VISVESVARAYA TECHNOLOGICAL UNIVERSITY
Jnana Sangama, Belagavi, Karnataka-590014

Project Report On

"APPLICATION OF CONVOLUTION NEURAL NETWORKS FOR STATIC AND REAL-TIME HAND GESTURE RECOGNITION UNDER DIFFERENT INVARIANT FEATURES"

Submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF ENGINEERING
in
ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

SUMUKHA MANJUNATH	**1BI14EC165**
SURYA PRAKASH	**1BI14EC168**
SYED MOHSIN	**1BI14EC169**
SMARAN P	**1BI14EC159**

Carried out at
Bangalore Institute of Technology
K.R.Road, V.V.Puram
Bangalore-560004

Under the Guidance of

Mrs. K.NIRMALA KUMARI
Associate Professor
Department of ECE
BIT, Bangalore- 04

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
BANGALORE INSTITUTE OF TECHNOLOGY
K .R. Road, V.V Puram, Bangalore-560004
2017-2018

BANGALORE INSTITUTE OF TECHNOLOGY

V. V Puram, K R Road, Bangalore-560 004

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

Certified that the project work entitled **"APPLICATION OF CONVOLUTION NEURAL NETWORKS FOR STATIC AND REAL-TIME HAND GESTURE RECOGNITION UNDER DIFFERENT INVARIANT FEATURES"** carried out by **Mr. SUMUKHA MANJUNATH (USN: 1BI14EC165), Mr. SURYA PRAKASH (USN: 1BI14EC168), Mr. SYED MOHSIN (USN: 1BI14EC169), Mr. SMARAN P (USN: 1BI14EC159),** bonafide students of the **Bangalore Institute of Technology**, in partial fulfillment for the award of **Bachelor of Engineering** in **Electronics and Communication Engineering** of Visvesvaraya Technological University, Belgaum during the year 2017-2018. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the department library. The project report has been approved as it satisfies the academic requirements in report of Project work prescribed for the said Degree.

K. NIRMALAKUMARI Associate Professor Dept. of ECE B.I.T, Bangalore-04	**Dr. K.V. PRASAD** Professor & HOD Dept. of ECE B.I.T, Bangalore-04	**Dr. A.G. NATARAJ** Principal B.I.T, Bangalore-04

External Viva

Name of the examiner	**Signature with date**
1) ……	……
2) ……	……

ACKNOWLEDGEMENT

The success of this project required a lot of guidance and assistance from a lot of people and we were fortunate enough to get all the necessary inputs throughout the completion of our project. So, we would like to take this opportunity to thank all of them.

Firstly, we would like to thank **BANGALORE INSTITUTE OF TECHNOLOGY** for providing us with all the facilities and a comfortable environment to carry out our project work.

We heartily thank our internal project guide **Mrs. K. NIRMALA KUMARI** , Associate Professor, Department of Electronics and Communication, Bangalore Institute of Technology, for her guidance and suggestions during the course of our project.

We would also like to thank **Dr. K V PRASAD**, Professor and Head of the Department of Electronics and Communication, Bangalore Institute of Technology, without whose support this project would not have been successful.

We would also like to acknowledge, in particular **Dr. A.G NATARAJ**, Principal, B.I.T for his invaluable support during this endeavor and the freedom he gives to think and timely support in all forms he extends.

SUMUKHA MANJUNATH **USN:1BI14EC165**

SURYA PRAKASH **USN:1BI14EC168**

SYED MOHSIN **USN:1BI14EC169**

SMARAN P **USN:1BI14EC159**

ABSTRACT

The present work proposes to recognize both static and real-time hand gestures taken under invariations features as scale, rotation, translation, illumination, noise and background. We use the alphabet of American sign language (ASL). For this purpose, digital image processing techniques are used to eliminate or reduce noise, to improve the contrast under a variant illumination, to separate the hand from the background of the image. We make use of convolutional neural networks (CNN) ,which is one of the deep learning algorithms that comes under neural networks to classify the 24 hand gestures. Two CNN architectures were developed with different amounts of layers and parameters per layer. The tests showed that the first CNN(LeNet-5) has an accuracy of 90.25% and the second CNN(Inception V3) has an accuracy of 93.6% in terms of recognition of the 24 static hand gestures using the database developed. We compared the two architectures developed in accuracy level for each type of invariance presented in this paper. We compared the two architectures developed and usual techniques of machine learning in results of accuracy. We have also presented the accuracy with which the network predicts the hand gestures shown in real-time.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Gestures are the most raw and natural form of languages and could be dated back to as early as the advent of human civilization. Gestures include movements of the hands, face or other parts of the body like shrugging of shoulders, nodding heads etc. A sign language is a non-verbal method of communication composed of various gestures formed by different hand shapes, movements and orientation of hands and is a more organized form than gestures. There are various sign languages across the world, each with its own vocabulary. These include American Sign Language (ASL) in North America, British Sign Language (BSL) in Great Britain, Indian Sign Language (ISL) etc.

By using image processing techniques, a vision based interaction can be established. The most contributing reason for the emerging gesture recognition is that they can create a simple communication path between human and computer called HCI (Human Computer Interaction). However, a vision based hand tracking and gesture recognition is a challenging problem due to the complexity of hand gestures which are rich in diversities due to high degrees of freedom (DOF) involved by the human hand. In order to successfully fulfill their role, the hand gesture HCIs have to meet the requirements in terms of real-time performance, recognition accuracy and robustness against transformations and cluttered background.

Recognition of hand gestures is a natural medium used for human computer interaction (HCI), is a very active area of research in computer vision and in Machine Learning, each year receives more attention due to its multiple applications in various areas as robotics, Video games, sign language and virtual reality.

CHAPTER 2

HAND GESTURE DETECTION

Hand gesture is the most powerful and frequently used gesture in people's daily life. In linguistics, it is an important component of body language. Applications of hand gesture recognition includes teleconferencing, interpretation and learning of sign languages, telerobotics, controlling television set remotely, enabling hand as a 3D mouse and so on. Hand gesture recognition can be considered as a complex problem since gestures vary between individuals and for the same individual based on different contexts. The challenges in this technology include uncontrolled environment and lighting conditions, skin colour detection, rapid hand motions and self occlusions.

Several approaches have been introduced for handling hand gesture recognition in which some makes use of mathematical algorithms whereas some is based on soft computing. The practical implementation of gesture recognition requires devices or gadgets such as those for tracking hand motion, imaging etc. Glove based technique require user to wear devices that needs loads of cables to connect it to the computer thereby reducing the ease of interaction. Vision based techniques makes use of cameras and can obtain properties such as texture and colour of hand for identifying gesture, while dealing with problems due to illumination changes, complicated background, camera movement, specific user variance and self occlusions.

The present work proposes a method whose sequence of steps is shown in Figure 2.1

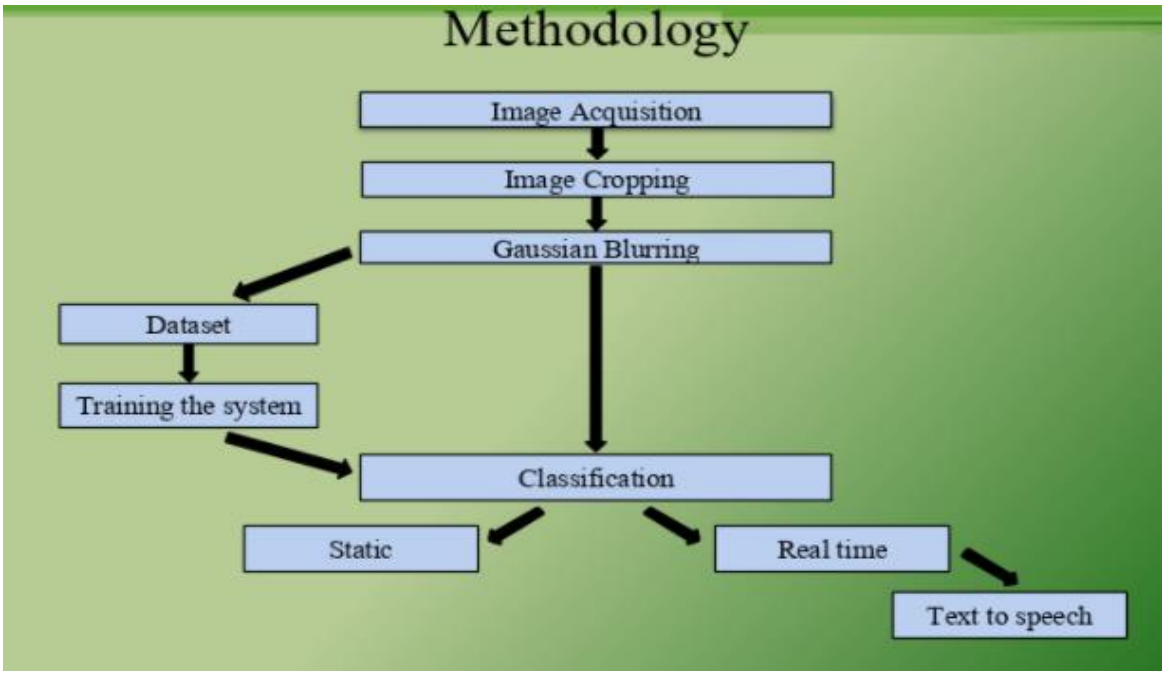


Figure 2.1 Flow diagram of the proposed method

2.1 Capturing Images

The database presents photos of hand gestures based on the alphabet of ASL of a total of 4 people. Generally, for the hand identification process it should not be complex, if the image was taken with a simple background and regular illumination, as seen in figure 2.2. However, images taken with complex backgrounds and varied illumination were used with invariant features. The database consists of 19200 images for the training set and 4800 images for testing, each image was taken with a size of 250×276, are .jpg format. The developed database was called “Dataset”.

Figure 2.2 Images from the database “Dataset”

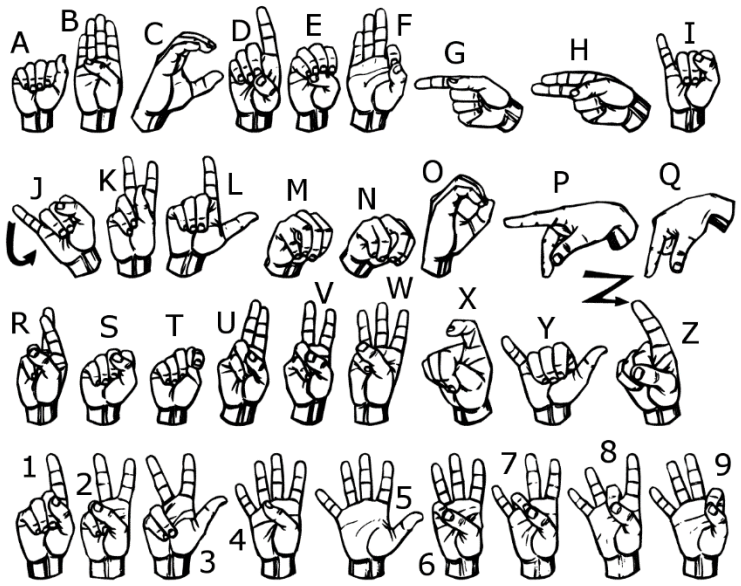


Figure 2.3 Images of hand gestures developed based on the alphabet of the ASL

2.2 Preprocessing

This describes the methods used to prepare images for further analysis, including interest point and feature extraction. Some of these methods are also useful for global and local feature description, particularly the metrics derived from transforms and basis spaces.

To improve the image, we proceed to reduce impulsive noise (salt and pepper), making a smoothing based on the median filter, this type of nonlinear filter is widely used in digital image processing because it works very well with impulsive noises on the other hand reduces noise levels.

2.2.1 Image Cropping

Cropping is the removal of unwanted areas from a photographic or illustrated image. The process often consists the removal of the outer parts or background of an image to improve framing or change aspect ratio to accentuate or isolate the subject matter.

The Image.open() function returns a value of the Image object data type, which is how Pillow represents an image as a Python value. You can load an Image object from an image file (of any format) by passing the `img = Image.open(Image location)` function a string of the filename. Then, using the function `image_crop = img.crop( starting_x_axis_pixel_location, starting_x_axis_pixel_location, starting_x_axis_pixel_location, starting_x_axis_pixel_location).` Here a cropping of (x->350:600, y->128:400) is done for getting the right size of the hand gesture and is shown in figure 2.2.

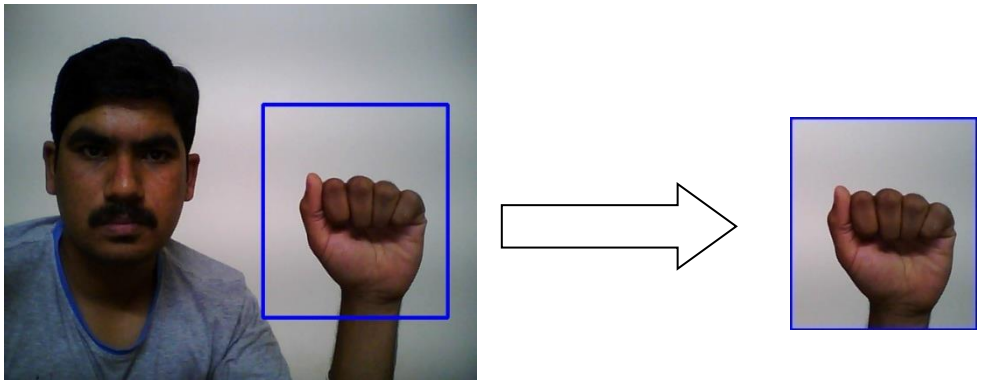


Figure 2.4 Detection and cropping of the gesture of the hand

2.2.2 Gaussian Blurring

In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely used effect in graphics software, typically to reduce image noise and reduce detail. The visual effect of this blurring technique is a smooth blur resembling that of viewing the image through a translucent screen, distinctly different from the bokeh effect produced by an out-of-focus lens or the shadow of an object under usual illumination. Gaussian smoothing is also used

as a pre-processing stage in computer vision algorithms in order to enhance image structures at different scales.

A Gaussian blur effect is typically generated by convolving an image with a kernel of Gaussian values. In practice, it is best to take advantage of the Gaussian blur's separable property by dividing the process into two passes. In the first pass, a one-dimensional kernel is used to blur the image in only the horizontal or vertical direction. In the second pass, the same one-dimensional kernel is used to blur in the remaining direction. The resulting effect is the same as convolving with a two-dimensional kernel in a single pass, but requires fewer calculations.

Discretization is typically achieved by sampling the Gaussian filter kernel at discrete points, normally at positions corresponding to the midpoints of each pixel. This reduces the computational cost but, for very small filter kernels, point sampling the Gaussian function with very few samples leads to a large error. In these cases, accuracy is maintained (at a slight computational cost) by integration of the Gaussian function over each pixel's area.

The equation of a Gaussian function in two dimension is

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{(2\sigma^2)}} \quad \text{.....(2.1)}$$

Here, it is done with the function, cv2.GaussianBlur(). We should specify the width and height of kernel which should be positive and odd. We also should specify the standard deviation in X and Y direction, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as same as sigmaX. If both are given as zeros, they are calculated from kernel size. Gaussian blurring is highly effective in removing gaussian noise from the image. A Gaussian kernel can be created with the function, cv2.getGaussianKernel(). The effect of the Gaussian blurring is shown in figure 2.5.

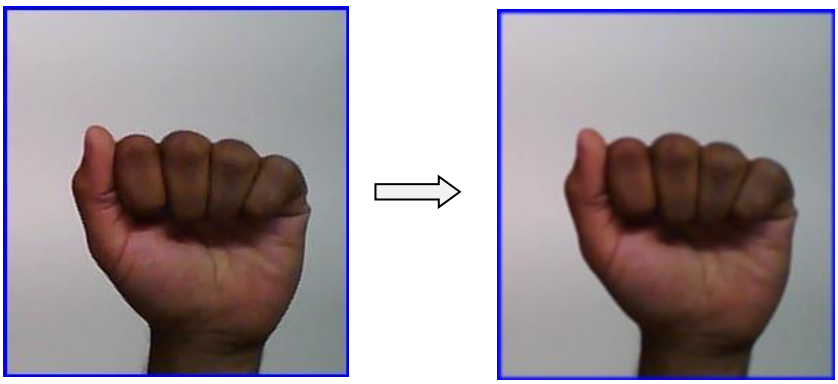


Figure 2.5 Image of a hand gesture before and after gaussian blurring

CHAPTER 3

CONVOLUTIONAL NEURAL NETWORKS

3.1 Neural Networks

Machine learning uses algorithms to parse data, learn from that data, and make informed decisions based on what it has learned. Deep learning structures algorithms in layers to create an “artificial neural network” that can learn and make intelligent decisions on its own. The main difference between machine learning and deep learning is in the feature engineering. In traditional machine learning, we need to handcraft the features .In contrary, in deep learning, feature engineering is done by the algorithm itself. Feature engineering is time consuming and requires domain expertise the promise of deep learning is more accurate algorithms than traditional machine learning with less or no feature engineering at all. These are shown in figure 3.1.

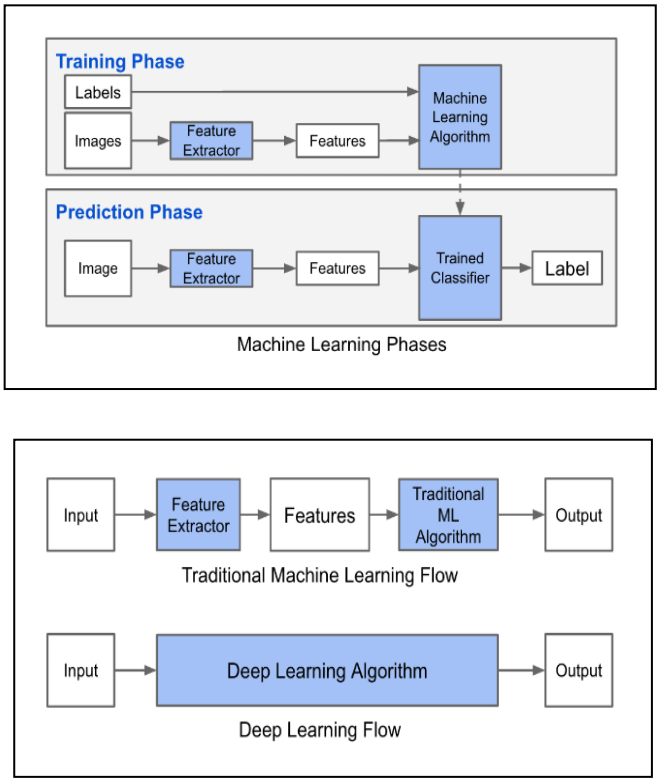


Figure 3.1 Traditional machine learning and deep learning flow

In information technology, a neural network is a system of hardware and/or software patterned after the operation of neurons in the human brain. Neural networks -- also called artificial neural networks -- are a variety of deep learning technologies. Commercial applications of these technologies generally focus on solving complex signal processing or pattern recognition problems. Examples of significant commercial

applications since 2000 include handwriting recognition for check processing, speech-to-text transcription, oil-exploration data analysis, weather prediction and facial recognition.

A neural network usually involves a large number of processors operating in parallel and arranged in tiers. The first tier receives the raw input information -- analogous to optic nerves in human visual processing. Each successive tier receives the output from the tier preceding it, rather than from the raw input -- in the same way neurons further from the optic nerve receive signals from those closer to it. The last tier produces the output of the system.

Each processing node has its own small sphere of knowledge, including what it has seen and any rules it was originally programmed with or developed for itself. The tiers are highly interconnected, which means each node in tier *n* will be connected to many nodes in tier *n-1* -- its inputs -- and in tier *n+1*, which provides input for those nodes. There may be one or multiple nodes in the output layer, from which the answer it produces can be read.

Neural networks are notable for being adaptive, which means they modify themselves as they learn from initial training and subsequent runs provide more information about the world. The most basic learning model is centered on weighting the input streams, which is how each node weights the importance of input from each of its predecessors. Inputs that contribute to getting right answers are weighted higher.

Typically, a neural network is initially trained, or fed large amounts of data. Training consists of providing input and telling the network what the output should be. For example, to build a network to identify the faces of actors, initial training might be a series of pictures of actors, non-actors, masks, statuary, animal faces and so on.

Neural networks are made up many layers, and each layer was consisted of different number of neurons which have trainable weights and biases. All the neurons are fully connected to the neurons in previous and post layers. The first layer is input layer which was viewed as a single vector. The last layer is output layer whose output view be as predict result. Other layers between input and output layer are call hidden layer which will process and pass the ’message’ from previous layer to post layer. Every neuron will receive some inputs from neurons in previous layer. Then it performs a dot product of all the inputs, following with a non-linearity optionally function as output of this neuron. The structure of a deep neural network is shown in figure 3.2.

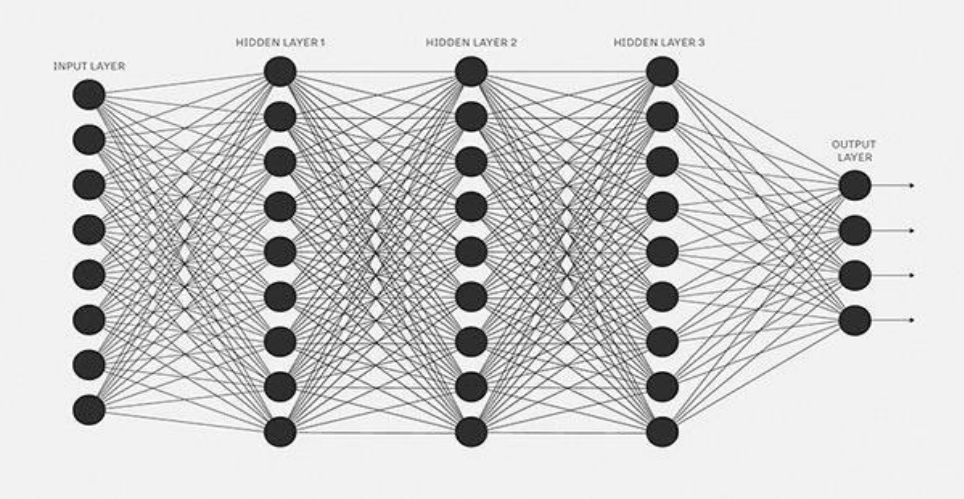
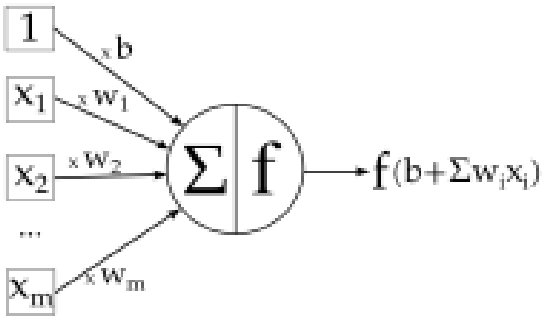


Figure 3.2 Structure of a deep neural network

Neural Networks are essentially mathematical models to solve an optimization problem. They are made of neurons, the basic computation unit of neural networks. A neuron takes an input(say x), do some computation on it(say: multiply it with a variable w and adds another variable b) to produce a value (say; z= wx+b). This value is passed to a non-linear function called activation function(f) to produce the final output(activation) of a neuron. There are many kinds of activation functions. One of the popular activation function is Sigmoid, which is:

$$y = \frac{1}{1+e^{-z}} \text{ .....(3.1)}$$

The neuron which uses sigmoid function as an activation function will be called Sigmoid neuron. Depending on the activation functions, neurons are named and there are many kinds of them like ReLU, TanH etc. One neuron can be connected to multiple neurons, like this:



In this example, we can see that the weights are the property of the connection, i.e. each connection has a different weight value while bias is the property of the neuron. This is the complete picture of a sigmoid neuron which produces output y.

3.2 Convolutional Neural Networks

In machine learning, a convolutional neural network (CNN, or ConvNet) is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyzing visual imagery. CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics.

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

CNNs use relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.They have applications in image and video recognition, recommender systems and natural language processing.

A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolution layers, pooling layers, fully connected layers and normalization layers. Description of the process as a convolution in neural networks is by convention. Mathematically it is a cross-correlation rather than a convolution. This only has significance for the indices in the matrix, and thus which weights are placed at which index.

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume (e.g. holding the class scores) through a differentiable function. A few distinct types of layers are commonly used. The networks which have many hidden layers tend to be more accurate and are called deep network and hence machine learning algorithms which uses these deep networks are called deep learning. the structure of a CNN is shown in figure 3.3.

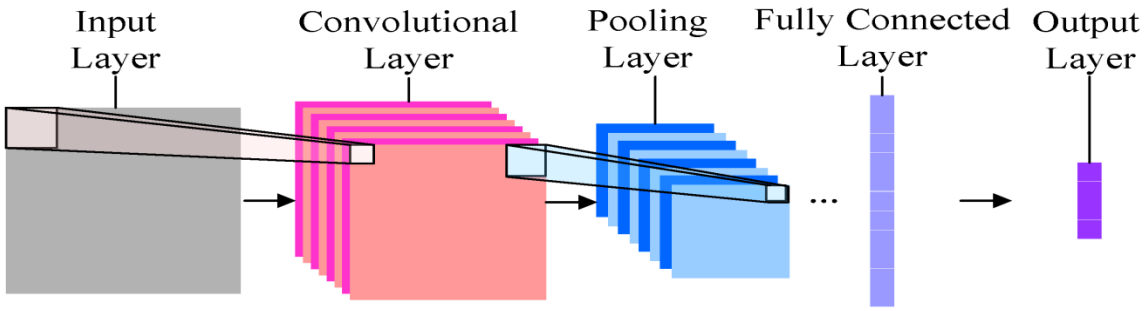


Figure 3.3 Structure of a CNN

The input to CNN , which are images are processed as tensors. Tensors are matrices of numbers with additional dimensions. The images at the input of convolutional layer are 3-D tensors. This is shown in figure 3.4. After convolution is performed they are 4-D tensors.

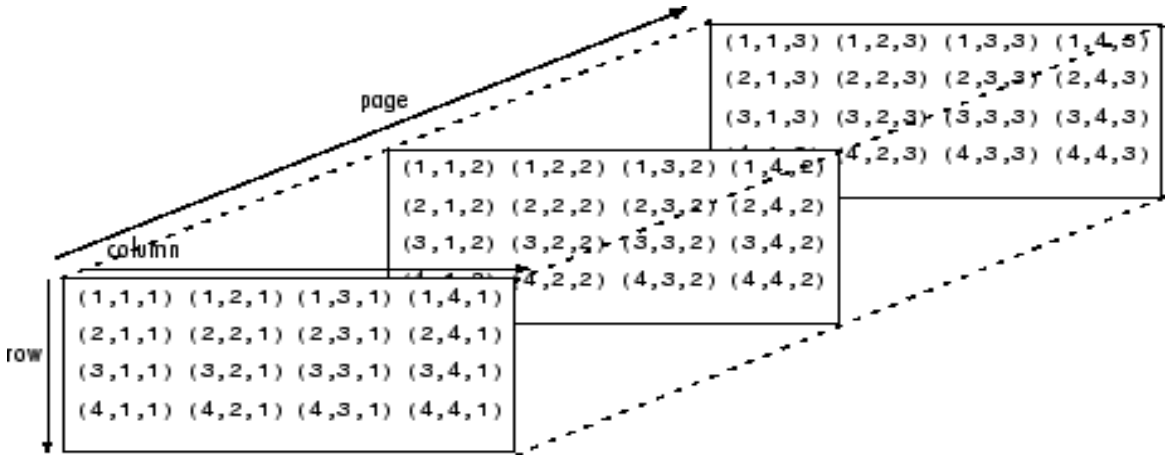


Figure 3.4 3D tensors given as input to the CNN

3.2.1 Convolutional layer

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli.

Each convolutional neuron processes data only for its receptive field.Although fully connected feedforward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5,

each with the same shared weights, requires only 25 learnable parameters. In this way, it resolves the vanishing or exploding gradients problem in training traditional multi-layer neural networks with many layers by using backpropagation. An example of convolution filter of 5×5×3 being applied to a 32×32×3 image is shown in figure 3.5.

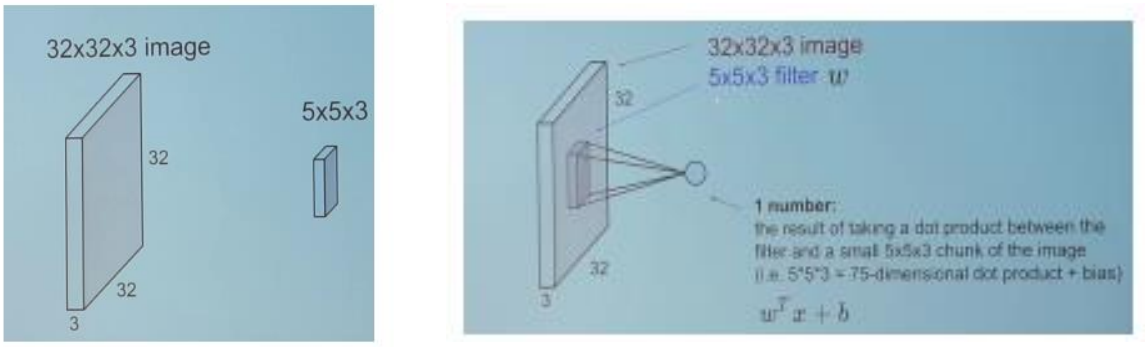


Figure 3.5 Input image of size 32×32 fed into a convolution neuron with 3 channels

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

Convolution is a mathematical operation that is used in signal processing to filter signals and to find patterns in the signal. In a convolutional layer, all neurons apply convolution operation to the inputs, hence they are called convolutional neurons. The process of convolution being applied to an image is shown in figure 3.6 and 3.7.

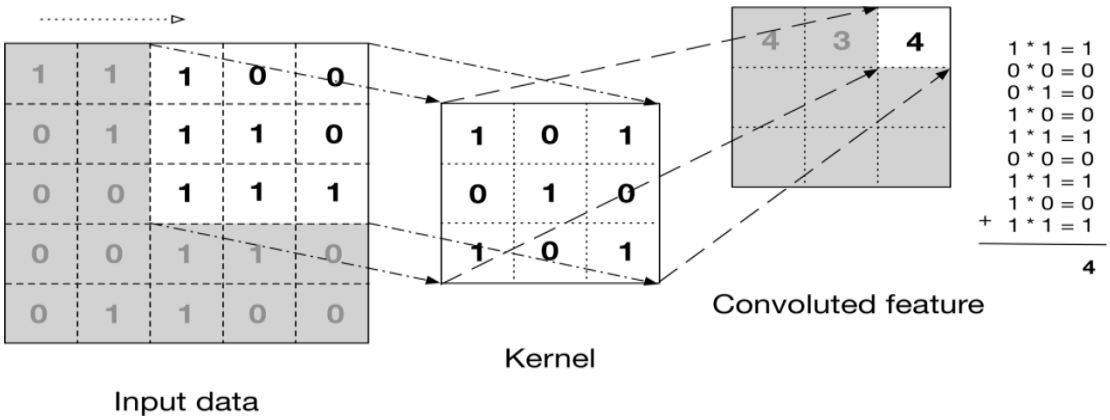


Figure 3.6 An image and its convolution layer output

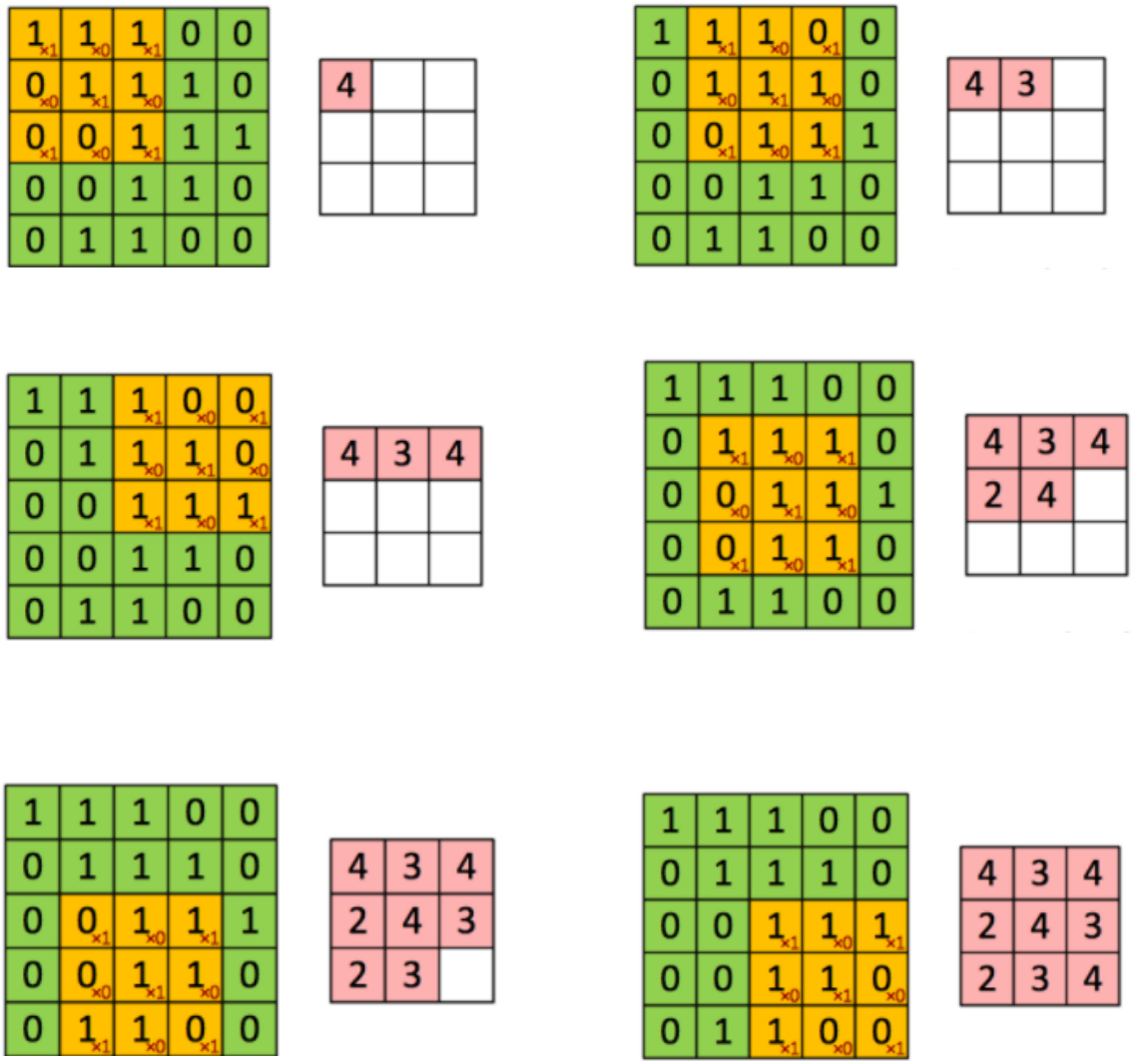
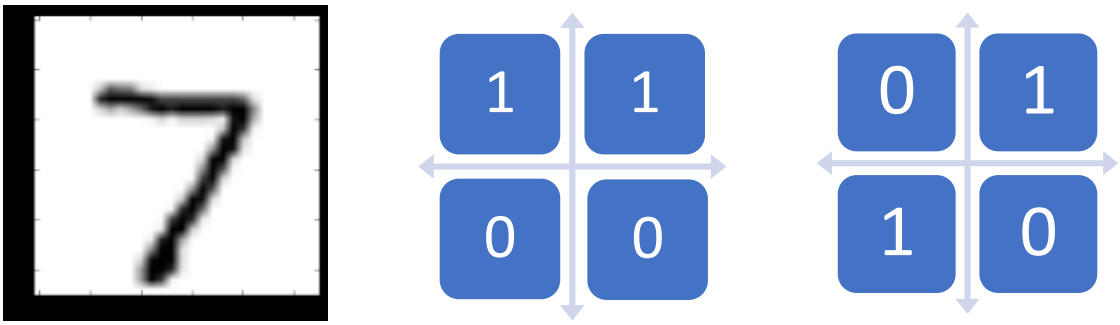


Figure 3.7 Convolution filter being applied on the input image at different patches

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

In the convolutional layer, we are going to take the dot product of the filter with a patch of the image. The output of the dot product can tell us whether the pixel pattern expressed by the filter is present in the underlying image. When the output of convolution performed on a particular patch of pixels produces a high output, then it means that the patch has a shape of the filter being used. This is shown in figure 3.8.



a) Input image b) Filter coefficients for horizontal line and for the diagonal

Figure 3.8 Handwritten text and its filter coefficients

3.2.2 Pooling layer

Convolutional networks may include local or global pooling layers, which combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For example, max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Another example is average pooling, which uses the average value from each of a cluster of neurons at the prior layer.

Pooling is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. It is common to periodically insert a pooling layer between successive convolutional layers in a CNN architecture. The pooling operation provides another form of translation invariance.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form is a pooling layer with filters of size 2x2 with a stride of 2. This is shown in figure 3.9. This configuration of the filter downsamples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every max operation is over 4 numbers. The depth dimension remains unchanged.The number of pixels by which the window is slid is called the stride.

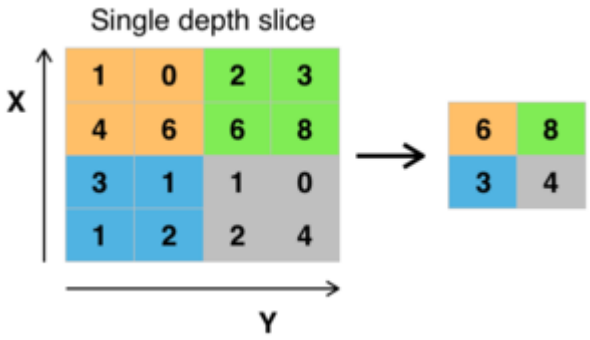


Figure 3.9 Max pooling with a 2x2 filter and stride = 2

In addition to max pooling, the pooling units can use other functions, such as average pooling or L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to max pooling, which works better in practice.

Due to the aggressive reduction in the size of the representation, the trend is towards using smaller filters or discarding the pooling layer altogether. Pooling is an important component of convolutional neural networks for object detection based on Fast R-CNN architecture. A max pooling filter of 2×2 with a stride of 2 applied on the set of images is shown in figure 3.10.

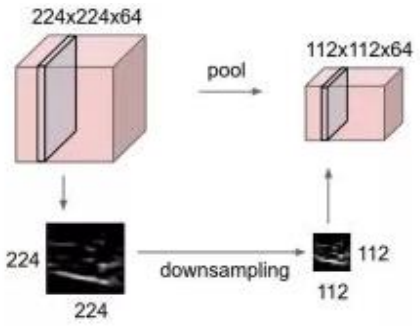


Figure 3.10 Pooling and downsampling in a pooling layer

3.2.3 ReLU layer

ReLU is the abbreviation of Rectified Linear Units. This layer applies the non-saturating activation function f(x)=max(0,x). It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.

Other functions are also used to increase nonlinearity, for example the saturating hyperbolic tangent f(x)=tanh(x), f(x)=|tanh(x)|, and the sigmoid function .ReLU is often preferred to other functions, because it trains the neural network several times faster without a significant penalty to generalisation accuracy.

We use rectified linear unit(ReLU) to threshold the input. These are usually the

next layer after convolutions in convolutional neural networks. We define ReLU the following way that, for any input we are given, we will make sure the output always has values or greater. Rectified linear units find applications in computer vision and speech recognition using deep neural nets.

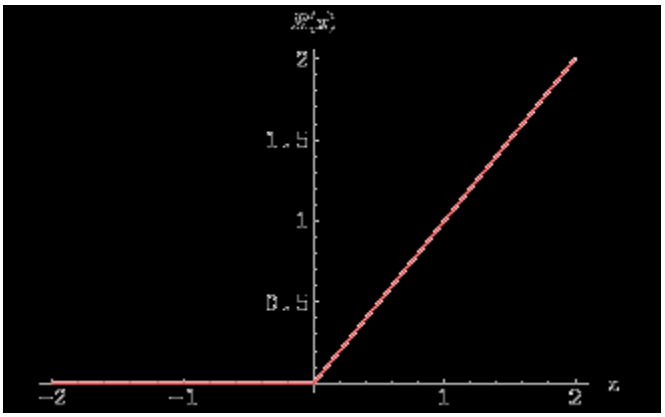


Figure 3.11 A plot of Rectified linear unit

3.2.4 Fully connected layer

Fully connected layers connect every neuron in one layer to every neuron in another layer as shown in figure 3.12. It is in principle the same as the traditional multi-layer perceptron neural network (MLP).

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have 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.

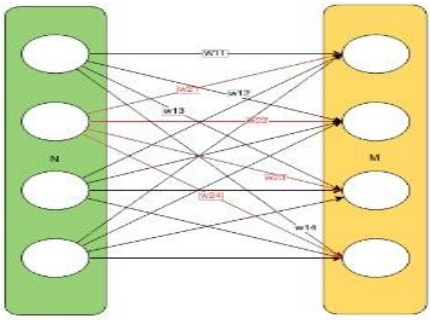


Figure 3.12 Fully connected layer

The final classification in a CNN is performed at the fully connected layer which produces particular labels for the specific class of the input image which is shown in figure 3.13.

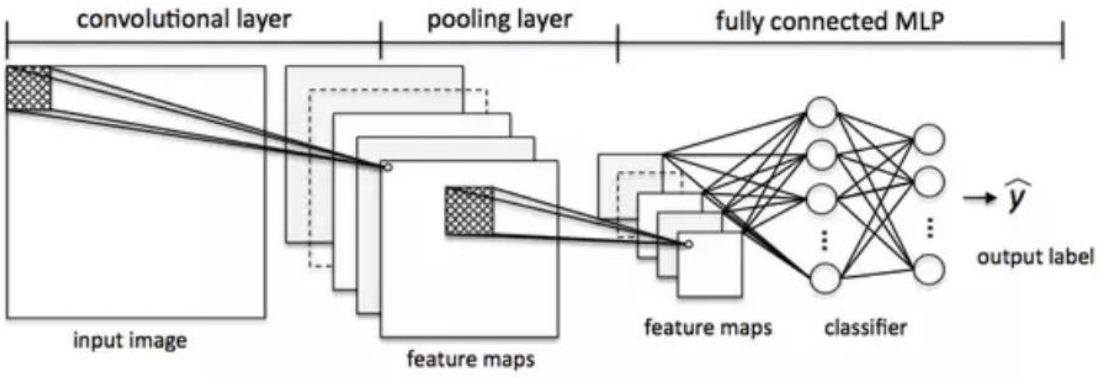


Figure 3.13 The process of classifying images through CNN

3.2.5Weights/Parameters

CNNs share weights in convolutional layers, which means that the same filter (weights bank) is used for each receptive field in the layer; this reduces memory footprint and improves performance.

Once the architecture of the network is decided, the second biggest variable is the weights(w) and biases(b) or the parameters of the network. The objective of the training is to get the best possible values of the all these parameters which solve the problem reliably. For example, while trying to build the classifier between dog and cat, we are looking to find parameters such that output layer gives out probability of dog as 1(or at least higher than cat) for all images of dogs and probability of cat as 1((or at least higher than dog) for all images of cats.

The best set of parameters are found using a process called Backward propagation, i.e., start with a random set of parameters and keep changing these weights such that for every training image we get the correct output. There are many optimizer methods to change the weights that are mathematically quick in finding the correct weights. Gradient Descent is one such method(Backward propagation and optimizer methods to change the gradient is a very complicated topic. But we don't need to worry about it now as Tensorflow takes care of it).

So, if we start with some initial values of parameters and feed 1 training image(in reality multiple images are fed together) of dog and we calculate the output of the network as 0.1 for it being a dog and 0.9 of it being a cat. Now, backward propagation is done to slowly change the parameters such that the probability of this image being a dog increases in the next iteration. There is a variable that is used to govern how fast do we change the parameters of the network during training, it's called learning rate. If we want to maximise the total correct classifications by the network i.e. we care for the

whole training set; we want to make these changes such that the number of correct classifications by the network increases. So we define a single number called cost which indicates if the training is going in the right direction. Typically cost is defined in such a way that; as the cost is reduced, the accuracy of the network increases. So, we keep an eye on the cost and we keep doing many iterations of forward and backward propagations(10s of thousands sometimes) till cost stops decreasing. There are many ways to define cost. One of the simple one is mean root square cost. Let's say yprediction is the vector containing the output of the network for all the training images and yactual is the vector containing actual values(also called ground truth) of these labeled images. So, if we minimize the distance between these two variables, it would be a good indicator of the training. So, we define the cost as the average of these distances for all the images:

$$\text{cost}=0.5\sum_{i=0}^{n}(yactual - yprediction)^2 \quad \text{.....(3.2)}$$

This is a very simple example of cost, but in actual training, we use much more complicated cost measures, like cross-entropy cost. But Tensorflow implements many of these costs so we don't need to worry about the details of these costs at this point in time.

After training is done, these parameters and architecture will be saved in a binary file. In production set-up when we get a new image of dog/cat to classify, we load this model in the same network architecture and calculate the probability of the new image being a cat/dog. This is called inference or prediction.

For computational simplicity, not all training data is fed to the network at once. Rather, let's say we have total 1600 images, we divide them in small batches say of size 16 or 32 called batch-size. Hence, it will take 100 or 50 rounds(iterations) for complete data to be used for training. This is called one epoch, i.e. in one epoch the networks sees all the training images once. There are a few more things that are done to improve accuracy but let's not worry about everything at once.

3.3 Salient features of CNN

While traditional multilayer perceptron (MLP) models were successfully used for image recognition, due to the full connectivity between nodes they suffer from the curse of dimensionality, and thus do not scale well to higher resolution images.

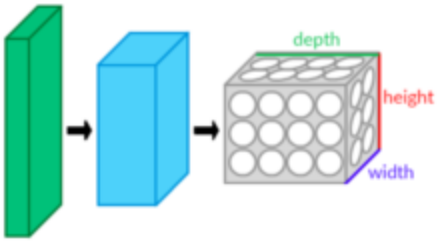


Figure 3.14 CNN layers arranged in 3 dimensions

For example, 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*32*3 = 3,072 weights. A 200x200 image, however, would lead to neurons that have 200*200*3 = 120,000 weights.

Also, such network architecture does not take into account the spatial structure of data, treating input pixels which are far apart in the same way as pixels that are close together. Thus, full connectivity of neurons is wasteful for purposes such as image recognition that are dominated by spatially local input patterns.

Convolutional neural networks are biologically inspired variants of multilayer perceptrons, designed to emulate the behaviour of a visual cortex[citation needed]. These models mitigate the challenges posed by the MLP architecture by exploiting the strong spatially local correlation present in natural images. As opposed to MLPs, CNNs have the following distinguishing features:

- 3D volumes of neurons: The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth. The neurons inside a layer are connected to only a small region of the layer before it, called a receptive field. Distinct types of layers, both locally and completely connected, are stacked to form a CNN architecture.
- Local connectivity: following the concept of receptive fields, CNNs exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers. The architecture thus ensures that the learnt "filters" produce the strongest response to a spatially local input pattern. Stacking many such layers leads to non-linear filters that become increasingly global (i.e. responsive to a larger region of pixel space) so that the network first creates representations of small parts of the input, then from them assembles representations of larger areas.
- Shared weights: In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias)

and form a feature map. This means that all the neurons in a given convolutional layer respond to the same feature within their specific response field. Replicating units in this way allows for features to be detected regardless of their position in the visual field, thus constituting the property of translation invariance.

Together, these properties allow CNNs to achieve better generalization on vision problems. Weight sharing dramatically reduces the number of free parameters learned, thus lowering the memory requirements for running the network and allowing the training of larger, more powerful networks.

3.3.1 Dropout

Because a fully connected layer occupies most of the parameters, it is prone to overfitting. One method to reduce overfitting is dropout. At each training stage, individual nodes are either "dropped out" of the net with probability 1-p or kept with probability p, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights.In the training stages, the probability that a hidden node will be dropped is usually 0.5; for input nodes, this should be much lower, intuitively because information is directly lost when input nodes are ignored.

At testing time after training has finished, we would ideally like to find a sample average of all possible 2^n dropped-out networks; unfortunately this is unfeasible for large values of n. However, we can find an approximation by using the full network with each node's output weighted by a factor of p, so the expected value of the output of any node is the same as in the training stages. This is the biggest contribution of the dropout method: although it effectively generates 2^n neural nets, and as such allows for model combination, at test time only a single network needs to be tested.

By avoiding training all nodes on all training data, dropout decreases overfitting. The method also significantly improves training speed. This makes model combination practical, even for deep neural nets. The technique seems to reduce node interactions, leading them to learn more robust features that better generalize to new data.

3.4 CNN Architectures

Designing the architecture is slightly complicated and advanced topic and takes a lot of research. There are many standard architectures which work great for many

standard problems. Examples being AlexNet, GoogleNet, InceptionResnet, VGG etc.

For example, while training, images from both the classes(dogs/cats) are fed to a convolutional layer which is followed by 2 more convolutional layers. After convolutional layers, we flatten the output and add two fully connected layer in the end. The second fully connected layer has only two outputs which represent the probability of an image being a cat or a dog as shown in figure 3.15.

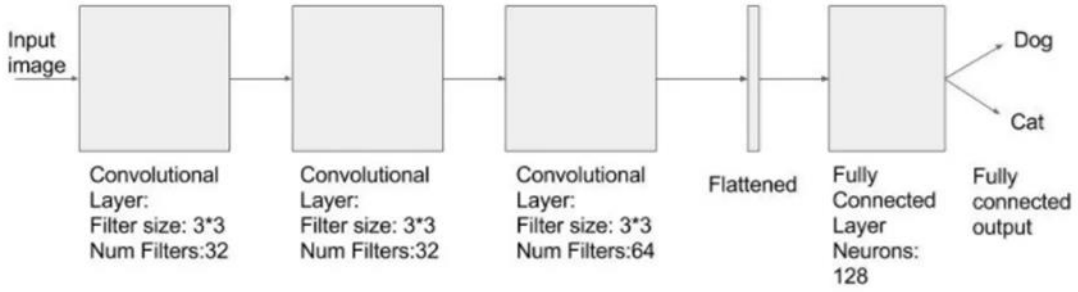


Figure 3.15 CNN structure for the example

3.4.1 LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel images. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources. LeNet architecture is shown in figure 3.16.

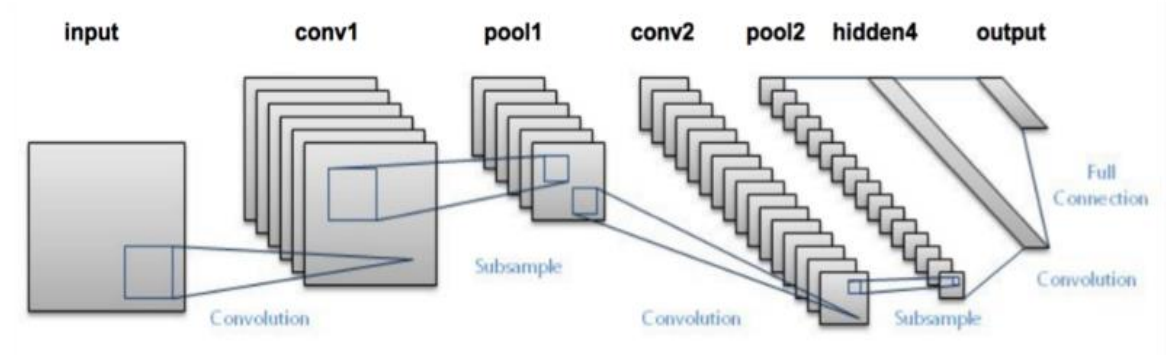


Figure 3.16 LeNet architecture

3.4.2 AlexNet (2012)

In 2012, Alex Net significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.

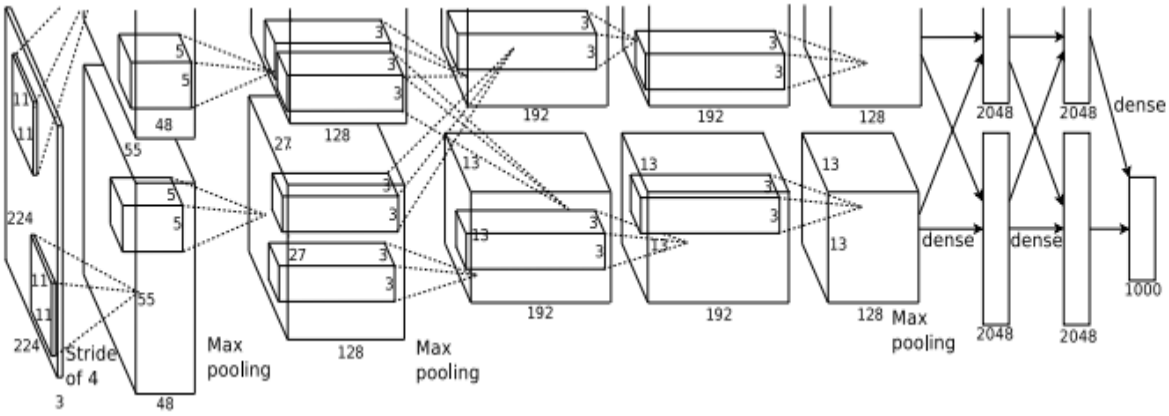


Figure 3.17 AlexNet architecture

AlexNet has parallel two CNN line trained on two GPUs with cross-connections, GoogleNet has inception modules, ResNet has residual connections. The architecture of AlexNet is shown above in figure 3.17.

The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. AlexNet was trained simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines.

3.4.3 ZFNet(2013)

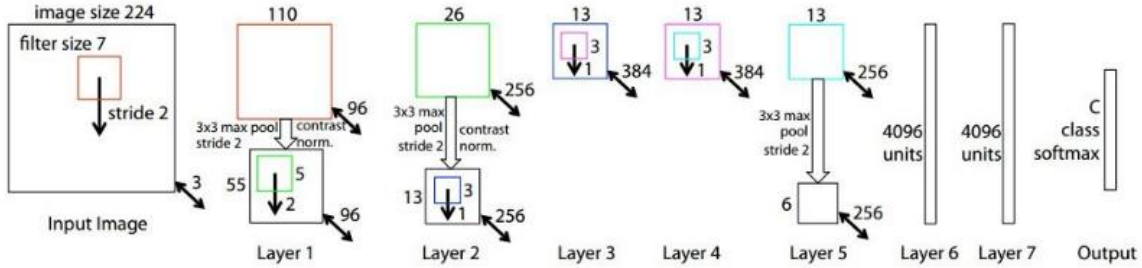


Figure 3.18 ZFNet architecture

Similar framework to LeNet but has a Max pooling and a ReLU nonlinearity layer. It has more data and is a bigger model with 7 hidden layers, 650K units, 60M parameters. ZFNet has achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate. It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements. A general ZFNet architecture is shown in figure 3.18.

3.4.4 GoogleNet/Inception(2014)

The GoogleNet(Inception) is a model designed from Google. It achieved a top-5 error rate of 6.67%! This was very close to human level performance which the organisers of the challenge were now forced to evaluate. As itturns out, this was actually rather hard to do and required some human training in order to beatGoogLeNets accuracy. After a few days of training, the human expert was able to achieve a top-5 error rate of 5.1%. The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million. The architecture of Inception is shown in figure 3.19.

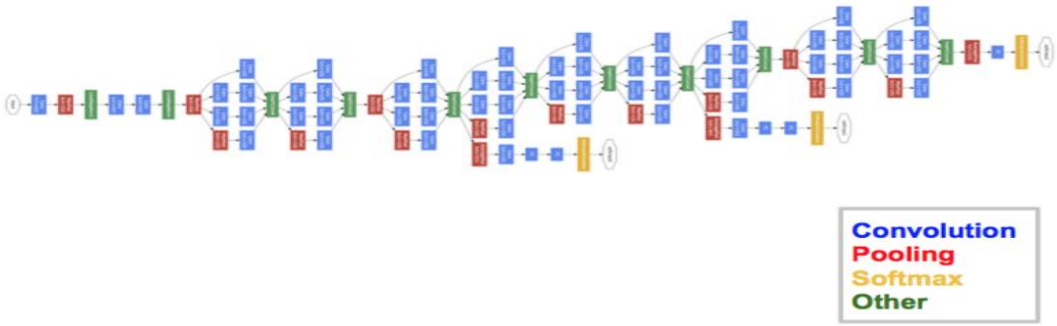


Figure 3.19 Inception architecture

3.4.5 VGGNet (2014)

VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture, which is shown in figure 3.20. It only performs 3×33\times 33×3 convolutions and 2×22\times 22×2 pooling all the way through. It is currently the most preferred choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor. However, VGGNet consists of 140 million parameters, which can be a bit challenging to handle.

Figure 3.20 VGGNet architecture

3.4.6 ResNet(2015)

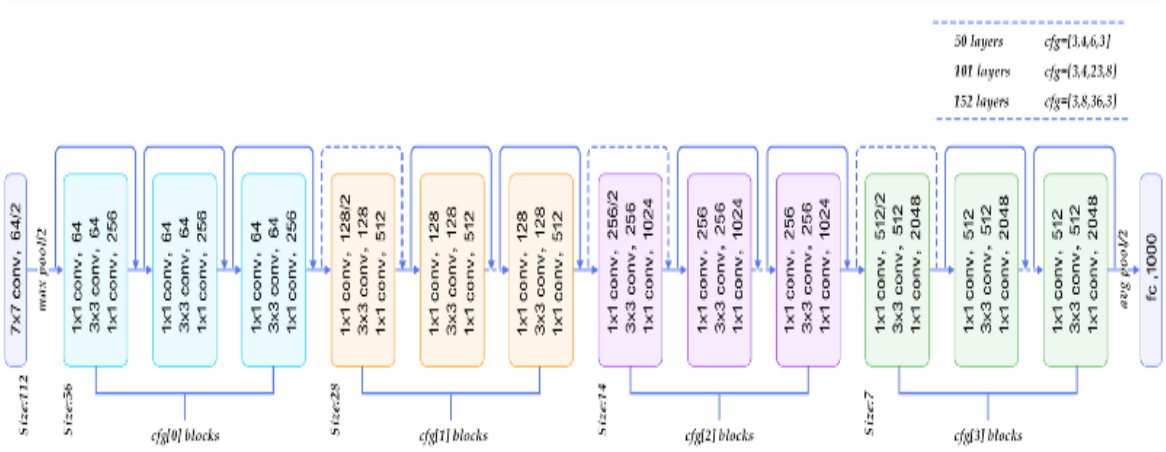


Figure 3.21 RESNet architecture

Residual Neural Network (ResNet) by Kaiming He et al introduced a novel architecture with “skip connections” and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs. Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset. The RESNet architecture is shown in figure 3.21.

The comparison of accuracies of different architectures is shown in figure 3.22.

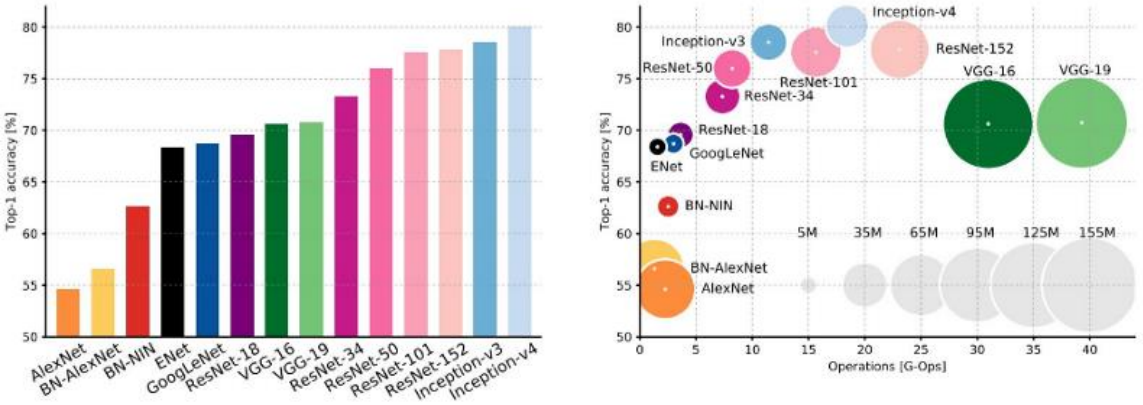


Figure 3.22 Comparison of accuracy of different architectures

3.5 Building an image classifier

The pre-requisites for building a neural network based image classifier are:

- OpenCV: We use openCV to read the images.

- Softmax: It is a function that converts K-dimensional vector ‘x’ containing real values to the same shaped vector of real values in the range of (0,1), whose sum is 1. We shall apply the softmax function to the output of our convolutional neural network in order to, convert the output to the probability for each class.

3.6 Tensorflow

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. It is used for both research and production at Google, often replacing its closed-source predecessor, DistBelief.

TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open source license.TensorFlow is Google Brain's second generation system. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA and SYCL extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays. These arrays are referred to as "tensors".

TensorFlow provides a Python API as well as C++ Haskell,Java, G and Rust APIs. Third party packages are available for C#, Julia, R, Scala and OCaml.

Among the applications for which TensorFlow is the foundation, are automated image captioning software, such as DeepDream. RankBrain now handles a substantial number of search queries, replacing and supplementing traditional static algorithm-based search results.

CHAPTER 4

EXPERIMENTAL PROCEDURE

The networks are trained with the database “Dataset”. The database has 24000 processed images of 250 x 276 pixels of each image. Two deep convolutional neural network architectures are developed, each convolutional neural network receives as input 4096 neurons. All networks were trained and simulated with a Nvidia Geforce GPU, using the Python programming language and using libraries such as Tensorflow, scikit-learn and others.

4.1 Reading The Inputs

The dataset is divided into:

- Training data: we shall use 80% i.e. 19200 images for training.
- Validation data: 20% images(4800) will be used for validation. These images are taken out of training data to calculate accuracy independently during the training process.
- Test set: separate independent data for testing which has around 400 images. Sometimes due to something called Overfitting; after training, neural networks start working very well on the training data(and very similar images) i.e. the cost becomes very small, but they fail to work well for other images. For example, if you are training a classifier between dogs and cats and you get training data from someone who takes all images with white backgrounds. It’s possible that your network works very well on this validation data-set, but if you try to run it on an image with a cluttered background, it will most likely fail. So, that’s why we try to get our test-set from an independent source.

4.2 Comparison Of The Two CNNs Used

The first CNN is based on the LeNET-5 architecture and the second CNN, on Inception V3. In this work two convolutional networks (CNNs) with different depths and parameters are presented.

4.2.1 CNN 1(LeNet-5)

The first convolution neural network uses a LeNet-5 architecture. The LeNet architecture is an excellent “first architecture” for Convolutional Neural Networks

(especially when trained on the MNIST dataset, an image dataset for handwritten digit recognition). A LeNet architecture is shown in figure 4.1.

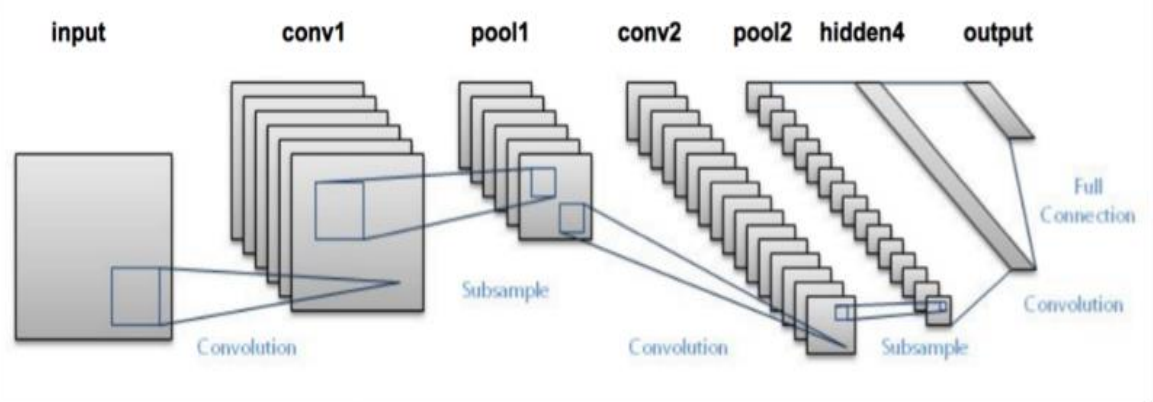


Figure 4.1 LeNet architecture

Table 4.1 CNN 1 Configuration

Convolution 32 filters, 3x3 kernel and ReLU
Max pooling 2x2 kernel
Convolution 32 filters, 3x3 kernel and ReLU
Max pooling 2x2 kernel
Dropout 50%
Fully connected with 1024 neurons
Dropout 50%
Fully connected with 1024 neurons
Dropout 50%
Softmax 24 classes

LeNet is small and easy to understand — yet large enough to provide interesting results. Furthermore, the combination of LeNet + MNIST is able to run on the CPU, making it easy for beginners to take their first step in Deep Learning and Convolutional Neural Networks.

To keep our code organized, we'll define a package named pyimagesearch . And within the pyimagesearch module, we'll create a CNN sub-module — this is where we'll store our Convolutional Neural Network implementations, along with any helper utilities related to CNNs.

Taking a look inside CNN , the networks sub-module is seen where the network implementations themselves will be stored. As the name suggests, the lenet.py file will define a class named LeNet , which is our actual LeNet implementation in Python .

The lenet_mnist.py script is a driver program used to instantiate the LeNet network architecture, train the model (or load the model, if our network is pre-trained), and then evaluate the network performance on the MNIST dataset.

Finally, the output directory will store our LeNet model after it has been trained, allowing us to classify digits in subsequent calls to lenet_mnist.py without having to re-train the network. The CNN configuration for the process is shown above in table 3.1. The flow graph of he architecture used is shown in figure 4.2.

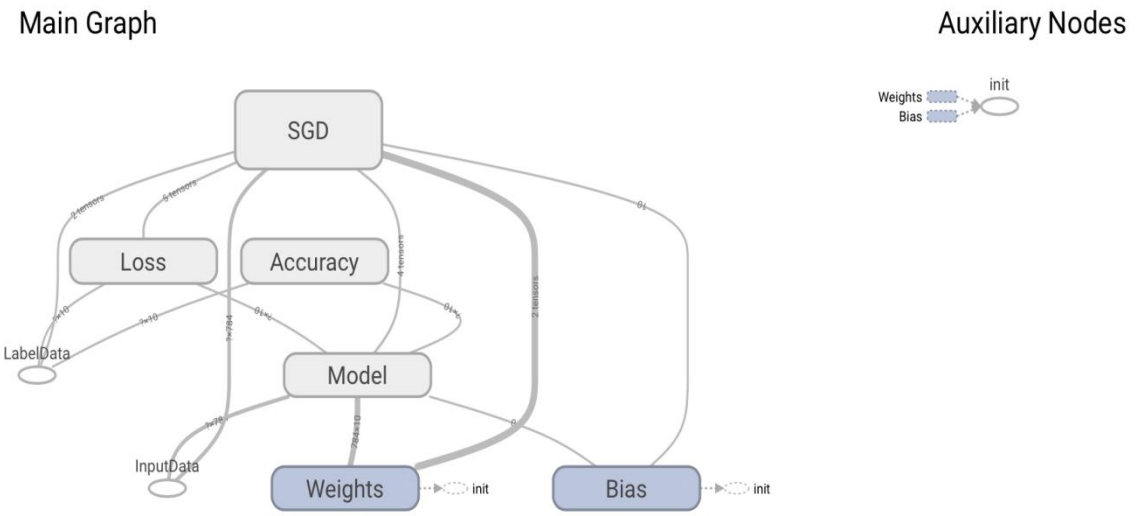


Figure 4.2 Flow graph of the LeNet architecture used

4.2.2 CNN 2(InceptionV3)

The second CNN uses the Inception version 3 model. It consists of input layer ,output layer and many hidden layer.The hidden layers consist of 11 modules and each module consists of 7 layers. Each module is made up of 4 paths. Finally ,the output of each path is concatenated. The architecture of InceptionV3 model, containing many inception modules, is shown in figure 3.22.

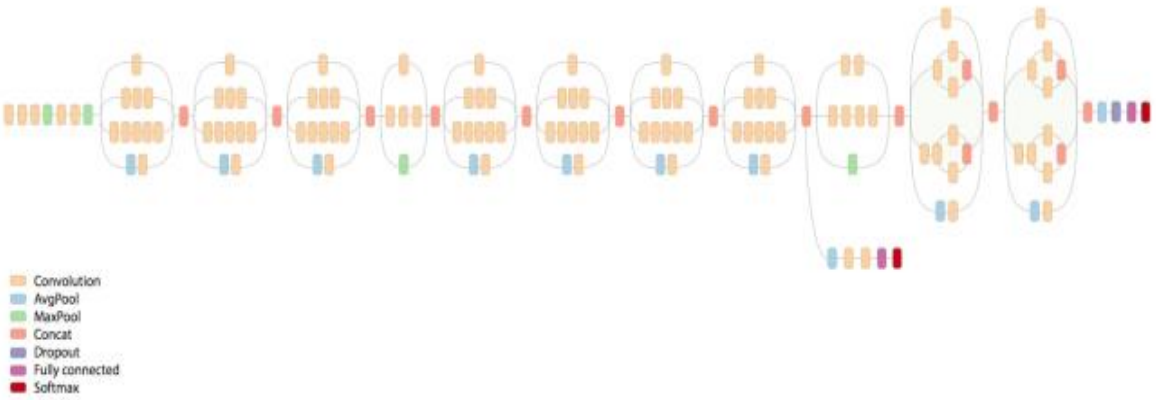


Figure 4.3 Inception v3 model

The configuration of each module in the inception v3 model is as follows.

Table 4.2 CNN 2 Configuration

Convolution 1x1 kernel and ReLU
Convolution 1x1 kernel with 3x3 kernel and ReLU
Convolution 1x1 kernel with 5x5 kernel and ReLU
Max pooling 3x3 kernel with convolution 1x1 kernel
Concatenation of all the above layers

The resulting features obtained after these convolutions are concatenated before passing on to the next module.The reason why inception modules are used is that they would overcome the need to specify the type of convolution we need at each layer.So multiple convolution filters are used in a module letting the neural network decide which gives better features.Extracting multiple features from multiple convolutions improves the performance of the network.This architecture also allows model to recover both local and abstract features of the input through smaller and larger convolutins respectively.

Coding this up in Tensorflow is more straightforward than elegant. For data, we'll use the classic MNIST dataset. We didn't use the typical formatted version, but found this neat site that has the csv version. The architecture of our model will include two Inception modules and one fully connected hidden layer before our output layer. We have to definitely want to use our GPU to run the code, or else it'll will take hours to days to train. If you don't have a GPU, you can check to see if your model works by using just a couple hundred training steps.

The libraries opencv and numpy are imported to preprocess the data and Tensorflow is used for the neural net.

The flow graph of the architecture of InceptionV3 as viewed on Tensorboard is shown in figure 4.4.

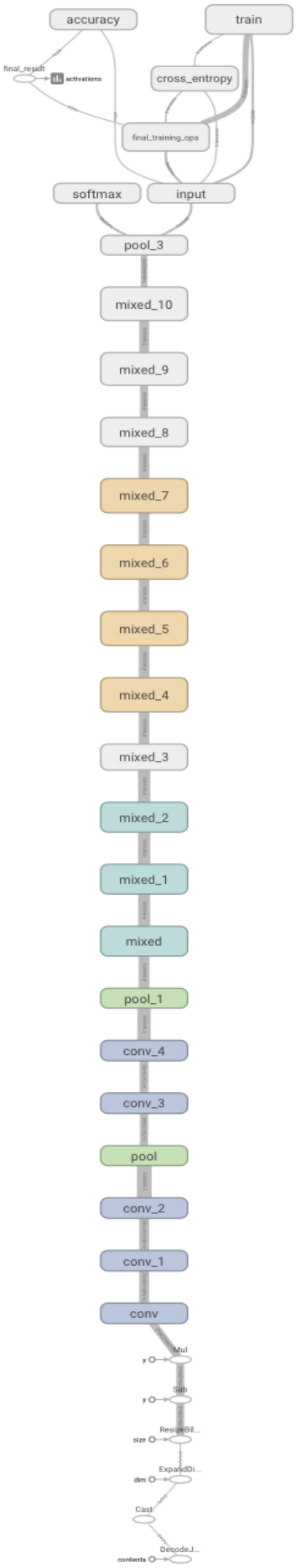


Figure 4.4 Model of inception v3

4.3 Tools used for training

4.3.1 Inception in Tensorflow

ImageNet is a common academic data set in machine learning for training an image recognition system. Code in this directory demonstrates how to use TensorFlow to train and evaluate a type of convolutional neural network (CNN) on this academic data set. Retraining an Inception v3 network is a novel task and back-propagating is done to find the errors and fine tune the network weights.

4.3.2 Tensorboard

The computations that are used in TensorFlow, for training a massive deep neural network can be complex and confusing. To make it easier to understand, debug, and optimize TensorFlow programs, a suite of visualization tools are included called TensorBoard. It can be used to visualize our TensorFlow graph, plot quantitative metrics about the execution of your graph, and show additional data like images that pass through it.

4.4 Training the System

We use Google's pre-trained Inception Convolutional Neural Network to perform image recognition without the need to build and train our own CNN. Inception V3 achieved such impressive results—rivaling or besting those of humans—by using a very deep architecture, incorporating inception modules, and training on 1.2 million images. If we want to classify different objects or perform slightly different image-related tasks, then we will need to train the parameters—connection weights and biases—of at least one layer of the network. The theory behind this approach is that the lower layers of the convolutional neural network are already very good at identifying lower-level features that differentiate images in general (such as shapes, colors, or textures), and only the top layers distinguish the specific, higher-level features of each class (lines between fingers of a human hand, etc.). Training the entire network on a reasonably sized new dataset is unfeasible on a personal laptop, but if we limit the size of the dataset and use a consumer-grade GPU, we can train the last layer of the network in a reasonable amount of time. We probably will not achieve record results on our task, but we can at least see the principles involved in adapting an existing model to a new dataset.

Selecting an appropriate dataset is the most important aspect of this project. The dataset needs to contain enough valid labeled images in each class to allow the neural network to learn every label. There is no magic threshold for number of image per class,

but more images, so long as they are correctly labeled, will always improve the performance of the model. Our dataset contains an average of 1000 images per class and a total number of 24 classes. We choose to use the images that had been processed through deep funneling, a technique of image alignment that seeks to reduce intra-class variability in order to allow the model to learn inter-class differences. Ideally, we want images belonging to the same hand gesture to be similarly aligned so the network learns to differentiate between images based on the hand in the image and not the particular orientation of the hand, lighting in the image, or background behind the hand. Deep funneling was shown to improve the performance of a neural network on image verification. Throughout this project, we have to know that the Inception network has been trained on 1.2 million images of objects and not a single human hand. Our task is to train a single layer of the network (out of more than 100 total) to differentiate between 24 different human hand gestures. We are adapting a CNN that has never previously seen a human hand to accomplish image identification.

4.4.1 Downloading the Latest Checkpoint of Pre-Trained Inception Model

We first need to make sure that we have the most up-to-date Inception V3model with the parameters learned through training on the ImageNet dataset. The following code will download the latest version and extract the required checkpoint:

```
# Trusty machine learning imports
import tensorflow as tf
import numpy as np
# Make sure to run notebook within slim folder
from datasets import dataset_utils
import os
# Base url
TF_MODELS_URL = "http://download.tensorflow.org/models/"
# Modify this path for a different CNN
INCEPTION_V3_URL = TF_MODELS_URL +
"inception_v3_2016_08_28.tar.gz"
# Directory to save model checkpoints
MODELS_DIR = "models/cnn"
INCEPTION_V3_CKPT_PATH = MODELS_DIR + "/inception_v3.ckpt"
# Make the model directory if it does not exist
if not tf.gfile.Exists(MODELS_DIR):
 tf.gfile.MakeDirs(MODELS_DIR)

# Download the appropriate model if haven't already done so
if not os.path.exists(INCEPTION_V3_CKPT_PATH):

dataset_utils.download_and_uncompress_tarball(INCEPTION_V3_U
RL, MODELS_DIR)
```

The dataset is structured with a top-level folder called ‘dataset’ that contains 24 folders labeled as ‘A’, ’B’, and so on, each with the images of a single hand gesture.

```
|
---- /dataset
|   |
|   |
|   ---- /A
|   |   A1.jpg
|   |   A2.jpg
|   |   ...
|   |
|   ---- /B
|       B1.jpg
|       B2.jpg
|       ...
|
```

We can now split the data into training, validation, and testing datasets with a few basic operations. We need a training set to allow the network to learn the classes, a validation set to implement early stopping when training, and a testing set to evaluate the performance of each model. We will put 80% of each class in a training set, 10% in a validation set, and 10% in a testing set.

4.4.2 Processing the Images

The ImageNet CNN requires that the images be provided as arrays in the shape [batch_size, image_height, image_width, color_channels]. Batch size is the number of images in a training or testing batch, the image size is 299 x 299 for Inception V3, and the number of color channels is 3 for Red-Green-Blue. In a given color channel, each specific x,y location denotes a pixel with a value between 0 and 255 representing the intensity of the particular color. ImageNet requires that these pixel values are normalized between 0 and 1 which simply means dividing the entire array by 255. Inception has built-in functions for processing an image to the right size and format, but we can also use scipy and numpy to accomplish the same task. All of the images in the dataset are 250 x 276 x 3 and these will be converted to 299 x 299 x 3 and normalized to pixel values between 0 and 1. The following code accomplishes this processing (which is applied during training):

```
from scipy.misc import imresize

# Function takes in an image array and returns the resized
and normalized array
def prepare_image(image, target_height=299,
target_width=299):
 image = imresize(image, (target_width, target_height))
 return image.astype(np.float32) / 255
```

4.4.3 Defining a layer of Inception CNN to train

In order to adapt the CNN to learn new classes, we must train at least one layer of the network and define a new output layer. We use the parameters that Inception V3 has learned from ImageNet for every layer except the last one before the predictions in the hope that whatever weights and biases are helpful in differentiating objects can also be applied to our facial recognition task. We therefore need to take a look at the structure of the network to determine the trainable layer. Inception has many layers and parameters (about 12 million), but we do not want to attempt to train them all.

The inception_v3 function available in the TensorFlow slim library (in the nets folder) returns the unscaled outputs, known as logits, as well as the endpoints, a dictionary with each key containing the outputs from a different layer. If we look at the endpoints dictionary, we can find the final layer before the predictions:

```
from nets import inception
from tensorflow.contrib import slim
tf.reset_default_graph()
X = tf.placeholder(tf.float32, [None, 299, 299, 3],
name='X')
is_training = tf.placeholder_with_default(False, [])
# Run inception function to determine endpoints
with slim.arg_scope(inception.inception_v3_arg_scope()):
 logits, end_points = inception.inception_v3(X,
num_classes=1001, is_training=is_training)
# Create saver of network before alterations
inception_saver = tf.train.Saver()
print(end_points)
output:
{'AuxLogits': <tf.Tensor
'InceptionV3/AuxLogits/SpatialSqueeze:0' shape=(?, 1001)
dtype=float32>,
 'Conv2d_1a_3x3': <tf.Tensor
'InceptionV3/InceptionV3/Conv2d_1a_3x3/ReLU:0' shape=(?,
149, 149, 32) dtype=float32>,
 'Conv2d_2a_3x3': <tf.Tensor
'InceptionV3/InceptionV3/Conv2d_2a_3x3/ReLU:0' shape=(?,
147, 147, 32) dtype=float32>,
 'Conv2d_2b_3x3': <tf.Tensor
'InceptionV3/InceptionV3/Conv2d_2b_3x3/ReLU:0' shape=(?,
147, 147, 64) dtype=float32>,
 'Conv2d_3b_1x1': <tf.Tensor
'InceptionV3/InceptionV3/Conv2d_3b_1x1/ReLU:0' shape=(?, 73,
73, 80) dtype=float32>,
 'Conv2d_4a_3x3': <tf.Tensor
'InceptionV3/InceptionV3/Conv2d_4a_3x3/ReLU:0' shape=(?, 71,
71, 192) dtype=float32>,
 'Logits': <tf.Tensor 'InceptionV3/Logits/SpatialSqueeze:0'
```

```
shape=(?, 1001) dtype=float32>,
 'MaxPool_3a_3x3': <tf.Tensor
'InceptionV3/InceptionV3/MaxPool_3a_3x3/MaxPool:0' shape=(?,
73, 73, 64) dtype=float32>,
 'MaxPool_5a_3x3': <tf.Tensor
'InceptionV3/InceptionV3/MaxPool_5a_3x3/MaxPool:0' shape=(?,
35, 35, 192) dtype=float32>,
 'Mixed_5b': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_5b/concat:0' shape=(?, 35,
35, 256) dtype=float32>,
 'Mixed_5c': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_5c/concat:0' shape=(?, 35,
35, 288) dtype=float32>,
 'Mixed_5d': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_5d/concat:0' shape=(?, 35,
35, 288) dtype=float32>,
 'Mixed_6a': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_6a/concat:0' shape=(?, 17,
17, 768) dtype=float32>,
 'Mixed_6b': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_6b/concat:0' shape=(?, 17,
17, 768) dtype=float32>,
 'Mixed_6c': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_6c/concat:0' shape=(?, 17,
17, 768) dtype=float32>,
 'Mixed_6d': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_6d/concat:0' shape=(?, 17,
17, 768) dtype=float32>,
 'Mixed_6e': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_6e/concat:0' shape=(?, 17,
17, 768) dtype=float32>,
 'Mixed_7a': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_7a/concat:0' shape=(?, 8, 8,
1280) dtype=float32>,
 'Mixed_7b': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_7b/concat:0' shape=(?, 8, 8,
2048) dtype=float32>,
 'Mixed_7c': <tf.Tensor
'InceptionV3/InceptionV3/Mixed_7c/concat:0' shape=(?, 8, 8,
2048) dtype=float32>,
 'PreLogits': <tf.Tensor
'InceptionV3/Logits/Dropout_1b/cond/Merge:0' shape=(?, 1, 1,
2048) dtype=float32>,
 'Predictions': <tf.Tensor
'InceptionV3/Predictions/Reshape_1:0' shape=(?, 1001)
dtype=float32>}
```

The ‘PreLogits’ layer is exactly what we are looking for. The size of this layer is [None, 1, 1, 2048] so there are 2048 filters each with size 1 x 1. This layer is created as a result of applying an average pooling operation with an 8 x 8 kernel to the Mixed layers and then applying dropout. We can change this to a fully connected layer by eliminating (squeezing) the two dimensions that are 1, leaving us with a fully connected layer with

2048 neurons. We then create a new output layer that takes the prelogits as input with the number of neurons corresponding to the number of classes. To train only a single layer, we specify the list of trainable variables in the training operation. What the network is learning during training is the connection weights between the 2048 neurons in the prelogits layer and the 10 output neurons as well as the bias for each of the output neurons. We also apply a softmax activation function to the logits returned by the network in order to calculate probabilities for each class:

```
# Isolate the trainable layer
prelogits = tf.squeeze(end_points['PreLogits'], axis=[1,2])
# Define the training layer and the new output layer
n_outputs = len(class_mapping)
with tf.name_scope("new_output_layer"):
 people_logits = tf.layers.dense(prelogits, n_outputs,
name="people_logits")
 probability = tf.nn.softmax(people_logits,
name='probability')
# Placeholder for labels
y = tf.placeholder(tf.int32, None)
# Loss function and training operation
# The training operation is passed the variables to train
which includes only the single layer
with tf.name_scope("train"):
 xentropy =
tf.nn.sparse_softmax_cross_entropy_with_logits(logits=people
_logits, labels=y)
 loss = tf.reduce_mean(xentropy)
 optimizer = tf.train.AdamOptimizer(learning_rate=0.01)
 # Single layer to be trained
 train_vars =
tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
scope="people_logits")
 # The variables to train are passed to the training
operation
 training_op = optimizer.minimize(loss, var_list=train_vars)

# Accuracy for network evaluation
with tf.name_scope("eval"):
 correct = tf.nn.in_top_k(predictions=people_logits,
targets=y, k=1)
 accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

# Intialization function and saver
with tf.name_scope("init_and_saver"):
 init = tf.global_variables_initializer()
 saver = tf.train.Saver()
```

After isolating the single layer to train, the rest of the code is fairly straightforward and typical for a neural network used for classifcation. We use average cross entropy as a loss function and use the Adam Optimization function with a learning rate of 0.01. The

accuracy function will measure top-1 accuracy. Finally, we use an initializer and a saver so that we will be able to save the model during training and restore it at a later time.

4.4.4 TensorBoard Visualization Operations

TensorBoard is an extremely vital tool for visualizing the structure of the network, the training curves, the images passed to the network, the weights and biases evolution over training, and a number of other features of the network. This information can guide training and optimization of the network. We will stick to relatively basic features of TensorBoard such as looking at the structure of the network, and recording the training accuracy, validation accuracy, and validation loss. We will use the date and time at the start of each training run to create a new TensorBoard log file:

```
with tf.name_scope("tensorboard_writing"):
 # Track validation accuracy and loss and training accuracy
 valid_acc_summary = tf.summary.scalar(name='accuracy_1',
tensor=accuracy)
 valid_loss_summary =
tf.summary.scalar(name='cross_entropy_1', tensor=loss)
# train_acc_summary = tf.summary.scalar(name='train_acc',
tensor=accuracy)
# Merge the validation stats
 valid_merged_summary =
tf.summary.merge(inputs=[valid_acc_summary,
valid_loss_summary])
# Use the time to differentiate the different training
sessions
from datetime import datetime
import time
# Specify the directory for the FileWriter
now = datetime.now().strftime("%Y%m%d_%H%M%S")
model_dir = "{}_unaugmented".format(now)
logdir = "tensorboard/faces/" + model_dir
file_writer = tf.summary.FileWriter(logdir=logdir,
graph=tf.get_default_graph())
```

When we want to look at the statistics, we can run TensorBoard from Windows Powershell (or the command prompt) by navigating to the directory and typing:

```
C:\Users\sumukha\Desktop\CNN>
tensorboard --logdir= /path/to/logs
```

4.4.5 Training

Finally, we are ready to train the neural network (or at least one layer) for our facial recognition task. We will train using early stopping, which is one method for

reducing overfitting on the training set (having too high of a variance). Early stopping requires periodically testing the network on a validation set to assess the score on the cost function (in this case average cross entropy). If the loss does not decrease for a specified number of epochs, training is halted. In order to retain the optimal model, each time the loss improves, we save that model. Then, at the very end of training, we can restore the model that achieved the best loss on the validation set. Without early stopping, the model continues to learn the training set better with each epoch but at the cost of generalization to new instances. There are many implementations of early stopping, but we will use a single validation set and stop training if the loss does not improve for 20 epochs.

Storage of feature in the form of bootlenecks are shown below in figure 4.5.

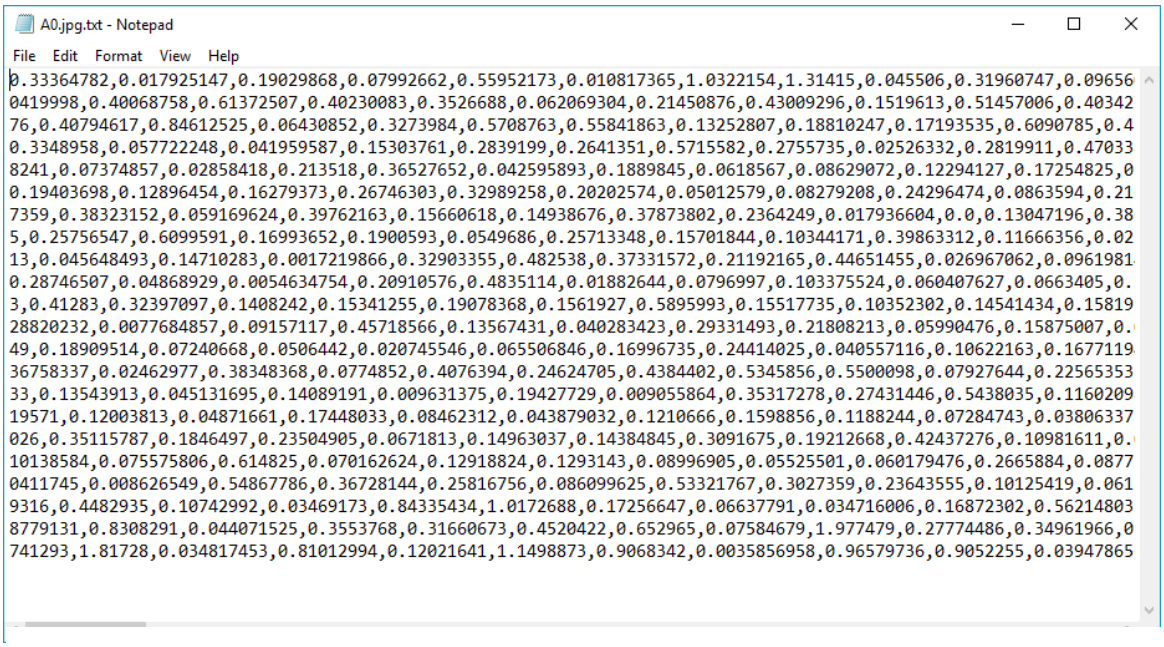


Figure 4.5 Storage of an image in the dataset in the form of bottleneck

4.5 TensorBoard Training Curves

Finally, if we want to analyze our CNN performance in even greater depth, we can look at the training accuracy, validation loss, and validation accuracy as a function of the epoch for training on the various datasets. This will allow us to see the extent of overfitting (or maybe underfitting) on the training set as well as the effect of data augmentation on the training performance. We kept the statistics relatively simple and did not take advantage of the capabilities of TensorBoard, such as visualizing histograms that show the evolution of weights and biases over the epochs. The full features of TensorBoard (including embedded visualizations) are incredible and are a necessity to use when developing a full CNN or any neural network. Various types of graphs can be

obtained using tensorboard from which we use scalar and histograms. Some graphs which we obvtained are shown in figure 4.6.

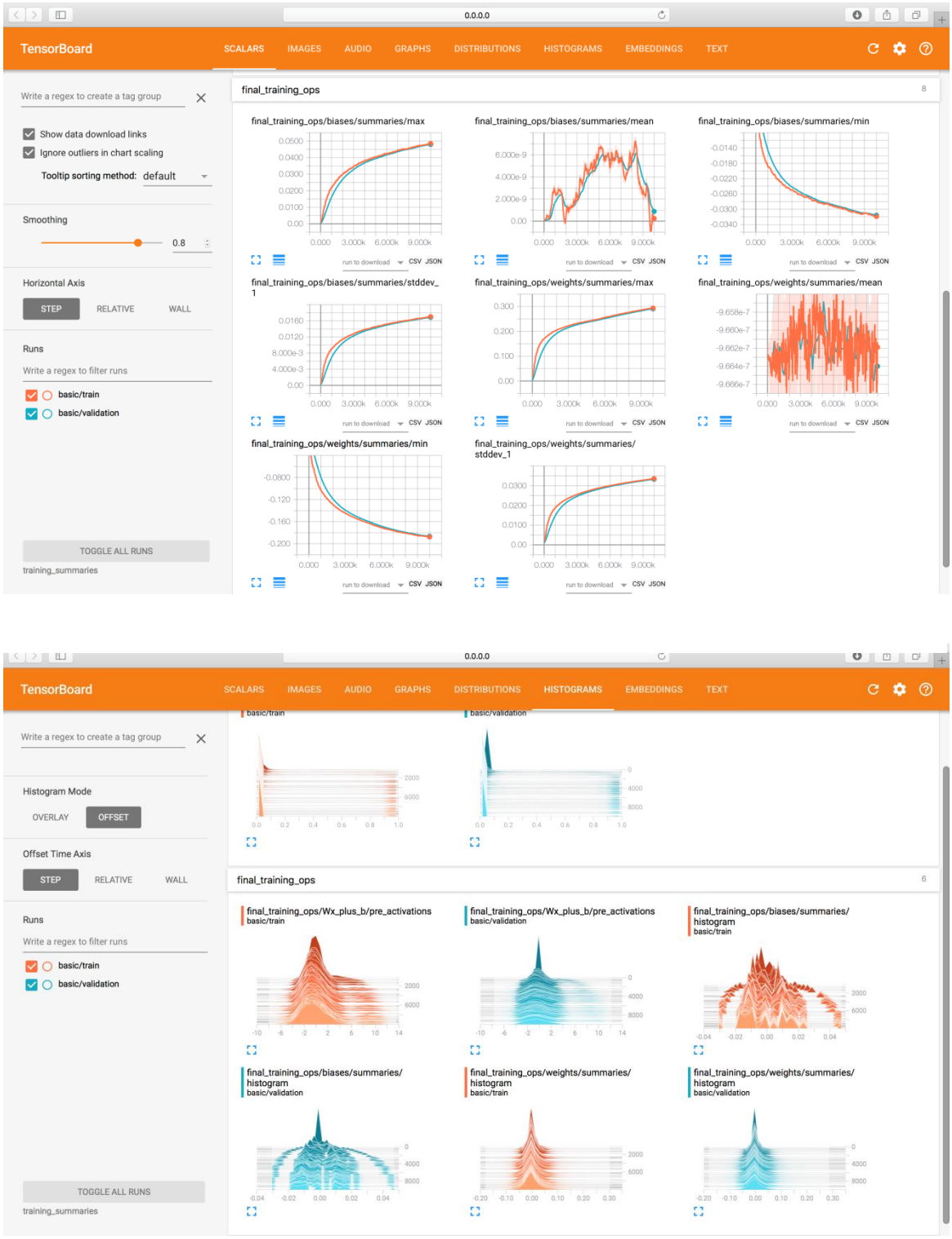


Figure 4.6 Various training result graphs and histograms being displayed in tensorboard

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Comparison of LeNet-5 and InceptionV3

Training is done for 6000 steps, number of epochs 10 and a batch size of 1000. After the training, the following results were obtained for both architectures.

5.1.1 CNN 1(LeNet-5)

Fig 5.1 shows that the accuracy percentage obtained for the inception v3 architecture to be 90.25% With respect to the error function, its evolution is shown based on a batch size of 100 for 6000 steps and 10 epochs. Figure 4.2 shows that the cost function value is found to be 0.39.

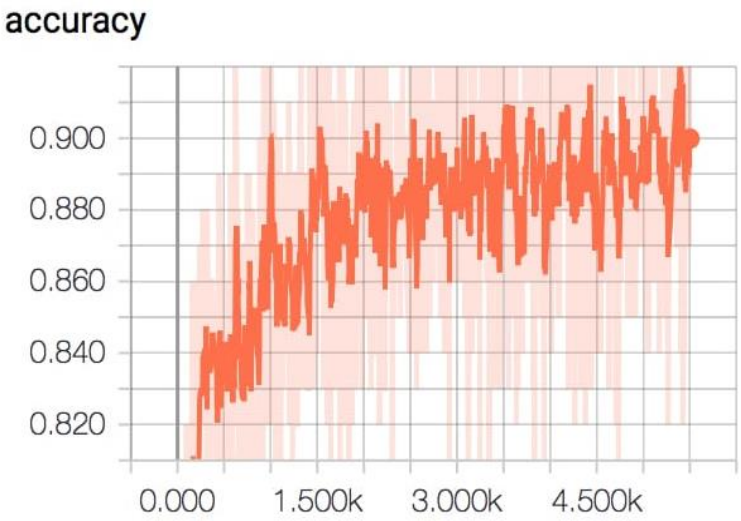


Figure 5.1 accuracy graph of CNN 1

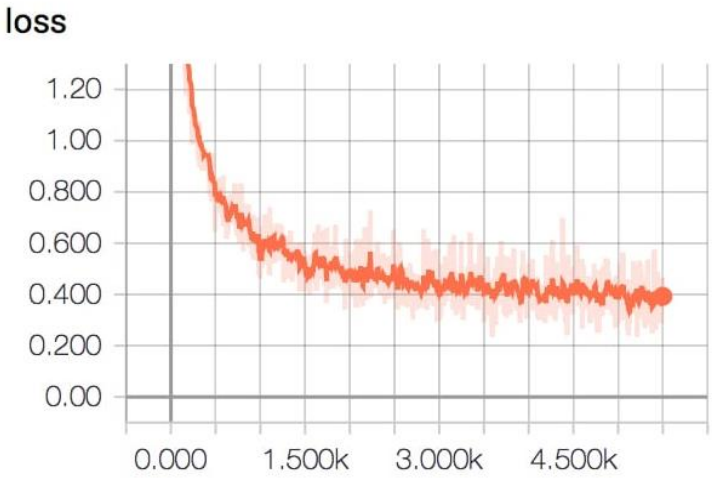


Figure 5.2 loss graph of CNN 1

5.1.2 CNN 2 (InceptionV3)

Fig 5.3 shows that the accuracy percentage obtained for the inception v3 architecture to be 93.6% With respect to the error function, its evolution is shown based on a batch size of 100 for 6000 steps and 10 epochs. Figure 4.4 shows that the cost function value is found to be 0.26.

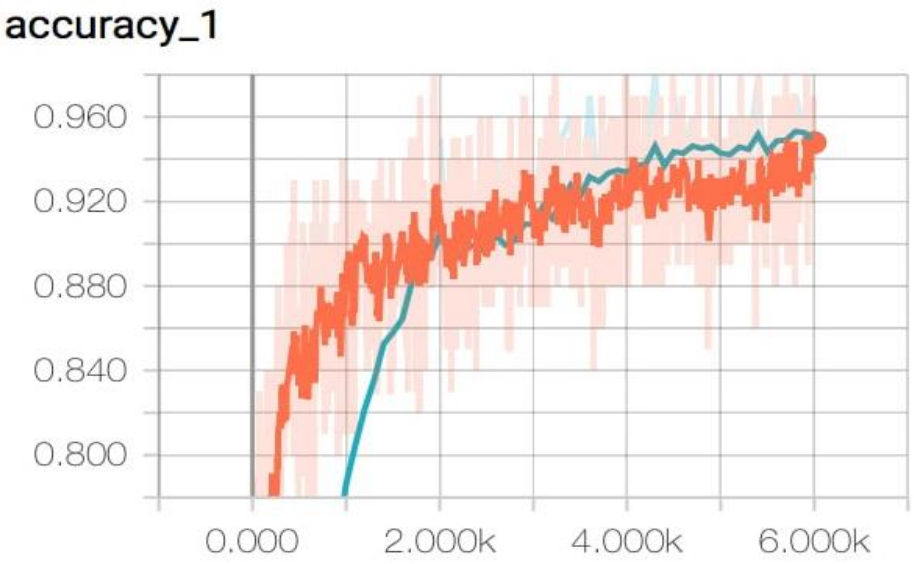


Figure 5.3 accuracy plot of CNN 2

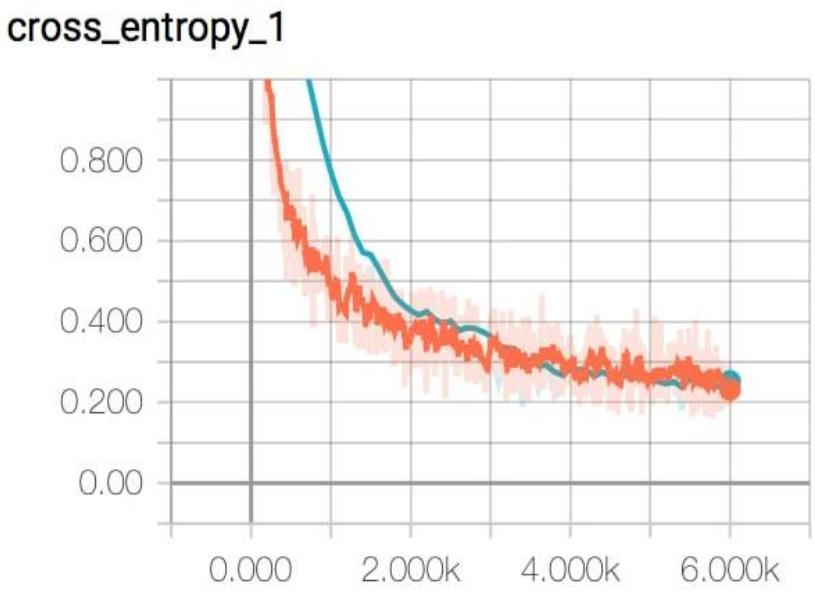


Figure 5.4 cross entropy(loss) plot of CNN 2

5.2 Real-Time classification

Training is done for 10000 steps, number of epochs 25 and a batch size of 100.After the training, the accuracy was found to be 99.6% and the loss is found to be 0.04 . With this trained model, classification is performed in real time.

```
Step: 9700, Train accuracy: 99.0000%, Cross entropy: 0.059578, Validation accuracy: 100.0% (N=100)
Step: 9800, Train accuracy: 100.0000%, Cross entropy: 0.039000, Validation accuracy: 100.0% (N=100)
Step: 9900, Train accuracy: 99.0000%, Cross entropy: 0.032260, Validation accuracy: 100.0% (N=100)
Step: 9999, Train accuracy: 100.0000%, Cross entropy: 0.044197, Validation accuracy: 100.0% (N=100)
Final test accuracy = 99.6% (N=2340)
Converted 2 variables to const ops.
```

Figure 5.5 Accuracy obtained for real time classification

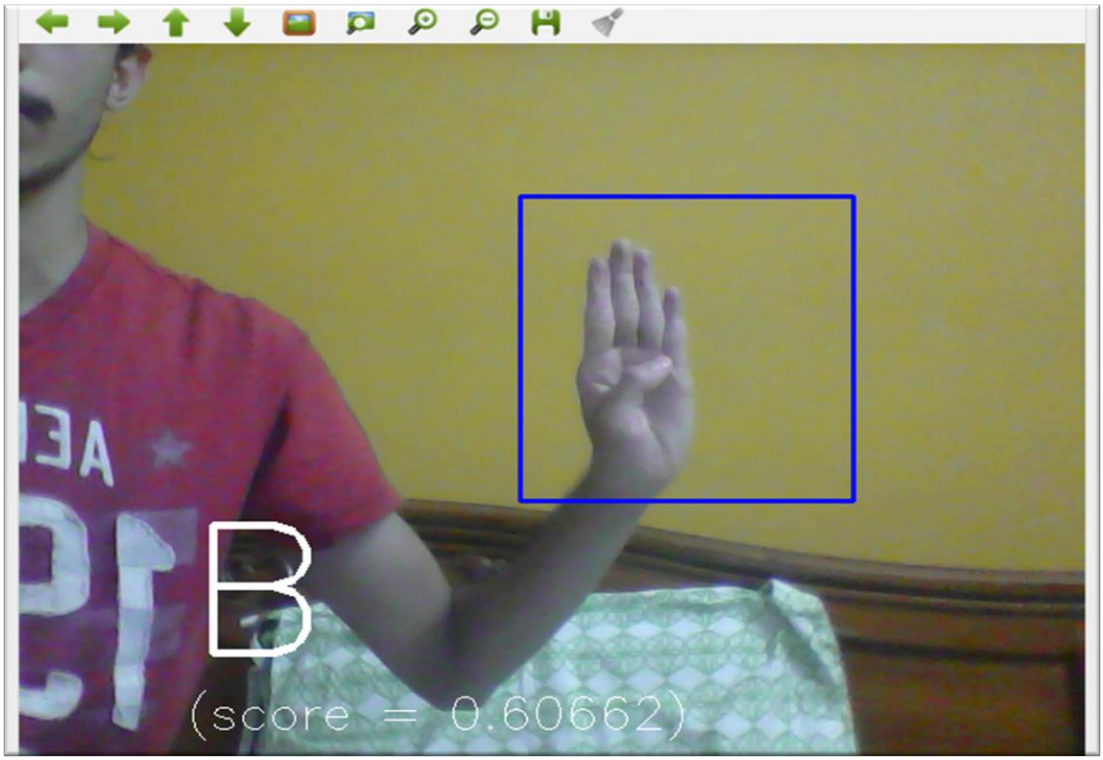


Figure 5.6 Real time prediction of the hand gesture being displayed

The rectangle specifies the region where the hand is placed .The system predicts the letter being displayed and the result shown is the prediction done by the trained model. The score represents the accuracy of the prediction done by the system.

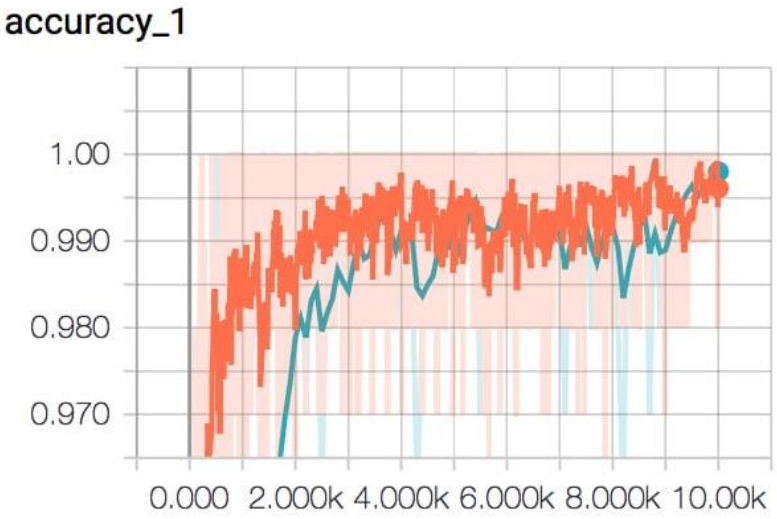


Figure 5.7 Accuracy plot for real time prediction

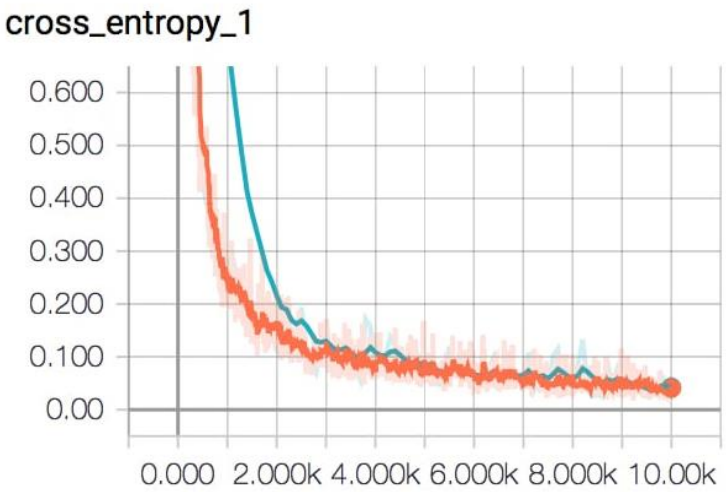


Figure 5.8 Cross entropy plot for real time prediction

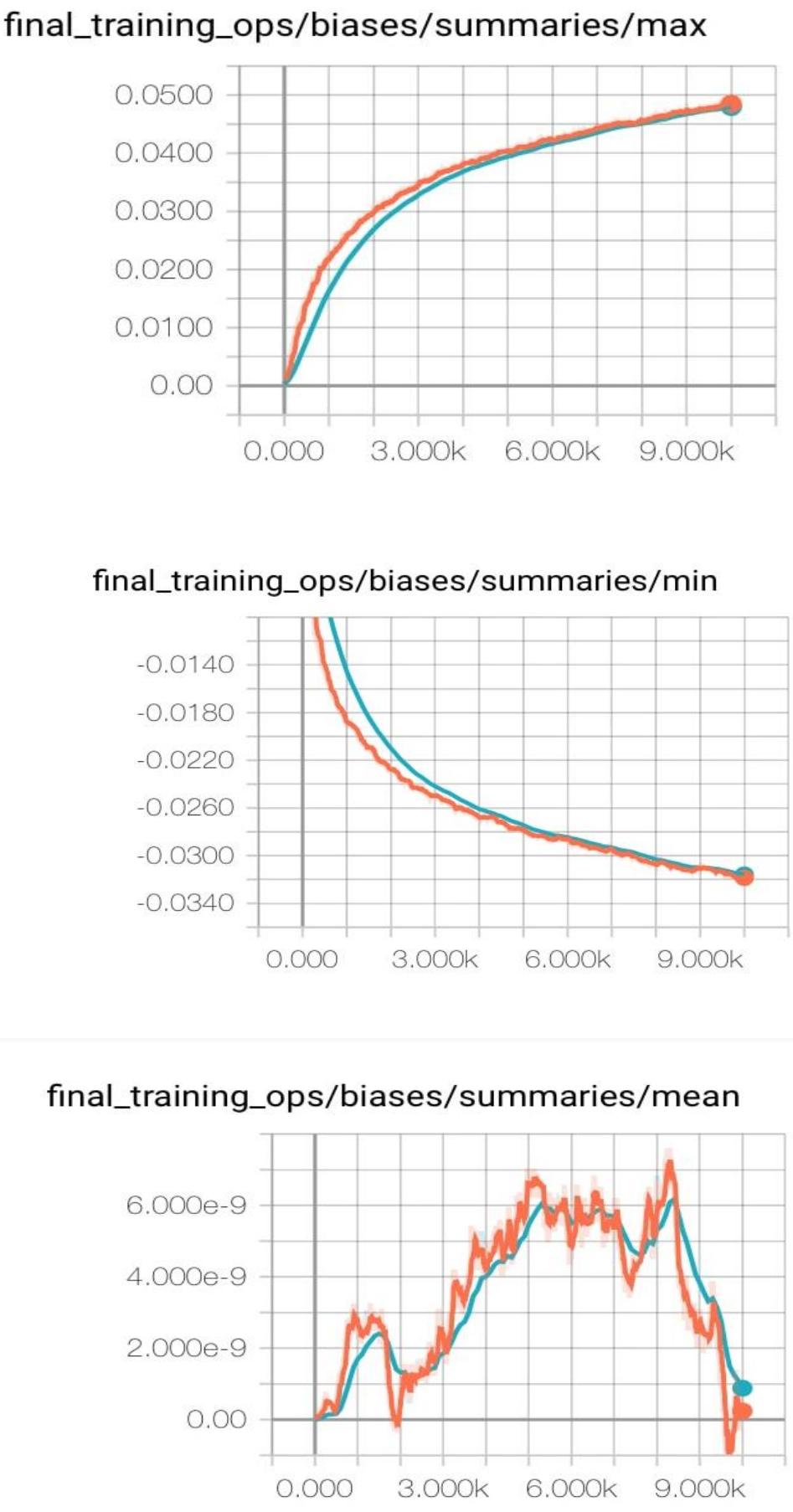


Figure 5.9 Plot of bias terms

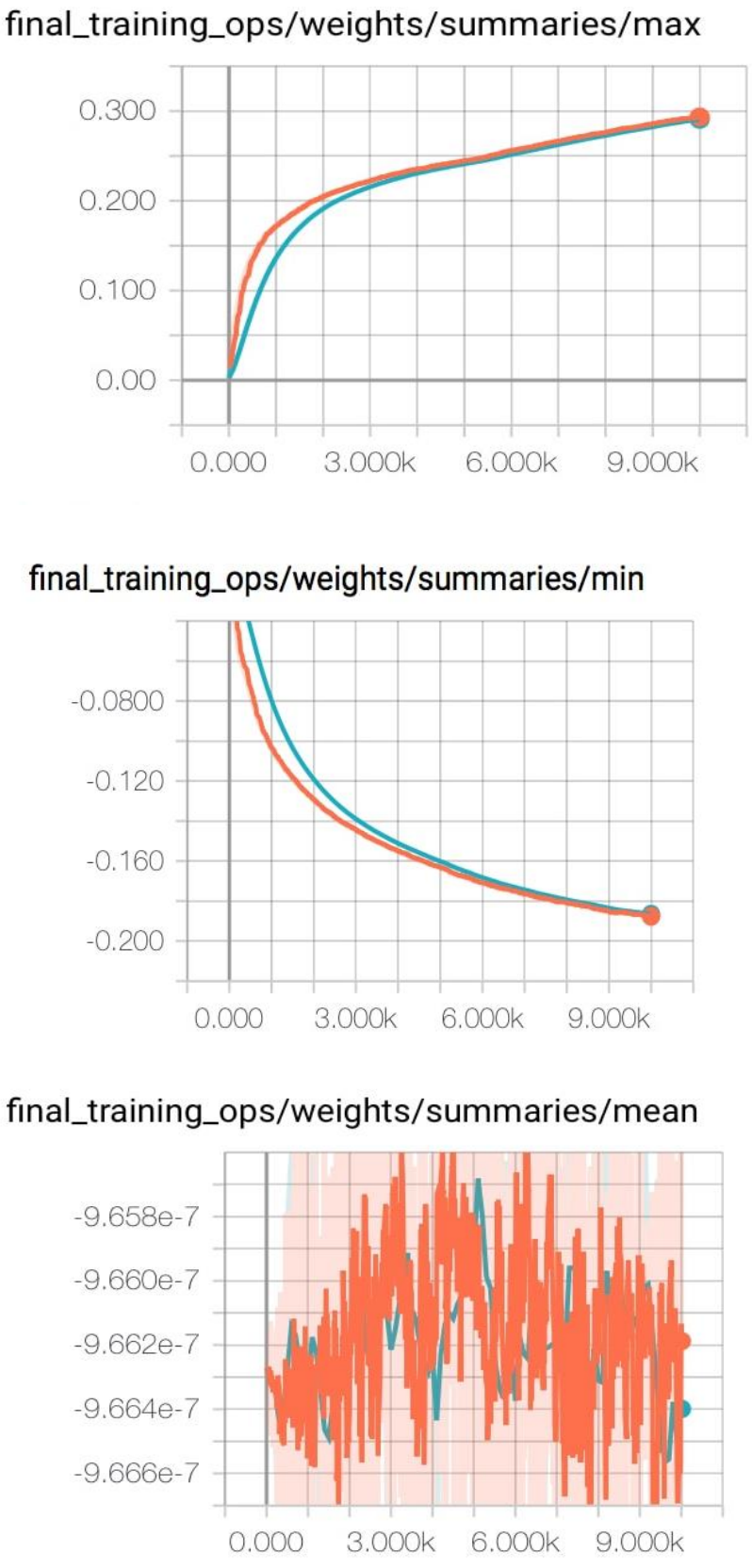


Figure 5.10 Plot of weight terms

CHAPTER 6

FUTURE WORKS AND CONCLUSION

6.1 Future works

This can also be implemented as a mobile application for recognition of the hand gestures. Developments can be made such that the device is compatible for many sign languages and not just ASL and can recognize symbols from any of them .The system can be fine tuned and altered so that a particular hand gesture is specified for most commonly used words for improving the ease of communication and also making the process faster. This model can be implemented on a portable device (like a spectacle or a cap) where Bluetooth cameras can be made use of and a mobile application can be used to monitor and control the process of start and stop of the gesture recognition which will be useful for the visually and speech impaired for communication.

6.2 Conclusion

There is a difficulty in finding systems that recognize hand gestures with great precision for different types of lighting, noise and complex backgrounds, as well as invariations of scale, rotation and translation. However, CNNs were used as a solution to improve the rate of recognition accuracy for these types of invariance. It was proposed to develop two convolutional networks, each with a different number of layers, depth and number of parameters per layer. Comparing the results of precision and error obtained, networks with greater depth or number of hidden layers have greater recognition accuracy compared to networks with fewer hidden layers. The CNNs show better results in comparison to the usual techniques of machine learning. A good use of digital image processing techniques will help to detect a better the region that contains the hand gesture, minimizing the error caused by complex image background and varied illumination. Complex background images obtain low accuracy compared to other invariations, skin color based segmentation does not provide optimum separation compared to detection techniques. However this segmentation is most commonly used for its simplicity to separate interest object from complex background.

BIBLIOGRAPHY

[1] Jose L. Flores C.,E. Gladys Cutipa A.,Lauro Enciso R., "Application of Convolutional Neural Networks for Static Hand Gestures Recognition Under Different Invariant Features,"

[2] R. Y. Wang and J. Popovic,´ "Real-time hand-tracking with a color glove," in ACM SIGGRAPH 2009 Papers, ser. SIGGRAPH '09. New York, NY, USA: ACM, 2009, pp. 63:1–63:8. [Online]. Available: http://doi.acm.org/10.1145/1576246.1531369

[3] N. Y. Y. Kevin, S. Ranganath, and D. Ghosh, "Trajectory modeling in gesture recognition using cyberglovesreg; and magnetic trackers," in 2004 IEEE Region 10 Conference TENCON 2004., vol. A, Nov 2004,pp. 571–574 Vol. 1.

[4] L. Yun and Z. Peng, "An automatic hand gesture recognition system based on viola-jones method and svms," in 2009 Second International Workshop on Computer Science and Engineering, vol. 2, Oct 2009, pp. 72–76.

[5] T. N. T. Huong, T. V. Huu, T. L. Xuan, and S. V. Van, "Static hand gesture recognition for vietnamese sign language (vsl) using principle components analysis," in 2015 International Conference on Communications, Management and Telecommunications (ComManTel), Dec 2015, pp. 138–141.

[6] M. A. Aowal, A. S. Zaman, S. M. M. Rahman, and D. Hatzinakos, "Static hand gesture recognition using discriminative 2d zernikemo-ments," in TENCON 2014 - 2014 IEEE Region 10 Conference, Oct 2014, pp. 1–5.

[7] M. A. Amin and H. Yan, "Sign language finger alphabet recognition from gabor-pca representation of hand gestures," in 2007 International Conference on Machine Learning and Cybernetics, vol. 4, Aug 2007,pp. 2218–2223.

[8] K. p. Feng and F. Yuan, "Static hand gesture recognition based on hog characters and support vector machines," in 2013 2nd International Symposium on Instrumentation and Measurement, Sensor Network and Automation (IMSNA), Dec 2013, pp. 936–938.

[9] Y. Ding, H. Pang, X. Wu, and J. Lan, "Recognition of hand-gestures using improved local binary pattern," in 2011 International Conference on Multimedia Technology, July 2011, pp. 3171–3174.

[10] M. Murugeswari and S. Veluchamy, “Hand gesture recognition system for real-time application,” in 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies, May 2014, pp. 1220–1225.

[11] H. M. Gamal, H. M. Abdul-Kader, and E. A. Sallam, “Hand gesture recognition using fourier descriptors,” in 2013 8th International Confer-ence on Computer Engineering Systems (ICCES), Nov 2013, pp. 274– 279.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[13] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, and G. Wang, “Recent advances in convolutional neural networks,” CoRR, vol. abs/1512.07108, 2015. [Online]. Available: http://arxiv.org/abs/1512.07108

[14] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, “Face recognition: a convolutional neural-network approach,” IEEE Transactions on Neural Networks, vol. 8, no. 1, pp. 98–113, Jan 1997.

[15] K. B. Shaik, G. P., V. Kalist, B. S. Sathish, and J. M. M. Jenitha, “Comparative study of skin color detection and segmentation in hsv and ycbcr color space,” in 3rd International Conference on Recent Trends in Computing 2015, Jul 2015, pp. 41–48.

Online References

- https://github.com/tensorflow/models/tree/master/research/inception
- https://www.tensorflow.org/programmers_guide/graph_viz
- https://github.com/tensorflow/models
- http://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/
- https://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/