

Architectures

# Know your data - many types of networks

Fixed length representation



*Images*

Variable length representation



*Online video sequences, or samples of different sizes*

Specific architectures for different types of data

# Fully connected architecture - standard (D)NN

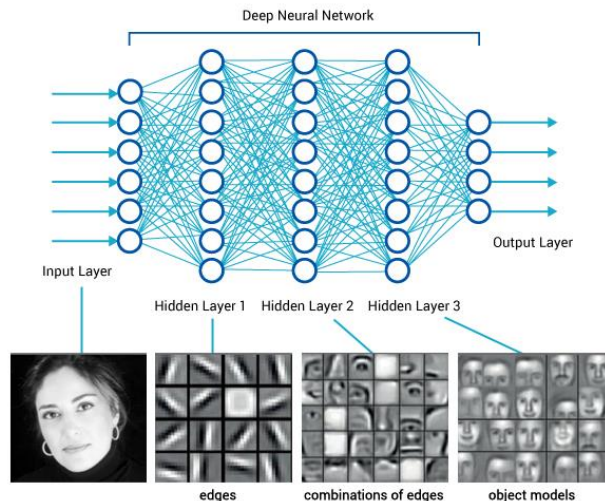
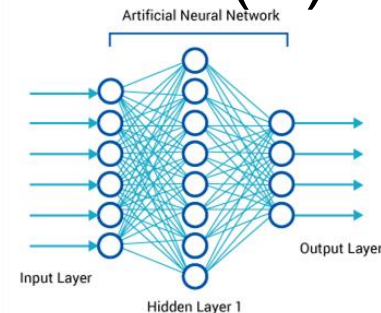
(D)NN receives an input (a single vector), and transform it through a series of hidden layers.

Each hidden layer = a set of neurons

each neuron is fully connected to all neurons in the previous layer,

neurons in a single layer function completely independently, no connections.

the last (fully-connected) layer is called the “output layer”, representing the class scores.



# Sharing the weights - CNN, RNN

GOAL: build a space or time invariant model

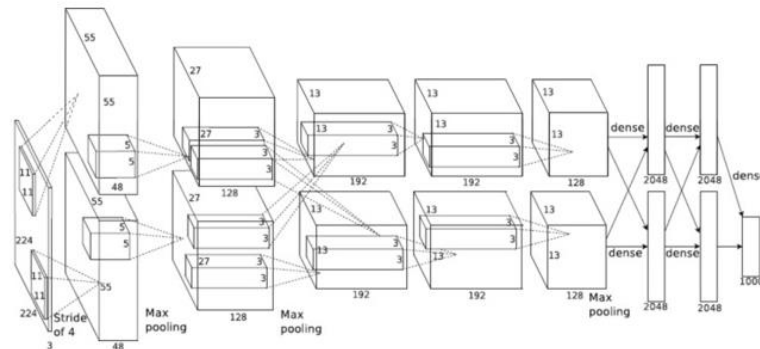
IDEA: use the same set of weights over different parts of the data, by exploiting the known type of the inputs



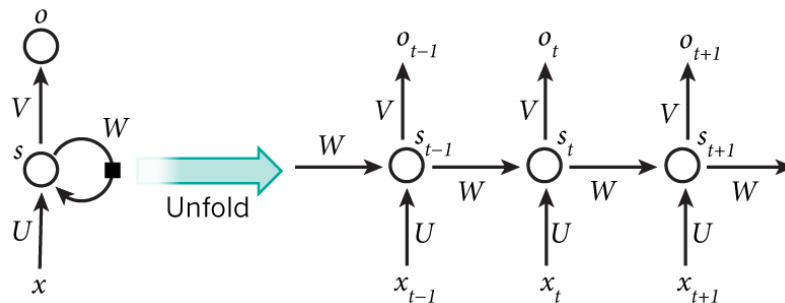
## weight sharing

share and train the weights jointly  
for those inputs that contain  
the same information

## Convolutional Neural Networks (**CNN**) - *images*

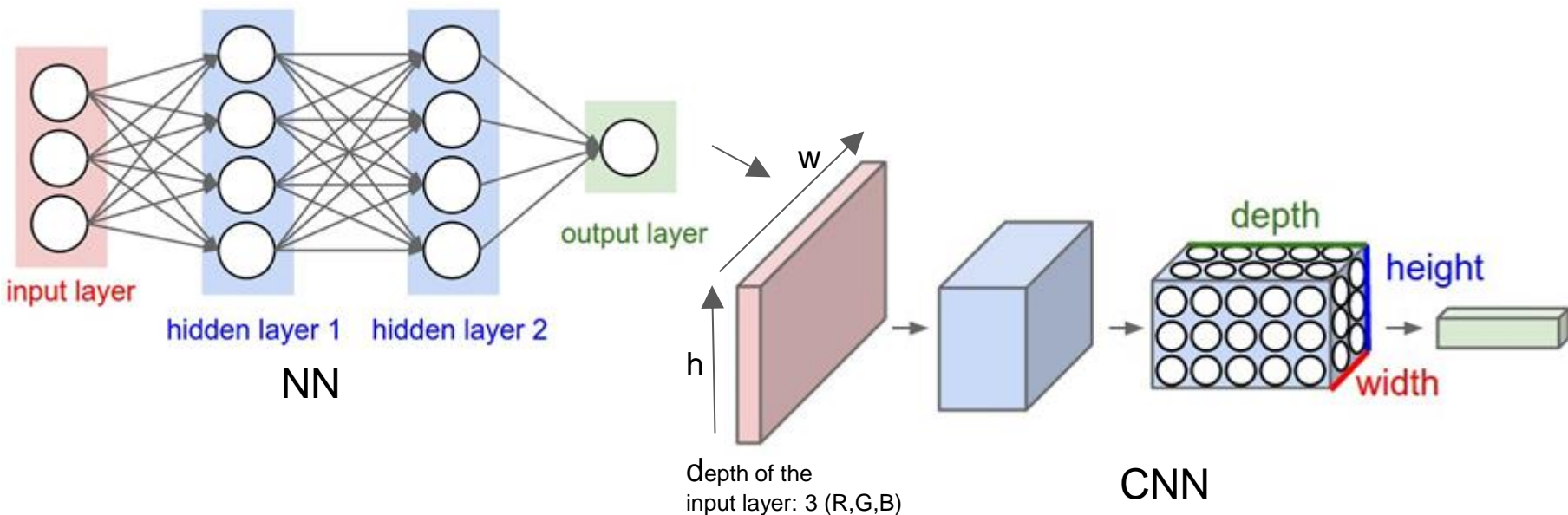


## Recurrent Neural Networks (**RNN**) - *time series*



# CNN: Convolutional Neural Networks or *ConvNets*

- CNN have neurons **arranged in 3 dimensions**: width, height, *depth* (*Depth*  $\neq$  depth of the network)
- A **layer** transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters



# CNN: different types of layers

- A CNN layer may be of 5 different kinds, 2 of them are CNN specific (in **blank**)
- Input Layer, **Convolutional Layer**, RELU Layer, **Pooling Layer**, and Fully-Connected Layer
- We explain the layer on a simple architecture over the CIFAR-10

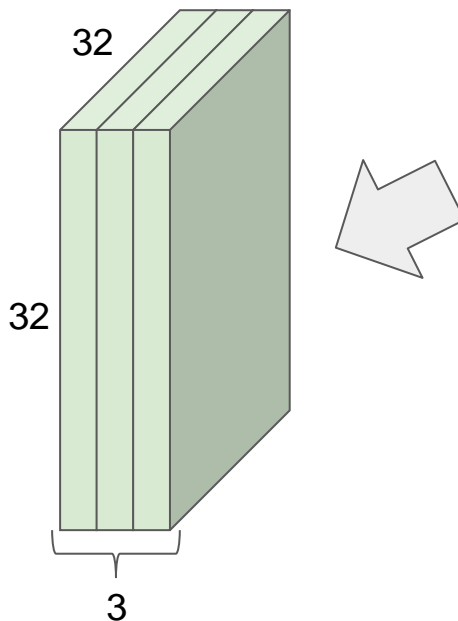
## CIFAR-10 (2010, Hinton/Krizhevsky)

- 10 classes
- 32x32 color image
- 6,000 images per class
- 5,000 training / 1,000 test images per class



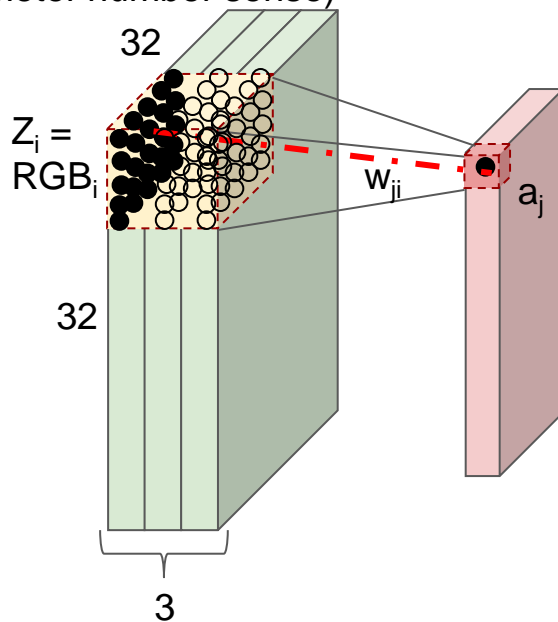
# CNN: Input layer

- **WHERE:** this is the very bottom of the network
- **WHAT DOES IT SERVE:** To feed the network
- **HOW IT IS STRUCTURED:** The input layer takes as input the CCD of the sensors. It has stacked in depth the R,G,B channels of the network
  - In the CIFAR example, it results in a 32x32x3 sized layer of integer numbers



# CNN: Convolutional layer (CONV)

- The reason why we are talking about Cnn
- **WHERE**: the early layers of the CNN are CONV
- **WHAT DOES IT SERVE**: to highlight low-level - poorly semantic - highly perceptually salient patterns in an economic way (in a parameter number sense)
- **HOW IT IS STRUCTURED**: A CONV layer's parameters consist of a set of NEURONS = **learnable filters**
  - *Let's see an example of neuron that perform convolution*
  - The red dash-dotted line is just one of the  $25 \times 3 = 75$  dendrites that arrive to one neuron of the CONV layer
  - Each dendrite has a weight
  - Convolution is nothing else that:



$$a_j = \sum_i^{75} w_{ji} z_i$$

# CNN: Convolutional layer (CONV)

- A CONV layer needs *parameters*:
  - input volume size ( $W$ ),
  - the receptive field size of the Conv Layer neurons ( $F$ ),
  - the stride with which they are applied ( $S$ )
  - the amount of zero padding used ( $P$ ) on the border.

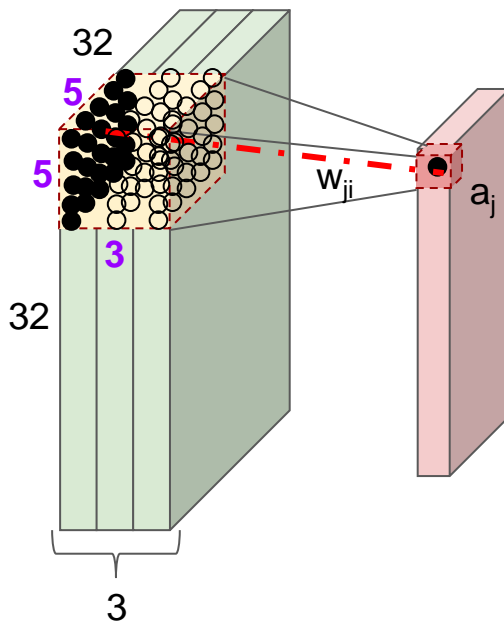
- In this case

- $W = (32 \times 32 \times 3)$
- $F = (5 \times 5 \times 3)$
- $S =$  not visible in the figure, could be 1, 2, (rare) 3
- $P = 0$

- How many neurons? In the case of 1D structure  $\rightarrow (W - F + 2P) / S + 1$

