COMP755-Lect20

November 4, 2018

1 COMP 755

Plan for today

1. Deep Learning overview
2. Neural Networks
3. Activation function
4. Forward propagation
5. Google Dream
6. Filters and convolution

2 Shallow models

What is a shallow model?

1. Linear regression
2. Logistic regression
3. Naive Bayes

Shallow models tend to use linearly weighted features.

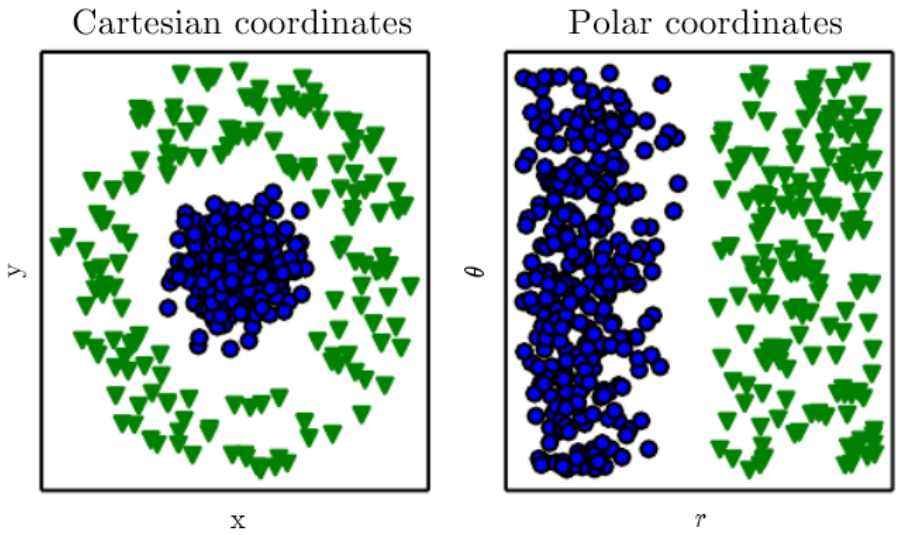
3 XOR is hard for shallow models

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Linear combinations of `a` and `b` cannot be used to predict `a XOR b`.

4 New features can make prediction easier

However, if we have an additional feature `(a AND b)`



a	b	a AND b	a XOR b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

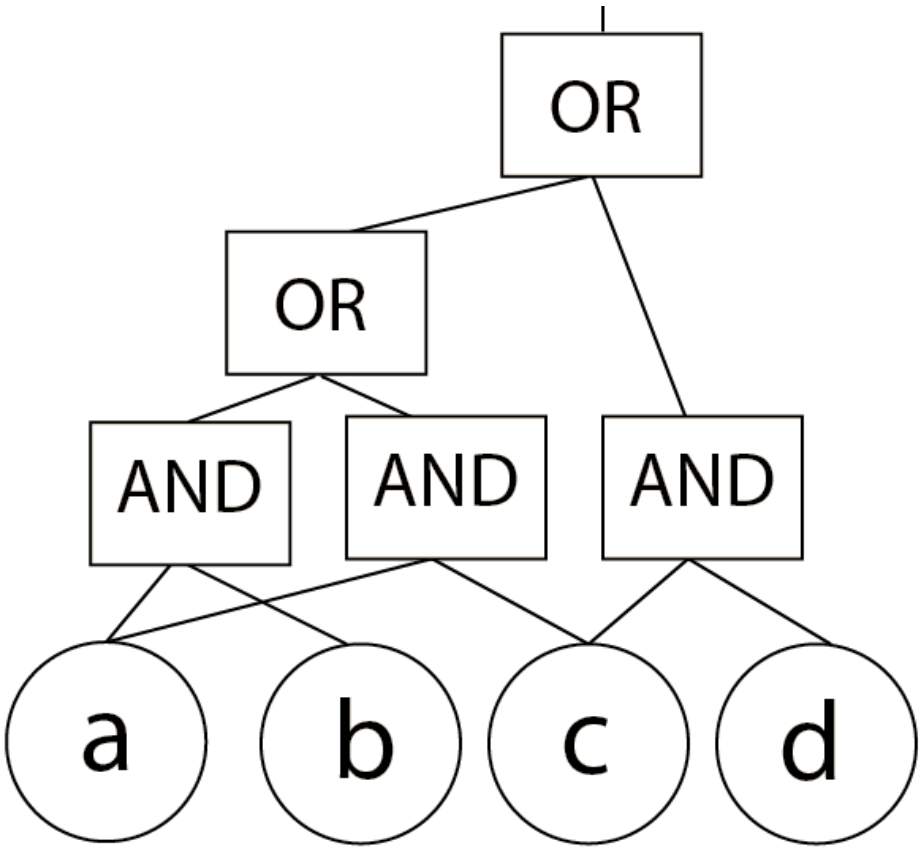
Then it is easy `a + b - (a AND b)`
We will show an example of deep learning network that learns such features from scratch.

