Linear regression using the gradient descent

In this notebook, I'll show you the basics of neural networks. One of the most important concepts in neural networks is the gradient descent, this is the base of training a neural network. In this notebook, we will optimize a simple linear regression with the most simple gradient descent. In real problems optimizers are used in order to converge faster.

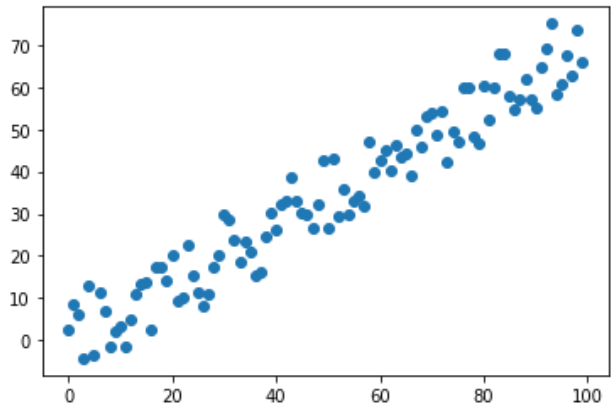
In [1]:
```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Linear regression is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). Imagine that we had the following data:

In [25]:
```
x = np.arange(100)
y = 0.7 * x + np.random.uniform(-10,10, size=(100,))

plt.scatter(x,y)
```

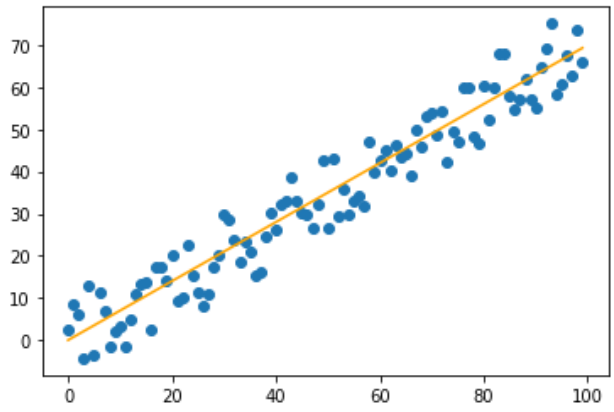
Out[25]:
```
<matplotlib.collections.PathCollection at 0x7fd4cc59d750>
```



As you see there's a dependency between x and y. A linear dependency, while x is increased y is increased. Linear regression would fit this model like the orange line on the following picture.

In [27]:
```
plt.scatter(x,y)
plt.plot(x, x*0.7, c = "orange")
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x7fd4cc304750>]
```



This was synthetic data, in real problemes, there's more than 1 variable and the relation isn't that simple. In the following section, we'll present the data that we'll be using.

1. Prepare the data

In this example, we'll use a simple dataset of sklearn. This is dataset contains 10 variables and a target. In a real problem, we would inspect the data but, in this study, we just care about the gradient descent algorithm. As you can see there are 10 dependent variables and an independent variable "target". Using the dependent variables we want to fit a model that can predict the target. In this case, we can't plot the data, there are too many dimensions and our head is just able to imagine 3 of them.

In [5]:
```
from sklearn import datasets

diabetes = datasets.load_diabetes()

df = pd.DataFrame(data= np.c_[diabetes['data'], diabetes['target']],
                  columns= diabetes['feature_names'] + ['target'])

df.head()
```

Out[5]:

	age	sex	bmi	bp	s1	s2	s3	s4	s5	s6
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641

In [6]:
```
inputs = df[df.columns[:-1]].to_numpy()
inputs.shape
```

Out[6]:
```
(442, 10)
```

In [7]:
```
targets = df.target.to_numpy().reshape((-1,1))
targets.shape
```

Out[7]:
```
(442, 1)
```

2. Model

The first thing we have to do is defining the model, the model will take inputs and then will compute the target using the input variables. In this case, the model will be a linear function.

- We define the model as: $prediction = X * m + n$, where:
- n : The bias.
- m : The weights.
- x : The inputs.

We create two tensors, imagine tensors like tensorflow numpy arrays. This tensors will contain the weights. We initialize with 0.5. We don't care about the shape, they'll automatically be adapted. We define the predict function, the model itself.

In [8]:
```
m = tf.Variable(0.5, shape=tf.TensorShape(None))
b = tf.Variable(0.5, shape=tf.TensorShape(None))

def predict(X):
```

```
    return(tf.math.reduce_sum(X*m, 1) + b)
```

We have defined the inputs taking all columns and rows except the target. This will be the X of our model. As you can see we'll have 10 weights, so the model will predict with the following function:

- $prediction = x_1 . w_1 + x_2 . w_2 + \ldots + x_9 . w_{10} + b$

We define the loss function, this is the function that we want to minimize. In this case we want to minimize the error between the predicted value and the real value, to do this we'll use the mean squared error.

In [10]:
```
def mse(prediction, real):
    return tf.math.reduce_mean(((prediction - real)**2))
```

3. Gradient descent

Now we have all ready, here comes the most intereseting part, the gradient descent. The idea behind this algorithm is very simple. We'll just minimize the error of the model descending step by step, using the partial derivatives of the loss function respect each variable. Let's see how it works. First we present the pseudocode:

- While current_step < total_steps:
- 1. Compute the model prediction.
- 2. Compute the loss using the prediction and the real values.
- 3. Compute the gradient of the loss respect each variable.
- 4. Update the weights using the gradient and the learning rate.
- 5. Step ++

Now we define two important parameters for the alogrism. The learning rate, this is what will guide the descent. If is bigger the descent will be faster but incosistent, if is smaller the descent will be more consistent but slower. Deciding this parameter is so important becuase a high or low learning rate can cause that the model never converge, it afects directly the algorithm performance. In this case just for show we will compute just 10 steps. Then we define the steps, this will be the number of times that we update the weigths using all the dataset, we are using batches of all dataset.

In [11]:
```
steps = 20
lr = 0.1
```

In [12]:
```
# We will save the loss though the steps
loss_history = []

# For each step
for i in range(steps):
    # We open a tensorflow's tape
    with tf.GradientTape() as tape:
        # Say to tensorflow which variables to trace
        tape.watch([m,b])
        # Get the prediction by the model
        prediction = predict(inputs)
        # Compute the loss
        loss = mse(prediction,targets)

    # Get the gradients respect the variables.
    grads = tape.gradient(loss, [m,b])

    # Update the weights
    m = m - grads[0]*lr
    b = b - grads[1]*lr

    loss_history.append(loss)
```

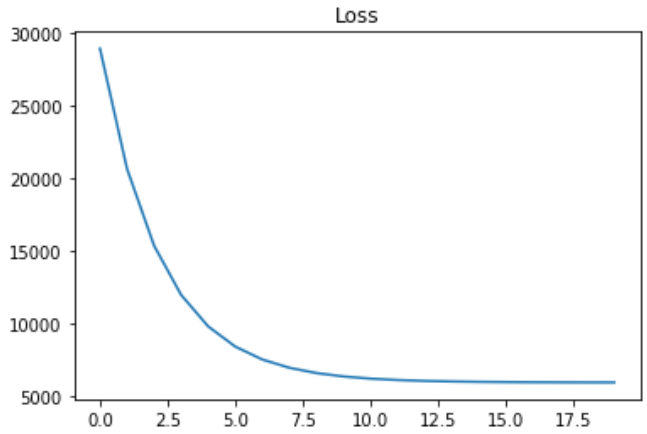
```
    print(loss)
    print("========")
```

```
tf.Tensor(28922.615, shape=(), dtype=float32)
========
tf.Tensor(20645.24, shape=(), dtype=float32)
========
tf.Tensor(15347.715, shape=(), dtype=float32)
========
tf.Tensor(11957.302, shape=(), dtype=float32)
========
tf.Tensor(9787.4375, shape=(), dtype=float32)
========
tf.Tensor(8398.724, shape=(), dtype=float32)
========
tf.Tensor(7509.9463, shape=(), dtype=float32)
========
tf.Tensor(6941.129, shape=(), dtype=float32)
========
tf.Tensor(6577.0854, shape=(), dtype=float32)
========
tf.Tensor(6344.0967, shape=(), dtype=float32)
========
tf.Tensor(6194.9854, shape=(), dtype=float32)
========
tf.Tensor(6099.553, shape=(), dtype=float32)
========
tf.Tensor(6038.477, shape=(), dtype=float32)
========
tf.Tensor(5999.3877, shape=(), dtype=float32)
========
tf.Tensor(5974.3706, shape=(), dtype=float32)
========
tf.Tensor(5958.3594, shape=(), dtype=float32)
========
tf.Tensor(5948.113, shape=(), dtype=float32)
========
tf.Tensor(5941.5537, shape=(), dtype=float32)
========
tf.Tensor(5937.3564, shape=(), dtype=float32)
========
tf.Tensor(5934.6704, shape=(), dtype=float32)
========
```

In the following figure we can see how the error between inputs and outputs is descending step by step.

In [13]:
```
fig ,ax = plt.subplots()
ax.plot(loss_history)
ax.set_title("Loss")
```

Out[13]:
```
Text(0.5, 1.0, 'Loss')
```



4. Prediction

Now we have the model trained, let's see how it works. We'll predict the first entry of the dataset, as you can see the difference between the prediction and the real value is so small. Now we can evaluate the model and take interesting metrics.

In [14]:
```
print("\n","Prediction: ", predict(inputs[0:1]).numpy(),"\n",
      "Real :", targets[0])
```

```
Prediction:  [150.40437]
Real : [151.]
```