1. Technical information

- This program was written in Python 3 (version: 3.6.4)
- This program was written and compiled using Windows (version: 10-10.0.17134-SP0)
- The following modules are used here:
    - matplotlib (version: 2.0.2)
        - Both the main module and the submodule 'pyplot'.
    - numpy (version: 1.14.2)
    - math (version: N/A)

2. Additional (slightly technical) information

2.1. Navigation in the course text

2.1.1. .docx

This course text is divided into different parts. You can easily navigate the text in Word (using Windows) by pressing CTRL + F, and then by going to the *headings (*'Koppen' in Dutch) section. Alternatively, you can go to the '*view'* tab in Word, and press '*Navigation pane'* ('*beeld' and 'Navigatiedeelvenster'* respectively in Dutch).
This feature allows you to skip certain parts of the course text (e.g. if you want to go immediately to the exercises). In Mac, you can also access the navigation pane by going to '*view'*. We are not aware of a keyboard shortcut that allows immediate access to a navigation pane in Mac.

2.1.2. .pdf

If you are reading this file in pdf format, you can profit from the hierarchical build by clicking on this symbol:

This can be found on the left part of your screen when you open the text in pdf format. Clicking on this '*bookmarks'* symbol will allow you to see the hierarchy, and should give you access to navigation. This should be the same across operating systems.
Alternatively, you can press F4 to show the navigation space, or access the bookmarks via the '*view*' tab.

2.2. Weblinks

We highlight that embedded links to useful websites are marked by a **bold, underlined** font. If you see a link to a webpage, this means that this link leads to a webpage that focuses on a certain concept that we also explain in our course text. Thus, we always try to explain a specific concept in the course material, so you only should visit the page if our explanation proves insufficient. In other words: let's hope you never feel the urge to click on an embedded weblink (but *please do try* the first one to follow). In Windows' word, you can follow **links to useful websites** by holding *ctrl* and clicking on the bold, underlined words. In the pdf version, only clicking should suffice. In Mac, you should be able to open the website by simply clicking on the words.

2.3. Code blocks

Lines of code (be it in Python or another language) will be marked by a black block with code in it. You should be able to copy-paste the code in your own interpreter/shell. The code is tested on the system specified above and should be executable without errors. For clarity, you can find such a *code block* immediately below:

```
print('Hello world')
```

Copy-paste this code in your Python shell, and press enter. This should show 'Hello world' as output. This will probably (and hopefully) not come as a surprise to you. When the code is not meant for a Python interpreter this will be emphasized in the text.

3. Final checks

Now that everyone is familiar with the text format, we can finally start preparing for the exercises.

3.1. About modules

As the program heavily relies on the three modules specified under point 1, it is imperative that these are installed prior to using the provided script. Without these modules, the script will not work. You can check whether you have these modules installed by entering the following command in your Python shell:

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

If this doesn't yield any errors, you're good to go! However, if it gives you an error similar to this one:

```
Traceback (most recent call last):
  File "C:/Users/Pieter/Modeling/code/exercises ch04/exc_01.py", line 5, in <module>
    import matplotlib
ModuleNotFoundError: No module named matplotlib
```

… you should first solve this issue by installing the missing module. Several ways of installing Python modules are described **on this website**. The most common ways are using *pip* or a *virtual environment*.

The way I usually install packages is by using *pip*. If you have installed Python from source (i.e. downloaded 'python.exe' from the official website) pip should be included in the installation, so you should have it on your system. Test and update pip by following the steps specified below. If you have recently installed Python, or you make a habit of updating your software frequently, you can skip this.

3.2. Checking and updating pip

- Open your command prompt
    - *Not* the python prompt, but the one that functions as a command-line interpreter on your operating system.
        - In Windows, this is called cmd.exe
        - Mac calls this program terminal
        - Unix has named this command shell
- Check whether pip is available on your system by punching in (or copy-pasting) the following code in your command prompt (if you are certain that pip is installed you can obviously skip this step):

```
pip --version
```

- If this outputs something like:

```
pip 18.0 from c:\users\pieter\appdata\local\programs\python\python36\lib\site-packages\pip (python 3.6)
```

- You're ready for the next steps! If this is not the case, you should install pip by following the (straightforward) guide available on the **official webpage**.
- If pip is available on your system, type in the following in your command prompt:

```
python -m pip install --upgrade pip setuptools wheel
```

- When this step is completed, your pip is updated, and can simply install missing packages by inputting something like this in your command prompt:

```
pip install matplotlib
```

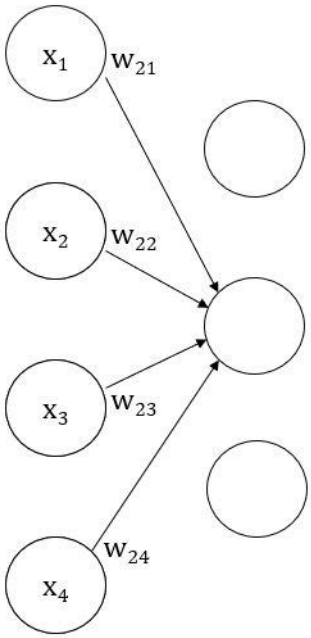
Python will yield a warning if your pip is older than the newest version: feel free to update as the need arises. Also, don't hesitate to ask for help if you encounter any issues while installing the required packages or pip itself.

4. Kicking the tires: chapter 4

4.1. Exercise 1

4.1.1. General idea

In this exercise, you will explore the properties of the activation function. We remember from our courses that the activation function is used to transform the input a unit receives. To grasp this concept, we can consider the following setup:



We know from the course material that the input to the receiving unit (in our example the second unit in the second layer) is calculated using the linearity principle (see the formula in chapter 4, which is reprinted right below).

$$in_i = \sum_j x_j w_{ij}$$

The activation level of the receiving unit is calculated by first multiplying the activation level of each sending unit (in our example this is denoted using the letter '*x*') with the strength of the connection between this sending unit and the receiving unit (denoted using the letter '*w*'). In the literature, the strength of the connection between a sending -, and a receiving unit is often referred to as '*weight*'. This makes sure that units that are strongly connected impact each other more than units that only share a weak connection. When this multiplication is completed for each sending unit-receiving unit pair, we add all results. Finally, we transform this result using an activation function.
This transformation can have several advantages, but one of the most useful properties of some types of activation functions (such as the logistic ones) is that it can prevent activation levels from going 'of the charts'.

To visualize this, we can consider a situation where the weight between the first sending unit and the receiving unit equals 1 000 000. For simplicity, we will assume that the activation level of each unit is .5, and the strength of all other connections between the sending units and the receiving units equals .6.

Thus:

- $x_1 = x_2 = \ldots = .5$
- $w_{21} = 1\ 000\ 000$
- $w_{22} = w_{23} = \ldots = .6$

Because programming is an essential part of this course, we will depict the linearity principle using Python code. You can copy-paste this code in a Python interpreter, and the code should run without any issues. *Code snippets* are marked with a '*coded*' banner, and can also be found under the bookmarks accessible through the navigation pane. All code snippets have a short, descriptive title.

Coded: linearity principle

```
first_weight = 1000000

weights = [first_weight, .6, .6, .6]
equal_activation = [.5]
activation_levels = equal_activation * 4

print('Weights:\n', weights)
print('Activation levels for sending units:\n', activation_levels)

results = []

for index in range(len(activation_levels)):
    multiplication = activation_levels[index] * weights[index]
    results.append(multiplication)

print('Activation level * weight for each sending unit:\n', results)

netinput = sum(results)
print('Transformed result using a linear activation function:\n', netinput)
```

This code is simply a depiction of the linearity principle applied on our little example network. The activation of each sending unit is multiplied with the weight between the sending and the receiving unit, these results are appended to a list, and finally, they are summed. *You should study the code, and its output until you understand it*! We see the output:

```
Activation level * weight for each sending unit:
 [500000.0, 0.3, 0.3, 0.3]
Transformed result using a linear activation function:
 500000.89999999997
```

This output should be fed to an activation function before it should be forwarded further into the network. For the sake of simplicity, we used a linear activation function here. When we use a linear activation function, we have an activation level of 500000.89999999997, because a linear activation function is of the form

$$f(result\ linearity\ principle) = result\ linearity\ principle$$

So, no transformation is performed, and an activation level of 500 000 for this unit is passed to the next cycle where it will be used in another multiplication, and thus it will also blow up the activation level of all units to which it is connected.

When a logistic activation function is used, such as the one we see in 4.1:

$$y_i = \frac{1}{1+\exp\left(-b'\ (in_i - q)\right)}$$

we see an entirely different result. Again, we will do this the Pythonic way:

Coded: logistic activation function

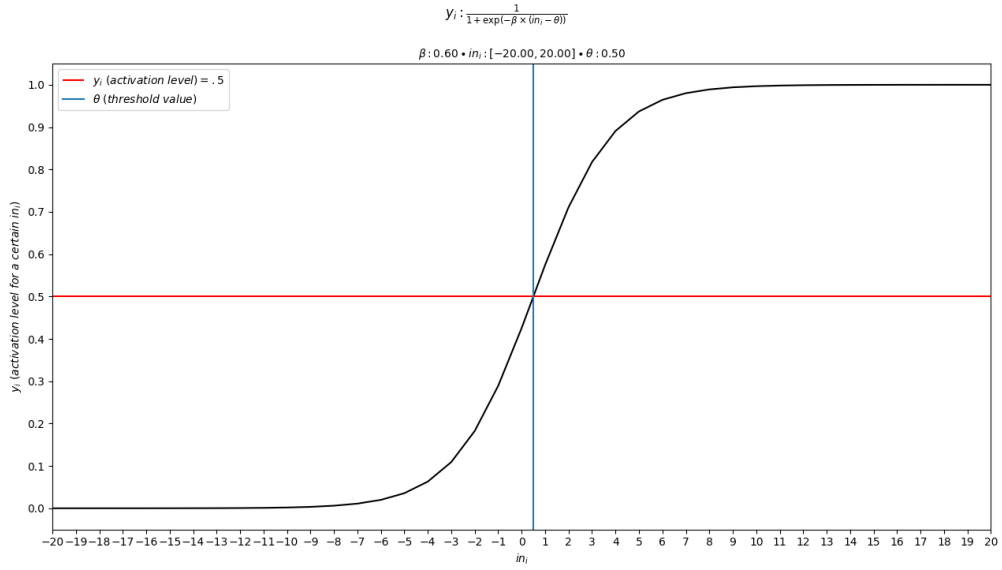
```
import math

first_weight = 1000000

weights = [first_weight, .6, .6, .6]
equal_activation = [.5]
activation_levels = equal_activation * 4

print('Weights:\n', weights)
print('Activation levels for sending units:\n', activation_levels)

results = []

for index in range(len(activation_levels)):
    multiplication = activation_levels[index] * weights[index]
    results.append(multiplication)

print('Activation level * weight for each sending unit:\n', results)

netinput = sum(results)
beta, theta = .6, .5

new_activation = 1 / (1 + math.exp(-beta*(netinput-theta)))
print('Transformed result using a linear activation function:\n', netinput)
print('Transformed result using a logistic activation function:\n',
new_activation)
```

Our new result:

```
Transformed result using a linear activation function:
 500000.89999999997
Transformed result using a logistic activation function:
 1.0
```

So, here we are able to contain our activation level between certain values, which makes sure that this unit will not have an 'exploding' effect in later cycles to come.

We can see this if we show the activation function, where the result of the linearity principle is ranged from -20 to 20:



we see that the activation function with β = .6, and θ = .5 has an S-like shape. This ensures that extreme inputs (such as 500 000) will still equal 1 after transformation (like we saw in the computation).

4.1.2. Assignment

4.1.2.1. General information

The aim of this exercise is to let you play around with the parameters in the following logistic activation function:

$$y_i = \frac{1}{1+\exp\left(-b'\ (in_i - q)\right)}$$

As we already briefly discussed, the activation function transforms the input a unit receives. Here, we see that the input calculated using the linearity principle (referred to as in_i) is fed into a function. However, it might be hard to imagine how the final activation level and the input fed into a unit are related. To visualize this, we made a script that lets you define values for the three parameters we see in the formula: β, in_i, and θ.

4.1.2.2. What should you do?

Simply download the script 'ch04_exercise_01.py', and compile it. This can be done in two ways:

- An editor:
    - Open the script in your favorite code editor (such as **notepad++**, **atom**, or **PyCharm**)
    - Compile the code
- Command prompt:
    - Find the file path of the specific script (when you download it, it will usually be located in the 'Downloads' directory)
        - In that case, the path will be something like this:
        - C:\Users\yourUsername\Downloads
    - Copy this path, and type the following in your command prompt:

```
cd C:\Users\yourUsername\Downloads
```

    - Be aware, of course, that you change everything after the *cd* (which stands for *change directory*) to the correct file path.
    - Then, type in the following in your command prompt:

```
python ch04_exercise_01.py
```

    - This should compile the script using the Python compiler, but the output will be in the command prompt window.

When the script has started, simply follow the instructions provided. You can play around with the parameters, and see what impact it has on the activation function. By doing so, you can more intuitively understand what an activation function is about, and how slope and the threshold value play a crucial role in it.