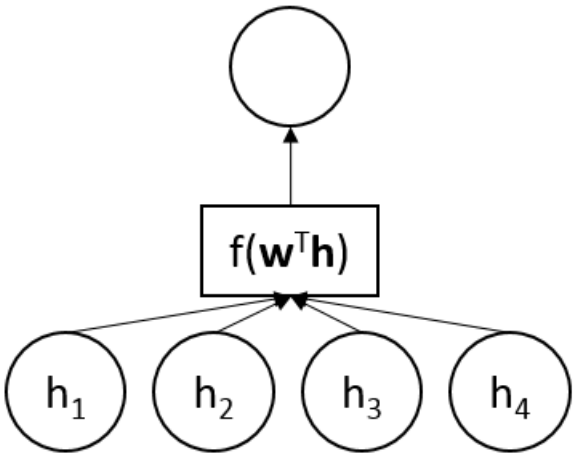
5 Different representation of the features can make task easier

Here is an example of a classification problem.
The two representation of the same data are shown.
Representing samples in polar coordinates makes classification task much easier.

6 Deep learning discovers new representations

Deep learning methods construct features/representations that make learning tasks easier.
How is this accomplished?
Let's consider a somewhat complicated feature
`((a AND b) OR (a AND c)) OR (c AND d)`
Q: How would you compute this expression on a CPU?


OR
OR
AND
AND
AND
a
b
c
d



7 Depth of computation

8 Building complex features from simple features -- forward propagation

Computation of the feature `((a AND b) OR (a AND c)) OR (c AND d)` requires us to build up simpler parts first

- h_1 = `(a AND b)`
- h_2 = `(a AND c)`
- h_3 = `(c AND d)`

Using these features we can compute more complex feature

- $h_4 = h_1$ `OR` h_2

And finally,

- $h_5 = h_4$ `OR` h_3

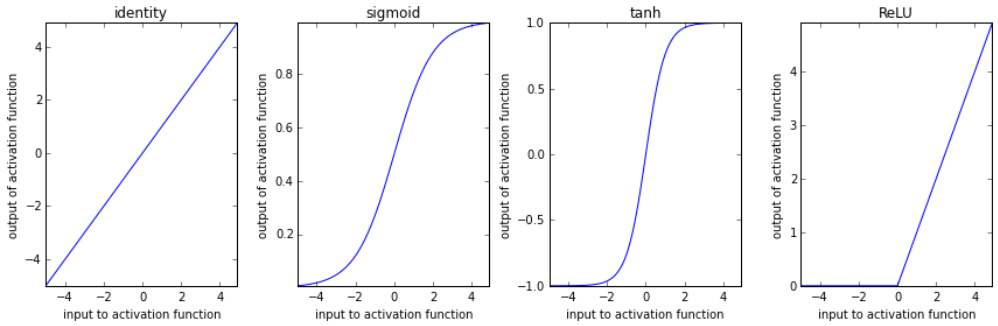
9 Units in deep learning networks

Our computational tree had nodes that computed logical operations (`AND` and `OR`).

Deep learning networks use more general nodes.

Given a set of inputs $\mathbf{h}$ a node evaluates $f(b + \mathbf{w}^T\mathbf{h}) = f(b + \sum_i w_i h_i)$.

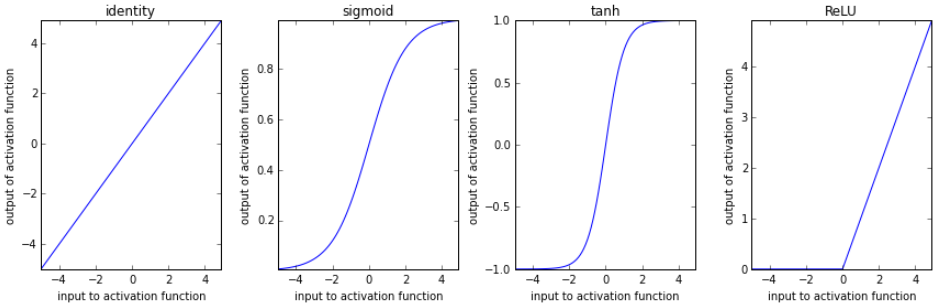
b is called **bias**. $\mathbf{w}$ are referred to as **weights**. f is commonly referred to as an **activation** function.



