10 DeepLearning

Our treatment of neural networks in the last chapter was basically state of the art in the 1980 and 1990s. However, neural networks have recently receiving considerable attention under the name of deep learning. Deep learning basically refer to neural networks with many layers, which will of course lead to more complex models and hence it is likely that they can outperform simpler models for complex tasks. While this was already clear in the 1990s, or to Frank Rosenblatt in the 1950s, we have only been able to train such networks in more complex tasks in the last few years. This advancement is due to several factors, including the availability of large supervised datasets, the enormous increase of computational power in particular with the help of graphical processor units (GPUs) that are applicable to matrix operations, and to more methods to restrict the weight space despite many layers. The field of deep learning has developed largely in the domain of image processing with convolutional neural networks, although it is by no means restricted to this area. We start our discussion here.

10.1 Filters and 1d convolution

Let us start by running a simple linear perceptron on a pattern identification problem. This consists of some training vectors with ten features. An example training set with 100 pattern vectors from two classes is shown in Fig. 10.2a. It is It is difficult to make out these training pattern as most of these attributes have been generated randomly with a uniform distribution between 0 and 1 expect the last two attributes which are always one. The difference between the classes can be seen in the plot below. One class was exactly generated as stated above but the second class had feature values in the middle was always set to the pattern 1,0.5,1. The program which generates the training examples and plots them is shown below.

```
import numpy as np
import matplotlib.pyplot as plt

n=100
X=np.random.rand(10,n); X[8:10,:]=1
a=np.array([1.0,0.5,1.0])
a=np.tile(a,(50,1))

X[3:6,:50]=a.T
plt.plot(X,'k'); plt.show()
plt.plot(X[:,:50],'b'); plt.show()
plt.plot(X[:,50:],'r'); plt.show()
```

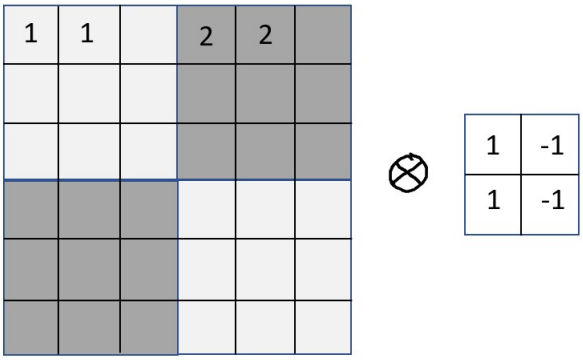

Fig. 10.1 Example of a pattern identification problem. A) the training patterns are mainly noise except for some pattern that have consistently larger values for features 2, 3 and 4. B) The weights of a linear perceptron after training on the patterns on the left. The weights represent a filter for the M-shaped pattern at position 3.

We can train a linear perceptron on these pattern and the result and the learning curve is shown in Fig. **??**a. From this we can see that the network learns rapidly to some degree but can not distinguish all the pattern. This is of course not expected as some of the random pattern of class 1 could have attribute values that resemble closely the pattern of class 2. However, the network gets around 90% correct in this example. The program that trains the linear perceptron is shown below. perceptron

