Numerical optimization for large scale problems
Stochastic optimization

Assignment 1: Neural networks

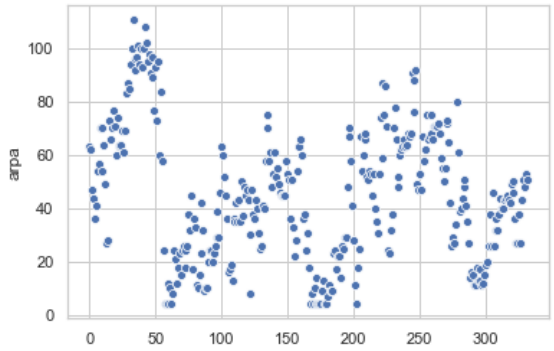
Giammarinaro Silvia, Gullotto Marco

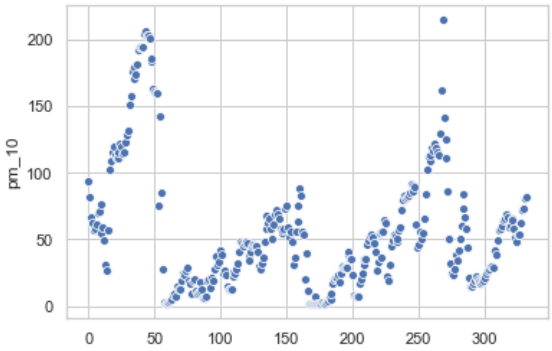
January 21, 2020

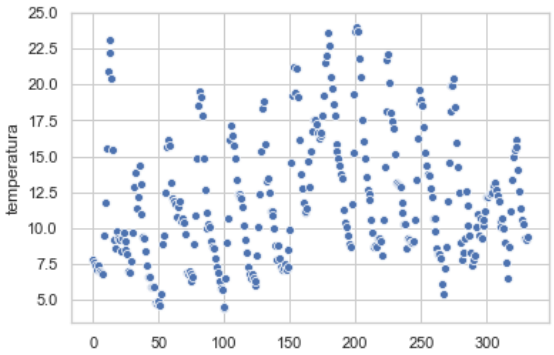
1 Data introduction

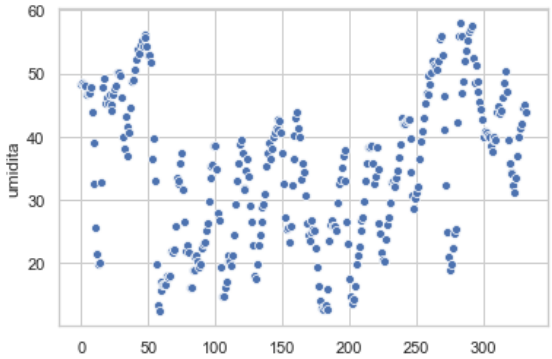
Let's start with a brief introduction, talking about the factual problem; then straight on with several analysis of the data, plotting different kind of graphs; finally, two Regression methods: the first one is a simple basic regression method, the other one involves neural network. The main goal of this research is to show how is possible to replace the 10000$ equipment used by ARPA in monitoring the amount of particulate matter (PM) in the air of city, with a mere 20$ tool. The DAUIN department of Politecnico tried to reach this goal by installing cheap sensors all over the city: they control values of PM, temperature, humidity and atmospheric pressure in real time. The question is: by acquiring these information, is it possible to make a forecast about the evolution of the volume of PM in air, with the same accuracy (or, at least, a comparable one) of the ARPA's PM sensors? To begin to understand the phenomenon it is necessary to realize, first, some plots of the evolution of measurements over time. Therefore, after uploading the part of the data relating to the period between August and November 2018, the production of some scatter plots is carried out. The aforementioned scatter plots allow to evaluate the trend of the measurements and to understand if there is a correlation between them and those which are made by ARPA.

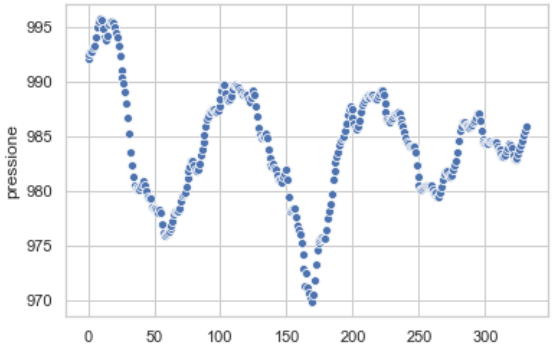
```
ds = pd.read_csv('Dati_arpa.csv', sep=";")
sns.scatterplot(np.arange(len(ds['arpa'])), ds['arpa'])
sns.scatterplot(np.arange(len(ds['pm_10'])), ds['pm_10'])
sns.scatterplot(np.arange(len(ds['temperatura'])), ds['temperatura'])
sns.scatterplot(np.arange(len(ds['umidita'])), ds['umidita'])
sns.scatterplot(np.arange(len(ds['pressione'])), ds['pressione'])
```