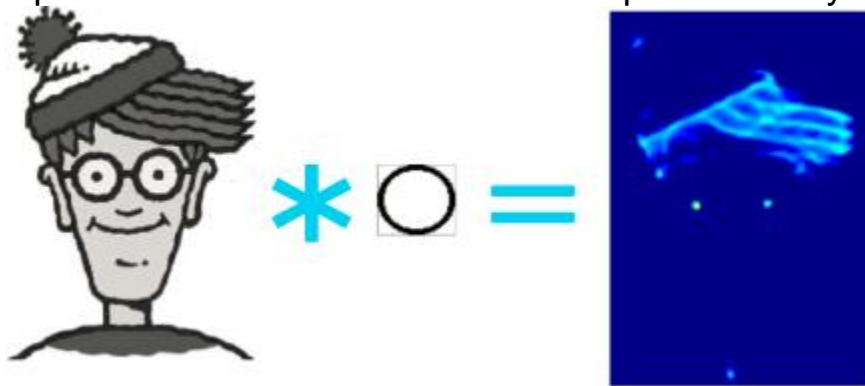
- How many parameters to tune, so far ( $S=1$ )?  
 $\quad \quad \quad F \quad \quad \quad \text{bias}$

$$[(32-4) \times (32-4)] \times [(5 \times 5 \times 3) + 1] = 59584 \rightarrow \text{TOO MANY!!!}$$



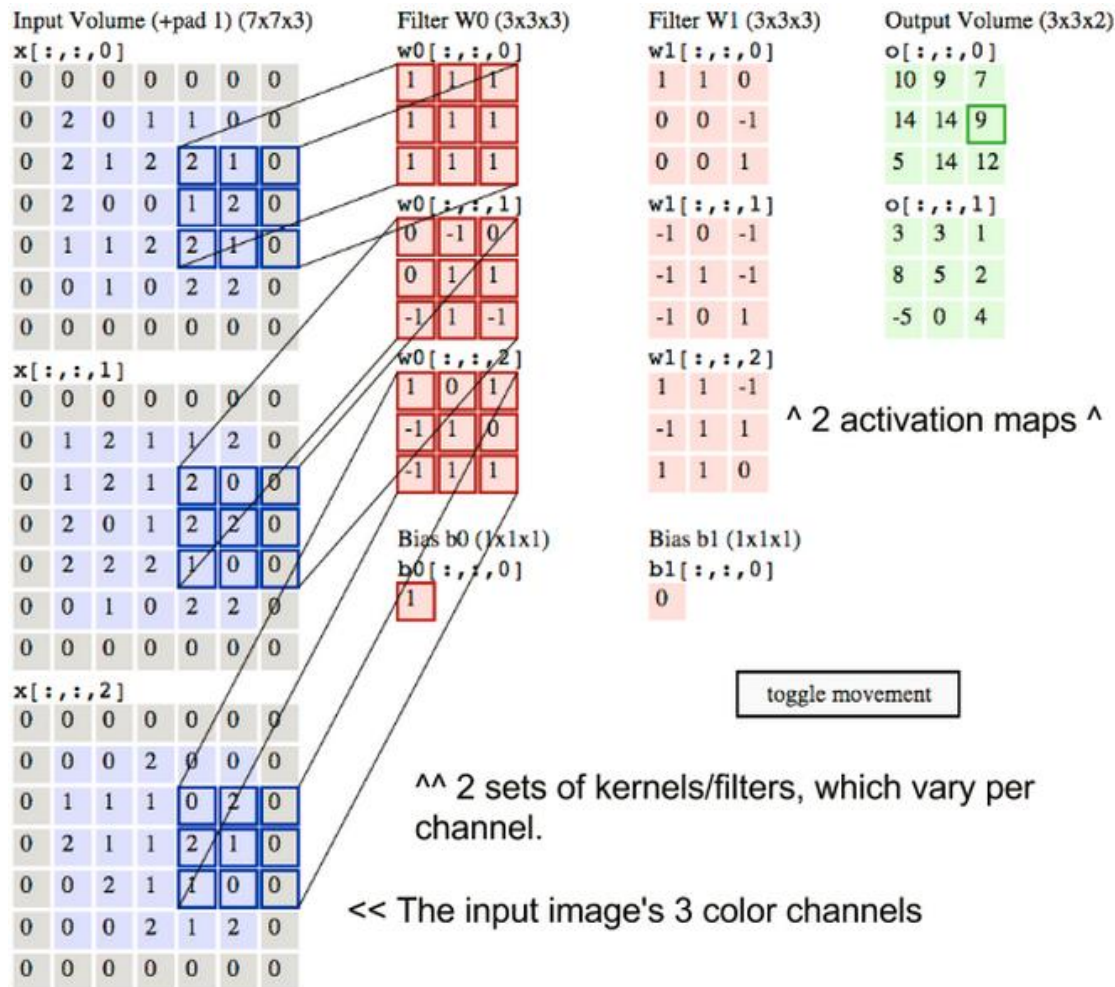
# CNN: Convolutional layer (CONV)

- The great idea is that **all the weights across one layer are shared**
- In practice, the output from a CONV slice is like it was processed by an image filter



- The good is, **that filter has been learned from data!**
- The trick is, back propagation over the entire layer is computed, but at the end all the backpropagated errors are summed together and the weights of the filter are updated just once

$$\cancel{[(32-4) \times (32-4)]} \times \overset{F}{(5 \times 5 \times 3)} + \overset{\text{bias}}{1} = 76 \rightarrow \overset{3}{\text{OK!!!}}$$