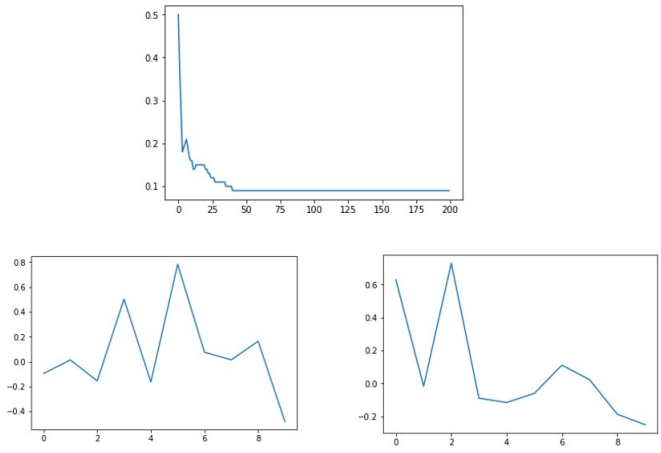

Fig. 10.2 Example of a pattern identification problem. A) the training patterns are mainly noise except for some pattern that have consistently larger values for features 2, 3 and 4. B) The weights of a linear perceptron after training on the patterns on the left. The weights represent a filter for the M-shaped pattern at position 3.

```
Y=np.zeros(n); Y[0:50]=1
```

```
# model specifications
Ni=10; No=1;

#parameter and array initialization
Ntrials=200
wo=np.random.rand(No,Ni)-0.5;
error=np.array([])

for trial in range(Ntrials):
y=wo@X  # prediction for all samples
wo=wo+0.001*(Y-y)@X.T          # weight update
error=np.append(error,np.sum(np.abs(Y-y)>0.5)/n)
plt.plot(wo[0,:]); plt.show()
plt.plot(error); plt.show()
```

It is now interesting to look at the weights to see what this network learned. The weights for the above example are shown in Fig. **??**b. The weight are essentially small except for attribute 3 and attribute 5. This clearly shows that the network tries to suppress all features except the ones that are relevant to distinguish the classes, or with other words the weight matrix represents a filter that loks for large values in attribute 3 and 5. The last attribute is negative to code for the threshold. In Fig. **??**c there is another example shown when the deterministic pattern of class 2 is moved to the beginning of the pattern vector.

We can learn two important point from this example. The first is that weights represent (learned) feature detectors. This is an important way of thinking about these weights, and plotting the weights will be a good way of visualizing some result of training. Of course, the one output node with these specific weights look for a pattern at specific locations. To detect the same pattern at a different location we would have to train a new node with examples of the pattern at this new location. This seems to be a lot of training and a lot of nodes and weights if we only want to know if a pattern is present regardless of the location. In many cases we want to look for patterns that can occur at any places. For example, if we are looking for a coffee mug than this can be placed at many locations. Having feature detectors for specific locations seem to be an inefficient way of doing pattern search.

A better way to do location-invariant pattern recognition is to use convolutions. Let us illustrate the idea before formally defining this operation. For this we will use again the specific M-shaped pattern in a series of data points. Let say that we know that we look for this M-shaped pattern, and to detect this we use a **filter** f which is a short vector describing this pattern, namely $f(0) = 1$, $f(1) = 0.5$, and $f(2) = 1$. To apply this filter we simply multiply this filter with the corresponding signal. Let's say the signal is the vector $\mathbf{x} = (0, 0, 1, 0.5, 1, 0, 0, 0, 0, 0)$. Let's place the filter at the beginning of this signal and let's multiply the components and then add them together. This results in the value of 1. Placing the filter starting at the second position and calculating this sum of the products gives a value of 1.5. After placing the filter at each possible location we get the sequence $(1, 1, 2.5, 1, 1, 0, 0, 0)$. The operation of multiplying the filter with the signal and adding the terms gives us some form of overlap measure between the pattern and the filter. A large value indicates that the

pattern is present and where it is present. The operation of multiplying and adding while shifting the filter is called a **convolution**. The resulting filtered signal represents how much and where a pattern is present in the original signal. For example, if we have a signal with a shiftend location, say $\mathbf{x} = (0, 0, 0, 0, 0, 1, 0.5, 1, 0, 0)$, we get a filtered signal $(0, 0, 0, 1, 1, 2.5, 1, 1)$.

You might have noticed that the filtered signal is smaller than the original signal. Indeed, we are loosing some components in the filtered signal proportional to the size of the filter as the filter has a finite extend and can not be placed with the beginning of the filter at the end of the original signal. There are different ways of handling this. For example, we could just add some zeros to the original signal. This is called zero **padding**. Or we could repeat some of the previous entries such as continuing with the start of the signal when we reach the end. Or we can just accept that the resulting filter signal is slightly smaller. Sometimes we are even interested in getting smaller versions of the signal which somewhat represent a compressed representation. One way of doing this is to not just shift the filter by one step in each iteration of the convolution, but use larger steps such as 2 or more. This is called a **stride**. A stride value of two would half the size of the filtered signal.

