DEEP LEARNING FOR VISUAL RECOGNITION

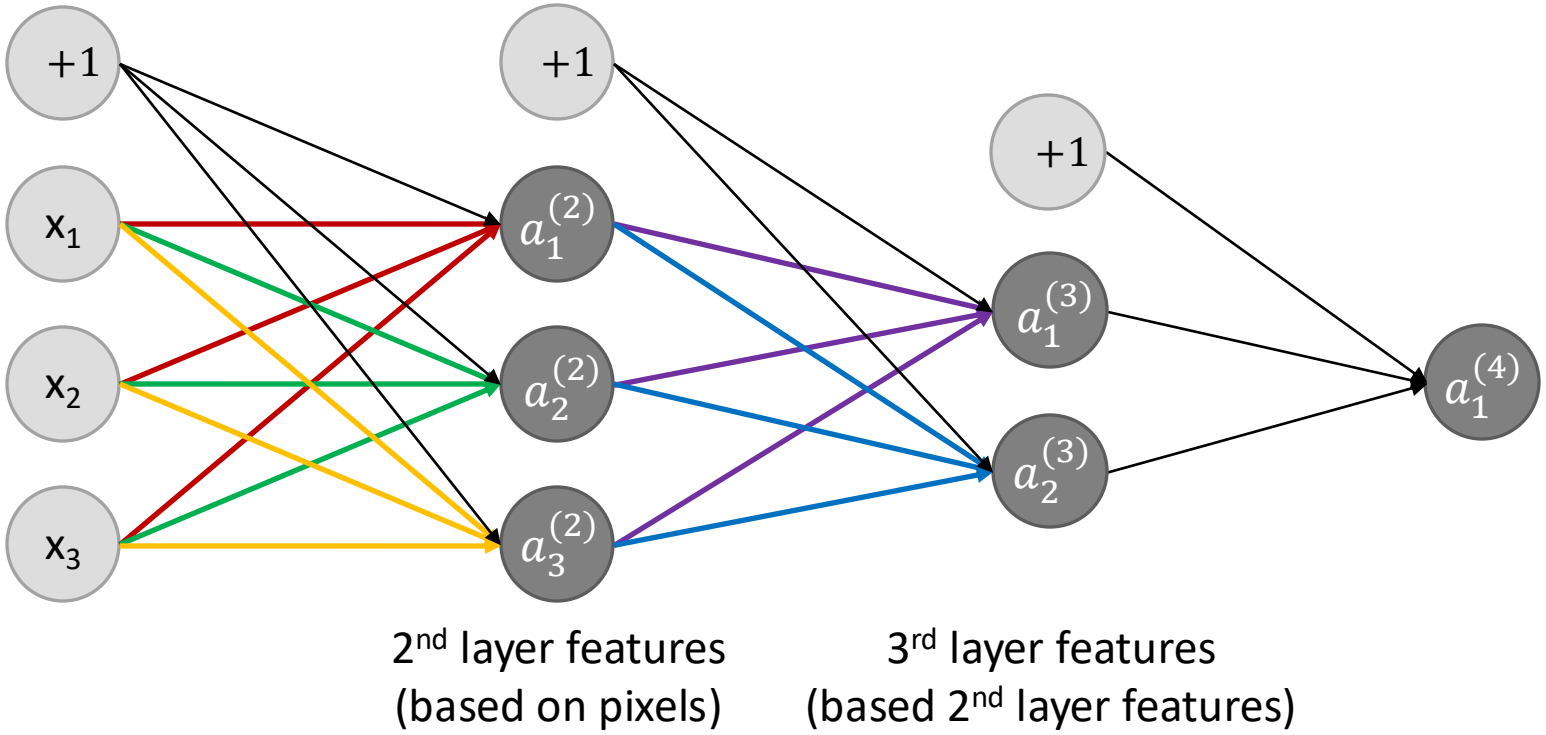
Lecture 4 – Convolutional Neural Networks

Henrik Pedersen, PhD
Part-time lecturer
Department of Computer Science
Aarhus University
hpe@cs.au.dk

Today's agenda

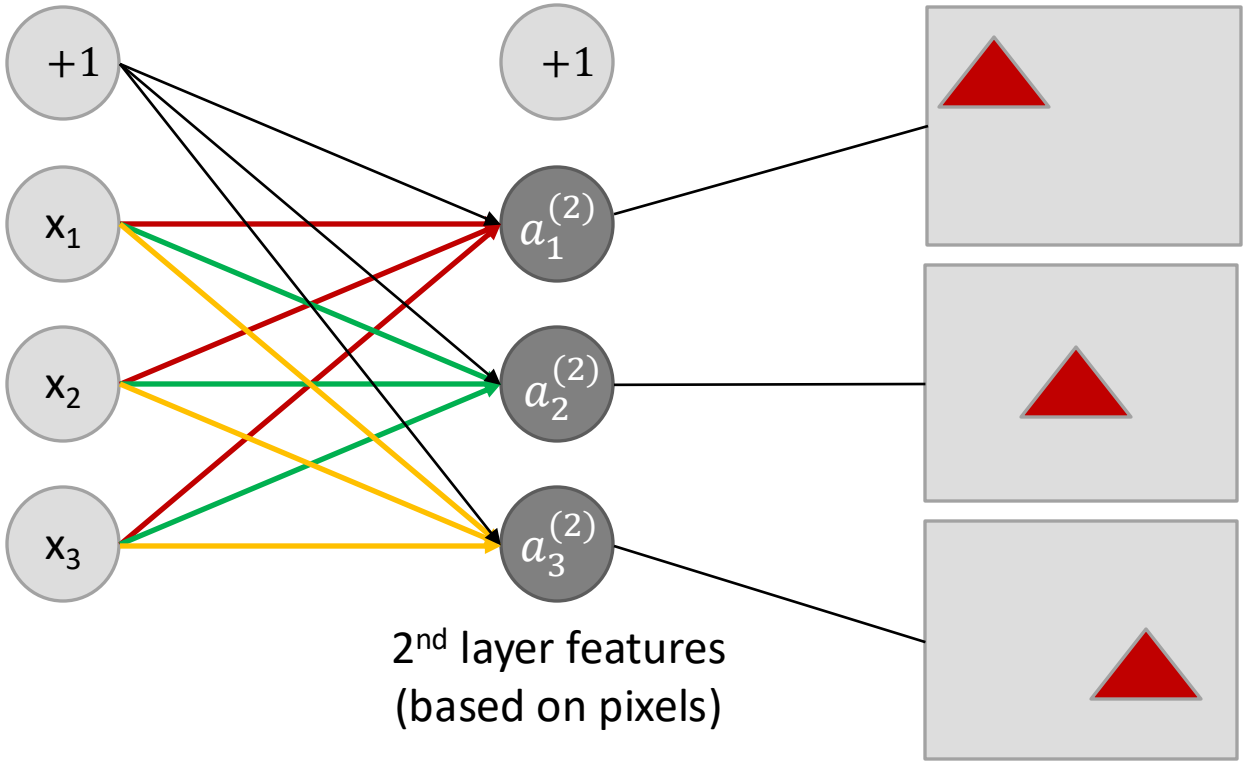
- You will learn about convolutional neural networks (CNNs): a special type of neural network architecture that can take images as input (rather than vectors).
- Topics
  - Feature extraction using convolution
  - Architecture of CNNs
  - Convolution layers
  - Pooling layers
  - Fully connected layers
  - Simple CNN architectures: LeNet-5 and AlexNet
  - Applications

Last time: Neural networks



Disadvantage: Layers are "fully connected", meaning that each neuron receives input from all neurons of the previous layer. Not ideal for detecting local features (like object parts in an image).

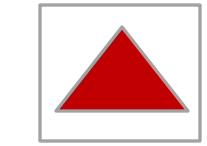
Last time: Neural networks



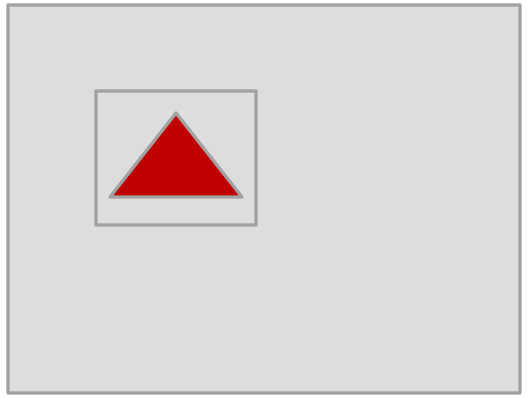
In a fully connected neural network, we would need a separate neuron to detect each of the red triangles above. Seems like a waste of network parameters…

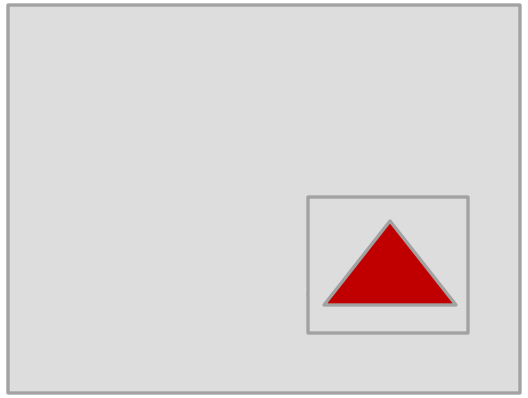
Feature detection using convolution

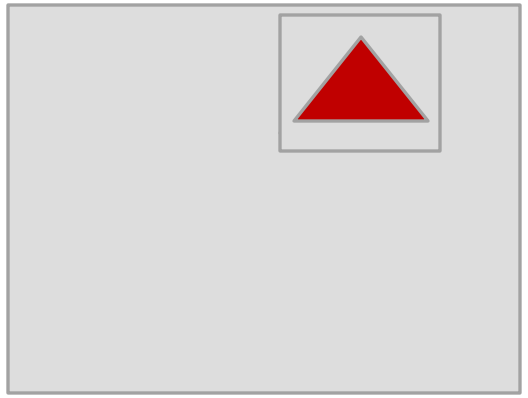
- **Simple idea –** use convolution to identify local features.
- **Advantages**
  - Fewer trainable parameters (sparse interactions)
  - Same parameters reused multiple times (parameter sharing)
  - Translation equivariance (and approximate translation invariance)



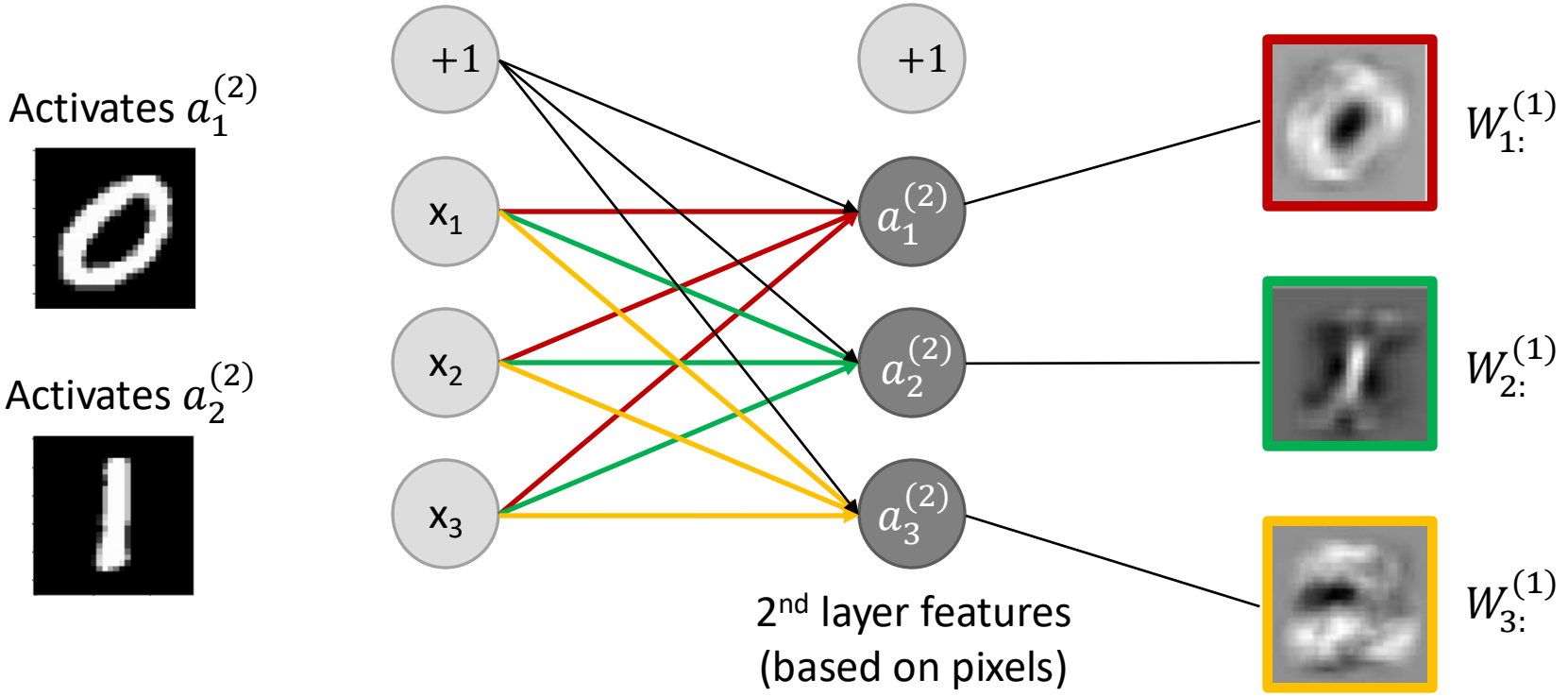
Convolution filter/kernel







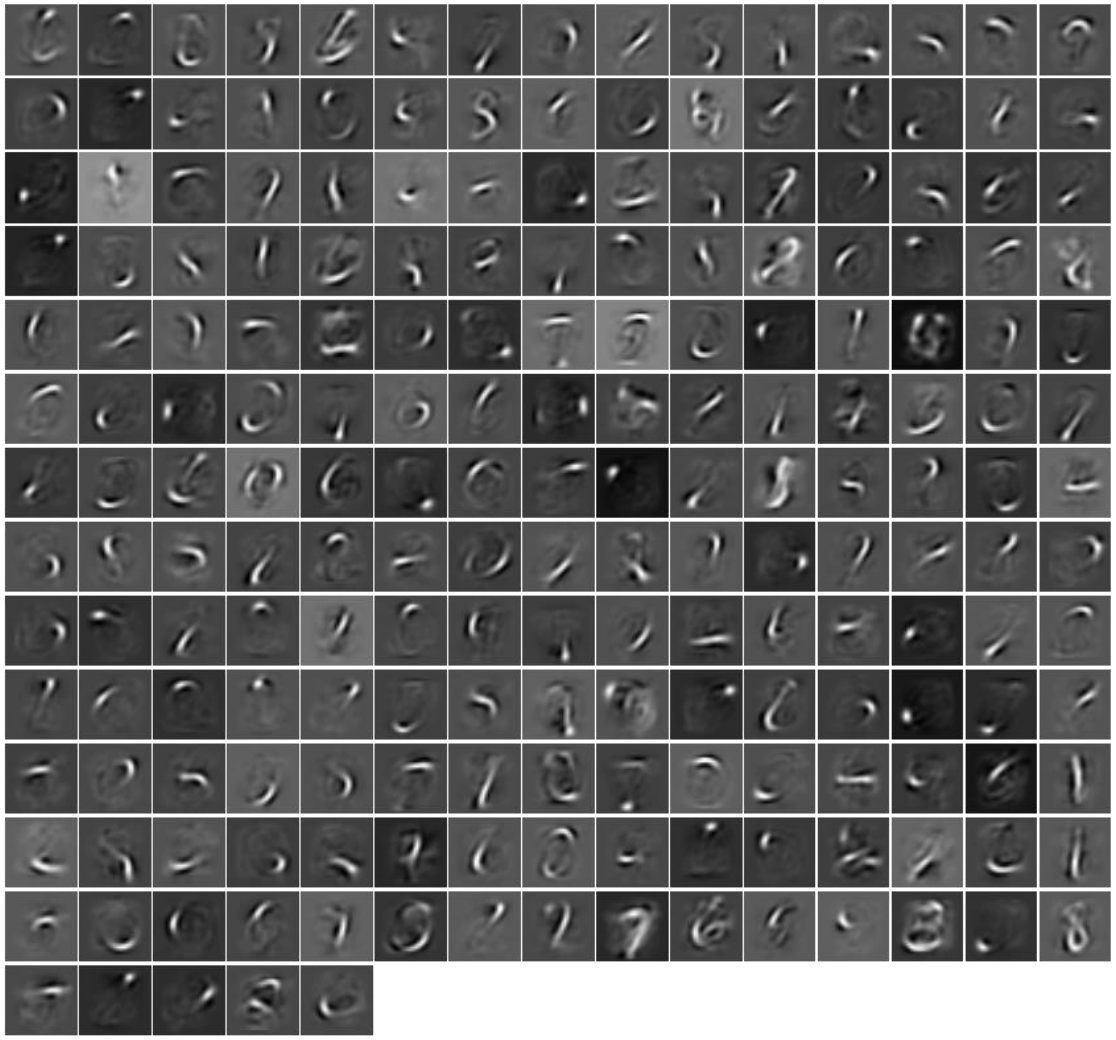
Example: MNIST



Recall that neurons are activated when the input matches a certain template (given by the weights W)

Example: MNIST

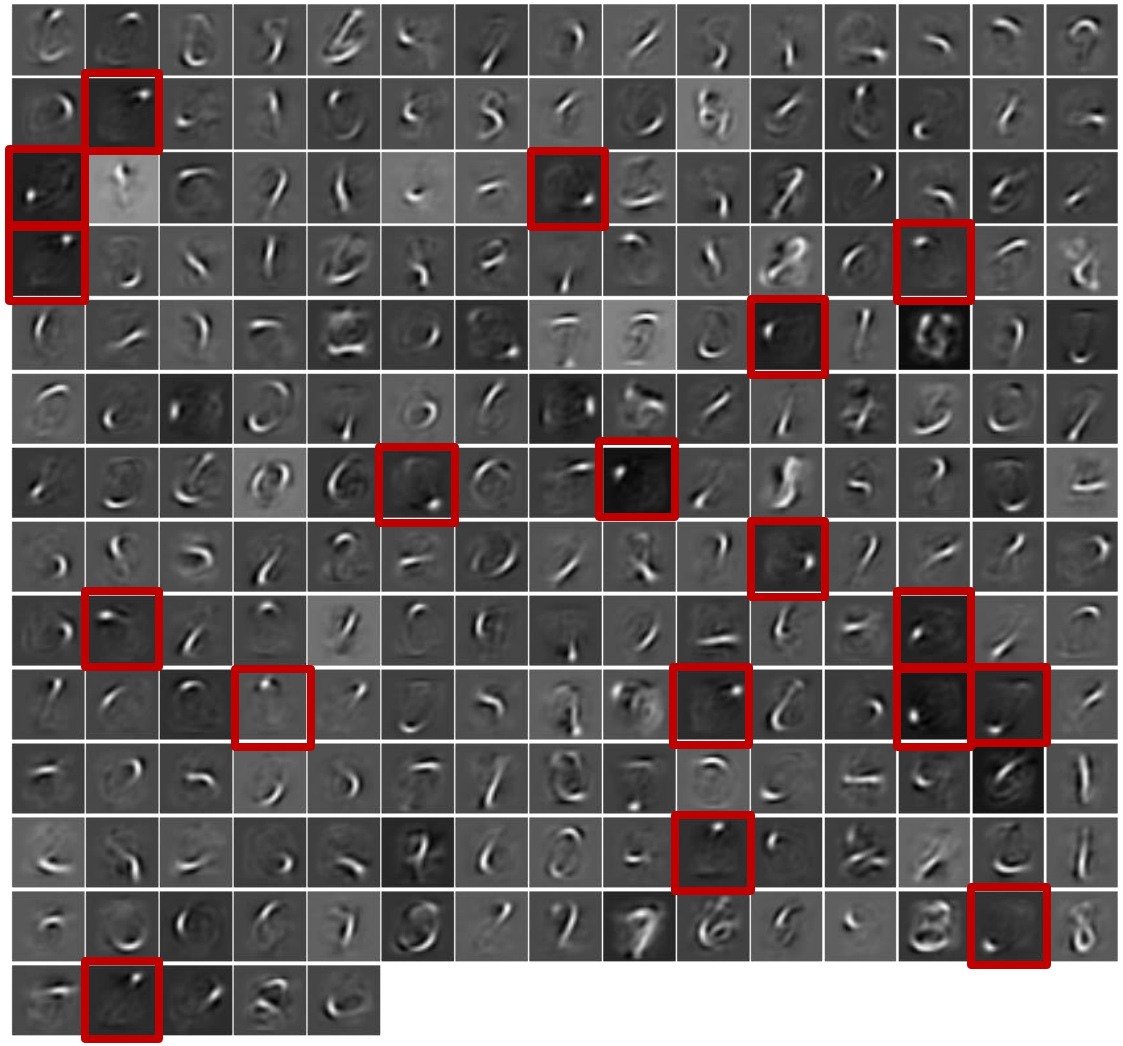
- The figure on the right shows the weights (or templates or features) learned in the first layer of a so-called sparse autoencoder trained on MNIST.
- The features are sparse, meaning that they represent **local features** in the input image.
- **Problem:** The majority of the weights are set to zero, i.e., they are not used at all.



https://stackoverflow.com/questions/42404040/sparse-autoencoder-cost-function-in-tensorflow

Example: MNIST

- All of the marked neurons detect more or less the same feature (a blob), but at different locations.
- All weights outside the blob are set to zero, i.e., they are just waste.
- It would be much more efficient to detect the blobs with a suitable filter, and then apply that filter at all locations in the image.
- That's exactly what convolution does.
- **Bottomline:** Fully connected networks need an enormous number of weights to represent the same local features as a convolutional neural network.



https://stackoverflow.com/questions/42404040/sparse-autoencoder-cost-function-in-tensorflow

