Explaining neural networks in raw Python: lectures in Jupyter

Wojciech Broniowski

Jul 06, 2021

CONTENTS

Wojciech Broniowski

Institute of Nuclear Physics PAN, Kraków, and

Jan Kochanowski University, Kielce, Poland

These lectures were originally given to students of computer engineering at the Jan Kochanowski University in Kielce, Poland, and for the Kraków School of Interdisciplinary PhD Studies. They explain the very basic concepts of neural networks at a level requiring only very rudimentary knowledge of Python, or actually any programming language. With simplicity in mind, the code for various algorithms of neural networks is written from absolute scratch, i.e. without any use of dedicated higher-level libraries. That way one can follow all the programming steps in an explicit manner.

Note: Built with Jupyter Book 2.0 tool set, as part of the ExecutableBookProject.

Important: The reader may download the complete code in the form of jupyter notebooks as well as some supplementary material from …

ISBN: **978-83-962099-0-0**

9 788396 209900

CHAPTER

ONE

INTRODUCTION

1.1 Purpose of these lectures

The purpose of this course is to teach some basics of the omnipresent neural networks with Python. Both the explanations of key concepts of neural networks and the illustrative programs are kept at a very elementary "high-school" level. The code, made very simple, is described in detail. Moreover, is is written without any use of higher-level libraries for neural networks, which brings in better understanding of the explaned algorithms and shows how to program them from scratch.

Important: The reader may thus be a complete novice, only slightly acquainted with Python (or actually any other programming language) and jupyter.

The material covers such classic topics as the perceptron, supervised learning with back-propagation and data classification, unsupervised learning and clusterization, the Kohonen self-organizing networks, and the Hopfield networks with feedback. This aims to prepare the necessary ground for the recent and timely advancements (not covered here) in neural networks, such as deep learning, convolutional networks, recurrent networks, generative adversarial networks, reinforcement learning, etc.

On the way of the course, some basic Python programing will be gently sneaked in for the newcomers.

Exercises

On-the-fly exercises are included, with the goal to familiarize the reader with the covered topics. Most of exercises involve simple modifications/extensions of appropriate pieces of the lecture code.

There are countless textbooks and lecture notes on the material of the course. With simplicity as guidance, our choice of topics took inspiration from the lectures by Daniel Kersten and from the on-line book by Raul Rojas. Further references include… test [Gut16].

1.2 Biological inspiration

Inspiration for computational mathematical models discussed in this course originates from the biological structure of our neural system. The central nervous system (the brain) contains a huge number ($\sim 10^{11}$) of neurons, which may viewed as tiny elementary processor units. They receive signal via the dendrites, and in case it is strong enough, the nucleus decides (a computation done here!) to "fire" an output signal along the axon, where it is subsequently passed via axon terminals to dendrites of other neurons. The axon-dendrite connections (the synaptic connections) may be weak or strong, modifying the stimulus. Moreover, the strength of the synaptic connections may change in time (Hebbian rule tels that the connections get stronger if they are being used repeatedly). In this sense, the neuron is "programmable".

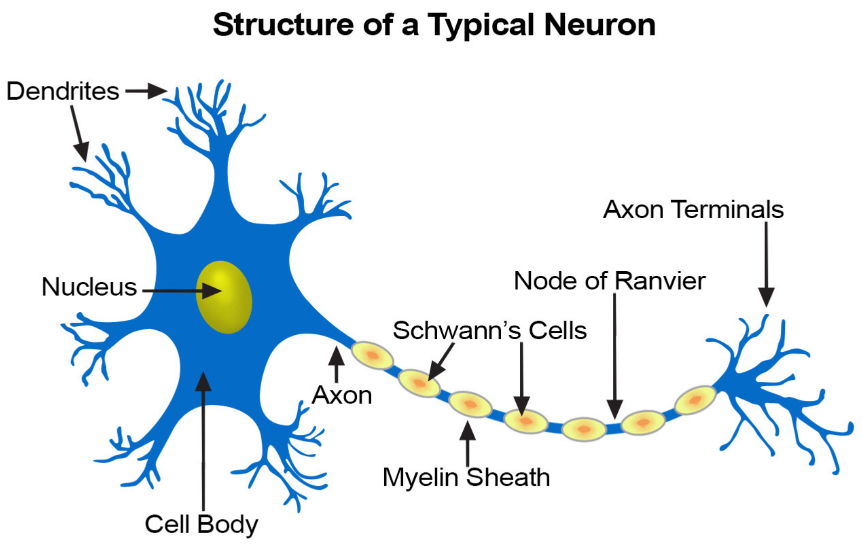


Fig. 1.1: Biological neuron (from here).

We may ask ourselves if the number of neurons in the brain should really be labeled so "huge" as usually claimed. Let us compare it to the computing devices which memory chips. The number of 10^{11} neurons roughly correspond to the number of transistors in a 10GB memory chip, which does not impress us so much, as these days we may buy such chips for 2$ or so.

Also, the speed of traveling of the nerve impulses, which is due to electrochemical processes, is not impresive, either. Fastest signals, such as those related to muscle positioning, travel at speeds up to 120m/s (the myelin sheaths are essential to achieve them). The touch signals reach about 80m/s, whereas pain is transmitted only at comparatively very slow speeds of 0.6m/s. This is the reason why when you drop a hammer on your toe, you sense it immediately, but the pain reaches your brain with a delay of ~1s, as it has to pass the distance of ~1.5m. On the other hand, in electronic devices the signal travels at the speed of light, $\sim 300000\text{km/s} = 3 \times 10^8\text{m/s}$!

For humans, the average reaction time is 0.25s to a visual stimulus, 0.17s for an audio stimulus, and 0.15s for a touch. Thus setting the threshold time for a false start in sprints at 0.1s is safely below a possible reaction of a runner. These are very slow reactions compared to electronic responses.

Based on the energy consumption of the brain, one can estimate that on the average a cortical neuron fires about once per 6 seconds. Likewise, it is unlikely that an average cortical neuron fires more than once per second. Multplying the above firing rate by the number of all the cortical neurons, $\sim 1.6 \times 10^{10}$, yields about 3×10^9 firings/s in the cortex, or 3GHz. This is the rate of a typical processor chip! So if a firing is identified with an elementary calculation, the combined power of the brain is comparable to that of standart computer processor.

The above facts indicate that, from a point of view of naive comparisons with silicon-based chips, the human brain is nothing so special. So what is it that gives us our unique abilities: amazing visual and audio pattern recognition, thinking, consciousness, intuition, imagination? The answer is linked to an amazing architecture of the brain, where each neuron (processor unit) is connected via synapses to, on the average, 10000 (!) other neurons (cf. Fig. Fig. 1.2). This feature makes it radically different and immensely more complicated than the architecture consisting of the control unit, processor, and memory in our computers (the von Neumann machine). There, the number of connections is of the order of the number of bits of memory. In contrast, there are about 10^{15} synaptic connections in the human brain. As mentioned, the connections may be "programmed" to get stronger or weaker. If, for he sake of a simple estimate, we approximated the connection strength by just two states of a synapse, 0 or 1, the total number of combinatorial configurations of such a system would be $2^{10^{15}}$ - a humongous number. Most of such confiburation, of course, never realize in practice, nevertheless the number of possible configuration states of the brain, or the "programs" it can run, is immense.

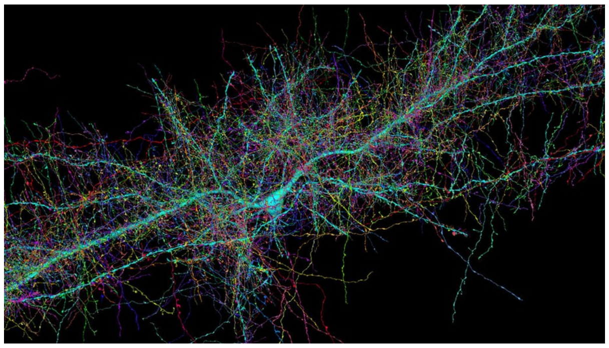


Fig. 1.2: A small brain sample with axons clearly visible (from here)

In recent years, with powerful imaging techniques, it became possible to map the connections in the brain with unprecedented resolution, where single nerve boundles are visible. The efforts are part of the Human Connectome Project, with the ultimate goal to map one-to-one the human brain architecture. For the fruit fly the drosophila connectome project is well advanced.

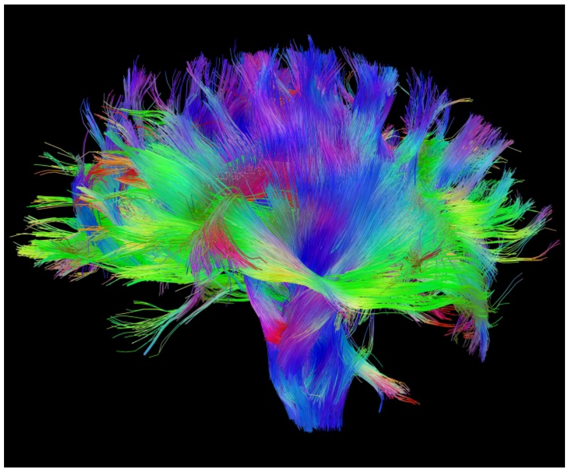


Fig. 1.3: White matter fiber architecture of the brain (from Human Connectome Project)

1.3 Feed-forward networks

The neurophysiological research of the brain provides important guidelines for mathematical models used in artificial neural networks (ANNs). Conversely, the advances in algorithmics of ANNs frequently bring us closer to understanding of how our brain computer may actually work!

The simplest ANNs are the so called **feed forward** networks, depicted in Fig. Fig. 1.4. They consist of the **input** layer (black dots), which just represents digitized data, and layers of neurons (blobs). The number of neurons in each layer may be different. The complexity of the network and the tasks it may accomplish increases with the number of layers and the number of neurons in the layers. Networks with one layer of neurons are called **single-layer** networks. The last layer (light blue blobs) is called the **output layer**. In multi-layer networks the layers preceding the output layer (purple blobs) are called **intermediate layers**. If the number of layers is large (e.g. as many as 64, 128, …), we deal with **deep networks**.

The neurons in various layers do not have work the same way, in particular the output neurons may act differently from the others.

The signal from the input travels along the links (edges, synaptic connections) to the neurons in subsequent layers. In feed-forward networks it can only move forward. No going back to preceding layers or propagation among the neurons of the same layer are allowed (that would be the **recurrent** feature). As we will describe in detail in the next chapter, the signal is appropriately processeded by the neurons.

In the sample network of Fig. Fig. 1.4 each neuron from a preceding layer is connected to each neuron in the following layer. Such ANNs are called **fully connected**.

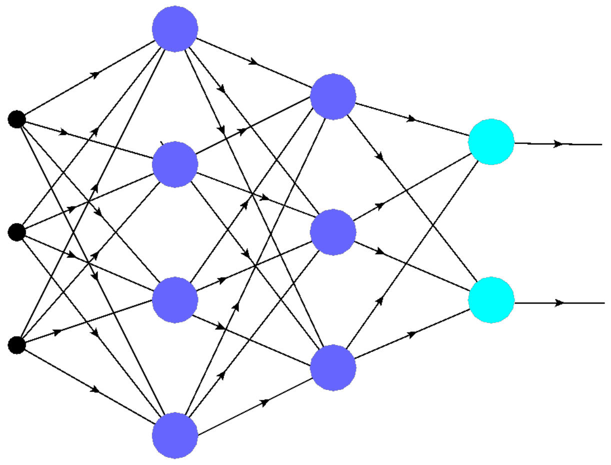


Fig. 1.4: A sample feed-foward fully connected artificial neural network. The blobs represent the neurons, and the edges indicate the synaptic connections between them. The signal propagates starting from the input (black dots), via the neurons in subsequent intermediate (hidden) layers (purple blobs) and the output layer (light blue blobs), to finally end up as the output (black dots). The strength of the connections is controled by weights (hyperparameters) assigned to the edges.

As we will learn in the following, each edge in the network has strength described with a number called **weight** (the weights are also termed **hyperparameters**). Even very small fully connected networks, such as the one of Fig. 1.4, have very many connections (here 30), hence carry a lot of parameters. Thus, while looking inoccuously, they are in fact complex multiparametric systems.

Also, a crucial feature here is an inherent nonlinearity of the neuron responses, as we discuss in chapter *MCP Neuron*

1.4 Why Python

The choice of Python for the little codes of this course needs almost no explanation. Let us only quote Tim Peters:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

According to SlashData, there are now over 10 million developers in the world who code using Python, just second after JavaScript (~14 million).

1.4.1 Imported packages

Throughout the course we use some standard Python libraries for the numerics and plotting (as stressed, we do not use any libraries dedicated to neural networks).

```
import numpy as np

# plots
import matplotlib.pyplot as plt

# display imported graphics
from IPython.display import display, Image, HTML
```

Important: Functions created in the course which are of repeated use, are placed in library **neural**, described in the *Appendix*.

Note: For brevity of the presenttion, some redundant or inessential pieces of code are present only in the source jupter notebooks, and are not included in the book. To repeat the calculations one-to-one, the reader should work with downloaded jupyter notebooks, which can be obtained from …

CHAPTER

TWO

MCP NEURON

2.1 Definition

We need the basic buiding block of the ANN: the (artificial) neuron. The first mathematical model dates back to Warren McCulloch and Walter Pitts (MCP), who proposed it in 1942, hence at the very beginning of the electronic computer age during World War II. The MCP neuron depicted in Fig. 2.1 in the basic ingredient of all ANNs and is built on very simple general rules, inspired neatly by the biological neuron:

- The signal enters the nucleus via dendrites from the previous layer.
- The synaptic connection for each dendrite may have a different (and adjustable) strength.
- In the nucleus, the signal from all the dendrites is combined (summed up) into s.
- If the combined signal is stronger than a given threshold, then the neuron fires along the axon, in the opposice case it remains still.
- In the siplest relization, the strenth of the fired signal has two possible levels: on or off, i.e. 1 or 0. No intermediate values are needed.
- Axon terminal connect to dendrites of the neurons in the next layer.

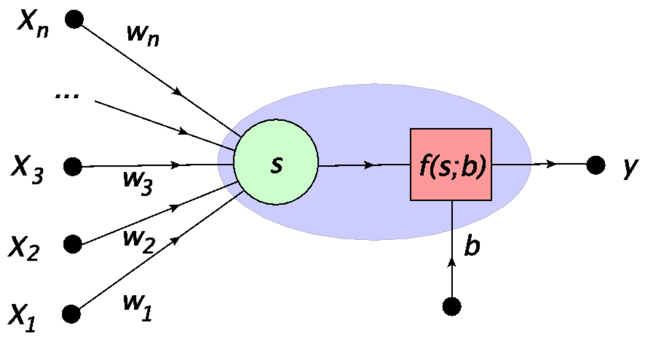


Fig. 2.1: MCP neuron: x_i are the inputs (different in each instance of the data), w_i are the weights, s is the signal, b is the bias, and $f(s;b)$ represents the acitvation function, yielding the output $y = f(s;b)$. The blue oval encircles the whole neuron, as used in Fig. 1.4.

Translating this into a mathematical prescription, one assigns to the input cells the numbers $x_1, x_2 \ldots, x_n$ (input data). The strength of the synaptic connections is controled with the **weights** w_i. Then the combined signal is defined as the weighted sum

$$s = \sum_{i=1}^{n} x_i w_i.$$

Thesignal becomes an argument of the **activation function**, which to begin takes the simple form of the step function

$$f(s;b) = \begin{cases} 1 \text{ for } s \geq b \\ 0 \text{ for } s < b \end{cases}$$

When the combined signal s is larger than the bias (threshold) b, the nucleus fires. i.e. the signal passed along the axon is 1. in the opposite case, the generated signal value is 0 (no firing). This is precisely what we need to mimick the biological prototype.

There is a convenient notational covnention which is frequently used. Instead of splitting the bias from the input data, we may treat it uniformly. The condition for firing may be triviallly transformed as

$$s \geq b \to s - b \geq 0 \to \sum_{i=1}^{n} x_i w_i - b \geq 0 \to \sum_{i=1}^{n} x_i w_i + x_0 w_0 \geq 0 \to \sum_{i=0}^{n} x_i w_i \geq 0,$$

where $x_0 = 1$ and $w_0 = -b$. In other words, we may treat the bias as a weight on the edge connected to an additional cell with input set to 1. This notation is shown in Fig. 2.2. Now, the activation function is simply

$$f(s) = \begin{cases} 1 \text{ for } s \geq 0 \\ 0 \text{ for } s < 0 \end{cases}, \tag{2.1}$$

with the summation index in s starting from 0:

$$s = \sum_{i=0}^{n} x_i w_i = x_0 w_0 + x_1 w_1 + \cdots + x_n w_n. \tag{2.2}$$

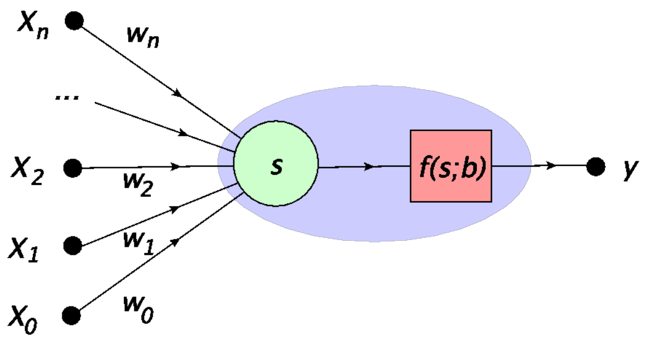


Fig. 2.2: Alternative, more uniform representation of the MCP neuron, with $x_0 = 1$ and $w_0 = -b$.

Hyperparameters

The weights $w_0 = -b, w_1, \ldots, w_n$ are referred to as hyperparameters. They determine the functionality of the MCP neuron and may be changed during the learning (training) process (see the following). However, they are kept fixed when using the trained neuron on a particular input data set.

Important: An essential property of neurons in ANNs is the **nonlinearity** of the activation function. Without this feature, the MCP neuron would simply represent a scalar product, and the feed-forward networks would involve trivial matrix multilications.

2.2 MCP neuron in Python

We will now implement the mathematical model of the neuron of Sec. *MCP Neuron*. First, we obviously need arrays (vectors), which in Python are represented as

```
x = [1,3,7]
w = [1,1,2.5]
```

and are indexed starting from 0, e.g.

```
x[0]
```

```
1
```

The numpy library functions carry the prefix **np**, which is the alias given at import. Note that these fumctions act *distributively* over arrays, e.g.

```
np.sin(x)
```

```
array([0.84147098, 0.14112001, 0.6569866 ])
```

which is a convenient feature. We also have the scalar product $x \cdot w = \sum_i x_i w_i$ handy, which we can use to build the combined signal s.

```
np.dot(x,w)
```

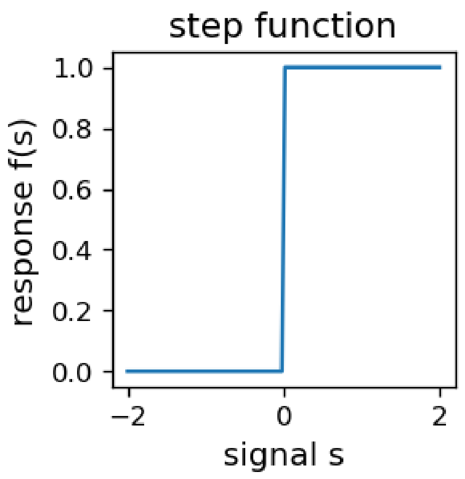
```
21.5
```

Next, we need to construct the neuron activation function, which presently is just the step function (2.1).

```
def step(s): # step function (in neural library)
     if s > 0:
        return 1
     else:
        return 0
```

where in the comment we indicate, that the function is also defined in the **neural** library, cf. *Appendix*. For the visualizers, the plot of the step function is following:

```
plt.figure(figsize=(2.3,2.3),dpi=120) # set the size of the figure

s = np.linspace(-2, 2, 100)    # array 100+1 equally spaced points between -2 and 2
fs = [step(z) for z in s]      # corresponding array of function values

plt.xlabel('signal s',fontsize=12)       # axes labels
plt.ylabel('response f(s)',fontsize=12)
plt.title('step function',fontsize=13)   # plot title

plt.plot(s, fs);
```



Since $x_0 = 1$ always, we do not want to explicitly carry this in the argument of the functions that will follow. We will be inserting $x_0 = 1$ into the input, for instance:

```
x=[5,7]
np.insert(x,0,1) # insert 1 in x at position 0
```

```
array([1, 5, 7])
```

Now we are ready to construct the *MCP neuron*:

```
def neuron(x,w,f=step): # (in the neural library)
    """
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=weighted sum w0 + x1 w1 + x2 w2 +...+ xn wn = x.w
    """
    return f(np.dot(np.insert(x,0,1),w)) # insert x0=1, signal s=x.w, output f(s)
```

We diligently put the comments in triple quotes to be able to get the help, when needed:

```
help(neuron)
```

```
Help on function neuron in module __main__:

neuron(x, w, f=<function step at 0x7fb008a4e320>)
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=weighted sum w0 + x1 w1 + x2 w2 +...+ xn wn = x.w
```

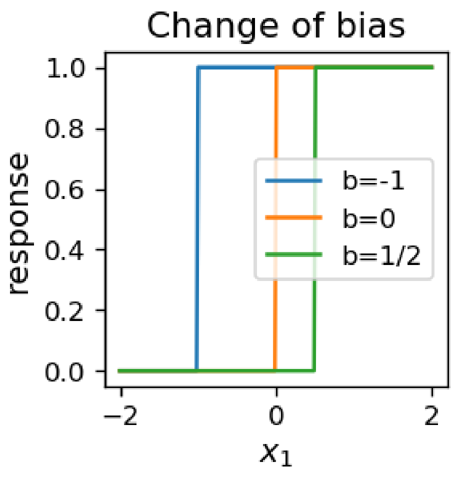
Note that the function f is an argument of neuron, but it has the default set to step and thus does not have to be present. The sample usage with $x_1 = 3$, $w_0 = -b = -2$, $w_1 = 1$ is

```
neuron([3],[-2,1])
```

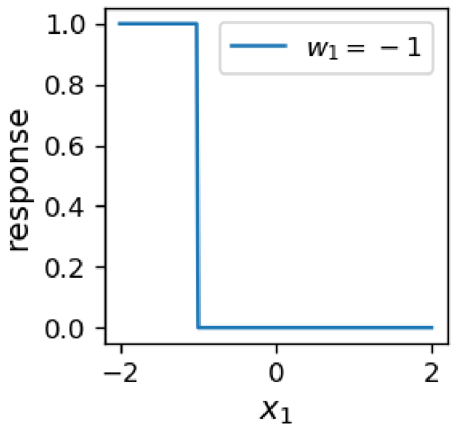
```
1
```

As we can see, the neuron fired in this case, as $s = 1 * (-2) + 3 * 1 > 0$. Next, we show how the neuron operates on a varying input x_1 taken in the range $[-2, 2]$. We also change the bias parameter, to illustrate its role. It is clear that the bias works as the threshold: if the signal $x_1 w_1$ is above $b = -x_0$, the neuron fires.

```
plt.figure(figsize=(2.3,2.3),dpi=120)

s = np.linspace(-2, 2, 200)
fs1 = [neuron([x1],[1,1]) for x1 in s]      # more function on one plot
fs0 = [neuron([x1],[0,1]) for x1 in s]
fsm12 = [neuron([x1],[-1/2,1]) for x1 in s]

plt.xlabel('$x_1$',fontsize=12)
plt.ylabel('response',fontsize=12)

plt.title("Change of bias",fontsize=13)

plt.plot(s, fs1, label='b=-1')
plt.plot(s, fs0, label='b=0')
plt.plot(s, fsm12, label='b=1/2')
plt.legend();                               # legend
```



When the sign of the weight w_1 is negative, we get in some sense a **reverse** behavior, where the neuron fires when $x_1|w_1| < w_0$:



Note that here (and similarly in other places) the trivial code for the above output is hidden and can be found in the corresponding jupyter notebook.

Admittedly, in the last example one departs from the biological pattern, as negative weights are not possible to realize in a biological neuron. However, this enriches the mathematical model, which one is free to use without constraints.

2.3 Boolean functions

Having constructed the MCP neuron in Python, the question is: *What is the simplest (but still non-trivial) application we can use it for?* We will show here that one can easily construct boolean functions, or logical networks, with the help of networks of MCP neurons. Boolean functions, by definition, have arguments and values in the set $\{0, 1\}$, or {True, False}.

To warm up, let us start with some guesswork, where we take the neuron with the weights $w = [w_0, w_1, w_2] = [-1, 0.6, 0.6]$ (why not). We shall here denote $x_1 = p$, $x_2 = q$, in accordance with the traditional notation for logical variables, where $p, q \in \{0, 1\}$.

```
print("p q n(p,q)") # print the header
print()

for p in [0,1]:       # loop over p
    for q in [0,1]:   # loop over q
        print(p,q,"",neuron([p,q],[-1,.6,.6])) # print all cases
```

```
p q n(p,q)

0 0  0
0 1  0
1 0  0
1 1  1
```

We immediately recognize in the above output the logical table for the conjunction, $n(p, q) = p \wedge q$, or the logical **AND** operation. It is clear how the neuron works. The condition for the firing $n(p, q) = 1$ is $-1+p*0.6+q*0.6 \geq 0$, and it is satisfied if and only if $p = q = 1$, which is the definition of the logical conjunction. Of course, we could use here 0.7 instead of 0.6, or in general w_1 and w_2 such that $w_1 < 1, w_2 < 1, w_1 + w_2 \geq 1$. In the electronics terminology, we can call the present system the **AND gate**.

We can thus define the short-hand

```
def neurAND(p,q): return neuron([p,q],[-1,.6,.6])
```

Quite similarly, we may define other boolean functions (or logical gates) of two variables. In particular, the NAND gate (the negation of conjunction) and the OR gate (alternative) are realized with the following MCP neurons:

```
def neurNAND(p,q): return neuron([p,q],[1,-0.6,-0.6])
def neurOR(p,q):   return neuron([p,q],[-1,1.2,1.2])
```

They correspond to the logical tables

```
print("p q  NAND OR") # print the header
print()

for p in [0,1]:
    for q in [0,1]:
        print(p,q," ",neurNAND(p,q)," ",neurOR(p,q))
```

```
p q  NAND OR

0 0   1   0
0 1   1   1
1 0   1   1
1 1   0   1
```

2.3.1 Problem with XOR

The XOR gate, or the **exclusive alternative**, is defined with the following logical table:

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

This is one of possible boolean functions of two arguments (in total, we have 16 different functions of this kind, why?). We could now try very hard to adjust the weights in our neuron to behave as the XOR gate, but we are doomed to fail. Here is the reson:

From the first row of the above table it follows that for the input 0 0 the neuron should not fire. Hence

$w_0 + 0 * w_1 + 0 * w_2 < 0$, or $-w_0 > 0$.

For the cases of rows 2 and 3 the neuron must fire, therefore

$w_0 + w_2 \geq 0$ and $w_0 + w_1 \geq 0$.

Adding side-by-side the three obtained inequalities we get $w_0 + w_1 + w_2 > 0$. However, the fourth row yields $w_0 + w_1 + w_2 < 0$ (no firing), so we encounter a contradiction. Therefore no choice of w_0, w_1, w_2 exists to do the job!

Important: A single MCP neuron cannot represent the **XOR** gate.

2.3.2 XOR from composition of AND, NAND and OR

One can solve the XOR problem by composing three MCP neurons, for instance

```
def neurXOR(p,q): return neurAND(neurNAND(p,q),neurOR(p,q))
```

```
print("p q XOR") # print the header
print()

for p in [0,1]:
    for q in [0,1]:
        print(p,q,"",neurXOR(p,q))
```

```
p q XOR

0 0  0
0 1  1
1 0  1
1 1  0
```

The above construction corresponds to the simple network of Fig. 2.3.

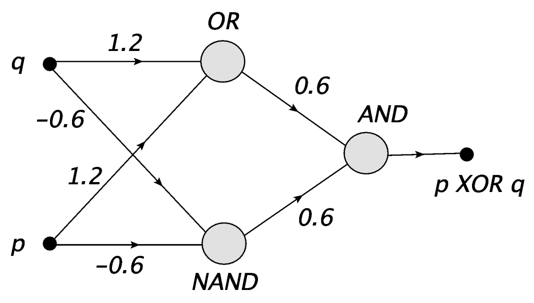


Fig. 2.3: The XOR gate compsed of the NAND, OR, and AND MCP neurons.

Note that we are dealing here, for the first time, with a network having an intermediate layer, consisting of the NAND and OR neurons. This layer is indispensable to construct the XOR gate.

2.3.3 XOR composed from NAND

Within the theory of logical networks, one proves that any network (or boolean function) can be composed of only NAND gates, or only the NOR gates. One says that the NAND (or NOR) gates are **complete**. In particular, the XOR gate can be constructed as

[p NAND (p NAND q)] NAND [q NAND (p NAND q)],

which we can write in Python as

```
def nXOR(i,j): return neurNAND(neurNAND(i,neurNAND(i,j)),neurNAND(j,neurNAND(i,j)))
```

```
print("p q XOR") # print the header
print()

for i in [0,1]:
    for j in [0,1]:
        print(i,j,"",nXOR(i,j))
```

```
p q  XOR

0 0   0
0 1   1
1 0   1
1 1   0
```

Exercises

Construct (all in Python)

- gates NOT, NOR
- gates OR, AND, NOT by composing gates NAND https://en.wikipedia.org/wiki/NAND_logic
- the half adder and full adder https://en.wikipedia.org/wiki/Adder_(electronics)

as networks of MCP neurons.

CHAPTER

THREE

MODELS OF MEMORY

3.1 Heteroassociative memory

3.1.1 Pair associations

We now pass to further illustrations of elementary capabilities of ANNs, desribing two very simple models of memory based on linear algebra, supplemented with (nonlinear) filtering (an implementation of these models in Mathematica is provided in http://vision.psych.umn.edu/users/kersten/kersten-lab/courses/Psy5038WF2014/IntroNeuralSyllabus.html). Speaking of memory here, we have very simple tools in mind, which is far from the actual complex and hitherto not completely understood memory mechanism operating in our brain.

The first model concerns the so called **heterassociative** memory, where some objects (here graphic bitmap symbols) are joined in pairs. In particular, we take the set of five graphical symbols, $\{A, a, I, i, Y\}$, and define two pair associations A $\leftrightarrow$ a and I $\leftrightarrow$ i between different (hetero) symbols. Y remain unassociated.