Before we move to higher dimensions, let us formalize somewhat our discussion and define the above described convolution for a 1-dimensional discrete signal mathematically as

$$(f * x)(t) = \sum_{t'=0}^{T} f(t')x(t + t'). \tag{10.1}$$

We can also formulate this for a continuous signal. Since we can then not start the signal at zero, as the signal runs formally from $-\infty$ to $+\infty$, it is more common to place the filtered value at the center of the filter

$$(f * x)(t) = \int_{-\infty}^{\infty} f(t')x(t - t')dt'. \tag{10.2}$$

Note that we consider here a filter that is reversed compared to the discrete definition. This is historically the more common way to define the convolution operation. A function like the filters appearing in an integral are mathematically called a kernel.

10.2 2-d convolution and image processing

In the last section we looked at convolutions for 1-dimensional data. It is straight forward to generalize this two 2-dimensional data. Mathematically, a complete n-dimensional convolution could be written as

$$(f * g)(x_1, x_2) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x'_1, x'_2)g(x_1 - x'_1, x_2 - x'_2)dx'_1dx'_2, \tag{10.3}$$

where we used the letter g for the signal, which is now an n dimensional array. Mathematically, this corresponds to a tensor. Note that the difference between an tensor and a n-dimensional array is usually that the mathematical construct of a tensor (or a matrix in 2 dimensions or a vector in 1 dimensions) represent the structure of an array together with some operations such as adding and multiplying such object.

In the following we will illustrate higher dimensional convolutions in a discrete 2-dimensional case, that of processing geyscale images. Let us first make a simple example of a greyscale imgae, say a matrix as shown in Fig. 10.3 where each entry corresponds to a pixel with a grey scale value, although we only used the grey sacale values 1 and 2.

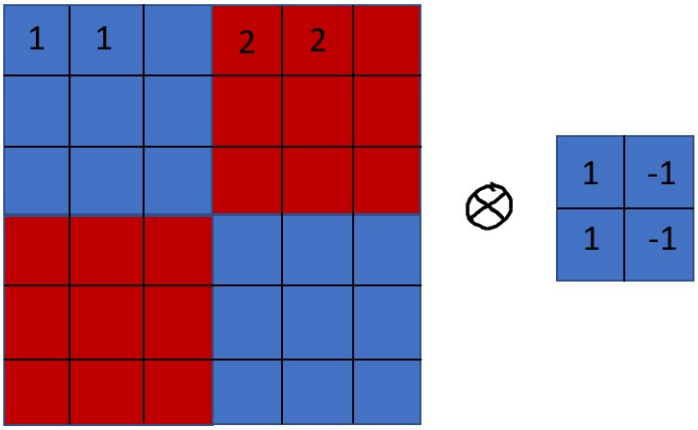

Fig. 10.3 Example of edge detector.

We apply to this image a 2×2 filter with 1s on the left column and -1s on the right column. Applying this filter means first to overlay the filter with the image in the upper left corner, multiplying the overlapping components and adding them all up. This results in the value of zero which we noted in the upper left corner of the resulting filtered image. Next we move the filter one position to the right and perform the same operation of component-wise multiplication and addition, which gives another value of zero. The third shift is more interesting as the pixles on the left and right are different. This filter would result in a value of -2 at this location. We repeat this procedure until we have placed the filter over all possible places in the image.

It is interesting to observe the resulting filtered image. What stands out in this image is a horizontal column in the middle with values different from zero. If we take the absolute of this value then we would get a resulting image with a line that corresponds to an edge in the original image. The filter is therefore a form of an edge detector, specifically for horizontal edges. An example on a real image is shown in Fig. 10.4. The first figure shows a color image in JPG format that we can read and display with the following Python code

```
from pylab import *

img=imread('photo.jpg'); imshow(img)
print('image shape', img.shape)
```

This image has 720*1280 pixels, where the color is represented in three channels. We pick the first channel, which is displaid as the second image, and then apply the

simple edge filter which results in the third image. This filtered image shows mainly horizontal lines. In a similar way we can also design edge detectors for vertical edges.

