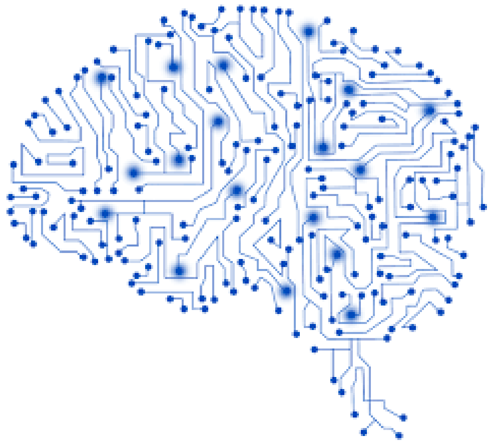
University of Minho
School of Engineering



Dados e Aprendizagem Automática

Artificial Neural Networks:

Multilayer Perceptron

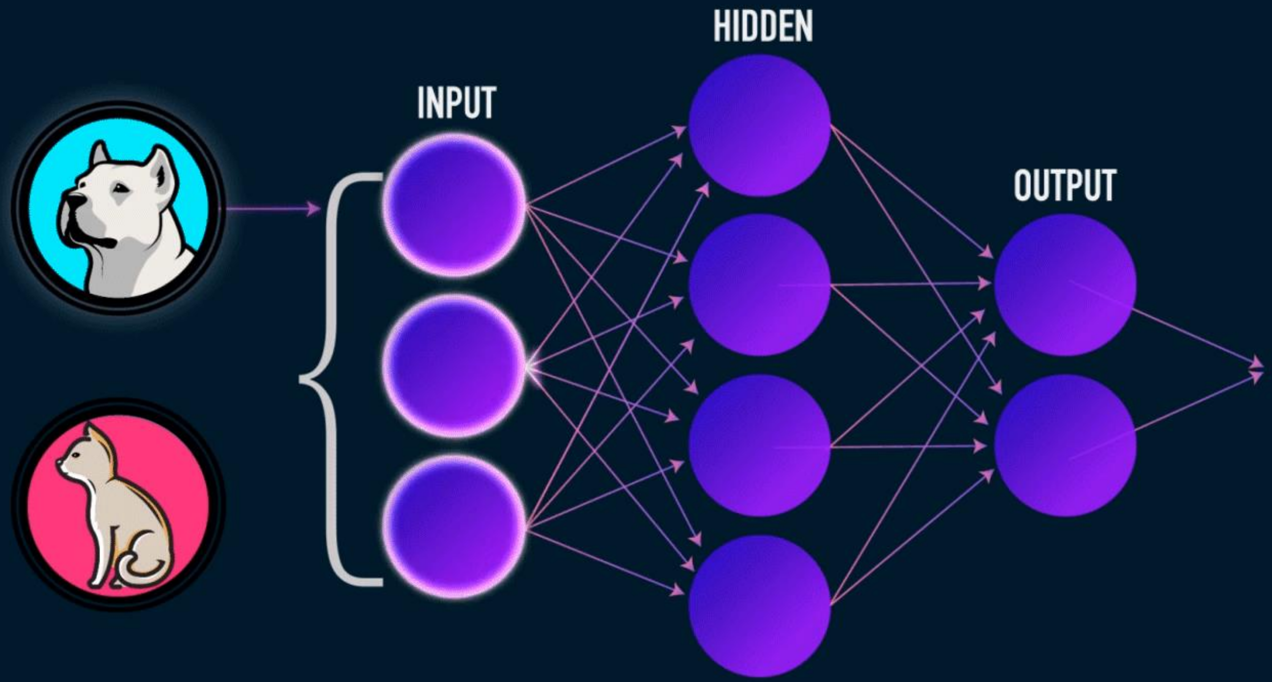
DAA @ MEI-1º/MiEI-4º – 1º Semestre

Bruno Fernandes, César Analide, Dalila Alves, Filipa Ferraz, Victor Alves

Part VIII

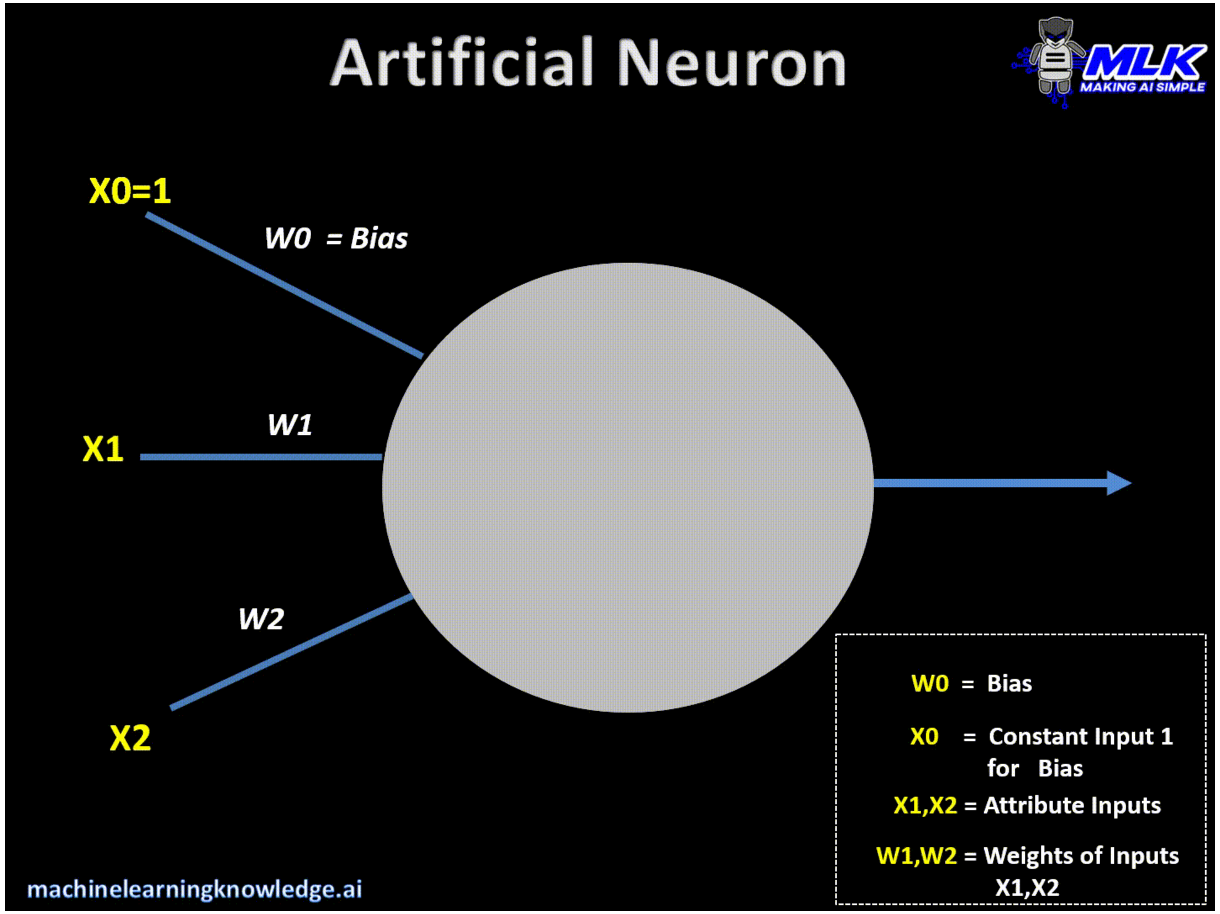
Contents

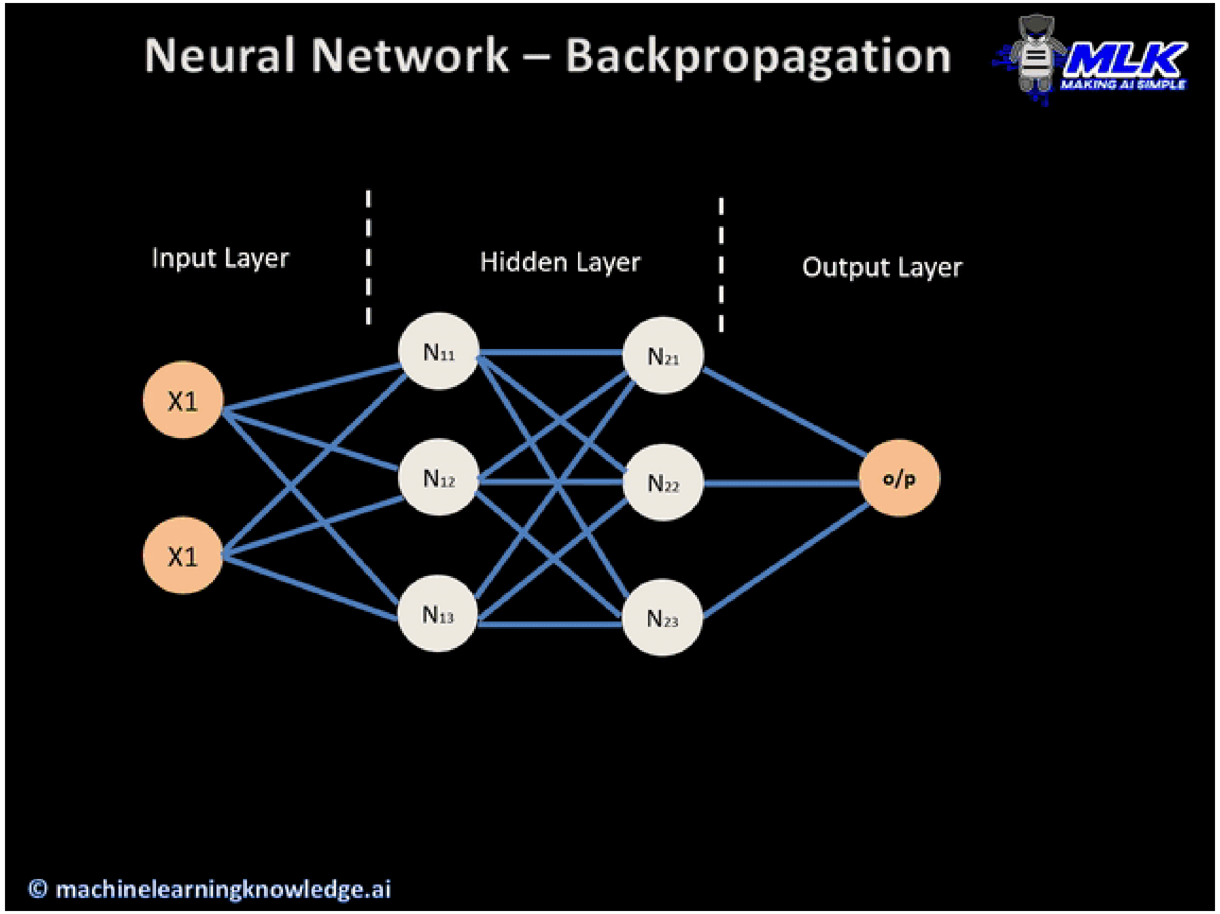
Artificial Neural Networks



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

Artificial Neural Networks

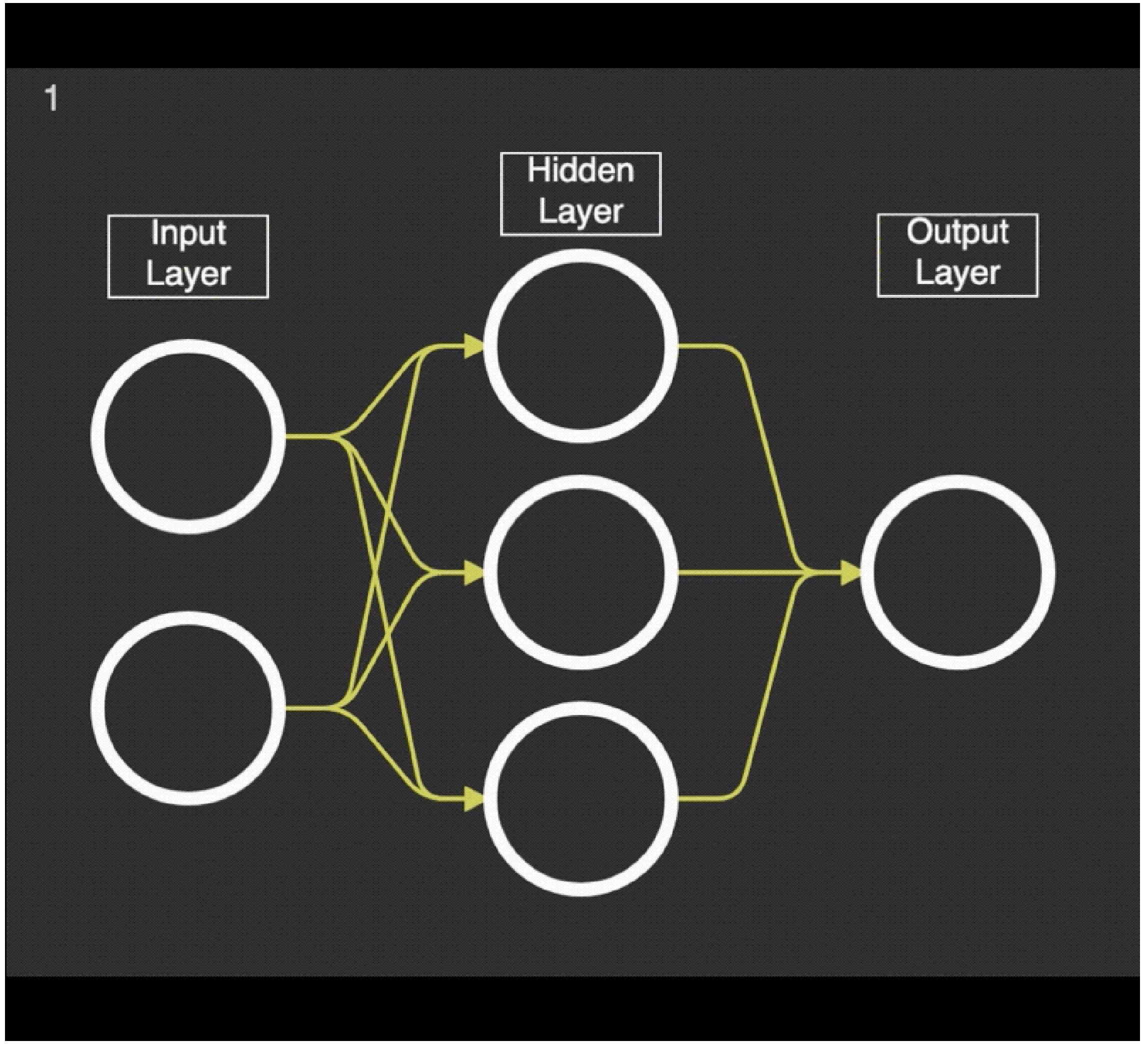




(https://machinelearningknowledge.ai/wp-content/uploads/2019/06/Artificial-Neuron-Working.gif)