Feature detection using convolution

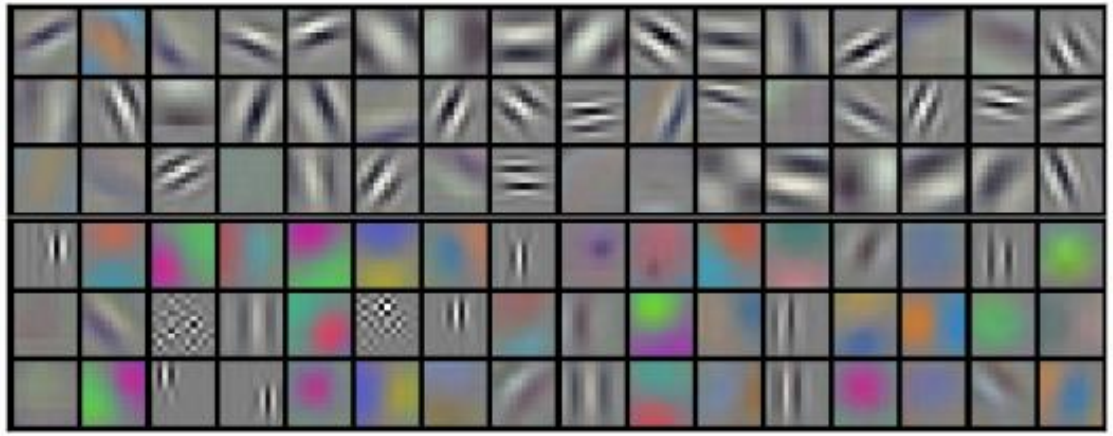
Input image

Convolution Kernel

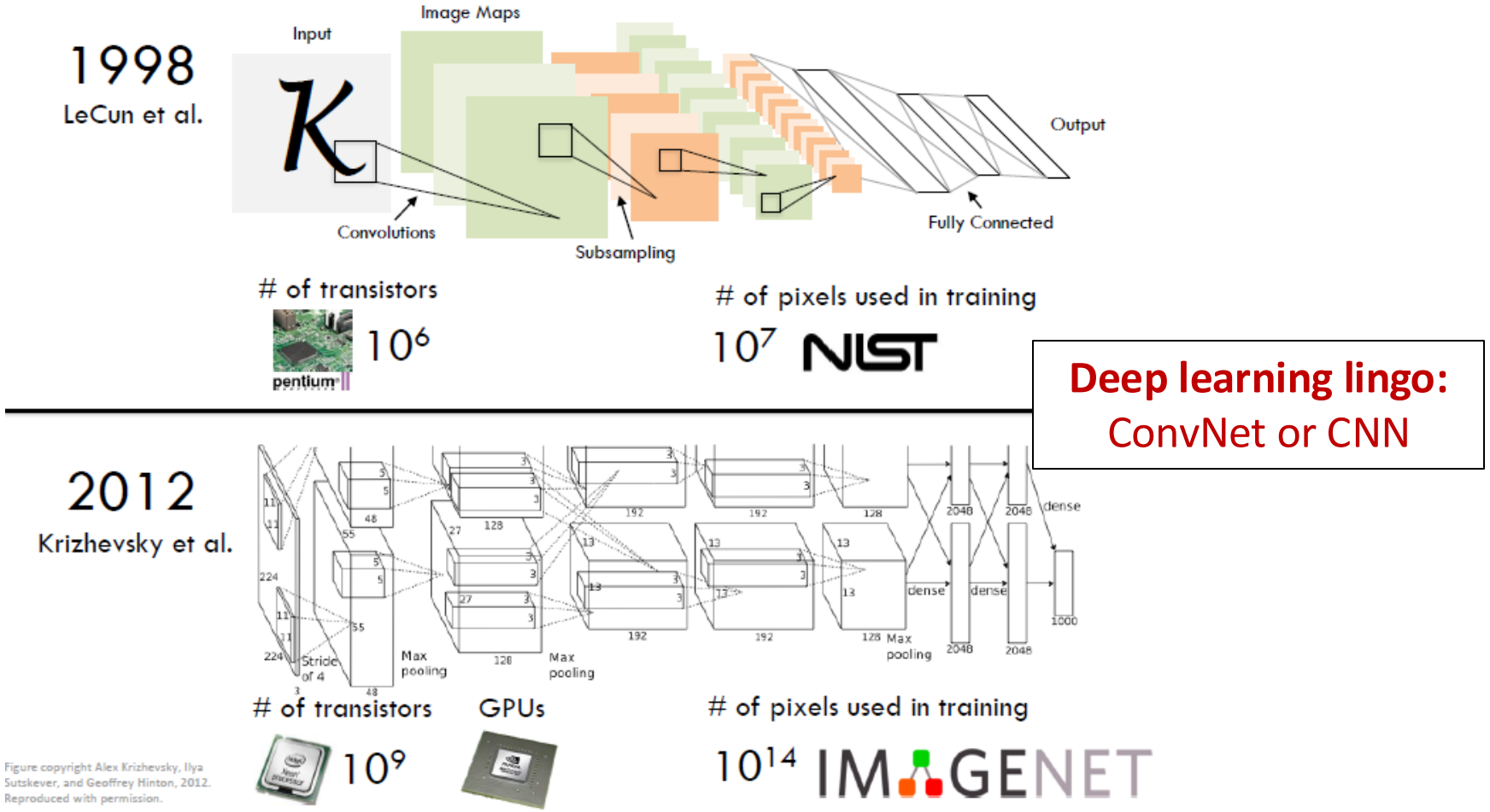
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Convolution kernels learned by a neural network



Today: Convolutional Neural Networks



Deep learning lingo:
ConvNet or CNN

http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

Convolution

FEATURE EXTRACTION

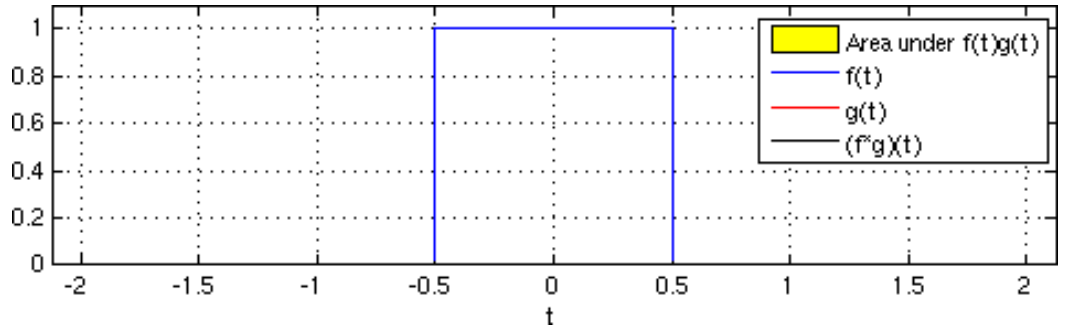
What is convolution?

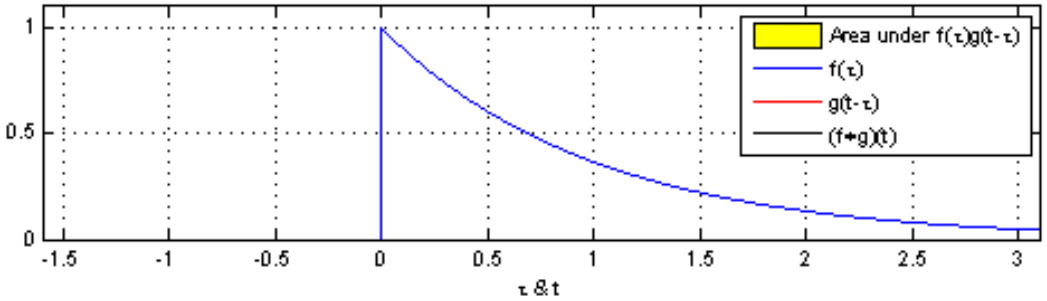
- Formal definition

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t - \tau) g(\tau)\, d\tau$$

- Seen as an average of f at the moment t weighted by $g(-\tau)$.
- Discrete convolution is formally defined as

$$(f * g)(t) =$$
$$\sum_{-\infty}^{\infty} f[k] g[t - k] = \sum_{-\infty}^{\infty} f[t - k] g[t]$$



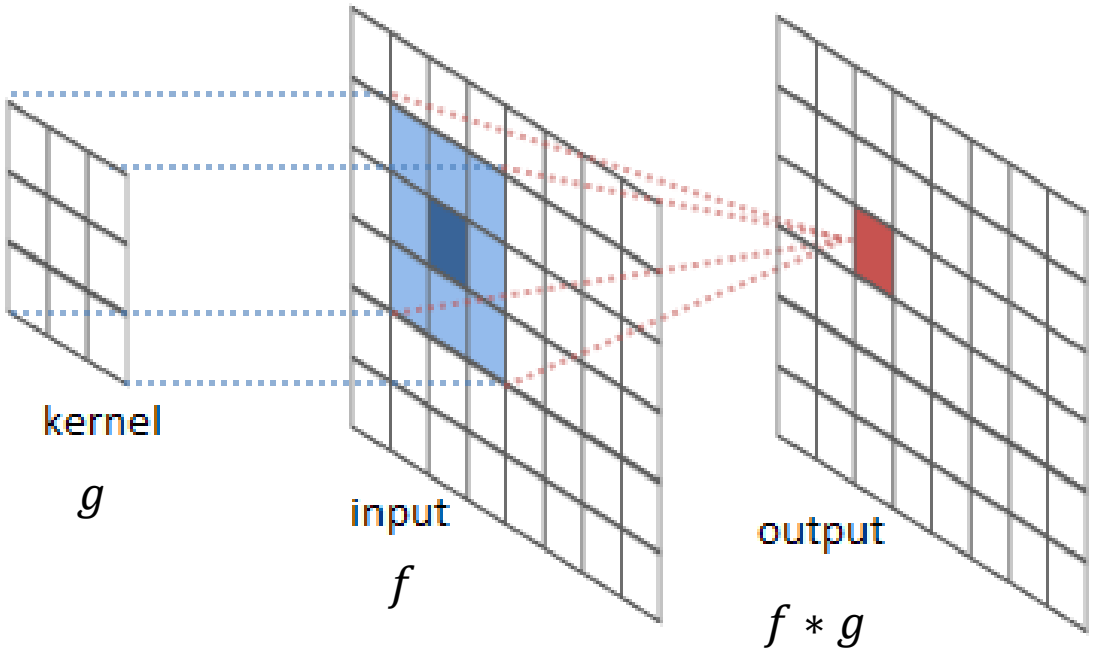


Click on images to view GIF animations

Convolution in images

- We refer to $g(x)$ as the 'filter' or 'kernel'.
- Each pixel in the output image is a weighted sum of pixels in the input image.
- The weighting is determined by the filter or kernel.
- When vectorized, this is just an **inner product***

$$(f * g)[x, y] = \sum_{n=-K}^{K} \sum_{m=-K}^{K} f[x - m, y - n] g[m, n]$$



* formally, it is the inner product between the filter (g) rotated by 180 degrees and the input (f).

Many variants and applications

- Very nice overview and introduction here: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

Convolution for a single channel

Input

Output

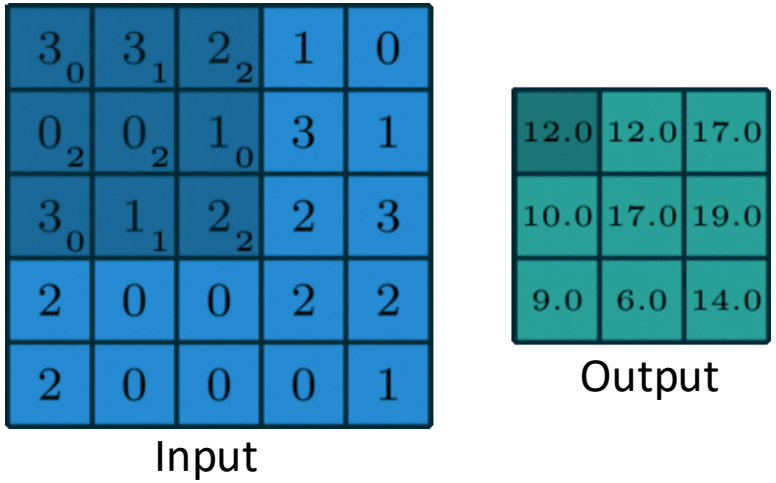
0	1	2
2	2	0
0	1	2

3x3 filter

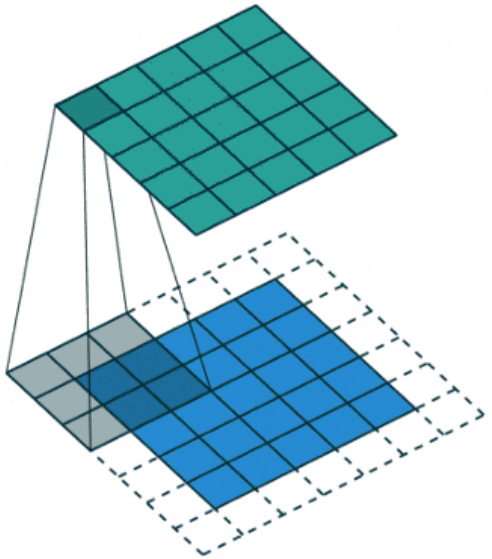
Many variants and applications

- Very nice overview and introduction here: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

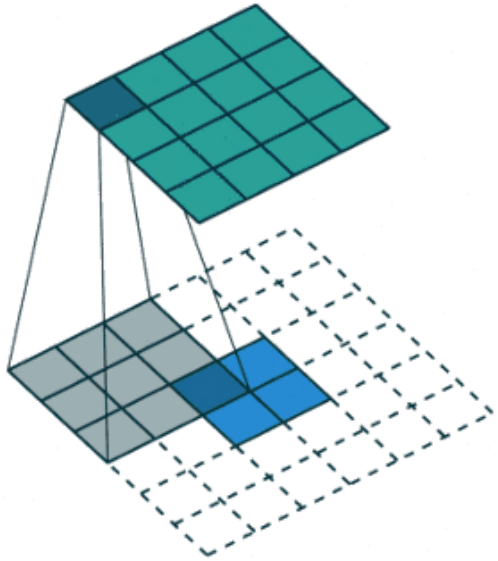
Convolution for a single channel



2D convolution using a kernel size of 3, stride of 1 and padding of 1

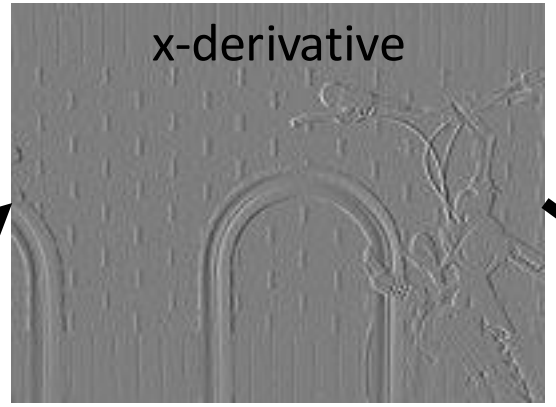


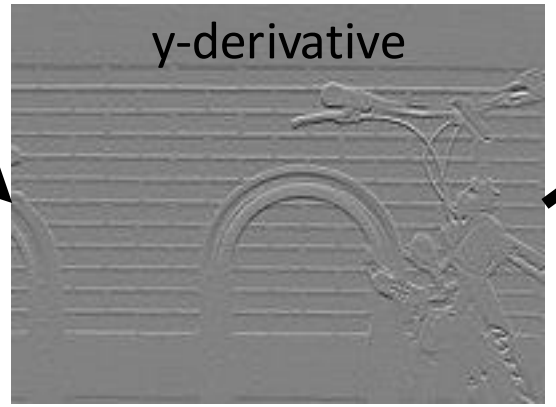
Up-sampling a 2 x 2 input to a 4 x 4 output

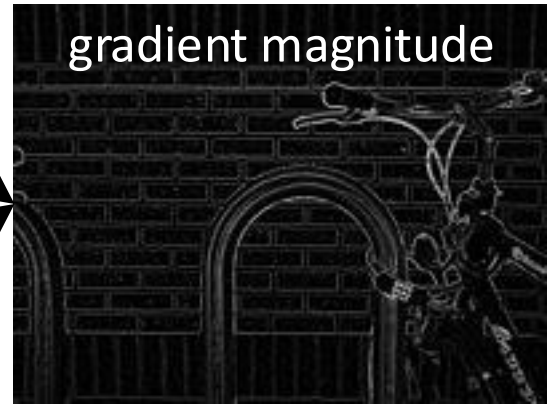


Example: Sobel filter

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$







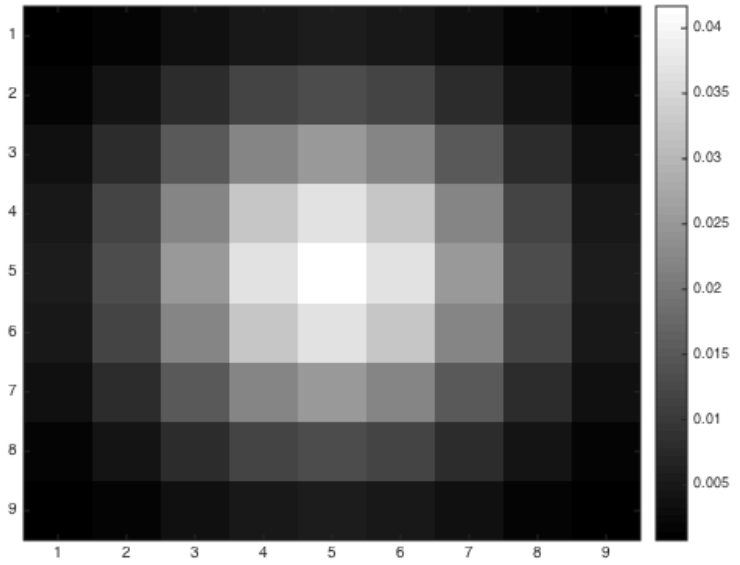
$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Sobel filters act as edge detectors.
(formally they estimate the derivative with some built-in noise reduction – we used them in Lab 1)

https://en.wikipedia.org/wiki/Sobel_operator

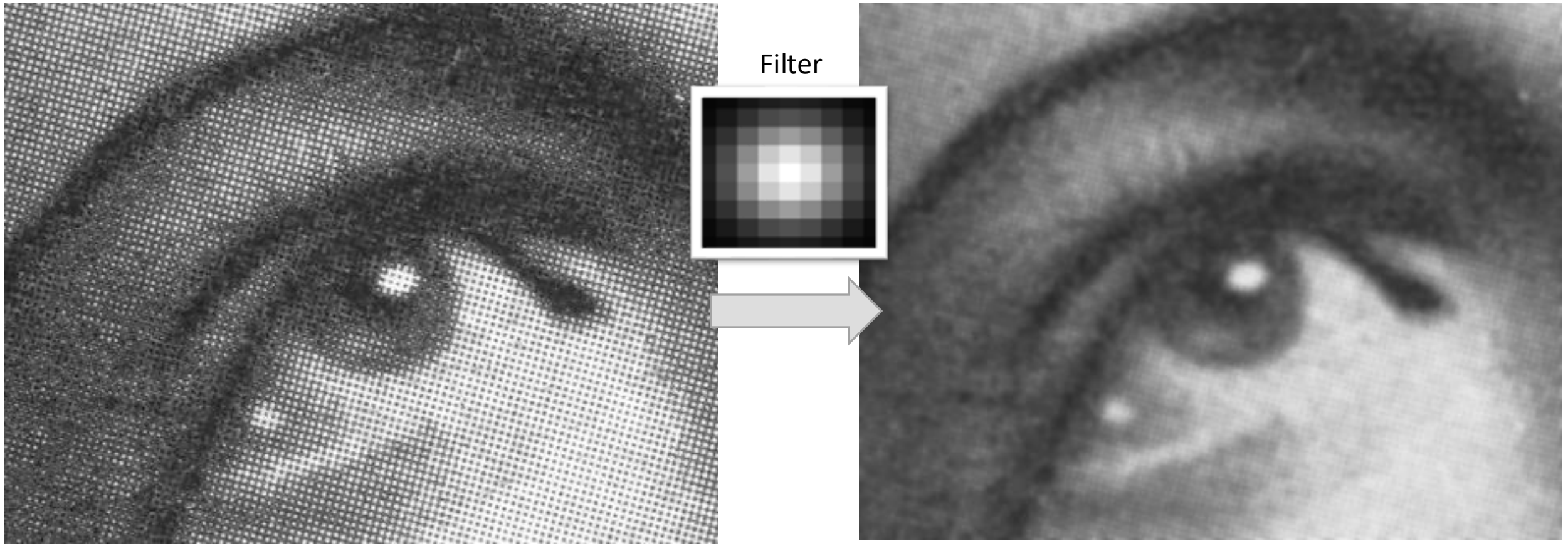
Example: Gaussian filter

- A Gaussian filter is a matrix whose value is distributed according to a Gaussian probability distribution.
- The closer to the center, the greater the value.
- Usually, we normalize the sum of the values to one.
- Want to learn more about filtering?
  - See “Basic image processing.pdf”
  - Brightspace → Course schedule → Week 1



A 9-by-9 Gaussian kernel
The center has the greatest value

Example: Gaussian filter

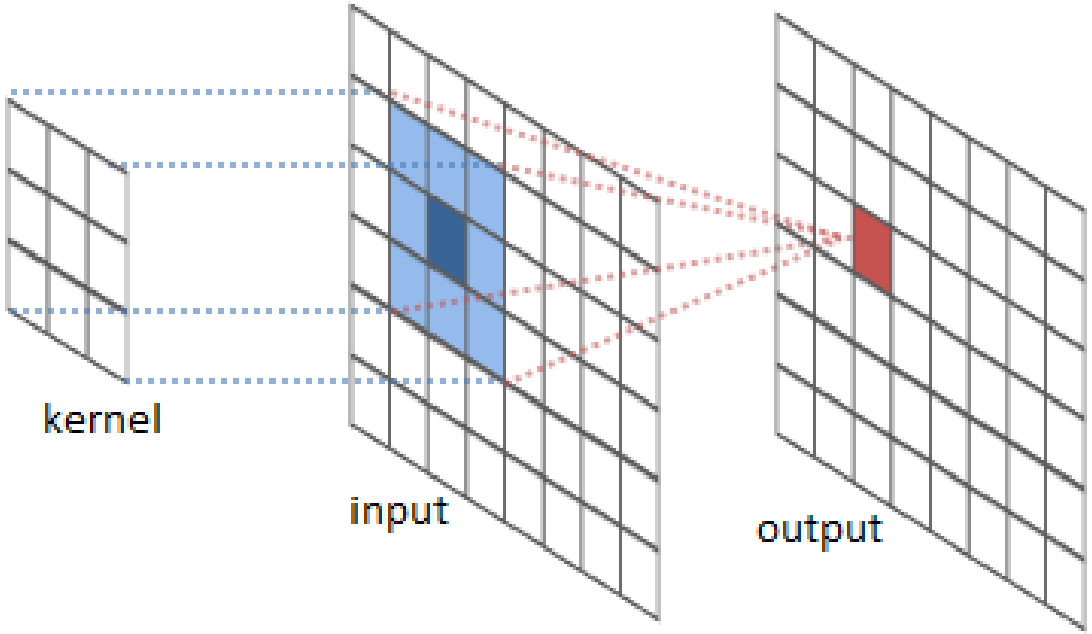


Gaussian filter acts as a smoothness filter.

Intensity at position (x,y) in the output image = weighted average around position (x,y) in the input image

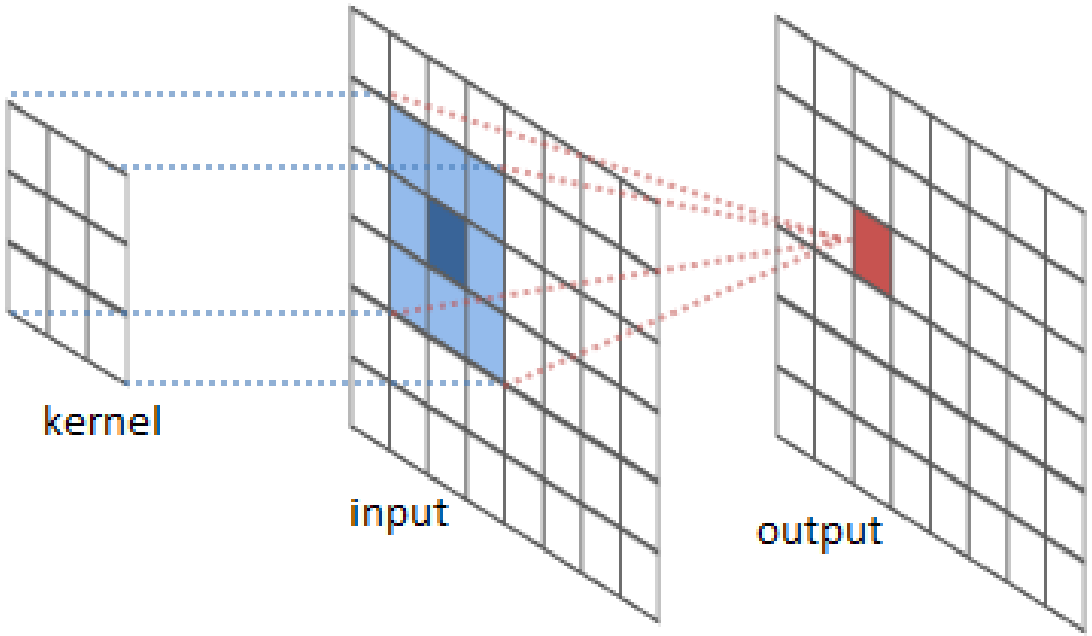
Convolution in a neural network

- The **output neuron** is connected only to those pixels in **a certain region**.
- All weights for neurons in the same layer are the same.
- The weights can hence be viewed as values of a convolution kernel.
- We want CNNs to learn kernels that extract “interesting” features of the image.



