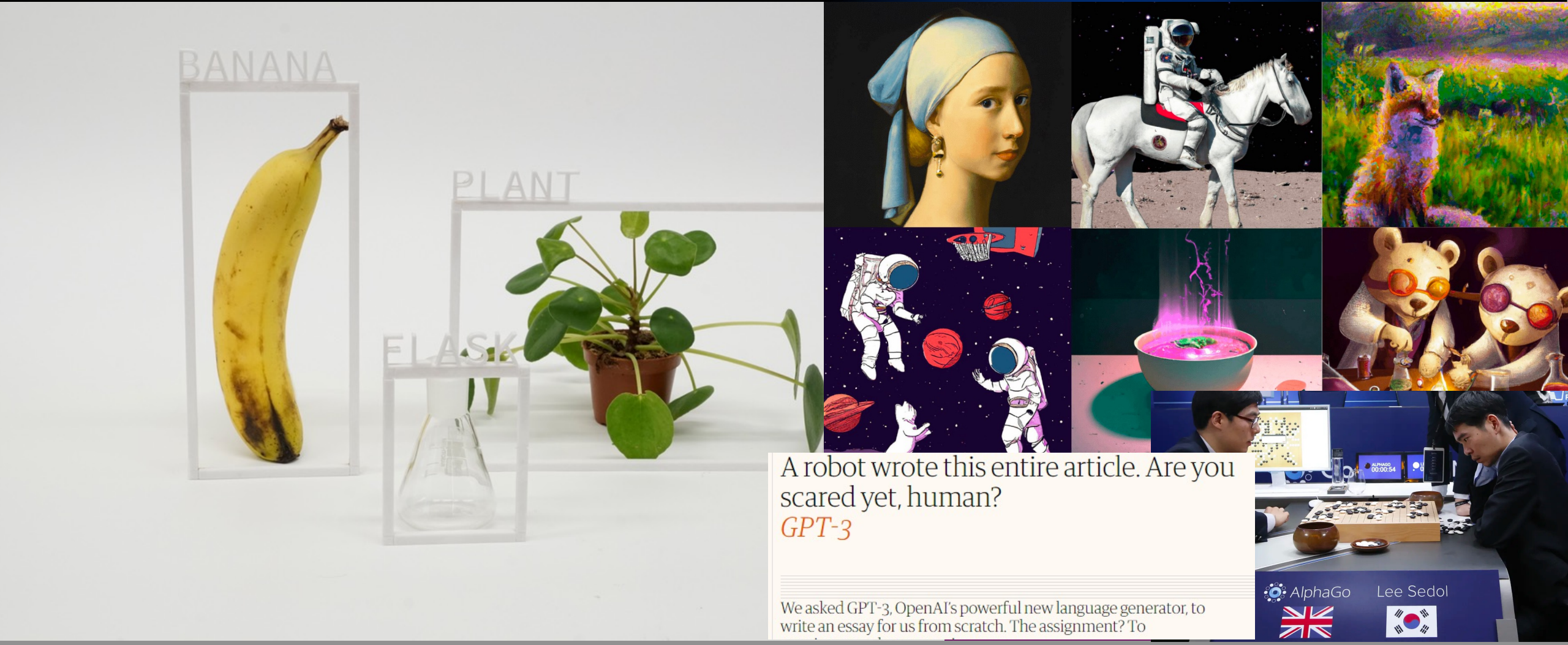


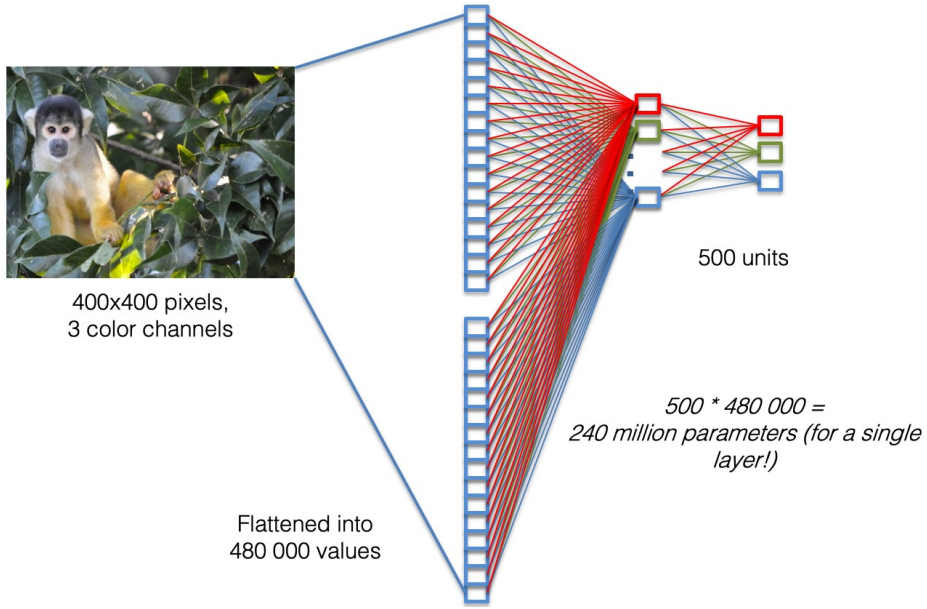
Lecture 5: Convolutional Neural Networks

Deep Learning 1 @ UvA

Yuki M. Asano

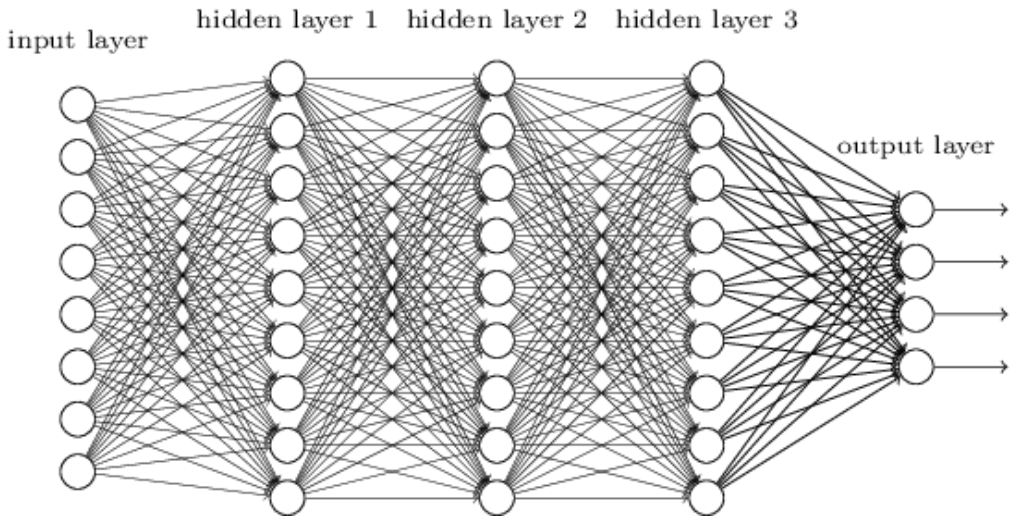
Multi-layer perceptrons (Recap)

- An input image of 400x400 pixels and 3 colour channels (red, green, blue)
- flattened into a vector of 480 000 input values
- fed into an MLP with a single hidden layer with 500 hidden units
- 500*248 000 = 240 million parameters for a single layer



Multi-layer perceptrons (Recap)

- From this fully-connected model, do we really need all the edges?
- Can some of these be "shared" (equal weight)?
- Can prior knowledge ("inductive biases") be incorporated into the design?



Consider an image

- Some patterns are much smaller than the whole image
- Can represent a small region with fewer parameters
- What about training a lot of such "small" detectors and each detector must be able to "move around"? (C.f. Barlow 1961)

The convolution operation

- We have a signal $x(t)$ and a weighting function $w(a)$
- We can generate a new function $s(t)$ by the following equation:

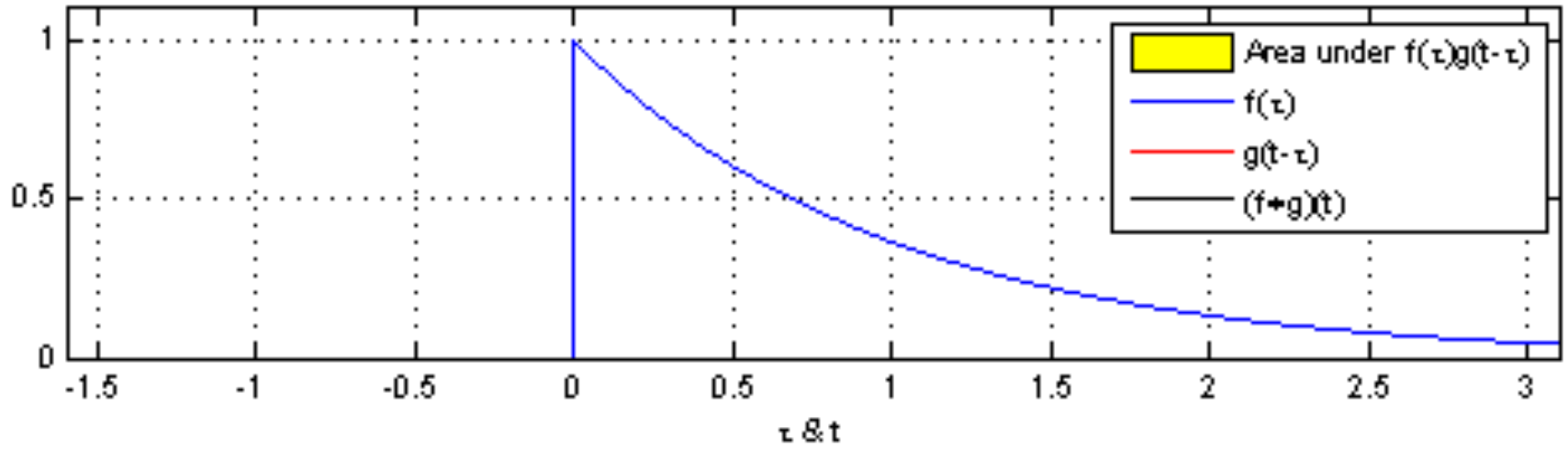
$$s(t) = \int x(a)w(t-a)da$$

- We refer to $w(a)$ *as a filter or a kernel*. This operation is called convolution.
- The convolution operation is typically denoted with asterisk:

$$s(t) = (x * w)(t)$$

The convolution operation

- The convolution $(f * g)(t)$ of two functions $f(t)$ and $g(t)$ *computes the overlap in area*



Convolution for 2D images

- For a two-dimensional image I as our input, we want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

- Convolution is commutative, and we can equivalently write:

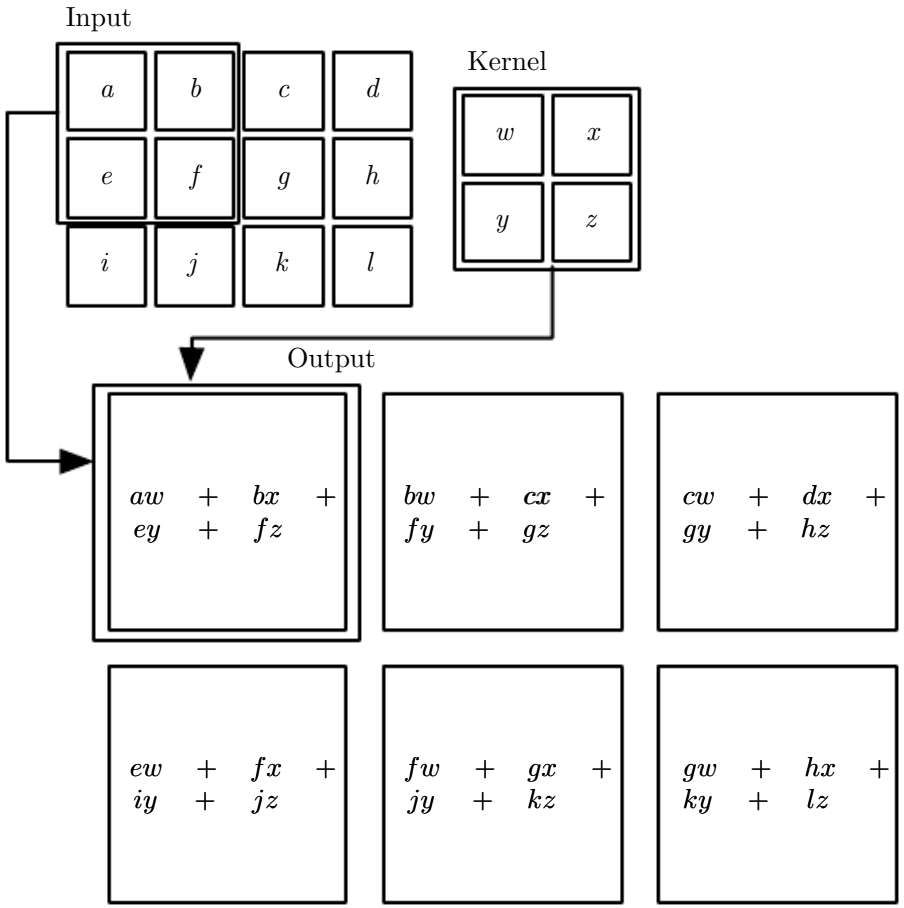
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

- Neural networks libraries implement cross-correlation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

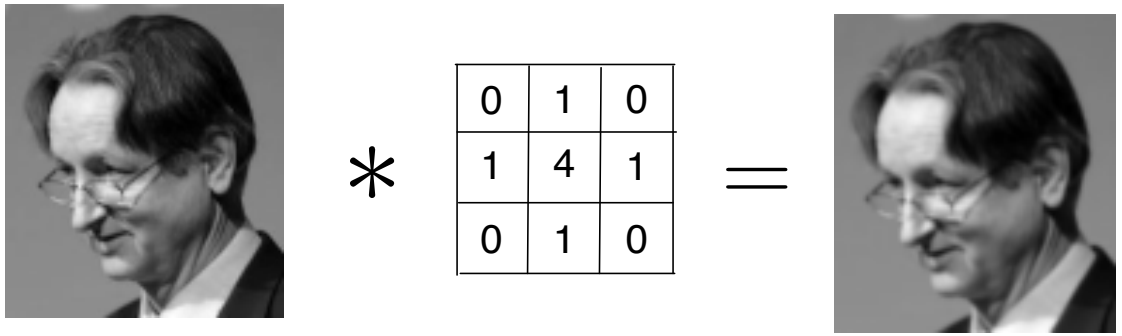
Convolution for 2D images

- An example of 2-D convolution

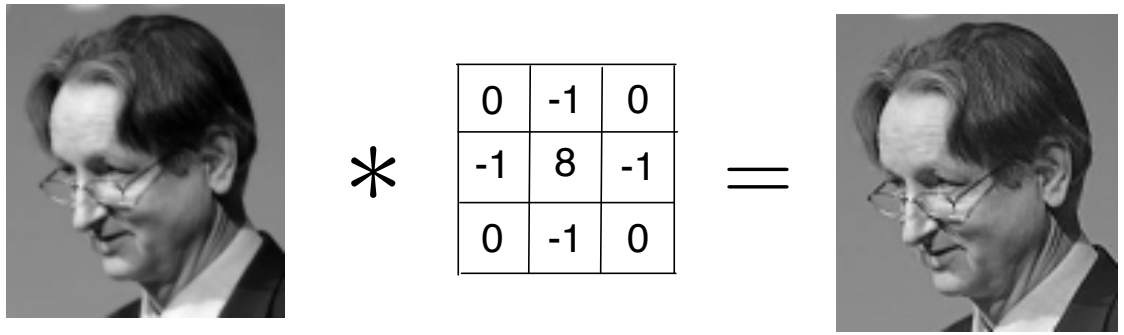


Examples

- We can blur images



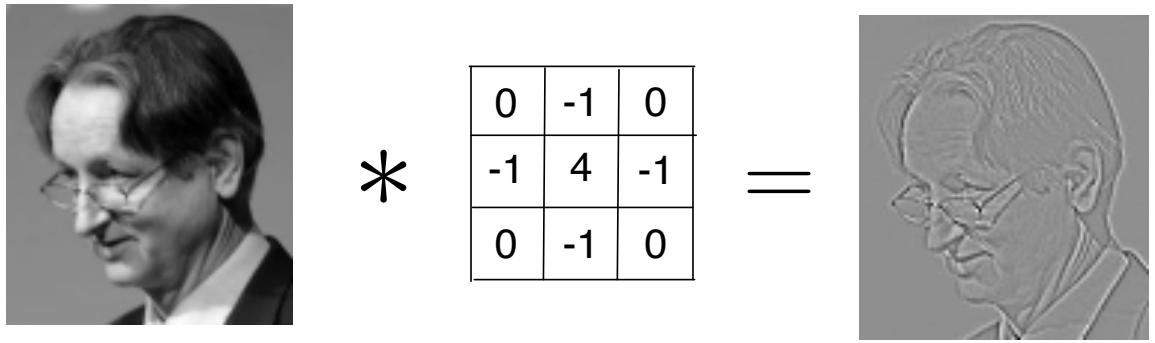
- We can sharpen images



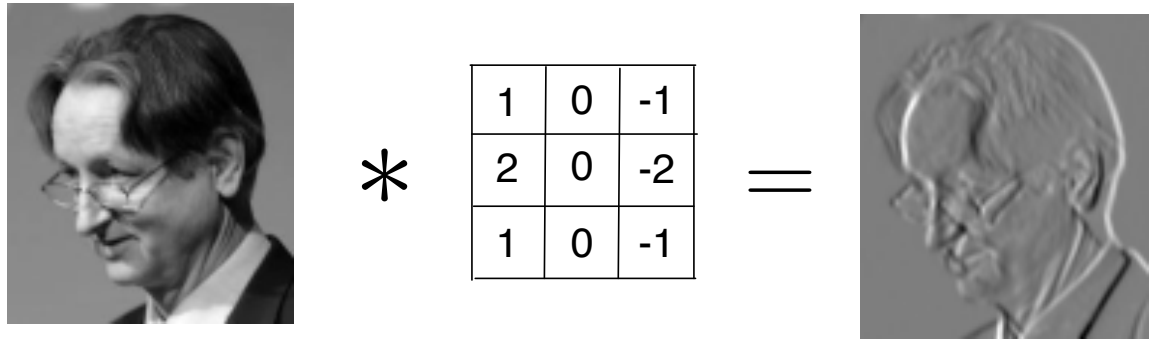
Picture credits from Roger Grosse

Examples

- We can detect edges



- Sobel filter



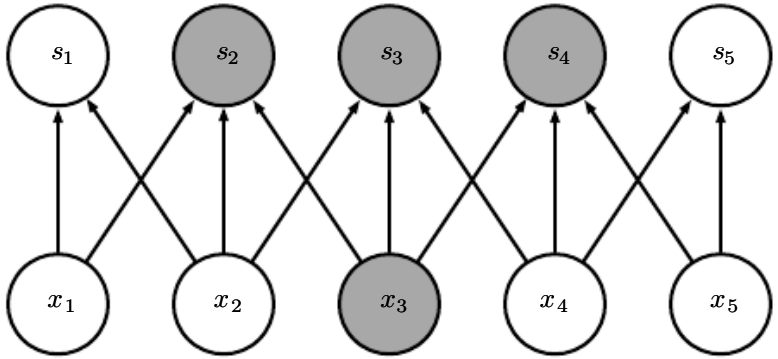
Picture credits from Roger Grosse

The motivation of convolutions

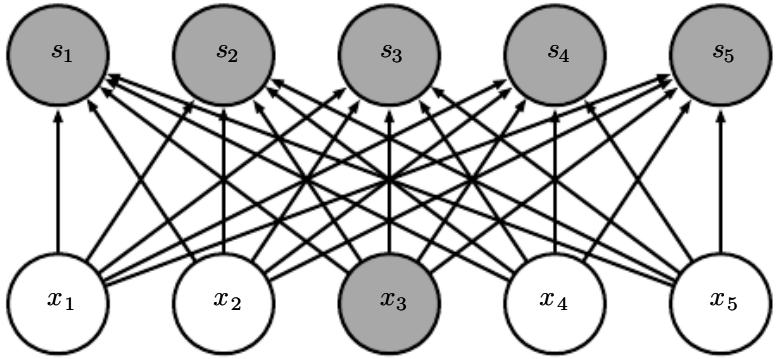
- Sparse interaction, or local connectivity
  - The receptive field of the neuron, or the filter size.
  - The connections are local in space (width and height), but always full in depth
  - A set of learnable filters
- Parameters sharing, the weights are tied
- Equivariant representation: same operation at different places

The motivation of convolutions

- Sparse interaction, or local connectivity
  - This is accomplished by making the kernel smaller than the input.
  - reduces the memory requirements of the model
  - improves its statistical efficiency.



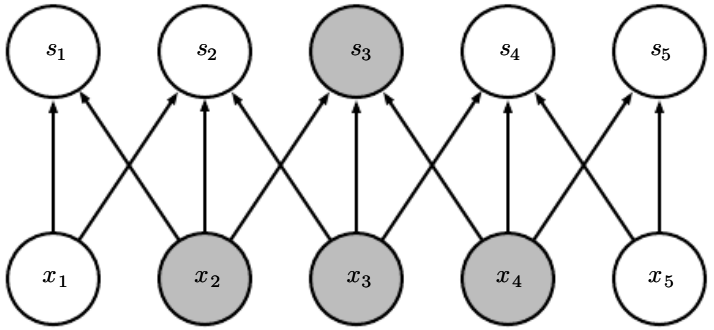
sparse connection



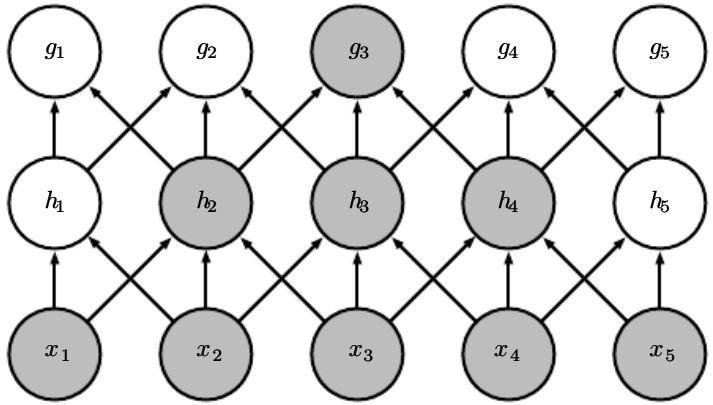
full connection

The motivation of convolutions

- Sparse interaction, or local connectivity
  - Receptive field is the kernel/filter size
  - The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers.



