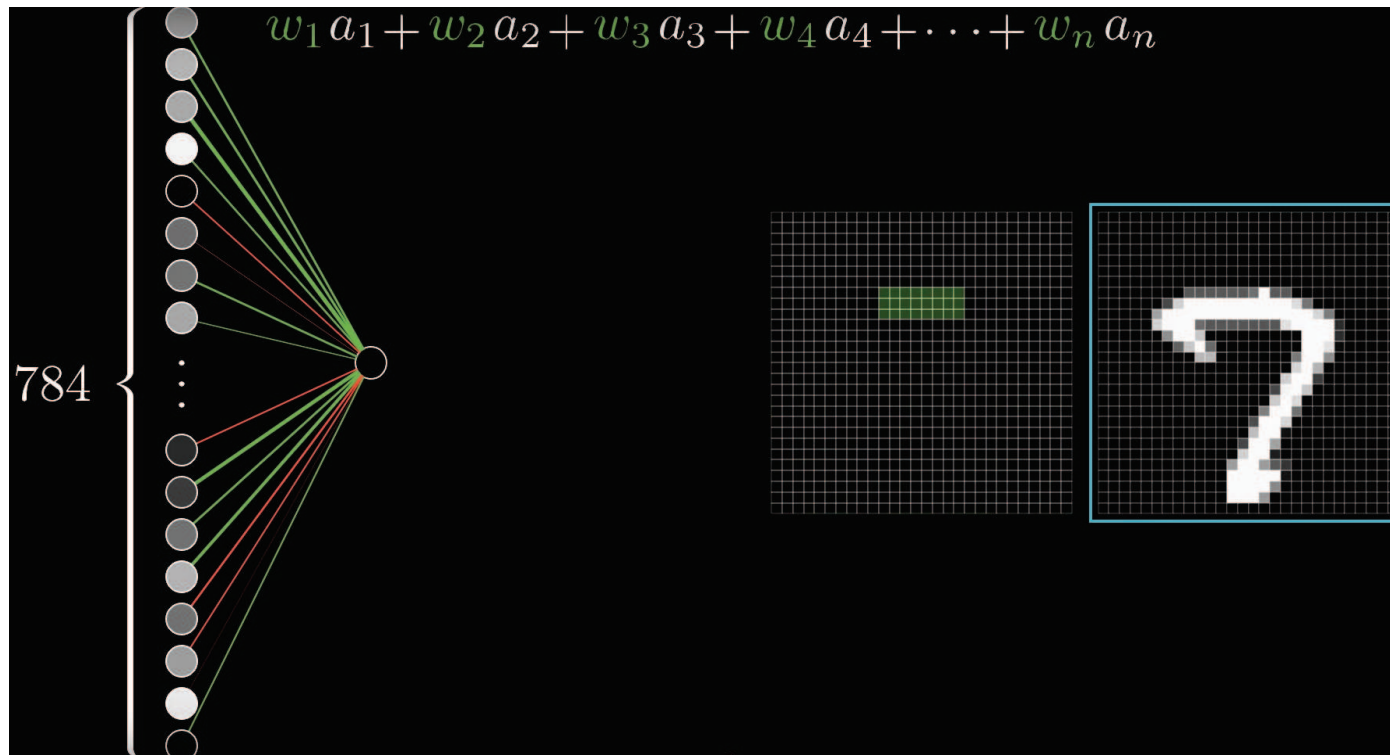
# Why would we need them?

Moreover, this requirement is inconsistent with the intuition that such large signals have some "invariance in translation". A representation meaningful at a certain location can / should be used everywhere.

A convolutional layer embodies this idea. It applies the same linear transformation locally, everywhere, and preserves the signal structure.

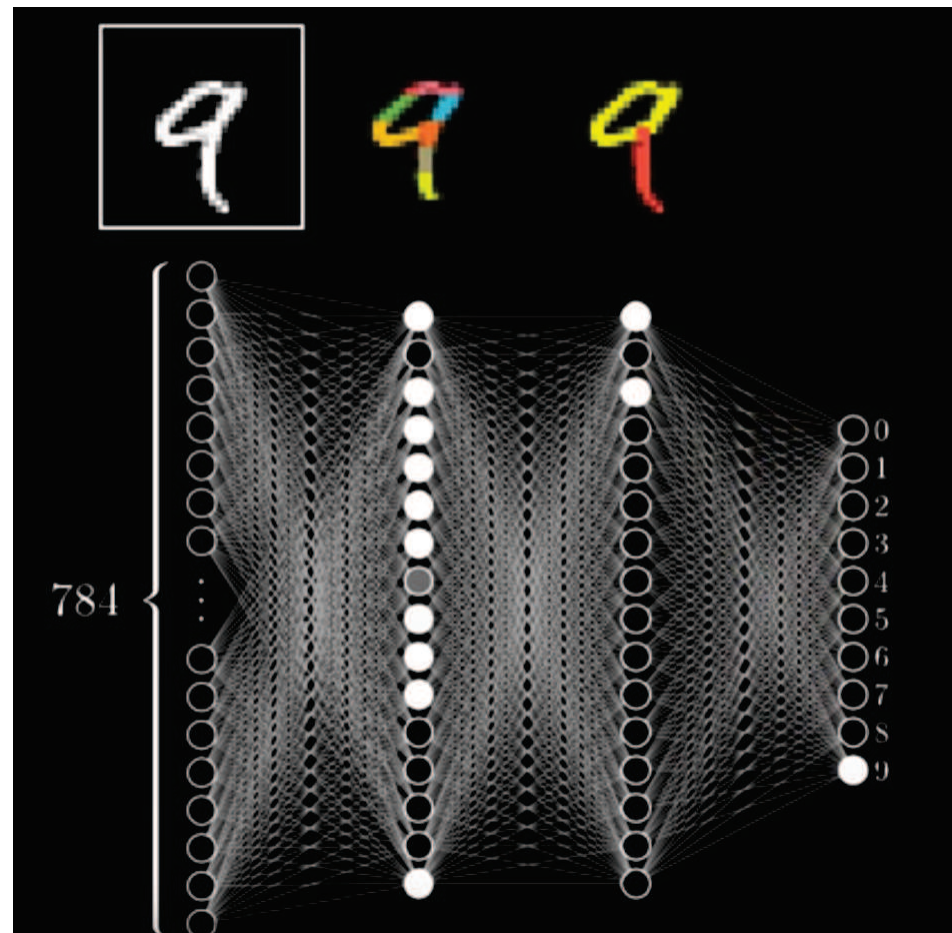
# Why would we need them?

- One neuron gets specialized for detecting a full-image pattern, while being sensible to translations

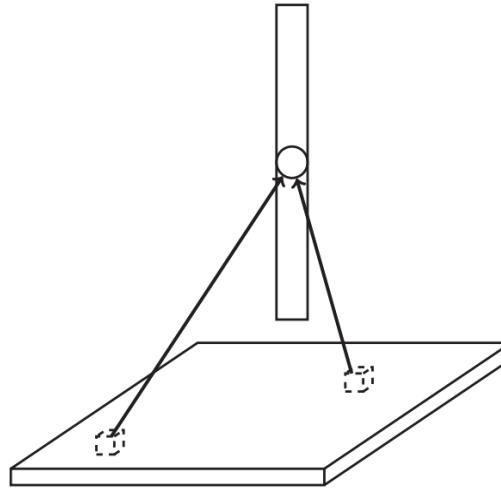


# Why would we need them?

- Each neuron gets specialized for detecting a full-image pattern.
- Neurons from later layer work similarly
- This is a big waste of parameters without good performance.



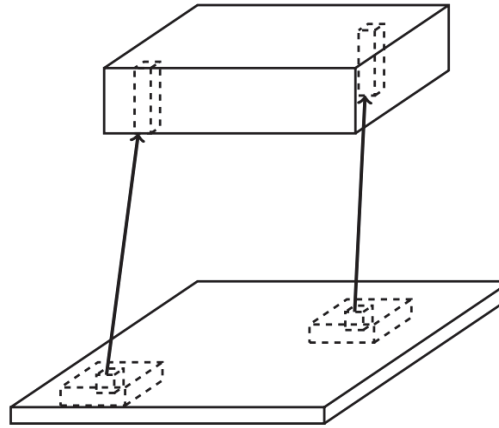
# Fully connected layer



In a **fully connected layer**, each hidden unit  $h_j = \sigma(\mathbf{w}_j^T \mathbf{x} + b_j)$  is connected to the entire image.

- Looking for activations that depend on pixels that are spatially far away is supposedly a waste of time and resources.
- Long range correlations can be dealt with in the higher layers.

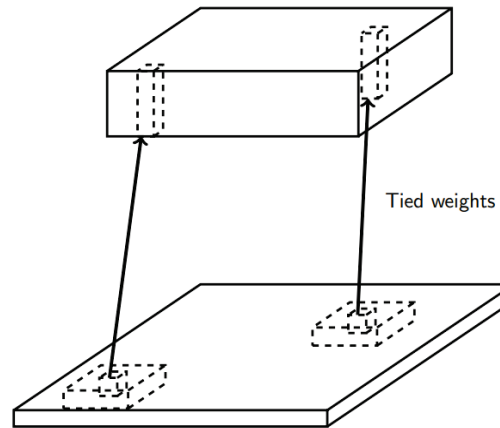
# Locally connected layer



In a **locally connected layer**, each hidden unit  $h_j$  is connected to only a patch of the image.

- Weights are specialized locally and functionally.
- Reduce the number of parameters.
- What if the object in the image shifts a little?

# Convolutional layer



In a **convolutional layer**, each hidden unit  $h_j$  is connected to only a patch of the image, and **share** its weights with the other units  $h_i$ .

- Weights are specialized functionally, regardless of spatial location.
- Reduce the number of parameters.

# Convolution

Discrete convolution (actually cross-correlation) between two functions  $f$  and  $g$ :

$$(f \star g)(x) = \sum_{a+b=x} f(a) \cdot g(b) = \sum_a f(a) \cdot g(x + a)$$

# Convolution

Discrete convolution (actually cross-correlation) between two functions  $f$  and  $g$ :

$$(f \star g)(x) = \sum_{a+b=x} f(a) \cdot g(b) = \sum_a f(a) \cdot g(x+a)$$

In computer vision, we typically use 2D-convolutions (actually 2D cross-correlation):

$$(f \star g)(x, y) = \sum_n \sum_m f(n, m) \cdot g(x+n, y+m)$$



# Convolution

Discrete convolution (actually cross-correlation) between two functions  $f$  and  $g$ :

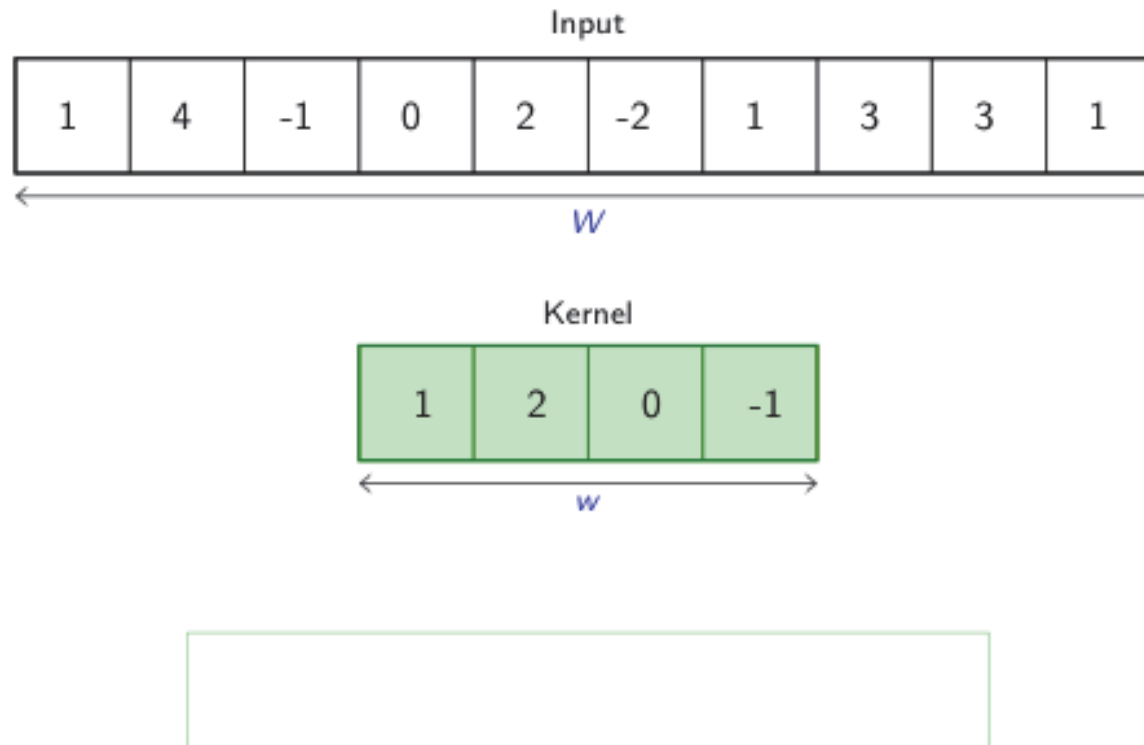
$$(f \star g)(x) = \sum_{a+b=x} f(a) \cdot g(b) = \sum_a f(a) \cdot g(x + a)$$

In computer vision, we typically use 2D-convolutions (actually 2D cross-correlation):

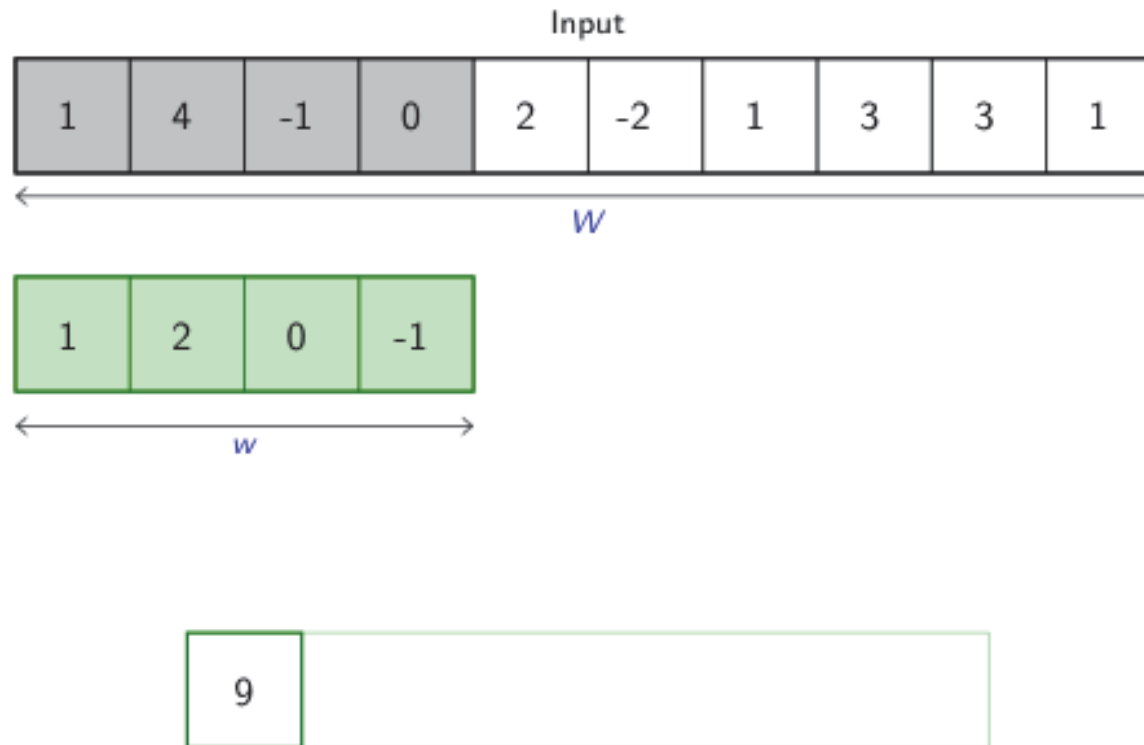
$$(f \star g)(x, y) = \sum_n \sum_m f(n, m) \cdot g(x + n, y + m)$$

$f$  is a convolution **kernel** applied to the 2-d map  $g$  (think image)

