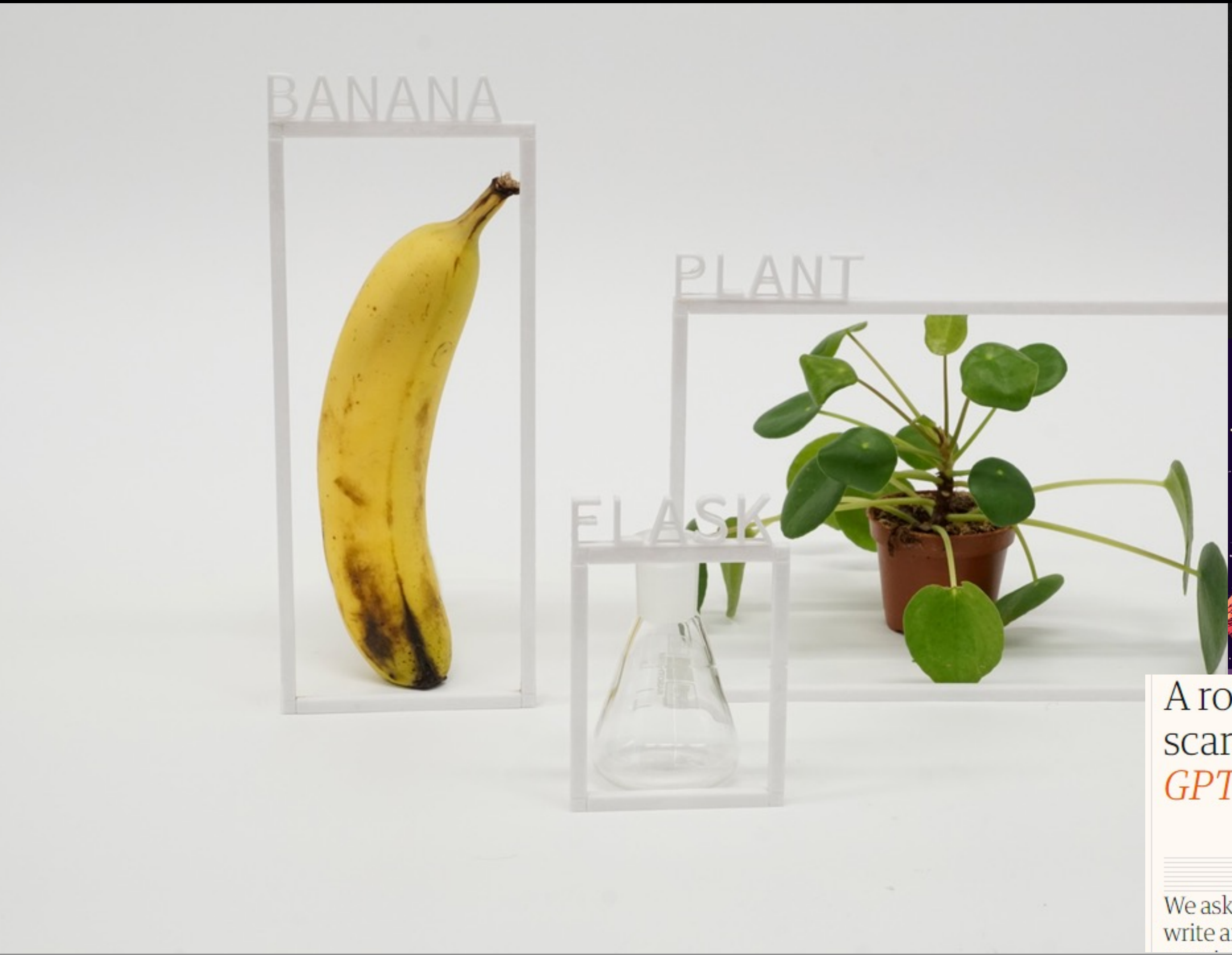
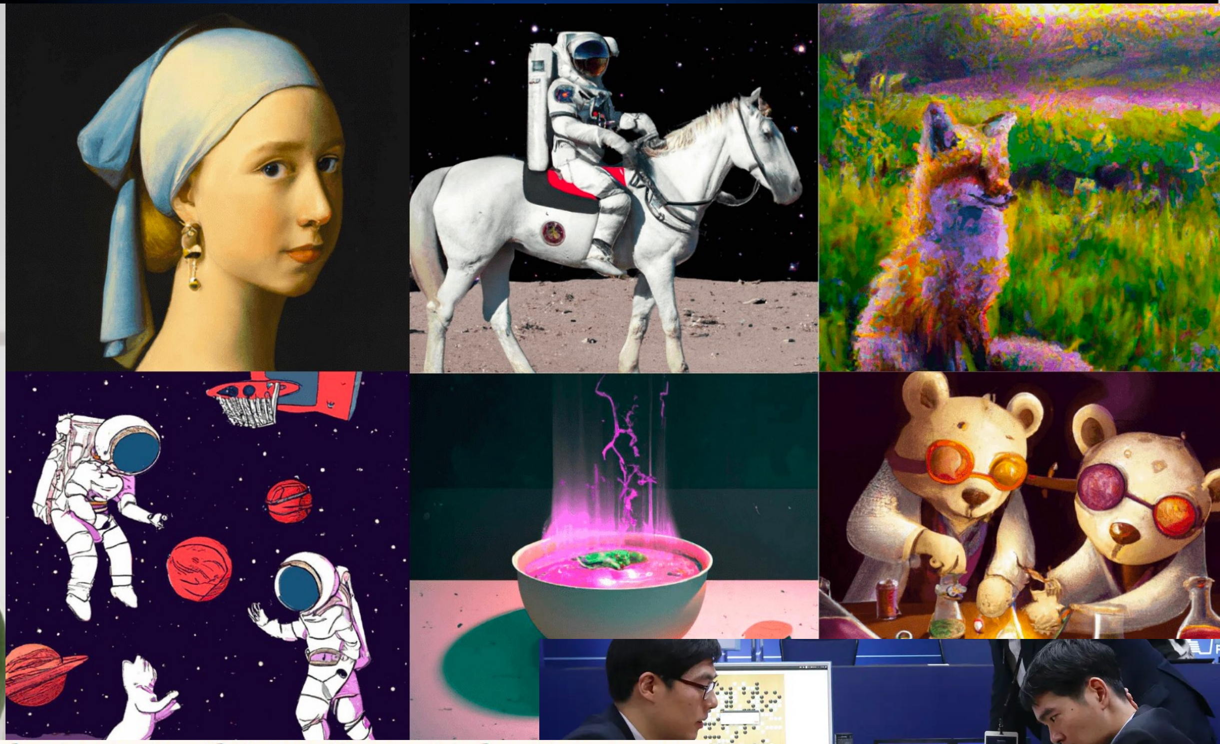
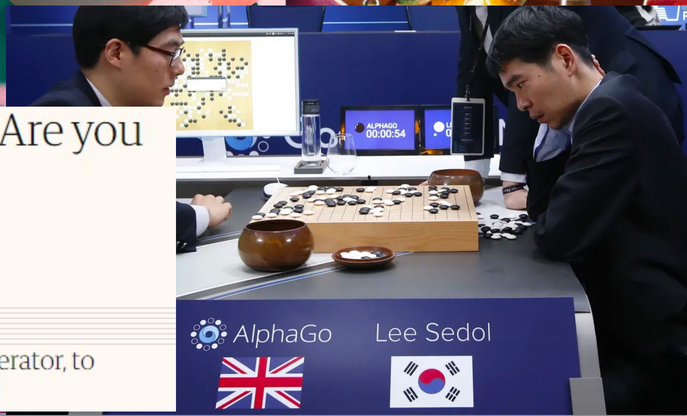







A robot wrote this entire article. Are you scared yet, human?
GPT-3

We asked GPT-3, OpenAI's powerful new language generator, to write an essay for us from scratch. The assignment? To

Lecture 5: Convolutional Neural Networks

Deep Learning 1 @ UvA
Yuki M. Asano

Optimizing neural networks

have model
- Architecture
- Weights initialisation

Today

compute loss
- Gradient Descent, dataloader
- Augmentations
- Regularisation

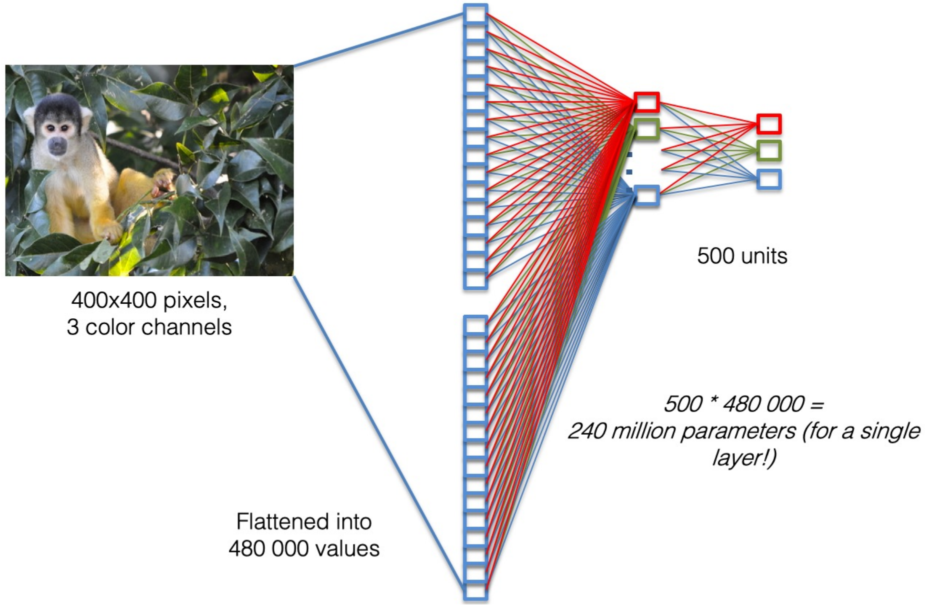
compute gradient
- Loss functions
- Backprop

update weights
- Gradient Descent
- Advanced Optimizers

Last times

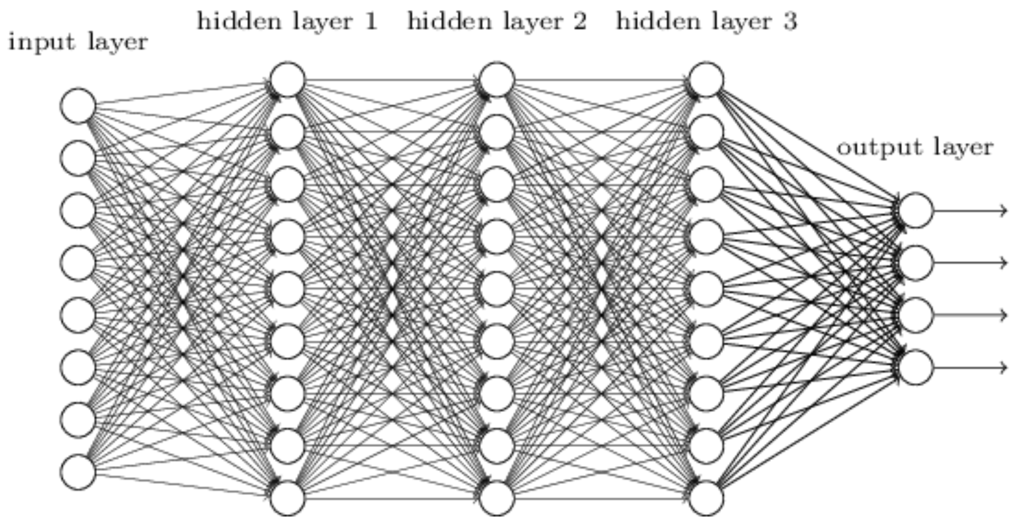
Multi-layer perceptrons (Recap)

- 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)

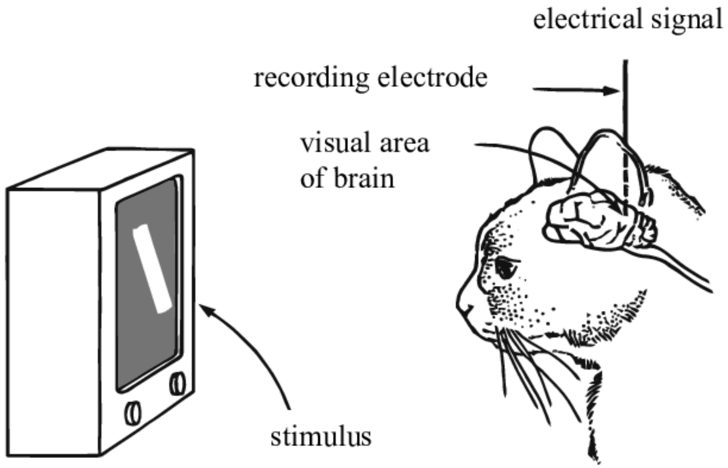
- From this fully-connected model, do we really need all the edges?
- Can some of these be "shared" (equal weight)?
- Can prior knowledge ("inductive biases") be incorporated into the design?



Consider an image

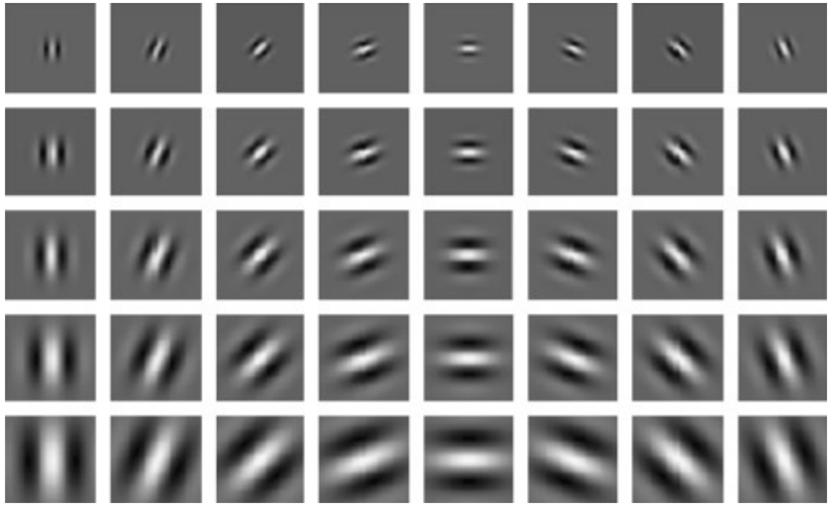
- Some patterns are much smaller than the whole image
- Can represent a small region with fewer parameters
- What about training a lot of such "small" detectors and each detector must be able to "move around"? (C.f. Barlow 1961)

Hubel and Wiesel: Nobel Prize for Physiology or Medicine in 1981



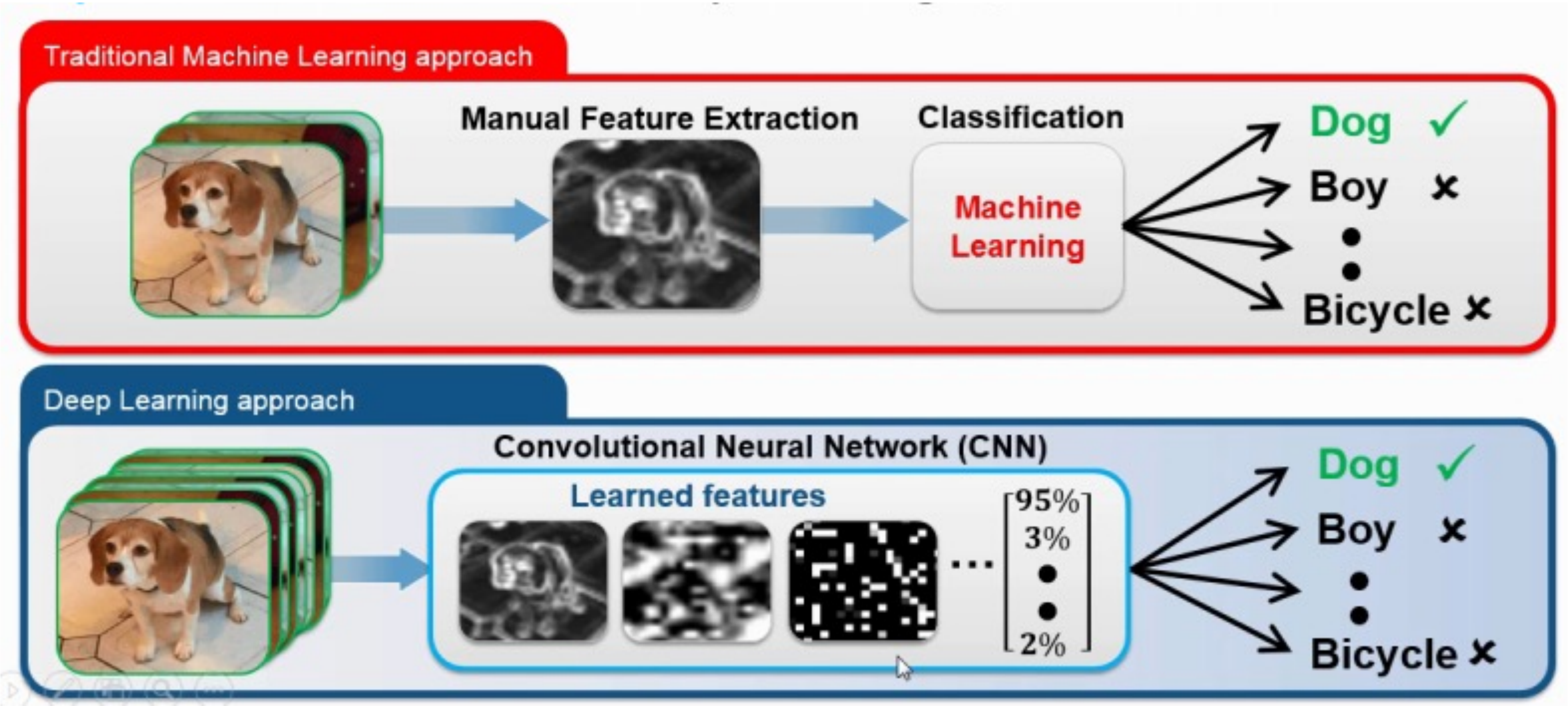
Filters, yes. How about learnable filters

- Hubel and Wiesel: Edge detectors in visual cortex
- See also “Grandmother cells” (next lecture)
- Handcrafted filters
  - Canny, Gabor filters
- Are they optimal for recognition?
- Can we learn optimal filters from our data instead?
- Are they going to resemble the handcrafted filters?



Gabor filters

Filters, yes. How about learnable filters



Manual features:
* Eg HOG, SIFT, GIST

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:

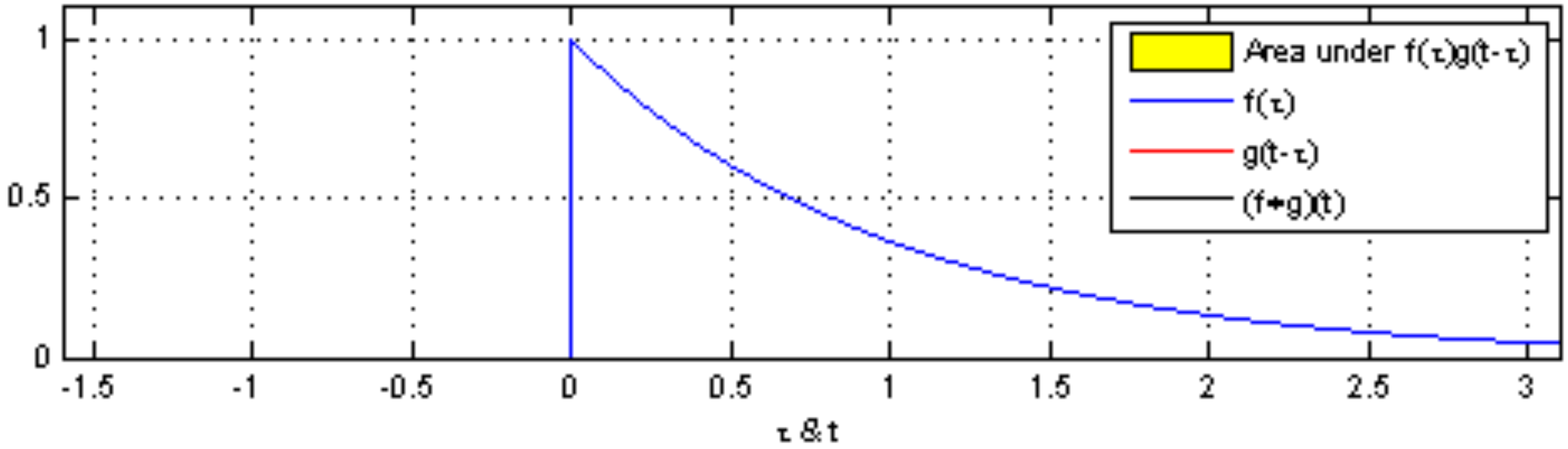
$$s(t) = \int x(a)w(t-a)da$$

- 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:

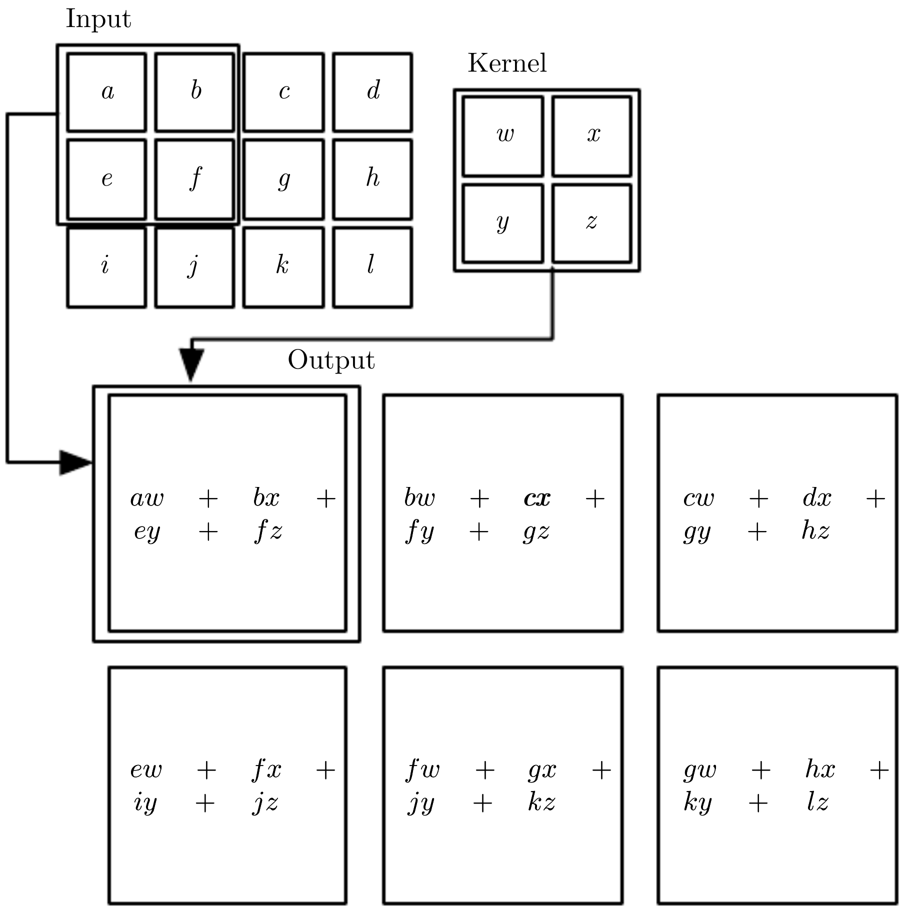
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

- 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

$$* \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix} =$$

- We can sharpen images

$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

Picture credits from Roger Grosse

Examples

- We can detect edges

$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

- Sobel filter

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} =$$

Picture credits from Roger Grosse

Quiz

What's the effect of running a normal-randomly initialized convolution on an image?
Hint: think about a 1x2 conv first.

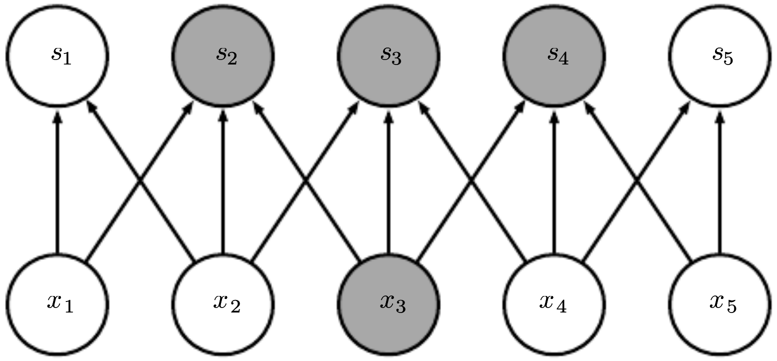
1) It will emphasize average-colored regions
2) It will emphasize edges
3) It will blur the image
4) It will sharpen the image

The motivation of convolutions

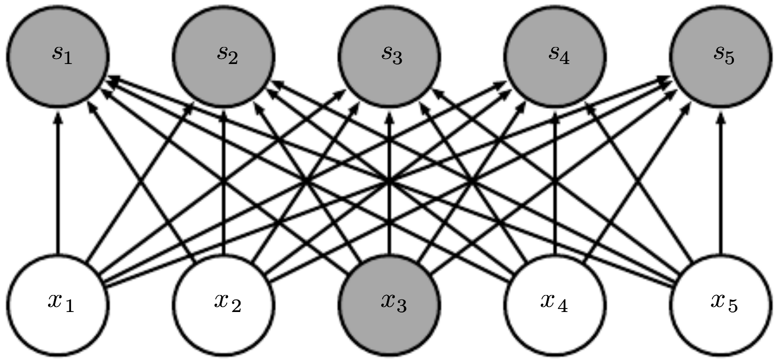
- Sparse interaction, or local connectivity
  - The receptive field of the neuron, or the filter size.
  - The connections are local in space (width and height), but always full in depth
  - A set of learnable filters
- Parameters sharing, the weights are tied
- Equivariant representation: same operation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



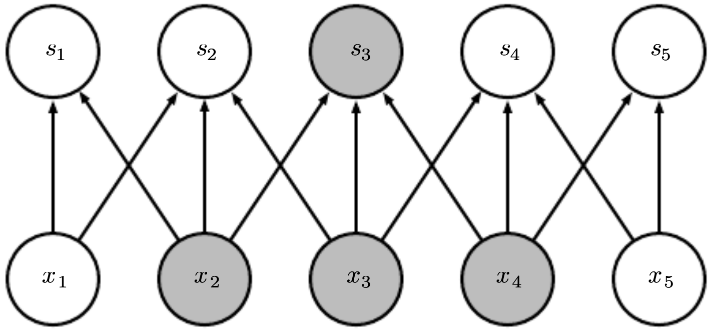
sparse connection



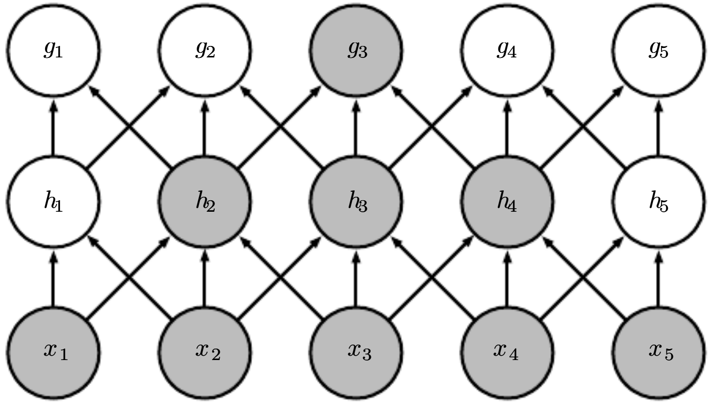
full connection

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.



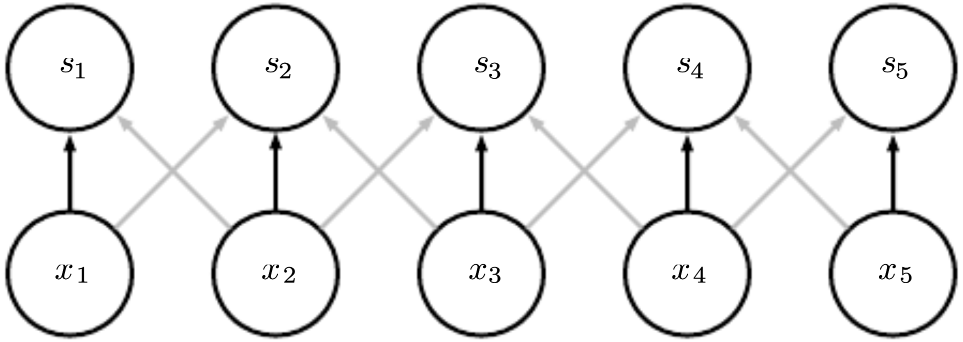
convolution with a kernel of width 3



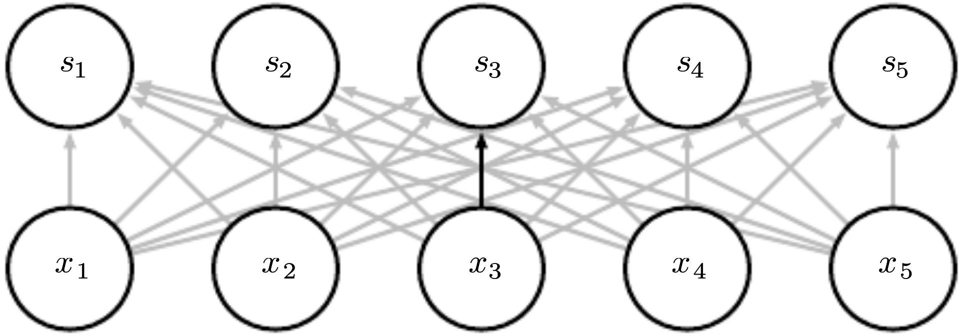
receptive field of the units in the deeper layers

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



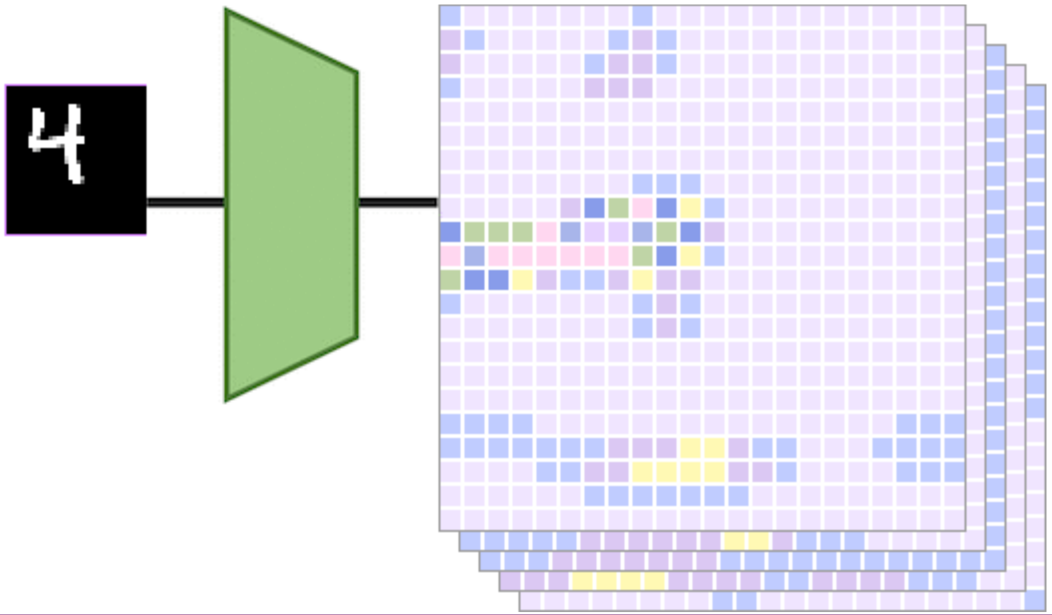
parameter sharing



no parameter sharing

The motivation of convolutions

- Equivariant representation
  - parameter sharing causes the layer to have a property called *equivariance* to translation.*
  - A function is equivariant if the input changes, the output changes in the same way.
  - why do we want equivariance to translation?



A simple convolution: saves space!

