A

SYNOPSIS

of

MINOR PROJECT

on

Real-Time Pedestrian Detection

Submitted by

Dino Jackson (21EGICS031)

Project Guide
Mr. Fateh Bahadur Kuwar

Head of Department
Dr. Mayank Patel

Geetanjali Institute of Technical Studies, Dabok , Udaipur (Raj.)
Department of Computer Science and Engineering
October,2024

Problem Statement:

Real-time pedestrian detection is a crucial technology for various applications, including autonomous driving, surveillance systems, and crowd analytics. This paper presents a comprehensive approach to pedestrian detection using **Python and OpenCV**, highlighting the methodologies, implementation details, and performance evaluation. We leverage the Histogram of **Oriented Gradients (HOG)** feature descriptor combined with a **Support Vector Machine (SVM**) classifier for detecting pedestrians in video streams.

Brief Description:

Pedestrian detection is vital for ensuring safety and efficiency in numerous systems. This section outlines the importance of real-time pedestrian detection, its applications, and the scope of the paper. We introduce the main techniques used, emphasizing the combination of HOG features and SVM classifier for effective detection.

Objective and Scope:

Objective

The primary objective of this project is to develop an efficient and robust system for real-time pedestrian detection using Python and OpenCV. The system aims to accurately identify and locate pedestrians in live video streams, providing real-time feedback. This involves several key goals:

1. **Accuracy**: Ensure high detection accuracy by minimizing false positives and false negatives.
2. **Real-time Performance**: Achieve low-latency processing to allow real-time detection and response.
3. **Scalability**: Develop a system that can be scaled to different environments and use cases, such as varying lighting conditions and crowded scenes.
4. **User-Friendliness**: Create a solution that is easy to set up and use, with minimal configuration requirements.

Scope

The scope of this project encompasses various aspects of real-time pedestrian detection, from initial setup and development to testing and potential applications.

Methodology:

1. Import the necessary packages

- Numpy is used for all data storing, retrieving from the model, and working with it.
- Opencv is used to read frames from our video file or our webcam feed, resize and reshape it according to the model requirement. It also provides the dnn module which we will use to work with our deep neural network model. Also draw the bounding boxes on the image and show it back to the user.
- Os for working with files reading paths and stuff.
- Imutils is another great library for performing different actions on images. It acts as a helper library providing some very useful functions to opencv which already is a vast library.

import numpy as np
import cv2
import os
import imutils

Declaring some threshold variables that we will use later. Nms threshold is the threshold for separating overlapping predictions, more about nms ahead. Min confidence is the threshold for confidence score returned by the model above which a prediction is considered true.

NMS_THRESHOLD=0.3
MIN_CONFIDENCE=0.2

2. Pedestrian Detection Function

Now we define the most important function that we will use for pedestrian detection. It takes the image frame by frame from opencv either from a webcam or video file, the model, output layer name from which we will get output, and a variable personidz.

We get the dimensions of the frame passed and initialize an array that will hold the result from the model.

```
def pedestrian_detection(image, model, layer_name,
personidz=0):
 (H, W) = image.shape[:2]
 results = []
```

Construct a blob with the frame we received and then pass it to the yolo model performing a forward pass, which will return the bounding box for the detections and the confidence scores for each. Model.forward will return the output from the layer that was passed to it, ie: the output layer.

```
blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416),
 swapRB=True, crop=False)
model.setInput(blob)
layerOutputs = model.forward(layer_name)
```

Create arrays for holding the resulting bounding box, centroid, and confidence associated with.

boxes = []
centroids = []
confidences = []
LayerOutputs is a list of lists containing outputs. Each list in layer output contains details about single prediction like its bounding box confidence etc. So together all the predictions are in the form of a list of lists, we loop through it.

for output in layerOutputs:
for detection in output:

From the individual detection we get the scores for all the predictions, classid and confidence for the class id with maximum score.

scores = detection[5:]
classID = np.argmax(scores)
confidence = scores[classID]

3. Getting the Detection

Now we need to get the detection that we need. The class id for detecting a person is 0 in yolo format, so we check that and also that the confidence is above the threshold so that we don't use false positives.

Yolo returns the centroid of the bounding box with the height and width of the box instead of the top-left coordinate of the box.

if classID == personidz and confidence > MIN_CONFIDENCE:
box = detection[0:4] * np.array([W, H, W, H])
(centerX, centerY, width, height) = box.astype("int")

Now because we don’t have the top right coordinate of the bounding box we will calculate that by subtracting half of the width and height from centroid x and centroid y point respectively.

Now that we have all the things that we need we will add them to the lists that we created earlier.

x = int(centerX - (width / 2))
y = int(centerY - (height / 2))
boxes.append([x, y, int(width), int(height)])
centroids.append((centerX, centerY))
confidences.append(float(confidence))

Out of the loop here we do nms or non maxima suppression to remove overlapping and weak bounding boxes. Because neighboring windows of the actual detection also contain high scores, hundreds of predictions pop up that we need to clean. We use the opencv’s dnn module provided nmsboxes function for this

idzs = cv2.dnn.NMSBoxes(boxes, confidences, MIN_CONFIDENCE, NMS_THRESHOLD)

Check if we have any detection, and loop through it. Extract the bounding box coordinates, the width, and height and add all the details to our results list that we created earlier. Finally, return that list and end the function.

```
if len(idzs) > 0:
 for i in idzs.flatten():
  (x, y) = (boxes[i][0], boxes[i][1])
  (w, h) = (boxes[i][2], boxes[i][3])

  res = (confidences[i], (x, y, x + w, y + h), centroids[i])
  results.append(res)
return results
```

Here we load the labels that the model was trained on. It has 80 class labels in coco.names file and coco format. Set the path to weights and cfg file for our model. Remember that these are relative paths, and in this case because the files are in the same folder we just use the name as paths.

