Convolutional Neural Networks Part I

Suleyman Demirel University

CSS634: Deep Learning

PhD Abay Nussipbekov

A bit of history:

Hubel & Wiesel,

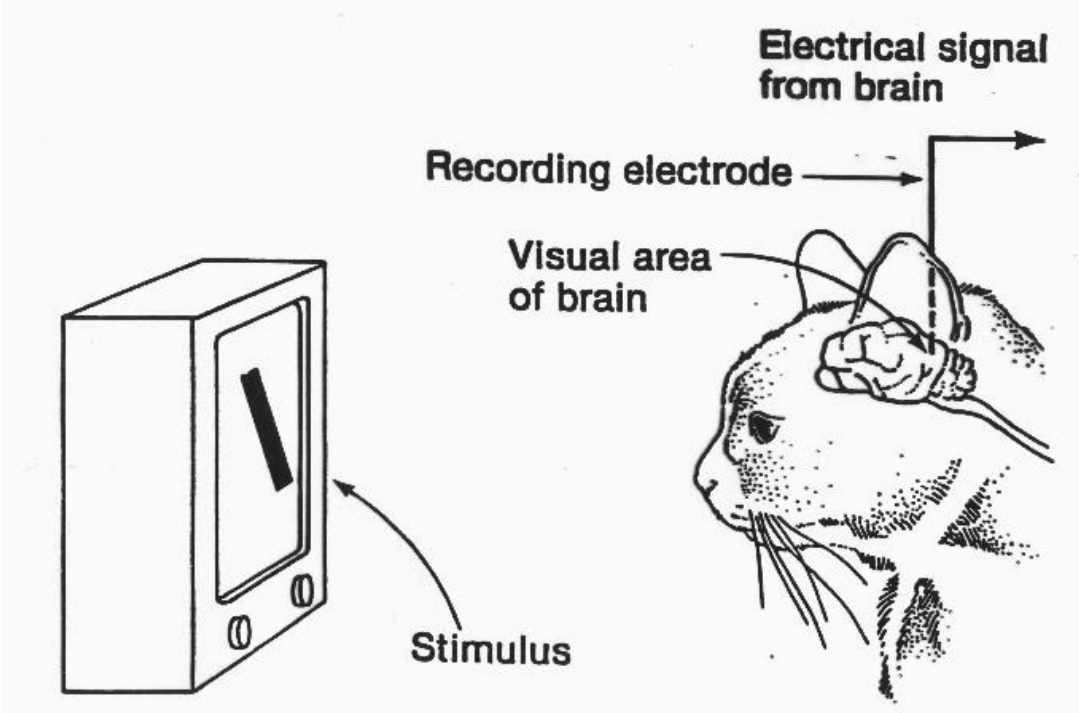
1959

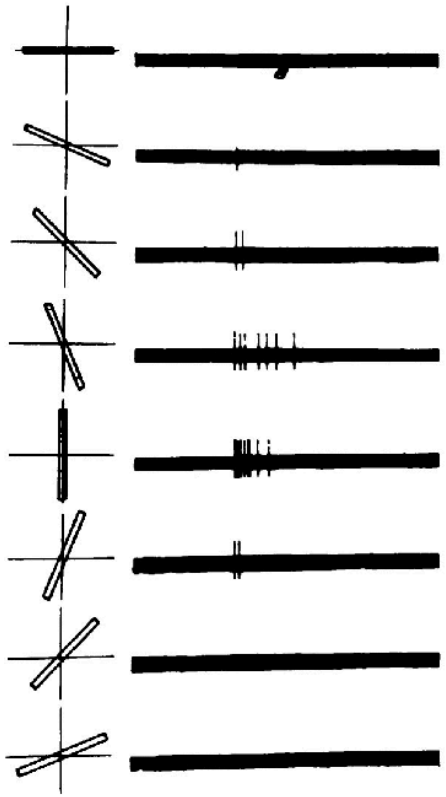
RECEPTIVE FIELDS OF SINGLE NEURONES IN
THE CAT'S STRIATE CORTEX

1962

RECEPTIVE FIELDS, BINOCULAR INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX

1968...





Hierarchical organization

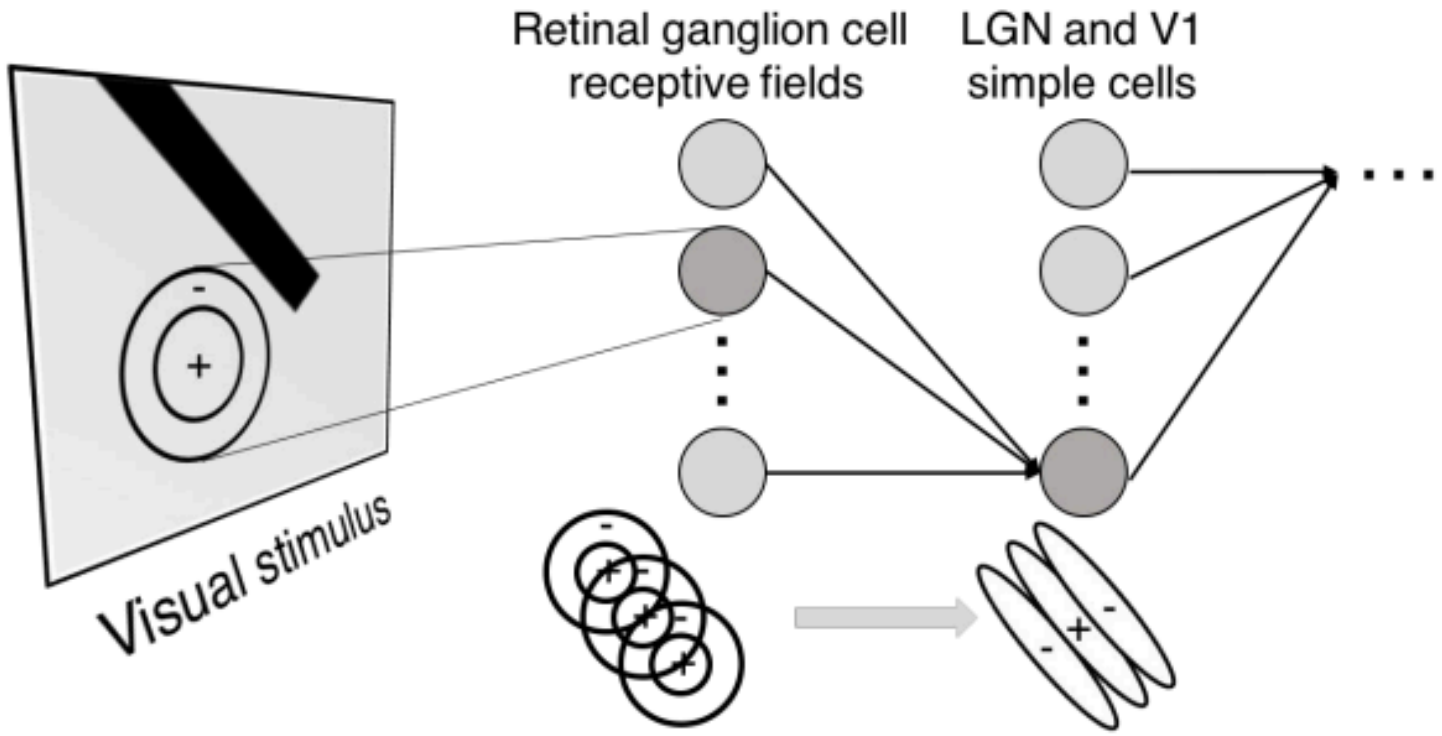
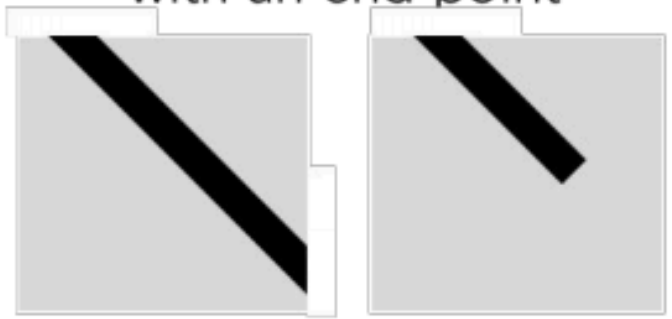


Illustration of hierarchical organization in early visual pathways by Lane McIntosh, copyright CS231n 2017

Simple cells:
Response to light orientation

Complex cells:
Response to light orientation and movement

Hypercomplex cells:
response to movement with an end point



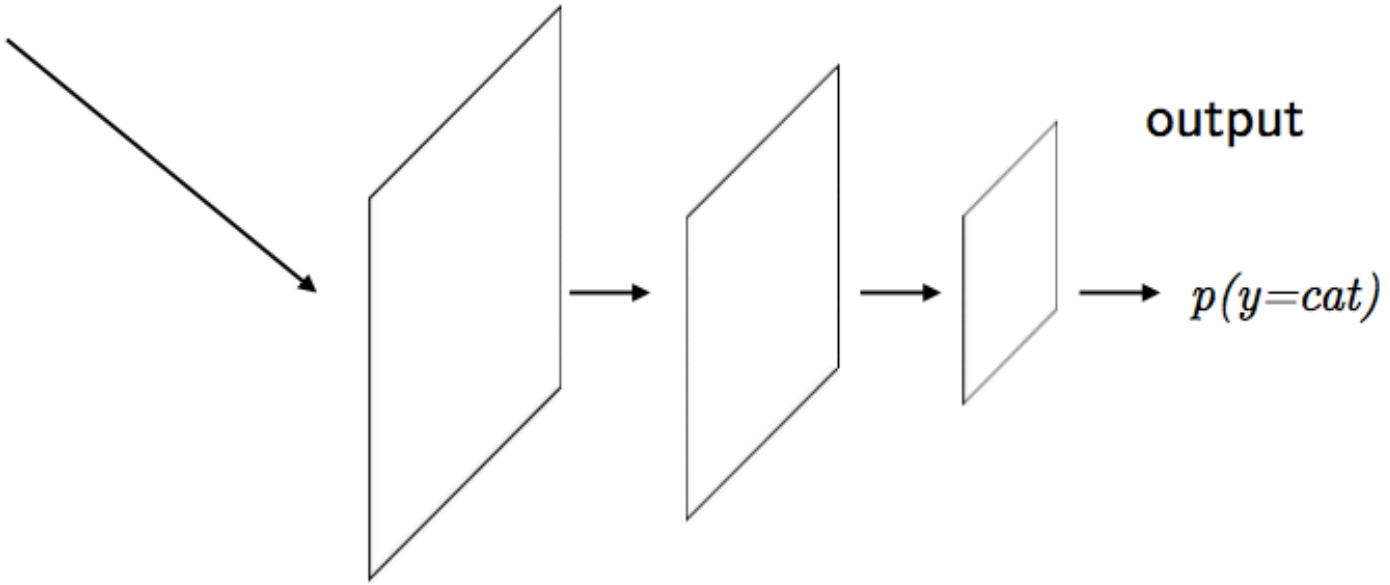
No response

Response (end point)

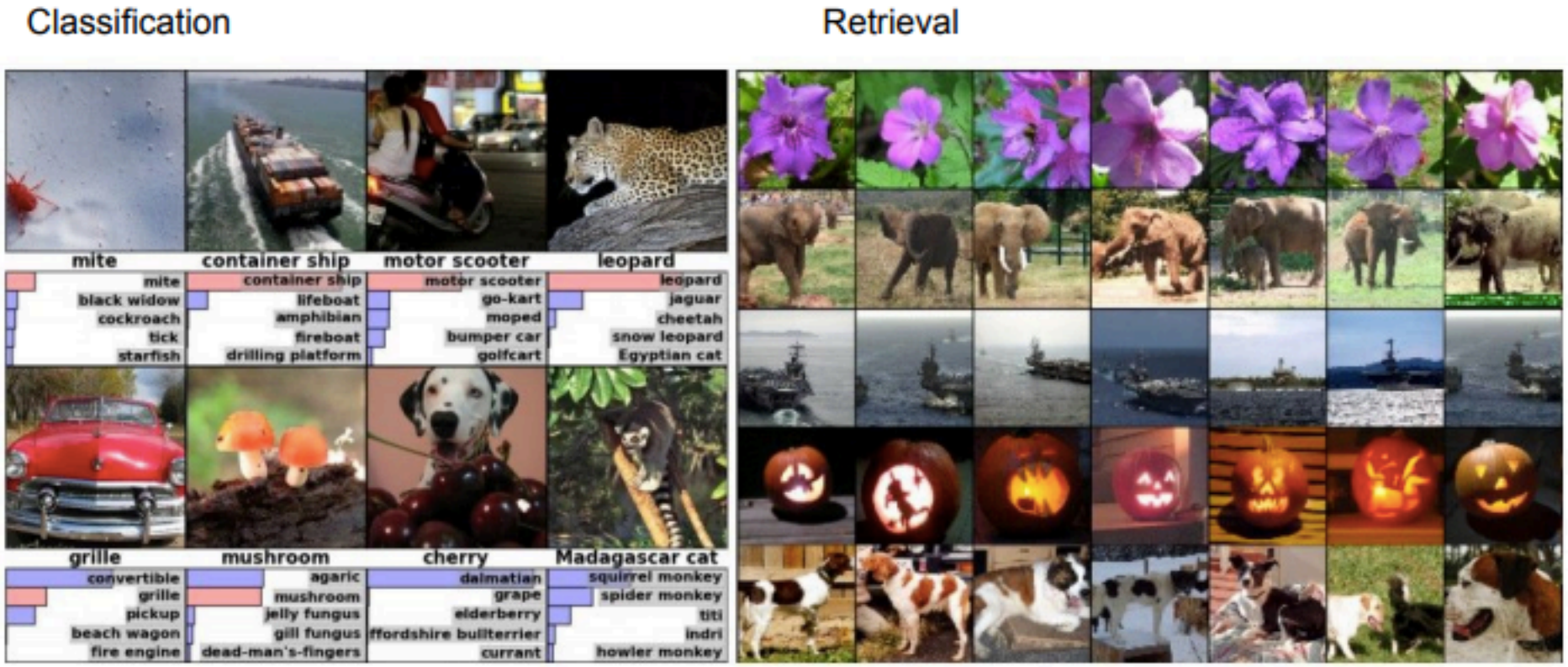
CNN for Image Classification

Image Source:
twitter.com%2Fcats&psig=AOvVaw30_o-PCM-K21DiMAJQimQ4&ust=1553887775741551

Image Source: https://www.pinterest.com/pin/244742560974520446



Some Applications of CNN



Figures copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

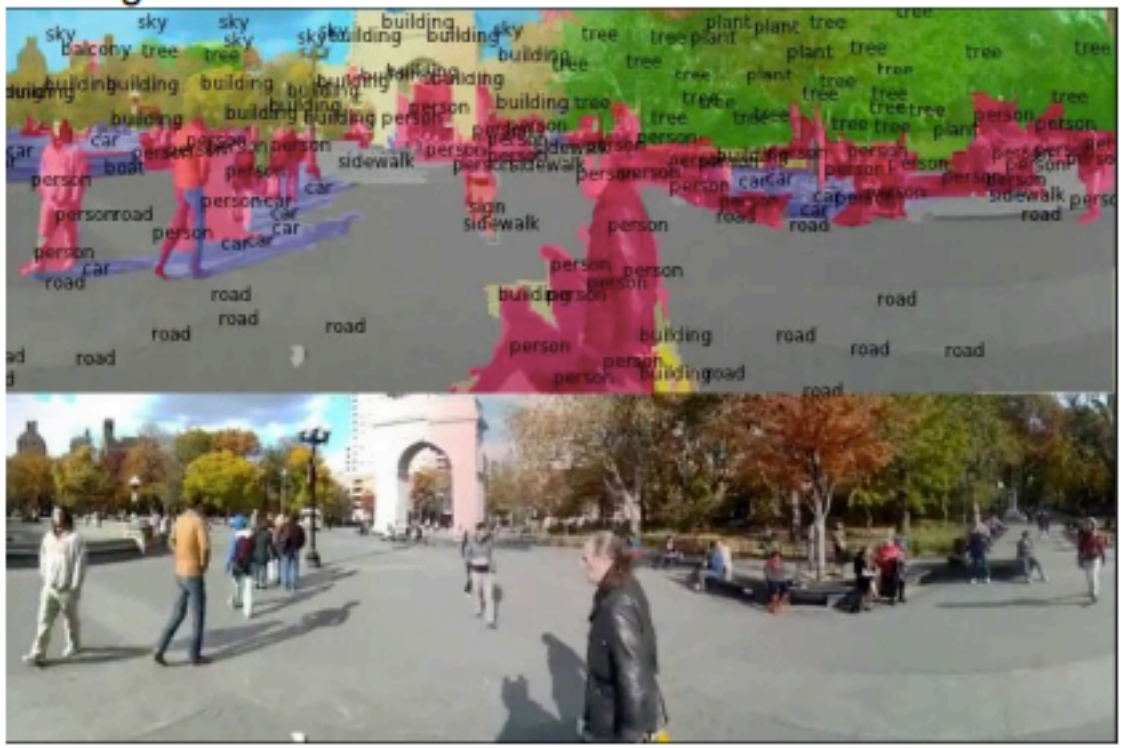
Some Applications of CNN

Detection

Figures copyright Shaoqing Ren, Kaiming He, Ross Girschick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

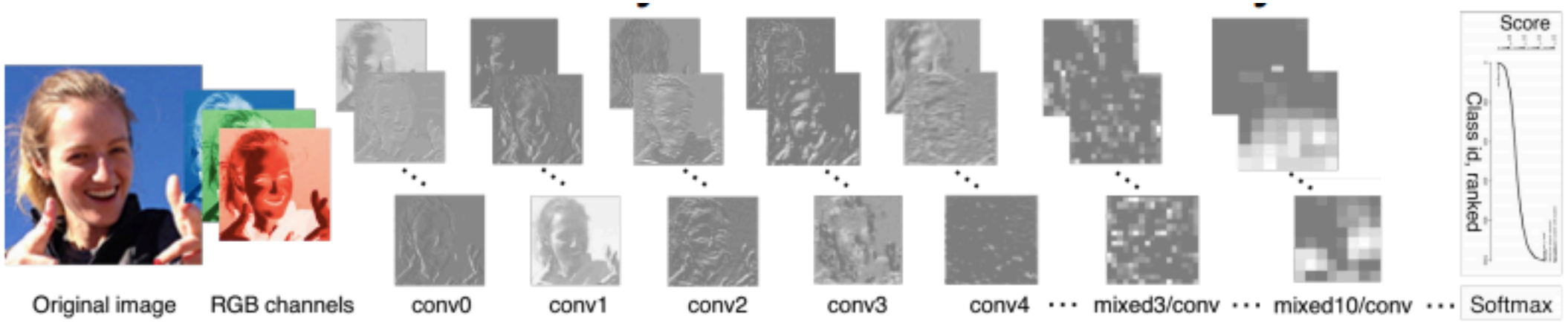
Segmentation



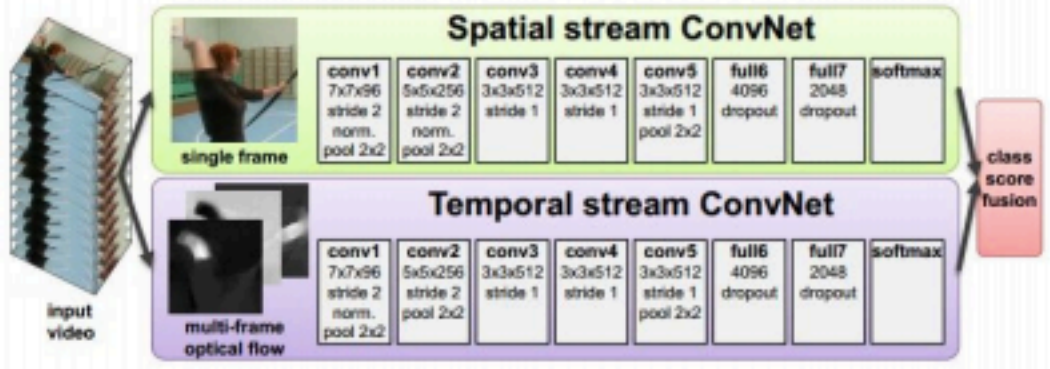
Figures copyright Clement Farabet, 2012. Reproduced with permission.