1D example

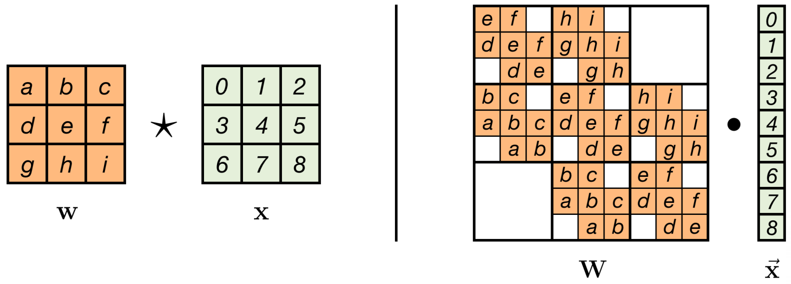
We can express convolution in terms of a matrix multiplication

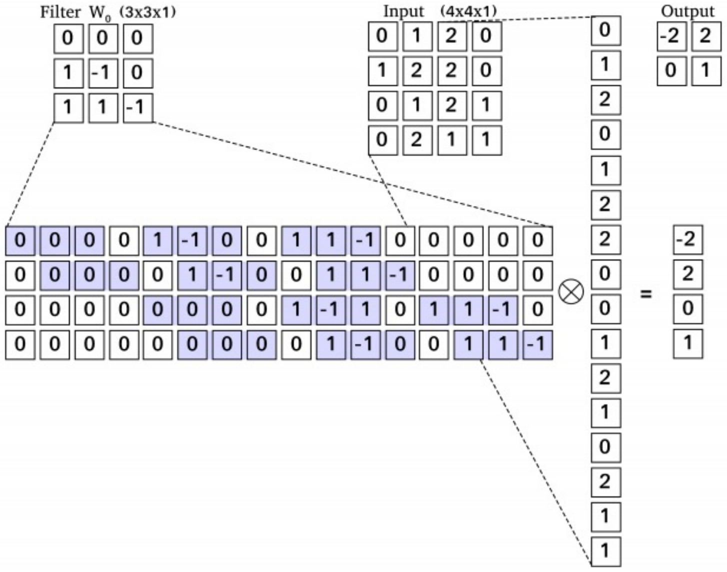
$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=1, padding=1

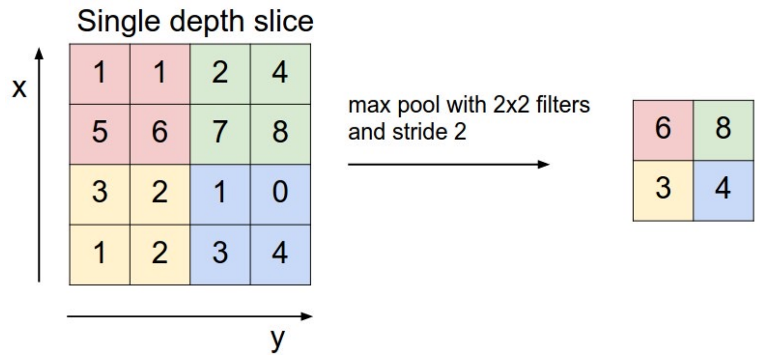
2D examples





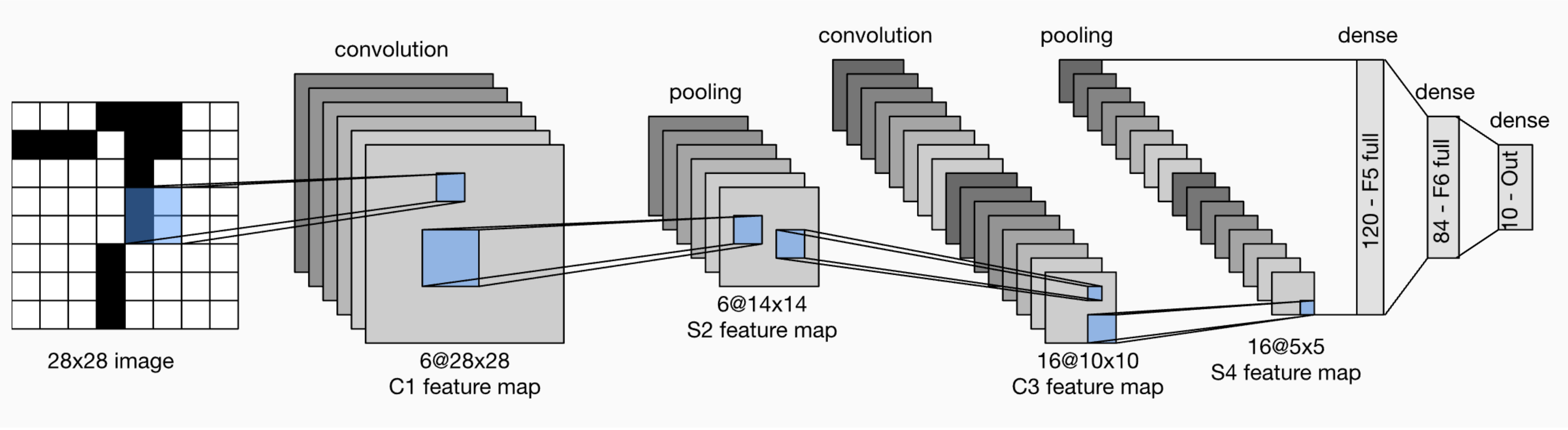
The pooling operations (not 🏊‍♀️)

- 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

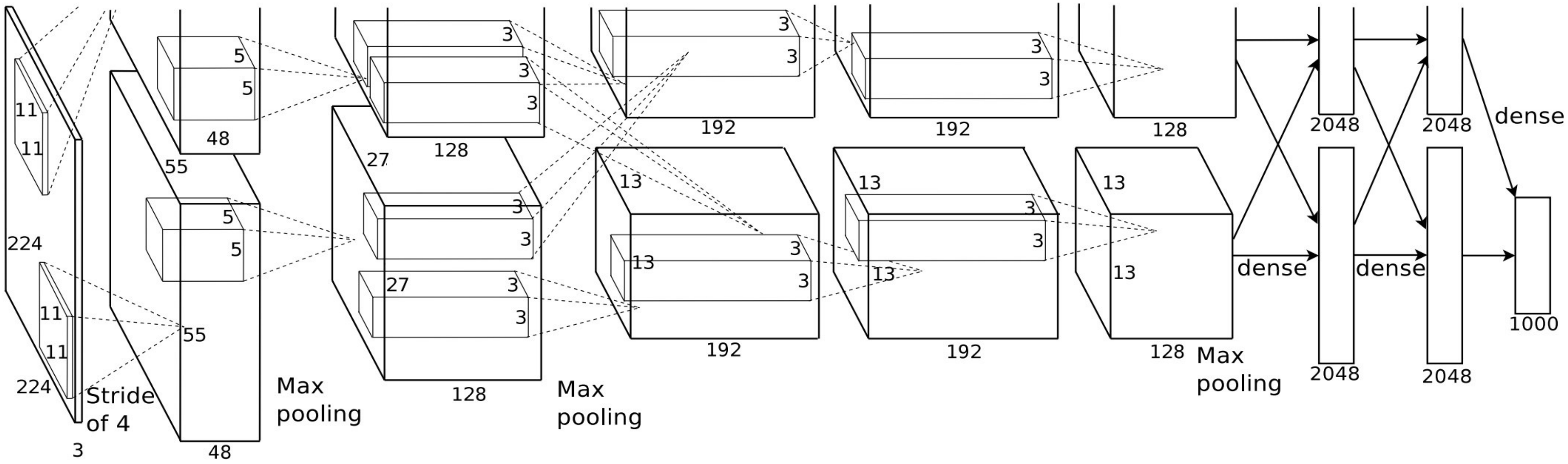


Fig … es
bet … rts
at t … nd
the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

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

[Krizhevsky, Sutskever, Hinton. NIPS 2012]

ConvNets

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

What shape should the activations have?

$$x \rightarrow \boxed{\text{Layer}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Layer}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$

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

What shape should the activations have?

$$x \rightarrow \boxed{\text{Layer}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Layer}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$

- 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?

$$x \to \boxed{\text{Layer}} \to h^{(1)} \to \boxed{\text{Layer}} \to h^{(2)} \to \cdots \to f$$

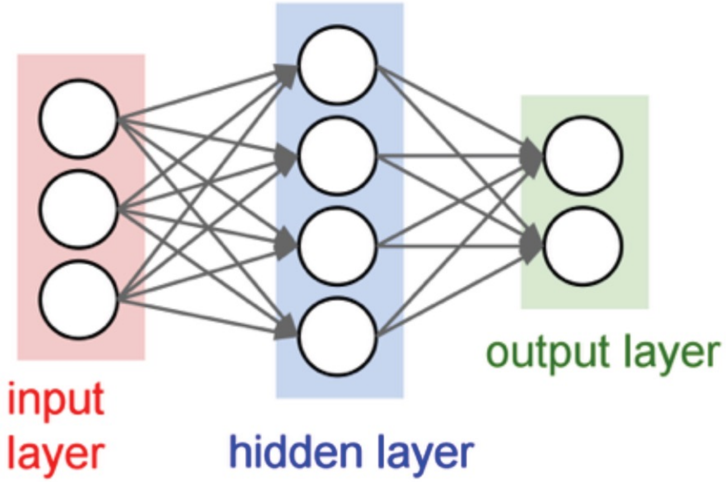
- The input is an image, which is 3D
(RGB channel, height, width)

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

- What about keeping everything in 3D?

3D Activations

before:

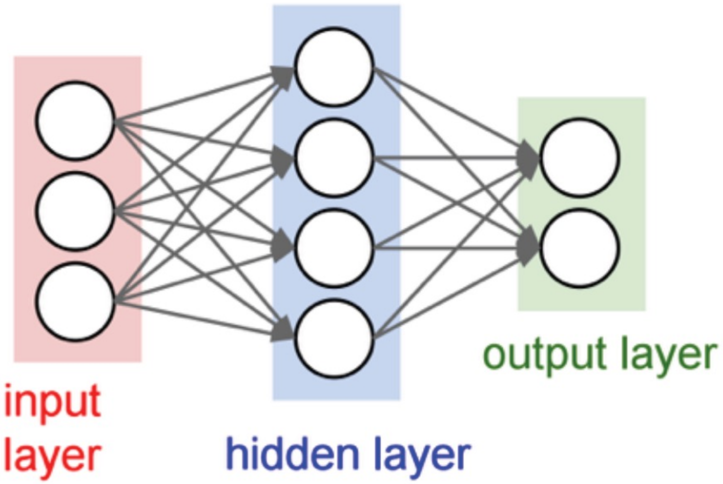


(1D vectors)

Figure: Andrej Karpathy

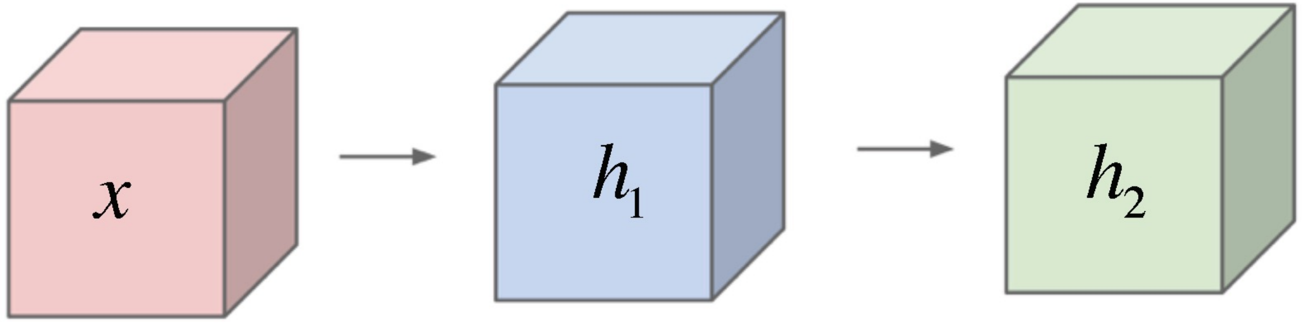
3D Activations

before:



(1D vectors)

now:



(3D arrays)

Figure: Andrej Karpathy

3D Activations

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

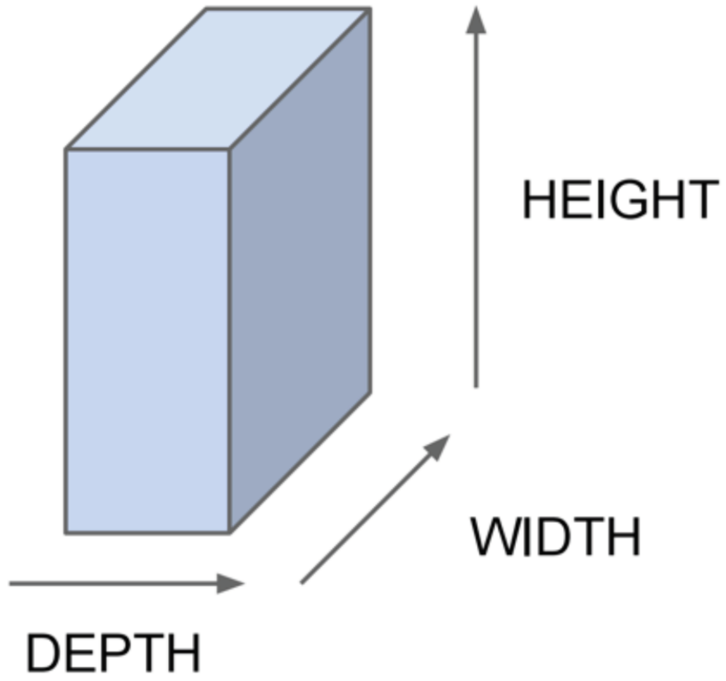
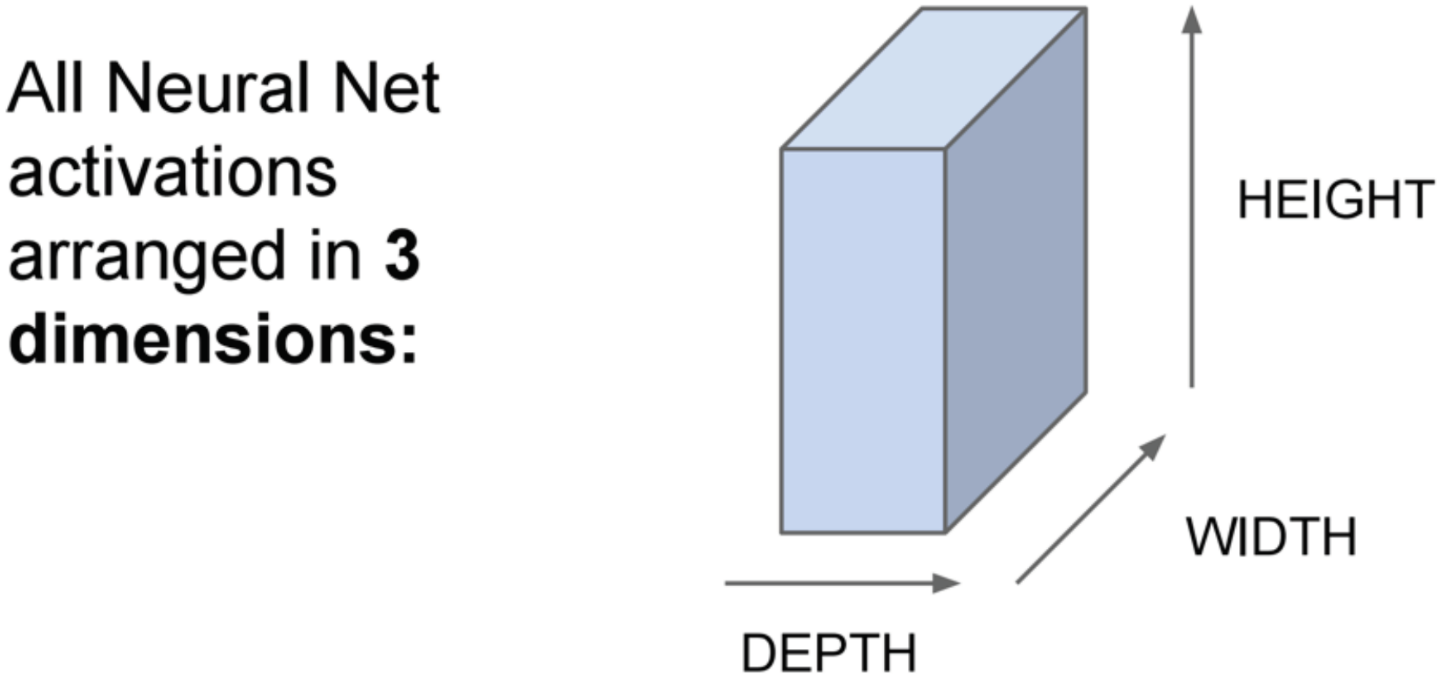


Figure: Andrej Karpathy

3D Activations



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:

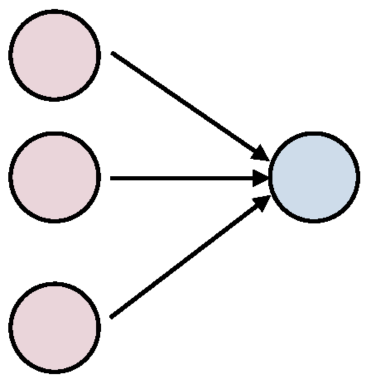
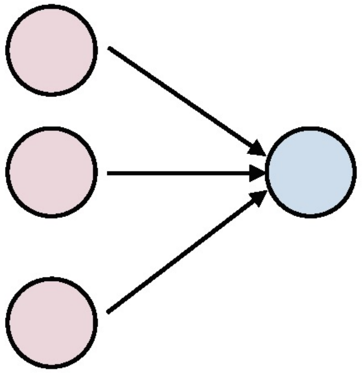


Figure: Andrej Karpathy

3D Activations

1D Activations:



3D Activations:

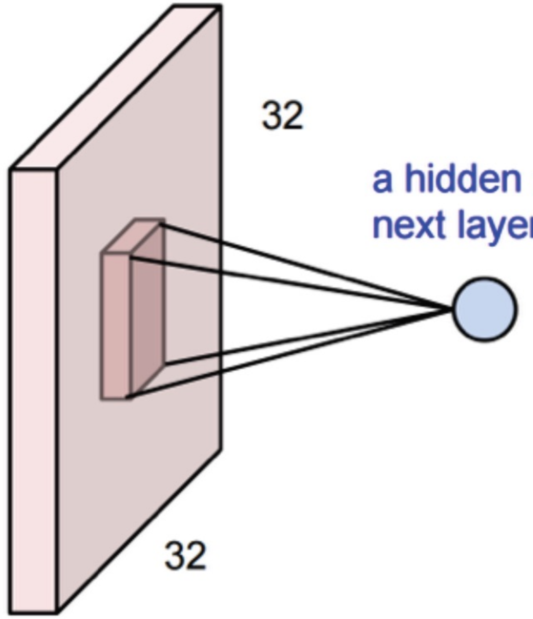


Figure: Andrej Karpathy

3D Activations

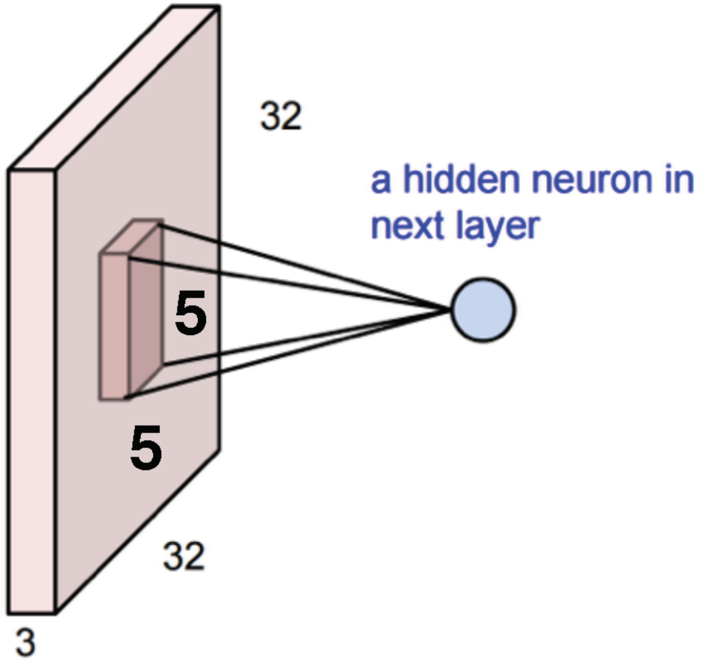


Figure: Andrej Karpathy

- 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)

3D Activations

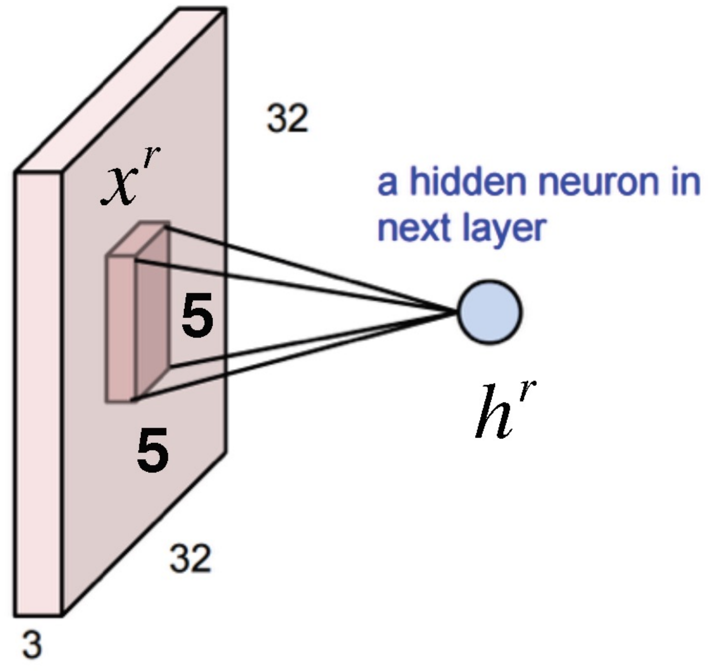


Figure: Andrej Karpathy

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

With output neuron h^r

3D Activations

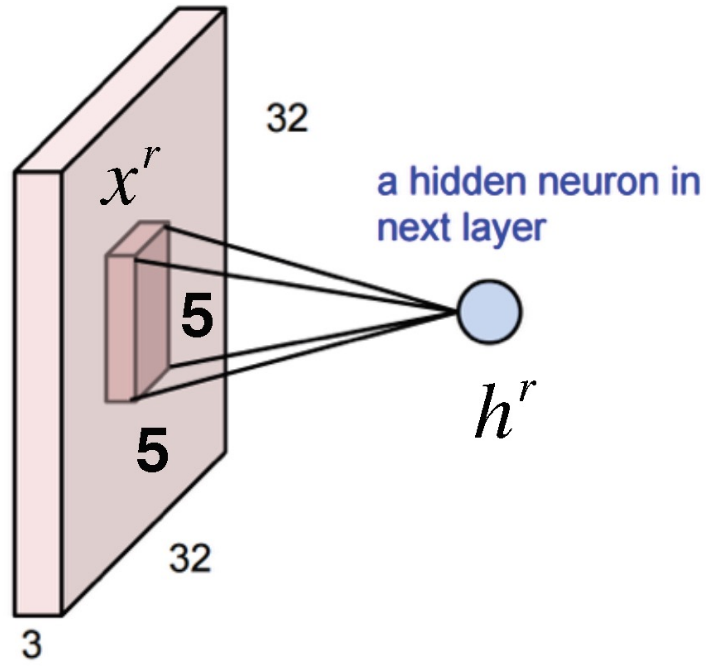


Figure: Andrej Karpathy

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

With output neuron h^r

Then the output is:

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

3D Activations

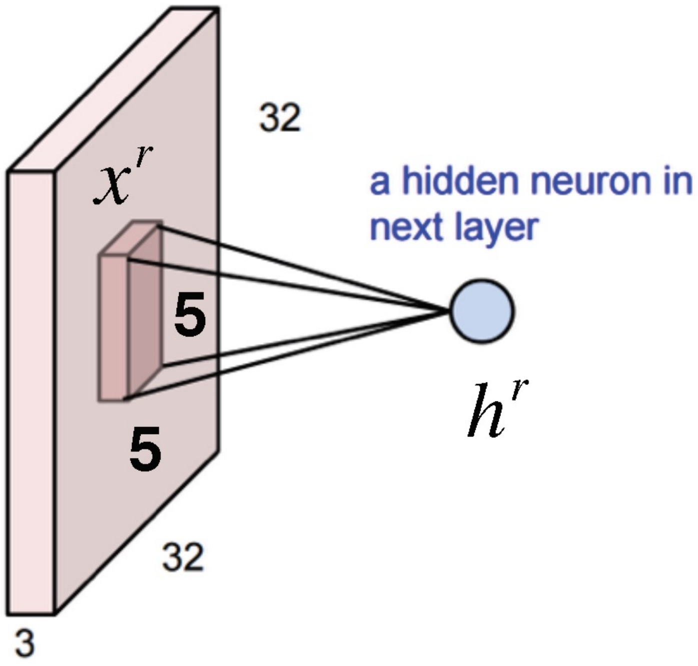


Figure: Andrej Karpathy

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

3D Activations

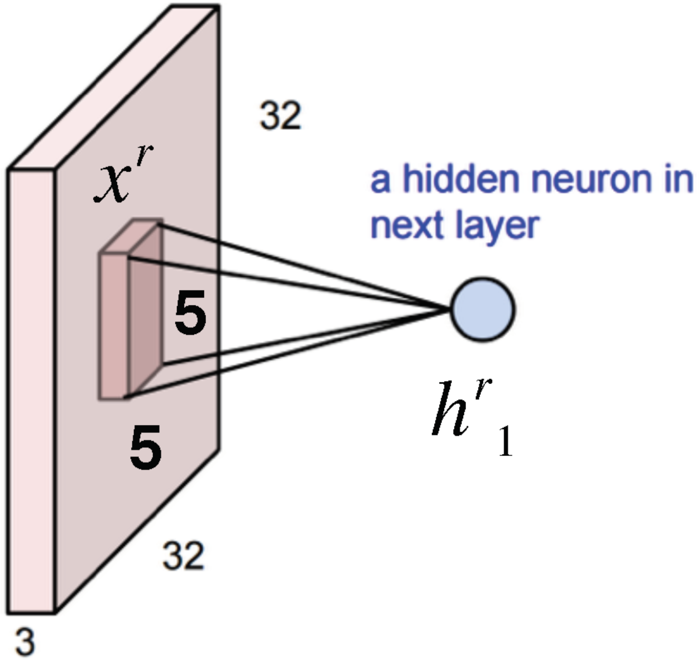


Figure: Andrej Karpathy

3D Activations

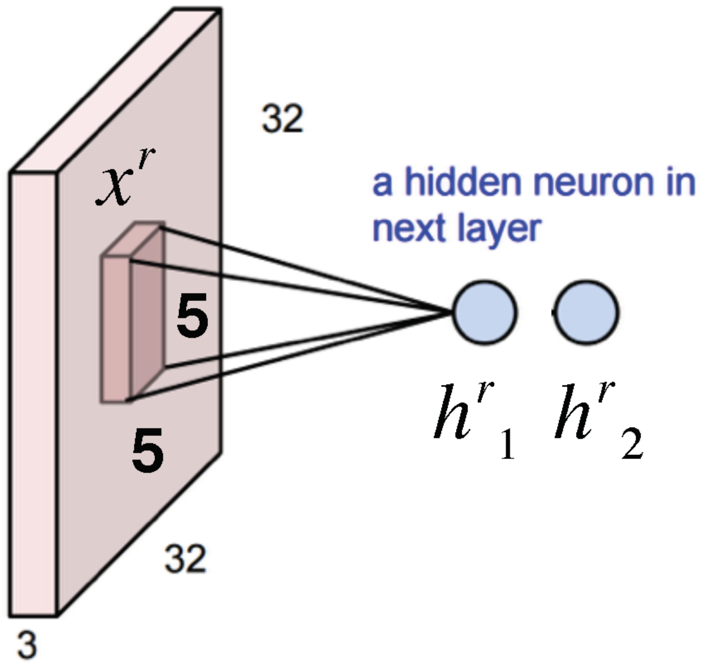
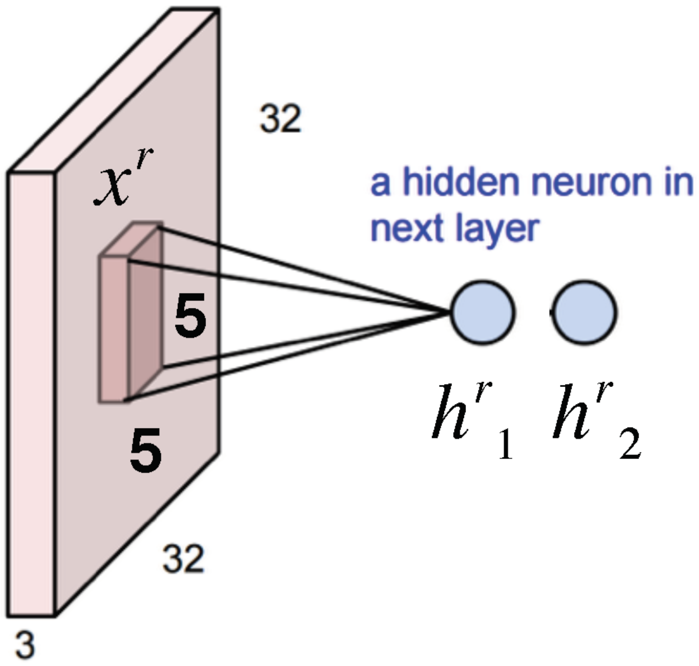


Figure: Andrej Karpathy

3D Activations



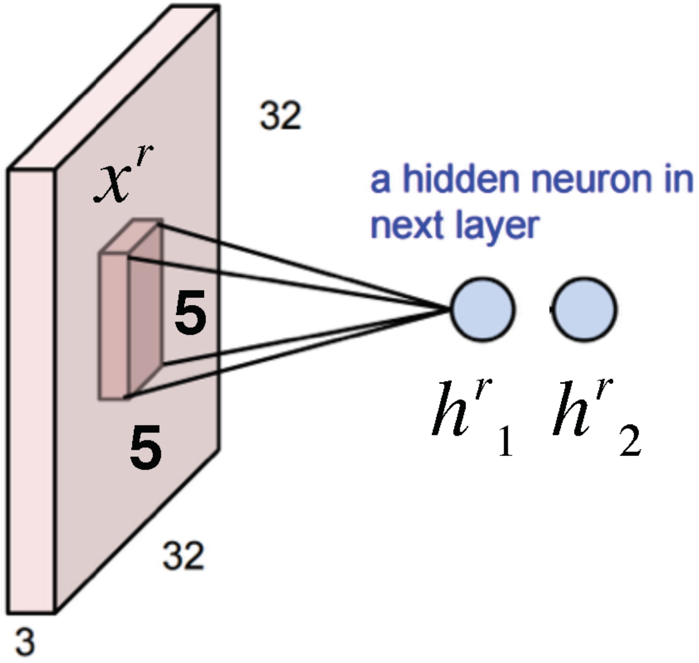
With **2** output neurons

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

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

Figure: Andrej Karpathy

3D Activations



With **2** output neurons

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

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

Figure: Andrej Karpathy

3D Activations

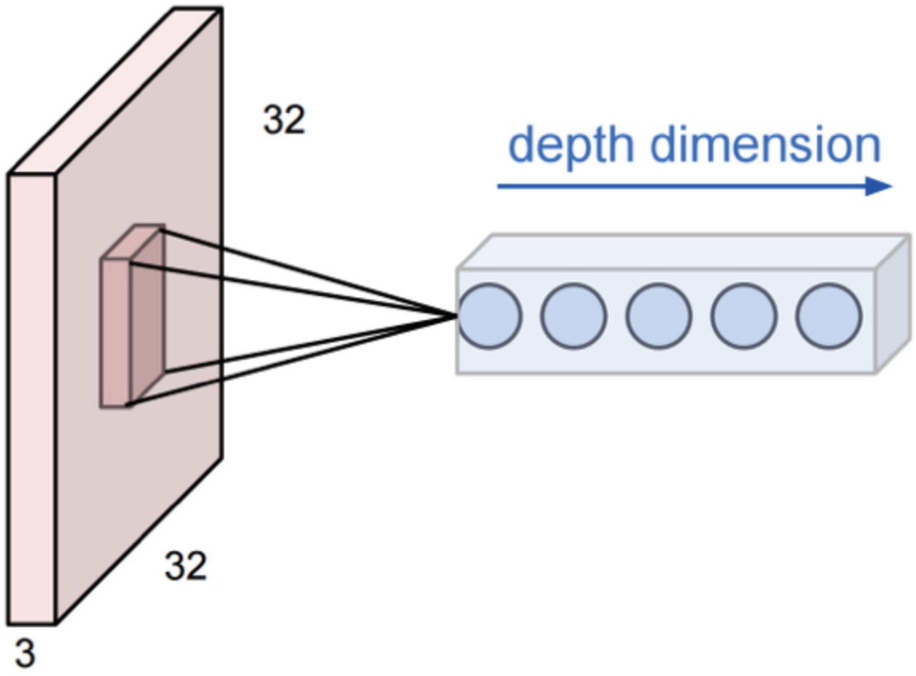


Figure: Andrej Karpathy

3D Activations

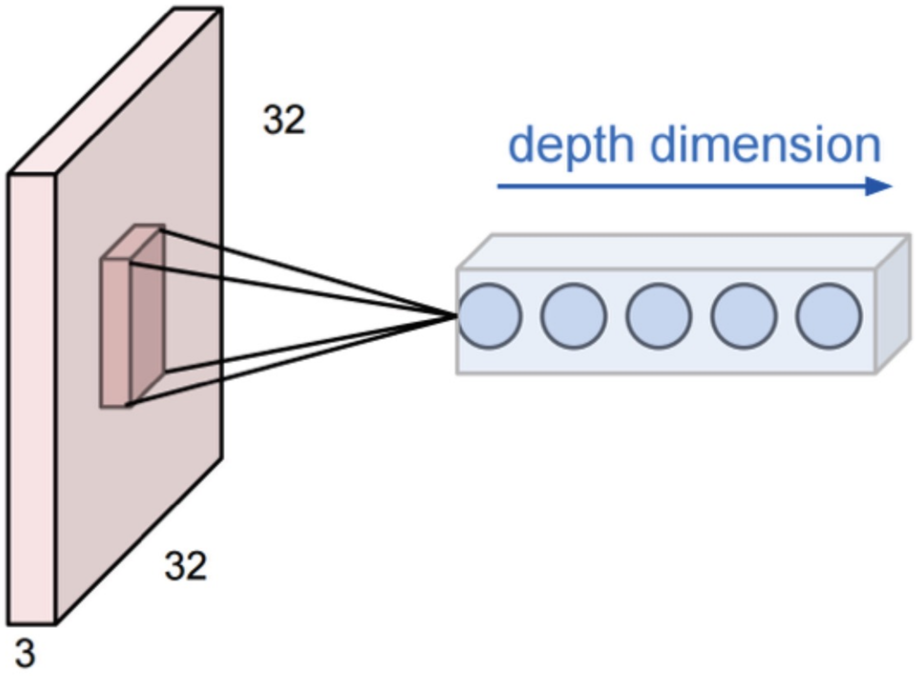
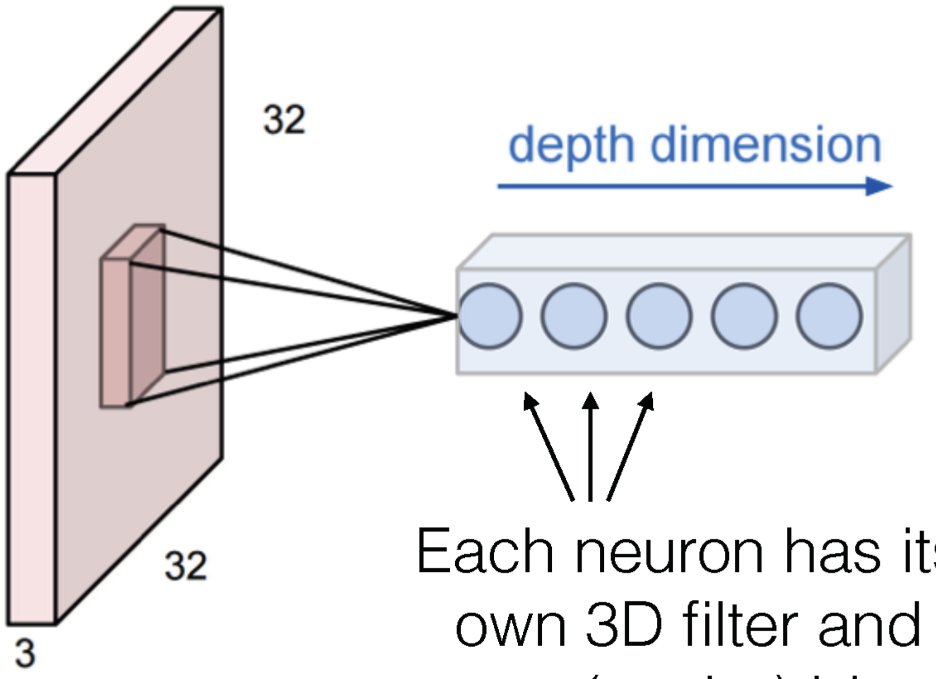