The symbols are defined as 2-dimensional 12×12 pixel arrays, for instance

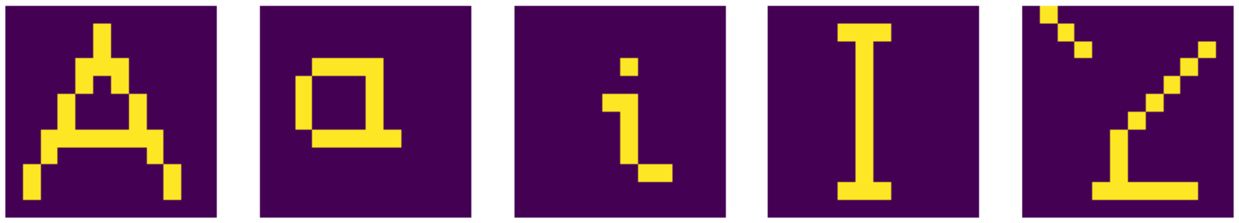
```
    A = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

The remaining symbols are defined smilarly.

The whole set looks like this, with yellow=1 and violet=0:

```
sym=[A,a,ii,I,Y] # array of symbols, numbered from 0 to 4
```

```
plt.figure(figsize=(16, 6)) # figure with horizontal and vertical size

for i in range(1,6):      # loop over 5 figure panels, i is from 1 to 5
    plt.subplot(1, 5, i) # panels, numbered from 1 to 5
    plt.axis('off')       # no axes
    plt.imshow(sym[i-1]) # plot symbol, numbered from 0 to 4
```



It is more convenient to work not with the above two-dimensional arrays, but with one-dimensional vectors obtained with the so-called **flattening** procedure, where a matrix is cut along its rows into a vector. For example

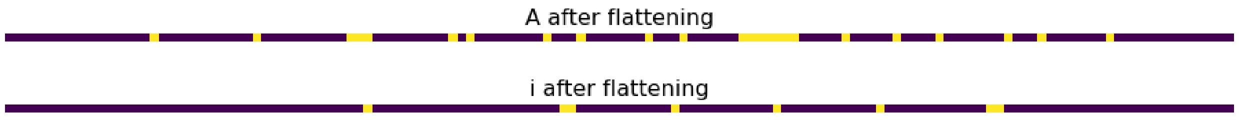
```
t=np.array([[1,2,3],[0,4,0],[3,2,7]]) # a matrix
print(t)
print(t.flatten())      # matrix flattened into a vector
```

```
[[1 2 3]
 [0 4 0]
 [3 2 7]]
[1 2 3 0 4 0 3 2 7]
```

We thus perform the flattenning:

```
fA=A.flatten()
fa=a.flatten()
fi=ii.flatten()
fI=I.flatten()
fY=Y.flatten()
```

to obtain, for instance



The advantage of working with vectors is that we can use the scalar product. Here, the scalar product between two symbols is just equal to the number of common yellow pixels. For instance, for the flattened symbols plotted above we have only two common yellow pixels:

```
np.dot(fA,fi)
```

```
2
```

It is clear that one can use the scalar product as a measure of similarity between the symbols. For the following method to work, the symbols should not be too similar.

3.1.2 Memory matrix

The next algebraic concept we need is the **outer product**. For two vectors v and w, it is defined as $vw^T = v \otimes w$ (as opposed to the scalar product, where $w^T v = w \cdot v$). T denotes transposition. The result is a matrix with the number of rows equal to the length of v, and the number of column equal to the length of w.

For example, with

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}, \quad w = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix},$$

we have

$$v \otimes w = vw^T = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} (w_1, w_2) = \begin{pmatrix} v_1 w_1 & v_1 w_2 \\ v_2 w_1 & v_2 w_2 \\ v_3 v_1 & v_3 w_2 \end{pmatrix}.$$

In numpy

```
print(np.outer([1,2,3],[2,7])) # outer product of two vectors
```

```
[[ 2  7]
 [ 4 14]
 [ 6 21]]
```

Next, we construct a **memory matrix** needed for modeling our heteroassociative memory. Suppose first for simplicity of notation that we only have two associations: $a \to A$ and $b \to B$. Let

$$M = Aa^T / a \cdot a + Bb^T / b \cdot b.$$

Then

$$Ma = A + B\, a \cdot b / b \cdot a,$$

and if a and b were **orthogonal**, i.e. $a \cdot b = 0$, then

$Ma = A$

yielding an exact association. Similarly, we would have $Mb = B$. However, since in a general case the vectors are not exactly orthogonal, an error $B\, b \cdot a / a \cdot a$ (for the association of a) is generated. It is usually small if the number of pixels in our symbols is large and the symbols are, loosely speaking, not too similar. As we will see, the emerging error can be efficiently "filtered out" with an appropriate neuron activation function.

Coming back to our particular case, we thus need four terms in M:

```
M=(np.outer(fA,fa)/np.dot(fa,fa)+np.outer(fa,fA)/np.dot(fA,fA)
   +np.outer(fi,fI)/np.dot(fI,fI)+np.outer(fI,fi)/np.dot(fi,fi)); # associated pairs
```

Now, for each flattened symbol s we will evaluate Ms. The result is a vector, which we want to bring back to the form of the 12×12 pixel array. The operation inverse to flattening in Python is **reshape**. For instance

```
tt=np.array([1,2,3,5]) # test vector
print(tt.reshape(2,2)) # cutting into 2 rows of length 2
```

```
[[1 2]
 [3 5]]
```

For our vectors we have

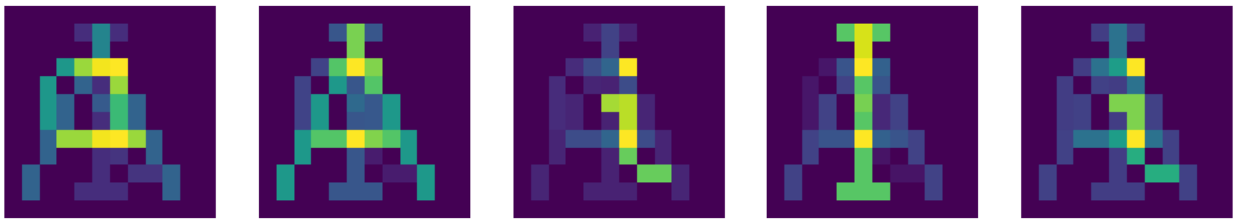
```
Ap=np.dot(M,fA).reshape(12,12)
ap=np.dot(M,fa).reshape(12,12)
Ip=np.dot(M,fI).reshape(12,12)
ip=np.dot(M,fi).reshape(12,12)
Yp=np.dot(M,fY).reshape(12,12) # we also try unassociated symbol Y

symp=[Ap,ap,Ip,ip,Yp] # array of associated symbols
```

For the case of association to A (which shou ld be a), it yields (we use rounding to 2 decimal digits)

```
print(np.round(Ap,2)) # pixel map for the association of the symbol A
```

```
[[0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.25 0.85 0.25 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.85 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   1.   1.6  1.85 1.89 0.   0.   0.   0.   0.  ]
 [0.   0.   1.   0.   0.6  0.25 1.6  0.   0.   0.   0.   0.  ]
 [0.   0.   1.   0.6  0.   0.54 1.29 0.6  0.   0.   0.   0.  ]
 [0.   0.   1.   0.6  0.   0.25 1.29 0.6  0.   0.   0.   0.  ]
 [0.   0.   0.6  1.6  1.6  1.85 1.89 1.6  0.6  0.   0.   0.  ]
 [0.   0.   0.6  0.   0.   0.25 0.29 0.   0.6  0.   0.   0.  ]
 [0.   0.6  0.   0.   0.   0.25 0.   0.29 0.29 0.6  0.   0.  ]
 [0.   0.6  0.   0.   0.25 0.25 0.25 0.   0.   0.6  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

We note that the strength of pixels is now not necessarily equal to 0 or 1, as it was in the original symbols. The graphic representation looks as follows:



We should be able to see in the above picture the sequence a, A, i, I, and nothing particular in the association of Y. We almost do, but the situation is not perfect due to the nonorthogonality error discussed above.

3.1.3 Applying a filter

The result improves greatly when a filter is applied to the pixel maps. Looking at the above print out or the plot of Ap (the symbol associated to A which shoul be a), we note that we should get rid of the "faint shadows", and leave only the pixels of suffient strength, which should then acquire the value 1. In other words, pixels below a bias (threshold) b should be reset to 0, and those above or equal to b should be reset to 1. This can be neatly accomplished with our **neuron** function from Sec. *MCP neuron in Python*. This function has been placed in the library **neural** (see *Appendix*), which we now read in:

```
import sys # system library
sys.path.append('./lib_nn') # my path (linux, Mac OS)

from neural import * # import my library packages
```

We thus define the filter as a neuron with weight $w_0 = -b$ and $w_1 = 1$:

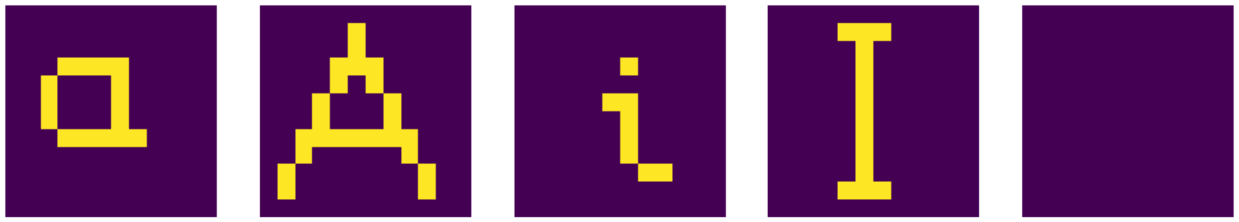
```
def filter(a,b): # a - symbol (2-dim pixel array), b - bias
    n=len(a)      # number of rows (and columns)
    return np.array([[func.neuron([a[i,j]],[-b,1]) for j in range(n)] for i in
↪range(n)])
       # 2-dim array with the filter applied
```

When operating on Ap with appropriately chosen $b = 0.9$ (the level of the bias is very much relevant), the result is

```
print(filter(Ap,.9))
```

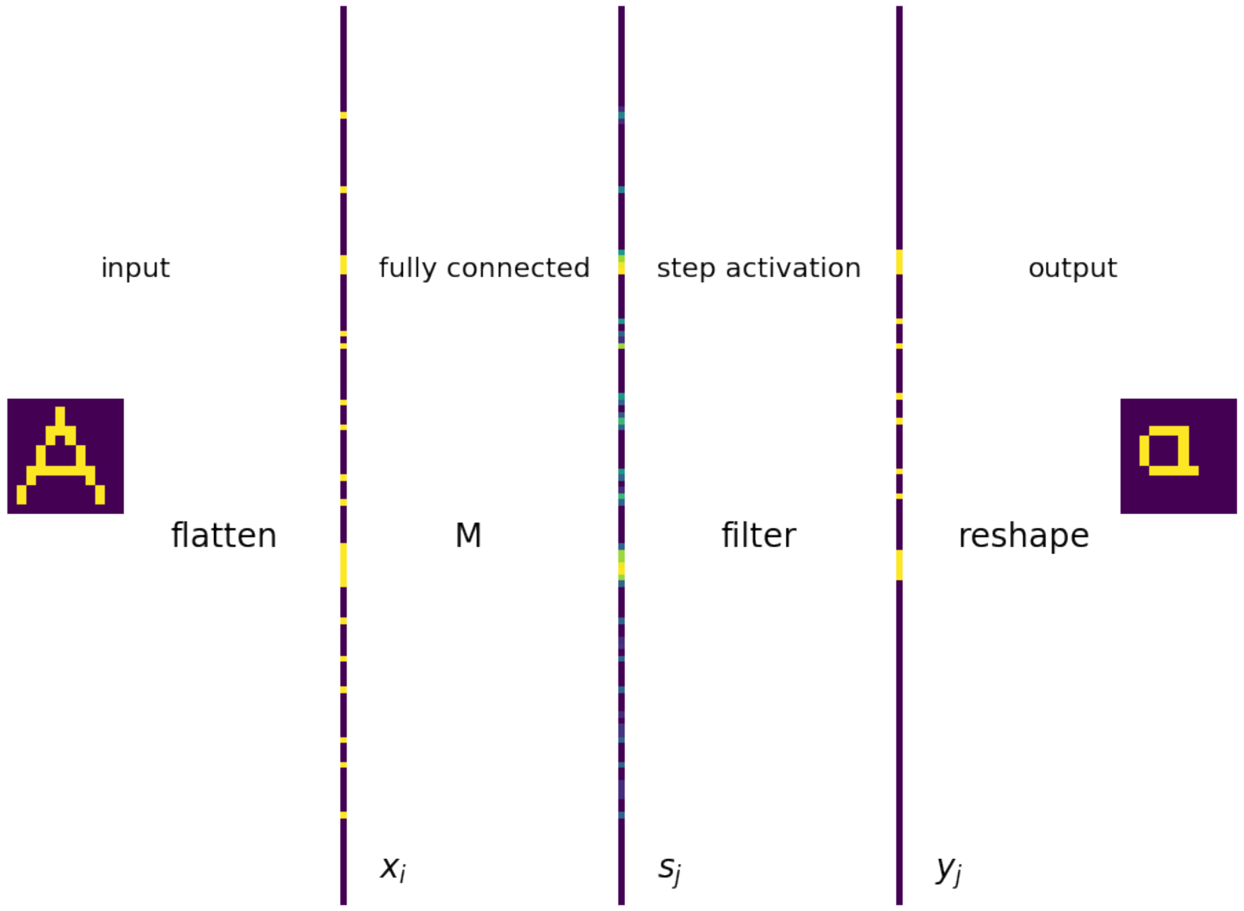
```
[[0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 1 1 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 1 0 0 0 1 0 0 0 0 0]
 [0 0 0 1 1 1 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0]]
```

where we can notice a “clean” symbol a. We check that it actually works perfectly well for all our accociations (such perfection is not always the case):



A representation of the presented model of the heteroassociative memory in terms of ANN can be readily given. In the plot below we indicate all the operations, going from left to right. The input symbol is flattened. The input and output layers are fully connected with edges (not shown) connecting the input cells to the neurons in the output layer. The weights of the edges are equal to the matrix elements M_{ij}, indicated with symbol M. The activation function is the same for all neurons and it has the form of a step function.

At the bottom we indicate the elements of the input vector, x_i, of the signal reaching the neuron j, $s_j = \sum_i x_i M_{ij}$, and the final output $y_j = f(s_j)$.



Summary of the model of the heteroassociative memory

1. Define pairs of associated symbols and construct the memory matrix M.
2. The input is a symbol in the form of a 2-dim array of pixels with values 0 or 1.
3. Flaten the symbol into a vector, which forms the layer of inputs x_i.
4. The weight matrix of the fully connected ANN is M.
5. The signal entering neuron j in the output layer is $s_j = \sum_i x_i M_{ij}$.
6. The activation (step) function with a properly chosen bias yields $y_j = f(s_j)$.
7. Cut the output vector into a matrix of pixels, which constitutes the final output. It should be the symbol associated to the input.

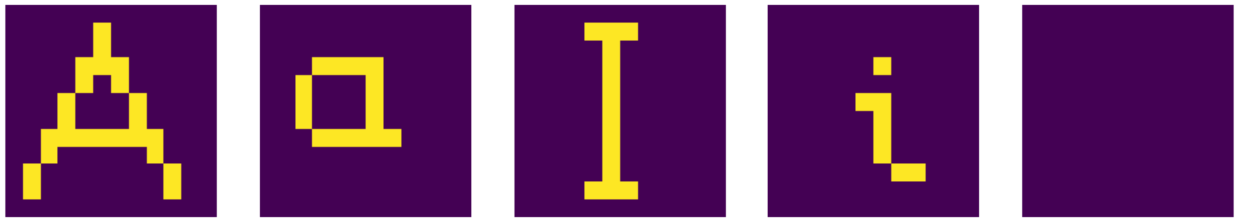
3.2 Autoassociative memory

3.2.1 Self-associations

The autoassociative memory model is in close analogy to the case of the heteroassociatine memory, but now the symbol is associated **to itself**. Why we do such a thing will become clear shortly, when we consider distorted input. We thus define the association matrix as follows:

```
Ma=(np.outer(fA,fA)/np.dot(fA,fA)+np.outer(fa,fa)/np.dot(fa,fa)
    +np.outer(fi,fi)/np.dot(fi,fi)+np.outer(fI,fI)/np.dot(fI,fI))
```

After multiplying the flattened symbol with matrix Ma, reshaping, and filtering (all steps as in the heteroassociative case) we properly get back th original symbols (except for Y, which was not associated).

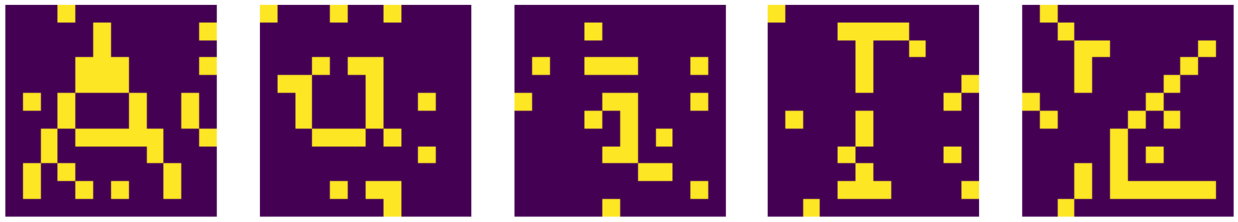


3.2.2 Distorting the image

Now imagine that the original input gets partially destroyed, with some pixels randomly altered from 1 to 0 and vice versa.

```
ne=12 # number of alterations

for s in sym:                         # loop over symbols
    for _ in range(ne):               # loop over alteratons
        i=np.random.randint(0,12)     # random position in row
        j=np.random.randint(0,12)     # random position in column
        s[i,j]=1-s[i,j]               # switching 1 and 0
```

After this the input symbols look like this:

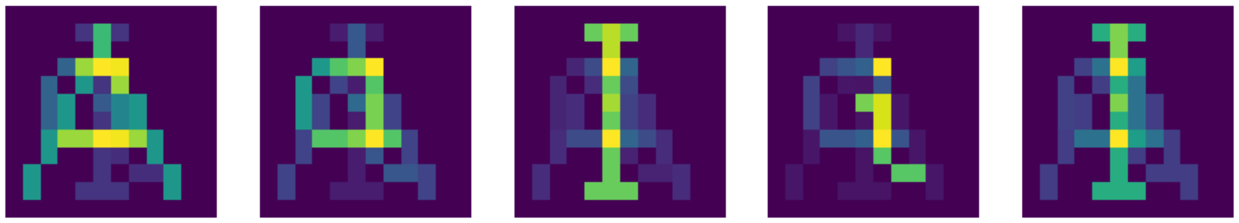


3.2.3 Restoring the symbols

We now apply our model of the autoassociative memory to all the "distroyed" symbols:

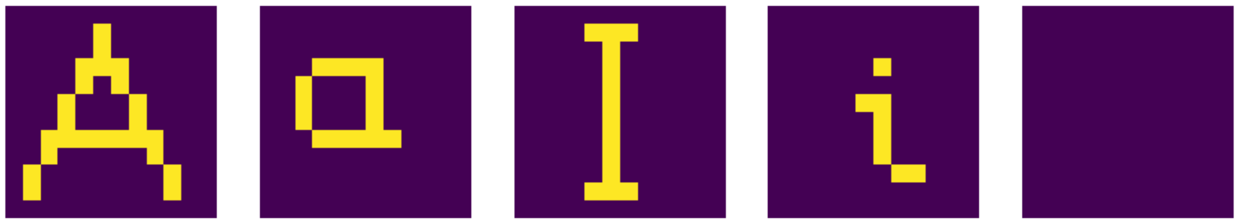
```python
Ap=np.dot(Ma,fA).reshape(12,12)
ap=np.dot(Ma,fa).reshape(12,12)
Ip=np.dot(Ma,fI).reshape(12,12)
ip=np.dot(Ma,fi).reshape(12,12)
Yp=np.dot(Ma,fY).reshape(12,12)

symp=[Ap,ap,Ip,ip,Yp] # array of self-associated symbols
```

which yields



After filtering, with $b = 0.9$, we obtain back the original symbols:

```python
plt.figure(figsize=(16, 6))

for i in range(1,6):       # loop over panels
    plt.subplot(1, 5, i)
    plt.axis('off')
    plt.imshow(filter(symp[i-1],0.9)) # plot filtered symbol
```



Summary of the model of the autooassociative memory

1. Construct the memory matrix Ma.
2. The input is a symbol in the form of a 2-dim array of pixels with values 0 or 1, with a certain number of pixels randomly distorted.
3. Flaten the symbol into a vector, which forms the layer of inputs x_i.
4. The weight matrix of the fully connected ANN is Ma.
5. The signal entering neuron j in the output layer is $s_j = \sum_i x_i M_{ij}$.
6. The activation (step) function with a properly chosen bias yields $y_j = f(s_j)$.
7. Cut the output vector into a matrix of pixels, which constitutes the final output. It should bring back the ariginal symbol.

The application can thus decifer a "destroyed" text, or, more generally, provide an error correcion mechanism.

Important: Message: ANN can serve as very simple models of memory!

Exercises

Play with the lecture code and

- add more and more symbols,
- change the filter level,
- increase the number of alterations.

Discuss your findings.

CHAPTER

FOUR

PERCEPTRON

4.1 Supervised learning

We have shown in the previous chapters that even the simplest ANNs can carry out useful tasks (emulate logical networks or provide simple memory models). Generally, each ANN has

- some **architecture**, i.e. the number of layers, number of neurons in each layer, scheme of connections between the neurons (fully connected or not, feed forward, recurrent, …),
- **weights (hyperparameters)**, with specific values, defining the network's functionality.

The prime practical question is how to set (for a given architecture) the weights such that a requested goal is realized, i.e., a given input yields a desired output. In the tasks discussed earlier, the weights could be constructed *a priori*, be it for the logical gates or for the memory models. However, for more involved applications we want to have an "easier" way of determining the weights. Actually, for complicated problems a "theoretical" determination of weights is not possible at all. This is the basic reason for inventing **learning algorithms**, which automatically adjust the weights with the help of a data sample.

In this chapter we begin to explore such algorithms with the **supervised learning**, used for data classification.

Supervised learning

In this strategy, the data must possess **labels** which a priori determine the correct category for each point. Think for example of pictures of animals (data) and their descriptions (cat,dog,…), which are the labels. The labeled data are split into a **training** sample and a **test** sample.

The basic steps of supervised learning for a given ANN are following:

- Initialize somehow the weights, for instance randomly or to zero.
- Read subsequently the data points from the training sample and pass them throught your ANN. The obtained answer may differ from the correct one, determimed by the label, in which case the weights are adjusted according to a specific prescription (to be discussed later on).
- Repeat, if needed, the previous step. Typically, the weights are changed less and less as the algorithm proceeds.
- Finish the training when a stopping criterion is reached (weights do not change much any more or the maximum number of iterations has been completed).
- Test the trained ANN on the test sample.

If satisfied, you have a desired trained ANN performing a specific task, which can be used on new, unlabeled data. If not, you can split the sample in the training and the test parts in a different way and repeat the procedure from the beginning. Also, you may try to acquire more data, or change your network's architecture.

This he term “supervised” comes form the interpretation of the procedure where the labels are held by a “teacher”, who thus knows which answers are correct and which are wrong, and who **supervises** the training process.

4.2 Binary classifier

The simplest supervised learning algorithm is the perceptron, invented in 1958 by Frank Rosenblatt. It can be used to construct **binary classifiers** for the data. *Binary* means that the network is used to assess if a data point has a particular feature, or not.

Remark

The term *perceptron* is also used for ANNs (without or with intermediate layers) consisting of the MCP neurons (cf. Fig. Fig. 1.4 and Fig. 2.1), on which the perceptron algorithm is executed.

4.2.1 Sample with a known classification rule

To begin, we need some training data, which we will generate as random points in a square. Thus the coordinates of the point, x_1 and x_2, are taken in the range $[0, 1]$. We define two categories: one for the points lying above the line $x_1 = x_2$ (call them pink), and the other for the points lying below (blue). During the generation, we check whether $x_2 > x_1$ or not, and assign a **label** to each data point equal to, correspongigly, 1 or 0.

The function generating the described data point with a label is

```
# returns random coordinates x1, x2 and 1 if x2>x1, 0 otherwise
def point():
    x1=np.random.random()          # random number from the range [0,1]
    x2=np.random.random()
    if(x2>x1):                     # condition met
        return np.array([x1,x2,1]) # add label 1
    else:                          # not met
        return np.array([x1,x2,0]) # add label 0
```

We generate a **training sample** of **npo**=300 labeled data points:

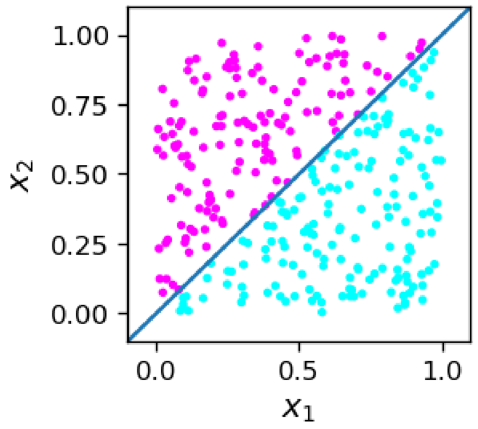
```
npo=300 # number of data points in the training sample

print('  x1         x2         label')        # header
samp=np.array([point() for _ in range(npo)]) # training sample, _ is dummy iterator
print(samp[:5, :])                           # first 5 data points
```

```
  x1         x2         label
[[0.19968208 0.34399239 1.        ]
 [0.11365587 0.21869436 1.        ]
 [0.10854327 0.43462234 1.        ]
 [0.80135246 0.69970322 0.        ]
 [0.41130912 0.80434883 1.        ]]
```

Not to print unnecessarily the very long table, we have used above for the first time the **ranges for array indices**. For example, 2:5 means from 2 to 4 (the last one is excluded!), :5 - from 0 to 4, 5: - from 5 to the end, and : - all the indices.

Graphically, our data are show in the figure below. We also plot the line $x_2 = x_1$, which separates the blue and purple points. In this case the division is a priori possible (we know the rule) in an exact manner.

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.xlim(-.1,1.1)                                 # axes limits
plt.ylim(-.1,1.1)
plt.scatter(samp[:,0],samp[:,1],c=samp[:,2],      # label determines the color
            s=5,cmap=mpl.cm.cool)                 # point size and color

plt.plot([-0.1, 1.1], [-0.1, 1.1])                # separating line

plt.xlabel('$x_1$',fontsize=12)
plt.ylabel('$x_2$',fontsize=12);
```



Linearly separable sets

Two sets of points (e.g. blue and pink) on a plane which are possible to separate with a straigh line are called **linearly separable**. In three dimensions, the sets must be separable with a plane, in general in n dimensions the sets must must be separable with a $n-1$ dimensional hyperplane.

Analitically, if the points in the n dimensional space have coordinates $(x_1, x_2, \dots, x_n)$, one may chose the parameters $(w_0, w_1, \dots, w_n)$ in such a way that one set of points must satisfy the condition

$$w_0 + x_1 w_1 + x_2 w_2 + \dots x_n w_n > 0 \tag{4.1}$$

and the other the opposite condition, with $>$ replaced with $\leq$.

Now a crucial, albeit simple observation: the above inequality is precisely the condition implemented in the *MCP neuron* (with the step activation function) in the convention of Fig. 2.2! We may thus enforce condition (4.1) with the **neuron** function from our **neural** library.

In our example we have for the pink points, by construction,

$$x_2 > x_1 \to s = -x_1 + x_2 > 0$$

from where, using Eq. (4.1), we can immediately read out

$$w_0 = 0, \quad w_1 = -1, w_2 = 1.$$

Thus the **neuron** function is used on a sample point p like this:

```
p=[0.6,0.8]      # sample point with x_2 > x_1
w=[0,-1,1]       # weights as given above

func.neuron(p,w)
```

```
1
```

The neuron fired, so point p is pink.

Observation

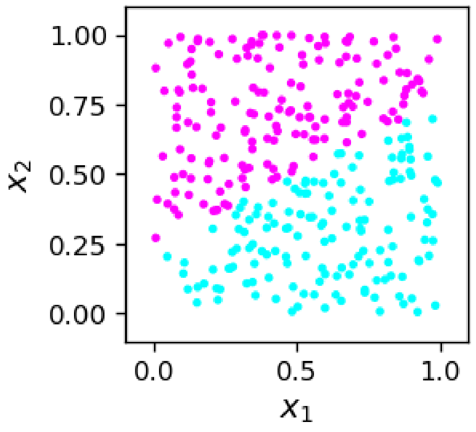
A single MCP neuron with properly chosen weights can be used as a binary classifier.

4.2.2 Sample with an unknown classification rule

At this point the reader may be a bit misled by the apparent triviality of the result. The confusion may stem from the fact that in our example we knew from the outset the rule defining the two classes of points ($x_2 > x_1$, or opposite). However, in a general "real life" situation this is frequently not the case! Imagine that we encounter the (labeled) data **samp2** looking like this:

```
print(samp2[:5])
```

```
[[0.52395167 0.07006229 0.        ]
 [0.46580068 0.17984878 0.        ]
 [0.88313537 0.80437289 1.        ]
 [0.22527853 0.93083163 1.        ]
 [0.67041055 0.97779144 1.        ]]
```



The situation is in some sense inverted now. We have abtained from somewhere the (linearly separable) data, and want to find the rule that defines the two classes. In other words, we need to draw a dividing line, which is equivalent to finding the weights of the MCP neuron of Fig. 2.2 that would carry out the classification.

4.3 Perceptron algorithm

We could still try to figure out somehow the proper weights for the present example and find the dividing line, for instance with a ruler and pencil, but this is not the point. We wish to have a systematic algorithmic procedure that will effortlessly work for this one and any similar situation. The answer is the already mentioned perceptron algorithm.

Before presenting the algorithm, let us remark that the MCP neuron with some set of weigths w_0, w_1, w_2 will always yield some answer for a labeled data point, correct or wrong. For example

```
w=[-0.5,1,0]            # arbitrary choice of weights

print("label  answer") # header

for i in range(5): # look at first 5 points
    print(int(samp2[i,2]),"    ",func.neuron(samp2[i,:2],w))
    # samp2[i,2] is the label, samp2[i,:2] is [x_1,x_2]
```

```
label  answer
0      1
0      0
1      1
1      0
1      1
```

We can see that some answers are equal to the corresponding labels (correct), and some are different (wrong). The general idea now is to **use the wrong answers** to adjust cleverly, in small steps, the weights, such that after many iterations we get all the answers for the training sample correct!

Perceptron algorithm

We iterate over the points of the training data sample. If for a given point the obtained result y_o is equal to the true value y_t (the label), i.e. the answer is correct, we do nothing. However, if it is wrong, we change the weights a bit, such that the chance of getting the wrong answer decreses. The explicit recipe is as follows:

$w_i \to w_i + \varepsilon(y_t - y_o)x_i,$

where ε is a small number (called the **learning speed**) and x_i are the coordinates of the input point, with $i = 0, \ldots, n$.

Let us follow how it works. Suppose first that $x_i > 0$. Then if the label $y_t = 1$ is greater than the obtained answer $y_o = 0$, the weight w_i is increased. Then $w \cdot x$ also increases and $y_o = f(w \cdot x)$ is more likely to acquire the correct value of 1 (we remember how the step function f looks like). If, on the other hand, the label $y_t = 0$ is less than the obtained answer $y_o = 1$, then the weight w_i is decreased, $w \cdot x$ decreases, and $y_o = f(w \cdot x)$ has a better chance of achieving the correct value of 0.

If $x_i < 0$ it is easy to use the same method to check that the recipe also works properly.

When the anwer is correct, $y_t = y_0$, then $w_i \to w_i$, so nothing changes. We do not “spoil” the perceptron!

The above formula can be used many times for the same point from the traing sample. Next, we loop over all the points of the sample, and the whole procedure can still be repeated in many rounds to obtain stable weights (not changing any more as we continue the procedure, or changing very slightly).

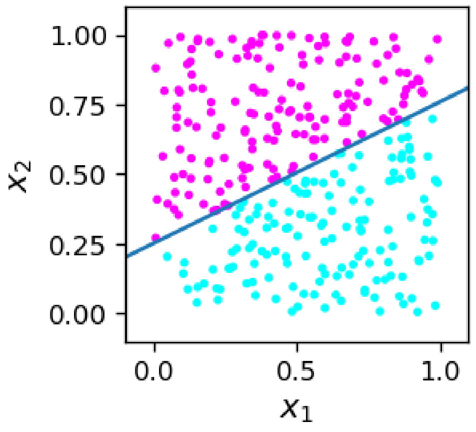
Typically, in such algorithms the learning speed ε is being decreased in successive rounds. This is technically very important, because too large steps can spoil the obtained solution.

The Python implementation of the perceptron algorithm for the 2-dimesional data is as follows:

```
w0=np.random.random()-0.5   # initialize weights randomly in the range [-0.5,0.5]
w1=np.random.random()-0.5
w2=np.random.random()-0.5

eps=.3  # initialize the learning speed

for _ in range(20):         # loop over 20 rounds
    eps=0.9*eps             # in each round decrease the learning speed

    for i in range(npo):    # loop over the points from the data sample

        for _ in range(5): # repeat 5 times for each points

            yo = func.neuron(samp2[i,:2],[w0,w1,w2]) # obtained answer

            w0=w0+eps*(samp2[i,2]-yo)    # weight update (perceptron formula)
            w1=w1+eps*(samp2[i,2]-yo)*samp2[i,0]
            w2=w2+eps*(samp2[i,2]-yo)*samp2[i,1]

print("  w0      w1      w2")         # header
w_o=np.array([w0,w1,w2])              # obtained weights
print(np.round(w_o,3))                # result, rounded to 3 decimal places
```

```
  w0      w1      w2
[-0.441 -0.88    1.74 ]
```

It yields the result



We can see that the algorithm works! All the pink points are above the line, and all the blue ones below. Let us emphasize that the dividing line, given by the equation

$$w_0 + x_1 w_1 + x_2 w_2 = 0,$$

does not result from our a priori knowledge, but from the training of the MCP neuron which sets its weights.

Note: One can prove that the perceptrom algorith converges if and only if the data are linearly seperable.

We may now reveal our secret! The data of our sample were labeled at the time of creation with the rule

$$x_2 > 0.25 + 0.52 x_1$$

which corresponds to weights $w_0^c = 0.25$, $w_1^c = -0.52$, $w_2^c = 1$.