10 Activation functions

Typical activation functions are

1. Identity $f(x) = x$
2. Sigmoid $f(x) = \frac{1}{1+e^{-x}}$
3. Hyperbolic tangent $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
4. Rectified linear unit (ReLU) $f(x) = \max(x, 0) = (x)_+ = [x > 0]x$



11 Thinking about what units do

Let's think through this a little bit. Let unit compute $f(w_1h_1 + w_2h_2 + w_3h_3 + w_4h_4)$

Questions:

1. Assume f is sigmoid, let's say that $w_1 = w_2 = w_3 = w_4 = \frac{1}{4}$. For which values of h_is is the output of the unit greater than 0.5?
2. Assume f is a ReLU, let's say that $w_1 = w_2 = 1.0$ and $w_3 = w_4 = 0$. For which values of h_is is output of the unit greater than zero?

12 One layer network captures generalized linear models

Observations:

1. A network with no hidden layers, single output node, and sigmoidal activation, computes logistic regression prediction.

$$f(b + \mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp\{-b - \mathbf{w}^t\mathbf{x}\}}$$

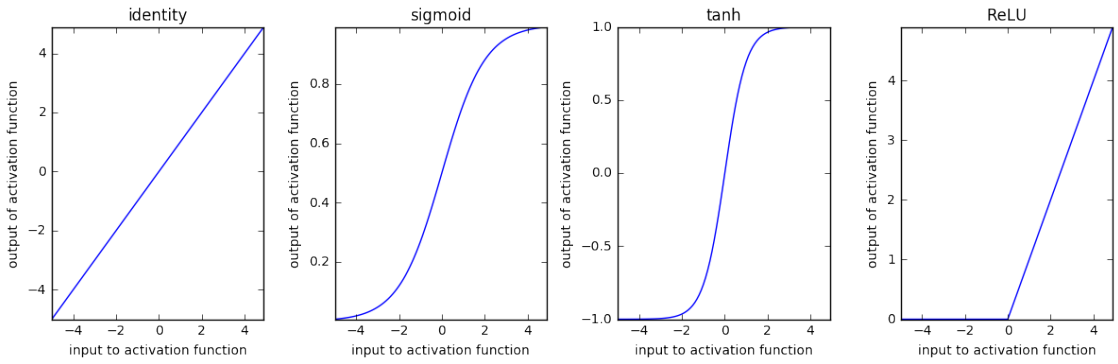
2. A network with no hidden layers, single output node, and identity activation, computes linear regression prediction.

