

AI BASED HANDWRITTEN RECOGNITION

Submitted in partial fulfillment of the requirements for
the award of
Bachelor of Engineering degree in Computer Science and Engineering

by

NELAKUDITI VENKATA PRANEETH(Reg.No.37110505)

MUSUNURI YUGANDHAR(Reg.No.37110471)



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF COMPUTING**

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

**Accredited with Grade "A" by NAAC
JEPPIAAR NAGAR, RAJIV GANDHI
SALAI, CHENNAI – 600 119**

MARCH - 2021



SATHYABAMA



INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with "A" grade by NAAC
Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600 119
www.sathyabama.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this project report is the bonafide work of **MUSUNURIYUGANDHAR (Reg. No. 37110471)** and **NELAKUDITI VENKATA PRANEETH (Reg. No.37110505)** who carried out the project entitled "**AI BASED HANDWRITTEN RECOGNITION**" under my supervision from **August 2020** to **March 2021**

Internal Guide
Mrs. L.Sujihelen

Head of the Department

Submitted for Viva voce Examination held on_____

Internal Examiner

External Examiner

DECLARATION

I **NELAKUDITI VENKATA PRANEETH** hereby declare that the Project Report entitled "**AI BASED HAND WRITTEN RECOGNITION**" is done by me under the guidance of **Dr. L.Sujihelen**, Department of Computer Science and Engineering at Sathyabama Institute of Science and Technology is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Computer Science and Engineering.

DATE:

PLACE: CHENNAI

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of Sathyabama Institute of Science and Technology** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T. Sasikala, M.E., Ph.D., Dean**, School of Computing, **Dr. S. Vigneswari, M.E., Ph.D., and Dr. L. Lakshmanan, M.E., Ph.D., Heads of the Department** of Computer Science and Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr. L.Sujihelen, M.E.,Ph.D.**, for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science and Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

In this project, the work has been performed to recognize Handwritten English character using optical character recognition model. After recognising text we use google translate API to translate recognised text to another language. Character is identified by studying its shape and comparing its features that differentiate each character. This process will be repeated till every character is recognised. This system helps to recognize and translate recognised text to another language. The system was trained using 1500 samples of handwritings given by both male and female participants of distinct age groups. Test result was performed on 500 samples different from training data and we got good recognition accuracy of 94% for recognition and translating to different language.

TABLE OF CONTENTS

	ABSTRACT	v
	LIST OF FIGURES	Viii
	TITLE	PAGE No.
1.	INTRODUCTION	1
	1.1 INTRODUCTION	1
	1.2 INTRODUCTION TO OCR	1
	1.2.1 Automatic Identification	1
2.	LITERATURE SURVEY	4
3.	AIM AND SCOPE	7
	3.1 AIM OF THE PROJECT	7
	3.1.1 Objectives	7
	3.2 SCOPE	7
	3.2.1 Advantages	8
4.	METHODOLOGY	10
	4.1 DATASET	10
	4.2 SEGMENTATION	11
	4.3 PREPROCESSING	12
	4.4 FEATURE EXTRACTION	14
	4.4.1 OpenCV	15
	4.4.2 Numpy	16
	4.4.3 Padding	19
	4.4.4 Stride	20
	4.5 CONVOLUTION NEURAL NETWORKS	21
	4.5.1 Convolution for images	24
	4.5.2 Filter Kernel	26
	4.5.3 Pooling	28
	4.5.4 ReLU activation function	30

4.6 RECURRENT NEURAL NETWORKS	32
4.6.1 Back Propagation	32
4.6.2 Long short term memory(LSTM)	33
4.6.3 CTC Loss and Decode	34
4.7 GOOGLE TRANSLATE API	35
5. RESULTS AND DISCUSSION	36
5.1 RESULTS	36

6.	SUMMARY AND CONCLUSION	38
	6.1 CONCLUSION	
	6.2 FUTURE ENHANCEMENT	
	REFERENCES	40

LIST OF FIGURES

FIGURE No.	FIGURE NAME	PAGE No.
1	CHARACTER RECOGNITION	3
2	HANDWRITTEN IMAGE	10
3	IMAGE SEGMENTATION	11
4	ARCHITECTURE DIAGRAM	14
5	CROSS-CORRELATION	21
6	KERNEL CALCULATION	26
7	MAXIMUM POOLING	29
8	OUTPUT MATRIX OF NN	33

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Nueral Networks are used in various fields nowadays. It is mainly used for recognising pattern in images. Each person has different style of handwritings distinguishing from each other. So it is difficult to recognize the Handwritten characters. In recognising of Handwritten character Nueral Networks plays important role. Numerous reports of character recognition are published but the accuarcy is not good which makes it as an open problem that needs to be solved. In this paper attempt have been made to develop good accurate model for Handwriting recognition and translation. The system was trained using 1500 samples of handwritten data which contains lines, words, and characters. We pre-process our training data and divide them in to batches. After pre-processing we send pre-processed data in batches to optical character recognition model for extracting features of the image and training the image data. Optical character recognition model contains CNN and RNN layers. CNN layers helps in feature extracting and RNN helps in training image data. We set a certain epoch number and train our model till every character is recognized. We use word beamsearch for recognising characters accurately. After recognising characters our next step is to translate the recognized text to another language. We use google translate API for translating the recognized text to another language. Observational result shows that the approach we used in this paper is producing good accuracy at recognition and translating text.

1.2 INTRODUCTION TO OCR

Optical character recognition belongs to the family of techniques performing automatic identification. Below we discuss these different techniques and define OCR's position among them.

1.2.1 Automatic Identification

The traditional way of entering data into a computer is through the keyboard. However, this is not always the best nor the most efficient solution. In many cases automatic identification

may be an alternative. Various technologies for automatic identification exist, and they cover needs for different areas of application. Below a brief overview of the different technologies

and their applications is given.

Speech recognition

In systems for speech recognition, spoken input from a predefined library of words are recognized. Such systems should be speaker-independent and may be used for instance for reservations or ordering of goods by telephone. Another kind of such systems are those used to recognize the speaker, rather than the words, for identification.

Magnetic ink

Printing in magnetic ink is mainly used within bank applications. The characters are written in ink that contains finely ground magnetic material and they are written in stylized fonts which are specifically designed for the application. Before the characters are read, the ink is exposed to a magnetic field. This process accentuates each character and helps simplify the detection. The characters are read by interpreting the waveform obtained when scanning the characters horizontally. Each character is designed to have its own specific waveform. Although designed for machine reading, the characters are still readable to humans. However, the reading is dependent on the characters being printed with magnetic ink.

Optical Character Recognition

Optical character recognition is needed when the information should be readable both to humans and to a machine and alternative inputs can not be predefined. In comparison with the other techniques for automatic identification, optical character recognition is unique in that it does not require control of the process that produces the information.

Optical Character Recognition deals with the problem of recognizing optically processed characters. Optical recognition is performed off-line after the writing or printing has been completed, as opposed to on-line recognition where the computer recognizes the characters as they are drawn. Both hand printed and printed characters may be recognized, but the performance is directly dependent upon the quality of the input documents.

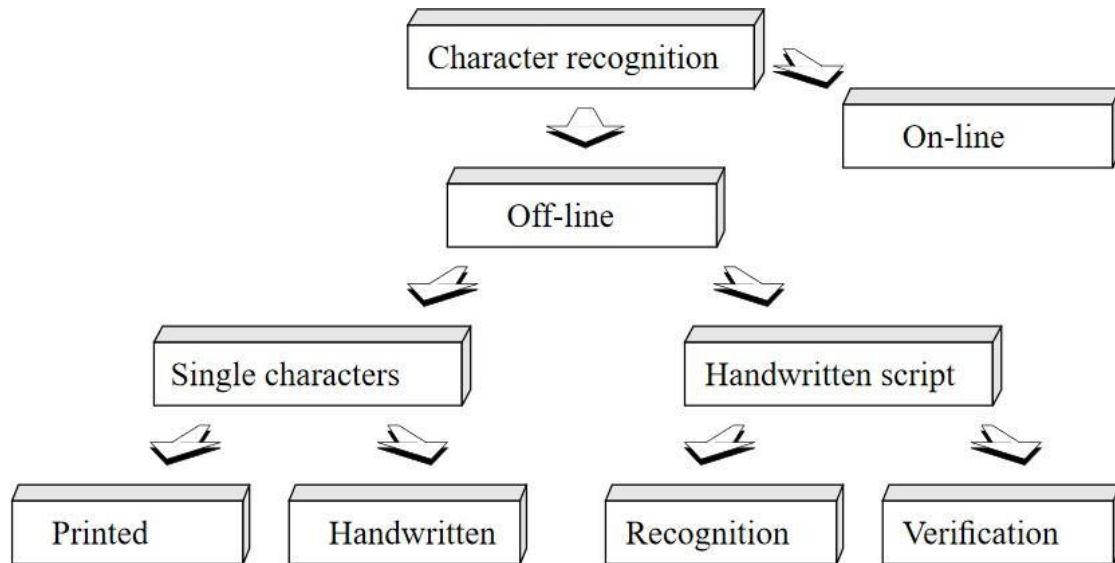


Figure 1 : The different areas of character recognition.

The more constrained the input is, the better will the performance of the OCR system be. However, when it comes to totally unconstrained handwriting, OCR machines are still along way from reading as well as humans. However, the computer reads fast and technical advances are continually bringing the technology closer to its ideal.

The main principle in automatic recognition of patterns, is first to teach the machine which classes of patterns that may occur and what they look like. In OCR the patterns are letters, numbers and some special symbols like commas, question marks etc., while the different classes correspond to the different characters. The teaching of the machine is performed by showing the machine examples of characters of all the different classes. Based on these examples the machine builds a prototype or a description of each class of characters. Then, during recognition, the unknown characters are compared to the previously obtained descriptions, and assigned the class that gives the best match.

CHAPTER 2

LITERATURE SURVEY

K. Gaurav, Bhatia P. K. [1] Et al, paper manages the different pre-preparing strategies associated with character recognition with various sort of images goes from straightforward Handwritten structure based reports, records containing hued, complex foundation and changed forces. The disconnected character recognition was proposed by sing slanting component extraction. It upheld the ANN model. There are two methodologies for making neural organization framework like 54 element and 69 highlights. In this, extraordinary preprocessing strategies like slant identification and revision, image upgrade the methods of difference extending, binarization, commotion evacuation procedures, standardization along with division, morphological processing procedures were talked about. It is reasoned utilizing a solitary method for preprocessing, we cannot totally handle image. In any case, subsequent to applying every one of the procedures may impractical to accomplish the full precision in a preprocessing framework. Salvador España-Boquera et al [2], in this paper cross breed Hidden Markov Model (HMM) model was proposed for perceiving unconstrained disconnected Handwritten writings. In this, primary piece of optical model has been displayed with Markov chains, and a Multi layer Perceptron was utilized to appraise the emanation probabilities A. Brakensiek et al[3] in this paper framework for the disconnected cursive penmanship recognition was depicted that depends on Hidden Markov Models (HMM) utilizing separate and mixture displaying procedures. Penmanship recognition tests utilizing discrete and two diverse mixture draws near, which comprise of a discrete and semi-constant construction, are analyzed. A division free methodology was considered to build up the framework. It is discovered that the recognition rate execution maybe improved of a half and half demonstrating strategy for HMMs, which relies upon neural vector quantizer (crossover MMI), contrasted with discrete and mixture HMMs, in view of weary combination structure (crossover -TP), which might be brought about general little informational index.

L. Bajaj et al[4] has utilized three distinct styles of highlights, for example, the thickness highlights, second highlights, and, clear segment highlights for characterization of the Devanagari Numerals. they acquired 89.6% precision for Handwritten Devanagari

numerals. have actualized this utilizing a fluffy set utilizing the container approach and the recognition is 90%. This model works on shifted data sources. In the past explores, plainly this model is effective with assorted data sources yet it comes up short on a limited quantity of precision

inside the instance of the long sentences. Many proposed models don't appear to be so effective inaccurately grouping long content information. On the opposite side models are consolidating CNN organizations and showing great outcomes which are result of its capacity of managing more extended content information.

M. Mohammed et al[6],this paper depicts which Recognition of characters enormously relies on highlights utilized. A few highlights of the written by hand Arabic characters are chosen and examined. A disconnected recognition framework dependent on the chose highlights which was fabricated. Frame work was prepared and tried with practical examples of Handwritten Arabic characters. Assessment of significance and exactness of chose highlights are made. The recognition dependent on chose highlights give normal exactness of 88% and 70% for the numbers and letters, separately. Further upgrades were accomplished by utilizing highlight freight dependent on bits of knowledge acquired from exactness of individual highlights. G. Pirlo et al[10], introduced another class of participation capacities, which are called Fuzzy membership capacities (FMFs), for drafting based characterization. These FMFs can be handily adjusted to a particular qualities of characterization issue to augment order execution. In this exploration, a realcoded hereditary calculation is introduced to discover, in a solitary improvement strategy, the ideal FMF, along with the ideal drafting depicted by Voronoi decoration. test results, which were completed in the field of Handwritten digit and character recognition, demonstrate which ideal FMF performs better compared to other participation capacities dependent on theoretical level, positioned level, and estimation level weighting models, which can be found in writing.

S.N O	AUTHOR & YEAR	TITLE	GOALS	FUTURE PERSPECTIVE
----------	---------------------	-------	-------	-----------------------

1	Som et al., (2011)	Handwritten Character Recognition Using Fuzzy Membership Function	Fluffy participation work based methodology for HCR.	It can be used to increase the recognition rate of handwritten letters.
2	Salvador España-Boquera et al.,(2011)	Improving Offline Handwritten Text Recognition with Hybrid HMM Models	(HMM) model was proposed for perceiving unconstrained disconnected Handwritten writings.	Deep learning method would lead us to better trained ANNs, which could improve every step of our recognition engine.
3	Renata et al.,(2011)	SVM Based Off-Line Handwritten Digit Recognizer	SVM based dis connected written by hand digit recognition. Creators guarantees that SVM outflanks the Multilayer perceptron classifier.	Recognition rate can be improved.

4	G. Pirlo et al.,(2011)	Fuzzy Zoning Based Classification for Handwritten Character	FMF performs better compared to other participation capacities dependent on theoretical level.	Box approach can be used for recognition of handwritten numeral characters.
---	------------------------	---	--	---

--	--	--	--	--

CHAPTER 3

AIM AND SCOPE

3.1 AIM OF PROJECT

The primary aim of this proposed system i.e. AI based hand written recognition is to recognize Handwritten English character using optical character recognition model. After recognizing text we use google translate API to translate recognized text to another language. Character is identified by studying its shape and comparing its features that differentiate each character. This process will be repeated till every character is recognized. This system helps to recognize and translate recognized text to another language. The system was trained using 1500 samples of hand writings given by both male and female participants of distinct age groups. Test result was performed on 500 samples different from training data and we got good recognition accuracy of 94% for recognition and translating to different language.

3.1.1 Objectives

The proposed model “AI based hand written recognition” is based on the new technologies which makes the whole process efficient. It is going to be very important and profitable for everyone. Each script has a set of icons, which are known as characters or letters, which have certain basic shapes. The goal of handwriting is to identify input characters or image correctly then analyzed to many automated process systems. This system will be applied to detect the writings of different format and after recognizing the text we will translate the recognized text to our convenient language.

3.2 SCOPE

This model will help in recognizing the hand written text and able to translate the recognized

text to different languages. During this approach, the system is trained to seek out the similarities, and also the differences among various **handwritten** samples. This application takes the image of a hand transcription **and** converts it into a digital text. After converting into digital text our application will help to translate the recognized text to different languages.

3.2.1 Advantages

Handwriting Recognition has many advantages that made it grow rapidly in the technology word now. There is much different kind of technologies that abide to enable others to take advantage of the handwriting recognition. The way this work was when people write letters a different way and they let the computer know what the intended letter was and change in to a text document. But the problem with this was the different way that the letters are written and this could make unnatural feel to the person who is writing it. Another way of appreciating this technology is that just write and the computer changes it to a text document but at the same time, the computer doesn't always get the right word and sometimes it insert the wrong letters. Certain cell phones have the handwriting recognition system in it. The advantage of this is that it allows people to write on the cell phones using stylus and then the phone software translates the written words to the phone in text. But at the same time, the problem or called as disadvantage of this again is the same problem that the original software face, which is the unnatural letters strokes and sometimes the software predict the letters wrongly. This is one of the reasons that not all phones have this software in it.

Additionally, there are many more advantages of handwriting recognition system. Data storage, for an example, there are many files, contracts and some personal records that contains some handwritten information and notes. Certain of these documents contains original signature or notes that might not be electronically stored but this problems are overcome with the use of handwriting recognition system. Handwriting recognition software allows user to translate all those signature and notes into electronic words in a text document format. The advantage of this electronic storage is that this data only requires far less physical space than the storage of the physical copies. Another advantage of the electronic storage is also that it requires fewer employees to sort the documents through, organize and to keep the data storage warehouse.

Other than that, data retrieval is another advantage of handwriting recognition. Physical data retrieval always requires personnel to sort through physical copies of old information. The data must always have been stored and correctly organized and also it must have proper

maintenance and upkeep on the physical copies. To retain this information or data, we perform electronic data retrieval by using a file search by using specific keywords, for example, like the names and the dates of the file or document. Handwriting recognition software allows the old files to be saved in a proper electronic format. This is how handwriting recognition software helps in saving the old files or important document. For an example, some clinics prefer keeping their patient's medical records and handwriting recognition helps in this situation by keeping these medical records safe in the computer. This data or files can be reviewed and updated time to time when needed without worrying if the data will be lost.

Moreover, another advantage of handwriting recognition is historical preservation. Historical papers mostly exist in physical format. Examples of historical papers are genealogical information, written manuscripts, old family records, and some personal diaries and sometimes even shared old past stories. But still sometimes, these historical papers might be damaged or corrupted due to some accidents and there is when handwriting recognition software is really very helpful. Handwriting recognition helps to transform the writings in the papers to a text document format which can also be said as readable electronic format. By this way, historical facts can be stored, reviewed and shared easily to many people.

Lastly, the advantage is textual studies. A textual study is a category of literature studies. Literature studies involve reviewing the original manuscripts of literature in comparison with the printed version. This means that the story has been dug and changed at the hands of the editor's hand after coming from the hands of the author. This is why original manuscripts are cared very well but still this requires a complete review of the manuscript. Handwriting recognition helps to keep safe these original manuscripts in an electronic format and it can be reviewed without damaging the original copy of the manuscripts.

4.1 DATASET

**C
H
A
P
T
E
R
4
M
E
T
H
O
D
O
L
O
G
Y**

IAM Handwriting Dataset is a collection of handwritten passages by several writers. Generally, they use that data to classify writers according to their writing styles. A traditional way of solving such problem is extracting features like spacing between letters, curvatures, etc. and feeding them into Support Vector Machines. But, I wanted to solve this problem by Deep learning using Keras and Tensorflow. For the purpose, we don't need the full IAM Handwriting Dataset, but some authentic subset which can be used for training such as a subset of images by top 50 persons who contributed the most towards the dataset.

The database contains forms of unconstrained handwritten text, which were scanned at a resolution of 300dpi and saved as PNG images with 256 gray levels. All forms and also all extracted text lines, words and sentences are available for download as PNG files, with corresponding XML meta-information included into the image files. All texts in the IAM database are built using sentences provided by the LOB Corpus.

The IAM Handwriting Database 3.0 is structured as follows:

- 657 writers contributed samples of their handwriting
- 1539 pages of scanned text
- 5685 isolated and labeled sentences
- 13353 isolated and labeled text lines
- 115320 isolated and labeled words

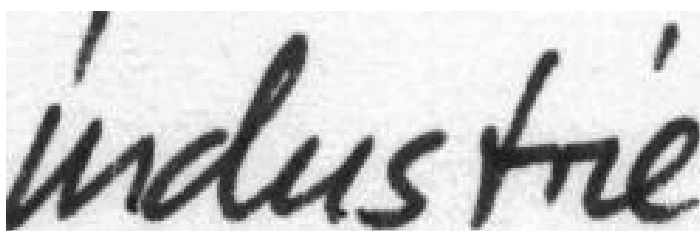


Figure 2: Handwritten image

The figure 2 provides samples of a complete form, a text line and some extracted words.

4.2 SEGMENTATION

Segmentation is a process that determines the constituents of an image. It is necessary to locate the regions of the document where data have been printed and distinguish them from figures and graphics. For instance, when performing automatic mail-sorting, the address must be located and separated from other print on the envelope like stamps and

company logos, prior to recognition.

Applied to text, segmentation is the isolation of characters or words. The majority of optical character recognition algorithms segment the words into isolated characters which are recognized individually. Usually this segmentation is performed by isolating each connected component, that is each connected black area. This technique is easy to implement, but problems occur if characters touch or if characters are fragmented and consist of several parts. The main problems in segmentation may be divided into four groups:

- Extraction of touching and fragmented characters.

Such distortions may lead to several joint characters being interpreted as one single character, or that a piece of a character is believed to be an entire symbol. Joints will occur if the document is a dark photocopy or if it is scanned at a low threshold. Also joints are common if the fonts are serified. The characters may be split if the document stems from a light photocopy or is scanned at a high threshold.

- Distinguishing noise from text.

Dots and accents may be mistaken for noise, and vice versa.

- Mistaking graphics or geometry for text.

This leads to non text being sent to recognition.

- Mistaking text for graphics or geometry. In this case the text will not be passed to the recognition stage.

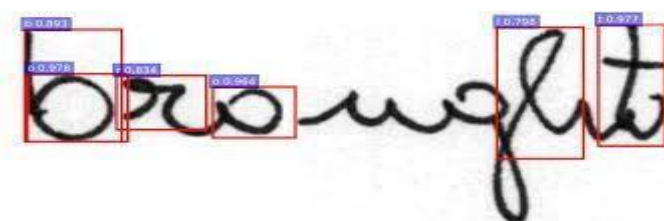


Figure 3: Image Segmentation.

4.3 PREPROCESSING

The image resulting from the scanning process may contain a certain amount of noise. Depending on the resolution on the scanner and the success of the applied technique for thresholding, the characters may be smeared or broken. Some of these defects, which may later cause poor recognition rates, can be eliminated by using a preprocessor to smooth the digitized characters. The smoothing implies both filling and thinning. Filling eliminates small breaks, gaps and holes in the digitized characters, while thinning reduces the width of

the line. The most common techniques for smoothing, moves a window across the binary image

of the character, applying certain rules to the contents of the window.

In addition to smoothing, preprocessing usually includes normalization. The normalization is applied to obtain characters of uniform size, slant and rotation. To be able to correct for rotation, the angle of rotation must be found. For rotated pages and lines of text, variants of Hough transform are commonly used for detecting skew. However, to find the rotation angle of a single symbol is not possible until after the symbol has been recognized. There are four methods of Data Preprocessing which are:

- Data Cleaning

Real-world data tend to be incomplete, noisy, and inconsistent. Data Cleaning/Cleansing routines attempt to fill in missing values, smooth out noise while identifying outliers, and correct inconsistencies in the data.

Data can be noisy, having incorrect attribute values. Owing to the following, the data collection instruments used may be faulty. Maybe human or computer errors occurred at data entry. Errors in data transmission can also occur.

“Dirty” data can cause confusion for the mining procedure. Although most mining routines have some procedures, they deal incomplete or noisy data, which are not always robust. Therefore, a useful Data Preprocessing step is to run the data through some Data Cleaning/Cleansing routines.

- Data Integration

Data Integration is involved in data analysis task which combines data from multiple sources into a coherent data store, as in data warehousing. These sources may include multiple databases, data cubes, or flat files. The issue to be considered in Data Integration is schema integration. It is tricky.

How can real-world entities from multiple data sources be ‘matched up’? This is referred as entity identification problem. For example, how can a data analyst be sure that `customer_id` in one database and `cust_number` in another refer to the same entity? The answer is metadata. Databases and data warehouses typically have metadata. Simply, metadata is

data about data.

Metadata is used to help avoiding errors in schema integration. Another important issue is redundancy. An attribute may be redundant, if it is derived from another table. Inconsistencies in attribute or dimension naming can also cause redundancies in the resulting data set.

▪ Data Transformation

In Normalisation, where the attribute data are scaled to fall within a small specified range, such as -1.0 to 1.0, or 0 to 1.0. Smoothing works to remove the noise from the data. Such techniques include binning, clustering, and regression. In Aggregation, summary or aggregation operations are applied to the data. For example, daily sales data may be aggregated so as to compute monthly and annual total amounts. This step is typically used in constructing a data cube for analysis of the data at multiple granularities. In Generalisation of the Data, low level or primitive/raw data are replaced by higher level concepts through the use of concept hierarchies. For example, categorical attributes are generalised to higher level concepts street into city or country. Similarly, the values for numeric attributes may be mapped to higher level concepts like, age into young, middle-aged, or senior.

▪ Data Reduction

Complex data analysis and mining on huge amounts of data may take a very long time, making such analysis impractical or infeasible. Data Reduction techniques are helpful in analysing the reduced representation of the data set without compromising the integrity of the original data and yet producing the qualitative knowledge. Strategies for data reduction include the following:

- In Data Cube Aggregation, aggregation operations are applied to the data in the construction of a data cube.
- In Dimension Reduction, irrelevant, weakly relevant, or redundant attributes or dimensions may be detected and removed.
- In Data Compression, encoding mechanisms are used to reduce data set size. The methods used for Data Compression are Wavelet Transform and Principle Component Analysis.
- In Numerosity Reduction, data is replaced or estimated by alternative and smaller data representations such as parametric models (which store only the model parameters instead of the actual data, e.g. Regression and Log-Linear Models) or non-parametric methods

(e.g. Clustering, Sampling, and the use of histograms).

- In Discretisation and Concept Hierarchy Generation, raw data values for attributes are

replaced by ranges or higher conceptual levels. Concept hierarchies allow the mining of data at multiple levels of abstraction and are powerful tools for data mining.

After preprocessing data set, our next step is extracting features and training our extracted features on a model.

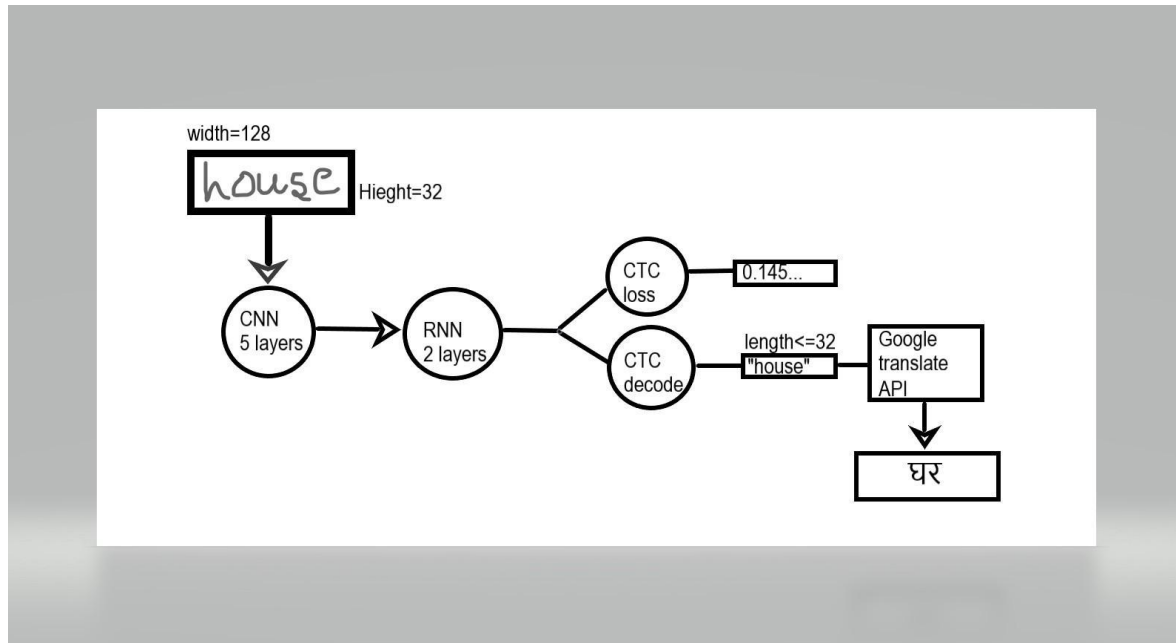


Figure 4: Architecture Diagram

4.4 FEATURE EXTRACTION

The objective of feature extraction is to capture the essential characteristics of the symbols, and it is generally accepted that this is one of the most difficult problems of pattern recognition. The most straight forward way of describing a character is by the actual raster image. Another approach is to extract certain features that still characterize the symbols, but leaves out the unimportant attributes. The different groups of features may be evaluated according to their sensitivity to noise and deformation and the ease of implementation and use.

The technique of extracting the features is useful when you have a large data set and need to reduce the number of resources without losing any important or relevant information. Feature extraction helps to reduce the amount of redundant data from the data set. In the end, the reduction of the data helps to build the model with less machine's efforts and also increase the speed of learning and generalization steps in the machine learning process. Image processing is one of the best and most interesting domain. In this domain basically

you will start playing with your images in order to understand them. So here we use many

many techniques which includes feature extraction as well and algorithms to detect features such as shaped, edges, or motion in a digital image or video to process them.

Machines see any images in the form of a matrix of numbers. The size of this matrix actually depends on the number of pixels of the input image. The Pixel Values for each of the pixels stands for or describe how bright that pixel is, and what color it should be. So In the simplest case of the binary images, the pixel value is a 1-bit number indicating either foreground or background. Smaller numbers which is closer to zero helps to represent black, and the larger numbers which is closer to 255 denote white. But, for the case of a coloured image, we have three Matrices or the channels. So in these three matrices, each of the matrix has values between 0-255 which represents the intensity of the colour of that pixel. So you can see we also have a three matrices which represents the channel of RGB – (for the three color channels – Red, Green, and Blue) On the right, we have three matrices. These three channels are superimposed and used to form a coloured image. So this is how a computer can differentiate between the images.

4.4.1 OpenCV

One of the most important and popular libraries is Opencv. It helps us to develop a system which can process images and real-time video using computer vision. OpenCv focused on image processing, real-time video capturing to detect faces and objects. OpenCV is one of the most popular and successful libraries for computer vision and it has an immense number of users because of its simplicity, processing time and high demand in computer vision applications. OpenCV-Python is like a python wrapper around the C++ implementation. OpenCv has more than 2500 implemented algorithms which are freely available for commercial purpose as well. There are many applications there using OpenCv which are really helpful and efficient. These applications are also taking us towards a more advanced world with less human efforts.

Here, the matter is straight forward. If pixel value is greater than a threshold value, it is assigned one value (may be white), else it is assigned another value (may be black). The function used is **cv2.threshold**. First argument is the source image, which **should be a grayscale image**. Second argument is the threshold value which is used to classify the pixel values. Third argument is the maxVal which represents the value to be given if pixel

value is more than (sometimes less than) the threshold value. OpenCV provides different styles of thresholding and it is decided by the fourth parameter of the function. We used a global value

as threshold value. But it may not be good in all the conditions where image has different lighting conditions in different areas. In that case, we go for adaptive thresholding. In this, the algorithm calculate the threshold for a small regions of the image. So we get different thresholds for different regions of the same image and it gives us better results for images with varying illumination. In global thresholding, we used an arbitrary value for threshold value, right? So, how can we know a value we selected is good or not? Answer is, trial and error method. But consider a **bimodal image** (*In simple words, bimodal image is an image whose histogram has two peaks*). For that image, we can approximately take a value in the middle of those peaks as threshold value, right ? That is what Otsu binarization does. So in simple words, it automatically calculates a threshold value from image histogram for a bimodal image. (For images which are not bimodal, binarization won't be accurate.) For this, our `cv2.threshold()` function is used, but pass an extra flag. **For threshold value, simply pass zero**. Then the algorithm finds the optimal threshold value and returns you as the second output. If Otsu thresholding is not used, `retVal` is same as the threshold value you used.

4.4.2 Numpy

NumPy stands for 'Numerical Python'. It is a package for data analysis and scientific computing with Python. NumPy uses a multidimensional array object, and has functions and tools for working with these arrays. The powerful n-dimensional array in NumPy speeds-up data processing. NumPy can be easily interfaced with other Python packages and provides tools for integrating with other programming languages like C, C++ etc. NumPy arrays are used to store lists of numerical data, vectors and matrices. The NumPy library has a large set of routines (built-in functions) for creating, manipulating, and transforming NumPy arrays. Python language also has an array data structure, but it is not as versatile, efficient and useful as the NumPy array. The NumPy array is officially called `ndarray` but commonly known as array. In rest of the chapter, we will be referring to NumPy array whenever we use "array". NumPy array takes up less space in memory as compared to a list because arrays do not require to store datatype of each element separately.

There are several ways to create arrays. To create an array and to use its methods, first we need to import the NumPy library. The NumPy's `array()` function converts a given list into an

array. An array with only single row of elements is called 1-D array. Let us try to create a 1-D array from a list which contains numbers as well as strings. `array2 = np.array([5,-`

7.4,'a',7.2])). Observe that since there is a string value in the list, all integer and float values have been promoted to string, while converting the list to array. We can create a two dimensional (2-D) arrays by passing nested lists to the `array()` function. Some important attributes of a NumPy ndarray object are:

- `ndarray.ndim`: gives the number of dimensions of the array as an integer value. Arrays can be 1-D, 2-D or n-D. In this chapter, we shall focus on 1-D and 2-D arrays only. NumPy calls the dimensions as axes (plural of axis). Thus, a 2-D array has two axes. The row-axis is called axis-0 and the column-axis is called axis-1. The number of axes is also called the array's rank.
- `ndarray.shape`: It gives the sequence of integers indicating the size of the array for each dimension.
- `ndarray.size`: It gives the total number of elements of the array. This is equal to the product of the elements of shape.
- `ndarray.dtype`: is the data type of the elements of the array. All the elements of an array are of same data type. Common data types are `int32`, `int64`, `float32`, `float64`, `U32`, etc.
- `ndarray.itemsize`: It specifies the size in bytes of each element of the array. Data type `int32` and `float32` means each element of the array occupies 32 bits in memory. 8 bits form a byte. Thus, an array of elements of type `int32` has `itemsize 32/8=4` bytes. Likewise, `int64/float64` means each item has `itemsize 64/8=8` bytes.

We can specify data type (integer, float, etc.) while creating array using `dtype` as an argument to `array()`. This will convert the data automatically to the mentioned type. In the following example, nested list of integers are passed to the array function. Since data type has been declared as float, the integers are converted to floating point numbers. We can create an array with all elements initialised to 0 using the function `zeros()`. By default, the data type of the array created by `zeros()` is float. The following code will create an array with 3 rows and 4 columns with each element set to 0. We can create an array with all elements initialised to 1 using the function `ones()`. By default, the data type of the array created by `ones()` is float. The following code will create an array with 3 rows and 2 columns. We can create an array with numbers in a given range and sequence using the `arange()` function. This function is analogous to the `range()` function of Python.

For 2-D arrays indexing for both dimensions starts from 0, and each element is referenced

through two indexes *i* and *j*, where *i* represents the row number and *j* represents the column number. Sometimes we need to extract part of an array. This is done through slicing. We can define which part of the array to be sliced by specifying the start and end index values using [start : end] along with the array name. If row indices are not specified, it means all the rows are to be considered. Likewise, if column indices are not specified, all the columns are to be considered. Once arrays are declared, we can access its element or perform certain operations. In the last section, we learnt about accessing elements. Arithmetic operations on NumPy arrays are fast and simple. When we perform a basic arithmetic operation like addition, subtraction, multiplication, division etc. on two arrays, the operation is done on each corresponding pair of elements. For instance, adding two arrays will result in the first element in the first array to be added to the first element in the second array, and so on. It is important to note that for element-wise operations, size of both arrays must be same. That is, array1.shape must be equal to array2.shape. In 2-D array, sorting can be done along either of the axes i.e., row-wise or column-wise. By default, sorting is done row-wise (i.e., on axis

= 1). It means to arrange elements in each row in ascending order. When axis=0, sorting is done column-wise, which means each column is sorted in ascending order. Concatenation means joining two or more arrays. Concatenating 1-D arrays means appending the sequences one after another. NumPy.concatenate() function can be used to concatenate two or more 2-D arrays either row-wise or column-wise. All the dimensions of the arrays to be concatenated must match exactly except for the dimension or axis along which they need to be joined. Any mismatch in the dimensions results in an error. By default, the concatenation of the arrays happens along axis=0. We can modify the shape of an array using the reshape() function. Reshaping an array cannot be used to change the total number of elements in the array. Attempting to change the number of elements in the array using reshape() results in an error. We can split an array into two or more subarrays. numpy.split() splits an array along the specified axis. We can either specify sequence of index values where an array is to be split; or we can specify an integer N, that indicates the number of equal parts in which the array is to be split, as parameter(s) to the NumPy.split() function. By default, NumPy.split() splits along axis = 0.

Sometimes, we may have data in files and we may need to load that data in an array for

processing. `numpy.loadtxt()` and `numpy.genfromtxt()` are the two functions that can be used to load data from text files. The most commonly used file type to handle large amount of data is called CSV (Comma Separated Values). Each row in the text file must have the same

number of values in order to load data from a text file into a numpy array. The parameter `skiprows=1` indicates that the first row is the header row and therefore we need to skip it as we do not want to load it in the array. The delimiter specifies whether the values are separated by comma, semicolon, tab or space (the four are together called whitespace), or any other character. The default value for delimiter is space. We can also specify the data type of the array to be created by specifying through the `dtype` argument. By default, `dtype` is float. We can load each row or column of the data file into different numpy arrays using the `unpack` parameter. By default, `unpack=False` means we can extract each row of data as separate arrays. When `unpack=True`, the returned array is transposed means we can extract the columns as separate arrays. `genfromtxt()` is another function in NumPy to load data from files. As compared to `loadtxt()`, `genfromtxt()` can also handle missing values in the data file. The `genfromtxt()` function converts missing values and character strings in numeric columns to `nan`. But if we specify `dtype` as `int`, it converts the missing or other non numeric values to

-1. We can also convert these missing values and character strings in the data files to some specific value using the parameter `filling_values`.

4.4.3 Padding

Padding is a term relevant to convolution neural networks as it refers to the amount of pixels added to an image when it is being processed by the kernel of a CNN. For example, if the padding in a CNN is set to zero, then every pixel value that is added will be of value zero. If, however, the zero padding is set to one, there will be a one pixel border added to the image with a pixel value of zero. Padding works by extending the area of which a convolution neural networks processes an image. The kernel is the neural networks filter which moves across the image, scanning each pixel and converting the data into a smaller, or sometimes larger, format. In order to assist the kernel with processing the image, padding is added to the frame of the image to allow for more space for the kernel to cover the image. Adding padding to an image processed by a CNN allows for more accurate analysis of images. In general, if we add a total of ph rows of padding (roughly half on top and half on bottom) and a total of pw columns of padding (roughly half on the left and half on the right), the output shape will be $(nh-kh+ph+1) \times (nw-kw+pw+1)$.

In many cases, we will want to set $ph=kh-1$ and $pw=kw-1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that kh is odd here, we will pad $ph/2$ rows on both

sides of the height. If kh is even, one possibility is to pad $\lceil ph/2 \rceil$ rows on the top of the input and $\lfloor ph/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way. CNNs commonly use convolution kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right. Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional tensor X , when the kernel's size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output $Y[i, j]$ is calculated by cross-correlation of the input and convolution kernel with the window centered on $X[i, j]$.

4.4.4 Stride

When computing the cross-correlation, we start with the convolution window at the top-left corner of the input tensor, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one element at a time. However, sometimes, either for computational efficiency or because we wish to down sample, we move our window more than one element at a time, skipping the intermediate locations. We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. Fig.5 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. The shaded portions are the output elements as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$. We can see that when the second element of the first column is outputted, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is outputted. When the convolution window continues to slide two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

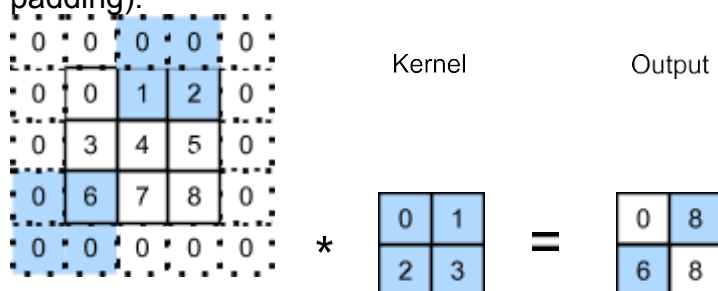


Fig.5 Cross-correlation with strides of 3 and 2 for height and width, respectively.

For the sake of brevity, when the padding number on both sides of the input height and width are ph and pw respectively, we call the padding (ph, pw) . Specifically, when $ph=pw=p$, the padding is p . When the strides on the height and width are sh and sw , respectively, we call the stride (sh, sw) . Specifically, when $sh=sw=s$, the stride is s . By default, the padding is 0 and the stride is 1. In practice, we rarely use inhomogeneous strides or padding, i.e., we usually have $ph=pw$ and $sh=sw$. Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input. The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1). Padding and stride can be used to adjust the dimensionality of the data effectively.

4.5 CONVOLUTION NEURAL NETWORKS

We had implemented 5 layers in Convolution neural networks. Each layer consists of some set of operations. The first operation we included is filter kernel, which helps in edge detection, sharpening. This is done by doing a convolution between a kernel and an image. Then the non-linear RELU function is applied $R(z)=\max(0,z)$. The main use of this non-linearity of RELU is we can build arbitrary shaped curves on the featured plane. After applying activation function pooling layer is used. In convolution neural networks we used pooling layer to reduce the size of the representation, which helps to speed up the computation process. The size of the image is downsized by summarizing input image into regions. For example we can reduce image of $4*4$ size to $2*2$ by applying pooling layer. By applying these operations to layers of my convolution neural network feature extraction of the image was done.

This chapter introduces convolution neural networks (CNNs), a powerful family of neural networks that are designed for precisely this purpose. CNN-based architectures are now ubiquitous in the field of computer vision, and have become so dominant that hardly anyone today would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without building off of this approach. Modern CNNs, as they are called colloquially owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their sample efficiency in achieving accurate models, CNNs tend to be computationally efficient,

both because they require fewer parameters than fully-connected architectures and because convolutions are easy to parallelize across GPU cores. Consequently, practitioners often

apply CNNs whenever possible, and increasingly they have emerged as credible competitors even on tasks with a one-dimensional sequence structure, such as audio, text, and time series analysis, where recurrent neural networks are conventionally used. Some clever adaptations of CNNs have also brought them to bear on graph-structured data and in recommender systems. First, we will walk through the basic operations that comprise the backbone of all convolutional networks. These include the convolutional layers themselves, nitty-gritty details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple channels at each layer, and a careful discussion of the structure of modern architectures. We will conclude the chapter with a full working example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning. In the next chapter, we will dive into full implementations of some popular and comparatively recent CNN architectures whose designs represent most of the techniques commonly used by modern practitioners.

To this day, the models that we have discussed so far remain appropriate options when we are dealing with tabular data. By tabular, we mean that the data consist of rows corresponding to examples and columns corresponding to features. With tabular data, we might anticipate that the patterns we seek could involve interactions among the features, but we do not assume any structure *a priori* concerning how the features interact. Sometimes, we truly lack knowledge to guide the construction of craftier architectures. In these cases, an MLP may be the best that we can do. However, for high-dimensional perceptual data, such structure-less networks can grow unwieldy. For instance, let us return to our running example of distinguishing cats from dogs. Say that we do a thorough job in data collection, collecting an annotated dataset of one-megapixel photographs. This means that each input to the network has one million dimensions. According to our discussions of parameterization cost of fully-connected layers. Even an aggressive reduction to one thousand hidden dimensions would require a fully-connected layer characterized by $10^6 \times 10^3 = 10^9$ parameters. Unless we have lots of GPUs, a talent for distributed optimization, and an extraordinary amount of patience, learning the parameters of this network may turn out to be infeasible. A careful reader might object to this argument on the basis that one megapixel resolution may not be necessary. However, while we might be able to get away with one hundred thousand pixels, our hidden layer of size 1000

grossly underestimates the number of hidden units that it takes to learn good representations of images, so a practical system will still require billions of parameters. Moreover, learning a classifier by fitting so many parameters might require

collecting an enormous dataset. And yet today both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these intuitions. That is because images exhibit rich structure that can be exploited by humans and machine learning models alike. Convolutional neural networks (CNNs) are one creative way that machine learning has embraced for exploiting some of the known structure in natural images. Convolution neural networks (CNNs) are a special family of neural networks that contain convolutional layers. In the deep learning research community, $\mathbf{V}$ is referred to as a *convolution kernel*, a *filter*, or simply the layer's *weights* that are often learnable parameters. When the local region is small, the difference as compared with a fully-connected network can be dramatic. While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred, without altering the dimensionality of either the inputs or the hidden representations. The price paid for this drastic reduction in parameters is that our features are now translation invariant and that our layer can only incorporate local information, when determining the value of each hidden activation. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models might struggle even to fit our training data. There is just one problem with this approach. So far, we blissfully ignored that images consist of 3 channels: red, green, and blue. In reality, images are not two-dimensional objects but rather third-order tensors, characterized by a height, width, and channel, e.g., with shape $1024 \times 1024 \times 3$ pixels. While the first two of these axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location. We thus index X as $[X]_{i,j,k}$. The convolutional filter has to adapt accordingly. Instead of $[\mathbf{V}]_{a,b}$, we now have $[V]_{a,b,c}$. Moreover, just as our input consists of a third-order tensor, it turns out to be a good idea to similarly formulate our hidden representations as third-order tensors H . In other words, rather than just having a single hidden representation corresponding to each spatial location, we want an entire vector of hidden representations corresponding to each spatial location. We could think of the hidden representations as comprising a number of two-dimensional grids stacked on top of each other. As in the inputs, these are sometimes called *channels*. They are also sometimes called *feature maps*, as each provides a spatialized set of learned features to

the subsequent layer. Intuitively, you might imagine that at lower layers that are closer to inputs, some channels could become specialized to recognize edges while others could recognize textures. There are still many operations that we need to address.

For instance, we need to figure out how to combine all the hidden representations to a single output, e.g., whether there is a Waldo *anywhere* in the image. We also need to decide how to compute things efficiently, how to combine multiple layers, appropriate activation functions, and how to make reasonable design choices to yield networks that are effective in practice. We turn to these issues in the remainder of the chapter.

4.5.1 Convolution for images

Recall that strictly speaking, convolutional layers are a misnomer, since the operations they express are more accurately described as cross-correlations. Based on our descriptions of convolutional layers. In such a layer, an input tensor and a kernel tensor are combined to produce an output tensor through a cross-correlation operation. Let us ignore channels for now and see how this works with two-dimensional data and hidden representations. In Fig.6, the input is a two-dimensional tensor with a height of 3 and width of 3. We mark the shape of the tensor as 3×3 or $(3, 3)$. The height and width of the kernel are both 2. The shape of the *kernel window* (or convolution window) is given by the height and width of the kernel (here it is 2×2).

Input		Kernel		Output																	
<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	$*$	<table border="1" style="border-collapse: collapse;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	$=$	<table border="1" style="border-collapse: collapse;"> <tr><td>19</td><td>25</td></tr> <tr><td>37</td><td>43</td></tr> </table>	19	25	37	43
0	1	2																			
3	4	5																			
6	7	8																			
0	1																				
2	3																				
19	25																				
37	43																				

Fig.6 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the top-left corner of the input tensor and slide it across the input tensor, both from left to right and top to bottom. When the convolution window slides to a certain position, the input subtensor contained in that window and the kernel tensor are multiplied elementwise and the resulting tensor is summed up yielding a single scalar value. This result gives the value of the output tensor at the corresponding location. Here, the output tensor has a height of 2 and width of 2 and the four elements are derived from the two-dimensional cross-correlation operation:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$

$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$

$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$

$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

Note that along each axis, the output size is slightly smaller than the input size. Because the kernel has width and height greater than one, we can only properly compute the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $nh \times nw$ minus the size of the convolution kernel $kh \times kw$ via $(nh - kh + 1) \times (nw - kw + 1)$. This is the case since we need enough space to “shift” the convolution kernel across the image. Later we will see how to keep the size unchanged by padding the image with zeros around its boundary so that there is enough space to shift the kernel. Next, we implement this process in the `corr2d` function, which accepts an input tensor X and a kernel tensor K and returns an output tensor Y . A convolutional layer cross-correlates the input and kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer. We are now ready to implement a two-dimensional convolutional layer based on the `corr2d` function defined above. In the `_init_constructor` function, we declare weight and bias as the two model parameters. The forward propagation function calls the `corr2d` function and adds the bias. In $h \times w$ convolution or a $h \times w$ convolution kernel, the height and width of the convolution kernel are h and w , respectively. We also refer to a convolutional layer with a $h \times w$ convolution kernel simply as a $h \times w$ convolutional layer. Designing an edge detector by finite differences $[1, -1]$ is neat if we know this is precisely what we are looking for. However, as we look at larger kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing manually. Now let us see whether we can learn the kernel that generated Y from X by looking at the input–output pairs only. We first construct a convolutional layer and initialize its kernel as a random tensor. Next, in each iteration, we will use the squared error to compare Y with the output of the convolutional layer. We can then calculate the gradient to update the kernel. For the sake of simplicity, in the following we use the built-in class for two-dimensional convolutional layers and ignore the bias.

4.5.2 Filter kernel

Convolution is using a kernel to extract certain features from the input image. Let me

explain. A kernel is a matrix, which is slid across the image and multiplied with the input such that the

output is enhanced in a certain desirable manner. For example, the kernel used above is useful for sharpening the image. But what is so special about this kernel?? Consider the two input image arrangements as shown in the example below. For the first image, the center value is $3 \times 5 + 2 \times -1 + 2 \times -1 + 2 \times -1 + 2 \times -1 = 7$. The value 3 got increased to 7. For the second image, the output is $1 \times 5 + 2 \times -1 + 2 \times -1 + 2 \times -1 + 2 \times -1 = -3$. The value 1 got decreased to -3. Clearly, the contrast between 3 and 1 is increased to 7 and -3, which will in turn sharpen the Image. Instead of using manually made kernels for feature extraction, through Deep CNNs we can learn these kernel values which can extract latent features.

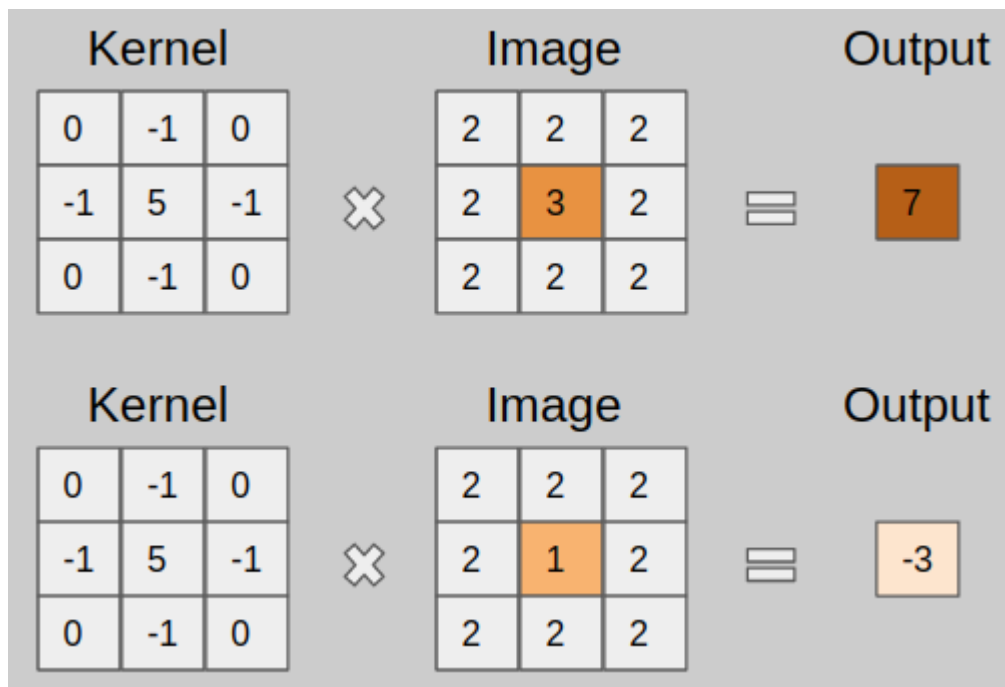


Fig.6 Kernel calculation

A kernel is, as described earlier, a matrix of weights which are multiplied with the input to extract relevant features. The dimensions of the kernel matrix is how the convolution gets its name. For example, in 2D convolutions, the kernel matrix is a 2D matrix. A filter however is a concatenation of multiple kernels, each kernel assigned to a particular channel of the input. Filters are always one dimension more than the kernels. For example, in 2D convolutions, filters are 3D matrices (which is essentially a concatenation of 2D matrices i.e. the kernels). So for a CNN layer with kernel dimensions $h \times w$ and input channels k , the filter dimensions are $k \times h \times w$. A common convolution layer actually consists of multiple such filters. For the sake of simplicity in the discussion to follow, assume the presence of only one filter unless specified, since the same behaviour is replicated across

all the filters.

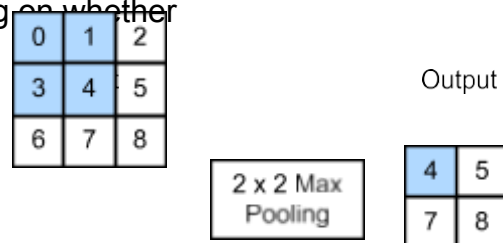
1D convolutions are commonly used for time series data analysis (since the input in such cases is 1D). As mentioned earlier, the 1D data input can have multiple channels. The filter can move in one direction only, and thus the output is 1D. Let's visualize a multi channel 2D convolution and try to wrap our heads around it. In the diagram below, the kernel dimensions are 3×3 and there are multiple such kernels in the filter (marked yellow). This is because there are multiple channels in the input (marked blue) and we have one kernel corresponding to every channel in the input. Clearly, here the filter can move in 2 directions and thus the final output is 2D. 2D convolutions are the most common convolutions, and are heavily used in Computer Vision. It is difficult to visualize a 3D filter (since it's a 4D dimensional matrix), so we will discuss single channel 3D convolution here. As you can see from the image below, in 3D convolutions, a kernel can move in 3 directions and thus the output obtained is also 3D. Most of the work done in modifying and customizing CNN layers have been focused towards 2D convolutions only and so from this point forward I will only be discussing these variations in context of 2D convolutions. Separable Convolution refers to breaking down the convolution kernel into lower dimension kernels. Separable convolutions are of 2 major types. First are spatially separable convolutions, However, spatially separable convolutions are not that common in Deep Learning. On the other hand, Depthwise separable convolutions are widely used in light weight CNN models and provide really good performances. There have been multiple variations of CNN layers which have been used independently or in combination with each other to create successful and complex architectures. Each variation was born out of an intuition of how feature extraction should work. Thus I believe that while these Deep CNN networks learn weights that we cannot explain, the intuitions involved in forming them is very important to their performance and further work in that direction is important for success of highly complex CNNs.

4.5.3 Pooling

Often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations, aggregating information so that the higher up we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive. Often our ultimate task asks some global question about the image, So typically the units of our final layer should be sensitive to the entire input. By gradually aggregating information, yielding

coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing. Moreover, when detecting lower-level features, such as edges, we often

want our representations to be somewhat invariant to translation. For instance, if we take the image X with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., $Z[i, j] = X[i, j + 1]$, then the output for the new image Z might be vastly different. The edge will have shifted by one pixel. In reality, objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem). Like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the pooling window). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no kernel). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called maximum pooling (max *pooling* for short) and average pooling, respectively. In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the top left of the input tensor and sliding across the input tensor from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether



max or average pooling .

Fig.7 Maximum pooling with a pooling window shape of 2×2 . The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

A pooling layer with a pooling window shape of $p \times q$ is called a $p \times q$ pooling layer. The pooling operation is called $p \times q$ pooling. Let us return to the object edge detection example mentioned at the beginning of this section. Now we will use the output of the convolutional layer as the input for 2×2 maximum pooling. Set the convolutional layer input as X and the pooling layer output as Y . Whether or not the values of $X[i, j]$ and $X[i, j + 1]$ are different, or

$X[i, j + 1]$ and $X[i, j + 2]$ are different, the pooling layer always outputs $Y[i, j] = 1$. That is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional

layer moves no more than one element in height or width. As with convolutional layers, pooling layers can also change the output shape. And as before, we can alter the operation to achieve a desired output shape by padding the input and adjusting the stride. We can demonstrate the use of padding and strides in pooling layers via the built-in two-dimensional maximum pooling layer from the deep learning framework. We first construct an input tensor X whose shape has four dimensions, where the number of examples and number of channels are both 1. When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate tensors X and $X + 1$ on the channel dimension to construct an input with 2 channels. Taking the input elements in the pooling window, the maximum pooling operation assigns the maximum value as the output and the average pooling operation assigns the average value as the output. One of the major benefits of a pooling layer is to alleviate the excessive sensitivity of the convolutional layer to location.

4.5.4 ReLU activation function

Almost anyone who has worked with neural networks has used the ReLU function. It's the modern standard for an activation function, but there are a few questions as to how it works. Neural network need nonlinearity to address nonlinear problems, and usually the only place where that nonlinearity can be injected is in the activation function. Yet ReLU, while strictly mathematically is not a linear function, is comprised of two linear functions (piecewise linear). First, it's important to gain a solid understanding of why adding nonlinearity into a neural network is desired. In a standard neural network, each layer multiplies the previous one by a set of weights and adds a bias. In order to transform vectors from a length of m into a length of n , the weights take the form of an n by m matrix (multiplied by a m by 1 matrix, or a vector). Even after the inputs are passed through several layers of the network, the transformations applied are purely linear; that is, allowing freedom of movement only by multiplying by a coefficient and adding a bias. Hence, the network will only be able to address problems that cannot be separated by the limited movement of the strictly linear function. With much of the complex problems that dominate the applications of neural networks, there is simply no place for the limited power of a linear

neural network. As a demonstration of a more rigorous necessity, the universal approximation theorem requires nonlinear activation functions for it to be valid. The theorem states, in essence, that under a few conditions, any continuous function f can be modelled with a neural network with one hidden layer and a sufficient

number of units. One of the requirements is that the activation function must have some form of nonlinearity. To generalize, in order to approach nonlinear problems, there must be some sort of nonlinearity in the network. ReLU has become the darling activation function of the neural network world. Short for Rectified Linear Unit, it is a piecewise linear function that is defined to be 0 for all negative values of x and equal to $a \times x$ otherwise, where a is a learnable parameter. What may seem unintuitive is how this piecewise linear function adds nonlinearity into a network. After all, it is still linear. While it is obviously not completely linear, the mathematical definition of nonlinearity is not satisfying or intuitive. Compared to other contenders for the activation function throne — SoftMax, sigmoid, and tanh — ReLU lacks the clean curves, and instead simply seems to be the linear function's brother.

Whereas tanh, a smooth, curved function, draws a clean envelope around the circle (and linear fails completely), ReLU draws a hexagon, with several pointed corners. In fact, this is what ReLU's advantage is: it can bend the linear function at a certain point, to a certain degree. Combined with the biases and weights from the previous layer, the ReLU can take the form of a bend at any location at any degree. These small bends form the building blocks of approximations. Any relationship or function can be roughly estimated by aggregating many ReLU functions together, which occurs when neurons are collapsed and combined in the following layer. This has been mathematically proven, for example, with a sine wave or an exponential function, much like how a lower-degree Taylor series fits a function. The strength of the ReLU function lies not in itself, but in an entire army of ReLUs. This is why using a few ReLUs in a neural network does not yield satisfactory results; instead, there must be an abundance of ReLU activations to allow the network to construct an entire map of points. In multi-dimensional space, rectified linear units combine to form complex polyhedra along the class boundaries. Here lies the reason why ReLU works so well: when there are enough of them, they can approximate any function just as well as other activation functions like sigmoid or tanh, much like stacking hundreds of Legos, without the downsides. There are several issues with smooth-curve functions that do not occur with ReLU — one being that computing the derivative, or the rate of change, the driving force behind gradient descent, is much cheaper with ReLU than with any other smooth-curve function. Another is that sigmoid and other curves have an issue with the vanishing gradient problem; because the derivative of the sigmoid function gradually slopes off for larger absolute values of x . Because the distributions of inputs may shift around

heavily earlier during training away from 0, the derivative will be so small that no useful information can be back propagated to update the weights. This is often a major problem in neural network training. On the other hand, the

derivative of the ReLU function is simple; it's the slope of whatever line the input is on. It will reliably return a useful gradient, and while the fact that $x = 0 \{x < 0\}$ may sometimes lead to a 'dead neuron problem', ReLU has still shown to be, in general, more powerful than not only curved functions (sigmoid, tanh) but also ReLU variants attempting to solve the dead neuron problem, like Leaky ReLU. ReLU is designed to work in abundance; with heavy volume it approximates well, and with good approximation it performs just as well as any other activation function, without the downsides.

4.6 RECURRENT NEURAL NETWORKS

After extracting features from the image, our next step is to train our model with extracted features. We had implemented Recurrent neural networks to train our model. Recurrent neural networks are the state of the art algorithm for sequential data and are used by Apple's Siri and Google's voice search. It is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for machine learning problems that involve sequential data. It is one of the algorithms behind the scenes of the amazing achievements seen in deep learning over the past few years. RNNs are a powerful and robust type of neural network, and belong to the most promising algorithms in use because it is the only one with an internal memory. Because of their internal memory, RNN's can remember important things about the input they received, which allows them to be very precise in predicting what's coming next. This is why they're the preferred algorithm for sequential data like time series, speech, text, financial data, audio, video, weather and much more. Recurrent neural networks can form a much deeper understanding of a sequence and its context compared to other algorithms. Sequential data is basically just ordered data in which related things follow each other. Examples are financial data or the DNA sequence. The most popular type of sequential data is perhaps time series data, which is just a series of data points that are listed in time order.

4.6.1 Back propagation

In neural networks, you basically do forward-propagation to get the output of your model and check if this output is correct or incorrect, to get the error. Backpropagation is nothing but going backwards through your neural network to find the partial derivatives of the error with respect to the weights, which enables you to subtract this value from the weights.

Those derivatives are then used by gradient descent, an algorithm that can iteratively minimize a

given function. Then it adjusts the weights up or down, depending on which decreases the error. That is exactly how a neural network learns during the training process. So, with back propagation you basically try to tweak the weights of your model while training. You can view a RNN as a sequence of neural networks that you train one after another with backpropagation. Within BPTT the error is backpropagated from the last to the first timestep, while unrolling all the timesteps. This allows calculating the error for each timestep, which allows updating the weights. Note that BPTT can be computationally expensive when you have a high number of time steps.

A gradient is a partial derivative with respect to its inputs. If you don't know what that means, just think of it like this: a gradient measures how much the output of a function changes if you change the inputs a little bit. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. A gradient simply measures the change in all weights with regard to the change in error. Exploding gradients are when the algorithm, without much reason, assigns a stupidly high importance to the weights. Fortunately, this problem can be easily solved by truncating or squashing the gradients. Vanishing gradients occur when the values of a gradient are too small and the model stops learning or takes way too long as a result. This was a major problem in the 1990s and much harder to solve than the exploding gradients. Fortunately, it was solved through the concept of LSTM by Sepp Hochreiter and Juergen Schmidhuber.

4.6.2 Long short term memory(LSTM)

Long short-term memory networks are an extension for recurrent neural networks, which basically extends the memory. Therefore it is well suited to learn from important experiences that have very long time lags in between. The units of an LSTM are used as building units for the layers of a RNN, often called an LSTM network. LSTMs enable RNNs to remember inputs over a long period of time. This is because LSTMs contain information in a memory, much like the memory of a computer. The LSTM can read, write and delete information from its memory. This memory can be seen as a gated cell, with gated meaning the cell decides whether or not to store or delete information (i.e., if it opens the gates or not), based on the importance it assigns to the information. The assigning of importance happens through weights, which are also learned by the algorithm. This simply means that

it learns over time what information is important and what is not. In an LSTM you have three gates: input, forget

and output gate. These gates determine whether or not to let new input in (input gate), delete the information because it isn't important (forget gate), or let it impact the output at the current timestep (output gate). The gates in an LSTM are analog in the form of sigmoids, meaning they range from zero to one. The fact that they are analog enables them to do backpropagation. The problematic issues of vanishing gradients is solved through LSTM because it keeps the gradients steep enough, which keeps the training relatively short and the accuracy high.

4.6.3 CTC Loss and Decode

We need to calculate the loss value for the training samples (pairs of images and GT texts) to train the NN. You already know that the NN outputs a matrix containing a score for each character at each time-step. A minimalistic matrix is shown in Fig. 8: there are two time-steps (t_0 , t_1) and three characters ("a", "b" and the blank "-"). The character-scores sum to 1 for each time-step.

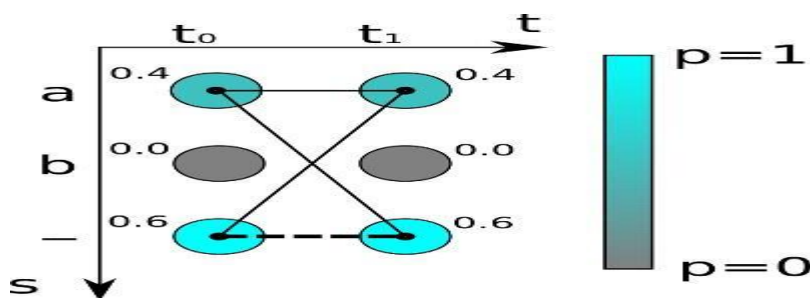


Fig. 8: Output matrix of NN. The character-probability is color-coded and is also printed next to each matrix entry. Thin lines are paths representing the text "a", while the thick dashed line is the only path representing the text "".

Further, you already know that the loss is calculated by summing up all scores of all possible alignments of the GT text, this way it does not matter where the text appears in the image. The score for one alignment (or **path**, as it is often called in the literature) is calculated by multiplying the corresponding character scores together. In the example shown above, the score for the path "aa" is $0.4 \cdot 0.4 = 0.16$ while it is $0.4 \cdot 0.6 = 0.24$ for "a-" and $0.6 \cdot 0.4 = 0.24$ for "- a". To get the score for a given GT text, we sum over the scores of all paths corresponding to this text. Let's assume the GT text is "a" in the example: we have to calculate all possible paths of length 2 (because the matrix has 2 time-steps), which are:

“aa”, “a-” and “-a”. We already calculated the scores for these paths, so we just have to sum over them and get

$0.4 \cdot 0.4 + 0.4 \cdot 0.6 + 0.6 \cdot 0.4 = 0.64$. If the GT text is assumed to be "", we see that there is only one corresponding path, namely "--", which yields the overall score of $0.6 \cdot 0.6 = 0.36$. We are now able to compute the probability of the GT text of a training sample, given the output matrix produced by the NN. The goal is to train the NN such that it outputs a high probability (ideally, a value of 1) for correct classifications. Therefore, we maximize the product of probabilities of correct classifications for the training dataset. For technical reasons, we re-formulate into an equivalent problem: minimize the loss of the training dataset, where the loss is the negative sum of log-probabilities. If you need the loss value for a single sample, simply compute the probability, take the logarithm, and put a minus in front of the result. To train the NN, the gradient of the loss with respect to the NN parameters (e.g., weights of convolutional kernels) is computed and used to update the parameters.

When we have a trained NN, we usually want to use it to recognize text in previously unseen images. Or in more technical terms: we want to calculate the most likely text given the output matrix of the NN. You already know a method to calculate the score of a given text. But this time, we are not given any text, in fact, it is exactly this text we are looking for. Trying every possible text would work if there are only a few time-steps and characters, but for practical use-cases, this is not feasible. It calculates the best path by taking the most likely character per time-step. It undoes the encoding by first removing duplicate characters and then removing all blanks from the path. What remains represents the recognized text. An example is shown in Fig. 8. The characters are "a", "b" and "-" (blank). There are 5 time-steps. Let's apply our best path decoder to this matrix: the most likely character of t_0 is "a", the same applies for t_1 and t_2 . The blank character has the highest score at t_3 . Finally, "b" is most likely at t_4 . This gives us the path "aaa-b". We remove duplicate characters, this yields "a-b", and then we remove any blank from the remaining path, which gives us the text "ab" which we output as the recognized text. Best path decoding is, of course, only an approximation. It is easy to construct examples for which it gives the wrong result: if you decode the matrix from Fig. 3, you get "" as the recognized text. But we already know that the probability of "" is only 0.36 while it is 0.64 for "a". However, the approximation algorithm often gives good results in practical situations. There are more advanced decoders such as beam-search decoding, prefix-search decoding or token passing, which also use information about language structure to improve the results.

4.7 GOOGLE TRANSLATE API

After recognizing hand written text, our next step is to implement google translate which helps in translating the recognized hand written text into different language. After downloading google trans module we give recognized text as input to google trans. At the time of implementing two key words are important, which are src and des. Src is text source text language and des is language in which text needed to be translated. The translated output is stored in text file.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 RESULTS

Each person has different style of hand writings distinguishing from each other. So it is difficult to recognize the Handwritten characters. In recognising of Handwritten character Neural Networks plays important role. Numerous reports of character recognition are published but the accuracy is not good which makes it as an open problem that needs to be solved. In this paper attempt have been made to develop good accurate model for Handwriting recognition and translation. The system was trained using 1500 samples of handwritten data which contains lines, words, and characters. We pre-process our training data and divide them into batches. After pre-processing we send pre-processed data in batches to optical character recognition model for extracting features of the image and training the image data. Optical character recognition model contains CNN and RNN layers. CNN layers helps in feature extracting and RNN helps in training image data. We set a certain epoch number and train our model till every character is recognized. We use word beamsearch for recognising characters accurately. After recognising characters our next step is to translate the recognized text to another language. We use google translate API for translating the recognized text to another language. Observational result shows that the approach we used in this paper is producing good accuracy of 94% at recognition and translating text.



Fig 5.1 Input image

```

Anaconda Powershell Prompt (Anaconda3)
with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-03-14 11:05:32.802623: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1261] Device interconnect StreamExecutor
with strength 1 edge matrix:
2021-03-14 11:05:32.802855: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1267]
2021-03-14 11:05:32.805079: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not creating XLA devices, tf_xla_enable_xla_
devices not set
2021-03-14 11:05:35.219348: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimizatio
n passes are enabled (registered 2)
WARNING:tensorflow:AutoGraph could not transform <function Model.make_predict_function.<locals>.predict_function at 0x00
0001A69EB1C4C8> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VER
BOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
-----PREDICT-----
[Word model]: paappy - BiirtsAlay
[Word model with spell]: happy a BiirtsAlay
WARNING:tensorflow:AutoGraph could not transform <function Model.make_predict_function.<locals>.predict_function at 0x00
0001A69E30C558> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VER
BOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
[Line model]: Happey Birthday ! !
Happey Birthday ! !
Happey Birthday ! !
Entering into TextFile

```

Fig.5.2 Hand written recognition

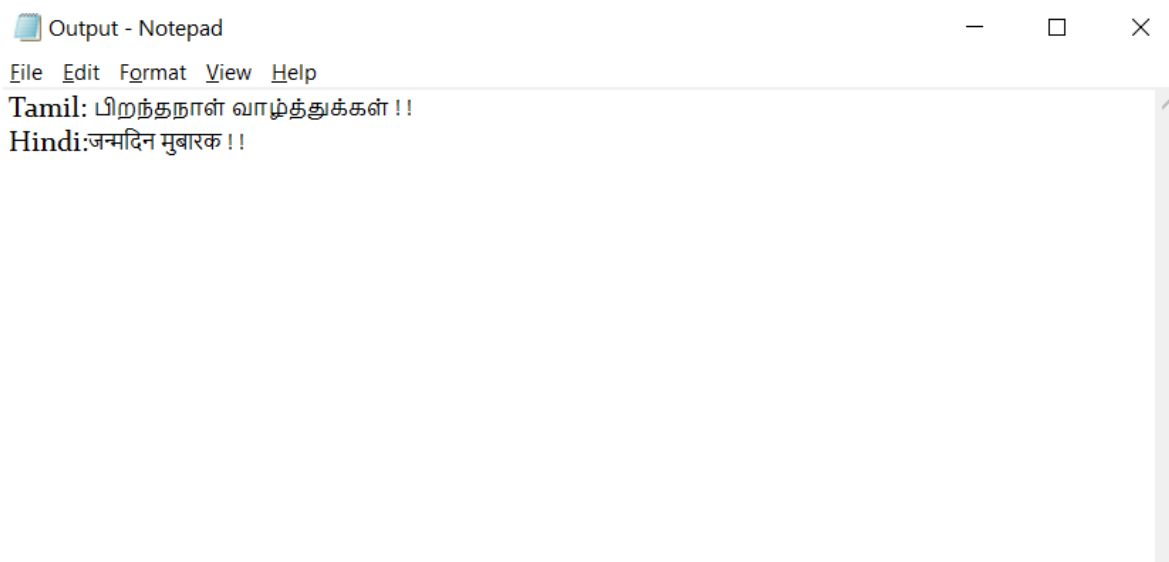


Fig.5.3 Translation

CHAPTER 6

SUMMARY AND CONCLUSION

6.1 CONCLUSION

The main purpose of this proposed system is to recognize handwritten words and translate to another language. The proposed work uses convolution neural networks, Recurrent neural networks, and Google translate API to recognize and translate handwritten words. Fig.1 describes the step by step procedure for the proposed system. In the first step images are taken as input. The image which is taken as input is pre-processed to improve image quality and helps to avoid image twisting. We implemented segmentation and image enhancement. Segmentation helps in dividing the lines as words and words as characters. After pre-processing our next step is to extract features. To extract features from the image we use convolution neural networks. We had implemented 5 layers in Convolution neural networks. Each layer consists of some set of operations. The first operation we included is filter kernel, which helps in edge detection, sharpening. This is done by doing a convolution between a kernel and an image. Then the non-linear RELU function is applied $R(z)=\max(0,z)$. The main use of this non-linearity of RELU is we can

build arbitrary shaped curves on the featured plane. After applying activation function pooling layer is used. In convolution nueral networks

we used pooling layer to reduce the size of the representation, which helps to speed up the computation process. The size of the image is downsized by summarizing input image into regions. For example we can reduce image of 4*4 size to 2*2 by applying pooling layer. By applying these operations to layers of my convolution neural network feature extraction of the image was done. After extracting features from the image we had implemented Recurrent neural networks. Recurrent neural networks helps to get sequential information from extracted features. The feature sequence contains features per time-step, so Recurrent neural networks generates applicable information through this sequence. The Long Short- Term Memory implementation of Recurrent neural networks is used. It helps us to generate sequence of relevant information with CTC loss and CTC decode. We get different CTC loss and decode at the time of training done repeatedly. We calculate CTC loss for every iteration of training. CTC loss is calculated between ground truth text and output matrix generated by Recurrent neural networks. If the loss is high then we train our model again with different epoch numbers. This process is repeated till CTC loss is low. Output matrix of Recurrent neural networks is given to CTC decode to decode into final text. After recognising handwritten data we need to translate recognized text to another language. The translation of recognized text is done by google translate API. After translating it displays the text.

6.2 FUTURE ENHANCEMENT

We can also increase the level of hand written recognition by using artificial neural networks, CRNN which is a combination of convolution neural networks and recurrent neural networks. We can try different activation functions like sigmoid, tanh, which helps in finding a way of which activation function works better. By using the above mentioned algorithms, we can get more accurate results.

REFERENCES

- [1] K. Gaurav and Bhatia P. K., "Analytical Review of Preprocessing Techniques for Offline Handwritten Character Recognition", 2nd International Conference on Emerging Trends in Engineering & Management, ICETEM, 2013.
- [2] Salvador España-Boquera, Maria J. C. B., Jorge G. M. and Francisco Z. M., "Improving Offline Handwritten Text Recognition with Hybrid HMM/ANN Models", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 33, No. 4, April 2011.
- [3] T.Som, Sumit Saha, "Handwritten Character Recognition Using Fuzzy Membership Function", International Journal of Emerging Technologies in Sciences and Engineering, Vol.5, No.2, pp. 11-15, Dec 2011
- [4] Renata F. P. Neves, Alberto N. G. Lopes Filho, Carlos A.B.Mello, CleberZanchettin, "ASVM Based Off-Line Handwritten Digit Recognizer", International conference on Systems, Man and Cybernetics, IEEE Xplore, pp. 510-515, 9-12 Oct, 2011, Brazil.
- [5] SG. Pirlo and D. Impedovo, "Fuzzy Zoning Based Classification for Handwritten Characters", IEEE Tranaction on pattern Recognition and Machine Intelligence, vol.19, no. 04, pp.780-785, August 2011.
- [6] Pranob K Charles, V.Harish, M.Swathi, CH. Deepthi, "A Review on the Various Techniques used for Optical Character Recognition", International Journal of Engineering Research and Applications, Vol. 2, Issue 1, pp. 659-662, Jan-Feb 2012.
- [7] Bhatia Neetu, "Optical Character Recognition Techniques", International Journal of Advanced Research in Computer Science and Software Engineering, Volume 4, Issue 5, May 2014.
- [8] H. El Abed and V. Märgner, "ICDAR 2009 – Arabic handwriting recognition competition", Inter. *Journal on Document Analysis and Recognition*, vol. 1433–2833, 2010.
- [9] A. Desai, "Gujarati handwritten numeral optical character reorga-nization through neural network,"Pattern Recognit., vol. 43, no. 7,pp. 2582–2589, Jul. 2010.
- [10] LeCun, Y. Bengio, and G. Hinton, "Deep learning,"Nature, vol. 521,no. 7553, pp. 436– 444, 2015.
- [11] . Kumar, S. R. Jindal, M. K. Jindal, and G. S. Lehal, "Improved recognition results of medieval handwritten Gurmukhi manuscripts using boosting and bagging methodologies,"Neural Process. Lett., vol. 50,pp. 43–56, Sep. 2018.

- [12] K. Pathan, A. A. Ali, and R. J. Ramteke, "Recognition of offlinehandwritten isolated Urdu character," *Adv. Comput. Res.*, vol. 4, no. 1, pp. 117–121, 2012.

APPENDIX

A. SOURCE CODE

```
import numpy as np
from Utils import *
from WordSegmentation import wordSegmentation, prepareImg
import matplotlib
matplotlib.use('TkAgg') # Use Aggo
import matplotlib.pyplot as plt
import numpy as np
import cv2
from Preprocessor import preprocess
from keras.models import model_from_json
import shutil
from keras import backend as K
from keras.utils import plot_model
from Spell import correction_list
from googletrans import Translator

translator = Translator()

if __name__ == '__main__':

    #l_model, l_model_predict = line_model()
    #with open('line_model_predict.json', 'w') as f:
    #    f.write(l_model_predict.to_json())
    with open('Resource/line_model_predict.json', 'r') as f:
        l_model_predict = model_from_json(f.read())
    with open('Resource/word_model_predict.json', 'r') as f:
        w_model_predict = model_from_json(f.read())
```

```
        #plot_model(l_model_predict, to_file='line_model.png', show_shapes=True,  
show_layer_names=True)
```

```

w_model_predict.load_weights('Resource/iam_words--20--1.425.h5'
)
l_model_predict.load_weights('Resource/iam_lines--12--17.373.h5')
test_img = 'Resource/test_img/5.png'

img = prepareImg(cv2.imread(test_img), 64)
img2 = img.copy()
res = wordSegmentation(img, kernelSize=25, sigma=11, theta=7, minArea=100)
if not os.path.exists('tmp'):
    os.mkdir('tmp')

for (j, w) in enumerate(res):
    (wordBox, wordImg) = w
    (x, y, w, h) = wordBox
    cv2.imwrite('tmp/%d.png'%j, wordImg)
    cv2.rectangle(img2,(x,y),(x+w,y+h),(0,255,0),1) # draw bounding box
    in
summary image

cv2.imwrite('Resource/summary.png', img2)
plt.imshow(img2)
imgFiles = os.listdir('tmp')
imgFiles = sorted(imgFiles)
pred_line = []
for f in imgFiles:
    pred_line.append(predict_image(w_model_predict, 'tmp/'+f, True))
print('_____PREDICT_____')
print('[Word model]: '+' '.join(pred_line))
pred_line = correction_list(pred_line)
print('[Word model with spell]: '+' '.join(pred_line))
print('[Line model]: ' + predict_image(l_model_predict, test_img, False))
k = predict_image(l_model_predict, test_img, False)
result = translator.translate(k, src='en', dest='ta')

```

```
result2 = translator.translate(k, src='en', dest='hi')  
print(result.text)  
print(result2.text)
```

```
text_file = open("Output.txt", "w", encoding="utf-8")
print("Entering into TextFile") text_file.write("Tamil:
%s\n" % result.text) text_file.write("Hindi:%s" %
result2.text) text_file.close()
plt.show()
shutil.rmtree('tmp')
```

B. SCREEN SHOTS

```
Anaconda Powershell Prompt (Anaconda3)
(base) PS C:\Users\dhanu> cd C:\Users\dhanu\OneDrive\Handwritten-Text-Recognition-master
(base) PS C:\Users\dhanu\OneDrive\Handwritten-Text-Recognition-master> python Prediction.py
```



```
Anaconda Powershell Prompt (Anaconda3)

with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2021-03-14 11:05:32.802623: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1261] Device interconnect StreamExecutor
with strength 1 edge matrix:
2021-03-14 11:05:32.802855: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1267]
2021-03-14 11:05:32.805079: I tensorflow/compiler/jit/xla_gpu_device.cc:99] Not creating XLA devices, tf_xla_enable_xla_
devices not set
2021-03-14 11:05:35.219348: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR optimizatio
n passes are enabled (registered 2)
WARNING:tensorflow:AutoGraph could not transform <function Model.make_predict_function.<locals>.predict_function at 0x00
0001A69EB1C4C8> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VER
BOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
-----PREDICT-----
[Word model]: paappy - BiirtsAlay
[Word model with spell]: happy a BiirtsAlay
WARNING:tensorflow:AutoGraph could not transform <function Model.make_predict_function.<locals>.predict_function at 0x00
0001A69E30C558> and will run it as-is.
Please report this to the TensorFlow team. When filing the bug, set the verbosity to 10 (on Linux, `export AUTOGRAPH_VER
BOSITY=10`) and attach the full output.
Cause: 'arguments' object has no attribute 'posonlyargs'
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
[Line model]: Happey Birthday ! !
Happey Birthday ! !
Happey Birthday ! !
Entering into TextFile
```

```
Output - Notepad
File Edit Format View Help
Tamil: பிறந்தநாள் வாழ்த்துக்கள் !!
Hindi:जन्मदिन मुबारक !!
```