Convolutional Neural Networks (CNN)

Dr. Omri Allouche

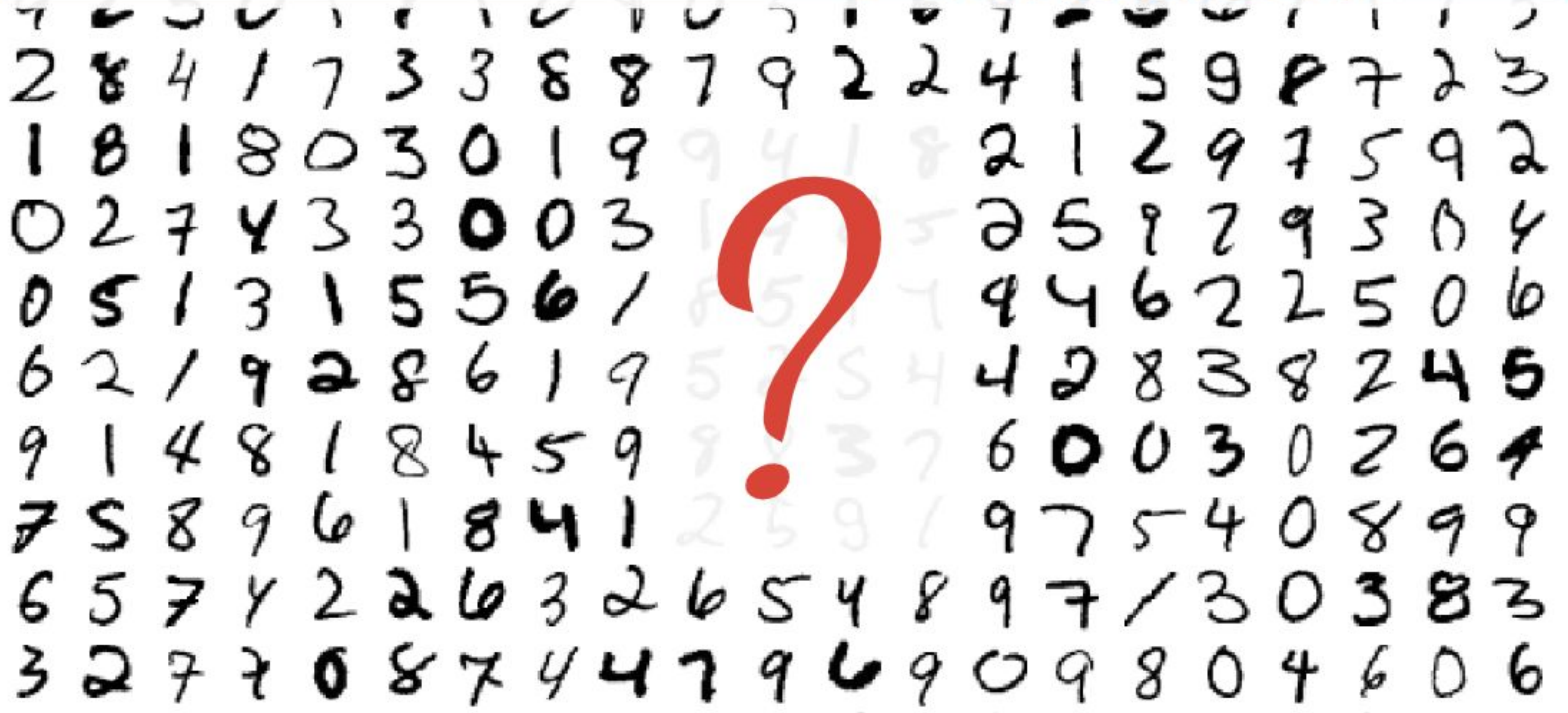
Y-Data Deep Learning Course

Suggested Reading

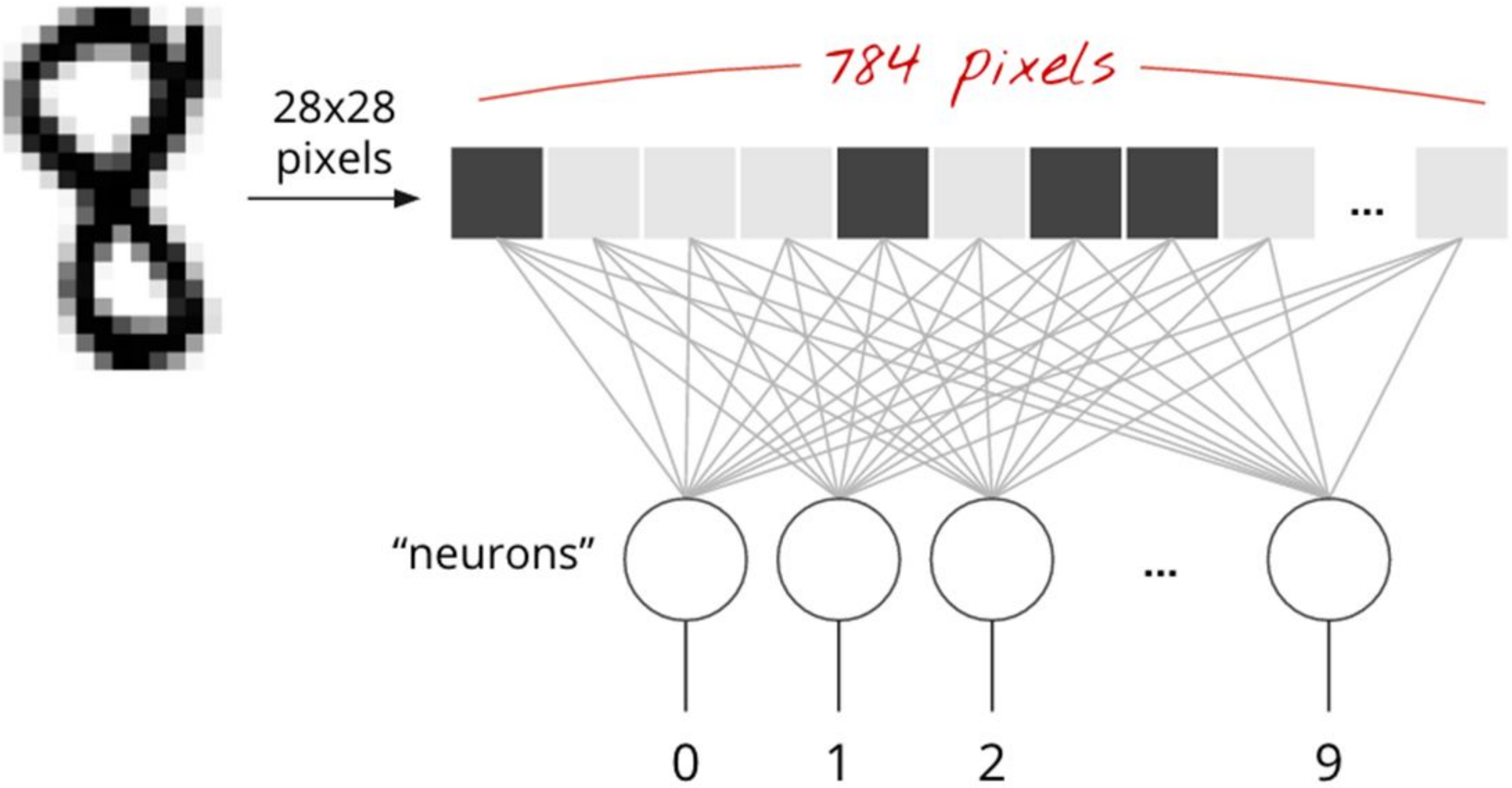
- https://cs231n.github.io/convolutional-networks/
- Chapter 9 of “Deep Learning” – https://www.deeplearningbook.org/contents/convnets.html

Refresher – NN Training

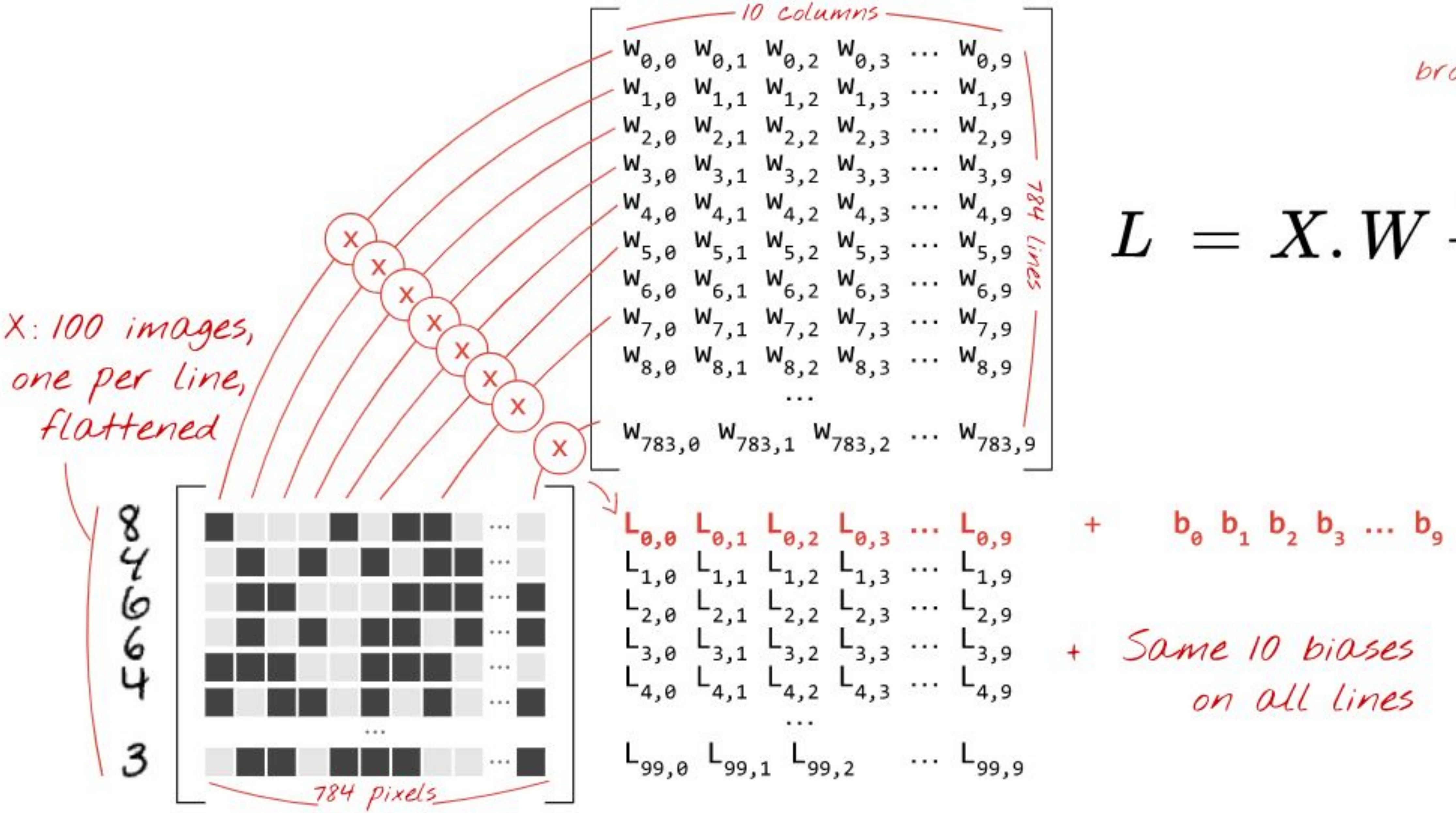
Hello World: handwritten digits classification - MNIST



MNIST = Mixed National Institute of Standards and Technology - Download the dataset at http://yann.lecun.com/exdb/mnist/

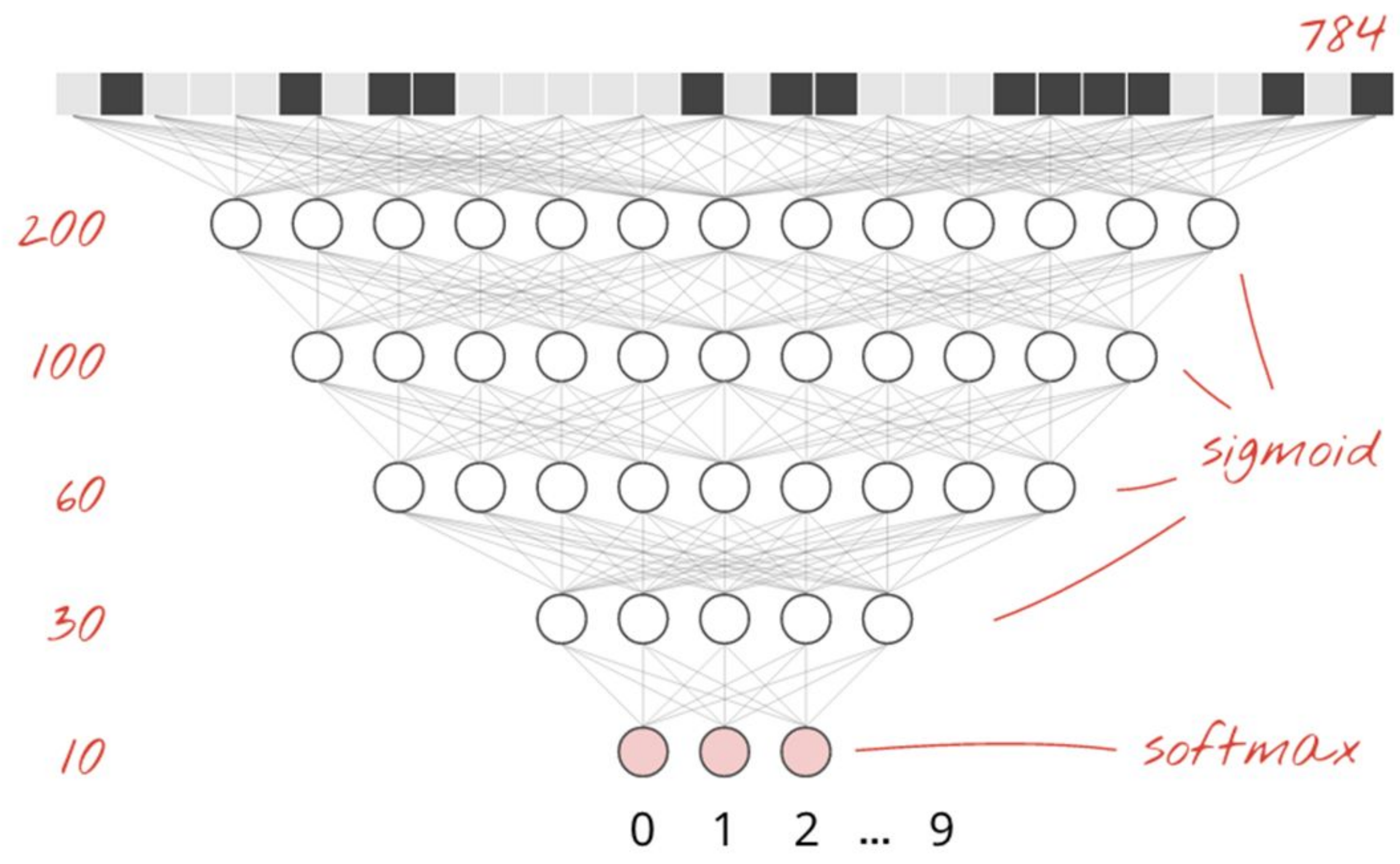

784 pixels
28x28
pixels
...
"neurons"
...
0
1
2
9

In matrix notation, 100 images at a time

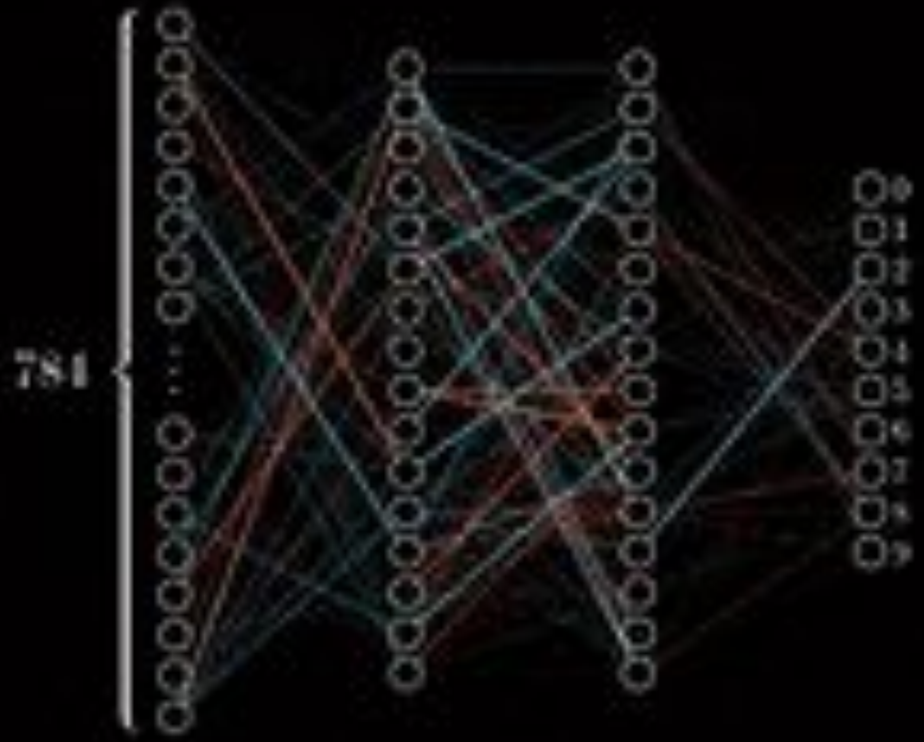


broadcast

$$L = X.W + b$$


784
200
100
60
30
10
sigmoid
softmax
0 1 2 ... 9

Training in progress...