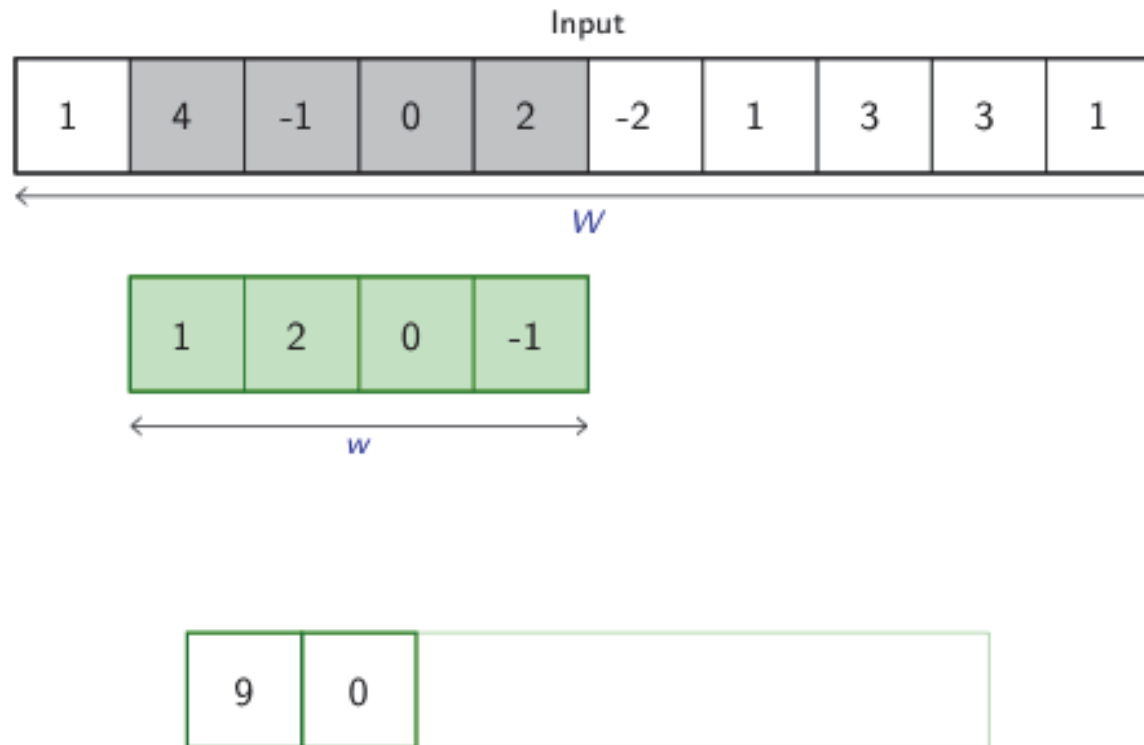
# Convolution 1d



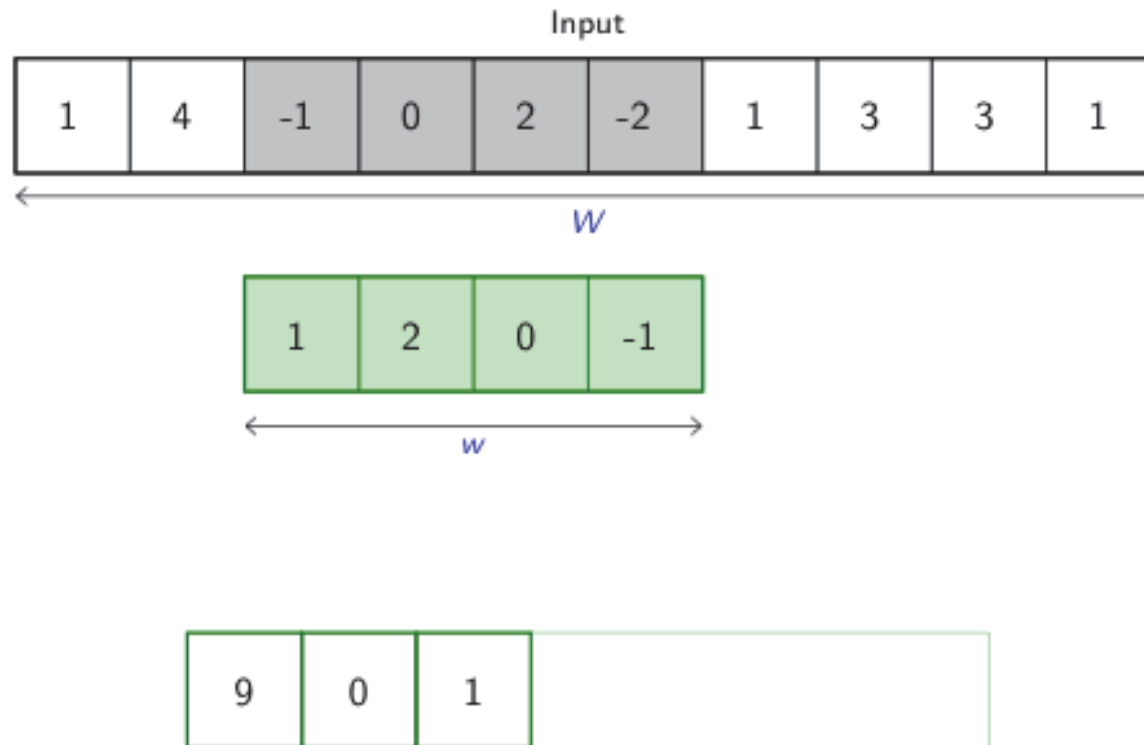
# Convolution 1d



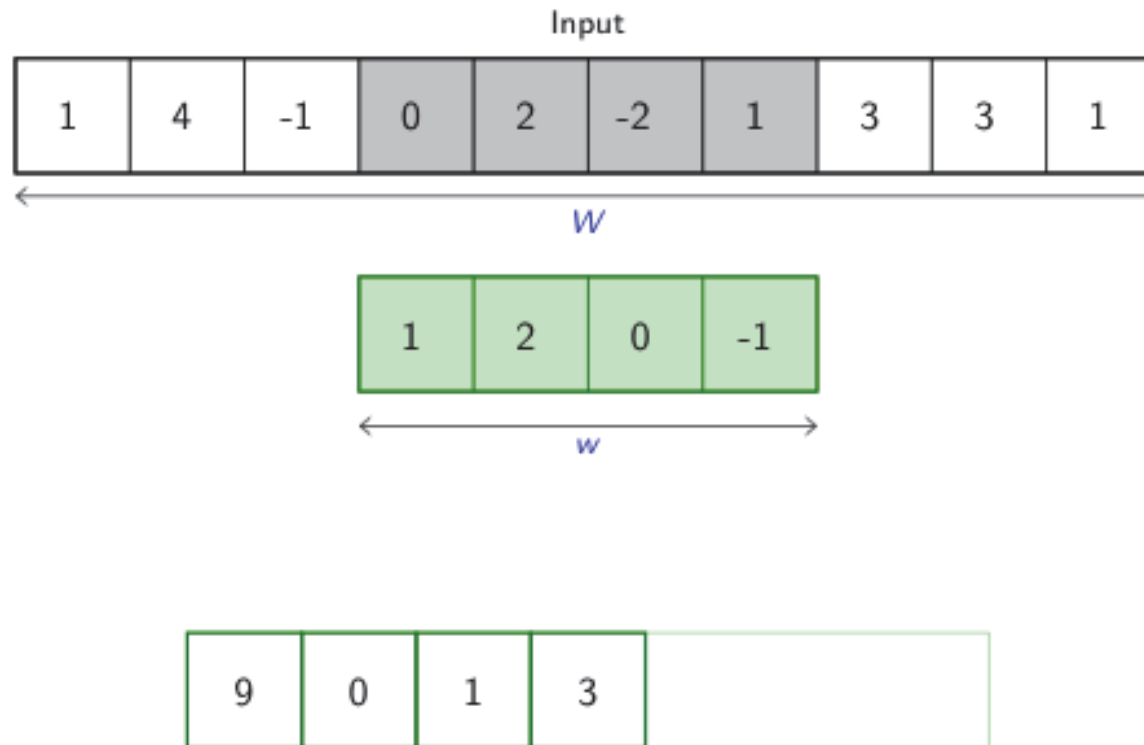
# Convolution 1d



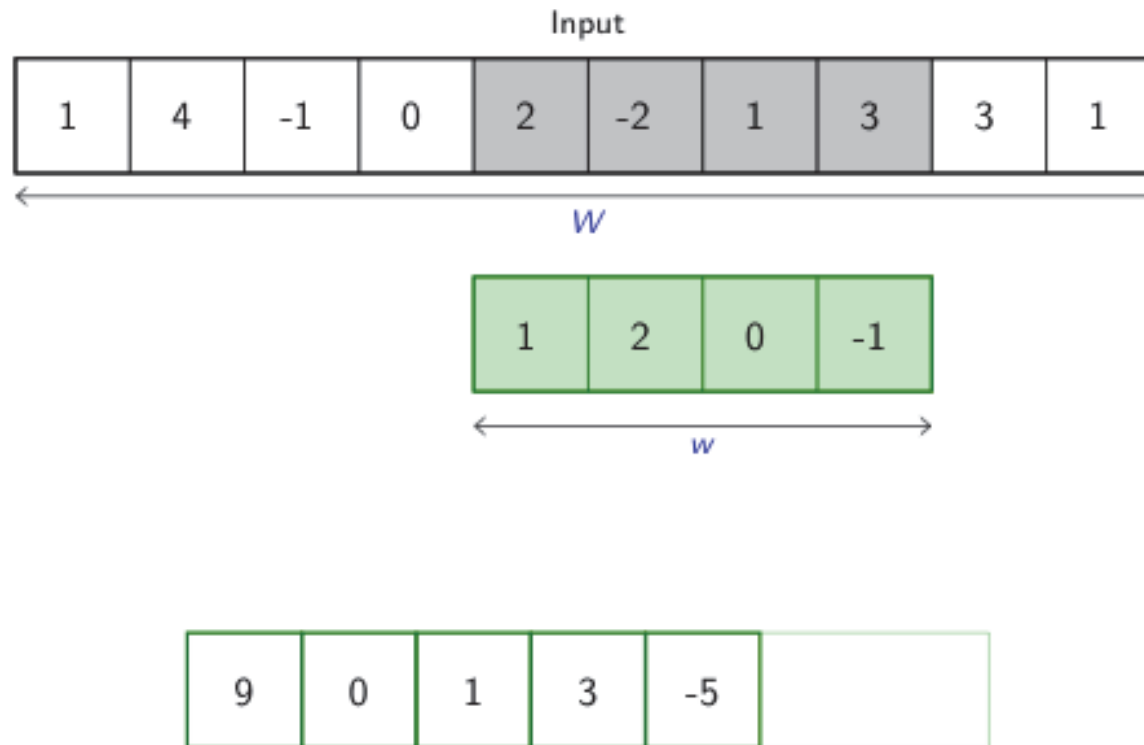
# Convolution 1d



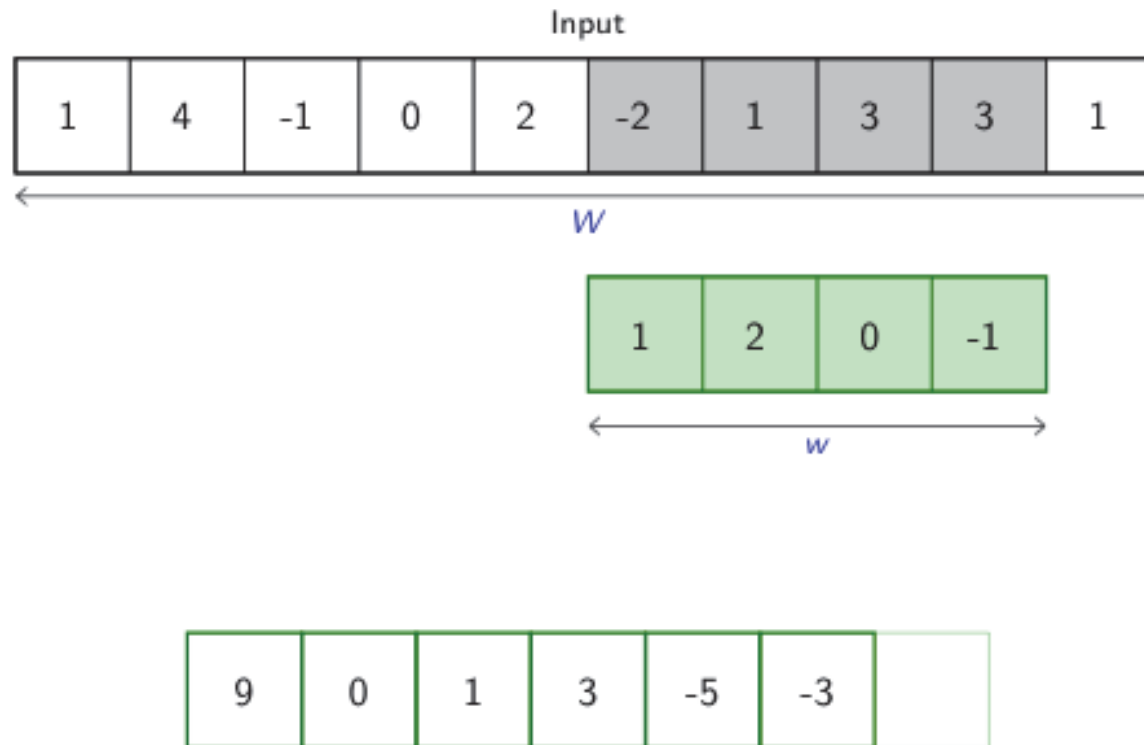
# Convolution 1d



# Convolution 1d

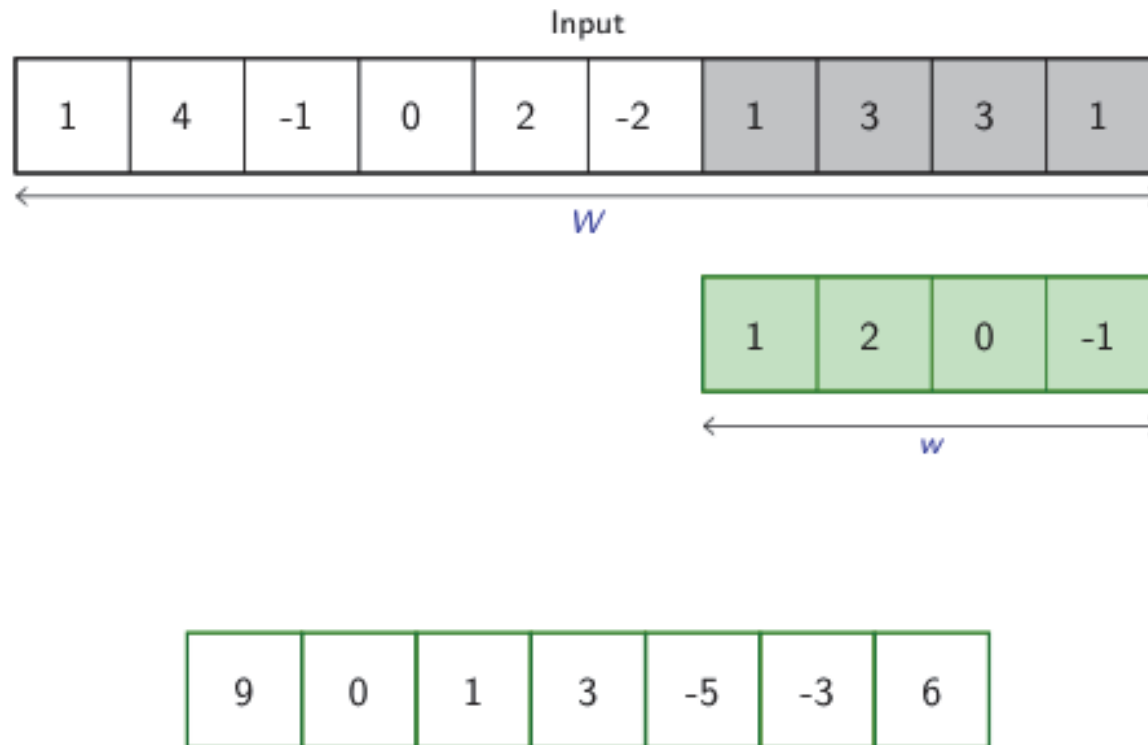


# Convolution 1d

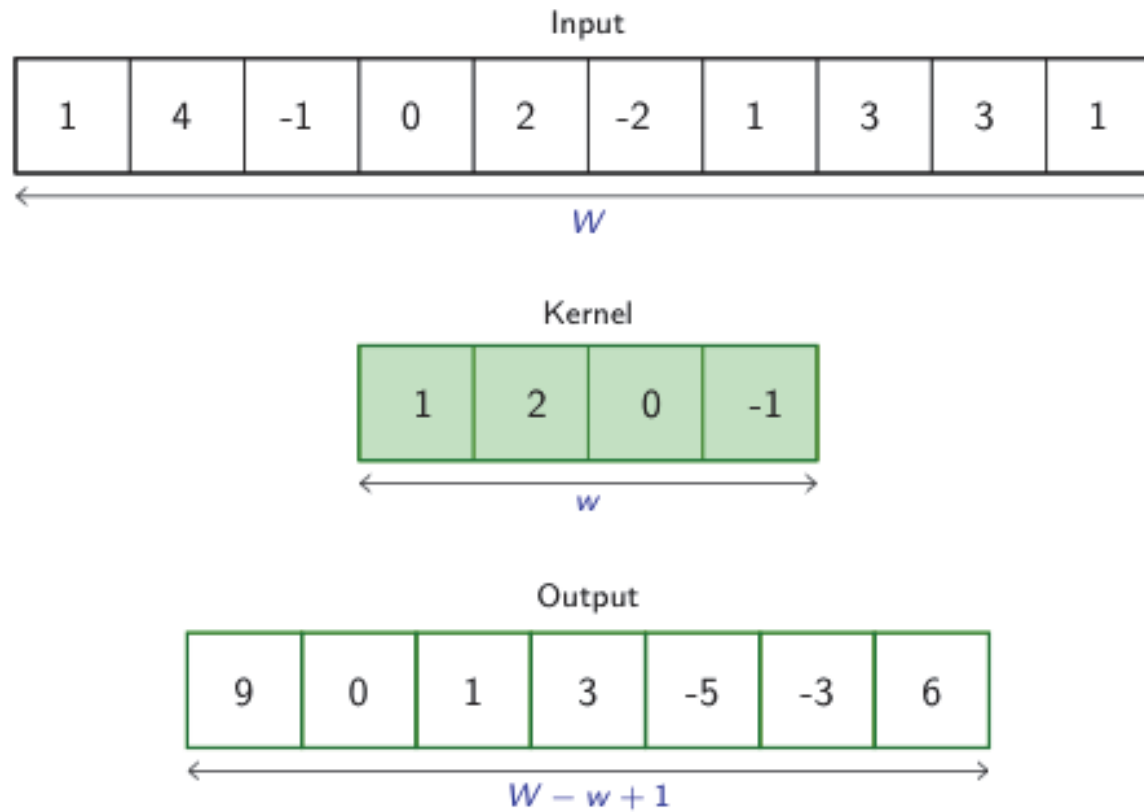




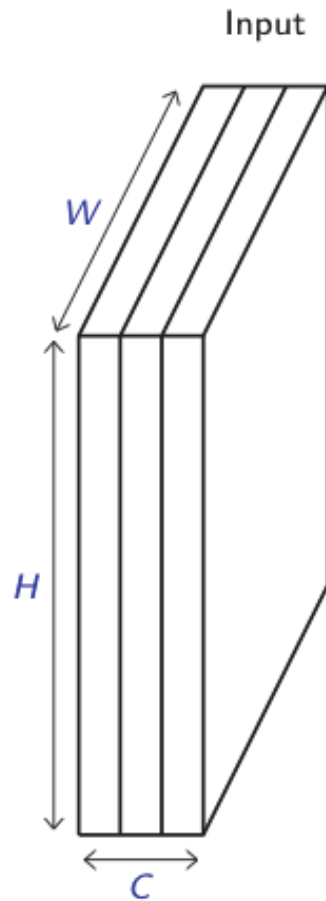
# Convolution 1d



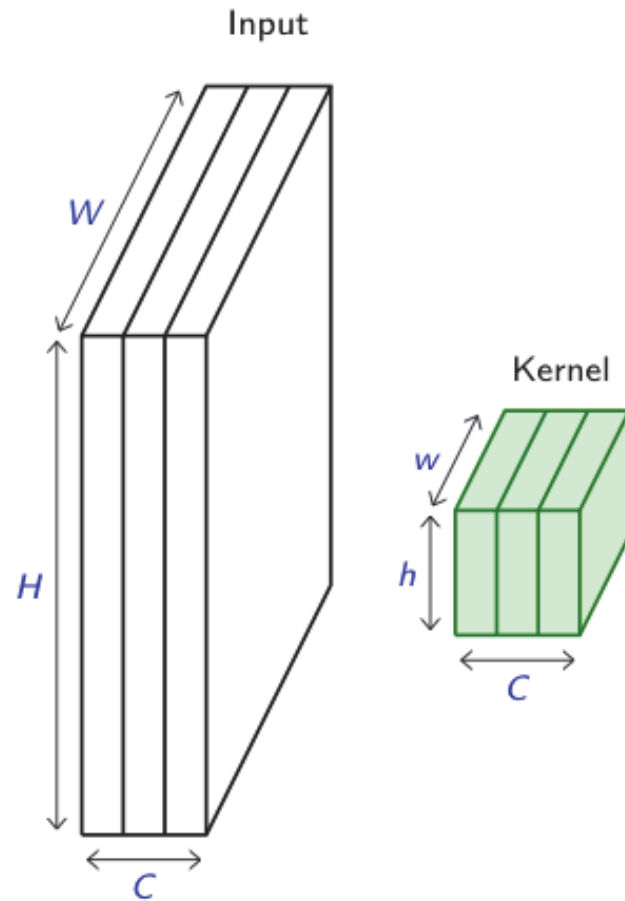
# Convolution 1d



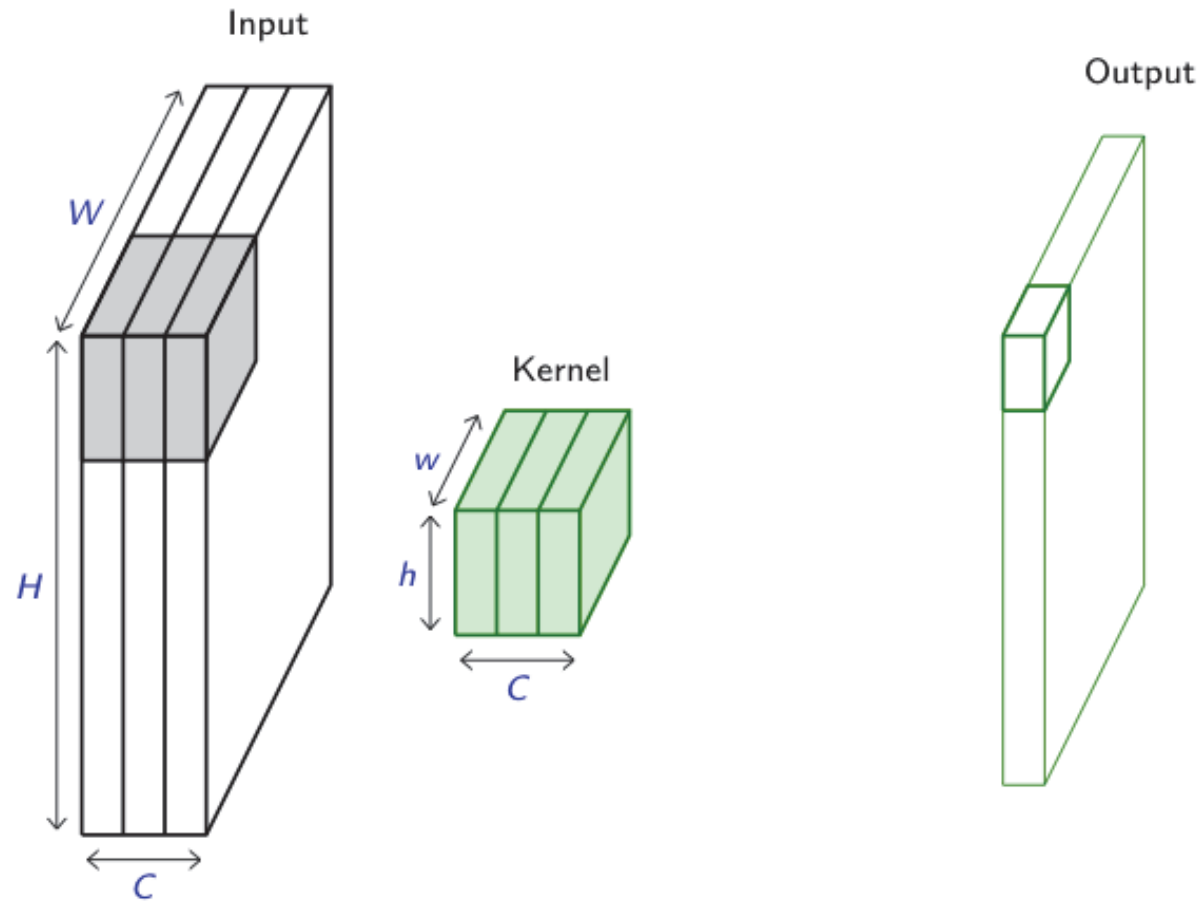
# Convolution 2d



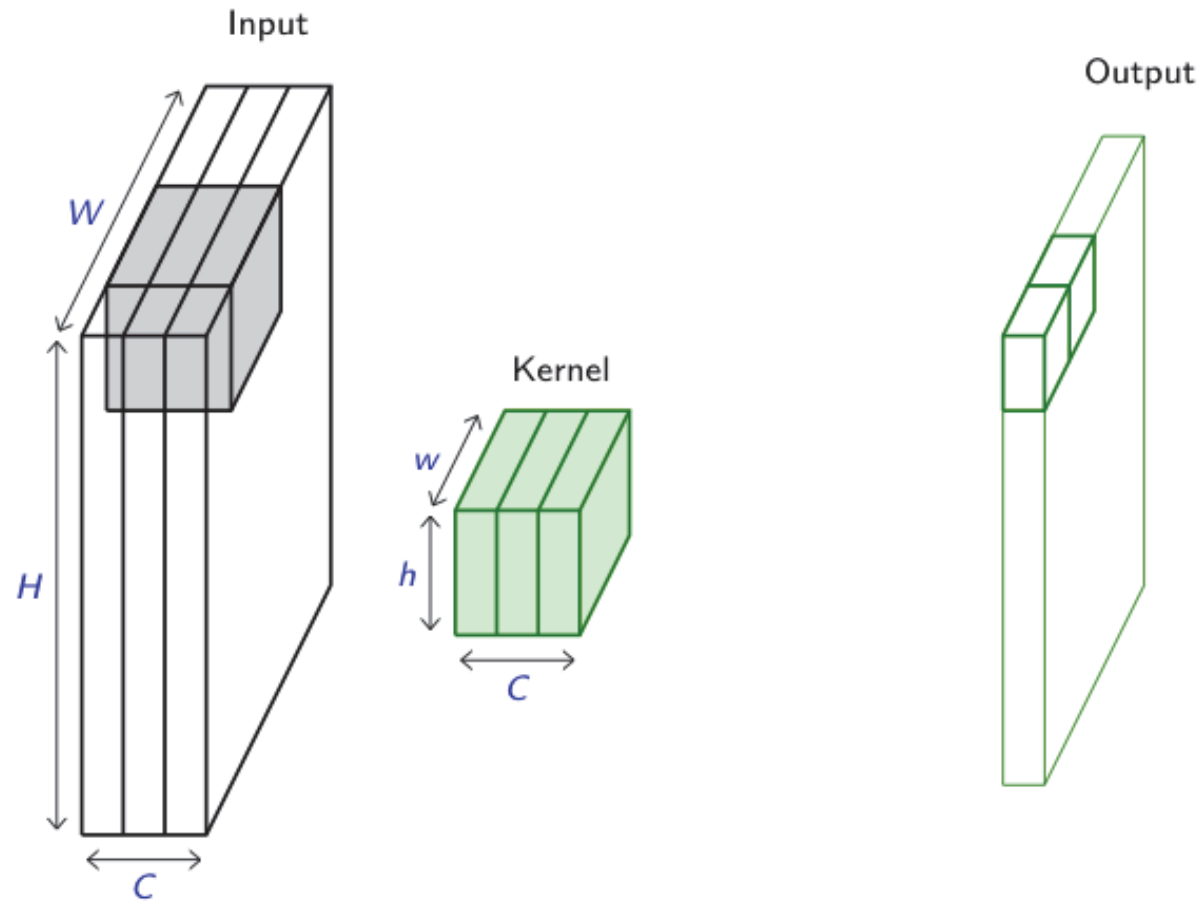
# Convolution 2d



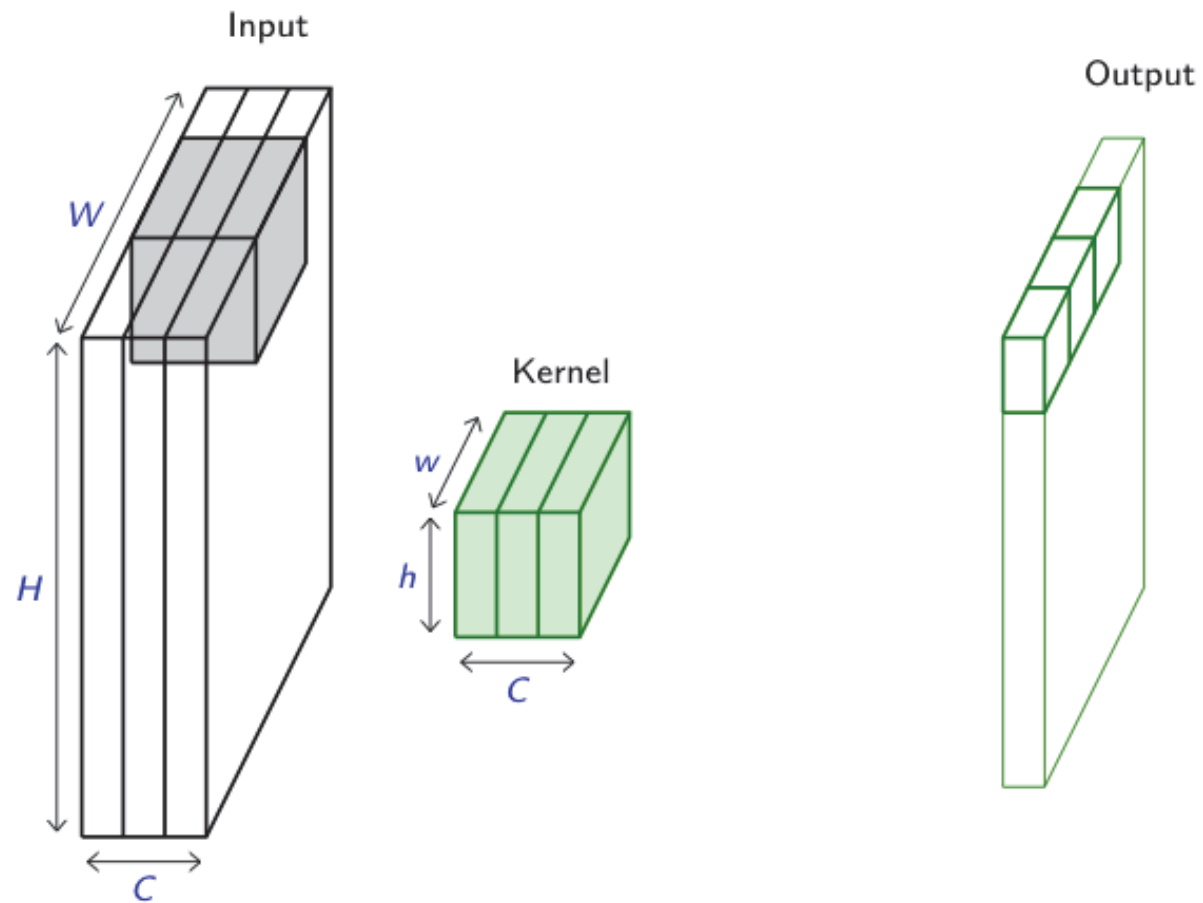
# Convolution 2d



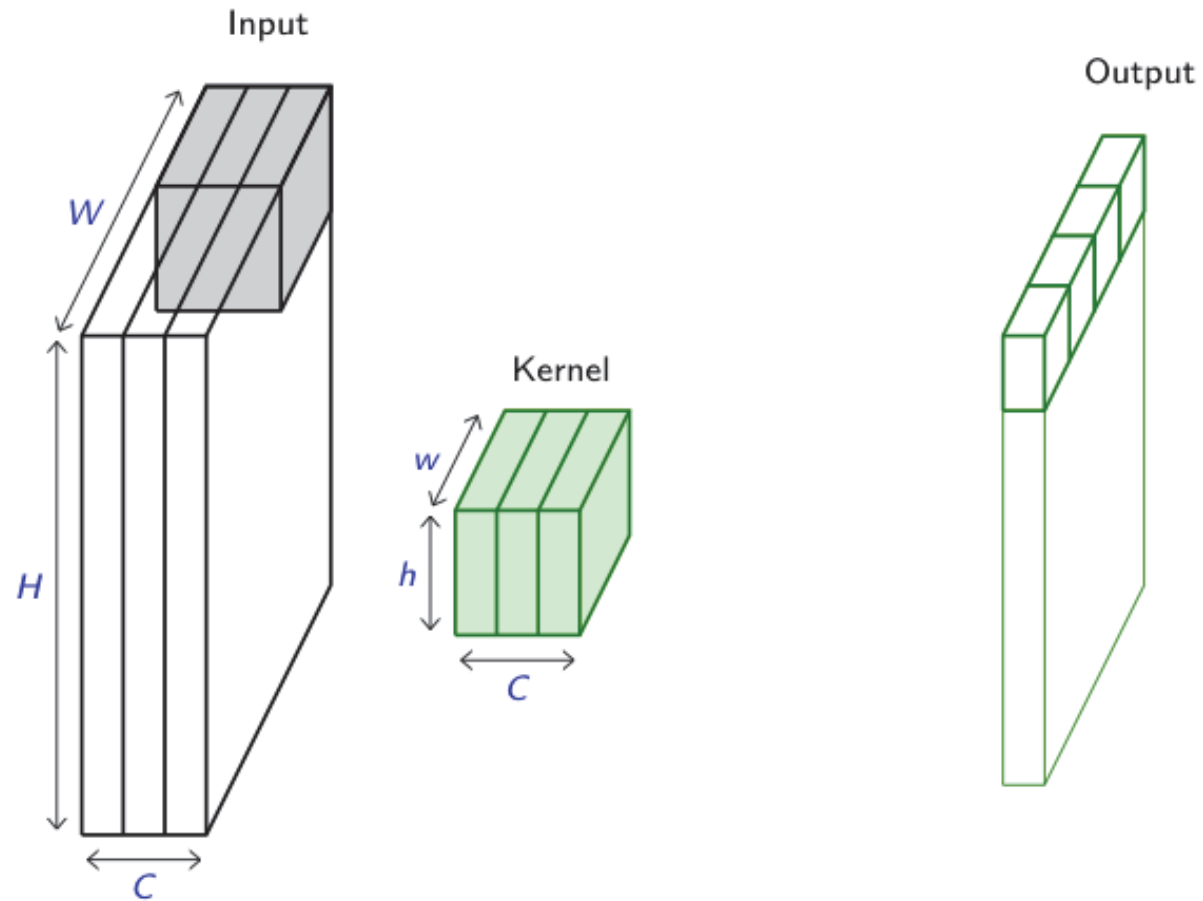
# Convolution 2d



# Convolution 2d

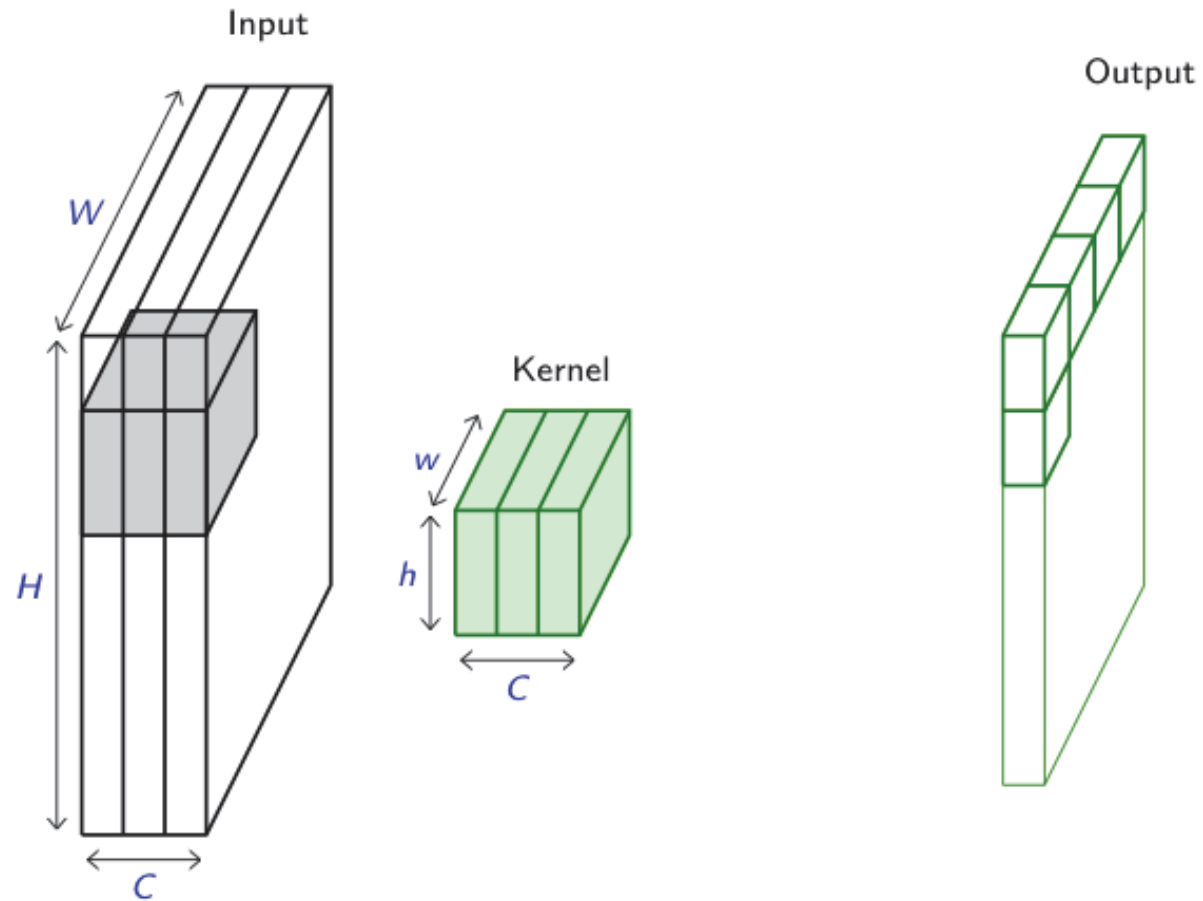


# Convolution 2d

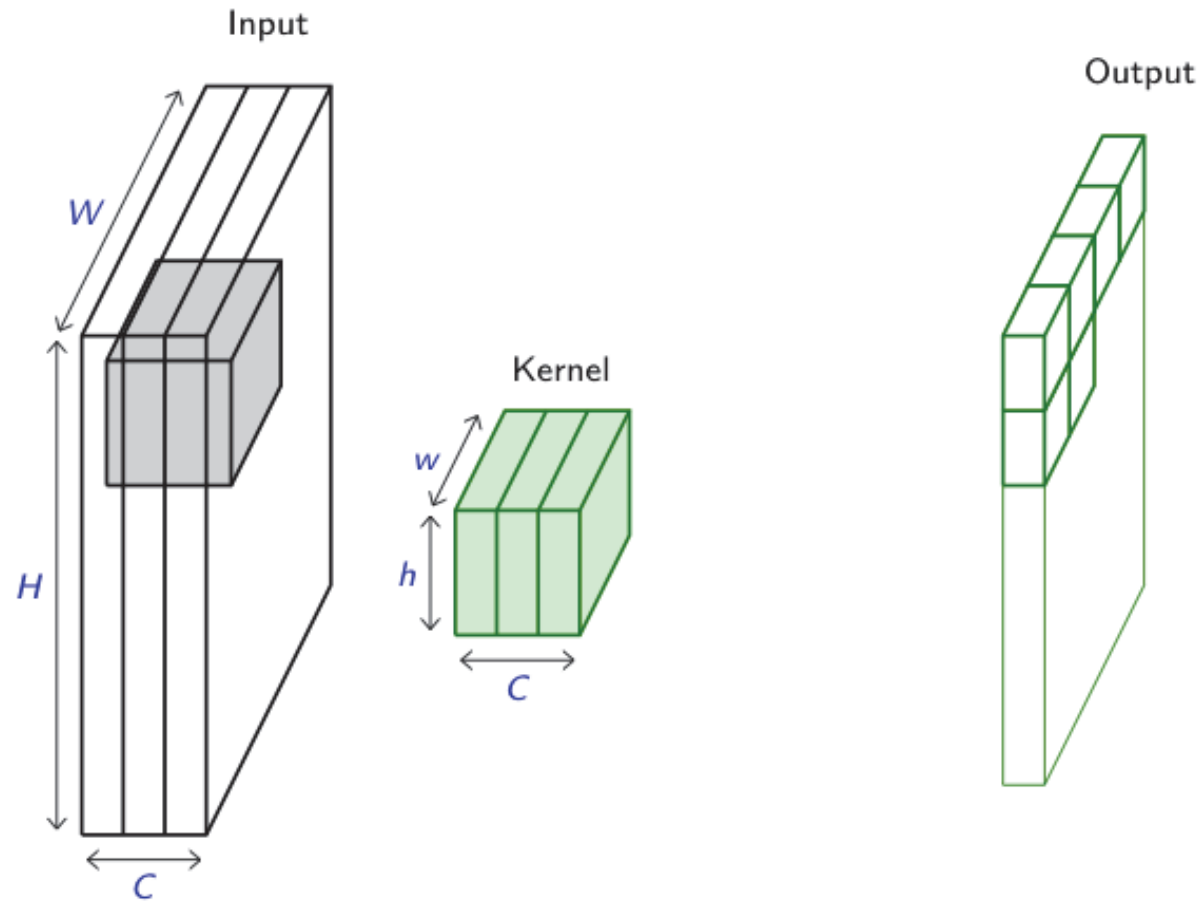




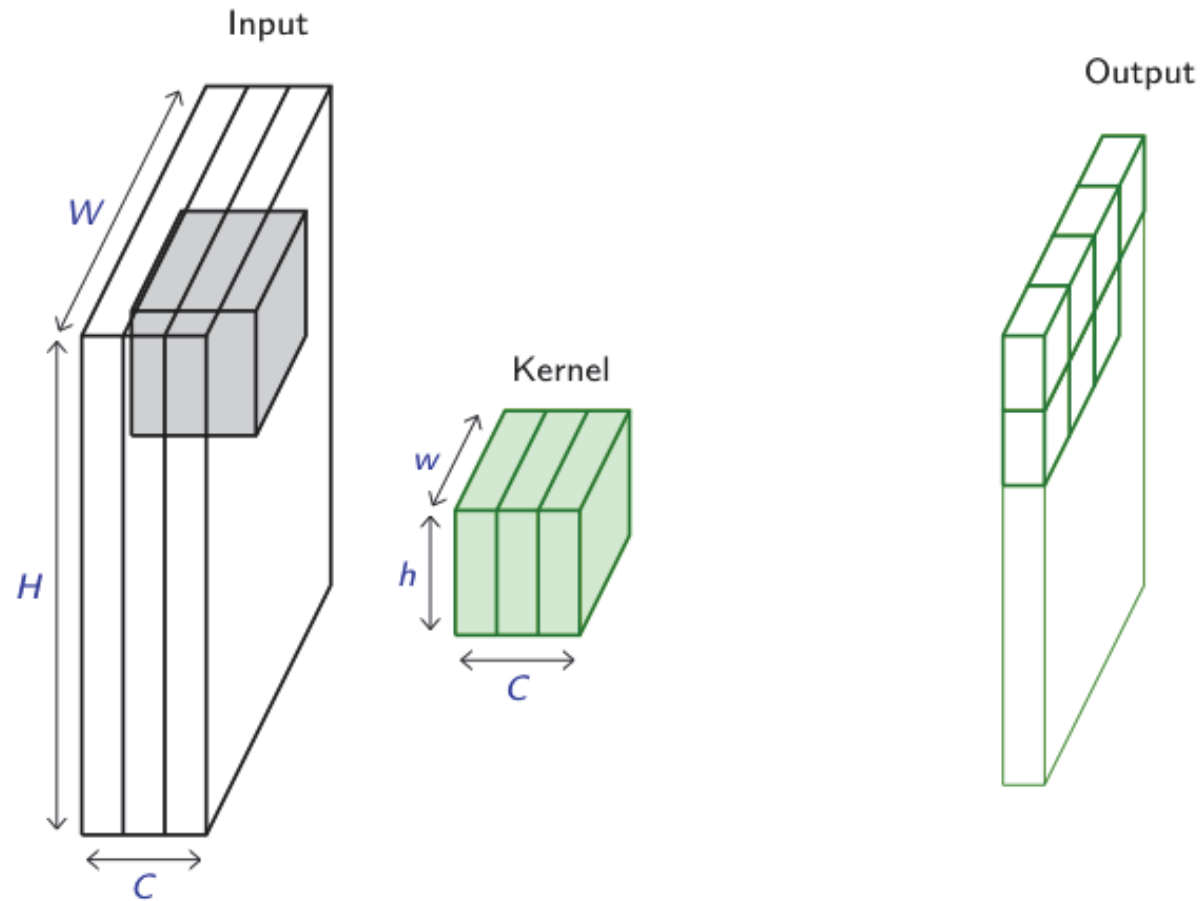
# Convolution 2d



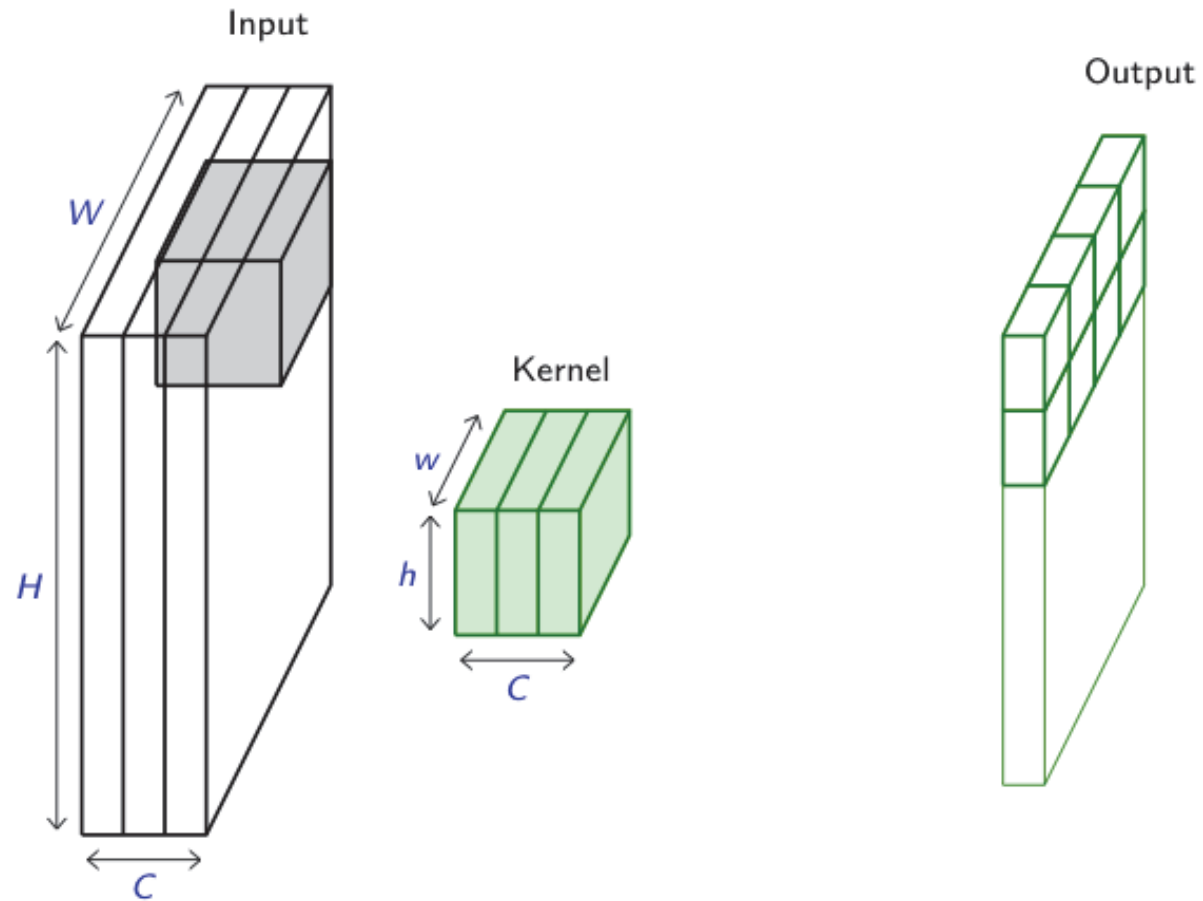
# Convolution 2d



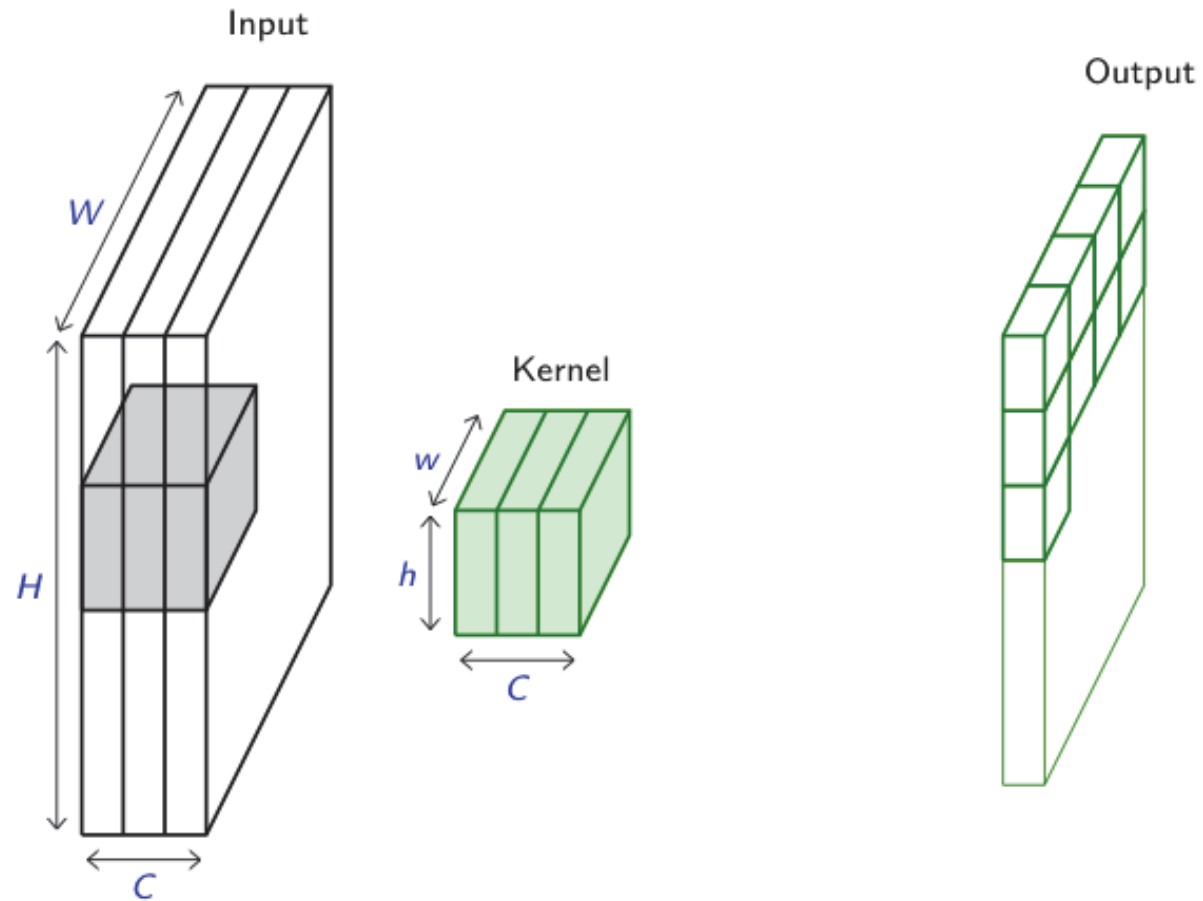
# Convolution 2d



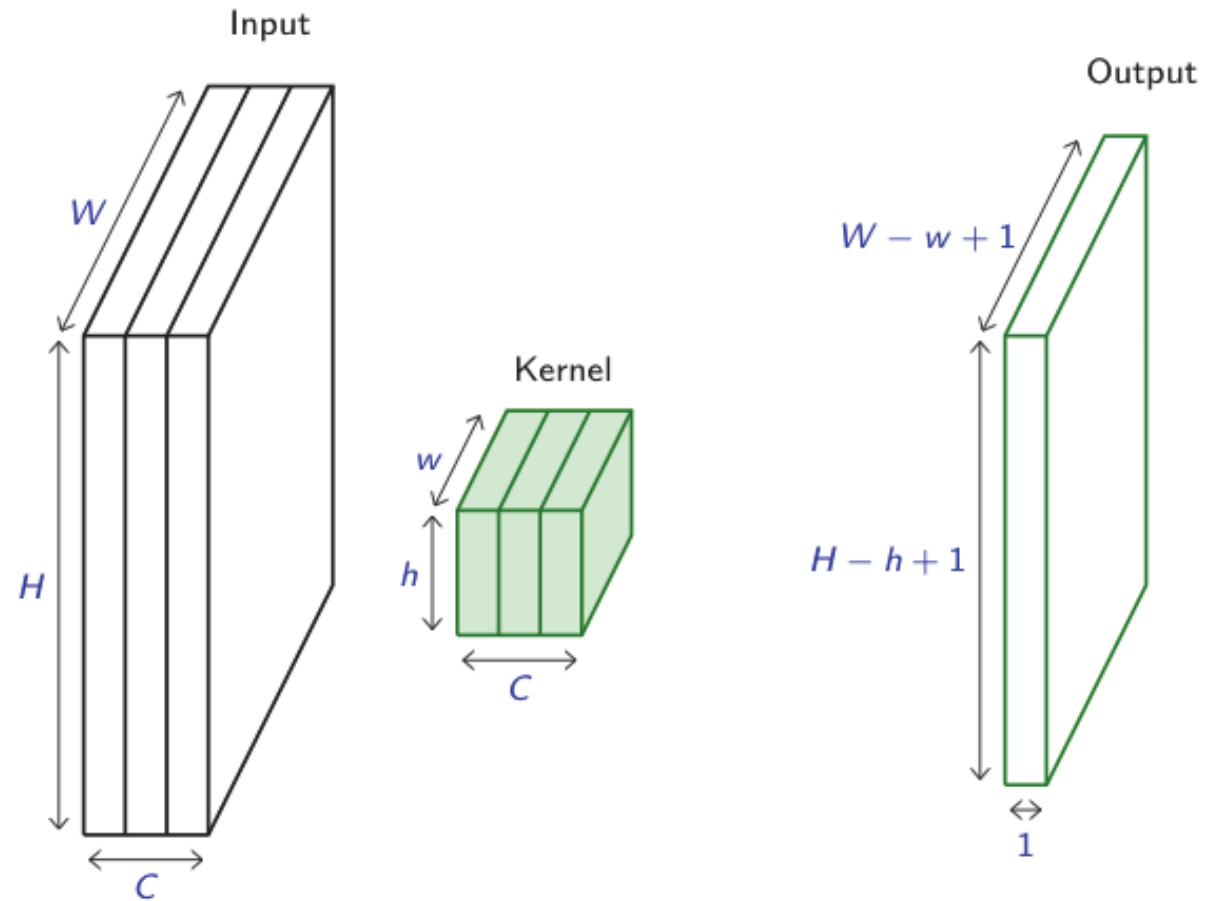
# Convolution 2d



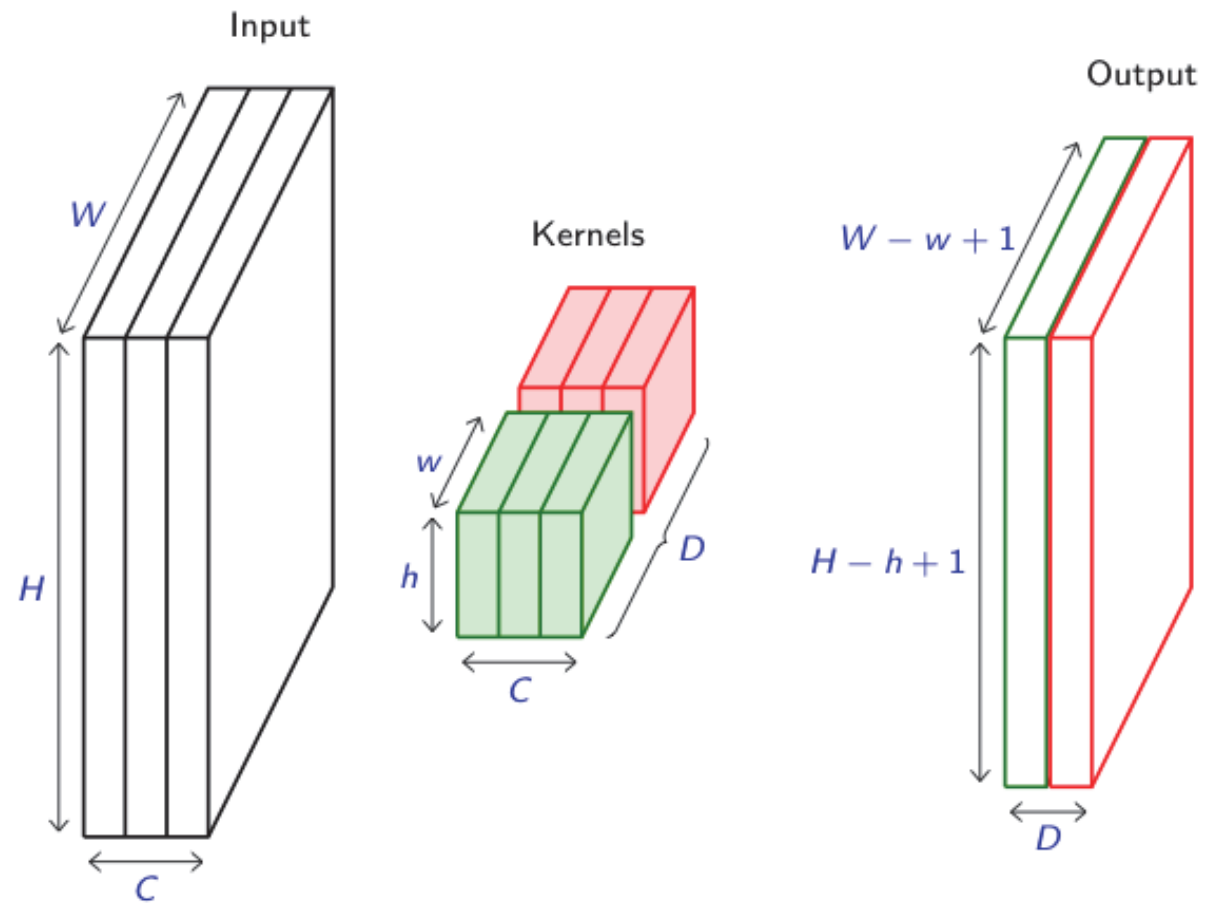
# Convolution 2d



# Convolution 2d



# Convolution 2d



# A convolution on an image

- Image:  $im$  of dimensions  $5 \times 5$
- Kernel:  $k$  of dimensions  $3 \times 3$

$3_0$	$3_1$	$2_2$	1	0
$0_2$	$0_2$	$1_0$	3	1
$3_0$	$1_1$	$2_2$	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

*These slides extensively use convolution visualisation by V. Dumoulin available at [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)*



# A convolution on an image

- Image:  $im$  of dimensions  $5 \times 5$
- Kernel:  $k$  of dimensions  $3 \times 3$

$3_0$	$3_1$	$2_2$	1	0
$0_2$	$0_2$	$1_0$	3	1
$3_0$	$1_1$	$2_2$	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

$$(k \star im)(x, y) = \sum_{n=0}^2 \sum_{m=0}^2 k(n, m) \cdot im(x + n - 1, y + m - 1)$$

*These slides extensively use convolution visualisation by V. Dumoulin available at [https://github.com/vdumoulin/conv\\_arithmetic](https://github.com/vdumoulin/conv_arithmetic)*

# Kernels as neural networks

$3_0$	$3_1$	$2_2$	1	0
$0_2$	$0_2$	$1_0$	3	1
$3_0$	$1_1$	$2_2$	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

- $x$  is a  $3 \times 3$  chunk of the image
- Each output neuron is parametrized with the kernel weights  $\mathbf{w}$

# Kernels as neural networks

$3_0$	$3_1$	$2_2$	1	0
$0_2$	$0_2$	$1_0$	3	1
$3_0$	$1_1$	$2_2$	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

- $x$  is a  $3 \times 3$  chunk of the image
- Each output neuron is parametrized with the kernel weights  $\mathbf{w}$

The activation obtained by sliding the  $3 \times 3$  window and computing:

$$z(x) = \text{relu}(\mathbf{w}^T x + b)$$

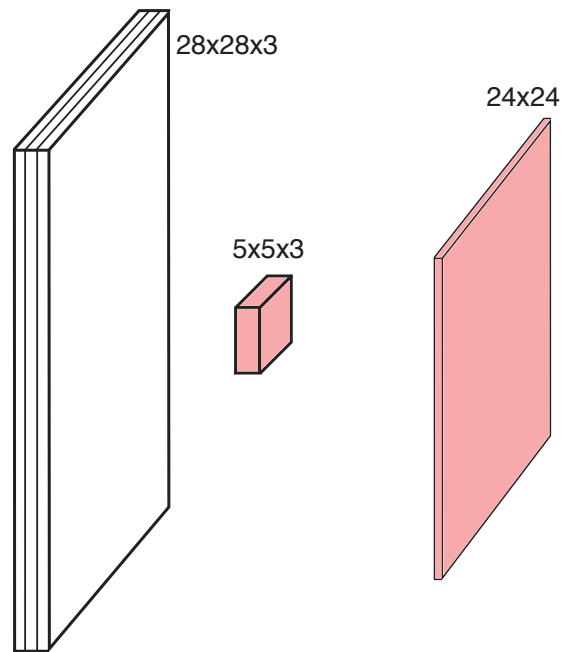
# Channels

Colored image = tensor of shape (height, width, channels)

# Channels

Colored image = tensor of shape (height, width, channels)

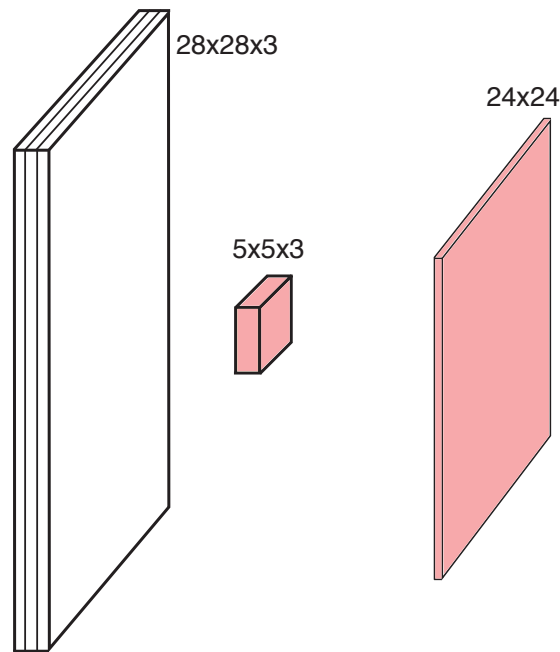
Convolutions can be computed across channels:



# Channels

Colored image = tensor of shape (height, width, channels)

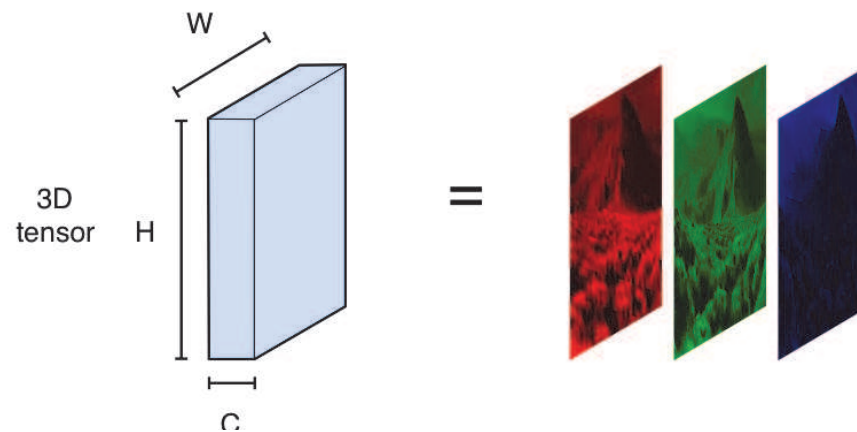
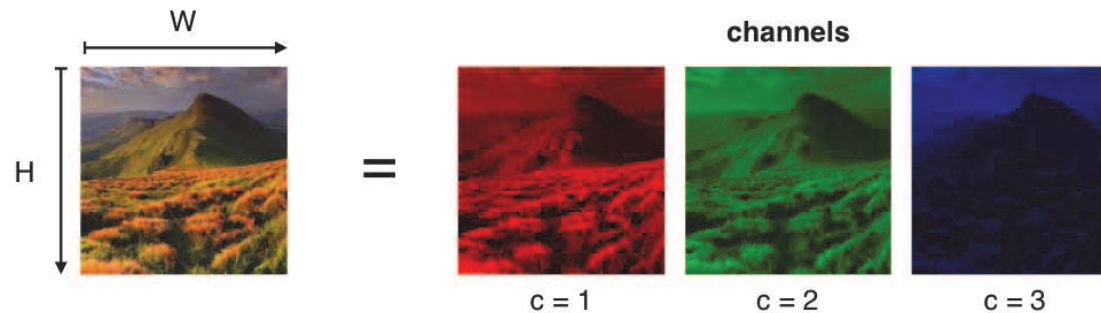
Convolutions can be computed across channels:



$$(k \star im)(x, y) = \sum_{c=0}^2 \sum_{n=0}^4 \sum_{m=0}^4 k(n, m, c) \cdot im(x + n - 2, y + m - 2, c)$$

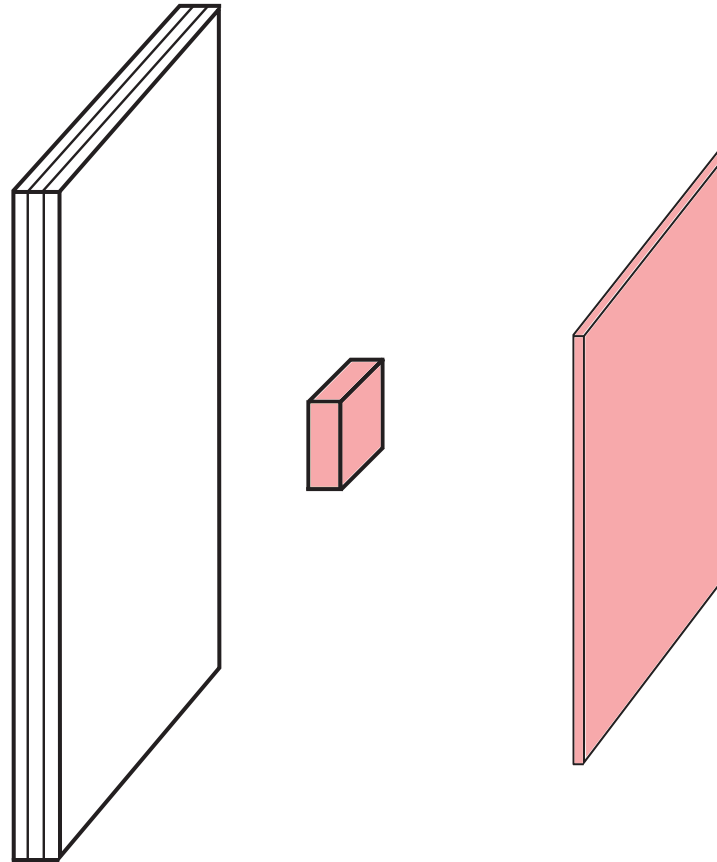
# Channels

- For first layer, RGB channels of input image can be easily visualized
- Number of channels is typically increased at deeper levels of the network



# Multiple convolutions

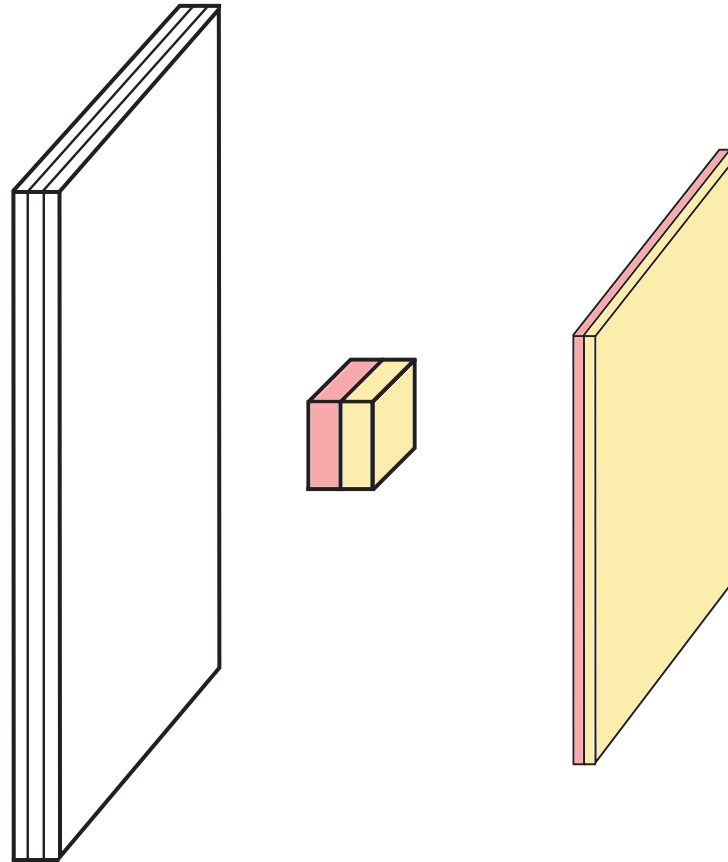
Each filter generates a one-channel feature map of responses.





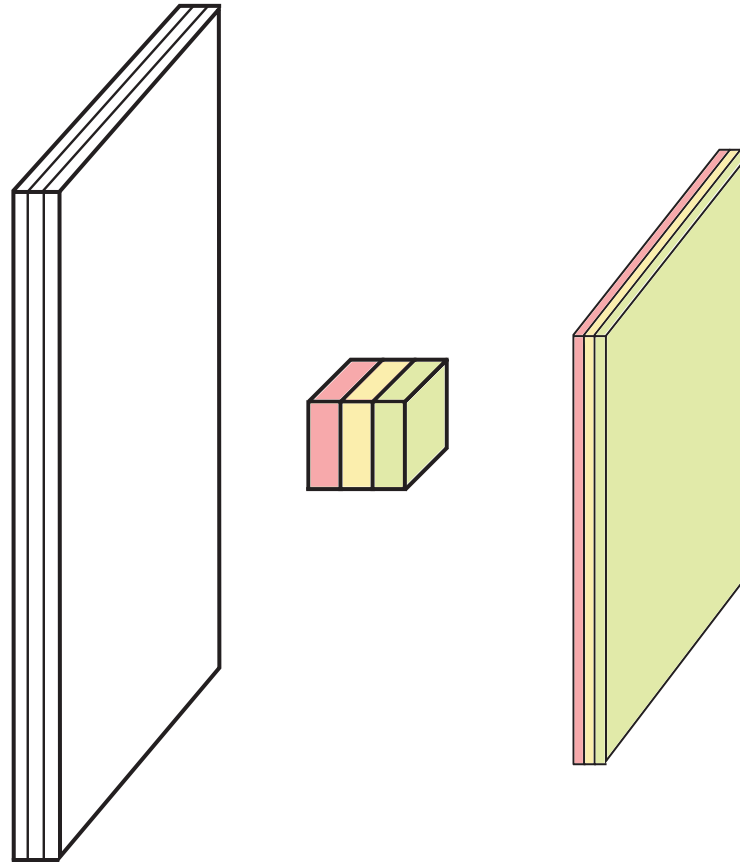
# Multiple convolutions

Each filter generates a one-channel feature map of responses.