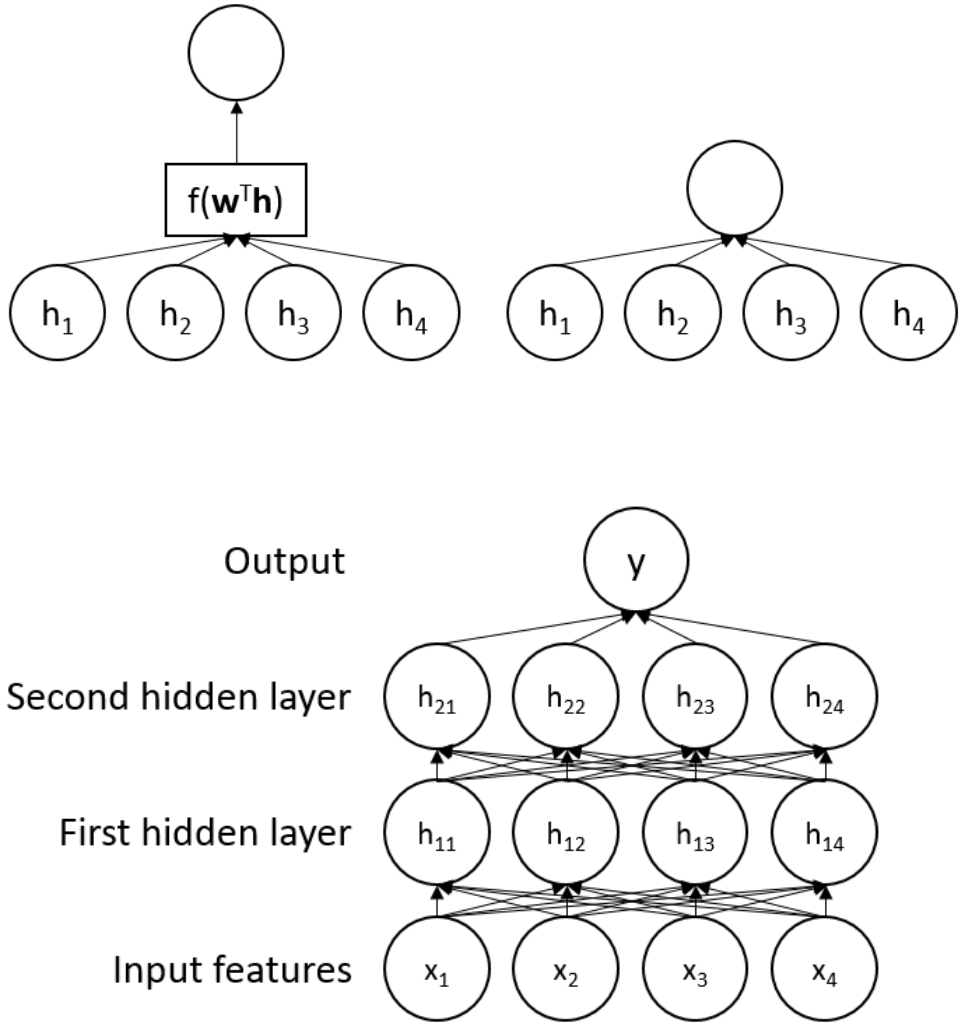
$$f(b + \mathbf{w}^T\mathbf{x}) = b + \mathbf{w}^T\mathbf{x}$$

```
In [19]: import numpy as np
         import matplotlib.pyplot as plt
         from __future__ import print_function
         %matplotlib inline
         plt.figure(figsize=(12,4))
         x = np.arange(-5.0,5.0,0.1)
         activations = [('identity',lambda x:x),
                        ('sigmoid' ,lambda x:1./(1. + np.exp(-x))),
                        ('tanh'    ,lambda x:(np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))),
                        ('ReLU'    ,lambda x: (x>0)*x)]

         for (i,act) in enumerate(activations):
             name = act[0]
             f = act[1]
             plt.subplot(1,len(activations),i+1)
             plt.title(name)
             plt.plot(x,f(x))
             plt.xlim([np.min(x),np.max(x)])
             plt.ylim([np.min(f(x)),np.max(f(x))])
             plt.xlabel('input to activation function')
             plt.ylabel('output of activation function')

         plt.tight_layout()
```





13 Conventions when drawing a network

Typically we don't explicitly denote the activation function but rather just connect the input variables to the output of the unit

14 Deep networks are composed of layers

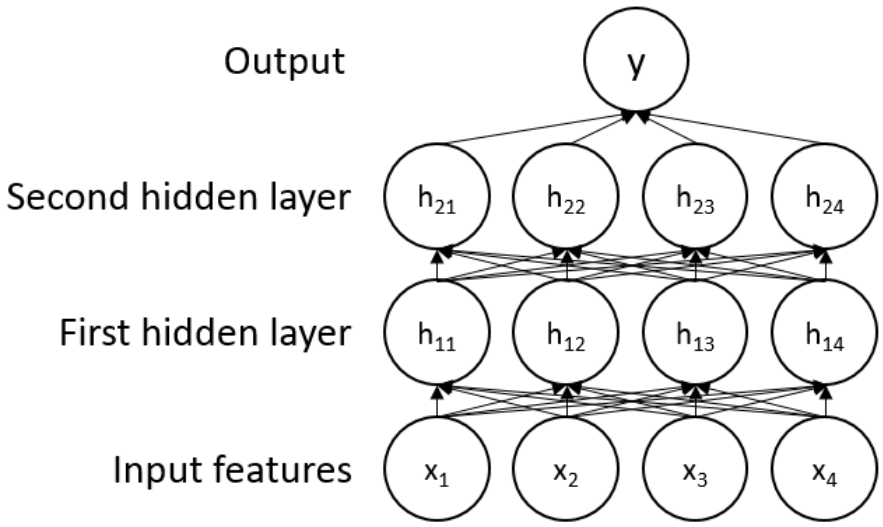
Typically the deep networks are built up out of layers.

15 Deep networks compute by forward propagation

For simplicity we can say that activation function is the same throughout the network.

