Facial recognition using Convolutional Neural Network

By: Kaustubh Chaudhuri
California State University, Chico

Guided by: Dr Choi Sukgi
California State University, Chico

Abstract

Facial recognition is improving, and with the emergence of deep learning, facial learning algorithms have become more robust and accurate without the need of hard coding. In this paper we start designing and implementing a Convolutional Neural Network(CNN) for facial recognition and observe what factors affect the accuracy of the predictions. Given a fixed input space, and with just three sets of layers (conv 2D, max pooling, dropout) and a dense layer our neural network is able to recognize faces with an accuracy of over 80%.

1. Introduction

With the era of new technologies new kinds of fields have come into existence. This has paved the way for better solutions with improved results. Living beings are excellent in recognizing objects, patterns, locations and images. Human beings are excellent examples and role models for image recognition. Earlier we had different kinds of image recognition algorithms, it all started with simple image processing strategies but slowly evolved with better results. Today we have come to the age of artificial intelligence whose main inspiration is the brain of a human being.

Artificial intelligence has gifted us with machine learning; a new kind of strategy and science to solve problems with a human touch. It has shown promising results in predicting patterns, recognize objects, interpret voice and synthesize speech. Using the similar pattern of the function of our brain we can mimic the way to approach an image recognition problem, this is where machine learning comes in.

In the paper we have described an algorithm in detail which recognizes human faces with high accuracy using Convolutional Neural Network (CNN). CNN is an evolving architecture of deep learning which is in turn one of the most advanced subject of machine learning. In this project we have used the facial recognition algorithm to recognize five individuals specifically. In the first part of the project we collect the data by taking multiple pictures of the individual's face and then proceed to preprocessing the image files into 4-dimensional tensors (data structures). Then in the next part we construct an CNN by using the keras library, further supplying it with the tensors for training. Lastly, we test the whole algorithm by supplying preprocessed data. The algorithm showed promising results by recognizing the individuals with an accuracy of 81%.

We also discuss the process of optimizing the CNN and the factors involving in it. We end the paper by analyzing the results and the accuracy of the CNN.

2. Facial Recognition in existing models

There are several facial recognition algorithms that recognize faces using different methods like determining facial features by extracting landmarks, or features such as particular patterns of eyes, and eyebrows, from an individual’s face in the image. Some find similarity in the shapes and the structure of the face to identify the person. Some try approaching the recognition problem by recording the ratio of the distances between each of the features in the face for example the distance between the eyes and the nose is unique for every person, this technique was used initially in the facial recognition software. Sometimes each of the features are recorded such as the eyes, nose, lips, cheekbones or hair and are used to search in the other images for matching features to identify the subject. Some techniques include normalizing a set of facial images of a particular individual to form a template which is used in future searches. This technique showed quite promising results for a long time. By averaging the facial images of a person, the resultant compressed image looks like a ghost image of the person.

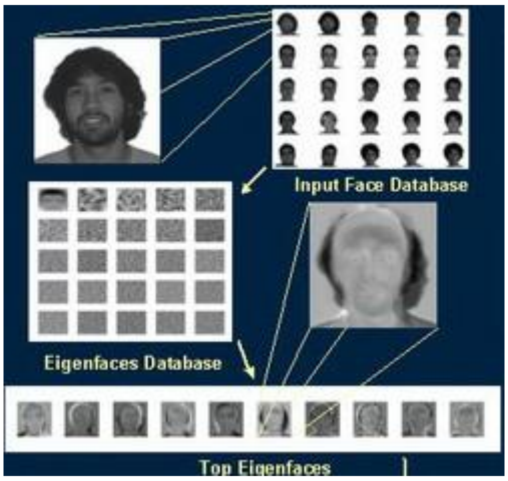


Fig 1: Example of eigen Faces (Ghost images of faces) used for facial detection

Recognition algorithms can be divided into two main approaches, geometric (Using the geometric structure of the facial features), which looks at distinguishing features, or photometric (image is converted to numerical values like the pixel values, here geometric structure is irrelevant), which is a statistical approach that distills an image into values and compares the values with templates to eliminate variances.

Popular recognition algorithms include principal component analysis using eigen faces, linear discriminant analysis, elastic bunch graph matching using the Fisherface algorithm, the hidden Markov model, the multilinear subspace learning using tensor representation, and the neuronal motivated dynamic link matching.

3. Convolutional Neural Network – A deep learning technique

Machine Learning(ML) algorithms solely work on the data it is provided with. Hence the more data an ML algorithm is provided, the better results it generates. Thus, with new technologies and sensors these algorithms have numerous kinds of data (training set) to learn from. This results with more accurate predictions than the prior models as discussed in the section 2. Since Machine Learning is not about explicit programming and modeling, scalability is not a weakness but a strength.

Machine learning has been evolving every year right from the simplest form, a decision tree to deep learning. Deep learning has become a new field by itself today and can be defined as the study of artificial neural networks (also known as neural networks) used for learning purposes.

Neural Network has been inspired from the structure of neurons in our brain. As our brain is made up of neurons, a neural network is made up of basic elements known as a perceptron.

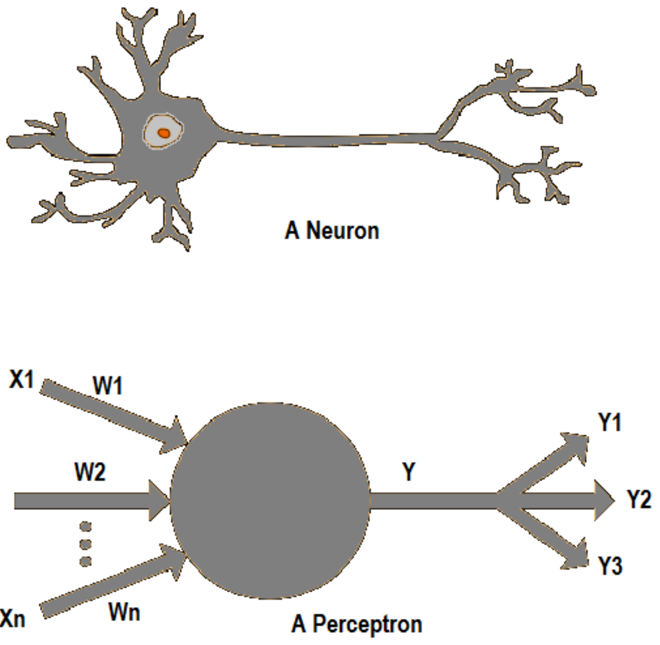


Fig 2: a biological neuron*(Top)*, A perceptron or computational neuron*(Bottom)*

