Introduction

Emotions are the essence of what makes us human. They impact our daily routines, our social interactions, our attention, perception, and our memory. One of the strongest indicators of emotions is our face. As we laugh or cry we're putting our emotions on display, allowing others to glimpse into our minds as they "read" our face based on changes in key facial features such as eyes, brows, nostrils, and lips.
Computer-based facial expression analysis mimics our human coding skills quite impressively as it captures raw, unfiltered emotional responses towards any type of emotionally engaging content. But how exactly does it work?

What are facial expressions?

Facial expressions are movements of the numerous muscles supplied by the facial nerve that are attached to and move the facial skin.

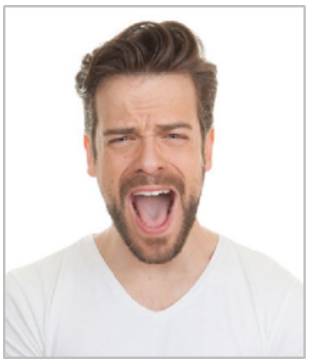
Some categorical emotions are:

JOY

ANGER

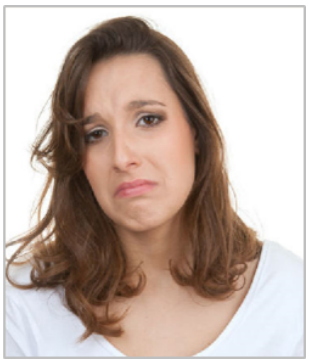
SURPRISE

FEAR

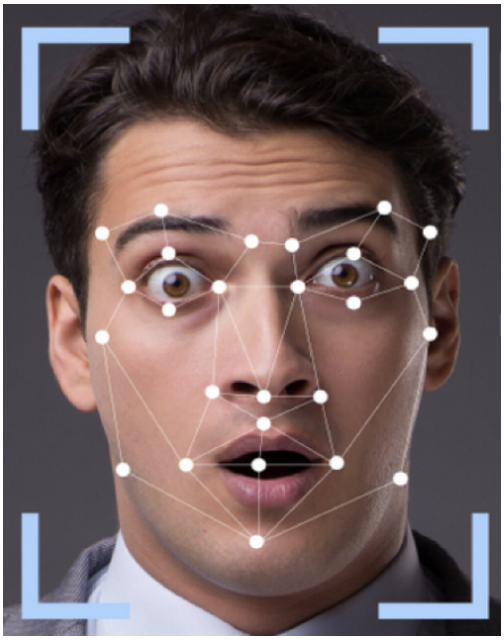


CONTEMPT

SADNESS



What is Facial Emotion Recognition???



Facial Emotion Recognition is a technology used for analyzing sentiments from different sources, such as pictures and videos. It belongs to the family of technologies often referred to as ``affective computing, a multidisciplinary field of research on computers' capabilities to recognize and interpret human emotions and affective states and it often builds on Artificial Intelligence technologies. Facial expressions are forms of non-verbal communication, providing hints for human emotions. For decades, decoding such emotional expressions has been researching interest in the field of psychology.

The finding that almost everyone can produce and recognize the associated facial expressions of these emotions has led researchers to assume that they are universal.

Facial expression for emotion detection has always been an easy task for humans, but achieving the same task with a computer algorithm is quite challenging. With the recent advancement in

computer vision and machine learning, it is possible to detect emotions from images. In this paper, we propose a novel technique called facial emotion recognition using convolutional neural networks (FERC).

The FERC is based on a two-part **convolutional neural network (CNN):** the first part removes the background from the picture, and the second part concentrates on the facial feature vector extraction. In the FERC model, an **expressional vector (EV)** is used to find the five different types of regular facial expressions. Supervisory data were obtained from the stored database of 350 images. It was possible to correctly highlight the emotion with **96% accuracy**, using an **EV of length 24 values**. The two-level CNN works in series, and the last layer of perceptron adjusts the weights and exponent values with each iteration.

FERC differs from generally followed strategies with single-level CNN, hence improving the accuracy. Furthermore, a novel background removal procedure, before EV generation, avoids dealing with multiple problems that may occur (for example, distance from the camera).

Methodology

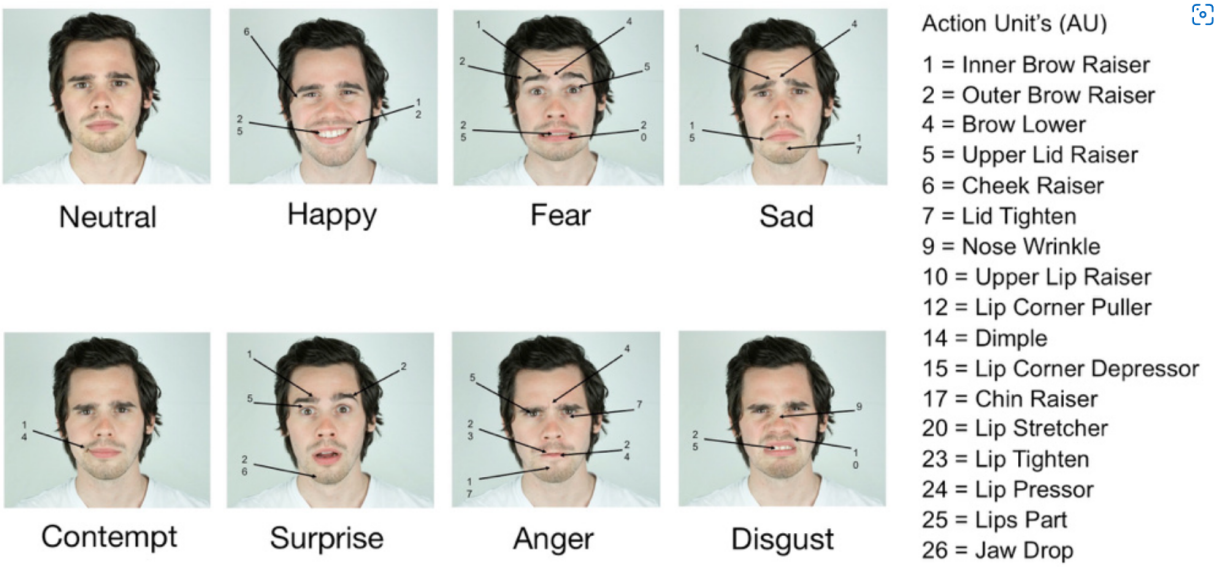
A convolutional neural network (CNN) is the most popular way of analyzing images. CNN is different from a multi-layer perceptron (MLP) as they have hidden layers, called convolutional layers. The

proposed method is based on a two-level CNN framework. The **first level** recommended is **background removal**, used to extract emotions from an image, as shown in Fig. 1. Here, the conventional CNN network module is used to extract the primary expressional vector (EV).
The expressional vector (EV) is generated by tracking down relevant facial points of importance. EV is directly related to changes in expression. The EV is obtained using a basic perceptron unit applied on a background-removed face image. In the proposed FERC model, we also have a non-convolutional perceptron layer as the last stage. Each convolutional layer receives the input data (or image), transforms it, and then outputs it to the next level. This transformation is a convolution operation. All the **convolutional layers** used are capable of pattern detection.
Within each convolutional layer, **four filters** were used.
The input image fed to the first-part CNN (used for background removal) generally consists of **shapes, edges, textures, and objects** along with the face. The edge detector, circle detector, and corner detector filters are used at the start of convolutional layer 1.
Once the face has been detected, the **second-part** CNN filter catches **facial features**, such as eyes, ears, lips, nose, and cheeks.
The second-part CNN consists of **layers with 3×3 kernel matrix, e.g., [0.25, 0.17, 0.9; 0.89, 0.36, 0.63; 0.7, 0.24, 0.82]**.
These numbers are selected between 0 and 1 initially. These numbers are optimized for EV detection, based on the ground truth we had, in the supervisory training dataset. Here, we used minimum error decoding to optimize filter values.
Once the filter is tuned by supervisory learning, it is then applied to the background-removed face (i.e., on the output image of the first-part CNN), for detection of different facial parts (e.g., eyes, lips. nose, ears, etc.