Say we computed state of layer $l-1$, $\mathbf{h}_{l-1}$, next we compute state of $\mathbf{h}_l$

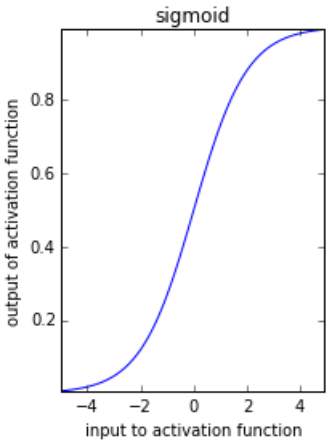
$$h_{l,i} = f(b_{l,i} + \mathbf{w}_{l,i}^T \mathbf{h}_{l-1})$$

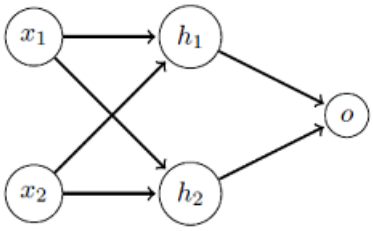


Observations: 1. States of nodes in layer l can be computed in parallel 2. Each node has its own set of weights and a bias -- plenty of parameters

16 We are going to play with a network for XOR

Sigmoid Activation XOR network parameterization





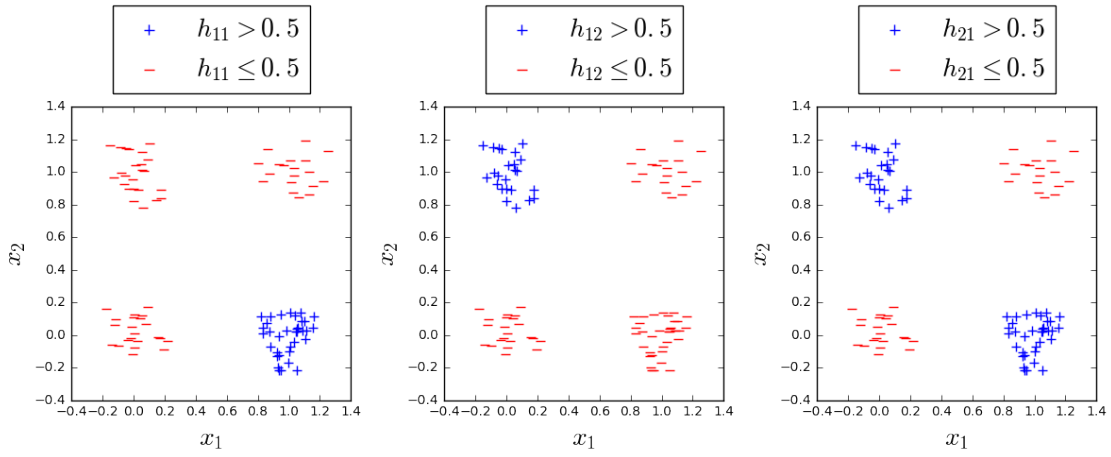
An XOR Network

x_1	0	1	0	1
x_2	0	0	1	1
$h_1 = \sigma(F(-0.5 + x_1 - x_2))$	0	1	0	0
$h_2 = \sigma(F(-0.5 - x_1 + x_2))$	0	0	1	0
$y = \sigma(F(-0.5 + h_1 + h_2))$	0	1	1	0

```
In [20]: import numpy as np
         N = 100
         activation = lambda x: 1./(1. + np.exp(-x))
         # input layer
         x1 = (np.random.rand(N,1)>0.5) + 0.1*np.random.randn(N,1)
         x2 = (np.random.rand(N,1)>0.5) + 0.1*np.random.randn(N,1)
         x = np.hstack((x1,x2))
         F = 20
         # hidden layer
         b11 = -0.5*F
         b12 = -0.5*F
         w11 = np.asarray([1, -1 ])*F
```

```
def show(x1,x2,h,name):
    pos = [i for i in range(N) if h[i]>0.5]
    neg = [i for i in range(N) if h[i]<=0.5]
    plt.scatter(x1[pos],x2[pos],marker='+',color='b',s=60)
    plt.scatter(x1[neg],x2[neg],marker='_',color='r',s=60)
    plt.xlabel('$x_1$',fontsize=20)
    plt.ylabel('$x_2$',fontsize=20)
    plt.legend(['$'+name+'>0.5$', '$' + name + '\leq 0.5$'],loc='lower center',
               bbox_to_anchor=(0.5, 1.0),
               scatterpoints=1,fontsize=20)

plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
show(x1,x2,h11,'h_{11}')
plt.subplot(1,3,2)
show(x1,x2,h12,'h_{12}')
plt.subplot(1,3,3)
show(x1,x2,h21,'h_{21}')
plt.tight_layout()
```



