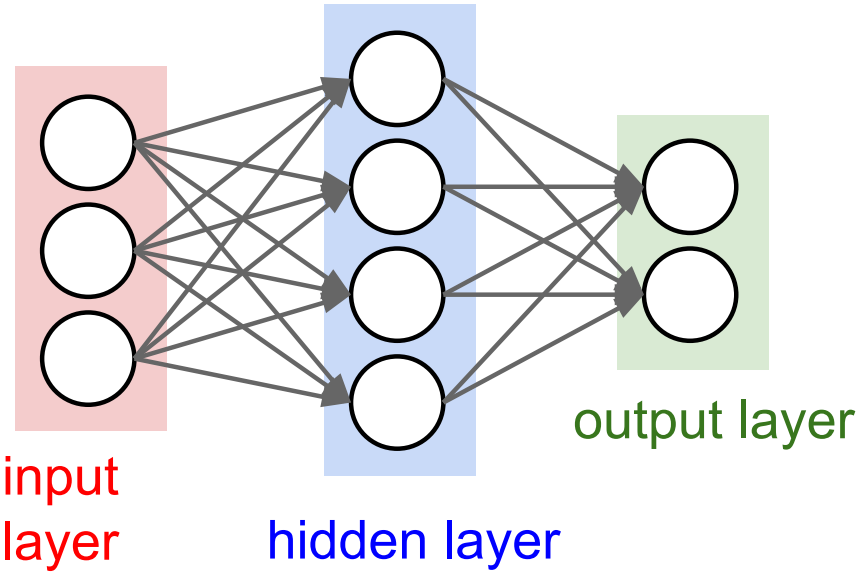
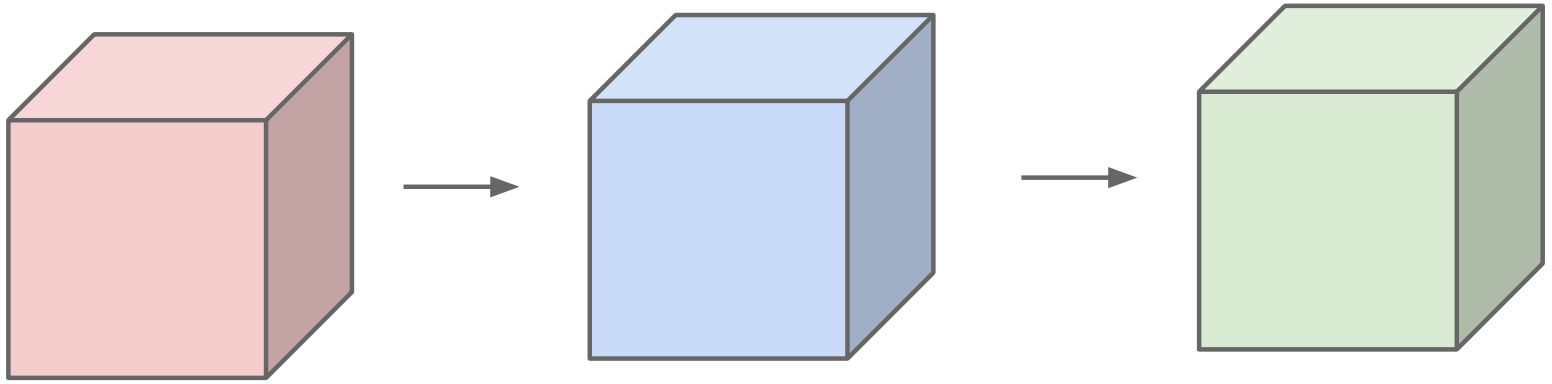

CONVNETS
HOW DO THEY WORK

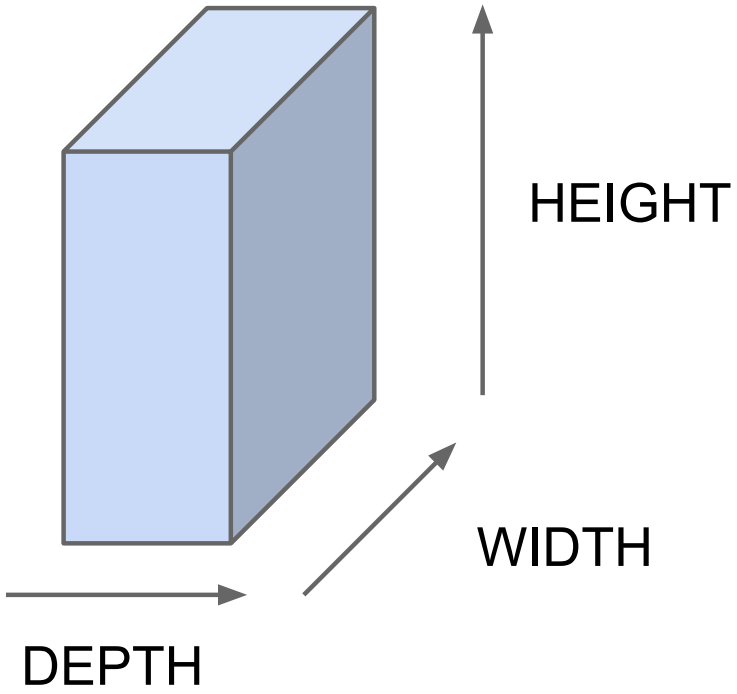
before:



now:



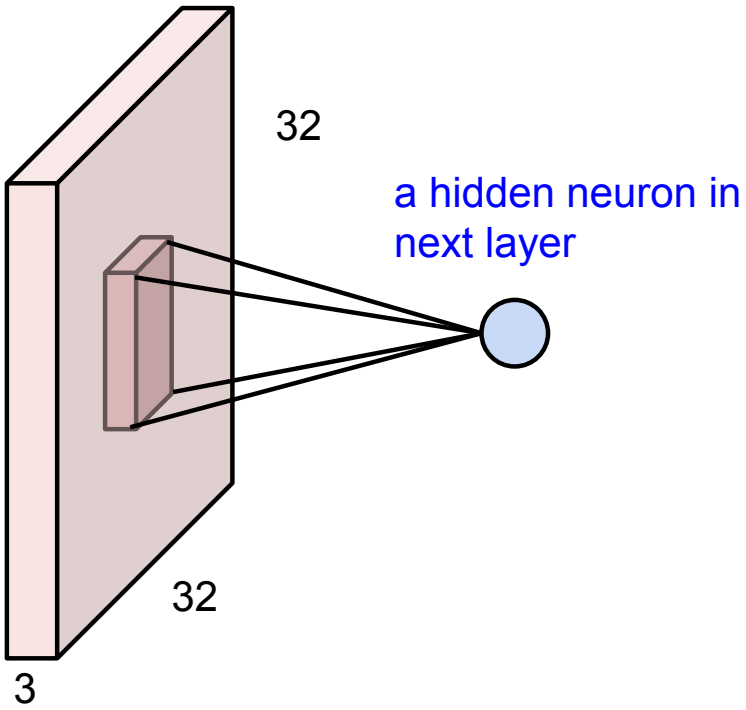
All Neural Net activations arranged in **3 dimensions:**

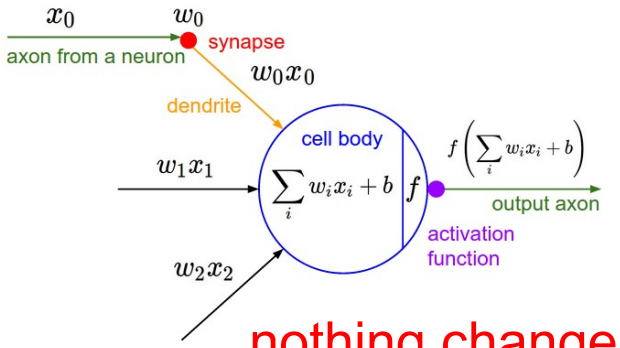


For example, a CIFAR-10 image is a 32x32x3 volume
32 width, 32 height, 3 depth (RGB channels)

Convolutional Neural Networks are just Neural Networks BUT:

1. **Local connectivity**

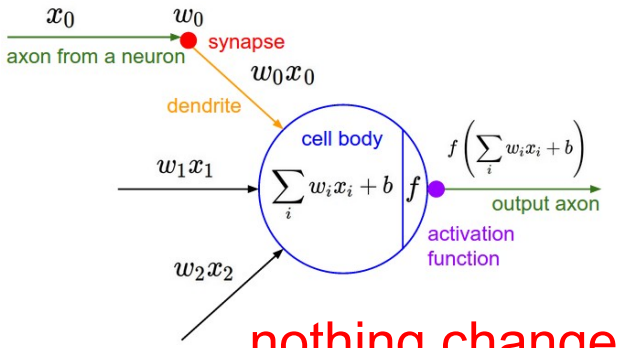




nothing changes

Convolutional Neural Networks are just Neural Networks BUT:

1. Local connectivity



nothing changes

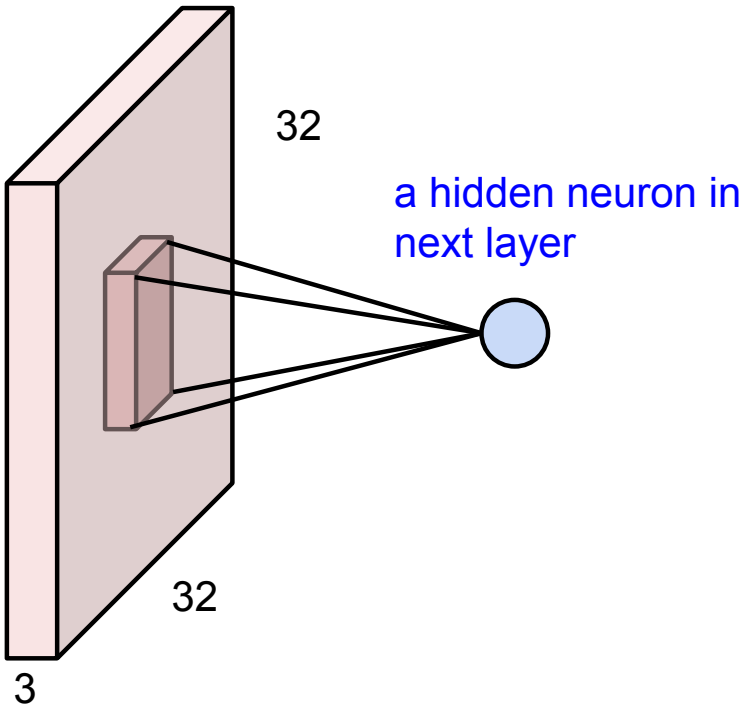


image: 32x32x3 volume
before: full connectivity: 32x32x3 weights

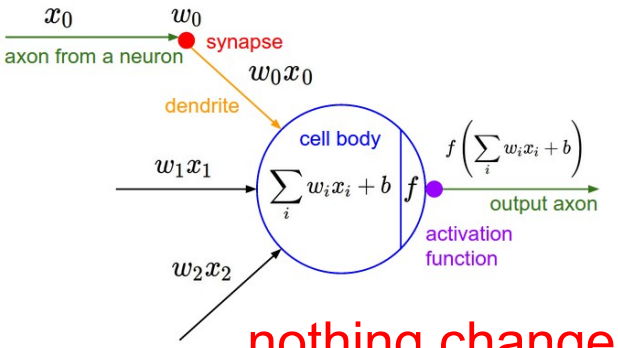
now: one neuron will connect to, e.g. 5x5x3 chunk and only have 5x5x3 weights.