Motivation for convolution in NNs

- Convolution leverages three important ideas that can help improve neural nets (NNs):
  - **Sparse interactions**
  - **Parameter sharing**
  - **Translation equivariance**



Motivation: Sparse interactions

Convolutional

Fully connected

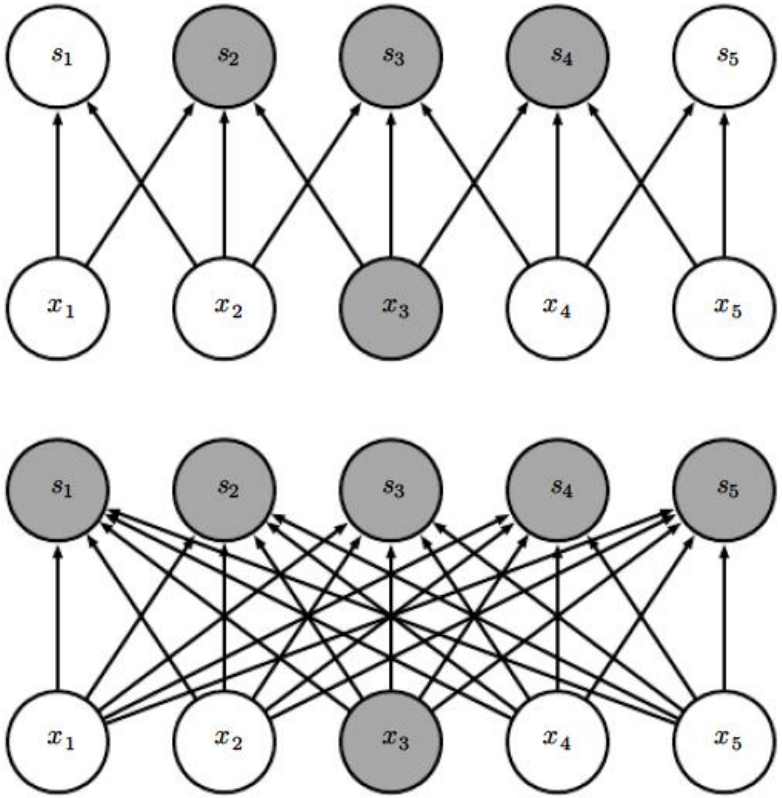


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3, and also highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3.

Motivation: Sparse interactions

- In traditional (fully connected) NNs **every output unit interacts with every input unit**.
- CNNs, however, typically have **sparse interactions**.
- When processing an image with a CNN, the input image might have millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only a few pixels.
- This means that we need to store fewer parameters, which
  - reduces the memory requirements of the model
  - requires fewer operations to compute the output.
- **Example:**
  - Suppose we have n^2 pixels in the first layer, and $(\mathrm{n} - k)^2$ cells in the second layer.
  - CNNs use m different k-by-k kernels between these two layers, where m is often smaller than n, and k is several orders of magnitude lower than n (typically between 3 and 11).
  - Hence, there are $n^2(n-k)^2$ weights in fully connected NN, but only $m \cdot k^2$ weights in CNN.

Motivation: Parameter sharing

Convolutional

Fully connected

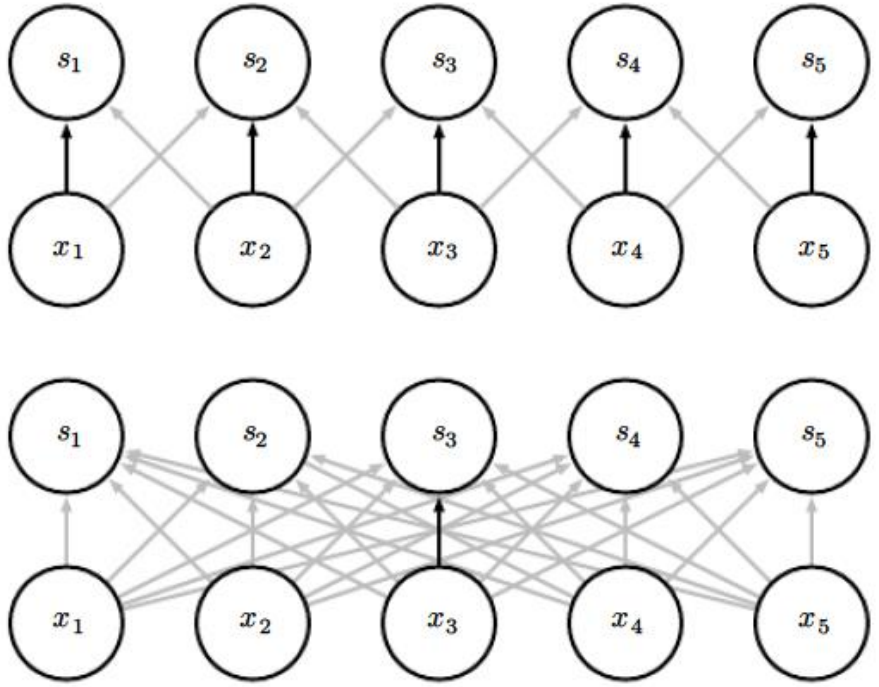


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. *(Top)*The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Bottom)*The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

Motivation: Parameter sharing

- **Parameter sharing (**or **tied weights**) refers to using the same parameter for more than one function in a model.
- In a traditional (fully connected) NN **each element of the weight matrix is used exactly once** when computing the output of a layer.
- In CNNs the value of **the weight applied to one input is tied to the value of a weight applied elsewhere**: each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary).
- The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn a single set that works for all locations.
- Parameter sharing **improves the statistical efficiency** of model training: during backpropagation multiple pixels/neurons contribute to each weight update.

Motivation: Translation equivariance

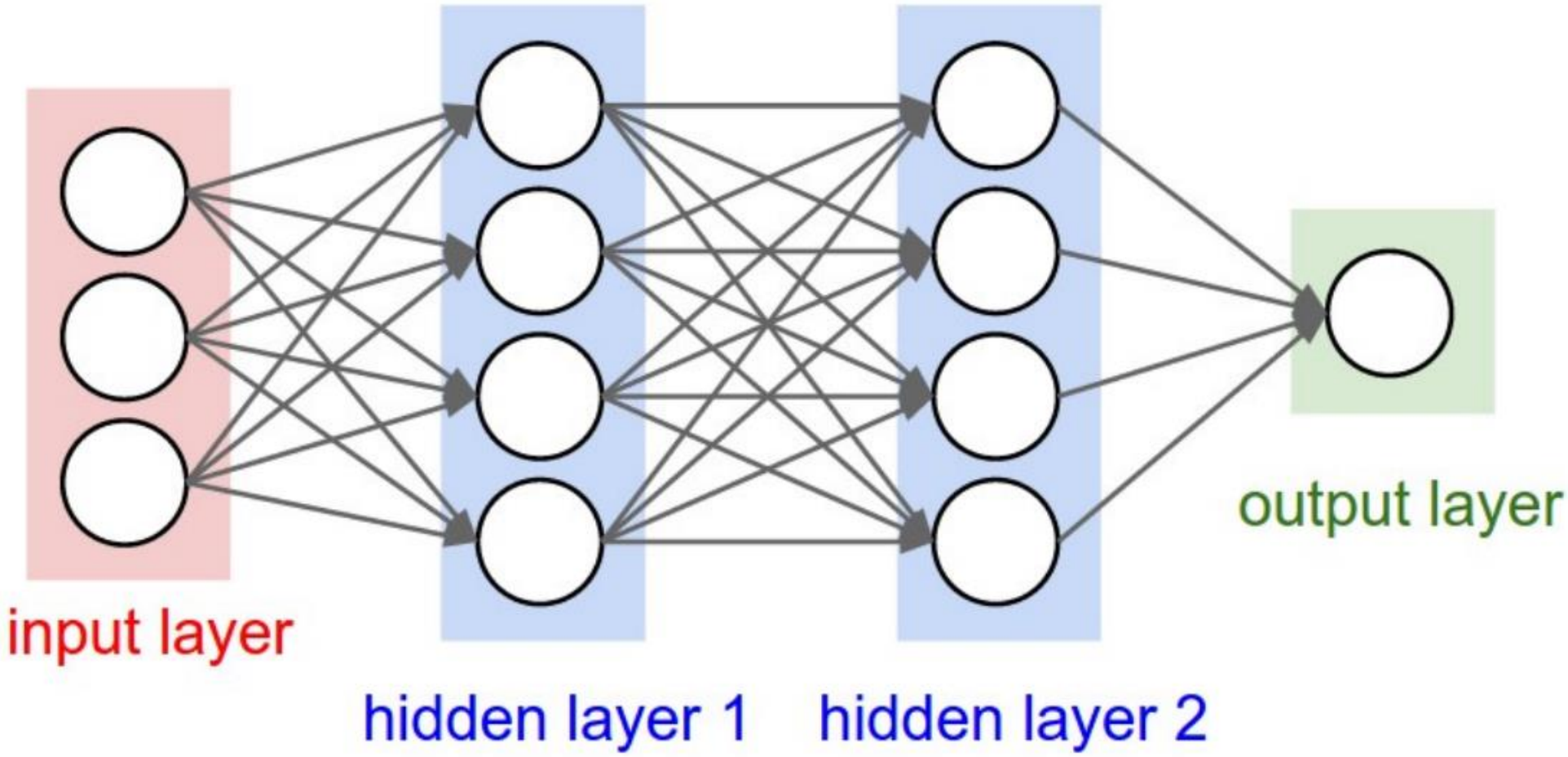
- **Translation equivariance** means: if you translate (shift) the input, the feature maps shift in the same way.
- Formally, if f is the convolution operation and T is a translation operator, then

$$f(T(x)) = T(f(x))$$

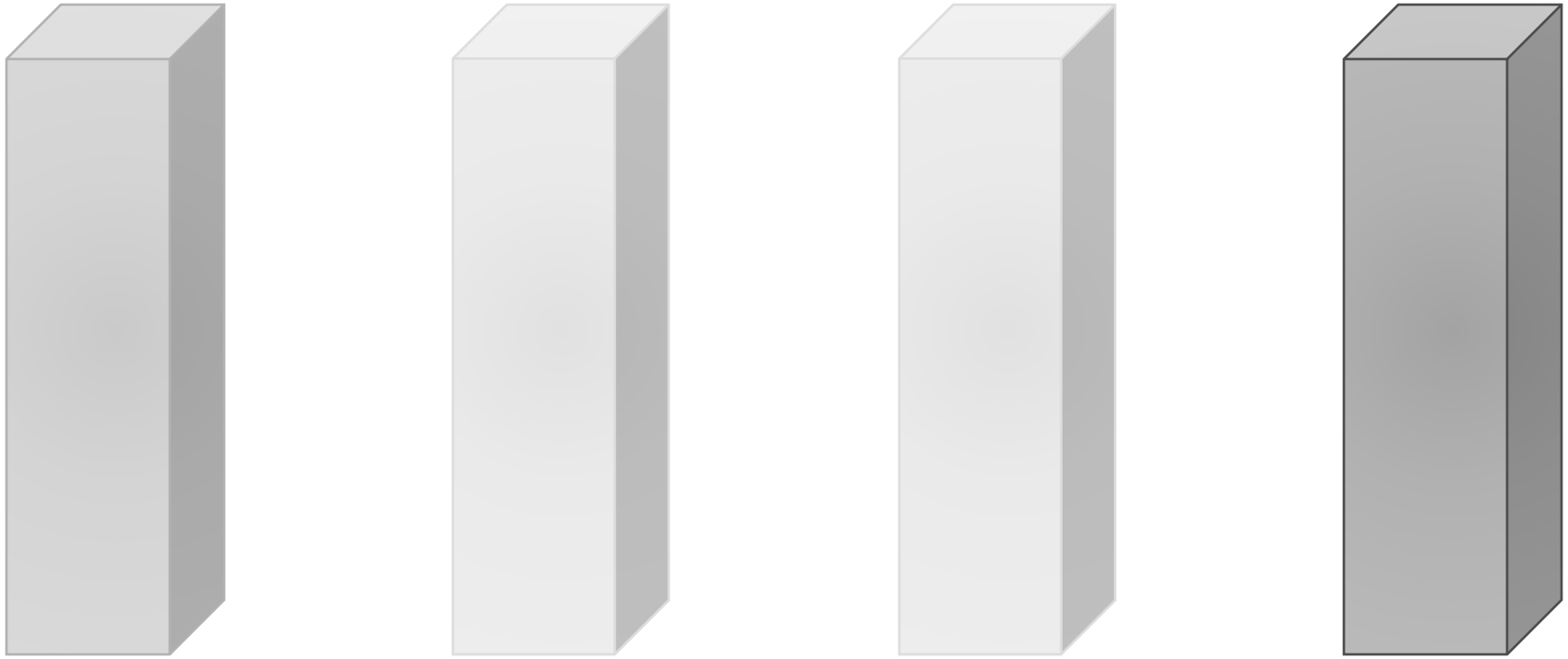
- With images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output.
- CNNs often introduce **pooling** steps (like max pooling or global average pooling), which move the representation closer to **translation invariance**: the same object with slight change of position eventually fires up the neuron (in the last layers of the network) that is supposed to recognize that object.

Architecture of CNNs

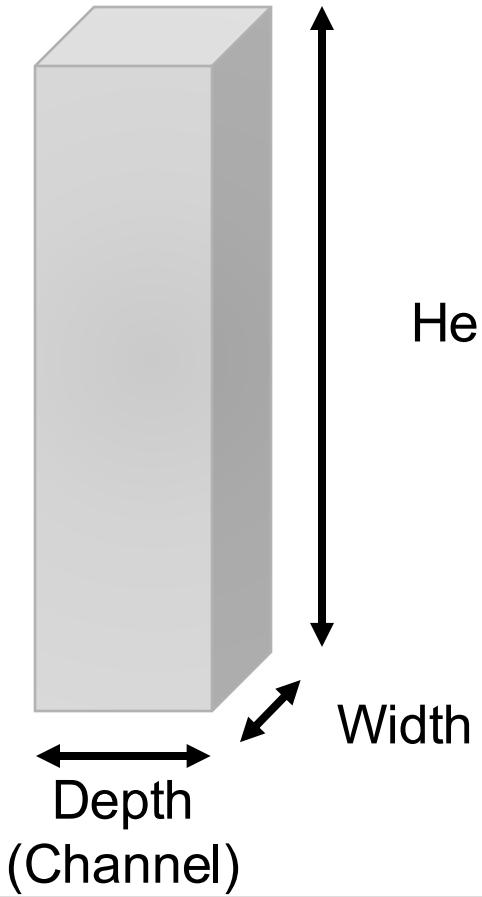
Recall: Fully connected neural networks



Convolutional Neural Networks

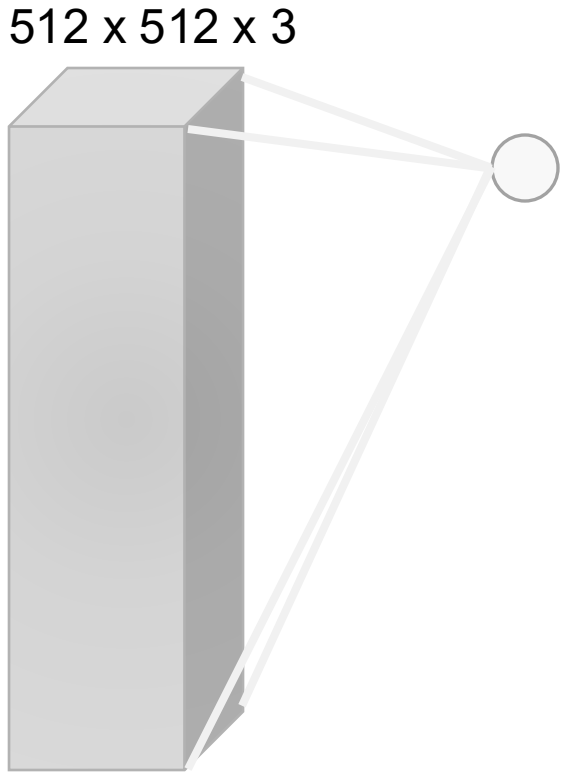


Convolutional Neural Networks



- Fully connected NNs operate on vectors.
- CNNs operate on **volumes**.
- For example, a 512x512 RGB image has height 512, width 512, and depth 3.

Fully connected neural network

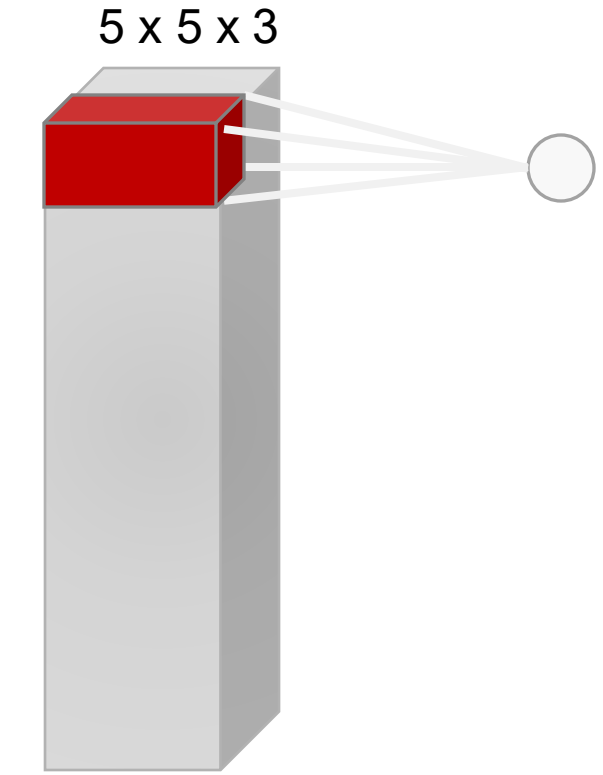


Each output neuron is connected to all neurons of the previous layer – not very efficient.

What should we do?

- The features of images are usually **local**.
- We use convolution to reduce the fully connected network to a **locally-connected** network.
- For example, if we set the window size to 5 …

Convolutional Neural Networks



The output is the dot product between a filter (w) and a small chunk (x) of the input image plus a bias:

$$w^T x + b$$

In this example, the dot product is between two 5x5x3 = 75 dimensional vectors.

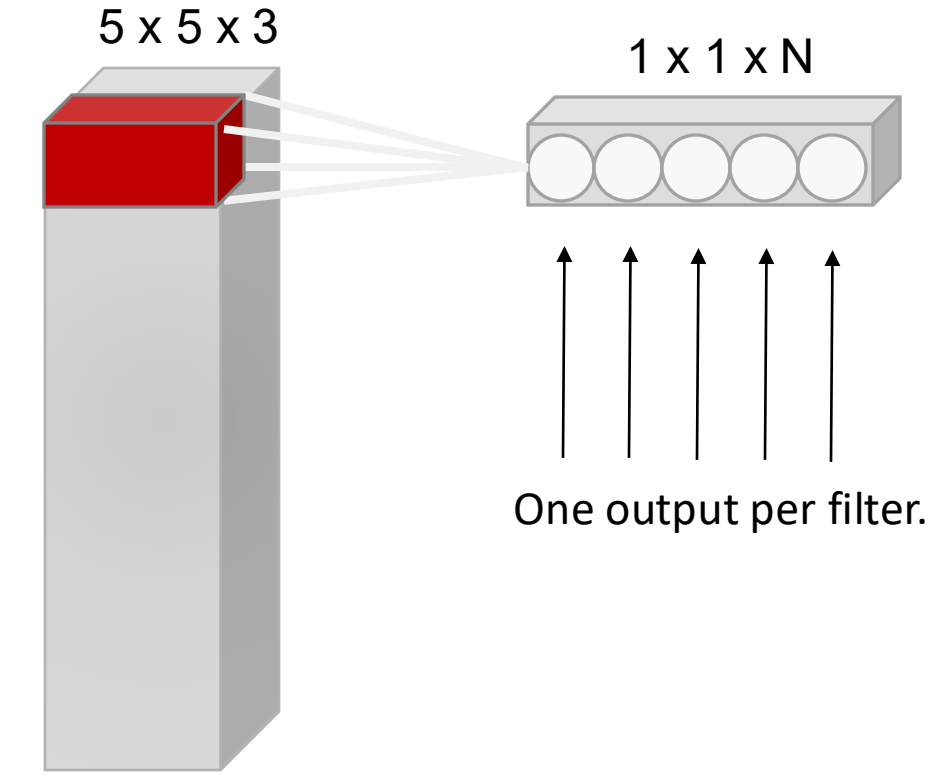
What should we do?

- The features of images are usually **local**.
- We use convolution to reduce the fully connected network to a **locally-connected** network.
- For example, if we set the window size to 5, we only need 5x5x3 = 75 weights per neuron.
- The connectivity is
  - Local in **space** (height and width)
  - Full in **depth** (all 3 RGB channels)
- For a CNN, all neurons in the output feature map share the same 75 weights.
- For a fully connected network we would have had 512 x 512 x 3 = 786,432 weights.

Replication at the same area

Usually we apply not just one, but many filters.

Assuming that we have N filters, the output at each location of the input image is a 1 x 1 x N tensor (array).