Deep Learning Training Recipe

- Randomly choose the initial weights
- While error is too large
    - For each training pattern (presented in random order)

      Feedforward:

        - Apply the inputs to the network
        - Calculate the output for every neuron from the input layer, through the hidden layer(s), to the output layer

      Loss:

        - Calculate the error at the outputs
        - Use the output error to compute the **loss** – error signals for pre-output layers

      Backpropage:

        - Use the loss to compute weight adjustments
        - Apply the weight adjustments
    - Periodically evaluate the network performance

Neural Networks in PyTorch

- The network architecture is often defined as a class inheriting from nn.Module
  - Inits parameters in the constructor
  - Implements the *forward* method
- We then define the loss function and the optimizer:

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

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


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

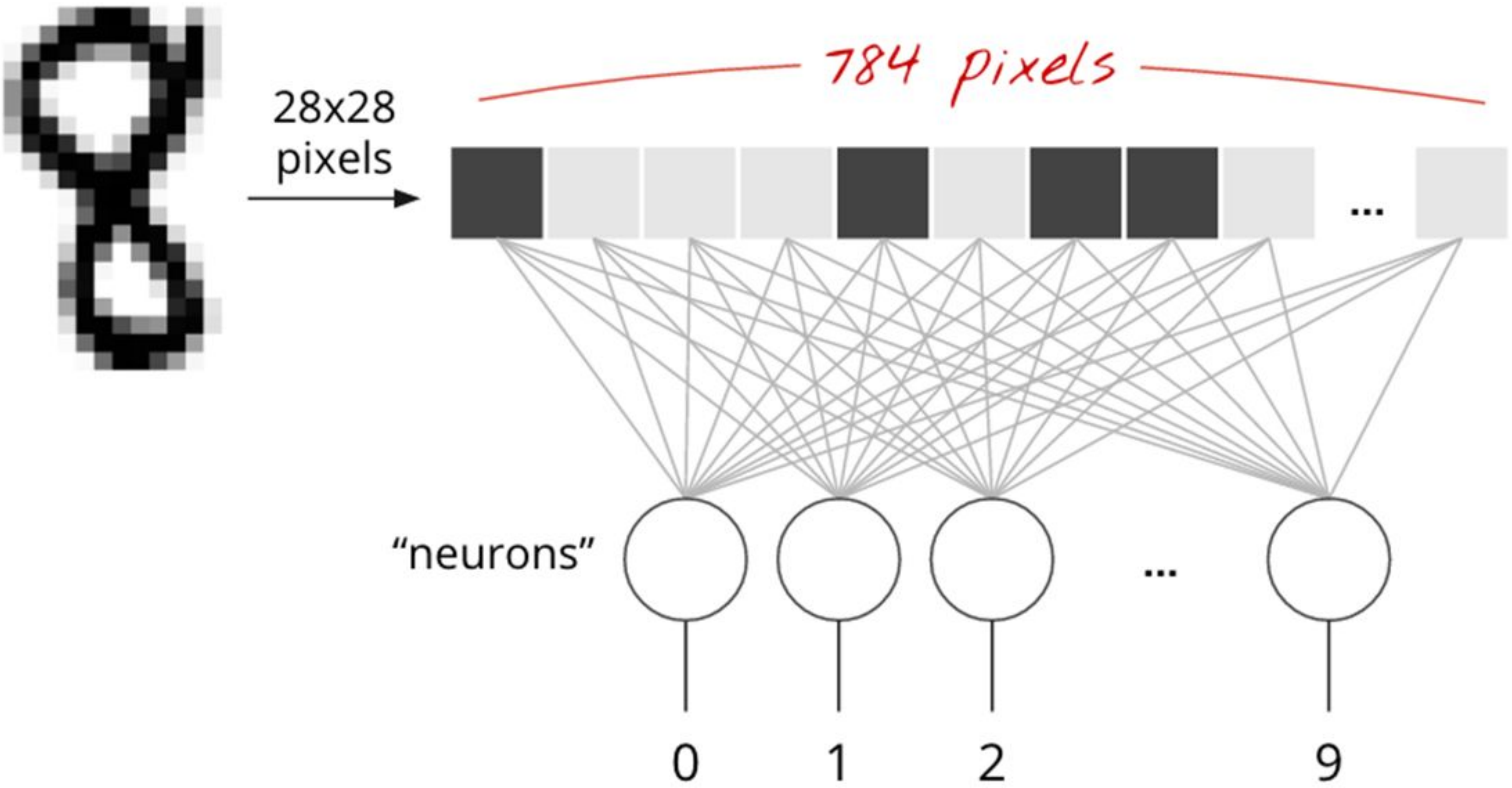
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Training the Network is done in a loop

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

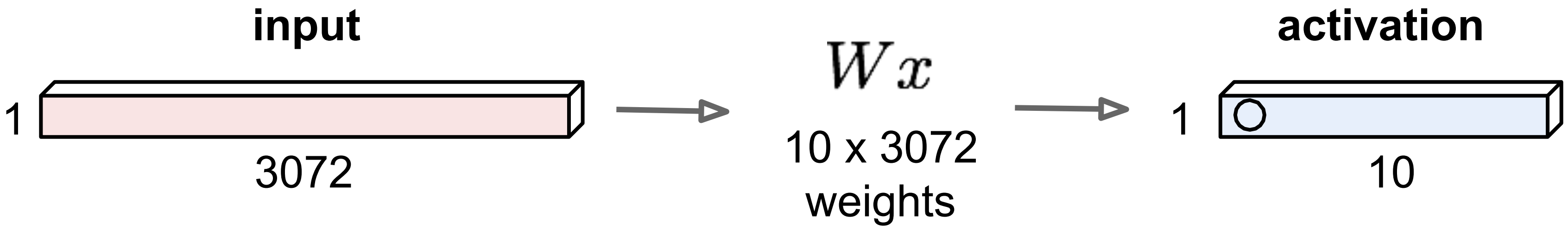
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

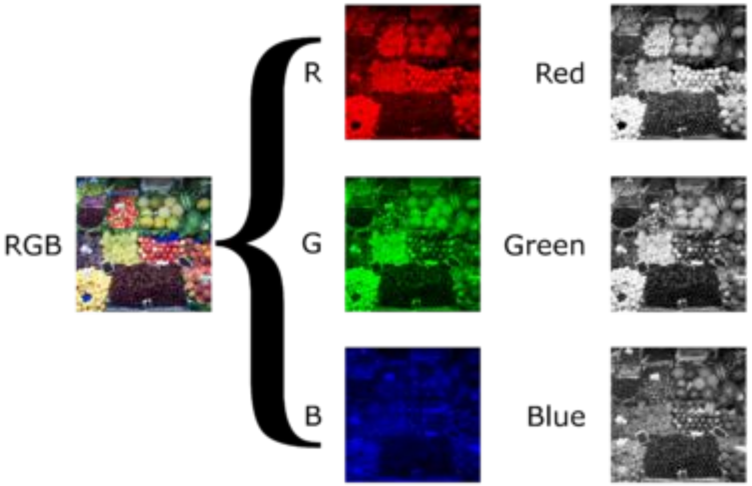
CNNs


784 pixels
28x28
pixels
...
"neurons"
...
0
1
2
9

Fully Connected Layer

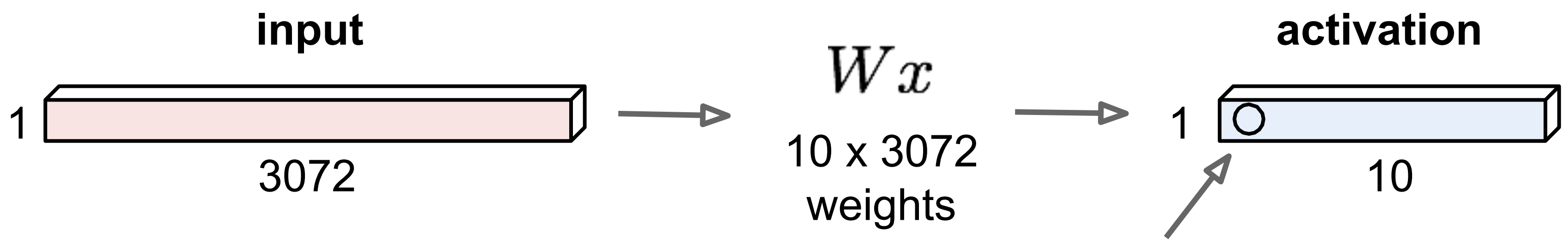
32x32x3 image -> stretch to 3072 x 1





Fully Connected Layer

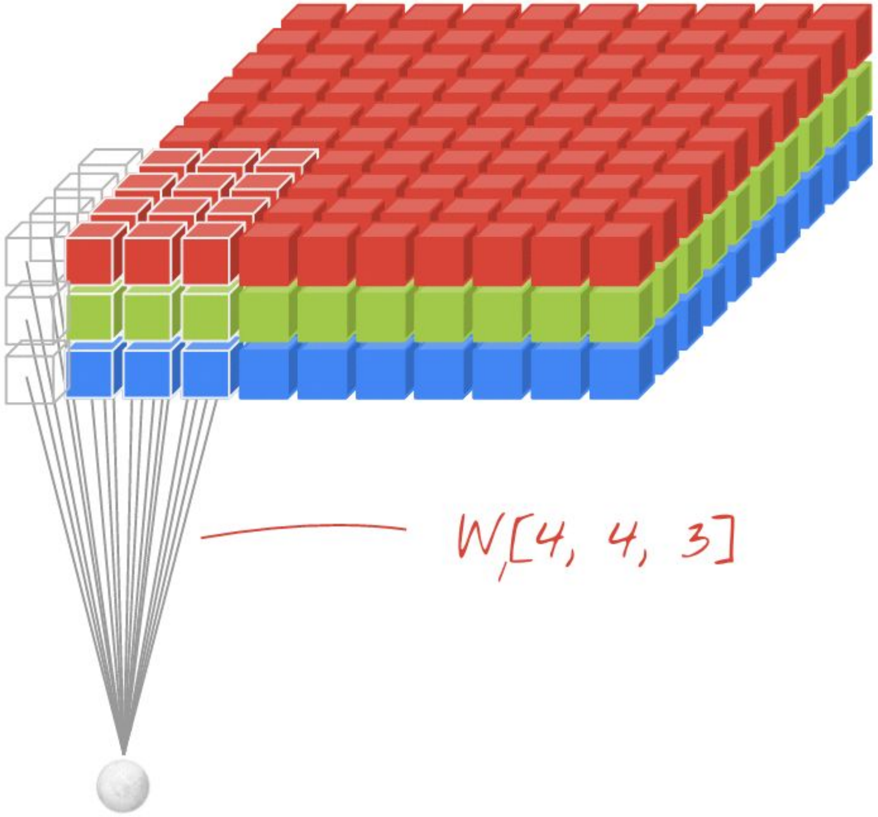
32x32x3 image -> stretch to 3072 x 1



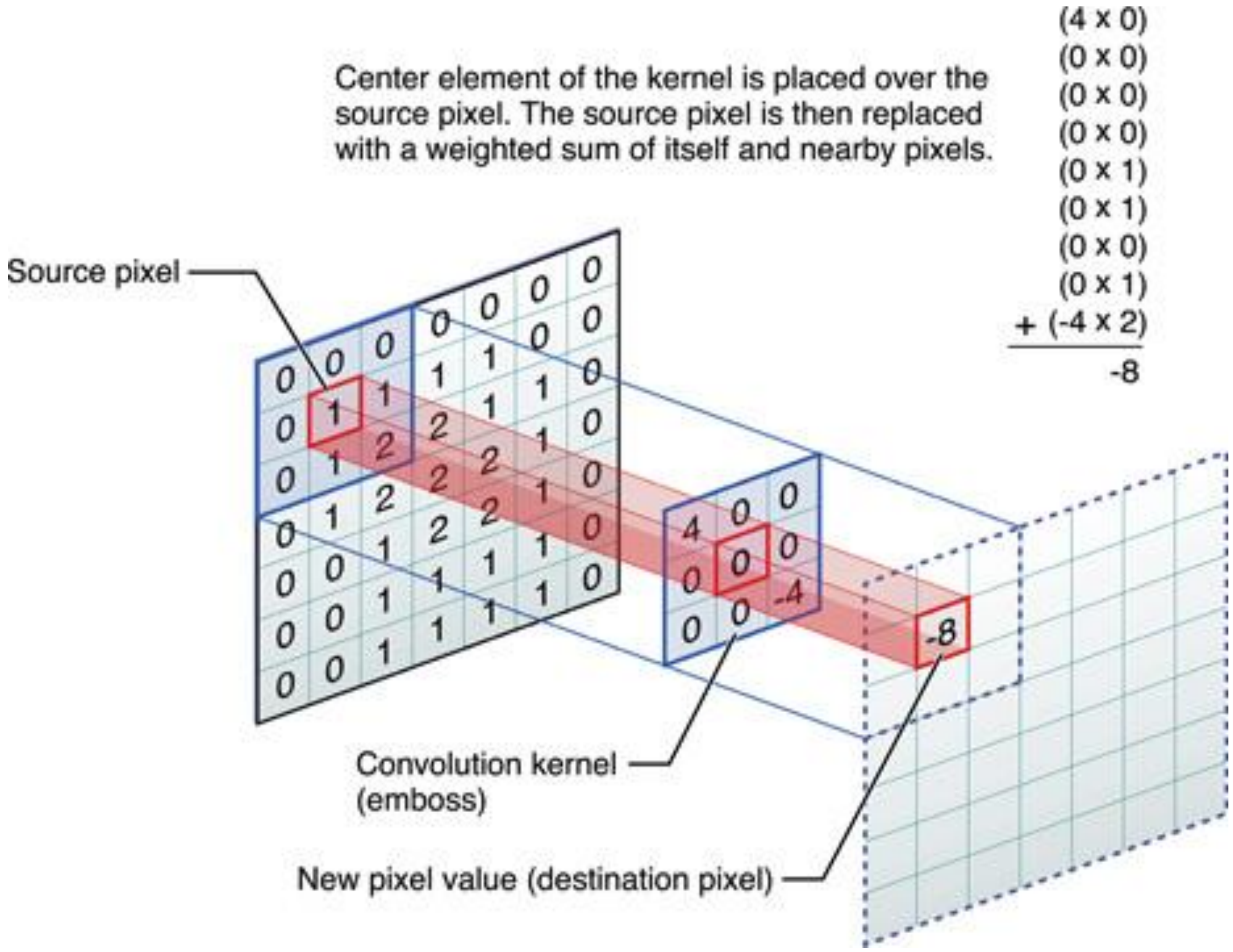
1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

Take local interactions into account using CNNs

Convolutional Neural Networks (CNNs) take a neighborhood of cells and learn the weights of filters



Convolution Layer



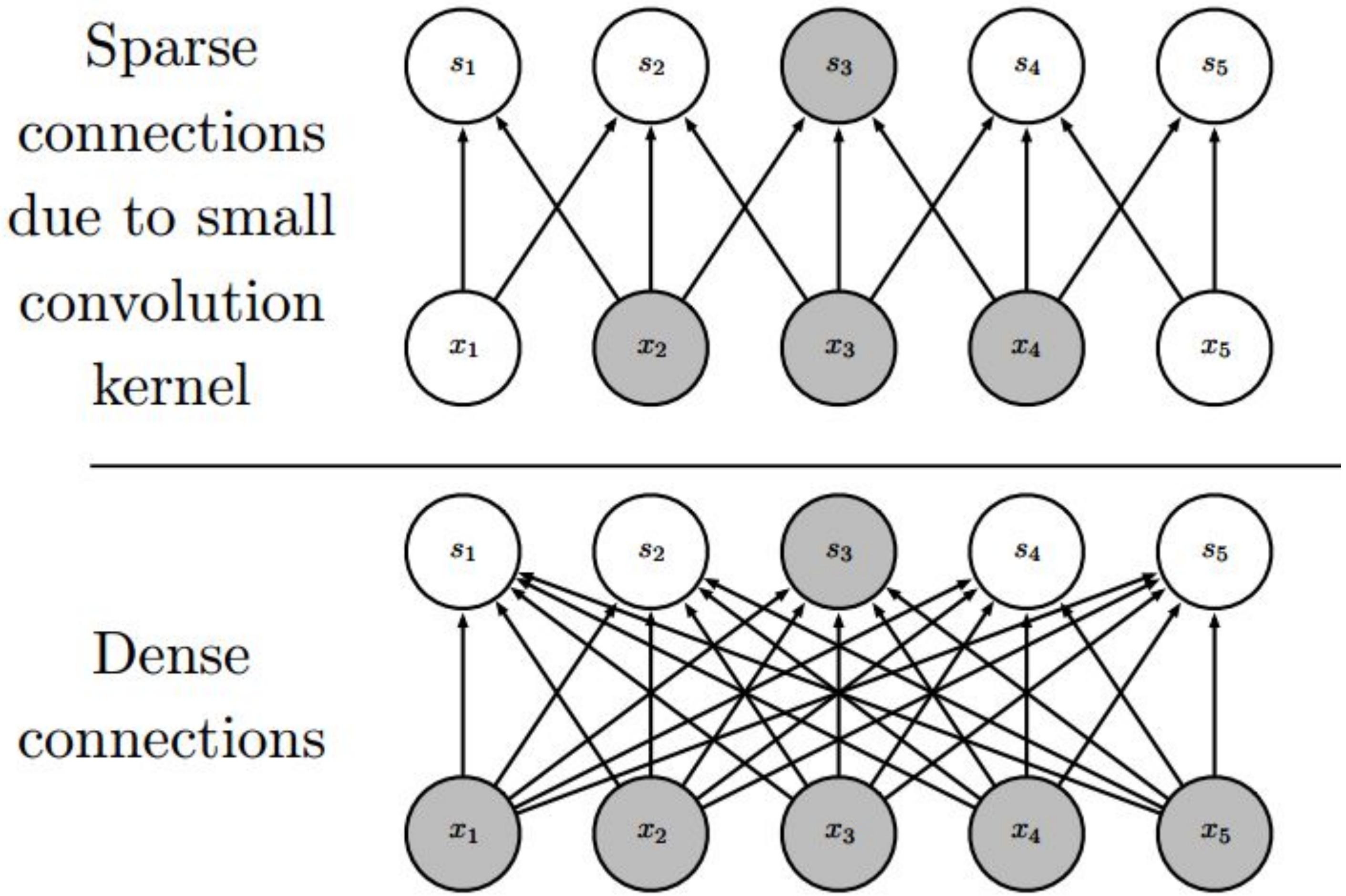
FILTER #1:
AVERAGING EACH PIXEL WITH ITS NEIGHBORING VALUES BLURS AN IMAGE:

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

	0	1	0	
	1	-4	1	
	0	1	0	

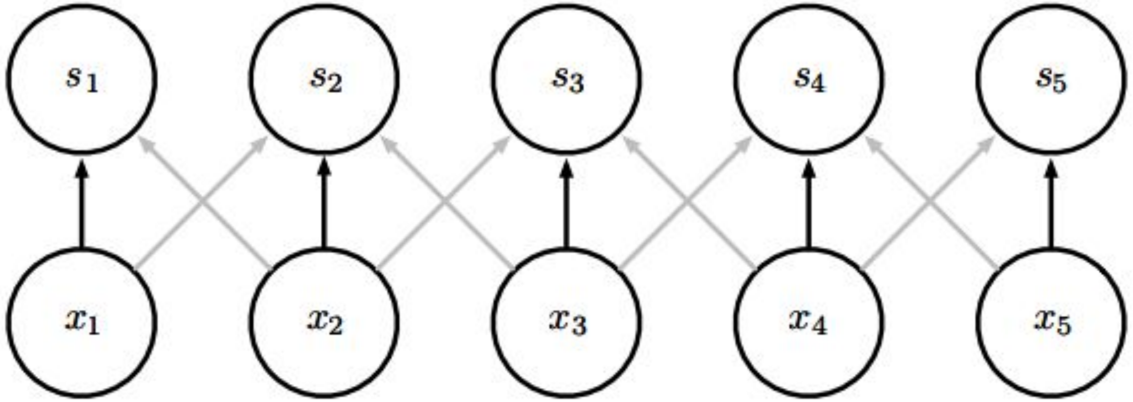
0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0

