Project 4

Mrugesh J. Shah
117074109

A. JAMES CLARK
SCHOOL OF ENGINEERING

ENPM 673

—

PERCEPTION FOR

AUTONOMOUS ROBOTS

—

Dr. Mohammed Samer Charifa

Contents

Problem 1

The first problem primarily deals with the optical flow and implementation of the Lucas-Kanade optical flow estimation technique. The problem is divided into two parts which are explained individually as follows.

1.1 Part 1

1.1.1 Introduction

The task here is to detect the optical flow, with the use of Lucas-Kanade optical flow estimator. In this part we need to implement this detector on the video of cars moving on highway, which is recorded using a stationary camera and the lightning conditions are kept constant. We have to plot the vector field of the frame. Each vector has to be 25 pixels apart.

Any built-in function is allowed to use for implementation.

1.1.2 Lucas-Kanade optical flow estimator

Lucas-Kanade method for optical flow estimation was invented by Bruce D. Lucas and Takeo Kanade. This method works on few assumptions which are as follows

1. Lightning condition do NOT change over time
2. Video has enough features to detect

Now, to understand why these assumptions matter, we need to understand how this flow estimation works. The Lucas–Kanade method assumes that the displacement of the image contents between two nearby instants (frames) is small and approximately constant within a neighborhood of the point P under consideration. Thus the optical flow equation can be assumed to hold for all pixels within a window centered at p. Namely, the local image flow (velocity) vector (V_x, V_y) must satisfy the following equation.

$$\begin{gathered} I_x(q1)V_x + I_y(q1)V_y = -I_t(q1) \\ I_x(q2)V_x + I_y(q2)V_y = -I_t(q2) \\ \dots \\ I_x(qn)V_x + I_y(qn)V_y = -I_t(qn) \end{gathered}$$

where $q_1,q_2,\dots,q_n$ are the pixels inside the window, and $I_x(q_i)$, $I_y(q_i)$, $I_t(q_i)$ are the partial derivatives of the image I with respect to position x, y and time t, evaluated at the point q_i and at the current time.

These equations can be written in matrix form Av=b, where

$$A = \begin{bmatrix} I_x(q1) & I_y(q1) \\ \vdots & \vdots \\ I_x(qn) & I_y(qn) \end{bmatrix}, V = \begin{bmatrix} V_x \\ V_y \end{bmatrix}, b = \begin{bmatrix} -I_t(q1) \\ \vdots \\ -I_t(qn) \end{bmatrix}$$

Since we have more equations than the unknowns, it is an over determinant system. We can use the Least Squares (LS) method to fit the values of V vector. This will give us the vector of the flow direction.

Now that we know that the LK (Lucas-Kanade) method works on the intensity parameter. It checks where did that pixel or kernel moved in the frame (or threshold). Now, since the method uses intensity to check the new frame location, if the lightning condition change, it will not be able to detect the same kernel in the next frame.

Also, If the video capturing device is moving, it will create the false movement, which in this application is not important and may cause false vectors.

1.1.3 Pipeline

First, we have to load the video frame by frame. And since the Lukas-Kanade method needs the previous frame to compare it with the current frame. We need to initiate the program by loading the old frame as the first frame. Thus, it will not throw an error and the frame has to be in grayscale for it to be processed by proceeding functions.

```
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Also, we can blur the image with median blur filter to remove the minor or false flows because of the vibrations of the camera.

```
frame = cv2.medianBlur(frame, 7)
```

After the first pair of frames, we can see the results, as when we detect optical flow between two same frame and plot the vector field, it will not have any vectors. In second iteration, first and second frames are compared. These two frames will be compared to get the movement of pixels. By the parameters set by the LKParam dictionary. The Lucas-Kanade method is built into the OpenCV library and can be implemented using following figure.

```
lk_param = dict(winSize=(45, 45),
                maxLevel=1,
                criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
10, 0.03))
```

```
new_points, status, error = cv2.calcOpticalFlowPyrLK(old_frame, frame,
points, None, **lk_param)
```

We will give the set of points that are 25 pixels apart, and it will give the new location of these points in the frame. These are stored as new points. Now, we have to draw the arrows that illustrate the vector field over the frame. Which can be done with the help of built-in function from the OpenCV library.