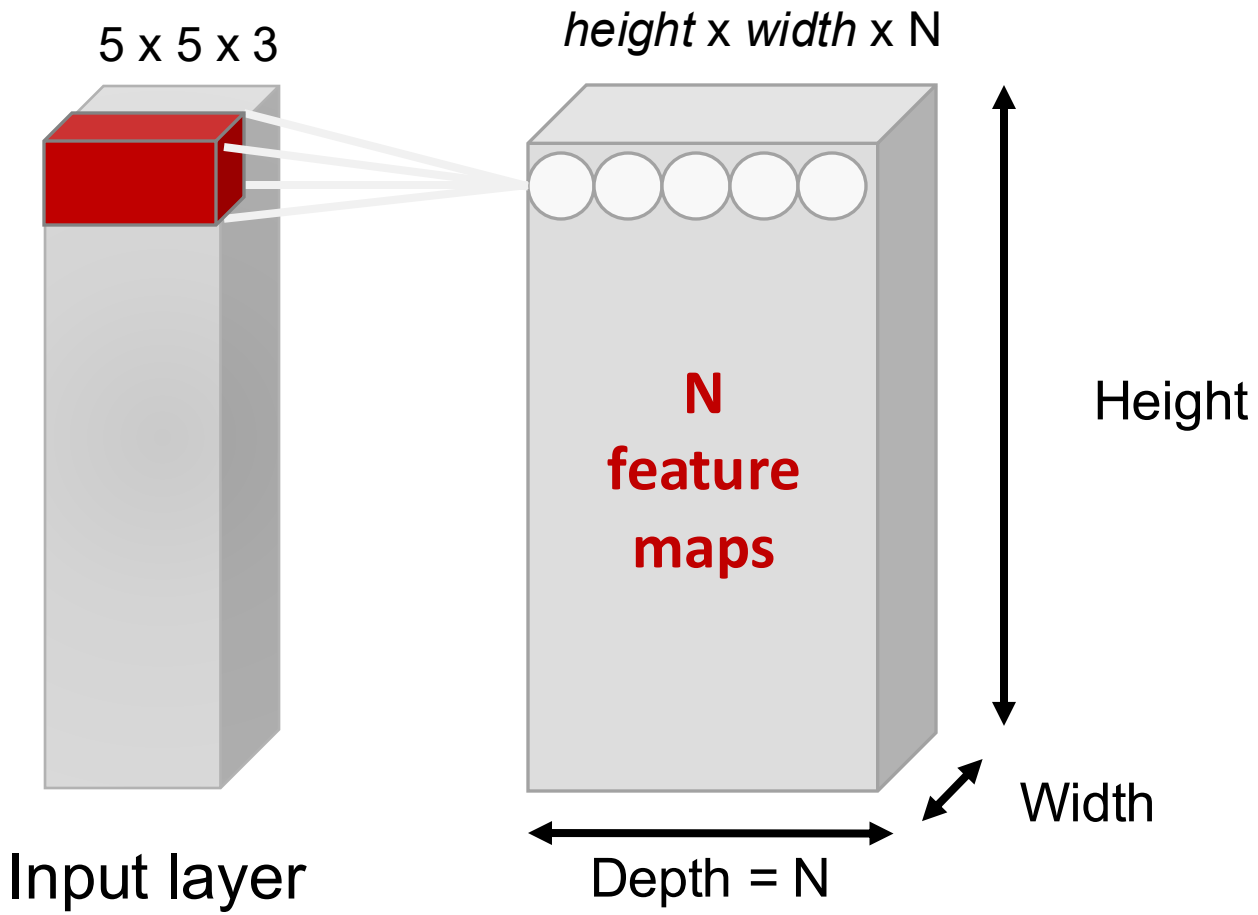


Feature maps

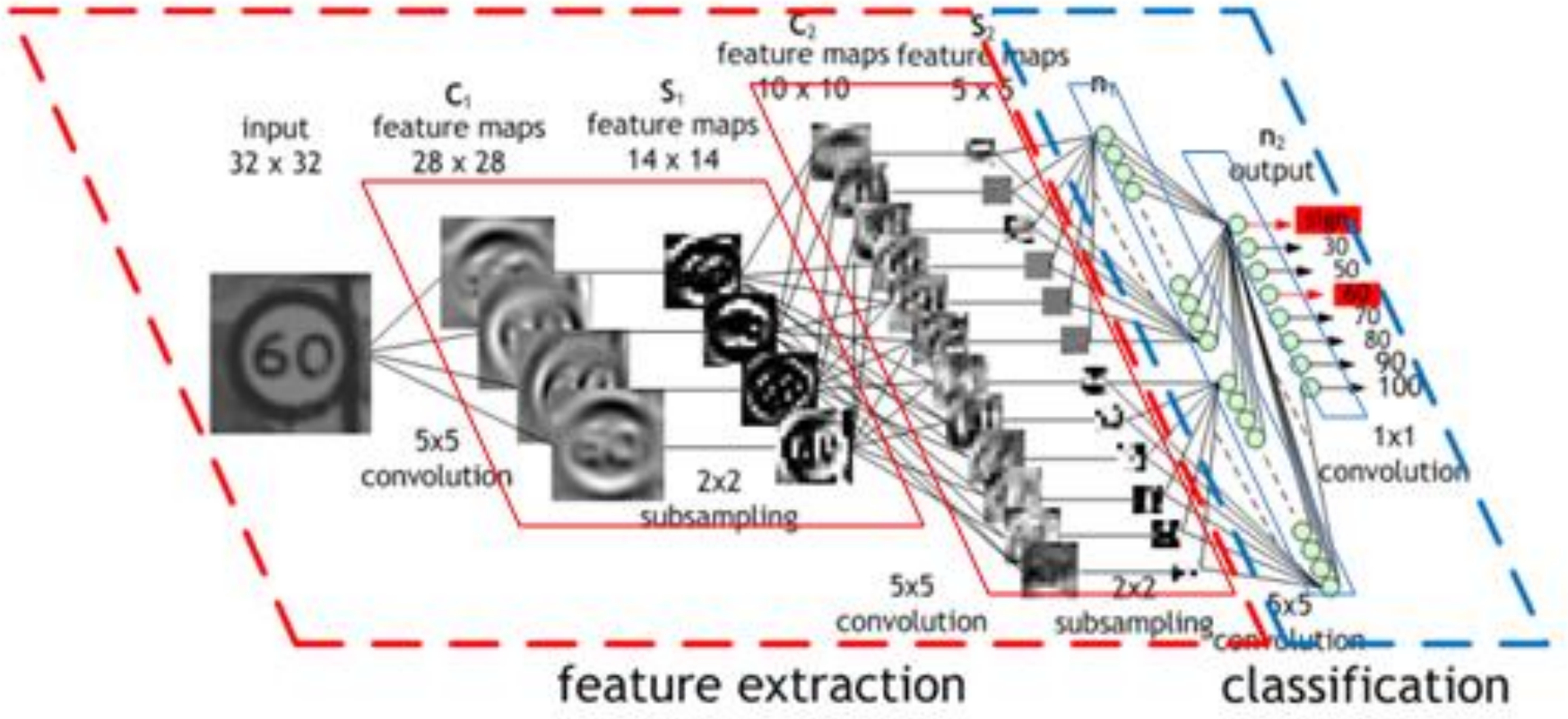
Once we start sliding the filters over the input, an output volume is generated.

We can think of the output as a multi-dimensional image with N channels.

Each channel corresponds to a **feature map** or activation map and is produced by one filter.



Example of feature maps



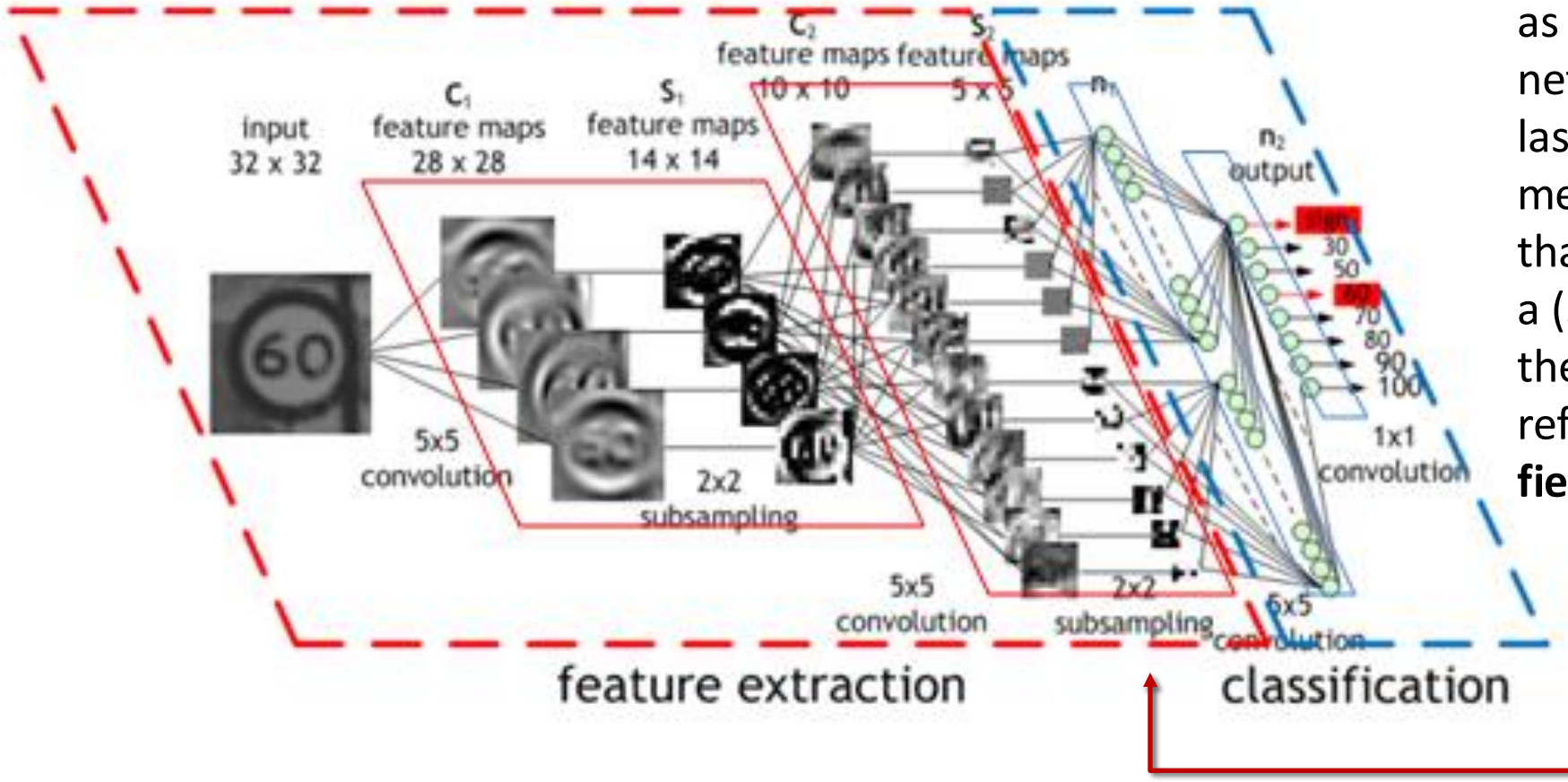
Q: How many feature maps are there in the first layer?
A: 4

Q: How many feature maps are there in the second layer?
A: 12

Q: How many channels do the filters in the first layer have?
A: 1

Q: What about the second layer?
A: 4

Example of feature maps



Also note that the feature maps get smaller and smaller as we move deeper into the network. In this example the last feature map is 5x5. This means that each "neuron" in that feature map maps back to a (32/5)x(32/5) ≈ 6x6 region in the input image. This region is referred to as the **receptive field** of the neuron.

Receptive field

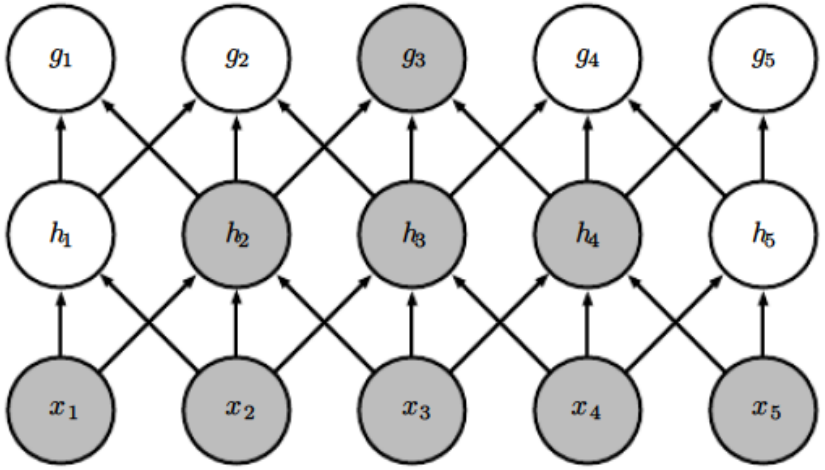


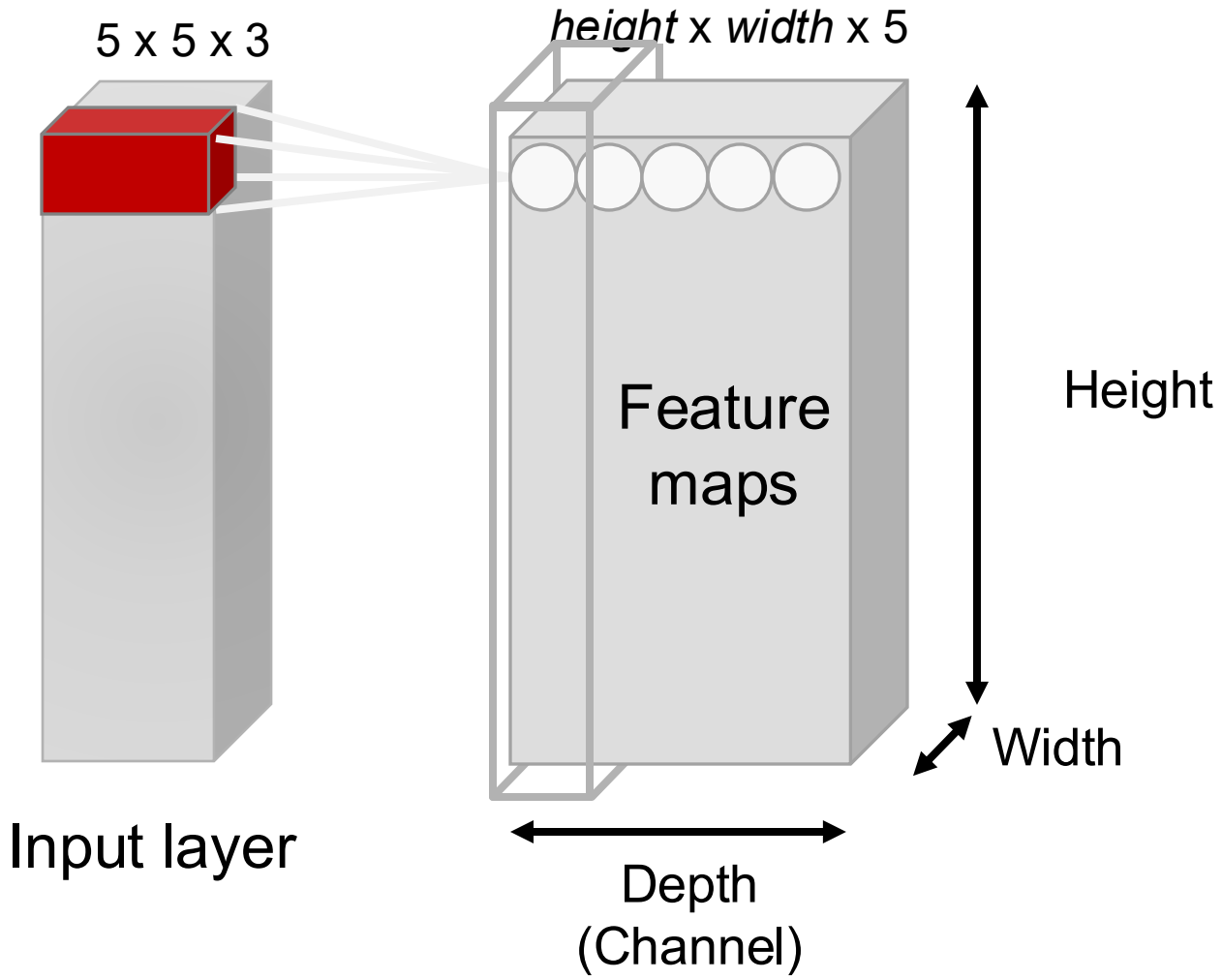
Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (figure 9.12) or pooling (section 9.3). This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.

In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

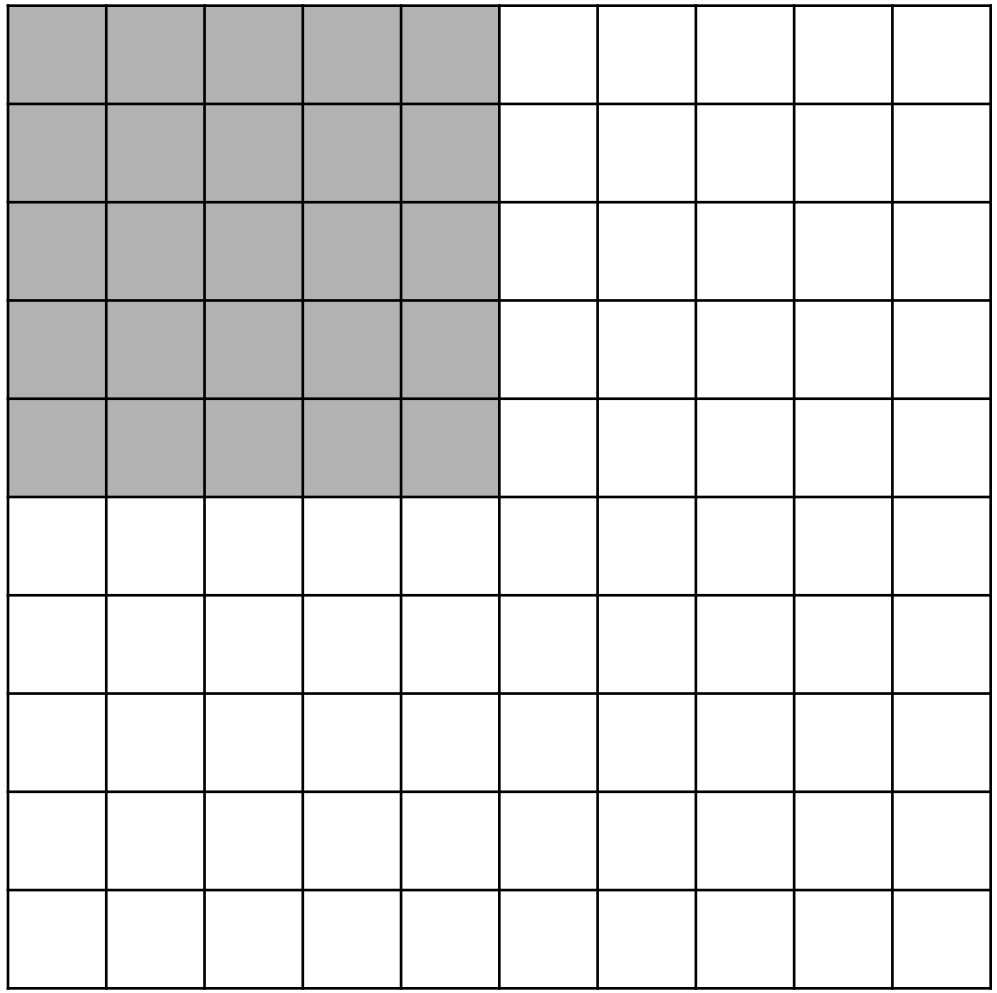
Parameter sharing

CNN feature maps share the same weight/filter parameters at different pixel locations.

So, if we have 5 filters of size 5x5x3, there are only 75x5 = 375 weights in total.



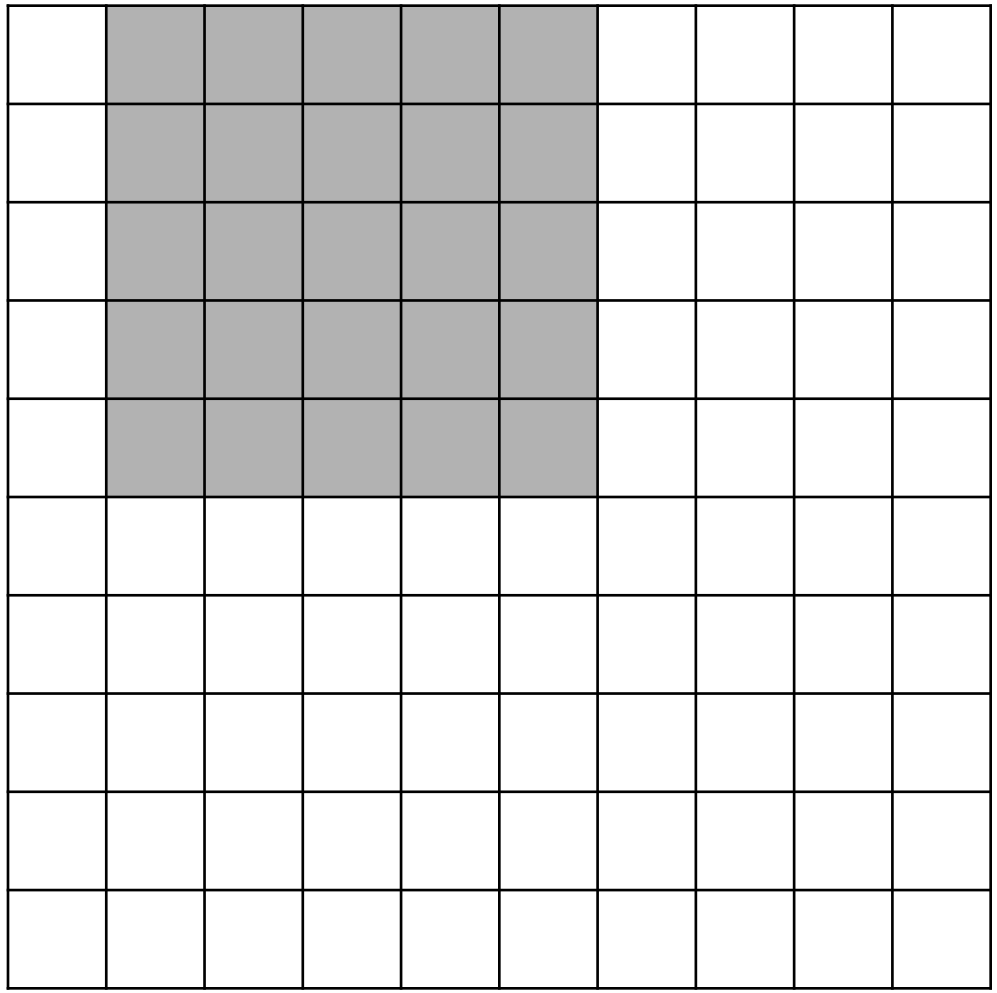
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1

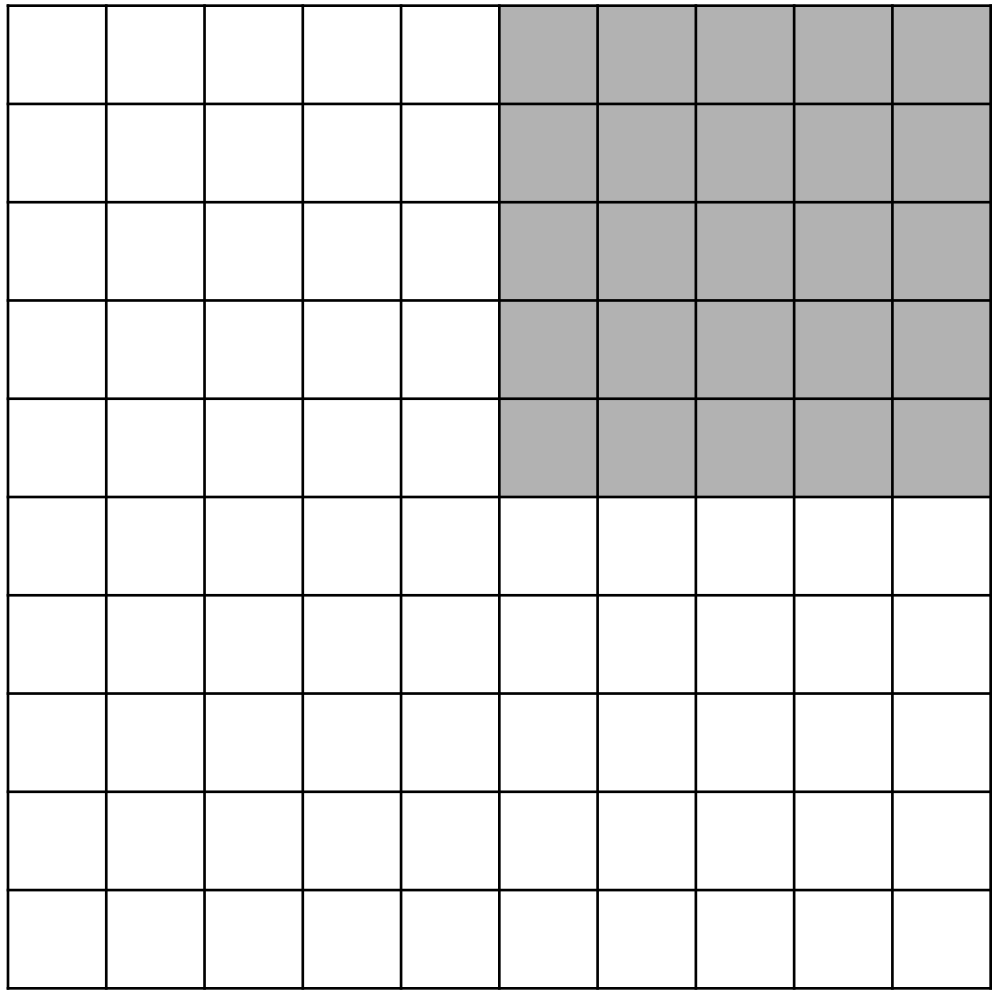
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1

