Project Report

on

Writer Verification on Multi-Language Script using Deep Learning

(A dissertation submitted in partial fulfilment of the requirements of Bachelor of Technology in Computer Science and Engineering of the Maulana Abul Kalam Azad University of Technology, West Bengal)

Submitted by

Souporno Ghosh
Soumya Nasipuri
Rahul Roy
Sharanya Saha

Under the guidance of
Prof. Jaya Paul

Asst. Prof.,
Dept. of Computer Science and Engineering

Government College of Engineering and Leather Technology

(Affiliated to MAKAUT, West Bengal)

Kolkata - 700106, WB

2020-2021

Certificate of Approval

This is to certify that the project report on "Writer Verification on Multi-Language Script using Deep Learning" is a record of bona fide work, carried out by Shri Souporno Ghosh, Shri Soumya Nasipuri, Shri Rahul Roy and Smt Sharanya Saha under my guidance and supervision.

In my opinion, the report in its present form is in conformity as specified by Government College of Engineering and Leather Technology and as per regulations of the Maulana Abul Kalam Azad University of Technology, West Bengal. To the best of my knowledge the results presented here are original in nature and worthy of incorporation in project report for the B.Tech. Program in Computer Science and Engineering.

Signature of	Signature of
Supervisor/ Guide	Head, Dept. of CSE

ACKNOWLEDGEMENT

With great pleasure, I would like to express my profound gratitude and indebtedness to Prof Jaya Paul, Department of Computer Science and Engineering, Government College of Engineering and Leather Technology, W.B. for her continuous guidance, valuable advice and constant encouragement throughout the project work. Her valuable and constructive suggestions at many difficult situations are immensely acknowledged. I am in short of words to express her contribution to this thesis through criticism, suggestions and discussions.

I would like to take this opportunity to thank Prof Jaya Paul, Project Coordinator and Dr. Santanu Halder, HOD, Department of Computer Science & Engineering, Government College of Engineering and Leather Technology.

I would like to express my gratitude to Sri S. Mondal for his valuable suggestions and help.

Signatures

Souporno Ghosh

1. Souporno Ghosh – 11200117028

Soumya Nasipuri

2. Soumya Nasipuri – 11200117029

Rahul Roy

3. Rahul Roy – 11200117039

Sharanya Saha

4. Sharanya Saha - 11200117033

Dedicated to

Ada Lovelace, Alan Turing and John Nash

The pioneers on whose work we expand upon.

OBJECTIVE

Significant amount of research has been done in the field of handwriting recognition, particularly for characters in the Latin-based alphabets (English, French, Spanish, German, etc). However, there is a significant shortage of literature and research on handwriting recognition for Devanagari based languages, such as Hindi, Bangla, Sanskrit, etc. In this project, we attempt to remedy that in an attempt to create an API that is able to recognize the writer for a passage written in Bangla handwriting. The primary goal of the API is to identify the author of a word, sentence or passage from the handwriting written in Bangla. We also provide the methods used by us in this attempt in order to facilitate further study and replication of this API for future research. In this project, we will apply the VGG16 model to identify the writer of a text written in Bangla. We will also compare our accuracy with previous works in this field.

CONTENTS

CHAPTER 1. INTRODUCTION

1. Motivation

Our great nation has produced many literary geniuses; Munshi Prem Chand, Rabindranath Tagore, Vikram Seth, Bankim Chandra Chatterjee, Sukumar Roy, etc. These pioneers have blessed us with a variety of literary masterpieces that provide not only an insight into their own minds but also insight on humanity and contemporary times. It is sufficed to say one learns a lot about mankind from their works. Moreover, their works and by extension, the manuscripts of said works, are a national treasure. Hence, it is imperative that their original works can be verified as their own. We can achieve this by analysing the handwritings of the writers.

Handwriting recognition will also help accelerate the field of forensic analysis, and in turn help the law enforcement authorities. Notes found on crime scenes and related to victims and suspects can be analysed and such analysis can help us identify perpetrators of a crime.

the field of education and academia, handwriting analysis can help us curb plagiarism. Plagiarism checking is a major field of research in academia and handwriting analysis can also help support those efforts.

These are only a few of the applications of handwriting recognition that we drove us to choose this topic for our project. Recognition and analysis of handwriting has applications in various fields such as archaeology, criminal detection, academia, education, etc. However, so far handwriting analysis has only been performed by human hands. In modern days, handwriting recognition has mostly only been attempted for languages based on Latin-based alphabet.

Literature related to handwriting recognition is limited for Devanagari related languages, such as Hindi and Bangla. Hence, our feeble attempt at remedying that.

2. Background

To decide what approach, we should take to recognise the writer from the handwriting, we decided to survey related literature. First, we look into image recognition basics to find out the best way to approach the image recognition problem. We looked into the validity of Deep Convolutional Neural Networks for image recognition as discussed by Krizhevsky et al. [1]. Additionally, we referenced the original article [2] where the VGG16 was first proposed.

To understand how handwriting can be treated as images, we looked into articles on handwritten character recognition. We referred to the following works specifically. Hasnat et al. proposed a domain specific OCR which classify machine printed as well as handwritten Bangla characters. For feature extraction they apply Discrete Cosine Transform (DCT) technique over the input image and for classification Hidden Markov Model (HMM) was used [5]. Paul et al. has attempted Bangla character recognition with Mobilenet v1 and Inception v3 [6] and Bangla number recognition with Convolutional Neural Networks [7]. While the works are different from what we are trying achieve, they help us find an appropriate way to approach handwriting.

We also referred to the following works. Christlein et al. [8] attempted author recognition with Convolutional Neural Network. Schlapbache et al. [9] analysed HMM based handwriting recognition systems and studied the effect of normalisation operations. Wu et al. [10] attempted writer identification on English and Chinese languages.

3. Summary of present work

As mentioned earlier, the work done in the field of Bangla character recognition is very limited. For reference and guidance, we looked into the work of Adak et al. [11].

In the article, high intra-variable handwriting-based writer identification/verification is attempted. Both handcrafted and auto-derived feature-based models are considered to study writer identification/verification performance.

Two offline Bangla intra-variable handwriting databases from two different sets of 100 writers are used; Controlled (D_C) and Uncontrolled (D_{UC}). For writer identification, multi-class classification was used (the number of classes is equal to the total count of writers) where the task was to assign the writer-id to the unknown handwritten specimens. For writer verification, a binary classification was used where the task is to answer “yes” or “no” to the handwritten sample in question.

Both writer identification and writer verification were tried out with two methods: Handcrafted Feature-Based Identification and Auto-Derived Feature-Based Identification. The process used for Handcrafter Feature extraction are: Macro-Micro Features (F_{MM}), Contour Direction and Hinge Features (F_{DH}) and Direction and Curvature Features at Key points (F_{DC}). For Auto-Derived Feature Extraction Basic_CNN, SqueezeNet, GoogLeNet, Xception Net, VGG16 and ResNet 101 are used.

After experimentation on the databases, it is observed that by training and testing on similar writing variability, the system produces encouraging outcomes. However, the system performance is comparatively lower for training and testing on disparate types of handwriting variability. Cross learning is also attempted and it is observed that the system performance improves with pre-training. Here, a practical scenario is imitated, whereby a certain writing style of an individual is unknown (i.e., absent during training), and we note that the state-of-the-art methods do not perform well.

The following tables contains the summarised result of their work.

The accuracies corresponding to writer verification on an enlarged set is shown below. Since, we have attempted writer verification with VGG16 model in our project, we have only mentioned the verification accuracy for VGG16 model as measured by Adak et al. here.

The following is the table containing accuracies when the model was trained with and tested on a controlled database.

Model	**Accuracy (%)**
VGG16 on Character Level Features	97.77
VGG16 on Stroke Level Features	97.20

Table 1.1. Accuracy for Writer Verification on Controlled Data Set

The following is the table containing accuracies when the model was trained with and tested on an uncontrolled database.

Model	**Accuracy (%)**
VGG16 on Character Level Features	97.36
VGG16 on Stroke Level Features	96.43

Table 1.2. Accuracy for Writer Verification on Uncontrolled Data Set

4. Hardware/Software used

Primary language used for programming is Python 3.8.

Packages used in Python are TensorFlow v2.0 (a deep learning framework by Google, Inc.), Keras (to help with integrating TensorFlow with Python), NumPy (NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays), Pandas (software library written for the Python programming language for data manipulation and analysis), Matplotlib (used for plotting data and graphs) and Seaborn (Python data visualization library based on matplotlib; It provides a high-level interface for drawing attractive and informative statistical graphics).

The primary algorithm used for image (handwriting) recognition is VGG16.

The primary hardware used comprise the personal laptop computers belonging to the team members; HP Pavilion with 2.7GHz Quad-Core Intel i5, Integrated Graphics Card, 8GB RAM; Dell G3 with 2.6GHz Hexa-Core Intel i7 Processor, Integrated Graphics Card, 8GB RAM and Asus Vivobook 2GHz Quad-Core AMD Ryzen 5 Processor, Integrated Graphics Card, 8GB RAM.

The processor and the RAM determine how fast the program will run and the dataset will be trained. The graphics processing unit would also have accelerated the process, if GPU parallel processing was implemented in the project. A major issue with deep learning projects is that if the processing power is too less, then the model might take as long as a few days to train. However, our initial project does not take as much time and can be replicated with a machine with as low specifications as 1.6 GHz Dual Core Processor, Integrated Graphics Card and 4GB RAM. Anything lower might take too long to process. (Training the Dataset will require much better specifications, however, since the original VGG16 model was trained on 2-3 weeks on a Nvidia Titan GPU [2]. In such a scenario, training the data on a cloud platform like GCP, AWS or Azure might yield better results).

CHAPTER 2. INITIAL STAGES

1. Collection of Dataset

A major criterion of success in working on problems whose solutions depend on machine or deep learning is the presence of large datasets. The larger the dataset, the better and more accurate the model becomes.

One of the main challenges of attempting handwriting recognition in a new language is the lack of sufficient datasets. Our supervisor for the project, Smt Jaya Paul, was generous enough to assist us with that. We used 121 volunteers of various native languages and asked them to write certain passages in their native and English languages. This provided us with the necessary dataset needed for proper model training.

For our project, we have used the dataset containing Bangla words only.

2. Preparation of Dataset

The handwritten passages were scanned into image. The scanned images passages were processed to eliminate noise and then segmented into word-sized images. The methods used for this was provided to us from a previous project attempted by Mondal et al. [12]. These word sized images are then arranged into several folders in the following method.

- The images are distinguished according to pairs of authors and for each author, there are five (5) sets of data.
- Each of these 5 sets are divided into two subsets; train containing three sets and test containing two sets.
- The images are in Tag Image File Format (TIFF). They are named in the following format: <Author Code>_<Set Number>_<Image Number>. For example, the first image of the first set of the first author (Author Code 0) is *0000_01_0.tiff.*
- There are five thusly organized datasets.

These images are now ready to be fed as inputs for the model.

3. Construction of the Model

Currently, the base model that we will be using is called VGG16. VGG16 is one of the most popular models for image recognition. It was introduced in 2014 [2]. VGG16 takes the input in 224 x 224 images and the pretrained VGG 16 model (with the ImageNet Database) will predict from the 1000 categories provided in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2014.

In our dataset, the size of the segmented images is random, and hence some pre-processing is required so that the input images can be converted to the required size of 224 x 224. We shall do that with the help of the in-built VGG 16 pre-processing tools provided in the Keras library. The Methodology mentions exactly how we do it.

CHAPTER 3. METHODOLOGY

1. About VGG 16 Model [2]

VGG 16 was proposed by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group Lab at the University of Oxford in 2014 in an article titled "Very Deep Networks for Large Scale Image Recognition." This model achieves 92.7% on the top 5 accuracy tests with the ImageNet dataset of 14 million images belonging to 1000 categories. The model inputs 224 x 244 pixels size images in RBG channels.

1.1 Architecture of VGG16

The input to the network is image of dimensions (224, 224, 3). The first two layers have 64 channels of 3*3 filter size and same padding. Then after a max pool layer of stride (2, 2), two layers which have convolution layers of 256 filter size and filter size (3, 3). This followed by a max pooling layer of stride (2, 2) which is same as previous layer. Then there are 2 convolution layers of filter size (3, 3) and 256 filter. After that there are 2 sets of 3 convolution layer and a max pool layer. Each have 512 filters of (3, 3) size with same padding. This image is then passed to the stack of two convolution layers. In these convolution and max pooling layers, the filters we use is of the size 3*3 instead of 11*11 in Alex Net and 7*7 in ZF-Net. In some of the layers, it also uses 1*1 pixel which is used to manipulate the number of input channels. There is a padding of 1-pixel (same padding) done after each convolution layer to prevent the spatial feature of the image.

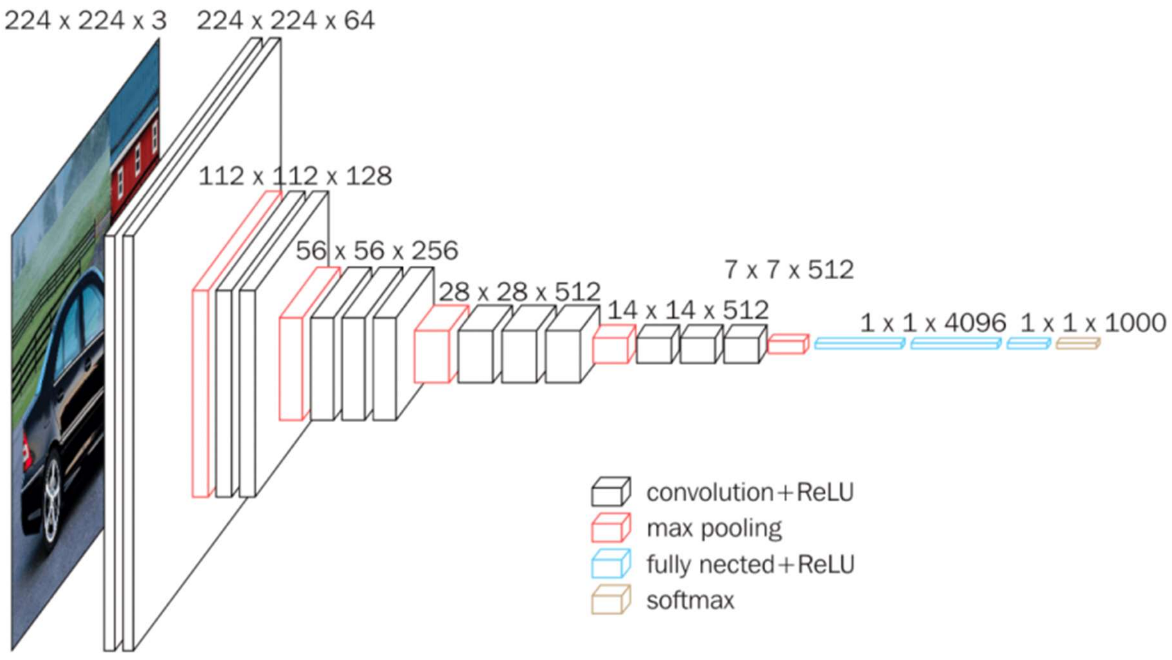


Fig 3.1. Architecture of VGG16 Model

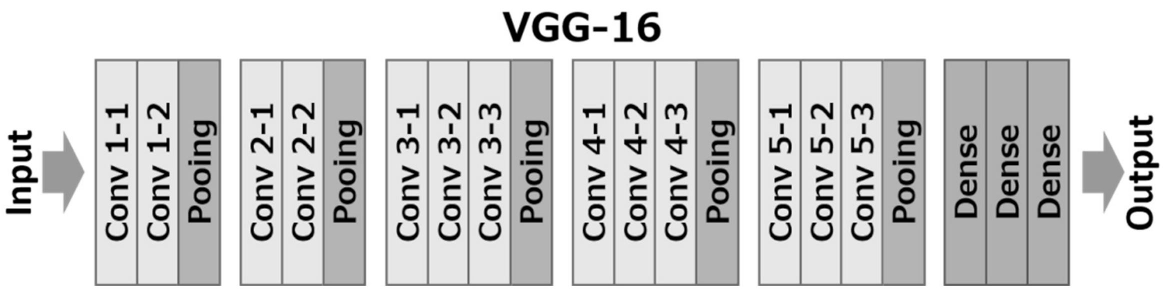


Fig 3.2. Flowchart of VGG16 Model

After the stack of convolution and max-pooling layer, we got a (7, 7, 512) feature map. We flatten this output to make it a (1, 25088) feature vector. After this there are 3 fully connected layer, the first layer takes input from the last feature vector and outputs a (1, 4096) vector, second layer also outputs a vector of size (1, 4096) but the third layer output 1000 channels for 1000 classes of ILSVRC challenge, then after the output of 3rd fully connected layer is passed to SoftMax layer in order to normalize the classification vector. After the output of classification vector top-5 categories for evaluation. All the hidden layers use ReLU as its activation function. ReLU is more computationally efficient because it results in faster learning and it also decreases the likelihood of vanishing gradient problem.

This is how the pretrained VGG16 model works.

1.2 Challenges of VGG16

- It is very slow to train (the original VGG model was trained on NVidia Titan GPU for 2-3 weeks).
- The size of VGG16 trained ImageNet weights is 528 MB. So, it takes quite a lot of disk space and bandwidth that makes it inefficient.

2. Why VGG16?

Recognizing images has always been more efficient with the help of deep neural networks. A. Krizhevsky, I. Sutskever and G.E. Hinton [1] have shown that while using a huge dataset like ImageNet, the results show that a large, deep convolutional neural network is capable of achieving record-breaking results on a highly challenging dataset using purely supervised learning. It is notable that our network's performance degrades if a single convolutional layer is removed.

VGG16 happens to be one of the best models for image recognition with accuracy 92.7% on the top 5 accuracy tests on the ImageNet dataset of 14 million images belonging to 1000 categories [2]. Handwriting can be considered as a pattern to be recognised. Hence, we decided that VGG16 model might be the best path to proceed with the handwriting recognition.

3. Initial Experimentation

Before we proceeded to handwriting recognition, we experimented with a cat and dog classifier [13]. The dataset we found was from Kaggle [14]. We attempted to create our own VGG16 model in order to better understand the VGG16 model.

We have used Keras, which is a package which helps in using TensorFlow 2.0 Library by Google, Inc.

We have imported the necessary modules from keras. This include but are not limited various algorithms that will constitute each layer of the model (namely Dense, Convolution, Max Pool and Flattening), image pre-processing tools and various keras utilities.

We have also used NumPy and Mat Plot Library. NumPy assists us with handling matrices and Mat Plot Library helps us to view graphs and plots.

The data is divided into two subsets; test and train and is used in the program.

We have added the layers relevant to VGG16 model. The first set of layers includes two convolutional networks.

The first one takes in the image of size 224 x 224 in RGB channels. The next one has 64 filters. We have used ReLU as the activation function. The output after these two layers is pooled with max pooling algorithm. Then we have added two more convolutional networks with 128 filters, activation function ReLU and the same padding. The output after these two layers is pooled with max pooling algorithm. We have again added three more convolutional networks with 256 filters, activation function ReLU and the same padding. The output after these two layers is pooled with max pooling algorithm.

Three more convolutional networks with 512 filters, activation function ReLU and the same padding are added. The output after these two layers is pooled with max pooling algorithm. Another set of three more convolutional networks with 512 filters, activation function ReLU and the same padding are added. The output after these two layers is pooled with max pooling algorithm.

We have used Rectified Linear Unit (ReLU) for activation so that the negative values are not passed to the next layers. After this, we have added two Dense layers with ReLU activation with 4096 units followed by a SoftMax Dense layer with two units. We have used Adam optimiser to reach to the global minima while training out model. If the algorithm is stuck in local minima while training then the Adam optimiser will help us to get out of local minima and reach global minima. We have also specified the learning rate of the optimiser, here in this case it is set at 0.001. We have used the ModelCheckpoint Utility from Keras library to save the model only when the validation accuracy of the model in current epoch is greater than what it was in the last epoch. The EarlyStopping Utility from Keras library to stop the learning in case there is no increase in validation accuracy for 20 epochs.

This above experimental program gives us in depth understanding of the VGG16 algorithm and its architecture.

4. Extracting Features

Before we could start training the actual model, we needed to pre-process the input. This is primarily because the input for the VGG16 model requires the images to be of size 224 x 224 pixels and RGB channel. Additionally, before training the model, we have extracted the necessary features required for prediction.

To obtain the necessary features, we take the pretrained VGG16 model available in the Keras library and remove the top layer, where the prediction (with the 1000 categories from ILSVRC challenge) actually happens. Then we pass the data through it to get a "prediction" to get a feature matrix as an output.

From the existing pretrained model, we have removed the top layer. The top layer is responsible for actually predicting the category of the image and it does that from the 1000 categories provided by the 1000 categories from ILSVRC challenge. If this layer is removed, then the classification does not occur. Instead, we get the features necessary for the classification of the images. The weights of this pretrained model are configured according to the ImageNet dataset.

In the actual program, we have looped through all the images so that we can get the features for all the images. We have stored the feature matrices in CSV files. We have created one CSV file for each set of data. Since there are 5 sets of data for each author pair (3 for training and 2 for verification), we have 5 CSV files containing the features of all the images. We have used these feature matrices to train and validate our model.

5. Extracted Features

We will input the following image to the updated (i.e., pretrained VGG16 without the top layer) model.

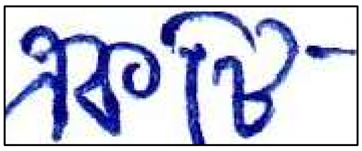


Fig 3.3. Segmented image input to the updated model

Running the earlier program gives us the following output.

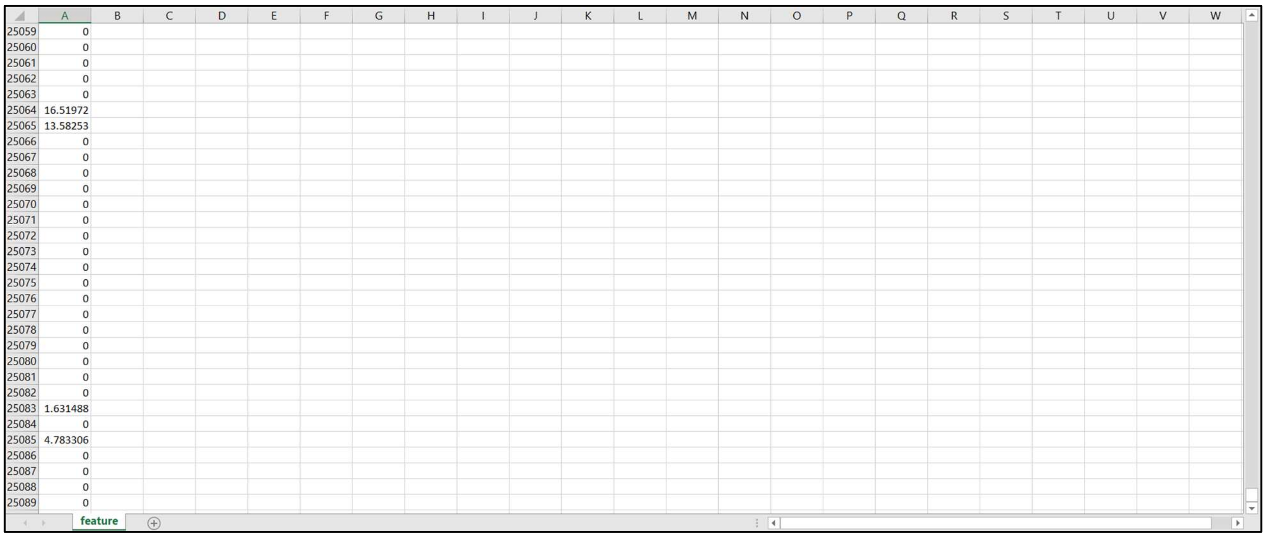
```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, None, None, 3)]   0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
```

```
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
(224, 224, 3)
(1, 224, 224, 3)
(1, 224, 224, 3)
2021-03-27 23:23:36.385607: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR
optimization passes are enabled (registered 2)
(1, 7, 7, 512)
(25088,)
```

The following is part of the feature matrix (NumPy does not print all of the values if the array is too big). It is a 1 x 7 x 7 x 512 matrix. Our goal will be appending the matrices for all the sets to obtain the features relevant to an author and then use the features to verify the handwriting with respect to the writer.

```
[[[[ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   ...
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]]

  [[ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   [ 0.          0.          0.         ...  0.          0.
     0.        ]
   ...
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]]

  [[ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   ...
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]]

  ...

  [[ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   ...
   [ 0.         0.         0.        ...  0.         0.9801242
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]]

  [[ 0.         0.         0.        ...  0.         1.8023927
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   ...
   [ 0.         0.         0.        ...  0.         6.553742
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]]

  [[ 0.         0.         0.        ...  0.         6.669153
     0.       ]
   [ 0.         0.         0.        ...  0.         3.7987955
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]
   ...
   [ 0.         0.         0.        ...  0.        14.567184
     0.       ]
   [ 0.         0.         0.        ...  0.         4.3287916
     0.       ]
   [ 0.         0.         0.        ...  0.         0.
     0.       ]]]]
```

We will flatten this to create a feature matrix for a set for particular author. The resultant matrix is 25088 x 1. The following is the screenshot of the output CSV file after flattening the 1 x 7 x 7 x 512 feature matrix.



	A
25059	0
25060	0
25061	0
25062	0
25063	0
25064	16.51972
25065	13.58253
25066	0
25067	0
25068	0
25069	0
25070	0
25071	0
25072	0
25073	0
25074	0
25075	0
25076	0
25077	0
25078	0
25079	0
25080	0
25081	0
25082	0
25083	1.631488
25084	0
25085	4.783306
25086	0
25087	0
25088	0
25089	0

Fig 3.4. Screenshot of the CSV file with the feature matrix

6. Verification

After completing feature extraction, we have updated the VGG16 model again. This time we have only updated the last layer of the model to provide two outputs instead of the original 1000 (similar to the dog and cat classifier). Then we have trained the model with every writer pair. This trains the model to be able to distinguish between the handwriting of the primary subject of the writer pair and the rest. After training with all the available dataset, we have used the testing part of the dataset to test the accuracy of our model. Using this, we have calculated the accuracy of our method of verification.

After the extraction of the features is complete, we update the VGG16 model again. This time we will only update the last layer of the model to provide two outputs representing true if the handwriting in the image belongs to the writer specific to the model and false otherwise instead of the original 1000 (similar to the dog and cat classifier). Then we will train the model with every writer pair. This trains the model to be able to distinguish between the handwriting of the primary subject of the writer pair and the rest. After training with all the available dataset, we will use the testing part of the dataset to test the accuracy of our model. Using this, we will calculate the accuracy of our method of verification.

6.1 Updating the last layer

The last layer is updated by adding dense layers with 2 outputs as mentioned earlier. Activation functions like relu and softmax are used. The rectified linear activation function or ReLU for short is a piecewise linear function that outputs the input directly if it is positive, otherwise, it outputs zero. Softmax is used in the output layer of neural network models which predicts a multinomial probability distribution. It converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector.

In the present work, we have used Adam optimizer to optimize gradient descent. Using Adam optimizer, we can control the rate of gradient descent in such a way that there is minimum oscillation when it reaches the global minimum while taking big enough steps (step-size) so as to pass the local minima hurdles along the way.

It is a combination of the ‘gradient descent with momentum’ algorithm and the ‘RMSP’ algorithm.

Momentum is used to accelerate the gradient descent algorithm by taking the ‘exponentially weighted average’ of the gradients into consideration. Using averages makes the algorithm converge towards the minima at a faster pace.

Here, we have taken the learning rate to be 0.001. We have used the ModelCheckpoint Utility from Keras library to save the model only when the validation accuracy of the model in current epoch is greater than what it was in the last epoch.

Batch size is the number of training examples in one forward/backward pass. We have taken the batch size to as 32, mostly because the value should be in powers of 2 and it is directly proportional to the memory needed. The EarlyStopping Utility from Keras library is used to stop the learning in case there is no increase in validation accuracy for 20 epochs.

We also stored the model as a .h5 file.

7. Additional Models

While VGG-16 was our primary model, we also used ResNet and AlexNet to compare the performance of these models with VGG-16.

A residual neural network (ResNet) is an artificial neural network (ANN) of a kind that builds on constructs known from pyramidal cells in the cerebral cortex. Residual neural networks do this by utilizing skip connections, or shortcuts to jump over some layers. Typical ResNet models are implemented with double- or triple- layer skips that contain nonlinearities (ReLU) and batch normalization in between [15].

AlexNet is the name of a convolutional neural network (CNN) architecture, designed by Alex Krizhevsky in collaboration with Ilya Sutskever and Geoffrey Hinton, who was Krizhevsky's Ph.D. advisor [16]. AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge on September 30, 2012. [17] The network achieved a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up. The original paper's primary result was that the depth of the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training. [1]

The structure of the ResNet model is as follows.

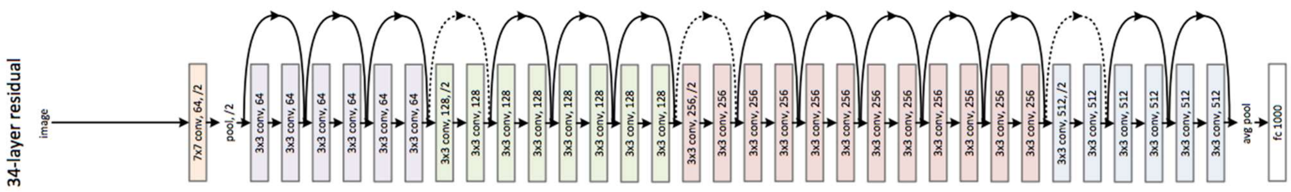


Fig 3.5 Structure of ResNet

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (3): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (3): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (4): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (5): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
```

```
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=2, bias=True)
)

End of ResNet
```

This is the summary of the ResNet model as used in the present work.

The structure of the AlexNet model is as follows.

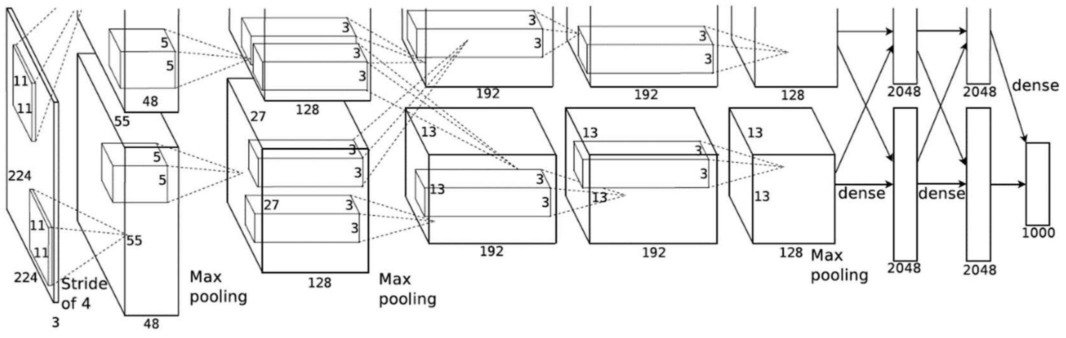


Fig 3.6 Structure of AlexNet

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)

features.0.weight --> torch.Size([64, 3, 11, 11])
features.0.bias --> torch.Size([64])
features.3.weight --> torch.Size([192, 64, 5, 5])
features.3.bias --> torch.Size([192])
features.6.weight --> torch.Size([384, 192, 3, 3])
features.6.bias --> torch.Size([384])
features.8.weight --> torch.Size([256, 384, 3, 3])
features.8.bias --> torch.Size([256])
features.10.weight --> torch.Size([256, 256, 3, 3])
features.10.bias --> torch.Size([256])
classifier.1.weight --> torch.Size([4096, 9216])
classifier.1.bias --> torch.Size([4096])
classifier.4.weight --> torch.Size([4096, 4096])
classifier.4.bias --> torch.Size([4096])
classifier.6.weight --> torch.Size([1000, 4096])
classifier.6.bias --> torch.Size([1000])
Total number of parameters: 61100840

End of AlexNet
```

This is the summary of the AlexNet model as used in the present work.

These models too had to be updated to work for our specific purpose. The change in both the model was to reduce the output size from their original to two.

CHAPTER 4. OUTPUTS AND RESULTS

1. VGG16 Models

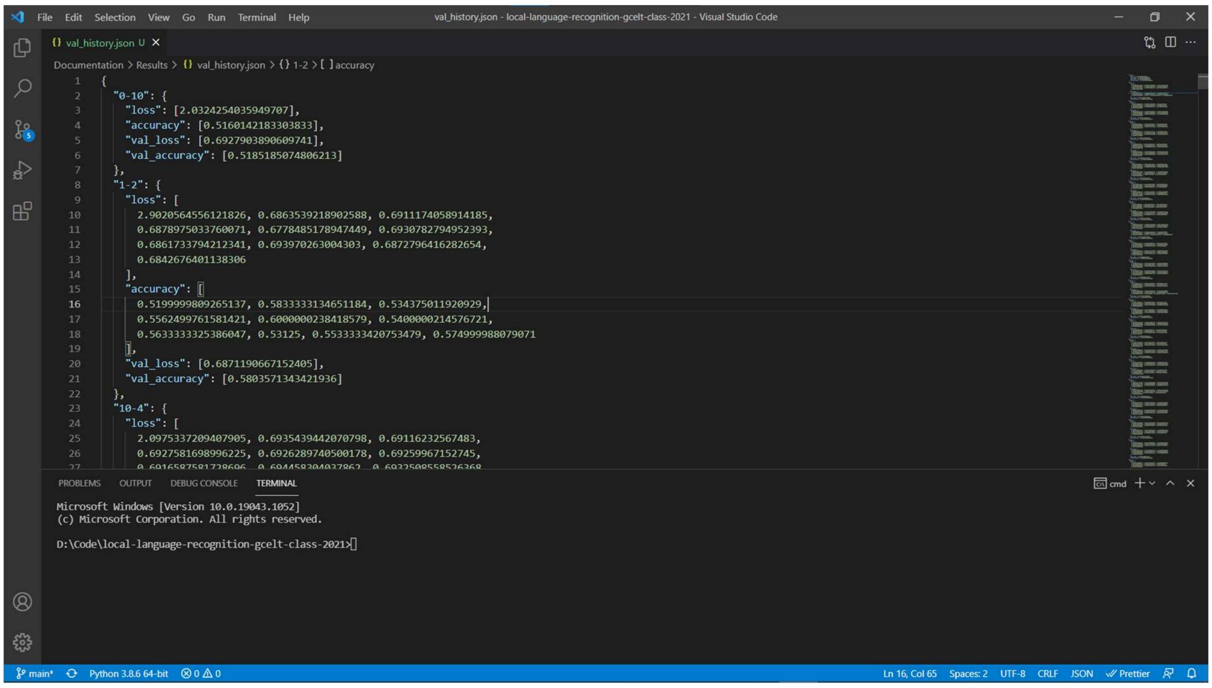
Upon the creation of the model, we examine the model to check if the layers are correct. The model looks like this.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 224, 224, 64)      1792      
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 224, 224, 64)      36928     
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 64)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 128)     73856     
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 128)     147584    
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 128)       0         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 56, 56, 256)       295168    
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 56, 56, 256)       590080    
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 256)       590080    
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 256)       0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 512)       1180160   
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 28, 28, 512)       2359808   
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 512)       2359808   
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 512)       0         
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 14, 14, 512)       2359808   
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 14, 14, 512)       2359808   
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 512)       2359808   
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 512)         0         
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0         
_________________________________________________________________
dense (Dense)                (None, 4096)              102764544 
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312  
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 8194      
=================================================================
Total params: 134,268,738
Trainable params: 134,268,738
Non-trainable params: 0
_________________________________________________________________
```

As we see, the model also includes the last three layers, as well as the modification to the last dense layer i.e., reducing the number of outputs from 1000 to 2.

2. Accuracy

After training models for all of the writers, we average the obtained accuracies. All the accuracies are stored in the JSON file titled *val_history.json*. The file looks as shown below.


```
{
  "0-10": {
    "loss": [2.0324254035949707],
    "accuracy": [0.5160142183303833],
    "val_loss": [0.6927903890609741],
    "val_accuracy": [0.5185185074806213]
  },
  "1-2": {
    "loss": [
      2.9020564556121826, 0.6863539218902588, 0.6911174058914185,
      0.6878975033760071, 0.6778485178947449, 0.6930782794952393,
      0.6861733794212341, 0.693970263004303, 0.6872796416282654,
      0.6842676401138306
    ],
    "accuracy": [
      0.5199999809265137, 0.5833333134651184, 0.534375011920929,
      0.5562499761581421, 0.6000000238418579, 0.5400000214576721,
      0.5633333325386047, 0.53125, 0.5533333420753479, 0.574999988079071
    ],
    "val_loss": [0.6871190667152405],
    "val_accuracy": [0.5803571343421936]
  },
  "10-4": {
    "loss": [
      2.0975337209407905, 0.6935439442070798, 0.69116232567483,
      0.6927581698996225, 0.6926289740500178, 0.69259967152745,
```

```
Microsoft Windows [Version 10.0.19043.1052]
(c) Microsoft Corporation. All rights reserved.

D:\Code\local-language-recognition-gcelt-class-2021>
```

This gives us the final accuracy of the algorithm. By this method, the final accuracy of the model is 62.75% (0.6275384060487466).

```
D:\Code\local-language-recognition-gcelt-class-2021>cd Documentation

D:\Code\local-language-recognition-gcelt-class-2021\Documentation>cd FinalYearProject2021

D:\Code\local-language-recognition-gcelt-class-2021\Documentation\FinalYearProject2021>py
readAccuracy.py
0.6275384060487466
```

Below, we have accuracy v/s epoch graph, which tells us about the variation of accuracy and loss with increase in number of epochs.

2.1 Accuracy for 1-2 Pair Graph

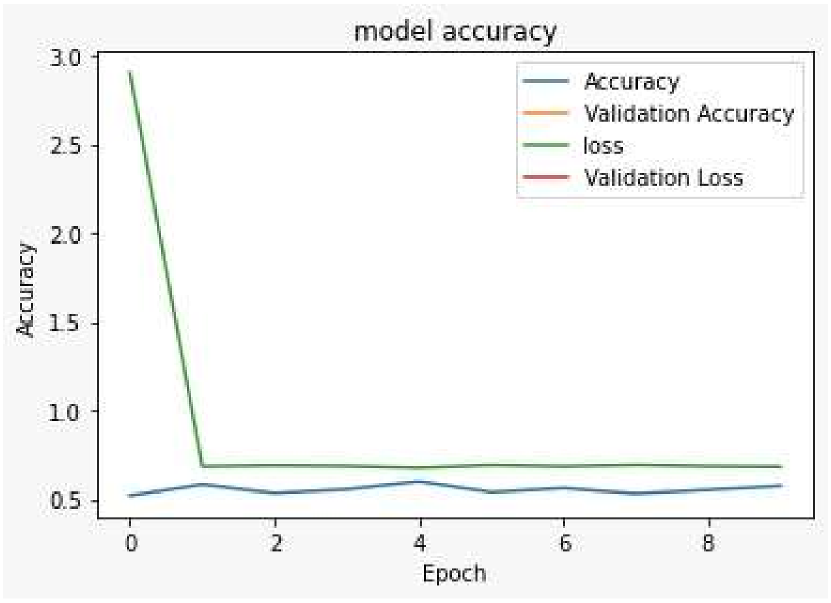


Fig 4.1. Graph depicting accuracy, validation accuracy, loss and validation loss over multiple epochs

2.2 Accuracy for 1-2 Writer Pair in JSON format

The following JavaScript Object represents the loss values and accuracies that we achieved over multiple epochs of training.

```
  {
        "loss": [
            2.9020564556121826,
            0.6863539218902588,
            0.6911174058914185,
            0.6878975033760071,
            0.6778485178947449,
            0.6930782794952393,
            0.6861733794212341,
            0.693970263004303,
            0.6872796416282654,
            0.6842676401138306
        ],
        "accuracy": [
            0.5199999809265137,
            0.5833333134651184,
            0.534375011920929,
            0.5562499761581421,
            0.6000000238418579,
            0.5400000214576721,
            0.5633333325386047,
            0.53125,
            0.5533333420753479,
            0.574999988079071
        ],
        "val_loss": [
            0.6871190667152405
        ],
        "val_accuracy": [
            0.5803571343421936
        ]
  }
```

2.3 Final Accuracy

Above lines show the accuracy, loss, the validation accuracy and the validation loss. The final accuracy for the model is calculated by taking the average and is 62.75% (0.6275384060487466).

3. Accuracies for other models

In this section, we will take a look at the results we obtained from training and testing with other models like ResNet and AlexNet.

3.1 ResNet

This is the JavaScript Object representing the accuracy and loss of the ResNet Model.

```
{
  "inLoss": [
    0.7038170880243659, 0.6930449021668621, 0.6927556230080085,
    0.6937197682272073, 0.6934835749588827, 0.6923972921863569,
    0.6937763277321948, 0.6833371921273419, 0.6930004186477525,
    0.6837705390832761
  ],
  "trainAcc": [
    51.245551601423486, 50.1779359430605, 53.380782918149464,
    53.380782918149464, 51.95729537366548, 53.380782918149464,
    53.380782918149464, 54.09252669039146, 49.11032028469751, 56.58362989323843
  ],
  "testLoss": [
    1.7409996078719223, 1.7271149366620988, 1.729944820441897,
    1.7253038501613354, 1.7283374288725475, 1.7243495684452157,
    1.7241112548838218, 1.708699175654344, 1.7236164376218484,
    1.7177339366651954
  ],
  "testAcc": [
    48.148148148148145, 48.148148148148145, 48.148148148148145,
    48.148148148148145, 48.148148148148145, 48.148148148148145,
    48.148148148148145, 51.851851851851855, 48.148148148148145,
    48.148148148148145
  ]
}
```

Below, we have accuracy v/s epoch graph, which tells us about the variation of accuracy and loss with increase in number of epochs.

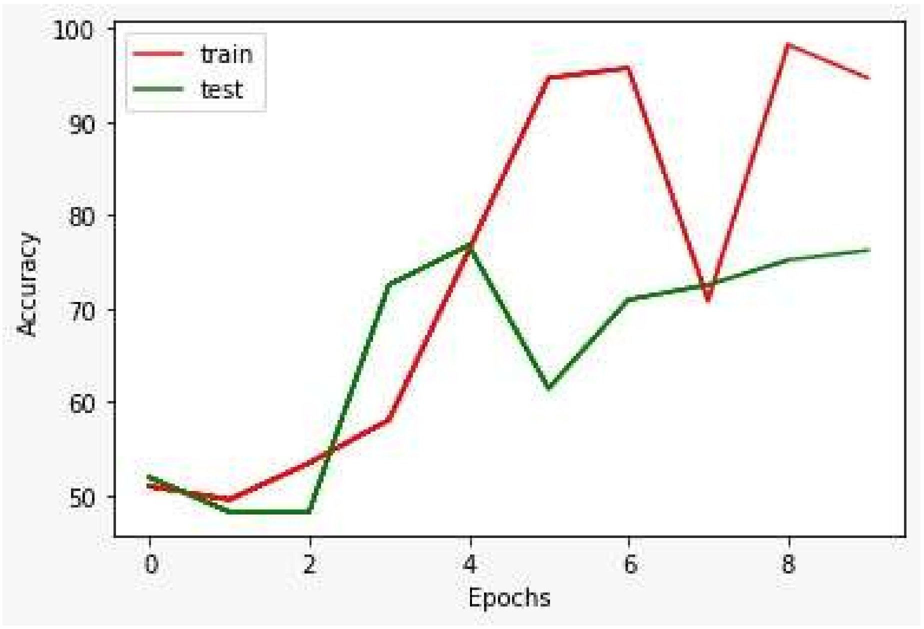


Fig 4.2. Graph depicting accuracy, validation accuracy, loss and validation loss over multiple epochs

The final accuracy with the resnet model while testing is 0.7208723421981679 (72.09%).

3.2 AlexNet

This is the JavaScript Object representing the accuracy and loss of the AlexNet Model.

```
"0-10": {
    "inLoss": [
      0.7038170880243659, 0.6930449021668621, 0.6927556230080085,
      0.6937197682272073, 0.6934835749588827, 0.6923972921863569,
      0.6937763277321948, 0.6833371921273419, 0.6930004186477525,
      0.6837705390832761
    ],
    "trainAcc": [
      51.245551601423486, 50.1779359430605, 53.380782918149464,
      53.380782918149464, 51.95729537366548, 53.380782918149464,
      53.380782918149464, 54.09252669039146, 49.11032028469751,
      56.58362989323843
    ],
    "testLoss": [
      1.7409996078719223, 1.7271149366620988, 1.729944820441897,
      1.7253038501613354, 1.7283374288725475, 1.7243495684452157,
      1.7241112548838218, 1.708699175654344, 1.7236164376218484,
      1.7177339366651954
    ],
    "testAcc": [
      48.73139383490597, 47.997395723068166, 49.096909990733714,
      48.11446282200495, 47.48887418751436, 47.23522455500937,
      48.781711941063726, 51.47819519032348, 48.794508071323065,
      47.78338640942724
    ]
 }
```

Below, we have accuracy v/s epoch graph, which tells us about the variation of accuracy and loss with increase in number of epochs.

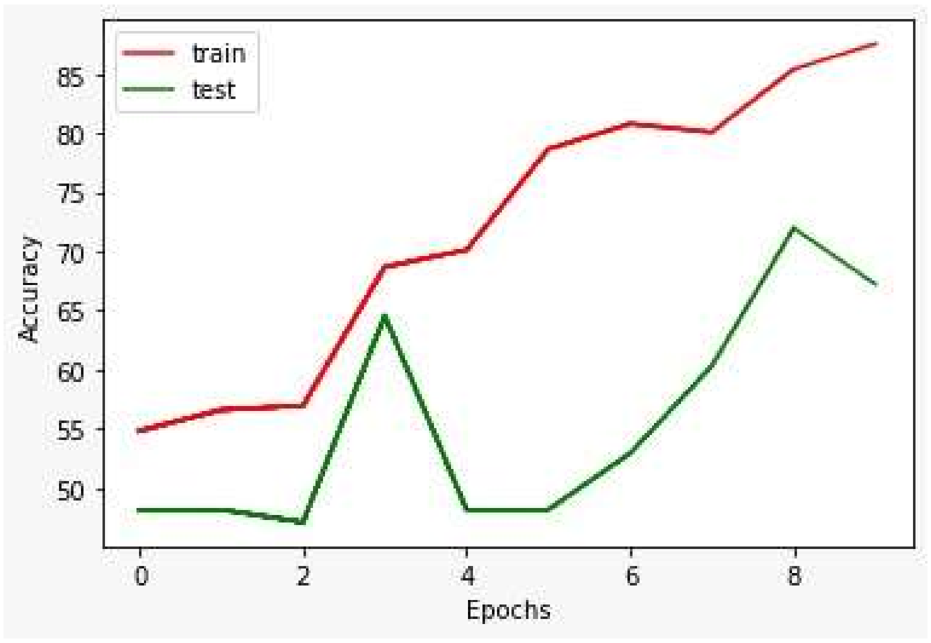


Fig 4.3. Graph depicting accuracy, validation accuracy, loss and validation loss over multiple epochs

The final accuracy with the alexnet model while testing is 0.744527566163736 (74.45%).

CONCLUSIONS

Automated Writer Verification seems to be necessary in current times, an effective implementation of writer verification can be of great help in the field of forensics, signature analysis, historical analysis and many more.

In this project, we have worked on writer verification for Bengali language. We have used a unique dataset which contains writings from more than 100 volunteers. The dataset was pre-processed by Spandan et al [12]. We have extracted the features from the pre-processed dataset and have performed pairwise writer verification. i.e., verify whether a piece of text is written by writer_1 or writer_2 for Bangla language using VGG16, and have achieved an accuracy of 62.75%.

Model	**Number of Layers**	**Number of WriterPairs**	**Final Accuracy**
VGG16	16	100	62.75%
ResNet	34	10	72.09%
AlexNet	12	10	74.45%

Table b. Comparison between various models in present work

We notice the accuracies from the various models that we used in the present work. We observe that ResNet and AlexNet do significantly better than VGG16. While this is a surprising result, the significant improvement can be attributed to the fact that VGG16 was originally designed recognise objects in images [2], whereas AlexNet is an algorithm that work with features of images themselves [1] and do not look for object resemblences, that an abstract image like handwriting lacks. ResNet in itself is a general-purpose neural network. This is why, while it does worse than AlexNet, it does better than VGG16.

More work is clearly required here to understand the various methods through which the accuracy for this task can be increased.

REFERENCES

[1] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet classification with deep Convolutional Neural Networks”, *Communications of the ACM*, Vol. 60 Issue 6, pp. 84–90, Jun 2017, doi: 10.1145/3065386.

[2] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, in *Proc. International Conference on Learning Representations (ICLR 2015)*, 2015 [Online], Available: http://arxiv.org/abs/1409.1556.

[3] A. Rehman, S. Naz, M. I. Razzak and I. A. Hameed, "Automatic Visual Features for Writer Identification: A Deep Learning Approach," *IEEE Access*, Vol. 7, 2019, pp. 17149-17157, Jan. 21, 2019, doi: 10.1109/ACCESS.2018.2890810.

[4] J. John, Pramod K. V. and K. Balakrishnan, “Handwritten Character Recognition of South Indian Scripts: A Review”, in *Proc. National Conference on Indian Language Computing*, Feb. 19-20, 2011.

[5] M. A. Hasnat, S. M. Habib, M. Khan, Eds., “A High Performance Domain Specific Ocr For Bangla Script”, *Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics*, Dordrecht: Springer, 2008, doi: 10.1007/978-1-4020-8737-0_31.

[6] J. Paul, A. Roy and A. Sarkar, “Bangla character recognition based on Mobilenet v1 and Inception v3”, in *Proc. International Conference on Emerging Technologies for Sustainable Development (ICETSD ‘19)*, Mar. 5-6, 2019, pp. 511-514.

[7] J. Paul, A. Dattachaudhuri and A. Sarkar, “CNN implementation based on Bangla numeral character recognition”, in *Proc. International Conference on Emerging Technologies for Sustainable Development (ICETSD ‘19)*, Mar. 5-6, 2019, pp. 520-523.

[8] V. Christlein, D. Bernecker, A. Maier, and E. Angelopoulou, ‘‘Offline writer identification using convolutional neural network activation features,’’ In Proc. German Conf. Pattern Recognition, 2015, pp. 540–552. DOI:10.1007/978-3-319-24947-6_45.

[9] A. Schlapbach and H. Bunke, ‘‘Writer identification using an HMM-based handwriting recognition system: To normalize the input or not,’’ in *Proc. Conf. IGS, 2005*, pp. 138–142.

[10] X. Wu, Y. Tang and W. Bu, "Offline Text-Independent Writer Identification Based on Scale Invariant Feature Transform," in *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 3, pp. 526-536, March 2014, DOI: 10.1109/TIFS.2014.2301274.

[11] C. Adak, B. B. Chaudhuri and M. Blumenstein, “An Empirical Study on Writer Identification and Verification from Intra-Variable Individual Handwriting”, *IEEE Access*, Vol. 7, 2021, pp. 24738-24758, Feb 18, 2019, doi: 10.1109/ACCESS.2019.2899908.

[12] T. Mondal, S. A. Hossain, S. Mondal, R. Afroz and A. Hossain, “Preprocess the handwritten document image for preparing writer recognition”, Government College of Engineering and Leather Technology, Kolkata, India, Project Report, June 2020.

[13] R. Thakur, “Step by step VGG16 implementation in Keras for beginners”, *towardsdatascience.com*, Aug. 6, 2019. [Online]. Available: https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c [Accessed Mar. 3, 2021].

[14] Kaggle, Inc., “Dogs vs. Cats: Create an algorithm to distinguish dogs from cats”, Kaggle, Inc., Available: https://www.kaggle.com/c/dogs-vs-cats/data.

[15] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition”, in *Proc. IEEE conference on computer vision and pattern recognition 2016*, 2016, pp. 770-778, [Online], Available: https://arxiv.org/abs/1512.03385

[16] D. Gershgorn, “The data that transformed AI research—and possibly the world”, *The Quartz*, para. 38, July 26, 2017, [Online], Available: https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/, [Accessed June, 7, 2021].

[17] Stanford Vision Lab, Stanford University, Princeton University, “Large Scale Visual Recognition Challenge 2012 (ILSVRC2012)”, *Stanford Vision Lab, Stanford University, Princeton University*, [Online], Available: https://image-net.org/challenges/LSVRC/2012/results.html