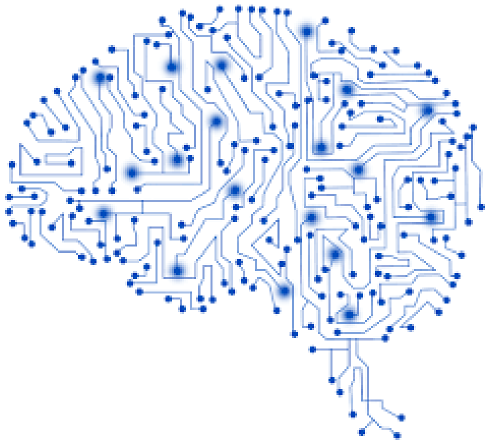
University of Minho
School of Engineering



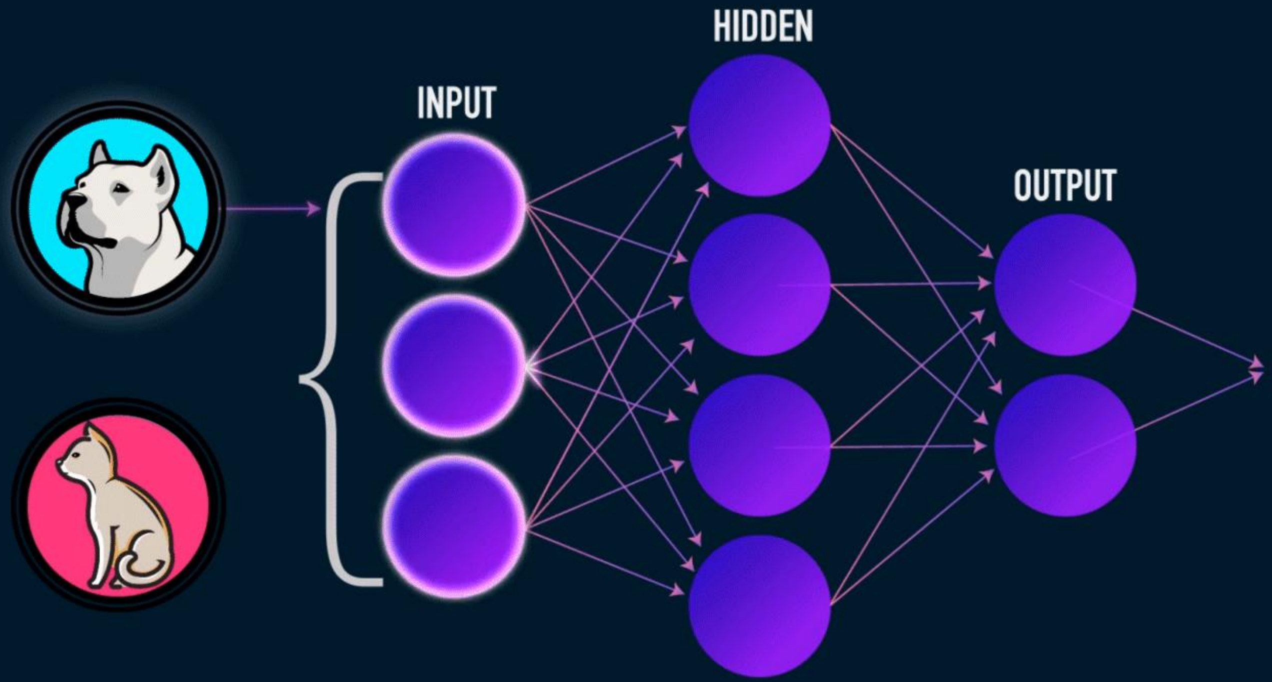
Dados e Aprendizagem Automática

Artificial Neural Networks

DAA @ MEI-1º/MiEI-4º/MMC-1º – 1º Semestre

Bruno Fernandes, Filipe Gonçalves, Víctor Alves, Cesar Analide

Part VII



(https://miro.medium.com/max/2000/1*bhFifratH9DjKqMBTeQG5A.gif)

What about Artificial Neural Networks?

Let's get back to our "real estate agent" problem and help our neighbor (the real estate agent) to predict housing prices for regions in the USA (dataset here).