Generally, a perceptron as shown above performs a linear function with respect to the inputs X1, X2 to Xn and its weights W1, W2 to Wn as coefficients. *f(*X*)* can be any linear function, in our project we use 'Max Pooling function' as the linear function (discussed later). The result obtained is Y.

$$W1X1 + W2X2 + W3X3 + ... + WnXn = Y$$

Or

$$f(X) = Y$$

This result Y can be divided into various classes of results like Y1, Y2 and Y3 according to the activation function being used for the perceptron. Then a collection of these perceptrons in each level form a layer in the neural network as shown below:

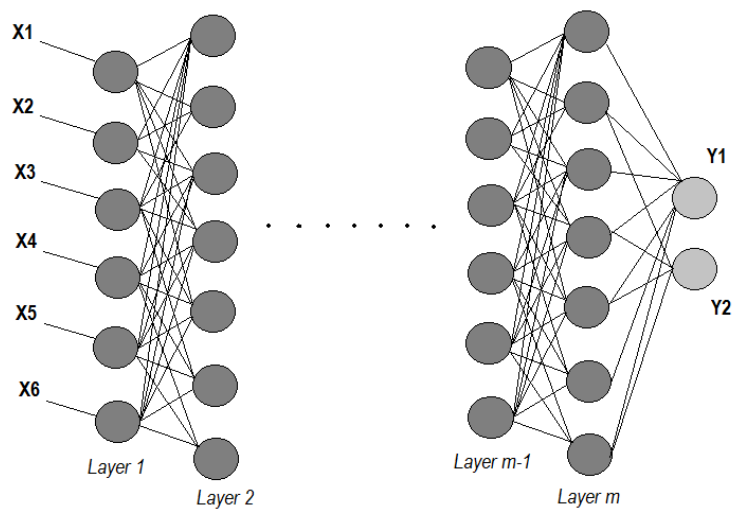


Fig 3: Layers in a neural network

In the above figure we can see that grouping the perceptrons in each level we form different layers in the neural network right from layer *1* to layer *m.*

Now with this necessary background of the structure of a neural network we proceed to Convolutional Neural Networks. As explained in [2], Convolutional Neural Networks are a form of a Neural Networks that have mostly convolutional layers and a few densely connected layers. Each neuron(perceptron) 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. Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

In CNN architectures we make the explicit assumption that the inputs are images in the form of tensors, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters (weights) in the network.

A simple Neural Networks receive an input (a one dimensional array), 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. If the images were 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 = 3072 weights(connections). 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, 224x224x3, would lead to neurons that have 224*224*3 = 150,528 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 CNN 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 used are an input volume of activations, and the volume has dimensions 224x224x3 (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 have dimensions 1x1x5 (5 since we are recognizing one of the five individuals), because by the end of the CNN architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.

4. Requirements

To build a neural network we need a substantially powerful computer so that the system can support the preprocessing and the training of the neural network. Hence a computer system should have a RAM of at least 8 Gb and a powerful GPU. When there is no access to such a system then an alternative way to house a neural network would be on a GPU instance of a cloud, like he Google Cloud or the Amazon Web Services. This project was carried out with the following system configurations.

System - Google Cloud computing instance 32 Gb RAM

Operating System – Ubuntu 16.04

Language - Python 3.6 or Python 2.7

Libraries - numpy, scipy, pandas, tensorflow, keras, cv2, PIL, tqdm, glob

Integrated Development Environment – Jupyter

We use python as the first choice of language since this scripting language is perfect for prototyping and machine learning. The system is installed with the above libraries of Python.

Each of the python libraries give the following functionality

- numpy – This library is a basic library used for scientific computations, just like MATLAB. With the help of this library we can perform complicated computations using matrices. Since the images are converted into matrices, this library becomes an important part of the project.

- scipy – This is another library used for complicated calculations. This library takes the support of numpy library and is used for higher level of computations.

- pandas – This library is used for data science. It is a very commonly used library for fabricating, extracting, converting and analyzing data. This library makes it easy to work with data in tabular form.

- tensorflow – This library is an open source machine learning library. It is a widely used library for neural networks and has been improving every year since the past few years. We do not use the library directly but use this library in the backend of the keras library.

- Keras – This library is mainly used for deep learning. It has inbuilt functions to model, build and train a neural network. This library is based on tensorflow and hence uses 'tensor' data structure as the basic form of data structures.

- cv2 – Is the latest library of OpenCV 3.3.0. This library gives us the capability to process images. It has a few inbuilt face detection algorithms as well which we use in the project.

- PIL – Is an image processing library for python, usually used to converts image files from one format to another.

- tqdm – is a library that supports visual aided meter for showing the progress bar.

- glob - The glob module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell.

5. Dataset – What does it contain?

The dataset is made up of multiple photographs of individuals with as many variations as possible. This is done to make the machine learning algorithm more efficient and robust. The more variation in the dataset the better the CNN will perform. Hence a dataset is always good when there are more samples which covers all the possible variations for a required prediction.