Figure: Andrej Karpathy

We can keep adding more outputs

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

3D Activations



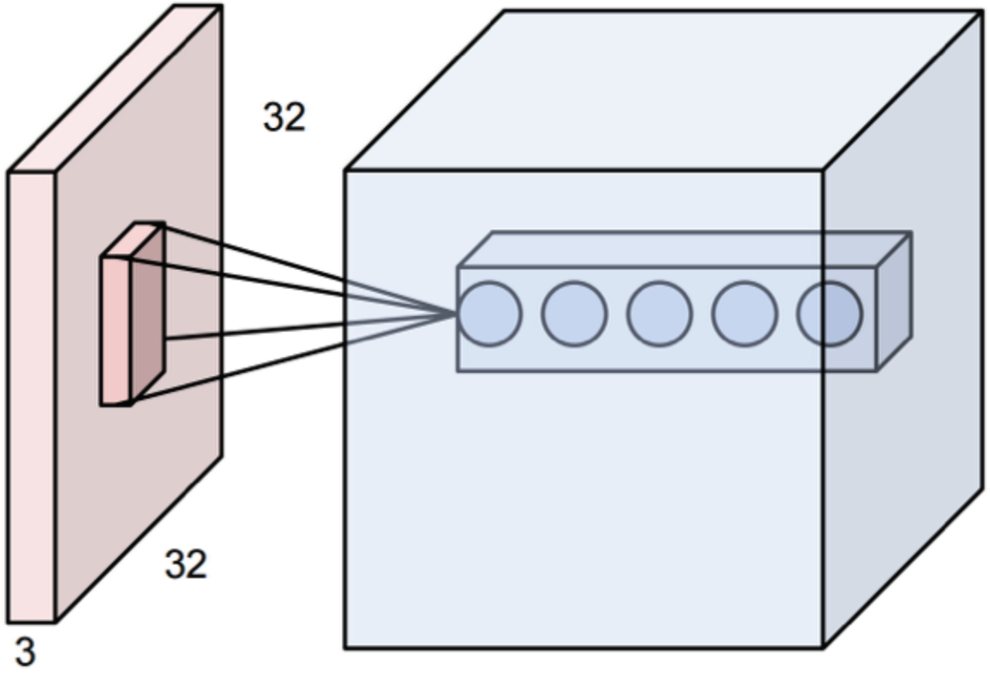
Each neuron has its own 3D filter and own (scalar) bias

We can keep adding more outputs

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

Figure: Andrej Karpathy

3D Activations

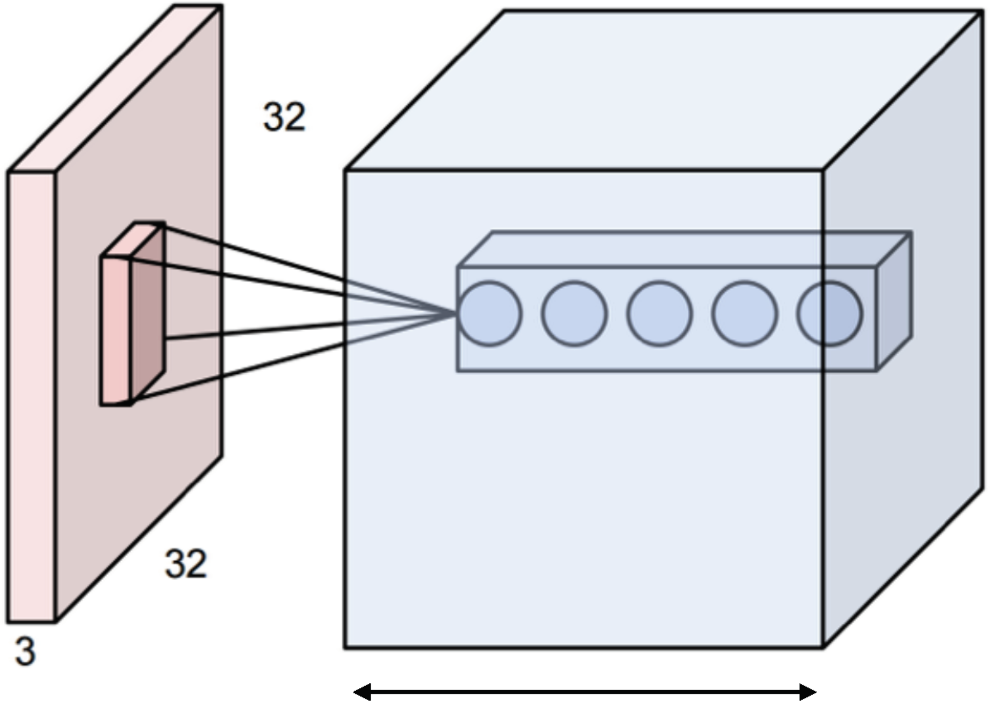


D sets of weights
(also called filters)

Now repeat this across the input

Figure: Andrej Karpathy

3D Activations



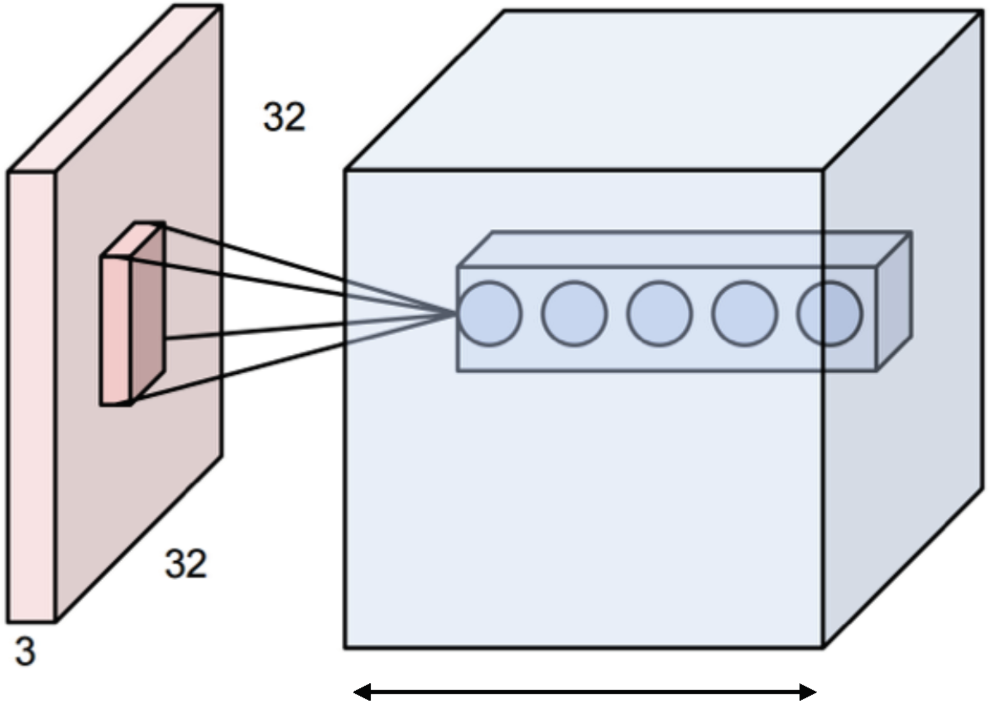
D sets of weights
(also called filters)

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

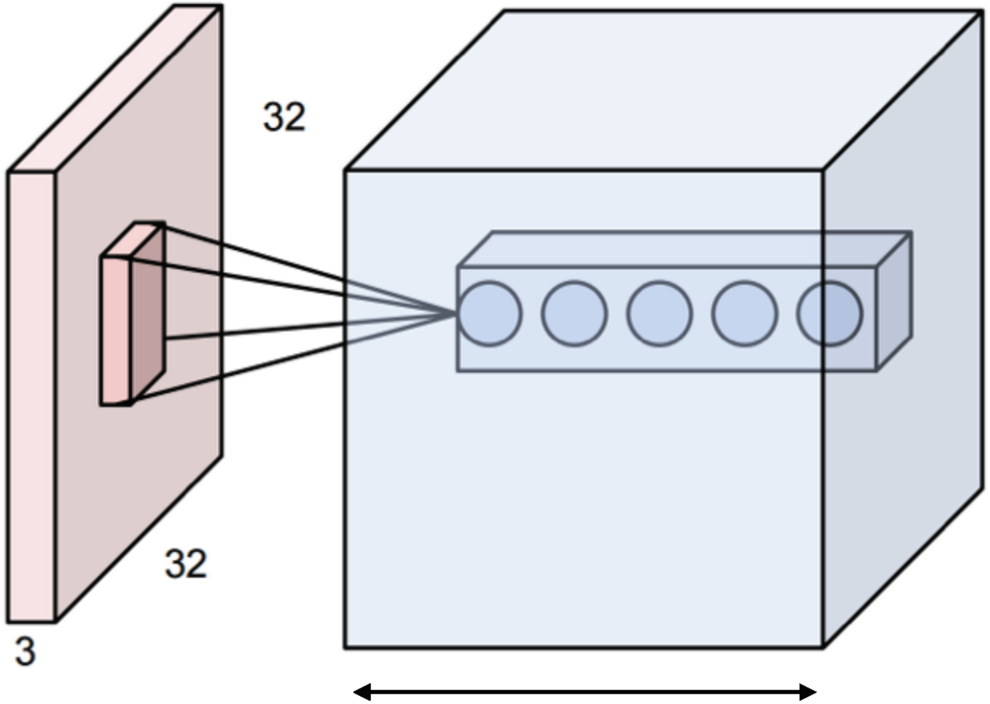
3D Activations



D sets of weights
(also called filters)

Figure: Andrej Karpathy

3D Activations

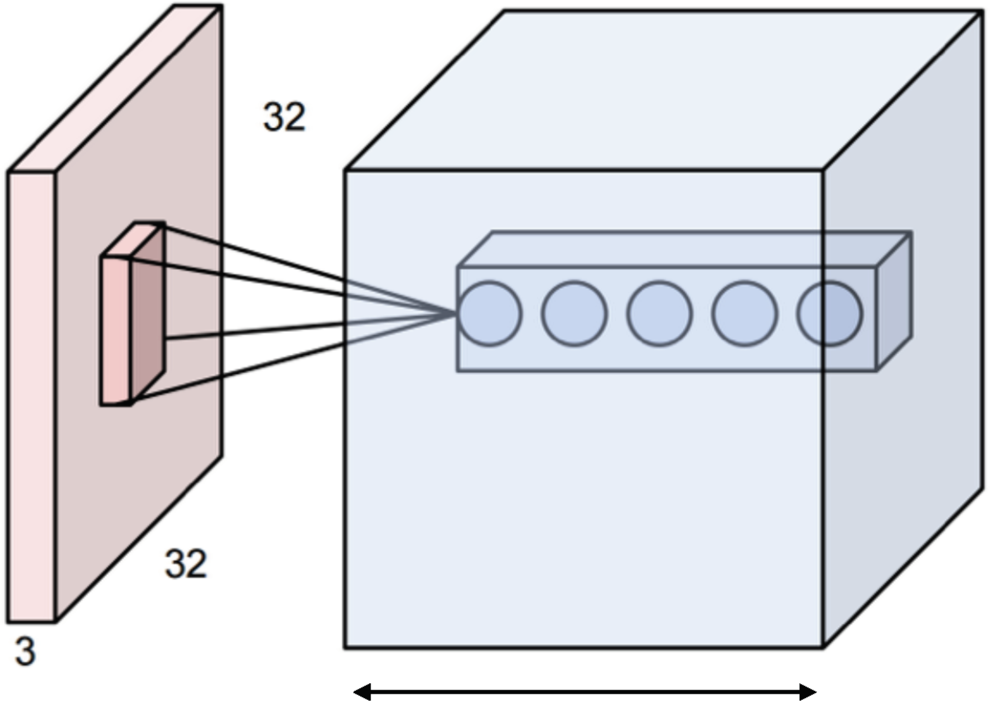


D sets of weights
(also called filters)

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

Figure: Andrej Karpathy

3D Activations



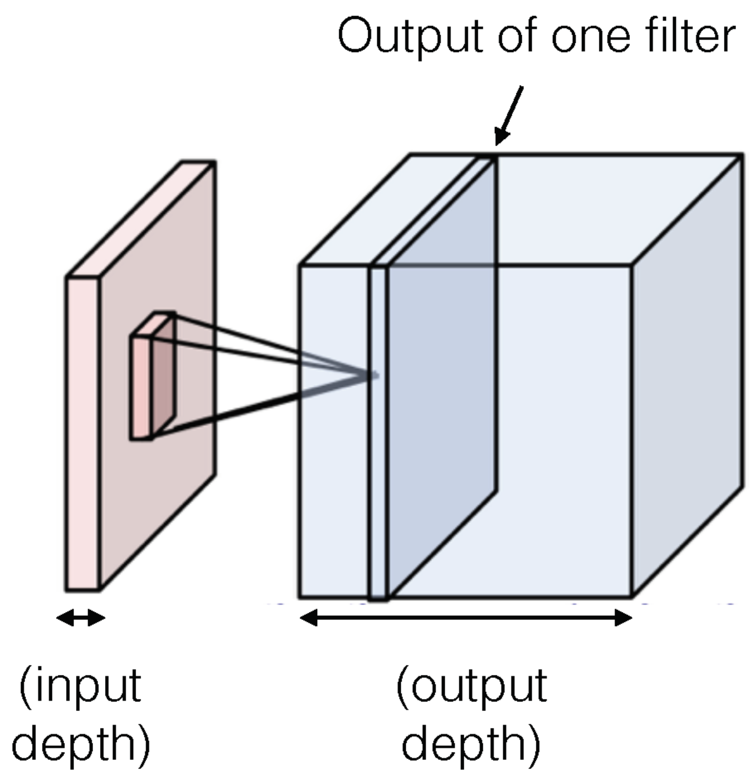
D sets of weights
(also called filters)

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

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

Figure: Andrej Karpathy

3D Activations

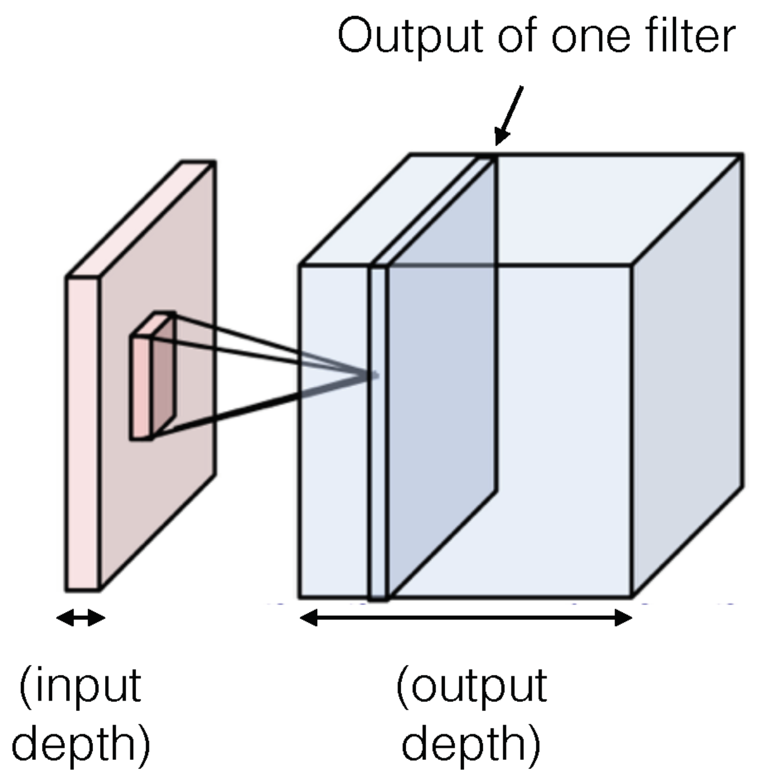


One set of weights gives one slice in the output

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

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

3D Activations



One set of weights gives one slice in the output

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

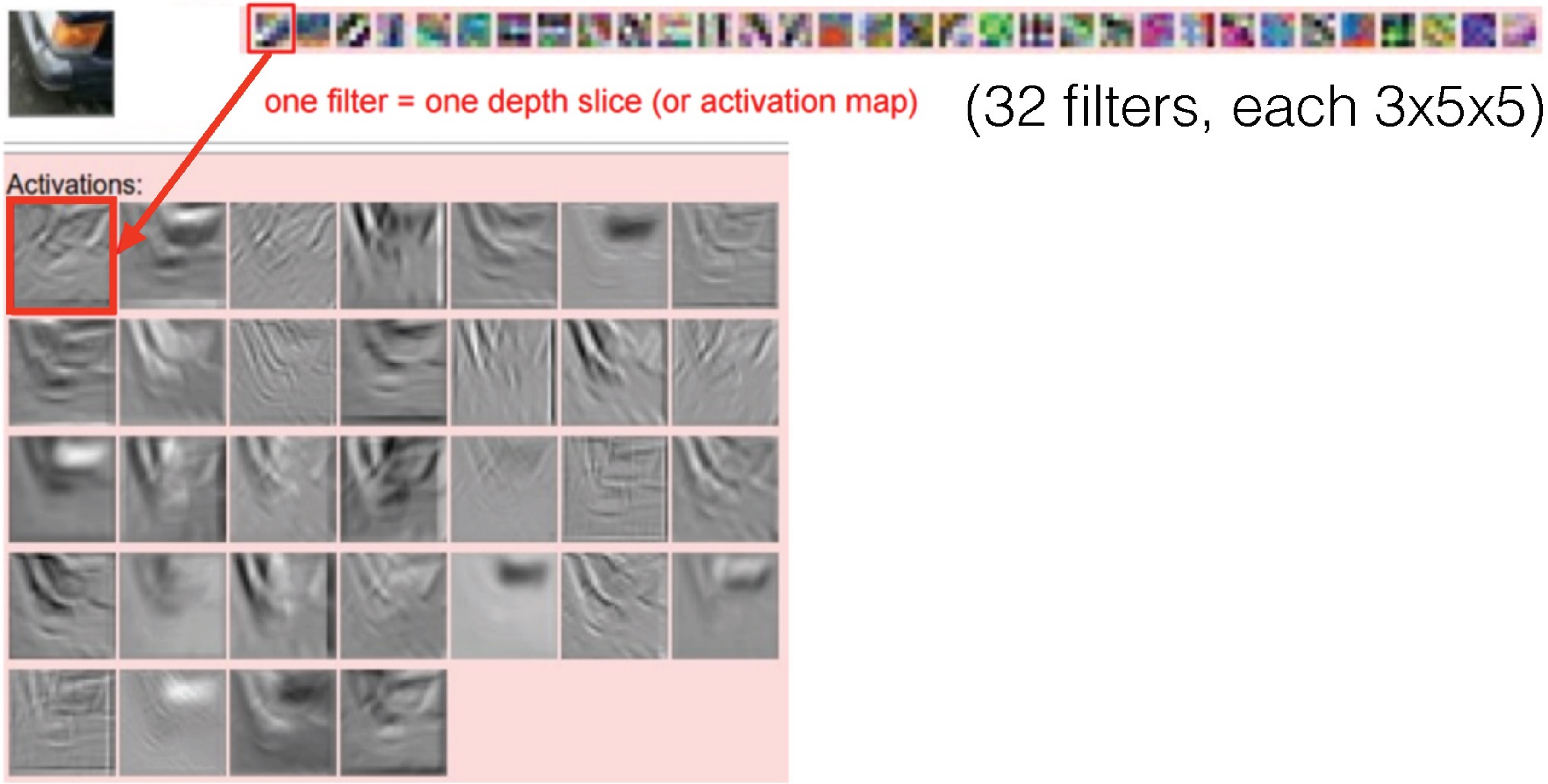
In practice, ConvNets use many filters (~64 to 1024)

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:

(Input)



(32 filters, each 3x5x5)

Figure: Andrej Karpathy

3D Activations

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

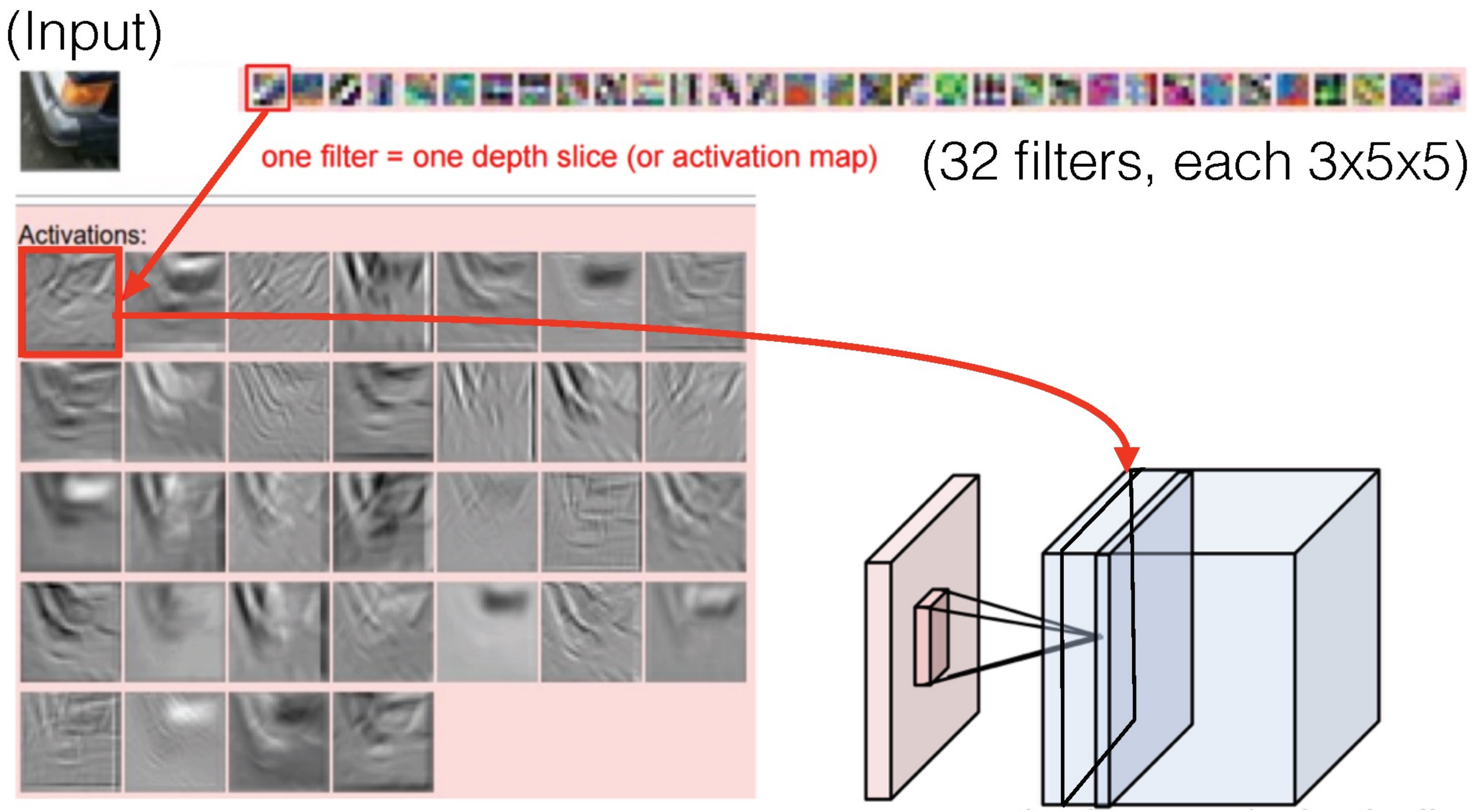


Figure: Andrej Karpathy

3D Activations

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

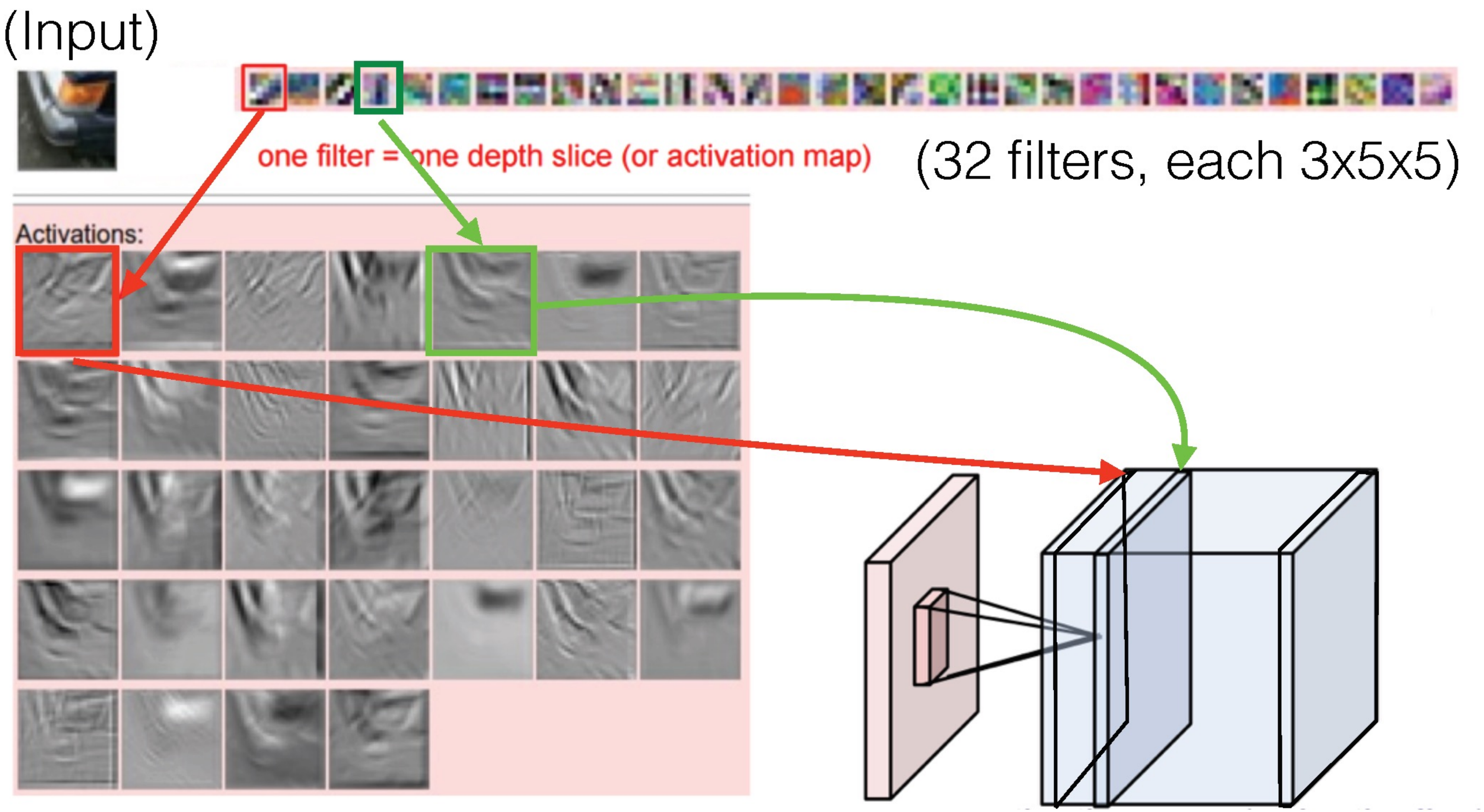


Figure: Andrej Karpathy

3D Activations

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

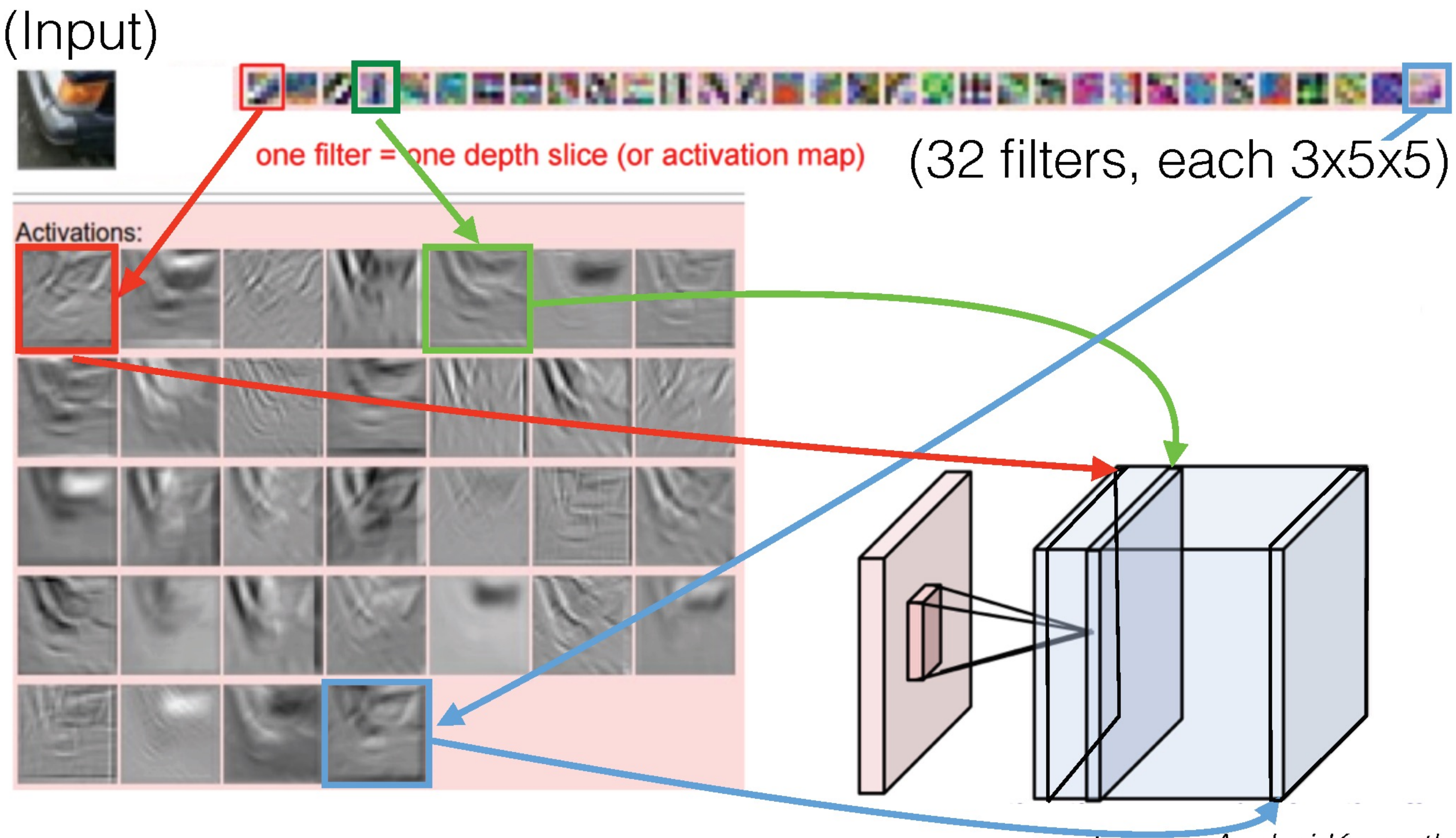
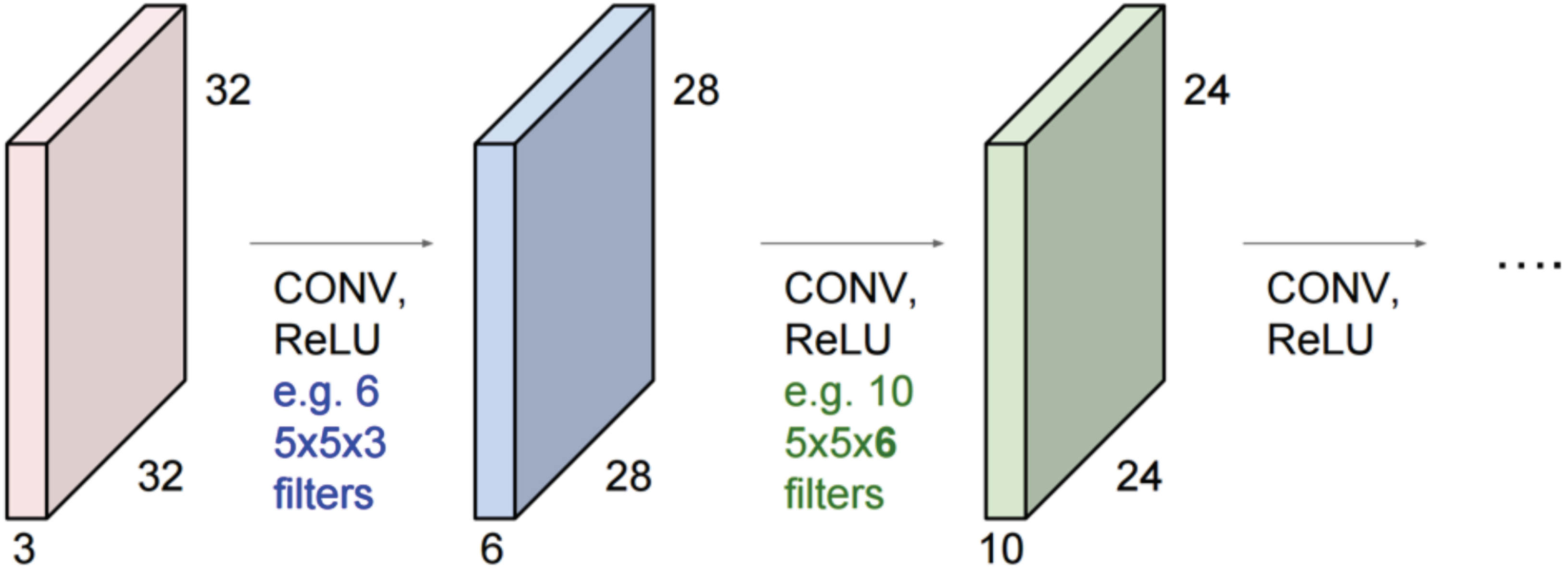


Figure: Andrej Karpathy

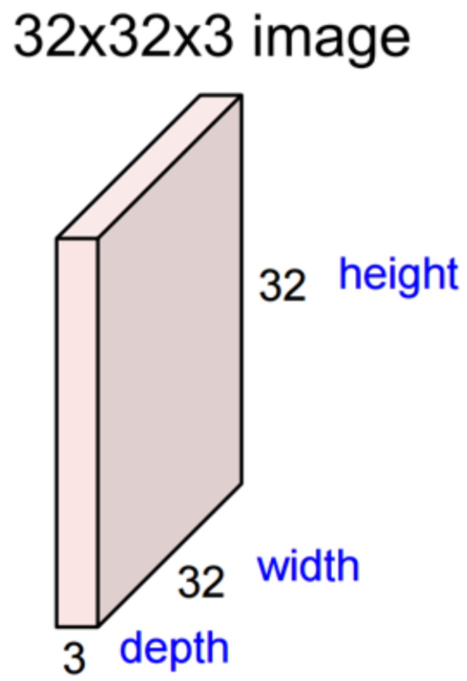
Putting it together

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



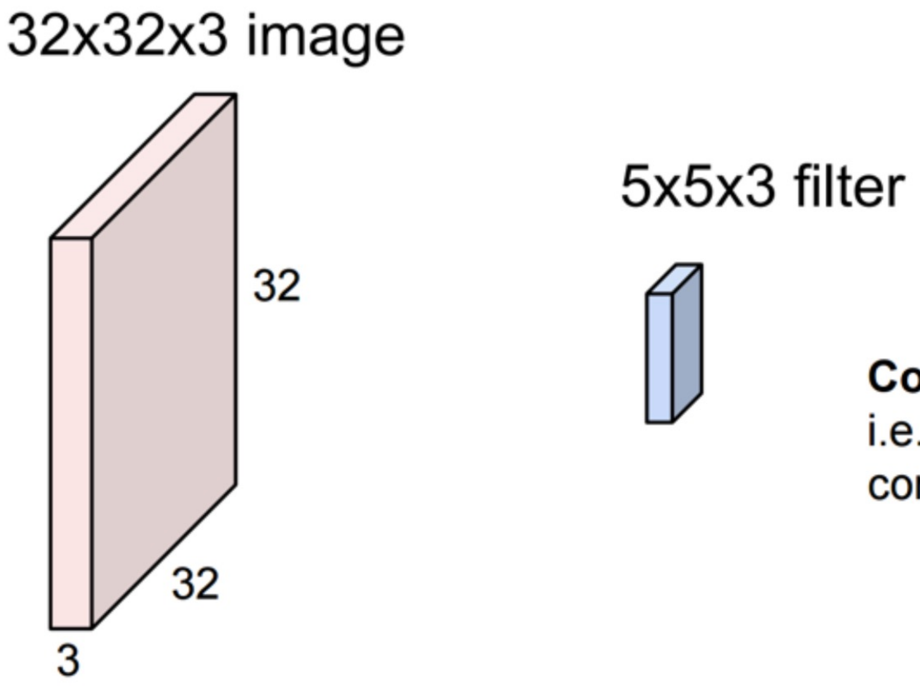
Putting it together

Convolution Layer



Putting it together

Convolution Layer



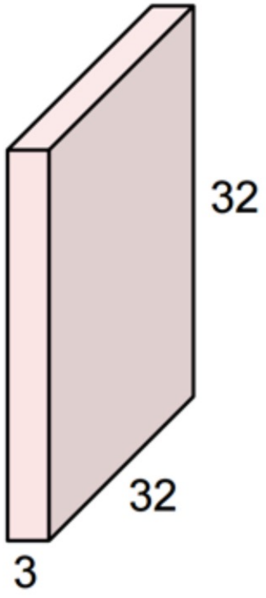
Convolve the filter with the image
i.e. "slide over the image spatially,
computing dot products"

Putting it together

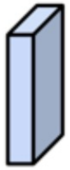
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image



5x5x3 filter



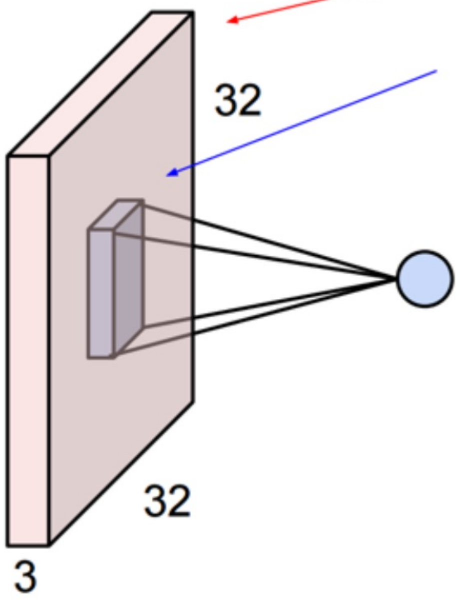
Convolve the filter with the image i.e. "slide over the image spatially, computing dot products"

Putting it together

Convolution Layer

32x32x3 image

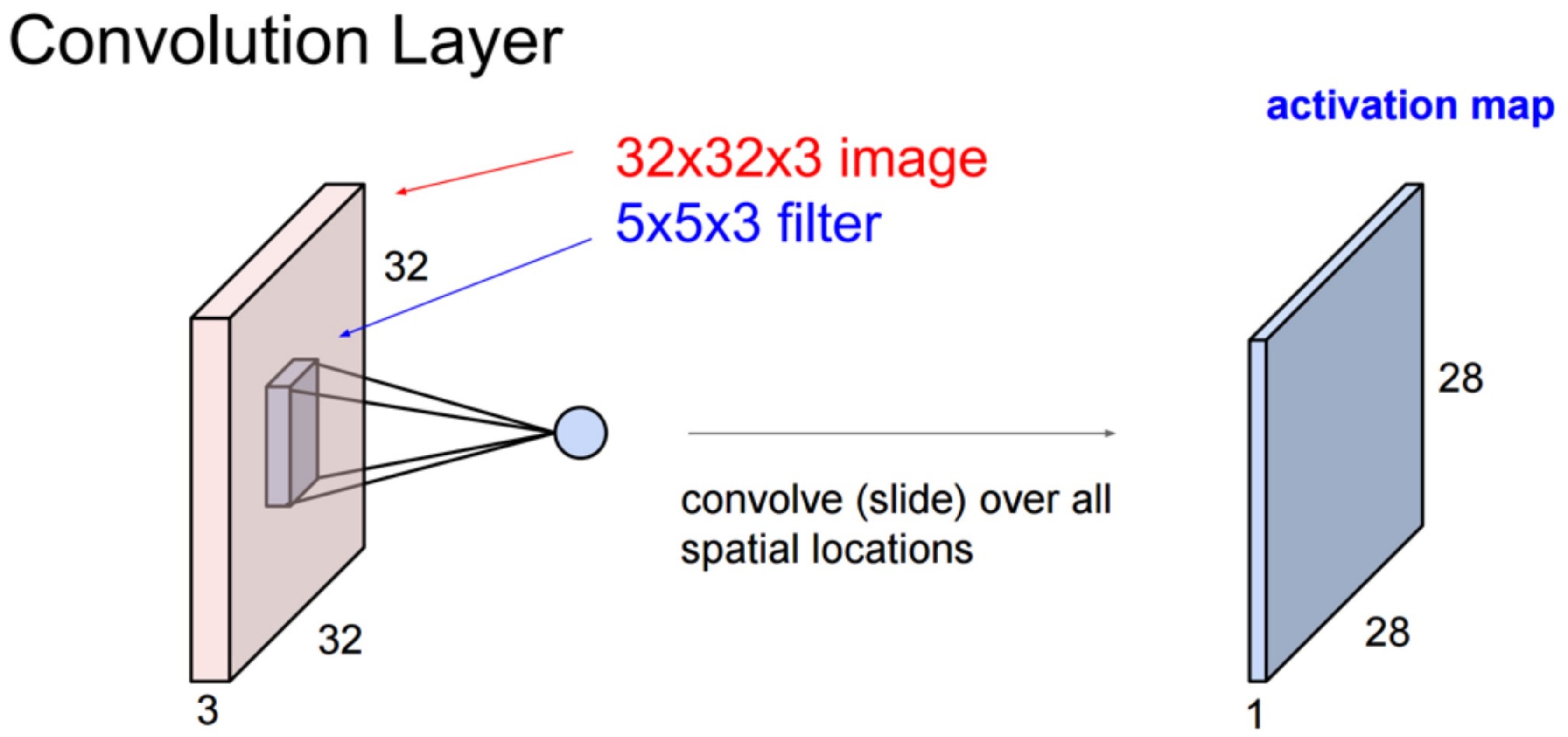
5x5x3 filter w



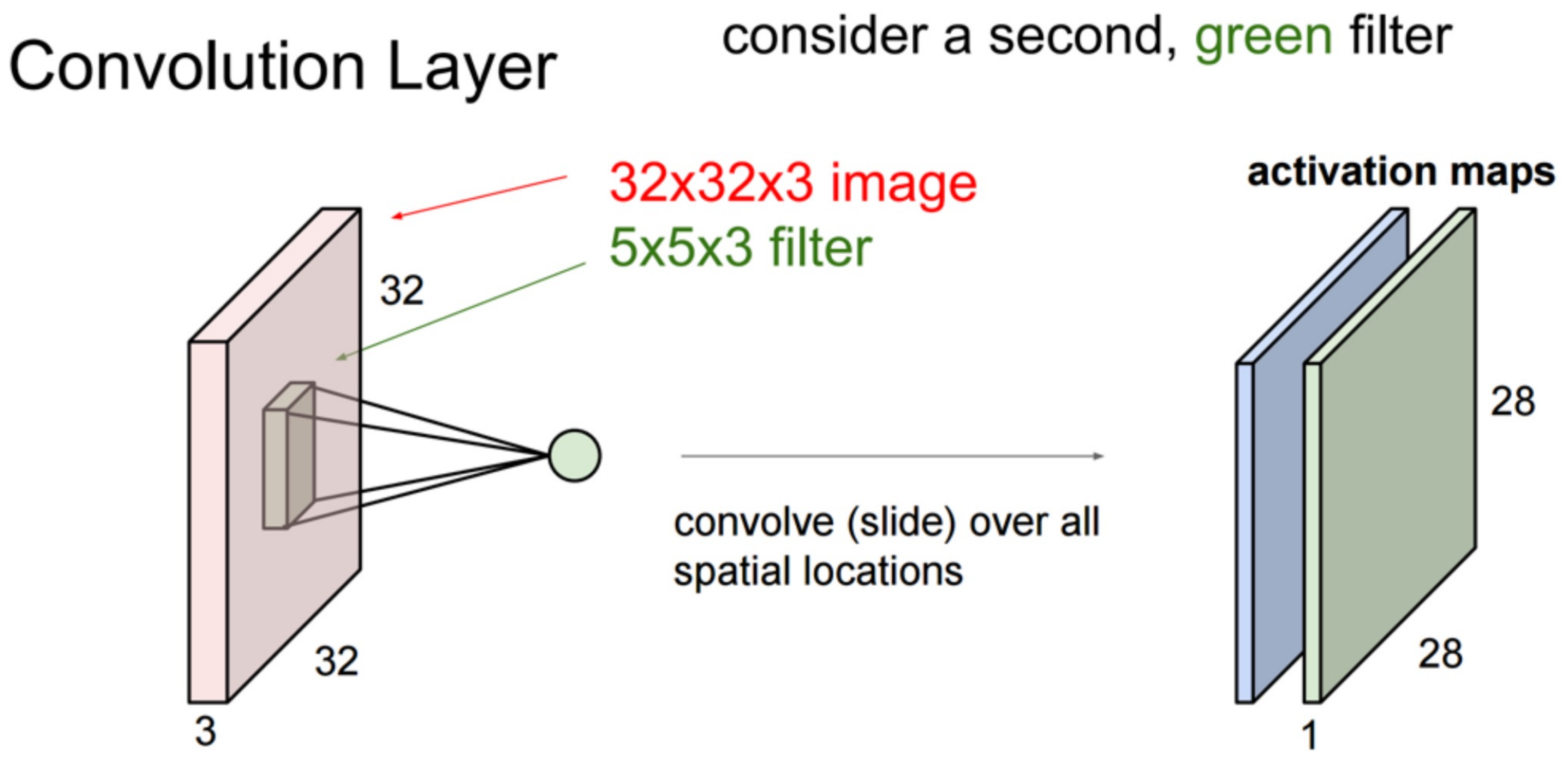
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

Putting it together

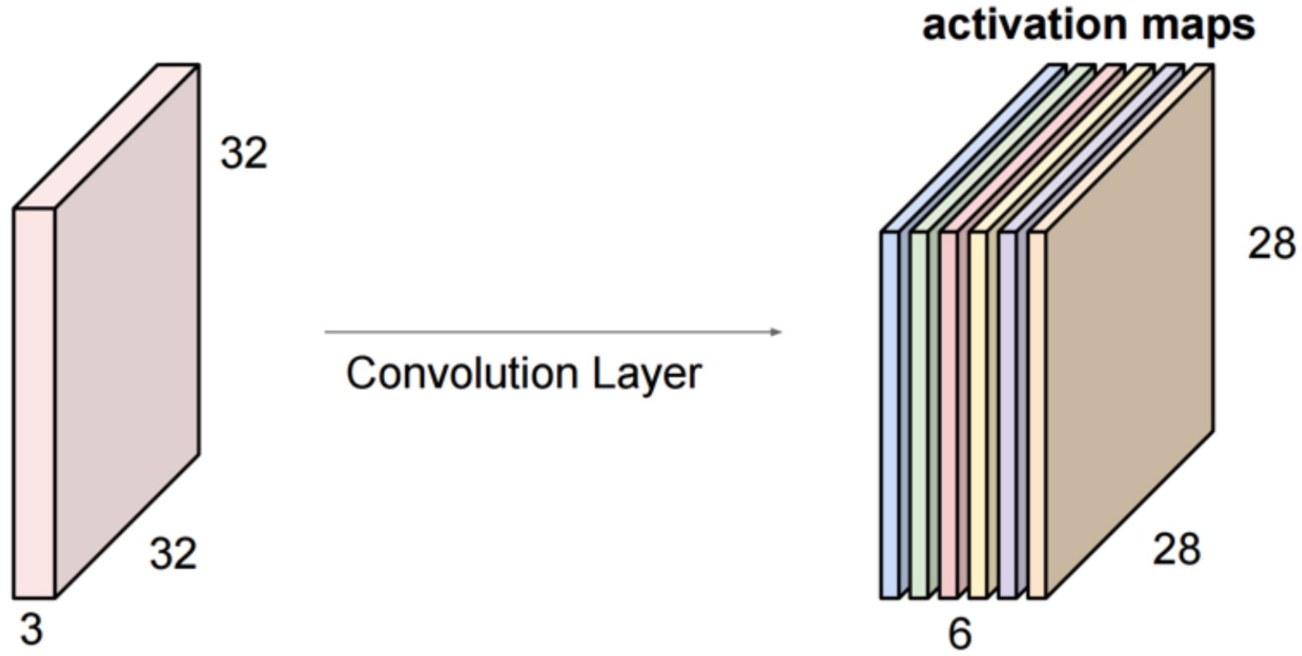


Putting it together



Putting it together

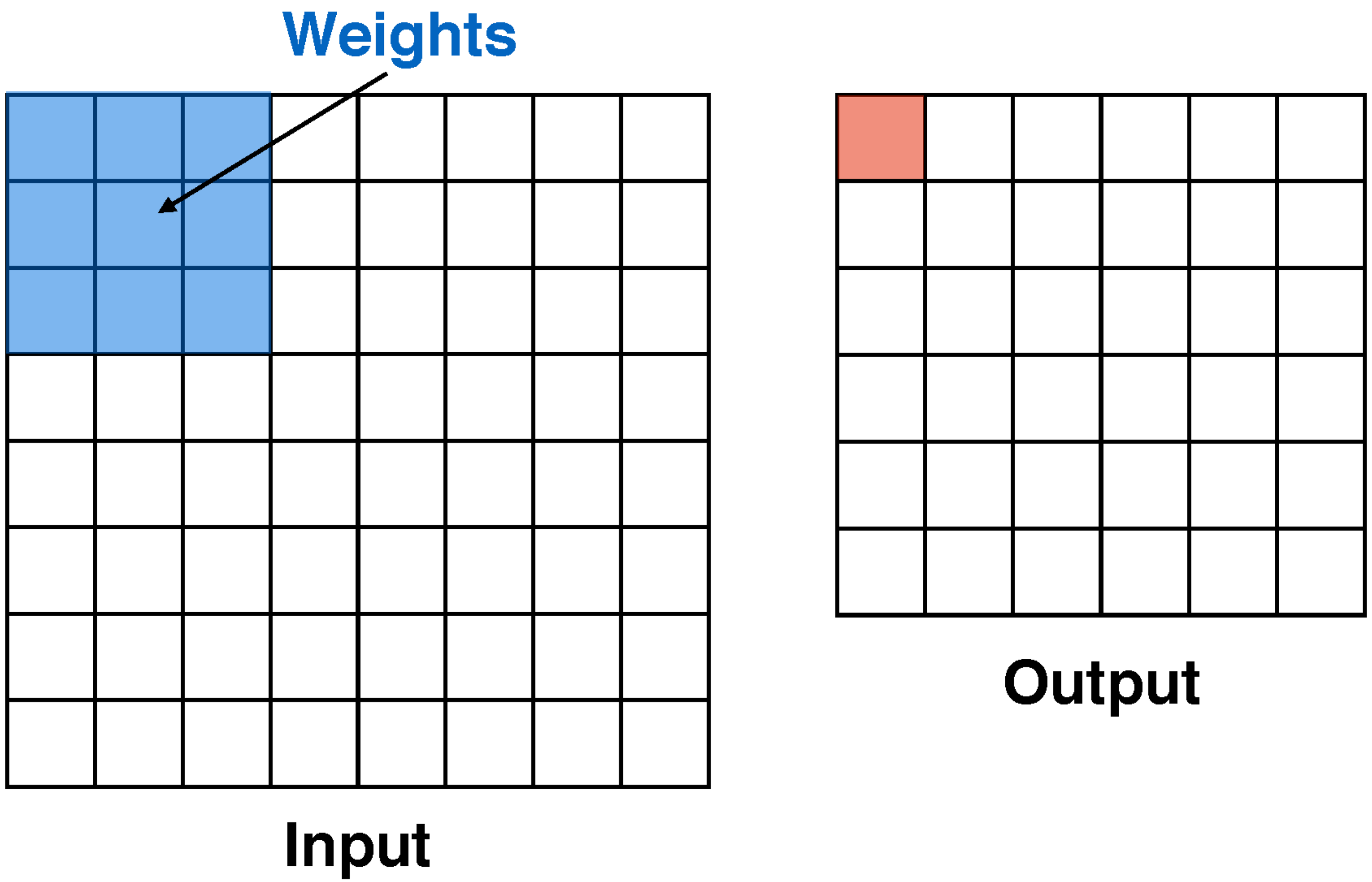
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



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

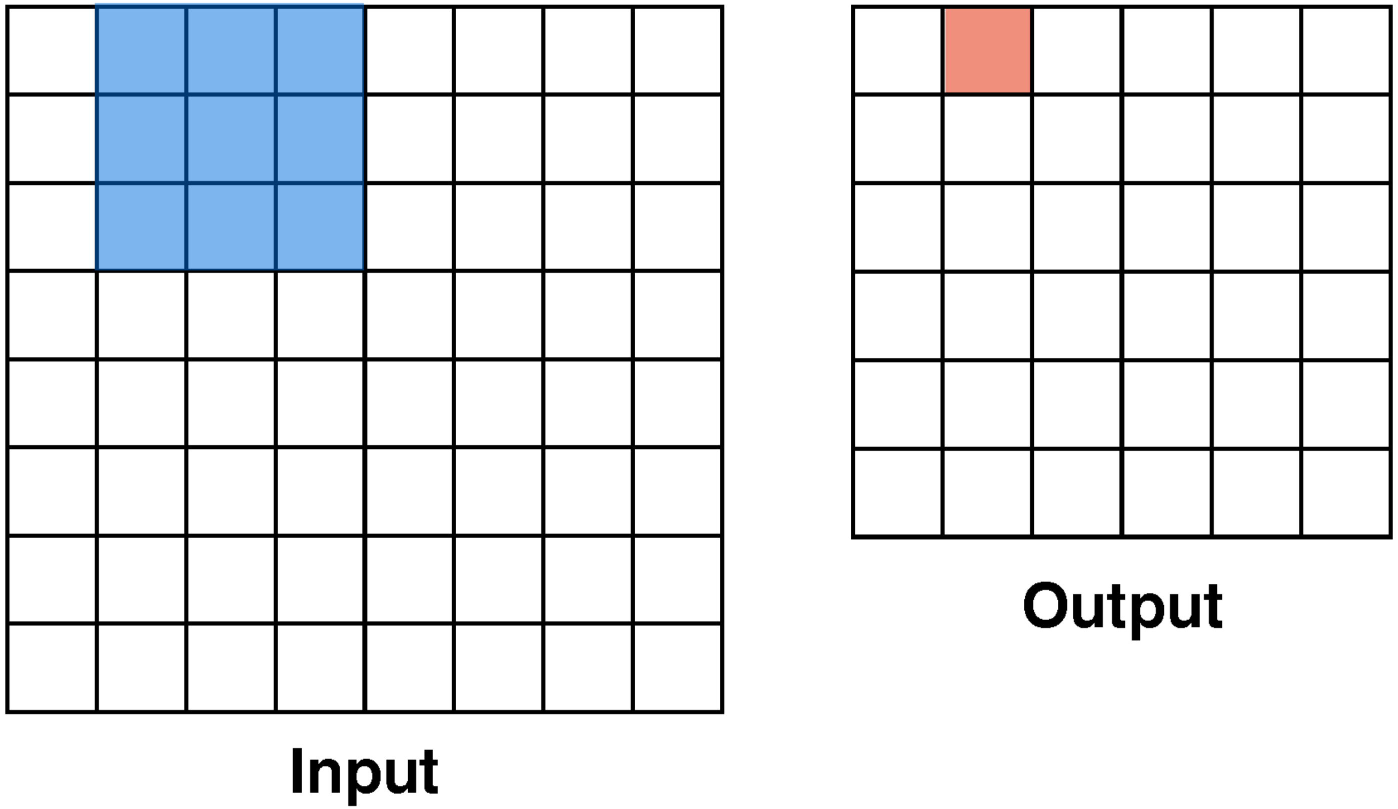
Convolution: Stride

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



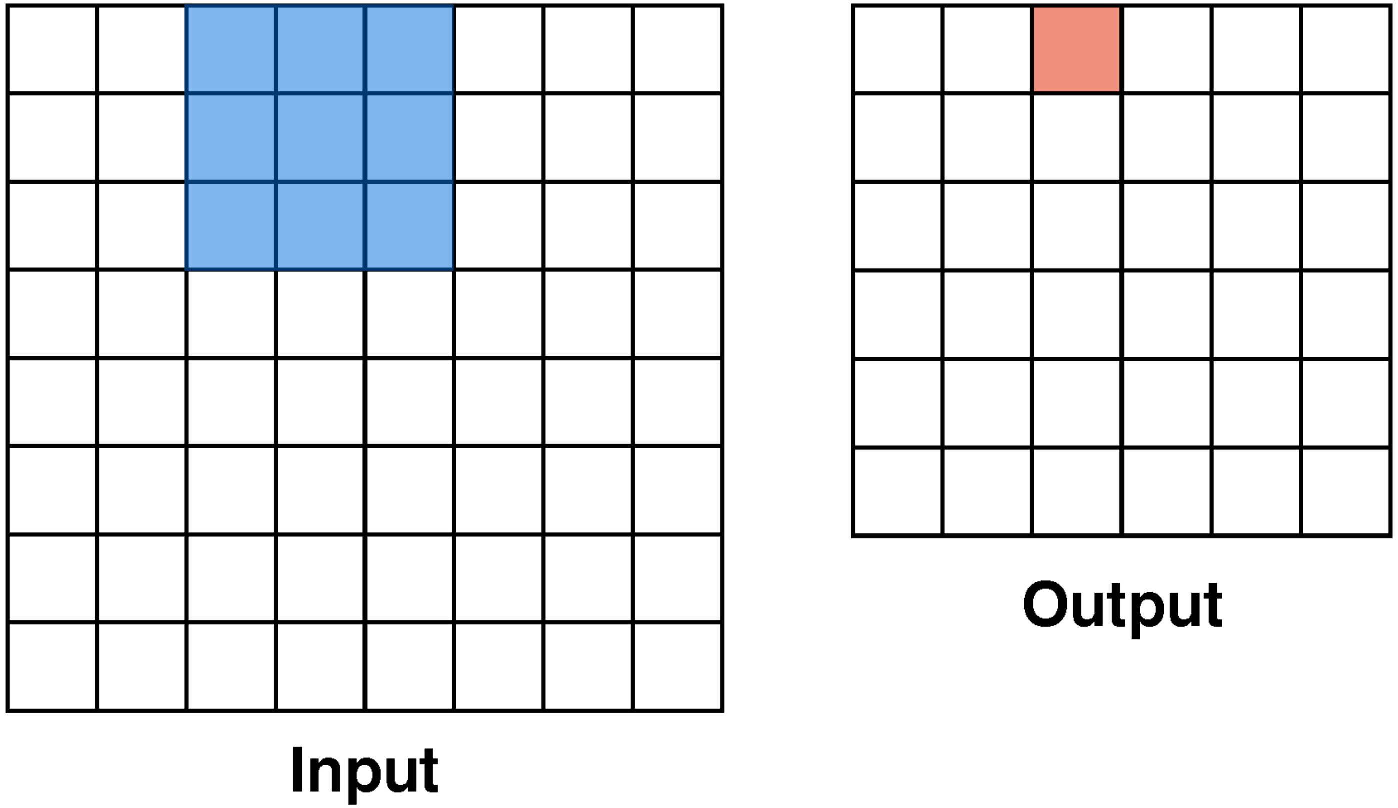
Convolution: Stride

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



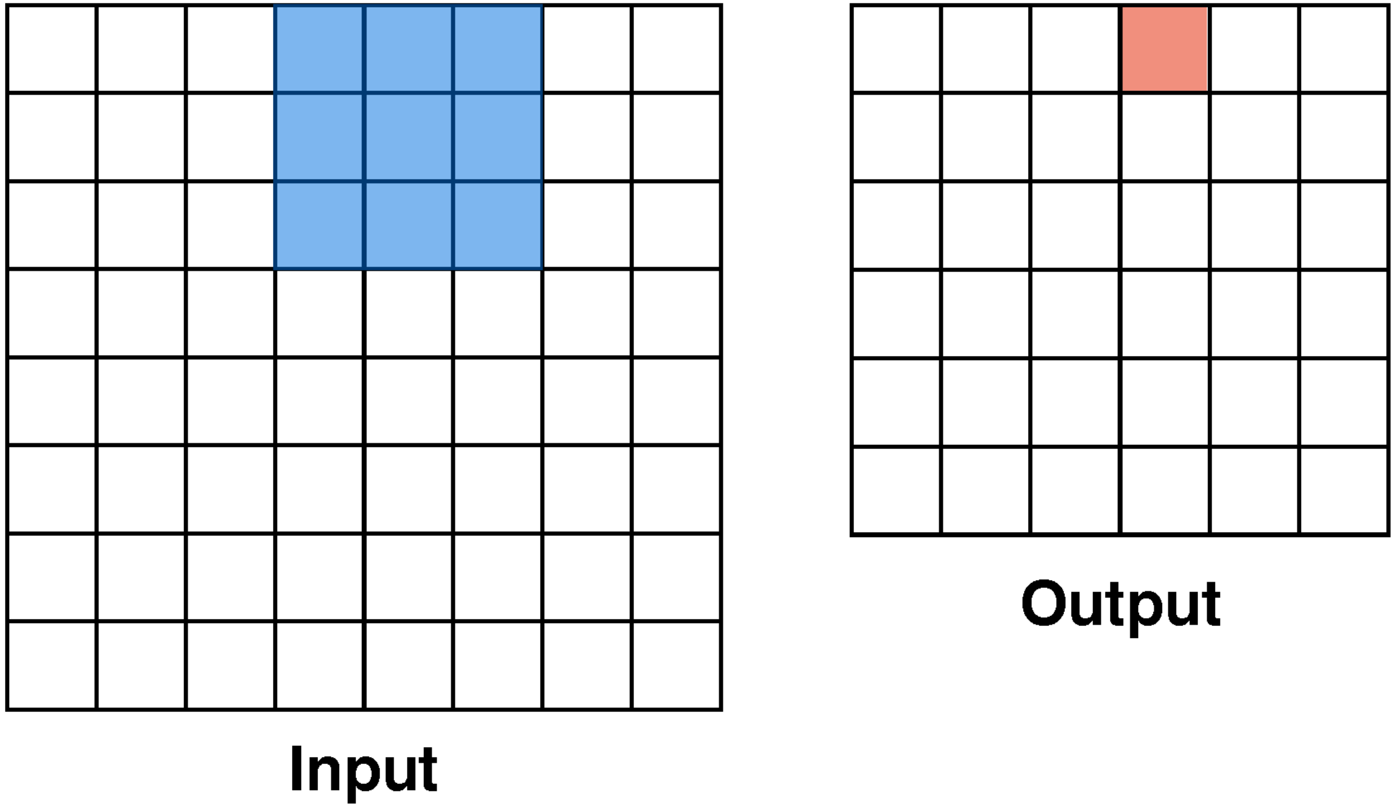
Convolution: Stride

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



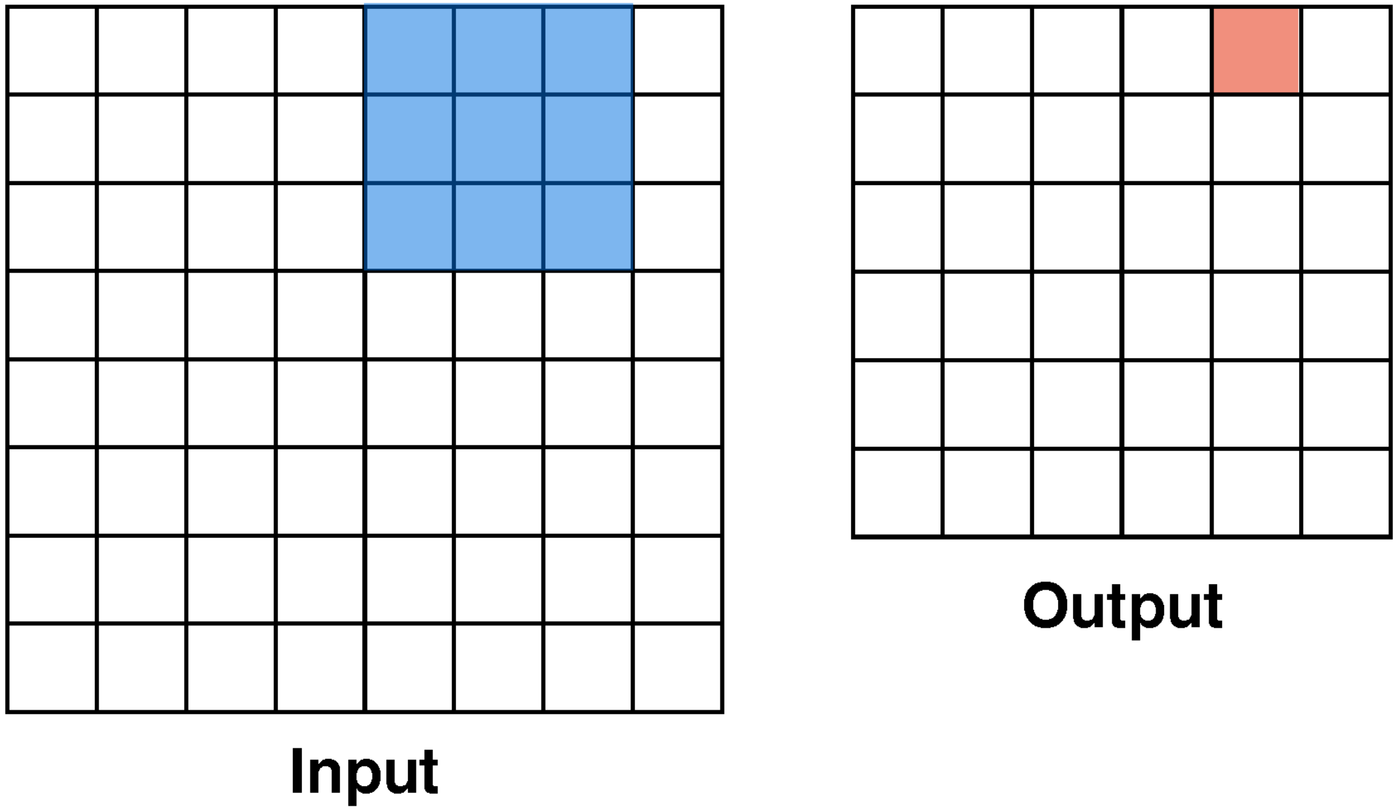
Convolution: Stride

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



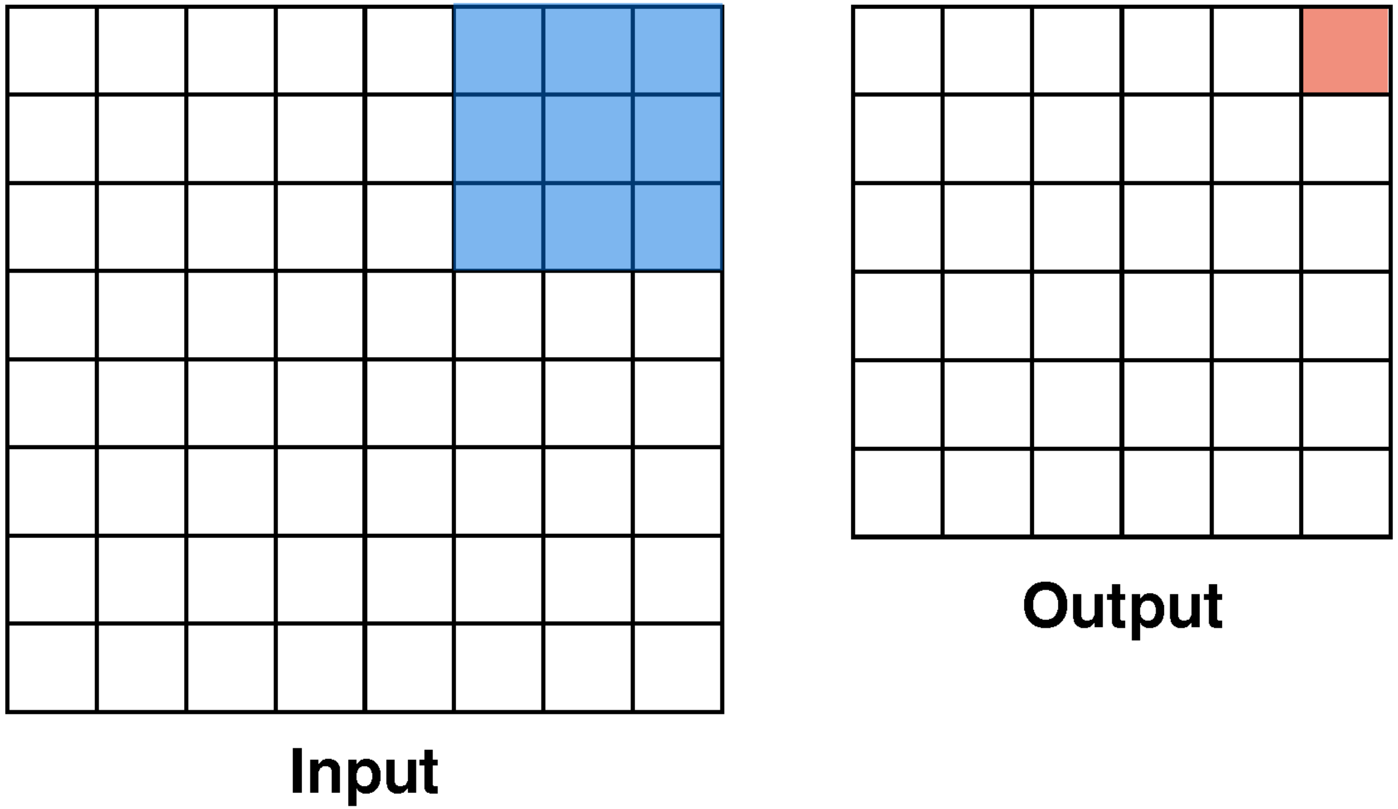
Convolution: Stride

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



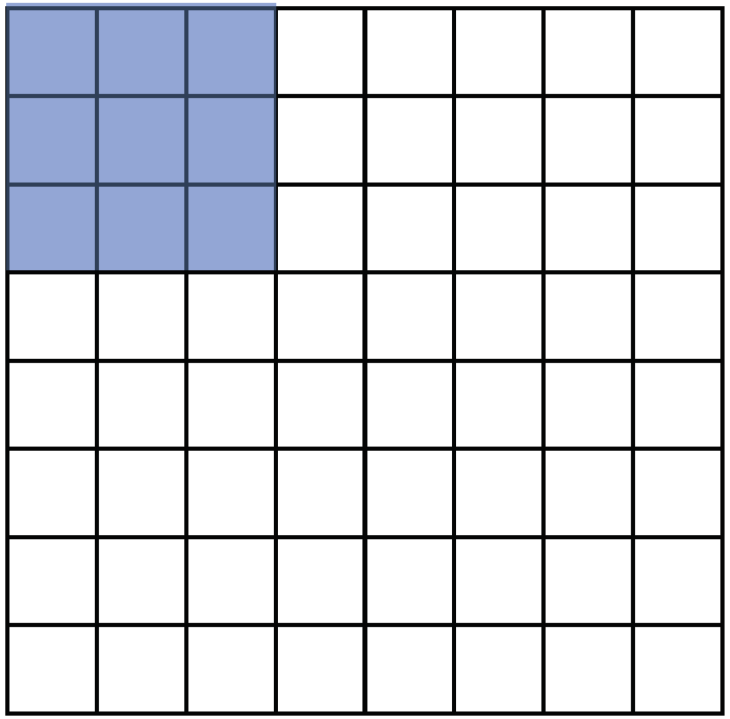
Convolution: Stride

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



Convolution: Stride

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



Input

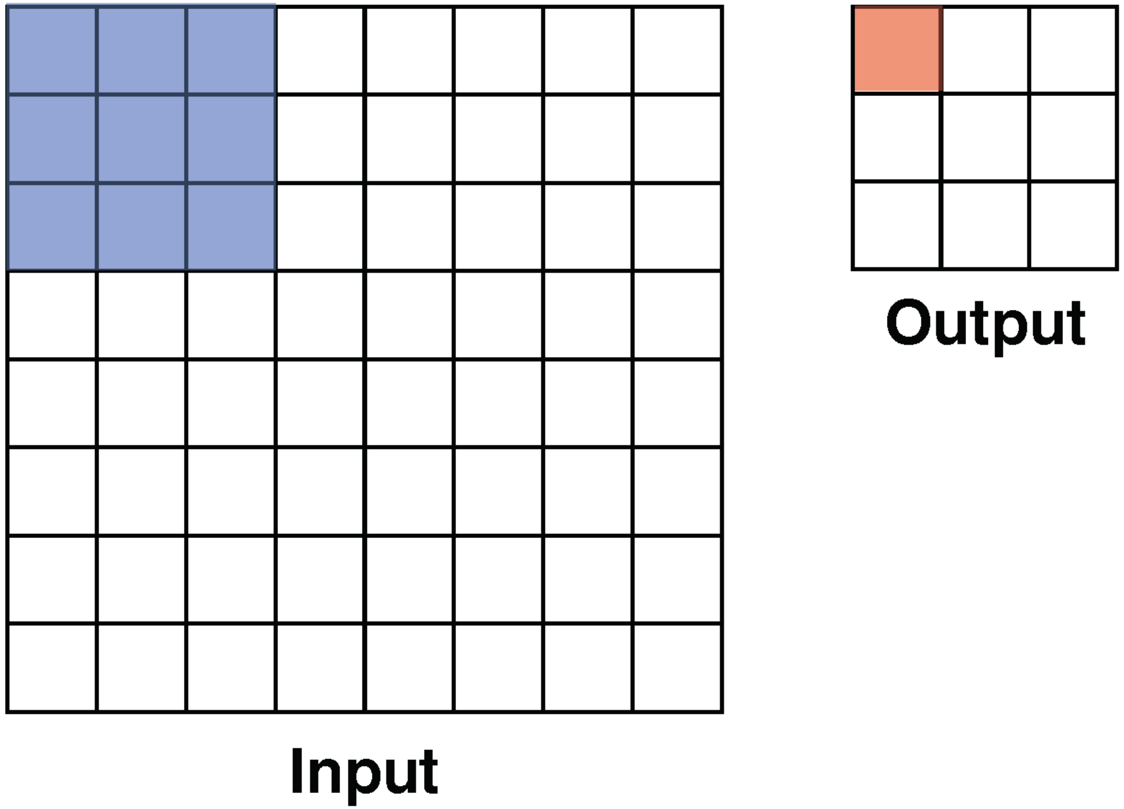
Recall that at each position, we are doing a **3D** sum:

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

(channel, row, column)

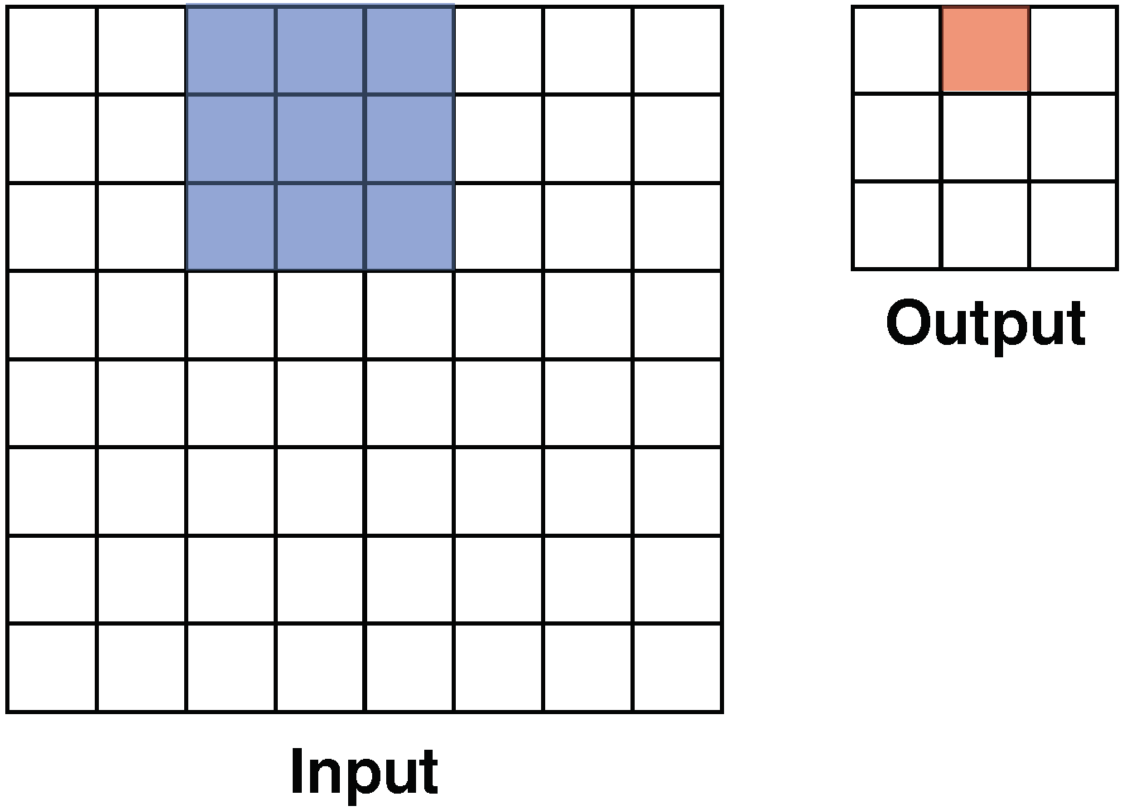
Convolution: Stride

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



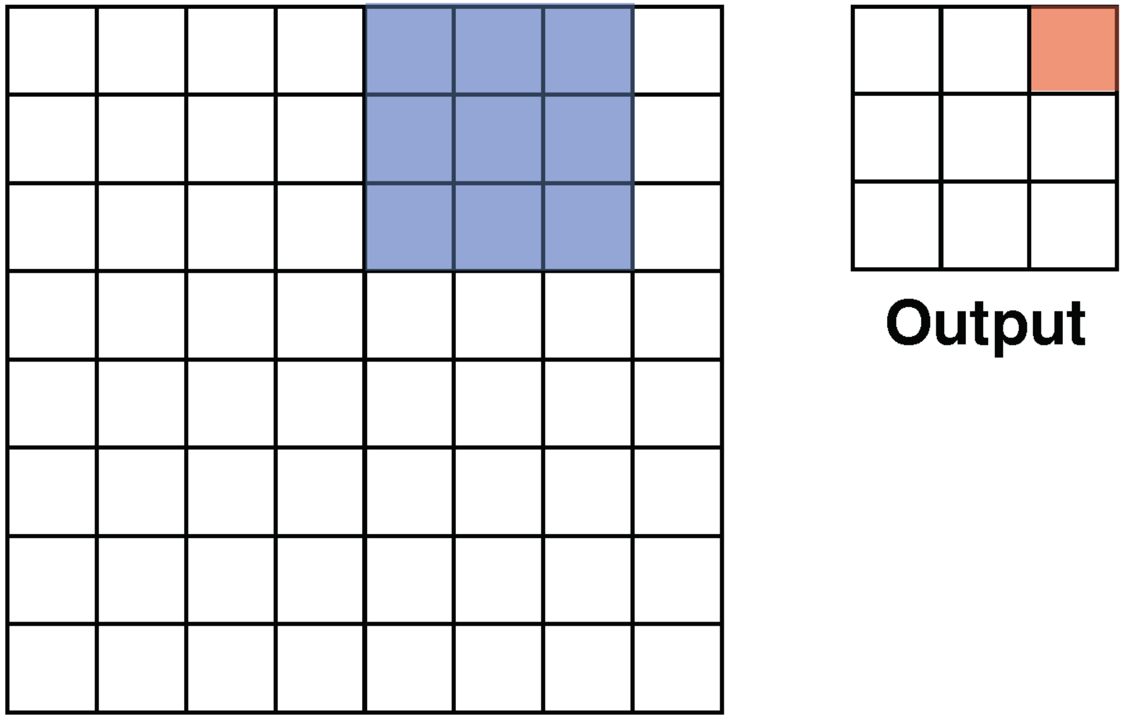
Convolution: Stride

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



Convolution: Stride

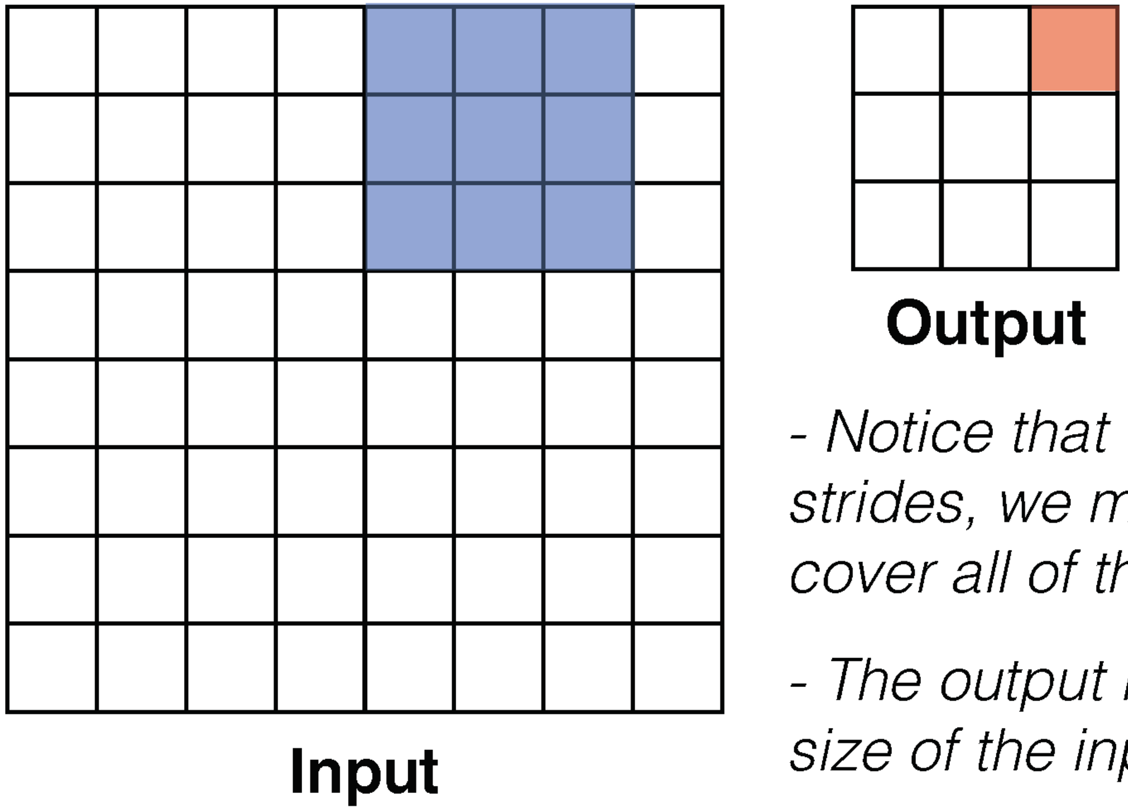
But we can also convolve with a **stride**, e.g. stride = 2



Input

Convolution: Stride

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

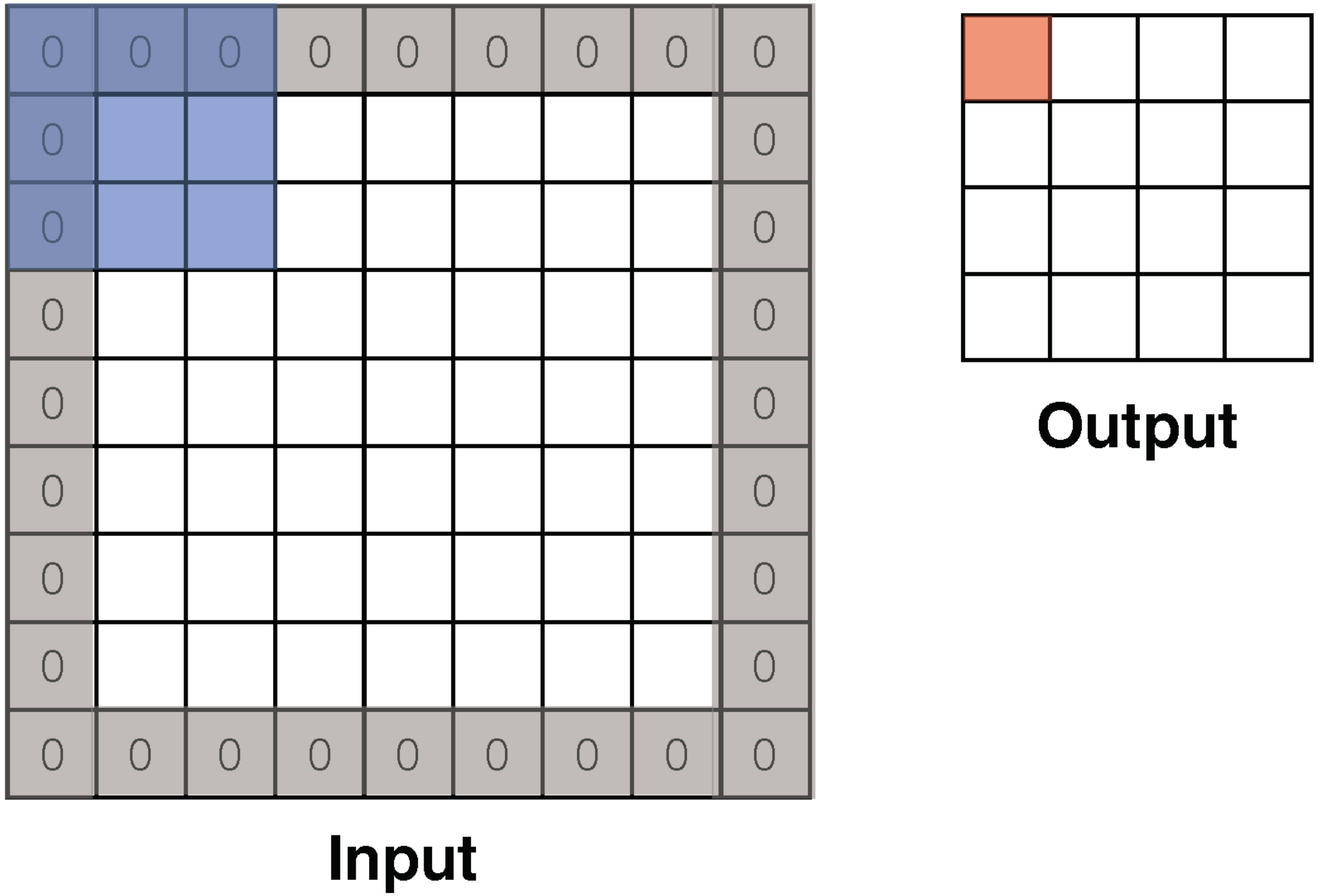


- *Notice that with certain strides, we may not be able to cover all of the input*

- *The output is also half the size of the input*

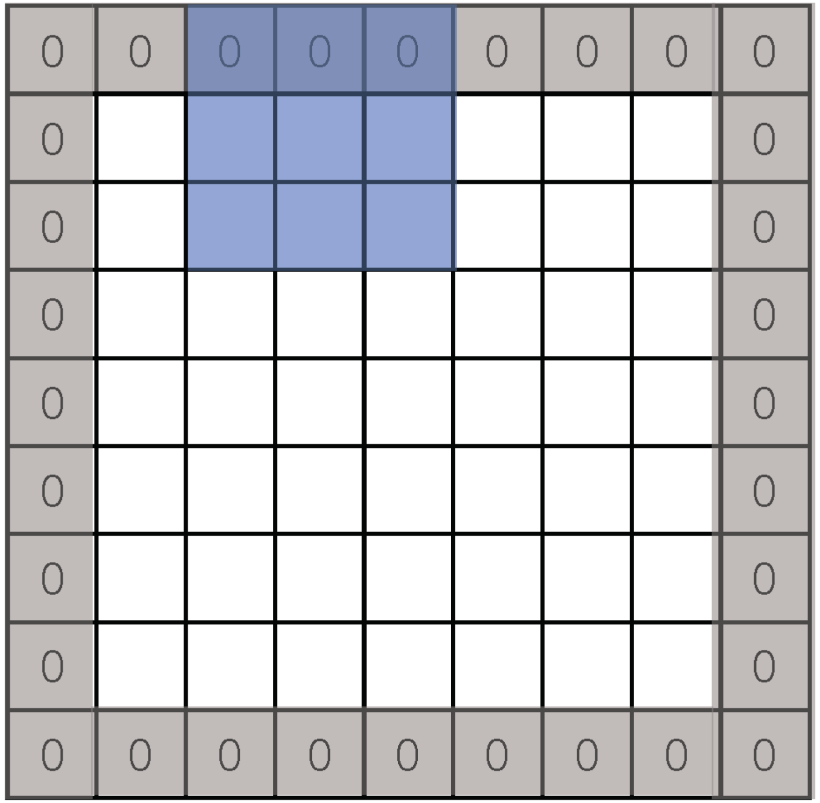
Convolution: Padding

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

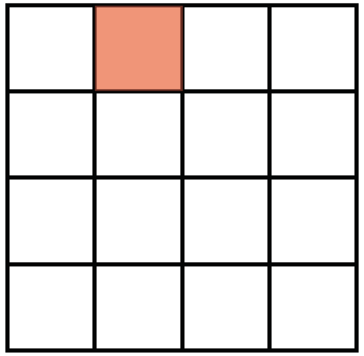


Convolution: Padding

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



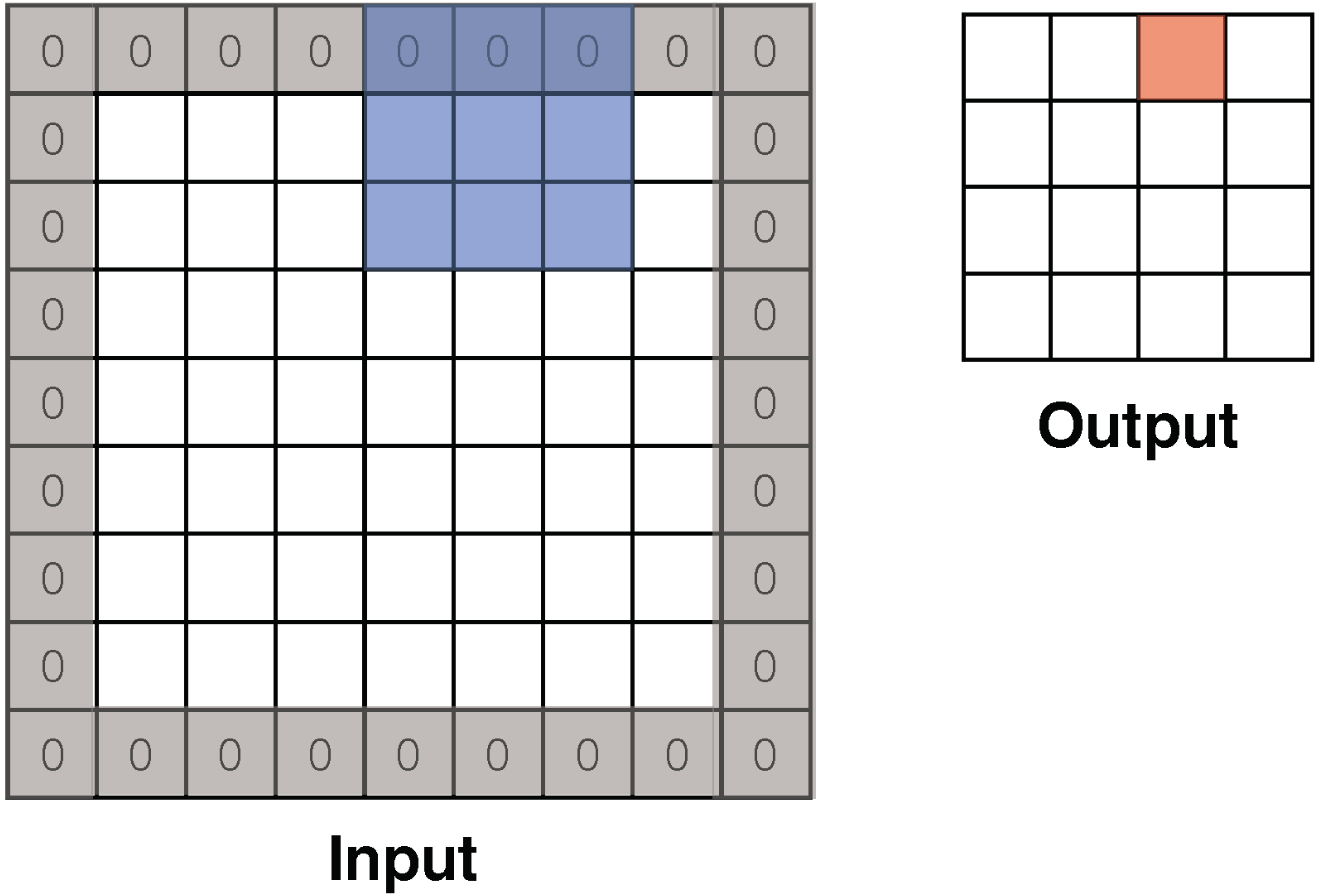
Input



Output

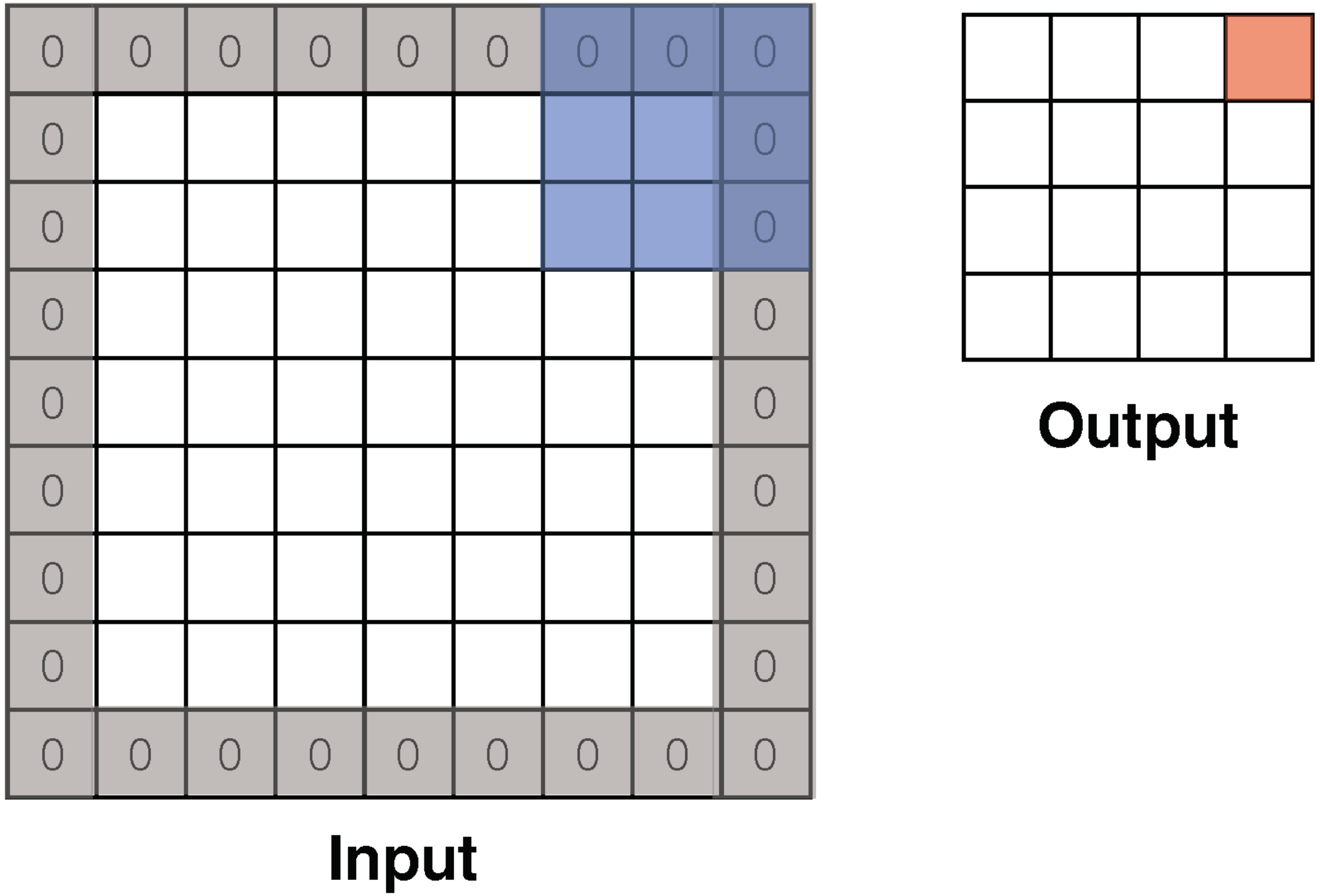
Convolution: Padding

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



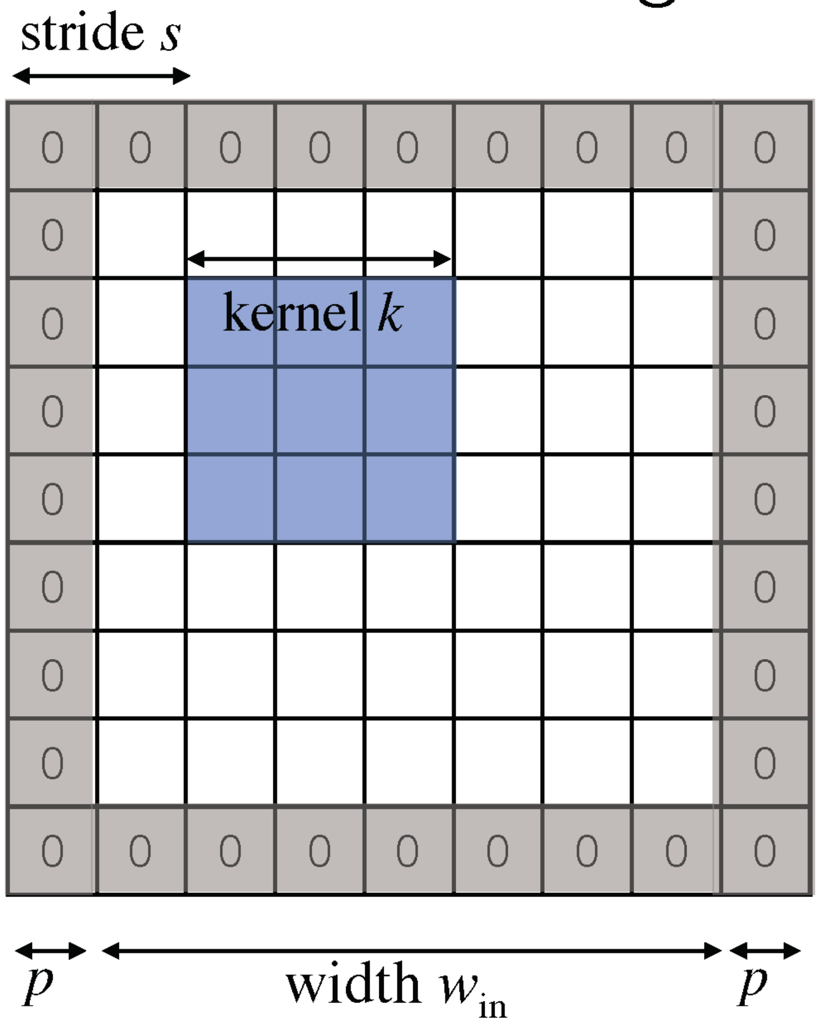
Convolution: Padding

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



Convolution:

How big is the output?

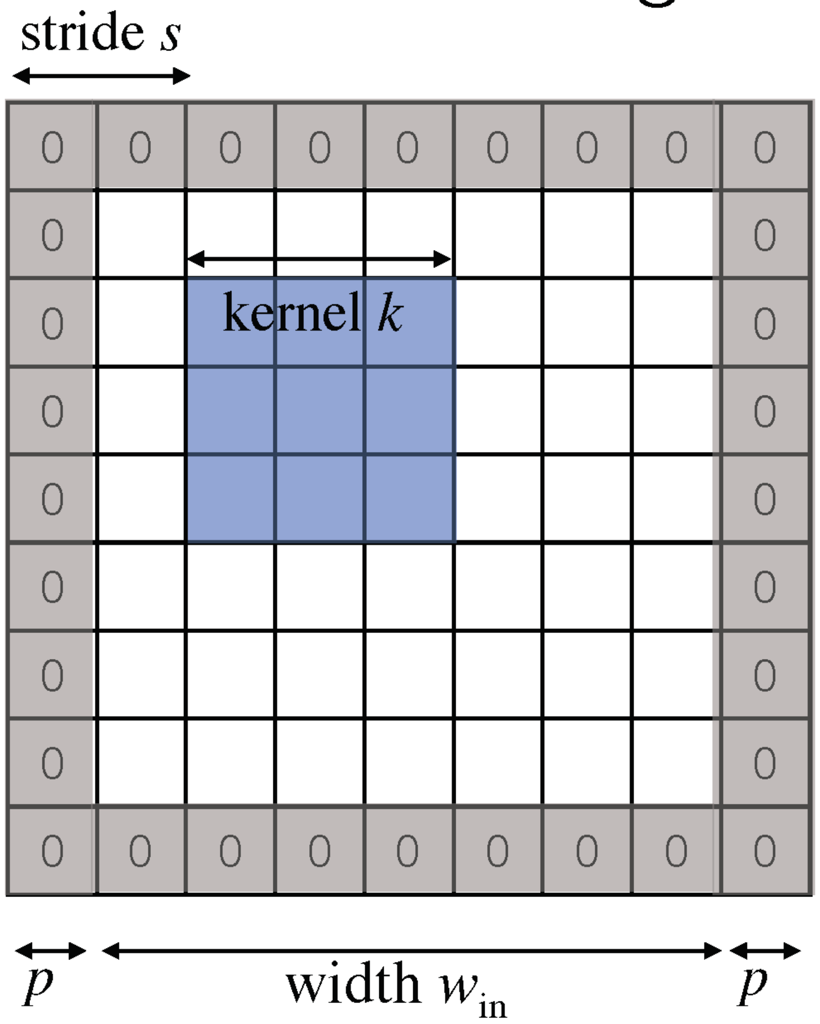


In general, the output has size:

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

Convolution:

How big is the output?



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

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

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

$$= w_{\text{in}}$$

VGGNet [Simonyan 2014] uses filters of this shape

1x1 Convolution

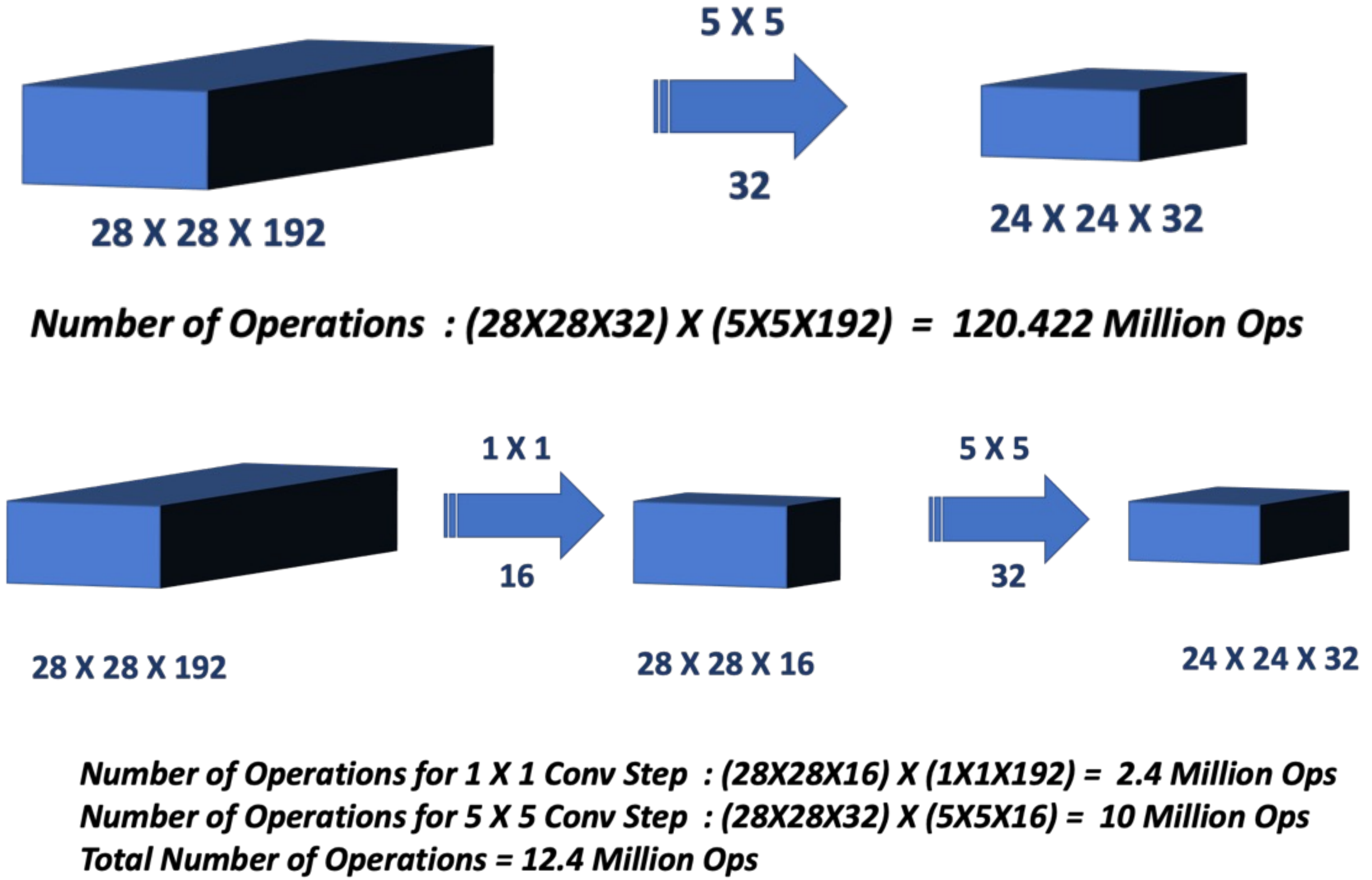
- If signals are 2-dimensional, 1x1 convolutions do not make sense. Why?

- However, in ConvNets this is not the case because one must remember that we operate over *3-dimensional volumes*, and that the filters always extend through the full depth of the input volume.

- if the input is [32x32x3] then doing 1x1 convolutions would effectively be doing 3-dimensional dot products (since the input depth is 3 channels).
- “mixing information across channels”

1x1 Convolution: a computationally cheap method

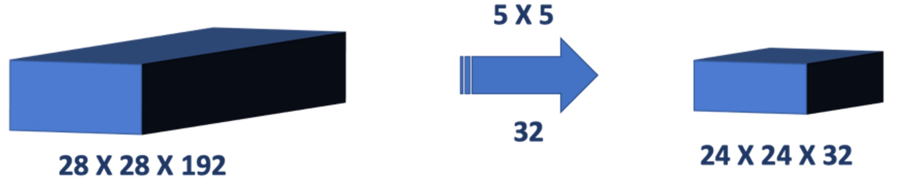
- Eg 28x28x192 -> 24x24x32



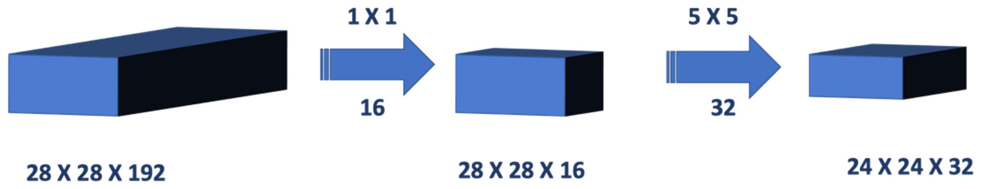
Take 3min: discuss with your neighbor:
A) What's the difference of a 1x1 conv to a fully-connected layer?
B) When is this 1x1 conv approach good, when is not good?

1x1 Convolution: a computationally cheap method

- Eg 28x28x192 -> 24x24x32



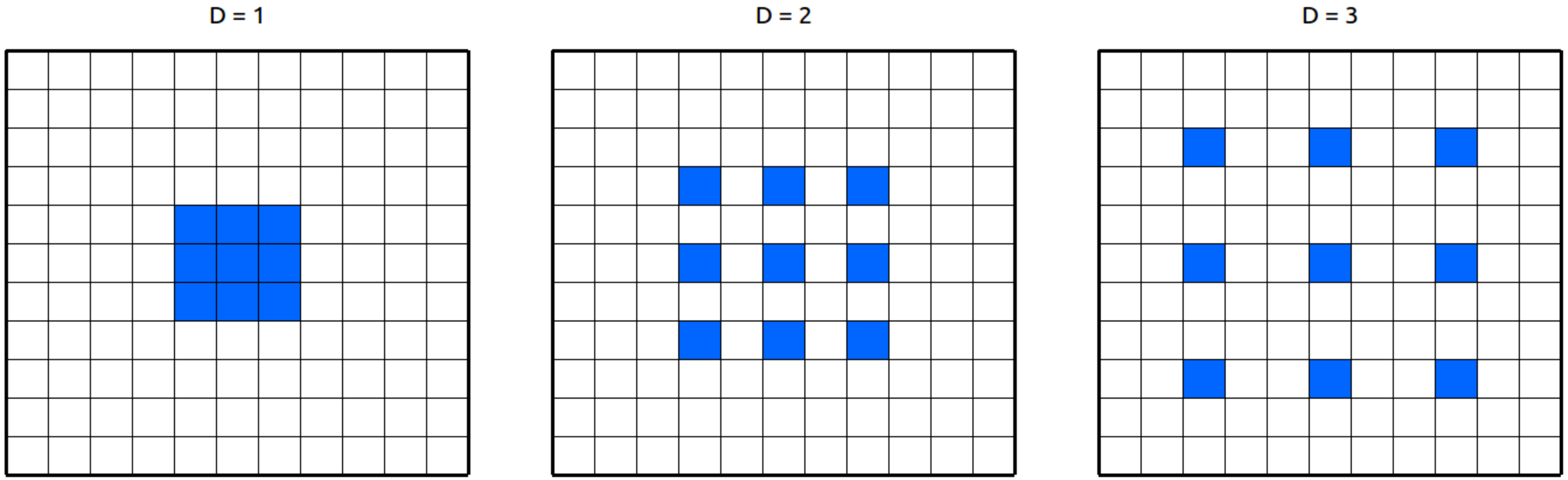
Number of Operations : (28X28X32) X (5X5X192) = 120.422 Million Ops



Number of Operations for 1 X 1 Conv Step : (28X28X16) X (1X1X192) = 2.4 Million Ops
Number of Operations for 5 X 5 Conv Step : (28X28X32) X (5X5X16) = 10 Million Ops
Total Number of Operations = 12.4 Million Ops

Dilated Convolutions

- enlarge receptive field but keep the the parameter number



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?

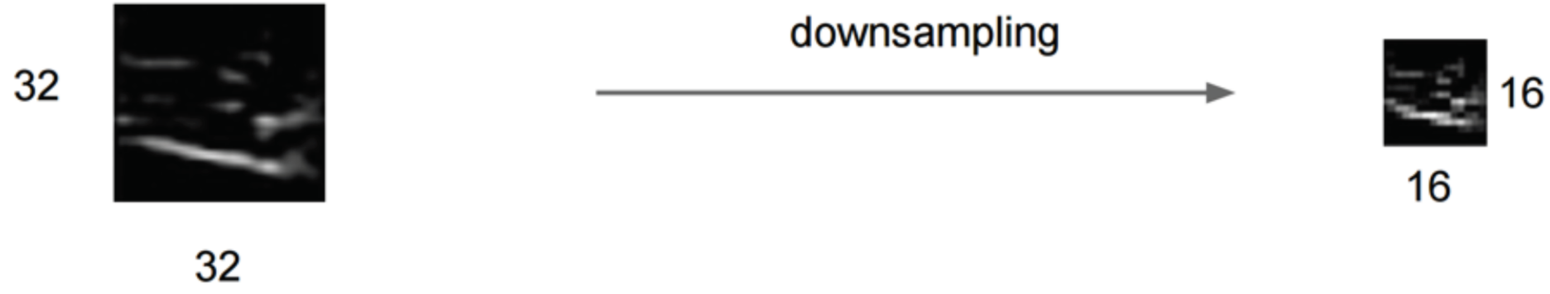
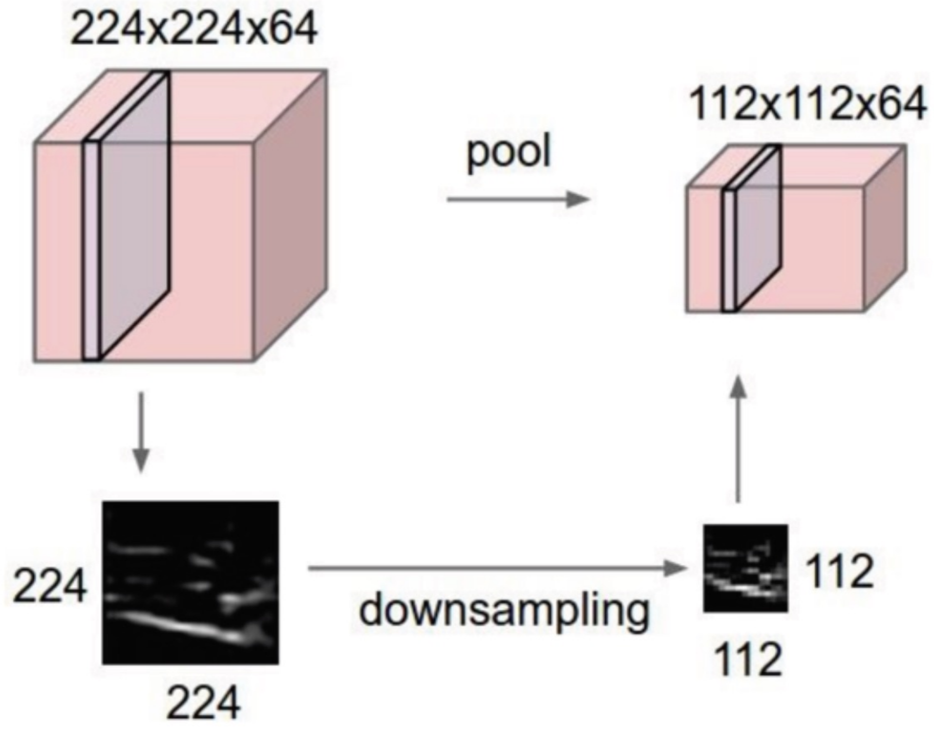


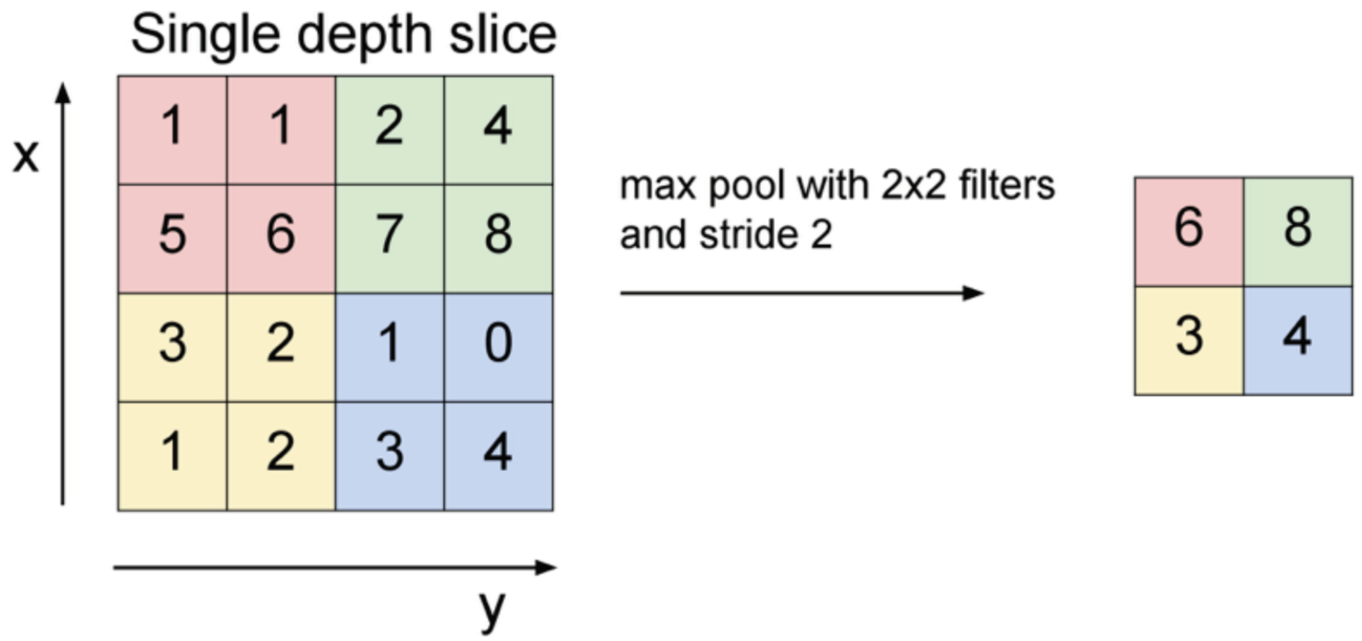
Figure: Andrej Karpathy

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

Getting rid of pooling

- To reduce the size of the representation
  - using larger stride in conv layer once in a while

- Discarding pooling layers has also been found to be important in training good generative models
  - variational autoencoders (VAEs)
  - generative adversarial networks (GANs)
- ”Isotropic” models, see Vision Transformer (Lecture 7)

Example ConvNet

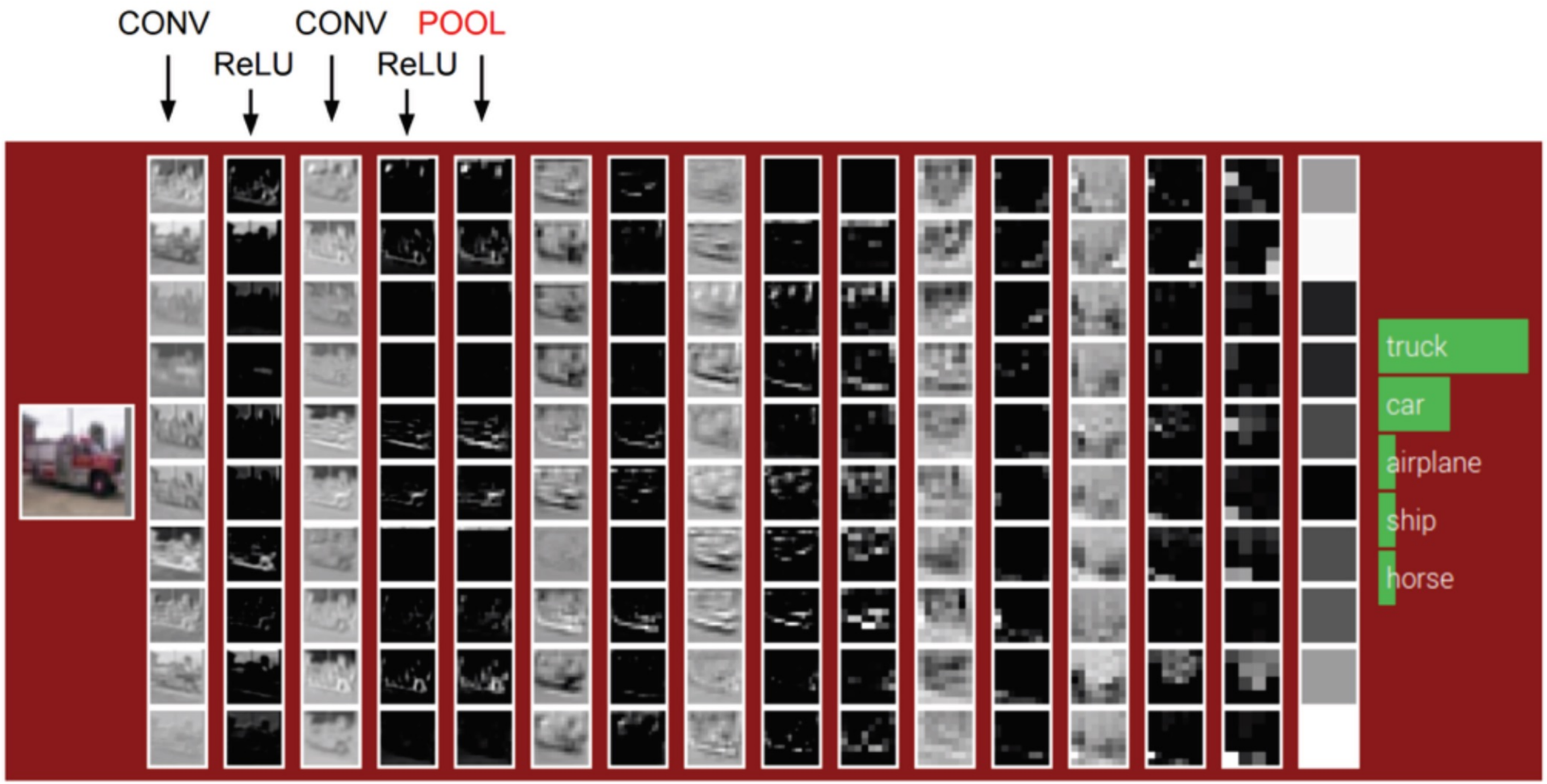


Figure: Andrej Karpathy

Example ConvNet

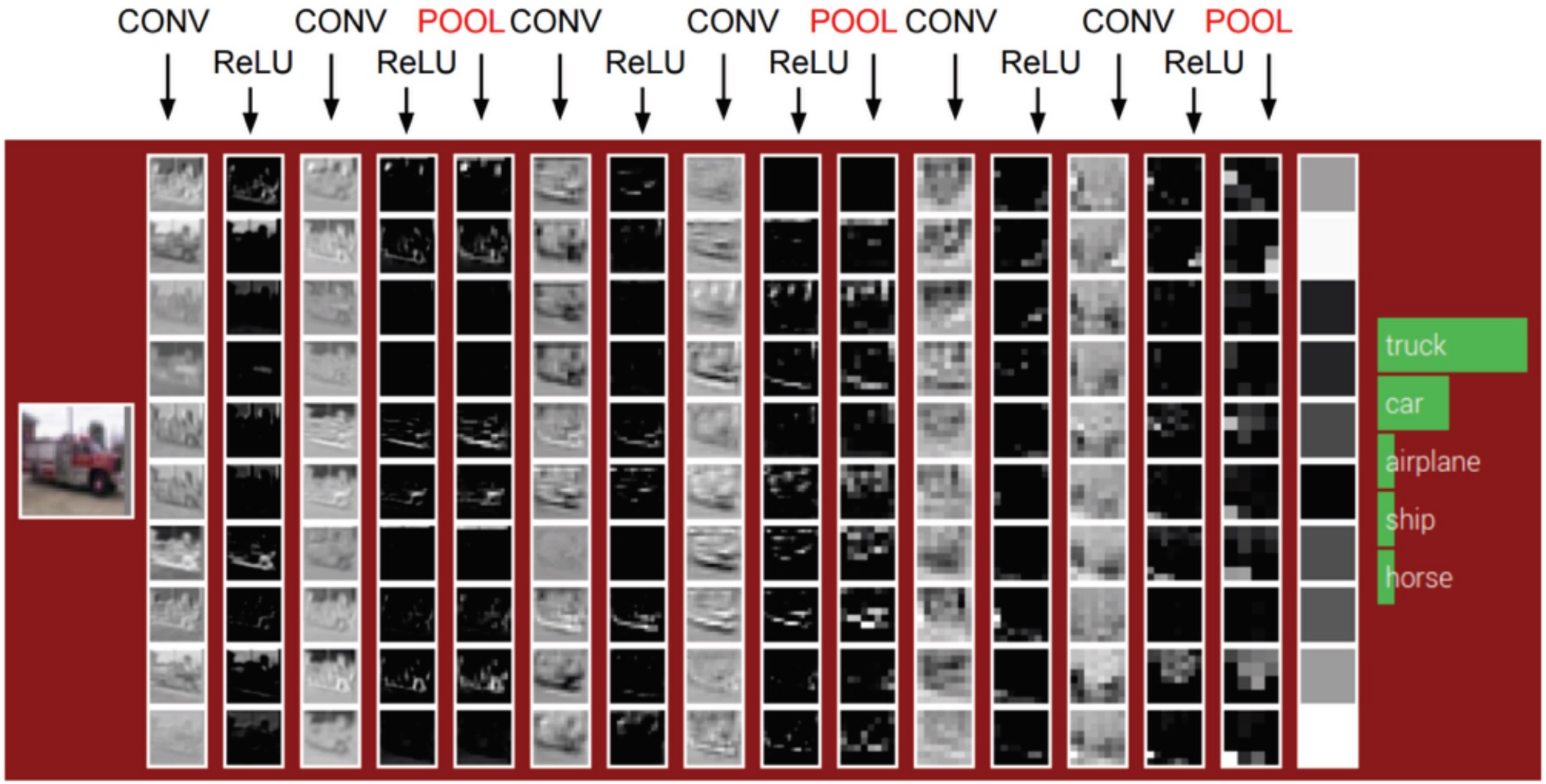


Figure: Andrej Karpathy

Example ConvNet

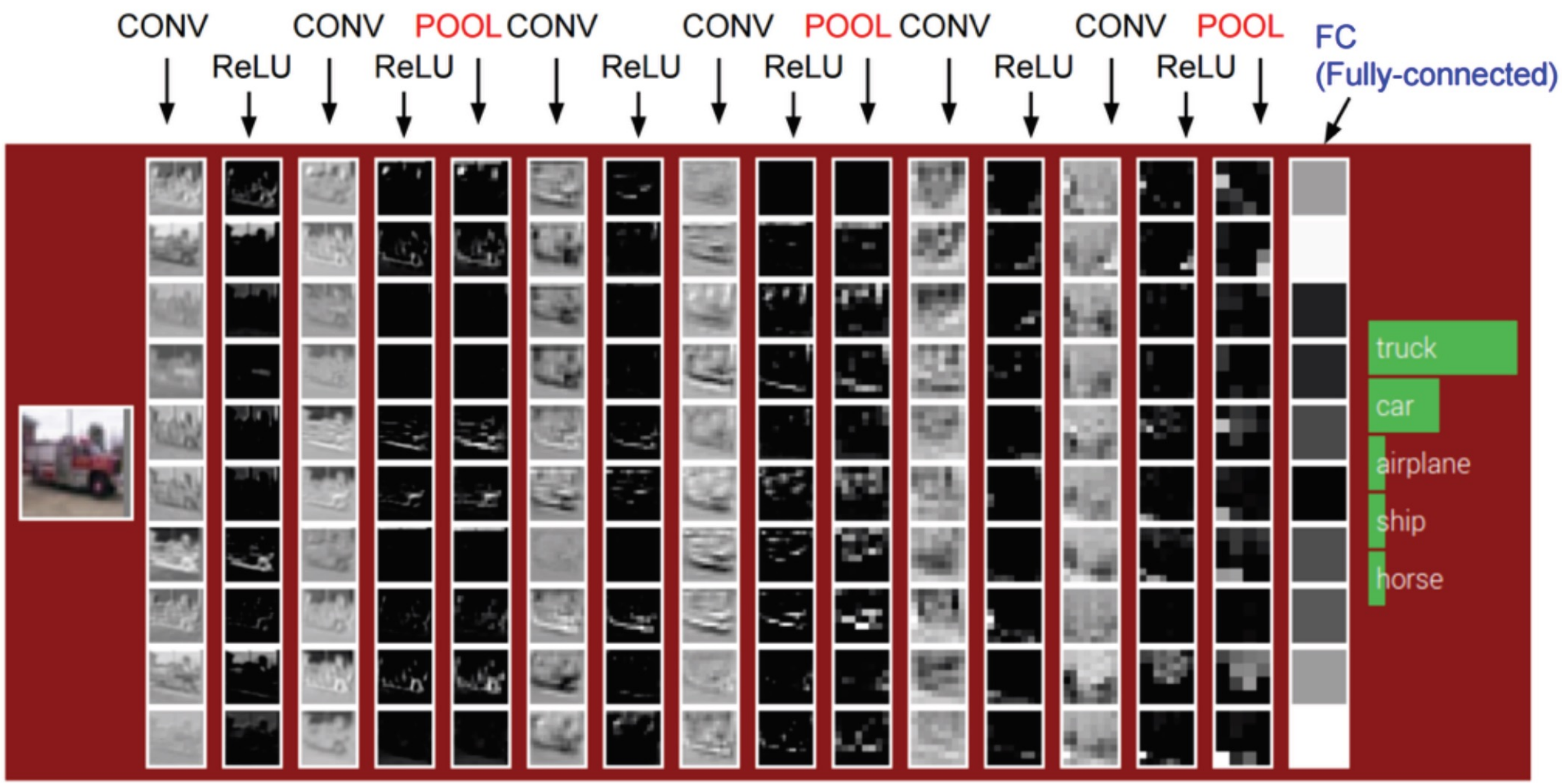
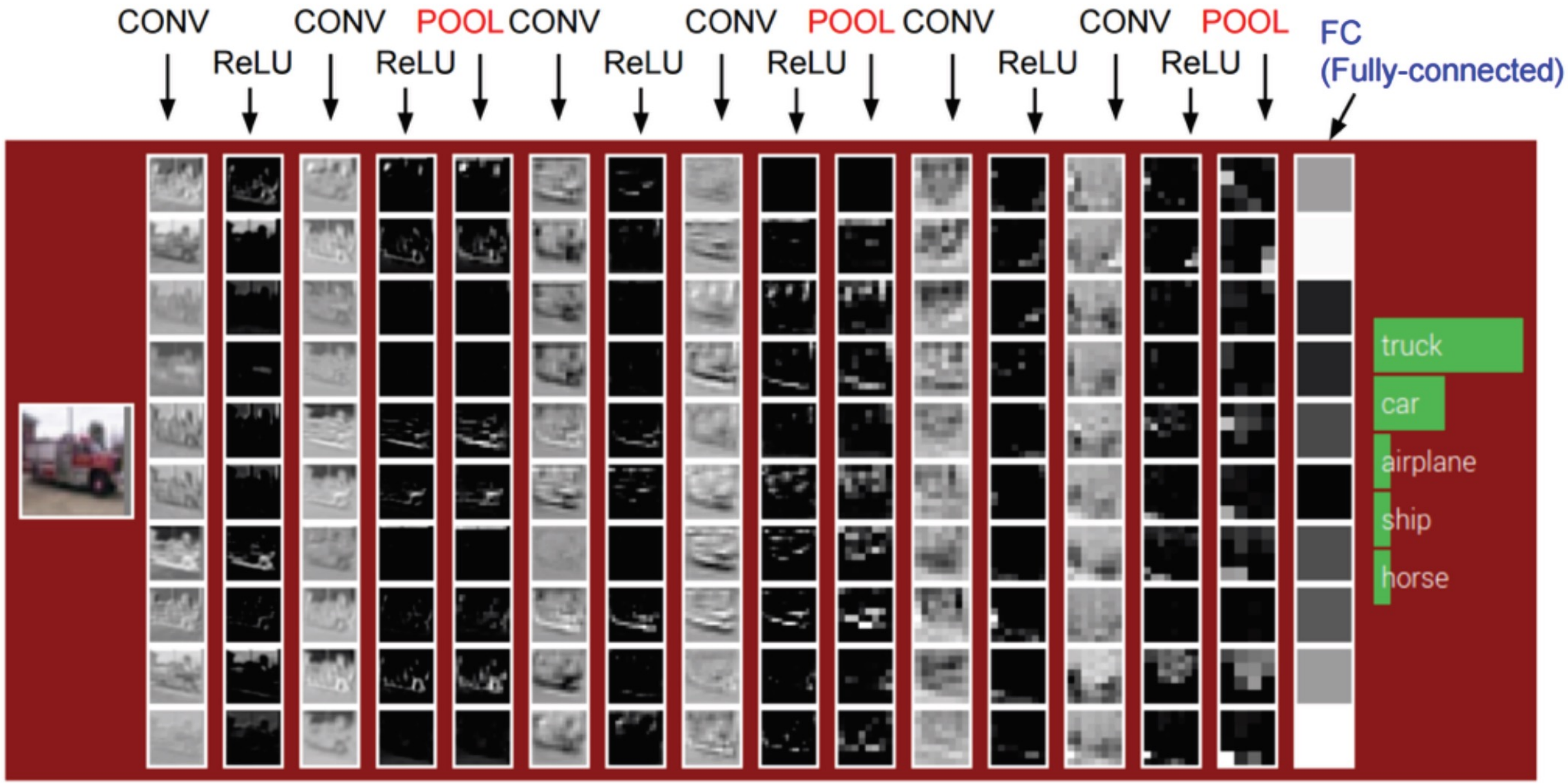


Figure: Andrej Karpathy

Example ConvNet

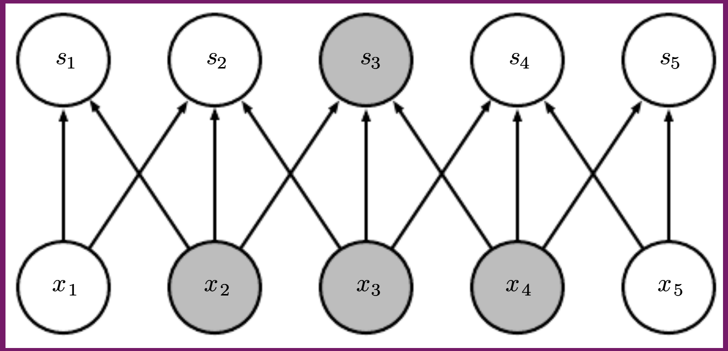


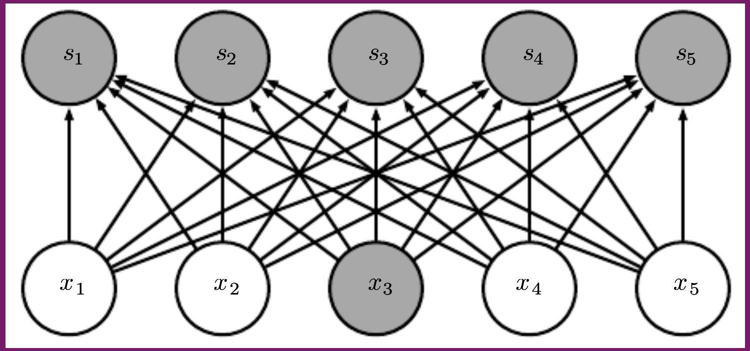
10x3x3 conv filters, stride 1, pad 1

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

How research gets done part 4

Previous parts:
[fundamental understanding, read papers, how-to-read-papers, implement & tinker with code]

Today:

- Be curious.
- **“The most exciting phrase to hear in science, the one that heralds new discoveries, is not ‘Eureka!’ but ‘That’s funny…’” --- *Isaac Asimov***
- “That’s funny…” --> there’s a delta between what’s expect *vs.* what’s observed.
  - For this: need to be able to trust a) your experiment, b) understand broader context to form expectation
  - (see also incongruity theory re: why jokes are funny)
  - (see also Hubel & Wiesel from earlier)
- For this you need to be able to hold two opposing thoughts/ideas in your head (at least for a while)
- The advice: Don’t ignore this moment. Instead follow this up by *structured tinkering (next part)*

[kinda like this]