note that connectivity is:

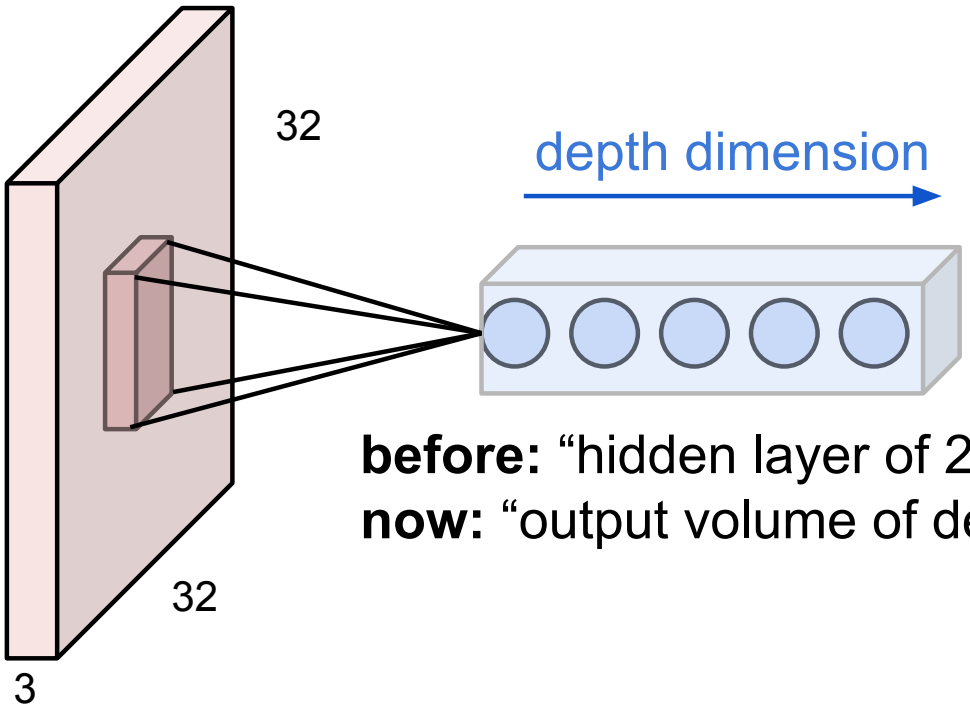
- local in space (5x5 inside 32x32)
- but full in depth (all 3 depth channels)

Convolutional Neural Networks are just Neural Networks BUT:

1. **Local connectivity**



nothing changes

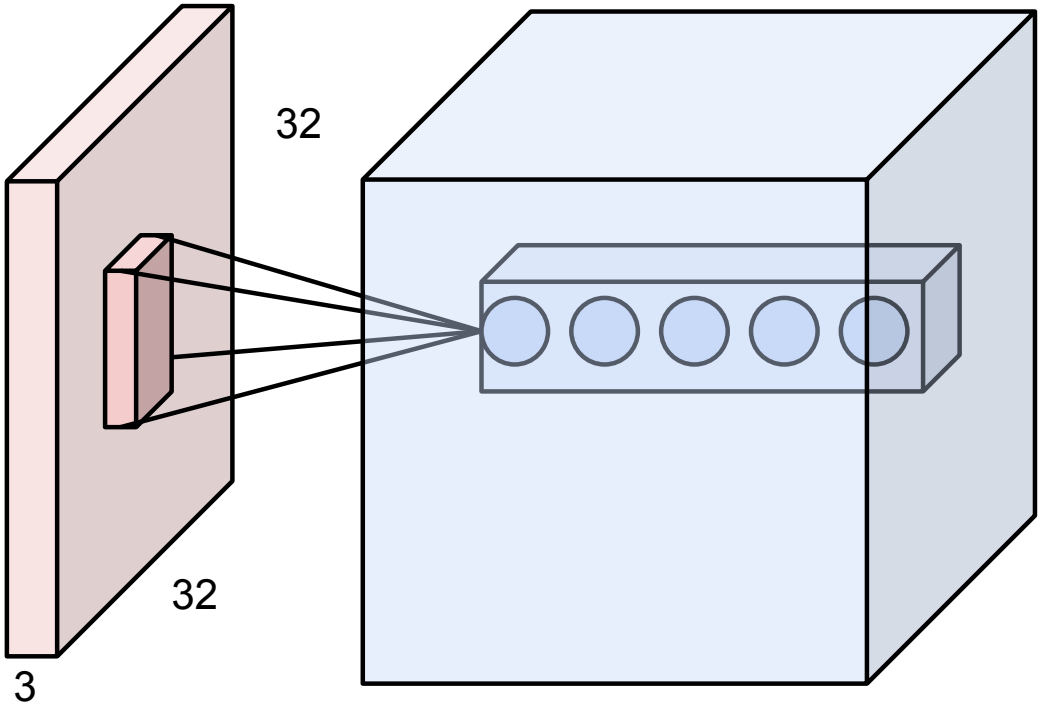


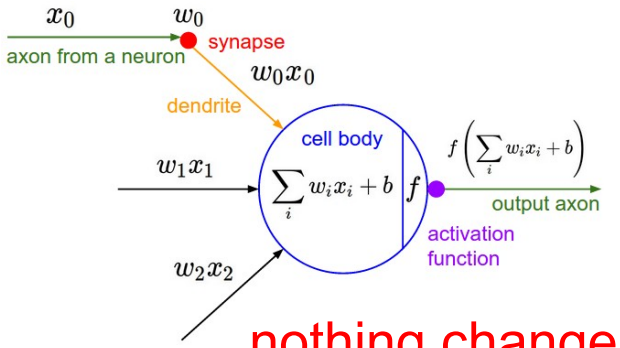
before: “hidden layer of 200 neurons”
now: “output volume of depth 200”

Multiple neurons all looking at the same region of the input volume, stacked along depth.

Convolutional Neural Networks are just Neural Networks BUT:

1. **Local connectivity**

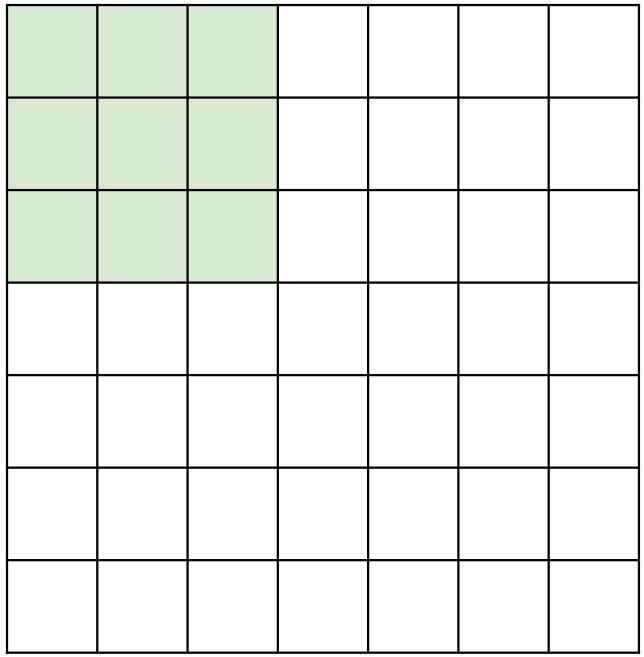




nothing changes

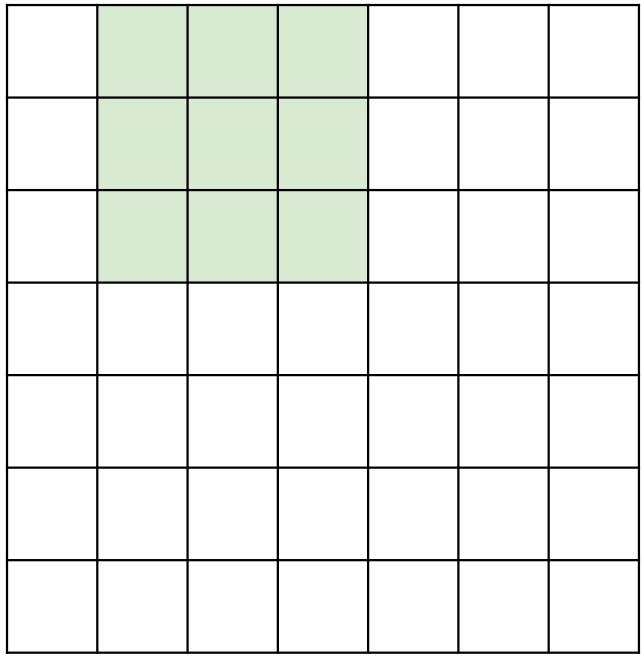
These form a single [1 x 1 x depth] "depth column" in the output volume

Replicate this column of hidden neurons across space, with some **stride.**



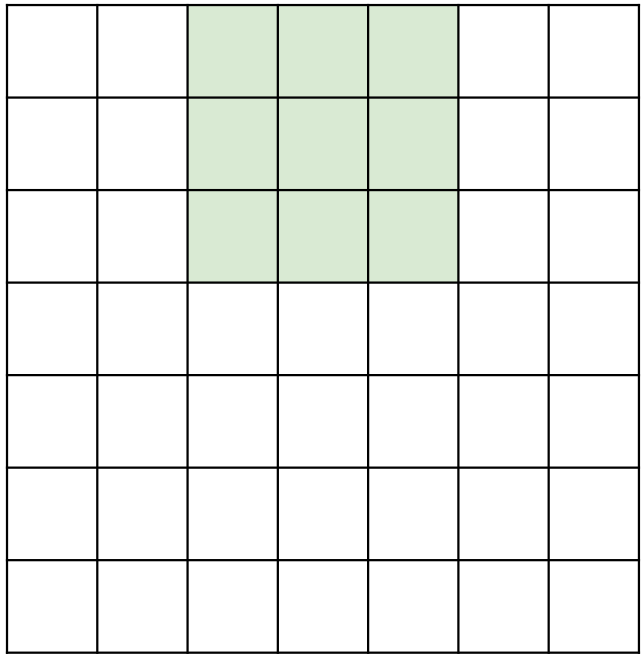
7x7 input
assume 3x3 connectivity, stride 1

Replicate this column of hidden neurons across space, with some **stride.**



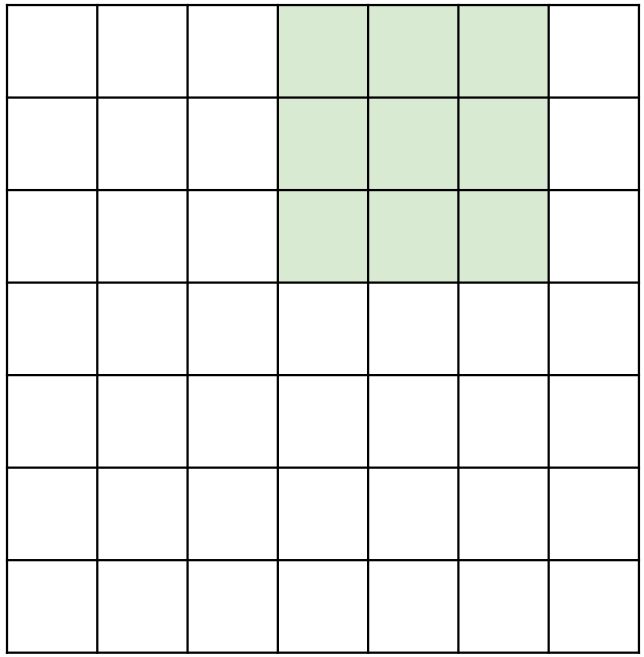
7x7 input
assume 3x3 connectivity, stride 1

