Chapter 8 : Part 4

**Deep Learning**

**ANN: Artificial Neural Network**

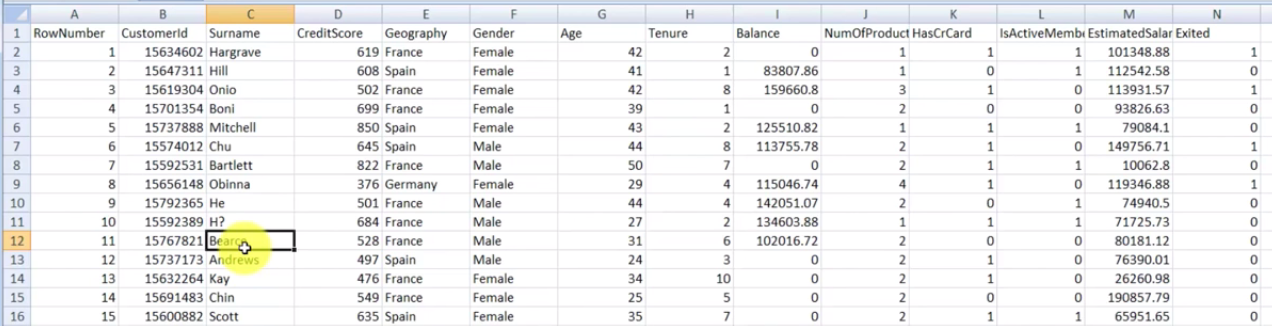
Python Implementation

* Here in the first branch of deepening (ANN), we're going to study this classification problem using ANN.
* In the next chapter we will study CNN (convolutional neural networks) another branch of deep learning which are which work very well for computer vision tasks unlike artificial neural networks (ANN).

**8.4.1 Business Problem Description**

*Deep* *learning* is one of the *most fascinating* and also the *most powerful* branch of *Machine* *Learning*. We will build a very powerful ***ML*** ***models*** based on Deep Neural Networks.

* Today deep learning is used for many of demanding and highly *compute intensive* tasks for example computer vision (like recognizing faces and pictures or videos) & medicine (recognizing tumors in some brain images).
* In fact deep learning can be used for making predictions (regression)or classificationsfor Business Problems.
* And it is also used for *recommended systems* with the use of *deep Boltzmann machines*.



* Data-set: We have a data-set, containing columns: *RowNumber*, *CustomerId*, *Surname*, *Geography*(country), and some other info.. Those data are the customers of a Bank. The dependent variable is, "***Exited***", it indicates if a customer ***leaves*** a bank or not. The data-set contains data of 10000 customers.
* The bank has been seeing unusually ***high*** ***churn*** ***rates***. Churn is when people leave the company. They want to understand what the problem is. We are here to look into this data-set for them and give them some insights.
* This fictional bank operates in Europe in three countries ***France***, ***Spain*** and ***Germany***. They have lots and lots of customers so they took this sample of 10000 customers and they measured six months ago everything they knew about them,
* Their ***CustomerId***, their ***Surname***, ***CreditScore***, ***Geography***, ***Gender*** , ***Age*** their ***Tenure*** (how long they've been with the bank),
* The ***Balance*** of the customers at that point in time, the ***NumberOfProducts*** they had at that point in time (i.e. savings account, credit card or loan), ***HasCrCard*** (did the customer have a credit card or not),
* ***IsActiveMember*** (active member or not active member can be measured differently by different organizations. It could be whether or not the customer *logged* into their *online* *banking* in the past month whether they did a *transaction* in the past two months or some other measurement like that.),
* ***EstimatedSalary*** (the bank doesn't know the salary of the customers but based on the other things they know they could estimate a salary for that customer).
* So six months ago they measured all of these things and they waited six months and observed which customer is living the bank and which stayed.
* In the data-set, the column ***Exited*** tells you whether or not the person left the bank within those six months. Here ***1 means*** that the person is ***left*** the bank and ***0 means*** the person ***stayed*** in the bank.
* Churn rate: Churn rate, in its broadest sense, is a measure of the number of individuals or items moving out of a collective group over a specific period.
* Goal: Our goal is to create a Geodemographic segmentation model to tell the bank which of their customers are at *highest risk of leaving*. We are going to solve this business problem using Artificial Neural Networks.
* Other applications: For a lot of customer centric organization this is going to be valuable, it doesn't have to be for churn rates.
* *Geodemographics segmentation models* can be applied to millions of scenarios, for instance even in a bank the same scenario could work should the person *get a loan or not*, should the person be approved for *credit-card or not*.
* A person is reliable or not: You'd have a binary outcome. So based on prior experience you would know whether or not *a person is reliable* and you'd build a model and say which people are more likely to be reliable and which people are more likely to default.
* And that could govern the Bank's decision on whether "to give" or "not to give" loans.
* You could apply a demographic segmentation and it doesn't even have to be demographic!! For example, when you have a binary outcome and you have lots of independent variables you can apply this Technique.

**8.4.2 Data Preprocessing**

We already know that the problem that we're about to deal with is a classification problem.

We have these *independent variables*: ***CustomerId***, ***Surname***, ***CreditScore***, ***Geography***, ***Gender*** , ***Age*** ***Tenure***. And based on these *independent variables* we are going to *predict* which customers are *leaving the bank*.

* Installing Libraries: We need to install 3 libraries: ***Theano***, ***TensorFlow*** and ***Keras***.
* Theano: Theano is a Python library that allows you to define, optimize, and efficiently evaluate mathematical expressions involving multi-dimensional arrays. Primarily developed by the ***Montreal Institute for Learning Algorithms (MILA) at the Université de Montréal.***.
* What is also great about this library is that it can run on CPU and also on GPU.
* TensorFlow: TensorFlow is another open source Numerical-computation library that runs very fast inn you CPU or GPU. This library was originally developed by the Google brain team at Google and it's now under the Apache 2.0 license.
* These two libraries Theano and TensorFlow are used mostly for research and development purposes. You have to use these two libraries to build a *Deep Neural Network* from scratch. That is with many lines of code. If you want to do some *Research* in order to *Improve* the *Deep Neural Networks* invent and Develop a new kind of Neural Network or any other kind of Deep Learning Models.
* Keras: Now we are not going to directly use Theano and TensorFlow, we're going to use another library called ***Keras*** that's in some way *wraps* the *two* *libraries* (Theano and TensorFlow).
* ***Keras*** helps to build *Deep Neural Networks* in a very few lines of code. We will use ***Keras***to build *Deep Learning Models* very efficiently as we used ***scikit-learn*** to build *ML models*.
* TensorFlow download & setup: Tensorflow runs at 64bit OS. Just need python 3.7 or python 3.8 version and then just use this command: " ***pip install tensorflow*** ". Use command prompt to install following:

|  |  |  |
| --- | --- | --- |
| * Anaconda Environment for TF * Create conda environment for TF. Choose the Python version. Execute on cmd. * After that install TF using pip. * Choose and install for GPU or CPU. * Finish the installation using command   Following might be useful  ***pip*** *install scikit-learn*  ***pip*** *install numpy*  ***pip*** *install matplotlib*  ***pip*** *install statsmodels* | Conda Environment: You might need to re-install ***spyder*** in the created ***environment***. Also install ***scikit-learn***, ***matplotlib*** etc.  ***conda*** *create --name tensorflow python=3.8*  ***conda*** *activate tensorflow*  Above will install tensorflow  conda install jupyter  ***conda*** *install scipy*  *##* ***pip*** *install tensorflow-gpu*  ***pip*** *install tensorflow*  ***conda*** *update -n base -c defaults conda*  Install Keras and Theano:  ***pip*** *install keras*  ***pip*** *install Theano* | |
| * *Another method:* Without creating virtual environment in Anaconda/Conda:  1. Open anaconda Powershell prompt 2. Run following codes to install TensorFlow in base(root):   **conda** activate base  **pip** install --user tensorflow | | ***--user*** makes pip install packages in your home directory instead, which doesn't require any special privileges.   * To remove conda virtual environment, use:   ***conda env remove --name env\_name*** |

* Data Preprocessing: We're going to build this model in two parts:

1. Data processing
2. Creating the ANN model.

* Since our business problem is classification problem. The *independent* *variables* are some information about customers in a bank. We are trying to predict a binary outcome for the *dependent* *variable* which is:

***1*** : if the customer ***leaves*** the bank

***0*** : if the customer ***stays*** in the bank.

* So we will use our classification template (or previously done classification model). Also we don’t need the visualization part, because current problem has 10 *independent* *variables*.
* We will use the ***confusion*** ***matrix*** to evaluate the performance of our ***ANN model***.
* Choosing the independent variables: We don’t need the first 3 – variables, ***RowNumber***, ***Customerld***, ***Surname***. Because these variables has no impact on the "Exited" (dependent variable). We take the rest of the variables excluding the last "Exited". "Exited" will be our dependent variable.

X = dataSet.iloc[:, 3:13].values

* In [:, 3:13], 3:13 means "all columns from index 3, excluding 13 indexed column". If the index of last column is unknown, then we can use "3:-1" which means "all columns from index 3, excluding the last column"
* How to find out the most impact independent variables: So with our own logic we are able to say which independent variables might have impact on the dependent variable. But we don't know which ***independent variable*** has the ***most impact*** on the ***dependent variable***. And that's what our ANN will spot and find the correlations by giving the bigger weights in the *neural network* to those independent variables that have the most impact.
* Encode Categorical Data: [1, 2] means that we want to convert 2nd and 3rd columns. We need to encode before splitting the data.

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [1, 2])], remainder = 'passthrough')

* However we are not using the above code to encode columns.
* We will use Label-encoder for "**Gender**", variable because it has only 2-categories.
* We will use One-hot-encoder for "**Geography**" (country variable), because it has more than two categories. So it will create 3 dummy variables.
* Encode "Gender" column: "**.values**" converts data-set into **array**. If we use **X = dataSet.iloc[:, 3:13]** instead of **X = dataSet.iloc[:, 3:13].values**

Then X won't be an Array it will be a Dataset. And we cannot use X[:, 2] for encoding. To encode a column in a Dataset, we need to use "key" of the column. We have to use ***X["Gender"]*** instead of ***X[:, 2]***.

***X["Gender"] = label\_encode.fit\_transform(X["Gender"])*** *# in case of data-set instead of Array*

**from** sklearn.preprocessing **import** LabelEncoder, OneHotEncoder

**from** sklearn.compose **import** ColumnTransformer

#*Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]*

label\_encode = **LabelEncoder**()

# *Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"*

# *X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied*

X[:, 2] = **label\_encode.fit\_transform**(X[:, 2])

**print**(X)

# *For a data-set we can still encode it using Columns "key"*

# *X["Gender"] = label\_encode.fit\_transform(X["Gender"])*

* Encode "Geography" column: Using ***OneHotEncoder*** will create 3-dummy variables.
* Avoiding dummy variable trap: To avoid the dummy variable trap, we exclude our 1st column (index 0). Using

**X = X[:, 1:]**

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [1])], remainder = 'passthrough')

X = **ct.fit\_transform**(X)

X = **np.array**(X) # *convert this output to NumPy array*

X = X[:, 1:] # *Excluding 0-index column to avoid dummy-variable trap*

* Data-Spit: We use ***test\_size = 0.2*** because of 10000 observations, so 8000 for training and 2000 for testing.

# *Data Split*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.2, random\_state = 0)

* Feature-Scaling: In ANN and in general deep learning we need feature scaling. It is thoroughly compulsory and that is because there is going to be a *lot of* *computation* and it is all *parallel computations*. So we need to ***apply feature-scaling to ease of these calculations***.
* And besides we don't want to have one independent variable dominating another one.

**from** sklearn.preprocessing **import** StandardScaler

# *y dependent variable, need not to be scaled: categorical variable, 0 and 1*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

**Data-Preprocessing All at once**

# *Artificial Neural Network*

# *Install : Tensorflow, Keras and Theano libraries.*

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *Data Extract*

dataSet = **pd.read\_csv**("Churn\_Modelling.csv")

# *X = dataSet.iloc[:, 3:-1].values # this can be used too*

X = dataSet.iloc[:, 3:13].values # *all columns from index 3, excluding 13 indexed column*

y = dataSet.iloc[:, 13].values # *the last column*

# *------------- Encode Categorical Data  -----------*

**from** sklearn.preprocessing **import** LabelEncoder, OneHotEncoder

**from** sklearn.compose **import** ColumnTransformer

#*Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]*

label\_encode = **LabelEncoder**()

# *Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"*

# *X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied*

X[:, 2] = **label\_encode.fit\_transform**(X[:, 2])

**print**(X)

# *For a data-set we can still encode it using Columns "key"*

# *X["Gender"] = label\_encode.fit\_transform(X["Gender"])*

#*Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]*

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [1])], remainder = 'passthrough')

# *remainder = 'passthrough' for remaining columns to be unchanged*

X = **ct.fit\_transform**(X)

X = **np.array**(X) # *convert this output to NumPy array*

**print**(X)

X = X[:, 1:] # *Excluding 0-index column to avoid dummy-variable trap*

# *------------------ Data Split -----------------------*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.2, random\_state = 0)

        # *Feature-Scaling after Data Split*

# *------------------ Feature-Scaling ----------------------*

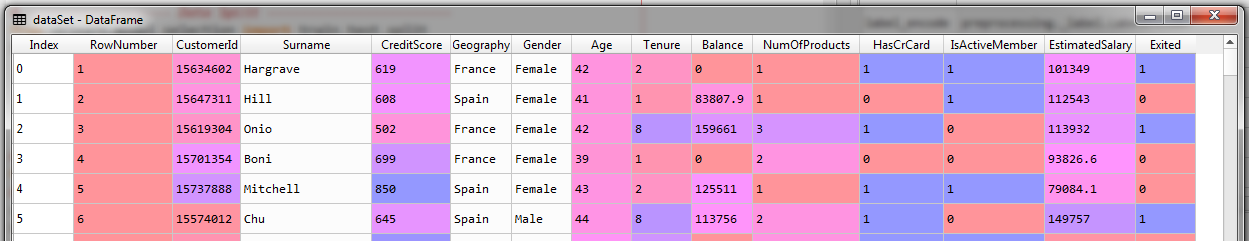
**from** sklearn.preprocessing **import** StandardScaler

# *y dependent variable, need not to be scaled: categorical variable, 0 and 1*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)



Dataset

|  |  |
| --- | --- |
| Feature matrix | Output-vector |
|  |  |

**8.4.3 Creating the ANN model**

Now everything is ready and we can eventually get into make the artificial neural network.

* Import Keras and packages: First step is to ***import*** the ***Keras*** libraries and the ***required*** ***packages (***some modules of the Keres library***)*** to build the Neural Network.

# *importing "keras" libraries and packages*

# *from tensorflow import keras*

**import** keras # *using TensorFlow backend*

**from** keras.models **import** Sequential

**from** keras.layers **import** Dense

* Actually ***keras*** is using TensorFlow as back-end. That is the ***keras*** library will build the *Deep Neural Network* based on TensorFlow. You can also use ***Theano*** as backend. But TF will be fast.
* Also we need to import two modules here.
* The ***Sequential*** module that is required to ***initialize*** our *neural network* and
* the ***Dense*** module is required *to build the* layers of our ANN.
* Initializing the ANN: Initializing the ANN means ***"defining it as a sequence of layers"***. There are actually two ways of initializing a *Deep Learning Model*. It's either by defining the ***sequence of layers*** or defining a ***graph***.
* Since we'll make ANN with successive layers (as you saw in pervious sections of this chapter), we'll initialize our Deep Learning Model by defining it as a Sequence Of Layers.
* We just need to create an Object of the ***Sequential*** class. This ***object*** that we're going to create is *nothing else than the* ***model itself***. i.e. the Neural Network that will have a ***role of a Classifier*** (this NN-model is going to be a classifier) here because our problem is a *Classification Problem* where we have to *predict* *a class*.

ann\_classifier = **Sequential**()

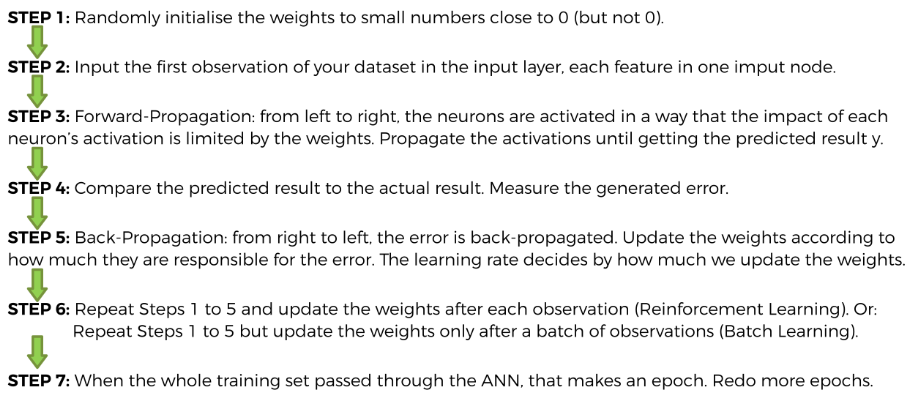
So this classifier object is nothing else than the future ANN that we're going to build.

* We *don't need to use any arguments* because we will define the layers step by step afterwards.
* We will start with the ***input layer*** and the first ***Hidden-layer*** and then we'll add some more ***hidden-layers*** then finally we'll add the ***output layer***.

So that's how we initialize our *Artificial Neural Network Classifier*.

* INPUT LAYER and first HIDDEN LAYER: We are going to add the first layer of our ANN-model which is the input-layer and the first hidden-layer. Recall the ANN has 7-steps to follow:

**SGD**



1. Step-1 ***"randomly initialize the weights of each of the nodes to small numbers close to 0"*** ***Dense*** module is going to take care of this first step.
2. Step-2: Our *first* *observation* goes into the *NN* and *each* *feature* as an *input* *node*. We already know the number of nodes of the input layer which is the number of independent variables we have in our Feature-Matrix.

* After data preprocessing, we had ***11*** independent variables. Hence in our input-layer will have ***11*** input nodes.

1. Step-3 is forward propagation. So from *left to right* the neurons are *activated* by the activation function in such a way that the "***higher*** the ***value*** of the ***activation function*** is for the ***neuron*** the ***more impact*** this ***neuron*** is going to have" in the network.

* Choosing an activation function: We'll choose the Rectifier-Activation-Function for the Hidden-Layers and the Sigmoid-Activation-Function for the Output-Layer.
* The best one based on experiments research is the rectifier-activation-function for Hidden-layers

|  |  |
| --- | --- |
|  |  |

* We also use sigmoid-rectifier-function for Output Layer. Since using the ***SIGMOID FUNCTION for the Output Layer*** will allow us to get the probabilities of the different.

i.e. for each observations of the test set we'll get the probability that the *customer* *leaves the bank* and the probability that the *customer stays in the bank*.

* Since we are trying to build a SEGMENTATION MODEL and by getting the probability we will be able to see which *customers* have the highest probabilities to *leave the bank*. So we'll be able to make a ranking of the customers by their probability to leave the bank.

And then you can *Segment* your *Customers* according to their *probability to leave the bank*. So that you can decide what to do in terms of *business constraints* and *business* *goals*

1. Step-4 the algorithm *compare* the predicted-result and actual-result and generates *Error*.
2. Step-5 the Error will be *back-propagated* and algorithm *updates* the *weights* of Synapses. *Weights* are *updated* according to how much they are responsible for this *generated error*.

* There are several ways of updating these weights. It is defined by the ***learning rate parameter*** which decides by *how much the weights are updated*.

1. In step-6 and step-7 above steps are repeated and minimize the cost-function.

* Adding input and first-hidden layer: We take the object ***ann\_classifier*** and use the ***add()*** method. ***add()*** method is used to *add the different* layers in our NN.
* Parameters of ***add()***: There is only one arguments and this argument is the ***layer*** that we want to add to our CNN. We are going use ***Dense*** function to define this ***layer*** argument.
* Parameters of ***Dense()***: We're going to add two layers the *input-layer* and the *first hidden-layer*. There are a *lot of arguments* for ***Dense*** function.
* These arguments are going to be all the parameters, such as: how the *weights are updated*, the type of *activation* function, number of *nodes* *for* *layers*, number of *input nodes* etc. Those things happens in this ***Dense()*** function.
* **output\_dim:** That is simply the number of nodes you want to add in this hidden layer.
* Here ***add()*** function doesn't know it is adding *input-layer* & the *first hidden-layer*. It just adding a *hidden layer* so for these *hidden layers* we have to specify the *number of inputs* in the *previous layers* (which is in the input-layer at very first).
* Choosing the number of nodes: It is the ***Art***. There is no rule of thumb on what would be the optimal number of nodes in these hidden layers. However, we can give some rules like for example :
* If your data is linearly separable, you don't even need a hidden layer and in fact you don't even need a neural network.
* Choose the number of nodes in the hidden-layer as the ***average*** of the number of nodes in the ***input*** ***layer*** and the number of nodes in the ***output*** ***layer***. It is not a rule but as a tip, if you ***don't want to be an Artist***. It is *not based on theory* but rather based on *experiments*.
* PARAMETER TUNING: If you want to be an artist (i.e. pro), then you have to experimenting with a technique called PARAMETER TUNING. Parameter Tuning is about using some techniques like ***K-fold cross-validation*** (we will study it In "Model selection and Ensemble model" later).
* ***K-fold cross-validation*** technique consists of creating a separate set in your data set besides the ***training-set*** and the ***test-set*** that is called a ***cross-validation-set***.
* Basically in this **cross-validation-set**, you *experiment different parameters* of your *model*. Such as: number of hidden layers and the number of nodes in the hidden layers. And then you test the *performance* of your *different models* with the *different parameters*.
* We won't do this  PARAMETER TUNING here, we will study ***K-fold cross-validation*** later. which will help us choose the *optimal parameters* of our model. But for now we're going to take the average of the *number of nodes in the input layer* and a *number of nodes in the output layer*.
* In our case, number of nodes in the input layer is ***11***, because the *number of nodes in the input layer* is the *number of independent variables*.
* And the *number of nodes in the output layer* is ***1*** because we have a binary outcome ***one*** or ***zero***. So the ***average*** is ***6***, i.e. *six nodes* in the *hidden layer*.

**output\_dim** = 6

Note: In the new documentation of ***Dense*** function ***output\_dim*** is replaced by ***units***, and the ***input\_dim*** is replaced by ***input\_shape***. However in the ***input\_shape*** argument you have to specify a ***tuple***.

* **init:** It corresponds to the step-1 of SGD algorithm: "randomly initialized the weights as small numbers close to zero". We can randomly initialize them with a uniform-function.
* For example: " ***glorot\_uniform*** " to initialize the weights. Or,
* More simple "***uniform*** " it's a simple-uniform-function to initialize the weights according to a uniform distribution, it will also make sure that the weights are small numbers close to zero.

**init** = "uniform"

Note: In the new documentation of ***Dense*** function ***init*** is replaced by ***kernel\_initializer***.

* **activation:** Here we specify the type of activation-funstion. We use the Rectifier-Activation-Function for the Hidden-Layers. The parameter is corresponds to Rectifier-Activation-Function the is "***relu***".

**activation** = "relu"

* **input\_dim:** It is a compulsory agument. It specify the *number of nodes in the* ***input-layer***. (i.e. *number of independent variables*.). Without ***input\_dim*** our ANN is simply initialized, *we haven't created any layer yet* and it doesn't know which nodes this hidden layer is expecting as inputs. Since we have 11 *independent variables*.

**input\_dim** = 11

Since this first layer will already be created we won't need to specify this ***input\_dim*** for the next hidden layers because the next hidden layers will know what to expect.

# *ann\_classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu", input\_dim = 11))*

**ann\_classifier.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu", input\_dim = 11))

* Adding second hidden-layer: The next step is to add a second hidden layer. To be honest, it is not necessarily useful for our data set but we're going to add it anyway for two reasons.
* First of all because the deep learning is defined as an artificial neural network with many hidden layers.
* And the second reason is simply that you need to add more hidden layers in your neural networks.
* We will use the same method ***add()*** and ***Dense()***, without ***input\_dim*** (because it is specified in previous layer). So we use following code:

**ann\_classifier.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu"))

* The number of nodes will be ***(11+1)/2 = 6*** i.e. (input layer nodes+ output layer nodes)/2.
* Also we initialize the weights using *uniform-initializer* to randomly initialize the *weights* to given *small* *numbers* close to *zero*.
* Since we are creating second hidden-layer, we use ***Rectifier-Activation-Function*** i.e. "***relu*** " again (we'll use *sigmoid function* for the *output* *layer*).
* Output-layer: Since we already have two hidden layers in ANN (one with input-layer & another is 2nd hidden layer). Now we add the output-layer. The code is similar to 2nd hidden layer but with sigmoid activation function*.* From the *logistic* *regression* of chapter 3, we know that, the *sigmoid* *function* is the heart of this *probabilistic* *approach*.
* Since we are making a ***Geo-demographic segmentation model*** we want to have *probabilities for the outcome* because we want to have the probability of each customer leaves the Bank.

**ann\_classifier.add**(**Dense**(units = 1, kernel\_initializer = "uniform", activation = "sigmoid"))

* We need to change the output parameter " ***units*** " because in our *output-layer* we want *only one node* because our dependent variable is a categorical variable with a binary outcome ***0 (stay)*** and ***1 (out)***.
* We will keep the ***uniform-initializer***.
* If you are dealing with a *dependent variable* that has *more than two categories,* for example: three categories then you will need to change two things here.
* First is the output parameter "units" that will be set as the number of classes( because it will be based on the *one-VSL method* while the dependent variable is *one-hot-encoded*.). So it will be " ***units = 3*** ".
* Second thing that you need to *change* is the *activation* *function* that in this situation would be ***Softmax-activation-function***, it is actually the *Sigmoid function* but applied to a *dependent* *variable* that has more than *two* *categories*.
* The ***Softmax*** function, also known as ***Softargmax*** or ***Normalized Exponential Function***, is a generalization of the logistic function to multiple dimensions. It is used in *multinomial logistic regression* and is often used as the last *activation function* of a *neural* *network* to *normalize* *the output* of a network to a probability distribution over predicted output classes, based on Luce's choice axiom.

**8.4.4 Compile and Train the ANN model**

* Compile the ANN model: We are done adding the layers of our artificial neural network. Now what we're going to do is to compile the ANN, applying SGD on the whole ANN.

**ann\_classifier.compile**(optimizer = 'adam', loss = 'binary\_crossentropy', metrics = ['accuracy'])

We apply ***compile()*** method on our ***ann\_classifier*** object. This ***compile()*** method contains several parameters.

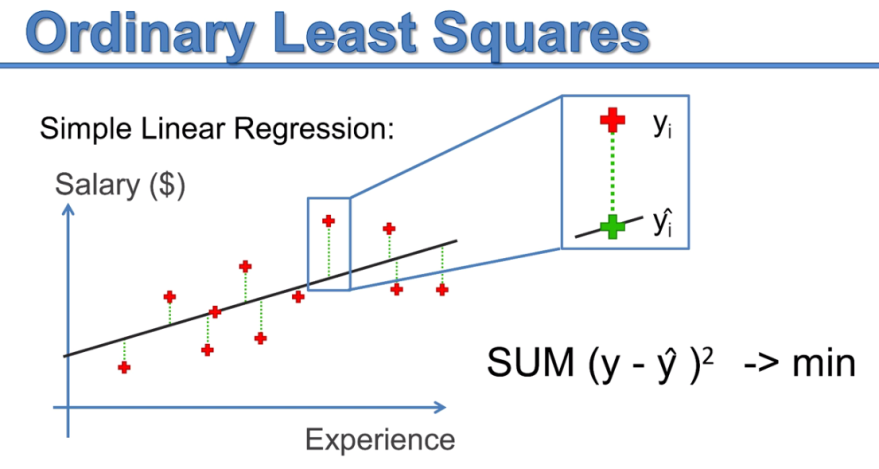
1. ***optimizer:*** "***optimizer***" is simply the Algorithm you want to use to find the OPTIMAL set of WEIGHTS in the neural networks. This algorithm is the SGD algorithm.

* There are several types of SGD algorithms. A very efficient one is called "adam" we're going to use it here.

***optimizer*** = "adam"

1. ***loss:*** Loss-function. And this corresponds to the Loss function within the ***SGD algorithm*** (i.e. within the "adam" algorithm).

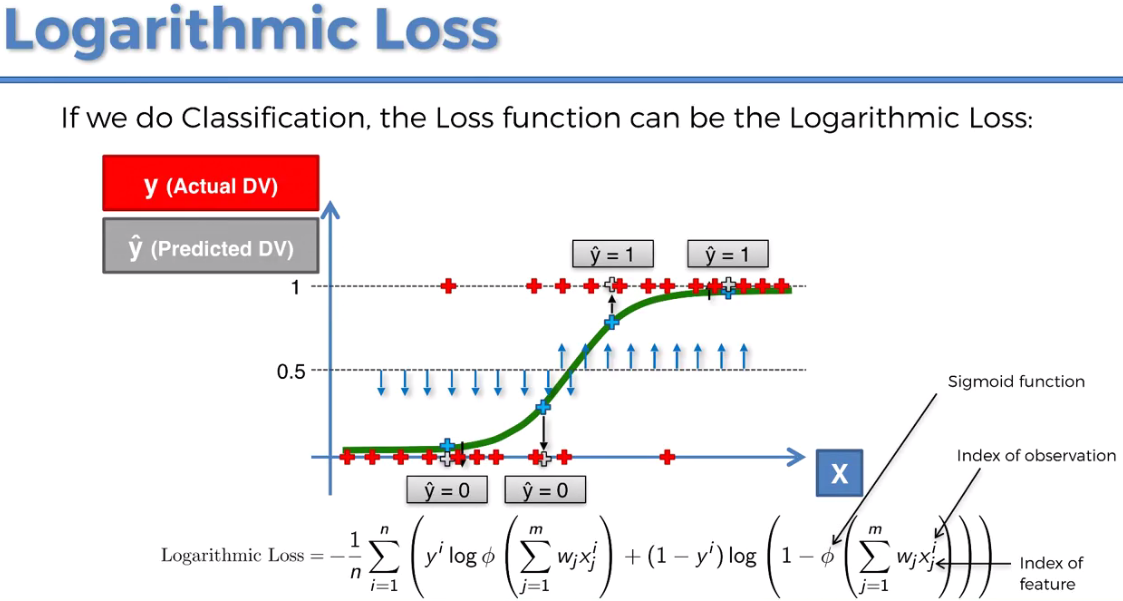
* Loss function: If you go deeper into the ***mathematical details*** of ***SGD***, you will see that it is based on a ***Loss-function***, that you need to optimize *to find the* optimalweights.
* For examples, we saw the loss-function when we studied Linear Regression. The Loss-Function was the sum of the squared errors (sum of the square differences between the real value and the predicted value). It is used to optimize the parameters of the regression model.



* Now the idea is exactly same in the case of SGD here. We have some parameters which are the weights in the neural network and so we need to specify a *Loss-function* that will *optimize* through *SGD* to find *optimal* *weights*.
* LOGARITHMIC-LOSS: Loss-function for the neural networks: This *Loss-function* is going to be kind of the same as for Logistic Regression because when you take a simple neural network (a PERCEPTION) and if you use a Sigmoid Activation Function for this PERCEPTION then you obtain a Logistic Regression Model

|  |  |
| --- | --- |
|  |  |

* If we go for the mathematical details of SGD for Logistic Regression, you will find out that the *Loss-Function* is not *the* sum of the squared errors like for Linear Regression. It's going to be a ***Logarithmic-Function*** that is called the ***Logarithmic-Loss***.



* Since the *activation function* for *output layer* is nothing else than the *sigmoid* *function*, the Loss-function that we're going to use compile our ANN and on which SGD-adam algorithm is based on: is the Logarithmic-Loss.
* If your dependent variable has a binary outcome (like here) then Logarithmic-Loss is called ***'binary\_crossentropy'***,
* If your dependent variable has more than two outcomes, like three categories then this logarithmic function is called ***'*** ***categorical\_crossentropy'***.

***loss*** = "binary\_crossentropy"

1. ***metrics:*** It is just a ***criterion*** that you choose to ***evaluate your model***. Typically we use the ***'accuracy'*** criterion. Basically what happens is that, when the weights are updated after each observation or after each batch of observations, the algorithm uses this ***'accuracy'*** criterion to *improve* the *models performance*.

* When we fit the ANN into our training-set, the accuracy is going to increase little by little until reaching a top accuracy and that will happen because we choose here the ***"accuracy" metric***.

***metrics*** = ['accuracy']

* Since this ***metrics*** argument is expecting a *list of metrics*. But here we only use onemetric which is the ***accuracy*** ***metric*** we need to add this *accuracy metric* as a list. This list will only contain *one* *element* which is the *accuracy* *metric*.
* Train the ANN model: We are going to use the ***fit()*** method into the ***training-set***.
* We will apply ***fit()*** to our object ***ann\_classifier*** .
* We pass the parameters ***X\_train*** and ***y\_train***.
* We will add two additional arguments for the ***"batch\_size"*** (separate dataset into several batches) and ***"number of epoch"***. This is where you're Deep Learning Artist Soul comes into play, because there is no rule of thumb. We need to experiment to find some *optimal choice* for this *batch size* here and this *number of epochs*.

1. ***batch\_size:*** We can choose to *update the weights* either after *each observation passing* through the *ANN* or after a *batch of observations*.

* ***batch\_size*** is the number of observations after which you want to update the weights. This could be any number depends on how many parts you want *to* *divide* the *train-data-set*.

1. ***epochs:*** An epoch is basically a round when the whole training set passed through the ANN. And in reality *training ANN* consists of *applying all steps of the ANN* over *many* *epochs*.

* Right now we're not going to experiment. We will go with some fixed choice of ***batch\_size*** and number of ***epochs***:

**batch\_size** = 10, **epochs** = 100

So that, we can see the *algorithm in action* and so that we can see the *accuracy improving over the rounds/epochs.*

* We can execute the model now. And eventually our model will be ready. And if we check the result we will converge to the accuracy of 86%.

|  |
| --- |
|  |

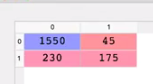
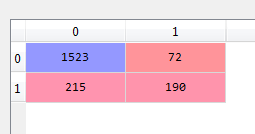
**8.4.5 Prediction and Evaluation of trained ANN model**

|  |  |
| --- | --- |
| We just trained ANN on the ***train-set*** and now time to make the ***predictions*** and the ***test-set***. We can execute following line as we did our classification models.  y\_prd = **ann\_classifier.predict**(X\_test)   * After execution we get the ***predicted*** ***probabilities***. These are the probabilities of ***leaving the bank*** of the 2000 customers of the test-set. * Now we can obtain the ***accuracy*** of the model using ***Confusion-matrix***. We got accuracy 86% on training-set, now we're going to use test-set. * Then if we get a good accuracy on the test-set , then the Bank is use this model on all the customers of the bank by ranking the *probabilities* from the *highest to the lowest* of the customers most likely to leave the Bank. * So then for example the bank to have a look at the 10 percent highest probabilities of their customers to leave the bank and so make it a SEGMENT and then analyzed in more depth using ***Data Mining Techniques*** to understand why the customers of this segment are the *most likely to leave the bank*. |  |

* Then the Bank itself can take some measures to prevent these customers from leaving.
* However, ***predict()*** method returns the ***probabilities*** in range ***[0, 1]***. But in confusion matrix we just need **0** or **1** i.e. "**false**" or "**true**". So we need to convert this predicted probabilities into "false" or "true".
* We have to set a threshold value to decide when the predicted result is ***1(true)*** and when the predicted result is ***0 (false)***. And the threshold value we set is 0.5 or 50%.
* The code is simple, we just need to apply a ***condition*** over the ***y\_pred*** vector.

y\_pred = (y\_pred > 0.5)

* ***y\_pred > 0.5*** is used because ***1*** means a customer will ***leave*** the ***bank***. So all values of ***y\_pred*** greater than ***0.5*** will become ***1***.
* Now we can *proceed* to the *confusion-matrix*:
* In ***medicine*** we can take a higher ***threshold*** if what we have to predict is sensitive information like for example if we have to ***predict*** if a ***tumor*** is ***malignant***.
* But here we're just predicting if a customer is ***leaving*** or ***staying*** in the ***Bank***. So 50 percent threshold is fine.

Instructor-version:  Practiced-version: 

* So out of 2000 new observations we get ***1550 + 175*** *correct* predictions and ***230 + 45*** *incorrect* predictions.
* On new observations i.e observations on which we didn't train ANN we got an accuracy of 86%. (We get this prediction without being an Pro ! We didn't do any parameter tuning. So maybe we can still get an even better accuracy.)

Practiced-Version

# *Artificial Neural Network*

# *Install : Tensorflow, Keras and Theano libraries.*

# *Library*

**import** pandas **as** pd

**import** matplotlib.pyplot **as** pLt

**import** numpy **as** np

# *-------------------------------- Part 1 : Data Preprocessing ------------------------------------------*

# *Data Extract*

dataSet = **pd.read\_csv**("Churn\_Modelling.csv")

# *X = dataSet.iloc[:, 3:-1].values # this can be used too*

X = dataSet.iloc[:, 3:13].values # *all columns from index 3, excluding 13 indexed column*

y = dataSet.iloc[:, 13].values # *the last column*

# *------------- Encode Categorical Data  -----------*

**from** sklearn.preprocessing **import** LabelEncoder, OneHotEncoder

**from** sklearn.compose **import** ColumnTransformer

#*Encode "Gender" using LabelEncoder."Gender" is in "3rd-column", hence X[:, 2]*

label\_encode = **LabelEncoder**()

# *Following is applicable to numpy array, if we used "X = dataSet.iloc[:, 3:13]"*

# *X = np.array(X) # it is needed if X is not an Arry. i.e. ".values" not applied*

X[:, 2] = **label\_encode.fit\_transform**(X[:, 2])

**print**(X)

# *For a data-set we can still encode it using Columns "key"*

# *X["Gender"] = label\_encode.fit\_transform(X["Gender"])*

#*Encode 'Gegraohy' using OneHotEncoder. "Gegraohy" is in "2nd-column", hence [1]*

ct = **ColumnTransformer**(transformers = [("encoding", **OneHotEncoder**(), [1])], remainder = 'passthrough')

# *remainder = 'passthrough' for remaining columns to be unchanged*

X = **ct.fit\_transform**(X)

X = **np.array**(X) # *convert this output to NumPy array*

**print**(X)

X = X[:, 1:] # *Excluding 0-index column to avoid dummy-variable trap*

# *------------------ Data Split -----------------------*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size= 0.2, random\_state = 0)

        # *Feature-Scaling after Data Split*

# *------------------ Feature-Scaling ----------------------*

**from** sklearn.preprocessing **import** StandardScaler

# *y dependent variable, need not to be scaled: categorical variable, 0 and 1*

st\_x= **StandardScaler**()

X\_train= **st\_x.fit\_transform**(X\_train)

X\_test= **st\_x.transform**(X\_test)

# *-------------------------------- Part 2 : Creating ANN model ------------------------------------------*

    # *1. importing "keras" libraries and packages*

# *from tensorflow import keras*

**import** keras # *using TensorFlow backend*

**from** keras.models **import** Sequential

**from** keras.layers **import** Dense

    # *2. initialize the ANN*

ann\_classifier = **Sequential**()

    # *3. Add the "input-layer" and  "first Hidden-layer"*

# *ann\_classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu", input\_dim = 11))*

**ann\_classifier.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu", input\_dim = 11))

    # *4. Add the "second Hidden-layer"*

**ann\_classifier.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu"))

    # *5. Add the "output-layer"*

**ann\_classifier.add**(**Dense**(units = 1, kernel\_initializer = "uniform", activation = "sigmoid"))

    # *6. Compile the ANN*

**ann\_classifier.compile**(optimizer = "adam", loss = "binary\_crossentropy", metrics= ["accuracy"])

    # *7. Train the model: fit the ANN to Training-set (batch\_size and epoch)*

**ann\_classifier.fit**(X\_train, y\_train, batch\_size= 10, epochs= 100)

# *--------------------- Part 3 : Predictions and Evaluating the model ---------------------------------*

# *Predict*

y\_prd = **ann\_classifier.predict**(X\_test)

# *coverting probabilities into "true/false" form. because 1 for leaving the Bank*

y\_prd = (y\_prd **>** 0.5)

# *Making the confusion matrix use the function "confusion\_matrix"*

# *Class in capital letters, functions are small letters*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_true= y\_test, y\_pred= y\_prd)

# *parameters of cm: y\_true: Real values, y\_pred: Predicted value*

accURacy = (cm[0][0] + cm[1][1])/X\_test.shape[0]

**print**(f"Accuracy = {accURacy}%")

# *python prctc\_ANN.py*

Instructor-Version (updated?)

Now in latest version of tensorflow the Dummy-variable trap can be fixed automatically. So in following version there are all 3-dummy variables.

# *Artificial Neural Network*

# *Importing the libraries*

**import** numpy **as** np

**import** pandas **as** pd

**import** tensorflow **as** tf

tf.\_\_version\_\_

# *Part 1 - Data Preprocessing*

# *Importing the dataset*

dataset = **pd.read\_csv**('Churn\_Modelling.csv')

X = dataset.iloc[:, 3:-1].values

y = dataset.iloc[:, -1].values

**print**(X)

**print**(y)

# *Encoding categorical data*

# *Label Encoding the "Gender" column*

**from** sklearn.preprocessing **import** LabelEncoder

le = **LabelEncoder**()

X[:, 2] = **le.fit\_transform**(X[:, 2])

**print**(X)

# *One Hot Encoding the "Geography" column*

**from** sklearn.compose **import** ColumnTransformer

**from** sklearn.preprocessing **import** OneHotEncoder

ct = **ColumnTransformer**(transformers=[('encoder', **OneHotEncoder**(), [1])], remainder='passthrough')

X = **np.array**(**ct.fit\_transform**(X))

**print**(X)

# *Feature Scaling*

**from** sklearn.preprocessing **import** StandardScaler

sc = **StandardScaler**()

X = **sc.fit\_transform**(X)

**print**(X)

# *Splitting the dataset into the Training set and Test set*

**from** sklearn.model\_selection **import** train\_test\_split

X\_train, X\_test, y\_train, y\_test = **train\_test\_split**(X, y, test\_size = 0.2, random\_state = 0)

# *Part 2 - Building the ANN*

# *Initializing the ANN*

ann = **tf.keras.models.Sequential**()

# *Adding the input layer and the first hidden layer*

**ann.add**(**tf.keras.layers.Dense**(units=6, activation='relu'))

# *Adding the second hidden layer*

**ann.add**(**tf.keras.layers.Dense**(units=6, activation='relu'))

# *Adding the output layer*

**ann.add**(**tf.keras.layers.Dense**(units=1, activation='sigmoid'))

# *Part 3 - Training the ANN*

# *Compiling the ANN*

**ann.compile**(optimizer = 'adam', loss = 'binary\_crossentropy', metrics = ['accuracy'])

# *Training the ANN on the Training set*

**ann.fit**(X\_train, y\_train, batch\_size = 32, epochs = 100)

# *Part 4 - Making the predictions and evaluating the model*

# *Predicting the Test set results*

y\_pred = **ann.predict**(X\_test)

y\_pred = (y\_pred **>** 0.5)

**print**(**np.concatenate**((**y\_pred.reshape**(len(y\_pred),1), **y\_test.reshape**(len(y\_test),1)),1))

# *Making the Confusion Matrix*

**from** sklearn.metrics **import** confusion\_matrix

cm = **confusion\_matrix**(y\_test, y\_pred)

**print**(cm)

Note

* Deep learning on GPU: GPU is a processor for graphic purposes. In terms of power and in terms of computations efficiency well the GPU is much more powerful because it has many more cores and it's able to run a lot more floating points calculations per second than the CPU.
* GPU is much more specialized for highly compute intensive task and parallel computations exactly as it is the case for neural networks.
* When we are for *propagating* the *activations* of the different *neurons* in the NN that exactly involves parallel computations and same when the error is back propagated and the NN. So lots of parallel computing, hence GPU is the better option.
* Tensorflow warnings: The ***W*** in the beginning stands for ***warnings***, ***errors*** have an ***E*** (or ***F*** for ***fatal*** ***errors***)
* In conda environment you need to install all the packages, Tensorflow, Theano, Keras, pandas, Scikit-learn, matplotlib etc.
* The updated instructor version uses the ***TF.keras*** and we used only ***keras***.
* New documentation:
* Add the first ANN layers (Input and Hidden Layers)

classifier.**add**(**Dense**(**units**=6, **activation**='relu', **kernel\_initializer**='uniform', **input\_dim** = 11))

* Adding the second hidden layer

classifier.**add**(**Dense**(**units** = 6, **kernel\_initializer** = 'uniform', **activation** = 'relu'))

* To get ***TF 1.x*** like behaviour in ***TF 2.0*** one can run

**import tensorflow.compat.v1 as tf**

**tf.disable\_v2\_behavior()**

but then one cannot benefit of many improvements made in TF 2.0. For more details please refer to the migration guide <https://www.tensorflow.org/guide/migrate>

* Test tensorflow for first time:

>>> import tensorflow as tf >>> hello = tf.constant("hello TensorFlow!") >>> sess=tf.Session()

To verify your installation just type:

>>> print(sess.run(hello))

If the installation is okay, you'll see the following output:

Hello TensorFlow!