```
w_c=np.array([-0.25,-0.52,1]) # weights used for labeling the training sample
print(w_c)
```

```
[-0.25 -0.52  1.  ]
```

Note that this is not at all the same as the weights obtained from the training:

```
print(np.round(w_o,3))
```

```
[-0.441 -0.88   1.74 ]
```

The reason is twofold. First, note that the inequality condition (4.1) is unchanged if we multiply both sides by a **positive** constant c. We may therefore scale all the weight by c, and the situation (the answers of the MCP neuron, the dividing line) remains exactly the same (we encounter here an **equivalece class** of weight scaled with a positive factor).

For that reason, when we divide correspondingly the obtained weights by the weights used to label the sample, we get (almost) constant values:

```
print(np.round(w_o/w_c,3))
```

```
[1.764 1.693 1.74 ]
```

The reason why the ratio values for $i = 0, 1, 2$ are not exactly the same is that the sample has a finite number of points (here 300). Thus, there is gap between the two classes of points and there is some room for "jiggling" the separating line a bit. With more data points this mismatch effect would decrease (see the homework problem below).

4.3.1 Testing the classifier

Due to the limited size of the training sample and the "jiggling" effect desribed above, the classification result on a test sample is sometimes wrong. This always applies to the points near the dividing line, which is determined with accuracy depending on the multiplicity of the training sample. The code below carries out the check on a test sample. The test sample consists of labeled data generated randomly "on the flight" with the same function **point2** as for the training data used before:

```
def point2():
    x1=np.random.random()          # random number from the range [0,1]
    x2=np.random.random()
    if(x2>x1*0.52+0.25):           # condition met
        return np.array([x1,x2,1]) # add label 1
    else:                          # not met
        return np.array([x1,x2,0]) # add label 0
```

The code for testing is as follows:

```
er= np.empty((0,3))  # initialize an empty 1 x 3 array to store misclassified points

ner=0                 # initial number of misclassified points
nt=3000               # number of test points

for _ in range(nt): # run for nt points
    ps=point2()       # a test point
    if(func.neuron(ps[:2],[w0,w1,w2])!=ps[2]): # if wrong answers
↪
```

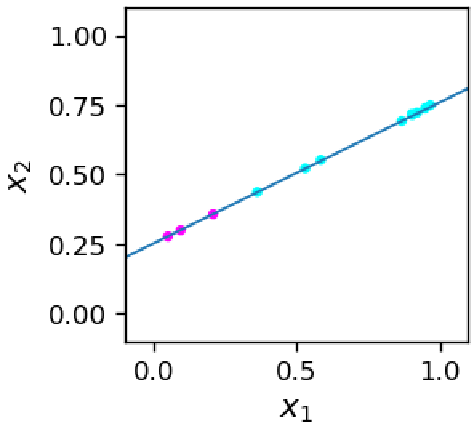
(continues on next page)

(continued from previous page)

```
        er=np.append(er,[ps],axis=0)           # add the point to er
        ner+=1

print("number of misclassified points = ",ner," per ",nt," (", np.round(ner/nt*100,1),
↪"% )")
```

```
number of misclassified points =  12  per  3000  ( 0.4 % )
```

As we can see, a small number of test points are misclassified. All these points lie near the separating line.



Misclassification

As it became clear, the reason for misclassification comes from the fact that the training sample does not determine the separating line precisely, but with some uncertainty, as there is a gap between the points of the training smaple. For a better result, the training points would have to be “denser” in the vicinity of the separation line, or the training sample would have to be larger.

Exercises

- Play with the lecture code and see how the percentage of misclassified points decreases with the increasing size of the training sample.
- As the perceptron algorithm converges, at some point the weights stop to change. Improve the lecture code by implementing stopping when the weights do not change when passing to the next round.
- Generalize the above classifier to points in 3-dimensional space.

CHAPTER

FIVE

MORE LAYERS

5.1 Two layers of neurons

In the previous chapter we have seen that the MCP neuron with the step activation fuction realizes the inequality $x \cdot w = w_0 + x_1 w_1 + ... x_n w_n > 0$, where n in the dimensionality of the input space. It is instructive to come up here with a geometric interpretation. Taking for definiteness $n = 2$ (the plane), the above inequality corresponds to a division into two half-planes. The line given by the equation

$$x \cdot w = w_0 + x_1 w_1 + ... x_n w_n = 0$$

is the **dividing line**.

Imagine now that we have more such conditions: two, three, etc., in general k independent conditions. Taking a conjunction of these conditions, we can build regions as shown, e.g., in Fig. 5.1.

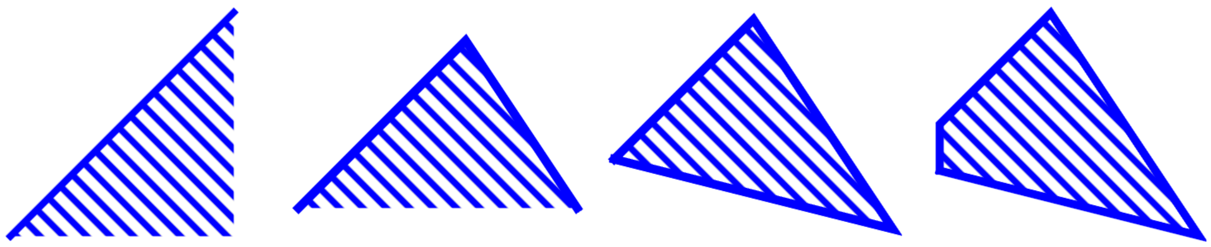


Fig. 5.1: Sample convex regions in the plane obtained, from left to right, with one inequality condition, and a conjunction of 2, 3, or 4 inequality conditions, yielding **polygons**.

Convex region

By definition, region A is convex if and only if a straight line between any two points in A is contained in A. A region which is not convex is called **concave**.

Clearly, k inequality conditions can be imposed with k MCP neurons. Recall from section *Boolean functions* that we can straightforwardly build boolean functions with the help of the neural networks. In particular, we can make a conjunction of k conditions by taking a neuron with the weights $w_0 = -1$ and $1/k < w_i < 1/(k-1)$, where $i = 1, \dots, k$. One possibility is, e.g.,

$$w_i = \frac{1}{k - \frac{1}{2}}.$$

Indeed, let $p_0 = 0$, and the conditions imposed by the inequalities be denoted as p_i, $i = 1, \dots, k$, which may take values 1 or 0 (true or false). Then

$$p \cdot w = -1 + p_1 w_1 + \dots + p_k w_k = -1 + \frac{p_1 + \dots p_k}{k - \frac{1}{2}} > 0$$

if and only if all $p_i = 1$, i.e. all the conditions are true. This builds th AND gate for the incoming k boolean signals.

The architecture of networks for $k = 1, 2, 3$, or 4 conditions is shown in Fig. 5.2. Starting from the second panel, we have networks with two layers of neurons, with k neurons in the intermediate layer, providing the inequality conditions, and one neuron in the output layer, acting as the AND gate. Of course, for one condition it is sufficient to have a single neuron, as in the left panel of Fig. 5.2.

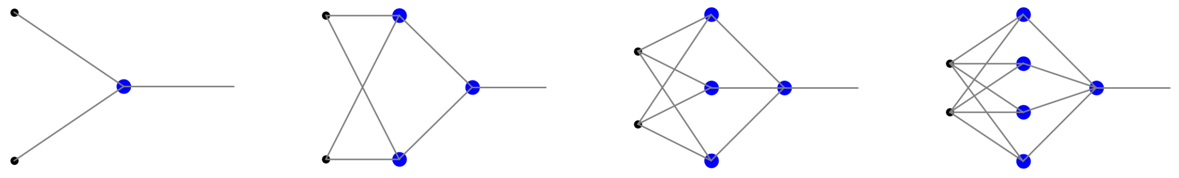


Fig. 5.2: Networks capable of classifying data in the corresponding regions of Fig. 5.1.

With the geometric interpretation, the first neuron layer represents the k half-planes, and the neuron in the second layer correspond to a convex region with k sides.

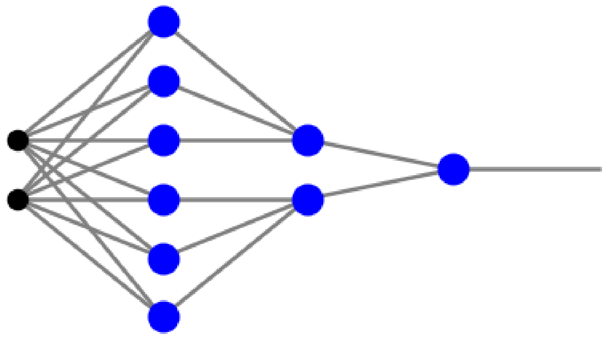
The situation generalizes in an obvious way to data in more dimensions. In that case we have more black dots in the inputs in Fig. 5.2. Geometrically, non $n = 3$ we deal with dividing planes and convex polyhedrons, and for $n > 3$ with dividing hyperplanes and convex polytopes.

Note: If there are numerous neurons k in the intermediate layer, the resulting polygon has many sides which may approximate a smooth boundary, such as an arc. The approximation is better and better as k increases.

Important: A percepton with two-layers of neurons can classify points belonging to a convex region in n-dimensional space.

5.2 Three or more layers of neurons

We have just shown that a two-layer network may classify a convex polygon. Imagine now that we produce two such figures in the second layer of neurons, for instane as in the following network:



Note that the first and second neuron layers are not fully connected here, as we "stack on top of each other" two networks producing triangles, as in the third panel of Fig. 5.2. Next, in the third neuron layer (here having a single neuron) we implement a $p \wedge \sim q$ gate, i.e. the conjunction of the conditions that the points belong to one triangle and do not belong to the other one. As we will show shortly, with appropriate weights, the above network may produce a concave region, for example a triangle with a triangular hollow:

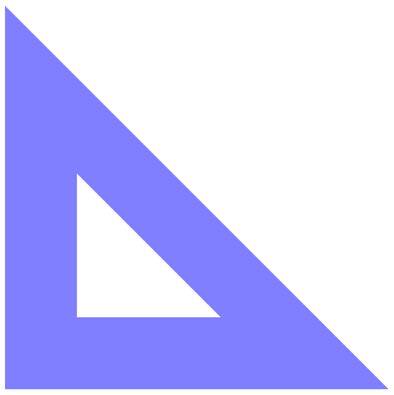


Fig. 5.3: Triangle with a tringular hollow.

Generalizing this argument to other shapes, one can show an important theorem that

Important: A perceptron with three or more layers with sufficiently many neurons can classify points belonging to **any** region in n-dimensional space with $n-1$-dimensional hyperplane boundaries.

It is worth stressing here that three layers provide full functionality! Adding more layers to a classifier does not increase its capabilities.

5.3 Feeding forward in Python

Before proceeding with an explicit example, we need a Python code for propagation of the signal in a general fully-connected feed-forward network. First, we represent the architecture of a network with l neuron layers as an array of the form

$$[n_0, n_1, n_2, ..., n_l],$$

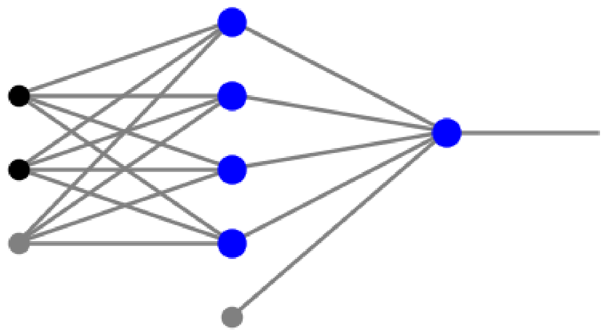
where n_0 in the number of the input nodes, and n_i are the numbers of neurons in layers $i = 1, \dots, l$. For instance, the architecture of the network from the fourth panel of Fig. 5.2 is

```
arch=[2,4,1]
arch
```

```
[2, 4, 1]
```

In the codes of this course we use the convention of Fig. 2.2, namely, the bias is treated uniformly with the remaining signal. However, the bias notes are not included in the numbers n_i defined above. In particular, a more detailed view of the fourth panel of Fig. 5.2 is

```
draw.plot_net(arch);
```



Here, black dots mean input, gray dots indicate the bias nodes carrying input =1, and the blue blobs are the neurons.

Next, we need the weights of the connections. There are l sets of weights, each one corresponding to the set of edges entering a given neuron layer from the left. In the above example, the first neuron layer (blue blobs to the left) has weights which form a 3×4 matrix. Here 3 is the number of nodes in the preceding layer (including the bias node) and 4 is the number of neurons in the first neuron layer. Similarly, the weights associated with the second (output) layer form a 4×1 matrix. Hence, in our convention, the weight matrices corresponding to subsequent neuron layers $1, 2, \ldots, l$ have dimensions

$$(n_0 + 1) \times n_1, \ (n_1 + 1) \times n_2, \ \ldots \ (n_{l-1} + 1) \times n_l.$$

To store all the weights of a network we actually need **three** indices: one for the layer, one for the number of nodes in the preceding layer, and one for the number of nodes in the given layer. We could have used a three-dimensional array here, but since we number the neuron layers staring from 1, and arrays start numbering from 0, it is more convenient to use the Python **dictionary** structure. We will then store the weights as

$$w = \{1 : arr^1, 2 : arr^2, ..., l : arr^l\},$$

where arr^i is a **two-dimensional** array (matrix) of weights for the neuron layer i. For the case of the above figure we could thus take, for instance

```
w={1:np.array([[1,2,1,1],[2,-3,0.2,2],[-3,-3,5,7]]),2:np.array([[1],[0.2],[2],[2],[-0.
↪5]])}

print(w[1])
print("")
print(w[2])
```

```
[[ 1.   2.   1.   1. ]
 [ 2.  -3.   0.2  2. ]
 [-3.  -3.   5.   7. ]]

[[ 1. ]
 [ 0.2]
 [ 2. ]
 [ 2. ]
 [-0.5]]
```

For the signal propagating along the network we also use a dictionary of the form

$$x = \{0 : x^0, 1 : x^1, 2 : x^2, ..., l : x^l\},$$

where x^0 is the input, and x^i is the output leaving the neuron layer i, with $i = 1, \ldots, l$. All symbols x^j, $j = 0, \ldots, l$, are one-dimensional arrays. The bias nodes are included, hence the dimensions of x^j are $n_j + 1$, except for the ouput layer which has no bias node, hence x^l has dimension n_l. In other words, the dimensions of the signal arrays are equal to the total number of nodes in each layer.

Next, we present the corresponding formulas in rather painful detail, as this is key to avoid any possible confusion related to the notation. We already know from (2.2) that for a single neuron with n inputs its incoming signal is calculated as

$$s = x_0 w_0 + x_1 w_1 + x_2 w_2 + ... + x_n w_n = \sum_{\beta=0}^{n} x_\beta w_\beta .$$

With more layers (index i) and neurons (n_i in layer i), the notation generalizes into

$$s_\alpha^i = \sum_{\beta=0}^{n_{i-1}} x_\beta^{i-1} w_{\beta\alpha}^i, \quad \alpha = 1 \dots n_i, \quad i = 1, \dots, l.$$

Note that the summation starts from $\beta = 0$ to account for the bias node in the preceding layer $(i-1)$, but α starts from 1, as only neurons (and not the the bias node) in layer i receive the signal (see the figure below).

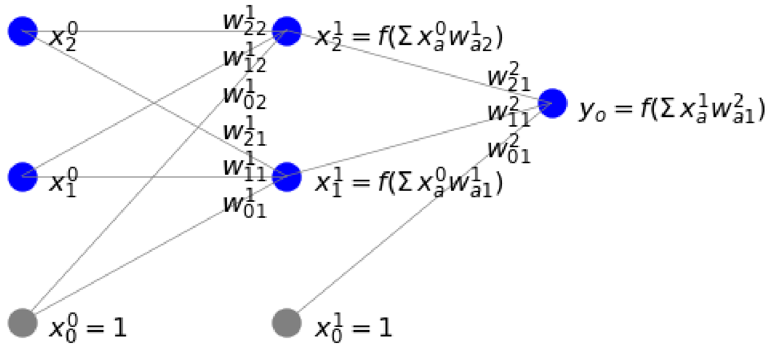
In the algebraic matrix notation, we can also write more compactly $s^{iT} = x^{(i-1)T} W^i$, with T denoting transposition. Explicitly,

$$\begin{pmatrix} s_1^i & s_2^i & \dots & s_{n_i}^i \end{pmatrix} = \begin{pmatrix} x_0^{i-1} & x_1^{i-1} & \dots & x_{n_{i-1}}^{i-1} \end{pmatrix} \begin{pmatrix} w_{01}^i & w_{02}^i & \dots & w_{0,n_i}^i \\ w_{11}^i & w_{12}^i & \dots & w_{1,n_i}^i \\ \dots & \dots & \dots & \dots \\ w_{n_{i-1}1}^i & w_{n_{i-1}2}^i & \dots & w_{n_{i-1}n_i}^i \end{pmatrix}.$$

As we already know very well, the output from a neuron is obtained by acting on its incoming input with an activation function. Thus we finally have

$$x_\alpha^i = f(s_\alpha^i) = f\left(\sum_{\beta=0}^{n_{i-1}} x_\beta^{(i-1)} w_{\beta\alpha}^i \right), \quad \alpha = 1 \dots n_i, \quad i = 1, \dots, l,$$
$$x_0^i = 1, \quad i = 1, \dots, l-1,$$

with the bias nodes set to one. The figure below illustrates the scheme:



The implementation of the feed-forward propagation explained above in Python is following:

```
def feed_forward(ar, we, x_in, f=func.step):
    """
    Feed-forward propagation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
```

(continues on next page)

(continued from previous page)

```
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    f - activation function (default: step)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)

    """

    l=len(ar)-1                     # number of neuron layers
    x_in=np.insert(x_in,0,1)        # input, with the bias node inserted

    x={}                            # empty dictionary
    x.update({0: np.array(x_in)}) # add input signal

    for i in range(1,l):            # loop over layers except the last one
        s=np.dot(x[i-1],we[i])      # signal, matrix multiplication
        y=[f(s[k]) for k in range(arch[i])] # output from activation
        x.update({i: np.insert(y,0,1)}) # add bias node and update x

                                    # the last layer - no adding of the bias node
        s=np.dot(x[l-1],we[l])      # signal
        y=[f(s[q]) for q in range(arch[l])] # output
        x.update({l: y})            # update x

    return x
```

Next, we test how **feed_forward** works on a sample input. For brevity, we do not pass the input bias node of the input in the argument. It is added inside the function.

```
xi=[2,-1]
x=func.feed_forward(arch,w,xi,func.step)
print(x)
```

```
{0: array([ 1,  2, -1]), 1: array([1, 1, 0, 0, 0]), 2: [1]}
```

The final output of this network is obtained as

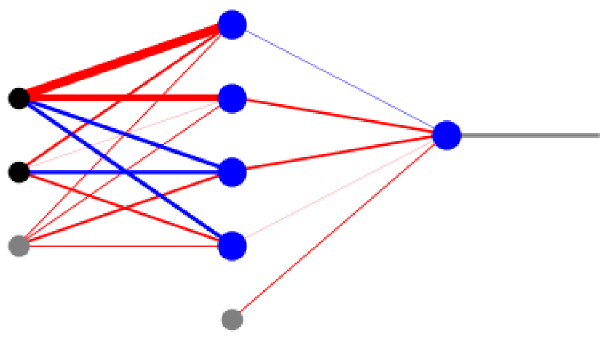
```
x[2][0]
```

```
1
```

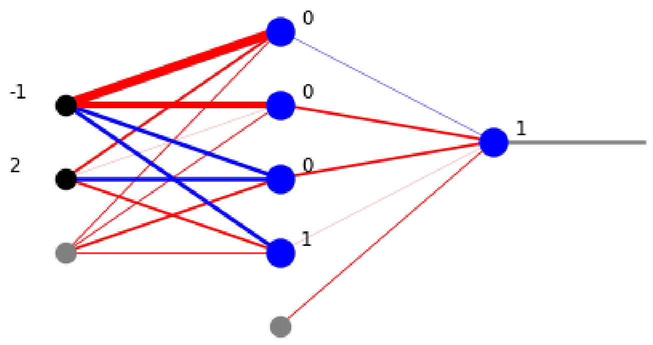
5.4 Visualization

For visualization of simple networks, in the **neural** library we provide some drawing functions which show the weights, as well as the signals. Function **plot_net_w** draws positive weights in red and negative in blue, with the widths reflecting their magnitude. The last parameter, here 0.5, rescales the widths such that the graphics looks nice. Function **plot_net_w_x** prints in addition the values of the signal leaving the neurons in each layer.

```
draw.plot_net_w(arch,w,0.5);
```



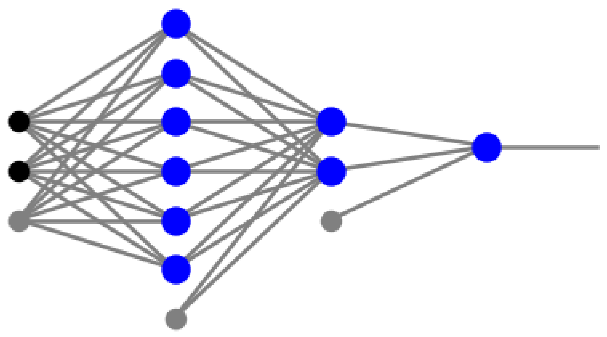
```
draw.plot_net_w_x(arch,w,0.5,x);
```



5.5 Classifier with three neuron layers

We are now ready to explicitly construct an example of a binary classifier of points in a concave region: a triagle with a triangular hollow of Fig. 5.3. The network architecture is

```
arch=[2,6,2,1]
draw.plot_net(arch);
```



The geometric conditions and the corresponding weights for the first neuron layer are

α	inequality condition	$w^1_{0\alpha}$	$w^1_{1\alpha}$	$w^1_{2\alpha}$
1	$x_1 > 0.1$	-0.1	1	0
2	$x_2 > 0.1$	-0.1	0	1
3	$x_1 + x_2 < 1$	1	-1	-1
4	$x_1 > 0.25$	-0.25	1	0
5	$x_2 > 0.25$	-0.25	0	1
6	$x_1 + x_2 < 0.8$	0.8	-1	-1

Conditions 1-3 provide boundaries for the bigger traingle, and 4-6 for the smaller one contained in the bigger one. In the second neuron layer we need to realize two AND gates for conditions 1-3 and 4-6, espoectively, hence we take

α	$w^2_{0\alpha}$	$w^2_{1\alpha}$	$w^2_{2\alpha}$	$w^2_{3\alpha}$	$w^2_{4\alpha}$	$w^2_{5\alpha}$	$w^2_{6\alpha}$
1	-1	0.4	0.4	0.4	0	0	0
2	-1	0	0	0	0.4	0.4	0.4

Finally, in the output layer we take the $p \wedge \sim q$ gate, hence

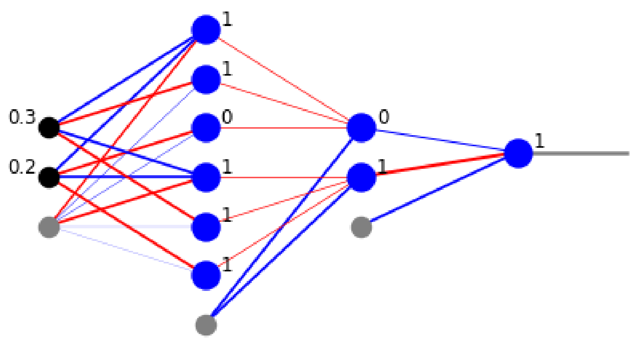
α	$w^3_{0\alpha}$	$w^3_{1\alpha}$	$w^3_{2\alpha}$
1	-1	1.2	-0.6

Putting all togethr, the weight dictionary is

```
w={1:np.array([[-0.1,-0.1,1,-0.25,-0.25,0.8],[1,0,-1,1,0,-1],[0,1,-1,0,1,-1]]),
   2:np.array([[-1,-1],[0.4,0],[0.4,0],[0.4,0],[0,0.4],[0,0.4],[0,0.4]]),
   3:np.array([[-1],[1.2],[-0.6]])}
```

Feeding forward a sample input yields

```
xi=[0.2,0.3]
x=func.feed_forward(arch,w,xi)
draw.plot_net_w_x(arch,w,1,x);
```



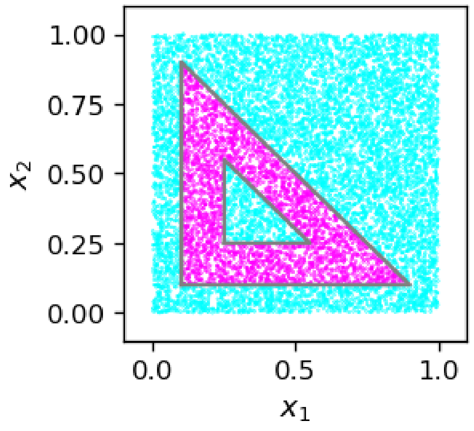
We have just found that point [0.2,0.3] is within our region (1 from the output layer). Actually, we we have more information from the intermediae layers. From the second neuron layer we know that the point belongs to the bigger triangle (1 from the lower neuron) and does not belong to the smaller triangle (0 from the upper neuron). From the first neuron layer we may read the condition from the six inequalities.

Next, we will test how our network works for other points. First, we define a function generating a random point in the square $[0,1] \times [0,1]$ and pass it through the network. We assign to it label 1 if it belongs to the requested triangle with the hollow, and 0 otherwise. Subsequently, we create a large sample of such points and generate the graphics, using pink for label 1 and blue for label 0.