1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

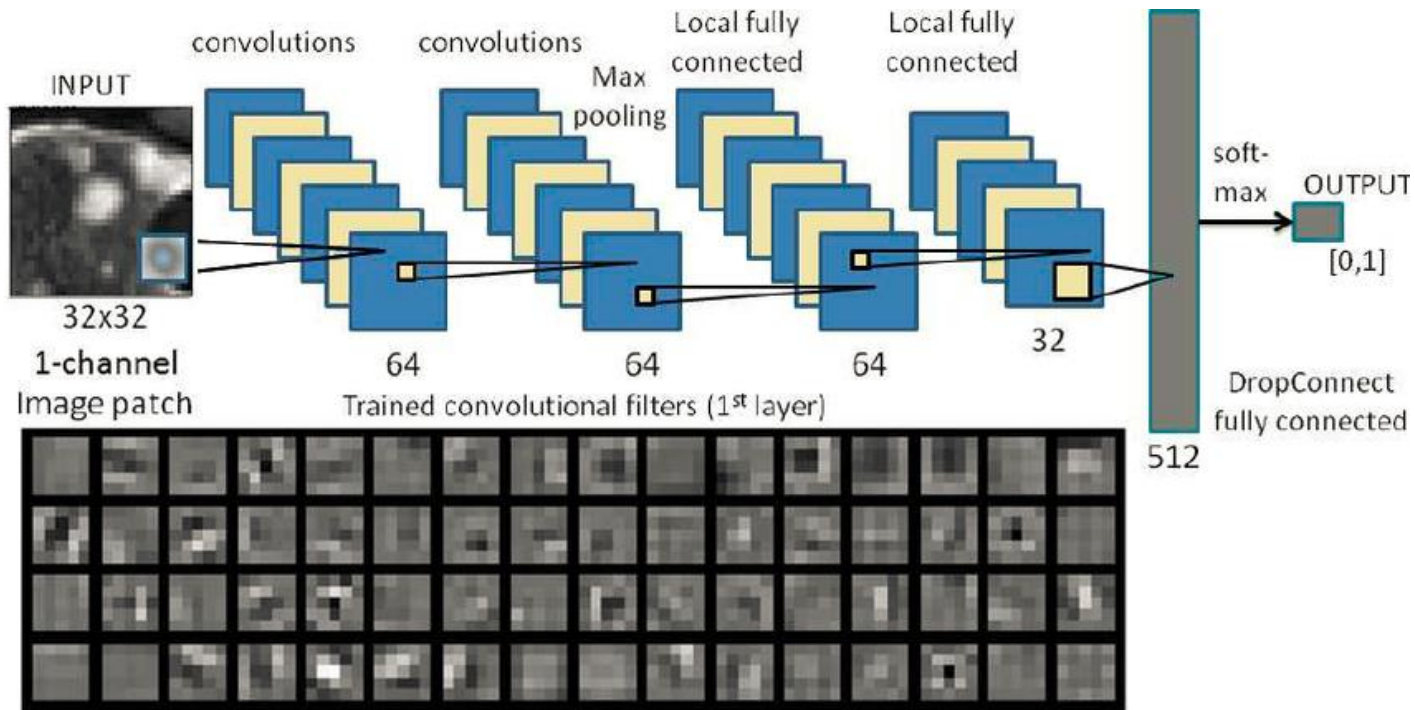
Convolved  
Feature

$\text{patch.size} < \text{image.size} \Rightarrow$   
**fewer parameters**

**Translation invariance:** it's not important where the object is since the kernel of the filter is the same for each patch of the input image

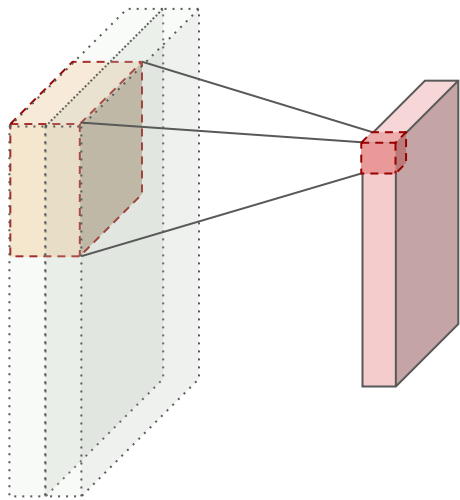
# CNN: Convolutional layer (CONV)

- Due to the parameters sharing, many filters can be instantiated (in the figure, 64 at L1)

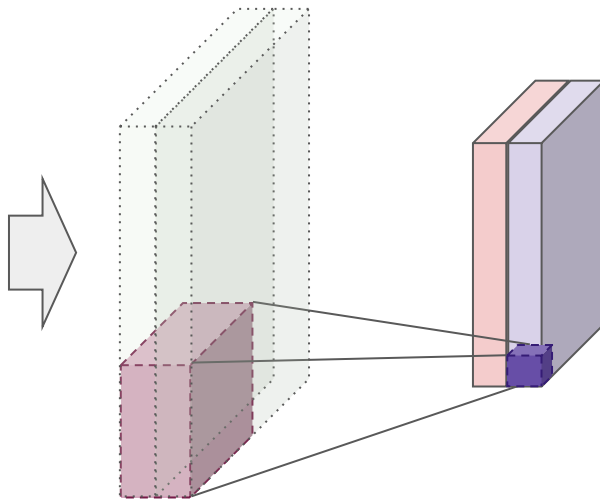


# CNN

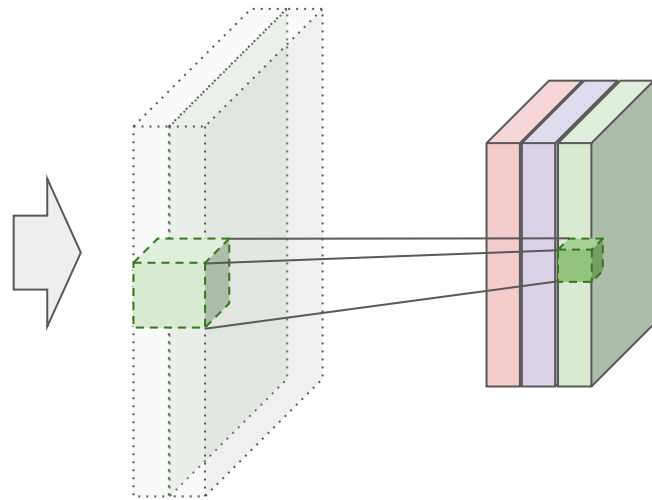
In a specific layer, we want to apply  $K = 3$  filters...



Apply the 1<sup>st</sup> filter to the whole image and generate the first feature map...