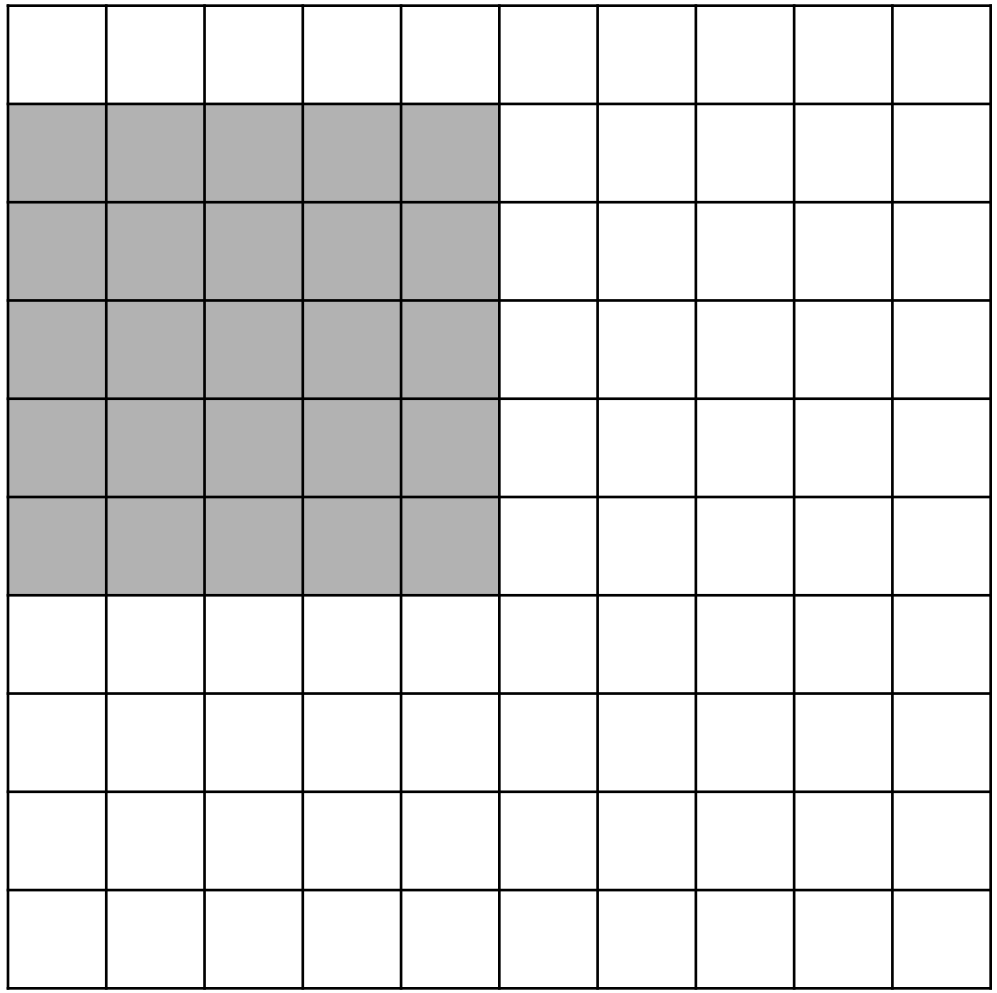
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1

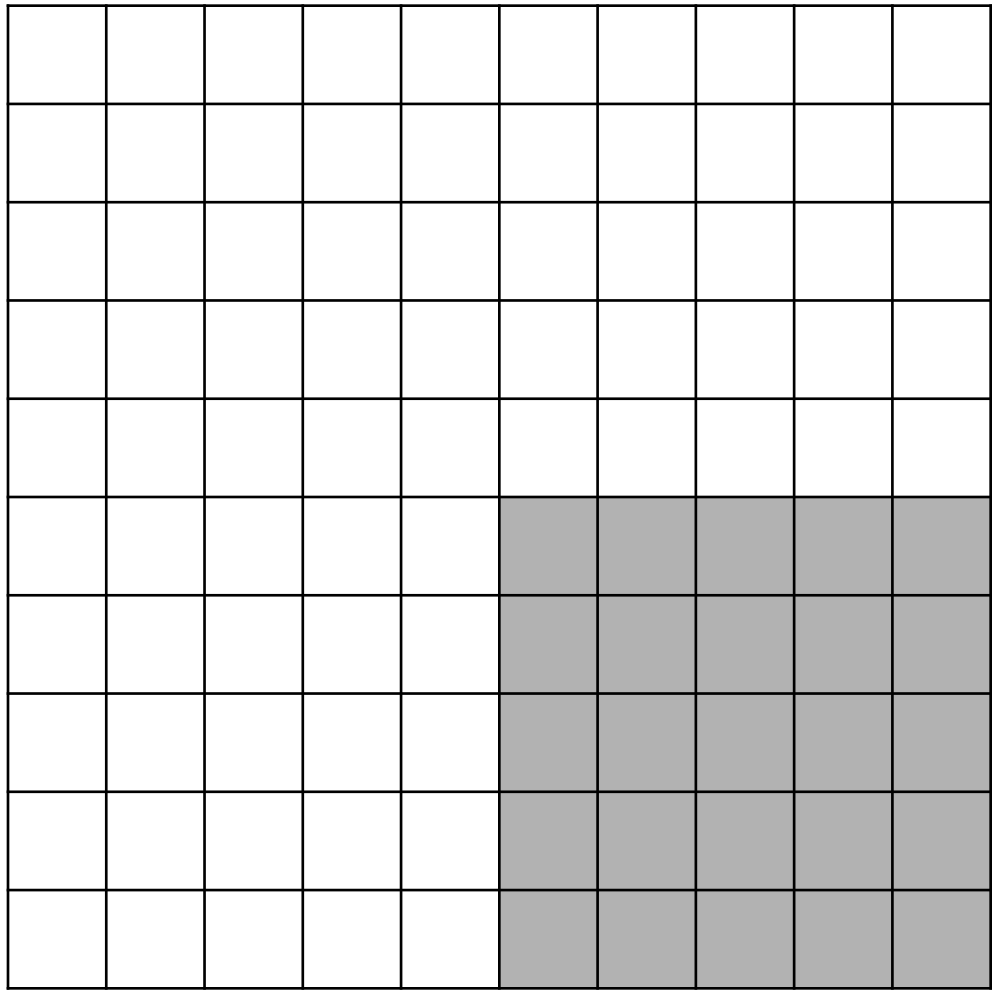
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1

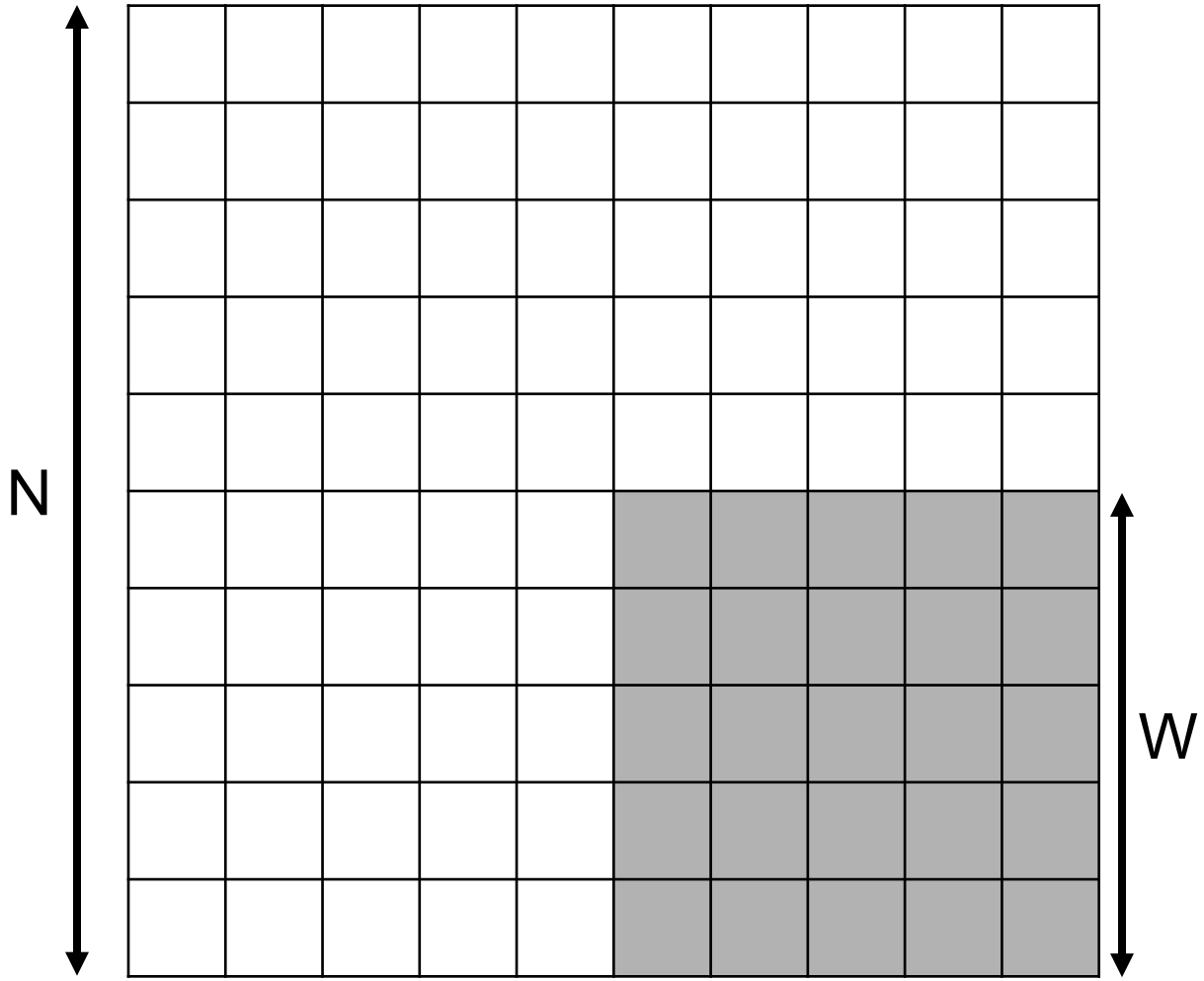
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1
- We get 6x6 outputs.

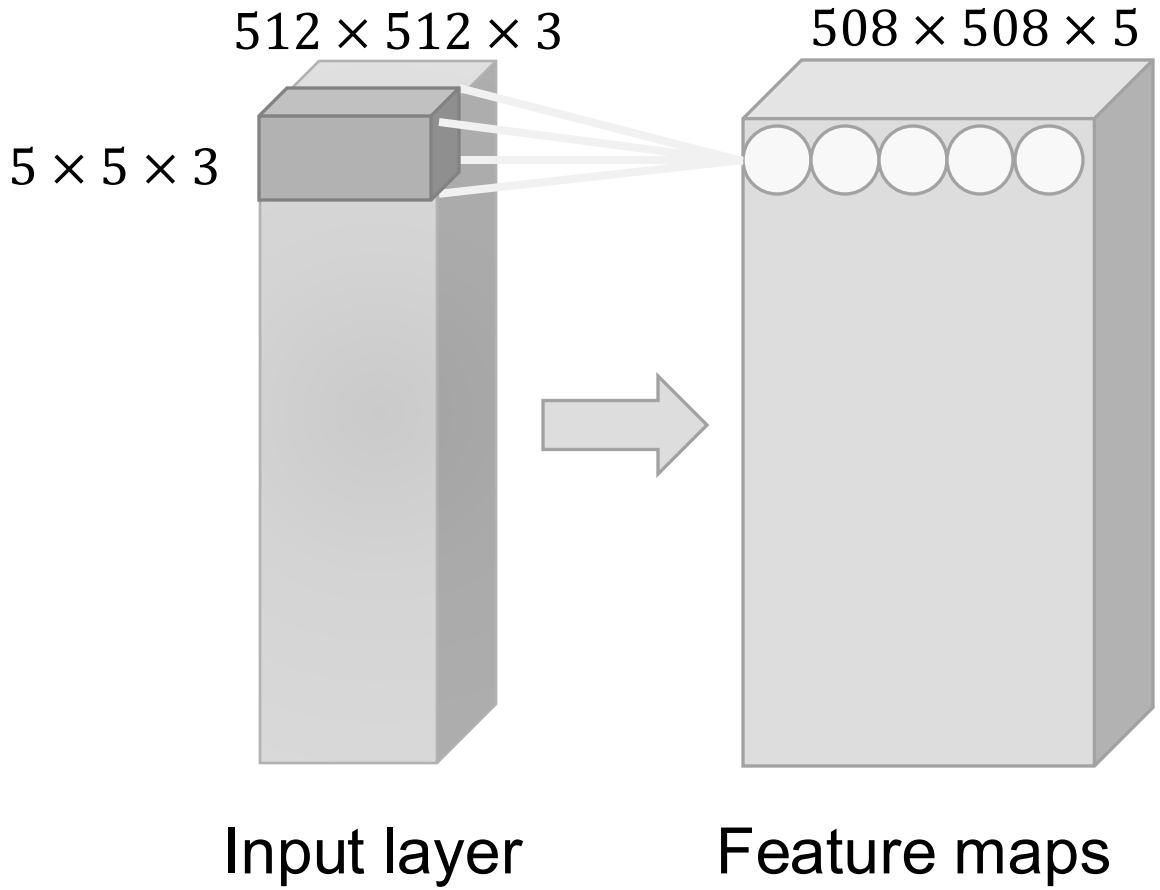
Stride



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1
- We get 6x6 outputs.
- The outputs size: $\frac{(N-W)}{\text{stride}} + 1$

Example: Stride 1 (and no padding…)



Note: The output has depth 5, because there are 5 filters (not shown).

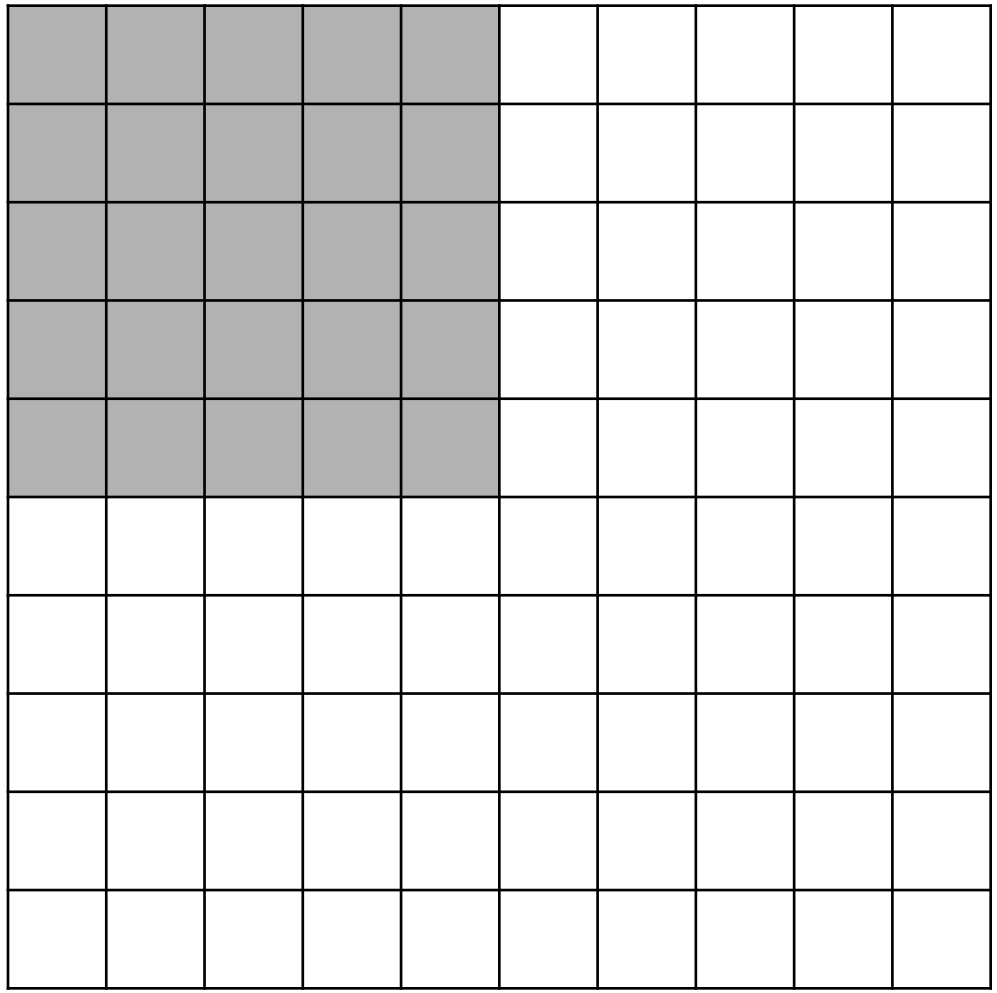
Output size:

$$\frac{(N - W)}{\text{stride}} + 1$$

$$= \frac{512 - 5}{1} + 1$$

$$= 508$$

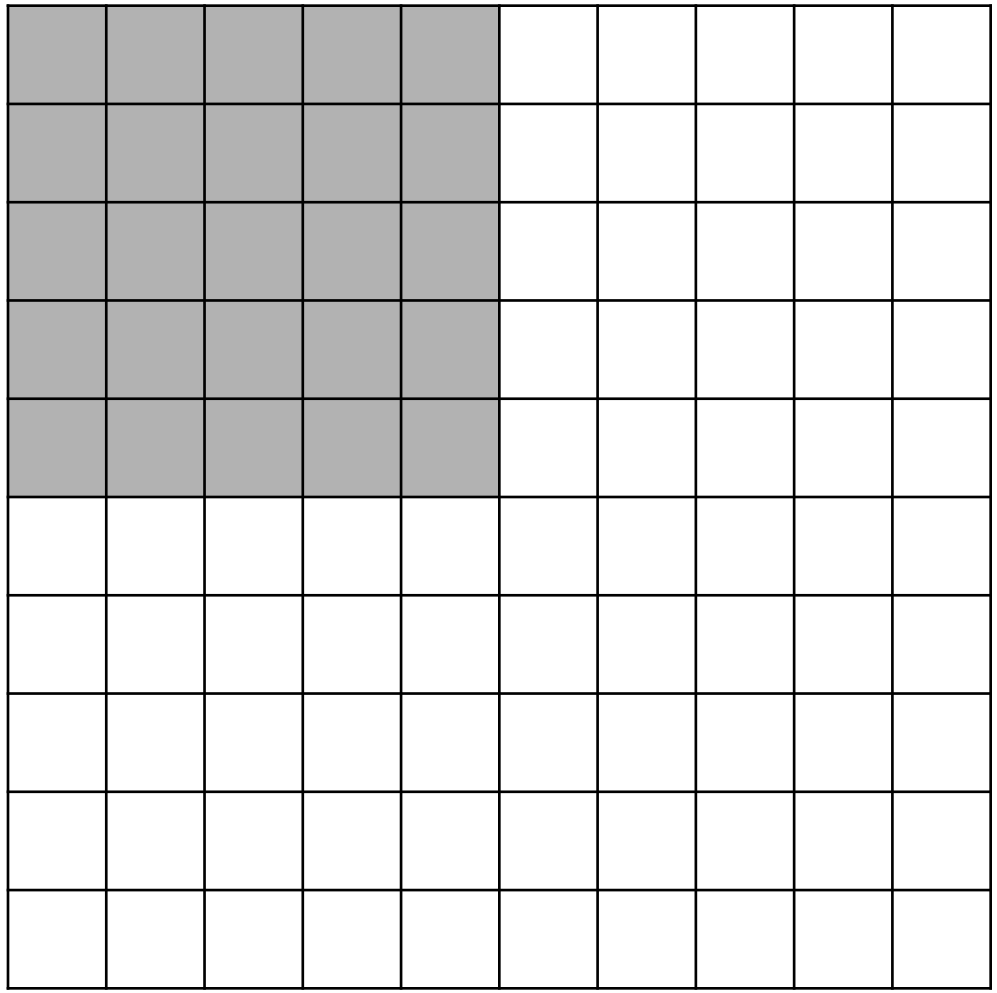
What about stride 2?



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 2

What about stride 2?



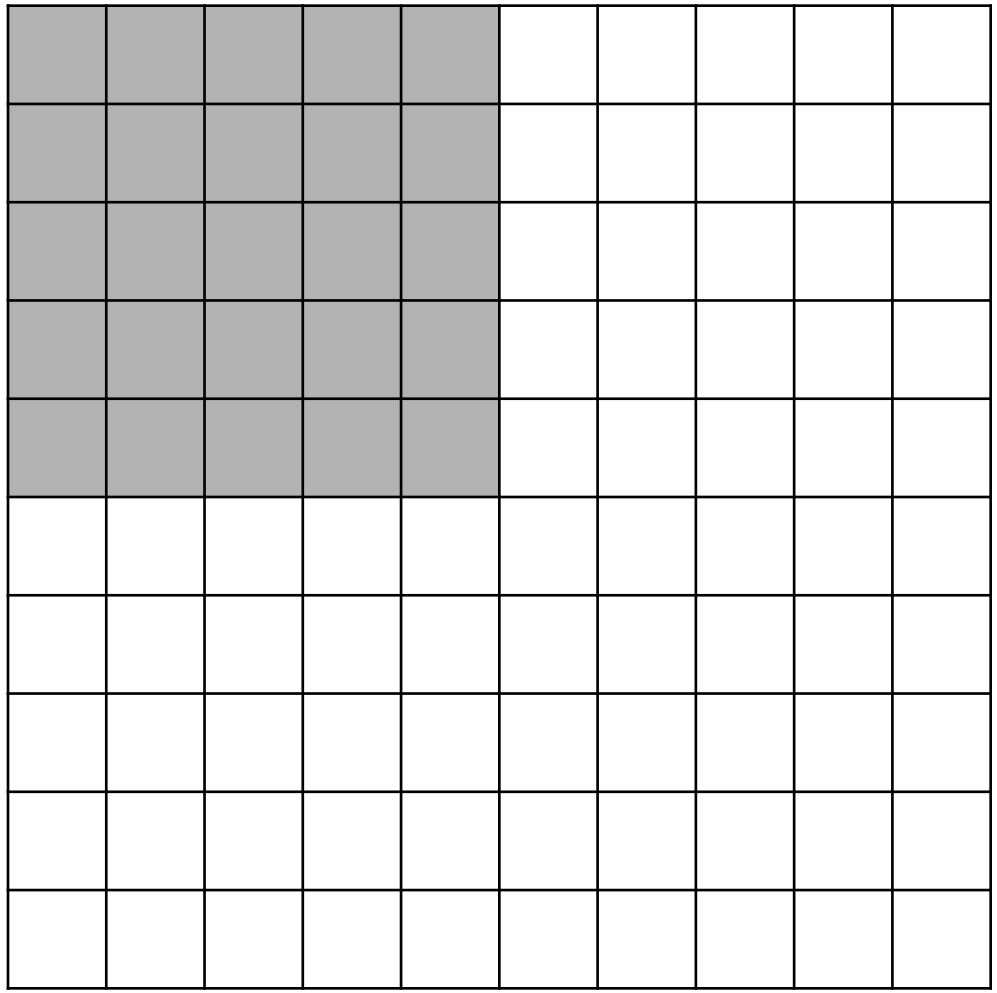
Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 2

- Output size:

$$\frac{(N-W)}{\text{stride}}+1=\frac{(10-5)}{2}+1$$

What about stride 2?