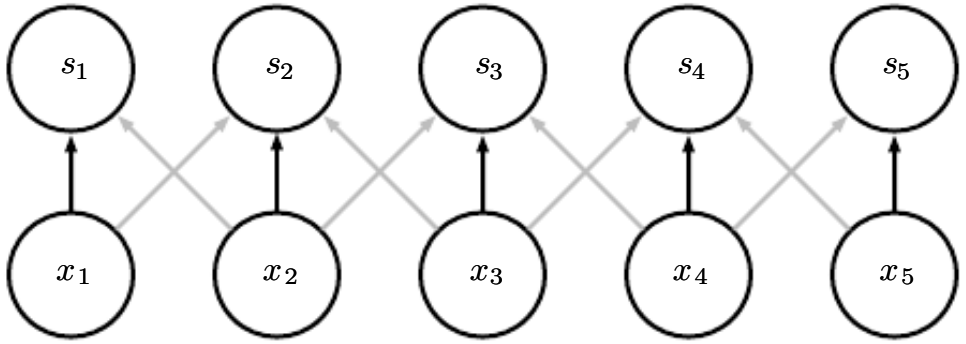
convolution with a kernel of width 3



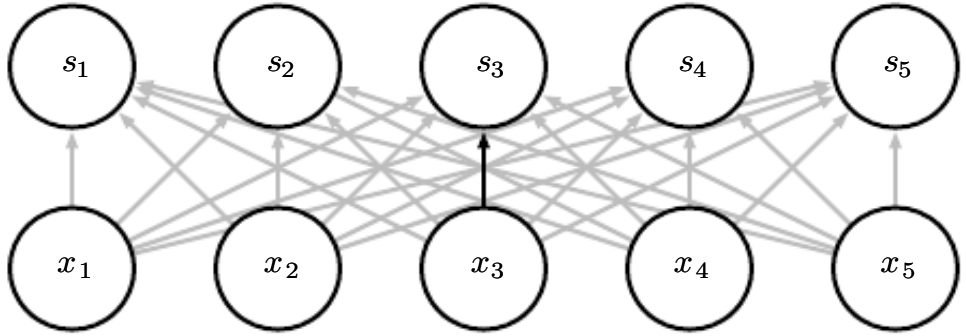
receptive field of the units in the deeper layers

The motivation of convolutions

- Parameters sharing, the weights are tied
  - refers to using the same parameter for more than one function in a model
  - each member of the kernel is used at every position of the input



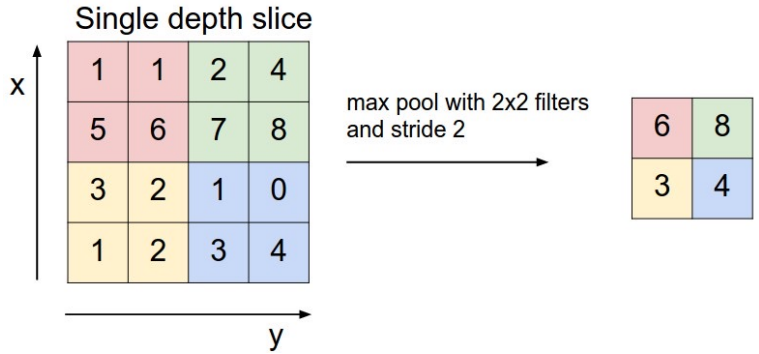
parameter sharing



no parameter sharing

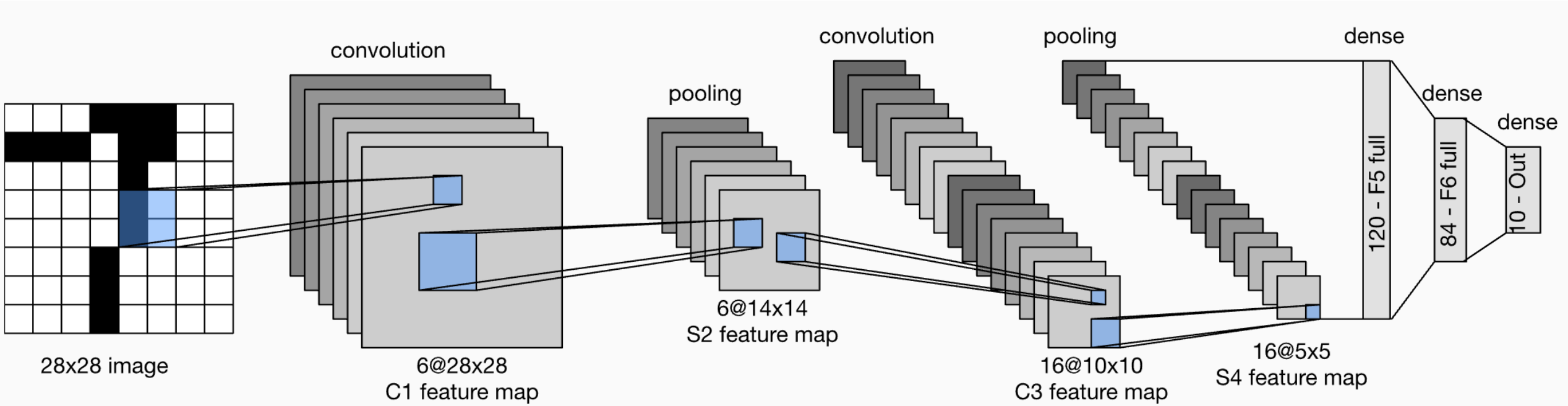
The pooling operations (not 🏊)

- A pooling function
  - replaces the output of the net at a certain location with a summary statistic of the nearby outputs.
  - e.g., max pooling operation reports the maximum output within a rectangular neighbourhood.
    - How could you implement circle-shaped pooling?
  - can reduce spatial size and thus improve the computational efficiency
  - pooling over spatial regions produces invariance to translation
  - pooling is essential for handling inputs of varying size.



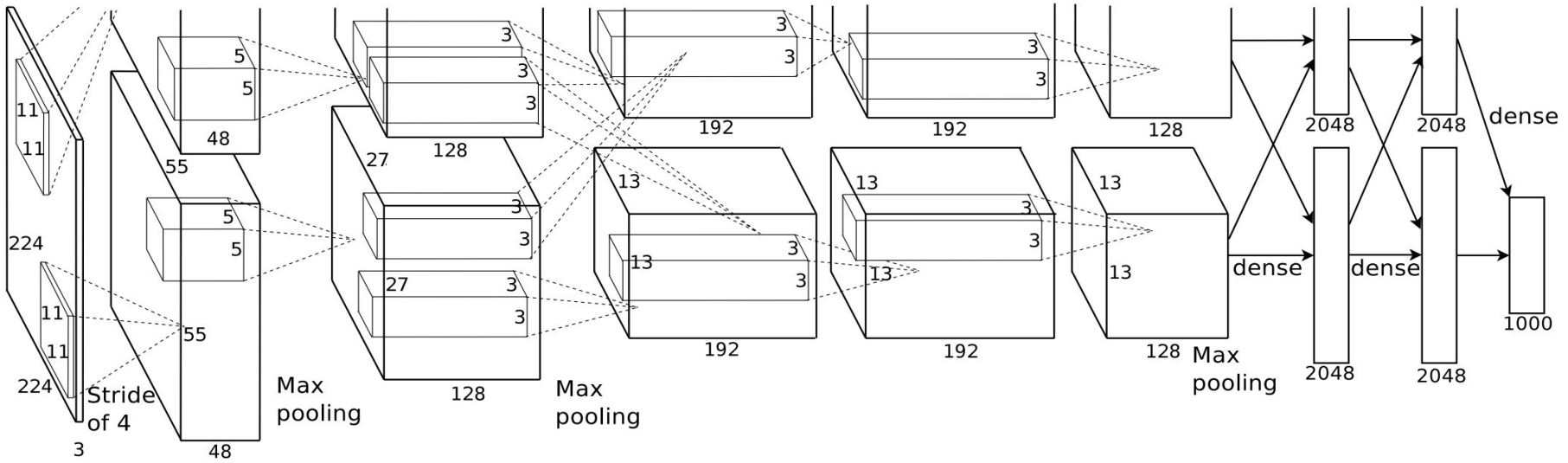
LeNet-5

- LeNet-5 was one of the earliest convolutional neural networks
- Yann LeCun et al. first applied the backpropagation algorithm
- Two convolutional layers and three fully-connected layers (--> 5-layer deep)



AlexNet: similar principles, but some extra engineering.

"AlexNet" — Won the ILSVRC2012 Challenge



Major breakthrough: 15.3% Top-5 error on ILSVRC2012 (Next best: 25.7%)

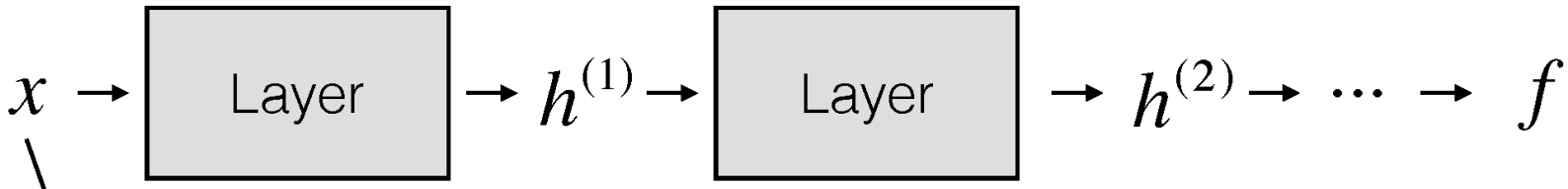
the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

[Krizhevsky, Sutskever, Hinton. NIPS 2012]

ConvNets

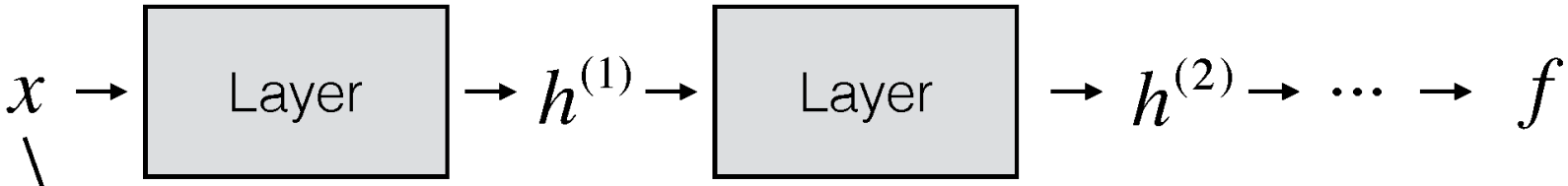
They’re just neural networks with
3D activations and weight sharing

What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)

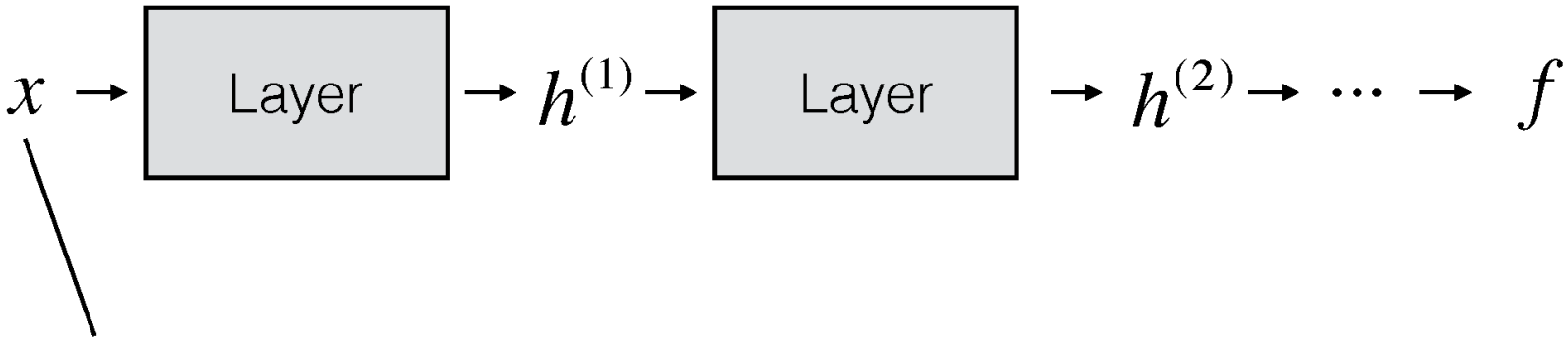
What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)

- We could flatten it to a 1D vector, but then we lose structure

What shape should the activations have?



- The input is an image, which is 3D (RGB channel, height, width)

- We could flatten it to a 1D vector, but then we lose structure

- What about keeping everything in 3D?

3D Activations

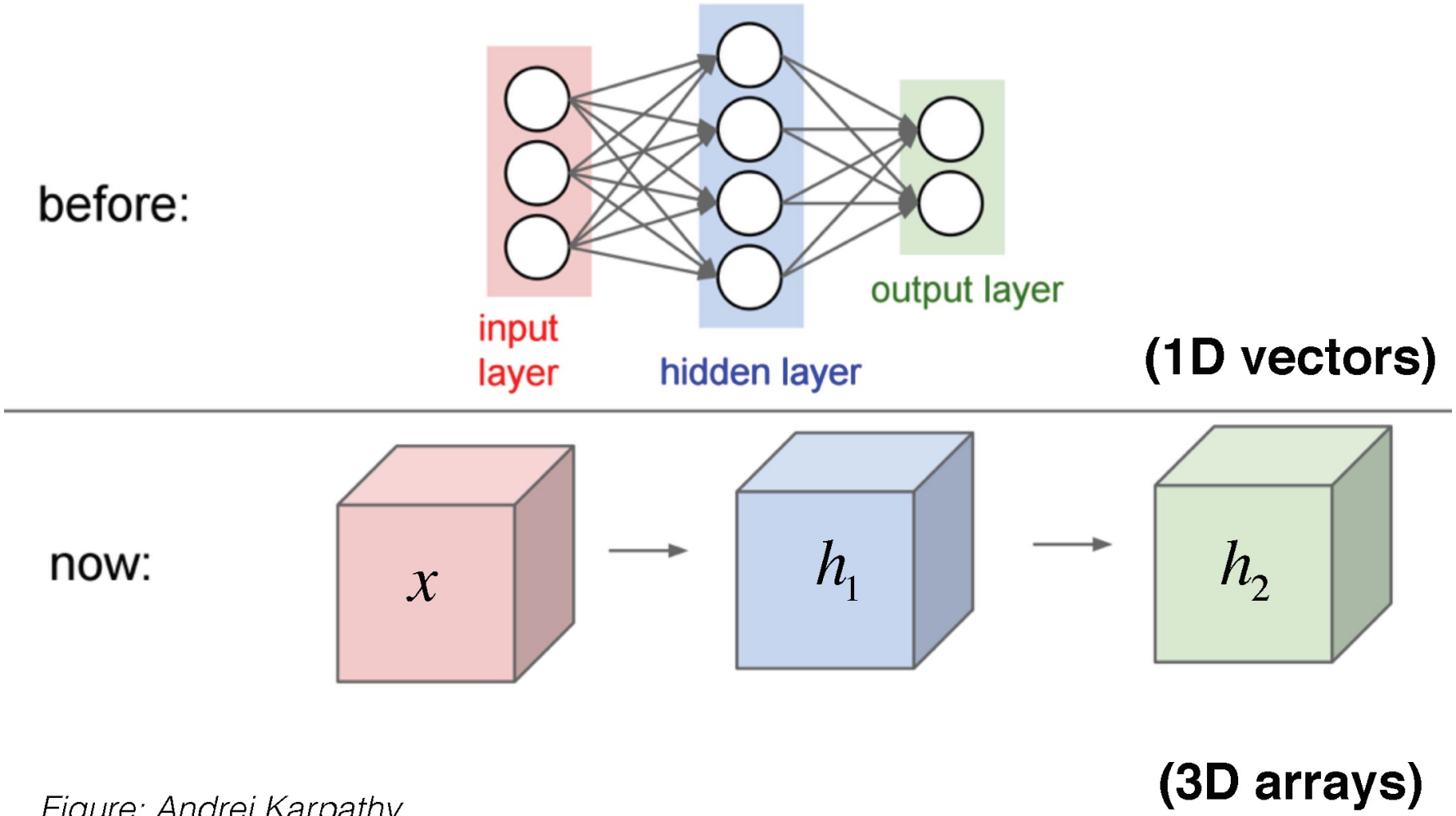
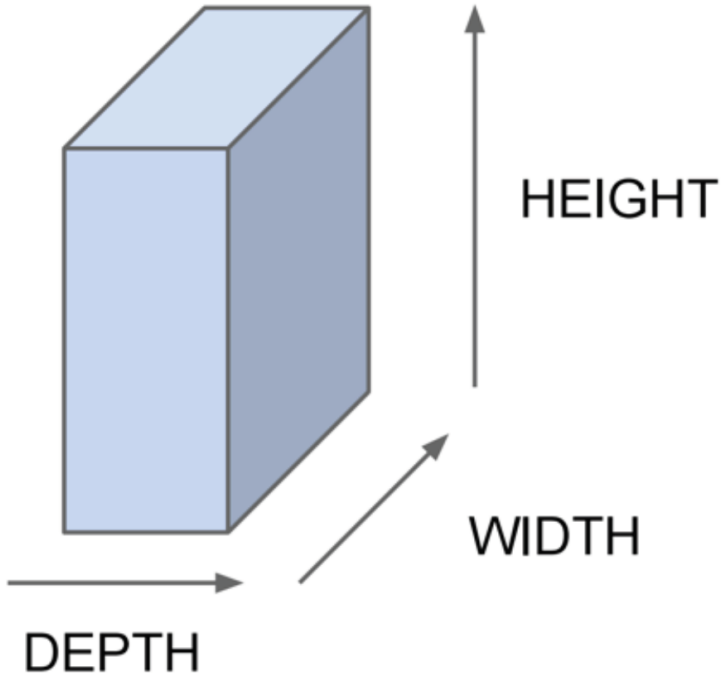


Figure: Andrej Karpathy

3D Activations

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

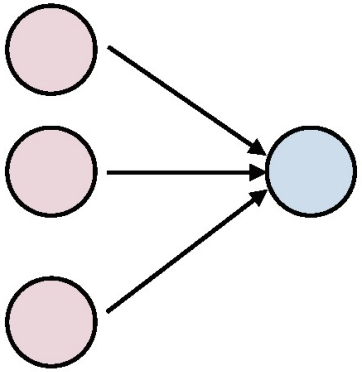


For example, a CIFAR-10 image is a 3x32x32 volume (3 depth — RGB channels, 32 height, 32 width)

Figure: Andrej Karpathy

3D Activations

1D Activations:



3D Activations:

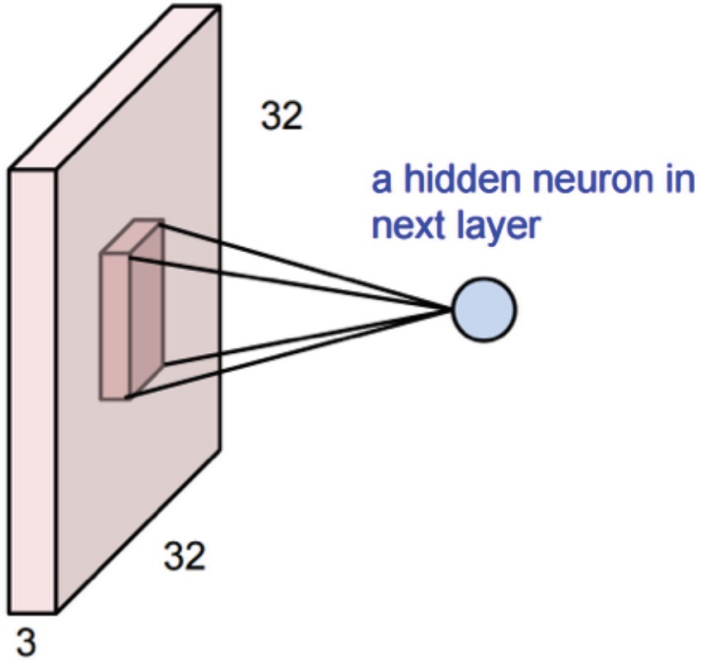
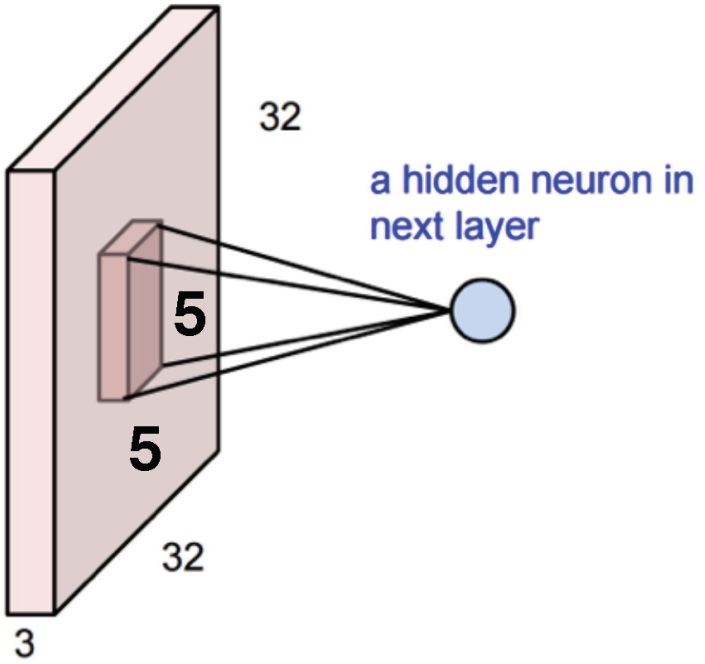


Figure: Andrej Karpathy

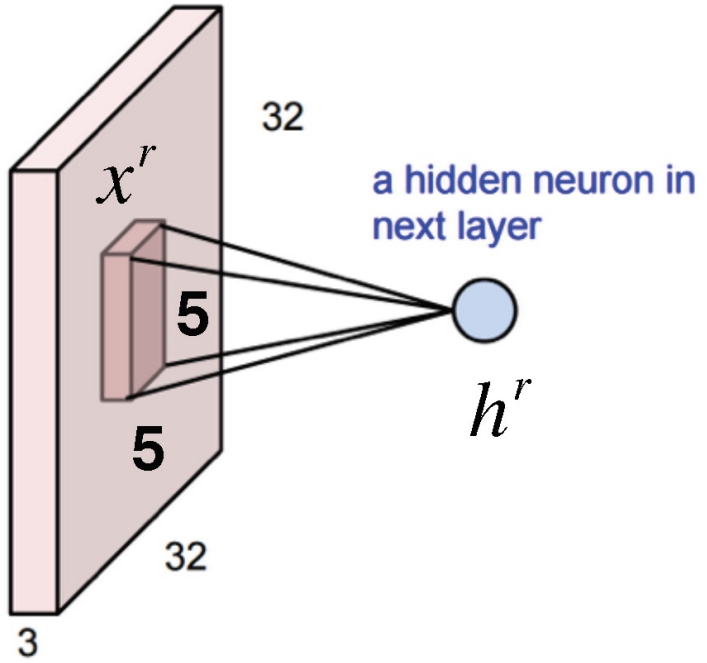
3D Activations



- The input is 3x32x32
- This neuron depends on a 3x5x5 chunk of the input
- The neuron also has a 3x5x5 set of weights and a bias (scalar)

Figure: Andrej Karpathy

3D Activations



Example: consider the region of the input "x^r"

With output neuron h^r

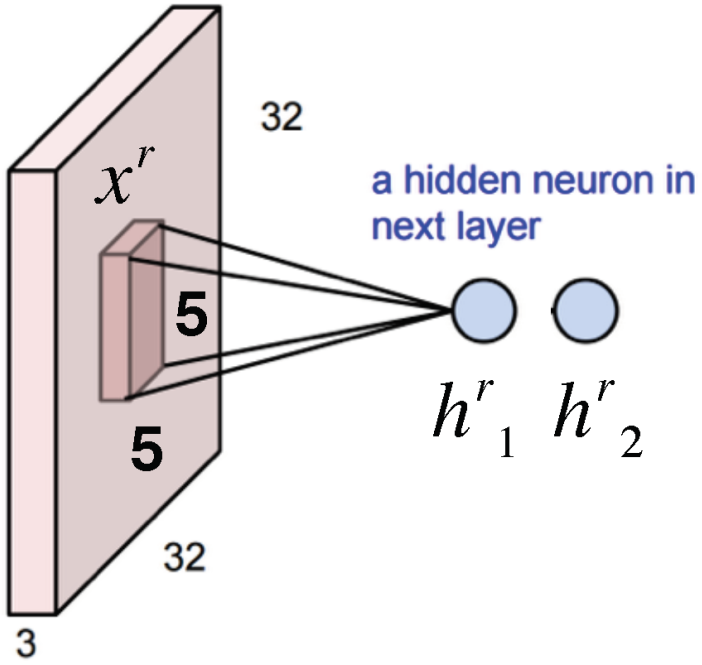
Then the output is:

$$h^r = \sum_{ijk} x^r{}_{ijk} W_{ijk} + b$$

Sum over 3 axes

Figure: Andrej Karpathy

3D Activations



With **2** output neurons

$$h^r{}_1 = \sum_{ijk} x^r{}_{ijk} W_{1ijk} + b_1$$

$$h^r{}_2 = \sum_{ijk} x^r{}_{ijk} W_{2ijk} + b_2$$

Figure: Andrej Karpathy

3D Activations

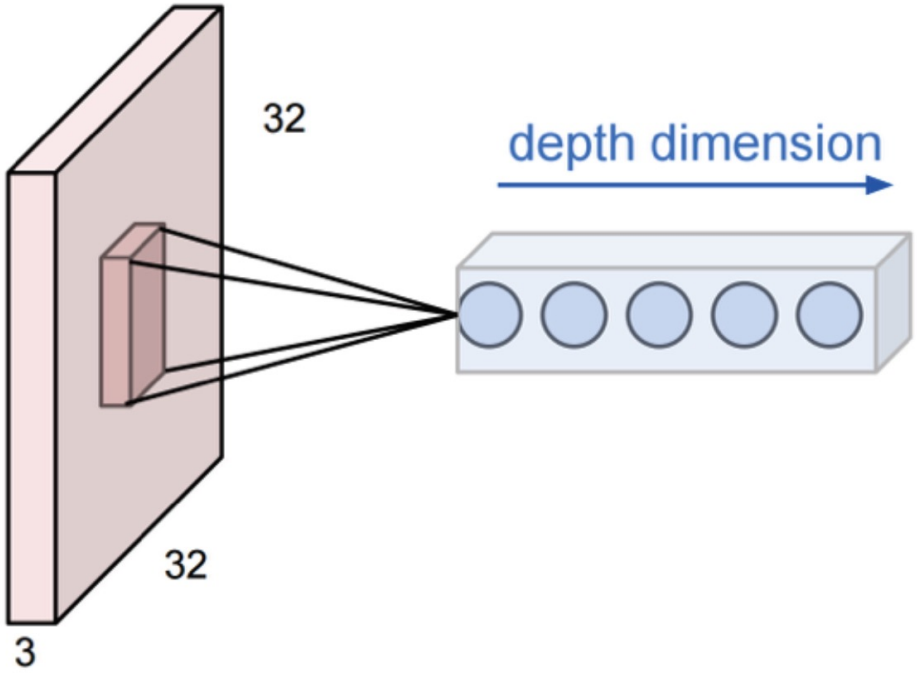
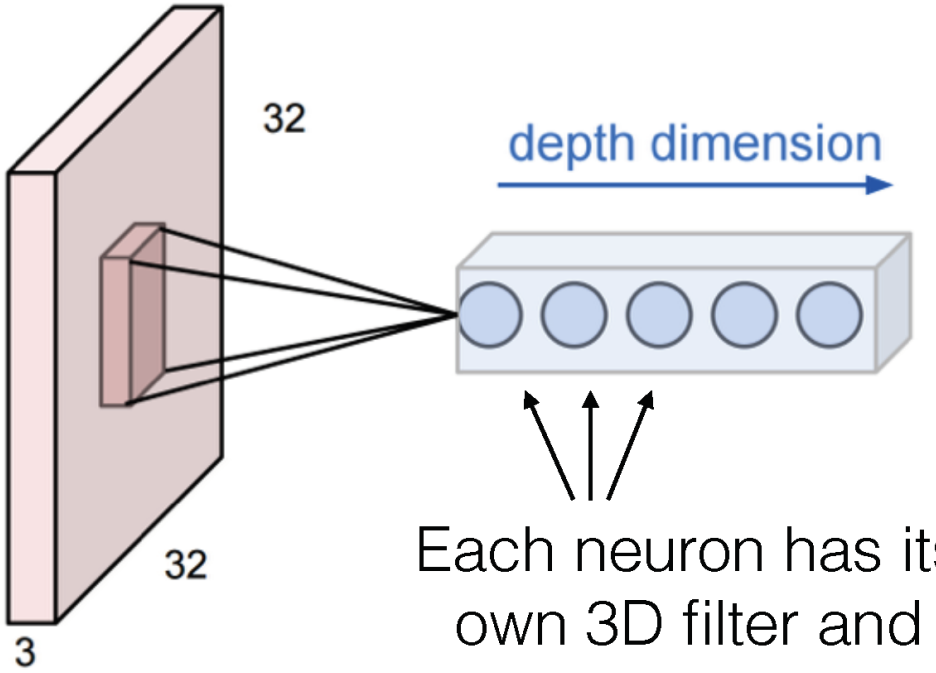