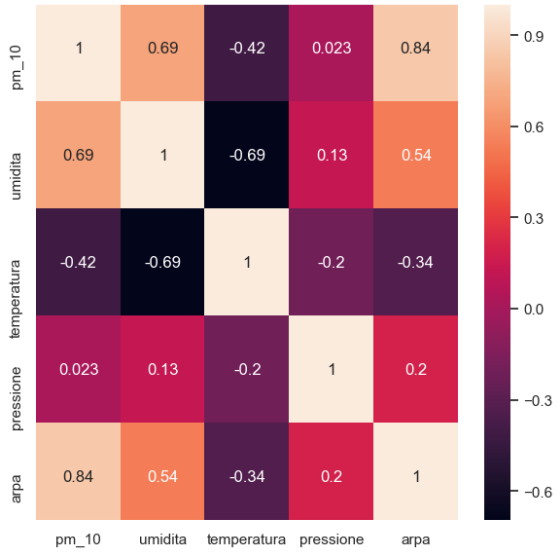
2 Data correlation

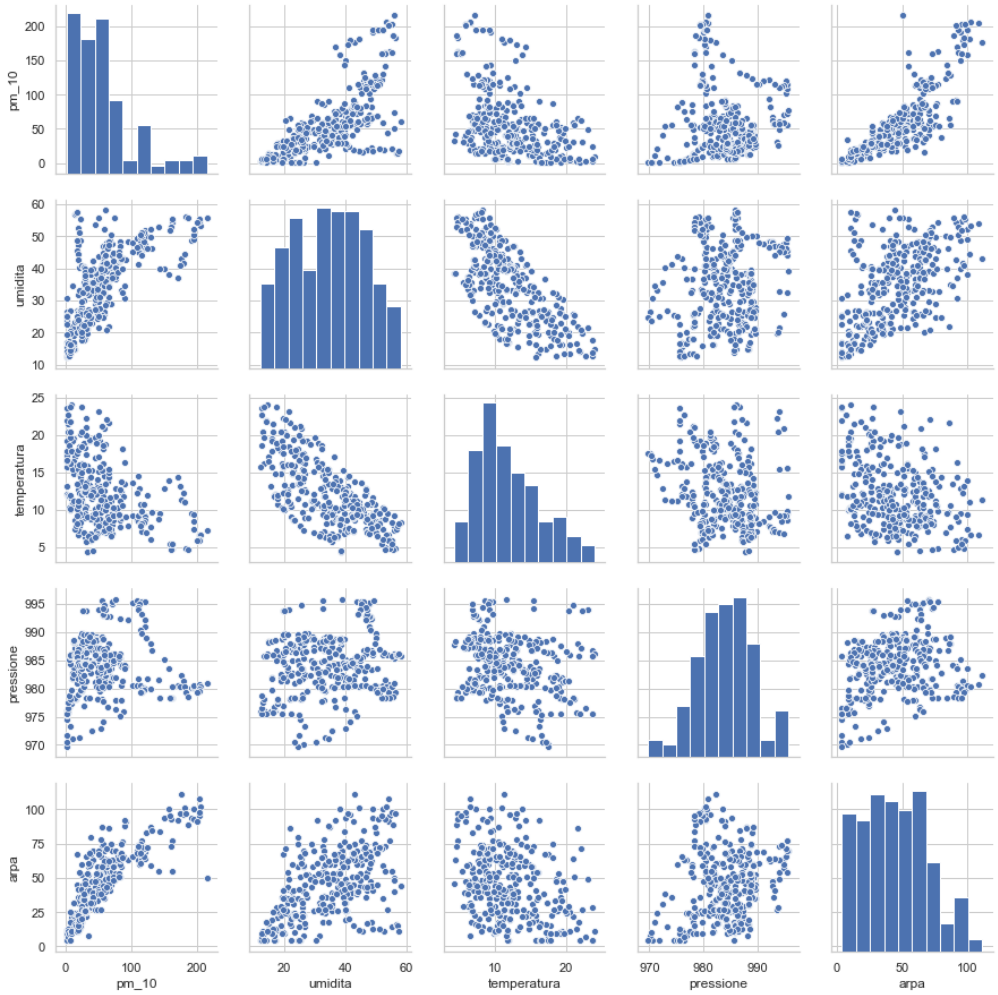
The first plots give a rough idea of the phenomenon that occurs, but to have a better overview, it is necessary to investigate the existence of a possible correlation between our data and the kind of distribution they have. It's possible to do it by plotting some boxplots that show if there are some outliers between the data.