Replicate this column of hidden neurons across space, with some **stride.**



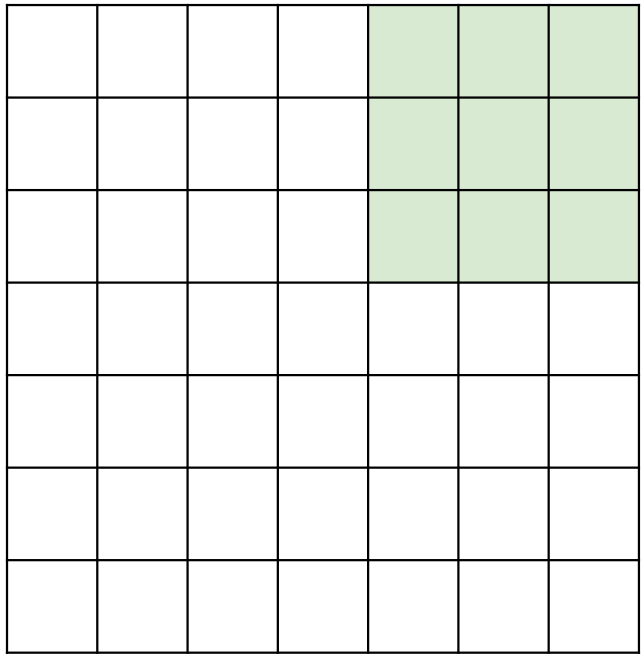
7x7 input
assume 3x3 connectivity, stride 1

Replicate this column of hidden neurons across space, with some **stride.**



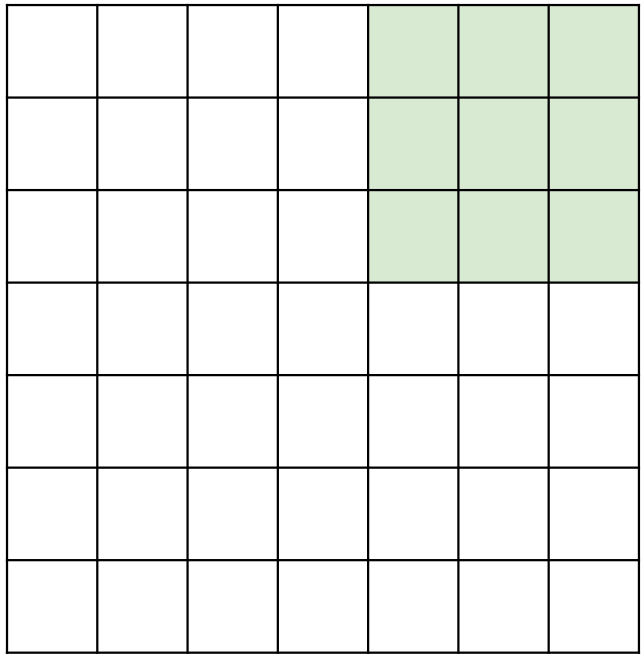
7x7 input
assume 3x3 connectivity, stride 1

Replicate this column of hidden neurons across space, with some **stride.**



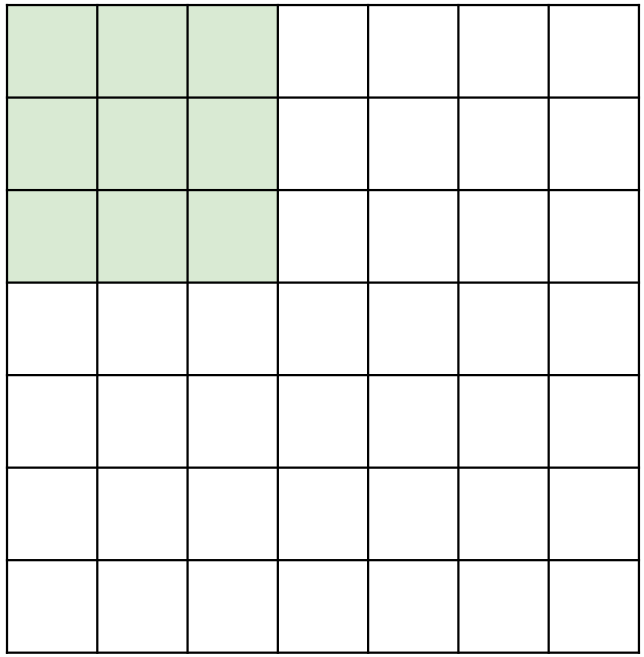
7x7 input
assume 3x3 connectivity, stride 1

Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

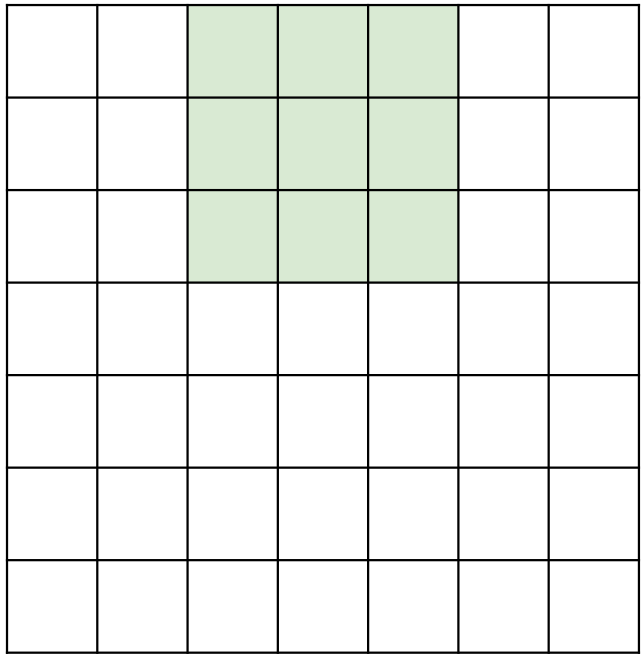
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?

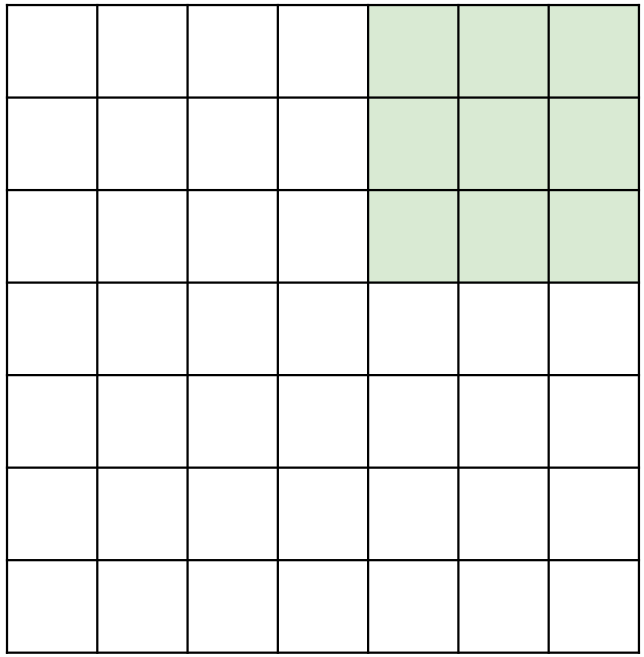
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?

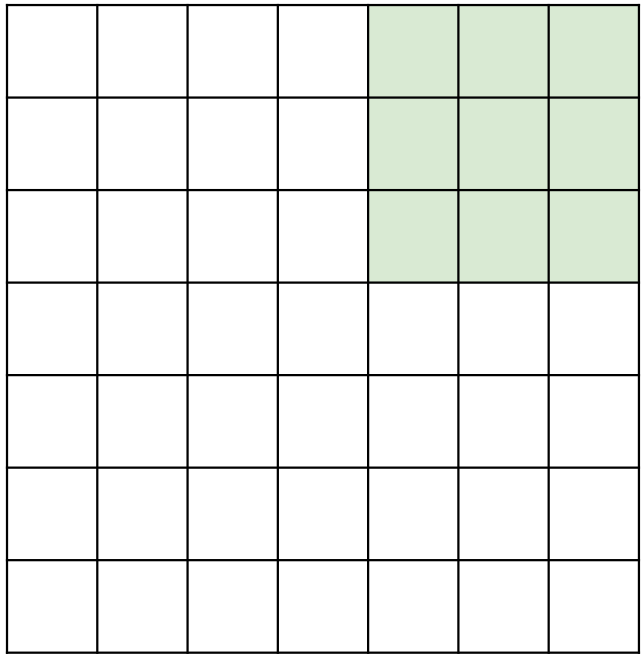
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?

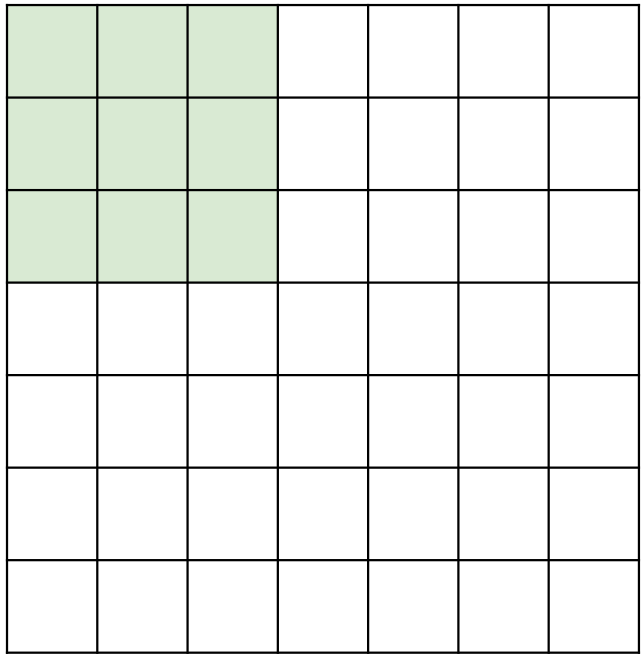
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?
=> **3x3 output**

Replicate this column of hidden neurons across space, with some **stride.**

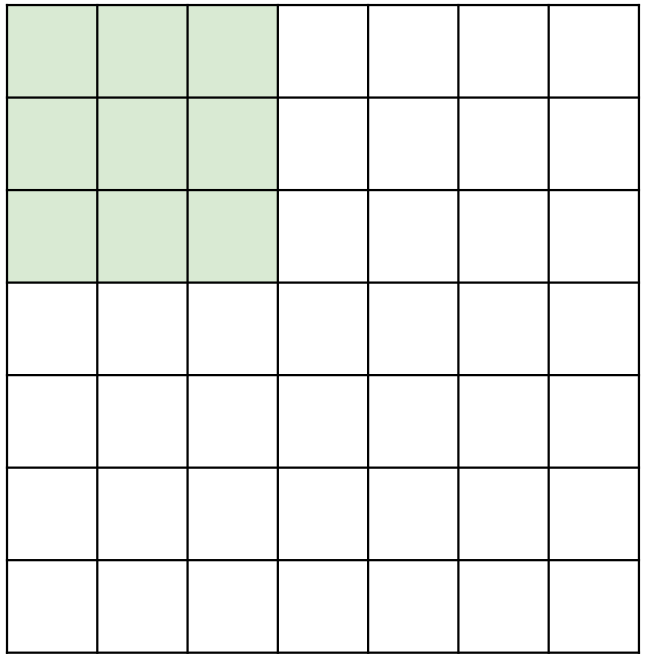


7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?
=> **3x3 output**

what about stride 3?