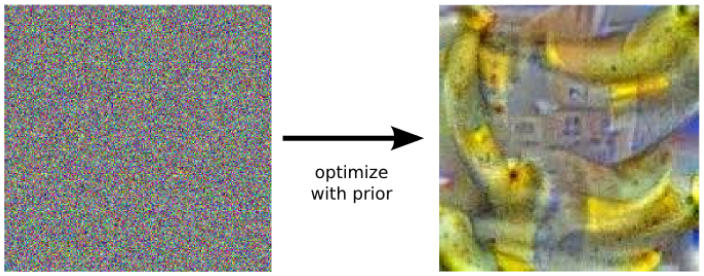
17 Google Deep Dream

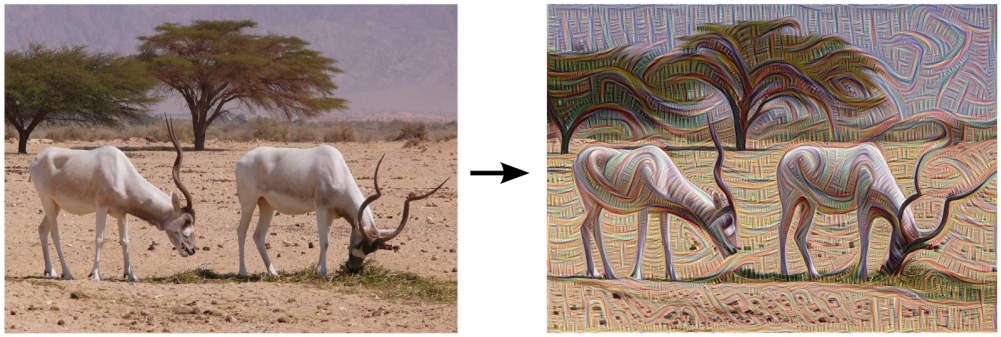
Given a trained neural network you can think of forward propagation as a mapping from features $\mathbf{x}$ to output y.

One question you can ask is: Given an input $\mathbf{x}$ what is a nearby input $\tilde{\mathbf{x}}$ that produces particular output y?

In our XOR network, we can take an input $(0.2, 0.4)$ which yields output 0.0 and ask for a nearby input that yields output 1.0. With minor caveats, we'd get $(0.2, 0.51)$.

More interesting example: Take a neural network trained to recognize objects in images. Take a complete noise image and ask for a nearby image that classifies as a banana.





18 The idea taken further

In principle, we can also ask which nearby input maximally activates any unit in the network -- not just the output.

19 Convolutional layers

Convolutional layers are typical in computer vision applications.

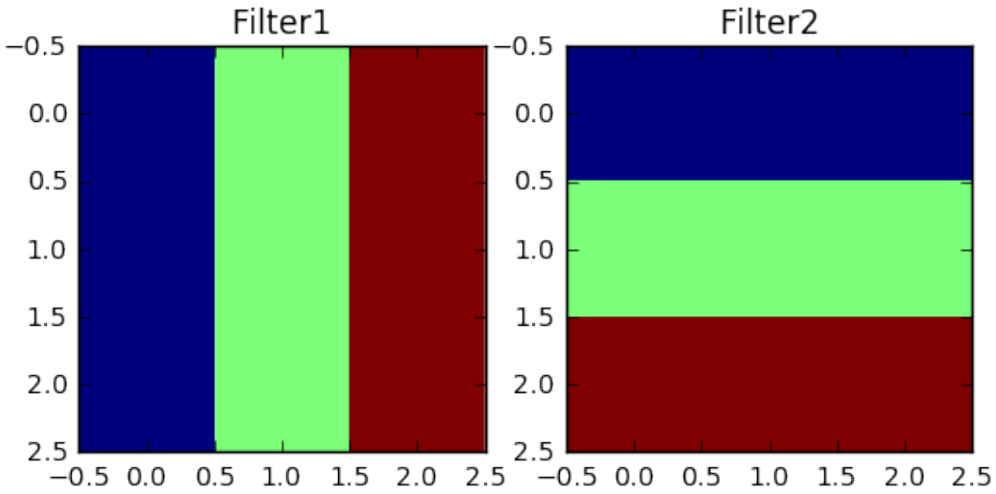
In order to understand them we need to introduce concept of **filters** and **convolution**.

```
In [21]: from scipy import misc
         import numpy as np
         import matplotlib.pyplot as plt


         filter1 = np.asarray([[-1.0, 0.0, 1.0],
                               [-1.0, 0.0, 1.0],
                               [-1.0, 0.0, 1.0]])
         filter2 = np.asarray([[-1.0,-1.0,-1.0],
                               [ 0.0, 0.0, 0.0],
                               [+1.0,+1.0,+1.0]])
         filter1vec = filter1.reshape(9,1)
         filter2vec = filter2.reshape(9,1)
         plt.subplot(1,2,1)
```

```
        plt.imshow(filter1,interpolation='None')
        plt.title('Filter1')
        plt.subplot(1,2,2)
        plt.imshow(filter2,interpolation='None')
        plt.title('Filter2')
        print("Filter 1 as matrix:\n",filter1)
        print("Filter 1 as vector:\n",filter1vec.T)
        print("Filter 2 as matrix:\n",filter2)
        print("Filter 2 as vector:\n",filter2vec.T)
```

```
Filter 1 as matrix:
 [[-1.  0.  1.]
 [-1.  0.  1.]
 [-1.  0.  1.]]
Filter 1 as vector:
 [[-1.  0.  1. -1.  0.  1. -1.  0.  1.]]
Filter 2 as matrix:
 [[-1. -1. -1.]
 [ 0.  0.  0.]
 [ 1.  1.  1.]]
Filter 2 as vector:
 [[-1. -1. -1.  0.  0.  0.  1.  1.  1.]]
```