```
frame = cv2.arrowedLine(frame, points_, new_points__, (0, 0, 255), 2,
tipLength=0.5)
```

1.1.4 Results

The following Figure 1 and 2. Shows the vector field of the optical flow, the vectors are nonzero at the places where there is a car moving. All other places have zero vectors i.e., points.

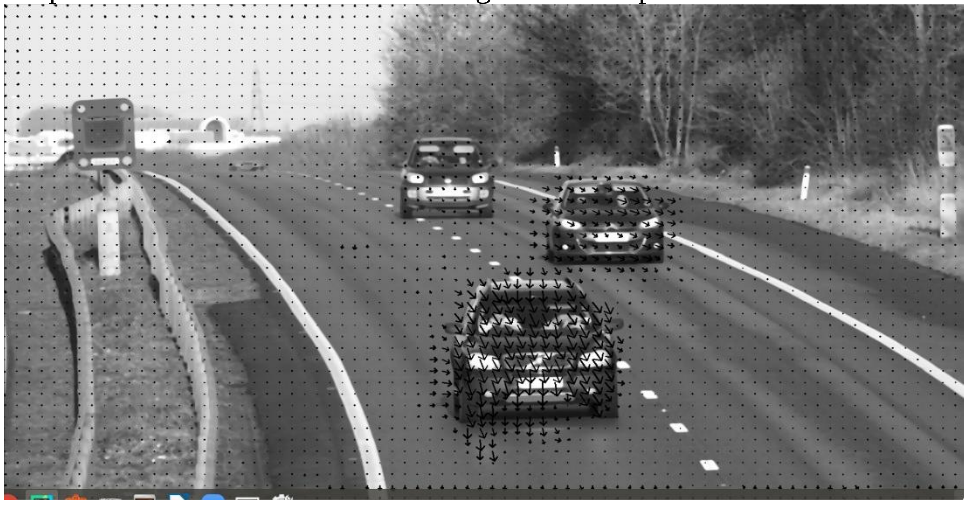


Figure 1. Vector Field 1

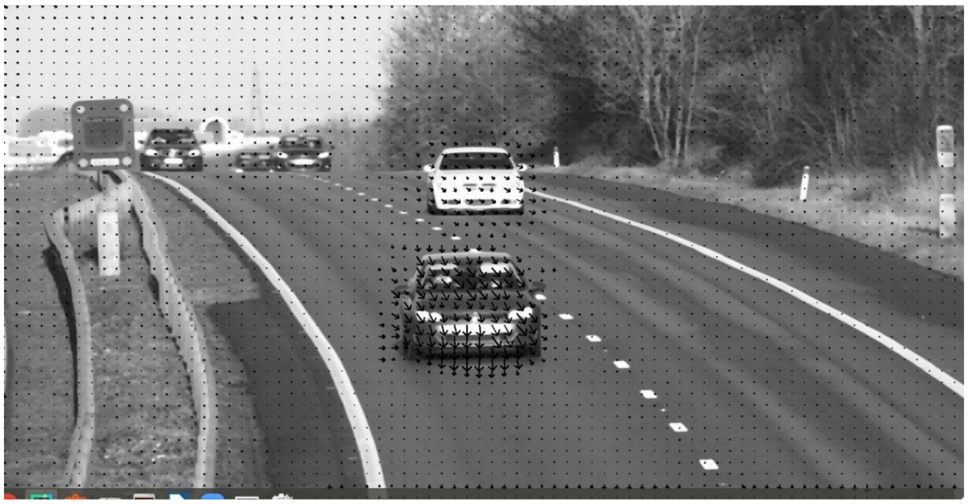


Figure 2. Vector Field 2

We can conclude that the moving car will have the vectors in those points and all the other places will have a zero vector.

1.2 Part 2

1.2.1 Introduction

In this part, we have to use the data we got from the part 1 and show only the moving cars and remove the static background.

1.2.2 Pipeline

We can use the data of the pixel locations which are moved, and according to the parameters we set in the LKParam dictionary, to get the size of the mask that we will impose on the pixel that is having movement.

```
mask[int(moving_points[i][1])-12:int(moving_points[i][1]+12),
int(moving_points[i][0])-12:int(moving_points[i][0])+12] = 255
```

Here we have created a mask for the compete frame, by making the kernel sized pixels white where the program has detected the movement.

```
frame = cv2.bitwise_or(frame, frame, mask=mask)
frame = cv2.resize(frame, (540, 450))
```

This piece of code will be doing the logical operation on the image to generate the final output which can be seen in the results.

1.2.3 Result

Figure 1 and 2. Will show the expel of this masking technique with the help of the Lucas-Kanade flow detection and masking the are where there is movement or has nonzero vectors in vector field,

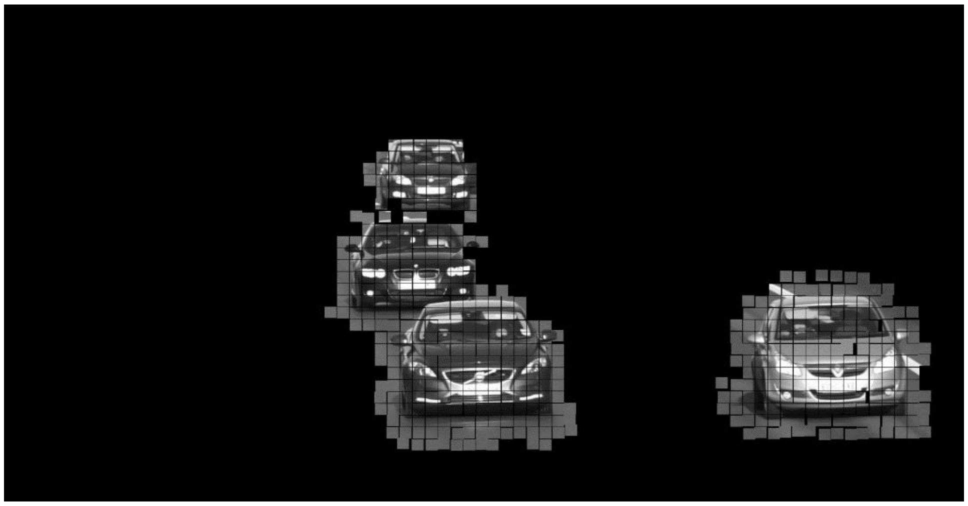


Figure 2. Background Elimination 1

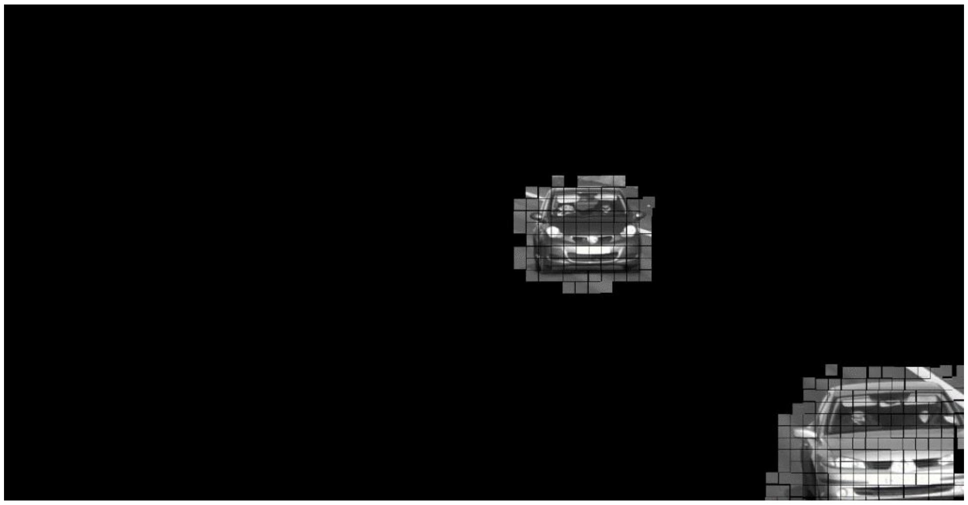


Figure 3. Background Elimination 2

Only the moving elements will not be masked out. The masking kernel can be enlarged to get better results but there is a tradeoff between the clarity of the output and making elimination of the background.

Problem 2

2.1.1 Introduction

Here we have to use Convolutional Neural Network (CNN), create our own model and then train it over the data set provided.

2.1.2 What is CNN?

CNN stands for Convolutional Neural Network, which is a class of deep neural network that is mostly applied to the visual imagery. It is important to note that it is shift invariant and space invariant. Figure 4. Shows the generic CNN architecture.

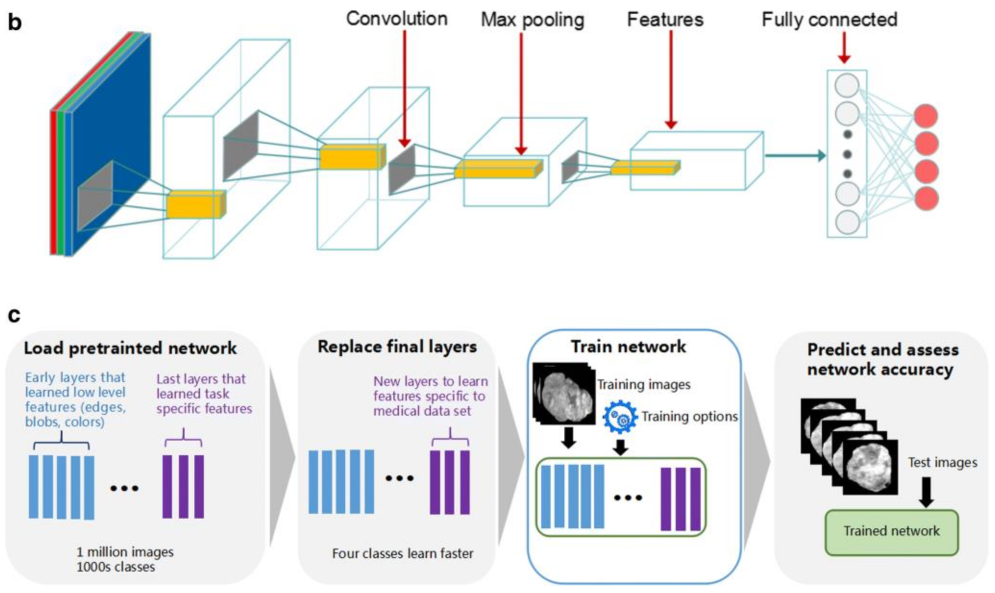

b
Convolution
Max pooling
Features
Fully connected
c
Load pretrainted network
Early layers that learned low level features (edges, blobs, colors)
Last layers that learned task specific features
1 million images 1000s classes
Replace final layers
New layers to learn features specific to medical data set
Four classes learn faster
Train network
Training images
Training options
Predict and assess network accuracy
Test images
Trained network

Figure 4. Generic CNN layers

The CNN has layers, sometimes referred to as convolutional blocks such as 2D convolution, Max pooling, and ReLU. Each model in CNN has its architecture which is just the order of adding the blocks. Each model can have multiple layers in different order.

The one of the most crucial layer is the convolutional layer, it will hove the kernel of the feature over the given image and generate the feature map. Note that we some times use the padding to preserve the dimensions of the image. Figure 5. Shown below explains this layer in graphical manner.

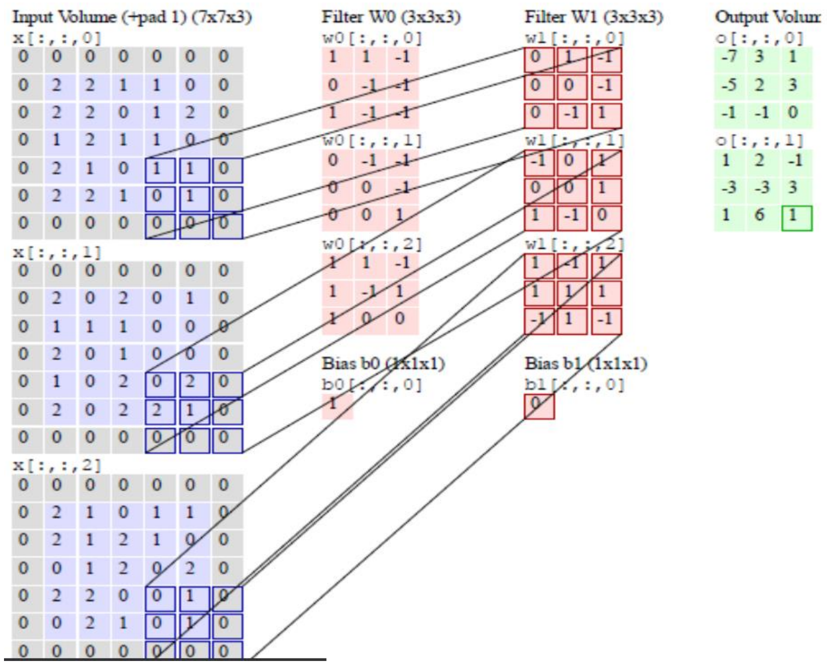


Figure 5. Convolution

We can see that there are some negative elements in the image kernel that is generated after the convolution, we need to make sure that these negative values are zero, and for that there is a layer named ReLU layer. Which will make all the negative values to zero and keep the positive values same as earlier. This will introduce nonlinearity to the images. Below is the visual representation of the implementation of the ReLU on a sample image. Figure 6. Shows the visual representation of affect of implementing ReLU on the image.

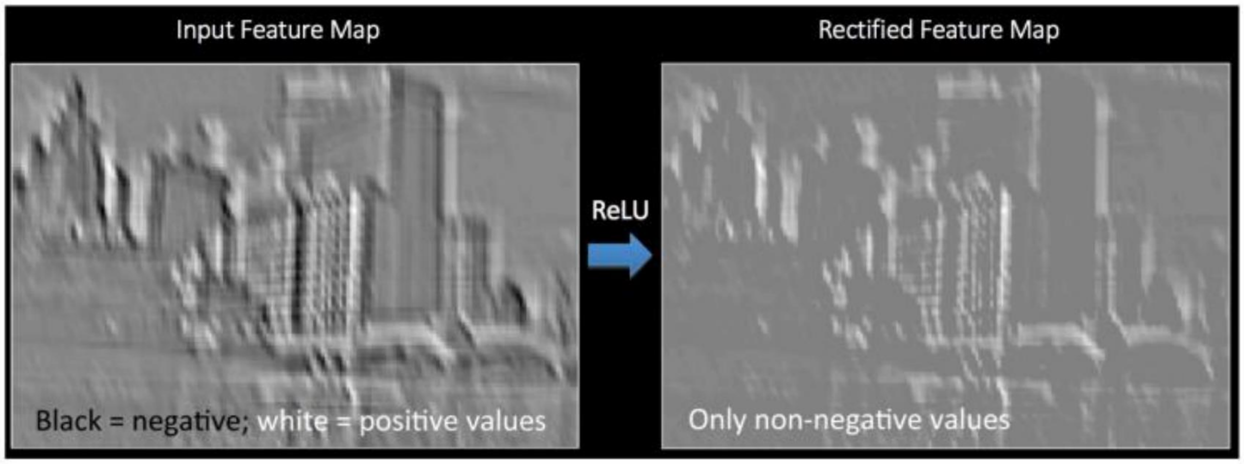


Figure 6. ReLU

Although this will make the CNN mode more efficient, but the time taken by the processor or GPU to process this is much larger. Thus, we need to reduce the size of this image. There is a layer CALLED Max Pooling. Which will be taking the maximum of the kernel size we have defined. We can see in the following image that how the first iteration took the value 6 and placed it into the new matrix. Apart from this there is soft max, which will be based on the probability of each case. Figure 7. Shows how the Max Pooling works on the image frame pixels and how it reduces the image size.

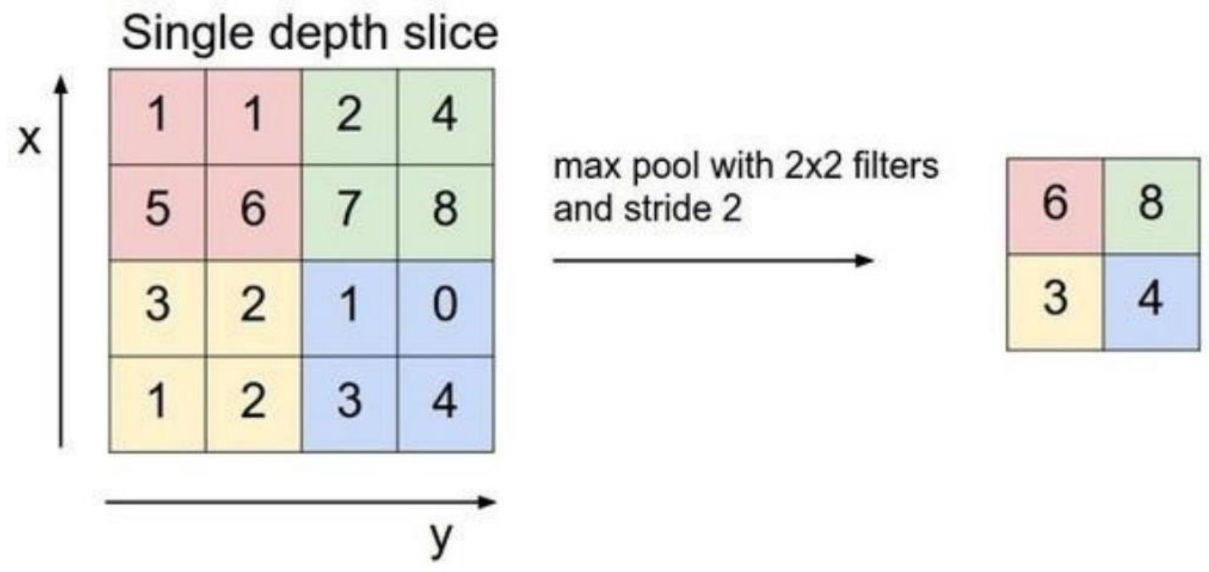


Figure 7. Max Pooling

The convolution is done for each feature and then a separate channel is generated as shown in the figure above. To reduce the processing power, we have to do max pooling and sub sample the data. There is also a ReLU layer which will assign negative layers to zero and introduce nonlinearity to the model. This is sone several times and then finally we have to make the data in 1D so that we can classify it. The posses of converting it into a 1 dimension is called flattening which is usually the last step of the CNN.

2.1.2 VGG-16

VGG16 is a convolutional neural network (CNN) architecture that won the 2014 ILSVR(Imagenet) competition. It is regarded as one of the best vision model architectures ever developed. The most distinguishing feature of VGG16 is that instead of making a large number of hyper-parameters, they concentrated on having 3x3 filter convolution layers with a stride 1 and still used the same padding and maxpool layout with the stride 2. Throughout the design, the convolution and max pool layers are arranged in the same way. It has two FC (fully connected layers) at the top, followed by a SoftMax for output. The 16 in VGG16 refers to the fact that it has 16 layers of different weights. This network is very wide, with approximately 138 million (approx) parameters. Figure 8. Is the visual representation of the VGG-16 architecture and how it changes the number of channels as well as the size of the image frame after the frame passes through its layers. It also shows that the soft pooling in the end on flattened layer generates the probability of each class as prediction.

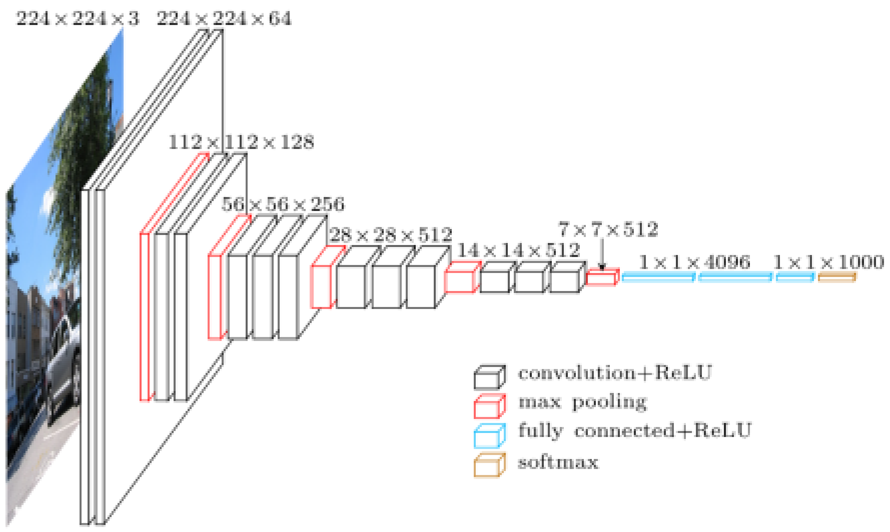


Figure 8. VGG-16 Architecture

Here we have taken an example of the 244x244 image with 3 channels R, G, and B. The key factor to consider here is that how max pooling reduces the size of the image frame. This will make the computation less intensive and can be computed easily. Also, the flattening is done in the end of all the layers. The final layer will be the soft pool, which will be taking the data received by the 1D layer and generate probability of each class. We have to predict the output that as highest probability.

2.1.3 Pipeline

The first step is to load the data set. I have used the Kaggle as cloud computing IDE. But there are several other such as Google Collab and Azure Notebook, and AWS. The first step is to preprocess the data. First, we have to know the number of classes and data set images' resolution to make a model that fits the data sets as input. We have 9 classes, and each image has 100x100 resolution. We have to setup the data by zipping the data and labels. After that, we have to split the data set into two parts namely Training and Testing. Since a well-trained model also splits the training data into two categories namely training and validation. Thus, we have created three parts of the data sets namely Training, Validating, Testing.

```
Train_set, test_set = train_test_split(df, test_size = 0.3, random_state = 42)
train_set, val_set = train_test_split(Train_set, test_size= 0.2, random_state = 42)
```

After this, we have to make data set more versatile by rotating the dataset image in different angles and flipping it as well to make sure that the trained model will not be limited to get accurate predictions in only one orientation.

```
img_gen = ImageDataGenerator(preprocessing_function =
tf.keras.applications.mobilenet_v2.preprocess_input, rescale=1/255)
```

I then created the architecture or the framework of the model and then stored it into the model variable. Then on that model I used the altered test data to train it and fed the training data set as well. There were 15 epochs as increasing the epochs to insanely large number will not only increase the computation time but also make the mode overfit and will perform much poorly in real world data.

The model designing can done using following piece of code for each layer of the model.

```
model.add(tf.keras.layers.Conv2D(filters = 32, kernel_size=(3, 3), strides=(1, 1),
    activation='relu', padding='valid', input_shape = input_shape))
```
(The convolution Layer)

```
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
```
(Max Pooling Layer)

```
model.add(tf.keras.layers.Dense(64, activation='relu'))
```
(ReLU layer)

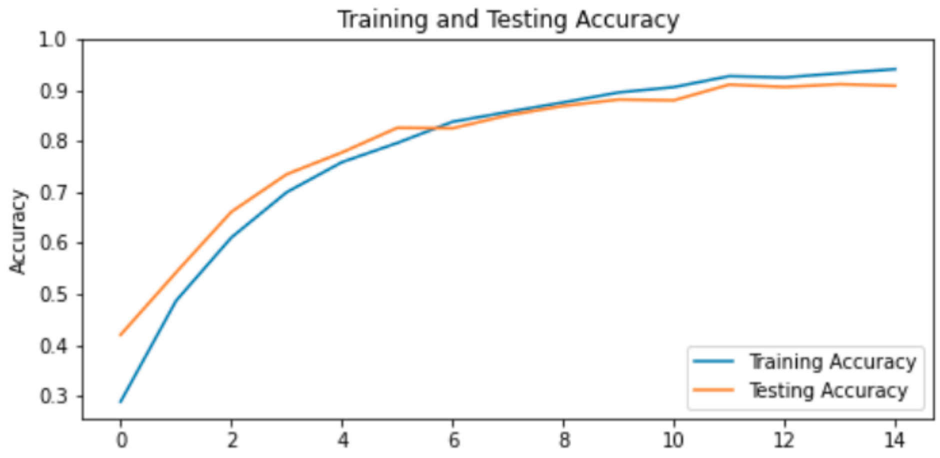
```
model.add(tf.keras.layers.Flatten())
```
(Flattening of image kernel layer)

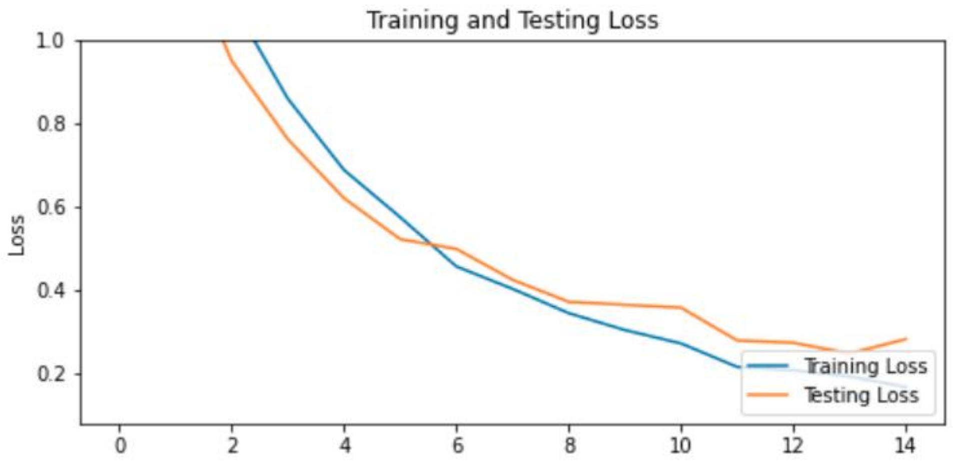
We have added these layers into our model as described in the architecture keeping in mind the order as well as the repetition. After that, I had to plot the graphs of the change in Accuracy and Loss v/s Epochs. You can see that all of the functions are the one liners, because I have used the library named Keras for model generation and tensorflow for data set processing as well as sklearn for getting the statistical data to plot.

```
history = model.fit(train,validation_data = val,epochs = 15, verbose = 1)
```
(Getting the model statistics)

2.1.4 Results

The generated graphs are as follows.





We can see that as we add more and more epochs the accuracy of the model increases on both the training and testing data sets. But, as we discussed earlier having insanely large number of epochs does not mean it will be generating a better model even though it increases the accuracy. It is the overfitted model which will perform poorly in the real world scenario.

Media Link

4.1 Drive link

https://drive.google.com/drive/folders/1RxptaIfqH1uRik2RHYy_5GzTlZpJFanC?usp=sharing

References

[1] O.Ulucan, D.Karakaya, and M.Turkan.(2020)
A large-scale dataset for fish segmentation and classification.In Conf. Innovations Intell. Syst. Appli. (ASYU)

[2] Designer, D.
Designer, D. (2021). VGG-16 convolutional neural network - MATLAB vgg16. Retrieved 18 May 2021, from https://www.mathworks.com/help/deeple

[3] Convolutional neural network - Wikipedia
Convolutional neural network - Wikipedia. (2019). Retrieved 18 May 2021, from https://en.wikipedia.org/wiki/Convolutional

[4] OpenCV: Contours Hierarchy
OpenCV: Contours Hierarchy. (2021). Retrieved 8 March 2021, from https://docs.opencv.org/master/d9/d8b/tutorial

[5] Optical flow - Wikipedia
Optical flow - Wikipedia. (2021). Retrieved 18 May 2021, from https://en.wikipedia.org/wiki/Optical_flow

[6] VGG Neural Networks: The Next Step After AlexNet
VGG Neural Networks: The Next Step After AlexNet. (2019). Retrieved 18 May 2021, from https://towardsdatascience.com/vgg-neural-networks-the-next-step-after-alexnet-3f91fa9ffe2c

[7] Vector field - Wikipedia
Vector field - Wikipedia. (2021). Retrieved 18 May 2021, from https://en.wikipedia.org/wiki/Vector_field

[8] Lucas–Kanade method - Wikipedia
Lucas–Kanade method - Wikipedia. (2021). Retrieved 18 May 2021, from https://en.wikipedia.org/wiki/Lucas%E2%

[9] Installation Guide — Matplotlib 3.3.4 documentation
Installation Guide — Matplotlib 3.3.4 documentation. (2021). Retrieved 15 February 2021,
from https://matplotlib.org/stable/users/installing.htm

[10] OpenCV: Introduction to OpenCV-Python Tutorials
OpenCV: Introduction to OpenCV-Python Tutorials. (2021). Retrieved 15 February 2021,

from https://docs.opencv.org/master/d0/de3/tutorial_py_intro.html

[11] OpenCV: Install OpenCV-Python in Ubuntu
OpenCV: Install OpenCV-Python in Ubuntu. (2021). Retrieved 15 February 2021, from https://docs.opencv.org/master/d2/de6/tutorial_py_setup_in_ubuntu.html

[12] Homography (computer vision)
Homography (computer vision). (2021). Retrieved 15 February 2021, from https://en.wikipedia.org/wiki/Homography

[13] Anon
(2021). Retrieved 9 March 2021, from http://iihm.imag.fr/publs/2019/iss_2019_cam