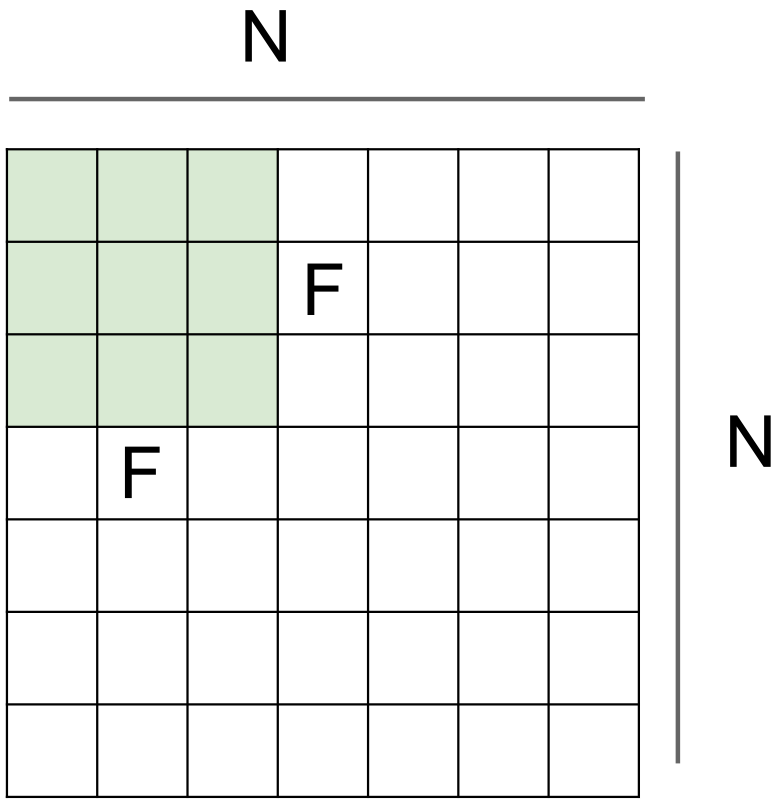
Replicate this column of hidden neurons across space, with some **stride.**



7x7 input
assume 3x3 connectivity, stride 1
=> **5x5 output**

what about stride 2?
=> **3x3 output**

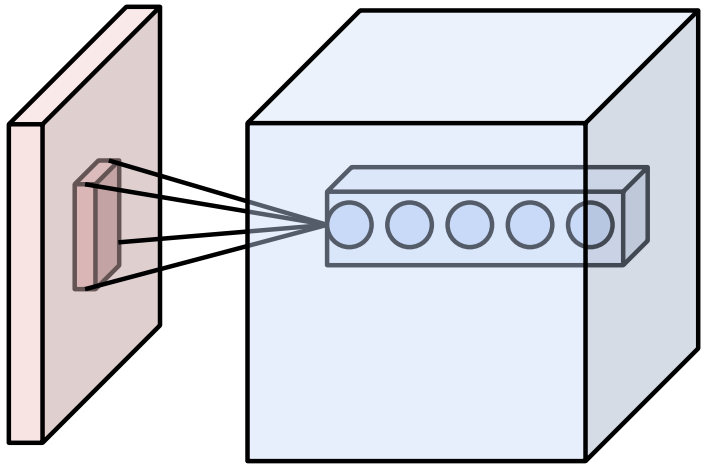
what about stride 3? Cannot.



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 = … :\

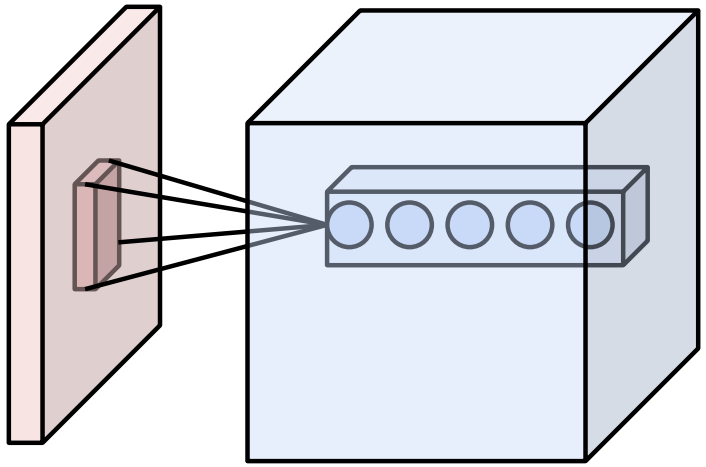
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: ?

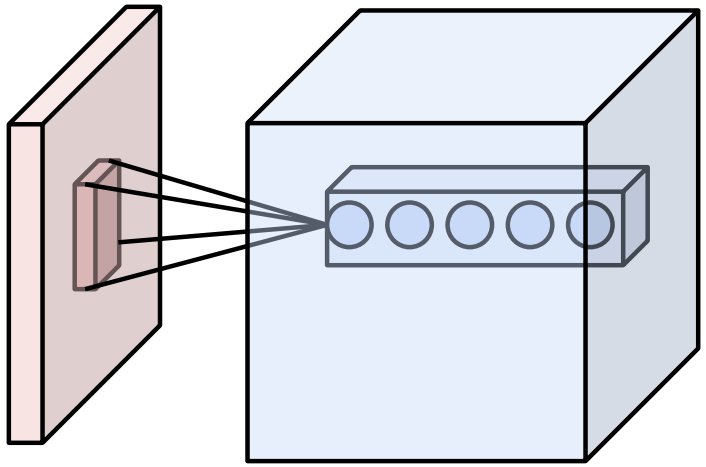
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: (32 - 5) / 1 + 1 = 28, so: **28x28x5**
How many weights for each of the 28x28x5 neurons?

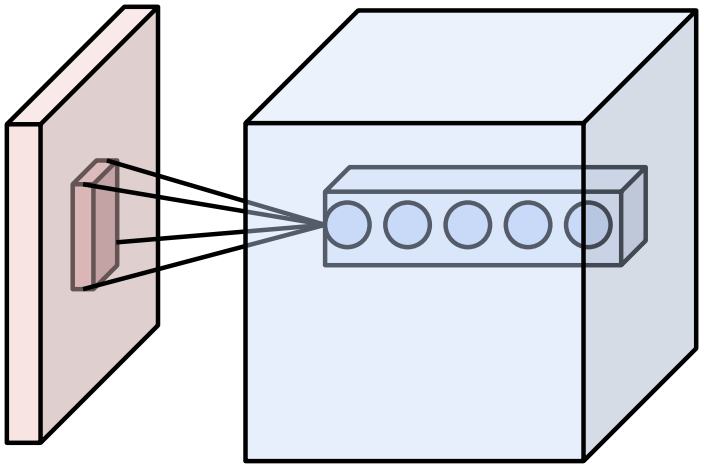
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 1**
Number of neurons: **5**

Output volume: (32 - 5) / 1 + 1 = 28, so: **28x28x5**
How many weights for each of the 28x28x5 neurons? **5x5x3 = 75**

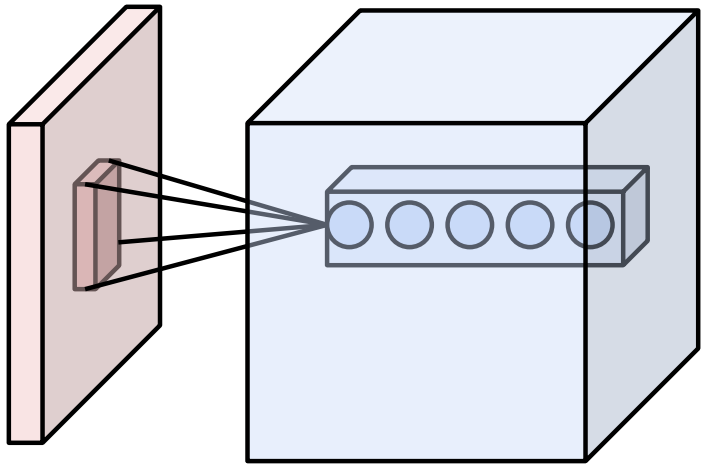
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 2**
Number of neurons: **5**

Output volume: ?

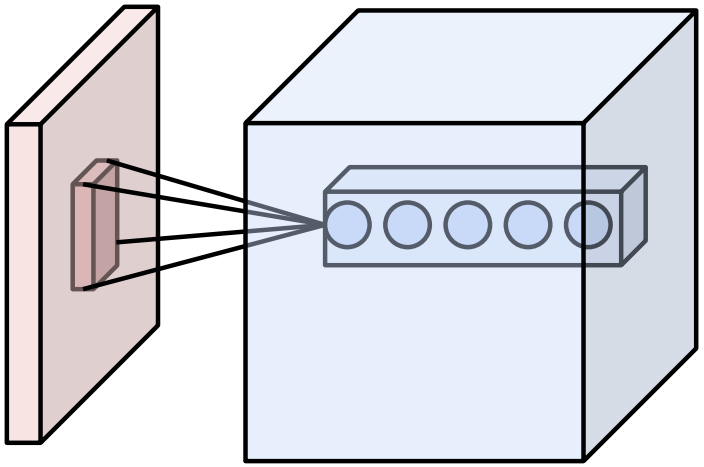
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 2**
Number of neurons: **5**

Output volume: ? Cannot: (32-5)/2 + 1 = 14.5 :\

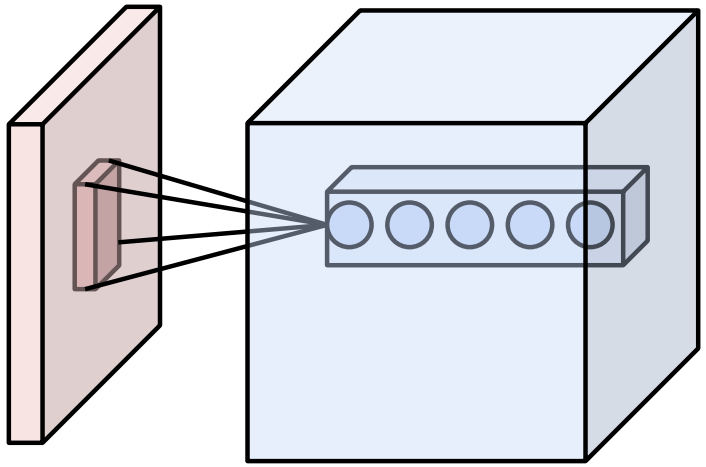
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 3**
Number of neurons: **5**

Output volume: ?

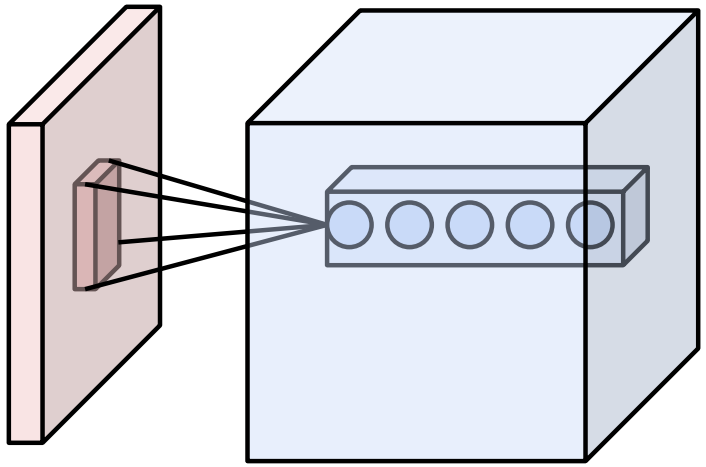
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 3**
Number of neurons: **5**

Output volume: (32 - 5) / 3 + 1 = 10, so: **10x10x5**
How many weights for each of the 10x10x5 neurons?