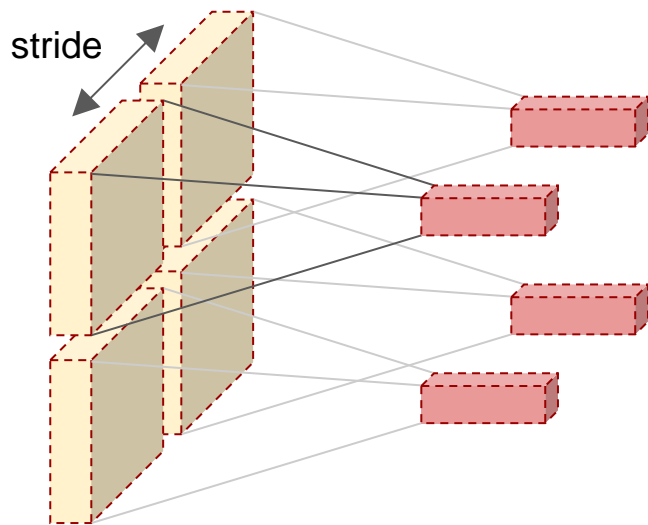


... then, compute the convolution also for the 2<sup>nd</sup> filter to generate another feature map...



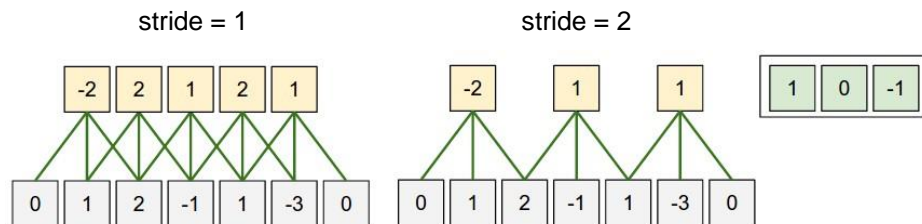
... and so the 3<sup>rd</sup> filter!

# CNN: stride



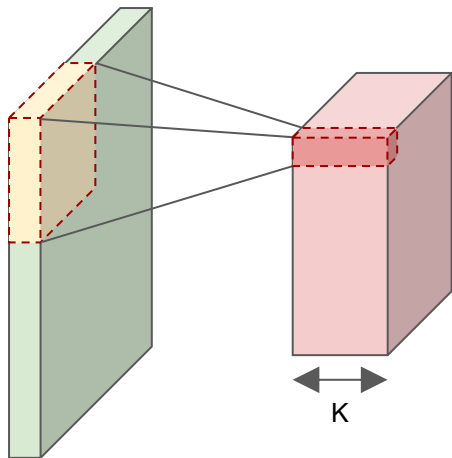
**stride:** pixel interval used to move the conv. filter

↓ stride > 1  
subsampling



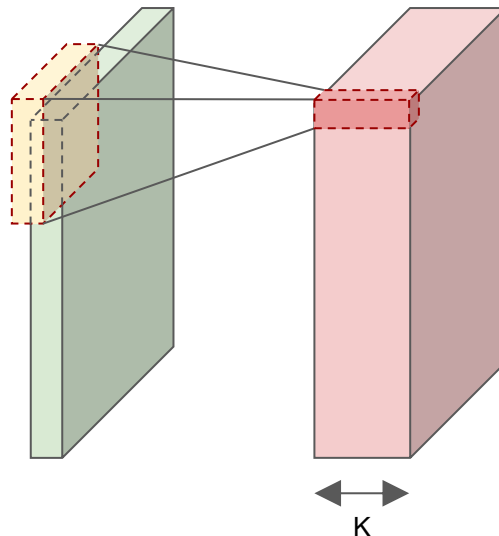
# CNN: padding

**valid padding**



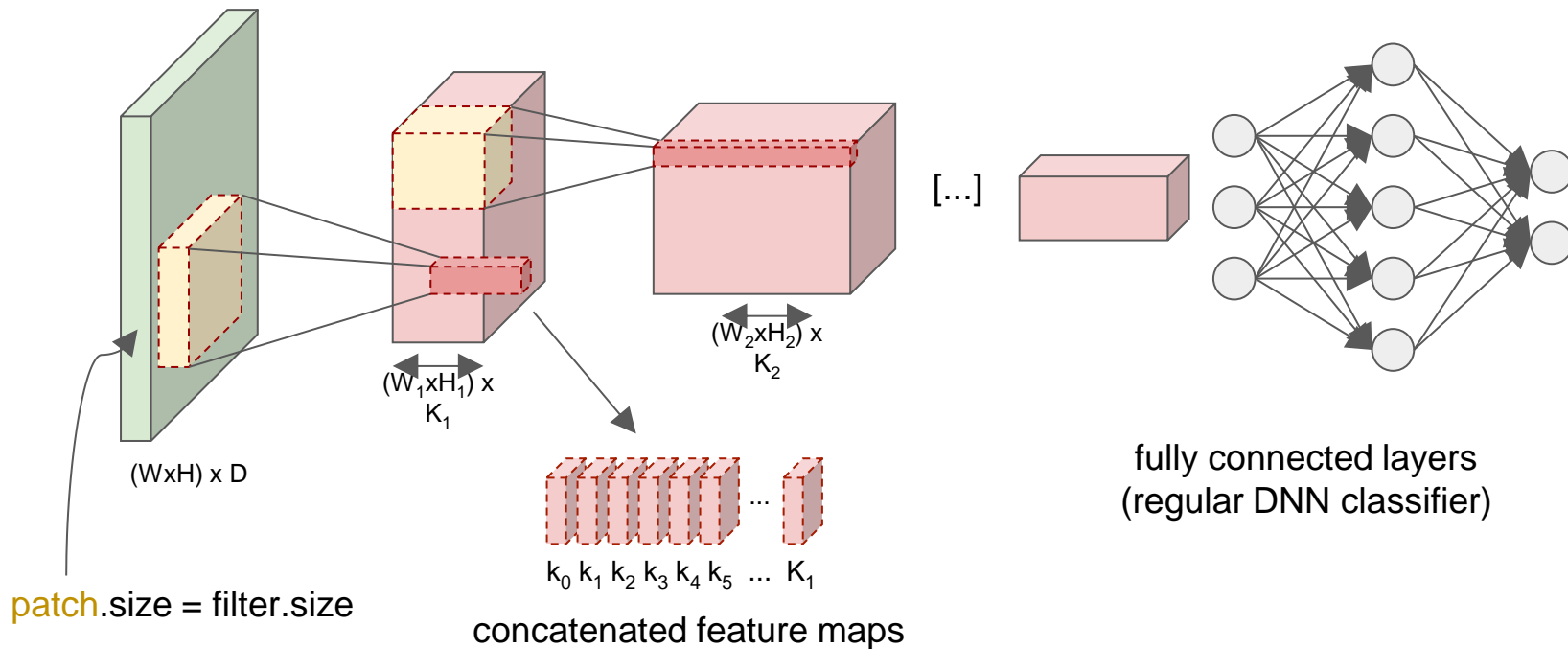
reduces data according  
to the filter shape

**same padding**



maintains data original  
data size but  
needs zero-padding

# CNN: convolutional (depth) pyramid

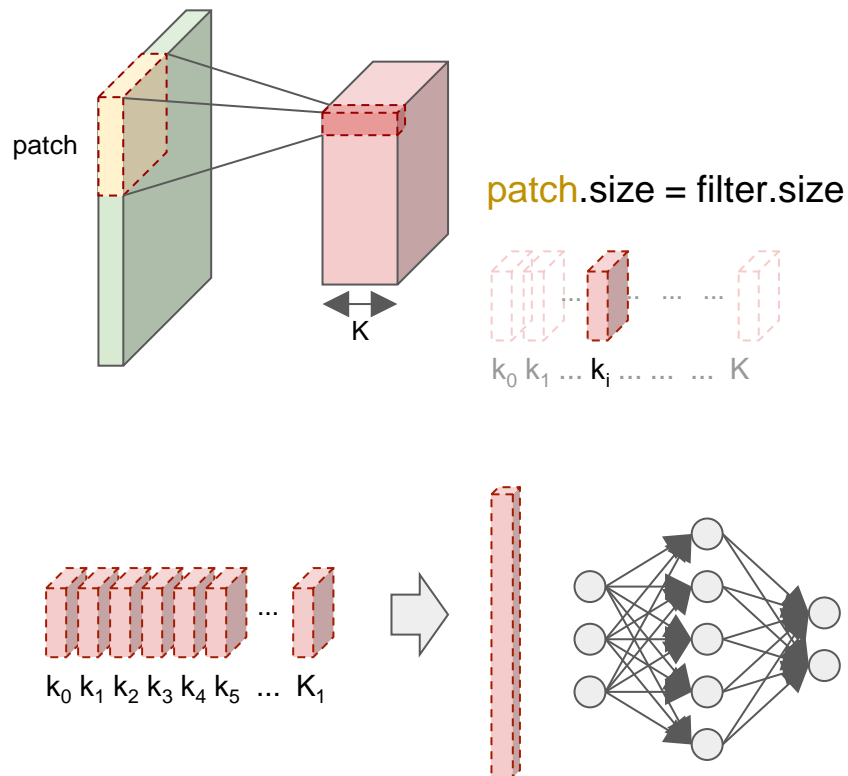


$D$  = input depth, number of feature maps or channels;  $W$  = width;  $H$  = height;  $K_i$  = number of filters of the  $i$ -th level of size  $(W_i \times H_i)$

# CNN --verbose

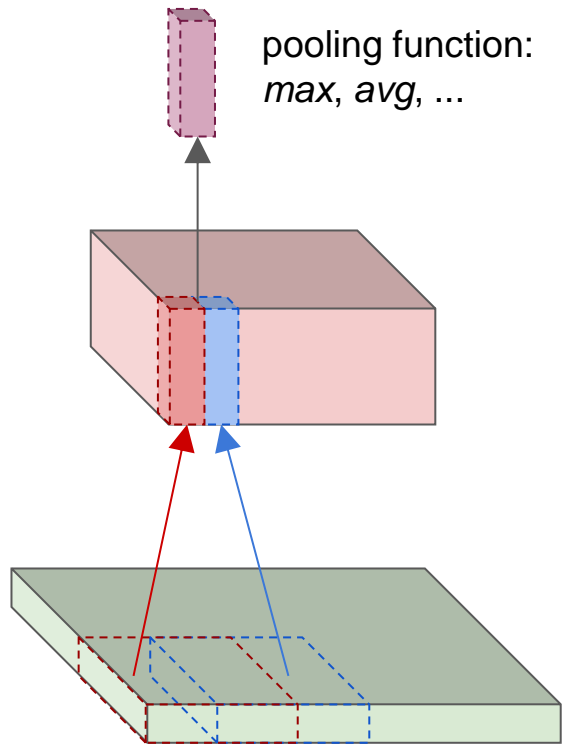
## Algorithm idea:

- set the size of the  $K$  filters ( $F_w, F_h$ ) to train for each convolutional layer, the stride value and the padding type
- the  $K$ -th convolution generates a feature map of size  $(F_w, F_h, 1)$
- concatenate the  $K$ -th feature maps in a  $(F_w, F_h, K)$  matrix
- train a regular fully connected (DNN + classifier) on the last stack of feature maps



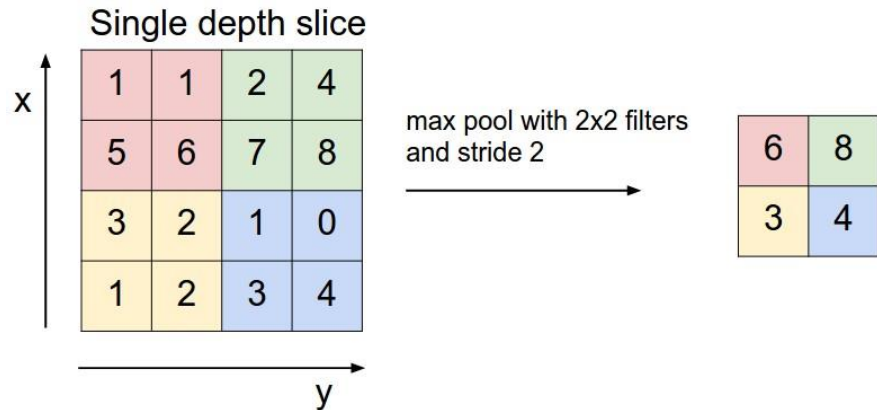
# CNN: Pooling layer

- **WHERE:** It is common to periodically insert a Pooling layer in-between successive Conv layers
- **WHAT DOES IT SERVE:** it reduces
  - the spatial size of the representation
  - the amount of parameters and computation, controlling the overfitting.
- **HOW IT IS STRUCTURED:**
  - it operates independently on every depth slice of the input and resizes it spatially, using the MAX (AVERAGE/L2NORM) operation
  - The depth dimension remains unchanged
  - It is efficient as for the backpropagation updates (the max operation in the forward says where to push down the gradient in the backward pass)
  - *its usefulness is not universally accepted*



# CNN: Pooling layer

- Accepts a volume of size  $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - the spatial extent  $F$
  - the stride  $S$
- Produces a volume of size  $W_2 \times H_2 \times D_2$ , where:
  - $W_2 = (W_1 - F) / S + 1$
  - $H_2 = (H_1 - F) / S + 1$
  - $D_2 = D_1$
- Introduces zero parameters, since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers
- In general we have a pooling layer with  $F=3, S=2$  (also called *overlapping pooling*), and more commonly  $F=2, S=2$ . Pooling sizes with larger receptive fields *are too destructive*.



# CNN: Common architectures

- The most common form of a ConvNet architecture stacks
  - a few CONV-RELU layers,
  - follows them with POOL layers, and
  - repeats this pattern until the image has been merged spatially to a small size
  - At some point, it is common to transition to fully-connected layers
  - The last fully-connected layer holds the output, such as the class scores

INPUT  $\rightarrow$   $[[\text{CONV} \rightarrow \text{RELU}]^N \rightarrow \text{POOL?}]^M \rightarrow [\text{FC} \rightarrow \text{RELU}]^K \rightarrow \text{FC}$

- where the  $*$  indicates repetition, and the POOL? indicates an optional pooling layer
- Moreover,  $N \geq 0$  (and usually  $N \leq 3$ ),  $M \geq 0$ ,  $K \geq 0$  (and usually  $K < 3$ ).

# CNN: State of the art architectures

- **LeNet.** The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Used to read zip codes, digits, etc.
- **AlexNet.** Developed by Alex Krizhevsky, et al. was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- **ZF Net.** The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.

# CNN: State of the art architectures

- **GoogLeNet.** The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.
- **VGGNet.** Runner-up in ILSVRC 2014, its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Downside, is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer
- **ResNet.** Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special skip connections and a heavy use of *batch normalization*. The architecture is also missing fully connected layers at the end of the network.

# Visualizing CNNs

(what's going on under the hood)

# Visualization techniques

- Several approaches for understanding and visualizing Convolutional Networks have been developed in the literature
- They answer the common criticism that the learned features in a Neural Network *are not interpretable*
- Techniques:

*Visualizing the activations and first-layer weights*

*Retrieving images that maximally activate a neuron*

*Embedding the codes with t-SNE*

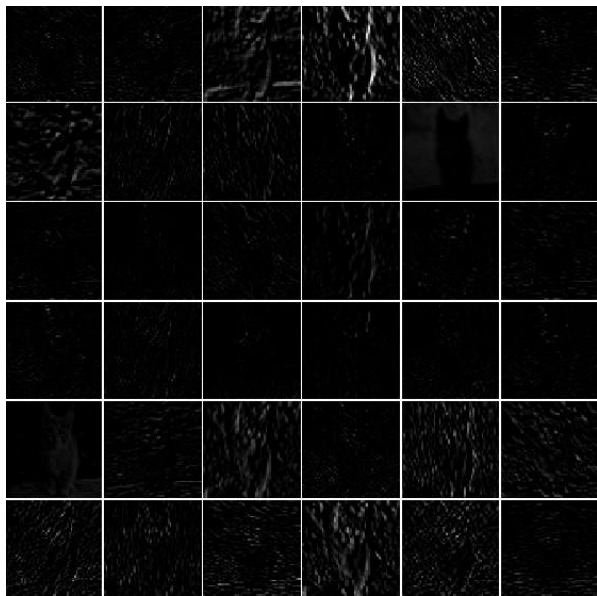
*Occluding parts of the image*

Others at <http://cs231n.github.io/understanding-cnn/>

# Visualizing the **activations** and first-layer weights

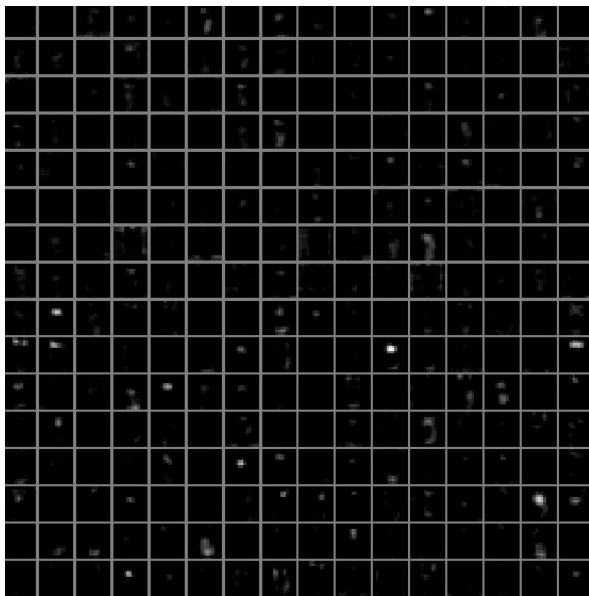
+

- Interpretable, if at the first layers



-

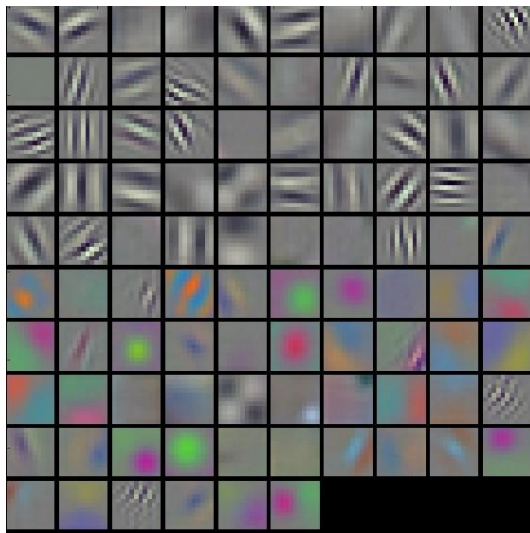
- no meaning in the deep layers
- Low activations or dead neurons?



# Visualizing the activations and **first-layer weights**

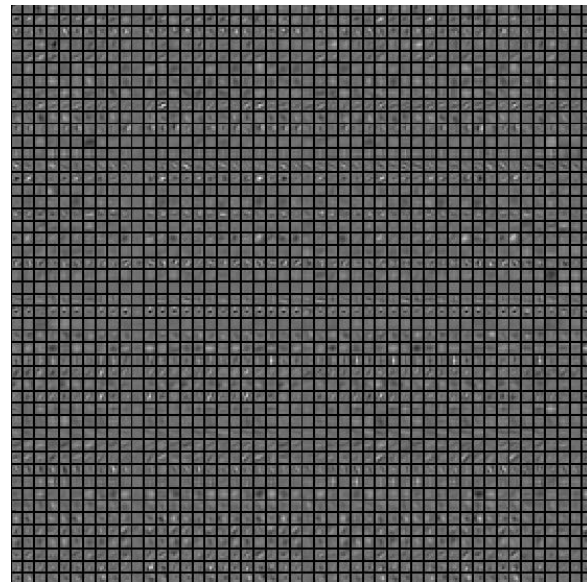
+

- **More** interpretable, if at the first layers
- Noisy patterns can be an indicator of a network that hasn't been trained for long enough, or possibly a very low regularization strength that may have led to overfitting.



-

- few meaning in the deep layers



# Retrieving images that maximally activate a neuron

+

- Gives meaning
- Serves for object detection

-

- Not well suited for RELU layers (no meaning associated to them)



# Embedding the codes with t-SNE

+

- Gives an idea of the CNN space

-

- Highly dependent on the type of embedding



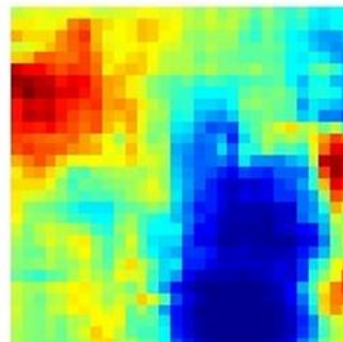
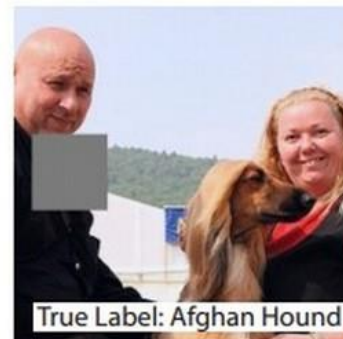
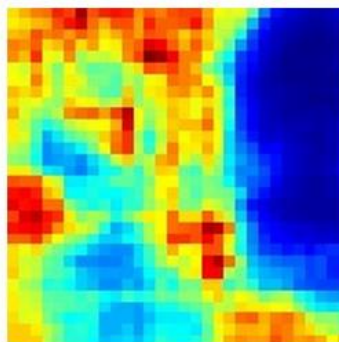
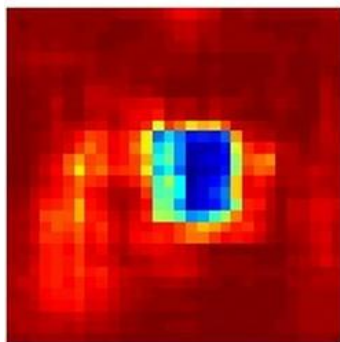
# Occluding parts of the image

+

- Useful for evaluating an entire network

-

- It is a discriminative, not generative way to understand things



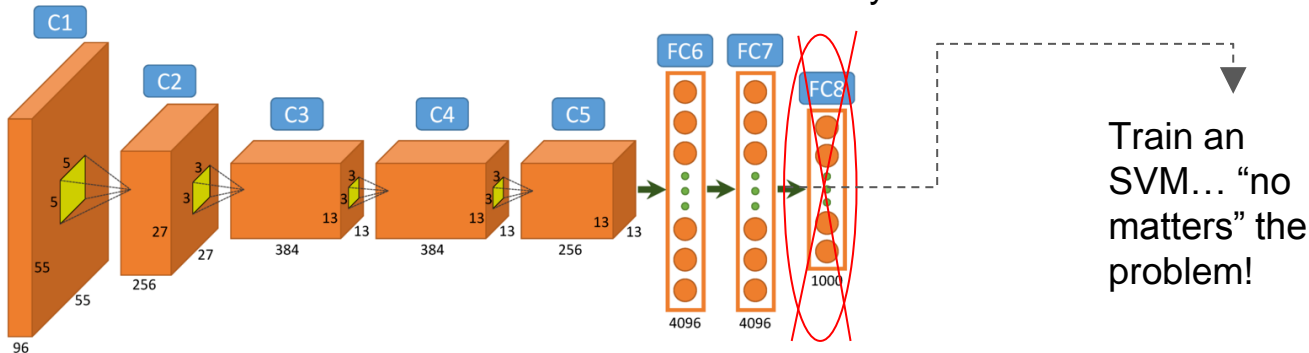
# Transfer Learning

# The transfer learning and its strategies - CNN codes

- Very few people train from scratch (with random initialization) a CNN (no data, time - weeks!)
- Instead, it is common to (*let others*) pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then:

## 1. ConvNet as fixed feature extractor:

- a. Take a ConvNet pretrained on ImageNet,
- b. remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet)
- c. treat the rest of the ConvNet as a fixed feature extractor for the new dataset.
  - In an AlexNet, this would compute a 4096-D vector (Relud IT!!!) for every image that contains the activations of the hidden layer immediately before the classifier.



# Transfer learning - Fine tuning

## 1. Fine tuning:

- a. Start with an initialization already computed by backpropagation
- b. Do backpropagation on the layers you want
  - Usually, only the last layers are trained, the earlier are more generic and are preferred to be left unchanged
- c. In particular, four scenarios are available
  - *New dataset is small and similar to original dataset*(**NO FINE TUNING**). Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.
  - *New dataset is large and similar to the original dataset*. Since we have more data, we can have more confidence that we won't overfit if we were to try to fine-tune through the full network.

# Transfer learning - Fine tuning (2)

- *New dataset is small but very different from the original dataset.* Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier from the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.
- *New dataset is large and very different from the original dataset.* Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pretrained model. In this case, we would have enough data and confidence to fine-tune through the entire network.

More on <http://cs231n.github.io/transfer-learning/>