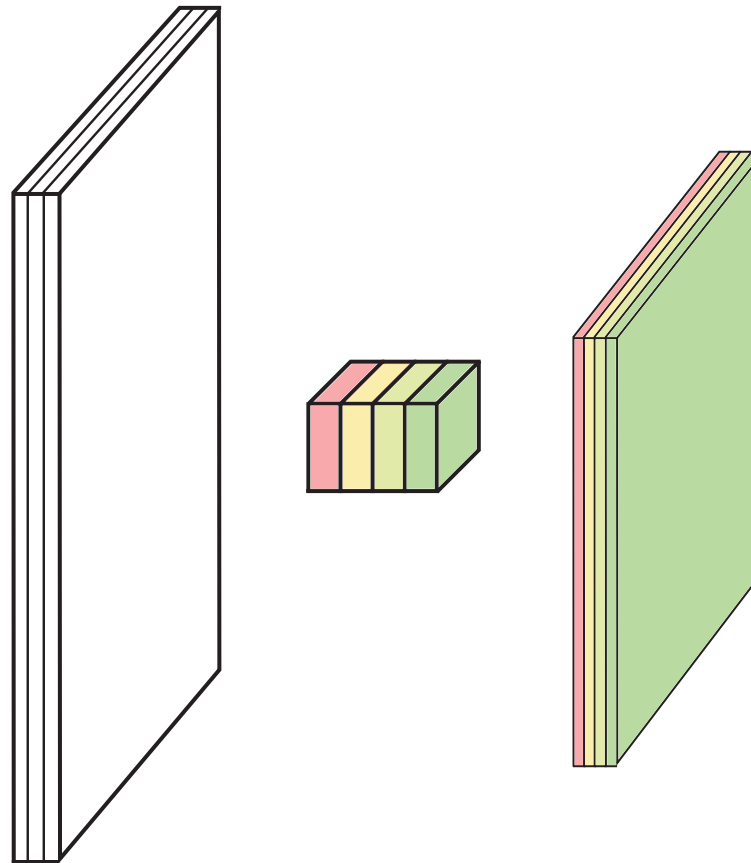
# Multiple convolutions

Each filter generates a one-channel feature map of responses.



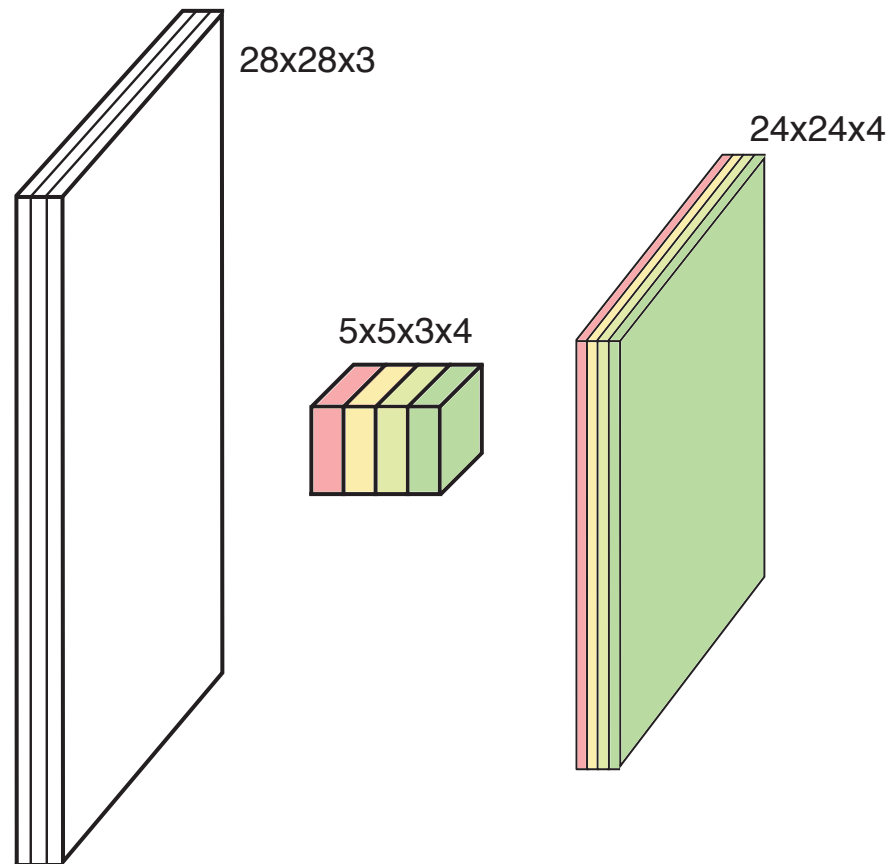
# Multiple convolutions

Each filter generates a one-channel feature map of responses.



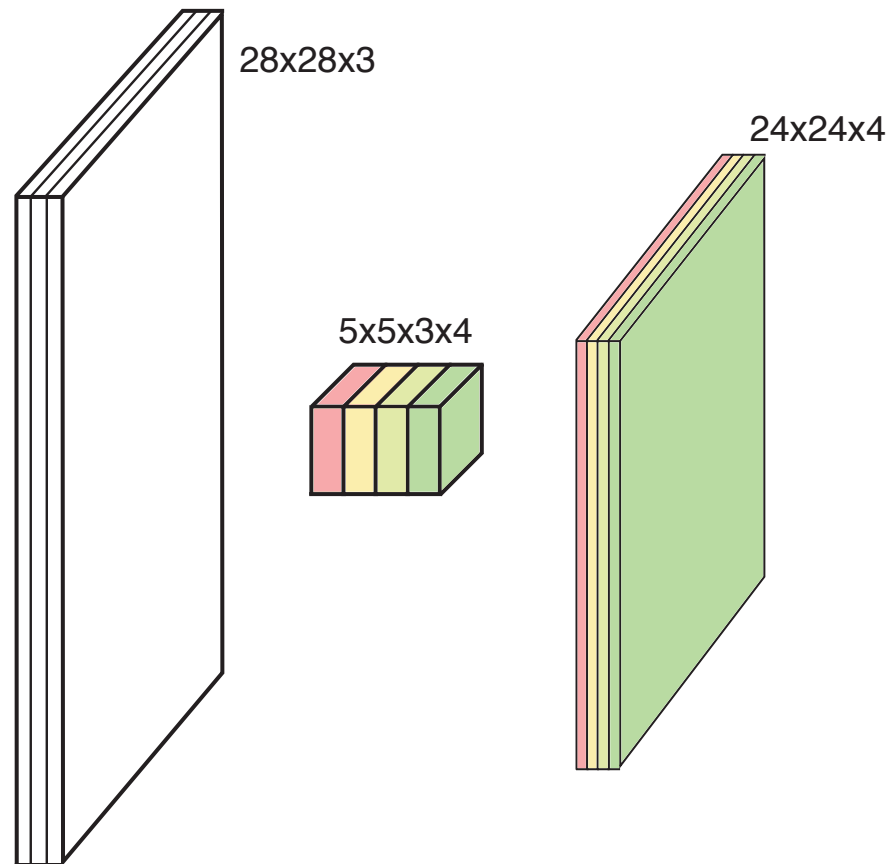
# Multiple convolutions

Each filter generates a one-channel feature map of responses.



# Multiple convolutions

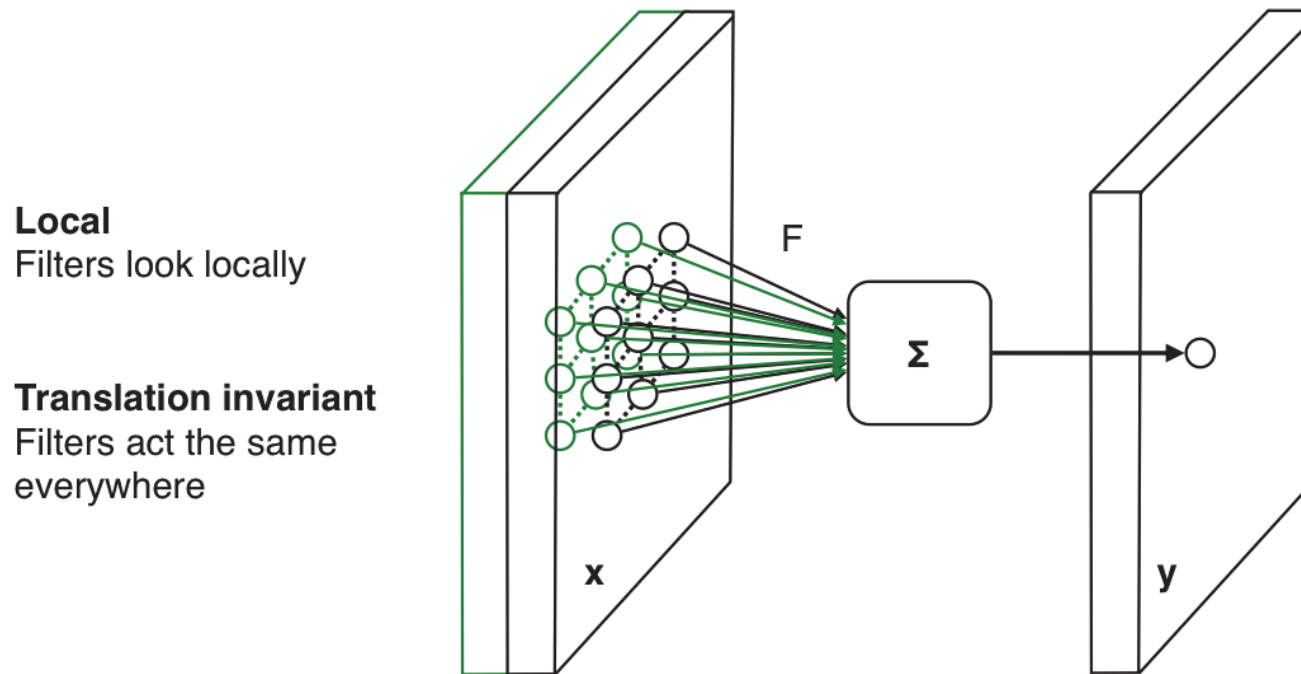
Each filter generates a one-channel feature map of responses.



- Kernel size aka receptive field (usually 1, 3, 5, 7, 11)
- Output dimension:  $\text{length} - \text{kernel\_size} + 1$

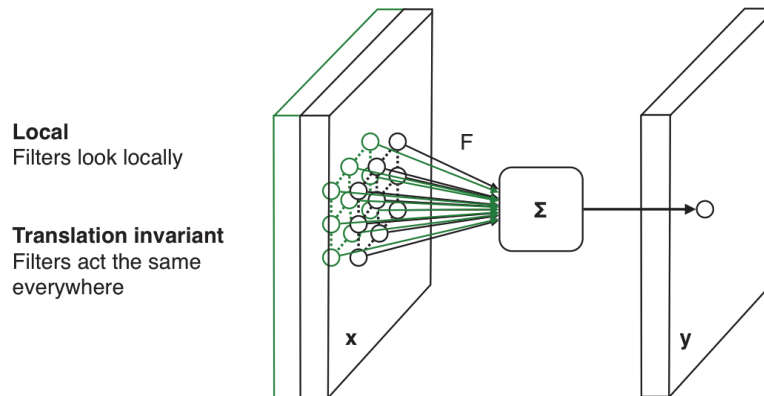
# Multiple convolutions

- Since convolutions output one scalar at a time, they can be seen as an individual neuron from a MLP with a receptive field limited to the dimensions of the kernel
- The same neuron is "fired" over multiple areas from the input.

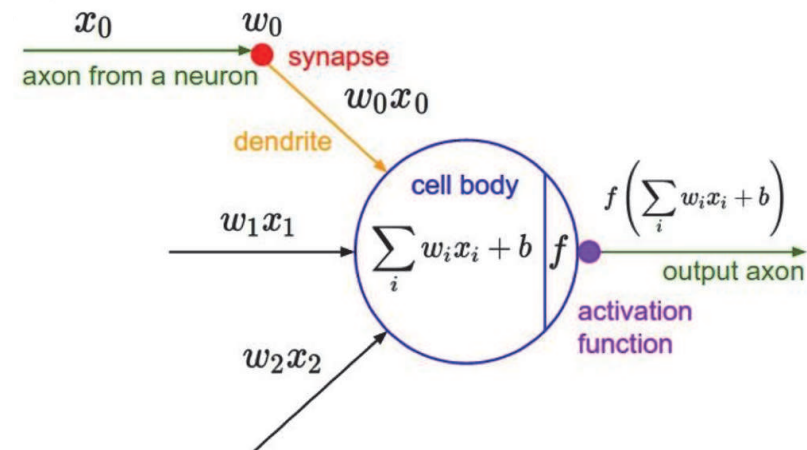


# Multiple convolutions

- Since convolutions output one scalar at a time, they can be seen as an individual neuron from a MLP with a receptive field limited to the dimensions of the kernel
- The same neuron is "fired" over multiple areas from the input.

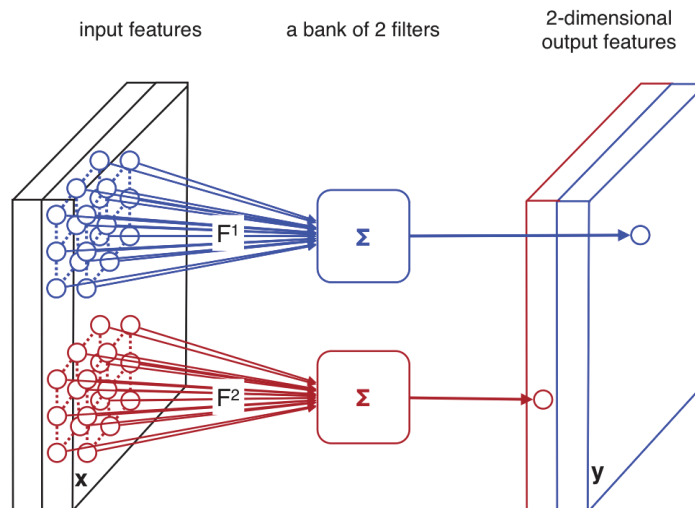


Remember this?

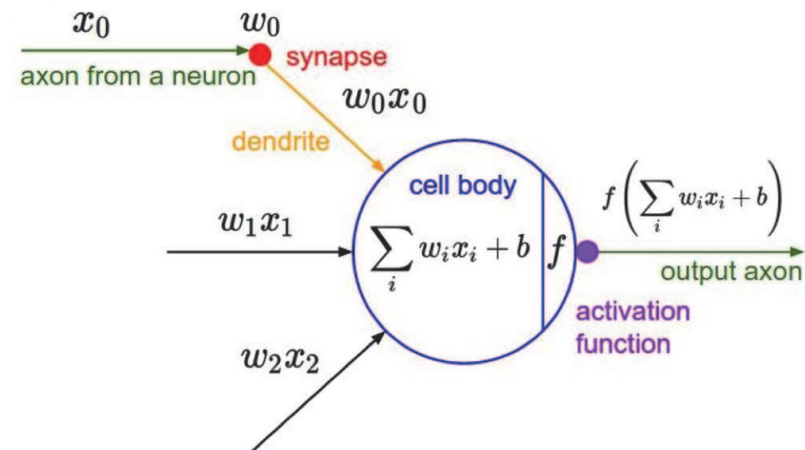


# Multiple convolutions

- Since convolutions output one scalar at a time, they can be seen as an individual neuron from a MLP with a receptive field limited to the dimensions of the kernel
- The same neuron is "fired" over multiple areas from the input.



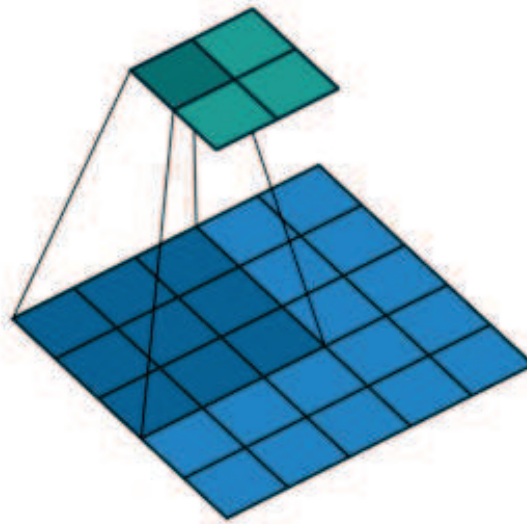
Remember this?





# Strides

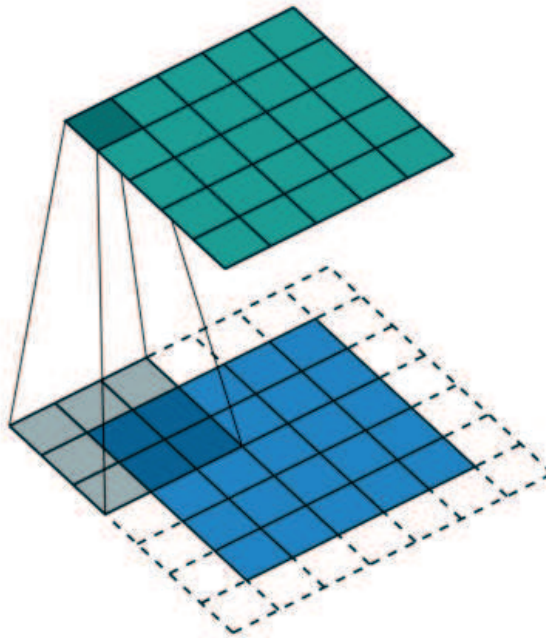
- Strides: increment step size for the convolution operator
- Reduces the size of the output map



Example with kernel size  $3 \times 3$  and a stride of 2 (image in blue)

# Padding

- Padding: artificially fill borders of image
- Useful to keep spatial dimension constant across filters
- Useful with strides and large receptive fields
- Usually: fill with 0s



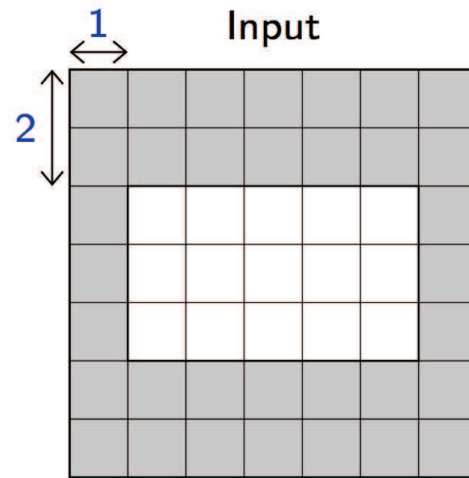
# Padding

- Example: input  $C \times 3 \times 5$

Input

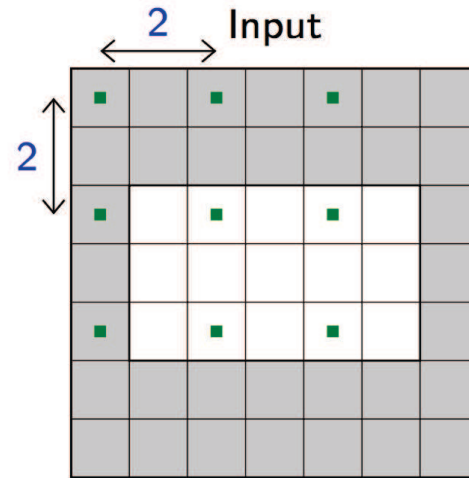

# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1)



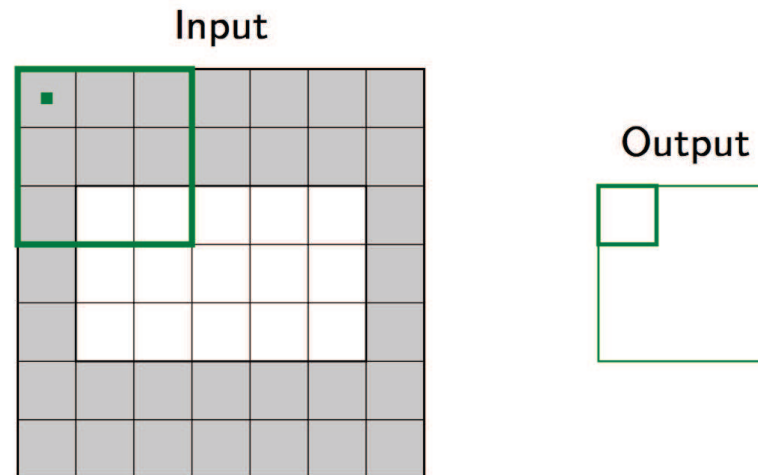
# Padding

- Example: input  $C \times 3 \times 5$ , padding of  $(2, 1)$ , a stride of  $(2, 2)$



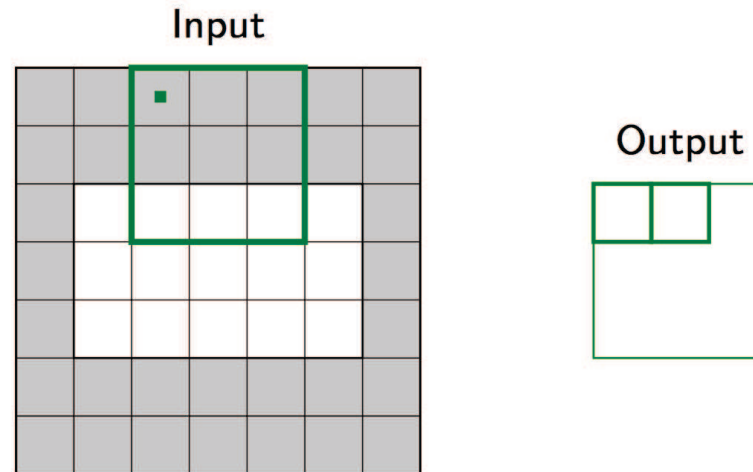
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



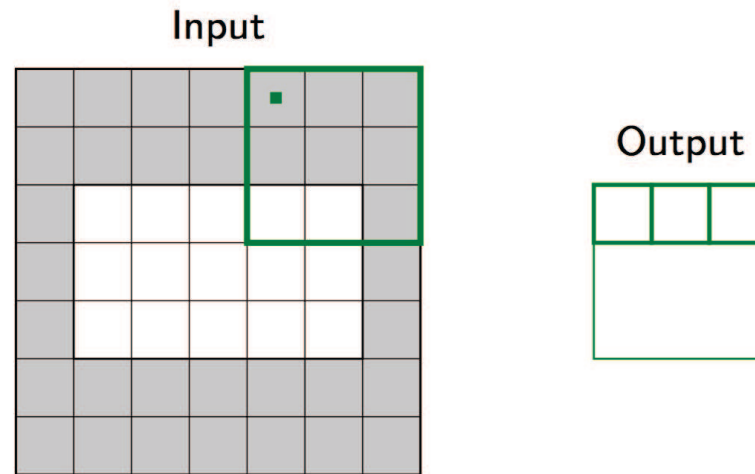
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



# Padding

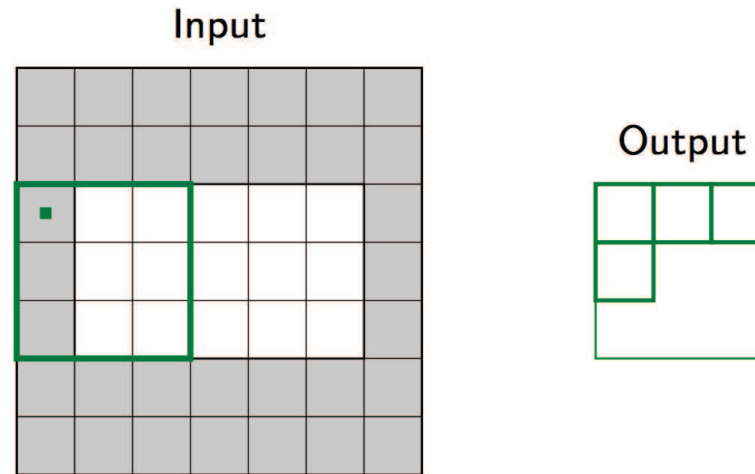
- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$





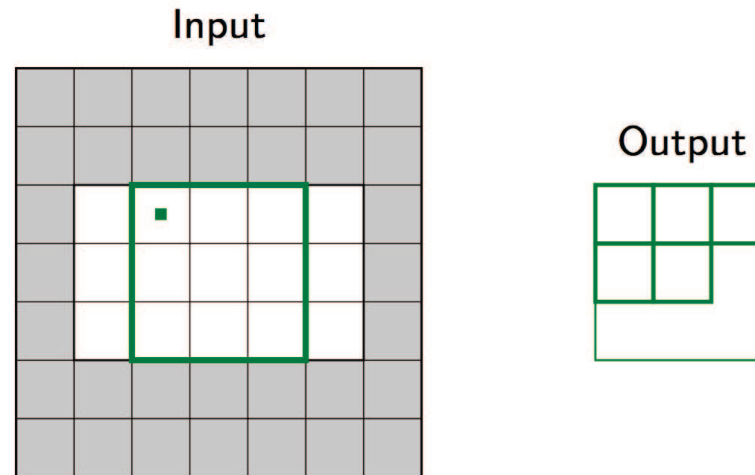
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



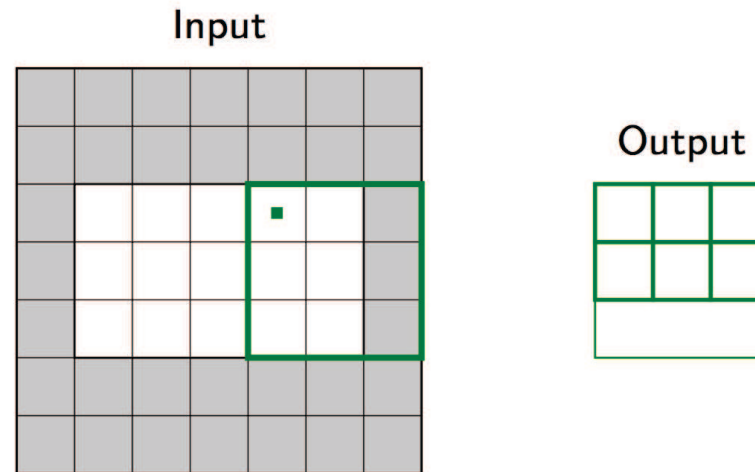
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



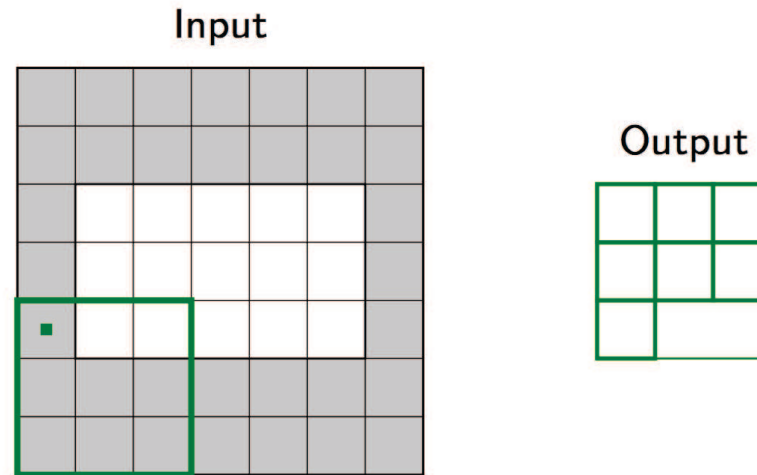
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



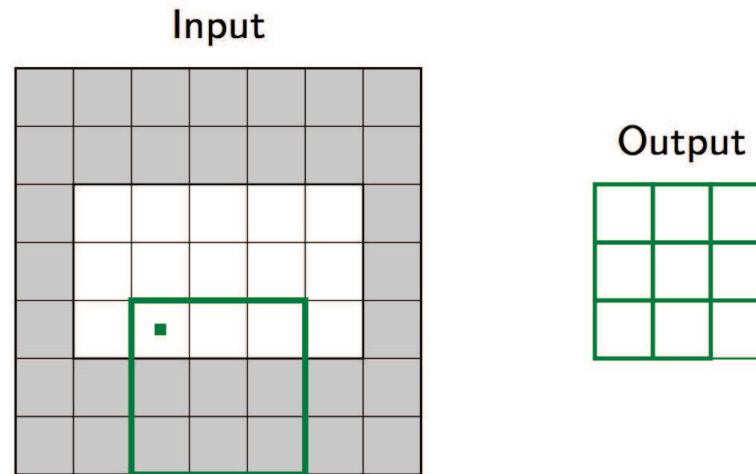
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



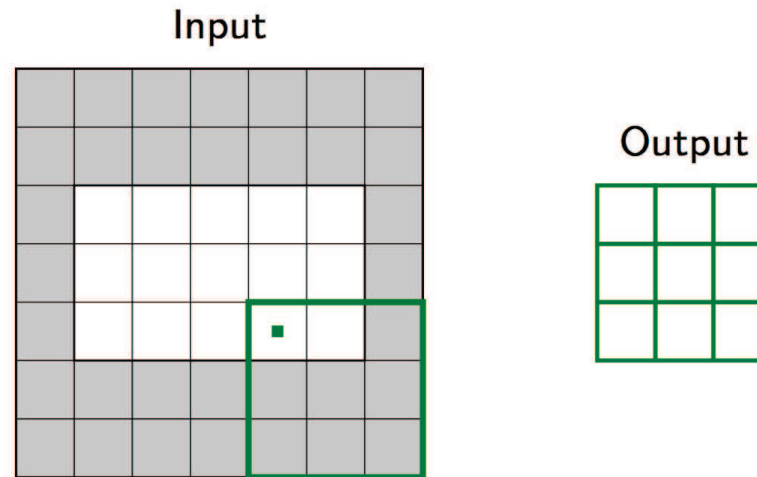
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



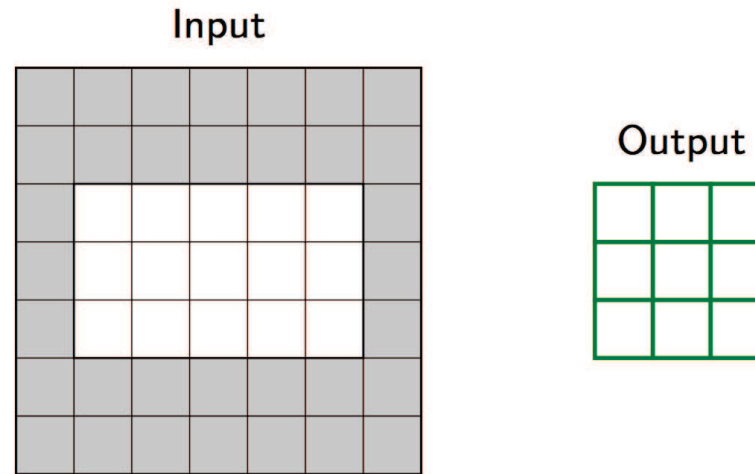
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



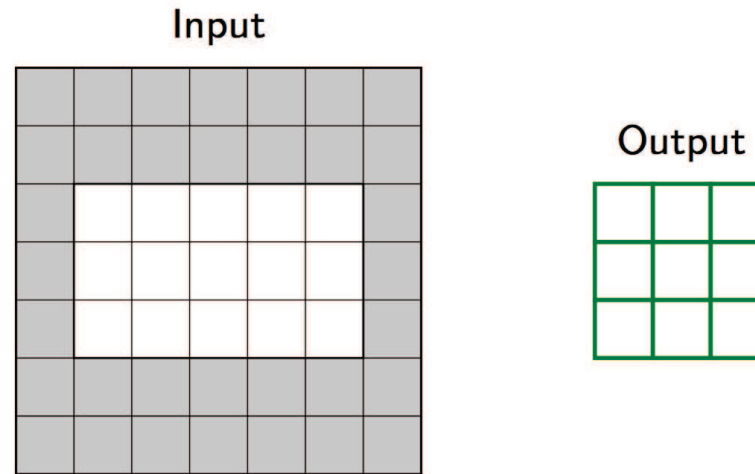
# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



# Padding

- Example: input  $C \times 3 \times 5$ , padding of (2, 1), a stride of (2, 2), kernel of size  $C \times 3 \times 5$



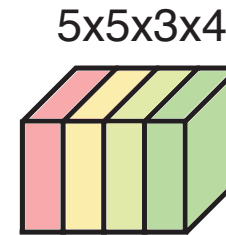
- Pooling operations have a default stride equal to their kernel size, and convolutions have a default stride of 1.
- Padding can be useful to generate an output of same size as the input.



# Dealing with shapes

Kernel shape  $(F, F, C^i, C^o)$

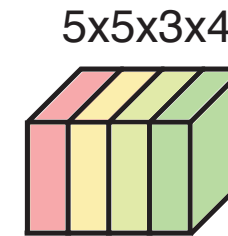
- $F \times F$  kernel size,
- $C^i$  input channels
- $C^o$  output channels



# Dealing with shapes

Kernel shape  $(F, F, C^i, C^o)$

- $F \times F$  kernel size,
- $C^i$  input channels
- $C^o$  output channels

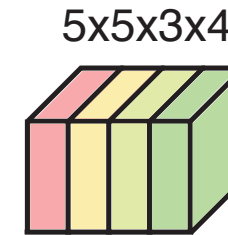


Number of parameters:  $(F \times F \times C^i + 1) \times C^o$

# Dealing with shapes

Kernel shape  $(F, F, C^i, C^o)$

- $F \times F$  kernel size,
- $C^i$  input channels
- $C^o$  output channels



Number of parameters:  $(F \times F \times C^i + 1) \times C^o$

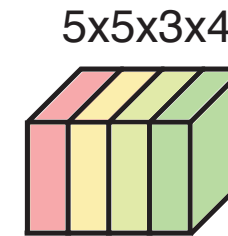
Activation shapes:

- Input  $(W^i, H^i, C^i)$
- Output  $(W^o, H^o, C^o)$

# Dealing with shapes

Kernel shape  $(F, F, C^i, C^o)$

- $F \times F$  kernel size,
- $C^i$  input channels
- $C^o$  output channels



Number of parameters:  $(F \times F \times C^i + 1) \times C^o$

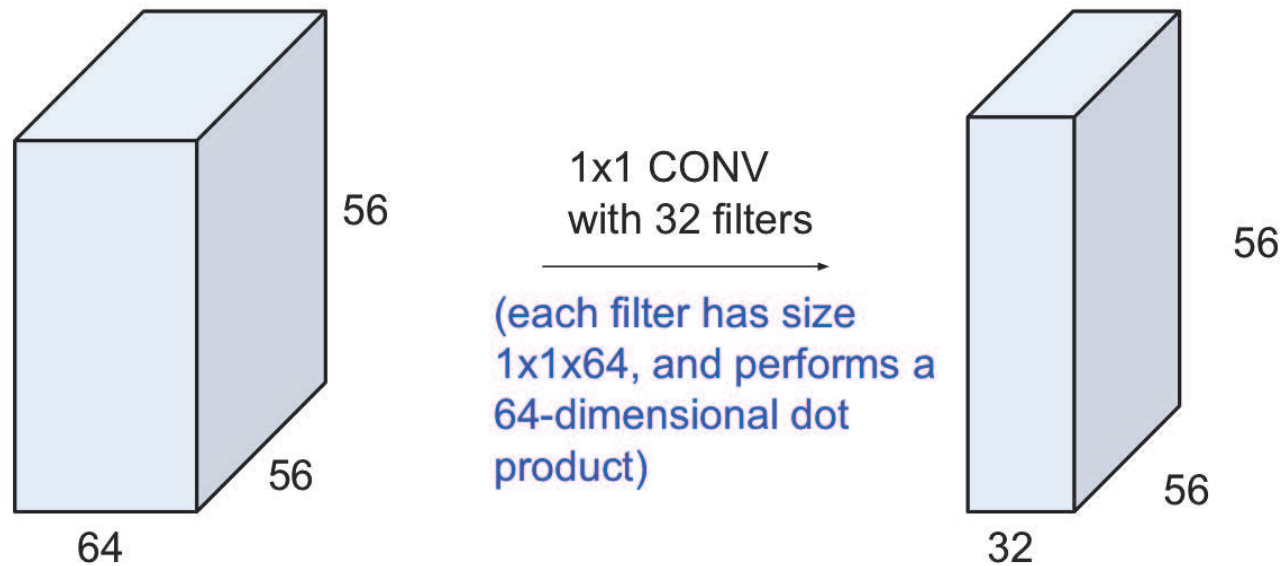
Activation shapes:

- Input  $(W^i, H^i, C^i)$
- Output  $(W^o, H^o, C^o)$

$$W^o = (W^i - F + 2P)/S + 1$$

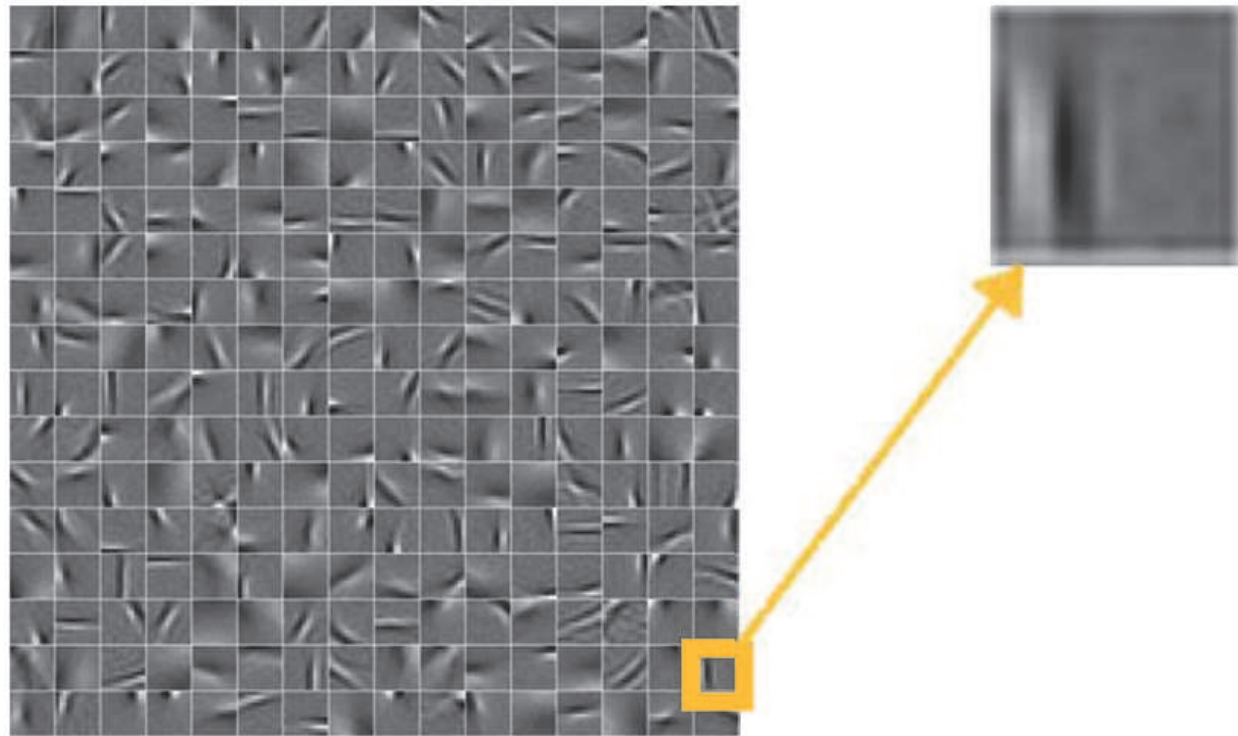
# Convolutions

1x1 convolution layers: aggregating pixel information from all feature maps



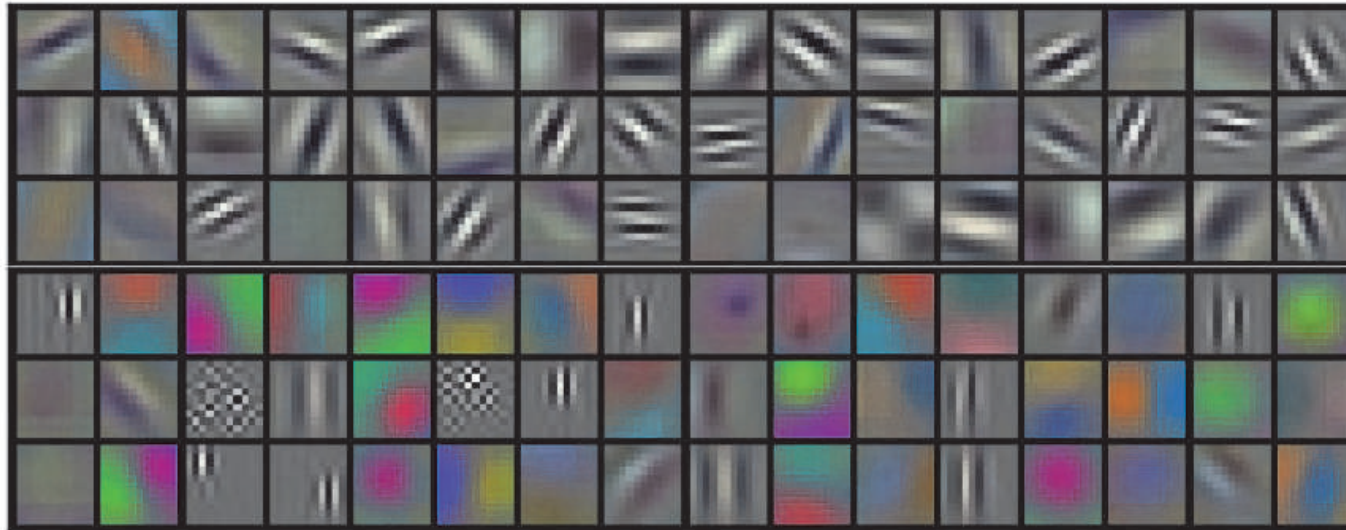
# Convolutions

- A bank of 256 filters (learned from data)
- Each filter is 1d (it applies to a grayscale image)
- Each filter is 16 x 16 pixels



# Convolutions

- A bank of 256 filters (learned from data)
- 3D filters for RGB inputs



# Convolutions

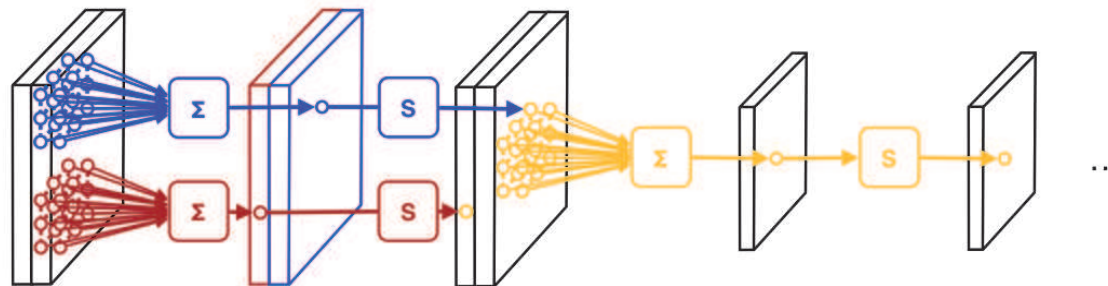
## Implementation

- Arrange data for optimized matrix multiplication (using GEMM)
- Makes life easier for backprop

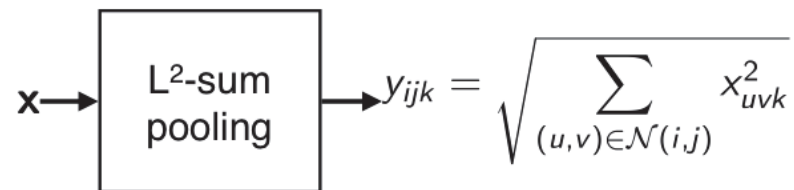
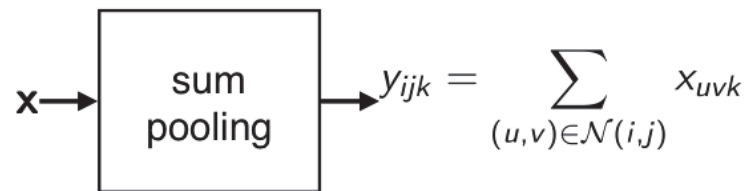
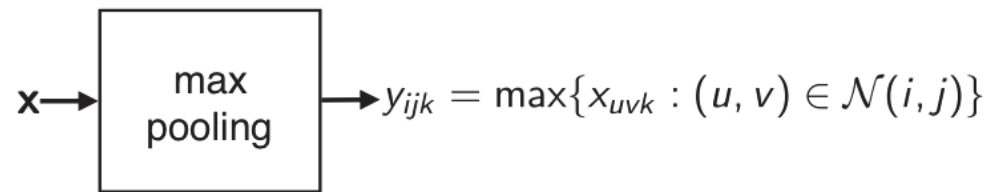


# Downsampling

- Downsampling by a factor  $S$  amount to keeping only one every  $S$  pixels, discarding others
- Filter banks often incorporate or are followed by **2x** output downsampling
- Downsampling is often matched with an increase in the number of feature channels
- Overall the volume of the tensors decreases slowly



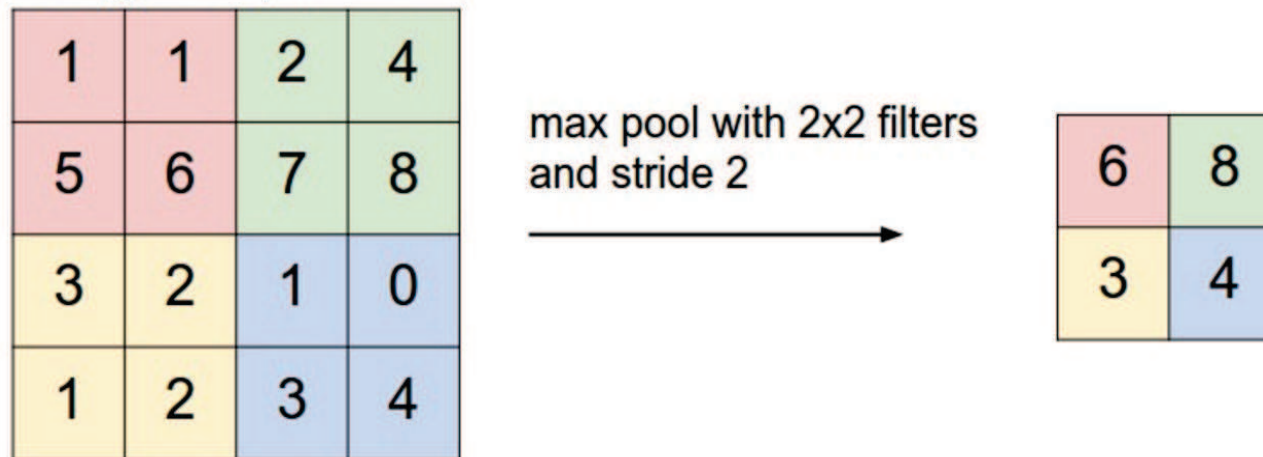
# Spatial pooling



By far, the most common variant is **max pooling**.

