mlp

April 26, 2022

1 Multi-Layer Perceptron

1.1 Introduction

The objective of this lab is to dive into particular kind of neural network: the *Multi-Layer Perceptron* (MLP).

To start, let us take the dataset from the previous lab (hydrodynamics of sailing boats) and use scikit-learn to train a MLP instead of our hand-made single perceptron. The code below is already complete and is meant to give you an idea of how to construct an MLP with scikit-learn. You can execute it, taking the time to understand the idea behind each cell.

[69]:
```
# Importing the dataset
import numpy as np
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1]
Y = dataset[:, -1]
```

[70]:
```
# Preprocessing: scale input data
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
```

[71]:
```
# Split dataset into training and test set
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y,random_state=1,␣
↪test_size = 0.20)
```

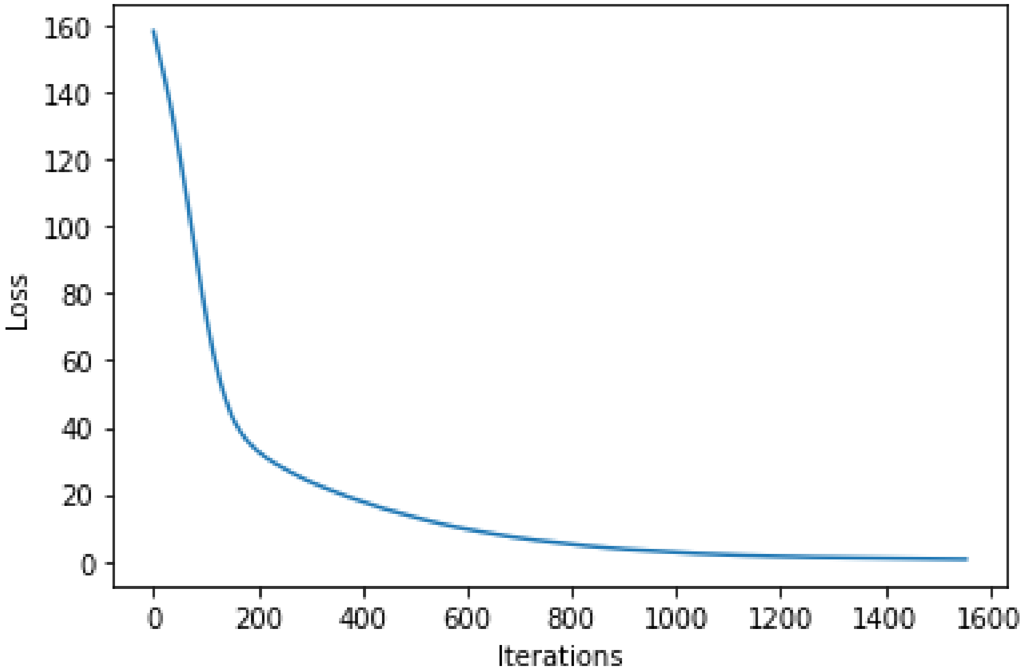
[72]:
```
# Define a multi-layer perceptron (MLP) network for regression
from sklearn.neural_network import MLPRegressor
mlp = MLPRegressor(max_iter=3000, random_state=1) # define the model, with␣
↪default params
mlp.fit(x_train, y_train) # train the MLP
```

