Graded Practical Session

December 19, 2023

1 Exercise #1: Supervised Learning

In a regression problem, we have to predict a continuous dependent variable, like a price, from independent variables. We will use the dataset Auto MPG site and build some models to predict the energy efficiency (MPG) of vehicles of the end of 1970's and the beginning of 1980's. The independent variables (attributes) in this dataset are : # of cylinders, displacement, horse power, weight, acceleration, model year, origin, car name. 1. Download the dataset at: dataset 2. Read the data with pandas. 3. Clean the data by possibly removing rows with unknown values (dropna with pandas). 4. Visualize the data with the most adapted techniques to have a glance about the correlation of some pairs of attributes. 5. Divide the data into a training set and a test set (80% training, 20% test). 6. Normalize the data (preprocessing by normalization). 7. Train a linear regression model first, then train a Deep Neuron Network with two dense layers with relu activation functions. For the neuron network, use the optimizer Adam, and test the loss functions mean_absolute_error and mean_squared_error. 8. Compare the results of the different models (linear regression and MLP) on the test set.

For steps 2 to 8, write Python codes.

[1]:
```python
# 2. Read the data with pandas.

import pandas as pd

# Define the column names
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
                'acceleration', 'model_year', 'origin', 'car_name']

# Read the data
data = pd.read_csv('auto-mpg.data', delim_whitespace=True, names=column_names)

data.head()
```

[1]:
```
    mpg  cylinders  displacement horsepower  weight  acceleration  model_year  \
0  18.0          8         307.0      130.0  3504.0          12.0          70
1  15.0          8         350.0      165.0  3693.0          11.5          70
2  18.0          8         318.0      150.0  3436.0          11.0          70
3  16.0          8         304.0      150.0  3433.0          12.0          70
4  17.0          8         302.0      140.0  3449.0          10.5          70
```

```
   origin                   car_name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

[2]:
```python
# 3. Clean the data by possibly removing rows with unknown values (dropna with␣
 ↪pandas).

# Replace '?' by NaN values␣
 ↪(horsepower        Feature        Continuous                                  missing␣
 ↪: yes)
data['horsepower'].replace('?', pd.NA, inplace=True)

# Drop rows with NaN values
data.dropna(inplace=True)

data.head()
```

[2]:
```
    mpg  cylinders  displacement horsepower  weight  acceleration  model_year  \
0  18.0          8         307.0      130.0  3504.0          12.0          70
1  15.0          8         350.0      165.0  3693.0          11.5          70
2  18.0          8         318.0      150.0  3436.0          11.0          70
3  16.0          8         304.0      150.0  3433.0          12.0          70
4  17.0          8         302.0      140.0  3449.0          10.5          70

   origin                   car_name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

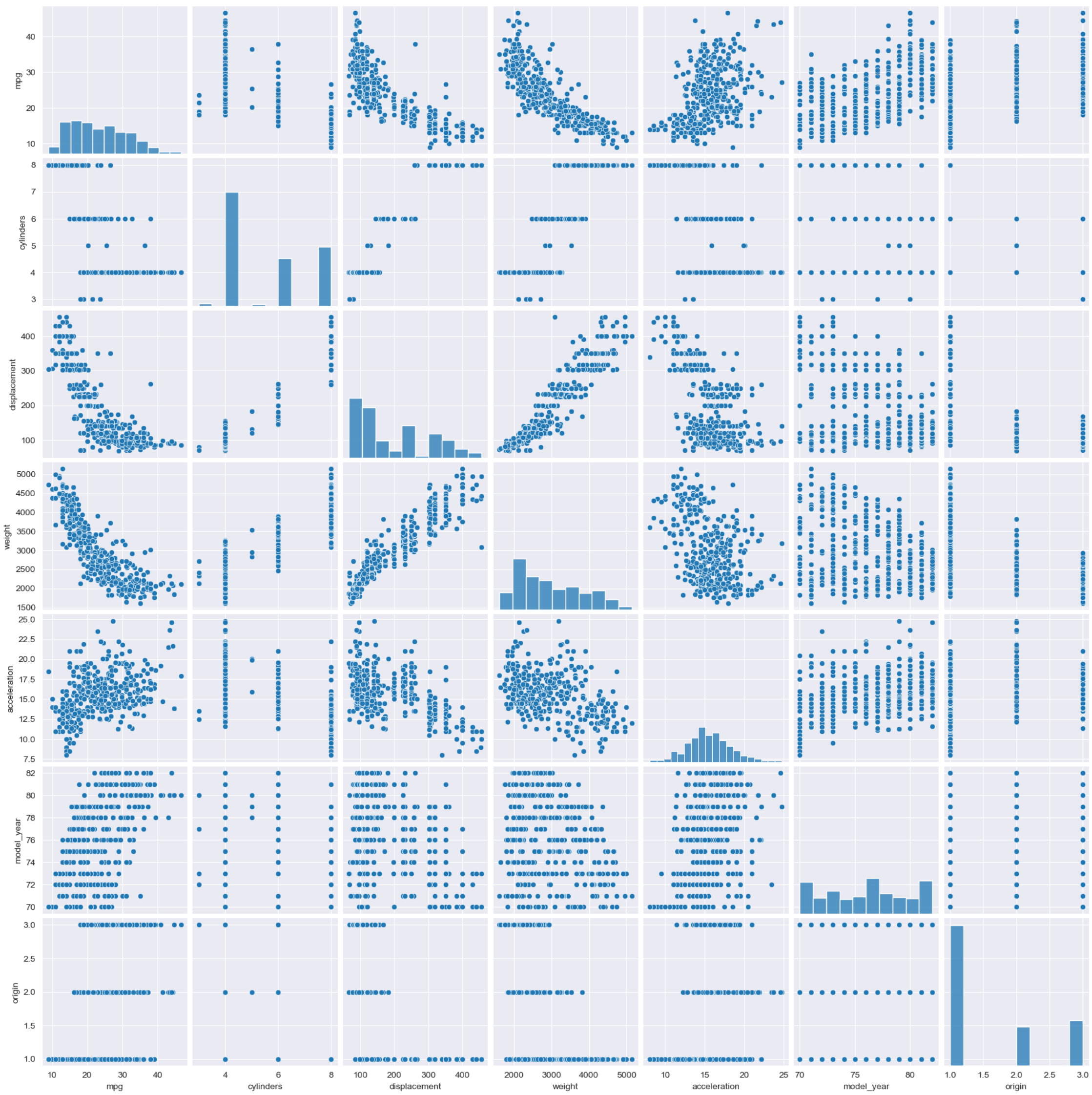
[3]:
```python
# 4. Visualize the data with the most adapted techniques to have a glance about␣
 ↪the correlation of some pairs of attributes.

import seaborn as sns
import matplotlib.pyplot as plt

# We drop the car_name column because it is not useful for the visualization
data_num = data.drop(['car_name'], axis=1)
```

[4]:
```python
# Pairplot to visualize the correlation between the attributes
sns.pairplot(data_num)
```

