Exercises:

```
1. Change the number of observations to 100,000 and see what happens.
2. Play around with the learning rate. Values like 0.0001, 0.001, 0.1, 1 are all interesting to
observe.
3. Change the loss function. An alternative loss for regressions is the Huber loss.
The Huber loss is more appropriate than the L2-norm when we have outliers, as it is less sensitive to
them (in our example we don't have outliers, but you will surely stumble upon a dataset with outliers
in the future). The L2-norm loss puts all differences *to the square*, so outliers have a lot of
influence on the outcome.
The proper syntax of the Huber loss is 'huber_loss'
```

Import the relevant libraries

In [2]:
```python
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```

Data generation

We generate data using the exact same logic and code as the example from the previous notebook. The only difference now is that we save it to an npz file. Npz is numpy's file type which allows you to save numpy arrays into a single .npz file. We introduce this change because in machine learning most often:

- you are given some data (csv, database, etc.)
- you preprocess it into a desired format (later on we will see methods for preprocesing)
- you save it into npz files (if you're working in Python) to access later

Nothing to worry about - this is literally saving your NumPy arrays into a file that you can later access, nothing more.

In [3]:
```python
# First, we should declare a variable containing the size of the training set we want to generate.
observations = 1000

# We will work with two variables as inputs. You can think about them as x1 and x2 in our previous examples.
# We have picked x and z, since it is easier to differentiate them.
# We generate them randomly, drawing from an uniform distribution. There are 3 arguments of this method (low, h
# The size of xs and zs is observations x 1. In this case: 1000 x 1.
xs = np.random.uniform(low=-10, high=10, size=(observations,1))
zs = np.random.uniform(-10, 10, (observations,1))

# Combine the two dimensions of the input into one input matrix.
# This is the X matrix from the linear model y = x*w + b.
# column_stack is a Numpy method, which combines two matrices (vectors) into one.
generated_inputs = np.column_stack((xs,zs))

# We add a random small noise to the function i.e. f(x,z) = 2x - 3z + 5 + <small noise>
noise = np.random.uniform(-1, 1, (observations,1))

# Produce the targets according to our f(x,z) = 2x - 3z + 5 + noise definition.
# In this way, we are basically saying: the weights should be 2 and -3, while the bias is 5.
generated_targets = 2*xs - 3*zs + 5 + noise

# save into an npz file called "TF_intro"
np.savez('TF_intro', inputs=generated_inputs, targets=generated_targets)
```

Solving with TensorFlow

Note: This intro is just the basics of TensorFlow which has way more capabilities and depth than that.

In [4]:
```python
# Load the training data from the NPZ
training_data = np.load('TF_intro.npz')
```

In [5]:
```python
# Declare a variable where we will store the input size of our model
# It should be equal to the number of variables you have
input_size = 2
# Declare the output size of the model
# It should be equal to the number of outputs you've got (for regressions that's usually 1)
output_size = 1

# Outline the model
# We lay out the model in 'Sequential'
# Note that there are no calculations involved - we are just describing our network
model = tf.keras.Sequential([
                            # Each 'layer' is listed here
                            # The method 'Dense' indicates, our mathematical operation to be (xw + b)
                            tf.keras.layers.Dense(output_size,
                                                 # there are extra arguments you can include to customize your 
                                                 # in our case we are just trying to create a solution that is 
                                                 # as close as possible to our NumPy model
                                                 kernel_initializer=tf.random_uniform_initializer(minval=-0.1, 
                                                 bias_initializer=tf.random_uniform_initializer(minval=-0.1, ma
                                                 )
                            ])

# We can also define a custom optimizer, where we can specify the learning rate
custom_optimizer = tf.keras.optimizers.SGD(learning_rate=0.02)
# Note that sometimes you may also need a custom loss function
# That's much harder to implement and won't be covered in this course though

# 'compile' is the place where you select and indicate the optimizers and the loss
model.compile(optimizer=custom_optimizer, loss='mean_squared_error')

# finally we fit the model, indicating the inputs and targets
# if they are not otherwise specified the number of epochs will be 1 (a single epoch of training),
# so the number of epochs is 'kind of' mandatory, too
# we can play around with verbose; we prefer verbose=2
model.fit(training_data['inputs'], training_data['targets'], epochs=100, verbose=2)
```

```
Epoch 1/100
32/32 - 1s - loss: 38.1068 - 943ms/epoch - 29ms/step
Epoch 2/100
32/32 - 0s - loss: 1.1673 - 82ms/epoch - 3ms/step
Epoch 3/100
32/32 - 0s - loss: 0.4342 - 70ms/epoch - 2ms/step
Epoch 4/100
32/32 - 0s - loss: 0.3932 - 69ms/epoch - 2ms/step
Epoch 5/100
32/32 - 0s - loss: 0.3879 - 69ms/epoch - 2ms/step
Epoch 6/100
32/32 - 0s - loss: 0.4463 - 71ms/epoch - 2ms/step
Epoch 7/100
32/32 - 0s - loss: 0.3609 - 72ms/epoch - 2ms/step
Epoch 8/100
32/32 - 0s - loss: 0.3794 - 89ms/epoch - 3ms/step
Epoch 9/100
32/32 - 0s - loss: 0.3770 - 78ms/epoch - 2ms/step
Epoch 10/100
32/32 - 0s - loss: 0.3779 - 82ms/epoch - 3ms/step
Epoch 11/100
32/32 - 0s - loss: 0.3958 - 71ms/epoch - 2ms/step
Epoch 12/100
32/32 - 0s - loss: 0.3789 - 77ms/epoch - 2ms/step
Epoch 13/100
32/32 - 0s - loss: 0.3684 - 70ms/epoch - 2ms/step
Epoch 14/100
32/32 - 0s - loss: 0.3831 - 69ms/epoch - 2ms/step
Epoch 15/100
32/32 - 0s - loss: 0.4226 - 71ms/epoch - 2ms/step
Epoch 16/100
32/32 - 0s - loss: 0.3839 - 71ms/epoch - 2ms/step
Epoch 17/100
32/32 - 0s - loss: 0.4059 - 81ms/epoch - 3ms/step
Epoch 18/100
32/32 - 0s - loss: 0.3874 - 68ms/epoch - 2ms/step
Epoch 19/100
32/32 - 0s - loss: 0.3831 - 69ms/epoch - 2ms/step
Epoch 20/100
32/32 - 0s - loss: 0.3913 - 78ms/epoch - 2ms/step
Epoch 21/100
32/32 - 0s - loss: 0.3818 - 86ms/epoch - 3ms/step
Epoch 22/100
32/32 - 0s - loss: 0.3860 - 84ms/epoch - 3ms/step
Epoch 23/100
32/32 - 0s - loss: 0.3810 - 86ms/epoch - 3ms/step
Epoch 24/100
32/32 - 0s - loss: 0.3866 - 69ms/epoch - 2ms/step
Epoch 25/100
32/32 - 0s - loss: 0.4784 - 95ms/epoch - 3ms/step
Epoch 26/100
32/32 - 0s - loss: 0.3842 - 66ms/epoch - 2ms/step
Epoch 27/100
32/32 - 0s - loss: 0.3832 - 80ms/epoch - 2ms/step
Epoch 28/100
32/32 - 0s - loss: 0.4056 - 87ms/epoch - 3ms/step
Epoch 29/100
32/32 - 0s - loss: 0.4416 - 83ms/epoch - 3ms/step
Epoch 30/100
32/32 - 0s - loss: 0.3682 - 81ms/epoch - 3ms/step
Epoch 31/100
32/32 - 0s - loss: 0.3771 - 84ms/epoch - 3ms/step
Epoch 32/100
32/32 - 0s - loss: 0.3728 - 88ms/epoch - 3ms/step
Epoch 33/100
32/32 - 0s - loss: 0.3834 - 96ms/epoch - 3ms/step
Epoch 34/100
32/32 - 0s - loss: 0.4007 - 90ms/epoch - 3ms/step
Epoch 35/100
32/32 - 0s - loss: 0.4033 - 86ms/epoch - 3ms/step
Epoch 36/100
32/32 - 0s - loss: 0.4140 - 82ms/epoch - 3ms/step
Epoch 37/100
32/32 - 0s - loss: 0.3677 - 85ms/epoch - 3ms/step
Epoch 38/100
32/32 - 0s - loss: 0.3738 - 82ms/epoch - 3ms/step
Epoch 39/100
32/32 - 0s - loss: 0.4206 - 78ms/epoch - 2ms/step
Epoch 40/100
32/32 - 0s - loss: 0.3932 - 81ms/epoch - 3ms/step
Epoch 41/100
32/32 - 0s - loss: 0.3958 - 72ms/epoch - 2ms/step
Epoch 42/100
32/32 - 0s - loss: 0.3851 - 79ms/epoch - 2ms/step
Epoch 43/100
32/32 - 0s - loss: 0.4019 - 85ms/epoch - 3ms/step
Epoch 44/100
32/32 - 0s - loss: 0.3845 - 99ms/epoch - 3ms/step
Epoch 45/100
32/32 - 0s - loss: 0.3722 - 89ms/epoch - 3ms/step
Epoch 46/100
32/32 - 0s - loss: 0.3890 - 88ms/epoch - 3ms/step
Epoch 47/100
32/32 - 0s - loss: 0.3870 - 79ms/epoch - 2ms/step
Epoch 48/100
32/32 - 0s - loss: 0.3935 - 80ms/epoch - 3ms/step
Epoch 49/100
32/32 - 0s - loss: 0.4199 - 84ms/epoch - 3ms/step
Epoch 50/100
32/32 - 0s - loss: 0.3983 - 85ms/epoch - 3ms/step
Epoch 51/100
32/32 - 0s - loss: 0.3734 - 85ms/epoch - 3ms/step
Epoch 52/100
32/32 - 0s - loss: 0.3636 - 82ms/epoch - 3ms/step
Epoch 53/100
32/32 - 0s - loss: 0.3927 - 79ms/epoch - 2ms/step
Epoch 54/100
32/32 - 0s - loss: 0.3974 - 80ms/epoch - 3ms/step
Epoch 55/100
32/32 - 0s - loss: 0.3987 - 82ms/epoch - 3ms/step
Epoch 56/100
32/32 - 0s - loss: 0.4133 - 94ms/epoch - 3ms/step
Epoch 57/100
32/32 - 0s - loss: 0.3772 - 98ms/epoch - 3ms/step
Epoch 58/100
32/32 - 0s - loss: 0.3873 - 78ms/epoch - 2ms/step
Epoch 59/100
32/32 - 0s - loss: 0.3866 - 79ms/epoch - 2ms/step
Epoch 60/100
32/32 - 0s - loss: 0.3919 - 78ms/epoch - 2ms/step
Epoch 61/100
32/32 - 0s - loss: 0.4267 - 84ms/epoch - 3ms/step
Epoch 62/100
32/32 - 0s - loss: 0.3949 - 84ms/epoch - 3ms/step
Epoch 63/100
32/32 - 0s - loss: 0.3765 - 86ms/epoch - 3ms/step
Epoch 64/100
32/32 - 0s - loss: 0.3687 - 81ms/epoch - 3ms/step
Epoch 65/100
32/32 - 0s - loss: 0.3775 - 79ms/epoch - 2ms/step
Epoch 66/100
32/32 - 0s - loss: 0.4069 - 91ms/epoch - 3ms/step
Epoch 67/100
32/32 - 0s - loss: 0.4174 - 92ms/epoch - 3ms/step
Epoch 68/100
32/32 - 0s - loss: 0.3677 - 85ms/epoch - 3ms/step
Epoch 69/100
32/32 - 0s - loss: 0.3778 - 81ms/epoch - 3ms/step
Epoch 70/100
32/32 - 0s - loss: 0.3949 - 85ms/epoch - 3ms/step
Epoch 71/100
32/32 - 0s - loss: 0.3935 - 89ms/epoch - 3ms/step
Epoch 72/100
32/32 - 0s - loss: 0.4158 - 80ms/epoch - 3ms/step
Epoch 73/100
32/32 - 0s - loss: 0.3913 - 96ms/epoch - 3ms/step
Epoch 74/100
32/32 - 0s - loss: 0.3912 - 78ms/epoch - 2ms/step
Epoch 75/100
32/32 - 0s - loss: 0.3898 - 80ms/epoch - 3ms/step
Epoch 76/100
32/32 - 0s - loss: 0.4200 - 79ms/epoch - 2ms/step
Epoch 77/100
32/32 - 0s - loss: 0.3768 - 78ms/epoch - 2ms/step
Epoch 78/100
32/32 - 0s - loss: 0.3708 - 93ms/epoch - 3ms/step
Epoch 79/100
32/32 - 0s - loss: 0.4149 - 82ms/epoch - 3ms/step
Epoch 80/100
32/32 - 0s - loss: 0.4305 - 90ms/epoch - 3ms/step
Epoch 81/100
32/32 - 0s - loss: 0.4021 - 78ms/epoch - 2ms/step
Epoch 82/100
32/32 - 0s - loss: 0.3717 - 78ms/epoch - 2ms/step
Epoch 83/100
32/32 - 0s - loss: 0.3763 - 79ms/epoch - 2ms/step
Epoch 84/100
32/32 - 0s - loss: 0.3972 - 69ms/epoch - 2ms/step
Epoch 85/100
32/32 - 0s - loss: 0.3863 - 83ms/epoch - 3ms/step
Epoch 86/100
32/32 - 0s - loss: 0.3861 - 64ms/epoch - 2ms/step
Epoch 87/100
32/32 - 0s - loss: 0.3952 - 79ms/epoch - 2ms/step
Epoch 88/100
32/32 - 0s - loss: 0.4178 - 73ms/epoch - 2ms/step
Epoch 89/100
32/32 - 0s - loss: 0.3604 - 72ms/epoch - 2ms/step
Epoch 90/100
32/32 - 0s - loss: 0.3977 - 87ms/epoch - 3ms/step
Epoch 91/100
32/32 - 0s - loss: 0.3941 - 81ms/epoch - 3ms/step
Epoch 92/100
32/32 - 0s - loss: 0.4321 - 79ms/epoch - 2ms/step
Epoch 93/100
32/32 - 0s - loss: 0.3891 - 67ms/epoch - 2ms/step
Epoch 94/100
32/32 - 0s - loss: 0.4006 - 66ms/epoch - 2ms/step
Epoch 95/100
32/32 - 0s - loss: 0.4131 - 66ms/epoch - 2ms/step
Epoch 96/100
32/32 - 0s - loss: 0.4037 - 72ms/epoch - 2ms/step
Epoch 97/100
32/32 - 0s - loss: 0.3879 - 70ms/epoch - 2ms/step
Epoch 98/100
32/32 - 0s - loss: 0.3769 - 73ms/epoch - 2ms/step
Epoch 99/100
32/32 - 0s - loss: 0.3834 - 72ms/epoch - 2ms/step
Epoch 100/100
32/32 - 0s - loss: 0.3892 - 72ms/epoch - 2ms/step
```

Out[5]:
```
<keras.callbacks.History at 0x2b7eeb974c0>
```

Extract the weights and bias

Extracting the weight(s) and bias(es) of a model is not an essential step for the machine learning process. In fact, usually they would not tell us much in a deep learning context. However, this simple example was set up in a way, which allows us to verify if the answers we get are correct.

In [6]:
```python
# Extracting the weights and biases is achieved quite easily
model.layers[0].get_weights()
```

Out[6]:
```
[array([[ 1.9857517],
        [-2.9535973]], dtype=float32),
 array([5.008535], dtype=float32)]
```

In [7]:
```python
# We can save the weights and biases in separate variables for easier examination
# Note that there can be hundreds or thousands of them!
weights = model.layers[0].get_weights()[0]
weights
```

Out[7]:
```
array([[ 1.9857517],
       [-2.9535973]], dtype=float32)
```

In [8]:
```python
# We can save the weights and biases in separate variables for easier examination
# Note that there can be hundreds or thousands of them!
bias = model.layers[0].get_weights()[1]
bias
```

Out[8]:
```
array([5.008535], dtype=float32)
```

Extract the outputs (make predictions)

Once more, this is not an essential step, however, we usually want to be able to make predictions.

In [9]:
```python
# We can predict new values in order to actually make use of the model
# Sometimes it is useful to round the values to be able to read the output
# Usually we use this method on NEW DATA, rather than our original training data
model.predict_on_batch(training_data['inputs']).round(1)
```