Stride: How many pixels we move the window in one time.

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 2

- Output size:

$$\frac{(N-W)}{\text{stride}}+1=\frac{(10-5)}{2}+1$$

Doesn't fit

In practice

- Common to **zero-pad** border – that means we add zeros at the border of the image.

Zero-padding

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1
  - Pad: $\frac{(W-1)}{2} = 2$

Zero-padding

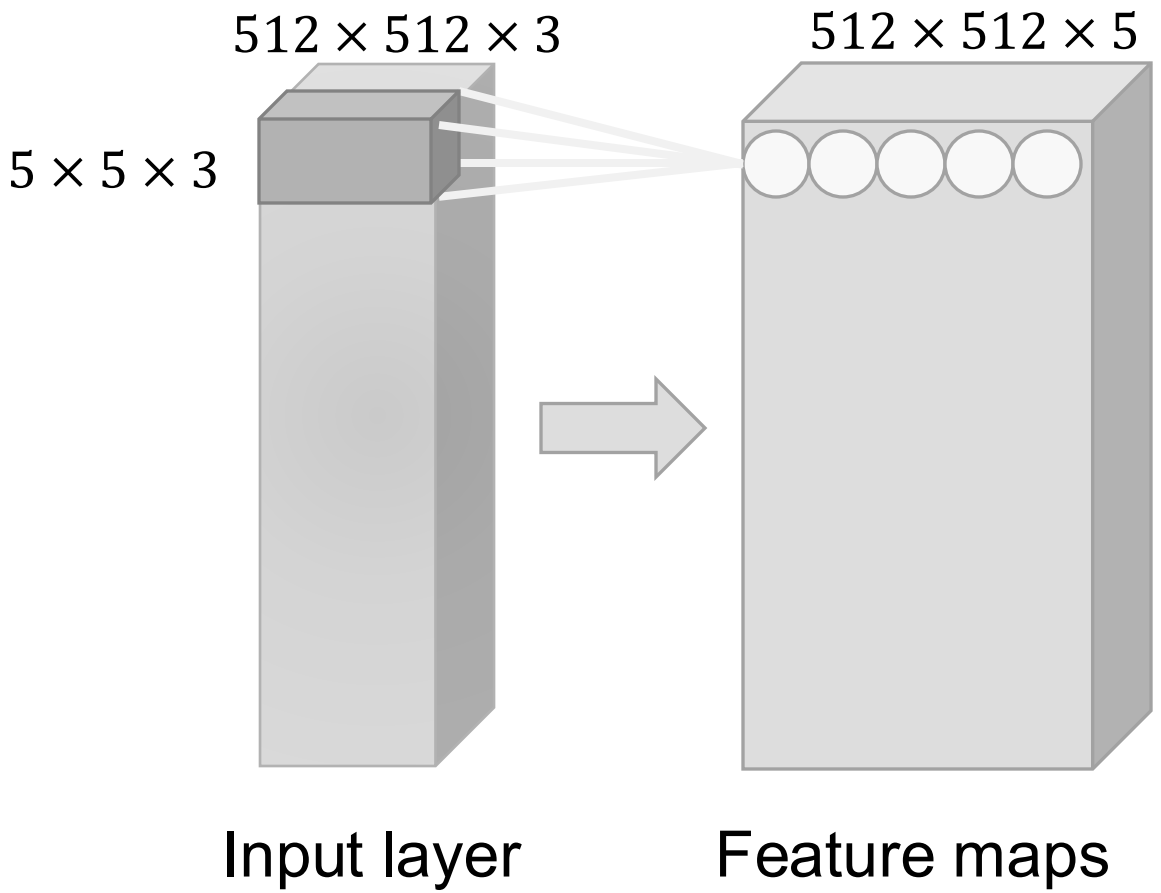
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0											0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

- For example
  - Inputs: 10x10
  - Window size: 5
  - Stride: 1
  - Pad: $\frac{(W-1)}{2} = 2$
- Output size: 10x10 (remains the same)

Zero-padding in practice

- We can keep the same output size by padding with zeros.
- Besides, we can avoid the border information "washing out".
- All common deep learning frameworks support zero-padding.
- Usually provided as input argument to convolution layer:
  - "valid" means no padding – convolution kernel must fit inside input image.
  - "same" means zero-pad such that output has same shape as input.
- See for instance Keras convolution layer: https://keras.io/layers/convolutional/

Example: Stride 1 and padding 2



Note: The output has depth 5, because there are 5 filters (not shown).

To summarize: two main ideas in CNNs

- Neurons are locally connected – output neurons in the feature map depend only on a small chunk of pixels in the input image.
- We use parameter sharing – same filter kernel for each feature map (= convolution)
- We call the resulting layers **convolution layers**

Convolution layer summary

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

Input/output size: Examples from Lab 2

Changing the size of the input image also changes the size of the feature maps.

Left: If the input to MobileNet is $224 \times 224 \times 3$, the output shape of the final convolutional layer is $7 \times 7 \times 1024$*

Right: Reducing the input shape to $120 \times 120 \times 3$, the output of the final layer reduces to $3 \times 3 \times 1024$*

* The last conv. layer is not shown in the figures.

```
mobilenet_full.summary()

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/kera
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `ra
Downloading data from https://github.com/fchollet/deep-learning-mod
17227776/17225924 [==============================] - 13s 1us/step
Model: "mobilenet_1.00_224"
```

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 224, 224, 3)	0
conv1_pad (ZeroPadding2D)	(None, 225, 225, 3)	0
conv1 (Conv2D)	(None, 112, 112, 32)	864
conv1_bn (BatchNormalization	(None, 112, 112, 32)	128
conv1_relu (ReLU)	(None, 112, 112, 32)	0
conv_dw_1 (DepthwiseConv2D)	(None, 112, 112, 32)	288
conv_dw_1_bn (BatchNormaliza	(None, 112, 112, 32)	128
conv_dw_1_relu (ReLU)	(None, 112, 112, 32)	0
conv_pw_1 (Conv2D)	(None, 112, 112, 64)	2048
conv_pw_1_bn (BatchNormaliza	(None, 112, 112, 64)	256
conv_pw_1_relu (ReLU)	(None, 112, 112, 64)	0
conv_pad_2 (ZeroPadding2D)	(None, 113, 113, 64)	0
conv_dw_2 (DepthwiseConv2D)	(None, 56, 56, 64)	576
conv_dw_2_bn (BatchNormaliza	(None, 56, 56, 64)	256
conv_dw_2_relu (ReLU)	(None, 56, 56, 64)	0

```
[ ] conv_base.summary()

Model: "mobilenet_1.00_224"
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 120, 120, 3)	0
conv1_pad (ZeroPadding2D)	(None, 121, 121, 3)	0
conv1 (Conv2D)	(None, 60, 60, 32)	864
conv1_bn (BatchNormalization	(None, 60, 60, 32)	128
conv1_relu (ReLU)	(None, 60, 60, 32)	0
conv_dw_1 (DepthwiseConv2D)	(None, 60, 60, 32)	288
conv_dw_1_bn (BatchNormaliza	(None, 60, 60, 32)	128
conv_dw_1_relu (ReLU)	(None, 60, 60, 32)	0
conv_pw_1 (Conv2D)	(None, 60, 60, 64)	2048
conv_pw_1_bn (BatchNormaliza	(None, 60, 60, 64)	256
conv_pw_1_relu (ReLU)	(None, 60, 60, 64)	0
conv_pad_2 (ZeroPadding2D)	(None, 61, 61, 64)	0
conv_dw_2 (DepthwiseConv2D)	(None, 30, 30, 64)	576
conv_dw_2_bn (BatchNormaliza	(None, 30, 30, 64)	256
conv_dw_2_relu (ReLU)	(None, 30, 30, 64)	0
conv_pw_2 (Conv2D)	(None, 30, 30, 128)	8192
conv_pw_2_bn (BatchNormaliza	(None, 30, 30, 128)	512

Convolution is a linear operation

- 2D convolution can be calculated as a matrix-vector multiplication (i.e., its linear).

Image patches used during convolution

Image Patch 1	Image Patch 2	Image Patch 3
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]
Image Patch 4	**Image Patch 5**	**Image Patch 6**
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]
Image Patch 7	**Image Patch 8**	**Image Patch 9**
[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]	[1., 2., 3., 4.] [5., 6., 7., 8.] [9., 10., 11., 12.] [13., 14., 15., 16.]

Convolution filter: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

Image patches flattened into a matrix