[72]:
```
MLPRegressor(max_iter=3000, random_state=1)
```

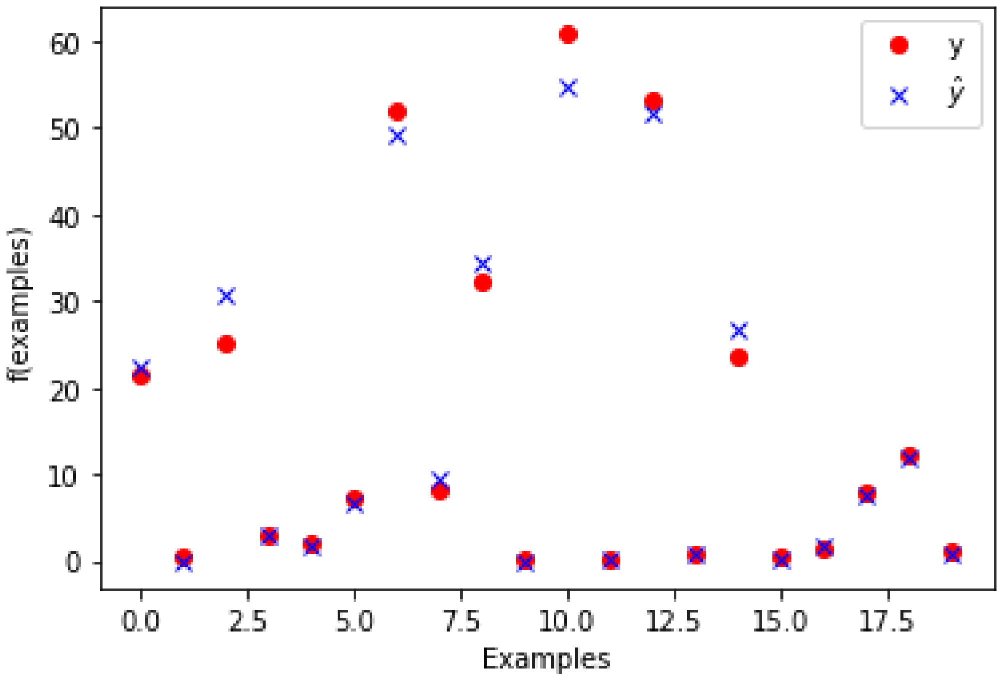
[73]:
```
# Evaluate the model
from matplotlib import pyplot as plt
```

```
print('Train score: ', mlp.score(x_train, y_train))
print('Test score:  ', mlp.score(x_test, y_test))
plt.plot(mlp.loss_curve_)
plt.xlabel("Iterations")
plt.ylabel("Loss")
plt.show()
```

```
Train score:  0.9940765369322633
Test score:   0.9899773031580283
```



[74]:
```
# Plot the results
num_samples_to_plot = 20
plt.plot(y_test[0:num_samples_to_plot], 'ro', label='y')
yw = mlp.predict(x_test)
plt.plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')
plt.legend()
plt.xlabel("Examples")
plt.ylabel("f(examples)")
plt.show()
```



1.1.1 Analyzing the network

Many details of the network are currently hidden as default parameters.

Using the documentation of the MLPRegressor, answer the following questions.

- What is the structure of the network?
- What it is the algorithm used for training? Is there algorithm available that we mentioned during the courses?
- How does the training algorithm decides to stop the training?

What is the structure of the network? * 3 layers, the hidden layer has 100 percetrons

What it is the algorithm used for training? Is there algorithm available that we mentioned during the courses? * The default value 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

How does the training algorithm decides to stop the training? * It stops the training after `max_iter` itterations are done, there is no early stopping by default.

1.2 Onto a more challenging dataset: house prices

For the rest of this lab, we will use the (more challenging) California Housing Prices dataset.

[75]:
```
# clean all previously defined variables for the sailing boats
%reset -f

# Import the required modules
```

```python
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import time
import copy
```

[76]:

```python
num_samples = 3000 # only use the first N samples to limit training time

cal_housing = fetch_california_housing()
X = pd.DataFrame(cal_housing.data,columns=cal_housing.feature_names)[:
 ↪num_samples]
Y = cal_housing.target[:num_samples]

X.head(10) # print the first 10 values
```

[76]:

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85
5  4.0368      52.0  4.761658   1.103627       413.0  2.139896     37.85
6  3.6591      52.0  4.931907   0.951362      1094.0  2.128405     37.84
7  3.1200      52.0  4.797527   1.061824      1157.0  1.788253     37.84
8  2.0804      42.0  4.294118   1.117647      1206.0  2.026891     37.84
9  3.6912      52.0  4.970588   0.990196      1551.0  2.172269     37.84

   Longitude
0    -122.23
1    -122.22
2    -122.24
3    -122.25
4    -122.25
5    -122.25
6    -122.25
7    -122.25
8    -122.26
9    -122.25
```

Note that each row of the dataset represents a **group of houses** (one district). The `target` variable denotes the average house value in units of 100.000 USD. Median Income is per 10.000 USD.

1.2.1 Extracting a subpart of the dataset for testing

- Split the dataset between a training set (75%) and a test set (25%)

Please use the conventional names `X_train`, `X_test`, `y_train` and `y_test`.

[77]:
```
X_train, X_test, Y_train,Y_test = train_test_split(X, Y,random_state=1,␣
↪test_size = 0.25)
```

1.2.2 Scaling the input data

A step of **scaling** of the data is often useful to ensure that all input data centered on 0 and with a fixed variance.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance). The function `StandardScaler` from `sklearn.preprocessing` computes the standard score of a sample as:

`z = (x - u) / s`

where `u` is the mean of the training samples, and `s` is the standard deviation of the training samples.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using transform.

- Apply the standard scaler to both the training dataset (`X_train`) and the test dataset (`X_test`).
- Make sure that **exactly the same transformation** is applied to both datasets.

Documentation of standard scaler in scikit learn

[78]:
```
scaler = StandardScaler()