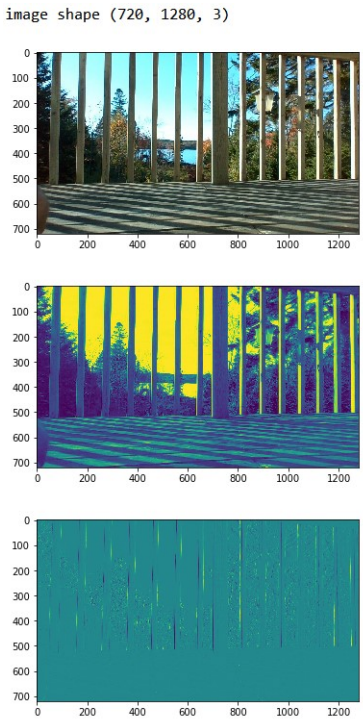

Fig. 10.4 Example of simple edge detector on the first component of a JPG image.

Of course, our filter is very small and only show edges with significant changes between two consecutive edges. There are therefore better designs of edge detectors, such as the Canny edge detector. These techniques combine such gradient filters with Gaussian smoothing and removal of some spurious cases. Also, a continuous version of edge filters is for example described by Gabor functions such as the ones shown in Fig. 10.5a and b. A Gabor function is described by a sinusoidally-modulated Gaussian,

$$f(u,v) = e^{-\frac{u^2+\gamma v^2}{2*\sigma^2}} \cos(\frac{2\pi}{\lambda}u + \varphi). \tag{10.4}$$

The example of a 64^2 pixel filter with parameters $\gamma = 0.5$, $\sigma = 10$, $\lambda = 32$, and $\varphi = \pi/2$ is shown in Fig. 10.5a. This filter can also be rotated with a rotation matrix

$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \cos(\varphi) & \sin(\varphi) \\ -\sin(\varphi) & \cos(\varphi) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{10.5}$$

as shown in Fig. 10.5b for $\varphi = \pi$. Interestingly, such functions describe some of the neurons in the primary visual cortex of primates. Detecting edges seems therefore a good first step to process images, a fact that we will encounter again in later discussion.

Attachments area

A. Gabor function with \phi = \pi/2 B. Rotated version of A

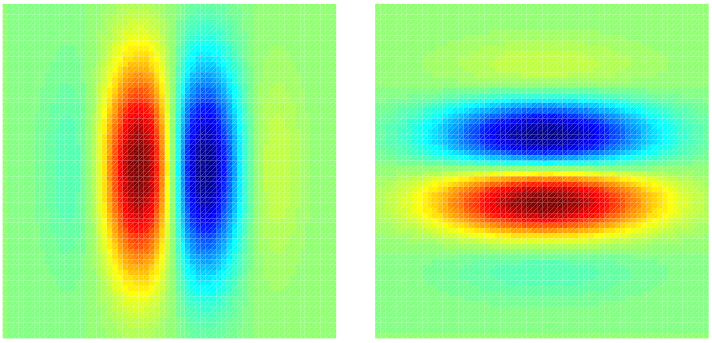

Fig. 10.5 Example of Gabor functions for (a) vertical and (b) horizontal edge detection.

10.3 Convolutions with tensors

We already discussed the case of a 1-dimensional convolution and a 2-dimensional convolution. It is easy to generalize this to n-dimensional data. Mathematically, an n-dimensional convolution could be written as

$$(f * g)(x_1, ..., x_n) = \int_{-\infty}^{\infty} ... \int_{-\infty}^{\infty} f(x'_1, ..., x'_n) g(x_1 - x'_1, ..., x_n - x'_n) dx'_1 ... dx'_n, \tag{10.6}$$

where we used the letter g for the signal, which is now an n-dimensional array, which is mathematically a tensor. For example, a 3-dimensional tensor would be a cube, and convolving them with another 3-dimensional tensor (cube) would result in another 3-dimensional tensor (cube).