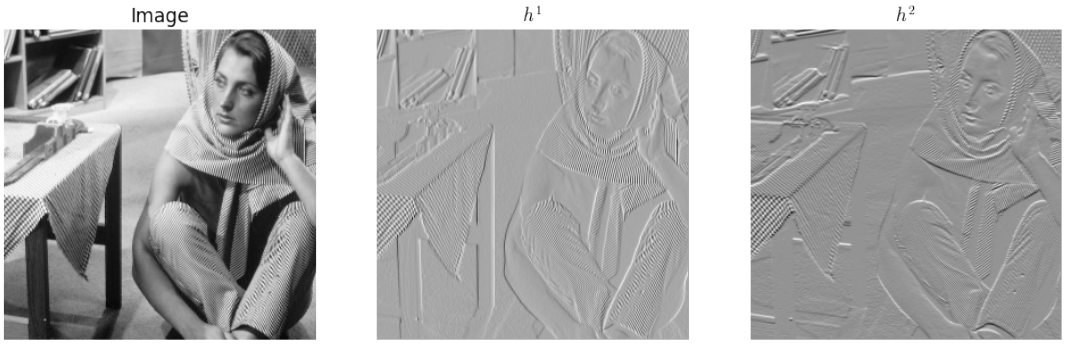



```
In [22]: im = misc.imread('barbara.png').astype('float32')
         response1 = np.zeros(im.shape)
         response2 = np.zeros(im.shape)
         for i in range(im.shape[0]-2):
             for j in range(im.shape[1]-2):
                 patch = im[i:i+3,j:j+3]
```

```
            patchvec = patch.reshape(9,1)
            response1[i,j] = np.dot(filter1vec.T,patchvec)
            response2[i,j] = np.dot(filter2vec.T,patchvec)

    plt.figure(figsize=(12,4))
    plt.subplot(1,3,1)
    plt.imshow(im,cmap='Greys_r')
    plt.title('Image')
    plt.axis('off')
    plt.subplot(1,3,2)
    plt.imshow(response1,cmap='Greys_r')
    plt.axis('off')
    plt.title('$h^1$')
    plt.subplot(1,3,3)
    plt.imshow(response2,cmap='Greys_r')
    plt.axis('off')
    plt.title('$h^2$')
```

```
Out[22]: <matplotlib.text.Text at 0xb33d278>
```



20 Filters and Convolution

We saw matrices

$$W_1 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, W_2 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

and their vector representations

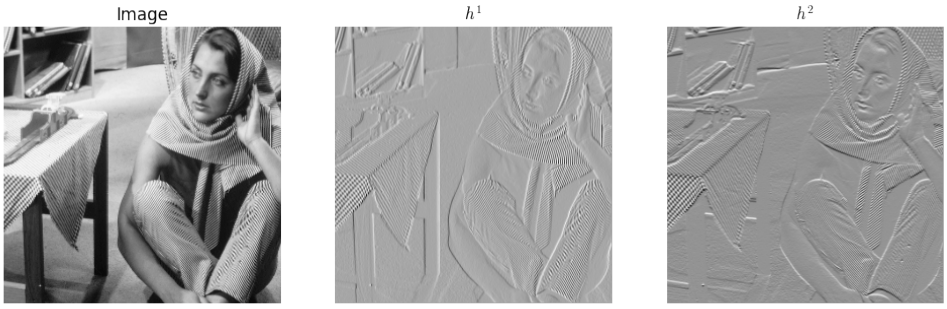
$$\mathbf{w}_1^T = \begin{bmatrix} -1 & 0 & 1 & -1 & 0 & 1 & -1 & 0 & 1 \end{bmatrix} \mathbf{w}_2^T = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

For each location (i, j) in the image, we extracted a 3×3 patches in the image, unrolled it into a vector $\mathbf{x}$ and recorded inner products

$$h_{i,j}^1 = \mathbf{w}_1^T \mathbf{x}$$

$$h^2_{i,j} = \mathbf{w}_2^T \mathbf{x}$$

$\mathbf{w}_1$ is an example of a **filter**. Computation of h^1 is called **2D convolution**.



