lesson-2-4

May 14, 2025

1 NN Heston Model - Model Training

[74]:
```
import pandas as pd
from sklearn.model_selection import train_test_split
```

1.1 Data Preprocessing

Initialization

[75]:
```
import os
import re

verbose    = True
TAG        = '10000_VFA'

workDir    = 'c:/data/'

inFile     = "full_%s.csv" %(TAG)
scalerFile = "scaler_%s.pkl" %(TAG)
mdlDir     = "model_%s.keras" %(TAG)

inFile     = os.path.join(workDir, inFile)
scalerFile = os.path.join(workDir, scalerFile)
mdlDir     = os.path.join(workDir, mdlDir)

print("inFile     : ", inFile)
print("scalerFile : ", scalerFile)
print("mdlDir     : ", mdlDir)

resFile    = re.sub("\..*$","_trained.png", inFile)
print("\n%s -> %s" %(inFile, resFile))
```

```
inFile     :  c:/data/full_10000_VFA.csv
scalerFile :  c:/data/scaler_10000_VFA.pkl
mdlDir     :  c:/data/model_10000_VFA.keras

c:/data/full_10000_VFA.csv -> c:/data/full_10000_VFA_trained.png
```

Read the training DB

[76]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
```

[77]:
```
# Read in training data
print("@ %-24s: reading from '%s'" %("Info", inFile))
db = pd.read_csv(inFile, sep=',')
```

```
@ Info                    : reading from 'c:/data/full_10000_VFA.csv'
```

check that it is is what we expect

[78]:
```
print("*"*82+"\n"+"* X"); print(db.keys()); print("*"*82)
print(db.head(4))
```

```
********************************************************************************
**
* X
Index(['k=0.800', 'k=0.825', 'k=0.850', 'k=0.875', 'k=0.900', 'k=0.925',
       'k=0.950', 'k=0.975', 'k=1.000', 'k=1.025', 'k=1.050', 'k=1.075',
       'k=1.100', 'k=1.125', 'k=1.150', 'k=1.175', 'T', 'Price', 'Strike'],
      dtype='object')
********************************************************************************
**
    k=0.800   k=0.825   k=0.850   k=0.875   k=0.900   k=0.925   k=0.950  \
0  0.573050  0.569880  0.566884  0.564055  0.561388  0.558878  0.556520
1  0.378935  0.371277  0.363728  0.356284  0.348945  0.341711  0.334584
2  0.422837  0.416391  0.410174  0.404189  0.398443  0.392944  0.387701
3  0.170908  0.163006  0.155253  0.147724  0.140550  0.133952  0.128297

    k=0.975   k=1.000   k=1.025   k=1.050   k=1.075   k=1.100   k=1.125  \
0  0.554308  0.552237  0.550303  0.548499  0.546823  0.545267  0.543828
1  0.327567  0.320668  0.313893  0.307252  0.300759  0.294428  0.288277
2  0.382721  0.378015  0.373589  0.369451  0.365608  0.362065  0.358823
3  0.124133  0.122079  0.122437  0.124881  0.128727  0.133368  0.138394

    k=1.150   k=1.175         T     Price   Strike
0  0.542502  0.541282  1.841204  0.264816  0.95396
1  0.282327  0.276600  1.432954  0.116339  0.91180
2  0.355884  0.353244  1.621171  0.137442  0.88340
3  0.143568  0.148754  1.757063  0.285559  1.26748
```

[79]:
```
from sklearn.preprocessing import StandardScaler

'''
    It is critical that any data preparation performed on a training dataset is␣
 ↪also performed
    on a new dataset in the future. This may include a test dataset when␣
 ↪evaluating a model
```

```
    or new data from the domain when using a model to make predictions.␣
↪Typically, the model fit
    on the training dataset is saved for later use. The correct solution to␣
↪preparing new data
    for the model in the future is to also save any data preparation objects,␣
↪like data scaling methods,
    to file along with the model.
'''

def preprocess(**keywrds):

    db = keywrds["db"]


    # Specify the target labels and flatten the array
    #t=np.ravel(db["Price"])
    t=db["Price"]

    # Specify the data
    X = db.drop(columns="Price")

    print("Info")
    print(X.info())

    print("Head")
    print(X.head(n=2))
    print("Tail")
    print(X.tail(n=2))

    print("Describe")
    print(X.describe())


    # Define the scaler
    scaler = StandardScaler().fit(X)

    # Split the data up in train and test sets
    X_train, X_test, t_train, t_test = train_test_split(X, t, test_size=0.33,␣
↪random_state=42)

    # Scale the train set
    X_train = scaler.transform(X_train)

    # Scale the test set
    X_test = scaler.transform(X_test)

    return scaler, X_train, X_test, t_train, t_test
```

[80]:
```
scaler, X_train, X_test, t_train, t_test = preprocess(db = db)
```