However, such higher dimensional operations do not always make sense even though out input data might be a tensor. A prominent example is a color image. A color image can be represented with three 2-dimensional intensity maps, one representing the red components, one the green components, and one the blue components (see Fig. 10.7). This is the basis of the RGB representation of a color image. While these data can be represented by a 3-dimensional tensor, convolving over the color channels does not really make sense since we only have size 3 in this direction. If in some cases with larger dimensions, convolutions might not always be useful. Convolution is only useful when we are looking for local patterns in the relation between neighboring entries in a tensor and only in situations where these pattern are position invariant meaning that they can occur anywhere in this dimension. In the case of color images, it is common to do one separate convolution over each channel but then to add these different convolved channels. Mathematically, averaging over (n-m)channels after applying m-dimensional convolutions of the (n-m) channels is given by

$$(f * g)(x_1, ..., x_m) = \int_{-\infty}^{\infty} ... \int_{-\infty}^{\infty} f(x'_1, ..., x'_n) ... \tag{10.7}$$

$$... g(x_1 - x'_1, ..., x_m - x'_m, x'_{m+1}, ..., x'_n) dx'_1 ... dx'_n \tag{10.8}$$

Thus, if we have one edge filter, say a horizontal edge detector, and apply this to a color image, we get one grey image that averages the lines of the r,g, and b channels of the color image. If we are using another filter, say vertical edge detector, then we would get a second filtered image. The different filtered images can be viewed as **different channels** generated by a **filter bank**. Of course, we can use filter banks that have many more filters in them. This is illustrated in the following figure.

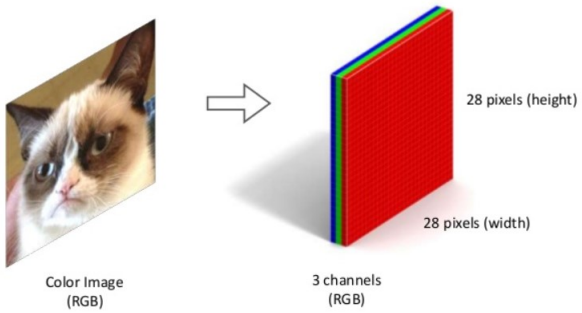

Fig. 10.6 Representation of color images with color channels.

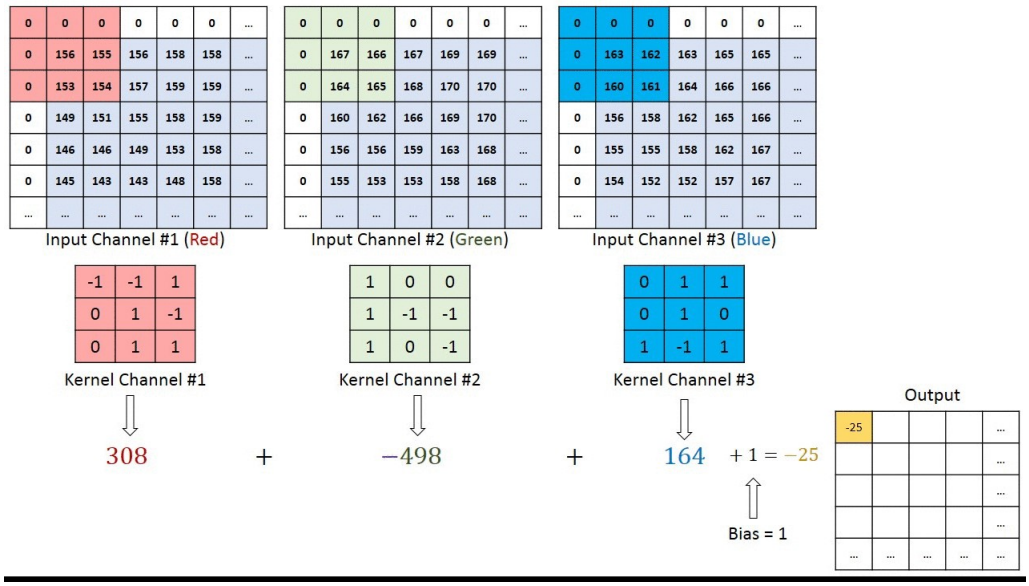

Fig. 10.7 Convolutions with tensors and channels.