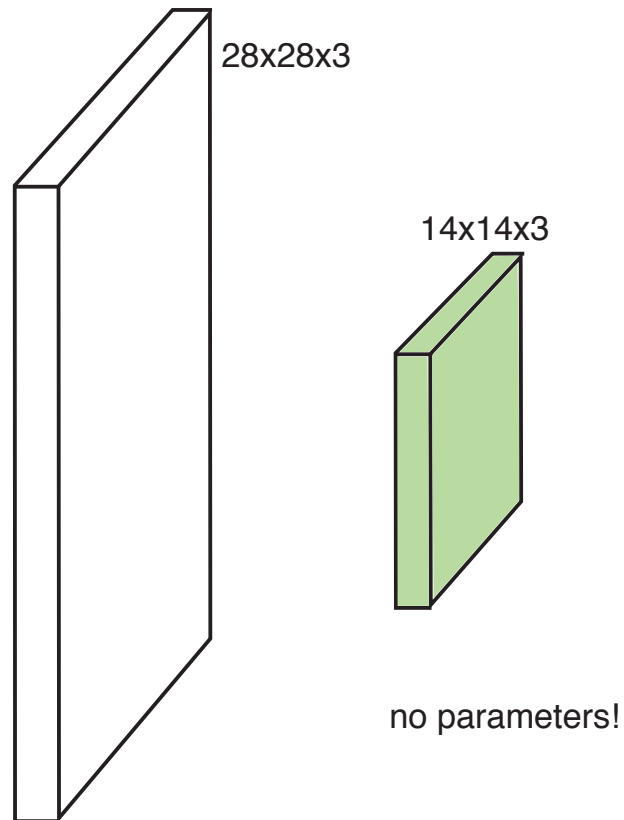
# Pooling

- Spatial dimension reduction
- Local invariance
- No parameters: max or average of 2x2 units

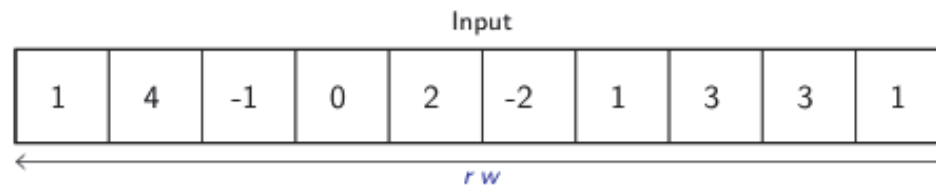


# Pooling

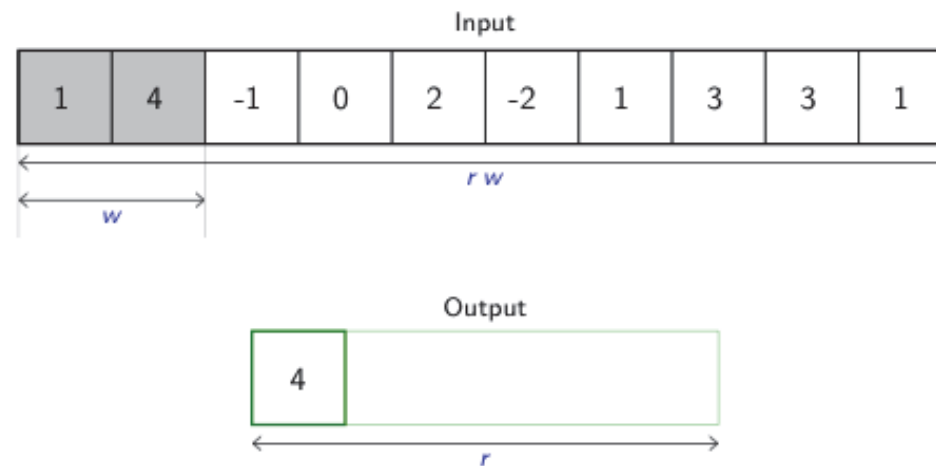
- Spatial dimension reduction
- Local invariance
- No parameters: max or average of 2x2 units