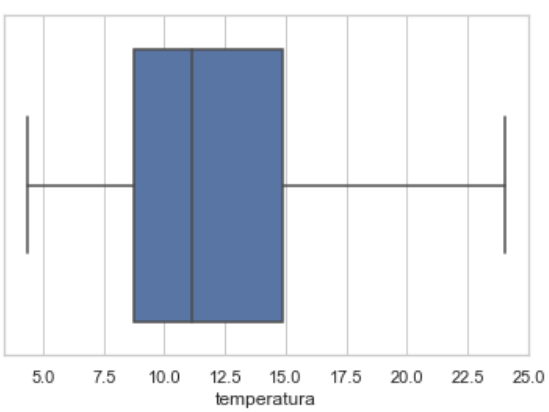
```
corr = ds.corr(method='pearson')
sns.heatmap(corr, annot=True)
sns.pairplot(ds)
sns.set(style="whitegrid")
```

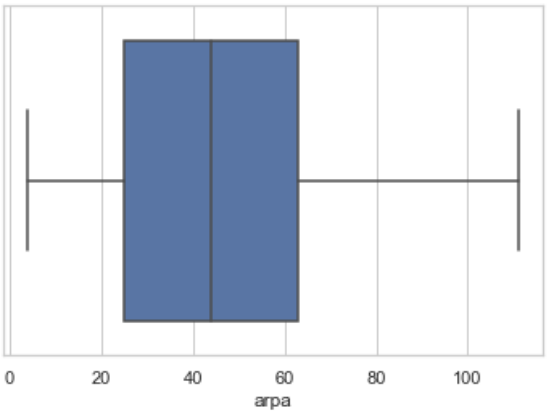
```
sns.boxplot(ds['temperatura'])
sns.boxplot(ds['arpa'])
sns.boxplot(ds['pm_10'])
sns.boxplot(ds['umidita'])
sns.boxplot(ds['pressione'])
```

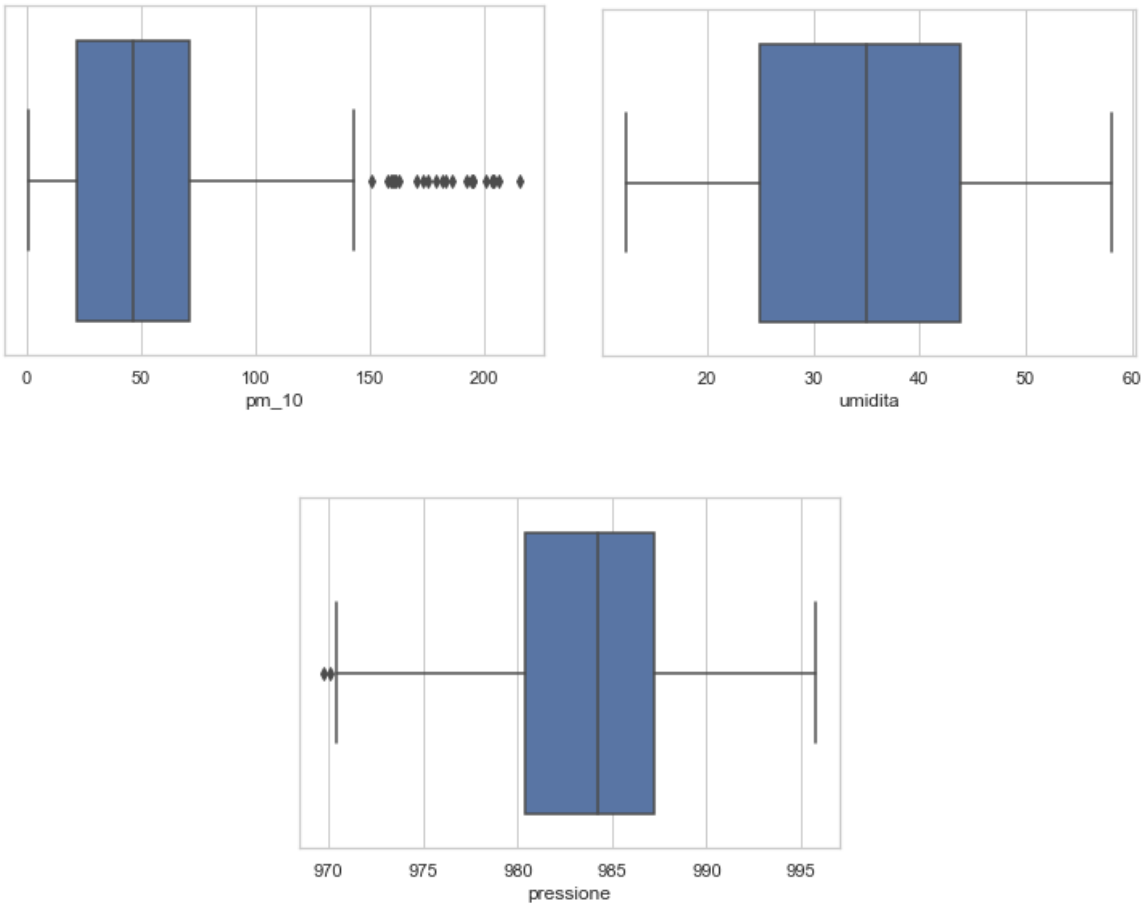
Analyzing the results obtained, it's impossible to not notice the strong correlation between Politecnico's PM10 sensors and ARPA's ones. However, factors such as air humidity and atmospheric pressure influence, although minimally, the measurements made by the Politecnico, and this aspect must be taken into consideration to obtain a better approximation. All the results are shown below.




pm_10
umidita
temperatura
pressione
arpa
0
100
200
20
40
60
10
20
970
980
990
0
50
100


