Sapienza University Of Rome

Feature Extraction Using Deep Convolutional Neural Networks For Offline Signature Verification

Biometrics Systems 2020

Prof. Maria De Marsico

Manoochehr Joodi Bigdello – 1860273

-

1- Introduction to biometric systems

In these days, reliable authentication and authorization of individuals are becoming more essential tasks in several aspects of daily activities and as well as many different important applications in e-commerce, forensic science, engineering, border control, access control, travel and immigration, healthcare, etc. Traditional authentication methods, which are based on knowledge (password-based authentication) or the utility of a token (photo ID cards, magnetic stripe cards, and key-based authentication), are less reliable because of loss, forgetfulness, and theft. These issues direct substantial attention towards biometrics as an alternative method for person authentication and identification. Therefore a biometric is the measurement and statistical analysis of unchanging biological characteristics. Biometrics evaluates a person's unique physical or behavioral traits to authenticate their identity. A large number of biometric traits have been investigated and some of them are nowadays used in several applications. Common physical traits include fingerprints, ear, hand or palm geometry, vein, retina, iris, and facial characteristics. Behavioral traits include voice, signature, keystroke pattern and gait [2].

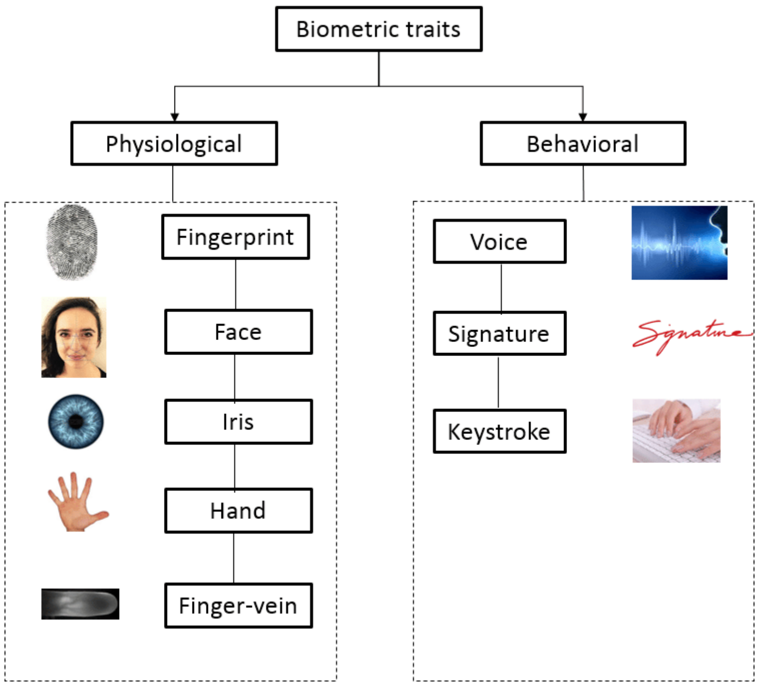


Figure 1. Classification of biometric traits [4]

Biometric authentication system is a system that provides process to identify and verify the user's identity and authenticate using dynamic features of the user because it has unique characteristics distinguish each user from another, also these biometric features must be perfect and unique, they can be evaluated easily, and they are cost-efficient and have great user approval. [3]

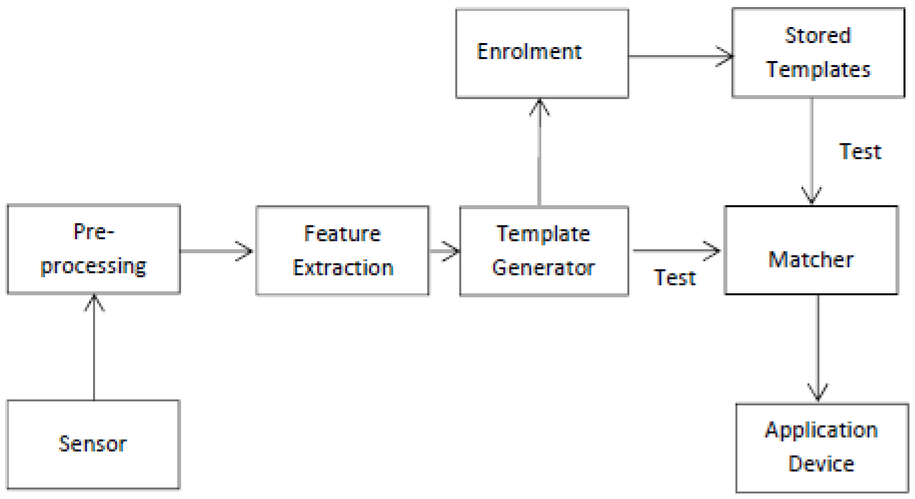


Figure 2. A Generic Biometric System[3]

2- Signature as Biometric

We focused on handwritten signatures as it is still one of the most used methods to verify the identity of a person. We try to tackle the problem in the presence of skilled forgeries, where a forger has access to a person's signature and deliberately attempts to imitate it. In offline (static) signature verification, the dynamic information of the signature writing process is lost, and it is difficult to design good feature extractors that can distinguish genuine signatures and skilled forgeries. [1]. We implemented CNN [1] to extract some features to distinguish not only between two different users but also between the two classes of users i.e. a genuine signature or a forged signature image.

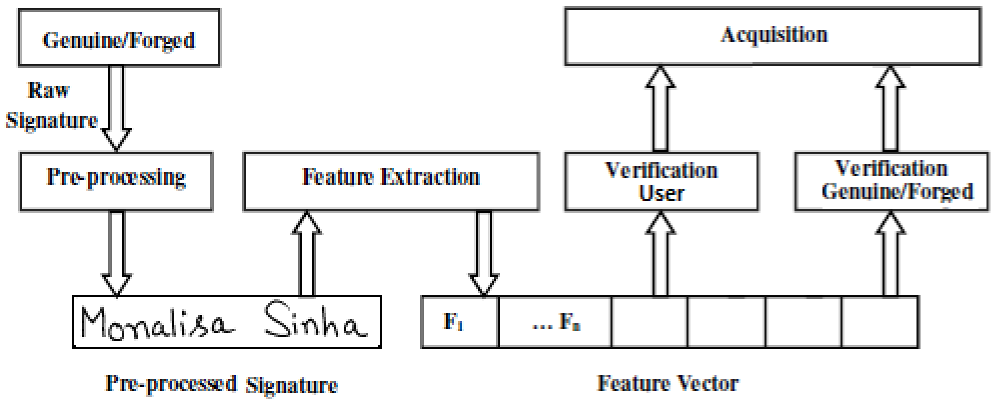


Figure 3. Block Diagram of our Automatic Signature Verification System

For our work we used [1] CNN model as our base to extract features first and we successfully implemented the model and reach to good results in forgery detection but not in user detection with SVM model, so we try different method's that we are going to explain and finally we reach to same results as paper, then we did further investigations and we understand that we can make the models extracted features smaller but keep the results same (only 1% decrease), then we try to implement LSTM model too but the results were not promising and also training time was a lot.

Signature Verification vs. Identification

In the field of automatic signature recognition, two different types of signature recognition systems are considered such as signature verification and signature identification systems. Signature-based biometric technologies are used for either one of those two purposes, verification or identification, and the implementation and selection of the technology and related procedures are closely tied to this aim. Technologies differ in their capabilities and effectiveness in addressing these purposes. Verification (Am I whom I claim I am?) includes confirming or rejecting a person's demanded identity. In identification, someone has to establish a person's identity (Who am I?). Each one of these approaches has its own complexities and could probably be solved by a signature authentication system. These two examples illustrate the difference between the two primary uses of biometrics: identification and verification.

Signature **identification** (1: N, one-to-many, recognition): A signature identification system must recognize a signature from a list of N signatures in the template database. The process of determining a person's identity by performing matches against multiple templates. Identification systems are designed to determine identity-based solely on signature information.

Signature **Verification** (1:1, matching, authentication): A signature verification system simply decides whether a given signature belongs to a claimed signature or not. It is the process of establishing the validity of a claimed identity by comparing a verification template to an enrollment template. Verification needs that individuality to be claimed, after which the individual's enrollment template is located and compared with the verification template. Verification responses the query, 'Am I who I claim to be?' [2]

Dynamic and Static Signature Verification

The biomechanical processes involved in the production of the human signature are very complex. In vastly simplified terms, the main excitation is thought to take place in the central nervous system, more specifically in the human brain, with predefined intensity and duration describing the intent of the movement. The signal of the intent (or the movement plan) is passed through the spinal cord to the particular muscles which are activated in the intended order and intensity. As a result of such activation and relaxation of the muscles and whilst holding a pen, the resultant arm movement is recorded in the form of a trail on paper as a handwritten signature. Based on the handwritten signature data acquisition method, two types of systems for handwritten signature verification can be identified: static (off-line) systems and dynamic (on-line) systems. [2]

Dynamic Signature Verification

Whenever handwriting is captured as a user writes for the purpose of recognition or analysis, it is called on-line handwriting recognition. This process requires special devices, such as stylus or digitizer pen and tablet, to capture the writing information on-the-fly. The temporal stream of information which is extracted as the writing is produced is called on-line features, which include local pressure, acceleration, speed, number of strokes, and order of strokes. The signature image can be simulated with high accuracy using this temporal-spatial feature information. Dynamic signature verification system uses a digitizer or an instrumented pen to give a representation of the written signature generating one or several signals which vary with time. The raw data are then pre-processed to remove spurious information, to filter the significant signals and to validate the acquisition process. The next step involves what is referred to as the feature extraction process. Specific and discriminant functions or parameters are computed from the filtered input data and are used to represent a signature. Dynamic signature verification methods can be classified into two principal groups. In the first group of dynamic signature verification, the techniques deal with functions [2]

Static Signature Verification

When the recognition is undertaken using only the static images of handwriting, the process is called off-line recognition. Despite the unique advantage over its on-line counterpart, as no specialized capture device is required, the amount of information obtained from off-line recognition is two orders greater, but much less meaningful and more difficult to interpret. Moreover, the traces of dynamic information are very difficult to compute. Traditionally, the recovery of such information requires professional skills and techniques whose implementation on computers is not easy [38]. Several evaluations performed by expert document analysts concluded that the detection of forgeries of high skill requires not just static information but also dynamic information, a survey by Plamondon and Lorette [37] reported. Some rare attempts to extract direct pressure information were made by Ammar et al. [40]. With the distinct characteristics mentioned above, on-line recognition systems are able to achieve better results than their off-line counterparts [41]. Off-line systems utilize the classic method of on-paper signatures for person verification. The signature obtained is digitized by an optical scanner or camera. An alternative is to input the image through a tablet or any other suitable device. Subsequently, respective applications determine the match of the person’s signature with a reference sample by comparing the overall trace (image) of the signature. Based on this particular principle, the current very unreliable methods, commonly practiced in banking and retail for example, are utilized to verify handwritten signatures, relying on the human factor in the form of a calligraphy expert. [2]

Types of Forgeries

There are usually three different types of forgeries to take into account. According to Coetzer et al. [42], the three basic types of forged signatures are indicated below:

• Random forgery: The forger is not familiar and has no access to the genuine signature (not even the author's name) and reproduces a random one.

• Simple forgery: The forger is familiar with the author's name, but has no access to a sample of the signature.

• Skilled forgery: The forger has access to one or more samples of the genuine signature and is able to reproduce it. But based on the various skilled levels of forgeries, it can also be divided into six different subsets. [2]

3- Feature learning for Signatures

In this project, we implement formulations for learning features for Offline Signature Verification from [1] and evaluate the performance of such features for training Writer-Dependent classifiers. We also put some steps further and minimize the number of output features 8 times less than the original paper and reach better results (explained in section 4)

Related work [1]

To address both the issue of obtaining a good feature representation for signatures, as well as improving classification performance, they propose a framework for learning the representations directly from the signature images, using Convolutional Neural Networks (CNN). In particular, they propose a novel formulation of the problem, which incorporates knowledge of skilled forgeries from a subset of users, using a **multi-task** learning strategy. The hypothesis is that the model can learn visual cues present in the signature images, that are discriminative between genuine signatures and forgeries in general (i.e. not specific to a particular individual).

Their main contributions are as follows:

1) They present formulations to learn features for offline signature verification in a Writer-Independent format. they introduce a novel formulation that uses skilled forgeries from a subset of users to guide the feature learning process, using a multi-task framework to jointly optimize the model to discriminate between users (addressing random forgeries), and to discriminate between genuine signatures and skilled forgeries.

2) They propose a strict experimental protocol, in which all design decisions are made using a validation set composed of a separate set of users. Generalization performance is estimated in a disjoint set of users, from whom they do not use any forgeries for training.

3) They present a visual analysis of the learned representations, which shows that genuine signatures and skilled forgeries get better separated in different parts of the feature space.

4) Lastly, they made two trained models available for the research community, so that other researchers can use them as specialized feature extractors for the task.

They note that a supervised feature learning approach directly applied for Writer-Dependent classification is not practical (since the number of samples per user is very small), while most feature learning algorithms have a large number of parameters (in the order of millions of parameters, for many computer vision problems, such as object recognition). On the other hand, they expected that signatures from different users share some properties, and they exploited this intuition by learning features across signatures from different writers.

Convolutional Neural Networks (CNN) is a particularly suitable architecture for signature verification. This type of architecture scales better than fully connected models for larger input sizes, having a smaller number of trainable parameters. This is a desirable property for the problem at hand, since we cannot reduce the signature images too much without risking losing the details that enable discriminating between skilled forgeries and genuine signatures (e.g. the quality of the pen strokes), also this type of architecture shares some properties with handcrafted feature extractors used in the literature, as features are extracted locally (in an overlapping grid of patches) and combined in non-linear ways (in subsequent layers). [1]

In CNN architecture used in this work, The input image goes through a sequence of transformations with convolutional layers, max-pooling layers and fully-connected layers. During feature learning, $P(y|X)$ (and also $P(f|X)$ in the formulation from sec 3.2.2) are estimated by performing forward propagation through the model. The weights are optimized by minimizing one of the loss functions defined in the next sections. For new users of the system, this CNN is used to project the signature images onto another feature space (analogous to “extract features”), by performing feed-forward propagation until one of the last layers before the final classification layer, obtaining the feature vector $\phi(X)$.

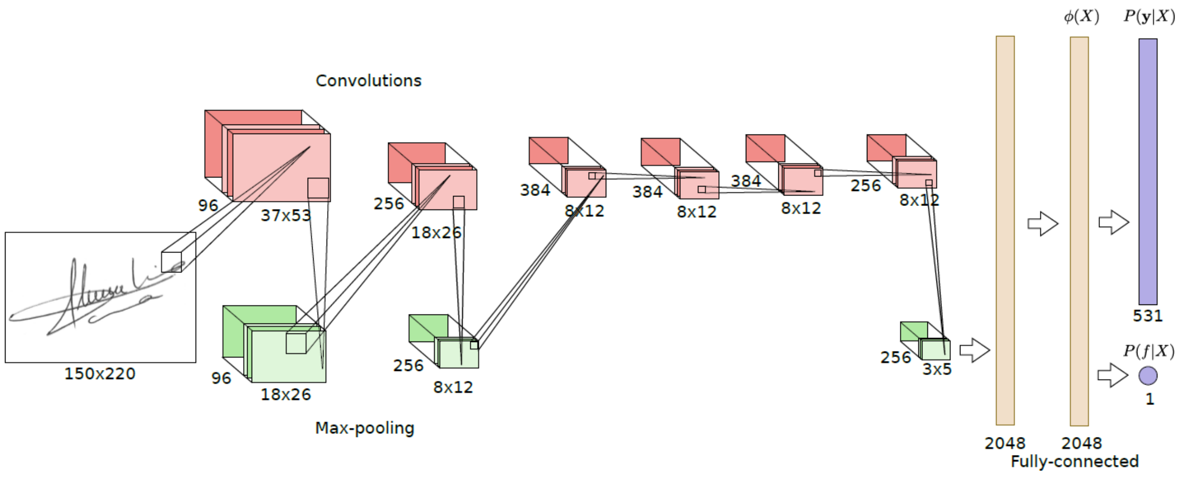


Figure 4. Illustration of the CNN architecture used in original paper.

Multi-task learning framework

Formally, they consider a training set composed of tuples (X; y) where X is the signature image, and y is the user. they create a neural network with multiple layers, where the objective is to discriminate between the users in the Development set. The last layer of the neural network has M units with a softmax activation, where M is the number of users in the Development set, and estimates P(y|X). Figure 4 illustrates one of the architectures used in this work, with M = 531 users. We train the network to minimize the negative log likelihood of the correct user given the signature image:

$$L = -\sum_j y_{ij} \log P(y_j|X_i) \qquad (1)$$

Where y_{ij} is the true target for example i (y_{ij} = 1 if the signature belongs to user j), X_i is the signature image, and $P(y_j|X_i)$ is the probability assigned to class j for the input X_i, given by the model. This cost function can then be minimized with a gradient-based method.

One limitation of the formulation above is that there is nothing in the training process to drive the features to be good in distinguishing skilled forgeries. Since this is one of the main goals of a signature verification system, it would be beneficial to incorporate knowledge about skilled forgeries in the feature learning process.

By Adding a separate output for detecting forgeries and considering two terms in the cost function for feature learning. The first term drives the model to distinguish between different users (as in the formulations above), while the second term drives the model to distinguish between genuine signatures and skilled forgeries. Formally, we consider another output of the model: P(f|X), a single sigmoid unit, that seeks to predict whether or not the signature is a forgery. The intuition is that in order to classify between genuine signatures and forgeries (regardless of the user), the network will need to learn visual cues that are particular to each class (e.g. bad line quality in the pen strokes, often present in forgeries).

they consider a training dataset containing tuples of the form (X, y, f), where X is the signature image, y is the author of the signature (or the target user, if the signature is a forgery), and f is a binary variable that reflects if the sample is a forgery or not (f = 1 indicates a forgery). Note that contrary to the previous formulation, genuine signatures and forgeries targeted to the same user have the same y. For training the model, they consider a loss function that combines both the classification loss (correctly classifying the user), and a loss on the binary neuron that predicts whether or not the signature is a forgery. The individual losses are shown in Equation 2, where the user classification loss (L_c) is a multi-class cross-entropy, and the forgery classification (L_f) is a binary cross-entropy:

$$\begin{aligned} L_c &= -\sum_j y_{ij} \log P(y_j|X_i) \\ L_f &= -f_i \log(P(f|X_i)) - (1-f_i)\log(1-P(f|X_i)) \end{aligned} \tag{2}$$

For training the model, they combine the two loss functions and minimize both at the same time. they considered two approaches for combining the losses. The first approach considers a weighted sum of both individual losses:

$$\begin{aligned} L_1 &= (1-\lambda)L_c + \lambda L_f \\ &= -(1-\lambda)\sum_j y_{ij} \log P(y_j|X_i) + \\ &\lambda\big(-f_i \log(P(f|X_i)) - (1-f_i)\log(1-P(f|X_i))\big) \end{aligned} \tag{3}$$

Where λ is a hyper parameter that trades-off between the two objectives (separating the users in the dataset, and detecting forgeries)

In a second approach they consider the user classification loss only for genuine signatures:

$$
\begin{aligned}
L_2 &= (1-f_i)(1-\lambda)L_c + \lambda L_f \\
&= -(1-f_i)(1-\lambda)\sum_j y_{ij}\log P(y_j|X_i) + \\
&\lambda\big(-f_i\log(P(f|X_i)) - (1-f_i)\log(1-P(f|X_i))\big)
\end{aligned}
\tag{4}
$$

In this case, the model is not penalized for misclassifying for which user a forgery was made.

In both cases, the expectation is that the first term will guide the model to learn features that can distinguish between different users (i.e. detect random forgeries), while the second term will focus on particular characteristics that distinguish between genuine signatures and forgeries.

Convolutional Neural Network training

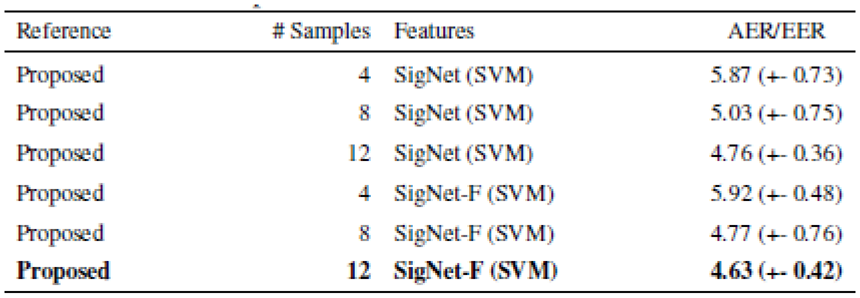
In the paper they used the architecture that performed best for this formulation, which is described in table 1. The CNN consists of multiple layers, considering the following operations: convolutions, max-pooling and dot products (fully-connected layers), where convolutional layers and fully-connected layers have learnable parameters, that are optimized during training. With the exception of the last layer in the network, after each learnable layer they applied Batch Normalization, followed by the ReLU non-linearity.

Table 1: Summary of the CNN layers

Layer	Size	Other Parameters
Input	1x150x220	
Convolution (C1)	96x11x11	stride = 4, pad=0
Pooling	96x3x3	stride = 2
Convolution (C2)	256x5x5	stride = 1, pad=2
Pooling	256x3x3	stride = 2
Convolution (C3)	384x3x3	stride = 1, pad=1
Convolution (C4)	384x3x3	stride = 1, pad=1
Convolution (C5)	256x3x3	stride = 1, pad=1
Pooling	256x3x3	stride = 2
Fully Connected (FC6)	2048	
Fully Connected (FC7)	2048	
Fully Connected + Softmax ($P(\mathbf{y}\|X)$)	M	
Fully Connected + Sigmoid ($P(f\|X)$)	1	

Optimization was conducted by minimizing the loss with Stochastic Gradient Descent with Nesterov Momentum, using mini-batches of size 32, and momentum factor of 0.9. As regularization, they applied L2 penalty with weight decay 10e-4. The models were trained for 60 epochs, with an initial learning rate of 10e-3, that was divided by 10 every 20 epochs. We used simple translations as data augmentation, by using random crops of size 150x220 from the 170x242 signature image.

They used SVM for classification at the end and report the results that is better than state of the art approaches in literature.



Reference	# Samples	Features	AER/EER
Proposed	4	SigNet (SVM)	5.87 (+- 0.73)
Proposed	8	SigNet (SVM)	5.03 (+- 0.75)
Proposed	12	SigNet (SVM)	4.76 (+- 0.36)
Proposed	4	SigNet-F (SVM)	5.92 (+- 0.48)
Proposed	8	SigNet-F (SVM)	4.77 (+- 0.76)
Proposed	**12**	**SigNet-F (SVM)**	**4.63 (+- 0.42)**

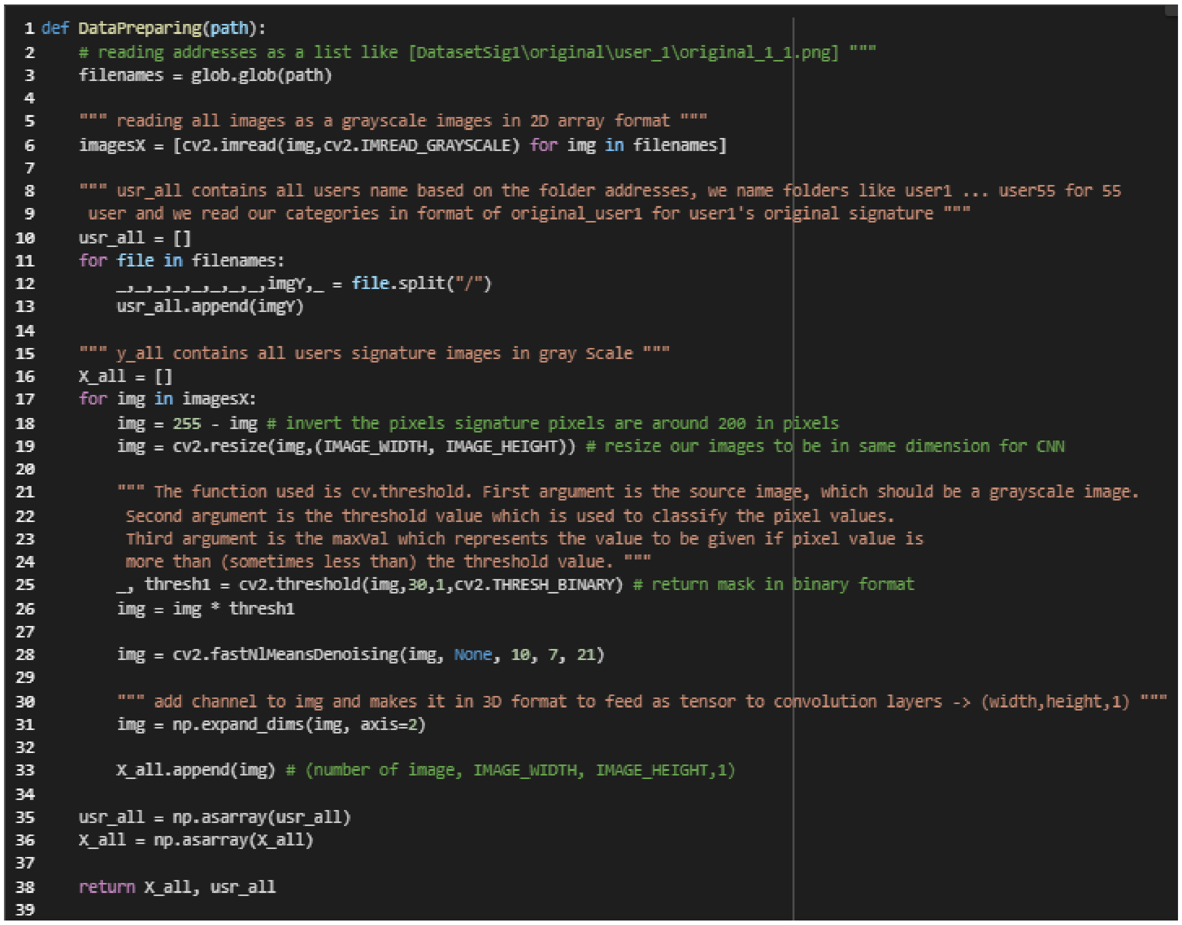
4- Our Implementation

We didn't follow all parts of paper and we just tried to implement their method with deep learning architectures like keras and tensorflow, so we just follow their feature extraction architecture to extract features and then we did some modification in their architecture and minimize the number of output features 8 times less than original paper and reach to good result.

Preprocessing

First we read all images and perform preprocessing with openCV in python like:

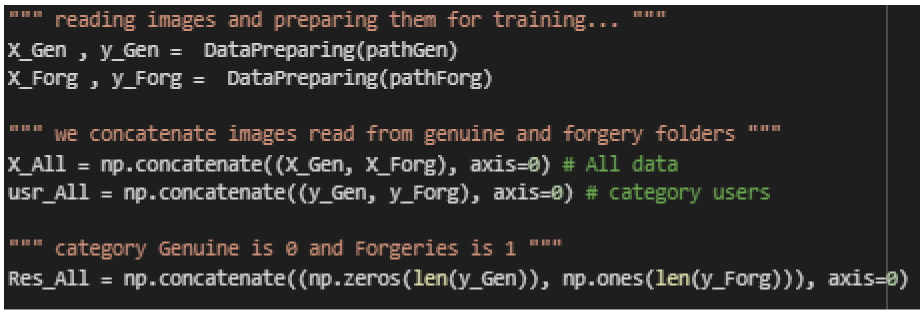
1. reading all images as a grayscale in 2D array format with openCV
2. extract users names from folder name of image addresses
3. for all images:
   - First we inverted each image by subtracting each pixel from the maximum brightness in grey scale format (image = 255 – image), such that the background is zero-valued.
   - Then we resize the image to the input size of the network. (cv2.resize (image, (200,320))).
   - Put threshold to remove some noises, we put 0 pixels less than 30 as they are noises (we get 30 by trying some numbers like 20, 30, 40, and 50).
   - Then we perform cv2.fastNlMeansDenoising function for making signatures without any noise (we tried different parameters)
   - Then to convert our Images to 3D format to feed in CNN, we used np.expand_dims() function of numpy library of python.
   - At the end we changed all arrays to numpy array.



```python
def DataPreparing(path):
    # reading addresses as a list like [DatasetSig1\original\user_1\original_1_1.png] """
    filenames = glob.glob(path)

    """ reading all images as a grayscale images in 2D array format """
    imagesX = [cv2.imread(img,cv2.IMREAD_GRAYSCALE) for img in filenames]

    """ usr_all contains all users name based on the folder addresses, we name folders like user1 ... user55 for 55
     user and we read our categories in format of original_user1 for user1's original signature """
    usr_all = []
    for file in filenames:
        _,_,_,_,_,_,_,_,imgY,_ = file.split("/")
        usr_all.append(imgY)

    """ y_all contains all users signature images in gray Scale """
    X_all = []
    for img in imagesX:
        img = 255 - img # invert the pixels signature pixels are around 200 in pixels
        img = cv2.resize(img,(IMAGE_WIDTH, IMAGE_HEIGHT)) # resize our images to be in same dimension for CNN

        """ The function used is cv.threshold. First argument is the source image, which should be a grayscale image.
         Second argument is the threshold value which is used to classify the pixel values.
         Third argument is the maxVal which represents the value to be given if pixel value is
         more than (sometimes less than) the threshold value. """
        _, thresh1 = cv2.threshold(img,30,1,cv2.THRESH_BINARY) # return mask in binary format
        img = img * thresh1

        img = cv2.fastNlMeansDenoising(img, None, 10, 7, 21)

        """ add channel to img and makes it in 3D format to feed as tensor to convolution layers -> (width,height,1) """
        img = np.expand_dims(img, axis=2)

        X_all.append(img) # (number of image, IMAGE_WIDTH, IMAGE_HEIGHT,1)

    usr_all = np.asarray(usr_all)
    X_all = np.asarray(X_all)

    return X_all, usr_all

```

Figure 5. Data preparing code in python

Our training dataset CEDAR contains separate images for genuine and forgery for each user (2640 signature of 55 user, 24 genuine and 24 forgery signature for each user). To prepare the dataset for training, we take all the data i.e. the genuine and the forgery images with their labels. Since we have a multi-output classification network with a single input, we merge the genuine and forgery images together to make the input vector of signature images, we also make separate label vectors for each output, The label vector for the **User** layer contains the user labels for each signature image in one hot encoded format and the label vector for the **Res** (Genuine =0, forgery = 1) layer is a binary.



```python
""" reading images and preparing them for training... """
X_Gen , y_Gen =  DataPreparing(pathGen)
X_Forg , y_Forg =  DataPreparing(pathForg)

""" we concatenate images read from genuine and forgery folders """
X_All = np.concatenate((X_Gen, X_Forg), axis=0) # All data
usr_All = np.concatenate((y_Gen, y_Forg), axis=0) # category users

""" category Genuine is 0 and Forgeries is 1 """
Res_All = np.concatenate((np.zeros(len(y_Gen)), np.ones(len(y_Forg))), axis=0)
```

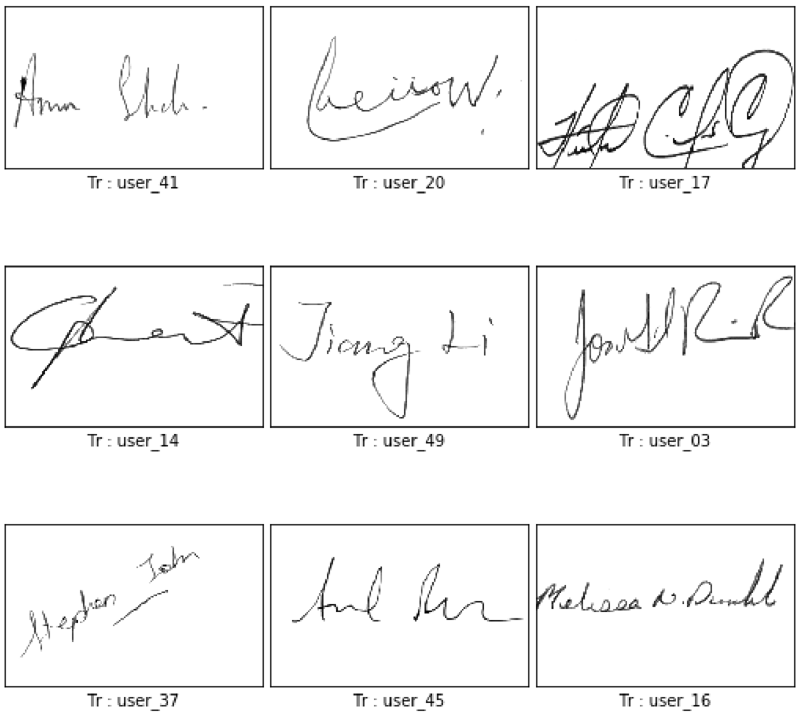


Figure 6. Sample images after preprocessing

The data is split accordingly into train and test sets, we used 25% of data as test set and also to have equal distribution of each users signature to train we used "stratify" functionality and from 24 image we used 18 for train and 6 for test for both genuine and forgery users.

Modified CNN Architecture

After preparing our images to input format of the network, then we implement the exact model and reach to good results that is mentioned before, then we tried to do some modifications to minimize the number of output features as it was two 2048 size FC layer, we understand that instead of these 2 FC layer we can put 3 FC layer of size 2048, 1024 and 256 and it speed up the learning process as number of learnable parameters in our CNN model is half of original model.

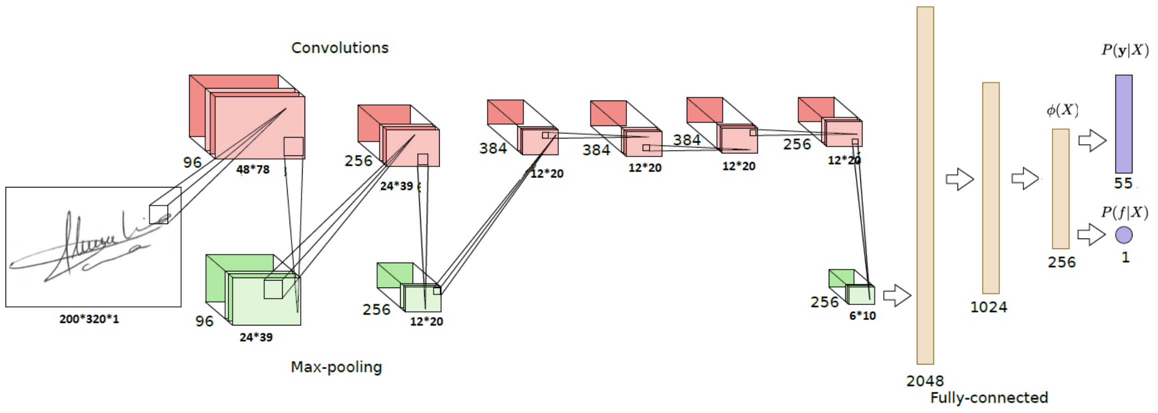


Figure 7. Our proposed CNN architecture

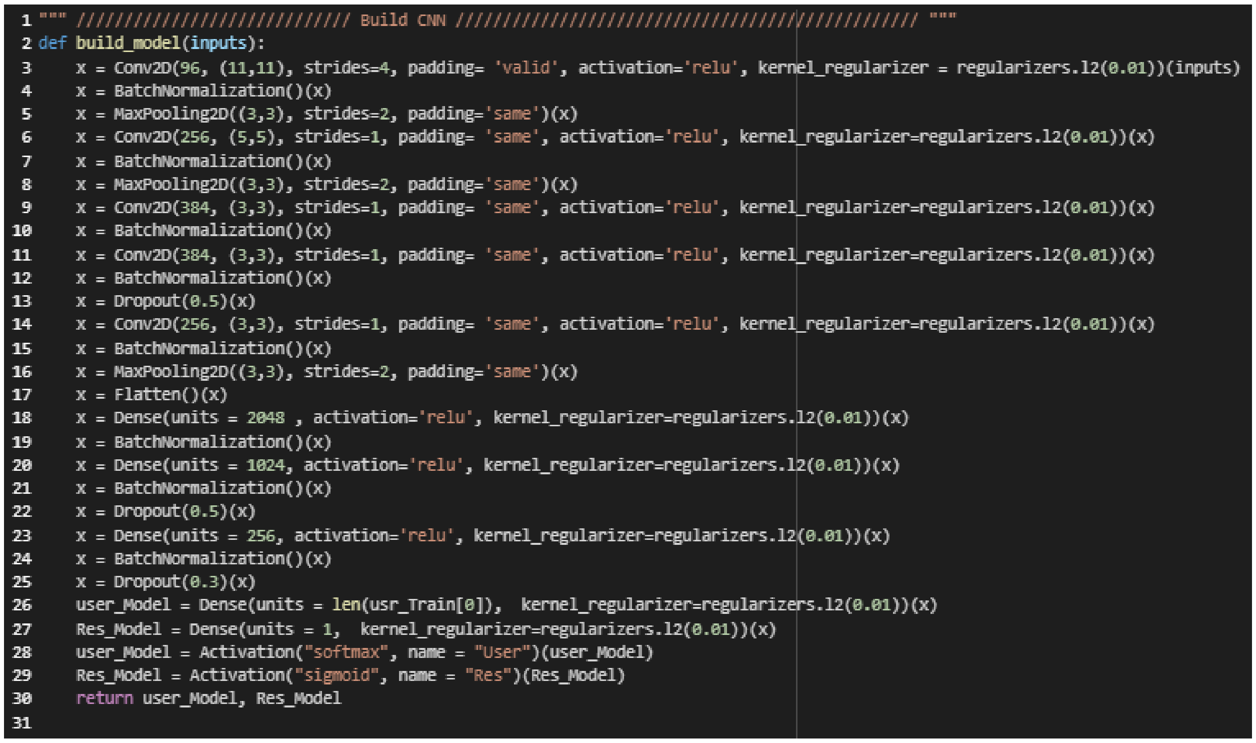
We also added dropout between some of layers, batch Normalization and also l2 regularization penalty to prevent overfitting of model.

Table 2: Summary of the CNN layers

Layer	Size	Other Parameters
Input	1x150x220	
Convolution (C1)	96x11x11	stride = 4, pad=0
Pooling	96x3x3	stride = 2
Convolution (C2)	256x5x5	stride = 1, pad=2
Pooling	256x3x3	stride = 2
Convolution (C3)	384x3x3	stride = 1, pad=1
Convolution (C4)	384x3x3	stride = 1, pad=1
Convolution (C5)	256x3x3	stride = 1, pad=1
Pooling	256x3x3	stride = 2
Fully Connected (FC6)	2048	
Fully Connected (FC7)	1024	
Fully Connected (FC8)	256	
Fully Connected + Softmax ($P(\mathbf{y}|X)$)	M	
Fully Connected + Sigmoid ($P(f|X)$)	1	

Multi-Output CNN Model Implementation

In our project we use Deep Convolutional Neural Networks to extract important features from genuine signature images. To initially train our network, our dataset D consists of 24 genuine images for 55 different users representing each user enrolled in our given system, although note that the implementation presented will work with almost any number of users enrolled in a system. Since most of the organizations would have employers in the range of 10-100, training our network to work on a dataset of 55 users with only 24 images per user looks to tackle the problem of not having a large amount of data which we usually require in order to train highly efficient neural networks. We train the network using these 1320 (24 * 55) genuine signature images with the idea that if our network is able to learn a feature space where all the 55 users are linearly separable then we can assert that our network has learned a good number of visual cues that are found in all signatures. Table 2 and Figure 6 gives the architecture of the used Convolutional Neural Network. The network creates 5 feature maps after the input is passed after each convolutional layer with regularization of 0.01. After each convolutional layer, batch normalization is applied. We implemented our CNN model in Keras tensorFlow.

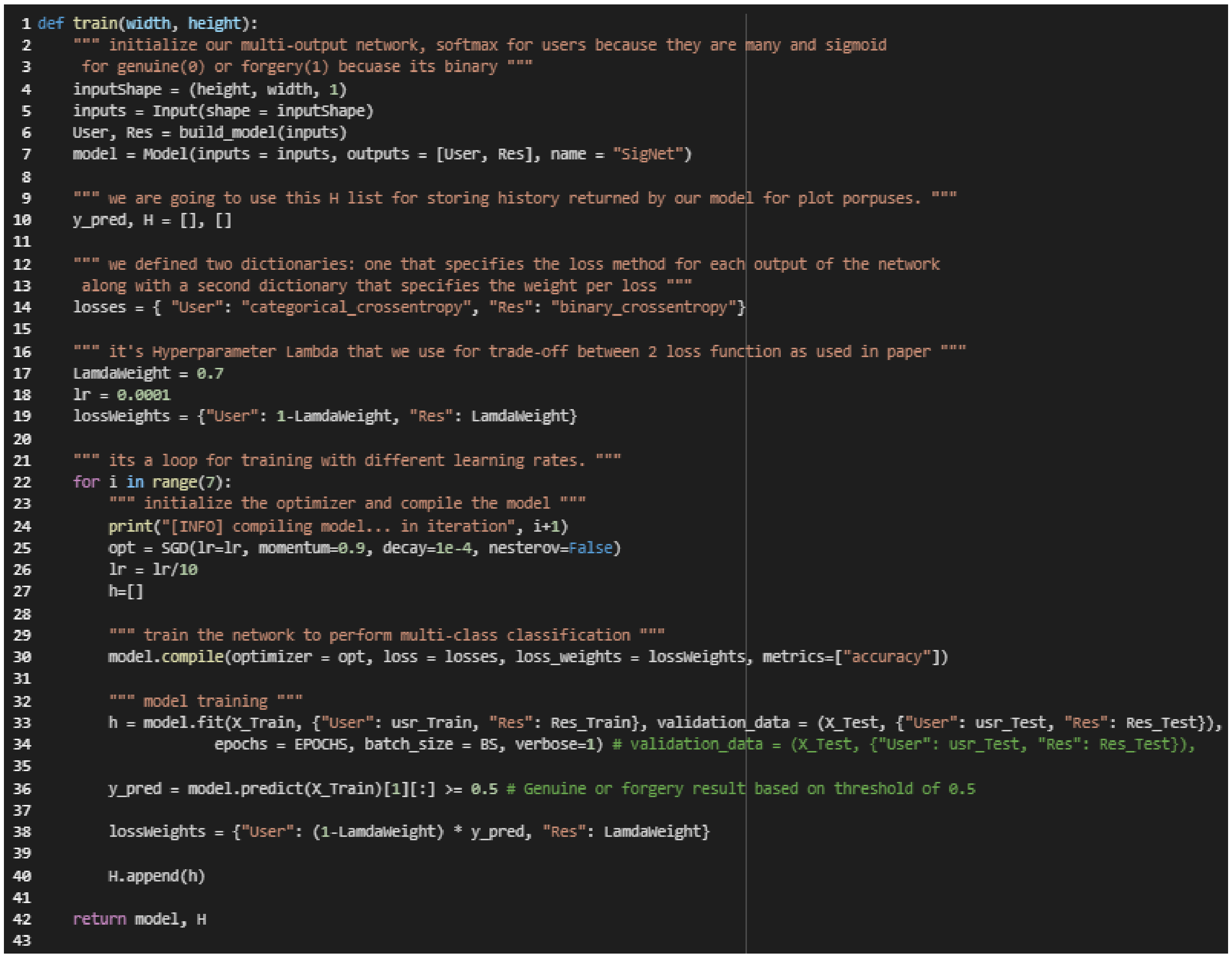


```python
""" ////////////////////////////// Build CNN ///////////////////////////////////////////////////// """
def build_model(inputs):
    x = Conv2D(96, (11,11), strides=4, padding= 'valid', activation='relu', kernel_regularizer = regularizers.l2(0.01))(inputs)
    x = BatchNormalization()(x)
    x = MaxPooling2D((3,3), strides=2, padding='same')(x)
    x = Conv2D(256, (5,5), strides=1, padding= 'same', activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((3,3), strides=2, padding='same')(x)
    x = Conv2D(384, (3,3), strides=1, padding= 'same', activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Conv2D(384, (3,3), strides=1, padding= 'same', activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    x = Conv2D(256, (3,3), strides=1, padding= 'same', activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((3,3), strides=2, padding='same')(x)
    x = Flatten()(x)
    x = Dense(units = 2048 , activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dense(units = 1024, activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.5)(x)
    x = Dense(units = 256, activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dropout(0.3)(x)
    user_Model = Dense(units = len(usr_Train[0]),  kernel_regularizer=regularizers.l2(0.01))(x)
    Res_Model = Dense(units = 1,  kernel_regularizer=regularizers.l2(0.01))(x)
    user_Model = Activation("softmax", name = "User")(user_Model)
    Res_Model = Activation("sigmoid", name = "Res")(Res_Model)
    return user_Model, Res_Model

```

Figure 8. Python code for our CNN model

Training our model

We have setup our network as a multi output network with a single input. One output called the **User** which contains one node for each enrolled user in the system, it's used to classify the user given the input signature, and the other output **Res** is a single node output that only classifies whether the given input signature is a Genuine or a Forgery. Similarly, we have used two different loss functions, one for each outputs, we use the categorical_crossentropy for the **User** output layer and binary_crossentropy for **Res** output layer. We also assign different loss weights to both of the loss functions as mentioned in formula 4 from paper to calculate the loss function, also the paper indicated that best results achieved when they used the lambda weight as 0.9999 but in our case after a lot of tuning best results achieved when we used 0.7. We used Stochastic Gradient Descent with Learning rate started from 0.0001 and divided it by 10 every 30 epochs, our batch size was 64, other hyper parameter we used as paper as their results was promising. Overall we trained 7*40=280 epochs.

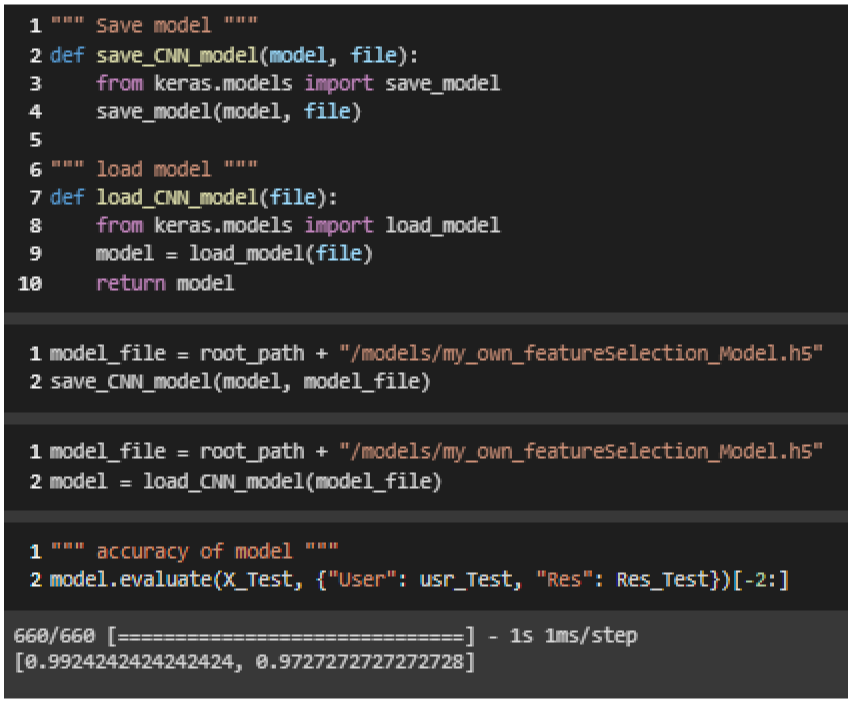


```python
def train(width, height):
    """ initialize our multi-output network, softmax for users because they are many and sigmoid
     for genuine(0) or forgery(1) becuase its binary """
    inputShape = (height, width, 1)
    inputs = Input(shape = inputShape)
    User, Res = build_model(inputs)
    model = Model(inputs = inputs, outputs = [User, Res], name = "SigNet")

    """ we are going to use this H list for storing history returned by our model for plot porpuses. """
    y_pred, H = [], []

    """ we defined two dictionaries: one that specifies the loss method for each output of the network
     along with a second dictionary that specifies the weight per loss """
    losses = { "User": "categorical_crossentropy", "Res": "binary_crossentropy"}

    """ it's Hyperparameter Lambda that we use for trade-off between 2 loss function as used in paper """
    LamdaWeight = 0.7
    lr = 0.0001
    lossWeights = {"User": 1-LamdaWeight, "Res": LamdaWeight}

    """ its a loop for training with different learning rates. """
    for i in range(7):
        """ initialize the optimizer and compile the model """
        print("[INFO] compiling model... in iteration", i+1)
        opt = SGD(lr=lr, momentum=0.9, decay=1e-4, nesterov=False)
        lr = lr/10
        h=[]

        """ train the network to perform multi-class classification """
        model.compile(optimizer = opt, loss = losses, loss_weights = lossWeights, metrics=["accuracy"])

        """ model training """
        h = model.fit(X_Train, {"User": usr_Train, "Res": Res_Train}, validation_data = (X_Test, {"User": usr_Test, "Res": Res_Test}),
                      epochs = EPOCHS, batch_size = BS, verbose=1) # validation_data = (X_Test, {"User": usr_Test, "Res": Res_Test}),

        y_pred = model.predict(X_Train)[1][:] >= 0.5 # Genuine or forgery result based on threshold of 0.5

        lossWeights = {"User": (1-LamdaWeight) * y_pred, "Res": LamdaWeight}

        H.append(h)

    return model, H

```

Figure 9. Python code for our model training

Save, Load and Evaluation of keras model

Model Save, load functions with evaluation of model on our test set of 660 signature (55 users, 6 genuine and 6 forgery image for each user), user accuracy is 99.24 and Res accuracy is 97.27 with threshold of 0.5.



```python
""" Save model """
def save_CNN_model(model, file):
    from keras.models import save_model
    save_model(model, file)

""" load model """
def load_CNN_model(file):
    from keras.models import load_model
    model = load_model(file)
    return model
```

```python
model_file = root_path + "/models/my_own_featureSelection_Model.h5"
save_CNN_model(model, model_file)
```

```python
model_file = root_path + "/models/my_own_featureSelection_Model.h5"
model = load_CNN_model(model_file)
```

```python
""" accuracy of model """
model.evaluate(X_Test, {"User": usr_Test, "Res": Res_Test})[-2:]
```

```
660/660 [==============================] - 1s 1ms/step
[0.9924242424242424, 0.9727272727272728]
```

Figure 10. Python code for Save and Load Keras model

Results from Keras Callback history

After evaluation of keras model we write a code to see the results in all epochs to track the performance and learning of our model in each epoch. We realize that the fluctuation in accuracy and loss plot in certain epochs like 40 and 80 is because we are dividing learning rate by 10 in that epochs and model gradient change maybe the direction to learn other characteristics of signatures, we train model with skipping first 80 epoch but results was not promising and we conclude that in first 80 epoch model learns some of characteristics that is necessary for further epochs to learn.

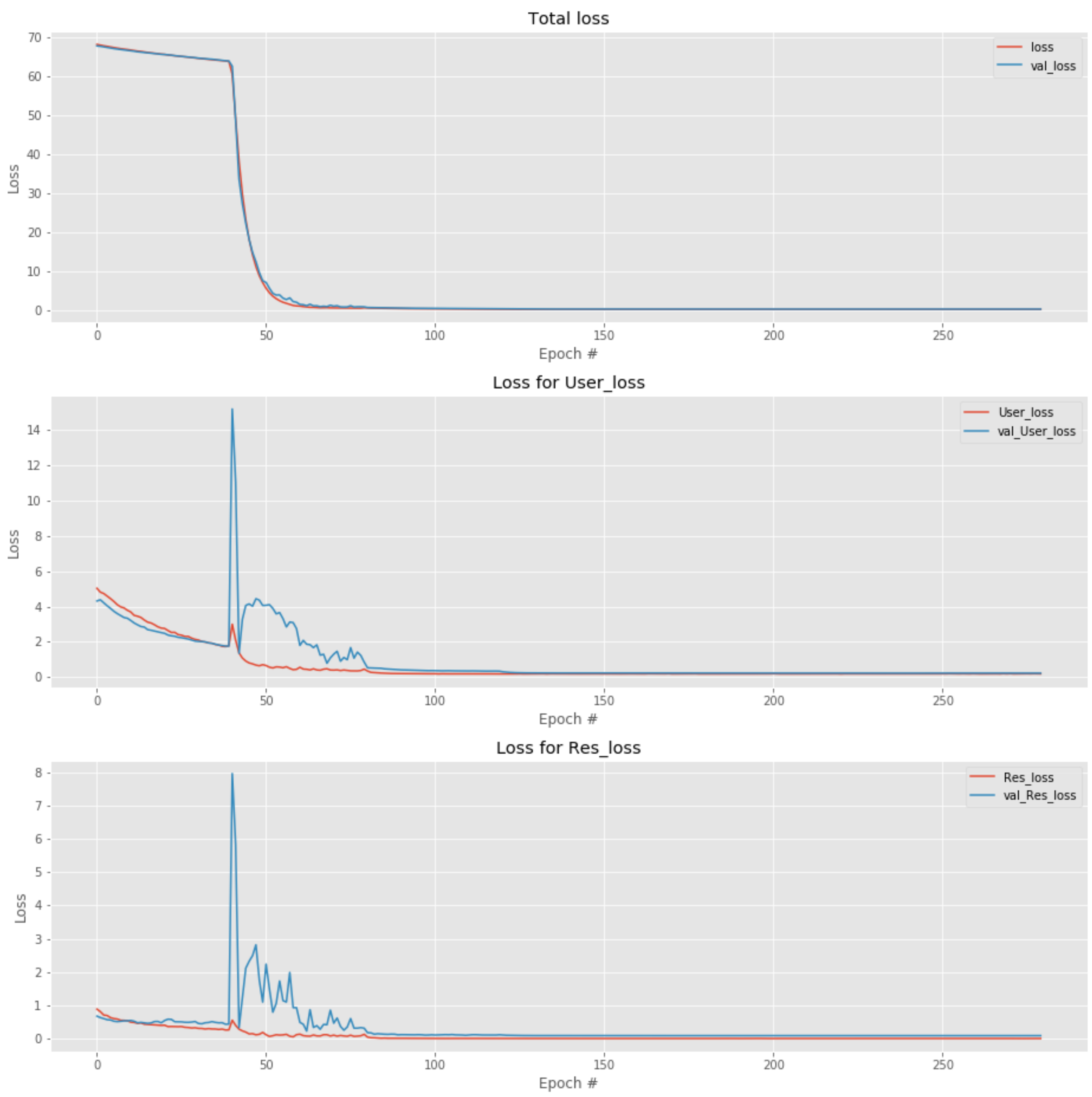


Figure 11. Loss in all epochs for total loss, User loss and Res loss for both train and validation set

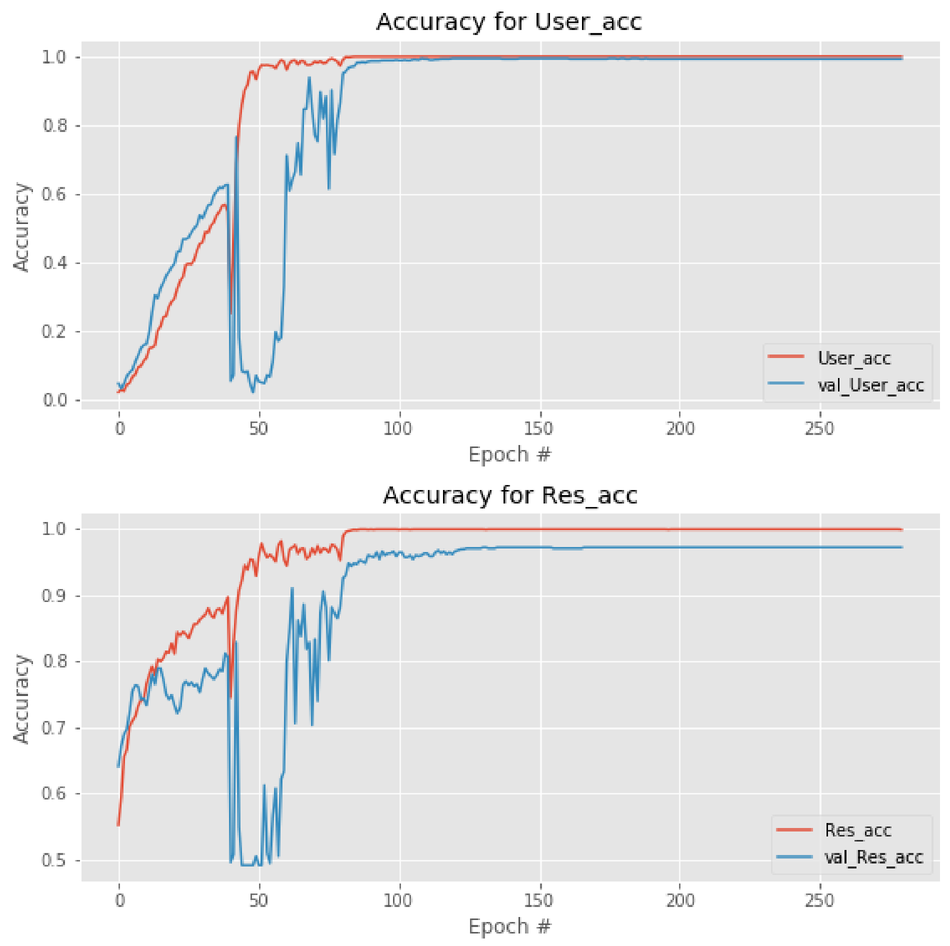
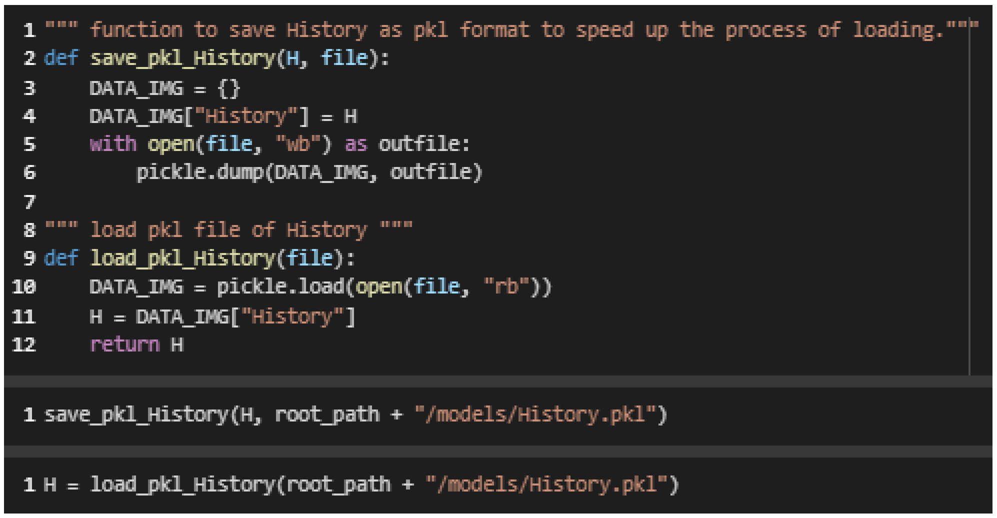


Figure 12. Accuracy in all epochs for User Accuracy and Res Accuracy for both train and validation set

Functions for Save and load history callback of keras as pkl file



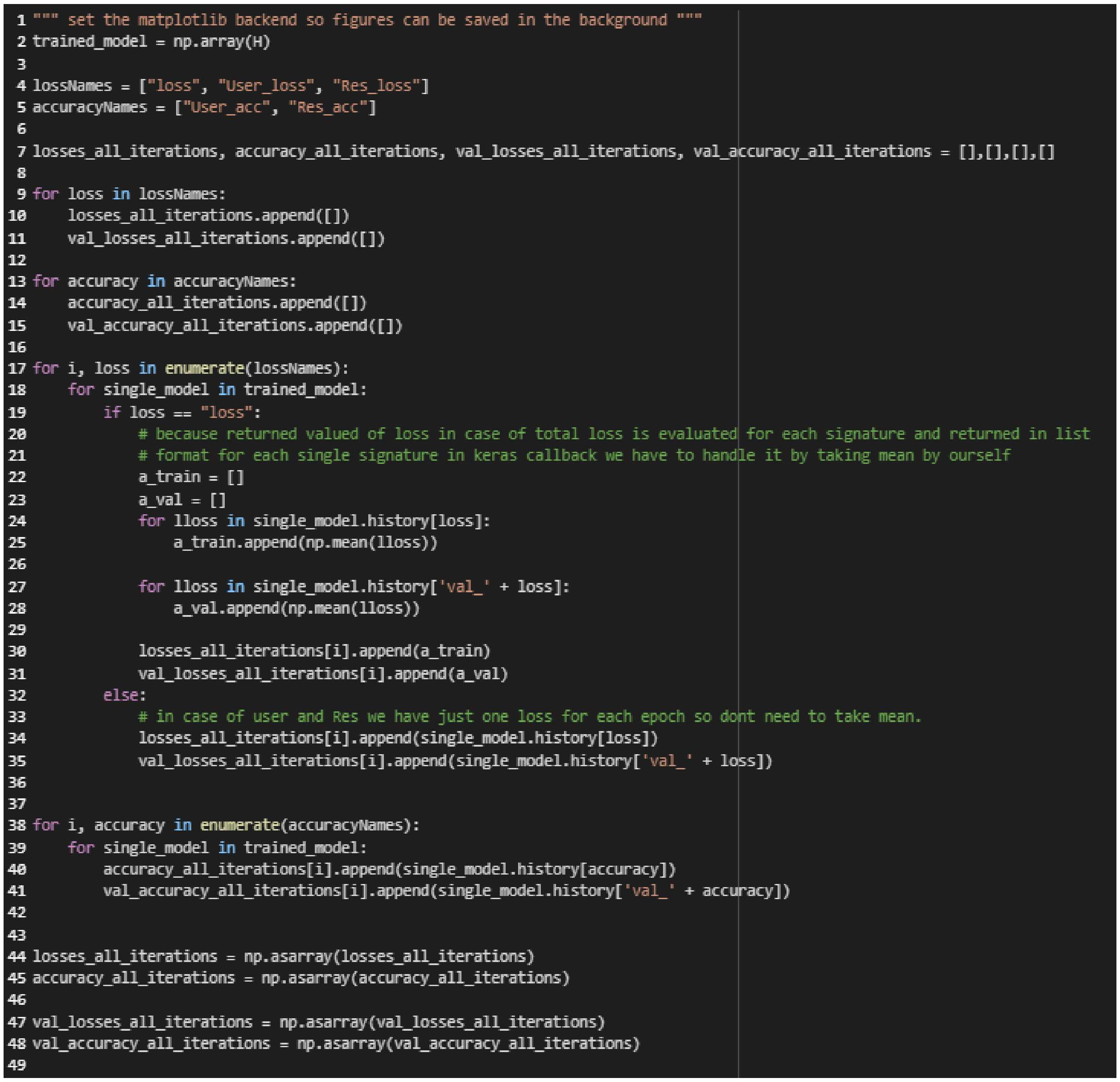
```
""" function to save History as pkl format to speed up the process of loading."""
def save_pkl_History(H, file):
    DATA_IMG = {}
    DATA_IMG["History"] = H
    with open(file, "wb") as outfile:
        pickle.dump(DATA_IMG, outfile)

""" load pkl file of History """
def load_pkl_History(file):
    DATA_IMG = pickle.load(open(file, "rb"))
    H = DATA_IMG["History"]
    return H
```

```
save_pkl_History(H, root_path + "/models/History.pkl")
```

```
H = load_pkl_History(root_path + "/models/History.pkl")
```

Figure 13. Save and load history callback

Because we used for loop and stored all callbacks in list, we have to write some code to extract all loss and accuracy from callbacks.

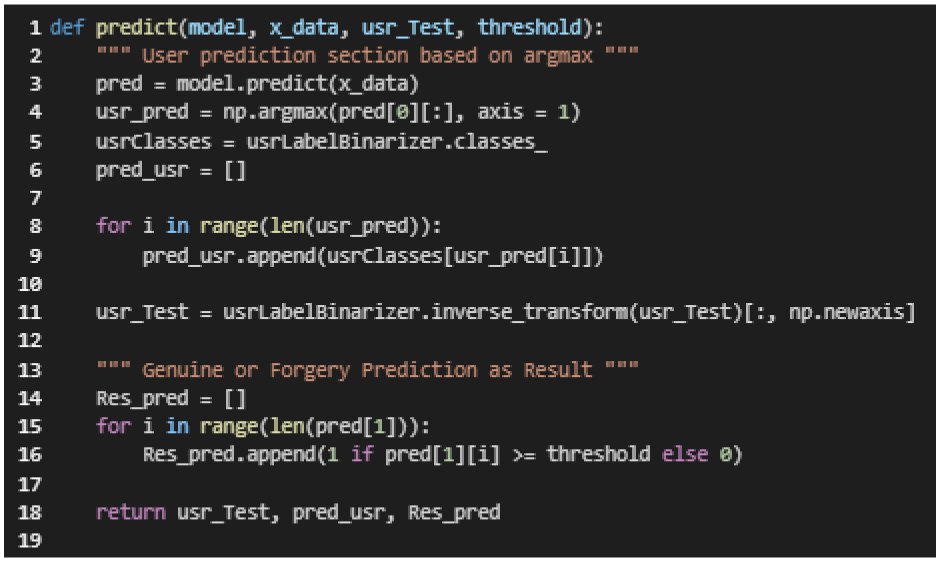


```python
""" set the matplotlib backend so figures can be saved in the background """
trained_model = np.array(H)

lossNames = ["loss", "User_loss", "Res_loss"]
accuracyNames = ["User_acc", "Res_acc"]

losses_all_iterations, accuracy_all_iterations, val_losses_all_iterations, val_accuracy_all_iterations = [],[],[],[]

for loss in lossNames:
    losses_all_iterations.append([])
    val_losses_all_iterations.append([])

for accuracy in accuracyNames:
    accuracy_all_iterations.append([])
    val_accuracy_all_iterations.append([])

for i, loss in enumerate(lossNames):
    for single_model in trained_model:
        if loss == "loss":
            # because returned valued of loss in case of total loss is evaluated for each signature and returned in list
            # format for each single signature in keras callback we have to handle it by taking mean by ourself
            a_train = []
            a_val = []
            for lloss in single_model.history[loss]:
                a_train.append(np.mean(lloss))

            for lloss in single_model.history['val_' + loss]:
                a_val.append(np.mean(lloss))

            losses_all_iterations[i].append(a_train)
            val_losses_all_iterations[i].append(a_val)
        else:
            # in case of user and Res we have just one loss for each epoch so dont need to take mean.
            losses_all_iterations[i].append(single_model.history[loss])
            val_losses_all_iterations[i].append(single_model.history['val_' + loss])


for i, accuracy in enumerate(accuracyNames):
    for single_model in trained_model:
        accuracy_all_iterations[i].append(single_model.history[accuracy])
        val_accuracy_all_iterations[i].append(single_model.history['val_' + accuracy])


losses_all_iterations = np.asarray(losses_all_iterations)
accuracy_all_iterations = np.asarray(accuracy_all_iterations)

val_losses_all_iterations = np.asarray(val_losses_all_iterations)
val_accuracy_all_iterations = np.asarray(val_accuracy_all_iterations)

```

Figure 14. Code for plot accuracy and loss returned by keras model

5- Test and Evaluations

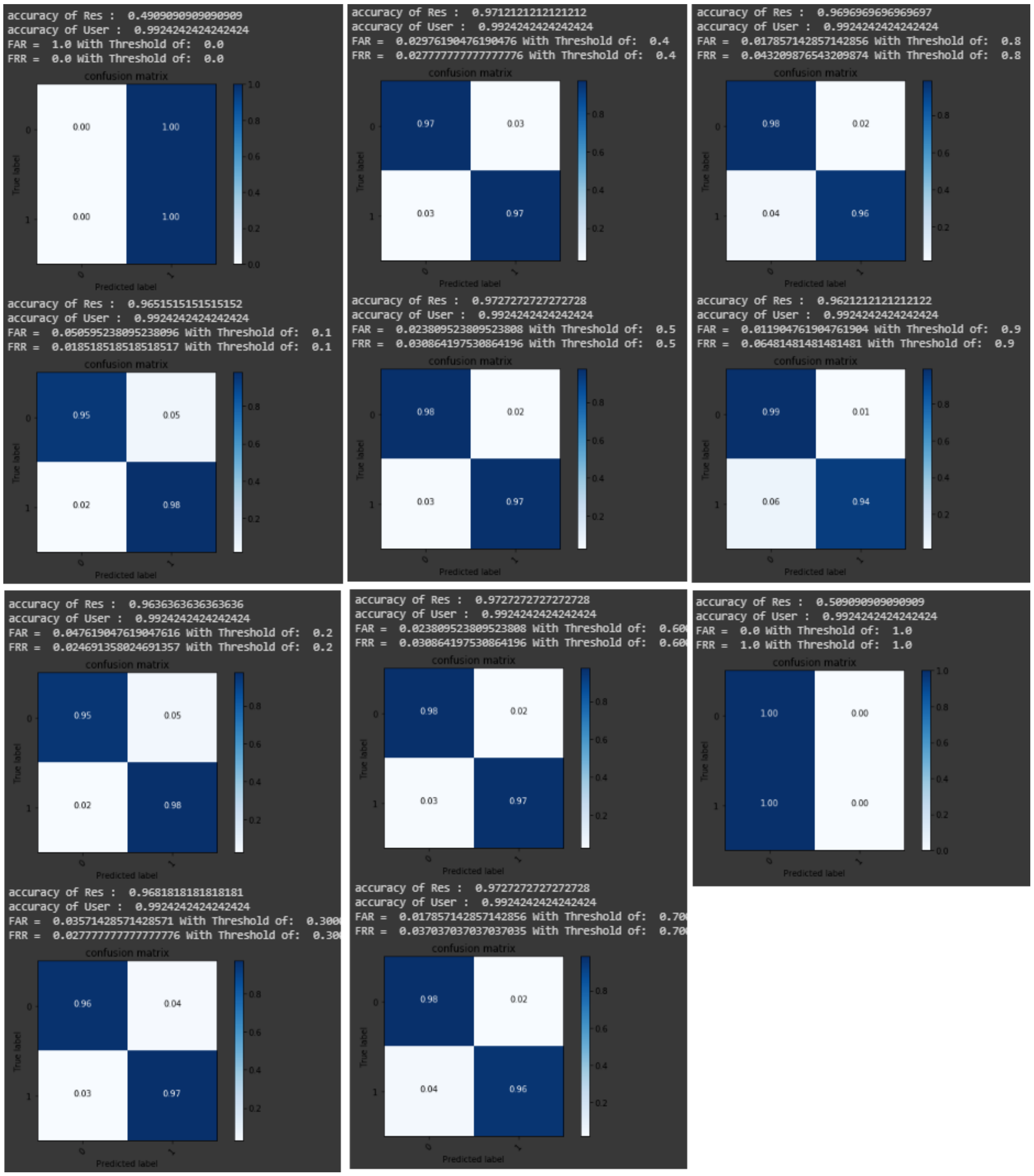
We used argmax to get user's number from predicted list and then convert it to users name by inverse transform of our label binarizer, also for getting forgery or genuine we used different thresholds to see which one is better based on applications need as in some cases the FAR must be as minimum as possible because of security and in some cases FRR must be minimum because of making access easy and also we get EER and ROC and DET curves to show our results in different cases.



```python
def predict(model, x_data, usr_Test, threshold):
    """ User prediction section based on argmax """
    pred = model.predict(x_data)
    usr_pred = np.argmax(pred[0][:], axis = 1)
    usrClasses = usrLabelBinarizer.classes_
    pred_usr = []

    for i in range(len(usr_pred)):
        pred_usr.append(usrClasses[usr_pred[i]])

    usr_Test = usrLabelBinarizer.inverse_transform(usr_Test)[:, np.newaxis]

    """ Genuine or Forgery Prediction as Result """
    Res_pred = []
    for i in range(len(pred[1])):
        Res_pred.append(1 if pred[1][i] >= threshold else 0)

    return usr_Test, pred_usr, Res_pred

```

Figure 15. Code for predict both outputs (User and Res)

For Res we define a threshold list [0, 0.1, …,1] and we predict Res for each threshold and calculate FRR, FAR, accuracy of user and Res with confusion matrix plot of them to have a general view of system to pick the best threshold for system based on the applications security level.



```
accuracy of Res :  0.4909090909090909
accuracy of User :  0.9924242424242424
FAR =  1.0 With Threshold of:  0.0
FRR =  0.0 With Threshold of:  0.0
```

```
accuracy of Res :  0.9651515151515152
accuracy of User :  0.9924242424242424
FAR =  0.050595238095238096 With Threshold of:  0.1
FRR =  0.018518518518518517 With Threshold of:  0.1
```

```
accuracy of Res :  0.9712121212121212
accuracy of User :  0.9924242424242424
FAR =  0.02976190476190476 With Threshold of:  0.4
FRR =  0.027777777777777776 With Threshold of:  0.4
```

```
accuracy of Res :  0.9727272727272728
accuracy of User :  0.9924242424242424
FAR =  0.023809523809523808 With Threshold of:  0.5
FRR =  0.030864197530864196 With Threshold of:  0.5
```

```
accuracy of Res :  0.9696969696969697
accuracy of User :  0.9924242424242424
FAR =  0.017857142857142856 With Threshold of:  0.8
FRR =  0.043209876543209874 With Threshold of:  0.8
```

```
accuracy of Res :  0.9621212121212122
accuracy of User :  0.9924242424242424
FAR =  0.011904761904761904 With Threshold of:  0.9
FRR =  0.06481481481481481 With Threshold of:  0.9
```

```
accuracy of Res :  0.9636363636363636
accuracy of User :  0.9924242424242424
FAR =  0.047619047619047616 With Threshold of:  0.2
FRR =  0.024691358024691357 With Threshold of:  0.2
```

```
accuracy of Res :  0.9681818181818181
accuracy of User :  0.9924242424242424
FAR =  0.03571428571428571 With Threshold of:  0.300
FRR =  0.027777777777777776 With Threshold of:  0.30
```

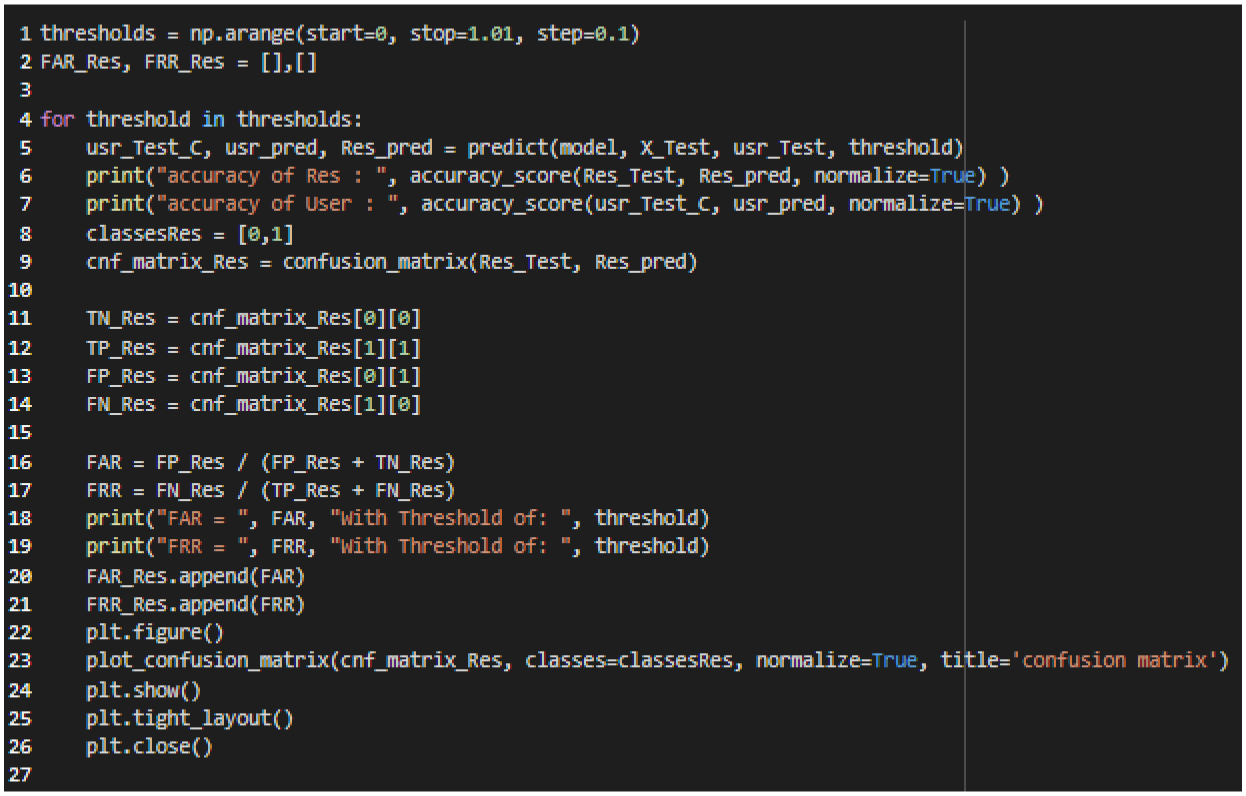
```
accuracy of Res :  0.9727272727272728
accuracy of User :  0.9924242424242424
FAR =  0.023809523809523808 With Threshold of:  0.60
FRR =  0.030864197530864196 With Threshold of:  0.60
```

```
accuracy of Res :  0.9727272727272728
accuracy of User :  0.9924242424242424
FAR =  0.017857142857142856 With Threshold of:  0.70
FRR =  0.037037037037037035 With Threshold of:  0.70
```

```
accuracy of Res :  0.509090909090909
accuracy of User :  0.9924242424242424
FAR =  0.0 With Threshold of:  1.0
FRR =  1.0 With Threshold of:  1.0
```

Figure 16. Results plot on test set, accuracies, FAR and FRR based on different thresholds

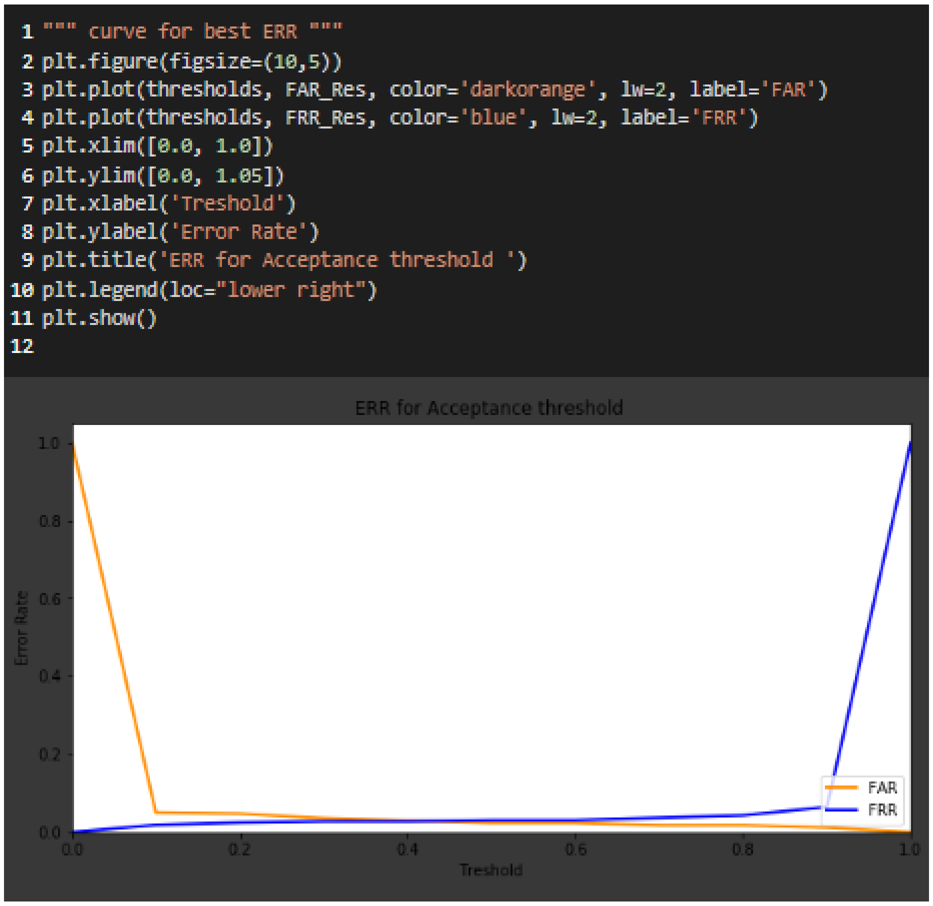
Code for plotting and calculating results.



```python
thresholds = np.arange(start=0, stop=1.01, step=0.1)
FAR_Res, FRR_Res = [],[]

for threshold in thresholds:
    usr_Test_C, usr_pred, Res_pred = predict(model, X_Test, usr_Test, threshold)
    print("accuracy of Res : ", accuracy_score(Res_Test, Res_pred, normalize=True) )
    print("accuracy of User : ", accuracy_score(usr_Test_C, usr_pred, normalize=True) )
    classesRes = [0,1]
    cnf_matrix_Res = confusion_matrix(Res_Test, Res_pred)

    TN_Res = cnf_matrix_Res[0][0]
    TP_Res = cnf_matrix_Res[1][1]
    FP_Res = cnf_matrix_Res[0][1]
    FN_Res = cnf_matrix_Res[1][0]

    FAR = FP_Res / (FP_Res + TN_Res)
    FRR = FN_Res / (TP_Res + FN_Res)
    print("FAR = ", FAR, "With Threshold of: ", threshold)
    print("FRR = ", FRR, "With Threshold of: ", threshold)
    FAR_Res.append(FAR)
    FRR_Res.append(FRR)
    plt.figure()
    plot_confusion_matrix(cnf_matrix_Res, classes=classesRes, normalize=True, title='confusion matrix')
    plt.show()
    plt.tight_layout()
    plt.close()

```

Figure 17. Code for plot confusion matrix and calculating FRR, FAR and accuracies

After this we tried to get EER for our system based on genuine or forgery signature detection, after plotting we see that best EER is happen when threshold is 0.4.



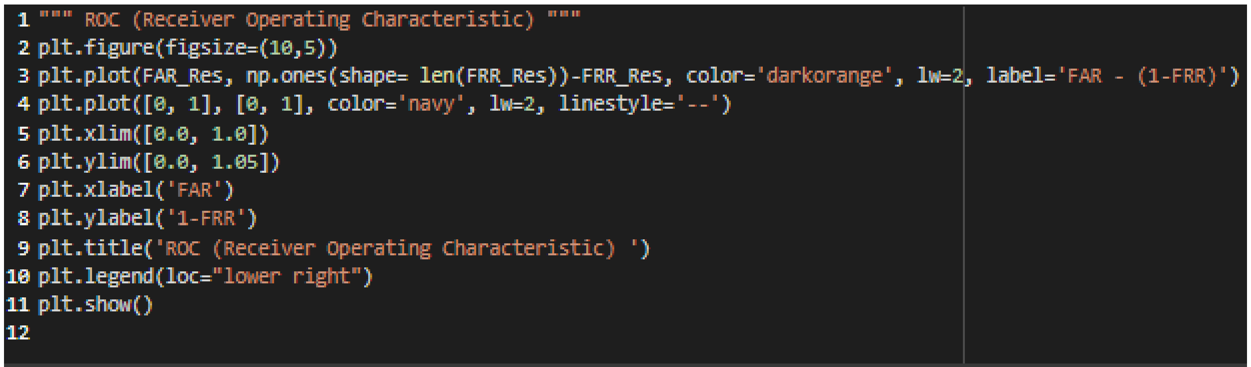
```python
""" curve for best ERR """
plt.figure(figsize=(10,5))
plt.plot(thresholds, FAR_Res, color='darkorange', lw=2, label='FAR')
plt.plot(thresholds, FRR_Res, color='blue', lw=2, label='FRR')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Treshold')
plt.ylabel('Error Rate')
plt.title('ERR for Acceptance threshold ')
plt.legend(loc="lower right")
plt.show()
```

Figure 18. Code and plot of EER

ROC and DET curves:

ROC curve: is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate against the false positive rate at various threshold settings. Wikipedia

In plot we can see that as TPR (1-FRR) increase the FAR increase too and if we want more user's signatures to be accepted in system we increase the risk of accepting more forgery signatures in our system



```
""" ROC (Receiver Operating Characteristic) """
plt.figure(figsize=(10,5))
plt.plot(FAR_Res, np.ones(shape= len(FRR_Res))-FRR_Res, color='darkorange', lw=2, label='FAR - (1-FRR)')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('FAR')
plt.ylabel('1-FRR')
plt.title('ROC (Receiver Operating Characteristic) ')
plt.legend(loc="lower right")
plt.show()

```

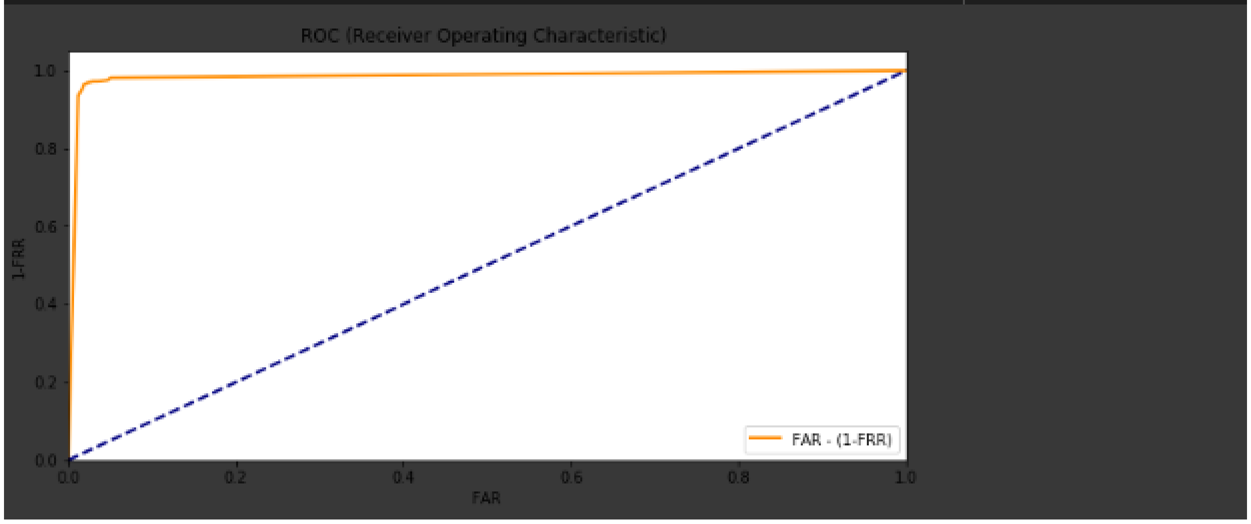
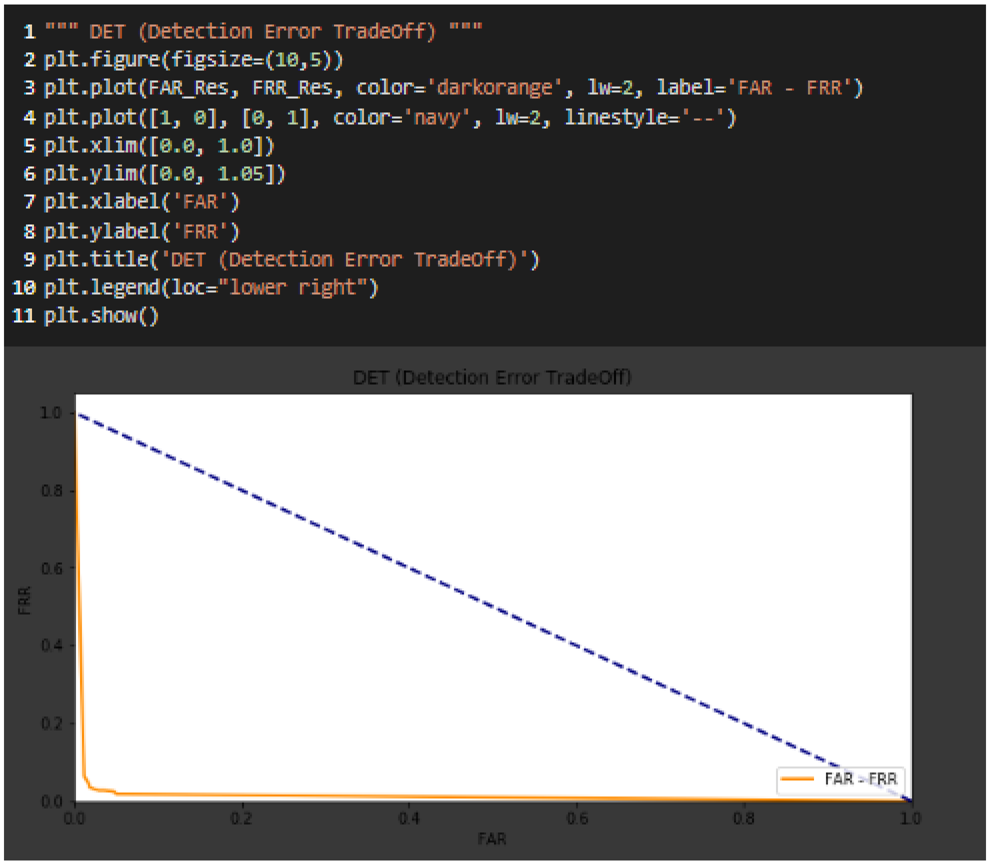


Figure 19. ROC curve code and plot

DET curve: is a graphical plot of error rates for binary classification systems, plotting the false rejection rate vs. false acceptance rate. In this image as the FRR decrease the FAR increase and we must have a balance tradeoff between them.

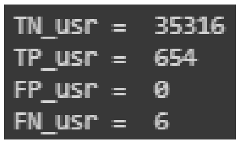


```
""" DET (Detection Error TradeOff) """
plt.figure(figsize=(10,5))
plt.plot(FAR_Res, FRR_Res, color='darkorange', lw=2, label='FAR - FRR')
plt.plot([1, 0], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('FAR')
plt.ylabel('FRR')
plt.title('DET (Detection Error TradeOff)')
plt.legend(loc="lower right")
plt.show()
```

Figure 20. DET curve code and plot

Confusion matrix and errors for user verification

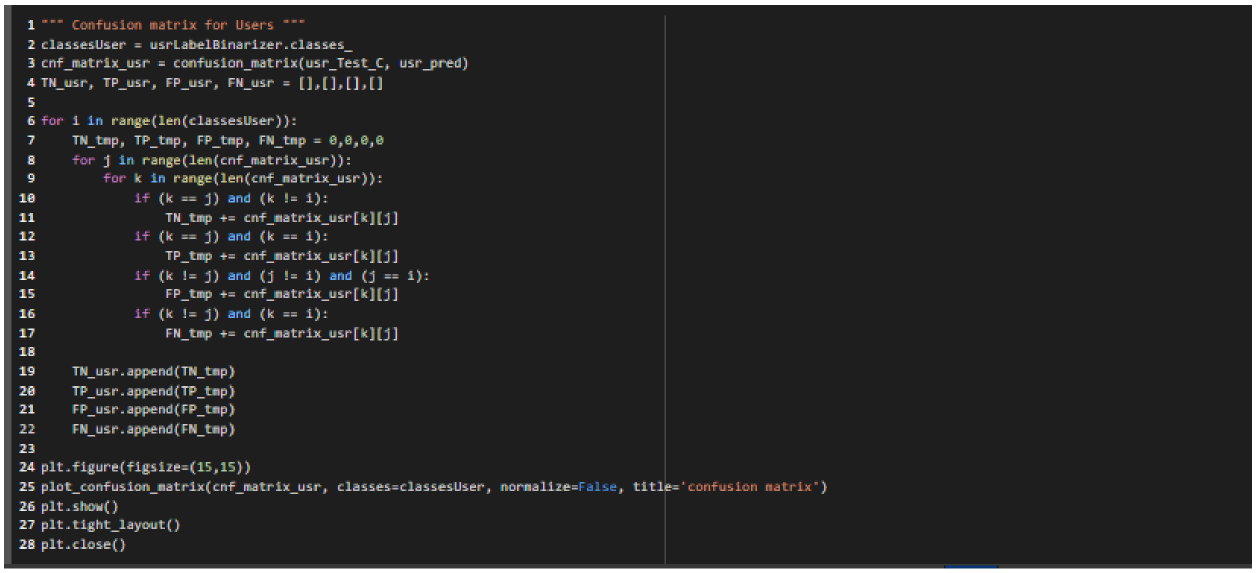
For user we write a code to extract all FA and FR's for all users and then some all of them to see the number of mistakes of our system out of 660 signatures for test and there was just 6 Genuine rejected users, we assume all other users images as imposter for specific user:



```
TN_usr =  35316
TP_usr =  654
FP_usr =  0
FN_usr =  6
```

Confusion matrix of users:

Below we put code and plot of confusion matrix for users and then we plot all mistakes of our system for **User** and **Res**.



```python
""" Confusion matrix for Users """
classesUser = usrLabelBinarizer.classes_
cnf_matrix_usr = confusion_matrix(usr_Test_C, usr_pred)
TN_usr, TP_usr, FP_usr, FN_usr = [],[],[],[]

for i in range(len(classesUser)):
    TN_tmp, TP_tmp, FP_tmp, FN_tmp = 0,0,0,0
    for j in range(len(cnf_matrix_usr)):
        for k in range(len(cnf_matrix_usr)):
            if (k == j) and (k != i):
                TN_tmp += cnf_matrix_usr[k][j]
            if (k == j) and (k == i):
                TP_tmp += cnf_matrix_usr[k][j]
            if (k != j) and (j != i) and (j == i):
                FP_tmp += cnf_matrix_usr[k][j]
            if (k != j) and (k == i):
                FN_tmp += cnf_matrix_usr[k][j]

    TN_usr.append(TN_tmp)
    TP_usr.append(TP_tmp)
    FP_usr.append(FP_tmp)
    FN_usr.append(FN_tmp)

plt.figure(figsize=(15,15))
plot_confusion_matrix(cnf_matrix_usr, classes=classesUser, normalize=False, title='confusion matrix')
plt.show()
plt.tight_layout()
plt.close()
```

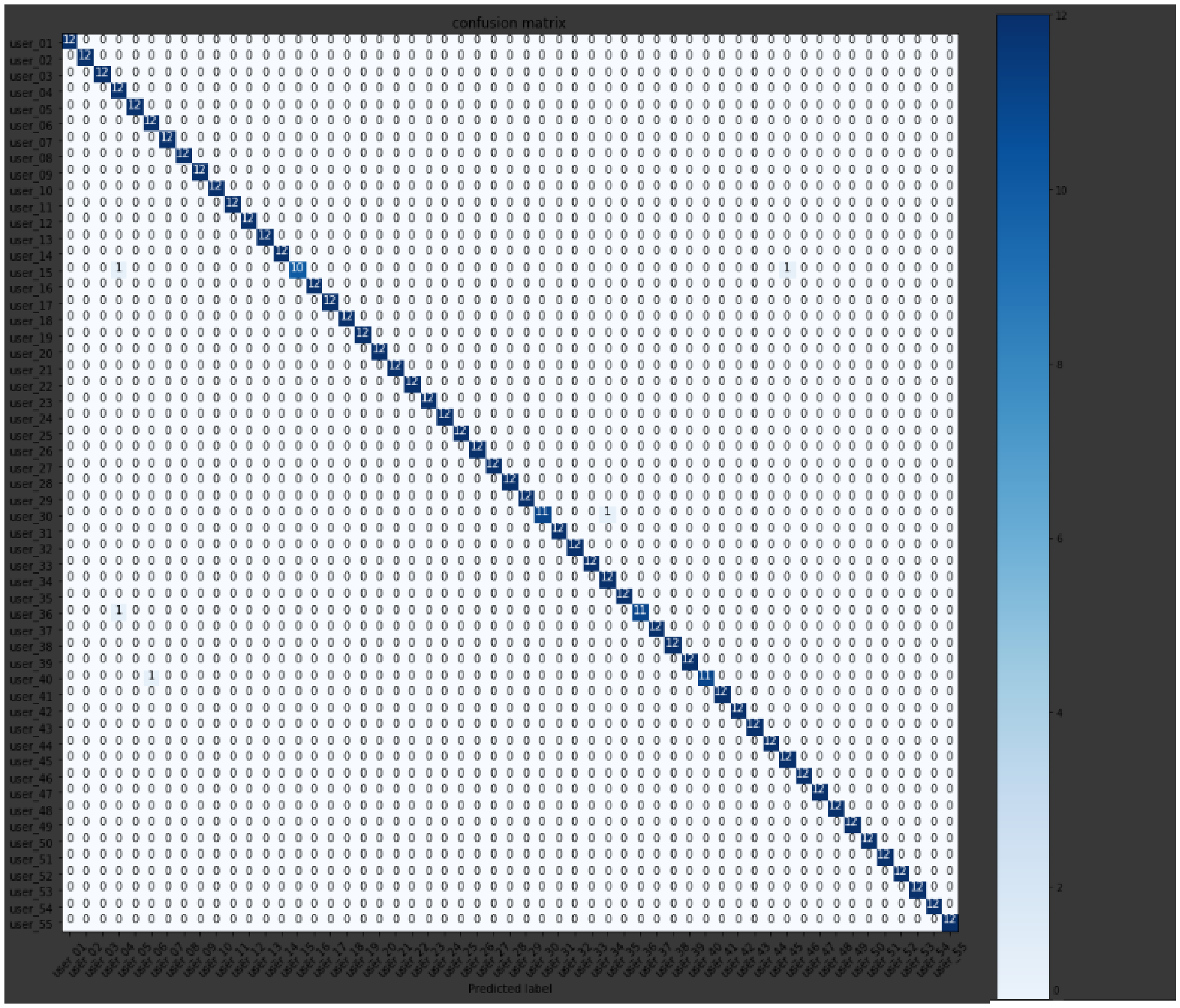


Figure 21. Code for confusion matrix and its plot for users

Our systems mistakes:

User

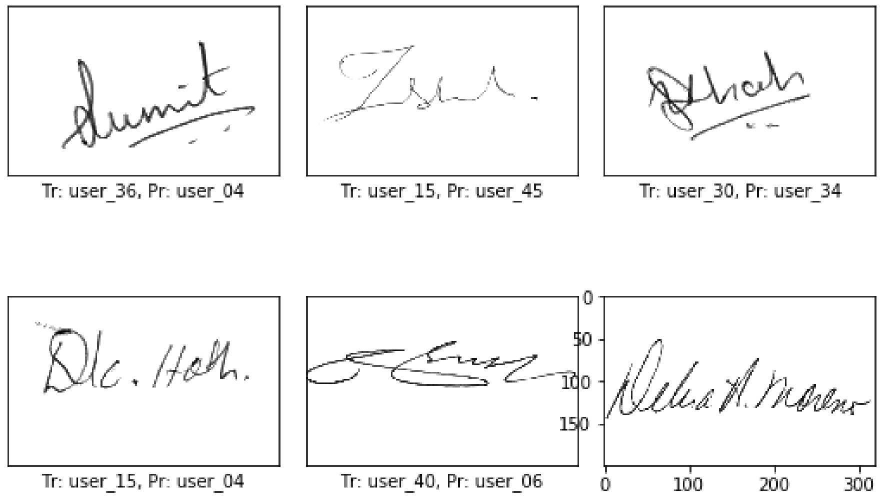


Figure 22. Our mistakes in user classification

Res

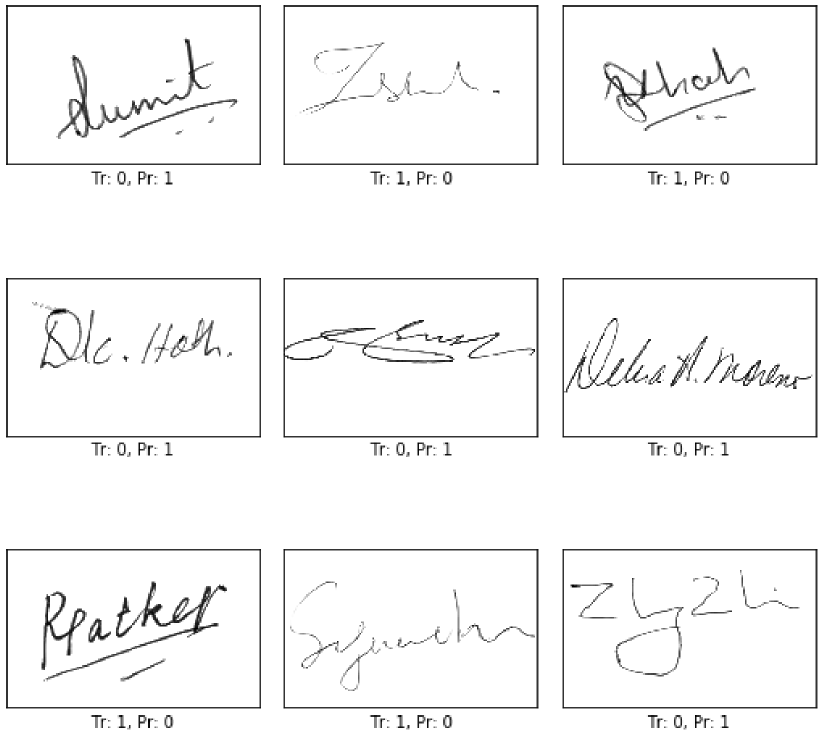


Figure 23. Our mistakes in Genuine or forgery detection.

We plot some of our predictions vs ground Truth of signatures too.

Users:

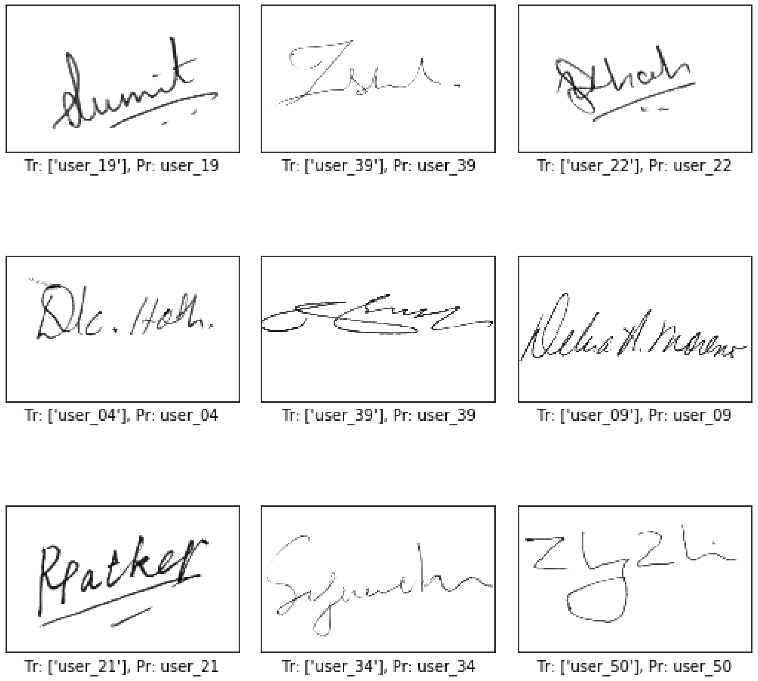

Tr: ['user_19'], Pr: user_19
Tr: ['user_39'], Pr: user_39
Tr: ['user_22'], Pr: user_22
Tr: ['user_04'], Pr: user_04
Tr: ['user_39'], Pr: user_39
Tr: ['user_09'], Pr: user_09
Tr: ['user_21'], Pr: user_21
Tr: ['user_34'], Pr: user_34
Tr: ['user_50'], Pr: user_50

Figure 24. Truth and Prediction for some users

Res:

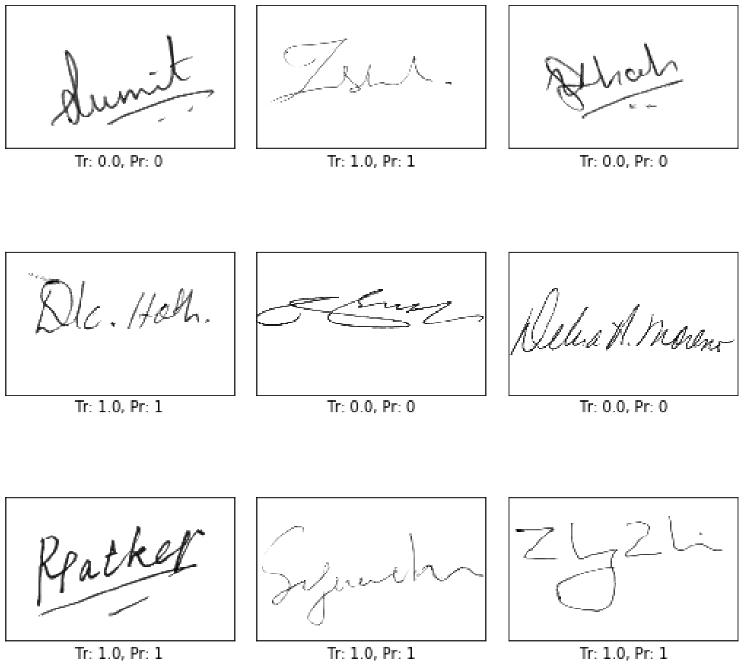


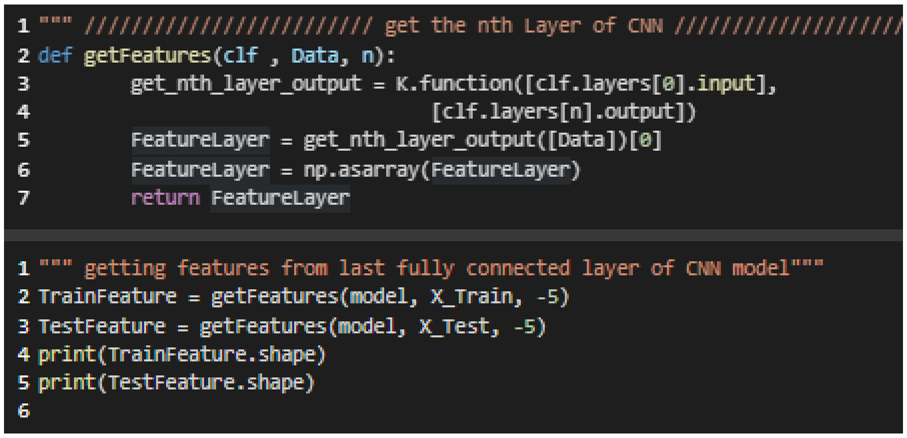
Figure 25. Genuine or forgery prediction and truth value for some signatures

Single signature verification and detection process

For this we feed our model with single signature of our test set in Raw format and we perform a full system architecture on that: we give user access if both user Detection and genuine or forgery verification was fully satisfied.

6- Some experimental work

After getting results of our CNN model we tried to do some works and test our features as an input to LSTM architecture, so we write a code to extract last feature layer from our keras model.

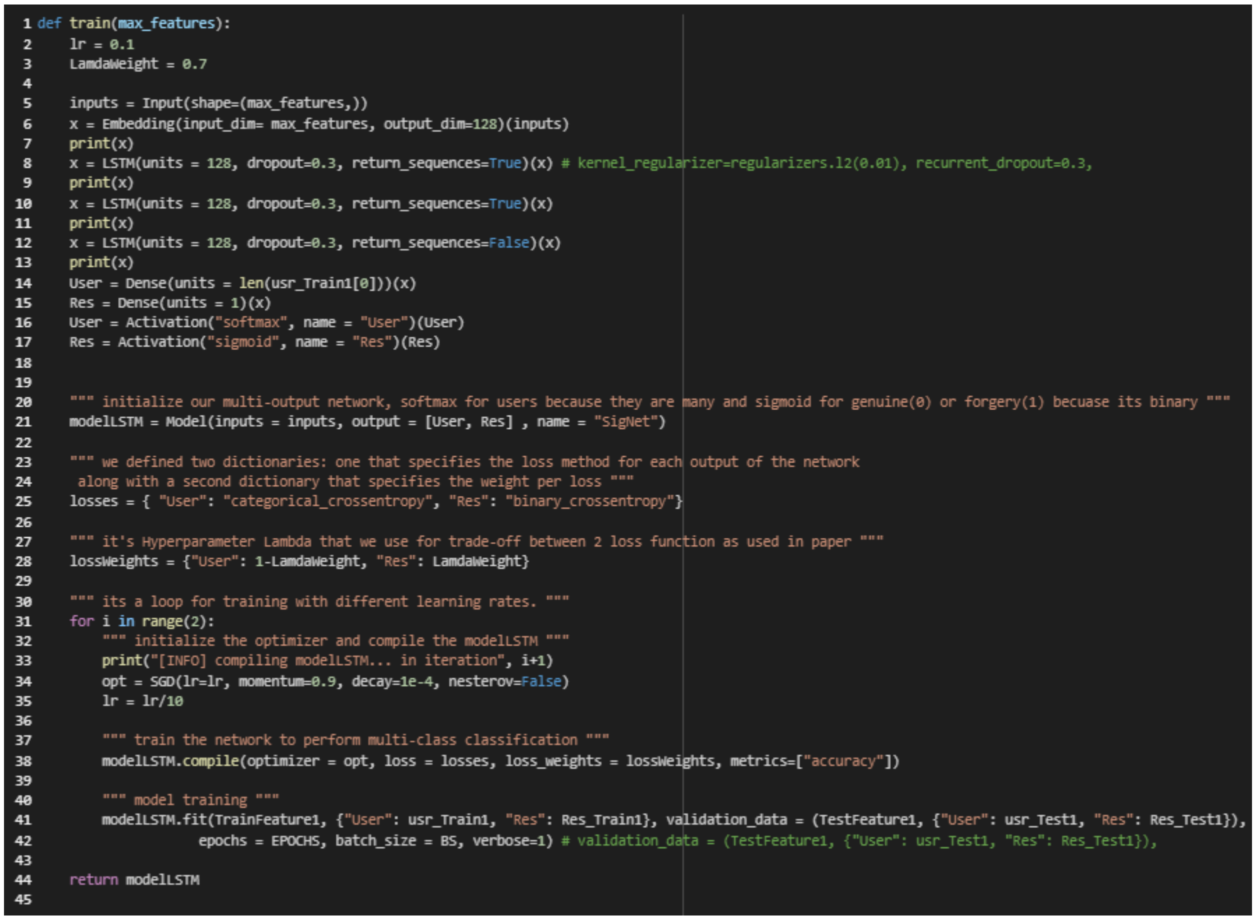


```
""" ///////////////////////// get the nth Layer of CNN ///////////////////
def getFeatures(clf , Data, n):
        get_nth_layer_output = K.function([clf.layers[0].input],
                                          [clf.layers[n].output])
        FeatureLayer = get_nth_layer_output([Data])[0]
        FeatureLayer = np.asarray(FeatureLayer)
        return FeatureLayer
```

```
""" getting features from last fully connected layer of CNN model"""
TrainFeature = getFeatures(model, X_Train, -5)
TestFeature = getFeatures(model, X_Test, -5)
print(TrainFeature.shape)
print(TestFeature.shape)

```

Figure 26. Code for getting feature layer for both train and test data.

As we explained we minimize the number of features to 256 so we feed that as input to our LSTM layers.



```python
def train(max_features):
    lr = 0.1
    LamdaWeight = 0.7

    inputs = Input(shape=(max_features,))
    x = Embedding(input_dim= max_features, output_dim=128)(inputs)
    print(x)
    x = LSTM(units = 128, dropout=0.3, return_sequences=True)(x) # kernel_regularizer=regularizers.l2(0.01), recurrent_dropout=0.3,
    print(x)
    x = LSTM(units = 128, dropout=0.3, return_sequences=True)(x)
    print(x)
    x = LSTM(units = 128, dropout=0.3, return_sequences=False)(x)
    print(x)
    User = Dense(units = len(usr_Train1[0]))(x)
    Res = Dense(units = 1)(x)
    User = Activation("softmax", name = "User")(User)
    Res = Activation("sigmoid", name = "Res")(Res)


    """ initialize our multi-output network, softmax for users because they are many and sigmoid for genuine(0) or forgery(1) becuase its binary """
    modelLSTM = Model(inputs = inputs, output = [User, Res] , name = "SigNet")

    """ we defined two dictionaries: one that specifies the loss method for each output of the network
     along with a second dictionary that specifies the weight per loss """
    losses = { "User": "categorical_crossentropy", "Res": "binary_crossentropy"}

    """ it's Hyperparameter Lambda that we use for trade-off between 2 loss function as used in paper """
    lossWeights = {"User": 1-LamdaWeight, "Res": LamdaWeight}

    """ its a loop for training with different learning rates. """
    for i in range(2):
        """ initialize the optimizer and compile the modelLSTM """
        print("[INFO] compiling modelLSTM... in iteration", i+1)
        opt = SGD(lr=lr, momentum=0.9, decay=1e-4, nesterov=False)
        lr = lr/10

        """ train the network to perform multi-class classification """
        modelLSTM.compile(optimizer = opt, loss = losses, loss_weights = lossWeights, metrics=["accuracy"])

        """ model training """
        modelLSTM.fit(TrainFeature1, {"User": usr_Train1, "Res": Res_Train1}, validation_data = (TestFeature1, {"User": usr_Test1, "Res": Res_Test1}),
                      epochs = EPOCHS, batch_size = BS, verbose=1) # validation_data = (TestFeature1, {"User": usr_Test1, "Res": Res_Test1}),

    return modelLSTM

```

Figure 27. LSTM model with Keras

7- References

1- Learning features for offline handwritten signature verification using deep convolutional neural networks https://arxiv.org/pdf/1705.05787.pdf

2- Pal S., Pal U., Blumenstein M. (2014) Signature-Based Biometric Authentication. In: Muda A., Choo YH., Abraham A., N. Srihari S. (eds) Computational Intelligence in Digital Forensics: Forensic Investigation and Applications. Studies in Computational Intelligence, vol 555. Springer, Cham

3- Hameed, Shaimaa & Ridha, Arwas. (2019). DIGITAL SIGNATURE METHODS BASED BIOMETRICS. International Journal of Advanced Research. 7. 2320-5407.

4- Yang, Wencheng & Wang, Song & Hu, Jiankun & Guanglou, Zheng & Chaudhry, Junaid & Adi, Erwin & Valli, Craig. (2018). Securing Mobile Healthcare Data: A Smart Card based Cancelable Finger-vein Bio-Cryptosystem. IEEE Access. PP. 1-1. 10.1109/ACCESS.2018.2844182.

5- Code for plotting confusion matrix. https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html

6- Keras Documentation https://keras.io

7- Scikit-learn documentation https://scikit-learn.org

8- Numpy Documentation http://www.numpy.org/

9- Matplotlib Documentation https://matplotlib.org/

10- Convolutional Neural Network Explanation https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
11- datasets from https://cedar.buffalo.edu/NIJ/data/