```
[ 1., 2., 3., 5., 6., 7., 9., 10., 11.]
[ 2., 3., 4., 6., 7., 8., 10., 11., 12.]
[ 5., 6., 7., 9., 10., 11., 13., 14., 15.]
[ 6., 7., 8., 10., 11., 12., 14., 15., 16.]
```

Filter flattened into a vector: `[1., 2., 3., 4.]`

Multiplying flattened filter vector with matrix of flattened patches gives the same output is the convolution, except that the output is a vector and not an image. The output vector can be reshaped into an image.

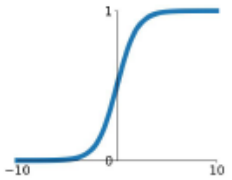
https://medium.com/@_init_/an-illustrated-explanation-of-performing-2d-convolutions-using-matrix-multiplications-1e8de8cd2544

Importance of activation layers

- Because convolution is linear, we need non-linear activation functions between convolution layers.
- Modern CNNs use ReLU by default.
- Stands for **Re**ctified **L**inear **U**nit.
- Why?
  - It's trivial to calculate the derivative, which helps speed up computation during optimization.
  - It avoids the saturation problem (small gradients in the flat regions of sigmoid and tanh). This speeds up the optimization (= faster convergence).

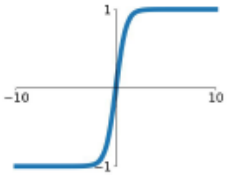
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



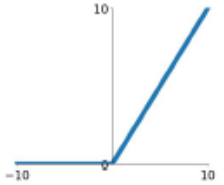
tanh

$$\tanh(x)$$



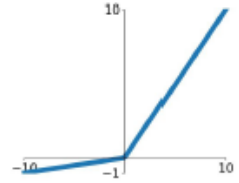
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

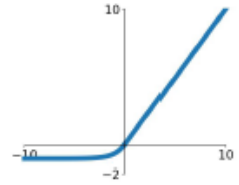


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

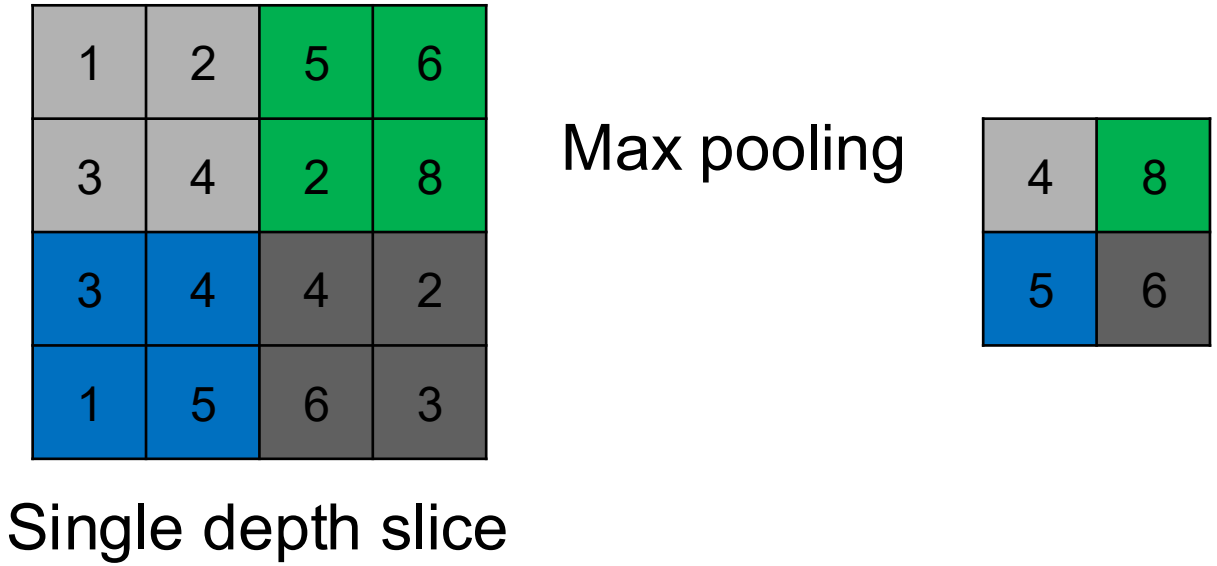


Other layer types in CNNs

- Pool layer
- Fully connected layer
- (Batch normalization)
- (Dropout)

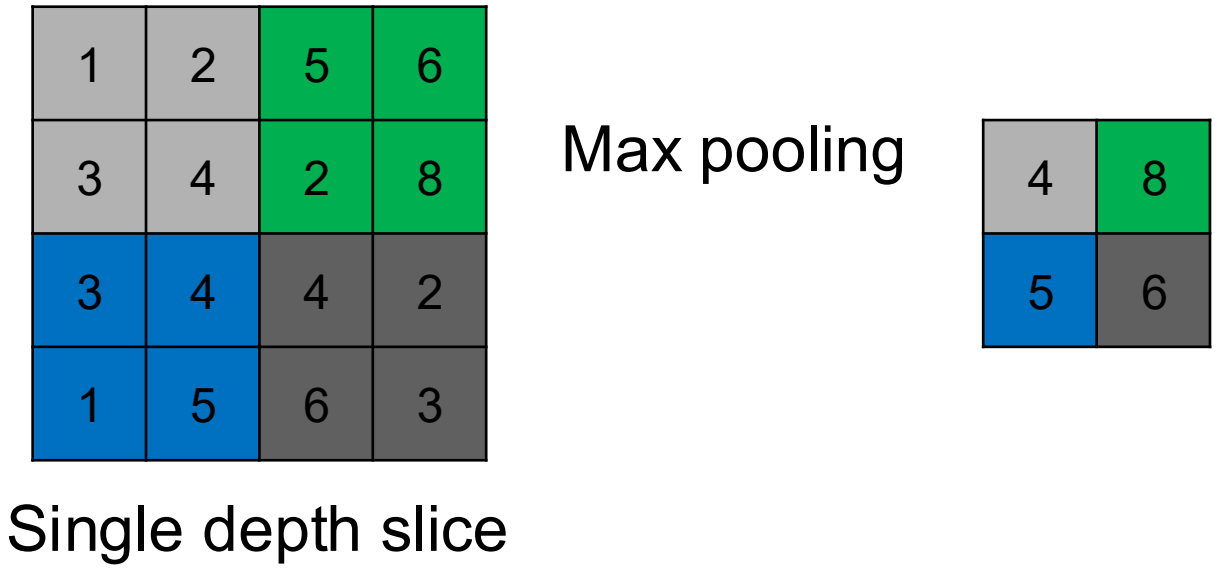
Pool layers

- The convolution layers are often followed by pool layers.
- It can reduce the number of weights by keeping only the most important information.
- We often use the **max** operation to do pooling, and less often **averaging**.
- Max pooling adds some **translation invariance** and it **increases the receptive field**.



Window size and stride in pool layers

- The window size is the pooling range.
- The stride is how many pixels the window moves.
- For this example, window size = stride = 2.



Translation invariance

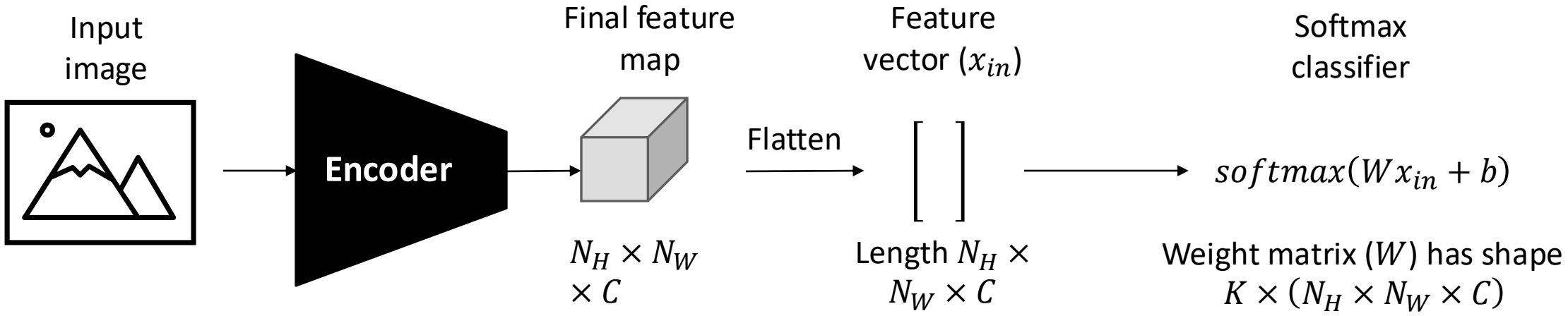
- A max pool over a small grid will detect a feature regardless of where in that grid that feature occurs.
- Max pooling therefore helps to make the representation become approximately **invariant** to small translations of the input.
- Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.
- *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.*

Is max pooling deprecated?

- One of the great founders of neural networks, Geoffrey Hinton, has pointed out that the max pooling operation throws away a lot of information about **where** a feature occurs.
- For example, if you want to do face recognition, it may be important to know exactly where the position of the mouth is relative to, say, the nose.
- This can't be done if the exact positional information is thrown away by repeated pooling operations.
- Hinton proposed alternative networks, including Transforming Autoencoders and Capsule Networks – a really interesting idea (that is hard to wrap your head around).
- What to do instead of max pooling?
  - Perform convolution with stride = 2.
  - Has the same effect of down-sampling by a factor of two (like max pooling normally does), but information is not thrown away in the same way as with max pooling.

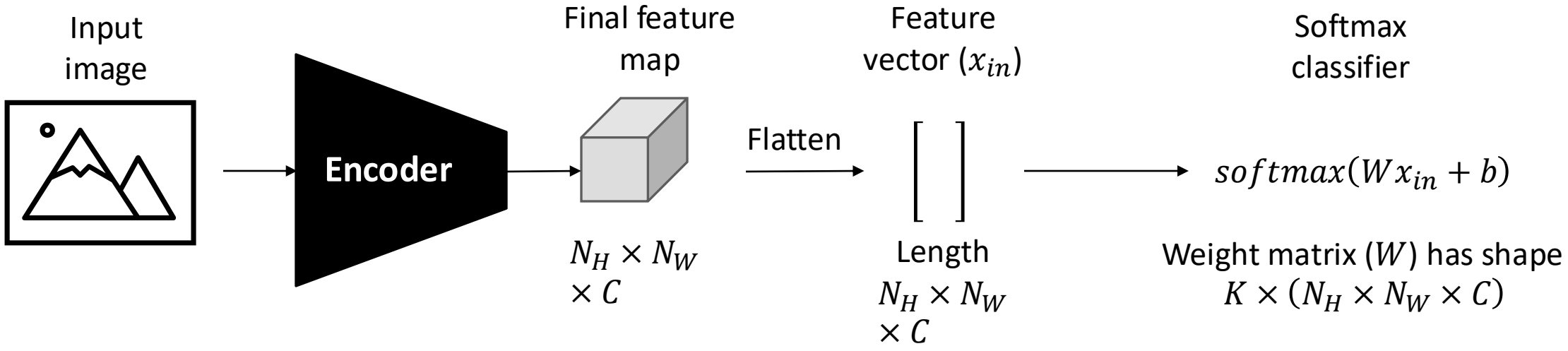
Pooling as a tool to handle varying sizes

- For many tasks, pooling is essential for handling inputs of varying size.
- **Problem:** Suppose we want to classify images using a softmax classifier. Then the input (x_{in}) to the classifier must be a vector of fixed length due to the matrix-vector product (Wx_{in}). However, if we change the size of the input image, the size of the final feature also changes, which eventually causes to have a shape x_{in} that doesn't match the shape of W.



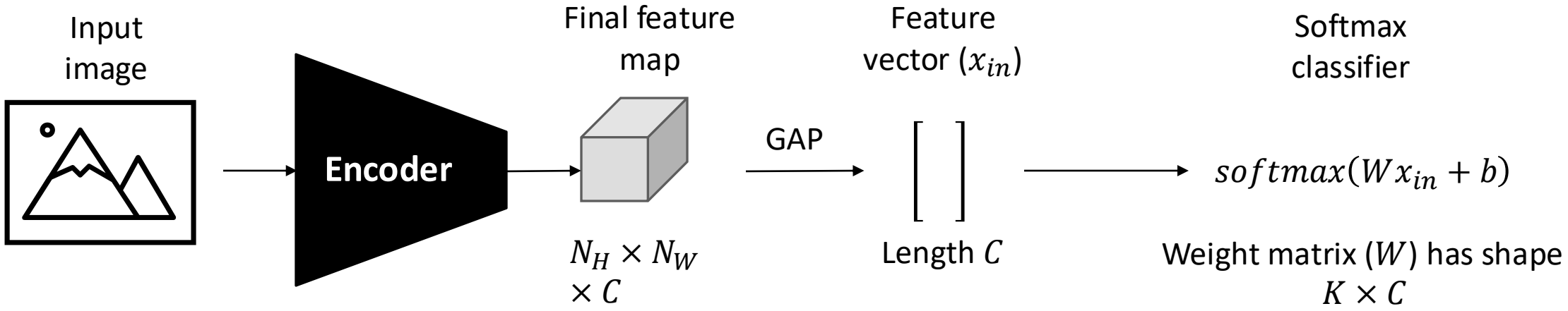
Pooling as a tool to handle varying sizes

- **Example (see Lab 2 or this slide)**
  - If the input shape to a pre-trained MobileNet is 120x120 pixels, the output of the convolutional base has shape $N_H \times N_W \times C = 3 \times 3 \times 1024$.
  - But if the input to the same network has shape 224x224, the output shape of the convolutional base is instead $N_H \times N_W \times C = 7 \times 7 \times 1024$.
- **Observation:** Only $N_H \times N_W$ are affected by the input shape, while C remains unchanged.



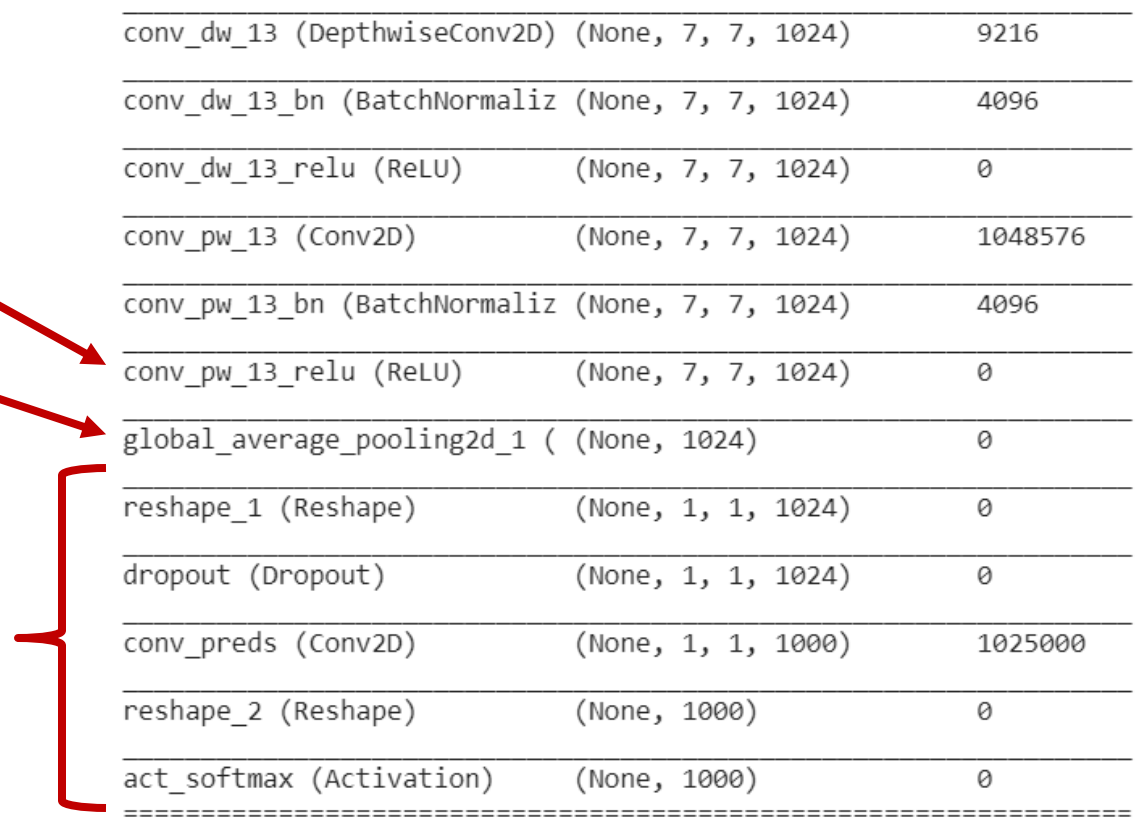
Pooling as a tool to handle varying sizes

- **Solution:** Use Global Average Pooling (GAP) pooling to create input vector of fixed length.
- **Example:**
  - The pooling function (GlobalAveragePooling2D) averages over the first two dimensions to produce a vector of length C (1024 in the example above).
  - The classifier now works for arbitrary input shapes, because the shapes of x_{in} and W have become independent of $N_H \times N_W$.



Pooling as a tool to handle varying sizes

- **Example**
  - MobileNet with input size 224x224 results in an output of the conv base with shape 7x7x1024.
  - Global average pooling averages over X and Y dimensions to produce a vector of length 1024.
  - 1024 is the input shape that the **classifier part** of the network expects.

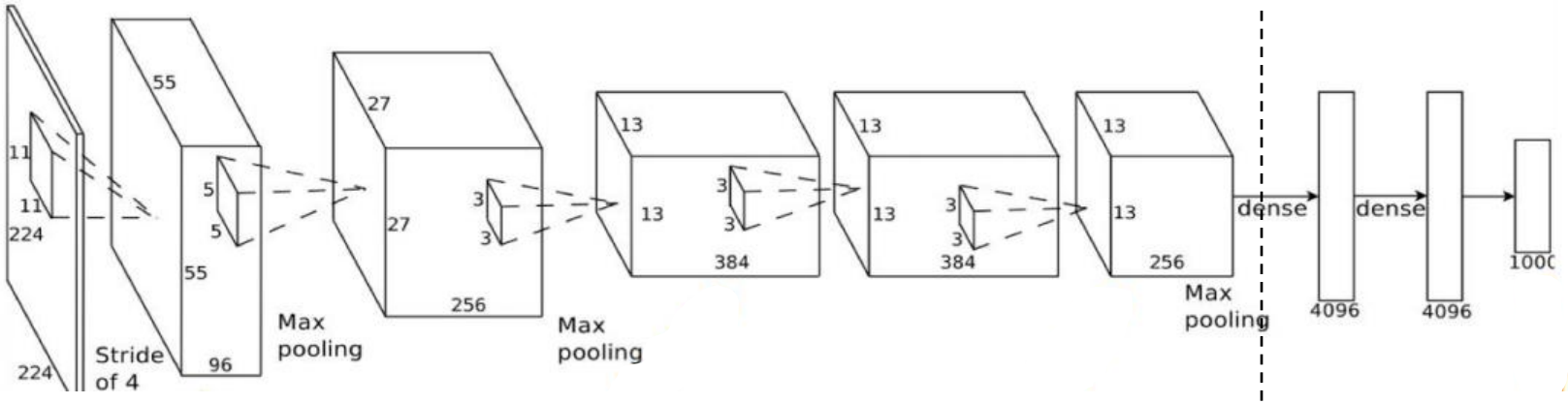


conv_dw_13 (DepthwiseConv2D)	(None, 7, 7, 1024)	9216
conv_dw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
global_average_pooling2d_1 (	(None, 1024)	0
reshape_1 (Reshape)	(None, 1, 1, 1024)	0
dropout (Dropout)	(None, 1, 1, 1024)	0
conv_preds (Conv2D)	(None, 1, 1, 1000)	1025000
reshape_2 (Reshape)	(None, 1000)	0
act_softmax (Activation)	(None, 1000)	0

Fully connected layer

- This layer type is the same as the layers of traditional neural networks.
- They are also called **dense** or **linear** layers.
- We often use this type of layer **at the end of CNNs**.

$$x_{out} = f(Wx_{in} + b)$$
$$x_{in} \in \mathbb{R}^n, \mathrm{W} \in \mathbb{R}^{m \times n}, \mathrm{b} \in \mathbb{R}^m, x_{out} \in \mathbb{R}^m$$
$$f = activation\ function$$



Convolutional base (encoder)

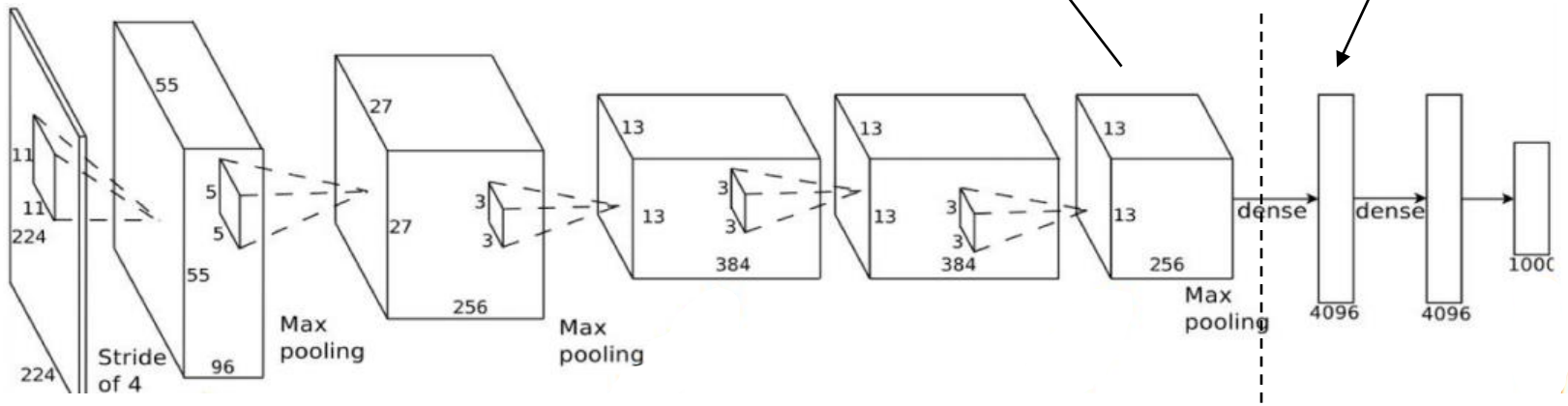
Fully connected NN (classifier)

Fully connected layer

Output of convolutional base is a 3D tensor.
Example: In AlexNet, the output shape is 6x6x256
(after max pooling – not shown in the figure)

Input to fully connected layers is a vector of length 6x6x256 = 9,216

(Reshaping is done explicitly in Keras using Flatten or Reshape layer).



Convolutional base (encoder)

Fully connected NN (classifier)

1x1 convolution

- We can use 1x1xM convolutions to mimic fully connected layers.
- Example (Keras' MobileNet):
  - Output of conv base: 7x7x1024
  - Perform global max pooling to get 1024 dim. vector
  - Reshape to "volume" of size 1x1x1024
  - Perform convolution with 1x1x1000 kernel (= dense)

conv_dw_13 (DepthwiseConv2D)	(None, 7, 7, 1024)	9216
conv_dw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
global_average_pooling2d_1 (	(None, 1024)	0
reshape_1 (Reshape)	(None, 1, 1, 1024)	0
dropout (Dropout)	(None, 1, 1, 1024)	0
conv_preds (Conv2D)	(None, 1, 1, 1000)	1025000
reshape_2 (Reshape)	(None, 1000)	0
act_softmax (Activation)	(None, 1000)	0

1x1 convolution

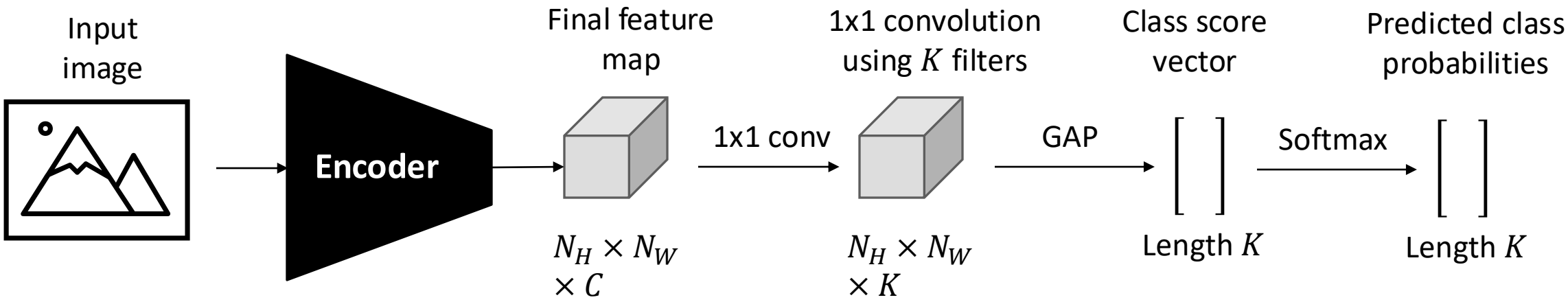
- We can use 1x1xM convolutions to mimic fully connected layers.
- Example (Keras' MobileNet):
  - Output of conv base: 7x7x1024
  - Perform global max pooling to get 1024 dim. vector
  - Reshape to "volume" of size 1x1x1024
  - Perform convolution with 1x1x1000 kernel (= dense)
- Alternative implementation

conv_dw_13 (DepthwiseConv2D)	(None, 7, 7, 1024)	9216
conv_dw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_dw_13_relu (ReLU)	(None, 7, 7, 1024)	0
conv_pw_13 (Conv2D)	(None, 7, 7, 1024)	1048576
conv_pw_13_bn (BatchNormaliz	(None, 7, 7, 1024)	4096
conv_pw_13_relu (ReLU)	(None, 7, 7, 1024)	0
~~global_average_pooling2d_1 (~~	~~(None, 1024)~~	~~0~~
~~reshape_1 (Reshape)~~	~~(None, 1, 1, 1024)~~	~~0~~
dropout (Dropout)	(None, 7, 7, 1024)	0
conv_preds (Conv2D)	(None, 7, 7, 1000)	1025000
~~reshape_2 (Reshape)~~	~~(None, 1000)~~	~~0~~
~~act_softmax (Activation)~~	~~(None, 1000)~~	~~0~~

Global average pooling	(None, 1, 1, 1000)
Reshape	(None, 1000)
Softmax	(None, 1000)

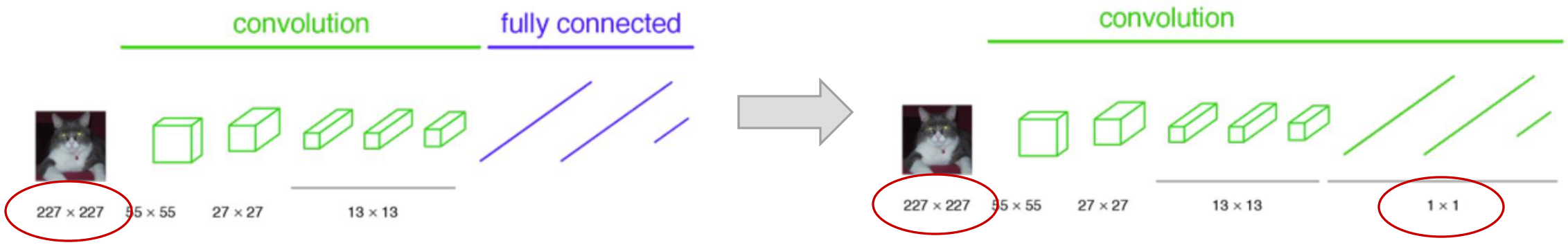
1x1 convolution

- **Example:**
  - In the example on the previous slide the shape of the final feature map is $N_H \times N_W \times C = 7 \times 7 \times 1024$ and the number of classes is $K = 1000$.
  - We can think of the 1x1 convolution layer as the equivalent of calculating $Wx_{in} + b$, except that we now do this for each pixel in the final feature map.
  - To get the predicted class probabilities, we perform GAP followed by softmax.

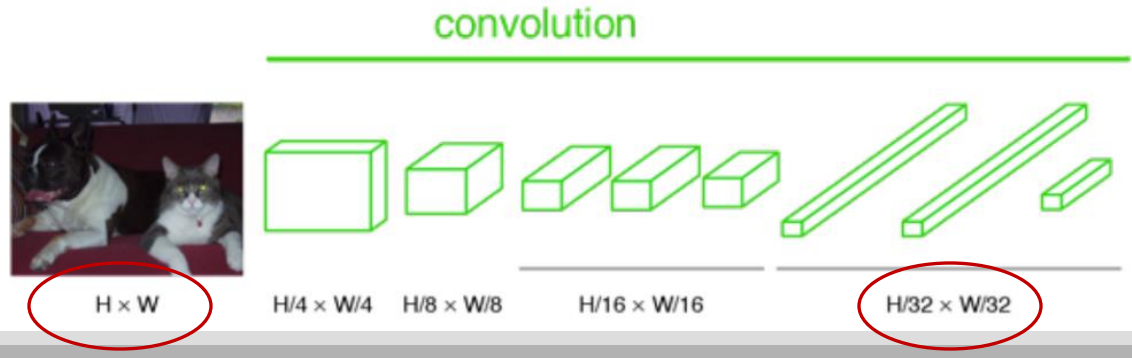


Historical note on 1x1 convolutions

- Fully Convolutional Networks were the first CNNs that could handle input image of arbitrary shape.
- They use 1x1xM convolutions to mimic fully connected layers.
- Allows to handle input images of arbitrary shape.



Input image with arbitrary shape: H x W



Output image: H/32 x W/32
Each pixel contains vector of class probabilities.

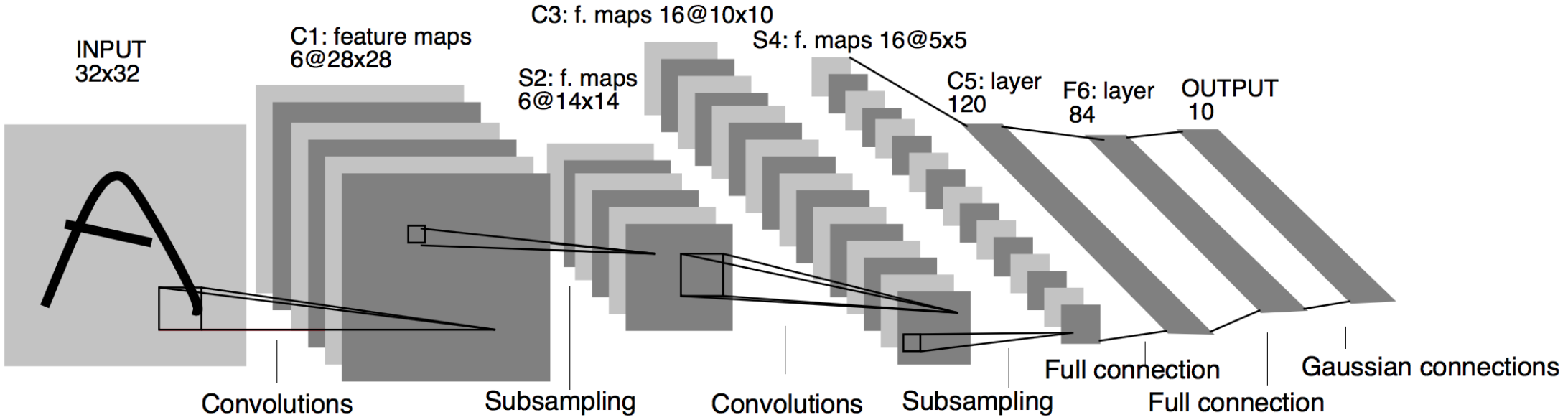
https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1

Simple CNN architectures

LENET-5 AND ALEXNET

Example: LeNet-5

- First-ever CNN (1998)
- Conv filters were 5x5, applied at stride 1.
- Subsampling (Pooling) layers were 2x2 applied at stride 2
- Architecture is [CONV-POOL-CONV-POOL-FC-FC]



https://medium.com/@pechyonkin/key-deep-learning-architectures-lenet-5-6fc3c59e6f4

LeNet-like architecture defined in Keras

```
model = keras.Sequential()
model.add(layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu', input_shape=(32,32,1)))
model.add(layers.AveragePooling2D())
model.add(layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'))
model.add(layers.AveragePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation = 'softmax'))
```

https://medium.com/@mgazar/lenet-5-in-9-lines-of-code-using-keras-ac99294c8086

https://github.com/BVLC/caffe/blob/master/examples/mnist/lenet.prototxt

LeNet-like architecture defined in Keras

```
model = keras.Sequential()
model.add(layers.Conv2D(filters=6, kernel_size=
model.add(layers.AveragePooling2D())
model.add(layers.Conv2D(filters=16, kernel_size
model.add(layers.AveragePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(units=120, activation='r
model.add(layers.Dense(units=84, activation='re
model.add(layers.Dense(units=10, activation = '
```

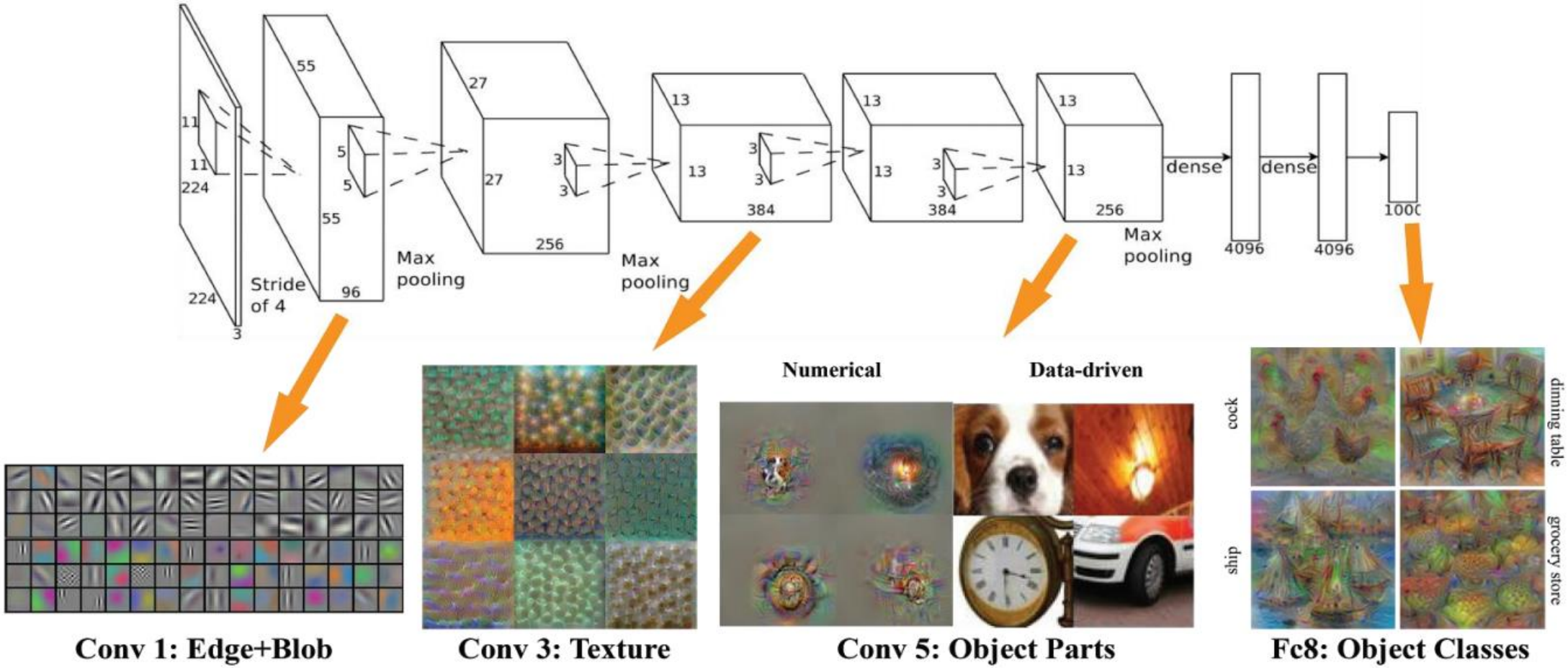
https://medium.com/@mgazar/lenet-5-in-9

https://github.com/BVLC/caffe/blob/r