ConvNet Case Study I: Alexnet

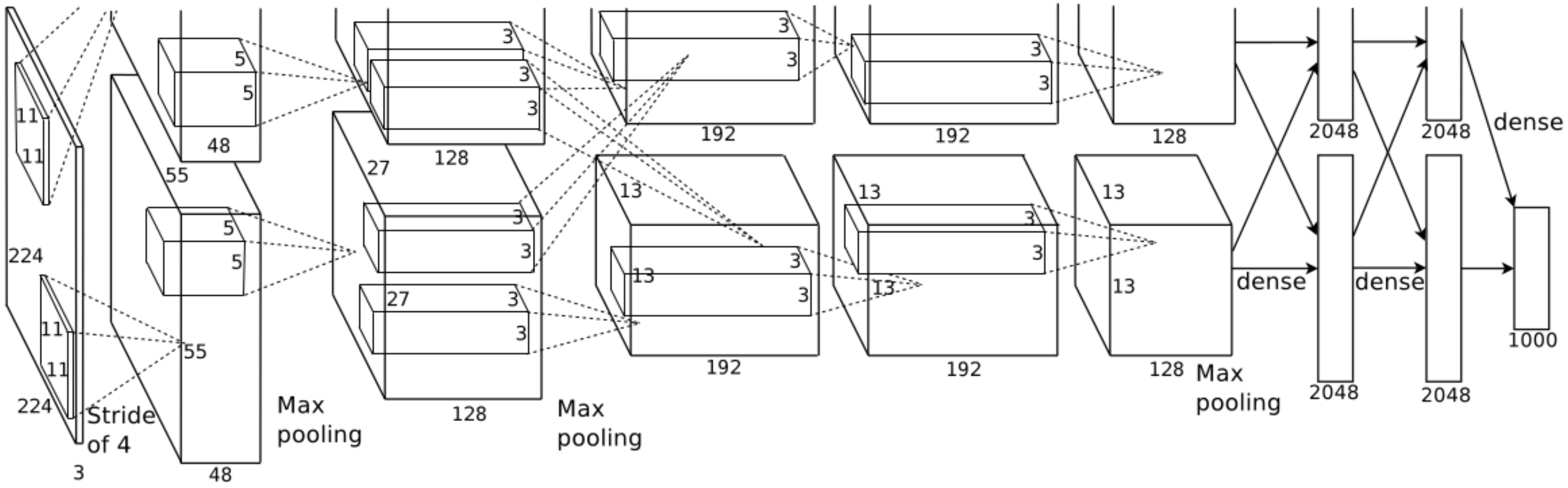
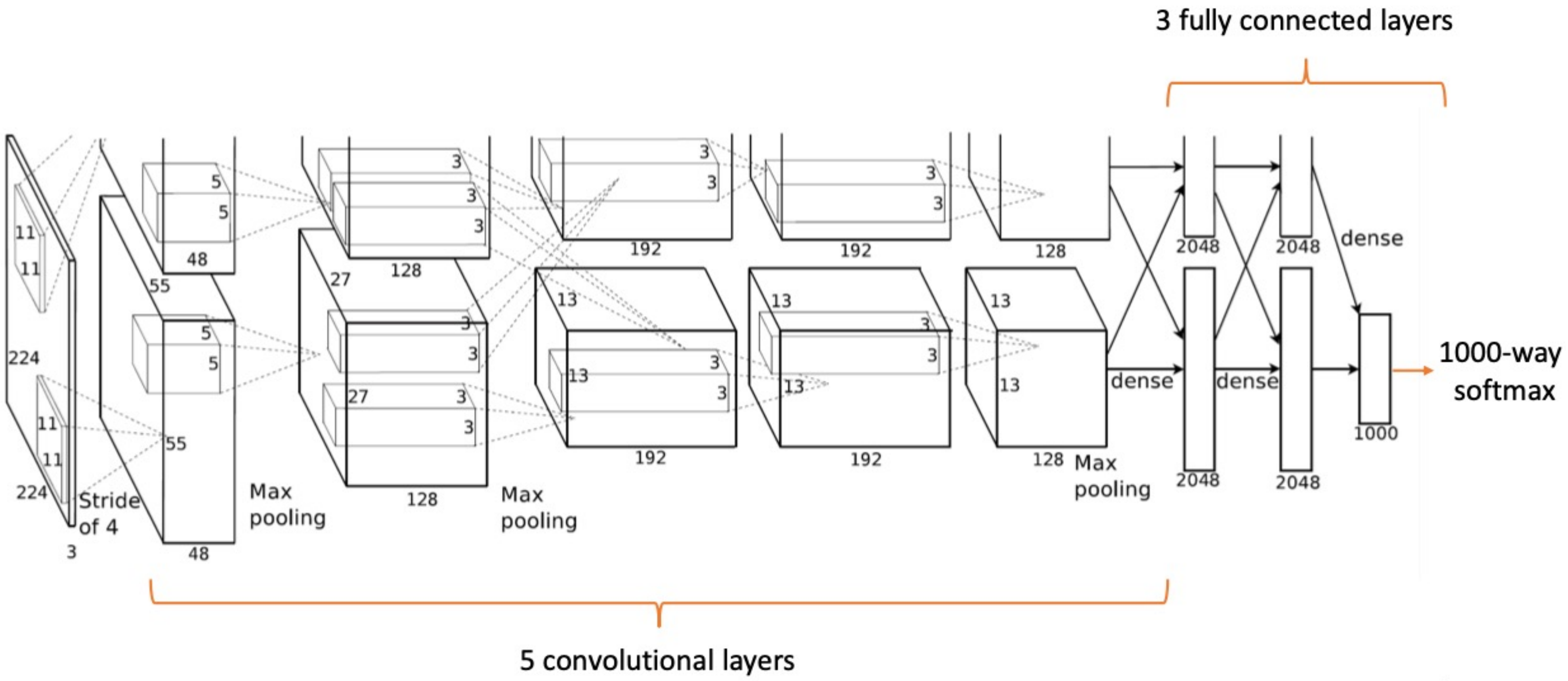


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input 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



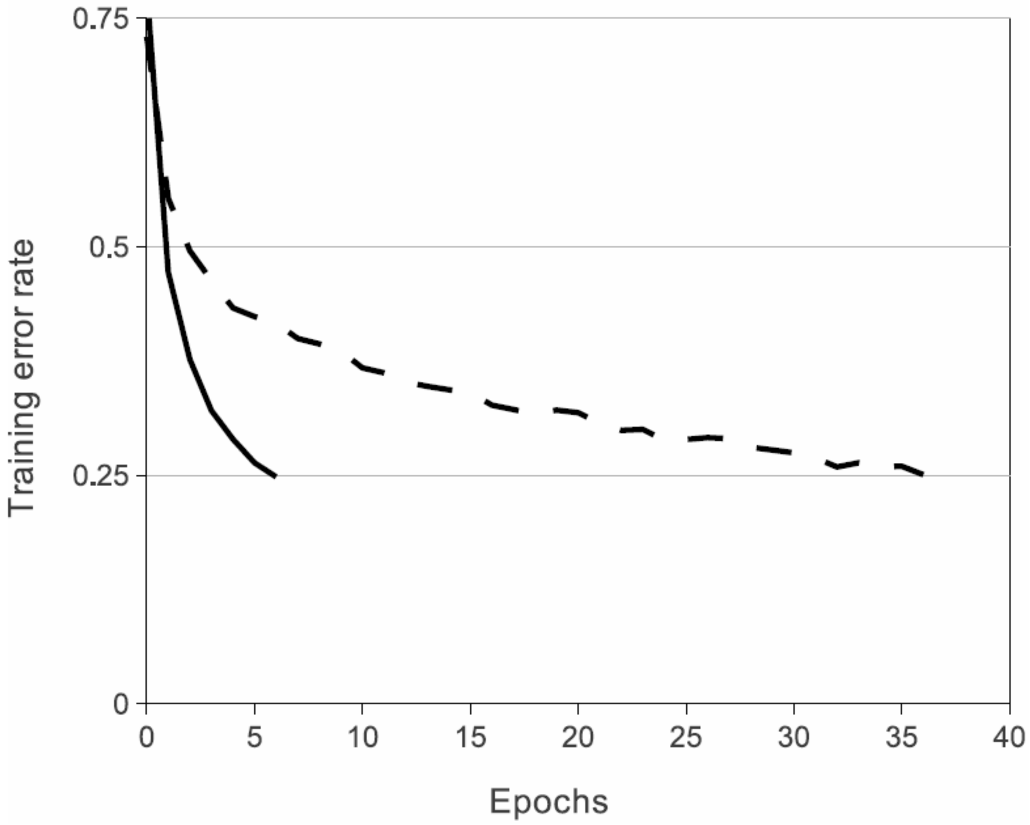
AlexNet

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

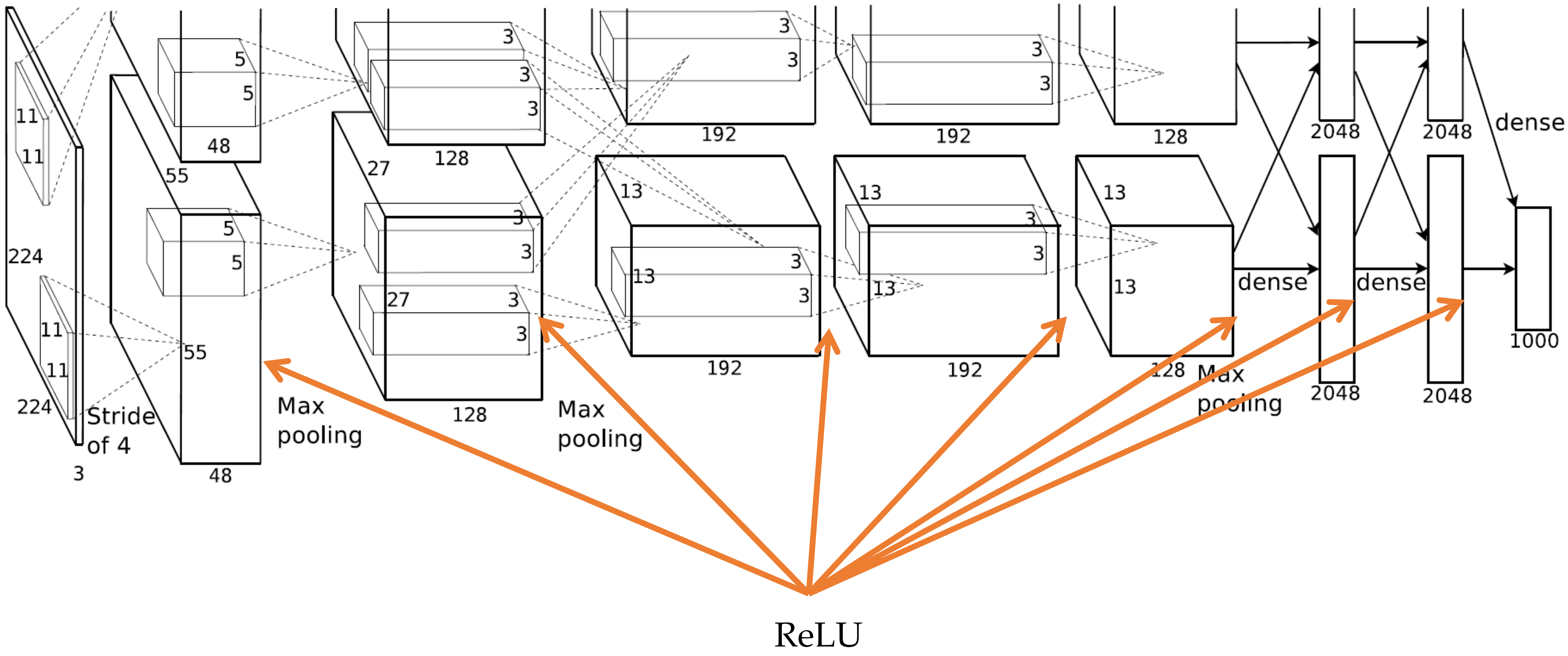
Activation function

- 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)



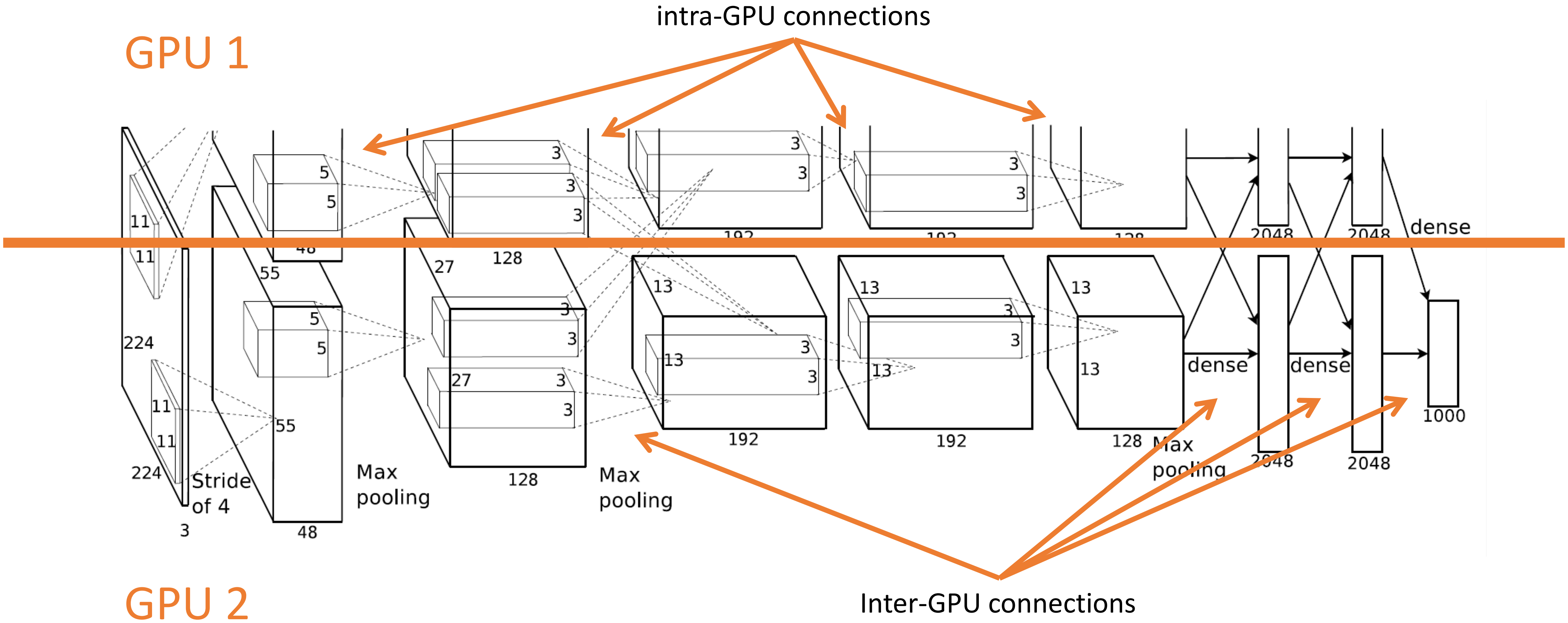
Activation function



Training with multiple GPUs

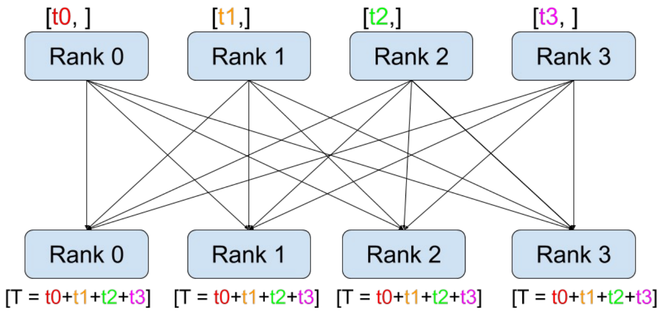
- Half of the neurons of an certain layer are on each GPU
- GPUs communicate only in certain layers
- Improvement (as compared with a net with half as many kernels in each convolutional layer trained on one GPU):
  - top-1 error rate by 1.7%
  - top-5 error rate by 1.2%
- ! Back then.. No Pytorch/JAX/TensorFlow/Caffe/Torch… (manual coding in CUDA)

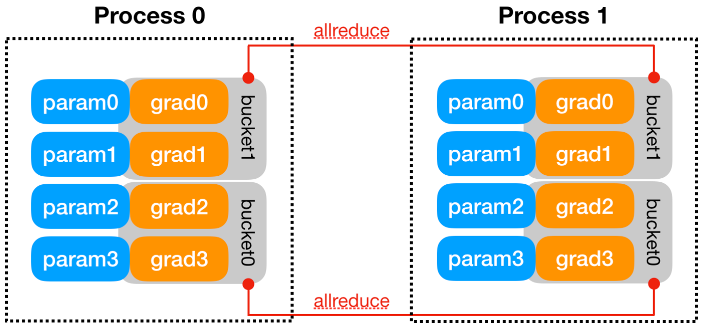
Training with multiple GPUs



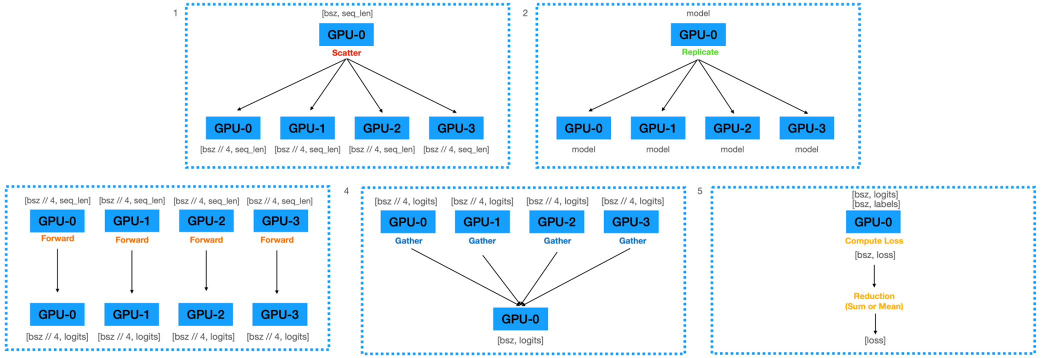
On that note: Communicating between GPUs: PyTorch

- DistributedDataParallel (AlexNet is "ModelParallel"):
  - Forward-mode: on one GPU each, same model on each GPU
  - Backward-mode: Gradients synced between GPUs
  - With All-reduce:





- Other distributed operations:



https://pytorch.org/tutorials/beginner/dist_overview.html

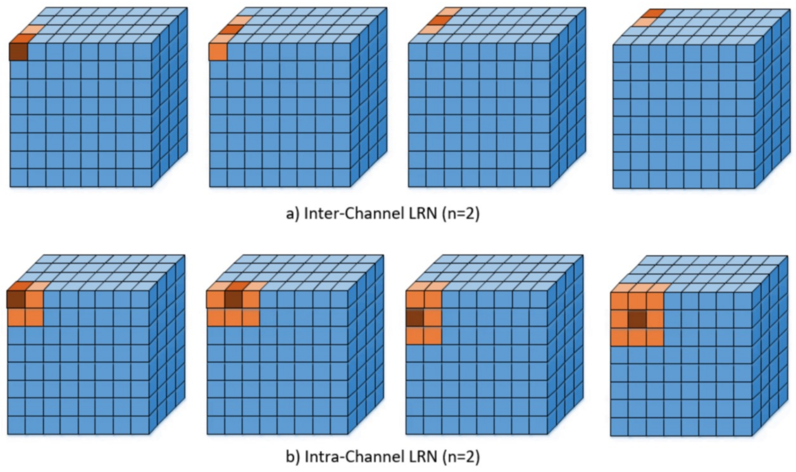
Local Response Normalization

- Local Response Normalization aids generalization (now-a-days: BN/LN)

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a^j_{x,y})^2 \right)^\beta$$

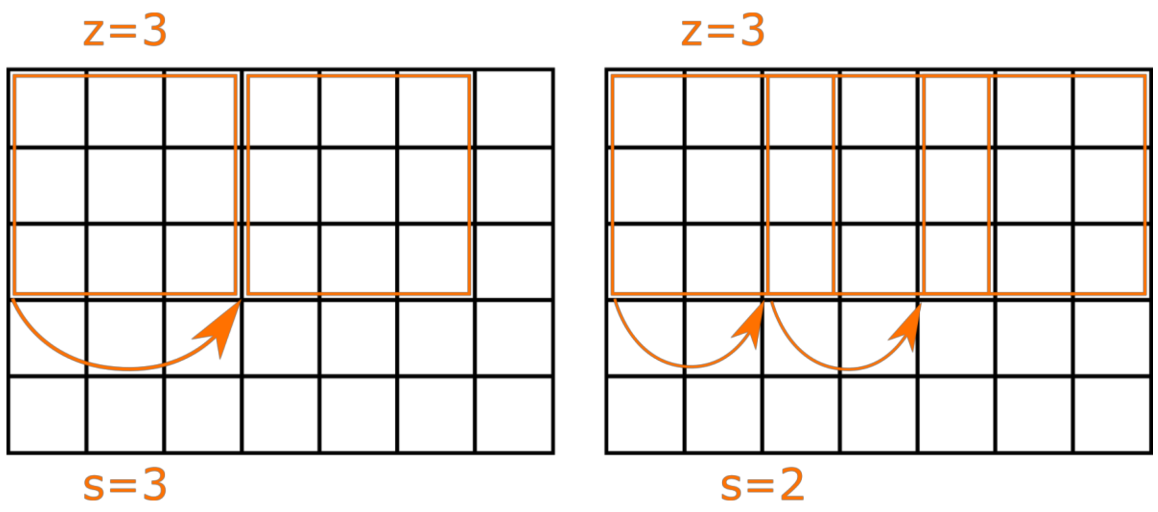
- Improvement:
  - top-1 error rate by 1.4%
  - top-5 error rate by 1.2%

two types of LRN-->

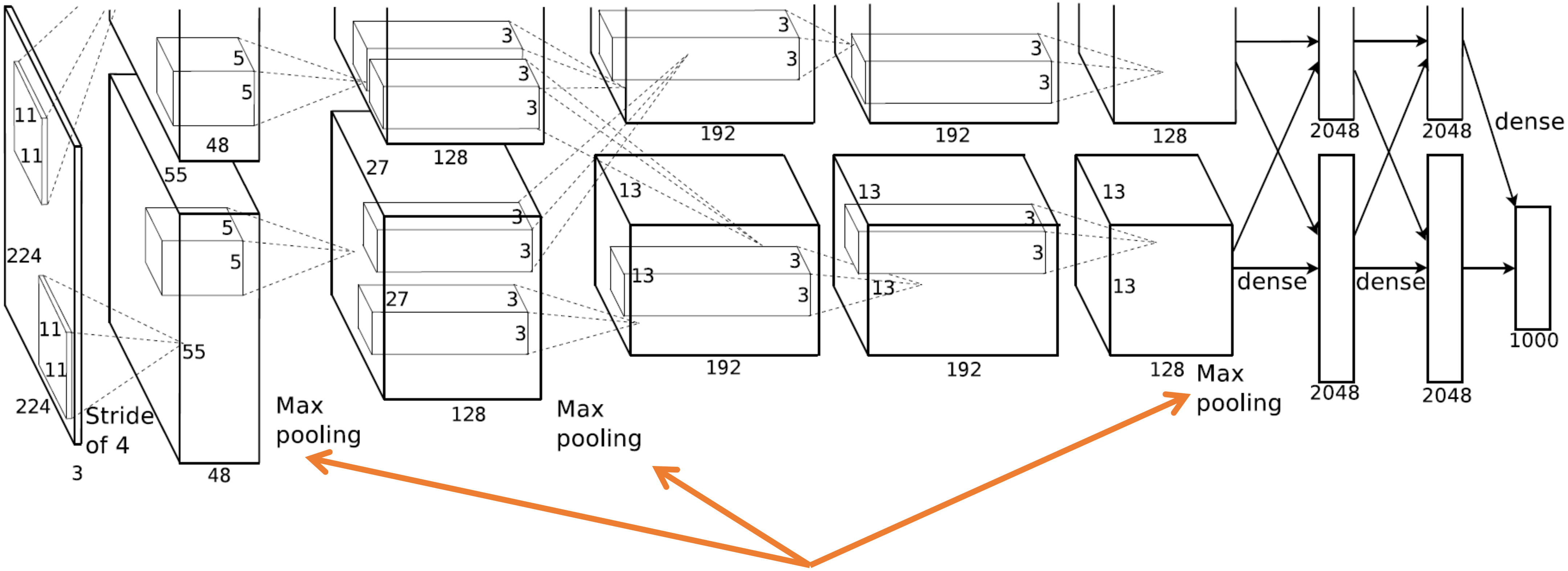


Overlapping Pooling

- Pooling layers summarize the outputs of neighbouring neurons in the same kernel map.
- Overlapping pooling: s < z
- Improvement over no-overlap:
  - top-1 error rate by 0.4%
  - top-5 error rates by 0.3%

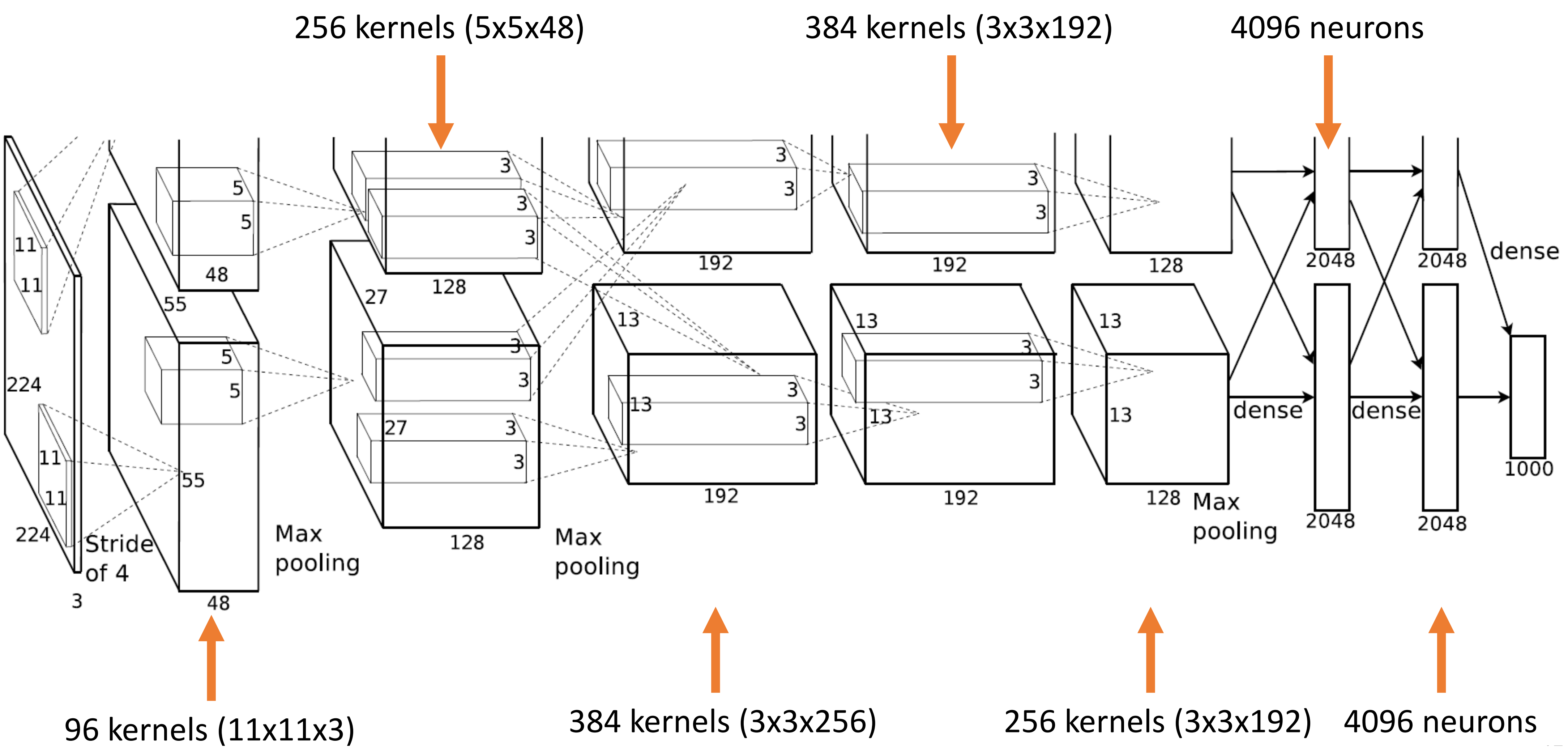


Overlapping Pooling



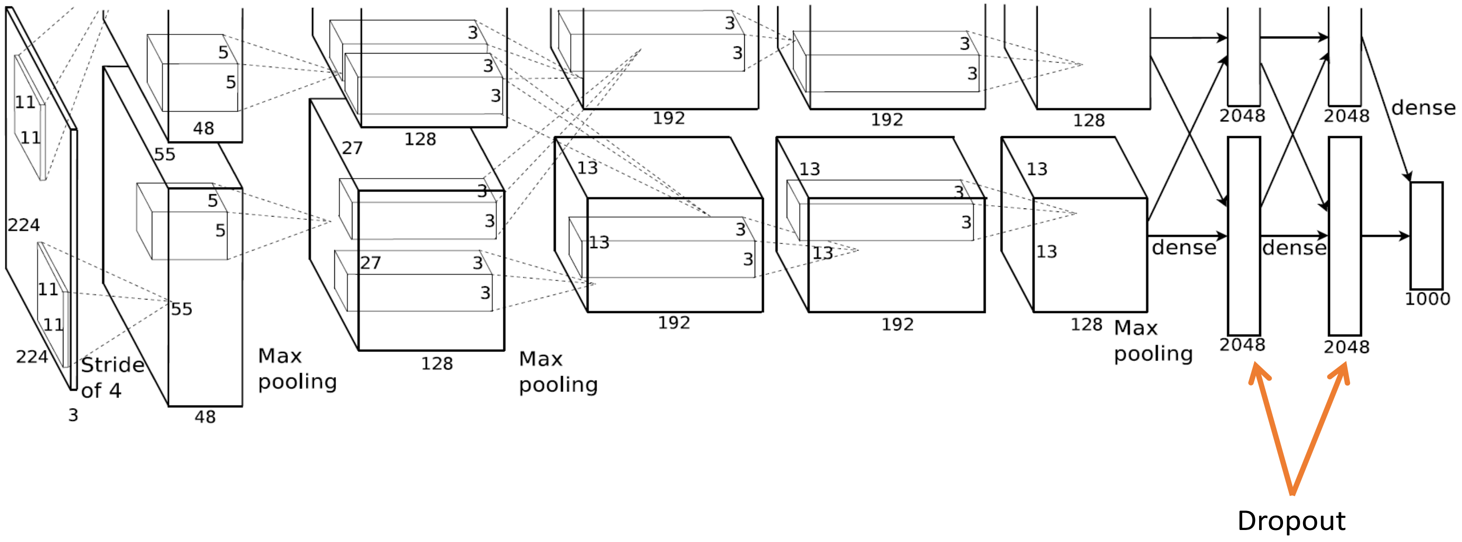
Overlapping Pooling

Overall architecture

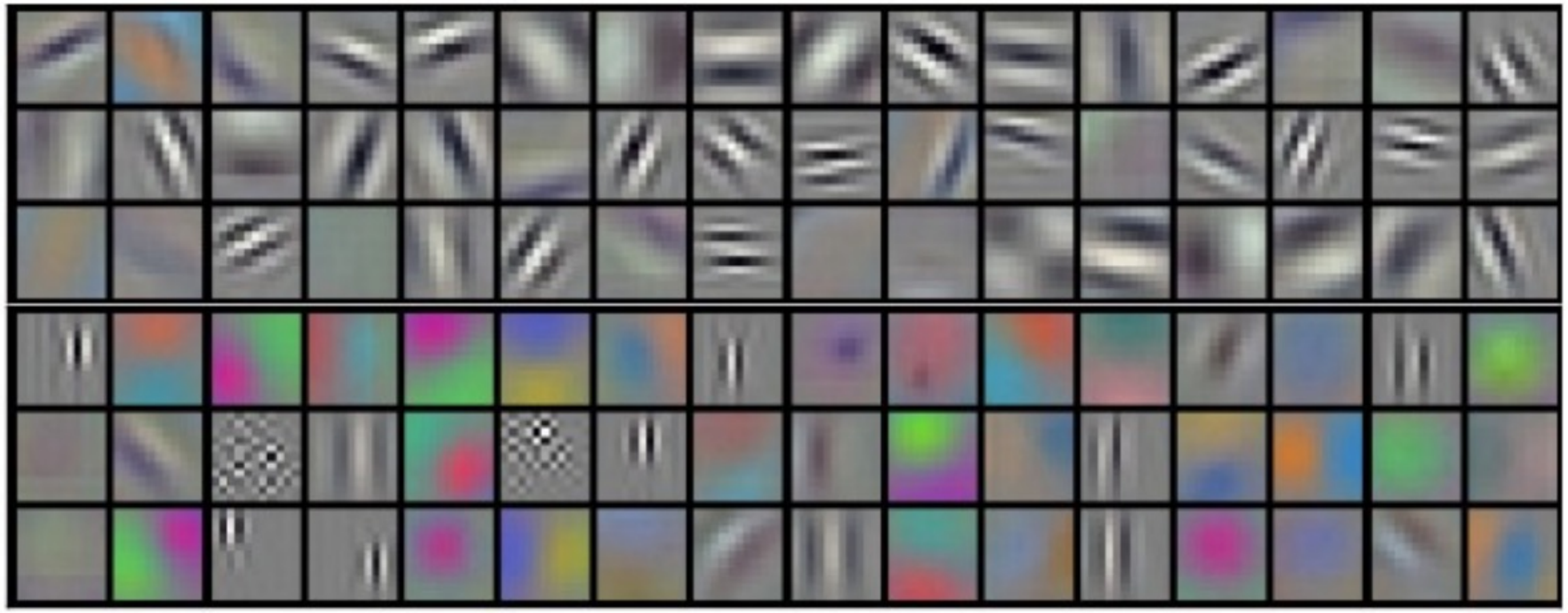


The Overfitting Problem

- 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



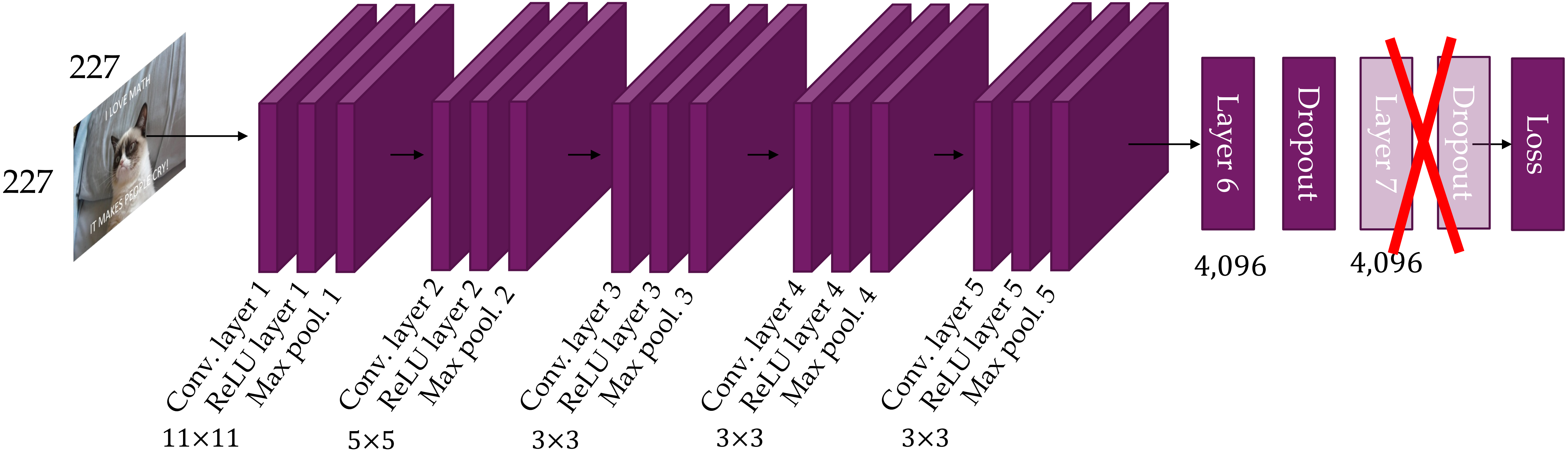
The learned filters



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.