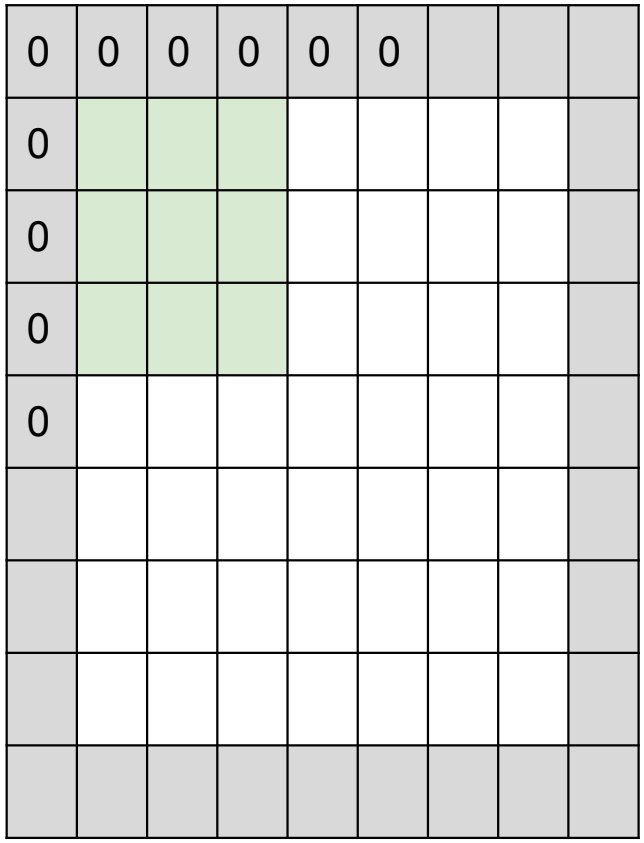
Examples time:



Input volume: **32x32x3**
Receptive fields: **5x5**, **stride 3**
Number of neurons: **5**

Output volume: (32 - 5) / 3 + 1 = 10, so: **10x10x5**
How many weights for each of the 10x10x5 neurons? **5x5x3 = 75** (unchanged)

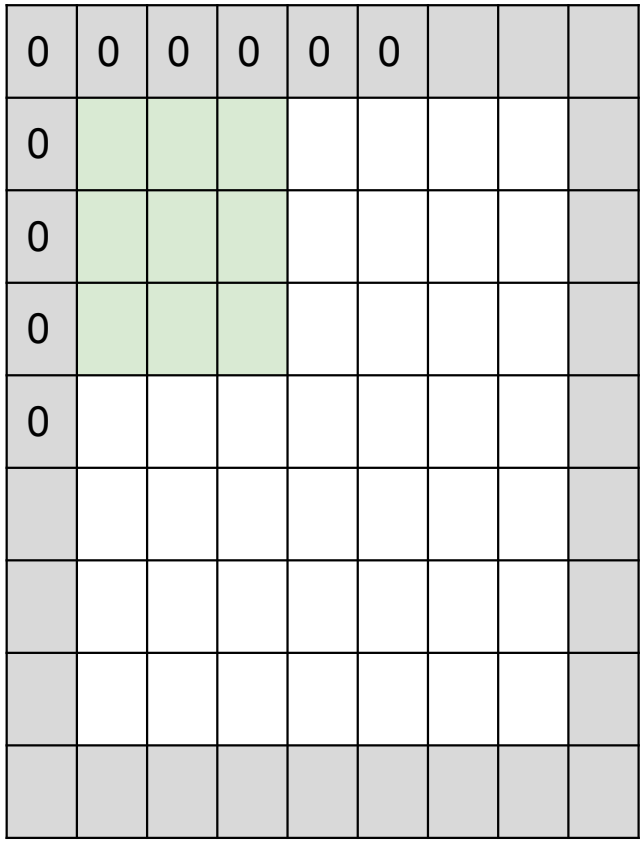
In practice: Common to zero pad the border



(in each channel)

e.g. input 7x7
neuron with receptive field 3x3, stride 1
pad with 1 pixel border => what is the output?

In practice: Common to zero pad the border



(in each channel)

e.g. input 7x7
neuron with receptive field 3x3, stride 1
pad with 1 pixel border => what is the output?

7x7 => preserved size!

in general, common to see stride 1, size F, and zero-padding with (F-1)/2.
(Will preserve input size spatially)

“Same convolution” (preserves size)

Input [9x9]

3x3 neurons, stride 1, pad **1** =>
[9x9]
3x3 neurons, stride 1, pad **1** =>
[9x9]

- No headaches when sizing architectures
- Works well

“Valid convolution” (shrinks size)

Input [9x9]

3x3 neurons, stride 1, pad **0** =>
[7x7]
3x3 neurons, stride 1, pad **0** =>
[5x5]

- **Headaches** with sizing the full architecture
- **Works Worse!** Border information will “wash away”, since those values are only used once in the forward function

Summary:

Input volume of size [W1 x H1 x D1]

using K neurons with receptive fields F x F and applying them at strides of S gives

Output volume: [W2, H2, D2]

W2 = (W1-F)/S+1	H2 = (H1-F)/S+1	D2 = K

There's one more problem…

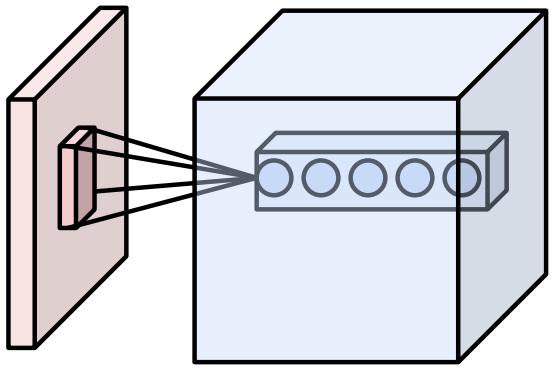
Assume input **[32 x 32 x3]**

30 neurons with receptive fields **5x5**, applied at **stride 1/pad1:**

=> Output volume: [32 x 32 x 30] (32*32*30 = 30720 neurons)

Each neuron has 5*5*3 (=75) weights

=> Number of weights in such layer: 39720 * 75 ~= **3 million :**



There's one more problem…

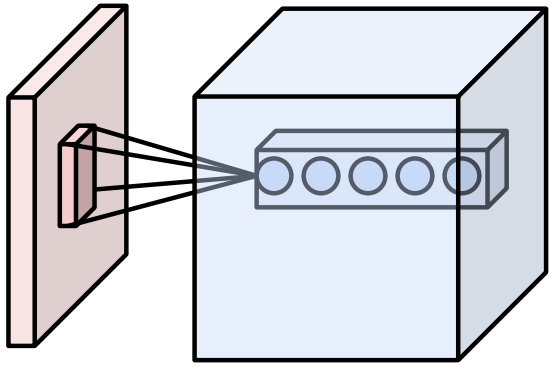
Assume input **[32 x 32 x3]**

30 neurons with receptive fields **5x5**, applied at **stride 1/pad 2:**

=> Output volume: [32 x 32 x 30] (32*32*30 = 30720 neurons)

Each neuron has 5*5*3 (=75) weights

=> Number of weights in such layer: 39720 * 75 ~= **3 million :**



Example trained filters

There's one more problem…

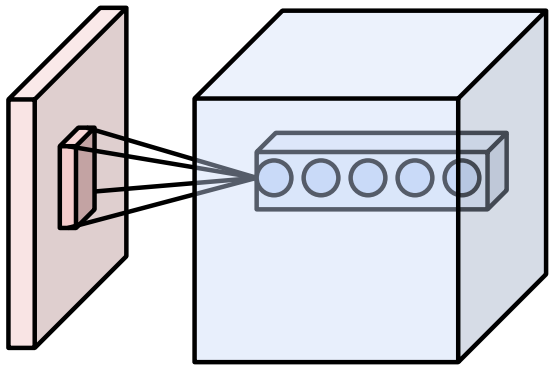
Assume input **[32 x 32 x3]**

30 neurons with receptive fields **5x5**, applied at **stride 1/pad1:**

=> Output volume: [32 x 32 x 30] (32*32*30 = 30720 neurons)

Each neuron has 5*5*3 (=75) weights

=> Number of weights in such layer: 30720 * 75 ~= **3 million :**



Example trained weights

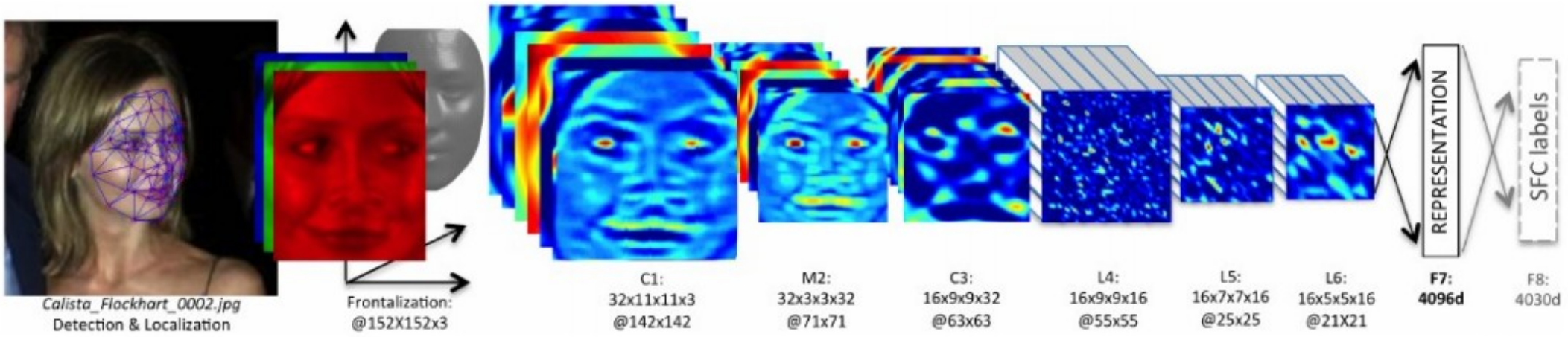
IDEA: lets not learn the same thing across all spatial locations

Our first ConvNet layer had size **[32 x 32 x3]**
If we had **30** neurons with receptive fields **5x5**, **stride 1, pad 1**
Output volume: [32 x 32 x 30] (32*32*30 = 30720 neurons)
Each neuron has 5*5*3 (=75) weights

Before:
#weights in such layer: (32*32*30) * 75 = **3 million :**

Now: (paramater sharing)
#weights in the layer: 30 * 75 = **2250**.