[Farabet et al., 2012]

Some Applications of CNN



[Taigman et al. 2014]



[Simonyan et al. 2014]

Figures copyright Simonyan et al., 2014.
Reproduced with permission.

Activations of inception-v3 architecture [Szegedy et al. 2015] to image of Emma McIntosh, used with permission. Figure and architecture not from Taigman et al. 2014.

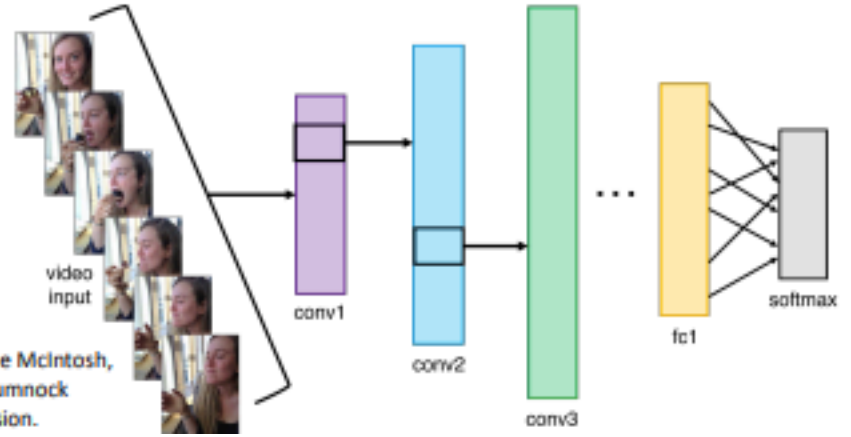
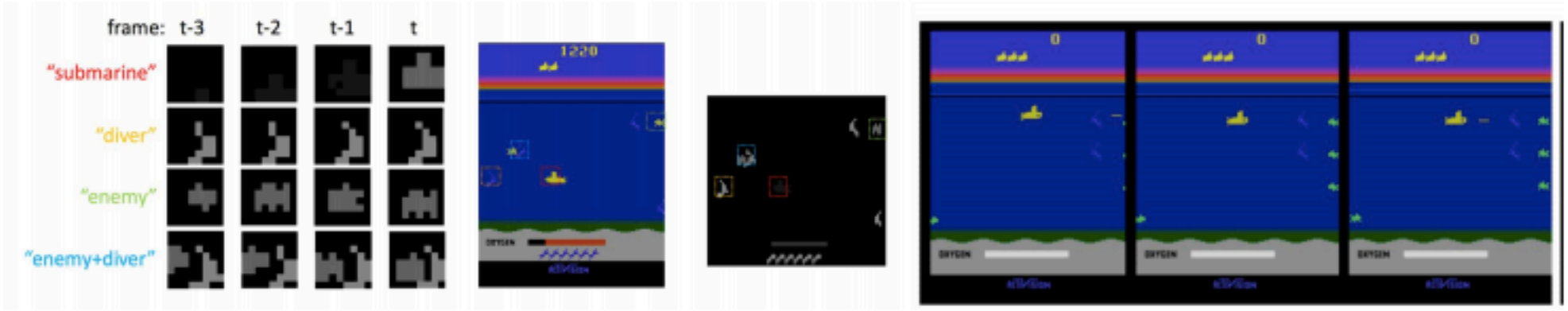


Illustration by Lane McIntosh, photos of Katie Cumnock used with permission.

Some Applications of CNN

Images are examples of pose estimation, not actually from Toshev & Szegedy 2014. Copyright Lane McIntosh.

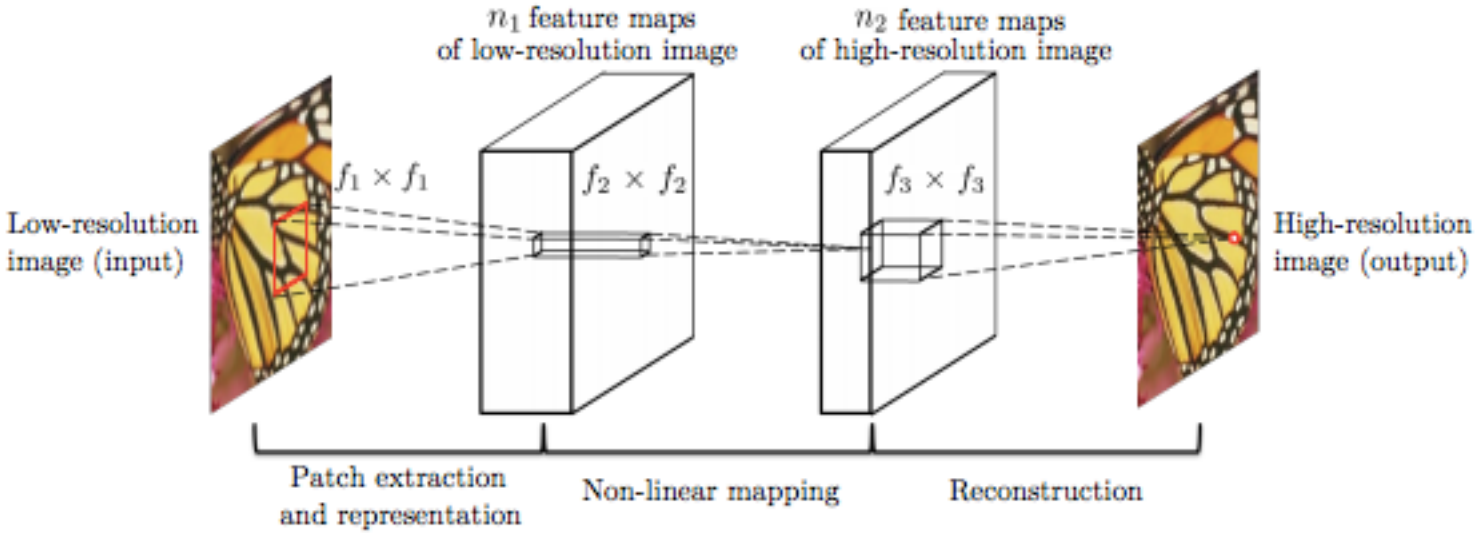
[Toshev, Szegedy 2014]



[Guo et al. 2014]

Figures copyright Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard Lewis, and Xiaoshi Wang, 2014. Reproduced with permission.

Some Applications of CNN



Some Applications of CNN

No errors

A white teddy bear sitting in the grass

Minor errors

A man in a baseball uniform throwing a ball

Somewhat related

A woman is holding a cat in her hand

A man riding a wave on top of a surfboard

A cat sitting on a suitcase on the floor

A woman standing on a beach holding a surfboard

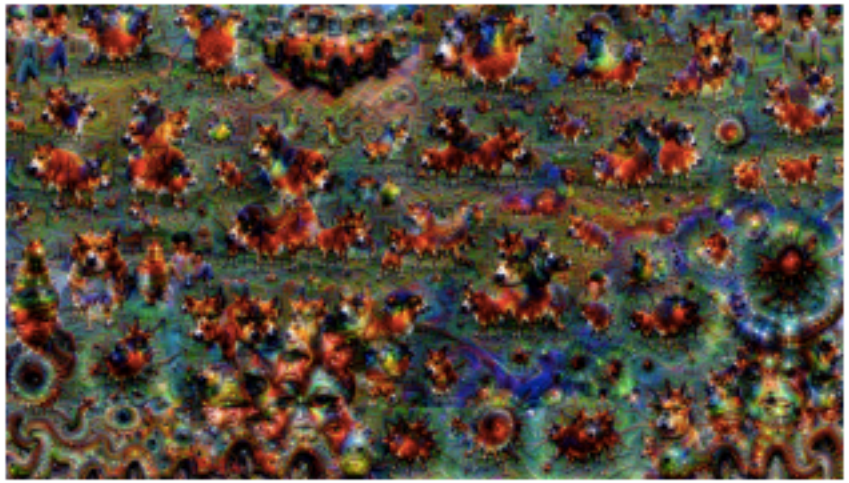
Image Captioning

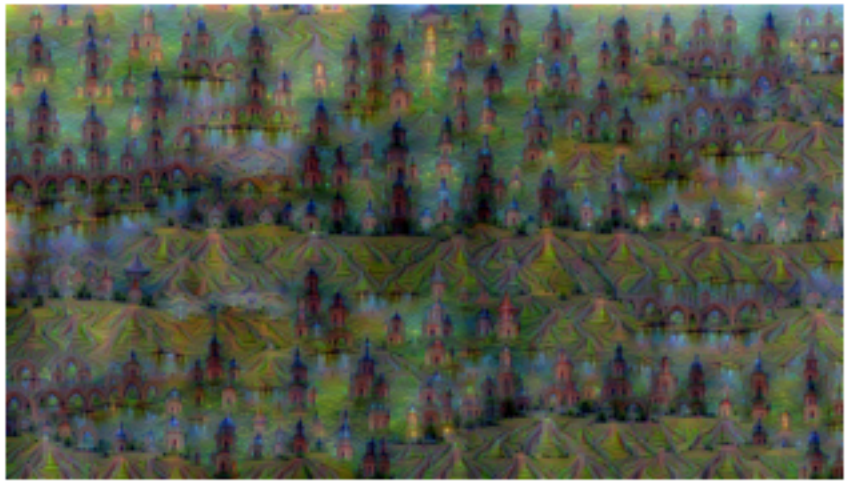
[Vinyals et al., 2015]
[Karpathy and Fei-Fei, 2015]

All images are CC0 Public domain:
https://pixabay.com/en/luggage-antique-cat-1643010/
https://pixabay.com/en/teddy-plush-bears-cute-teddy-bear-1623436/
https://pixabay.com/en/surf-wave-summer-sport-liforal-1668716/
https://pixabay.com/en/woman-female-model-portrait-adult-983967/
https://pixabay.com/en/handstand-lake-meditation-496008/
https://pixabay.com/en/baseball-player-shortstop-infield-1045263/

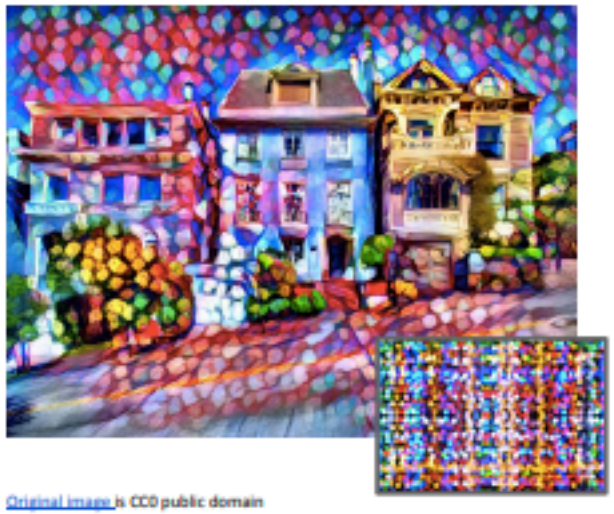
Captions generated by Justin Johnson using Neuraltalk2

Some Applications of CNN





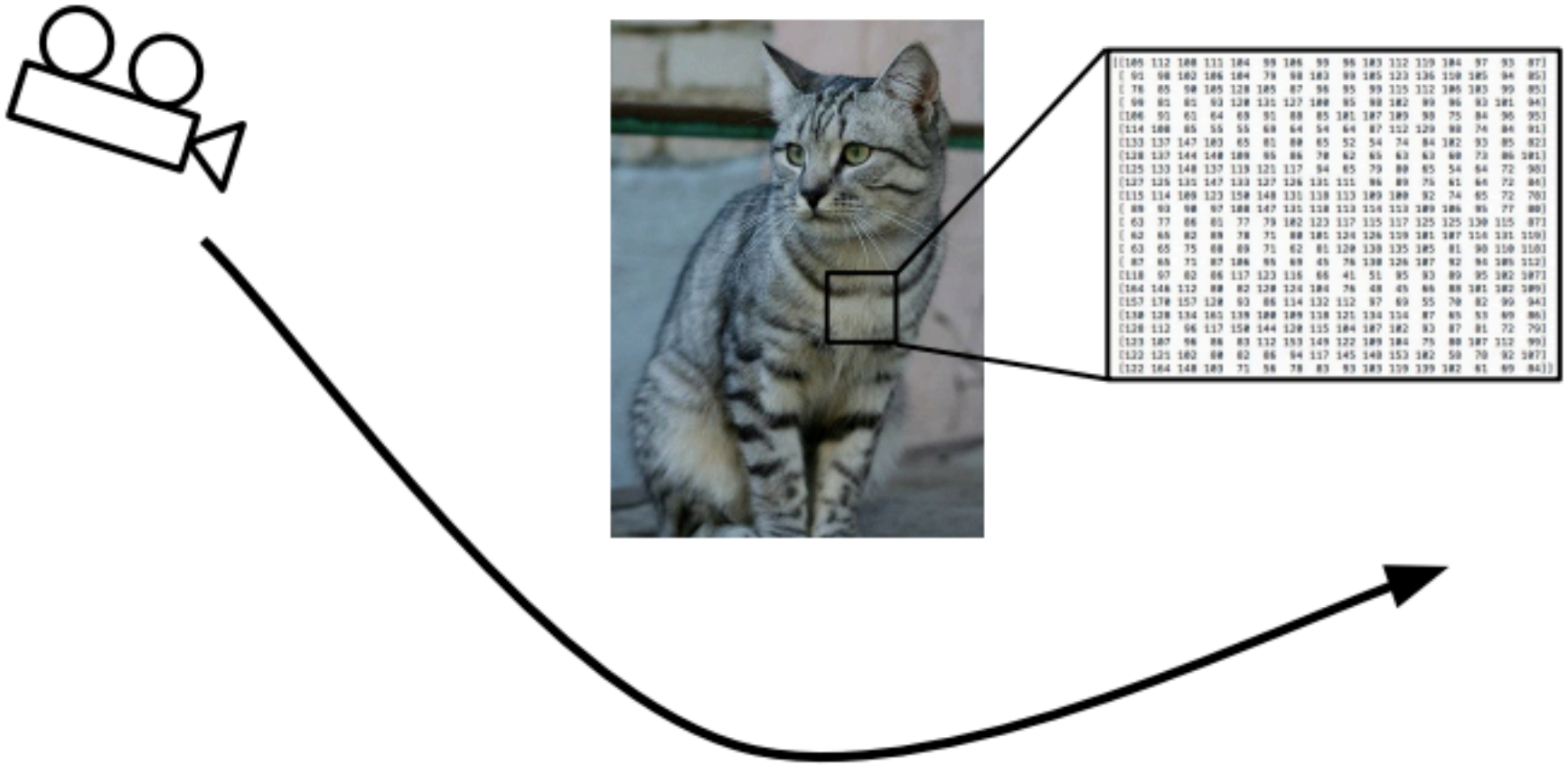
Figures copyright Justin Johnson, 2015. Reproduced with permission. Generated using the Inceptionism approach from a blog post by Google Research.



Original image is CC0 public domain
Starry Night and Tree Roots by Van Gogh are in the public domain
Bokeh image is in the public domain
Stylized images copyright Justin Johnson, 2017; reproduced with permission

Gatys et al, "Image Style Transfer using Convolutional Neural Networks", CVPR 2016
Gatys et al, "Controlling Perceptual Factors in Neural Style Transfer", CVPR 2017

Challenges: Viewpoint variation



All pixels change when the camera moves!

Challenges: Background Clutter

This image is CC0 1.0 public domain

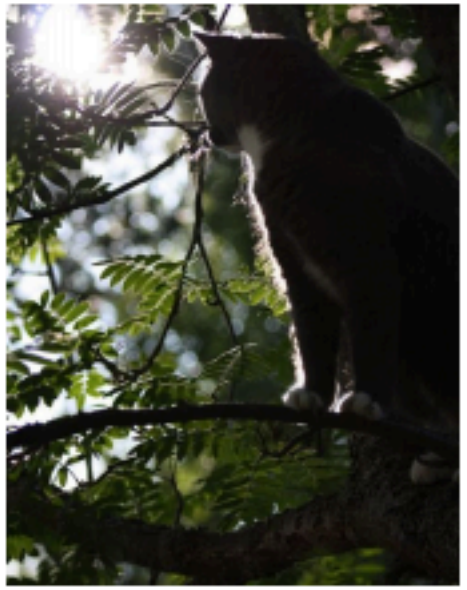
This image is CC0 1.0 public domain

Challenges: Illumination

This image is CC0 1.0 public domain

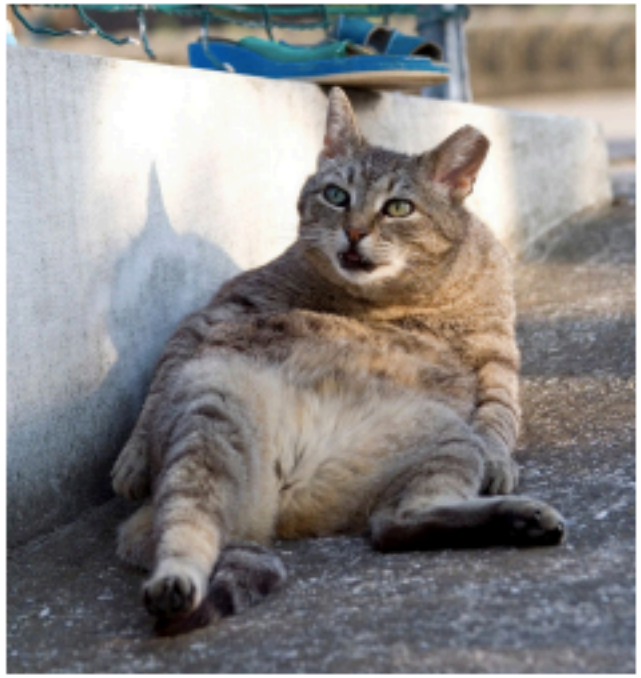
This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