```
def po():
    xi=[np.random.random(),np.random.random()] # random point from the [0,1]x[0,1]␣
↪square
    x=func.feed_forward(arch,w,xi)             # run feed forward
    return [xi[0],xi[1],x[3][0]]               # the point's coordinates and label
```

```
samp=np.array([po() for _ in range(10000)])
print(samp[:5])
```

```
[[0.83806748 0.62718519 0.        ]
 [0.46726263 0.34999938 1.        ]
 [0.69545367 0.95397076 0.        ]
 [0.19930639 0.52982584 1.        ]
 [0.41437336 0.69846145 0.        ]]
```



We can see that our little machine works perfectly well!

At this point the reader might rightly say that the preceding results are trivial: in essence, we have just been implementing some geometric conditions and their conjunctions.

However, there is an important case against this apparent triviality. Imagine we have the data sample with labels, and only this, as in the example of the single MCP neuron of chapter *MCP Neuron*. Then we do not have the dividing conditions to begin with and need some efficient way to find them. This is exacly what teaching of classifiers does: its sets the weights in such a way that the proper conditions are implicitly built in. After the material of this section, the reader should be convinced that this is perfectly possible.

CHAPTER

SIX

BACK PROPAGATION

6.1 Minimizing the error

We now go back to the perceptron algorithm of chapter *Perceptron* to look in some more detail at its performance as a function of weights. Note that in our example with points on the plane the condition for the pink points is given by the inequality

$w_0 + w_1 x_1 + w_2 x_2 > 0.$

We have already mentioned the equivalence class related to dividing this inequality with a positive constant. In general, at least one of the weights must be nonzero to have a nontrivial condition. Suppose or definiteness that $w_0 \neq 0$ (other cases may be treated analogously). Then we can divide both sides with $|w_0|$

$$\frac{w_0}{|w_0|} + \frac{w_1}{|w_0|} x_1 + \frac{w_2}{|w_0|} x_2 > 0.$$

Introducing $v_1 = \frac{w_1}{w_0}$ and $v_2 = \frac{w_2}{w_0}$, this can be rewritten in the form

$$\text{sgn}(w_0)(1 + v_1\, x_1 + v_2\, x_2) > 0,$$

where $\text{sgn}(w_0) = \frac{w_0}{|w_0|}$, hence we effectively have a two-parameter system (for each sign of w_0).

Obviously, with some values of v_1 and v_2 and for a given point from the sample, the perceptron will provide a correct or incorrect answer. It is thus natural to define the **error function** E such that each point of p from the sample contributes 1 if the answer is incorrect, and 0 if it is correct:

$$E(v_1, v_2) = \sum_p \begin{cases} 1 - \text{incorrect}, \\ 0 - \text{correct} \end{cases}$$

E is thus the number of misclassified points. We can easily construct this function for a labeled data sample in the format [x1, x2, label]:

```
def error(w0, w1 ,w2, sample, f=func.step):
    """
    error function for the perceptron (for 2-dim data with labels)

    inputs:
    w0, w1, w2 - weights
    sample - labeled data sample in format [x1, x1, label]
    f - activation function

    returns:
    error
    """
```

(continues on next page)

(continued from previous page)

```
    er=0                               # initial value of error
    for i in range(len(sample)):    # loop over data points
        yo=f(w0+w1*sample[i,0]+w2*sample[i,1]) # obtained answer
        er+=(yo-sample[i,2])**2
                        # sample[i,2] is the label
                        # adds the square of the difference of yo and the label
                        # this adds 1 if the answer is incorrect, and 0 if correct
    return er  # the error
```

Atually, we have used a little trick here, in b=view if the future developments. Denoting the obtained result for a given data point as $y_o^{(p)}$ and the true result (label) as $y_t^{(p)}$ (both have values 0 or 1), we may write equivalently

$$E(v_1, v_2) = \sum_p \left(y_o^{(p)} - y_t^{(p)} \right)^2,$$

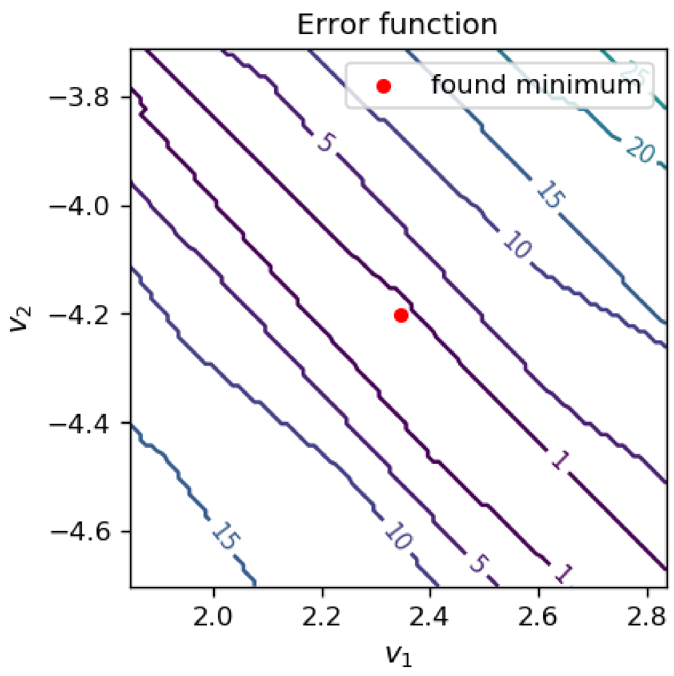
which is the programmed formula. Indeed, when $y_o^{(p)} = y_t^{(p)}$ (correct answer) the contribution of the point is 0, and when $y_o^{(p)} \neq y_t^{(p)}$ (wrong answer) the contribution is $(\pm 1)^2 = 1$.

We repeat the simulations of chapter *Perceptron* for the sample **samp2** of 200 points (the sample was built with $w_0 = -0.25$, $w_1 = -0.52$, and $w_2 = 1$, which corresponds to $v_1 = 2.08$ and $v_2 = -4$, with $\mathrm{sgn}(w_0) = -1$). Then we evaluate the error function $E(v_1, v_2)$.

Next, we run the perceptron alogorithm:

We note above that the final error is very small or 0 (depending on the particular simulation). It is illuminating to look at a contour map of the error function $E(v_1, v_2)$ in the vicinity of the optimal parameters:

```
fig, ax = plt.subplots(figsize=(3.7,3.7),dpi=120)

delta = 0.01 # grid step in v1 and v2 for the contour map
ran=0.5       # plot range around (v1_o, v2_o)

v1 = np.arange(v1_o-ran,v1_o+ran, delta) # grid for v1
v2 = np.arange(v2_o-ran,v2_o+ran, delta) # grid for v2
X, Y = np.meshgrid(v1, v2)

Z=np.array([[error(-1,-v1[i],-v2[j],samp2,func.step)
             for i in range(len(v1))] for j in range(len(v2))]) # values of E(v1,v2)

CS = ax.contour(X, Y, Z, [1,5,10,15,20,25,30,35,40,45,50])
ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=9) # contour labels

ax.set_title('Error function', fontsize=11)
ax.set_aspect(aspect=1)

ax.set_xlabel('$v_1$', fontsize=11)
ax.set_ylabel('$v_2$', fontsize=11)

ax.scatter(v1_o, v2_o, s=20,c='red',label='found minimum') # our found optimal point

ax.legend();
```



We can see in the plot that that the found minimum is in (or close to, depending on the simulation) the elongated region of v_1 and v_2 where the error vanishes.

6.2 Continuous activation function

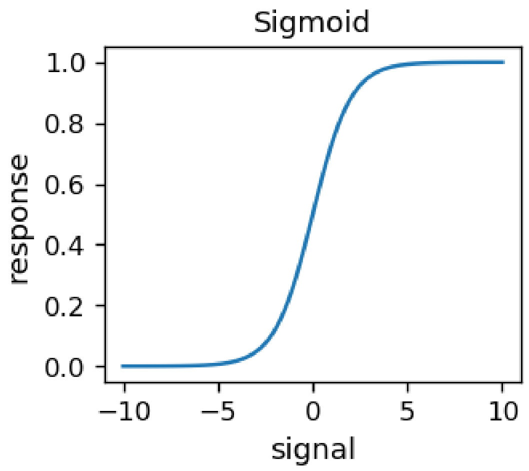
Coming back to the contour chart above, we can see that the lines are "serrated". This is because the error function, for an obvious reason, assumes integer values. It is therefore discontinuous and non-differentiable. The discontinuities obviously originate from the discontinuous activation function, i.e. the step function. Having in mind the techniques we will get to know soon, it is advantageous to use the continuous activation functions. Historically, the so-called **sigmoid**

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

has been used in many practical applications of ANNs.

```
# sigmoid, a.k.a. the logistic function, or simply (1+arctanh(-s/2))/2
def sig(s):
    return 1/(1+np.exp(-s))
```

```
draw.plot(sig,start=-10,stop=10,title='Sigmoid');
```



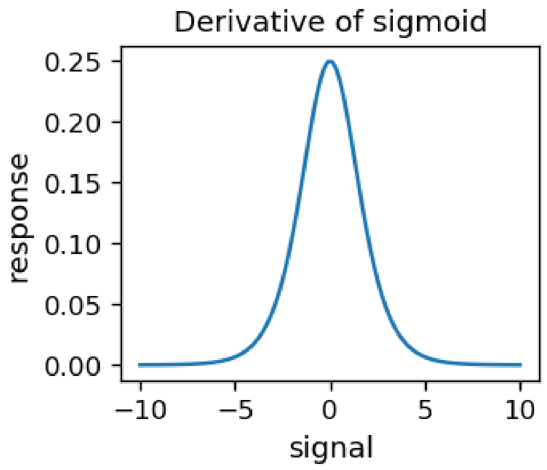
This function is of course differentiable. Moreover,

$\sigma'(s) = \sigma(s)[1 - \sigma(s)]$,

which is its special feature.

```
# derivative of sigmoid
def dsig(s):
     return sig(s)*(1-sig(s))
```

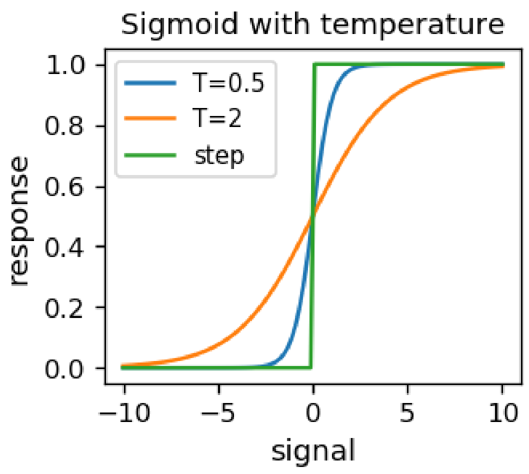
```
draw.plot(dsig,start=-10,stop=10,title='Derivative of sigmoid');
```



A sigmoid with "temperature" T is also introduced (this nomenclature is associated with similar expressions for thermodynamic functions in physics):

$$\sigma(s;T) = \frac{1}{1+e^{-s/T}}.$$

```
# sigmoid with temperature T
def sig_T(s,T):
    return 1/(1+np.exp(-s/T))
```



Note: For smaller and smaller T, it approaches the previously used step function. Note that the argument of the sigmoid is the quotient

$s/T = (w_0 + w_1 x_1 + w_2 x_2)/T = w_0/T + w_1/T\, x_1 + w_2/T\, x_2 = \xi_0 + xi_1 x_1 + xi_2 x_2$$,

which means that we can always assume $T = 1$ without losing generality (T is the "scale"). However, we now have three independent arguments ξ_0, ξ_1, and ξ_2. Thus, it is impossible to reduce the situation to two independent parameters, as was the case above.

We will now repeat our example with the classifier, but with the activation function given by the sigmoid. The error function, with

$$y_o^{(p)} = \sigma(w_0 + w_1 x_1^{(p)} + w_2 x_2^{(p)}).$$

becomes

$$E(w_0, w_1, w_2) = \sum_p \left[\sigma(w_0 + w_1 x_1^{(p)} + w_2 x_2^{(p)}) - y_t^{(p)}\right]^2.$$

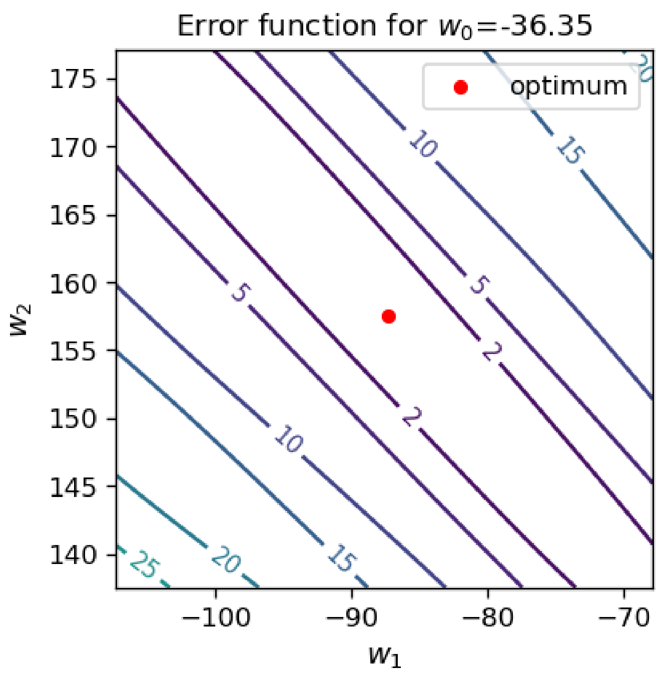
We run the perceptron alorithm with the sigmoid activation function 500 times:

```
weights=[[func.rn()],[func.rn()],[func.rn()]]      # random weights from [-0.5,0.5]

print("   w0   w1/w0  w2/w0 error")    # header

eps=0.7                        # initial learning speed
for r in range(1000):          # rounds
    eps=0.9995*eps             # decrease learning speed
    weights=teach_perceptron(samp2,eps,weights,func.sig) # update weights
    if r%100==99:
        w0_o=weights[0][0]                 # updated weights
        w1_o=weights[1][0]
        w2_o=weights[2][0]
        v1_o=w1_o/w0_o
        v2_o=w2_o/w0_o
        print(np.round(w0_o,3),np.round(v1_o,3),np.round(v2_o,3),
              np.round(error(w0_o, w0_o*v1_o, w0_o*v2_o, samp2, func.sig),5))
```

```
   w0    w1/w0   w2/w0  error
-19.439 2.253 -4.118 0.62695
-24.146 2.288 -4.16 0.43044
-27.015 2.32 -4.208 0.27756
-29.122 2.344 -4.246 0.19078
-30.81 2.362 -4.274 0.14195
-32.227 2.376 -4.295 0.11308
-33.452 2.386 -4.31 0.09482
-34.529 2.393 -4.32 0.08244
-35.487 2.398 -4.328 0.07349
-36.347 2.402 -4.334 0.06667
```

The error function now has 3 arguments, so it cannot be drawn in two dimensions. We can, however, look at its projections, e.g. with a fixed value of w_0.

```
fig, ax = plt.subplots(figsize=(3.7,3.7),dpi=120)

delta = 0.5
ran=20
r1 = np.arange(w1_o-ran, w1_o+ran, delta)
r2 = np.arange(w2_o-ran, w2_o+ran, delta)
X, Y = np.meshgrid(r1, r2)

Z=np.array([[error(w0_o,r1[i],r2[j],samp2,func.sig)
             for i in range(len(r1))] for j in range(len(r2))])

CS = ax.contour(X, Y, Z,[0,2,5,10,15,20,25,30,35,40,45,50])
ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=9)

ax.set_title('Error function for $w_0$='+str(np.round(w0_o,2)), fontsize=11)
ax.set_aspect(aspect=1)
ax.set_xlabel('$w_1$', fontsize=11)
ax.set_ylabel('$w_2$', fontsize=11)

ax.scatter(w1_o, w2_o, s=20,c='red',label='optimum') # our found optimal point

ax.legend();
```



Note: In the present example, when we carry out more and more iterations, we notice that the magnitude of weights becomes lager and lager, while the error naturally gets smaller. The reason is following: our data sample is separable, so in the case when the step function is used for activation, it is possible to separate the sample with the dividing line and get down with the error to zero. In the case of the sigmoid, there is always some (tiny) contribution to the error, as the values are between 0 and 1. As we have discussed above, in the sigmoid, whose argument is $(w_0 + w_1 x_1 + w_2 x_2)/T$, increasing the weights is equivalent to scaling down the temperature. Then, however, the sigmoid approaches the step function, and the error tends to zero. Precisely this behavior is seen in the simulations.

6.3 Steepest descent

Generic comment on minimization …

For a differentiable function of multiple variables, $F(z_1, z_2, ..., z_n)$, locally the steepest slope is defined by the minus gradient of the function F, i.e. the slope is in the direction of the vector

$$-\left(\frac{\partial F}{\partial z_1}, \frac{\partial F}{\partial z_2}, ..., \frac{\partial F}{\partial z_n}\right),$$

where the partial derivatives are defined as the limit

$$\frac{\partial F}{\partial z_1} = \lim_{\Delta \to 0} \frac{F(z_1 + \Delta, z_2, ..., z_n) - F(z_1, z_2, ..., z_n)}{\Delta},$$

and similarly for the other z_i.

The method of finding the minimum of a function by the steepest descent method is given by the iterative algorithm, where we update the position at each iteration step m with

$$z_i^{(m+1)} = z_i^{(m)} - \epsilon \frac{\partial F}{\partial z_i}.$$

In our case, we minimize the error function

$$E(w_0, w_1, w_2) = \sum_p [y_o^{(p)} - y_t^{(p)}]^2 = \sum_p [\sigma(s^{(p)}) - y_t^{(p)}]^2 = \sum_p [\sigma(w_0 x_0^{(p)} + w_1 x_1^{(p)} + w_2 x_2^{(p)}) - y_t^{(p)}]^2,$$

hence

$$\frac{\partial E}{\partial w_i} = \sum_p 2[\sigma(s^{(p)}) - y_t^{(p)}]\,\sigma'(s^{(p)})\,x_i^{(p)} = \sum_p 2[\sigma(s^{(p)}) - y_t^{(p)}]\,\sigma(s^{(p)})\,[1-\sigma(s^{(p)})]\,x_i^{(p)}$$

(derivative of square function $\times$ derivative of the sigmoid $\times$ derivative of $s^{(p)}$), where we have used the special property of the sigmoid derivative in the last equality. The steepest descent method updates the weights as follows:

$$w_i \to w_i - \varepsilon(y_o^{(p)} - y_t^{(p)}) y_o^{(p)} (1 - y_o^{(p)}) x_i.$$

Note that updating always occurs, because the response $y_o^{(p)}$ is never strictly 0 or 1 for the sigmoid, whereas the true value (label) $y_t^{(p)}$ is 0 or 1.

Because $y_o^{(p)}(1 - y_o^{(p)}) = \sigma(s^{(p)})[1 - \sigma(s^{(p)})]$ is nonzero only around $s^{(p)} = 0$ (see the sigmoid derivative plot earlier), thus updating only occurs near the "threshold". This is fine, as the "problems" are near the dividing line.

For comparison, the earlier perceptron algorithm is structurally very similar,

$$w_i \to w_i - \varepsilon\,(y_o^{(p)} - y_t^{(p)})\,x_i,$$

but here the updating occurs for all points of the sample, not just near the threshold.

The code for the learning algorith with the steepest descent update of weights is following:

```
# Steepest descent for a single perceptron

def teach_sd(sample, eps, w_in, f=func.sig):
    [[w0],[w1],[w2]]=w_in
    for i in range(len(sample)):
        for k in range(10):

            yo=f(w0+w1*sample[i,0]+w2*sample[i,1])

            # update of weights
            w0=w0+eps*(sample[i,2]-yo)*yo*(1-yo)*1
            w1=w1+eps*(sample[i,2]-yo)*yo*(1-yo)*sample[i,0]
            w2=w2+eps*(sample[i,2]-yo)*yo*(1-yo)*sample[i,1]
    return [[w0],[w1],[w2]]
```

Its performance is similar to perceptron algorithm studied above.

```
weights=[[func.rn()],[func.rn()],[func.rn()]]      # random weights from [-0.5,0.5]

print("   w0   w1/w0  w2/w0 error")   # header

eps=0.7                         # initial learning speed
for r in range(1000):           # rounds
    eps=0.9995*eps              # decrease learning speed
    weights=teach_sd(samp2,eps,weights) # update weights
    if r%100==99:
        w0_o=weights[0][0]                 # updated weights
        w1_o=weights[1][0]
        w2_o=weights[2][0]
```

(continues on next page)

(continued from previous page)

```
        v1_o=w1_o/w0_o
        v2_o=w2_o/w0_o
        print(np.round(w0_o,3),np.round(v1_o,3),np.round(v2_o,3),
              np.round(error(w0_o, w0_o*v1_o, w0_o*v2_o, samp2, func.sig),5))
```

```
   w0   w1/w0  w2/w0 error
-9.485 2.271 -4.147 1.63446
-11.975 2.297 -4.173 1.16447
-13.537 2.321 -4.21 0.92417
-14.687 2.339 -4.24 0.779
-15.607 2.353 -4.262 0.68315
-16.375 2.364 -4.279 0.6152
-17.036 2.372 -4.291 0.56424
-17.615 2.378 -4.3 0.52435
-18.129 2.382 -4.307 0.49208
-18.589 2.386 -4.313 0.46532
```

The problems with finding the minimum of multivariable functions are well known:

- There may be local minima, and therefore it may be difficult to find the global minimum (see example above with a function with two minima).
- The minimum can be at infinity (that is, it does not exist mathematically).
- The function around the minimum can be very flat, so the gradient is very small, and the update is extremely slow.
- Numerical accuracy can be a problem.

Overall, numerical minimization of functions is an art!

6.4 Backprop algorithm

The material of this section is absolutely **crucial** to understanding this very important idea of training neural networks. At the same time, it can be quite difficult for people less familiar with mathematical analysis, as there will be derivations and formulas with rich notation. However, this cannot be presented more simply than below, with the necessary accuracy.

In the example above, we knew in advance with what recipe we were generating points, so in this fortunate and rare situation, we were able to determine the weights exactly by simple reasoning. In general, this is not the case. As for the single neuron in the previous lecture, we want to train our network on the training sample. The difference is that now there are two layers (apart from the input) and two sets of weights: between the input and the middle layer and between the middle layer and the output.

The method we derive step by step here, which is the famous **back propagation algorithm (backprop)** [Arthur E. Bryson, Yu-Chi Ho, 1969] for updating the weights of a multi-layer network, uses two elements:

- the **chain rule** for computing the derivative of a composite function, known to you from the mathematical analysis, and
- **steepest descent method**, explained in the previous lecture.

Chain rule

For a composite function

$[f(g(x))]' = f'(g(x))g'(x).$

For a composition of more functions $[f(g(h(x)))]' = f'(g(h(x)))\, g'(h(x))\, h'(x)$, etc.

We now move on to formulating the back propagation algorithm for a perceptron with any number of neuron layers, l, and any number of n_l neurons in the output layer. This generalization is conceptually very simple and is based on the same reasoning as for the case of single neuron presented earlier, involving the chain rule and the steepest descent. On the other hand, the notation becomes quite cumbersome, which can make the material seem difficult.

The neurons in intermediate layers $j = 1, \ldots, l-1$ are numbered with corresponding indices $\alpha_j = 0, \ldots, n_j$, with 0 indicating the bias node. In the output layer, having no bias node, the numbering is $\alpha_l = 1, \ldots, n_l$. The error function introduced earlier is a sum over the points of the training sample and over the nodes in the otput layer:

$$E(\{w\}) = \sum_p \sum_{\alpha_l=1}^{n_l} \left[y_{o,\alpha_l}^{(p)}(\{w\}) - y_{t,\alpha_l}^{(p)} \right]^2,$$

where $\{w\}$ represent all the network weights. We will deal with a single point contribution to E, denoted as e. It is a sum over all neurons in the output layer:

$$e(\{w\}) = \sum_{\alpha_l=1}^{n_l} \left[y_{o,\alpha_l} - y_{t,\alpha_l} \right]^2,$$

where we have dropped the superscript (p) for brevity. For neuron α_j in layer j the entering signal is

$$s_{\alpha_j}^j = \sum_{\alpha_{j-1}=0}^{n_{j-1}} x_{\alpha_{j-1}}^{j-1} w_{\alpha_{j-1}\alpha_j}^j.$$

The outputs from the output layer are

$$y_{o,\alpha_l} = f\left(s_{\alpha_l}^l\right)$$

whereas the output signals in the intermediate layers $j = 1, \ldots, l-1$ are

$$x_{\alpha_j}^j = f\left(s_{\alpha_j}^j\right), \quad \alpha_j = 1, \ldots, n_j, \quad \text{and} \quad x_0^j = 1,$$

with the bias node having the value 1.

Subsequent explicit substitutions of the above formulas into e are as follows:

$e = \sum_{\alpha_l=1}^{n_l} \left(y_{o,\alpha_l} - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=0}^{n_{l-1}} x_{\alpha_{l-1}}^{l-1} w_{\alpha_{l-1}\alpha_l}^l\right) - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=0}^{n_{l-2}} x_{\alpha_{l-2}}^{l-2} w_{\alpha_{l-2}\alpha_{l-1}}^{l-1}\right) w_{\alpha_{l-1}\alpha_l}^l + x_0^{l-1} w_{0\gamma}^l\right) - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=1}^{n_{l-2}} f\left(\sum_{\alpha_{l-3}=0}^{n_{l-3}} x_{\alpha_{l-3}}^{l-3} w_{\alpha_{l-3}\alpha_{l-2}}^{l-2}\right) w_{\alpha_{l-2}\alpha_{l-1}}^{l-1} + x_0^{l-2} w_{0\alpha_{l-1}}^{l-1}\right) w_{\alpha_{l-1}\alpha_l}^l + x_0^{l-1} w_{0\alpha_l}^l\right) - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\ldots f\left(\sum_{\alpha_0=0}^{n_0} x_{\alpha_0}^0 w_{\alpha_0\alpha_1}^1\right) w_{\alpha_1\alpha_2}^2 + x_0^1 w_{0\alpha_2}^2 \ldots\right) w_{\alpha_{l-1}\alpha_l}^l + x_0^{l-1} w_{0\alpha_l}^l\right) - y_{t,\alpha_l}\right)^2$

Calculating successive derivatives with respect to the weights, and going backwards, i.e. from $j = l$ down to 1, we get (the evaluation requires dilligence and noticing the emerging regularity)

$$\frac{\partial e}{\partial w_{\alpha_{j-1}\alpha_j}^j} = x_{\alpha_{j-1}}^{j-1} D_{\alpha_j}^j, \quad \alpha_{j-1} = 0, \ldots, n_{j-1}, \quad \alpha_j = 1, \ldots, n_j,$$

where

$D^l_{\alpha_l} = 2(y_{o,\alpha_l} - y_{t,\alpha_l})\, f'(s^l_{\alpha_l}),$

$D^j_{\alpha_j} = \sum_{\alpha_{j+1}} D^{j+1}_{\alpha_{j+1}}\, w^{j+1}_{\alpha_j \alpha_{j+1}}\, f'(s^j_{\alpha_j}), \quad j = l-1, l-2, \dots, 1.$

The last expression is a recurrence going backward. We note that to obtain D^j, we need D^{j+1}, which we have already obtained in the previos step, as well as the signal s^j, which we know from the outset, as we first carry out the feed forward propagation. This provides a simplification in the evaluation of derivatives and updating the weights.

With the steepest descent prescription, the weights are updated as

$$w^j_{\alpha_{j-1}\alpha_j} \to w^j_{\alpha_{j-1}\alpha_j} - \varepsilon x^{j-1}_{\alpha_{j-1}} D^j_{\alpha_j},$$

For the case of sigmoid we can use

$$\sigma'(s^{(i)}_A) = \sigma'(s^{(i)}_A)(1 - \sigma'(s^{(i)}_A)) = x^{(i)}_A (1 - x^{(i)}_A).$$

Note: The above formulas explain the name **back propagation**, because in updating the weights we start from the last layer and then we go back recursively to the beginning of the network. At each step, we need only the signal in the given layer and the properties of the next layer! These features follow from

1. the feed-forward nature of the network, and
2. the chain rule in evaluation of derivatives.

If activation functions are diferent in various layers (denote them with f_j for layer j), then there is an obvious modification:

$D^l_{\alpha_l} = 2(y_{o,\alpha_l} - y_{t,\alpha_l})\, f'_l(s^l_{\alpha_l}),$

$D^j_{\alpha_j} = \sum_{\alpha_{j+1}} D^{j+1}_{\alpha_{j+1}}\, w^{j+1}_{\alpha_j \alpha_{j+1}}\, f'_j(s^j_{\alpha_j}), \quad j = l-1, l-2, \dots, 1.$

6.4.1 Code for backprop

Now we present a code that implements our algorithm for networks with any number of layers and any number of neurons in the output layer. The previous one was for a single intermediate layer and one output neuron. It is simpy a programmatic implementation of the formulas derived above. In the code, we keep the notation from the above derivation.

The code has 12 lines only, not counting the comments!

```
def back_prop(fe,la, p, ar, we, eps,f=func.sig,df=func.dsig):
    """
    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - disctionary of weights
    eps - learning speed
    f   - activation function
    df  - derivaive of f
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
    nl=ar[l]    # number of neurons in the otput layer

    x=func.feed_forward(ar,we,fe[p],ff=f) # feed-forward of point p

```

(continues on next page)

(continued from previous page)

```
    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                    df(np.dot(x[l-1],we[l]))[gam] for gam in range(nl)]})
    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])
```

6.5 Example with the circle

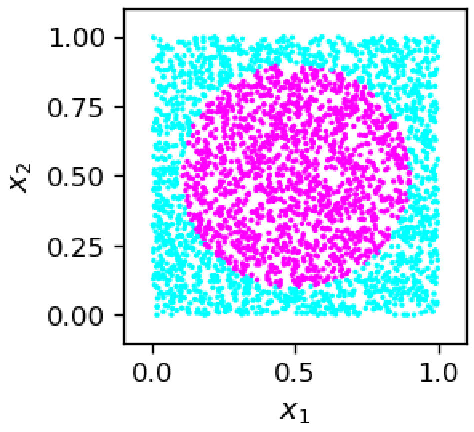
We illustrate the code on the example of a binary classifier of points inside a circle.

```
def cir():
    x1=np.random.random() # coordinate 1
    x2=np.random.random() # coordinate 2
    if((x1-0.5)**2+(x2-0.5)**2 < 0.4*0.4): # inside the circle of radius 0.4
                                            # centered at (0.5,0.5)
        return np.array([x1,x2,1])
    else:                                              # outside
        return np.array([x1,x2,0])
```

For future generality **(new convention)**, we split the sample into an array of features and labels:

```
sample_c=np.array([cir() for _ in range(3000)]) # sample
features_c=np.delete(sample_c,2,1)
labels_c=np.delete(np.delete(sample_c,0,1),0,1)
```

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)
plt.scatter(sample_c[:,0],sample_c[:,1],c=sample_c[:,2],
            s=1,cmap=mpl.cm.cool,norm=mpl.colors.Normalize(vmin=0, vmax=.9))

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11);
```



We take the following architecture and initial parameters:

```
arch_c=[2,4,4,1]                  # architecture
weights=func.set_ran_w(arch_c,10) # scaled random initial weights
eps=.7                            # initial learning speed
```

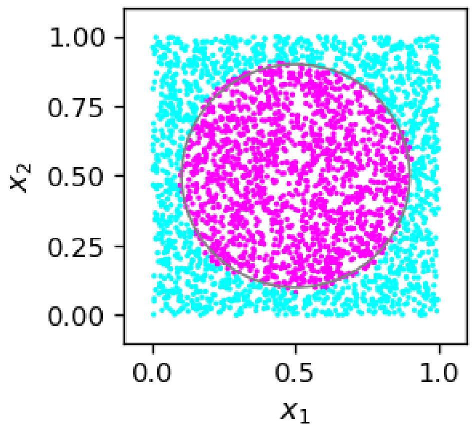
The simulation takes a few minutes,

```
for k in range(300):    # rounds
    eps=.995*eps        # decrease learning speed
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features_c)):                # loop over points
        func.back_prop(features_c,labels_c,p,arch_c,weights,eps,
                       f=func.sig,df=func.dsig) # backprop
```

```
100  200  300
```

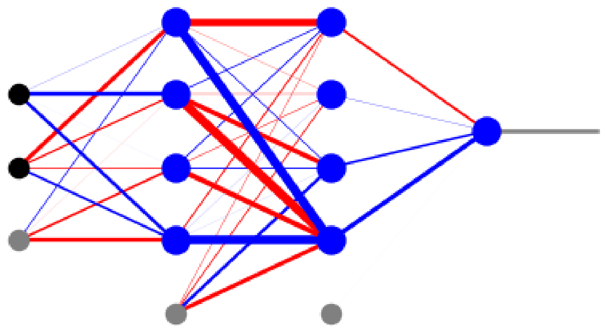
whereas testing is very fast:

```
test=[]

for k in range(3000):
    po=[np.random.random(),np.random.random()]
    xt=func.feed_forward(arch_c,weights,po,ff=func.sig)
    test.append([po[0],po[1],np.round(xt[len(arch_c)-1][0],0)])

tt=np.array(test)

fig=plt.figure(figsize=(2.3,2.3),dpi=120)

# drawing the circle
ax=fig.add_subplot(1,1,1)
circ=plt.Circle((0.5,0.5), radius=.4, color='gray', fill=False)
ax.add_patch(circ)

plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)
plt.scatter(tt[:,0],tt[:,1],c=tt[:,2],
            s=1,cmap=mpl.cm.cool,norm=mpl.colors.Normalize(vmin=0, vmax=.9))

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11);
```



The trained network looks like this:

```
fnet=draw.plot_net_w(arch_c,weights,.1);
```

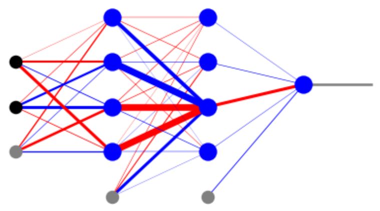
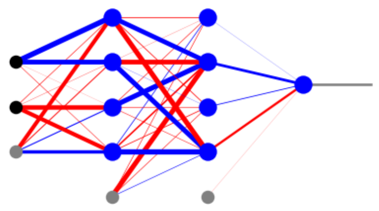
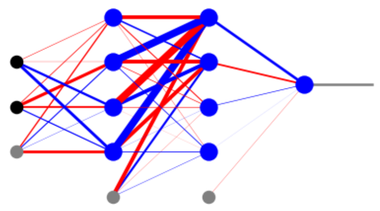


Note: It is fascinating that we have trained the network to recognize if a point is in a circle, and it has no concept whatsoever of geometry, Euclidean distance, equation of the circle, etc. The network just learned "empirically" how to proceed on a training sample!

Note: The result in the plot is very good, perhaps except, as always, near the boundary. In view of our discussion of chapter *More layers*, where we have set the weights of a network with three neuron layers from geometric considerations, the quality of the present result is stunning. We do not see any straight sides of a polygon, but a nicely rounded boundary.