Figure: Andrej Karpathy

3D Activations



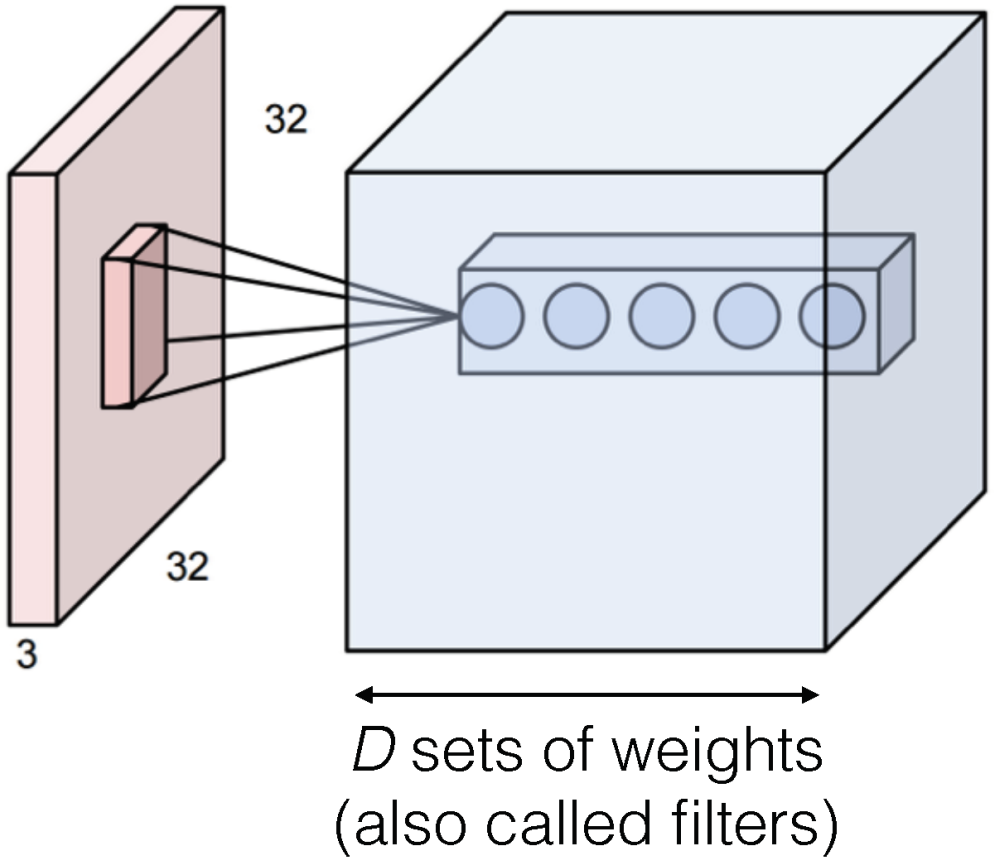
Each neuron has its own 3D filter and own (scalar) bias

We can keep adding more outputs

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

Figure: Andrej Karpathy

3D Activations



Now repeat this across the input

Weight sharing:

Each filter shares the same weights (but each depth index has its own set of weights)

Figure: Andrej Karpathy

3D Activations

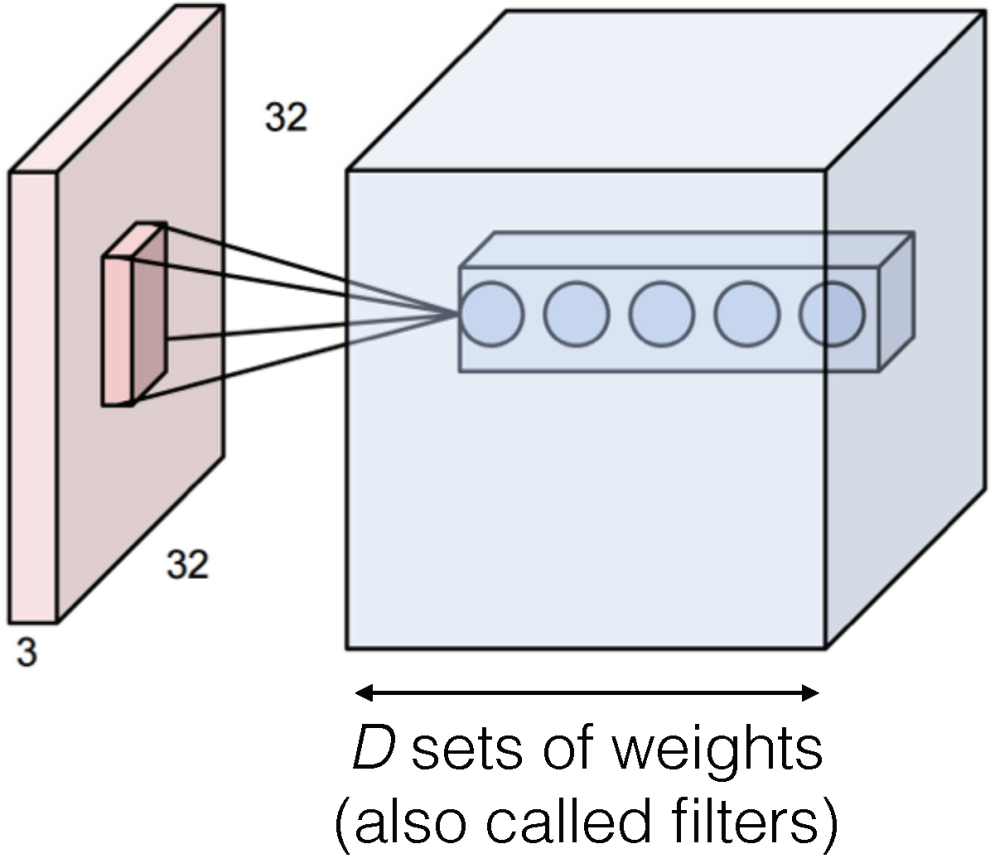
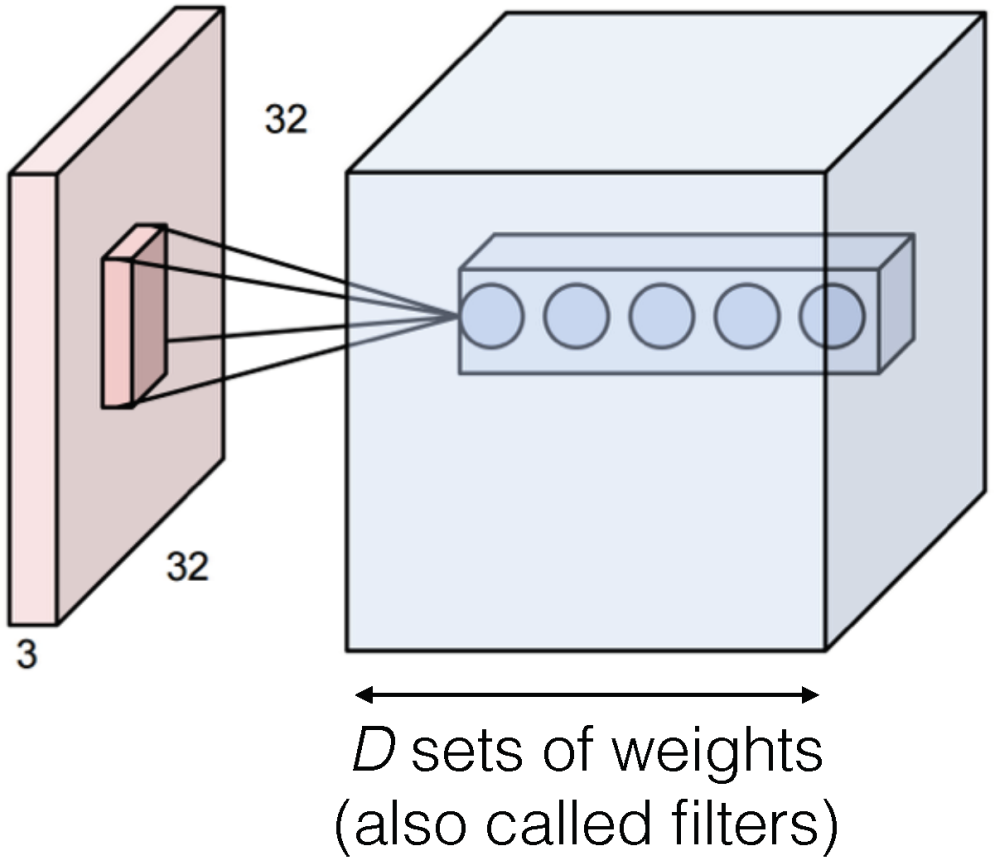


Figure: Andrej Karpathy

3D Activations

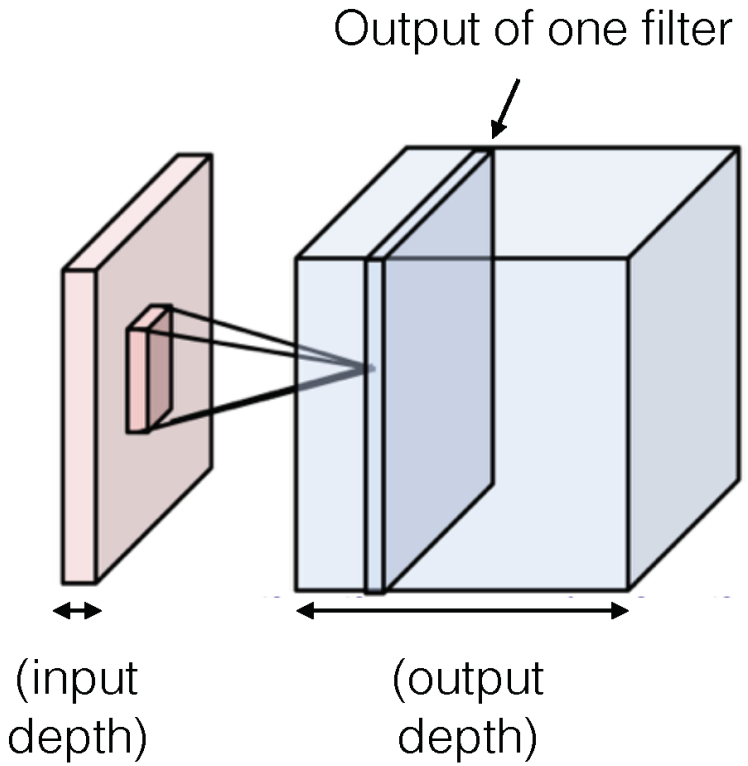


With weight sharing, this is called **convolution**

Without weight sharing, this is called a **locally connected layer**

Figure: Andrej Karpathy

3D Activations



One set of weights gives one slice in the output

To get a 3D output of depth *D*, use *D* different filters

In practice, ConvNets use many filters (~64 to 1024)

All together, the weights are **4** dimensional:
(output depth, input depth, kernel height, kernel width)

3D Activations

We can unravel the 3D cube and show each layer separately:

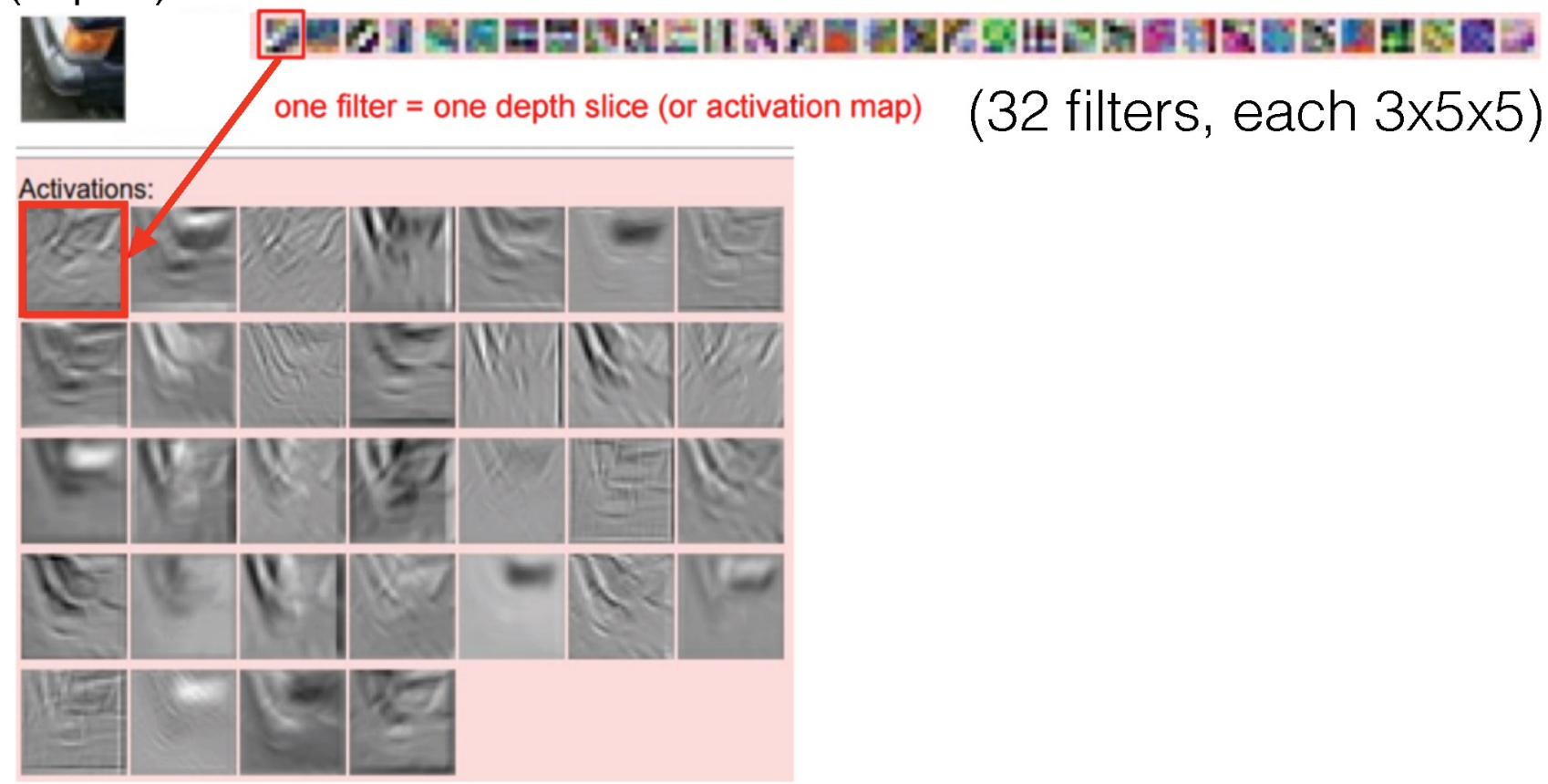


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

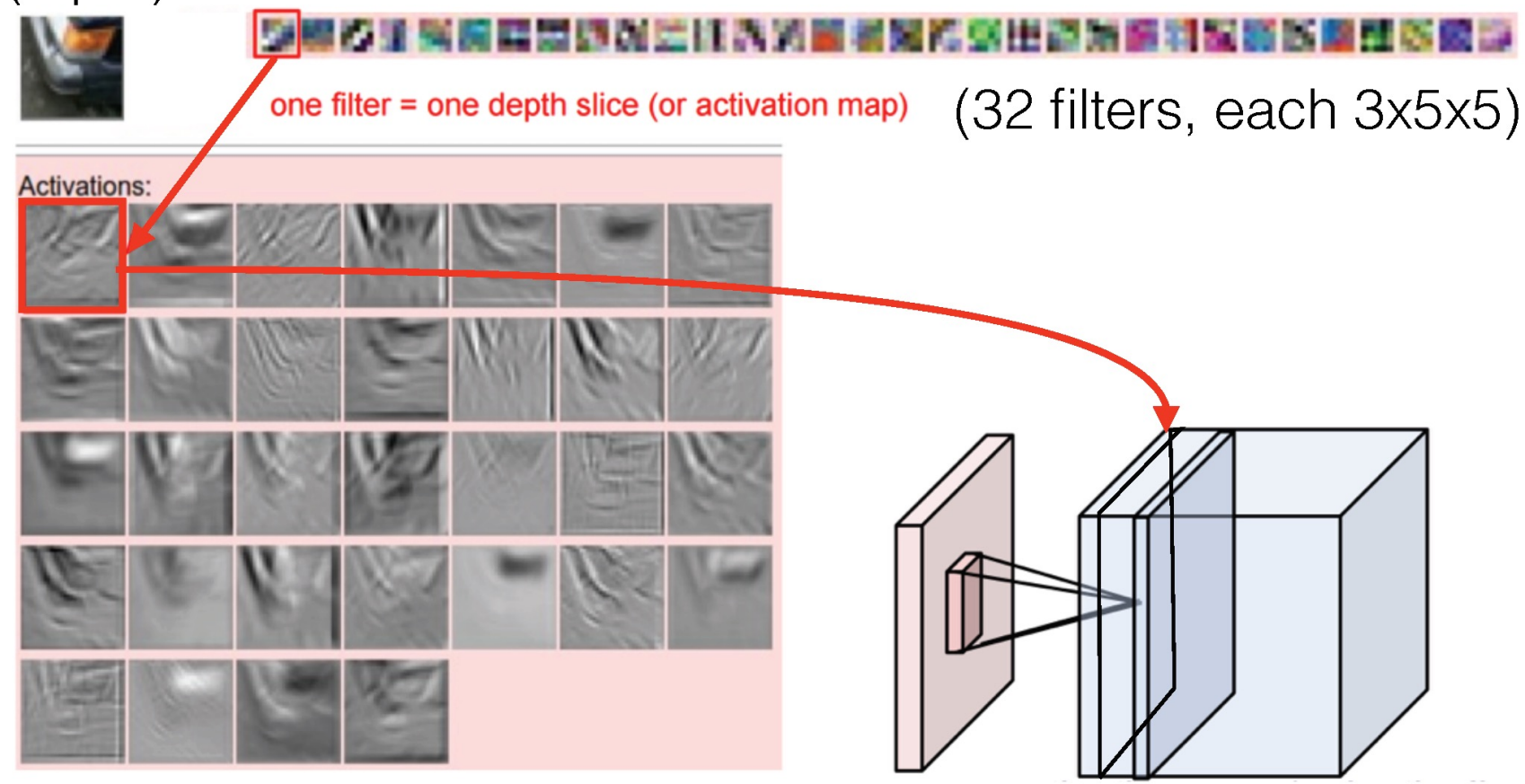


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

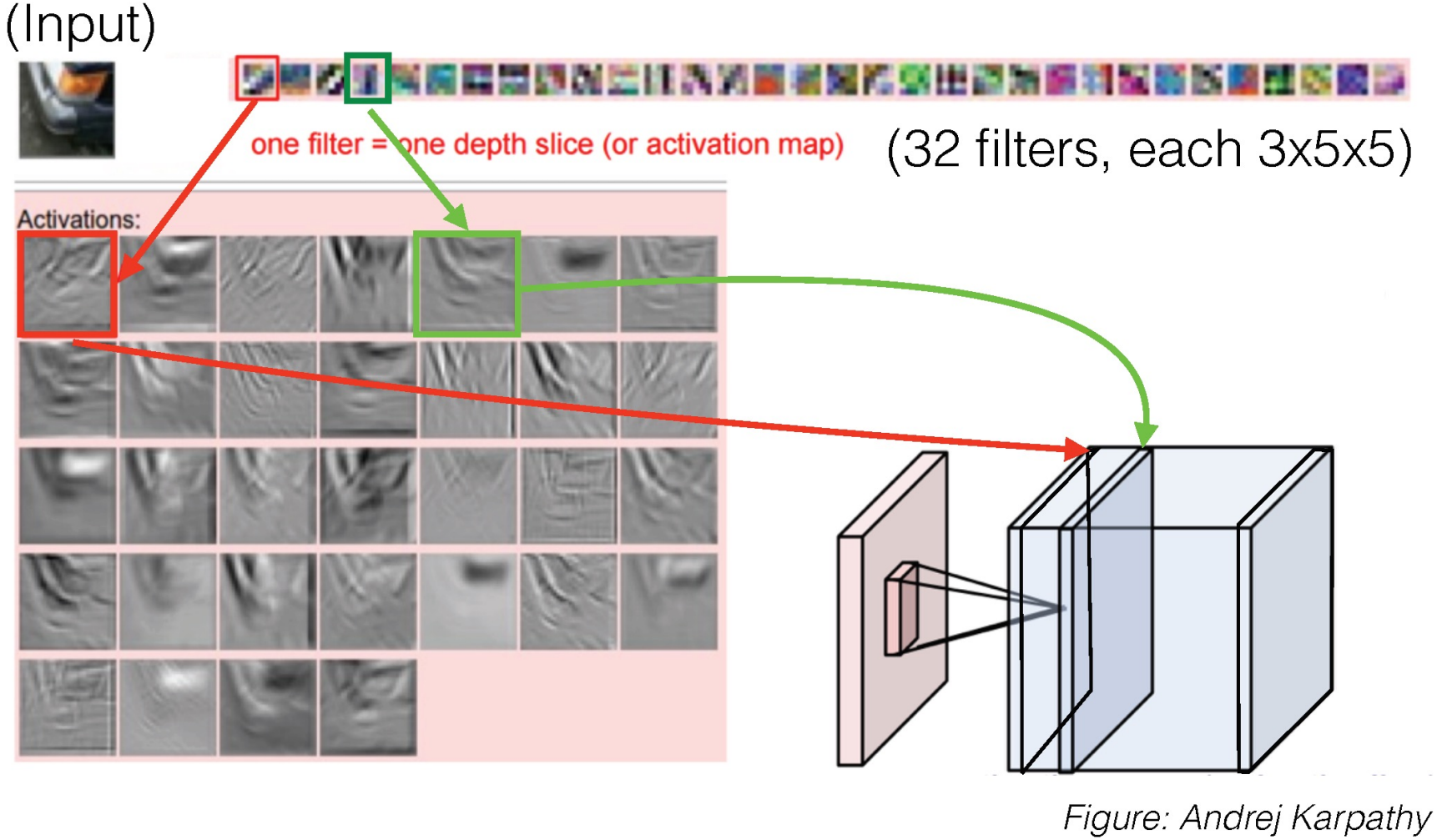


Figure: Andrej Karpathy

3D Activations

We can unravel the 3D cube and show each layer separately:

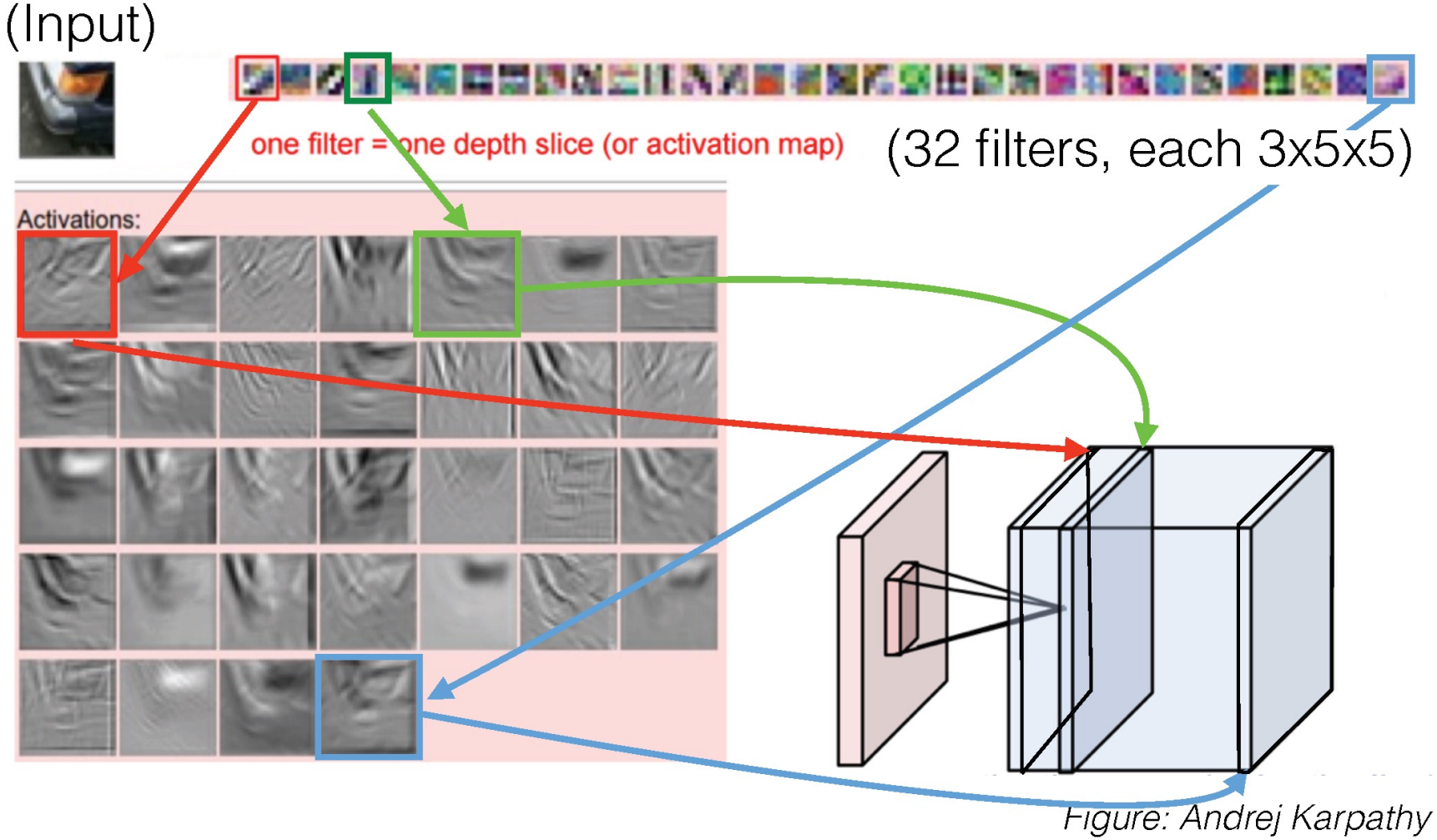
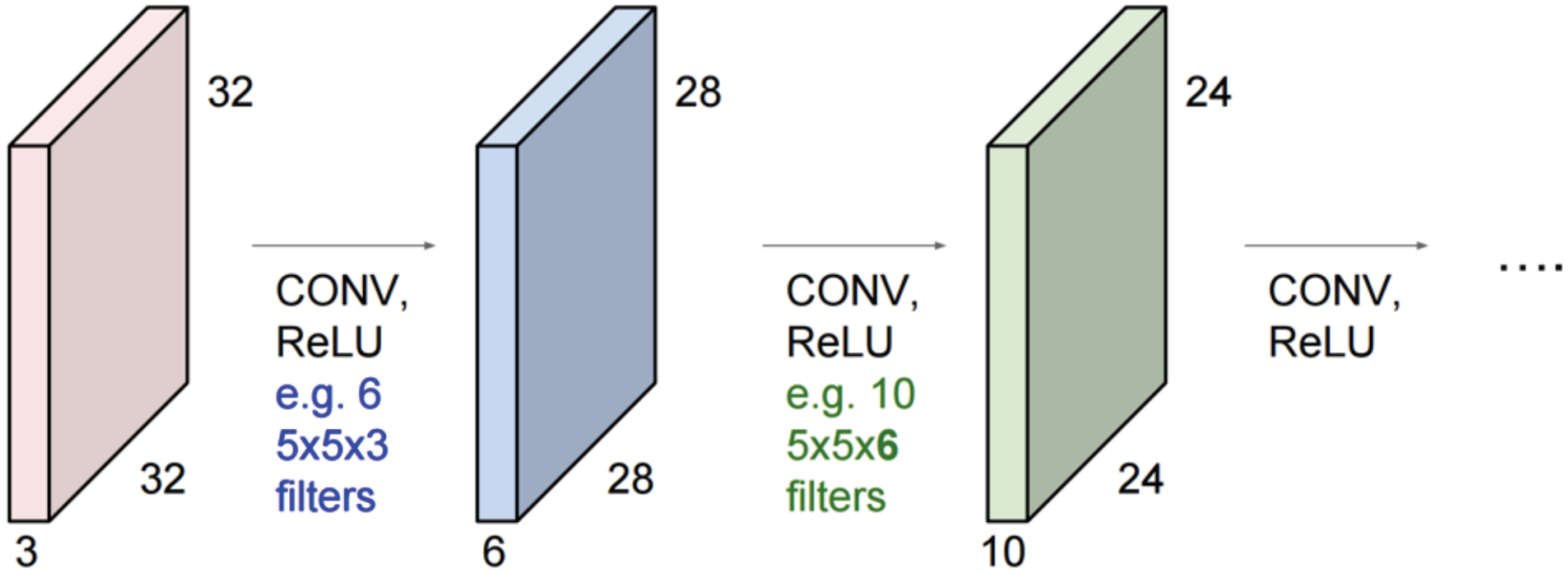


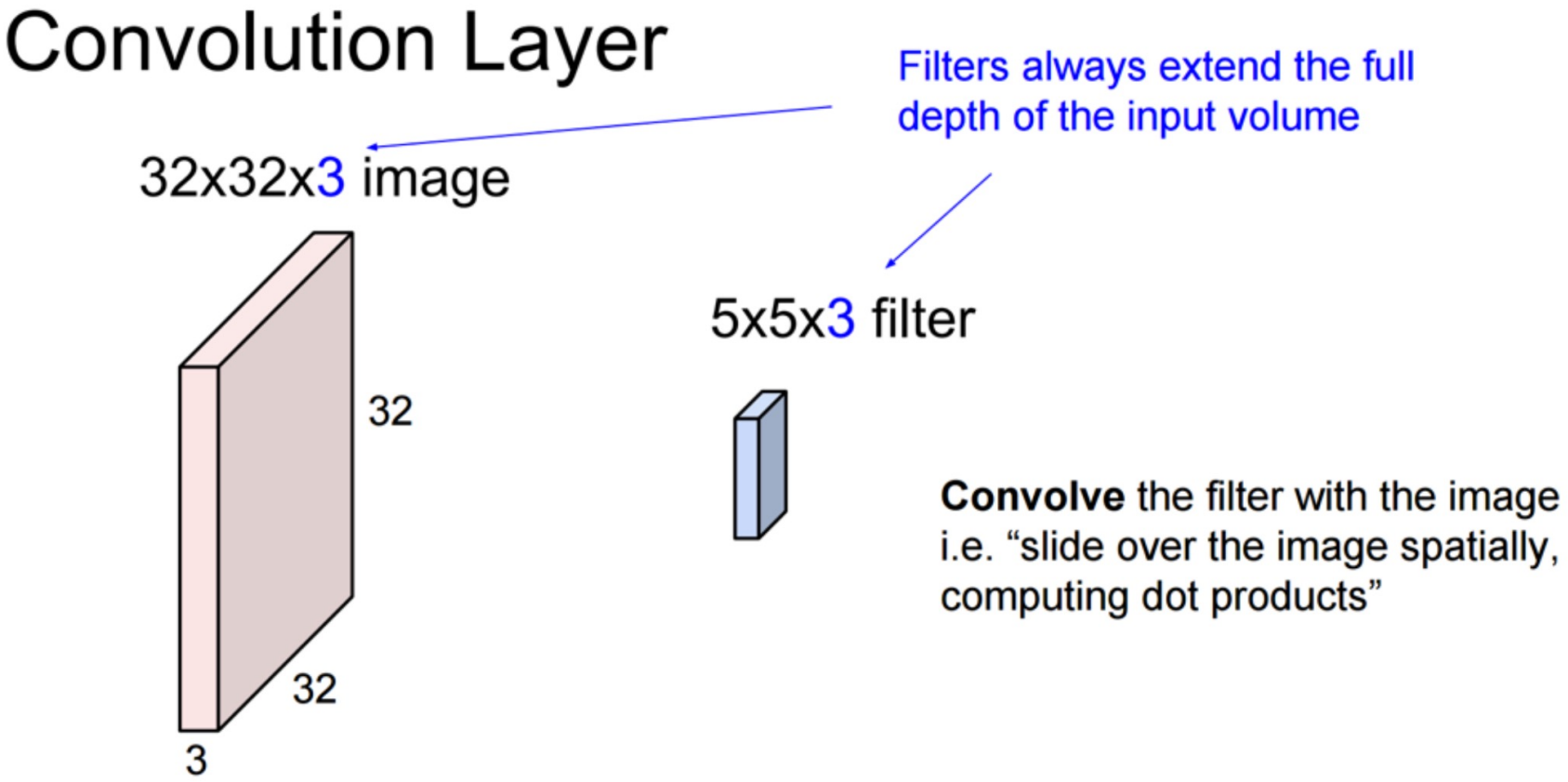
Figure: Andrej Karpathy

Putting it together

A **ConvNet** is a sequence of convolutional layers, interspersed with activation functions (and possibly other layer types)

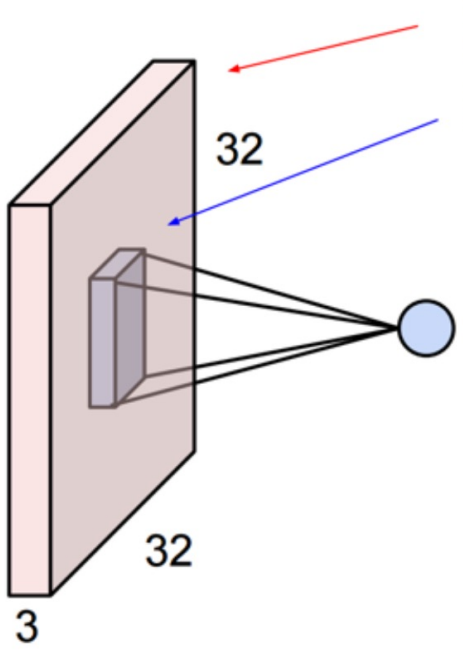


Putting it together



Putting it together

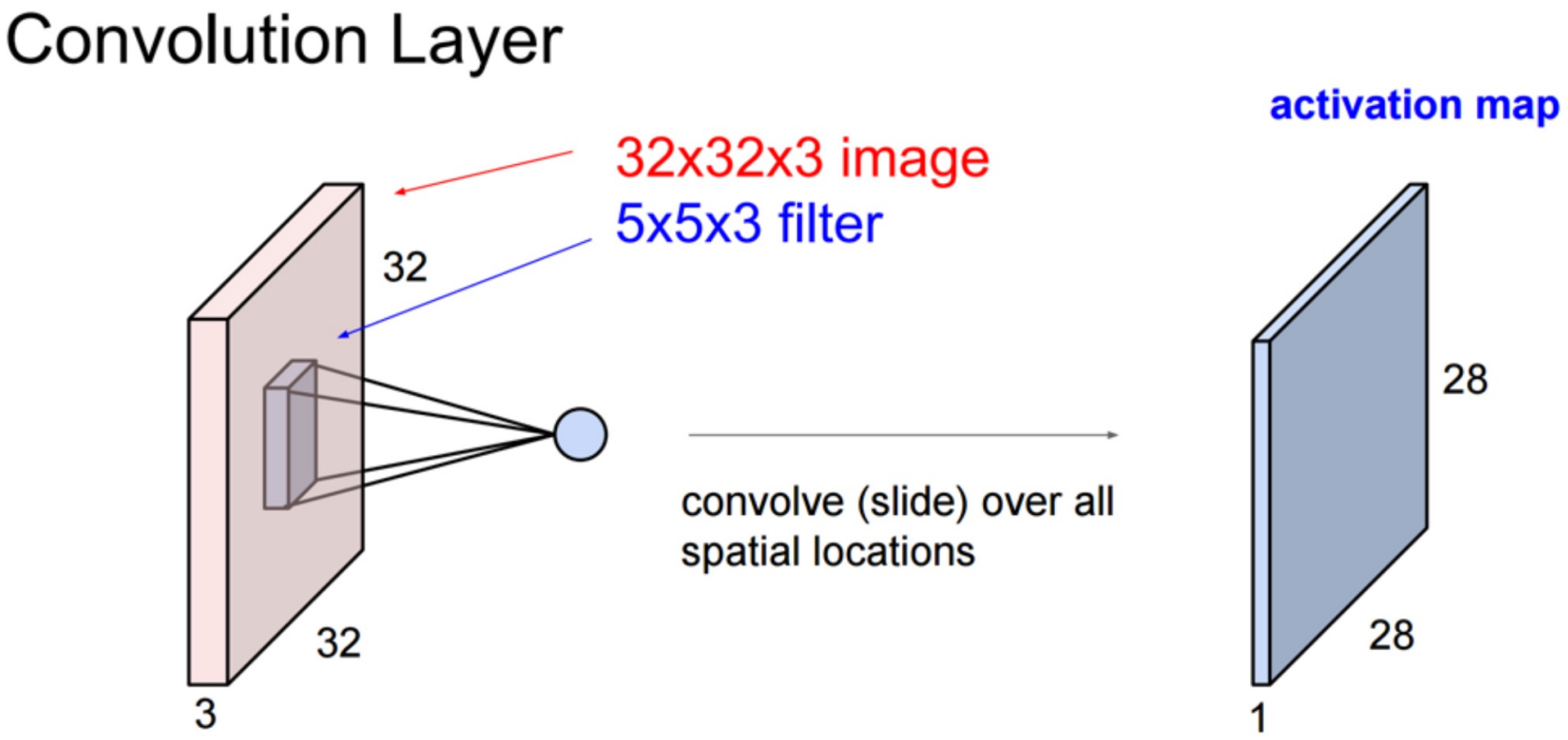
Convolution Layer



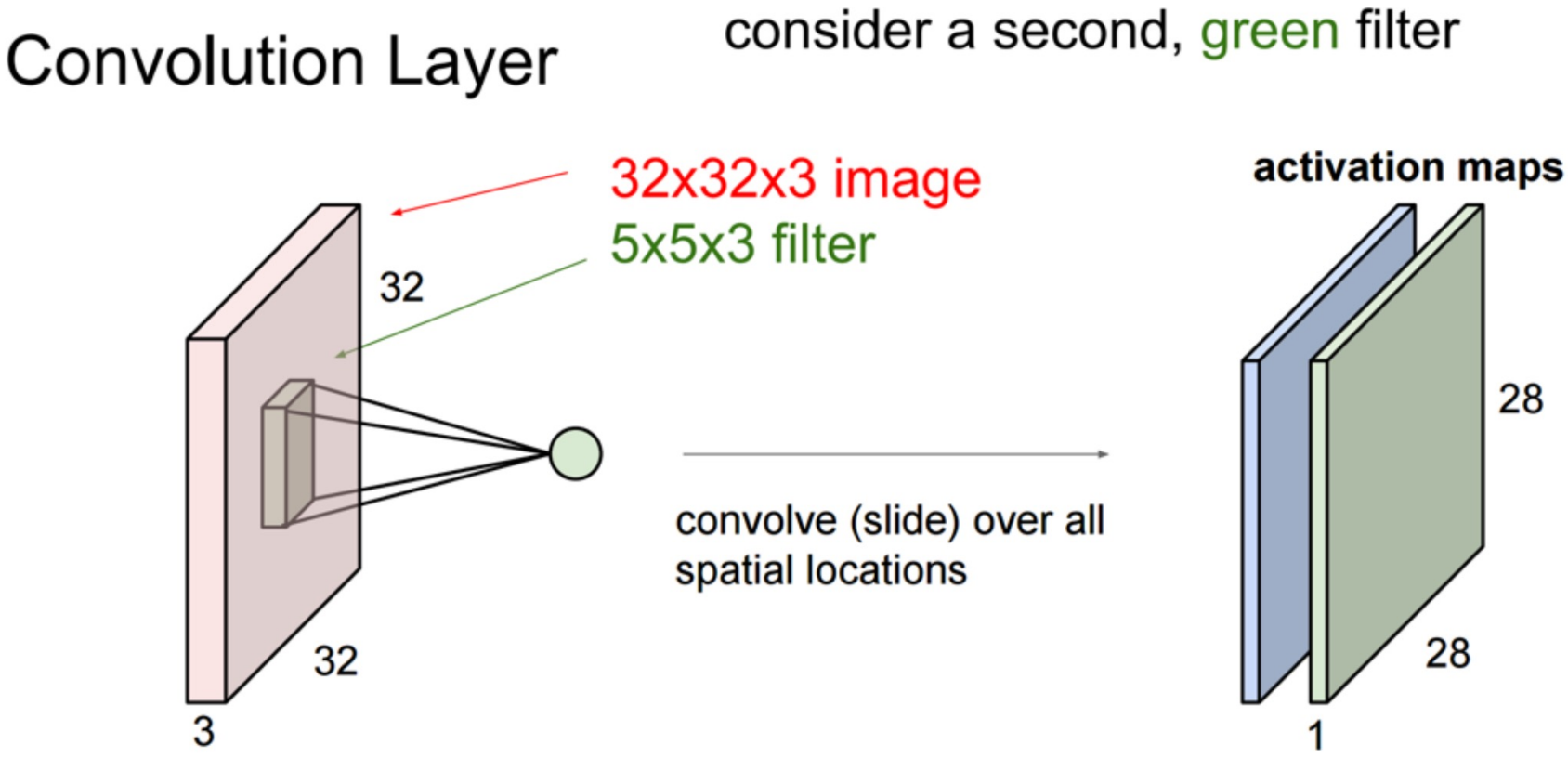
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Putting it together

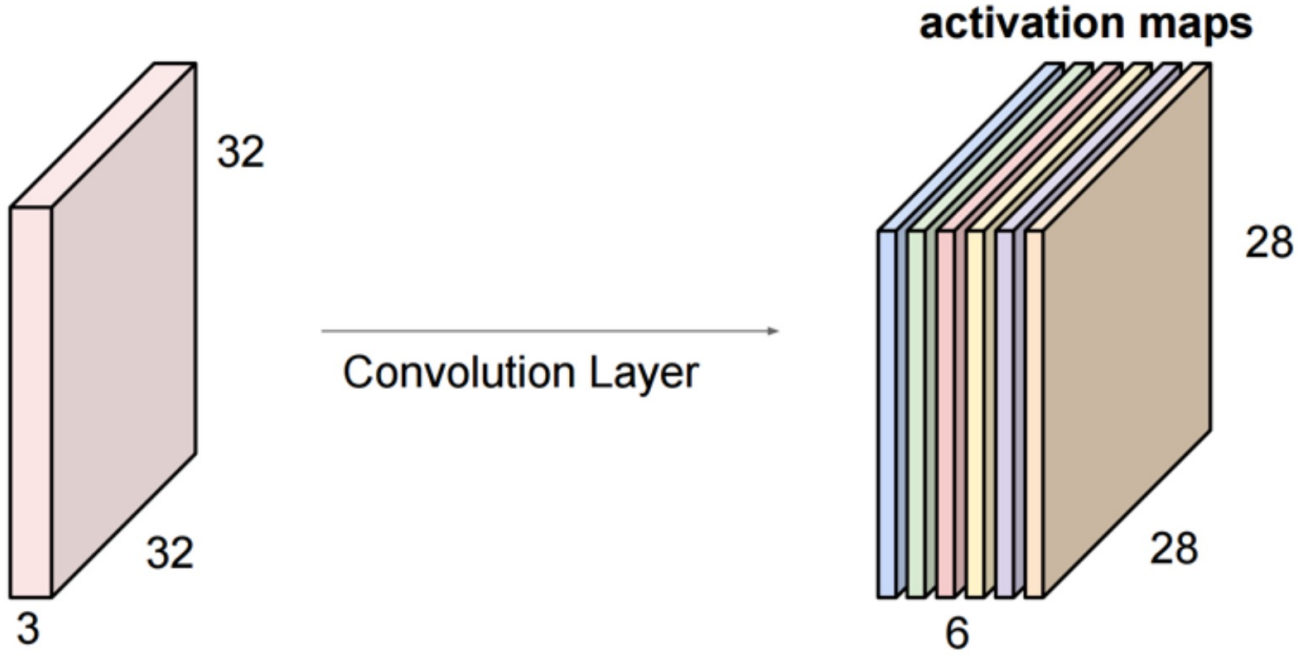


Putting it together



Putting it together

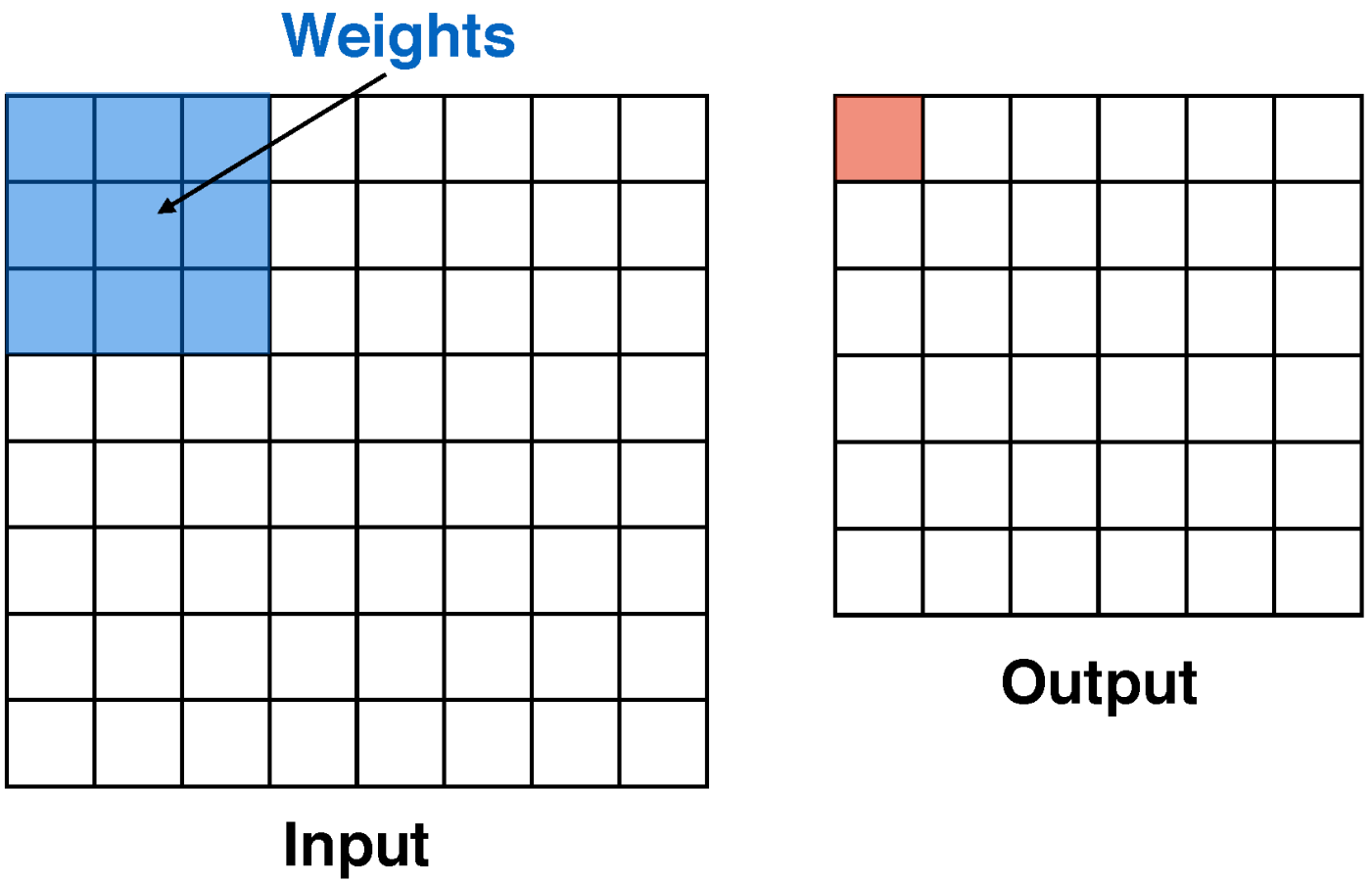
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