The Facial Action Coding System often referred to as ‘FACS’, is an internationally recognized, sophisticated research tool that precisely measures the entire spectrum of human facial expressions. FACS has elucidated the physiological presence of emotion with very high levels of reliability.

The system dissects observed expressions by determining how facial muscle contractions alter appearance. Each movement is categorized into specific Action Units (AUs), representing the contraction or relaxation of one or more muscles. All facial expressions can be decomposed into their constituent AUs and described by duration, intensity, and asymmetry. Trained experts examine patterns in the changing nature of facial appearance, including movement, changes in shape and location of the features, and the gathering, pouching, bulging, and wrinkling of the skin. Understanding the coordination between action units and certain expressions illuminates the implications of human body language and non-verbal behavior.

THE 7 UNIVERSAL FACIAL EXPRESSIONS OF EMOTION – FACS CODED



How does Facial Expression Recognition work?

A facial expression recognition system is a computer-based technology and therefore, it uses algorithms to instantaneously detect faces, code facial expressions, and recognize emotional states. It does this by analyzing faces in images or video through computer-powered cameras embedded in laptops, mobile phones, and digital signage systems, or cameras that are mounted onto computer screens. Facial analysis through computer-powered cameras generally follows three steps:

1. Face detection

Locating faces in the scene, in an image, or video footage.

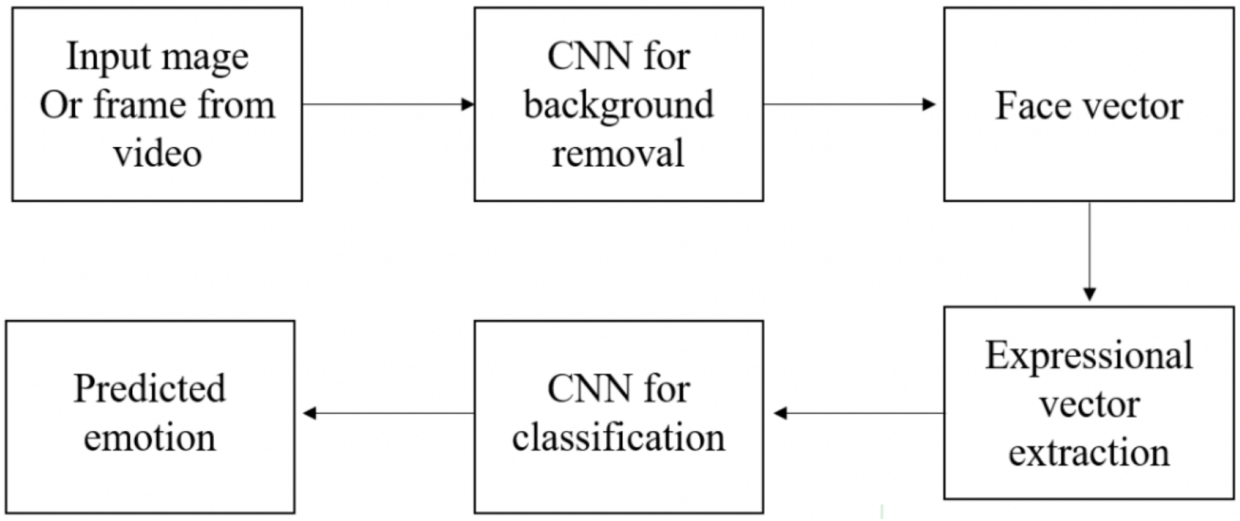
2. Facial landmark detection

Extracting information about facial features from detected faces. For example, detecting the shape of facial components or describing the texture of the skin in a facial area.

3. Facial expression and emotion classification

Analyzing the movement of facial features and/or changes in the appearance of facial features and classifying this information into expression-interpretative categories such as facial muscle activations like smile or frown; emotion categories happiness or anger; attitude categories like (dis)liking or ambivalence.

In the past, we got to know the so-called densely connected neural networks. These are networks whose neurons are divided into groups forming successive layers. Each such unit is connected to every single neuron from the neighboring layers. An example of such architecture is shown in the figure below.



(a)

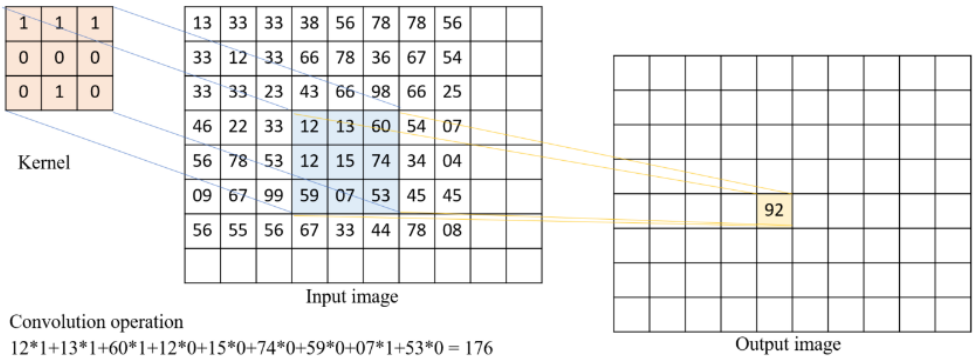
-1	-1	-1	-1	1	0	0	0	0	0	1	-1
1	1	1	-1	1	0	1	1	1	0	1	-1
0	0	0	-1	1	0	-1	-1	-1	0	1	-1

(b)

22	7	7	8	4	4	9	9	7	7	2	9	8	6	6	4	5	5	4	11	12	4	10	6
LE-RE	LE-LY	RE-RY	RY-LY	RY-CN	LY-CN	LE-CN	RE-CN	LE-LP	RE-RP	CN-CP	CP-LE	CP-RE	CF-RE	CF-LE	CF-CN	CF-LY	CF-RY	LH-LE	LH-RE	RH-LE	RH-CE	LH-RH	CF-CP

L: Left RE: Right C: Center Y: Eye E: Ear N: Nose P: Lip F: forehead H:Cheek

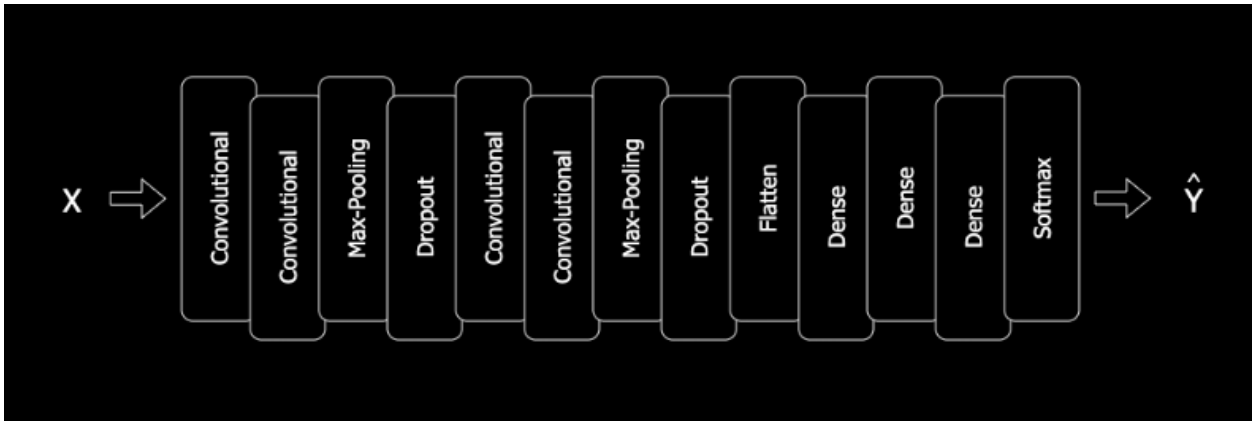
c)



Convolution operation

12*1+13*1+60*1+12*0+15*0+74*0+59*0+07*1+53*0 = 176

CONVOLUTIONAL NEURAL NETWORK ARCHITECTURE



Convolution

Kernel convolution is not only used in CNN's but is also a key element of many other Computer Vision algorithms. **It is a process where we take a small matrix of numbers (called kernel or filter), pass it over our image, and transform it based on the values from the filter.** Subsequent feature map values are calculated according to the following formula, where the input image is denoted by *f* and our kernel by *h*. The indexes of rows and columns of the result matrix are marked with *m* and *n* respectively.