X_train=scaler.fit_transform(X_train)
X_test =scaler.transform(X_test) #we only use scaler.transform to keep the same␣
↪parametter
```

1.3 Overfitting

In this part, we are only interested in maximizing the **train score**, i.e., having the network memorize the training examples as well as possible.

- Propose a parameterization of the network (shape and learning parameters) that will maximize the train score (without considering the test score).

While doing this, you should (1) remain within two minutes of training time, and (2) obtain a score that is greater than 0.90.

- Is the **test** score substantially smaller than the **train** score (indicator of overfitting) ?
- Explain how the parameters you chose allow the learned model to overfit.

[97]:
```
h_layer=[]
train_score_layers=[]
test_score_layers=[]
nb_layers=[i for i in range(2,12)]
for i in range(10):
    h_layer.append(100)
    mlp = MLPRegressor(max_iter=3000,␣
 ↪random_state=1,hidden_layer_sizes=tuple(h_layer))
    mlp.fit(X_train, Y_train)
    train_score_layers.append(mlp.score(X_train, Y_train))
    test_score_layers.append(mlp.score(X_test, Y_test))
```

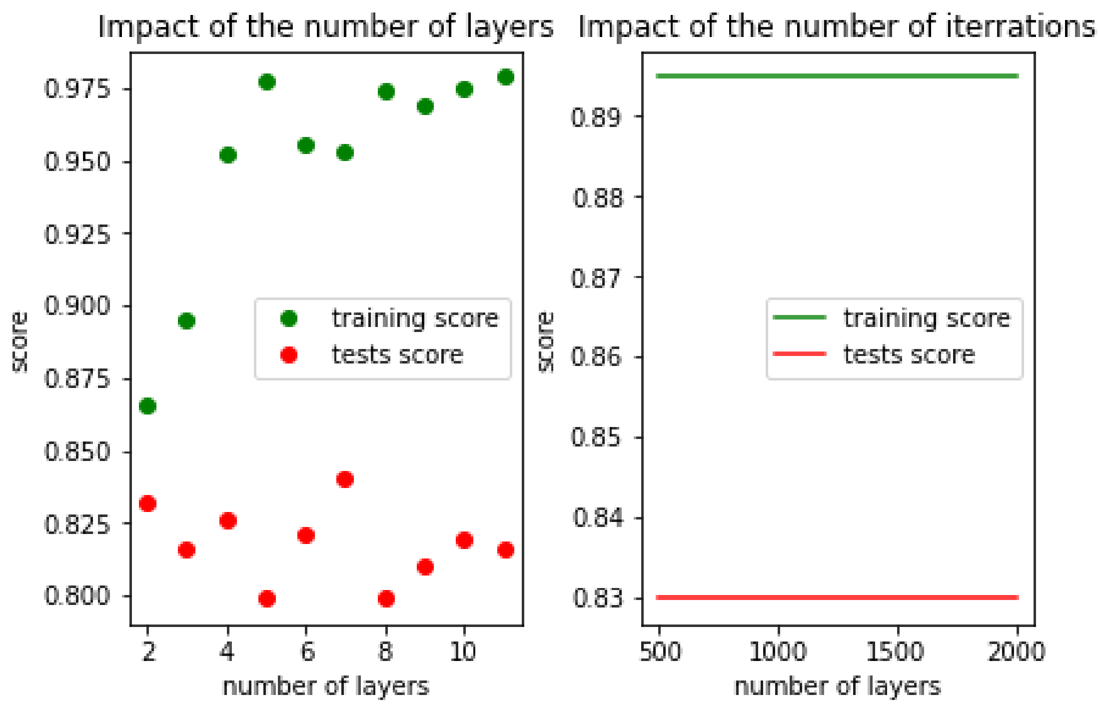
[95]:
```
iter=np.linspace(500,2000,10).astype(int)
train_score_iterations=[]
test_score_iterations=[]
for nb_iter in iter:
    mlp = MLPRegressor(max_iter=nb_iter, random_state=1,␣
 ↪hidden_layer_sizes=(50,50))
    mlp.fit(X_train, Y_train)
    train_score_iterations.append(mlp.score(X_train, Y_train))
    test_score_iterations.append(mlp.score(X_test, Y_test))
```

[102]:
```
fig, axs = plt.subplots(1, 2, constrained_layout=True)