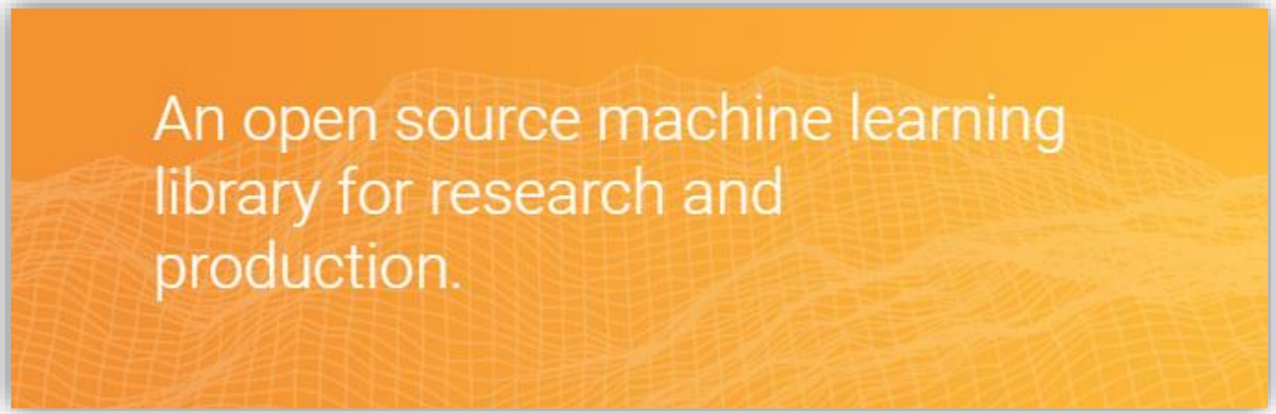
We have already used Linear Regression... But now let's try using Multilayer Perceptrons (MLPs), a class of Artificial Neural Networks!

To implement our first Artificial Neural Network we will use:

TensorFlow for Artificial Neural Networks

Why?

- Open-source software library for high performance numerical computation
- Strong support for machine learning and deep learning
- It has seen tremendous growth and popularity in the machine learning community



Companies using TensorFlow

Implementing a MLP - The Dataset

We already know this dataset (let's drop the *address* - we don't need it)! We will work a regression problem!

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Check the data (again...)

```
USAhousing = pd.read_csv('USA_Housing.csv')
USAhousing.drop('Address', axis=1, inplace=True)
USAhousing.head()
```

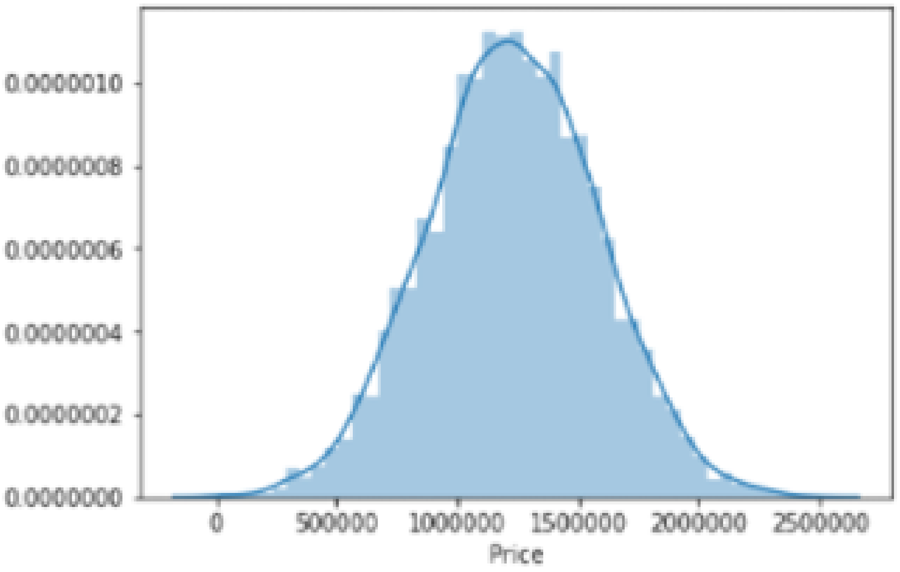
	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06
1	79248.642455	6.002900	6.730821	3.09	40173.072174	1.505891e+06
2	61287.067179	5.865890	8.512727	5.13	36882.159400	1.058988e+06
3	63345.240046	7.188236	5.586729	3.26	34310.242831	1.260617e+06
4	59982.197226	5.040555	7.839388	4.23	26354.109472	6.309435e+05

Implementing a MLP - Some Data Viz.

We have already explored it before so… Let’s move on!

```
sns.distplot(USAhousing['Price'])
```

<matplotlib.axes._subplots.AxesSubplot at 0x2972eb2ea90>




```
sns.heatmap(USAhousing.corr())
```

<matplotlib.axes._subplots.AxesSubplot at 0x2972f437240>

Implementing a MLP - More imports!

To implement our first MLP we will need some more libraries! Let's import them all at once! You'll need to install TensorFlow - use the Navigator or the Prompt:

conda install -c conda-forge tensorflow

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import GridSearchCV, KFold, train_test_split
from sklearn.preprocessing import MinMaxScaler

RANDOM_SEED = 2021

print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.3.0
```

Implementing a MLP - Scaling the data

We can then define our X and y, as usual!

```
X = USAhousing.drop('Price', axis=1)
y = USAhousing[['Price']]
```

Artificial neural networks are picky - they prefer scaled data! Let's scale the data to be in the interval [0, 1].

```
# Let's scale the features between [0-1]
scaler_X = MinMaxScaler(feature_range=(0, 1)).fit(X)
scaler_y = MinMaxScaler(feature_range=(0, 1)).fit(y)
X_scaled = pd.DataFrame(scaler_X.transform(X[X.columns]), columns=X.columns)
y_scaled = pd.DataFrame(scaler_y.transform(y[y.columns]), columns=y.columns)
```

Implementing a MLP - Scaling the data

Let's just make some checks!

Everything is OK!

```
X.head()
```

	Avg. Area Income	Avg. Area House Age
0	79545.458574	5.682861
1	79248.642455	6.002900
2	61287.067179	5.865890
3	63345.240046	7.188236
4	59982.197226	5.040555

```
X_scaled.head()
```

	Avg. Area Income	Avg. Area House Age
0	0.686822	0.441986
1	0.683521	0.488538
2	0.483737	0.468609
3	0.506630	0.660956
4	0.469223	0.348556

```
y.head()
```

	Price
0	1.059034e+06
1	1.505891e+06
2	1.058988e+06
3	1.260617e+06
4	6.309435e+05

```
y_scaled.head()
```

	Price
0	0.425210
1	0.607369
2	0.425192
3	0.507384
4	0.250702

Implementing a MLP - Let's build the model!

Let's build our model using the Sequential API (there are others but, for now, let's keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Implementing a MLP - Let's build the model!

Let's build our model using the Sequential API (there are others but, for now, let's keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Layers stacked one
over the other!

Implementing a MLP - Let's build the model!

Let's build our model using the Sequential API (there are others but, for now, let's keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Layers stacked one over the other!

Number of neurons in each layer (last one is the output).

Implementing a MLP - Let’s build the model!

Let’s build our model using the Sequential API (there are others but, for now, let’s keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Layers stacked one over the other!

Number of neurons in each layer (last one is the output).

Number of input features.

Implementing a MLP - Let’s build the model!

Let’s build our model using the Sequential API (there are others but, for now, let’s keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Layers stacked one over the other!

Number of neurons in each layer (last one is the output).

Number of input features.

Activation function (here, as an argument to the *build_model* function).

Implementing a MLP - Let’s build the model!

Let’s build our model using the Sequential API (there are others but, for now, let’s keep it simple)!

```
def build_model(activation='relu', learning_rate=0.01):
    #Create a sequential model (with three layers - last one is the output)
    model = Sequential()
    model.add(Dense(16, input_dim=5, activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(1, activation='relu'))

    #Compile the model
    #Define the loss function, the otimizer and metrics to be used
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

After building the stacked MLP, we need to compile the model by setting the loss function (MSE as we are solving a regression problem), the optimizer (which implements the gradient descent and updates the weights), and a set of metrics (to further understand the performance of the model - not used when backpropagating the error).

Implementing a MLP - Train/Test data

Let's use the train_test_split API to hold-out some data for testing! We will use cross validation over the training data!

Test and training data

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled,
                                                    test_size=0.2,
                                                    random_state=RANDOM_SEED)
```

Implementing a MLP - Tuning the network

We want to find the best possible MLP to solve our problem so… Let's tune it (at least, some hyperparameters)!

We must first define a dictionary of {*key -> list of values*}. For now, we will only tune the activation function and the learning rate used by the optimizer (trying two values for each hyperparameter - so, we will fit 4 candidate models). Note that these are arguments of the *build_model*() function.

```
TUNING_DICT = {
    'activation' :      ['relu', 'sigmoid'],
    'learning_rate' :   [0.01, 0.001]
}
```

Implementing a MLP - Tuning the network

Now, let's use the KFold API with *k*=5, the KerasRegressor API to point to our *build_model*() function, and use the GridSearchCV API as we've done before!

```
kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

model = KerasRegressor(build_fn=build_model, epochs=20, batch_size=32)
grid_search = GridSearchCV(estimator = model,
                           param_grid = TUNING_DICT,
                           cv = kf,
                           scoring = 'neg_mean_absolute_error',
                           refit = 'True',
                           verbose = 1)
grid_search.fit(X_train, y_train, validation_split=0.2, verbose=1)
```

Implementing a MLP - Tuning the network

Now, let's use the KFold API with *k*=5, the KerasRegressor API to point to our *build_model*() function, and use the GridSearchCV API as we've done before!

```
kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

model = KerasRegressor(build_fn=build_model, epochs=20, batch_size=32)
grid_search = GridSearchCV(estimator = model,
                           param_grid = TUNING_DICT,
                           cv = kf,
                           scoring = 'neg_mean_absolute_error',
                           refit = 'True',
                           verbose = 1)
grid_search.fit(X_train, y_train, validation_split=0.2, verbose=1)
```

We must pass our build_model function, define the number of epochs (*number of passes of the entire training dataset*) and the batch size (*number of training samples in one forward/backward pass*). The KerasRegressor API will return an instance of our MLP.

Implementing a MLP - Tuning the network

Now, let's use the KFold API with *k*=5, the KerasRegressor API to point to our *build_model*() function, and use the GridSearchCV API as we've done before!

```
kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

model = KerasRegressor(build_fn=build_model, epochs=20, batch_size=32)
grid_search = GridSearchCV(estimator = model,
                           param_grid = TUNING_DICT,
                           cv = kf,
                           scoring = 'neg_mean_absolute_error',
                           refit = 'True',
                           verbose = 1)
grid_search.fit(X_train, y_train, validation_split=0.2, verbose=1)
```

Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and the model metrics on this data at the end of each epoch.

We must pass our build_model function, define the number of epochs (*number of passes of the entire training dataset*) and the batch size (*number of training samples in one forward/backward pass*). The KerasRegressor API will return an instance of our MLP.

Implementing a MLP - Tuning the network

Now, let's use the KFold API with *k*=5, the KerasRegressor API to point to our *build_model*() function, and use the GridSearchCV API as we've done before!

```
kf = KFold(n_splits=5, shuffle=True, random_state=RANDOM_SEED)

model = KerasRegressor(build_fn=build_model, epochs=20, batch_size=32)
grid_search = GridSearchCV(estimator = model,
                           param_grid = TUNING_DICT,
                           cv = kf,
                           scoring = 'neg_mean_absolute_error',
                           refit = 'True',
                           verbose = 1)
grid_search.fit(X_train, y_train, validation_split=0.2, verbose=1)
```

```
100/100 [==============================] - 0s 522us/step - loss: 0.0342 - mae: 0.0342 - mse: 0.0018 - val_loss: 0.0345 - val_
mae: 0.0341 - val_mse: 0.0018
Epoch 20/20
100/100 [==============================] - 0s 520us/step - loss: 0.0335 - mae: 0.0335 - mse: 0.0017 - val_loss: 0.0343 - val_
mae: 0.0343 - val_mse: 0.0018

GridSearchCV(cv=KFold(n_splits=5, random_state=2021, shuffle=True),
             estimator=<tensorflow.python.keras.wrappers.scikit_learn.KerasRegressor object at 0x000001B5463DF748>,
             param_grid={'activation': ['relu', 'sigmoid'],
                         'learning_rate': [0.01, 0.001]},
             refit='True', scoring='neg_mean_absolute_error', verbose=1)
```

Implementing a MLP - Cross Validation results

We can now analyze the performance of our MLP!