Local minima

We have mentioned before the emergence of local minima in multivariable optimization as a potential problem. In the figure below we show three different results of the backprop code for our classifier of points in a circle. We note that each of them has a radically different set of weights, whereas the results on the test sample are equally good for each case. This shows that the backprop optimization end up, as anticipated, in a local minimum. However, each of these local minima works well and equally good. This is actually the reason why backprop can be used in practical problems: there are zillions of local mnima, but it does not matter!

6.6 General remarks

There are some more important nad general observations:

Note:

- Supervised training of an ANN takes a very long time, but using a trained ANN takes a blink of an eye. The asymmetry originates from the simple fact that the multi-parameter optimization takes very many function calls (here **feed-forward**), but the usage on a point involves just one function call.
- The classifier trained with backprop may work inaccurately for the points near the boundary lines. A remedy is to trained more for improvement, and/or increase the size of the training sample.
- However, a too long learning on the same training sample does not actually make sense, because the accuracy stops improving at some point.
- Local minima occur in bckprop, but this is by no means an obstacle to the use of the backprop algorithm. This is an important practical feature.
- Various improvements of the steepest descent method, or altogether different minimization methods may be used (see homework). They can largely increase the efficiency of the algorithm.
- When going backwards with updating the weights in subsequent layers, one may introduce an increamnet factor (see homework below). This helps with performance.
- Finally, different activation functions may be used to improve performance (see the following).

Exercises

1. Prove (analytically) by taking the derivative that $\sigma'(s) = \sigma(s)[1-\sigma(s)]$. Show that the sigmoid is the only function with this property.
2. Modify the lecture example of the classifier of points in a circle by replacing the figure into
   - semicircle,
   - two circles,
   - ring, or
   - any of your favorite shapes.
3. Repeat 2., experimenting with the number of layers and neurons, but remember that a large number of them increases the computation time and does not necessarily improve the result. Rank each case by the fractin of misclassified points in a test sample. Find an optimum architecture for each of the considered figures.
4. If the network has a lot of neurons and connections, little signal flows through each synapse, hence the network is resistant to a small random damage. This is what happens in the brain, which is constantly “damaged” (cosmic

rays, alcohol, $\dots$). Besides, such a network after destruction can be (already with a smaller number of connections) retrained. Take your trained network from problem 2. and remove one of its **weak** connections, setting the coresponding weight to 0. Test this demaged network on a test sample and draw conclusions.

5. **Scaling weights in back propagation.** A disadvantage of using the sigmoid in the back propagation algorithm is a very slow updating of weights in layers distant from the output layer (the closer to the beginning of the network, the slower it is). A remedy here is a re-scaling the weights, where the learning speed in the layers, counting from the back, is successively increased by a certain factor. We remember that successive derivatives contribute factors of the form $\sigma'(s) = \sigma(s)[1-\sigma(s)] = y(1-y)$ to the update rate, where y is in the range $(0,1)$. Thus the value of $y(1-y$ cannot exceed 1/4, so in the following layers (counting from the back) the product $[y(1-y]^n \leq 1/4^n$. To prevent this "shrinking", the learning rate can be multiplied by 4^n: $4, 16, 64, 256, \dots$. Another heuristic argument [Rigler, Irvine, & Vogl, 1989] suggests even faster growing factors of the form 6^n: $6, 36, 216, 1296, \dots$

   - Enter the above recipes into the code for backpropagation.
   - Check if they improve the algorithm performance for the deeper networks used, e.g. circle point classifier, etc.
   - For assessment of performance, carry out the execution time measurement (e.g., using the Python **time** library packet).

6. **Steepest descent improvement.** The method of the steepest descent of finding the minimum of a function of many variables used in the lecture depends on the local gradient. There are much better approaches that give a better convergence to the (local) minimum. One of them is the recipe of Barzilai-Borwein explained below. Implement this method in the back propagation algorithm. Vectors x in n-dimensional space are updated in subsequent iterations as $x^{(m+1)} = x^{(m)} - \gamma_m \nabla F(x^{(m)})$, where m numbers the iteration, and the speed of learning depends on the behavior at the two (current and previous) points:

$$\gamma_m = \frac{\left|\left(x^{(m)} - x^{(m-1)}\right) \cdot \left[\nabla F(x^{(m)}) - \nabla F(x^{(m-1)})\right]\right|}{\left\|\nabla F(x^{(m)}) - \nabla F(x^{(m-1)})\right\|^2}.$$

CHAPTER

SEVEN

INTERPOLATION

7.1 Simulated data

So far we have been concerned with **classification**, i.e. with networks recognizing whether a given object (in our case a point on a plane) has certain features. Now we pass to another practical application, namely **interpolating functions**. This use of ANNs has become widely used in scientific data analysis. We illustrate the method on a simple example, which explains the basic idea and shows how the method works.

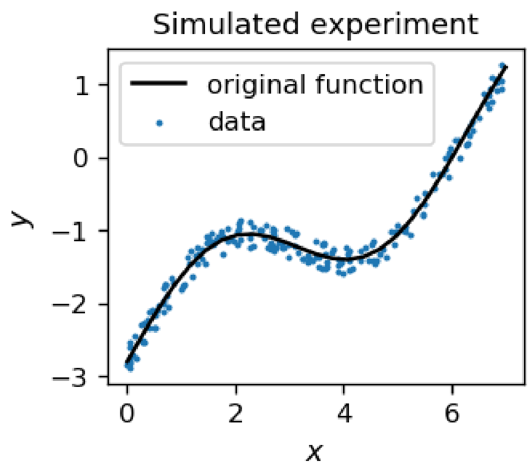
Imagine you have some experimental data. Here we simulate them in an artificial way, e.g.

```
def fi(x):
    return 0.2+0.8*np.sin(x)+0.5*x-3 # some function

def data():
    x = 7.*np.random.rand() # random x coordinate
    y = fi(x)+0.4*func.rn() # y coordinate = the function + noise from [-0.2,0.2]
    return [x,y]
```

We should now think in terms of supervised learning: x is the "feature", and y the "label".

We table our noisy data points and plot them together with the function **fi(x)** around which they concentrate. It is an imitation of an experimental measurement, which is always burdened with some error, here mimicked wih random noise.

```
tab=np.array([data() for i in range(200)])     # data sample
features=np.delete(tab,1,1)                     # x coordinate
labels=np.delete(tab,0,1)                       # y coordinate
```



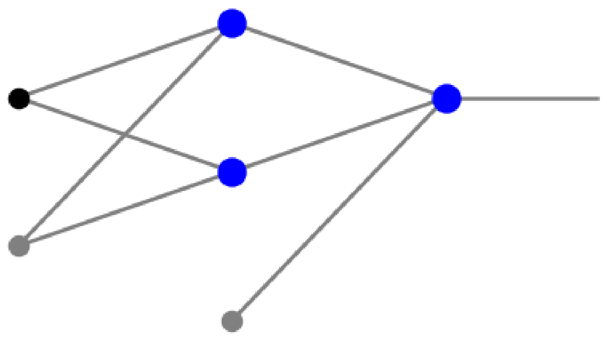
In our current language of ANNs, we therefore have a training sample consisting of points with the input (feature) x and the true output (label) y. As before, we minimize the error function from an appropriate neural network,

$$E(\{w\}) = \sum_p (y_o^{(p)} - y^{(p)})^2.$$

Since the obained y_o is a certain (weight-dependent) function of x, this method is a variant of the **least squares fit**, commonly used in data analysis. The difference is that in the standard least squares method the model function that we fit to the data has some simple analytic form (e.g. $f(x) = A + Bx$), while now it is some "disguised" weight-dependent function provided by the neural network.

7.2 ANNs for interpolation

To understand the fundamental idea, consider a network with just two neurons in the middle layer, with the sigmoid activation function:



The signals entering the two neurons in the middle layer are, in the notation of chapter *More layers*,

$$s_1^1 = w_{01}^1 + w_{11}^1 x,$$

$$s_2^1 = w_{02}^1 + w_{12}^1 x,$$

and the outgoing signals are, correspondingly,

$$\sigma\left(w_{01}^1 + w_{11}^1 x\right),$$

$$\sigma\left(w_{02}^1 + w_{12}^1 x\right).$$

Therefore the combined signal entering the output neuron is

$$s_1^1 = w_{01}^2 + w_{11}^2 \sigma\left(w_{01}^1 + w_{11}^1 x\right) + w_{21}^2 \sigma\left(w_{02}^1 + w_{12}^1 x\right).$$

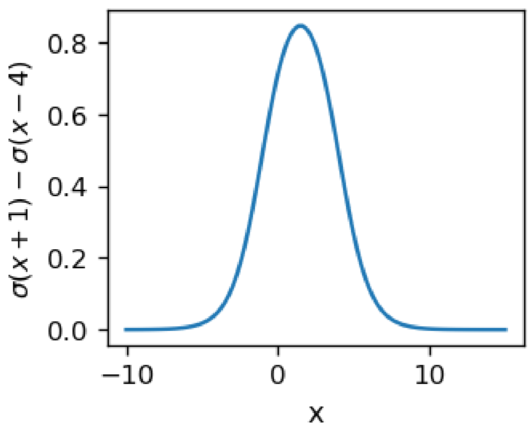
Taking, for an illustation, the weight values

$$w_{01}^2 = 0,\ w_{11}^2 = 1,\ w_{21}^2 = -1,\ w_{11}^1 = w_{12}^1 = 1,\ w_{01}^1 = -x_1,\ w_{02}^1 = -x_2,$$

where x_1 and x_2 are parameters with a natural interpretation explained below, we get

$s_1^1 = \sigma(x - x_1) - \sigma(x - x_2)$.

This function is shown in the plot below, with $x_1 = -1$ and $x_2 = 4$. It tends to 0 at $-\infty$, then grows with x to achieve a maximum at $(x_1 + x_2)/2$, and then decreases, tending to 0 at $+\infty$. At $x = x_1$ and $x = x_2$, the values are around 0.5.



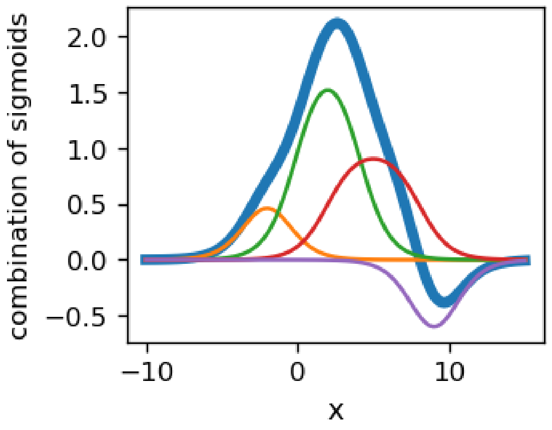
This is an important observation: We are able to form, with a pair of neurons, a "hump" signal located around a given value, here $(x_1 + x_2)/2 = 2$, and with a spread of the order of $|x_2 - x_1|$. Changing the weights, we are able to modify its shape, width, and height.

One may think as follows: Imagine we have many neurons to our disposal in the intemediate layer. We may join them in pairs, forming humps "specializing" in particular regions of coordinates. Then, adjusting the heights of the humps, we may readily approximate a given function.

In an actual fitting procedure, we do not need to "join the neurons in pairs", but make a combined fit of all parameters simutaneously, as we did in the case of classifiers. The example below shows a composition of 8 sigmoids,

$$f = \sigma(z+3) - \sigma(z+1) + 2\sigma(z) - 2\sigma(z-4) + \sigma(z-2) - \sigma(z-8) - 1.3\sigma(z-8) - 1.3\sigma(z-10).$$

In the figure, the component functions (the thin lines representing single humps) add up to a function of a rather complicated shape, marked with a thick line.



There is an important difference in ANNs used for function approximation compared to the binary classifiers discussed earlier. There, the answers were 0 or 1, so we were using a step function in the output layer, or rather its smooth sigmoid variant. For function approximation, the answers form a continuum from the range of the function values. For that reason, in the output layer we just use the **linear** function, i.e., we just pass the incoming signal through. Of course, sigmoids remain in the intermediate layers.

Output layer for function approximation

In ANNs used for function approximation, the activation function in the output layer is **linear**.

7.2.1 Backprop for one-dimensional functions

Minimization of the error function leads to the backprop algorithm (with a modified output layer), which we employ to fit our experimental data. Let us take the architecture:

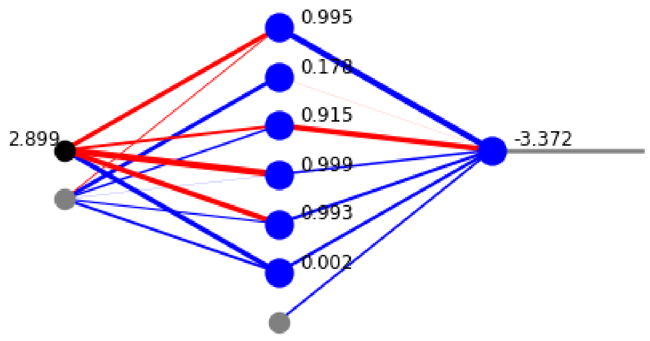
```
arch=[1,6,1]
```

and the weights

```
weights=func.set_ran_w(arch, 5)
```

As just discussed, the output is no longer between 0 and 1:

```
x=func.feed_forward_o(arch, weights,features[1],ff=func.sig,ffo=func.lin)
draw.plot_net_w_x(arch, weights,1,x);
```

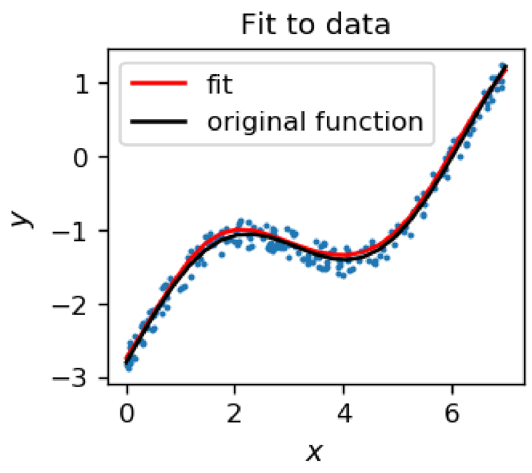


In the library module **func** we have the function for the backprop algorithm which allows for one activation function in the intermediate layers and a different one in the output layer. We carry out the training in two stages:

```
eps=0.02                           # initial learning speed
for k in range(30):                # rounds
    for p in range(len(features)): # loop over the data sample points
        pp=np.random.randint(len(features)) # random point
        func.back_prop_o(features,labels,pp,arch,weights,eps,
                         f=func.sig,df=func.dsig,fo=func.lin,dfo=func.dlin)
```

```
for k in range(400):               # rounds
    eps=0.999*eps                  # dicrease of the learning speed
    for p in range(len(features)): # loop over points taken in sequence
        func.back_prop_o(features,labels,p,arch,weights,eps,
                         f=func.sig,df=func.dsig,fo=func.lin,dfo=func.dlin)
```

which nicely does the job:



We note that the obtained red curve is very close to the function used to generate the data sample (black line). This shows that the approximation works. A construction of a quantitative measure (least square sum) is a topic of a homework problem.

7.3 Remarks

Note: The activation function in the output layer may be any function with values containing the values of the interpolated function, not necessarily linear.

More dimensions

To interpolate general functions of two or more arguments, one needs use ANNs with at least 3 neuron layers.

Tip: The number of neurons reflects the behavior of the interpolated function. If the function varies a lot, one needs more neurons, typically at least twice as many as the number of extrema.

Overfitting

There must be much more data for fitting than the network parameters, to avoid the so-called overfitting problem.

Exercises

1. Fit the data points generated by your favorite function (of one variable) with noise. Play with the network architecture.
2. Compute the sum of squared distances of the values of the data points and the corresponding approximating function, and use it as a measure of the goodness of the fit. Use it to test how the number of neurons in the network affects the result.
3. Use a network with more layers (at least 3 neuron layers) to fit the data points generated with your favorite two-variable function. Make two-dimensional contour plots for this function and for the function obtained from the neural network and compare the results (of course, they should be very similar if everything works).

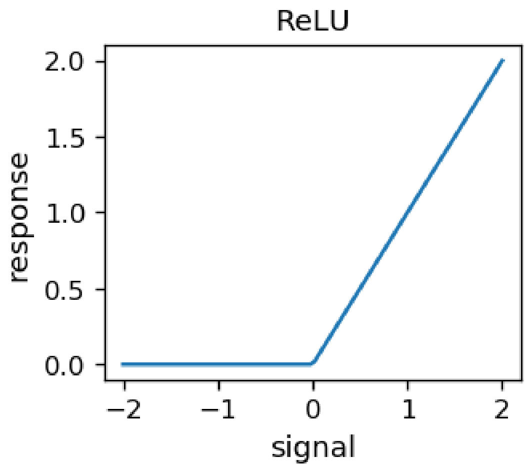
CHAPTER EIGHT

RECTIFICATION

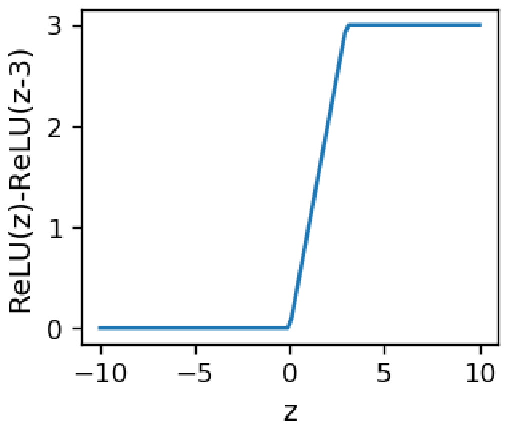
In the previous chapter we have made a hump function from two sigmoids, we may now ask a question: can we make a sigmoid as a linear combination (or simply difference) of some other functions. Then we could use these functions for activation of neurons in place of the sigmoid. The answer is yes. For instance, the **Rectified Linear Unit (ReLU)** function

$$\mathrm{ReLU}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{for } x < 0 \end{cases} = \max(x, 0)$$

does (approximately) the job. The a bit ackward name comes from electonics, where such a unit is used to cut out the negative values of an electric signal. Rectification means "straightening up". The plot of ReLU looks as follows:



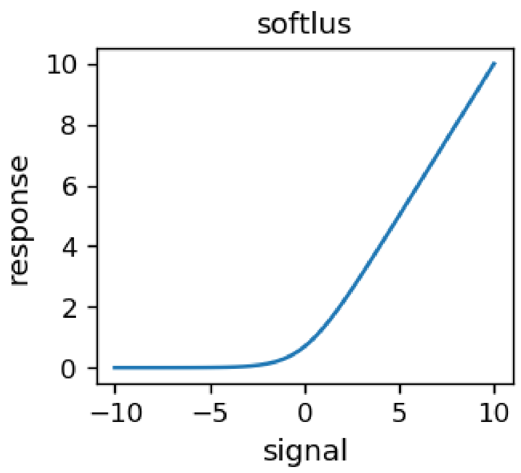
Taking a difference of two ReLU functions with shifted arguments yield, for example,



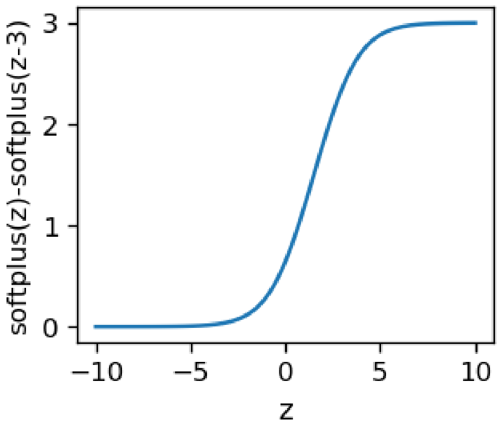
which looks pretty much as a sigmoid, apart for the sharp corners. One can make things smooth by taking a different function, the **softplus**,

$$\text{softplus}(x) = \log\left(1 + e^x\right),$$

which looks as



A difference of two softplus functions yields a result very similar to the sigmoid.



One may then use the ReLU of softplus, or a plethora of other similar functions, for the activation. Why one should actually do this will be dicussed later.

8.1 Interpolation with ReLU

We will now approximate our simulated data with an ANN with ReLU acivation in the intermediate layers (and a linear function is the output layer, as above). The fuctions are taken from the module **func**.

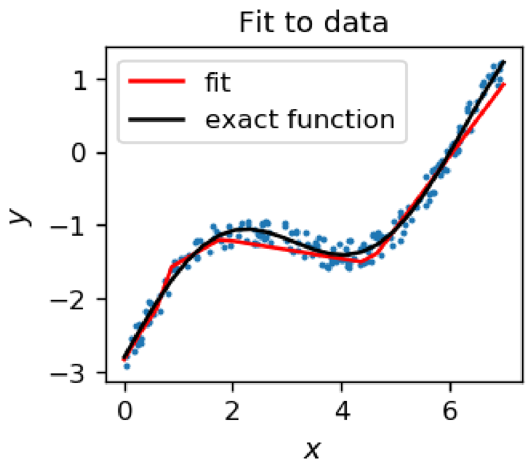
```
fff=func.relu
dfff=func.drelu
```

The network must now have more neurons:

```
arch=[1,30,1]                       # architecture
weights=func.set_ran_w(arch, 5)  # initialize weights randomly in [-2.5,2.5]
```

```
eps=0.0003           # small learning speed
for k in range(30): # rounds
    for p in range(len(features)): # loop over the data sample points
        pp=np.random.randint(len(features)) # random point
        func.back_prop_o(features,labels,pp,arch,weights,eps,
                         f=fff,df=dfff,fo=func.lin,dfo=func.dlin) # teaching
```

```
for k in range(600): # rounds
    eps=eps*.999
    for p in range(len(features)): # points in sequence
        func.back_prop_o(features,labels,p,arch,weights,eps,
                         f=fff,df=dfff,fo=func.lin,dfo=func.dlin) # teaching
```



We note again a quite safisfactory result, noticing that the plot of the fitting function is a sequence of straight lines, simply reflecting the feature of the ReLU activation function.

8.2 Classifiers with rectification

There are technical reasons in favor of using rectified functions rather than sigmoid-like ones in backprop. The derivatives of sigmoid are very close to zero apart for the narrow region near the threshold. This makes updating the weights unlikely, especially when going many layers back, as the small numbers multiply (this is known as the **vanishing gradient problem**). With rectified functions, the range where the derivative is large is big (for ReLU it holds for all positive coordinates), hence the problem is cured. For that reason, rectified functions are used in deep ANNs, where there are many layers, impossible to train when the activation function is of a sigmoid type. Application of rectified activation functions was one of the key tricks that allowed a breakthrough in deep ANNs in 2011.

On the other hand, with ReLU it may happen that weights are set into such values that many neurons become inactive, i.e. never fire for any input, and so are effectively eliminated. This is known as the "dead neuron" probem, which arises especially when the learning speed parameter is too high. A way to reduce the problem is to use an activation function which does not have a range with zero derivative, such as the leaky ReLU. Here we take it in the form

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0.1\,x & \text{for } x < 0 \end{cases} .$$

Next, we repeat our example with the classification of points in the circle.

We take the following architecture and initial parameters,

```
arch_c=[2,20,1]                   # architecture
weights=func.set_ran_w(arch_c,3)  # scaled random initial weights
eps=.01                           # initial learning speed
```

and run the algorithm in two stages: with leaky ReLu, and then with ReLU.

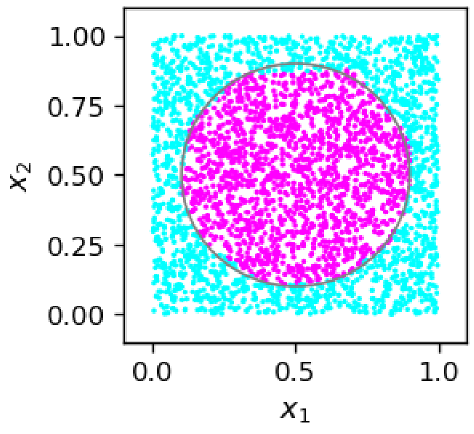
```
for k in range(300):     # rounds
    eps=.9999*eps        # decrease learning speed
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features_c)):                # loop over points
        func.back_prop_o(features_c,labels_c,p,arch_c,weights,eps,
            f=func.lrelu,df=func.dlrelu,fo=func.sig,dfo=func.dsig) # backprop
```

```
100  200  300
```

```
for k in range(700):     # rounds
    eps=.9999*eps        # decrease learning speed
    if k%100==99: print(k+1,' ',end='')             # print progress
    for p in range(len(features_c)):                # loop over points
        func.back_prop_o(features_c,labels_c,p,arch_c,weights,eps,
            f=func.relu,df=func.drelu,fo=func.sig,dfo=func.dsig) # backprop
```

```
100  200  300  400  500  600  700
```

The result is quite satistactory, showing that the method works. With the present architecture, not surprisingly, we can notice below a polygon approxiating the cirle.



8.3 General remarks on backprop

Conclude here our discussion of the supervised learning and the back proparation, we provide a number of remarks nd hints. First, in programmers life, bulding a well-functioning ANN, even for simple problems as used for illustrations up to now, can be a frustrating eperience! …

Initial conditions for minimization, basen of convergence, learning stategy, improvements of steepest descent …

Professional libraries, experience verified with success …

Exercises

1. Use various rectified activation functions for the binary classifiers and test then on various figures (in analogy to the example with the circe above).

CHAPTER

NINE

UNSUPERVISED LEARNING

Motto

teachers! leave those kids alone!

Supervised learning discussed in previous lectures needs a teacher or a training sample with labels, where we know a priori the characteristics of the data (e.g., in our example, whether the point is inside or outside the circle).

However, this is quite a special situation, because most often the data that we encounter do not have pre-assigned labels and "are what they are". Also, from the neurobiological or methodological point of view, we learn many facts and activities "on an ongoing basis", classifying and then recognizing them, whilst the process goes on without any external supervision.

Imagine an alien botanist who enters a meadow and encounters various species of flowers. He has no idea what they are and what to expect at all, as he has no prior knowledge on earthly matters. After finding the first flower, he records its features: color, size, number of petals, etc. He goes on, and finds a different flower, and records its features, and so on with the next flowers. At some point, however, he finds a flower that he already had met. More precisely, its features will be close, though not identical (the size may easily differ somewhat, so the color, etc.), to the previous instance. Hence he concludes it belongs to the same category. The exploration goes on, and new flowers either start a new category, of join one already present. At the end he has a catalog of flowers and now he can assign names (labels) to each species: corn poppy, bluebottle, mullein,… They are useful in sharing the knowledge with ohers, as they summarize, so to speak, the features of the flower. Note, however, that these labels have actually never been used in the meadow exploration (learning) process.

Formally, this problem of **unsupervised learning** is related to data classification (division into categories, or **clusters**, i.e. subsets of tah sample where the suitably defined distances between individual data are small, smaller than the assumed distances between clusters). Colloquially speaking, we are looking for similarities between individual data points and try to divide the sample into groups of similar objects.

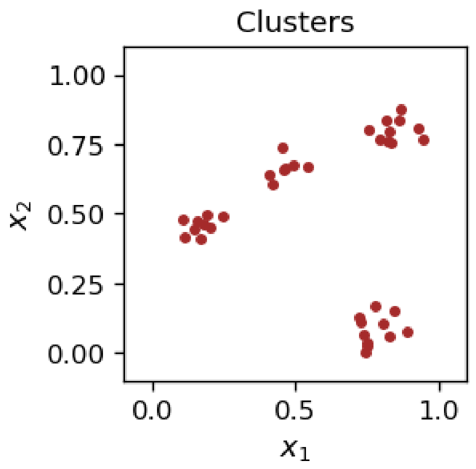
9.1 Clusters of points

```
def pA():
    return [np.random.uniform(.75, .95), np.random.uniform(.7, .9)]

def pB():
    return [np.random.uniform(.4, .6), np.random.uniform(.6, .75)]

def pC():
    return [np.random.uniform(.1, .3), np.random.uniform(.4, .5)]

def pD():
    return [np.random.uniform(.7, .9), np.random.uniform(0, .2)]
```

Let us create data samples with a few points from each category:

```
samA=np.array([pA() for _ in range(10)])
samB=np.array([pB() for _ in range(7)])
samC=np.array([pC() for _ in range(9)])
samD=np.array([pD() for _ in range(11)])
```

Our data looks like this:



If we show the above picture to someone, he will undoubtedly state that the are four clusters. But what algorithm is he using to determine this? We will construct such an algorithm shortly and will be able to carry out clusterization. For the moment let us jump ahead and assume we know the clusters. Clearly, in our example the clusters are well defined, i.e. visibly separated from each other.

One can represent clusters with **representative points** that lie somewher within the cluster. For example, one could take an item belonging to a given cluster as its representative, or in each cluster one can evaluate the mean position of its points and use it as representative points:

```
rA=[st.mean(samA[:,0]),st.mean(samA[:,1])]
rB=[st.mean(samB[:,0]),st.mean(samB[:,1])]
rC=[st.mean(samC[:,0]),st.mean(samC[:,1])]
rD=[st.mean(samD[:,0]),st.mean(samD[:,1])]
```

We add thus defined characteristic points to our graphics. For visual convenience, we assign a color for each category (after having the clusters, we may assign labels, and the color here serves this purpose).

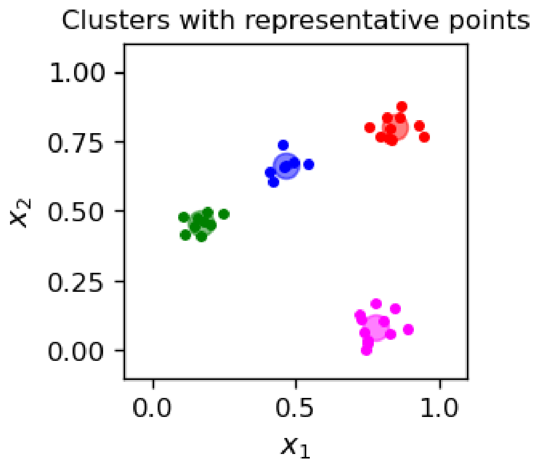
```
col=['red','blue','green','magenta']
```

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("Clusters with representative points",fontsize=10)
plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)

plt.scatter(samA[:,0],samA[:,1],c=col[0], s=10)
plt.scatter(samB[:,0],samB[:,1],c=col[1], s=10)
plt.scatter(samC[:,0],samC[:,1],c=col[2], s=10)
plt.scatter(samD[:,0],samD[:,1],c=col[3], s=10)

plt.scatter(rA[0],rA[1],c=col[0], s=90, alpha=0.5)
plt.scatter(rB[0],rB[1],c=col[1], s=90, alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
plt.scatter(rC[0],rC[1],c=col[2], s=90, alpha=0.5)
plt.scatter(rD[0],rD[1],c=col[3], s=90, alpha=0.5)

plt.xlabel('$x_1$',fontsize=11)
plt.ylabel('$x_2$',fontsize=11);
```



9.2 Voronoi areas

Having the situation as in the figure above, i.e. some representative points present, we can divide the entire plane into areas according to the following Voronoi criterion, which is a simple geometric notion:

Voronoi areas

Consider point P in a metric space, where we have a number representative points (also called the Voronoi points). The representative point R closest to P determines its category. All such points P form a Voronoi area of R.

Let us define the color of the point P in our plane as the color of the nearest representative point. For this we first need (the square of) a distance function (here: Euclidean) between two points in 2-dim. space:

```
def eucl(p1,p2): # square of the Euclidean distance
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2
```

Then we find the nearest representative point and determine its color:

```
def col_char(p):
    dist=[eucl(p,rA),eucl(p,rB),eucl(p,rC),eucl(p,rD)]
    ind_min = np.argmin(dist) # index of the nearest point
    return col[ind_min]
```

```
# e.g.
col_char([.5,.5])
```

The result is following, with straight-line boundaries between naighboring areas:

```
plt.figure(figsize=(3.2,3.2))
plt.title("Voronoi areas",fontsize=11)
plt.xlim(-.1,1.1)
plt.ylim(-.1,1.1)

for x1 in np.linspace(0,1,70): # 70 points in x
    for x2 in np.linspace(0,1,70): # 70 points in y
        plt.scatter(x1,x2,c=col_char([x1,x2]), s=50, alpha=0.6, edgecolors='none')

plt.scatter(samA[:,0],samA[:,1],c='black', s=20)
plt.scatter(samB[:,0],samB[:,1],c='black', s=20)
plt.scatter(samC[:,0],samC[:,1],c='black', s=20)
plt.scatter(samD[:,0],samD[:,1],c='black', s=20)

plt.scatter(rA[0],rA[1],c='black', s=150, alpha=.6)
plt.scatter(rB[0],rB[1],c='black', s=150, alpha=.6)
plt.scatter(rC[0],rC[1],c='black', s=150, alpha=.6)
plt.scatter(rD[0],rD[1],c='black', s=150, alpha=.6)

plt.xlabel('$x_1$',fontsize=18)
plt.ylabel('$x_2$',fontsize=18);
```

Note: A practical message here is that once we have the characteristic points, we can use Voronoi's criterion for classification of data

9.3 Naive clusterization

Now the "real" botanist's problem: imagine we have our sample, but we know nothing about how its points were generated (we do not have any labels A, B, C, D, nor colors of the points). Moreover, the data is mixed, i.e., the data points appear in a random order. So we merge our points,

```
alls=np.concatenate((samA, samB, samC, samD))
```

and shuffle them,

```
np.random.shuffle(alls)
```

The data visualization looks as in the first plot of thsi chapter.

We now want to somehow create representative points, but a priori we don't know where they should be, or even how many of them there are. Very different strategies are possible here. Their common feature is that the position of the representative points is updated as the sample data is processed.

Let us start with just one representative point, $\vec{R}$. Not very ambitious, but in the end we will at least know some mean characteristics of the sample. The initial position position is $R = (R_1, R_2)$, a two dimensional vector in $[0, 1] \times [0, 1]$. After reading a data point P with coordinates (x_1^P, x_2^P), R changes as follows:

$$(R_1, R_2) \to (R_1, R_2) + \varepsilon(x_1^P - R_1, x_2^P - R_2),$$

or in the vector notation

$$\vec{R} \to \vec{R} + \varepsilon(\vec{x}^P - \vec{R}).$$

The step is repeated for all points of the sample, and then many such round may be carried out. As in the previous chapters, ε is the learning rate that (preferably) decreases as the algorithm proceeds. The above formula realizes the "snapping" of the point $\vec{R}$ by the data point $\vec{P}$.

The following code implements the above prescription:

```
R=np.array([np.random.random(),np.random.random()])
print("initial location:")
print(np.round(R,3))
print("rounds  location")

eps=.5

for j in range(50): # rounds
    eps=0.85*eps    # decrease the update speed
    np.random.shuffle(alls) # reshuffle the sample
    for i in range(len(alls)): # loop over points of the whole sample
        R+=eps*(alls[i]-R) # update/learning
    if j%5==4: print(j+1, "    ",np.round(R,3))
```

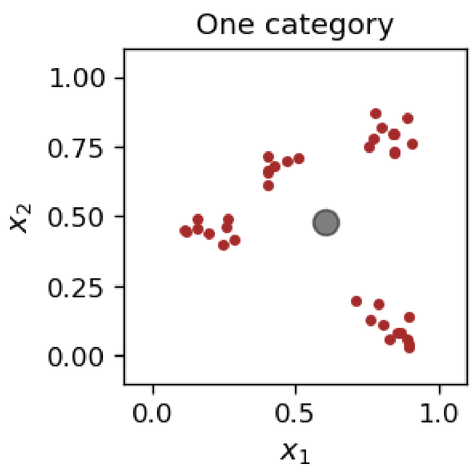
```
initial location:
[0.005 0.512]
rounds  location
5      [0.553 0.517]
10      [0.637 0.503]
15      [0.612 0.481]
20      [0.602 0.48 ]
25      [0.604 0.481]
30      [0.603 0.482]
35      [0.603 0.482]
40      [0.603 0.482]
45      [0.603 0.482]
50      [0.603 0.482]
```

We can see that the position of the characteristic point converges. Actually, it becomes very close to the average of the locatio of all points,

```
R_mean=[st.mean(alls[:,0]),st.mean(alls[:,1])]
print(np.round(R_mean,3))
```

```
[0.602 0.481]
```

We have decided a priori to have one category, and here is our plot of the result fot the characeristic point, indicated with a gray blob:



Let us try to generalize the above algorithm for the case of several ($n_R > 1$) representative points.

- We initialize randomly representative vectors $\vec{R}^i$, $i = 1, \dots, n_R$.
- Round: We take the sample points P one by one and update only the **closest** point R^m to the point P in a given step:

$$\vec{R}^m \to \vec{R}^m + \varepsilon(\vec{x} - \vec{R}^m).$$

- The position of the other representative points remains the same. This strategy is called the **winner-take-all**.
- We repeat the rounds, reducing the learning speed ε each time.

Note: The **winner-take-all** strategy is an important concept in ANN modeling. The competing neurons in a layer fight for the signal, and the one that wins, takes it all (its weighs get updated), while the loosers get nothing.

So let's consider two representative points that we initialize randomly:

```
R1=np.array([np.random.random(), np.random.random()])
R2=np.array([np.random.random(), np.random.random()])
```

Then we carry out the above algorithm. For each data point we find the nearest representative point out of the two, and update only this one:

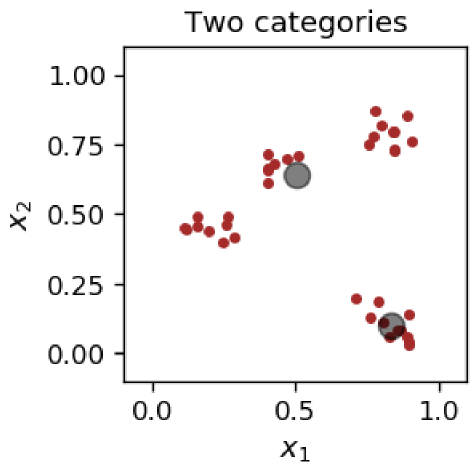
```
print("initial locations:")
print(np.round(R1,3), np.round(R2,3))
print("rounds  locations")

eps=.5

for j in range(40): # rounds
    eps=0.85*eps
    np.random.shuffle(alls)
    for i in range(len(alls)):
        p=alls[i] # data point
#         print(p)
        dist=[func.eucl(p,R1), func.eucl(p,R2)] # squares of distances
        ind_min = np.argmin(dist)     # minimum
        if ind_min==0: # if R1 closer to the new data point
            R1+=eps*(p-R1)
```

(continues on next page)

(continued from previous page)

```
        else:           # if R2 closer ...
            R2+=eps*(p-R2)

    if j%5==4: print(j+1,"    ", np.round(R1,3), np.round(R2,3))
```

```
initial locations:
[0.905 0.549] [0.419 0.255]
rounds  locations
5      [0.836 0.104] [0.476 0.613]
10      [0.837 0.107] [0.559 0.678]
15      [0.838 0.101] [0.509 0.645]
20      [0.837 0.102] [0.504 0.643]
25      [0.837 0.102] [0.507 0.644]
30      [0.837 0.102] [0.505 0.643]
35      [0.837 0.102] [0.505 0.642]
40      [0.837 0.102] [0.504 0.642]
```



One of the characteristic poits “specializes” in the lower right cluster, and the other in the remaining three.

We continue, anologously, with four representative points.

Running the case for four categories, we notice that it does not always give the correct answer. Quite often one of the representative points is not updated at all and becomes the so-called **dead body**. This is because the other representative points always “win”, i.e. one of them is always closer to each point of the sample than the “corpse”.

When we set up five characteristic points, several situation may occur, as shown in the figure below. Sometimes, depending on the initializtion, a cluster is split into two smaller ones, sometimes dead bodies occur.

Enforcing more representative points leads to the formation of dead bodies even more often. Of course, we may disregard them, but the example shows that the current startegy is problematic.

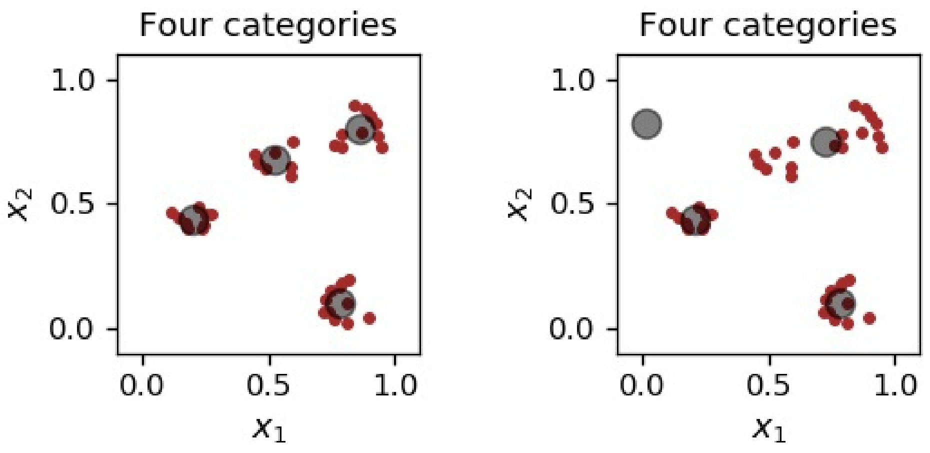


Fig. 9.1: Left: proper characteristic points. Right: one "dead body".

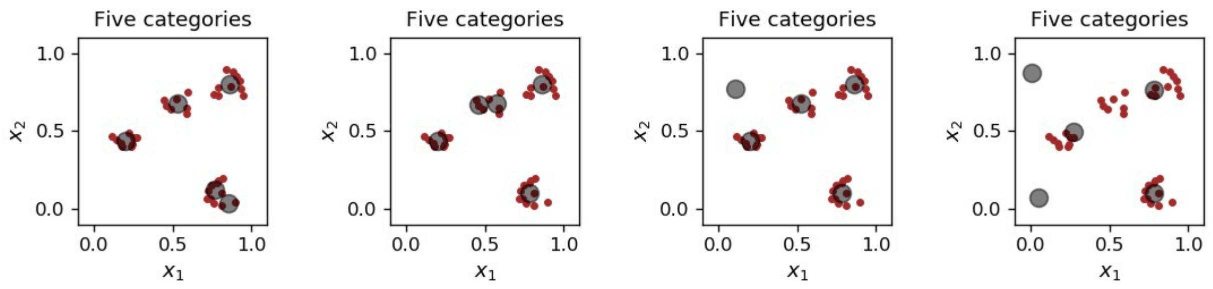


Fig. 9.2: From left to right: 5 characteristic points with one cluster split into two, with another cluster split into two, one dead body, and two dead bodies.

9.4 Clustering scale

In the previous section we were trying to guess from the outset how many clusters there are in the data. This lead to problems, also many times we do not really know how many clusters there are. Actually, up to now we have not even defined what precisely a cluster is, using some intuition only. This intuition told us that the points in the same cluster must be close to one another, or close to a characteristi point, but how close? Actually, the definition must involve a scale telling us "how close is close". Fo instance, in our example we may take a scale of about 0.2, where there are 4 clusters, but we may take a smaller one and resolve the bigger clusters inti smaller ones, as in the left panels of Fig. 9.2.

Definition of cluster

A cluster of scale d associated with a characteristic point R is the set of data points P, whose distance from R is less than d, whereas the distance from other characteristic points is $\geq d$. The characteristic points must be selected in such a way that each data point belongs to a cluster, and no characteristic point is a dead body (i.e., its cluster must contain at least one data point).

Various strategies can be used to implement this prescription. We use here the **dynamical clusterization**, where a new cluster/representative point is created whenever an encoutered data point is farther than d from any present characteristic point up to now.

Dynamical clusterization

1. Set the clustering scale d and the learning speed ε. Shuffle the sample.
2. Read the first data point P_1 and set the first characteristic point $R^1 = P_1$. Add it to an array R. Mark P_1 as belonging to cluster 1.
3. Read the next data points P. If the distance of P to the **closest** characteristic point, R^m, is $\leq d$, then
   - mark P as belonging to cluster m.
   - move R^m towards P with the learning speed ε.Otherwise, add to R a new characteristic point a location of the point P.
4. Repeat from 2. until all the data points are read.
5. Repeat from 2. a number of rounds, decreasing each time ε. The result is a division of the sample into a number of clusters, and ghe location of corresponding charactristic points. The result may depend on the reshuffling, hence does not have to the same when the procedure is repeated.

A Python implementation finding dynamically the representative points is following:

```
d=0.2  # clustering scale
eps=0.5 # initial learning speed

for r in range(20):                 # rounds
    eps=0.85*eps                    # decrease the learning speed
    np.random.shuffle(alls)         # shuffle the sample
    if r==0:                        # in the first round
        R=np.array([alls[0]])       # R - array of representative points
                                    # initialized to the first data point
    for i in range(len(alls)):      # loop over the sample points
        p=alls[i]                   # new data point
        dist=[func.eucl(p,R[k]) for k in range(len(R))]
         # array of squares of distances of p from the current repr. points in R
        ind_min = np.argmin(dist) # index of the closest repr. point
```

(continues on next page)

(continued from previous page)

```
        if dist[ind_min] > d*d:    # if its distance square > d*d
                                   # dynamical creation of a new category
            R=np.append(R, [p], axis=0)     # add new repr. point to R
        else:
            R[ind_min]+=eps*(p-R[ind_min]) # otherwise, apdate the "old" repr. point

print("Number of representative points: ",len(R))
```

```
Number of representative points:  4
```

The outcome for various values of the clustering scale d is shown in Fig. 9.3. At very low values of d, smaller than the minimum separation betwenn the points, there are as many clusters as the data points. Then, as we increase d, the number of clusters decreases. At very large d, order of the of the span of the sample, there is only one cluster.

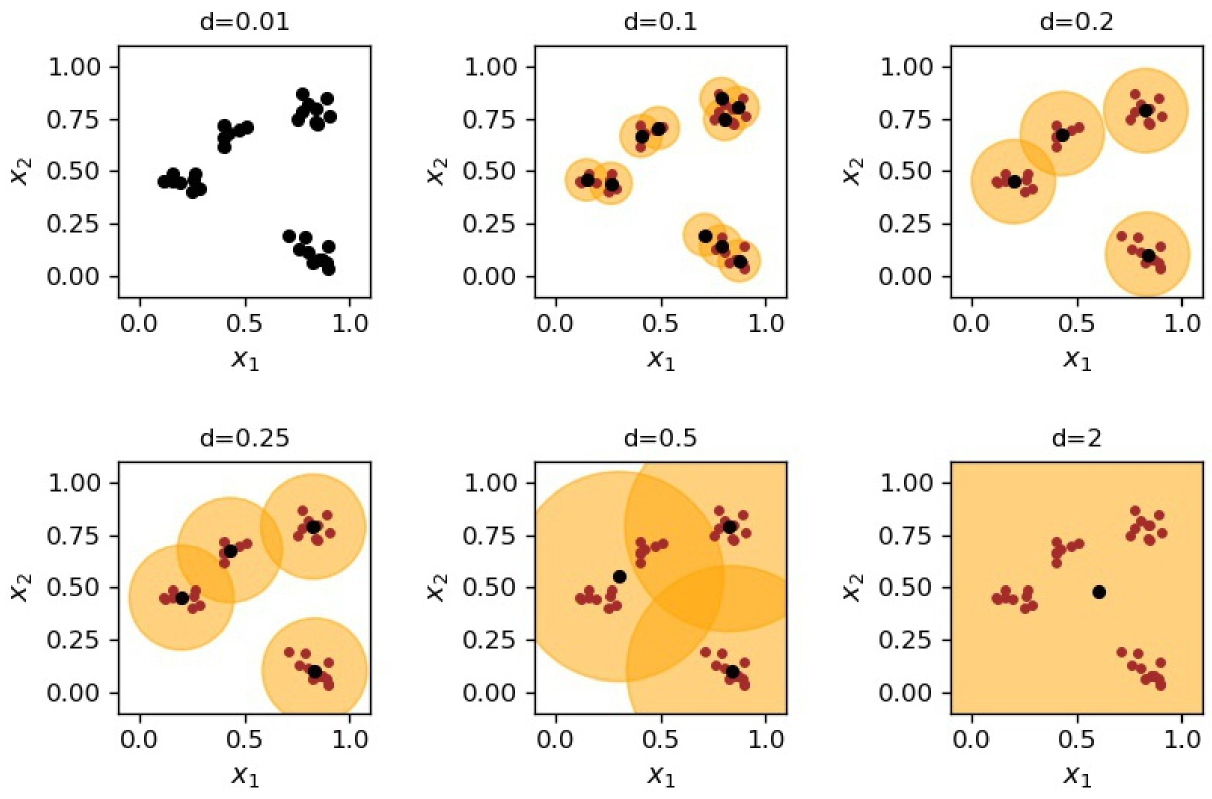


Fig. 9.3: Dynamical clustering for various values of the scale d.

Certainly, an algorithm will not tell us which clustering scale to use. The proper value depends on the nature of the problem. Recall our botanist. If he used a very small d, he would get as many categories as there are flowers in the meadow, as all flowers, even of the same species, are slightly different from one another. That would be useless. On the other extreme, if his d is too large, then the classification is too crude. Something in between is just right!

Labels

After forming the clusters, we may assign them **labels** for convenience. They are not used in the learning (cluster formation) process.

Having determined the clusters, we have a **classifier**. We may use it in a two-fold way:

- continue the dynamical update as new data are encountered, or
- “close” it, and see where the new data falls in.

In the first case, we assign the corresponding cluster label to the data point (our botanist knows what new flower he found), or initiate a new category if the point does not belong to any of the existing clusters. This is just a continuation of the dynamical algorithm descibed above on the new incoming data

In the latter case (we bought the ready botanist’s catalogue), a data point may

- belong to a cluster (we know its label),
- fall outside any cluster, then we just do not know what it is, or
- fall into an overlapping region of two or more clusters (cf. Fig. 9.3, where we only get “partial” classification.

9.4.1 Interpretation via steepest descent

Let us denote a given cluster with C_i, $i = 1, ..., n$, where n is the total number of clusters. The sum of the squared distances of data points in C_i to its representative point R^i is

$$\sum_{P \in C_i} |\vec{R}^i - \vec{x}^P|^2.$$

Summing up over all clusters, we obtain a function analogous to the previously discussed error function:

$$E(\{R\}) = \sum_{i=1}^{n} \sum_{P \in C_i} |\vec{R}^i - \vec{x}^P|^2.$$

Its derivative with respect to $\vec{R}_i$ is

$$\frac{\partial E(\{R\})}{\partial \vec{R}^i} = 2 \sum_{P \in C_i} (\vec{R}^i - \vec{x}^P).$$

The steepest descent method results **exactly** in the recipe used in the dynamic clasterization algorithm presented above, i.e.

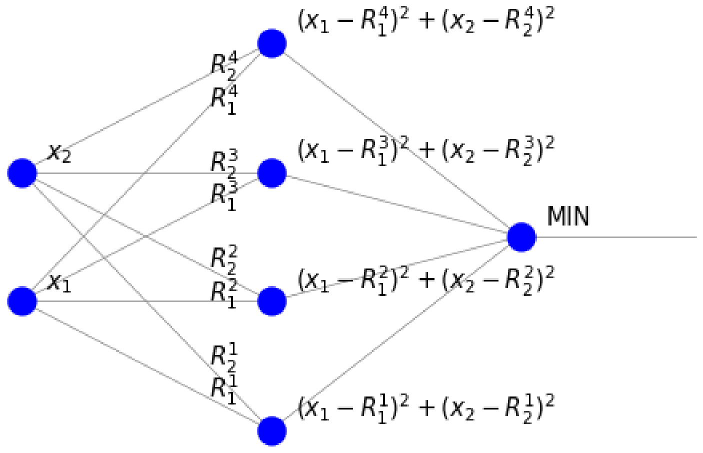
$$\vec{R} \to \vec{R} - \varepsilon(\vec{R} - \vec{x}^P).$$

To summarize, the algorithm used here actually involves the steepest descent method for the function $E(\{R\})$, as discussed in previous lectures.

Note: Note, however, that the minimization used in the present algorithms also takes into account different division of points into clusters. In particular, a given data point may change its cluster assignment during the execution of the algorithm. This happens when the closest representative point is changed.

9.5 Interpretation via neural networks

We will now interpret the above unsupervised learning algorithm with the winner-take-all strategy in the neural network language.


$(x_1 - R_1^4)^2 + (x_2 - R_2^4)^2$
$(x_1 - R_1^3)^2 + (x_2 - R_2^3)^2$
$(x_1 - R_1^2)^2 + (x_2 - R_2^2)^2$
$(x_1 - R_1^1)^2 + (x_2 - R_2^1)^2$
x_2
x_1
R_2^4
R_1^4
R_2^3
R_1^3
R_2^2
R_1^2
R_2^1
R_1^1
MIN

Our example network has four neurons in the intermediate neuron layer, each corresponding to one characteristic point $\vec{R}^i$. The weights are the coordinates of $\vec{R}^i$. There is one node in the output layer. We note significant differences from the perceptron discussed earlier.

- There are no threshold nodes.
- In the intermediate layer, the signal equals the distance squared of the input from the correspoding characteristic point. It is not a weighted sum.
- The node in the last layer (MIN) indicates in which neuron of the intermediate layer the signal is the smallest, i.e., where we have the shortest distance. Hence it works as a control unit selecting the minimum.

During (unsupervised) learning, an input point "attracts" the closest characteristic point, whose weights are updated.

The application of the above network classifies the point with the coordinates (x_1, x_2), assigning it the index of the representative point of a given category (here it is the number 1, 2, 3, or 4).

9.5.1 Representation with spherical coordinates

Even with our vast "mathematical freedom", calling the above system a neural network is quite abusive, as it seems very far away from any neurobiological pattern. In particular, the use of a (non-linear) signal of the form $\left(\vec{R}^i - \vec{x}\right)^2$ contrasts with the perceptron, where the signal entering the neurons is a (linear) weighted sum of inputs, i.e.

$$s^i = x_1 w_1^i + x_2 w_2^i + ... + w_1^m x_m = \vec{x} \cdot \vec{w}^i.$$

We can alter our problem with a simple geometric construction to make it similar to the perceptron principle. For this purpose we enter a (spurious) third coordinate defined as

$$x_3 = \sqrt{r^2 - x_1^2 - x_2^2},$$

where r is chosen such that for all data points $r^2 \geq x_1^2 + x_2^2$. From the construction $\vec{x} \cdot \vec{x} = x_1^2 + x_2^2 + x_3^2 = r^2$, so the data points lie on the hemisphere ($x_3 \geq 0$) of radius r. Similarly, for the representative points we introduce:

$$w_1^i = R_1^i, \; w_2^i = R_2^i, \; w_3^i = \sqrt{r^2 - (R_1^i)^2 - (R_2^i)^2}.$$

It is geometrically obvious that two points in a plane in coordinates 1 and 2 are close to each other if and only if their extensions to the hemisphere are close. We support this statement with a simple calculation:

The dot product of two points $\vec{x}$ and $\vec{y}$ on a hemisphere can be written as

$$\vec{x} \cdot \vec{y} = x_1 y_1 + x_2 y_2 + \sqrt{r^2 - x_1^2 - x_2^2}\sqrt{r^2 - y_1^2 - y_2^2}.$$

For simplicity, let us consider a situation when $x_1^2 + x_2^2 \ll r^2$ and $y_1^2 + y_2^2 \ll r^2$, i.e. both points lie near the pole of the hemisphere. Using your knowledge of mathematical analysis

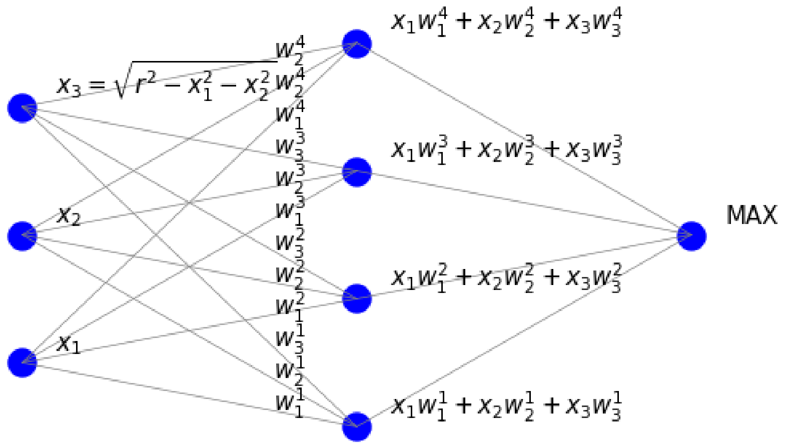
$$\sqrt{r^2 - a^2} \simeq r - \frac{a^2}{2r}, \quad a \ll r,$$

hence

$\vec{x} \cdot \vec{y} \simeq x_1 y_1 + x_2 y_2 + \left(r - \frac{x_1^2+x_2^2}{2r}\right)\left(r - \frac{y_1^2+y_2^2}{2r}\right)$
$\simeq r^2 - \frac{1}{2}(x_1^2 + x_2^2 + y_1^2 + y_2^2) + x_1 y_1 + x_2 y_2$
$= r^2 - \frac{1}{2}[(x_1 - x_2)^2 + (y_1 - y_2)^2].$

So it is (for points close to the pole) the constant r^2 minus half the square of the distance between the points (x_1, x_2) and (y_1, y_2) on the plane! It then follows that instead of finding a minimum distance for points on the plane, as in the previous algorithm, we can find a maximum scalar product for their 3D extensions to a hemisphere.

With the extension of the data to a hemisphere, the appropriate neural network can be viewed as follows:


$x_1w_1^4 + x_2w_2^4 + x_3w_3^4$
$x_3 = \sqrt{r^2 - x_1^2 - x_2^2}$
w_3^4
w_2^4
w_1^4
w_3^3
w_2^3
w_1^3
w_3^2
w_2^2
w_1^2
w_3^1
w_2^1
w_1^1
x_2
x_1
$x_1w_1^3 + x_2w_2^3 + x_3w_3^3$
MAX
$x_1w_1^2 + x_2w_2^2 + x_3w_3^2$
$x_1w_1^1 + x_2w_2^1 + x_3w_3^1$

Thanks to our efforts, the signal in the intermediate layer is now just a dot product of the input and the weights, as it should be in the artificial neuron. The unit in the last layer (MAX) indicates where the dot product is largest.

This MAX unit is still problematic to interpret within our present framework. Actually, it is possible, but requires going beyond feed-forward networks. When the neurons in the layer can communicate (recurrent Hopfield networks), they can compete, and with proper feed-back it is possible to enforce the winner-take-all mechanism.

Hebbian rule

On the conceptul side, we here touch upon a very important and intuitve principle in neural networks, known as the Hebbian rule/ Essentially, it applies the truth “What is used, gets stronger” no synaptic connections. A repeated use of a connection makes it stronger.

In our formulation, if a signal passes through a given connection, its weight changes accordingly, and other connections remain the same. The process takes place in an unsupervised manner and its implementation is biologically well motivated.

On the other hand, it is difficult to find a biological justification for backpropagation in supervised learning, where all weights are updated, also in layers very distant from the output. According to many researchers, it is rather a mathematical concept (but nevertheless extremely useful)

9.5.2 Scalar product maximization

The algorithm is now as follows:

- Extend the points from the sample with the third coordinate, $x_3 = \sqrt{r^2 - x_1^2 - x_2^2}$, choosing appropriately large r so that $r^2 > x_1^2 + x_2^2$ for all sample points.
- Initialize the weights such that $\vec{w}_i \cdot \vec{w}_i = r^2$.

Then loop over the data points:

- Find the neuron in the intermediate layer for which the dot product $x \cdot \vec{w}_i$ is the largest. Change the weights of this neuron according to the recipe

$$\vec{w}^i \to \vec{w}^i + \varepsilon(\vec{x} - \vec{w}^i).$$

- Renormalize the updated weight vector $\vec{w}_i$ such that $\vec{w}_i \cdot \vec{w}_i = r^2$:

$$\vec{w}^i \to \vec{w}^i \frac{r}{\sqrt{\vec{w}_i \cdot \vec{w}_i}}.$$

The remaining steps of the algorithm, such as determining the initial positions of the representative points, their dynamic creation as they encounter successive data points, etc., remain as in the previously discussed procedure.

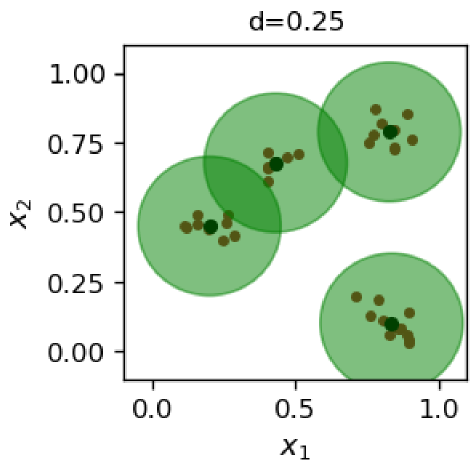
The generalization for n dimensions is obvious: we enter an additional coordinate

$$x_{n+1} = \sqrt{r^2 - x_1^2 - ... - x_n^2},$$

hence we have a point on the hyper-hemisphere $x_1^2 + \cdots + x_n^2 + x_{n+1}^2 = r^2$, $x_{n+1} > 0$.