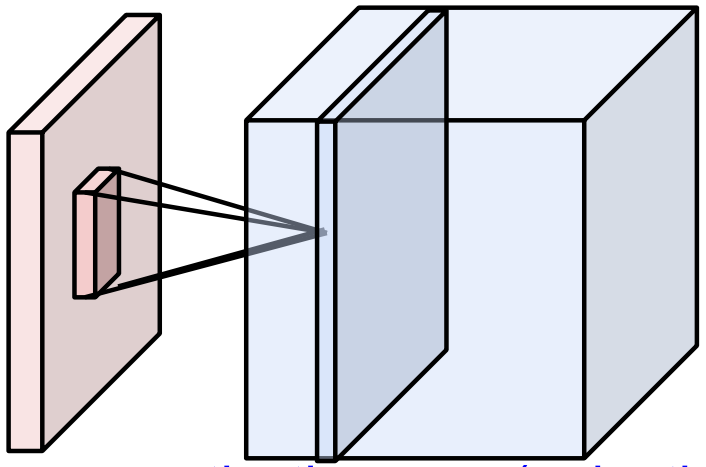
Note: sometimes it's not a good idea to share the parameters



We'd like to be able to learn different things at different spatial positions

These layers are called **Convolutional Layers**

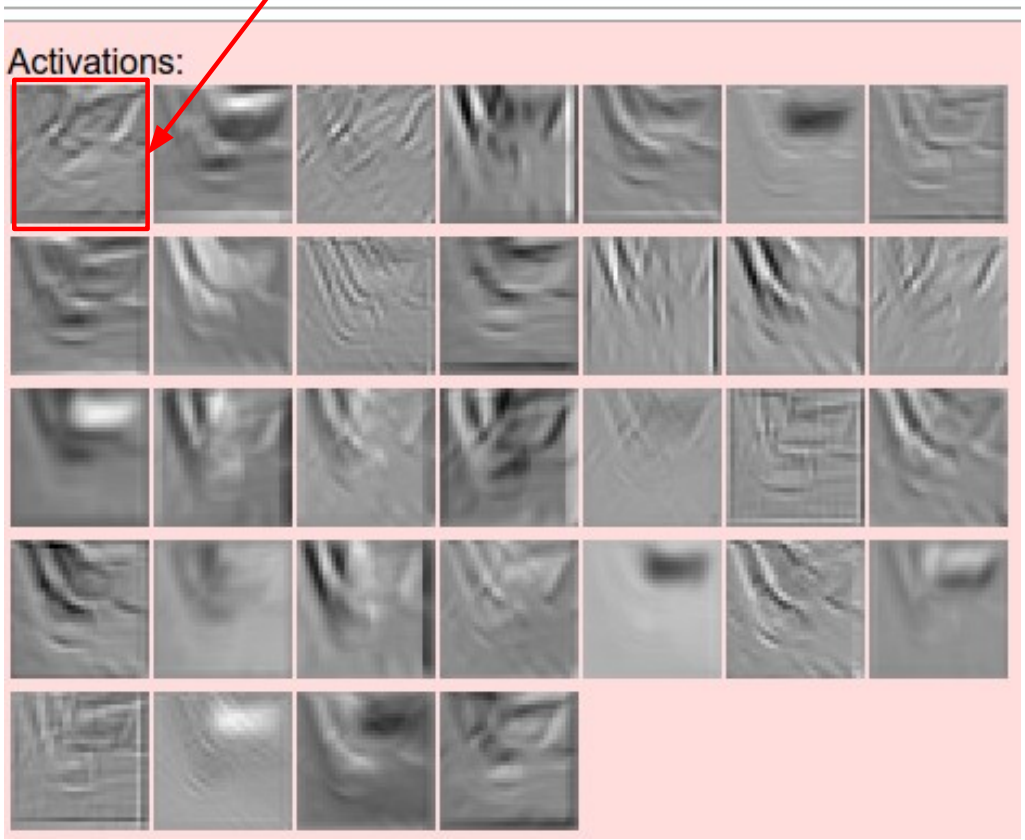
1. Connect neurons only to local receptive fields
2. Use the same neuron weight parameters for neurons in each "depth slice" (i.e. across spatial positions)



one activation map (a depth slice), computed with one set of weights

one filter = one depth slice (or activation map)

5x5 filters



Can call the neurons “filters”

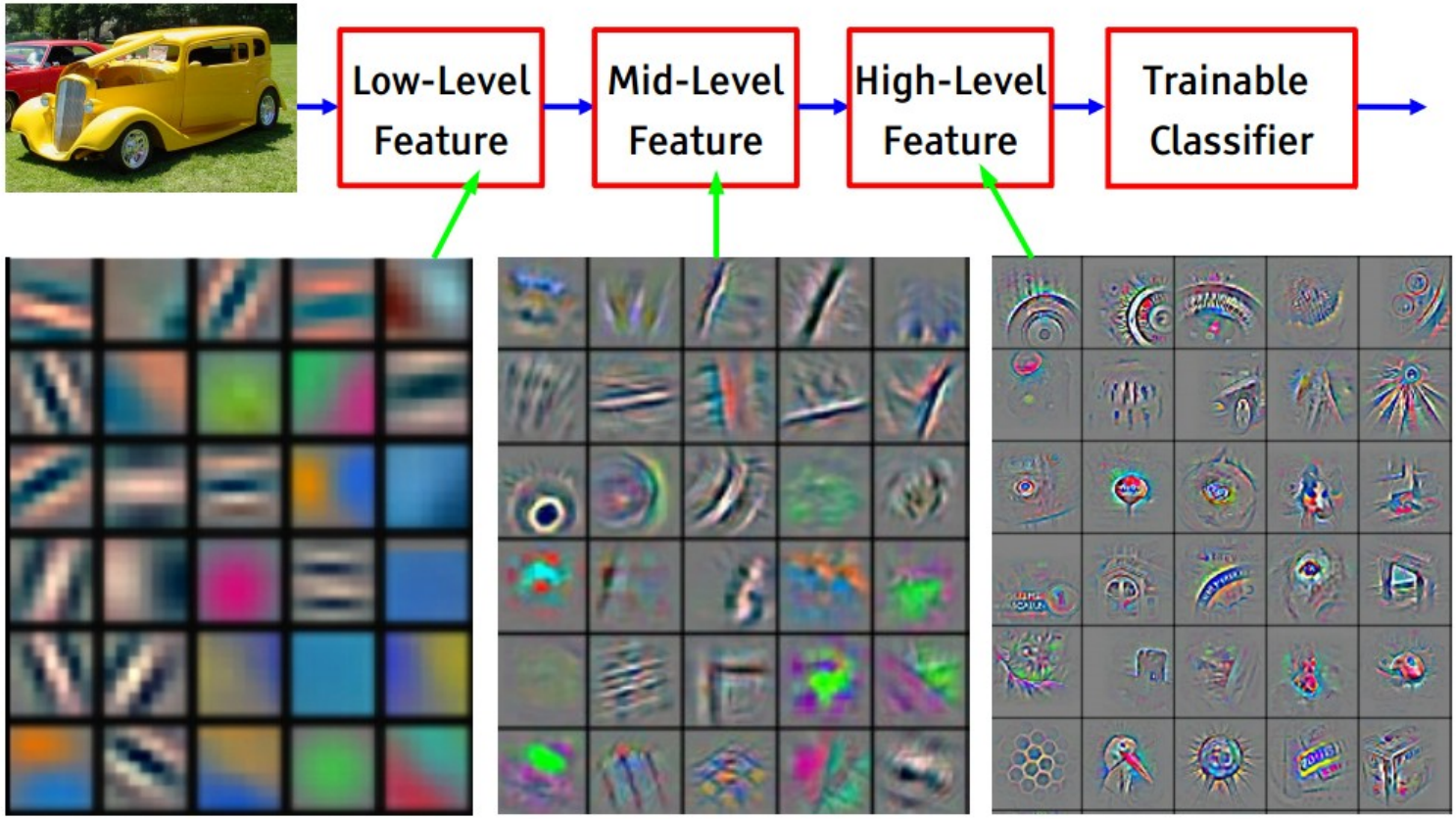
We call the layer convolutional because it is related to convolution of two signals (kind of):

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum o
a filter and the signal (image)
= np.dot(w,x) + b

Fast-forward to today

[From recent Yann LeCun slides]