5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
temperatura

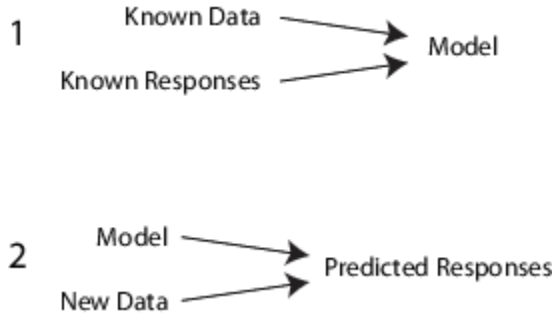

0
20
40
60
80
100
arpa



3 Least squares method

The initial approach to the problem involves the use of a **Supervised learning** algorithm. This term means:

> *methods which have the task to of learning a function that maps an input to an output based on example input-output pairs.*
>
> Stuart J. Russell, Peter Norvig (2010) Artificial Intelligence: A Modern Approach



This method represents the baseline result for next analysis. The basic idea behind the last squares is very easy: we have some *ith* inputs and we denote this quantity as x_i , and for each input we

want a corresponding output $\hat{y}_i$. Our aim is to find a value c_i so that:

$$\hat{y}_i = \sum_i c_i x_i$$

If we denote by y_i the ground-truth we want to minimize the error function:

$$e_i^2 = \sum_i (y_i - \sum_j c_{ij} x_{ij})^2$$

To obtain the values of c which minimize the error function, we have to derive for c_k and then equal the result to 0. The result of these tedious counts is the following:

$$c_k = \frac{\sum_i y_i x_{ik}}{\sum_i x_{ik}^2}$$

The numerator of this fraction is simply a vectorial product between a vector and a matrix, which returns a vector again. Instead, the denominator is a matrix but luckily it's easy to demonstrate that it's a symmetric positive definite matrix so we can solve this problem with a naive implementation of a (conjugate) gradient method. By this way, the nature of this matrix is fully exploited. The gradient method is an iterative method which starts from an initial point (i.e.) x_0 and every iteration (under suitable assumption) built a sequence of approximation which converges to the right solution x_k.

```
def conjugate_gradient_method(A, b, rip, x0):
    xk = x0
    rk = b - np.dot(A, xk)
    dk = rk
    for i in range(rip):
        zk = np.dot(A, dk)
        alpha = np.dot(rk.T, dk)/np.dot(dk.T, zk)
        xk = xk + alpha*dk
        rk_old = rk
        rk = rk - alpha*zk
        beta = - np.dot(rk.T, rk)/np.dot(rk_old.T, rk_old)
        dk = rk + beta*dk
    return xk
```

```
def gradient_method(A, b, rip, x0):
    x = np.zeros(A.shape[1])
    xk = x0
    rk = b - np.dot(A, xk)
    for i in range(rip):
        zk = np.dot(A, rk)
        alpha = np.dot(rk.T, rk)/(np.dot(rk.T, zk))
        xk = xk + alpha * rk
        rk = rk - alpha * zk
    return xk
```

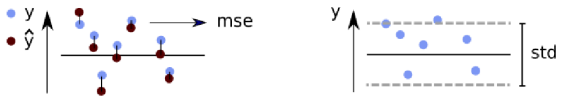
```
def least_sqare_method(x, y):
    numeroColonne = x.shape[1]
    numeroRighe = x.shape[0]
    vetty = np.dot(y, x).reshape(numeroColonne, 1)

    pa = np.array(np.zeros(numeroColonne ** 2)).reshape(numeroColonne,␣
 ↪numeroColonne)

    for i in range(numeroColonne):
        for j in range(numeroColonne):
            for k in range(numeroRighe):
                pa[i][j] += x[k][i] * x[k][j]

    return conjugate_gradient_method(pa, vetty, 10000, np.array([1,1,1,1]).
 ↪reshape(4,1))
```

The determination coefficient (called R^2) can be used to check the goodness of the result. R^2 describes the proportion of the variance in the dependent variable that is predictable from the independent variable(s). So, R^2 is defined by the following:

$$R^2 = 1 - \frac{MSE}{STD^2}$$

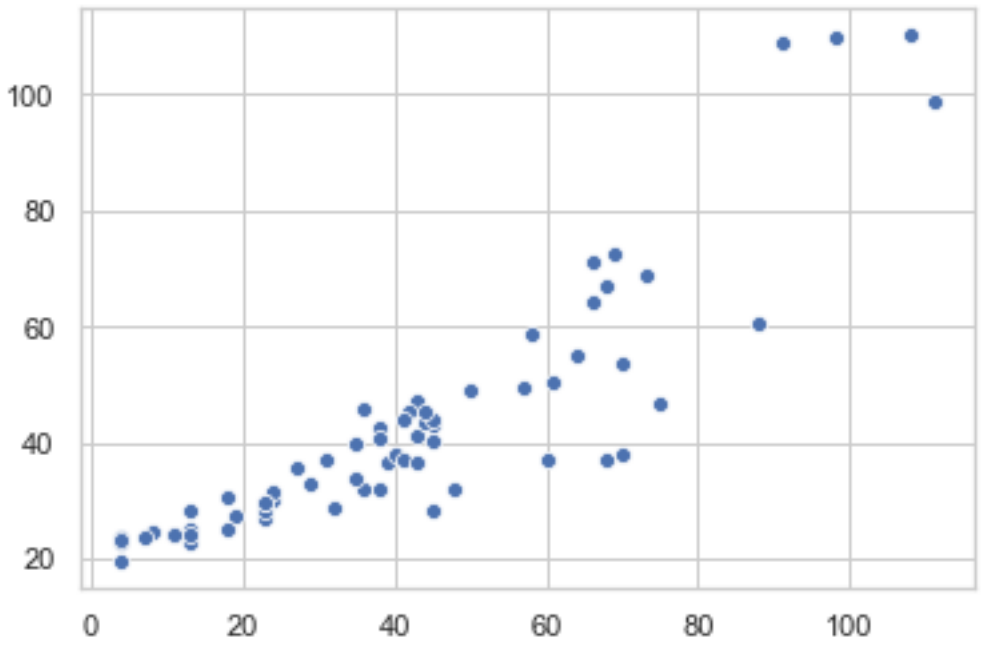
Whereas MSE represents the mean squared error and STD^2 is the variance of our sample, in formula:

$$MSE = \frac{1}{n}\sum_i (y_i - \hat{y}_i)^2$$




```
X_train, X_test, y_train, y_test = train_test_split(uns, y, test_size=0.2,␣
 ↪random_state=42)
ris = mean_sqare_method(X_train, y_train)
y_1 = np.dot(X_test, ris)
y_pred = y_1
print(r2_score(y_test,y_pred))
sns.scatterplot(y_test, y_pred.reshape(1, len(y_pred))[0])
```

```
R2: 0.7796309388114154 so this value will be our baseline.
```



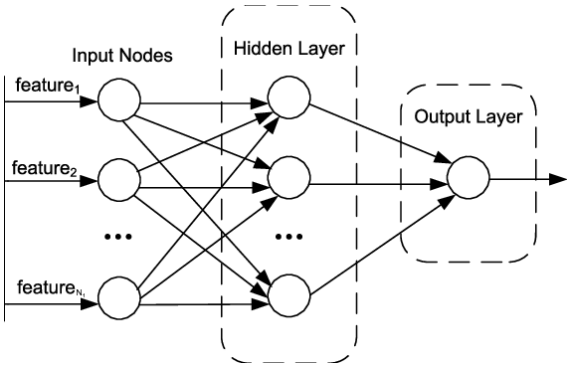
The x-axis of this plot represent the ground-truth values produced by the ARPA's sensors, instead in y-axis is situated the probability based on input values (PM10, humidity, temperature, atmospheric pressure).

4 Non-linear Neural Networks

Although a linear approach displays good result with a R^2 score close enough to 1, it may be appropriate to evaluate whether better performances are obtained through a non-linear approach. There are several ways in which it is possible to implement a non-linear regressor, but a backpropagation method seems what it is needed to solve this problem. Backpropagation needs for every possible entry tuple an exit value in order to compute the loss function. It is then minimized with a gradient-method as it's shown in the least-squared approach. This is the basic idea behind neural network (NN). These kind of networks are able to optimally recognize various types of patterns, and they are used in different fields such as regression. However, it is necessary to tune several hyperparameters to obtain an adequate result. NN are essentially based on nodes:

- Input nodes, which are in finite number, that usually equals the number of inputs of the function. Our problem, for example, needs four nodes, because we have PM10, humidity, temperature and atmospheric pressure as inputs. The set of all input nodes constitutes the so-called input layer.
- Hidden nodes make up that part of the NN that introduces a non-linearity into the model. They are organized in one or more layers. There are no fixed rules to select the correct number of hidden nodes or layers: only a "trial and error" approach can find an appropriate solution to problems (for our purpose, one hidden layer is enough but the problem of the number of nodes remains).
- Output node, which is one and only one for a regression task. NN can acquire more than

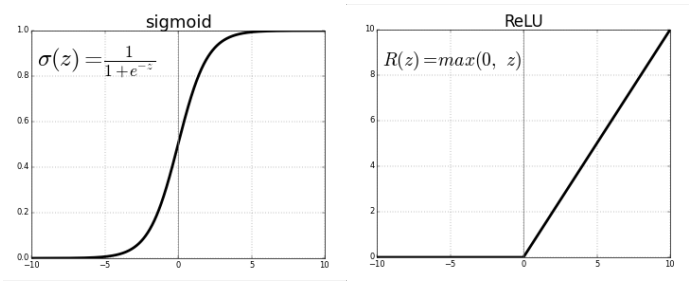
one output node, but in these cases they are used in other situations and fields which are not relevant in our situation.



It's important to emphasize (as shown in the image above) that every node of the input node is fully linked with the nodes of the hidden layer and, in turn, every node of hidden layer is connected with the output node. Each connection is only in one direction and it's associated with a scalar value called weight. From now on, I will denote by $w(i,h)$ the weight which joins the ith input layer with the hth hidden one. Indeed, I will denote by $x(h)$ the weight on the link from the hth hidden node to the output node. So, every node has an internally stored value: for the input layer, we have the input values of dataset, while for the hidden and the output node we can compute this value using w and x. we denote by $u(j)$ the value of the jth input node, we can compute the vector $\hat{v}$ as:

$$\hat{v}(j) = \sum_{j=1}^{I} w(i,j)u(j)$$

This doesn't exactly match the value of hidden nodes, because we have to do a sort of 'normalization' using a function where the result is a number in the range between $[0,1]$. There are different functions that can be used, but the most popular ones are the sigmoid and the ReLU function. These two are the only ones that work with the data we have available.



Computed the value v of each hidden node, the output value of the NN is given by:

$$o = \sum_{h=1}^{H} x(h)v(h)$$

At this point, the backpropagation algorithm is used to minimize the squared differences between actual function values and predicted values. At this point, if I denote by e_p the quantity:

$$e_p = y_p - o_p$$

I can determinate the normalized mean sum of squared error (SSE) as:

$$SSE_p = \sum_{p=1}^{n} (y_p - o_p)^2$$

At last, the backpropagation algorithm, like any other steepest-descent method, minimize $SSE/2$. If $SSE/2$ is minimized, SSE will be minimized too. Finally, we can update the values of w and x as:

$$w(i,h) \leftarrow w(i,h) + \mu \sum_{p=1}^{n} (y_p - o_p) x(h) v_p(h)(1 - v_p(h)) u(i)$$

$$x(h) \leftarrow x(h) + \mu \sum_{p=1}^{n} (y_p - o_p) v_p(h)$$

Where μ is a scalar value also called **learning rate**. This value is one of the most important in all the NN. The hyperparameter μ can heavily influence convergence and the speed of our network. This process is repeated a number of times which is called **Epochs** and this parameter is also chosen by the user. If the number of epochs is too large, the model can have the so-called **overfitting** problem. To avoid this problem it's possible to choose a tolerance: in this way the algorithm only stops when the difference:

$$SSE_{new} - SSE_{old} <= tolerance$$

Before starting to implement the NN, it's important to emphasize that the values of w and x must be chosen in a smart way. A recommended way to initialize that, is to choose values from a uniform distribution $U(0, 0.1)$.

At this point, everything is almost ready: since the number of input and output nodes are fixed, the only problem is to decide a 'correct' number of hidden nodes and a suitable value for μ. Starting with choosing hidden nodes, there are two problems:

- As NN use a steepest-descent algorithm, a well-known problem arises: they can only guarantee convergence to a local minimum. If there is more than one local minimum, this could means that the solution found is incorrect.
- The use of a large number of hidden nodes can avoid the problem of multiple optimal, but this means a huge computational effort to compute both w and x and can cause an overfitting problem.

Therefore, a balance between the number of hidden nodes and the performance of the network must be found. The only possibility is to make attempts, and see the results. I start with a reduced number of hidden node (one or two), but the SSE start to converge to local minima, which is not the result that I hope to find out, so I decide to increase the number of nodes in the hidden layers. With ten nodes it begins to converge towards the correct solution, and the algorithm still remains very fast, so I go on with the increment phase. Reaching forty nodes, the margin of error is reduced, but the algorithm works very slowly. So a good balance is to chose seventeen nodes. The last hyperparameter to choose is the value of μ. This is a crucial parameter, as said before, because it can generate another well-known problem called **gradient explosion**. This phenomenon produce large values of $\hat{v}$ and this means that the weights w and x are very close to 1. This implies that for wide values of weights the network lose its discriminatory power. Many books suggest to update the value of μ at the each iteration in the following way:

$\mu = log(m)/m$ where m is the number of iteration

This is a very nice sequence which has some well-known conditions as:

$$\sum_m \mu_m = \infty$$

$$\sum_m (\mu_m)^2 < \infty$$

but the problem is that this sequence, at the beginning, has too many high numbers and the NN diverge immediately. Numerous different solutions can be implemented, but the best one is to keep the μ fixed to a very small value. This is the only solution that permits to the NN to converge from the beginning and maintain the values of w and x far from 1. Last but not least, it's important to normalize or standardize the input values. If we skip this step, the result obtained will cause a w and a x very close to one, as in the case previously mentioned. Initially, I decide to normalize the input values (which I denote with u) so, if the minimum possible value for the ith input is a(i) and the maximum is b(i):

$$u(i) = \frac{u_{raw}(i) - a(i)}{b(i) - a(i)}$$

With this method I need a lot of Epochs to reach a good value of R^2 , but after 1000 iterations the score is the best I can reach. Instead, if I standardize the input values in the following way:

$$u(i) = \frac{u_r(i) - \mu}{\sigma}$$

the algorithm is able to quickly reach a good R^2 score, but after 100 iterations I have no further improvements. All the implementations I have presented until this moment, are reported in the following code:

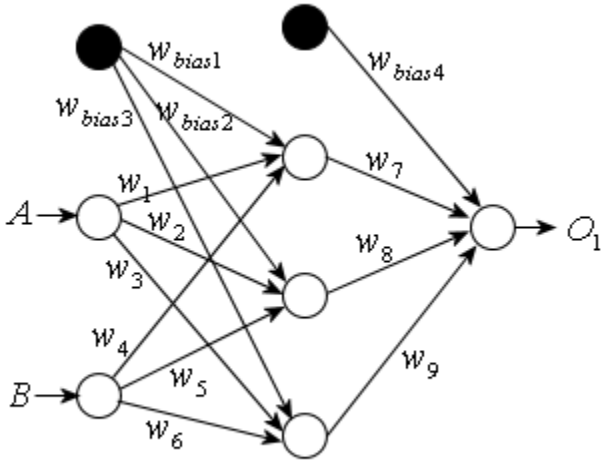
```
class NeuralNetwork():

    def __init__(self, shape_w, shape_x, learning_rate):
        self.shape_w = shape_w
        self.shape_x = shape_x
        self.w = np.random.uniform(0,0.1,(shape_w[0], shape_w[1]))
        self.x = np.random.uniform(0,0.1,(shape_x[0]))
        self.b = 0.5
        self.H = shape_w[1]
        self.I = shape_w[0]
        self.vb =  np.random.uniform(0,0.1,H)
        self.learning_rate = learning_rate


    def sigmoid(self, x):
        return scipy.special.expit(x)

    def ReLU(self,x):
        return x * (x > 0)

    def train(self, training_inputs, training_outputs, training_iterations,tol,␣
↪verbose = False):
        m = 1
        mu = self.learning_rate
        SSEold = 10000
```

```
        for iteration in range(training_iterations):
            v_star = np.dot(training_inputs, self.w)
           # v_star += self.vb

            v = self.sigmoid(v_star+self.vb)
            output = self.b + np.dot(v, self.x)
            output_error = training_outputs-output

            #mu *= 0.9999

            self.b += mu*output_error.sum()
            hidden_deltas = np.zeros((training_inputs.shape[0], self.H))
             #compute hidden deltas
            for h in range(self.H):
                for p in range(training_inputs.shape[0]):
                    hidden_deltas[p, h]=output_error[p]*self.x[h]


            for i in range(self.I):
                for h in range(self.H):
                    for p in range(training_inputs.shape[0]):
                        self.w[i,h] += mu*output_error[p]*self.
↪x[h]*v[p,h]*(1-v[p,h])*training_inputs[p,i]

            for h in range (self.H):
                change=0;
                for p in range(training_inputs.shape[0]):
                    change=change+hidden_deltas[p][h];

                self.vb[h]=self.vb[h]+(change*self.learning_rate);



            appo = np.dot(v.T, output_error)
            self.x += mu * appo

            SSEnew = ((output_error)**2).sum()
            if verbose == 1:
                print(m, SSEnew)
            if abs(SSEnew-SSEold) < tol:
                return
            else:
                if(math.isnan(SSEnew)):
                    return
            SSEold = SSEnew
```

```
            m+=1

    def think(self, inputs):
        v_star = np.dot(inputs, self.w)
        v = self.sigmoid(v_star+self.vb)
        output = np.dot(v, self.x) + self.b
        return output

    def save_model(self, path_file):
        with open(path_file, 'w') as file:
            file.write(str(neural_network.w).replace("\n", " ").replace("]", "␣
↪").replace("[", " ")+"\n")
            file.write(str(neural_network.x).replace("\n", " ").replace("]", "␣
↪").replace("[", " ")+"\n")
            file.write(str(neural_network.b).replace("\n", " ").replace("]", "␣
↪").replace("[", " ")+"\n")

    def load_model(self, path_file):
        with open(path_file, 'r') as file:
            filecontents = file.readlines()
            appo = list(str(filecontents[0]).split(" "))
            appo.remove("\n")
            vett = [val for val in appo  if val != '' ]
            self.w = np.array(vett, dtype = float).reshape(self.shape_w)
            appo = list(str(filecontents[1]).split(" "))
            appo.remove("\n")
            vett = [val for val in appo if val != '' ]
            self.x = np.array(vett, dtype = float).reshape(self.shape_x)
            self.b = float(filecontents[2])
```

5 Bias impact

In the aforementioned implementation of NN, I add new types of nodes called "bias nodes". One of these is connected directly to the output node, the other has the same behavior of an input node. This is clearly shown in the following image:



The first bias assume the same function of a constant term, as we can see from the update of the output node:

$$o(h) = b + \sum_{h=1}^{H} x(h)v(h)$$

This bias node is initialized at constant value, which is not so influential if we choose a small value such as 0.5. After that, at every iteration, we update the bias weight using:

$$b \leftarrow b + \mu \sum_{p=1}^{n} (y_p - o_p)$$

In this way the NN are able to provide a better result in term of R^2 in a shorter time. Introducing the second type of bias (which is also called "virtual bias node") NN can definitively improve, in a substantial way, the correctness of the result and the stability of the network, but I have to pay a larger computational effort. In practice, I have to increase the number of the nodes from 17 to 50, otherwise the network doesn't converge anymore but, as said before, the time to get the result increase significantly. To get, at each iteration, the weight of the virtual bias node, I need to compute:

$$\Delta[p,h] = \sum_{p=1}^{n} (y_p - o_p)x(h)$$
$$vb[h] \leftarrow vb[h] + \mu \sum_p \Delta[p,h]$$

For the initialization of this type of nodes I adopt the same strategy of the input nodes because they have a very similar behavior. So they are initialized to random numbers which are $U[0, 0.1]$ as before.

6 Predict and remember

The function "think" accepts one or more vectors as input, and predict the result starting from the weights it computes during the training phases. This method is really fast because all the operations are very simple for the CPU but, as I said before, training takes a lot of time and CPU

elaboration. So, to split the work into different moments, I also implement a "save_model" and a "load_model" function. In this way, I can store the weights of my model temporarily in a file stored in the solid disk and load it up again when I reboot the program.

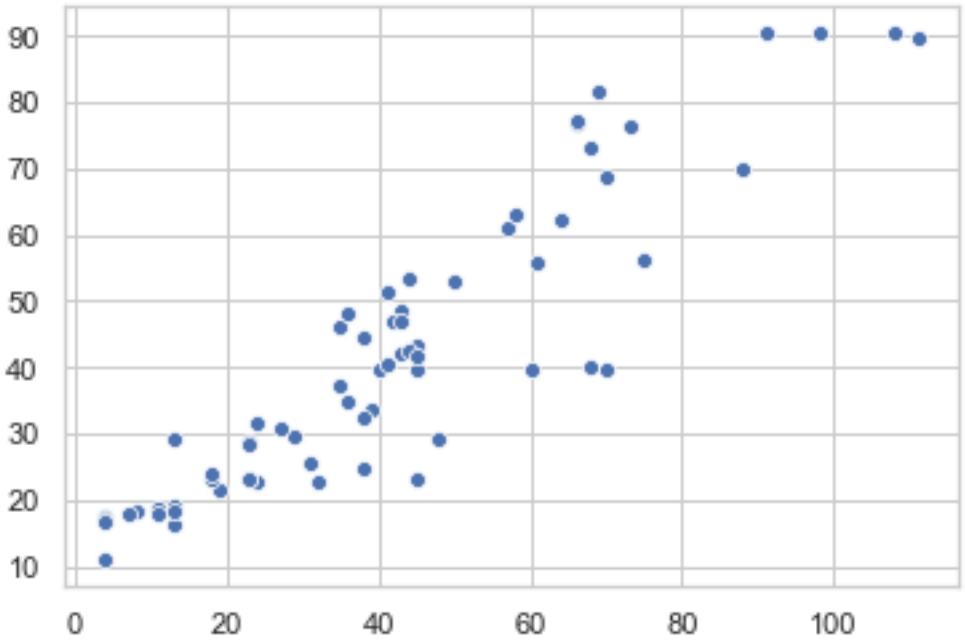
7 Result

In regards to the least squared method, I split my dataset into two sections: one contains the 80% of the data, the other one the remaining part. After that, I train my model on the larger dataset and I make the prediction on the other set. The result is impressive: in fact there is a great improvement of the R^2. Finally, I plot the ARPA values on the x axis and my prediction on the y axis. As it can be seen, the result is more linear than the one obtained with the least squared method. The only drawback is that with a NN we lost information about the weight applied on each input nodes. Unluckily, this is a well-known problem in the NN field, and many researcher are trying to fix this problem but the solution is not yet at hand.

```
X_train, X_test, y_train, y_test = train_test_split(u, y, test_size=0.2,␣
 ↪random_state=42)
shape_w = (I,H)
shape_x = (H,)

neural_network = NeuralNetwork(shape_w, shape_x, 0.0001)



neural_network.train(X_train, y_train, 120, 1e-1)

print("Synaptic weights after training: ")
print(neural_network.w)
print(neural_network.x)
print(neural_network.b)
```

```
y_pred = neural_network.think(X_test)
print(r2_score(y_test,y_pred))
sns.scatterplot(y_test, y_pred)
```

```
R2: 0.8309453423614469 better than before
```



8 Further improvements

Further improvements are implemented using the function "PolynomialFeatures" of the SKlearn library. This method

> Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.

From the sklearn documentation.

```
poly = PolynomialFeatures(2)
X_train2 = poly.fit_transform(X_train)
X_test2 = poly.fit_transform(X_test)
shape_w = (len(X_train2[0]),H)
shape_x = (H,)

neural_network = NeuralNetwork(shape_w, shape_x, 0.0001)
neural_network.train(X_train2, y_train, 100, 1e-1)
y_pred = neural_network.think(X_test2)
print(r2_score(y_test,y_pred))
sns.scatterplot(y_test, y_pred)
```

```
R2: 0.8517054963998268
```

As we can see, with a degree of two I obtain a better R^2 and also the usual graph ground-truth prediction shows that we obtain a result that is closer to linearity.