Computer Vision Training Documentation

Table of Contents:

1. Introduction

This documentation provides a comprehensive overview of the computer vision training conducted at Adhiparasakthi Engineering College as part of the Artificial Intelligence (AI). The training aimed to equip students with the knowledge and practical skills to design, implement, and train computer vision models using modern deep learning techniques. The document covers the entire process, from dataset preparation to model training and deployment.

2. Objectives

2.1 Understanding the Fundamentals of Computer Vision and Deep Learning

In this objective, students will gain a solid understanding of the core concepts of computer vision and deep learning. They will learn about image representation, feature extraction, convolutional neural networks (CNNs), and their applications in various computer vision tasks.

Example content:

- Lectures and hands-on sessions covering image representation techniques, such as RGB and grayscale images.
- Explanation of the role of CNNs in computer vision and how they can learn hierarchical features from images.
- Practical exercises to implement basic image processing techniques and visualize feature maps in CNNs

2.2 Learning Data Preprocessing and Augmentation

This objective focuses on data preprocessing techniques and data augmentation, which are crucial for successful model training. Students will learn how to clean and preprocess raw data to ensure data quality and how to augment datasets to improve model generalization.

Example content:

- Instruction on handling missing values, removing outliers, and normalizing data for training.
- Demonstrations of various data augmentation techniques, including random rotation, flipping, and brightness adjustments.
- Hands-on exercises to preprocess and augment images using popular Python libraries like OpenCV and PIL.

2.3 Implementing Computer Vision Models and Transfer Learning

In this objective, students will learn to implement various computer vision models using deep learning frameworks. They will also understand the concept of transfer learning and how to leverage pre-trained models to solve real-world problems effectively.

Example content:

- Detailed explanations of different computer vision models, such as VGG, ResNet, and MobileNet.
- Step-by-step tutorials to build and train these models using popular deep learning frameworks like TensorFlow or PyTorch.

- Practical sessions to fine-tune pre-trained models for specific tasks and datasets.

2.4 Optimizing Model Performance and Deployment

This objective focuses on optimizing the performance of computer vision models and deploying them in real-world applications. Students will learn techniques to improve model efficiency, such as model quantization, and deploy trained models on different platforms.

Example content:

- Discussions on techniques like model pruning and quantization to reduce model size and improve inference speed.
- Demonstrations of deploying computer vision models in web applications or mobile apps using tools like Flask or TensorFlow Lite.
- Practical exercises to evaluate the performance of the deployed models and measure their inference time.
- By achieving these objectives, participants in the computer vision training program will be well-equipped to tackle a wide range of computer vision challenges and develop cutting-edge solutions in their future endeavors.

3. Preparing the Dataset

3.1 Data Collection

The FER2013 dataset is a popular facial expression recognition dataset available on Kaggle. It contains over 35,000 grayscale images of size 48x48 pixels, categorized into seven emotion classes: anger, disgust, fear, happiness, sadness, surprise, and neutral. To obtain this dataset, follow these steps:

1. Go to Kaggle (www.kaggle.com) and create a free account if you don't have one.
2. Search for the "FER2013: Facial Expression Recognition" dataset on Kaggle.
3. Read the dataset's description, terms of use, and download the dataset in CSV format.
4. Extract the downloaded ZIP file to get the CSV file and the images folder containing the facial expression images.

3.2 Data Preprocessing

Once you have downloaded the dataset, you need to preprocess it before using it for training your computer vision model. The preprocessing steps typically include data cleaning, normalization, and splitting.

Data Cleaning:

1. Load the CSV file containing the image labels and emotion classes.
2. Check for any missing or erroneous data entries
3. Handle any missing data appropriately, such as removing or imputing them.

Data Normalization:

1. Convert the grayscale pixel values in the images to a range between 0 and 1.

2. Normalize the pixel values by dividing them by 255 (the maximum pixel value for grayscale images).

Data Splitting:

1. Split the dataset into training, validation, and test sets.
2. Allocate a certain percentage of the data to each set, such as 70% for training, 15% for validation, and 15% for testing.

4. Selecting the Computer Vision Model

Selecting the right computer vision model is crucial for the success of a computer vision task. Each model has its strengths and weaknesses, and choosing the most suitable one depends on factors like task complexity, dataset size, and computational resources. Here are brief descriptions and use cases for four popular computer vision models: CNN, YOLO (You Only Look Once), MediaPipe, and MobileNet.

4.1 Convolutional Neural Network (CNN):

Description:

CNNs are deep learning models designed to process and analyze visual data, particularly images. They use multiple convolutional layers to automatically extract hierarchical features from images.

Use Case:

CNNs are widely used in image classification, object detection, and segmentation tasks. They excel at learning complex patterns in images and are capable of achieving high accuracy in various computer vision tasks.

4.2 YOLO (You Only Look Once):

Description:

YOLO is an object detection algorithm that can detect multiple objects in an image in real-time. Unlike traditional object detectors, YOLO processes the entire image in one pass, making it incredibly fast.

Use Case:

YOLO is commonly used in real-time applications such as surveillance, autonomous vehicles, and robotics, where low latency is critical. It can efficiently detect objects with high accuracy in real-time scenarios.

4.3 MediaPipe:

Description:

MediaPipe is a library developed by Google that provides pre-trained models and building blocks for various computer vision tasks, including facial landmark detection, hand tracking, and pose estimation.

Use Case:

MediaPipe is versatile and can be used in a wide range of applications, from adding AR (Augmented Reality) effects in videos to developing interactive experiences with gesture recognition.

4.4 MobileNet

Description:

MobileNet is a lightweight and efficient deep learning architecture optimized for mobile and embedded devices. It uses depthwise separable convolutions to reduce the number of parameters and computational complexity.

Use Case:

MobileNet is suitable for resource-constrained environments, such as mobile phones and edge devices. It is used in applications where model size and speed are crucial, like image classification on mobile devices.

In summary, selecting the right computer vision model requires careful consideration of the task requirements and available resources. CNNs are a solid choice for various computer vision tasks, while YOLO provides real-time object detection capabilities. MediaPipe offers a collection of pre-trained models for various tasks, and MobileNet is ideal for deployment on mobile and embedded devices with limited resources.

5. Setting up the Training Environment

Setting up the training environment for computer vision tasks involves configuring the necessary hardware, software, and libraries to efficiently train deep learning models. Below is an example of setting up the training environment using Python, TensorFlow, and CUDA for GPU acceleration.

5.1 Hardware Requirements

For a basic computer vision project, you can set up the training environment on a reasonably capable desktop or laptop computer. Here's an example of hardware requirements:

- **Processor**: Intel Core i5 or equivalent (or higher) with at least 4 cores.
- **RAM**: 16 GB or more to handle large datasets and deep learning models effectively.
- **Graphics Card**: NVIDIA GPU (optional but highly recommended) with CUDA support for accelerated deep learning training. Example: NVIDIA GeForce RTX 2060 or higher.
- **Storage**: At least 256 GB SSD for faster data access.

Note: While a GPU is optional, it significantly speeds up the training process for deep learning models. Without a GPU, training deep neural networks can be time-consuming.

5.2 Software Requirements

Here's an example of the software setup using Visual Studio Code and Python 3.10:

- **Operating System**: Windows, macOS, or Linux (choose based on your preference).
- **Visual Studio Code**: A lightweight, open-source code editor. Download and install it from the official website: https://code.visualstudio.com/
- **Python 3.10**: Download and install the latest version of Python 3.10 from the official Python website: https://www.python.org/downloads/
- **Deep Learning Framework**: Install TensorFlow or any other deep learning framework of your choice using pip within the activated virtual environment Example: TensorFlow, PyTorch.
- **IDE (Integrated Development Environment)**: An IDE provides an efficient coding environment with code completion, debugging, and version control support.Example: Visual Studio Code, PyCharm.

- **Additional Libraries**: Install additional Python libraries required for data manipulation, visualization, and other specific tasks. Example: OpenCV, NumPy, Matplotlib.

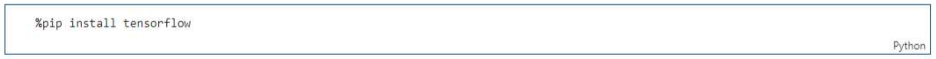
5.3 Framework and Libraries

Selecting the right deep learning framework and utilizing relevant libraries are essential for the successful implementation of computer vision tasks. Below are popular frameworks and libraries commonly used in computer vision projects:

5.3.1 Deep Learning Frameworks:

1. **TensorFlow**:

    - Description: TensorFlow is an open-source deep learning library developed by Google. It provides a flexible ecosystem for building and training machine learning and deep learning models, including computer vision models.



```
%pip install tensorflow
```

    - Use Case: TensorFlow is widely used in computer vision tasks, such as image classification, object detection, and segmentation, due to its scalability and excellent support for GPUs.

2. **PyTorch**:

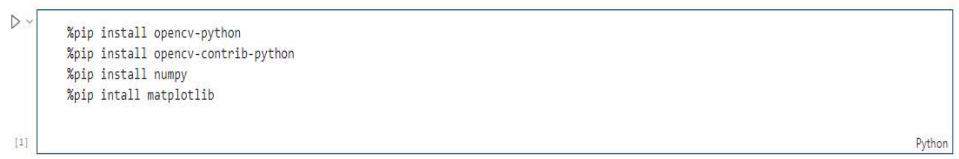
    - Description: PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It provides dynamic computation graphs and is highly popular among researchers and developers.
    - Use Case: PyTorch is often preferred for research-based computer vision projects, as it offers an intuitive and Pythonic syntax and extensive support for complex model architectures.

5.3.2 Computer Vision Libraries

1. **OpenCV (Open Source Computer Vision Library)**:

    ✓ Description: OpenCV is a popular open-source library for computer vision and image processing tasks. It offers a wide range of functionalities, such as image and video processing, object detection, and feature extraction.

    ✓ Use Case: OpenCV is widely used for basic image processing operations, like resizing, cropping, and filtering, as well as for more advanced tasks, such as face detection and recognition.



```
%pip install opencv-python
%pip install opencv-contrib-python
%pip install numpy
%pip intall matplotlib
```

2. **PIL (Python Imaging Library)**:

    ✓ Description: PIL is a library for opening, manipulating, and saving various image file formats. It is often used in conjunction with other libraries for handling image data in Python.

- ✓ Use Case: PIL is helpful for reading and preprocessing image data before feeding it into deep learning models

3. **Matplotlib**:

   - ✓ Description: Matplotlib is a popular data visualization library in Python. It is commonly used to visualize and plot images, training metrics, and model predictions during the development process.

   - ✓ Use Case: Matplotlib helps to visualize and analyze model performance, identify potential issues, and make data-driven decisions during model training.

5.3.3 Additional Libraries:

1. **NumPy**:

   - Description: NumPy is a fundamental library for numerical computations in Python. It is widely used for handling multi-dimensional arrays and matrices, which are prevalent data structures in computer vision tasks.

   - Use Case: NumPy is essential for data manipulation, feature extraction, and data preprocessing in computer vision projects.

2. **scikit-image**:

   - Description: scikit-image is an image processing library built on top of NumPy. It provides various tools and algorithms for image analysis and manipulation.

   - Use Case: scikit-image complements OpenCV and provides additional image processing functionalities for computer vision tasks.

Choosing the right framework and libraries largely depends on your project requirements, familiarity with the tools, and specific tasks you aim to accomplish. TensorFlow and PyTorch are two powerful frameworks that cover most computer vision needs, while OpenCV and PIL offer robust image processing capabilities. NumPy and Matplotlib are essential libraries for data manipulation and visualization in computer vision projects

6. Model Architecture and Implementation

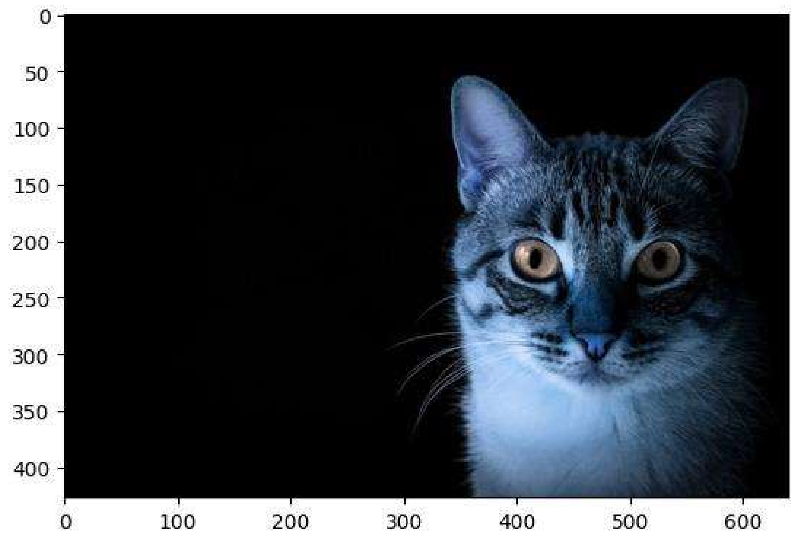
In this section, we will discuss the model architecture and implementation for face detection using the OpenCV (cv2), NumPy, and Matplotlib libraries.

6.1 Basic of image processing

The code provided loads an image using the **cv2.imread()** function and then prints the shape of the loaded image. Let's go through each step and explain the meaning of the image shape:

```
import cv2 as cv
import matplotlib.pyplot as plt
img=cv.imread('C:/Users /Desktop/ComputerVision/Resources/Photos/cat)
plt.imshow(img)
```

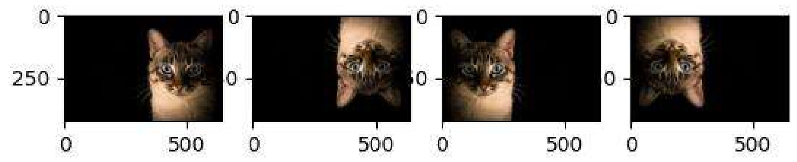
Output:



Rotate:

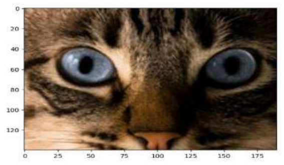
```
img=cv.imread('C:/Users/Desktop/ComputerVision/Resources/Photos/cat.jpg')
img=cv.cvtColor(img,cv.COLOR_BGR2RGB)
fig,axes=plt.subplots(nrows=1,ncols=4)
axes[0].imshow(img)
axes[1].imshow(cv.flip(img,0))
axes[2].imshow(cv.flip(img,1))
axes[3].imshow(cv.flip(img,-1))
```

Output:



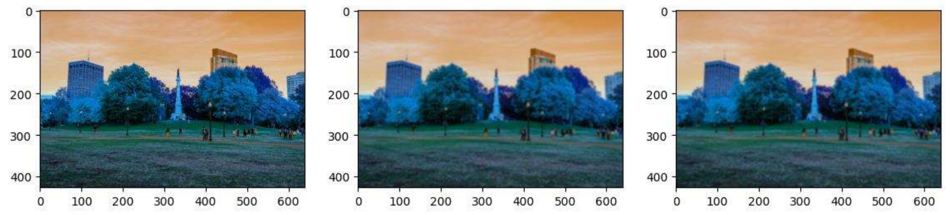
```
#CROP
plt.imshow(img[160:300,370:560])
```

Output:



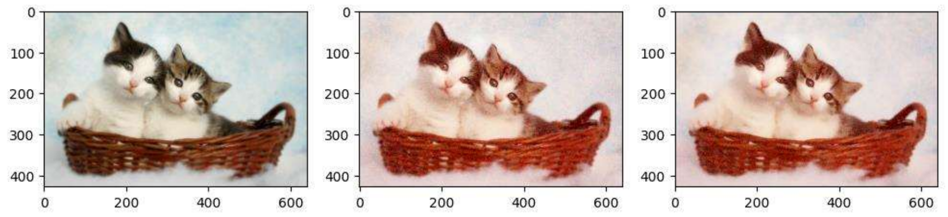
```
img=cv.imread('C:/Users/Desktop/ComputerVision/Resources/Photos/park.jpg')
img_rgb=cv.cvtColor(img,cv.COLOR_BGR2RGB)
blurred=cv.blur(img,(5,5))
g_blurred=cv.GaussianBlur(img,(5,5),10)
fig,axes=plt.subplots(nrows=1,ncols=3,figsize=(14,14))
axes[0].imshow(img)
axes[1].imshow(blurred)
axes[2].imshow(g_blurred)
```

Output:



```
import numpy as np
img=cv.imread('C:/Users/Desktop/ComputerVision/Resources/Photos/cats 2.jpg')
img=cv.cvtColor(img,cv.COLOR_BGR2RGB)
noise=np.zeros(img.shape,np.uint8)
cv.randn(noise,0,180)
noisy_img=cv.add(img,noise)
fig,axes=plt.subplots(nrows=1,ncols=3,figsize=(12,12))
axes[0].imshow(img)
axes[1].imshow(noisy_img)
noise_removed=cv.bilateralFilter(noisy_img,10,35,25)
axes[2].imshow(noise_removed)
```

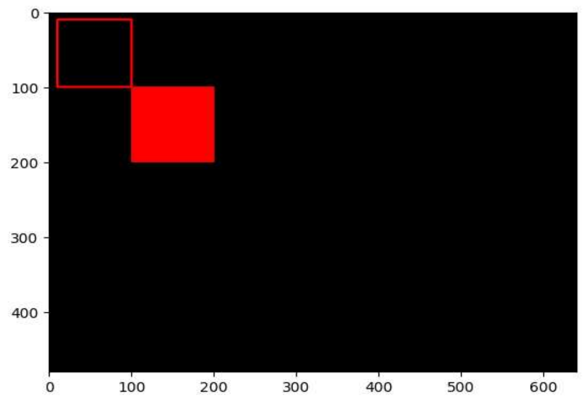
Output:



Making square in a blank space:

```
blank=np.zeros((480,640),np.uint8)
blank=cv.cvtColor(blank,cv.COLOR_BGR2RGB)
cv.rectangle(blank,(10,10),(100,100),(255,0,0),2)
cv.rectangle(blank,(100,100),(200,200),(255,0,0),-1)
plt.imshow(blank)
```

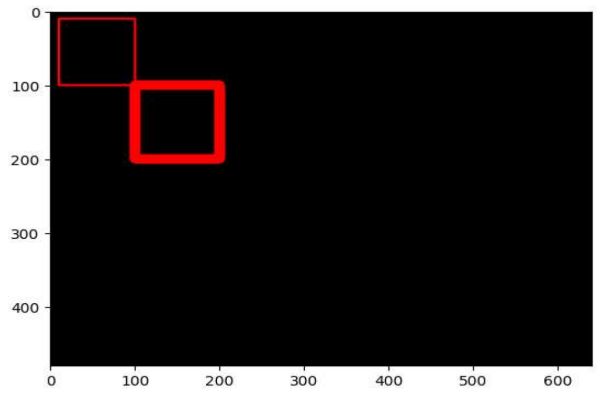
Output:



Program:

```
import numpy as np
blank=np.zeros((480,640),np.uint8)
blank=cv.cvtColor(blank,cv.COLOR_BGR2RGB)
cv.rectangle(blank,(10,10),(100,100),(255,0,0),2)
cv.rectangle(blank,(100,100),(200,200),(255,0,0),11)
plt.imshow(blank)
```

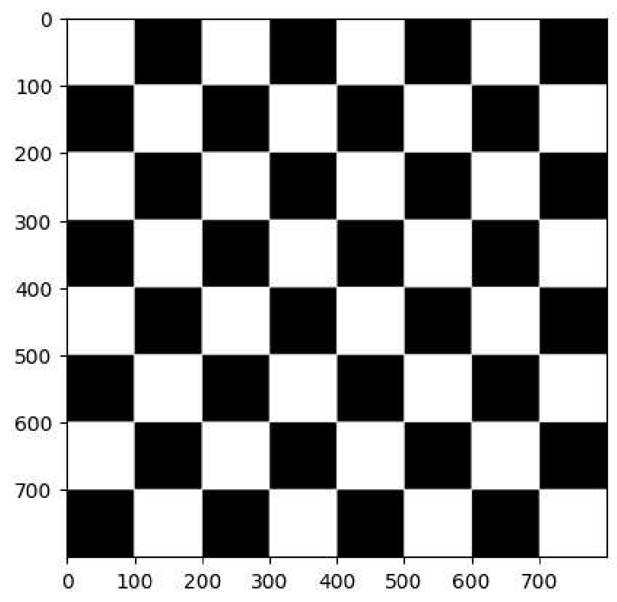
Output:



Program for creating a chessboard:

```
nrows=8
ncols=8
COLOR=(255,255,255)
SQ_SIZE=100
blank=np.zeros((800,800),np.uint8)
blank=cv.cvtColor(blank,cv.COLOR_BGR2RGB)
for r in range(nrows):
    if(r%2)==0:
        IS_WHITE=True
    else:
        IS_WHITE=False\

    for c in range(ncols):
        if IS_WHITE:
            sp=(SQ_SIZE*r,SQ_SIZE*c)
            ep=(SQ_SIZE*(r+1),SQ_SIZE*(c+1))
            cv.rectangle(blank,sp,ep,COLOR,-1)
            IS_WHITE=False
        else:
            IS_WHITE=True
plt.imshow(blank)
```

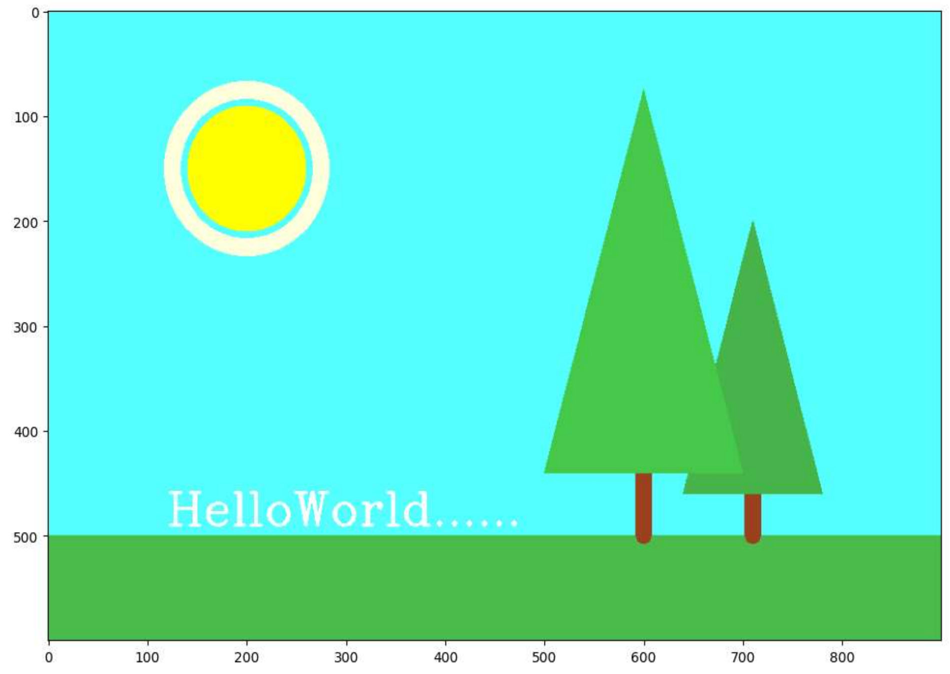
Output:



Painting in a image:

```
H,W =600,900
img=np.zeros((H,W,3),dtype=np.uint8)
img=cv.cvtColor(img,cv.COLOR_BGR2RGB)
cv.rectangle(img,(0,0),(900,500),(85,255,255),-1)
cv.rectangle(img,(0,500),(900,600),(75,188,75),-1)
cv.circle(img, (200,150),60,(255,255,0),-1)
cv.circle(img, (200,150),75,(255,255,220),15)
cv.line(img,(710,500),(710,420),(155,65,30),15)
pts=np.array([[640,460],[780,460],[710,200]],dtype=np.int32)
cv.fillPoly(img,[pts],(70,180,75))
cv.line(img,(600,500),(600,420),(155,65,30),15)
pts=np.array([[500,440],[700,440],[600,75]],dtype=np.int32)
cv.fillPoly(img,[pts],(70,200,75))
cv.putText(img,'HelloWorld......',(120,490),cv.FONT_HERSHEY_COMPLEX,1.5,(255,2
55,255),2)
plt.figure(figsize=(12,12))
plt.imshow(img)
```

Output:



Haar Cascade Classifier

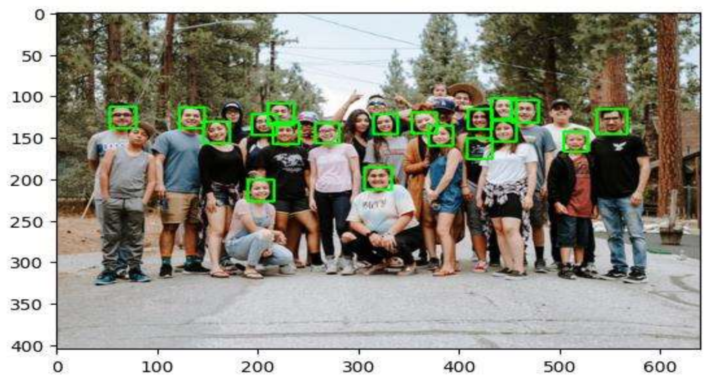
The Haar Cascade classifier is an effective object detection method based on machine learning. It uses Haar-like features and a machine learning algorithm to detect objects of interest. For face detection, the classifier has been trained to identify facial features based on the Haar-like features.

Importing Libraries: First, we need to import the required libraries for face detection. We will be using OpenCV (cv2) for image processing, NumPy for numerical computations, and Matplotlib for visualization.

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
```

Here, the code reads an image named 'group 1.jpg' from the 'Resources/Photos/' directory using the **cv.imread()** function. The image is loaded as a three-channel BGR (Blue-Green-Red) image. Next, the code converts the loaded BGR image to a grayscale image using the **cv.cvtColor()** function. Grayscale images have a single channel, which simplifies further processing. The **cv.detectMultiScale()** function takes the grayscale image as input along with two parameters: **scaleFactor** and **minNeighbors**. The **scaleFactor** controls the size of the image pyramid, and **minNeighbors** specifies the number of neighbors a candidate rectangle should have to be considered a face. Smaller values of **minNeighbors** will result in more detections but can also include false positives.

```
import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt


img=cv.imread('Resources/Photos/group 1.jpg')
img_gray=cv.cvtColor(img,cv.COLOR_BGR2GRAY)
haar_cascade=cv.CascadeClassifier('Resources/Config/haar_face.xml')
detected_faces=haar_cascade.detectMultiScale(img_gray,scaleFactor=1.1,minNeigh
bors=1)
for (x,y,w,h) in detected_faces:
    cv.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
plt.imshow(cv.cvtColor(img, cv.COLOR_BGR2RGB))
```



Face Emotion Detection with MTCNN:

MTCNN is a widely used deep learning model for face detection. It can detect faces in images and provide bounding boxes around the detected faces. MTCNN is a cascaded model with three stages:

a. **Stage 1 (Proposal Network)**: In this stage, the model proposes candidate regions likely to contain faces. It uses a set of bounding box regression and facial landmark localization to refine the proposals.

b. **Stage 2 (Refine Network)**: This stage refines the candidate regions proposed by the first stage. It rejects false positives and further adjusts the bounding boxes.

c. **Stage 3 (Output Network)**: The final stage filters the detected faces, discarding overlapping or redundant detections, and outputs the final bounding boxes of the detected faces along with facial landmark points

Emotion Recognition using MobileNet_7:

- MobileNet is a lightweight deep learning architecture designed for mobile and embedded vision applications. MobileNet_7 refers to a variant of MobileNet with seven layers
- The face emotion recognition component uses MobileNet_7 to classify emotions from the detected face regions. After obtaining the bounding box coordinates of the detected faces from MTCNN, these regions are cropped and resized to a fixed size suitable for MobileNet_7.
- The MobileNet_7 model is then applied to the cropped face images to predict the emotions. The output of the MobileNet_7 model will be a probability distribution over different emotions, such as happy, sad, angry, surprised, etc.

Emotion Recognition:

The extracted features from the MobileNet_7 model are used as input to an emotion recognition model.

This model is typically a deep learning classifier trained on a dataset of labeled facial expressions, containing emotions like happy, sad, angry, etc.

The classifier's task is to predict the emotion category based on the features extracted from the face.

Inference and Output:

After the emotion recognition model processes the extracted features, it outputs a prediction for the emotion expressed in the detected face. The model's output may include probabilities for each emotion category, and the highest probability is usually considered as the predicted emotion for that face.

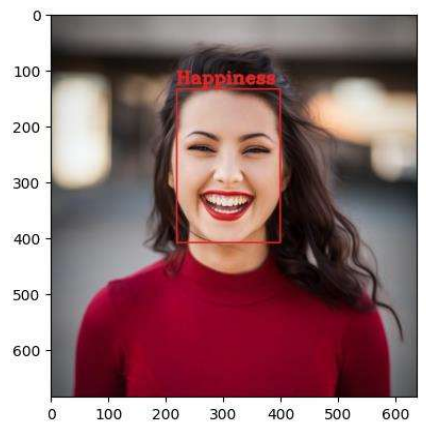
Overall, the combined approach of using MTCNN for face detection and MobileNet_7 for feature extraction, followed by an emotion recognition model, allows for real-time or efficient emotion analysis from images and videos.

This kind of system finds applications in various areas like social robotics, human-computer interaction, and emotion-aware user interfaces.

Program:

```
%pip install mediapipe
import cv2 as cv
import mediapipe as mp

mtcnn_model=mtcnn.MTCNN()
mobilenet_model=load_model('C:/Users/Desktop/ComputerVision/Resources/config/m
obilenet_7.h5')
mobilenet_model.summary()

INPUT_SIZE=(224,224)
idx_to_class={0:'Anger',
              1:'Disgust',
              2:'Fear',
              3:'Happiness',
              4:'Neutral',
              5:'Sadness',
              6:'Surprise'}
colors=[(31,119,180),(255,127,14),(44,160,44),(219,39,40),(148,103,189),(140,8
6,75),(227,119,194)]

img=cv.imread('C:/Users/Desktop/ComputerVision/Resources/Photos/lady.jpg')
img_rgb=cv.cvtColor(img,cv.COLOR_BGR2RGB)
x,y,w,h=mtcnn_model.detect_faces(img_rgb)[0]['box']
face=img_rgb[y:y+h,x:x+w]
face=cv.resize(face,INPUT_SIZE)
data=np.array([face],ndmin=2)
scores=mobilenet_model.predict(data)
class_idx=np.argmax(scores,axis=1)[0]
cv.rectangle(img_rgb,(x,y),(x+w,y+h),colors[class_idx],2)
cv.putText(img_rgb,idx_to_class[class_idx],(x,y-
10),cv.FONT_HERSHEY_COMPLEX,1,colors[class_idx],4)
plt.imshow(img_rgb)
```

Output



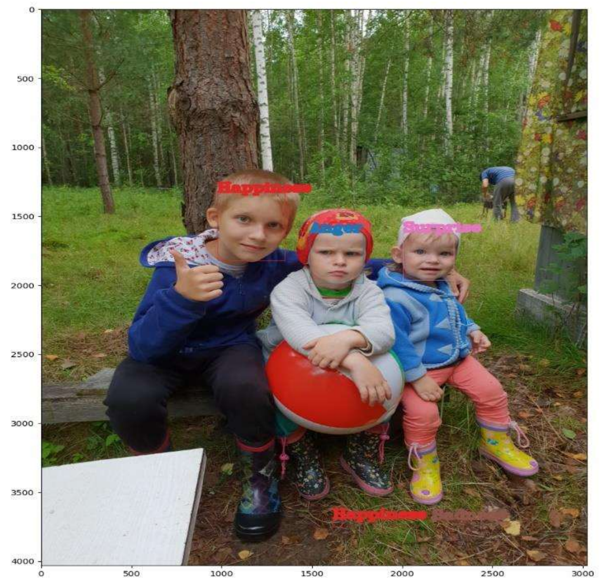
Multiple face detection

To perform multiple face detection on an image using the Haar Cascade classifier, you can modify the code you provided to detect and draw rectangles around all the detected faces. Here's the updated code to detect and draw rectangles around multiple faces:

Program:

```
img=cv.imread('C:/Users/Desktop/ComputerVision/Resources/Photos/boys.jpg')
img_rgb=cv.cvtColor(img,cv.COLOR_BGR2RGB)
boxes=mtcnn_model.detect_faces(img_rgb)
faces= []
for box in boxes:
     x,y,w,h=box['box']
     face=img_rgb[y:y+h,x:x+w]
     face=cv.resize(face,INPUT_SIZE)
     faces.append(face)

data=np.array(faces,ndmin=2)
scores=mobilenet_model.predict(data)
class_idx=np.argmax(scores,axis=1)
for idx,box in zip (class_idx,boxes):
     x,y,w,h=box['box']
     cv.rectangle(img_rgb,(x,y),(x+w,y+h),colors[idx],2)
     cv.putText(img_rgb,idx_to_class[idx],(x,y-
10),cv.FONT_HERSHEY_COMPLEX,3,colors[idx],20)
plt.figure(figsize=(14,14))
plt.imshow(img_rgb)
```

Output:



Video detector

To read and process a video using computer vision, you can use the OpenCV library in Python. OpenCV provides functions to read, display, and process video frames. Here's a basic example of how to read and display a video using OpenCV:

```
#VIDEO READER#
vcap=cv.VideoCapture('C:/Users/pushpasri/Desktop/ComputerVision/Resources/Vide
os/Lane.mp4')
while True:
    ret,frame=vcap.read()
    if ret:
        cv.imshow('out',frame)
        cv.waitKey(int(vcap.get(cv.CAP_PROP_FPS)))
    else:
        break
vcap.release()
cv.destroyAllWindows()
```

Lane Detector

Lane detection is a common computer vision application used in autonomous vehicles and driver assistance systems. It involves detecting and marking the lanes on a road in a video frame to assist in lane keeping and lane departure warning systems. Here's a step-by-step guide to building a simple lane detection system using OpenCV in Python:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

Make sure to replace 'Lane.mp4' with the actual path to your video file. This code will read the video frame by frame, detect the lanes using the Canny edge detection and Hough Transform, and then draw the detected lanes on the video frames. The processed video will be displayed in a window, and you can stop it by pressing the 'q' key.

```
#VIDEO READER#
vcap=cv.VideoCapture('C:/Users/pushpasri/Desktop/ComputerVision/Resources/Vide
os/Lane.mp4')
while True:
    ret,frame=vcap.read()
    if ret:
        edge_,mask_,detected_=lane_detector(frame)
        cv.imshow('Input',frame)
        cv.imshow('Edge',edge_)
        cv.imshow('Mask',mask_)
        cv.imshow('Detected',detected_)
        cv.waitKey(int(vcap.get(cv.CAP_PROP_FPS)))
    else:
        break
vcap.release()
cv.destroyAllWindows()
```

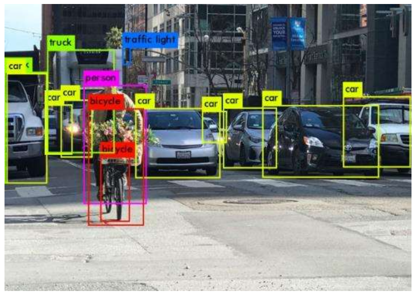
```
#LANE DETECTOR#
def roi(img,vertices):
    mask=np.zeros_like(img)
    match_mask_color=(255)
    cv.fillPoly(mask,[vertices],match_mask_color)
    return cv.bitwise_and(img,mask)
def draw_lines(img,lines):
    img=np.copy(img)
    blank_img=np.zeros(img.shape,np.uint8)
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv.line(blank_img,(x1,y1),(x2,y2),(0,255,0),2)
    return cv.addWeighted(img,0.8,blank_img,1,0.0)

def lane_detector(img):
    h,w=img.shape[:-1]
    roi_vertices=[(200,h),(w/2,h/1.37),(w-300,h)]
    gray_img=cv.cvtColor(img,cv.COLOR_BGR2GRAY)
    edge=cv.Canny(gray_img,50,100,apertureSize=3)
    cropped_img=roi(edge,np.array(roi_vertices,np.int32))
    lines=cv.HoughLinesP(cropped_img,rho=2,theta=np.pi/180,threshold=50,lines=
np.array([]),minLineLength=10)
    img_lines=draw_lines(img,lines)
    return edge,cropped_img,img_lines
```

Object detection

The **YOLO** class is part of the Ultralytics YOLO implementation**pt** is the pre-trained model checkpoint that the class loads.

```
from ultralytics.models.yolo import YOLO
import cv2 as cv
model = YOLO('yolov8s.pt')
# Load the input image
image_path = 'road_image.jpg'
# Replace with the path to your image
image = cv.imread(image_path)
results = model(image)
results.show()
print(results.pred)
O
```

Output:

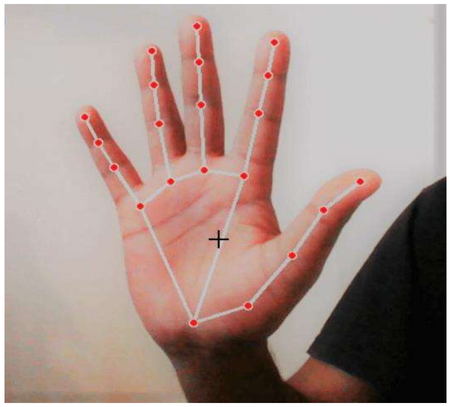


Hand detection

Hand detection is a computer vision task that involves identifying and localizing human hands in images or video frames. It finds applications in various fields, including human-computer interaction, sign language recognition, gesture-based control systems, virtual reality, and augmented reality. To perform hand detection, you'll need ce

```
mpHands=mp.solutions.hands
hands=mpHands.Hands()
mpDraw=mp.solutions.drawing_utils
def find_digit(p):
    d=0
    TIPS=[4,8,12,16,20]
    if p[12].x>p[20].x:
        if p[TIPS[0]].x>p[TIPS[0]-1].x:
            d+=1
    else:
         if p[TIPS[0]].x<p[TIPS[0]-1].x:
            d+=1
    for tip in TIPS[1:]:
        if p[tip].y<p[tip-2].y:
            d+=1
    return d

cap=cv.VideoCapture(0)
cap.set(cv.CAP_PROP_FRAME_WIDTH,1080)
cap.set(cv.CAP_PROP_FRAME_HEIGHT,720)
while True:
    ret,frame=cap.read()
    if ret:
        cv.imshow('HandDetection',frame)
        frame_rgb=cv.cvtColor(frame,cv.COLOR_BGR2RGB)
        results=hands.process(frame_rgb)
        if results.multi_hand_landmarks:
            single_hand=results.multi_hand_landmarks[0]
            keypoints=single_hand.landmark
            digit=find_digit(keypoints)
            mpDraw.draw_landmarks(frame,single_hand,mpHands.HAND_CONNECTIONS)
            cv.putText(frame,'Digit-
{0}'.format(digit),(10,70),cv.FONT_HERSHEY_COMPLEX,2,[0,255,0],4)
        cv.imshow('HandDetection',frame)
    else:
        break
    k=cv.waitKey(1)
    if k==113:
        break
cv.destroyAllWindows()
```

Output:



Face Mesh detection

Face detection is a computer vision task that involves locating and identifying human faces within an image or a video frame. This technology is widely used in various applications, such as facial recognition, video surveillance, photography, and social media tagging.

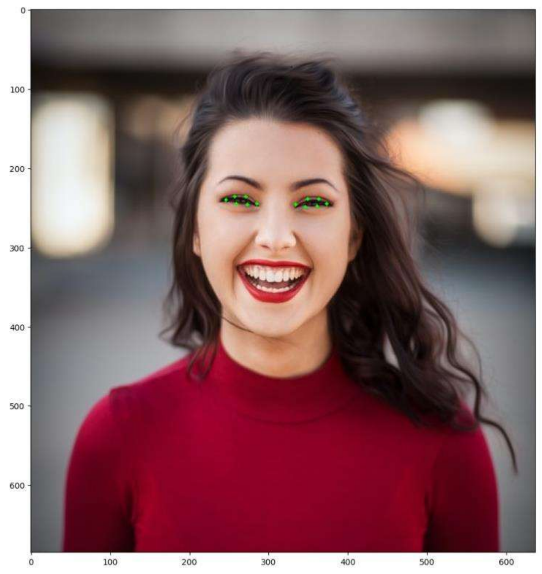
To perform face detection using these deep learning models, you can use various libraries and frameworks, such as:

1. OpenCV: OpenCV is an open-source computer vision library that provides pre-trained face detection models and functions for real-time face detection.
2. TensorFlow: TensorFlow is a popular deep learning framework that offers pre-trained face detection models, and you can build custom face detection pipelines using it.
3. PyTorch: PyTorch is another deep learning framework that provides various pre-trained models for face detection.

```
import numpy as np
import math
def euclidean_dist(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2)+((p1[1]-p2[1])**2)
img=cv.imread('Resources/Photos/lady.jpg')
img_rgb=cv.cvtColor(img,cv.COLOR_BGR2RGB)
LEFT_EYE_IDX=list(set(np.ravel(list(fm_con.FACEMESH_LEFT_EYE))))
RIGHT_EYE_IDX=list(set(np.ravel(list(fm_con.FACEMESH_RIGHT_EYE))))
ALL_EYE_IDX=LEFT_EYE_IDX+RIGHT_EYE_IDX
print('LEFT EYE::{0}'.format(LEFT_EYE_IDX))
print('RIGHT EYE::{0}'.format(RIGHT_EYE_IDX))
LEFT_EYE_RP=[362,385,387,263,373,380]
RIGHT_EYE_RP=[33,160,158,133,153,144]
print('LEFT EYE RP::{0}'.format(LEFT_EYE_RP))
print('RIGHT EYE RP::{0}'.format(RIGHT_EYE_RP))
ALL_EYE_RP=LEFT_EYE_RP+RIGHT_EYE_RP
```

```python
h,w=img_rgb.shape[:-1]
for idx in ALL_EYE_RP:
    lm=landmarks.landmark[idx]
    x, y,=int(lm.x*w),int(lm.y*h)
    cv.circle(img_rgb,(x, y),2,[0,255,0],-1)

def calculate_ear(eye_rp):
    EAR_IDX=[(0,3),(1,5),(2,4)]
    EAR_VAL=[]
    for idx in EAR_IDX:
        point1=landmarks.landmark[eye_rp[idx[0]]]
        point1=[int(point1.x*w),int(point1.y*h)]
        point2=landmarks.landmark[eye_rp[idx[1]]]
        point2=[int(point2.x*w),int(point2.y*h)]
        EAR_VAL.append(euclidean_dist(point1, point2))
    return(EAR_VAL[1]+EAR_VAL[2])/(2.0*EAR_VAL[0])
LEFT_EAR_VAL=calculate_ear(LEFT_EYE_RP)
print('LEFT EYE EAR:',LEFT_EAR_VAL)
RIGHT_EAR_VAL=calculate_ear(RIGHT_EYE_RP)
print('RIGHT EYE EAR:',RIGHT_EAR_VAL)
AVG_EAR_VAL=(LEFT_EAR_VAL+RIGHT_EAR_VAL)/2
print('Eye Aspect Ratio:',AVG_EAR_VAL)

plt.subplots(figsize=(12,12))
plt.imshow(img_rgb)
```

Output:



Selfie Segmentation:

Selfie segmentation is a computer vision task that involves segmenting or separating the foreground (the person's face or body) from the background in a selfie image. The goal of selfie segmentation is to accurately identify the boundaries of the person's face or body and distinguish it from the surrounding environment.

Selfie segmentation is an essential component in many applications, especially in the field of augmented reality (AR), where virtual objects or effects are superimposed on the user's face or body in real-time. By accurately segmenting the selfie image, these AR effects can be applied more realistically and seamlessly.

To perform selfie segmentation, deep learning-based semantic segmentation models are often employed. Semantic segmentation is a technique in computer vision that assigns a specific label or class to each pixel in an image, effectively dividing the image into different regions or segments based on their semantic meaning.

Several deep learning architectures can be adapted for selfie segmentation, such as Fully Convolutional Networks (FCN), U-Net, DeepLab, and Mask R-CNN. These models are trained on large datasets with annotated selfie images where the ground truth segmentation masks are provided.

Several open-source libraries and frameworks, such as TensorFlow, PyTorch, and OpenCV, provide pre-trained models and tools for semantic segmentation, making it more accessible for developers to implement selfie segmentation in their applications.

Code:

```
import cv2 as cv
import numpy as np
import mediapipe as mp
mpSelfieSeg=mp.solutions.selfie_segmentation
SelfieSeg=mpSelfieSeg.SelfieSegmentation()
cap=cv.VideoCapture(0)
cap.set(cv.CAP_PROP_FRAME_WIDTH,1080)
cap.set(cv.CAP_PROP_FRAME_HEIGHT,720)
bg_image=cv.imread('C:/Users/Desktop/ComputerVision/Resources/BG.jpg')
while True:
    ret,frame=cap.read()
    bg_image=cv.resize(bg_image,frame.shape[:-1][::-1])
    if ret:
        frame_rgb=cv.cvtColor(frame,cv.COLOR_BGR2RGB)
        results=SelfieSeg.process(frame_rgb)
        mask=results.segmentation_mask > 0.8
        mask_rgb=np.dstack((mask,mask,mask))
        frame=np.where(mask_rgb,frame,bg_image)
    cv.imshow('Out',frame)
    k=cv.waitKey(1)
    if k==113:
        break
cv.destroyAllWindows()
```

Conclusion

Over the course of five intensive days of training in computer vision, I acquired valuable knowledge in various libraries, frameworks, and techniques related to face detection, emotion detection, video detection, lane detection, hand detection, selfie segmentation, and more. Let me provide a brief overview of the key topics covered during each day:

Day 1: Introduction to Computer Vision Basics

- Introduction to computer vision and its applications.
- Basics of image processing using OpenCV library (cv2) for tasks like reading images, resizing, and color space conversions.
- Understanding and implementing basic image filters and edge detection.
- Hands-on exercises and simple image manipulation tasks.

Day 2: Lane Detection and Object detection

- Introduction to lane detection for autonomous vehicles and lane departure warning systems.
- Learning about popular lane detection algorithms and techniques.
- Implementing lane detection on road images and videos.
- Implementing selfie segmentation using semantic segmentation models.

Day 3: Face Detection and Emotion Detection

- Introduction to face detection and its importance in computer vision applications.
- Learning about popular face detection techniques like Haar cascades Face Detection.
- Introduction to emotion detection using pre-trained Convolutional Neural Networks (CNNs).
- Implementing face detection and emotion detection on images and videos using MTCNN, and CNN-based models.

Day 4: Video Analysis and Hand Detection

- Understanding video analysis techniques using OpenCV.
- Implementing video face detection and emotion detection on live webcam feeds.
- Introduction to hand detection using MediaPipe Hands.
- Implementing real-time hand detection in video streams.

Day 5: Selfie Segmentation

- In-depth discussion of useful utility libraries like NumPy and PIL for image processing and manipulation.
- Introduction to selfie segmentation and its importance in augmented reality applications.
- receive feedback from peers and instructors.

Throughout the 5-day training program, the participants will gain hands-on experience in various computer vision tasks, including face detection, emotion detection, video analysis, lane detection, and hand detection. They will also learn about popular libraries and frameworks like MediaPipe, MTCNN, CNNs, OpenCV, and utilities like NumPy and PIL for efficient image processing. By the end of the training, participants should have a solid foundation in computer vision and be able to apply their knowledge to real-world applications.

References

1. Kaggle (www.kaggle.com): Kaggle is a well-known platform for data science and machine learning competitions. It hosts a wide range of computer vision challenges and datasets, allowing you to participate in competitions and learn from the community's solutions.

2. OpenCV (opencv.org): OpenCV is an open-source computer vision library that provides extensive documentation, tutorials, and resources to learn about computer vision algorithms and techniques. It also offers pre-trained models and utilities for various computer vision tasks.

3. PyImageSearch (www.pyimagesearch.com): PyImageSearch is a website created by Adrian Rosebrock, offering numerous tutorials and practical guides on various computer vision topics using Python and OpenCV. The website covers a wide range of applications, from beginner to advanced levels.

4. Papers with Code (paperswithcode.com): Papers with Code is a platform that collects research papers along with their code implementations. You can find many computer vision papers and their corresponding code on this website, which is an excellent resource for staying up-to-date with the latest research.

5. TensorFlow (www.tensorflow.org): TensorFlow, an open-source deep learning framework by Google, provides a wealth of resources for computer vision. The website offers tutorials, guides, and pre-trained models for various vision tasks, along with TensorFlow Hub for accessing reusable models.

6. PyTorch (pytorch.org): PyTorch, another popular deep learning framework, also offers extensive resources for computer vision. The website provides tutorials, documentation, and pre-trained models to help you get started with PyTorch in computer vision projects.

7. Papers and Proceedings of Computer Vision and Pattern Recognition (CVPR) Conference: The CVPR conference is a major event in the computer vision community. Its website hosts proceedings from past conferences, allowing you to explore cutting-edge research in computer vision.