Setting up the working environment

Setting up the working environment

We will need to install **Anaconda**

- **Anaconda** is a software package that contains **Python** and **Jupyter Notebook**
- It also contains many useful libraries (NumPy, pandas, …)

We will need to also install the required packages to use **TensorFlow**

Why Python?

You have a lot of choices, but usually you see projects made in **Python** or **R**. But why are we going to use Python?

Python is:

- easy to learn
- general-purpose
- very high-level
- very nice and useful packages for ML (for example, scikit learn)
- also free!

Why Python?

Isn't Python very slow?

- Yes, it is ~100x slower than C
- But several packages are actually written in C/C++

  well, even Python is…
- Also, integration with **heterogenous programming** makes it a good choice for ML

Why Jupyter Notebook?

Jupyter is a server-client application that:

- runs your code in your web-browser
- it’s used at Google, Microsoft, IBM, …
- incorporates several languages and allows easy sharing

Installing Anaconda

1. Go to: https://www.anaconda.com/distribution/
2. Select your OS
3. Download Python 3.8 (64-bit)

Installing Anaconda

4. For compatibility issues, check: "*Register Anaconda as my default Python 3.8*"
   - It could raise an alert if you already have Python installed

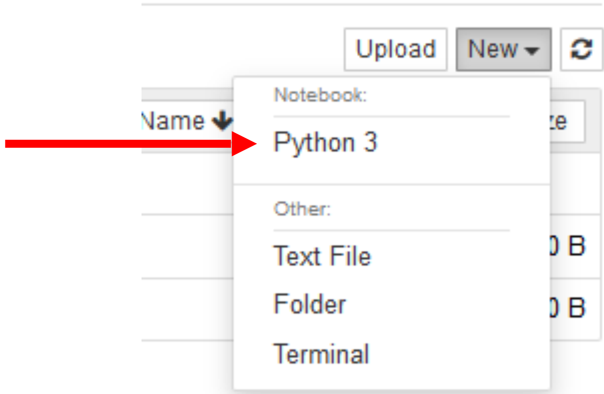
5. Don't select the Anaconda Cloud

Let's start

- Run **Jupyter Notebook** (now installed with Python)
    - It will open in the Browser
        - Don't close the "shell" window that might open

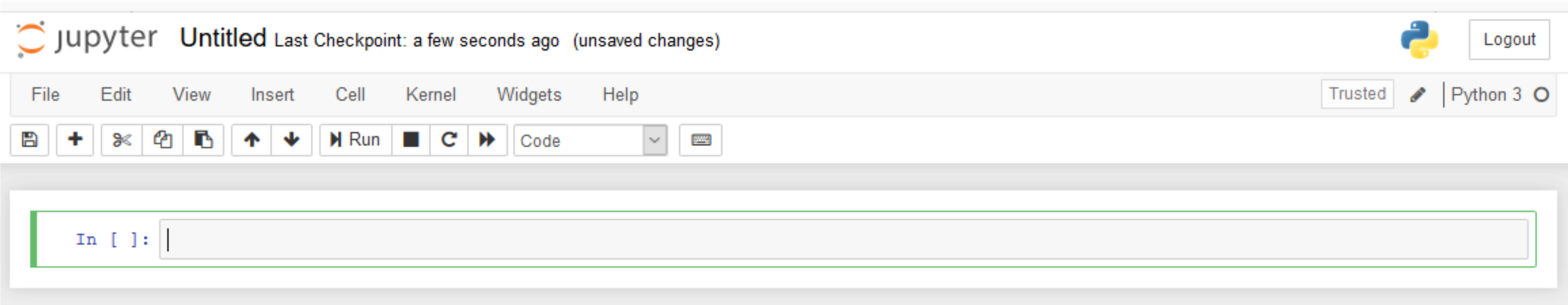
- Jupyter uses IPython Notebook Format (.ipynb)

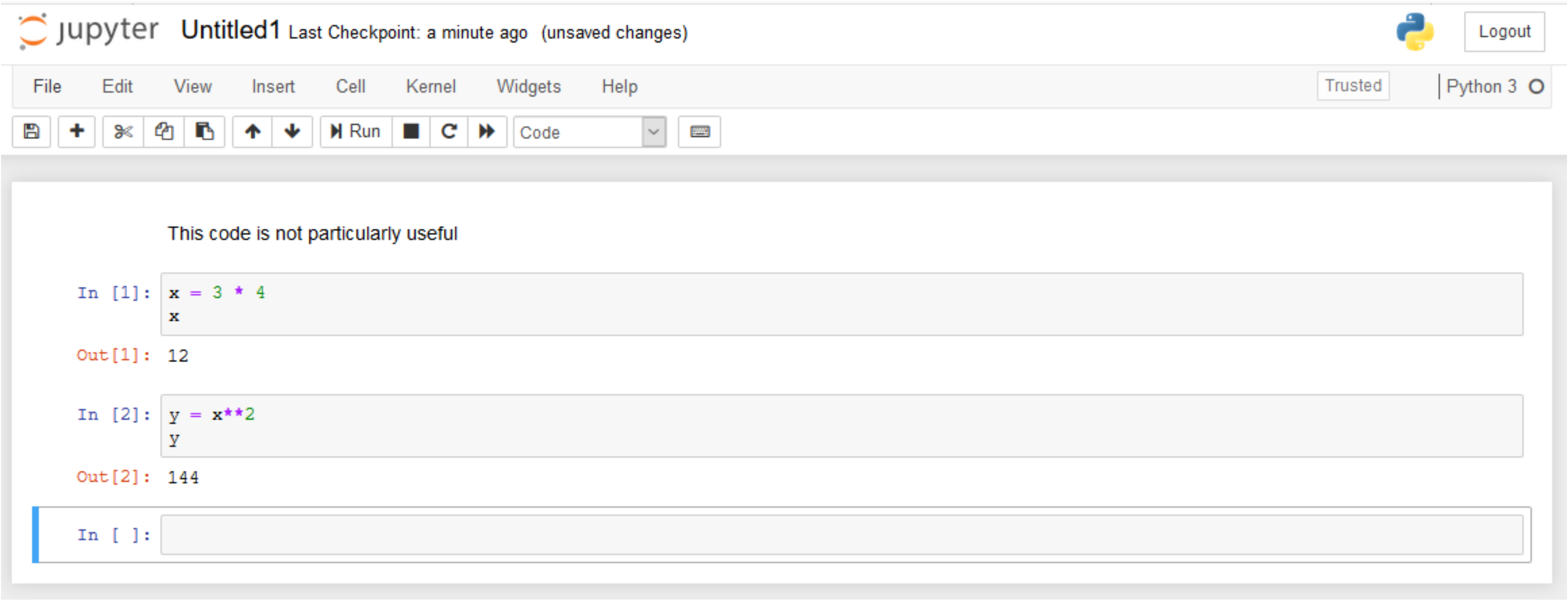
- Create a new Python 3 Notebook



Let’s start

- It should open a new tab like this:





- It splits the code in cells that you can run individually
  Very nice for debugging!

Installing TensorFlow 2.1.0

- Run “Anaconda Prompt” from your OS Menu
- Commands:

```
conda info --envs
```

```
(base) C:\Users\fabio>conda info --envs
# conda environments:
#
base                  *  C:\Users\fabio\Anaconda3
```

```
conda create --name env_name python=3
```

```
(base) C:\Users\fabio>conda create --name Python3-TensorFlow2 python=3
```

```
The following NEW packages will be INSTALLED:

  ca-certificates    pkgs/main/win-64::ca-certificates-2019.11.27-0
  certifi            pkgs/main/win-64::certifi-2019.11.28-py38_0
  openssl            pkgs/main/win-64::openssl-1.1.1d-he774522_3
  pip                pkgs/main/win-64::pip-19.3.1-py38_0
  python             pkgs/main/win-64::python-3.8.1-h5fd99cc_1
  setuptools         pkgs/main/win-64::setuptools-44.0.0-py38_0
  sqlite             pkgs/main/win-64::sqlite-3.30.1-he774522_0
  vc                 pkgs/main/win-64::vc-14.1-h0510ff6_4
  vs2015_runtime     pkgs/main/win-64::vs2015_runtime-14.16.27012-hf0eaf9b_1
  wheel              pkgs/main/win-64::wheel-0.33.6-py38_0
  wincertstore       pkgs/main/win-64::wincertstore-0.2-py38_0


Proceed ([y]/n)? y
```

Installing TensorFlow 2.1.0

- Commands:

```
conda activate env_name
```

```
(base) C:\Users\fabio>conda activate Python3-TensorFlow2

(Python3-TensorFlow2) C:\Users\fabio>
```

```
conda install tensorflow
pip install --upgrade tensorflow
pip install ipykernel
```

ipykernel installation ensures that Jupyter has the right kernel

Installing TensorFlow 2.1.0

Finally:

```
python -m ipykernel install --name env_name
```

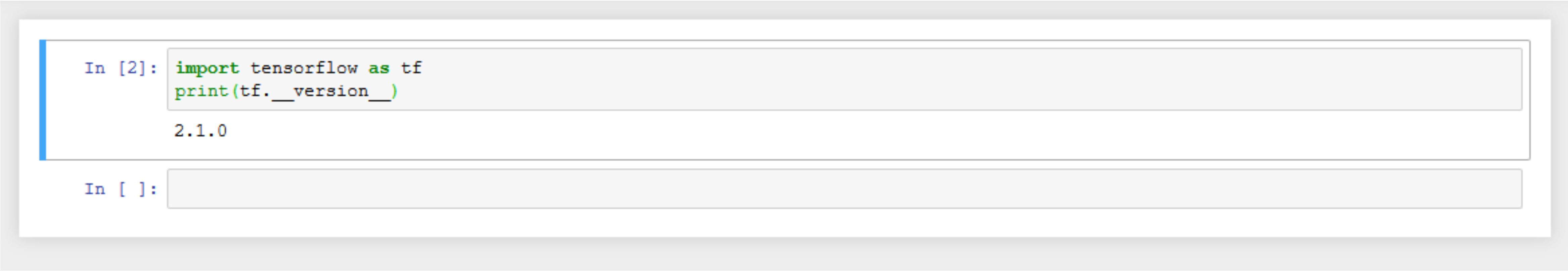
```
(Python3-TensorFlow2) C:\Users\fabio>python -m ipykernel install --name Python3-TensorFlow2
Installed kernelspec Python3-TensorFlow2 in C:\ProgramData\jupyter\kernels\python3-tensorflow2
```

```
conda info --envs
```

```
(Python3-TensorFlow2) C:\Users\fabio>conda info --envs
# conda environments:
#
base                     C:\Users\fabio\Anaconda3
Python3-TensorFlow2   *  C:\Users\fabio\Anaconda3\envs\Python3-TensorFlow2
```

Installing TensorFlow 2.1.0

- Now, close and open again Jupyter to test if everything went right
- Be sure to select the right Kernel



Installing Packages

- Let’s install scikit-learn and the TensorFlow datasets

```
pip install scikit-learn
pip install tensorflow-datasets
```

Let’s “machine learning” with Numpy

Elements of the model in supervised learning

- Inputs
- Weights
- Biases
- Targets
- Outputs

Import the relevant libraries

First: `pip install matplotlib`

Then:

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

In [ ]:
```

Matplotlib and Axes3D are not required, but they make cool graphs

Generating random input data for training

We will start with a textbook example

- We generate random data with a linear relationship

```
In [3]: observations_nr = 1000
        x_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))
        z_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))

        inputs = np.column_stack((x_values, z_values))
```

  - We select how many samples/observations we need (`observations_nr`)
  - We use `np.random.uniform` to generate these values
  - We use `np.column_stack` to literally stack two vectors in a matrix
  - Note that `inputs` is $(1000 \text{ x } 2)$

Create Targets

We now define the linear function that will act as target for our model

For example: $f(x,z) = \underset{w_1}{4}x - \underset{w_2}{3}z + \underset{b}{2} + \text{noise}$

- The **noise** ensures that the data looks more random

  In fact, real data always contains noise

In [4]:
```
noise = np.random.uniform(-1,1,(observations_nr,1))
targets = 4*x_values - 3*z_values + 2 + noise
```

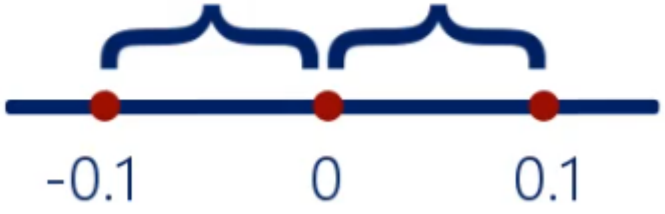
Let’s set up the initial variables

We saw previously that for gradient descent we could select a random starting point

- In this case, though, it is better to force the hand a little bit

➢We select random small initial weights and biases

We keep them within a small range



In [5]:
```
boundary_range = 0.1
weights = np.random.uniform(-boundary_range, boundary_range, (2,1))
biases = np.random.uniform(-boundary_range, boundary_range, 1)
```

Note that $\boldsymbol{W}$ is $(2 \text{ x } 1)$, while $\boldsymbol{b}$ is $(1 \ x \ 1)$

Now, let’s set up the leaning rate

- You can try different values and see what happens.

  Here an example:

```
learning_rate = 0.02
```

  Okay, we are all set.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
observations_nr = 1000
x_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))
z_values = np.random.uniform(low=-10,high=10,size=(observations_nr, 1))

inputs = np.column_stack((x_values, z_values))
```

In [3]:
```
noise = np.random.uniform(-1,1,(observations_nr,1))
targets = 4*x_values - 3*z_values + 2 + noise
```

Elements of the model in supervised learning

- ✓Inputs
- ✓Weights
- ✓Biases
- ✓Targets
- Outputs

We can now train the model

In [10]:
```
for i in range (100):
    outputs = np.dot(inputs, weights) + biases
    deltas = outputs - targets

    loss = np.sum(deltas**2) / 2 / observations_nr
    print(loss)

    deltas_scaled = deltas / observations_nr

    weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
    biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We run the algorithm over 100 iterations

  This is an arbitrary number

- For this problem, it is more than enough

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We calculate the outputs for the given weights and biases

  They were random, so likely far from the targets

We can now train the model

In [10]:

```
for i in range (100):
    outputs = np.dot(inputs, weights) + biases
    deltas = outputs - targets

    loss = np.sum(deltas**2) / 2 / observations_nr
    print(loss)

    deltas_scaled = deltas / observations_nr

    weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
    biases = biases - learning_rate * np.sum(deltas_scaled)
```

$$f(x) = x_1 * w_1 + x_2 * w_2 + b$$

$$f(x) = \underset{x}{\begin{bmatrix} x_1 & x_2 \end{bmatrix}} \cdot \underset{w}{\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}} + \begin{bmatrix} b \end{bmatrix}$$

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We compute the deltas, that is, the error between outputs and targets

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We compute the Loss function that compares the outputs with the targets
- We used:

$$L(y,t) = \frac{L2-norm}{2} = \frac{\sum_i (y_i - t_i)^2}{2}$$

We can now train the model

In [10]:
```
for i in range (100):
    outputs = np.dot(inputs, weights) + biases
    deltas = outputs - targets

    loss = np.sum(deltas**2) / 2 / observations_nr
    print(loss)

    deltas_scaled = deltas / observations_nr

    weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
    biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We also divided by the number of observations
  - We do this to make the **learning independent of the number of observations**

Also, it does not affect the logic of the Loss (it's just a division by a constant)

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We `print` the `loss` because we want to see if it is **decreasing**
- Otherwise, we need to change the **learning rate**

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- Even this step is done to make the algorithm independent of the number of observations

We can now train the model

In [10]:
```
for i in range (100):
    outputs = np.dot(inputs, weights) + biases
    deltas = outputs - targets

    loss = np.sum(deltas**2) / 2 / observations_nr
    print(loss)

    deltas_scaled = deltas / observations_nr

    weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
    biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We update the weights and the biases following the gradient descent methodology
- Note the `inputs.T`

  We are computing the transpose

$$w_{i+1} = w_i - \eta \nabla_w L(y, t)$$
$$b_{i+1} = b_i - \eta \nabla_b L(y, t)$$

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- We update the weights and the biases following the gradient descent methodology

$$w_{i+1} = w_i - \eta \sum_i \boldsymbol{x}_i \delta_i$$

$$b_{i+1} = b_i - \eta \sum_i \delta_i$$

- Note the `inputs.T`

  We are computing the transpose

We can now train the model

```
In [10]: for i in range (100):
             outputs = np.dot(inputs, weights) + biases
             deltas = outputs - targets

             loss = np.sum(deltas**2) / 2 / observations_nr
             print(loss)

             deltas_scaled = deltas / observations_nr

             weights = weights - learning_rate * np.dot(inputs.T, deltas_scaled)
             biases = biases - learning_rate * np.sum(deltas_scaled)
```

- The transpose is required, because sometime `inputs` and `deltas_scaled` matrices cannot be multiplied (dot) together

  In fact, it would be a $(1000x2) \bullet (1000x1)$

  - However, `inputs.T is` $(2x1000)$ and the dot product would now work

230.4481472811235
39.37426484576863
14.846919473742070
11.36348899028492
10.55340112064197
10.09573104165190
9.69666553008133
9.31851895180305
8.9559932818640
8.60790503266826
8.27361108614917
7.95255617403567
7.64421490183837
7.34808367703663
7.06367897491724
6.79053643382422
6.52821008240352
6.27627160960492
6.03430966512812
5.80192918776475
5.57875076036200
5.36440999033215
5.15855691469380
4.96085542867288
4.77098273693060

...

It's fast
It's minimizing the error (Loss function)

We trained the model correctly!

0.752536016397956
0.729308844629343
0.707001447178622
0.685577401841476
0.66500172869887
0.645240833003935
0.626262450330373
0.608035593893072
0.59053050395467
0.573718599235653
0.557572430248445
0.542065634479583
0.527172893346519
0.512869890858936
0.4991332739170389
0.485940614181972
0.4732703714561702
0.4611018585137761
0.4494152073237661
0.4381913366105956
0.4274119206994213
0.417059359595011
0.4071167502455157
0.397567858944157
0.388397094823794

Bonus: let's check

```
print(weights, biases)
```

```
[[ 3.99920143]
 [-2.99934675]] [1.75006155]
```

$$f(x, z) = \boxed{4x - 3z} + \boxed{2} + \text{noise}$$

The weights seem right.

The bias… almost. Why is that?

- Probably, not enough iterations (100) or not perfect learning rate

Bonus: let's check

```
print(weights, biases)
```

```
[[ 3.99920143]
 [-2.99934675]] [1.75006155]
```

$$f(x, z) = \boxed{4x - 3z} + \boxed{2} + \text{noise}$$

➢Let's run again the block in Jupyter that contains the loop

- This will run just that part of the code, so obtaining other 100 iterations

Bonus: let’s check

```
print(weights, biases)
```

```
[[ 3.99645043]
 [-2.99907267]] [1.98452474]
```

$$f(x, z) = 4x - 3z + 2 + \text{noise}$$

If we print again the values we get so much closer!

Recap of the “tuning” parameters

- Number of observations
- Learning rate
- Number of iterations (or, better, **epochs**)
- Initial range for initializing the weights and biases