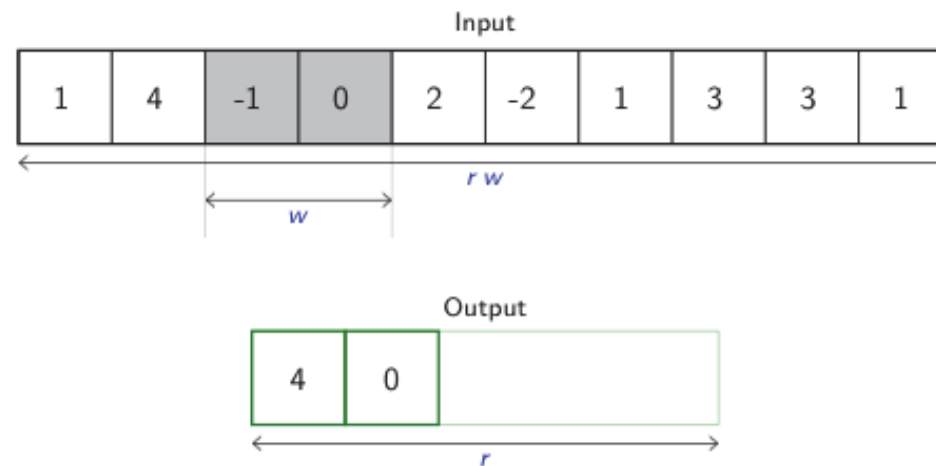
# Max-Pooling 1d



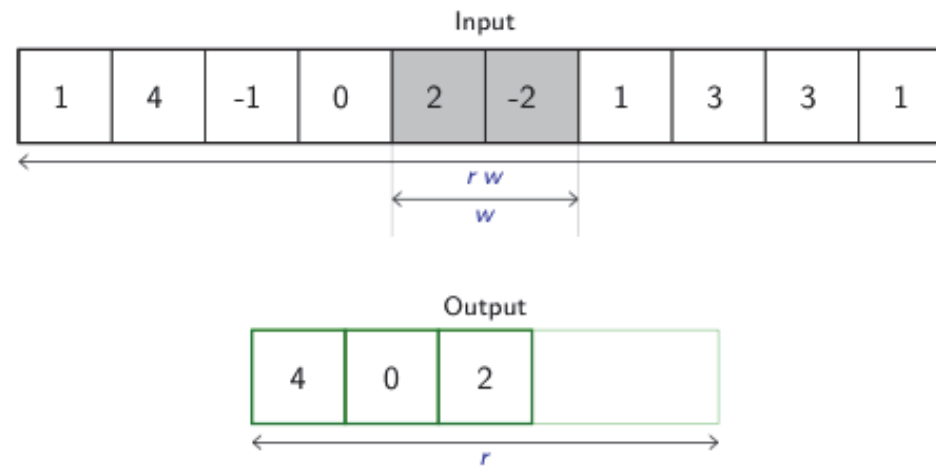
# Max-Pooling 1d



# Max-Pooling 1d

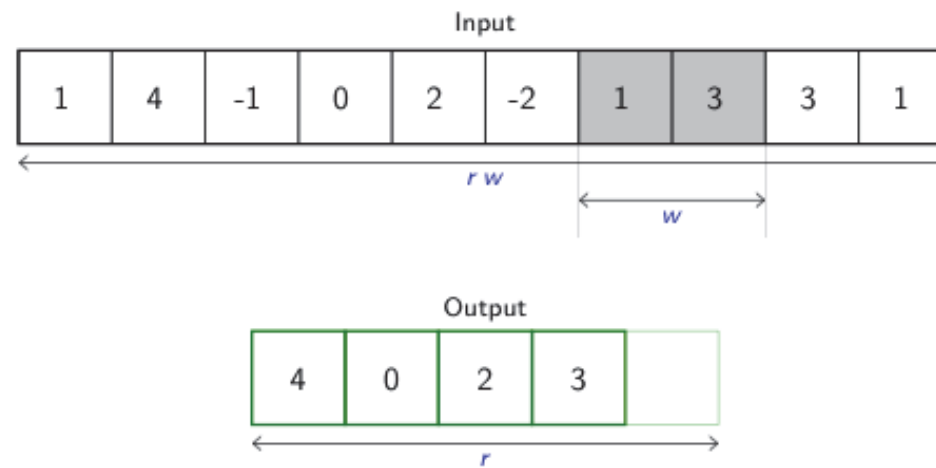


# Max-Pooling 1d

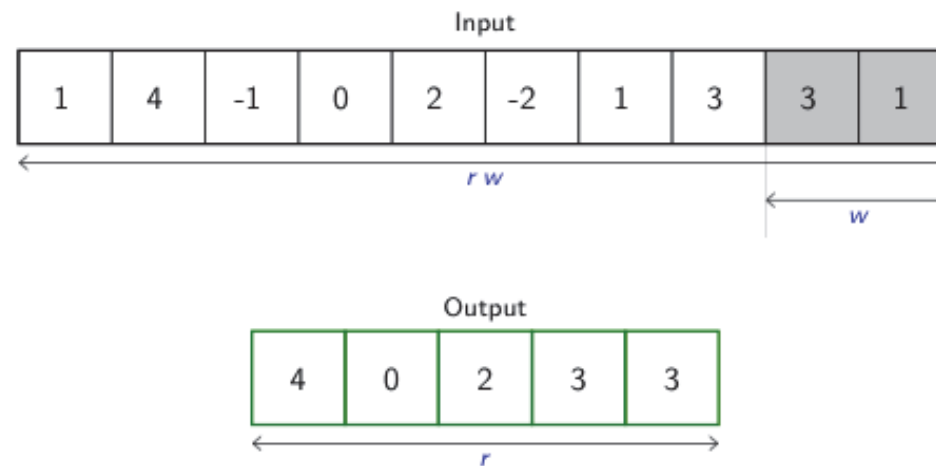




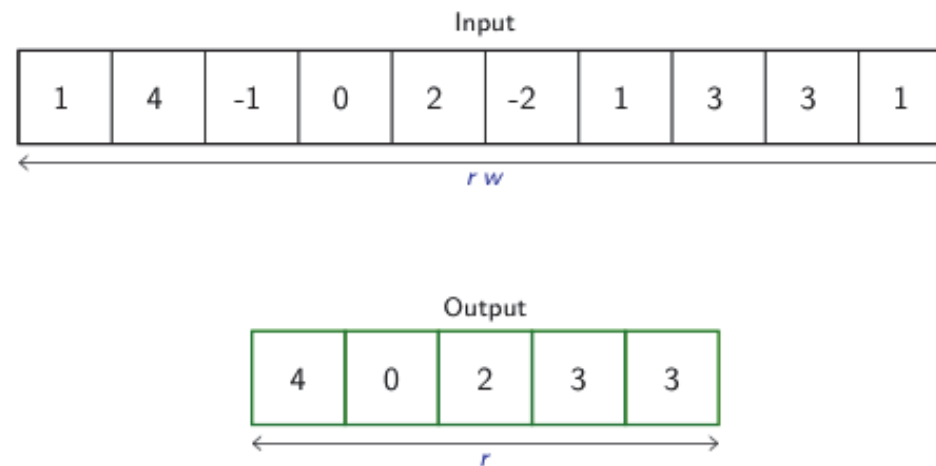
# Max-Pooling 1d



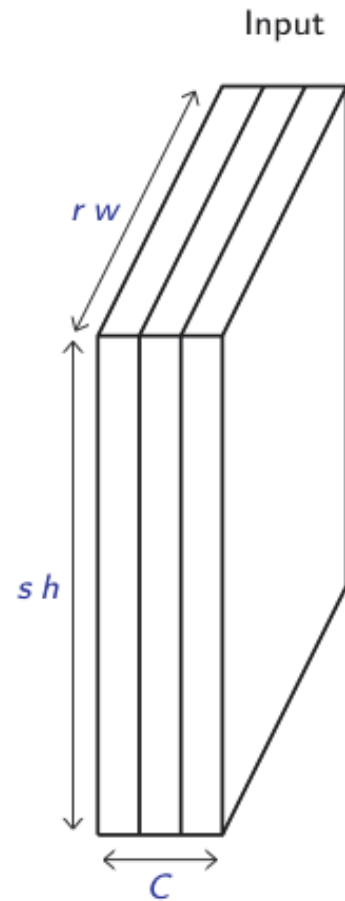
# Max-Pooling 1d



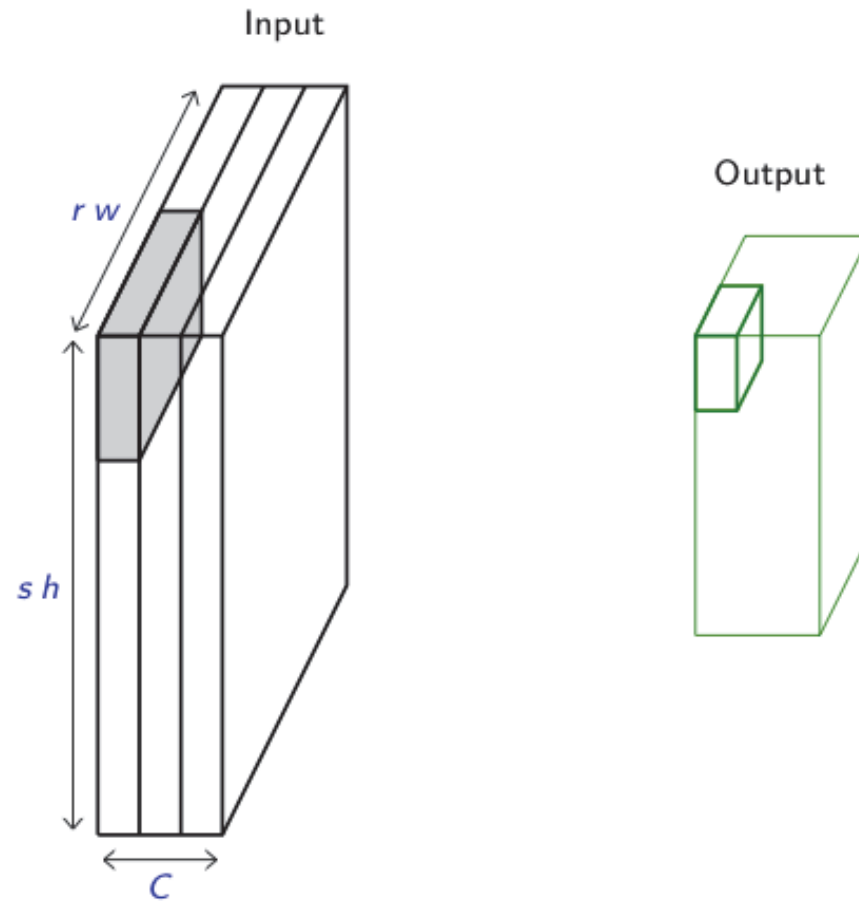
# Max-Pooling 1d



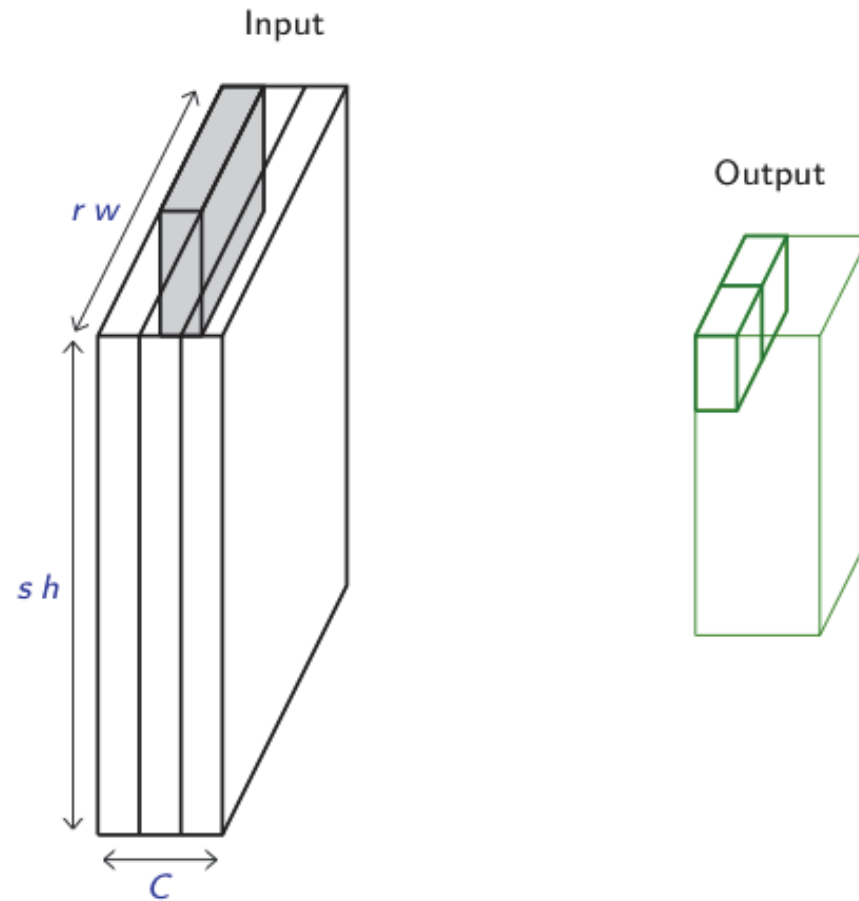
# Max-Pooling 2d



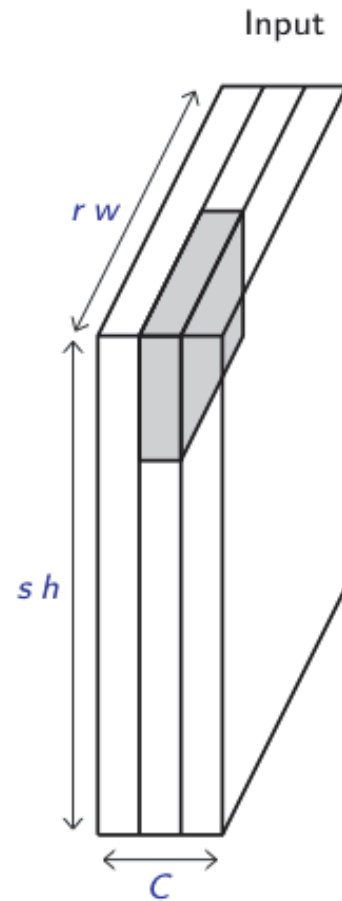
# Max-Pooling 2d



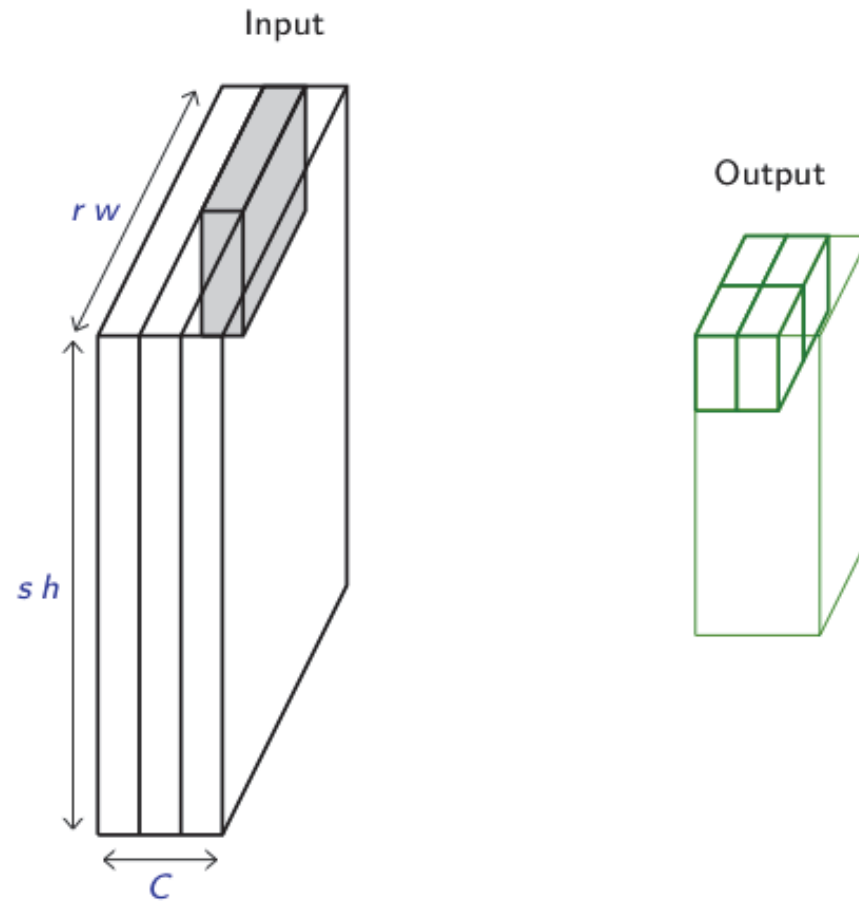
# Max-Pooling 2d



# Max-Pooling 2d

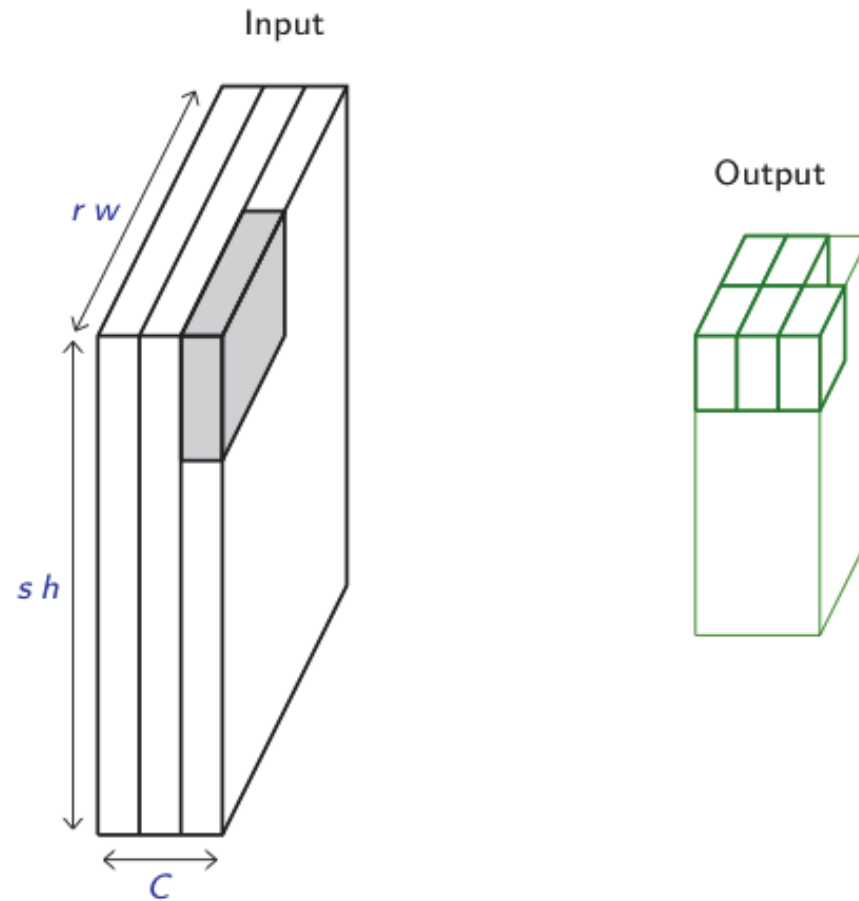


# Max-Pooling 2d

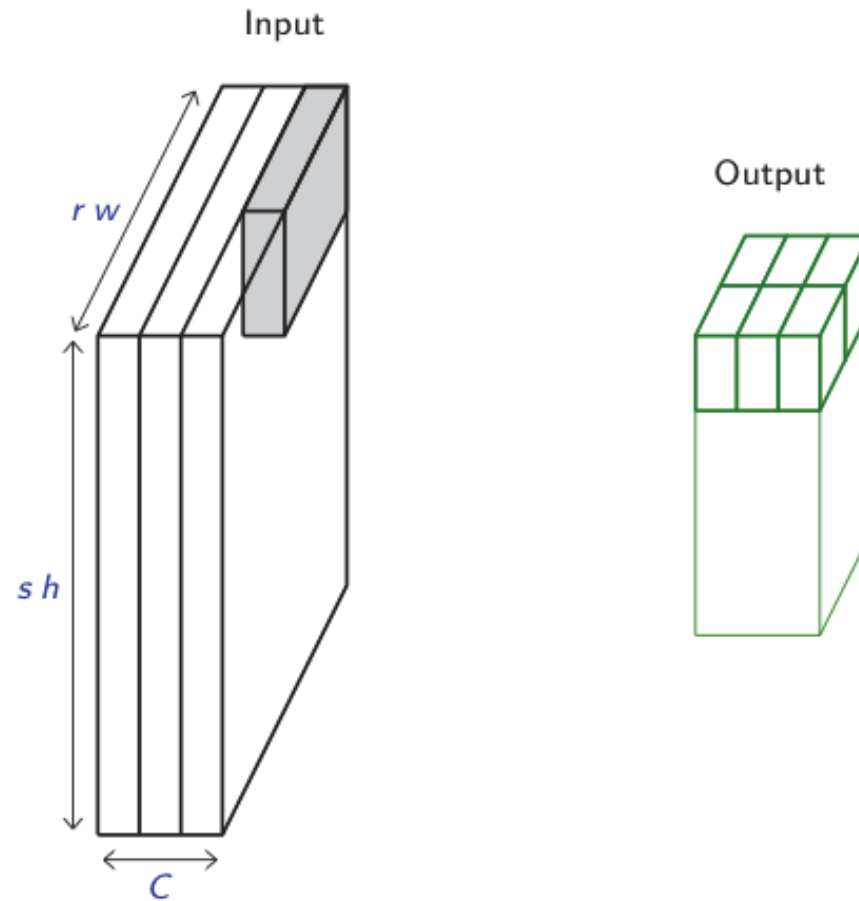




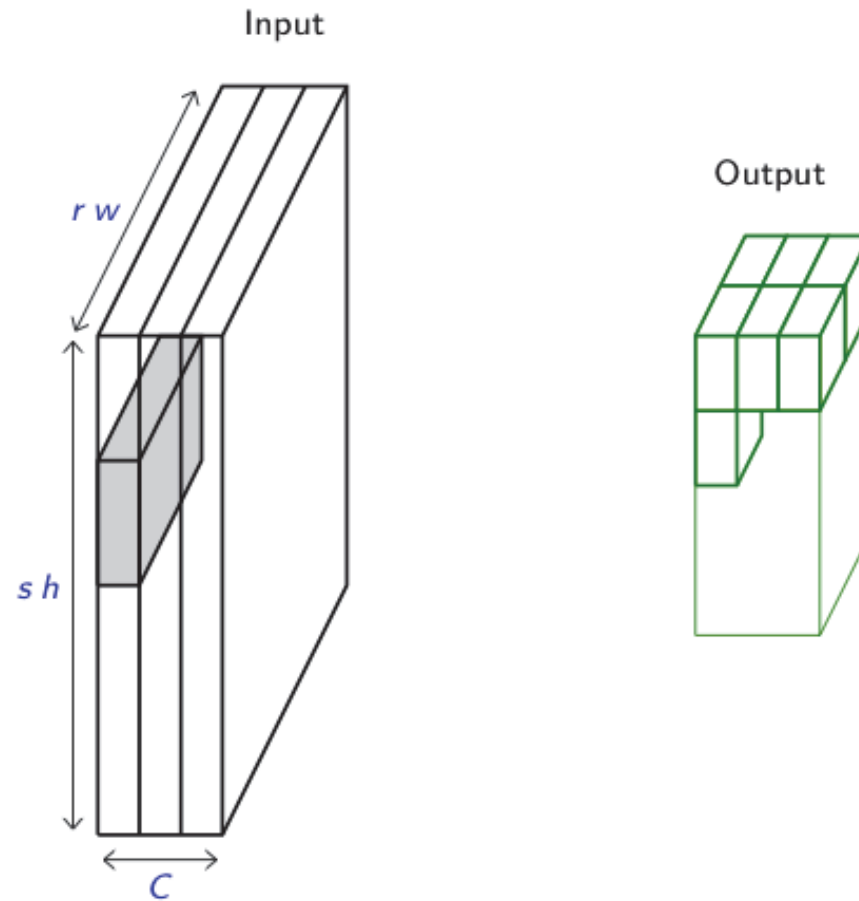
# Max-Pooling 2d



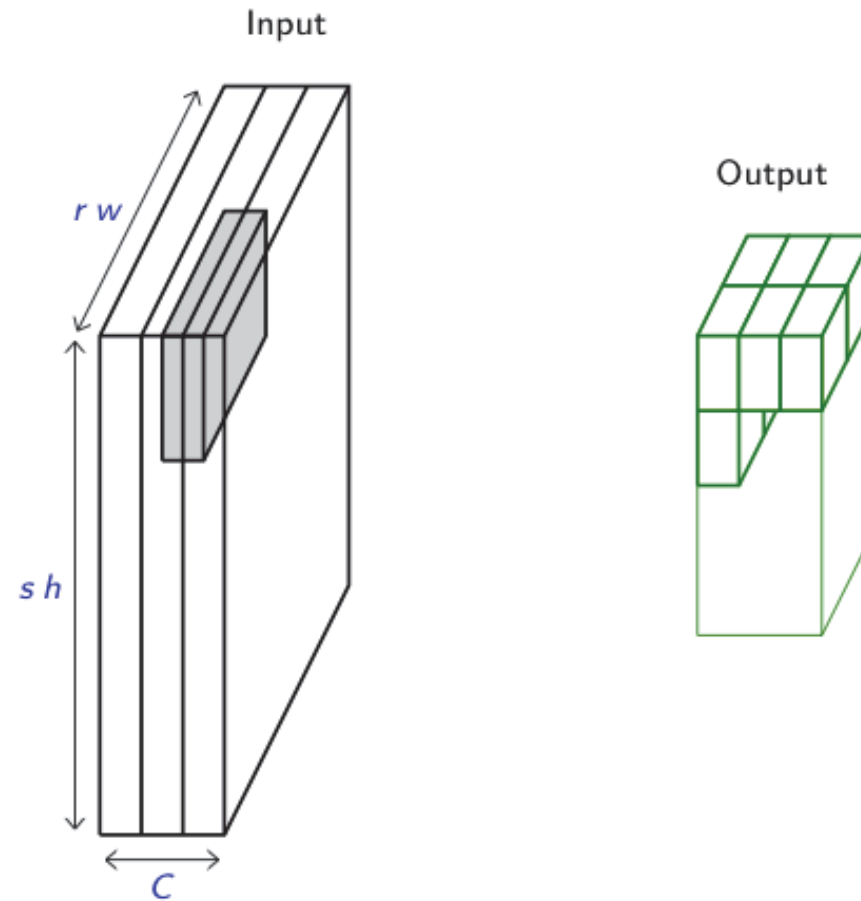
# Max-Pooling 2d



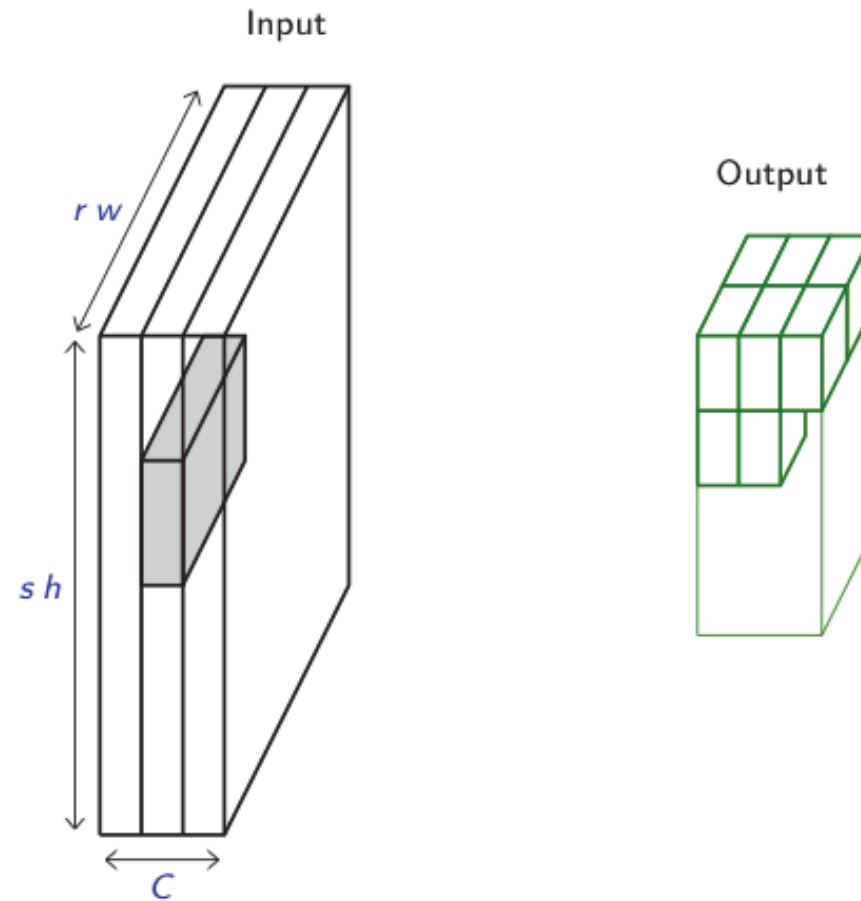
# Max-Pooling 2d



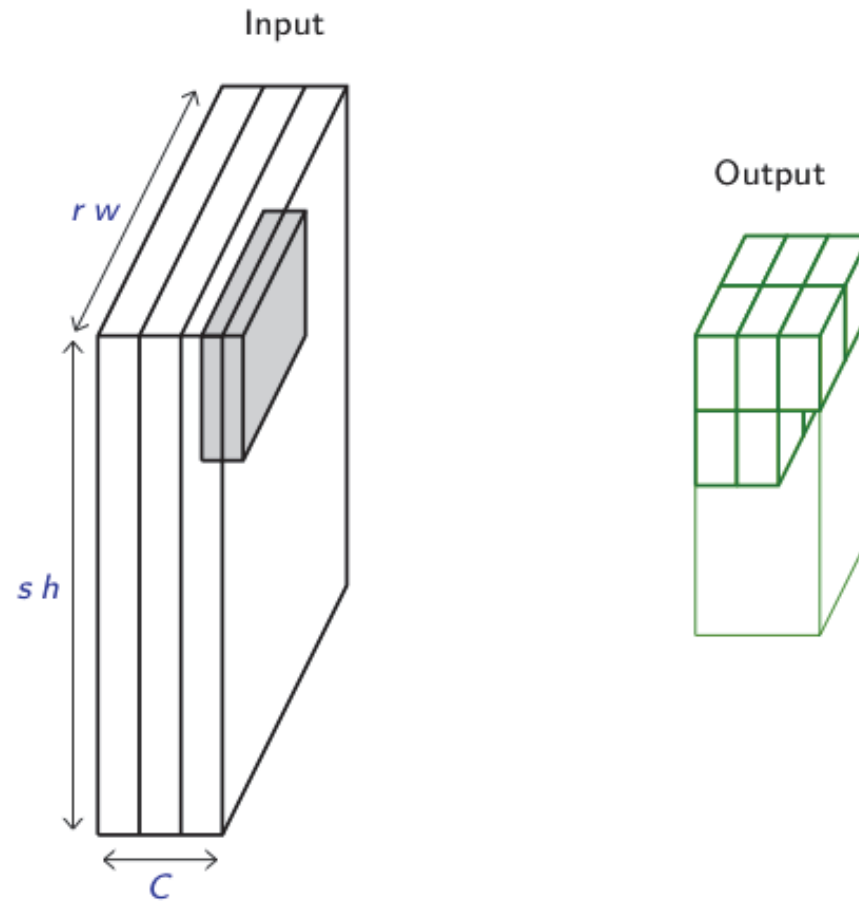
# Max-Pooling 2d



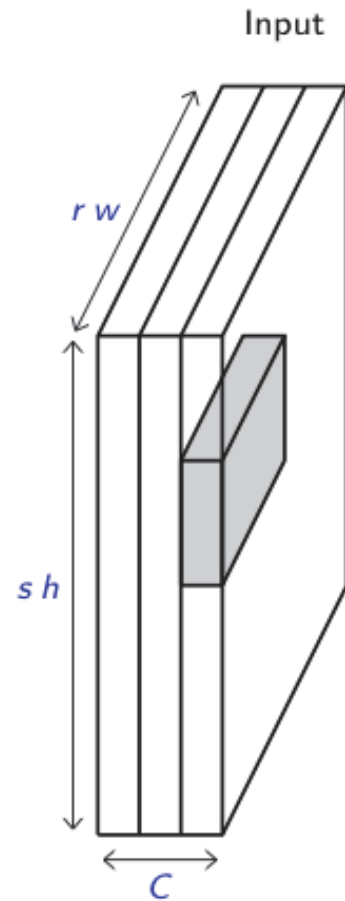
# Max-Pooling 2d



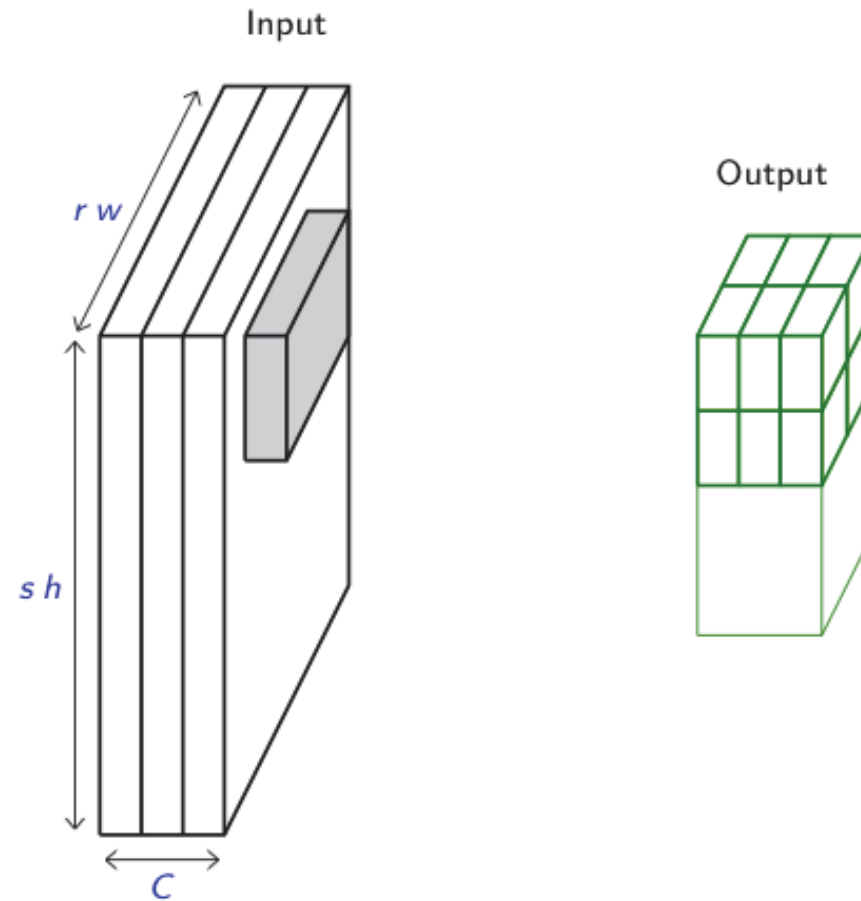
# Max-Pooling 2d



# Max-Pooling 2d

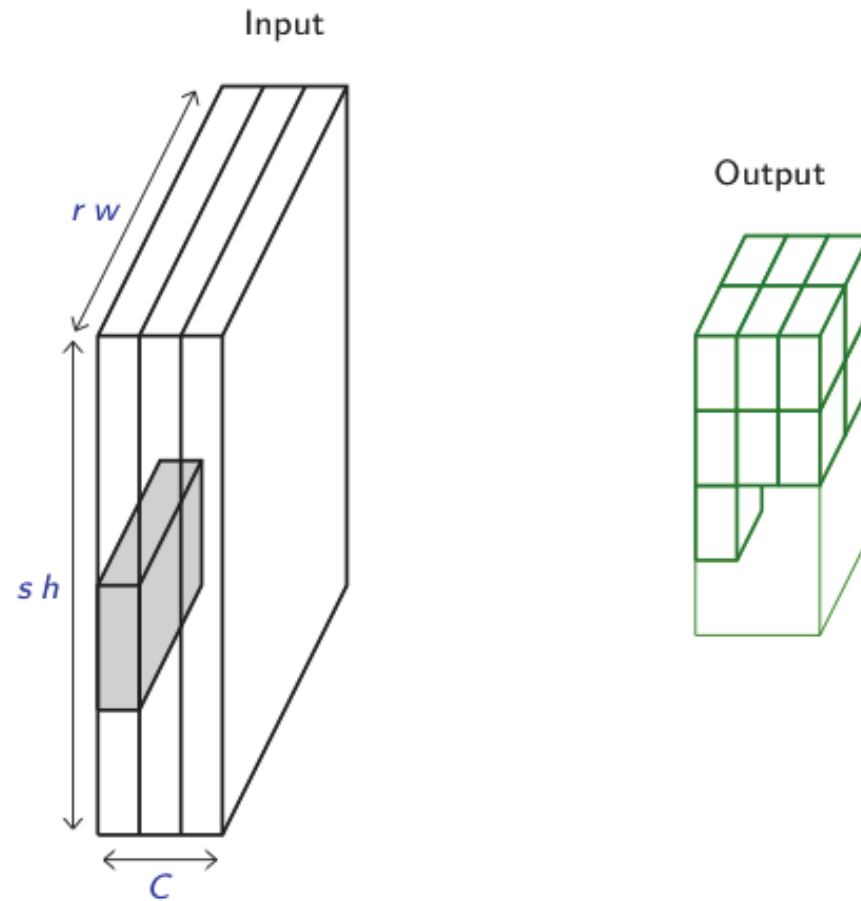


# Max-Pooling 2d

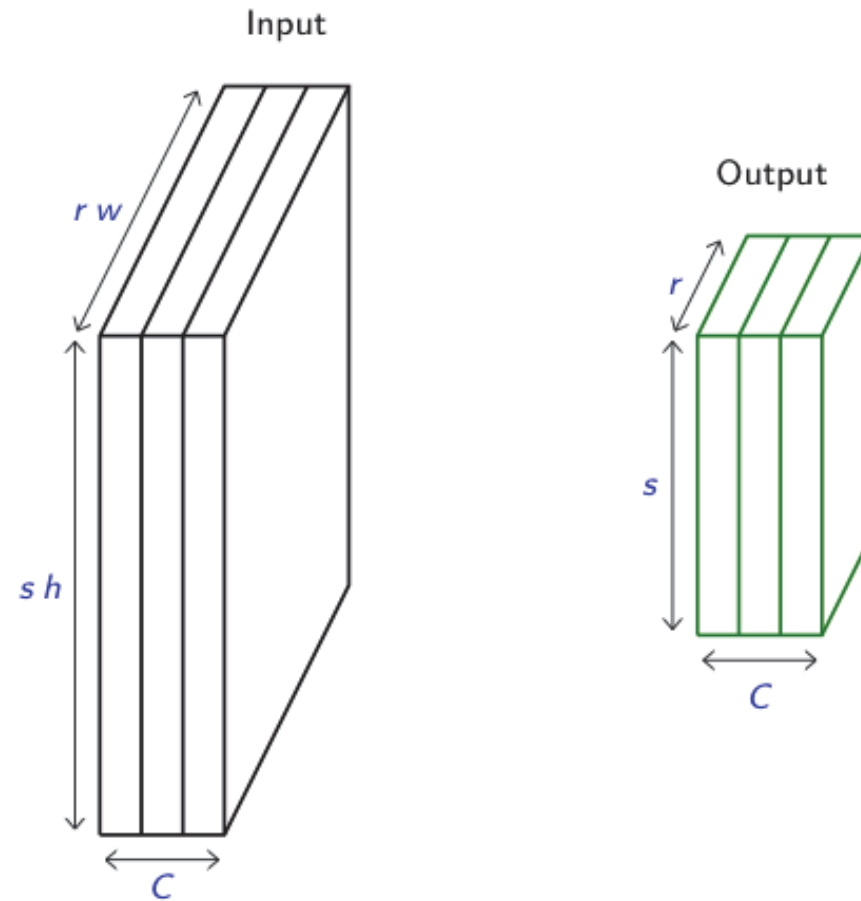




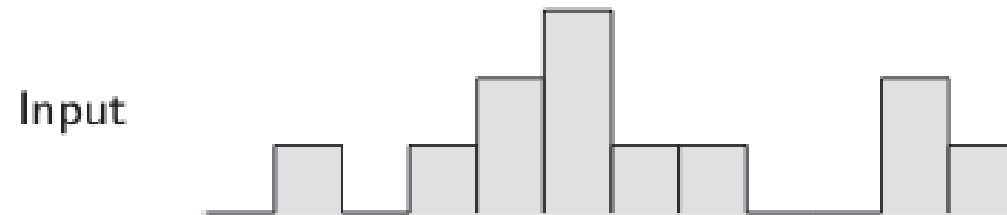
# Max-Pooling 2d



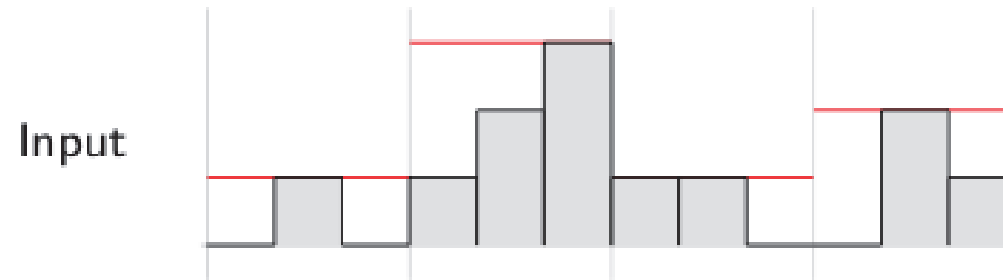
# Max-Pooling 2d



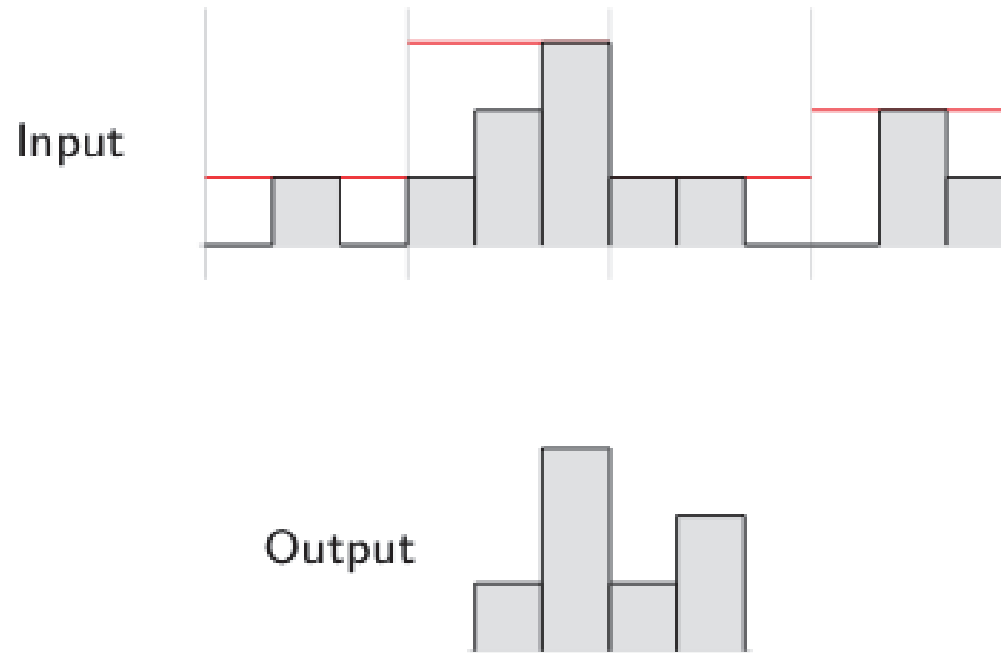
# Translation invariance from pooling



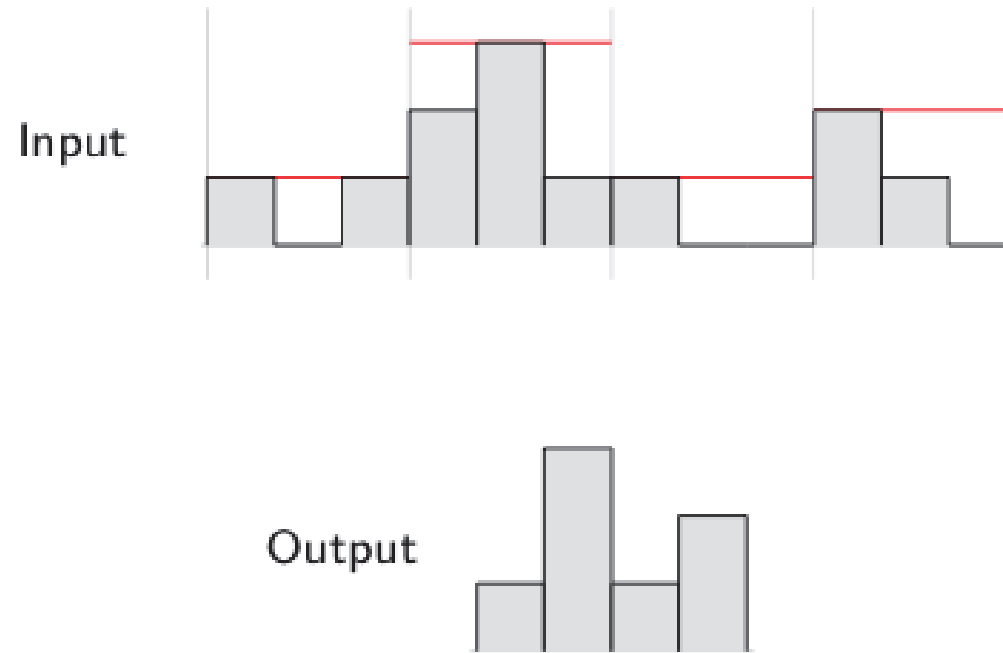
# Translation invariance from pooling



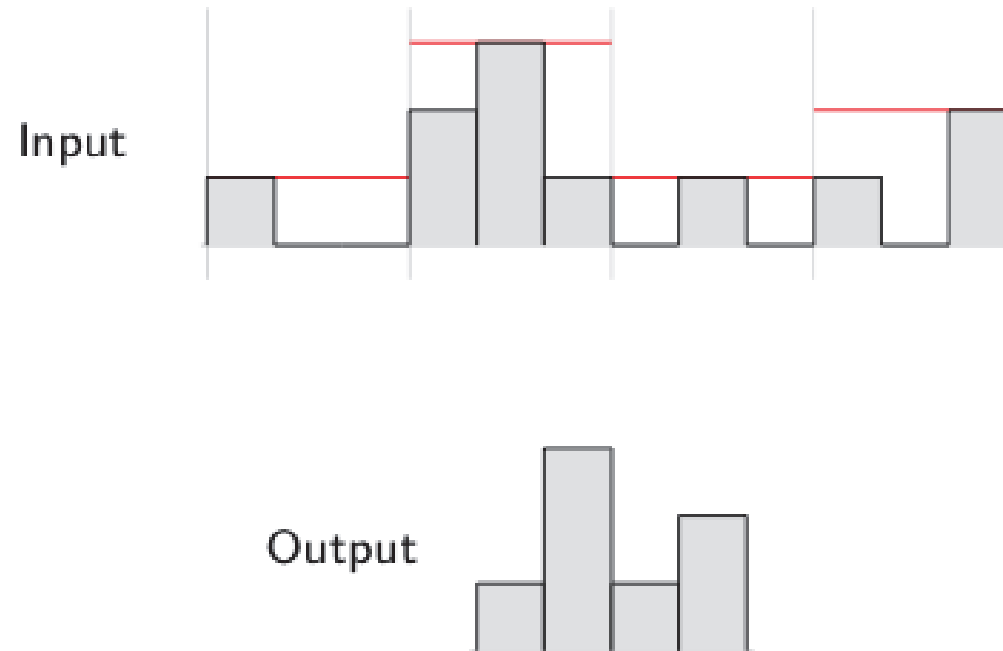
# Translation invariance from pooling



# Translation invariance from pooling

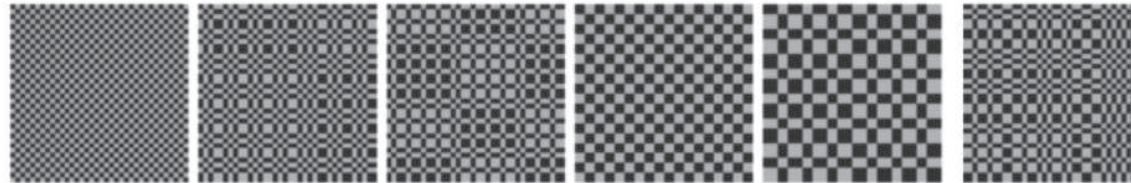


# Translation invariance from pooling



# Stochastic pooling

Random pooling mask at each pass





# Spectral pooling

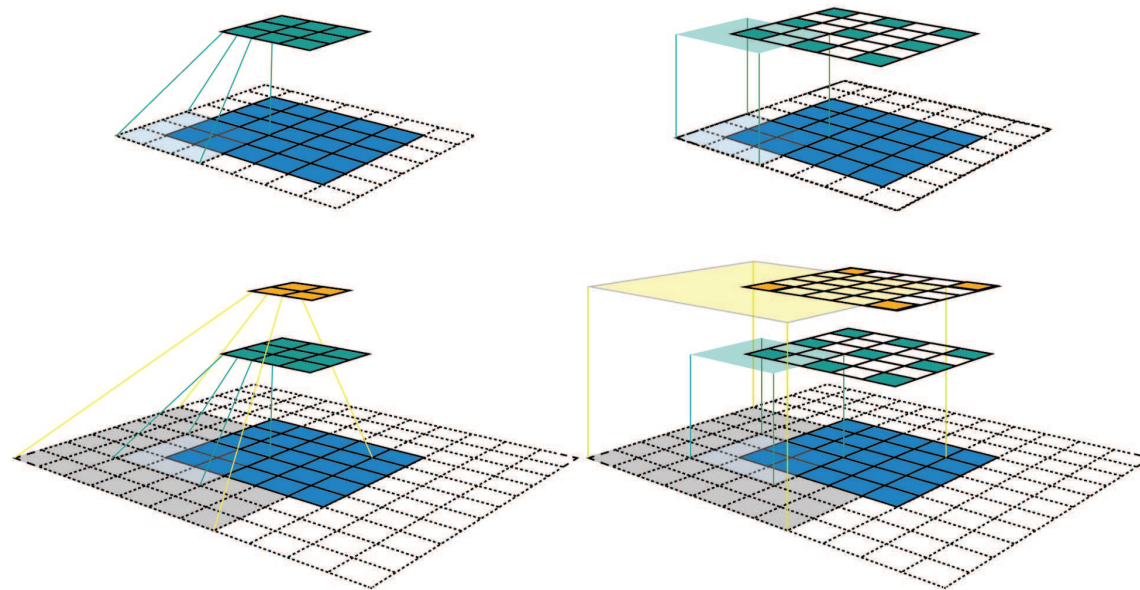
Pooling in the frequency domain



Figure 2: Approximations for different pooling schemes, for different factors of dimensionality reduction. Spectral pooling projects onto the Fourier basis and truncates it as desired. This retains significantly more information and permits the selection of any arbitrary output map dimensionality.

# Receptive field

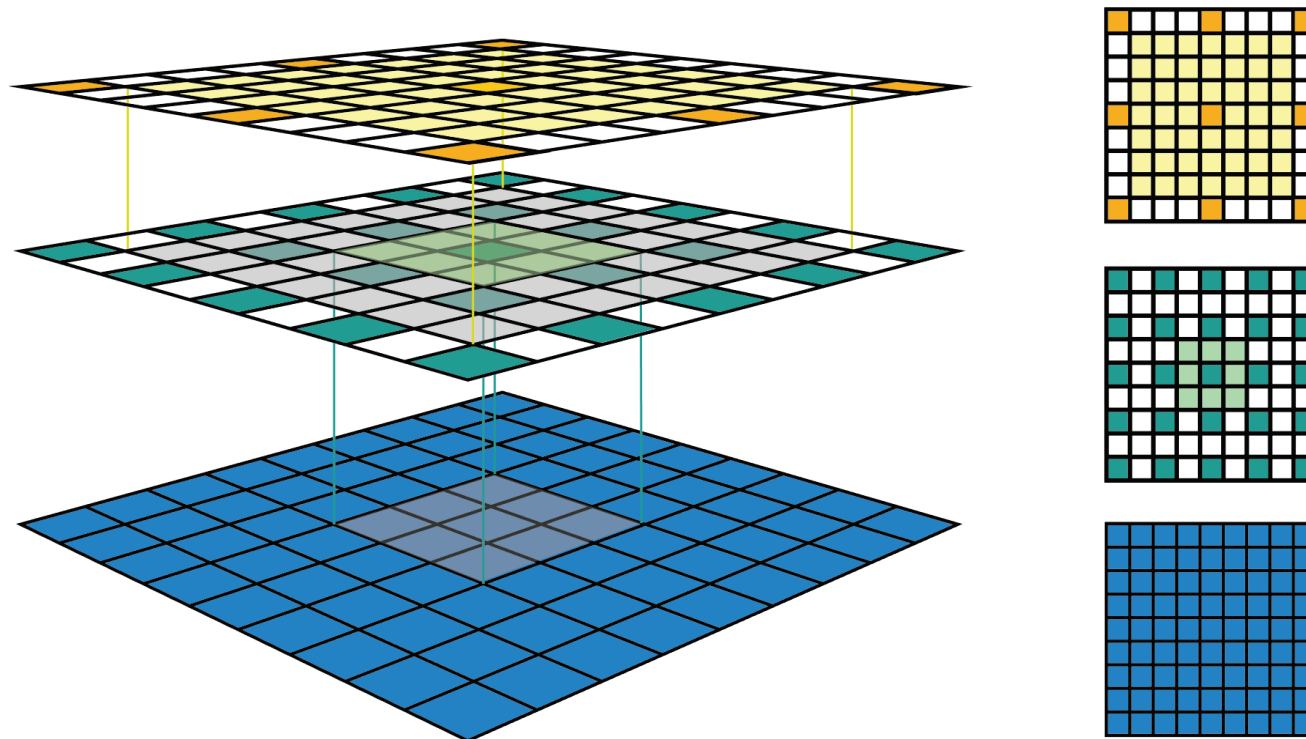
- The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).
- A receptive field of a feature can be fully described by its center location and its size
- Example:  $k = 3 \times 3$ ;  $p = 1 \times 1$ ;  $s = 2 \times 2$ ;  $input = 3 \times 3$



Common way to visualize a CNN feature map.

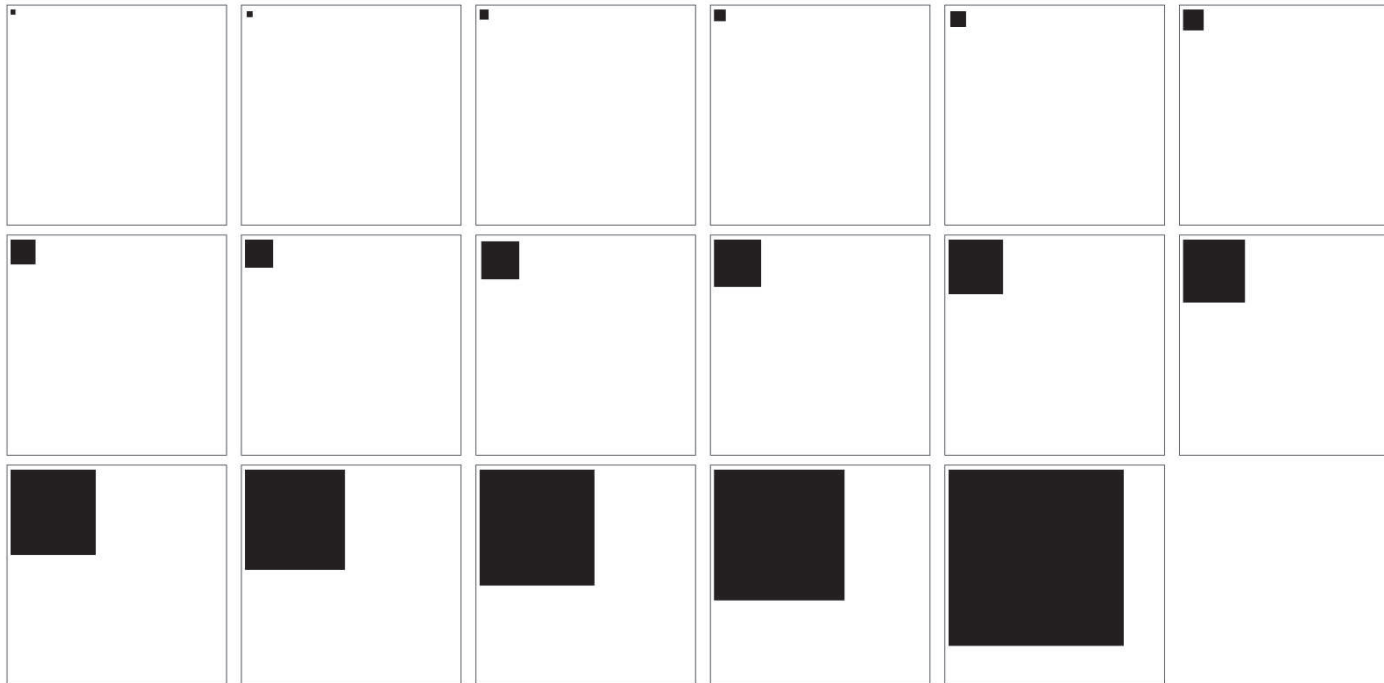
# Receptive field

- The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).
- A receptive field of a feature can be fully described by its center location and its size
- Example:  $k = 3 \times 3$ ;  $p = 1 \times 1$ ;  $s = 2 \times 2$ ;  $input = 7 \times 7$



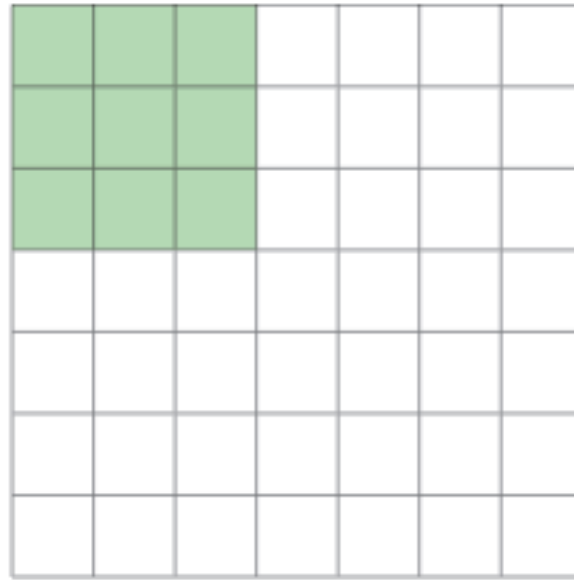
# Receptive field

- The receptive field is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).
- A receptive field of a feature can be fully described by its center location and its size

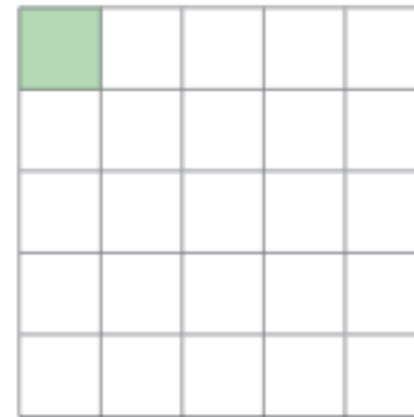


Receptive fields for convolutional and pooling layers of VGG-16

# Dilated convolutions

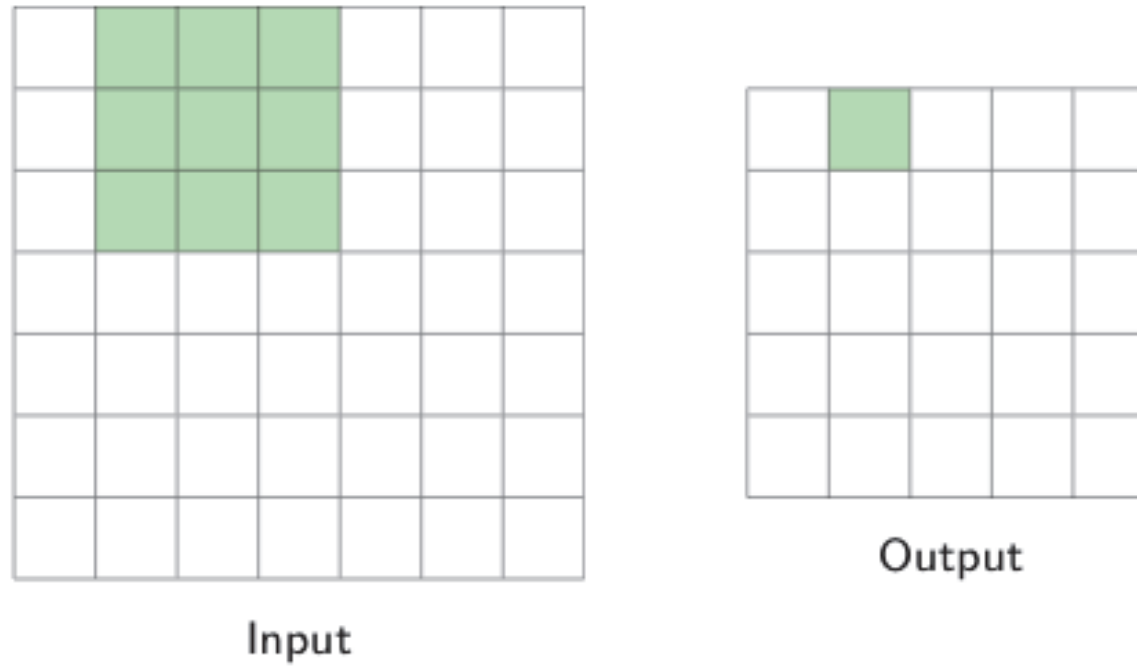


Input

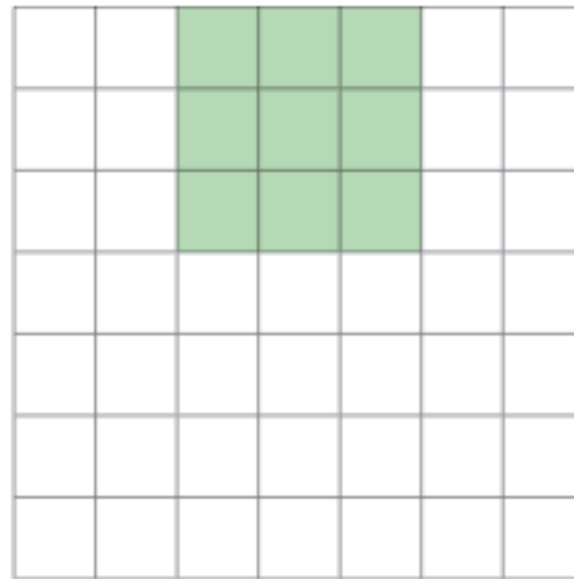


Output

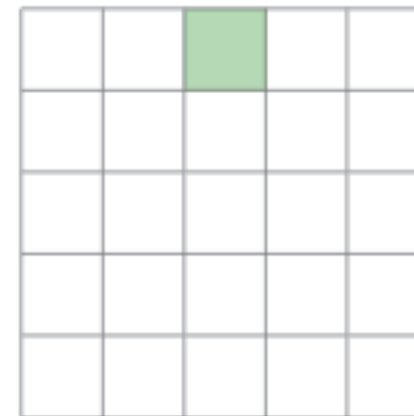
# Dilated convolutions



# Dilated convolutions

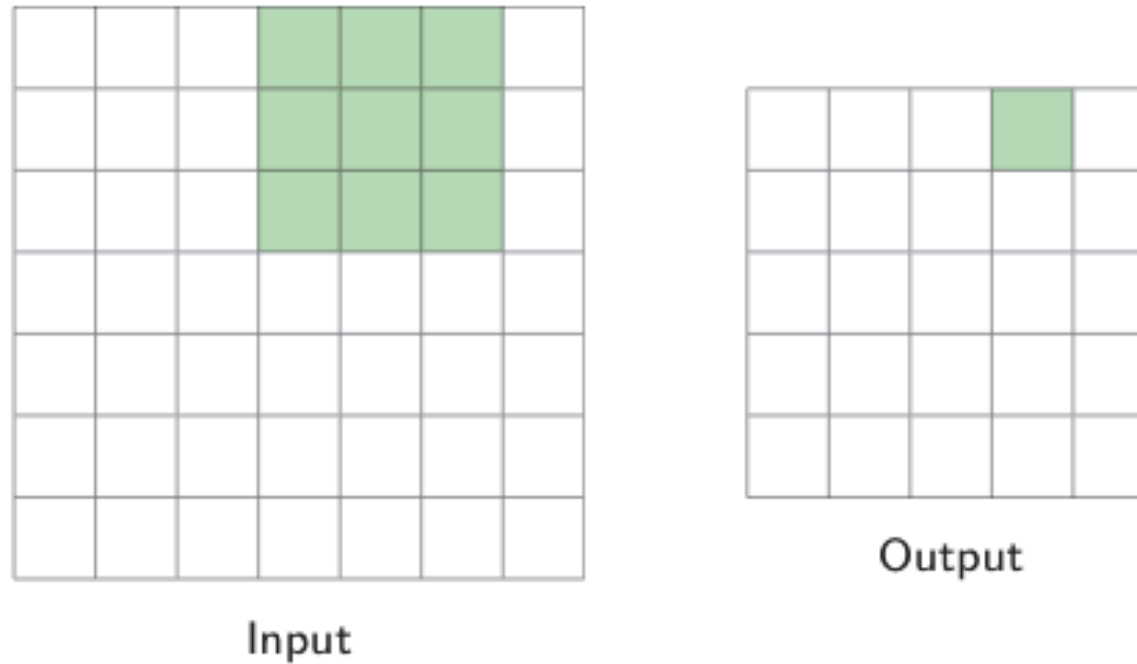


Input



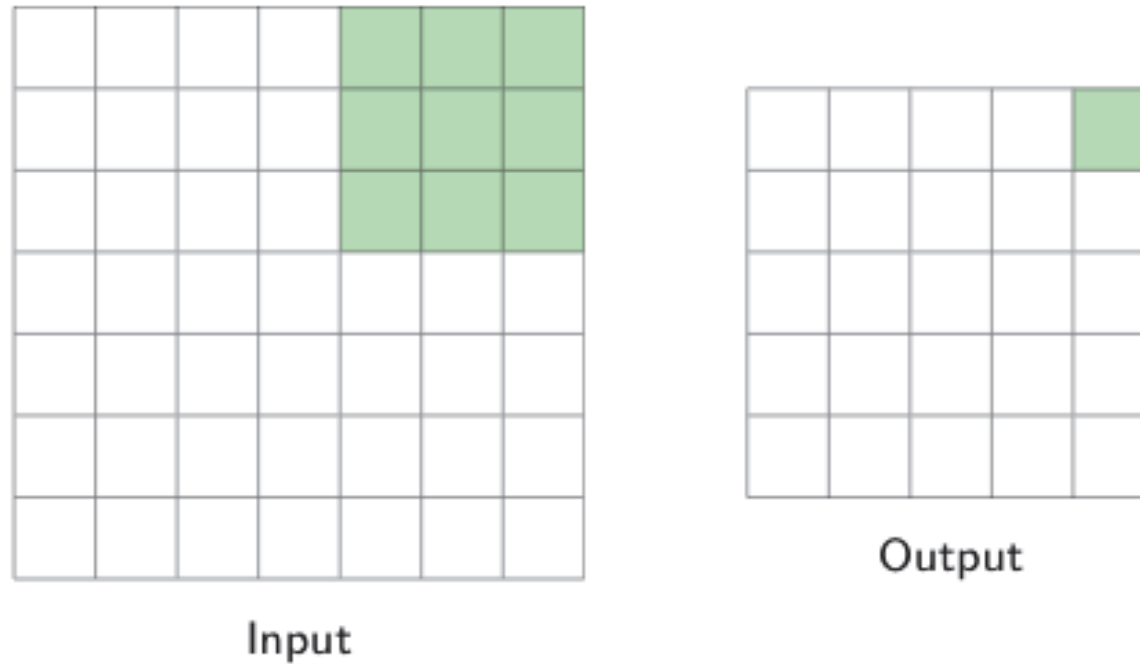
Output

# Dilated convolutions

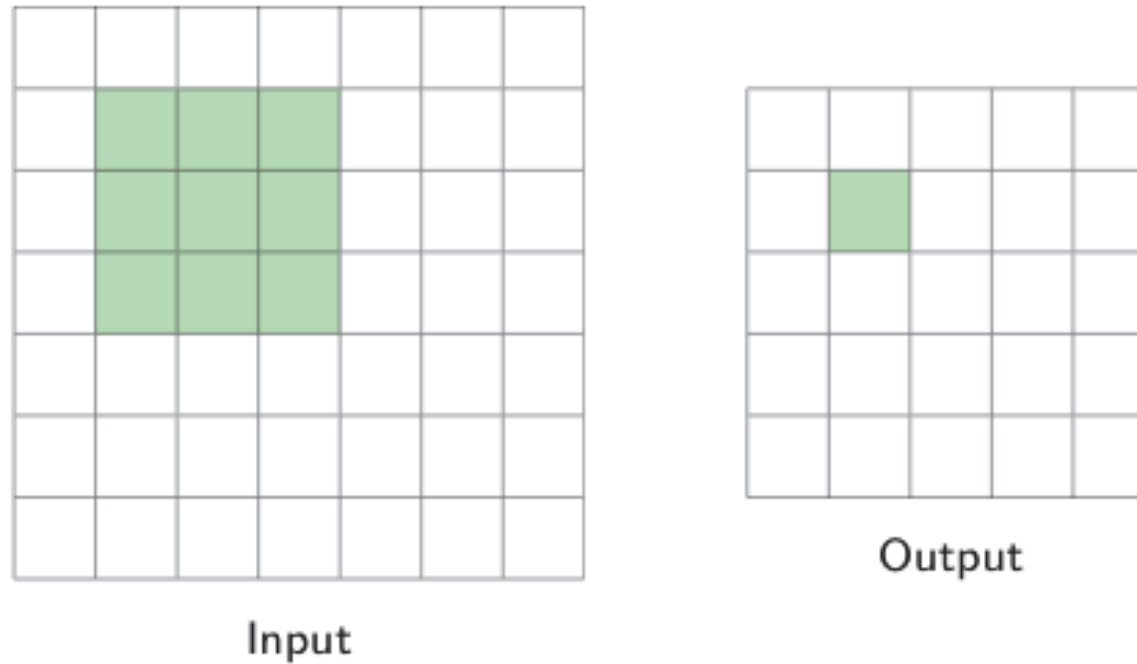




# Dilated convolutions



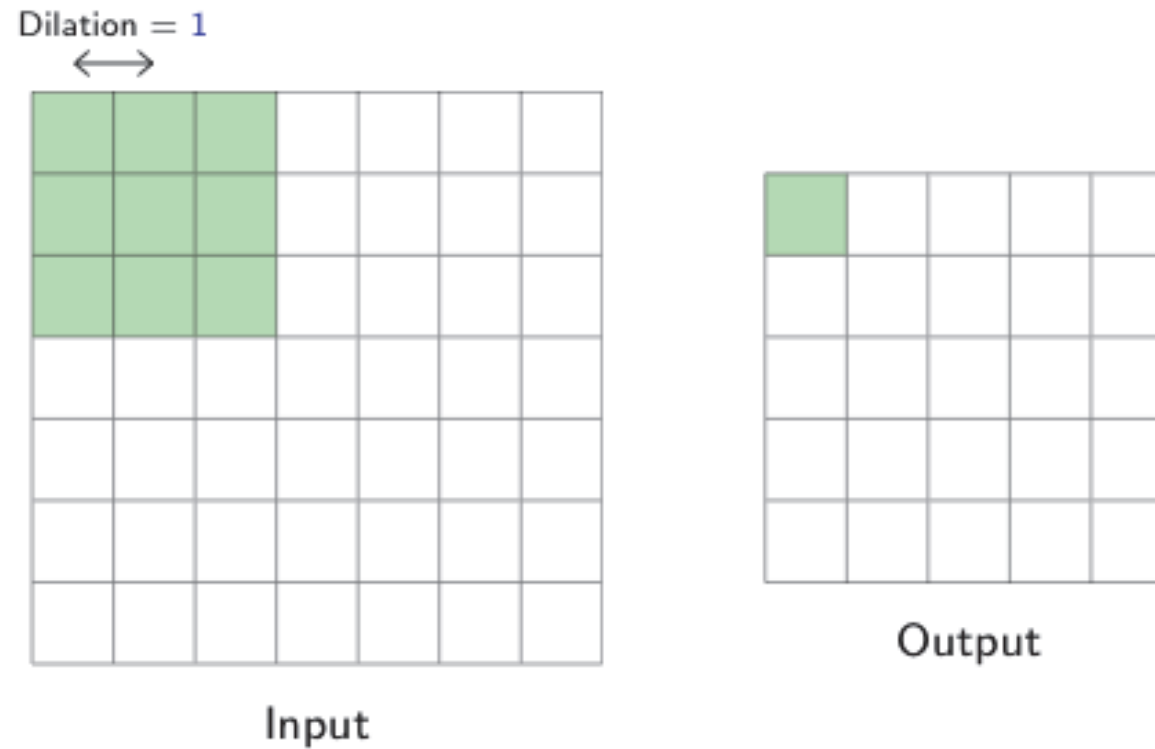
# Dilated convolutions



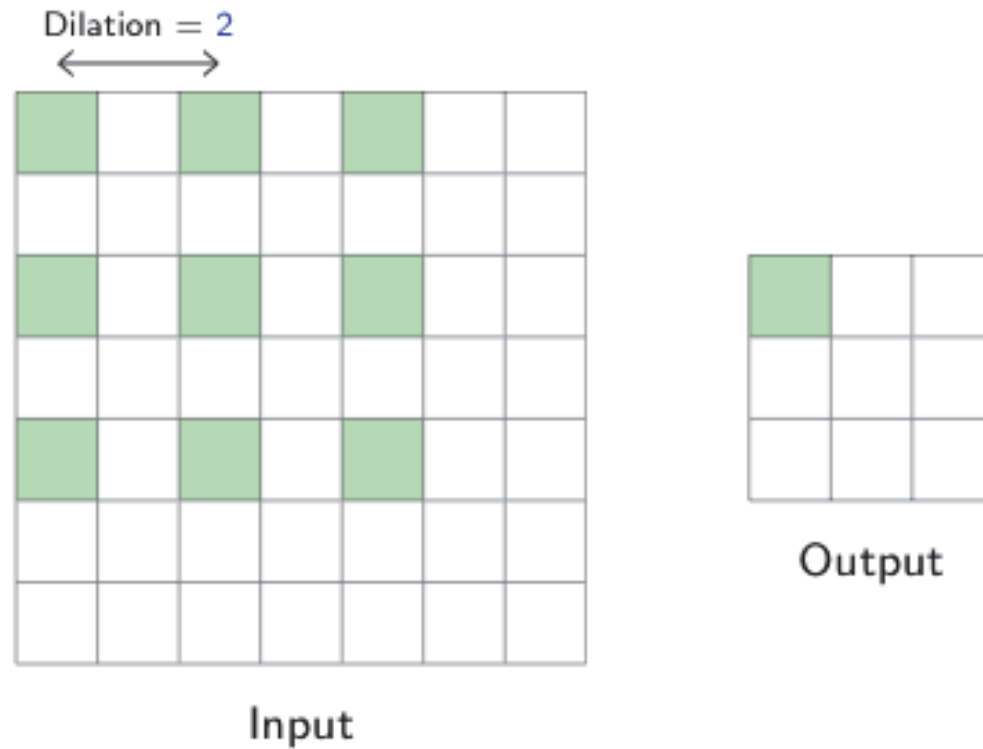
# Dilated convolutions

- Can we do better?
- ... Without adding parameters?

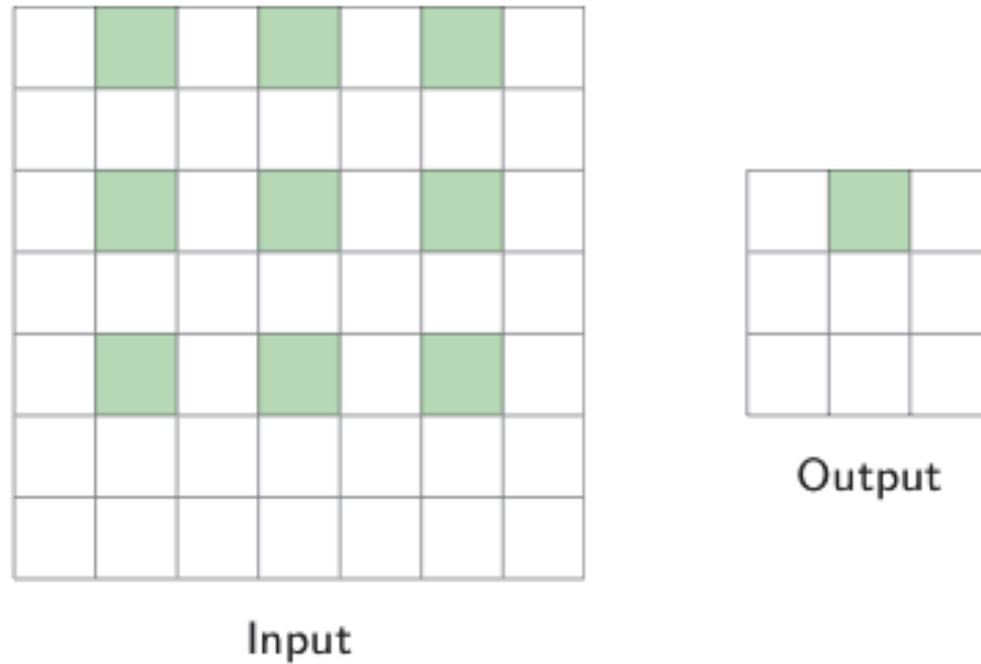
# Dilated convolutions



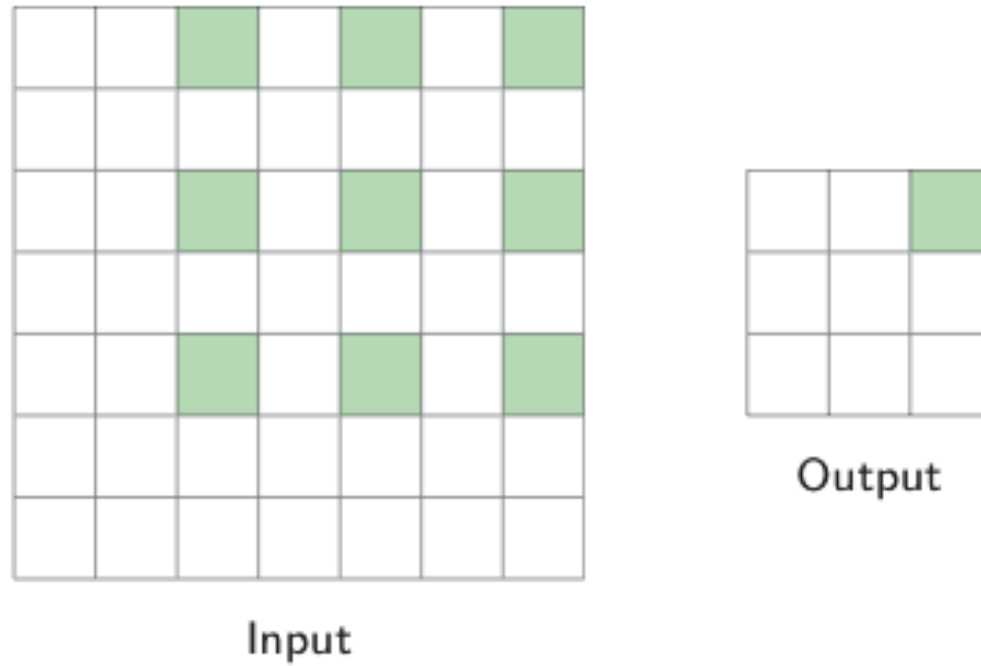
# Dilated convolutions



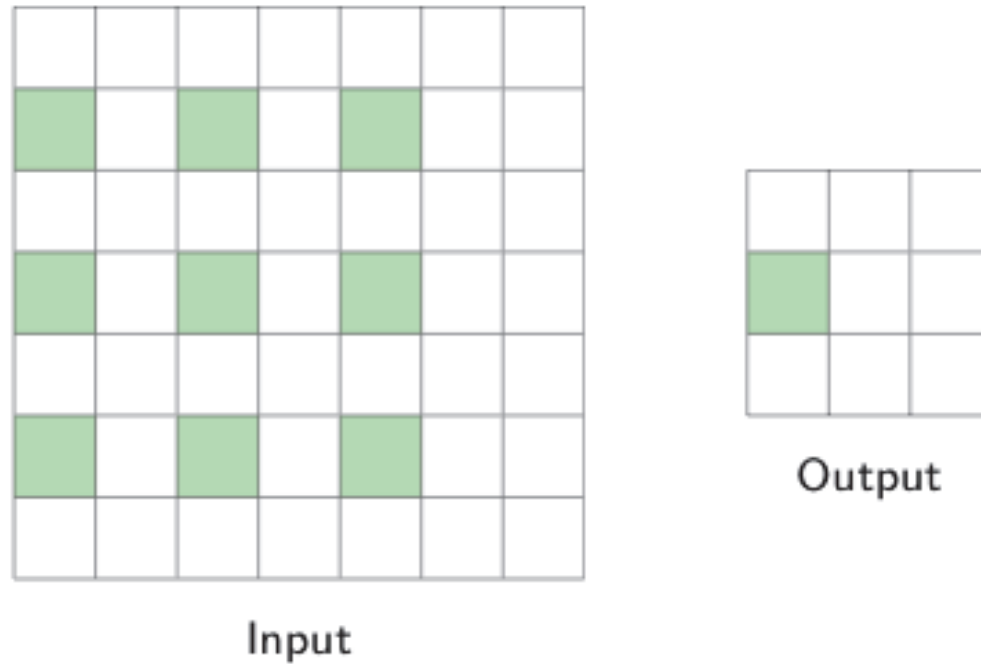
# Dilated convolutions



# Dilated convolutions

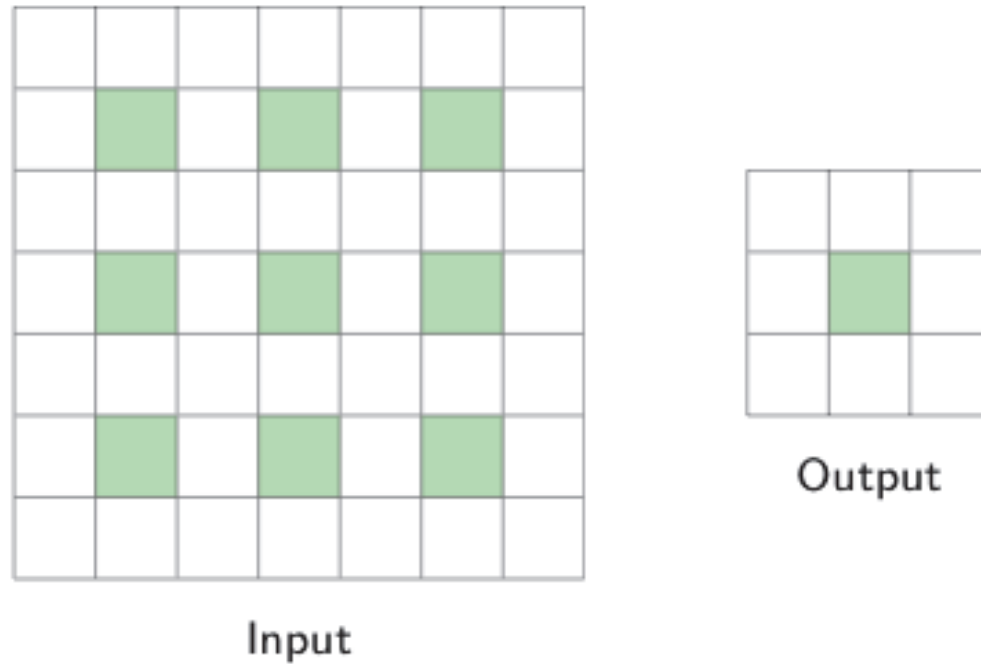


# Dilated convolutions

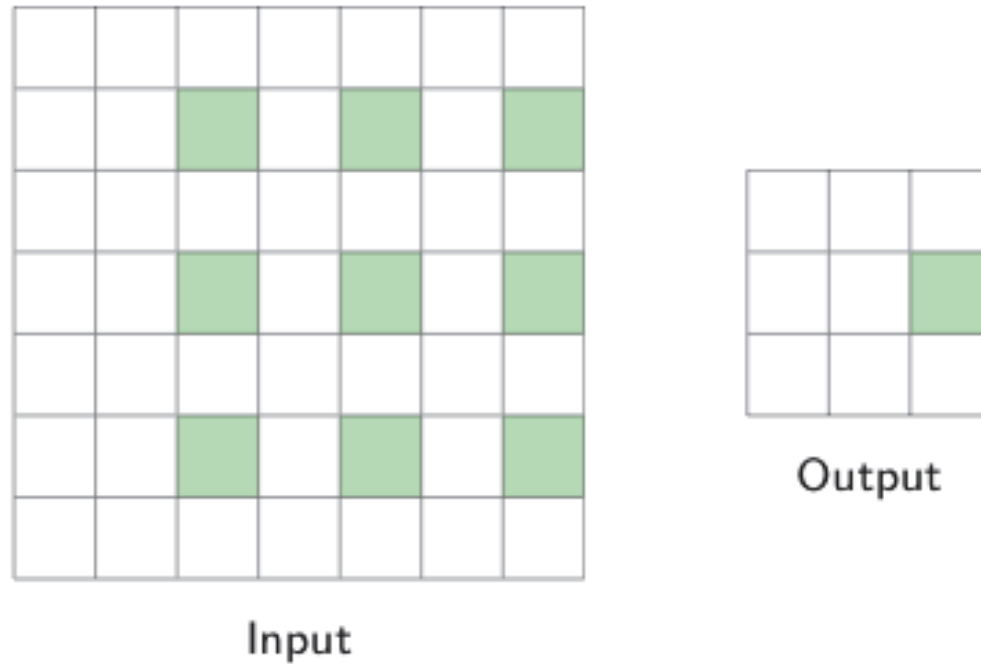




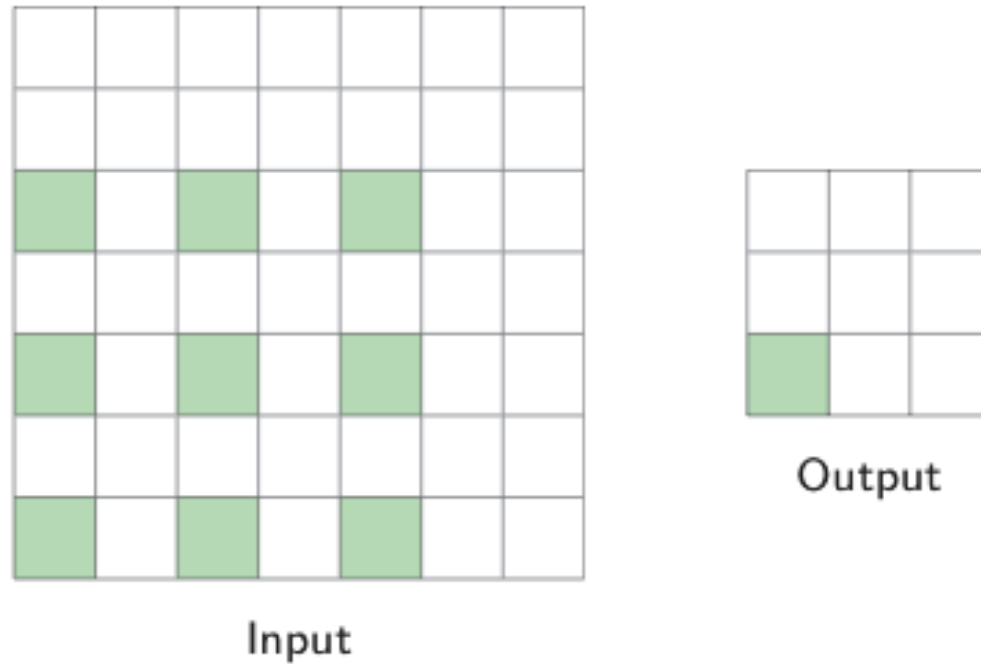
# Dilated convolutions



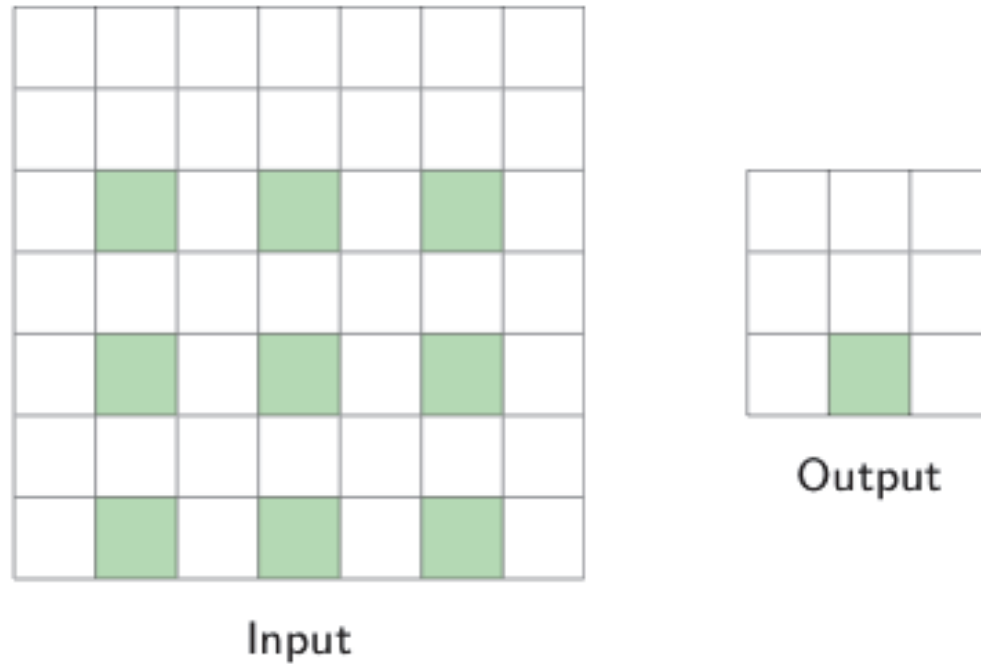
# Dilated convolutions



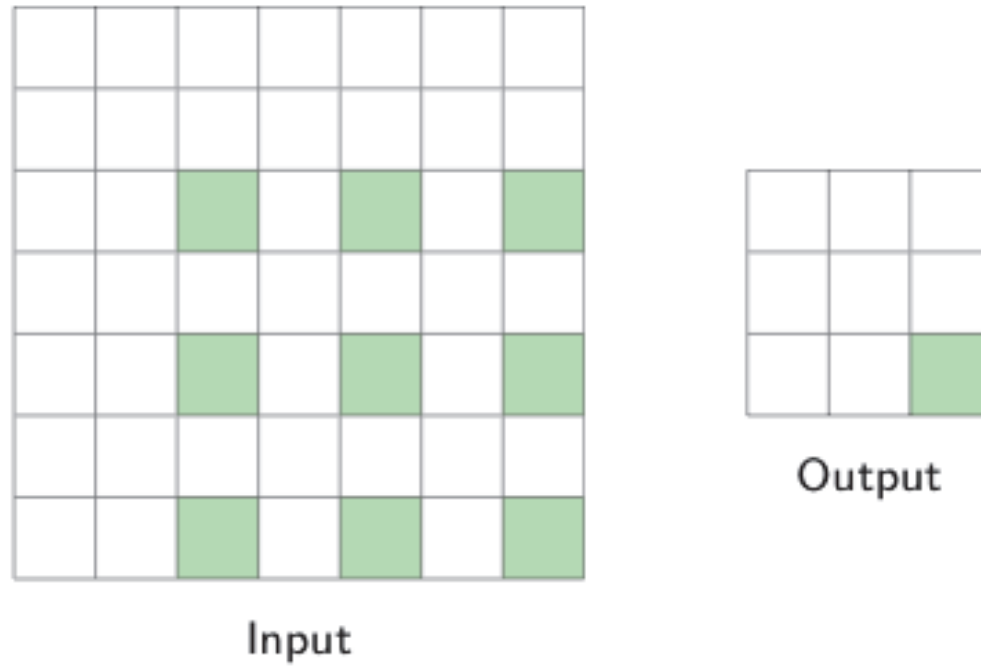
# Dilated convolutions



# Dilated convolutions

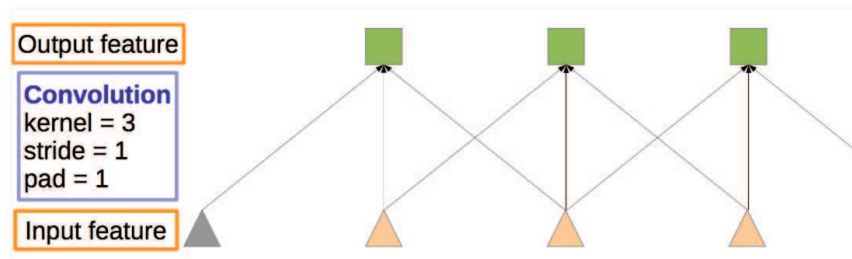


# Dilated convolutions

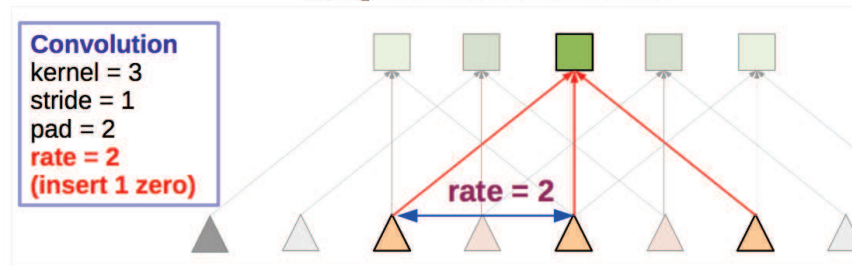


# Dilated convolutions

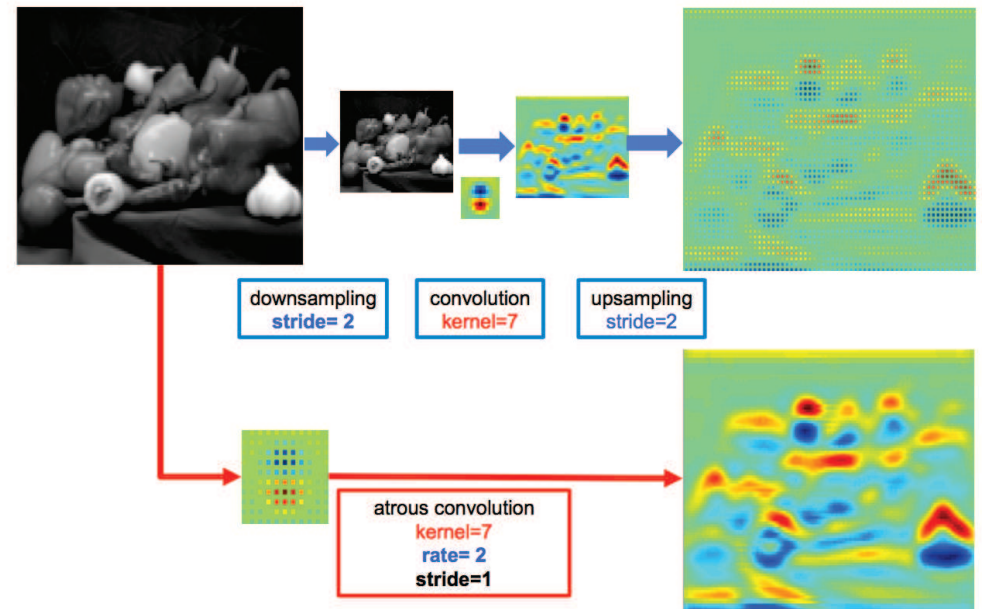
- also goes by the name **convolutions à trous**



(a) Sparse feature extraction

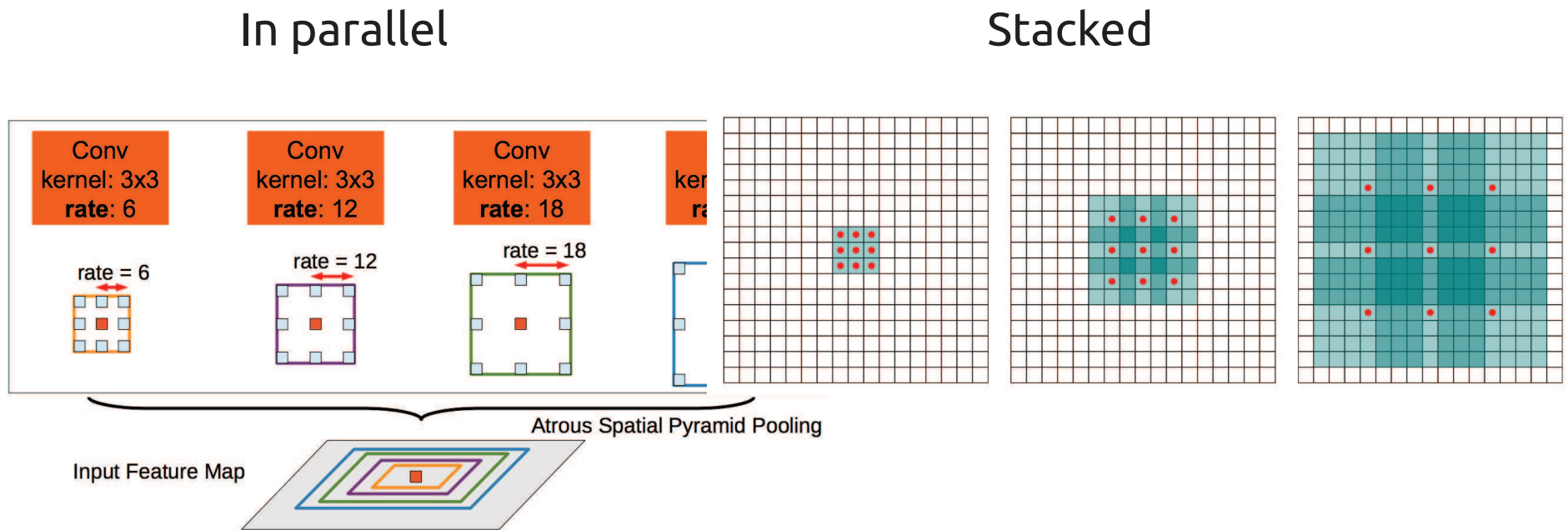


(b) Dense feature extraction



# Dilated convolutions

## Usage



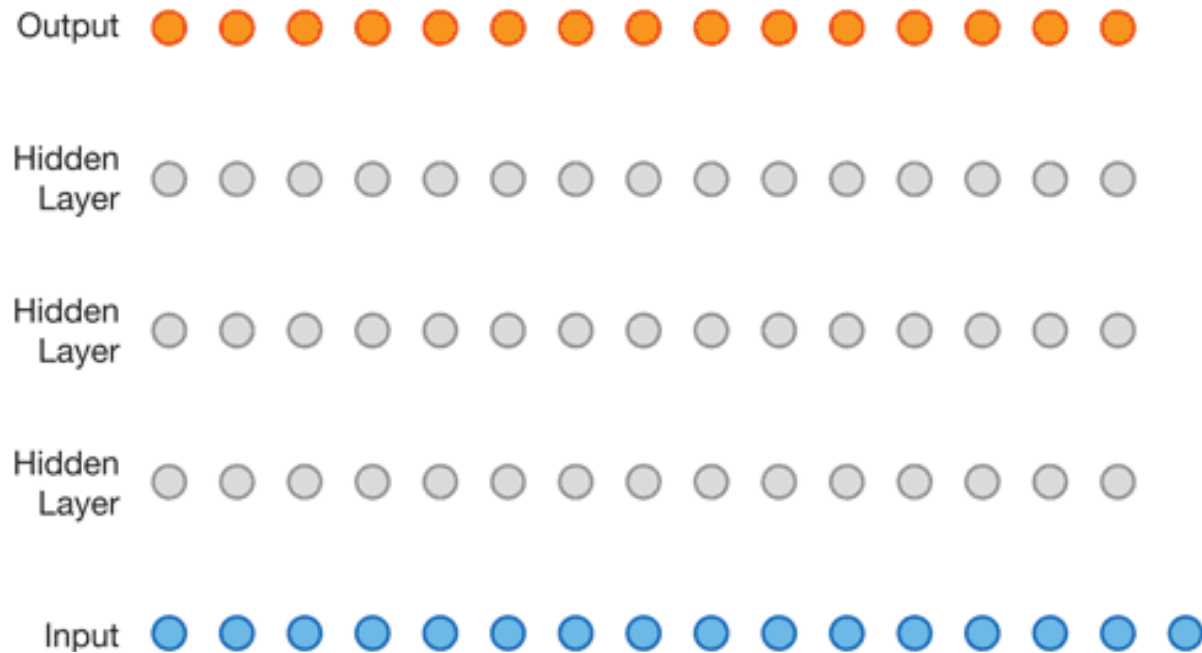
More frequently used

*DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs; Chen et al., PAMI 2016*

*Multi-scale context aggregation by dilated convolutions; Yu and Koltun, ICLR 2016*

# Dilated convolutions

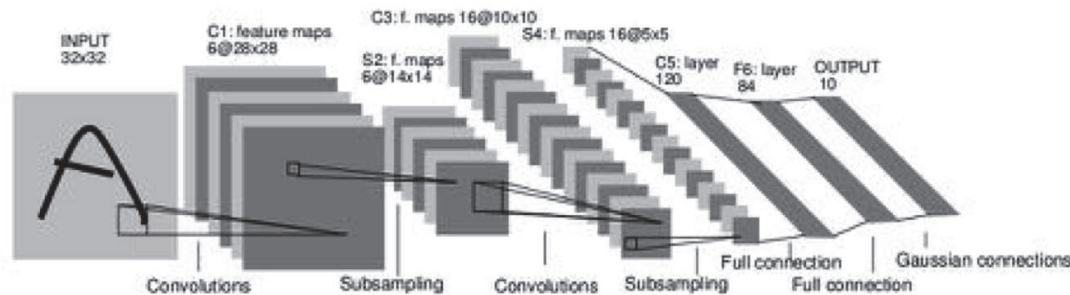
- works for 1d as well
- appealing alternative to recurrent neural networks





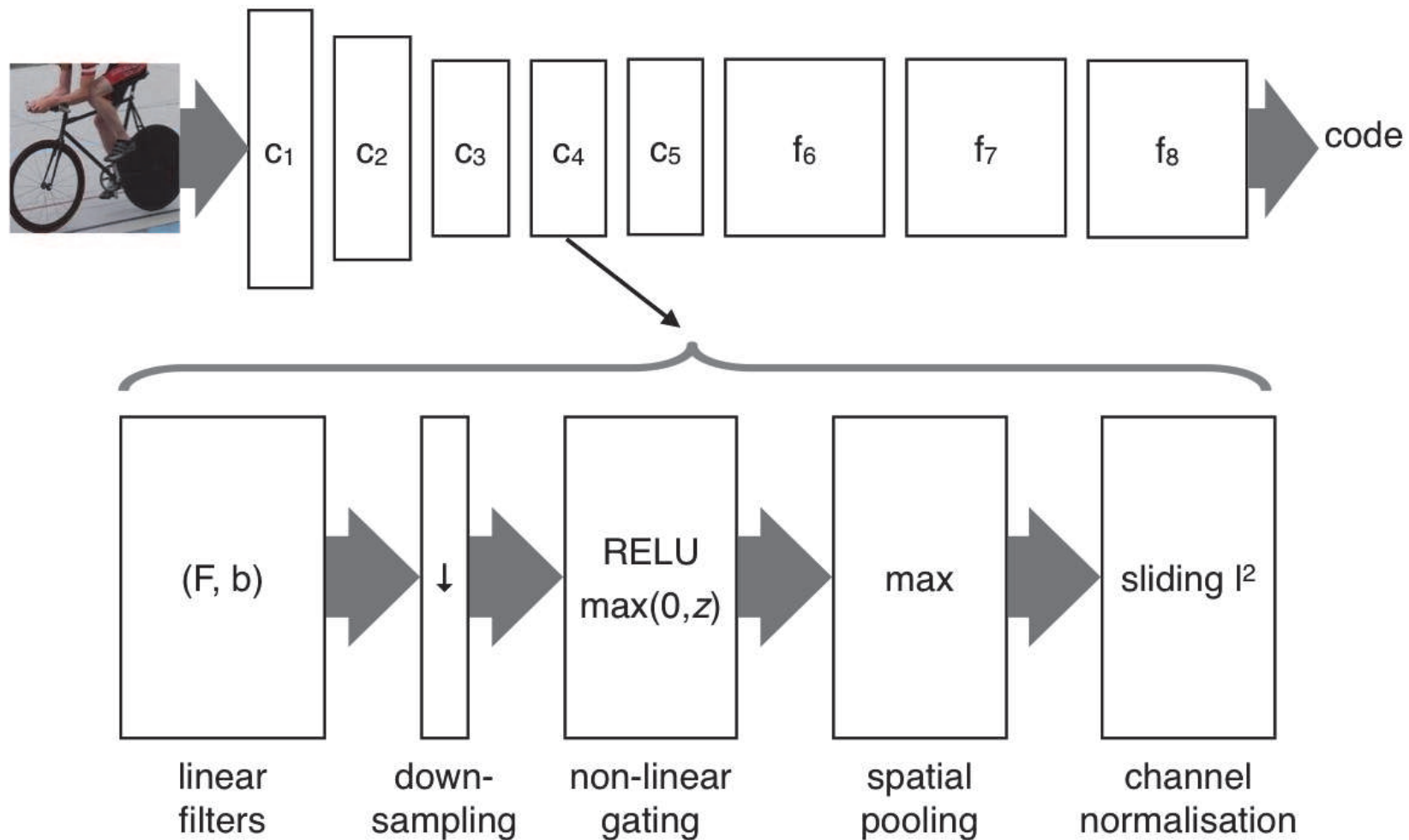
# ConvNet

- Neural network with specialized connectivity structure
- Stack multiple stage of feature extractors
- Higher stages compute more global, more invariant features
- Classification layer at the end



# ConvNet

A convolutional layer is composed of convolution, activation and downsampling layers.



# ConvNet

Input

# ConvNet

## Input

## Conv blocks

- Convolution + activation (relu)
- Convolution + activation (relu)
- ...
- Maxpooling 2x2

# ConvNet

## Input

## Conv blocks

- Convolution + activation (relu)
- Convolution + activation (relu)
- ...
- Maxpooling 2x2

## Output

- Fully connected layers
- Softmax

# Motivations

## Local connectivity

- A neuron depends only on a few local neurons
- Translation invariance

# Motivations

## Local connectivity

- A neuron depends only on a few local neurons
- Translation invariance

## Comparison to Fully connected

- Parameter sharing
- Make use of spatial structure

# Motivations

## Local connectivity

- A neuron depends only on a few local neurons
- Translation invariance

## Comparison to Fully connected

- Parameter sharing
- Make use of spatial structure

## Some analogy to animal vision

Hubel & Wiesel, RECEPTIVE FIELDS OF SINGLE NEURONES IN THE CAT'S STRIATE CORTEX (1959)



Going deeper

For a fixed parameter budget deeper is better

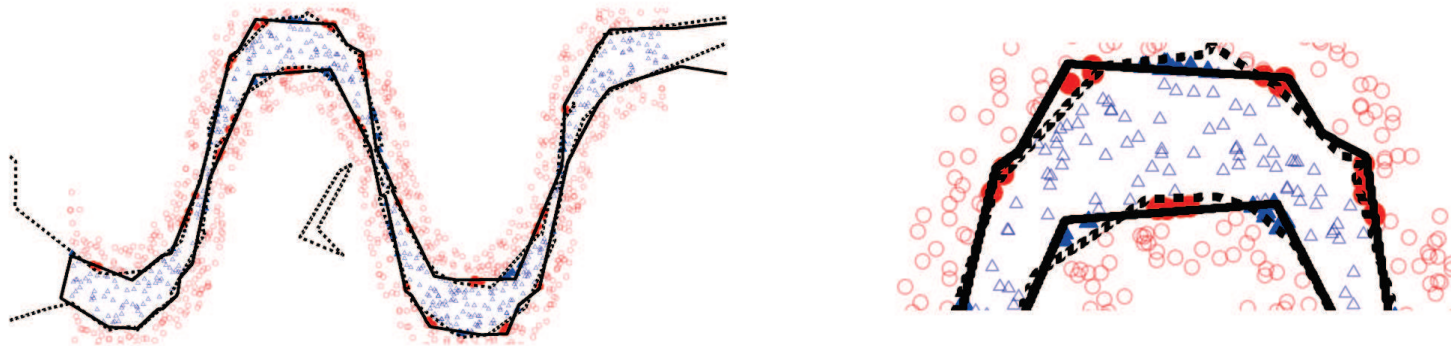
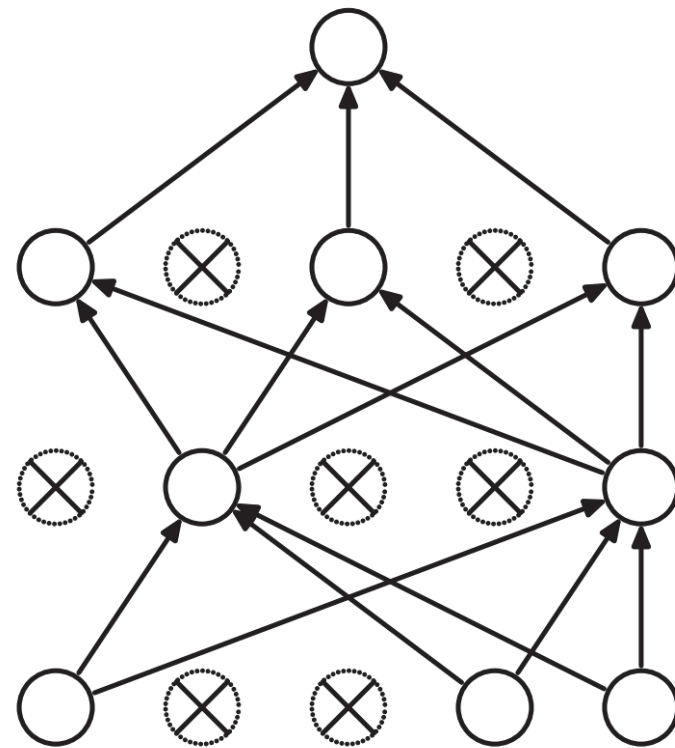
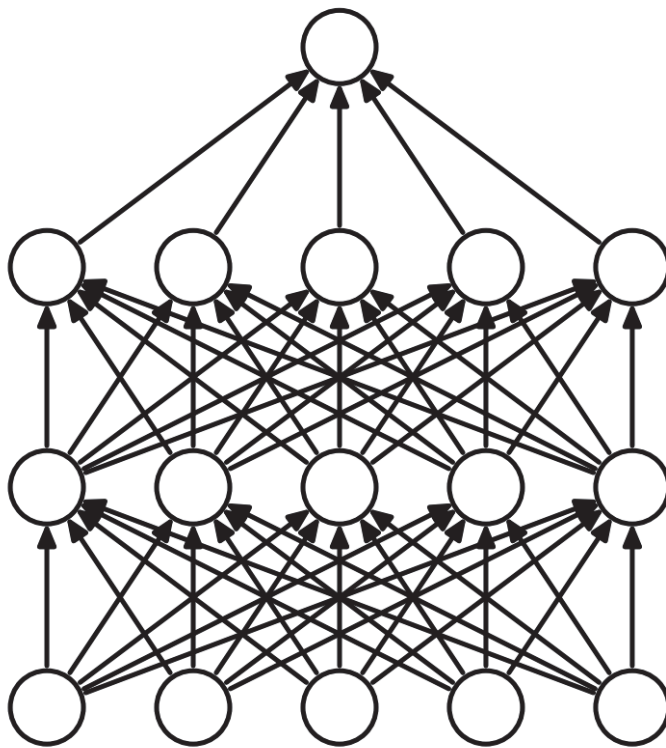


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

# Dropout

- First "deep" regularization technique
- Remove units at random during the forward pass on each sample
- Put them all back during test



(a) Standard Neural Net (b) After applying dropout.  
*Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014*

# Dropout

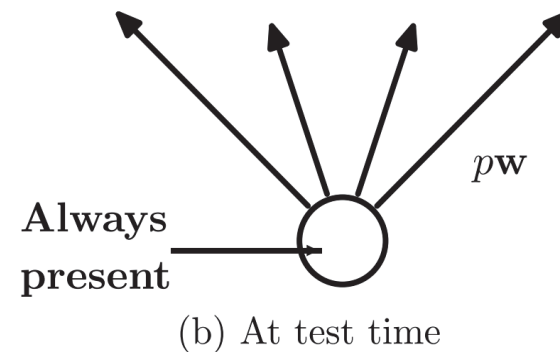
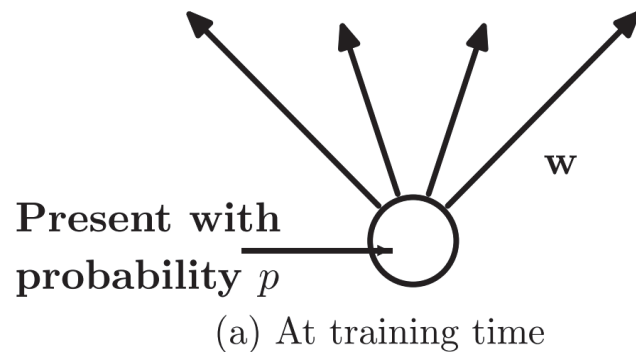
## Interpretation

- Reduces the network dependency to individual neurons and distributes representation
- More redundant representation of data

## Ensemble interpretation

- Equivalent to training a large ensemble of shared-parameters, binary-masked models
- Each model is only trained on a single data point
- A network with dropout can be interpreted as an ensemble of  $2^N$  models with heavy weight sharing (Goodfellow et al., 2013)

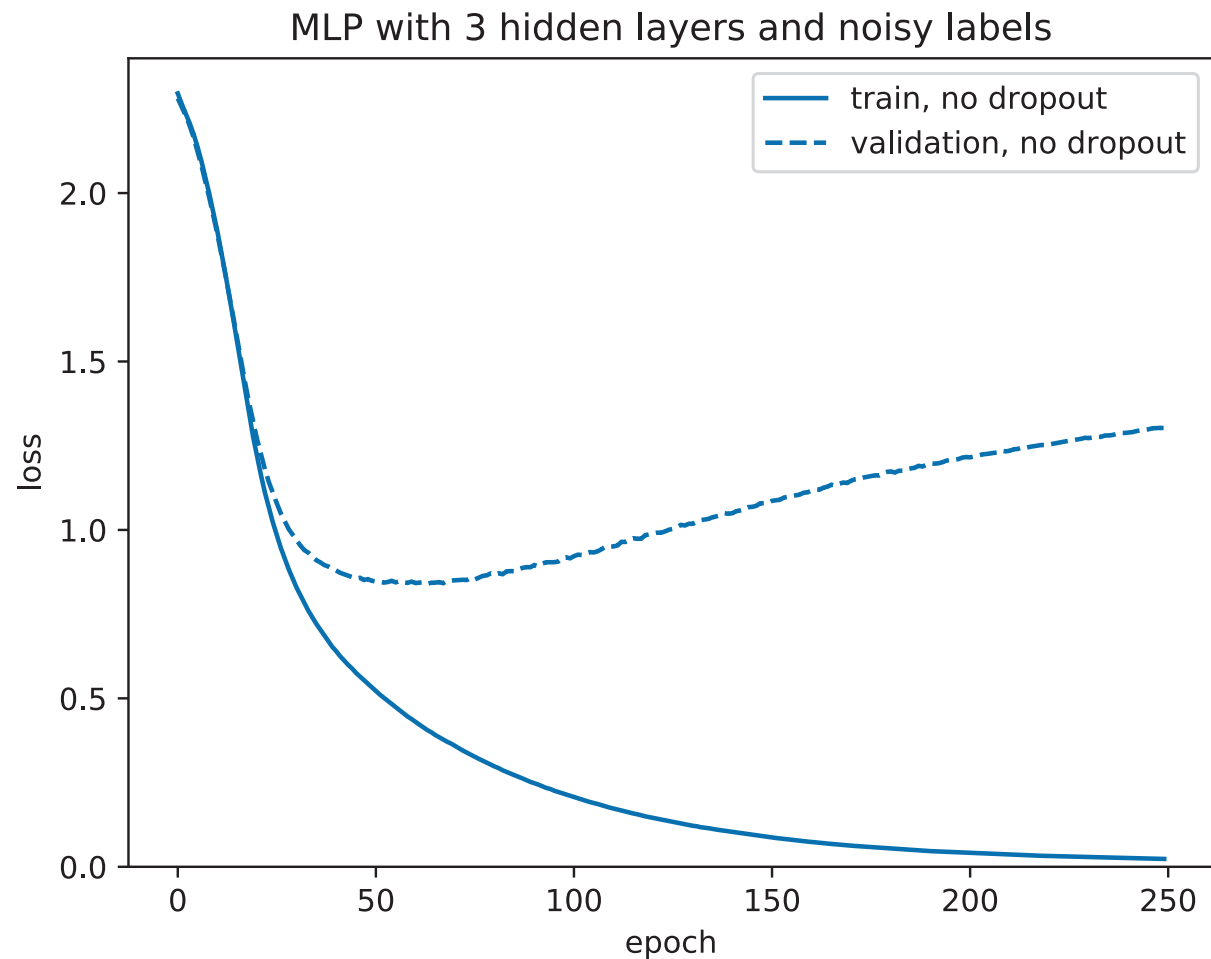
# Dropout



- One has to decide on which units/layers to use dropout, and with what probability  $p$  units are dropped.
- During training, for each sample, as many Bernoulli variables as units are sampled independently to select units to remove.
- To keep the means of the inputs to layers unchanged, the initial version of dropout was multiplying activations by  $p$  during test.
- The standard variant is the "inverted dropout": multiply activations by  $\frac{1}{1-p}$  during training and keep the network untouched during test.

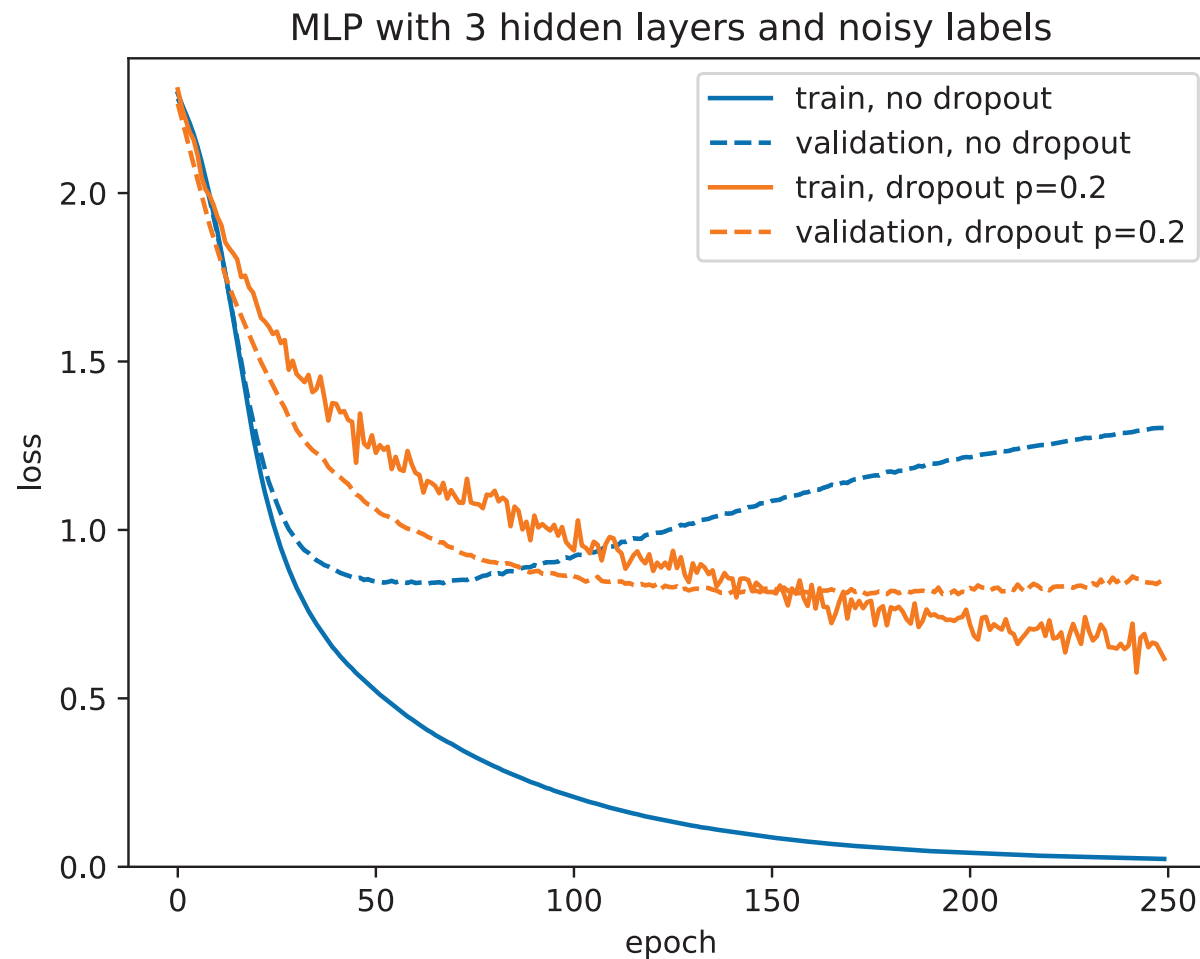
# Dropout

## Overfitting noise



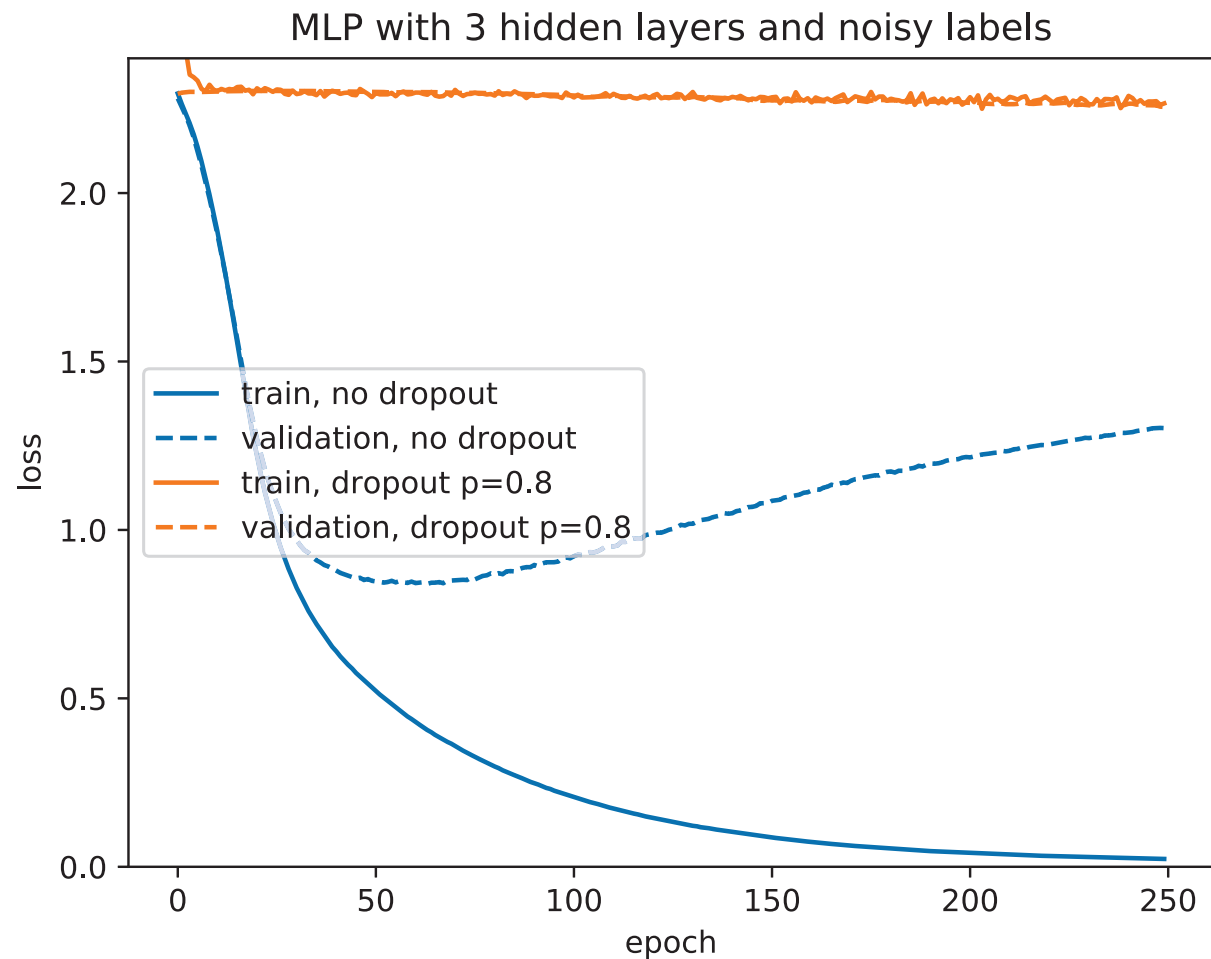
# Dropout

## A bit of Dropout



# Dropout

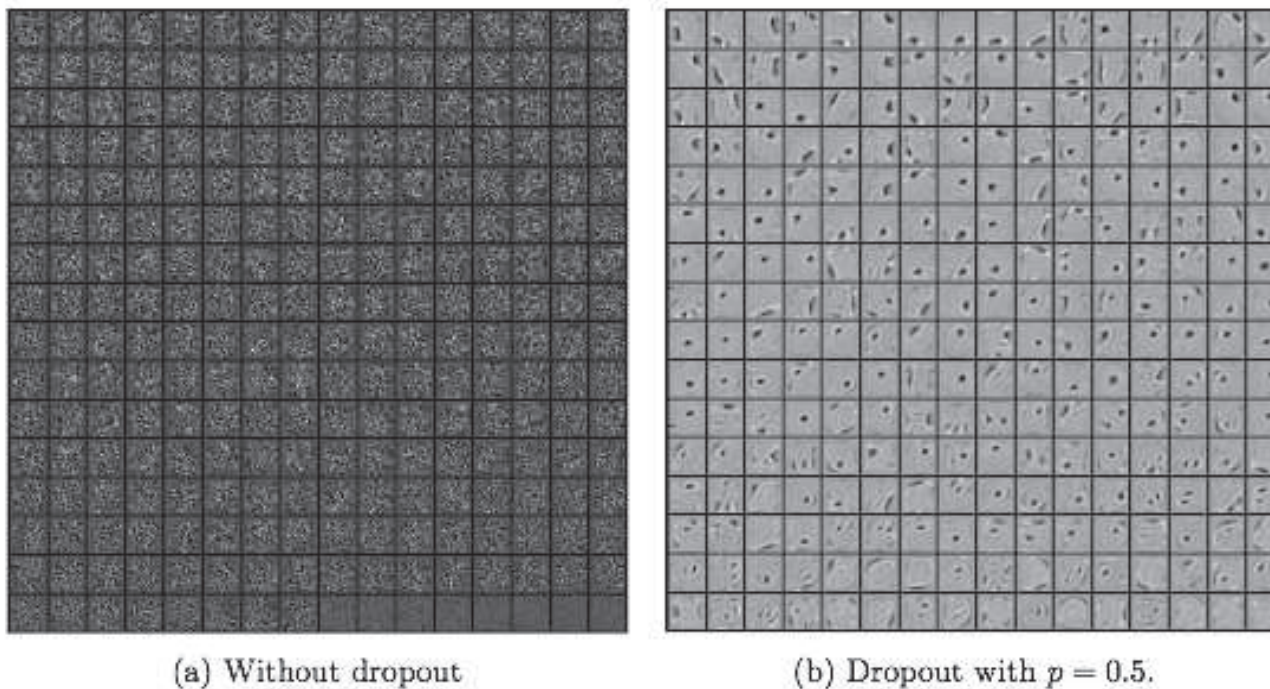
Too much: underfitting





# Dropout

Features learned on MNIST with one hidden layer autoencoders having 256 rectified linear units



*Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al., JMLR 2014*

# Dropout

```
>>> x = Variable(torch.Tensor(3, 9).fill_(1.0), requires_grad = True)
>>> x.data
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
[torch.FloatTensor of size 3x9]

>>> dropout = nn.Dropout(p = 0.75)
>>> y = dropout(x)
>>> y.data
4 0 4 4 4 0 4 0 0
4 0 0 0 0 0 0 0 0
0 0 0 0 4 0 4 0 4
[torch.FloatTensor of size 3x9]

>>> l = y.norm(2, 1).sum()
>>> l.backward()
>>> x.grad.data
1.7889 0.0000 1.7889 1.7889 0.0000 0.0000 1.7889 0.0000 0.0000
4.0000 0.0000 0.0000 1.7889 0.0000 0.0000 0.0000 2.3094 0.0000
0.0000 0.0000 0.0000 0.0000 2.3094 0.0000 0.0000 0.0000 2.3094
[torch.FloatTensor of size 3x9]
```

# Dropout

For a given network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),  
                      nn.Linear(100, 50), nn.ReLU(),  
                      nn.Linear(50, 2));
```

# Dropout

For a given network

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),  
                      nn.Linear(100, 50), nn.ReLU(),  
                      nn.Linear(50, 2));
```

we can simply add dropout layers

```
model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(),  
                      nn.Dropout(),  
                      nn.Linear(100, 50), nn.ReLU(),  
                      nn.Dropout(),  
                      nn.Linear(50, 2));
```

# Dropout

A model using dropout has to be set in "train" or "test" mode

# Dropout

A model using dropout has to be set in "train" or "test" mode

The method `nn.Module.train(mode)` recursively sets the flag training to all sub-modules.

```
>>> dropout = nn.Dropout()
>>> model = nn.Sequential(nn.Linear(3, 10), dropout, nn.Linear(10, 3))
>>> dropout.training
True
>>> model.train(False)
Sequential (
  (0): Linear (3 -> 10) (1): Dropout (p = 0.5) (2): Linear (10 -> 3)
)
>>> dropout.training
False
```

# Spatial Dropout

As pointed out by Tompson [et al.](#) (2015), units in a 2d activation map are generally locally correlated, and dropout has virtually no effect.

They proposed SpatialDropout, which drops channels instead of individual units.

# Spatial Dropout

```
>>> dropout2d = nn.Dropout2d()
>>> x = Variable(Tensor(2, 3, 2, 2).fill_(1.0))
>>> dropout2d(x)
Variable containing:
(0 ,0 ,.,.) =
0 0
0 0

(0 ,1 ,.,.) =
0 0
0 0

(0 ,2 ,.,.) =
2 2
2 2

(1 ,0 ,.,.) =
2 2
2 2

(1 ,1 ,.,.) =
0 0
0 0

(1 ,2 ,.,.) =
2 2
2 2
[torch.FloatTensor of size 2x3x2x2]
```



# Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).

# Batch normalization

We saw that maintaining proper statistics of the activations and derivatives was a critical issue to allow the training of deep architectures.

It is the main motivation behind weight initialization rules (we'll cover them later).

A different approach consists of explicitly forcing the activation statistics during the forward pass by re-normalizing them.

**Batch normalization** proposed by Ioffe and Szegedy (2015) was the first method introducing this idea.

# Batch normalization

Normalize activations in each **mini-batch** before activation function:  
**speeds up** and **stabilizes** training (less dependent on init)

Batch normalization forces the activation first and second order moments, so that the following layers do not need to adapt to their drift.

# Batch normalization

Normalize activations in each **mini-batch** before activation function:  
**speeds up** and **stabilizes** training (less dependent on init)

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;  
Parameters to be learned:  $\gamma, \beta$   
**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

*Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015*

# Batch normalization

During training batch normalization **shifts and rescales according to the mean and variance estimated on the batch.**

<b>Input:</b> Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\};$ Parameters to be learned: $\gamma, \beta$	
<b>Output:</b> $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$	
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$	// mini-batch mean
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$	// mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$	// normalize
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$	// scale and shift

As for dropout, the model behaves differently during train and test.

# Batch normalization

At **inference time**, use average and standard deviation computed on **the whole dataset** instead of batch

Widely used in **ConvNets**, but requires the mini-batch to be large enough to compute statistics in the minibatch.

# Batch normalization

As dropout, batch normalization is implemented as a separate module `torch.BatchNorm1d` that processes the input components separately.

```
>>> x = torch.Tensor(10000, 3).normal_()
>>> x = x * torch.Tensor([2, 5, 10]) + torch.Tensor([-10, 25, 3])
>>> x = Variable(x)
>>> x.data.mean(0)
-9.9898
24.9165
2.8945
[torch.FloatTensor of size 3]

>>> x.data.std(0)
2.0006
5.0146
9.9501
[torch.FloatTensor of size 3]
```

# Batch normalization

Since the module has internal variables to keep statistics, it must be provided with the sample dimension at creation.

```
>>> bn = nn.BatchNorm1d(3)
>>> bn.bias.data = torch.Tensor([2, 4, 8])
>>> bn.weight.data = torch.Tensor([1, 2, 3])
>>> y = bn(x)
>>> y.data.mean(0)
```

```
2.0000
4.0000
8.0000
[torch.FloatTensor of size 3]
>>> y.data.std(0)
```

```
1.0000
2.0001
3.0001
[torch.FloatTensor of size 3]
```



# Batch normalization

## BatchNorm2d example

```
>>> x = Variable(torch.randn(20, 100, 35, 45))
>>> bn2d = nn.BatchNorm2d(100)
>>> y = bn2d(x)
>>> x.size()
```

```
torch.Size([20, 100, 35, 45])
>>> bn2d.weight.data.size()
```

```
torch.Size([100])
>>> bn2d.bias.data.size()
```

```
torch.Size([100])
```

# Batch normalization

Results on ImageNet LSVRC 2012:

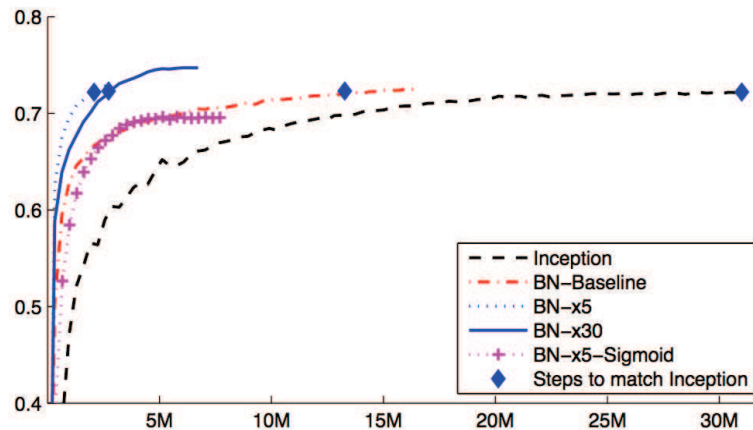


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.

*Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015*

# Batch normalization

Results on ImageNet LSVRC 2012:

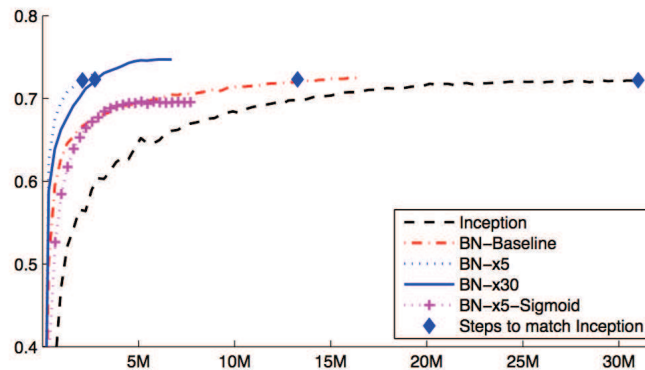


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.

- learning rate can be greater
- dropout and local normalization are not necessary
- $L^2$  regularization influence should be reduced

*Batch normalization: Accelerating deep network training by reducing internal covariate shift, Ioffe and Szegedy, ICML 2015*

# Batch normalization

Deep MLP on a 2d "disc" toy example, with naive Gaussian weight initialization, cross-entropy, standard SGD,  $\eta = 0.1$ .

```
def create_model(with_batchnorm, nc = 32, depth = 16):
    modules = []

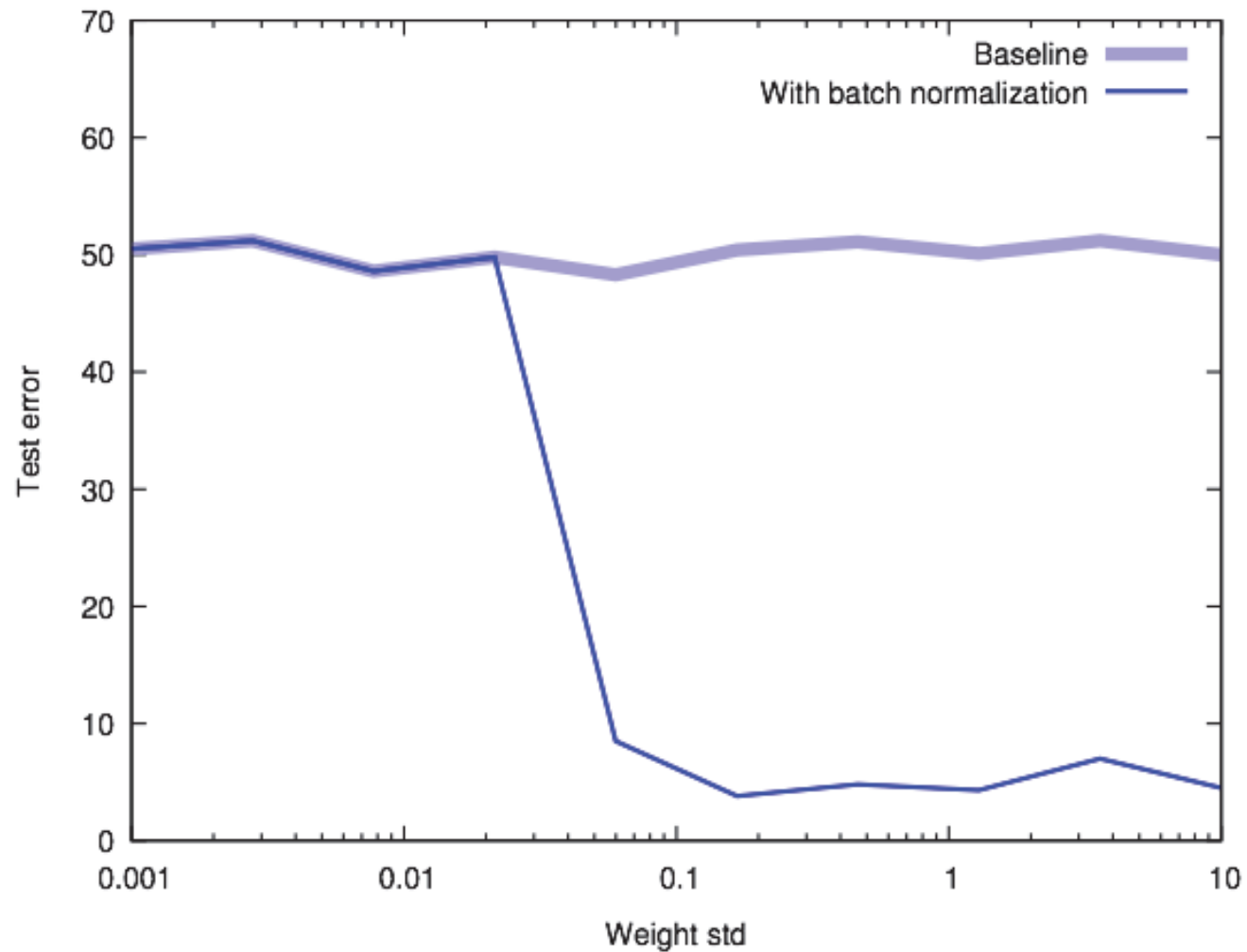
    modules.append(nn.Linear(2, nc))
    if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
    modules.append(nn.ReLU())

    for d in range(depth):
        modules.append(nn.Linear(nc, nc))
        if with_batchnorm: modules.append(nn.BatchNorm1d(nc))
        modules.append(nn.ReLU())

    modules.append(nn.Linear(nc, 2))

    return nn.Sequential(*modules)
```

# Batch normalization



# Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

# Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

The algorithm is then similar as Batch Normalization

# Layer Normalizations

Normalize on the statistics of the **layer activations** instead of mini-batch.

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

The algorithm is then similar as Batch Normalization

Suited for **RNNs**, degrades performance of **CNNs**



# Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v}$$

*Weight normalization: A simple reparameterization to accelerate training of deep neural networks, Salimans, NIPS 2016.*

# Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v}$$

One new parameter  $g$  to learn per neuron

*Weight normalization: A simple reparameterization to accelerate training of deep neural networks, Salimans, NIPS 2016.*

# Weight Normalization

Reparametrize weights of neurons, to decouple **direction** and **norm** of the weight:

$$\mathbf{w} = \frac{g}{\|\mathbf{v}\|} \mathbf{v}$$

One new parameter  $g$  to learn per neuron

Careful **data-based** initialization of  $g$  and neuron bias  $b$  is better (not applicable to RNNs)

*Weight normalization: A simple reparameterization to accelerate training of deep neural networks, Salimans, NIPS 2016.*

# Multiple variants

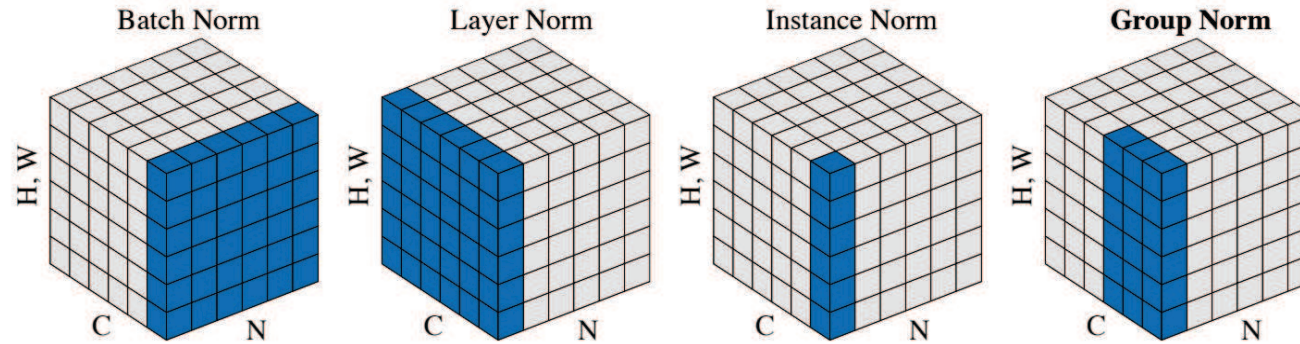


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with  $N$  as the batch axis,  $C$  as the channel axis, and  $(H, W)$  as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

# Architectures

# Architectures

`torchvision.models` provides a collection of reference networks for computer vision, e.g.:

```
import torchvision  
alexnet = torchvision.models.alexnet()
```

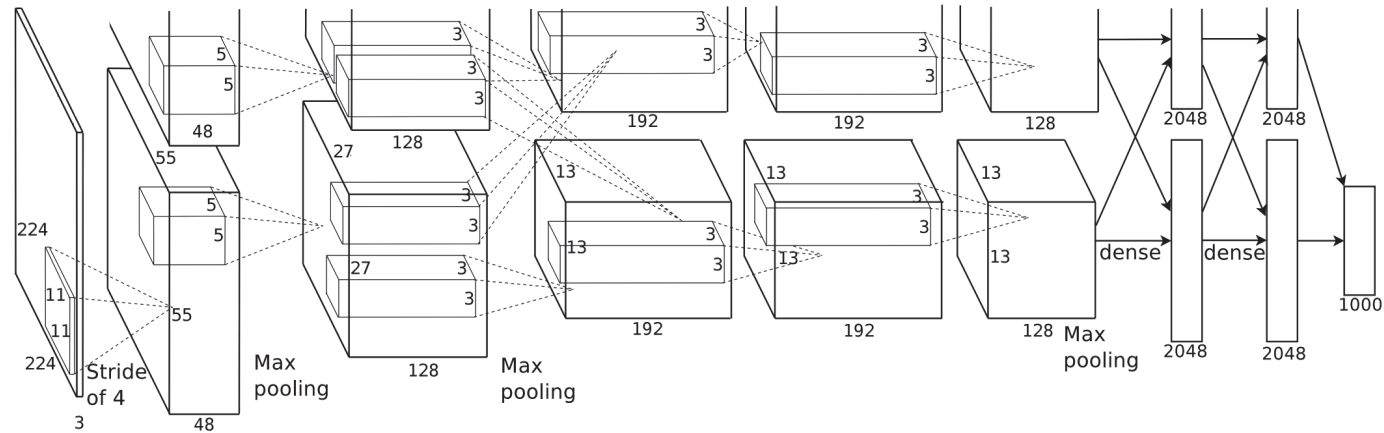
The trained models can be obtained by passing `pretrained = True` to the constructor(s). This may involve an heavy download given there size.

# LeNet5

10 classes, input 1 x 28 x 28

```
(features): Sequential (  
  (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))  
  (1): ReLU (inplace)  
  (2): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
  (3): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))  
  (4): ReLU (inplace)  
  (5): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))  
)  
(classifier): Sequential (  
  (0): Linear (400 -> 120)  
  (1): ReLU (inplace)  
  (2): Linear (120 -> 84)  
  (3): ReLU (inplace)  
  (4): Linear (84 -> 10) )
```

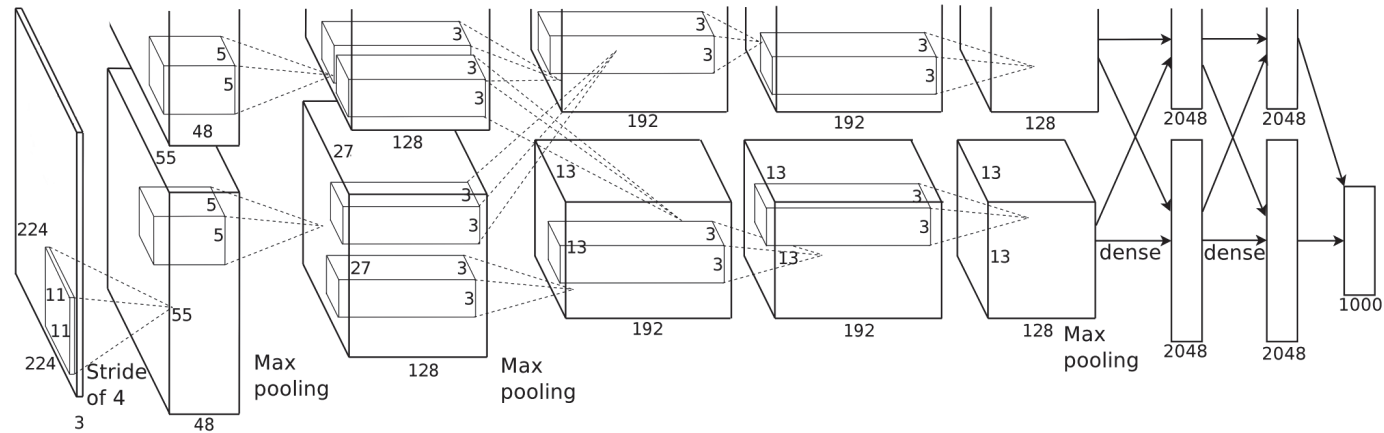
# AlexNet



*Imagenet classification with deep convolutional neural networks, Krizhevsky et al., NIPS 2012*



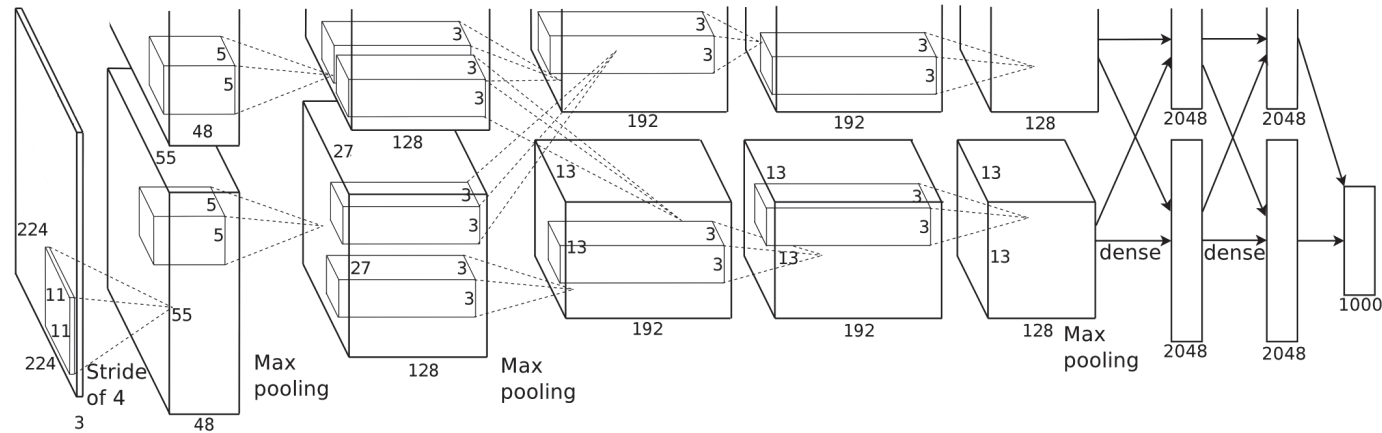
# AlexNet



First conv layer: kernel 11x11x3x96 stride 4

*Imagenet classification with deep convolutional neural networks, Krizhevsky et al., NIPS 2012*

# AlexNet

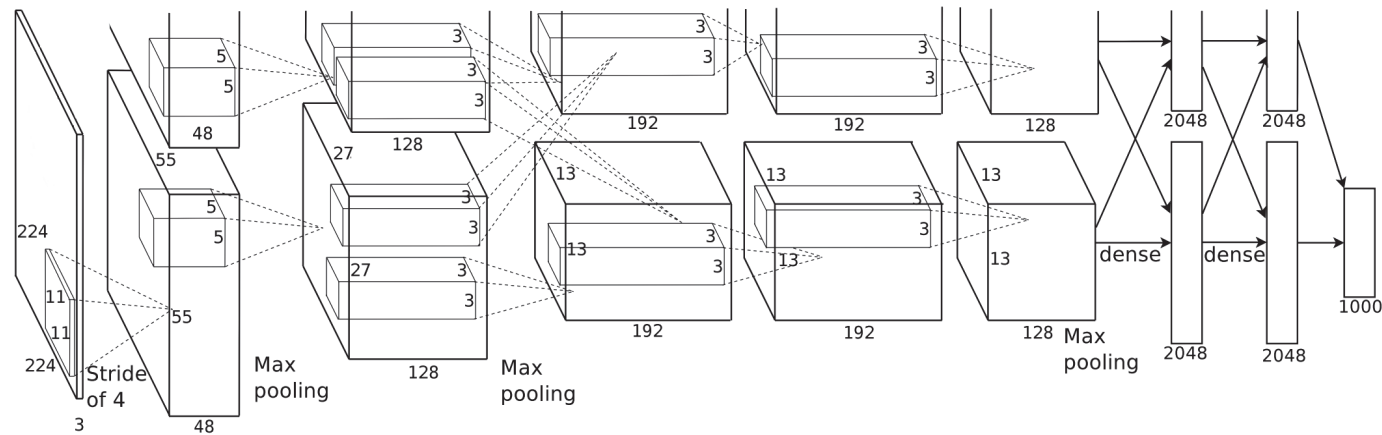


First conv layer: kernel 11x11x3x96 stride 4

- Kernel shape: (11,11,3,96)
- Output shape: (55,55,96)
- Number of parameters: 34,944
- Equivalent MLP parameters:  $43.7 \times 10^9$

*Imagenet classification with deep convolutional neural networks, Krizhevsky et al., NIPS 2012*

# AlexNet



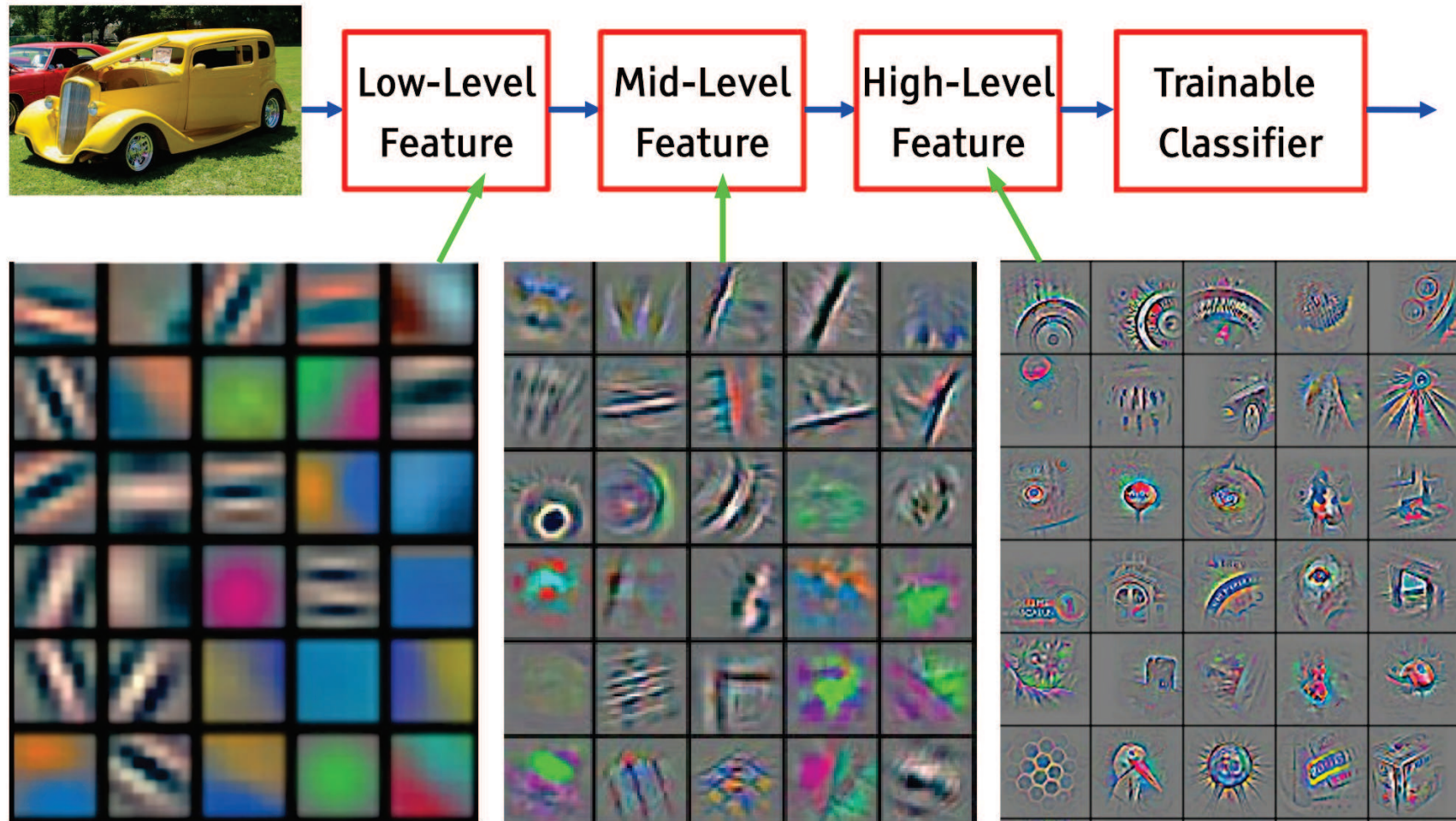
INPUT: [227x227x3]  
 CONV1: [55x55x96] 96 11x11 filters at stride 4, pad 0  
 MAX POOL1: [27x27x96] 3x3 filters at stride 2  
 CONV2: [27x27x256] 256 5x5 filters at stride 1, pad 2  
 MAX POOL2: [13x13x256] 3x3 filters at stride 2  
 CONV3: [13x13x384] 384 3x3 filters at stride 1, pad 1  
 CONV4: [13x13x384] 384 3x3 filters at stride 1, pad 1  
 CONV5: [13x13x256] 256 3x3 filters at stride 1, pad 1  
 MAX POOL3: [6x6x256] 3x3 filters at stride 2  
 FC6: [4096] 4096 neurons  
 FC7: [4096] 4096 neurons  
 FC8: [1000] 1000 neurons (softmax logits)

# AlexNet

```
(features): Sequential (  
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))  
(1): ReLU (inplace)  
(2): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))  
(4): ReLU (inplace)  
(5): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(7): ReLU (inplace)  
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(9): ReLU (inplace)  
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))  
(11): ReLU (inplace)  
(12): MaxPool2d (size=(3, 3), stride=(2, 2), dilation=(1, 1))  
)
```

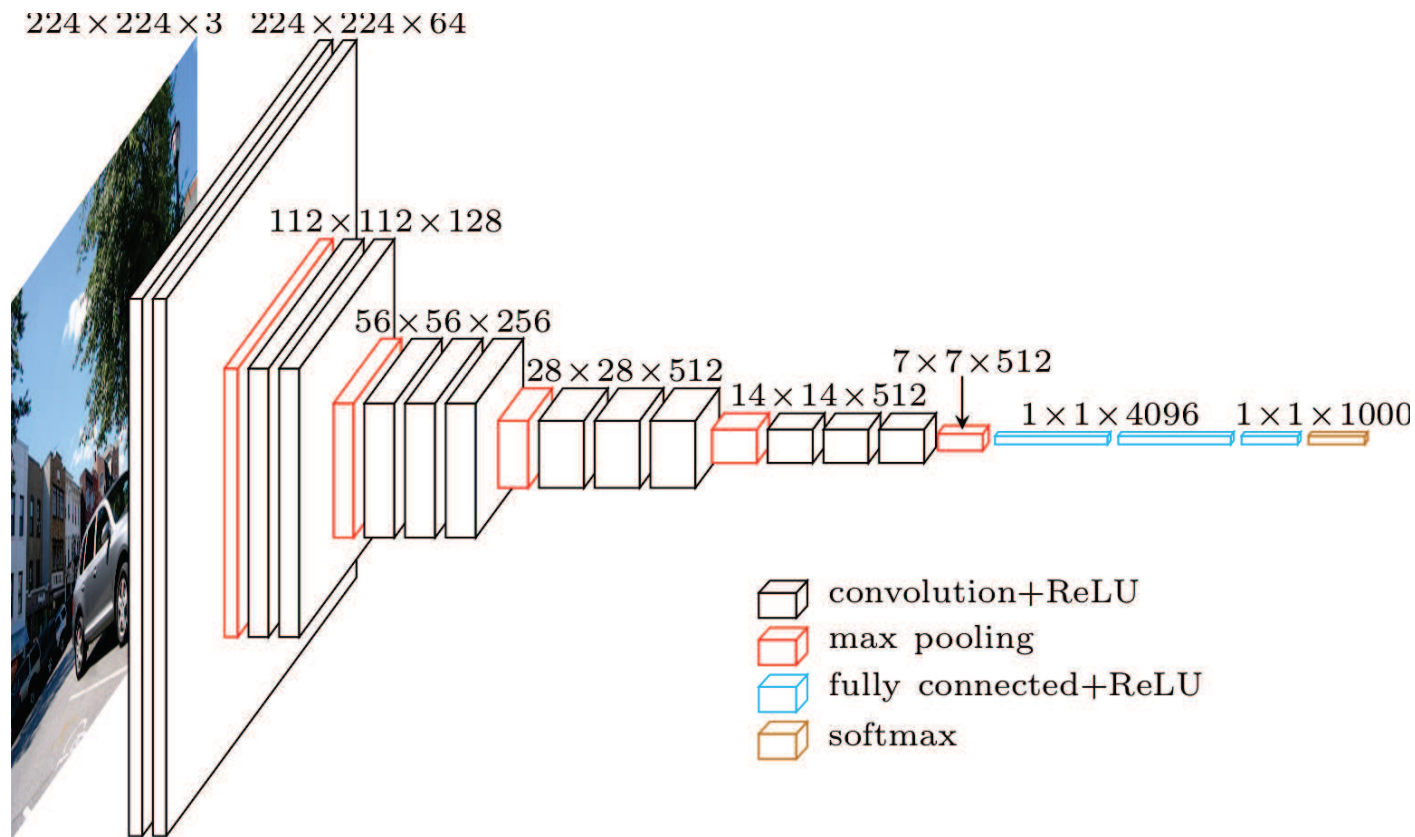
```
(classifier): Sequential (  
(0): Dropout (p = 0.5)  
(1): Linear (9216 -> 4096)  
(2): ReLU (inplace)  
(3): Dropout (p = 0.5)  
(4): Linear (4096 -> 4096)  
(5): ReLU (inplace)  
(6): Linear (4096 -> 1000)  
)
```

# Hierarchical representation



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

# VGG-16



*Very deep convolutional networks for large-scale image recognition, Simonyan and Zisserman, NIPS 2014*



# Memory and Parameters

	Activation maps	Parameters
INPUT:	[224x224x3] = 150K	0
CONV3-64:	[224x224x64] = 3.2M	(3x3x3)x64 = 1,728
CONV3-64:	[224x224x64] = 3.2M	(3x3x64)x64 = 36,864
POOL2:	[112x112x64] = 800K	0
CONV3-128:	[112x112x128] = 1.6M	(3x3x64)x128 = 73,728
CONV3-128:	[112x112x128] = 1.6M	(3x3x128)x128 = 147,456
POOL2:	[56x56x128] = 400K	0
CONV3-256:	[56x56x256] = 800K	(3x3x128)x256 = 294,912
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
POOL2:	[28x28x256] = 200K	0
CONV3-512:	[28x28x512] = 400K	(3x3x256)x512 = 1,179,648
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
POOL2:	[14x14x512] = 100K	0
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
POOL2:	[7x7x512] = 25K	0
FC:	[1x1x4096] = 4096	7x7x512x4096 = 102,760,448
FC:	[1x1x4096] = 4096	4096x4096 = 16,777,216
FC:	[1x1x1000] = 1000	4096x1000 = 4,096,000

TOTAL activations: 24M x 4 bytes ~= 93MB / image (x2 for backward)

TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)

# Memory and Parameters

	Activation maps	Parameters
INPUT:	[224x224x3] = 150K	0
CONV3-64:	[224x224x64] = 3.2M	(3x3x3)x64 = 1,728
CONV3-64:	[224x224x64] = 3.2M	(3x3x64)x64 = 36,864
POOL2:	[112x112x64] = 800K	0
CONV3-128:	[112x112x128] = 1.6M	(3x3x64)x128 = 73,728
CONV3-128:	[112x112x128] = 1.6M	(3x3x128)x128 = 147,456
POOL2:	[56x56x128] = 400K	0
CONV3-256:	[56x56x256] = 800K	(3x3x128)x256 = 294,912
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
CONV3-256:	[56x56x256] = 800K	(3x3x256)x256 = 589,824
POOL2:	[28x28x256] = 200K	0
CONV3-512:	[28x28x512] = 400K	(3x3x256)x512 = 1,179,648
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
CONV3-512:	[28x28x512] = 400K	(3x3x512)x512 = 2,359,296
POOL2:	[14x14x512] = 100K	0
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
CONV3-512:	[14x14x512] = 100K	(3x3x512)x512 = 2,359,296
POOL2:	[7x7x512] = 25K	0
FC:	[1x1x4096] = 4096	7x7x512x4096 = 102,760,448
FC:	[1x1x4096] = 4096	4096x4096 = 16,777,216
FC:	[1x1x1000] = 1000	4096x1000 = 4,096,000

TOTAL activations: 24M x 4 bytes ~= 93MB / image (x2 for backward)

TOTAL parameters: 138M x 4 bytes ~= 552MB (x2 for plain SGD, x4 for Adam)



# VGG-19

```
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU (inplace)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU (inplace)
(4): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(6): ReLU (inplace)
(7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(8): ReLU (inplace)
(9): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU (inplace)
(12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(13): ReLU (inplace)
(14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(15): ReLU (inplace)
(16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(17): ReLU (inplace)
(18): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(20): ReLU (inplace)
(21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(22): ReLU (inplace)
(23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(24): ReLU (inplace)
(25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(26): ReLU (inplace)
(27): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU (inplace)
```

...

# VGG-19

...

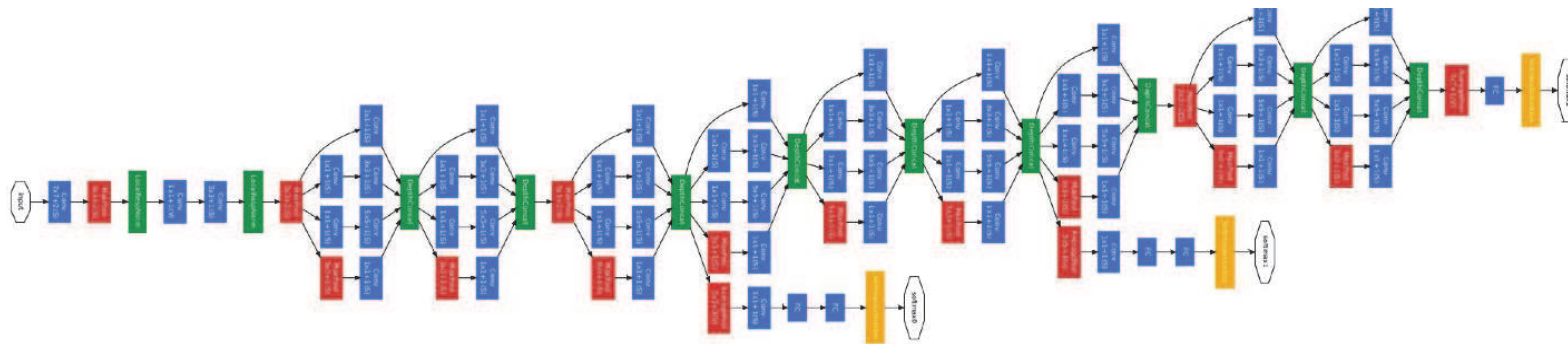
```
(30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(31): ReLU (inplace)
(32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(33): ReLU (inplace)
(34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(35): ReLU (inplace)
(36): MaxPool2d (size=(2, 2), stride=(2, 2), dilation=(1, 1))

(classifier): Sequential (
  (0): Linear (25088 -> 4096)
  (1): ReLU (inplace)
  (2): Dropout (p = 0.5)
  (3): Linear (4096 -> 4096)
  (4): ReLU (inplace)
  (5): Dropout (p = 0.5)
  (6): Linear (4096 -> 1000)
)
```

# GoogLeNet / Inception

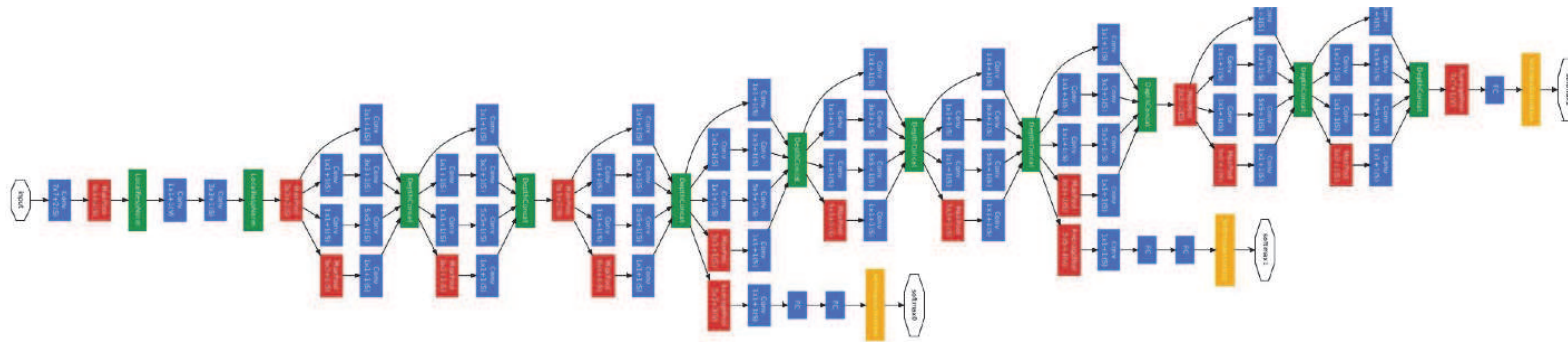
Szegedy et al. (2015) also introduce the idea of "auxiliary classifiers" to help the propagation of the gradient in the early layers.

This is motivated by the reasonable performance of shallow networks that indicates early layers already encode informative and invariant features.



# GoogLeNet / Inception

The resulting GoogLeNet has 12 times less parameters than AlexNet and is more accurate on ILSVRC14 (Szegedy et al., 2015).



It was later extended with batch-normalization (Ioffe and Szegedy, 2015) and pass-through a la resnet (Szegedy et al., 2016)

# GoogLeNet / Inception

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Fun features:

- Only 5 million params!  
(Removes FC layers completely)

**Compared to AlexNet:**

- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

# A saturation point

If we continue stacking more layers on a CNN:



# A saturation point

If we continue stacking more layers on a CNN:



Deeper models are harder to optimize

# ResNet

A block learns the residual w.r.t. identity

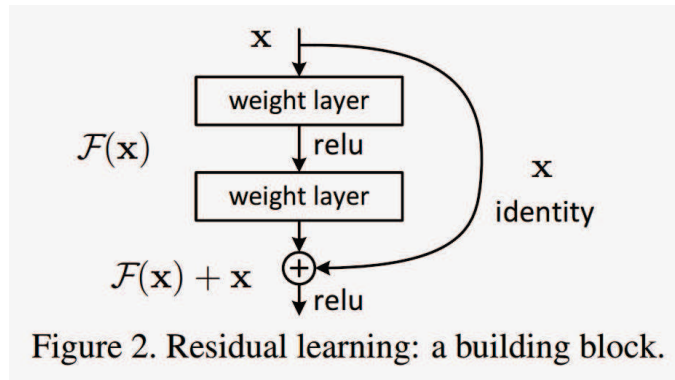


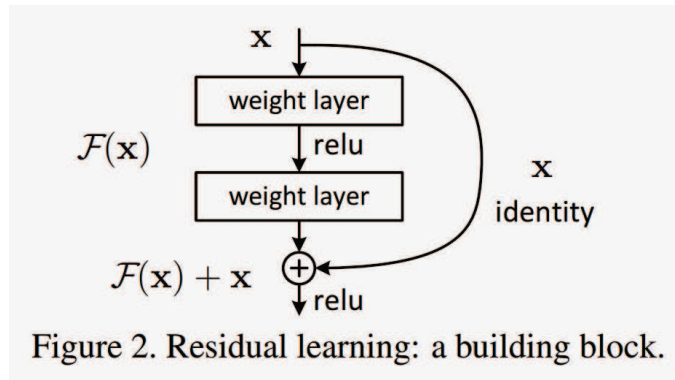
Figure 2. Residual learning: a building block.

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*



# ResNet

A block learns the residual w.r.t. identity



- Good optimization properties

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*

# ResNet

Even deeper models:

34, 50, 101, 152 layers

*Deep residual learning for image  
recognition, K. He et al., CVPR 2016.*

# ResNet

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

**5.25%** top-5 error vs 7.1%

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*

# ResNet

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

5.25% top-5 error vs 7.1%

Less parameters

25M vs 138M

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*

# ResNet

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

**5.25%** top-5 error vs 7.1%

Less parameters

**25M** vs 138M

Computational complexity

**3.8B Flops** vs 15.3B Flops

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*

# ResNet

ResNet50 Compared to VGG:

Superior accuracy in all vision tasks

**5.25%** top-5 error vs 7.1%

Less parameters

**25M** vs 138M

Computational complexity

**3.8B Flops** vs 15.3B Flops

Fully Convolutional until the last layer

*Deep residual learning for image recognition, K. He et al., CVPR 2016.*

# ResNet

## Performance on ImageNet

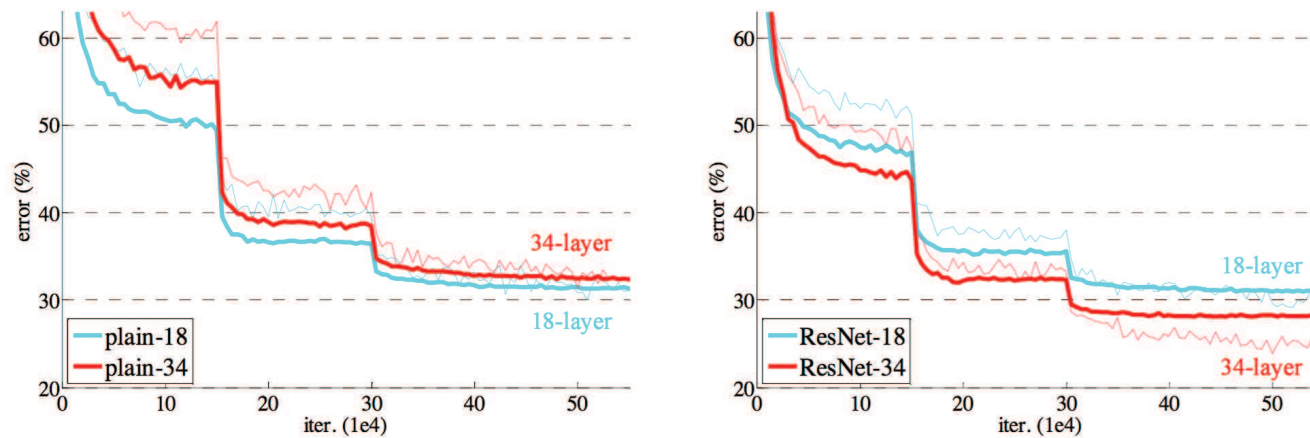
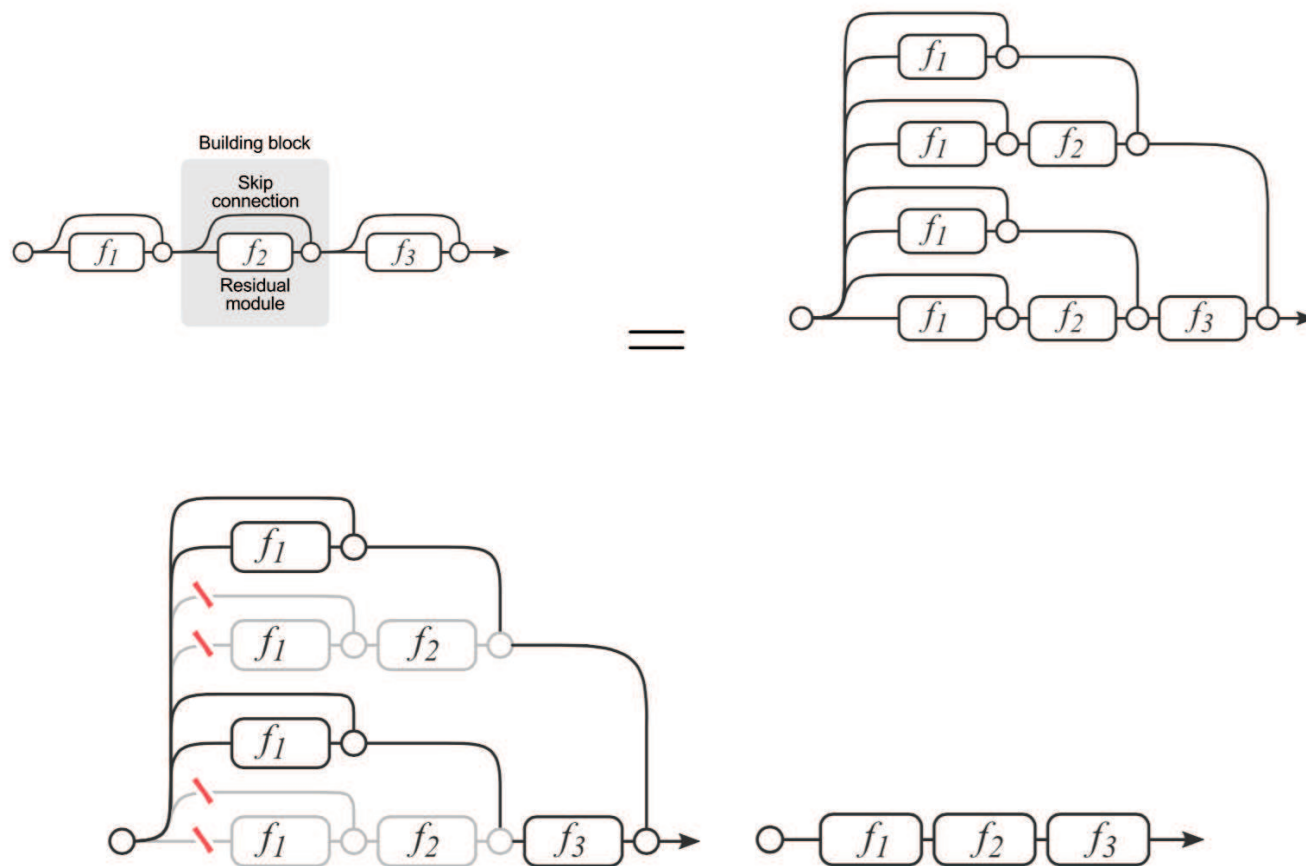


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

# ResNet

The output of a residual network can be understood as an ensemble, which explains in part its stability





# ResNet

## Results

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3\_1, conv4\_1, and conv5\_1 with a stride of 2.

# ResNet

## Results

method	top-5 err. (test)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PreLU-net [13]	4.94
BN-inception [16]	4.82
<b>ResNet (ILSVRC'15)</b>	<b>3.57</b>

Table 5. Error rates (%) of **ensembles**. The top-5 error is on the test set of ImageNet and reported by the test server.

# ResNet

In PyTorch:

```
def make_resnet_block(num_feature_maps , kernel_size = 3):  
    return nn.Sequential(  
        nn.Conv2d(num_feature_maps , num_feature_maps ,  
                  kernel_size = kernel_size ,  
                  padding = (kernel_size - 1) // 2),  
        nn.BatchNorm2d(num_feature_maps),  
        nn.ReLU(inplace = True),  
        nn.Conv2d(num_feature_maps , num_feature_maps ,  
                  kernel_size = kernel_size ,  
                  padding = (kernel_size - 1) // 2),  
        nn.BatchNorm2d(num_feature_maps),  
    )
```

# ResNet

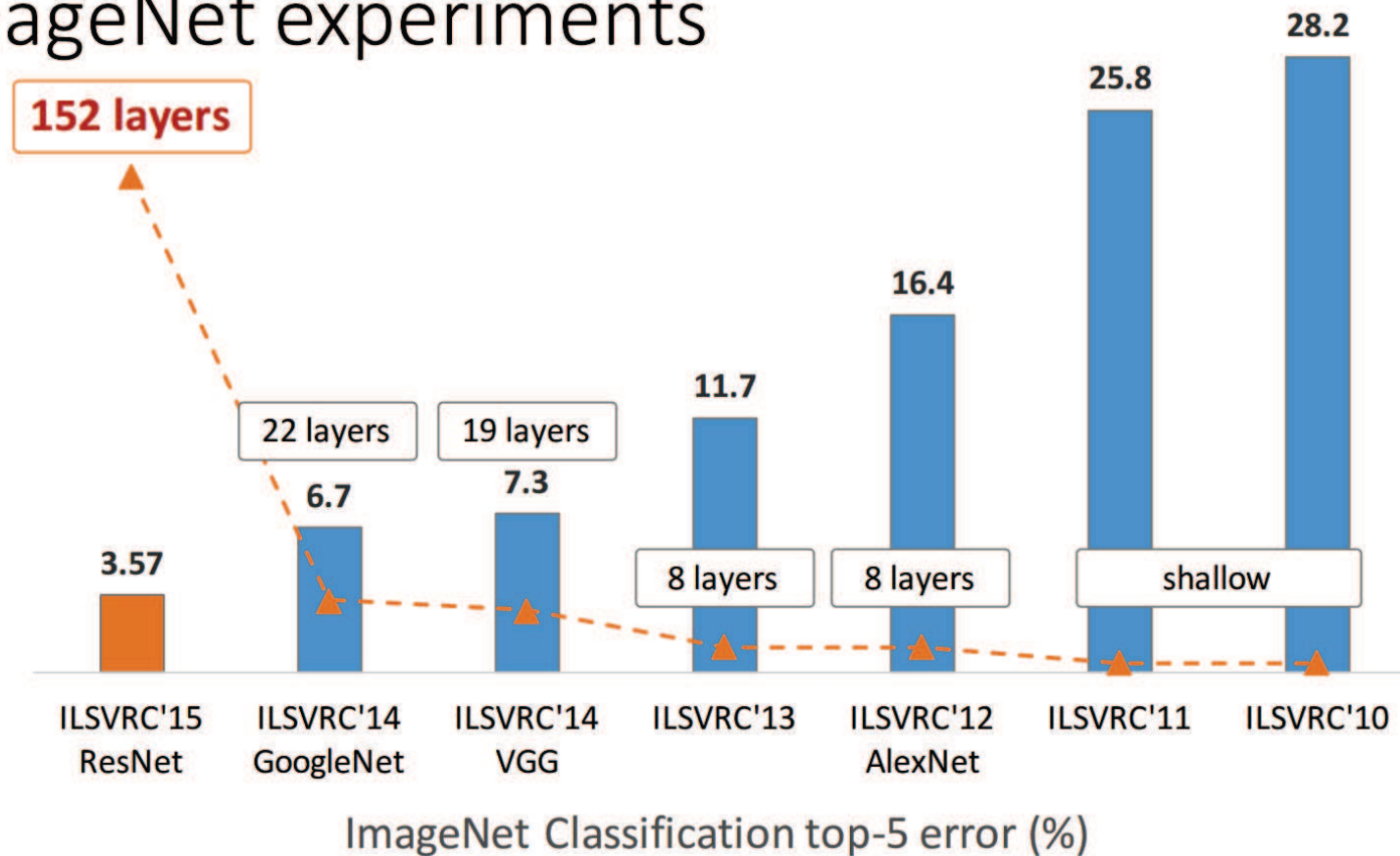
In PyTorch:

```
def __init__(self, num_residual_blocks, num_feature_maps)
...
    self.resnet_blocks = nn.ModuleList()
    for k in range(nb_residual_blocks):
        self.resnet_blocks.append(make_resnet_block(num_feature_maps , 3))
...

def forward(self,x):
...
    for b in self.resnet_blocks:
        x = x + b(x)
...
    return x
```

# Deeper is better

## ImageNet experiments

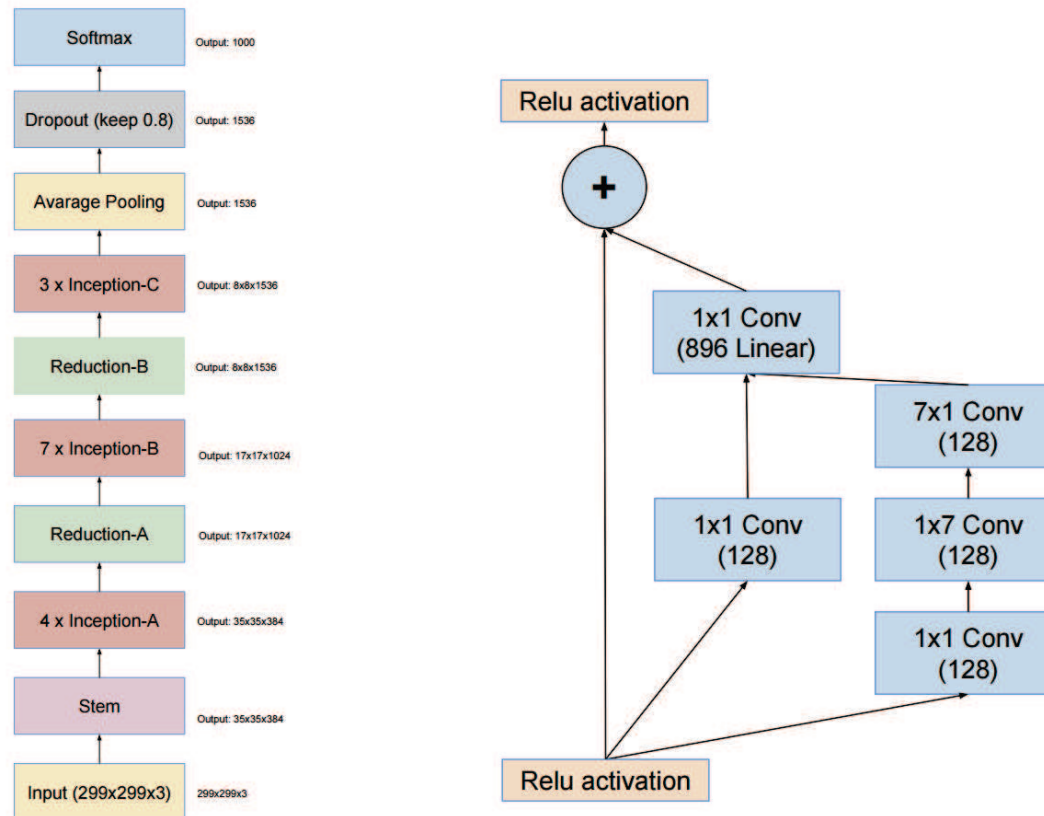


*from Kaiming He slides "Deep residual learning for image recognition." ICML. 2016.*

# Inception-V4 / -ResNet-V2

Deep, modular and state-of-the-art

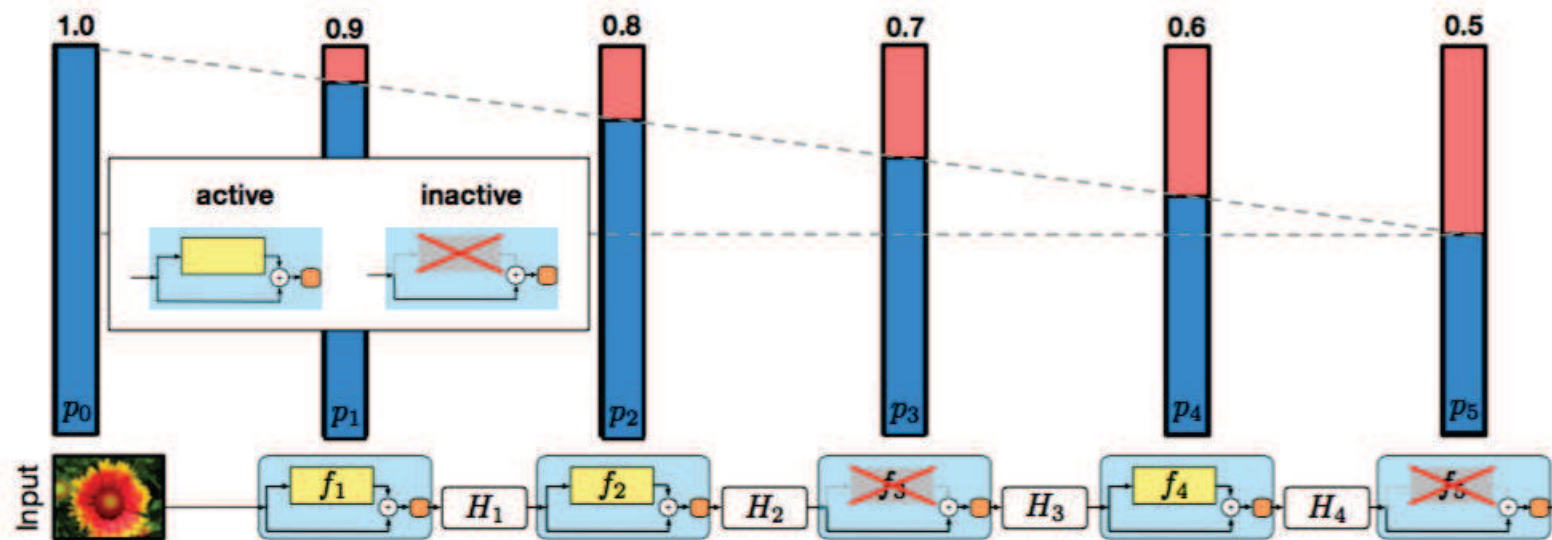
Achieves **3.1% top-5** classification error on imagenet



*Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016*

# Resnet variants: Stochastic Depth Networks

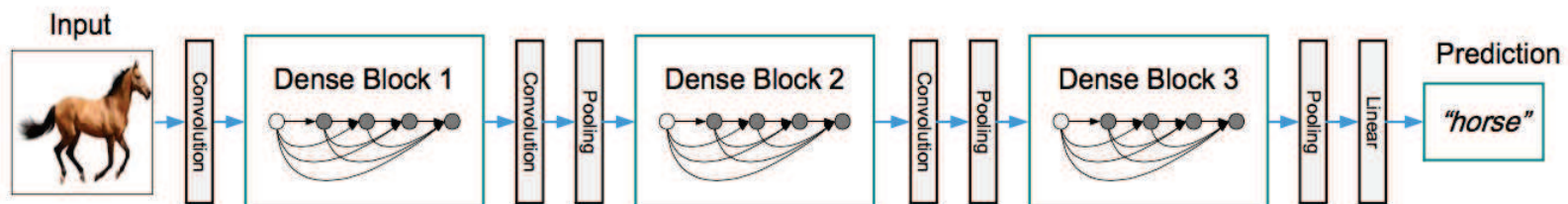
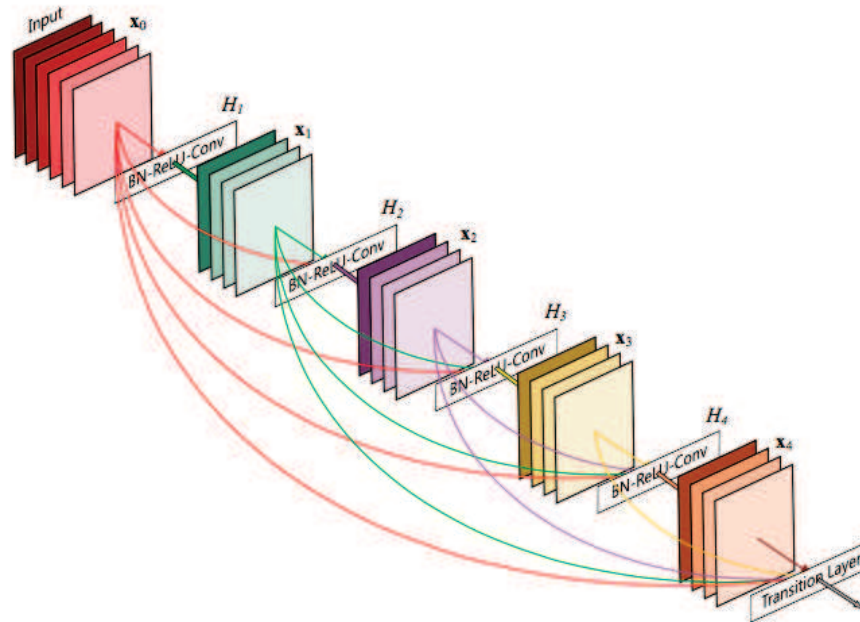
- DropOut at layer level
- Allows training up to 1K layers



**Fig. 2.** The linear decay of  $p_\ell$  illustrated on a ResNet with stochastic depth for  $p_0 = 1$  and  $p_L = 0.5$ . Conceptually, we treat the input to the first ResBlock as  $H_0$ , which is always active.

# Resnet variants: DenseNet

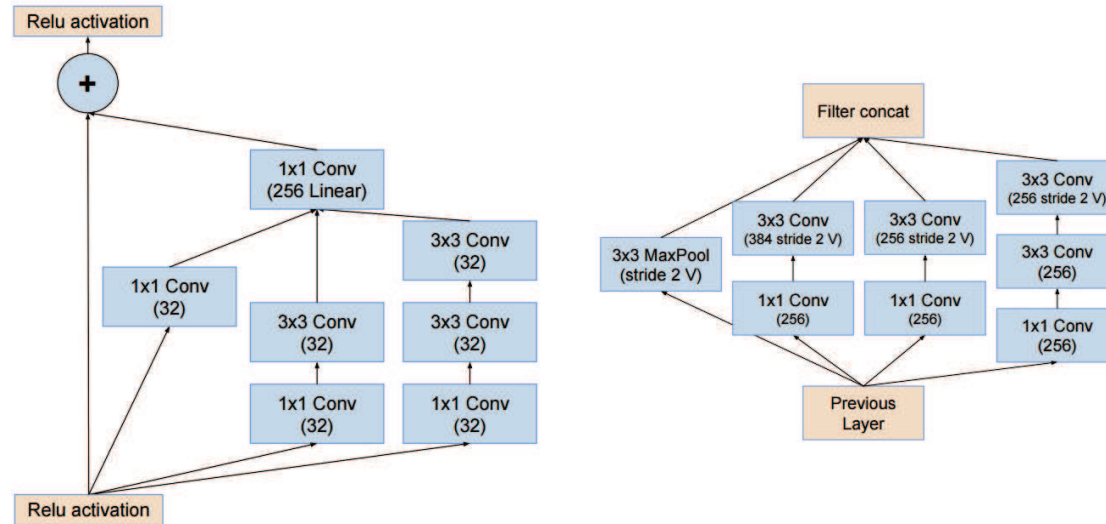
- Copying feature maps to upper layers via skip-connections
- Better reuse of parameters and redundancy avoidance





# Inception-V4 / -ResNet-V2

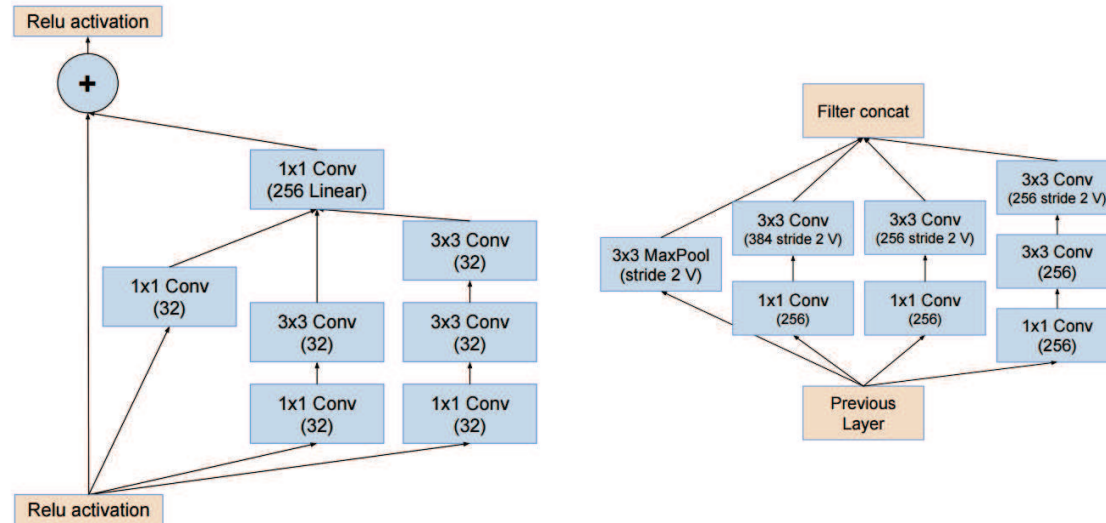
More building blocks engineering...



*Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016*

# Inception-V4 / -ResNet-V2

More building blocks engineering...

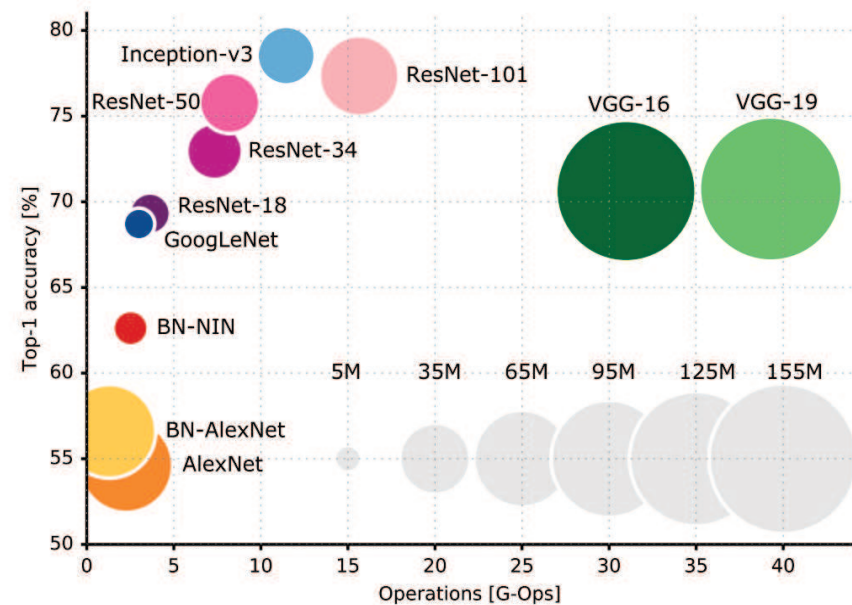
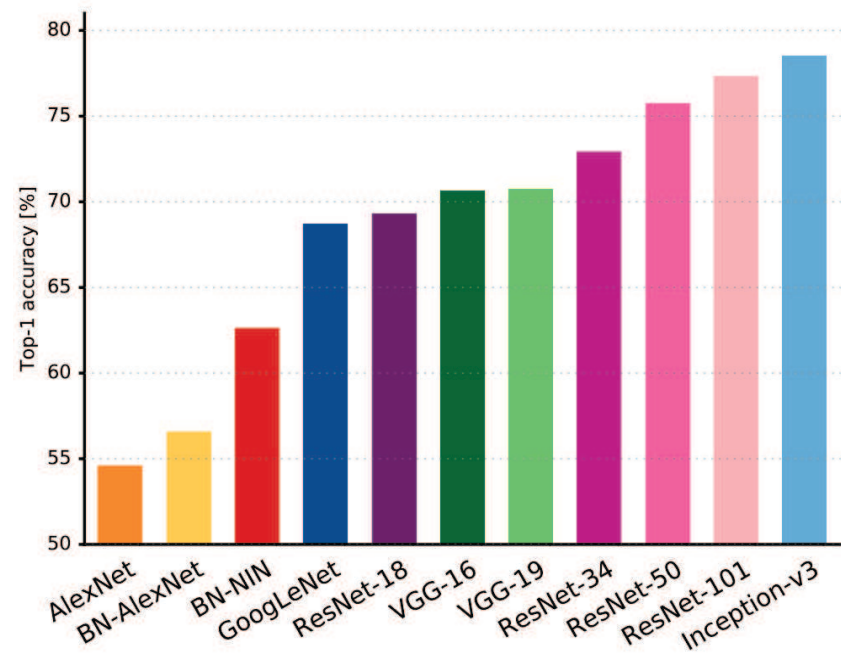


- Active area or research
- See also DenseNets, Wide ResNets, Fractal ResNets, ResNeXts, Pyramidal ResNets...

*Inception-v4, inception-resnet and the impact of residual connections on learning, C. Szegedy et al., 2016*

# Comparison of models

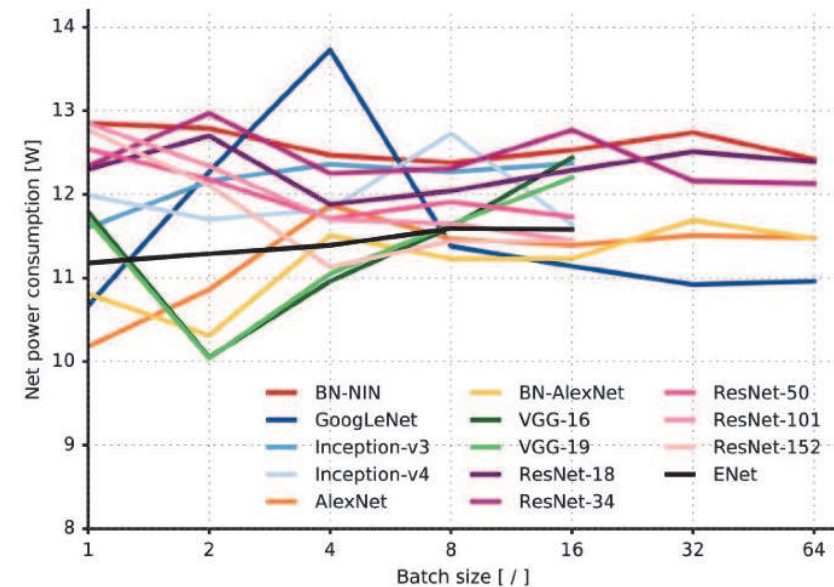
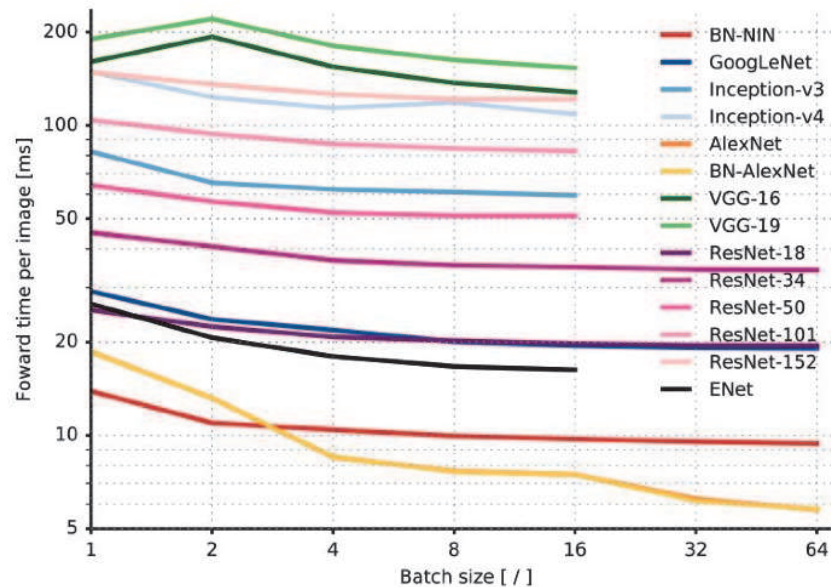
Top 1-accuracy, performance and size on ImageNet



*An Analysis of Deep Neural Network Models for Practical Applications, Canziani et al., 2016*

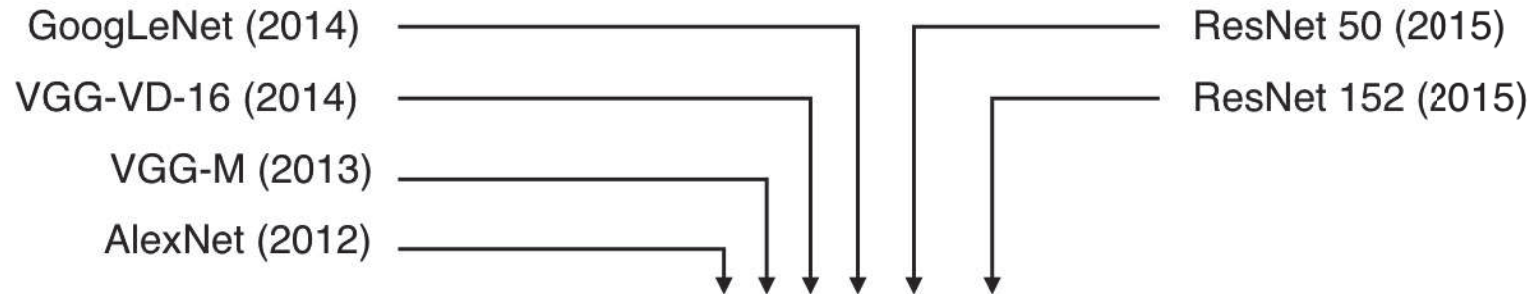
# Comparison of models

## Forward pass time and power consumption



*An Analysis of Deep Neural Network Models for Practical Applications, Canziani et al., 2016*

# Comparison of models



16 convolutional layers

50 convolutional layers

152 convolutional layers

Krizhevsky, I. Sutskever, and G. E. Hinton. *ImageNet classification with deep convolutional neural networks*. In Proc. NIPS, 2012.

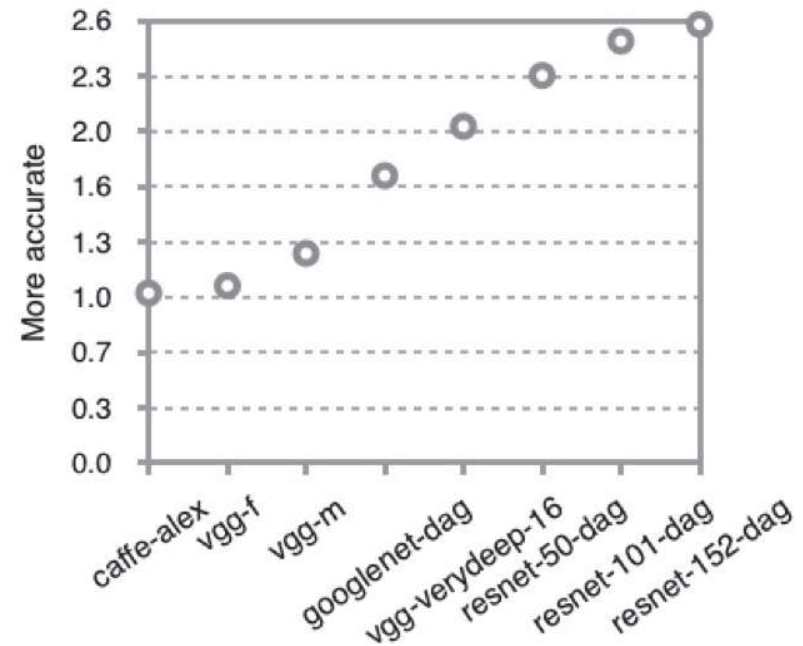
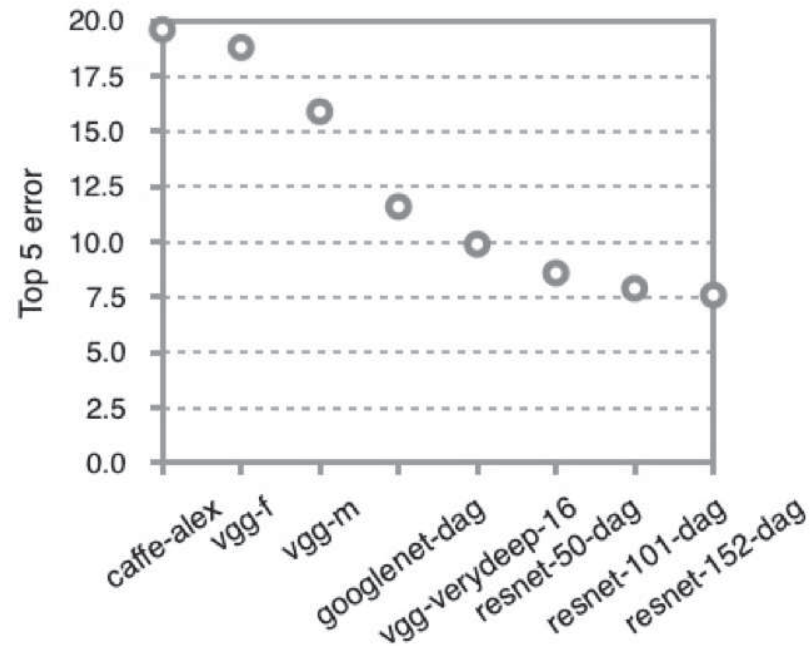
C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. *Going deeper with convolutions*. In Proc. CVPR, 2015.

K. Simonyan and A. Zisserman. *Very deep convolutional networks for large-scale image recognition*. In Proc. ICLR, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. *Deep residual learning for image recognition*. In Proc. CVPR, 2016.

# Comparison of models

3 x more accurate in 3 years

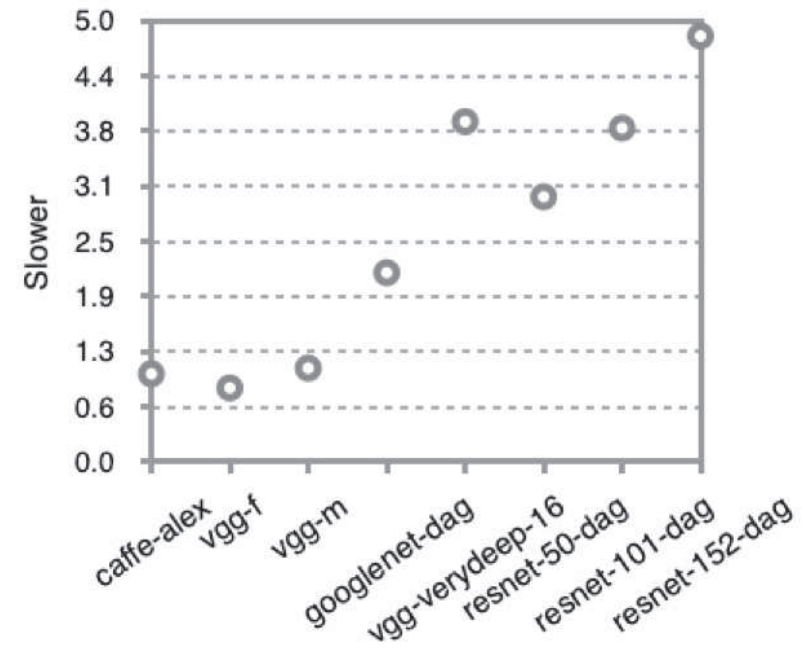
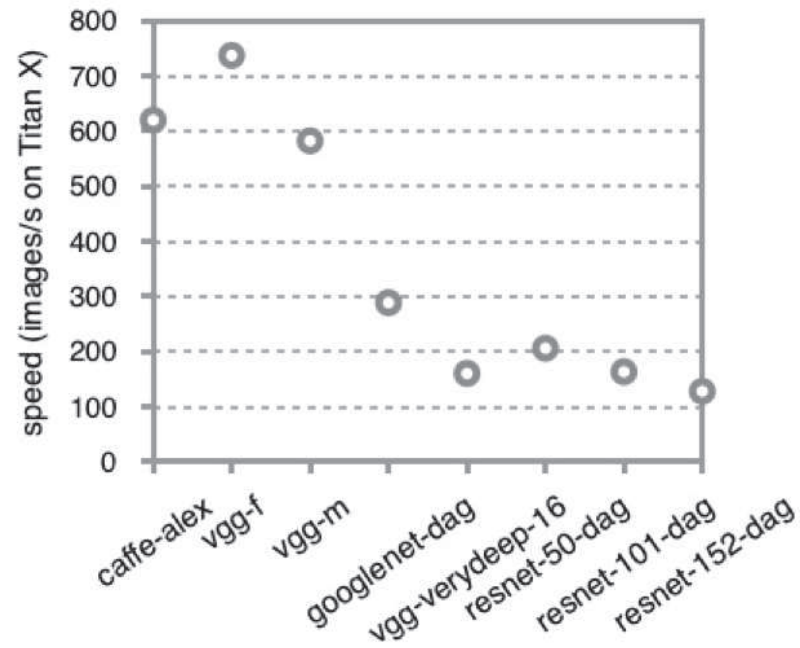


101 ResNet Layers same size/speed as 16 VGG-VD layers



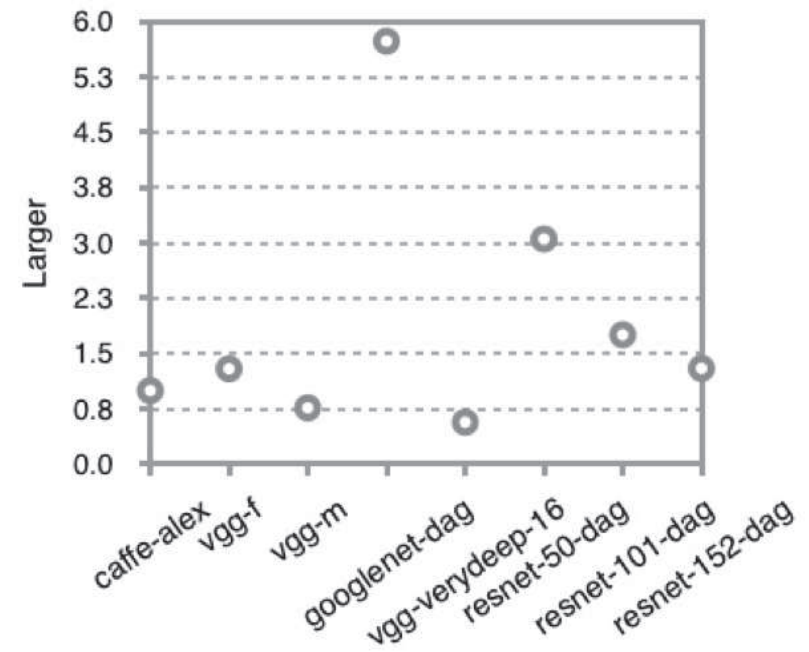
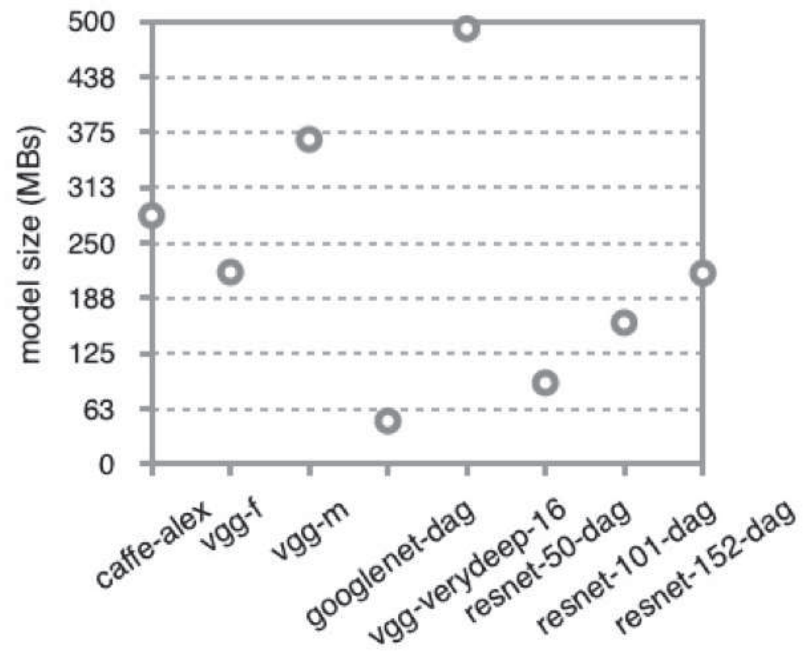
# Comparison of models

Number of parameters is about the same



# Comparison of models

5 x slower





# Comparison of models