```
d=0.25
eps=.5

rad=2 # radius r

for r in range(25):
    eps=0.85*eps
    np.random.shuffle(alls)
    if r==0:
        p=alls[0]
        R=np.array([np.array([p[0],p[1],np.sqrt(rad**2 - p[0]**2 - p[1]**2)])])
    for i in range(len(alls)):
        p=np.array([alls[i][0], alls[i][1],
                    np.sqrt(rad**2 - alls[i][0]**2 - alls[i][1]**2)])
          # rozszerzenie do półsfery
        dist=[np.dot(p,R[k]) for k in range(len(R))]
        ind_max = np.argmax(dist)                    # maximum
        if dist[ind_max] < rad**2 - d**2/2:
             R=np.append(R, [p], axis=0)
        else:
            R[ind_max]+=eps*(p-R[ind_max])

print("Number of representative points: ",len(R))
```

```
Number of representative points:  4
```



We can see that the dot product maximization algorithm yields an almost exactly the same result as the distance squared minimization (cf. Fig. 9.3.

Exercises

1. The city (Manhattan) metric is defined as $d(\vec{x}, \vec{y}) = |x_1 - y_1| + |x_2 - y_2|$ for points $\vec{x}$ and $\vec{y}$. Repeat the simulations of this lecture with this metric. Draw conclusions.
2. Run the classification algorithms for more categories in the data sample (generate your own sample).
3. One-dimensional data variant: Consider the problem of clustering points of different grayscale. To do this, enter the grayscale as s from the $[0, 1]$ range and the character data vector $(s, \sqrt{1 - s^2})$ which is normalized to 1 (such points lie on a semicircle). Generate a sample of these points and run dynamical clusterization.

CHAPTER

TEN

SELF ORGANIZING MAPS

A very important and ingenious application of unsupervised learning are the so-called **Kohonen nets** (Teuvo Kohonen, i.e. **self-organizing mappings (SOM)**. Consider a mapping f between a **discrete** k-dimensional set (**grid**) of neurons and n-dimensional input data D (continuous or discrete),

$$f : N \to D.$$

Since N is descrete, each neuron carrries an index consisting of k natural numbers, denoted as $\bar{i} = (i_1, i_2, ..., i_k)$. Typically, the dimensions satisfy $n \geq k$. When $n > k$, one talks about **reduction of dimensionality**, as the input space D has more dimensions than the grid of neurons.

Two examples of such networks are visualized in {numref]`koh-fig`. The left panel shows a 2-dim. input space D, and a one dimensional grid on neurons. The input point (x_1, x_2) enters all the neuron of the grid, and one of them (with best-suited weights) becomes the winner (red dot). The gray oval indicates the neighborhood of the winner. The right panel shows an analogous situation for the case of a 3-dim. input and 2-dim. grid of neurons. Here, for clarity, we only indicated the edges entering the winner, but they also enter all the other neurons, as in the left panel.

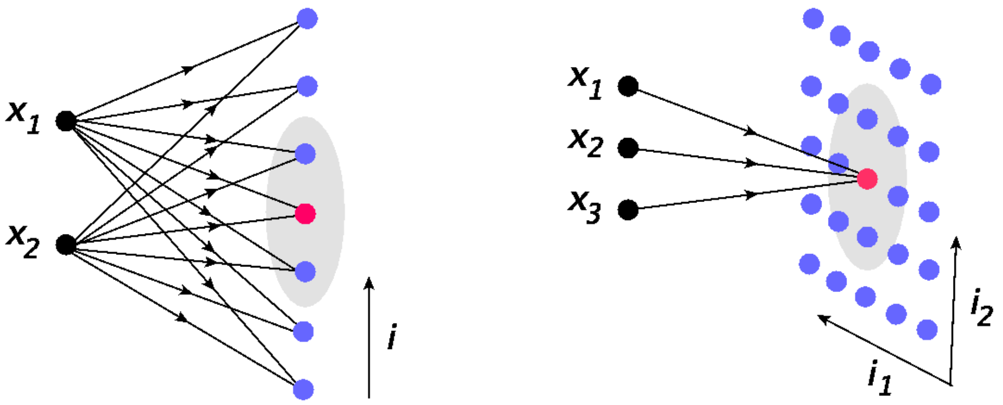


Fig. 10.1: Example of Kohonen networks. Left: 1-dim. grid of neurons N and 2-dim. input space D. Right: 2-dim. grid of neurons N and 3-dim. input space D. The red dot indicates the winner, and the gray oval marks its neighborhood.

One defines the neuron **proximity function**, $\phi(\bar{i}, \bar{j})$, which assigns, to a pair of neurons, a real number depending on their relative position in the grid. This function must decrease with the distance between the neuron indices. A popular choice is a Gaussian,

$$\phi(\bar{i}, \bar{j}) = \exp\left[-\frac{(i_1 - j_1)^2 + ... + (i_k - j_k)^2}{2\delta^2}\right],$$

where δ is the **neighborhood radius**. For a 1-dim. grid it becomes $\phi(i, j) = \exp\left[-\frac{(i-j)^2}{2\delta^2}\right]$.

10.1 Kohonen's algorithm

The set up of Kohonen's algorithm is similar to the unsupervised learning discussed in the previous chapter. Each neuron $\bar{i}$ obtains weights $f\left(\bar{i}\right)$, which are elements of D, i.e. form n-dimensional vectors. One may simply think of this as placing the neurons in some locations in D. When an input point P form D is fed into the network, one looks for a closest neuron, which becomes the **winner**, exactly as in the algorithm from section *Interpretation via neural networks*. However, here comes an important difference: Not only the winner is attracted (updated) a bit towards P, but also its neighbors, to a lesser and lesser extent the farther they are from the winner.

Winner-take-most strategy

Kohonen's algorithm involves the "winner take most" strategy, where not only the winner neuron is updated (as in the winner-take-all case), but also its neighbors. The neighbors update is strongest for the nearest neighbors, and gradually weakens with the distance from the winner.

Kohnen's algorithm

1. Initialize (for instance randomly) n-dimensional weights w_i, $i-1,\dots,m$ for all the m neurons in the grid. Set an initial neighborhood radius δ and an initial learning speed ε.
2. Choose (randomly) a data point P wigh coordinates x from the input space (possibly with an appropriate probability distribution).
3. Find the $\bar{l}$ neuron (the winner) for which the distance from P is the smallest.
4. The weights of the winner and its neighbors are updated according to the **winner-take-most** recipe:

$$w_{\bar{i}} \to w_{\bar{i}} + \varepsilon\phi(\bar{i},\bar{l})(x - w_{\bar{i}}), \qquad i = 1, ..., m.$$

1. Loop from 1. for a specified number of points.
2. Repeat from 1. in rounds, until a satisfactory result is obtained or the maximum number of rounds is reached. In each round **reduce** ε and δ according to a chosen policy.

The way the reduction of ε and δ is done is very important for the desired outcome of the algorithm (see exercises).

10.1.1 2-dim. data and 1-dim. neuron grid

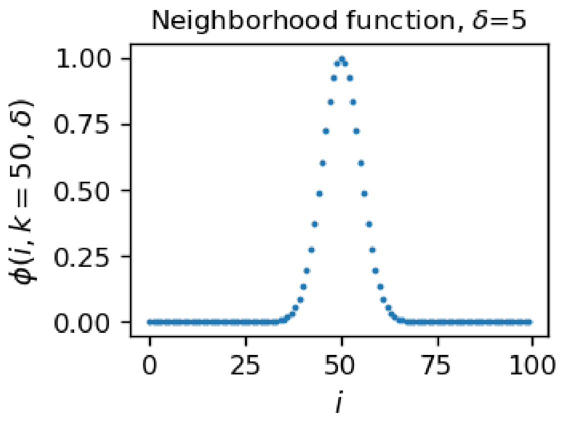
Let us see how the procedure works on a simple example. We map a grid of **num** neurons into (our favorite!) circle. So we have here the reduction of dimensions: $n = 2$, $k = 1$.

```
num=100 # number of neurons
```

The proximity function

```
def phi(i,k,d):                       # proximity function
    return np.exp(-(i-k)**2/(2*d**2)) # Gaussian
```

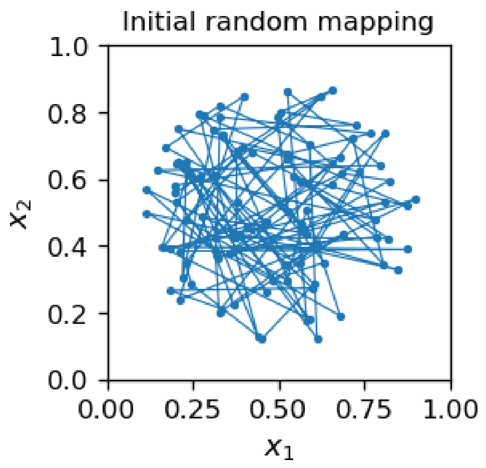
looks as follows around $k = 50$ and for the width parameter $\delta = 5$:



As a feature of the Gaussian, at $|k-i|=\delta$ it drops to 60% of the central value, and at $|k-i|=3\delta$ to 1%, a tiny fraction. Hence δ controls the size of the neighborhood of the winner. The neuron farther away from the winner than 3δ are practically left unupdated.

We initiate the network by by placing the grid on the plane (on the square $[0,1]\times[0,1]$), with a random location of each neuron. The line is drawn to guide the eye along the neuron indices: $1,2,3,\ldots m$, which are chaotically distributed.

```
W=np.array([func.point_c() for _ in range(num)]) # random initialization of weights
```



The line connects the subsequent neurons … They are chaotically scattered around the region.

```
eps=.5    # initial learning speed
de = 10   # initial neighborhood distance
ste=0     # inital number of caried out steps
```

```
# Kohonen's algorithm
for _ in range(150): # rounds
    eps=eps*.98      # dicrease learning speed
    de=de*.95        # ... and the neighborhood distance
    for _ in range(100):        # loop over points
        p=func.point_c()        # random point
        ste=ste+1               # steps
        dist=[func.eucl(p,W[k]) for k in range(num)]
         # array of squares of Euclidean disances between p and the neuron locations
```

(continues on next page)

(continued from previous page)

```
        ind_min = np.argmin(dist) # index of the winner
        for k in range(num):      # for all the neurons
            W[k]+=eps*phi(ind_min,k,de)*(p-W[k])
             # update of the neuron locations (weights), depending on proximity
```

As the algorith progresses (see Fig. 10.2) the neuron grid first "straightens up", and then gradually fills the whole space D (circle) in such a way that the neurons with adjacent indices are located close to each other. Figuratively speaking, a new point x attracts towerds itself the nearest neuron (the winner), and to a weaker extent its neighbors. At the beginning of the algorithm the neighborhood distance **de** is large, so large chunks of the nighboring point in input grid are pulled together, and the arrangement looks as the top right corner of Fig. 10.2. At later stages **de** is smaller, so only the winner and possibly its immediate neighbors are attracted to a new point. After completion, individual neurons "specialize" (are close to) in a certain data area.

In the present example, after about 20000 steps the result practically stops to change.

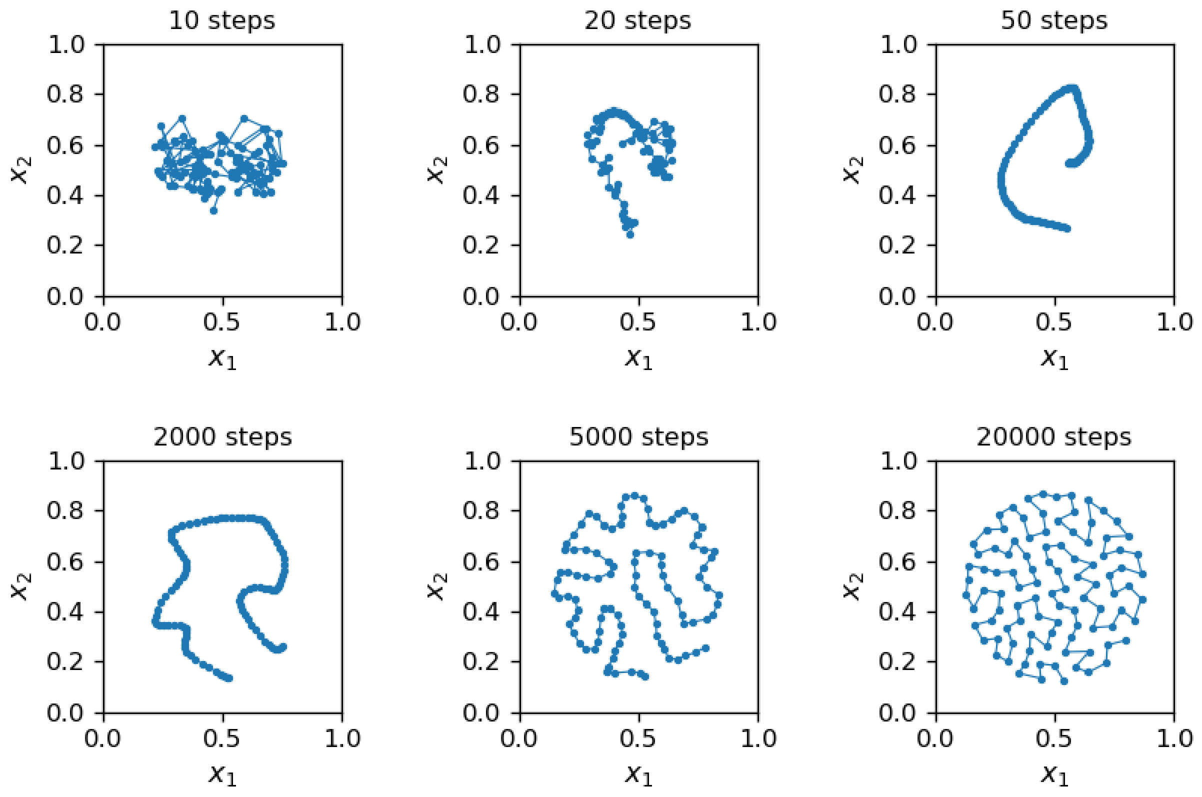
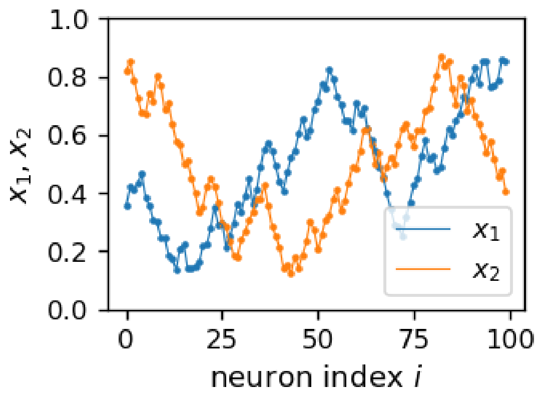


Fig. 10.2: Progress of Kohonen's algorithm. The lines, drawn to guide the eye, connects neurons with adjacent indices.

Kohonen network as a classifier

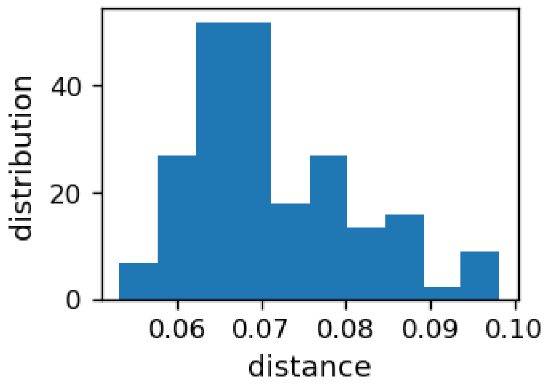
Having the trained network, we may use it as classifier similarly as in chapter {\ref}`un-lab`. We label a point from D with the index of the nearest neuron.

The plots in Fig. 10.2 are plotted in coordinates (x_1, x_2), that is, from the "point of view" of the D-space. One may also look at the result from the point of view of the N-space, i.e. plot x_1 and x_2 as functions of the neuron index i:



We note that the jumps in the above plotted curves are small. This can be seen quntitatively in the histogram below, where the average distance is about 0.07.

```
dd=[np.sqrt((W[i+1,0]-W[i,0])**2+(W[i+1,1]-W[i,1])**2) for i in range(num-1)]
        # array of distances between subsequent neurons in the grid

plt.figure(figsize=(2.8,2),dpi=120)

plt.xlabel('distance',fontsize=11)
plt.ylabel('distribution',fontsize=11)

plt.hist(dd, bins=10, density=True);    # histogram
```



Remarks

- We took a situation in which the data space with the dimension $n = 2$ is "sampled" by a discrete set of neurons forming $k = 1$-dimensional grid. Hence we have dimensional reduction.
- The outcome of the algorithm is a network in which a given neuron "focuses" on data from its vicinity. In a general case where the data are non-uniformly distributed, the neurons would fill the area containing more data more densely.
- The fact that there are no line intersections is a manifestation of topological features, discussed in detail below.
- The policy of choosing initial δ and ε parameters and reducing them appropriately in subsequent rounds is based on experience and non-trivial.
- The final result is not unequivocal, i.e. running the algorithm with a different initialization of the weights (positions of neurons) yields a different outcome, equally "good".

- Finally, the progress and result of the algorithm is reminiscent of the construction of the Peano curve in mathematics, which fills an area with a line.

10.1.2 2 dim. color map

Now we come a case of 3-dim. data and 2-dim. neuron grid, which is a situation from the right panel of Fig. 10.1. An RGB color is described with three numbers from $[0, 1]$, so it can nicely serve as input in our example.

The distance squared between two colors (this is just a distance between two points in the 3-dim. space) is taken in the Euclidean form

```
def dist3(p1,p2):
    """
    Square of the Euclidean distance between points p1 and p2
    in 3 dimensions.
    """
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2+(p1[2]-p2[2])**2
```

```
def phi2(ix,iy,kx,ky,d):  # proximity function for 2-dim. grid
    return np.exp(-((ix-kx)**2+(iy-ky)**2)/(d**2))  # Gaussian
```

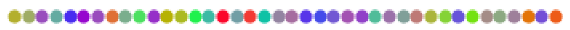
```
def rgbn():
    r,g,b=np.random.random(),np.random.random(),np.random.random()
    norm=np.sqrt(r*r+g*g+b*b)
    return np.array([r,g,b]/norm)
```

```
rgbn()
```

```
array([0.1234176 , 0.55504897, 0.82261093])
```

Next, we generate a sample of **ns** points with RGB colors:

```
ns=40  # number of colors in the sample

samp=[rgbn() for _ in range(ns)]
       # random sample

pls=plt.figure(figsize=(4,1),dpi=120)
plt.axis('off')
for i in range(ns): plt.scatter(i,0,color=samp[i], s=15);
```



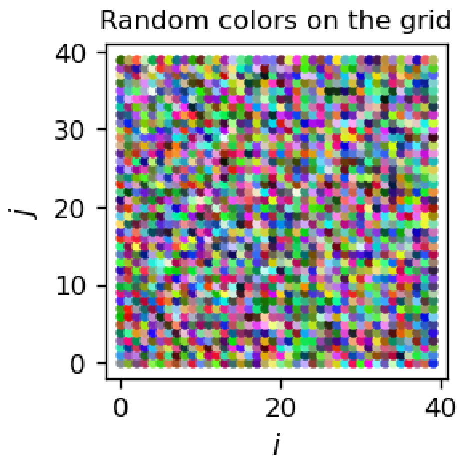
We use **size** x **size** grid of neurons. Each neuron's position (color, or weight) in the 3-dim. space is initialized randomly:

```
size=40  # neuron array of size x size (40 x 40)

tab=np.zeros((size,size,3))
```

(continues on next page)

(continued from previous page)

```
for i in range(size):           # i index in the grid
    for j in range(size):       # j index in the grid
        for k in range(3):      # RGB: 0-red, 1-green, 2-blue
            tab[i,j,k]=np.random.random() # random number form [0,1]
            # 3 RGB components for neuron in the grid positin (i,j)
```



Now we are ready to run Kohonen's algorithm:

```
eps=.5   # initial parameters
de = 20
```

```
for _ in range(150):    # rounds
    eps=eps*.995
    de=de*.96           # de shrinks faster than eps
    for s in range(ns): # loop over the points in the data sample
        p=samp[s]       # point from the sample
#         p=[np.random.random() for _ in range(3)]
        dist=[[dist3(p,tab[i][j]) for j in range(size)] for i in range(size)]
                        # distance to all neurons
        ind_min = np.argmin(dist) # the winner index
        ind_x=ind_min//size       # a trick to get a 2-dim index
        ind_y=ind_min%size

        for j in range(size):
            for i in range(size):
                tab[i][j]+=eps*phi2(ind_x,ind_y,i,j,de)*(p-tab[i][j]) # update      ␣
↪
```

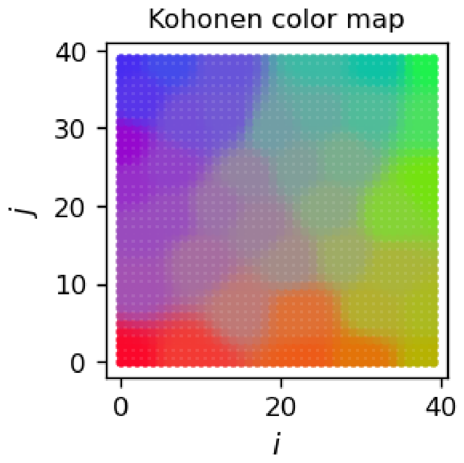
As a result we get an arrangement of our color sample in two dimensions in such a way that the neighboring areas have a similar color:

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("Kohonen color map",fontsize=10)

for i in range(size):
    for j in range(size):
        plt.scatter(i,j,color=tab[i][j], s=8)

plt.xlabel('$i$',fontsize=11)
```

(continues on next page)

(continued from previous page)

```
plt.ylabel('$j$',fontsize=11);
```



Remarks

- The areas for the individual colors of the sample have a comparable area. Generally, the area is proportional to the sample size.
- To get sharper boundaries between regions, de has to shrink faster than eps. Then, in the final stage of learning, the neuron update process takes place with small neighborhood radius.

10.2 U-matrix

A convenient way to present the results of the Kohonen algorithm when the grid is 2-dimensional is via the **unified distance matrix** (shortly **U-matrix**). The idea is to plot a 2-dimensional grayscale map with the intensity given by the averaged distance (in D-space) of the neuron to its immediate neighbors, and not the neuron poperty itself (such as the color in the plot above). This is particularly useful when the dimension of the input space is large, when it is difficult to visualize the results directly.

The definition of a U-matrix element U_{ij} is shown in Fig. 10.3. Let d be the distance in D-space and $[i, j]$ denote the neuron of indices i, j . We take

$$U_{ij} = \sqrt{d\left([i,j],[i+1,j]\right)^2 + d\left([i,j],[i-1,j]\right)^2 + d\left([i,j],[i,j+1]\right)^2 + d\left([i,j],[i,j-1]\right)^2}$$

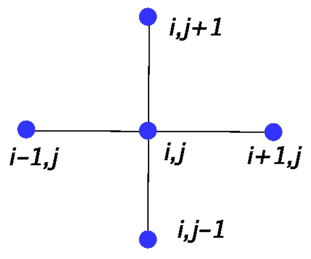
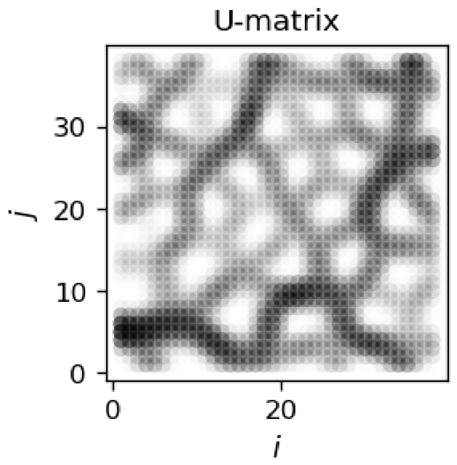


Fig. 10.3: Construction of U_{ij}: geometric average of the distance along the indicated links.

```
udm=np.zeros((size-2,size-2)) # create U-matrix with elements = 0

for i in range(1,size-1):
    for j in range(1,size-1):
        udm[i-1][j-1]=np.sqrt(dist3(tab[i][j],tab[i][j+1])+dist3(tab[i][j],tab[i][j-
↪1])+
                                        dist3(tab[i][j],tab[i+1][j])+dist3(tab[i][j],tab[i-1][j]))
                          # U-matrix as explained above
```

```
plt.figure(figsize=(2.3,2.3),dpi=120)
plt.title("U-matrix",fontsize=11)

for i in range(size-2): # loops over indices exclude the first and last point of the␣
↪grid
    for j in range(size-2):
        plt.scatter(i+1,j+1,color=[0,0,0,2*udm[i][j]], s=10)
            # color format: [R,G,B,intensity], 2 just scales up
plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11);
```



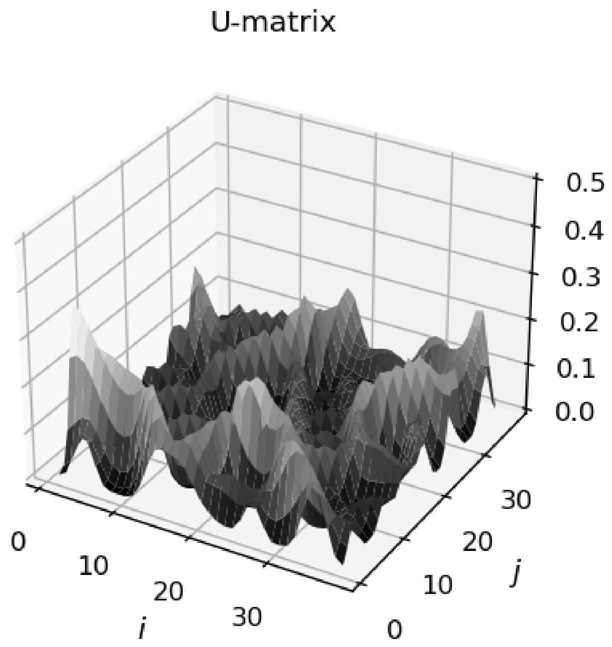
The white regions in the above figure correspond to clusters, separated with the darker regions. The higher is the boundary between clusters, the darker the plot. The same may be visualized in a 3D figure:

```
fig = plt.figure(figsize=(4,4),dpi=120)
axes1 = fig.add_subplot(111, projection="3d")
ax = fig.gca(projection='3d')

xx_1 = np.arange(1, size-1, 1)
xx_2 = np.arange(1, size-1, 1)

x_1, x_2 = np.meshgrid(xx_1, xx_2)

Z=np.array([[udm[i][j] for i in range(size-2)] for j in range(size-2)])

ax.set_zlim(0,.5)

ax.plot_surface(x_1,x_2, Z, cmap=cm.gray)

plt.xlabel('$i$',fontsize=11)
```

(continues on next page)

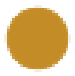
(continued from previous page)

```
plt.ylabel('$j$',fontsize=11);

plt.title("U-matrix",fontsize=11);
```



We an now classify a given (new) data point according to the obtained map.

```
nd=[np.random.random(),np.random.random(),np.random.random()]
```



It is useful to obtain a distance map from from this point:

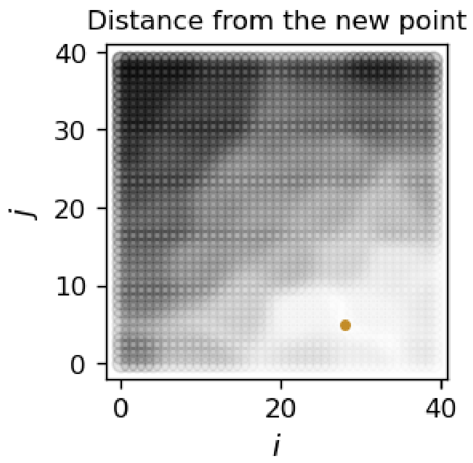
```
tad=np.zeros((size,size))

for i in range(size):
    for j in range(size):
        tad[i][j]=dist3(nd,tab[i][j])


ind_m = np.argmin(tad) # winner
in_x=ind_m//size
in_y=ind_m%size

da=np.sqrt(tad[in_x][in_y])
```

(continues on next page)

(continued from previous page)

```
print("Closest neuron grid coordinates: (",in_x,",",in_y,")")
print("Distance: ",np.round(da,3))
```

```
Closest neuron grid coordinates: ( 28 , 5 )
Distance:  0.032
```



The lightest region indicates the cluster, to which the point belongs. The darker the region, the lager the distance to the corresponding neurons.

10.2.1 Mapping colors on a line

In this subsection we present an example of reduction of 3-dim. data in a 1-dim. neuron grid. This proceeds along the lines of the previous evaluation, so we are brief in comments.

```
ns=8

samp=[[np.random.random(),np.random.random(),np.random.random()] for _ in range(ns)]

plt.figure(figsize=(4,1))
plt.title("Sample colors",fontsize=16)

plt.axis('off')

for i in range(ns):
    plt.scatter(i,0,color=samp[i], s=400);
```

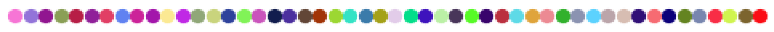
Sample colors

```
si=50   # 1D grid of si neurons
```

(continues on next page)

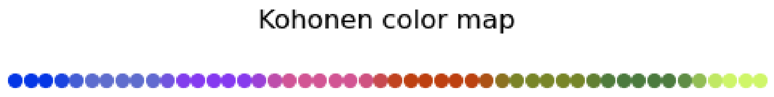
(continued from previous page)

```
tab2=np.zeros((si,3))

for i in range(si):
    for k in range(3):                # RGB components
        tab2[i][k]=np.random.random() # random initialization
```



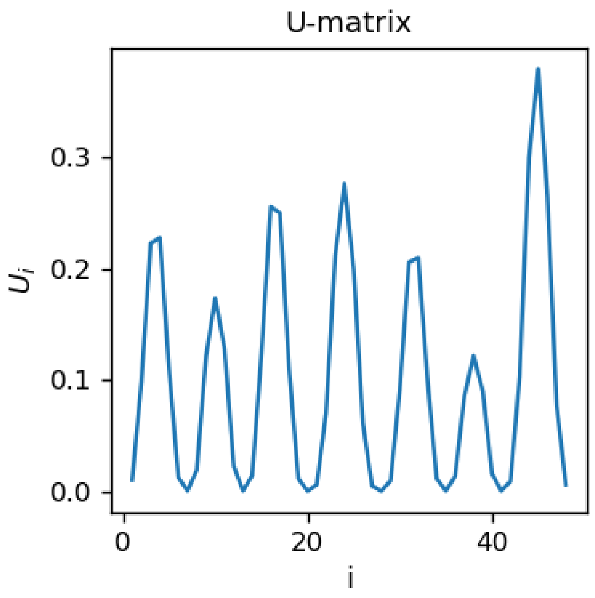
```
eps=.5
de = 20
```

```
for _ in range(200):
    eps=eps*.99
    de=de*.96
    for s in range(ns):
        p=samp[s]
        dist=[dist3(p,tab2[i]) for i in range(si)]
        ind_min = np.argmin(dist)
        for i in range(si):
            tab2[i]+=eps*phi(ind_min,i,de)*(p-tab2[i])
```



We note smooth transitions between colors. The formation of clusters can be seen with U-matrix, which now is of course one-dimensional:

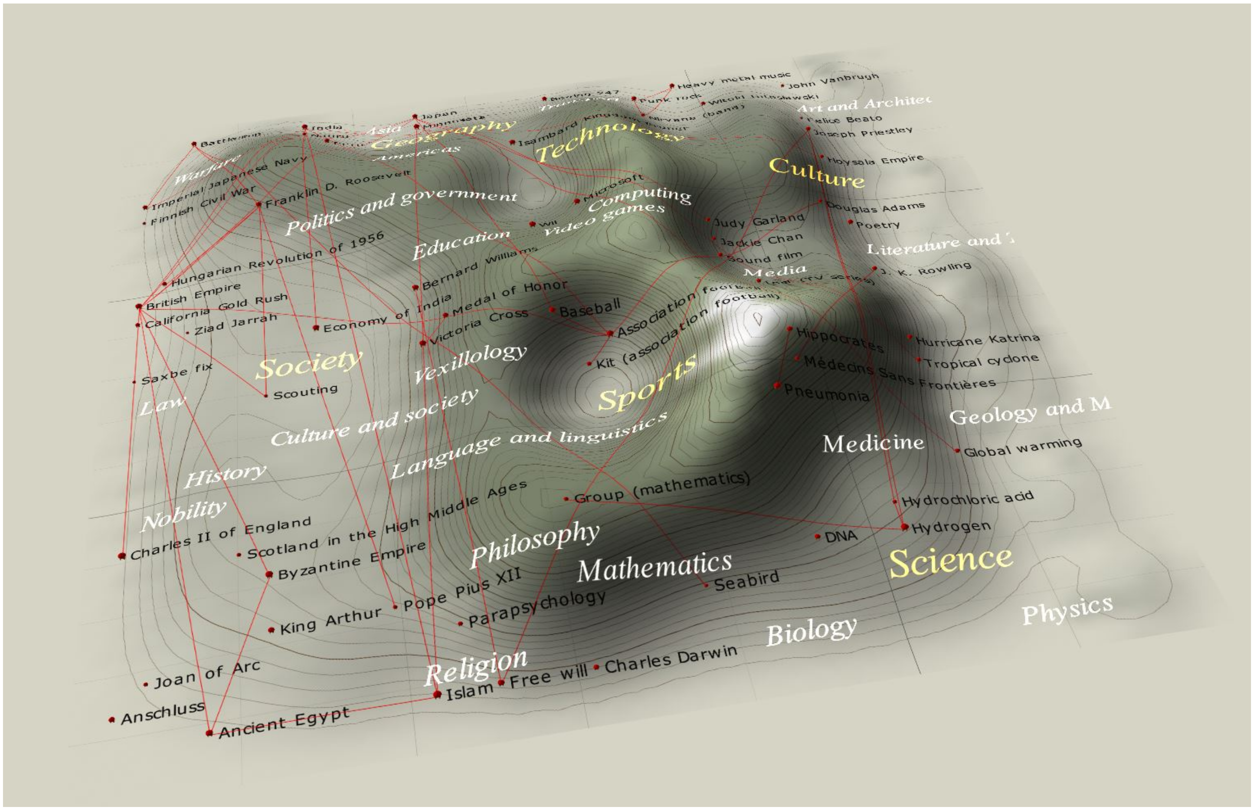
```
ta2=np.zeros(si-2)

for i in range(1,si-1):
    ta2[i-1]=np.sqrt(dist3(tab2[i],tab2[i+1])+dist3(tab2[i],tab2[i-1]))
```



The minima (there are 8 of them, which is the multiplicity of the sample) indicate the clusters. The height of the separating peaks shows how much the colors differ.

