

# **Bengali and Hindi Signature Verification using Convolution Siamese Network**

*Project 2(CS892): Report submitted in partial fulfillment for  
the award of degree of*

**Bachelor of Technology**

*in*

**Computer Science and Engineering**

*by*

**Arpita Halder**

**Roll No. 27600117072**

**Registration No. 172760110239**

Under the supervision of

**Prof(Dr.)Bimal Datta**



**BUDGE-BUDGE INSTITUTE OF TECHNOLOGY**

**(Affiliated Under MAULANA ABUL KALAM AZAD UNIVERSITY OF  
TECHNOLOGY, WEST BENGAL)**

# Certificate of Recommendation

This is to certify that the work in this Thesis Report entitled “**Bengali and Hindi Signature Verification using Convolution Siamese Network**” submitted by Arpita Halder in the partial fulfillment of the requirement for the award of degree of Bachelor of Technology, in Computer Science and Engineering, during session 2020-2021 in the Department of Computer Science and Engineering, Budge-Budge Institute of Technology (affiliated under Maulana Abul Kalam Azad University of Technology, West Bengal), and this work has not been submitted elsewhere for a degree.

Place: Budge-Budge, Kolkata

Date:

Prof.(Dr.) Bimal Datta  
Head of Computer Science and Engineering Department

# Acknowledgement

*It is with great satisfaction and pride that we present our thesis on the project undertaken during 8th semester, for partial fulfillment of our Bachelor of Technology degree at Budge-Budge Institute of Technology (affiliated under Makaut). We are thankful to Prof. Bimal Datta for being our mentor during this endeavor. He has been the corner stone of our project and has guided us during periods of doubts and uncertainties. His ideas and inspirations have helped us make this nascent idea of ours into a fully fledged project. We are also thankful to all the professors of our department for being a constant source of inspiration and motivation during the course of the project. We offer our heartiest thanks to our friends for their constant inspirations. Last but not the least, we want to acknowledge the contributions of our parents and family members, for their constant motivation, inspirations and their belief in us that we could make it happen.*

*Arpita Halder*

# Motivation

The motivation behind the project is the growing need for a full proof signature verification scheme which can guarantee maximum possible security from fake signatures. The main contribution of our project is a Convolutional Siamese Network for Offline Bengali and Hindi Signature Verification. This, in contrast to other methods based on hand crafted features, has the ability to model generic signature forgery techniques and many other related properties that envelops minute inconsistency in signatures from the training data. In contrary to other one-shot image verification tasks, the problem with signature is far more complex because of subtle variations in writing styles independent of scripts, which could also encapsulate some degrees of forgery. Here we mine this ultra fine anamorphosis and created a generic model.

# Abstract

Verification of off-line signatures is one of the most challenging tasks in biometrics and document forensic science. In this thesis, we deal with Convolutional Siamese Network model which is capable of doing verification of Bengali and Hindi Signature. One particular advantage of Siamese Neural networks is the ability to generalize to new classes that it has not been trained on, and in fact, the number of classes that it is expected to support does not have to be known at training time. Also, the architecture commonly known as the Siamese network helped reduce the amount of training data needed for its implementation. The twin networks with shared weights were trained to learn feature space where similar observations are placed in proximity. Writer Independent verification model has been designed where an accuracy of **91.82%** has been obtained for **Bengali Dataset** and **84%** for **Hindi Dataset**

# Contents

<b>Certificate of Recommendation</b>	<b>i</b>
<b>Acknowledgement</b>	<b>ii</b>
<b>Motivation</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Online Signature Verification . . . . .	1
1.2 Offline Signature Verification . . . . .	2
1.3 Types of Forgeries . . . . .	2
<b>2 Related Works</b>	<b>3</b>
<b>3 Datasets</b>	<b>4</b>
<b>4 Methodology</b>	<b>7</b>
4.1 CNN and Siamese Network . . . . .	7
4.2 Loss Function . . . . .	7
4.3 Architecture . . . . .	8
4.4 Hardware and Software Requirements . . . . .	9
<b>5 Analysis and Results</b>	<b>10</b>
5.1 Analysis of Signature Schemes . . . . .	10
5.2 Result and Discussion . . . . .	10
<b>6 Conclusion and Future Work</b>	<b>13</b>
6.1 Conclusion . . . . .	13
6.2 Future Work . . . . .	13
<b>References</b>	<b>15</b>

# List of Figures

1.1	Types of Forgery (a). Original Signature (b). Random Forgery (c). Simple Forgery (d). Skilled Forgery . . . . .	2
3.1	Genuine and Forged signature image samples of Bengali dataset. . . . .	5
3.2	Genuine and Forged signature image samples of Hindi dataset. . . . .	6
4.1	Common Architecture of the model used for training Bengali and Hindi Dataset. . . . .	9

# List of Tables

4.1	Training Hyper-parameters . . . . .	9
5.1	Result of our framework obtained on Bengali and Hindi Dataset . . . . .	10
5.2	<b>Bengali Signature Verification</b> Left: Original; Right: Predicted . . . . .	11
5.3	<b>Hindi Signature Verification</b> Left: Original; Right: Predicted . . . . .	12



# Chapter 1

## Introduction

Signature is among the most popular and well-known biometric features that has been used since antiquity to verify different human-related entities, namely documents, forms, bank checks, individuals, etc. Therefore, signature verification is a critical task and many efforts have been made to remove the uncertainty involved in the manual authentication procedure, which makes signature verification an important research line in the field of machine learning and pattern recognition [1], [2]. Depending on the input format, signature verification can be of two types: (1) online and (2) offline. Capturing online signature needs an electronic writing pad together with a stylus, which can mainly record a sequence of coordinates of the electronic pen tip while signing. Apart from the writing coordinates of the signature, these devices are also capable of fetching the writing speed, pressure, etc., as additional information, which are used in the online verification process. On the other hand, the offline signature is usually captured by a scanner or any other type of imaging devices, which basically produces two dimensional signature images. As signature verification has been a popular research topic through decades and substantial efforts are made both on offline as well as on online signature verification purpose.

### 1.1 Online Signature Verification

Online verification systems generally perform better than their offline counter parts [4] due to the availability of complementary information such as stroke order, writing speed, pressure, etc. However, this improvement in performances comes at the cost of requiring a special hardware for recording the pen-tip trajectory, rising its system cost and reducing the real application scenarios. There are many cases where authenticating offline signature is the only option such as check transaction and document verification. Because of its broader application area, in this paper, we focus on the more challenging task- automatic offline signature verification. Our objective is to propose a convolutional Siamese neural network model to discriminate the genuine signatures and skilled forgeries.

## 1.2 Offline Signature Verification

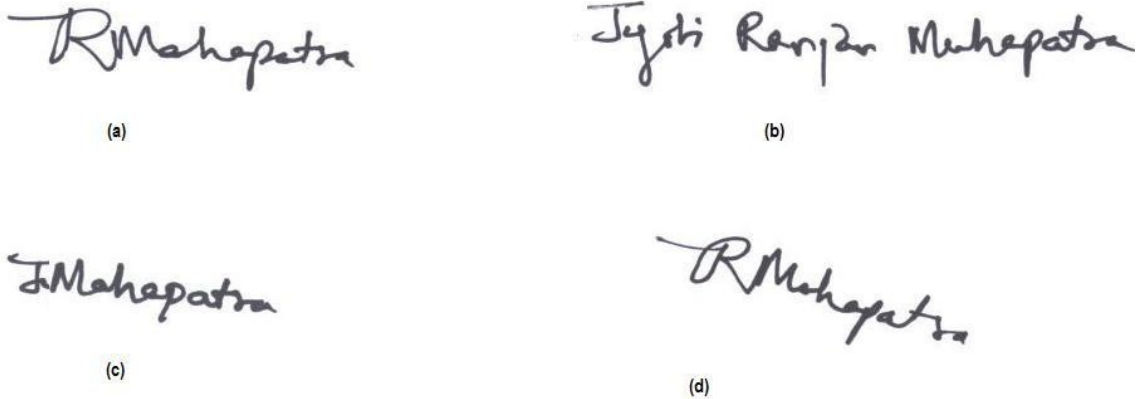
Offline signature verification can be addressed with (1) writer dependent and (2) writer independent approaches [3]. The writer independent scenario is preferable over writer dependent approaches, as for a functioning system, a writer dependent system needs to be updated (retrained) with every new writer (signer). For a consumer based system, such as bank, where every day new consumers can open their account this incurs huge cost. Whereas, in writer independent case, a generic system is built to model the discrepancy among the genuine and forged signatures. Training a signature verification system under a writer independent scenario, divides the available signers into train and test sets. For a particular signer, signatures are coupled as similar (genuine, genuine) or dissimilar (genuine, forged) pairs. From all the tuples of a single signer, equal number of tuples similar and dissimilar pairs are stochastically selected for balancing the number of instances. This procedure is applied to all the signers in the train and test sets to construct the training and test examples for the classifier.

## 1.3 Types of Forgeries

The various types of forgery include figure 1:

- a. **Random Forgery:** Random forgery is done by a person who doesn't know the shape and structure of the original signature. figure 1.1(b)
- b. **Simple Forgery:** In this type of forgery the person concerned has a vague idea of the actual signature, but is signing without much practice. figure 1.1(c)
- c. **Skilled Forgery:** This type of forgery considers appropriate knowledge about the original signature along with ample time for proper practice. Our proposed scheme eliminates random and simple forgeries and also reduces skilled forgery to a great extent. figure 1.1(d)

Figure 1.1: Types of Forgery (a). Original Signature (b). Random Forgery (c). Simple Forgery (d). Skilled Forgery



# Chapter 2

## Related Works

In this regard a signature verifier can be efficiently modelled by a Siamese network which consists of twin convolutional networks accepting two distinct signature images coming from the tuples that are either similar or dissimilar. The constituting convolutional neural networks (CNN) are then joined by a cost function at the top, which computes a distance metric between the highest level feature representation on each side of the network. The parameters between this twin networks are shared, which in turns guarantees that two extremely similar images could not possibly be mapped by their respective networks to very different locations in feature space because each network computes the same function. Different hand crafted features have been proposed for offline signature verification tasks. Many of them take into account the global signature image for feature extraction, such as, block codes, wavelet and Fourier series etc [5]. Some other methods consider the geometrical and topological characteristics of local attributes, such as position, tangent direction, blob structure, connected component and curvature [3]. Projection and contour based methods [6] are also quite popular for offline signature verification. Apart from the above mentioned methods, approaches fabricated on direction profile [6], [7], surroundness features [8], grid based methods [9], methods based on geometrical moments [10], and texture based features [11] have also become famous in signature verification task. Few structural methods that consider the relations among local features are also explored for the same task. Examples include graph matching [12] and recently proposed compact correlated features [13]. On the other hand, Siamese like networks are very popular for different verification tasks, such as, online signature verification [14], face verification [15], [16] etc. Furthermore, it has also been used for one-shot image recognition [17], as well as for sketch-based image retrieval task [18].

# Chapter 3



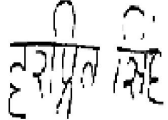
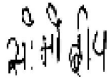
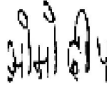

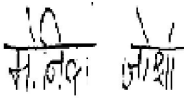
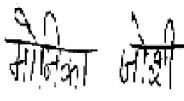
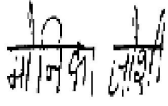
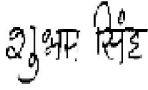
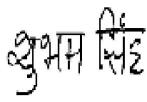
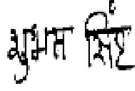
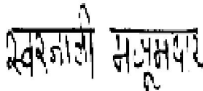
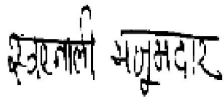
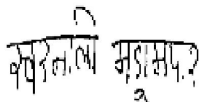
## Datasets

In the field of signature verification, there is a sparseness of publicly available signature datasets. The quality of the available datasets also varies, as there has been no standard data collection protocol for creation of datasets. Besides, it is very costly to create a large corpus with different types of forgeries, especially skilled forgeries. Two off-line signature databases, which are widely used in the literature, are GPDS-960 [19] and MCYT [20]. As there has been no public signature corpus available for Bangla and Hindi script, it was necessary to create a dataset of off-line Bangla and Hindi signatures. This Bangla and Hindi signature (BHSig260) dataset consists of 260 sets of handwritten off-line signatures of which 100 sets were written in Bangla script for the Bangla part and the rest (160 sets) were written in Hindi script for the Hindi part of the BHSig260 dataset. The handwritten off-line signatures were collected from 260 different individuals with different educational backgrounds and ages. Each set consists of 24 genuine signatures and 30 skilled forgeries. Signatures were collected during 2 different sessions. In the first session, the genuine signatures were collected, whereas in the second session the skilled forgeries were collected, showing a genuine signature to an individual to train and mimic the forgeries. A total number of 6240 genuine and 7800 skilled forgery signatures were collected from all 260 individuals. The collected data was acquired using a Flatbed scanner with a resolution of 300DPI in grey scale and stored in TIFF format (Tagged Image File Format). A histogram-based threshold technique was applied for binarization to convert digitized grey-level images to twotone images. The skilled forgery signatures collected are quite similar to the genuine signatures, which makes the dataset quite a challenging one. To illustrate the complexity of the forged signatures, some binary genuine signature samples of the BHSig260 dataset, with their corresponding forgeries, are displayed in Figure 2.3. The BHSig260 dataset introduced by S.Pal et.al.,[21] is publicly available for research purposes.

Figure 3.1: Genuine and Forged signature image samples of Bengali dataset.

Bengali		
Genuine Copy	Genuine Copy	Forged Copy
Genuine Copy	Genuine Copy	Forged Copy
Genuine Copy	Genuine Copy	Forged Copy
Genuine Copy	Genuine Copy	Forged Copy
Genuine Copy	Genuine Copy	Forged Copy

Figure 3.2: Genuine and Forged signature image samples of Hindi dataset.

Hindi		
Genuine Copy	Genuine Copy	Forged Copy
		
Genuine Copy	Genuine Copy	Forged Copy
		
Genuine Copy	Genuine Copy	Forged Copy
		
Genuine Copy	Genuine Copy	Forged Copy
		
Genuine Copy	Genuine Copy	Forged Copy
		

# Chapter 4

## Methodology

### 4.1 CNN and Siamese Network

Deep Convolutional Neural Networks (CNN) are multilayer neural networks consists of several convolutional layers with different kernel sizes interleaved by pooling layers, which summarizes and down samples the output of its convolutions before feeding to next layers. To get nonlinearity rectified linear units are also used. In this work, we used different convolutional kernels with sizes starting with  $11 \times 11$  to  $3 \times 3$ . Generally a differentiable loss function is chosen so that Gradient descent can be applied and the network weights can be optimized. Given a differentiable loss function, the weights of different layers are updated using back propagation. As the optimization cannot be applied to all training data where training size is large batch optimizations gives a fair alternative to optimize the network.

Siamese neural network is a class of network architectures that usually contains two identical subnetworks. The twin CNNs have the same configuration with the same parameters and shared weights. The parameter updating is mirrored across both the subnetworks.

### 4.2 Loss Function

This framework has been successfully used for dimensionality reduction in weakly supervised metric learning and for face verification. These subnetworks are joined by a loss function at the top, which computes a similarity metric involving the Euclidean distance between the feature representation on each side of the Siamese network. One such loss function that is mostly used in Siamese network is the contrastive loss defined as follows:

$$L(s_1, s_2, y) = (1 - y)D_w^2 + y \max(0, m - D_w)^2 \quad (4.1)$$

where  $s_1$  and  $s_2$  are two samples (here signature images),  $y$  is a binary indicator function denoting whether the two samples belong to the same class or not, and  $m$  are two constants and  $m$  is the margin equal to 1 in our case.  $D_w = ||f(s_1; w_1) - f(s_2; w_1)||$

is the Euclidean distance computed in the embedded feature space,  $f$  is an embedding function that maps a signature image to real vector space through CNN, and  $w_1, w_2$  are the learned weights for a particular layer of the underlying network. Unlike conventional approaches that assign binary similarity labels to pairs, Siamese network aims to bring the output feature vectors closer for input pairs that are labelled as similar, and push the feature vectors away if the input pairs are dissimilar. Each of the branches of the Siamese network can be seen as a function that embeds the input image into a space. Due to the loss function selected (Eqn.4.1), this space will have the property that images of the same class (genuine signature for a given writer) will be closer to each other than images of different classes (forgeries or signatures of different writers). Both branches are joined together by a layer that computes the Euclidean distance between the two points in the embedded space. Then, in order to decide if two images belong to the similar class (genuine, genuine) or a dissimilar class (genuine, forged) one needs to determine a threshold value on the distance.

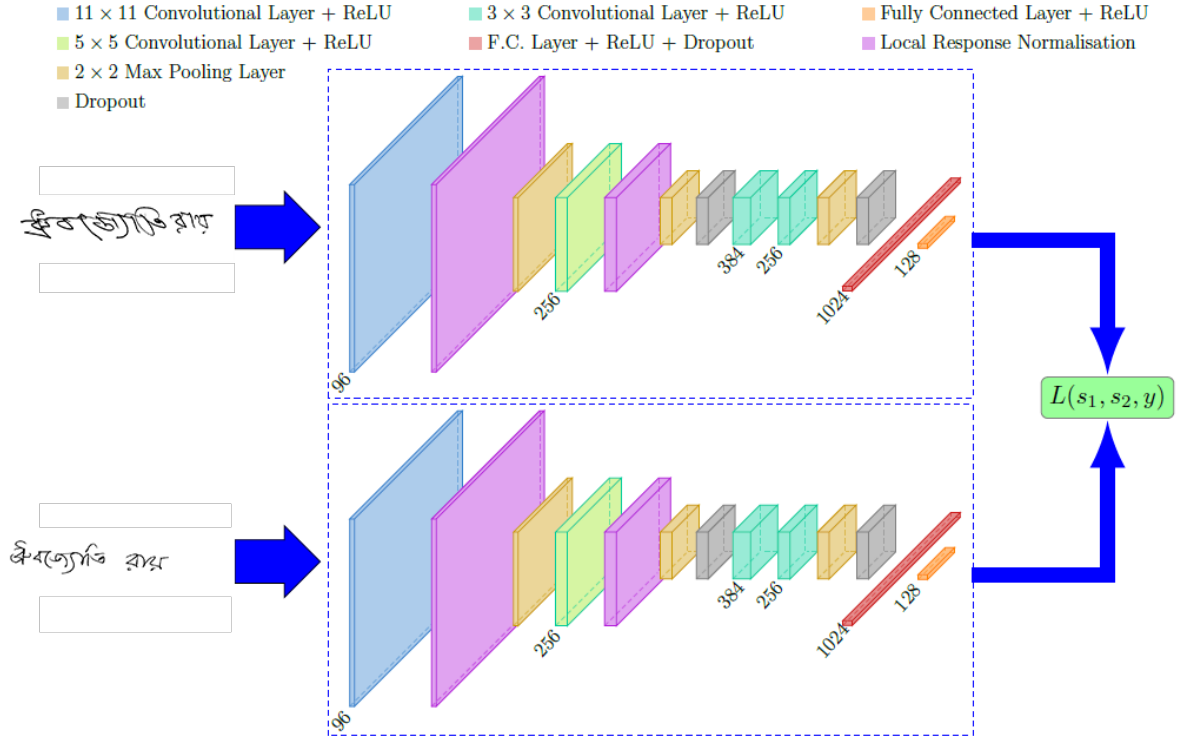
### 4.3 Architecture

We have used a CNN architecture that is inspired by Krizhevsky et al. [19] for an image recognition problem. For the easy reproducibility of our results, we present a full list of parameters used to design the CNN layers in Table 1. For convolution and pooling layers, we list the size of the filters as  $N \times H \times W$ , here  $N$  is the number of filters,  $H$  is the height and  $W$  is the width of the corresponding filter. Here, stride signifies the distance between the application of filters for the convolution and pooling operations, and pad indicates the width of added borders to the input. Here it is to be mentioned that padding is necessary in order to convolve the filter from the very first pixel in the input image. Throughout the network, we use Rectified Linear Units (ReLU) as the activation function to the output of all the convolutional and fully connected layers. For generalizing the learned features, Local Response Normalization is applied according to [19], with the parameters shown in the corresponding row in Table 4.1. With the last two pooling layers and the first fully connected layer, we use a Dropout with a rate equal to 0.3 and 0.5, respectively. The first convolutional layers filter the  $155 \times 220$  input signature image with 96 kernels of size  $11 \times 11$  with a stride of 2 pixels. The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size  $5 \times 5$ . The third and fourth convolutional layers are connected to one another without any intervention of pooling or normalization of layers. The third layer has 384 kernels of size  $3 \times 3$  connected to the (normalized, pooled, and dropout) output of the second convolutional layer. The fourth convolutional layer has 256 kernels of size  $3 \times 3$ . This leads to the neural network learning fewer lower level features for smaller receptive fields and more features for higher level or more abstract features. The first fully connected layer has 1024 neurons, whereas the second fully connected layer



has 128 neurons. This indicates that the highest learned feature vector from each side of the architecture has a dimension equal to 128. We initialize the weights of the model

Figure 4.1: Common Architecture of the model used for training Bengali and Hindi Dataset.



according to the work of Glorot and Bengio [20], and the biases equal to 0. We trained the model using RMSprop for 15 epochs, using momentum rate equal to 0.9, and mini batch size equal to 128. We started with an initial learning rate (LR) equal to  $1e-4$  with hyper parameters = 0.9 and  $\gamma = 1e-8$ .

Table 4.1: Training Hyper-parameters

Parameter	Value
Initial Learning Rate	$1e-4$
Learning Rate Schedule	$LR \leftarrow LR \times 0.1$
Weight Decay	0.0005
Momentum	0.9
Fuzz Factor	$1e-8$
Batch Size	128

## 4.4 Hardware and Software Requirements

Our entire framework is implemented using Keras library with the TensorFlow as backend. The training was done using Jupyter Notebook and CPU, and it took about 12 hours approximately, depending on different databases.

# Chapter 5

## Analysis and Results

### 5.1 Analysis of Signature Schemes

The analysis of a signature scheme involves the evaluation of 2 parameters: FAR FRR.

**FAR – False Acceptance Ratio-** The false acceptance ratio is given by the number of fake signatures accepted by the system with respect to the total number of comparisons made.

**FRR – False Rejection Ratio-** The false rejection ratio is the total number of genuine signatures rejected by the system with respect to the total number of comparisons made. Both FAR and FRR depend on the threshold variance parameter taken to decide the genuineness of an image. If we choose a high threshold variance then the FRR is reduced, but at the same time the FAR also increases. If we choose a low threshold variance then the FAR is reduced, but at the same time the FRR also increases.

**ERR – Error rejection Rate-** If the FAR of a system is same as the FRR then the system is said to be in an optimal state. In this condition, the FAR and FRR are also known as ERR.

### 5.2 Result and Discussion

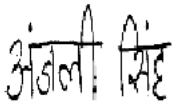
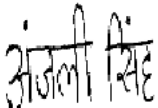

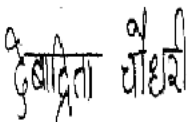
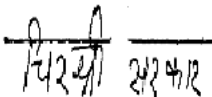
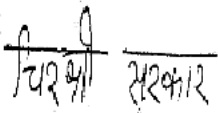
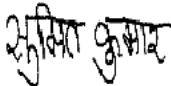
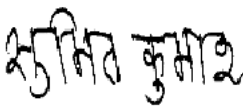
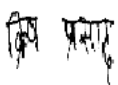
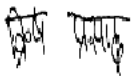
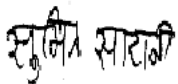
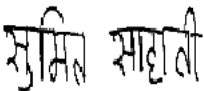
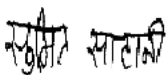
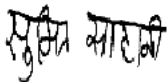
Table 5.1: Result of our framework obtained on Bengali and Hindi Dataset

Database	Signers	Accuracy	Threshold	FAR	FRR
Bengali	100	91.82%	0.61	12.80	11.80
Hindi	160	84%	0.34	13.10	15.05

Table 5.2: Bengali Signature Verification Left: Original; Right: Predicted

Genuine	Forged
	
Genuine	Forged
	
Genuine	Genuine
	
Genuine	Forged
	
Genuine	Genuine
	
Genuine	Genuine
	
Genuine	Forged
	

Table 5.3: **Hindi Signature Verification** Left: Original; Right: Predicted

Genuine	Genuine
	
Genuine	Genuine
	
Genuine	Genuine
	
Genuine	Forged
	
Genuine	Genuine
	
Genuine	Forged
	
Genuine	Genuine
	

# Chapter 6

## Conclusion and Future Work

### 6.1 Conclusion

In this project, we have presented a framework based on Siamese network for offline signature verification, which uses writer independent feature learning. This method does not rely on hand-crafted features unlike its predecessors, instead it learns them from data in a writer independent scenario. Our experiments made on cross domain datasets emphasize how well our architecture models the fraudulence of different handwriting style of different signers and forgers with diverse background and scripts.

### 6.2 Future Work

In future, more regional languages such as Tamil, Telugu, Malayalam etc can be implemented for signature verification. Training data having complex writing style can be used to develop a more enriched network for better verification. Better network can be designed to reduce training time. Efficiency of the current model can be improved by tweaking parameters and formulating a better loss function.

# Bibliography

- [1] R. Plamondon, S. Srihari, Online and o-line handwriting recognition: a comprehensive survey, *IEEE TPAMI* 22 (1) (2000) 63–84.
- [2] D. Impedovo, G. Pirlo, Automatic signature verification: The state of the art, *IEEE TSMC* 38 (5) (2008) 609–635.
- [3] M. E. Munich, P. Perona, Visual identification by signature tracking, *IEEE TPAMI* 25 (2) (2003) 200–217.
- [4] D. Bertolini, L. Oliveira, E. Justino, R. Sabourin, Reducing forgeries in writer-independent o-line signature verification through ensemble of classifiers, *PR* 43 (1) (2010) 387–396.
- [5] M. K. Kalera, S. N. Srihari, A. Xu, Oine signature verification and identification using distance statistics, *IJPRAI* 18 (7) (2004) 1339–1360.
- [6] G. Dimauro, S. Impedovo, G. Pirlo, A. Salzo, A multi-expert signature verification system for bankcheck processing, *IJPRAI* 11 (05) (1997) 827–844.
- [7] M. A. Ferrer, J. B. Alonso, C. M. Travieso, Oine geometric parameters for automatic signature verification using fixed-point arithmetic, *IEEE TPAMI* 27 (6) (2005) 993–997.
- [8] R. Kumar, J. Sharma, B. Chanda, Writer-independent o-line signature verification using surroundedness feature, *PRL* 33 (3) (2012) 301–308.
- [9] K. Huang, H. Yan, O-line signature verification based on geometric feature extraction and neural network classification, *PR* 30 (1) (1997) 9–17.
- [10] V. Ramesh, M. N. Murty, O-line signature verification using genetically optimized weighted features, *PR* 32 (2) (1999) 217–233.
- [11] S. Pal, A. Alaei, U. Pal, M. Blumenstein, Performance of an o-line signature verification method based on texture features on a large indic-script signature dataset, in: *DAS*, 2016, pp. 72–77.

- [12] S. Chen, S. Srihari, A new o-line signature verification method based on graph, in: ICPR, 2006, pp. 869–872.
- [13] A. Dutta, U. Pal, J. Lladós, Compact correlated features for writer independent signature verification, in: ICPR, 2016, pp. 3411–3416.
- [14] J. Bromley, I. Guyon, Y. LeCun, E. Sackinger, R. Shah, Signature verification using a "siamese" time delay neural network, in: NIPS, 1994, pp.737–744.
- [15] S. Chopra, R. Hadsell, Y. LeCun, Learning a similarity metric discriminatively, with application to face verification, in: CVPR, 2005, pp. 539–546.
- [16] F. Schro, D. Kalenichenko, J. Philbin, Facenet: A unified embedding for face recognition and clustering, in: CVPR, 2015, pp. 815–823.
- [17] G. Koch, R. Zemel, R. Salakhutdinov, Siamese neural networks for oneshot image recognition, in: ICML, 2015, pp. 1–8.
- [18] Y. Qi, Y. Z. Song, H. Zhang, J. Liu, Sketch-based image retrieval via siamese convolutional neural network, in: ICIP, 2016, pp. 2460–2464.
- [19] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: NIPS, 2012, pp. 1097–1105.
- [20] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: AISTATS, 2010, pp. 249–256.

```

In [1]: import sys
import numpy as np
import pickle
import os
import matplotlib.pyplot as plt
%matplotlib inline

import cv2
import time
import itertools
import random

from sklearn.utils import shuffle
from tensorflow.keras.models import load_model
import tensorflow as tf
from keras.models import Sequential
from keras.optimizers import Adam, RMSprop
from keras.layers import Conv2D, ZeroPadding2D, Activation, Input, concatenate, [
from keras.models import Model

from keras.layers.normalization import BatchNormalization
from keras.layers.pooling import MaxPooling2D
from keras.layers.merge import Concatenate
from keras.layers.core import Lambda, Flatten, Dense
from keras.initializers import glorot_uniform

from keras.engine.topology import Layer
from keras.regularizers import l2
from keras import backend as K
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

```

Using TensorFlow backend.

```

In [2]: path = "C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Ne

```

```

In [3]: # Get the list of all directories and sort them
dir_list = next(os.walk(path))[1]
dir_list.sort()

```

```

In [4]: # For each person segregate the genuine signatures from the forged signatures
# Genuine signatures are stored in the list "orig_groups"
# Forged signatures are stored in the list "forged_groups"
orig_groups, forg_groups = [], []
for directory in dir_list:
    images = os.listdir(path+directory)
    images.sort()
    images = [path+directory+'/'+x for x in images]
    forg_groups.append(images[:30]) # First 30 signatures in each folder are forged
    orig_groups.append(images[30:]) # Next 24 signatures are genuine

```





```
In [12]: def visualize_sample_signature():
'''Function to randomly select a signature from train set and
print two genuine copies and one forged copy'''
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (10, 10))
k = np.random.randint(len(orig_train))
orig_img_names = random.sample(orig_train[k], 2)
forg_img_name = random.sample(forg_train[k], 1)
orig_img1 = cv2.imread(orig_img_names[0], 0)
orig_img2 = cv2.imread(orig_img_names[1], 0)
forg_img = plt.imread(forg_img_name[0], 0)
orig_img1 = cv2.resize(orig_img1, (img_w, img_h))
orig_img2 = cv2.resize(orig_img2, (img_w, img_h))
forg_img = cv2.resize(forg_img, (img_w, img_h))

ax1.imshow(orig_img1, cmap = 'gray')
ax2.imshow(orig_img2, cmap = 'gray')
ax3.imshow(forg_img, cmap = 'gray')

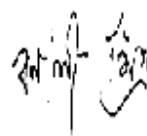
ax1.set_title('Genuine Copy')
ax1.axis('off')
ax2.set_title('Genuine Copy')
ax2.axis('off')
ax3.set_title('Forged Copy')
ax3.axis('off')
```

```
In [13]: visualize_sample_signature()
```

Genuine Copy



Genuine Copy

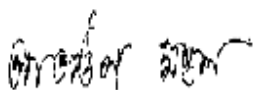


Forged Copy

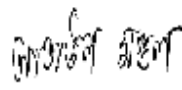


```
In [14]: visualize_sample_signature()
```

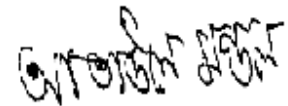
Genuine Copy



Genuine Copy

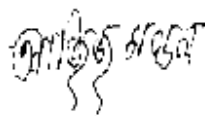


Forged Copy

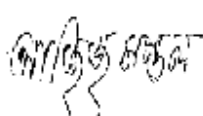


In [15]: visualize\_sample\_signature()

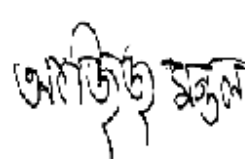
Genuine Copy



Genuine Copy

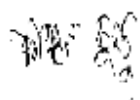


Forged Copy

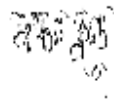


In [16]: visualize\_sample\_signature()

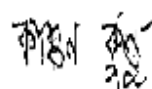
Genuine Copy



Genuine Copy

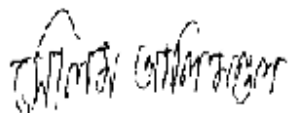


Forged Copy

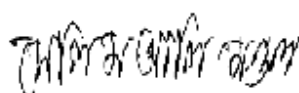


In [17]: visualize\_sample\_signature()

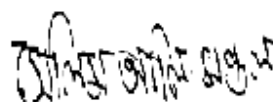
Genuine Copy



Genuine Copy



Forged Copy

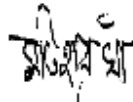


In [18]: visualize\_sample\_signature()

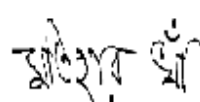
Genuine Copy



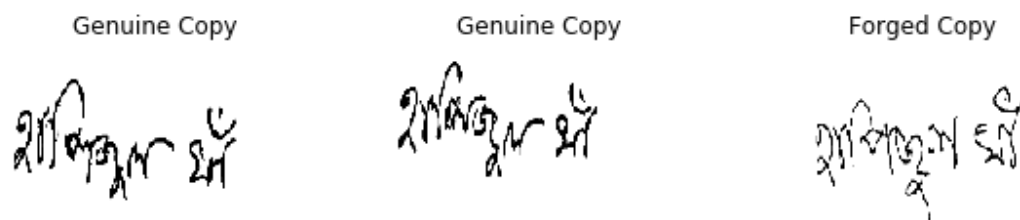
Genuine Copy



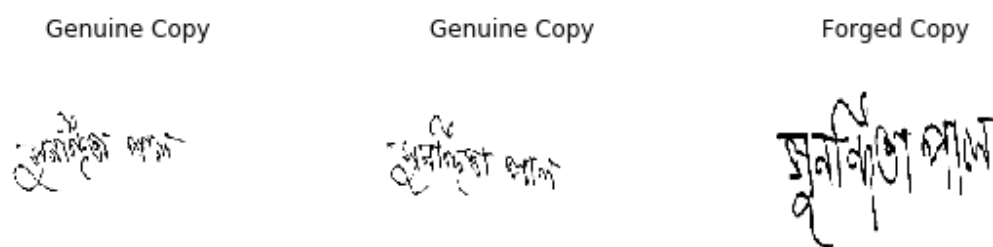
Forged Copy



In [19]: visualize\_sample\_signature()



In [20]: visualize\_sample\_signature()



In [21]: visualize\_sample\_signature()



```

In [22]: def generate_batch(orig_groups, forg_groups, batch_size = 32):
'''Function to generate a batch of data with batch_size number of data points
Half of the data points will be Genuine-Genuine pairs and half will be Genuine-Forged pairs'''
while True:
    orig_pairs = []
    forg_pairs = []
    gen_gen_labels = []
    gen_for_labels = []
    all_pairs = []
    all_labels = []

    # Here we create pairs of Genuine-Genuine image names and Genuine-Forged
    # For every person we have 24 genuine signatures, hence we have
    # 24 choose 2 = 276 Genuine-Genuine image pairs for one person.
    # To make Genuine-Forged pairs, we pair every Genuine signature of a person
    # with 12 randomly sampled Forged signatures of the same person.
    # Thus we make 24 * 12 = 300 Genuine-Forged image pairs for one person.
    # In all we have 120 person's data in the training data.
    # Total no. of Genuine-Genuine pairs = 120 * 276 = 33120
    # Total number of Genuine-Forged pairs = 120 * 300 = 36000
    # Total no. of data points = 33120 + 36000 = 69120
    for orig, forg in zip(orig_groups, forg_groups):
        orig_pairs.extend(list(itertools.combinations(orig, 2)))
        for i in range(len(forg)):
            forg_pairs.extend(list(itertools.product(orig[i:i+1], random.sample(forg, 12))))

    # Label for Genuine-Genuine pairs is 1
    # Label for Genuine-Forged pairs is 0
    gen_gen_labels = [1]*len(orig_pairs)
    gen_for_labels = [0]*len(forg_pairs)

    # Concatenate all the pairs together along with their labels and shuffle
    all_pairs = orig_pairs + forg_pairs
    all_labels = gen_gen_labels + gen_for_labels
    del orig_pairs, forg_pairs, gen_gen_labels, gen_for_labels
    all_pairs, all_labels = shuffle(all_pairs, all_labels)

    # Note the Lists above contain only the image names and
    # actual images are loaded and yielded below in batches
    # Below we prepare a batch of data points and yield the batch
    # In each batch we load "batch_size" number of image pairs
    # These images are then removed from the original set so that
    # they are not added again in the next batch.

    k = 0
    pairs=[np.zeros((batch_size, img_h, img_w, 1)) for i in range(2)]
    targets=np.zeros((batch_size,))
    for ix, pair in enumerate(all_pairs):
        img1 = cv2.imread(pair[0], 0)
        img2 = cv2.imread(pair[1], 0)
        img1 = cv2.resize(img1, (img_w, img_h))
        img2 = cv2.resize(img2, (img_w, img_h))
        img1 = np.array(img1, dtype = np.float64)
        img2 = np.array(img2, dtype = np.float64)
        img1 /= 255
        img2 /= 255

```

```

img1 = img1[..., np.newaxis]
img2 = img2[..., np.newaxis]
pairs[0][k, :, :, :] = img1
pairs[1][k, :, :, :] = img2
targets[k] = all_labels[ix]
k += 1
if k == batch_size:
    yield pairs, targets
    k = 0
pairs=[np.zeros((batch_size, img_h, img_w, 1)) for i in range(2)]
targets=np.zeros((batch_size,))

```

```

In [23]: def euclidean_distance(vects):
    '''Compute Euclidean Distance between two vectors'''
    x, y = vects
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))

```

```

In [24]: def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)

```

```

In [25]: def contrastive_loss(y_true, y_pred):
    '''Contrastive loss from Hadsell-et-al.'06
    http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf
    '''
    margin = 1
    return K.mean(y_true * K.square(y_pred) + (1 - y_true) * K.square(K.maximum(n

```

```
In [26]: def create_base_network_signet(input_shape):
'''Base Siamese Network'''

seq = Sequential()
seq.add(Conv2D(96, kernel_size=(11, 11), activation='relu', name='conv1_1', s
          init='glorot_uniform', dim_ordering='tf'))
seq.add(BatchNormalization(epsilon=1e-06, mode=0, axis=1, momentum=0.9))
seq.add(MaxPooling2D((3,3), strides=(2, 2)))
seq.add(ZeroPadding2D((2, 2), dim_ordering='tf'))

seq.add(Conv2D(256, kernel_size=(5, 5), activation='relu', name='conv2_1', st
seq.add(BatchNormalization(epsilon=1e-06, mode=0, axis=1, momentum=0.9))
seq.add(MaxPooling2D((3,3), strides=(2, 2)))
seq.add(Dropout(0.3))# added extra
seq.add(ZeroPadding2D((1, 1), dim_ordering='tf'))

seq.add(Conv2D(384, kernel_size=(3, 3), activation='relu', name='conv3_1', st
seq.add(ZeroPadding2D((1, 1), dim_ordering='tf'))

seq.add(Conv2D(256, kernel_size=(3, 3), activation='relu', name='conv3_2', st
seq.add(MaxPooling2D((3,3), strides=(2, 2)))
seq.add(Dropout(0.3))# added extra
seq.add(Flatten(name='flatten'))
seq.add(Dense(1024, W_regularizer=l2(0.0005), activation='relu', init='glorot
seq.add(Dropout(0.5))

seq.add(Dense(128, W_regularizer=l2(0.0005), activation='relu', init='glorot_

return seq
```

```
In [27]: input_shape=(img_h, img_w, 1)
```

```
In [28]: import tensorflow as tf
import keras.backend.tensorflow_backend as tfback

print("tf.__version__ is", tf.__version__)
print("tf.keras.__version__ is:", tf.keras.__version__)

def _get_available_gpus():
    """Get a list of available gpu devices (formatted as strings).

    # Returns
        A list of available GPU devices.
    """
    #global _LOCAL_DEVICES
    if tfback._LOCAL_DEVICES is None:
        devices = tf.config.list_logical_devices()
        tfback._LOCAL_DEVICES = [x.name for x in devices]
    return [x for x in tfback._LOCAL_DEVICES if 'device:gpu' in x.lower()]

tfback._get_available_gpus = _get_available_gpus

tf.__version__ is 2.1.0
tf.keras.__version__ is: 2.2.4-tf
```



```
In [29]: # network definition
base_network = create_base_network_sigmet(input_shape)

input_a = Input(shape=(input_shape))
input_b = Input(shape=(input_shape))

# because we re-use the same instance `base_network`,
# the weights of the network
# will be shared across the two branches
processed_a = base_network(input_a)
processed_b = base_network(input_b)

# Compute the Euclidean distance between the two vectors in the latent space
distance = Lambda(euclidean_distance, output_shape=eucl_dist_output_shape)([processed_a, processed_b])

model = Model(input=[input_a, input_b], output=distance)
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:6: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(96, kernel_size=(11, 11), activation="relu", name="conv1_1", strides=4, input_shape=(155, 220,..., data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:7: UserWarning: Update your `BatchNormalization` call to the Keras 2 API: `BatchNormalization(epsilon=1e-06, axis=1, momentum=0.9)`
```

```
import sys
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:9: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((2, 2), data_format="channels_last")`
```

```
if __name__ == '__main__':
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:11: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(5, 5), activation="relu", name="conv2_1", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
# This is added back by InteractiveShellApp.init_path()
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:12: UserWarning: Update your `BatchNormalization` call to the Keras 2 API: `BatchNormalization(epsilon=1e-06, axis=1, momentum=0.9)`
```

```
if sys.path[0] == '':
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:15: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((1, 1), data_format="channels_last")`
```

```
from ipykernel import kernelapp as app
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:17: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(384, kernel_size=(3, 3), activation="relu", name="conv3_1", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:18: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((1, 1), data_format="channels_last")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:20: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(3, 3), activation="relu", name="conv3_2", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:21: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(3, 3), activation="relu", name="conv3_2", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
er.py:24: UserWarning: Update your `Dense` call to the Keras 2 API: `Dense(1
024, activation="relu", kernel_initializer="glorot_uniform", kernel_regulariz
er=<keras.reg...)`
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launch
er.py:27: UserWarning: Update your `Dense` call to the Keras 2 API: `Dense(1
28, activation="relu", kernel_initializer="glorot_uniform", kernel_regulariz
er=<keras.reg...)`
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launch
er.py:16: UserWarning: Update your `Model` call to the Keras 2 API: `Model(i
nputs=[<tf.Tenso..., outputs=Tensor("la...)`
app.launch_new_instance()
```

```
In [31]: batch_sz = 128
num_train_samples = 276*80 + 300*80
num_val_samples = num_test_samples = 276*20 + 300*20
num_train_samples, num_val_samples, num_test_samples
```

```
Out[31]: (46080, 11520, 11520)
```

```
In [32]: # compile model using RMSProp Optimizer and Contrastive Loss function defined above
rms = RMSprop(lr=1e-4, rho=0.9, epsilon=1e-08)
model.compile(loss=contrastive_loss, optimizer=rms)
```

```
In [33]: # Using Keras Callbacks, save the model after every epoch
# Reduce the Learning rate by a factor of 0.1 if the validation loss does not improve
# Stop the training using early stopping if the validation loss does not improve
callbacks = [
    EarlyStopping(patience=12, verbose=1),
    ReduceLRonPlateau(factor=0.1, patience=5, min_lr=0.000001, verbose=1),
    ModelCheckpoint('C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-master/model_128.h5', save_best_only=True, monitor='val_loss', verbose=1)
]
```

```
In [34]: results = model.fit_generator(generate_batch(orig_train, forg_train, batch_sz),
                                       steps_per_epoch = num_train_samples//batch_sz,
                                       epochs = 100,
                                       validation_data = generate_batch(orig_val, forg_val),
                                       validation_steps = num_val_samples//batch_sz,
                                       callbacks = callbacks)
```

Epoch 00003: saving model to C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-master/weights\_bengali/signet-bhsig260-003.h5

Epoch 4/100

360/360 [=====] - 2564s 7s/step - loss: 0.0826 - val\_loss: 0.1767

Epoch 00004: saving model to C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-master/weights\_bengali/signet-bhsig260-004.h5

Epoch 5/100

360/360 [=====] - 2817s 8s/step - loss: 0.0400 - val\_loss: 0.0995

Epoch 00005: saving model to C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-master/weights\_bengali/signet-bhsig260-005.h5

Epoch 6/100

360/360 [=====] - 2647s 7s/step - loss: 0.0260 - val\_loss: 0.1841

**After observing continuous fall in validation accuracy, I interrupted the training manually**

```
In [35]: def compute_accuracy_roc(predictions, labels):
'''Compute ROC accuracy with a range of thresholds on distances.
...
dmax = np.max(predictions)
dmin = np.min(predictions)
nsame = np.sum(labels == 1)
ndiff = np.sum(labels == 0)

step = 0.01
max_acc = 0
best_thresh = -1

for d in np.arange(dmin, dmax+step, step):
    idx1 = predictions.ravel() <= d
    idx2 = predictions.ravel() > d

    tpr = float(np.sum(labels[idx1] == 1)) / nsame
    tnr = float(np.sum(labels[idx2] == 0)) / ndiff
    acc = 0.5 * (tpr + tnr)
#    print ('ROC', acc, tpr, tnr)

    if (acc > max_acc):
        max_acc, best_thresh = acc, d

return max_acc, best_thresh
```

**Load the weights from the epoch which gave the best validation accuracy**

```
In [36]: using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-
```

```
In [37]: test_gen = generate_batch(orig_test, forg_test, 1)
pred, tr_y = [], []
for i in range(num_test_samples):
    (img1, img2), label = next(test_gen)
    tr_y.append(label)
    pred.append(model.predict([img1, img2])[0][0])
```

```
In [38]: tr_acc, threshold = compute_accuracy_roc(np.array(pred), np.array(tr_y))
tr_acc, threshold
```

```
Out[38]: (0.9182139488232696, 0.61)
```

**#### Accuracy = 91.82% and Threshold = 0.61**

Thus if the difference score is less than 0.61, we predict the test image as Genuine and if the difference score is greater than 0.61, we predict it to be as forged

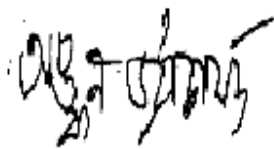
**#### Below we see some sample results**

```
In [84]: def predict_score():
'''Predict distance score and classify test images as Genuine or Forged'''
test_point, test_label = next(test_gen)
img1, img2 = test_point[0], test_point[1]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 10))
ax1.imshow(np.squeeze(img1), cmap='gray')
ax2.imshow(np.squeeze(img2), cmap='gray')
ax1.set_title('Genuine')
if test_label == 1:
    ax2.set_title('Genuine')
else:
    ax2.set_title('Forged')
ax1.axis('off')
ax2.axis('off')
plt.show()
result = model.predict([img1, img2])
diff = result[0][0]
print("Difference Score = ", diff)
if diff > threshold:
    print("Its a Forged Signature")
else:
    print("Its a Genuine Signature")
```

```
In [40]: predict_score()
```

Genuine



Genuine



Difference Score = 0.16581872  
Its a Genuine Signature

**Note: The first image is always Genuine. Score prediction and classification is done for the second image**

```
In [41]: predict_score()
```

Genuine



Forged



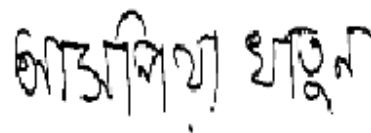
Difference Score = 1.6177862  
Its a Forged Signature

```
In [42]: predict_score()
```

Genuine



Forged

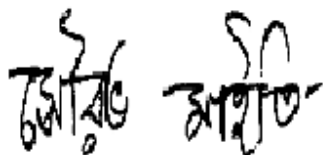
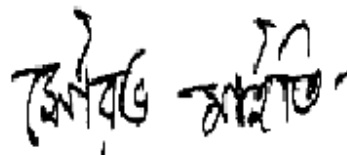


Difference Score = 1.1082009  
Its a Forged Signature

In [43]: `predict_score()`

Genuine

Genuine

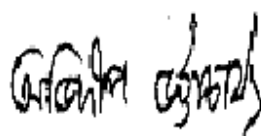
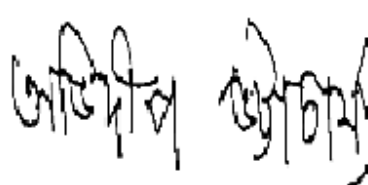



Difference Score = 0.70687795  
Its a Forged Signature

In [44]: `predict_score()`

Genuine

Forged

Difference Score = 1.1100439  
Its a Forged Signature

```

In [35]: import sys
import numpy as np
import pickle
import os
import matplotlib.pyplot as plt
%matplotlib inline

import cv2
import time
import itertools
import random

from sklearn.utils import shuffle
from tensorflow.keras.models import load_model
import tensorflow as tf
from keras.models import Sequential
from keras.optimizers import Adam, RMSprop
from keras.layers import Conv2D, ZeroPadding2D, Activation, Input, concatenate, [
#from keras.models import Model

from keras.layers.normalization import BatchNormalization
from keras.layers.pooling import MaxPooling2D
from keras.layers.merge import Concatenate
from keras.layers.core import Lambda, Flatten, Dense
from keras.initializers import glorot_uniform

from keras.engine.topology import Layer
from keras.regularizers import l2
from keras import backend as K
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

```

```

In [36]: path = "C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Ne

```

```

In [37]: # Get the list of all directories and sort them
dir_list = next(os.walk(path))[1]
dir_list.sort()

```

```

In [38]: # For each person segregate the genuine signatures from the forged signatures
# Genuine signatures are stored in the list "orig_groups"
# Forged signatures are stored in the list "forged_groups"
orig_groups, forg_groups = [], []
for directory in dir_list:
    images = os.listdir(path+directory)
    images.sort()
    images = [path+directory+'/'+x for x in images]
    forg_groups.append(images[:30]) # First 30 signatures in each folder are forged
    orig_groups.append(images[30:]) # Next 24 signatures are genuine

```





```
In [46]: def visualize_sample_signature():
'''Function to randomly select a signature from train set and
print two genuine copies and one forged copy'''
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize = (10, 10))
k = np.random.randint(len(orig_train))
orig_img_names = random.sample(orig_train[k], 2)
forg_img_name = random.sample(forg_train[k], 1)
orig_img1 = cv2.imread(orig_img_names[0], 0)
orig_img2 = cv2.imread(orig_img_names[1], 0)
forg_img = plt.imread(forg_img_name[0], 0)
orig_img1 = cv2.resize(orig_img1, (img_w, img_h))
orig_img2 = cv2.resize(orig_img2, (img_w, img_h))
forg_img = cv2.resize(forg_img, (img_w, img_h))

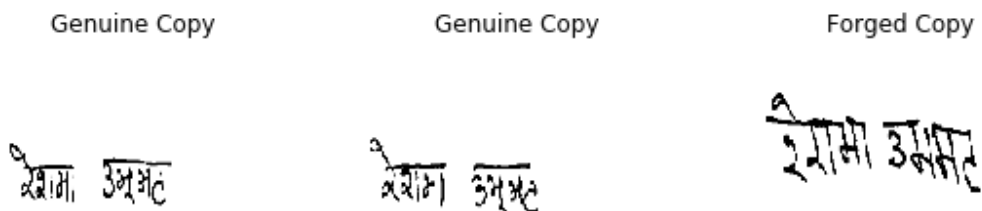
ax1.imshow(orig_img1, cmap = 'gray')
ax2.imshow(orig_img2, cmap = 'gray')
ax3.imshow(forg_img, cmap = 'gray')

ax1.set_title('Genuine Copy')
ax1.axis('off')
ax2.set_title('Genuine Copy')
ax2.axis('off')
ax3.set_title('Forged Copy')
ax3.axis('off')
```

```
In [13]: visualize_sample_signature()
```

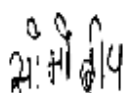


```
In [14]: visualize_sample_signature()
```

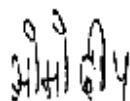


In [15]: visualize\_sample\_signature()

Genuine Copy



Genuine Copy

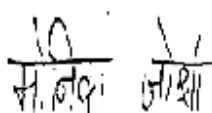


Forged Copy

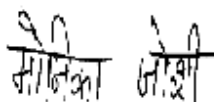


In [16]: visualize\_sample\_signature()

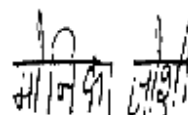
Genuine Copy



Genuine Copy

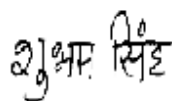


Forged Copy

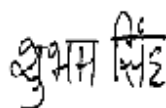


In [17]: visualize\_sample\_signature()

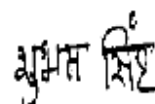
Genuine Copy



Genuine Copy

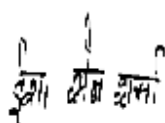


Forged Copy

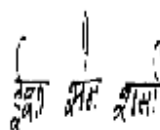


In [21]: visualize\_sample\_signature()

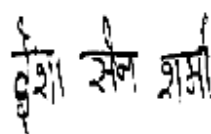
Genuine Copy



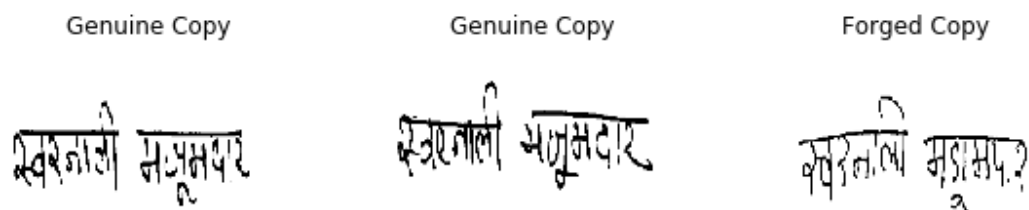
Genuine Copy



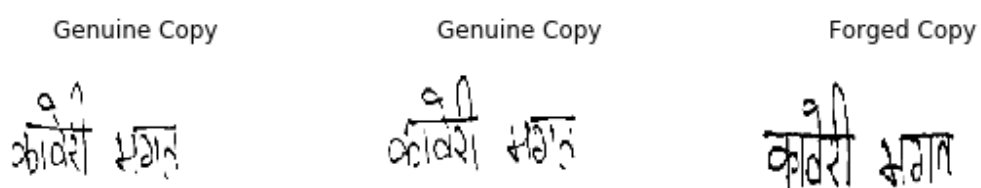
Forged Copy



In [22]: visualize\_sample\_signature()



In [23]: visualize\_sample\_signature()



In [24]: visualize\_sample\_signature()



In [ ]:

In [ ]:

```

In [47]: def generate_batch(orig_groups, forg_groups, batch_size = 32):
    '''Function to generate a batch of data with batch_size number of data points
    Half of the data points will be Genuine-Genuine pairs and half will be Genuine-Forged pairs
    while True:
        orig_pairs = []
        forg_pairs = []
        gen_gen_labels = []
        gen_for_labels = []
        all_pairs = []
        all_labels = []

        # Here we create pairs of Genuine-Genuine image names and Genuine-Forged
        # For every person we have 24 genuine signatures, hence we have
        # 24 choose 2 = 276 Genuine-Genuine image pairs for one person.
        # To make Genuine-Forged pairs, we pair every Genuine signature of a person
        # with 12 randomly sampled Forged signatures of the same person.
        # Thus we make 24 * 12 = 300 Genuine-Forged image pairs for one person.
        # In all we have 120 person's data in the training data.
        # Total no. of Genuine-Genuine pairs = 120 * 276 = 33120
        # Total number of Genuine-Forged pairs = 120 * 300 = 36000
        # Total no. of data points = 33120 + 36000 = 69120
        for orig, forg in zip(orig_groups, forg_groups):
            orig_pairs.extend(list(itertools.combinations(orig, 2)))
            for i in range(len(forg)):
                forg_pairs.extend(list(itertools.product(orig[i:i+1], random.sample(forg, 12))))

        # Label for Genuine-Genuine pairs is 1
        # Label for Genuine-Forged pairs is 0
        gen_gen_labels = [1]*len(orig_pairs)
        gen_for_labels = [0]*len(forg_pairs)

        # Concatenate all the pairs together along with their labels and shuffle
        all_pairs = orig_pairs + forg_pairs
        all_labels = gen_gen_labels + gen_for_labels
        del orig_pairs, forg_pairs, gen_gen_labels, gen_for_labels
        all_pairs, all_labels = shuffle(all_pairs, all_labels)

        # Note the Lists above contain only the image names and
        # actual images are loaded and yielded below in batches
        # Below we prepare a batch of data points and yield the batch
        # In each batch we load "batch_size" number of image pairs
        # These images are then removed from the original set so that
        # they are not added again in the next batch.

        k = 0
        pairs=[np.zeros((batch_size, img_h, img_w, 1)) for i in range(2)]
        targets=np.zeros((batch_size,))
        for ix, pair in enumerate(all_pairs):
            img1 = cv2.imread(pair[0], 0)
            img2 = cv2.imread(pair[1], 0)
            img1 = cv2.resize(img1, (img_w, img_h))
            img2 = cv2.resize(img2, (img_w, img_h))
            img1 = np.array(img1, dtype = np.float64)
            img2 = np.array(img2, dtype = np.float64)
            img1 /= 255
            img2 /= 255

```

```

img1 = img1[..., np.newaxis]
img2 = img2[..., np.newaxis]
pairs[0][k, :, :, :] = img1
pairs[1][k, :, :, :] = img2
targets[k] = all_labels[ix]
k += 1
if k == batch_size:
    yield pairs, targets
    k = 0
pairs=[np.zeros((batch_size, img_h, img_w, 1)) for i in range(2)]
targets=np.zeros((batch_size,))

```

```

In [48]: def euclidean_distance(vects):
    '''Compute Euclidean Distance between two vectors'''
    x, y = vects
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))

```

```

In [49]: def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)

```

```

In [50]: def contrastive_loss(y_true, y_pred):
    '''Contrastive loss from Hadsell-et-al.'06
    http://yann.lecun.com/exdb/publis/pdf/hadsell-chopra-lecun-06.pdf
    '''
    margin = 1
    return K.mean(y_true * K.square(y_pred) + (1 - y_true) * K.square(K.maximum(n

```

```

In [51]: def create_base_network_signet(input_shape):
          '''Base Siamese Network'''

          seq = Sequential()
          seq.add(Conv2D(96, kernel_size=(11, 11), activation='relu', name='conv1_1', s
                    init='glorot_uniform', dim_ordering='tf'))
          seq.add(BatchNormalization(epsilon=1e-06, mode=0, axis=1, momentum=0.9))
          seq.add(MaxPooling2D((3,3), strides=(2, 2)))
          seq.add(ZeroPadding2D((2, 2), dim_ordering='tf'))

          seq.add(Conv2D(256, kernel_size=(5, 5), activation='relu', name='conv2_1', st
          seq.add(BatchNormalization(epsilon=1e-06, mode=0, axis=1, momentum=0.9))
          seq.add(MaxPooling2D((3,3), strides=(2, 2)))
          seq.add(Dropout(0.3))# added extra
          seq.add(ZeroPadding2D((1, 1), dim_ordering='tf'))

          seq.add(Conv2D(384, kernel_size=(3, 3), activation='relu', name='conv3_1', st
          seq.add(ZeroPadding2D((1, 1), dim_ordering='tf'))

          seq.add(Conv2D(256, kernel_size=(3, 3), activation='relu', name='conv3_2', st
          seq.add(MaxPooling2D((3,3), strides=(2, 2)))
          seq.add(Dropout(0.3))# added extra
          seq.add(Flatten(name='flatten'))
          seq.add(Dense(1024, W_regularizer=l2(0.0005), activation='relu', init='glorot
          seq.add(Dropout(0.5))

          seq.add(Dense(128, W_regularizer=l2(0.0005), activation='relu', init='glorot_

          return seq

```

```

In [52]: input_shape=(img_h, img_w, 1)

```

```
In [54]: import tensorflow as tf
import keras.backend.tensorflow_backend as tfback

print("tf.__version__ is", tf.__version__)
print("tf.keras.__version__ is:", tf.keras.__version__)

def _get_available_gpus():
    """Get a list of available gpu devices (formatted as strings).

    # Returns
        A list of available GPU devices.
    """
    #global _LOCAL_DEVICES
    if tfback._LOCAL_DEVICES is None:
        devices = tf.config.list_logical_devices()
        tfback._LOCAL_DEVICES = [x.name for x in devices]
    return [x for x in tfback._LOCAL_DEVICES if 'device:gpu' in x.lower()]

tfback._get_available_gpus = _get_available_gpus

tf.__version__ is 2.1.0
tf.keras.__version__ is: 2.2.4-tf
```



```
In [55]: # network definition
base_network = create_base_network_signet(input_shape)

input_a = Input(shape=(input_shape))
input_b = Input(shape=(input_shape))

# because we re-use the same instance `base_network`,
# the weights of the network
# will be shared across the two branches
processed_a = base_network(input_a)
processed_b = base_network(input_b)

# Compute the Euclidean distance between the two vectors in the latent space
distance = Lambda(euclidean_distance, output_shape=eucl_dist_output_shape)([processed_a, processed_b])

model = Model(input=[input_a, input_b], output=distance)
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:6: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(96, kernel_size=(11, 11), activation="relu", name="conv1_1", strides=4, input_shape=(155, 220,..., data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:7: UserWarning: Update your `BatchNormalization` call to the Keras 2 API: `BatchNormalization(epsilon=1e-06, axis=1, momentum=0.9)`
```

```
import sys
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:9: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((2, 2), data_format="channels_last")`
```

```
if __name__ == '__main__':
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:11: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(5, 5), activation="relu", name="conv2_1", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
# This is added back by InteractiveShellApp.init_path()
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:12: UserWarning: Update your `BatchNormalization` call to the Keras 2 API: `BatchNormalization(epsilon=1e-06, axis=1, momentum=0.9)`
```

```
if sys.path[0] == '':
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:15: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((1, 1), data_format="channels_last")`
```

```
from ipykernel import kernelapp as app
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:17: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(384, kernel_size=(3, 3), activation="relu", name="conv3_1", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:18: UserWarning: Update your `ZeroPadding2D` call to the Keras 2 API: `ZeroPadding2D((1, 1), data_format="channels_last")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:20: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(3, 3), activation="relu", name="conv3_2", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launcher.py:21: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(256, kernel_size=(3, 3), activation="relu", name="conv3_2", strides=1, data_format="channels_last", kernel_initializer="glorot_uniform")`
```

```
er.py:24: UserWarning: Update your `Dense` call to the Keras 2 API: `Dense(1
024, activation="relu", kernel_initializer="glorot_uniform", kernel_regulari
zer=<keras.reg...)`
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launch
er.py:27: UserWarning: Update your `Dense` call to the Keras 2 API: `Dense(1
28, activation="relu", kernel_initializer="glorot_uniform", kernel_regulariz
er=<keras.reg...)`
C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\ipykernel_launch
er.py:16: UserWarning: Update your `Model` call to the Keras 2 API: `Model(i
nputs=[<tf.Tenso..., outputs=Tensor("la...)`
app.launch_new_instance()
```

```
In [56]: batch_sz = 128
num_train_samples = 276*120 + 300*120
num_val_samples = num_test_samples = 276*20 + 300*20
num_train_samples, num_val_samples, num_test_samples
```

```
Out[56]: (69120, 11520, 11520)
```

```
In [57]: # compile model using RMSProp Optimizer and Contrastive Loss function defined above
rms = RMSprop(lr=1e-4, rho=0.9, epsilon=1e-08)
model.compile(loss=contrastive_loss, optimizer=rms)
```

```
In [58]: # Using Keras Callbacks, save the model after every epoch
# Reduce the Learning rate by a factor of 0.1 if the validation loss does not improve
# Stop the training using early stopping if the validation loss does not improve
callbacks = [
    EarlyStopping(patience=12, verbose=1),
    ReduceLROnPlateau(factor=0.1, patience=5, min_lr=0.000001, verbose=1),
    ModelCheckpoint('C:/Users/ARPITA/Downloads/Offline-Signature-Verification-using-Siamese-Network-master/Offline-Signature-Verification-using-Siamese-Network-master/model_128.h5',
                    save_best_only=True, verbose=1)
]
```

```
In [59]: results = model.fit_generator(generate_batch(orig_train, forg_train, batch_sz),
                                       steps_per_epoch = num_train_samples//batch_sz,
                                       epochs = 100,
                                       validation_data = generate_batch(orig_val, forg_val, batch_sz),
                                       validation_steps = num_val_samples//batch_sz,
                                       callbacks = callbacks)
```

Epoch 1/100

1/540 [.....] - ETA: 5:16:29 - loss: 154.1226

C:\Users\ARPITA\anaconda3\envs\tensorflow\lib\site-packages\keras\utils\data\_utils.py:718: UserWarning: An input could not be retrieved. It could be because a worker has died. We do not have any information on the lost sample.  
UserWarning)

**After observing continuous fall in validation accuracy, I interrupted the training manually**

In [ ]:

```
In [60]: def compute_accuracy_roc(predictions, labels):
'''Compute ROC accuracy with a range of thresholds on distances.
...
dmax = np.max(predictions)
dmin = np.min(predictions)
nsame = np.sum(labels == 1)
ndiff = np.sum(labels == 0)

step = 0.01
max_acc = 0
best_thresh = -1

for d in np.arange(dmin, dmax+step, step):
    idx1 = predictions.ravel() <= d
    idx2 = predictions.ravel() > d

    tpr = float(np.sum(labels[idx1] == 1)) / nsame
    tnr = float(np.sum(labels[idx2] == 0)) / ndiff
    acc = 0.5 * (tpr + tnr)
#    print ('ROC', acc, tpr, tnr)

    if (acc > max_acc):
        max_acc, best_thresh = acc, d

return max_acc, best_thresh
```

**Load the weights from the epoch which gave the best validation accuracy**

```
In [61]: model.load_weights('C:/Users/ARPITA/Downloads/Offline-Signature-Verification-usir
```

```
In [62]: test_gen = generate_batch(orig_test, forg_test, 1)
pred, tr_y = [], []
for i in range(num_test_samples):
    (img1, img2), label = next(test_gen)
    tr_y.append(label)
    pred.append(model.predict([img1, img2])[0][0])
```

```
In [63]: tr_acc, threshold = compute_accuracy_roc(np.array(pred), np.array(tr_y))
tr_acc, threshold
```

```
Out[63]: (0.8466330243713449, 0.3400007599592209)
```

**Accuracy = 84.66% and Threshold = 0.34**

Thus if the difference score is less than 0.34, we predict the test image as Genuine and if the difference score is greater than 0.34, we predict it to be as forged

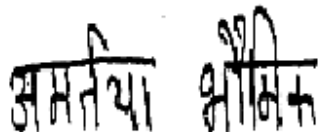
**Below we see some sample results**

```
In [66]: def predict_score():
'''Predict distance score and classify test images as Genuine or Forged'''
test_point, test_label = next(test_gen)
img1, img2 = test_point[0], test_point[1]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 10))
ax1.imshow(np.squeeze(img1), cmap='gray')
ax2.imshow(np.squeeze(img2), cmap='gray')
ax1.set_title('Genuine')
if test_label == 1:
    ax2.set_title('Genuine')
else:
    ax2.set_title('Forged')
ax1.axis('off')
ax2.axis('off')
plt.show()
result = model.predict([img1, img2])
diff = result[0][0]
print("Difference Score = ", diff)
if diff > threshold:
    print("Its a Forged Signature")
else:
    print("Its a Genuine Signature")
```

```
In [67]: predict_score()
```

Genuine



Forged



Difference Score = 0.7450892

Its a Forged Signature

**Note: The first image is always Genuine. Score prediction and classification is done for the second image**

In [68]: `predict_score()`

Genuine

सोना सिंह

Genuine

सोना सिंह

Difference Score = 0.24016695  
Its a Genuine Signature

In [69]: `predict_score()`

Genuine

जयवंत सिंह

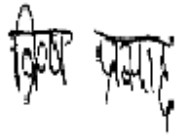
Genuine

जयवंत सिंह

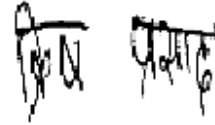
Difference Score = 0.19460675  
Its a Genuine Signature

```
In [70]: predict_score()
```

Genuine



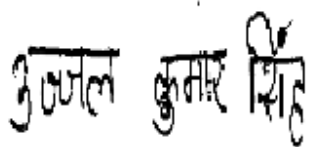
Genuine



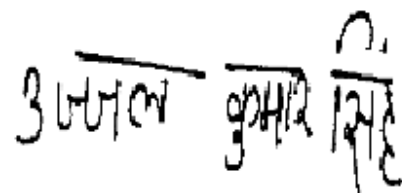
Difference Score = 0.018434573  
Its a Genuine Signature

```
In [71]: predict_score()
```

Genuine



Forged



Difference Score = 1.0802819  
Its a Forged Signature