```
Info
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6697 entries, 0 to 6696
Data columns (total 18 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   k=0.800  6697 non-null   float64
 1   k=0.825  6697 non-null   float64
 2   k=0.850  6697 non-null   float64
 3   k=0.875  6697 non-null   float64
 4   k=0.900  6697 non-null   float64
 5   k=0.925  6697 non-null   float64
 6   k=0.950  6697 non-null   float64
 7   k=0.975  6697 non-null   float64
 8   k=1.000  6697 non-null   float64
 9   k=1.025  6697 non-null   float64
 10  k=1.050  6697 non-null   float64
 11  k=1.075  6697 non-null   float64
 12  k=1.100  6697 non-null   float64
 13  k=1.125  6697 non-null   float64
 14  k=1.150  6697 non-null   float64
 15  k=1.175  6697 non-null   float64
 16  T        6697 non-null   float64
 17  Strike   6697 non-null   float64
dtypes: float64(18)
memory usage: 941.9 KB
None
Head
    k=0.800   k=0.825   k=0.850   k=0.875   k=0.900   k=0.925   k=0.950  \
0  0.573050  0.569880  0.566884  0.564055  0.561388  0.558878  0.556520
1  0.378935  0.371277  0.363728  0.356284  0.348945  0.341711  0.334584

    k=0.975   k=1.000   k=1.025   k=1.050   k=1.075   k=1.100   k=1.125  \
0  0.554308  0.552237  0.550303  0.548499  0.546823  0.545267  0.543828
1  0.327567  0.320668  0.313893  0.307252  0.300759  0.294428  0.288277

    k=1.150   k=1.175         T   Strike
0  0.542502  0.541282  1.841204  0.95396
1  0.282327  0.276600  1.432954  0.91180
Tail
       k=0.800  k=0.825   k=0.850   k=0.875   k=0.900   k=0.925   k=0.950  \
6695  0.829568  0.82951  0.829454  0.829399  0.829347  0.829295  0.829245
6696  0.351620  0.34969  0.347851  0.346102  0.344441  0.342863  0.341368

       k=0.975   k=1.000   k=1.025   k=1.050   k=1.075   k=1.100   k=1.125  \
6695  0.829197  0.829149  0.829103  0.829058  0.829014  0.828972  0.828930
```

```
6696  0.339953  0.338616  0.337354  0.336165  0.335047  0.333997  0.333015

       k=1.150   k=1.175          T   Strike
6695  0.828889  0.828849  0.736821  1.37860
6696  0.332096  0.331239  1.836221  1.17684
Describe
           k=0.800      k=0.825      k=0.850      k=0.875      k=0.900  \
count  6697.000000  6697.000000  6697.000000  6697.000000  6697.000000
mean      0.606839     0.603743     0.600735     0.597816     0.594982
std       0.158626     0.159693     0.160778     0.161876     0.162982
min       0.146909     0.142970     0.132600     0.122145     0.111587
25%       0.492756     0.489320     0.486449     0.482658     0.479295
50%       0.624001     0.621531     0.619070     0.616246     0.613567
75%       0.736599     0.734297     0.731935     0.729554     0.727380
max       0.919215     0.912689     0.906287     0.900002     0.893828

           k=0.925      k=0.950      k=0.975      k=1.000      k=1.025  \
count  6697.000000  6697.000000  6697.000000  6697.000000  6697.000000
mean      0.592236     0.589578     0.587012     0.584548     0.582204
std       0.164090     0.165190     0.166267     0.167292     0.168211
min       0.100332     0.087885     0.075302     0.059323     0.038954
25%       0.476248     0.472942     0.469556     0.466668     0.463882
50%       0.611268     0.609422     0.606685     0.604351     0.601585
75%       0.725465     0.723450     0.721901     0.719500     0.718018
max       0.890617     0.890244     0.889887     0.889543     0.889213

           k=1.050      k=1.075      k=1.100      k=1.125      k=1.150  \
count  6697.000000  6697.000000  6697.000000  6697.000000  6697.000000
mean      0.579987     0.577891     0.575910     0.574038     0.572269
std       0.169002     0.169682     0.170258     0.170745     0.171153
min       0.043279     0.050258     0.056551     0.053186     0.052685
25%       0.461256     0.458251     0.456006     0.453856     0.451856
50%       0.599463     0.597354     0.594991     0.592908     0.591008
75%       0.716498     0.714337     0.712927     0.711778     0.710486
max       0.888896     0.888591     0.888465     0.888456     0.888448

           k=1.175            T       Strike
count  6697.000000  6697.000000  6697.000000
mean      0.570596     1.035687     0.999372
std       0.171489     0.552914     0.232575
min       0.053117     0.083429     0.600040
25%       0.450155     0.557996     0.797320
50%       0.589713     1.030838     0.998760
75%       0.708710     1.516329     1.202200
max       0.888441     1.999904     1.399960
```

1.2 Auxiliary Functions

[81]:
```
def show_scattered( y, t, tag, ax = None):
    #x      = model.predict(X)
    #y      = np.ravel(x)
    xMin = min(t)
    xMax = max(t)
    v    = np.arange(xMin, xMax, (xMax-xMin)/100.)