```
#summarize results
print("Best: %f using %s" % (grid_search.best_score_, grid_search.best_params_))
means = grid_search.cv_results_['mean_test_score']
stds = grid_search.cv_results_['std_test_score']
params = grid_search.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: -0.033692 using {'activation': 'relu', 'learning_rate': 0.001}
-0.130550 (0.184629) with: {'activation': 'relu', 'learning_rate': 0.01}
-0.033692 (0.000963) with: {'activation': 'relu', 'learning_rate': 0.001}
-0.403154 (0.182433) with: {'activation': 'sigmoid', 'learning_rate': 0.01}
-0.221557 (0.222091) with: {'activation': 'sigmoid', 'learning_rate': 0.001}
```

Our best model!

Implementing a MLP - Overfitting Analysis

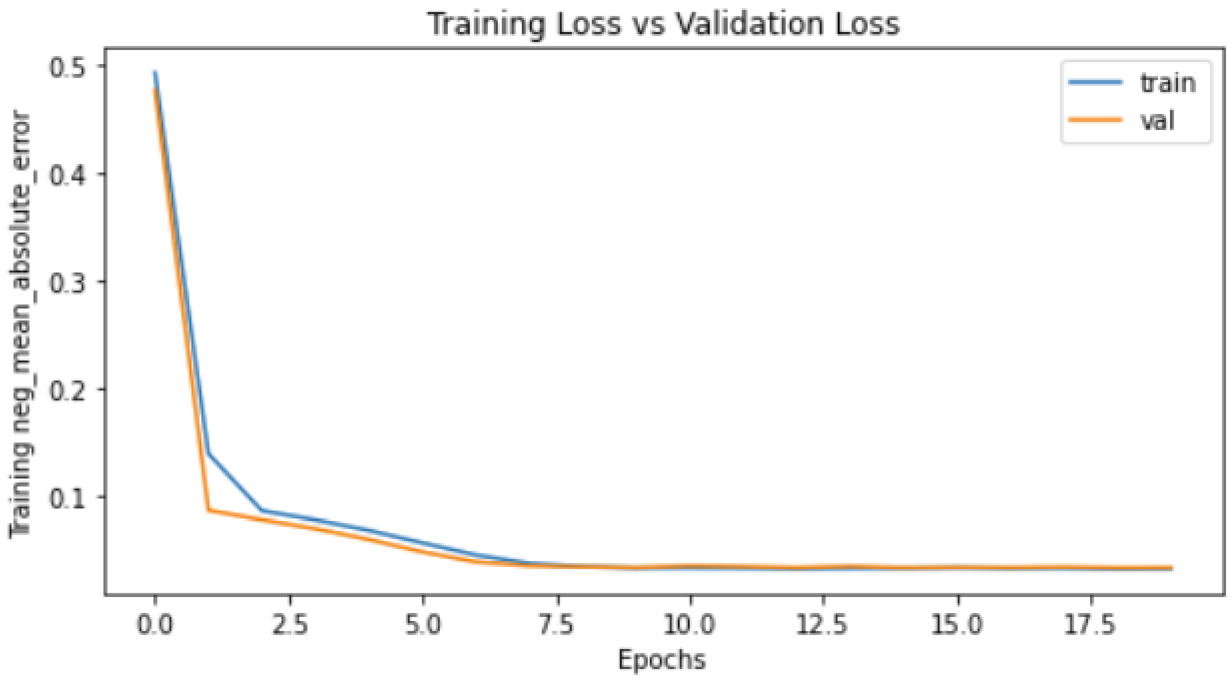
We can also try to understand if our model overfitted!

```
#Our best model (remember we set refit=True?)
best_mlp_model = grid_search.best_estimator_
```

```
#Did the model overfit?
def plot_learning_curve(history, metric='neg_mean_absolute_error'):
    plt.figure(figsize=(8,4))
    plt.title('Training Loss vs Validation Loss')
    plt.plot(history.epoch, history.history['loss'], label='train')
    plt.plot(history.epoch, history.history['val_loss'], label='val')
    plt.ylabel('Training ' + metric)
    plt.xlabel('Epochs')
    plt.legend()

plot_learning_curve(best_mlp_model.model.history, metric='neg_mean_absolute_error')
```

Implementing a MLP - Overfitting Analysis

We can also try to understand if our model overfitted! I guess not!



Implementing a MLP - Performance Analysis

What about MLP predictions?

```
#Obtain predictions
predictions = best_mlp_model.predict(X_test)
predictions = predictions.reshape(predictions.shape[0], 1)
predictions[:5]
```

```
array([[0.62522596],
       [0.35357618],
       [0.27740446],
       [0.6624256 ],
       [0.30916768]], dtype=float32)
```

Yap… Predictions are scaled!!! We can, however, use the *inverse_transform*() function to obtain the real values!

Implementing a MLP - Performance Analysis

What about MLP predictions?

```
#Obtain predictions
predictions = best_mlp_model.predict(X_test)
predictions = predictions.reshape(predictions.shape[0], 1)
predictions[:5]
```

```
array([[0.62522596],
       [0.35357618],
       [0.27740446],
       [0.6624256 ],
       [0.30916768]], dtype=float32)
```

Yap… Predictions are scaled!!! We can, however, use the *inverse_transform*() function to obtain the real values!

```
#And now let's unscale the model's predictions to see real prices!
predictions_unscaled = scaler_y.inverse_transform(predictions)
predictions_unscaled[:5]
```

```
array([[1511417.2 ],
       [ 849883.9 ],
       [ 670154.06],
       [1599453.5 ],
       [ 734485.9 ]], dtype=float32)
```

Implementing a MLP - Performance Analysis

Let's do the same for the y_test data (you can check that you will obtain the original values)!

```
#Let's unscale y_test to get the original values
y_test_unscaled = scaler_y.inverse_transform(y_test)
y_test_unscaled[:5]
```

```
array([[1409892.08977612],
       [ 889385.90158426],
       [ 635429.23051901],
       [1613414.23305073],
       [ 774491.65328819]])
```

```
#And now let's unscale the model's predictions to see real prices!
predictions_unscaled = scaler_y.inverse_transform(predictions)
predictions_unscaled[:5]
```

```
array([[1511417.2 ],
       [ 849883.9 ],
       [ 670154.06],
       [1599453.5 ],
       [ 734485.9 ]], dtype=float32)
```

Not that bad!!

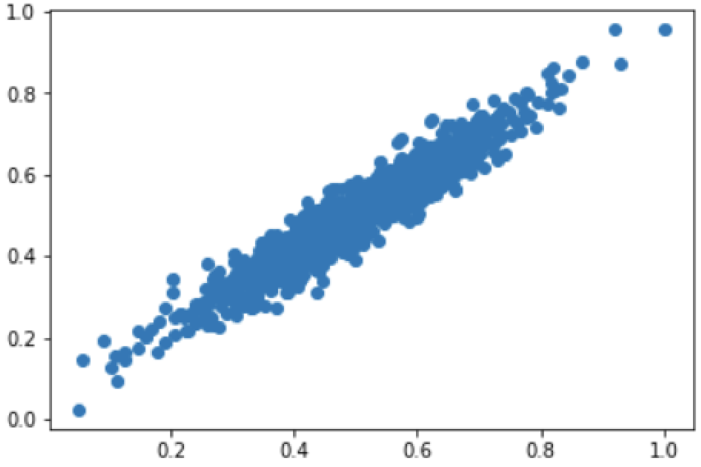
Implementing a MLP - Performance Analysis

Some more metrics (scaled version)…

```
from sklearn import metrics
print('MAE:', metrics.mean_absolute_error(y_test, predictions))
print('MSE:', metrics.mean_squared_error(y_test, predictions))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test, predictions)))
```

```
MAE: 0.033830947270123486
MSE: 0.0017417010324501016
RMSE: 0.041733691814289584
```

```
plt.scatter(y_test, predictions)
```



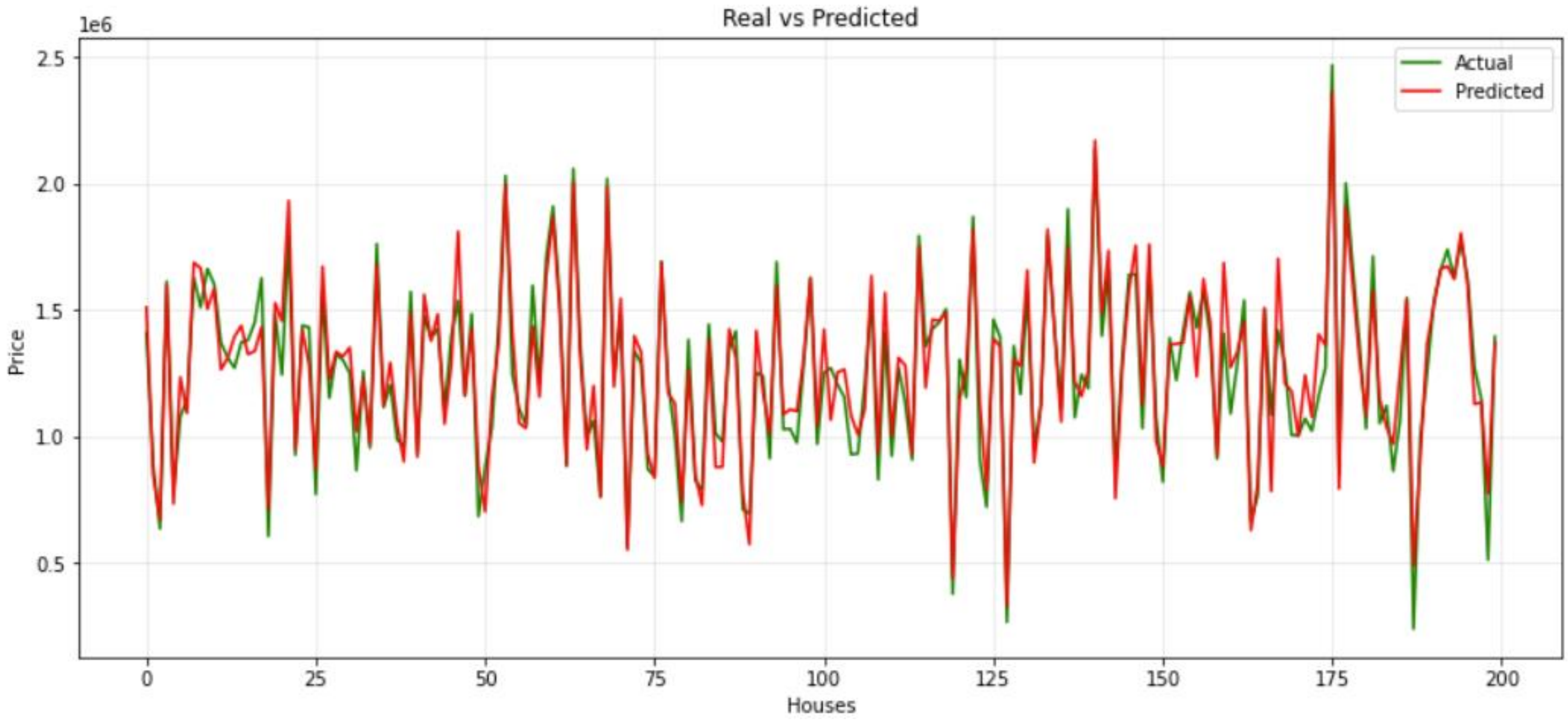
Implementing a MLP - Performance Analysis

Comparing real and predicted values…

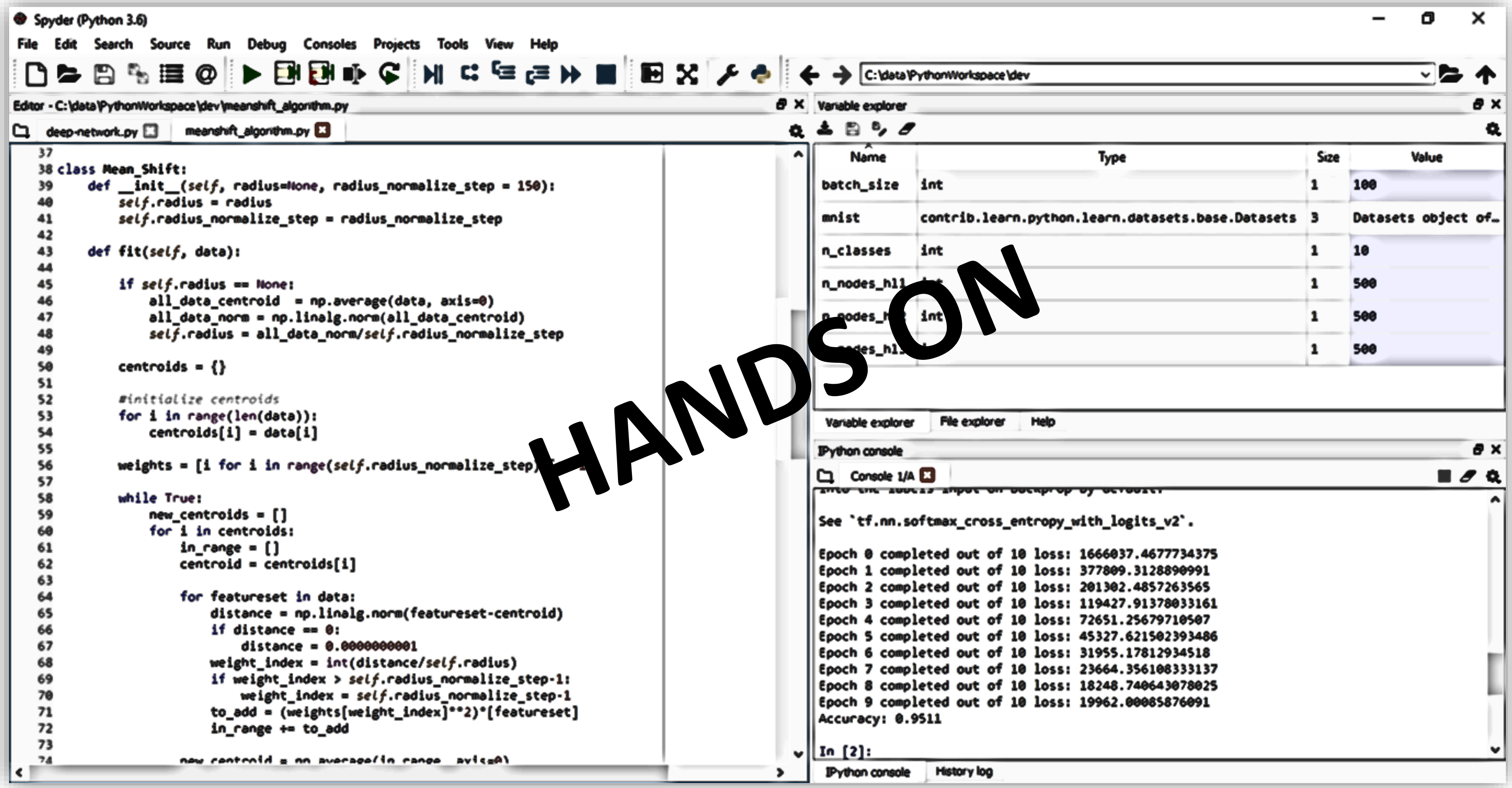
```
#Visualising the actual and predicted result
def real_predicted_viz(limit):
    plt.figure(figsize=(14,6))
    plt.plot(y_test_unscaled[:limit], color = 'green', label = 'Actual')
    plt.plot(predictions_unscaled[:limit], color = 'red', label = 'Predicted')
    plt.grid(alpha = 0.3)
    plt.xlabel('Houses')
    plt.ylabel('Price')
    plt.title('Real vs Predicted')
    plt.legend()
    plt.show()

#let's limit to 200 comparasions for better visualization
real_predicted_viz(200)
```

Implementing a MLP - Performance Analysis

Comparing real and predicted values…



Hands On


HANDS ON
Spyder (Python 3.6)
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
Variable explorer
IPython console