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

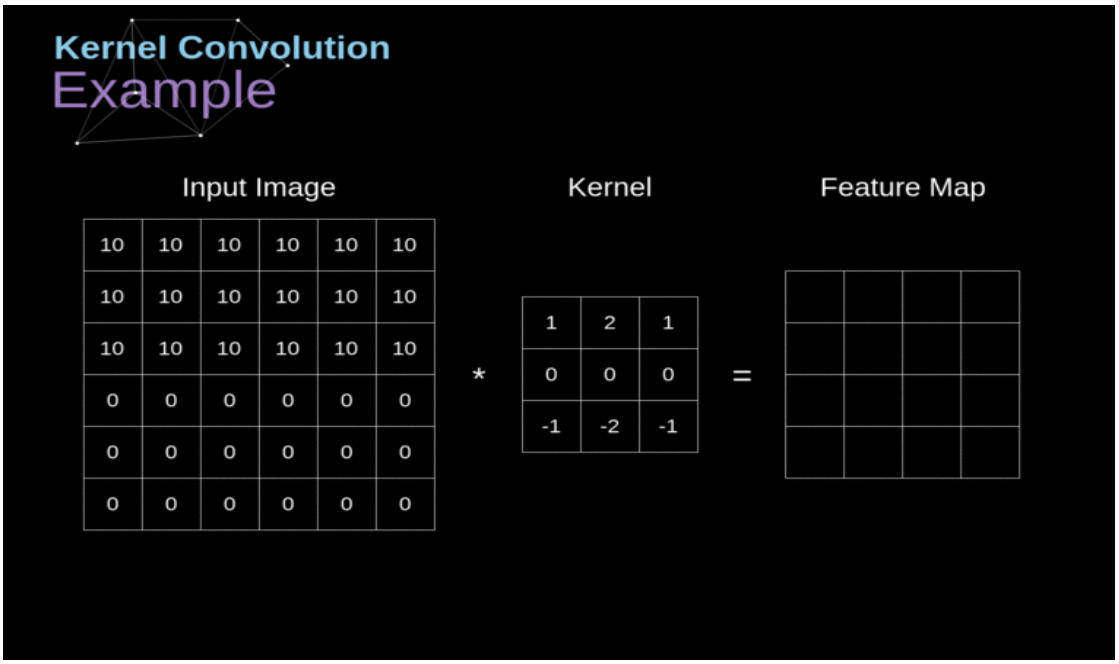


Figure 3. Kernel convolution example

After placing our filter over a selected pixel, we take each value from the kernel and multiply them in pairs with corresponding values from the image. Finally, we sum up everything and put the result in the right place in the output feature map. Above we can see what such an operation looks like on a micro scale, but what is even more interesting, is what we can achieve by performing it on a full image. Figure 4 shows the results of the convolution with several different filters.

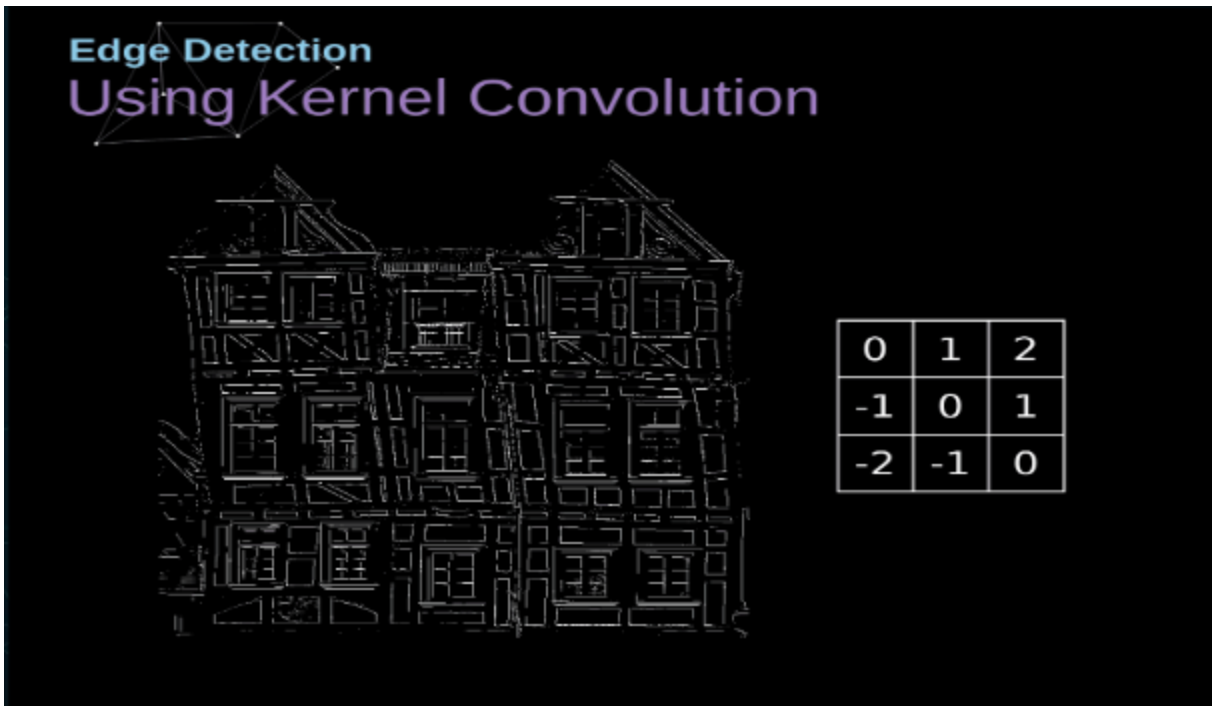


Figure 4. Finding edges with kernel convolution [Original Image]

Valid and Same Convolution

As we have seen in Figure 3, when we perform convolution over the 6x6 image with a 3x3 kernel, we get a 4x4 feature map. This is because there are only 16 unique positions where we can place our filter inside this picture. **Since our image shrinks every time we perform convolution, we can do it only a limited number of times, before our image disappears completely. What's more, if we look at how our kernel moves through the image we see that the impact of the pixels located on the outskirts is much smaller than those in the center of the image.** This way we lose some of the information contained in the picture. Below you can see how the position of the pixel changes its influence on the feature map.

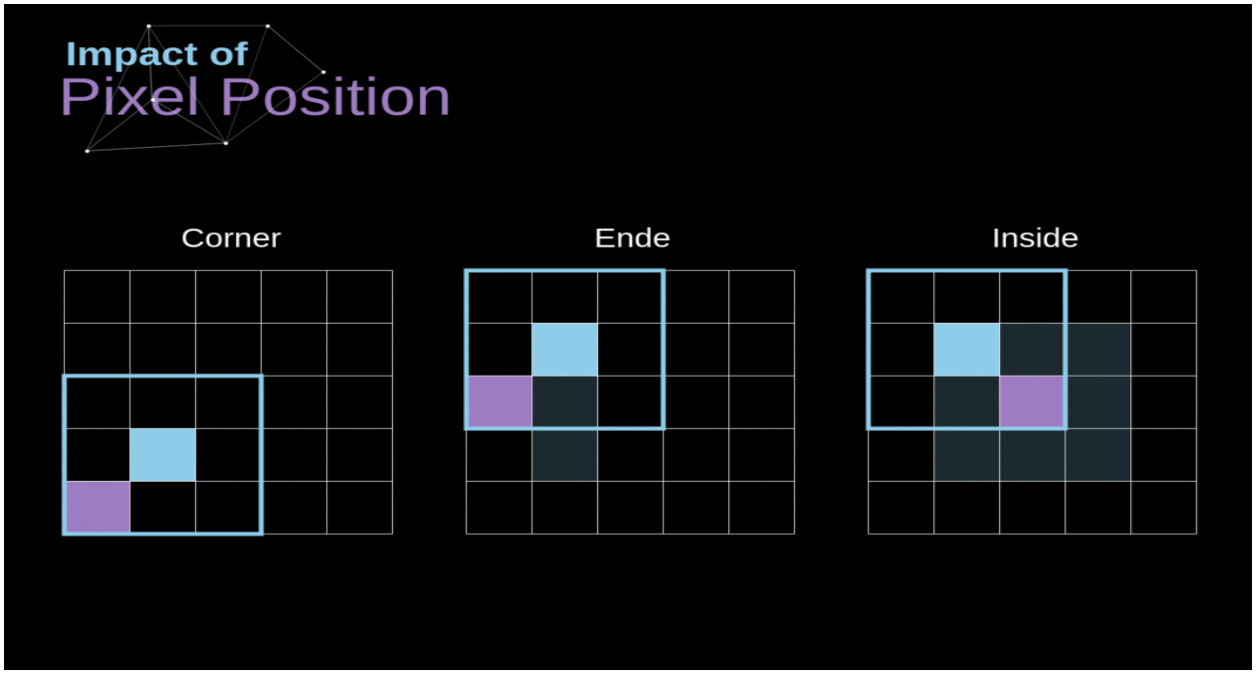


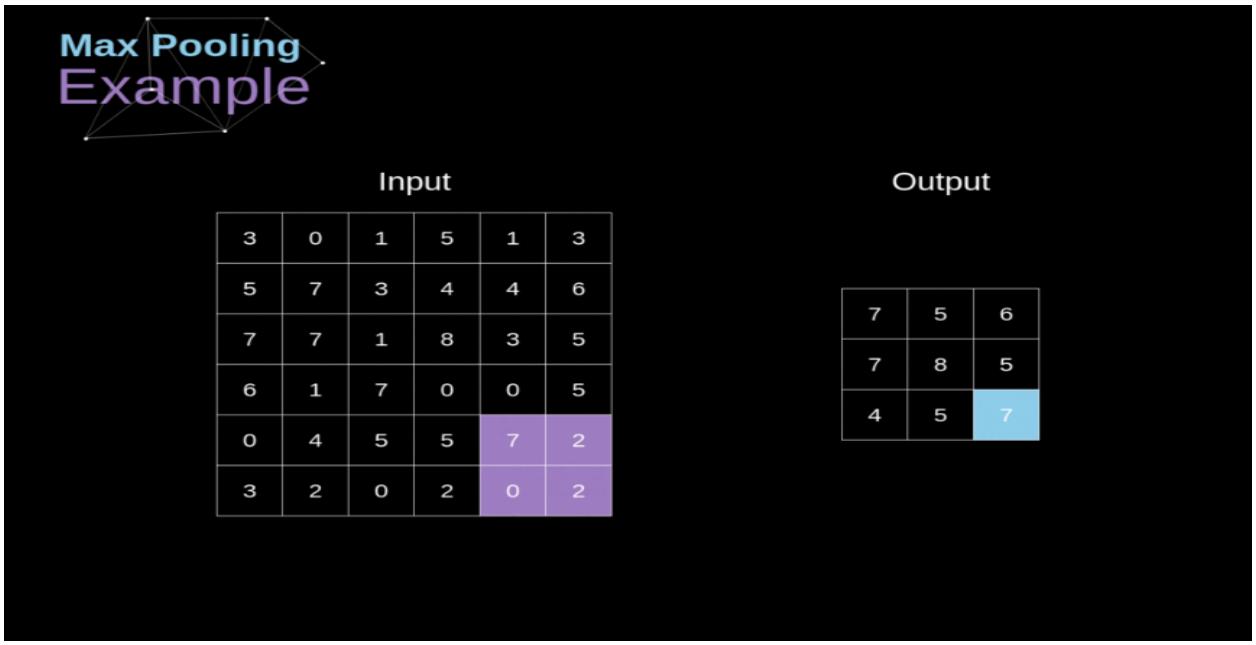
Figure 5. Impact of pixel position

To solve both of these problems we can pad our image with an additional border. For example, if we use 1px padding, we increase the size of our photo to 8x8, so that the output of the convolution with the 3x3 filter will be 6x6. Usually, in practice, we fill in additional padding with zeros. Depending on whether we use padding or not, we are dealing with two types of convolution – Valid and Same. Naming is quite unfortunate, so for the sake of clarity: **Valid – means that we use the original image, Same – we use the border around it** so that the images at the input and output are the same size. In the second case, the padding width should meet the following equation, where p is padding and f is the filter dimension (usually odd).

$$p = \frac{f - 1}{2}$$

Pooling Layers

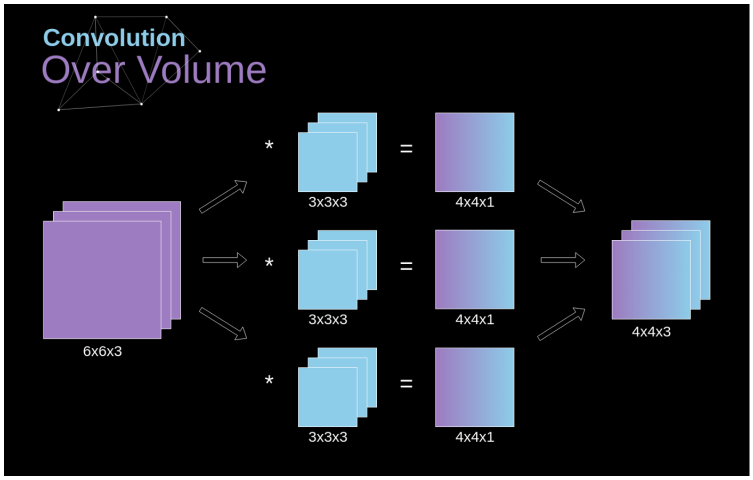
Besides convolution layers, CNNs very often use so-called pooling layers. They are used primarily to reduce the size of the tensor and speed up calculations. These layers are simple - we need to divide our image into different regions, and then perform some operations for each of those parts. For example, for the Max Pool Layer, we select a maximum value from each region and put it in the corresponding place in the output. As in the case of the convolution layer, we have two hyperparameters available – filter size and stride. Last but not least, if you are performing pooling for a multi-channel image, the pooling for each channel should be done separately.



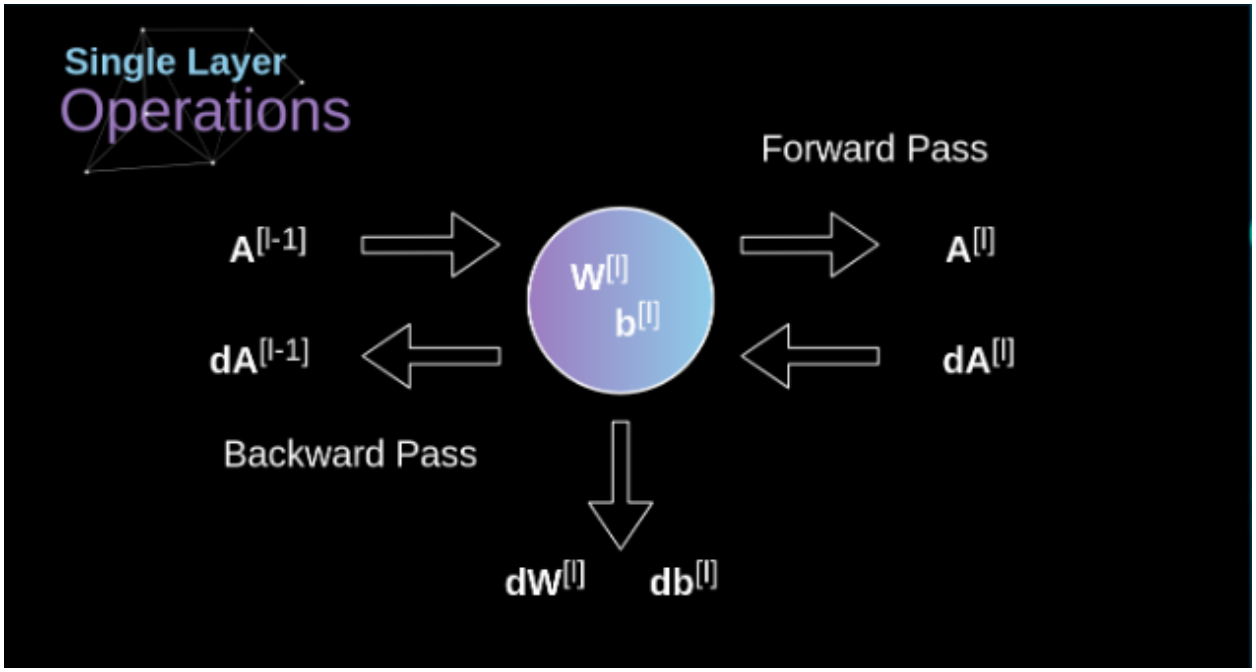
Convolution over volume:

The first important rule is that the filter and the image you want to apply it to must have the same number of channels. Basically, we proceed very much like in the example from Figure 3, nevertheless this time we multiply the pairs of values from the three-dimensional space. **If we want to use multiple filters on the same image, we carry out the convolution for each of them separately, stack the results one on top of the other and combine them into a whole.** The dimensions of the received tensor (as our 3D matrix can be called) meet the following equation, in which: *n* – image size, *f* – filter size, *nc* – number of channels in the image, *p* –used padding, *s* – used stride, *nf* – number of filters.

$$[n, n, n_c] * [f, f, n_c] = \left[\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, n_f \right]$$



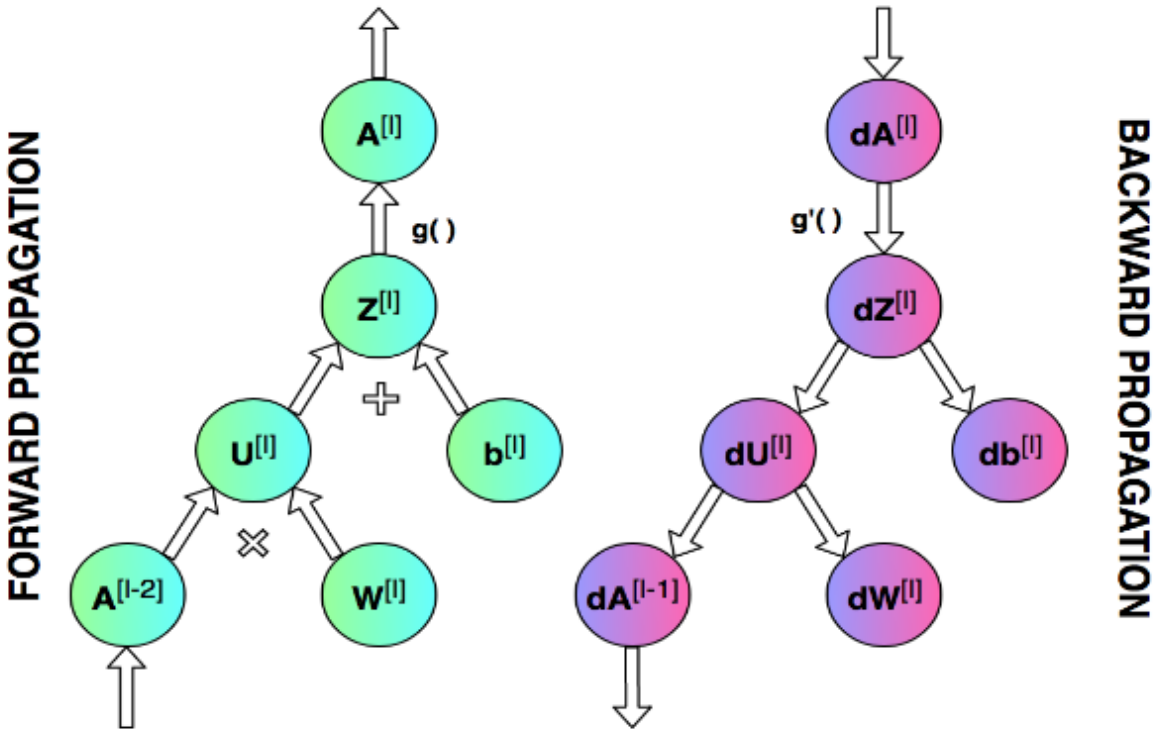
SINGLE LAYER OPERATION



Types of Propagation

There are two flow types – **forward and backward.**
We use **forward propagation to make predictions** based on already accumulated knowledge and new data provided as an input X. On the other hand, **backpropagation is all about comparing our predictions Y_hat with real values Y** and drawing conclusions.

Thus, each layer of our network will have to provide two methods: forward_pass and backward_pass, which will be accessible by the model. Some of the layers – Dense and Convolutional – will also have the ability to gather knowledge and learn. They keep their own tensors called weights and update them at the end of each epoch. In simple terms, a **single epoch of model training is comprised of three elements: forward and backward pass as well as weights update.**

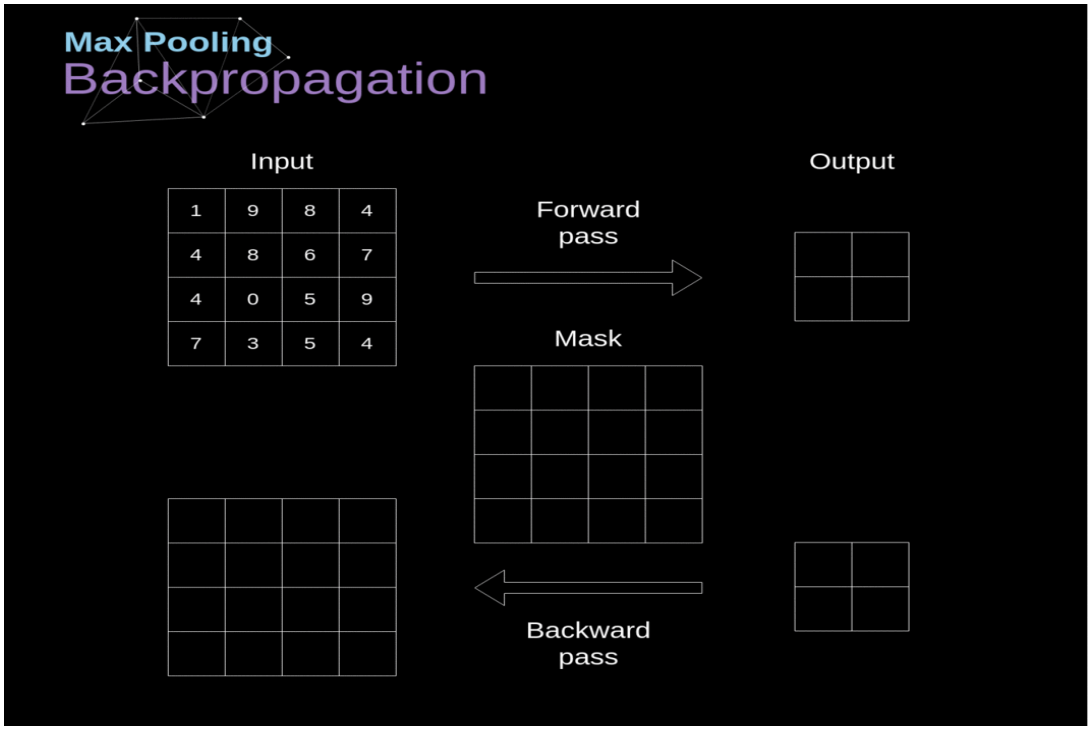


Forward Propagation

The designed neural network will have a simple architecture. The information flows in one direction — it is delivered in the form of an **X** matrix, and then travels through hidden units, resulting in the vector of predictions **Y_hat**. To make it easier to read, I split forward propagation into two separate functions — step forward for a single layer and step forward for the entire NN.

This part of the code is probably the most straightforward and easy to understand. Given the input signal from the previous layer, we compute affine transformation **Z** and then apply the selected activation function. By using NumPy, we can leverage vectorization — performing matrix operations, for the whole layer and whole batch of examples at once. This eliminates iteration and significantly speeds up our calculations. In addition to the calculated matrix **A**, our function also returns an intermediate value **Z**. What for? The answer is shown in Figure 2. **We will need Z during the backward step.**

BACKPROPAGATION



Sadly, backward propagation is regarded by many inexperienced deep learning enthusiasts as an algorithm that is intimidating and difficult to understand. The combination of differential calculus and linear algebra very often deters people who do not have solid mathematical training.

Often people confuse backward propagation with gradient descent, but in fact, these are two separate matters. The purpose of the first one is to calculate the gradient effectively, whereas the second one is to use the calculated gradient to optimize. In NN, we calculate the gradient of the cost function (discussed earlier) with respect to parameters, but backpropagation can be used to calculate derivatives of any function. **The essence of this algorithm is the recursive use of a chain rule known from differential calculus – calculate a derivative of functions created by assembling other functions, whose derivatives we already know.** This process - for one network layer - is described by the following formulas. Unfortunately, because this article focuses mainly on practical implementation, I'll omit the derivation. Looking at the formulas, it becomes obvious why we decided to remember the values of the **A** and **Z** matrices for intermediate layers in a forward step.

DROPOUT

It’s one of the most popular methods for regularization and preventing Neural Network overfitting. The idea is simple – every unit of the dropout layer is given the probability of being temporarily ignored during training. Then, in each iteration, we randomly select the neurons that we drop according to the assigned probability. The visualization below shows an example of a layer subjected to a dropout. We can see how, in each iteration, random neurons are deactivated. As a result, the values in the weight matrix become more evenly distributed. The model balances the risk and avoids betting all the chips on a single number. During inference, the dropout layer is turned off so we have access to all parameters.

FLATTEN

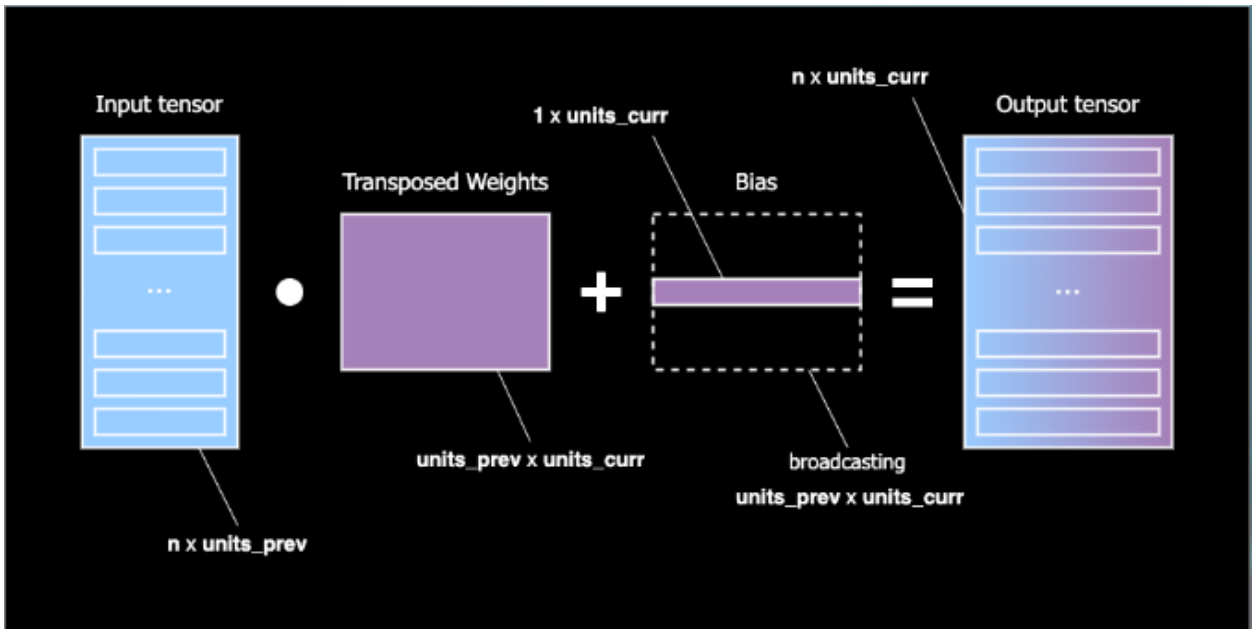
It’s surely the simplest layer that we implement during our journey. However, it serves a vital role of a link between the convolutional and densely connected layers. As the name suggests, during the forward pass, its task is to flatten the input and change it from a multidimensional tensor to a vector. We will reverse this operation during the backward pass.

ACTIVATION

Activation functions can be written in a single line of code, but they give the Neural Network non-linearity and expressiveness that it desperately needs. **Without activations, NN would become a combination of linear functions so that it would be just a linear function itself**. There are many activation functions, but in this project I decided to provide the possibility of using two of them – sigmoid and ReLU. To be able to go full circle and pass both forward and backward propagation, we also have to prepare their derivatives.

DENSE

Similar to activation functions, dense layers are the bread and butter of Deep Learning. You can create fully functional Neural Networks – like the one you can see on the illustration below – using only those two components. Unfortunately, despite obvious versatility, they have a fairly large drawback – they are computationally expensive. Each dense layer neuron is connected to every unit of the previous layer. A dense network like that requires a large number of trainable parameters. This is particularly problematic when processing images.



DENSE LAYER

Applications of Facial Emotion Detection

A facial expression recognition system is a computer-based technology and therefore, it uses algorithms to instantaneously detect faces, code facial expressions, and recognize emotional states.

- **Market Research**

user's reactions are observed while interacting with a brand or a product

- **Video game testing**

game developers can gain insights and draw conclusions about the emotions experienced during gameplay

- **Health Care**

the software helps to decide when patients need medicine, assess their emotional response in clinical trials, or help physicians in deciding how to best triage their patients.

- **Online admissions and interview**

used to understand how candidates feel during interviews and to measure how they react to certain questions.

Here is the code that we have been successfully able to run on Google Colab using python libraries (eg.Keras) which uses CNN:

```
import NumPy as np
import pandas as pd
import matplotlib.pyplot as plt

from Keras. layers import Flatten, Dense
from Keras. models import Model
from Keras.preprocessing.image import ImageDataGenerator , img_to_array,
load_img
from Keras.applications.mobile net import MobileNet, preprocess_input
from Keras.losses import categorical_crossentropy
# Working with pre trained model

base_model = MobileNet( input_shape=(224,224,3), include_top= False )

for layer in base_model.layers:
  layer.trainable = False

x = Flatten()(base_model.output)
x = Dense(units=7 , activation='softmax' )(x)
# creating our model.
model = Model(base_model.input, x)
model.compile(optimizer='adam', loss= categorical_crossentropy ,
metrics=['accuracy']  )
train_datagen = ImageDataGenerator(
     zoom_range = 0.2,
     shear_range = 0.2,
     horizontal_flip=True,
     rescale = 1./255
)
```

Preparing our data using a data generator

```
train_data = train_datagen.flow_from_directory(directory= /content/train",
target_size=(224,224), batch_size=32,)

train_data.class_indices
```

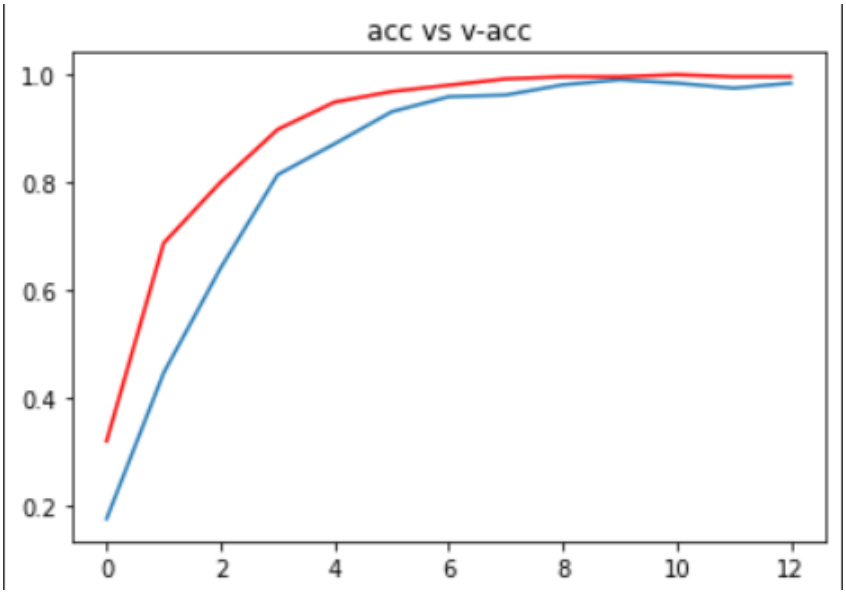
```
Found 350 images belonging to 7 classes.
 'Angry': 0,
 'Disgust': 1,
 'Fear': 2,
 'Happy': 3,
 'Neutral': 4,
 'Sad': 5,
 'Surprise': 6
```

```
t_img , label = train_data.next()

#-------------------------------------------------------------------------
----
# function when called will prot the images
def plotImages(img_arr, label):
  """
  input  :- images array
  output :- plots the images
  """
  count = 0
  for im, l in zip(img_arr,label) :
    plt.imshow(im)
    plt.title(im.shape)
    plt.axis = False
    plt.show()
```

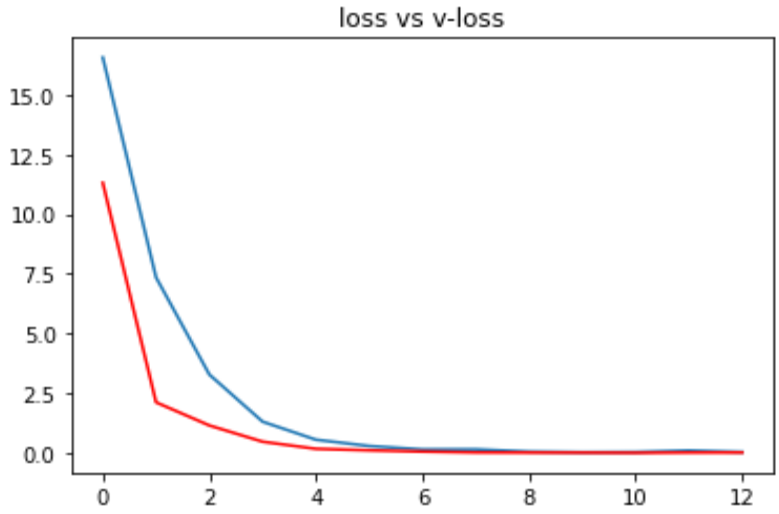
```
    count += 1
    if count == 10:
      break

#-------------------------------------------------------------------------
----
# function call to plot the images
plotImages(t_img, label)
```


```
## having early stopping and model check point
```

```
from keras.callbacks import ModelCheckpoint, EarlyStopping

# early stopping
es = EarlyStopping(monitor='val_accuracy', min_delta= 0.01 , patience= 5,
verbose= 1, mode='auto')

# model check point
mc = ModelCheckpoint(filepath="best_model.h5", monitor= 'val_accuracy',
verbose= 1, save_best_only= True, mode = 'auto')

# puting call back in a list
call_back = [es, mc]
hist = model.fit_generator(train_data,
                           steps_per_epoch= 10,
                           epochs= 30,
                           validation_data= val_data,
                           validation_steps= 8,
                           callbacks=[es,mc])
# Loading the best fit model
from keras.models import load_model
model = load_model("/content/best_model.h5")
h =  hist.history
h.keys()
plt.plot(h['accuracy'])
plt.plot(h['val_accuracy'] , c = "red")
plt.title("acc vs v-acc")
plt.show()
```




```
plt.plot(h['loss'])
plt.plot(h['val_loss'] , c = "red")
plt.title("loss vs v-loss")
plt.show()
```




```
# just to map o/p values
op = dict(zip( train_data.class_indices.values(),
train_data.class_indices.keys()))
```

Using Camera Capture:

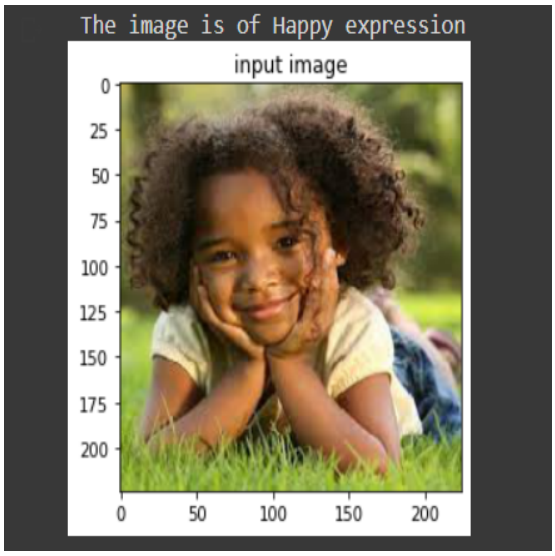
```
from IPython.display import display, Javascript
from google.colab.output import eval_js
from base64 import b64decode

def take_photo(filename='photo.jpg', quality=0.8):
  js = Javascript('''
    async function takePhoto(quality) {
      const div = document.createElement('div');
      const capture = document.createElement('button');
      capture.textContent = 'Capture';
      div.appendChild(capture);

      const video = document.createElement('video');
      video.style.display = 'block';
      const stream = await navigator.mediaDevices.getUserMedia({video: true});

      document.body.appendChild(div);
      div.appendChild(video);
      video.srcObject = stream;
      await video.play();

      // Resize the output to fit the video element.
      google.colab.output.setIframeHeight(document.documentElement.scrollHeight, true);

      // Wait for Capture to be clicked.
      await new Promise((resolve) => capture.onclick = resolve);

      const canvas = document.createElement('canvas');
      canvas.width = video.videoWidth;
      canvas.height = video.videoHeight;
      canvas.getContext('2d').drawImage(video, 0, 0);
      stream.getVideoTracks()[0].stop();
      div.remove();
      return canvas.toDataURL('image/jpeg', quality);
    }
    ''')
  display(js)
  data = eval_js('takePhoto({})'.format(quality))
  binary = b64decode(data.split(',')[1])
  with open(filename, 'wb') as f:
   f.write(binary)
  return filename
```