10.4 CNN

The idea of designing position invariant feature detectors in neural networks was first described by Fukushima in 1980. Fukushima worked at this time for NHK (the Japanese public broadcaster) together with physiologists as NHK was interested to understand the mechanisms of human vision. It was well known since the early 1960s form the experiments by Hubel and Wiesel that some neurons in the primary visual cortex, the first stage of visual processing in the cortex, are edge detectors as already discussed above. Furthermore, object recognition must be somehow position invariant as we can place objects at different locations in the visual field.

Edge detectors are also the workhorse of traditional computer vision, and we discussed in the first section how such filters are implemented with convolutions. The neural networks that we discussed before had to learn individual weights to each pixel location. Even if this network would learn to represent an edge detector, such detectors have to be learned for each location in an image since edges could usually appear in all locations. So another way of thinking about convolution is that a neuron (specific filter) is applied to every possible location in the image. This leads us to the idea of weight sharing and **convolutional neural networks (CNNs)**.

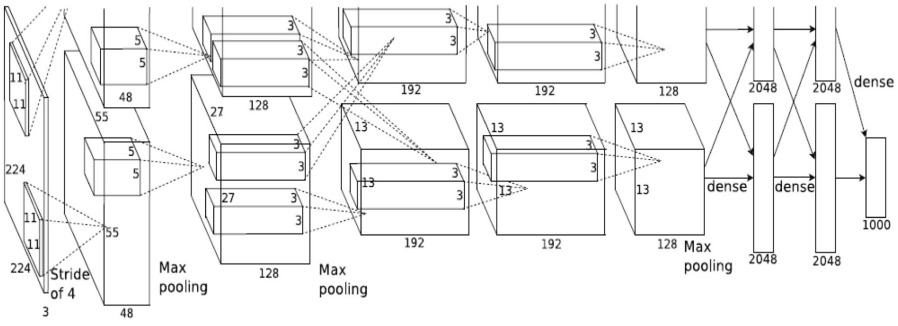

Fig. 10.8 A famous implementation (so called AlexNet) of a convolutional neural network for image classification Representation (Krizhevsky, Sutskever, Hinton, 2012).

A famous example called AlexNet is shown that was used to classify a big database of images from many different classes is shown in Fig. 10.8. This network takes three dimensional images (e.g. RGB values of pixels) and applies a layer of filters to it. This specific network divided the pathways though the network into to stream in order to facilitate the computation with two GPUs, but at this point it is only necessary to look at one stream. In this example, the inputs are of size 224x224 with three channels (RGP). We then perform the following layered computation on this input:

convolution The first level filters are of size 11x11, so that the resulting image is of size 55x55. This layer also consists of 48 different filters so that we end up with 48 filters.

gain function At this stage we are using a gain (or activation) function that is not explicitely shown in the figure but is sometimes indicated as separate computing step. It is common to use the RELU activation function for reasons discussed later.

pooling If we would only apply convolutions on convolutions, then we would end with filtered images of filtered images. However, what we really hope to achieve are high level representation such as nodes that represent class labels. Such labels are likely not to depend on individual pixels and rather represent a highly compressed summary of an image. To help with this we add **pooling layers** after each convolution layer. A pooling operation is usually just taking the average or the maximum of the responses in a certain area of the filtered image, thus compressing the images down by only considering the average or maximal feature represented by the filter in this area.

These three steps are the typical components of a convolutional layer of a CNN, and these steps are repeated to build a deep convolutional network. At the end we use a **fully connected layer (MLP)** to gather all the information and make the final classification based on the features extracted by the network.

While we have just outlined the operation of this network, an important part of using this network is of course the training. Indeed, for our further discussion it is good to realize that the filters are not chosen by hand but are learned from examples. This training was trained with the back propagation algorithm. This is fairly straight forward except when back propagating though the pooling layers. One approach is thereby to give all credit for the error (average pooling), or just change the winning unit (max pooling).