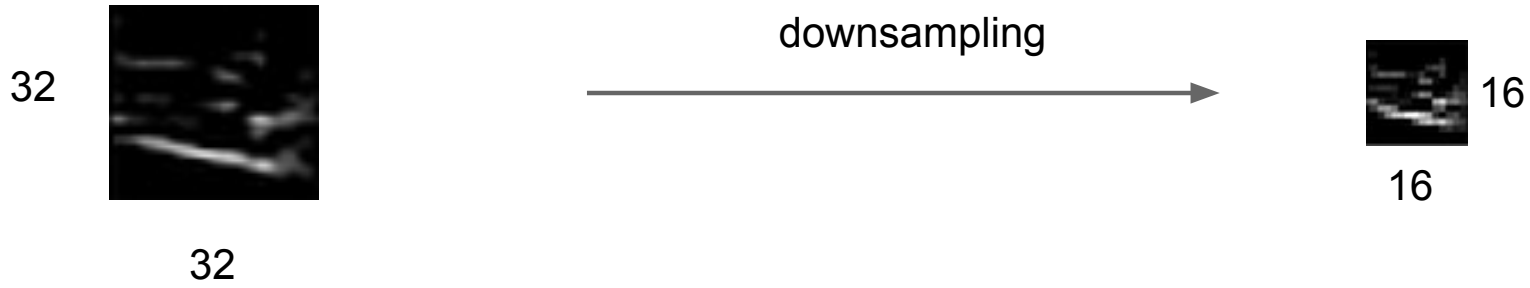


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

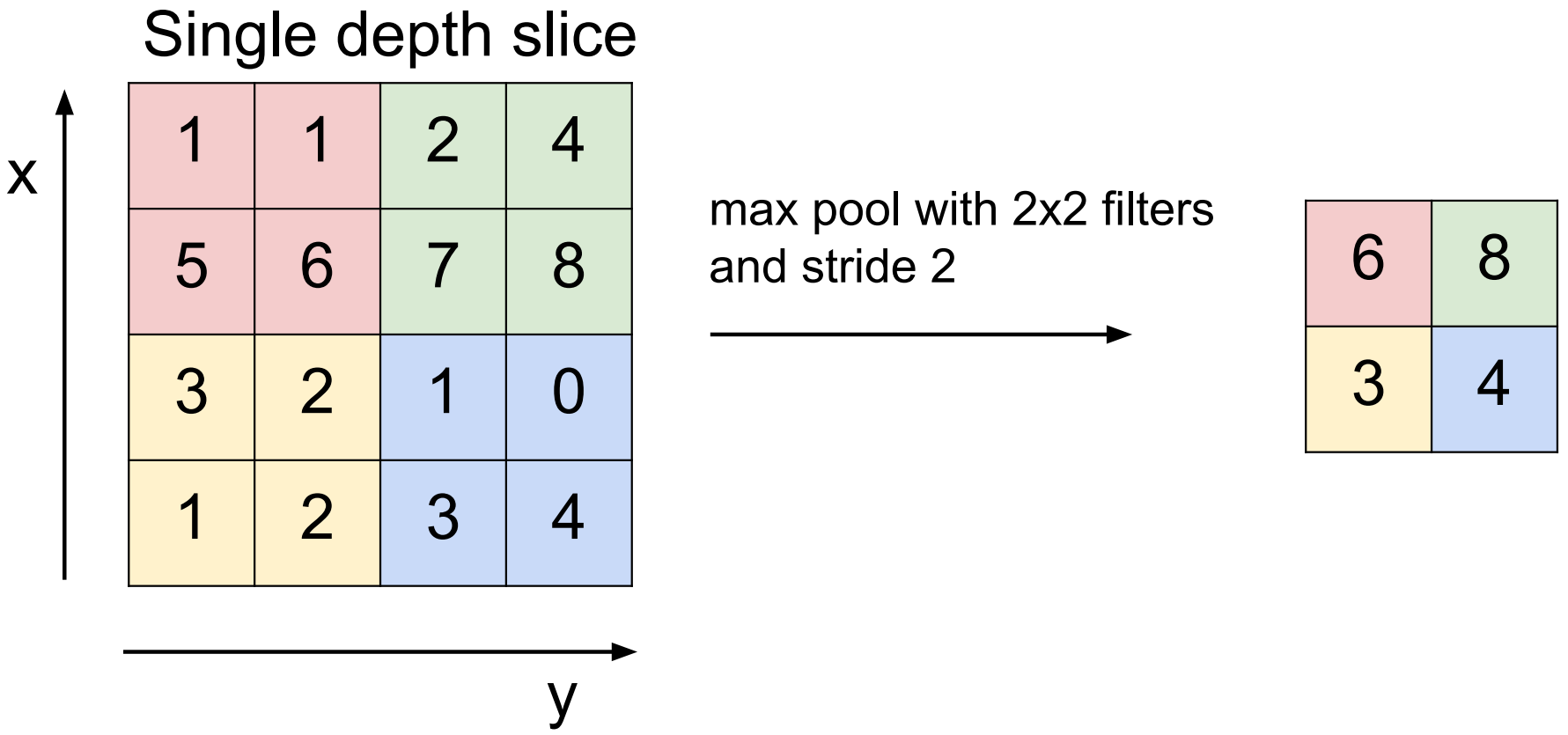

CONV
ReLU
CONV
ReLU
truck
car
airplane
ship
horse

In ConvNet architectures, **Conv** layers are often followed by **Pool** layers

- convenience layer: makes the representations smaller and more manageable without losing too much information. Computes MAX operation (most common)



MAX POOLING

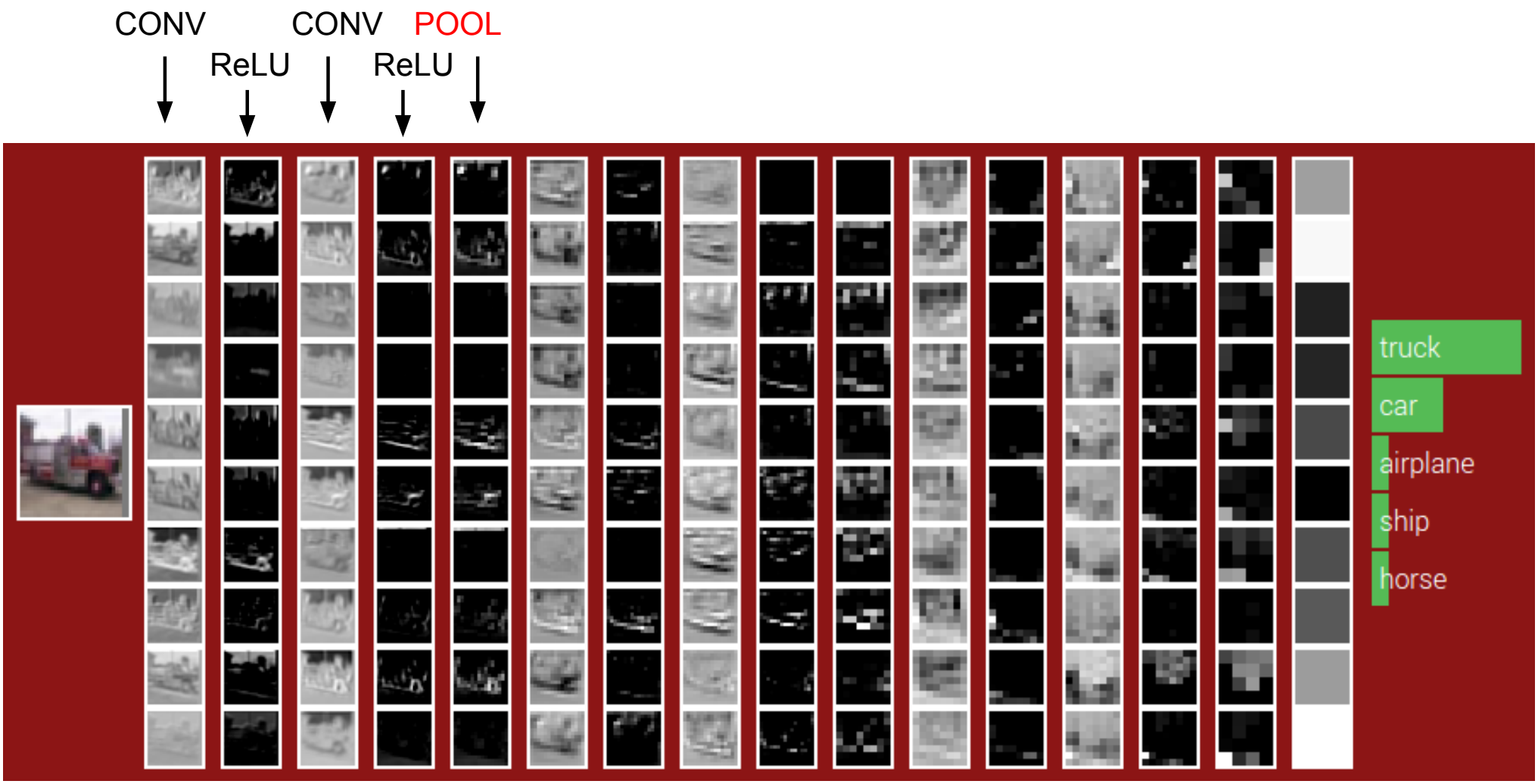

Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

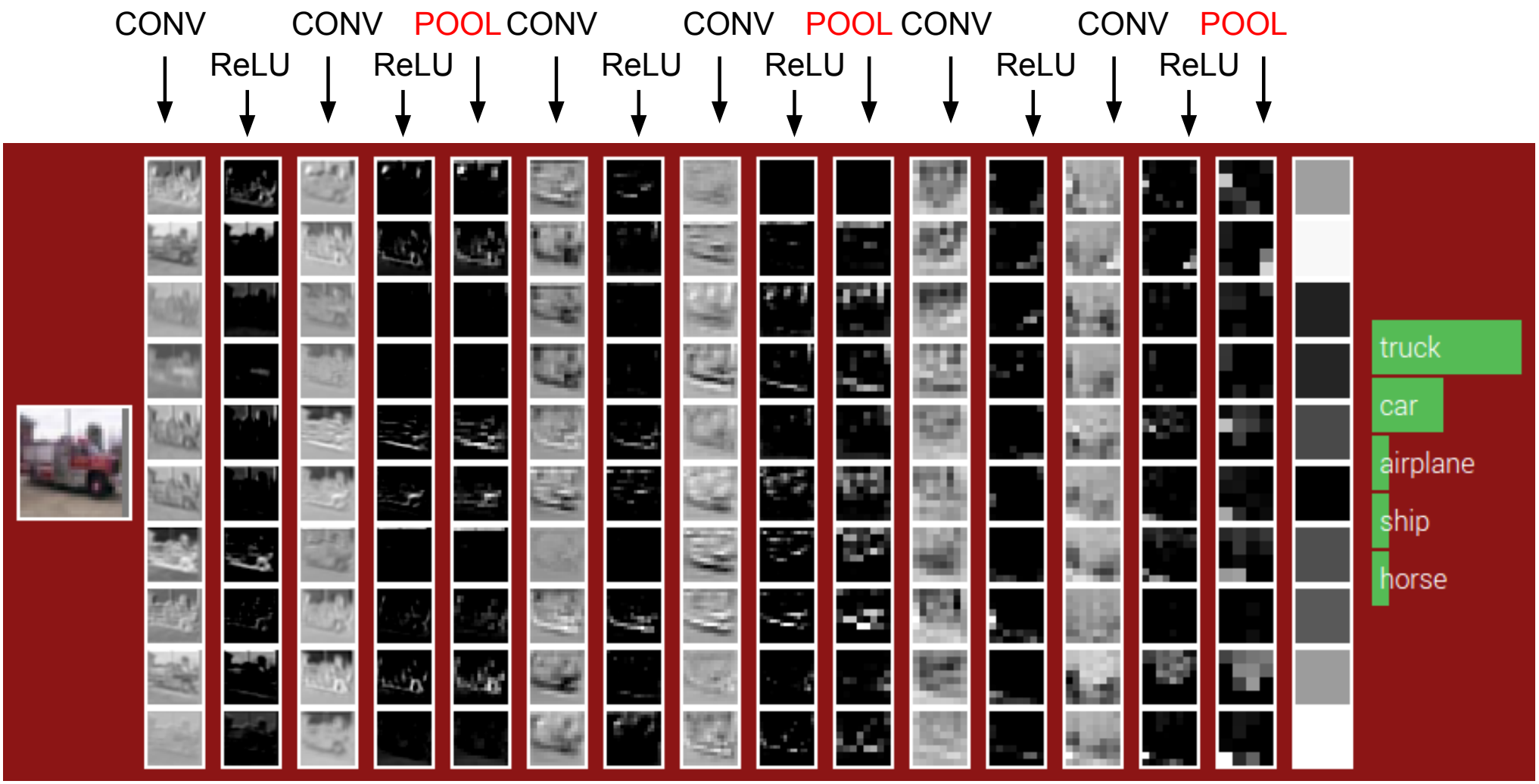
In ConvNet architectures, **Conv** layers are often followed by **Pool** layers

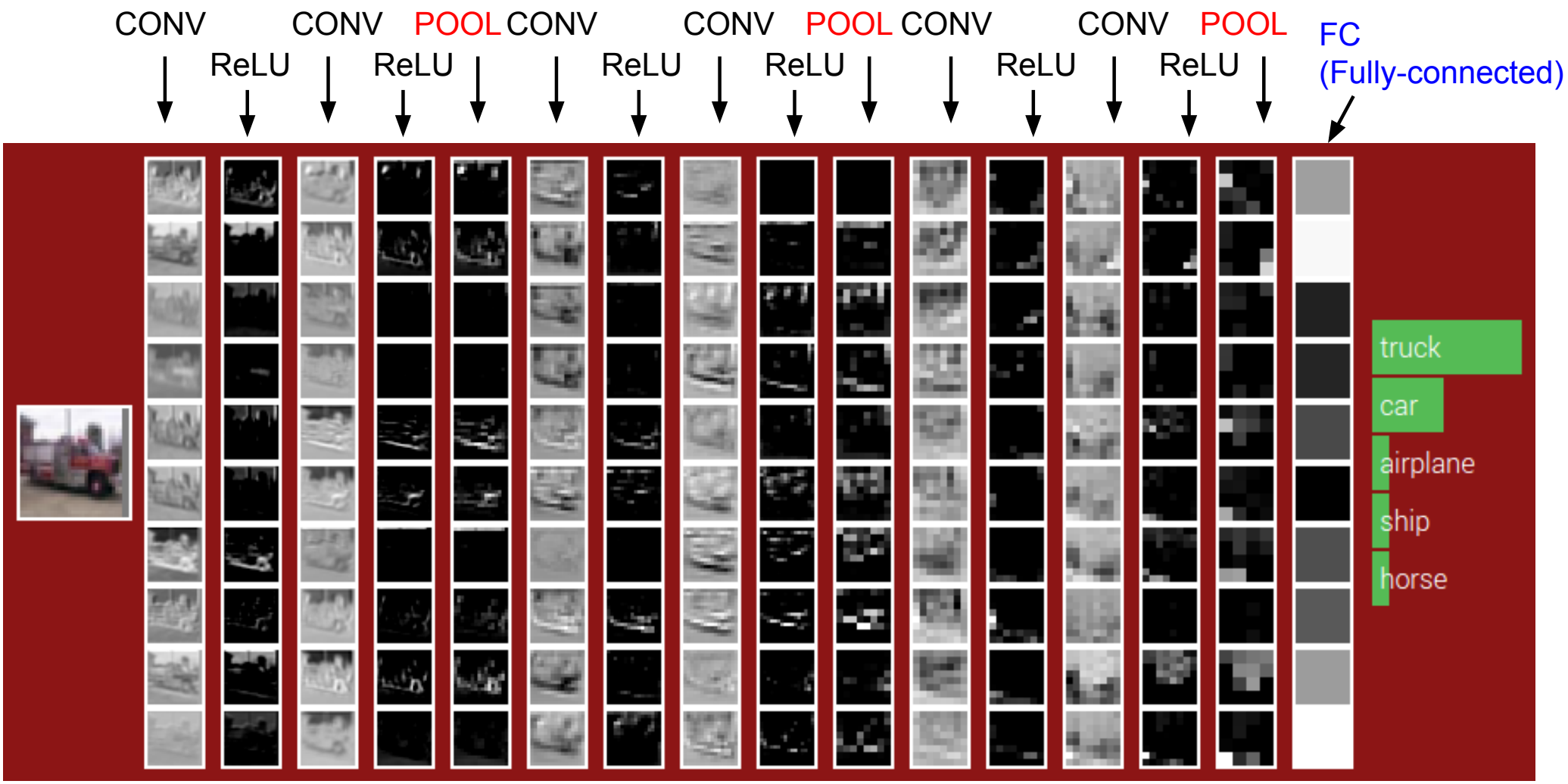
- convenience layer: makes the representations smaller and more manageable without losing too much information. Computes MAX operation (most common)

Input volume of size [W1 x H1 x D1]
Pooling unit receptive fields F x F and applying them at strides of S gives
Output volume: [W2, H2, D1]
W2 = (W1-F)/S+1, H2 = (H1-F)/S+1

Note: pooling happens independently across each slice, preserving number of slices
E.g. a pooling “neuron” of size 2x2 will perform MAX operation over 4 numbers.


CONV
ReLU
CONV
ReLU
POOL
truck
car
airplane
ship
horse


CONV
ReLU
CONV
ReLU
POOL
CONV
ReLU
CONV
ReLU
POOL
CONV
ReLU
CONV
ReLU
POOL
truck
car
airplane
ship
horse


CONV
ReLU
CONV
ReLU
POOL
CONV
ReLU
CONV
ReLU
POOL
CONV
ReLU
CONV
ReLU
POOL
FC
(Fully-connected)
truck
car
airplane
ship
horse

Modern CNNs:

- use **filter sizes of 3x3** (maybe even 2x2 or 1x1!)
- use **pooling sizes of 2x2** (maybe even less - e.g. fractional pooling!)
- **stride 1**
- **very deep**

(if too expensive for time/space, might have to downsample more, or make bigger strides, etc)

Eliminate sizing headaches TIPS/TRICKS

- start with image that has power-of-2 size
- for **conv layers**, use stride 1 filter size 3x3 pad input with a border of zeros (1 spatially)

This makes it so that: [W1,H1,**D1**] -> [W1,H1,**D2**] (i.e. spatial size exactly preserved)

- for **pool layers**, use pool size 2x2 (more = worse)

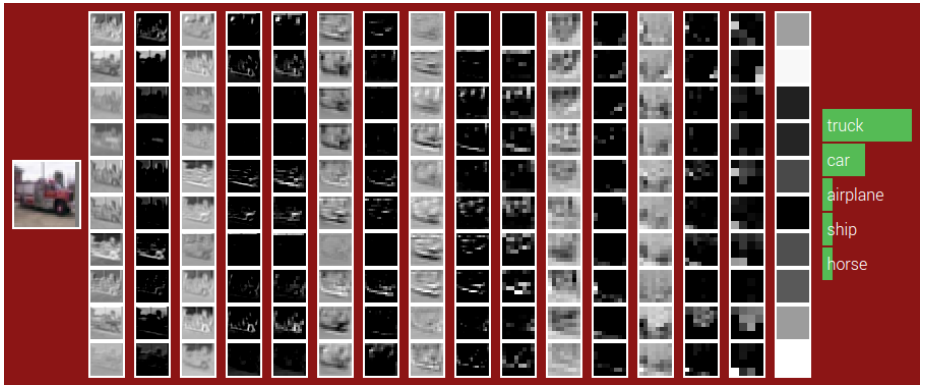
For big images, good heuristic is to violate the previous rules but only for the first layer. E.g. “AlexNet”:

input 227x227x3
first CONV layer: 96 11x11 filters, stride 4
=> (227 - 11)/4 + 1 = 55
=> output volume 55 x 55 x 96

...and from then stride 1 “same“ convolutions (with padding).

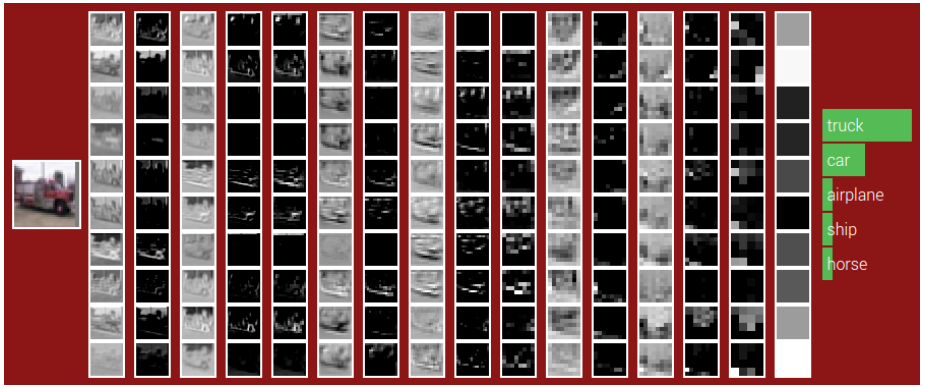
Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives:



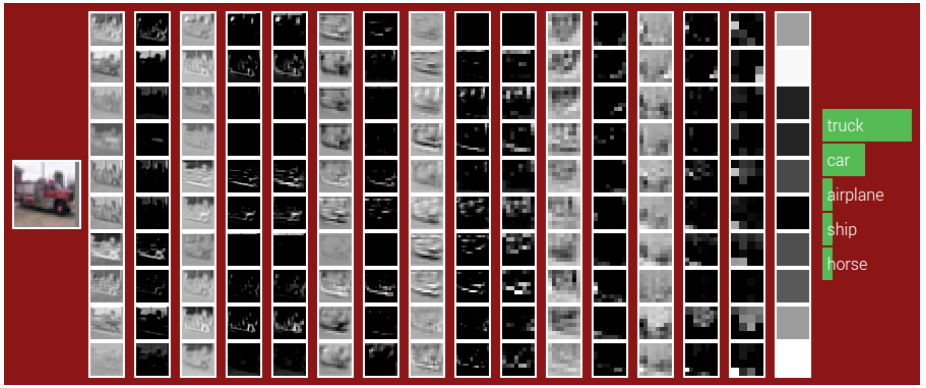
Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters:



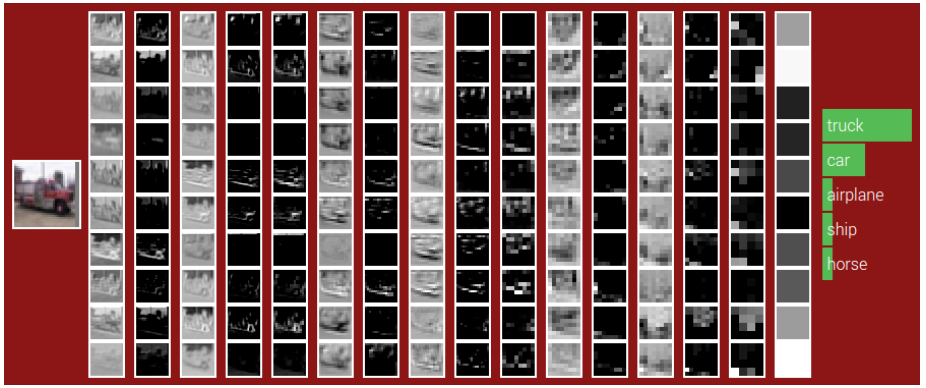
Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*3)*10 + 10 = 280
RELU



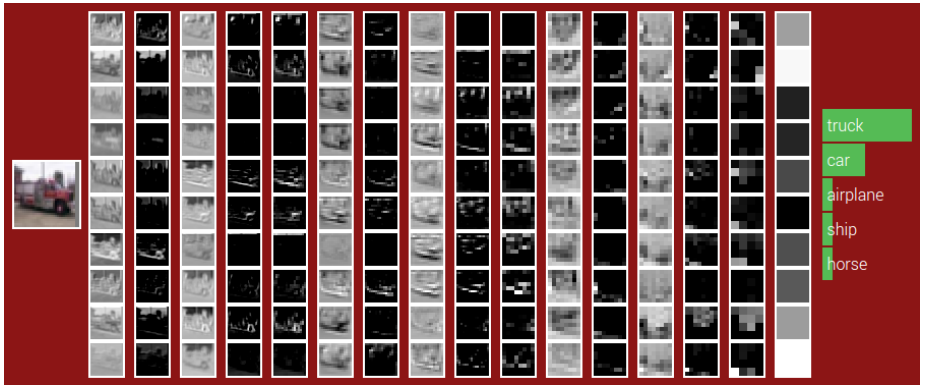
Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*3)*10 + 10 = 280
RELU
CONV with 10 3x3 filters, stride 1, pad 1:
gives:



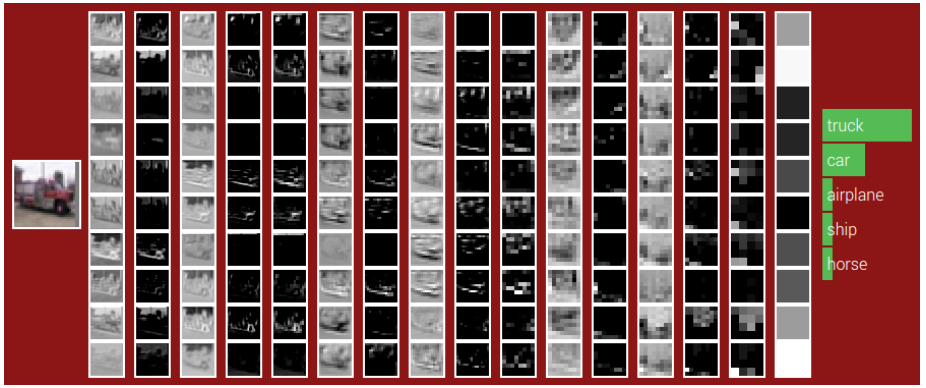
Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*3)*10 + 10 = 280
RELU
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters:



Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*3)*10 + 10 = 280
RELU
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*10)*10 + 10 = 910
RELU



Example:

input: [32x32x3]
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*3)*10 + 10 = 280
RELU
CONV with 10 3x3 filters, stride 1, pad 1:
gives: [32x32x10]
new parameters: (3*3*10)*10 + 10 = 910
RELU
POOL with 2x2 filters, stride 2:
gives: