Chapter 11 : Part 3

**Deep Learning**

**ANN:** Evaluate performance using K-fold-CV

&

Hyper parameter tuning

K-fold-CV or K-fold-Cross-Validation, keras-wrapper, Dropout Regularization

**11.3.1 K-fold-CV in ANN**

|  |  |
| --- | --- |
| * K-fold-CV in ANN: In the previous section we discussed *K-fold-CV* and applied to *kernel-SVM* & *XGBoost*. In this section we'll apply K-fold-CV to ANN, it is our first Deep-learning Algorithm. * In Chapter 8: ANN, we trained our *artificial neural network* twice and we noticed that the *accuracies* are different. So we want to apply *K-fold-CV* to evaluate Model performance accurately. * Bias-Variance Tradeoff: We're trying to *train* a *model* that will not only be *accurate*, but also that should not have *too much* *variance* of *accuracy*, when we ***train*** it ***several times***. * Previously we trained our ANN twice and we obtained two different accuracies, ***85%*** and then ***84%***. Which one of these two accuracies we should take to *evaluate* our *models performance*? |  |

* We did *split* our data set between a *training-set* and a *test-set*. It results the *Variance Problem*. When we get the *accuracy* on the *test set*, if we *run* the model *again* and *test* *again* it's performance on *another test set*, well, we can get a very *different* *accuracy*.
* So, judging our model performance only on one accuracy on one test set, is not the most relevant way, to evaluate the model performance.
* That's why we use where train-set (or whole dataset) divided into equal k-groups and hence k-combination of (k-1)-train-group and 1-test-group, hence k-iterations. Finally we get a vector of k-accuracies for whicg we calculate the mean-accuracy and standared-deviation. It gives us the better evaluation of the model performance.
* Most of the time, K= 10, we train our model on 9-folds and we test it on the last remaining fold. With 10-folds there are 10 different combinations of 9-folds-train & 1-fold-test.
* That means we can *train* and *test* the *model* on ten *combinations* of training and test sets. Hence *10* different *accuracies*.
* That will give us a much better idea of the model performance, by taking average of the different accuracies of the ten evaluations and also compute the standard deviation.
* We then categorize the model-performance into 4-categiries:

1. good accuracy - small variance;
2. large accuracy - high variance;
3. small accuracy - low variance;
4. low accuracy - high variance

**11.3.2 Implement K-fold CV in ANN**

We implement K-fold CV after the data pre-processing phase.

* Keras wrapper: We implemented our ANN model with ***Keras-TensorFlow***. But the ***K-fold-CV*** belongs to ***Scikit-learn***.
* ***Keras wrapper*** combines ***Keras*** and ***Scikit-learn*** together. *Keras wrapper* module belongs to *Keras*. It will wrap *K-fold cross* *validation* by *Scikit-learn*, into the *Keras model*.
* We will be able to include K-fold cross validation in our Keras classifier.

**from** keras.wrappers.scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn.model\_selection **import** cross\_val\_score

* We use ***KerasClassifier*** to prepare to fit an ANN for each iteration of K-fold-CV. That ANN is invoked using ***cross\_val\_score***. That’s how both ***Keras*** & ***Scikit-Learn*** are combined together.
* ***build\_ANN\_clsfire():*** Since the ANN-model is not an *one-line code* like previous Regression/Classification models, we need a function to define our ANN-model. That's why we need to create ***build\_ANN\_clsfire()***, it just *define* our *ANN-model* and returns the ANN-classifier.
* Remember we don't need to *fit* the model *inside* this defined *function*.
* Also notice we Compile the ANN-classifier inside this ***build\_ANN\_clsfire()*** function, i.e. a *compiled ANN-classifier* is returned from this function.
* This function is simply a function that *returns* the *classifier* that we made here with all this *architecture for our ANN-model* (the initial-layer, different layers and *Compile*). Basically this function just builds the architecture of our ANN.

**def** **build\_ANN\_clsfire**():

*# initialize the ANN*

ANN\_clsfire = **Sequential**()

*# "input-layer" &  "first Hidden-layer"*

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

*#  "second Hidden-layer"*

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

*# "output-layer"*

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

*# Compile the ANN*

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

**return** ANN\_clsfire

* It is exactly as we first built our ANN. Basically we just copied all the code from that section including compiling (part 2) except fitting part. Later we'll fit each ANN-model (returned by this function) using ***cross\_val\_score***.
* We need to return the ANN-classifier before this function get out from the scope. Hence the return statement.
* We're also gonna define this function because the ***KerasClassifier()*** expects for one of its arguments a function, **build\_fn =** build\_ANN\_clsfire. Following is called after the defined function:

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire, batch\_size= 10, epochs= 100)

* ***KerasClassifier():***The ***KerasClassifier*** is the wrapper of *K-fold cross validation*.
* The variable ***ann\_clsfier\_for\_eval*** stores the classifier that is created by ***KerasClassifier*** (by invoking ***build\_ANN\_clsfire***, with specified ***batch\_size*** & ***epochs***).
* ***KerasClassifier*** actually prepare each ANN for ***sklearn*** class ***cross\_val\_score***, which expecting a ***sklearn*** based classifier:

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire, batch\_size= 10, epochs= 100)

* Here ***ann\_clsfier\_for\_eval*** is the global classifier variable (is the object of ***KerasClassifier*** class), because the classifier **ANN\_clsfire** inside **build\_ANN\_clsfire**() is a local variable because it is a variable inside the function.
* Classifier ***ann\_clsfier\_for\_eval*** will be built through *k-fold cross validation* on *10* different *training folds* and by each time measuring the model performance on *one test fold*.
* Parameters:
* **build\_fn:** It takes the defined function ***build\_ANN\_clsfire*** that builds the architecture of our ANN. So, we set **build\_fn=** build\_ANN\_clsfire
* ***batch\_size:*** Since we're not using ***fit()***, we need to specify the ***batch-size*** here. In the **cross\_val\_score** we just use ***X\_train***, ***y\_train***. We set batch\_size= 10
* ***epoch:*** Same goes for epoch we also need to specify it here. We set epochs= 100
* ***cross\_val\_score():*** As we did previously, we create a variable for accuracy-vector named **acuRacies** and we set our global classifier variable ***ann\_clsfier\_for\_eval*** as the ***estimator***.

acuRacies = **cross\_val\_score**(estimator= ann\_clsfier\_for\_eval, X = X\_train, y = y\_train, cv = 10, n\_jobs = **None**)

* All other things are same as we did before.
* ***n\_jobs*** *(-1 is important in DL):* Note that here ***n\_jobs = -1*** is crucial for ANN and all Deep-Learning techniques. Because all *Deep-Learning techniques* are ***parallel computation process*** that makes them slow. Moreover we're applying K-fold-CV, that makes the model K-time slower, because the model literally runs K-times and execution time is *K-time longer*.
* Hence we need to use CPU's full capacity for faster computing. For this reason we have to use ***n\_jobs = -1*** for all ***Deep-learning techniques***. (here we use n\_jobs = None in case you need to do other works in your PC) .

# *------------------ Part 4 : Evaluating, Improving  & Tuning the ANN ----------------------------------*

# *--------  Evaluating ANN : K-fold-CV -----------*

**from** keras.wrappers.scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn.model\_selection **import** cross\_val\_score

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**():

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

**ANN\_clsfire.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu", input\_dim = 11)) # *"input-layer" &  "first Hidden-layer"*

**ANN\_clsfire.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu")) # *"second Hidden-layer"*

**ANN\_clsfire.add**(**Dense**(units = 1, kernel\_initializer = "uniform", activation = "sigmoid")) # *"output-layer"*

**ANN\_clsfire.compile**(optimizer = "adam", loss = "binary\_crossentropy", metrics= ["accuracy"]) # *Compile the ANN*

**return** ANN\_clsfire

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire, batch\_size= 10, epochs= 100) # *creates/compile an ANN classifier*

acuRacies = **cross\_val\_score**(estimator= ann\_clsfier\_for\_eval, X = X\_train, y = y\_train, cv = 10, n\_jobs = **None**) # *compile ANN 10-times with 10-fold*

mean\_accu = **acuRacies.mean**()

st\_dvi\_accu = **acuRacies.std**()

|  |  |
| --- | --- |
| * To calculate mean and standard deviation we use following:   mean\_accu = **acuRacies.mean**()  st\_dvi\_accu = **acuRacies.std**()   * The mean accuracy is 83.3% and standard-deviation is 1%. So we are in following category: |  |
|  |

**All code at once (practiced)**

# *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}%")

# *------- prediction of new data-point using the built model ----------------*

"""

Use our ANN model to predict if the customer with the following informations will leave the bank:

    Geography:      France

    Credit Score:   600

    Gender:         Male

    Age:            40 years old

    Tenure:         3 years

    Balance:        $60000

    Number of Products:     2

    Does this customer have a credit card ?     Yes

    Is this customer an Active Member:  Yes

    Estimated Salary:   $50000

So should we say goodbye to that customer ?

"""

# *first create a 2D "NumPy array" in our X\_train's format.*

    # *it will be smilar to a single row of our X\_train*

    # *2 "[" used to define a single row of a 2-D array*

# *new\_dt\_pt = np.array([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])*

# *new\_dt\_pt= st\_x.transform(new\_dt\_pt) # scaling*

# *predict\_data\_pt = (ann\_classifier.predict(new\_dt\_pt) > 0.5) # Predict the data-point*

# *------------------ Part 4 : Evaluating, Improving  & Tuning the ANN ----------------------------------*

# *--------  Evaluating ANN : K-fold-CV -----------*

**from** keras.wrappers.scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn.model\_selection **import** cross\_val\_score

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**():

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

**ANN\_clsfire.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu", input\_dim = 11)) # *"input-layer" &  "first Hidden-layer"*

**ANN\_clsfire.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu")) # *"second Hidden-layer"*

**ANN\_clsfire.add**(**Dense**(units = 1, kernel\_initializer = "uniform", activation = "sigmoid")) # *"output-layer"*

**ANN\_clsfire.compile**(optimizer = "adam", loss = "binary\_crossentropy", metrics= ["accuracy"]) # *Compile the ANN*

**return** ANN\_clsfire

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire, batch\_size= 10, epochs= 100) # *creates/compile an ANN classifier*

acuRacies = **cross\_val\_score**(estimator= ann\_clsfier\_for\_eval, X = X\_train, y = y\_train, cv = 10, n\_jobs = **None**) # *compile ANN 10-times with 10-fold*

mean\_accu = **acuRacies.mean**()

st\_dvi\_accu = **acuRacies.std**()

# *-------- Improving ANN : Dropout-Regularization ------------*

# *--------  Tuning the ANN : Grid-Search ------------*

# *python prctc\_ANN\_eval\_KFldCV.py*

**11.3.3 Dropout-Regularization:** Improving the ANN

From above we see that our models accuracy is ***83%***, using ***K-fold-CV***, here we can improve that by modifying our ***ANN-layer*** using ***Dropout-regularization***.

* In deep learning *Dropout* *Regularization* is a very important *technique* that *prevents* *Overfitting*. (***Overfitting:*** Model was trained too much on the training-set, that it becomes much less performance on the test-set.)

|  |  |
| --- | --- |
| * In that case we have a large difference of *accuracies* between *training set* and the *test set*. * Generally, when *overfitting* happens, you have a much *higher* *accuracy* on the *training* *set* than the *test* *set*. * And another way to detect overfitting is when you observe a high variance when applying k-fold cross-validation * In this case, in your *vector* of *accuracies* of k-fold-CV, you'll get some *high accuracies* and some *low accuracies* and therefore you have *high* *variance* and that's how you detect *overfitting* in your model. |  |

|  |  |
| --- | --- |
| * Since we definitely didn't get overfitting in our ANN-model, because we get "Low-Bias & Low-variance accuracy", we actually don’t need to use ***Dropout*** (however we're doing it here for learning purpose). The accuracies were more or less around 83%. |  |

* Where do we need to apply dropout to our ANN:
* Dropout works as: At each iteration of the training, some ***neurons*** of your ***ANN*** are ***randomly disabled*** to prevent them from being *too dependent* on *each other* when they learn the correlations.
* Therefore, by over-writing these neurons, the *ANN* learns *several independent correlations* in the data, because each time there is not the same configuration of the neurons.
* We get these *independent correlations* of the data, because now the *neurons* work more *independently*, that prevents the *neurons* from learning *too much* and therefore that *prevents* *overfitting*.

So we need to apply ***Dropout*** to our Hidden-layers, after defining a layer. For example: following Dropout applied to first-hidden layer.

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

**ann\_classifier.add**(**Dropout**(p = 0.1))

* Implementing Dropout-Regularization:
* First we need to import a new class besides the ***Sequential*** class and the ***Dense*** class to modify our ANN This class is the ***Dropout*** class, from ***keras.layers***.

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

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

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

**from** keras.layers **import** Dropout

* Where exactly in the *neural network* are we going to apply dropout?
* Since ***Dropout*** is applied to the *neurons* so that some of them *randomly* become *disabled* at each *iteration*. So basically we need to *apply* ***Dropout*** to the *layers*. It can be to *one layer*, or it can be to *several layers*.
* Here we're going to apply to several layers (for demonstration purpose). We'll apply it to the *first* *hidden* *layer* and to the *second hidden layer*.

|  |
| --- |
| * One advice is that when you have *overfitting*, you should apply **Dropout** to *all the layers*, because that will give you *more* *chance* to *reduce* overfitting. |

* Applying dropout: As usual we use ***add()*** to apply ***Dropout*** in a layer. We do it after defining each layer.

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

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

**ann\_classifier.add**(**Dropout**(p = 0.1))

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

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

**ann\_classifier.add**(**Dropout**(p = 0.1))

* **p = 0.1:** **p** is *floating-point-number* in range ***(0, 1)***. It's the *fraction* of the *input* you want to drop. Basically that's the fraction of the neurons you want to drop, or *disable* at each *iteration*. In newer version ***p*** is replaced with ***rate***.
* For example: Suppose we have ***10*** ***neurons***, if we choose ***P*** equals ***0.1***, i.e. ***10%***, that means that at each iteration, ***one*** *neuron* will be disabled. If ***p*** equals ***0.2***, ***two*** *neurons* will be disabled.
* Which value of ***P(rate)*** should we input? The key "***p***" is replaced with "***rate***."
* Our advice is that when you have *overfitting*, you start trying with ***p*** or ***rate*** equals ***0.1***, ***10%***, and then if it *doesn't solve* the problem, if you still have *overfitting*, then you try a *higher* value of ***rate***. And you *increment* it for example by ***0.1***.
* So you first try with ***0.1*** and then if you still have *overfitting*, you try with ***0.2***. If you still have *overfitting* you try with ***0.3***. And until you manage to *reduce overfitting*.
* Too higher value of ***rate*** (***p***) is bad, when you disable *most* of the *neurons* of a *layer*, you'll get is *not overfitting* but *underfitting*. For example if *p=0.99* nothing will be *learnt*, *most* *neuron* will be *turned* *off*.
* So in general don't try to go over ***0.5*** because then you'll get ***too*** ***close*** to ***underfitting***.
* And so what we recommend is to try with ***0.1***, then try ***0.2***, ***0.3***, ***0.4*** and that should do it.
* After applying ***Dropout*** to *first-hidden layer*, we just need to copy the line-of-code and apply it to the *second-hidden-layer*.

**All ANN-structure at once with Dropout applied**

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

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

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

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

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

**from** keras.layers **import** Dropout

    # *2. initialize the ANN*

ann\_classifier = **Sequential**()

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

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

**ann\_classifier.add**(**Dropout**(rate = 0.1)) *# used 'rate' instead of 'p'*

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

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

**ann\_classifier.add**(**Dropout**(rate = 0.1)) *# used 'rate' instead of 'p'*

    # *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)

**11.3.4 ANN Hyper-Parameter tuning**

* Model-parameter: Are the parameters that the model learn by itself.
* Hyper parameters: Are the parameters that we fix to build a model. Those are the number of ***epoch***, the ***batch*** ***size***, the ***optimizer***, or the ***number of neurons*** in the layers.
* Parameter-Tuning & Grid-Search: It's a real deal about ***improving*** our model's ***performance*** because we're going to tune our model and we're gonna find the best ***hyper*** ***parameters***, like the best of number of ***epoch***, the best ***batch*** ***size***, the best ***optimizer***, so that we get the best ANN that will allow us to maximize our accuracy.
* So *Parameter tuning* is all about *finding* the *best values* of these *hyper parameters* and we are gonna do this using Grid Search technique.
* Grid Search will test several *combinations* of these *values* and will eventually return the *best selection* that leads to the *best accuracy* with ***K-fold-CV***.
* Implement parameter tuning in ANN: It's actually quite the same as implementing ***K-fold-CV*** because we will use the ***KerasClassifier*** class to wrap our ANN in a classifier, so that it can be used for ***scikit\_learn***.
* Because we'll use another class ***GridSearchCV*** from the same ***sklearn.model\_selection*** (used for k-fold-CV, ***cross\_val\_score*** function).
* For old version, try to import it from ***scikitlearn.grid\_search***

**from** sklearn.model\_selection **import** GridSearchCV

* Basically we will create an object from ***GridSearchCV*** we name it ***gridsearch\_ann*** that will apply parameter tuning on our ***KerasClassifier*** that is our *traditional NN*.
* So we can copy all our code from *K-fold-CV section* except the accuracy-vector ***acuRacies***, and edit it for Grid-Search. (changed lines are highlighted).

**from** keras.wrappers.scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn.model\_selection **import** GridSearchCV

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**():

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

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

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

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

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

**return** ANN\_clsfire

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire) # *creates/compile an ANN classifier*

* But in our ***KerasClassifier*** object we will not input the ***batch\_size*** and ***epoch*** arguments because that's the arguments we're gonna tune in ***Grid-Search***.
* Creating parameter dictionary: To use Grid-search for Hyper-parameter tuning, we need to ***input*** those ***Hyper-parameters*** as a ***list*** of ***dictionary***. So first we create this dictionary of parameters. We name this dictionary of parameters as ***params\_dic***.
* It is the same procedure described in the section 11.1.3 Grid-Search earlier in this chapter.
* In this dictionary we'll define different *combinations* of *hyper-parameters* and the *Grid-Search* uses the *k-fold-CV* with those different combinations and at the end it will return the *best accuracy* with the *best selection* of these *parameters*.
* Selecting parameters:

1. ***batch size:*** we can try several batch sizes because one of them will lead us to a better accuracy. We're going to input different values of the ***batch\_size*** in the dictionary.
2. ***epoch:*** We can also tune the number of epoch, there is an optimal number of epoch.
3. ***Optimizer:*** We can also tune some hyper parameters in our ANN architecture, like the optimizer.

params\_dic = [

    {

        'batch\_size': [25, 32],

        'epochs' : [2, 3],

        'opTmzr' : ["rmsprop"]

    },

    {

        'batch\_size': [24],

        'epochs' : [5],

        'opTmzr': ["adam", "rmsprop"]

    }

]

* Tuning optimizer: We want to tune the optimizer. How do we input some different values, considering that we already have a values of the optimizer in ***compile()***?
* You know we didn't have values for the number of epoch and the batch size, so we can try several of them here, but here we already have a value for the ***optimizer***. So how can we test some other ones?
* We have to use a parameter in the ***build\_ANN\_clsfire***, function, and this new argument is of course going to be, an argument that will give us choice for the optimizer.

**def** **build\_ANN\_clsfire**(opTmzr):

* We replace the Adam optimizer **'adam'** by this optimizer argument **opTmzr** that now plays the role of a variable.

ANN\_clsfire**.compile**(optimizer = opTmzr, loss = "binary\_crossentropy", metrics= ["accuracy"])

* Note that: the parameter **opTmzr** in the defined function "**def** build\_ANN\_clsfire(**opTmzr**):" must be the same as the key in the parameters-dictionary:

    {

        'batch\_size': [25, 32],

        'epochs' : [2, 3],

**'opTmzr'** : ["rmsprop"]

    },

* i.e. ***parameter*** in the function & ***key*** in the dictionary must be the same.
* And so now, we have the *right* to *input* *different* *values* that we want to test for our optimizer this key **opTmzr**, will be associated to key of the dictionary and so when we give different values for this key, well the ***different*** ***values*** are going to be tested in this optimizer in ***compile()***.

|  |
| --- |
| * So that's the trick, and therefore if you want to tune a hyper parameters that are in this architecture here, you have to create a new parameter in the ***buid\_function***,   **build\_ANN\_clsfire**(param1, param2, . .) |

**def** **build\_ANN\_clsfire**(**opTmzr**):

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

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

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

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

    ANN\_clsfire**.compile**(optimizer = **opTmzr**, loss = "binary\_crossentropy", metrics= ["accuracy"])

**return** ANN\_clsfire

ann\_clsfier\_for\_tune = **KerasClassifier**(build\_fn= build\_ANN\_clsfire) # *creates/compile an ANN classifier*

params\_dic = [

    {

        'batch\_size': [25, 32],

        'epochs' : [2, 3],

**'opTmzr'** : ["rmsprop"]

    },

    {

        'batch\_size': [24],

        'epochs' : [5],

**'opTmzr'**: ["adam", "rmsprop"]

    }

]

* For optimizers, we're going to try "**adam**", and ' **rmsprop** '.
* Sometimes ***rmsprop*** optimizer is better one for *deep learning models*. It is another excellent optimizer based on *stochastic gradient descent*.
* ***Keras*** documentation recommend to use this ***rmsprop*** for RNN, this is generally a ***better choice*** and indeed this is the ***optimizer*** that we're going to use for RNN.
* But lets still try it for our ANN, maybe this will lead us to better results.
* Implement Grid Search: To implement grid search, we use the same code that we did earlier in this chapter.

grid\_sch = **GridSearchCV**(

    estimator= ann\_clsfier\_for\_tune,

    param\_grid= params\_dic,

    scoring= 'accuracy',

    cv = 5

)

* **estimator**: is the classifier that we created for tuning, ***ann\_clsfier\_for\_tune***
* **param\_grid**: to use different parameter combination we use the list of parameter-dictionaries ***params\_dic***
* **scoring**: is the scoring matrix , we use ***'accuracy'*** as scoring matrix.
* **cv** = is the no. of ***fold*** for ***k-fold-CV***. We use ***5 folds*** here. When we tune our model with *gridsearch*, *k-fold cross validation* is going to be used to ***evaluate*** the ***accuracy***.
* To fit grid search object: We need to fit grid search to the training set, it is same as before.

grid\_sch = grid\_sch**.fit**(X\_train, y\_train)

* To view the best parameter-combination use the following lines of code:

best\_parametrs = grid\_sch**.**best\_params\_

best\_accuracy = grid\_sch**.**best\_score\_

**print**(f"\n\tBest parameters = {best\_parametrs} \n\tBest Accuracy = {best\_accuracy}")

**Hyper-Parameter Tuning Part**

# *--------  Tuning the ANN : Grid-Search ------------*

**from** keras**.**wrappers**.**scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn**.**model\_selection **import** GridSearchCV

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**(opTmzr):

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

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

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

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

    ANN\_clsfire**.compile**(optimizer = opTmzr, loss = "binary\_crossentropy", metrics= ["accuracy"])

**return** ANN\_clsfire

ann\_clsfier\_for\_tune = **KerasClassifier**(build\_fn= build\_ANN\_clsfire) # *creates/compile an ANN classifier*

params\_dic = [

    {

        'batch\_size': [25, 32],

        'epochs' : [2, 3],

        'opTmzr' : ["rmsprop"]

    },

    {

        'batch\_size': [24],

        'epochs' : [5],

        'opTmzr': ["adam", "rmsprop"]

    }

]

grid\_sch = **GridSearchCV**(

    estimator= ann\_clsfier\_for\_tune,

    param\_grid= params\_dic,

    scoring= 'accuracy',

    cv = 5

)

grid\_sch = grid\_sch**.fit**(X\_train, y\_train)

best\_parametrs = grid\_sch**.**best\_params\_

best\_accuracy = grid\_sch**.**best\_score\_

**print**(f"\n\tBest parameters = {best\_parametrs} \n\tBest Accuracy = {best\_accuracy}")

# *python prctc\_ANN\_imprv\_DrpRg\_grd\_sch.py*

* Execute-by-part (do not execute all-code): Since we're not gonna execute previous ANN-part-2 with Dropout-regularization and Evaluations-part (k-fold-CV):

1. First we need to import the data set and run the data reprocessing phase,
2. Finally execute the last section, parameter tuning, we have the whole tuning ready.

**All code at once (practiced, Used small values in parameter tuning)**

# *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 with  "Dropout-Regularization" ---------------------*

    # *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

**from** keras**.**layers **import** Dropout

    # *2. initialize the ANN*

ann\_classifier = **Sequential**()

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

# *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))

# *ann\_classifier.add(Dropout(p = 0.1))*

ann\_classifier**.add**(**Dropout**(rate = 0.1)) # *used 'rate' instead of 'p'*

    # *4. Add the "second Hidden-layer" and applying "Dropout-Regularization"*

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

# *ann\_classifier.add(Dropout(p = 0.1))*

ann\_classifier**.add**(**Dropout**(rate = 0.1)) # *used 'rate' instead of 'p'*

    # *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}%")

# *------- prediction of new data-point using the built model ----------------*

"""

Use our ANN model to predict if the customer with the following informations will leave the bank:

    Geography:      France

    Credit Score:   600

    Gender:         Male

    Age:            40 years old

    Tenure:         3 years

    Balance:        $60000

    Number of Products:     2

    Does this customer have a credit card ?     Yes

    Is this customer an Active Member:  Yes

    Estimated Salary:   $50000

So should we say goodbye to that customer ?

"""

# *first create a 2D "NumPy array" in our X\_train's format.*

    # *it will be smilar to a single row of our X\_train*

    # *2 "[" used to define a single row of a 2-D array*

new\_dt\_pt = np**.array**([[0.0, 0.0, 600, 1, 40, 3, 60000.0, 2, 1, 1, 50000.0]])

new\_dt\_pt= st\_x**.transform**(new\_dt\_pt) # *scaling*

predict\_data\_pt = (ann\_classifier**.predict**(new\_dt\_pt) **>** 0.5) # *Predict the data-point*

# *------------------ Part 4 : Evaluating, Improving  & Tuning the ANN ----------------------------------*

# *--------  Evaluating ANN : K-fold-CV -----------*

**from** keras**.**wrappers**.**scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn**.**model\_selection **import** cross\_val\_score

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**():

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

    ANN\_clsfire**.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu", input\_dim = 11)) # *"input-layer" &  "first Hidden-layer"*

    ANN\_clsfire**.add**(**Dense**(units = 6, kernel\_initializer = "uniform", activation = "relu")) # *"second Hidden-layer"*

    ANN\_clsfire**.add**(**Dense**(units = 1, kernel\_initializer = "uniform", activation = "sigmoid")) # *"output-layer"*

    ANN\_clsfire**.compile**(optimizer = "adam", loss = "binary\_crossentropy", metrics= ["accuracy"]) # *Compile the ANN*

**return** ANN\_clsfire

ann\_clsfier\_for\_eval = **KerasClassifier**(build\_fn= build\_ANN\_clsfire, batch\_size= 10, epochs= 100) # *creates/compile an ANN classifier*

acuRacies = **cross\_val\_score**(estimator= ann\_clsfier\_for\_eval, X = X\_train, y = y\_train, cv = 10, n\_jobs = **None**) # *compile ANN 10-times with 10-fold*

mean\_accu = acuRacies**.mean**()

st\_dvi\_accu = acuRacies**.std**()

# *-------- Improving ANN : Dropout-Regularization ------------*

# *Done in "part -2 Creating ANN model"*

# *--------  Tuning the ANN : Grid-Search ------------*

**from** keras**.**wrappers**.**scikit\_learn **import** KerasClassifier     # *to combine Keras & Scikit-learn*

**from** sklearn**.**model\_selection **import** GridSearchCV

# *----------- make a function that creates ANN ------------*

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

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

# *structure of our ANN model*

**def** **build\_ANN\_clsfire**(opTmzr):

    ANN\_clsfire = **Sequential**() # *initialize the ANN*

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

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

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

    ANN\_clsfire**.compile**(optimizer = opTmzr, loss = "binary\_crossentropy", metrics= ["accuracy"])

**return** ANN\_clsfire

ann\_clsfier\_for\_tune = **KerasClassifier**(build\_fn= build\_ANN\_clsfire) # *creates/compile an ANN classifier*

params\_dic = [

    {

        'batch\_size': [25, 32],

        'epochs' : [2, 3],

        'opTmzr' : ["rmsprop"]

    },

    {

        'batch\_size': [24],

        'epochs' : [5],

        'opTmzr': ["adam", "rmsprop"]

    }

]

grid\_sch = **GridSearchCV**(

    estimator= ann\_clsfier\_for\_tune,

    param\_grid= params\_dic,

    scoring= 'accuracy',

    cv = 5

)

grid\_sch = grid\_sch**.fit**(X\_train, y\_train)

best\_parametrs = grid\_sch**.**best\_params\_

best\_accuracy = grid\_sch**.**best\_score\_

**print**(f"\n\tBest parameters = {best\_parametrs} \n\tBest Accuracy = {best\_accuracy}")

* To get even more accuracies you will have several options:

1. Change the architecture of your neural network
2. Do some more parameter tuning.

* Result: best parameters are ***bach size*** of ***25*** a number of ***epoch*** of ***500*** and an ***rmsprop*** optimizer and so it's with these parameters that we manage to get an 85% accuracy.
* Note: You result may be different due to choice of different parameters & parameters value.
* I'll give you some hints and a solution to achieve more accuracy

So much repetition, so we have to manage the code. Our code is so wet.