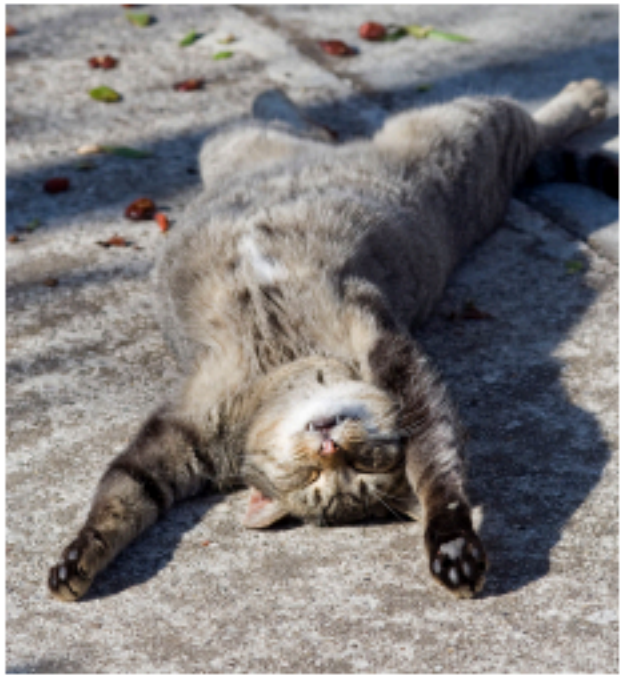


This image is CC0 1.0 public domain

Challenges: Deformation

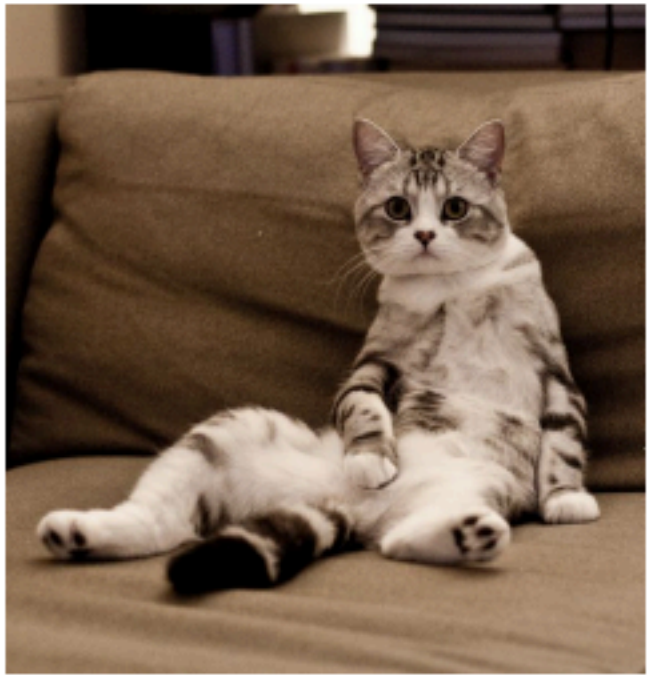


This image by Umberto Salvagnin is licensed under CC-BY 2.0



This image by Umberto Salvagnin is licensed under CC-BY 2.0

This image by sare bear is licensed under CC-BY 2.0



This image by Tom Thai is licensed under CC-BY 2.0

Challenges: Occlusion

This image is CC0 1.0 public domain

This image is CC0 1.0 public domain

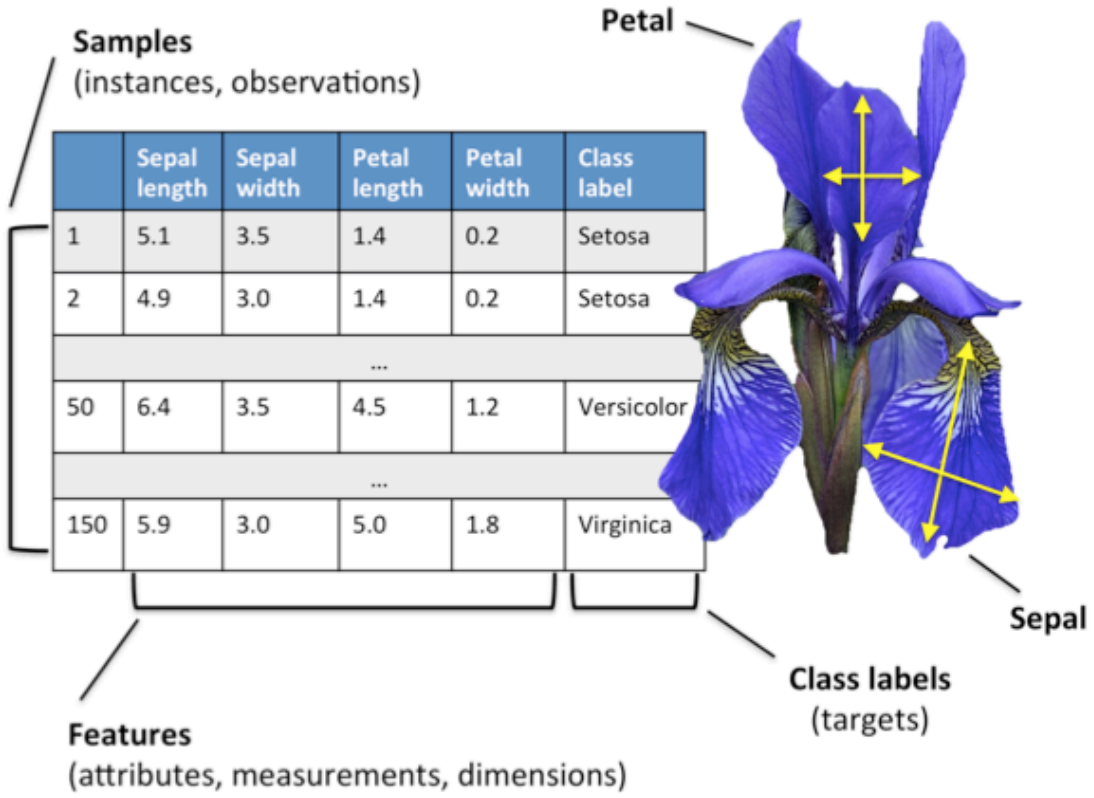
This image by jonsson is licensed under CC-BY 2.0

Challenges: Intraclass variation

This image is CC0 1.0 public domain

Traditional Approaches

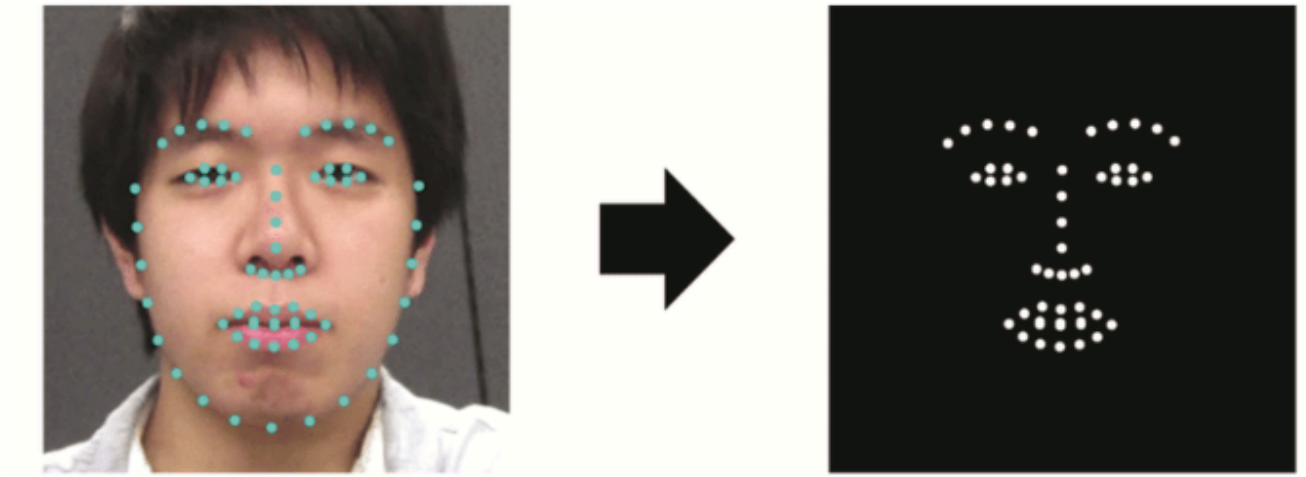
a) Use hand-engineered features



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Traditional Approaches

a) Use hand-engineered features



(a) Detected facial keypoints (b) Facial organ keypoints

Sasaki, K., Hashimoto, M., & Nagata, N. (2016). Person Invariant Classification of Subtle Facial Expressions Using Coded Movement Direction of Keypoints. In *Video Analytics. Face and Facial Expression Recognition and Audience Measurement* (pp. 61-72). Springer, Cham.

Traditional Approaches

b) Preprocess images (centering, cropping, etc.)

Traditional Approaches

- Using filters (kernels) isn't something new. It came from traditional computer vision.

Main Concepts Behind Convolutional Neural Networks

- Sparse-connectivity: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, in the case of multi-layer perceptrons.)
- Parameter-sharing: The same weights are used for different patches of the input image.

Convolutional Neural Networks

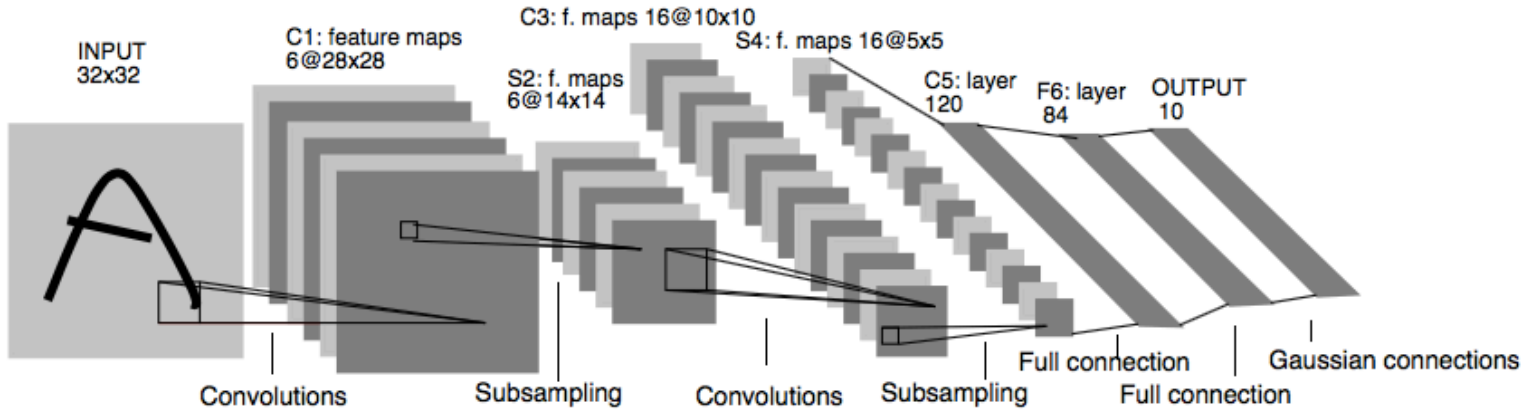


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner: Gradient Based Learning Applied to Document Recognition, Proceedings of IEEE, 86(11):2278–2324, 1998.

Convolutional Neural Networks

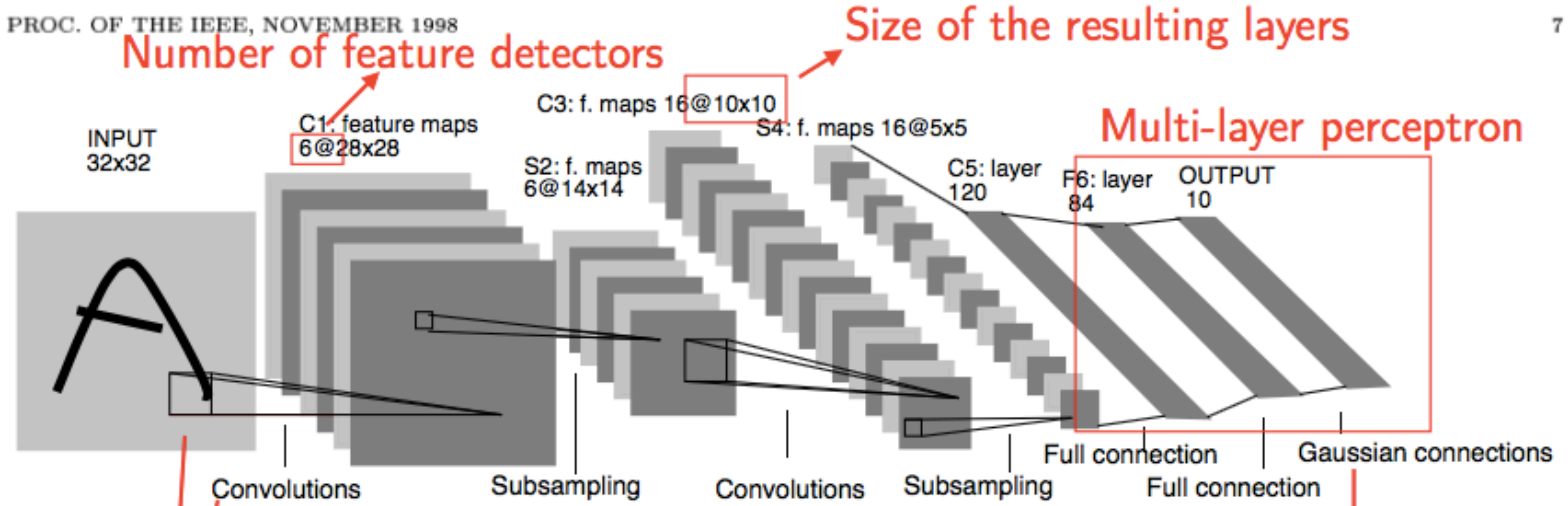


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

nowadays called "pooling"

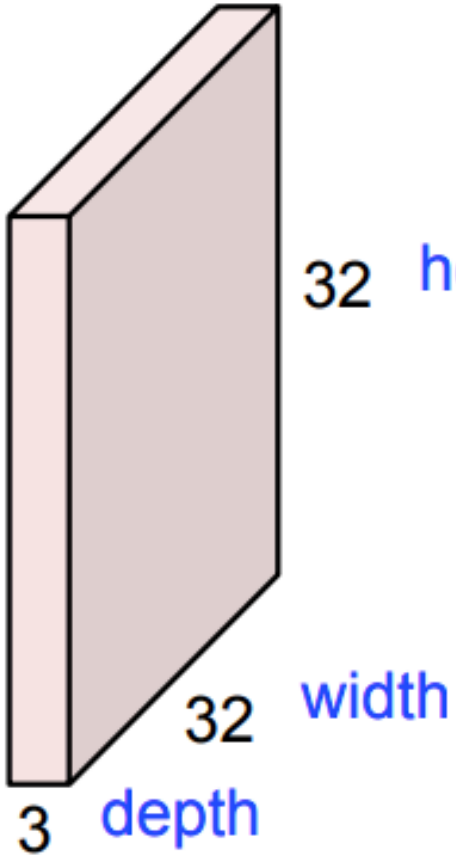
basically a fully-connected layer + MSE loss
(nowadays better to use
fc-layer + softmax
+ cross entropy

"Feature detectors" (weight matrices)
that are being reused ("weight sharing")
=> also called "kernel" or "filter"

Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner: Gradient Based Learning Applied to Document Recognition, Proceedings of IEEE, 86(11):2278–2324, 1998.

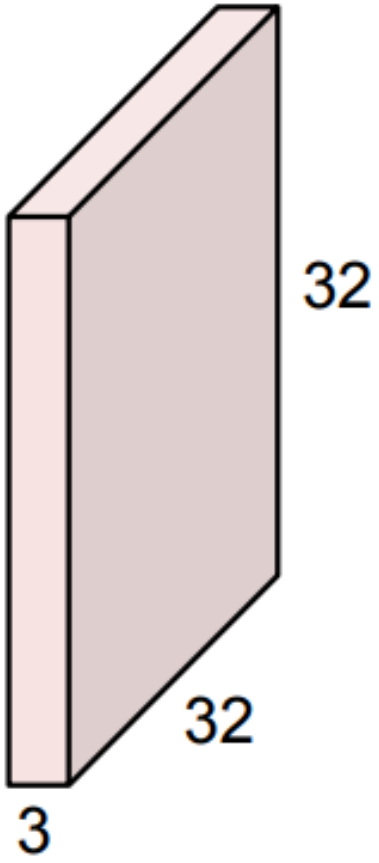
Convolution Layer

32x32x3 image -> preserve spatial structure

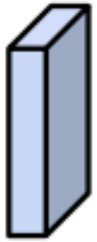


Convolution Layer

32x32x3 image



5x5x3 filter

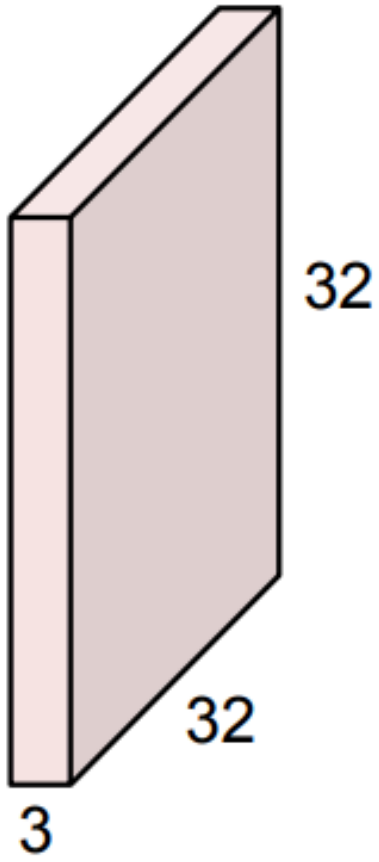


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

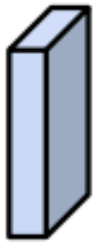
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

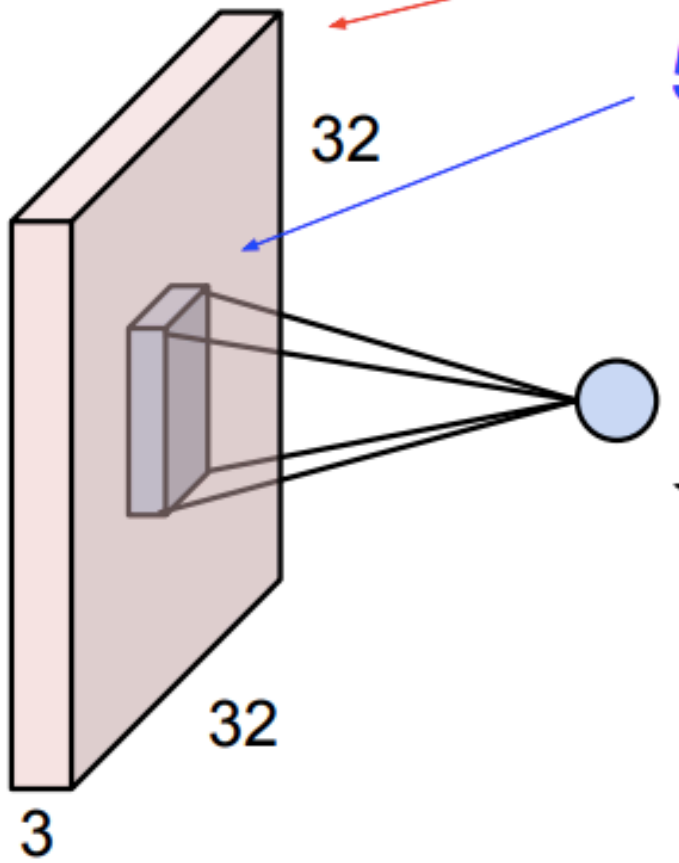


5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially, computing dot products”

