

College of Engineering Guindy, Anna University

Department of Computer Science and Engineering

CS6301

Machine Learning

Project Documentation

Off-Line Handwritten Signature Verification

Aditya Lakshmanan	2019103003
Kavya Sridhar	2019103027
Ishwarya S	2019103528

Table of Contents

Introduction.....	3
Dataset Used	4
Model 1: Writer Dependent Model	5
Block Diagram	5
Detailed Block Diagram.....	5
Image Pre-processing	6
Feature Extraction.....	6
Training and Testing	7
Verification Result	8
Model 2: Writer Dependent Model	10
Block Diagram	10
Detailed Block Diagram.....	10
Generating Labelled Dataset	11
Image Pre-processing	11
Model Development.....	12
Training the Model.....	14
Performance Metrics and Evaluation	16
Sample Test.....	18
Conclusion and Final Summary	19
References.....	20

Introduction

Handwritten signature is one of the most important forms of biometric authorizations that are used universally to authorize and verify documents. However, they are among the more vulnerable metrics since they can be easily forged thereby leading to drastic effects such as identity theft, information theft, etc. Thus, a need for identifying the genuineness of a signature is essential.

The primary difficulty faced by signature recognition systems is the intra-class variation, i.e., an individual's original sign may not be the same each time they sign it because several external and internal factors affect it, such as, the pen used, the surface used, etc. Thus, any model that aims at performing signature recognition must account for these variations.

There are two ways to identify the signatures, off-line signature recognition and on-line signature recognition. On-line signature assesses the parameters such as speed of writing, the force employed on the signing instrument, the position of the hand, etc., all of which are physical factors that are outside the scope of this project. Off-line signature verification essentially treats the signature as an image and uses Neural Networks to identify the validity of the signature.

In a writer dependent model, the characteristics included are writer dependent threshold, features and classifiers. However, this simple image classifying neural network is not sufficient since a new neural network would have to be trained for each and every individual writer, which is highly inefficient and ineffective.

The neural network proposed by the base paper (see references), is a Siamese Neural Network, i.e., a network which consists of twin convolutional networks accepting two distinct signature images that are either similar or dissimilar. This makes the neural network writer independent and requires a genuine signature to be passed along with the sample signature to determine its validity.

This project implements both the methods to contrast their performances.

Dataset Used

CEDAR Signature is a database of off-line signatures for signature verification. Each of 55 individuals contributed 24 signatures thereby creating 1,320 genuine signatures. Some were asked to forge three other writers' signatures, eight times per subject, thus creating 1,320 forgeries. The database has 24 genuine and 24 forgeries available for each writer.

[Link to CEDAR Dataset](#)

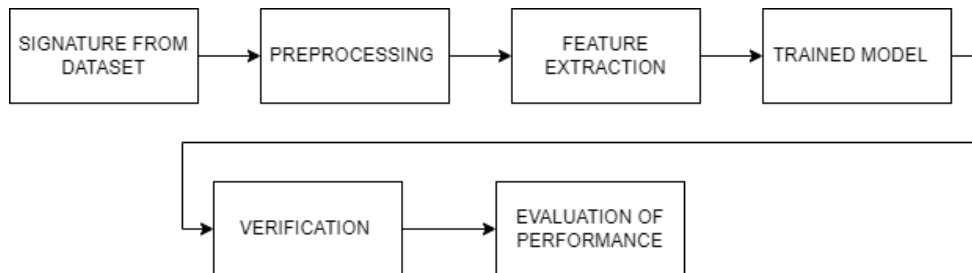


Sample from the dataset, all the signatures belong to one individual. The first 3 rows are genuine signatures while the last 3 rows are forged signatures.

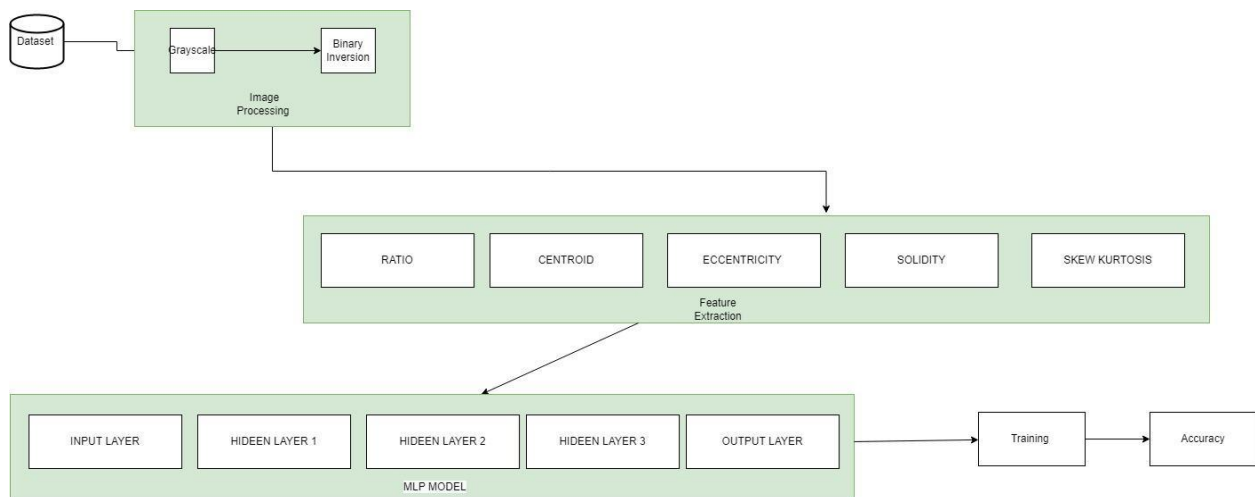
MODEL 1: WRITER DEPENDENT MODEL

This model is based on writer dependent features which are selected based on a score computed for each feature of the respective writer, thereby resulting in selection of different sets of features for different writers.

Block Diagram



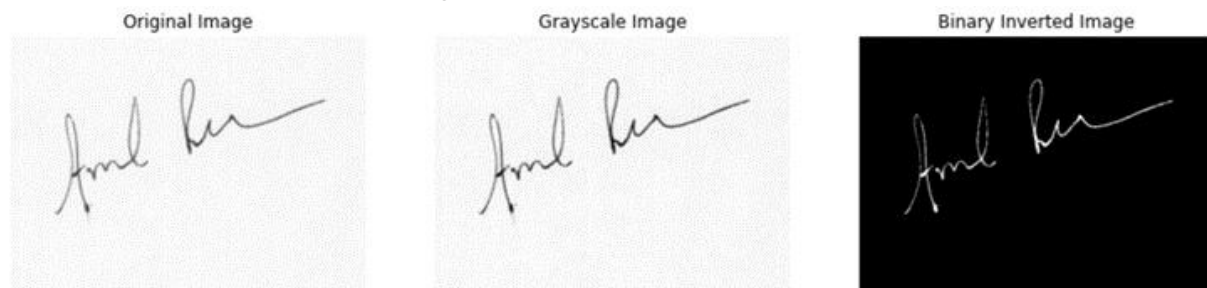
Detailed Block Diagram



Pre-Processing

The images are preprocessed using the skimage library and the following are the steps:

- Gray Scale Conversion: colored images are converted to grayscale
- Image thresholding and binary inversion: that finds the optimal threshold value, to invert the image



Feature Extraction

- Ratio
 - It is the ratio of area of signature image to the area of signature enclosed in a bounding box. Area of a signature is the number of pixels comprising it.
- Centroid
 - To find the centroid of a group of pixels in a binary image you must calculate the average coordinate. In case of a grayscale image you can use the pixels' gray values to calculate the weighted average position.
- Eccentricity
 - Eccentricity measures the shortest length of the paths from a given vertex v to reach any other vertex w of a connected graph.
 - Computed for every vertex v it transforms the connectivity structure of the graph into a set of values. For a connected region of a digital image it is defined through its neighbourhood graph and the given metric.
- *Solidity*
 - Solidity is an area fraction of the region as compared to its convex hull.
 - Solidity is what fraction of the actual area your region is.
- *Skew*
 - Skewness is a measure of (lack of) symmetry .
- *Kurtosis*
 - It is the average (or expected value) of the standardized data raised to the fourth power.

Extracted Features

The below is the csv file containing extracted features of person-001.

	A	B	C	D	E	F	G	H	I	J
1	ratio	cent_y	cent_x	eccentricity	solidity	skew_x	skew_y	kurt_x	kurt_y	output
2	0.09948580371	0.45702916	0.4715883315	0.983489911	0.2038946163	-0.1244224392	0.1028318142	-1.157850239	-1.061720273	1
3	0.08860759494	0.4580049417	0.434849747	0.9813790058	0.1894384756	0.05193217083	0.008243988213	-1.07416505	-1.099963143	1
4	0.08802572984	0.4303218399	0.4500960822	0.980818192	0.1973158829	-0.001757620816	0.1478897099	-1.163584582	-0.7820605896	1
5	0.08770614693	0.4179606625	0.4801542191	0.9821748249	0.203301477	-0.05162119329	0.2963692638	-1.293496409	-1.068801346	0
6	0.1009933775	0.4869521579	0.4916088262	0.9718915028	0.2243039748	-0.08225708915	-0.04465535156	-1.221924757	-1.121507753	0
7	0.09649122807	0.5121548822	0.5132974482	0.9685224662	0.1974274604	-0.1171870432	-0.1374379904	-1.305131554	-1.210357481	0

The below is the csv file containing extracted features of person-005.

	A	B	C	D	E	F	G	H	I	J
1	ratio	cent_y	cent_x	eccentricity	solidity	skew_x	skew_y	kurt_x	kurt_y	output
2	0.1167108753	0.3987652972	0.4060173393	0.9157903137	0.1826673586	0.2625918924	0.6410775715	-0.8745919151	-0.3215103684	1
3	0.1098197343	0.4068835783	0.3933188286	0.9184126334	0.1788335265	0.5235080593	0.4819844505	-0.2634696506	-0.3247110212	1
4	0.09264705882	0.4362411631	0.3833981212	0.8747355239	0.1461232604	0.3508114778	0.2175652309	-0.5741265402	-0.3348734218	1
5	0.0963362069	0.5361087808	0.4271387796	0.8967015447	0.1572836031	0.6870442903	-0.4266353879	-0.4287001034	-0.06384794063	0
6	0.06951423786	0.4822132976	0.4038707594	0.8598167928	0.1269501377	0.7706418956	-0.4668607196	0.4176178795	-0.14631009	0
7	0.08540372671	0.5005036515	0.3662040477	0.8771676115	0.1585735964	0.8269717899	-0.2871376913	-0.1626819295	-0.08520309319	0

Training and Testing

```
#define the model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with sigmoid activation function
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.sigmoid(layer_1)
    # Hidden layer with sigmoid activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.sigmoid(layer_2)
    # Hidden layer with sigmoid activation
    layer_3 = tf.add(tf.matmul(layer_2, weights['h3']), biases['b3'])
    layer_3 = tf.nn.sigmoid(layer_3)
    # Hidden layer with sigmoid activation
    layer_4 = tf.add(tf.matmul(layer_3, weights['h4']), biases['b4'])
    layer_4 = tf.nn.sigmoid(layer_4)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_4, weights['out']) + biases['out']
    return out_layer
```

```

def evaluate(train_path, test_path, type2=False):
    if not(type2):
        train_input, corr_train, test_input, corr_test = readCSV(train_path, test_path)
    else:
        train_input, corr_train, test_input = readCSV(train_path, test_path, type2)
    ans = 'Random'
    with tf.compat.v1.Session() as sess:
        sess.run(init)
        # Training cycle
        for epoch in range(training_epochs):
            # Run optimization op (backprop) and cost op (to get loss value)
            _, cost = sess.run([train_op, loss_op], feed_dict={x: train_input, y_: corr_train})
            if cost<0.0001:
                break
        # Finding accuracies
        accuracy1 = accuracy.eval({x: train_input, y_: corr_train})
        print("Accuracy for training model:", accuracy1)
        prediction = pred.eval({x: test_input})

        if prediction[0][1]>prediction[0][0]:
            print('Genuine Image')
            return True
        else:
            print('Forged Image')
            return False

```

Verification Result & Accuracy

For Person with id 001

1. When a real signature is given.

```

n_input = 9
train_person_id = input("Enter person's id : ")
test_image_path = input("Enter path of signature image : ")
train_path = 'drive/MyDrive/ML MINI PROJECT/FEATURES/Training/training_'+train_person_id+'.csv'
testing(test_image_path)
test_path = 'drive/MyDrive/ML MINI PROJECT/TestFeatures/testcsv.csv'

```

Enter person's id : 001
Enter path of signature image : drive/MyDrive/ML MINI PROJECT/real/001001_001.png

```

evaluate(train_path, test_path, type2=True)

```

Accuracy for training model: 0.5
Genuine Image
True

2. When a forged signature is given

```
h_input = 9
train_person_id = input("Enter person's id : ")
test_image_path = input("Enter path of signature image : ")
train_path = 'drive/MyDrive/ML MINI PROJECT/FEATURES/Training/training_'+train_person_id+'.csv'
testing(test_image_path)
test_path = 'drive/MyDrive/ML MINI PROJECT/TestFeatures/testcsv.csv'
```

Enter person's id : 001
Enter path of signature image : drive/MyDrive/ML MINI PROJECT/forged/021001_001.png

```
evaluate(train_path, test_path, type2=True)
```

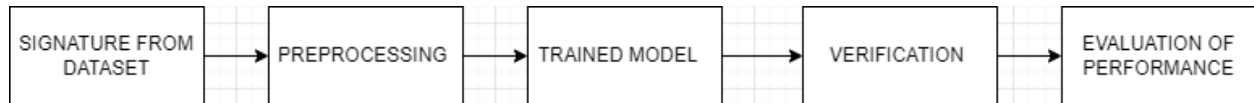
Accuracy for training model: 0.5
Forged Image
False

Thus, the model achieved only 50% accuracy as expected for a simple MLP model that is writer dependent.

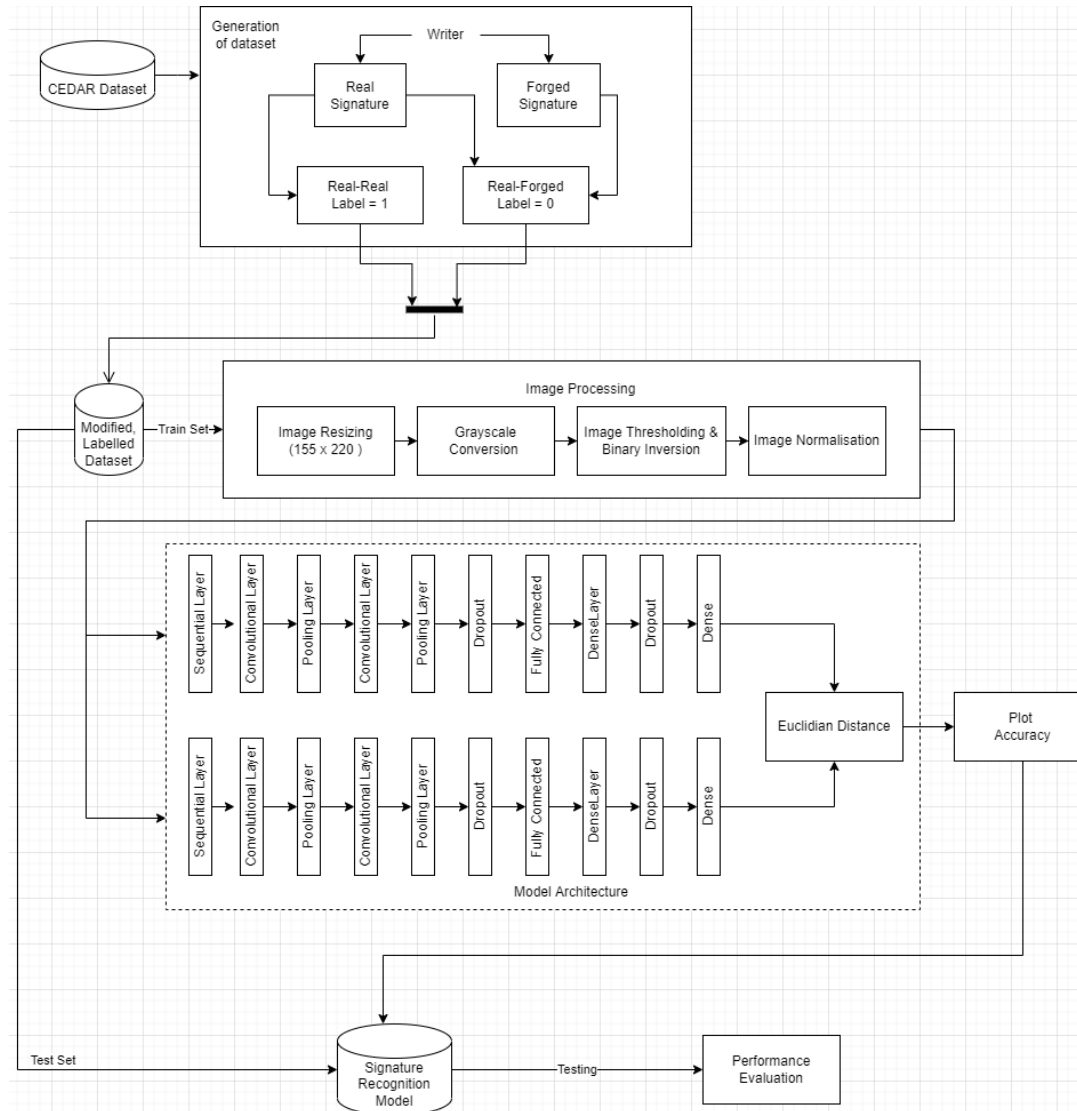
MODEL 2: WRITER INDEPENDENT MODEL

To better the results of the model above, the SigNet proposed in the base paper was implemented as below.

Block Diagram



Detailed Block Diagram



Generation of Labelled Dataset

A labelled dataset is generated from the original dataset, consisting of rows of pairs of images along with the label.

Two kinds of pairs created, a real-real with label 1, and a real-forged with label 0.

index	sign_1	sign_2	label
22	drive/MyDrive/ML PROJECT/full_org/original_1_23.png	drive/MyDrive/ML PROJECT/full_org/original_1_23.png	1
23	drive/MyDrive/ML PROJECT/full_org/original_1_24.png	drive/MyDrive/ML PROJECT/full_org/original_1_24.png	1
24	drive/MyDrive/ML PROJECT/full_org/original_1_1.png	drive/MyDrive/ML PROJECT/full_forg/forgeries_1_1.png	0
25	drive/MyDrive/ML PROJECT/full_org/original_1_2.png	drive/MyDrive/ML PROJECT/full_forg/forgeries_1_2.png	0

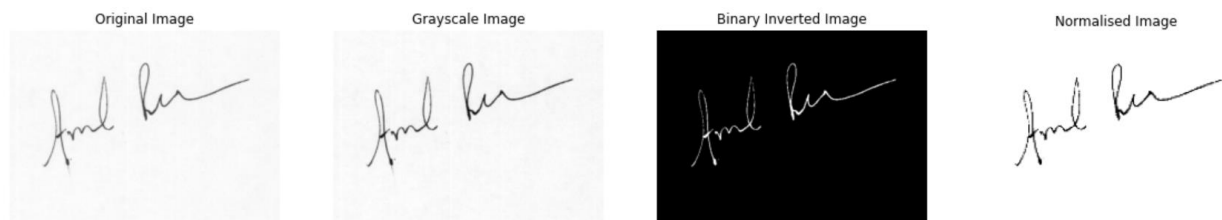
The dataset is then split into testing and training, with test_ratio = 0.3.

Image Processing

The images are preprocessed using the OpenCV library and the following are the steps:

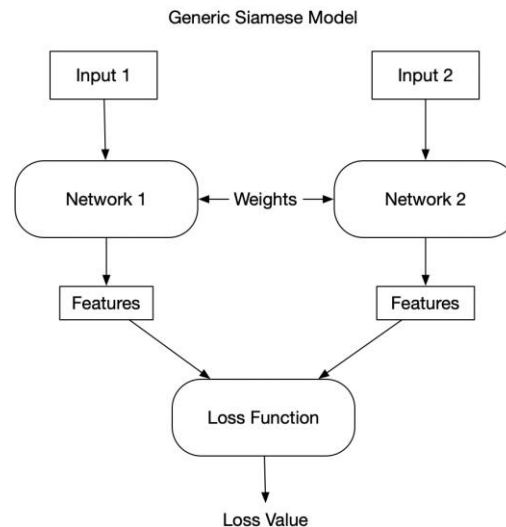
1. Resizing: the images are resized to 220 x 155 pixels using bilinear interpolation
2. Gray Scale Conversion: colored images are converted to grayscale
3. Image thresholding and binary inversion: that finds the optimal threshold value, to invert the image
4. Normalization: the images are normalized by 255

```
def preprocess_image(img):  
    resized_signature = cv2.resize(img,(220,155),cv2.IMREAD_GRAYSCALE)  
    gray_signature=cv2.cvtColor(resized_signature, cv2.COLOR_BGR2GRAY)  
    ret,thr_img = cv2.threshold(gray_signature, 0, 255, cv2.THRESH_OTSU + cv2.THRESH_BINARY_INV)  
    normalized_signature=thr_img/255  
    signature_expanded = normalized_signature[:, :, np.newaxis]  
  
    return signature_expanded
```



The training dataset is passed through a Sequence class where they are split into batch sizes, 64 and are randomized after every epoch. Further, a generator is used to generate the images in the processed form, and the generator is returned as input to the model.

Model Development



The model was built using TensorFlow with slight modifications to the model presented in the base paper, which produced a greater accuracy.

The input accepted by the model is (155,220,1) and both the images are passed to the model, where they now split into left_image and right_image. They are trained separately to extract the features using the layers described below:

1. **Sequential:** Sequential groups a linear stack of layers into a Model. Sequential provides training and inference features on this model
2. **Convolutional 2D:** Convolutional neural networks apply a filter to an input to create a feature map that summarizes the presence of detected features in the input.
3. **MaxPooling :** Downsamples the input along its spatial dimensions (height and width) by taking the maximum value over an input window
4. **DropOut :** The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting
5. **Flatten:** A flatten operation on a tensor reshapes the tensor to have the shape that is equal to the number of elements contained in tensor non including the batch dimension.
6. **Dense:** Dense Layer is deeply connected with its preceding layer, relu activation is used.

They are then combined and passed to the Model() where the dense layers produced on both sides of the Siamese network is measured by a similarity metric containing Euclidean distance.

```
def sig_network_model():
    input_shape=(155,220,1)
    in_imgLeft = Input(shape=input_shape, name="left_image")
    in_imgRight = Input(shape=input_shape, name="right_image")

    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation="relu", input_shape=input_shape))
    model.add( MaxPooling2D(pool_size=(3,3)) )
    model.add( Conv2D(64, (3, 3), activation="relu") )
    model.add( MaxPooling2D(pool_size=(3,3),strides=(2,2)) )

    model.add( Conv2D(128, (3, 3), activation="relu") )
    model.add( Conv2D(64, (3, 3), activation="relu") )
    model.add( Conv2D(128, (3, 3), activation="relu") )

    model.add( MaxPooling2D(pool_size=(3,3),strides=(2,2)) )
    model.add( Dropout(0.3) )
    model.add( Flatten() )

    model.add( Dense(256, activation="relu") )
    model.add( Dropout(0.5) )
```

```
model.add( Dense(256, activation="relu") )

left_branch = model(in_imgLeft)
right_branch = model(in_imgRight)

cal = Lambda(euclidean_distance,output_shape=output_shape)
distance=cal([left_branch, right_branch])

model = Model([in_imgLeft, in_imgRight], distance)
return model
```

Model Summary

Layer (type)	Output Shape	Param #	Connected to
=====			
left_image (InputLayer)	(None, 155, 220, 1)	0	
right_image (InputLayer)	(None, 155, 220, 1)	0	
sequential_1 (Sequential)	(None, 256)	3714240	left_image[0][0] right_image[0][0]
lambda_2 (Lambda)	(None, 1)	0	sequential_1[1][0] sequential_1[2][0]
=====			
Total params: 3,714,240			
Trainable params: 3,714,240			
Non-trainable params: 0			
=====			

Euclidean Distance: is the metric used to measure the similarity between the two images

```
def euclidean_distance(vectors):
    xi, yi = vectors
    return K.sqrt(K.sum(K.square(xi - yi), axis=1, keepdims=True))
```

Training the Model

Contrastive Loss Function is used as the loss function of the model. It aims at dimensionality reduction by learning an invariant mapping that produces high to low dimensional space maps similar input vectors to nearby points on the output manifold and dissimilar vectors to distant points.

Contrastive loss takes the output of the network for a positive example and calculates its distance to an example of the same class and contrasts that with the distance to negative examples. Thus, the loss is low if positive samples are encoded to similar (closer) representations and negative examples are encoded to different (farther) representations.

For two inputs X_1 and X_2 , contrastive loss function is calculated as such:

$$D_W(\vec{X}_1, \vec{X}_2) = \|G_W(\vec{X}_1) - G_W(\vec{X}_2)\|_2$$

$$\mathcal{L}(W) = \sum_{i=1}^P L(W, (Y, \vec{X}_1, \vec{X}_2)^i) \quad (2)$$

$$L(W, (Y, \vec{X}_1, \vec{X}_2)^i) = (1 - Y)L_S(D_W^i) + YL_D(D_W^i) \quad (3)$$

```
def contrastive_loss(l, y_pred):
    margin = 1
    margin_square = (K.maximum(margin - y_pred, 0))**2
    return K.mean((1 * (y_pred)**2)*1 + (1 - 1)* margin_square)
```

```
def accuracy(y_true, y_pred):
    casted_ytrue=K.cast(y_pred < 0.5, 'float32')
    return K.mean(K.equal(y_true,casted_ytrue ))
```

RMS Prop is used as the optimizer for the model.

The gist of RMSprop is to:

- Maintain a moving (discounted) average of the square of gradients
- Divide the gradient by the root of this average

This implementation of RMSprop uses plain momentum.

```

model = sig_network_model()

optimizer = keras.optimizers.RMSprop(lr=1e-4, rho=0.9, epsilon=1e-08)
model.compile(loss=contrastive_loss, optimizer=optimizer, metrics=[accuracy])

history=model.fit_generator(generator=data_train, validation_data=data_validation, epochs=20,
                           steps_per_epoch=56, validation_steps=24, use_multiprocessing=True,
                           workers=6)

```

The model is fit to the training dataset for 20 epochs.

```

Epoch 1/20
56/56 [=====] - 457s 8s/step - loss: 0.2453 - accuracy: 0.5759 - val_loss: 0.2875 - val_accuracy: 0.5703
Epoch 2/20
56/56 [=====] - 436s 8s/step - loss: 0.1827 - accuracy: 0.7280 - val_loss: 0.1680 - val_accuracy: 0.7116
Epoch 3/20
56/56 [=====] - 440s 8s/step - loss: 0.1509 - accuracy: 0.7782 - val_loss: 0.1482 - val_accuracy: 0.8073
Epoch 4/20
56/56 [=====] - 438s 8s/step - loss: 0.1281 - accuracy: 0.8195 - val_loss: 0.1666 - val_accuracy: 0.8164
Epoch 5/20
56/56 [=====] - 437s 8s/step - loss: 0.1102 - accuracy: 0.8477 - val_loss: 0.1122 - val_accuracy: 0.8464
Epoch 6/20
56/56 [=====] - 440s 8s/step - loss: 0.0954 - accuracy: 0.8722 - val_loss: 0.1135 - val_accuracy: 0.8542
Epoch 7/20
56/56 [=====] - 441s 8s/step - loss: 0.0834 - accuracy: 0.8906 - val_loss: 0.0391 - val_accuracy: 0.8770
Epoch 8/20
56/56 [=====] - 436s 8s/step - loss: 0.0735 - accuracy: 0.9079 - val_loss: 0.0725 - val_accuracy: 0.8848
Epoch 9/20
56/56 [=====] - 440s 8s/step - loss: 0.0655 - accuracy: 0.9216 - val_loss: 0.0448 - val_accuracy: 0.8880
Epoch 10/20
56/56 [=====] - 444s 8s/step - loss: 0.0592 - accuracy: 0.9266 - val_loss: 0.1117 - val_accuracy: 0.8867
Epoch 11/20
56/56 [=====] - 431s 8s/step - loss: 0.0506 - accuracy: 0.9442 - val_loss: 0.1000 - val_accuracy: 0.8763
Epoch 12/20
56/56 [=====] - 438s 8s/step - loss: 0.0449 - accuracy: 0.9537 - val_loss: 0.0334 - val_accuracy: 0.9121
Epoch 13/20
56/56 [=====] - 443s 8s/step - loss: 0.0404 - accuracy: 0.9626 - val_loss: 0.0571 - val_accuracy: 0.9108

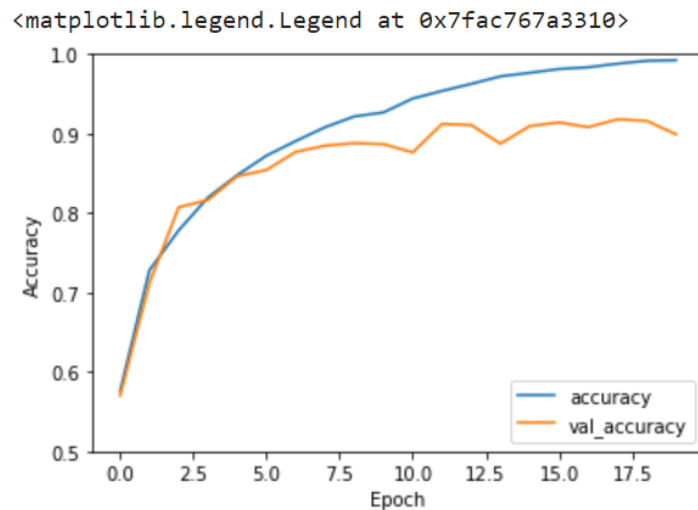
Epoch 14/20
56/56 [=====] - 441s 8s/step - loss: 0.0338 - accuracy: 0.9721 - val_loss: 0.0532 - val_accuracy: 0.8874
Epoch 15/20
56/56 [=====] - 434s 8s/step - loss: 0.0304 - accuracy: 0.9766 - val_loss: 0.0593 - val_accuracy: 0.9095
Epoch 16/20
56/56 [=====] - 441s 8s/step - loss: 0.0280 - accuracy: 0.9813 - val_loss: 0.0627 - val_accuracy: 0.9141
Epoch 17/20
56/56 [=====] - 445s 8s/step - loss: 0.0247 - accuracy: 0.9835 - val_loss: 0.0957 - val_accuracy: 0.9082
Epoch 18/20
56/56 [=====] - 436s 8s/step - loss: 0.0212 - accuracy: 0.9880 - val_loss: 0.0243 - val_accuracy: 0.9180
Epoch 19/20
56/56 [=====] - 443s 8s/step - loss: 0.0190 - accuracy: 0.9916 - val_loss: 0.0344 - val_accuracy: 0.9160
Epoch 20/20
56/56 [=====] - 449s 8s/step - loss: 0.0173 - accuracy: 0.9925 - val_loss: 0.0839 - val_accuracy: 0.8991

```

The validation accuracy improved from 0.5703 to a maximum of 0.9180.

Training Curve : The training and validation accuracies are plotted as shown and it evident that the model doesn't overfit or underfit the data.

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
```



Testing and Performance Evaluation

For testing the model, the labelled dataset is once again used without randomization after every epoch and the performance is derived as described below.

To mitigate the bias of class imbalance, if any, a moving threshold is calculated for calculating the precision and recall. Thus the ROC accuracy is computed with a range of thresholds on distances.

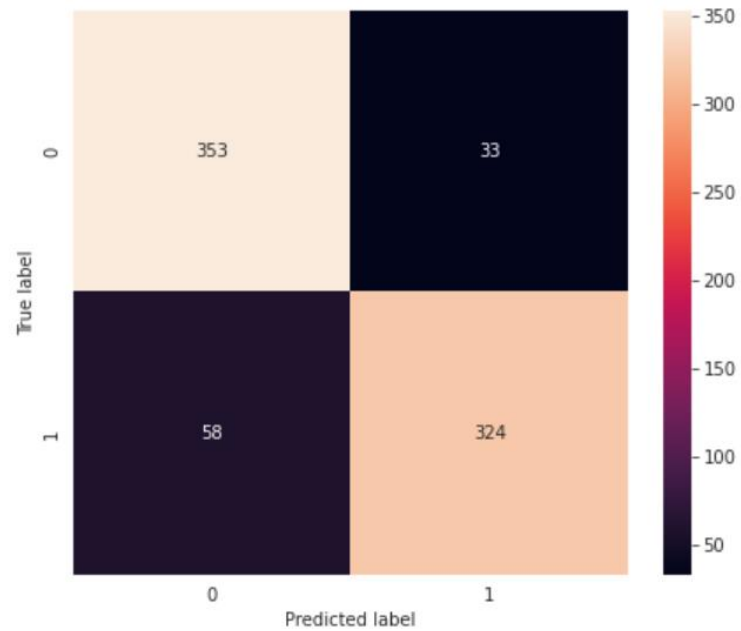
PERFORMANCE OF MODEL ON TESTING DATA

Best Threshold:0.713634087042883

Thus the Confusion matrix is obtained as such

Confusion Matrix:

```
[[353  33]
 [ 58 324]]
```

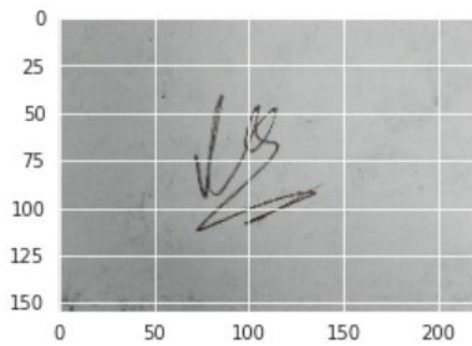
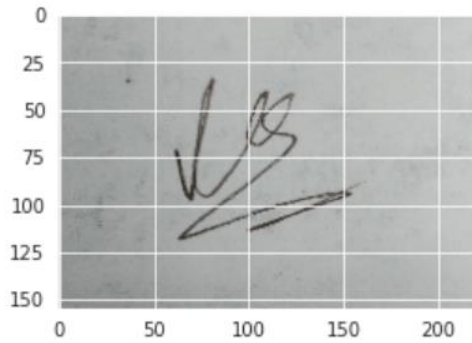



Accuracy 0.8815104166666666

Classification Report				
	precision	recall	f1-score	support
0	0.86	0.91	0.89	386
1	0.91	0.85	0.88	382
accuracy			0.88	768
macro avg	0.88	0.88	0.88	768
weighted avg	0.88	0.88	0.88	768

Model tested Sample

Predicted Y Value: `[[0.0660877]]`
Rounded Predicted Y Value: `[[0.]]`
The second signature is genuine



Conclusions and Final Summary

Thus, the importance of efficient recognition of handwritten signatures was explored in this project and suitable models were developed in accordance to the references.

For a writer dependent model, the images were processed and 9 features were extracted, including, Ratio, Centroid, Eccentricity, Solidity, Skew, Kurtosis, and an MLP was trained which gave an accuracy of 50%. This proved to be an inefficient model.

The writer independent approach proposed by the base paper was executed using a Siamese Neural Network (SigNet) and it exposed the network to a pair of similar and dissimilar observations and minimizing the Euclidean distance between similar pairs while simultaneously maximizing it between dissimilar pairs. The model trained and tested on the CEDAR Dataset gave an accuracy of 88.15%. The contrastive loss function was used as the loss function for this network.

References

- [1] Dey, Sounak & Dutta, Anjan & Toledo, J. & Ghosh, Suman & Lladós, Josep & Pal, Umapada. (2017). SigNet: Convolutional Siamese Network for Writer Independent Offline Signature Verification.
- [2] R. Hadsell, S. Chopra and Y. LeCun, "Dimensionality Reduction by Learning an Invariant Mapping," 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), 2006, pp. 1735-1742, doi: 10.1109/CVPR.2006.100.
- [3] Luiz G. Hafemann, Robert Sabourin and Luiz S. Oliveira, "Writer-independent Feature Learning for Offline Signature Verification using Deep Convolutional Neural Networks," 2016
- [4] Ashwini Pansare and Shalini Bhatia, "Handwritten Signature Verification using Neural Network," 2012, Foundation of Computer Science FCS, New York, USA