(https://machinelearningknowledge.ai/wp-content/uploads/2019/10/Backpropagation.gif)

Artificial Neural Networks



(https://mlfromscratch.com/content/images/size/w2000/2019/12/nn2.png)

Pratical Application of Artificial Neural Networks

For this example, we will use the already known dataset about USA housing. This is the real estate problem, and our goal is to help the real estate agent predict housing prices for regions in the USA (dataset here).

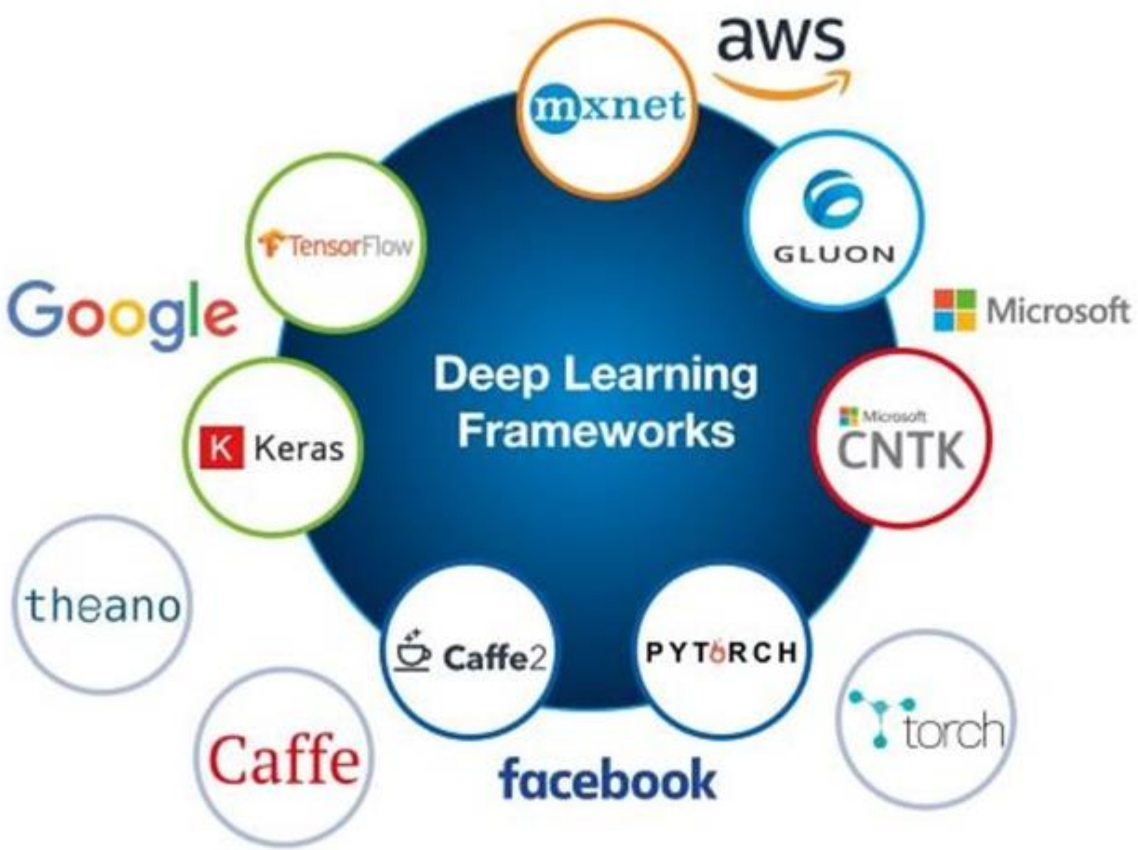
We have used Linear Regression in this context; now, we are going to try Artificial Neural Networks. Let's try using Multilayer Perceptrons (MLPs), a class of Artificial Neural Networks!

To implement our first Artificial Neural Network, we will use:

TensorFlow™

Remembering Frameworks

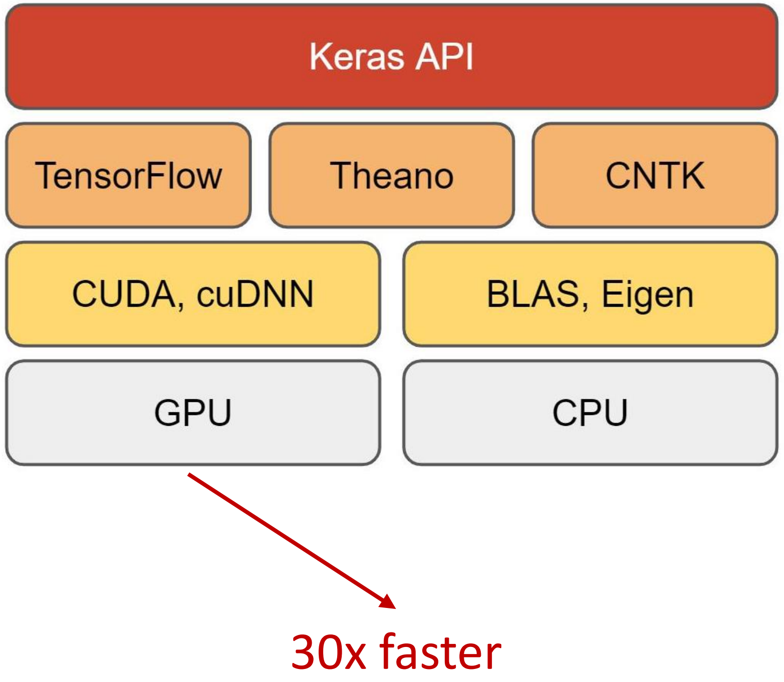
Theano	Python library that allows you to define, optimize and calculate mathematical expressions with multidimensional arrays efficiently. It works with GPUs and efficiently performs differential calculations. *University of Montreal's lab, MLA*
Lasagne	Light library for building and training neural networks using Theano
Blocks	Theano-based framework for building and training neural networks
TensorFlow	Open-source library for numerical computation using graphs *Google Brain team*
Keras	Deep learning library for Python. Runs on Theano or TensorFlow
MXNet	Deep learning framework designed for efficiency and flexibility *Amazon*
PyTorch	Flexible tensors and neural networks with strong GPU support *Facebook Artificial Intelligence Research team (FAIR)*
Torch	*Ronan Collobert*
Caffe	*Berkeley Vision and Learning Center*
CNTK	*Microsoft*
Deeplearning4j	*Skymind*



TensorFlow

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



Implementing a Multilayer Perceptron

The data

It will be used data frame with 5000 observations on the following 7 variables:

- **Avg. Area Income** - Avg. Income of residents of the city house is located in.
- **Avg. Area House Age** - Avg Age of Houses in same city
- **Avg. Area Number of Rooms** - Avg Number of Rooms for Houses in same city
- **Avg. Area Number of Bedrooms** - Avg Number of Bedrooms for Houses in same city
- **Area Population** - Population of city house is located in
- **Price** - Price that the house sold at
- **Address** - Address for the house

Implementing a Multilayer Perceptron

Import libraries

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

Get the data

Create the data frame

```
USAhousing = pd.read_csv('USA_Housing.csv')
```

```
USAhousing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
 #   Column                        Non-Null Count  Dtype  
---  ------                        --------------  -----  
 0   Avg. Area Income              5000 non-null   float64
 1   Avg. Area House Age           5000 non-null   float64
 2   Avg. Area Number of Rooms     5000 non-null   float64
 3   Avg. Area Number of Bedrooms  5000 non-null   float64
 4   Area Population               5000 non-null   float64
 5   Price                         5000 non-null   float64
 6   Address                       5000 non-null   object 
dtypes: float64(6), object(1)
memory usage: 273.6+ KB
```

Implementing a Multilayer Perceptron

Since the feature *Address* is the one categoric and not needed for the purpose of the exercise, let's drop it:

```
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 Multilayer Perceptron


```
USAhousing.describe()
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5.000000e+03
mean	68583.108984	5.977222	6.987792	3.981330	36163.516039	1.232073e+06
std	10657.991214	0.991456	1.005833	1.234137	9925.650114	3.531176e+05
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04
25%	61480.562388	5.322283	6.299250	3.140000	29403.928702	9.975771e+05
50%	68804.286404	5.970429	7.002902	4.050000	36199.406689	1.232669e+06
75%	75783.338666	6.650808	7.665871	4.490000	42861.290769	1.471210e+06
max	107701.748378	9.519088	10.759588	6.500000	69621.713378	2.469066e+06

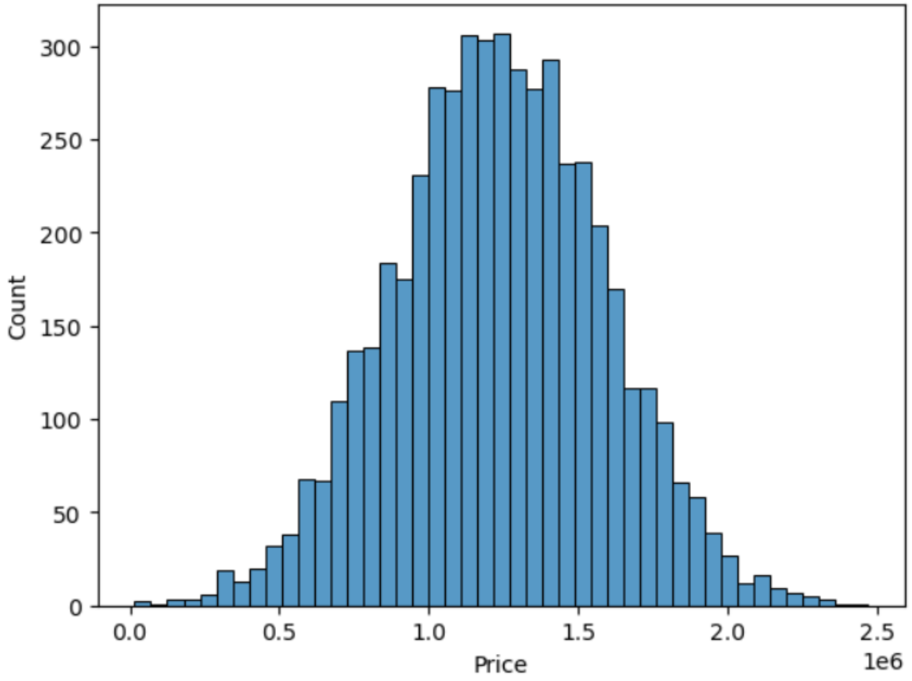
Implementing a Multilayer Perceptron

EDA

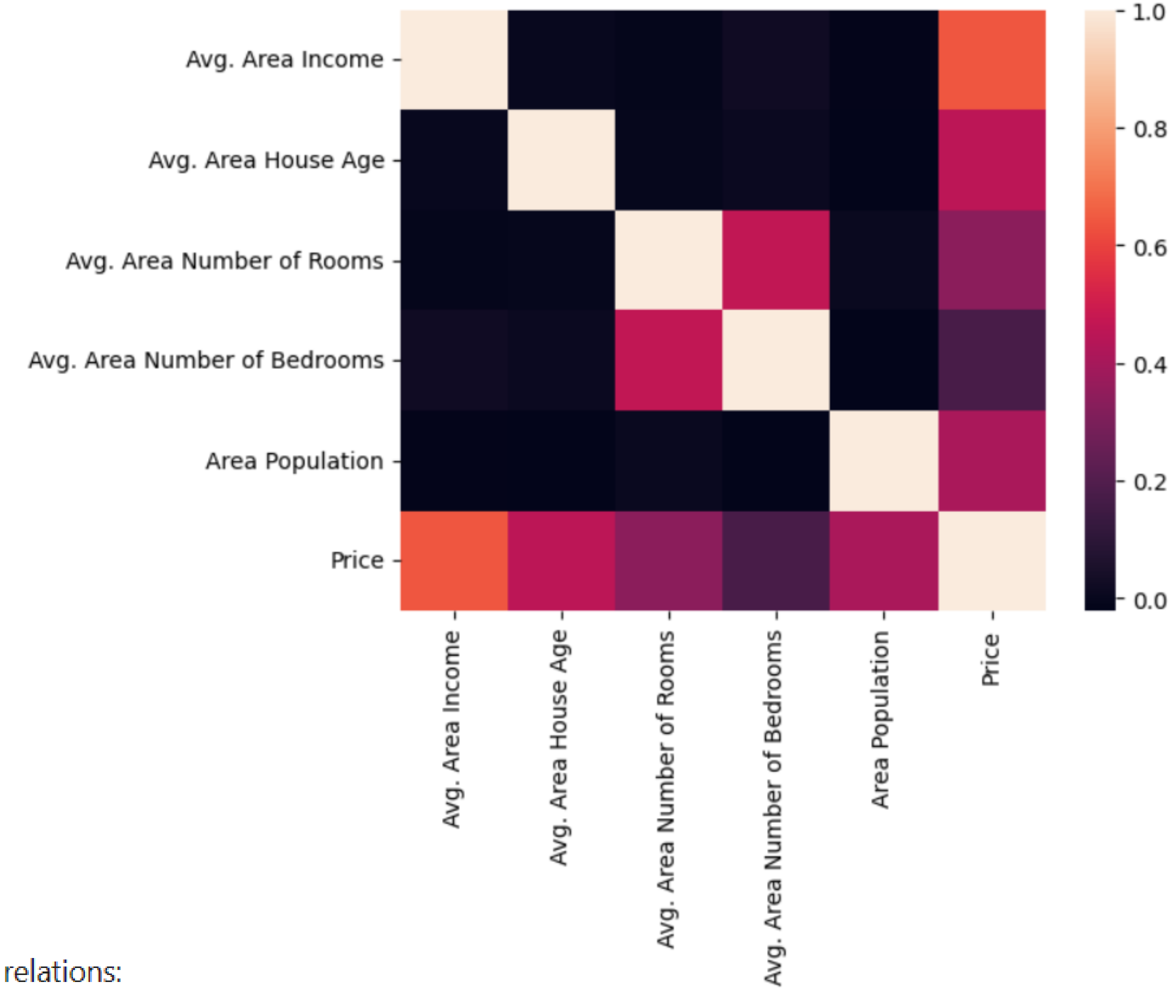
Create the histogram of the target - column *Price*:

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

```
<Axes: xlabel='Price', ylabel='Count'>
```



Create a heatmap of the features:



Create a pairplot to visualize relations:

Implementing a Multilayer Perceptron

Train Test Split

Define *X* and *Y*:

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

Divide the subsets of test and training data:

```
from sklearn.model_selection import GridSearchCV, KFold, train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2021)
```

Implementing a Multilayer Perceptron

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

`conda install -c conda-forge tensorflow`	or	`pip install tensorflow`
`conda install -c conda-forge keras`	or	`pip install –upgrade keras`
`conda install -c conda-forge scikeras`	or	`pip install scikeras[tensorflow]`

Import more libraries

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from scikeras.wrappers import KerasRegressor
```

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

```
TensorFlow version: 2.14.0
```

Implementing a Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Implementing a Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```python
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    model.compile(
        loss = 'mae',
        optimizer = tf.optimizers.Adam(learning_rate),
        metrics = ['mae', 'mse'])
    return model
```

Layers stacked one over the other

Implementing a Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    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 Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    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 Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    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 Multilayer Perceptron

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

Structure the MLP

Define a model with:

- *ReLu* as activation function
- sequential topology
- three layers
- *MAE* as loss function
- *Adam* as optimizer
- learning rate of *0.01*
- *MAE* and *MSE* as metrics

```
def build_model(activation = 'relu', learning_rate = 0.01):
    model = Sequential()
    model.add(Dense(16, input_dim = 5, activation = activation))
    model.add(Dense(8, activation = activation))
    model.add(Dense(1, activation = activation)) #output

    #Compile the model
    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 (MAE 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 Multilayer Perceptron

Build the model:

```
model = build_model()
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                96

 dense_1 (Dense)             (None, 8)                 136

 dense_2 (Dense)             (None, 1)                 9

=================================================================
Total params: 241 (964.00 Byte)
Trainable params: 241 (964.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Implementing a Multilayer Perceptron

GridSearchCV

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 optimizer (trying three values - so, we will fit 3 candidate models). Note that these are arguments of the *build_model*() function.

Define the grid parameters using a dictionary:

```
optimizer = ['SGD', 'RMSprop', 'Adagrad']
param_grid = dict(optimizer = optimizer)
```

Define a *KFold* with *5 splits*, *shuffle* and a *random state*:

```
kf = KFold(n_splits = 5, shuffle = True, random_state = 2021)
```

Use a *KerasRegressor* with a *batch size* of *32*, *validation split* of *0.2* and *20 epochs*:

```
model = KerasRegressor(model = build_model, batch_size = 32, validation_split = 0.2, epochs = 20)
```

Compute a *GridSearchCV* with *NegMAE scoring*, *refit* and a *verbose* of *1*:

```
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = kf, scoring = 'neg_mean_absolute_error', refit = 'True', verbose = 1)
```

Implementing a Multilayer Perceptron

GridSearchCV

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 optimizer (trying three values - so, we will fit 3 candidate models). Note that these are arguments of the *build_model*() function.

Define the grid parameters using a dictionary:

```
optimizer = ['SGD', 'RMSprop', 'Adagrad']
param_grid = dict(optimizer = optimizer)
```

Define a *KFold* with *5 splits*, *shuffle* and a *random state*:

```
kf = KFold(n_splits = 5, shuffle = True, random_state = 2021)
```

Use a *KerasRegressor* with a *batch size* of *32*, *validation split* of *0.2* and *20 epochs*:

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.

```
model = KerasRegressor(model = build_model, batch_size = 32, validation_split = 0.2, epochs = 20)
```

Compute a *GridSearchCV* with *NegMAE scoring*, *refit* and a *verbose* of *1*:

```
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = kf, scoring = 'neg_mean_absolute_error', refit = 'True', verbose = 1)
```

Implementing a Multilayer Perceptron

GridSearchCV

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 optimizer (trying three values - so, we will fit 3 candidate models). Note that these are arguments of the *build_model*() function.

Define the grid parameters using a dictionary:

```
optimizer = ['SGD', 'RMSprop', 'Adagrad']
param_grid = dict(optimizer = optimizer)
```

Define a *KFold* with *5 splits*, *shuffle* and a *random state*:

```
kf = KFold(n_splits = 5, shuffle = True, random_state = 2021)
```

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.

Use a *KerasRegressor* with a *batch size* of *32*, *validation split* of *0.2* and *20 epochs*:

```
model = KerasRegressor(model = build_model, batch_size = 32, validation_split = 0.2, epochs = 20)
```

Compute a *GridSearchCV* with *NegMAE scoring*, *refit* and a *verbose* of *1*:

```
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = kf, scoring = 'neg_mean_absolute_error', refit = 'True', verbose = 1)
```

Implementing a Multilayer Perceptron

GridSearchCV

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 optimizer (trying three values - so, we will fit 3 candidate models). Note that these are arguments of the *build_model*() function.

Define the grid parameters using a dictionary:

```
optimizer = ['SGD', 'RMSprop', 'Adagrad']
param_grid = dict(optimizer = optimizer)
```

Define a *KFold* with *5 splits*, *shuffle* and a *random state*:

```
kf = KFold(n_splits = 5, shuffle = True, random_state = 2021)
```

Use a *KerasRegressor* with a *batch size* of *32*, *validation split* of *0.2* and *20 epochs*:

```
model = KerasRegressor(model = build_model, batch_size = 32, validation_split = 0.2, epochs = 20)
```

Compute a *GridSearchCV* with *NegMAE scoring*, *refit* and a *verbose* of *1*:

```
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = kf, scoring = 'neg_mean_absolute_error', refit = 'True', verbose = 1)
```

Implementing a Multilayer Perceptron

Fit the model:

```
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
Epoch 1/20
80/80 [==============================] - 2s 8ms/step - loss: 1227161.2500 - mae: 1227161.2500 - mse: 1634915057664.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 2/20
80/80 [==============================] - 0s 4ms/step - loss: 1227161.1250 - mae: 1227161.1250 - mse: 1634914664448.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 3/20
80/80 [==============================] - 0s 4ms/step - loss: 1227161.1250 - mae: 1227161.1250 - mse: 1634914795520.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 4/20
80/80 [==============================] - 0s 3ms/step - loss: 1227161.3750 - mae: 1227161.3750 - mse: 1634914795520.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 5/20
80/80 [==============================] - 0s 3ms/step - loss: 1227161.2500 - mae: 1227161.2500 - mse: 1634914402304.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 6/20
80/80 [==============================] - 0s 3ms/step - loss: 1227161.2500 - mae: 1227161.2500 - mse: 1634914795520.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 7/20
80/80 [==============================] - 0s 4ms/step - loss: 1227161.5000 - mae: 1227161.5000 - mse: 1634914533376.0000
- val_loss: 1247080.1250 - val_mae: 1247080.1250 - val_mse: 1679838281728.0000
Epoch 8/20
```

…

Implementing a Multilayer Perceptron

We can now analyze the performance of our MLP:

```
print("Best: %f using %s" % (grid_search.best_score_, grid_search.best_params_))
```

```
Best: -202760.793380 using {'optimizer': 'Adagrad'}
```

The best model

Find the *mean test score*, *std test score* and *params* for each search:

```
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))
```

```
-614376.884116 (499562.792066) with: {'optimizer': 'SGD'}
-408853.059894 (411606.908481) with: {'optimizer': 'RMSprop'}
-202760.793380 (1187.979771) with: {'optimizer': 'Adagrad'}
```

Implementing a Multilayer Perceptron


```
best_mlp_model = grid_search.best_estimator_
print(best_mlp_model)
```

```
KerasRegressor(
	model=<function build_model at 0x000002E011306660>
	build_fn=None
	warm_start=False
	random_state=None
	optimizer=Adagrad
	loss=None
	metrics=None
	batch_size=32
	validation_batch_size=None
	verbose=1
	callbacks=None
	validation_split=0.2
	shuffle=True
	run_eagerly=False
	epochs=20
)
```

Implementing a Multilayer Perceptron

Let’s try to understand if our model overfitted:

```
best_mlp_model.fit(X_train, y_train, epochs = 20, validation_data = (X_test, y_test), verbose = 1)
```

```
Epoch 1/20
125/125 [==============================] - 2s 5ms/step - loss: 561256.3125 - mae: 561256.3125 - mse: 559574679552.000
0 - val_loss: 195579.8438 - val_mae: 195579.8438 - val_mse: 59914842112.0000
Epoch 2/20
125/125 [==============================] - 0s 3ms/step - loss: 203607.3906 - mae: 203607.3906 - mse: 64345059328.0000
- val_loss: 198204.0000 - val_mae: 198204.0000 - val_mse: 61535883264.0000
Epoch 3/20
125/125 [==============================] - 0s 3ms/step - loss: 203205.4844 - mae: 203205.4844 - mse: 63991103488.0000
- val_loss: 195700.2344 - val_mae: 195700.2344 - val_mse: 60022448128.0000
Epoch 4/20
125/125 [==============================] - 0s 3ms/step - loss: 202106.3750 - mae: 202106.3750 - mse: 63488376832.0000
- val_loss: 198870.7969 - val_mae: 198870.7969 - val_mse: 61997703168.0000
Epoch 5/20
125/125 [==============================] - 0s 3ms/step - loss: 204026.1250 - mae: 204026.1250 - mse: 64430120960.0000
- val_loss: 196261.9844 - val_mae: 196261.9844 - val_mse: 60319195136.0000
Epoch 6/20
125/125 [==============================] - 0s 3ms/step - loss: 203777.8438 - mae: 203777.8438 - mse: 64410632192.0000
- val_loss: 195595.4062 - val_mae: 195595.4062 - val_mse: 59969236992.0000
Epoch 7/20
125/125 [==============================] - 0s 4ms/step - loss: 203254.9219 - mae: 203254.9219 - mse: 64242946048.0000
- val_loss: 198508.5781 - val_mae: 198508.5781 - val_mse: 61783752704.0000
Epoch 8/20
125/125 [==============================] - 0s 3ms/step - loss: 203353.2812 - mae: 203353.2812 - mse: 63824674816.0000
- val_loss: 197179.8750 - val_mae: 197179.8750 - val_mse: 60830883840.0000
...
```

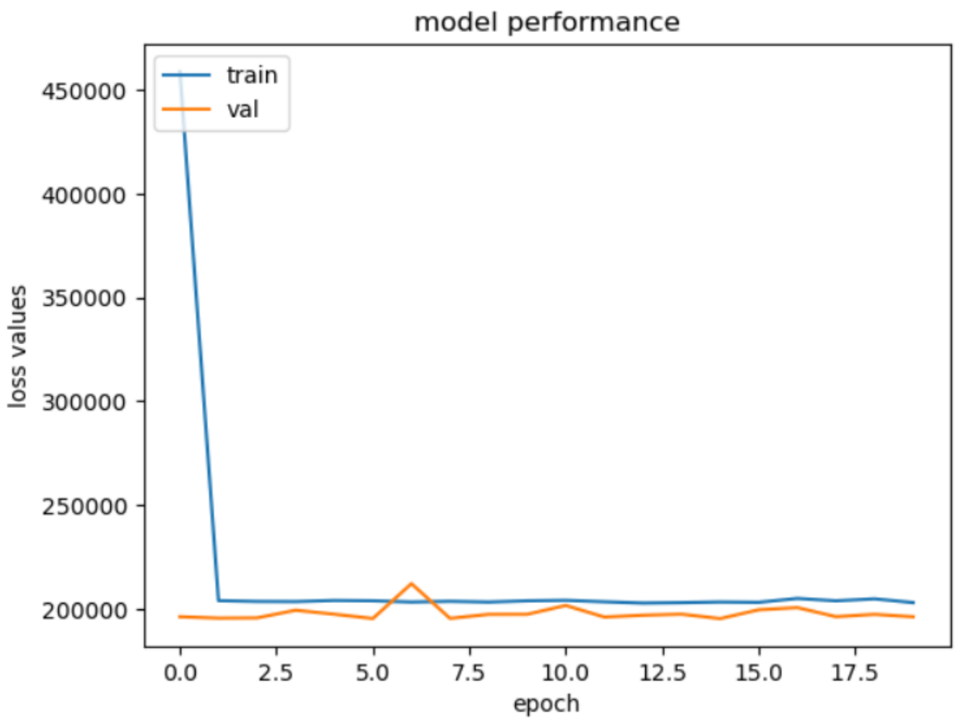
Implementing a Multilayer Perceptron

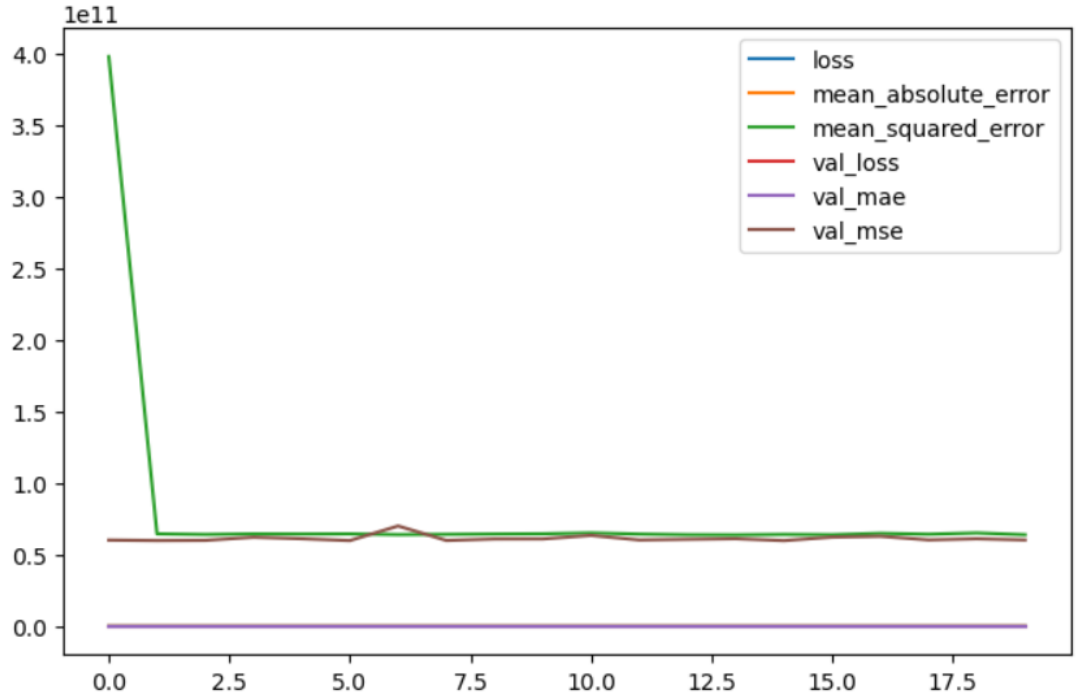
Let's plot it:

```
plt.plot(best_mlp_model.history_['loss'])
plt.plot(best_mlp_model.history_['val_loss'])
plt.title('model performance')
plt.ylabel('loss values')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Did the model overfit?

```
pd.DataFrame(best_mlp_model.history_).plot(figsize = (8, 5))
plt.show()
```





Implementing a Multilayer Perceptron

Predictions

Obtain the predictions:

```
predictions = best_mlp_model.predict(X_test)
```

```
32/32 [==============================] - 0s 2ms/step
```

Print the first five:

```
array([[1394124. ],
       [1224608.1],
       [1049303.8],
       [1335269.2],
       [1235241. ]], dtype=float32)
```

Evaluate the model

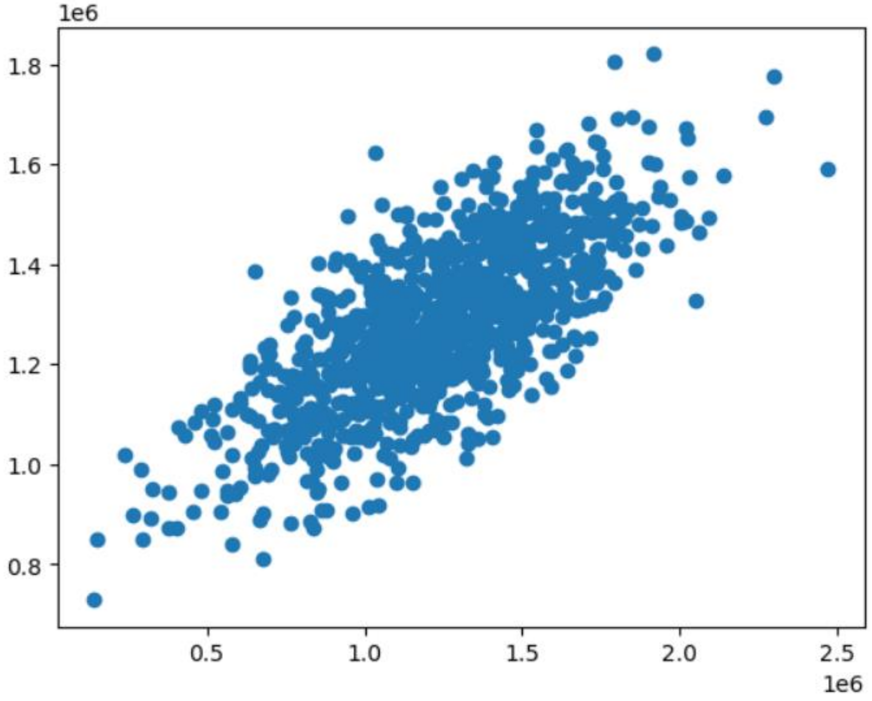
```
from sklearn import metrics
```

```
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: 197376.8433273626
MSE: 61088757946.53201
RMSE: 247161.40059995616
```

Scatter the real values with the predictions:

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

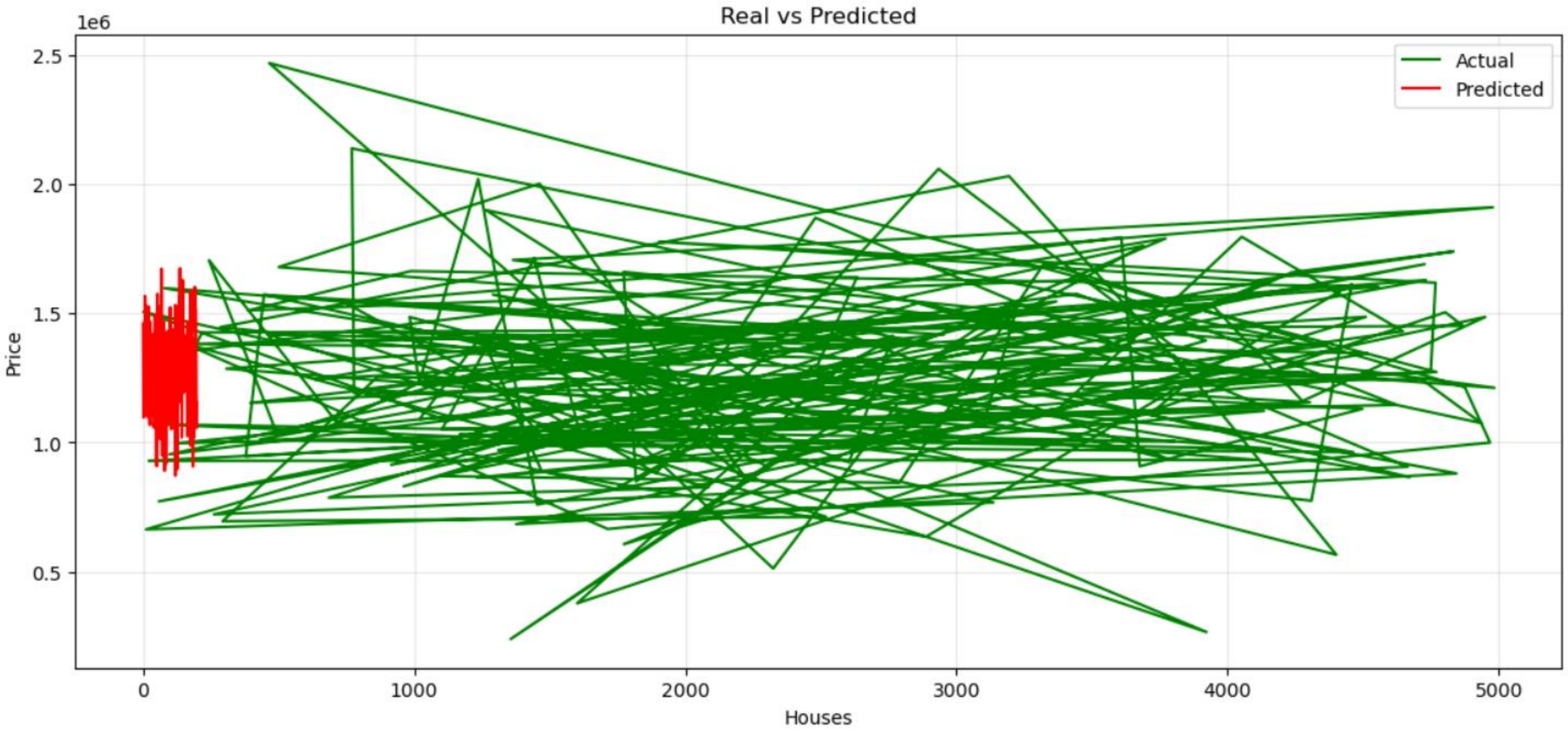


Implementing a Multilayer Perceptron

Create a visualization of the actual and predicted results. Limit it to 200 comparisons:

```
def real_predicted_viz(limit):
    plt.figure(figsize = (14, 6))
    plt.plot(y_test[:limit], color = 'green', label = 'Actual')
    plt.plot(predictions[: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()
```

```
real_predicted_viz(200)
```



Implementing a Multilayer Perceptron

Data scaling

Data scaling or **normalization** is a process of making model data in a standard format so that the training is improved, accurate, and faster.

```
USAhousing.describe()
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5.000000e+03
mean	68583.108984	5.977222	6.987792	3.981330	36163.516039	1.232073e+06
std	10657.991214	0.991456	1.005833	1.234137	9925.650114	3.531176e+05
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04
25%	61480.562388	5.322283	6.299250	3.140000	29403.928702	9.975771e+05
50%	68804.286404	5.970429	7.002902	4.050000	36199.406689	1.232669e+06
75%	75783.338666	6.650808	7.665871	4.490000	42861.290769	1.471210e+06
max	107701.748378	9.519088	10.759588	6.500000	69621.713378	2.469066e+06

Implementing a Multilayer Perceptron

Artificial neural networks are "picky" - they prefer scaled data! Therefore, and since our data have a large variation of values, let's scale the data to be in the interval between [0, 1]:

```
from sklearn.preprocessing import MinMaxScaler
```

```
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)
```

Let’s check the data before and after transformations.

Implementing a Multilayer Perceptron


```
X.head()
```

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

And now scaled:

```
X_scaled.head()
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population
0	0.686822	0.441986	0.501502	0.464444	0.329942
1	0.683521	0.488538	0.464501	0.242222	0.575968
2	0.483737	0.468609	0.701350	0.695556	0.528582
3	0.506630	0.660956	0.312430	0.280000	0.491549
4	0.469223	0.348556	0.611851	0.495556	0.376988

```
y.head()
```

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

And now scaled:

```
y_scaled.head()
```

	Price
0	0.425210
1	0.607369
2	0.425192
3	0.507384
4	0.250702

Implementing a Multilayer Perceptron

We need to split the training and test sets again:

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size = 0.2, random_state = 2021)
```

And then recall the MLP:

```
model = build_model()
model.summary()
```

```
Model: "sequential_18"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_54 (Dense)            (None, 16)                96

 dense_55 (Dense)            (None, 8)                 136

 dense_56 (Dense)            (None, 1)                 9

=================================================================
Total params: 241 (964.00 Byte)
Trainable params: 241 (964.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
model = KerasRegressor(model = build_model, batch_size = 32, validation_split = 0.2, epochs = 20)
```

```
grid_search = GridSearchCV(estimator = model, param_grid = param_grid, cv = kf, scoring = 'neg_mean_absolute_error', refit = 'True', verbose = 1)
```

Implementing a Multilayer Perceptron

Fit the model:

```
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
Epoch 1/20
80/80 [==============================] - 2s 7ms/step - loss: 0.0902 - mae: 0.0902 - mse: 0.0148 - val_loss: 0.0546 - val_mae: 0.0546
- val_mse: 0.0048
Epoch 2/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0416 - mae: 0.0416 - mse: 0.0028 - val_loss: 0.0344 - val_mae: 0.0344
- val_mse: 0.0018
Epoch 3/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0353 - mae: 0.0353 - mse: 0.0019 - val_loss: 0.0466 - val_mae: 0.0466
- val_mse: 0.0032
Epoch 4/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0363 - mae: 0.0363 - mse: 0.0021 - val_loss: 0.0352 - val_mae: 0.0352
- val_mse: 0.0019
Epoch 5/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0370 - mae: 0.0370 - mse: 0.0021 - val_loss: 0.0359 - val_mae: 0.0359
- val_mse: 0.0019
Epoch 6/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0391 - mae: 0.0391 - mse: 0.0024 - val_loss: 0.0345 - val_mae: 0.0345
- val_mse: 0.0018
Epoch 7/20
80/80 [==============================] - 0s 4ms/step - loss: 0.0346 - mae: 0.0346 - mse: 0.0019 - val_loss: 0.0351 - val_mae: 0.0351
- val_mse: 0.0019
Epoch 8/20
...
```

Implementing a Multilayer Perceptron

Find the *best score* and the *best params*:

```
print("Best: %f using %s" % (grid_search.best_score_, grid_search.best_params_))
```

```
Best: -0.219534 using {'optimizer': 'SGD'}
```

Find the *mean test score*, *std test score* and *params* for each search:

```
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))
```

```
-0.219534 (0.225235) with: {'optimizer': 'SGD'}
-0.220775 (0.226018) with: {'optimizer': 'RMSprop'}
-0.310961 (0.225857) with: {'optimizer': 'Adagrad'}
```

```
best_mlp_model_2 = grid_search.best_estimator_
print(best_mlp_model_2)
```

```
KerasRegressor(
	model=<function build_model at 0x000002E011306660>
	build_fn=None
	warm_start=False
	random_state=None
	optimizer=RMSprop
	loss=None
	metrics=None
	batch_size=32
	validation_batch_size=None
	verbose=1
	callbacks=None
	validation_split=0.2
	shuffle=True
	run_eagerly=False
	epochs=20
)
```