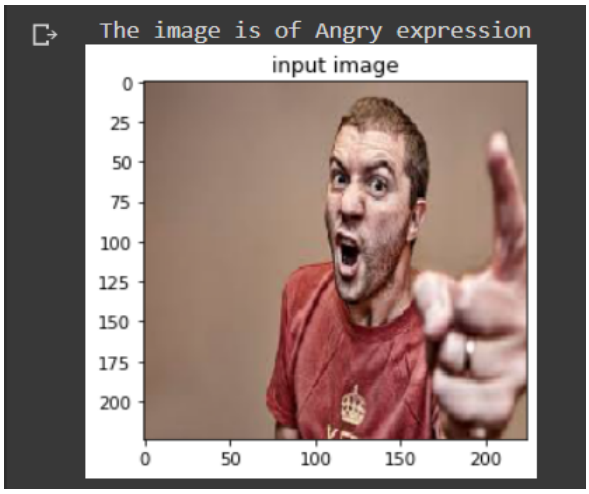
```
from IPython.display import Image
try:
  filename = take_photo()
  print('Saved to {}'.format(filename))

  # Show the image which was just taken.
  display(Image(filename))
except Exception as err:
  # Errors will be thrown if the user does not have a webcam or if they do not
  # grant the page permission to access it.
  print(str(err))
```

Using an Inbuilt image from the library:

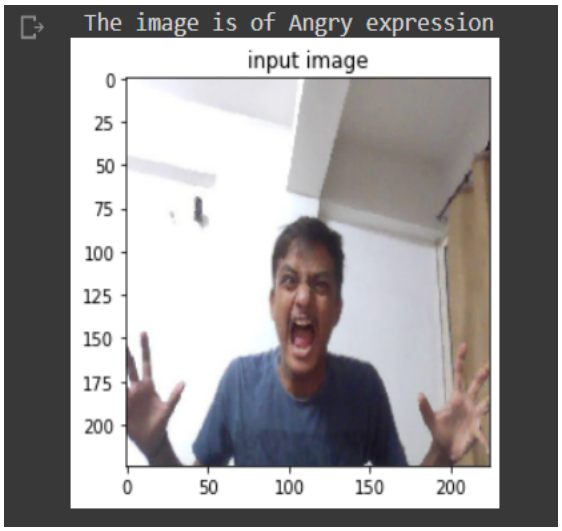
```
# path for the image to see if it predics correct class

path = "/content/photo.jpg"
img = load_img(path, target_size=(224,224) )

i = img_to_array(img)/255
input_arr = np.array([i])
input_arr.shape

pred = np.argmax(model.predict(input_arr))

print(f" The image is of {op[pred]} expression")

# to display the image
plt.imshow(input_arr[0])
plt.title("input image")
plt.show()
```

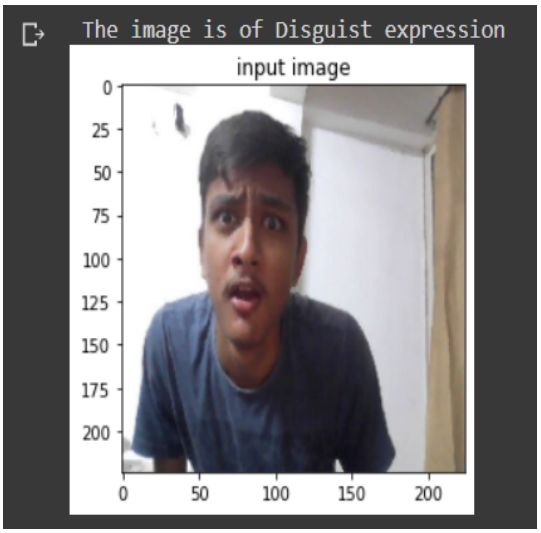
Some examples of emotion recognition are as under:

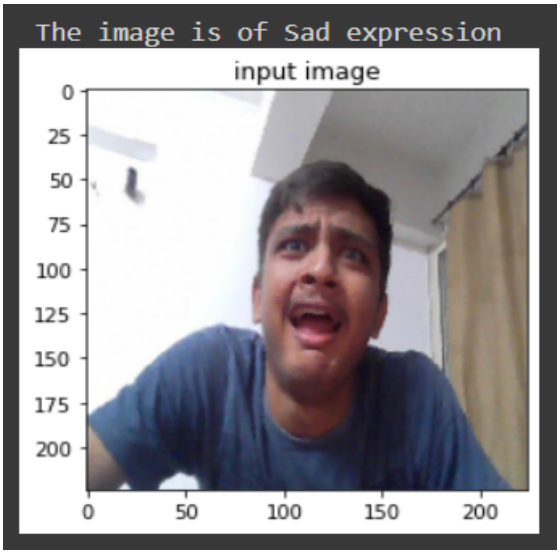


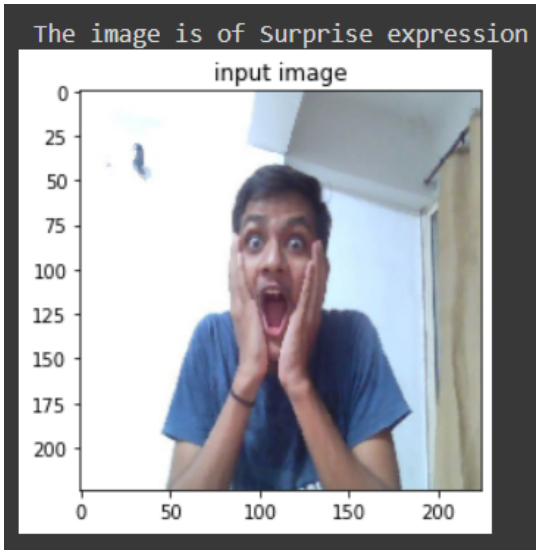


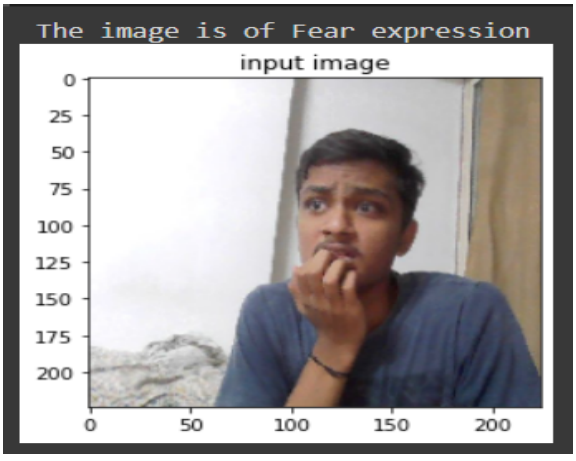
Some in-house examples:











References:

- **“Let.” n.d. Accessed July 31, 2022. https://towardsdatascience.com/lets-code-convolutional-neural-network-in-plain-numpy-ce48e732f5d5.**
- **“Creating Your Own Emotion Recognition Model | Towards AI.” n.d. Accessed July 31, 2022. https://pub.towardsai.net/step-by-step-guide-in-creating-your-own-emotion-recognition-system-b8aba98134c8.**
- **“Emotion Recognition With Deep Learning On Google Colab.” n.d. Accessed July 31, 2022. https://www.clairvoyant.ai/blog/emotion-recognition-with-deep-learning-on-google-colab.**
- **“Emotion Detection Using OpenCV & Python | Edureka - YouTube.” n.d. Accessed July 31, 2022. https://www.youtube.com/watch?v=G1Uhs6NVi-M.**
- **“Four-Layer ConvNet to Facial Emotion Recognition With Minimal ..” n.d. Accessed July 31, 2022. https://www.nature.com/articles/s41598-022-11173-0.pdf?origin=ppub**
- **“Emotion Recognition Using Convolutional Neural Network (CNN ..” n.d. Accessed July 31, 2022. https://iopscience.iop.org/article/10.1088/1742-6596/1962/1/012040/meta.**
- **“Facial Emotion Recognition Using Convolutional Neural Networks ..” n.d. Accessed July 31, 2022. https://www.authorea.com/users/254699/articles/347940-facial-emotion-recognition-using-convolutional-neural-networks-ferc.**
- **Paper-30-01-2019.pdf (bmsce.ac.in)**