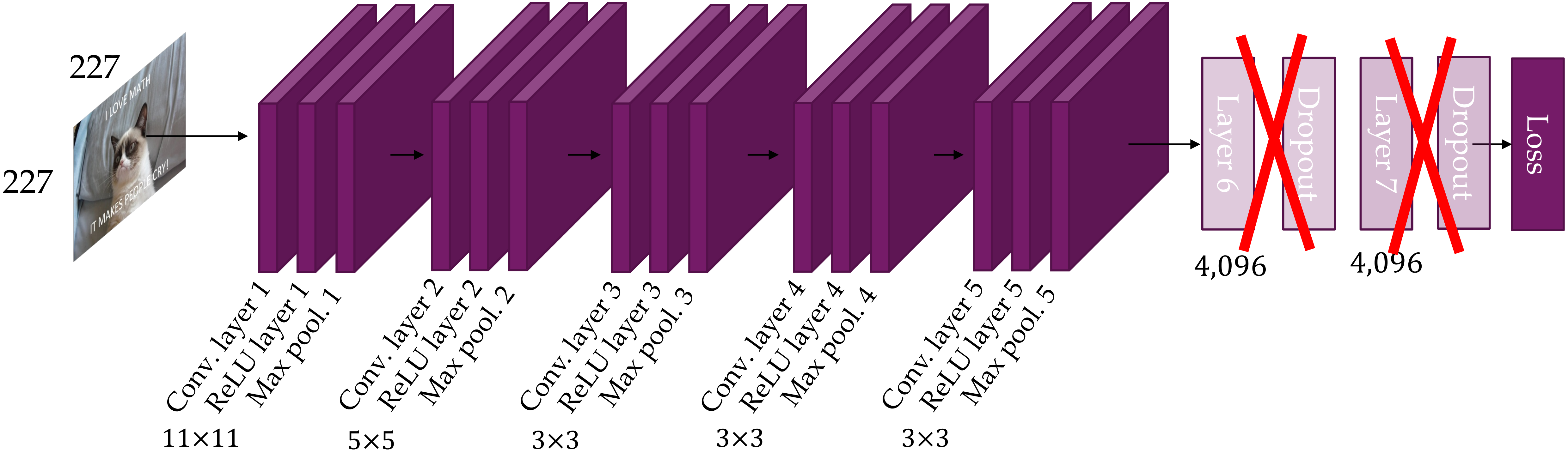
Removing layer 7

1.1% drop in performance, 16 million less parameters



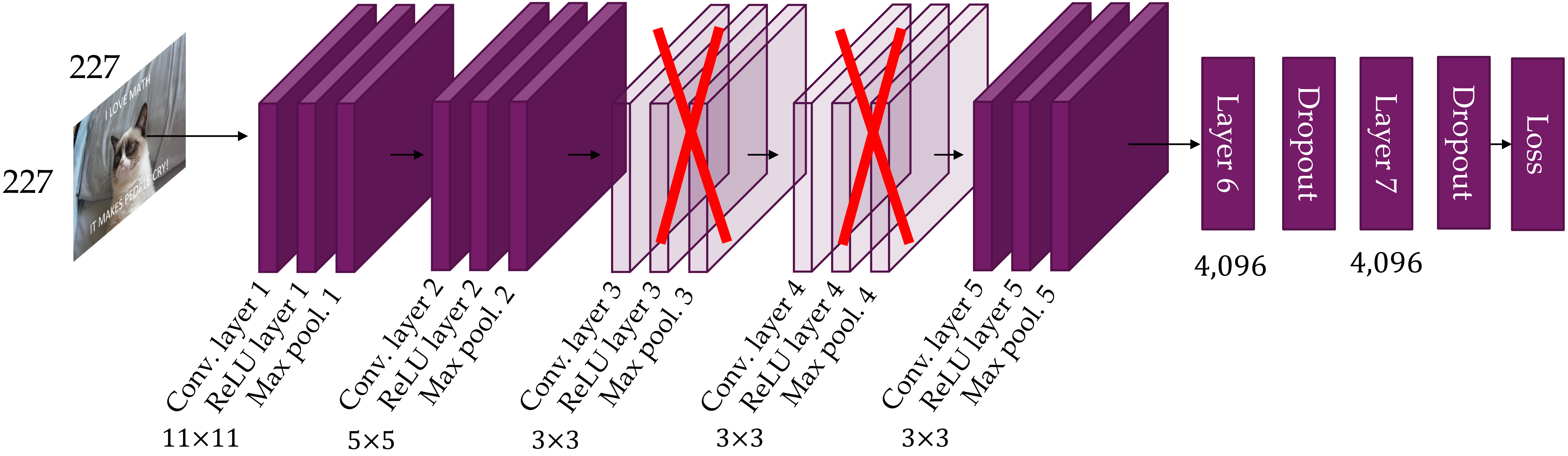
Removing layer 6, 7

5.7% drop in performance, 50 million less parameters



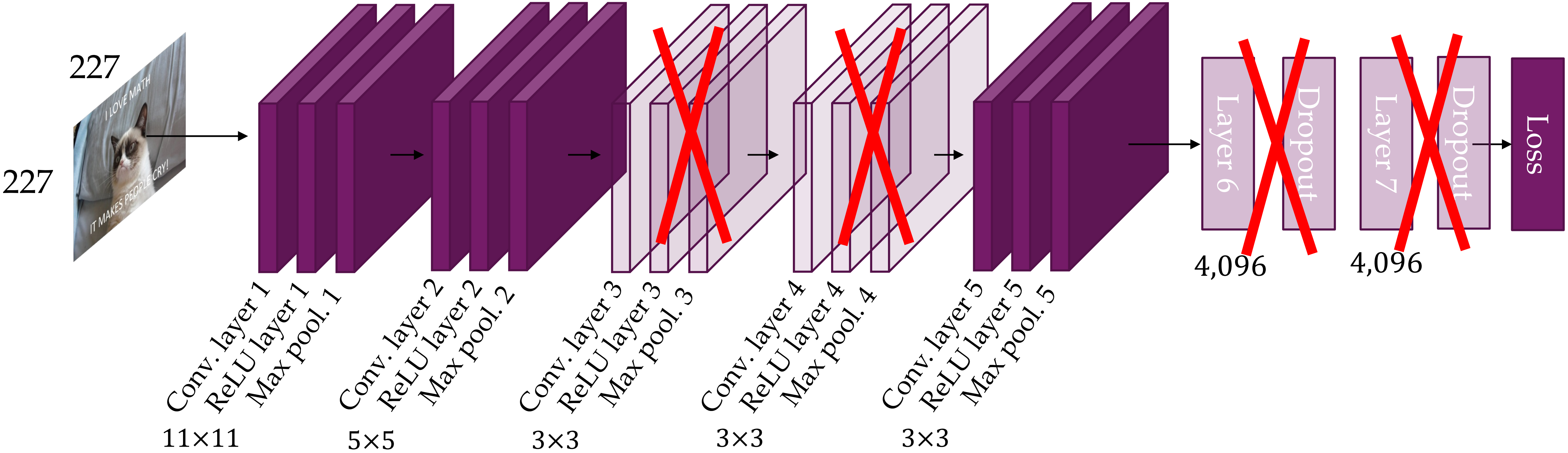
Removing layer 3, 4

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



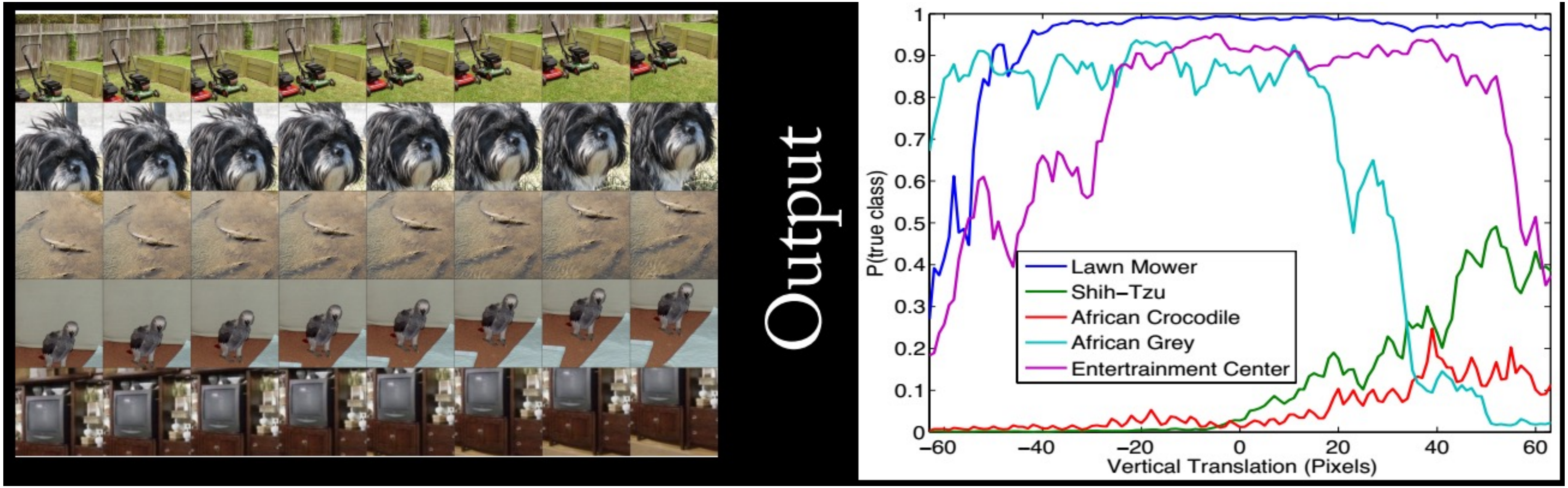
Removing layer 3, 4, 6, 7

33.5% drop in performance. Depth is crucial.



Translation invariance

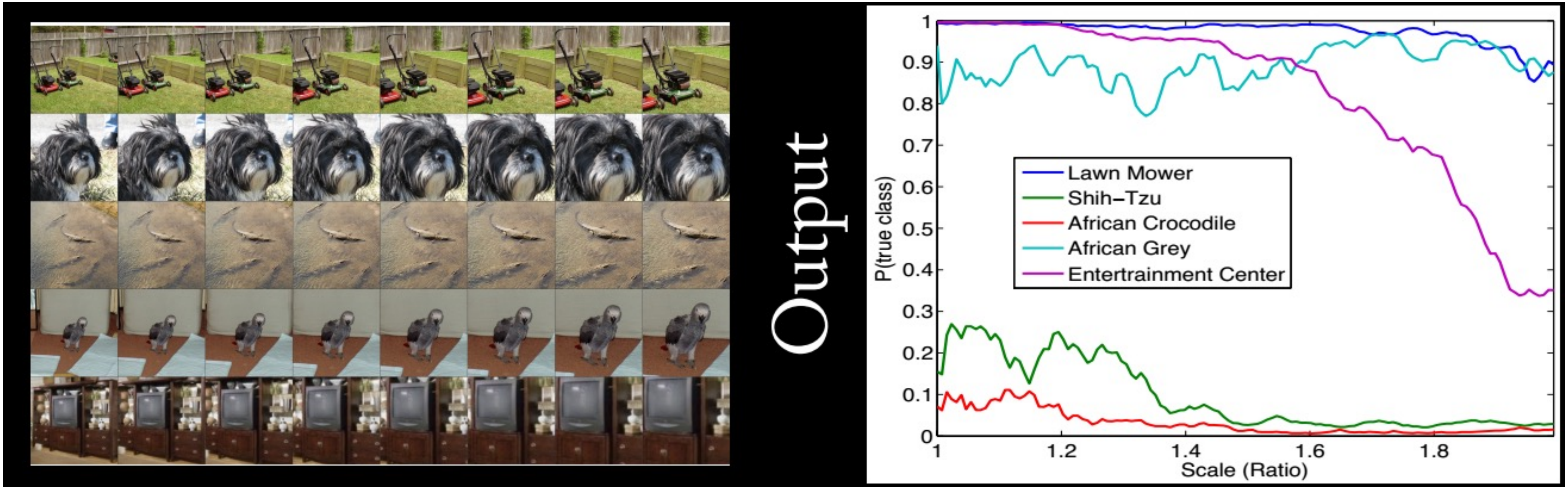
- CNNs are more or less translation invariant



Credit: R. Fergus slides

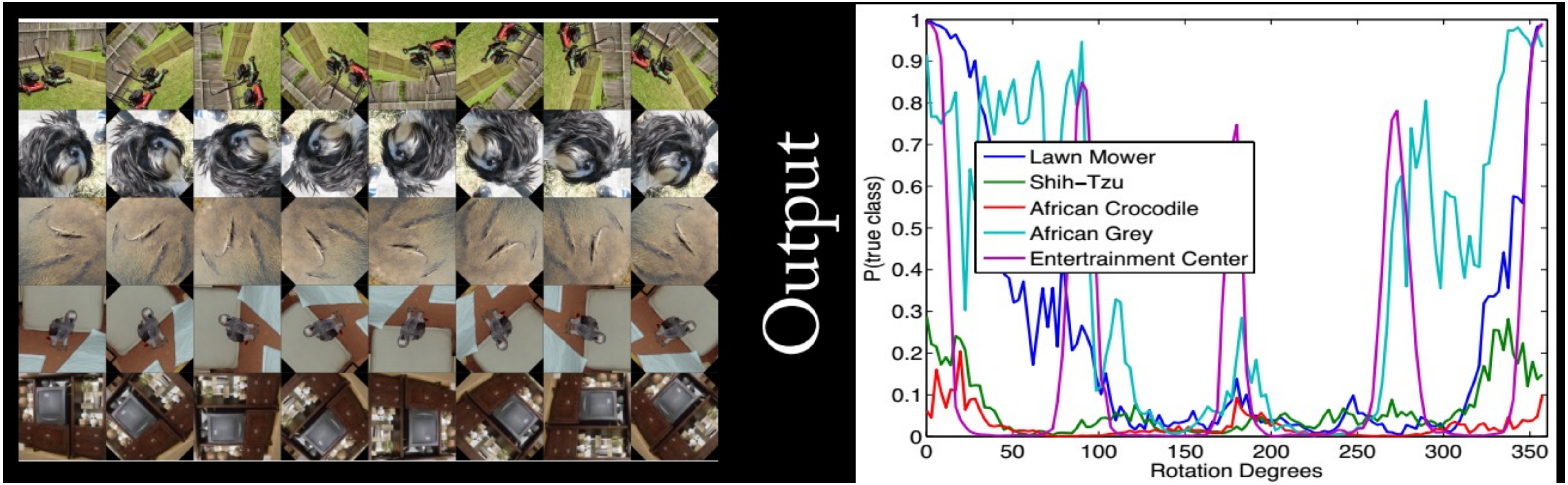
Scale invariance

- 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



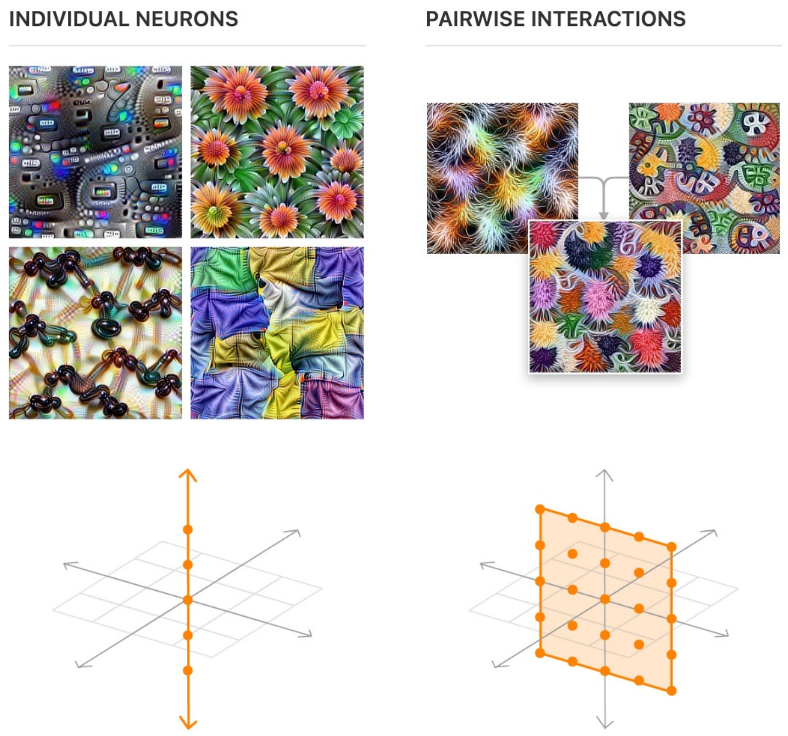
Rotation invariance

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



Further reading

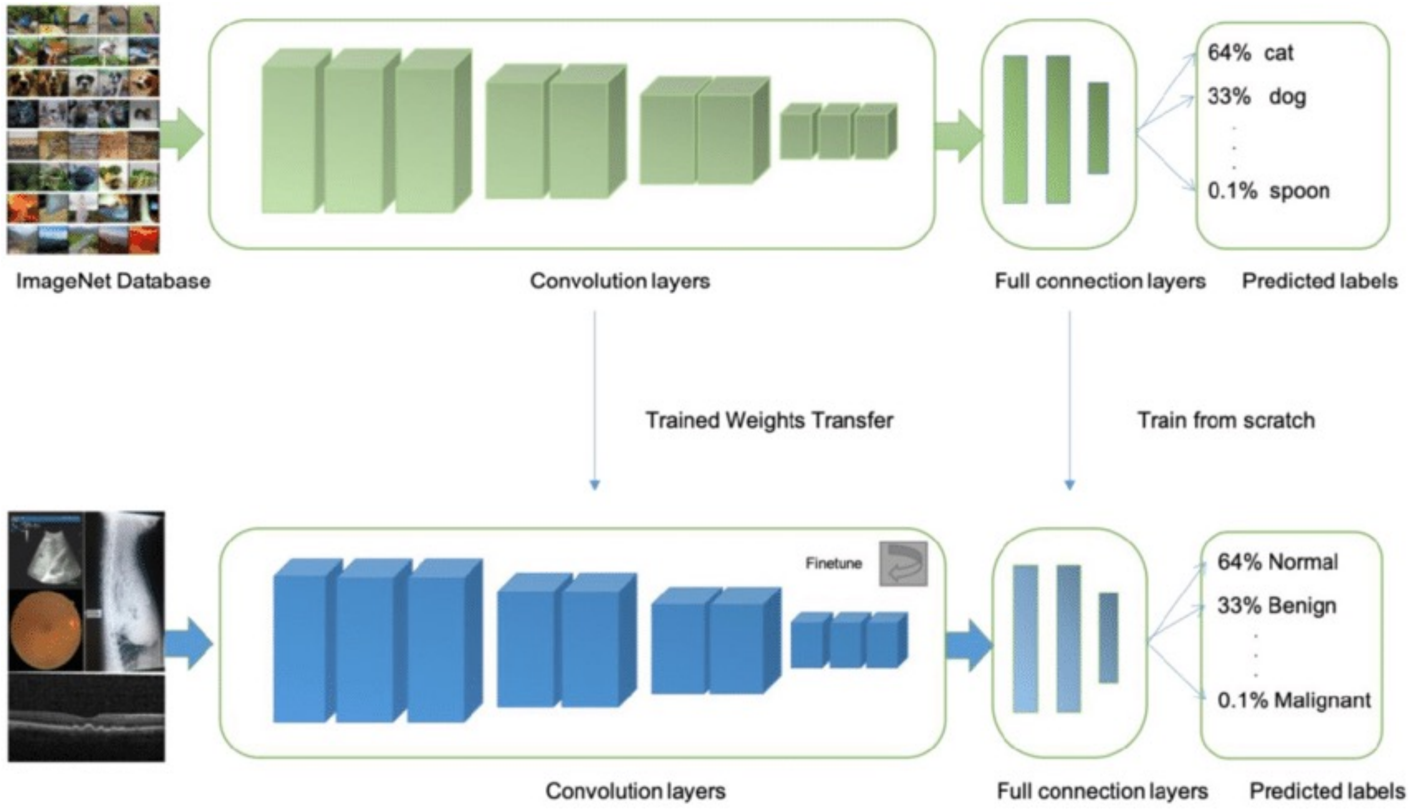
- https://distill.pub/2019/activation-atlas/
- https://distill.pub/2018/differentiable-parameterizations/
- https://distill.pub/2020/circuits/



Visualizing individual neurons make hidden layers somewhat meaningful, but misses interactions between neurons — it only shows us one-dimensional, orthogonal probes of the high-dimensional activation space.

Pairwise interactions reveal interaction effects, but they only show two-dimensional slices of a space that has hundreds of dimensions, and many of the combinations are not realistic.

Transfer learning



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

Imagenet: 1million

- 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

h_S

"My dataset": 1,000

h_T

UPDATE: Transfer learning

- Assume two datasets, T and S
- Dataset S is
  - ~~fully annotated, plenty of images~~
  - HUGE
  - We can build a model h_S (using self-supervised learning)
- 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

Instagram: 5Billion+…

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

Solution I: Fine-tune h_T using h_S as initialization

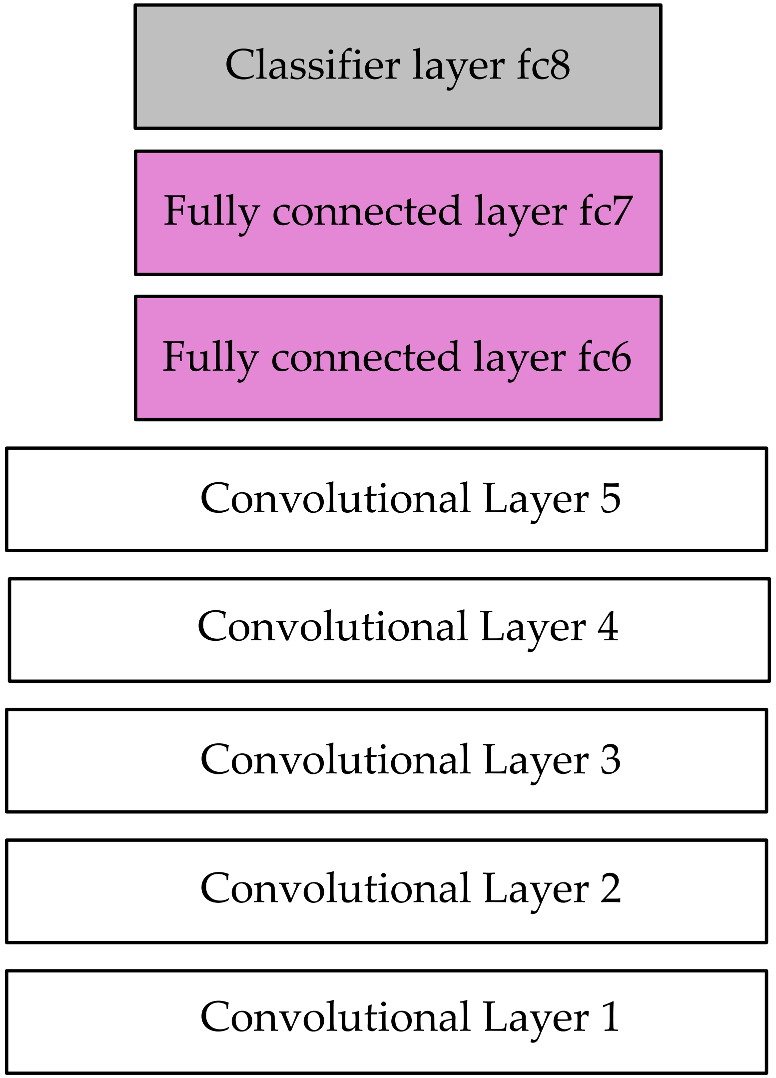
- Assume parameters trained on S are already a good initial solution
- Use them as the initial parameters for our new CNN for the target dataset

$$w_l^S = w_{l,init}^T \text{ for layers } l = 1, 2, \ldots$$

- Better when your source S is large and target T is small (relatively)
  - E.g. reuse parameters from Imagenet models for smaller datasets
- What layers to initialize and how?

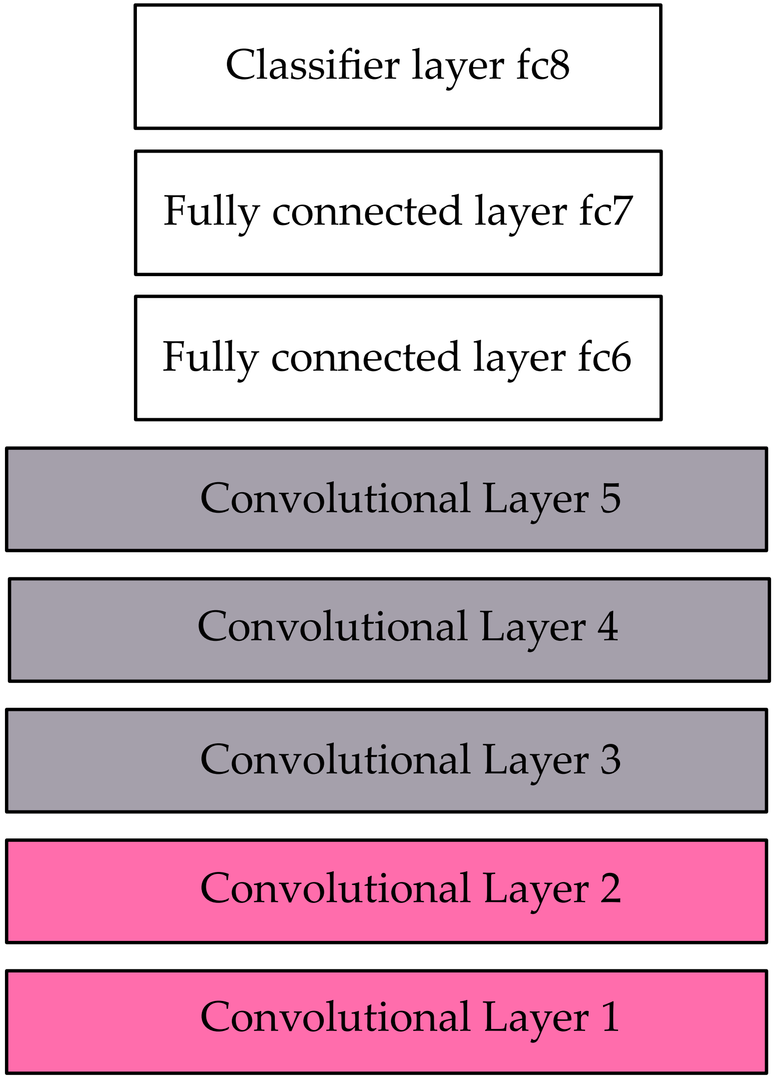
Initializing h_T with h_S

- Classifier layer to loss
  - The loss layer essentially is the "classifier"
  - Same labels → keep the weights from h_S
  - Different labels → delete the layer and start over
  - When too few data, fine-tune only this layer
- Fully connected layers
  - Very important for fine-tuning
  - Maybe delete the last layer before the classification layer if datasets are very different
  - Combine spatial features, more semantics
  - If you have more data, fine-tune these layers first



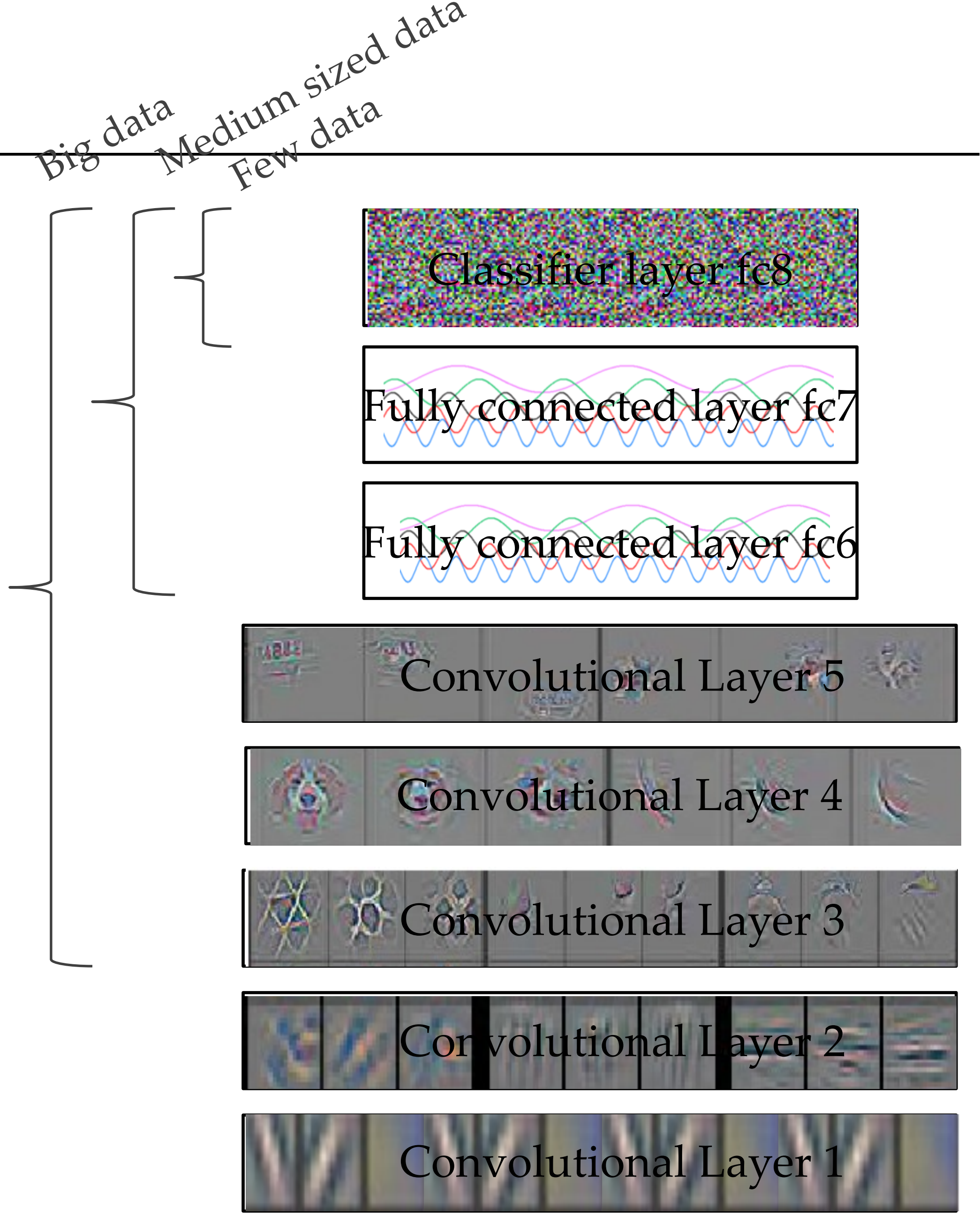
Initializing h_T with h_S

- Upper convolutional layers (conv4, conv5)
  - Mid-level spatial features (face, wheel detectors …)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tune if dataset is big enough
- Lower convolutional layers (conv1, conv2)
  - They capture low level information
  - This information does not change usually
  - Probably, no need to fine-tune but no harm trying



How to fine-tune?

- For layers initialized from h_S use a mild learning rate
  - Your network is already close to a near optimum
  - If too aggressive, learning might diverge
  - A learning rate of 0.001 is a good starting choice (assuming 0.01 was the original learning rate)
- For completely new layers (e.g. loss) use aggressive learning rate
  - If too small, the training will converge very slowly
  - The rest of the network is near a solution, this layer is very far from one
  - A learning rate of 0.01 is a good starting choice
- If datasets are very similar, fine-tune only fully connected layers
- If datasets are different and you have enough data, fine-tune all layers
- You can also sometimes just learn the norm layers (works surprisingly well)



Solution II: Use h_s as a feature extractor for h_T

- Similar to a case of solution I where you train only the loss layer
  - But can be used with 'external classifiers'
  - Essentially use the network as a pretrained feature extractor
- Use when the target dataset T is very small
  - Any fine-tuning of layer might cause overfitting
  - Or when we don't have the resources to train a deep net
  - Or when we don't care for the best possible accuracy

Transfer learning benchmarks & techniques

- Key idea: can the model *generalize* to different datasets and distributions
- See Self-Supervised Learning lecture

Summary

- Convolutions
- Convolutional Neural Networks
- Alexnet case study
- Transfer learning

Reading material

- Deep Learning Book Chapter 9
- UDL book Chapter 10