axs.flat[0].plot(nb_layers,train_score_layers,'go',label='training score')
axs.flat[0].plot(nb_layers,test_score_layers,'ro',label='tests score')
axs.flat[0].legend()
axs.flat[0].set_title('Impact of the number of layers')
axs.flat[0].set_xlabel("number of layers")
axs.flat[0].set_ylabel('score')

axs.flat[1].plot(iter, train_score_iterations,'g',label='training score')
axs.flat[1].plot(iter, test_score_iterations,'r',label='tests score')
axs.flat[1].legend()
axs.flat[1].set_title('Impact of the number of iterrations')
axs.flat[1].set_xlabel("number of layers")
axs.flat[1].set_ylabel('score')

plt.show()
```



1.3.1 Remarks

- The number of itteration doesn't seams to have a huge impact on the score

1.4 Hyperparameter tuning

In this section, we are now interested in maximizing the ability of the network to predict the value of unseen examples, i.e., maximizing the **test** score. You should experiment with the possible parameters of the network in order to obtain a good test score, ideally with a small learning time.

Parameters to vary:

- number and size of the hidden layers
- activation function
- stopping conditions
- maximum number of iterations
- initial learning rate value

Results to present for the tested configurations:

- Train/test score
- training time

Present in a table the various parameters tested and the associated results. You can find in the last cell of the notebook a code snippet that will allow you to plot tables from python structure. Be methodical in the way your run your experiments and collect data. For each run, you should record the parameters and results into an external data structure.

(Note that, while we encourage you to explore the solution space manually, there are existing methods in scikit-learn and other learning framework to automate this step as well, e.g., GridSearchCV)

[84]:
```
parameters= [
    {'activation': 'tanh', 'max_iter': 4000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,50), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'relu', 'max_iter': 1000, 'early_stopping': True,␣
 ↪'hidden_layer_sizes':(50, 50,  50), 'learning_rate_init':0.001,␣
 ↪'learning_rate':'constant', 'val_score': None, 'test_score': None,␣
 ↪'train_score': None, 'training_time':None},
    {'activation': 'tanh', 'max_iter': 10000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.001, 'learning_rate':
 ↪'adaptive', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'relu', 'max_iter': 1000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,50), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'logistic', 'max_iter': 3000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,50), 'learning_rate_init':0.001, 'learning_rate':
 ↪'adaptive', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'tanh', 'max_iter': 1000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'tanh', 'max_iter': 1000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'tanh', 'max_iter': 1000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'logistic', 'max_iter': 600, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.01, 'learning_rate':
 ↪'adaptive', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
    {'activation': 'tanh', 'max_iter': 1000, 'early_stopping': False,␣
 ↪'hidden_layer_sizes':(50,), 'learning_rate_init':0.001, 'learning_rate':
 ↪'constant', 'val_score': None, 'test_score': None, 'train_score': None,␣
 ↪'training_time':None},
]
```

[85]:
```
def find_best_parametter(X_train,X_val,Y_train, Y_val, parameters):
    Parameters = copy.deepcopy(parameters)
    for prm in Parameters:
        mlp = MLPRegressor(
            activation=prm['activation'],
            max_iter=prm['max_iter'],
            hidden_layer_sizes=prm['hidden_layer_sizes'],
            learning_rate_init=prm['learning_rate_init'],
            learning_rate=prm['learning_rate'],
            early_stopping=prm['early_stopping']
        )

        st = time.time()
        mlp.fit(X_train, Y_train)
        et = time.time() - st
        prm['train_score'] = mlp.score(X_train, Y_train)
        prm['val_score'] = mlp.score(X_val, Y_val)
        prm['training_time'] = et
    return Parameters
```

[86]:
```
# Code snippet to display a nice table in jupyter notebooks  (remove from␣
↪report)
checked_prms = find_best_parametter(X_train,X_test,Y_train, Y_test, parameters)

table = pd.DataFrame.from_dict(checked_prms)
table = table.replace(np.nan, '-')
table = table.sort_values(by='val_score', ascending=False)
table
```

[86]:

	activation	max_iter	early_stopping	hidden_layer_sizes	learning_rate_init \
0	tanh	4000	False	(50, 50)	0.001
7	tanh	1000	False	(50,)	0.001
5	tanh	1000	False	(50,)	0.001
8	logistic	600	False	(50,)	0.010
9	tanh	1000	False	(50,)	0.001
6	tanh	1000	False	(50,)	0.001
2	tanh	10000	False	(50,)	0.001
4	logistic	3000	False	(50, 50)	0.001
1	relu	1000	True	(50, 50, 50)	0.001
3	relu	1000	False	(50, 50)	0.001

	learning_rate	val_score	test_score	train_score	training_time
0	constant	0.863287	-	0.870984	3.257044
7	constant	0.862584	-	0.862283	2.135387
5	constant	0.859862	-	0.860521	1.783975
8	adaptive	0.859290	-	0.855744	0.617061
9	constant	0.857227	-	0.854386	1.882055