    diff   = np.fabs(y - t)
    print("@ %-24s: E[y-t]: %.6f Std(y-t): %.6f" %( tag, np.mean(diff), np.
 ↪std(diff)))
    if ax == None: return

    ax.plot( y, t, ".")
    ax.plot( v, v, color="red")
    ax.set_title("%s mae=%8.4f, std=%8.4f" %(tag, np.mean(diff), np.std(diff)))
    ax.set_xlabel("predicted")
    ax.set_ylabel("target")
```

[82]:
```
def display_nn_results( model, X_train, X_test, t_train, t_test, resFile=None):

    fig, ax = plt.subplots(1,2, figsize=(12,6))
    fig.suptitle("Scattered plots")

    y_train  = np.ravel(model.predict(X_train))
    show_scattered( y_train, t_train, "InSample", ax = ax[0])

    diff   = np.fabs(y_train - t_train)
    RES    = pd.DataFrame({"predicted": y_train, "target": t_train, "err:":␣
 ↪diff})
    RES.to_csv(os.path.join(workDir, "res_in__sample.csv"), sep=',',␣
 ↪float_format="%.6f", index=True)
    print("@")
    y_test  = np.ravel(model.predict(X_test))
    show_scattered( y_test , t_test, "OutOfSample", ax= ax[1])

    diff   = np.fabs(y_test-t_test)
    RES    = pd.DataFrame({"predicted": y_test, "target": t_test, "err:": diff})
    RES.to_csv(os.path.join(workDir, "res_out_sample.csv"), sep=',',␣
 ↪float_format="%.6f", index=True)

    print("@")

    if resFile != None:
        plt.savefig(resFile, format="png")
        print("@ %-12s: results saved to '%s' "%("Info", resFile))
```

```
    plt.show()


    score = model.evaluate(X_test, t_test, verbose=1)
    print('Score:'); print(score)
```

1.3 Build the model

[83]:
```
from keras.models import Sequential
from keras.layers import Dense

def model_builder( inputShape = (1,)):

    # Initialize the constructor
    model = Sequential()

    # Add an input layer
    model.add(Dense(128, activation='relu', input_shape=inputShape))

    # Add one more hidden layer
    model.add(Dense(64, activation='relu'))

    # Add one more hidden layer
    model.add(Dense(32, activation='relu'))

    # Add one more hidden layer
    model.add(Dense(16, activation='relu'))

    # Add an output layer
    model.add(Dense(1))
    # End model construction

    # Model output shape
    print("model.output_shape: %s" %(str(model.output_shape)))

    # Model summary
    print("Model.summary"); model.summary()

    # Model config
    print("Model.config"); model.get_config()

    model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
    return model
```

Let's go through this code line by line:

- The first line creates a **Sequential** model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially. This is called the sequential API.

- Next, we build the first layer and add it to the model. It is a **Dense** hidden layer with XXX neurons. It will use the **ReLu** activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron).
- Next we add a second Dense hidden layer with XXX neurons, also using the ReLu activation function and a third one …
- Finally, we add a Dense output layer with only 1 neurons, using the ReLu activation function (because…).

[84]:
```
model = model_builder( inputShape = (X_train.shape[1],))
```

```
model.output_shape: (None, 1)
Model.summary
```

```
Model: "sequential_3"



 Layer (type)                        Output Shape                     Param␣
↪#

 dense_15 (Dense)                    (None, 128)                    ␣
↪2,432

 dense_16 (Dense)                    (None, 64)                     ␣
↪8,256

 dense_17 (Dense)                    (None, 32)                     ␣
↪2,080

 dense_18 (Dense)                    (None, 16)                       ␣
↪528

 dense_19 (Dense)                    (None, 1)                         ␣
↪17



 Total params: 13,313 (52.00 KB)


 Trainable params: 13,313 (52.00 KB)


 Non-trainable params: 0 (0.00 B)
```

```
Model.config
```

Note that Dense layers often have a lot of parameters. For example, the first hidden layer has n × n connection weights, plus 300 bias terms, which adds up to XXX parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data.

You can easily get a model's list of layers, to fetch a layer by its index, or you can fetch it by name:

[85]:
```
model.layers
```

[85]:
```
[<Dense name=dense_15, built=True>,
 <Dense name=dense_16, built=True>,
 <Dense name=dense_17, built=True>,
 <Dense name=dense_18, built=True>,
 <Dense name=dense_19, built=True>]
```

[86]:
```
model.layers[1].name
```

[86]:
```
'dense_16'
```

After a model is created, you must call its ***compile()*** method to specify the loss function and the optimizer to use. Optionally, you can also specify a list of extra metrics to compute during training and evaluation. In this case we have chosen

```
model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
```

1.4 Train the model

Now the model is ready to be trained. For this we simply need to call its ***fit()*** method. We pass it the input features (***X_train***) and the target classes (***y_train***), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We could also pass a validation set (this is optional): Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs: if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set (or there is a bug, such as a data mismatch between the training set and the validation set).

[87]:
```
history = model.fit(X_train, t_train, epochs=50, verbose=verbose)
```

```
Epoch 1/50
141/141  1s 3ms/step -
loss: 0.0140 - mae: 0.0744
Epoch 2/50
141/141  0s 2ms/step -
loss: 0.0012 - mae: 0.0260
Epoch 3/50
141/141  0s 3ms/step -
loss: 6.3182e-04 - mae: 0.0181
Epoch 4/50
141/141  0s 2ms/step -
loss: 4.3688e-04 - mae: 0.0144
```