CNN vs Plain NN: Sparse Connectivity

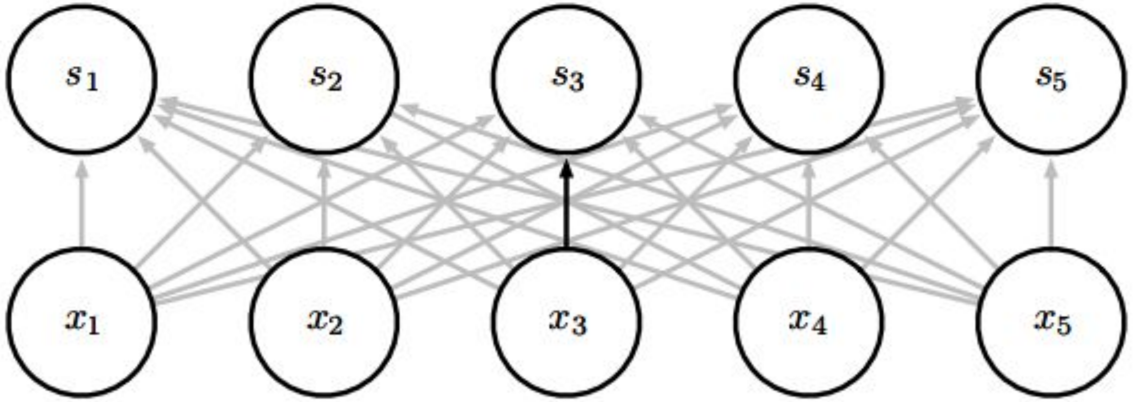


CNN vs Plain NN: Parameter Sharing

Convolution shares the same parameters across all spatial locations

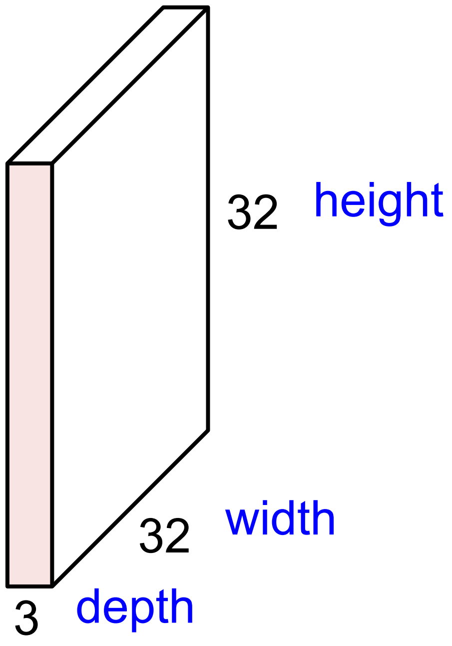


Traditional matrix multiplication does not share any parameters



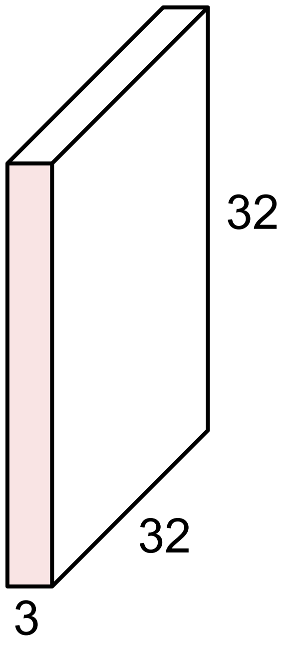
Convolution Layer

32x32x3 image -> preserve spatial structure

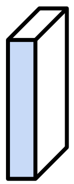


Convolution Layer

32x32x3 image



5x5x3 filter

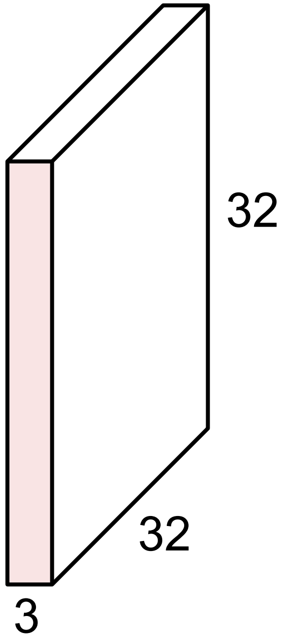


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

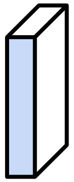
Convolution Layer

Filters almost always extend the full depth of the input volume

32x32x3 image

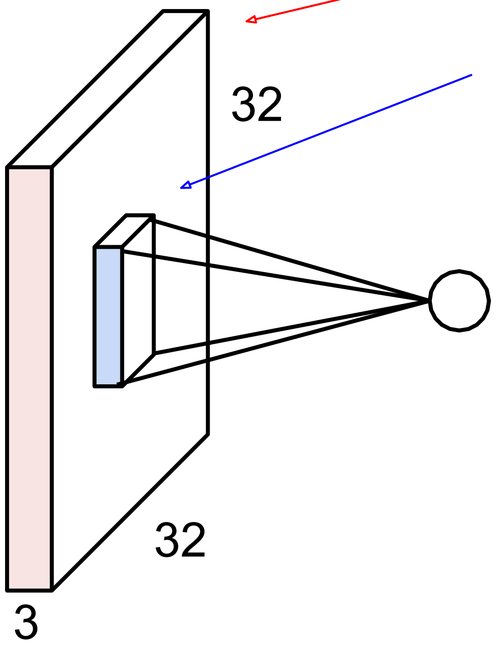


5x5x3 filter



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

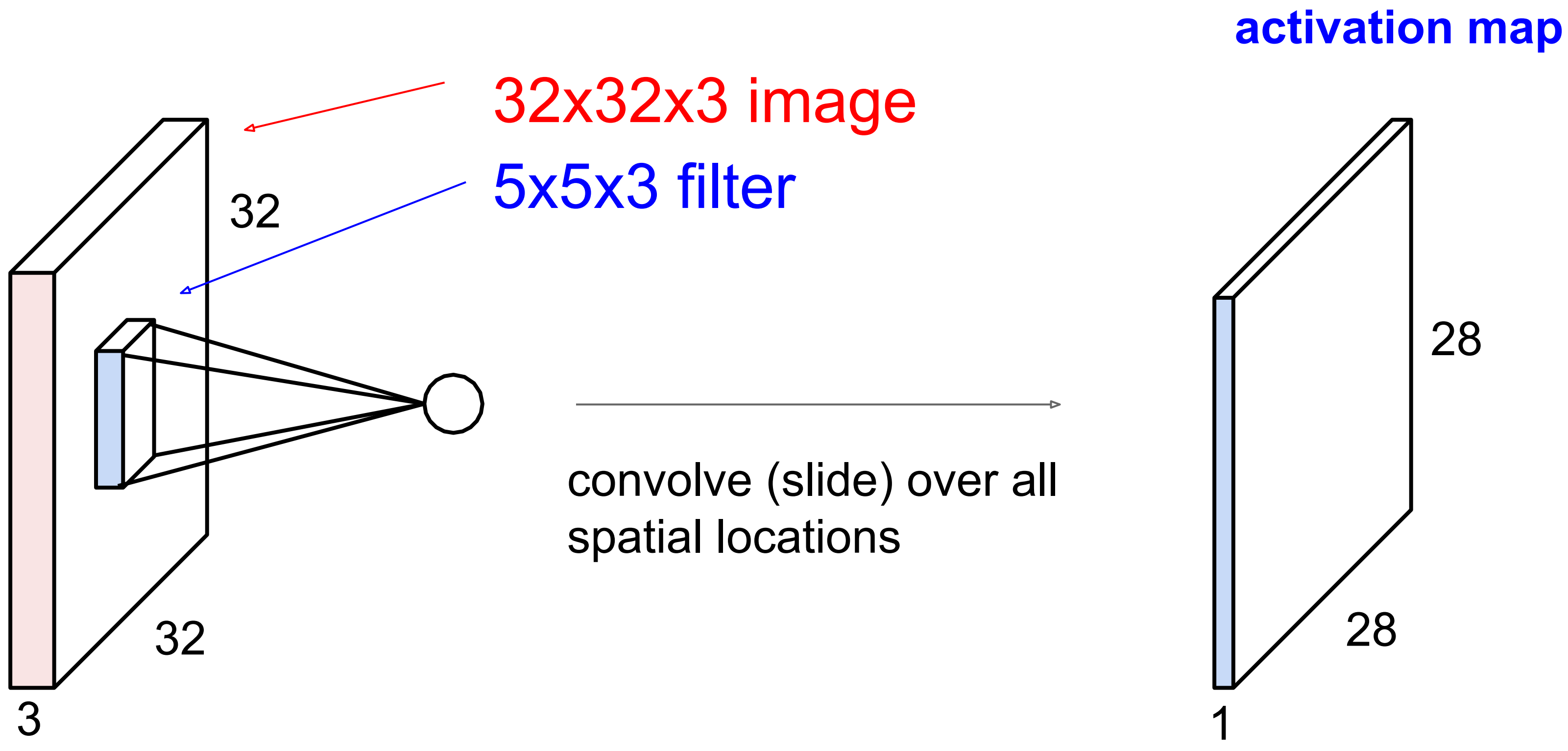
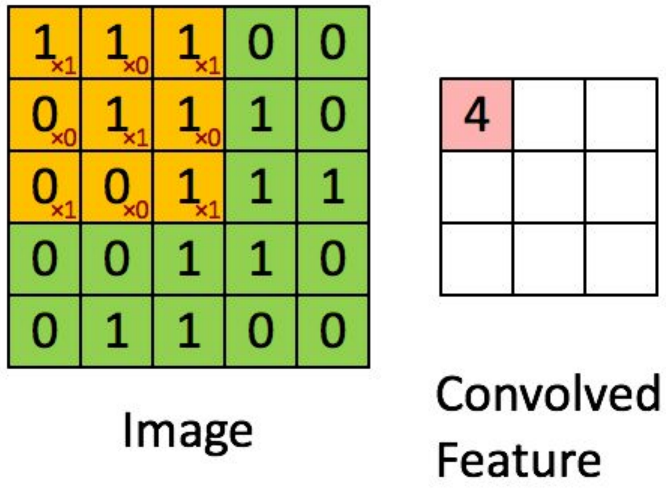


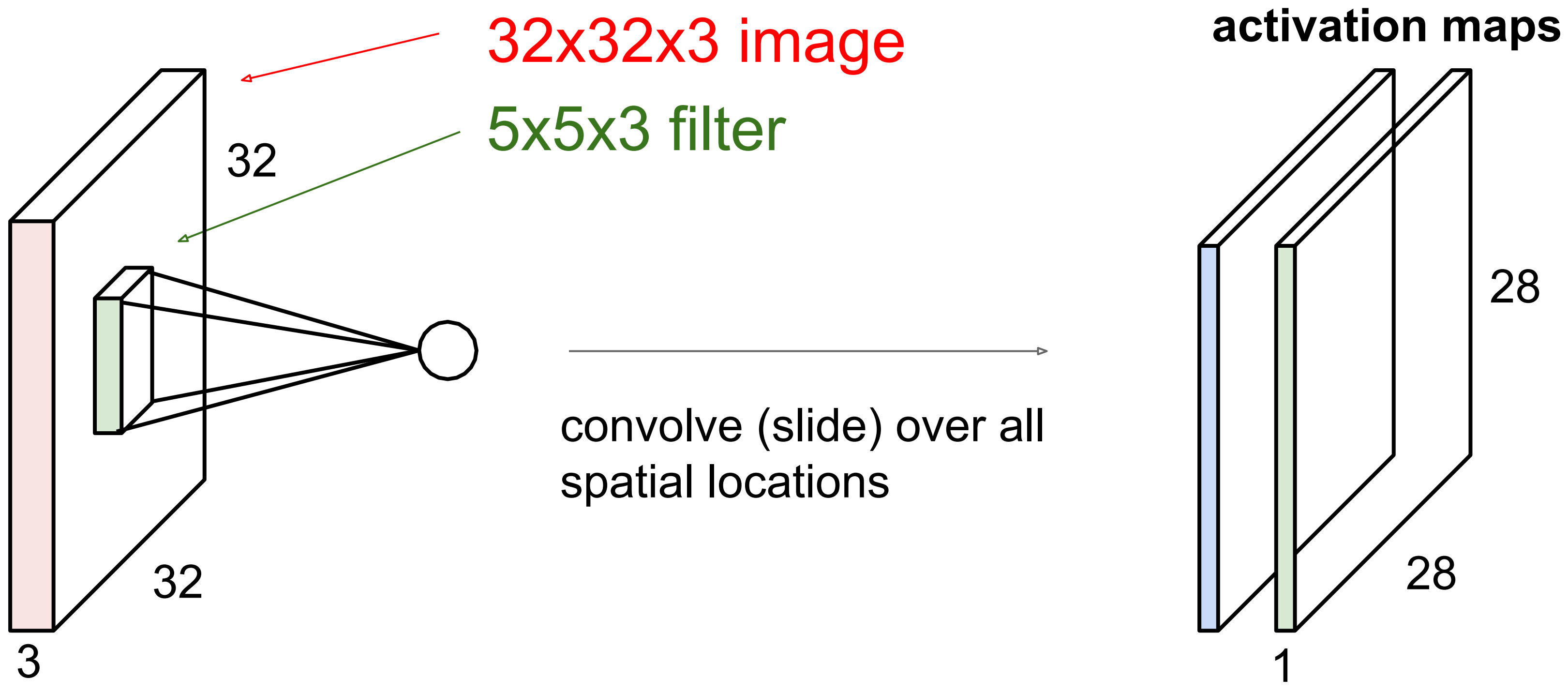
Illustration of the Convolution Stage

- Think of it as a sliding window function applied to a matrix.
- The sliding window is called a *kernel, filter,* or *feature detector.* Here we use a 3×3 filter, multiply its values element-wise with the original matrix, then sum them up

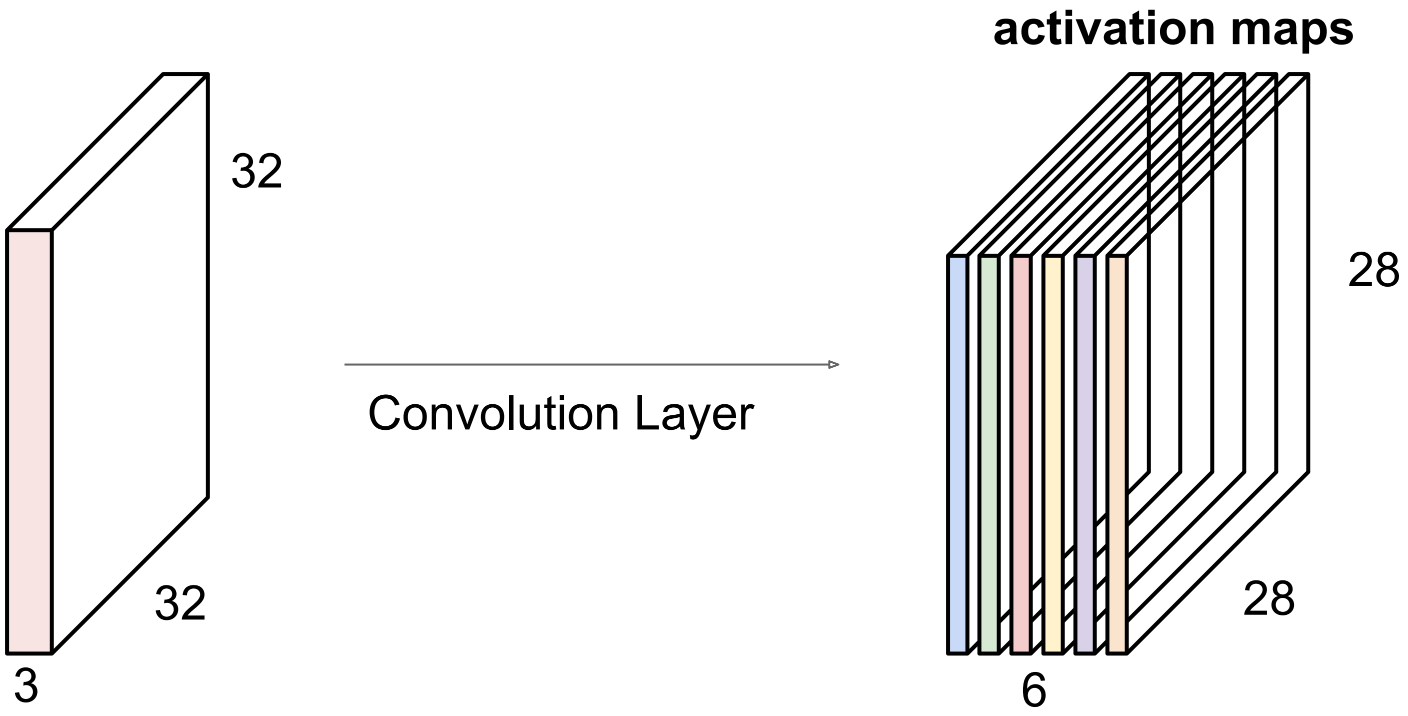


consider a second, green filter

Convolution Layer

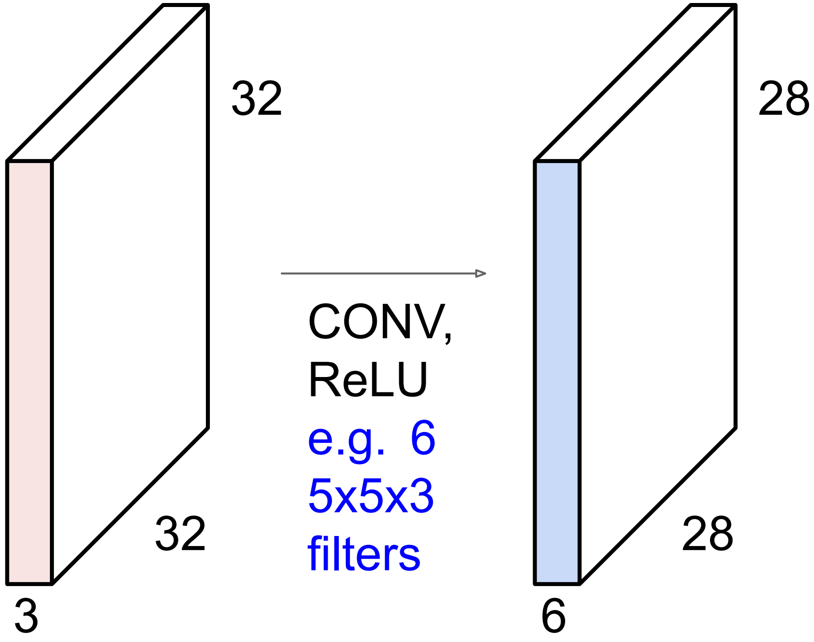


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

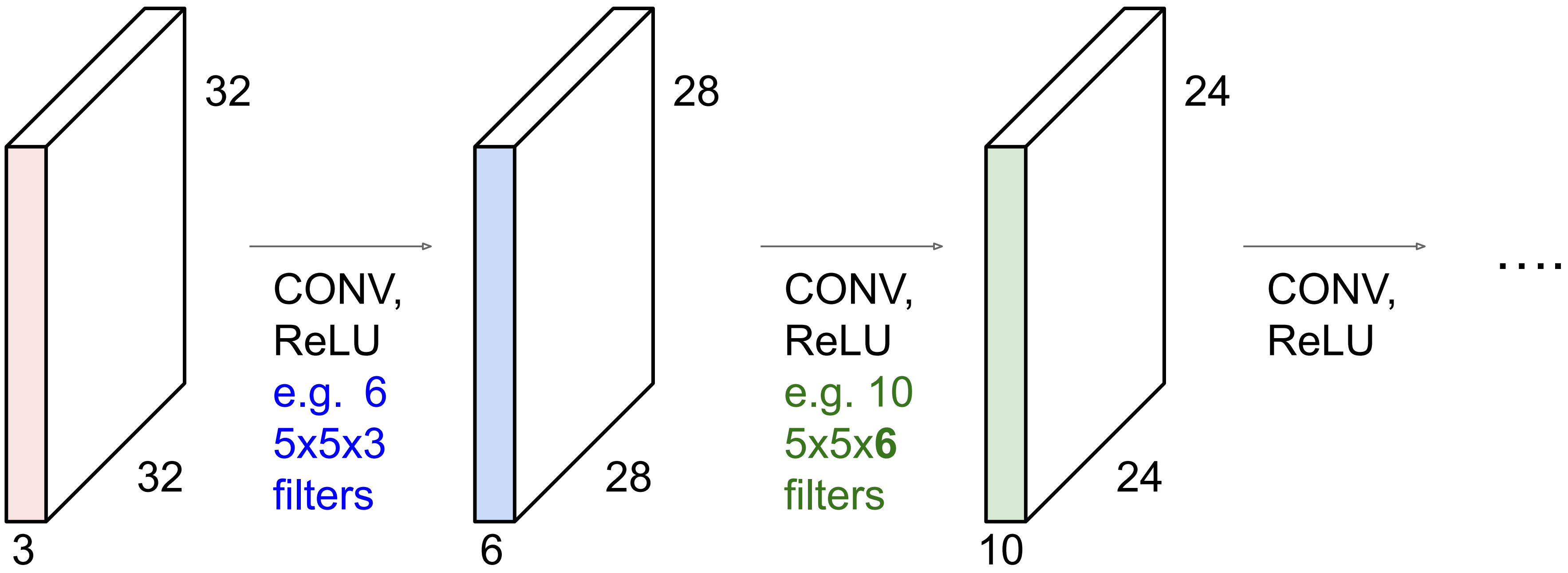


We stack these up to get a "new image" of size 28x28x6!

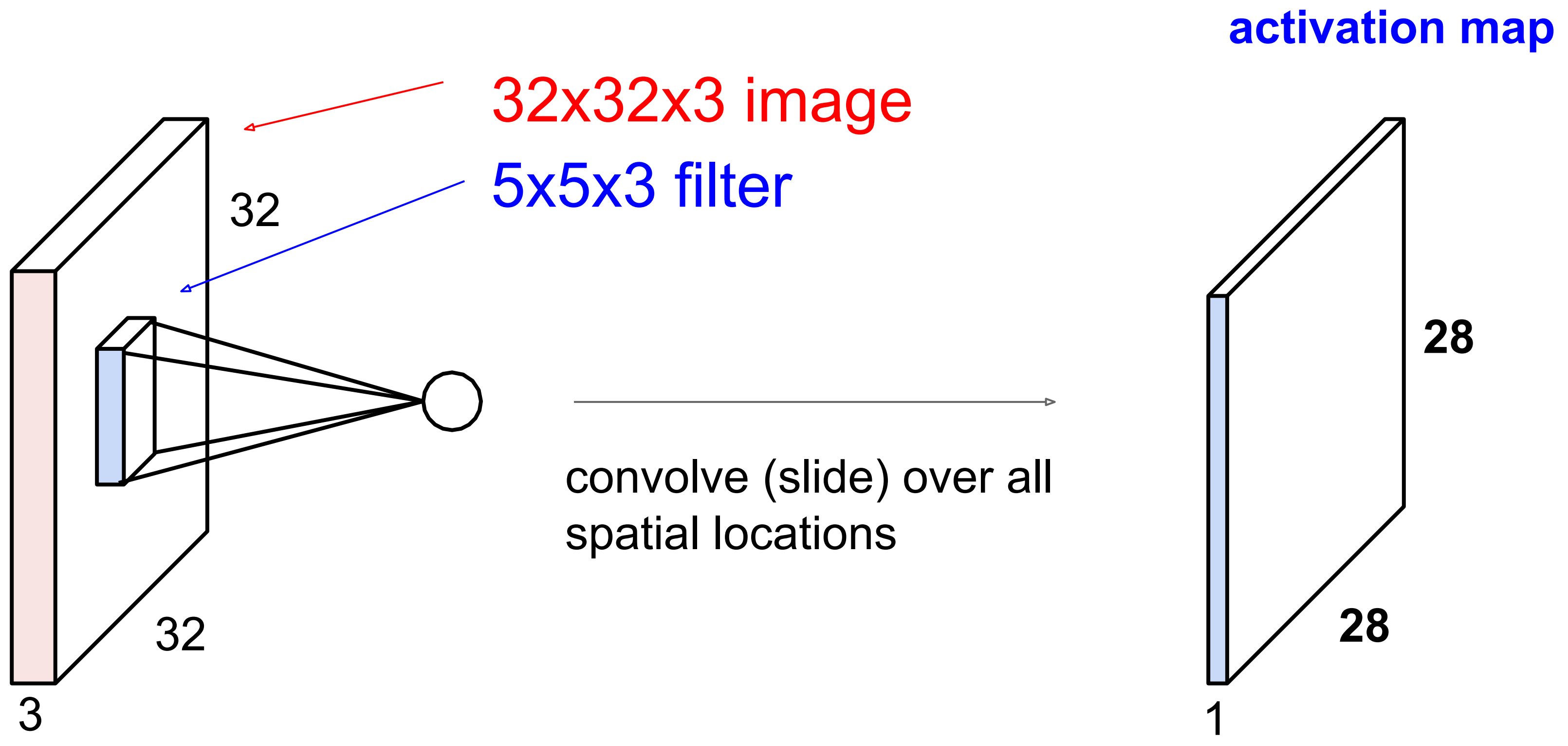
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



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

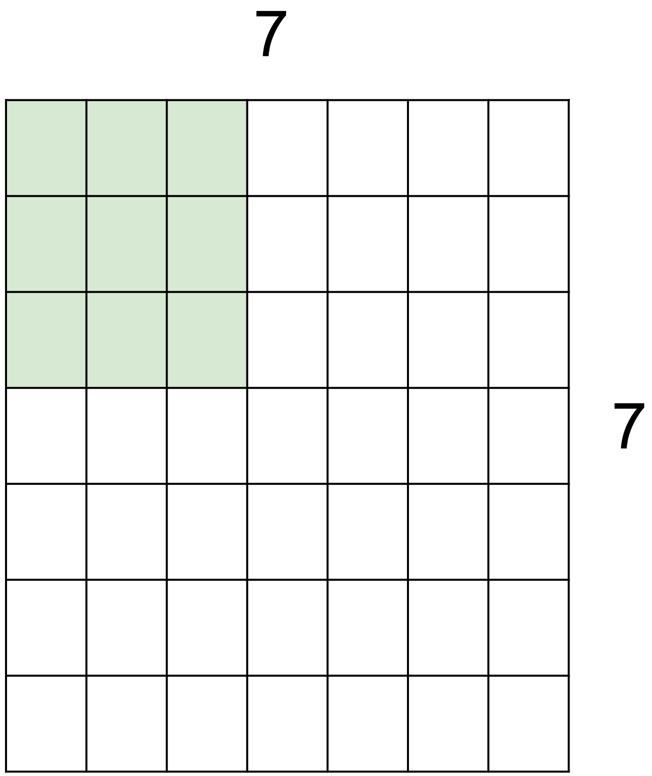


A closer look at spatial dimensions:

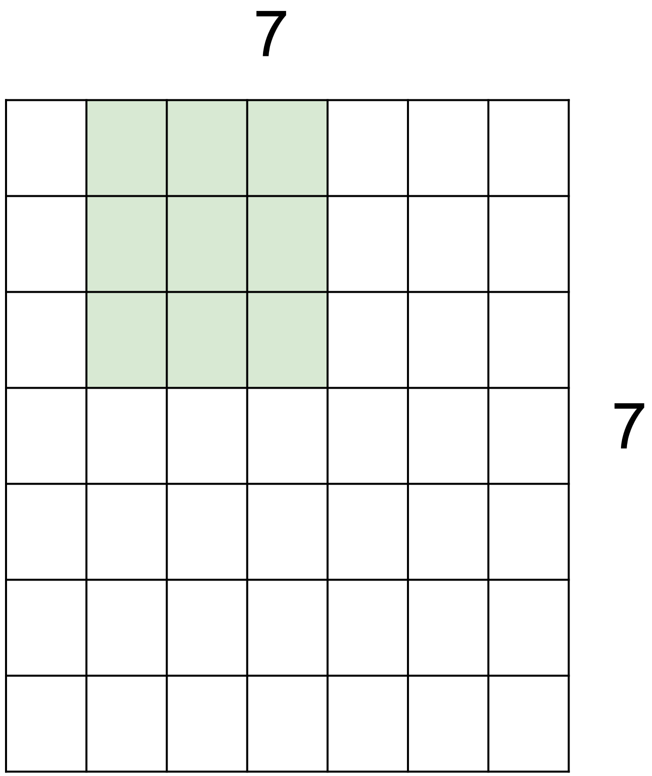


A closer look at spatial dimensions:

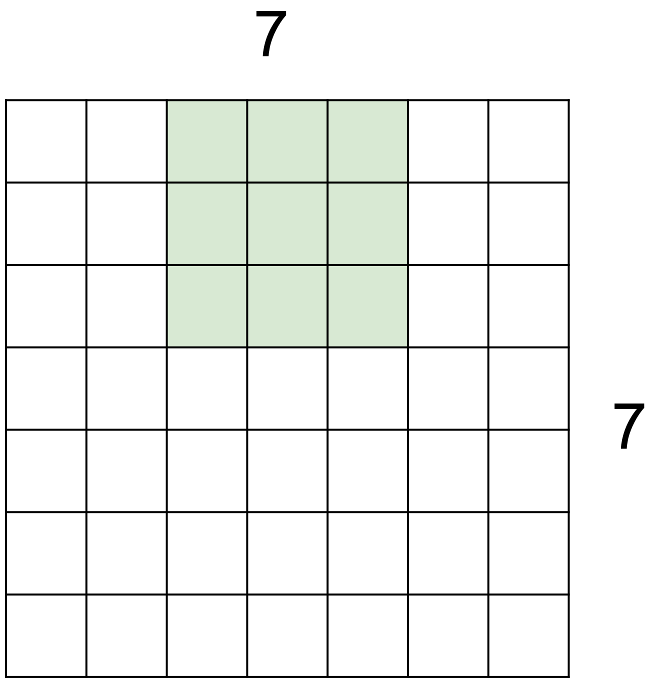
- 7x7 input (spatially) assume 3x3 filter



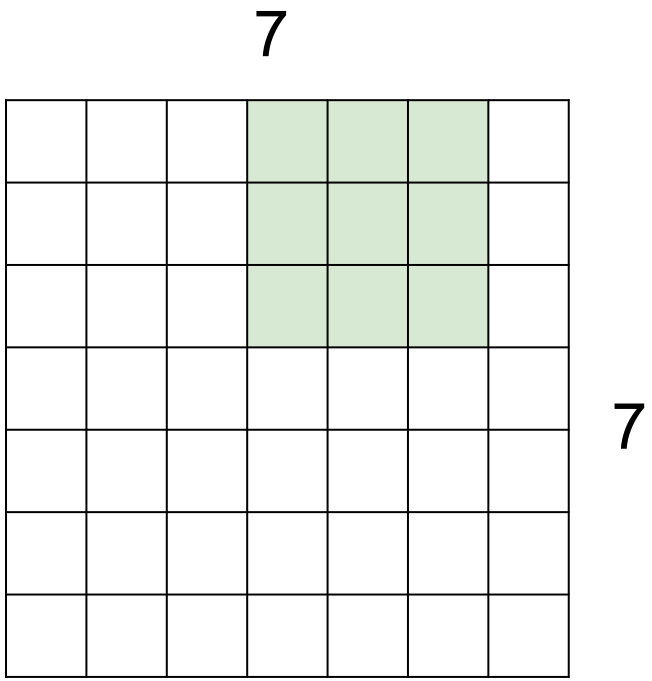
A closer look at spatial dimensions:



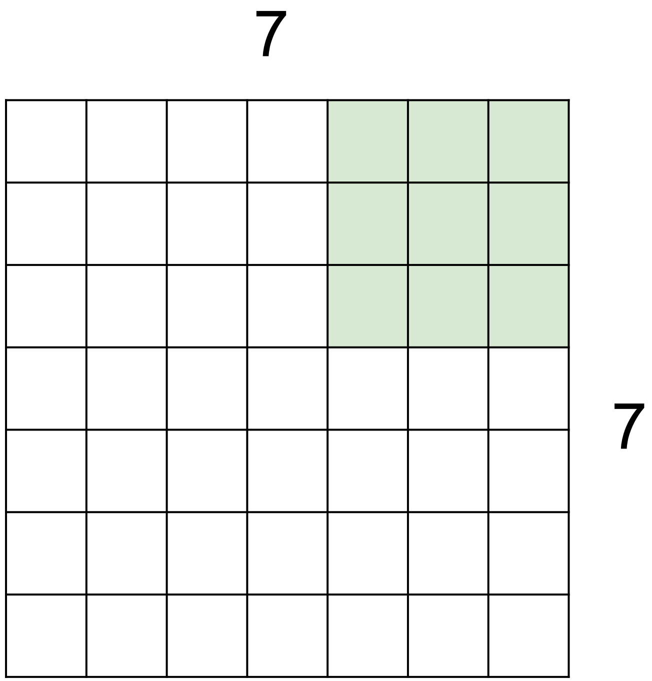
A closer look at spatial dimensions:



A closer look at spatial dimensions:

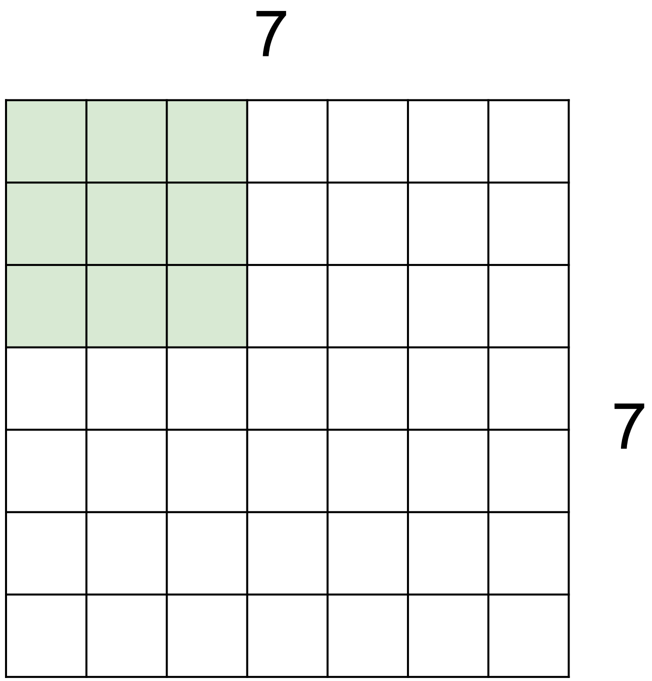


A closer look at spatial dimensions:



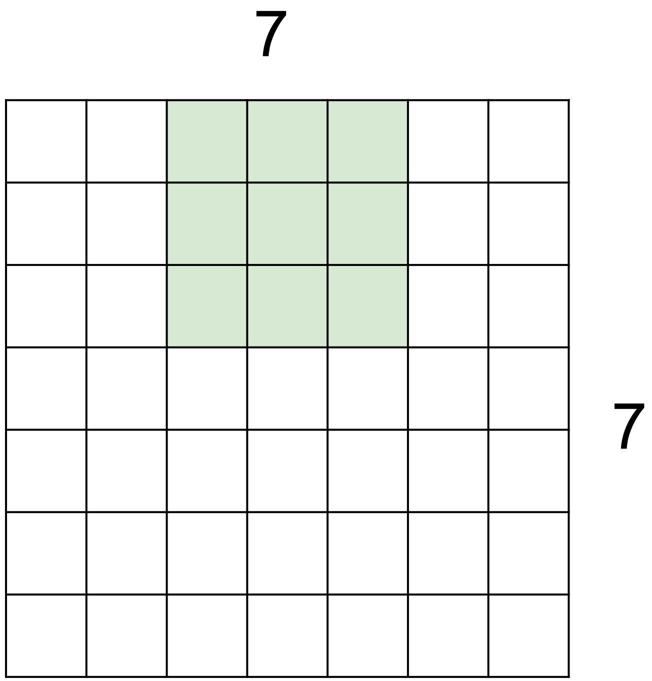
=> 5x5 output

A closer look at spatial dimensions:



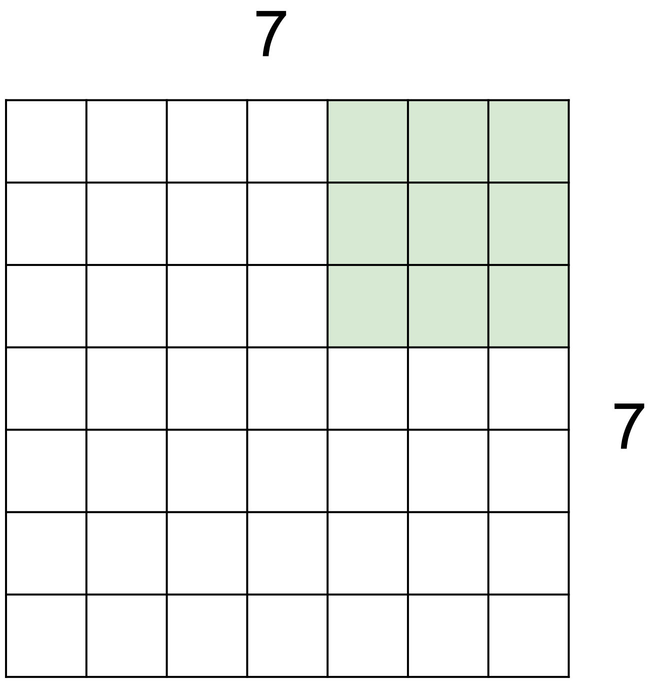
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



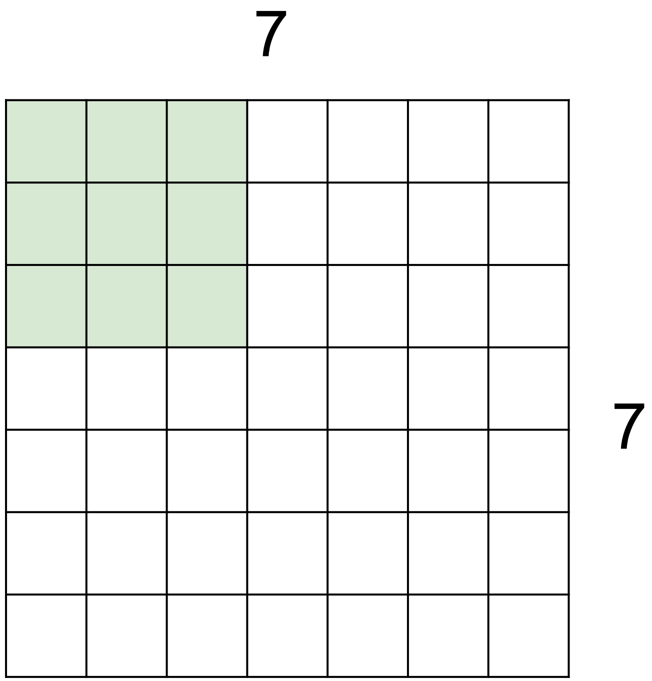
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



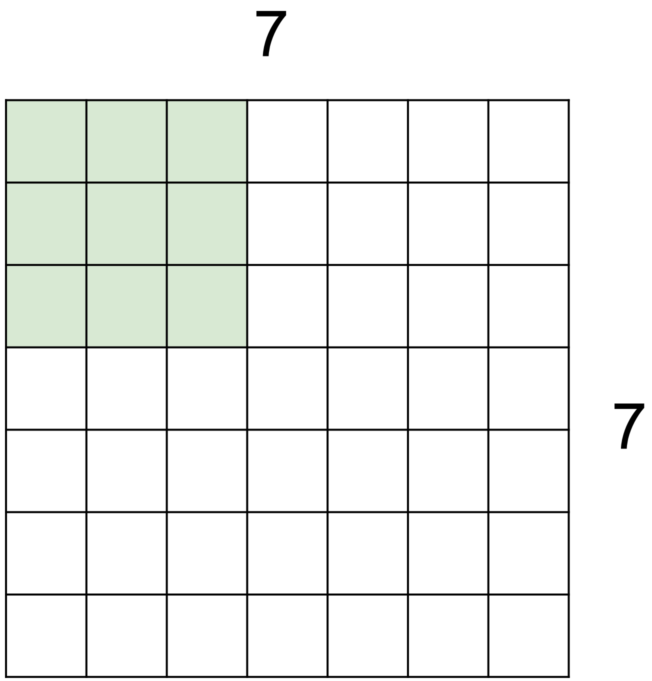
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



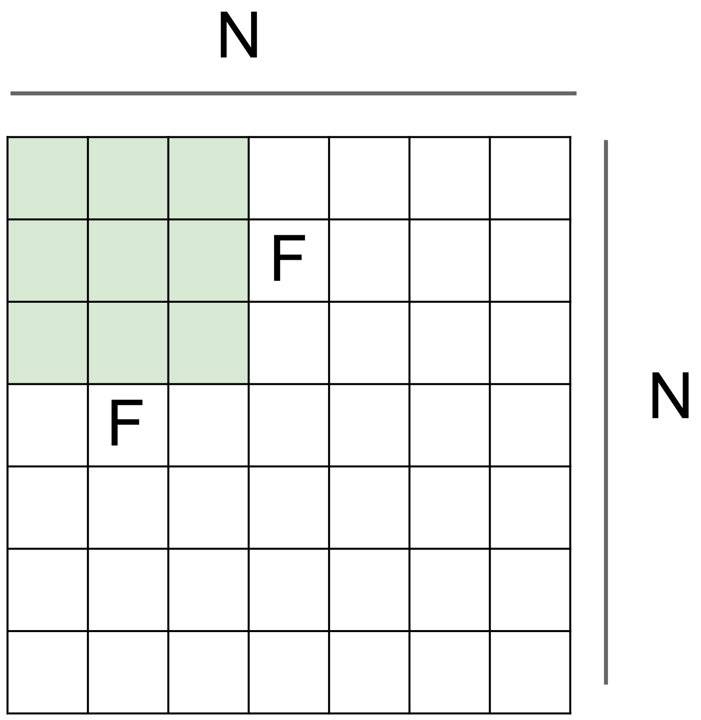
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

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.



Output size:

(N - F) / stride + 1

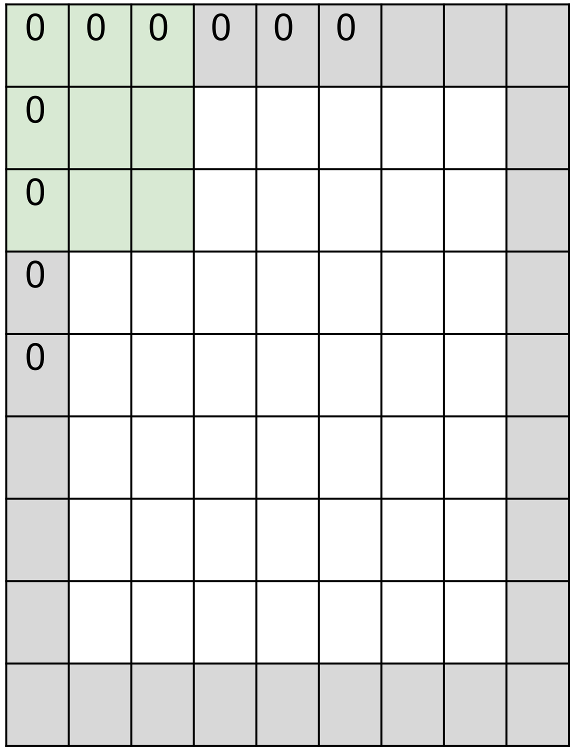
e.g. N = 7, F = 3:

stride 1 => (7 - 3)/1 + 1 = 5

stride 2 => (7 - 3)/2 + 1 = 3

stride 3 => (7 - 3)/3 + 1 = 2.33 :\

In practice: Common to zero pad the border

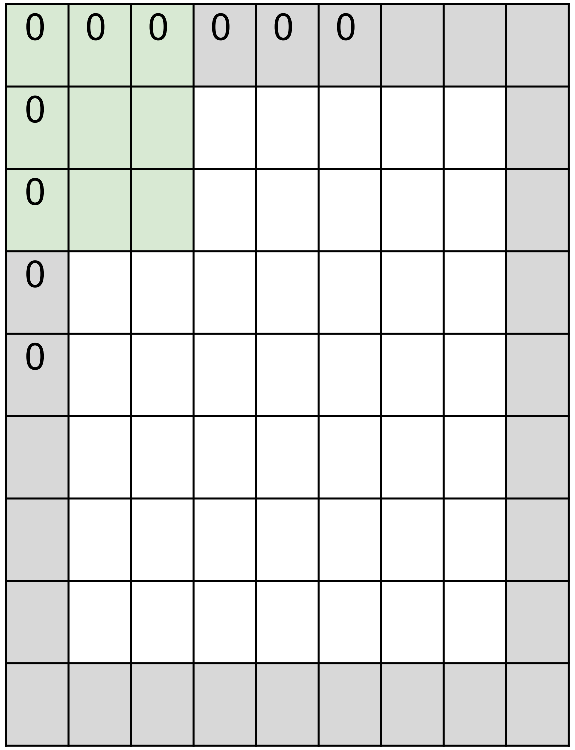


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

(recall:)

(N - F) / stride + 1

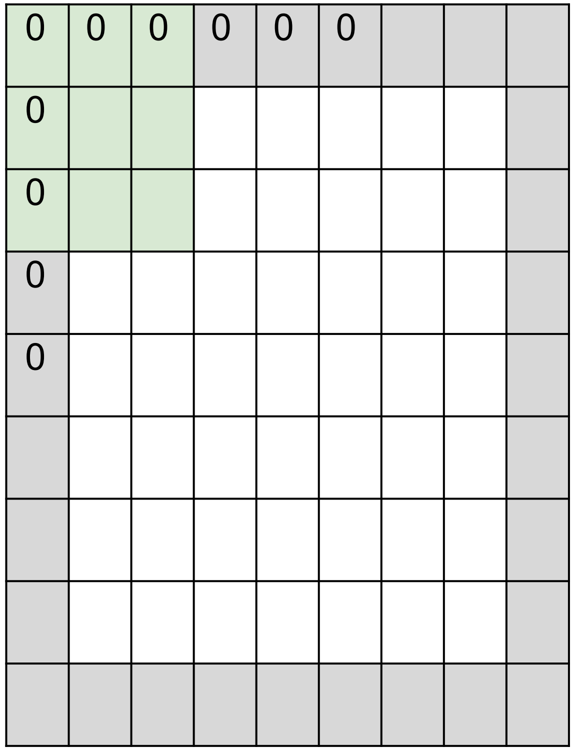
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

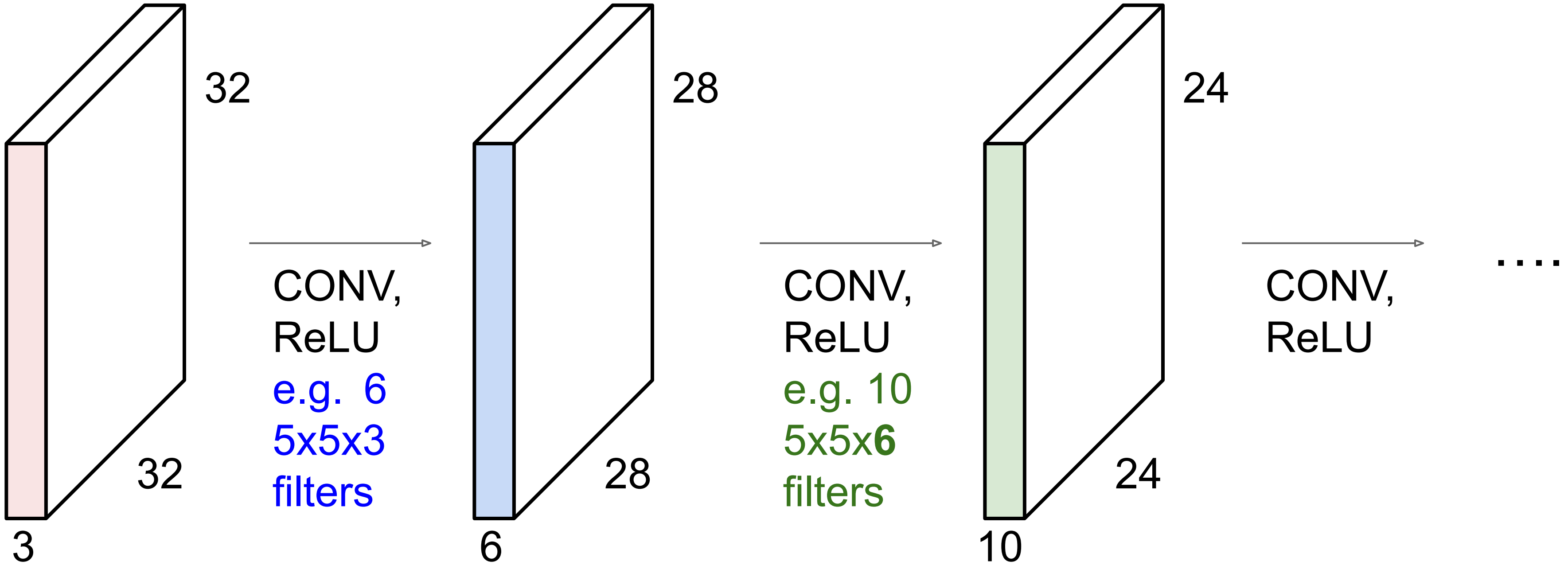
In practice: Common to zero pad the border



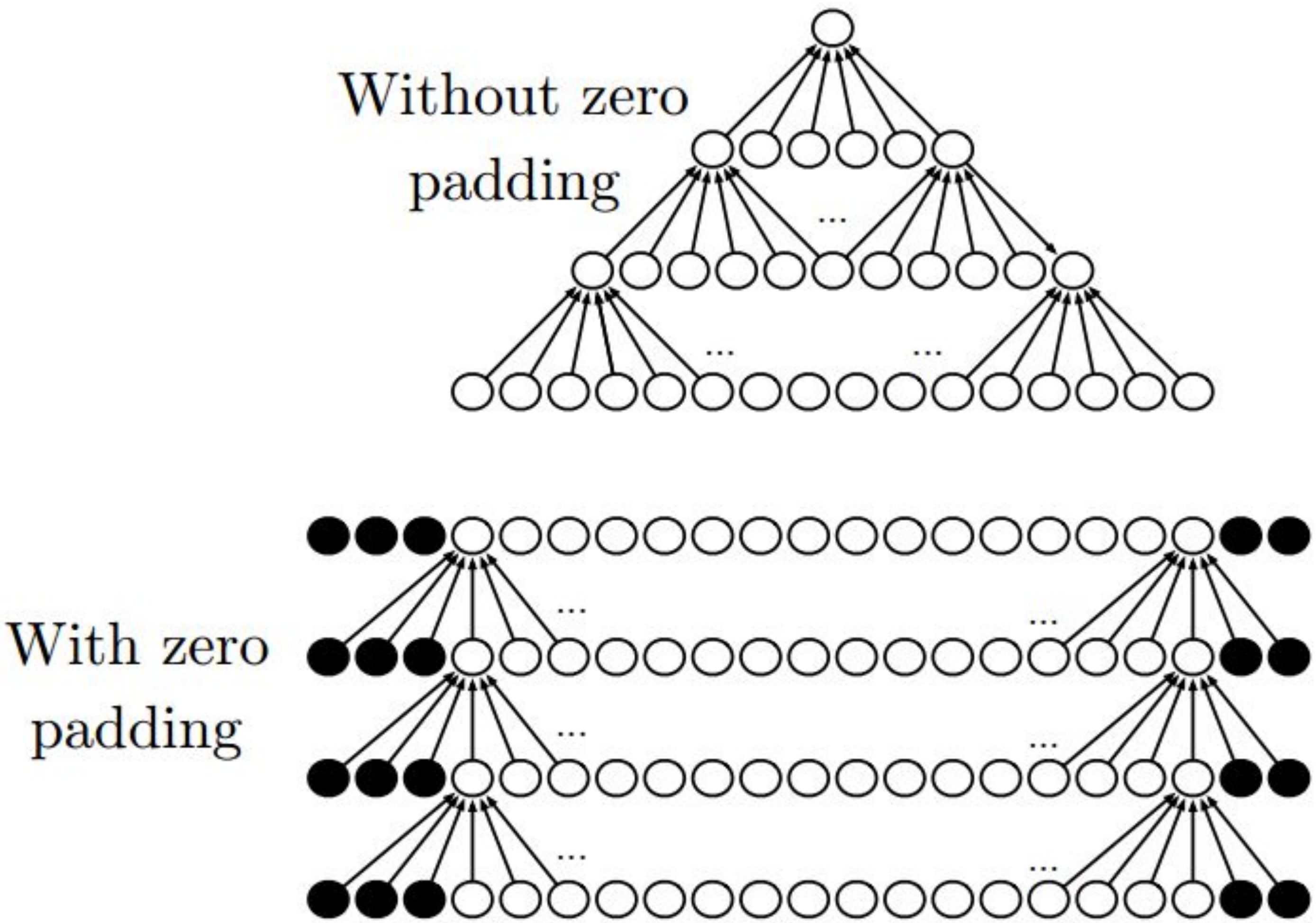
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not recommended - it often doesn’t work well.



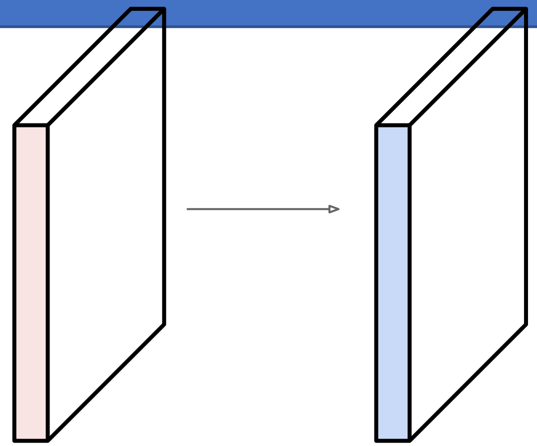
Zero Padding Helps to Control Size



Example time

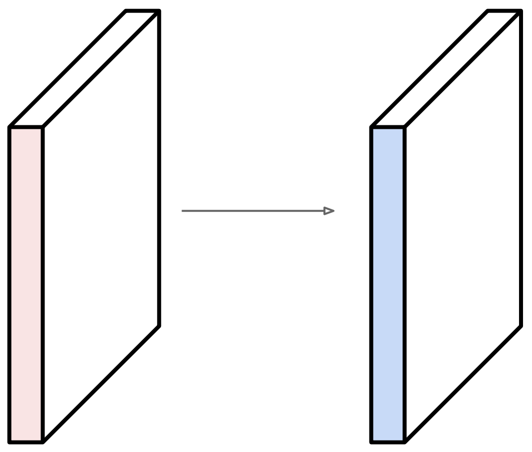
Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Output volume size: ?



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

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



Width of original figure

Filter width

Padding

$$steps = \frac{w - f + 2p}{s} + 1$$

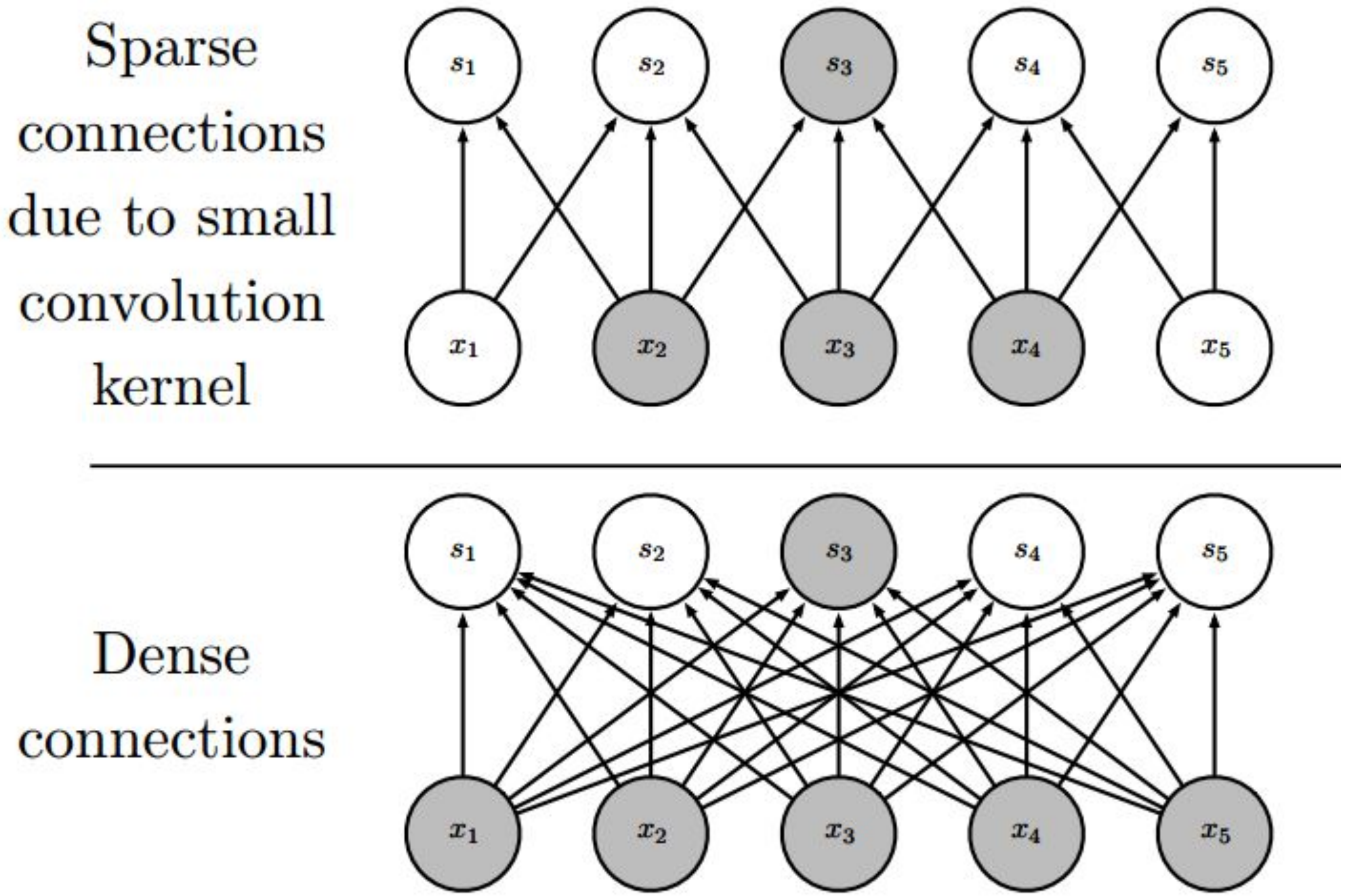
Stride

Setting the parameters

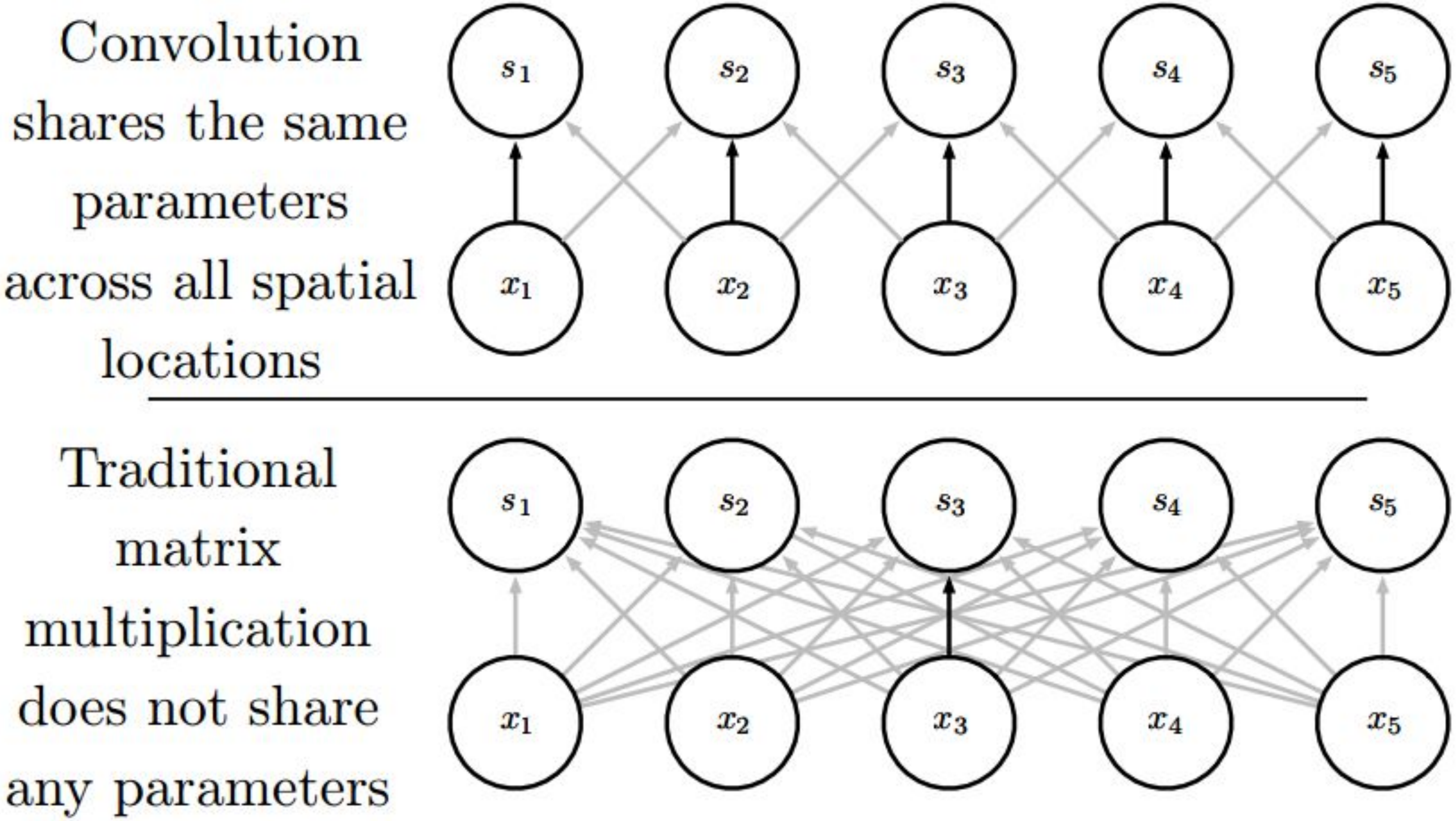
- In order to preserve the original image size, one should set *f*, *p* and *s* such that steps=original image size.

$$steps = \frac{\overset{\text{Width of original figure}}{w} - \overset{\text{Filter width}}{f} + \overset{\text{Padding}}{2p}}{\underset{\text{Stride}}{s}} + 1$$

CNN vs Plain NN: Sparse Connectivity



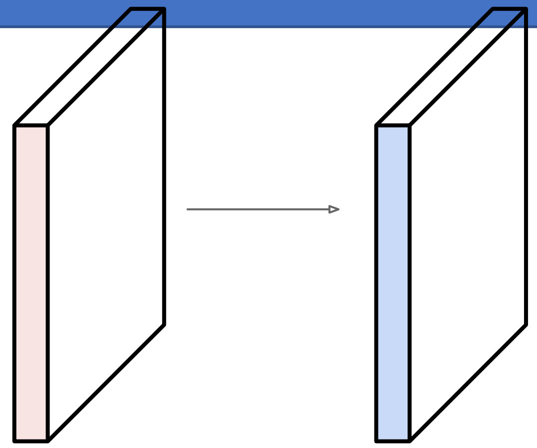
CNN vs Plain NN: Parameter Sharing



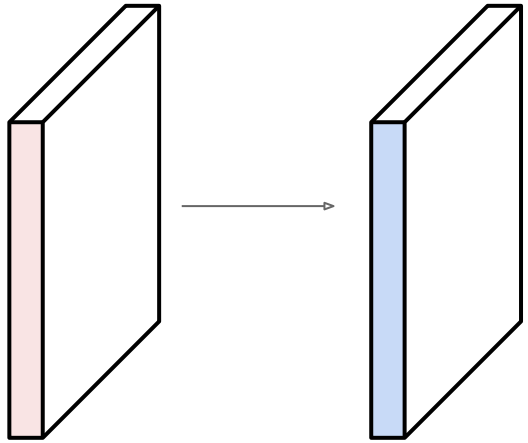
An Example

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

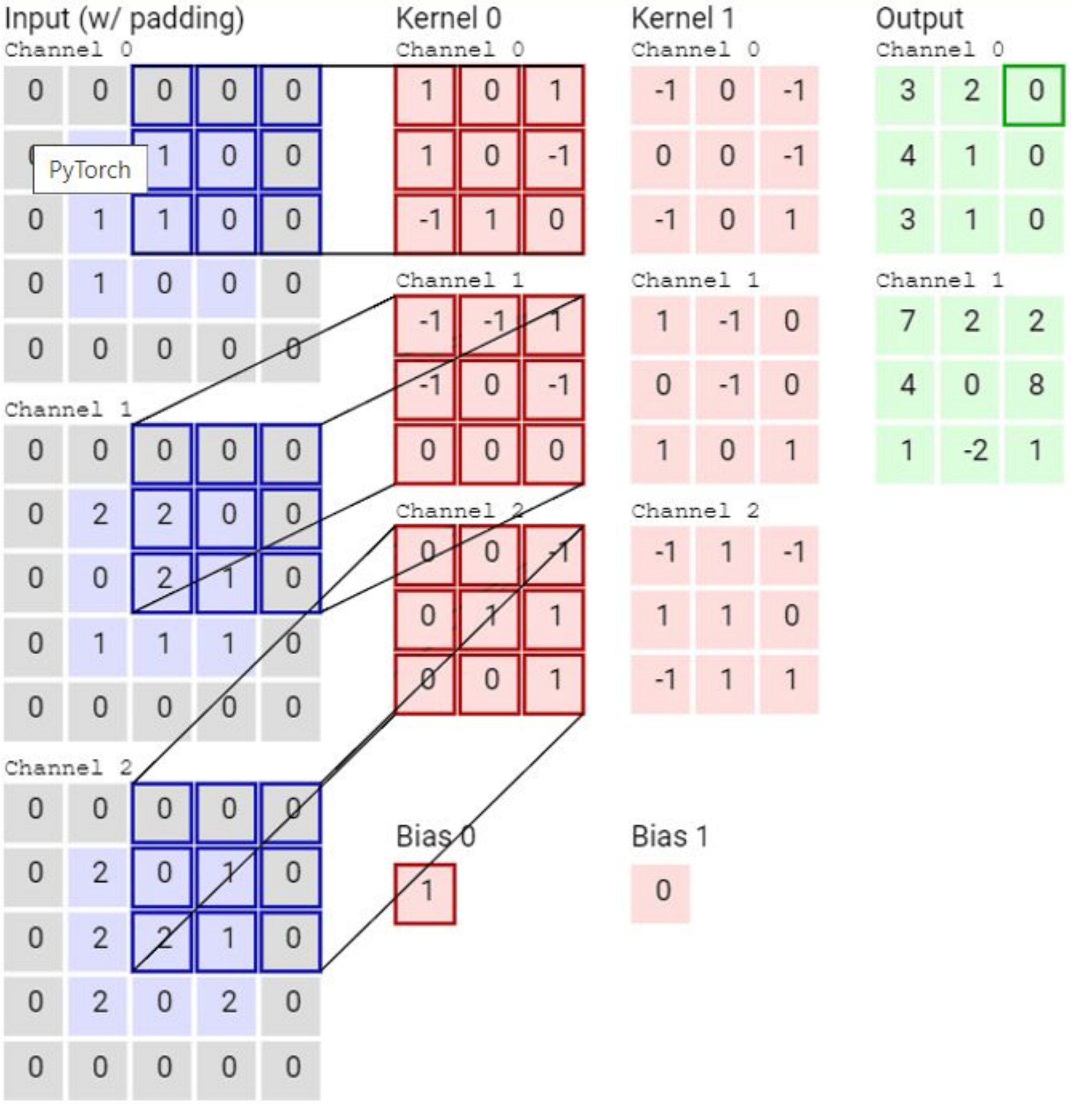
Number of parameters in this layer?

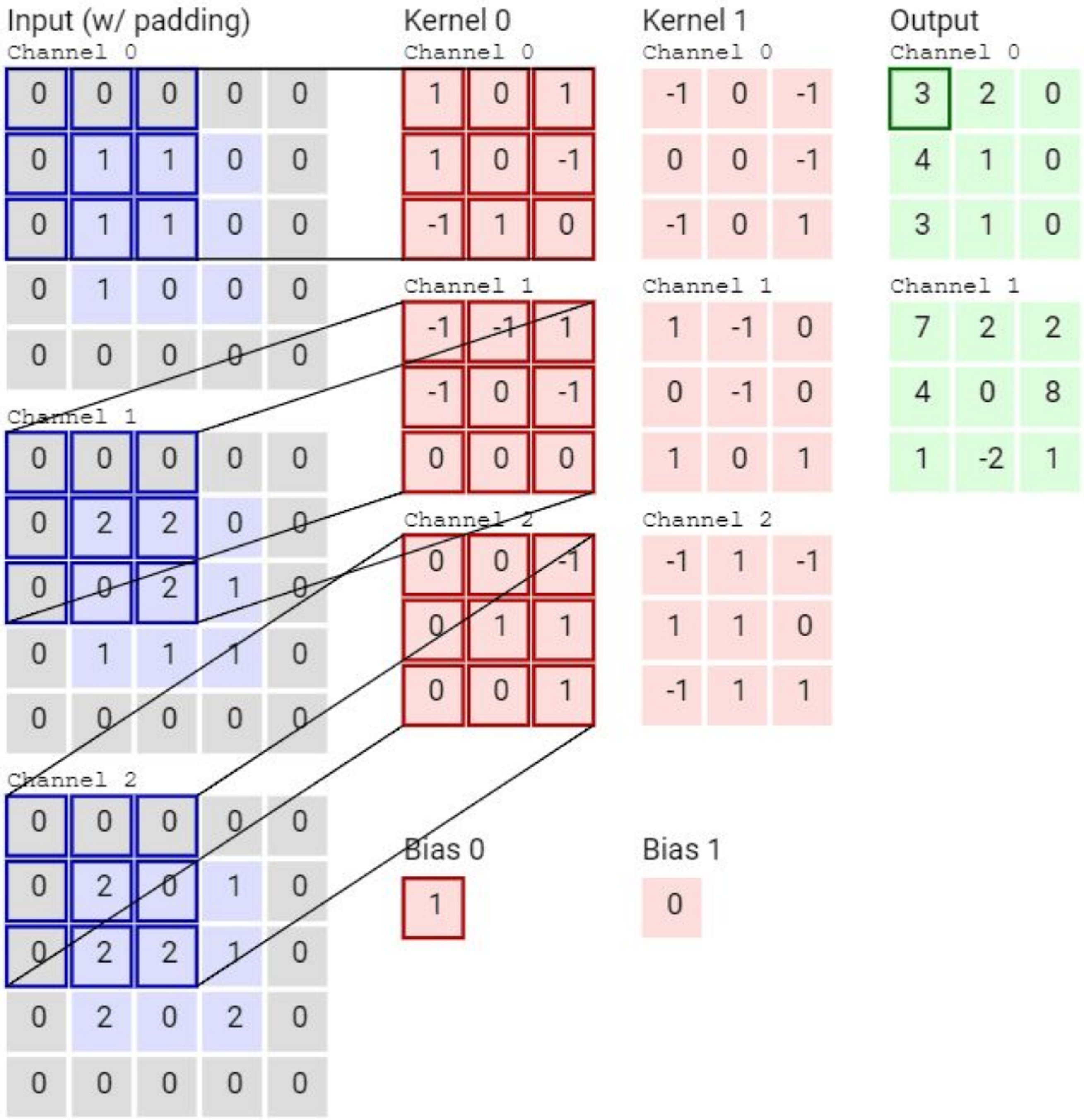


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2



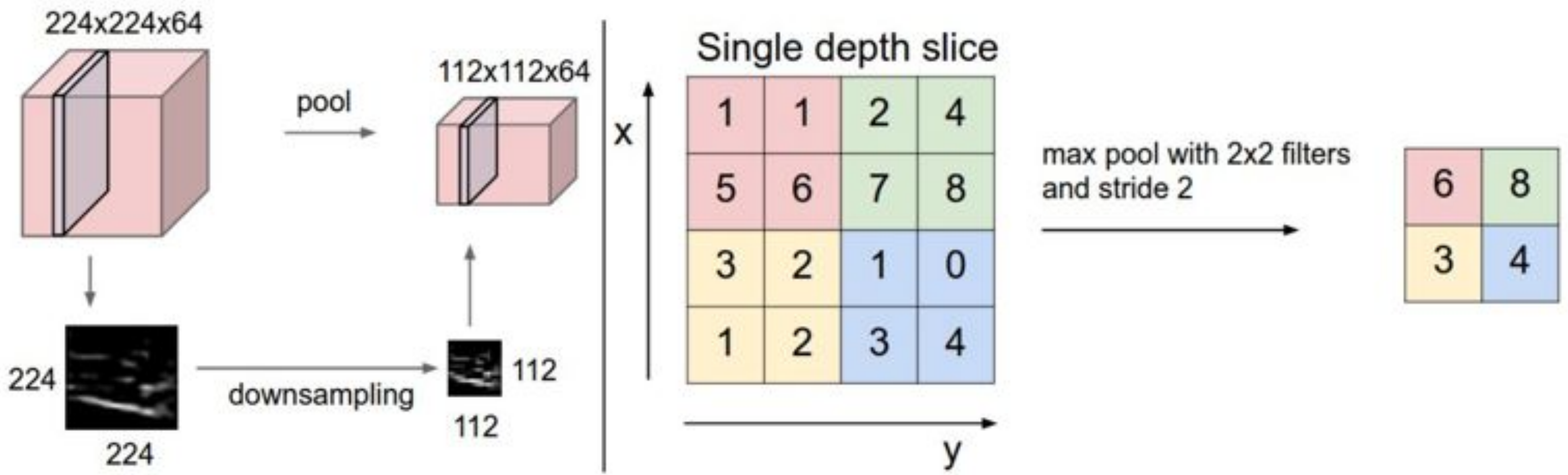
Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**


Input (w/ padding)
Channel 0
0 0 0 0 0
PyTorch 1 0 0
0 1 1 0 0
0 1 0 0 0
0 0 0 0 0
Channel 1
0 0 0 0 0
0 2 2 0 0
0 0 2 1 0
0 1 1 1 0
0 0 0 0 0
Channel 2
0 0 0 0 0
0 2 0 1 0
0 2 2 1 0
0 2 0 2 0
0 0 0 0 0
Kernel 0
Channel 0
1 0 1
1 0 -1
-1 1 0
Channel 1
-1 -1 1
-1 0 -1
0 0 0
Channel 2
0 0 -1
0 1 1
0 0 1
Bias 0
1
Kernel 1
Channel 0
-1 0 -1
0 0 -1
-1 0 1
Channel 1
1 -1 0
0 -1 0
1 0 1
Channel 2
-1 1 -1
1 1 0
-1 1 1
Bias 1
0
Output
Channel 0
3 2 0
4 1 0
3 1 0
Channel 1
7 2 2
4 0 8
1 -2 1



Input (w/ padding)
Channel 0
0 0 0 0 0
0 1 1 0 0
0 1 1 0 0
0 1 0 0 0
0 0 0 0 0
Channel 1
0 0 0 0 0
0 2 2 0 0
0 0 2 1 0
0 1 1 1 0
0 0 0 0 0
Channel 2
0 0 0 0 0
0 2 0 1 0
0 2 2 1 0
0 2 0 2 0
0 0 0 0 0
Kernel 0
Channel 0
1 0 1
1 0 -1
-1 1 0
Channel 1
-1 -1 1
-1 0 -1
0 0 0
Channel 2
0 0 -1
0 1 1
0 0 1
Bias 0
1
Kernel 1
Channel 0
-1 0 -1
0 0 -1
-1 0 1
Channel 1
1 -1 0
0 -1 0
1 0 1
Channel 2
-1 1 -1
1 1 0
-1 1 1
Bias 1
0
Output
Channel 0
3 2 0
4 1 0
3 1 0
Channel 1
7 2 2
4 0 8
1 -2 1

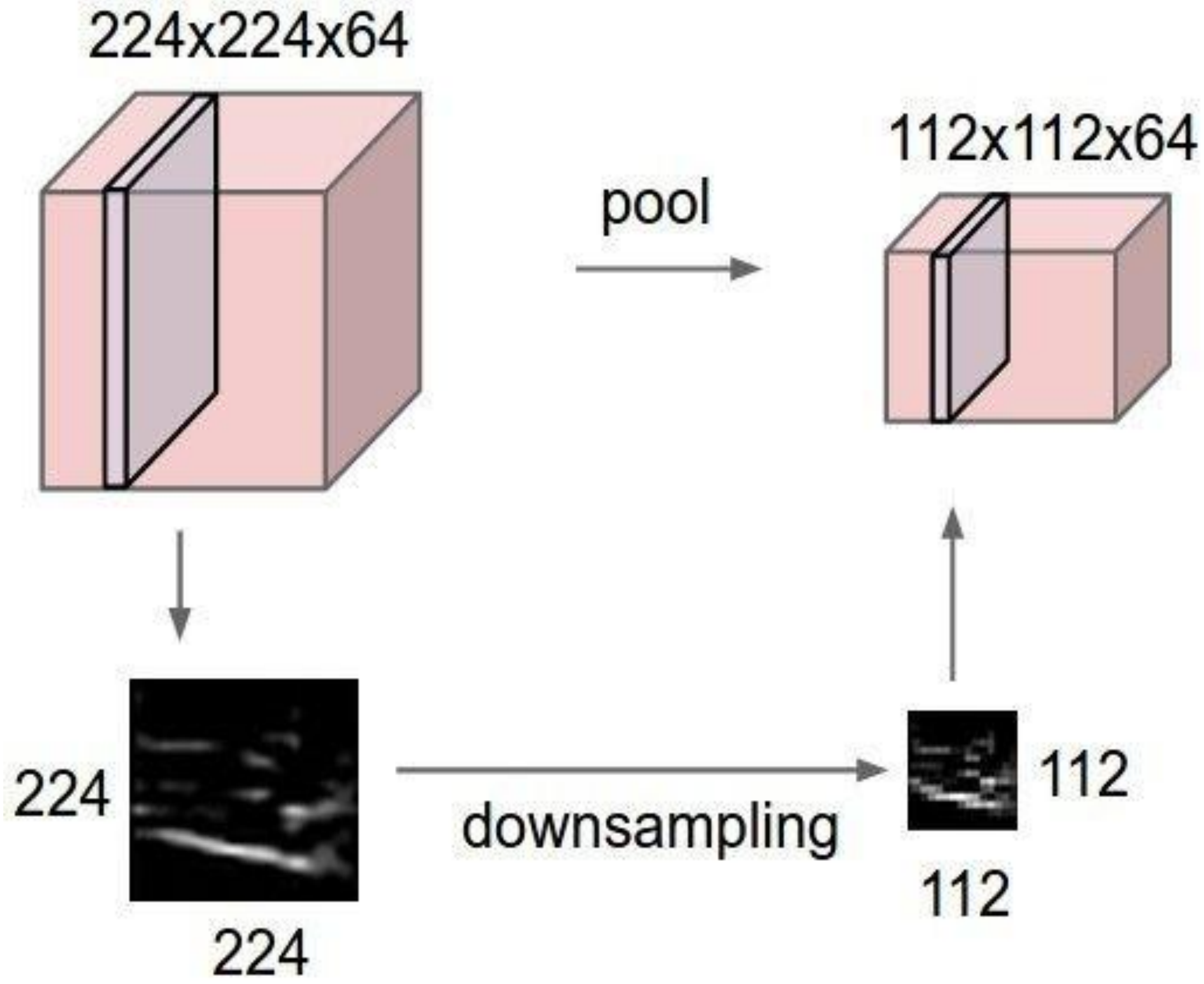
Max Pooling



224x224x64
pool
112x112x64
224
224
downsampling
112
112
x
y
Single depth slice
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters and stride 2
6 8
3 4

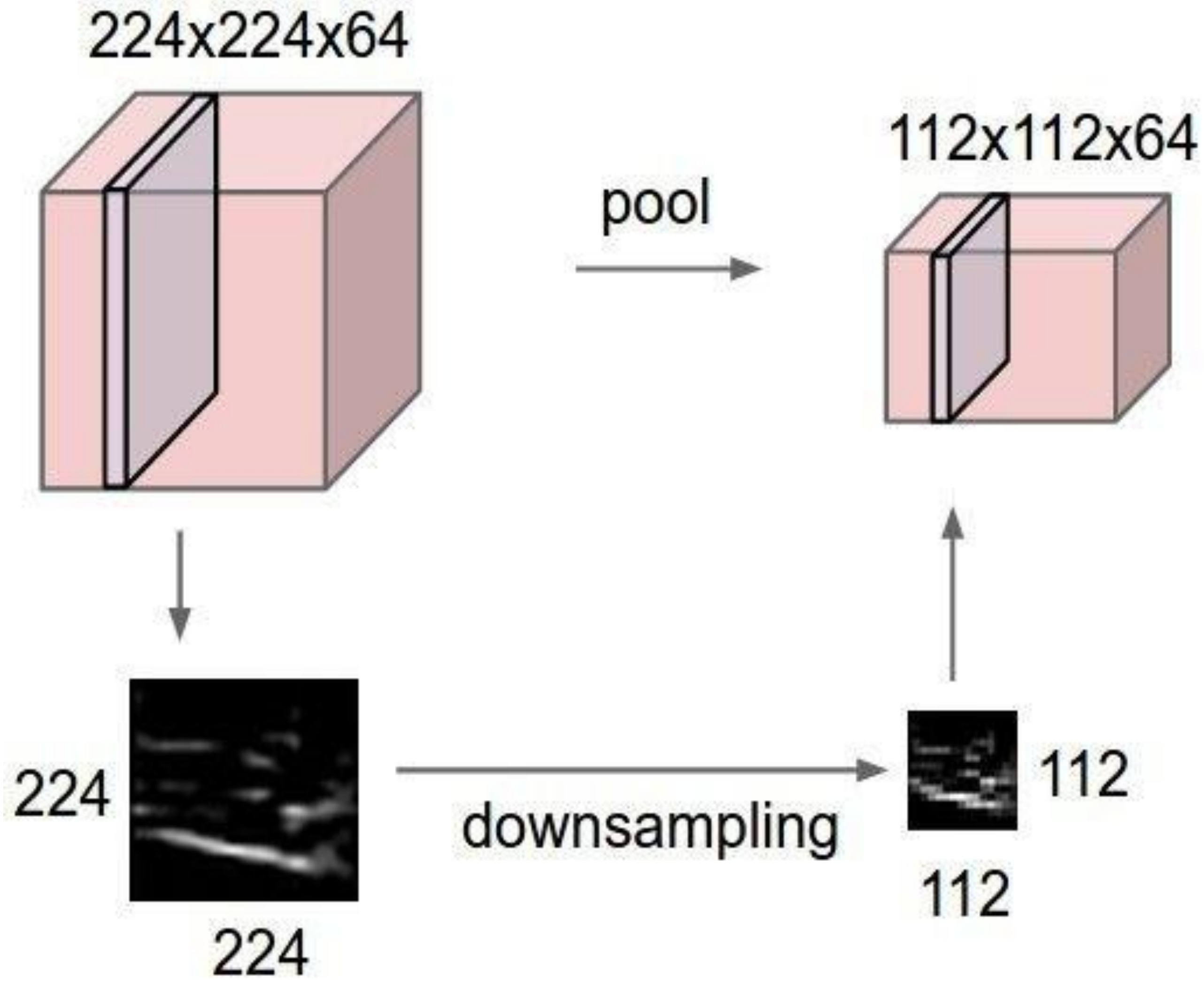
Pooling layer (a.k.a Subsampling Layer)

- What are the benefits?
- How does it apply to multiple activation maps (filters)?

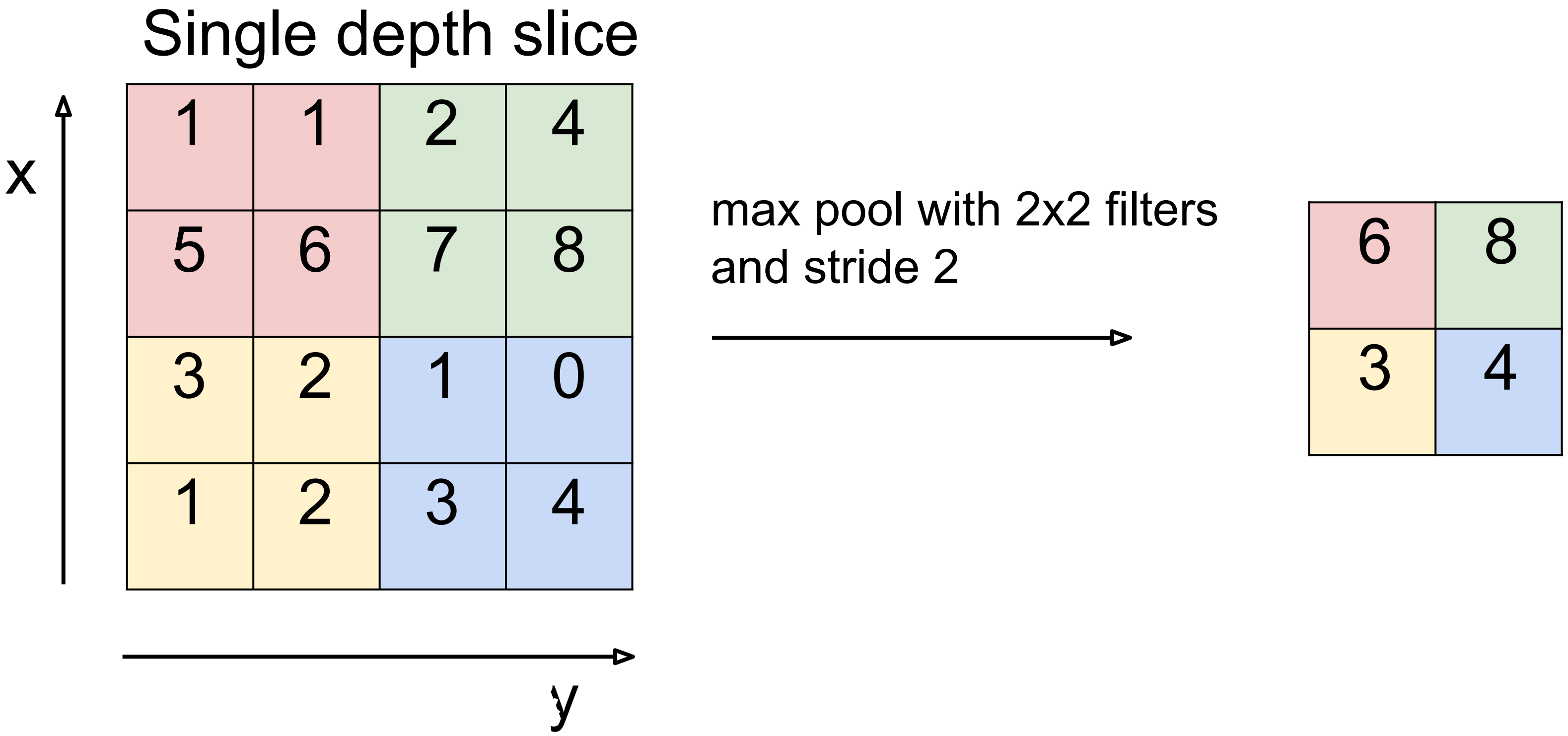


Pooling layer (a.k.a Subsampling Layer)

- Makes the representations smaller and more manageable
- Operates over each activation map independently

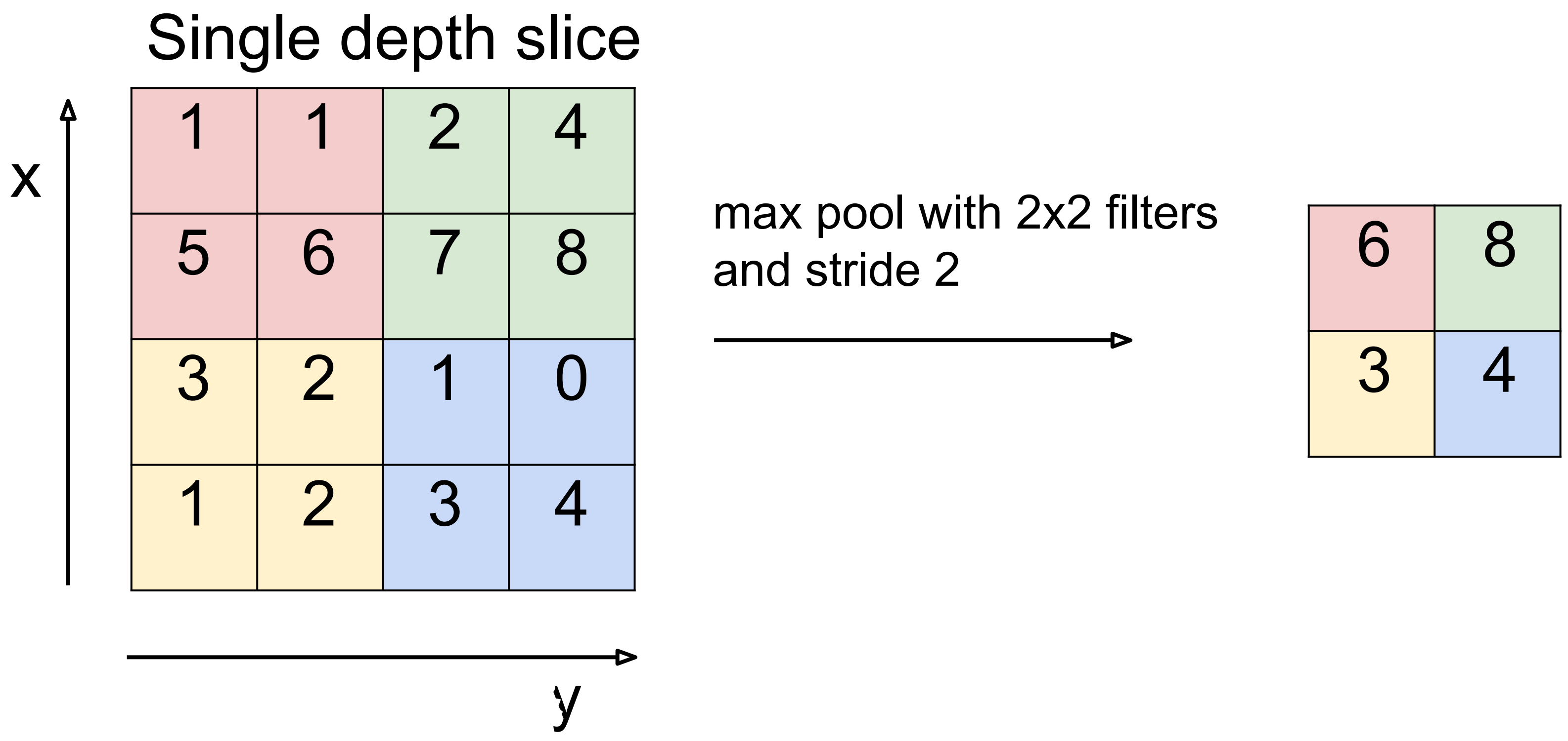


Max Pooling – why does it work better than Average Pooling?

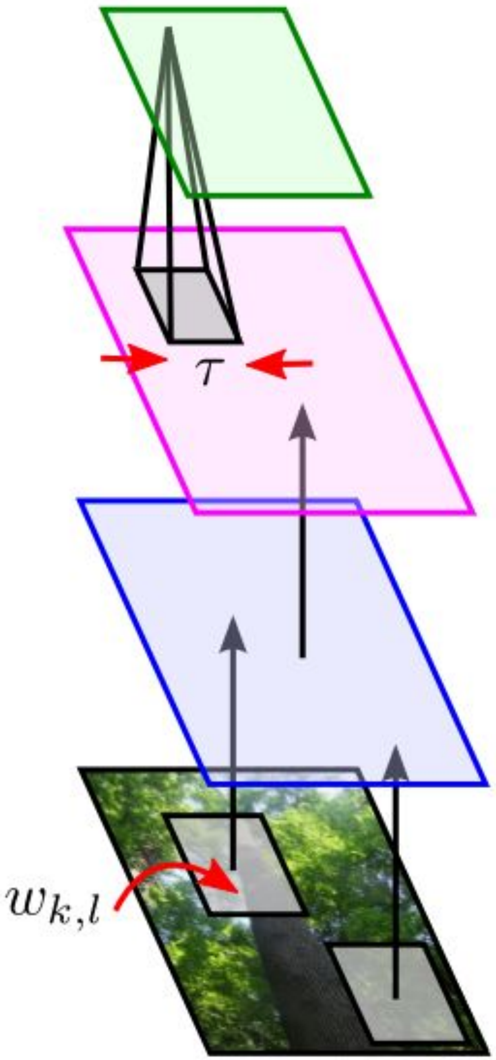


Max Pooling – why does it work better than Average Pooling?

- features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map
- The maximal presence of different features is often more informative than their average presence



Building block of a convolutional neural network



$$x_{i,j} = \max_{|k|<\tau, |l|<\tau} y_{i-k,j-l}$$

mean or subsample also used

pooling stage

$$y_{i,j} = f(a_{i,j})$$

e.g. $f(a) = [a]_+$

$f(a) = \text{sigmoid}(a)$

non-linear stage

$$a_{i,j} = \sum_{k,l} w_{k,l} z_{i-k,j-l}$$

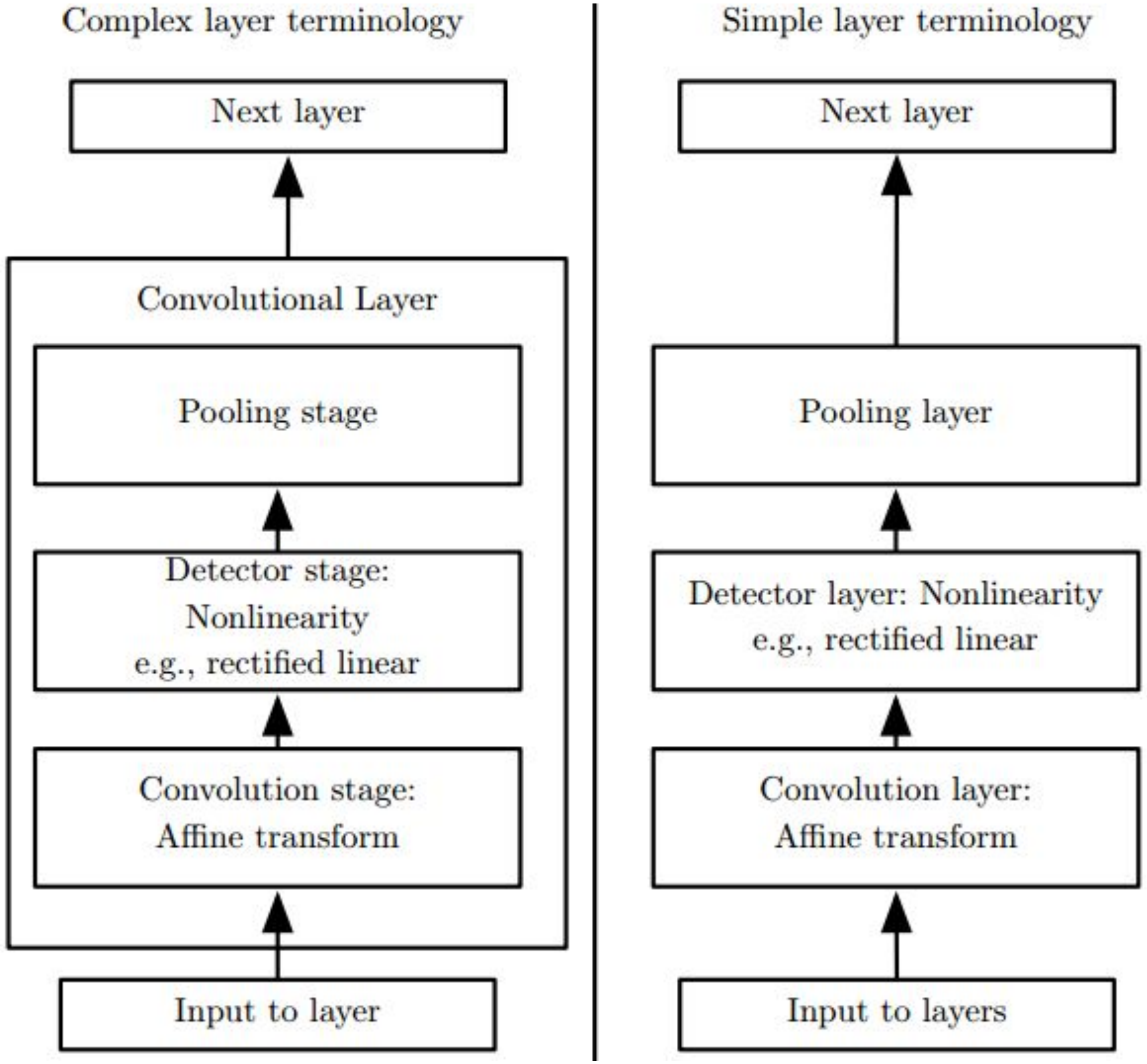
only parameters

convolutional stage

$z_{i,j}$

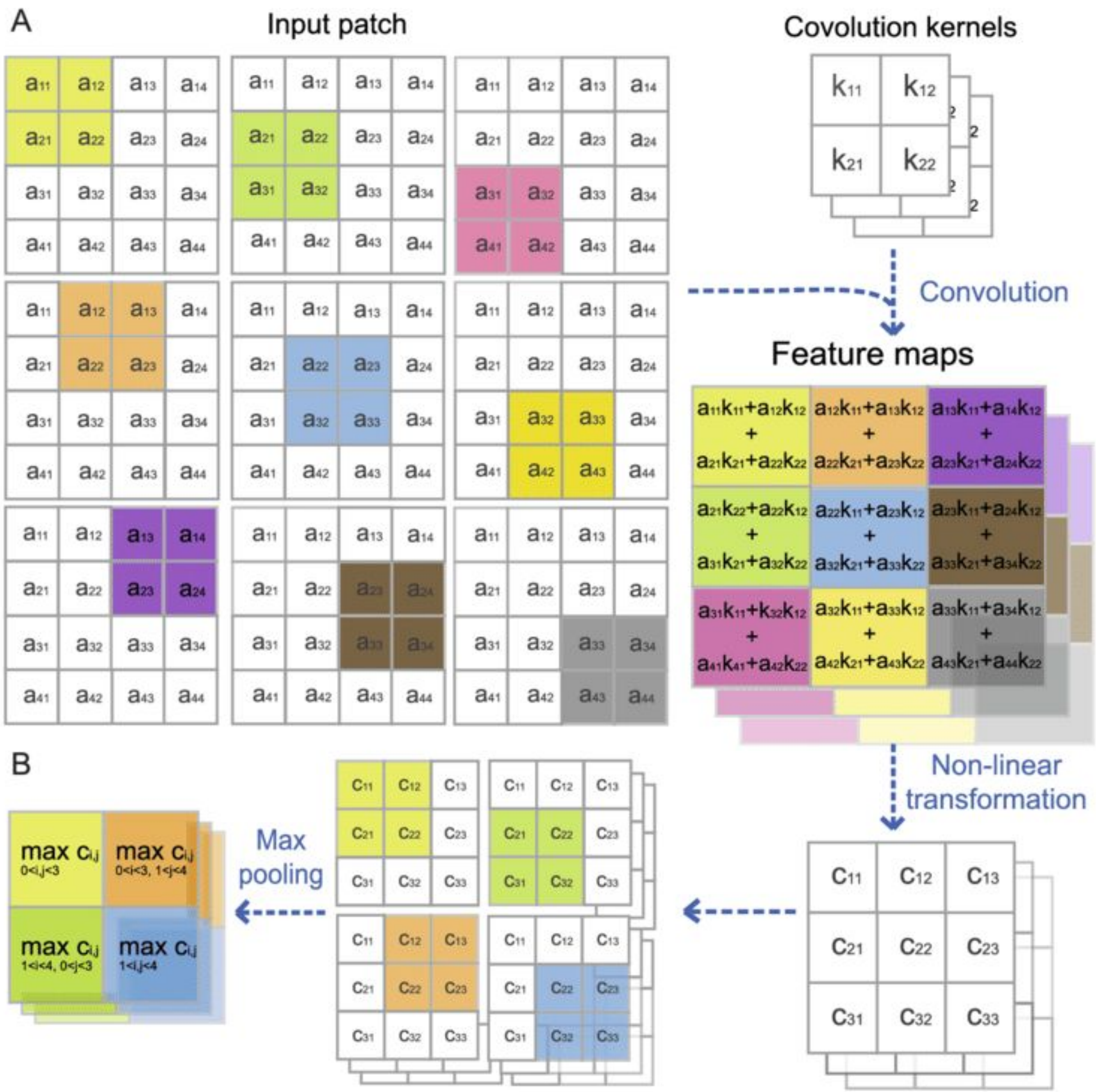
input image

CNN terminology



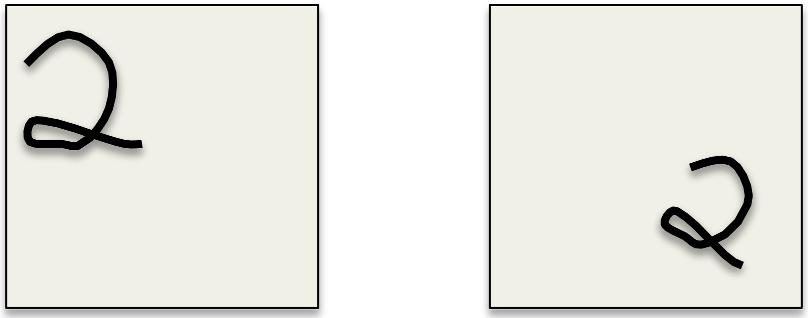
Convolutional Neural Networks

- Each layer applies different filters, possibly **hundreds or thousands** and combines their results
- During the training phase, a CNN **automatically** learns the values of its filters based on the task you want to perform
- Note that convolutional neural networks **share weights** – each filter is actually a set of weights that are identically used over **all** windows in the image

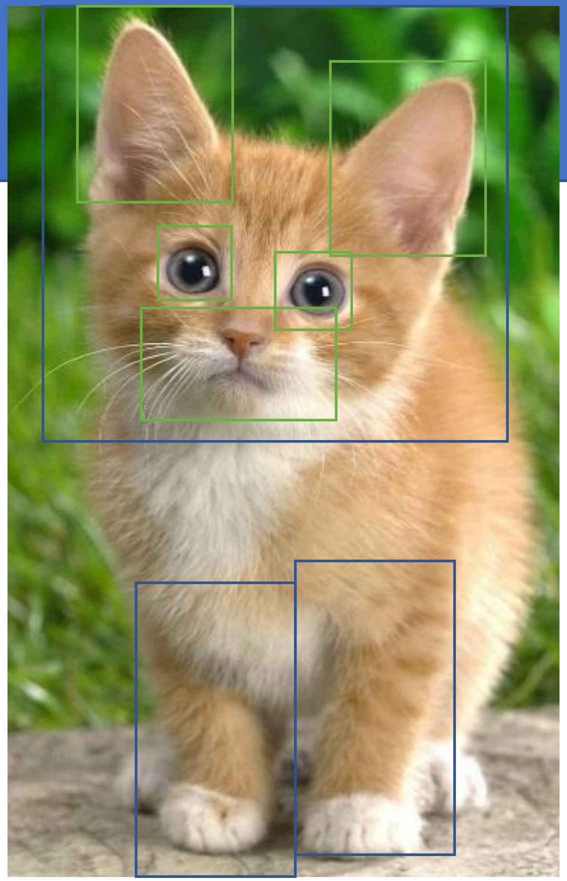

A
Input patch
Covolution kernels
Convolution
Feature maps
Non-linear
transformation
B
Max
pooling

Why is Objection Detection Difficult?

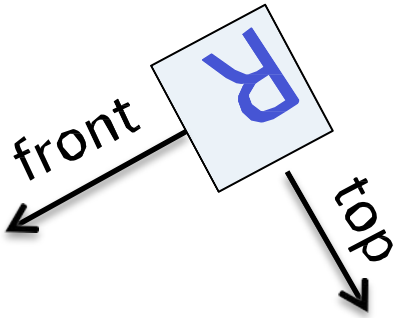
- Real scenes have multiple objects. These are often not segmented together.
- Occlusion
- Lightning
- Deformations
- Functionality – what makes a “Chair”?
- Invariance:
    - Translation
    - Rotation
    - Scale
    - Stretch
    - Sheer
- Mostly insensitive to rotation, but not always
- Different viewpoints



Possible Approaches



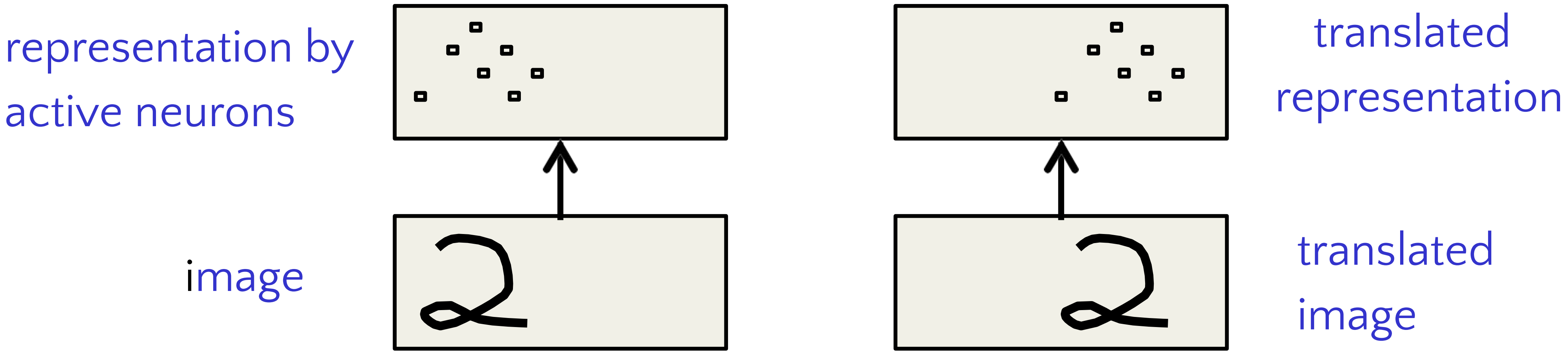
- Should we work in steps?
  - First put a bounding box
  - Then rotate and standardize
  - Finally classify
- Should we define an object through its constituent parts?
- Should we care about the location of sub objects?
- What happens if we combine features from different objects?
- Cognitive studies show we recognize the letter before we mentally rotate it



Slide credit: G. Hinton

What does replicating the feature detectors achieve?

- Equivariant activities: Replicated features do not make the neural activities invariant to translation. The activities are equivariant.



- Invariant knowledge: If a feature is useful in some locations during training, detectors for that feature will be available in all locations during testing.

The brute force normalization approach

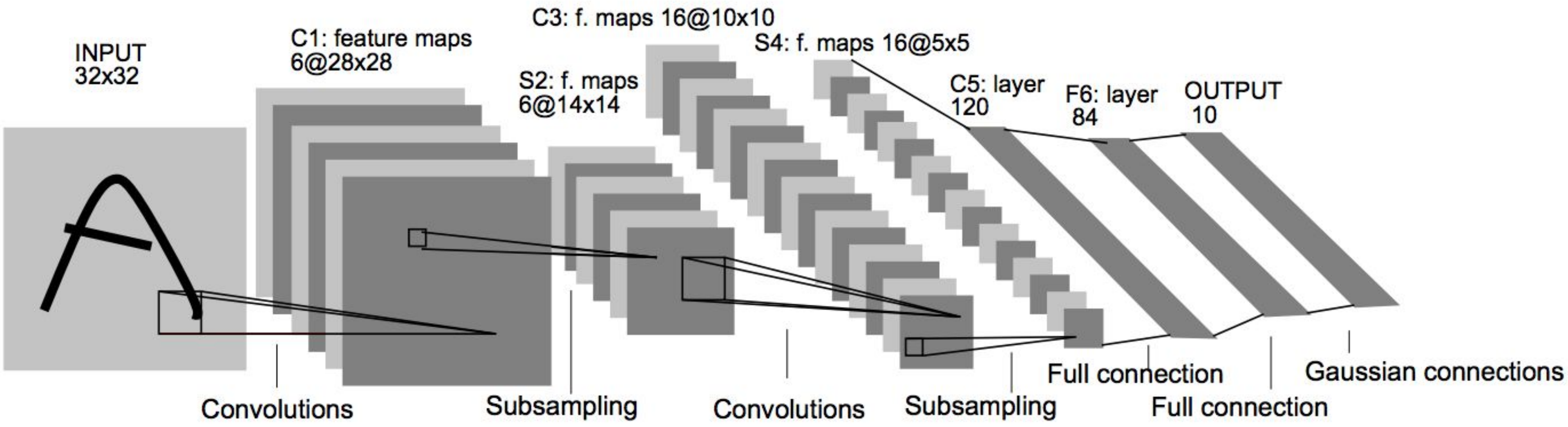
- When training the recognizer, use well-segmented, upright images to fit the correct box
- At test time try all possible boxes in a range of positions and scales
  - This approach is widely used for detecting upright things like faces and house numbers in unsegmented images
  - It is much more efficient if the recognizer can cope with some variation in position and scale so that we can use a coarse grid when trying all possible boxes

Slide credit: G. Hinton

Shifting our efforts

- The neural network is supposed to “free” us from the need to engineer features for the problem
- Instead, we engineer the network – give it power to account for things we deem important
- The advantage of DL in these cases – it improves much more with more data
- We can also use pre-knowledge for coming up with more data

LeNet



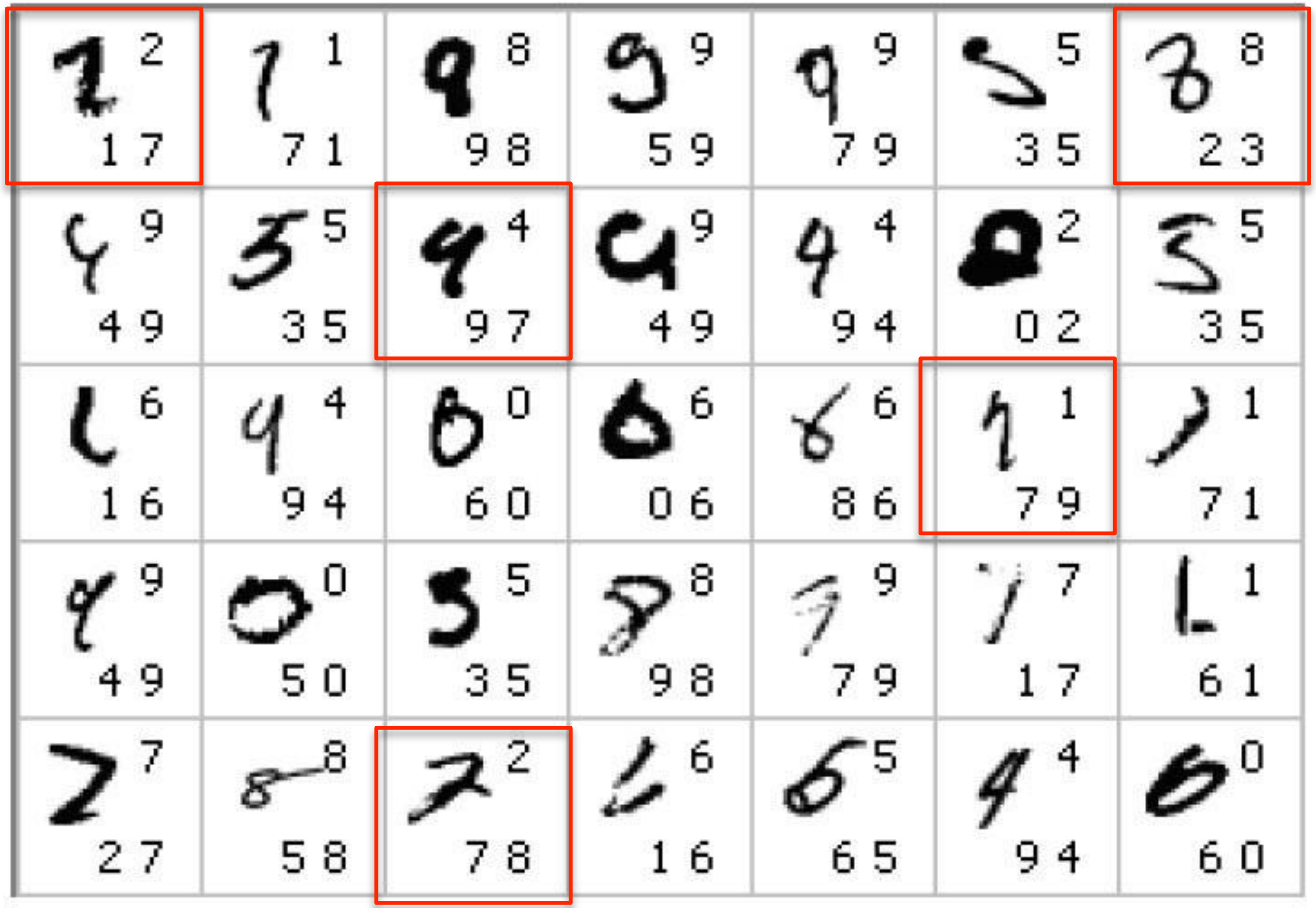
- Created by Yann LeCun in the 1990's (1998)
- Recognize hand-written digits
- Used to process 10% of the checks in the US at the time
- Required clean images, properly rotated and cropped
- 60,000 parameters
- Introduced a common architecture pattern – convolution layers with pooling followed by fully

The brute force approach

- LeNet uses knowledge about the invariances to design:
  - the local connectivity
  - the weight-sharing
  - the pooling
- This achieves about 80 errors
  - This can be reduced to about 40 errors by using many different transformations of the input and other tricks (Ranzato 2008)
- Ciresan *et. al.* (2010) inject knowledge of invariances by creating a huge amount of carefully designed extra training data:
  - For each training image, they produce many new training examples by applying many different transformations
  - They can then train a large, deep, dumb net on a GPU without much overfitting
- They achieve about 35 errors

Slide credit: G. Hinton

The errors made by the *Ciresan et. al.* net



The top printed digit is the right answer. The bottom two printed digits are the network's best two guesses.

The right answer is almost always in the top 2 guesses.

With model averaging they can now get about 25 errors.

Slide credit: G. Hinton

Common Patterns in CNNs

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

- Prefer a stack of small filter CONV to one large receptive field CONV layer
- Don’t reinvent architectures – use whatever works best on ImageNet
- The conv layers should be using small filters (e.g. 3x3 or at most 5x5), using a stride of S=1, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input

Common Patterns in CNNs

- The pool layers are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. F=2), and with a stride of 2 (i.e. S=2)
- The CONV layers preserve the spatial size of their input, while the POOL layers alone are in charge of down-sampling the volumes spatially

Recommended Reading

- CNNs in PyTorch –
  - https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/
  - https://towardsdatascience.com/pytorch-basics-how-to-train-your-neural-net-intro-to-cnn-26a14c2ea29
  - https://algorithmia.com/blog/convolutional-neural-nets-in-pytorch
  - https://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-in-pytorch/
  - https://deeplizard.com/learn/video/MasG7tZj-hw
  - https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/

Home Experiments

- Build a CNN in PyTorch, using 3 conv layers and 2 FF layers, to classify MNIST
- Write a PyTorch layer for Conv2D using your own implementation and compare it to the built-in PyTorch layer above