Designing artificial neural networks (ANN) for practical application

1. Objectives

This laboratory is focused on its first part on implementing an AND / XOR gates using artificial neural networks (ANN). The purpose of this section is to help the student gain an intuition of the basic concepts of ANN such as: input signals, synaptic weights, linear aggregator, bias, activation functions, output signals, single layer neuron, multiple-layer neurons, learning function and back-propagation. By using the programming language Python the students will implement from scratch an ANN that is able to predict the output of an AND/XOR gate. The ANN development will be re-implemented in Pyhton by using some dedicated machine learning platforms: Tensorsflow with Keras API.

In the second part, the laboratory is focused on developing and evaluating a neural network using Keras for a regression problem. The purpose of this section is to discover how to: load a CSV dataset and make it available to Keras, create and train a neural network model for a regression problem and evaluate the model.

2. Theoretical aspects

Machine learning is a branch in computer science that studies the design of algorithms that can learn specific patterns in ordered / unordered sets of data. The present laboratory is focused on deep learning methods, which represent a subfield of machine learning. The deep learning is one of the hottest fields in data science with many case studies that have astonishing results in robotics, image recognition and Artificial Intelligence (AI).

In deep learning, a computer model learns to perform classification tasks directly from images, text, or sound. Deep learning models can achieve state-of-the-art accuracy, sometimes exceeding human-level performance. Models are trained by using a large set of labeled data.

2.1. Artificial Neural Network

The Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way the biological nervous system such as brain process information. It is composed of large number of highly interconnected processing elements (neurons) working in unison to solve a specific problem.

The biological neurons (also called nerve cells – Fig.2.1) or simply neurons are the fundamental units of the brain and nervous system, the cells responsible for receiving sensory input from the external world via dendrites, process it and give the output through axons.

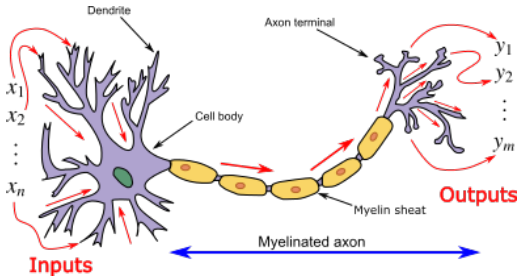

Fig. 2.1. The biological neuron

The following diagram represents the basic model of an ANN architecture (*i.e.*, perceptron) which is inspired by a biological neuron.

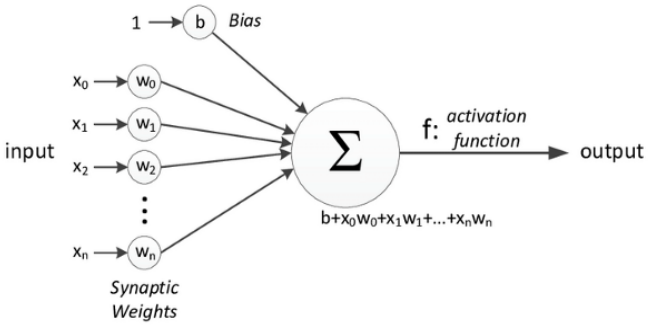

Fig. 2.2. Schematic representation of a perceptron

In Fig. 2.2, $x_0, x_1, \dots, x_n$ represent the various inputs (independent variables) of the network. Each of these inputs is multiplied by a connection weight or synapse. The weights are represented as w_0, $w_1, \dots, w_n$. The weights are used in order to show the strength of a particular node. While, b is a bias value. The bias allows shifting the activation function up or down. In the simplest case, these products are summed, fed to a transfer function (activation function) to generate a result and this result is sent as output. Activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The motivation is to introduce non-linearity into the output of a neuron. Without the activation function the output signal would be a linear function (one-degree polynomial). A linear function is easy to solve, but it is very limited in complexity. Without a nonlinear activation function, our model cannot learn and model real life data such as images, videos or audio. Let's consider a two-input perceptron with one neuron, as shown in Fig. 2.3.

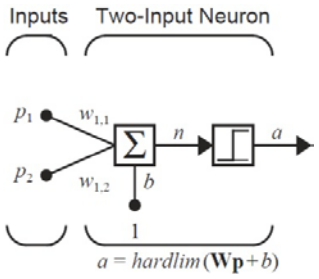

Fig. 2.3. Two-input/single output neuron

The output of this network is determined by:

$$a = hardlim(n) = hardlim(\,{}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b) = hardlim(w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + b)$$

where $hardlim(\cdot)$ is the step function that will assume unitary positive values when the neuron activation potential is greater than a threshold, otherwise, the result will be null. Thus, we have:

$$f(u) = \begin{cases} 1, & if\ u \geq 0 \\ 0, & if\ u < 0 \end{cases}$$

The *decision boundary* is determined by the input vectors for which the net input *n* is zero:

$$n = {}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b = w_{1,1} \cdot p_1 + w_{1,2} \cdot p_2 + b = 0$$

To make the example more concrete, let's assign the following values for the weights and bias:

$$w_{1,1} = 1;\ w_{1,2} = 1; b = -1$$

The decision boundary is then:

$$n = {}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b = 1 \cdot p_1 + 1 \cdot p_2 + 1 = 0$$

This defines a line in the input space. On one side of the line the network output will be 0, on the line and on the other side of the line the output will be 1. To draw the line, we can find the points where it intersects the p_1 and p_2 axes. To find p_2 set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{-1}{1} = 1$$

To find p_1 set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{-1}{1} = 1$$

The resulting decision boundary is illustrated in Fig. 2.4.

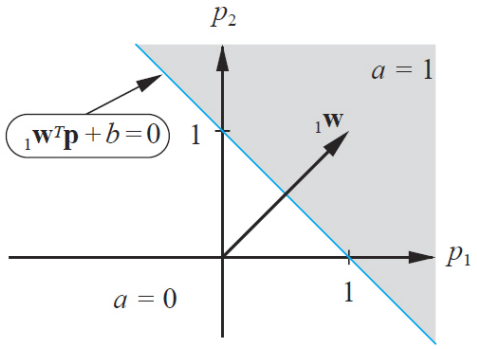

Fig. 2.4. Decision boundary for two-input perceptron

To find out which side of the boundary corresponds to an output of 1, we just need to test one point. For the input $\boldsymbol{p} = [2 \quad 0]^T$, the network output will be:

$$a = hardlim(\,{}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b) = hardlim\left([1 \quad 1] \cdot \begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1\right) = 1$$

Therefore, the network output will be 1 for the region above and to the right of the decision boundary.

Let's apply some of these concepts to the design of a perceptron network to implement a simple logic function: the AND gate. The input/target pairs for the AND gate are:

$$\left\{\boldsymbol{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}; \left\{\boldsymbol{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}; \left\{\boldsymbol{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}; \left\{\boldsymbol{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$

Fig. 2.5 illustrates the problem graphically. It displays the input space, with each input vector labeled according to its target (*t*). The dark circles indicate that the target is 1, and the light circles indicate that the target is 0. The first step of the design is to select a decision boundary. We want to have a line that separates the dark circles and the light circles. There are an infinite number of solutions to this problem. It seems reasonable to choose the line that falls "halfway" between the two categories of inputs.

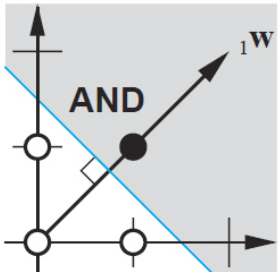

Fig. 2.5. Perceptron acting as an AND gate

Next we want to choose a weight vector that is orthogonal to the decision boundary. The weight vector can be any value, so there are infinite possibilities. One choice is: ${}_1\boldsymbol{w}^T = [2 \;\; 2]$. Finally, we need to find the bias, *b*. We can do this by picking a point on the decision boundary and satisfying: ${}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b = 0$. If we use $\boldsymbol{p}^T = [1.5 \;\; 0]$ we find:

$${}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b = [2 \;\; 2] \cdot \begin{bmatrix}1.5\\0\end{bmatrix} + b = 3 + b = 0 \rightarrow b = -3$$

We can now test the network on one of the input/target pairs. If we apply $\boldsymbol{p}_2$ to the network, the output will be:

$$a = hardlim\left({}_1\boldsymbol{w}^T \cdot \boldsymbol{p} + b\right) = hardlim\left([2 \;\; 2] \cdot \begin{bmatrix}0\\1\end{bmatrix} - 3\right) = hardlim(-1) = 0$$

which is equal to the target output t_2. It can be verified that all inputs are correctly classified.

The previous example illustrated the case of a neural network with a single layer perceptron. We will focus next on presenting the capabilities of a neural network with multilayer perceptrons for pattern classification. Consider the classic exclusive-or (XOR) problem. The input/target pairs for the XOR gate are:

$$\left\{\boldsymbol{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}, \left\{\boldsymbol{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\}, \left\{\boldsymbol{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\}, \left\{\boldsymbol{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$

Because the two categories are not linearly separable (Fig. 2.6), a single-layer perceptron cannot perform the classification.

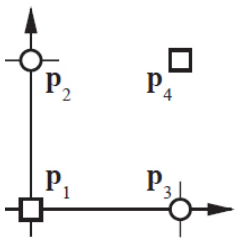

Fig. 2.6. XOR classification problem

A two-layer network can solve the XOR problem. In fact, there are many different multilayer solutions. One solution is to use two neurons in the first layer to create two decision boundaries. The first boundary separates $\boldsymbol{p}_1$ from the other patterns, and the second boundary separates $\boldsymbol{p}_4$. Then the second layer is used to combine the two boundaries together using an AND operation. The decision boundaries for each first-layer neuron are shown in Fig. 2.7.

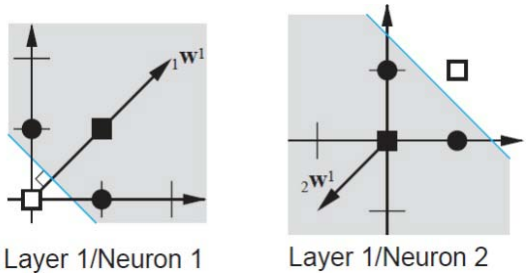

Fig. 2.7. Decision boundaries for XOR network

The resulting two-layers, 2-2-1 network is shown in Fig. 2.8. The overall decision regions for this network are shown in the Fig. 2.9. The shaded region indicates those inputs that will produce a network output of 1.

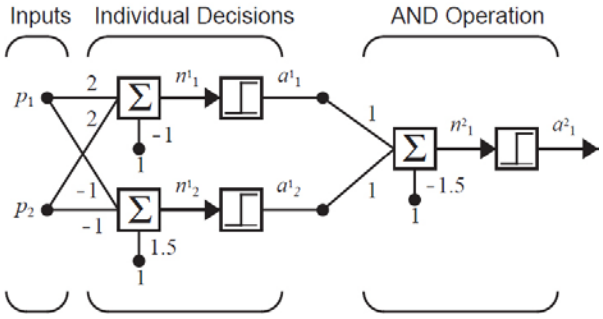

Fig. 2.8. Decision boundaries for XOR network

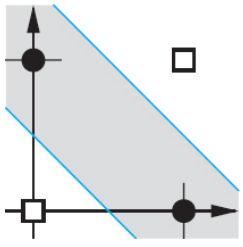

Fig. 2.9. The overall decision boundaries for XOR network

2.2. Artificial Neural Network for Regression

Regression models have been around for many years and have proven very useful in modeling real world problems and providing useful predictions, both in scientific and in industry and business environments.

Regression analysis can help model the relationship between a dependent variable (which should be predicted) and one or more independent variables (the input of the model). Regression analysis can show if there is a significant relationship between the independent variables and the dependent variable, and the strength of the impact - when the independent variables move, by how much you can expect the dependent variable to move. The simplest, linear regression equation looks like this:

$$y = \beta_1 + \beta_2 X_2 + \beta_3 X_3 + \cdots + \beta_k X_k + \varepsilon$$

where:

- y is the dependent variable - the value the regression model is aiming to predict;
- $X_1, X_2, \ldots, X_k$ are the independent variables - one or more values that the model takes as input, using them to predict the dependent variable;
- $\beta_1, \beta_2, \ldots, \beta_k$ are the coefficients - these are weights that define how important each of the variables is for predicting the dependent variable;
- ε is the error - the distance between the value predicted by the model and the actual dependent variable y. Statistical methods can be used to estimate and reduce the size of the error term, to improve the predictive power of the model.

There are various types of regression mechanisms, some of them are presented below:

- *Linear regression* (Fig. 2.10) - Suitable for dependent variables which are continuous and can be fitted with a linear function (straight line).

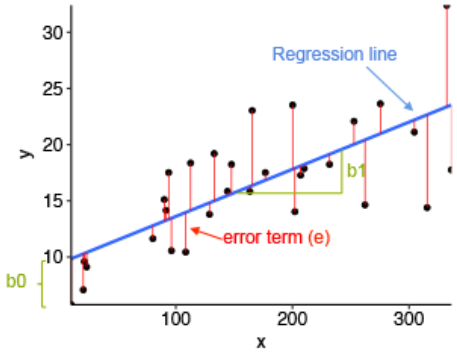

Fig. 2.10. Linear regression

- *Polynomial regression* (Fig. 2.11) - Suitable for dependent variables which are best fitted by a curve or a series of curves. Polynomial models are prone to over-fitting, so it is important to remove outliers which can distort the prediction curve.

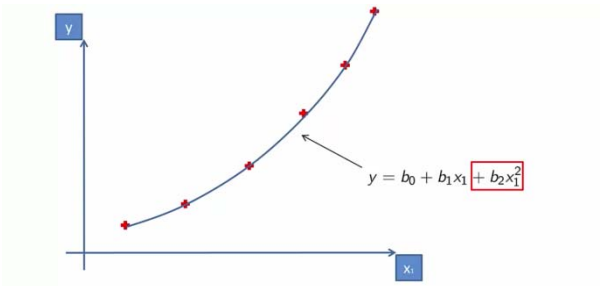

Fig. 2.11. Polynomial regression

- *Logistic regression* (Fig. 2.12) - Suitable for dependent variables which are binary. Binary variables are not normally distributed; they follow a binomial distribution, and cannot be fitted with a linear regression function.

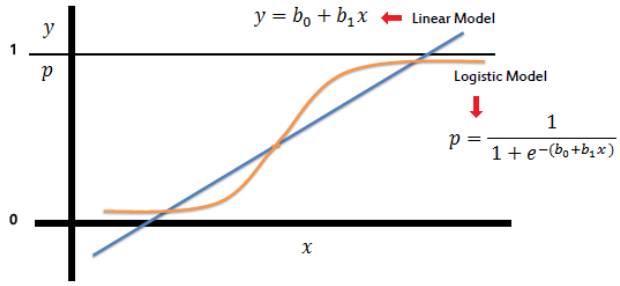

Fig. 2.12. Logistic regression

Neural networks are reducible to regression models - a neural network can "pretend" to be any type of regression model. For example, this very simple neural network, with only one input neuron, one hidden neuron, and one output neuron, is equivalent to a logistic regression. It takes several dependent variables (input parameters), multiplies them by their coefficients (weights), and runs them through a sigmoid activation function and a unit step function, which closely resembles the logistic regression function with its error term.

3. Practical implementation

Pre-requirements: Copy the ANDGate.py script into the project "ArtificialIntelligenceProject" main directory. The following example presents all the steps necessary to develop from scratch an artificial neural network (ANN) that simulates the AND gate behavior.

Application 1: Consider the input/target pairs for the AND gate:

$$\left\{\boldsymbol{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}; \left\{\boldsymbol{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\}; \left\{\boldsymbol{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\}; \left\{\boldsymbol{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$

Design an ANN system that is able to predict the output of the AND gate by analyzing the input vector. The network will contain one neuron with two inputs and will use the step unit activation function.

Step 1: *Load the input data/labels into Python as:*

```
#Input data
P = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
    ]

#Labels
t = [0, 0, 0, 1]
```

Step 2: *Initialize the weights vector and bias with zero.*

Step 3: *Set the number of training steps (epochs)* - The epoch is defined as one complete pass through the network of the entire training data. It is typical to train a deep neural network for multiple *epochs*.

Step 4: *Perform the ANN forward propagation* – For each input data, in each epoch, we forward the input through the network in order to obtain the predicted output.

```
for ep in range(epochs):
    for i in range(len(t)):
```

In order to perform the forward propagation through the artificial neural network define a function denoted `forwardPropagation(p, weights, bias)` that takes as input three parameters: one input data element, the weights and the bias. The function should return as output the predicted label for the current input (`return a`).

```
def forwardPropagation(p, weights, bias):
```

Within the `forwardPropagation` function the following sub-steps should be performed:

a. Multiply the weights with the input vector and add the bias (Fig. 2.3);
b. Define a function denoted `activationFunction(n)` that implements the *hardlim* function. The *hardlim(·)* will assume unitary positive values when the neuron activation potential is greater than a threshold, otherwise, the result will be null. Thus, we have:

$$hardlim(n) = \begin{cases} 1, & if\ n \geq 0 \\ 0, & if\ n < 0 \end{cases}$$

c. Pass the result obtained at step a. to the activation function `activationFunction(n)`.

Step 5: *Compute the prediction error* – The error is defined as the difference between the ground truth label, associated to the current input, and the predicted label (estimated by the ANN through the forward propagation – Step 4).

Step 6: *Update the weights* – The weights: $w_{1,1}$; $w_{1,2}$ (Fig. 2.3) values are updated in order to correct the prediction error. The new values for the weights are computed as:

$$_1\boldsymbol{w}^{new} = {}_1\boldsymbol{w}^{old} + e \cdot \boldsymbol{p}$$

```
weights[0] = weights[0] + error * P[i][0]
weights[1] = weights[1] + error * P[i][1]
```

Step 7: *Update the bias* – The bias: b (Fig. 2.3) value is updated in order to correct the prediction error. The new value for the bias is computed as:

$$b^{new} = b^{old} + e$$

Step 8: *Display the results* – Pass each input through the network and predict the output label. Print the input data, the predicted values and the ground truth labels.

Exercise 1: Determine the minimum number of epochs necessary to train the model in order to obtain only valid predictions.

Exercise 2: Modify the program in order to perform the prediction for an OR gate.

Exercise 3: Change the neuron activation function from step unit to sigmoid. Are there any differences when performing the prediction?

Application 2: Consider the input/target pairs for the XOR gate:

$$\left\{\boldsymbol{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}, \left\{\boldsymbol{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\}, \left\{\boldsymbol{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\}, \left\{\boldsymbol{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$

Design an ANN system that is able to predict the output of the XOR gate by analyzing the input vector. The network will contain two layers: the first layer will have two neurons, while the second layer is composed of one neuron. For all neurons the sigmoid activation function will be considered.

Observation: In this application all the vectors (input, weights, etc.) will be defined as numpy arrays. NumPy is a package for scientific computing which has support for powerful N-dimensional array objects. Before you can use NumPy, you need to install it and import it: `import numpy as np`. In the context of the current application, the library will be used in order to perform basic matrix operations as: matrices addition / multiplication or matrix transpose.

Copy the XORGate.py script into the project "ArtificialIntelligenceProject" main directory. The script implements from scratch an artificial neural network (ANN) used to predict the output of an XOR gate.

Exercise 4: Determine the minimum number of epochs necessary to train the model in order to obtain a prediction error inferior to 0.01. Consider the following error function:

$$C = \frac{1}{2N}\sum_{i=1}^{N}(out - pred)^2$$

where *N* is the total number of samples applied as input, out represents the desired output label, while $pred$ is the predicted output.

Exercise 5: Modify the activation function for the neurons from sigmoid to hyperbolic tangent. The hyperbolic tangent is given by the following equation:

$$tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}} = \frac{2}{1 + e^{-2u}} - 1$$

In the context of back-propagation the derivative of the $tanh(u)$ is:

$$\frac{dtanh(u)}{du} = (1 + u) * (1 - u)$$

What can be observed about the predicted values? Justify the results.

Exercise 6: How many trainable-parameters are involved in the ANN architecture existent for the XOR gate. Is the ANN performances influenced by the random initialization of the different variables?

Exercise 7: By using Python and some dedicated machine learning libraries (*i.e.,* Tensorsflow with Keras API) write a novel script that re-implements the ANN architecture of the AND gate developed at Application 1 (consider as activation function sigmoid). Extend the code in order to predict the output for the XOR gate in Application 2. Both networks will use Adam as optimizer.

Application 3: Copy the Regression.py script into the project "ArtificialIntelligenceProject" main directory. The following example presents all the steps necessary to develop from scratch an artificial neural network (ANN) that models the price of houses variation in thousands of dollars.

The problem that we will address is the house price market dataset. The dataset is saved within the "*housing.csv*" and you can put it in the current working directly. The dataset describes 13 numerical properties of houses and is concerned with modeling the price of houses in thousands of dollars. As such, this is a regression predictive modeling problem. Input attributes include things like: crime rate, proportion of nonretail business acres, chemical concentrations in the area and more. It is convenient to work with it because all of the input and output attributes are numerical and there are 506 instances.

The variable in the "*housing.csv*" document are in order:

- crimeRate - per capita crime rate by town;
- residentialZone - proportion of residential land zoned for lots over 25.000 m^2;
- industry - proportion of non-retail business acres per town;
- river - 1 if the house is on the river bounds; 0 otherwise;
- noxious - nitric oxides concentration (parts per 10 million);
- rooms - average number of rooms per house over ;
- age - the house building age;
- distance - weighted distances to five city employment centers;
- index - index of accessibility to radial highways;
- tax - full-value property-tax rate per $10.000;
- prRatio - pupil-teacher ratio by town
- black - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town;
- status – percent of the lower status of the population;
- ***medValue*** - median value of owner-occupied homes in $1000's.

Step 1: *Load the input data/labels into Python.*

```
#Input data
csvFile = pd.read_csv("./Houses.csv").values
```

Step 2: *Shuffle the data* – Shuffling data prevent the model to learn the order of the training variable and to have as input highly correlated examples consecutively.

```
np.random.shuffle(data)
```

Step 3: *Separate the input data from the labels* – the system has to predict the houses price values in thousands of dollars (medValue).

Step 4: *Split the data into training/testing datasets* – consider for training 80% of the dataset while for testing 20%.

Step 5: *Define the ANN network* – The ANN network will be defined within a function denoted `modelDefinition()`.

The function is used to create the baseline model to be evaluated. It is a simple model that has a single fully connected hidden layer with 8 neurons. The network uses good practices such as the rectifier activation function for the hidden layer. No activation function is used for the output layer because it is a regression problem and we are interested in predicting numerical values directly without transform.

The efficient ADAM optimization algorithm and a mean squared error loss function are used. This will be the same metric employed to evaluate the performance of the model. It is a desirable metric because by taking the square root, it gives us an error value that we can directly understand in the context of the considered problem (thousands of dollars).

Within the `modelDefinition` function the following sub-steps should be performed:

a. We start by instantiating a Sequential model:

```
# create model
model = Sequential()
```

Every Keras model is either built using the Sequential class, which represents a linear stack of layers, or the functional Model class, which is more customizable. We will use the simpler Sequential model, since our network is indeed a linear stack of layers.

b. Since we would like to build a standard feed-forward network, we only need the Dense type layer, which is a regular fully-connected network layer.

```
model.add(Dense(8, input_dim=13, kernel_initializer='normal', activation='relu'))
model.add(Dense(1,kernel_initializer='normal'))
```

c. Before we can begin training, we need to configure the training process. We set 2 key factors for the compilation: the optimizer (*i.e.*, the Adam gradient-based optimizer), the loss function (*i.e.*, mean square error).

```
model.compile(loss="mean_squared_error", optimizer="adam")
```

d. Return the model.

```
return model
```

Step 6: *Train the model* – Training a model in Keras literally consists only in calling the `fit()` function and specifying some training parameters as: training data and labels, the number of epochs to train (iterations over the entire dataset = 100) and the batch size to use when training (number of samples per gradient update = 16).

```
model.fit(x_train, y_train, epochs=100, batch_size=16, verbose=2)
```

Step 7: *Predict the house price for all the samples in the testing dataset* – Finally, the test dataset is used to evaluate the model.

```
predictions = model.predict(x_test)
```

Note: Your results may vary given the stochastic nature of the algorithm, evaluation procedure or differences in numerical precision. Consider running the example a few times and compute the average outcome.

Exercise 8: Determine the MSE (mean square error) for the testing dataset when using the regression system implemented in Application 3.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y is the ground truth house score, while $\hat{y}$is the predicted house score.

Exercise 9: Explain why at each run of the program the MSE has different values?

Exercise 10: Increase the depth of the ANN with one additional hidden layer with 16 neurons. Determine the MSE (mean square error) for the testing dataset for the new system architecture.

Exercise 11: Consider the ANN architecture defined at Exercise 10. Using the "*housing.csv*" dataset, re-train the system in order to define a regression model able to predict the nitric oxides concentration (noxious) for the test dataset. In this case the median value of owner-occupied homes (medValue) is considered known and is taken as input variable. Determine the MSE (mean square error) for the testing dataset.

OBSERVATION: The laboratory report (*.pdf file) will contain:

- The code for all the exercises. The code lines introduced in order to solve an exercise will be marked with a different color and will be explained with comments.
- Print screens with the results displayed in the PyCharm console.
- The exercises solutions: tables of results and responses to the questions.