```
Epoch 5/50
141/141  0s 2ms/step -
loss: 2.4109e-04 - mae: 0.0112
Epoch 6/50
141/141  0s 2ms/step -
loss: 2.3481e-04 - mae: 0.0100
Epoch 7/50
141/141  0s 2ms/step -
loss: 1.9908e-04 - mae: 0.0099
Epoch 8/50
141/141  0s 3ms/step -
loss: 1.6886e-04 - mae: 0.0093
Epoch 9/50
141/141  0s 2ms/step -
loss: 1.4077e-04 - mae: 0.0083
Epoch 10/50
141/141  0s 2ms/step -
loss: 1.2219e-04 - mae: 0.0075
Epoch 11/50
141/141  0s 2ms/step -
loss: 8.5534e-05 - mae: 0.0066
Epoch 12/50
141/141  0s 2ms/step -
loss: 9.4579e-05 - mae: 0.0067
Epoch 13/50
141/141  0s 2ms/step -
loss: 1.2803e-04 - mae: 0.0075
Epoch 14/50
141/141  0s 2ms/step -
loss: 1.0178e-04 - mae: 0.0074
Epoch 15/50
141/141  0s 2ms/step -
loss: 9.6532e-05 - mae: 0.0069
Epoch 16/50
141/141  0s 3ms/step -
loss: 8.2868e-05 - mae: 0.0067
Epoch 17/50
141/141  0s 2ms/step -
loss: 7.4655e-05 - mae: 0.0062
Epoch 18/50
141/141  0s 2ms/step -
loss: 6.8049e-05 - mae: 0.0058
Epoch 19/50
141/141  0s 2ms/step -
loss: 6.0877e-05 - mae: 0.0057
Epoch 20/50
141/141  0s 3ms/step -
loss: 6.7638e-05 - mae: 0.0061
```

```
Epoch 21/50
141/141  0s 3ms/step -
loss: 5.6566e-05 - mae: 0.0055
Epoch 22/50
141/141  0s 3ms/step -
loss: 6.7410e-05 - mae: 0.0059
Epoch 23/50
141/141  0s 2ms/step -
loss: 6.0776e-05 - mae: 0.0056
Epoch 24/50
141/141  0s 2ms/step -
loss: 4.6231e-05 - mae: 0.0050
Epoch 25/50
141/141  0s 2ms/step -
loss: 5.0525e-05 - mae: 0.0054
Epoch 26/50
141/141  1s 2ms/step -
loss: 5.6933e-05 - mae: 0.0055
Epoch 27/50
141/141  0s 3ms/step -
loss: 4.8129e-05 - mae: 0.0050
Epoch 28/50
141/141  0s 2ms/step -
loss: 4.2539e-05 - mae: 0.0049
Epoch 29/50
141/141  0s 3ms/step -
loss: 3.7992e-05 - mae: 0.0046
Epoch 30/50
141/141  0s 3ms/step -
loss: 5.2889e-05 - mae: 0.0053
Epoch 31/50
141/141  0s 3ms/step -
loss: 4.4937e-05 - mae: 0.0048
Epoch 32/50
141/141  1s 3ms/step -
loss: 3.4116e-05 - mae: 0.0044
Epoch 33/50
141/141  0s 3ms/step -
loss: 3.8293e-05 - mae: 0.0046
Epoch 34/50
141/141  1s 4ms/step -
loss: 3.5580e-05 - mae: 0.0043
Epoch 35/50
141/141  0s 3ms/step -
loss: 3.5239e-05 - mae: 0.0043
Epoch 36/50
141/141  0s 3ms/step -
loss: 3.6729e-05 - mae: 0.0045
```

