NepNet: Simple and Efficient CNN model for Handwritten Devanagari Character Recognition

Paras Mani Gurung
190608163
Ebroul Izquierdo
MSc. Big Data Science
Queen Mary University of London

Abstract—This paper presents NepNet, an effective and simple convolutional neural network architect for recognition of offline handwritten Devanagari characters. The presented model, NepNet was trained on the UCI Devanagari Handwritten Character Dataset (DHCD). NepNet has been able to surpass all other models' test accuracy by achieving the test accuracy of 99.64% on the dataset. During the development of the model, research and experiments were done to reduce the number of trainable parameters while achieving the high-test accuracy at the same time. Therefore, numbers of trainable parameters are also relatively very less in the model which cutbacks the computing cost.

A simple web application is developed using Gradio library which enables users to write/draw Devanagari characters and displays its prediction using the proposed NepNet model. A simple OCR (Optical Character Recognition) web system has also been developed using the previous works, that segments characters from word in an image and converts them into texts. This application implements the NepNet model for the classification of the characters.

Keywords—Computer Vision; Devanagari Handwritten Character Dataset; Deep Convolutional Neural Network, Image Processing, Optical Character Recognition

I. INTRODUCTION

Optical character recognition (OCR) is a technique to process and convert the text images into machine-encoded text i.e. word file or text file, so that they can be easily manipulated and stored. (Regev, 2019) In around First World War, Emanuel Goldberg, a physicist invented a device that was able to read characters and convert them into telegraph code[15]. This was one of the very first steps on the OCR field. Later, Samuel created the first electronic document retrieval system in the 1920s. Since then, a lot of development has happened in the field. The use of deep learning algorithms has made OCR system more effective and practical.

A lot of research and works has been executed in the field of English characters recognition. However, not much of work has been done in the field of Devanagari script. Devanagari is one of the most adopted writing systems in the

Fig. 1. Devanagari Consonant Characters

world which was developed in ancient India from the 1st to the 4th century, based on ancient Brahmi script. Nepali, Hindi, etc. are the language that uses the Devanagari script. It consists of 45 primary characters, 12 vowels and 33 consonants.

Fig. 2. Devanagari Vowel Characters

Fig. 3. Devanagari Numbers

Fig. 4. Combination of consonant Pa and Vowels

A. Motivation

Due to the enormous development in the processing and memory ability of the computers such as large-scale distributed system, GPUs, etc., the deeper neural networks are very trendy since the last decade for computer vision solutions. The digitalisation of system results in easy availability of big data resources for the training of the deep networks also makes it more powerful. The CNN (Convolutional Neural Network) has brought the revolution in the computer vision field with its effectiveness. Therefore, leaving all other machine learning classification techniques CNN has been chosen for the classification of the images of the handwritten characters. However, the deep neural network architectures have the problem of overfitting as the network goes deeper.

ResNet[8], DenseNet[9], GoogLeNet/Inception[20], etc. are the examples of very deep CNN architecture which has overcome the overfitting issues and are very effective and reliable in computer vision. So, what is the purpose of proposing the new model for recognition of Devanagari handwritten characters when there are already efficient and successful models? Although these acknowledged deep CNN models are able to achieve high accuracy by going deeper and eliminating the overfitting issue, they include tens to hundreds of millions of trainable parameters which results in enormous computing cost and memory usage. Considering the computing cost, training time and memory usage, it is not very appropriate to use those very deep networks for the
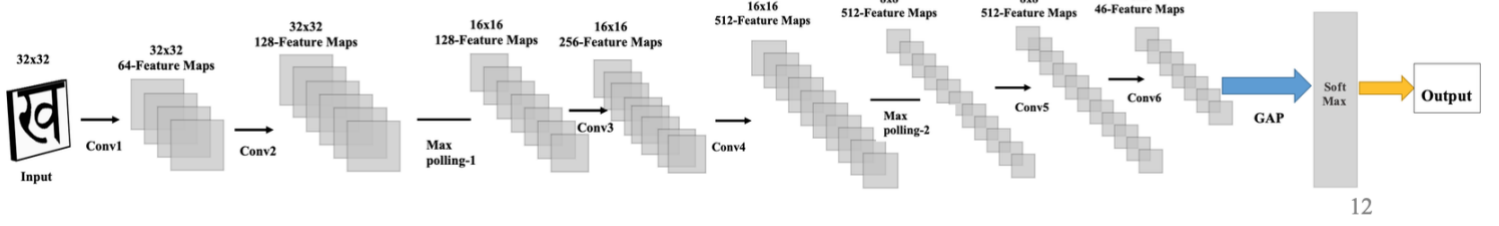
Fig. 5. Architecture of NepNet

recognition of Devanagari handwritten characters. Therefore, a new model, NepNet, customised CNN model for recognition of the Devanagari handwritten characters has been proposed. NepNet uses a smaller number of parameters which reduces the computing cost, memory usage and storage space.

II. RELATED WORK

The related work has been categorised into two section. First on Devanagari handwritten character recognition and second on building efficient and light weight convolution neural network architecture.

A. Devanagri Handwritten Character Recognition

Shailesh and his team (Acharya, et al., 2015) proposed two neural network models, the first model with three convolutional layers and one fully connected layer and the second one with two convolutional layers and two fully connected layers[2]. 5x5 filter size was used in the both models. Dropout and dataset increment are the techniques that are used to increase the accuracy of the models. The first model uses Rectified Linear Unit Layers (RELU), Local Response Normalisation and max pooling after each convolutional layer and is able to achieve the test accuracy of 98.47%. The second model consists two convolutional layers and two fully connected layers which was derived from LeNet-5[13] architecture. It is able to obtain the test accuracy of 98.26%. Riya and her team (Guha, et al., 2019) were able to obtain the very high test accuracy of 99.54% on the same dataset by resizing the image into 64x64 pixel size and applying CNN model[5]. The model consists of 5 convolutional layers and doesn't adopt any fully connected layers. Batch Normalisation, RELU and max pooling have been used between one convolution layer to next layer. It uses 5x5 filter size and the numbers of filters are doubled from one layer to another layer. Mohite's team (Mohite & Shelke, 2018)utilizes AlexNet[12] for the Devanagari handwritten character recognition and were able to obtain the accuracy of 91.23% on the Devanagari characters and 100% on the Devanagari numbers[14]. (Swethalakshmi, et al., 2006) (Gaue & Yadav, 2015) K-Means clustering and SVM methods were also used by researchers for Devanagari handwritten character recognition[4][19]. However, such techniques have been outclassed by the CNN.

B. Building efficient and light weight CNN architecture

The first recognized work on the simplicity and the effectiveness of a network was published on 2014 by Springenberg's team (Springenberg, et al., 2014)[18]. Their architecture with very lesser number of parameters was able to surpass the state-of-the-art test accuracy on the CIFAR-10 dataset and obtained comparable results on the CIFAR-100 and ImageNet dataset. They introduced the idea of using convolution layers with the stride 2 instead of max pooling and demonstrated that it improved the accuracy of their model. The architecture also backed up the idea of using global average pooling instead of fully connected layer before the classification, introduced by Lin's team, in their paper, "Network In Network"[3]. Both of the model demonstrated that the use of fully connected layers consistently performs 0.5%-1% worse. In 2016 Iandola's team (Iandola, et al., 2016) introduced SqueezeNet which was able to achieve the AlexNet-level accuracy on ImageNet with 50 times fewer number of parameters[10]. SqueezeNet uses fire module which consists of squeeze layer of 1x1 filters and expand layer of 1x1 and 3x3 filters inspiring from inception module of GoogLeNet.

In 2018 S.H. Hasanpour and his team (Hasanpour, et al., 2018) introduced SimpleNet, which achieved state of the art result on CIFAR10 with 2 to 25 times fewer number of parameters and operations than the other deep neural network models[6][7]. The model consists of 13 convolutional layers of 1x1 sized filters on 11th and, 12th layer and 3x3 sized filters on all the other layers.

III. NEPNET: OVERVIEW

The proposed model, NepNet is a convolutional neural network based architecture customized for offline handwritten Devanagari character recognition. The main objective of the NepNet is to absorb distinguish feature of the Devanagari characters in order to achieve the higher accuracy rate on their classification/recognition using lesser number of parameters when compared to the existing models. NepNet facilities lower computing cost and memory usage during training.

A. Convolution Neural Network

CNN (Convolution Neural Network) has achieved huge success in the field of large-scale image and video recognition. The ILSVRC (ImageNet Large Scale Visual Recognition Competition) has played significant role in the development of the deep visual recognition architectures. LeNet-5 introduced by LeCun et al. in 1998 is a pioneer CNN[13]. Inspired from LeNet-5, Krizhevsky et al. (2012),

proposed AlexNet, which won ILSVRC in 2012. The model also started utilization of the dropout technique to prevent overfitting issue. Various CNN architectures have been emerged thoroughly with the trend of increasing the depth and breadth of the network by increasing the number of layers and number of filters. The use of dropout in the model has also been trendy and useful for preventing overfitting.

In 2014, two impressive model VGG[17] and GoogLeNet[20] arose. GoogLeNet was the winner of ILSVRC 2014 and the VGG was the first runner-up. However, VGG showed significant improvement over all the previous winners of ILSVRC and also beat the GoogLeNet on the localization task. 2014 was the first year, deep learning networks achieved the error rate below 10 percentage. As the networks were going deeper and deeper, it was more difficult to train them without overfitting. In 2015, the Residual Learning framework (He, et al., 2016) also known as ResNet was introduced to ease the training of the deep networks[8]. The model was the winner at classification task in ILSVRC 2015. Since then, various versions of those models have been introduced.

B. Basic CNN Architecture

A convolutional neural network has two sections i.e. feature extraction and classification. In feature extraction different convolutional elements are used to extract the relevant features of the input images. A simple CNN consists of convolution layers with activation function, pooling layers, fully connected layers and a classifier at the end. In a convolution layer, filters are applied to extract the relevant features from the input image and is passed to next layer in the network. Pooling layer are also known as subsampling layer where the image size from the previous layer is reduced usually in half. Stacking those layers in a systematic order, a decent convolutional neural network can be designed.
Calculating the number of parameters in a convolution layer of a CNN,

$$\rho = \omega_+ \beta \qquad (1)$$

$$\omega = \mathcal{F}_h \times \mathcal{F}_l \times \mathcal{C} \times \mathcal{N} \qquad (2)$$

$$\beta = \mathcal{N} \qquad (3)$$

$$\rho = \mathcal{F}_h \times \mathcal{F}_l \times \mathcal{C} \times \mathcal{N} + \mathcal{N} \qquad (4)$$

$$\rho = \mathcal{N} (\mathcal{F}_h \times \mathcal{F}_l \times \mathcal{C} + 1) \qquad (5)$$

where ρ is the number of parameters, ω is the number of weights, β is the number of bias, $\mathcal{F}_h$ is the height of the filter, $\mathcal{F}_l$ is the length of the filter, $\mathcal{C}$ is the number of channels in the input and $\mathcal{N}$ is the number of filters in the convolutional layer.

Calculation of the convolution output height or length of feature map, **Conv_out** from each convolutional layer of a CNN,

$$Conv_{out} = \frac{In_h + 2 \times p - F_h}{s} + 1 \qquad (6)$$

where In_h is the height or length of the input, p is the padding, F_h is the height or length of the filter and s is stride size.

Calculation of the pooling output height or weight, Pool_out from each pooling layer of a CNN,

$$Pool_{out} = \frac{In_h - F_h}{s} + 1 \qquad (7)$$

where In_h is the height or length of the input, F_h is the height or length of the filter and s is stride size.

C. NepNet Architecture

NepNet is a simple convolutional neural network that uses fundamental CNN components. It consists of 6 CLs (Convolutional Layers), 2 max-pooling layers and a GAP (Global Average Pooling). The first 4 CLs uses 3x3 filters and the remaining 2 CLs uses 1x1 filters. In each convolution layers, BN (batched normalisation) is being used as regularisation method in order to reduce internal covariate shift and accelerate the model's training. Non- linear activation function, ReLu has been applied after the batch normalisation. In aggregate two max-pooling are applied, first-one after the second convolutional layer and the second after the fourth convolutional layer. The architecture of the NepNet is shown in the "Fig. 5". Stride of 1 and same padding has been used in all the convolutional layers in order to prevent the spatial resolution of the input image when convolution is applied. The number of filters is doubled from one layer to another until the fifth CL. In the first CL 64 filters have been used, 128 in the second, 256 in the third, 516 in the fourth, 516 in the fifth and 46 in the sixth or the last layer. The max-polling uses the 2x2 filters with the stride of 2 which reduces the feature map's size into half by taking the maximum values from each 2x2 area of the feature maps.

Table I. NepNet Architecture

Layers	Architecture in detail			
	Filters	No. of filters	Padding	Strides
Conv-1	3x3	64	Same	1
Conv-2	3x3	128	Same	1
Max-Pooling	2x2			2
Conv-3	3x3	256	Same	1
Conv-4	3x3	512	Same	1
Max-Pooling	2x2			2
Conv-5	1x1	512	Same	1
Conv-6	1x1	512	Same	1
Global Average Pooling				
Classification (SoftMax)				

Table II. Summary of Convolutional and Pooling output of NepNet

	Summary of convolutional and pooling output in each layer		
Layers	**Input**	**Calculation**	**Output**
Conv-1 3x3,64	32x32x1	$$Conv_{out} = \frac{In_h + 2 \times p - F_h}{s} + 1$$ In$_h$ =32, F$_h$ =3, p=1, s=1 ((32+2-3)/1)+1=32 Num. of channels=Num. of filters=64	32x32x64
Conv-2 3x3,128	32x32x64	In$_h$ =32, F$_h$ =3, p=1, s=1 ((32+2-3)/1)+1=32 Num. of channels =Num. of filters=128	32x32x128
Max-Pooling 2x2, s=2	32x32x128	$$Pool_{out} = \frac{In_h - F_h}{s} + 1$$ In$_h$ =32, F$_h$ =2, s=2 ((32-2)/2)+1=16 Num. of channels = Num. of input channel =128	16x16x128
Conv-3 3x3,256	16x16x128	In$_h$ =16, F$_h$ =3, p=1, s=1 ((16+2-3)/1)+1=16 Num. of channels =Num. of filters=256	16x16x256
Conv-4 3x3,512	16x16x256	In$_h$ =16, F$_h$ =3, p=1, s=1 ((16+2-3)/1)+1=16 Num. of channels =Num. of filters=512	16x16x512
Max-Pooling 2x2, s=2	16x16x512	In$_h$ =16, F$_h$ =2, s=2 ((16-2)/2)+1=8 Num. of channels = Num. of input channel =512	8x8x512
Conv-5 1x1, 512	8x8x512	In$_h$ =8, F$_h$ =3, s=1 ((8-1)/1)+1=8 Num. of channels =Num. of filters=512	8x8x512
Conv-6 1x1, 46	8x8x512	In$_h$ =8, F$_h$ =3, s=1 ((8-1)/1)+1=8 Num. of channels =Num. of filters=46	8x8x46

The "Table. II" simplifies how the images are being processed from one convolution layer to another and shows the sub-sampling of the image using the max-pooling layers to reduce the size of the processed image. The demonstration of use of formulae/equations- (6 and 7) also delivers the deep insight of a CNN architecture. In the model, the same padding has been used in the convolution layers which keeps the same height and length of the output from the input by padding the input with a certain value.

For example:

In convolutional layer-1,

Input height/length (In$_h$)= 32x32
Aim to make output height/length (Conv$_{out}$)= 32x32
Filters height/ length (F$_h$) = 3
Strides (s)= 1
Padding being used (p)= ?

Now using the formula 6,

$$Conv_{out} = \frac{In_h + 2 \times p - F_h}{s} + 1$$

32= ((32 + 2 x p − 3)/1) + 1
32 = 30 + 2p
p = (32 -30)/ 2
p = 1

Therefore, padding of 1 is being used on the layer.

The summary of the parameters of the NepNet can be seen in the "Fig. 6". The model has the total parameters of 1,844,312 where 3,036 are non-trainable and 1,841,276 are trainable parameters. The number of parameters on each layer of the model are clearly indicated in the figure. Using the formula-5, we can easily calculate the number of parameters of a convolutional layer.

For examples:
In convolution layer-1:

height of the filter ($\mathcal{F}_h$) = 3
length of the filter ($\mathcal{F}_l$) = 3
number of filters ($\mathcal{N}$) = 64
number of channels in the input ($\mathcal{C}$) = 1

Now using the formula-5,
    parameters, $\rho = \mathcal{N} (\mathcal{F}_h \times \mathcal{F}_l \times \mathcal{C} + 1)$
        p= 64 (3 x 3 x 1 + 1)
        p= 64 x 10
        p= 640

Therefore, there are 640 parameters in the convolution layer-1. All the parameters of the convolutional operation are trainable.

D. Derivation of the NepNet

NepNet architecture does resembles few techniques used by the VGG[17] CNN model such as the use of 3x3 filters, max-pooling layer after two convolutional layer and doubling of the number of filters starting from 64. Inspiring from the

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 64)	640
batch_normalization (BatchNo	(None, 32, 32, 64)	256
activation (Activation)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 32, 32, 128)	73856
batch_normalization_1 (Batch	(None, 32, 32, 128)	512
activation_1 (Activation)	(None, 32, 32, 128)	0
max_pooling2d (MaxPooling2D)	(None, 16, 16, 128)	0
conv2d_2 (Conv2D)	(None, 16, 16, 256)	295168
batch_normalization_2 (Batch	(None, 16, 16, 256)	1024
activation_2 (Activation)	(None, 16, 16, 256)	0
conv2d_3 (Conv2D)	(None, 16, 16, 512)	1180160
batch_normalization_3 (Batch	(None, 16, 16, 512)	2048
activation_3 (Activation)	(None, 16, 16, 512)	0
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 512)	0
conv2d_4 (Conv2D)	(None, 8, 8, 512)	262656
batch_normalization_4 (Batch	(None, 8, 8, 512)	2048
activation_4 (Activation)	(None, 8, 8, 512)	0
conv2d_5 (Conv2D)	(None, 8, 8, 46)	23598
batch_normalization_5 (Batch	(None, 8, 8, 46)	184
activation_5 (Activation)	(None, 8, 8, 46)	0
global_average_pooling2d (Gl	(None, 46)	0
dense (Dense)	(None, 46)	2162

Total params: 1,844,312
Trainable params: 1,841,276
Non-trainable params: 3,036

Fig. 6. Summary of parameters of the NepNet

Network in Network architecture[3], we have used 1x1 filters towards the end of the network . Along with achieving higher test accuracy rate, reducing the trainable parameters and computing costs also were the focus of the architect, therefore use of the 3x3 and 1x1 filters were very essential. We also avoided using fully connected layer and used global average pooling (GAP) introduced in Network in Network model. The GAP takes the average of each feature map of all convolution layers and the outcome vector is directly fed into the softmax layer and use no parameters to optimise. Use of GAP instead of fully connected layers reduce the number of parameters of the model and also increases the test accuracy of the NepNet.

Different alternative models such as model using 5x5 filters, adding fully connected layers, etc. were experimented before the derivation of the NepNet. Conduction of those experiments has driven to the formation of the NepNet architecture. The detailed information on the accuracy of those different alternative models is demonstrated on the "Table IV".

E. Regularisation

As the model, NepNet doesn't consist any fully connected layers but has only convolutional layers, Batch Normalization(BN) has been used for regularisation. (Ioffeand & Szegedy, 2015)S. Ioffeand's team demonstrates that the BN regularises a CNN model effectively. Following the instruction from the paper[11], dropout has been removed from the model and through shuffling of training dataset have been performed in order to accelerate the learning rate and

improve the test accuracy of the model. Each training examples are observed fewer times in BN model which helps to reduce the photometric distortions.

IV. RESULTS AND ANALYSIS

NepNet has been trained on the dataset by using python-based library, TensorFlow. The experiments were being performed on Jupyter Notebook and Google Collab (GPU) .

A. Dataset

The proposed CNN architecture, NepNet has been tested on the DHCD introduced by A.K. Pant et al[2]. The dataset contains 92 thousand images of 46 different classes of Devanagari script; 36 classes of consonant characters and 10 classes of numerals. The vowels of the Devanagari script have not been included in the dataset. Handwritten documents from wide range of individuals were scanned and manually cropped in order to produce such a huge dataset. The dataset contains 2,000 images of each classes, 85 percentage of the dataset is split into training and 15 percentage for testing. During the training, 0.05% of the training data i.e. 3910 images is used for validation. The images are in .png format of resolution 32x32. Padding of 2 pixels is added on all the four sides of the actual character which is centred with 28x28 pixel. This dataset is freely and publicly accessible and also has the huge volume of the images. Therefore, this dataset was being used for the experiment.

Table III. Details of the datasets

UCI Devanagari Handwritten Character Dataset		
Training size	Validation size	Testing size
74290	3910	13800

B. Dataset Pre-processing

In both training and testing folders there was one folder for each character. With the use of the Python OpenCV library all the testing and training images are saved in the matrix format along with its class in arrays. Both the training and test data are shuffled and saved by using Python pickle module. Pickle module serializes python object to enable saving them on disk. It also de-serializes them when they are being loaded. Pickling our dataset converts them into a character stream. While training the model, they are unpickled and loaded.

C. Results

The proposed NepNet architecture is able to obtain the test accuracy **of 99.64%** on the UCI handwritten Devanagari character dataset. To my knowledge, it the highest accuracy obtained on the dataset. When batch normalisation is applied after the activation function, ReLu, the test accuracy of the model was reduced to 99.46%. This shows that batch normalisation applied before the activation function performs better.

Table IV. Comparison of different alternative models

Classification Accuracy and Error rate			
NepNet with	Accuracy (%)	Error (%)	Learnable parameters
BN after ReLu	99.46	0.54	1,841,276
Fully Connected layers instead of GAP	98.69	1.30	1,974,584
Using 5x5 instead of 3x3	99.53	0.46	4,594,812
NepNet	**99.64**	**0.35**	**1,841,276**

These experiments were performed in order to raise the best model.

The below charts are the results of training the model using Adam optimiser with 40 epochs. Using the ModelCheckpoint function of the Keras, the model with the lowest validation loss during the training was saved.
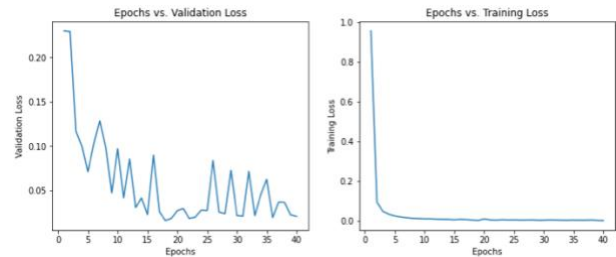

Fig. 7. Visualisation of validation and training loss with number of epochs
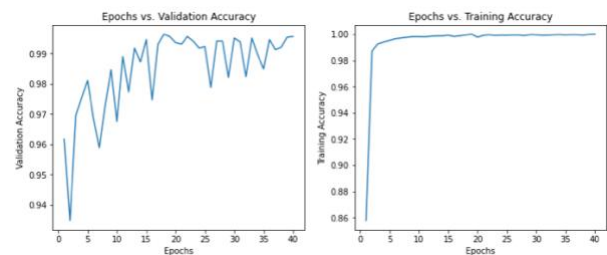

Fig. 8. Visualisation of validation and training loss with number of epochs

The proposed architecture NepNet was also compared to the other models that are already introduced by the researchers. From the comparison, we are able to conclude that the NepNet architecture beats all the other introduced models in the test accuracy with the low number of trainable parameters. The details of the comparison of the different models is shown in the "Table V".

Table V. Comparison of NepNet with different models

Classification Accuracy and Error rate				
Models	*Accuracy (%)*	*parameters*	*Layers*	*Source*
LeNet-5	95.42	~0.7M	7	[5]
ResNet-18	99.47	~11M	18	[5]
ResNet-34	99.42	~21M	34	[5]
AlexNet	97.84	~61M	8	[5]
DesnseNet-121	99.60	~7.9M	121	[5]
Inception-v3	99.56	~26M	42	[5]
A.K Pant et al.	99.47	~2M	3	[2]
DevNet (Riya Guha et al.)	99.54	~4M	5	[5]
Proposed Model (NepNet)	**99.62**	1,841,276	6	

V. CHARACTER RECOGNITION APPLICATION
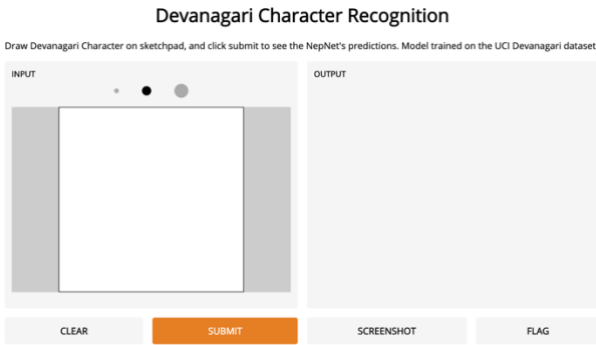
Fig. 9. Interface of the web application system

NepNet has been implemented for the recognition of character drawn on the sketchpad in a web application system. In the application's interface, on the left hand side there is a sketchpad where users can draw/write a Devanagari character using mouse cursor. When the submit button is pressed, it will take the sketchpad as an image and makes predictions by using the NepNet model. Then, the characters with top 3 prediction score is displayed on the right hand side, on the output section. The clear button is to clear the sketch and the Screenshot button allows to take the screenshot of the

application's window. Flag button is to report when the model makes the wrong prediction.

The developed Devanagari character recognition application system facilitates the visualisation of the predictions of the proposed model, NepNet. The application system allows us to observe the efficiently of the NepNet, as it takes the drawn/written character by users for the recognition. Using the system, the similar Devanagari characters were drawn and tested. The characters that are similar to each other are listed in the table below. The system was able to correctly classify all those similar characters that were drawn by me on the sketchpad.

Table VI. Similar Characters

Characters	*Images*	*Images*
Ka and Pha		
Kha and Sha (Patalo)		
Gha and Dha		
Tha and Yaw		
Ma and Bha		
Ba and woh		
Sha(Petchirya) and Pa		

For the development of the application, Gradio library has been utilised. Gradio library provides simple interface and can wrap python function with it, for the development of python based web application system. The library enables users to easily build web application system implementing their trained models.
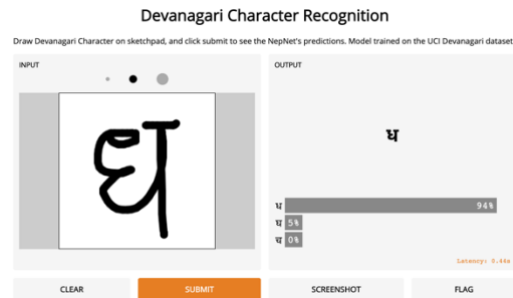
Fig. 10. A Prediction by the Model

VI. WORD RECOGNISTION APPLICATION

A simple OCR web application system has been developed which can segments characters of a Devanagari word and the characters are recognized by using the trained NepNet model. The code for the algorithm is used from GitHub by Ram Krishna Acharya[1].
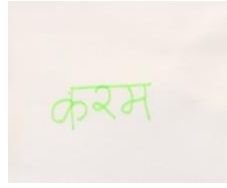

Fig. 11. Input Image Sample

A. Pre-processing of image

It is very vital to pre-process the input image in order to remove noises before processing it. Firstly, the image is converted into Gray image. For the removal of noise, Gaussian Blurring is being used. Gaussian Blurring is very efficient for the removal of general noise from an image. During the training of the model, images have black pixeled background and white pixel on characters. In almost all the cases, input images are always likely to have white pixeled background and back pixel on characters. Therefore, the binary pixels of the images are inverted during this stage.
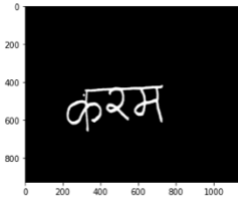
Fig. 12. Image after pre-processing

B. Border Detection

As the whole image might not be occupied with the written characters, the blank space is removed from the image. In other words, boarders of the written characters in the image is detected. For recognising the background of the image, the first 5 pixels from the top-left corner is checked and the detected colour is taken as the background. Likewise, the alternating colour is taken as the characters colour.

Fig. 13. Boarder detection

C. Segmentation

Using detected colour of the characters and its comparison on each pixel, and determining the space between them, each characters of the word are segmented. In the beginning of the segmentation, the top-line on characters called 'dika' is removed and the gap is identified between the characters. These techniques are learned from the paper by Sahai & Sahu (2016)[16]. As the NepNet model takes 32x32 input, the segmented images are resized into 32x32 size and are proceed to the

Fig. 14. Removal of 'Dika'

prediction/recognition stage. An example of segmentation of characters from a word is shown in the "Fig. 15".
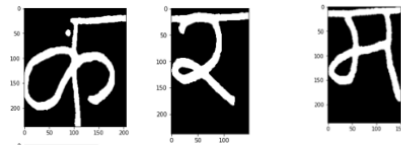
Fig. 15. Characters Segmentation

D. Recognition/Prediction

In this stage, NepNet predicts each character of the segmented images. The character with the highest prediction score is displayed. When the highest prediction score is less than 50%, the character is not displayed as the confidence of the prediction is low. The prediction confidence is the average of the max prediction probabilities of the characters.
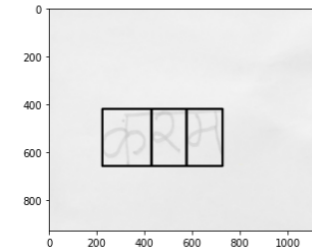
Fig. 16. Final Outcome

After the algorithms have displayed good work, it has been deployed to web application system using Python Flask.
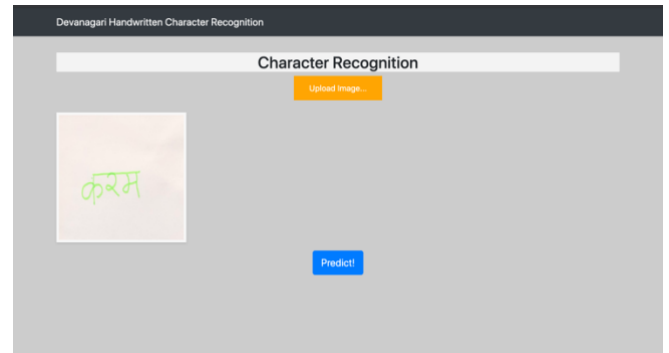

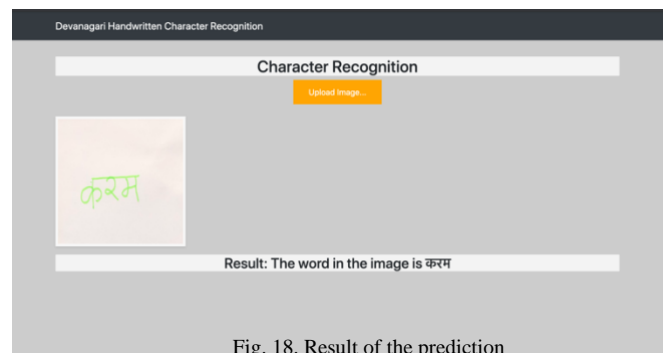Fig. 17. Interface of the web application system

Fig. 18. Result of the prediction

VII. Conclusion

Devanagari handwriting character recognition is a very challenging field due to the complexity of the language. There are still many researches going on the Devanagari handwriting character recognition. The proposed CNN based architecture, NepNet has achieved highest test accuracy of 99.64% on the UCI Devanagari Handwritten Character Dataset surpassing all the existing models. Research and efforts were made on to build efficient and light weight CNN architecture. NepNet also has few numbers of layers and low number of parameters which reduces the computing cost and memory usage of the system.

In order to check and visualize the efficiency of the NepNet model, a simple web application system is developed using Gradio library which allows users to write/draw Devanagari characters and displays its prediction. The Devanagari characters that are similar to other characters were drawn and tested in the system, and the system obtained brilliant prediction accuracy. Following the previous works, a simple OCR system that can recognise word in an image, is also been developed.

VIII. Future Works

The proposed convolutional neural network architecture, NepNet has able to achieve the highest accuracy on the UCI Devanagari Handwritten Character Dataset with the lower number of trainable parameters. The test accuracy achieved by the NepNet can be used as the benchmark on the dataset for the evaluation of a convolutional neural network architectures designed for the Devanagari handwritten character recognition.

The UCI dataset is limited to 10 numeral characters and base form of 36 consonant characters. It is very essential to further add all the vowel to the dataset. All the consonant characters form sub characters when they are combined with the vowels, an example is shown in the Fig. 4. Devanagari writing also contains half characters and special characters attached to the base characters which make the implementation of OCR on handwritten characters very challenging. Further works can be done on simplifying the challenge and develop efficient algorithms on segmentation of attached special characters and half characters.

Reference

[1]. Acharya, R. K., 2019. *Github.* [Online] Available at: https://github.com/q-viper/final-devanagari-word-char-detector [Accessed July 2020].

[2]. Acharya, S., Pant, A. K. & Gyawali, P. K., 2015. Deep learning based large scale handwritten Devanagari character recognition. *International conference on software, knowledge, information management and applications (SKIMA).*

[3]. Chen, Q., Lin, M. & Yan, S., 2014. Network in network. *In ICLR: Conference Track.*

[4]. Gaue, A. & Yadav, S., 2015. Handwritten Hindi character recognition using K-means clustering and SVM. *International symposium on emerging trends and technologies in libraries and information services,* Volume 4th.

[5]. Guha, R., Das, N., Kundu, M. & Nasipuri, M., 2019. DevNet: an efficient CNN architecture for handwritten Devanagari character recognition. *International journal of pattern recognition and artificial intelligence,* 10 July.

[6]. Hasanpour, S. H., Rouhani, M., Fayyaz, M. & Sabokrou, M., 2018. Let's keep it simple, using simple architectures to outperform deeper and more complex architectures. February .

[7]. Hasanpour, S. H. et al., 2018. Towards principle design of deep convolutional networks: Introducing SimpleNet. February.

[8]. He, K., Zhang, X., Ren, S. & Sun, J., 2016. Deep residual learning for image recognition. *Conference on Computer Vision and Pattern Recognition (CVPR).*

[9]. Huang, G., Liu, Z. & Maaten, L. v. d., 2018. Densely Connected Convolutional Networks.

[10]. Iandola, F. N. et al., 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size.

[11]. Ioffeand, S. & Szegedy, C., 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 2 March.

[12]. Krizhevsky, A., Sutskever, I. & Hinton, G. E., 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Conference on Neural Information Processing Systems.*

[13]. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P., 1998. Gradient-based learning applied to document recognition. November.

[14]. Mohite, A. & Shelke, S., 2018. Handwritten Devanagari character recognition using convolutional neural network. *International conference for convergence in technology,* Volume 4th.

[15]. Regev, R., 2019. *ScanMarker.* [Online] Available at: https://scanmarker.com/2019/04/02/a-brief-history-of-ocr-the-technology-inside-your-scanmarker/

[16]. Sahai, M. & Sahu, N., 2016. Segmentation of Devanagari Character.

[17]. Simonyan, K. & Zisserman, A., 2015. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations.*

[18]. Springenberg, J. T., Dosovitskiy, A., Brox, T. & Riedmiller, M., 2014. Striving for simplicity: The all convolutional net.

[19]. Swethalakshmi, H., Jayaraman, A., Chakravarthy, V. S. & Sekhar, C. C., 2006. Online handwritten character recognition of Devanagari and Telgu characters using Support Vector Machine. 6 October.

[20]. Szegedy, C. et al., 2015. Going deeper with convolutions. *Conference on Computer Vision and Pattern Recognition (CVPR).*

Appendix

Sample of each Devanagari characters from the dataset

Characters	Images	Characters	Images	Characters	Images
1. Ka		13. Daa		25. Ma	
2. Kha		14. Dhaa		26. Yaw	
3. Ga		15. Adna		27. Ra	
4. Gha		16. Ta		28. Lah	
5. Na'		17. Tha		29. Woh	
6. Cha		18. The		30. Moto Sha	
7. Chha		19. Dha		31. Petchirya Sha	
8. Ja		20. Na		32. Patalo Sha	
9. Jha		21. Pa		33. Ha	
10. Yna		22. Pha		34. Chhya	
11. Taa		23. Ba		35. Tra	
12. Thaa		24. Bha		36. Gya	

Sample of Devanagari numbers from the dataset

Numbers	Images	Numbers	Images
0 (Sunya)		Pach (5)	
1 (Ek)		Chha (6)	
2 (Dui)		Sat (7)	
3 (Tin)		Aath (8)	
4 (Char)		Nau (9)	

Preparation of the Dataset
Screenshots of the process on preparing the data step by step

```python
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from tqdm import tqdm
import random
```

```python
DATA_train= "/Users/parasgurung/Desktop/Dataset/Train"
CATEGORIES = ["character_1_ka","character_2_kha","character_3_ga","character_4_gha","character_5_kna","character_6_cha'
DATA_test= "/Users/parasgurung/Desktop/Dataset/Test"
```

```python
training_data = []

def create_training_data():
    for category in CATEGORIES:  # characters

        path = os.path.join(DATA_train,category)  # create path to characters
        class_num = CATEGORIES.index(category)  # get the classification

        for img in tqdm(os.listdir(path)):  # iterate over each image per categories
            try:
                img_array = cv2.imread(os.path.join(path,img) ,cv2.IMREAD_GRAYSCALE)  # convert to array

                training_data.append([img_array, class_num])  # add this to our training_data
            except Exception as e:  # in the interest in keeping the output clean...
                pass

create_training_data()

print(len(training_data))
```

```python
testing_data=[]
def create_testing_data():
    for category in CATEGORIES:  # characters

        path = os.path.join(DATA_test,category)  # create path to characters
        class_num = CATEGORIES.index(category)  # get the classification

        for img in tqdm(os.listdir(path)):  # iterate over each image per categories
            try:
                img_array = cv2.imread(os.path.join(path,img) ,cv2.IMREAD_GRAYSCALE)  # convert to array

                testing_data.append([img_array, class_num])  # add this to our training_data
            except Exception as e:  # in the interest in keeping the output clean...
                pass

create_testing_data()

print(len(testing_data))
```

```
100%|          | 300/300 [00:00<00:00, 1926.00it/s]
100%|          | 300/300 [00:00<00:00, 2264.27it/s]
100%|          | 300/300 [00:00<00:00, 1988.93it/s]
100%|          | 300/300 [00:00<00:00, 2383.09it/s]
100%|          | 300/300 [00:00<00:00, 2210.30it/s]
100%|          | 300/300 [00:00<00:00, 1942.95it/s]
100%|          | 300/300 [00:00<00:00, 1797.88it/s]
100%|          | 300/300 [00:00<00:00, 2130.32it/s]
100%|          | 300/300 [00:00<00:00, 1568.24it/s]
100%|          | 300/300 [00:00<00:00, 2052.12it/s]
100%|          | 300/300 [00:00<00:00, 2323.18it/s]
100%|          | 300/300 [00:00<00:00, 2242.66it/s]
100%|          | 300/300 [00:00<00:00, 2122.00it/s]
100%|          | 300/300 [00:00<00:00, 1934.55it/s]
100%|          | 300/300 [00:00<00:00, 2004.50it/s]
100%|          | 300/300 [00:00<00:00, 1680.70it/s]
100%|          | 300/300 [00:00<00:00, 1698.94it/s]
100%|          | 300/300 [00:00<00:00, 1076.95it/s]
100%|          | 300/300 [00:00<00:00, 1670.05it/s]
100%|          | 300/300 [00:00<00:00, 1216.84it/s]
100%|          | 300/300 [00:00<00:00, 2057.39it/s]
100%|          | 300/300 [00:00<00:00, 2885.45it/s]
100%|          | 300/300 [00:00<00:00, 2371.11it/s]
100%|          | 300/300 [00:00<00:00, 2274.07it/s]
100%|          | 300/300 [00:00<00:00, 2914.26it/s]
100%|          | 300/300 [00:00<00:00, 1587.33it/s]
100%|          | 300/300 [00:00<00:00, 1827.10it/s]
100%|          | 300/300 [00:00<00:00, 1827.30it/s]
100%|          | 300/300 [00:00<00:00, 1591.07it/s]
100%|          | 300/300 [00:00<00:00, 1221.38it/s]
100%|          | 300/300 [00:00<00:00, 1093.77it/s]
100%|          | 300/300 [00:00<00:00, 2124.36it/s]
100%|          | 300/300 [00:00<00:00, 2283.30it/s]
100%|          | 300/300 [00:00<00:00, 2237.52it/s]
100%|          | 300/300 [00:00<00:00, 2843.69it/s]
100%|          | 300/300 [00:00<00:00, 2949.73it/s]
100%|          | 300/300 [00:00<00:00, 2939.28it/s]
100%|          | 300/300 [00:00<00:00, 2673.32it/s]
100%|          | 300/300 [00:00<00:00, 2550.95it/s]
100%|          | 300/300 [00:00<00:00, 2182.40it/s]
100%|          | 300/300 [00:00<00:00, 2880.24it/s]
100%|          | 300/300 [00:00<00:00, 2875.77it/s]
100%|          | 300/300 [00:00<00:00, 2537.24it/s]
100%|          | 300/300 [00:00<00:00, 2275.69it/s]
100%|          | 300/300 [00:00<00:00, 2912.96it/s]
```

```
100%|          | 300/300 [00:00<00:00, 2057.39it/s]
100%|          | 300/300 [00:00<00:00, 2885.45it/s]
100%|          | 300/300 [00:00<00:00, 2371.11it/s]
100%|          | 300/300 [00:00<00:00, 2274.07it/s]
100%|          | 300/300 [00:00<00:00, 2914.26it/s]
100%|          | 300/300 [00:00<00:00, 1587.33it/s]
100%|          | 300/300 [00:00<00:00, 1827.10it/s]
100%|          | 300/300 [00:00<00:00, 1827.30it/s]
100%|          | 300/300 [00:00<00:00, 1591.07it/s]
100%|          | 300/300 [00:00<00:00, 1221.38it/s]
100%|          | 300/300 [00:00<00:00, 1093.77it/s]
100%|          | 300/300 [00:00<00:00, 2124.36it/s]
100%|          | 300/300 [00:00<00:00, 2283.30it/s]
100%|          | 300/300 [00:00<00:00, 2237.52it/s]
100%|          | 300/300 [00:00<00:00, 2843.69it/s]
100%|          | 300/300 [00:00<00:00, 2949.73it/s]
100%|          | 300/300 [00:00<00:00, 2939.28it/s]
100%|          | 300/300 [00:00<00:00, 2673.32it/s]
100%|          | 300/300 [00:00<00:00, 2550.95it/s]
100%|          | 300/300 [00:00<00:00, 2182.40it/s]
100%|          | 300/300 [00:00<00:00, 2880.24it/s]
100%|          | 300/300 [00:00<00:00, 2875.77it/s]
100%|          | 300/300 [00:00<00:00, 2537.24it/s]
100%|          | 300/300 [00:00<00:00, 2912.96it/s]
100%|          | 300/300 [00:00<00:00, 2223.56it/s]

13800
```

```python
#shuffling the data
random.shuffle(training_data)
random.shuffle(testing_data)
```

```python
#making x and y array and adding the features and labels to it for training data
X_train = []
y_train = []
IMG_SIZE=32
for features,label in training_data:
    X_train.append(features)
    y_train.append(label)
```

```python
#making x and y array and adding the features and labels to it for testing data
X_test = []
y_test = []
for features,label in testing_data:
    X_test.append(features)
    y_test.append(label)
```
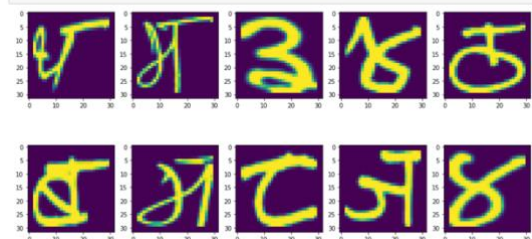
```python
#plotting the first 10 images of training dataset
fig1 = plt.figure(figsize = (15,15))

for i in range(5):
    ax1 = fig1.add_subplot(1,5,i+1)
    ax1.imshow(X_train[i], interpolation='none')
    ax2 = fig1.add_subplot(2,5,i+6)
    ax2.imshow(X_train[i+6], interpolation='none')
plt.show()
```



```python
import pickle
#save data by pickle module
pickle_out = open("X_train.pickle","wb")
pickle.dump(X_train, pickle_out)
pickle_out.close()

pickle_out = open("y_train.pickle","wb")
pickle.dump(y_train, pickle_out)
pickle_out.close()

pickle_out = open("X_test.pickle","wb")
pickle.dump(X_test, pickle_out)
pickle_out.close()

pickle_out = open("y_test.pickle","wb")
pickle.dump(y_test, pickle_out)
pickle_out.close()
```

Training the NepNet on the dataset
Screenshots of processes while training the NepNet step by step

```
[14] import tensorflow as tf

     from tensorflow.keras.models import Sequential, load_model
     from tensorflow.keras.layers import Input, Dropout, Flatten, Conv2D, MaxPooling2D, Dense, Activation, Reshape, BatchNormalization, GlobalAveragePooling2D
     from tensorflow.keras.preprocessing.image import ImageDataGenerator
     from tensorflow.keras.callbacks import ModelCheckpoint, Callback, EarlyStopping
     from tensorflow.keras.optimizers import SGD, Adam
     from tensorflow.keras.utils import to_categorical

     import numpy as np
     import matplotlib.pyplot as plt
     import pickle
```

```
[16] #unpacking the data saved using pickle
     pickle_in = open("X_train.pickle","rb")
     X_train_o = pickle.load(pickle_in)

     pickle_in = open("y_train.pickle","rb")
     y_train_o = pickle.load(pickle_in)

     pickle_in = open("X_test.pickle","rb")
     X_test_o = pickle.load(pickle_in)

     pickle_in = open("y_test.pickle","rb")
     y_test_o = pickle.load(pickle_in)
```

```
#categorising the y dataset and reshaping the x dataset

IMG_SIZE=32
y_train = to_categorical(y_train_o)
y_test = to_categorical(y_test_o)
X_train = np.array(X_train_o).reshape(-1, IMG_SIZE, IMG_SIZE,1)
X_test = np.array(X_test_o).reshape(-1, IMG_SIZE, IMG_SIZE,1)

print('The shape of x train data: {}'.format(X_train.shape))
print('The shape of y train data: {}'.format(y_train.shape))
print('The shape of x test data: {}'.format( X_test.shape))
print('The shape of y test data: {}'.format( y_test.shape))
```

```
The shape of x train data: (78200, 32, 32, 1)
The shape of y train data: (78200, 46)
The shape of x test data: (13800, 32, 32, 1)
The shape of y test data: (13800, 46)
```

```
#parameter for compiling the model
opt= Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
objective = 'categorical_crossentropy'

#parameter for the training (fit)
epoch = 40
batch_size = 64

# Callback for loss logging per epoch
class LossHistory(Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
        self.val_losses = []
        self.acc = []
        self.val_acc = []

    def on_epoch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        self.val_losses.append(logs.get('val_loss'))
        self.acc.append(logs.get('accuracy'))
        self.val_acc.append(logs.get('val_accuracy'))
#save the best model i.e. with lowest validation loss during the training
mc = ModelCheckpoint('best_model2.h5', monitor='val_loss', mode='min', verbose=1,save_best_only=True)


def NepNet():
    model = Sequential()
    model.add(Conv2D(64, (3, 3), padding="same", input_shape=((32,32,1)))) #Convolutional layer-1
    model.add(BatchNormalization()) #Batch Normalistion
    model.add(Activation("relu"))#activation funtion

    model.add(Conv2D(128, (3, 3), padding="same")) #Convolutional layer-2
    model.add(BatchNormalization())
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))#max-pooling

    model.add(Conv2D(256, (3, 3), padding="same")) #Convolutional layer-3
    model.add(BatchNormalization())
    model.add(Activation("relu"))

    model.add(Conv2D(512, (3, 3), padding="same"))#Convolutional layer-4
    model.add(BatchNormalization())
    model.add(Activation("relu"))
    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))#max-pooling

    model.add(Conv2D(512, (1, 1), padding="same")) #Convolutional layer-5
    model.add(BatchNormalization())
    model.add(Activation("relu"))

    model.add(Conv2D(46, (1, 1), padding="same"))#Convolutional layer-6
    model.add(BatchNormalization())
    model.add(Activation("relu"))

    model.add(GlobalAveragePooling2D()) #Global Average Pooling

    model.add(Dense(46, activation='softmax'))
    model.compile(loss=objective, optimizer=opt, metrics=['accuracy'])
    return model
```

```python
model= NepNet()
def run_NepNet():

    history = LossHistory()
    model.fit(X_train, y_train, batch_size=batch_size, epochs=epoch,
              validation_split=0.05, verbose=1, shuffle=True, callbacks=[history,mc])


    predictions = model.predict(X_test, verbose=0)
    return predictions, history
```

```python
predictions, history = run_NepNet()
```

```
Epoch 00025: val_loss did not improve from 0.02598
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0029 - accuracy: 0.9993 - val_loss: 0.0276 - val_accuracy: 0.9931
Epoch 26/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0037 - accuracy: 0.9991
Epoch 00026: val_loss improved from 0.02598 to 0.02097, saving model to best_model2.h5
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0037 - accuracy: 0.9991 - val_loss: 0.0210 - val_accuracy: 0.9944
Epoch 27/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0016 - accuracy: 0.9996
Epoch 00027: val_loss did not improve from 0.02097
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0016 - accuracy: 0.9996 - val_loss: 0.0846 - val_accuracy: 0.9762
Epoch 28/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0034 - accuracy: 0.9991
Epoch 00028: val_loss did not improve from 0.02097
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0034 - accuracy: 0.9991 - val_loss: 0.0266 - val_accuracy: 0.9959
Epoch 29/40
1160/1161 [=============================>.] - ETA: 0s - loss: 3.1413e-04 - accuracy: 1.0000
Epoch 00029: val_loss improved from 0.02097 to 0.01678, saving model to best_model2.h5
1161/1161 [==============================] - 25s 21ms/step - loss: 3.1404e-04 - accuracy: 1.0000 - val_loss: 0.0168 - val_accuracy: 0.9967
Epoch 30/40
1161/1161 [==============================] - ETA: 0s - loss: 8.4624e-05 - accuracy: 1.0000
Epoch 00030: val_loss improved from 0.01678 to 0.01669, saving model to best_model2.h5
1161/1161 [==============================] - 25s 21ms/step - loss: 8.4624e-05 - accuracy: 1.0000 - val_loss: 0.0167 - val_accuracy: 0.9974
Epoch 31/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0033 - accuracy: 0.9989
Epoch 00031: val_loss did not improve from 0.01669
1161/1161 [==============================] - 25s 21ms/step - loss: 0.0033 - accuracy: 0.9989 - val_loss: 1.3377 - val_accuracy: 0.7606
Epoch 32/40
1160/1161 [=============================>.] - ETA: 0s - loss: 0.0038 - accuracy: 0.9990
Epoch 00032: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0038 - accuracy: 0.9990 - val_loss: 0.0260 - val_accuracy: 0.9939
Epoch 33/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0021 - accuracy: 0.9995
Epoch 00033: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0021 - accuracy: 0.9995 - val_loss: 0.0502 - val_accuracy: 0.9882
Epoch 34/40
1160/1161 [=============================>.] - ETA: 0s - loss: 0.0021 - accuracy: 0.9995
Epoch 00034: val_loss did not improve from 0.01669
1161/1161 [==============================] - 25s 21ms/step - loss: 0.0021 - accuracy: 0.9995 - val_loss: 0.0597 - val_accuracy: 0.9839
Epoch 35/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0024 - accuracy: 0.9994
Epoch 00035: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0024 - accuracy: 0.9994 - val_loss: 0.0397 - val_accuracy: 0.9923
Epoch 36/40
1159/1161 [=============================>.] - ETA: 0s - loss: 0.0025 - accuracy: 0.9993
Epoch 00036: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0025 - accuracy: 0.9993 - val_loss: 0.0243 - val_accuracy: 0.9951
Epoch 37/40
1160/1161 [=============================>.] - ETA: 0s - loss: 4.6972e-04 - accuracy: 1.0000
Epoch 00037: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 4.6956e-04 - accuracy: 1.0000 - val_loss: 0.0218 - val_accuracy: 0.9957
Epoch 38/40
1161/1161 [==============================] - ETA: 0s - loss: 0.0039 - accuracy: 0.9988
Epoch 00038: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 0.0039 - accuracy: 0.9988 - val_loss: 0.0339 - val_accuracy: 0.9918
Epoch 39/40
1160/1161 [=============================>.] - ETA: 0s - loss: 6.5752e-04 - accuracy: 0.9999
Epoch 00039: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 6.5719e-04 - accuracy: 0.9999 - val_loss: 0.0313 - val_accuracy: 0.9934
Epoch 40/40
1159/1161 [=============================>.] - ETA: 0s - loss: 2.7866e-04 - accuracy: 1.0000
Epoch 00040: val_loss did not improve from 0.01669
1161/1161 [==============================] - 24s 21ms/step - loss: 2.8313e-04 - accuracy: 1.0000 - val_loss: 0.0257 - val_accuracy: 0.9946
```

```python
#checking the test accuracy of the moel after 40 epoch
def test_accuracy():
    err = []
    t = 0
    for i in range(predictions.shape[0]):
        if (np.argmax(predictions[i]) == y_test_o[i]):
            t = t+1
        else:
            err.append(i)
    return t, float(t)*100/predictions.shape[0], err, len(err)*100/predictions.shape[0]

result=test_accuracy()
print("Test accuracy: {} %".format(result[1]))
print("No. of mis-classification: {} ".format(len(result[2])))
print("Error rate: {} %".format(result[3]))
```

```
Test accuracy: 99.54347826086956 %
No. of mis-classification: 63
Error rate: 0.45652173913043476 %
```
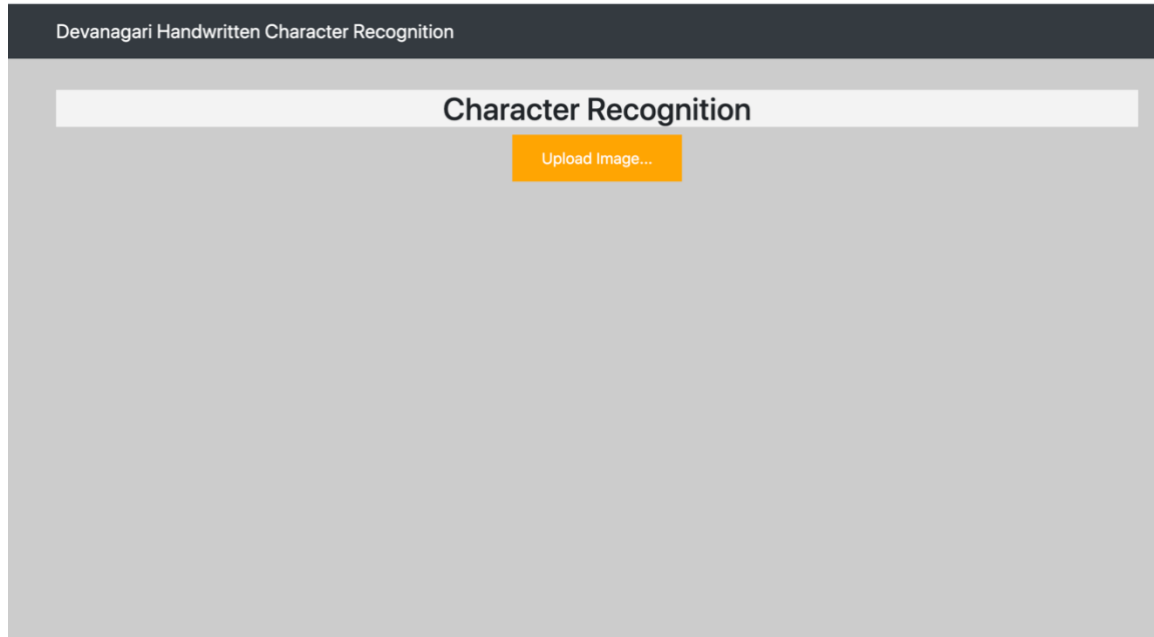
```python
# checking the test accuracy of the best model i.e with lowest validation loss over the 40 epochs
savedmodel = load_model('best_model2.h5')
prediction_2 = savedmodel.predict(X_test, verbose=0)
def test_accuracy_2():
    err = []
    t = 0
    for i in range(prediction_2.shape[0]):
        if (np.argmax(prediction_2[i]) == y_test_o[i]):
            t = t+1
        else:
            err.append(i)
    return t, float(t)*100/prediction_2.shape[0], err, len(err)*100/predictions.shape[0]

t_acc=test_accuracy_2()
print("Test accuracy: {} %".format(t_acc[1]))
print("No. of mis-classification: {} ".format(len(t_acc[2])))
print("Error rate: {} %".format(t_acc[3]))
```

```
Test accuracy: 99.64492753623189 %
No. of mis-classification: 49
Error rate: 0.35507246376811596 %
```

**Word Recognition Web Application**

Screenshot of Interface of the

Using the Python Flask, HTML, JavaScript and CSS, the web application system was developed. It enables users to upload images containing Devanagari words, from their local drive by clicking on the on the upload image button. When an image is uploaded, it will display the uploaded image and the predict button as shown in the screenshot below.
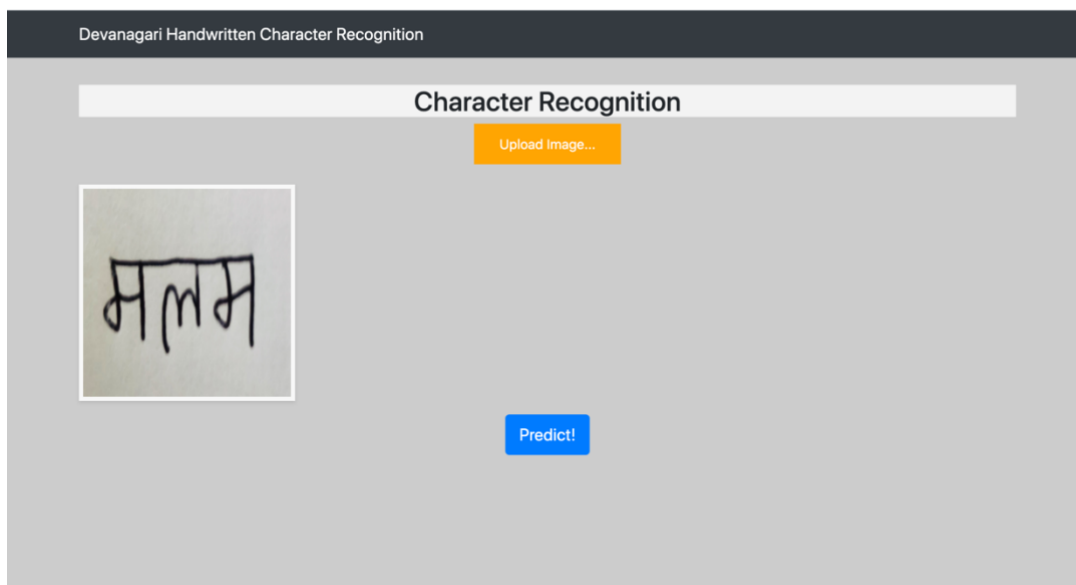
Screenshot of Interface after upload of an image

Screenshot after prediction

Few Screenshots of the word recognition system

Character Recognition

Upload Image...
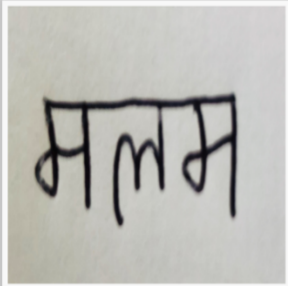
Result: The word in the image is हषठ७

Few screenshots on the operation of the character recognition application

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

क

क   95%
फ 3%
ठ 1%

Latency: 0.06s

CLEAR   SUBMIT   SCREENSHOT   FLAG

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

फ

फ   100%
झ 0%
0 0%

Latency: 0.06s
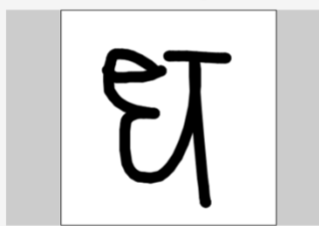
CLEAR   SUBMIT   SCREENSHOT   FLAG

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

ख

ख   100%
स 0%
श 0%

Latency: 0.06s

CLEAR   SUBMIT   SCREENSHOT   FLAG

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

स

स   100%
म 0%
श 0%

Latency: 0.06s

CLEAR   SUBMIT   SCREENSHOT   FLAG

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

घ

घ   97%
ध 2%
छ 1%

Latency: 0.06s

CLEAR   SUBMIT   SCREENSHOT   FLAG

Devanagari Character Recognition

Draw Devanagari Character on sketchpad, and click submit to see the NepNet's predictions. Model trained on the UCI Devanagari dataset.

INPUT   OUTPUT

ध

ध   83%
थ 13%
घ 4%

Latency: 0.06s

CLEAR   SUBMIT   SCREENSHOT   FLAG

Screenshots of the character that are misclassified

Drew 'Tha' but 'Yah' is predicted

Drew 'Dha' but 'Gha is predicted

When the characters are misclassified, they are flagged using the 'Flag' button and the drawn character image is saved in the flagged folder for the future reformation.