21 Filters and Convolution

A little bit more formally, discrete (1D) convolution is computed by

$$\text{conv}(\mathbf{y}, \mathbf{x})_i = \sum_j \mathbf{y}_{i-j} \mathbf{x}_j$$

Discrete 2D convolution is computed by

$$\text{conv}(\mathbf{Y}, \mathbf{X})_{i,j} = \sum_k \sum_l \mathbf{y}_{i-k,j-l} \mathbf{x}_{j,k}$$

Note that the formal definition is a little bit different (indices are decreasing in $\mathbf{y}$).

22 Convolution can be made fast

Suppose both $\mathbf{y}$ and $\mathbf{x}$ are n long

$$\text{conv}(\mathbf{y}, \mathbf{x})_i = \sum_j \mathbf{y}_{i-j} \mathbf{x}_j.$$

Naive computation can take $O(N^2)$, since both i and j range from 1 to N.

This can be sped up to $O(N \log(N))$ using discrete fourier transform -- details of this are beyond our course, but the idea is to regress both $\mathbf{y}$ and $\mathbf{x}$ onto an orthonormal basis in which convolution becomes elementwise addition.

But we will look at the speed-up.

```
In [23]: from scipy.signal import convolve2d
         import time

         def slow_convolve(im,w):
             s = w.shape
             l = np.prod(s)
             vec = w.reshape(np.prod(l))
             response = np.zeros((im.shape[0]-s[0]+1,
                                  im.shape[1]-s[1]+1))
             for i in range(im.shape[0]-s[0]+1):
```

```
                for j in range(im.shape[1]-s[1]+1):
                    patch = im[i:i+s[0],j:j+s[1]]
                    patchvec = patch.reshape(l,1)
                    response[i,j] = np.dot(patchvec.T,vec)
            return response

        start = time.time()
        im = misc.imread('barbara.png').astype('float32')

        response1 = slow_convolve(im,filter1)
        end = time.time()
        slow = end - start
        print("naive convolution takes ",slow, " seconds")

        start = time.time()
        response1fast = convolve2d(im,np.flipud(np.fliplr(filter1)),
                                   mode='valid')
        end = time.time()
        fast = end - start

        print("fast (DFT) approach takes ",fast, " seconds")
        print("speed-up factor: ",slow/fast)
        print("difference: ", np.sum(np.sum(np.abs(response1 - response1fast))))
```

```
naive convolution takes  1.01900005341  seconds
fast (DFT) approach takes  0.0160000324249  seconds
speed-up factor:  63.6873742717
difference:  0.0
```

23 Convolutional layers in neural nets

Convolving an image with multiple filters gives responses for each.

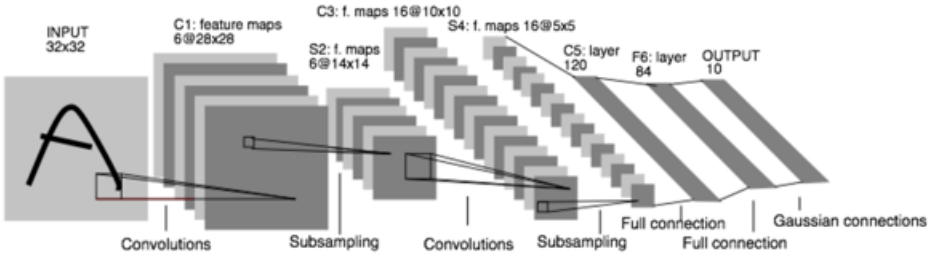
This is typically the first step in deep learning nets.

This example is from 1989! In the illustration 6 filters are applied in the first layer of the network.

24 Ways to go still ...

We still need to talk about training neural networks, more complex layers, how to preprocess data ...

You should remember following: 0. Deep learning computes representations useful for prediction 1. Computation is done using forward propagation 2. Activation functions transform weighted inputs into a real value -- typically nonlinearly 3. Depth is necessary to compute interesting quantities 4. There are quite a few parameters involved


INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections