Devanagari Handwritten Character Recognition based on Convolutional Neural Networks

Mini Project submitted in partial fulfilment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by

G. Eeshapriya 221710308016

Under the esteemed guidance of

Mr. Joshi Vinay Kumar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM

(Deemed to be University)

HYDERABAD

DECEMBER 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM

(Deemed to be University)

DECLARATION

We hereby declare that the mini project entitled "**Devanagari Handwritten Character Recognition based on Convolutional Neural Networks**" is an original work done in the Department of Computer Science and Engineering, GITAM School of Technology, GITAM (Deemed to be University) submitted in partial fulfilment of the requirements for the award of the degree of B.Tech. in Computer Science and Engineering. The work has not been submitted to any other college or University for the award of any degree or diploma.

Date: 10-12-2020

G. Eeshapriya **221710308016**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM

(Deemed to be University)

CERTIFICATE

This is to certify that Mini Project entitled "**Devanagari Handwritten Character Recognition based on Convolutional Neural Networks**" is submitted by **G. Eeshapriya (221710308016), Perisetla Hari Naga Sai (221710308043), Thaniparthi Likhitha (221710308057), Palnati Architha Goud (221710308041)** in partial fulfillment of the requirements for the award of degree of Bachelor of Technology in Computer Science and Engineering. The Mini Project has been approved as it satisfies the academic requirements.

Mr. Joshi Vinay Kumar
Assistant Professor
Dept. of Computer
Science and Engineering

ACKNOWLEDGMENT

Our Mini Project would not have been successful without the help of several people. we would like to thank the personalities who were part of our seminar in numerous ways, those who gave us outstanding support from the birth of the project.

We are extremely thankful to our honourable Pro-Vice Chancellor, **Prof. N. Siva Prasad** for providing necessary infrastructure and resources for the accomplishment of our project.

We are highly indebted to **Prof. N. Seetharamaiah**, Principal, School of Technology, for his support during the tenure of the Project.

We are very much obliged to our beloved **Prof. S. Phani Kumar**, Head of the Department of Computer Science & Engineering for providing the opportunity to undertake this project and encouragement in completion of this project.

We hereby wish to express our deep sense of gratitude to **Dr. S Aparna**, Assistant Professor, Department of Computer Science and Engineering, School of Technology and to **Mr. Joshi Vinay Kumar**, Assistant Professor, Department of Computer Science and Engineering, School of Technology for the esteemed guidance, moral support and invaluable advice provided by them for the success of the project.

We are also thankful to all the staff members of Computer Science and Engineering department who have cooperated in making our project a success. We would like to thank all our parents and friends who extended their help, encouragement and moral support either directly or indirectly in our project.

Sincerely,

G. Eeshapriya 221710308016

TABLE OF CONTENTS

CONTENT **PAGE NO**

ABSTRACT

Images of handwritten digits are unique in relation to natural pictures as the orientation of a digit, as well as the similitude of features of various digits, makes confusion. On the other hand, deep convolutional neural systems are making tremendous progress in computer vision problems, especially in image classification. Regardless of triumphs in different areas, DCNNs have seen remarkable progress in image classification tasks, setting the state of the art on several challenging classification benchmarks and dominating numerous image classification related challenges and competitions.

Here, we propose a task-oriented model called Devanagari handwritten character recognition based on densely connected convolutional neural networks. It is based on the article, BDNet: Bengali handwritten numerical digit recognition model that has achieved the test precision of 99.78% (baseline was 99.58%) on the test dataset of ISI Bengali handwritten numerals.

Inspired from the same, we are utilizing the concepts and the techniques to classify (recognize) Devanagari handwritten characters. During training, untraditional data preprocessing and augmentation techniques are utilized so that the trained model works on a distinct dataset.

LIST OF FIGURES

FIGURES **PAGE NO**

1. INTRODUCTION

In the last few years, deep learning approaches have been successfully applied to various areas such as image classification, speech recognition, cancer cell detection, video search, face detection, satellite imagery, recognizing traffic signs and pedestrian detection, etc. The outcome of deep learning approaches is also prominent, and in some cases the results are superior to human experts in the past years.

Most of the problems are also being re-experimented with deep learning approaches with the view to achieving improvements in the existing findings. Different architectures of deep learning have been introduced in recent years, such as convolutional neural networks, deep belief networks, and recurrent neural networks.

The entire architecture has shown the proficiency in different areas. Character recognition is one of the areas where machine learning techniques have been extensively experimented. The first deep learning approach, which is one of the leading machine learning techniques, was proposed for character recognition in 1998 on MNIST database.

The deep learning techniques are basically composed of multiple hidden layers, and each hidden layer consists of multiple neurons, which compute the suitable weights for the deep network. A lot of computing power is needed to compute these weights, and a powerful system was needed, which was not easily available at that time.

Since then, the researchers have drawn their attention to finding the technique which needs less power by converting the images into feature vectors. In the last few decades, a lot of feature extraction techniques have been proposed such as HOG (histogram of oriented J. Imaging 2018, 4, 41; doi:10.3390/jimaging4020041 www.mdpi.com/journal/jimaging J. Imaging 2018, 4, 41 2 of 14 gradients), SIFT (scale-invariant feature transform), LBP (local binary pattern) and SURF (speeded up robust features). These are prominent feature extraction methods, which have been experimented for many problems like image recognition, character recognition, face

detection, etc. and the corresponding models are called shallow learning models, which are still popular for the pattern recognition.

Feature extraction is one type of dimensionality reduction technique that represents the important parts of a large image into a feature vector. These features are handcrafted and explicitly designed by the research community. The robustness and performance of these features depend on the skill and the knowledge of each researcher.

There are the cases where some vital features may be unseen by the researchers while extracting the features from the image and this may result in a high classification error. Deep learning inverts the process of handcrafting and designing features for a particular problem into an automatic process to compute the best features for that problem.

A deep convolutional neural network has multiple convolutional layers to extract the features automatically. The features are extracted only once in most of the shallow learning models, but in the case of deep learning models, multiple convolutional layers have been adopted to extract discriminating features multiple times. This is one of the reasons that deep learning models are generally successful.

The LeNet is an example of deep convolutional neural network for character recognition. Recently, many other examples of deep learning models can be listed such as AlexNet , ZFNet , VGGNet and spatial transformer networks . These models have been successfully applied for image classification and character recognition. Owing to their great success, many leading companies have also introduced deep models.

Google Corporation has made a GoogLeNet having 22 layers of convolutional and pooling layers alternatively. Apart from this model, Google has also developed an open-source software library named Tensorflow to conduct deep learning research. Microsoft also introduced its own deep convolutional neural network architecture named ResNet in 2015. ResNet has 152-layer network architectures which made a new record in detection, localization, and classification.

This model introduced an idea of CNN model. Character recognition is a field of image processing where the image is recognized and converted into a machine-readable format. As discussed above, the deep learning approach and especially convolutional neural networks have been used for image detection and recognition. It has also been successfully applied on Bengali language. In this work, a deep convolutional neural network is applied for handwritten Devanagari characters.

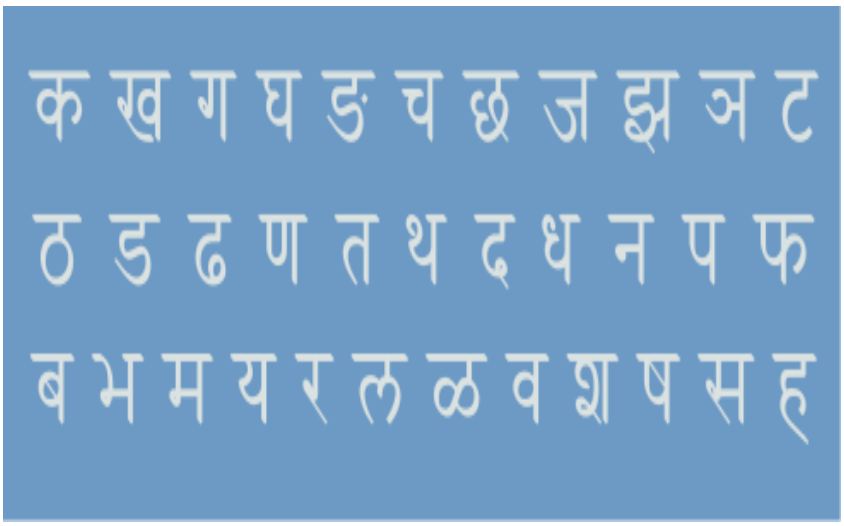

Fig 1. Devanagari Characters

Handwritten text recognition is the process of automatic conversion of handwritten text into machine-encoded text. It has been a popular research area for many years due to various applications such as digitizing handwritten manuscripts, postal automation etc. When considering a limited lexicon size or a limited number of writers, robust solutions such as have already been found. However, not enough work has been done in the space of unconstrained text recognition in Indic scripts.

In this work, we address the challenges associated with creating a handwritten word recognizer for the most popular Indic script – Devanagari. Devanagari is the most popular Indic script, used by nearly 400 million people in northern India. It is used to write many languages such as Hindi, Sanskrit, etc. It is read from left to right.

One of the main reasons for the neglect of Devanagari and other Indic scripts in the field of handwriting recognition is the lack of a large publicly available

handwriting datasets, which inhibits the training of modern deep learning architectures. These modern architectures contain millions of parameters and require substantial amount of data for training.

1.1. Purpose of Project

To recognize handwritten Devanagari characters and numerals using CNN. We propose a task-oriented model called Devanagari handwritten character recognition based on densely connected convolutional neural networks to recognize handwritten characters

1.2. Overview of Project

- We will be using Dataset taken from Kaggle to perform the Modelling and classification.
- Original dataset consists of 46 classes of approx. 2000 images.

1.3. About Devanagari

Devanagari also called Nagari is a left-to-right alpha syllabary based on the ancient Brāhmī script used in the Indian subcontinent. It was developed in ancient India from the 1st to the 4th century CE and was in regular use by the 7th century CE. The Devanagari script, composed of 47 primary characters including 14 vowels and 33 consonants, is the fourth most widely adopted writing systemin the world, being used for over 120 languages.

The orthography of this script reflects the pronunciation of the language. Unlike the Latin alphabet, the script has no concept of letter case. It is written from left to right, has a strong preference for symmetrical rounded shapes within squared outlines, and is recognisable by a horizontal line, known as a Shiro Rekha, that runs along the top of full letters. In a cursory look, the Devanagari script appears different from other

Brahmic scripts such as Bengali, Odia or Gurmukhi, but a closer examination reveals they are very similar except for angles and structural emphasis.

Among the languages using it – as either their only script or one of their scripts – are Marathi, Pāḷi, Sanskrit (the ancient Nagari script for Sanskrit had two additional consonantal characters), Hindi, Nepali, Sherpa, Prakrit, Apabhramsa, Awadhi, Bhojpuri, Braj Bhasha, Chhattisgarhi, Haryanvi, Magahi, Nagpuri, Rajasthani, Bhili, Dogri, Maithili, Kashmiri, Konkani, Sindhi, Bodo, Nepalbhasa, Mundari and Santali. The Devanagari script is closely related to the Nandinagari script commonly found in numerous ancient manuscripts of South India, and it is distantly related to a number of southeast Asian scripts.

1.4. History

Devanagari is part of the Brahmic family of scripts of India, Nepal, Tibet, and Southeast Asia. It is a descendant of the 3rd century BCE Brahmi script, which evolved into the Nagari script which in turn gave birth to Devanagari and Nandinagari. Both were used to write Sanskrit, until the latter was merged into the former. The resulting script is widely adopted across India to write Sanskrit, Marathi, Hindi and its dialects, and Konkani.

Some of the earliest epigraphical evidence attesting to the developing Sanskrit Nagari script in ancient India is from the 1st to 4th century CE inscriptions discovered in Gujarat. Variants of script called Nāgarī, recognisably close to Devanagari, are first attested from the 1st century CE Rudradaman inscriptions in Sanskrit, while the modern standardised form of Devanagari was in use by about 1000 CE.

Medieval inscriptions suggest widespread diffusion of the Nagari-related scripts, with biscripts presenting local script along with the adoption of Nagari scripts. For example, the mid-8th-century Pattadakal pillar in Karnataka has text in both Siddha Matrika script, and an early Telugu-Kannada script; while, the Kangra Jawalamukhi inscription in Himachal Pradesh is written in both Sharada and Devanagari scripts.

The Nagari script was in regular use by the 7th century CE, and it was fully developed by about the end of first millennium. The use of Sanskrit in Nagari script in medieval India is attested by numerous pillar and cave temple inscriptions, including the 11th-century Udayagiri inscriptions in Madhya Pradesh, and an inscribed brick found in Uttar Pradesh, dated to be from 1217 CE, which is now held at the British Museum.

Fig 1.4.1 Handwritten Characters

The script's proto- and related versions have been discovered in ancient relics outside of India, such as in Sri Lanka, Myanmar and Indonesia; while in East Asia, Siddha Matrika script considered as the closest precursor to Nagari was in use by Buddhists. Nagari has been the primus inter pares of the Indic scripts.

It has long been used traditionally by religiously educated people in South Asia to record and transmit information, existing throughout the land in parallel with a wide variety of local scripts (such as Modi, Kaithi, and Mahajani) used for administration, commerce, and other daily uses.

Sharada remained in parallel use in Kashmir. An early version of Devanagari is visible in the Kutila inscription of Bareilly dated to Vikram Samvat 1049 (i.e., 992 CE), which demonstrates the emergence of the horizontal bar to group letters belonging to a word.

One of the oldest surviving Sanskrit texts from the early post-Maurya period consists of 1,413 Nagari pages of a commentary by Patanjali, with a composition date of about 150 BCE, the surviving copy transcribed about 14th century CE.

1.5. Issues in Devanagari Script Recognition

The Devanagari script consists of 37 consonants and 10 numericals. A horizontal line called the Shirorekha is present in the script from which the characters hang. When multiple characters are written together the Shirorekha gets extended. When a consonant is followed by a vowel in Devanagari script, the shape of consonant character gets modified and such a character is referred as a modified character or modifier.

When a consonant is followed by another consonant and a diacritic called virama, a new character gets formed which has an orthographic shape and is called a compound character. Unlike Latin scripts, Devanagari script has no concept of lowercase and uppercase letters.

Beside the aforementioned characters, there are various characters from ancient languages such as Avestan, Sanskrit, etc. that are part of the Devanagari script in order to ensure compatibility of the script with ancient manuscripts. Even if we ignore modifiers and conjuncts, the number of characters in the script is more than double of the number of characters present in English.

With the inclusion of modifiers and conjuncts, the number of distinct characters in Devanagari script is well over a thousand. In addition to the script level challenges, there are various issues associated with how the Devanagari script is written, in the case of compound characters.

The same compound character can be correctly written in two different ways, one in which the virama is hidden and another where the virama is explicitly written. Any Devanagarihandwriting recognition system, for all possible compound characters, must map both these variations to the same compound character.

Also, a handwritten word recognizer for Devanagari scripts has to deal with challenges associated with recognizing the varying styles of different writers and the cursive nature of the handwriting. The presence of circular arcs in the character shape causes distortions such as skew while writing along with merging up of adjacent characters.

1.6. CNN

In deep learning, a convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, brain-computer interfaces,and financial time series.

CNNs are regularized versions of multilayer perceptrons. Multilayer perceptrons usually mean fully connected networks, that is, each neuron in one layer is connected to all neurons in the next layer. The "fully-connectedness" of these networks makes them prone to overfitting data.

Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. CNNs take a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns. Therefore, on the scale of connectedness and complexity, CNNs are on the lower extreme.

Convolutional networks were inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

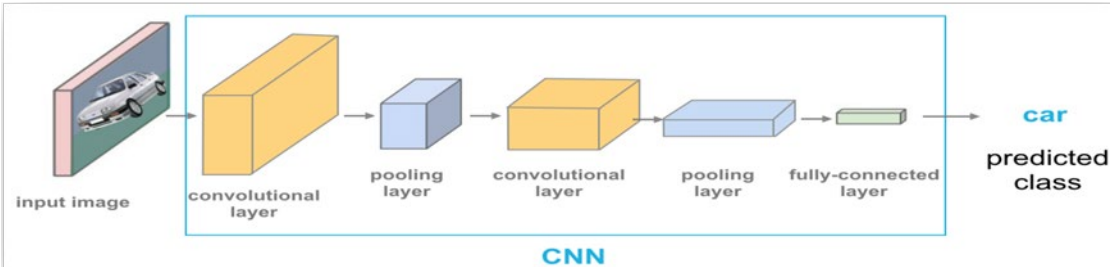

Fig 1.6.1 CNN Basic Overview

Convolutional neural network (CNN), a class of artificial neural networks that has become dominant in various computer vision tasks, is attracting interest across a variety of domains, including radiology. CNN is designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers.

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

CNN is a type of deep learning model for processing data that has a grid pattern, such as images, which is inspired by the organization of animal visual cortex and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns.

CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classification.

A convolution layer plays a key role in CNN, which is composed of a stack of mathematical operations, such as convolution, a specialized type of linear operation. In digital images, pixel values are stored in a two-dimensional (2D) grid, i.e., an array of

numbers, and a small grid of parameters called kernel, an optimizable feature extractor, is applied at each image position, which makes CNNs highly efficient for image processing, since a feature may occur anywhere in the image. As one layer feeds its output into the next layer, extracted features can hierarchically and progressively become more complex.

The process of optimizing parameters such as kernels is called training, which is performed so as to minimize the difference between outputs and ground truth labels through an optimization algorithm called backpropagation and gradient descent, among others.

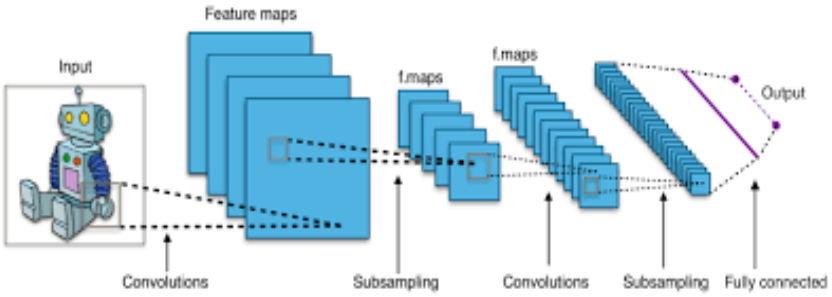

Fig 1.6.2 CNN Overview

2. LITERATURE SURVEY

2.1. Existing works on Bengali handwritten numerical recognition

The Bengali handwritten numeral acknowledgment is one of the most seasoned design acknowledgment issues. Numerous analysts have been working in this field since the 90s of the only remaining centuries (Dutta and Chaudhury, 1993; Pal and Chaudhuri, 2000). Through this subsection, we have surveyed most striking deals with this Bengali transcribed numeral acknowledgment. Subhadip Basu et al. proposed Manually written Bangla Digit Recognition utilizing Classifier Combination Through Dempster-Shafer (DS) Technique (Basu et al., 2005).

They have utilized the DS procedure and MLP classifier for arrangement and likewise utilized 3-overlay cross-approval on the preparation dataset of 6000 manually written examples. Their plan accomplished 95.1% test exactness In Pal et al. (2006) proposed a plan where unconstrained disconnected Bengali transcribed numerals were perceived.

This plan has perceived diverse manually written styles. The plan chooses the necessary highlights utilizing the idea of water flood from the repository, and furthermore gather topological and underlying highlights of the numerals. They applied this plan on their own gathered dataset of size 12000 and got acknowledgment exactness of around 92.8%. U. Bhattacharya and B. B. Choudhury introduced a transcribed numeral information base alongside the Devanagari data set and proposed a classifier model (Bhattacharya and Chaudhuri, 2009).

Their information base contains 23392 transcribed Bengali numeral pictures. Their classifier model is a multi-stage fell acknowledgment plot where they utilized wavelet-based multi-goal portrayals what's more, multilayer discernment as classifiers. They have referenced 99.14% preparing and 98.20% testing exactness on this dataset.

Cheng-Lin Liu and Ching Y. Suen proposed a benchmark model (Liu and Suen, 2009) on the ISI numeral dataset (Bhattacharya also, Chaudhuri, 2009) alongside a

Farsi numeral information base. They pre-processed the dataset into dark scale pictures and applied numerous customary component extraction models.

This benchmark model accomplished the most noteworthy test exactness of 99.40%. Ying Wen and Lianghua He proposed a classifier model (Wen and He, 2012) for Bengali transcribed numeral acknowledgment. This model attempted to fathom huge dataset high dimensionality issue.

They joined Bayesian discriminant with portion approach with UCI dataset and another dataset, for example, MNIST (Leun, 2020). The pace of blunder is 1.8%, the acknowledgment rate is 99.08% and acknowledgment time is 7.46 ms. Nearby locale distinguishing proof, where ideal unambiguous highlights are separated, is one of the critical undertakings in the field of character acknowledgment.

This thought is embraced by N. Das et al. in their transcribed digit acknowledgment procedure (Das et al., 2012) in view of a hereditary algorithm(GA). GA is applied to seven arrangements of nearby districts. For each set, GA chooses a negligible neighbourhood area bunch with a Support Vector Machine (SVM) based classifier.

The entire digit pictures are utilized for worldwide component extraction while neighbourhood highlights are separated for shape data. The quantity of worldwide highlights is steady while the quantity of nearby highlights depends on the quantity of the nearby districts. The test precision rate was 95.50% for this model.

M. K. Nasir and M. S. Uddin proposed a plot (Nasir and Uddin, 2013) where they utilized K-Means grouping, Bayes' hypothesis and Maximum a Posteriori for highlight extraction, also, for order, SVM is utilized. In the wake of changing over the pictures into parallel qualities, a few focuses are discovered, which was disposed of utilizing a flood fill calculation.

The plinth steps are Clipping, Division, Horizontal, and Vertical Thinning Scan. Here test exactness rate was 99.33%. In Rahman et al. (2015) M. M. Rahaman et

al. proposed a CNN based model. This technique standardizes the composed character pictures and afterward utilized CNN to characterize singular characters.

It doesn't utilize any component extraction strategy like recently referenced works. The significant advances are pre-preparing of crude pictures by changing over them into a dark scale pictures and afterward preparing the model. For this situation, test precision was 85.36%. On another paper, we have seen the presence of auto-encoder for unaided pre-preparing through Deep CNN which comprises of more than one shrouded layer with 3 convolutional layers.

Each layer was trailed by 2 * 2 max-pooling layer. This plan Shopon et al. (2017) was proposed by Md Shopon et al. The layers have 32 * 3 * 3 number of portions. In a similar way, the decoder has a design with each convolutional layer with 5 neurons, as opposed to 32. The ReLU (Maas et al., 2013) initiation is available in all layers.

For preparing purposes, the model upgraded the preparation dataset by haphazardly turning each picture between 0 degrees and 50 degrees and furthermore by moving vertically by an arbitrary sum between 0 pixels and 6 pixels. This model was prepared in 3 different arrangements SCM, SCMA, and ACMA. They have accomplished a test exactness of 99.50%.

Another model (Akhand et al., 2016), proposed by M. A. H. Akhand et al. utilized pre-handling by utilizing a straightforward pivot based methodology to create examples and it additionally makes all pictures of ISI manually written information base into a similar goal, measurement, and size. CNN structure of this model has two convolutional layers with 5 * 5 estimated nearby open fields and two sub-testing layers with 2 * 2 measured nearby averaging regions alongside info and yield layers.

The info layer contains 784 responsive fields for 28 * 28 pixels picture. The first convolutional activity produces six element maps. Convolution activity with portion spatial element of 5 diminishes 28 spatial measurement to 24 (i.e., 28 + 1 - 5) spatial measurement.

In this way, every first level element map size is 24 * 24. The precision pace of the testing is 98.45% on ISI transcribed Bengali numerals. In Choudhury et al. (2018), A. Choudhury et al. proposed a histogram of arranged inclination (Hoard) and shading histogram for the choice of highlight calculations.

Here, Hoard is utilized as the component set to speak to every numeral thing at the component space and SVM is utilized to deliver the yield from input. The test exactness of this calculation is 98.05% on CMATERDB 3.1.1 dataset (which is a benchmark Bengali manually written numeral information base made by CMATER lab of Jadavpur College, India).

M. M. Hasan et al. proposed a Bengali manually written digit acknowledgment model dependent on ResNet (He et al., 2016). Their gathering model from their six best models, applied on the NumtaDB dataset (Alam, 2020), accomplished 99.3359% test precision. In Noor et al. (2018) R. Noor et al. proposed a gathering model based Convolutional Neural Organization for perceiving Bengali transcribed numerals.

They train their model in numerous uproarious conditions utilizing altered NumtaDB dataset (Alam, 2020). In all cases, their model accomplished over 96% test exactness on this NumtaDB dataset. An ongoing Bengali manually written numeral acknowledgment work (Rabby et al., 2019) was there, proposed by AKM S. A. Rabby et.al. Here creators utilized a lightweight CNN model to characterize the manually written numeral digits.

This model is prepared utilizing ISI manually written Bengali numeral (Bhattacharya and Chaudhuri, 2009) and CMATERDB 3.1.1 information bases with 20% information for approval. Their work accomplished approval and test exactness as 99.74% and 99.58% on ISI manually written numerals, which is the past best exactness on that dataset.

Their model likewise shows great precision on other benchmark datasets, for example, CMATERDB 3.1.1 and so on M.S. Islam et.al proposed Bayanno-Net (Islam et al., 2019), where they attempted to perceive Bangla transcribed numerals utilizing CNN. They dealt with NumtaDB dataset (Alam, 2020). Their model accomplished 97%

exactness. An ongoing audit article (Hoq et al., 2020) composed by M.N. Hoq et.al given a similar diagram of existing characterization calculations to perceived Bengali manually written numerals.

2.2. Advancements of deep learning for image classification

After the achievement of AlexNet (Krizhevsky et al., 2012), a profound learning-based model for picture arrangement, numerous analysts moved to this zone of exploration of PC vision and example acknowledgment (Ghosh et al., 2019). Subsequently, numerous progressive cutting edge models came surprisingly close to time since 2012 (Sultana et al., 2018; Lim and Keles, 2018). In this subsection, we quickly checked on the improvement of profound adapting particularly Convolutional Neural Organizations in the field of picture arrangements.

CNN is an exceptional kind of multi-layer neural organization roused by the vision component of the creature (Ghosh et al., 2019). Hubel and Wiesel tested and said that visual cortex cells of creature distinguish light in the little responsive field (Fukushima, 1980).

Kunihiko Fukushima got inspiration from this examination and proposed a multi-layered neural organization, called NEOCOGNITRON (Hubel and Wiesel, 1968), fit for perceiving visual examples progressively through learning. This model is considered as the motivation for CNN.

A traditional CNN model is made out of one or more squares of convolutional and sub-inspecting or pooling layer, at that point single or various completely associated layers, and a yield layer work as appeared in Fig. 2. The advantages of utilizing CNN are mechanized highlights extraction, boundary sharing and some more (Lecun et al., 1998; Krizhevsky et al., 2012; Nanni et al., 2017).

Old style CNN is adjusted from numerous points of view as indicated by target area (Sultana et al., 2018; Liang et al., 2017; Baldominos et al., 2018; Sultana et al., 2019). Yann LeCun et al. presented the principal complete CNN model, called LeNet-

5 (Lecun et al., 1998), to order English transcribed digit pictures. It has 7 layers among which 3 convolutions, 2 normal pooling, 1 completely associated, and 1 yield layer.

They utilized SIGMOID capacity as the enactment work for non-linearity before a normal pooling layer. The yield layer utilized Euclidean Spiral Premise Function (RBF) for characterization of MNIST (Leun, 2020) dataset. The loads of each layer were prepared utilizing the back-spread calculation (Rumelhart et al., 1986).

AlexNet (Krizhevsky et al., 2012) was the main CNN based model which won the ILSVRC challenge (Russakovsky et al., 2015) in 2012 with a critical decrease of mistake. AlexNet's blunder rate was 16.4% while the second-best mistake rate was 26.17%. This model was proposed by Alex Krizhevsky et al. what's more, it is prepared by utilizing ImageNet dataset (Deng et al., 2009), this dataset contains 15 million high goal marked pictures more than 22 thousand classifications.

AlexNet has 11 teachable layers, and the structure is nearly like LeNet-5, however here max-pooling utilized rather than the normal pooling, ReLU actuation instead of the SIGMOID capacity, softmax work instead of RBF, and 11 * 11 instead of 5 * 5 channel size in the principal layer. Furthermore, unexpectedly dropout system, Srivastava et al. (2014) and GPU were utilized to prepare the model. In Zeiler and Fergus (2013) Zeiler and Fergus introduced ZFNet which was the victor of the ILSVRC challenge in 2013.

The structure squares of ZFNet are practically like AlexNet with barely any progressions, for example, the principal layer channel size is 7 * 7 rather than 11 * 11 in AlexNet. Creators of ZFNet clarified how CNN functions with the assistance of Deconvolutional Neural Organizations (DeconvNet). DeconvNet is the exact inverse of CNN. The blunder pace of ZFNet was 11.7%. K. Simonyan and A. Zisserman proposed VGGNet (Simonyan and Zisserman, 2014), which resembles a more profound rendition of AlexNet.

Here, creators utilized little channels 3 * 3 sizes for all layers. They have utilized an absolute 6 diverse CNN designs with various weight layers. This VGGNet made sure about second spot in ILSVRC challenge in 2014 with a mistake pace of 7.3%

simply 0.6% more than the mistake pace of the champ GoogLeNet (Szegedy et al., 2015).

Google- Net, Going Further with Convolutions (Szegedy et al., 2015), is proposed by Christian Szegedy et al. which was an examination group of Google. The Structure of GoogLeNet is unique in relation to customary CNN, it is more extensive and more profound than past models yet computationally effective.

Through beginning engineering, various equal channels with various sizes are utilized, and for this, issues of disappearing inclination and over-fitting were handled. Completely associated layers are not utilized in GoogLeNet yet the normal pooling layer is utilized prior to the classifier.

This model won the ILSVRC challenge 2014 with a mistake pace of 6.7%. The expanding layer could give more exactness yet will experience the ill effects of disappearing inclination issue. To handle this issue, Kaiming He et al. from Microsoft Exploration proposed ResNet (He et al., 2016). ResNet is a profound model where each layer has a leftover square with a skip association with the layer previously the past layer.

ResNet is the champ of the ILSVRC challenge with a blunder pace of 3.57% and this is an achievement of past the human level. Gao Hunag et al. proposed DenseNet (Huang et al., 2017), where each layer is associated with all past layers of the model. DenseNet beats the disappearing inclination issue just as it gathers required highlights, everything being equal, and engenders to all progressive layers in feed-forward styles for highlights reuse.

Consequently, this model requires less number of boundaries to accomplish exactness, so it is computationally productive. Motivated by the achievement of ResNet, Jie Hu et al. proposed SENet (Hu et al., 2018) with the primary concentration to build channel relationship between progressive layers. SENet has added "Press and-Excita tion" (SE) block into each square (ResNet Square), and for this, the model adaptively re-adjusts channel-wise element reactions between channels.

SENet has won the ILSVRC-2017 test with a blunder pace of 2.252%. As of late another idea in convolutional neural organizations proposed through CapsNet (Sabour et al., 2017). Here sub-testing ideas are annulled and characterization is conceivable of pictures of various directions. In spite of the fact that the calculation is not light for dynamic steering among containers but rather it is amazingly promising (Patrick et al., 2019).

2.3. Existing works on Devanagari handwritten character recognition

Devanagari handwritten character recognition has been investigated by different feature extraction methods and different classifiers. Researchers have used structural, statistical and topological features. Neural networks, KNN (K-nearest neighbours), and SVM (Support vector machine) are primarily used for classification.

However, the first research work was published by I. K. Sethi and B. Chatterjee in 1976. The authors recognized the handwritten Devanagari numerals by a structured approach which found the existence and the positions of horizontal and vertical line segments, D-curve, C-curve, left slant and right slant. A directional chain code-based feature extraction technique was used by N. Sharma.

A bounding box of a character sample was divided into blocks and computed 64-D direction chain code features from each divided block, and then a quadratic classifier was applied for the recognition of 11,270 samples. The authors reported an accuracy of 80.36% for handwritten Devanagari characters. Deshpande et al. used the same chain code features with a regular expression to generate an encoded string from characters and improved the recognition accuracy by 1.74%.

A two-stage classification approach for handwritten characters was reported by S. Arora where she used structural properties of characters like shirorekha and spine in the first stage and in another stage used intersection features.

These features further fed into a neural network for the classification. She also defined a method for finding the shirorekha properly. This approach has been tested on 50,000 samples and obtained 89.12% accuracy. In, S. Arora combined different features such as chain codes, four side views, and shadow-based features. These features were fed into a multilayer perceptron neural network to recognize 1500 handwritten Devanagari characters and obtain 89.58% accuracy.

A fuzzy model-based recognition approach has reported by M. Hanmandlu. The features areextracted by the box approach which divided the character into 24 cells (6 × 4 grid), and a normalized vector distance for each box was computed except the empty cells. A reuse policy is also used to enhance the speed of the learning of 4750 samples and obtained 90.65% accuracy.

The work presented in computed shadow features, chain code features and classified the 7154 samples using two multilayer perceptrons and a minimum edit distance method for handwritten Devanagari characters. They reported 90.74% accuracy.

Kumar has tested five different features named Kirsch directional edges, chain code, directional distance distribution, gradient, and distance transform on the 25,000 handwritten Devanagari characters and reported 94.1% accuracy. During the experiment, he found the gradient feature outperformed the remaining four features with the SVM classifier, and the Kirsch directional edges feature was the weakest performer.

A new kind of feature was also created that computed total distance in four directions after computing the gradient map and neighbourhood pixels' weight from the binary image of the sample. In the paper, Pal applied the mean filter four times before extracting the direction gradient features that have been reduced using the Gaussian filter.

They used modified quadratic classifier on 36,172 samples and reported 94.24% accuracy using cross-validation policy. Pal has further extended his work with SVM and MIL classifier on the same database and obtained 95.13% and 95.19% recognition

accuracy respectively. Despite the higher recognition rate achieved by existing methods, there is still room for improvement of the handwritten Devanagari character recognition.

The basic pattern recognition ideas, understanding of various research models and related algorithms for classification and clustering are introduced in. The paper presents the algorithms for the classification, regression, clustering, parsing and sequence labeling on pattern recognition.

There are three steps in segmentation: Line, Word and Character detection and segmentation. In, horizontal projection profile method for line, vertical projection profile method for word and both horizontal and vertical projection profile method to segment characters from words. In, new segmentation techniques are suggested, in which scanned handwritten image is divided into lines, these lines to words and words to characters.

The feature and template-based method was applied for older Devanagari character recognition. In template-based method, each input letter is matched with a standard template pattern of characters and similarity between two patterns is used to decide which letter it is. The results of older OCRs were enhanced by making use of feature-based method in addition to traditional template-based method.

Feature-based methods find the unique aspect of letters and these aspects are then used in classification.The performance of recognition system is improved by using principal component analysis and Linear Discriminant Analysis . In which, chain coding, edge detection and direction feature techniques are used for extraction of raw features which are then reduced by LDA and PCA. The SVM is employed for classification of characters.

The new techniques of Devanagari OCR such as training, feature extraction, classification and matching are presented in [29]. A comparative study concerning feature extraction approaches and classifiers on Devanagari OCR is explained.

In, Devanagari handwritten characters are recognized by use of convolution neural networks (CNN). The architecture consisting of 7 convolution and two fully connected layers has obtained highest accuracy of 96.10%. Authors discussed, Euclidean distance based KNN techniques for feature extraction, which achieved higher recognition rate than SVM.

In, curvatures and gradient features are extracted and applied on Euclidean distance Neighbour based Kohonen NN. In, features are detected from lines and curves by using Hough transform and classification is accomplished by SVM. The accuracy secured from these two methods is up to 90%.

Fuzzy technique is used in , for recognition of Hindi handwritten characters and secured accuracy of 90.65 percent for handwritten Devanagari characters. In, Hindi OCR is developed by removal of shirorekha in pre-processing stage, K – means clustering approach is used for feature extraction and linear kernel based technique is used for classification.

An OCR system is presented in, which classifies and modifies ShirorekhaLess character of handwritten Sanskrit, Hindi and Marathi image documents using support vector machine. The system was developed on various datasets of these languages and achieved better result of 98.35%.

A strong algorithm in Devanagari and Latin scripts for segmentation and recognition is proposed in. In which, primary segmentation paths are acquired through structural property, joined and overlapped characters are segmented using graph distance and then SVM classifier validates the segmentation results.

KNN classifier is utilized for handwritten and printed input characters and obtained recognition rates of 97.05% for Devanagari script and of 97.10% for Latin script respectively. Feature extraction based on array and BPNN for recognition of Marathi handwritten characters is proposed in. The test is performed on 500 handwritten characters obtained from 10 different persons. The recognition accuracy obtained is 92%.

A pre-trained model using transfer learning for DCNN is presented in. Convolutional, pooling and fully connected layers of CNN are applied for feature extraction, reduction of dimensions and image representation. In this work, 15 epochs are implemented for each of 7 pre-trained models.

The recognition results show maximum accuracy of 99% for handwritten Devanagari alphabets. Two deep learning models are used for recognition and to train the dataset in. This work also analyses the effect of dataset increment and dropping out units' approach to prevent over-fitting of the networks. The test results suggest that DCNN with dataset increment technique and added Dropout approach results in accuracy of 98.47%.

Earlier works in the field of Devanagari script recognition were limited to the domain of printed documents. Various methods such as KNN, multi-layer perceptrons were tried out for the recognition task. A summary of these methods can be found in. In this work, we focus on handwritten word images which are more challenging than printed words.

There are three popular ways of building handwritten word recognizers in Devanagari. The first approach consists of segmenting out the various characters and then to use an isolated character (symbol) classifier like SVM's or lately CNN's. In, Roy et al. segment the Devanagari word image into three regions, namely the upper, middle and lower zone, using image processing techniques such as morphological operations and shape matching.

The upper and lower zones are recognized by using support vector machines while the middle zone was recognized using a HMM decoded with a lexicon of middle zone characters. Finally, the results from the recognizers in all three zones are combined.

The second approach for building recognizers is to use segmentation free methods which train on recognizing the whole word or find a holistic representation. The limitation of both first and second method is that they are limited to recognizing a limited size lexicon.

The third and most popular approach, uses Recurrent Neural Networks (RNN), which defines the input as a sequence of feature vectors. They do not require explicit symbol segmentation for recognition and are not bound to recognizing a limited size lexicon.

In this work, we use the above approach, using a CNN-RNN hybrid network and do a sequence-to-sequence transcription. In this work, we present results in both lexicon-based and lexicon-free (unconstrained) setting. Our proposed method gives the state-of-the-art results on the publicly available Devanagari track of the RoyDB dataset.

3. PROBLEM ANALYSIS

There are Certain Challenges involved in this project for example local machine computation limitations, accuracy results, letting the model learn patterns.

3.1. Problem Statement

- Research on the best combination of models that yield high accuracy or better model performance on a data set.
- Also, the model combination that give the least Computational Complexity.
- Implementation of real-time Devanagari handwritten Recognition.

3.2. Software Technologies Used

3.2.1. Python Programming Language

Python is a great object-oriented, interpreted, and interactive programming language. It is often compared to Lisp, Tcl, Perl, Ruby, C#, Visual Basic, Visual Fox Pro, Scheme or Java, and it's much more fun. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high-level dynamic data types, and dynamic typing. There are interfaces to many systems calls and libraries, as well as to various windowing systems. New built-in modules are easily written in C or C++ (or other languages, depending on the chosen implementation). Python is also usable as an extension language for applications written in other languages that need easy-to-use scripting or automation interfaces.

Fig 3.2.1.1 Python Logo

3.2.2. Anaconda

Anaconda is popular because it brings many of the tools used in data science and machine learning with just one install, so it's great for having short and simple setup. Like Virtualenv, Anaconda also uses the concept of creating environments so as to isolate different libraries and versions.

Fig 3.2.2.1 Anaconda Logo

3.2.3. Jupyter Notebook

The Jupyter Notebook is quite useful not only for learning and teaching a programming language such as Python but also for sharing your data JupyterLab incorporates Jupyter Notebook into an Integrated Development type Editor that you run in your browser.

Fig 3.2.3.1 Jupyter

3.3. Hardware Requirements Used

3.3.1. Nvidia CUDA enabled GPU – Nvidia GTX1050Ti

CUDA is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

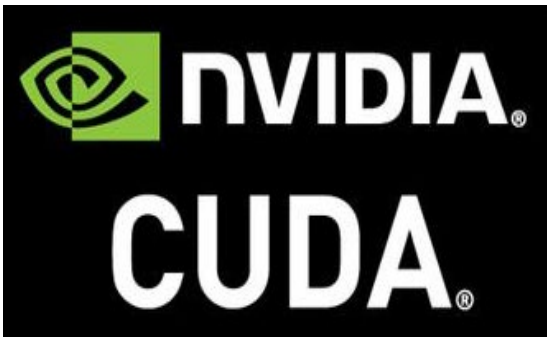

Fig 3.3.1.1 Nvidia CUDA Logo

3.4. Packages Used

3.4.1. TensorFlow

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that 11 lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

Fig 3.4.1.1 TensorFlow Logo

3.4.2. Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research, if you need a deep learning library that:

- Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility.
- Supports both convolutional networks and recurrent networks, as well as combinations of the two.
- Runs seamlessly on CPU and GPU

Fig 3.4.2.1 Keras Logo

3.4.3. Pillow

The most popular and de facto standard library in Python for loading and working with image data is Pillow. Pillow is an updated version of the Python Image Library, or PIL, and supports a range of simple and sophisticated image manipulation functionality. It is also the basis for simple image support in other Python libraries such as SciPy and Matplotlib.

Fig 3.4.3.1 Pillow Logo

3.4.4. NumPy

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open-source project and you can use it freely.

Fig 3.4.4.1 NumPy Logo

3.5. Field of Project

Each of the application areas described above employ a range of computer vision tasks; more or less well-defined measurement problems or processing problems, which can be solved using a variety of methods. Example of typical computer vision tasks are presented below.

Computer vision tasks include methods for acquiring, processing, analysing and understanding digital images, and extraction of high-dimensional data from the given character in order to recognize the character or numerical. This helps in recognizing the handwritten characters.

4. DESIGN

4.1. TKinter

The tkinter package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

Running python -m tkinter from the command line should open a window demonstrating a simple Tk interface, letting you know that tkinter is properly installed on your system, and also showing what version of Tcl/Tk is installed, so you can read the Tcl/Tk documentation specific to that version.

Most of the time, tkinter is all you really need, but a number of additional modules are available as well. The Tk interface is located in a binary module named _tkinter. This module contains the low-level interface to Tk, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

In addition to the Tk interface module, tkinter includes a number of Python modules, tkinter.constants being one of the most important. Importing tkinter will automatically import tkinter.constants, so, usually, to use Tkinter all you need is a simple import statement:

importtkinter

class tkinter.Tk(screenName=None, baseName=None, className='Tk', useTk=1)

The Tk class is instantiated without arguments. This creates a toplevel widget of Tk which usually is the main window of an application. Each instance has its own associated Tcl interpreter.

tkinter.Tcl(screenName=None, baseName=None, className='Tk', useTk=0)

The Tcl() function is a factory function which creates an object much like that created by the Tk class, except that it does not initialize the Tk subsystem. This is most often useful when driving the Tcl interpreter in an environment where one doesn't want to create extraneous toplevel windows, or where one cannot (such as Unix/Linux systems without an X server). An object created by the Tcl() object can have a Toplevel window created (and the Tk subsystem initialized) by calling its loadtk() method.

4.2. TKinter Programming

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

- Import the Tkinter module.
- Create the GUI application main window.
- Add one or more of the above-mentioned widgets to the GUI application.
- Enter the main event loop to take action against each event triggered by the user.

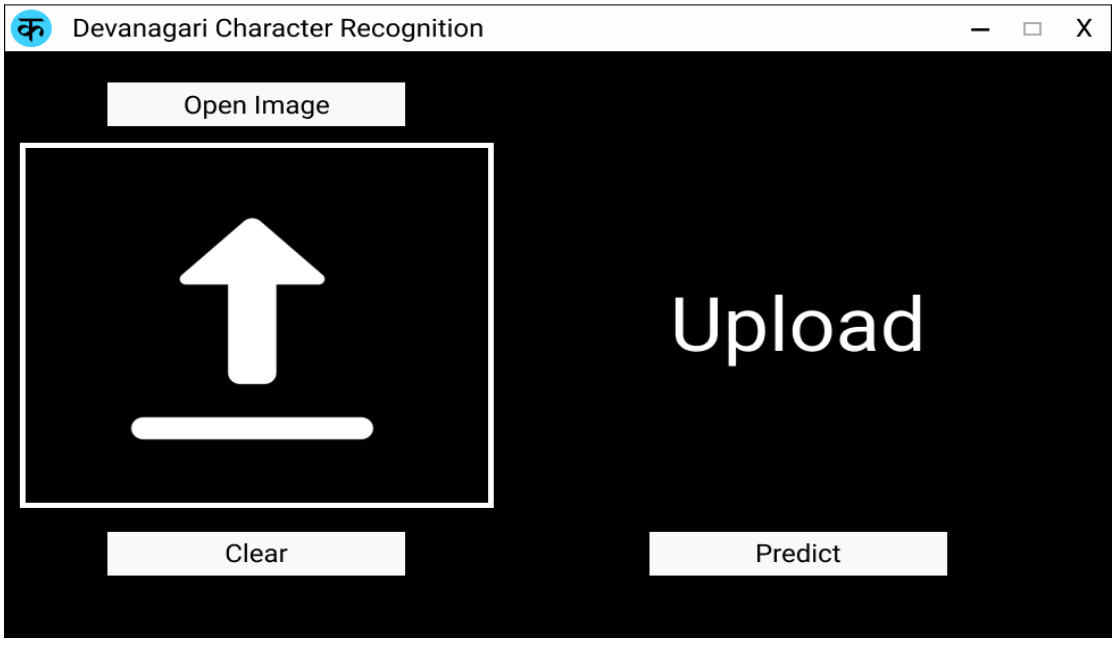

Fig 4.2.1 Devanagari character recognition GUI Application Design Screen - 1

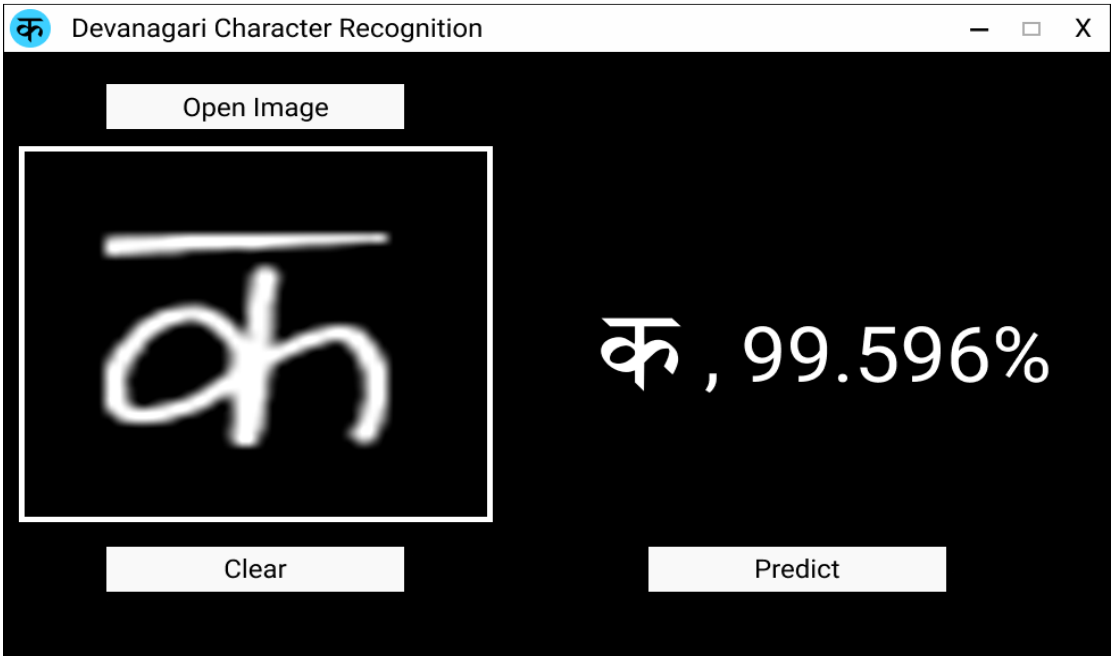

Fig 4.2.2 Devanagari character recognition GUI Application Design Screen - 2

5. IMPLEMENTATION

5.1. Implementation using CNN

Images are in the form of vectors in a csv file. Each row of the image represents a character. Vectors are converted into matrix and reconstructs the image before applying the algorithms. The basic architecture of convolution neural network has the main aspects of the CNN are the convolution layer, activation and subsampling or pooling.

In the convolutional layer each unit is connected to a smaller number of nearby units in the next layer. Each CNN layer looks at an increasingly larger part of the image. Convolutional layers are only connected to pixels in their respective fields. The convolutional layer consists of several filters, which are specifically designed for edge detection. Edges when represented as pixels are a large difference in the actual darkness of two pixels that are next to each other. Each filter detects a different feature.

Filters are visualized with grids system, where there is a filter of weights on a grid which is multiplied with the input and the result obtained is added to provide the output value for the convoluted image. Pooling layer subsamples the input image which reduces the memory use in computer load and reduces the number of parameters. Pooling refers to creating a n x n pool of pixels, known as kernel and evaluating the maximum value which makes it to the next layer.

This maximum value is going to be representative of the entire kernel. The pooling layer will end up removing a lot of information. After various rounds of convolution, activation, and pooling, the resultant is then fed to the last layer which is the fully connected layer for classification.

In fully connected layer, neurons are connected to all activations of the previous layer. A simple matrix multiplication is performed, along with a bias offset. In the fully connected layer, a dropout function is used to reduce the total number of parameters along with over-fitting

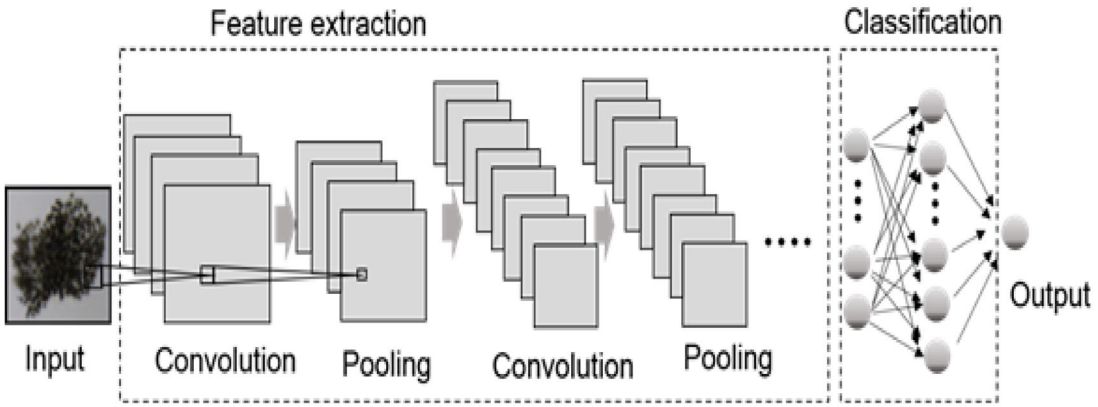

Fig 5.1.1 CNN

5.2. GUI

Graphical user interfaces would become the standard of user-centered design in software application programming, providing users the capability to intuitively operate computers and other electronic devices through the direct manipulation of graphical icons such as buttons, scroll bars, windows, tabs, menus, cursors, and the mouse pointing device. Many modern graphical user interfaces feature touchscreen and voice-command interaction capabilities.

Graphical user interface design principles conform to the model–view–controller software pattern, which separates internal representations of information from the manner in which information is presented to the user, resulting in a platform where users are shown which functions are possible rather than requiring the input of command codes.

Users interact with information by manipulating visual widgets, which are designed to respond in accordance with the type of data they hold and support the actions necessary to complete the user's task.

The appearance, or "skin," of an operating system or application software may be redesigned at will due to the nature of graphical user interfaces being independent from application functions. Applications typically implement their own unique

graphical user interface display elements in addition to graphical user interface elements already present on the existing operating system.

A typical graphical user interface also includes standard formats for representing graphics and text, making it possible to share data between applications running under common graphical user interface design software.

Graphical user interface testing refers to the systematic process of generating test cases in order to evaluate the functionality of the system and its design elements. Graphical user interface testing tools, which are either manual or automated and typically implemented by third-party operators, are available under a variety of licenses and are supported by a variety of platforms. Popular examples include: Tricentis Tosca, Squish GUI Tester, Unified Functional Testing (UFT), Maveryx, Appium, and eggPlant Functional.

5.3. Code Implementation

- **Tensor Flow**

  TensorFlow is a free and open-source software library for machine learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

  Tensorflow is a symbolic math library based on dataflow and differentiable programming. It is used for both research and production at Google.

  TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 in 2015.

- **Conv2D**

  Keras Conv2D is a 2D Convolution Layer, this layer creates a convolution kernel that is wind with layers input which helps produce a tensor of outputs.

- **Max Pooling 2D**

  Max pooling operation for 2D spatial data. Downsamples the input representation by taking the maximum value over the window defined by pool_size for each dimension along the feature's axis. The window is shifted by strides in each dimension.

- **Flatten**

  The role of the Flatten layer in Keras is super simple: A flatten operation on a tensor reshapes the tensor to have the shape that is equal to the number of elements contained in tensor non including the batch dimension.

- **Dense layer**

  Dense layer is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.

- **Dropout**

  Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

- **Classifier**

  Classification is a type of supervised machine learning algorithm used to predict a categorical label. A few useful examples of classification include predicting whether a customer will churn or not, classifying emails into spam or not, or whether a bank loan will default or not.

- **Sequential**

  The Keras Python library makes creating deep learning models fast and easy. The sequential API allows you to create models layer-by-layer for most problems. It is limited in that it does not allow you to create models that share layers or have multiple inputs or outputs.

- **Activation (relu)**

  Activation function, such as tf.nn.relu, or string name of built-in activation function, such as "relu".

- **Optimizer (adam)**

  Adam is a replacement optimization algorithm for stochastic gradient descent for training deep learning models. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

- **Categorical cross entropy**

  All losses are available both via a class handle and via a function handle. The class handles enable you to pass configuration arguments to the constructor (e.g. loss_fn = CategoricalCrossentropy(from_logits=True)), and they perform reduction by default when used in a standalone way.

- **Metrics**

  A metric is a function that is used to judge the performance of your model.Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model.

Checking for GPU

```
In [1]:
from distutils.version import LooseVersion
import warnings
import tensorflow as tf

# Check TensorFlow Version
assert LooseVersion(tf.__version__) >= LooseVersion('1.0'), 'Please use TensorFlow version 1.0 or newer.  You are using {}'.f
print('TensorFlow Version: {}'.format(tf.__version__))

# Check for a GPU
if not tf.test.gpu_device_name():
    warnings.warn('No GPU found. Please ensure you have installed TensorFlow correctly')
else:
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))

TensorFlow Version: 2.1.0
Default GPU Device: /device:GPU:0
```

Fig 5.3.1 Checking for GPU

Creating the CNN model

```
In [2]:
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

Using TensorFlow backend.
```

```
In [3]:
classifier = Sequential()

classifier.add(Conv2D(filters = 32, kernel_size = (3, 3), activation = 'relu', input_shape = (32, 32, 3)))
classifier.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))
classifier.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu'))
classifier.add(Conv2D(filters = 64, kernel_size = (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))
classifier.add(Dropout(.2))

classifier.add(Flatten())

classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dropout(.2))

classifier.add(Dense(units = 46, activation = 'softmax'))

classifier.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

Fig 5.3.2 Creating the CNN model

Fitting the CNN model to the Dataset

```
In [4]:
from keras.preprocessing.image import ImageDataGenerator
from PIL import ImageFile
```

```
In [5]:
ImageFile.LOAD_TRUNCATED_IMAGES = True

train_datagen = ImageDataGenerator(rescale = 1/255, shear_range = .2, rotation_range = 25)
test_datagen = ImageDataGenerator(rescale = 1/255)

training_set = train_datagen.flow_from_directory('Dataset/Train', target_size = (32, 32),
                                                 batch_size = 256, class_mode = 'categorical')
test_set = test_datagen.flow_from_directory('Dataset/Test', target_size = (32, 32),
                                            batch_size = 256, class_mode = 'categorical')

Found 46000 images belonging to 46 classes.
Found 11500 images belonging to 46 classes.
```

```
In [6]: history = classifier.fit(training_set, epochs = 25,
                                validation_data = test_set)

Epoch 1/25
180/180 [==============================] - 36s 200ms/step - loss: 1.5527 - accuracy: 0.5772 - val_loss: 1.3925 - val_accurac
y: 0.6659
Epoch 2/25
180/180 [==============================] - 30s 166ms/step - loss: 0.4499 - accuracy: 0.8661 - val_loss: 0.9108 - val_accurac
y: 0.7186
Epoch 3/25
180/180 [==============================] - 30s 168ms/step - loss: 0.2858 - accuracy: 0.9128 - val_loss: 0.6757 - val_accurac
y: 0.7639
Epoch 4/25
180/180 [==============================] - 30s 168ms/step - loss: 0.2082 - accuracy: 0.9380 - val_loss: 0.8750 - val_accurac
y: 0.7722
Epoch 5/25
180/180 [==============================] - 30s 168ms/step - loss: 0.1657 - accuracy: 0.9487 - val_loss: 0.7533 - val_accurac
y: 0.7990
Epoch 6/25
180/180 [==============================] - 30s 168ms/step - loss: 0.1414 - accuracy: 0.9563 - val_loss: 0.9322 - val_accurac
y: 0.7925
Epoch 7/25
180/180 [==============================] - 31s 171ms/step - loss: 0.1253 - accuracy: 0.9606 - val_loss: 0.8049 - val_accurac
y: 0.8190
Epoch 8/25
180/180 [==============================] - 31s 171ms/step - loss: 0.1102 - accuracy: 0.9657 - val_loss: 0.9132 - val_accurac
y: 0.8103
Epoch 9/25
180/180 [==============================] - 31s 171ms/step - loss: 0.1016 - accuracy: 0.9680 - val_loss: 0.6382 - val_accurac
y: 0.8294
Epoch 10/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0934 - accuracy: 0.9715 - val_loss: 0.6434 - val_accurac
y: 0.8361
Epoch 11/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0862 - accuracy: 0.9723 - val_loss: 1.0215 - val_accurac
y: 0.7840
Epoch 12/25
180/180 [==============================] - 31s 172ms/step - loss: 0.0771 - accuracy: 0.9757 - val_loss: 0.7713 - val_accurac
y: 0.8310
Epoch 13/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0762 - accuracy: 0.9756 - val_loss: 0.9255 - val_accurac
y: 0.8343
Epoch 14/25
180/180 [==============================] - 31s 171ms/step - loss: 0.0683 - accuracy: 0.9779 - val_loss: 0.4756 - val_accurac
y: 0.8313
Epoch 15/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0649 - accuracy: 0.9789 - val_loss: 0.6861 - val_accurac
y: 0.8356

Epoch 16/25
180/180 [==============================] - 31s 175ms/step - loss: 0.0597 - accuracy: 0.9803 - val_loss: 0.7776 - val_accurac
y: 0.8370
Epoch 17/25
180/180 [==============================] - 31s 175ms/step - loss: 0.0579 - accuracy: 0.9817 - val_loss: 0.6734 - val_accurac
y: 0.8365
Epoch 18/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0575 - accuracy: 0.9809 - val_loss: 0.6526 - val_accurac
y: 0.8553
Epoch 19/25
180/180 [==============================] - 31s 171ms/step - loss: 0.0530 - accuracy: 0.9827 - val_loss: 0.5141 - val_accurac
y: 0.8425
Epoch 20/25
180/180 [==============================] - 30s 169ms/step - loss: 0.0522 - accuracy: 0.9834 - val_loss: 0.6587 - val_accurac
y: 0.8306
Epoch 21/25
180/180 [==============================] - 30s 169ms/step - loss: 0.0499 - accuracy: 0.9840 - val_loss: 0.7514 - val_accurac
y: 0.8503
Epoch 22/25
180/180 [==============================] - 30s 169ms/step - loss: 0.0472 - accuracy: 0.9842 - val_loss: 0.5086 - val_accurac
y: 0.8485
Epoch 23/25
180/180 [==============================] - 31s 170ms/step - loss: 0.0493 - accuracy: 0.9831 - val_loss: 0.5540 - val_accurac
y: 0.8456
Epoch 24/25
180/180 [==============================] - 31s 174ms/step - loss: 0.0497 - accuracy: 0.9844 - val_loss: 0.7542 - val_accurac
y: 0.8336
Epoch 25/25
180/180 [==============================] - 31s 174ms/step - loss: 0.0460 - accuracy: 0.9848 - val_loss: 0.4721 - val_accurac
y: 0.8606
```

Fig 5.3.3 Fitting the CNN model to dataset

Summary of Network

```
In [7]: classifier.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        36928
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 10, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               204928
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 46)                5934
=================================================================
Total params: 304,110
Trainable params: 304,110
Non-trainable params: 0
_________________________________________________________________
```

Fig 5.3.4 Summary of Network

Accuracy

```
In [8]: scores = classifier.evaluate(training_set)
        print("Accuracy: %.2f%%" % (scores[1]*100))

180/180 [==============================] - 29s 160ms/step
Accuracy: 99.68%
```

Fig 5.3.5 Accuracy

Saving the model

```
In [9]: classifier_json = classifier.to_json()

        with open("CNN_DevanagariHandWrittenCharacterRecognition.json", "w") as json_file:
            json_file.write(classifier_json)

        classifier.save_weights("CNN_DevanagariHandWrittenCharacterRecognition.h5")
        print('Saved model to disk')

Saved model to disk
```

Fig 5.3.6 Saving the model

Checking the data labels predicted

In [10]:
```
training_set.class_indices
```

Out[10]:
```
{'character_01_ka': 0,
 'character_02_kha': 1,
 'character_03_ga': 2,
 'character_04_gha': 3,
 'character_05_kna': 4,
 'character_06_cha': 5,
 'character_07_chha': 6,
 'character_08_ja': 7,
 'character_09_jha': 8,
 'character_10_yna': 9,
 'character_11_taamatar': 10,
 'character_12_thaa': 11,
 'character_13_daa': 12,
 'character_14_dhaa': 13,
 'character_15_adna': 14,
 'character_16_tabala': 15,
 'character_17_tha': 16,
 'character_18_da': 17,
 'character_19_dha': 18,
 'character_20_na': 19,
 'character_21_pa': 20,
 'character_22_pha': 21,
 'character_23_ba': 22,
 'character_24_bha': 23,
 'character_25_ma': 24,
 'character_26_yaw': 25,
 'character_27_ra': 26,
 'character_28_la': 27,
 'character_29_waw': 28,
 'character_30_motosaw': 29,
 'character_31_petchiryakha': 30,
 'character_32_patalosaw': 31,
 'character_33_ha': 32,
 'character_34_chhya': 33,
 'character_35_tra': 34,
 'character_36_gya': 35,
 'digit_0': 36,
 'digit_1': 37,
 'digit_2': 38,
 'digit_3': 39,
 'digit_4': 40,
 'digit_5': 41,
 'digit_6': 42,
 'digit_7': 43,
 'digit_8': 44,
 'digit_9': 45}
```

Fig 5.3.7 Checking the data labels predicted

Creating a graphical user interface to draw the character

In [11]:
```
import PIL
from PIL import ImageTk, ImageDraw, Image, ImageOps
from tkinter import *
from keras.preprocessing import image
import os
```

Fig 5.3.8 Creating a graphical user interface to draw the character

Function to determine the character according to the max value predicted in the numpy array

In [12]:

```
def determine_character(res,acc):
    print(res)
    if res == 0:
        print('prediction : क')
        character.configure(text= 'क,')
        accuracy.configure(text=str(acc) + '%')
    elif res == 1:
        print('prediction : ख')
        character.configure(text= 'ख,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 2:
        print('prediction : ग')
        character.configure(text= 'ग,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 3:
        print('prediction : घ')
        character.configure(text= 'घ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 4:
        print('prediction : ङ')
        character.configure(text= 'ङ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 5:
        print('prediction : च')
        character.configure(text= 'च,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 6:
        print('prediction : छ')
        character.configure(text= 'छ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 7:
        print('prediction : ज')
        character.configure(text= 'ज,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 8:
        print('prediction : झ')
        character.configure(text= 'झ,')
        accuracy.configure(text=str(acc) + '%')
```

```
    elif res == 9:
        print('prediction : ञ')
        character.configure(text= 'ञ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 10:
        print('prediction : ट')
        character.configure(text= 'ट,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 11:
        print('prediction : ठ')
        character.configure(text= 'ठ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 12:
        print('prediction : ड')
        character.configure(text= 'ड,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 13:
        print('prediction : ढ')
        character.configure(text= 'ढ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 14:
        print('prediction : ण')
        character.configure(text= 'ण,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 15:
        print('prediction : त')
        character.configure(text= 'त,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 16:
        print('prediction : थ')
        character.configure(text= 'थ,')
        accuracy.configure(text=str(acc) + '%')

    elif res == 17:
        print('prediction : द')
        character.configure(text= 'द,')
        accuracy.configure(text=str(acc) + '%')
```

```
elif res == 18:
    print('prediction : ध')
    character.configure(text= 'ध,')
    accuracy.configure(text=str(acc) + '%')

elif res == 19:
    print('prediction : न')
    character.configure(text= 'न,')
    accuracy.configure(text=str(acc) + '%')

elif res == 20:
    print('prediction : प')
    character.configure(text= 'प,')
    accuracy.configure(text=str(acc) + '%')

elif res == 21:
    print('prediction : फ')
    character.configure(text= 'फ,')
    accuracy.configure(text=str(acc) + '%')

elif res == 22:
    print('prediction : ब')
    character.configure(text= 'ब,')
    accuracy.configure(text=str(acc) + '%')

elif res == 23:
    print('prediction : भ')
    character.configure(text= 'भ,')
    accuracy.configure(text=str(acc) + '%')

elif res == 24:
    print('prediction : म')
    character.configure(text= 'म,')
    accuracy.configure(text=str(acc) + '%')

elif res == 25:
    print('prediction : य')
    character.configure(text= 'य,')
    accuracy.configure(text=str(acc) + '%')
```

```
elif res == 26:
    print('prediction : र')
    character.configure(text= 'र,')
    accuracy.configure(text=str(acc) + '%')

elif res == 27:
    print('prediction : ल')
    character.configure(text= 'ल,')
    accuracy.configure(text=str(acc) + '%')

elif res == 28:
    print('prediction : व')
    character.configure(text= 'व,')
    accuracy.configure(text=str(acc) + '%')

elif res == 29:
    print('prediction : श')
    character.configure(text= 'श,')
    accuracy.configure(text=str(acc) + '%')

elif res == 30:
    print('prediction : ष')
    character.configure(text= 'ष,')
    accuracy.configure(text=str(acc) + '%')

elif res == 31:
    print('prediction : स')
    character.configure(text= 'स,')
    accuracy.configure(text=str(acc) + '%')

elif res == 32:
    print('prediction : ह')
    character.configure(text= 'ह,')
    accuracy.configure(text=str(acc) + '%')
```

```
elif res == 33:
    print('prediction : क्ष')
    character.configure(text= 'क्ष,')
    accuracy.configure(text=str(acc) + '%')

elif res == 34:
    print('prediction : त्र')
    character.configure(text= 'त्र,')
    accuracy.configure(text=str(acc) + '%')

elif res == 35:
    print('prediction : ज्ञ')
    character.configure(text= 'ज्ञ,')
    accuracy.configure(text=str(acc) + '%')

elif res == 36:
    print('prediction : ०')
    character.configure(text= '०,')
    accuracy.configure(text=str(acc) + '%')

elif res == 37:
    print('prediction : १')
    character.configure(text= '१,')
    accuracy.configure(text=str(acc) + '%')

elif res == 38:
    print('prediction : २')
    character.configure(text= '२,')
    accuracy.configure(text=str(acc) + '%')

elif res == 39:
    print('prediction : ३')
    character.configure(text= '३,')
    accuracy.configure(text=str(acc) + '%')

elif res == 40:
    print('prediction : ४')
    character.configure(text= '४,')
    accuracy.configure(text=str(acc) + '%')
```

```
elif res == 41:
    print('prediction : ५')
    character.configure(text= '५,')
    accuracy.configure(text=str(acc) + '%')

elif res == 42:
    print('prediction : ६')
    character.configure(text= '६,')
    accuracy.configure(text=str(acc) + '%')

elif res == 43:
    print('prediction : ७')
    character.configure(text= '७,')
    accuracy.configure(text=str(acc) + '%')

elif res == 44:
    print('prediction : ८')
    character.configure(text= '८,')
    accuracy.configure(text=str(acc) + '%')

elif res == 45:
    print('prediction : ९')
    character.configure(text= '९,')
    accuracy.configure(text=str(acc) + '%')
```

Fig 5.3.9 Function to determine the character according to the max value predicted in the numpy array

GUI Application using Tkinter

In [ ]:

```
# Final Application GUI
import numpy as np
import matplotlib.pyplot as plt
from tkinter import *
import tkinter as tk
# loading Python Imaging Library
from PIL import ImageTk, Image
from keras.models import load_model

# To get the dialog box to open when required
from tkinter import filedialog

classifier.load_weights('CNN_DevanagariHandWrittenCharacterRecognition.h5')

def single_prediction(test_img):

    test_img_arr = image.img_to_array(test_img)

    test_img_arr = np.expand_dims(test_img_arr, axis = 0)
#     accuracy = classifier.predict(test_img_arr)[0]
    prediction = classifier.predict(test_img_arr)

    result = np.argmax(prediction[0])
    acc = round(prediction[0][np.argmax(prediction[0])]*100,3)
    print(acc)
    res = classifier.predict([test_img_arr])[0]

    print(res, np.argmax(res), max(res))
    determine_character(result, str(acc))

def model():
    accuracy.configure(text="Predicting...")
    test_img = image.load_img('C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/imag
    single_prediction(test_img)
    plt.imshow(test_img)

def delete_created_image():
    os.remove('C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/image.jpg')
```

```
def clear():
    delete_created_image()
    path = "C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/placeholder.jpg"
    img = Image.open(path)
    img = img.resize((300, 300), Image.ANTIALIAS)
    img = ImageTk.PhotoImage(img)
    panel = Label(root, image = img)

    # set the image as img
    panel.image = img
    panel.grid(row = 1, pady=5, padx=5)
    character.configure(text="")
    accuracy.configure(text="Upload")

def open_img():
    # Select the Imagename  from a folder
    x = openfilename()
    print(x)
    # opens the image
    img = Image.open(x)

    # resize the image and apply a high-quality down sampling filter
    img = img.resize((300, 300), Image.ANTIALIAS)

    #img = ImageOps.invert(img)
    filename = 'C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/image.jpg'
    img.save(filename)
    # PhotoImage class is used to add image to widgets, icons etc
    img = ImageTk.PhotoImage(img)


    panel = Label(root, image = img)

    # set the image as img
    panel.image = img
    panel.grid(row = 1, pady=5, padx=5)

def openfilename():

    # open file dialog box to select image
    filename = filedialog.askopenfilename(title ='Upload Image')
    return filename

# Create a window
root = Tk()
```

```
# Set Title as Image Loader
root.title("Devanagari Character Recognition")
root.resizable(0,0)
# Set the resolution of window
root.geometry("700x450+400+150")
root['bg'] = '#000'

path = "C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/placeholder.jpg"
img = Image.open(path)
img = img.resize((300, 300), Image.ANTIALIAS)
img = ImageTk.PhotoImage(img)
panel = Label(root, image = img)

# set the image as img
panel.image = img
panel.grid(row = 1, pady=5, padx=5)

icon = PhotoImage(file = "C:/Users/peris/Documents/Devanagari Character Recognition - 2020/Dataset/SinglePrediction/icon.png"

# Setting icon of master window
root.iconphoto(False, icon)


character = tk.Label(root, text="", font=("Helvetica", 70), bg="black", fg="white")
character.grid(row=1, column=2,pady=2, padx=2)
accuracy = tk.Label(root, text="     Upload     ", font=("Helvetica", 38), bg="black", fg="white")
accuracy.grid(row=1, column=5,pady=2, padx=2)

btnPredict=tk.Button(text="Predict",command=model, height=2, width=30, relief=GROOVE, bg="white", fg="black")

btnPredict.grid(row=3, column=5, pady=2, padx=5)

btnClear=tk.Button(text="Clear",command=clear, height=2, width=30, relief=GROOVE, bg="white", fg="black")
btnClear.grid(row=3, column=0, pady=2)

btn = Button(root, text ='Open Image', command = open_img, pady=2, padx=2, height=2, width=30, relief=GROOVE, bg="white", fg=

root.mainloop()
```

Fig 5.3.10 GUI Application using TKinter

6. TESTING AND VALIDATION

We need to choose optimal number of epochs to train a neural network in Keras as one of the critical issues while training a neural network on the sample data is Overfitting. When the number of epochs used to train a neural network, model is more than necessary, the training model learns patterns that are specific to sample data to a great extent.

This makes the model incapable to perform well on a new dataset. This model gives high accuracy on the training set (sample data) but fails to achieve good accuracy on the test set. In other words, the model loses generalization capacity by overfitting to the training data.

To mitigate overfitting and to increase the generalization capacity of the neural network, the model should be trained for an optimal number of epochs. A part of training data is dedicated for validation of the model, to check the performance of the model after each epoch of training. Loss and accuracy on the training set as well as on validation set are monitored to look over the epoch number after which the model starts overfitting.

In the training section, we trained our CNN model on the dataset, and it seemed to reach a reasonable loss and accuracy. If the model can take what it has learned and generalize itself to new data, then it would be a true testament to its performance.

Step 1:

We will create our validation set with the help of our training dataset, which we have created in the training section.

Step 2:

Now, similar to why we have declared a training loader in the training section, we will define a validation loader. Validation loader will also create in the same way as we have created training loader, but this time we pass training loader rather than training the dataset, and we set shuffle equals to false because we

will not be trained our validation data. There is no need to shuffle it because it is only for testing purpose.

Step 3:

Our next step is to analyze the validation loss and accuracy at every epoch. For this purpose, we have to create two lists for validation running lost, and validation running loss corrects.

Step 4:

In the next step, we will validate the model. The model will validate the same epoch. After we finished iterating through the entire training set to train our data, we will now iterate through our validation set to test our data.

We will first measure for two things. The first one is the performance of our model, i.e., how many correct classifications. Our model makes on the test set on the validation set to check for overfitting.

Step 5:

We can now loop through our test data.

Step 6:

We are dealing with the convolutional neural network to which the inputs are first being passed. We will focus on the four dimensionalities of these images. So, there is no need to flatten them.

Step 7:

With the help of the outputs, we will calculate the total categorical cross-entropy loss, and the output is ultimately compared with the actual labels.

Step 8:

Now, we will calculate the validation loss and accuracy in the same way as we have calculated the training loss and training accuracy.

Step 9:

Now, we will calculate the validation epoch loss which will be done as same as how we calculate the training epoch loss where we divide the total running loss by the length of the dataset.

Step 10:

We will print validation loss and validation accuracy.

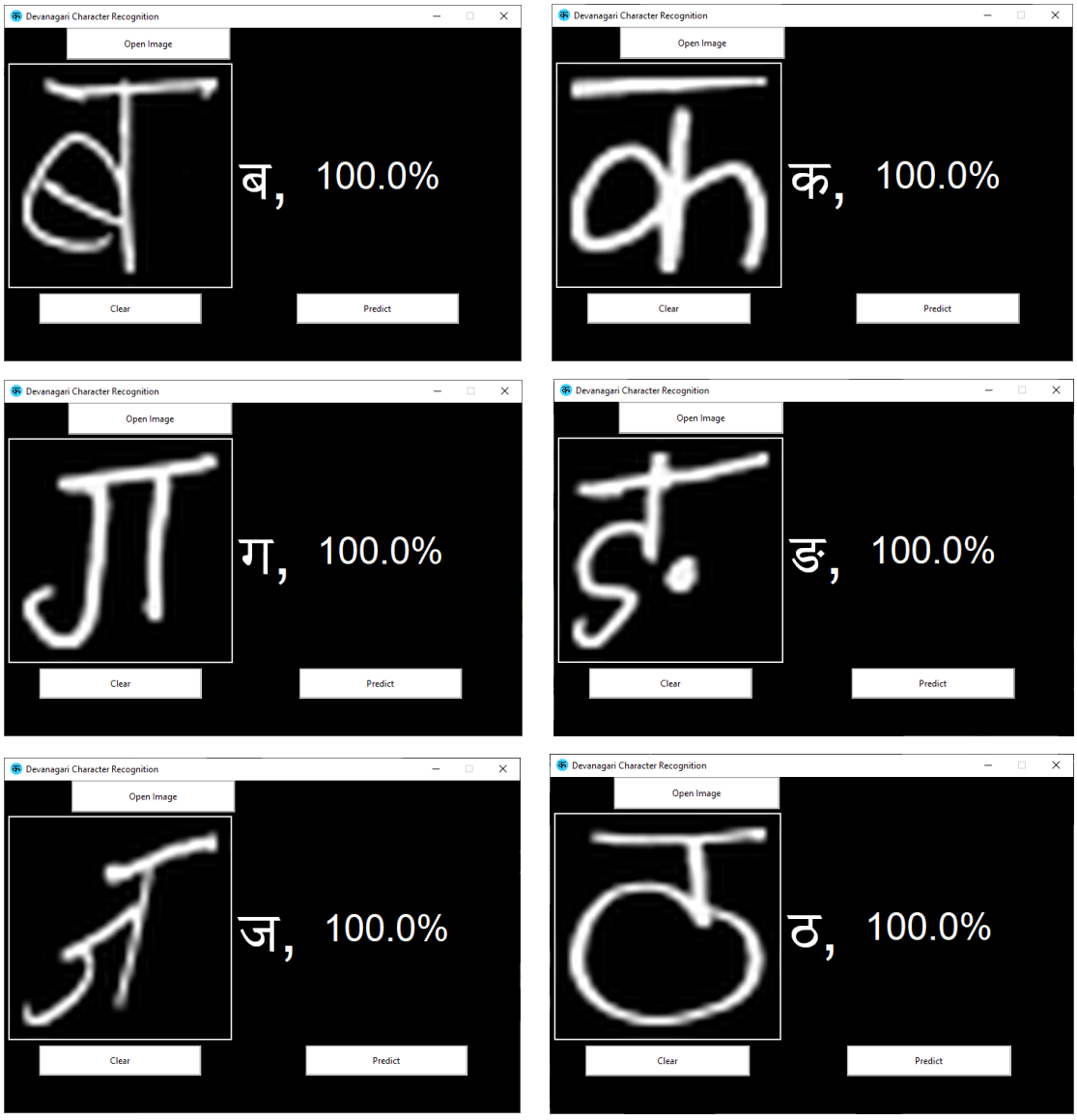

Fig 6.1 Alphabet Outputs

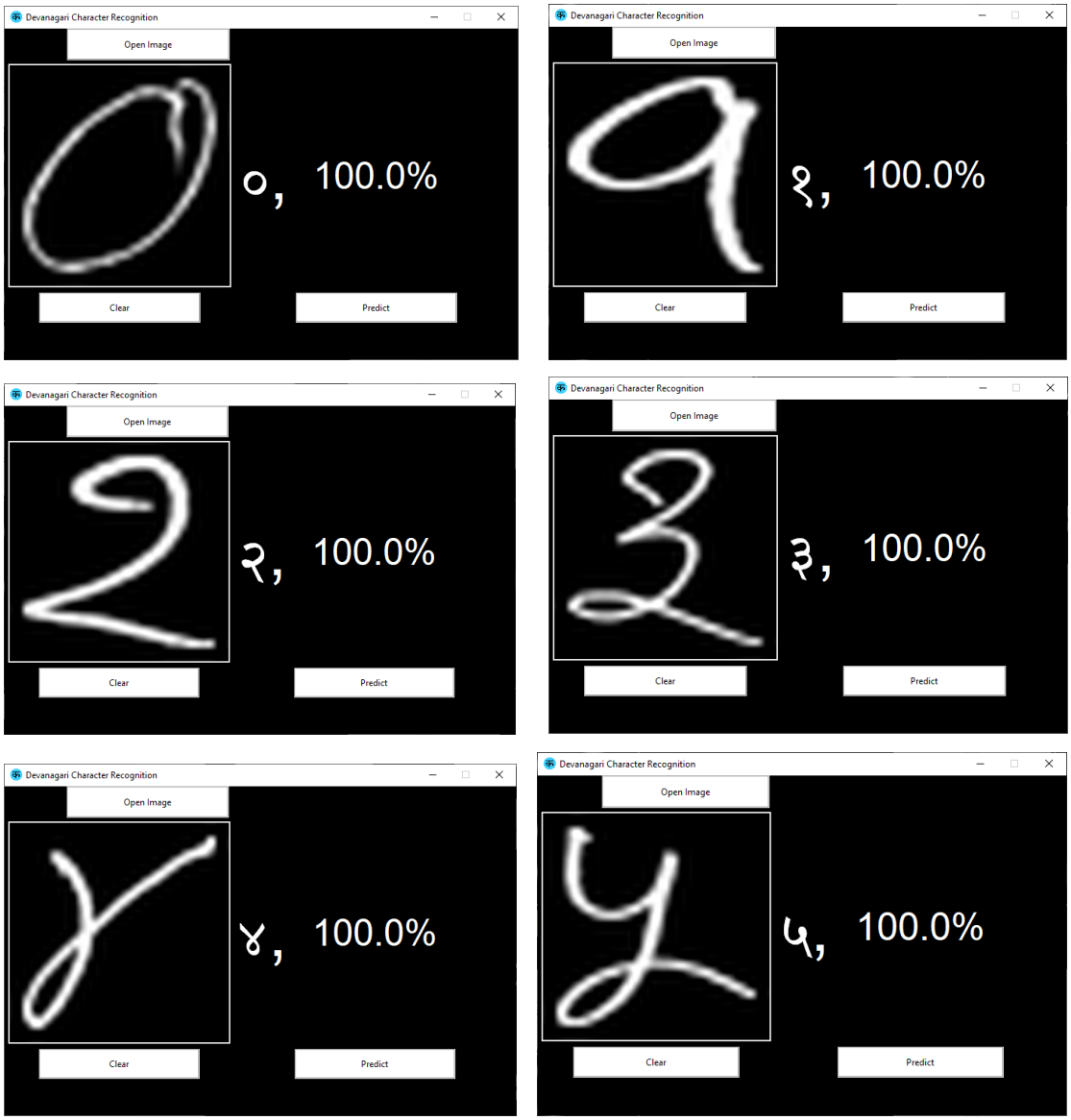

Fig 6.2 Numerical Outputs

7. RESULTS

There are many techniques ANN, CNN that can be used in recognition the handwritten characters. ANN is a group of multiple perceptrons or neurons at each layer. ANN is also known as a Feed-Forward Neural network because inputs are processed only in the forward direction.

This type of neural networks are one of the simplest variants of neural networks. They pass information in one direction, through various input nodes, until it makes it to the output node. The network may or may not have hidden node layers, making their functioning more interpretable. But using of ANN have some disadvantages like Hardware dependence, Unexplained behaviour of the network, Determination of proper network structure.

In our project we are using technique of CNN in recognition of Devanagari handwritten characters as CNN is considered to be more powerful than ANN, RNN. CNN are one of the most popular models used today.

This neural network computational model uses a variation of multilayer perceptrons and contains one or more convolutional layers that can be either entirely connected or pooled. These convolutional layers create feature maps that record a region of image which is ultimately broken into rectangles and sent out for nonlinear processing. The advantage of using CNN in character recognition is:

- Very High accuracy in image recognition problems.
- Automatically detects the important features without any human supervision.
- Weight sharing.

```
In [15]:
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Fig 7.1 Code to plot the graphs

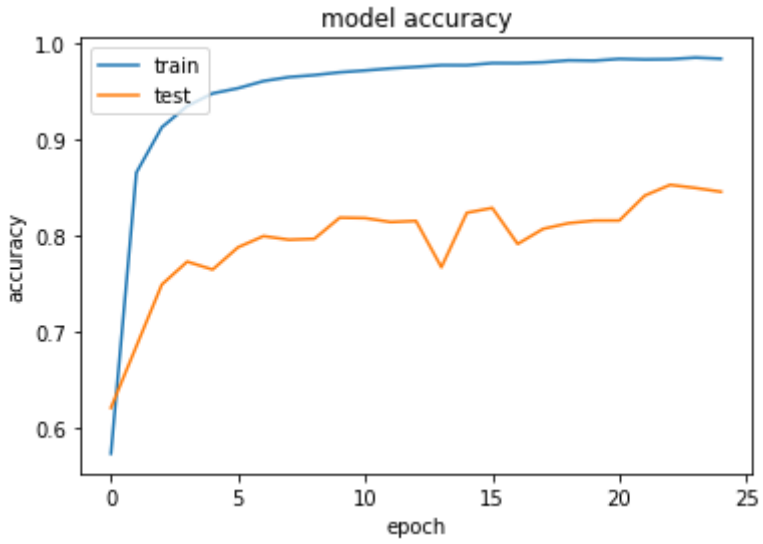

Fig 7.1 Model accuracy graph x-axis – epoch, y-axis - accuracy

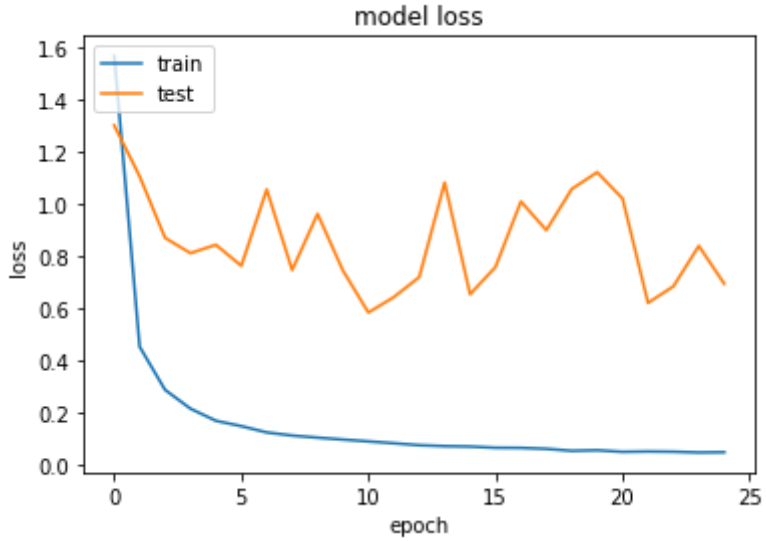

Fig 7.2 Model loss graph x-axis – epoch, y-axis - loss

8. CONCLUSION

The purpose of this project was to introduce convolutional neural networks in recognizing the handwritten Devanagari characters and digits. In general, these networks provide excellent results for classification and recognition tasks. They are also used to interpret sound, text and video data. If the problem to solve is to look for a recognizing the handwritten characters, then convolution networks will be good candidates.

Technology is advancing day-by-day and the traditional methods of handwritten content may get extinct with the same. In the future we might reach a point where we use the touch screens more than the traditional paper and pen. Even then we might face a problem where the ancient or the literature which were made in the past or the present may give a way to learn how we or the ancestral beings used the traditional methods. By this recognition of the text people can easily identify the characters without having many assumptions for a single character. This may help to decrease the time to think and increase the production of the work to be done. In this way this technology may not only help in single language but also various languages with more advanced tweaking in the algorithm.

9. REFERENCES

1. Sufian, A. Ghosh, A. Naskar et al., BDNet: Bengali Handwritten Numeral Digit Recognition based on Densely connected Con-volutional Neural Networks, Journal of King Saud University – Computer and Information Sciences, URL:https://doi.org/10.1016/j.jksuci.2020.03.002

2. Devanagari Character Set
   URL: https://www.kaggle.com/rishianand/devanagari-character-set

3. Python:
   URL: https://www.python.org/

4. Keras:
   URL: https://keras.io/

5. Tensorflow:
   URL: https://www.tensorflow.org/

6. NumPy:
   URL: https://numpy.org/

7. Jupyter:
   URL: https://jupyter.org/

8. Anaconda:
   URL: https://www.anaconda.com/

9. Pillow:
   URL: https://pillow.readthedocs.io/en/stable/

10. TKinter:
    URL: https://docs.python.org/3/library/tkinter.html