```
6      constant   0.855926              -      0.854648      2.526917
2      adaptive   0.854960              -      0.848747      1.218045
4      adaptive   0.849598              -      0.843675      4.639305
1      constant   0.842588              -      0.838878      0.612407
3      constant   0.840748              -      0.886872      2.884510
```

1.5 Evaluation

- From your experiments, what seems to be the best model (i.e. set of parameters) for predicting the value of a house?

Unless you used cross-validation, you have probably used the "test" set to select the best model among the ones you experimented with. Since your model is the one that worked best on the "test" set, your selection is *biased*.

In all rigor the original dataset should be split in three:

- the **training set**, on which each model is trained
- the **validation set**, that is used to pick the best parameters of the model
- the **test set**, on which we evaluate the final model

Evaluate the score of your algorithm on a test set that was not used for training nor for model selection.

[87]:
```
X_tr, X_test, Y_tr,Y_test = train_test_split(X, Y,random_state=1, test_size = 0.
 ↪20) # 20% for testing
X_train, X_val, Y_train,Y_val = train_test_split(X_tr, Y_tr,random_state=1,␣
 ↪test_size = 0.25) # 20% for validation

scaler = StandardScaler()
X_train=scaler.fit_transform(X_train)
X_val =scaler.transform(X_val) #we only use scaler.transform to keep the same␣
 ↪parametter
X_test =scaler.transform(X_test) #we only use scaler.transform to keep the same␣
 ↪parametter

df =  pd.DataFrame.from_dict(find_best_parametter(X_train,X_val,Y_train,␣
 ↪Y_val,parameters))
df = df.replace(np.nan, '-')

df = df.sort_values(by='val_score', ascending=False)
df
```

[87]:
```
  activation  max_iter  early_stopping hidden_layer_sizes  learning_rate_init  \
0       tanh      4000           False           (50, 50)               0.001
8   logistic       600           False              (50,)               0.010
5       tanh      1000           False              (50,)               0.001
6       tanh      1000           False              (50,)               0.001
2       tanh     10000           False              (50,)               0.001
```

```
7        tanh      1000           False             (50,)              0.001
3        relu      1000           False          (50, 50)              0.001
9        tanh      1000           False             (50,)              0.001
4    logistic      3000           False          (50, 50)              0.001
1        relu      1000            True      (50, 50, 50)              0.001

  learning_rate  val_score test_score  train_score  training_time
0      constant   0.833238          -     0.871523       3.631586
8      adaptive   0.821273          -     0.867381       0.656902
5      constant   0.818847          -     0.855696       1.676839
6      constant   0.817998          -     0.861784       2.038256
2      adaptive   0.816932          -     0.867567       2.375080
7      constant   0.814875          -     0.855830       1.230809
3      constant   0.813052          -     0.886733       2.857791
9      constant   0.808340          -     0.861950       2.663785
4      adaptive   0.807099          -     0.850284       6.157162
1      constant   0.783508          -     0.855461       0.699804
```

[88]:
```
best_params = df[df.val_score == df.val_score.max()].to_dict('records')[0]
```

[89]:
```
mlp = MLPRegressor(
            activation=best_params['activation'],
            max_iter=best_params['max_iter'],
            hidden_layer_sizes=best_params['hidden_layer_sizes'],
            learning_rate_init=best_params['learning_rate_init'],
            learning_rate=best_params['learning_rate'],
            early_stopping=best_params['early_stopping']
        )

st = time.time()
mlp.fit(X_train, Y_train)
et = time.time() - st
# best_params['train_score'] = mlp.score(X_train.values, Y_train)
best_params['test_score'] = mlp.score(X_test, Y_test)
best_params['training_time'] = et
results = pd.DataFrame.from_dict([best_params])
results
```

[89]:
```
  activation  max_iter  early_stopping hidden_layer_sizes  learning_rate_init  \
0       tanh      4000           False           (50, 50)               0.001

  learning_rate  val_score  test_score  train_score  training_time
0      constant   0.833238    0.873047     0.871523       2.791494
```

1.5.1 Remarks

- In the current version of our code we train our model two times

- For selecting the best parameters
- For the final test of th model

It will be better if our function `find_best_paramettter` returned the model that performed best during selection phase along side with the parameters.