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

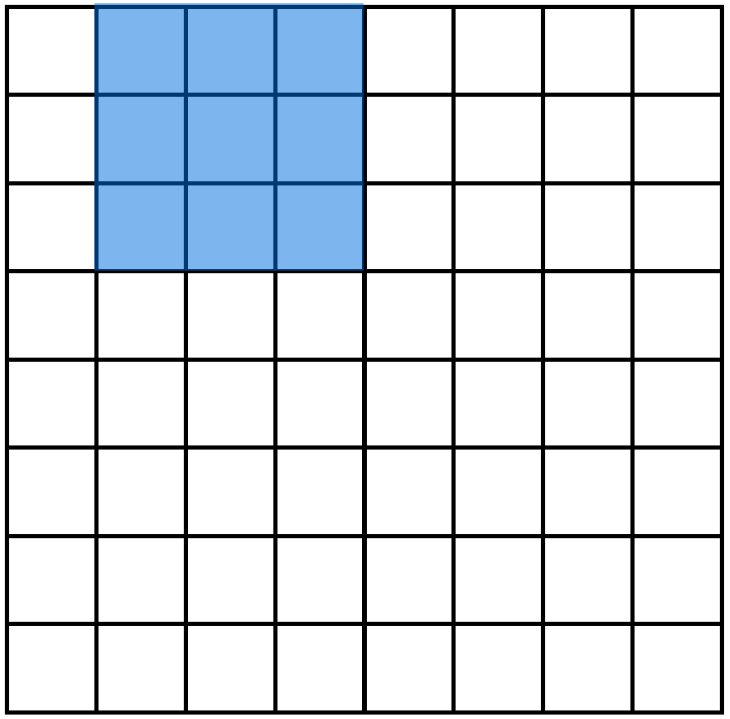
Convolution: Stride

During convolution, the weights “slide” along the input to generate each output

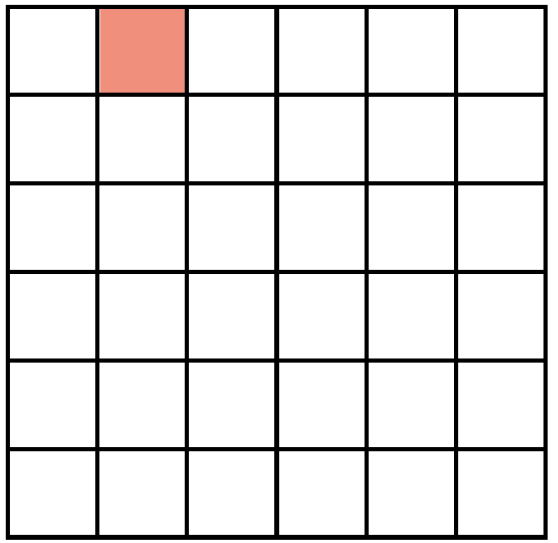


Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



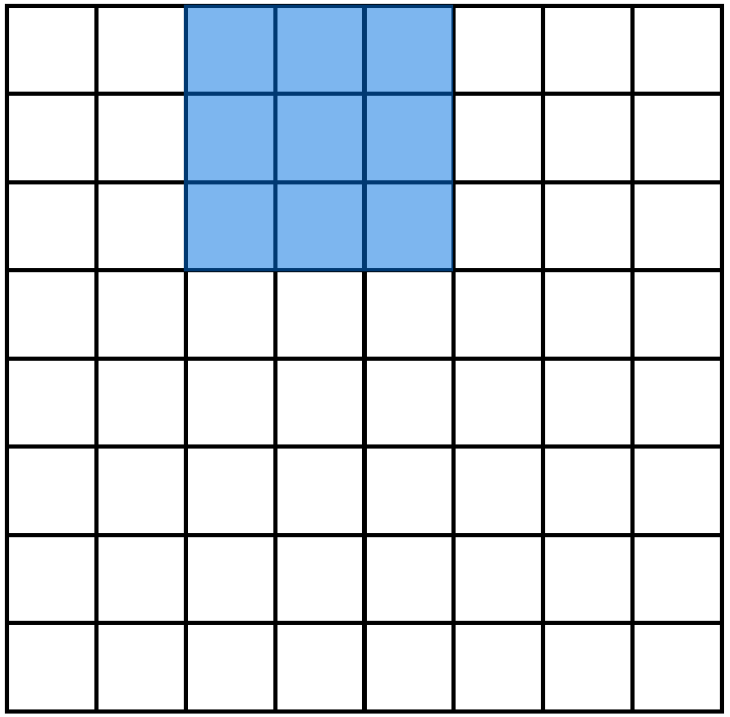
Input



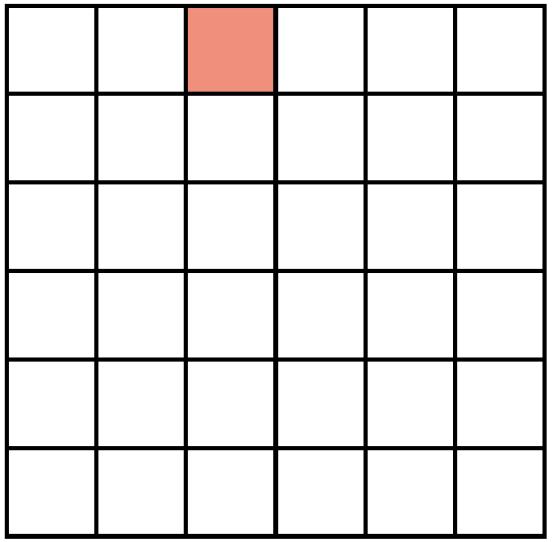
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



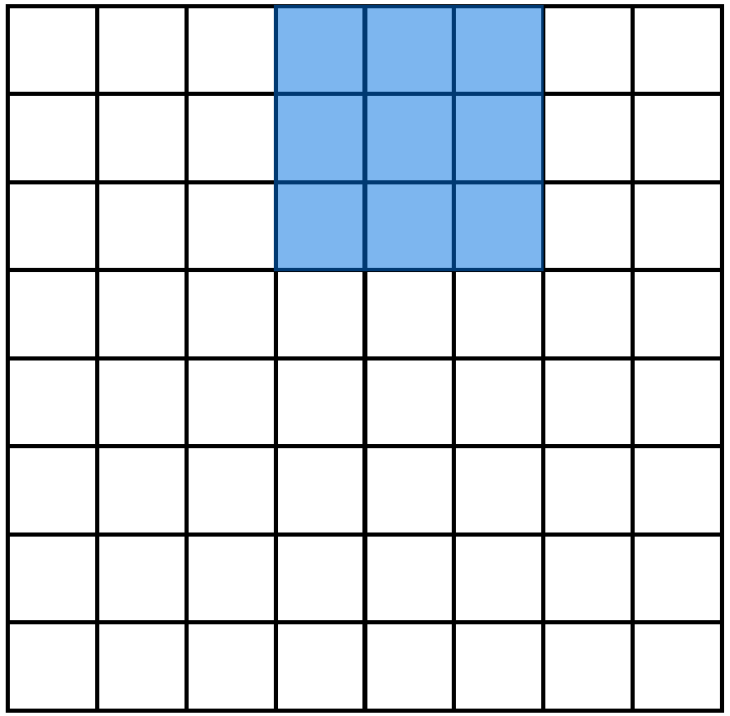
Input



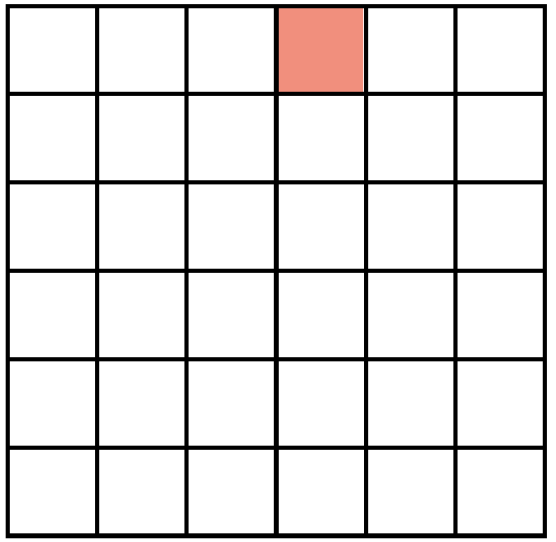
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



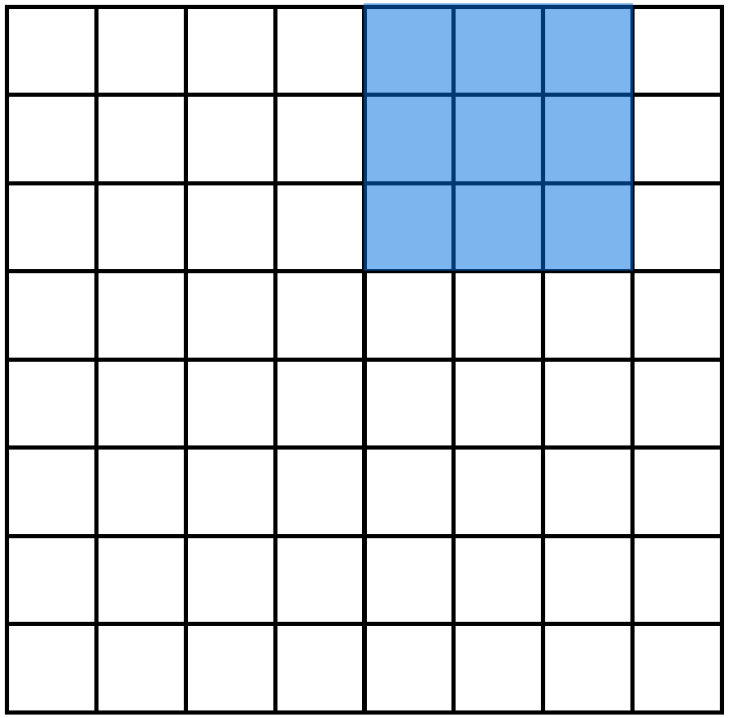
Input



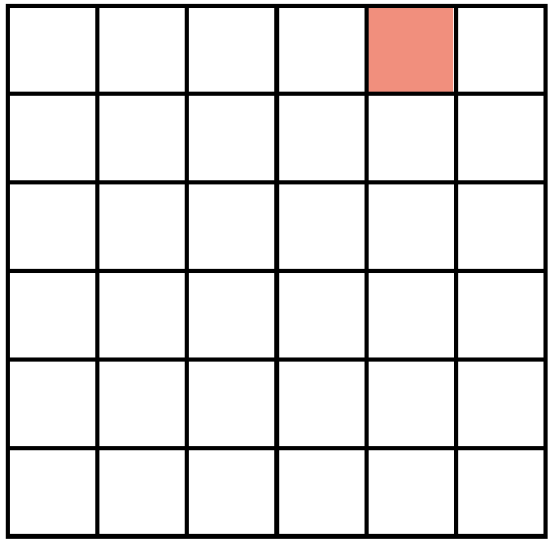
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



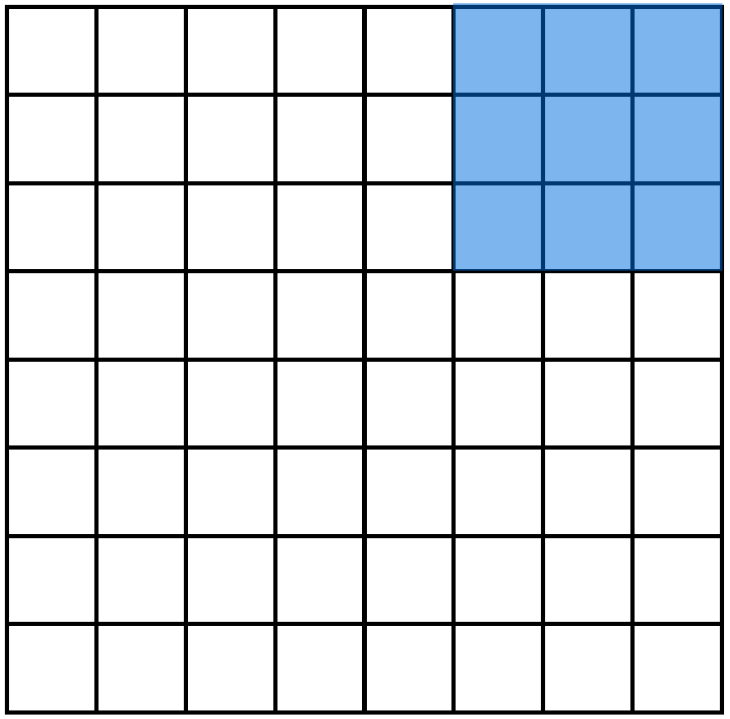
Input



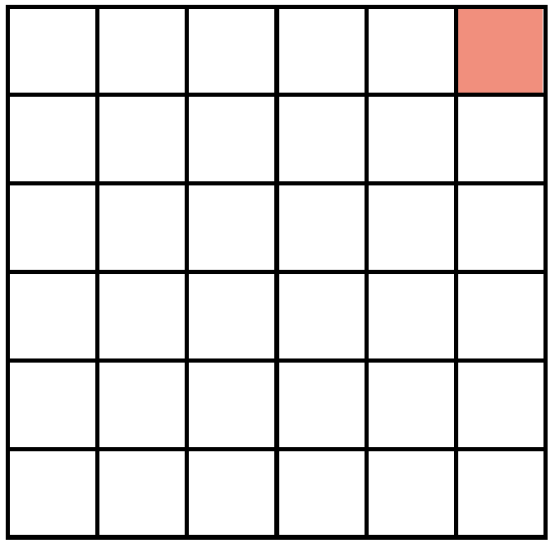
Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



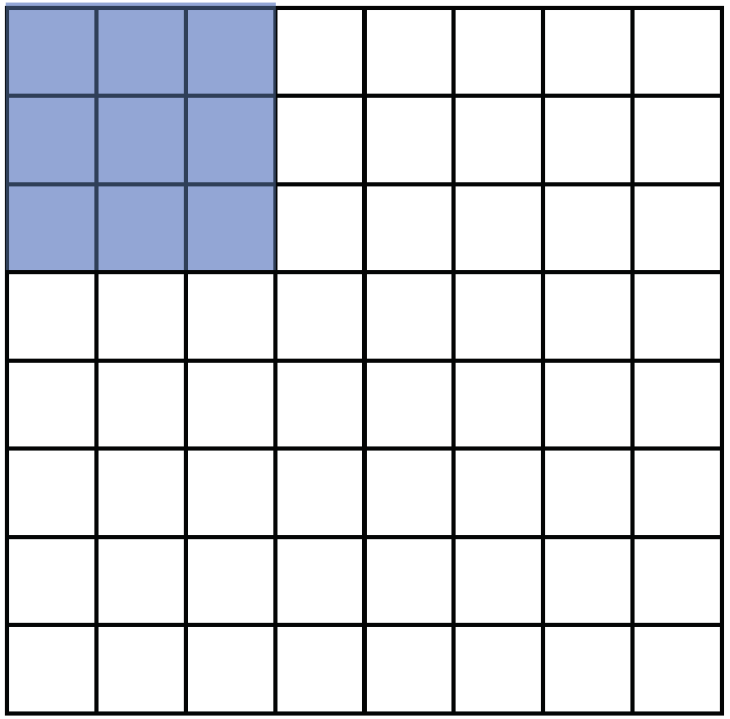
Input



Output

Convolution: Stride

During convolution, the weights “slide” along the input to generate each output



Input

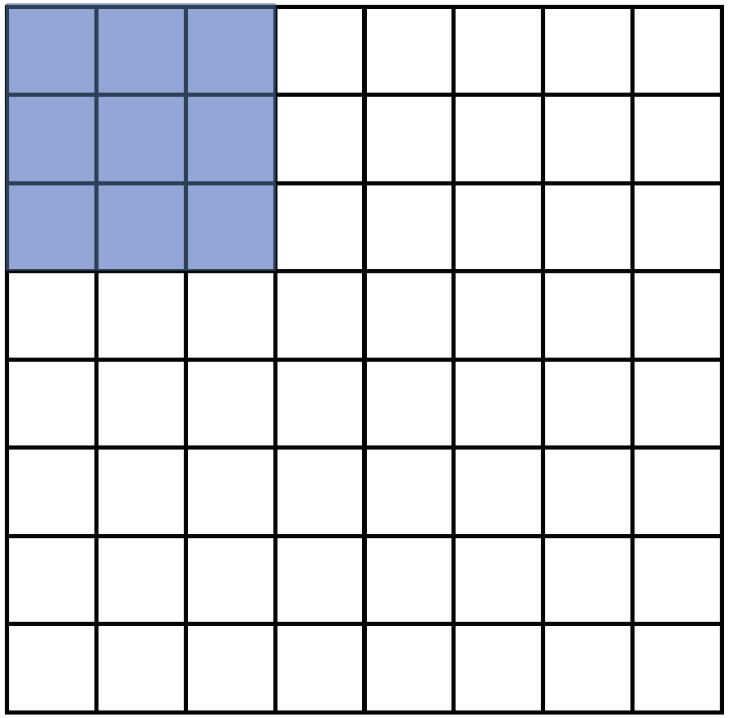
Recall that at each position, we are doing a **3D** sum:

$$h^r = \sum_{ijk} x^r{}_{ijk} W_{ijk} + b$$

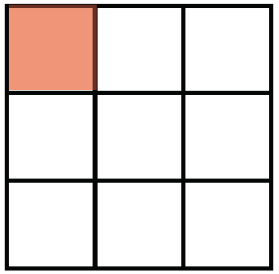
(channel, row, column)

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



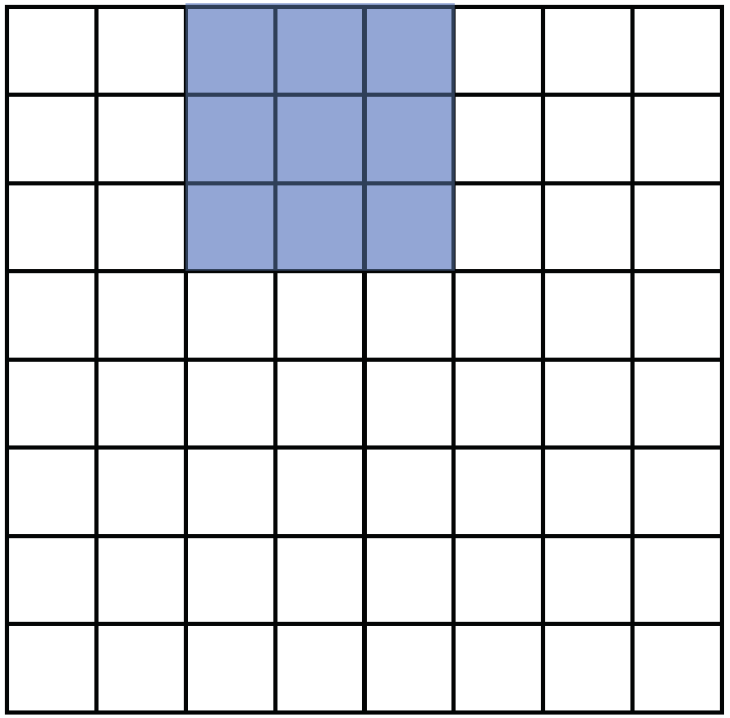
Input



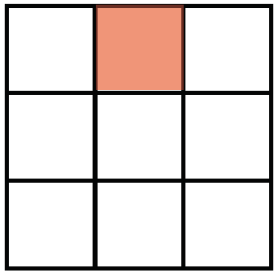
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



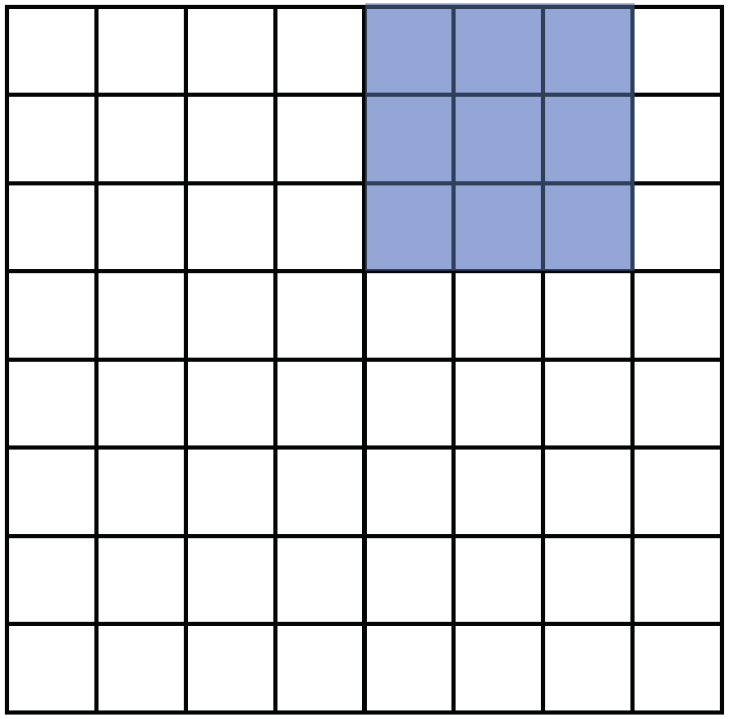
Input



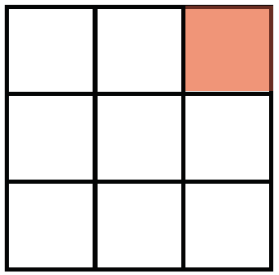
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



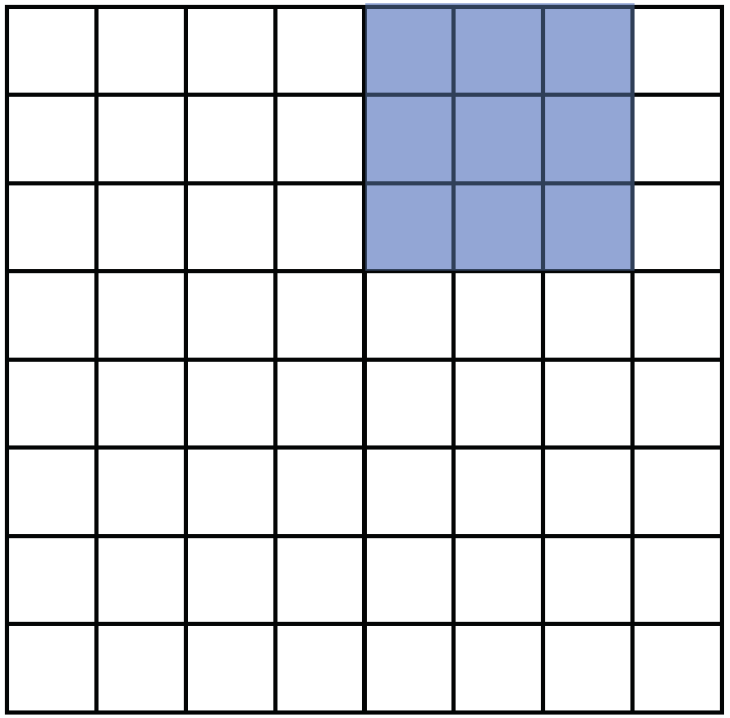
Input



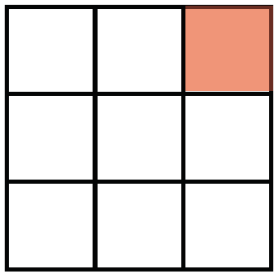
Output

Convolution: Stride

But we can also convolve with a **stride**, e.g. stride = 2



Input



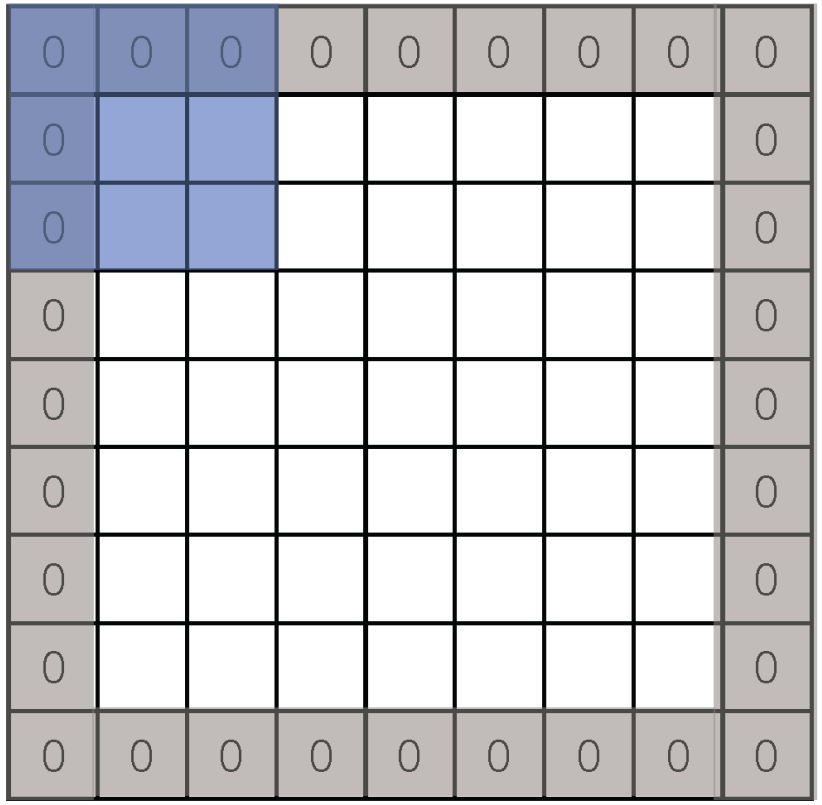
Output

- Notice that with certain strides, we may not be able to cover all of the input

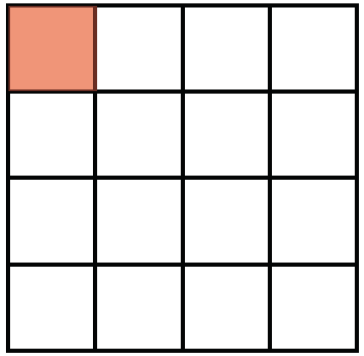
- The output is also half the size of the input

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



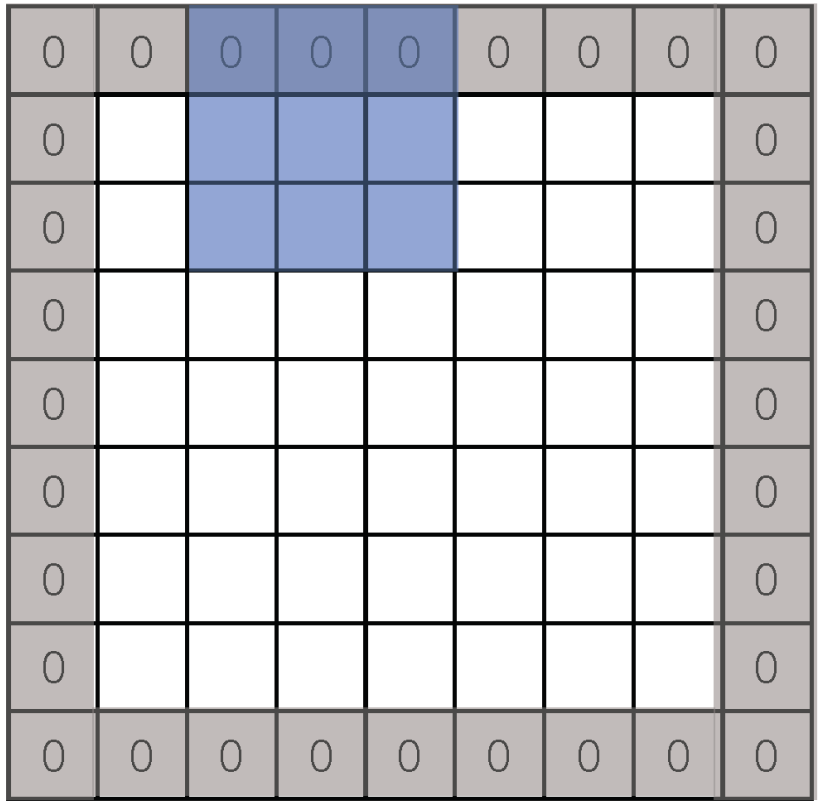
Input



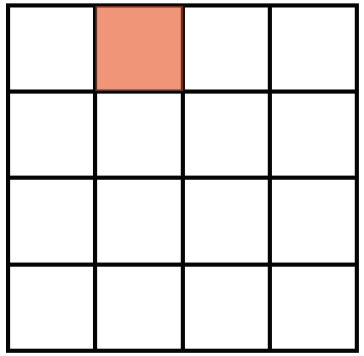
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



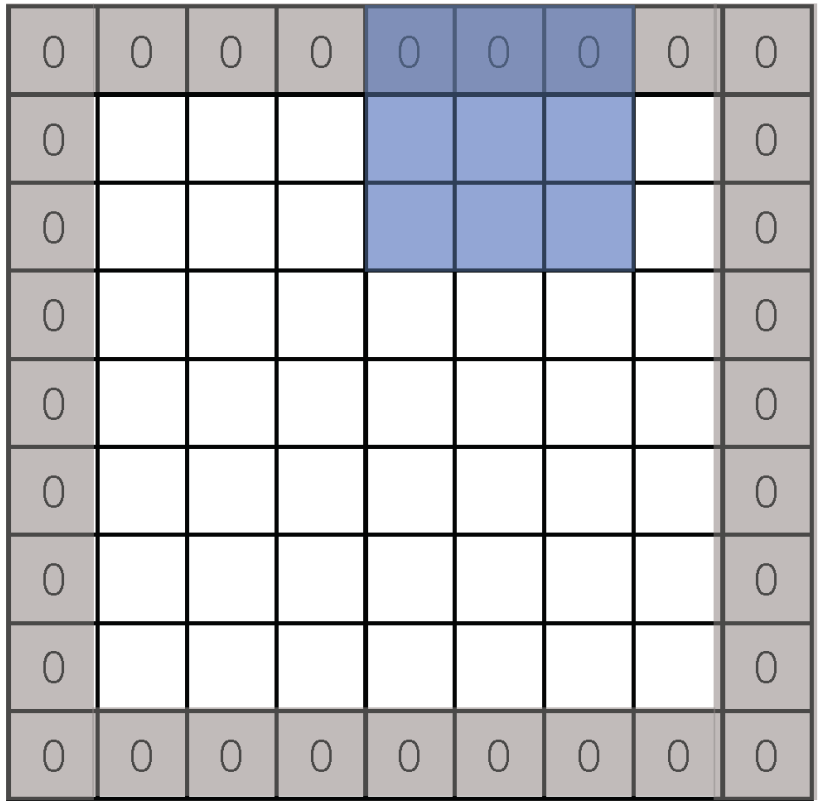
Input



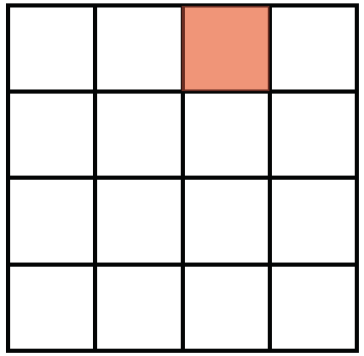
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



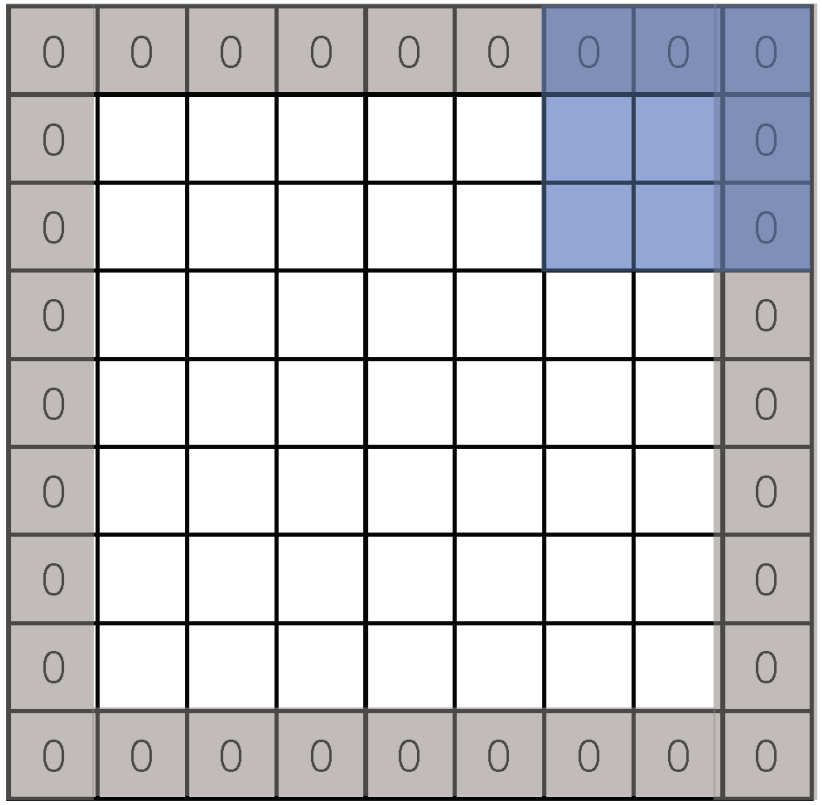
Input



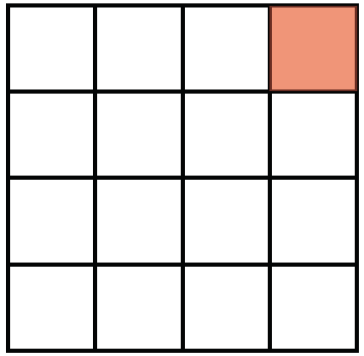
Output

Convolution: Padding

We can also pad the input with zeros.
Here, **pad = 1, stride = 2**



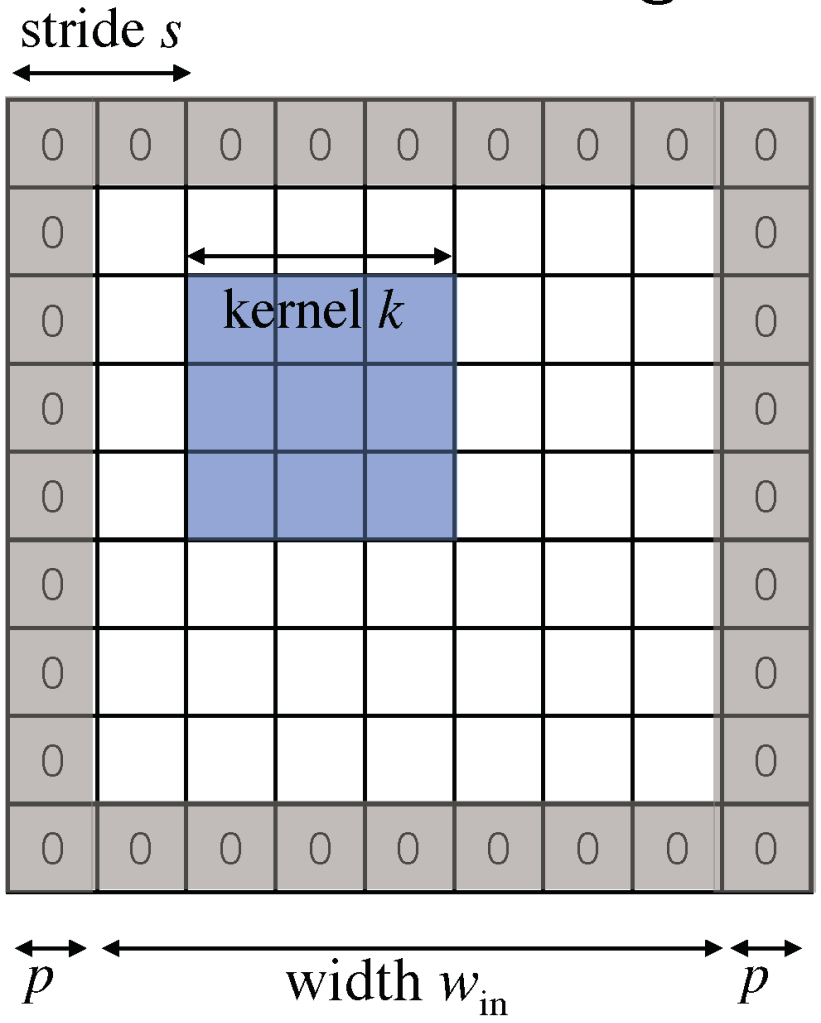
Input



Output

Convolution:

How big is the output?

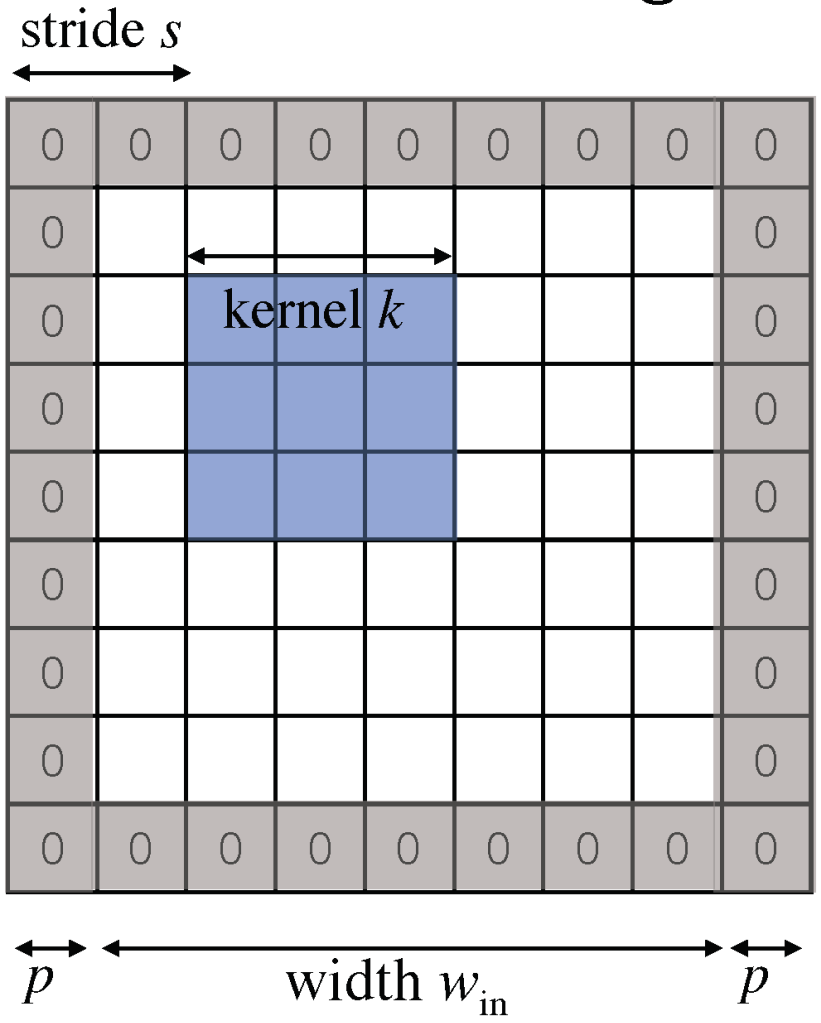


In general, the output has size:

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

Convolution:

How big is the output?



Example: k=3, s=1, p=1

$$w_{out} = \left\lfloor \frac{w_{in} + 2p - k}{s} \right\rfloor + 1$$

$$= \left\lfloor \frac{w_{in} + 2 - 3}{1} \right\rfloor + 1$$

$$= w_{in}$$

VGGNet [Simonyan 2014] uses filters of this shape

Pooling

For most ConvNets, **convolution** is often followed by **pooling**:

- Creates a smaller representation while retaining the most important information
- The “max” operation is the most common
- Why might “avg” be a poor choice?

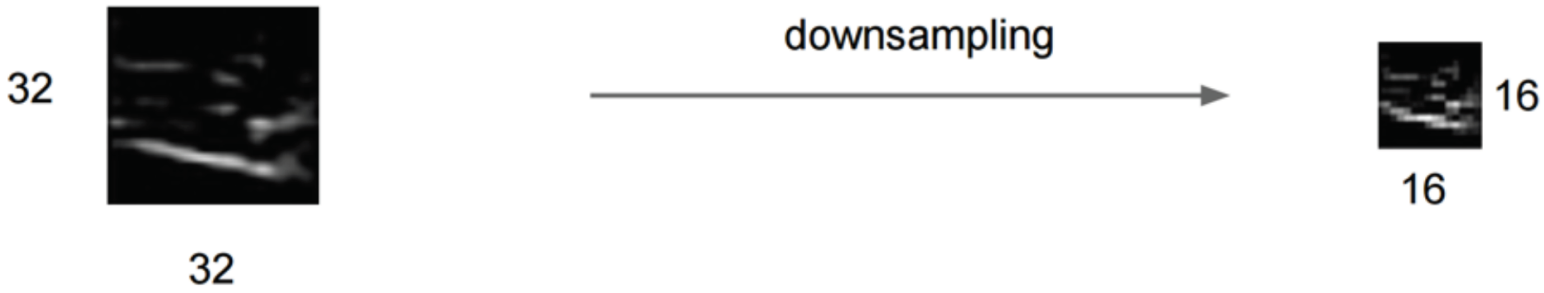
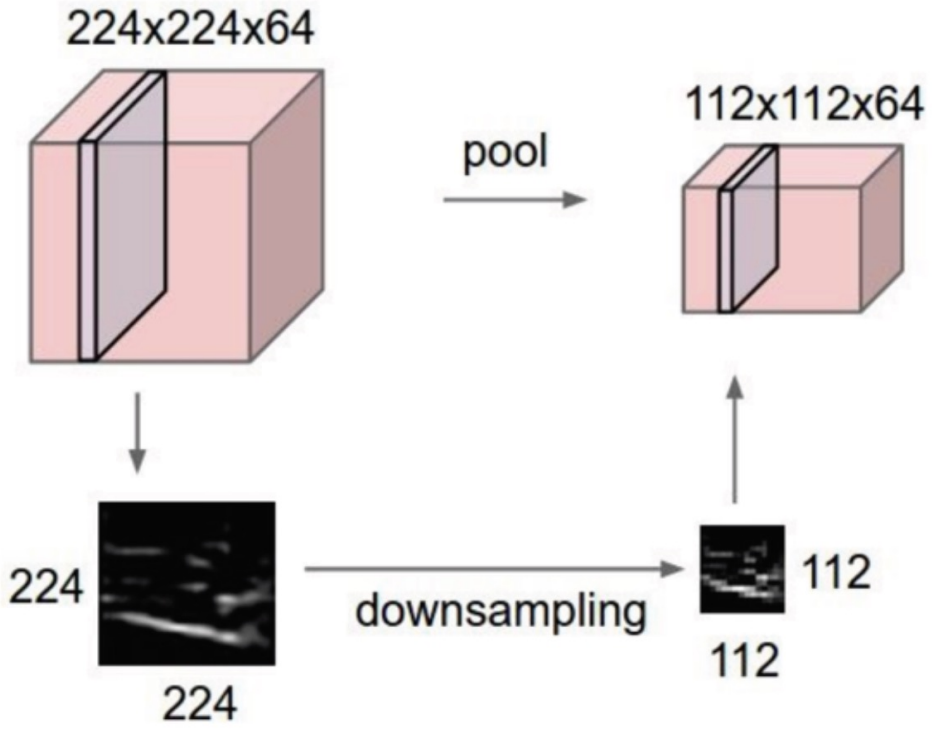


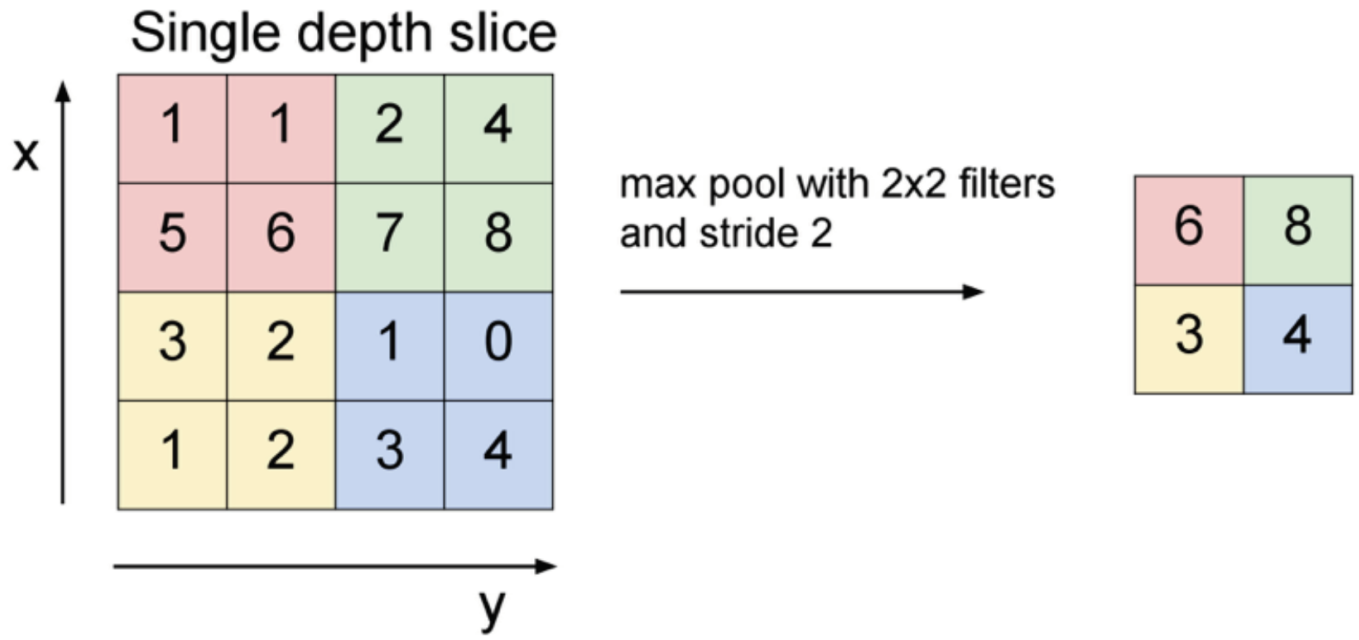
Figure: Andrej Karpathy

Pooling

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



Max Pooling

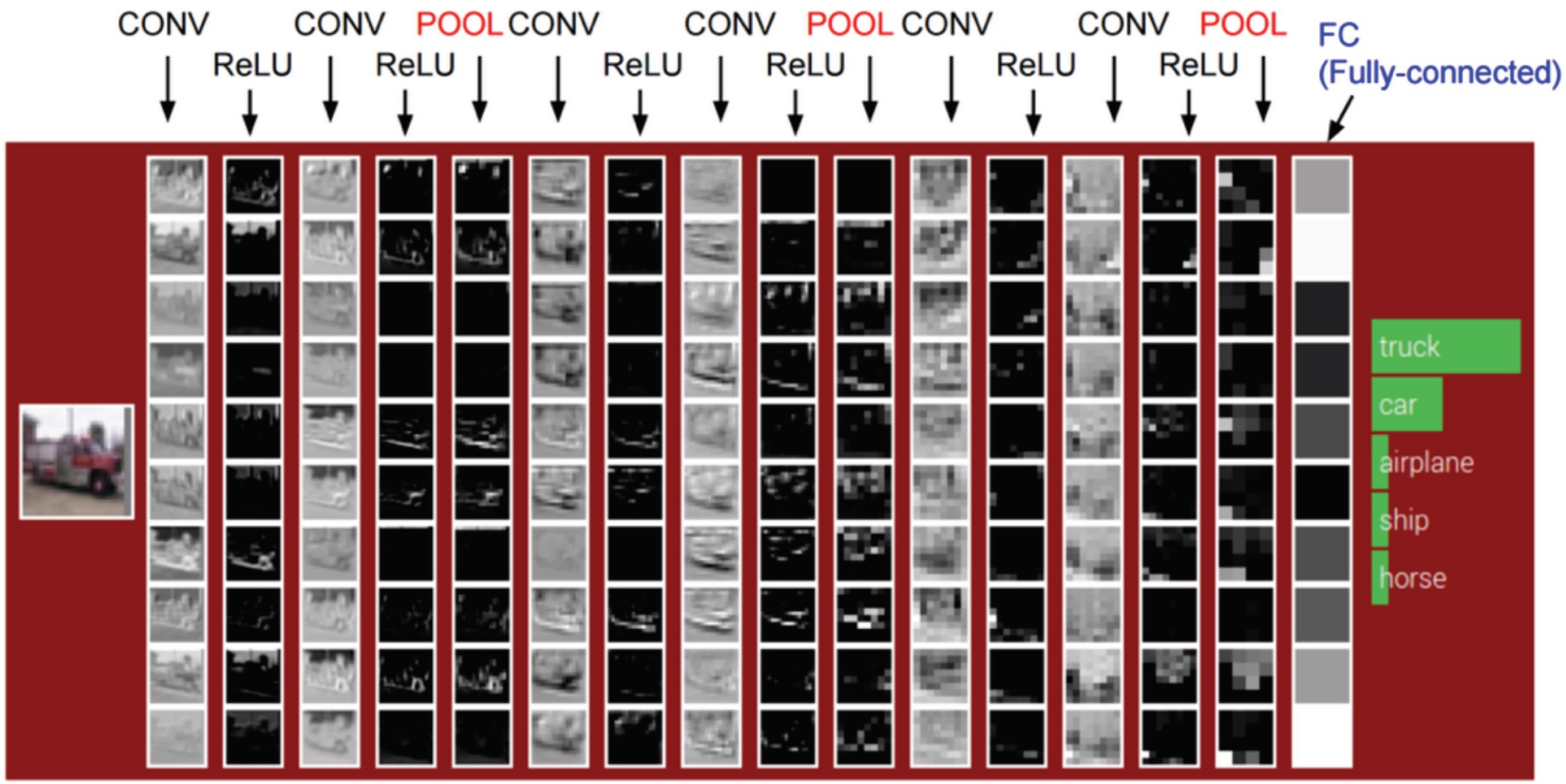


What's the backprop rule for max pooling?

- In the forward pass, store the index that took the max
- The backprop gradient is the input gradient at that index

Figure: Andrej Karpathy

Example ConvNet

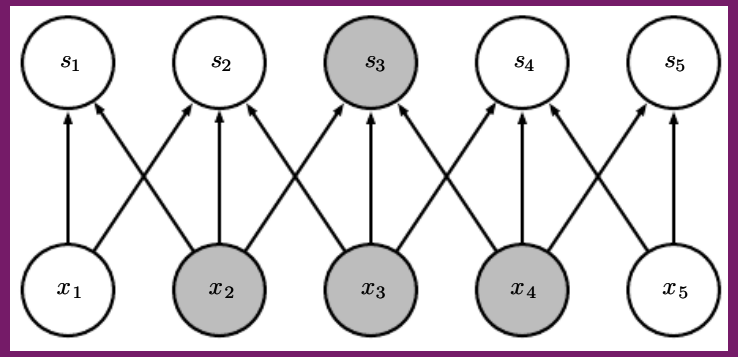


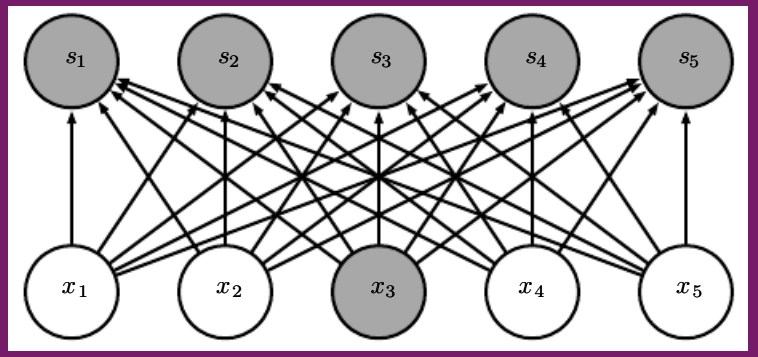
10x3x3 conv filters, stride 1, pad 1

2x2 pool filters, stride 2

Figure: Andrej Karpathy

Quiz





Think back about fully connected layers and convolutional layers, which statement is true?

1) FC layers are a special case of convolutional layers
2) Conv layers are a special case of fully-connected layers

"special case" == it can be recovered by corresponding settings

ConvNet Case Study I: Alexnet

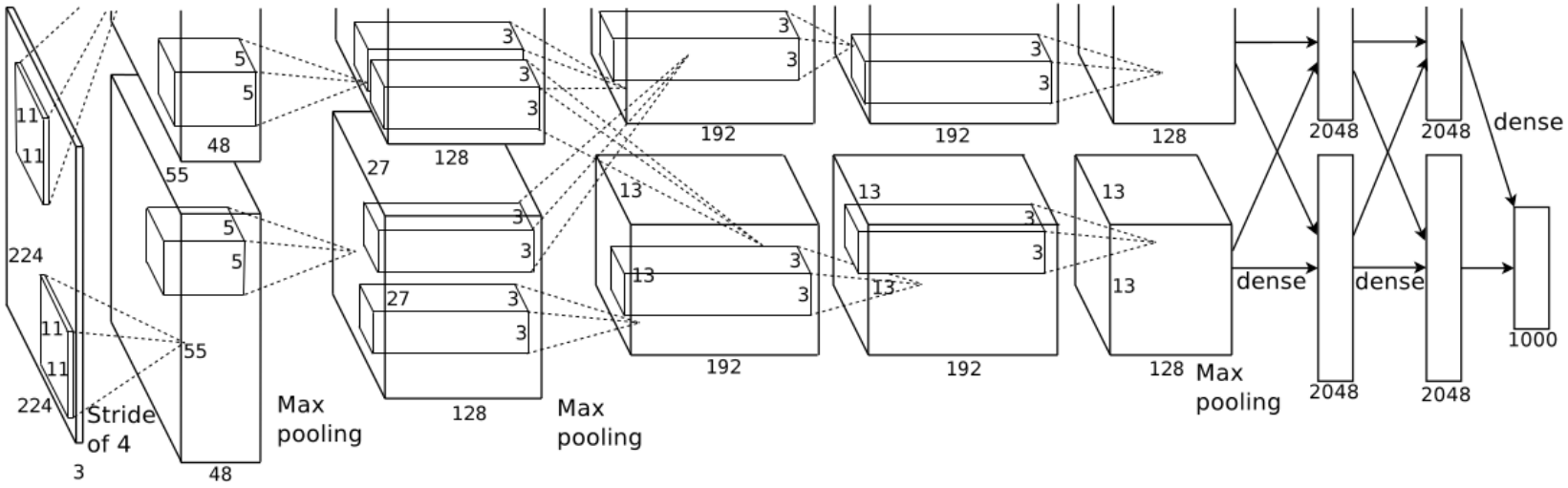
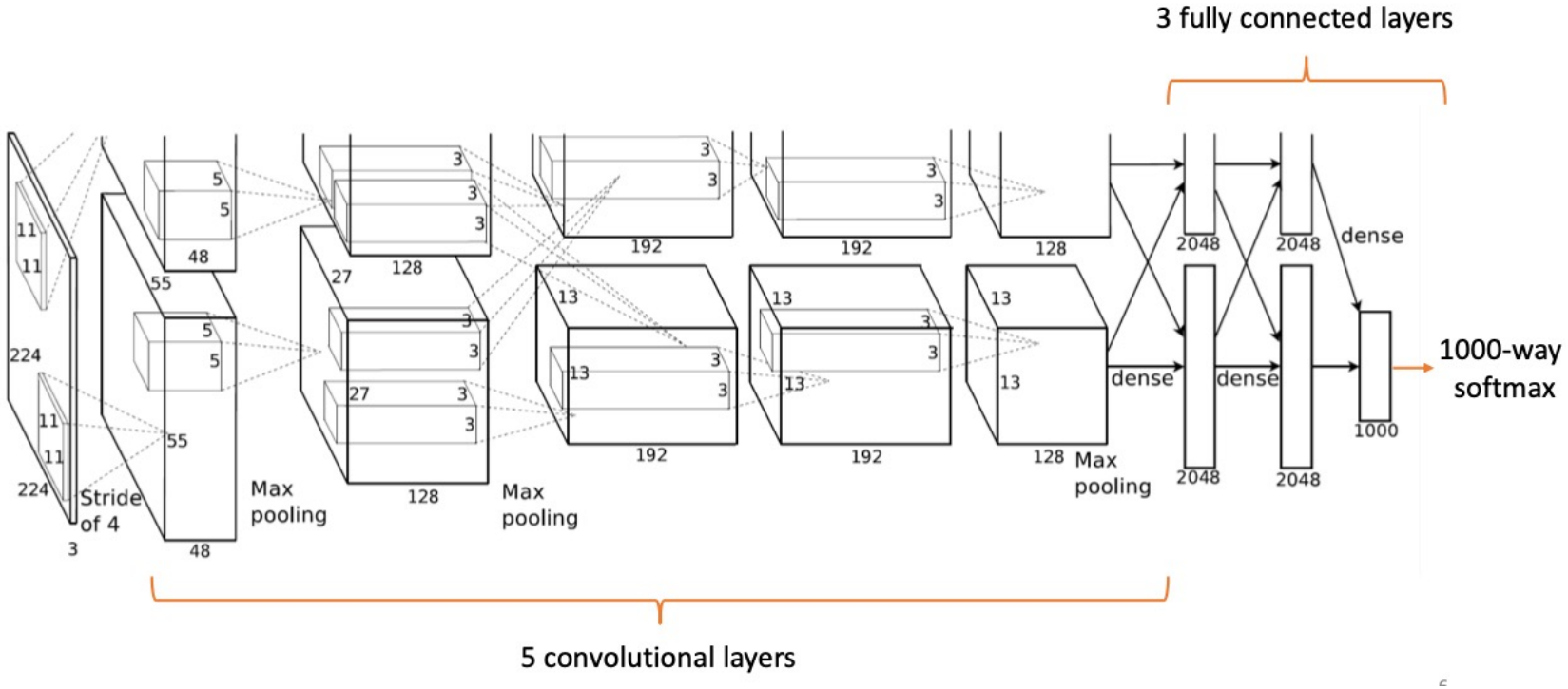


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet

- The architecture consists of eight layers:
  - five convolutional layers
  - three fully-connected layers



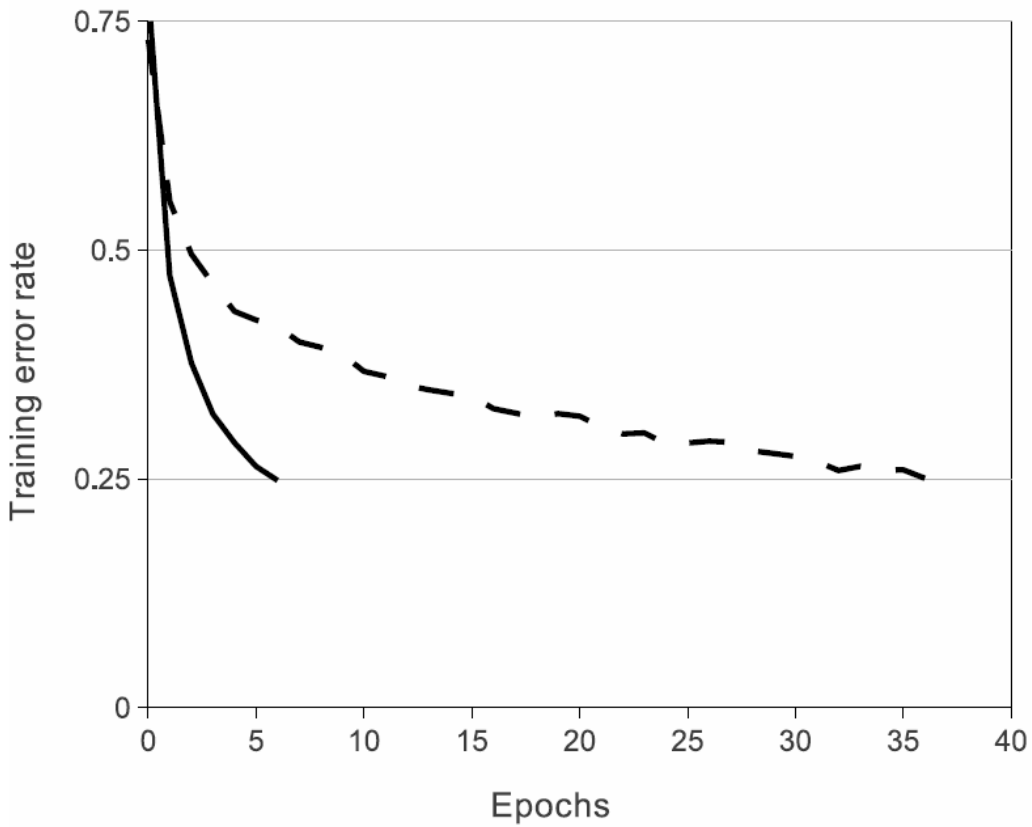
AlexNet

- ReLU Nonlinearity
  - replace the tanh function
- Multiple GPUs
- Local Response Normalization
- Overlapping Pooling
  - a reduction in error by about 0.5%
  - harder to overfit

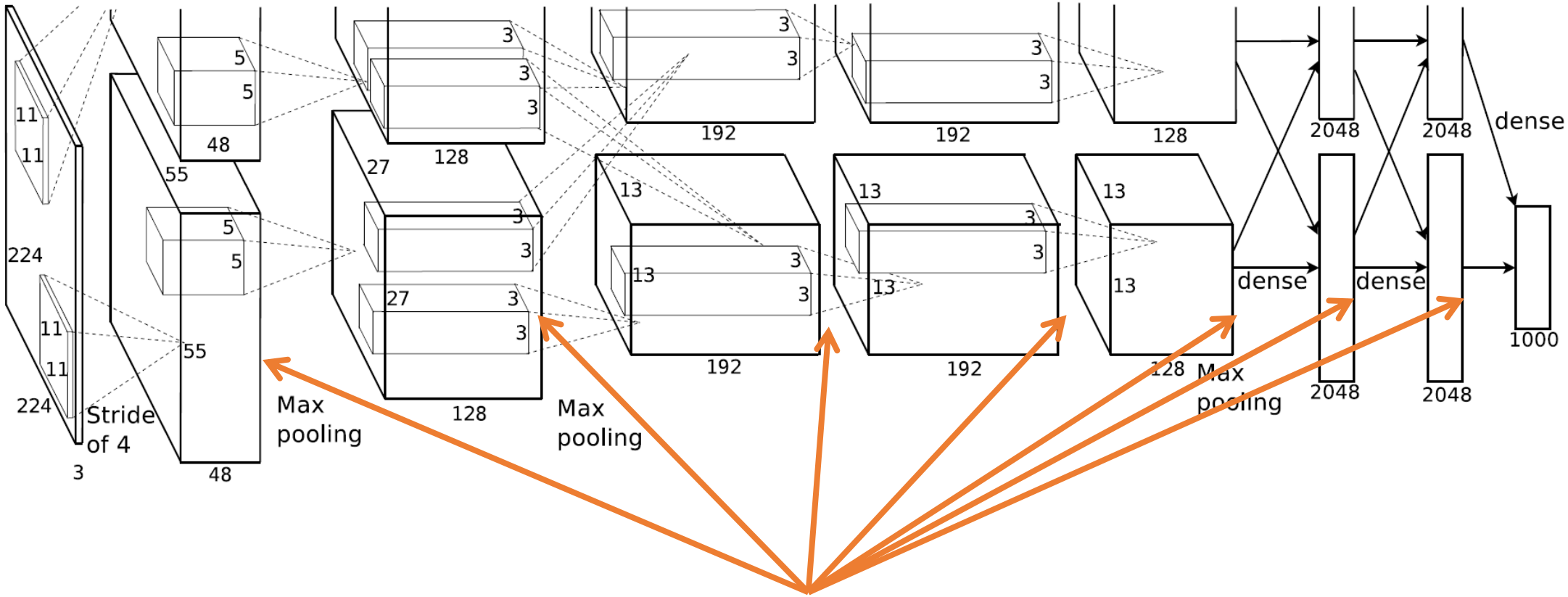
Activation function

- Traditionally, saturating nonlinearities:
  - hyperbolic tangent function
  - sigmoid function
  - slow to train
- ReLU Nonlinearity
  - Non-saturating
  - quick to train

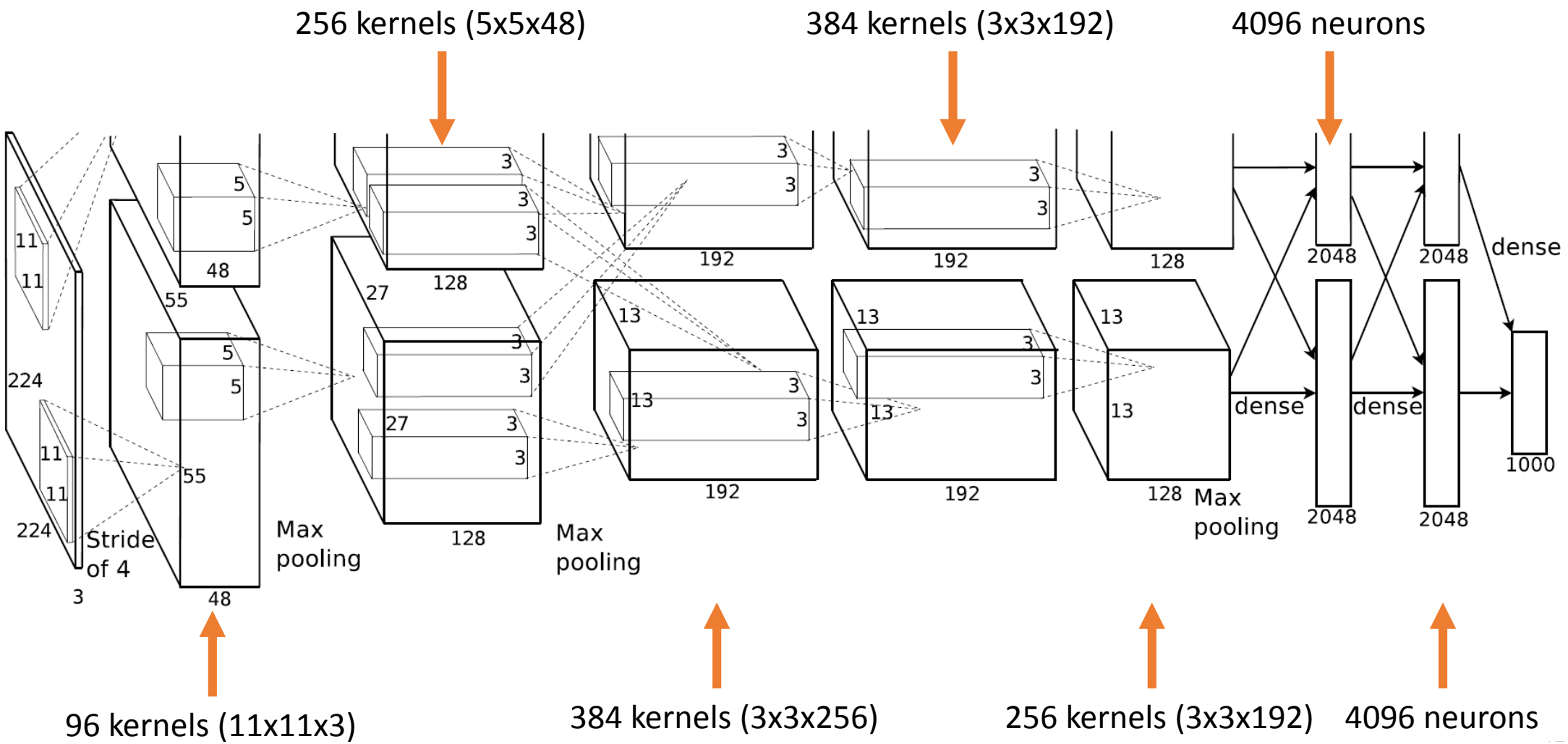
ReLUs (solid line) vs. tanh (dashed line)



Activation function

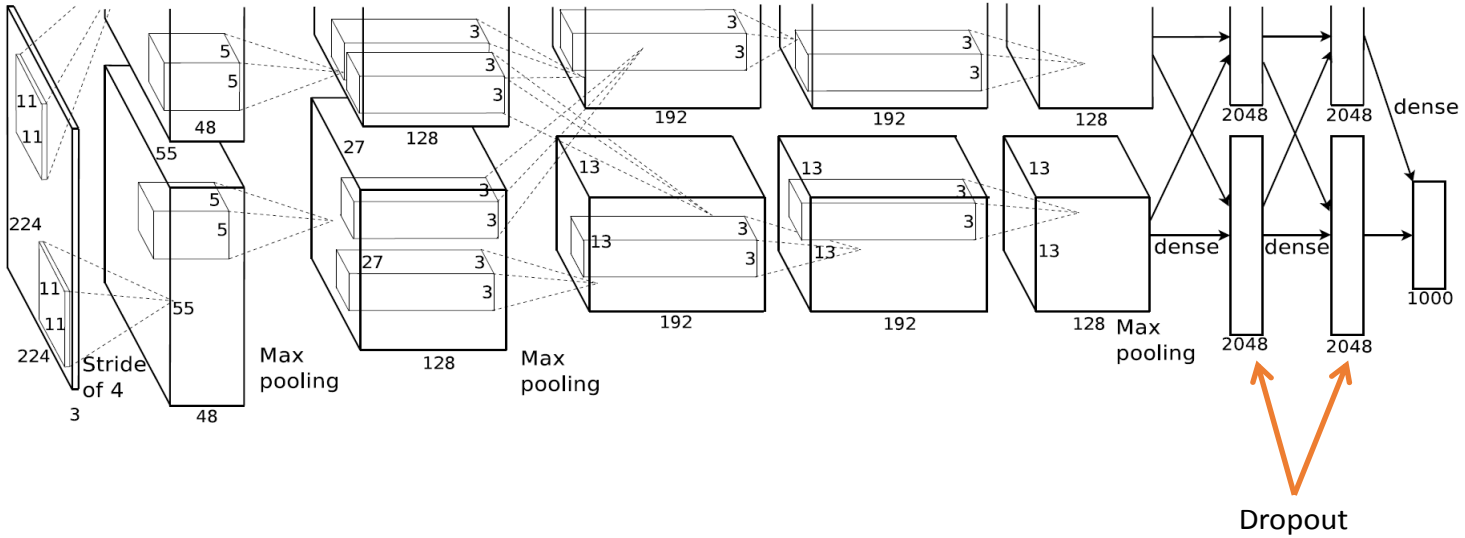


Overall architecture

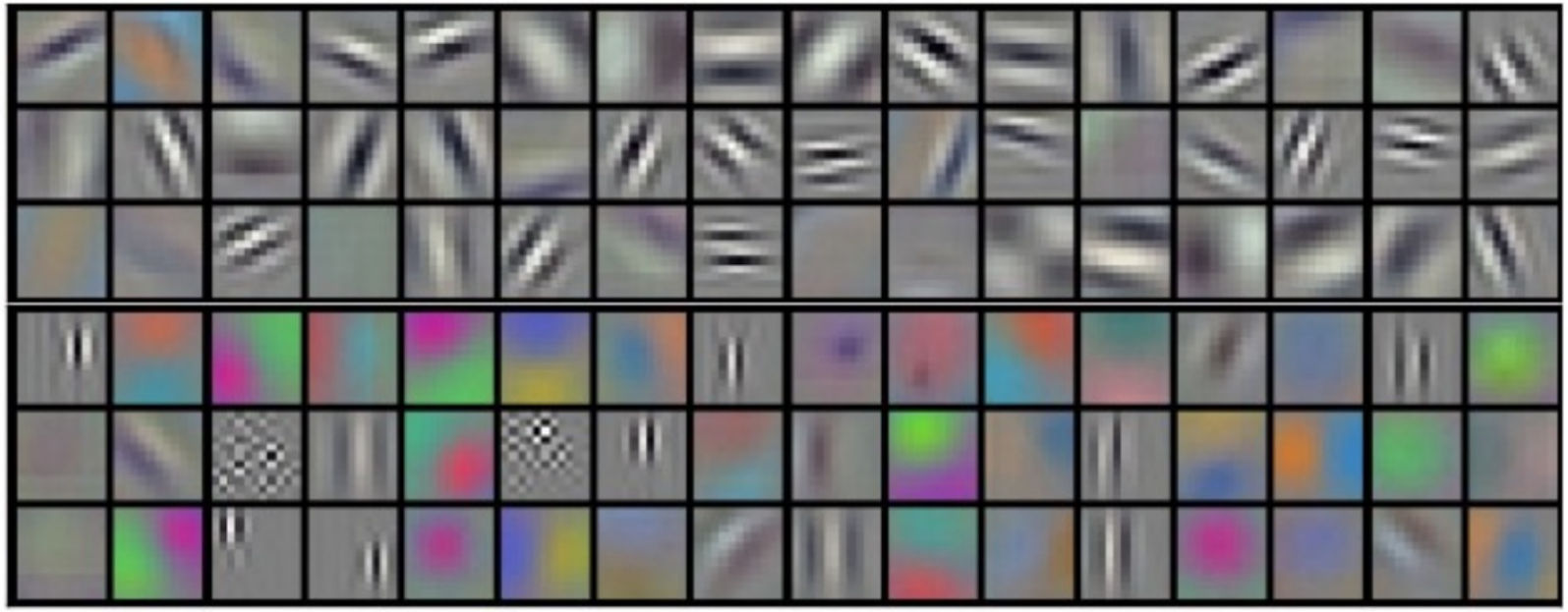


The Overfitting Problem

- A total of 60 million parameters (careful, there's two main "versions" of AlexNet)
- Data Augmentation
  - translations and horizontal reflections
  - change the intensity of RGB channels
- Dropout
  - more robust features
  - increases the training time



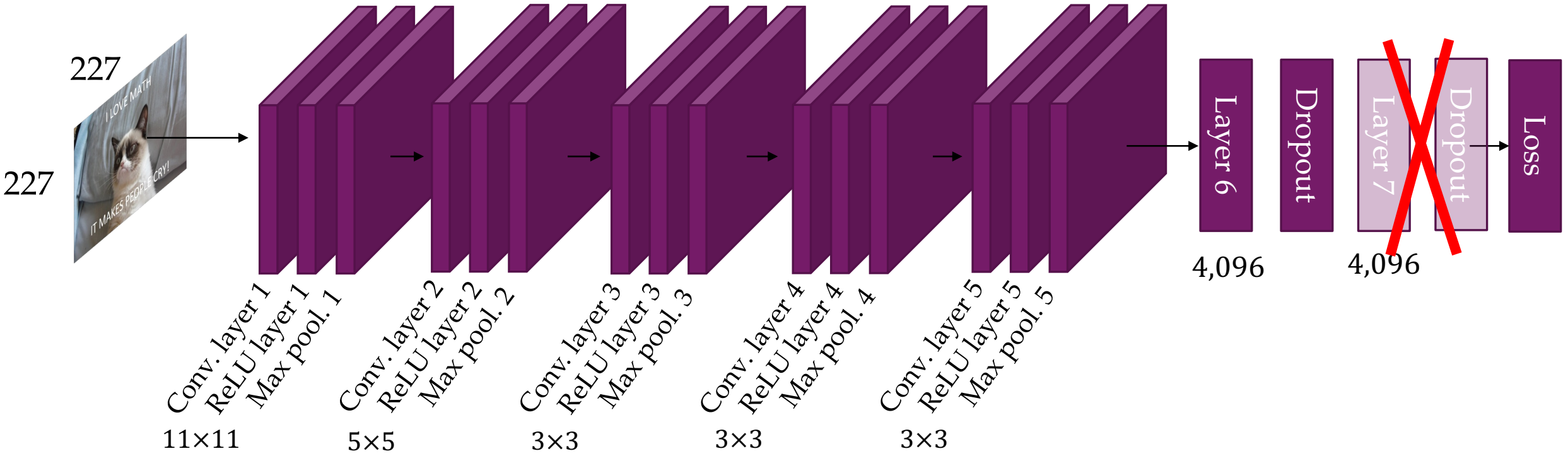
The learned filters



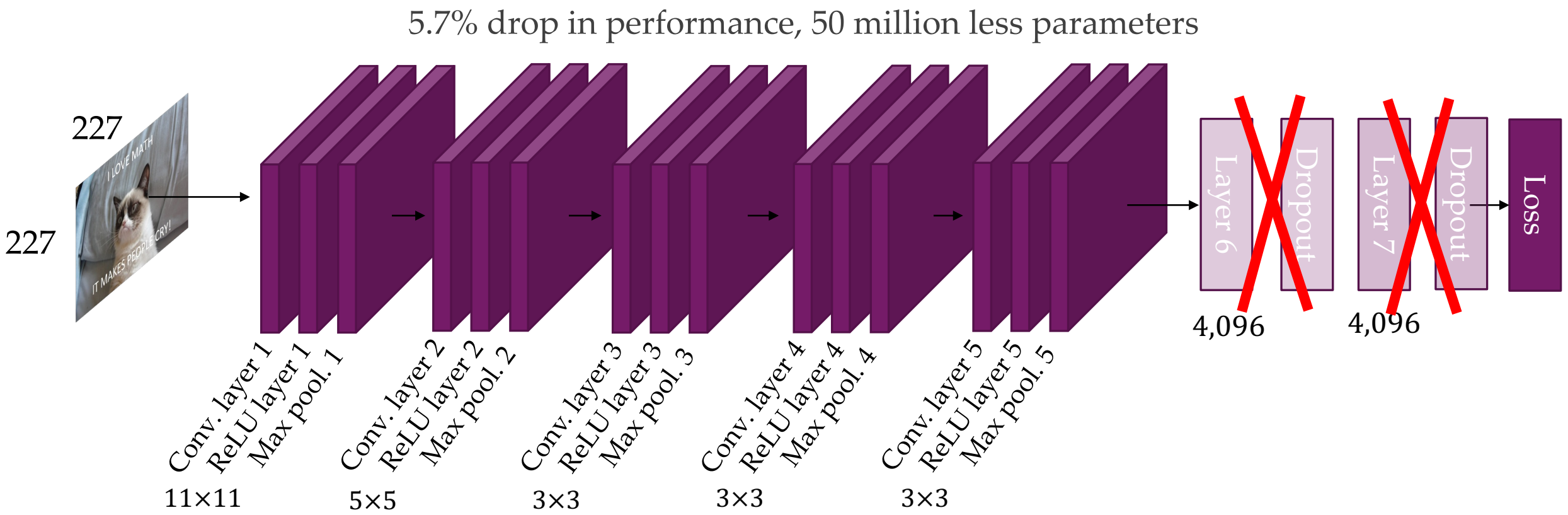
Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Removing layer 7

1.1% drop in performance, 16 million less parameters

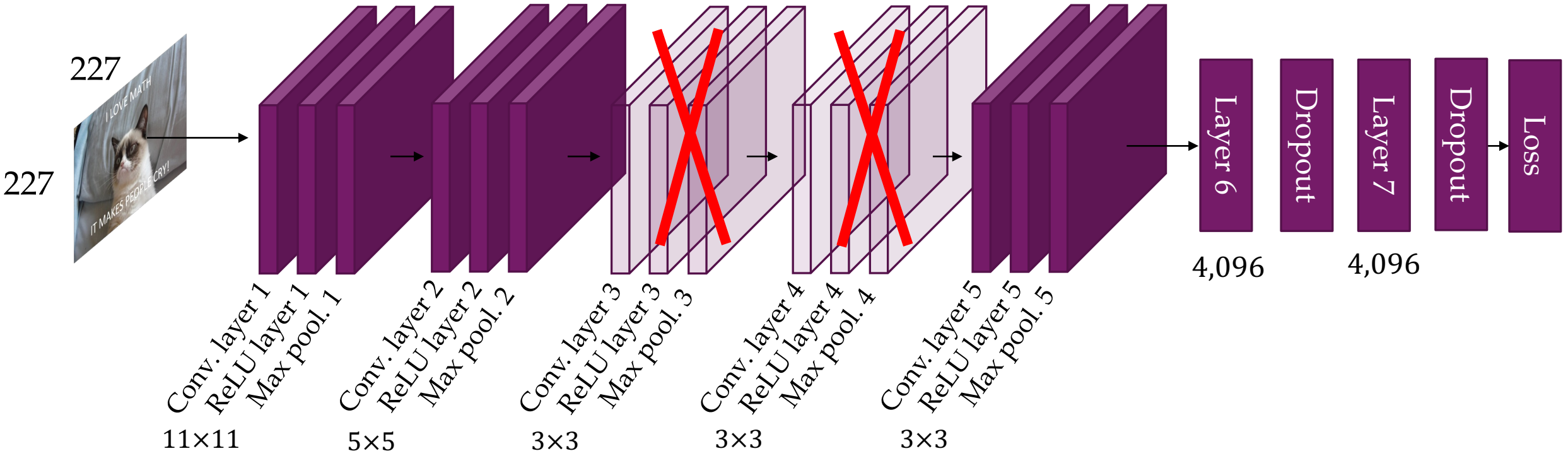


Removing layer 6, 7



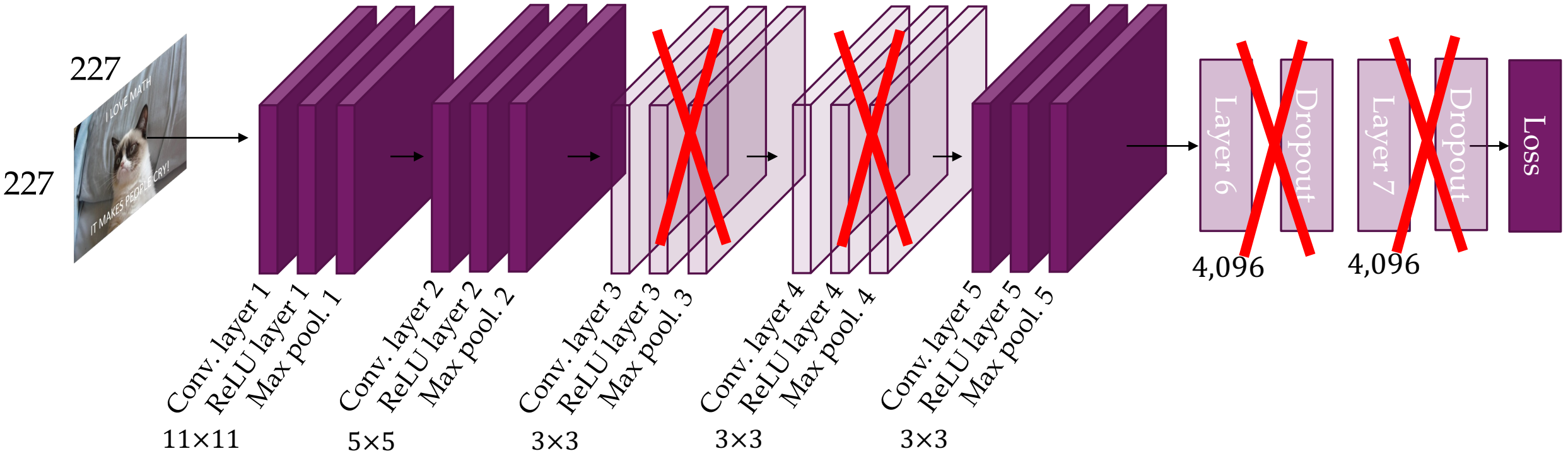
Removing layer 3, 4

3.0% drop in performance, 1 million less parameters. Why?



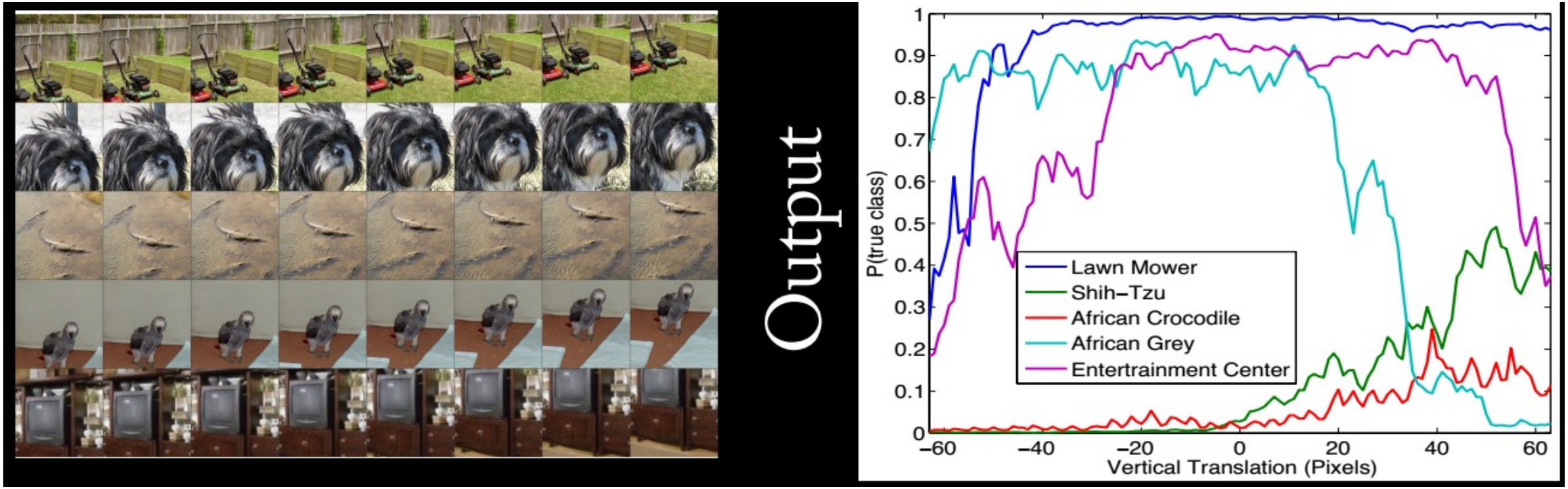
Removing layer 3, 4, 6, 7

33.5% drop in performance. Depth is crucial.



Translation invariance

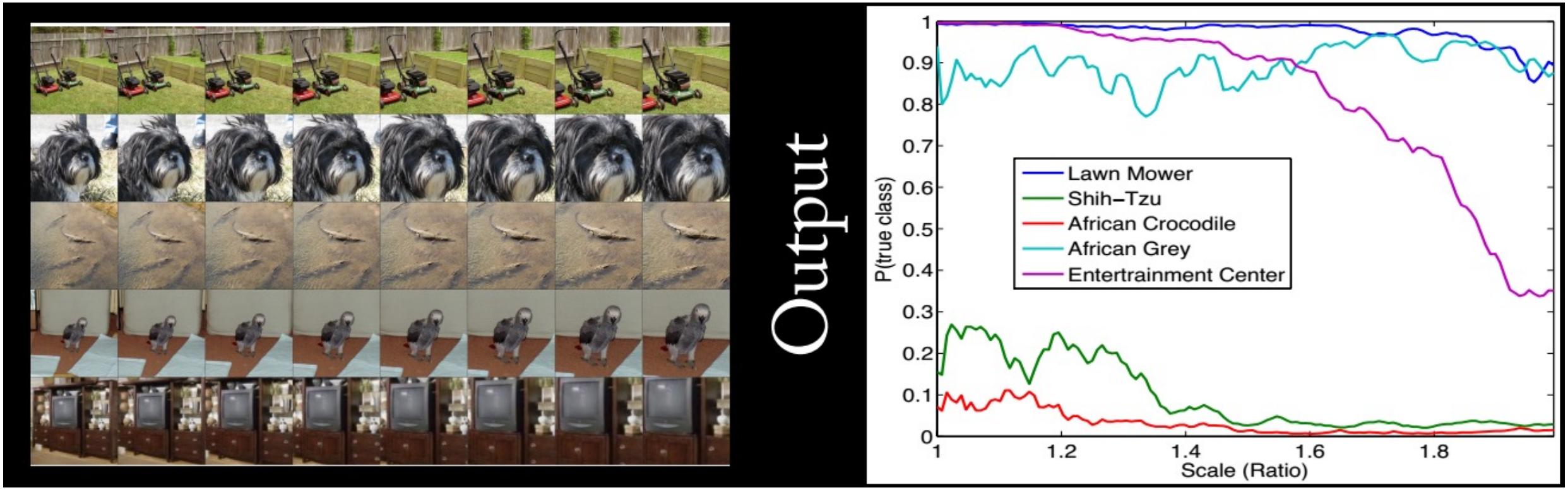
- CNNs are more or less translation invariant



Credit: R. Fergus slides

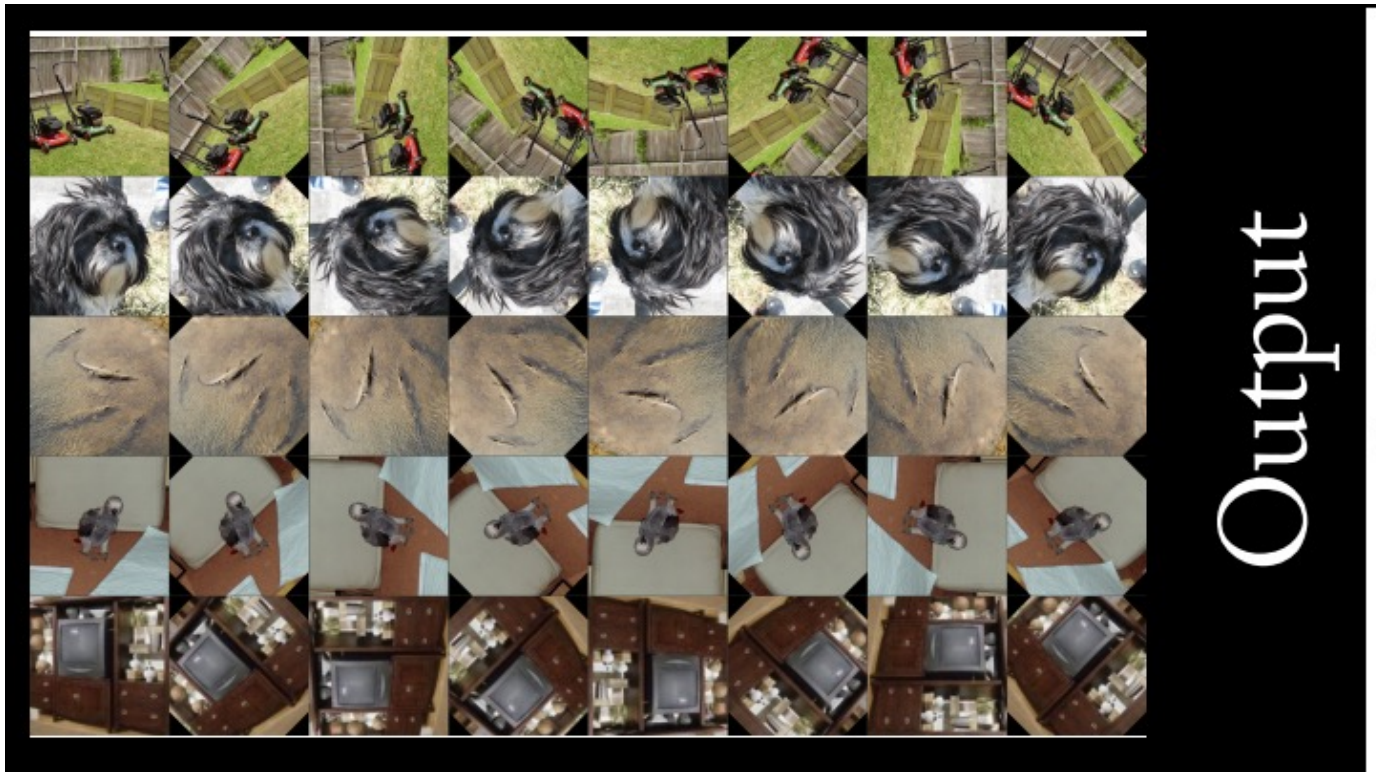
Scale invariance

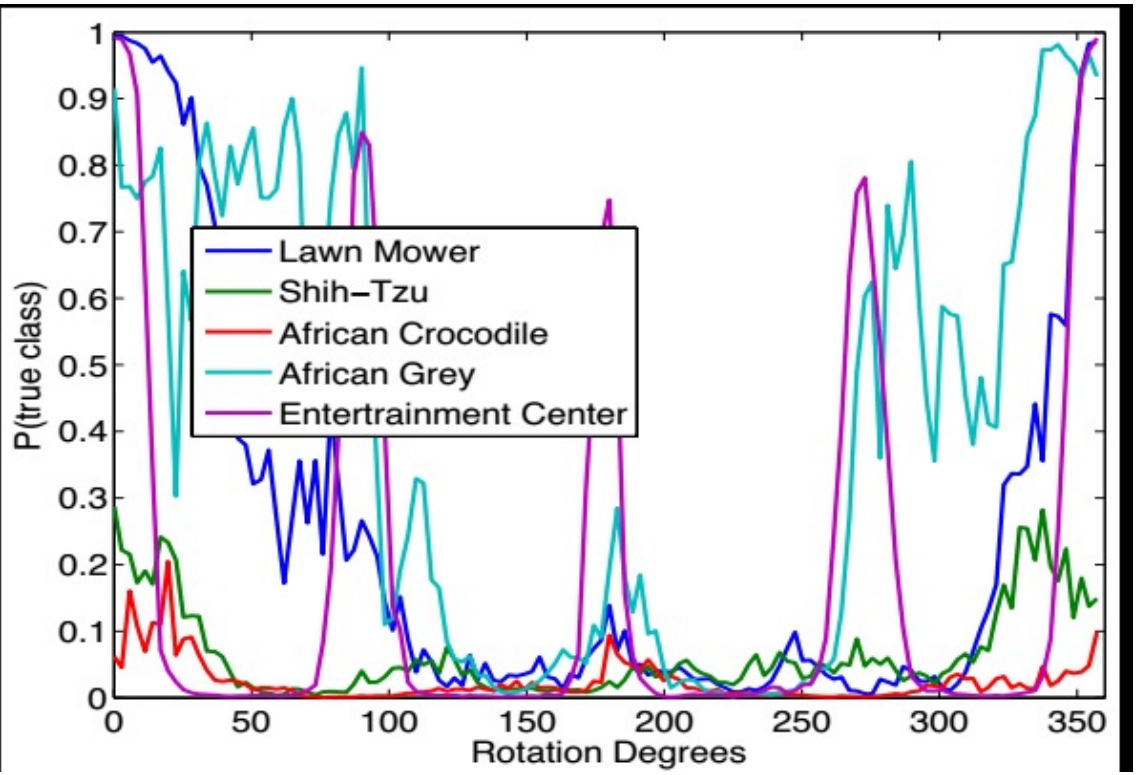
- CNNs are scale invariant to some degree
  - The standard convolutional filters are not scale invariant
  - Scale invariance learnt depends on scale variations present in data



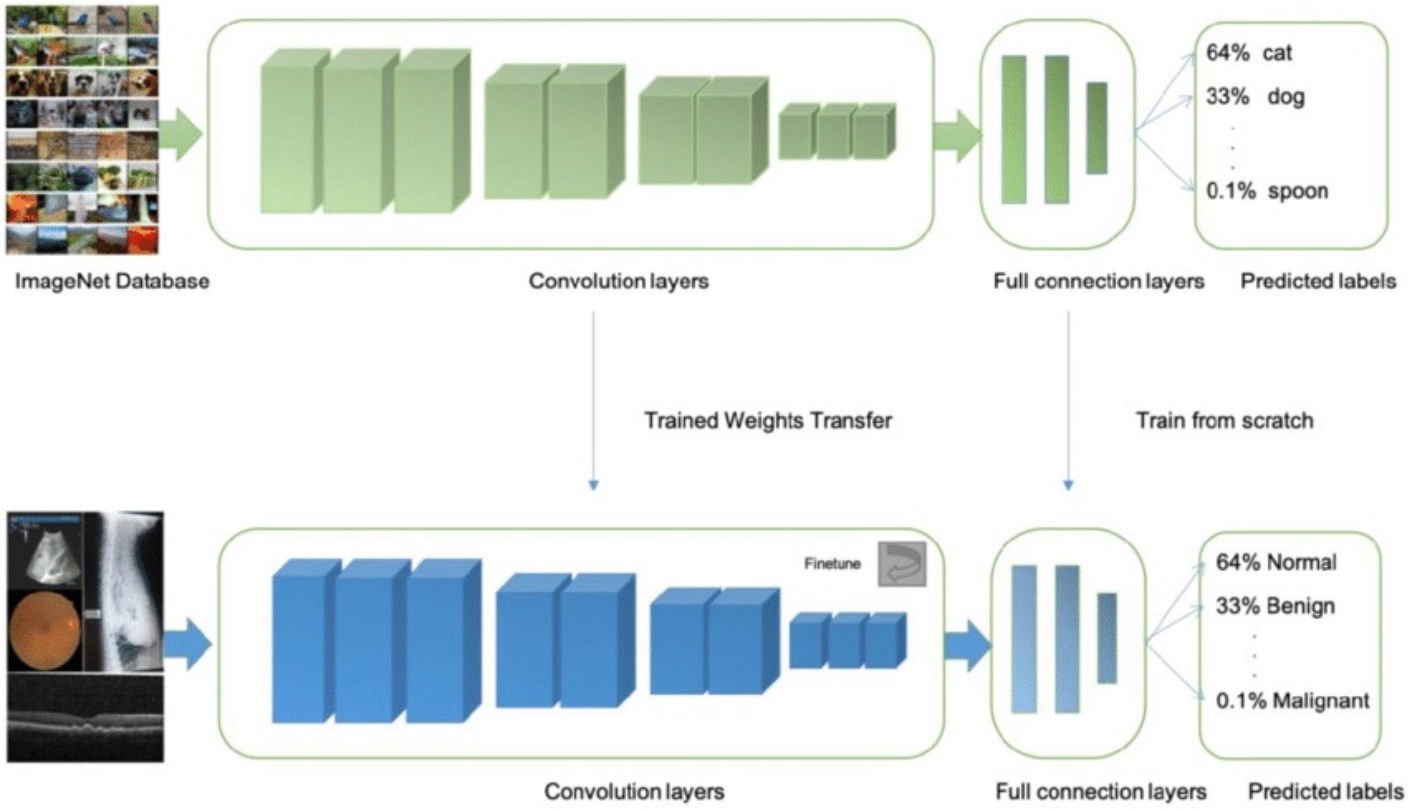
Rotation invariance

- CNNs are not rotation invariant
  - The standard convolutional filters not rotation invariant
  - And only few rotated examples in the training set. Augmentation can help





Transfer learning



Transfer learning: carry benefits from large dataset to the small one!

- Assume two datasets, T and S
- Dataset S is
  - fully annotated, plenty of images
  - We can build a model h_S
- Dataset T is
  - Not as much annotated, or much fewer images
  - The annotations of T do not need to overlap with S
- We can use the model h_S to learn a better specialised h_T
- This is called transfer learning

Imagenet: 1million

h_S

"My dataset": 1,000

h_T

Why use Transfer Learning?

- A CNN* can have millions of parameters
- But our datasets are not always as large
- Could we still train a CNN without overfitting problems?

*Not only CNNs, also ViTs (see Lecture 7)

Table of all available classification weights

Accuracies are reported on ImageNet-1K using single crops:

Weight	Acc@1	Acc@5	Params
AlexNet_Weights.IMAGENET1K_V1	56.522	79.066	61.1M
ConvNeXt_Base_Weights.IMAGENET1K_V1	84.062	96.87	88.6M
ConvNeXt_Large_Weights.IMAGENET1K_V1	84.414	96.976	197.8M
ConvNeXt_Small_Weights.IMAGENET1K_V1	83.616	96.65	50.2M
ConvNeXt_Tiny_Weights.IMAGENET1K_V1	82.52	96.146	28.6M
DenseNet121_Weights.IMAGENET1K_V1	74.434	91.972	8.0M
DenseNet161_Weights.IMAGENET1K_V1	77.138	93.56	28.7M
DenseNet169_Weights.IMAGENET1K_V1	75.6	92.806	14.1M
DenseNet201_Weights.IMAGENET1K_V1	76.896	93.37	20.0M
EfficientNet_B0_Weights.IMAGENET1K_V1	77.692	93.532	5.3M
EfficientNet_B1_Weights.IMAGENET1K_V1	78.642	94.186	7.8M
EfficientNet_B1_Weights.IMAGENET1K_V2	79.838	94.934	7.8M
ViT_H_14_Weights.IMAGENET1K_SWAG_E2E_V1	88.552	98.694	633.5M
ViT_H_14_Weights.IMAGENET1K_SWAG_LINEAR_V1	85.708	97.73	632.0M
ViT_L_16_Weights.IMAGENET1K_V1	79.662	94.638	304.3M
ViT_L_16_Weights.IMAGENET1K_SWAG_E2E_V1	88.064	98.512	305.2M
ViT_L_16_Weights.IMAGENET1K_SWAG_LINEAR_V1	85.146	97.422	304.3M
ViT_L_32_Weights.IMAGENET1K_V1	76.972	93.07	306.5M
Wide_ResNet101_2_Weights.IMAGENET1K_V1	78.848	94.284	126.9M
Wide_ResNet101_2_Weights.IMAGENET1K_V2	82.51	96.02	126.9M
Wide_ResNet50_2_Weights.IMAGENET1K_V1	78.468	94.086	68.9M
Wide_ResNet50_2_Weights.IMAGENET1K_V2	81.602	95.758	68.9M

https://pytorch.org/vision/stable/models.html

Convnets are good in transfer learning

- Even if our dataset *T* is not large, we can train a CNN for it
- Pre-train a network on the dataset *S*
- Then, there are two main solutions
  - Fine-tuning
  - CNN as feature extractor