Convolution Layer



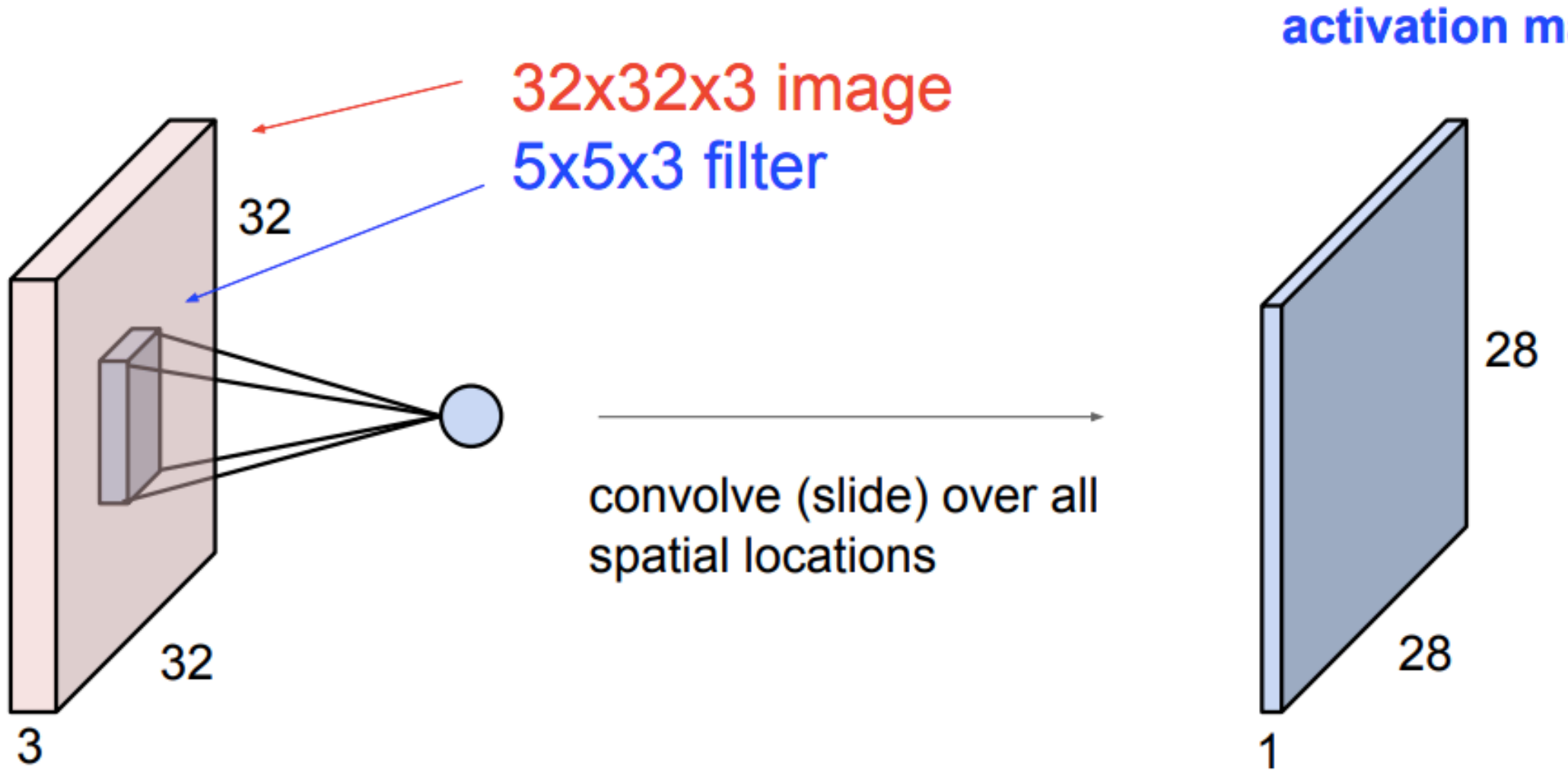
32x32x3 image

5x5x3 filter w

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

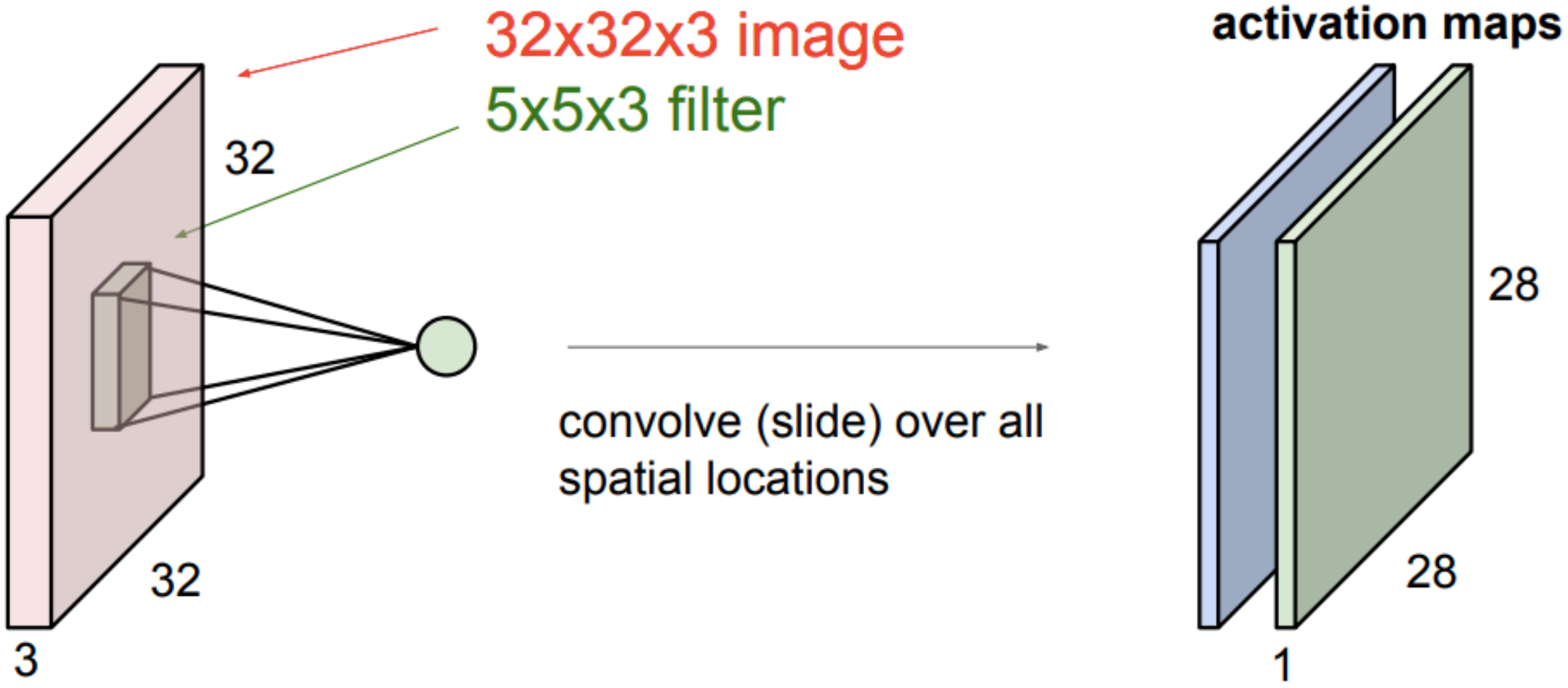
$$w^T x + b$$

Convolution Layer

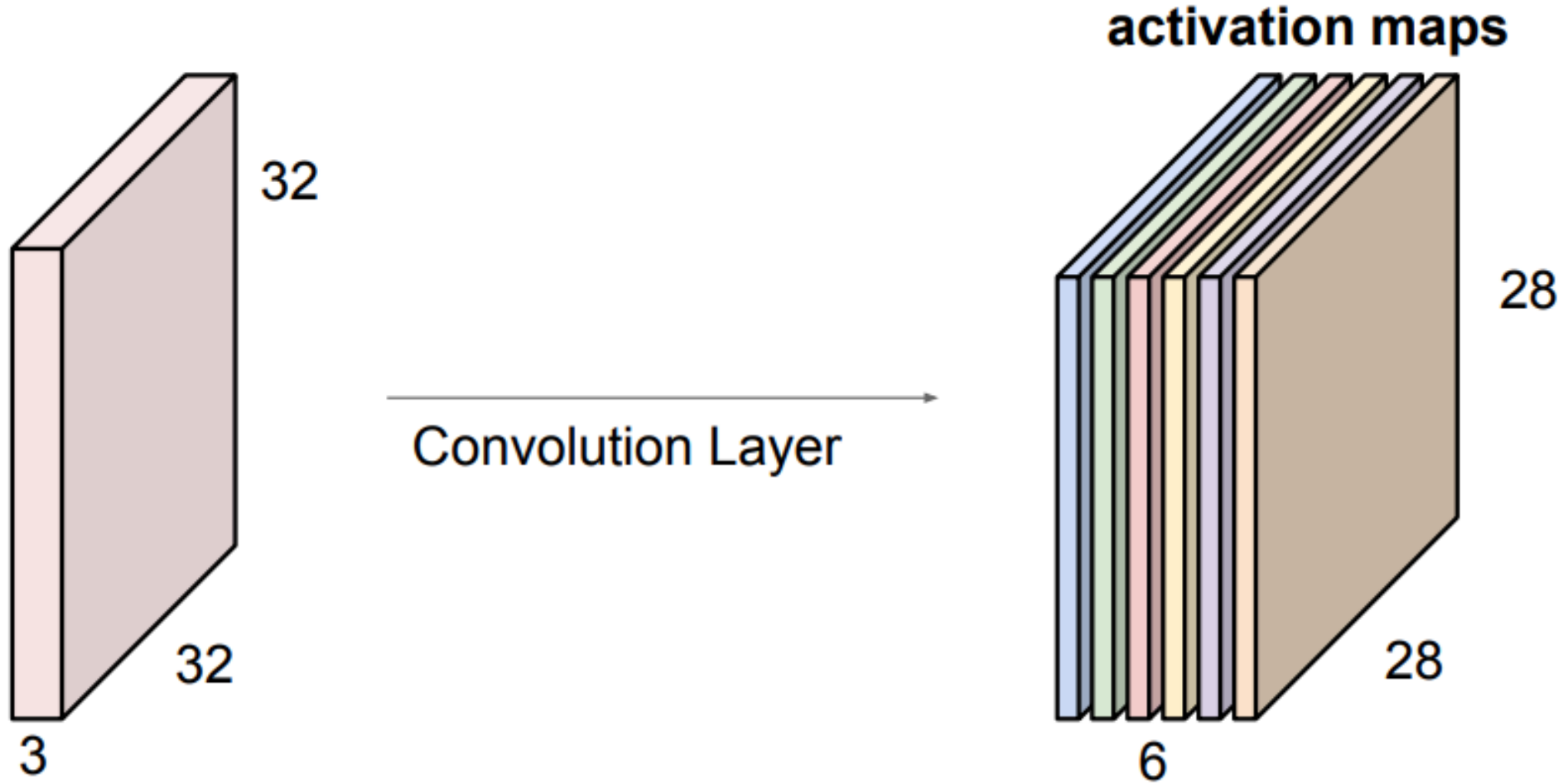


Convolution Layer

consider a second, green filter

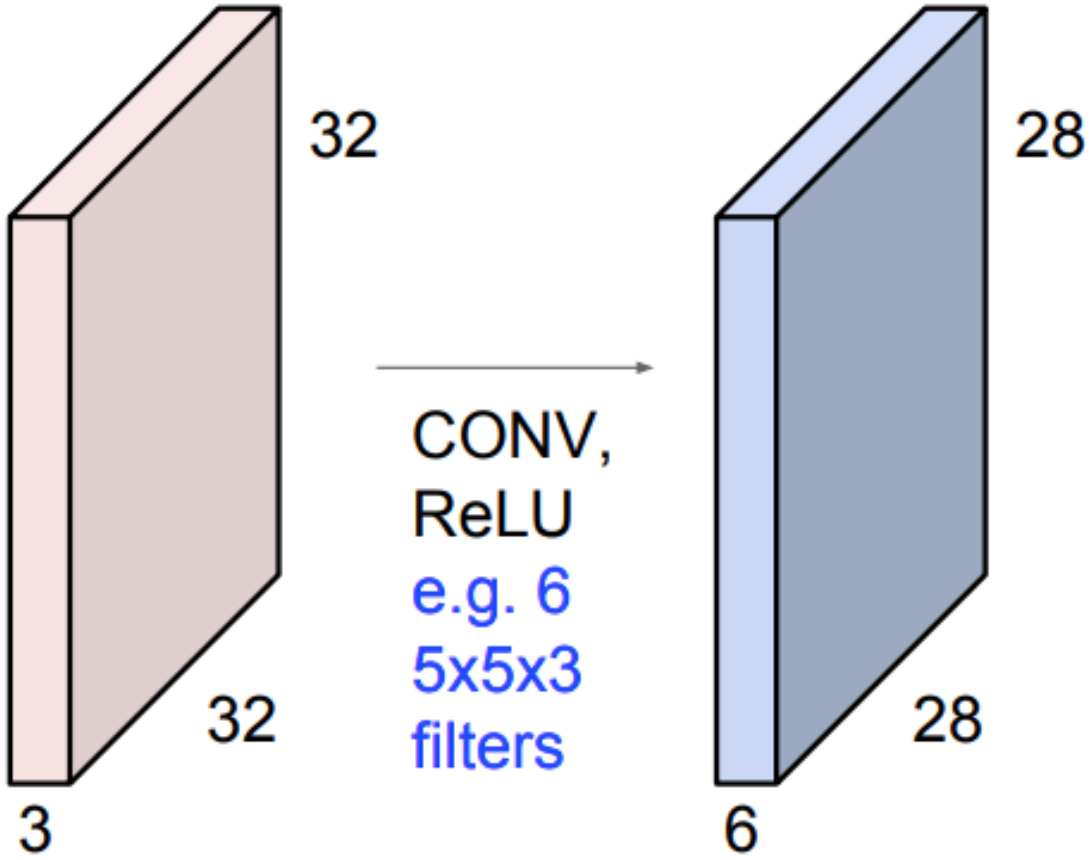


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

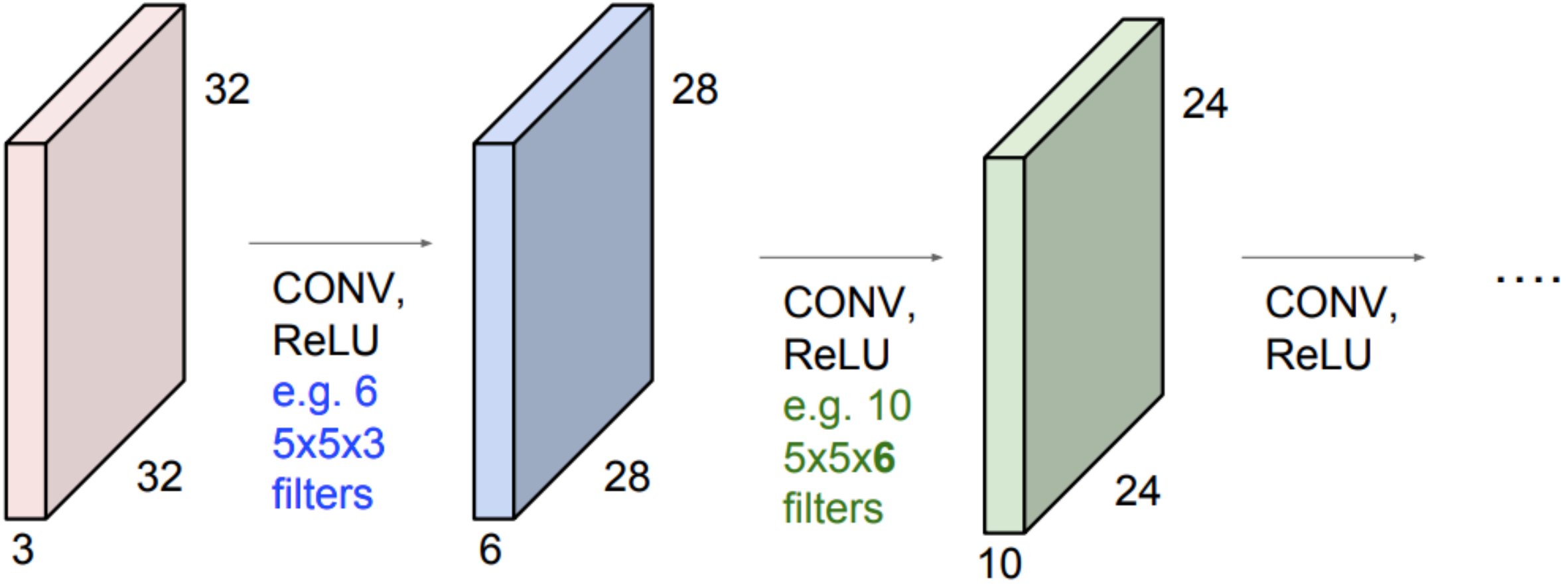


We stack these up to get a "new image" of size 28x28x6!

Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Hidden Layers

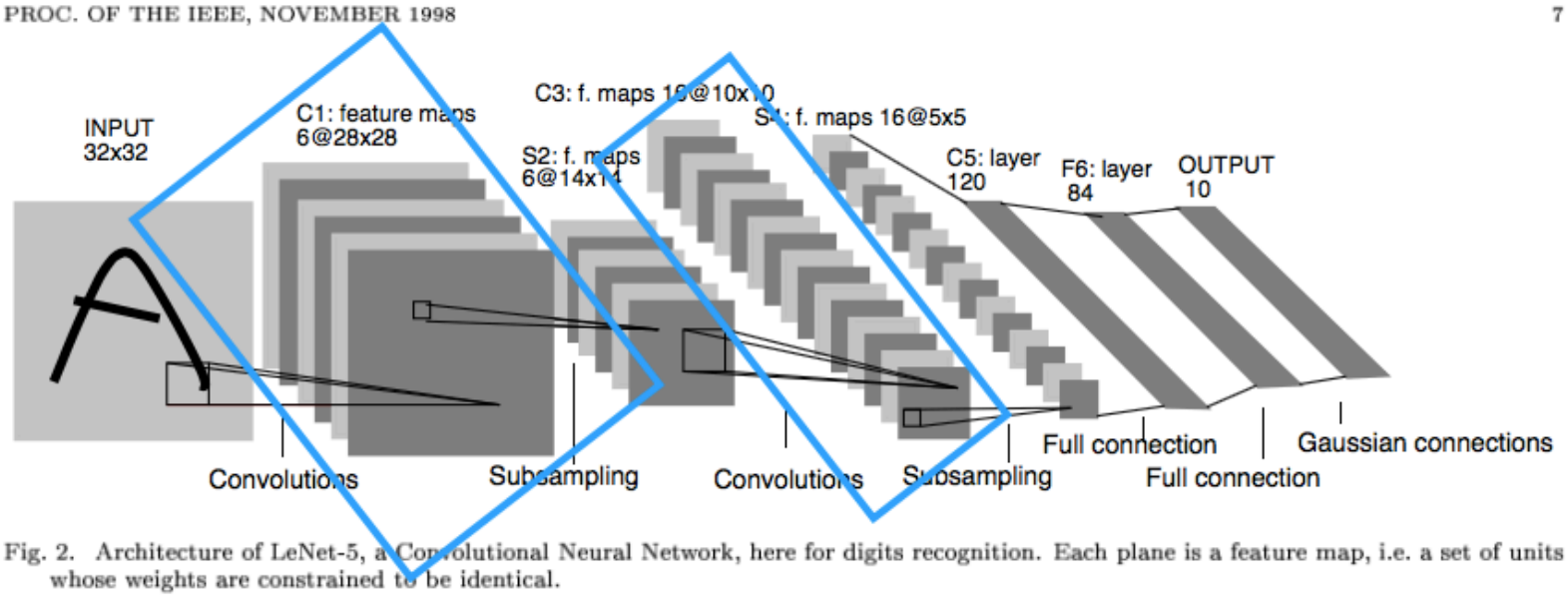
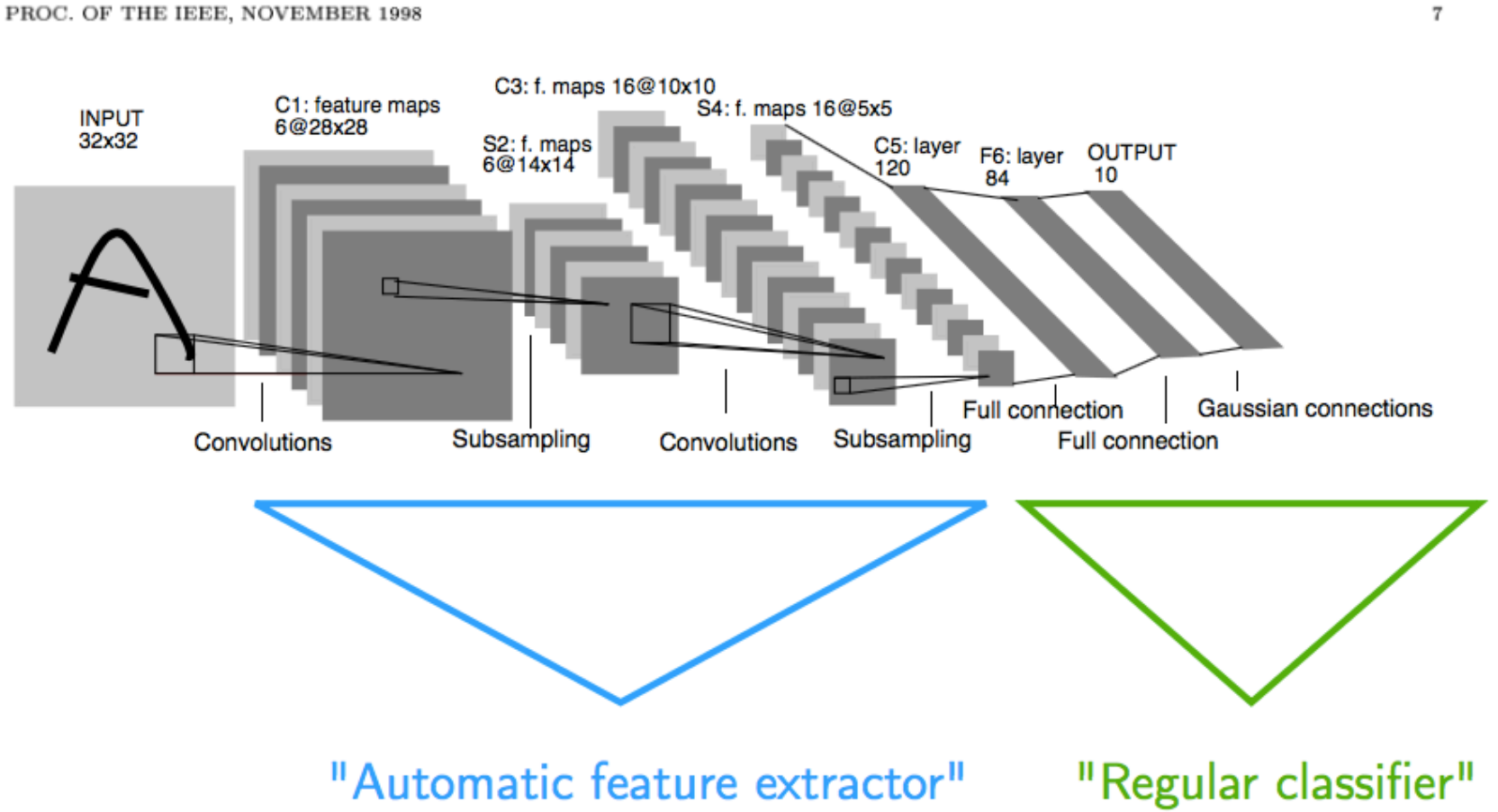


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

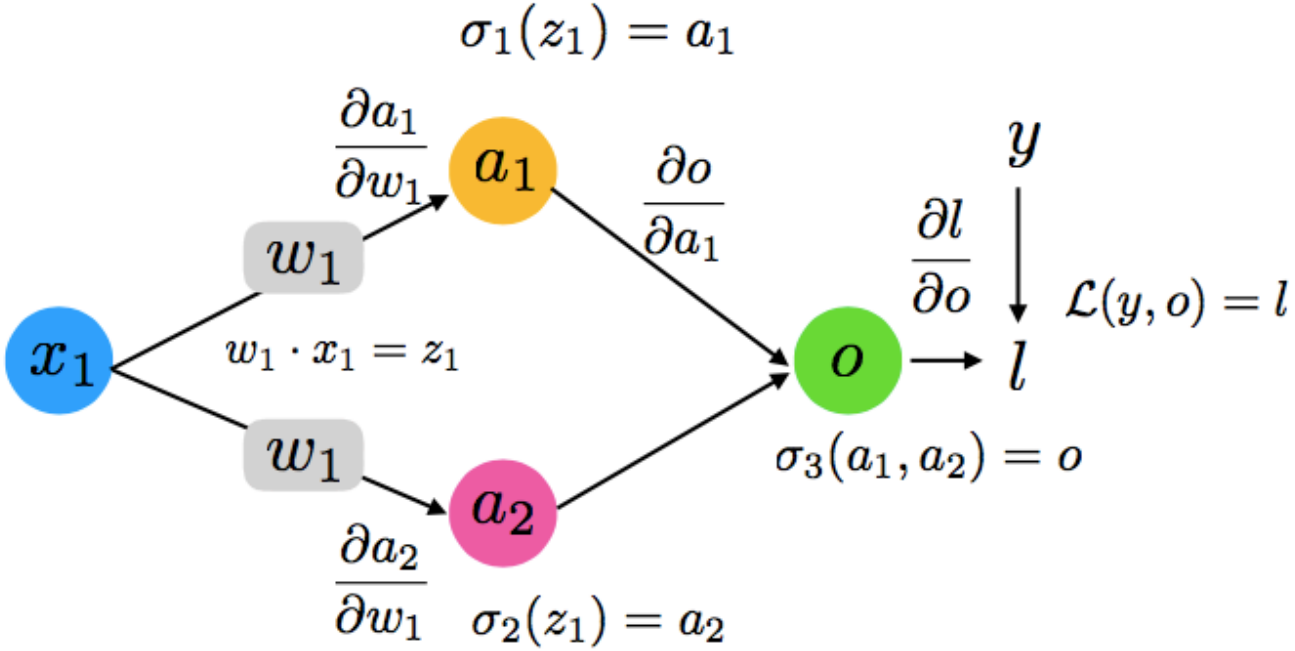
Each "bunch" of feature maps represents one hidden layer in the neural network.

Counting the FC layers, this network has 5 layers

Hidden Layers



Backpropagation in CNN



Upper path

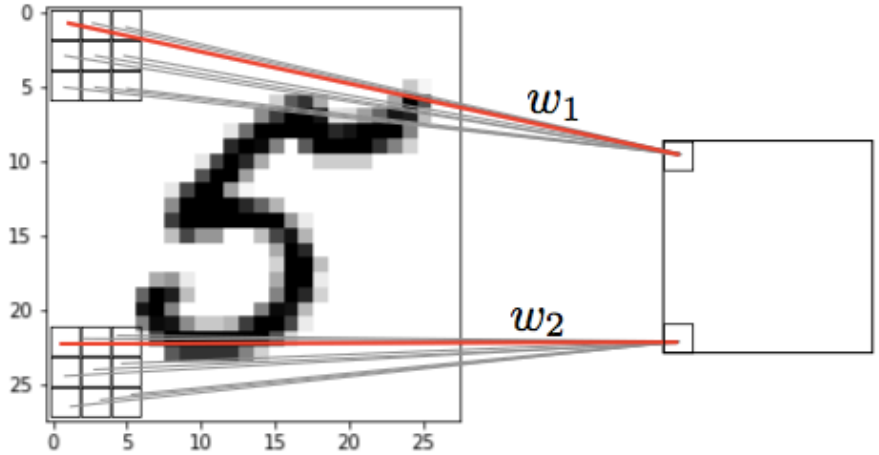
$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_1}$$ (multivariable chain rule)

Lower path

Backpropagation in CNN

Same overall concept as before: Multivariable chain rule, but now with an additional weight sharing constraint

Due to weight sharing: $w_1 = w_2$

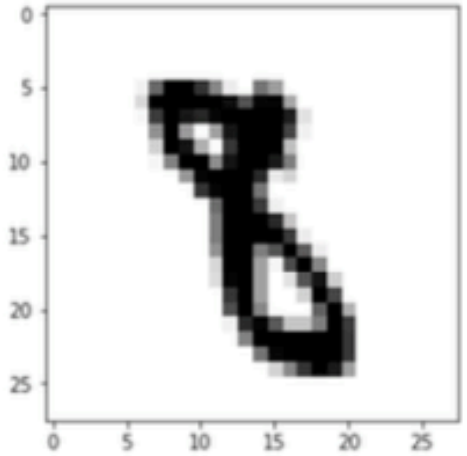


Optional averaging

weight update:

$$w_1 := w_2 := w_1 - \eta \cdot \frac{1}{2}\left(\frac{\partial \mathcal{L}}{\partial w_1} + \frac{\partial \mathcal{L}}{\partial w_2}\right)$$

Kernel Dimensions




```
a.shape
```

```
(1, 28, 28)
```

```
import torch
```

```
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=8,
                       kernel_size=(5, 5),
                       stride=(1, 1))
```

```
conv.weight.size()
```

```
torch.Size([8, 1, 5, 5])
```

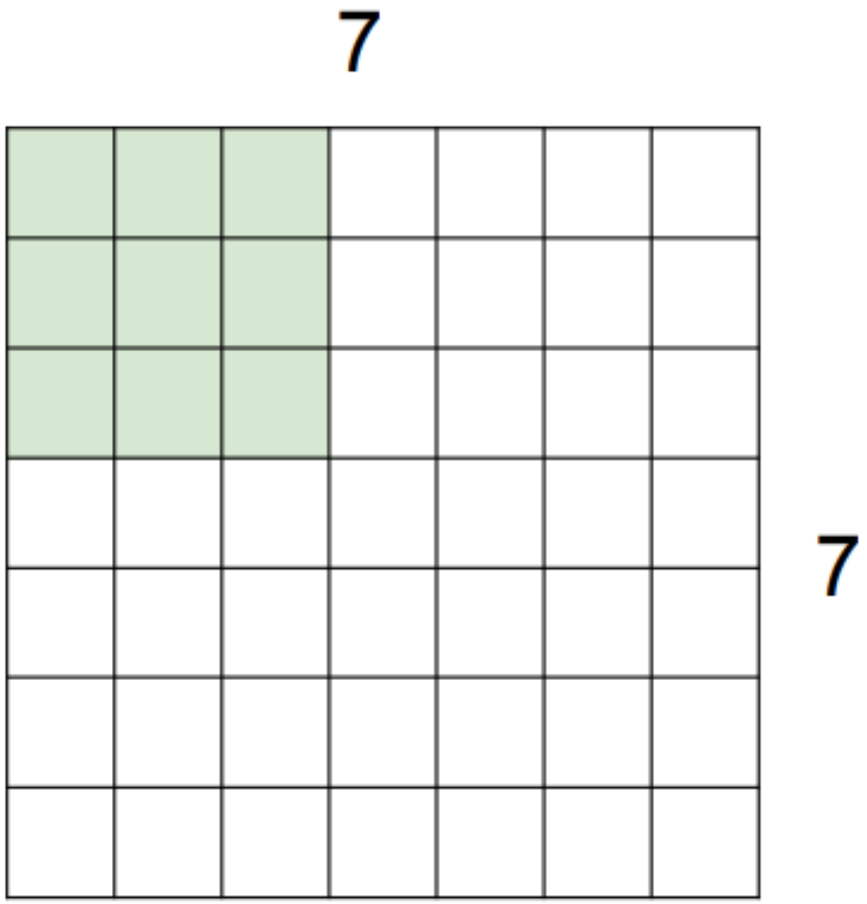
```
conv.bias.size()
```

```
torch.Size([8])
```

For a grayscale image with a 5x5 feature detector (kernel), we have the following dimensions (number of parameters to learn)

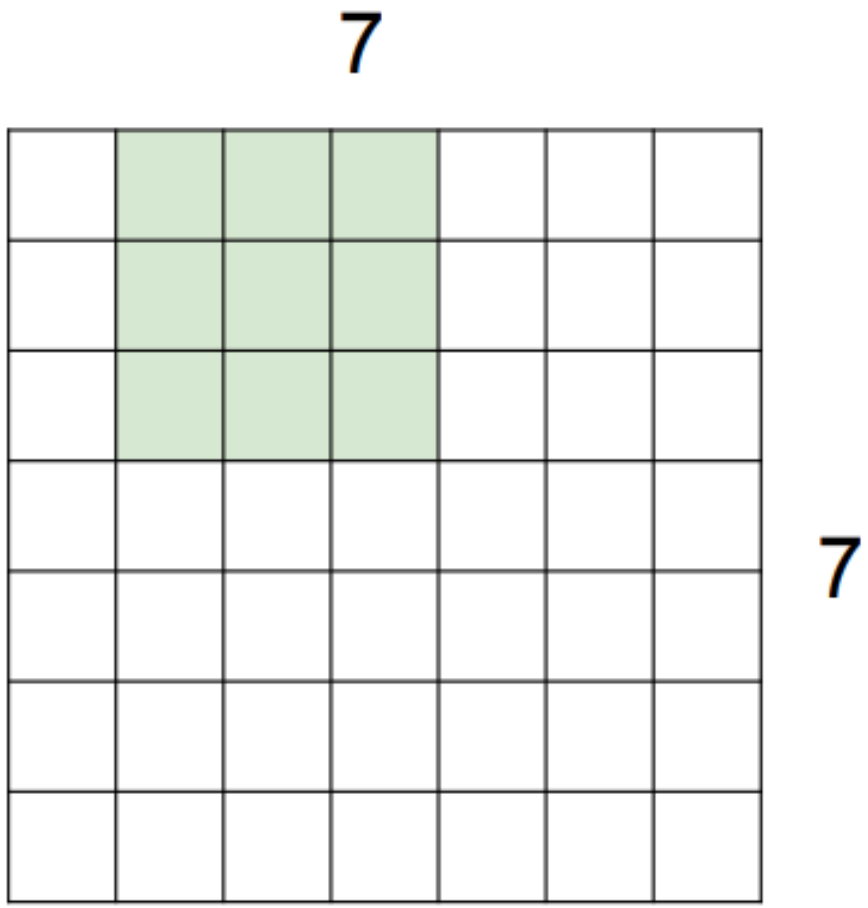
What do you think is the output size for this 28x28 image?

A closer look at spatial dimensions:



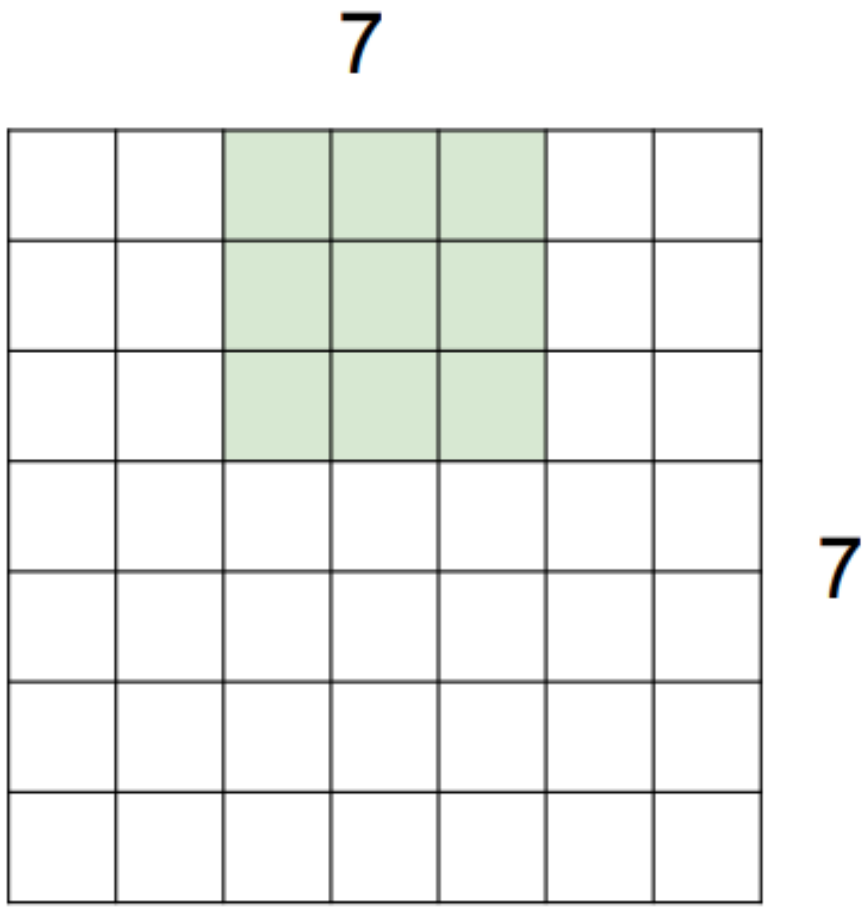
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



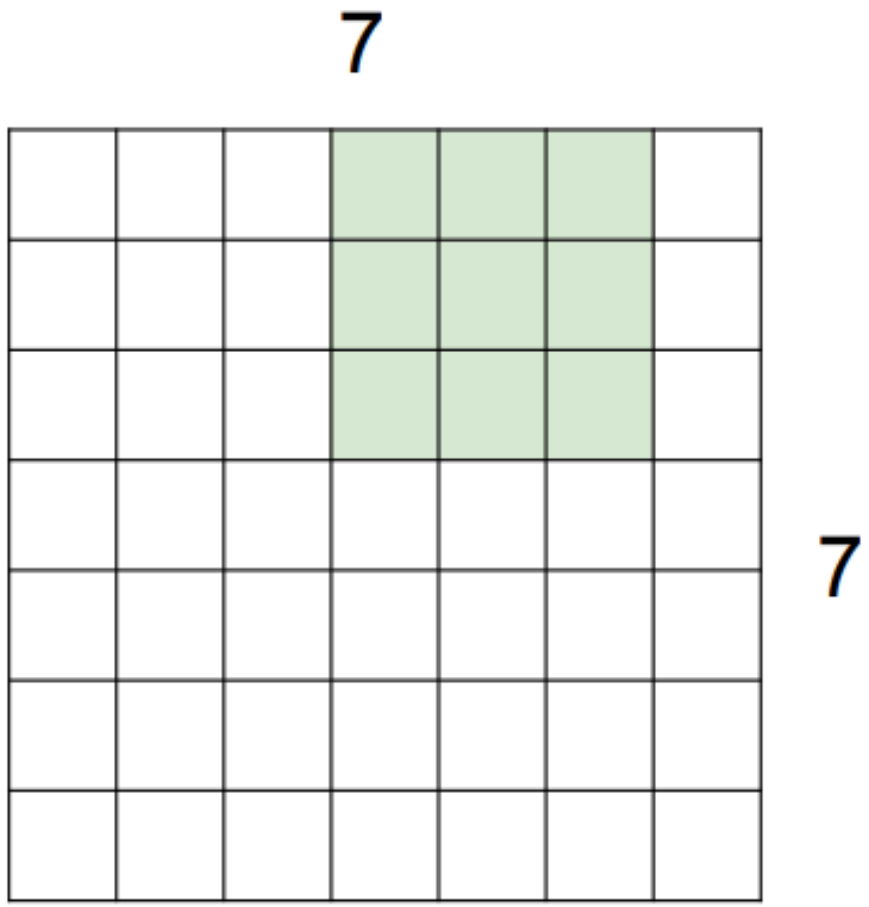
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



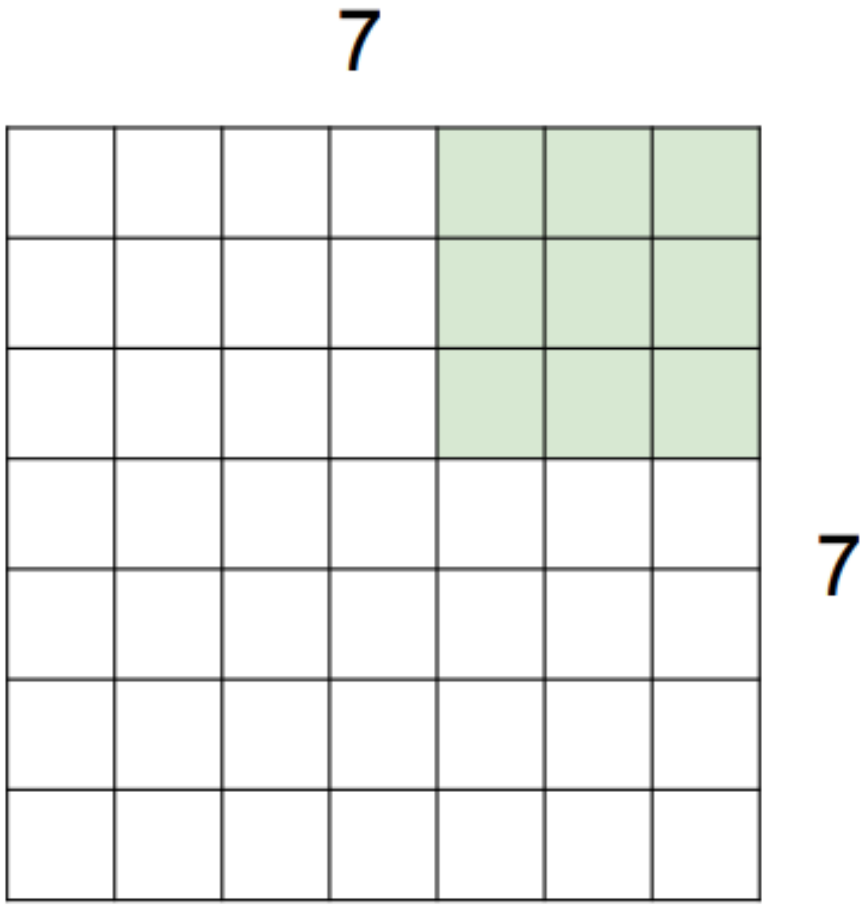
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

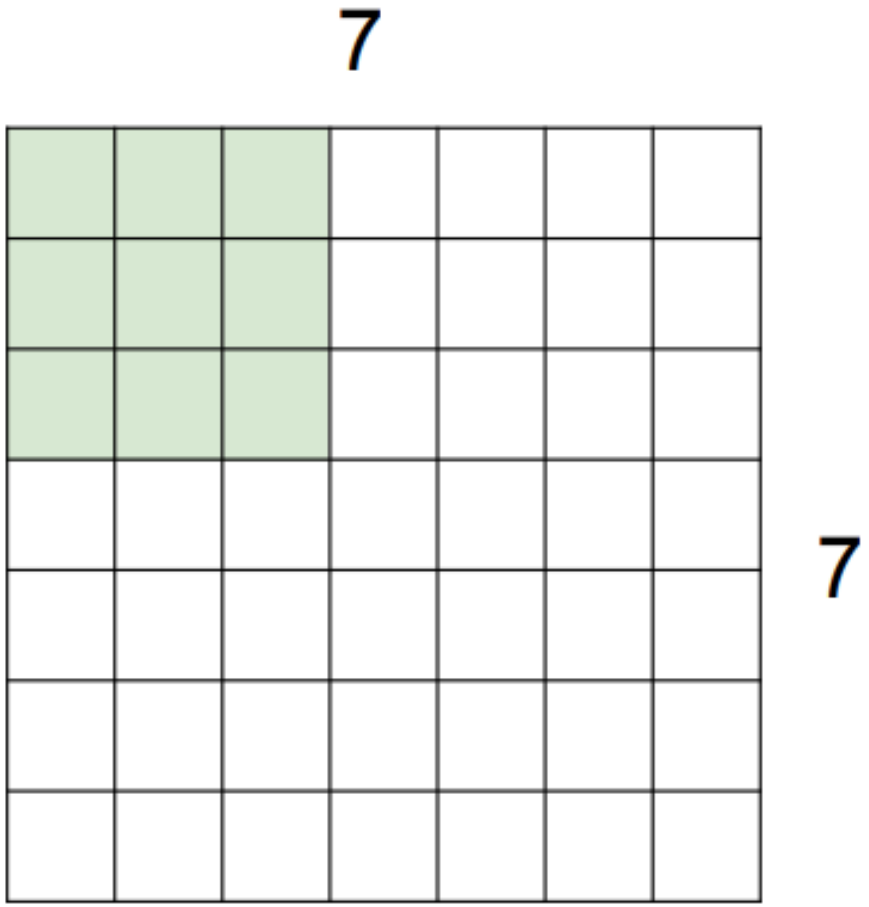
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

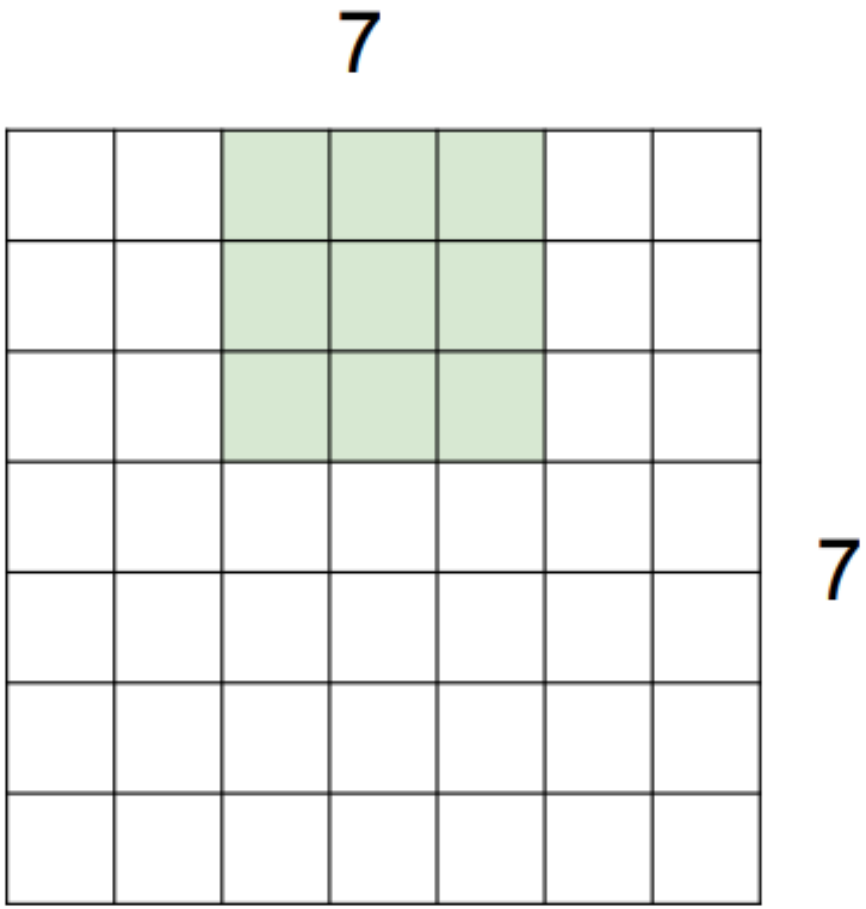
=> 5x5 output

A closer look at spatial dimensions:



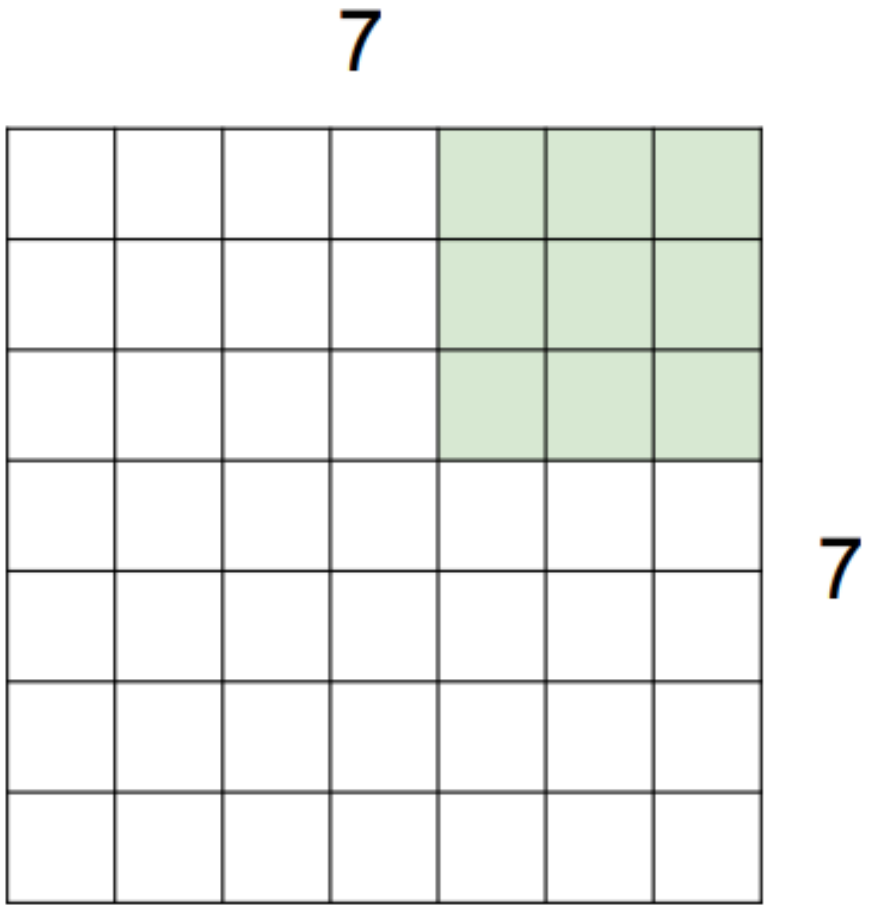
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



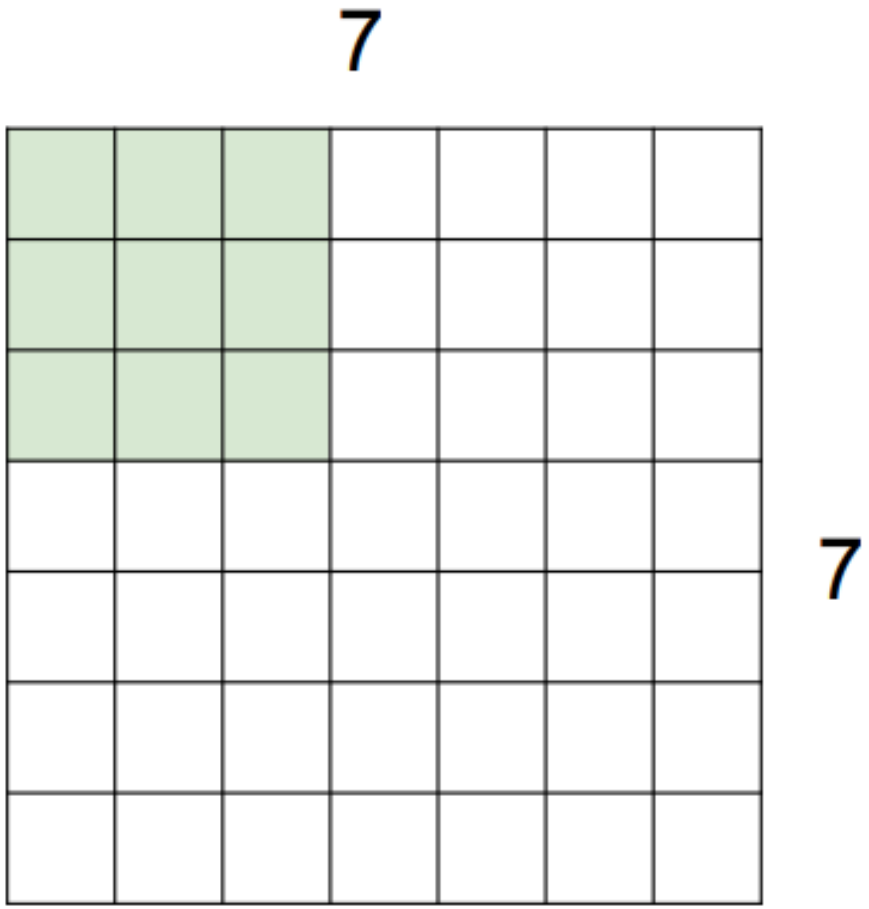
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



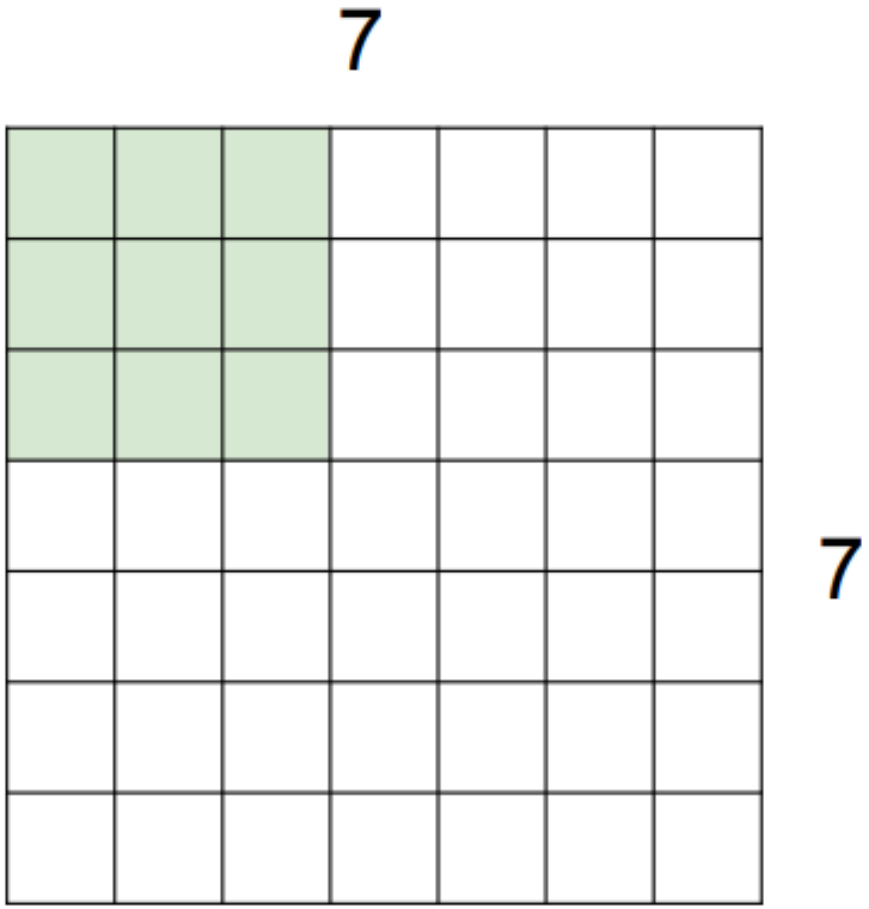
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:

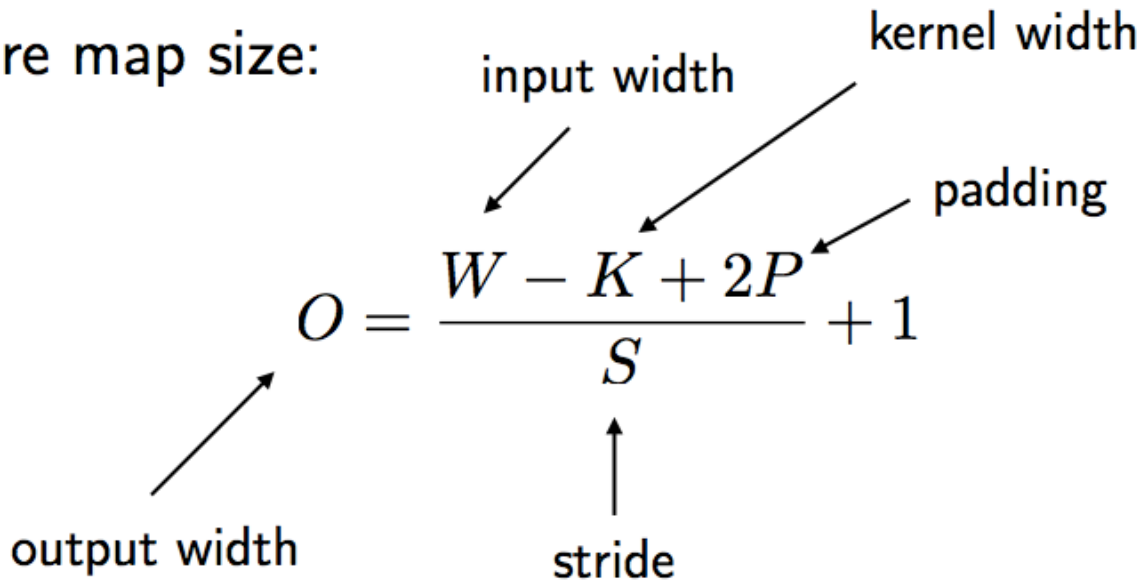


7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.

Size Before and After Convolutions

Feature map size:



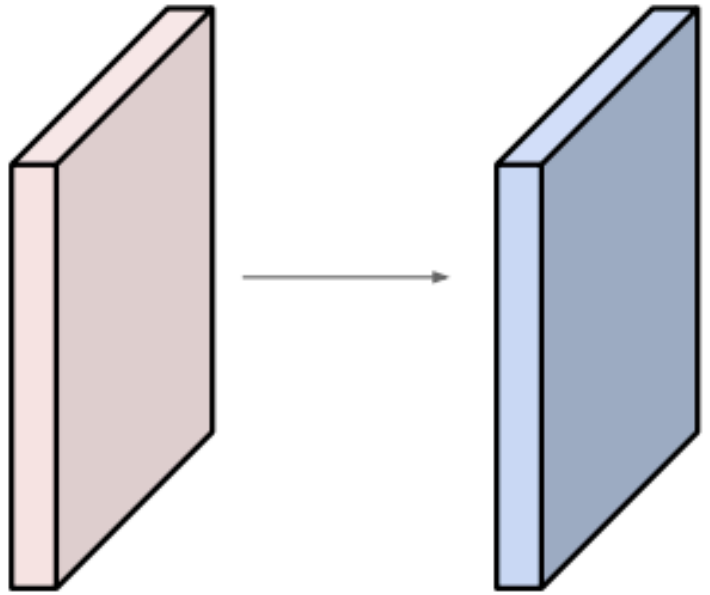
Padding Jargon

"valid" convolution: no padding (feature map may shrink)

"same" convolution: padding such that the output size is equal to the input size

Common kernel size conventions:
3x3, 5x5, 7x7 (sometimes 1x1 in later layers to reduce channels)

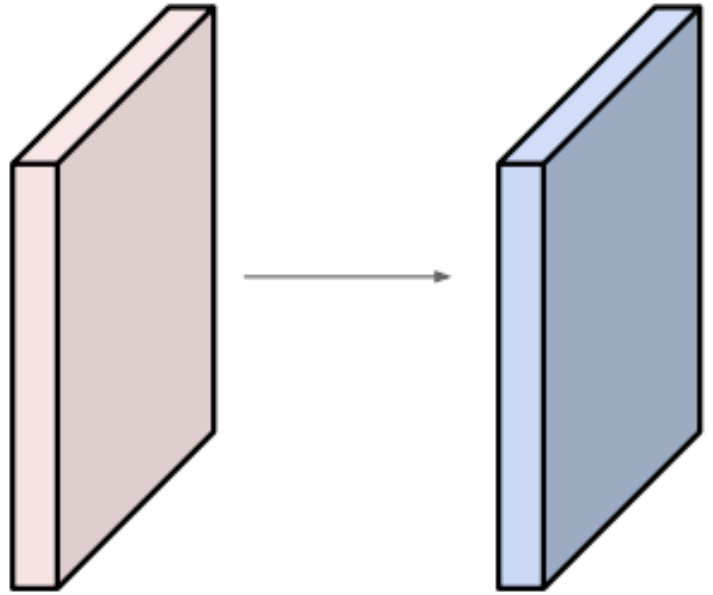
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?

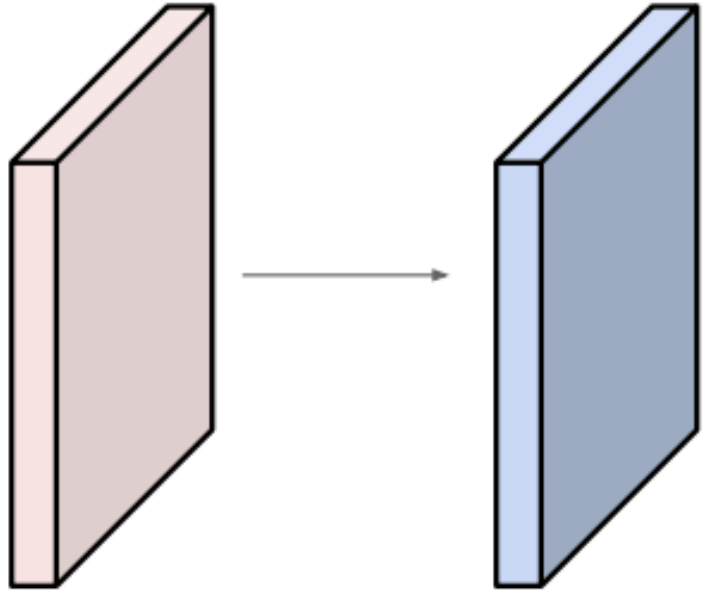
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size:
(32+2*2-5)/1+1 = 32 spatially, so
32x32x10

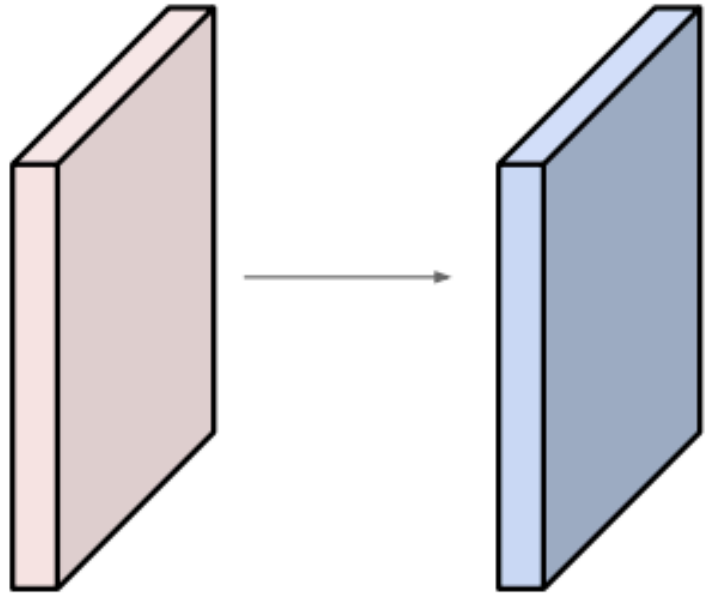
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?

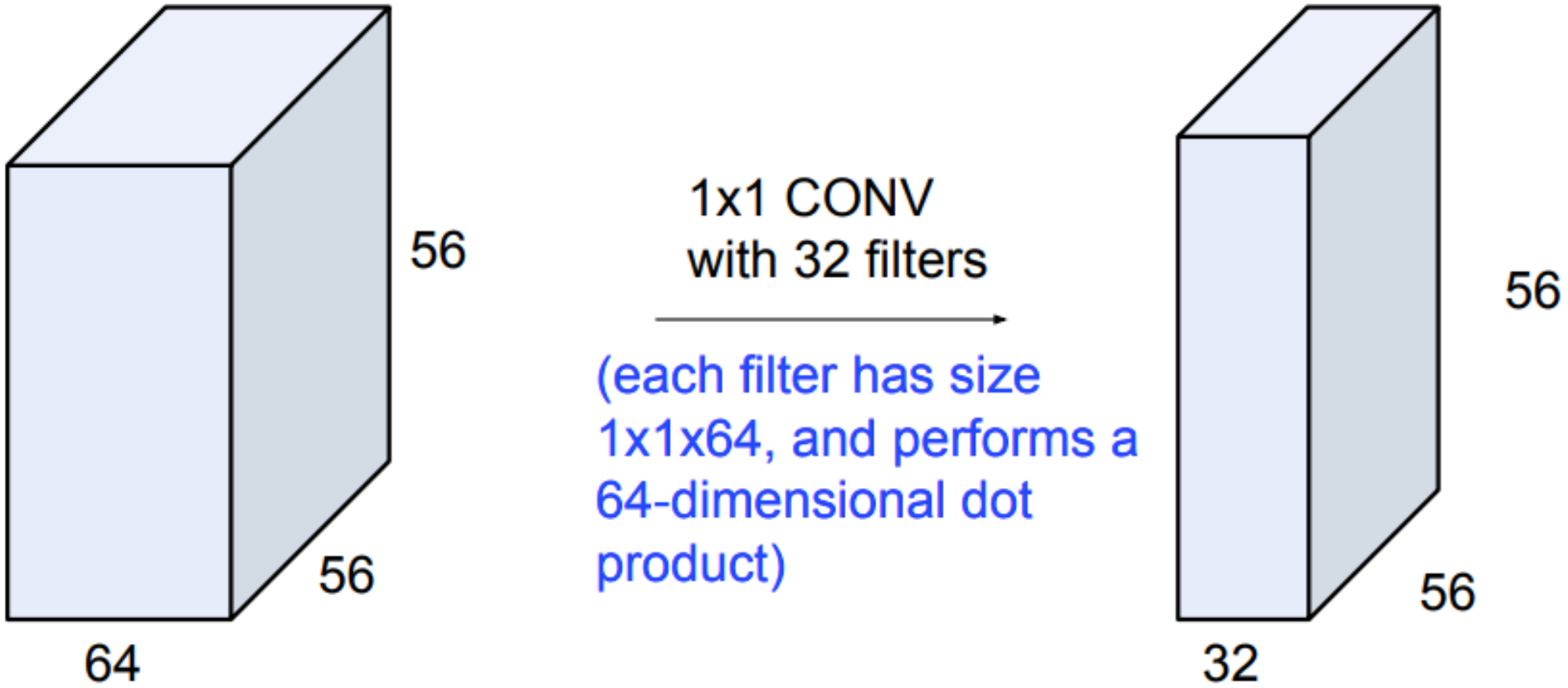
Examples time:



Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

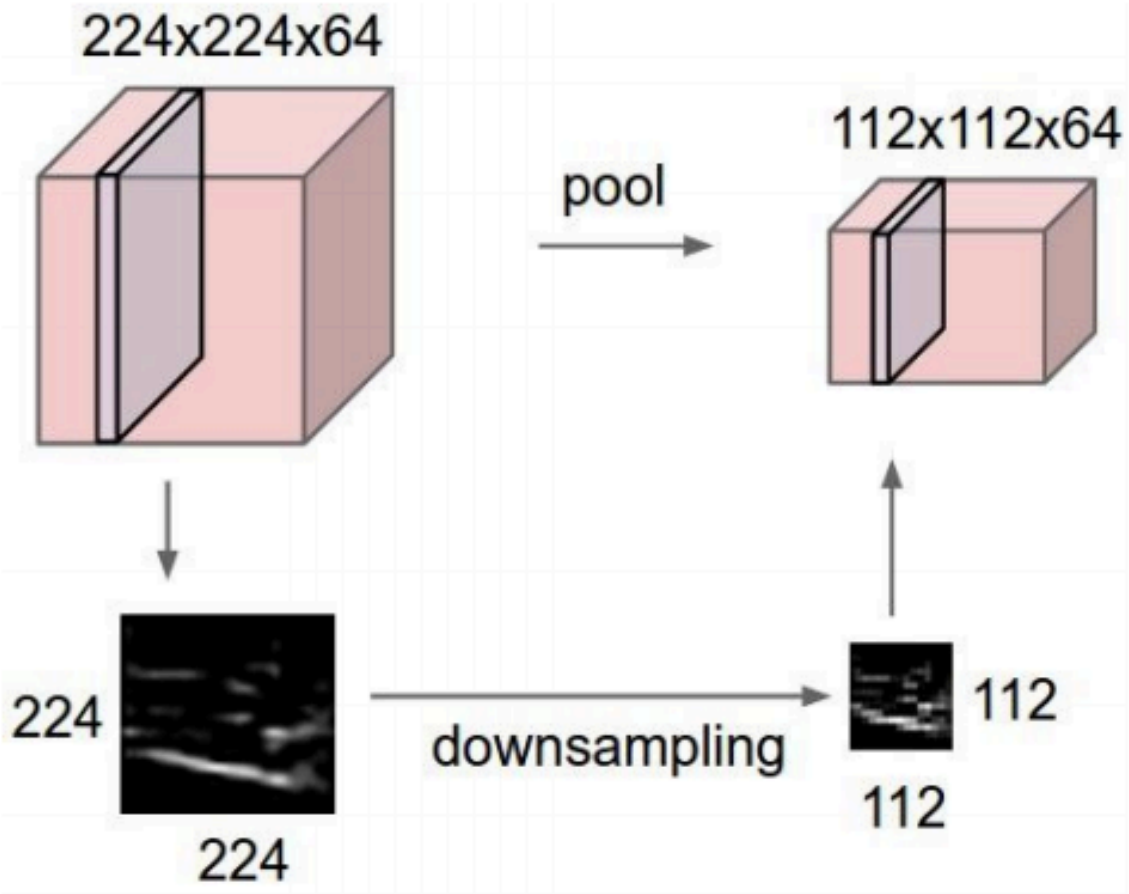
Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

(btw, 1x1 convolution layers make perfect sense)

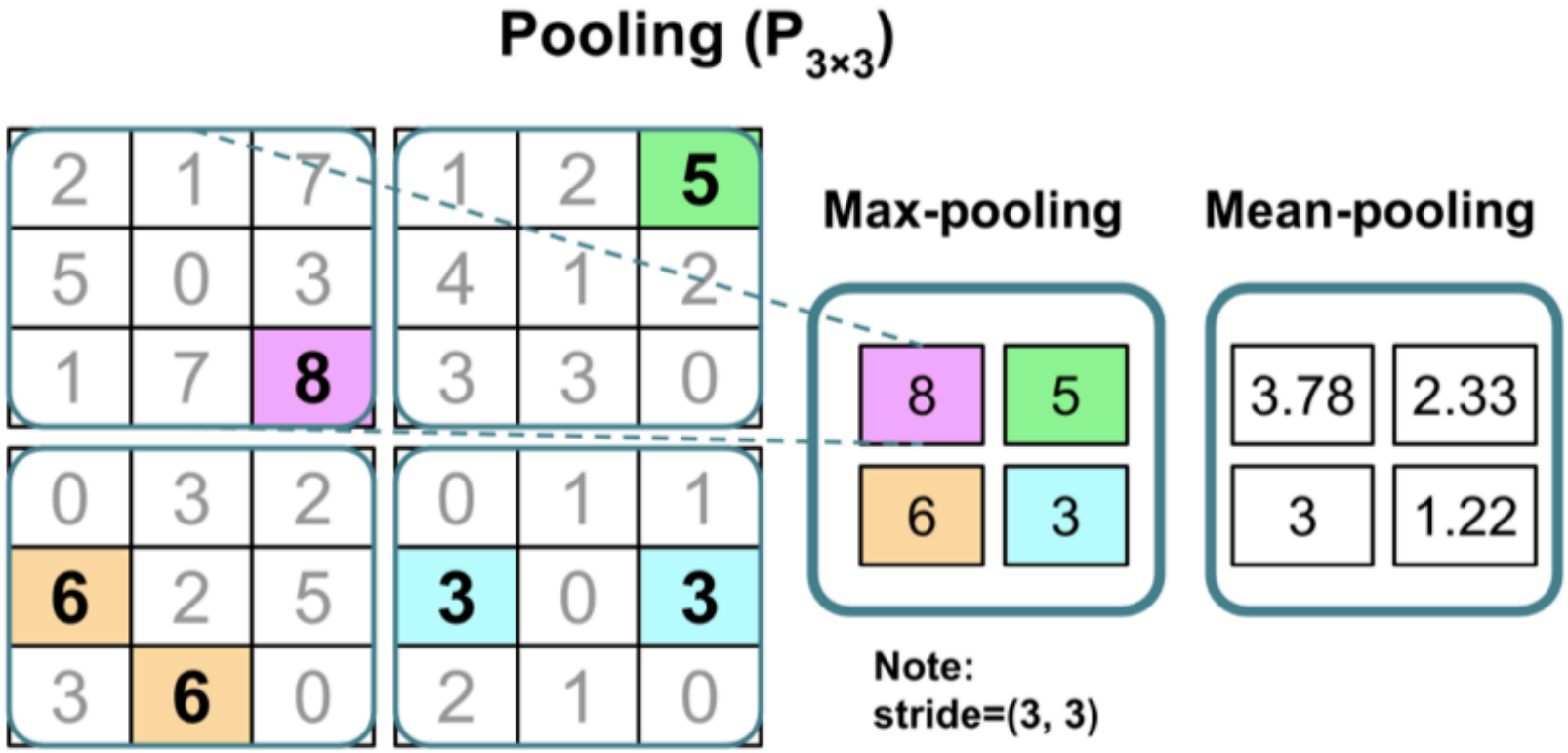


Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



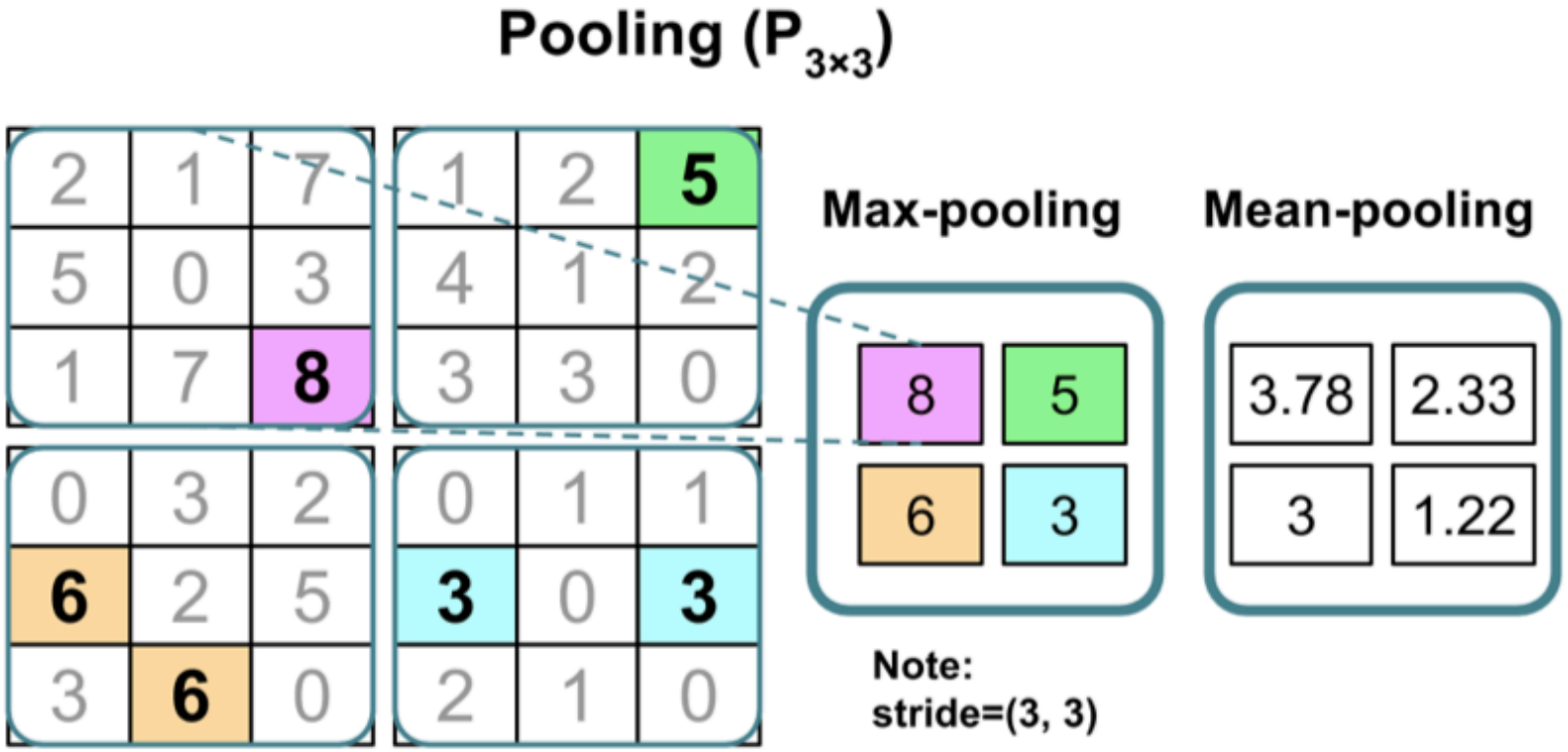
Pooling Layers Can Help With Local Invariance



Downside: Information is lost.

May not matter for classification, but applications where relative position is important (like face recognition)

Pooling Layers Can Help With Local Invariance



Note that typical pooling layers do not have any learnable parameters

Downside: Information is lost.

May not matter for classification, but applications where relative position is important (like face recognition)

Main Breakthrough for CNNs: AlexNet & ImageNet

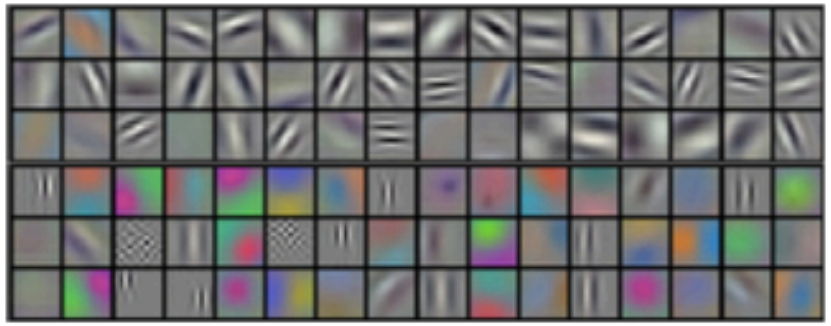
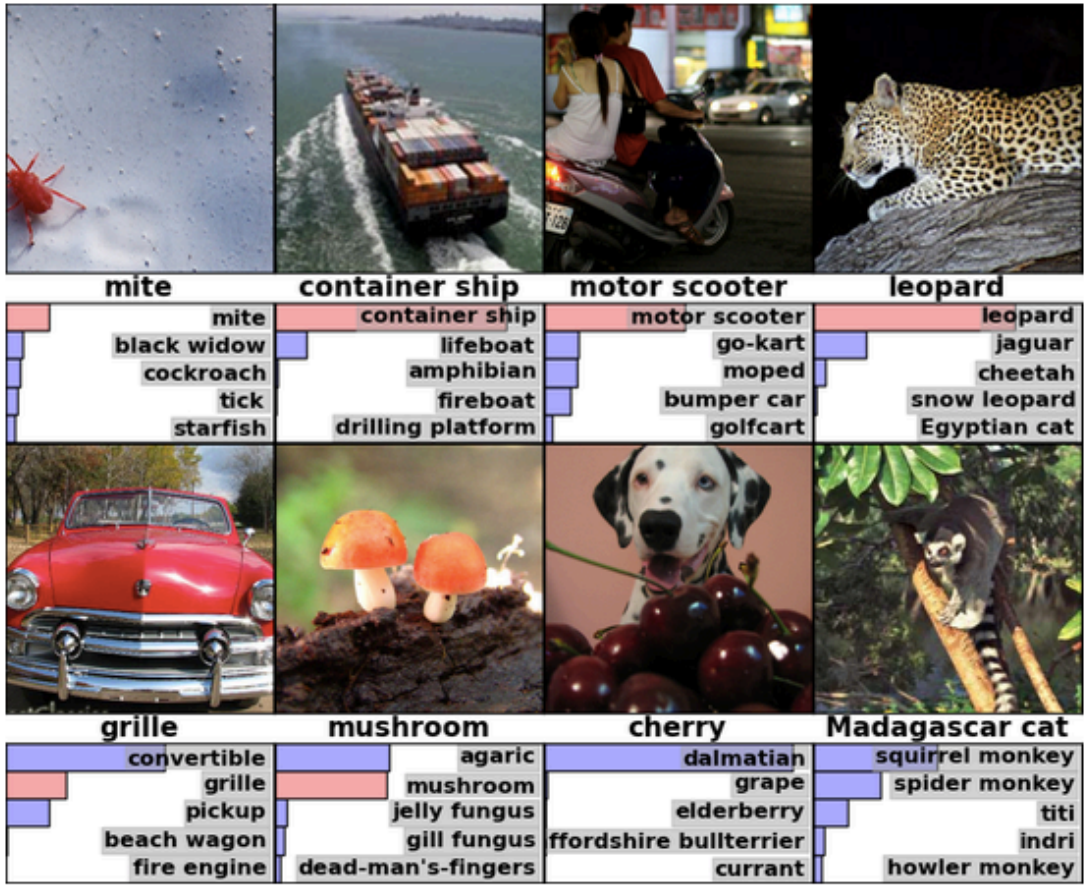


Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

Figure 4: **(Left)** Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). **(Right)** Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Main Breakthrough for CNNs: AlexNet & ImageNet



The ImageNet set that was used has ~1.2 million images and 1000 classes

Accuracy is measured as top-5 performance:
Correct prediction if the true label matches one of the top 5 predictions of the model

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Convolutions with Color Channels

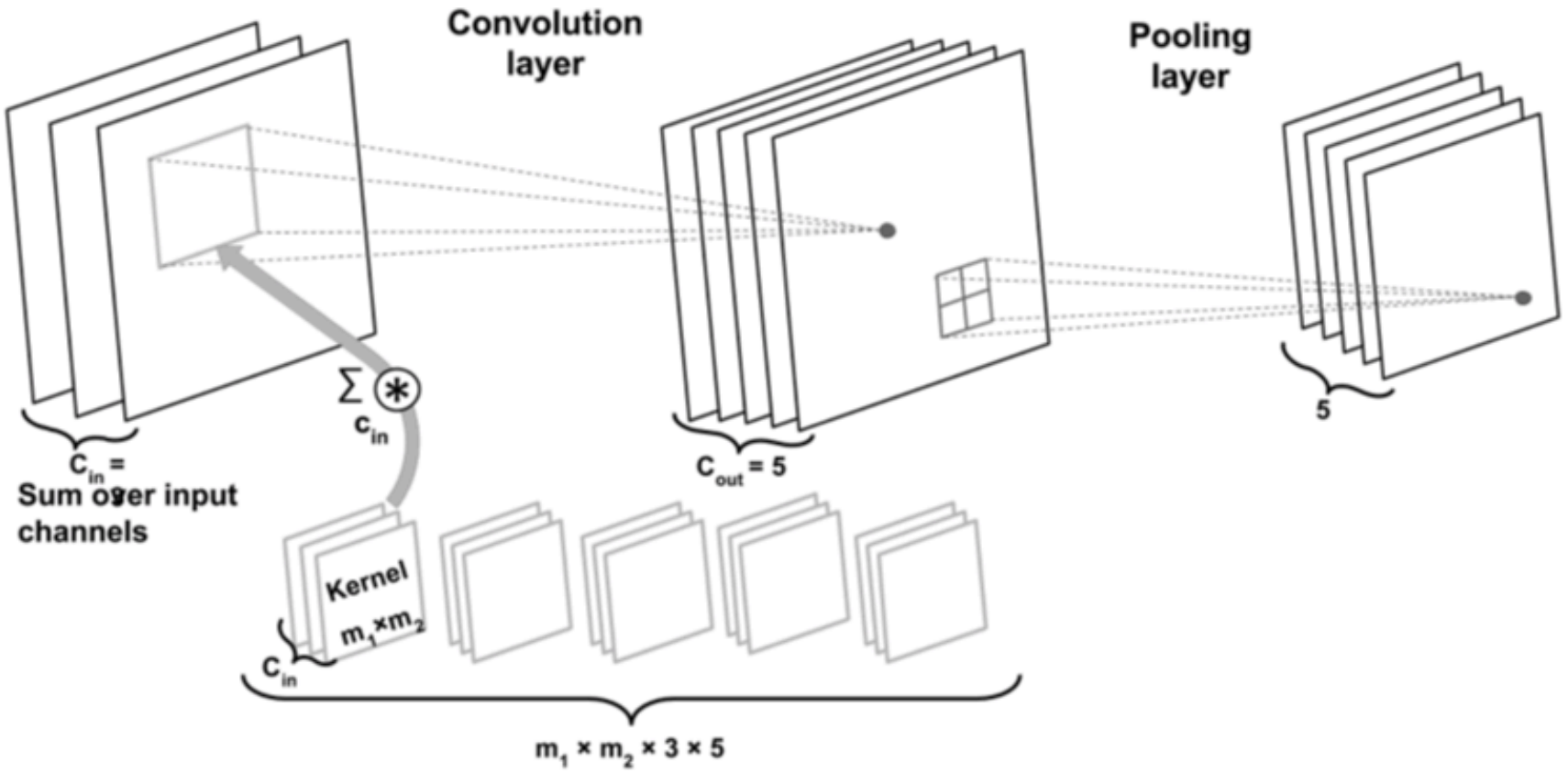


Image dimension: $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times c_{in}}$ in NWHC format,
CUDA & PyTorch use NCWH

Convolutions with Color Channels

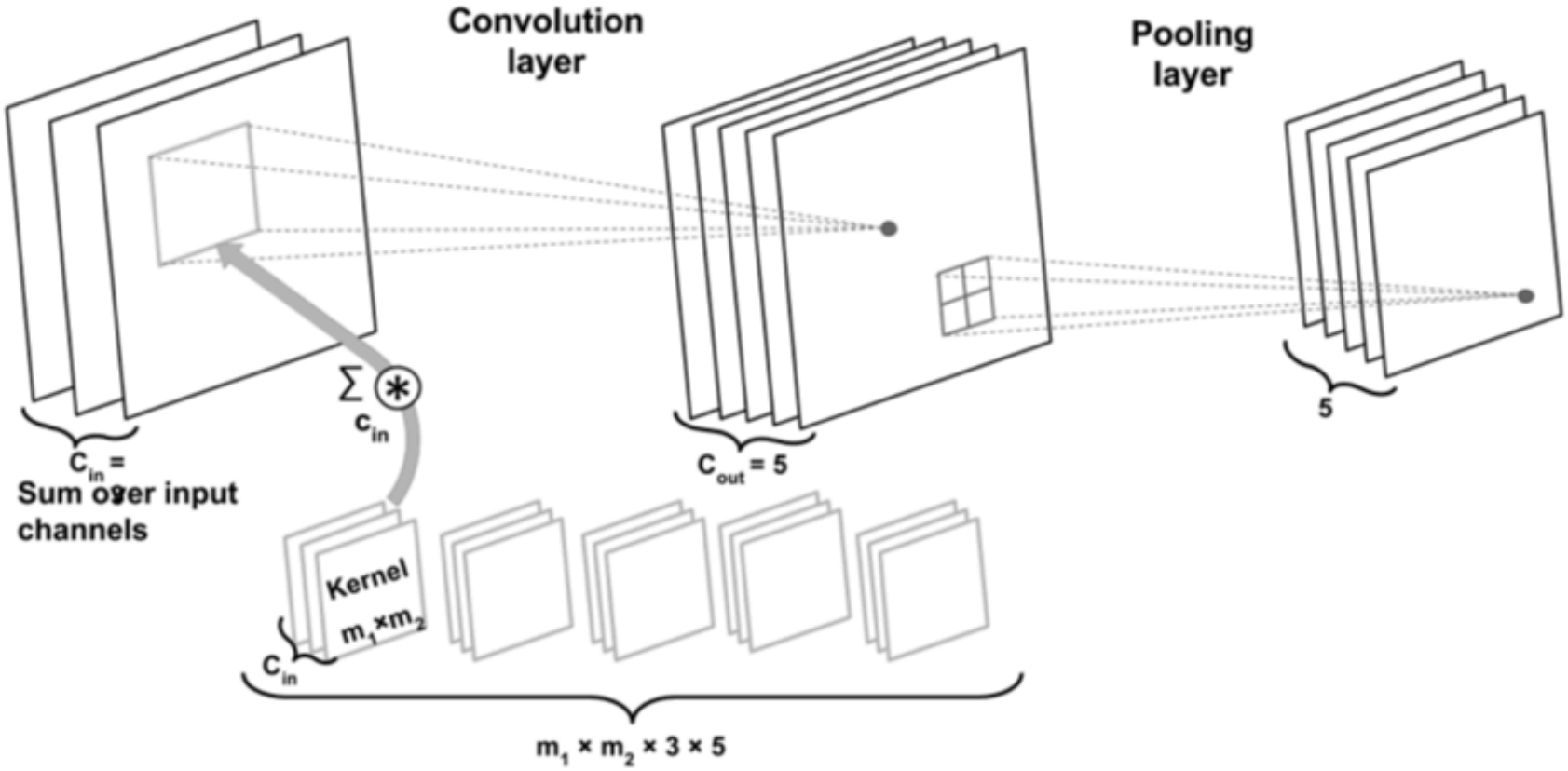
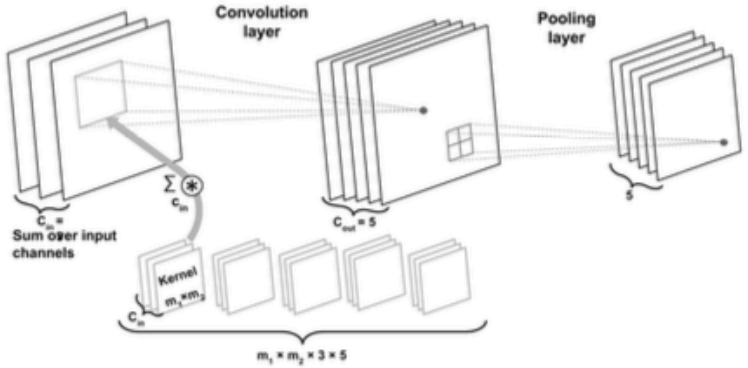


Image dimensions: $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2 \times c_{in}}$

Kernel dimensions: $\mathbf{W} \in \mathbb{R}^{m_1 \times m_2 \times c_{in} \times c_{out}}$ $\mathbf{b} \in \mathbb{R}^{c_{out}}$

Convolutions with Color Channels



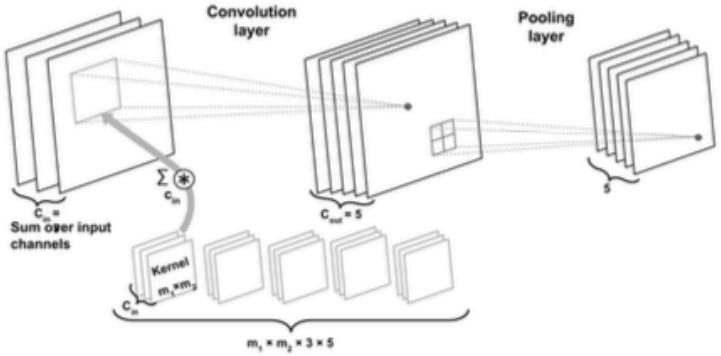
Number of parameters:

$$m_1 \times m_2 \times 3 \times 5 + 5$$

Assume 5x5 kernel:

$$5 \times 5 \times 3 \times 5 + 5 = 380$$

Convolutions with Color Channels



Assume "same" padding (more on padding later), such that the hidden layer has the same height and width as the original images

If we use a CNN:

Number of parameters:

$$m_1 \times m_2 \times 3 \times 5 + 5$$

Assume 128x128 images and 5x5 kernel:

$$5 \times 5 \times 3 \times 5 + 5 = 380$$

If we use a fully connected layer:

Number of parameters:

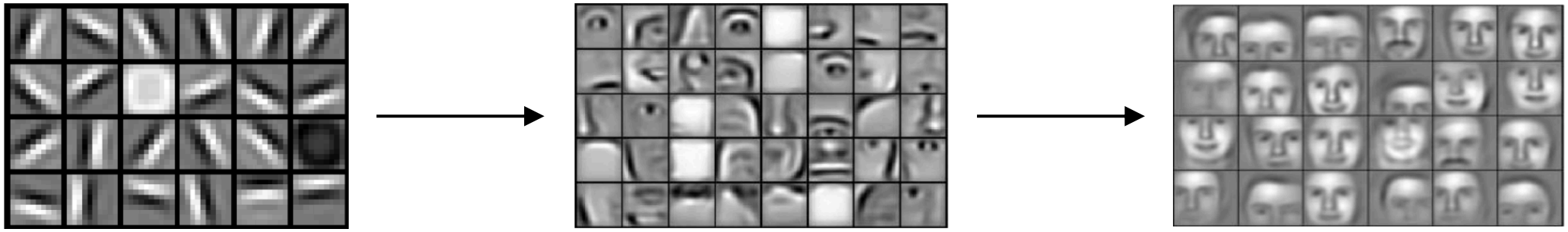
$$(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) + (n_1 \times n_2 \times 5)$$

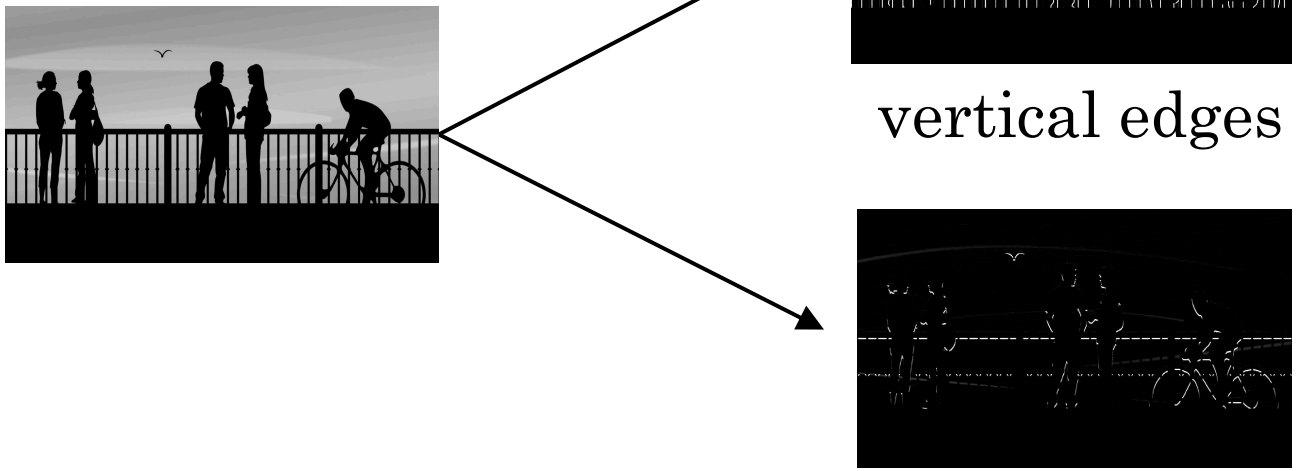
$$(n_1 \times n_2)^2 \times 3 \times 5 + n_1 \times n_2 \times 5$$

$$(128 \times 128)^2 \times 3 \times 5 + 128 \times 128 \times 5$$

$$= 4,026,613,760$$

Traditional Approaches





Traditional Approaches

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

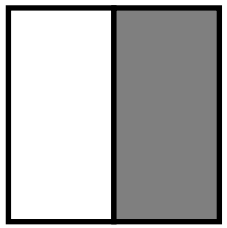
- Vertical edge detection

*

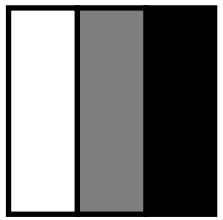
1	0	-1
1	0	-1
1	0	-1

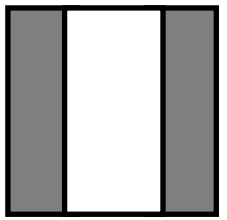
=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0



*





Traditional Approaches

Simple example: vertical edge detector

```
conv = torch.nn.Conv2d(in_channels=1,
                       out_channels=1,
                       kernel_size=(3, 3))
```

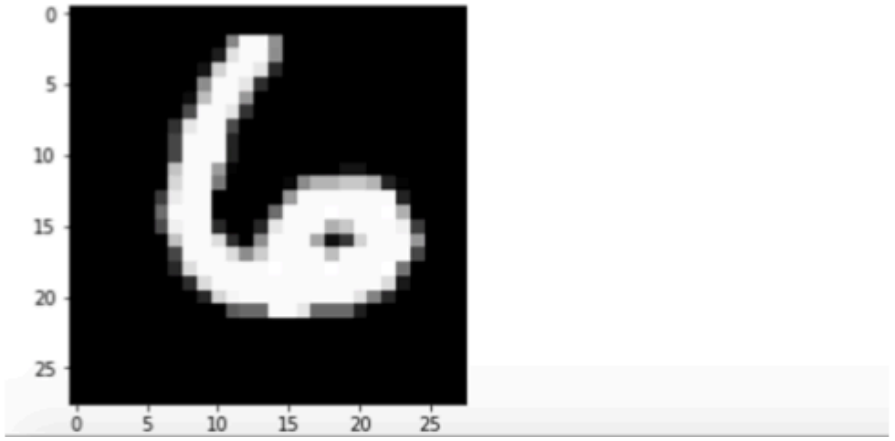
```
conv.weight.size()
```

```
torch.Size([1, 1, 3, 3])
```

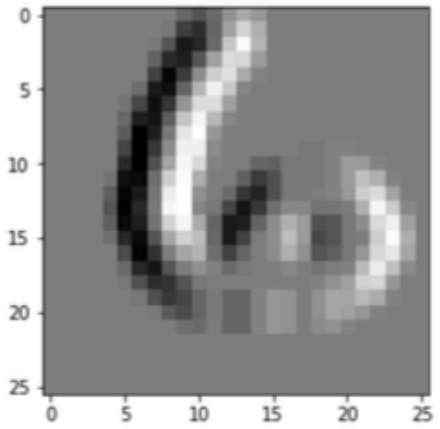
```
conv.weight[0, 0, :, :] = torch.tensor([[1, 0, -1],
                                        [1, 0, -1],
                                        [1, 0, -1]]).float()
conv.bias[0] = torch.tensor([0.]).float()
```

```
images_after = conv(images)
```

```
plt.imshow(images[5, 0], cmap='gray');
```




```
plt.imshow(images_after[5, 0].detach().numpy() , cmap='gray');
```



A Simple CNN in PyTorch

```
class ConvNet(torch.nn.Module):

    def __init__(self, num_classes):
        super(ConvNet, self).__init__()

        # calculate same padding:
        # (w - k + 2*p)/s + 1 = o
        # => p = (s(o-1) - w + k)/2

        # 28x28x1 => 28x28x4
        self.conv_1 = torch.nn.Conv2d(in_channels=1,
                                      out_channels=4,
                                      kernel_size=(3, 3),
                                      stride=(1, 1),
                                      padding=1) # (1(28-1) - 28 + 3) / 2 = 1
        # 28x28x4 => 14x14x4
        self.pool_1 = torch.nn.MaxPool2d(kernel_size=(2, 2),
                                         stride=(2, 2),
                                         padding=0) # (2(14-1) - 28 + 2) = 0
        # 14x14x4 => 14x14x8
        self.conv_2 = torch.nn.Conv2d(in_channels=4,
                                      out_channels=8,
                                      kernel_size=(3, 3),
                                      stride=(1, 1),
                                      padding=1) # (1(14-1) - 14 + 3) / 2 = 1
        # 14x14x8 => 7x7x8
        self.pool_2 = torch.nn.MaxPool2d(kernel_size=(2, 2),
                                         stride=(2, 2),
                                         padding=0) # (2(7-1) - 14 + 2) = 0

        self.linear_1 = torch.nn.Linear(7*7*8, num_classes)
```

(forward method on the next slide)

A Simple CNN in PyTorch

```
    def forward(self, x):
        out = self.conv_1(x)
        out = F.relu(out)
        out = self.pool_1(out)

        out = self.conv_2(out)
        out = F.relu(out)
        out = self.pool_2(out)

        logits = self.linear_1(out.view(-1, 7*7*8))
        probas = F.softmax(logits, dim=1)
        return logits, probas


torch.manual_seed(random_seed)
model = ConvNet(num_classes=num_classes)
```

(model parameters on the previous slide)

Spatial Dropout – Droupout2D

- Problem with regular dropout and CNNs:
  Adjacent pixels are likely highly correlated
  (thus, may not help with reducing the
  "dependency" much as originally intended by
  dropout)

- Hence, it may be better to drop entire feature maps

Idea comes from

Tompson, Jonathan, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler.
"Efficient object localization using convolutional networks." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 648-656. 2015.

Spatial Dropout – Droupout2D

Dropout2d will drop full feature maps (channels)

```
import torch

m = torch.nn.Dropout2d(p=0.5)
input = torch.randn(1, 3, 5, 5)
output = m(input)
```

```
output
```

```
tensor([[[[-0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000, -0.0000,  0.0000, -0.0000,  0.0000],
          [ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.0000, -0.0000, -0.0000]],

         [[-3.5274,  0.8163,  0.2440,  1.2410,  1.5022],
          [-1.2455,  6.3875, -2.6224,  0.0261,  1.7487],
          [ 1.6471,  0.7444, -2.1941, -2.0119, -1.5232],
          [ 0.3720, -1.5606,  0.7630,  0.9177, -0.1387],
          [-1.2817, -3.5804,  0.4367, -0.1384, -0.8148]],

         [[-0.0000, -0.0000, -0.0000, -0.0000,  0.0000],
          [ 0.0000, -0.0000, -0.0000, -0.0000,  0.0000],
          [ 0.0000, -0.0000,  0.0000, -0.0000, -0.0000],
          [-0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [-0.0000,  0.0000,  0.0000,  0.0000,  0.0000]]]])
```

Computing Convolutions on the GPU

- There are many different approaches to compute (approximate) convolution operations
- DL libraries usually use NVIDIA's CUDA & CuDNN libraries, which implement many different convolution algorithms
- These algorithms are usually more efficient than the CPU variants (convolutions on the CPU e.g., in CPU usually take up much more memory due to the algorithm choice compared to using the GPU)

If you are interested, you can find more info in:

Lavin, Andrew, and Scott Gray. "Fast algorithms for convolutional neural networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016.
https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Lavin_Fast_Algorithms_for_CVPR_2016_paper.pdf

Computing Convolutions on the GPU

- CuDNN is more geared towards engineers & speed rather than scientists and is unfortunately not deterministic/reproducible by default
- I.e., it determines which convolution algorithm to choose during run-time automatically, based on predicted speeds given the data flow
- For reproducibility and consistent results, I recommend setting the deterministic flag (speed is about the same, often even a bit faster, sometimes a bit slower)

```
import torch
import torch.nn as nn
import torch.nn.functional as F

if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True
```

Resources Used

- CS231n Convolutional Neural Networks for Visual Recognition by Fei-Fei Li, Justin Johnson, Seran Yeung
- STAT 479: Deep Learning by Sebastian Raschka
- Deeplearnging.ai by Andrew Ng