Sapienza University Of Rome

Feature Extraction Using Deep Convolutional Neural Networks For Offline Signature Verification

Biometric Systems Project

Manoochehr Joodi Bigdello - 1860273

Ziba Khani - 1799920

Prof. Maria De Marsico

June 2020

Table of Contents

Table of Figures

Abstract

These days, reliable authentication and authorization of individuals is challenging in the presence of skilled forgeries, where a forger has access to a person's signature and deliberately attempts to imitate it. the dynamic information of the signature in offline signature verification systems is lost, and it is difficult to design good feature extractors that can distinguish genuine signatures and skilled forgeries. There are some works in the literature about tackling the problem with Convolutional Neural Networks (CNN) but the extracted features and learnable parameters is a lot and learning process is slow, so we propose a modified multi-task CNN architecture to tackle the problem with a high-speed learning and fewer features while making better accuracy. We proposed a Two-phase mechanism involves identification of the person and then its authenticity by performing multi-task learning. Then we used learned feature space as input for SVM classifier in WD mode and test it on a disjoint set of users and achieved around 0 of EER on skilled forgeries in CEDAR.

Dataset

CEDAR signature database [11] contains signatures of 55 signers belonging to various cultural and professional backgrounds. Each of these signers signed 24 genuine signatures 20 minutes apart. Each of the forgers tried to emulate the signatures of 3 persons, 8 times each, to produce 24 forged signatures for each of the genuine signers. Hence the dataset comprise 55 × 24 = 1,320 genuine signatures as well as 1,320 forged signatures. The signature images in this dataset are available in gray scale mode.

Introduction to Authentication 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. 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 features must be perfect and unique, they can be evaluated easily, and they are cost-efficient and have great user approval. [3]

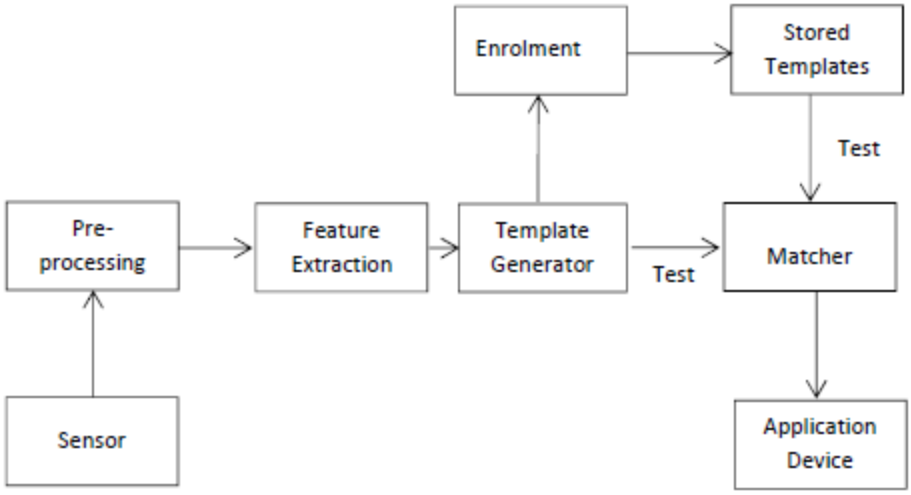

Figure 1. A Generic Authentications System[3]

Signature as Authentication System Input

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.

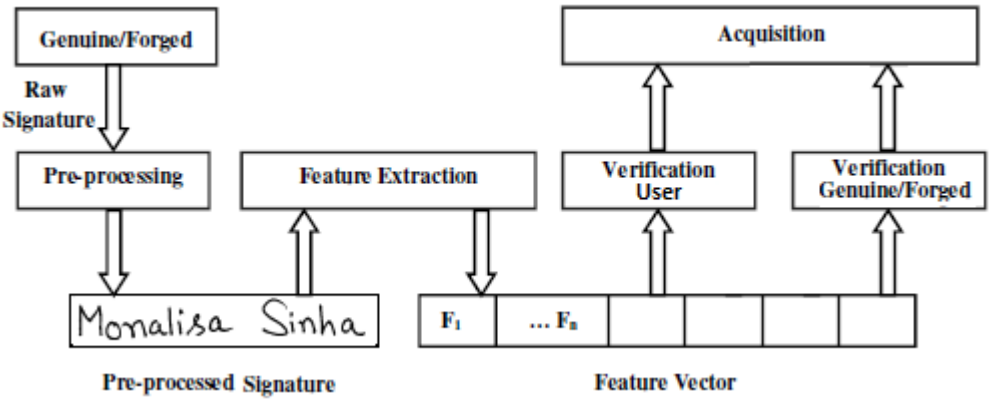

Figure 2. Block Diagram of our Automatic Signature Verification System

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.

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 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 signatures: 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. [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. [2]

Types of Forgeries

There are usually three different types of forgeries to take into account, 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]

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).

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 ϕ(X).

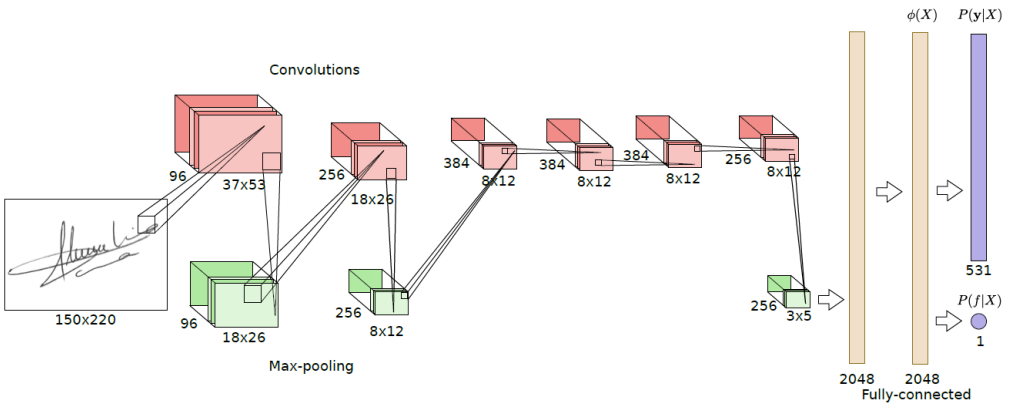

Figure 3. Illustration of the CNN architecture used in original paper.

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 3 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:

$$L_c = -\sum_j y_{ij} \log P(y_j|X_i)$$
$$L_f = -f_i \log(P(f|X_i)) - (1 - f_i) \log(1 - P(f|X_i)) \quad (2)$$

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:

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

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:

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

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)**

Figure 4. Results of original Paper on CEDAR

Our Implementation

We conducted experiments using all Genuine and Forgery signatures of each user in WI set to train our CNN model on all datasets. To evaluate the quality of learned features we used a disjoint set of users to train CNN and then transfer learning from CNN model to train SVM classifiers in WD mode. for each dataset containing M users in WD-Set having G genuine signature and F skilled forgery signature, we considered G/2 and F/2 signature for training and G/2 and F/2 for evaluating. We trained one classifier for each user by oversampling of training data by assuming we have enough forgery and genuine signature for each user.

To address both the issue of obtaining a good feature representation for signatures, as well as improving classification performance, we used Convolutional Neural Networks (CNN) for learning the representations directly from the signature images. We inspired from proposed formulation [1] as loss functions, which incorporates knowledge of skilled forgeries, using a multi-task learning strategy. The hypothesis is that visual cues present in the signature images can be learned by the model. These visual cues (features) are discriminative between genuine signatures and forgeries in general (i.e. not specific to individuals).

We perform these steps on datasets: (i) pre-processing, (ii) Extracting Features, (iii) Multi-task learning, shown in Fig. 1.

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, (150,220))).
    - Put threshold to remove some noises, we put 0 pixels less than 30 as they are noises (we get 10 by trying some numbers like 10, 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.

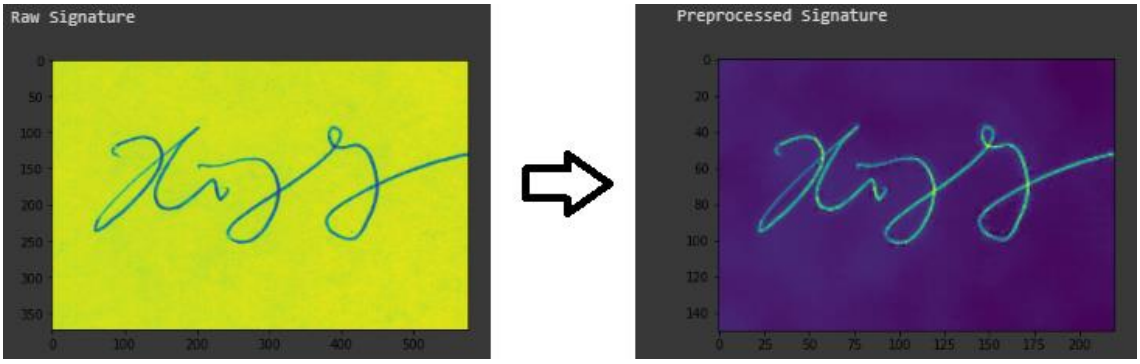

Figure 5. Sample image after preprocessing

After all these steps we have our dataset in the form of (X, y, f) where X is our input image for train, y is user, and f is a binary (0 for Genuine and 1 for forgery signatures).

```python
def get_UserName_Res(self, 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 """
    print("[INFO] Getting username and Results ... from {}".format(self.dataset_name))
    usr_all = []
    Res_all = []
    for file in filenames:
        _,_,_,_,_,_,_,possibleForg, imgY, imgRes = file.split("/")
        x = imgRes.split("-")
        usr_all.append(imgY)
        if possibleForg == "forgeries":
            Res = 1
        if possibleForg == "original":
            Res = 0
        Res_all.append(Res)

    return np.asarray(usr_all), np.asarray(Res_all)

def get_Images(self, filenames):
    print("[INFO] Getting Images ... from {}".format(self.dataset_name))
    """ reading all images as a grayscale images in 2D array format """
    X_all = []
    for filename in filenames:
        img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
        img = 255 - img # invert the pixels signature pixels are around 200 in pixels
        img = cv2.resize(img,(IMAGE_WIDTH, IMAGE_HEIGHT))
        """ 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, 10, 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,3)

    return np.asarray(X_all)
```

Figure 6. Data preparing code in python

After preprocessing we split our dataset in WI and WD sets.

Dataset	Users	WI set	WD set	Genuine	Forgery
CEDAR	55	40	15	24	24

```
def split_data(self, X_All, c_usr_All, Res_All, user_classes, numUsrForFL):
    """ this is function to split our data into 2 parts, we are going to use all signatures of some users
    for part 1 and some users for part 2. we are creating two separate user sets to have WD form of learning
    part 1 is going to be used for train our CNN model for extracting features.
    part 2 will be used to test those features in WD form. """

    user_classes_WD = user_classes[numUsrForFL:]
    user_classes_FL = user_classes[0: numUsrForFL]

    X_FL, c_usr_FL, Res_FL = [], [], []
    X_WD, c_usr_WD, Res_WD = [], [], []

    for i, c_usr in enumerate(c_usr_All):
        if c_usr in user_classes_FL:
            X_FL.append(X_All[i])
            c_usr_FL.append(c_usr_All[i])
            Res_FL.append(Res_All[i])
        if c_usr in user_classes_WD:
            X_WD.append(X_All[i])
            c_usr_WD.append(c_usr_All[i])
            Res_WD.append(Res_All[i])

    X_FL = np.asarray(X_FL)
    c_usr_FL = np.asarray(c_usr_FL)
    Res_FL = np.asarray(Res_FL)
    X_WD = np.asarray(X_WD)
    c_usr_WD = np.asarray(c_usr_WD)
    Res_WD = np.asarray(Res_WD)

    return X_FL, c_usr_FL, Res_FL, X_WD, c_usr_WD, Res_WD, user_classes_WD, user_classes_FL
```

Figure 7. Code for splitting datasets to WD and WI sets

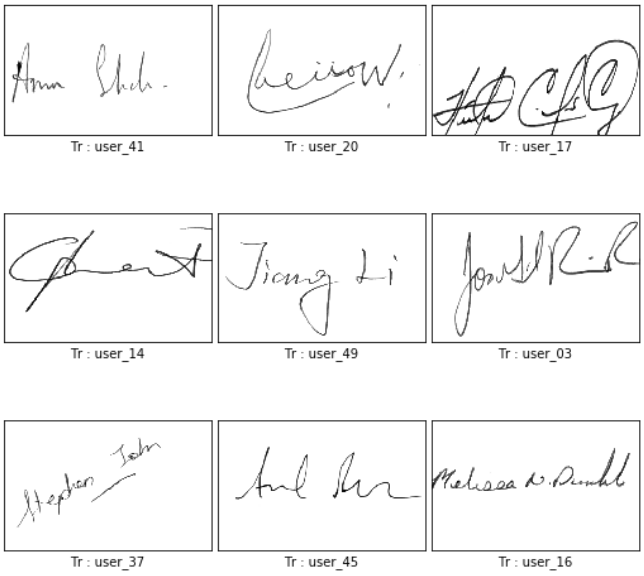

Figure 8. Sample images after preprocessing from CEDAR

Modified CNN Architecture

For feature extraction, we used multi-task CNN. We used SigNet-F [1] as our base model because they achieved significant improvements in results in both WD and WI format. But the problem of SigNet-F is the fact that it learns a 2048 sized feature vector, and 2 FC layer of size 2048 with a lot of layers without stride makes their model slow to learn for large benchmarks (38 million learnable parameters based on published model). We designed our CNN model shown in Figure 7 with 3 difference in comparison with [1], (i) we added one Max Pooling layer between CL4 and CL5 layers with stride of 2 to compress the input for FC1 layer as much as possible,

because it has a great impact on number of learnable parameters, (ii) we changed FC1 size to 1024, (iii) to have fewer features to learn we changed last FC layer to 1024 (we performed experiments to find best FC layers), by doing this we achieved less feature set, and also fasten the learning process (8 million learnable parameters).

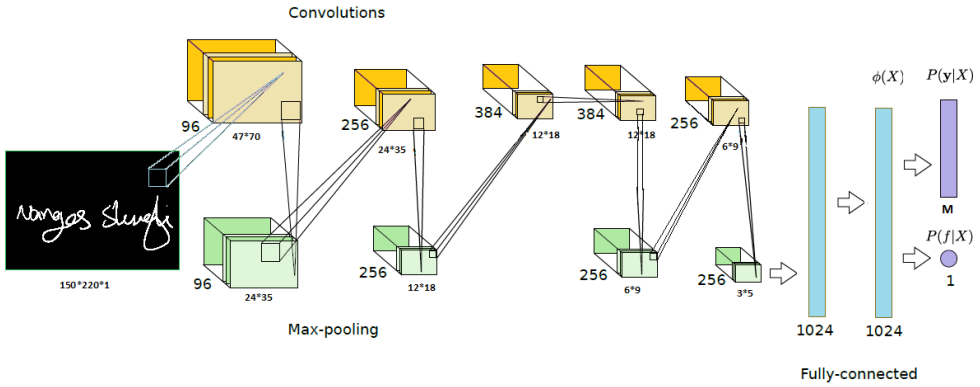

Figure 9. Our proposed CNN architecture

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

Table 1 . Summary of CNN layers

Layer	Size	Parameters
Input	150*220*1	
Conv2D (CL1)	11*11*96	Stride=3, pad=0
MaxPooling	3*3*96	Sride=2
Conv2D (CL2)	5*5*256	Sride=1, pad=same
MaxPooling	3*3*256	Sride=2
Conv2D (CL3)	3*3*384	Sride=1, pad=same
Conv2D (CL4)	3*3*384	Sride=1, pad=same
MaxPooling	3*3*384	Sride=2
Conv2D (CL5)	3*3*256	Sride=1, pad=same
MaxPooling	3*3*256	Sride=2
Fully Connected (FC1)	1024	-
Fully Connected (FC2)	1024	-
Softmax	M	-
Sigmoid	1	-

Multi-Output CNN Model Implementation

We trained our CNN model as a multi-task learner in WI mode, and we used disjoint set of users for WD verification part as shown in Table V. we used all Genuine and Forgery signatures of each user in WI-Set to train by considering our dataset in a form of (X, y, f) where X is our input image for train, y is user, and f is a binary (0 for Genuine and 1 for forgery signatures). We exploited loss functions defined in [1] as they achieved good performance. We implemented our CNN model in Keras tensorFlow.

```
""" ////////////////////////////// Build CNN ////////////////////////////////////////////////// """
def build_OurCNN_model(self, inputs):
    # val_User_acc: 1.0000 - val_Res_acc: 0.9874
    x = Conv2D(96, (11,11), strides=3, 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 = MaxPooling2D((3,3), strides=2, padding='same')(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 = self.fc1_Size , activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)
    x = Dense(units = self.fc2_Size, activation='relu', kernel_regularizer=regularizers.l2(0.01))(x)
    x = BatchNormalization()(x)

    user_Model = Dense(units = self.NumOfUser,  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 10. Python code for our CNN model

Training our model

For training first, we initialized our CNN model learnable parameters with random weights and then train them with stochastic gradient descent for 120 epochs to minimize the defined objective function. We use mini-batches of size 64 (which is required in order to use batch normalization and also speeds up training), as hyperparameters we used momentum nestrov, momentum rate equal to 0.9, and the decay of 1e-4. We started with an initial learning rate (LR) equal to 1e-5 and divided by 10 every 30 epochs. All these values are shown in Table 2. Our entire framework is implemented using Keras library with the TensorFlow as backend. Plots for total loss (objective function), verification (Lf) and identification (Lc) tasks for both training and validation sets for CEDAR dataset in all epochs shown in Figure 12, and Figure 13 is the accuracy plot of our model in those sets. Both plots extracted from Keras model callback. As shown in figures there is fluctuation in first 40 epochs results and then model converge to good results training model without first epochs decrease the performance a lot and we understand that model learns proper wights at first that helps it to converge in further learning rates.

Table 2 . List Of Hyperparameters

Parameter	Value
Nestrov	True
Momentum rate	0.9
Weight Decay	1e-4
Batch-size	64
Initial Learning rate (LR)	1e-5
Learning Rate Schedule	LR ← LR × 0.1
Epochs	120

```python
def train(self, X_Train, usr_Train_LB, Res_Train, X_Test, usr_Test_LB, Res_Test):
    LamdaWeight = 0.7
    lr = 1e-4
    """ 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"}

    """ initialize our multi-output network, softmax for users because they are many and sigmoid
    for genuine(0) or forgery(1) becuase its binary """
    inputShape = (self.IMAGE_HEIGHT, self.IMAGE_WIDTH, 1)
    inputs = Input(shape = inputShape)
    """ it's Hyperparameter Lambda that we use for trade-off between 2 loss function as used in paper """
    lossWeights = {"User": (1-LamdaWeight) * (1-Res_Train), "Res": LamdaWeight}

    if self.originalPaper:
        print("[INFO] .... original paper model training .... ")
        User, Res = self.build_Paper_model(inputs)
    else:
        print("[INFO] .... our CNN model training .... ")
        User, Res = self.build_OurCNN_model(inputs)

    self.model = Model(inputs = inputs, outputs = [User, Res], name = "SigNet")

    # This function keeps the learning rate at 0.001 for the first ten epochs and decreases it exponentially after that.
    def scheduler(epoch, lr):
        if epoch % self.EPOCHS == 0:
            print("[INFO] lr is  ... ", lr*1e-1)
            return lr*1e-1
        else:
            return lr

    callback = tf.keras.callbacks.LearningRateScheduler(scheduler)

    """ initialize the optimizer and compile the model """
    opt = SGD(lr=lr, momentum=0.9, decay=1e-4, nesterov=True)

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

    """ model training """
    self.history = self.model.fit(X_Train, {"User": usr_Train_LB, "Res": Res_Train}, validation_data = (X_Test, {"User": usr_Test_LB, "Res": Res_Test}),
                epochs = 120, callbacks=[callback], batch_size = self.batch_size, verbose = 2)          # verbose=0 will show you nothing (silent), verbo
```

Figure 11. Python code for our model training

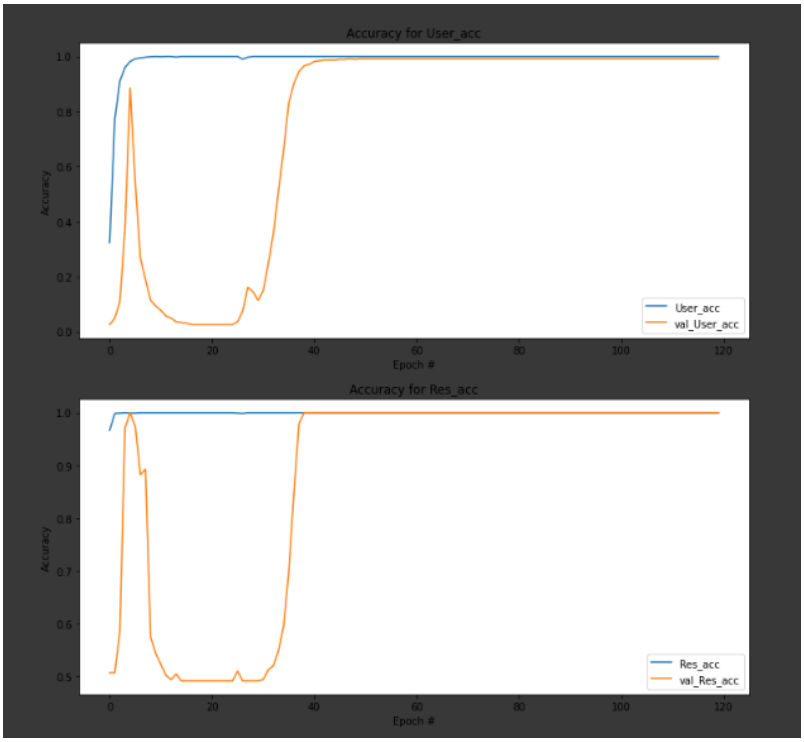

Figure 12. Accuracy in all epochs for User Accuracy and Res Accuracy for both train and validation set

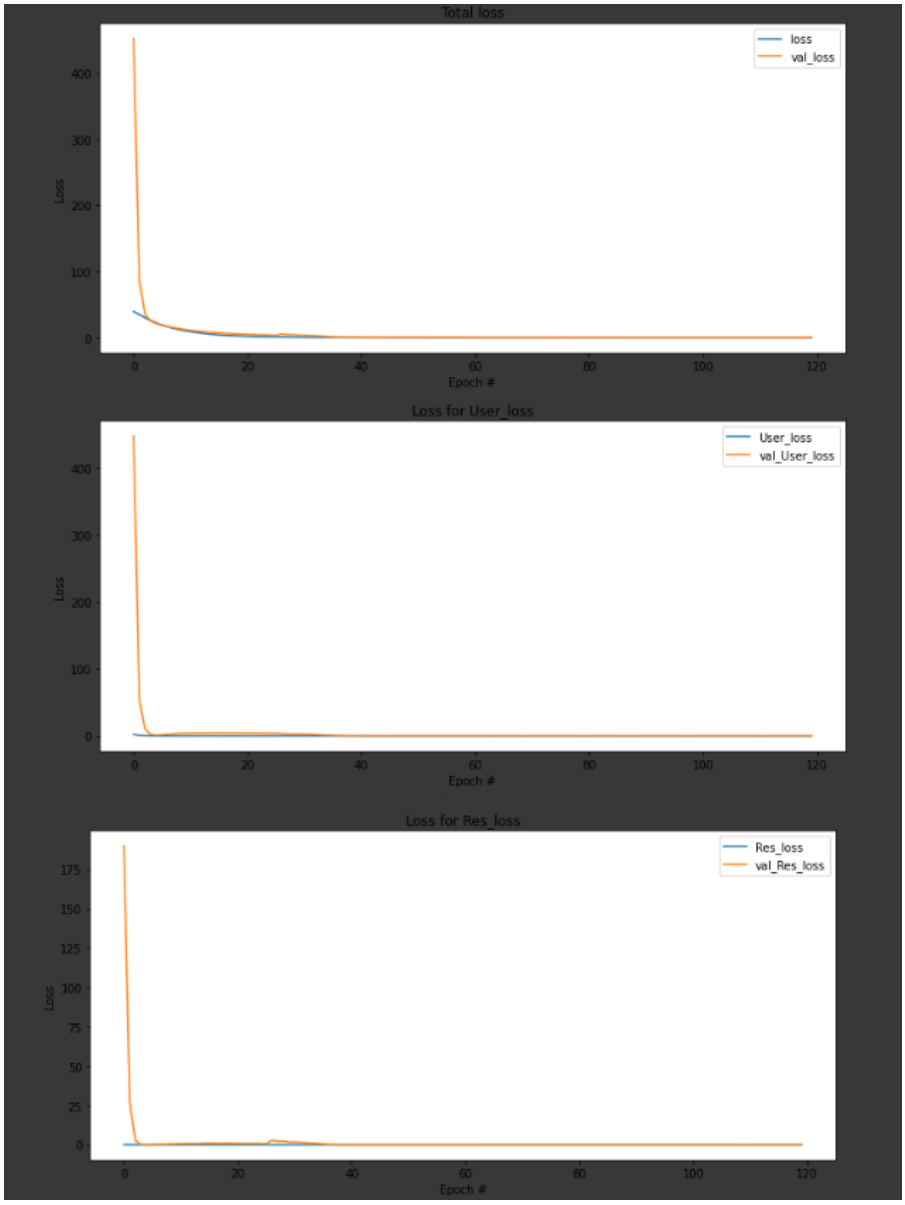

Figure 13. Loss in all epochs for total loss, User loss and Res loss for both train and validation set

Test and Evaluations of CNN model

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.

Table3 . mean Results with 5-FOLD using stratifiedkfold

Dataset	Accuracy Gen/Forg	Accuracy User	Run time (min)
CEDAR	100	99.21	13

```
199
200     # function to predict both output values
201     def predict(self, usr_pred_proba, Res_pred_proba, usrclasses_, threshold):
202         """ User prediction section based on argmax """
203         usrClasses = usrclasses_
204         usr_pred = []
205         for i in range(len(usr_pred_proba)):
206             usr_pred.append(usrClasses[np.argmax(usr_pred_proba[i], axis = 0)])
207 
208         """ Genuine or Forgery Prediction as Result """
209         Res_pred = []
210         for i in range(len(Res_pred_proba)):
211             Res_pred.append(1 if Res_pred_proba[i] >= threshold else 0)
212 
213         return usr_pred, Res_pred
214 
215     # function to return probability of predictions for users as vector of size number of users.
216     def predict_usr_proba(self, x_data):
217         pred = self.model.predict(x_data)
218         return pred[0][:]
219 
220     # function to return probability of predictions for res as one float number between [0,1]
221     def predict_Res_proba(self, x_data):
222         pred = self.model.predict(x_data)
223         return pred[1][:]
224 
```

Figure 14. 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. The model was rebust enough and in threshold range of 0.1-0.9 we had promising results, and the analysis of model in these threshold rates are shown in Figure 15.

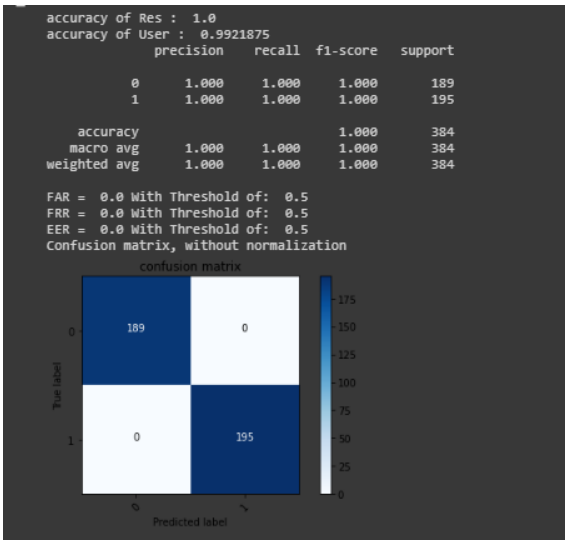

Figure 15. Results plot on test set, accuracies, FAR and FRR based on thresholds

Code for plotting and calculating results

```python
def Res_Classification_Evaluation(trainCNN, X_Test, c_usr_Test, Res_Test, thresholds, usrclasses_):
    EER_Res, FAR_Res, FRR_Res = [],[],[]
    Res_pred_proba = trainCNN.predict_Res_proba(X_Test)
    Roc_Auc_Score_Res = roc_auc_score(Res_Test, Res_pred_proba)
    print("Roc_Auc_Score_Res = ", Roc_Auc_Score_Res)

    usr_pred_proba = trainCNN.predict_usr_proba(X_Test)
    Roc_Auc_Score_usr = roc_auc_score(c_usr_Test, usr_pred_proba, multi_class = "ovo")
    print("Roc_Auc_Score_usr = ", Roc_Auc_Score_usr)

    print("\n ================================================\n")
    for threshold in thresholds:
        usr_pred, Res_pred = trainCNN.predict(usr_pred_proba, Res_pred_proba, usrclasses_, threshold)
        print("accuracy of Res : ", accuracy_score(Res_Test, Res_pred, normalize=True) )
        print("accuracy of User : ", accuracy_score(c_usr_Test, usr_pred, normalize=True) )

        print(metrics.classification_report(Res_Test, Res_pred, digits=3))
        classesRes = [0,1]
        cnf_matrix_Res = confusion_matrix(Res_Test, Res_pred, labels= classesRes)

         # https://www.geeksforgeeks.org/confusion-matrix-machine-learning/

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

        FAR = FP_Res / (FP_Res + TN_Res)
        FRR = FN_Res / (TP_Res + FN_Res)
        EER = (FRR + FAR)/2
        # EER or AER is AER = (FRR + FARrandom + FARsimple + FARskilled)/4. in our case just = (FRR + FARskilled)/2

        print("FAR = ", FAR, "With Threshold of: ", threshold)
        print("FRR = ", FRR, "With Threshold of: ", threshold)
        print("EER = ", EER, "With Threshold of: ", threshold)
        FAR_Res.append(FAR)
        FRR_Res.append(FRR)
        EER_Res.append(EER)
        plt.figure()
        plot_confusion_matrix(cnf_matrix_Res, classes=classesRes, normalize=False, title='confusion matrix')
        plt.show()
        plt.tight_layout()
        plt.close()
        print("============================================================ \n")
```

Figure 16. 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.

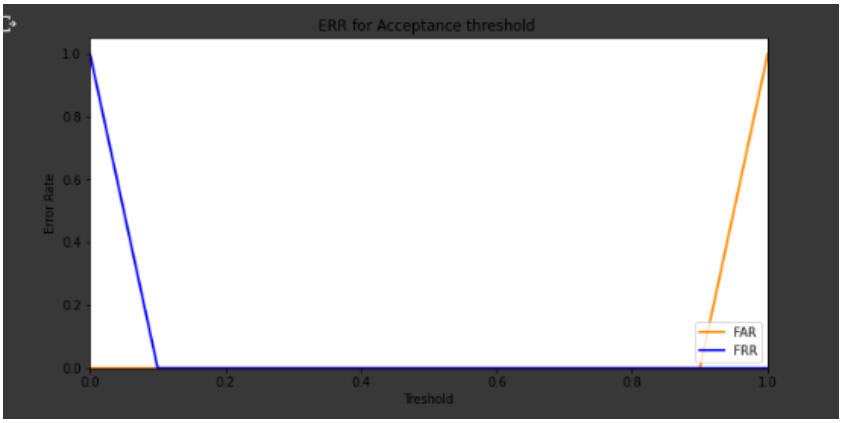

Figure 17. plot of EER

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

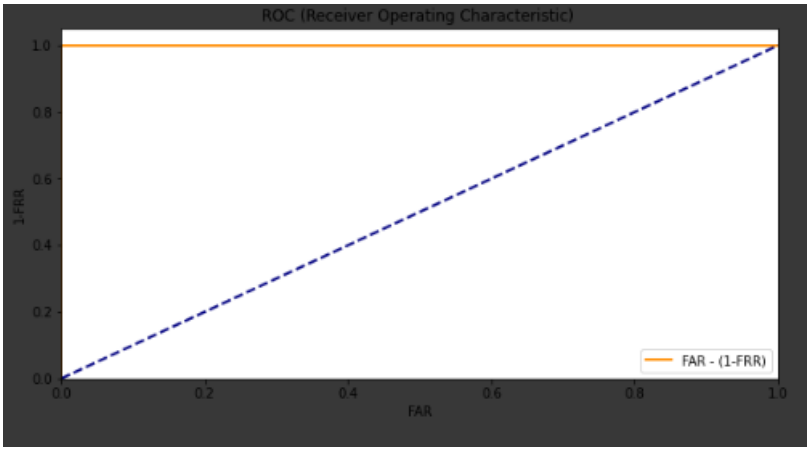

Figure 18. ROC curve code and plot

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 381 (40 user eah user has 9-10 Signatures) signatures for test and there was just 3 Genuine rejected users and 0 Imposter Acceptence, we assume all other users images as imposter for specific user.

```
Roc_Auc_Score_usr =  0.9999857549857549


TN_usr =  14859
TP_usr =  381
FP_usr =  0
FN_usr =  3
```

Figure 19. Results for User output

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**.

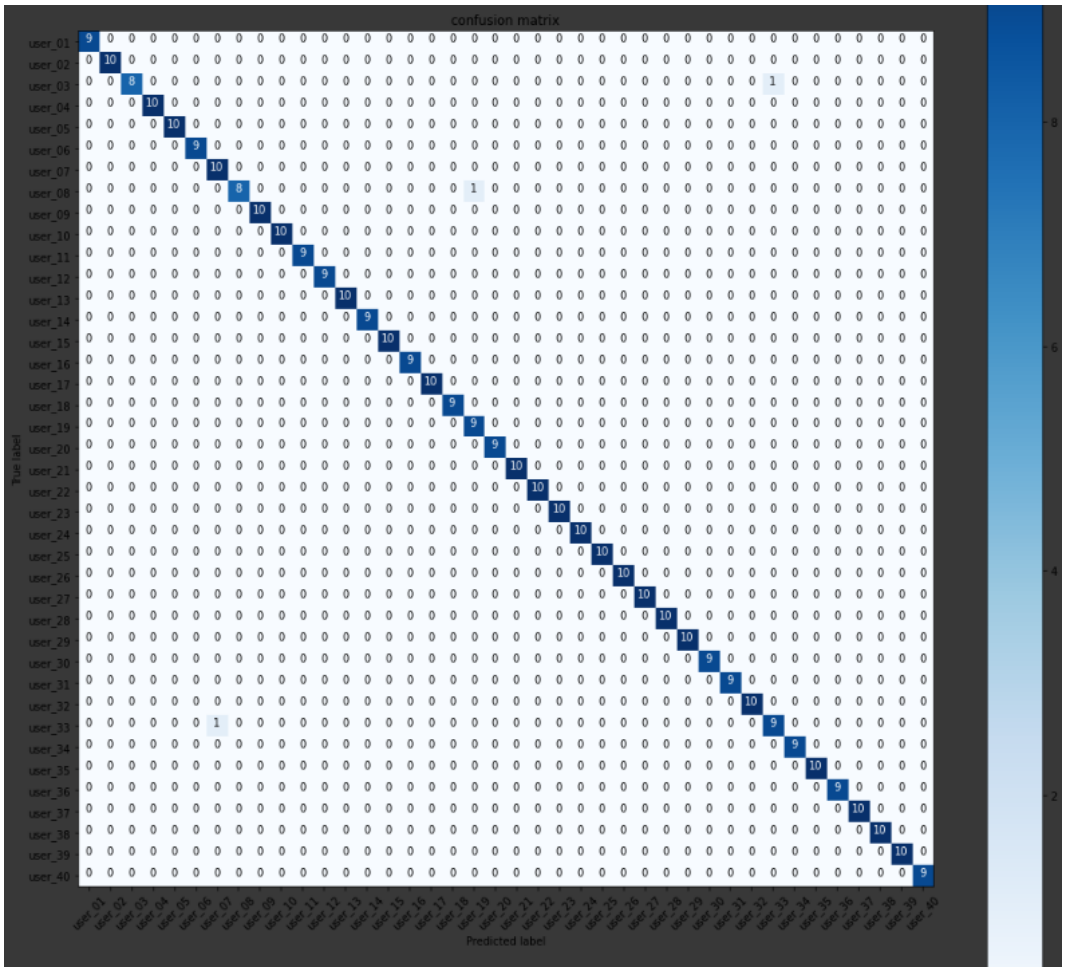

Figure 20. confusion matrix plot for users

```
def user_Identification_Evaluaion(trainCNN, X_Test, c_usr_Test, classesUser):
    """ Confusion matrix for Users """
    usr_pred_proba = trainCNN.predict_usr_proba(X_Test)
    Res_pred_proba = trainCNN.predict_Res_proba(X_Test)
    Roc_Auc_Score_usr = roc_auc_score(c_usr_Test, usr_pred_proba, multi_class = "ovo")
    print("Roc_Auc_Score_usr = ", Roc_Auc_Score_usr)

    usr_pred, Res_pred = trainCNN.predict(usr_pred_proba, Res_pred_proba, classesUser, threshold = 0.5)
    cnf_matrix_usr = confusion_matrix(c_usr_Test, 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)

    TN_usr = np.asarray(TN_usr)
    TP_usr = np.asarray(TP_usr)
    FP_usr = np.asarray(FP_usr)
    FN_usr = np.asarray(FN_usr)
    print("\n")
    print("TN_usr = ", sum(TN_usr))
    print("TP_usr = ", sum(TP_usr))
    print("FP_usr = ", sum(FP_usr))
    print("FN_usr = ", sum(FN_usr))
    FAR_usr = FP_usr / (FP_usr + TN_usr)
    FRR_usr = FN_usr / (TP_usr + FN_usr)
    print("FAR_usr = ", FAR_usr)
    print("FRR_usr = ", FRR_usr)
    plt.figure(figsize=(15,15))
    plot_confusion_matrix(cnf_matrix_usr, classes=classesUser, normalize=False, title='confusion matrix')
    plt.show()
    plt.close()
```

Figure 22. Code for confusion matrix for users

Some of our predictions

Figure 21. some of our predictions and their ground truth for user Identification

Figure 22. some of our predictions and their ground truth for Genuine/Forgery Detection

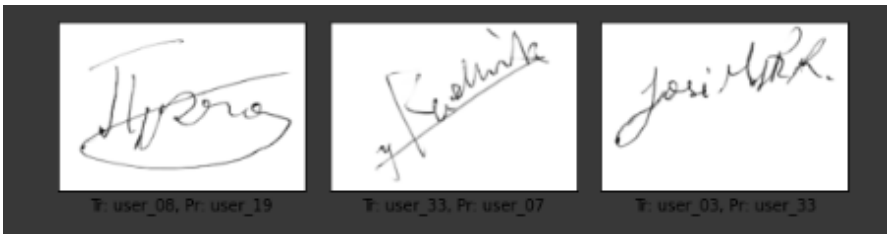

Figure 23. Misclassified Users

Simulating Sample Biometric System Authentication

To have real sense of Biometric system based Authentication we write a Authentication protocol based on our CNN model. After getting new persons signature our system is performing preprocessing and gave the preprocessed data to our model and then after prediction we do our Authentication mechanism mentioned in Figure 2 and give access in case the probe passes our test.

```python
def SingleTestImg(img_Add, threshold):
    img = cv2.imread(img_Add ,cv2.IMREAD_GRAYSCALE)
    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
    _, thresh1 = cv2.threshold(img,15,1,cv2.THRESH_BINARY) # return mask in binary format
    img = img * thresh1

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

    img = np.expand_dims(img, axis=2)
    img = np.expand_dims(img, axis=0)
    img = np.asarray(img)

    print(img.shape)

    """ User prediction section based on argmax """
    usr_pred_proba = trainCNN.predict_usr_proba(img)
    Res_pred_proba = trainCNN.predict_Res_proba(img)

    usr_pred = np.argmax(usr_pred_proba, axis = 1)
    pred_usr = usrAllLabelBinarizer.classes_[usr_pred[0]]
    pred_Res = (Res_pred_proba >= threshold)[0]

    return pred_usr, pred_Res

```

Figure 24. Function to get prediction after giving threshold and Raw Signature

```python
threshold = 0.4
Forg = True
if Forg:
    img_Add = root_path + "/forgeries_3_2.png"
    claimed_usr = "user_03"
else:
    img_Add = root_path + "/original_5_1.png"
    claimed_usr = "user_06"

pred_usr, pred_Res = SingleTestImg(img_Add, threshold)


print("\npred_usr: ", pred_usr, " - ", "pred_Res: ", pred_Res)


if claimed_usr == pred_usr:
    print("\n[Access Attempt for User] : ", claimed_usr)
    if pred_Res == True:
        print("\n[Forgery Attempt for] : ", claimed_usr)
        print("\n[Access Denied ....] ")
    else:
        print("[Original Attempt for] : ", claimed_usr)
        print("\n[.... Welcome ....] ", claimed_usr)
else:
    print("\n[Signature is not for User]: ", claimed_usr)
    print("\n[Access Denied ....] ")

```

Figure 25. Authentication Process

Writer-Dependent (WD)

After learning feature vector, we used them to train Writer-Dependent classifiers on a disjoint set of users from WI-Set to analyze the learned feature space performance and prove that these features are generalized to other users. We trained different SVM classifiers for each user as we know in real-world applications the number of users varies in the system. We used RBF kernel SVM classifier with C=1000 as our WD classifier. first, we used half of Genuine and Forgery Signatures of each user to train and evaluate SVM classifiers, then we realize that oversampling of train data by three times, improves the performance of our classifiers two times as we have more data to train. We reported mean FAR, FRR, and EER computed based on performance evaluation section on all users in WD-Set. We used KFOLD cross validation with 5-fold using stratifiedkfold and reported mean results for all metrics., we also listed used number of signatures to train and evaluate with evaluation results. As we can see CEDAR has best results of 0% EER.

Table4 . Results for SVMs in WD mode for CEDAR using stratifiedkfold (k=5)

Dataset	Train samples	Test samples	FAR	FRR	EER
CEDAR	15*24=360	36*24=720	0%	0%	0%

```
skf = StratifiedKFold(n_splits= 5)
skf_FAR_All, skf_FRR_All, skf_EER_All, skf_scoreUsr, skf_scoreRes = [],[],[],[],[]
for train, test in skf.split(X_FL, c_usr_FL):
    start_time_Fold = time.clock()
    X_Train_FL, X_Test_FL = X_FL[train] , X_FL[test]
    usr_Train_FL, usr_Test_FL = usr_FL_LB[train], usr_FL_LB[test]
    Res_Train_FL, Res_Test_FL = Res_FL[train], Res_FL[test]
    c_usr_Train_FL, c_usr_Test_FL = c_usr_FL[train], c_usr_FL[test]

    skf_number+=1
    model = None
    trainCNN = TrainCNN(model, num_usr_FL, originalPaper, origEpochs, ourEpoch, batch_size,\
                        IMAGE_WIDTH, IMAGE_HEIGHT, dataset_name1, fc1_Size, fc2_Size)

    trainCNN.train(X_Train_FL, usr_Train_FL, Res_Train_FL, X_Test_FL, usr_Test_FL, Res_Test_FL)
    trainCNN.save_CNN_model(skf_number)
    """ accuracy of model """
    score = trainCNN.model.evaluate(X_Test_FL, {"User": usr_Test_FL, "Res": Res_Test_FL})
    print("[INFO] skf_number: ", skf_number ," accuaracy of User: ", score[-2], "accuaracy of Res: ", score[-1])
    skf_scoreUsr.append(score[-2])
    skf_scoreRes.append(score[-1])

    # getting the last FC layer
    model1 = Model(inputs= trainCNN.model.inputs, outputs= trainCNN.model.layers[-5].output)
    FAR_All, FRR_All, EER_All = [],[],[]

    for c_usr in user_classes_WD:
        X_perUsr_WD_Train, Res_perUsr_WD_Train, X_perUsr_WD_Test, Res_perUsr_WD_Test = getTrainWD_PerUser(X_WD, Res_WD, c_usr_WD, c_usr)

        # get extracted features
        TrainFeature = model1.predict(X_perUsr_WD_Train)
        TestFeature = model1.predict(X_perUsr_WD_Test)

        FAR, FRR, EER = train_ReportResults(TrainFeature, Res_perUsr_WD_Train, TestFeature, Res_perUsr_WD_Test)

        FAR_All.append(FAR)
        FRR_All.append(FRR)
        EER_All.append(EER)

    print("FAR_All mean = ", np.mean(FAR_All))
    print("FRR_All mean = ", np.mean(FRR_All))
    print("EER_All mean = ", np.mean(EER_All))

    skf_FAR_All.append(np.mean(FAR_All))
    skf_FRR_All.append(np.mean(FRR_All))
    skf_EER_All.append(np.mean(EER_All))
    running_time_Fold = (time.clock() - start_time_Fold)/60 # in minutes
    print(running_time_Fold, "minutes")

print("skf_FAR_All mean = ", np.mean(skf_FAR_All))
print("skf_FRR_All mean = ", np.mean(skf_FRR_All))
print("skf_EER_All mean = ", np.mean(skf_EER_All))
print("skf_scoreUsr mean = ", np.mean(skf_scoreUsr))
print("skf_scoreRes mean = ", np.mean(skf_scoreRes))
print("==========================================Microsoft To Do==========")
```

Figure 26. code for Feature Transfer and training our SVM with 5 Fold

```
[ ]  1 def getTrainWD_PerUser(X_WD, Res_WD, c_usr_WD, c_usr):
     2     X_perUsr_WD_Train, Res_perUsr_WD_Train, X_perUsr_WD_Test, Res_perUsr_WD_Test = [], [], [], []
     3     GenNumber, usrForgNumber, ForgNumber = 0, 0, 0
     4     for i, X in enumerate(X_WD):
     5         if c_usr_WD[i] == c_usr:
     6             if Res_WD[i] == 0:
     7                 if GenNumber % 2 == 1: #half genuine to tain half to test
     8                     X_perUsr_WD_Train.append(X_WD[i])
     9                     Res_perUsr_WD_Train.append(Res_WD[i])
    10                     X_perUsr_WD_Train.append(X_WD[i])
    11                     Res_perUsr_WD_Train.append(Res_WD[i])
    12                     X_perUsr_WD_Train.append(X_WD[i])
    13                     Res_perUsr_WD_Train.append(Res_WD[i])
    14                     GenNumber = GenNumber + 1
    15                 else:
    16                     X_perUsr_WD_Test.append(X_WD[i])
    17                     Res_perUsr_WD_Test.append(Res_WD[i])
    18                     GenNumber = GenNumber + 1
    19             else: # all forgeries in test
    20                 if usrForgNumber % 2 == 1:
    21                     X_perUsr_WD_Train.append(X_WD[i])
    22                     Res_perUsr_WD_Train.append(Res_WD[i])
    23                     X_perUsr_WD_Train.append(X_WD[i])
    24                     Res_perUsr_WD_Train.append(Res_WD[i])
    25                     X_perUsr_WD_Train.append(X_WD[i])
    26                     Res_perUsr_WD_Train.append(Res_WD[i])
    27                     usrForgNumber = usrForgNumber + 1
    28                 else:
    29                     X_perUsr_WD_Test.append(X_WD[i])
    30                     Res_perUsr_WD_Test.append(Res_WD[i])
    31                     usrForgNumber = usrForgNumber + 1
    32
    33     X_perUsr_WD_Train = np.asarray(X_perUsr_WD_Train)
    34     Res_perUsr_WD_Train = np.asarray(Res_perUsr_WD_Train)
    35     X_perUsr_WD_Test = np.asarray(X_perUsr_WD_Test)
    36     Res_perUsr_WD_Test = np.asarray(Res_perUsr_WD_Test)
    37
    38     return X_perUsr_WD_Train, Res_perUsr_WD_Train, X_perUsr_WD_Test, Res_perUsr_WD_Test
    39
```

Figure 27. function to get Train and Test data for each user, we used over Sampling on train data

```
    40
    41 def train_ReportResults(TrainFeature, Res_perUsr_WD_Train, TestFeature, Res_perUsr_WD_Test):
    42     clf =  SVC(kernel="rbf", C=1000, gamma='scale', probability=False)
    43     clf.fit(TrainFeature, Res_perUsr_WD_Train)
    44     ensemble_pred = clf.predict(TestFeature)
    45     classesRes = [0,1]
    46     from sklearn import metrics
    47
    48     cnf_matrix_Res = confusion_matrix(Res_perUsr_WD_Test, ensemble_pred, labels=classesRes)
    49
    50     TP_Res = cnf_matrix_Res[0][0]
    51     TN_Res = cnf_matrix_Res[1][1]
    52     FN_Res = cnf_matrix_Res[0][1]
    53     FP_Res = cnf_matrix_Res[1][0]
    54
    55     FAR = FP_Res / (FP_Res + TN_Res)
    56     FRR = FN_Res / (TP_Res + FN_Res)
    57     EER = (FRR + FAR)/2
    58
    59     return FAR, FRR, EER
```

Figure 28. function for SVM and it's Evaluation

Conclusion

In this work, we tried to tackle the HSV problem in WD mode in the presence of Skilled forgeries. we designed CNN model with last FC layer of size 1024 to extract feature space to distinguish between Genuine and Forgery signatures of each user and we used the learned features to train a WD SVM classifier to train on each user individually as we know the number of users in real-world applications is not fixed. We tried to tackle a problem by surpassing state-of-the-art performances and presenting more compact feature space to train WD or WI classifiers using these features, we introduced feature space of size 1024 and also a CNN model with less learnable parameters in comparison to literature works that makes our model faster in training phases. We reach better performance on CEDAR dataset with EER of 0% in WD model and accuracy of almost 100% in WI feature learning using stratifiedkfold (k=5) validation.

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/