

## List of Figures

Figure No.	Title	Page No.
1.	Regression Model	5
2.	Flow Chart	6
3.	UML	7
4.	Architecture Diagram	8
5.	Block Diagram	10
6.	Activity Diagram	36
7.	Screen Shots	35-37

# CONTENTS

Topic	Page No.
<i>Acknowledgment</i>	<i>i</i>
<i>Candidate's Declaration</i>	<i>ii</i>
<i>Abstract</i>	<i>iii</i>
<b>CHAPTER 1 Introduction</b>	
1.1 Purpose	
1.2 Scope	
1.3 Technologies Used	
<b>CHAPTER 2 SRS</b>	
2.1 Goals of proposed system	
2.2 Block Diagram	
2.3 Feasibility Study	
2.4 Project Requirements	
2.5 User Characteristics	
2.6 Constraints	
<b>CHAPTER 3 Architecture Diagram</b>	
<b>CHAPTER 4 Project methodology</b>	
<b>CHAPTER 5 Screen shots</b>	
<b>CHAPTER 6 Conclusion and Future Scope</b>	
<b>References</b>	

# CHAPTER 1

## INTRODUCTION

### 1.1) Purpose

Handwritten digit recognition and pattern analysis is one of the active research topics in digital image processing. The technology is leaping into so much advancement that image recognition will become part and parcel of our daily lives. Handwritten Devanagari characters are quite complex for recognition due to presence of header line, conjunct characters and similarity in shapes of multiple characters. The main purpose of this paper is to introduce a method for recognition of handwritten Devanagari characters using segmentation and neural networks. The whole process of recognition works in stages as preprocessing on document image, segmentation of document into lines, line into words and word into characters, finally recognition using feed forward neural network. Important steps in any preprocessing, segmentation, feature extraction and recognition using neural network. .

### 1.2) Scope

This application developed as an in house effort to develop an evaluation system which can be used for grading exams, teacher evaluation and performance assessment of employees. For future work we plan to use the same technique to identify signatures for processing cheques in banking Industry and secondly to develop a recognition system for Education purpose and furthure researchs.

### 1.3) Technologies to be used

This project will be a desktop application to be developed in PYTHON 3.6.5 having numpy,open cv as imported files.

- Desktop Application
- Form Design -PYTHON 3.6.5
- Anaconda(spyder)
- Testing (Anaconda cloud)

## **CHAPTER 2**

### **SOFTWARE REQUIREMENT SPECIFICATIONS**

#### **Feasibility Study**

Depending on the results of the initial investigation the survey is now expanded to a more detailed feasibility study. “**FEASIBILITY STUDY**” is a test of system proposal according to its workability, impact of the organization, ability to meet needs and effective use of the resources. It focuses on these major questions:

1. What are the user’s demonstrable needs and how does a candidate system meet them?
2. What resources are available for given candidate system?
3. What are the likely impacts of the candidate system on the organization?
4. Whether it is worth to solve the problem?

During feasibility analysis for this project, following primary areas of interest are to be considered. Investigation and generating ideas about a new system does this.

#### **Steps in feasibility analysis:-**

Eight steps involved in the feasibility analysis are:

- Form a project team and appoint a project leader.
- Prepare system flowcharts.
- Enumerate potential proposed system.
- Define and identify characteristics of proposed system.
- Determine and evaluate performance and cost effective of each proposed system.
- Weight system performance and cost data.

- Select the best-proposed system.
- Prepare and report final project directive to management.

### **Technical feasibility**

A study of resource availability that may affect the ability to achieve an acceptable system. This evaluation determines whether the technology needed for the proposed system is available or not.

- Can the work for the project be done with current equipment existing software technology & available personal?
- Can the system be upgraded if developed?
- If new technology is needed then what can be developed?

## **Software and Hardware Requirements**

### **Project Requirements**

#### 1. Hardware Requirements:-

Processor: Pentium II, Pentium III, Pentium IV or higher

RAM :- 4GB or higher

#### 2. Software Requirements:-

GUI :- Anaconda (Jupyter cloud)

Database :- SQLITE 3

#### 3. Python Implementation:-

1. Dataset- MNIST dataset
2. Images of size 28 X 28
3. Classify digits from 0 to 9
4. Logistic Regression, Shallow Network and Deep Network Support added.

## **User Characteristics**

Every user should be:

- Comfortable of working with computer.
- He must also have basic knowledge of English too.

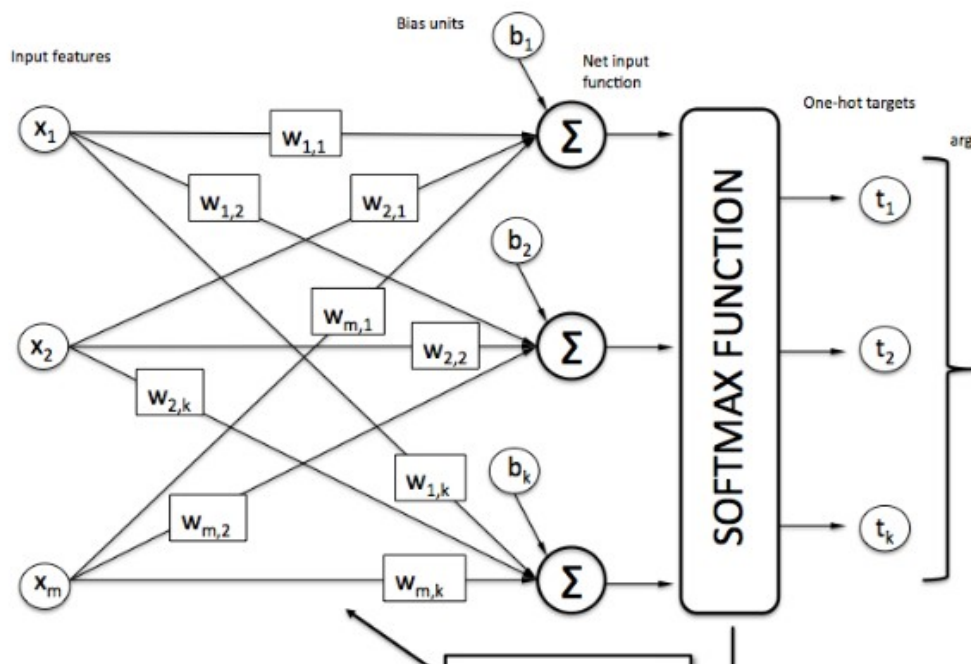
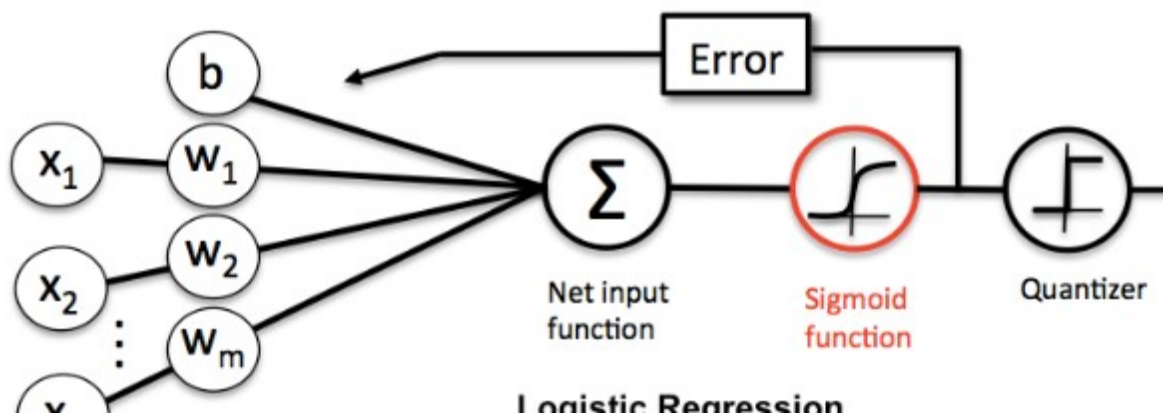
## **Constraints**

- GUI is only in English.

# Chapter 3

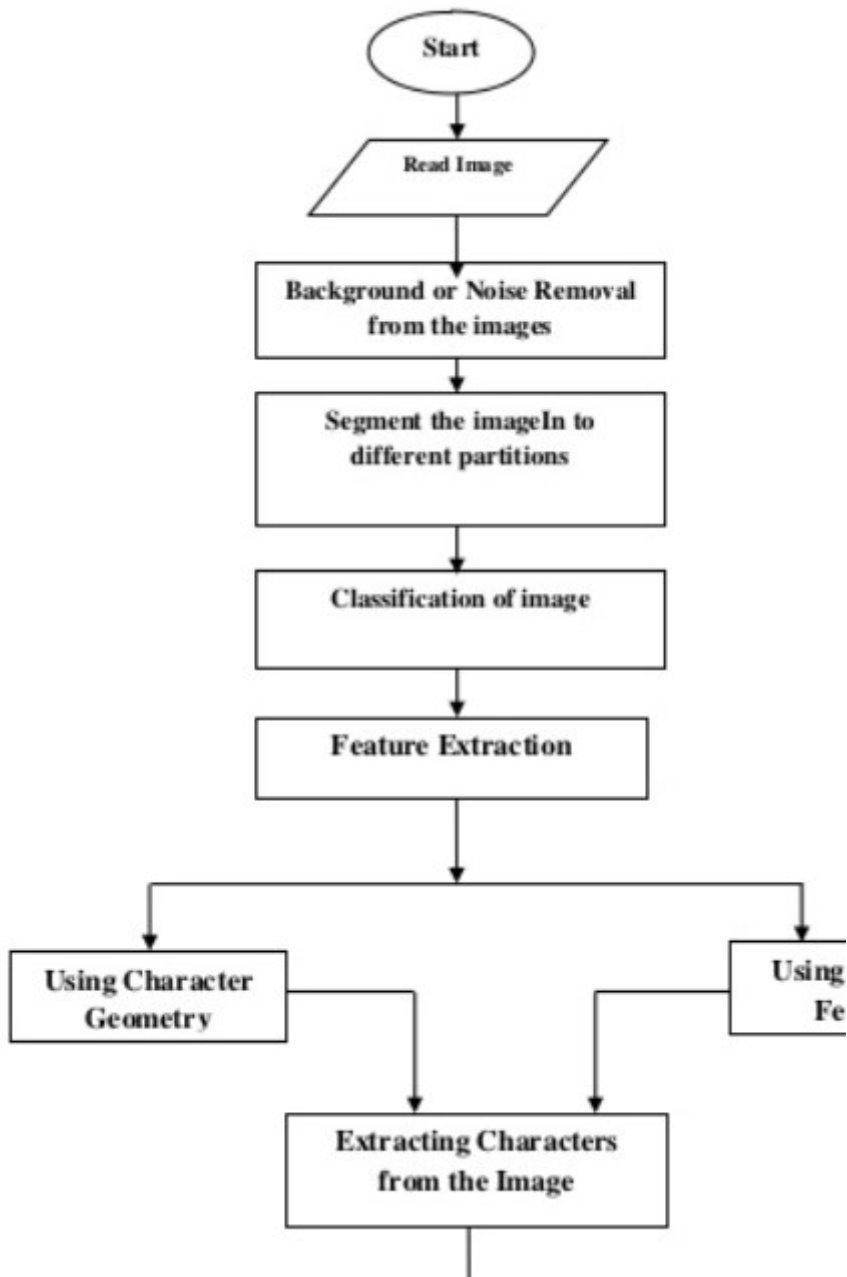
## ARCHITECTURE DIAGRAM

SoftMax Regression (synonyms: Multinomial Logistic, Maximum Entropy Classifier, or just Multi-class Logistic Regression) is a generalization of logistic regression that we can use for multi-class classification (under the assumption that the classes are mutually exclusive). In contrast, we use the (standard) Logistic Regression model in binary classification tasks.



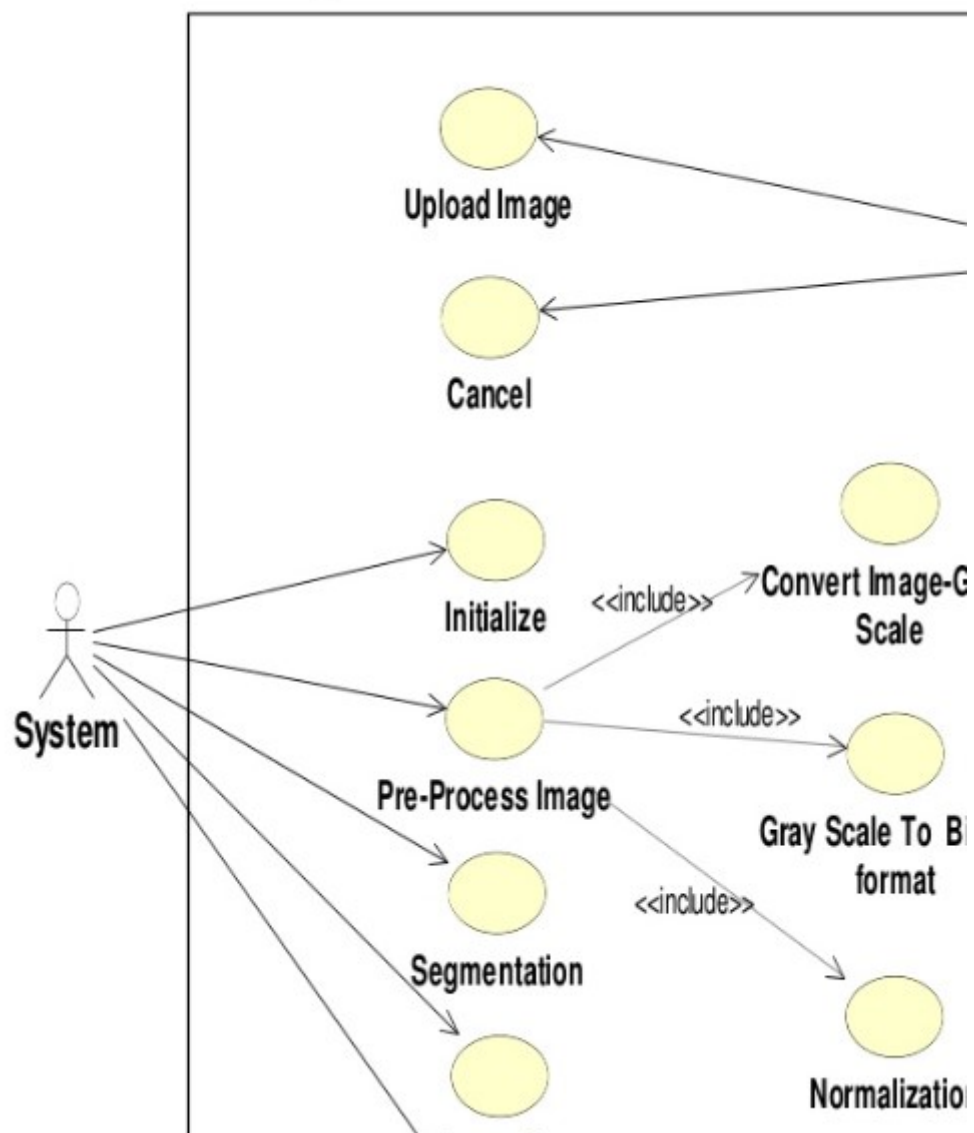
## FLOW CHART :

Flowchart of the process

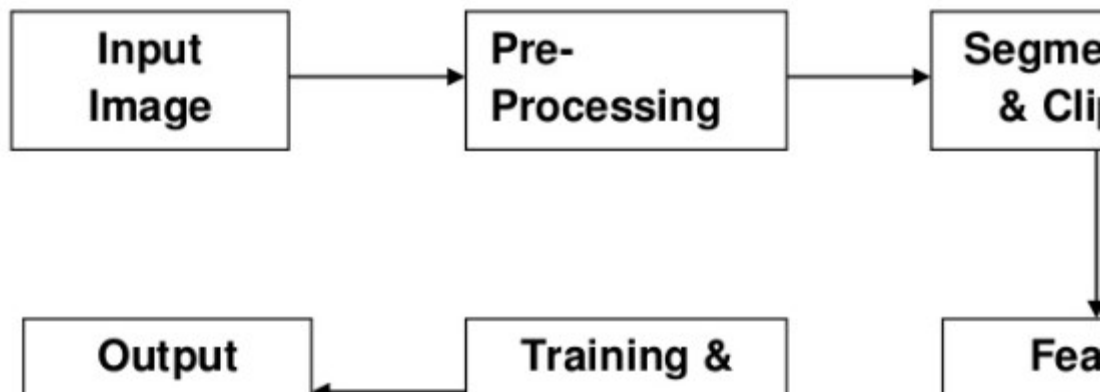




## UML DIAGRAM:



## Architecture of the system



## Chapter 4

# PROJECT METHODOLOGY

Devanagari script has features different from other languages. Devanagari character set has 13 vowels, 36 consonants and 10 numerals with optional modifier symbols. Characters are organized into three zones as upper, middle and lower zone. Core characters are positioned in middle zone, while optional modifiers in upper and lower zones. Two characters may be connected to each other. In Devanagari script, the concept of uppercase and lowercase characters is absent. Fig. 2 represents Devanagari character set. It represents Devanagari character modifier set. Modifiers are optional symbols arranged in upper and lower zones.

### 4.1) Methodology

**Image acquisition:** We will acquire an image to our system as an input .this image should have a specific format, for example, bmp format and with a determined size such as 30×20 pixels. This image can be acquired through the scanner or, digital camera or other digital input devices.

**.Preprocessing:** After acquiring the image, it will be processed through sequence of preprocessing steps to be ready for the next step.

**Noise removal:** reducing noise in an image. For on-line there is no noise to eliminate so no need for the noise removal. In off-line mode, the noise may come from the writing style or from the optical device captures the image[10] .

**Normalization-scaling:** standardize the font size within the image. This problem appears clearly in handwritten text, because the font size is not restricted when using handwriting.

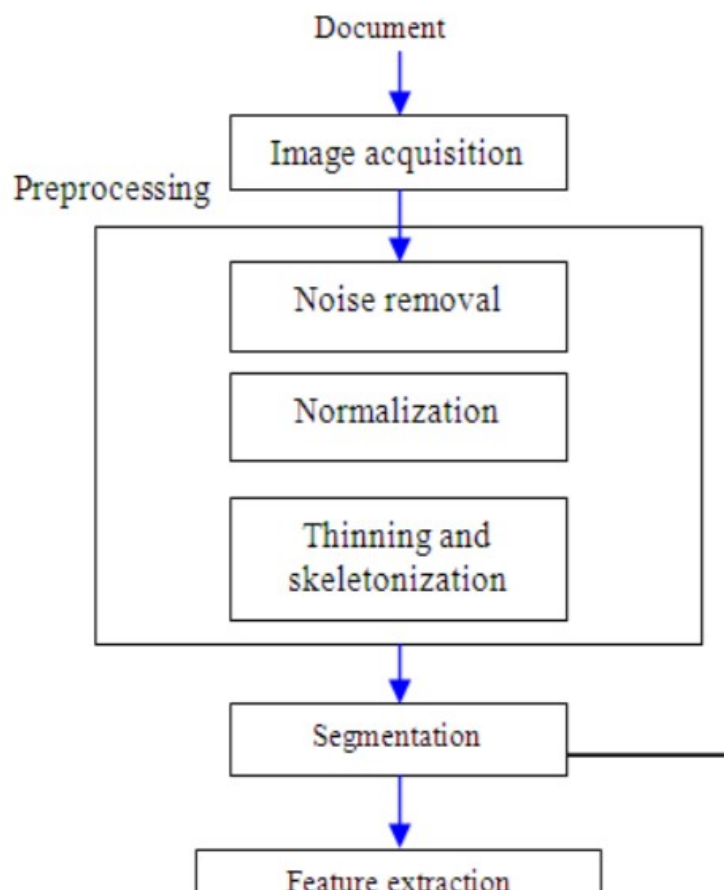
**Thinning and skeletonization:** Representing the shape of the object in a relatively smaller number of pixels[9] . Thinning algorithms can be parallel or sequential. Parallel is applied on all

pixels simultaneously. Sequential examine pixels and transform them depending on the preceding processed results.

**Segmentation:** Since the data are isolated, no need for segmentation. With regards to the isolated digits, to apply vertical segmentation on the image containing more than digit will isolate each digit alone.

**Normalization scaling and translation:** Handwriting produces variability in size of written digits. This leads to the need of scaling the digits size within the image to a standard size, as this may lead to better recognition accuracy

## 4.2 Block diagram

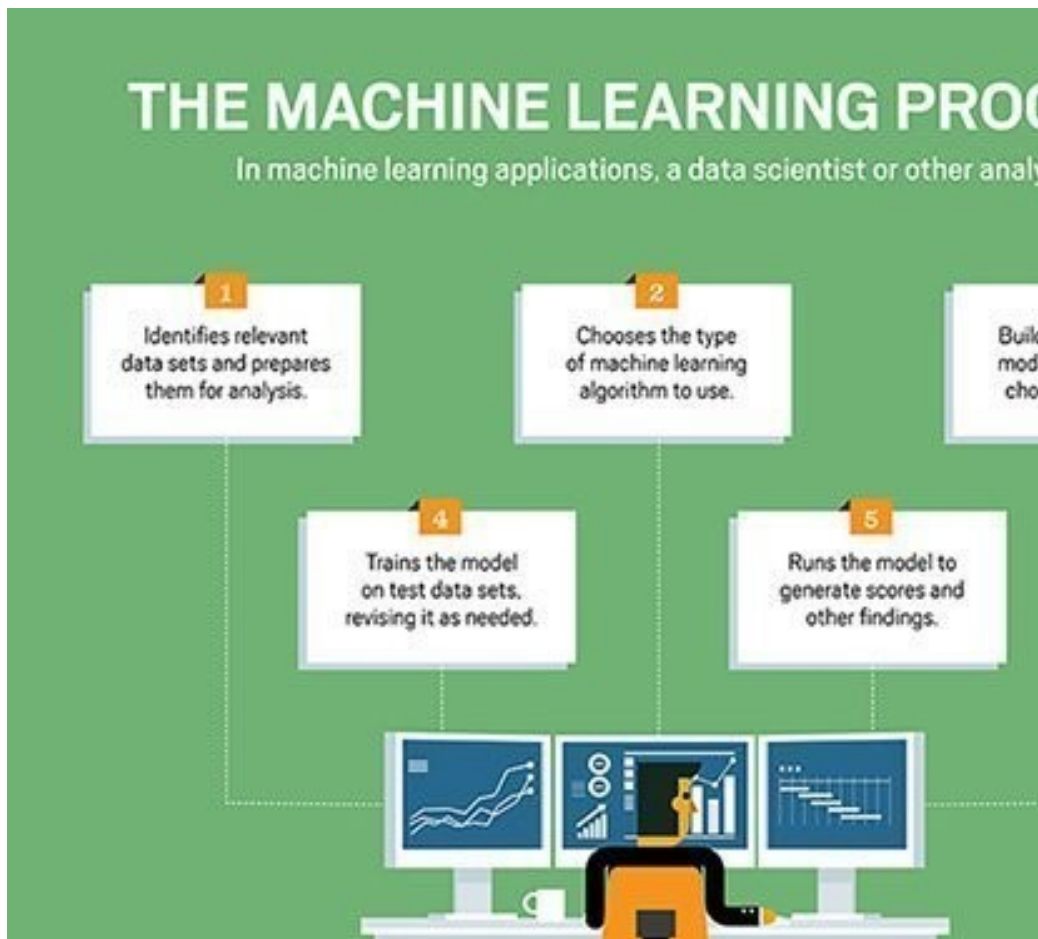


### 4.3 Deep Learning ( deep neural network)

Deep learning is an aspect of artificial intelligence (AI) that is concerned with emulating the learning approach that human beings use to gain certain types of knowledge. At its simplest, deep learning can be thought of as a way to automate predictive analytics.

#### How deep learning works

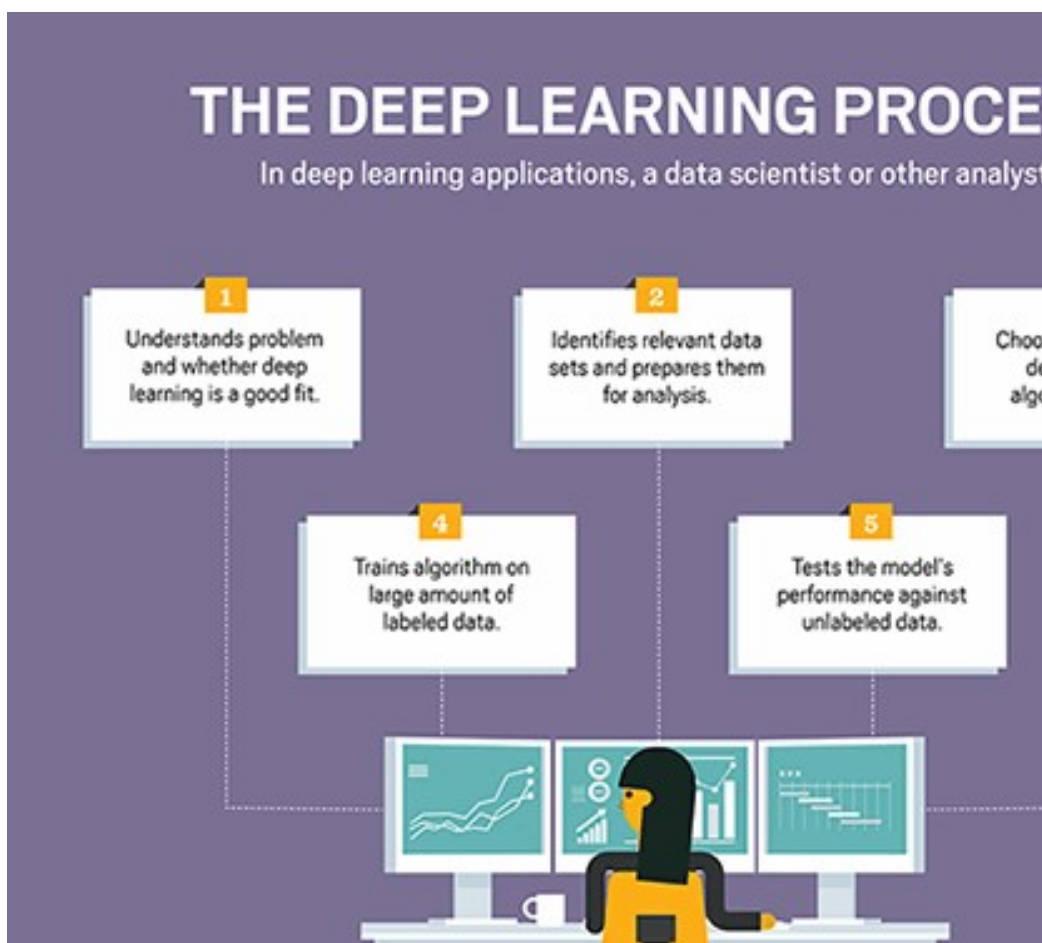
Computer programs that use deep learning go through much the same process. Each algorithm in the hierarchy applies a nonlinear transformation on its input and uses what it learns to create a statistical model as output. Iterations continue until the output has reached an acceptable level of accuracy. The number of processing layers through which data must pass is what inspired the label *deep*.



In traditional machine learning, the learning process is supervised and the programmer has to be very, very specific when telling the computer what types of things it should be looking for when

deciding if an image contains a dog or does not contain a dog. This is a laborious process called *feature extraction* and the computer's success rate depends entirely upon the programmer's ability to accurately define a feature set for "dog." The advantage of deep learning is that the program builds the feature set by itself without supervision. Unsupervised learning is not only faster, but it is usually more accurate.

Initially, the computer program might be provided with training data, a set of images for which a human has labeled each image "dog" or "not dog" with meta tags. The program uses the information it receives from the training data to create a feature set for dog and build a predictive model. In this case, the model the computer first creates might predict that anything in an image that has four legs and a tail should be labeled "dog." Of course, the program is not aware of the labels "four legs" or "tail;" it will simply look for patterns of pixels in the digital data. With each iteration, the predictive model the computer creates becomes more complex and more accurate.



Because this process mimics a system of human neurons, deep learning is sometimes referred to as deep neural learning or deep neural networking. Unlike the toddler, who will take weeks or even months to understand the concept of "dog," a computer program that uses deep learning algorithms can be shown a training set and sort through millions of images, accurately identifying which images have dogs in them within a few minutes.

To achieve an acceptable level of accuracy, deep learning programs require access to immense amounts of training data and processing power, neither of which were easily available to programmers until the era of big data and cloud computing. Because deep learning programming is able to create complex statistical models directly from its own iterative output, it is able to create accurate predictive models from large quantities of unlabeled, unstructured data. This is important as the internet of things (IoT) continues to become more pervasive, because most of the data humans and machines create is unstructured and is not labeled.

Use cases today for deep learning include all types of big data analytics applications, especially those focused on natural language processing (NLP), language translation, medical diagnosis, stock market trading signals, network security and image identification.

## **Using neural networks**

A type of advanced machine learning algorithm, known as neural networks, underpins most deep learning models. Neural networks come in several different forms, including recurrent neural networks, convolutional neural networks, artificial neural networks and feedforward neural networks, and each has their benefit for specific use cases. However, they all function in somewhat similar ways, by feeding data in and letting the model figure out for itself whether it has made the right interpretation or decision about a given data element.

Neural networks involve a trial-and-error process, so they need massive amounts of data to train on. It's no coincidence that neural networks became popular only after most enterprises embraced big data analytics and accumulated large stores of data. Because the model's first few iterations involve somewhat-educated guesses on the contents of image or parts of speech, the data used during the training stage must be labeled so the model can see if its guess was accurate. This means that, though many enterprises that use big data have large amounts of data,

unstructured data is less helpful. Unstructured data can be analyzed by a deep learning model once it has been trained and reaches an acceptable level of accuracy, but deep learning models can't train on unstructured data.

## 4.4 Convolutional Neural Networks (CNNs / ConvNets)

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what changes? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

### Architecture Overview

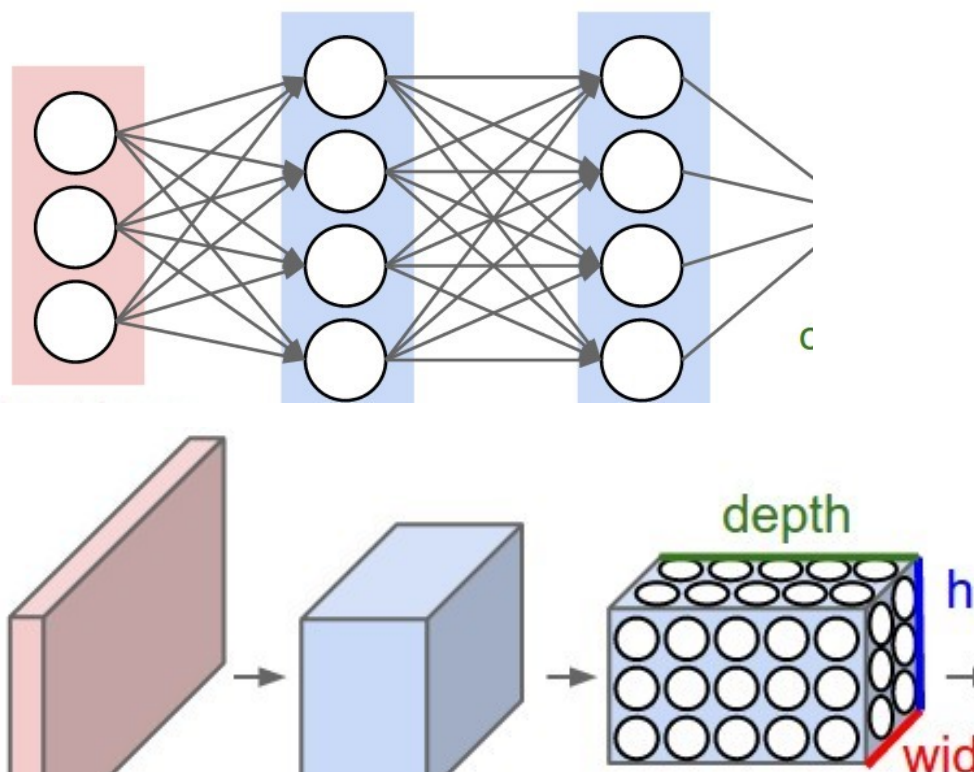
*Recall: Regular Neural Nets.* As we saw in the previous chapter, Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores.

*Regular Neural Nets don't scale well to full images.* In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have  $32 \times 32 \times 3 = 3072$  weights. This amount



still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g.  $200 \times 200 \times 3$ , would lead to neurons that have  $200 \times 200 \times 3 = 120,000$  weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

*3D volumes of neurons.* Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**. (Note that the word *depth* here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions  $32 \times 32 \times 3$  (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions  $1 \times 1 \times 10$ , because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

## Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

*Example Architecture: Overview.* We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the  $\max(0, x)$  thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].

- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size  $[1 \times 1 \times 10]$ , where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

#### **In summary:**

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)

#### **Spatial arrangement.**

We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven't yet discussed how many neurons there are in the output volume or how they are arranged. Three hyperparameters control the size of the output volume: the **depth**, **stride** and **zero-padding**. We discuss these next:

1. First, the **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
2. Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
3. As we will soon see, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we'll see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

We can compute the spatial size of the output volume as a function of the input volume size (WW), the receptive field size of the Conv Layer neurons (FF), the stride with which they are applied (SS), and the amount of zero padding used (PP) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by  $(W-F+2P)/S+1(W-F+2P)/S+1$ . For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output. Lets also see one more graphical example:

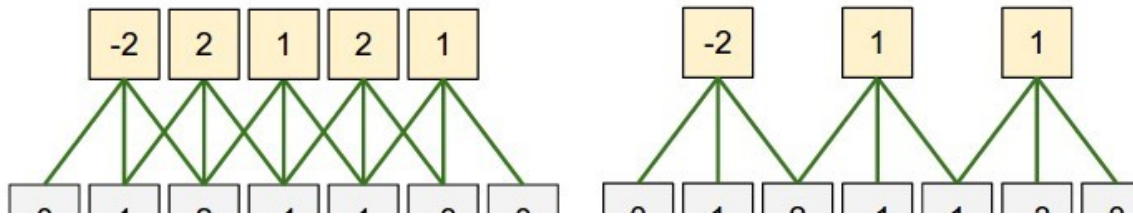


Illustration of spatial arrangement. In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of  $F = 3$ , the input size is  $W = 5$ , and there is zero padding of  $P = 1$ . **Left:** The neuron strided across the input in stride of  $S = 1$ , giving output of size  $(5 - 3 + 2)/1 + 1 = 5$ . **Right:** The neuron uses stride of  $S = 2$ , giving output of size  $(5 - 3 + 2)/2 + 1 = 3$ . Notice that stride  $S = 3$  could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since  $(5 - 3 + 2) = 4$  is not divisible by 3. The neuron weights are in this example  $[1, 0, -1]$  (shown on very right), and its bias is zero. These weights are shared across all yellow neurons (see parameter sharing below).

*Use of zero-padding.* In the example above on left, note that the input dimension was 5 and the output dimension was equal: also 5. This worked out so because our receptive fields were 3 and we used zero padding of 1. If there was no zero-padding used, then the output volume would have had spatial dimension of only 3, because that is how many neurons would have “fit” across the original input. In general, setting zero padding to be  $P = (F - 1)/2$  when the stride is  $S = 1$  ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way and we will discuss the full reasons when we talk more about ConvNet architectures.

*Constraints on strides.* Note again that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size  $W = 10$ , no zero-padding is used  $P = 0$ , and the filter size is  $F = 3$ , then it would be impossible to use stride  $S = 2$ , since  $(W - F + 2P)/S + 1 = (10 - 3 + 0)/2 + 1 = 4.5$ , i.e. not an integer, indicating that the neurons don't “fit” neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a ConvNet library could throw an

exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. As we will see in the ConvNet architectures section, sizing the ConvNets appropriately so that all the dimensions “work out” can be a real headache, which the use of zero-padding and some design guidelines will significantly alleviate.

*Real-world example.* The [Krizhevsky et al.](#) architecture that won the ImageNet challenge in 2012 accepted images of size  $[227 \times 227 \times 3]$ . On the first Convolutional Layer, it used neurons with receptive field size  $F=11$ , stride  $S=4$  and no zero padding  $P=0$ . Since  $(227 - 11)/4 + 1 = 55$ , and since the Conv layer had a depth of  $K=96$ , the Conv layer output volume had size  $[55 \times 55 \times 96]$ . Each of the  $55 \times 55 \times 96$  neurons in this volume was connected to a region of size  $[11 \times 11 \times 3]$  in the input volume. Moreover, all 96 neurons in each depth column are connected to the same  $[11 \times 11 \times 3]$  region of the input, but of course with different weights. As a fun aside, if you read the actual paper it claims that the input images were  $224 \times 224$ , which is surely incorrect because  $(224 - 11)/4 + 1$  is quite clearly not an integer. This has confused many people in the history of ConvNets and little is known about what happened. My own best guess is that Alex used zero-padding of 3 extra pixels that he does not mention in the paper.

**Parameter Sharing.** Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. Using the real-world example above, we see that there are  $55 \times 55 \times 96 = 290,400$  neurons in the first Conv Layer, and each has  $11 \times 11 \times 3 = 363$  weights and 1 bias. Together, this adds up to  $290400 * 364 = 105,705,600$  parameters on the first layer of the ConvNet alone. Clearly, this number is very high.

It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position  $(x,y)$ , then it should also be useful to compute at a different position  $(x_2,y_2)$ . In other words, denoting a single 2-dimensional slice of depth as a **depth slice** (e.g. a volume of size  $[55 \times 55 \times 96]$  has 96 depth slices, each of size  $[55 \times 55]$ ), we are going to constrain the neurons in each depth slice to use the same weights and bias. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of  $96 \times 11 \times 11 \times 3 = 34,848$  unique weights, or 34,944 parameters (+96 biases). Alternatively, all  $55 \times 55$  neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but

these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a **convolution** of the neuron's weights with the input volume (Hence the name: Convolutional Layer). This is why it is common to refer to the sets of weights as a **filter** (or a **kernel**), that is convolved with the input.



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size  $[11 \times 11 \times 3]$ , and each one is shared by the  $55 \times 55$  neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the  $55 \times 55$  distinct locations in the Conv layer output volume.

Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than another. One practical example is when the input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features

could (and should) be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a **Locally-Connected Layer**

**Numpy examples.** To make the discussion above more concrete, let's express the same ideas but in code and with a specific example. Suppose that the input volume is a numpy array `X`. Then:

- A *depth column* (or a *fibre*) at position `(x,y)` would be the activations `X[x,y,:]`.
- A *depth slice*, or equivalently an *activation map* at depth `d` would be the activations `X[:, :, d]`.

*Conv Layer Example.* Suppose that the input volume `X` has shape `X.shape: (11,11,4)`. Suppose further that we use no zero padding ( $P=0$ ), that the filter size is  $F=5$ , and that the stride is  $S=2$ . The output volume would therefore have spatial size  $(11-5)/2+1 = 4$ , giving a volume with width and height of 4. The activation map in the output volume (call it `V`), would then look as follows (only some of the elements are computed in this example):

- `V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0`
- `V[1,0,0] = np.sum(X[2:7,:5,:] * W0) + b0`
- `V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0`
- `V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0`

Remember that in numpy, the operation `*` above denotes elementwise multiplication between the arrays. Notice also that the weight vector `W0` is the weight vector of that neuron and `b0` is the bias. Here, `W0` is assumed to be of shape `W0.shape: (5,5,4)`, since the filter size is 5 and the depth of the input volume is 4. Notice that at each point, we are computing the dot product as



seen before in ordinary neural networks. Also, we see that we are using the same weight and bias (due to parameter sharing), and where the dimensions along the width are increasing in steps of 2 (i.e. the stride). To construct a second activation map in the output volume, we would have:

- $V[0,0,1] = \text{np.sum}(X[:5,:5,:] * W1) + b1$
- $V[1,0,1] = \text{np.sum}(X[2:7,:5,:] * W1) + b1$
- $V[2,0,1] = \text{np.sum}(X[4:9,:5,:] * W1) + b1$
- $V[3,0,1] = \text{np.sum}(X[6:11,:5,:] * W1) + b1$
- $V[0,1,1] = \text{np.sum}(X[:5,2:7,:] * W1) + b1$  (example of going along y)
- $V[2,3,1] = \text{np.sum}(X[4:9,6:11,:] * W1) + b1$  (or along both)

where we see that we are indexing into the second depth dimension in  $V$  (at index 1) because we are computing the second activation map, and that a different set of parameters ( $W1$ ) is now used. In the example above, we are for brevity leaving out some of the other operations the Conv Layer would perform to fill the other parts of the output array  $V$ . Additionally, recall that these activation maps are often followed elementwise through an activation function such as ReLU, but this is not shown here.

**Summary.** To summarize, the Conv Layer:

- Accepts a volume of size  $W1 \times H1 \times D1$
- Requires four hyperparameters:
  - Number of filters  $K$ ,
  - their spatial extent  $FF$ ,
  - the stride  $SS$ ,
  - the amount of zero padding  $PP$ .
- Produces a volume of size  $W2 \times H2 \times D2$  where:

- $W_2 = (W_1 - F + 2P) / S + 1$   $W_2 = (W_1 - F + 2P) / S + 1$
- $H_2 = (H_1 - F + 2P) / S + 1$   $H_2 = (H_1 - F + 2P) / S + 1$  (i.e. width and height are computed equally by symmetry)
- $D_2 = K$   $D_2 = K$
- With parameter sharing, it introduces  $F \cdot F \cdot D_1$  weights per filter, for a total of  $(F \cdot F \cdot D_1) \cdot K$  weights and  $K$  biases.
- In the output volume, the  $dd$ -th depth slice (of size  $W_2 \times H_2$ ) is the result of performing a valid convolution of the  $dd$ -th filter over the input volume with a stride of  $S$ , and then offset by  $dd$ -th bias.

A common setting of the hyperparameters is  $F=3, S=1, P=1$ . However, there are common conventions and rules of thumb that motivate these hyperparameters.

### **Backpropagation.**

The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters). This is easy to derive in the 1-dimensional case with a toy example (not expanded on for now).

### **1x1 convolution.**

As an aside, several papers use 1x1 convolutions, as first investigated by [Network in Network](#). Some people are at first confused to see 1x1 convolutions especially when they come from signal processing background. Normally signals are 2-dimensional so 1x1 convolutions do not make sense (it's just pointwise scaling). However, in ConvNets this is not the case because one must remember that we operate over 3-dimensional volumes, and that the filters always extend through the full depth of the input volume. For example, if the input is  $[32 \times 32 \times 3]$  then doing 1x1 convolutions would effectively be doing 3-dimensional dot products (since the input depth is 3 channels).

### **Dilated convolutions.**

A recent development (e.g. see [paper by Fisher Yu and Vladlen Koltun](#)) is to introduce one more hyperparameter to the CONV layer called the *dilation*. So far we've only discussed CONV filters that are contiguous. However, it's possible to have filters that have spaces between each cell, called dilation. As an example, in one dimension a filter  $w$  of size 3 would compute over input  $x$  the following:  $w[0]*x[0] + w[1]*x[1] + w[2]*x[2]$ . This is dilation of 0. For dilation 1 the filter would instead compute  $w[0]*x[0] + w[1]*x[2] + w[2]*x[4]$ ; In other words there is a gap of 1 between the applications. This can be very useful in some settings to use in conjunction with 0-dilated filters because it allows you to merge spatial information across the inputs much more aggressively with fewer layers. For example, if you stack two 3x3 CONV layers on top of each other then you can convince yourself that the neurons on the 2nd layer are a function of a 5x5 patch of the input (we would say that the *effective receptive field* of these neurons is 5x5). If we use dilated convolutions then this effective receptive field would grow much quicker.

### **Pooling Layer :**

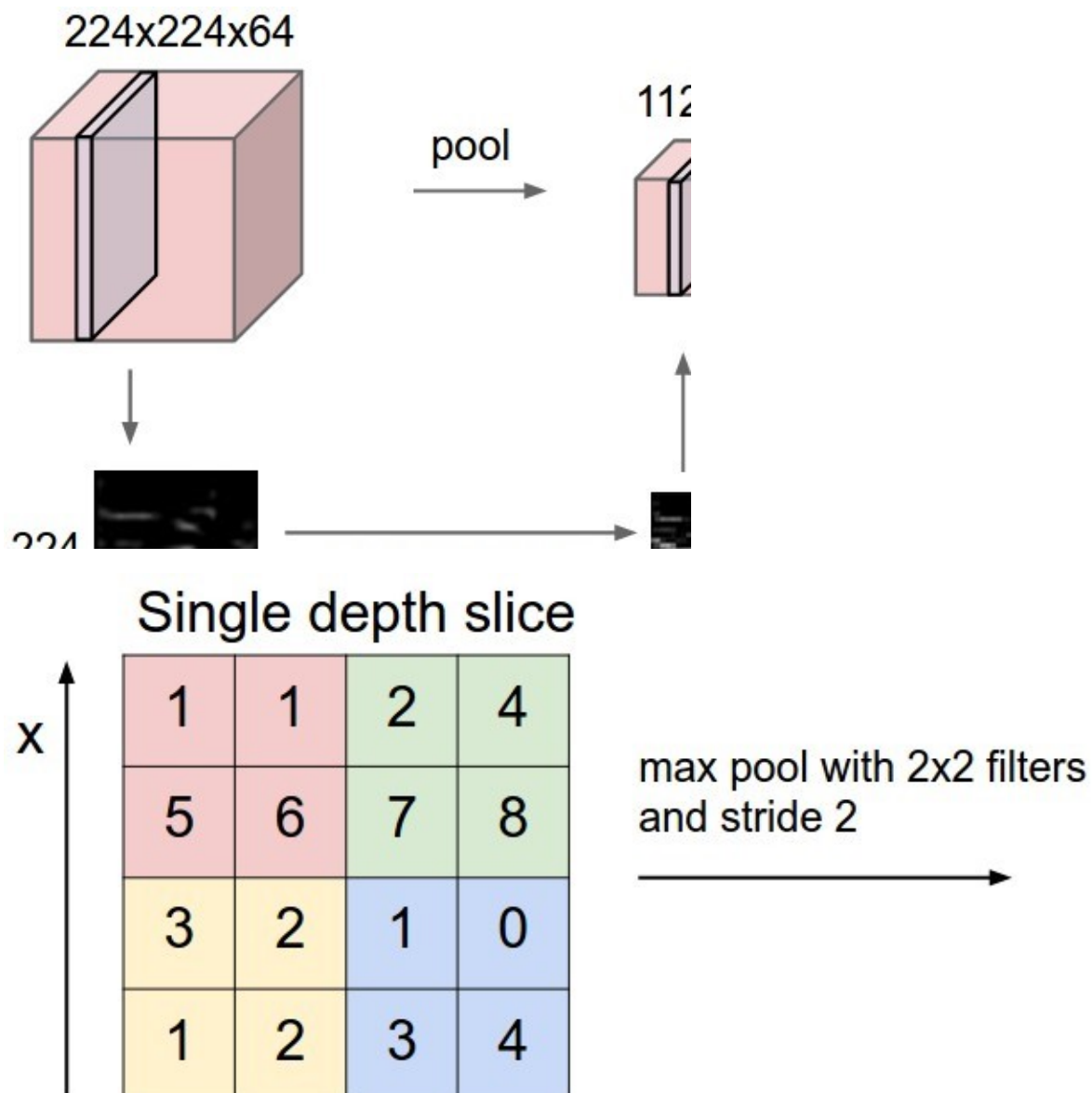
It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

- Accepts a volume of size  $W1 \times H1 \times D1$

- Requires two hyperparameters:
  - their spatial extent  $FF$ ,
  - the stride  $SS$ ,
- Produces a volume of size  $W_2 \times H_2 \times D_2$  where:
  - $W_2 = (W_1 - F) / S + 1$
  - $H_2 = (H_1 - F) / S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with  $F=3, S=2$  (also called overlapping pooling), and more commonly  $F=2, S=2$ . Pooling sizes with larger receptive fields are too destructive.

**General pooling.** In addition to max pooling, the pooling units can also perform other functions, such as *average pooling* or even *L2-norm pooling*. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size  $[224 \times 224 \times 64]$  is pooled with filter size 2, stride 2 into output volume of size  $[112 \times 112 \times 64]$ . Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little  $2 \times 2$  square).

**Backpropagation.** Recall from the backpropagation chapter that the backward pass for a  $\max(x, y)$  operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is

common to keep track of the index of the max activation (sometimes also called *the switches*) so that gradient routing is efficient during backpropagation.

**Getting rid of pooling.** Many people dislike the pooling operation and think that we can get away without it. For example, Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

### *Normalization Layer*

Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any. For various types of normalizations, see the discussion in Alex Krizhevsky's cuda-convnet library API.

### *Fully-connected layer*

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See the *Neural Network* section of the notes for more information.

### *Converting FC layers to CONV layers*

It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical. Therefore, it turns out that it's possible to convert between FC and CONV layers:

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
- Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with  $K=4096$  that is looking at some input volume of size  $7 \times 7 \times 512 \times 7 \times 512$  can be equivalently expressed as a CONV layer with  $F=7, P=0, S=1, K=4096$ . In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be  $1 \times 1 \times 4096 \times 1 \times 4096$  since only a single depth column “fits” across the input volume, giving identical result as the initial FC layer.

## 4.5 TensorFlow :

TensorFlow is a Python-friendly open source library for numerical computation that makes machine learning faster and easier

Machine learning is a complex discipline. But implementing machine learning models is far less daunting and difficult than it used to be, thanks to machine learning frameworks—such as Google’s TensorFlow—that ease the process of acquiring data, training models, serving predictions, and refining future results.

### *How TensorFlow works*

TensorFlow allows developers to create *dataflow graphs*—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or *tensor*.

TensorFlow provides all of this for the programmer by way of the Python language. Python is easy to learn and work with, and provides convenient ways to express how high-level abstractions can be coupled together. Nodes and tensors in TensorFlow are Python objects, and TensorFlow applications are themselves Python applications.

The actual math operations, however, are not performed in Python. The libraries of transformations that are available through TensorFlow are written as high-performance C++ binaries. Python just directs traffic between the pieces, and provides high-level programming abstractions to hook them together.

TensorFlow applications can be run on most any target that's convenient: a local machine, a cluster in the cloud, iOS and Android devices, CPUs or GPUs. If you use Google's own cloud, you can run TensorFlow on Google's custom TensorFlow Processing Unit (TPU) silicon for further acceleration. The resulting models created by TensorFlow, though, can be deployed on most any device where they will be used to serve predictions.

### *TensorFlow benefits*

The single biggest benefit TensorFlow provides for machine learning development is *abstraction*. Instead of dealing with the nitty-gritty details of implementing algorithms, or figuring out proper ways to hitch the output of one function to the input of another, the developer can focus on the overall logic of the application. TensorFlow takes care of the details behind the scenes.

## 4.6 Keras: The Python Deep Learning library

**Keras** is an open source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, or Theano.<sup>[1]</sup> Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. It was developed as part of the research effort of project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System),<sup>[2]</sup> and its primary author and maintainer is François Chollet, a Google engineer.

In 2017, Google's TensorFlow team decided to support Keras in TensorFlow's core library. Chollet explained that Keras was conceived to be an interface rather than a standalone machine-learning framework. It offers a higher-level, more intuitive set of abstractions that make it easy to develop deep learning models regardless of the computational backend used.<sup>[3]</sup> Microsoft added a CNTK backend to Keras as well, available as of CNTK.



## Features

---

Keras contains numerous implementations of commonly used neural network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier. The code is hosted on GitHub, and community support forums include the GitHub issues page, and a Slack channel.

Keras allows users to productize deep models on smartphones (iOS and Android), on the web, or on the Java Virtual Machine.<sup>[6]</sup> It also allows use of distributed training of deep learning models on clusters of Graphics Processing Units (GPU) and Tensor processing units (TPU).

### MODEL CLASS :

```
def keras_model(image_x,image_y):

    num_of_classes = 37

    model = Sequential()

    model.add(Conv2D(filters=32, kernel_size=(5, 5), input_shape=(image_x, image_y, 1),
activation='sigmoid'))

    model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='same'))

    model.add(Conv2D(64, (5, 5), activation='sigmoid'))

    model.add(MaxPooling2D(pool_size=(5, 5), strides=(5, 5), padding='same'))

    model.add(Flatten())

    model.add(Dense(num_of_classes, activation='softmax'))

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

    filepath = "devanagari_model.h5"

    checkpoint1 = ModelCheckpoint(filepath, monitor='val_acc', verbose=1,
save_best_only=True, mode='max')
```

```
#checkpoint2 = ModelCheckpoint(filepath, monitor='val_loss', verbose=1,
save_best_only=True, mode='min')
```

```
callbacks_list = [checkpoint1]
```

```
return model, callbacks_list
```

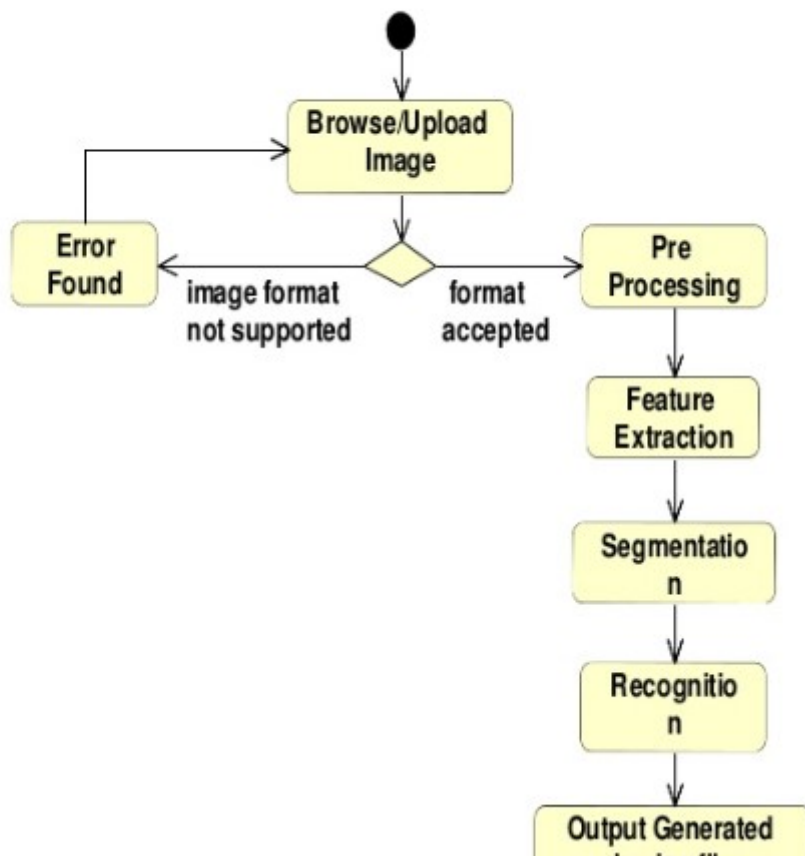
### Driver Class / Application Class :

```
def main():
    if len(pts) != []:
        blackboard_gray = cv2.cvtColor(blackboard, cv2.COLOR_BGR2GRAY)
        blur1 = cv2.medianBlur(blackboard_gray, 15)
        blur1 = cv2.GaussianBlur(blur1, (5, 5), 0)
        thresh1 = cv2.threshold(blur1, 0, 255, cv2.THRESH_BINARY)
        blackboard_cnts = cv2.findContours(thresh1.copy(), cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
        if len(blackboard_cnts) >= 1:
            cnt = max(blackboard_cnts, key=cv2.contourArea)
            print(cv2.contourArea(cnt))
            if cv2.contourArea(cnt) > 2000:
                x, y, w, h = cv2.boundingRect(cnt)
                digit = blackboard_gray[y:y + h, x:x + w]
                # newImage = process_letter(digit)
                pred_probab, pred_class = keras_predict(model1, digit)
                print(pred_class, pred_probab)

        pts = deque(maxlen=512)
        blackboard = np.zeros((480, 640, 3), dtype=np.uint8)
        cv2.putText(img, "Conv Network : " + str(letter_count[pred_class]),
                    (10, 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
        cv2.imshow("Frame", img)
        cv2.imshow("Contours", thresh1)
        k = cv2.waitKey(10)
        if k == 27:
            break

def keras_predict(model, image):
    processed = keras_process_image(image)
    print("processed: " + str(processed.shape))
    pred_probab = model.predict(processed)[0]
    pred_class = list(pred_probab).index(max(pred_probab))
    return max(pred_probab), pred_class
```

**Activity diagram :**

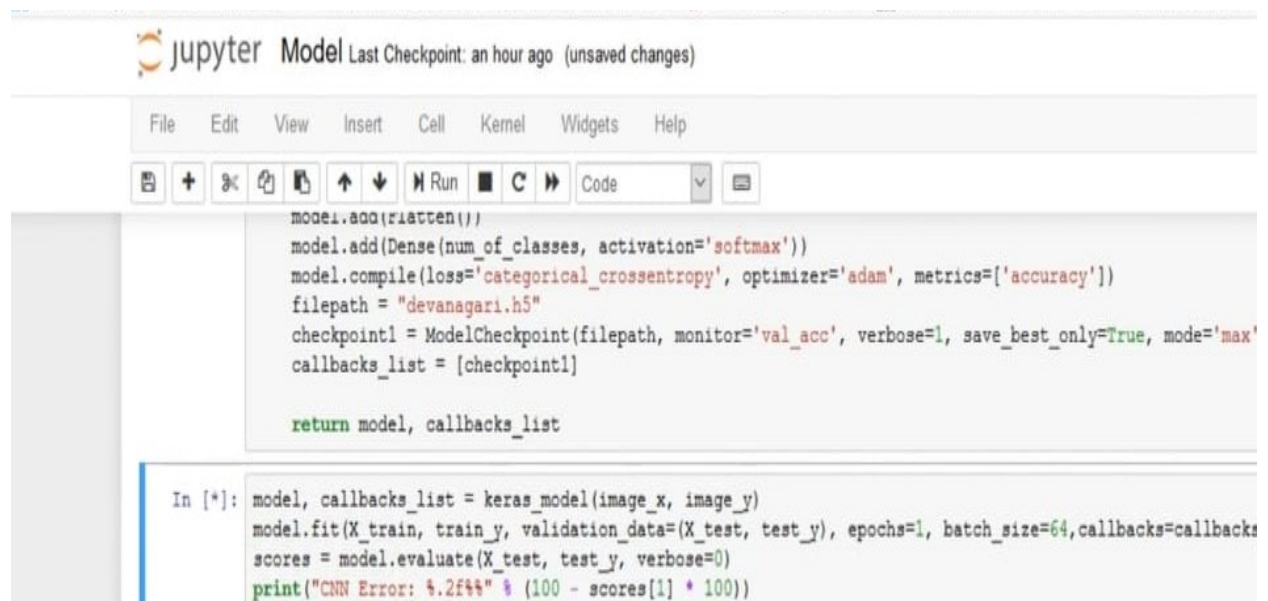


# CHAPTER 5

## SCREENSHOTS

Training Class Model :

Training process takes almost 2 hours to train the model.



The screenshot shows a Jupyter Notebook window titled "Model" with a subtitle "Last Checkpoint: an hour ago (unsaved changes)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, cell navigation, and execution. The code is written in a light gray cell and consists of two parts: a function definition and an execution cell.

```
model.add(Flatten())
model.add(Dense(num_of_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
filepath = "devanagari.h5"
checkpoint1 = ModelCheckpoint(filepath, monitor='val_acc', verbose=1, save_best_only=True, mode='max')
callbacks_list = [checkpoint1]

return model, callbacks_list
```

The execution cell, labeled "In [\*]:", contains the following code:

```
model, callbacks_list = keras_model(image_x, image_y)
model.fit(X_train, train_y, validation_data=(X_test, test_y), epochs=1, batch_size=64, callbacks=callbacks_list)
scores = model.evaluate(X_test, test_y, verbose=0)
print("CNN Error: %.2f%%" % (100 - scores[1] * 100))
```

Trained model output :

```
In [12]: model, callbacks_list = keras_model(image_x, image_y)
model.fit(X_train, train_y, validation_data=(X_test, test_y), epochs=1, batch_size=240, verbose=1)
scores = model.evaluate(X_test, test_y, verbose=0)
print("CNN Error: %.2f%%" % (100 - scores[1] * 100))
print_summary(model)
model.save('devanagari.h5')
```

Train on 70000 samples, validate on 2000 samples  
Epoch 1/1  
70000/70000 [=====] - 243s 3ms/step - loss: 0.778  
240

Epoch 00001: val\_acc improved from -inf to 0.92400, saving model to devanagari.h5  
CNN Error: 7.60%

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0

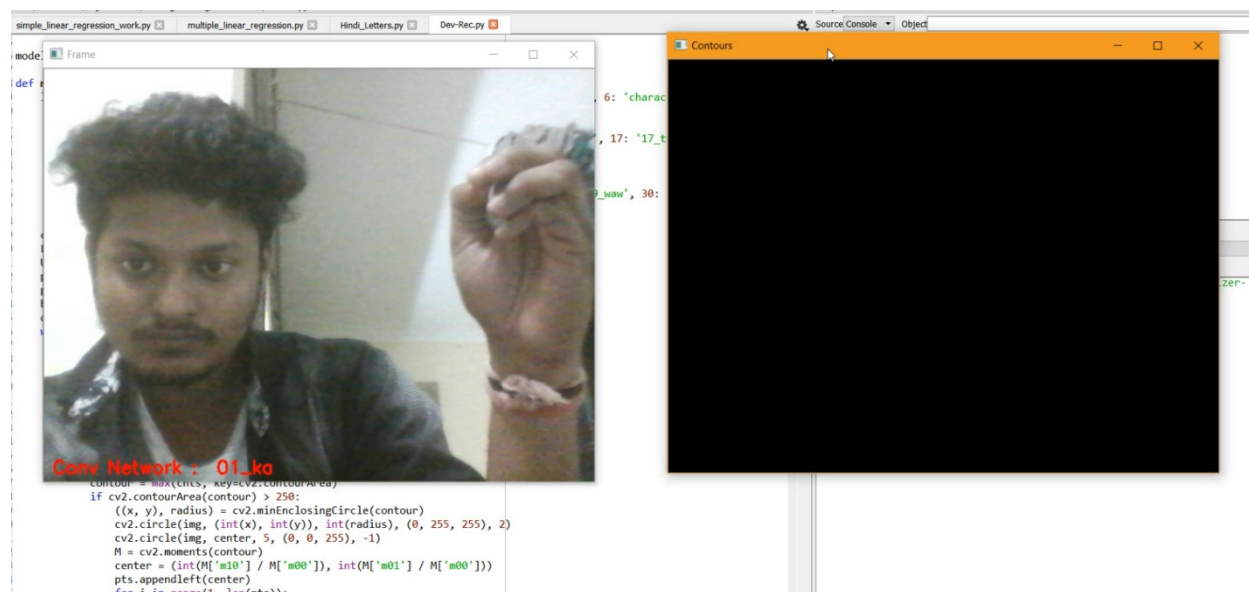
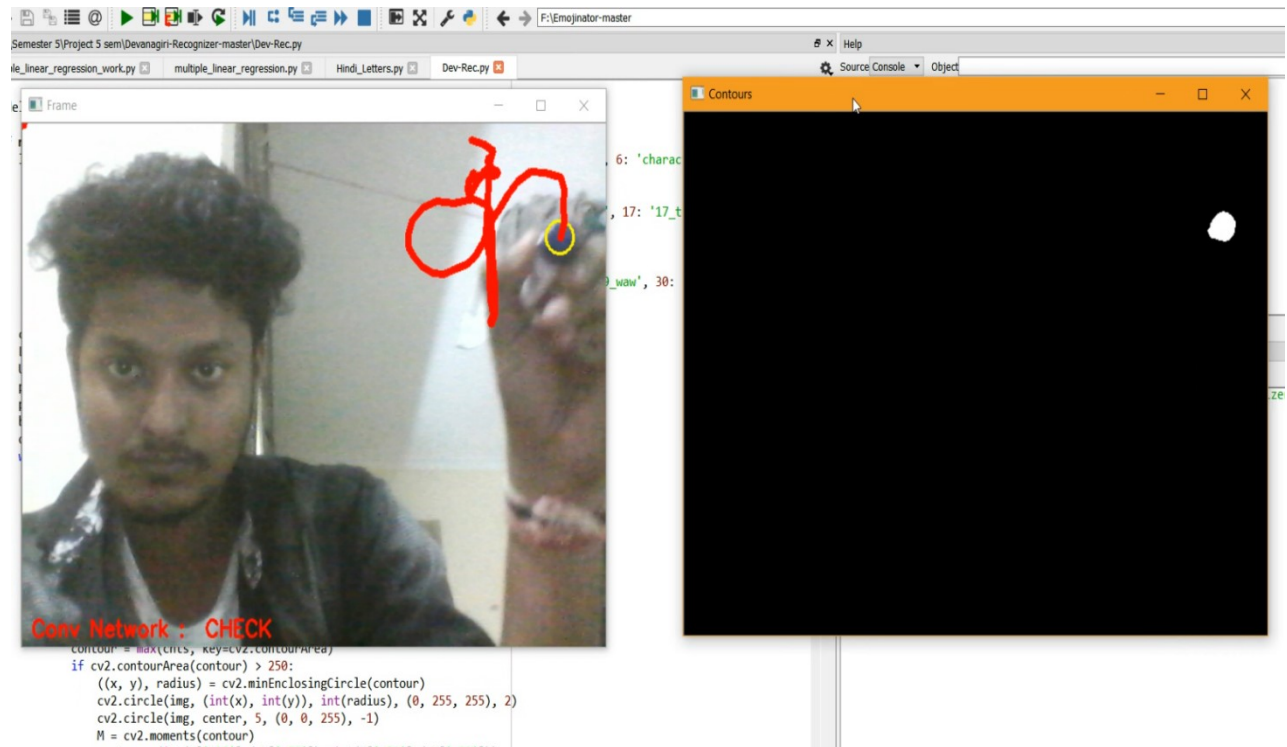
```
240
```

Epoch 00001: val\_acc improved from -inf to 0.92400, saving model to devanagari.h5  
CNN Error: 7.60%

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_2 (Conv2D)	(None, 10, 10, 64)	51264
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 64)	0
flatten_1 (Flatten)	(None, 256)	0

## Application Class:

Giving input to machine using blue object, as its gonna recognized by camera as contour.



Predicted /Recognized value :





## CHAPTER 6

### CONCLUSION AND FUTURE SCOPE

#### 6.0 Approached Result

The techniques for the recognition of handwritten Hindi text by segmenting and classifying the characters have been proposed in this thesis work. The problems in handwritten Hindi text written by different persons are identified after carefully analyzing the text. To solve these problems new techniques have been developed for segmentation, feature extraction and recognition. In the present work, four new segmentation algorithms have been proposed. These include text line segmentation, segmentation of half characters, segmentation of lower modifiers and segmentation of touching left modifier from consonant in the middle region of the word. A new technique based on header line and base line detection to segment the overlapped lines of text in handwritten Hindi text have been proposed. Determination of the header line is very tough. The position of the header line in particular line of text and header line in a particular word of the same line may vary. Determining the presence of lower modifier, presence of half character, touching left modifier with consonant in the middle region or to determine the presence of touching characters is very arduous. The new threshold values for their presence in the word have been proposed. For segmentation of half characters from consonants structural properties of the text are considered. For segmentation of lower modifiers, a new technique based on shape of lower modifiers is proposed. A technique based on position and length of the left modifier is proposed for segmentation of left modifier from touching consonant in the middle region of the word. For the validity of the algorithms, the proposed algorithms are also tested on printed Hindi text and obtained pleasing results. A new technique called merging of features for the feature extraction has been proposed in the present work. A particular feature of particular character may depend upon other feature of the character. In such cases, next feature is extracted only if previous feature is available otherwise not. It leads to reduce in number of features to be extracted. Further, the problems in feature extraction are identified and many heuristics are applied to solve those problems. The overall results obtained with proposed algorithms for segmentation and recognition of handwritten Hindi text is very challenging. SVM and Rule based classifiers are used for the classification of characters of handwritten Hindi text.



## 6.1 Conclusion and Futurescope

The proposed algorithms used for segmentation of handwritten Hindi text can be extended further for recognition of other Indian scripts.

- The proposed algorithms of segmentation can be modified further to improve accuracy of segmentation.
- New features can be added to improve the accuracy of recognition. • These algorithms can be tried on large database of handwritten Hindi text.
- There is a need to develop the standard database for recognition of handwritten Hindi text.
- The proposed work can be extended to work on degraded text or broken characters.
- Recognition of digits in the text, half characters and compound characters can be done to improve the word recognition rate.

## REFERENCES:

1. W.K. Pratt, Digital Image Processing, Wiley, 1978.
2. Digital image processing (Rafeel C.Gonzalez,Richard E. Woods).
3. Fundamental of Electronic Image Processing (Arthur R. Weeks, Jr. )
4. M. Minsky, S. Papert, Perceptrons, MIT Press, Cambridge
5. Dejm Gorgevik, Dusan Cakmakov',
6. Cheng-Lin Liu, Kazuki Nakashima, Hiroshi Sako, Hiromichi Fujisawa, ] Handwritten Digit Recognition Using State-of-the-Art Techniques 2002 IEEE
7. Gang Liu, Di Wu, Hong-Gang Zhang, Jun Guo , A new Feature Extraction Method Based on Fourier Transform in Handwriting Digits Recognition, IEEE 2002
8. Dejan Gorgevik, Dusan Cakmakov'', Vladimir Radevski', Handwritten digit recognition by combining support vector machines using rule based ITI 2001