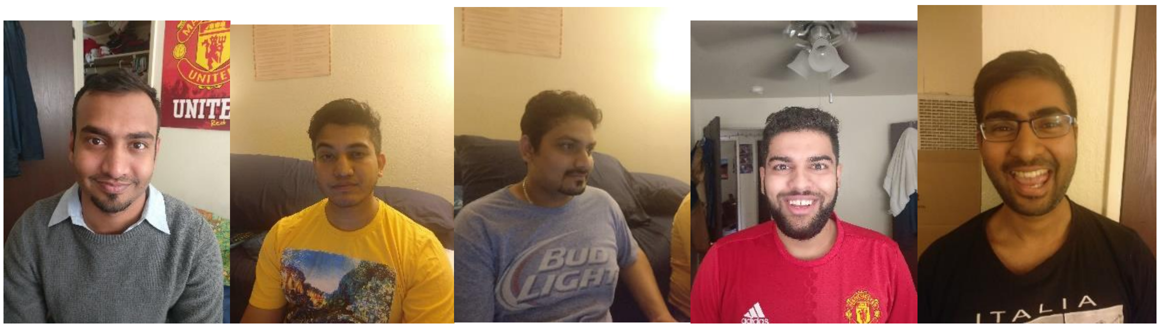


Fig 4: Sample images of the five individuals

The dataset is divided into three sets:

I. Training Set
II. Test Set
III. Validation Set

Each of these sets contain five categories(classes) each, namely my five friends; *'Ankit'*, *'Deepansh'*, *'Hitesh'*, *'Omkar'* and *'Tanay'*. Since my output space has 5 possible individuals we use five classes for each of the sets (training, testing and validation)

There are in total 291 images, the training set contains 224 images, the validation set contains 45 images and the test set contains 22 images. All of these images are segregated into three kinds of sets (training, validation and testing sets) **randomly**. The dataset is split into these three sets because:

Training Set: this set constitutes the major part of the dataset (usually 75% - 80% of the total size of the dataset), it is used to train the neural network. By training the neural network the weights are set and the losses are calculated so that the CNN can produce better predictions.

Validation Set: this data set is used to minimize overfitting. Here we are not adjusting the weights of the network with this data set, we are just verifying that any increase in accuracy over the training data set actually yields an increase in accuracy over a data set that has not been shown to the network before, or at least the network hasn't trained on it (i.e. validation data set). If the accuracy over the training data set increases, but the accuracy over then validation data set stays the same or decreases, then we are overfitting our CNN and should stop training.

Testing Set: this data set is used only for testing the final dataset in order to confirm the actual predictive power of the network. After all the tweaking of the hyperparameters in the CNN and obtaining the best architecture of the CNN, we reveal this dataset to measure the accuracy of the CNN.

6. Pre-processing the data

The preprocessing is divided into two stages, the first stage includes is facial detection and extraction, the second stage involves conversion of the images into a 4D tensor.

Stage 1 – Detect and Extract

In the first stage the we detect the face of the person and then crop it. This is important to increase the efficiency of the CNN. The CNN should be provided only with relevant information. Any background in the picture can be considered relevant and be a major part of the training process. Hence, we use Haar Cascade facial detection algorithm to detect any faces in the picture. And for simplicity we have used the facial detection algorithm to detect and choose only one face in the images. A later improvement can be made for multiple face detection and extraction for the CNN from one image.

Haar Cascade Facial Detection algorithm was created by Paul Viola and Michael Jones in their paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. Their algorithm uses a machine learning model which has been trained with several pictures. OpenCV 3.3.0 (cv2) provides a function which allows us to use the Haar Cascade model without going into the details.

The face extraction algorithm

From the OpenCV library (cv2) we use the Haar Cascade algorithm by downloading it into a folder using and creating an object known as 'face_cascade', this is achieved with the following command:

```
face_cascade = 
cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml'
)
```

As ‘face_cascade’ object is created, it gives us access to all the functions associated with it, hence we use ‘detectMultiScale()’ function to detect the face in an image. The function takes in the image in matrix form as an argument and returns a list of values for each of the detected face in the image. As specified before for simplicity we just choose one face from one image. Thus, for each detected face the returned list of values includes the x coordinate, y coordinate, height and the width of the box which works as a kernel around the face.

Taking the x, y, height and width of the box, we know exactly where the face is located and hence just use this part of the image for training the CNN and thus forward these values to the second stage of the preprocessing step. If there is no face detected in the image then value zero is forwarded for the corresponding x, y, height and width to the next stage.

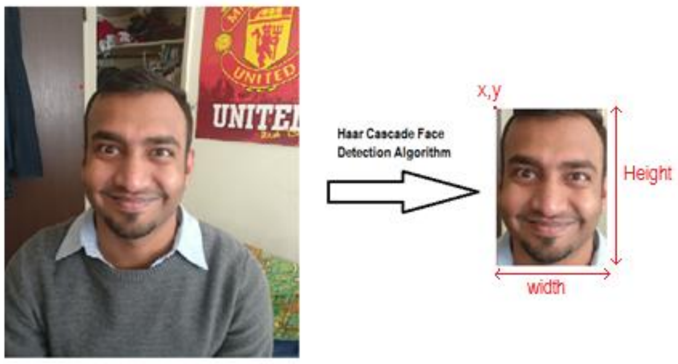


Fig 5: Extracting the face using Haar Cascade

Stage 2 – Convert an image to a 4D Tensor (data structure for CNN)

The image is of an individual face can be of any size so before doing anything, we need to convert the image into a fixed dimension since the CNN cannot process images of different sizes. Hence, we take each image received from stage 1 and rescale it into 224 x 224-pixel size. The following line of code performs this task for us:

```
cv2.resize (raw_img[cy:cy+ch,cx:cx+cw], (int(224), int(224)), interpolation = cv2.INTER_CUBIC)
```

resize function ↓ image file ↓ cutting for face ↓ length ↓ breadth ↓ a interpolation step for the process ↓

In the above command *cx, cy, ch* and *cw* are the values which we get from the previous stage after the face has been extracted. They correspond to the x, y, height, and width of the box within which a face is detected. *raw_img* is the raw image matrix which is a 2D array, by passing *[cy:cy+ch,cx:cx+cw]* we extract the image and resize it using the command *cv2.resize* by passing the image and size as the arguments.

The above command gives us an image with a standard size which is to be supplied to the CNN. Apart from having the size as the dimension we need another dimension such as the three-color channels: Red, Blue and Green (RGB). This gives the intensity of the image with respect to each of the primary color. Lastly there are **n** number of samples for the CNN hence we converge all the images together to produce a single data structure.

Resulting to a 4-Dimensional 'tensor'(datastucture or matrix) as shown below:

(n , 224 , 224 , 3)

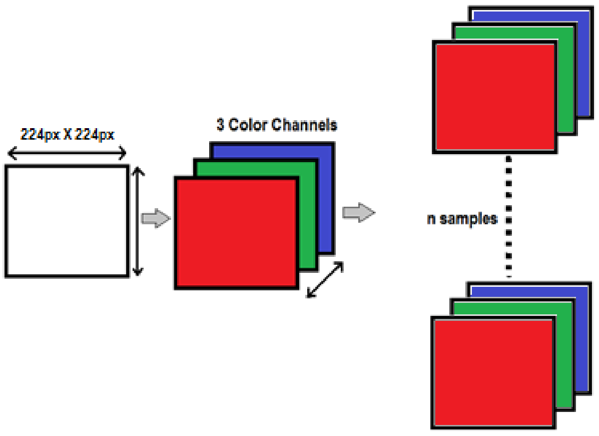


Fig 6: Conversion of an image to a 4D tensor

Now if in 'stage 1' there is no face detected then the we simply create a blank image matrix of the same size using numpy library:

```
np.zeros(shape=(224, 224, 3))
```

7. Architecture

Getting the right architecture for the neural network is an iterative process. The CNN consists of multiple layers of different kinds. We start with a basic structure with only one or two layers and then proceed with improvements in the architecture by making few changes at a time and keeping track of them. Firstly, lets describe the kinds of layers our CNN comprises of. This will help us understand the why we are using these layers. CNN architecture mainly consists of the following layers and their functionalities are:

I. Convolutional layer 2D

A layer that consists of a set of "filters". The filters take a subset of the input data at a time, but are applied across the full input (by sweeping over the input). The operations performed by this layer are still linear/matrix multiplications, but they go through an activation function at the output, which is usually a non-linear operation. This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs.

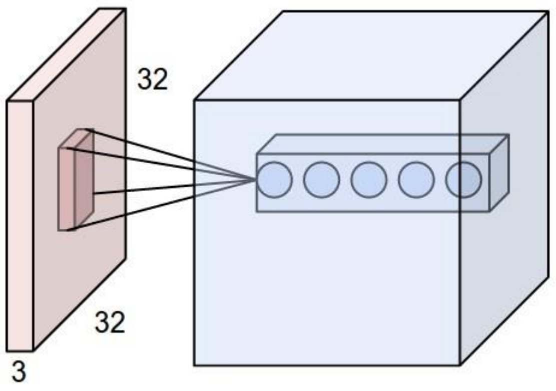


Fig 7: Visual depiction of how 32x32x3 input layer passes data to the convolutional layer (cubical)

II. Max Pooling Layer 2D

After each ReLU layer, we choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are several layer options, with maxpooling being the most popular. This layer basically takes the kernel (2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every sub region that the filter convolves around as shown below:

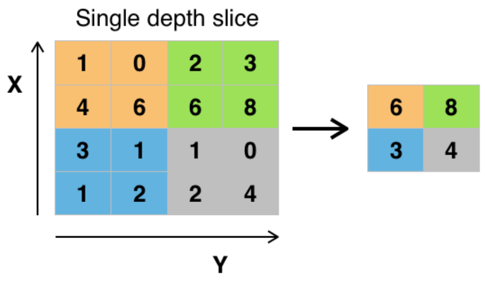


Fig 8: Max pooling example

III. Dropout Layer

Dropout layers have a very specific function in neural networks. Sometimes we have the problem of overfitting, where after training, the weights of the network are so tuned towards the training dataset they are given that the network doesn’t perform well when given new examples. The idea of dropout is simplistic in nature. This layer “drops out” a random set of activations in that layer by setting them to zero. This forces the network to be redundant which means the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn’t getting too “fitted” to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

IV. Dense Layer

Dense (fully connected) layers, which perform classification on the features extracted by the convolutional layers and down sampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the preceding layer.

V. Global Average Pooling Layer 2D

Global max pooling operation for temporal data. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer

An example visualization of a convolutional neural network is shown below as per [6] :

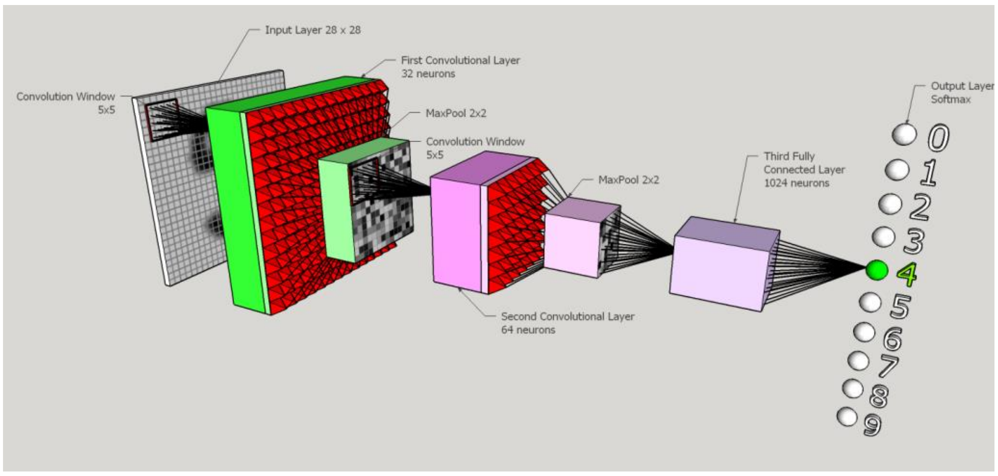
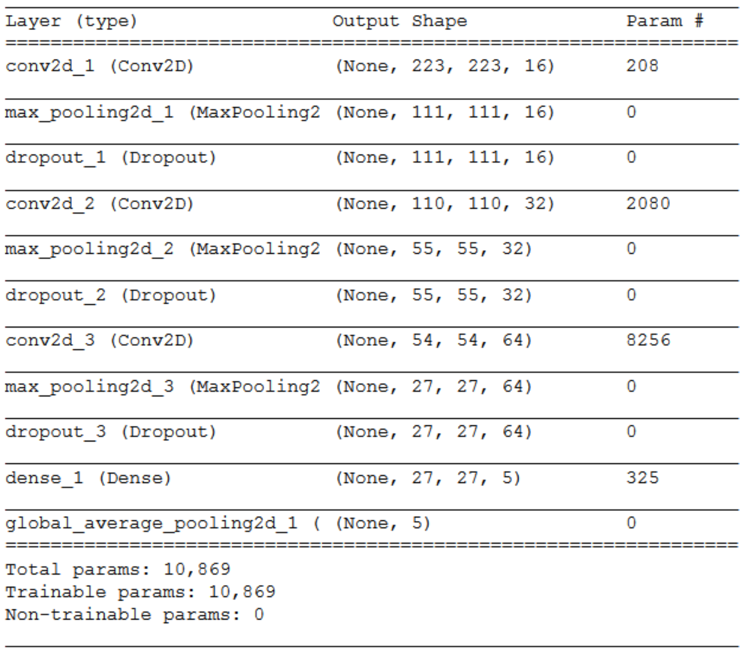