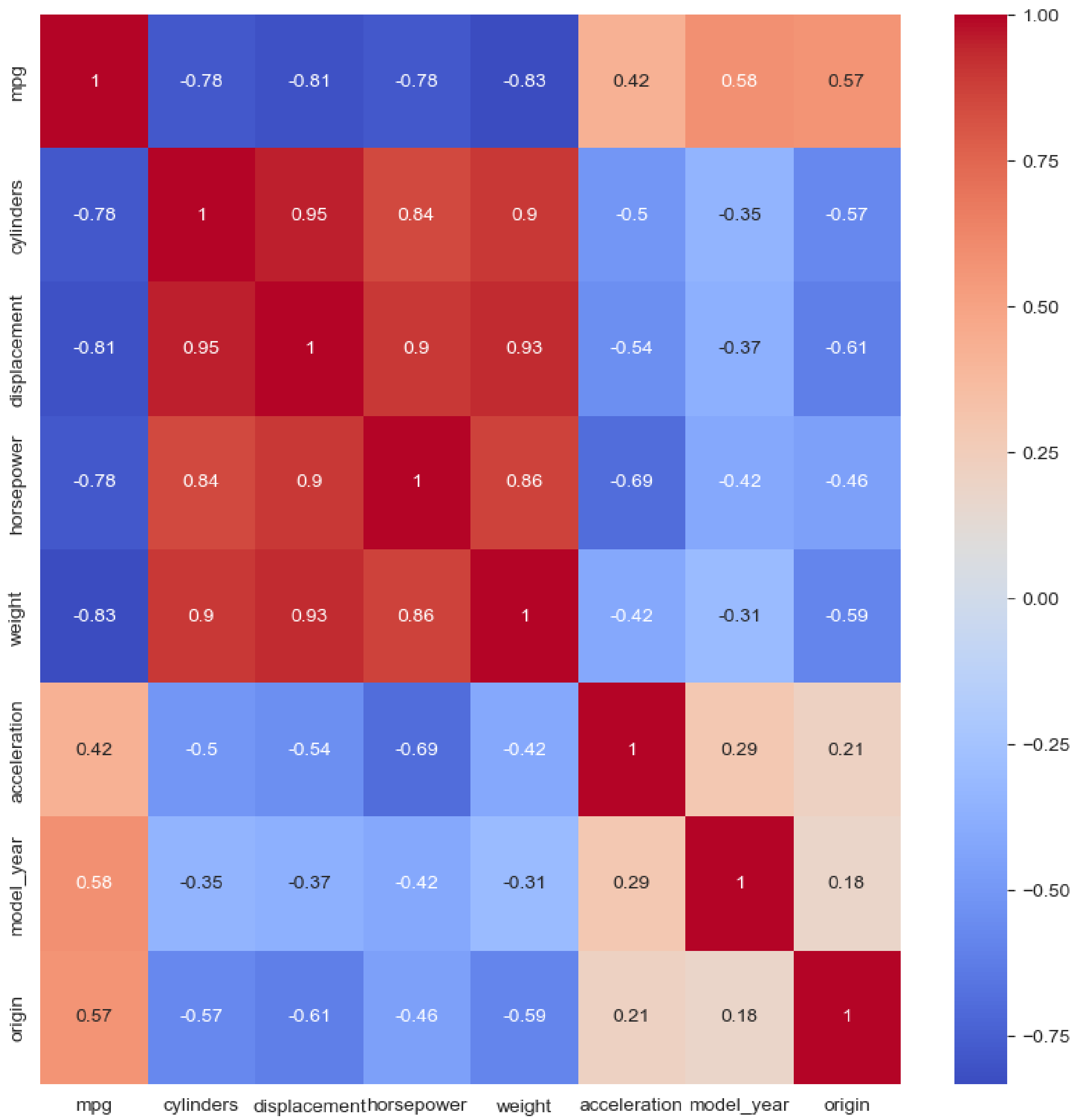
```
/Users/thibaultchausson/miniconda3/envs/AI53/lib/python3.8/site-
packages/seaborn/axisgrid.py:118: UserWarning: The figure layout has changed to
```

```
tight
  self._figure.tight_layout(*args, **kwargs)
```

[4]: <seaborn.axisgrid.PairGrid at 0x15e034c40>



[5]:
```
# Heatmap of the correlation between the attributes
plt.figure(figsize=(10, 10))
sns.heatmap(data_num.corr(), annot=True, cmap='coolwarm')
plt.show()
```



The most correlated attributes are : - mpg and model_year - mgp and origin - cylinders and displacement - cylinders and horsepower - cylinders and weight - displacement and horsepower - displacement and weight - horsepower and weight

[6]:
```python
# 5. Divide the data into a training set and a test set (80% training, 20% test).

from sklearn.model_selection import train_test_split

# Split the data into X and y
X = data.drop(['mpg', 'car_name', 'origin'], axis=1)
y = data['mpg']  # Target
```

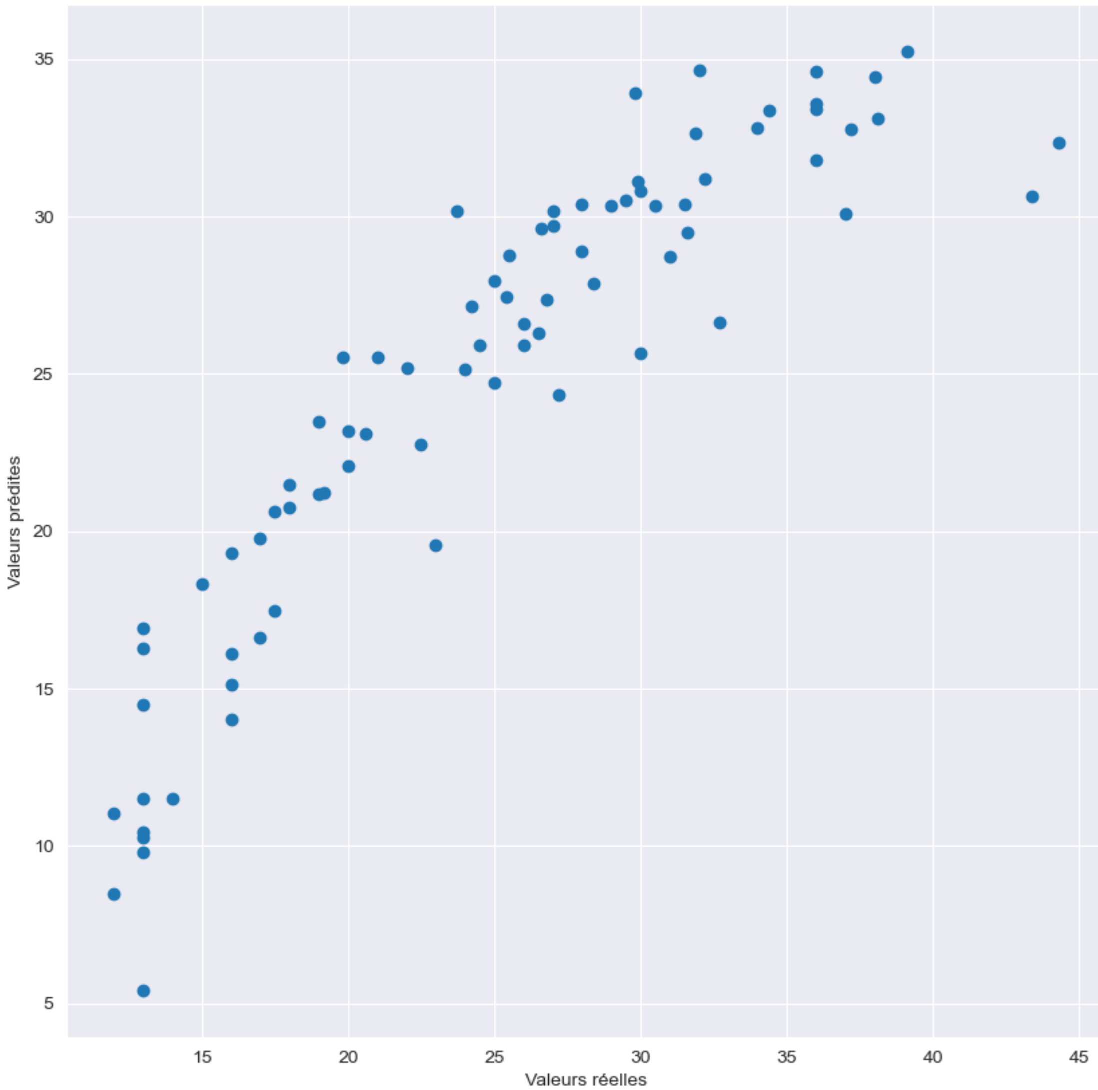
```
# Slit the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)  # We␣
↪can use also random_state=??
```

[7]:

```
# 6. Normalize the data (preprocessing by normalization).

from sklearn.preprocessing import StandardScaler

# Normalize the data, with the StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

[8]:

```
# 7. Train a linear regression model first, then train a Deep Neuron Network␣
↪with two dense layers with relu activation functions.

from sklearn.linear_model import LinearRegression

# Training the model linear regression
lr = LinearRegression()
lr.fit(X_train_scaled, y_train)

# Predictions
y_pred_lr = lr.predict(X_test_scaled)

# Plot the predictions
plt.figure(figsize=(10, 10))
plt.scatter(y_test, y_pred_lr)
plt.xlabel('Valeurs réelles')
plt.ylabel('Valeurs prédites')
plt.show()
```



[9]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dense(64, activation='relu'),
    Dense(1)
])

model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train_scaled, y_train, epochs=100)

y_pred_m = model.predict(X_test_scaled)
```

```python
plt.figure(figsize=(10, 10))
plt.scatter(y_test, y_pred_m)
plt.xlabel('Valeurs réelles')
plt.ylabel('Valeurs prédites')
plt.show()
```

```
Epoch 1/100

2023-12-19 18:42:51.868674: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M1 Pro
2023-12-19 18:42:51.868728: I metal_plugin/src/device/metal_device.cc:296]
systemMemory: 16.00 GB
2023-12-19 18:42:51.868743: I metal_plugin/src/device/metal_device.cc:313]
maxCacheSize: 5.33 GB
2023-12-19 18:42:51.868818: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:303]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
2023-12-19 18:42:51.868851: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:269]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0
MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
<undefined>)

 1/10 [==>...] - ETA: 3s - loss: 552.6580

2023-12-19 18:42:52.244746: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114]
Plugin optimizer for device_type GPU is enabled.

10/10 [==============================] - 0s 14ms/step - loss: 588.2451
Epoch 2/100
10/10 [==============================] - 0s 6ms/step - loss: 576.4590
Epoch 3/100
10/10 [==============================] - 0s 6ms/step - loss: 564.2844
Epoch 4/100
10/10 [==============================] - 0s 6ms/step - loss: 548.8628
Epoch 5/100
10/10 [==============================] - 0s 6ms/step - loss: 532.4020
Epoch 6/100
10/10 [==============================] - 0s 6ms/step - loss: 511.8532
Epoch 7/100
10/10 [==============================] - 0s 6ms/step - loss: 491.9503
Epoch 8/100
10/10 [==============================] - 0s 6ms/step - loss: 474.2872
Epoch 9/100
10/10 [==============================] - 0s 6ms/step - loss: 469.3499
Epoch 10/100
10/10 [==============================] - 0s 6ms/step - loss: 460.5447
```

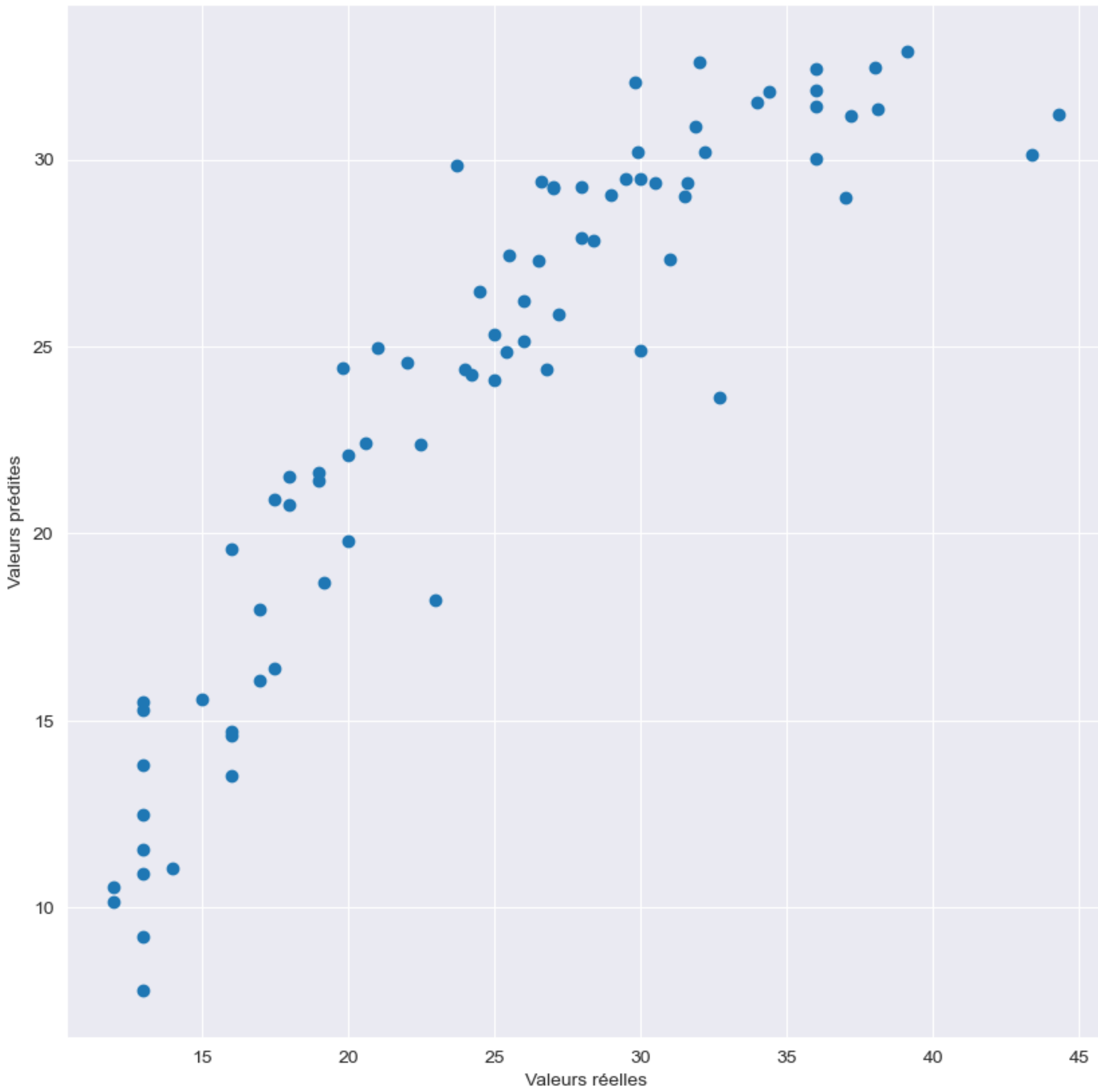
```
Epoch 11/100
10/10 [==============================] - 0s 6ms/step - loss: 448.7555
Epoch 12/100
10/10 [==============================] - 0s 6ms/step - loss: 434.0291
Epoch 13/100
10/10 [==============================] - 0s 6ms/step - loss: 417.3637
Epoch 14/100
10/10 [==============================] - 0s 6ms/step - loss: 400.4912
Epoch 15/100
10/10 [==============================] - 0s 6ms/step - loss: 385.8094
Epoch 16/100
10/10 [==============================] - 0s 6ms/step - loss: 371.6675
Epoch 17/100
10/10 [==============================] - 0s 6ms/step - loss: 357.7856
Epoch 18/100
10/10 [==============================] - 0s 6ms/step - loss: 339.0591
Epoch 19/100
10/10 [==============================] - 0s 6ms/step - loss: 324.7419
Epoch 20/100
10/10 [==============================] - 0s 6ms/step - loss: 308.6317
Epoch 21/100
10/10 [==============================] - 0s 6ms/step - loss: 287.8429
Epoch 22/100
10/10 [==============================] - 0s 6ms/step - loss: 275.0320
Epoch 23/100
10/10 [==============================] - 0s 6ms/step - loss: 255.3445
Epoch 24/100
10/10 [==============================] - 0s 6ms/step - loss: 242.9864
Epoch 25/100
10/10 [==============================] - 0s 6ms/step - loss: 223.8944
Epoch 26/100
10/10 [==============================] - 0s 6ms/step - loss: 209.1921
Epoch 27/100
10/10 [==============================] - 0s 6ms/step - loss: 194.1037
Epoch 28/100
10/10 [==============================] - 0s 6ms/step - loss: 183.2771
Epoch 29/100
10/10 [==============================] - 0s 6ms/step - loss: 167.0127
Epoch 30/100
10/10 [==============================] - 0s 6ms/step - loss: 151.4143
Epoch 31/100
10/10 [==============================] - 0s 6ms/step - loss: 139.3593
Epoch 32/100
10/10 [==============================] - 0s 6ms/step - loss: 129.1301
Epoch 33/100
10/10 [==============================] - 0s 6ms/step - loss: 125.4392
Epoch 34/100
10/10 [==============================] - 0s 6ms/step - loss: 113.9862
```

```
Epoch 35/100
10/10 [==============================] - 0s 6ms/step - loss: 99.2163
Epoch 36/100
10/10 [==============================] - 0s 6ms/step - loss: 85.7128
Epoch 37/100
10/10 [==============================] - 0s 6ms/step - loss: 79.2580
Epoch 38/100
10/10 [==============================] - 0s 6ms/step - loss: 72.7921
Epoch 39/100
10/10 [==============================] - 0s 6ms/step - loss: 64.3281
Epoch 40/100
10/10 [==============================] - 0s 6ms/step - loss: 55.8739
Epoch 41/100
10/10 [==============================] - 0s 6ms/step - loss: 48.6070
Epoch 42/100
10/10 [==============================] - 0s 6ms/step - loss: 42.9457
Epoch 43/100
10/10 [==============================] - 0s 6ms/step - loss: 40.9757
Epoch 44/100
10/10 [==============================] - 0s 6ms/step - loss: 35.5096
Epoch 45/100
10/10 [==============================] - 0s 6ms/step - loss: 32.1649
Epoch 46/100
10/10 [==============================] - 0s 6ms/step - loss: 29.1743
Epoch 47/100
10/10 [==============================] - 0s 6ms/step - loss: 27.2038
Epoch 48/100
10/10 [==============================] - 0s 6ms/step - loss: 26.7095
Epoch 49/100
10/10 [==============================] - 0s 6ms/step - loss: 23.7697
Epoch 50/100
10/10 [==============================] - 0s 6ms/step - loss: 23.8458
Epoch 51/100
10/10 [==============================] - 0s 6ms/step - loss: 22.5669
Epoch 52/100
10/10 [==============================] - 0s 6ms/step - loss: 21.9943
Epoch 53/100
10/10 [==============================] - 0s 6ms/step - loss: 22.6254
Epoch 54/100
10/10 [==============================] - 0s 6ms/step - loss: 21.7633
Epoch 55/100
10/10 [==============================] - 0s 6ms/step - loss: 18.5290
Epoch 56/100
10/10 [==============================] - 0s 6ms/step - loss: 17.8456
Epoch 57/100
10/10 [==============================] - 0s 6ms/step - loss: 16.0783
Epoch 58/100
10/10 [==============================] - 0s 6ms/step - loss: 17.5194
```

```
Epoch 59/100
10/10 [==============================] - 0s 6ms/step - loss: 16.9029
Epoch 60/100
10/10 [==============================] - 0s 6ms/step - loss: 15.4360
Epoch 61/100
10/10 [==============================] - 0s 6ms/step - loss: 15.0719
Epoch 62/100
10/10 [==============================] - 0s 6ms/step - loss: 15.6680
Epoch 63/100
10/10 [==============================] - 0s 6ms/step - loss: 15.2935
Epoch 64/100
10/10 [==============================] - 0s 6ms/step - loss: 18.5763
Epoch 65/100
10/10 [==============================] - 0s 6ms/step - loss: 15.7739
Epoch 66/100
10/10 [==============================] - 0s 6ms/step - loss: 18.5788
Epoch 67/100
10/10 [==============================] - 0s 6ms/step - loss: 15.3381
Epoch 68/100
10/10 [==============================] - 0s 6ms/step - loss: 15.1738
Epoch 69/100
10/10 [==============================] - 0s 6ms/step - loss: 17.9011
Epoch 70/100
10/10 [==============================] - 0s 6ms/step - loss: 15.1687
Epoch 71/100
10/10 [==============================] - 0s 6ms/step - loss: 14.8854
Epoch 72/100
10/10 [==============================] - 0s 6ms/step - loss: 13.9522
Epoch 73/100
10/10 [==============================] - 0s 6ms/step - loss: 14.3290
Epoch 74/100
10/10 [==============================] - 0s 6ms/step - loss: 15.1516
Epoch 75/100
10/10 [==============================] - 0s 6ms/step - loss: 15.5304
Epoch 76/100
10/10 [==============================] - 0s 6ms/step - loss: 13.5889
Epoch 77/100
10/10 [==============================] - 0s 6ms/step - loss: 13.8792
Epoch 78/100
10/10 [==============================] - 0s 6ms/step - loss: 14.3180
Epoch 79/100
10/10 [==============================] - 0s 6ms/step - loss: 13.8890
Epoch 80/100
10/10 [==============================] - 0s 6ms/step - loss: 14.2448
Epoch 81/100
10/10 [==============================] - 0s 6ms/step - loss: 14.0660
Epoch 82/100
10/10 [==============================] - 0s 6ms/step - loss: 14.6394
```

```
Epoch 83/100
10/10 [==============================] - 0s 6ms/step - loss: 14.0566
Epoch 84/100
10/10 [==============================] - 0s 6ms/step - loss: 13.7054
Epoch 85/100
10/10 [==============================] - 0s 6ms/step - loss: 15.7645
Epoch 86/100
10/10 [==============================] - 0s 6ms/step - loss: 15.9438
Epoch 87/100
10/10 [==============================] - 0s 6ms/step - loss: 15.7292
Epoch 88/100
10/10 [==============================] - 0s 6ms/step - loss: 17.0639
Epoch 89/100
10/10 [==============================] - 0s 6ms/step - loss: 13.6736
Epoch 90/100
10/10 [==============================] - 0s 6ms/step - loss: 13.0268
Epoch 91/100
10/10 [==============================] - 0s 6ms/step - loss: 13.1386
Epoch 92/100
10/10 [==============================] - 0s 6ms/step - loss: 13.4674
Epoch 93/100
10/10 [==============================] - 0s 6ms/step - loss: 13.7303
Epoch 94/100
10/10 [==============================] - 0s 6ms/step - loss: 14.0973
Epoch 95/100
10/10 [==============================] - 0s 6ms/step - loss: 19.4503
Epoch 96/100
10/10 [==============================] - 0s 6ms/step - loss: 20.3662
Epoch 97/100
10/10 [==============================] - 0s 6ms/step - loss: 18.3522
Epoch 98/100
10/10 [==============================] - 0s 6ms/step - loss: 19.4924
Epoch 99/100
10/10 [==============================] - 0s 6ms/step - loss: 15.2722
Epoch 100/100
10/10 [==============================] - 0s 6ms/step - loss: 15.2105
3/3 [==============================] - 0s 8ms/step
```

```
2023-12-19 18:42:58.802134: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:114]
Plugin optimizer for device_type GPU is enabled.
```



[10]:
```
# 8. Compare the results of the different models (linear regression and MLP) on
 ↪the test set.

from sklearn.metrics import mean_squared_error, mean_absolute_error

print('Linear Regression')
print('MSE :', mean_squared_error(y_test, y_pred_lr))
print('MAE :', mean_absolute_error(y_test, y_pred_lr))

print('\nMLP')
print('MSE :', mean_squared_error(y_test, y_pred_m))
print('MAE :', mean_absolute_error(y_test, y_pred_m))
```

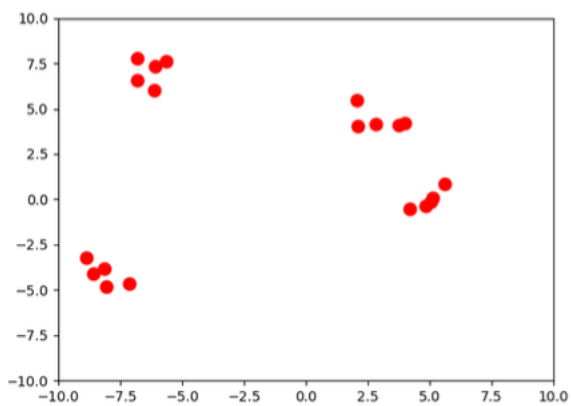
```
Linear Regression
```

```
MSE : 12.62476754954062
MAE : 2.7425887154000312

MLP
MSE : 13.962086354786473
MAE : 2.6886178125309037
```

We note that the linear regression model performs slightly less well than the neural network model. But the computation time of the neural network is much longer than that of linear regression. For this problem, linear regression is more appropriate.

2 Exercise #2: Unsupervised Learning

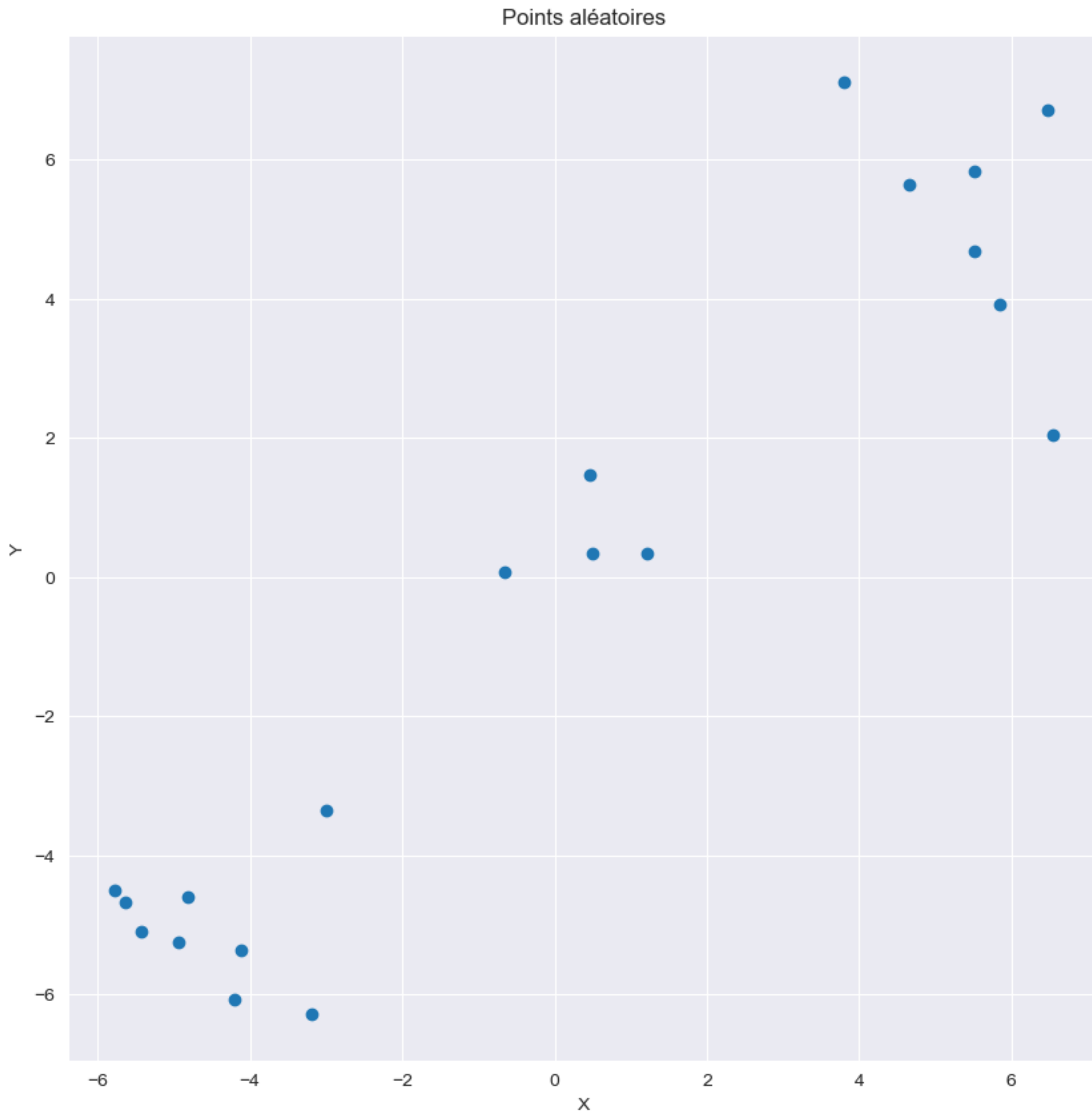
We assume the following cloud of 20 points randomly chosen in the interval [-10;10] :



Program a Python code that: 1. generates the random points, 2. by either using the template of the graded practice session about K-Means, program the K- Means algorithm with N centers, or use it from a Python library (scikit-learn for example), in order to find the best possible clustering of the previous data consisting of 20 random points ; we assume that the coordinates of the cluster centers in K-Means are also chosen in the interval [-10;10]. 3. Finally, program the algorithm that consists in executing multiple K-Means with different numbers of centers and displaying the best result.

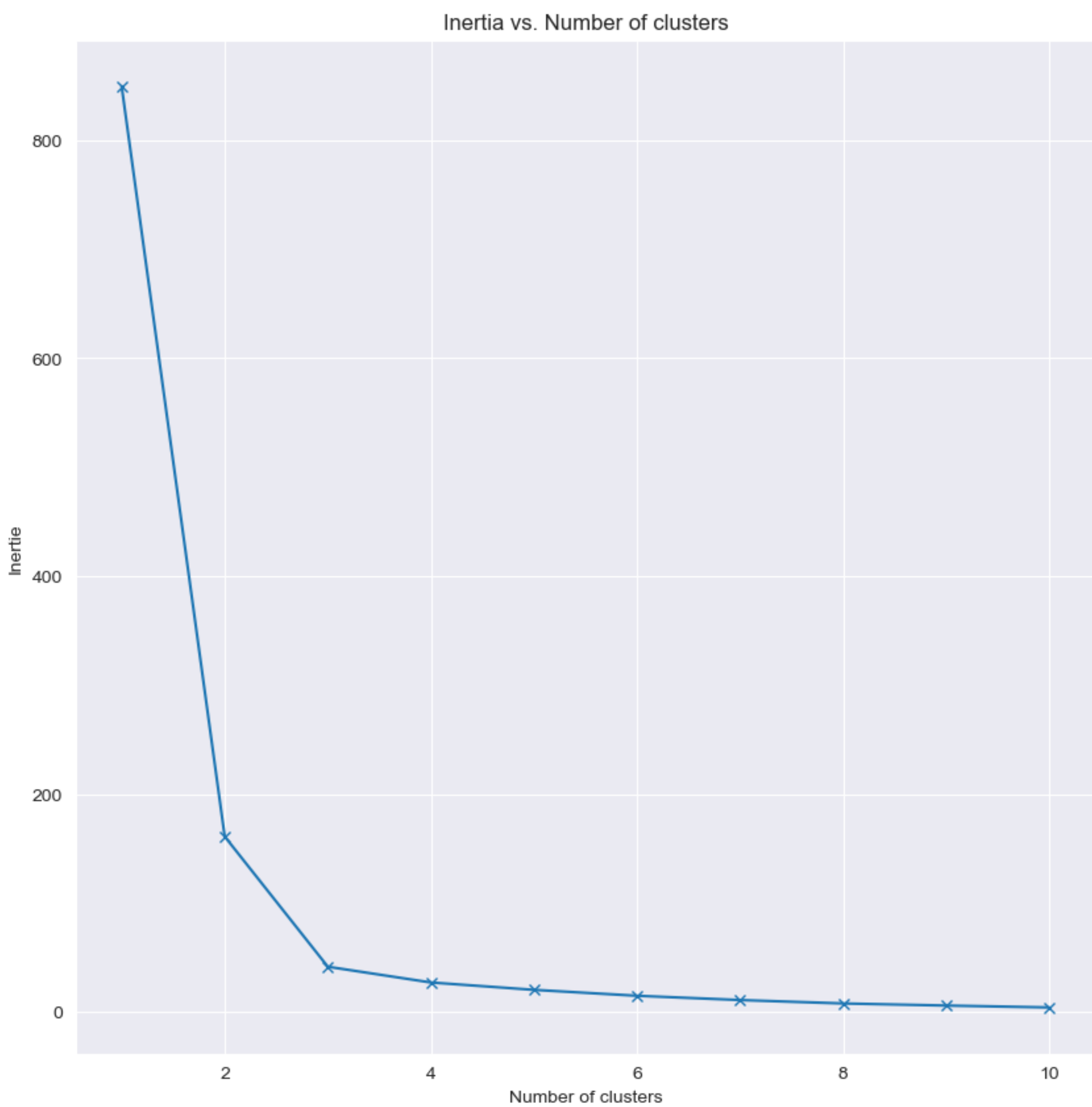
[11]:
```python
# 1. Generate the random points

import numpy as np

# Number of points per group
n_points = [4, 7, 9]

# All the centers of the groups
centers = [(0, 0), (5, 5), (-5, -5)]

# Generate points around each center
```

```python
points = []
k = 0
for center in centers:
    x_points = np.random.normal(center[0], 1, n_points[k])  # 1 est l'écart-type
    y_points = np.random.normal(center[1], 1, n_points[k])
    points.append(np.column_stack((x_points, y_points)))
    k += 1

# Merge all the points
points = np.vstack(points)

# Plot the points
plt.figure(figsize=(10, 10))
plt.scatter(points[:, 0], points[:, 1])
plt.title("Points aléatoires")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



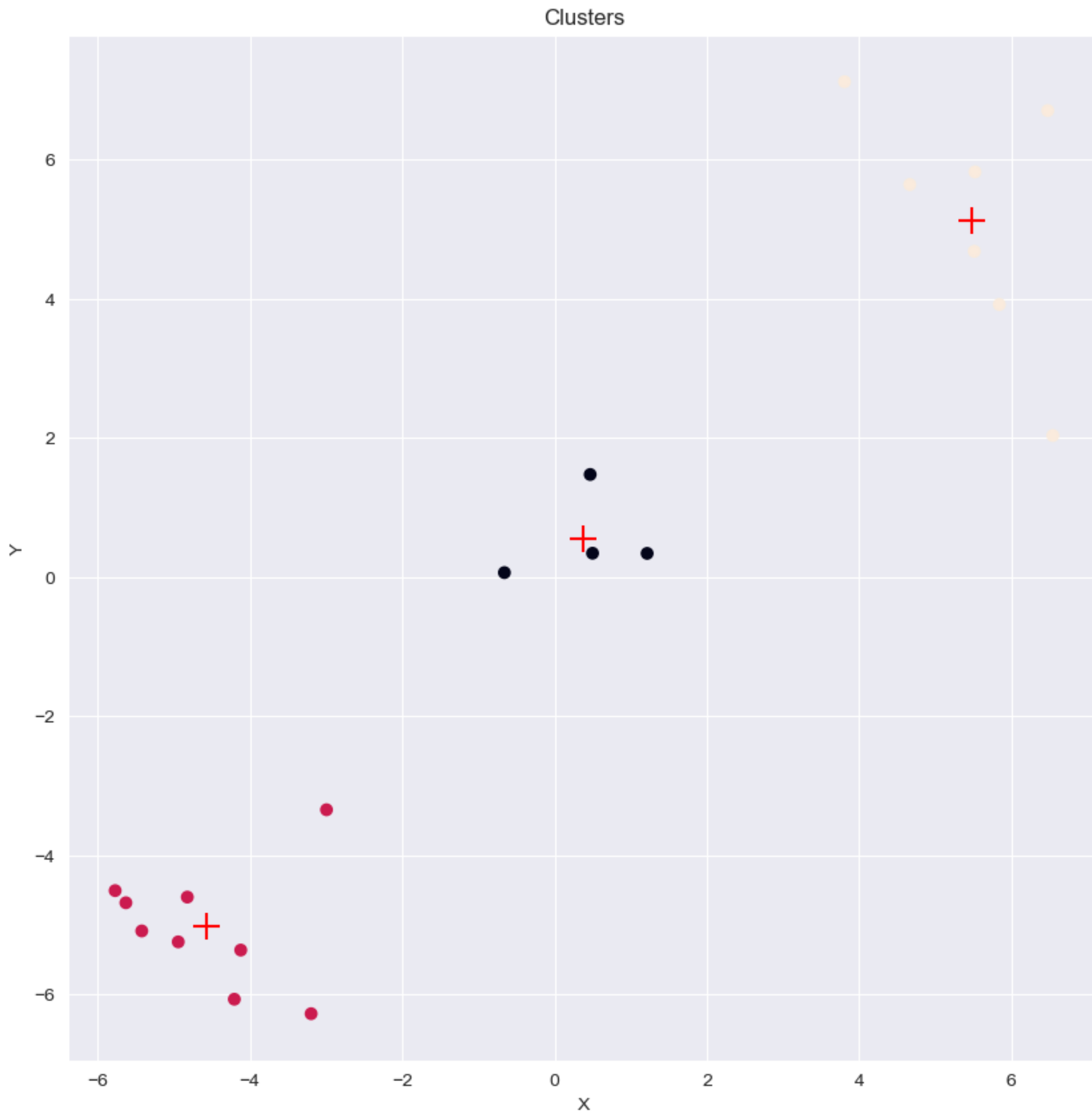
[12]:
```
# 2. Program the algorithm that consists in executing multiple K-Means with␣
↪different numbers of centers and displaying the best result.

from sklearn.cluster import KMeans


def range_kmeans(nb_cluster):
    # Run K-Means with different numbers of centers
    inertia = []
    K = [i for i in range(1, nb_cluster + 1)]
    for k in K:
        kmeans_fct = KMeans(n_clusters=k, n_init=10)
```

```
        kmeans_fct.fit(points)
        inertia.append(kmeans_fct.inertia_)

    # Plot the results
    plt.figure(figsize=(10, 10))
    plt.plot(K, inertia, 'x-')
    plt.xlabel('Number of clusters')
    plt.ylabel('Inertie')
    plt.title('Inertia vs. Number of clusters')
    plt.show()


range_kmeans(10)
```



We note that the optimal number of clusters is 3. Because the inertia decreases significantly from 1 to 3, then it decreases slightly from 3 to 10.

[13]:
```
# 3. Show the best result

nb_best_cluster = 3
kmeans = KMeans(n_clusters=nb_best_cluster, n_init=10)
kmeans.fit(points)

# Plot the points
plt.figure(figsize=(10, 10))
plt.scatter(points[:, 0], points[:, 1], c=kmeans.labels_)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='r',␣
 ↪marker='+', s=200)
plt.title("Clusters")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```



3 Exercise #3: Reinforcement Learning

In the problem of the arrows, n arrows are positioned vertically and oriented upwards or downwards. We want to swap the orientation of each arrow (see the figures below).

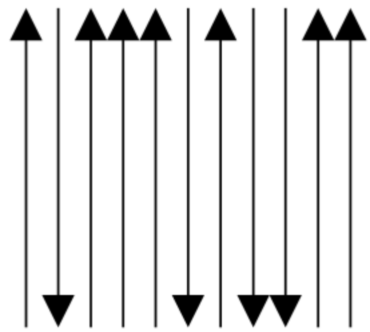


Figure 1a. Situation initiale

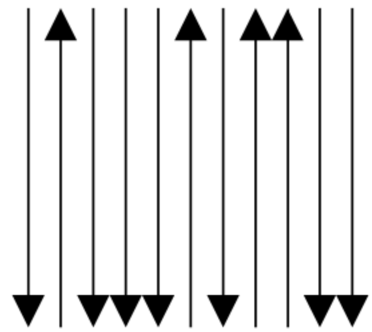


Figure 1b. Situation finale

We can modify a position with the two following cases: - by swapping a sequence of three adjacent arrows having the same orientation (all upwards or all downwards) - by swapping two adjacent arrows having opposite orientations (one is upwards and the other is downwards). Questions: 1. Explain how you can define the MDP for this problem. Describe as clearly as possible how do you represent a state. 2. Program the environment using the template available on Moodle. You can modify it as you wish. 3. Give the optimal policy obtained by Q-Iteration for the initial state above.

1. MDP Definition

We need to define the following elements (States, Actions, Rewards, Transitions, Termination Criterion): **States:**

A state can be represented by an array with the following letters `+1` when the arrow points up or `-1` when the arrow points down, where each element represents an arrow oriented upwards ('↑') or downwards ('↓'). For example, the initial state of our problem can be represented by [1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1].

Actions:

The two types of actions are: - Inverting a sequence of three consecutive arrows of the same orientation. - Inverting two consecutive arrows of opposite orientations.

Transitions:

State transitions are deterministic in this case, as one is either up or down (no in-between possible). Each action leads to a new state in a predictable manner, if down → up, if up → down.

Rewards:

The reward is used to encourage reaching the state where all arrows are oriented in the opposite direction. We can define the reward as follows: - -1 for each action - 0 for the final state or more

Termination Criterion:

The game ends when all arrows have been turned in the opposite direction.

[14]:
```
# 2. Program the environment using the template available on Moodle. You can modify it as you wish.
```

```
import itertools


class ArrowProblem:
    def __init__(self, n_arrows):
        self.n_arrows = n_arrows
        self.states = self.generate_states()

    def generate_states(self):
        return list(itertools.product([-1, 1], repeat=self.n_arrows))

    def state_space(self):
        return self.states

    def action_space(self, s):
        actions = []
        for i in range(self.n_arrows - 1):
            if s[i] != s[i + 1]:  # Reverse two adjacent arrows in opposite␣
↪directions
                actions.append((i, 2))
        for i in range(self.n_arrows - 2):
            if s[i] == s[i + 1] == s[i + 2]:  # Reverse three adjacent arrows␣
↪with the same orientation
                actions.append((i, 3))
        return actions

    def T(self, s, a):
        new_state = list(s)
        if a[1] == 2:  # Reverse two arrows
            new_state[a[0]] = -s[a[0]]
            new_state[a[0] + 1] = -s[a[0] + 1]
        elif a[1] == 3:  # Reverse three arrows
            new_state[a[0]] = -s[a[0]]
            new_state[a[0] + 1] = -s[a[0] + 1]
            new_state[a[0] + 2] = -s[a[0] + 2]
        return tuple(new_state)

    def R(self, s, a):
        return -1  # Simple -1 reward for each action
```

[15]:
```
# 3. Give the optimal policy obtained by Q-Iteration for the initial state above.

import numpy as np
import copy
```

```
class QIteration:
    def __init__(self, mdp, gamma=0.9, precision=1e-13):
        self.MDP = mdp
        self.gamma = gamma
        self.precision = precision
        self.Q = dict()

        nba = 0
        for s in self.MDP.state_space():
            nba = max(nba, len(self.MDP.action_space(s)))
        for s in self.MDP.state_space():
            self.Q[s] = np.zeros(nba)

    def run(self, N=-1):
        l = 0
        while l != N:
            l += 1
            oldQ = copy.deepcopy(self.Q)

            for s in self.MDP.state_space():
                for a_idx, a in enumerate(self.MDP.action_space(s)):
                    next_state = self.MDP.T(s, a)
                    self.Q[s][a_idx] = self.MDP.R(s, a) + self.gamma * np.
↪max(self.Q[next_state])

            if N == -1 and all(np.linalg.norm(oldQ[s] - self.Q[s]) < self.
↪precision for s in self.MDP.state_space()):
                print('QIteration has stopped after', l, 'iterations!')
                break

    def find_optimal_policy(self):
        policy = {}
        for s in self.MDP.state_space():
            possible_actions = self.MDP.action_space(s)
            if possible_actions:  # Vérifier s'il y a des actions possibles
                q_values = [self.Q[s][i] for i, _ in enumerate(possible_actions)]
                a_idx = np.argmax(q_values)
                policy[s] = possible_actions[a_idx]
            else:
                policy[s] = None
        return policy
```

[16]:
```
# Create an instance of the arrow problem
arrow_problem = ArrowProblem(n_arrows=3)

# Create an instance of Q-Iteration
q_iteration = QIteration(arrow_problem)
```

```
# Run the Q-Iteration
q_iteration.run()

# Find the optimal policy
optimal_policy = q_iteration.find_optimal_policy()

# Print the optimal policy for some states
for state in arrow_problem.state_space()[:5]:
    print(f"État: {state}, Meilleure action: {optimal_policy[state]}")
```

```
QIteration has stopped after 3 iterations!
État: (-1, -1, -1), Meilleure action: (0, 3)
État: (-1, -1, 1), Meilleure action: (1, 2)
État: (-1, 1, -1), Meilleure action: (0, 2)
État: (-1, 1, 1), Meilleure action: (0, 2)
État: (1, -1, -1), Meilleure action: (0, 2)
```

1. (-1, -1, -1), Best Action: (0, 3): State: All arrows pointing down (-1). Action: (0, 3) means to start inverting arrows from index 0 and affect 3 consecutive arrows. In this state, it would flip all arrows to point upwards.
2. (-1, -1, 1), Best Action: (1, 2): State: The first two arrows point down, and the last one points up. Action: (1, 2) means to start inverting arrows from index 1 and affect 2 consecutive arrows. This would flip the second and third arrow.
3. (-1, 1, -1), Best Action: (0, 2): State: The first arrow points down, the second up, and the third down. Action: (0, 2) indicates inverting the first two adjacent arrows that have opposite orientations.
4. (-1, 1, 1), Best Action: (0, 2): State: The first arrow points down, the other two point up. Action: (0, 2) again indicates inverting the first two adjacent arrows with opposite orientations.
5. (1, -1, -1), Best Action: (1, 2): State: The first arrow points up, the other two point down. Action: (1, 2) indicates inverting the second and third arrows.

[17]:
```
# For our initial problem with 11 arrows, we can display the optimal policy for␣
↪each state:

arrow_problem = ArrowProblem(n_arrows=11)

# Run the Q-Iteration
q_iteration = QIteration(arrow_problem)

# Run the Q-Iteration
q_iteration.run()

# Find the optimal policy
optimal_policy = q_iteration.find_optimal_policy()

# Print the optimal policy for some states
for state in arrow_problem.state_space()[:5]:
```

```
    print(f"État: {state}, Meilleure action: {optimal_policy[state]}")
```

```
QIteration has stopped after 3 iterations!
État: (-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1), Meilleure action: (0, 3)
État: (-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1), Meilleure action: (9, 2)
État: (-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, -1), Meilleure action: (8, 2)
État: (-1, -1, -1, -1, -1, -1, -1, -1, -1, 1, 1), Meilleure action: (8, 2)
État: (-1, -1, -1, -1, -1, -1, -1, -1, 1, -1, -1), Meilleure action: (7, 2)
```

[18]:
```
# Find the optimal policy for the initial state represented by [1, -1, 1, 1, 1,␣
↪-1, 1, -1, -1, -1, 1, 1].

print(f"État initial : [1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1]")
print(f"Meilleure action : {optimal_policy[(1, -1, 1, 1, 1, -1, 1, -1, -1, 1,␣
↪1)]}")
```

```
État initial : [1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1]
Meilleure action : (0, 2)
```

We perform the action recommended by the Q-Iteration algorithm and look for the optimal policy for the following state: [1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1, 1] → [1, 1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1] ETC.