```
model.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 30, 30, 6)         60
_________________________________________________________________
average_pooling2d_2 (Average (None, 15, 15, 6)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 13, 16)        880
_________________________________________________________________
average_pooling2d_3 (Average (None, 6, 6, 16)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_3 (Dense)              (None, 120)               69240
_________________________________________________________________
dense_4 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_5 (Dense)              (None, 10)                850
=================================================================
Total params: 81,194
Trainable params: 81,194
Non-trainable params: 0
```

AlexNet



Where? → **What?**

VGGNet

- **General pattern:** Double the number of channels after each maxpool operation.
- Why?
- Recall:
  - Early layers detect **where** things are.
  - Late layers extract information about **what** is in the image.
- To represent what is in the image, we need more dimensions (= more channels).
- This is why we often double the number of channels after maxpool.

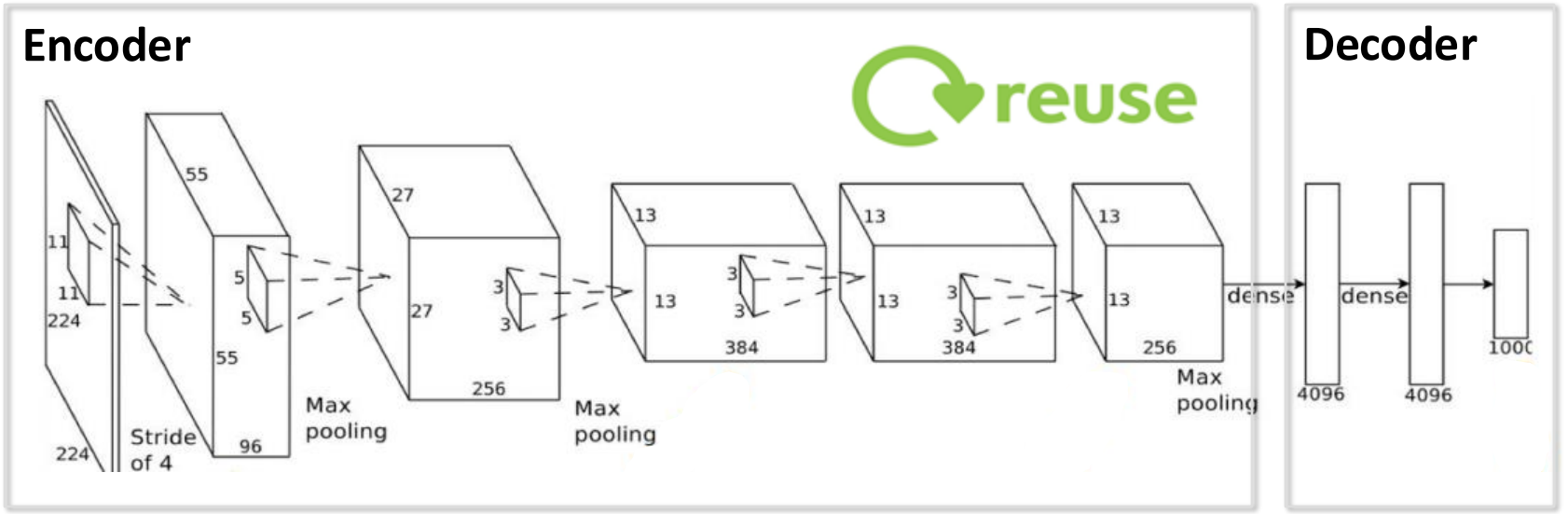
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Training CNNs requires large data sets

- There are still many weights in CNNs because of the large depth, big image size and deep CNN structure.
- Training CNNs from scratch is very **time-consuming**.
- We **need large training data sets** and/or regularization to **avoid overfitting**.
- More about regularization techniques in a later lecture…

One possible solution: Transfer Learning



- Training a ConvNet from scratch can take days
- Use pre-trained encoder trained on ImageNet
- Fine-tune weights of decoder
- Training time: Typically less than 1 hour
- More details: How transferable are features in deep neural networks?

http://cs231n.github.io/transfer-learning/

Applications

Tons and tons of useful applications!!!

- See for example Lab 3:
  - Image classification
  - Convolutional auto encoders
  - Denoising auto encoders
  - Image super resolution
  - Image regression
  - Object detection
  - Image segmentation
  - Few-shot learning with Siamese networks
  - Generative Adversarial Networks (GANs)

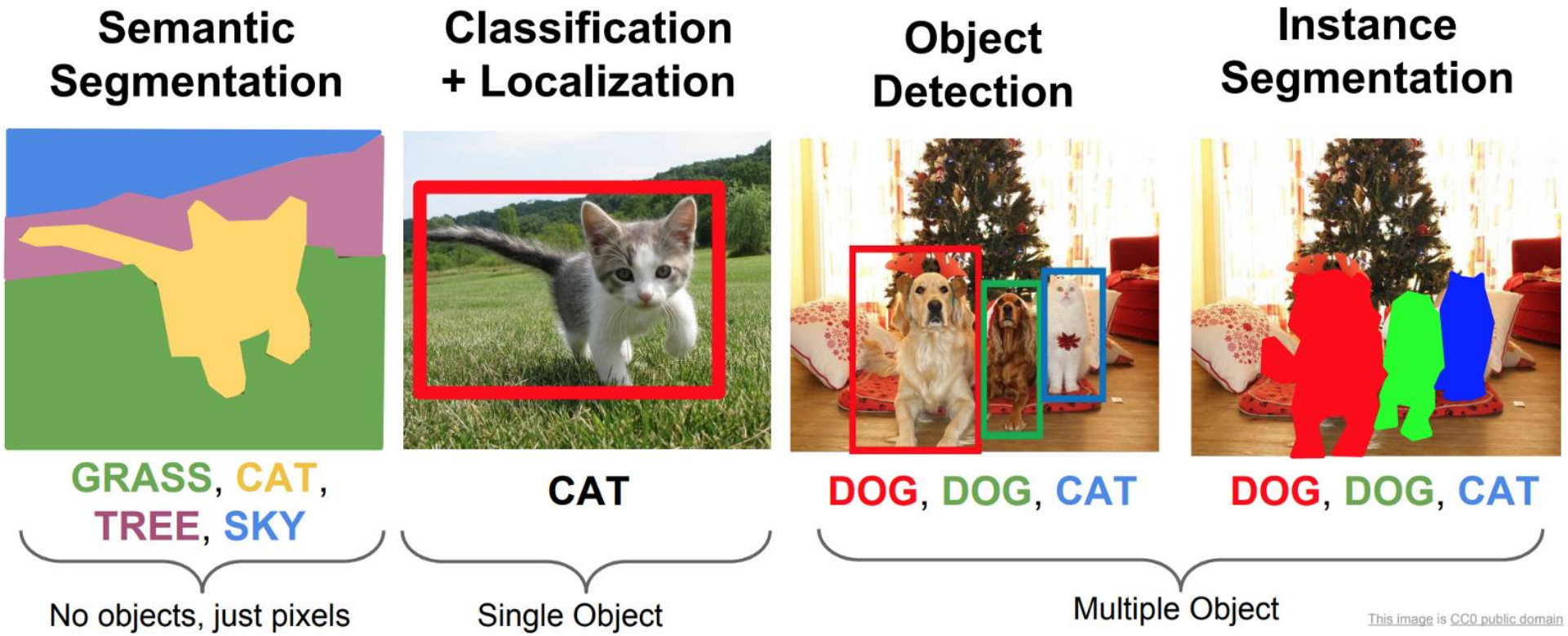
Image classification and image retrieval

Classification

Retrieval = find similar images with CNN features + K-NN

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

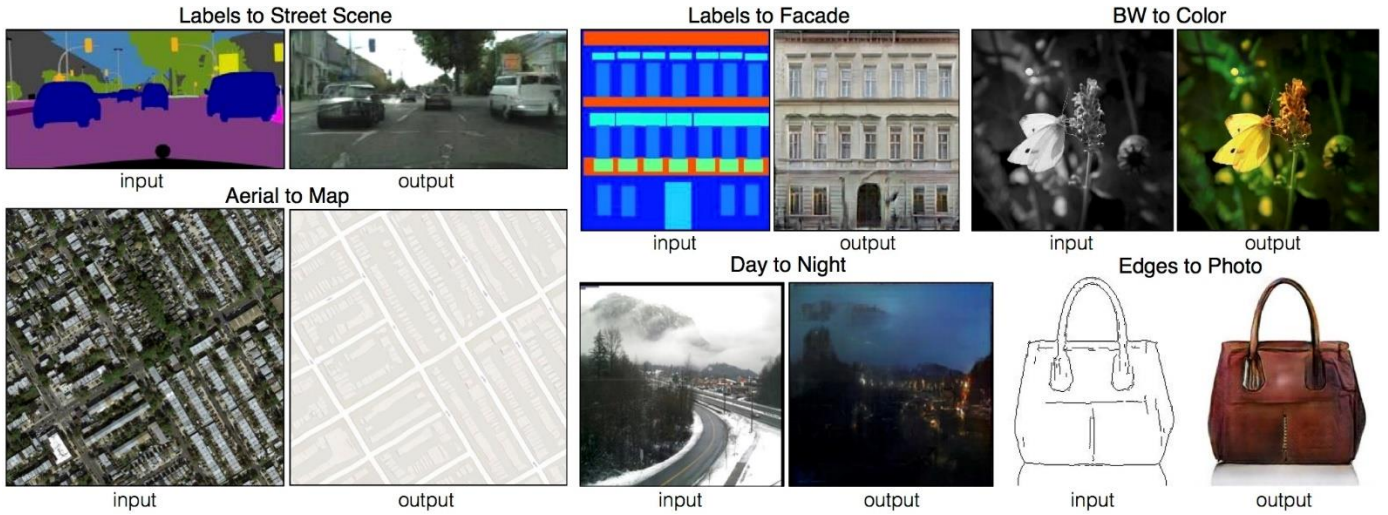
Object detection and segmentation

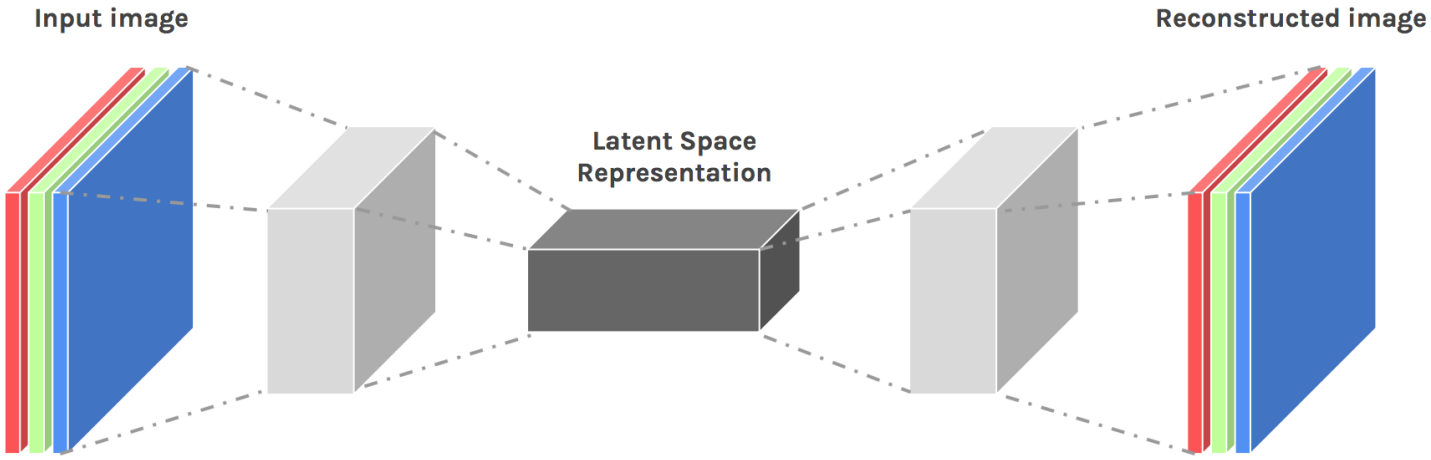


https://mlwhiz.com/blog/2018/09/22/object_detection/

https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/

Image-to-image translation





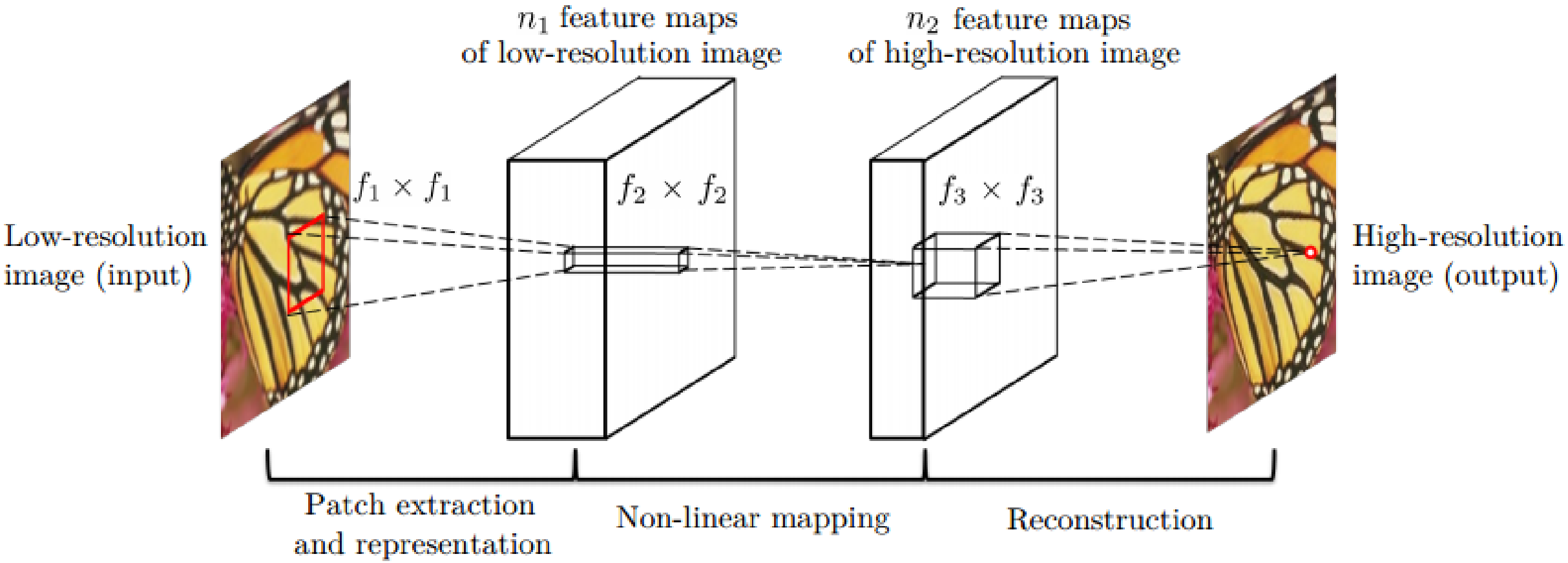
https://phillipi.github.io/pix2pix/

Human Pose Estimation

Fig. 2: Overall pipeline. (a) Our method takes the entire image as the input for a CNN to jointly predict (b) confidence maps for body part detection and (c) PAFs for part association. (d) The parsing step performs a set of bipartite matchings to associate body part candidates. (e) We finally assemble them into full body poses for all people in the image.

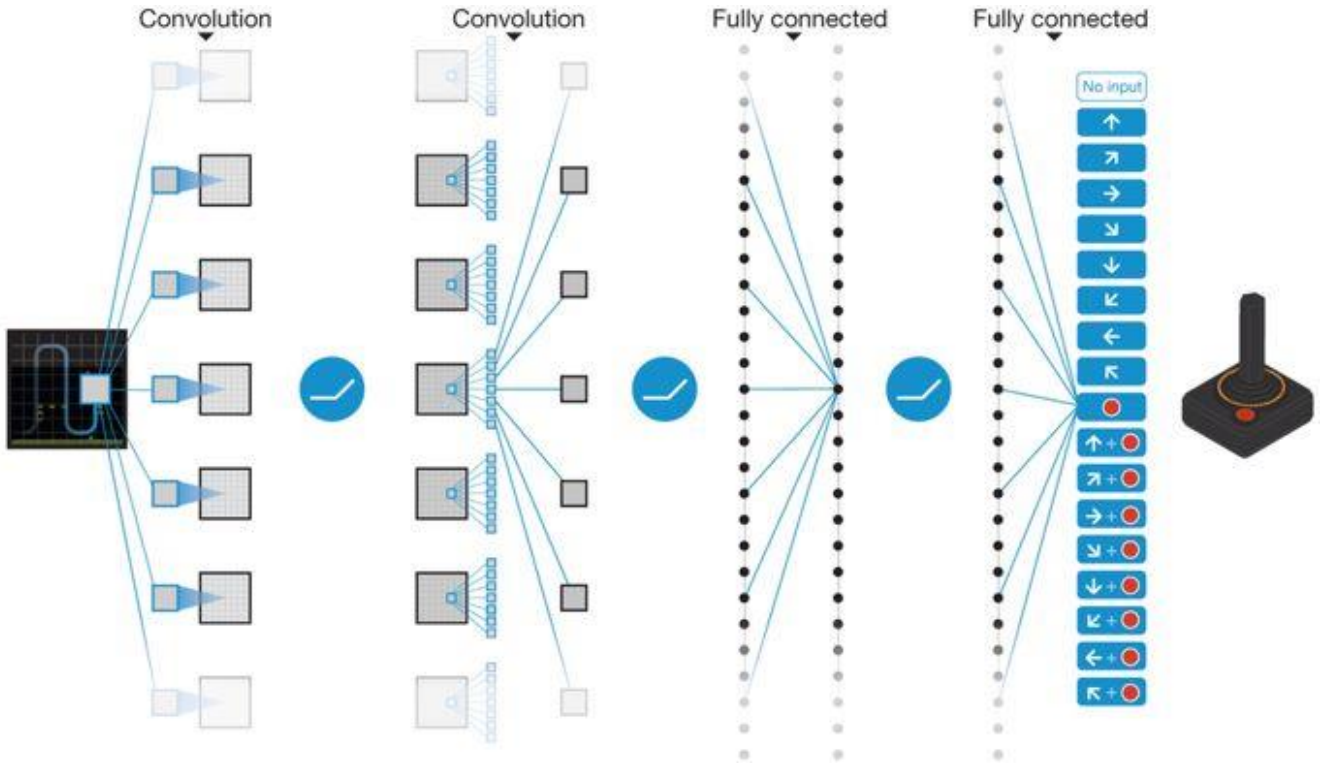
OpenPose: https://arxiv.org/pdf/1812.08008.pdf

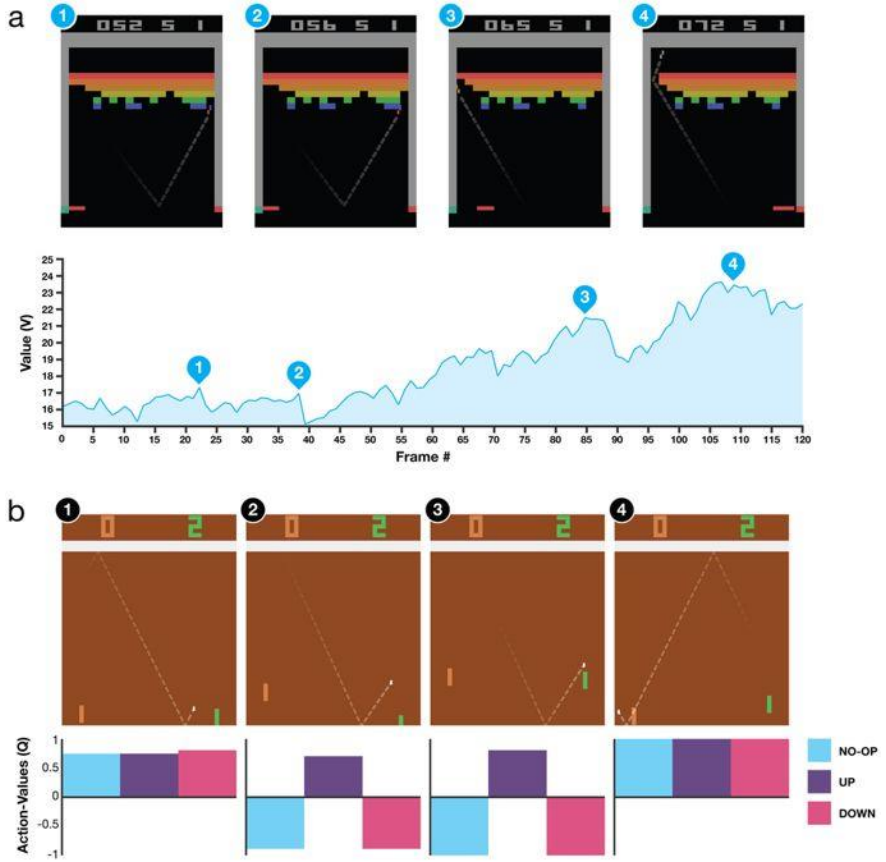
Super Resolution



Play computer games

Deep reinforcement learning – learning to play Atari games





… and much more

Image-to-image transformations

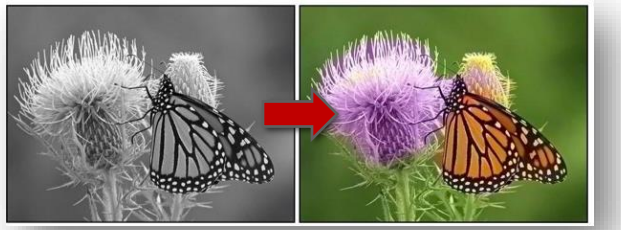
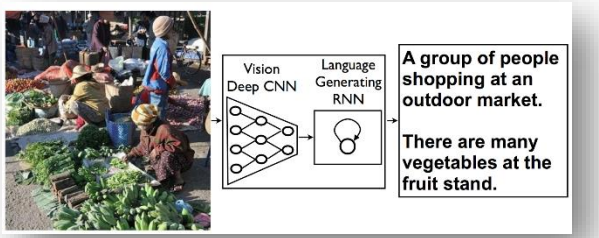
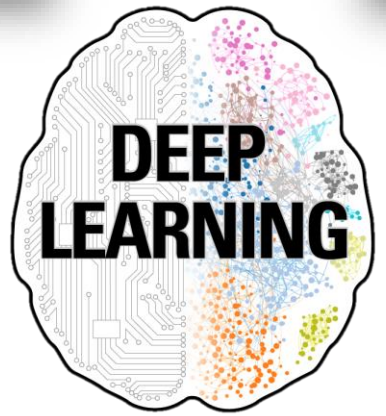


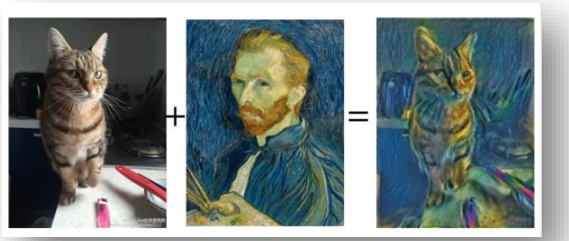
Image captioning



CycleGAN

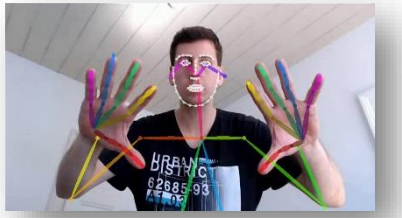


Neural Style Transfer



Generative Adversarial Networks (GANs)

Detect human body pose



Summary

- CNNs are based on **sparse interactions** and **parameter sharing**.
- Though we can get good performance by using CNNs, there are two things we need to notice:
  - Training is time-consuming
  - Prone to overfitting.
- Often, we use pretrained models instead of training from scratch. This is called **Transfer Learning**.

References

- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
- Toshev, Alexander, and Christian Szegedy. "Deeppose: Human pose estimation via deep neural networks." *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*. IEEE, 2014.
- Dong, C., Loy, C. C., He, K., & Tang, X. (2014). Image Super-Resolution Using Deep Convolutional Networks. *arXiv preprint arXiv:1501.00092*.
- Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

Further reading + online videos/demos

- Stanford Deep Learning tutorial (Andrew Ng):
  - http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/
  - http://ufldl.stanford.edu/tutorial/supervised/Pooling/
  - http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/
- Computational graphs and backpropagation in CNNs:
  - https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199
  - https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c
  - http://cs231n.github.io/optimization-2/
  - http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture04.pdf
- AlexNet paper (it's a seminal paper and its not that hard to read):
  - https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf