Lecture 15: Deep Learning

Applied Machine Learning

Volodymyr Kuleshov, Jin Sun
Cornell Tech

Part 1: What is Deep Learning?

Deep learning is a relatively new and powerful subfield of machine learning closely tied to neural networks.

Let's find out what deep learning is, and then we will see some deep learning algorithms.

Review: Neural Network Layers

A neural network layer is a model $f : \mathbb{R}^d \to \mathbb{R}^p$ that applies p neurons in parallel to an input x.

$$f(x) = \sigma(W \cdot x) = \begin{bmatrix} \sigma(w_1^\top x) \\ \sigma(w_2^\top x) \\ \vdots \\ \sigma(w_p^\top x) \end{bmatrix},$$

where each w_k is the vector of weights for the k-th neuron and $W_{kj} = (w_k)_j$. We refer to p as the *size* of the layer.

Review: Neural Networks

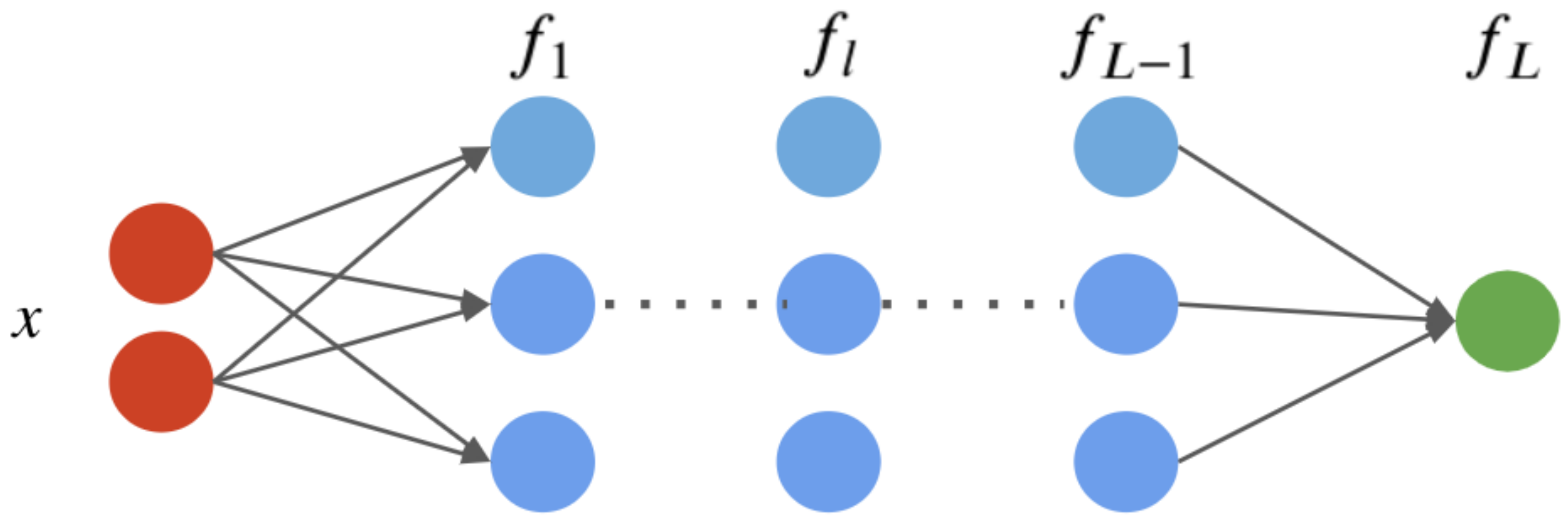
A neural network is a model $f : \mathbb{R} \to \mathbb{R}$ that consists of a composition of L neural network layers:

$$f(x) = f_L \circ f_{L-1} \circ \dots f_1(x).$$

The final layer f_L has size one (assuming the neural net has one ouput); intermediary layers f_l can have any number of neurons.

The notation $f \circ g(x)$ denotes the composition $f(g(x))$ of functions

We can visualize this graphically as follows.



What is Deep Learning?

In a nutshell, deep learning is a modern evolution of the field of artificial neural networks that emphasizes:

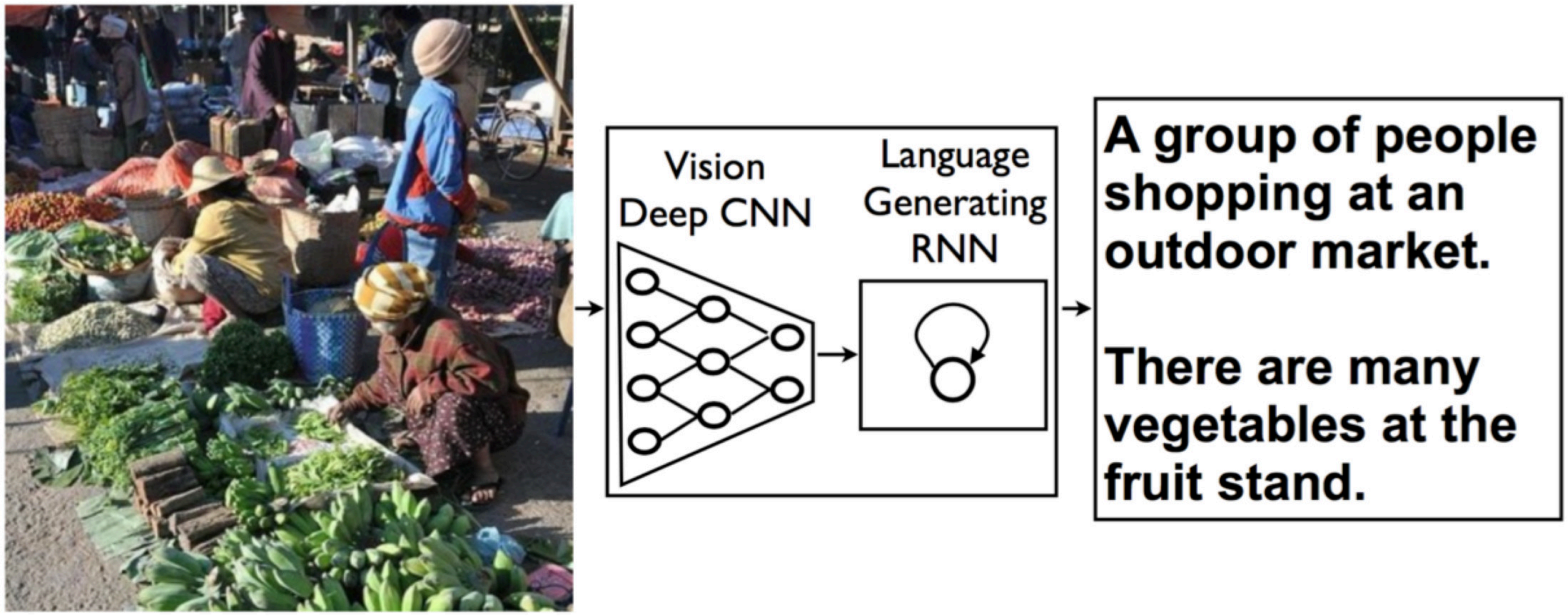
- Very deep neural networks
- Large unstructured datasets, especially images, text, and audio
- Modern computational resources, like GPUs

Expressivity of Deep Models

Why is deep learning powerful? One reason is deep neural networks can represent complex models very compactly.

- Shallow neural networks can represent any function, but need very large hidden layers.
- Deep networks can represent very complex $\mathcal{X} \rightarrow \mathcal{Y}$ mappings with fewer parameters.

In practice, deep neural networks can learn very complex mappings such as $\text{image} \rightarrow \text{text description}$ that other algorithms cannot.



A group of people shopping at an outdoor market.

There are many vegetables at the fruit stand.

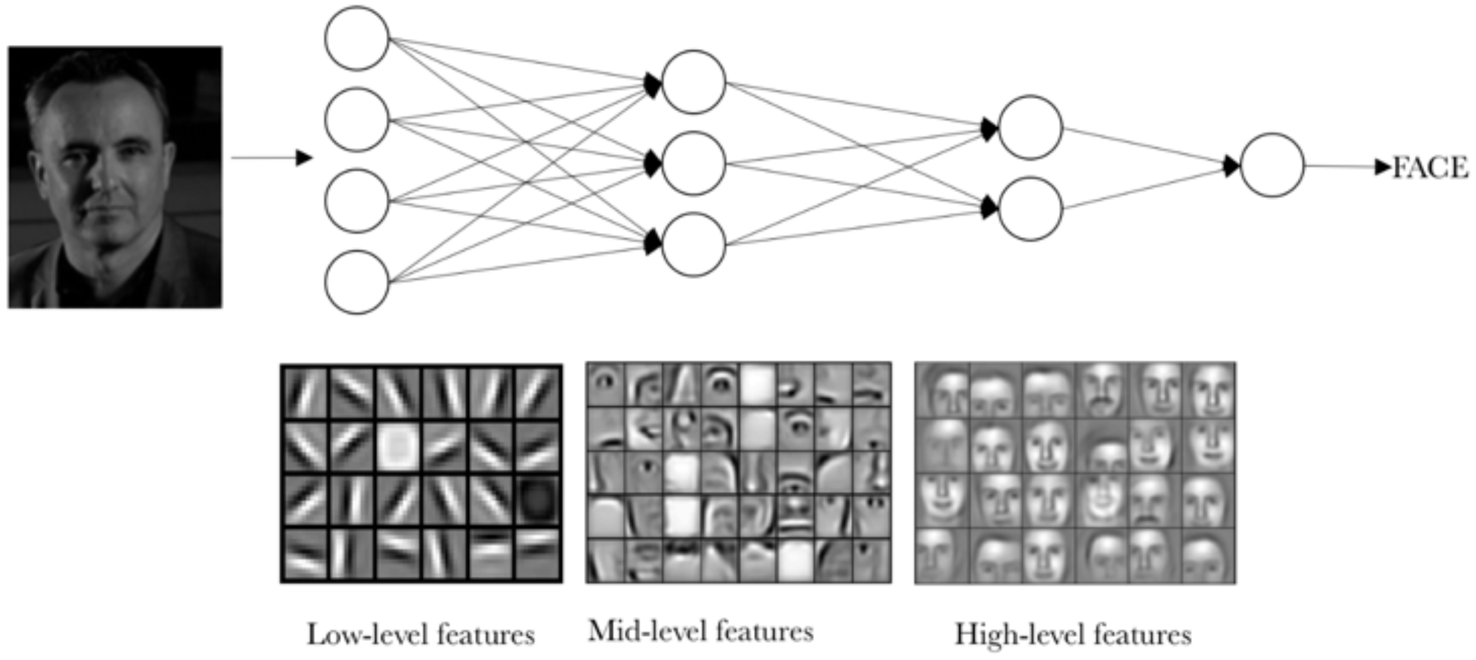
Ref: Learning CNN-LSTM Architectures for Image Caption Generation; https://cs224d.stanford.edu/reports/msoh.pdf

Representation Learning

How does a deep neural network use its representational power?

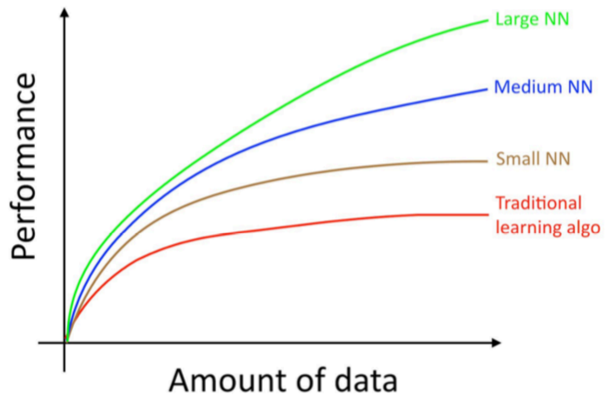
- The sequential layers can learn to represent data at an increasing level of abstraction
- This can also be interpreted as learning: each layer maps input to a more abstract feature space that is *learned* from data.

This is an example of representations that a deep neural network learns from data.



Scaling to Large Datasets

Deep learning models also scale to very large datasets.



Classical algorithms like linear regression saturate after a certain dataset size. Deep learning models keep improving as we add more data.

Computational Scalability

Deep learning datasets benefit from large datasets because they can easily use specialzied computational hardware.

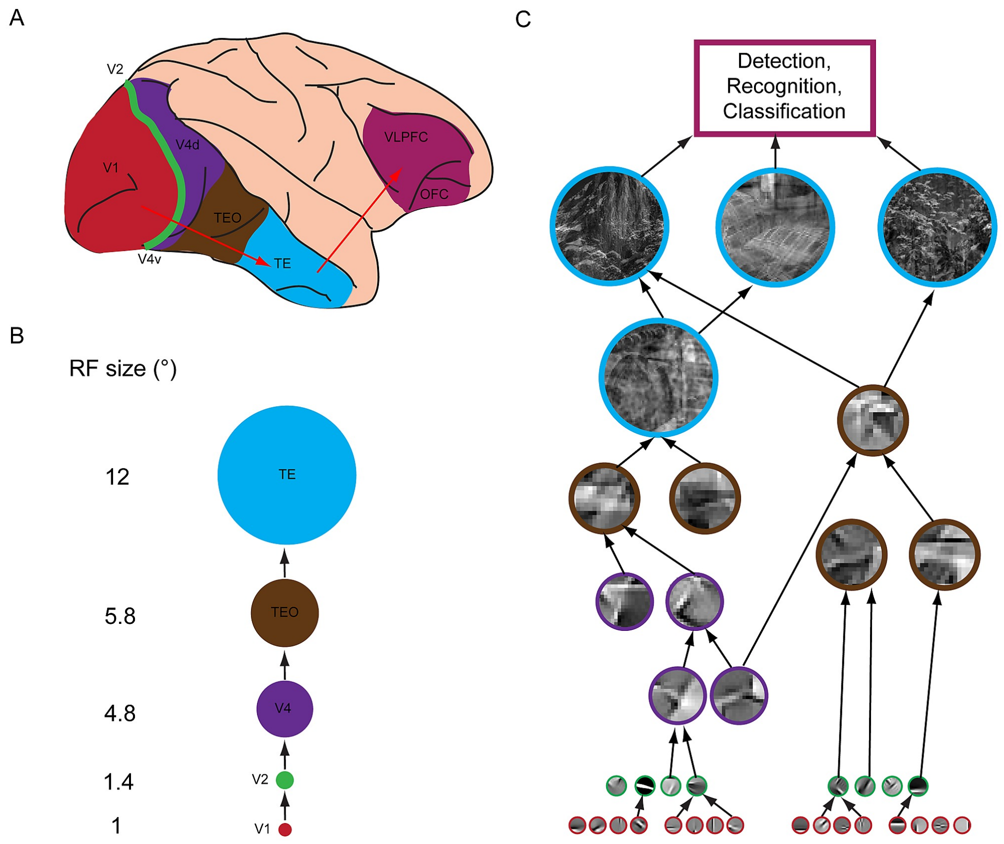
- Neural networks mostly implement linear algebra, and benefit from hardware acceleration.
- Today, they are trained on graphical processing units (GPUs), or even more specialzied hardware, resulting in order of magnitude speedups.

Connections to Neuroscience

As artificial neurons are designed to mimic the biological ones, deep neural networks have strong connections with biological neural systems.

One of the most interesting ones is the visual processing network in brains.

There is ample evidence that deep neural networks perform computation very simiilar to that of the visual cortex (Kravitz el al (https://plos.figshare.com/articles/_Hierarchical_representations_along_the_ventral_visual_



Successes of Deep Learning

Deep learning has been a real breakthrough in machine learning over the last decade.

- It has dramatically improved common technologies like speech recognition or machine translation.
- It has enabled new results that have captured the popular imagination.

One really recent impressive application is image generation.

These faces are not real! See: https://thispersondoesnotexist.com/ (https://thispersondoesnotexist.com/)

Pros and Cons of Deep Neural Networks

Deep neural nets (DNNs) are among the most powerful ML models.

- DNNs set a new state-of-the-art in many application areas.
- They can be combined with many algorithms (supervised, unsupervised), and significantly improve them.
- Many specialized software and hardware platforms for DNNs.

However, DNNs also can be slow and hard to train and require a lot of data.

Part 2: Convolutions and Pooling

These basic operations are the building blocks of modern convolutional neural networks.

Review: What is Deep Learning?

In a nutshell, deep learning is a modern evolution of the field of artificial neural networks that emphasizes:

- Very deep neural networks
- Large unstructured datasets, especially images, text, and audio
- Modern computational resources, like GPUs

Definition of Convolution

Let $f \in \mathbb{R}^n$ and $g \in \mathbb{R}^m$ be two vectors, called the *filter* and the *signal* respectively. Typically, $n < m$.

In deep learning, a convolution $(f * g) : \mathbb{Z} \to \mathbb{R}$ is typically defined as

$$(f * g)[p] \triangleq \underbrace{\sum_{t=1}^{n} f[t]g[p+t]}_{\text{dot product of } f \text{ with part of } g}$$

where $g[t] = 0$ when $t \leq 0$ or $t > m$.

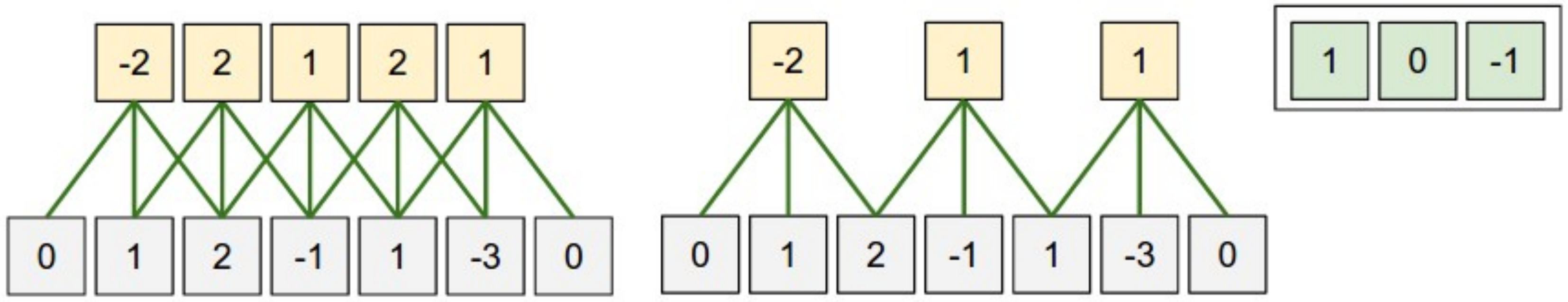
- We may think of the convolution

  $$\sum_{t=1}^{n} f[t]g[p+t]$$

  as a "local" dot product of `f` with `g[p:p+n]` (the subsequence of g that starts at position p).

- $(f * g)[p]$ is large when f is similar to g at p and small otherwise. It measures the extent to which f is "found" in g.

This is best understood via a picture:



The green sequence `[1,0,-1]` is the filter. The signal g is in gray, and the outputs of the convolution

$$(f * g)[p] = 1 \cdot g_p + 0 \cdot g_{p+1} - 1 \cdot g_{p+2}$$

are in yellow.

On a small technical note, what we have defined is called the cross-correlation in mathematics.

The convolution is technically defined as $\sum_{t=1}^{n} f[t]g[p-t]$, but in deep learning both formulas effectively give the same results and the cross-correlation is used in practice (and is called "convolution").

We can implement a convolution in Python as follows.

```
In [1]: # A naive (*slow*) implementation of convolution with loops, don't use this in p
        ractice!
        def conv_naive(f, g):
            M = len(f)
            N = len(g)
            result = [0]*N
            for n in range(N):
                for m in range(M):
                    if n-m>=0 and n-m<N:
                        result[n] += f[m]*g[n-m]
            return result[M-1:] #only return valid results

        # numpy also provides convolution function np.convolve(f,g,'valid')
```

Example: Edge Detection

To gain more intuition about convolution, let's look at another example in 1D.

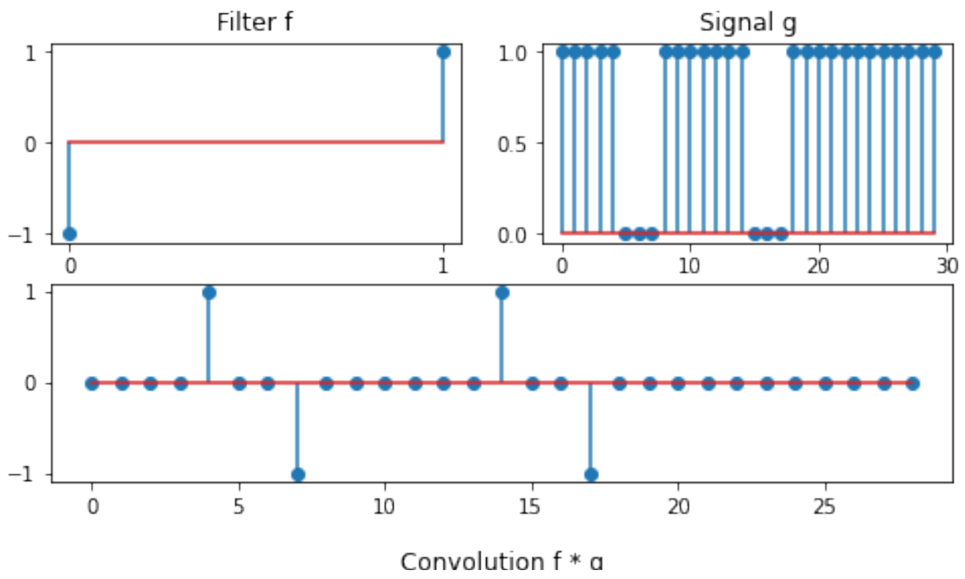
We start by defining a filter and a signal as numpy arrays.

```
In [11]: import numpy as np

         # create filter
         f = np.array([-1.,1.])

         # create signal
         g = np.ones(30)
         g[5:8] = 0
         g[15:18] = 0

         # create convolution
         conv_result = conv_naive(f,g)
```

Here, the signal has "jumps" or "edges" between 0 and 1 at a few locations.

The filter equals ± 1 at these edges, and zero everywhere else. If g was an audio signal, then f would detect boundaries between silence.

```
In [20]: import matplotlib.pyplot as plt
         plt.rcParams['figure.figsize'] = [8, 4]

         ax1 = plt.subplot(221)
         ax1.stem(range(len(f)), f, use_line_collection=True)
         ax1.set_title('Filter f')
         ax1.set_xticks([0,1])
         ax2 = plt.subplot(222)
         ax2.stem(range(len(g)), g, use_line_collection=True)
         ax2.set_title('Signal g')
         ax3 = plt.subplot(212)
         ax3.stem(range(len(conv_result)), conv_result, use_line_collection=True)
         ax3.set_title('Convolution f * g', y=-0.5)

         plt.show()
```



Example: Smoothing

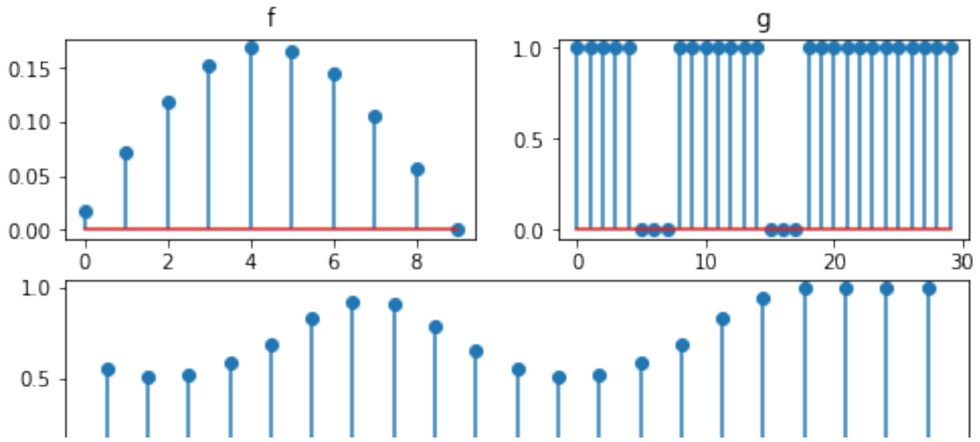
Another application of convolutions is to smooth the signal g.

We again define a filter and a signal as numpy arrays.

In [21]:
```
# create signal
g = np.ones(30)
g[5:8] = 0
g[15:18] = 0

# create filter
f = np.sin(np.linspace(0.1,np.pi,10))
f /= f.sum()

# create convolution
conv_result = conv_naive(f,g)
```

The filter f is a "mini bell curve". When applied to the same signal, it smoothes-out the "kinks".

```
In [22]: ax1 = plt.subplot(221)
         ax1.stem(range(len(f)), f, use_line_collection=True)
         ax1.set_title('f')
         ax2 = plt.subplot(222)
         ax2.stem(range(len(g)), g, use_line_collection=True)
         ax2.set_title('g')
         ax3 = plt.subplot(212)
         ax3.stem(range(len(conv_result)), conv_result, use_line_collection=True)
         ax3.set_title('f * g', y=-0.5)

         plt.show()
```



What Are Convolutions Doing?

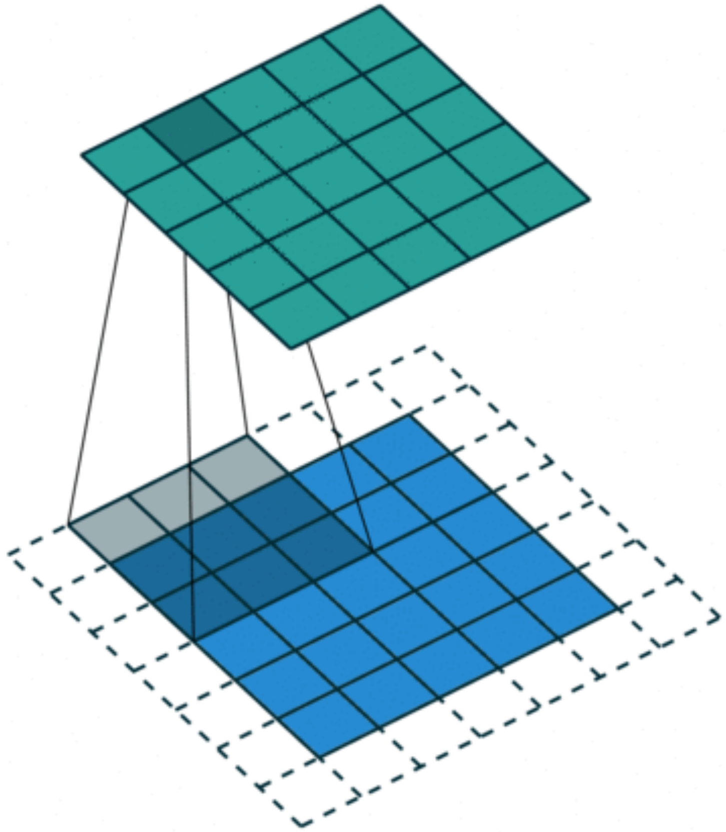
The convolution applies the filter f onto the signal g.

- This can change the behavior of g (e.g., smoothing it out).
- Or it can extract useful characteristics from g (e.g., edge detection, or when does the signal changes drastically).

We can use convolutions in machine learning models to *learn* the filters.

Convolutions in 2D

Convolutions can be extended to 2D by expanding dimensions of f and g.



In this example, f is a 3x3 array (gray area). g is a 5x5 array (blue area). The result of the convolution is the top green area.

In the context of neural networks, f are learnable weights, g are inputs from previous layers.

- The signal g can be an image.
- The filter f can be a "feature" of the image, like an edge or a more complex object.

The results of applying the convolution are fed to the next layer.

Convolutional Layers

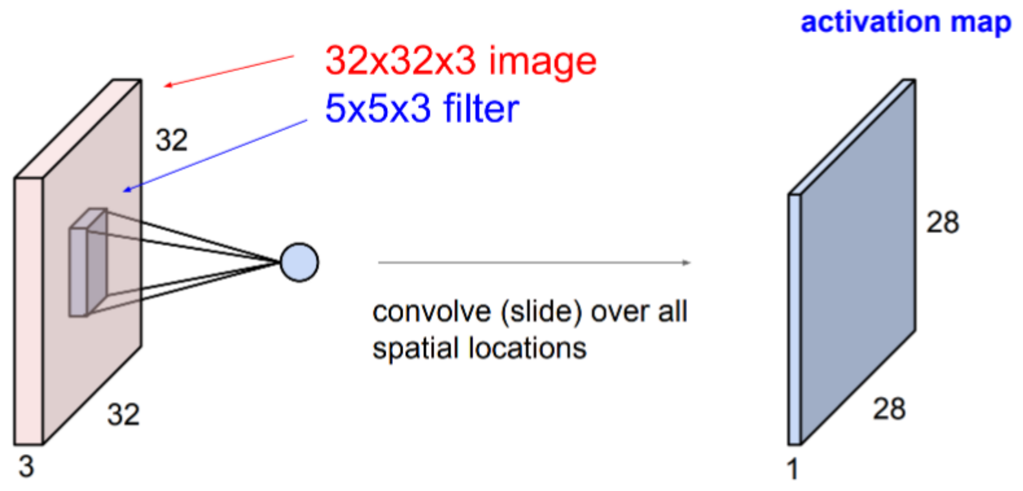
A convolution layer is a model $f : \mathbb{R}^d \to \mathbb{R}^p$ that applies p convolutions in parallel to an input x.

$$f(x) = \mathrm{conv}(W, x) = \begin{bmatrix} \mathrm{conv}(w_1, x) \\ \mathrm{conv}(w_2, x) \\ \vdots \\ \mathrm{conv}(w_p, x) \end{bmatrix},$$

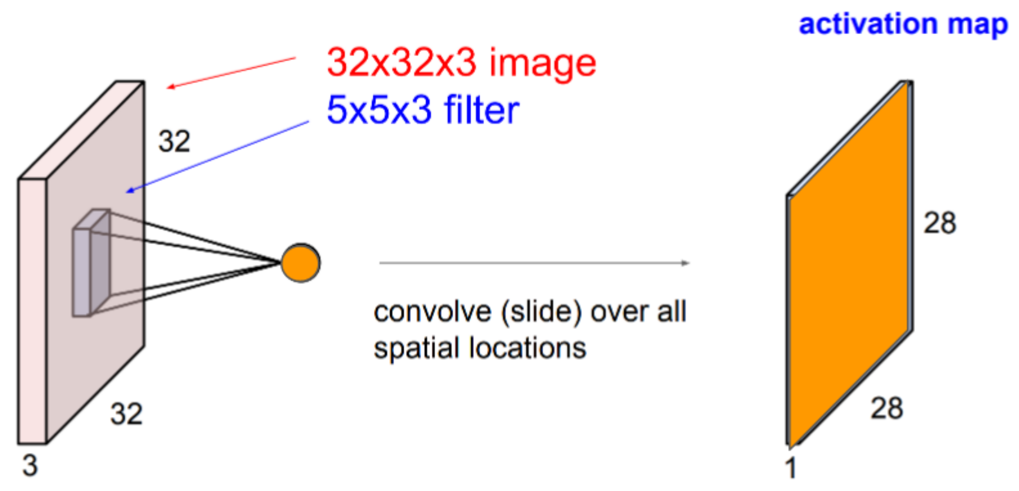
where each w_k is the weights for the k-th convolution filter (think f in our 1D example).

Understanding 2D Convolutional Layers

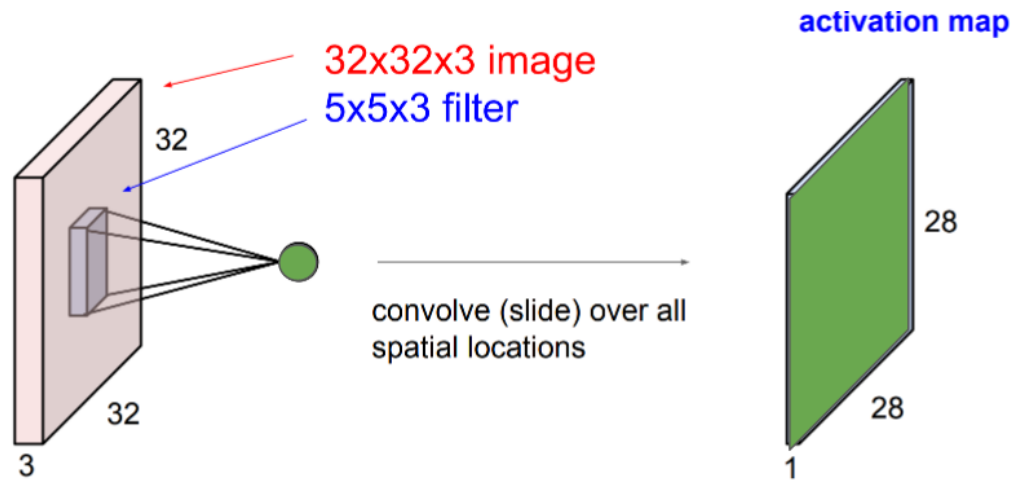
Each convolution filter in the conv layer produces an output map called the activation map.



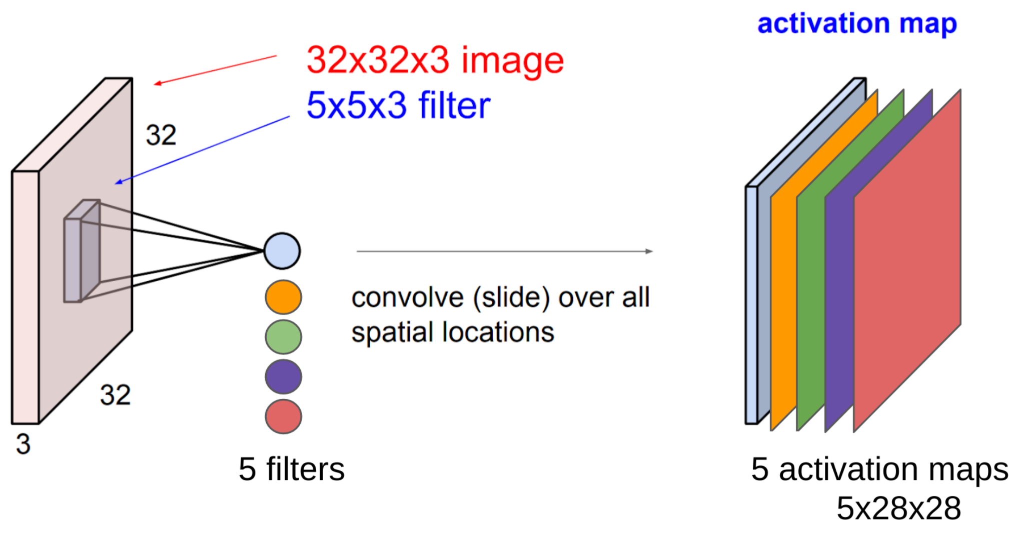
A different filter produces a different activation map.



Each filter produces its own activation map.



All activation maps in the same conv layer are concatenated together to form the final multi channel output:



The activation map contains useful information about the input image.

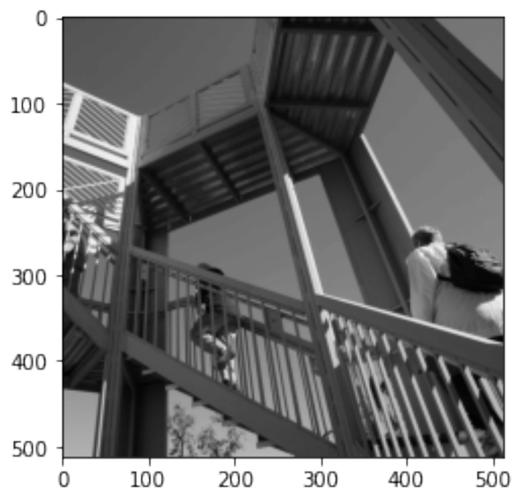
Example: Edge Detection in 2D

Let's revisit our edge detection example, but this time the signal g will be a 2D image.

We will use two filters, each of which is defined below. We will also load an image as the signal g.

In [25]:
```python
from scipy import misc

# define the filters
sobel_x = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
sobel_y = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])

# load and visualize the "signal"
img = misc.ascent()
plt.imshow(img, cmap='gray')
```

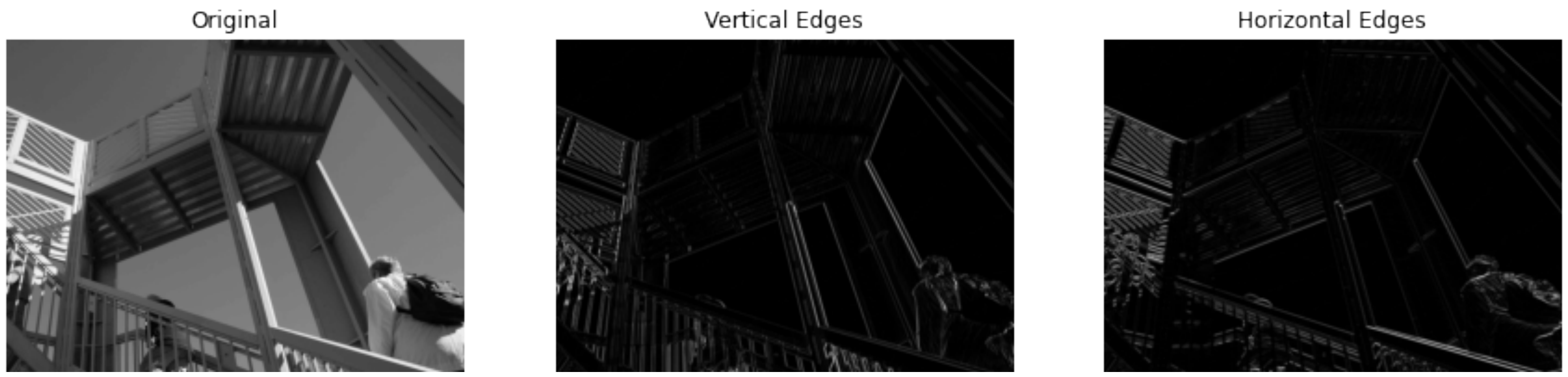
Out[25]:
```
<matplotlib.image.AxesImage at 0x1222c62b0>
```



We can now convolve and visualize the filters with the image.

The result is an activation map with high activation around horizontal and vertical edges -- parts of the image where it changes from light to dark.

```
In [26]: from scipy import signal

         # create the convolution with each filter
         grad_x = signal.convolve2d(img, sobel_x, boundary='symm', mode='same')
         grad_y = signal.convolve2d(img, sobel_y, boundary='symm', mode='same')

         # visualize the convolution
         fig, (ax_orig, ax_x, ax_y) = plt.subplots(1, 3, figsize=(15, 15))
         ax_orig.imshow(img, cmap='gray')
         ax_orig.set_title('Original')
         ax_orig.set_axis_off()
         ax_x.imshow(np.absolute(grad_x), cmap='gray')
         ax_x.set_title('Vertical Edges')
         ax_x.set_axis_off()
         ax_y.imshow(np.absolute(grad_y), cmap='gray')
         ax_y.set_title('Horizontal Edges')
         ax_y.set_axis_off()
```



When to Use Convolutions?

Convolutions work best on inputs containing "attributes" that are interesting, but whose location in the input is not important.

Images are by nature good examples of this assumption. Convolution layers can learn similar functions as fully connected layers, but with much smaller parameters.

Pooling Layers

A pooling layer is a model $f : \mathbb{R}^d \to \mathbb{R}^p$ that applies pooling operations to an input x.

$$f(x) = \text{pooling}(x)$$

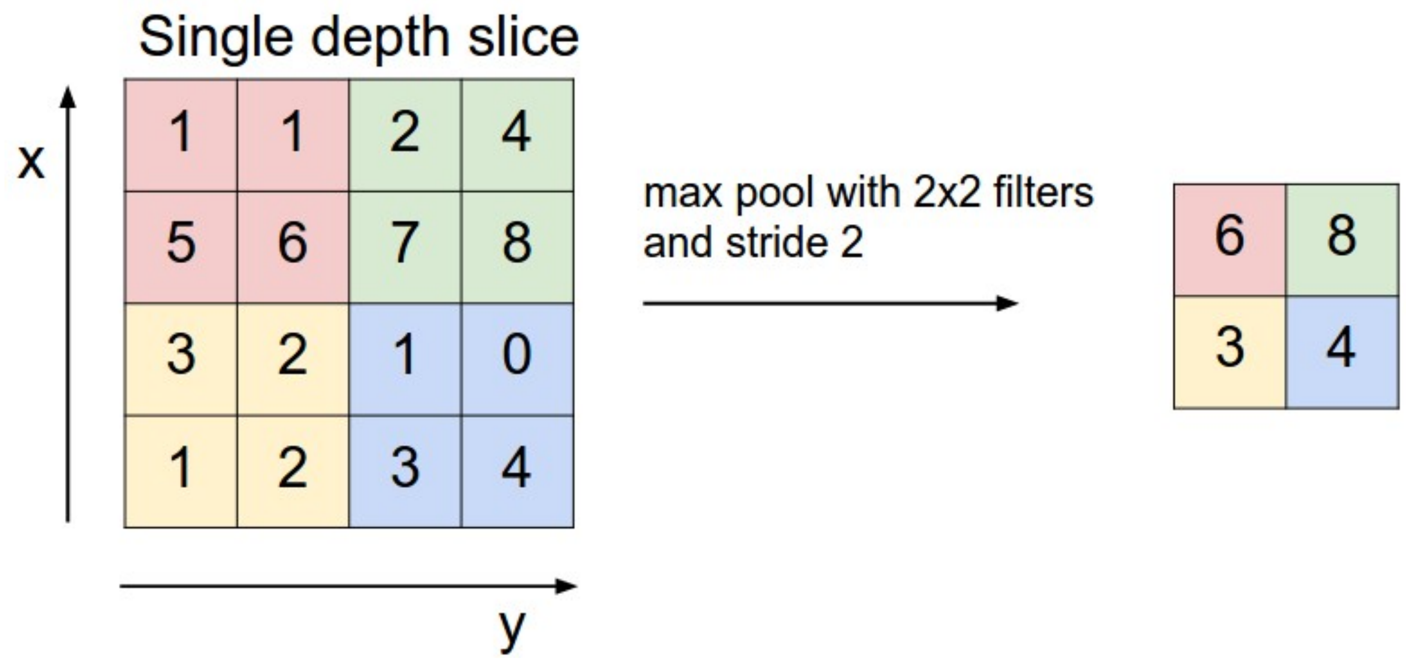
where $pooling$ is a pre-defined operation applied over the input x.

Pooling layer does not have learnable weights.

Pooling

Pooling is a common operation to be added after a convolution layer. It is applied to each activation map separately and reduces the spatial size of its input.

Max Pooling: for each region in the input, return the max value. This is the most common type.



Average Pooling: for each region in the input, return the average value.

Purpose of Pooling

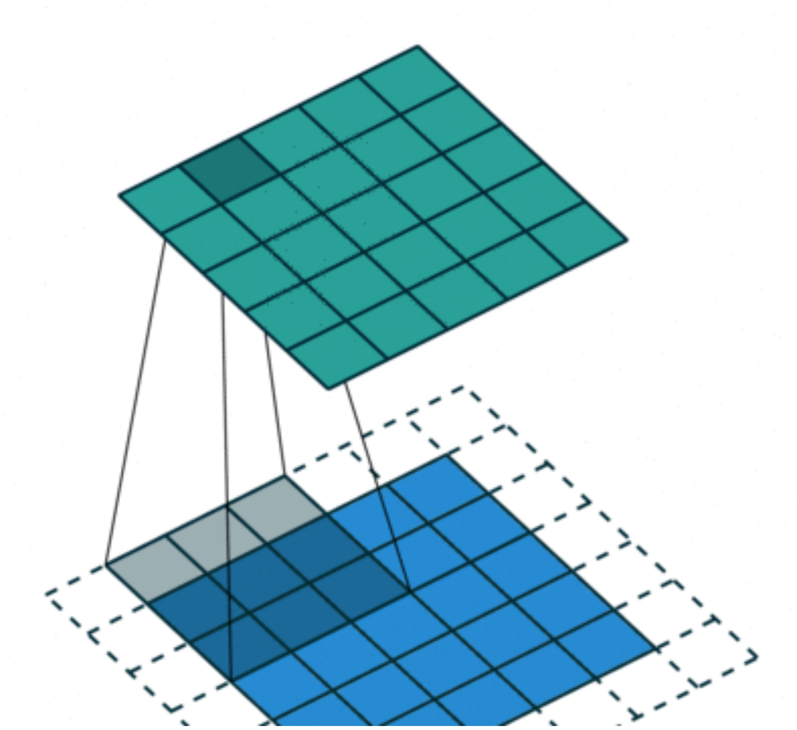
- Allows slight translation invariance inside the pooling window: exact location of the relevant feature is not critical. This improves robustness of feature detection.
- When used together with convolutions, pooling reduces spatial size of feature maps to capture more abstract representations.

Part 3: Convolutional Neural Networks

Convolutional neural networks use convolutional layers to process signals such as images, audio, and even text.

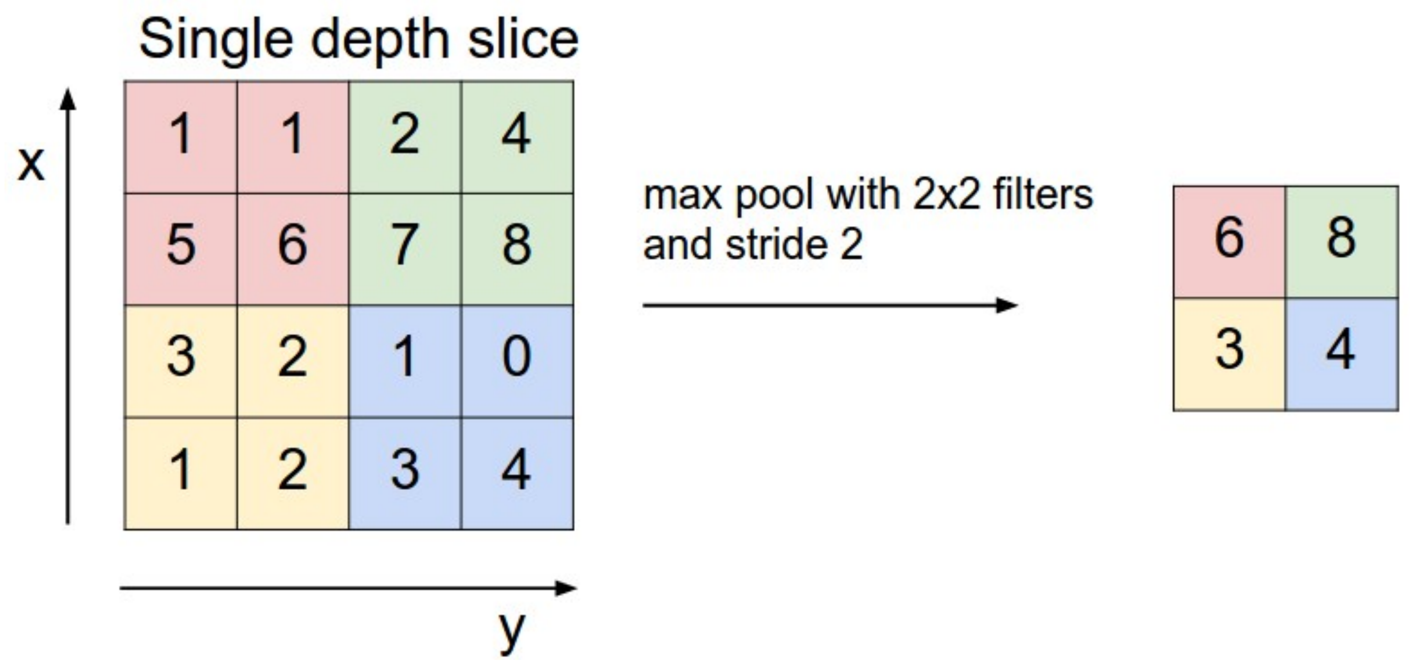
Review: Convolution

Convolutions apply a filter f (gray) over a signal g (blue). The output is an activation map (green).



Review: Pooling

Pooling is a common operation to be added after a convolution layer. It is applied to each activation map separately and reduces the spatial size of its input.



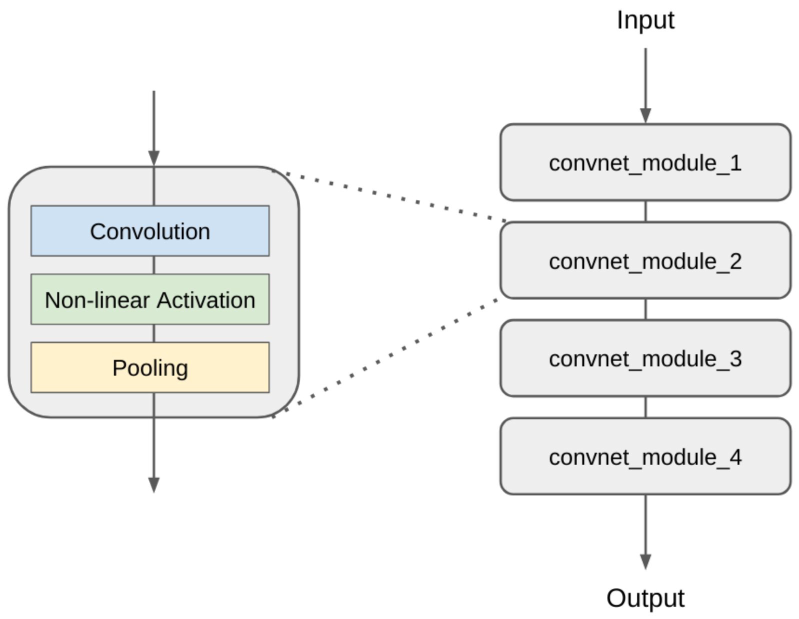
Convolutional Neural Networks

A convolutional neural network (CNN) is a model $f : \mathbb{R} \to \mathbb{R}$ that consists of a composition of L neural network layers that contain convolutions:

$$f(x) = f_L \circ f_{L-1} \circ \dots f_1(x).$$

The final f_L is often a fully connected output later of size one.

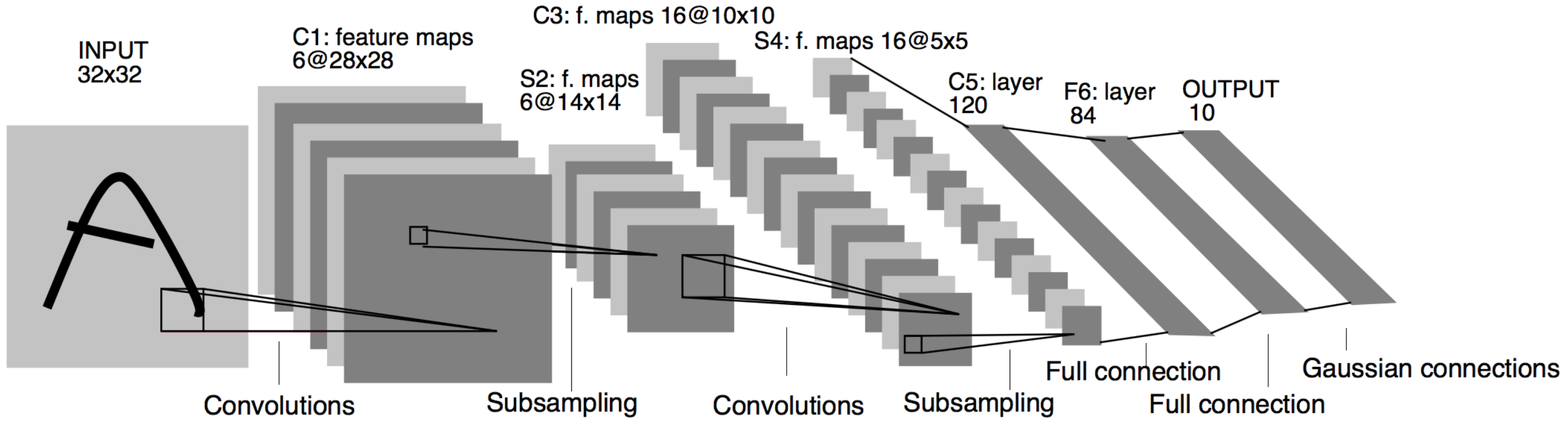
Typically, CNNs are made of consecutive convolution + activation + pooling layers that form into blocks.



Next we are going to see a few famous examples.

LeNet for MNIST Digits Recognition

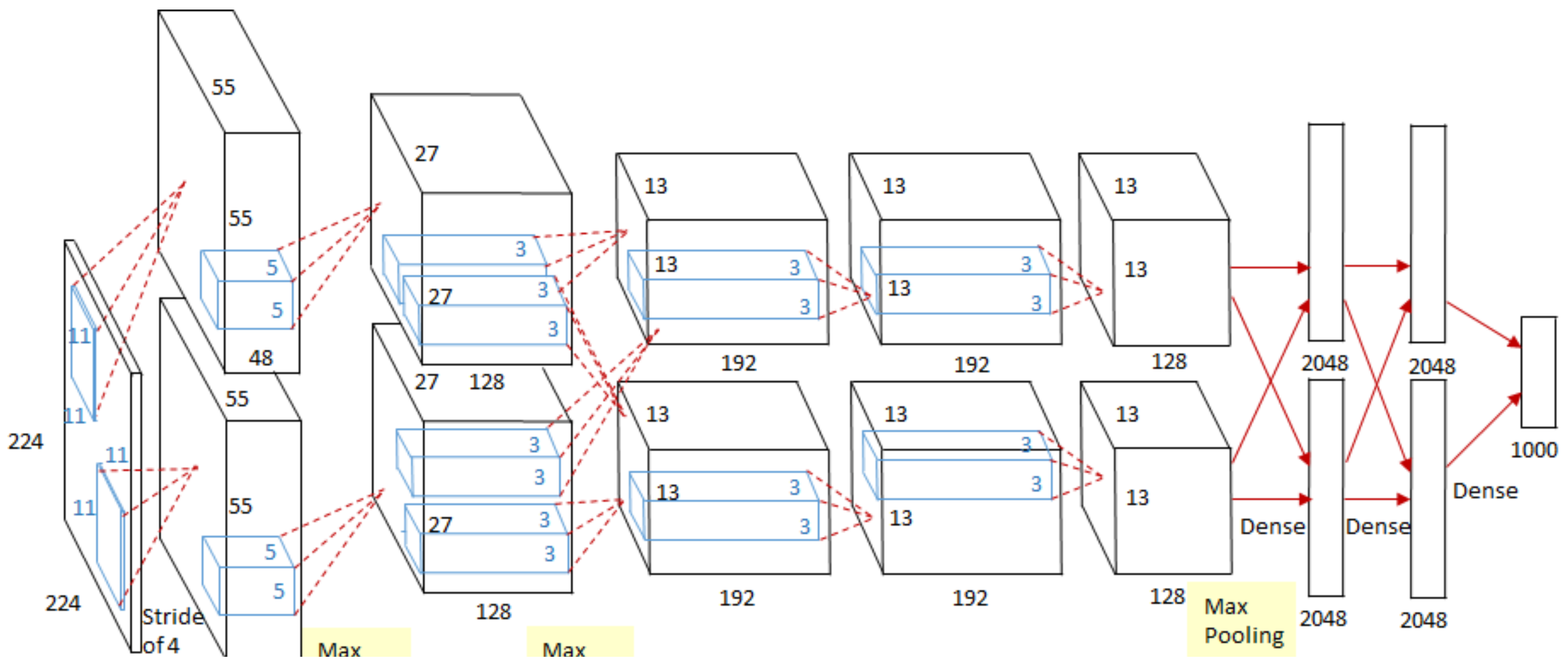
LeNet successfully used CNNs for digit recognition [LeCun, Bottou, Bengio, Haffner 1998].



- LeNet contains multiple convolutional and max-pooling (called subsampling in the original paper) layers and a fully connected MLP in the end.
- Given an image of a hand-written digit, LeNet predicts it to be one of the 10 classes (0-9).
- This work inspired many subsequent CNN architectures.

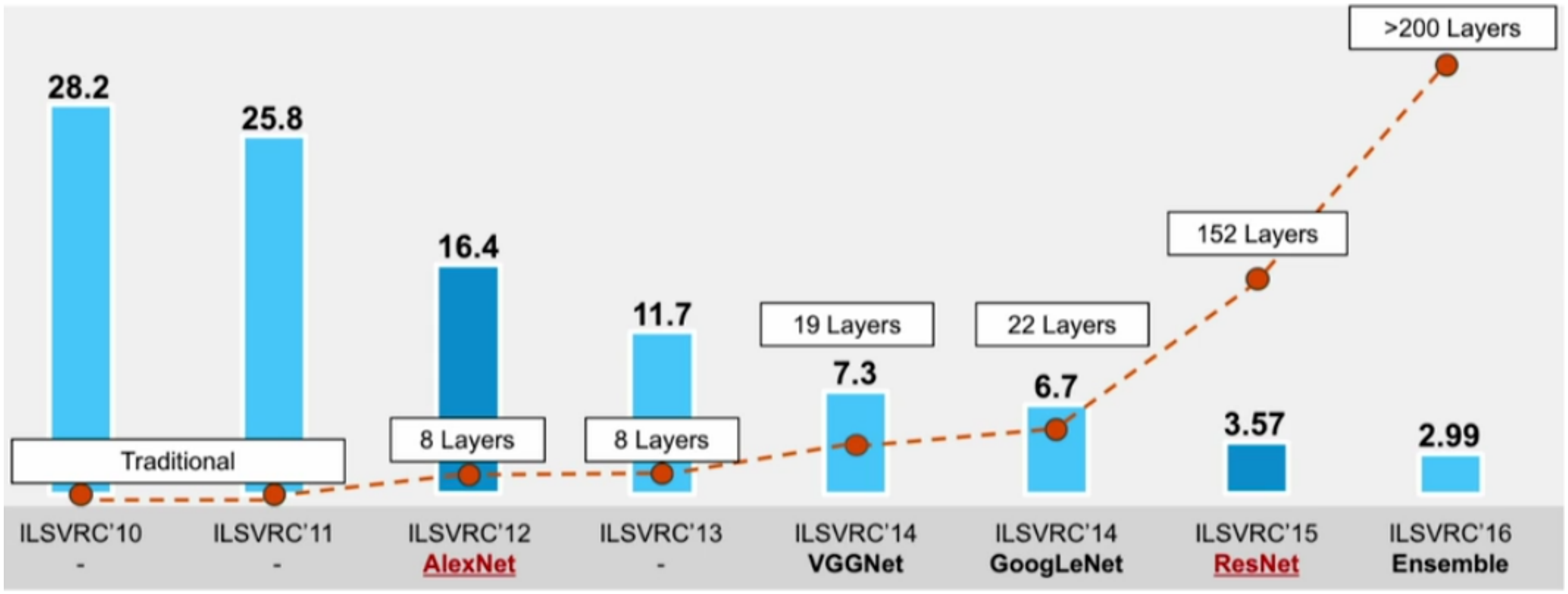
AlexNet and ImageNet

ImageNet classification with deep convolutional neural networks [Krizhevsky, Sutskever, Hinton 2012]



- AlexNet contains multiple convolutional layers followed by max pooling. The prediction is 1000 dimentional because that is the number of classes in the ImageNet classification task.
- Note that in the original implementation the image is separated into two parts that are in two different GPUs due to hardware limits.

Starting from AlexNet, the performance of image classification on ImageNet has entered a new era:



http://sqlml.azurewebsites.net/2017/09/12/convolutional-neural-network/

[illegible]

Algorithm: Convolutional Neural Network

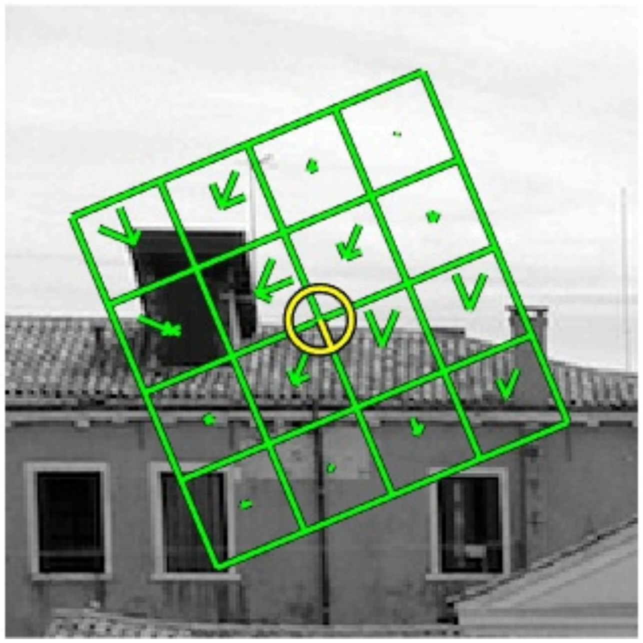
- **Type**: Supervised learning (regression and classification).
- **Model family**: Compositions of convolutional, pooling and other types of layers.
- **Objective function**: Any differentiable objective.
- **Optimizer**: Gradient descent.

CNNs and Feature Learning

Before neural networks, computer vision algorithms used hand-crafted features to extract information from raw image inputs.

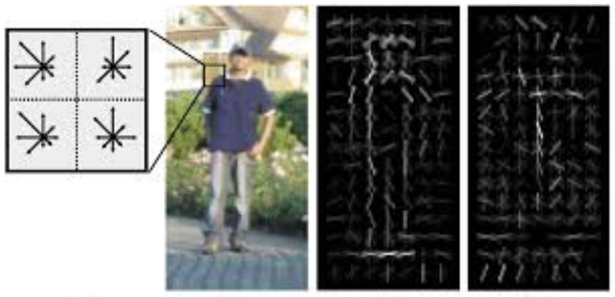
Designing such features is hard and most importantly they might not be optimal for a particular task.

One set of features that used to be popular was the Scale-Invariant Feature Transform (SIFT).



https://www.vlfeat.org/overview/sift.html

Another common set of features was the the Histogram of Oriented Gradients (HOG).


(a) (b) (c) (d)

https://sarthakahuja.org/public/docs/report_ped_detection.pdf

By using CNNs, we now can extract features from input images at different level of abstraction, without the need of designing hand-craft features.

Visualizing CNN Internals

CNNs can be seen as extracting features at low-, mid-, and high-levels of abstraction that are relevant to corresponding visual concepts.

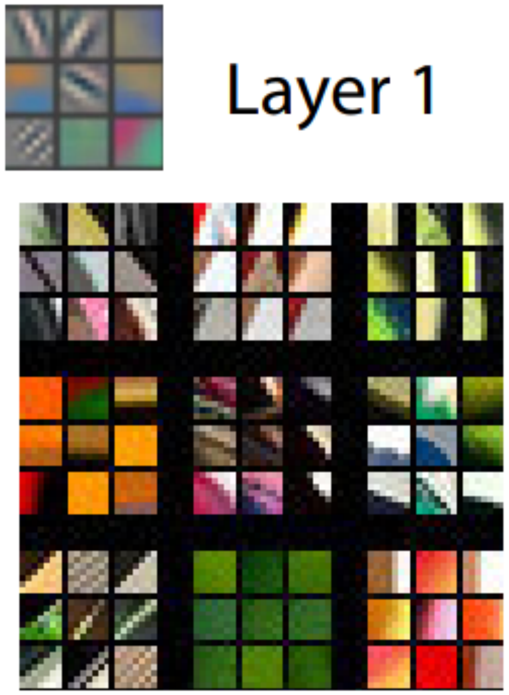
Below, we reproduce feature visualization from the paper Visualizing and Understanding Convolutional Networks [Zeiler, Fergus 2013].

For a convolutional network with

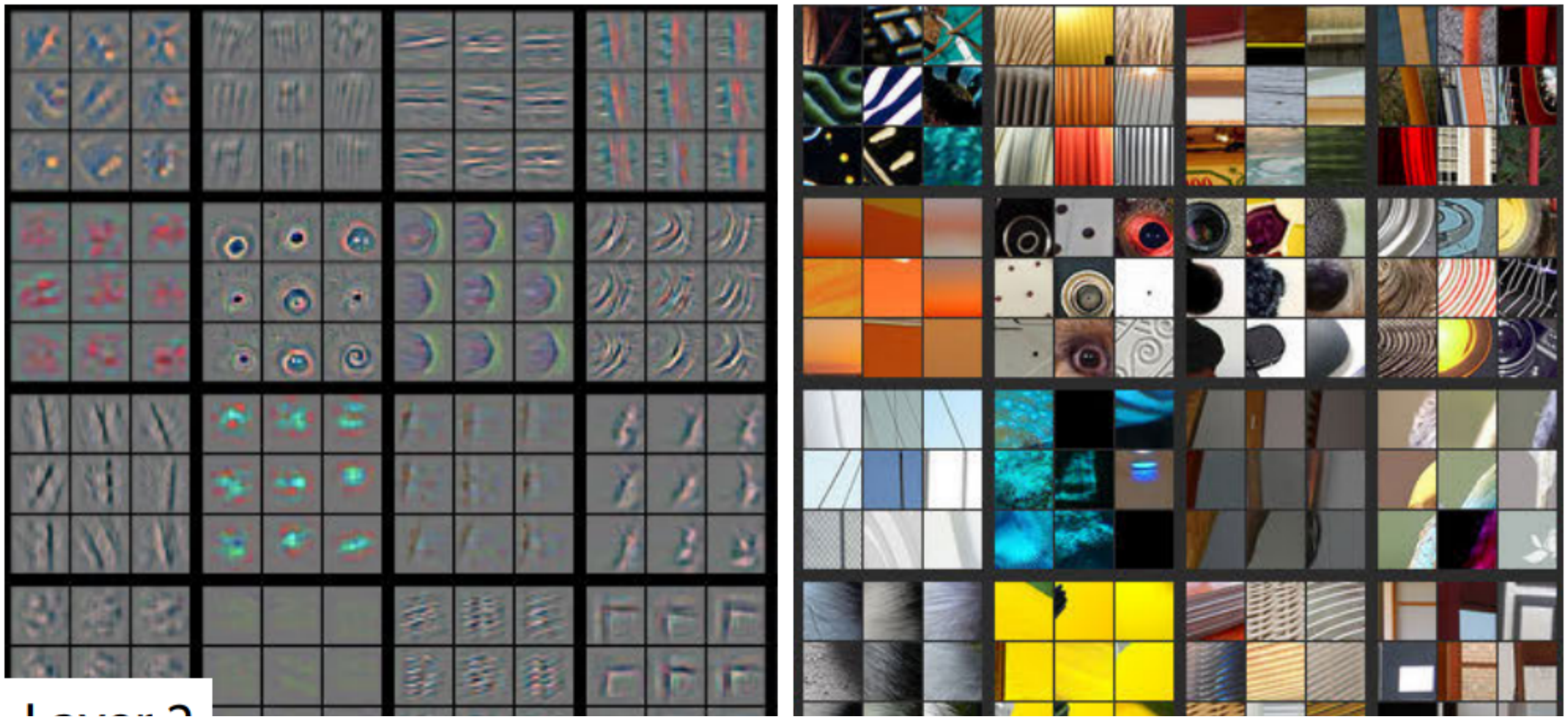
$$\text{Input} \to \text{Conv Layer1} \to \text{Conv Layer2} \to \cdots \to \text{Conv Layer5} \to \cdots \to \text{Output},$$

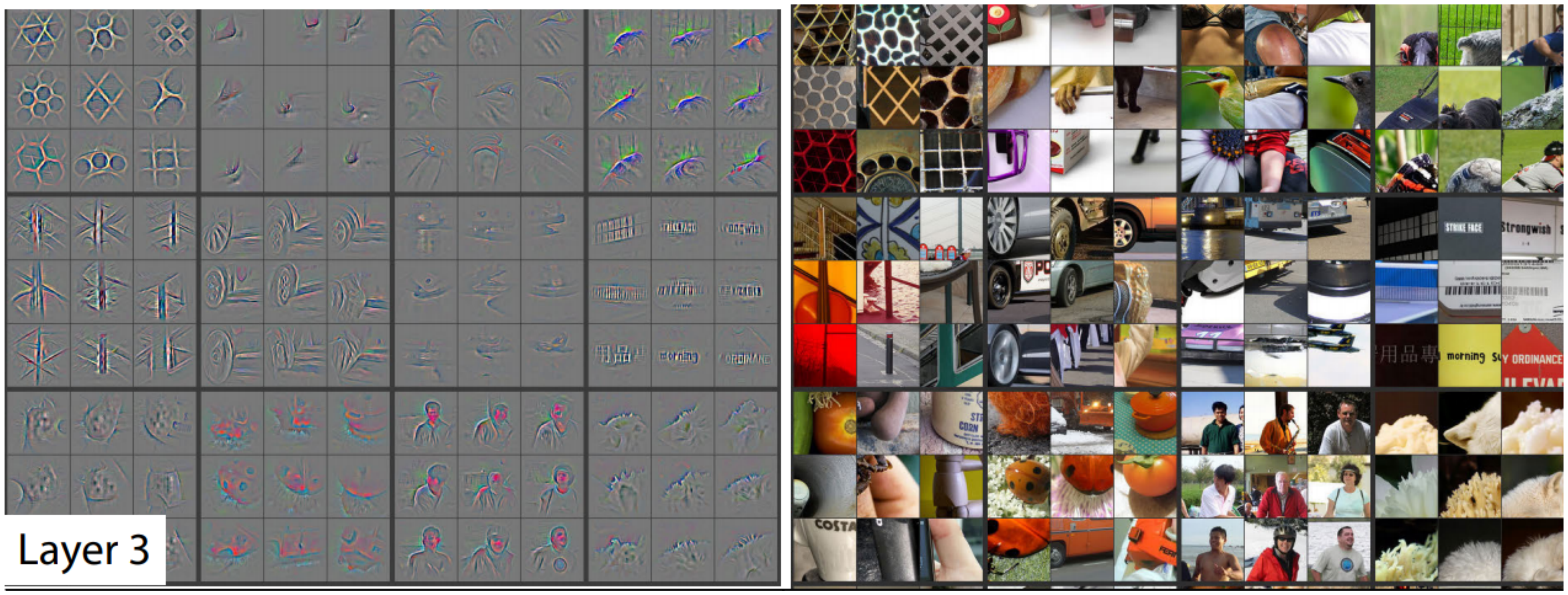
we can visualize what each layer is doing by finding the image patch with highest activation responses.

Layer 1 focuses on colored edges and blobs.

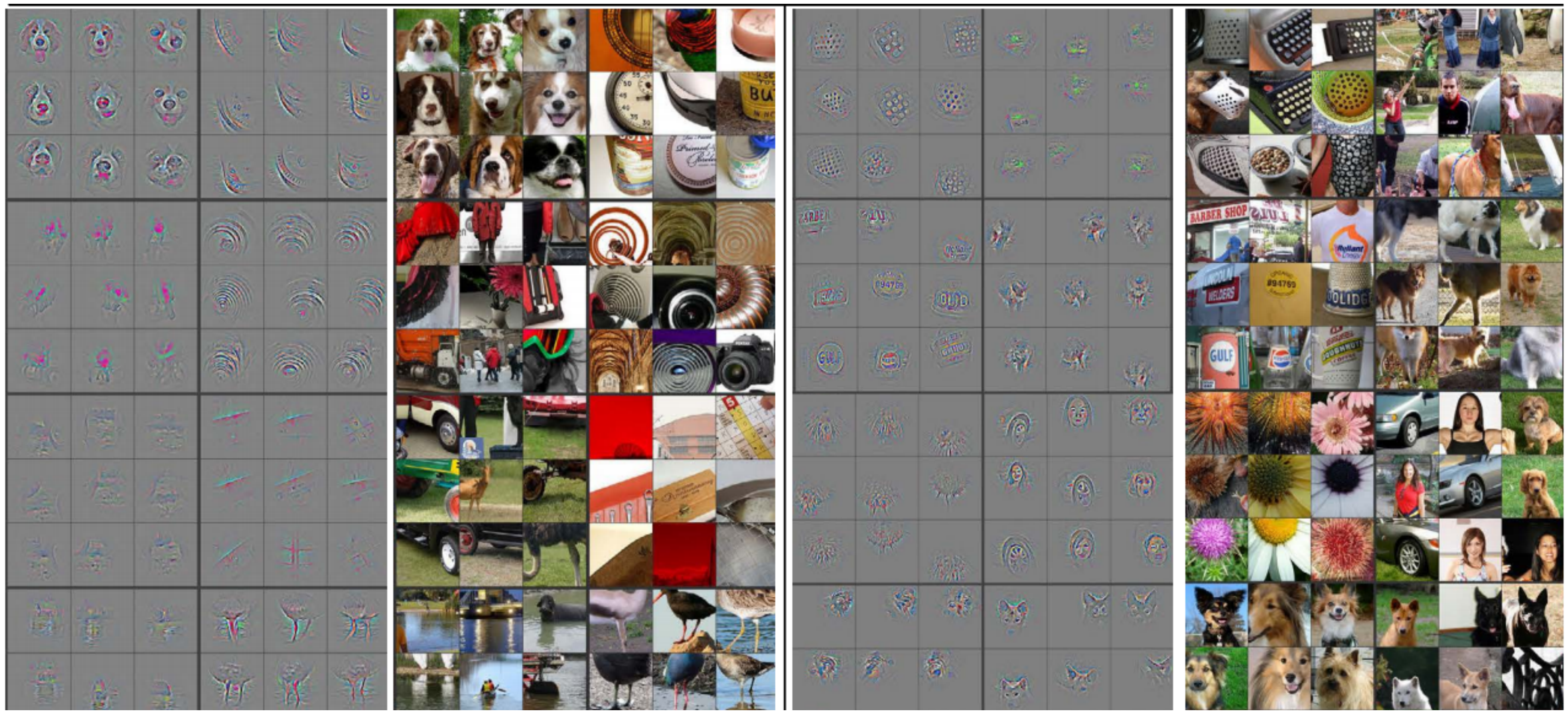


Layers 2-3 focus on object parts, such as the wheels of a car, or the beak of a bird.




Layer 3
morning
ORDINANCE
Strongwish
STRIKE FACE
COSTA

Layers 4-5 focus on object parts and even entire objects.



Pros and Cons of CNNs:

CNNs are powerful tools because they encode visual information efficiently.

- CNNs are now used as universal visual feature extractors. It is common to use ImageNet pretrained CNNs.
- They serve as building blocks in ML pipelines and can work with other types of networks such as a recurrent network for complex tasks.

Their main drawbacks are computational and data requirements.