```
Epoch 37/50
141/141  1s 2ms/step -
loss: 3.7614e-05 - mae: 0.0046
Epoch 38/50
141/141  0s 3ms/step -
loss: 3.8931e-05 - mae: 0.0044
Epoch 39/50
141/141  1s 4ms/step -
loss: 2.8541e-05 - mae: 0.0039
Epoch 40/50
141/141  1s 3ms/step -
loss: 3.2066e-05 - mae: 0.0041
Epoch 41/50
141/141  1s 4ms/step -
loss: 4.1623e-05 - mae: 0.0044
Epoch 42/50
141/141  1s 4ms/step -
loss: 3.1241e-05 - mae: 0.0041
Epoch 43/50
141/141  1s 4ms/step -
loss: 3.1261e-05 - mae: 0.0041
Epoch 44/50
141/141  1s 3ms/step -
loss: 3.6174e-05 - mae: 0.0041
Epoch 45/50
141/141  0s 3ms/step -
loss: 3.3008e-05 - mae: 0.0039
Epoch 46/50
141/141  1s 3ms/step -
loss: 3.0926e-05 - mae: 0.0040
Epoch 47/50
141/141  0s 3ms/step -
loss: 2.7272e-05 - mae: 0.0039
Epoch 48/50
141/141  0s 3ms/step -
loss: 4.3255e-05 - mae: 0.0043
Epoch 49/50
141/141  0s 3ms/step -
loss: 2.6775e-05 - mae: 0.0037
Epoch 50/50
141/141  1s 4ms/step -
loss: 2.6362e-05 - mae: 0.0038
```

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of instances processed so far (along with a progress bar), the mean training time per sample, the loss and accuracy (or any other extra metrics you asked for), both on the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached XXX% after 50 epochs, not too far from the training accuracy, so there does not seem to be much overfitting going on.

All the parameters of a layer can be accessed using its ***get_ weights()*** and ***set_ weights()*** method. For a Dense layer, this includes both the connection weights and the bias terms:

[88]:
```
weights, biases = model.layers[1].get_weights()
weights
```

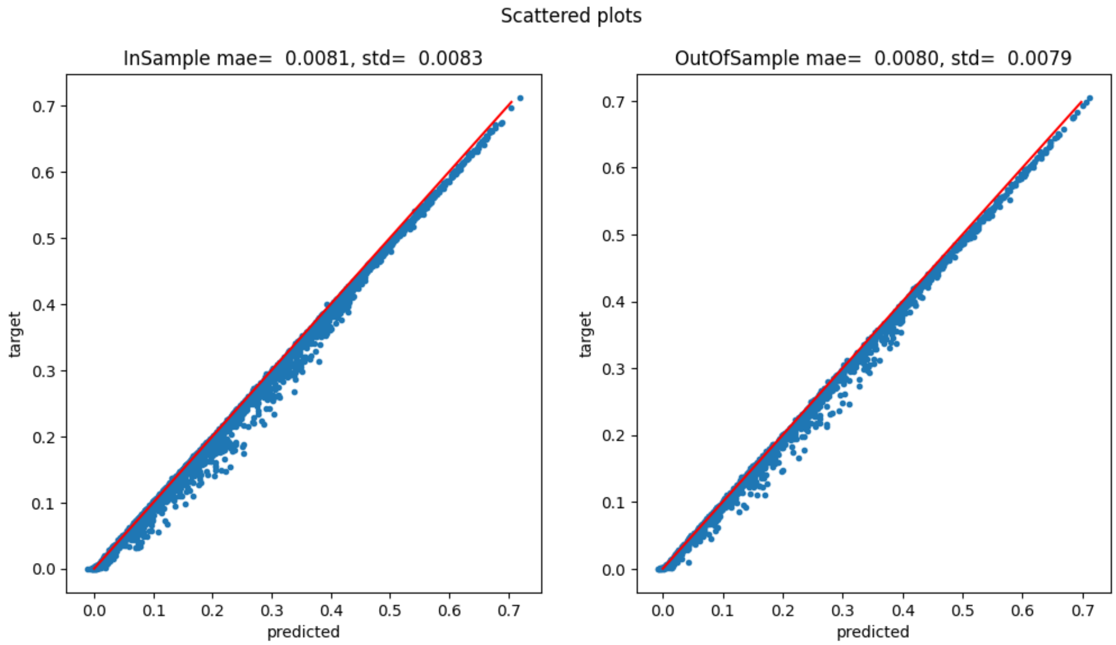
[88]:
```
array([[ 0.13675848, -0.16520654, -0.05728348, ...,  0.06568525,
        -0.07473467,  0.11186583],
       [ 0.09047444, -0.14670068,  0.03718283, ...,  0.0532075 ,
         0.01450639, -0.03485379],
       [-0.01020768,  0.15483963, -0.02219277, ..., -0.07704523,
        -0.09269592,  0.08660694],
       ...,
       [ 0.09854848,  0.09179175,  0.01159348, ...,  0.11240668,
        -0.17557618, -0.0304074 ],
       [-0.1667468 , -0.12398612,  0.07837373, ..., -0.02433819,
         0.0972604 , -0.07057186],
       [-0.09303889,  0.09912827,  0.11654822, ...,  0.07594239,
         0.0204009 ,  0.00023034]], dtype=float32)
```

[89]:
```
import warnings
warnings.simplefilter('ignore')

import matplotlib.pyplot as plt
import numpy             as np

display_nn_results(model, X_train, X_test, t_train, t_test, resFile = resFile)
```

```
141/141  0s 2ms/step
@ InSample                  : E[y-t]: 0.008105 Std(y-t): 0.008295
@
70/70  0s 2ms/step
@ OutOfSample               : E[y-t]: 0.008040 Std(y-t): 0.007902
@
@ Info        : results saved to 'c:/data/full_10000_VFA_trained.png'
```




```
70/70  1s 4ms/step - loss:
1.2836e-04 - mae: 0.0080
Score:
[0.0001270899665541947, 0.008039971813559532]
```

1.5 Save Scaler and Model on Disk

[90]:
```
from pickle import dump, load

dump(scaler, open(scalerFile, 'wb'))
print("@ %-24s: scaler saved to '%s'" %("Info", scalerFile))

model.save(mdlDir)
print("@ %-24s: model saved to '%s'" %("Info", mdlDir))
```

```
@ Info                    : scaler saved to 'c:/data/scaler_10000_VFA.pkl'
@ Info                    : model saved to 'c:/data/model_10000_VFA.keras'
```

[91]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    files.download(scalerFile)
    #files.download(mdlDir)
```

1.6 Using the History Object

The ***fit()*** method returns a History object containing the training parameters (history.params), the list of epochs it went through (history.epoch), and most importantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if any). If you create a Pandas DataFrame using this dictionary and call its plot() method, you get the learning curves.

