Wojciech Broniowski

Explaining neural networks in raw Python

Lectures in Jupyter

Explaining neural networks in raw Python: lectures in Jupyter

Wojciech Broniowski

Aug 19, 2021

CONTENTS

Wojciech Broniowski

These lectures were originally given to undergraduate 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 most elementary 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.

Brevity

The text is brief (the pdf printout has ~120 pages including the appendix), so a diligent student can complete the course in a few afternoons!

Links

- Jupyter Book: https://bronwojtek.github.io/neuralnets-in-raw-python/docs/index.html
- pdf and codes: www.ifj.edu.pl/~broniows/nn or www.ujk.edu.pl/~broniows/nn

How to run the book codes

A major advantage of executable books is that the reader may enjoy running the source codes himself, modifying them and playing around. No downloading, installation or configuration are required. Simply go to

https://bronwojtek.github.io/neuralnets-in-raw-python/docs/index.html,

in the left menu select any chapter below the Introduction, click the "rocket" icon at the top right of the screen, and choose "Binder". After some initialization time (for the first time it is rather long) the notebook can be run.

For local running, the codes for each chapter in the form of Jupyter notebooks can be downloaded by clicking the "arrow-down" icon at the top right of the screen. A complete set of files is also available from the links given above.

Appendix *How to run the book codes* explains step-by-step how to proceed with the local execution of the codes.

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

ISBN: **978-83-962099-0-0** (pdf version)

CHAPTER

ONE

INTRODUCTION

1.1 Purpose of these lectures

The goal of this course is to teach some basics of the omnipresent neural networks with Python [Bar16, Gut16, Mat19]. Both the explanations of key concepts of neural networks and the illustrative programs are kept at a very elementary undergraduate, almost "high-school" level. The codes, made very simple, are described in detail. Moreover, they are written without any use of higher-level libraries for neural networks, which helps in better understanding of the explained algorithms and shows how to program them from scratch.

Who is the book for?

The reader may 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 and its simplest applications, supervised learning with backpropagation for 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. Guiding explanations and comments in the codes are provided.

Exercises

At the end of each chapter some exercises are suggested, with the goal to familiarize the reader with the covered topics and the little codes. Most of exercises involve simple modifications/extensions of appropriate pieces of the lecture material.

Literature

There are countless textbooks and lecture notes devoted the matters discussed in this course, hence the author will not attempt to present an even incomplete list of literature. We only cite items which a more interested reader might look at.

With simplicity as guidance, our choice of topics took inspiration from detailed lectures by Daniel Kersten, illustrated in Mathematica, from an on-line book by Raul Rojas (also available in a printed version [FR13]), and the from **physicists'** (as myself!) point of view from [MullerRS12].

1.2 Biological inspiration

Inspiration for computational mathematical models discussed in this course originates from the biological structure of our neural system [KSJ+12]. The central nervous system (the brain) contains a huge number ($\sim 10^{11}$) of neurons, which may be viewed as tiny elementary processor units. They receive a signal via **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 passed stimulus. Moreover, the strength of the synaptic connections may change in time (Hebbian rule tells us that the connections get stronger if they are being used repeatedly). In this sense, the neuron is "programmable".

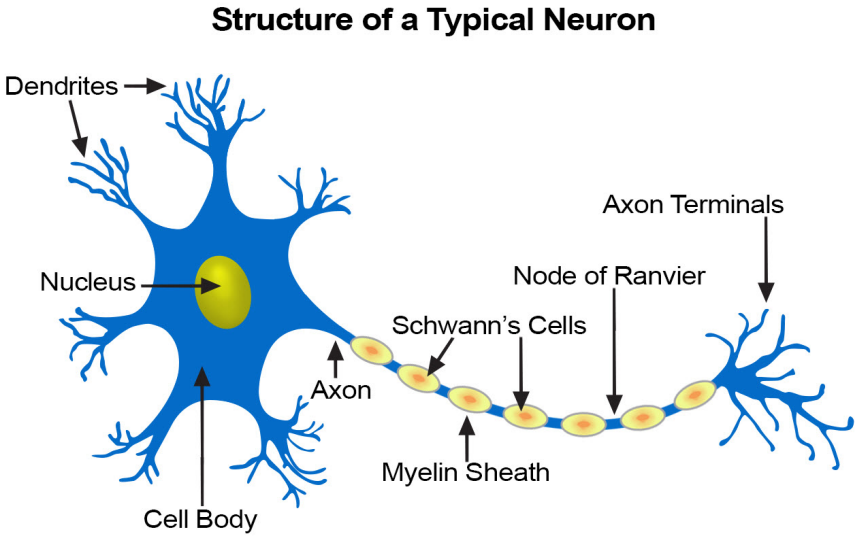


Fig. 1.1: Biological neuron (from https://training.seer.cancer.gov/anatomy/nervous/tissue.html).

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

Moreover, the speed of traveling of the nerve impulses, which is due to electrochemical processes, is not impressive, 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 along the wires at speeds of the order of 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 to an audio stimulus, and 0.15s to 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 extremely 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. Multiplying the 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! Hence if the firing is identified with an elementary calculation, the thus defined power of the brain is comparable to that of a standard computer processor.

The above facts might 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. This feature makes it radically different and immensely more complicated than the architecture consisting of a control unit, processor, and memory in our computers (the von Neumann machine architecture). There, the number of connections is of the

order of the number of bits of the 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 the 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 configuration, of course, never realize in practice, nevertheless the number of possible configuration states of the brain, or the "programs" it can run, is truly immense.

In recent years, with developing powerful imaging techniques, it became possible to map the connections in the brain with unprecedented resolution, where single nerve bundles 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 much simpler fruit fly, the drosophila connectome project is very well advanced.

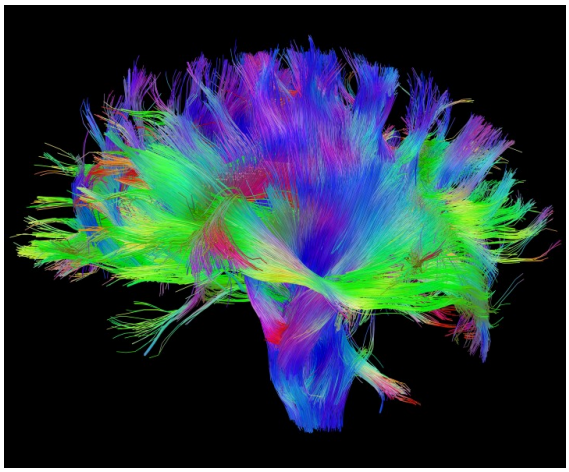


Fig. 1.2: White matter fiber architecture of the brain (from the Human Connectome Project humanconnectomeproject.org)

Important: The "immense connectivity" feature, with zillions of neurons serving as parallel elementary processors, makes the brain a completely different computational device from the Von Neumann machine (i.e. our everyday computers).

1.3 Feed-forward networks

The neurophysiological research of the brain provides important guidelines for mathematical models used in artificial neural networks (**ANN**s). 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, exemplified in Fig. 1.3. They consist of an **input** layer (black dots), which just represents digitized data, and layers of neurons (colored blobs). The number of neurons in each layer may be different. The complexity of the network and the tasks it may accomplish increase, naturally, with the number of layers and the number of neurons.

In the remaining part of this section we give, in a rather dense format, a bunch of important definitions:

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 (more than one neuron layer) networks the neuron 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 the recent "ground-breaking" **deep networks**.

Neurons in various layers do not have to function 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), indicated with arrows, to the neurons in subsequent layers. In feed-forward networks, as the one in Fig. 1.3, it can only move forward (from left to right in

the figure): from the input to the first neuron layer, from the first to the second, and so on until the output is reached. No going backward to preceding layers, or parallel propagation among the neurons of the same layer, are allowed. This would be a **recurrent** feature that we touch upon in section *Lateral inhibition*.

As we describe in detail in the following chapters, the traveling signal is appropriately **processed** by the neurons, hence the device carries out a computation: input is being transformed into output.

In the sample network of Fig. 1.3 each neuron from a preceding layer is connected to each neuron in the following layer. Such ANNs are called **fully connected**.

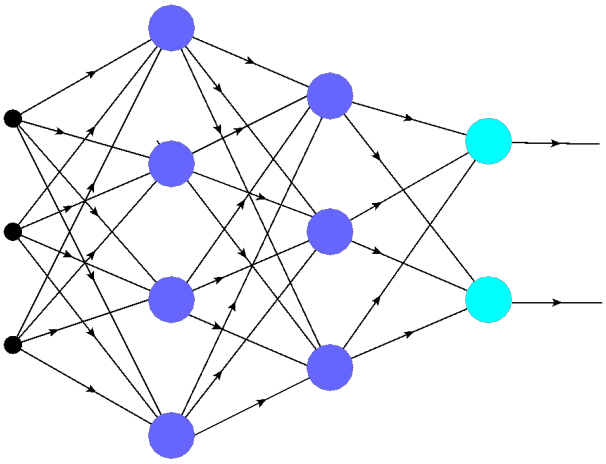


Fig. 1.3: A sample feed-foward fully connected artificial neural network. The colored blobs represent the neurons, and the edges indicate the synaptic connections. The signal propagates starting from the input (black dots), via the neurons in subsequent intermediate (hidden) layers (purple blobs), to the output layer (light blue blobs). The strength of the connections is controled by weights (hyperparameters) assigned to the edges.

As we will discuss in greater detail shortly, each edge (synaptic connection) in the network has a certain "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. 1.3, have very many connections (here 30), hence contain a lot of hyperparameters. Thus, while sometimes looking innocuously, ANNs are in fact very complex multi-parametric systems. Moreover, 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 using Python, just second after JavaScript (~14 million). In particular, it proves very practical in applications to ANNs.

1.4.1 Imported packages

Throughout this course we use some standard Python library packages for the numerics, plotting, etc. As stressed, we do not use any libraries specifically dedicated to neural networks. Each lecture's notebook starts with the inclusion of some of these libraries:

```
import numpy as np               # numerical
import statistics as st          # statistics
import matplotlib.pyplot as plt # plotting
import matplotlib as mpl         # plotting
import matplotlib.cm as cm       # contour plots

from mpl_toolkits.mplot3d.axes3d import Axes3D   # 3D plots
from IPython.display import display, Image, HTML # display imported graphics
```

---

neural package

Functions created during this course which are of repeated use, are placed in the private library package **neural**, described in appendix *neural package*.

Assuming the package is in the relative subdirectory **lib_nn**, it is imported in the following way:

```
import sys                    # system
sys.path.append('./lib_nn') # path to the lecture's package

from neural import *          # import the lecture's package
```

```
Invoking __init__.py for neural
```

See appendix *neural package* for further details.

Note: For brevity of presentation, some redundant (e.g. imports of libraries) or inessential pieces of the code are present only in the downloadable source Jupyter notebooks, and are not included/repeated in the book. This makes the text more concise and readable.

CHAPTER

TWO

MCP NEURON

2.1 Definition

We need a basic building block of ANNs: the artificial neuron. The first mathematical model dates back to Warren McCulloch and Walter Pitts (MCP)[MP43], 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 is a basic ingredient of all ANNs discussed in this course. It is built on very simple general rules, inspired neatly by the biological neuron:

- The signal enters the nucleus via dendrites from other neurons.
- The synaptic connection for each dendrite may have a different (and adjustable) strength (weight).
- 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 opposite case it remains still.
- In the simplest realization, the strength of the fired signal has two possible levels: on or off, i.e. 1 or 0. No intermediate values are needed.
- Axon terminal connects to dendrites of other neurons.

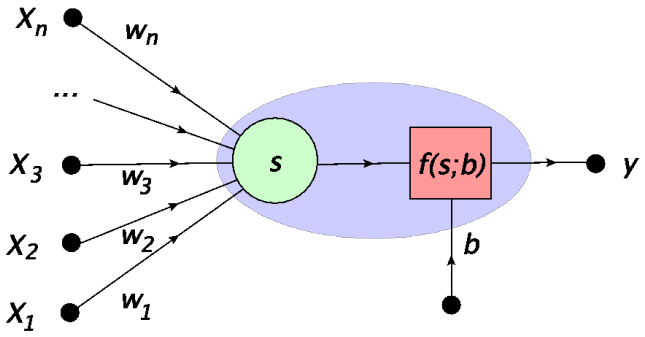


Fig. 2.1: MCP neuron: x_i is the input, w_i are the weights, s is the signal, b is the bias, and $f(s;b)$ represents an activation function, yielding the output $y = f(s;b)$. The blue oval encircles the whole neuron, as used e.g. in Fig. 1.3.

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

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

The signal becomes an argument of the **activation function**, which, in the simplest case, takes the 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 mimic the biological prototype.

There is a convenient notational convention that is frequently used. Instead of splitting the bias from the input data, we may treat all uniformly. The condition for firing may be trivially transformed as

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

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 the input always fixed 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:

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

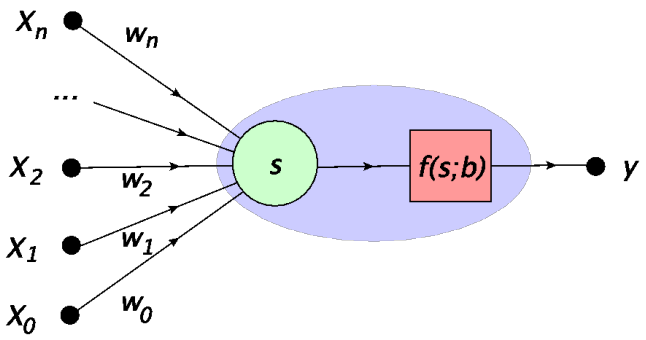


Fig. 2.2: Alternative, more uniform representation of the MCP neuron, with $x_0 = 1$ and $w_0 = -b$.

Hyperparameters

The weights $w_0 = -b, w_1, \ldots, w_n$ are generally 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 sample.

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

2.2 MCP neuron in Python

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

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

and (**important**) are indexed starting from 0, e.g.

```
x[0]
```

```
1
```

(note that typing a variable at the end of a notebook cell prints out its content). The numpy library functions carry the prefix **np**, which is the alias given at import. Note that these functions act *distributively* over arrays, e.g.

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

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

which is a very convenient feature when programming. We also have the scalar product $x \cdot w = \sum_i x_i w_i$ handy, which we use to build the combined signal s entering the MCP neuron:

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

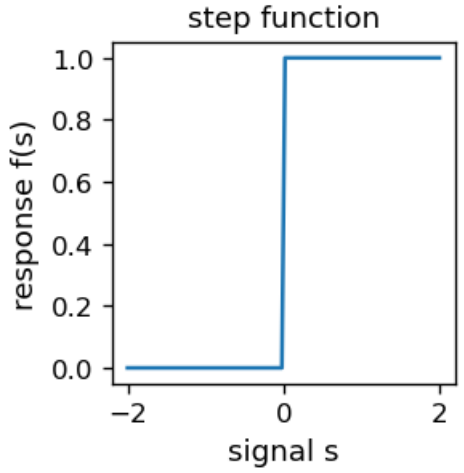
```
21.5
```

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

```
def step(s):        # step function
    if s > 0:       # condition satisfied
        return 1
    else:           # otherwise
        return 0
```

where in the top comments we have indicated that the function is also defined in the **neural** library package, 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 and resolution of the figure

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

plt.xlabel('signal s',fontsize=11)       # axes labels
plt.ylabel('response f(s)',fontsize=11)
plt.title('step function',fontsize=11)   # plot title

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



Since $x_0 = 1$ always, we do not want to explicitly carry this over in the arguments of functions that will follow. We will be frequently 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 into x, output f(x.w)
```

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

```
help(neuron)
```

```
Help on function neuron in module __main__:

neuron(x, w, f=<function step at 0x7ffe17063820>)
    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 function **f** is an argument of **neuron**. It is by default set to **step**, thus does not have to be present in the argument list. A 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, because $s = 1 * (-2) + 3 * 1 > 0$.

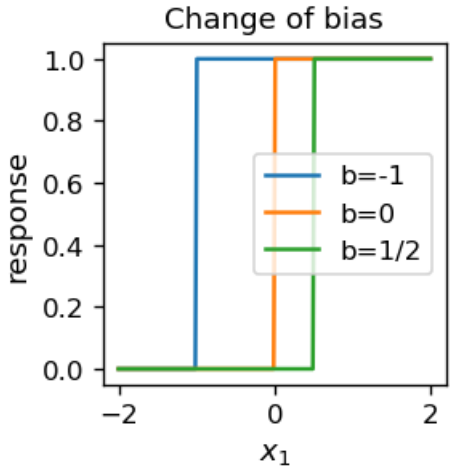
Next, we show how the neuron operates on an input sample x_1 taken in the range $[-2, 2]$. We also change the bias parameter, to illustrate its role. It is clear that the bias works as the threshold: if the signal $x_1 w_1$ is above $b = -x_0$, then the neuron fires.

```
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=11)
plt.ylabel('response',fontsize=11)

plt.title("Change of bias",fontsize=11)

plt.plot(s, fs1, label='b=-1')
plt.plot(s, fs0, label='b=0')
```

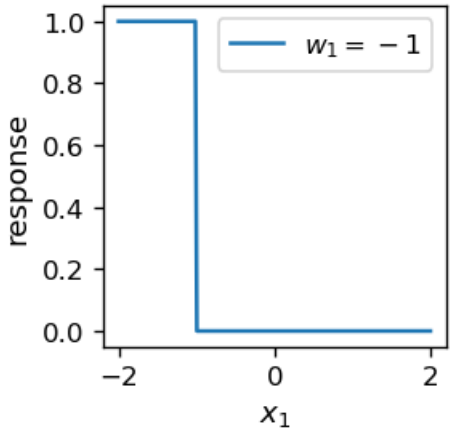
(continues on next page)

(continued from previous page)

```
plt.plot(s, fsm12, label='b=1/2')
plt.legend()      # legend
plt.show()
```



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



Note: From now on, for the brevity of the presentation, we hide some cells of the code with repeated structure. The reader may find the complete code in the corresponding Jupyter notebooks.

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 freedom enriches the mathematical model, which clearly can be built without biological 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 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()             # print space

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 have used 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 neuron 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 logical 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 the 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 make it behave as the XOR gate, but we are doomed to fail. Here is the reason:

From the first row of the above table, it follows that for the input 0, 0 (first row) 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()

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.

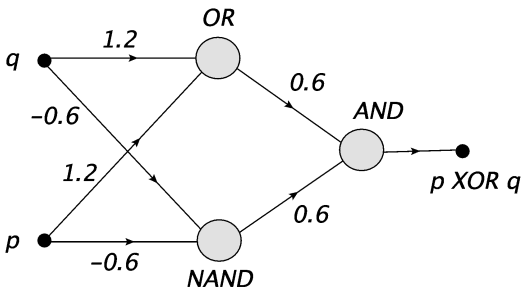


Fig. 2.3: The XOR gate composed of the NAND, OR, and AND MCP neurons.

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 any 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()

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
```

Note: One proves that logical networks are complete in the Church-Turing sense, i.e., (when sufficiently large) may carry over any possible calculation. This feature directly carries over to ANNs. Historically, that was the basic finding of the seminal MCP paper [MP43].

Conclusion

ANNs (sufficiently large) can perform any calculation!

2.4 Exercises

Construct (all in Python)

- a gate realizing conjunction of multiple boolean variables;
- gates NOT, NOR;
- gates OR, AND, NOT by composing NAND gates;
- the half adder and full adder,

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, describing two very simple models of memory based on linear algebra, supplemented with (nonlinear) filtering. Speaking of memory here, a word of caution is in place. We have rather simplistic tools in mind here, which are far from the actual complex and hitherto not comprehended memory mechanism operating in our brain. The present understanding is that these mechanism involve feed-back, which goes beyond the considered feed-forward networks.

The first considered 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, that is between different (hetero) symbols. Y remains 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.

We use the standard plotting package imported earlier. The whole set of our symbols 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

plt.show()
```

Warning: In Python, an integer range(i, j) includes i, but does not include j, i.e. equals $[i, i+1, \ldots, j-1]$. Also, range(i) includes $0, 1, \ldots, i-1$. This differs from conventions in some other programming languages.

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 flattening on our set,

```
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. Note that 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, as then they could be "confused".

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 $w^T v = w \cdot v$). Here T denotes transposition. The result is a matrix with the number of rows equal to the length of v, and the number of columns 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

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

(recall from algebra that we multiply "the columns by the rows"). 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 the 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 get $Mb = B$. 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 (do not have too many common pixels. As we will see shortly, 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, as $a \to A$, $A \to a$, $I \to i$, and $i \to I$:

```
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, as a test how it works, for each flattened symbol s we 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

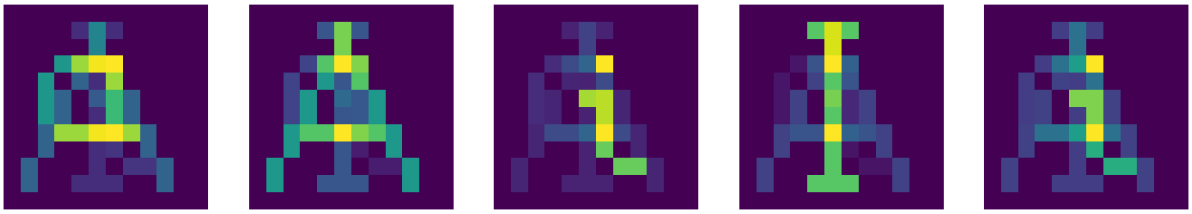
```
Ap=np.dot(M,fA).reshape(12,12)
ap=np.dot(M,fa).reshape(12,12)
Ip=np.dot(M,fI).reshape(12,12)
ip=np.dot(M,fi).reshape(12,12)
Yp=np.dot(M,fY).reshape(12,12) # we also try the unassociated symbol Y

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

For the case of association to A (which should be a), the procedure yields (we use rounding to 2 decimal digits with **np.round**)

```
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 above pixel maps. Looking at the above print out or the plot of Ap (the symbol associated to A which should be a), we note that we should get rid of the "faint shadows", and leave only the pixels of sufficient 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 an MCP neuron with weights $w_0 = -b$ and $w_1 = 1$:

```
def filter(a,b): # a - symbol (2-dim pixel array), b - bias
    n=len(a)     # number of rows (and columns)
```

(continues on next page)

(continued from previous page)

```
    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 assumed bias is very much relevant). This yields the result

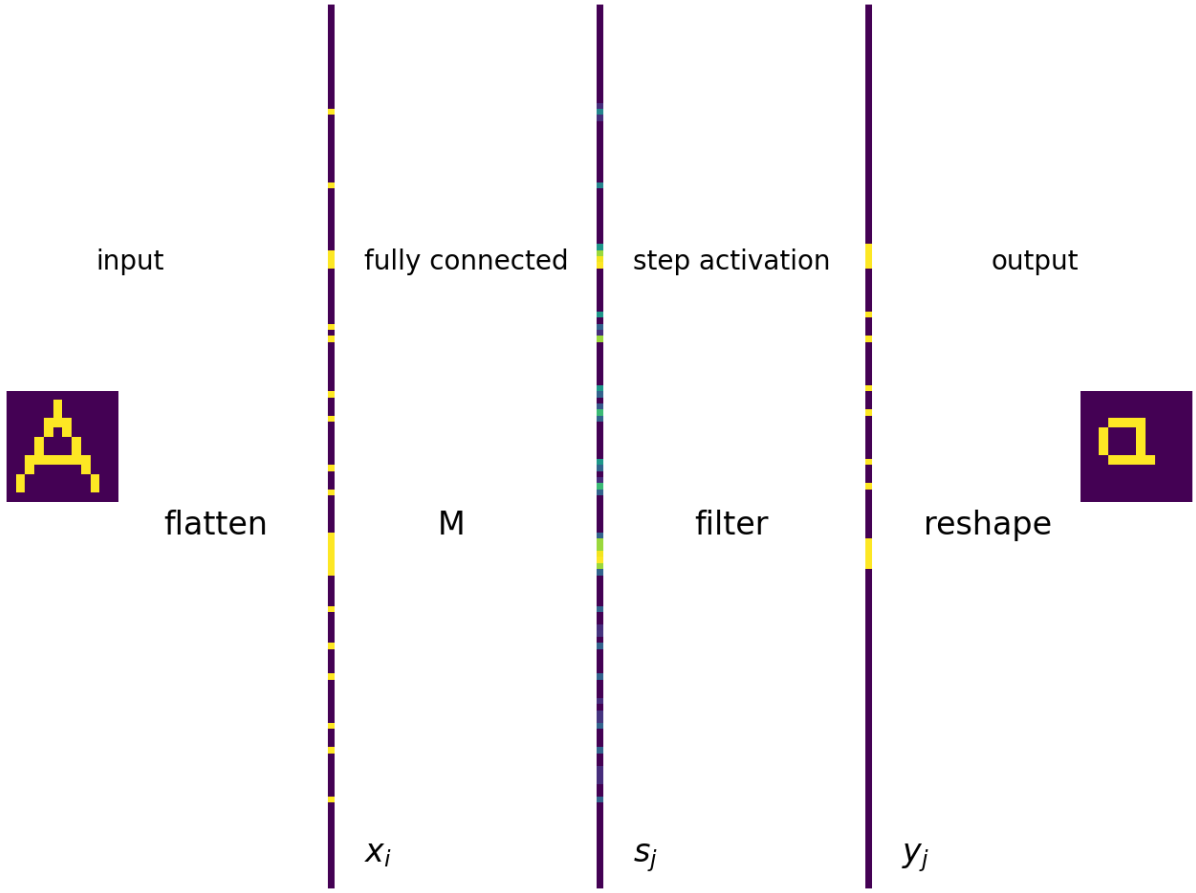
```
print(filter(Ap,.9))
```

```
[[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 associations (such perfection is not always the case):

A representation of the just presented model of the heteroassociative memory in terms of a **single-layer** network of MCP neurons 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 has the form of the 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. Flatten 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 **with 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 the original symbols (except for Y, which was not associated with anything).

3.2.2 Distorting the image

Now imagine that the original image 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]             # trick to switch 1 and 0
```

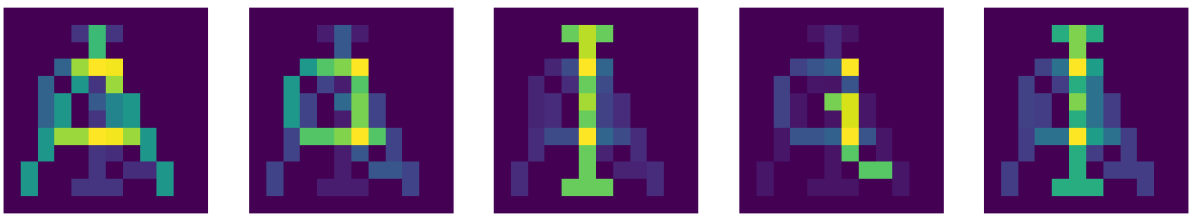
After this destruction, the input symbols look like this:

3.2.3 Restoring the symbols

We next apply our model of the autoassociative memory to all the "destroyed" symbols:

```
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]
```

which yields



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

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

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

plt.show()
```

The application of the algorithm can thus decipher a "destroyed" text, or, more generally, provide an error correction mechanism.

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. Flatten 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 original symbol.

Important: Message: ANNs with one layer of MPC neurons can serve as very simple models of memory!

Note, however, that we constructed the memory matrices algebraically, externally, so to speak. Hence, the network has not really learned the associations from experience. There are ways to do it, but they require more advanced methods (see, e.g., [FS91]), similar to those covered in the following parts of these lectures.

Note: An implementation of the discussed memory models in Mathematica is provided in [Fre93] (https://library.wolfram.com/infocenter/Books/3485) and in the already mentioned lectures by Daniel Kersten.

3.3 Exercises

Play with the lecture code and

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

Discuss your findings and the limitations of the models.

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

- a certain **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" a priori 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 the available data.

In this chapter we begin to explore such algorithms with the **supervised learning** approach, used i.a. 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 called the labels. The labeled data are then 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 through your ANN. The obtained answer may differ from the correct one, given 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 a test sample.

If satisfied, you have a desired trained ANN performing a specific task (like classification), 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 (which may be expensive), or change your network's architecture.

The term "supervised" comes form an 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** that way the training process. Of course, a computer program "supervises itself".

4.2 Perceptron as a binary classifier

The simplest supervised learning algorithm is the perceptron, invented in 1958 by Frank Rosenblatt. It can be used, i.a., to construct **binary classifiers** of the data. *Binary* means that the network is used to assess if a data item has a particular feature, or not - just two possibilities. Multi-label classification is also possible with ANNs (see exercises), but we do not discuss it in these lectures.

Remark

The term *perceptron* is also used for ANNs (without or with intermediate layers) consisting of the MCP neurons (cf. Fig. Fig. 1.3 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, correspondingly, 1 or 0. These labels are "true" answers.

The function generating the above described data point with a label is

```
def point():      # generates random coordinates x1, x2, and 1 if x2>x1, 0 otherwise
    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
[[0.30177711 0.36256864 1.        ]
 [0.74563557 0.63263509 0.        ]
 [0.46905767 0.66982076 1.        ]
 [0.04018207 0.20842031 1.        ]
 [0.04874155 0.21942538 1.        ]]
```

Loops in arrays

In Python, one can conveniently define arrays with a loop inside, e.g.

[i**2 for i in range(4)] yields [1,4,9].

In loops, if the index does not explicitly show in the expression, one can use a dummy index _, as for instance in the above code:

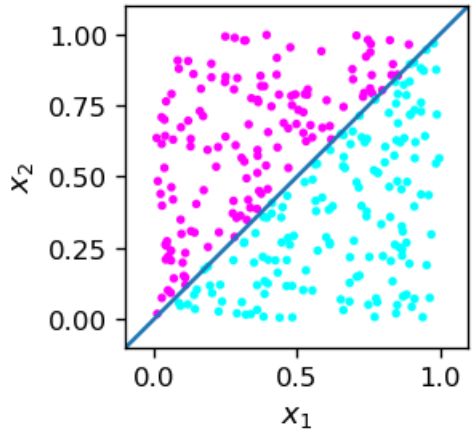
[point() for _ in range(npo)]

Ranges in arrays

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 (recall the last one is excluded!), :5 - from 0 to 4, 5: - from 5 to the end, and : - all the indices.

Graphically, our data are shown 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=11)
plt.ylabel('$x_2$',fontsize=11)

plt.show()
```



Linearly separable sets

Two sets of points (as here blue and pink) on a plane which are possible to separate with a straight 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 an $n-1$ dimensional hyperplane.

Analytically, 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 one the opposite condition, with $>$ replaced with $\leq$.

Now a crucial, albeit obvious 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 the **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 as follows:

```
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

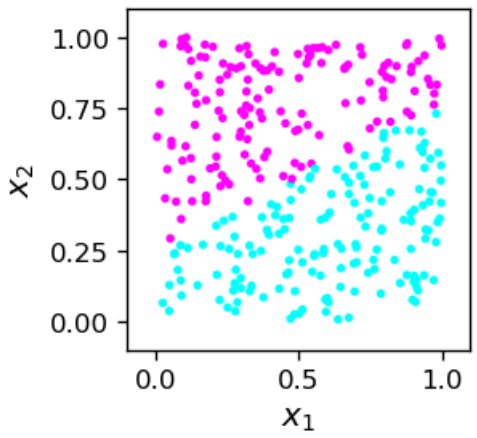
A single MCP neuron with properly chosen weights can be used as a binary classifier for n-dimensional separable data.

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 results. The confusion may stem from the fact that in the above 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 usually not the case! Imagine that we encounter the (labeled) data **samp2** looking like this:

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

```
[[0.39498232 0.30505482 0.        ]
 [0.11269714 0.95880028 1.        ]
 [0.29661374 0.11679508 0.        ]
 [0.89268244 0.35261853 0.        ]
 [0.89178862 0.25506102 0.        ]]
```



The situation is in some sense inverted now. We have obtained 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 binary 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 weights w_0, w_1, w_2 always yields 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      0
1      0
0      0
0      1
0      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 sufficiently 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 decreases. 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 analogously check that the recipe also works properly.

When the answer 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 training 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 only slightly).

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

The Python implementation of the perceptron algorithm for the 2-dimensional 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                     # initial  learning speed
```

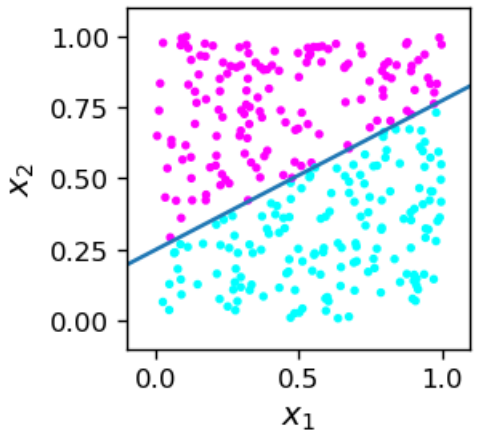
(continues on next page)

(continued from previous page)

```
for _ in range(20):          # loop over 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 (the perceptron formula)
            w1=w1+eps*(samp2[i,2]-yo)*samp2[i,0]
            w2=w2+eps*(samp2[i,2]-yo)*samp2[i,1]

print("Obtained weights:")
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
```

```
Obtained weights:
  w0      w1      w2
[-0.517 -1.084  2.071]
```

The obtained weights, as we know, define the dividing line. Thus, geometrically, the algorithm produces the dividing line as drawn below, together with the training sample as plotted above.



We can see that the algorithm works! All the pink points are above the dividing 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 perceptron algorithm converges if and only if the data are linearly separable.

We may now reveal our secret! The data of the training sample **samp2** were labeled at the time of creation with the rule

$$x_2 > 0.25 + 0.52 x_1,$$

which corresponds to the 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 these are not at all the same as the weights obtained from the training:

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

```
[-0.517 -1.084  2.071]
```

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 **equivalence class** of weights 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))
```

```
[2.068 2.085 2.071]
```

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 always some 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 decreases (see the exercises).

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** used to generate the training data 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:

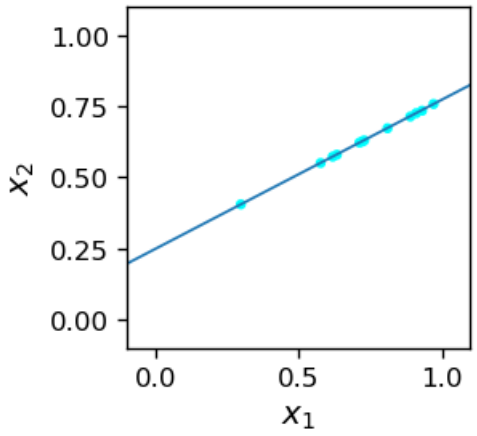
```
er= np.empty((0,3))  # initialize an empty 1 x 3 array to store misclassified␣
↪points

ner=0                  # initial number of misclassified points
nt=10000                # number of test points

for _ in range(nt):    # loop over the test points
    ps=point2()        # a test point
    if(func.neuron(ps[:2],[w0,w1,w2])!=ps[2]): # if wrong answer                ␣
↪
        er=np.append(er,[ps],axis=0)           # add the point to er
        ner+=1                                 # count the number of errors

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

(continues on next page)

(continued from previous page)

```
number of misclassified points =  12  per  10000  ( 0.1 % )
```

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 sample. For a better result, the training points would have to be "denser" in the vicinity of the separating line, or the training sample would have to be larger.

4.4 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 more than some value 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 function realizes the inequality $x \cdot w = w_0 + x_1 w_1 + \dots x_n w_n > 0$, where n in the dimensionality of the input space. It is instructive to follow up this geometric interpretation. Taking for definiteness $n = 2$ (the plane), the above inequality corresponds to a division into two half-planes. As we already know, the line given by the equation

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

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.

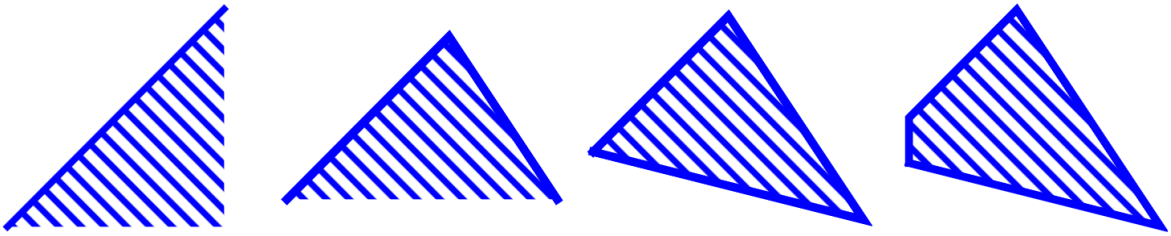


Fig. 5.1: Sample convex regions in the plane obtained, from left to right, with one inequality condition, and a conjunctions of 2, 3, or 4 inequality conditions, yielding **polygons**.

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.

Architectures of networks for $k = 1$, 2, 3, or 4 conditions are shown in Fig. 5.2. Going from left to right from the second panel, we have networks with two layers of neurons and 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 shown in the left panel of Fig. 5.2.

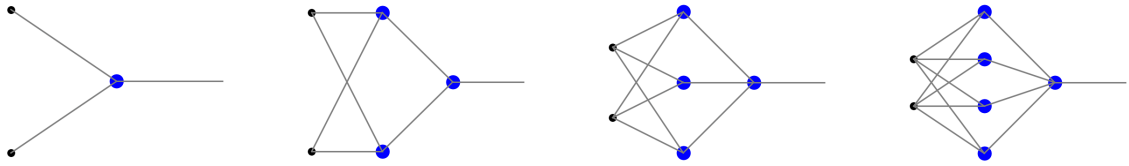


Fig. 5.2: Networks capable of classifying data in the corresponding regions of Fig. 5.1.

In 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.

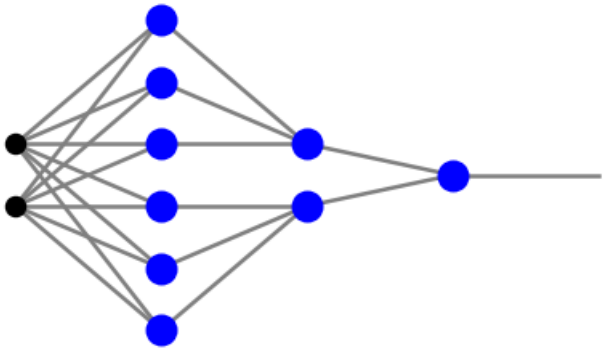
The situation generalizes in an obvious way to data in more dimensions. In that case we have more black dots in the inputs in Fig. 5.2. Geometrically, for $n = 3$ we deal with dividing planes and convex polyhedrons, and for $n > 3$ with dividing hyperplanes and convex polytopes.

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 neuron layers (with sufficiently many neurons in the intermediate layer) 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:

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

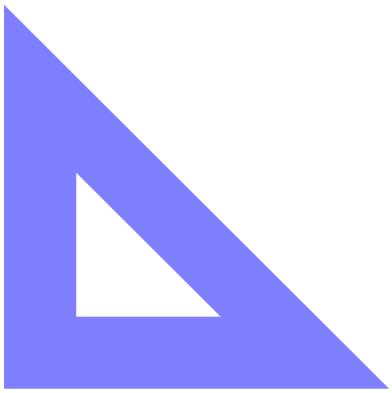


Fig. 5.3: Triangle with a tringular hollow.

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

Note: 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

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

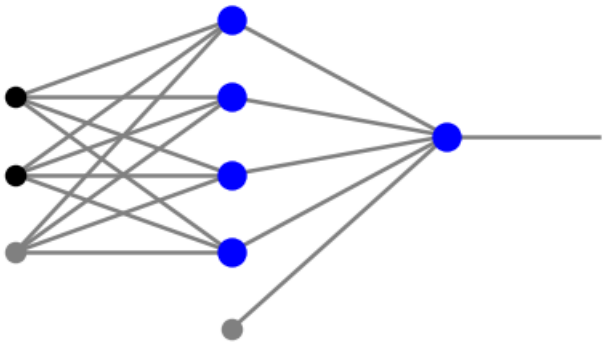
where n_0 in the number of the input nodes, and n_i are the numbers of neurons in layers $i = 1, \dots, l$. For instance, the architecture of the network from the fourth panel of Fig. 5.2 is

```
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 counting the numbers n_i defined above. In particular, a more detailed view of the fourth panel of Fig. 5.2 is

```
plt.show(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 (input) 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) neuron layer form a 4×1 matrix. Hence, in our convention, the weight matrices corresponding to subsequent neuron layers $1, 2, \ldots, l$ have dimensions

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

Thus, 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 somewhat more convenient to use the Python **dictionary** structure. We 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 can 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 correspondingly 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 output layer which has no bias node, hence x^l has the 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 a 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 (labeled with superscript i) and neurons (n_i in layer i), the notation generalizes into

$$s^i_\alpha = \sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha}, \quad \alpha = 1 ... n_i, \quad i = 1, ..., l.$$

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).

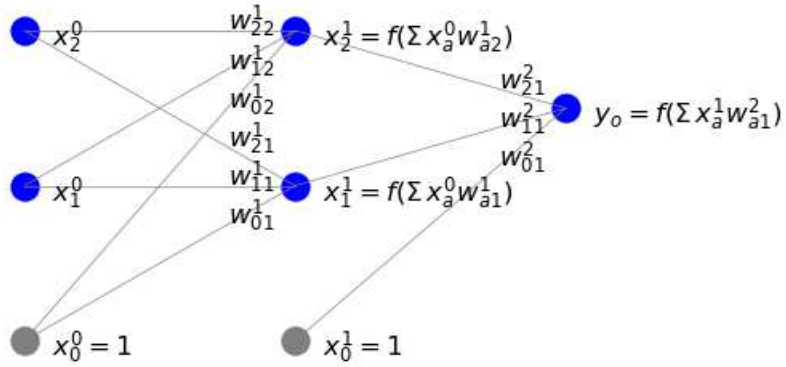
In the algebraic matrix notation, we can also write more compactly $s^{iT} = x^{(i-1)T} W^i$, with T denoting transposition. Explicitly,

$$\begin{pmatrix} s^i_1 & s^i_2 & ... & s^i_{n_i} \end{pmatrix} = \begin{pmatrix} x^{i-1}_0 & x^{i-1}_1 & ... & x^{i-1}_{n_{i-1}} \end{pmatrix} \begin{pmatrix} w^i_{01} & w^i_{02} & ... & w^i_{0,n_i} \\ w^i_{11} & w^i_{12} & ... & w^i_{1,n_i} \\ ... & ... & ... & ... \\ w^i_{n_{i-1}1} & w^i_{n_{i-1}2} & ... & w^i_{n_{i-1}n_i} \end{pmatrix}.$$

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

$$x^i_\alpha = f(s^i_\alpha) = f\left(\sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha} \right), \quad \alpha = 1 ... n_i, \quad i = 1, ..., l,$$
$$x^i_0 = 1, \quad i = 1, ..., l-1,$$

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



The implementation of the feed-forward propagation 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]
    (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)
```

(continues on next page)

(continued from previous page)

```
    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 the neuron layers
    x_in=np.insert(x_in,0,1)        # input, with the bias node inserted

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

    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 output layer l - 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
```

For brevity, we adopt a convention where we do not pass the input bias node of the input in the argument. It is inserted inside the function with **np.insert(x_in,0,1)**. As usual, we use **np.dot** for matrix multiplication.

Next, we test how **feed_forward** works on a sample input.

```
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 output from this network is obtained as

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

```
1
```

5.3.1 Digression on linear networks

Let us now make the following observation. Suppose we have a network with a linear activation function $f(s) = cs$. Then the last formula from the feed-forward derivation becomes

$$x^i_\alpha = c \sum_{\beta=0}^{n_{i-1}} x^{i-1}_\beta w^i_{\beta\alpha}, \;\; \alpha = 1 \dots n_i, \;\; i = 1, \dots, l,$$

or, in the matrix notation,

$$x^i = c x^{i-1} w^i.$$

Iterating this, we get for the signal in the output layer

$$x^l = c x^{l-1} w^i = c^2 x^{l-2} w^{l-1} w^l = \dots = c^l x^0 w^1 w^2 \dots w^l = x^0 W,$$

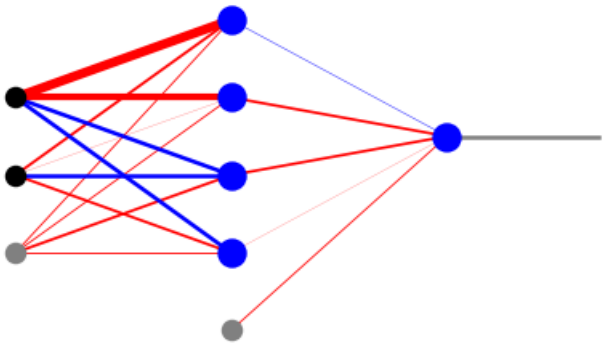
where $W = c^l w^1 w^2 \ldots w^l$. hence such a network is **equivalent** to a single-layer network with the weight matrix W as specified above.

Note: For that reason, it does not make sense to consider multiple layer networks with linear activation function.

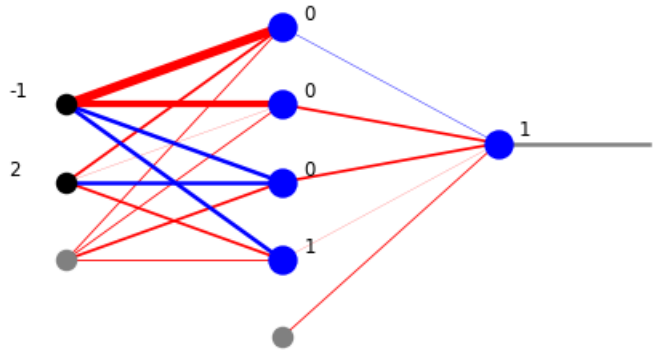
5.4 Visualization

For visualization of simple networks, in the **draw** module of he **neural** library we provide some drawing functions which show the weights, as well as the signals. Function **plot_net_w** draws the positive weights in red and the negative ones 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 nodes of each layer.

```
plt.show(draw.plot_net_w(arch,w,0.5))
```



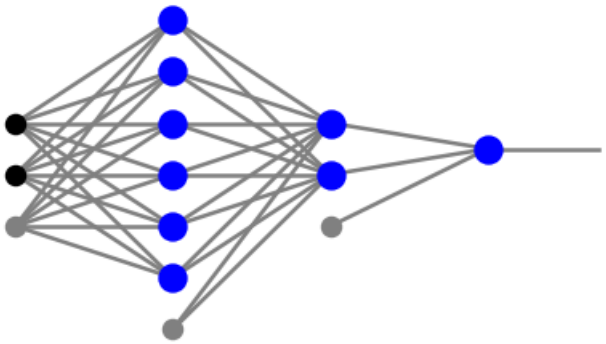
```
plt.show(draw.plot_net_w_x(arch,w,0.5,x))
```



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]
plt.show(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 triangle, 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, respectively, 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

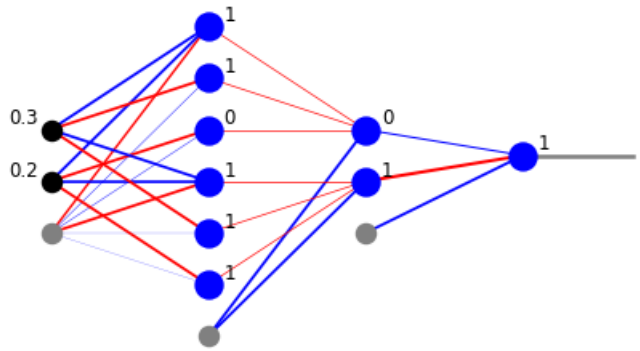
α	$w^3_{0\alpha}$	$w^3_{1\alpha}$	$w^3_{2\alpha}$
1	-1	1.2	-0.6

Putting all this together, 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)
plt.show(draw.plot_net_w_x(arch,w,1,x))
```



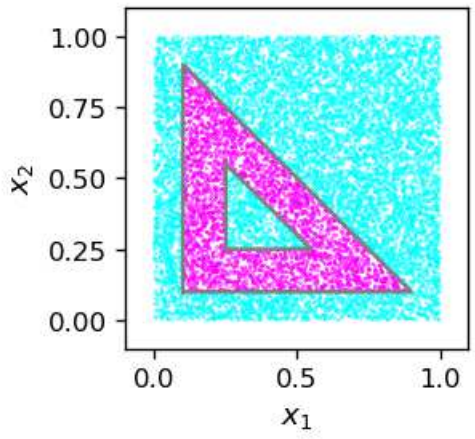
We have just found that point [0.2,0.3] is within our region (1 from the output layer). Actually, we we have more information from the intermediate layers. From the second neuron layer we know that the point belongs to the bigger triangle (1 from the lower neuron) and does not belong to the smaller triangle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

Next, we 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)             # 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])
```

```
[[0.71435539 0.05281912 0.        ]
 [0.54490285 0.59347251 0.        ]
 [0.95485103 0.77741598 0.        ]
 [0.71722458 0.42208055 0.        ]
 [0.54892879 0.62149029 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, as in the case of single-layer networks, there is an important argument against this apparent triviality. Imagine again we have a 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 exactly what teaching of classifiers will do: its sets the weights in such a way that the proper conditions are implicitly built in. After the material of this chapter, the reader should be convinced that this is perfectly possible and there is nothing magical about it! In the next chapter we will show how to do it.

5.6 Exercises

- Design a network and run the code from this lecture for a convex region of your choice.
- Design and program a classifier for four categories of points belonging to regions formed by 2 intersecting lines (hint: include four output cells).

CHAPTER

SIX

BACK PROPAGATION

In this chapter we show in detail how to carry out supervised learning for multilayer classifiers discussed in chapter *More layers*. Since the method is based on minimizing the number of erroneous answers on a test sample, we begin with a thorough discussion of the problem of error minimization in our setup.

6.1 Minimizing the error

Recall that in our example with points on the plane from chapter *Perceptron*, the condition for the pink points was given by the inequality

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

We have already briefly mentioned the equivalence class related to dividing this inequality with a positive constant c. In general, at least one of the weights in the condition 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 of the inequality with $|w_0|$ to obtain

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

Introducing the short-hand notation $v_1 = \frac{w_1}{w_0}$ 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 the sign $\text{sgn}(w_0) = \frac{w_0}{|w_0|}$, 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 data sample, the perceptron provides 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 interpreted as the number of misclassified points.

We can easily construct this function in Python:

```
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 - array of labeled data points p
             p in an array in the format [x1, x1, label]
    f - activation function

    returns:
```

(continues on next page)

(continued from previous page)

```
    error
    """
    er=0                                        # initial value of the 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
```

Actually, we have used a little trick here, in view of 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 formula programmed above. 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* to generate the labeled data sample **samp2** of 200 points (the sample is 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 $\mathrm{sgn}(w_0) = -1$).

We now need again the perceptron algorithm of section *Perceptron algorithm*. In our special case, it operates on a sample of two-dimensional labeled data. For convenience, a single round of the algorithm can be collected into a function as follows:

```
def teach_perceptron(sample, eps, w_in, f=func.step):
    """
    Supervised learning for a single perceptron (single MCP neuron)
    for a sample of 2-dim. labeled data

    input:
    sample - array of two-dimensional labeled data points p
             p is an array in the format [x1,x2,label]
             label = 0 or 1
    eps    - learning speed
    w_in   - initial weights in the format [[w0], [w1], [w2]]
    f      - activation function

    return: updated weights in the format [[w0], [w1], [w2]]
    """
    [[w0],[w1],[w2]]=w_in          # define w0, w1, and w2
    for i in range(len(sample)):   # loop over the whole sample
        for k in range(10):        # repeat 10 times

            yo=f(w0+w1*sample[i,0]+w2*sample[i,1]) # output from the neuron, f(x.w)

            # update of weights according to the perceptron algorithm formula
            w0=w0+eps*(sample[i,2]-yo)*1
            w1=w1+eps*(sample[i,2]-yo)*sample[i,0]
            w2=w2+eps*(sample[i,2]-yo)*sample[i,1]

    return [[w0],[w1],[w2]]        # updated weights
```

Next, we trace the action of the perceptron algorithm, watching how it modifies the error function $E(v_1, v_2)$ introduced above. We start from random weights, and then carry out 10 rounds of the above-defined **teach_perceptron** function, printing out the updated weights and the corresponding error:

```
weights=[[func.rn()], [func.rn()], [func.rn()]] # initial random weights

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

eps=0.7                 # initial learning speed
for r in range(10):     # rounds
    eps=0.8*eps         # decrease the learning speed
    weights=teach_perceptron(samp2,eps,weights,func.step)
                        # see the top of this chapter

    w0_o=weights[0][0]  # updated weights and ratios
    v1_o=weights[1][0]/weights[0][0]
    v2_o=weights[2][0]/weights[0][0]

    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.step),0))
```

```
Optimum:
   w0  w1/w0  w2/w0 error
-0.297 3.643 -7.231 22.0
-0.745 2.493 -3.373 28.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
-0.745 2.201 -4.032 0.0
```

We note that in subsequent rounds the error gradually decreases (depending on a simulation, it can sometimes jump up if the learning speed is too large, but this is not a problem as long as we can get down to the minimum), reaching a final value which is very small or 0 (depending on the particular instance of simulation). Therefore the perceptron algorithm, as we already experienced in chapter *Perceptron*, **minimizes the error on the training sample**.

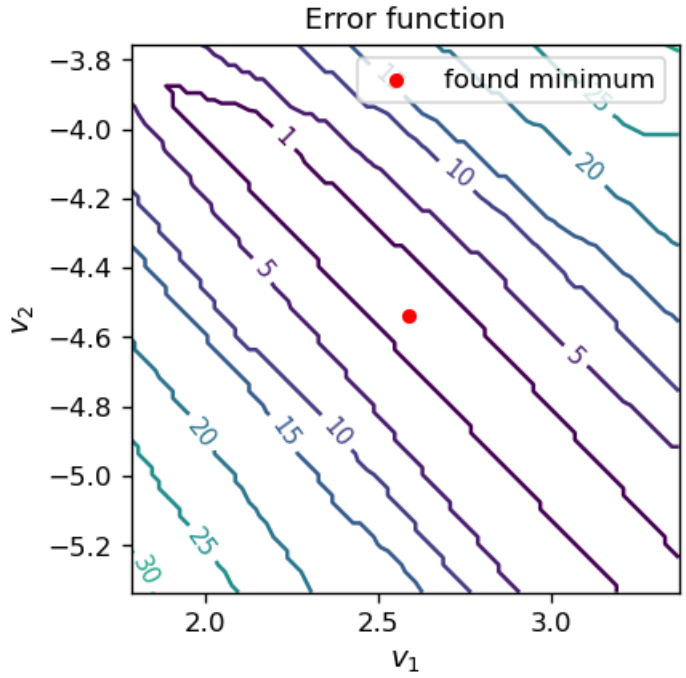
It is illuminating to look at a contour map of the error function $E(v_1, v_2)$ in the vicinity of the optimal parameters:

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

delta = 0.02  # grid step in v1 and v2 for the contour map
ran=0.8       # 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)               # mesh for the contour plot

Z=np.array([[error(-1,-v1[i],-v2[j],samp2,func.step)
             # we use the scaling property of the error function here
             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])
                         # explicit contour level values

ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=9) # contour label format

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)
```

(continues on next page)

(continued from previous page)

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

ax.legend()
plt.show()
```



The obtained minimum is inside (or close to, depending on the simulation) an elongated region of v_1 and v_2 where the error vanishes.

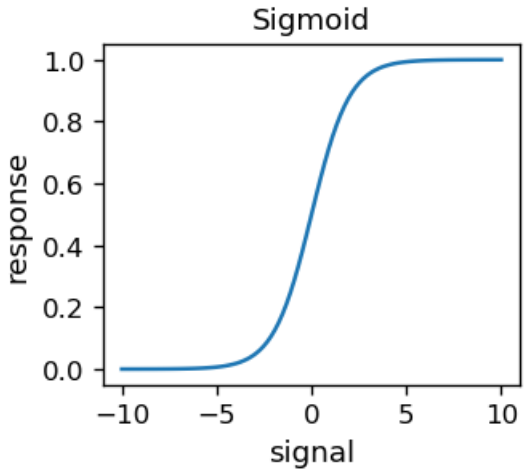
6.2 Continuous activation function

Looking closely at the contour map 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 hence non-differentiable. The discontinuities 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 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))
```

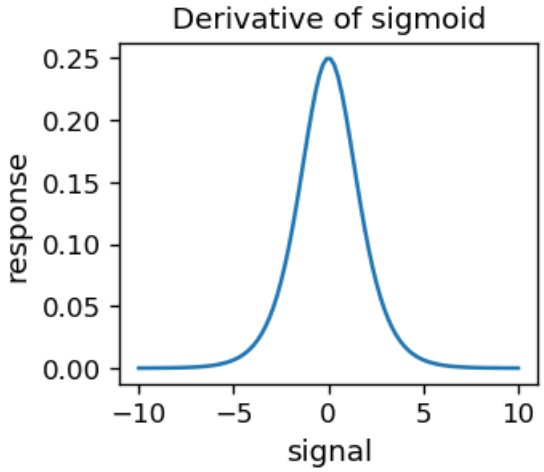


This function is of course differentiable. Moreover,

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

which is its special feature.

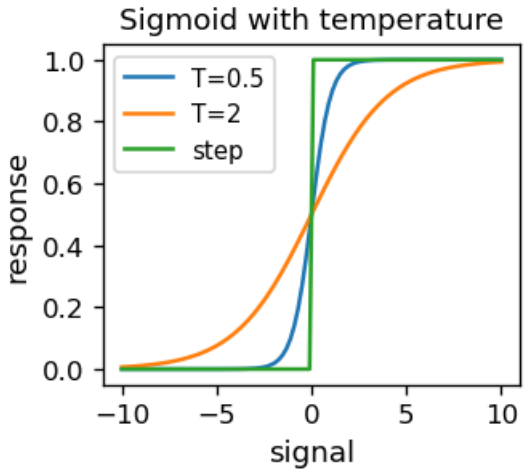
```
# derivative of sigmoid
def dsig(s):
     return sig(s)*(1-sig(s))
```



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))
```



For smaller and smaller T, the sigmoid 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 take $T = 1$ without losing generality (T is the "scale"). However, we now have three independent arguments ξ_0, ξ_1, and ξ_2, so it is impossible to reduce the present situation to only two independent parameters, as was the case in the previous section.

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

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

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 algorithm with the sigmoid activation function 1000 times, printing out every 100th step:

```
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                    # ratios of weights
        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
-16.036 2.665 -4.64 0.37383
-19.825 2.627 -4.588 0.25397
-22.375 2.595 -4.547 0.19764
-24.308 2.572 -4.52 0.16604
-25.869 2.555 -4.501 0.14589
```

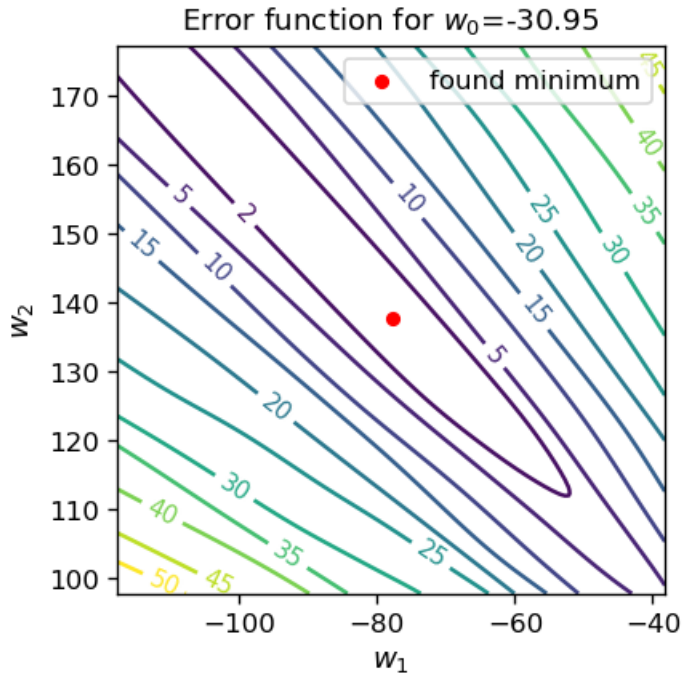
(continues on next page)

(continued from previous page)

```
-27.178 2.541 -4.487 0.13176
-28.306 2.531 -4.476 0.12106
-29.293 2.523 -4.467 0.11244
-30.169 2.516 -4.46 0.10517
-30.952 2.511 -4.454 0.09883
```

We notice, as expected, a gradual decrease of the error as the simulation proceeds. Since the error function now has three independent arguments, it cannot be drawn in two dimensions. We can, however, show its projection, e.g. with a fixed value of w_0, which we do below:

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

delta = 0.5
ran=40
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='found minimum') # our found optimal␣
 ↪point

ax.legend()
plt.show()
```



Note: As 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 all the way to zero. In the case of the sigmoid, there is always some (tiny) contribution to the error, as the values of the function are in the range (0,1). 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 T. Then, ultimately, the sigmoid approaches the step function, and the error tends to zero. Precisely this behavior is seen in the simulations above.

6.3 Steepest descent

The reason of carrying out the above simulations was to drive the reader into conclusion that the issue of optimizing the weights can be reduced to a generic problem of minimizing a multi-variable function. This is a standard (though in general difficult) problem in mathematical analysis and numerical methods. Issues associated with finding the minimum of multivariable functions are well known:

- there may be local minima, and therefore it may be very difficult to find the global minimum;
- the minimum can be at infinity (that is, it does not exist mathematically);
- The function around the minimum can be very flat, such that the gradient is very small, and the update in gradient methods is extremely slow;

Overall, numerical minimization of functions is an art! Many methods have been developed and a proper choice for a given problem is crucial for success. Here we apply the simplest variant of the **steepest descent** method.

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

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

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 is given by an iterative algorithm, where we update the coordinates (of a searched minimum) at each iteration step m (shown in superscripts) with

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

We need to minimize the error function

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

The **chain rule** is used to evaluate the derivatives.

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.

This yields

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

(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 far from zero only around $s^{(p)} = 0$ (see the sigmoid's derivative plot earlier), a significant updating occurs only near the threshold. This is fine, as the "problems" with misclassification happen near the dividing line.

Note: In 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 with all the points from the sample, not just near the threshold.

The code for the learning algorithm of our perceptron with the steepest descent update of weights is following:

```
def teach_sd(sample, eps, w_in): # Steepest descent for the perceptron

    [[w0],[w1],[w2]]=w_in              # initial weights
    for i in range(len(sample)):       # loop over the data sample
        for k in range(10):            # repeat 10 times

            yo=func.sig(w0+w1*sample[i,0]+w2*sample[i,1])  # obtained answer for␣
↪pont i

            w0=w0+eps*(sample[i,2]-yo)*yo*(1-yo)*1             # update of weights
            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 the original 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]
        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
-8.15 2.655 -4.624 1.15014
-10.036 2.651 -4.617 0.81162
-11.286 2.64 -4.601 0.66387
-12.231 2.629 -4.587 0.57811
-12.992 2.62 -4.575 0.52091
-13.63 2.612 -4.565 0.47946
```

(continues on next page)

(continued from previous page)

```
-14.178 2.605 -4.556 0.44769
-14.658 2.598 -4.549 0.42237
-15.084 2.593 -4.542 0.40159
-15.466 2.588 -4.536 0.38416
```

To summarize the development up to now, we have shown that one can teach a single-layer perceptron (single MCP neuron) efficiently with the help of the steepest descent method, minimizing the error function generated with the test sample. In the next section we generalize this idea to any multi-layer feed-forward ANN.

6.4 Backprop algorithm

The material of this section is absolutely **crucial** for understanding of the idea of training neural networks via supervised learning. At the same time, it can be quite difficult for a reader less familiar with mathematical analysis, as there appear derivations and formulas with rich notation. However, we could not find a way to present the material in a simpler way than below, keeping the necessary rigor.

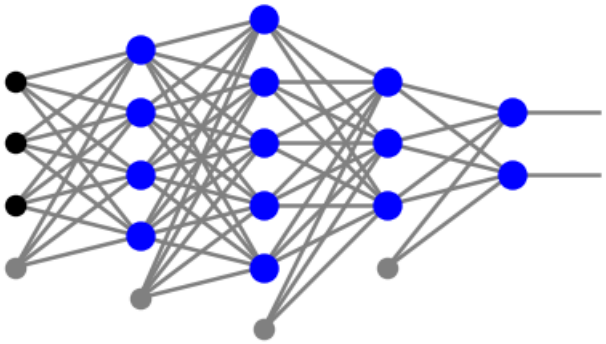
Note: The formulas we derive step by step here constitute the famous **back propagation algorithm (backprop)** [BH69] for updating the weights of a multi-layer perceptron. It uses just two ingredients:

- the **chain rule** for computing the derivative of a composite function, already used above, and
- the **steepest descent method**, explained in the previous lecture.

Consider a perceptron with any number of neuron layers, l. The neurons in intermediate layers $j = 1, \dots, l-1$ are numbered with corresponding indices $\alpha_j = 0, \dots, n_j$, with 0 indicating the bias node. In the output layer, having no bias node, the numbering is $\alpha_l = 1, \dots, n_l$. For example, the network form the plot below has

$$l = 4, \quad \alpha_1 = 0, \dots, 4, \quad \alpha_2 = 0, \dots, 5, \quad \alpha_3 = 0, \dots, 3, \quad \alpha_4 = 1, \dots, 2,$$

with the indices in each layer counted from the bottom.



The error function is a sum over the points of the training sample and, additionally, over the nodes in the output layer:

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

where $\{w\}$ represent all the network weights. A single point contribution to E, denoted as e, is a sum over all the 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

$$s^j_{\alpha_j} = \sum_{\alpha_{j-1}=0}^{n_{j-1}} x^{j-1}_{\alpha_{j-1}} w^j_{\alpha_{j-1}\alpha_j}.$$

The outputs from the output layer are

$$y_{o,\alpha_l} = f\left(s^l_{\alpha_l}\right)$$

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

$$x^j_{\alpha_j} = f\left(s^j_{\alpha_j}\right), \quad \alpha_j = 1, \ldots, n_j, \quad \text{and} \quad x^j_0 = 1,$$

with the bias node having the value 1.

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

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

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

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=0}^{n_{l-2}} x^{l-2}_{\alpha_{l-2}} w^{l-1}_{\alpha_{l-2}\alpha_{l-1}}\right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\gamma}\right) - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\sum_{\alpha_{l-2}=1}^{n_{l-2}} f\left(\sum_{\alpha_{l-3}=0}^{n_{l-3}} x^{l-3}_{\alpha_{l-3}} w^{l-2}_{\alpha_{l-3}\alpha_{l-2}}\right) w^{l-1}_{\alpha_{l-2}\alpha_{l-1}} + x^{l-2}_0 w^{l-1}_{0\alpha_{l-1}}\right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\alpha_l}\right) - y_{t,\alpha_l}\right)^2$

$= \sum_{\alpha_l=1}^{n_l} \left(f\left(\sum_{\alpha_{l-1}=1}^{n_{l-1}} f\left(\ldots f\left(\sum_{\alpha_0=0}^{n_0} x^0_{\alpha_0} w^1_{\alpha_0\alpha_1}\right) w^2_{\alpha_1\alpha_2} + x^1_0 w^2_{0\alpha_2} \ldots\right) w^l_{\alpha_{l-1}\alpha_l} + x^{l-1}_0 w^l_{0\alpha_l}\right) - y_{t,\alpha_l}\right)^2$

Calculating successive derivatives with respect to the weights, and going backwards, i.e. from $j = l$ down to 1, we get (see exercises)

$$\frac{\partial e}{\partial w^j_{\alpha_{j-1}\alpha_j}} = x^{j-1}_{\alpha_{j-1}} D^j_{\alpha_j}, \quad \alpha_{j-1} = 0, \ldots, n_{j-1}, \;\; \alpha_j = 1, \ldots, n_j,$$

where

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

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

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 previous step, as well as the signal s^j, which we know from the feed-forward stage. This recurrence 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 the 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.

Important: The significance of going back layer-by-layer is that one updates much less weights in one step: just those entering the layer, and not all of them. This has significance for convergence of the steepest descent method, especially for deep networks.

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

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

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

This is not infrequent, as for many applications one selects different activation function for the intermediate and the output layers.

6.4.1 Code for backprop

Next, we present a simple code that carries out the backprop algorithm. It is a straightforward implementation of the formulas derived above. In the code, we keep as much as we can 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

    # 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

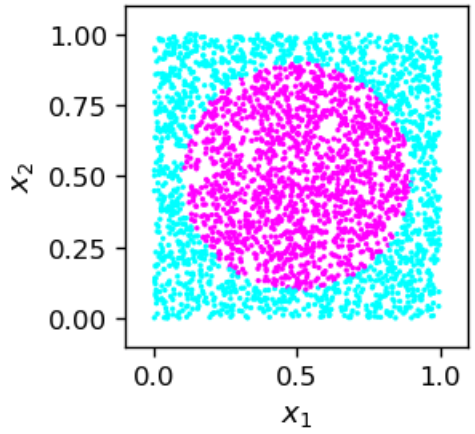
We illustrate the code on the example of a binary classifier of points inside a circle.

```
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 circle, radius 0.4, center (0.
↪5,0.5)
        return np.array([x1,x2,1])
    else:                                    # outside
        return np.array([x1,x2,0])
```

For a future use **(new convention)**, we split the sample into separate arrays of **features** (the two coordinates) and **labels** (1 if the point is inside the circle, 0 otherwise):

```
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)
plt.show()
```



We choose the following architecture and initial parameters:

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

The simulation takes a few minutes,

```
for k in range(1000):    # 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  400  500  600  700  800  900  1000
```

The reduction of the learning speed in each round gives the final value, which is small, but not too small:

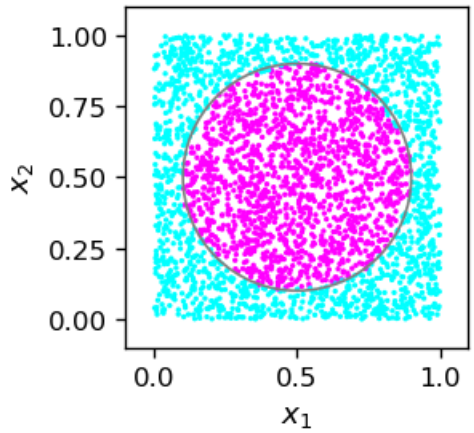
```
eps
```

```
0.004657778005182377
```

(a too small value would update the weights very little, so further rounds would be useless).

While the learning phase was rather long, the 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)
plt.show()
```



The trained network looks like this:

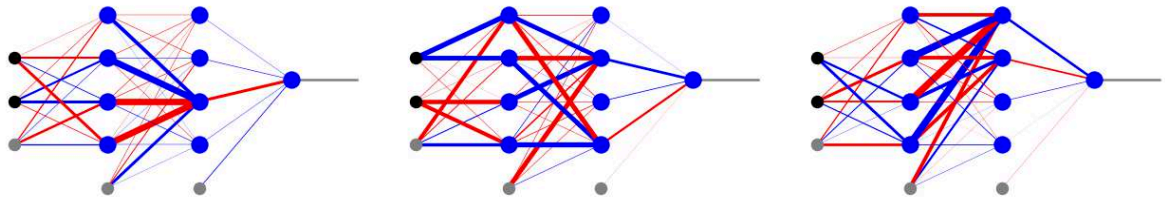
```
fnet=draw.plot_net_w(arch_c,weights,.1)
```

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 has just learned "empirically" how to proceed, using a training sample!

Note: The result in the plot is pretty 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. As always in these games, improving the result would require a bigger sample and longer training, which is time consuming.

Local minima

We have mentioned before the emergence of local minima in multi-variable 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 optimum weights, whereas the results on the test sample are, at least by eye, equally good for each case. This shows that the backprop optimization ends up, as anticipated, in different local minima. However, each of these local minima works sufficiently well and equally good. This is actually the reason why backprop can be used in practical problems: there are zillions of local minima, but it does not really matter!



6.6 General remarks

There are some more important and general observations to be made:

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**) and evaluations of derivatives in many rounds (we have used 1000 for the circle example), but the usage on a point involves just one function call.
- A classifier trained with backprop may work inaccurately for the points near the boundary lines. A remedy is to train more for improvement, and/or increase the size of the training sample, in particular near the boundary.
- 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 backprop, but this is by no means an obstacle to the use of the algorithm. This is an important practical feature.
- Various improvements of the steepest descent method, or altogether different minimization methods may be used (see exercises). They can largely increase the efficiency of the algorithm.
- When going backwards with updating the weights in subsequent layers, one may introduce an increasing factor (see exercises). This helps with performance.
- Finally, different activation functions may be used to improve performance (see the following lectures).

6.7 Exercises

1. Prove (analytically) by evaluating the derivative that $\sigma'(s) = \sigma(s)[1 - \sigma(s)]$. Show that the sigmoid is the **only** function with this property.
2. Derive backprop formulas for network with one- and two intermediate layers. Note an emerging regularity (recurrence), and prove the general formulas for any number of intermediate layers.
3. Modify the lecture example of the classifier of points in a circle by replacing the figure with
   - semicircle;
   - two disjoint circles;
   - ring;
   - any of your favorite shapes.
4. Repeat 3., experimenting with the number of layers and neurons, but remember that a large number of them increases the computational time and does not necessarily improve the result. Rank each case by the fraction of misclassified points in a test sample. Find an optimum/practical architecture for each of the considered figures.
5. 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, …). Besides, such a network after destruction can be (already with a smaller number of connections) retrained. Take your trained network from problem 3. and remove one of its **weak** connections (first, find it by inspecting the weights), setting the corresponding weight to 0. Test this damaged network on a test sample and draw conclusions.
6. **Scaling weights in back propagation.** A disadvantage of using the sigmoid in the backprop algorithm is a very slow update of weights in layers distant from the output layer (the closer to the beginning of the network, the slower). A remedy here is a re-scaling of 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, and in the subsequent 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 compensating factors 4^n: $4, 16, 64, 256, ...$ Another heuristic argument [RIV91] suggests even faster growing factors of the form 6^n: $6, 36, 216, 1296, ...$
   - Enter the above two recipes into the code for backprop.
   - Check if they indeed improve the algorithm performance for deeper networks, for instance for the circle point classifier, etc.
   - For assessment of performance, carry out the execution time measurement (e.g., using the Python **time** library packet).
7. **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 faster 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 the 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 examples 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 popular in scientific data analyses. We illustrate the method on a simple example, which explains the basic idea and shows how the method works.

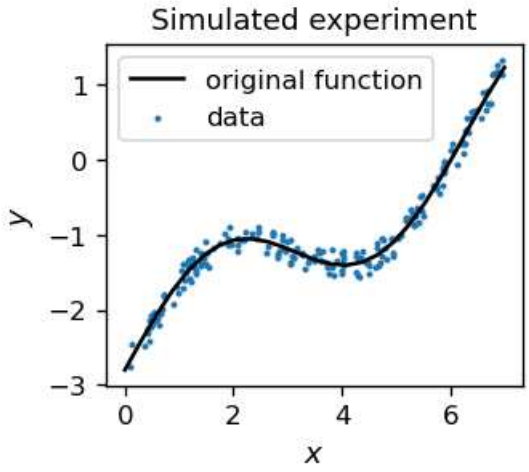
Imagine you have some experimental data. Here we simulate them in an artificial way, e.g.

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

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

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

We table our noisy data points and plot them together with the function **fi(x)** around which they fluctuate. It is an imitation of an experimental measurement, which is always burdened with some error, here mimicked with a 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 language of ANNs, we therefore have a training sample consisting of points with the input (feature) x and the

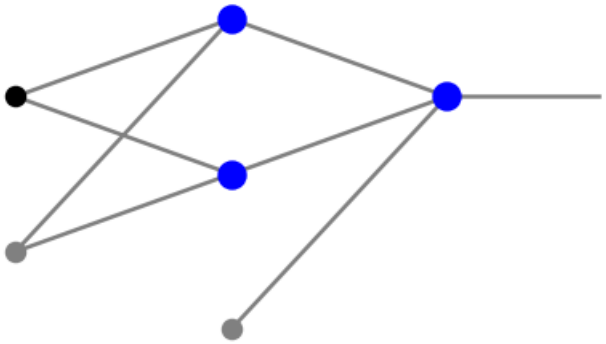
true output (label) y. As before, we minimize the error function from an appropriate neural network,

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

Since the generated 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. $f(x) = A + Bx$), whereas 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

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

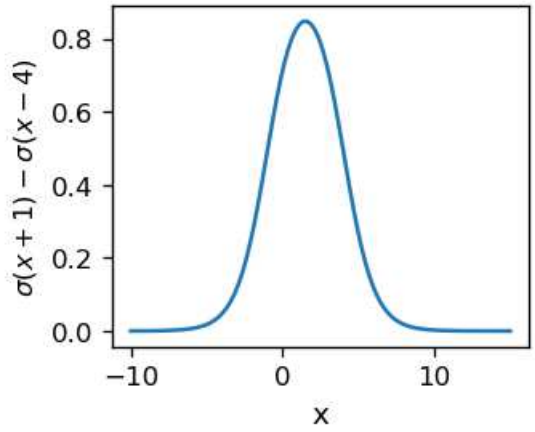
Taking, for illustration, the sample weight values

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

where x_1 and x_2 is a short-hand notation, 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, so one can say that the span of the function is between x_1 and x_2.



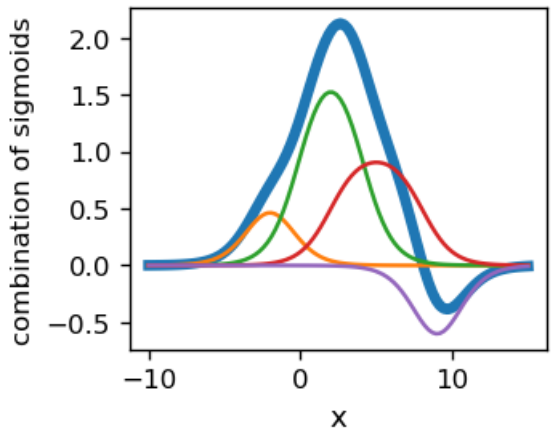
This is a straightforward but important finding: We are able to form, with a pair of neurons with sigmoids, a "hump" signal located around a given value, here $(x_1 + x_2)/2 = 2$, and with a a given spread of the order of $|x_2 - x_1|$. Changing the weights, we are able to modify its shape, width, and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

One may now think as follows: Imagine we have many neurons to our disposal in the intermediate layer. We can join them in pairs, forming humps "specializing" in particular regions of coordinates. Then, adjusting the heights of the humps, we can 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 the parameters simultaneously, 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.



Note: If the fitted function is regular, one expects that it can be approximated with a linear combination of sigmoids. With more sigmoids, a better accuracy can be accomplished.

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 activation function in the output layer, or rather its smooth sigmoid variant. For function approximation, the answers typically form a continuum in the range of the function values. For that reason, in the output layer we just use the **identity** function, i.e., we just pass the incoming signal through. Of course, sigmoids remain to operate in the intermediate layers. Then, the formulas used for backprop are those from section *Backprop algorithm*, with $f_l(s) = s$ in the output layer.

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

Let us take the architecture:

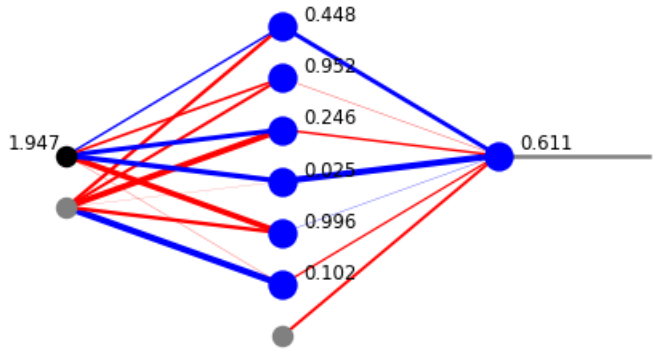
```
arch=[1,6,1]
```

and the random weights

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

As just discussed, the output is no longer between 0 and 1, as we can see below.

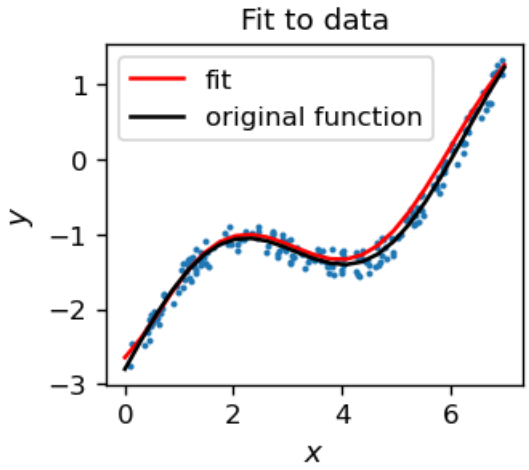
```
x=func.feed_forward_o(arch, weights,features[1],ff=func.sig,ffo=func.lin)
plt.show(draw.plot_net_w_x(arch, weights,1,x))
```



In the library module **func** we have the function for the backprop algorithm which allows for one activation function in the intermediate layers (we take sigmoid) and a different one in the output layer (we take the identity function). The training is carried out in two stages: First, we take the points from the training sample in a random order, and than we sweep over all the points sequentially, also decreasing the learning speed in subsequent rounds. This strategy is one of many, but here it nicely does the job:

```
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)
```



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 an exercise.

Note: The activation function in the output layer may be any smooth 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.

We may understand this as follows [MullerRS12]: Two neurons in the first neuron layer can form a hump in the x_1-coordinate, two other ones a hump in the x_2-coordinate, and so on for all the dimensions n. Taking a conjunction of these n humps in the second neuron layer yields a "basis" function specializing in a region around a certain point in the input space. A sufficiently large number of such basis functions can be used for approximating in n dimensions, in full analogy to the one-dimensional case.

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

Overfitting

There must be much more data for fitting than the network parameters, to avoid the so-called **overfitting problem**. Otherwise we could fit the data with a function "fluctuating from point to point".

7.3 Exercises

1. Fit the data points generated by your favorite function (of one variable) with noise. Play with the network architecture and draw conclusions.
2. Compute the sum of squared distances between the values of the data points and the corresponding approximating function, and use it as a measure of the goodness of the fit. 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 similar if everything works).

CHAPTER

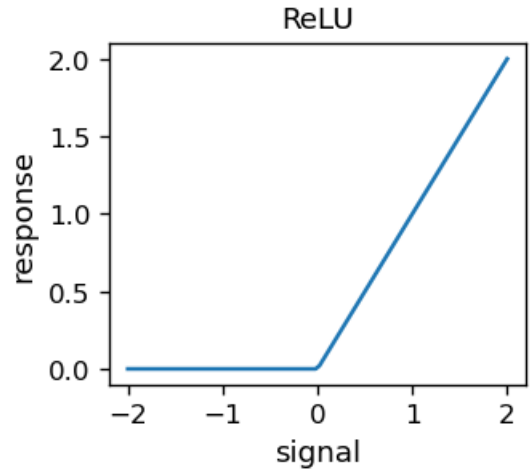
EIGHT

RECTIFICATION

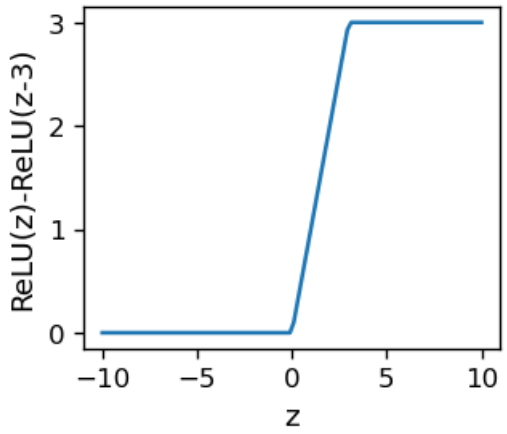
In the previous chapter we made a hump function from two sigmoids, which would form a basis function for approximation. We may now ask a follow-up question: can we make the sigmoid itself a linear combination (or simply difference) of some other functions. Then we could use these functions for activation 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 somewhat awkward name comes from electonics, where a "rectifying" (straightening up) unit is used to cut off negative values of an electric signal. The plot of ReLU looks as follows:



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

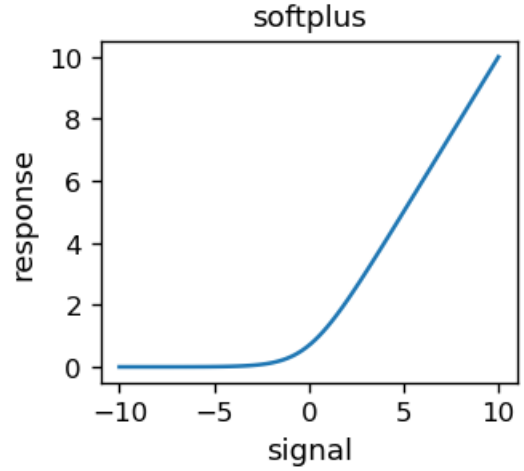


which looks pretty much as a sigmoid, apart from the sharp corners. One can make things smooth by taking a different

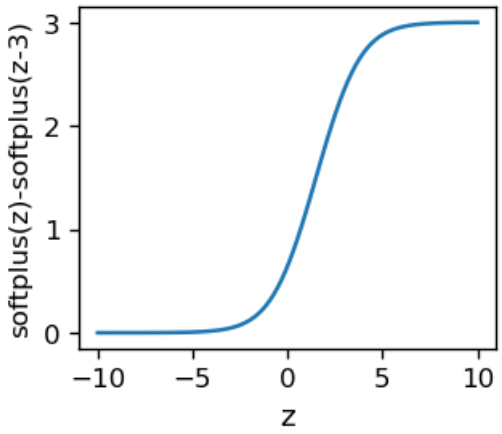
function, the **softplus**,

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

which looks like



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



Note: One may 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 can approximate our simulated data with an ANN with ReLU acivation in the intermediate layers (and the identity function is the output layer, as in the previous section). The functions are taken from the module **func**.

```
fff=func.relu     # short-hand notation
dfff=func.drelu
```

The network must now have more neurons, as the sigmoid "splits" into two ReLU functions:

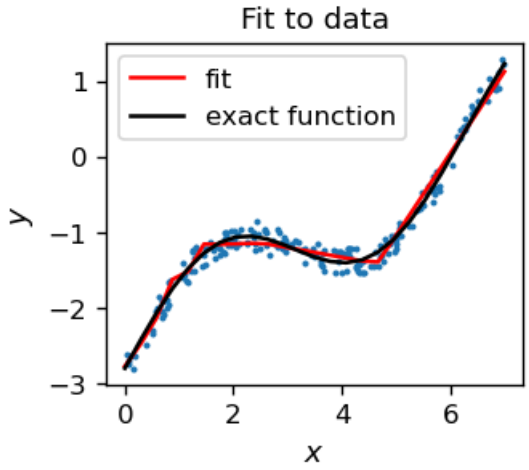
```
arch=[1,30,1]                    # architecture
weights=func.set_ran_w(arch, 5) # initialize weights randomly in [-2.5,2.5]
```

We carry the simulations exactly as in the previous case. Experience says one should stat with small learning speeds. Two sets of rounds (as in the previous chapter)

```
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*.995
    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
```

yield the result



We obtain again a quite satisfactory result (red line), noticing that the plot of the fitting function is a sequence of straight lines, simply reflecting the features 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 the 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 then very small numbers multiply yielding essentially no update (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.

Note: Application of rectified activation functions was one of the key tricks that allowed a breakthrough in deep ANNs around 2011.

On the other hand, with ReLU it may happen that some weights are set to 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” or “dead body” problem, 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 at all a range with zero derivative, such as the Leaky ReLU. Here

we take it in the form

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

For illustration, we repeat our example from section *Example with the circle* with the classification of points in the circle, now with Leaky ReLU.

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 in [-1.5,1.5]
eps=.01                              # initial learning speed
```

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

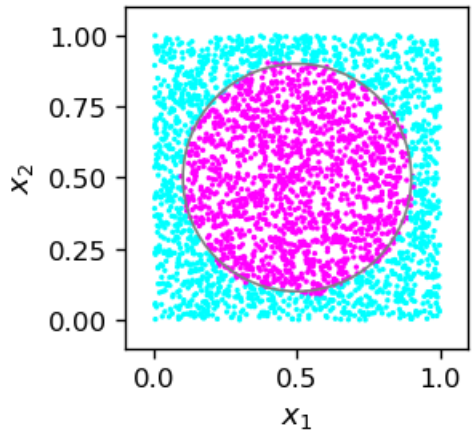
```
for k in range(300):     # rounds
    eps=.9999*eps        # decrease the learning speed
    if k%100==99: print(k+1,' ',end='')              # print progress
    for p in range(len(features_c)):                 # loop over points
        func.back_prop_o(features_c,labels_c,p,arch_c,weights,eps,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

```
100  200  300
```

```
for k in range(700):     # rounds
    eps=.9999*eps        # decrease the 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 with ReLU
```

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

The result is quite satisfactory, showing that the method works. With the present architecture and activation functions, not surprisingly, in the plot below we can notice traces of a polygon approximating the circle.



8.3 Exercises

1. Use various rectified activation functions for the binary classifiers and test them on various shapes (in analogy to the example with the circle above).
2. Convince yourself that starting backprop (with ReLU) with a too large initial learning speed leads to a "dead neuron" problem and a failure of the algorithm.

CHAPTER

NINE

UNSUPERVISED LEARNING

Motto

teachers! leave those kids alone!

(Pink Floyd, Another Brick In The Wall)

Supervised learning, discussed in previous lectures, needs a teacher or a training sample with labels, where we know **a priori** characteristics of the data (e.g., as in one of our examples, whether a given 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 preassigned 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 or labels floating around.

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, scent, etc. He goes on, finds a different flower, records its features, and so on and on with subsequent flowers. At some point, however, he finds a flower that he already had met. More precisely, its features are close, though not identical (the size may easily differ somewhat, so the color, etc.), to the previous instance. Hence he concludes that 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 of his quest, he has a catalog of flowers and now he can assign names (labels) to each species: corn poppy, bluebottle, mullein,... These labels, or names, are useful in sharing the knowledge with others, 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, the described problem of **unsupervised learning** is related to data classification (division into categories, or **clusters**, i.e. subsets of the 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

Here is our simplified version of the alien botanist exploration.

Consider points on the plane that are randomly generated. Their distribution is not homogeneous, but is concentrated in four clusters: A, B, C, and D. For example, we can set appropriate limits for the coordinates x_1 and x_2 when randomly generating points of a given category. We use the numpy **random.uniform(a,b)** function, giving a uniformly distributed number between a and b:

```
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)]
```

(continues on next page)

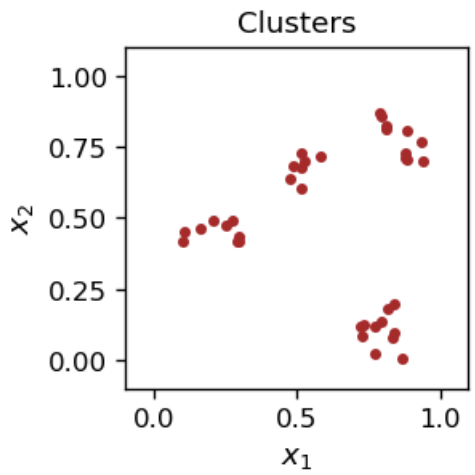
(continued from previous page)

```
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, This the person will undoubtedly state that there are four clusters. But what algorithm is being used 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** what the clusters are. 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 somewhere within the cluster. For example, one could take an item belonging to a given cluster as its representative, or for each cluster one can evaluate the mean position of its points and use it as a representative point:

```
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 have used the **statistics** module to evaluate the mean). We append 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 precisely this purpose).

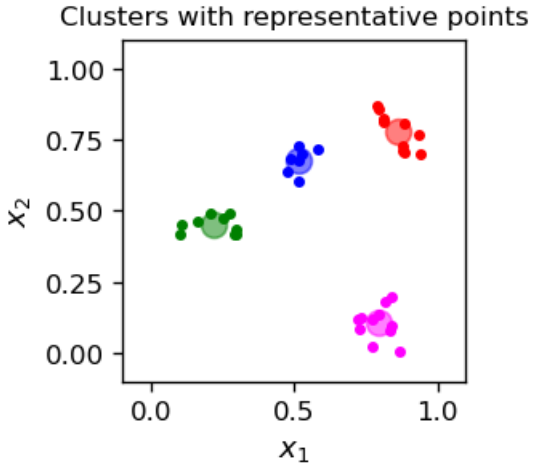
```
col=['red','blue','green','magenta']
```

```
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)
```

(continues on next page)

(continued from previous page)

```
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)
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)
plt.show()
```



9.2 Voronoi areas

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

Voronoi areas

Consider a metric space in which there are a number of representative points (the Voronoi points) R. For a given point P one determines the distances to all R. If there is a strict minimum among these distances (the closest point R_m), then by definition P belongs to the Voronoi area of R_m. If there is no strict minimum, then P belongs to a boundary between some Voronoi regions. The construction divides the whole space into Voronoi areas and their boundaries.

Returning to our example, let us then define the color of a point P in our square as the color of the nearest representative point. To do it, we first need (the square of) the 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, with **np.argmin**, we find the nearest representative point and determine its color:

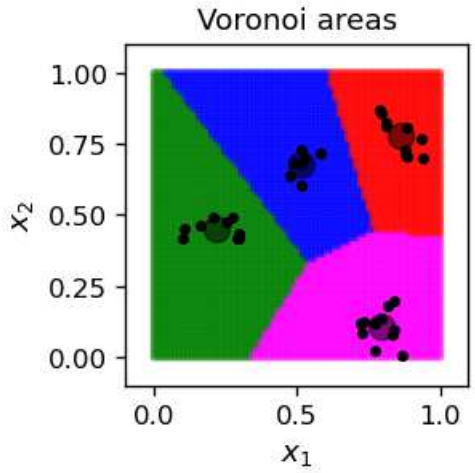
```
def col_char(p):
    dist=[eucl(p,rA),eucl(p,rB),eucl(p,rC),eucl(p,rD)] # array of distances
    ind_min = np.argmin(dist)                          # index of the nearest point
    return col[ind_min]                                # color of the nearest point
```

for instance

```
col_char([.5,.5])
```

```
'blue'
```

The result of running this coloring for points in our square (we take here a sufficiently dense sample of 70\tims 70$ points) is its following division into the Voronoi areas:



It is easy to prove that the boundaries between neighboring areas are straight lines.

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

9.3 Naive clusterization

Now we go back to the alien 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 with **np.concatenate**:

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

and shuffle them with **np.random.shuffle**:

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

The data visualization looks as in the first plot of this 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 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 the points of the sample, and then many such rounds 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()]) # initial location

print("initial location:")
print(np.round(R,3))
print("round   location")

eps=.5                          # initial learning speed

for j in range(50):             # rounds
    eps=0.85*eps                # decrease the learning 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))  # print every 5th step
```

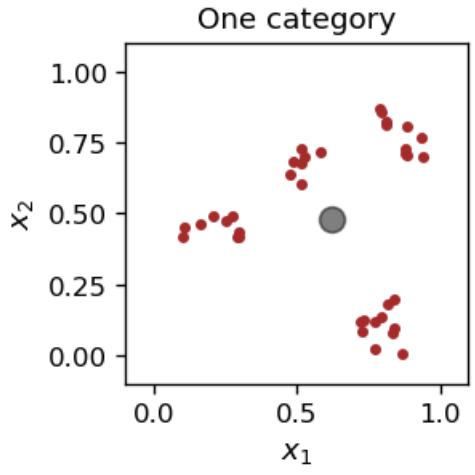
```
initial location:
[0.571 0.893]
round   location
5       [0.653 0.484]
10       [0.656 0.508]
15       [0.639 0.481]
20       [0.62  0.479]
25       [0.614 0.479]
30       [0.617 0.479]
35       [0.618 0.48 ]
40       [0.619 0.48 ]
45       [0.619 0.48 ]
50       [0.619 0.48 ]
```

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

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

```
[0.62 0.48]
```

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



One is, of course, not satisfied with the above (things are not classified with one category), so let us try to generalize the algorithm for the case of several ($n_R > 1$) representative points.

- We initialize randomly representative vectors $\vec{R}^i$, $i = 1, \ldots, n_R$.
- Round: We take the sample points P one by one and update only the **closest** representative 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 unchanged. This strategy is called **winner-take-all**.
- We repeat the rounds, reducing the learning speed ε each time, until we are happy with the result.

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

Let us then consider two representative points that we initialize randomly:

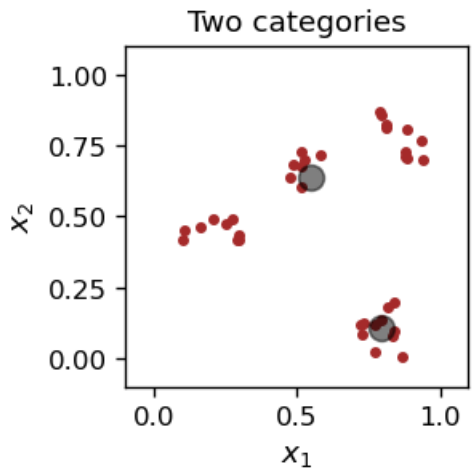
```
R1=np.array([np.random.random(), np.random.random()])
R2=np.array([np.random.random(), np.random.random()])
```

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

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

eps=.5

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

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

```
initial locations:
[0.641 0.449] [0.06  0.793]
rounds  locations
5       [0.798 0.103] [0.556 0.652]
10       [0.799 0.108] [0.486 0.612]
15       [0.792 0.106] [0.548 0.636]
20       [0.792 0.104] [0.549 0.64 ]
25       [0.793 0.105] [0.546 0.637]
30       [0.793 0.105] [0.546 0.638]
35       [0.793 0.105] [0.546 0.638]
40       [0.793 0.105] [0.546 0.638]
```

The result is this:



One of the characteristic points "specializes" in the lower right cluster, and the other in the remaining points.

Next, we continue, completely analogously, with four representative points.

The result for two different initial conditions of the characteristic points is shown in Fig. 9.1.

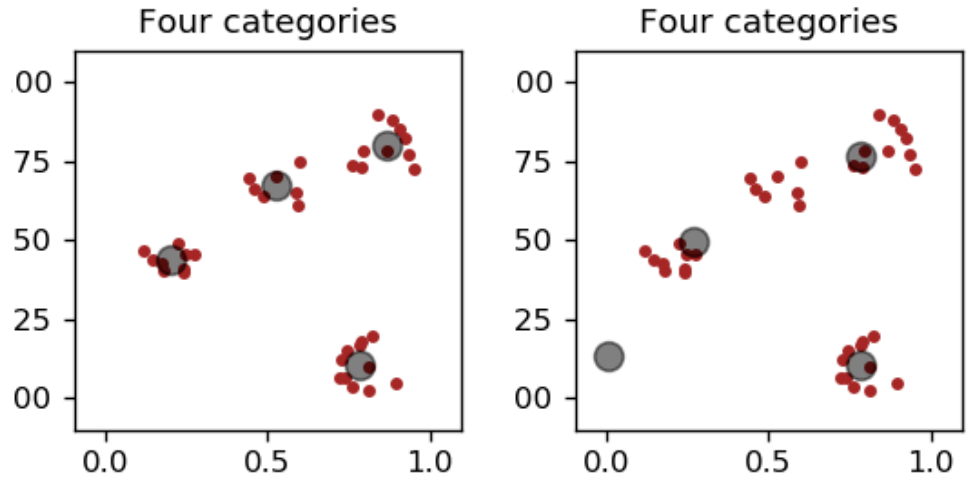


Fig. 9.1: Left: proper characteristic points. Right: one "dead body".

We notice that the procedure does not always give the "correct"/expected 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 data point of the sample than the "corpse". Certainly, this is an unsatisfactory situation.

When we set up five characteristic points, depending on the random initialization, several situation may occur, as shown in Fig. 9.2. Sometimes 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 strategy is highly problematic and we need something better.

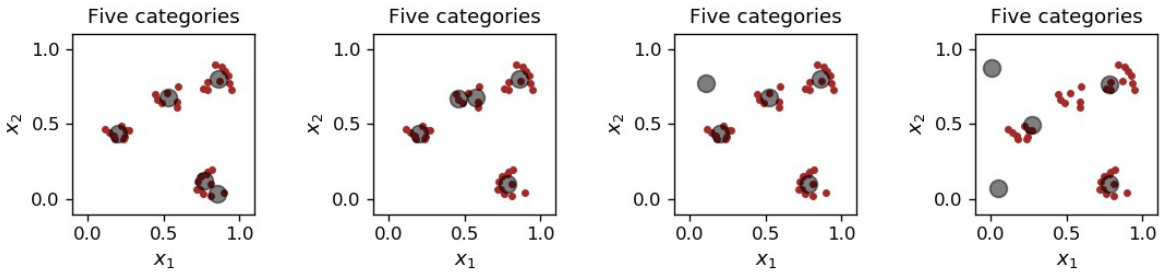


Fig. 9.2: From left to right: 5 characteristic points with one cluster split into two, with another cluster split into two, one dead body, and two dead bodies.

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 led to problems, as usually we do not even know how many clusters there are. Actually, up to now we have not defined what precisely a cluster is, and were 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 characteristic point, but how close? Actually, the definition must involve a **scale** (a characteristic distance) telling us "how close is close". For instance, in our example we may take a scale of about 0.2, where there are 4 clusters, but we may take a smaller scale and resolve the bigger clusters into smaller ones, as in two left panels of Fig. 9.2.

Definition of cluster

A cluster of scale d associated with a characteristic point R is a 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 encountered data point is farther than d from any characteristic point defined up to now.

Dynamical clusterization

1. Set the clustering scale d and the initial 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 of all characteristic points. 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 at the location of point P.
4. Repeat from 2. until all the data points are processed.
5. Repeat from 2. in a number of rounds, decreasing each time ε. The result is a division of the sample into a number of clusters, and the location of corresponding characteristic points. The result may depend on the random reshuffling, hence does not have to be 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
```

(continues on next page)

(continued from previous page)

```
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
        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:  5
```

The outcome of the algorithm 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 between 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, of the order of the span of the whole sample, there is only one cluster.

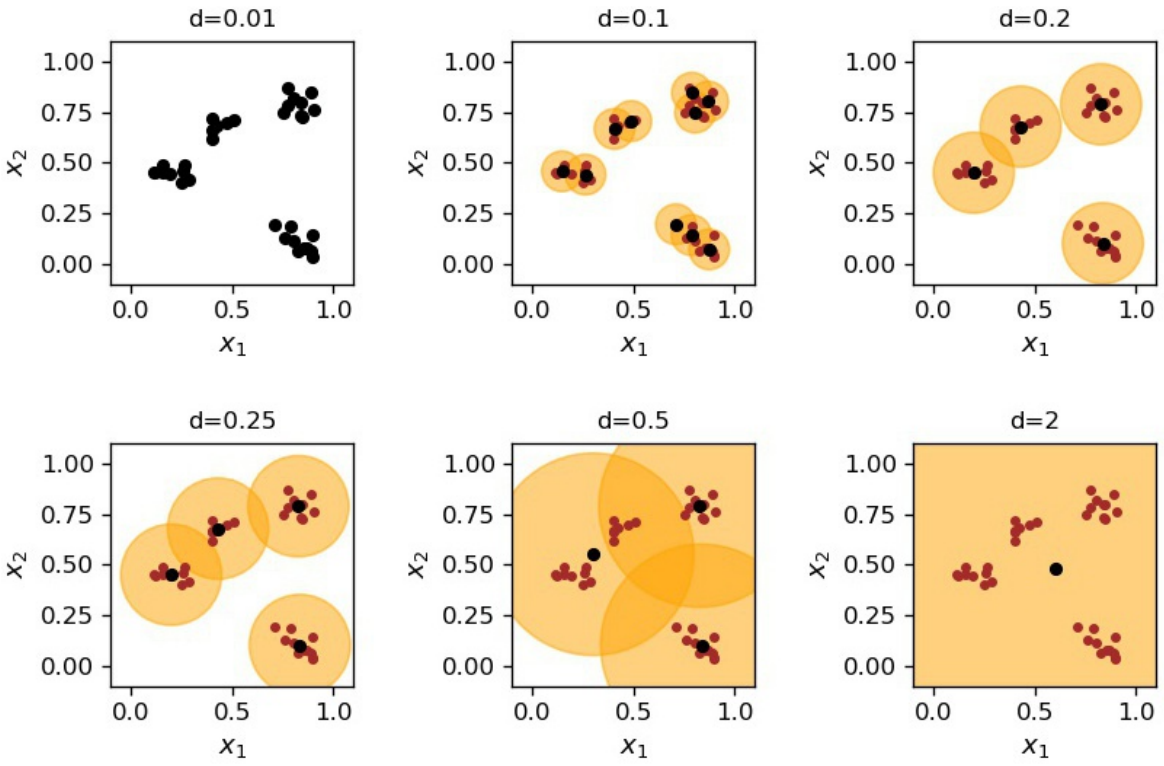


Fig. 9.3: Dynamical clustering for various values of the scale d.

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 a corresponding cluster label to the new 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 described above for new incoming data

In the latter case (we bought the ready and closed botanist's catalog), a data point may

- belong to a cluster (we know its label),
- fall outside of 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" or ambiguous classification.

Alternatively, we can use the Voronoi areas classification to get rid of the ambiguity.

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 clusterization algorithm presented above, i.e.

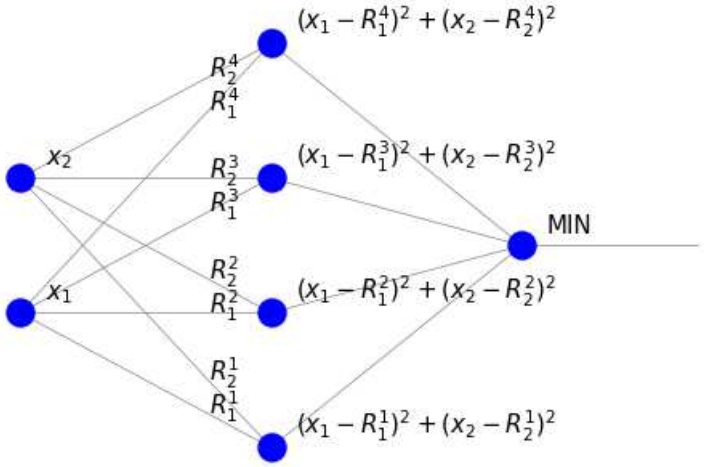
$$\vec{R} \to \vec{R} - \varepsilon(\vec{R} - \vec{x}^P).$$

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

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

9.5 Interpretation via neural networks

We shall now interpret the unsupervised learning algorithm used above with the winner-take-all strategy in the neural network language. We have the following sample network:



It consists of 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 corresponding 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 P "attracts" the closest characteristic point, whose weights are updated towards the coordinates of P.

The application of the above network classifies the point with coordinates (x_1, x_2), assigning it the index of the closest 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 liberty", calling the above system a neural network would be 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/trick to make it more similar to the perceptron principle. For this purpose we introduce 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 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 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

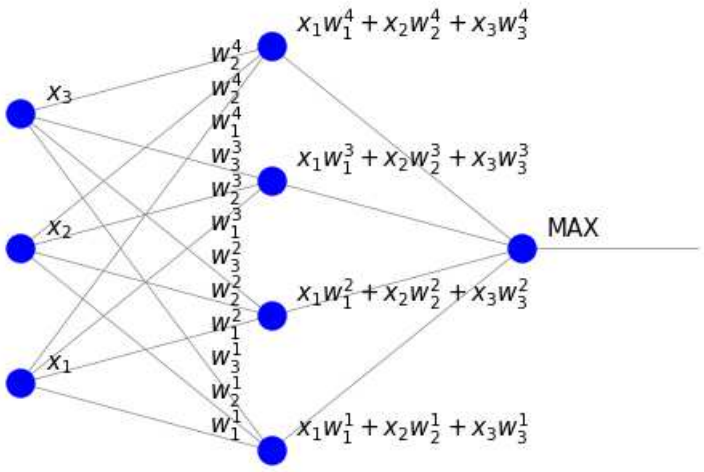
$$\sqrt{r^2 - a^2} \simeq r - \frac{a^2}{2r}, \quad a \ll r,$$

hence

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

It equals (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 3-dim. extensions to a hemisphere.

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



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 an 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 type 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. We discuss these aspects in section {ref}`lat-lab'.

Hebbian rule

On the conceptual side, we touch upon a very important and intuitive principle in biological neural networks, known as the Hebbian rule. Essentially, it applies the truth "What is being used, gets stronger" to 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, while other connections remain the same. The process takes place in an unsupervised manner and its implementation is biologically well motivated.

Note: On the other hand, it is difficult to find a biological justification for the backprop 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

Now the algorithm becomes as 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, such 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 exactly as in the previously discussed procedure.

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

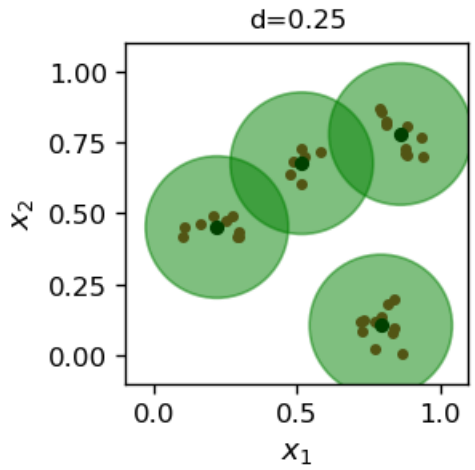
$$x_{n+1} = \sqrt{r^2 - x_1^2 - ... - x_n^2},$$

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

In Python:

```
d=0.25
eps=.5

rad=2    # radius of the hypersphere

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)])])
                                    # extension of R to the hypersphere
    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)])
                                    # extension of p to the hypersphere
        dist=[np.dot(p,R[k]) for k in range(len(R))]      # array of dot products
        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 promptly see that the dot product maximization algorithm yields an almost exactly the same result as the distance squared minimization (cf. Fig. 9.3.

9.6 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 chapter using this metric. Draw conclusions.
2. Run the classification algorithms for more categories in the data sample (generate your own sample).
3. Extend the dynamic clusterization algorithm to a three-dimensional input space.

CHAPTER

TEN

SELF ORGANIZING MAPS

A very important and ingenious application of unsupervised learning are the so-called **Kohonen networks** (Teuvo Kohonen, a class of **self-organizing mappings (SOM)**. Consider firs a mapping f between a **discrete** k-dimensional set (we call it a **grid** in this chapter) of neurons and n-dimensional input data D (continuous or discrete),

$$f : N \to D$$

(note that **this is not a Kohonen mapping yet!**). Since N is discrete, each neuron carries an index consisting of k natural numbers, denoted as $\bar{i} = (i_1, i_2, ..., i_k)$. Typically, the dimensions in Kohonen's networks satisfy $n \geq k$. When $n > k$, one talks about **reduction of dimensionality**, as then the input space D has more dimensions than the dimensionaiy of the grid of neurons N.

Two examples of such networks are visualized in Fig. 10.1. The left panel shows a 2-dim. input space D, and a one dimensional grid on neurons labeled with i. The input point (x_1, x_2) enters all the neurons in the grid, and one of the neurons (the one with best-suited weights) becomes the **winner** (red dot). The gray oval indicates the **neighborhood** of the winner, to be defined accurately in the following.

The right panel shows an analogous situation for the case of a 3-dim. input and 2-dim. grid of neurons, now labeled with a double index $\bar{i} = (i_1, i_2)$. Here, for clarity, we only indicate the edges entering the winner, but they also enter all the other neurons in the grid, similarly to the left panel.

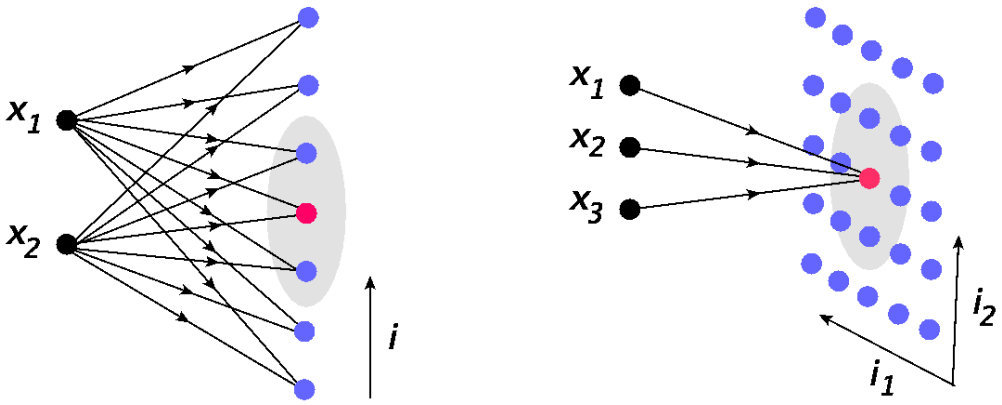


Fig. 10.1: Example of Kohonen's networks. Left: 1-dim. grid of neurons N and 2-dim. input space D. Right: 2-dim. grid of neurons N and 3-dim. input space D. The red dot indicates the winner, and the gray oval marks its neighborhood.

Next, 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 we have $\phi(i, j) = \exp\left[-\frac{(i-j)^2}{2\delta^2}\right]$.

10.1 Kohonen's algorithm

The set up for Kohonen's algorithm is similar to the unsupervised learning discussed in the previous chapter. Each neuron $\bar{i}$ obtains weights $f(\bar{i})$, which are elements of D, i.e. form n-dimensional vectors. One may simply think of this procedure as placing the neurons in some locations in D.

When an input point P from D is fed into the network, one looks for the closest neuron, which becomes the **winner**, exactly as in the unsupervised learning algorithm from section *Interpretation via neural networks*. However, now comes a **crucial 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, as quantified by the proximity function.

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, as given by the proximity function.

Kohnen's algorithm

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

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

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

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

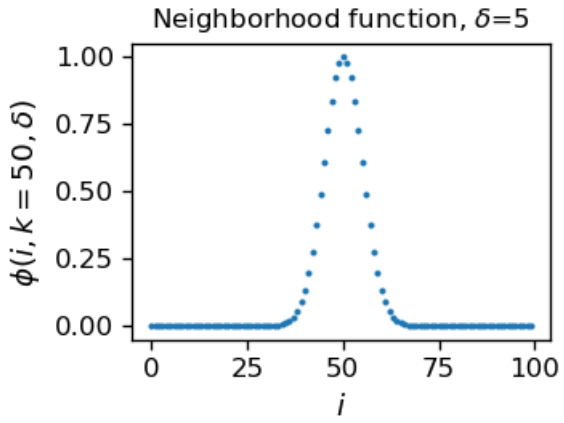
10.1.1 2-dim. data and 1-dim. neuron grid

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

and the Gaussian proximity function

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

This function looks as follows around the middle neuron ($k = 50$) and for the width parameter $\delta = 5$:



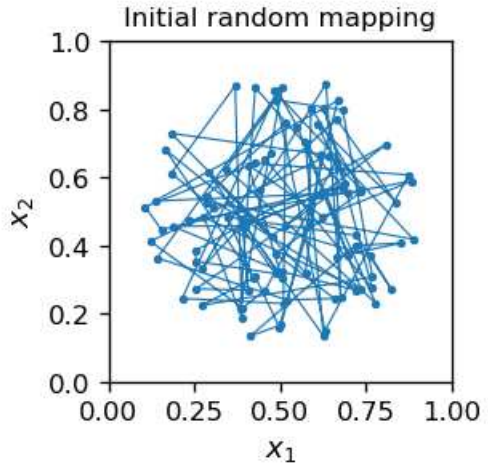
As a feature of a Gaussian, at $|k-i| = \delta$ the function drops to 60% of the central value, and at $|k-i| = 3\delta$ to 1%, a tiny fraction. Hence δ controls the size of the neighborhood of the winner. The neurons farther away from the winner than, say, 3δ are practically left uncharged.

We initiate the network by by placing the grid inside the circle, with a random location of each neuron. As said, this amounts to assigning weights to the neuron equal to its location. An auxiliary line is drawn to guide the eye sequentially along the neuron indices: $1, 2, 3, ... m$. The line has no other meaning.

The weights (neuron locations) are stored in array **W**:

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

As a result of the initial randomness, the neurons are, of course, "chaotically" distributed:



Next, we initialize the parameters **eps** amd **delta** and run the algorithm. Its structure is analogous to the previously discussed codes and is a straightforward implementation of the steps spelled out in the previous section. For that reason, we only provide the comments in the code.

```
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                   # count steps
```

(continues on next page)

(continued from previous page)

```
        dist=[func.eucl(p,W[k]) for k in range(num)]
         # array of squares of Euclidean disances between p and the neuron␣
↪locations
        ind_min = np.argmin(dist) # index of the winner
        for k in range(num):      # loop over 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 above algorithm progresses (see Fig. 10.2) the neuron grid first disentangles, 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 P attracts towards itself the nearest neuron (the winner), but also, to a weaker extent, its neighbors. At the beginning of the algorithm the neighborhood distance **de** is large, so large chunks of the neighboring neurons in the input grid are pulled together towards P, and the arrangement looks as in the top right corner of Fig. 10.2. At later stages **de** reduces, so only the winner and possibly its very immediate neighbors are attracted to a new point. After completion (bottom right panel), 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.

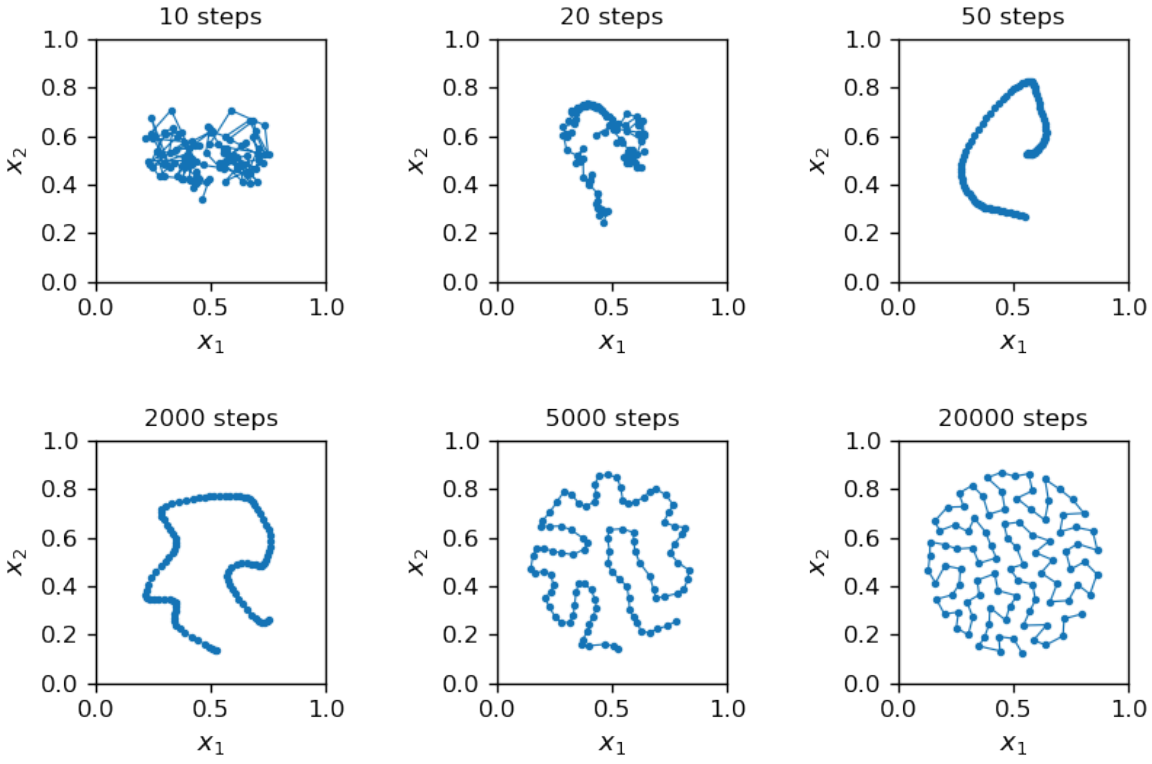


Fig. 10.2: Progress of Kohonen's algorithm. The line, drawn to guide the eye, connects neurons with adjacent indices.

Kohonen's network as a classifier

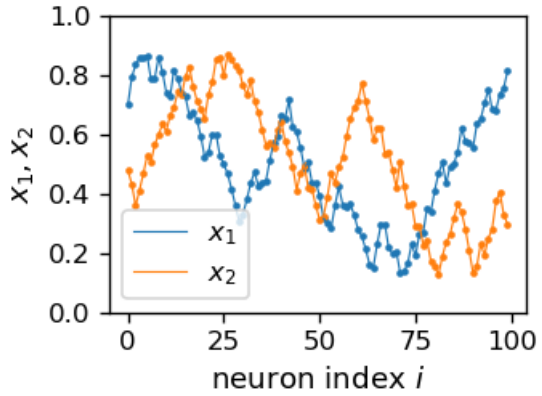
Having the trained network, we may use it as a classifier similarly as in chapter *Unsupervised learning*. We label a point from D with the index of the nearest neuron. One can interpret this as a Voronoi construction, see section *Voronoi areas*.

The plots in Fig. 10.2 are made in coordinates (x_1, x_2), that is, from the "point of view" of the input 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.

Caution

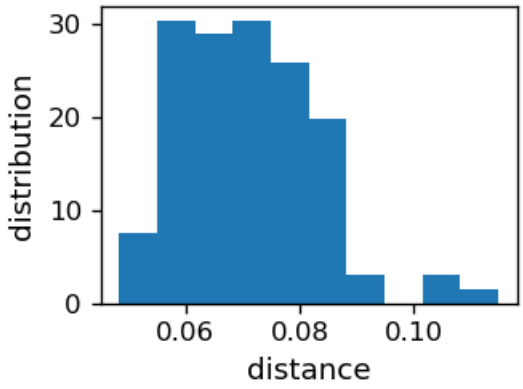
When presenting results of Kohonen's algorithm, one sometimes makes plots in D-space, and sometimes in N-space, which may lead to some confusion.

The plots in the N-space, fully equivalent in information to the plot in, e.g., the bottom right panel of Fig. 10.2, are following:



We note that the jumps in the above plotted curves are small, since the subsequent neurons are close to each other. This feature can be presented quantitatively as in the histogram below, where we can see that the average distance between the neurons is about 0.07, and the spread is between 0.05 and 0.10.

```
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
plt.show()
```



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 encounter 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 can be non-uniformly distributed, the neurons would fill the area containing more data more densely.
- The policy of choosing initial δ and ε parameters and reducing them appropriately in subsequent rounds is based on experience and is non-trivial. The results depend significantly on this choice.

- The final result, even with the same δ and ε strategy, is not unequivocal, i.e. running the algorithm with a different initialization of the weights (initial positions of neurons) yields different outcomes, usually equally "good".
- Finally, the progress and the result of the algorithm is reminiscent of the construction of the Peano curve in mathematics, which fills densely an area with a line. As we increase the number of neurons, the analogy gets closer and closer.

10.1.2 2 dim. color map

Now we pass to a case of 3-dim. data and 2-dim. neuron grid, which is a situation from the right panel of Fig. 10.1 (hence also with dimensionality reduction). As we know, an RGB color is described with three numbers $[r, g, b]$ 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
```

The proximity function is now a Gaussian in two dimensions:

```
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
```

We also decide to normalize the RGB colors such that $r^2 + g^2 + b^2 = 1$. This makes the perceived intensity of colors similar (this normalization could be dropped, as irrelevant for the method to work).

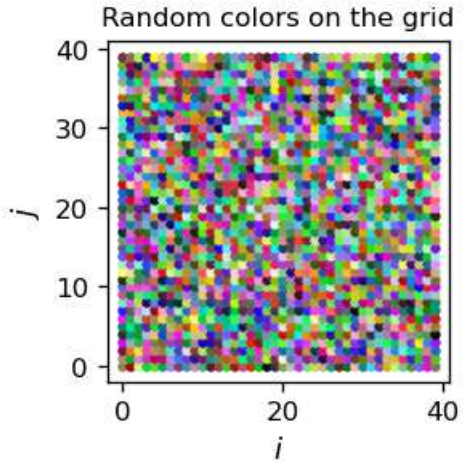
```
def rgbn():
    r,g,b=np.random.random(),np.random.random(),np.random.random() # random RGB
    norm=np.sqrt(r*r+g*g+b*b)                                      # norm
    return np.array([r,g,b]/norm)                                  # normalized RGB
```

Next, we generate and plot a sample of **ns** points with (normalized) 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)

plt.show()
```


We use a 2-dim. **size** x **size** grid of neurons. Each neuron's position (that is its color) in the 3-dim. D-space is initialized randomly:

```
size=40                          # neuron array of size x size (40 x 40)
tab=np.zeros((size,size,3))      # create array tab with zeros

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: k=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 a bit faster than eps
    for s in range(ns): # loop over the points in the data sample
        p=samp[s]        # point from the sample
        dist=[[dist3(p,tab[i][j]) for j in range(size)] for i in range(size)]
                         # distance of p from all neurons
        ind_min = np.argmin(dist) # the winner index
        ind_1=ind_min//size       # a trick to get a 2-dim index
        ind_2=ind_min%size

        for j in range(size):
            for i in range(size):
                tab[i][j]+=eps*phi2(ind_1,ind_2,i,j,de)*(p-tab[i][j]) # update
 ↪
```

A word of explanation is in place here, concerning the numpy **argmin** function. For a 2-dim. array it provides the index of the minimum in the corresponding **flattened** array (cf. section *Heteroassociative memory*). Hence, to get the indices in the two dimensions, we need to apply the operations **//** (integer division) and **%** (remainder). For instance, in an array **ind_min=53**, then **ind_1=ind_min//size=53//10=5** and **ind_2=ind_min%size=53//10=3**.

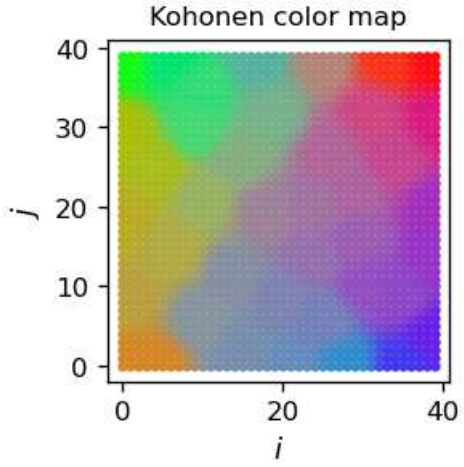
As a result of the above code, we get an arrangement of our color sample in two dimensions in such a way that the neighboring areas in the grid have a similar color "specializing" on the color of a given sample point (note the plot is in the N-space):

```
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):
```

(continues on next page)

(continued from previous page)

```
        plt.scatter(i,j,color=tab[i][j], s=8)

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



Remarks

- The areas for the individual colors of the sample have a comparable area. Generally, the area is proportional to the frequency of the data point in the sample.
- To get sharper boundaries between the regions, **de** would have to shrink even faster compared to **eps**. Then, in the final stage of learning, the neuron update process takes place within a smaller neighborhood radius and more resolution in the boundaries can be achieved.

10.2 U-matrix

A convenient way to present the results of Kohonen's 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 in N-space with the intensity given by the averaged distance (in D-space) of the given neuron to its immediate neighbors, and not a neuron property itself (such as its color in the figure above). This is particularly useful when the dimension of the input space is large, hence it is difficult to visualize the results directly.

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

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

The Python implementation of the above definition is following:

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

for i in range(1,size-1):         # loops over the neurons in the grid
    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
```

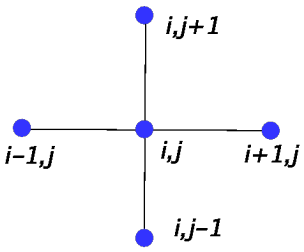
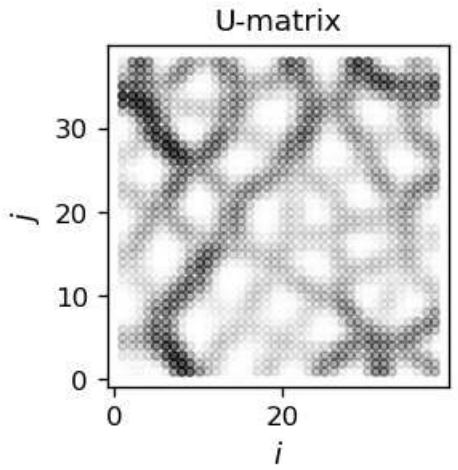


Fig. 10.3: Construction of U_{ij}: a geometric average of the distances along the indicated links.

The result, corresponding one-to-one to the color map above, can be presented in a contour plot:

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

for i in range(size-2): # loops over indices, excluding the boundaries 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)
plt.show()
```



The white regions in the above figure show the clusters (they correspond one-to-one to the regions of the same color in the previously shown color map). There, the elements $U_{ij} \simeq 0$. The clusters are separated with darker boundaries. The higher the dividing ridge between clusters, the darker the intensity.

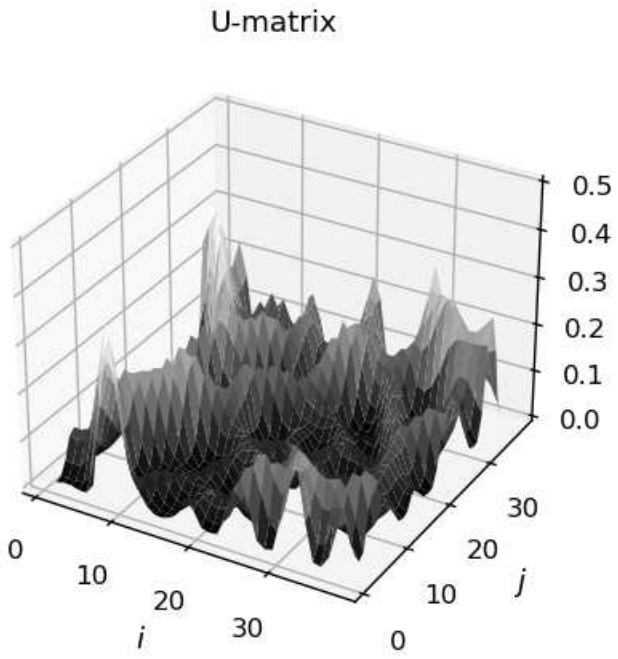
The result may also be visualized with a 3-dim. plot:

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

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)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('$i$',fontsize=11)
plt.ylabel('$j$',fontsize=11)
plt.title("U-matrix",fontsize=11)
plt.show()
```



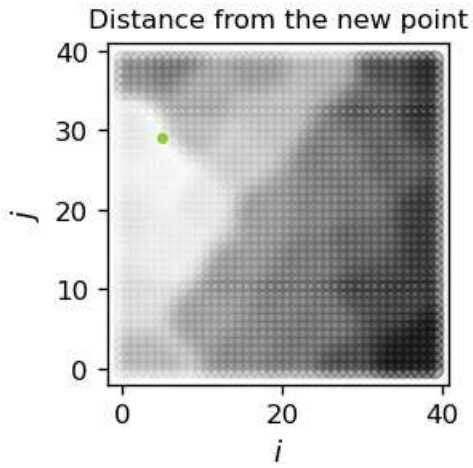
We can now classify a given (new) data point according to the obtained map. We generate a new (normalized) RGB color:

```
nd=rgbn()
```

It is useful to obtain a map of distances of our grid neurons 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])

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

```
Closest neuron grid indices: ( 5 , 29 )
Distance:  0.042
```



The lightest region in the above figure indicates the cluster, to which the new point belongs. The darker the region, the larger is the distance from the corresponding neuron.

One should stress that we have obtained a classifier which not only assigns a closest cluster to a probed point, but also provides its distances from all other clusters.

10.2.1 Mapping colors on a line

In this subsection we present an example of a mapping of 3-dim. data into a 1-dim. neuron grid, hence a reduction of three dimensions into one. This proceeds exactly along the lines of the previous subsection, so we are very brief in comments.

```
ns=8
samp=[rgbn() 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)

plt.show()
```

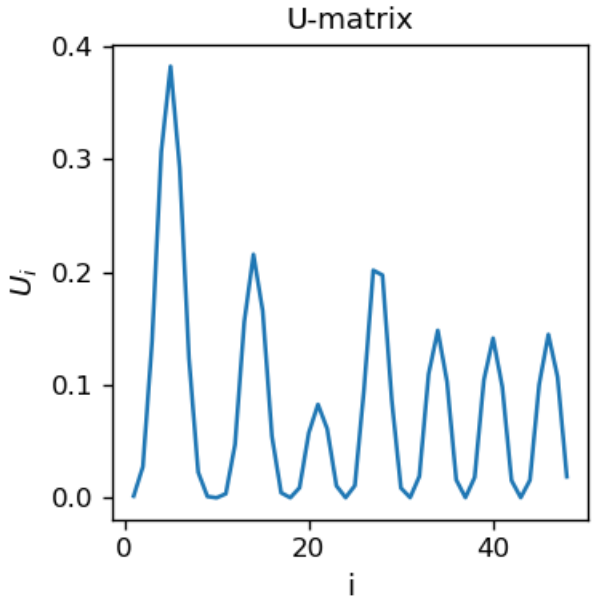

```
si=50                    # 1-dim. grid of si neurons, 3 RGB components
tab2=np.zeros((si,3))    # neuron gri

for i in range(si):
    tab2[i]=rgbn()     # 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])
```

As expected, we note smooth transitions between colors. The formation of clusters can be seen with the U-matrix, which now is, of course, one-dimensional:

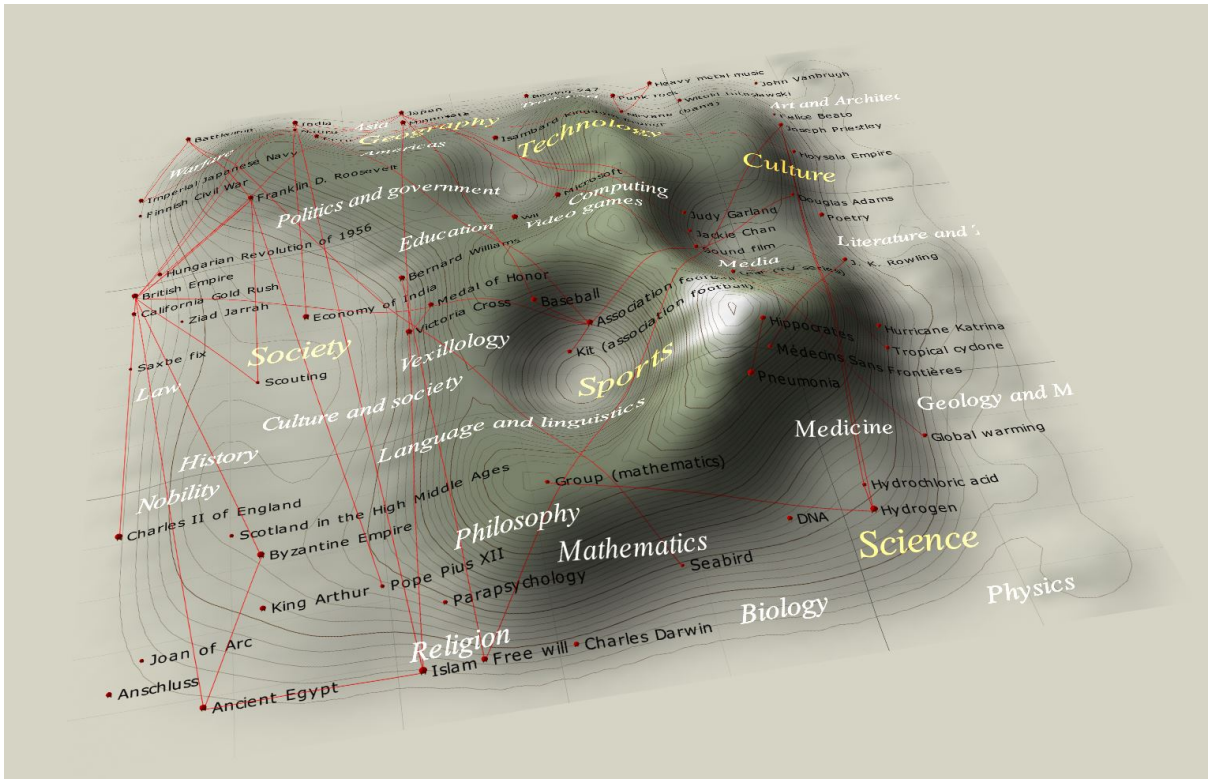
```
ta2=np.zeros(si-2)

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, equal to the multiplicity of the sample) indicate the clusters. The height of the separating peaks shows how much the neighboring colors differ. Again, we see a nicely produced classifier, this time with two dimensions "hidden away", as we reduce from three to one.

10.2.2 Large reduction of dimensionality

In many situations the input space may have a very large dimension. In the Wikipedia example quoted here, one takes articles from various fields and computes frequencies of used words (for instance, in a given article how many times the word "goalkeeper" has been used, divided by the total number of words in the article). Essentially, the dimensionality of D is of the order of the number of all English words, a huge number $\sim 10^5$! Then, with a properly defined distance depending on these frequencies, one uses Kohonen's algorithm to carry out a reduction into a 2-dim. grid of neurons. The resulting U-matrix can be drawn as follows:



Not surprisingly, we notice that articles on sports are special and form a very well defined cluster. The reason is that the sport's jargon is very specific. The Media are also distinguished, whereas other fields are more-less uniformly distributed. The example shows how we can, with a very simple method, comprehend data in a multidimensional space and see specific correlations/clusters. Whereas some conclusions may be obvious, such as the fact that sport has a unique jargon, other are less transparent, for instance the emergence of the media cluster and lack of well-defined clusters for other fields, e.g. for mathematics.

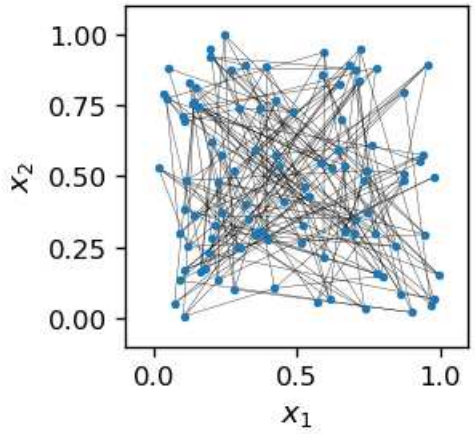
10.3 Mapping 2-dim. data into a 2-dim. grid

Finally, we come to a very important case of mapping 2-dim. data in a 2-dim. grid, i.e. with no dimensionality reduction. In particular, this case is realized in our vision system between the retina and the visual cortex.

The algorithm proceeds analogously to the previous cases. We initialize an $n \times n$ grid of neurons and place them randomly in the square $[0, 1] \times [0, 1]$.

```
n=10
sam=np.array([func.point() for _ in range(n*n)])
```

The lines, again drawn to guide the eye, join the adjacent index pairs in the grid: [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 only 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
```

```
# completely analogous to the previous codes of this chapter
for _ in range(nr):    # rounds
    eps=eps*.97
    de=de*.98
    for _ in range(rep):     # repeat for rep points
        ste=ste+1
        p=func.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])
```

Here is the history of a simulation:

As the algorithm progresses, the initial "chaos" gradually changes into a nearly perfect order, with the grid placed uniformly in the square of the data, with only slight displacements from a regular arrangement. On the way, near 40 steps, we notice a phenomenon called "twist", where the grid is crumpled. In the twist region, many neurons, also of distant indices, have a close location in (x_1, x_2).

10.4 Topology

Recall the Voronoi construction of categories introduced in section *Voronoi areas*. One can use it now again, 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).

The procedure is as follows: We take the final outcome of the algorith, such as in the bottom right panel of Fig. 10.4, construct the Voronoi areas for all the neurons, and thus obtain a mapping v for all the points in the (x_1, x_2) square. The reader may notice that there is an ambiguity for points lying exactly at the boundaries between the neighboring

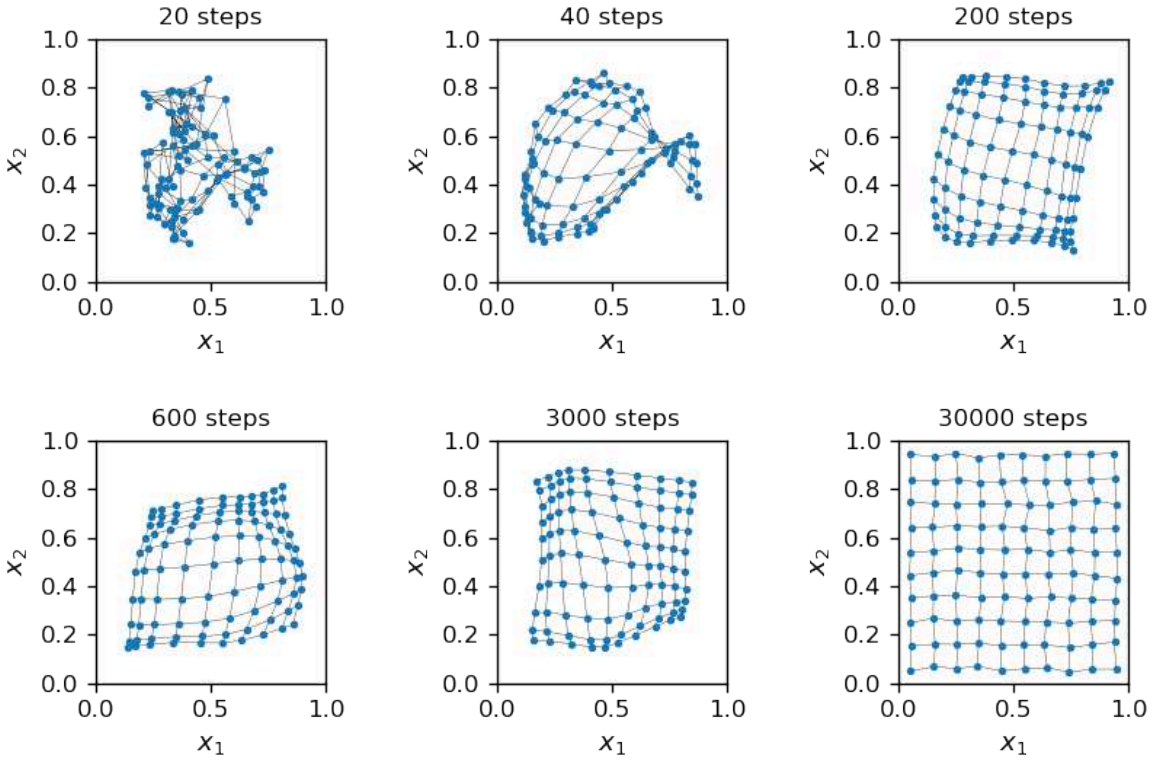


Fig. 10.4: Progress of Kohonen's algorithm. The lines, drawn to guide the eye, connects neurons with adjacent indices.

areas, but this can be taken care of by using an additional prescription (for instance, selecting a neuron lying at a direction which has the lowest azimuthal angle, etc.)

Now a key observation:

Topological property

For situations without twists, 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 straightforward to prove: Since d_1 and d_2 are close (and we mean very close, closer than the grid spacing), they must belong either to

- the same Voronoi area, where $v(d_1) = v(d_2)$, or
- a pair of neighboring Voronoi areas.

Since for the considered situation (without twists) the neighboring areas have the grid indices differing by 1, the conclusion that $v(d_1)$ and $v(d_2)$ are close follows immediately.

Note that this feature of Kohonen'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 advertised topological property no longer holds.

The discussed topological property has mathematically general and far-reaching consequences. First, it allows to carry over "shapes" from D to N. We illustrate it on an example.

Imagine that we have a circle C in D-space, of radius **rad** centered at **cent**. We need to find the winners in the N space for any point in C. For this purpose we go around C in **npoi** points equally spaced in the azimuthal angle, and for each one find a winner.

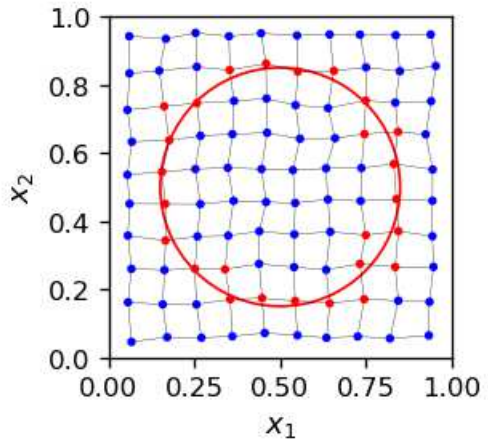
C is parametrized with polar coordinates:

$$x_1 = r\cos\left(\frac{2\pi\phi}{N}\right) + c_1, \quad x_2 = r\sin\left(\frac{2\pi\phi}{N}\right) + c_2.$$

Going to the mathematical notation to Python we use r =**rad**, ϕ=**ph**, N =**npoi**, (c_1, c_2) =**[cent]**. The loop over **ph** goes around the circle.

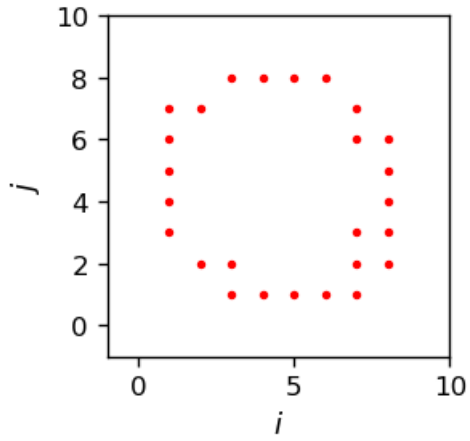
```
rad=0.35                        # radius of a circle
cent=np.array([0.5,0.5])        # center of the circle
npoi=400                        # number of points in the circle

wins=[]                         # table of winners

for ph in range(npoi):          # go around the circle
    p=np.array([rad*np.cos(2*np.pi/npoi*ph),rad*np.sin(2*np.pi/npoi*ph)])+cent
                                # the circle in polar coordinates
    dist=[func.eucl(p,sam[l]) for l in range(n*n)]
      # distances from the point on the circle to the neurons in the nxn grid
    ind_min = np.argmin(dist) # winner
    wins.append(ind_min)        # add winner to the table

ci=np.unique(wins)              # remove duplicates from the table
```

The result of Kohonen's algorithm is as follows:



The red neurons are the winners for certain sections of the circle. When we draw these winners alone in the N space (keep in mind we are going from D to N), 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)
plt.show()
```



This looks pretty much as a (rough and discrete) circle. Note that in our example we only have $n^2 = 100$ pixels to our disposal - a very low resolution. The image would look better and better with an increasing n. At some point one would reach the 10M pixel resolution of typical camera, and then the image would seem smooth! We have carried over our circle from D into N.

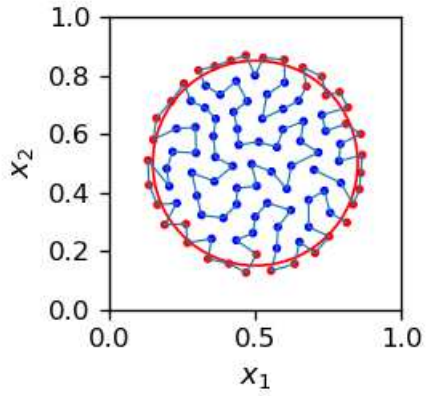
Vision

The topological property, such as the one in the discussed Kohonen mappings, has a prime importance in our vision system and the perception of objects. Shapes are carried over from the retina to the visual cortex and are not "warped up" on the way!

Another key topological feature is the preservation of **connectedness**. If an area A in D is connected (so to speak, 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 "torn into pieces" when transforming from D to N.

Note that the discussed topological features need not be present when the dimensionality is reduced, as in our previous examples. Take for instance the bottom right panel of Fig. 10.2. There, many neighboring pairs of the Voronoi areas correspond to distant indices, so it is no longer true that $v(d_1)$ and $v(d_2)$ in N are close for close d_1 and d_2 in D, as these points may belong to different Voronoi areas with **distant** indices.

For that case, our example with the circle looks like this:



When we go subsequently along the **grid indices** (i.e. along the blue connecting line), taking $i = 1, 2, \ldots, 100$, we obtain the plot below. We can see the image of our circle (red dots) as a bunch of **disconnected** red sections. The circle is torn into pieces, the **topology is not preserved!**

Here is the summarizing statement (NB not made sufficiently clear in the literature):

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 cases, $k < n$.

10.5 Lateral inhibition

In the last topic of these lectures, we return to the issue of how the competition for the "winner" is realized in ANNs. Up to now (cf. section *Interpretation via neural networks*), we have just been using the minimum (or maximum, when the signal was extended to a hyperphere) in the output, though this is embarrassingly outside of the neural framework. Such an inspection 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 within the rules of the game.

Actually, if the neurons in a layer "talk" to one another, we can have a "contest" from which a winner may emerge. In particular, an architecture as in Fig. 10.5 allows for an arrangement of competition and a natural realization of a **winner-take-most** mechanism.

The type of models as presented below is known as the Hopfield networks. Note that we depart here from the **feed-forward** limitation of Fig. 1.3 and allow for a recursive, or feed-back character.

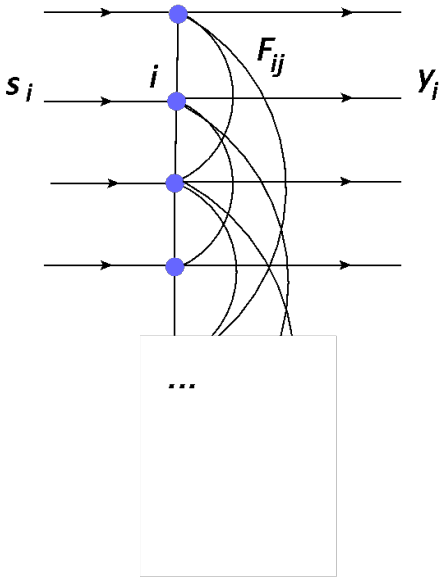


Fig. 10.5: Network with inter-neuron couplings used for modeling lateral inhibition. All the neurons are connected to one another in both directions (lines without arrows).

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 of neuron i. The neuron produces output y_i, where now a part of it is sent to neurons j as $F_{ji}y_i$. Here F_{ij} denotes the coupling strength (we assume $F_{ii} = 0$ - no self coupling). Reciprocally, neuron i also receives output from neurons j in the form $F_{ij}y_j$. The summation 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 formally

$$y = (I - F)^{-1} s. \tag{10.1}$$

One needs to model appropriately the coupling matrix F. We take

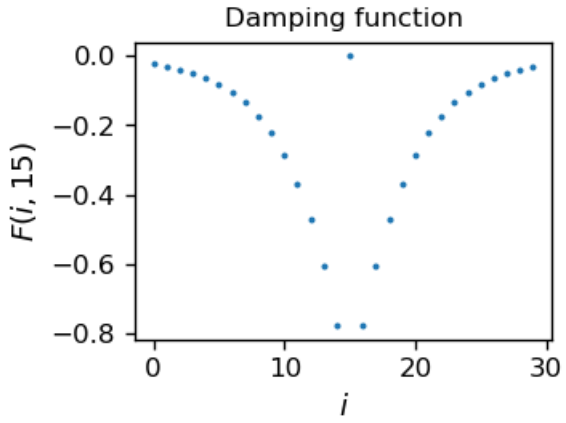
$F_{ii} = 0,$

$F_{ij} = -a \exp(-|i-j|/b) \quad \text{for } i \neq j, \quad a, b > 0,$

i.e. assume attenuation (negative feedback), which is strongest for close neighbors and decreases with distance. The decrease is controlled by a characteristic scale b.

The Python implementation is straightforward:

```
ns = 30;        # number of neurons
b = 4;          # parameter controlling 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-coupling
```



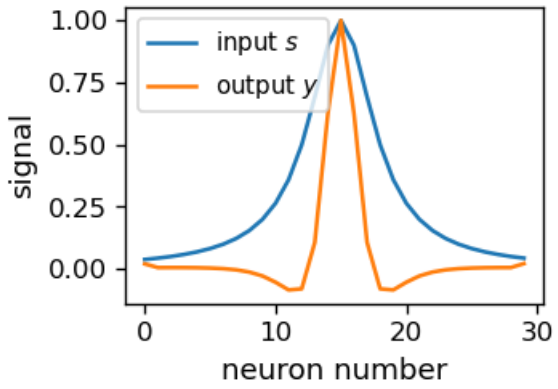
We assume a bell-shaped Lorentzian input signal s, with a maximum in the middle neuron. The width is controlled with **D**:

```
D=3
s = np.array([D**2/((i - ns/2)**2 + D**2) for i in range(ns)]) # Lorentzian␣
 ↪function
```

Next, we solve Eq. (10.1) via inverting the $(I - F)$ matrix, performed with the numpy **linalg.inv** function. Recall that **dot** multiplies matrices:

```
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 actually quite remarkable: the output signal y becomes much narrower from the input signal s. This may be interpreted as a realization of the "winner-take-all" scenario. The winner "damped" he guys around him, so he puts himself on airs! The effect is smooth, with the signal visibly sharpened.



Lateral inhibition

The damping of the response of neighboring neurons is called **lateral inhibition**. It was discovered in neurobiological networks [HR72].

The presented model is certainly too simplistic to be realistic from the point of view of biological networks. Also, it yields unnatural negative signal outside of the central peak (which we can remove with rectification). Nevertheless, the setup shows a possible way to achieve the "winner competition", essential for unsupervised learning: One needs to allow for the competing neurons to interact.

Note: Actually, **pyramidal neurons**, present i.a. in the neocortex, have as many as a few thousand dendritic spines and do realize a scenario with numerous synaptic connections. They are believed Quantamagazine to play a crucial role in learning and cognition processes.

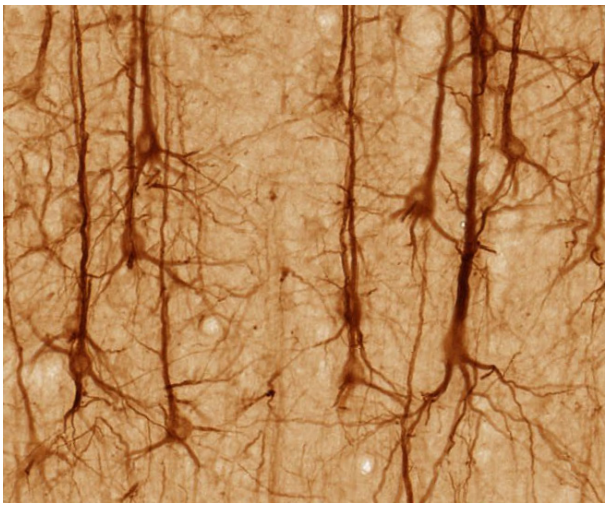


Fig. 10.6: Image of pyramidal neurons (from brainmaps.org)

10.6 Exercises

1. Construct a Kohonen mapping form a **disjoint** 2D shape into a 2D grid of neurons.
2. Construct a Kohonen mapping for a case where the points in the input space are not distributed uniformly, but denser in some regions.
3. Create, for a number of countries, fictitious flags which have two colors (hence are described with 6 RGB numbers). Construct a Kohonen map into a 2-dim. grid. Plot the resulting U-matrix and draw conclusions.
4. Lateral inhibition has "side-effects" seen in optical delusions. Describe the Mach illusion, programming it in Python.

CHAPTER

ELEVEN

CONCLUDING REMARKS

In a programmer's life, building a well-functioning ANN, even for simple problems as used for illustrations in these lectures, can be a truly frustrating experience! There are many subtleties involved on the way. To list just a few that we have encountered in this course, one faces a choice of the network architecture and freedom in the initialization of weights (hyperparameters). Then, one has to select an initial learning speed, a neighborhood distance, in general, some parameters controlling the performance/convergence, as well as their update strategy as the algorithm progresses. Further, one frequently tackles with an emergence of a massive number of local minima in the space of hyperparameters, and many optimization methods may be applied here, way more sophisticated than our simplest steepest-descent method. Moreover, a proper choice of the neuron activation functions is crucial for success, which relates to avoiding the problem of "dead neurons". And so on and on, many choices to be made before we start gazing in the screen in hopes that the results of our code converge …

Important: Taking the right decisions for the above issues is an **art** more than science, based on long experience of multitudes of code developers and piles of empty pizza boxes!

Now, having completed this course and understood the basic principles behind the simplest ANNs inside out, the reader may safely jump to using professional tools of modern machine learning, with the conviction that inside the black boxes there sit essentially the same little codes he met here, but with all the knowledge, experience, tricks, provisions, and options built in. Achieving this conviction, through appreciation of simplicity, has been one of the guiding goals of this course.

11.1 Acknowledgments

The author thanks Jan Broniowski for priceless technical help and for remarks to the text.

CHAPTER

TWELVE

APPENDIX

12.1 How to run the book codes

12.1.1 Locally

The reader may download the Jupyter notebooks for each chapter by clicking the download icon (downward arrow) on the right in the top bar when viewing the book. Alternatively, the complete set of files may be downloaded from www.ifj.edu.pl/~broniows/nn or www.ujk.edu.pl/~broniows/nn.

In the latter case, the file **nn-book.zip** should be unpacked in a chosen working directory. The proper directory tree structure of the library package **lib-nn** and of the graphics files **images** will be reproduced.

```
your_directory
└── nn_book
    ├── ...
    ├── backprop.ipynb
    ├── ...
    ├── lib-nn
    └── images
```

Having installed Python and Jupyter (preferably via conda), the reader can follow the instructions to open Jupyter (specific for the operating system) and execute one-by-one the lecture's notebooks stored in directory **nn-book**.

12.2 neural package

The structure of the library package tree is as follows:

```
lib_nn
└── neural
    ├── __init__.py
    ├── draw.py
    └── func.py
```

and consists of two modules: **func.py** and **draw.py**.

12.2.1 func.py module

```
"""
Contains functions used in the lecture
"""

import numpy as np


def step(s):
    """
    step function

    s: signal

    return: 1 if s>0, 0 otherwise
    """
    if s>0:
        return 1
    else:
        return 0


def neuron(x,w,f=step):
    """
    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=f(w0 + x1 w1 + x2 w2 +...+ xn wn) = f(x.w)
    """
    return f(np.dot(np.insert(x,0,1),w))


def sig(s,T=1):
    """
    sigmoid

    s: signal
    T: temperature

    return: sigmoid(s)
    """
    return 1/(1+np.exp(-s/T))


def dsig(s, T=1):
    """
    derivative of sigmoid

    s: signal
    T: temperature

    return: dsigmoid(s,T)/ds
    """
    return sig(s)*(1-sig(s))/T


def lin(s,a=1):
    """
```

(continues on next page)

(continued from previous page)

```
    linear function

    s: signal
    a: constant

    return: a*s
    """
    return a*s


def dlin(s,a=1):
    """
    derivative of linear function

    s: signal
    a: constant

    return: a
    """
    return a


def relu(s):
    """
    ReLU function

    s: signal

    return: s if s>0, 0 otherwise
    """
    if s>0:
        return s
    else:
        return 0


def drelu(s):
    """
    derivative of ReLU function

    s: signal

    return: 1 if s>0, 0 otherwise
    """
    if s>0:
        return 1
    else:
        return 0


def lrelu(s,a=0.1):
    """
    Leaky ReLU function

    s: signal
    a: parameter

    return: s if s>0, a*s otherwise
    """
    if s>0:
        return s
```

(continues on next page)

(continued from previous page)

```
    else:
        return a*s


def dlrelu(s,a=0.1):
    """
    derivative of Leaky ReLU function

    s: signal
    a: parameter

    return: 1 if s>0, a otherwise
    """
    if s>0:
        return 1
    else:
        return a


def softplus(s):
    """
    softplus function

    s: signal

    return: log(1+exp(s))
    """
    return np.log(1+np.exp(s))


def dsoftplus(s):
    """
    derivative of softplus function

    s: signal

    return: 1/(1+exp(-s))
    """
    return 1/(1+np.exp(-s))


def l2(w0,w1,w2):
    """for separating line"""
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]


def eucl(p1,p2):
    """
    Square of the Euclidean distance between two points in 2-dim. space

    input: p1, p2 - arrays in the format [x1,x2]

    return: square of the Euclidean distance
    """
    return (p1[0]-p2[0])**2+(p1[1]-p2[1])**2


def rn():
    """
    return: random number from [-0.5,0.5]
    """
```

(continues on next page)

(continued from previous page)

```
    return np.random.rand()-0.5


def point_c():
    """
    return: array [x,y] with random point from a cirle
            centered at [0.5,0.5] and radius 0.4
            (used for examples)
    """
    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():
    """
    return: array [x,y] with 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 network 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 network weights to a constant value

    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)

    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]}
    """
```

(continues on next page)

(continued from previous page)

```
    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):
    """
    back propagation algorithm

    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 - dictionary of weights - UPDATED
    eps - learning speed
    f   - activation function
    df  - derivative of f
    """

    l=len(ar)-1 # number of neuron layers (= index of the output layer)
```

(continues on next page)

(continued from previous page)

```
    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

    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):
    """
```

(continues on next page)

(continued from previous page)

```
    backprop with different output activation

    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 - dictionary of weights - UPDATED
    eps - learning speed
    f   - activation function
    df  - derivative 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])
```

12.2.2 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(func.step,func.sig)
    """
    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:
```

(continues on next page)

(continued from previous page)

```
        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 nodes

    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)

# 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 with 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):
```

(continues on next page)

(continued from previous page)

```
        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

    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:
```

(continues on next page)

(continued from previous page)

```
                    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
    {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):
```

(continues on next page)

(continued from previous page)

```
            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):
    """for separating line"""
    return [-.1,1.1],[-(w0-w1*0.1)/w2,-(w0+w1*1.1)/w2]
```

12.3 How to cite

If you would like to cite this Jupyter Book, here is the BibTeX entry:

```
@book{WB2021,
  title={"Explaining neural networks in raw Python: lectures in Jupiter"},
  author={Wojciech Broniowski},
  isbn={978-83-962099-0-0},
  year={2021},
  url={https://ifj.edu.pl/strony/~broniows/nn}
  publisher={Wojciech Broniowski}
}
```

BIBLIOGRAPHY

[Bar16] P. Barry. *Head First Python: A Brain-Friendly Guide*. O'Reilly Media, 2016. ISBN 9781491919491. URL: https://books.google.pl/books?id=NIqNDQAAQBAJ.

[BH69] A. E. Bryson and Y.-C. Ho. *Applied optimal control: Optimization, estimation, and control*. Waltham, Mass: Blaisdell Pub. Co., 1969.

[FR13] J. Feldman and R. Rojas. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 2013. ISBN 9783642610684.

[Fre93] James A. Freeman. *Simulating Neural Networks with Mathematica*. Addison-Wesley Professional, 1993. ISBN 9780201566291.

[FS91] James A. Freeman and David M. Skapura. *Neural Networks: Algorithms, Applications, and Programming Techniques (Computation and Neural Systems Series)*. Addison-Wesley, 1991. ISBN 9780201513769.

[Gut16] John Guttag. *Introduction to computation and programming using Python: With application to understanding data*. MIT Press, 2016.

[HR72] K. K. Hartline and F. Ratcliff. Inhibitory interactions in the retina of limulus. *Handbook of sensory physiology*, VII(2):381–447, 1972.

[KSJ+12] E.R. Kandel, J.H. Schwartz, T.M. Jessell, S.A. Siegelbaum, and A.J. Hudspeth. *Principles of Neural Science, Fifth Edition*. McGraw-Hill Education, 2012. ISBN 9780071810012. URL: https://books.google.pl/books?id=Z2yVUTnlIQsC.

[Mat19] E. Matthes. *Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming*. No Starch Press, 2019. ISBN 9781593279295. URL: https://books.google.pl/books?id=boBxDwAAQBAJ.

[MP43] Warren S. McCulloch and Walter Pitts. The logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943. URL: https://doi.org/10.1007/BF02478259.

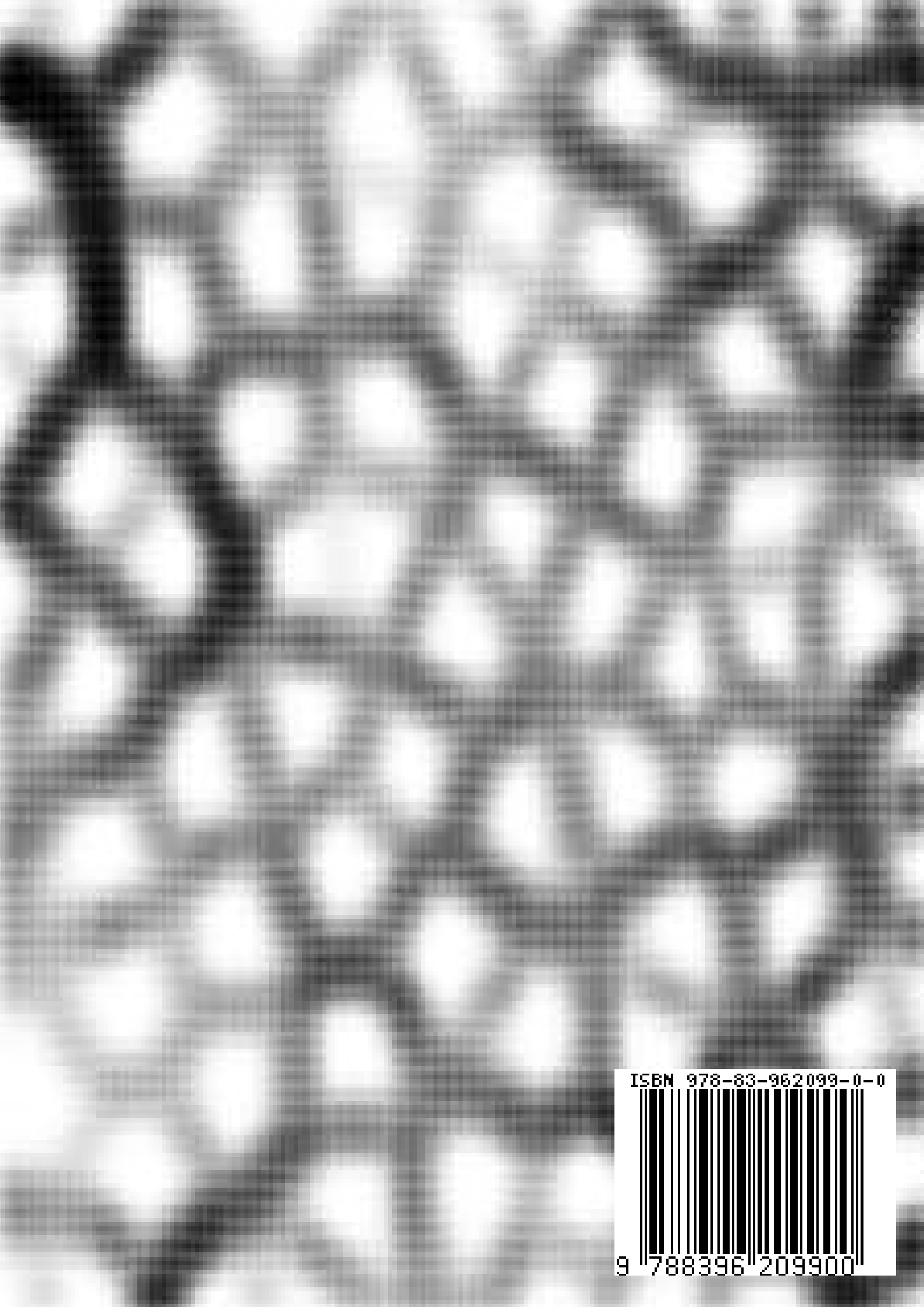
[MullerRS12] B. Müller, J. Reinhardt, and M.T. Strickland. *Neural Networks: An Introduction*. Physics of Neural Networks. Springer Berlin Heidelberg, 2012. ISBN 9783642577604. URL: https://books.google.pl/books?id=on0QBwAAQBAJ.

[RIV91] A. K. Rigler, J. M. Irvine, and Thomas P. Vogl. Rescaling of variables in back propagation learning. *Neural Networks*, 4(2):225–229, 1991. URL: https://doi.org/10.1016/0893-6080(91)90006-Q, doi:10.1016/0893-6080(91)90006-Q.

The author is a Professor of Physics in the Jan Kochanowski University, Kielce, Poland, and in the Henryk Niewodniczański Institute of Nuclear Physics PAN, Cracow.

Front cover: Kohonen's mapping of a sample of 40 normalized RGB colors on a square.
Back cover: the corresponding universal distance matrix.

Editor: Wojciech Broniowski
Wojciech.Broniowski@ifj.edu.pl
www.ifj.edu.pl/~broniows/nn
First Edition, 2021
ISBN: 978-83-962099-0-0



ISBN 978-83-962099-0-0
9 788396 209900