10.5 Tensorflow

To implement deep networks we use the Tensorflow package from Google. At this time you should follow the tensorfow tutorial

`https://www.tensorflow.org/get_started/get_started`

where some of the structure of Tensorflow is described. The please follow the MNIST tutorial

`https://www.tensorflow.org/get_started/mnist/beginners`

5

10.6 Representational learning and compression

Above we pointed out that we used back-propagation (gradient descent learning) to learn all the filters in the machines and hence the specific representations in each of the layers of the deep network. This is a great advantage over more traditional system in which filters had to be designed carefully by hand, which is not only time consuming, but also might be less optimal than learned filters. This evolution in our field is outlined in Fig. 10.9.

To illustrate again the re-representation of a signal with filters, consider basic signal analysis. We have time varying signal that is represented with floating point values for each time step. Say we are sampling with 500HZ, typical for EEG, that is, we have one data point every 0.002 second. If we assume that a floating point is typically represented by computer word of 64bits, then a 10 minute length would take over 2MB of storage.

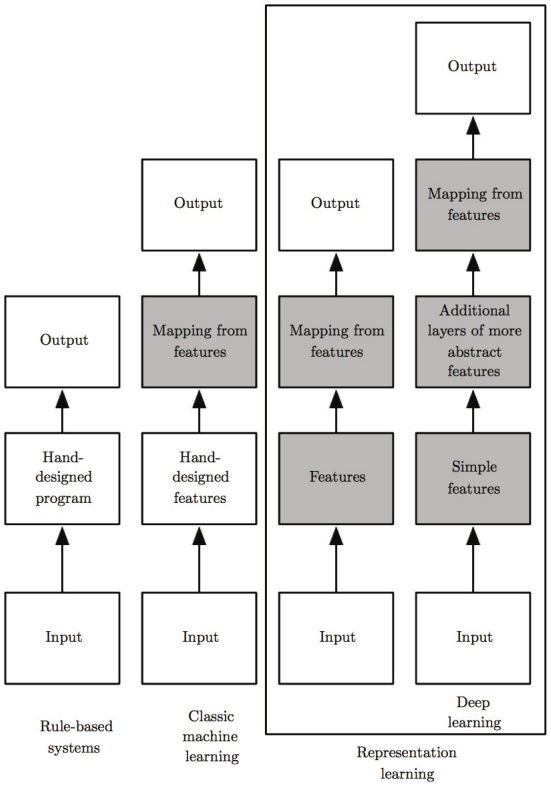

Fig. 10.9 Evolution of machine learning system (from Goodfellow, Bengio, Courville 2015).

An example signal,$x(t)$ is illustrate in Fig. 10.10. While this is a relative complicated signal, we have also created two template signals, which we could also call a **basis function**, with which we can reconstruct the signal as

$$x(t) = a_1 y_1(t) + a_2 y_2(t) \tag{10.9}$$

Representing the original signal with these basis functions has a big advantage. For example, if we all know about the basis functions, than I could transmit the signal with only two computer words for the coefficients $a_1 = 2$ and $a_2 = 3$ (arbitrarily chosen numbers), which takes 16Bytes. Of course, we can not expect real signals to be made up of only these two basis functions. So in practice we want to create a large list of "appropriate" basis functions. The filters in deep neural networks represent such basis functions, and the appropriateness should come from the fact that they are learned from the examples.

Now if we have a large list of basis functions, then we still might need to submit a large number of coefficients. Indeed, if we make the basis function a value at each time step, then we would just end up with the same representation as before. However, a very important insight to make efficient use of resources while extracting the "essence of signals" is to try and find sparse representations. A **sparse representation** is a