[92]:
```
X = db.drop(columns="Price")
Y = db["Price"]
```

[93]:
```
# Fit the model
history = model.fit(X, Y, validation_split=0.33, epochs=50, verbose=1)
```

```
Epoch 1/50
141/141  1s 5ms/step -
loss: 0.0162 - mae: 0.1024 - val_loss: 0.0054 - val_mae: 0.0588
Epoch 2/50
141/141  1s 4ms/step -
loss: 0.0039 - mae: 0.0496 - val_loss: 0.0012 - val_mae: 0.0279
Epoch 3/50
141/141  1s 3ms/step -
loss: 0.0025 - mae: 0.0382 - val_loss: 0.0015 - val_mae: 0.0345
Epoch 4/50
141/141  1s 4ms/step -
loss: 0.0021 - mae: 0.0369 - val_loss: 4.5011e-04 - val_mae: 0.0163
Epoch 5/50
141/141  0s 3ms/step -
loss: 0.0016 - mae: 0.0310 - val_loss: 3.8878e-04 - val_mae: 0.0145
Epoch 6/50
141/141  1s 3ms/step -
loss: 0.0013 - mae: 0.0266 - val_loss: 3.1502e-04 - val_mae: 0.0125
Epoch 7/50
141/141  1s 4ms/step -
loss: 0.0013 - mae: 0.0284 - val_loss: 0.0031 - val_mae: 0.0504
Epoch 8/50
141/141  1s 4ms/step -
loss: 0.0011 - mae: 0.0267 - val_loss: 4.4719e-04 - val_mae: 0.0161
Epoch 9/50
141/141  1s 3ms/step -
loss: 0.0010 - mae: 0.0248 - val_loss: 3.7700e-04 - val_mae: 0.0169
Epoch 10/50
141/141  1s 4ms/step -
loss: 9.4939e-04 - mae: 0.0241 - val_loss: 1.8114e-04 - val_mae: 0.0108
Epoch 11/50
141/141  1s 4ms/step -
loss: 0.0010 - mae: 0.0252 - val_loss: 7.6007e-04 - val_mae: 0.0243
Epoch 12/50
141/141  1s 4ms/step -
loss: 8.4206e-04 - mae: 0.0229 - val_loss: 8.7992e-04 - val_mae: 0.0269
Epoch 13/50
141/141  1s 5ms/step -
```

```
loss: 9.1144e-04 - mae: 0.0237 - val_loss: 1.7710e-04 - val_mae: 0.0109
Epoch 14/50
141/141  1s 3ms/step -
loss: 8.5230e-04 - mae: 0.0231 - val_loss: 8.8752e-04 - val_mae: 0.0267
Epoch 15/50
141/141  0s 3ms/step -
loss: 9.0189e-04 - mae: 0.0232 - val_loss: 8.4888e-05 - val_mae: 0.0069
Epoch 16/50
141/141  0s 3ms/step -
loss: 7.8519e-04 - mae: 0.0200 - val_loss: 1.1861e-04 - val_mae: 0.0089
Epoch 17/50
141/141  0s 3ms/step -
loss: 8.1565e-04 - mae: 0.0213 - val_loss: 0.0019 - val_mae: 0.0374
Epoch 18/50
141/141  1s 4ms/step -
loss: 7.8796e-04 - mae: 0.0222 - val_loss: 0.0012 - val_mae: 0.0327
Epoch 19/50
141/141  1s 4ms/step -
loss: 7.5303e-04 - mae: 0.0214 - val_loss: 4.6504e-04 - val_mae: 0.0183
Epoch 20/50
141/141  1s 3ms/step -
loss: 7.5200e-04 - mae: 0.0221 - val_loss: 3.2066e-04 - val_mae: 0.0158
Epoch 21/50
141/141  1s 3ms/step -
loss: 7.3363e-04 - mae: 0.0208 - val_loss: 2.0637e-04 - val_mae: 0.0121
Epoch 22/50
141/141  1s 3ms/step -
loss: 6.1495e-04 - mae: 0.0183 - val_loss: 7.9549e-04 - val_mae: 0.0255
Epoch 23/50
141/141  0s 3ms/step -
loss: 7.2190e-04 - mae: 0.0220 - val_loss: 0.0020 - val_mae: 0.0388
Epoch 24/50
141/141  1s 3ms/step -
loss: 6.5351e-04 - mae: 0.0204 - val_loss: 0.0011 - val_mae: 0.0298
Epoch 25/50
141/141  0s 3ms/step -
loss: 6.6565e-04 - mae: 0.0196 - val_loss: 1.0335e-04 - val_mae: 0.0080
Epoch 26/50
141/141  1s 4ms/step -
loss: 5.8220e-04 - mae: 0.0192 - val_loss: 5.5441e-04 - val_mae: 0.0197
Epoch 27/50
141/141  1s 3ms/step -
loss: 5.8565e-04 - mae: 0.0192 - val_loss: 0.0014 - val_mae: 0.0339
Epoch 28/50
141/141  1s 4ms/step -
loss: 6.3924e-04 - mae: 0.0209 - val_loss: 9.8665e-04 - val_mae: 0.0296
Epoch 29/50
141/141  1s 4ms/step -
```

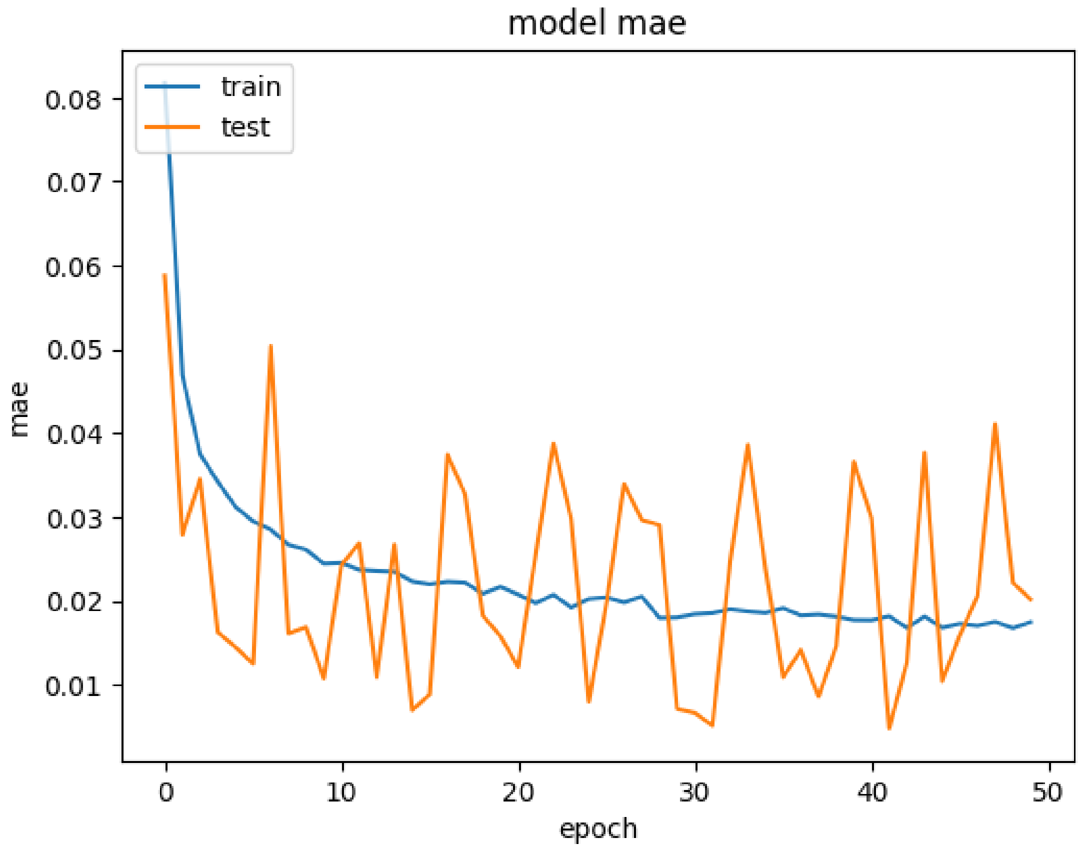
```
loss: 5.8318e-04 - mae: 0.0191 - val_loss: 0.0010 - val_mae: 0.0291
Epoch 30/50
141/141  1s 4ms/step -
loss: 7.0880e-04 - mae: 0.0195 - val_loss: 9.1977e-05 - val_mae: 0.0071
Epoch 31/50
141/141  1s 3ms/step -
loss: 5.2818e-04 - mae: 0.0176 - val_loss: 6.8119e-05 - val_mae: 0.0066
Epoch 32/50
141/141  1s 4ms/step -
loss: 5.1584e-04 - mae: 0.0175 - val_loss: 4.7948e-05 - val_mae: 0.0051
Epoch 33/50
141/141  1s 4ms/step -
loss: 5.7024e-04 - mae: 0.0187 - val_loss: 7.3503e-04 - val_mae: 0.0247
Epoch 34/50
141/141  1s 5ms/step -
loss: 5.7379e-04 - mae: 0.0198 - val_loss: 0.0018 - val_mae: 0.0386
Epoch 35/50
141/141  1s 5ms/step -
loss: 6.1296e-04 - mae: 0.0203 - val_loss: 6.7715e-04 - val_mae: 0.0239
Epoch 36/50
141/141  1s 3ms/step -
loss: 5.9373e-04 - mae: 0.0201 - val_loss: 1.5060e-04 - val_mae: 0.0109
Epoch 37/50
141/141  1s 4ms/step -
loss: 4.7866e-04 - mae: 0.0173 - val_loss: 3.0920e-04 - val_mae: 0.0141
Epoch 38/50
141/141  1s 5ms/step -
loss: 5.3700e-04 - mae: 0.0191 - val_loss: 1.0656e-04 - val_mae: 0.0086
Epoch 39/50
141/141  1s 4ms/step -
loss: 5.1516e-04 - mae: 0.0189 - val_loss: 2.5617e-04 - val_mae: 0.0146
Epoch 40/50
141/141  1s 4ms/step -
loss: 5.6023e-04 - mae: 0.0188 - val_loss: 0.0015 - val_mae: 0.0366
Epoch 41/50
141/141  1s 4ms/step -
loss: 5.2207e-04 - mae: 0.0182 - val_loss: 0.0011 - val_mae: 0.0298
Epoch 42/50
141/141  1s 5ms/step -
loss: 4.8018e-04 - mae: 0.0179 - val_loss: 3.7023e-05 - val_mae: 0.0048
Epoch 43/50
141/141  1s 4ms/step -
loss: 4.7038e-04 - mae: 0.0165 - val_loss: 2.2526e-04 - val_mae: 0.0126
Epoch 44/50
141/141  1s 5ms/step -
loss: 4.7472e-04 - mae: 0.0182 - val_loss: 0.0019 - val_mae: 0.0377
Epoch 45/50
141/141  1s 4ms/step -
```

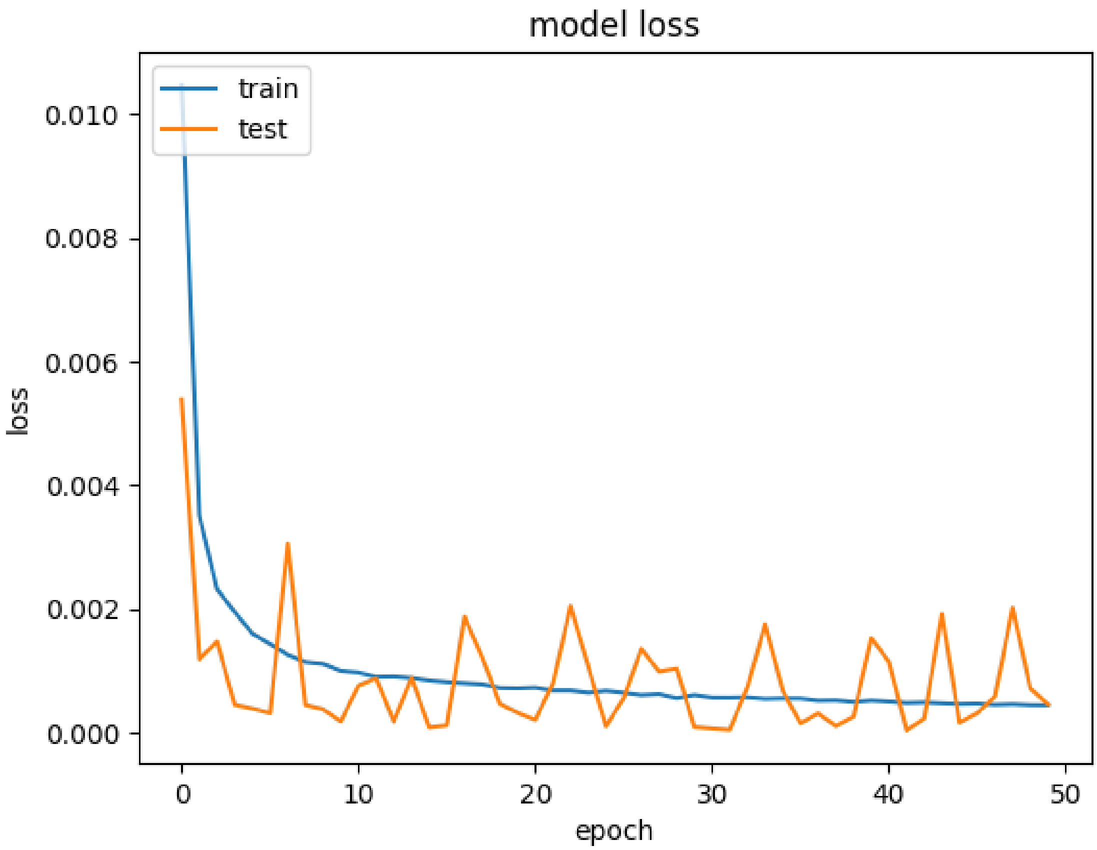
```
loss: 5.0719e-04 - mae: 0.0178 - val_loss: 1.6151e-04 - val_mae: 0.0104
Epoch 46/50
141/141  1s 4ms/step -
loss: 4.5769e-04 - mae: 0.0175 - val_loss: 3.1560e-04 - val_mae: 0.0159
Epoch 47/50
141/141  1s 4ms/step -
loss: 4.7072e-04 - mae: 0.0176 - val_loss: 5.8409e-04 - val_mae: 0.0206
Epoch 48/50
141/141  1s 4ms/step -
loss: 4.6448e-04 - mae: 0.0180 - val_loss: 0.0020 - val_mae: 0.0411
Epoch 49/50
141/141  1s 3ms/step -
loss: 4.6634e-04 - mae: 0.0170 - val_loss: 7.1563e-04 - val_mae: 0.0222
Epoch 50/50
141/141  1s 4ms/step -
loss: 4.1233e-04 - mae: 0.0159 - val_loss: 4.5794e-04 - val_mae: 0.0202
```

[94]:

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'mae', 'val_loss', 'val_mae'])
```


model mae
train
test
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
mae
0
10
20
30
40
50
epoch


model loss
train
test
0.010
0.008
0.006
0.004
0.002
0.000
loss
0
10
20
30
40
50
epoch

```
[ ]:
```