10.2.2 Wikipedia articles' similarity

The input space may have very large dimensions. In the Wikipedia example below, one takes articles from various fields and computes frequencies of words (for instance, how many times the word "goalkeeper" has been used, divided by the total number of words in the article). So essentially the dimensionality is of the oder of the number of English words, a huge number $\sim 10^5$! Then one uses Kohonen's algorith to carry out reduction into a 2-dim. grid of neurons. The U-matrix is following:



In particular we note that articles on sports are special and form a well-defined cluster. This is not surprising, as the jargon is very specific.

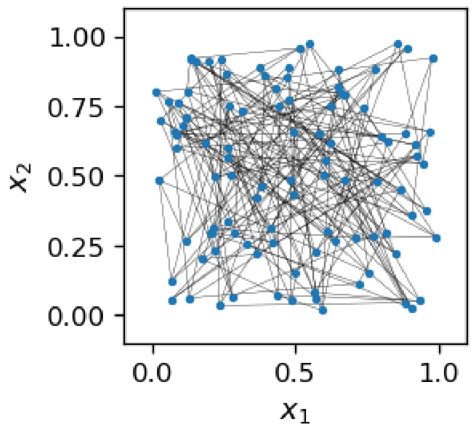
10.3 Mapping 2-dim. data into a 2-dim. grid

Finally, we come to a very important case of mapping 2-dim. data on a 2-dim. grid, as this is realized in our vision system between the retina and the visual cortex.

The algorithm proceedes analogously to the previous cases. We initialize an $n \times n$ grid of neurons, placing them randomly in the square $[0, 1] \times [0, 1]$.

```
n=10
sam=np.array([func.point() for _ in range(n*n)])
```

The lines, drawn to guide the eye, join the adjacent index pairs [i,j] and [i+1,j], or [i,j] and [i,j+1]. The neurons in the interior of the grid have 4 nearest neighbors, those at the boundary 3, except for the corners, which have 2.



We note a total initial "chaos", as the neurons are located randomly. Now comes Kohonen's miracle:

```
eps=.5    # initial learning speed
de = 3    # initial neighborhood distance
nr = 100 # number of rounds
rep= 300 # number of points in each round
ste=0     # inital number of caried out steps
```

```
# analogous to the previous codes
for _ in range(10):    # rounds
    eps=eps*.97
    de=de*.98
    for _ in range(rep):     # repeat for rep points
        ste=ste+1
        p=point()
        dist=[func.eucl(p,sam[l]) for l in range(n*n)]
        ind_min = np.argmin(dist)
        ind_i=ind_min%n
        ind_j=ind_min//n

        for j in range(n):
            for i in range(n):
                sam[i+n*j]+=eps*phi2(ind_i,ind_j,i,j,de)*(p-sam[i+n*j])
```

```
# fl.savefig('images/kb30000.png')
```

As the algorithm progresses, chaos changes into nearly perfect order, with the grid placed unifomly in the square of the data, with only slight displacements from a regular arrangement. On the way, at 600 steps, we notice a phenomenon called "twist", where many neurons have a close location an the grid is crumpled.

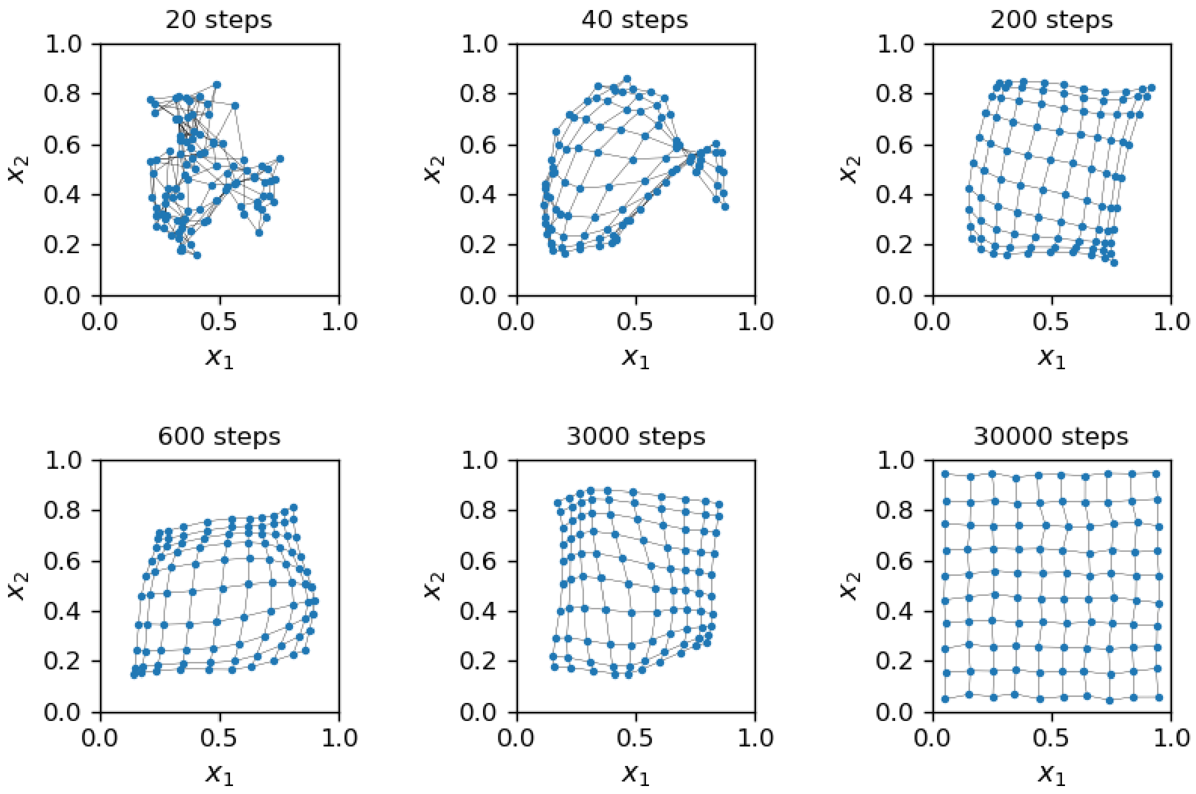


Fig. 10.4: Progress of Kohonen's algorithm. The lines, drawn to guide the eye, connects neurons with adjacent indices.

10.4 Topological properties

Recall the Voronoi construction of categories introduced in section *Voronoi areas*. One can simply use it now, treating the neurons from a grid as the Voronoi points. The Voronoi construction provides a mapping v from the data space D to the neuron space N,

$$v : D \to N$$

(note that this goes in the opposite direction than function f defined at the beginning of this chapter).

For instance, we take the bottom right panel of Fig. 10.4, construct the Voronoi areas for all the neurons, and thus obtain a mapping for all poins in the (x_1, x_2) square. The reader may notice that there is an ambiguity for points lying exactly at boundaries between areas, but this can be taken care of by using an additional prescription, for instance, selecting a neuron lying at direction which has the lowest angle, etc.

Now a key observation:

Topological property

For situations such as in the bottom right panel of Fig. 10.4, mapping v has the property that when d_1 and d_2 from D are close to each other, then also their corresponding neurons are close, i.e. the indices $v(d_1)$ and $v(d_2)$ are close.

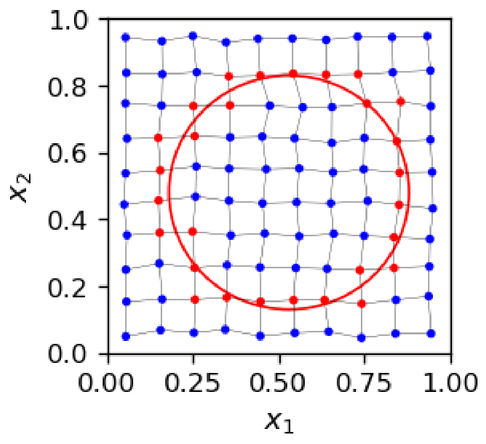
This observation is obvious. Since d_1 and d_2 are close (and we mean very close, closer than the resolution of the grid), then they must belong either to

- the same Voronoi area, where $v(d_1) = v(d_2)$, or
- neighboring Voronoi areas.

Since for the considered situation the neighboring areas have the grid indices differing by 1, the conclusion that $v(d_1)$ and $v(d_2)$ are close follows.

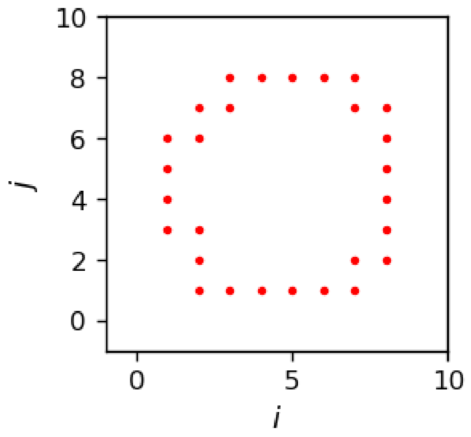
Note that this feature of Hohonen's maps is far from trivial and does not hold for a general mapping. Imagine for instance that we stop our simulations for Fig. 10.4 after 40 steps (top central panel) and are left with a "twisted" grid. In the vicinity of the twist, the indices of the adjacent Voronoi areas differ largely, and the considered topological property no longer holds.

The discussed topological propery has general and far-reaching consequences. First, it allows to carry over "shapes" from D to N. Imagine that we have a circle in the D-space:



where the red points indicate the winners for certain sections of the circle. When we draw these points alone in the N space, we get

```
plt.figure(figsize=(2.3,2.3),dpi=120)

plt.xlim(-1,10)
plt.ylim(-1,10)

plt.scatter(ci//10,ci%10,c='red',s=5)


plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11);
```



This looks as a (rough and discrete) circle. Note that in our example we only have $n^2 = 100$ pixels to our disposal, and the image would look better and better with increasing n. At some point one would reach the 10M pixels resolution of typical camera, and then the image would look smooth.

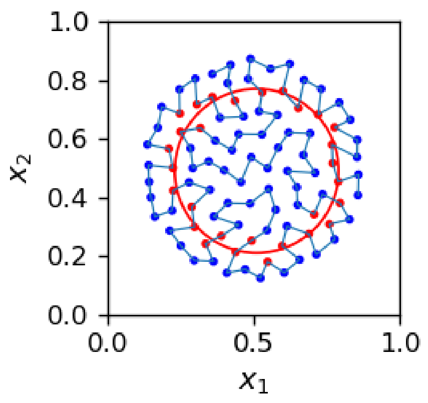
Vision

The topological property of the Kohonen mapping is believed to have a prime importance in our vision system and the perception of objects. $\dots$

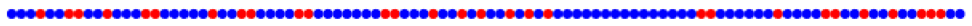
Another key topological feature is the preservation of connectedness. If an area A in D is connected (is in one piece), then its image $v(A)$ in N is also connected (we ignore the desired rigor here as to what "connected" means in a discrete space and rely on intuition). So things do not get "warped up" when transforming form D to N.

Note that the discussed topological features need not be present when dimensionality is reduced, as in our previous examples. Take for instance the bottom right panel of Fig. 10.2. There, many neighboring pairs of Voronoi areas correspond to distant indices, and it is no longer true that $v(d_1)$ and $v(d_2)$ are close for close d_1 and d_2, as these points may belong to different Voronoi areas with distant indices.

Our example with the circle now looks like this:



When we go along the grid indices, we see the image of our circle (red dots) as a bunch of **disconnected** sections. Topology is not preserved.



Note: Topological features of Kohonen's maps hold for equal dimensionalities of the input space and the neuron grid, $n = k$, and in general do not hold for the reduced dimensionality case, $k < n$.

10.5 Lateral inhibition

In a last topic of these lectures, we return to the issue of how the competition for the "winner" is realized in ANNs. Above, we have just used the minimum (or maximum, when the signal was extended to a hyperphere) functions, but this is embarrasingly outside of the framework. Such an insection of which neuron yields the strongest signal would require an "external wizard", or some sort of a control unit. Mathematically, it is easy to imagine, but the challenge is to build it from neurons.

Actually, if the neurons in a layer "talk" to one another, we can have a contest. An architecture as in Fig. 10.5 allows for an arrangement of competition and a natural realization of the winner-take-most mechanism.

Neuron number i receives the signal $s_i = xw_i$, where x is the input (the same for all the neurons), and w_i is the weight. It produces an output y_i, but part of it is sent neuron j as $F_{ji}y_i$, where F_{ij} is the coupling strength (we assume $F_{ii} = 0$ - no self coupling). Neuron i also receives output from neuron j in the form $F_{ij}y_j$. Summing over all the neurons yields

$$y_i = s_i + \sum_{j \neq i} F_{ij} y_j,$$

which in the matrix notation becomes $y = s + Fy$, or $y(I - F) = s$, where I is the identity matrix. Solving for y gives

$$y = (I - F)^{-1} s. \tag{10.1}$$

One needs to model appropriately the coupling matrix F. We take

$F_{ii} = 0,$

$F_{ij} = -a\exp(-|i-j|/b) \quad \text{for } i \neq j, \quad a, b > 0,$

i.e. assume attenuation (negative values), which is strongest for close neighbors and decreases with distance with a characteristic scale b.

The Python implemetation is as follows:

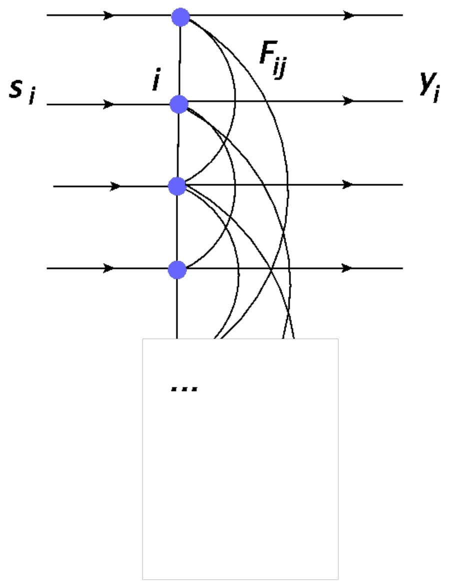
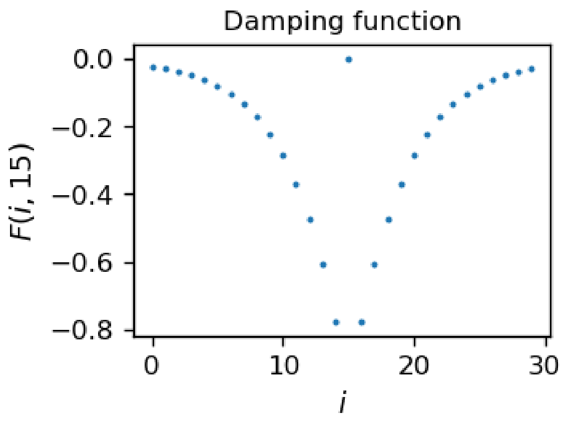


Fig. 10.5: Network with inter-neuron coupling used for modeling lateral inhibition. All the neurons are connected in both directions (lines without arrows).

```
ns = 30;        # number of neurons
b = 4;          # parameter controling the decrease of damping with distance
a = 1;          # magnitude of damping

F=np.array([[-a*np.exp(-np.abs(i-j)/b) for i in range(ns)] for j in range(ns)])
                    # exponential fall-off

for i in range(ns):
    F[i][i]=0       # no self-damping
```



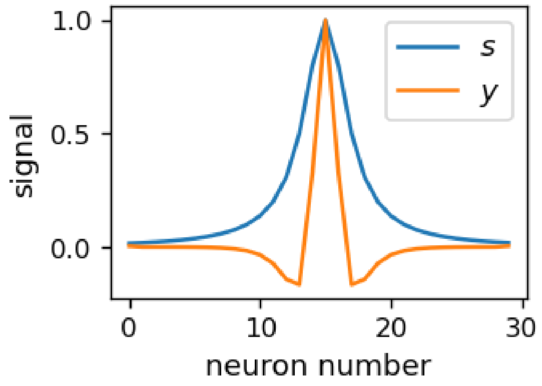
We assume a bell-shaped Lorentzian input signal s, with a maximum in the middle neuron. The width is controled with **D**.

```
D=2
s = np.array([D**2/((i - ns/2)**2 + D**2) for i in range(ns)]) # Lorentzian function
```

and solve Eq. eq-lat via inverting the $(I - F)$ matrix:

```
invF=np.linalg.inv(np.identity(ns)-F) # matrix inversion
y=np.dot(invF,s)                      # multiplication
y=y/y[15]                             # normalization (inessential)
```

What follows is quite remarkable: the output signal y is much narrower form the input signal, which is a realization of the "winner-take-all" scenario.



Pyramidal neurons

Exercises

Construct a Kohonen mapping form a 2D shape (square, cicle, 2 disjoint squares) on a 2D grid of neurons.

CHAPTER

ELEVEN

CONCLUDING REMARKS

… Conclude here our discussion of the supervised learning and the back proparation, we provide a number of remarks nd hints. First, in programmers life, bulding a well-functioning ANN, even for simple problems as used for illustrations up to now, can be a frustrating eperience! …

Neurological impossiblity of backprop.

HCP - Human Connectome Project http://www.humanconnectomeproject.org/

Initial conditions for minimization, basen of convergence, learning stategy, improvements of steepest descent …

Professional libraries, experience verified with success …

Note: The references provided in the text as hyperlinks are not repeated in the list below.

CHAPTER TWELVE

APPENDIX

12.1 neural package

The structure of the library tree is as follows:

```
lib_nn
└── neural
    ├── __init__.py
    ├── draw.py
    └── func.py
```

12.1.1 func.py module

```
"""
Contains functions repeatedy used in the lecture
"""

import numpy as np


def step(s):
    """ step """ # komentarz w potrójnym cudzysłowie
    if s>0:
        return 1
    else:
        return 0

def neuron(x,w,f=step): # (in the neural library)
    """
    MCP neuron

    x: array of inputs  [x1, x2,...,xn]
    w: array of weights [w0, w1, w2,...,wn]
    f: activation function, with step as default

    return: signal=weighted sum w0 + x1 w1 + x2 w2 +...+ xn wn = x.w
    """
    return f(np.dot(np.insert(x,0,1),w)) # insert x0=1, signal s=x.w, output f(s)

def sig(s,T=1):
    """ sigmoid """
    return 1/(1+np.exp(-s/T))
```

(continues on next page)

(continued from previous page)

```
def dsig(s, T=1):
    """derivative of sigmoid"""
    return sig(s)*(1-sig(s))/T

def lin(s):
    """ linear function """
    return s

def dlin(s):
    """ derivative of linear function """
    return 1

def relu(s):
    """ ReLU function """
    if s>0:
        return s
    else:
        return 0

def drelu(s):
    """ derivative of ReLU function """
    if s>0:
        return 1
    else:
        return 0

def lrelu(s,a=0.1):
    """ leaky ReLU function """
    if s>0:
        return s
    else:
        return a*s

def dlrelu(s,a=0.1):
    """ derivative of leaky ReLU function """
    if s>0:
        return 1
    else:
        return a

def softplus(s):
    """ softplus function """
    return np.log(1+np.exp(s))

def dsoftplus(s):
    """ derivative softplus function """
    return 1/(1+np.exp(-s))

def eucl(p1,p2): # square of the Euclidean distance
    """
    Squqre of Euclidean distance between to points in 2-dim. space

    input: p1, p1 - arrays in the format [x1,x2]
    return: square of Euclidean distance
    """
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2
```

(continues on next page)

(continued from previous page)

```
def l2(w0,w1,w2):
    """for separating line"""
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]

def rn():
    """
    random number from [-0.5,0.5]
    """
    return np.random.rand()-0.5

def point_c():
    """
    random point from a cirle
    """
    while True:
        x=np.random.random()
        y=np.random.random()
        if (x-0.5)**2+(y-0.5)**2 < 0.4**2:
            break
    return np.array([x,y])

def point():
    """
    random point from [0,1]x[0,1]
    """
    x=np.random.random()
    y=np.random.random()
    return np.array([x,y])

def set_ran_w(ar,s=1):
    """
    Set weights randomly

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    s - scale factor: each weight is in the range [-0.s, 0.5s]

    return:
    w - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}
    """
    l=len(ar)
    w={}
    for k in range(l-1):
        w.update({k+1: [[s*rn() for i in range(ar[k+1])] for j in range(ar[k]+1)]})
    return w


def set_val_w(ar,a=0):
    """
    Set weights to a constant value

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
```

(continues on next page)

(continued from previous page)

```
    (from input layer 0 to output layer l, bias nodes not counted)

    a - value for each weight

    return:
    w - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}
    """
    l=len(ar)
    w={}
    for k in range(l-1):
        w.update({k+1: [[a for i in range(ar[k+1])] for j in range(ar[k]+1)]})
    return w


def feed_forward(ar, we, x_in, ff=step):
    """
    Feed-forward propagation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    ff - activation function (default: step)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)

    """

    l=len(ar)-1                   # number of neuron layers
    x_in=np.insert(x_in,0,1)      # input, with the bias node inserted

    x={}                          # empty dictionary
    x.update({0: np.array(x_in)}) # add input signal

    for i in range(0,l-1):        # loop over layers till before last one
        s=np.dot(x[i],we[i+1])    # signal, matrix multiplication
        y=[ff(s[k]) for k in range(ar[i+1])] # output from activation
        x.update({i+1: np.insert(y,0,1)}) # add bias node and update x

    # the last layer - no adding of the bias node
    s=np.dot(x[l-1],we[l])
    y=[ff(s[q]) for q in range(ar[l])]
    x.update({l: y})          # update x

    return x

def back_prop(fe,la, p, ar, we, eps,f=sig, df=dsig):
    """
```

(continues on next page)

(continued from previous page)

```
    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - disctionary of weights
    eps - learning speed
    f   - activation function
    df  - derivaive of f
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
    nl=ar[l]    # number of neurons in the otput layer

    x=feed_forward(ar,we,fe[p],ff=f) # feed-forward of point p

    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                    df(np.dot(x[l-1],we[l])[gam]) for gam in range(nl)]})
    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])


def feed_forward_o(ar, we, x_in, ff=sig, ffo=lin):
    """
    Feed-forward propagation with different output activation

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    x_in - input vector of length n_0 (bias not included)

    f  - activation function (default: sigmoid)
    fo - activation function in the output layer (default: linear)

    return:
    x - dictionary of signals leaving subsequent layers in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}
    (the output layer carries no bias)

    """

    l=len(ar)-1                   # number of neuron layers
    x_in=np.insert(x_in,0,1)      # input, with the bias node inserted

    x={}                          # empty dictionary
    x.update({0: np.array(x_in)}) # add input signal
```

(continues on next page)

(continued from previous page)

```
    for i in range(0,l-1):          # loop over layers till before last one
        s=np.dot(x[i],we[i+1])      # signal, matrix multiplication
        y=[ff(s[k]) for k in range(ar[i+1])] # output from activation
        x.update({i+1: np.insert(y,0,1)}) # add bias node and update x

    # the last layer - no adding of the bias node
    s=np.dot(x[l-1],we[l])
    y=[ffo(s[q]) for q in range(ar[l])] # output activation function
    x.update({l: y})                    # update x

    return x


def back_prop_o(fe,la, p, ar, we, eps, f=sig, df=dsig, fo=lin, dfo=dlin):
    """
    fe - array of features
    la - array of labels
    p  - index of the used data point
    ar - array of numbers of nodes in subsequent layers
    we - disctionary of weights
    eps - learning speed
    f   - activation function
    df  - derivaive of f
    fo  - activation function in the output layer (default: linear)
    dfo - derivative of fo
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
    nl=ar[l]    # number of neurons in the otput layer

    x=feed_forward_o(ar,we,fe[p],ff=f,ffo=fo) # feed-forward of point p

    # formulas from the derivation in a one-to-one notation:

    D={}
    D.update({l: [2*(x[l][gam]-la[p][gam])*
                  dfo(np.dot(x[l-1],we[l])[gam]) for gam in range(nl)]})

    we[l]-=eps*np.outer(x[l-1],D[l])

    for j in reversed(range(1,l)):
        u=np.delete(np.dot(we[j+1],D[j+1]),0)
        v=np.dot(x[j-1],we[j])
        D.update({j: [u[i]*df(v[i]) for i in range(len(u))]})
        we[j]-=eps*np.outer(x[j-1],D[j])
```

12.1.2 draw.py module

```
"""
Plotting functions used in the lecture.
"""

import numpy as np
import matplotlib.pyplot as plt


def plot(*args, title='activation function', x_label='signal', y_label='response',
         start=-2, stop=2, samples=100):
    """
    Wrapper on matplotlib.pyplot library.
    Plots functions passed as *args.
    Functions need to accept a single number argument and return a single number.
    Example usage: plot(lambda x: x * x)
                   plot(func.step,func.sig)
    """

    # defines range and detail level of the plot
    s = np.linspace(start, stop, samples)

    ff=plt.figure(figsize=(2.8,2.3),dpi=120)
    plt.title(title, fontsize=11)
    plt.xlabel(x_label, fontsize=11)
    plt.ylabel(y_label, fontsize=11)

    for fun in args:
        data_to_plot = [fun(x) for x in s]
        plt.plot(s, data_to_plot)

    return ff;


def plot_net_simp(n_layer):
    """
    Draw the network architecture without bias nores

    input: array of numbers of nodes in subsequent layers [n0, n1, n2,...]
    return: graphics object
    """
    l_layer=len(n_layer)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(n_layer[0]):
            plt.scatter(0, j-n_layer[0]/2, s=50,c='black',zorder=10)

# neuron layer nodes
    for i in range(1,l_layer):
        for j in range(n_layer[i]):
            plt.scatter(i, j-n_layer[i]/2, s=100,c='blue',zorder=10)

# bias nodes
    for k in range(n_layer[l_layer-1]):
        plt.plot([l_layer-1,l_layer],[n_layer[l_layer-1]/2-1,n_layer[l_layer-1]/2-1],␣
↪s=50,c='gray',zorder=10)
```

(continues on next page)

(continued from previous page)

```
# edges
    for i in range(l_layer-1):
        for j in range(n_layer[i]):
            for k in range(n_layer[i+1]):
                plt.plot([i,i+1],[j-n_layer[i]/2,k-n_layer[i+1]/2], c='gray')

    plt.axis("off")

    return ff;


def plot_net(ar):
    """
    Draw network without bias nodes

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)

# neuron layer nodes
    for i in range(1,l):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c='gray')

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def plot_net_w(ar,we,wid=1):
    """
    Draw the network architecture with weights
```

(continues on next page)

(continued from previous page)

```
    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    wid - controls the width of the lines

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input nodes
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)

# neuron layer nodes
    for i in range(1,l):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                th=wid*we[i+1][j][k]
                if th>0:
                    col='red'
                else:
                    col='blue'
                th=abs(th)
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c=col,linewidth=th)

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def plot_net_w_x(ar,we,wid,x):
    """
    Draw the network architecture with weights and signals

    input:
    ar - array of numbers of nodes in subsequent layers [n_0, n_1,...,n_l]
    (from input layer 0 to output layer l, bias nodes not counted)

    we - dictionary of weights for neuron layers 1, 2,...,l in the format
```

(continues on next page)

(continued from previous page)

```
    {1: array[n_0+1,n_1],...,l: array[n_(l-1)+1,n_l]}

    wid - controls the width of the lines

    x - dictionary the the signal in the format
    {0: array[n_0+1],...,l-1: array[n_(l-1)+1], l: array[nl]}

    return: graphics object
    """
    l=len(ar)
    ff=plt.figure(figsize=(4.3,2.3),dpi=120)

# input layer
    for j in range(ar[0]):
            plt.scatter(0, j-(ar[0]-1)/2, s=50,c='black',zorder=10)
            lab=np.round(x[0][j+1],3)
            plt.text(-0.27, j-(ar[0]-1)/2+0.1, lab, fontsize=7)

# intermediate layer
    for i in range(1,l-1):
        for j in range(ar[i]):
            plt.scatter(i, j-(ar[i]-1)/2, s=100,c='blue',zorder=10)
            lab=np.round(x[i][j+1],3)
            plt.text(i+0.1, j-(ar[i]-1)/2+0.1, lab, fontsize=7)

# output layer
    for j in range(ar[l-1]):
        plt.scatter(l-1, j-(ar[l-1]-1)/2, s=100,c='blue',zorder=10)
        lab=np.round(x[l-1][j],3)
        plt.text(l-1+0.1, j-(ar[l-1]-1)/2+0.1, lab, fontsize=7)

# bias nodes
    for i in range(l-1):
            plt.scatter(i, 0-(ar[i]+1)/2, s=50,c='gray',zorder=10)

# edges
    for i in range(l-1):
        for j in range(ar[i]+1):
            for k in range(ar[i+1]):
                th=wid*we[i+1][j][k]
                if th>0:
                    col='red'
                else:
                    col='blue'
                th=abs(th)
                plt.plot([i,i+1],[j-(ar[i]+1)/2,k+1-(ar[i+1]+1)/2],c=col,linewidth=th)

# the last edge on the right
    for j in range(ar[l-1]):
        plt.plot([l-1,l-1+0.7],[j-(ar[l-1]-1)/2,j-(ar[l-1]-1)/2],c='gray')

    plt.axis("off")

    return ff;


def l2(w0,w1,w2):
```

(continues on next page)

(continued from previous page)

```
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]
```

BIBLIOGRAPHY

[Gut16] John Guttag. *Introduction to computation and programming using Python: With application to understanding data*. MIT Press, 2016.