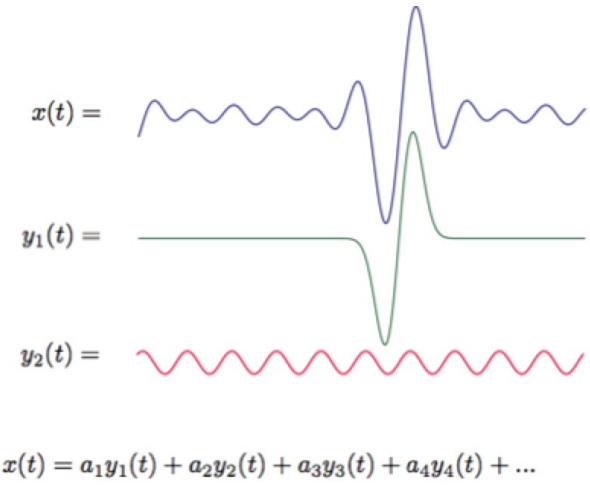

Fig. 10.10 Illustration of signal representation with templates.

representation where I might have a large number of basis functions in my dictionary (nodes in a deep network) but are only using a relatively small number of active nodes to represent each example. In our example this might correspond to

$$x(t) = a_1y_1(t) + a_2y_2(t) + a_3y_3(t) + a_4y_4(t) + a_5y_5(t) + a_6y_6(t) + ... \quad (10.10)$$

with a coefficient vector

$$\mathbf{a} = (2, 3, 0, 0, 0, 0, 0, ...). \quad (10.11)$$

Sparse representations do lead to considerable compressions, and I believe that compression like $\mathbf{a} = (2, 3)$, and particular sparse compression, is an essential ingredients of machines to gain some "semantic knowledge" of the world. For example, there are many instantiations of a tables, but my ability to characterize them with one word is equivalent to semantic knowledge and a form of sparse representation if we take as our communication dictionary a large list of names for furnitures.

10.7 Denoising autoencoders

If compressed (or sparse) representations are so useful, can we force such representations in our deep networks? There are different techniques that will indeed force some compressed representations including the architecture outlined here as well as more general regularization methods discussed in the next section.

A simple example of an **autoencoder** is shown in Fig. 10.11. In this network we start with an input layer that is connected to a smaller hidden layer and then to an output layer that is the same size as the input layer. The reason for choosing an output layer that has the same size as the input layer is that we want to build a mapping function that maps inputs to the same output. This is actually an example of **unsupervised learning** as we do not require labels just raw data such as pictures.

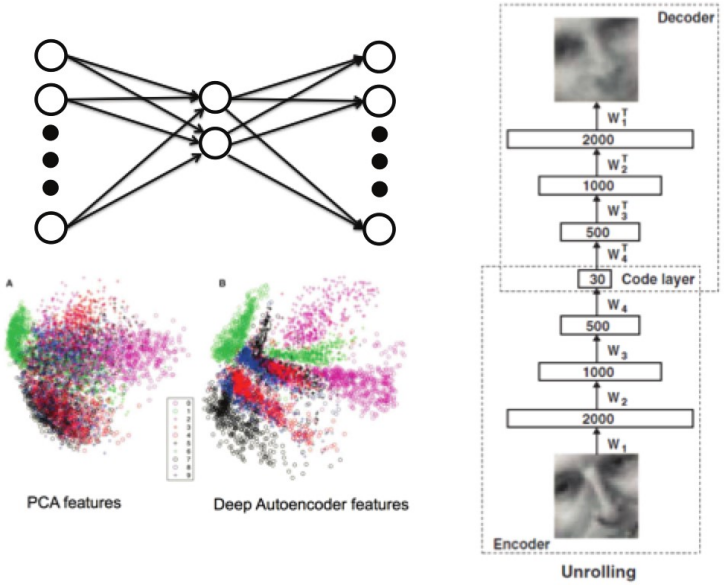

Fig. 10.11 Examples of autoencoders with the basic idea on top, the deep autoecncoder that kicked of deep learning by Hinton and Salakhutdinov 2006, and some comparisons of auto encoders to PCA.

Why would we want to build such functions and isn't this simply the identity function? It is not as we channel the input through a small layer that forces some compression. In this sense we try to extract useful filters, and some people have described this as **semantic hashing**. Most of the implementation use a version were the inputs are somewhat corrupted by noise and the labels are the noiseless pattern, which are called **denoisong autoencoders**. This will also help to force the solution away from the identity function and is an example of noise induction as a regularization technique as discussed further below.