Implementing a Multilayer Perceptron

Fit the best model:

```
best_mlp_model_2.fit(X_train, y_train, epochs = 20, validation_data = (X_test, y_test), verbose = 1)
```

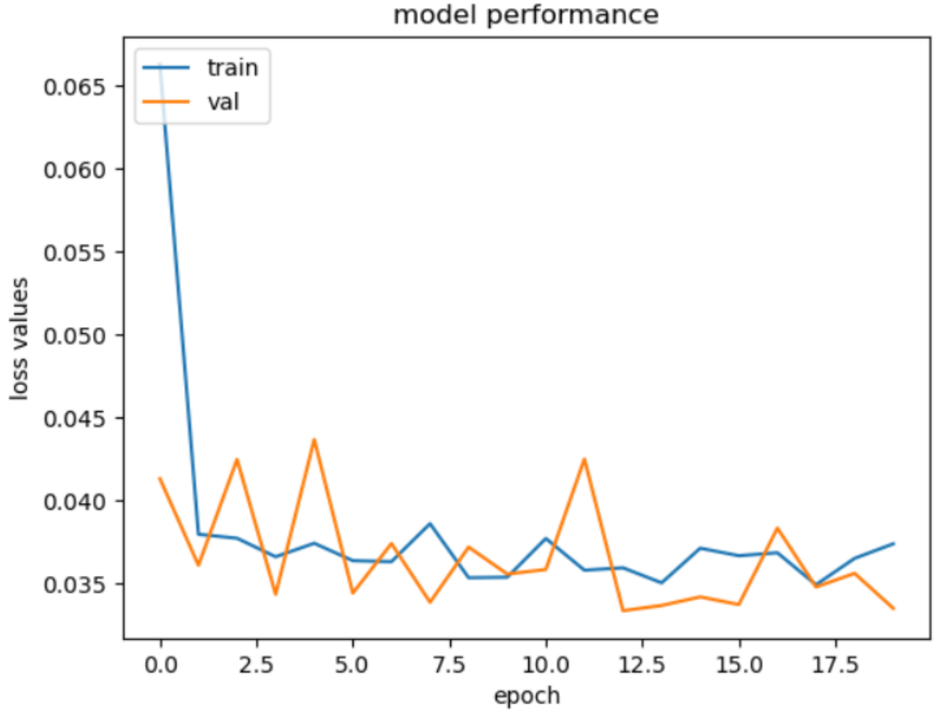
```
Epoch 1/20
125/125 [==============================] - 2s 6ms/step - loss: 0.0601 - mae: 0.0601 - mse: 0.0064 - val_loss: 0.0341 - val_mae: 0.03
41 - val_mse: 0.0018
Epoch 2/20
125/125 [==============================] - 0s 4ms/step - loss: 0.0400 - mae: 0.0400 - mse: 0.0025 - val_loss: 0.0339 - val_mae: 0.03
39 - val_mse: 0.0018
Epoch 3/20
125/125 [==============================] - 1s 4ms/step - loss: 0.0400 - mae: 0.0400 - mse: 0.0025 - val_loss: 0.0497 - val_mae: 0.04
97 - val_mse: 0.0036
Epoch 4/20
125/125 [==============================] - 0s 4ms/step - loss: 0.0365 - mae: 0.0365 - mse: 0.0020 - val_loss: 0.0339 - val_mae: 0.03
39 - val_mse: 0.0018
Epoch 5/20
125/125 [==============================] - 0s 4ms/step - loss: 0.0354 - mae: 0.0354 - mse: 0.0019 - val_loss: 0.0346 - val_mae: 0.03
46 - val_mse: 0.0019
Epoch 6/20
125/125 [==============================] - 1s 4ms/step - loss: 0.0357 - mae: 0.0357 - mse: 0.0020 - val_loss: 0.0335 - val_mae: 0.03
35 - val_mse: 0.0017
Epoch 7/20
125/125 [==============================] - 0s 4ms/step - loss: 0.0371 - mae: 0.0371 - mse: 0.0021 - val_loss: 0.0334 - val_mae: 0.03
34 - val_mse: 0.0017
Epoch 8/20
```

…

Implementing a Multilayer Perceptron

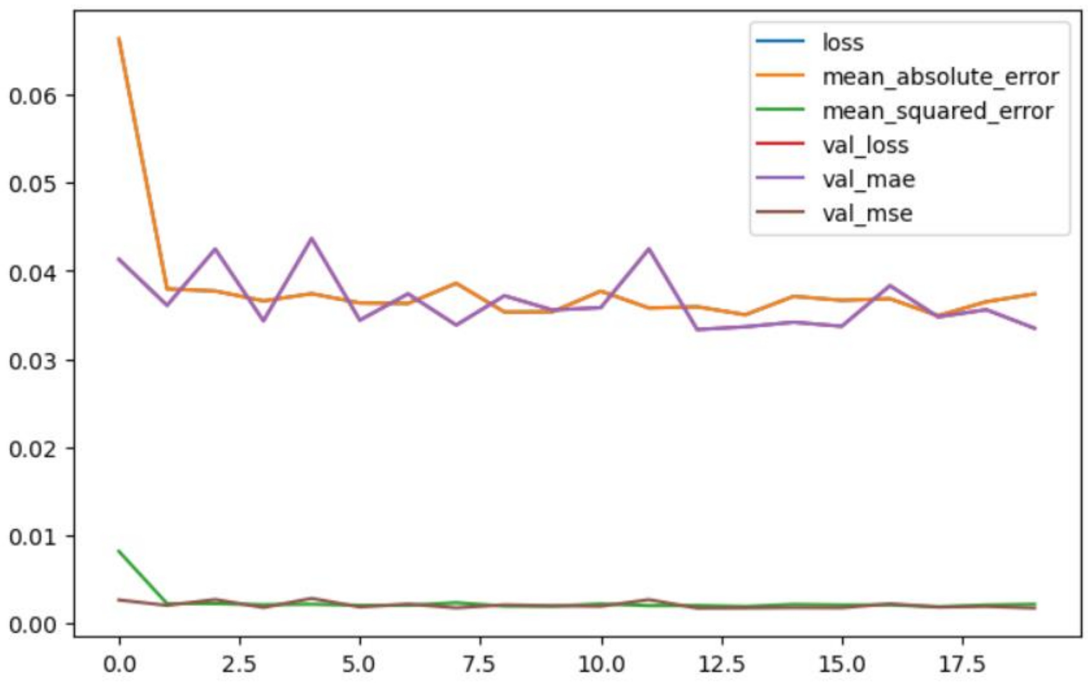
Plot the model performance:

```
plt.plot(best_mlp_model_2.history_['loss'])
plt.plot(best_mlp_model_2.history_['val_loss'])
plt.title('model performance')
plt.ylabel('loss values')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```



Did the model overfit?

```
pd.DataFrame(best_mlp_model_2.history_).plot(figsize = (8, 5))
plt.show()
```



Implementing a Multilayer Perceptron

Predictions

Obtain the predictions:

```
predictions = best_mlp_model_2.predict(X_test)
```

```
32/32 [==============================] - 0s 2ms/step
```

```
predictions = predictions.reshape(predictions.shape[0], 1)
```

Print the first five:

```
predictions[:5]
```

```
array([[0.5972543 ],
       [0.33833626],
       [0.26719126],
       [0.63041043],
       [0.29771176]], dtype=float32)
```

Predictions are scaled. We can use the *inverse_transform*() function to obtain the real values.

Unscale the predictions to see the real prices:

```
predictions_unscaled = scaler_y.inverse_transform(predictions)
```

Print the first five:

```
predictions_unscaled[:5]
```

```
array([[1481079.1 ],
       [ 845920.5 ],
       [ 671392.75],
       [1562415.4 ],
       [ 746263.44]], dtype=float32)
```

Unscale *y_test* to get the original values:

```
y_test_unscaled = scaler_y.inverse_transform(y_test)
```

Print the first five:

```
y_test_unscaled[:5]
```

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

Not that bad!!

Implementing a Multilayer Perceptron

Evaluate the model

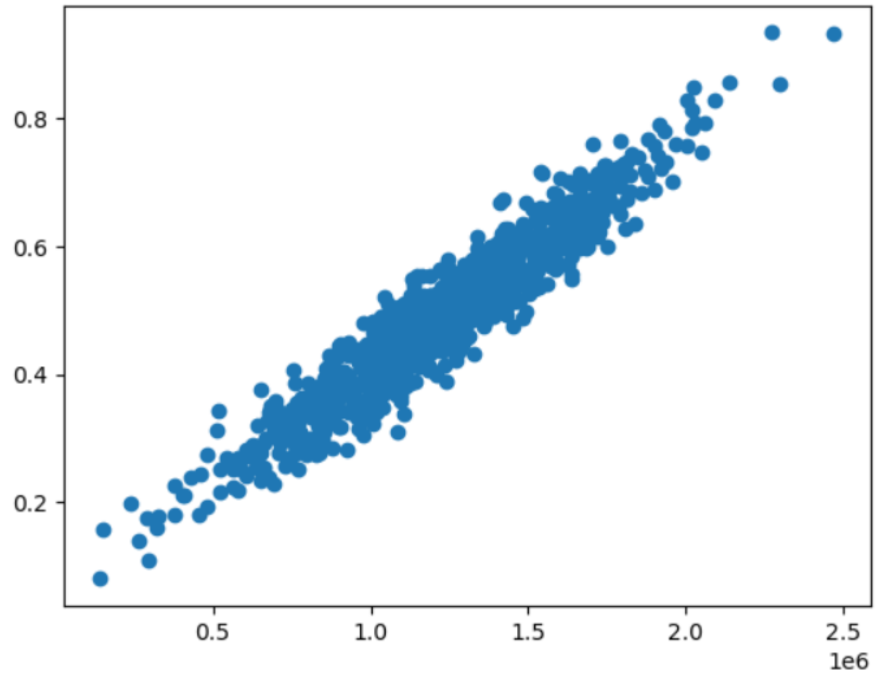
Assess by MAE, MSE and RMSE:

```
print('MAE:', metrics.mean_absolute_error(y_test_unscaled, predictions))
print('MSE:', metrics.mean_squared_error(y_test_unscaled, predictions))
print('RMSE:', np.sqrt(metrics.mean_squared_error(y_test_unscaled, predictions)))
```

```
MAE: 1235057.259318577
MSE: 1637743709840.513
RMSE: 1279743.610978587
```

Scatter the real values with the predictions:

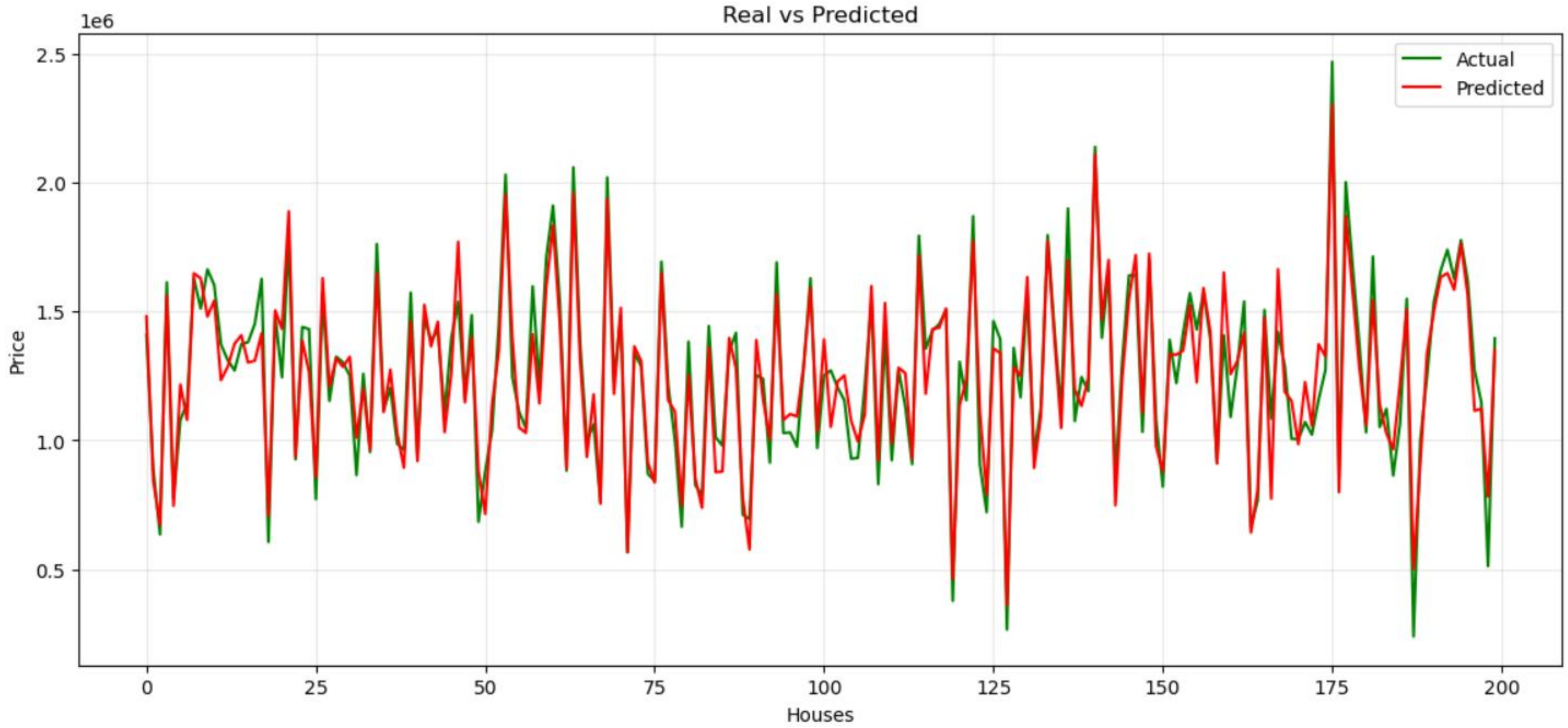
```
plt.scatter(y_test_unscaled, predictions)
```



Implementing a Multilayer Perceptron


```python
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()
```

```python
real_predicted_viz(200)
```

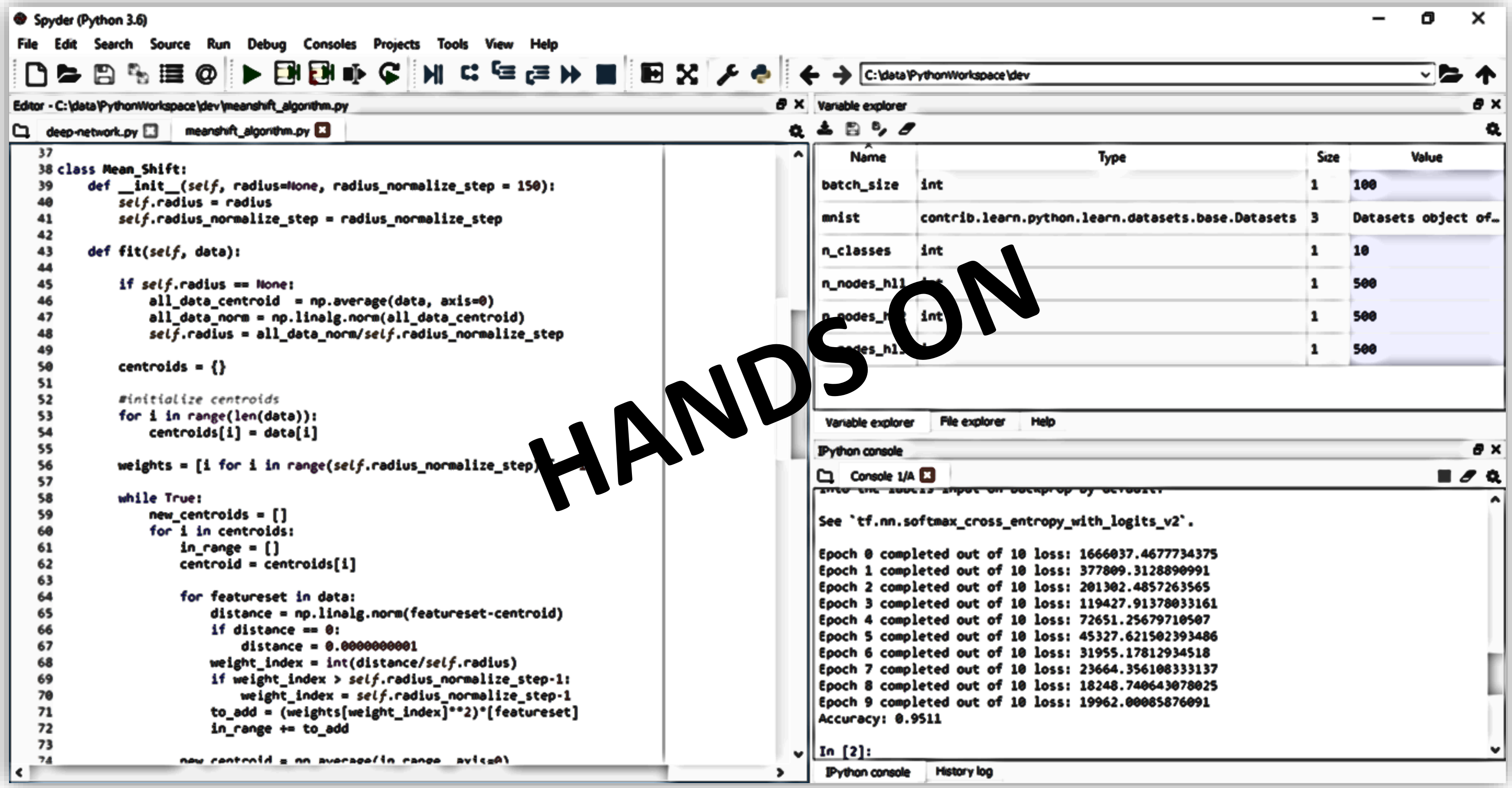


Implementing a Multilayer Perceptron

Compare the results with the ones obtained with the Linear Regression model created in class 4.

Which model performed better?

Hands On



Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
HANDS ON
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
def fit(self, data):
Variable explorer
Name Type Size Value
batch_size int 1 100
mnist contrib.learn.python.learn.datasets.base.Datasets 3 Datasets object of…
n_classes int 1 10
IPython console
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511