Fig 9: Visual representation of a similar Convolutional Neural Network

In the above architecture we can see there is an input layer followed by two sets of convolutional layers along with its max pooling layers, then followed by a fully connected layer which connects to ten possible output cases.

The final architecture obtained is the following where we can see the total number of params is **10,869**, these are the total number of connections between the layers hence the total number of weights. By increasing the number of layers, the number of connections increase by a fixed factor depending on the layer type. Hence a point to be noted is that more the layers in the CNN, better can be the results but with the increase in the number of connections the training process becomes more exhaustive hence consuming more time and memory.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
dropout_1 (Dropout)          (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
dropout_3 (Dropout)          (None, 27, 27, 64)        0
_________________________________________________________________
dense_1 (Dense)              (None, 27, 27, 5)         325
_________________________________________________________________
global_average_pooling2d_1 ( (None, 5)                 0
=================================================================
Total params: 10,869
Trainable params: 10,869
Non-trainable params: 0
_________________________________________________________________
```

Fig 10: Convolutional Neural Network model used in the experiment

As you can see above the model is a sequential model and three layers (conv 2d, max pooling 2D and dropout layer) are repeated continuously with decreasing output shape (due to the max pooling layer). We end up using this structure by trial and error method. By increasing the layers, the model was overfitting the data hence resulted in decreased accuracy. Similarly, by decreasing the layers the CNN model underfitted the data. Hence this configuration is the perfect balance for these layers.

8. Training

This is the process in which major computational power and time is consumed. The CNN or any machine learning algorithm consists of this necessary step. The training of the CNN starts by first compiling the CNN model which we built in the architecture section (section 7) with the command line:

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

Here we fix the optimizer and the loss function for the model and choose the metric form to measure the accuracy. The training consists of two main steps; namely the feed forward and the back propagation. These two steps are performed iteratively for all the epochs. Epochs is a hyperparameter which decides the number of training cycles for the CNN. Let us further discuss the two steps:

I. Feed forward

As we know that a 4D tensor consists of multiple samples of the dataset as **n** is one of the dimensions. The CNN while feed forward step takes one sample as an input at a time. Hence the input size of the data structure for one feed forward step is 224x224x3. So now in this step the data structure is passed through the CNN layers. As it propagates the weights of the connection between the layers affect the output from the previous layer. Therefore, reaching the last output layer with values that correspond to categorical distribution. One this step is completed the output of the CNN is compared to the actual output (labeled output) and the loss is calculated with the help of 'categorical_crossentropy' which is the loss function stated while compiling the CNN model. The accuracy of the prediction is also measured. It is clearly see in the results section that accuracy is inversely proportional to the loss.

II. Back propagation

As the loss is calculated, the goal of the training process is to reduce the loss so that the accuracy of the CNN can be increased. Hence in this step the gradient descent is calculated for each of the weights(parameters) while propagating backwards through the CNN. This step readjusts the weights and prepares the CNN for a better prediction in the next iteration. The type of gradient descent function used is 'rmsprop' which is also stated while compiling the CNN model.

These two steps are performed repeatedly in every iteration and the loss can be seen reducing with increasing accuracy.

For training we use the following command:

```
                model.fit(train_tensors, train_targets,
            validation_data=(valid_tensors, valid_targets),

   epochs=500, batch_size=20, callbacks=[checkpointer], verbose=1)
```

During the training process some weights are randomly dropped with a drop probability of 0.4. This helps in preventing overfitting of the training set with the CNN model.

9. Testing

In this process the images are converted into 3 dimensional tensors and provided to the CNN. Thus, each of the samples from the testing set is predicted one by one. The difference between this step and the training step is that here the weights are neither changed nor dropped due to the drop layer. There is no back propagation, but only feed forward. Testing phase is always the quicker phase when compared to the training phase. If the accuracy of the CNN model is very high during the training stage and very low during the testing stage, this means that the CNN model is overfitted and biased towards the training dataset. Therefore the accuracy of the model during the testing is more important than the accuracy during the training stage.

10. Results

After training the model with the training set we came up with some of the conclusions that there are a number of factors affecting the accuracy of the CNN in predicting the right individual. Following are the factors of that impact greatly:

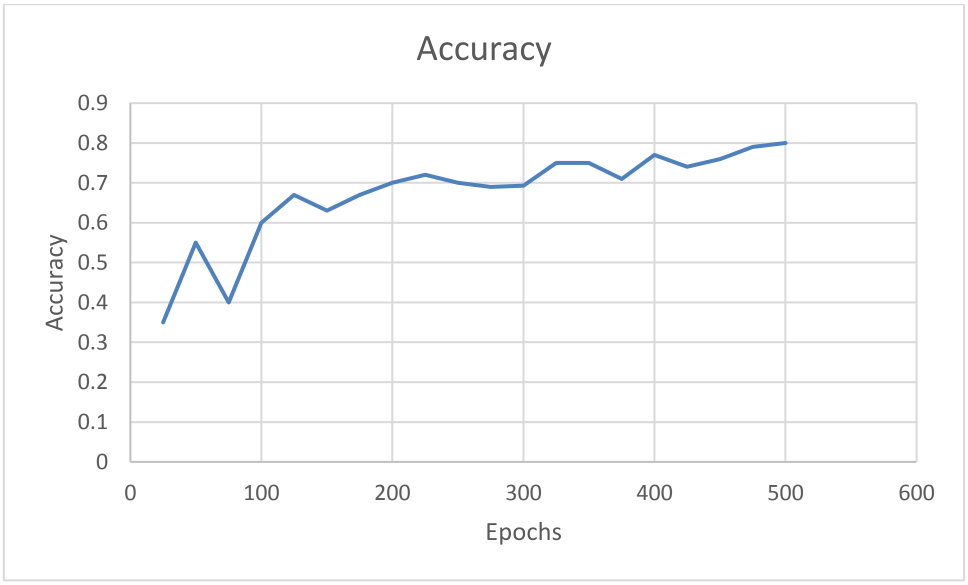
- Number of training images: the more the images the better be the prediction.
- Variation in the images: with more variations in the images the CNN can predict accurately any variation in the test image.
- Number of pixels in the images: with increase in the number of pixels in the input images the we could train the CNN for more detailed information but the training process will increase considerably also increasing the number of weights(connections) in the neural network.
- Number of iterations in the training process (Epochs): With increase in the iterations in the training process the accuracy of the CNN increases, but there is a tradeoff as the time taken to train the CNN increases as well.
- Batch size: can be variable to the dataset and the CNN, for some architectures the perfect batch can be small but for some it may be large. For our CNN model we have tried batch sizes 10, 15, and 20 out of which 20 is the best size.
- Dropout: This is a configuration made to the layer where the connections are the connections are dropped for each training iteration, thus preventing the CNN from overfitting any training set

While training the CNN model we get the following results

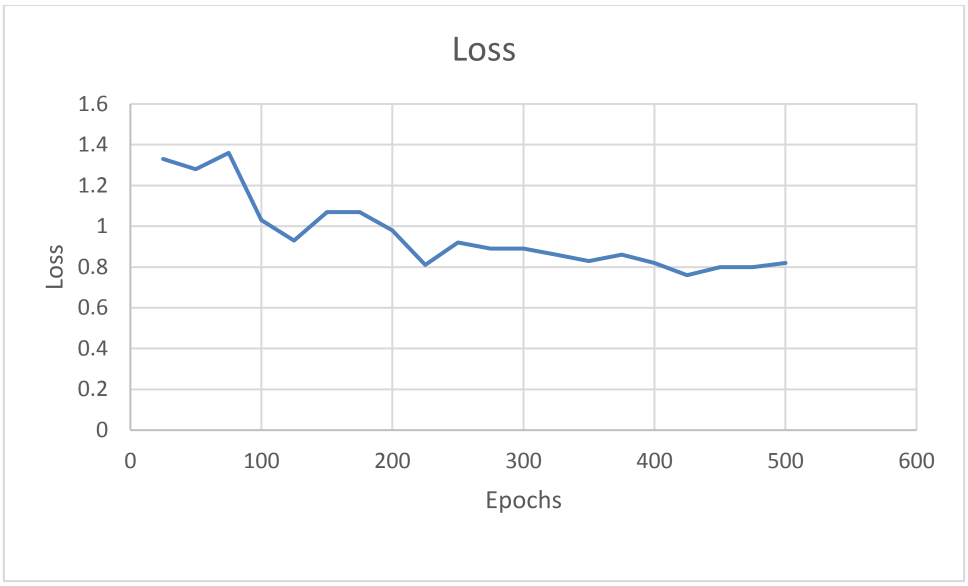
Epochs	Accuracy	Loss
25	0.35	1.33
50	0.55	1.28

Epochs	Accuracy	Loss
75	0.4	1.36
100	0.6	1.03
125	0.67	0.93
150	0.63	1.07
175	0.67	1.07
200	0.7	0.98
225	0.72	0.81
250	0.7	0.92
275	0.69	0.89
300	0.693	0.89
325	0.75	0.86
350	0.75	0.83
375	0.71	0.86
400	0.77	0.82
425	0.74	0.76
450	0.76	0.8
475	0.79	0.8
500	0.8	0.82

Table 1: Accuracy and Loss with respect to Epochs



Graph 1: Epochs vs Accuracy



Graph 2: Epochs vs Loss

As we can see in the first graph that accuracy increases with increase in the epochs (number of iterations) for training the CNN and slowly levels when the epochs reaches 500. Inversely the loss value calculated by the loss function, reduces with increase in the number of epochs and starts levelling when reaches 400 epochs.

This also shows that as the iterations reach higher levels we get smaller improvements in the accuracy and smaller reduction in the loss value.

We tried the following configurations to train the CNN to get the best accuracy on the test data set, with the final values as shown below:

Dropout	Epochs	batch-size	Accuracy %
0.4	200	20	81.8182
0.1	200	20	63.63
0.5	200	20	81.81
0.5	200	10	68.18
0.5	200	15	63.63

Table 2: How accuracy is affected by dropout layers, epochs and batch size

Using the best configuration that is the first configuration (first row from Table 2), we have a CNN that has dropout layers with 0.4 drop probability, 200 iterations and each iteration has a batch of 20 images randomly selected for the CNN training.

With the accuracy of 81.8182%, we supply the test input images to the CNN. The five output classes are one hot encoded and the 21 test images are supplied to the CNN. Due one hot encoding each individual corresponds to a number from 0 to 4, given below are the actual outputs and the predicted outputs.

Actual Input - [4 2 3 3 0 4 2 4 2 1 3 1 0 1 4 2 2 4 1 0 3 3]

Predicted Output - [4 2 3 3 4 4 2 4 2 1 3 1 2 1 4 2 2 2 1 4 3 3]

As we can see above the correctly predicted data can be seen by the green arrow, while the incorrectly predicted value is shown by the red arrow.

We can also see from the probability matrix(categorical distribution) of the first five input images below:

```
[[ 0.12438317  0.09723664  0.27288526  0.03357404  0.47192094]] -> class 4
[[ 0.13904156  0.2179421   0.39775792  0.08742823  0.15783022]] -> class 2
[[ 0.04653785  0.22481127  0.00097527  0.53150946  0.19616586]] -> class 3
[[ 0.15052773  0.03183563  0.24317318  0.50212592  0.07233757]] -> class 3
[[ 0.20796163  0.02719471  0.0051004   0.37375233  0.38599089]] -> class 4
```

Which can be obtained by using the following command:

```
for tensor in test_tensors:
    print((model.predict(np.expand_dims(tensor, axis=0))))
```

In the above matrix each row represents the predicted probability distribution of an input image with each column the corresponding output space. The highest probability from each row is chosen as the winning class (individual).

11. Conclusion

In this independent study, we build a Convolutional Neural Network with the help of keras library and OpenCV to recognize five individuals. We conclude that CNN is a very robust system that can evolve with the changes in the features of a face and still predict with a high accuracy. With only four sets of layers and 500 epochs we have attained an accuracy of more than 81.81%. We also observe the factors affecting the accuracy of the prediction. This also means that the CNN gets the ability to find out similarities in the images. Therefore, a CNN can be widely used to detect and recognize patterns in images.

12. Glossary

a) Activation functions – In computational networks activation functions are simple functions that produce an output for a certain input. These functions can be set to have different thresholds which also allow to categorize the outputs into various groups. In the CNN we us e two kinds of activation functions; the ‘ReLU’ and the ‘softmax’ functions.

b) Tensors – Are data structures used for the CNN. This term came into existence with the library ‘tensorflow’ which is the main library and runs in the background of the keras library. The keras library is a library used only for the neural networks. Tensors are the data structures which are passed through the CNN. Hence each data set which includes the images are converted into 4 dimensional tensors to pass through the CNN.

c) Eigen faces/ Ghost images –are the resultant images of a group of images when they are all averaged. Each image can be converted into a 2 dimensional matrix (grayscale image) where each element represents the pixel value from 0 to 255. Hence, when provided with a group of images of the same size, all the images can be combined (index value is averaged from all the image matrices) to form an image which is known as the eigen face or ghost image.

d) Epoch – Is the term used to denote the number of iterations for the training process. While training the CNN the number of iterations matter as the accuracy of the predictions are observed to increase with increase in the number of iterations for training. With more iterations the CNN is supplied with different arrangements of different data which makes the CNN more robust.

e) Image matrix/array – Any image file can be stored in a 2D array or matrix form. Each pixel can be represented as a value in the matrix. The image is spatially equivalent to the matrix. If the image is a grayscale image then the depth of the matrix is 1, but if the image is a colored image then the depth of the matrix becomes 3 hence resulting to a 3D matrix. This is due to the three-color channels present in the image; Red, Green and Blue.

f) Kernel – Is a mask or a small sub matrix that swipes through the input matrix (master) to perform a particular function to the elements of the master matrix. It acts like a funnel. This concept is used in image processing like for blurring, sharpening, embossing and edge detection. In the figure below we can visualize the kernel as the red grid that swipes through the grid over which it is.

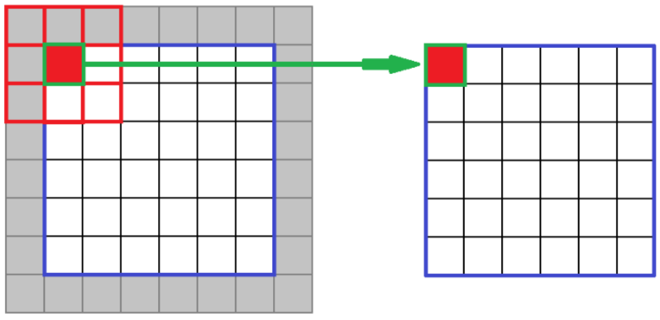


Fig 11: Kernel

g) Neuron –This term is interchangeably used with a perceptron

h) Object – Is the term used for defining an instance in software development. It can be a variable, a function, a data structure or just a memory location.

i) Over fitting – Is a statistical term used to denote when a model (the CNN) is heavily dependent and biased on the training dataset. For instance, if we have only one image for training the CNN and we increase the training iterations to a high number like 100, then the same image will be supplied to the CNN 100 times while making the CNN biased towards this image only. This will lead to drop in accuracy in the prediction if there is also a small change in the image used for the testing than the image used for the training process. Hence the CNN will be overfitted to the training image. Hence to avoid overfitting we increase the number of images with much more variations and each training iteration has a different kind of image.

j) Perceptron – is the basic functional element of a neural network. This unit is equivalent to the sum of the weighted inputs and a bias(constant) to result in an output. The perceptron includes three computational parts; the weights, the summing function and the activation function.

k) Under fitting – is a scenario which occurs when the CNN is very generalized. Like for instance if we do not extract the faces and supply the whole images to the CNN. Then the CNN would also train itself with the irrelevant information that is in the background. This could lead to a CNN with everchanging weights while training and the CNN might never reduce its loss value, hence the obtained prediction will never be too accurate as the CNN model will always give a generalized prediction.

Statistically underfitting and overfitting can be seen in the figures below:

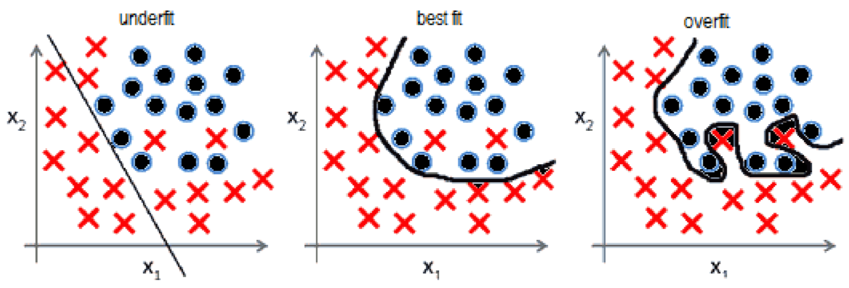
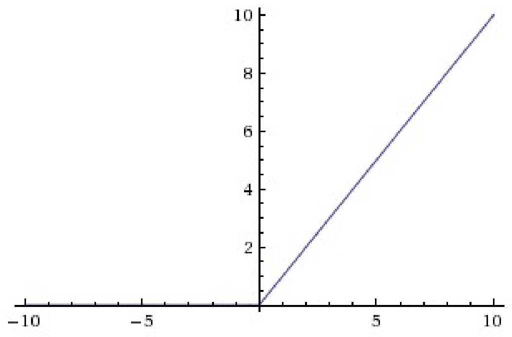


Fig 12: Fitting of the model

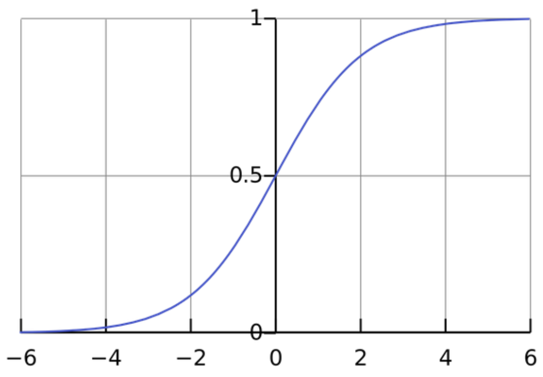
As we can see above, a line is drawn to show the separation of the crosses from the dots (considering them as the training data). If the line underfits (first graph from the left) then there are a lot of crosses in the wrong of the line. When the line overfit the graph (third from the left) then the line perfectly partitions the crosses from the dots or rather too perfectly, hence if a new data (testing data) is plotted on the graph then there is a high probability that the data plotted near the line will be partitioned by the line incorrectly.
Thus, neither underfitting nor overfitting is good.

l) ReLU function – Is an activation function that gives the output zero for negative values and for positive values the output stays the same. As shown below in the graph.



Graph 3 : Showing ReLU plot

m) Softmax function – is an activation function which outputs a value between 0 and 1 for a particular value. This activation function is usually used in the last layer of the CNN or any neural network. Due to this function we get a categorical distribution for the output classes. The plot below shows how a softmax function looks like graphically.



Graph 4 : Showing softmax plot

n) Params / Weights / Parameters – Mathematically these are the coefficients which are associated with the inputs of each layer. These weights are the deciding factors for the quality of the CNNs. If these weights are lost then the CNN has to be retrained to get a set of new weights. A point to be noted is that every training process will lead to different set of weights for the CNN. The weights of the CNN can be saved and loaded as to avoid the long training processes.

o) Gradient descent – is the process of reducing the loss in a CNN. If loss can be visually imagined as a gaussian distribution. With the peak(crest) signifying the highest loss value and the base(trough) signifying the least loss value then to find the slope of this distribution and reduce the loss value, each of the weights are derived with respect to the loss value. This whole procedure is known as gradient descent. The figure below is an example that shows the process of gradient descent visually as J of theta 0 and theta 1 the loss value. The two black lines shown in the plot denote the reduction in the loss value as the lines go down the slope.

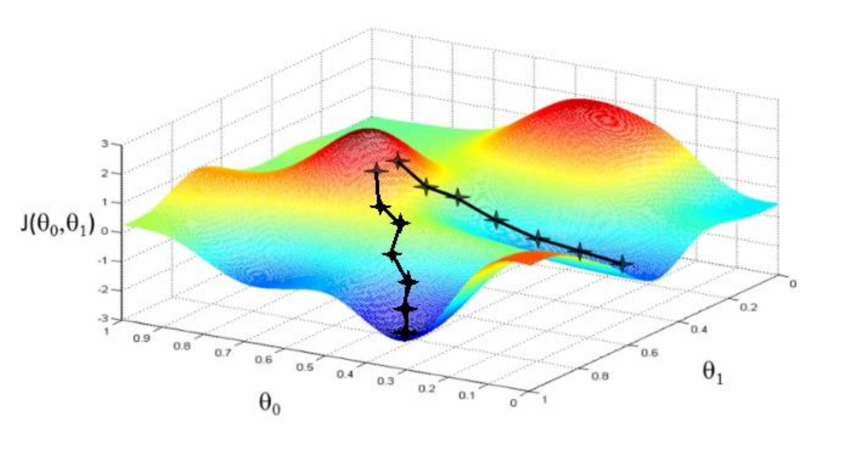


Fig 13: Gradient Descent visualization

p) Loss function – also known as the cost function, is a function that gives the penalty of a wrong prediction for the CNN model. It is used during the training step only

q) Output classes – Since the CNN has a fixed input space, the number of output categories are also fixed. These output categories are known as output classes. In our CNN the output classes are one hot encoded as following:

   Ankit - 0

   Deepansh - 1

   Hitesh - 2

   Omkar - 3

   Tanay - 4

r) Categorical distribution – In probability theory and statistics, a categorical distribution is a discrete probability distribution that describes the possible results of a random variable that can take on one of the output classes, with the probability of each elementary event separately specified.

s) Hyperparameters – are parameters whose values are set before the training process of the CNN. These parameters are small tweaks which can lead to overfitting and underfitting of the models. Eg. Epochs and batch size.

t) Optimizer – is the class used in tensorflow and keras for selecting the gradient descent algorithm. For example, we used ‘rmsprop’ as the optimizer [12].

u) rmsprop – is a mathematical function which is used by the CNN to improve its performance. For further details please refer to [12]

13. References

1. http://udacity.com
2. http://cs231n.github.io/convolutional-networks/
3. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/
4. https://www.tensorflow.org/versions/master/tutorials/layers
5. https://en.wikipedia.org/wiki/TensorFlow
6. http://zderadicka.eu/revival-of-neural-networks/
7. https://en.wikipedia.org/wiki/Object_(computer_science)
8. http://mlwiki.org/index.php/Overfitting
9. http://cs231n.github.io/neural-networks-1/
10. https://en.wikipedia.org/wiki/Categorical_distribution
11. https://en.wikipedia.org/wiki/Categorical_distribution
12. http://ruder.io/optimizing-gradient-descent/index.html#rmsprop
13. http://aicat.inf.ed.ac.uk/entry.php?id=636