Now we load the model using opencvdnn module into the model variable.

```
labelsPath = "coco.names"
LABELS = open(labelsPath).read().strip().split("\n")
weights_path = "yolov4-tiny.weights"
config_path = "yolov4-tiny.cfg"
model = cv2.dnn.readNetFromDarknet(config_path,
weights_path)
```

If your Opencvdnn compiled with cuda and you want to run this on the GPU uncomment these lines to use cuda enabled GPUs.

```
'''
model.setPreferableBackend(cv2.dnn.DNN_BACKEND_CUDA)
model.setPreferableTarget(cv2.dnn.DNN_TARGET_CUDA)
'''
```

We need the name of the last layer of deep learning to get the output from it. After this, initialize the videocapture from opencv. If you want to use the webcam pass 0 instead of the filename to the videocapture function.

```
layer_name = model.getLayerNames()
layer_name = [layer_name[i[0] - 1] for i in
model.getUnconnectedOutLayers()]
cap = cv2.VideoCapture("streetup.mp4")
writer = None
```

4. Reading the frames and processing it

Now, start an infinite loop, and read the frames from videocapture. If the frames end, ie: video ends we break the infinite loop.

Resize the image according to requirements without changing the aspect ratio of the image with imutils resize function.

Pass the image model output layer name, and classid for label person to the pedestrian_detection function we created.

```
while True:
(grabbed, image) = cap.read()
if not grabbed:
break
image = imutils.resize(image, width=700)
results = pedestrian_detection(image, model, layer_name,
personidz=LABELS.index("person"))
```

Getting back the results from the function we loop through it, and draw the bounding box rectangles on the predictions and show it to the user. We capture the esc key press with the waitkey function from opencv and break the loop. Finally releasing the capture and closing all windows.

```
for res in results:
cv2.rectangle(image, (res[1][0],res[1][1]), (res[1][2],res[1][3]),
(0, 255, 0), 2)
cv2.imshow("Detection",image)
key = cv2.waitKey(1)
if key == 27:
break
cap.release()
cv2.destroyAllWindows()
```

Hardware and Software Requirements:

Hardware Requirements

1. **Computer**: A modern computer with sufficient processing power to handle real-time video processing. The specifications will vary depending on the complexity of the task and the desired frame rate, but generally:
    - **Processor (CPU)**: A multi-core processor, preferably Intel i5 or higher, or AMD Ryzen 5 or higher.
    - **Graphics Processing Unit (GPU)**: An optional dedicated GPU can significantly accelerate the processing, especially if using deep learning models. NVIDIA GPUs with CUDA support are recommended.
    - **Memory (RAM)**: At least 8 GB of RAM is recommended, with 16 GB or more being preferable for smoother performance.
    - **Storage**: An SSD (Solid State Drive) for faster read/write speeds, especially useful when handling large video files.
2. **Camera**: A high-definition camera for capturing video input. Common options include:
    - USB webcams with 1080p resolution.
    - IP cameras or other high-definition cameras for better image quality and detail.
3. **Power Supply**: Ensure a stable and reliable power supply to maintain consistent performance.
4. **Cooling**: Proper cooling systems to prevent overheating during prolonged processing tasks.

Software Requirements

1. **Operating System**: The system should be compatible with the libraries and tools required. Commonly used operating systems include:
   - **Windows**: Windows 10 or later.
   - **Linux**: Ubuntu 18.04 or later is highly recommended for its compatibility with various libraries and tools.
   - **macOS**: macOS Catalina or later.
2. **Python Environment**: Ensure you have Python installed. Python 3.6 or later is recommended.
   - You can download and install Python from python.org.
3. **Python Libraries**:
   - **OpenCV**: For image processing and computer vision tasks.

     ```
     bash
     Copy code
     pip install opencv-python
     ```

   - **NumPy**: For numerical operations.

     ```
     bash
     Copy code
     pip install numpy
     ```

   - **imutils**: For convenience functions related to image processing.

     ```
     bash
     Copy code
     pip install imutils
     ```

4. **Development Environment**: An IDE or text editor for writing and running your Python code. Some popular options include:
    - **PyCharm**: A powerful IDE for Python development.
    - **Visual Studio Code**: A versatile code editor with support for Python.
    - **Jupyter Notebook**: Useful for prototyping and iterative development.
5. **Additional Libraries (if using advanced models)**:
    - **TensorFlow** or **PyTorch**: If you plan to integrate deep learning models for enhanced detection accuracy.

```
bash
Copy code
pip install tensorflow
pip install torch
```

6. **Optional Libraries**:
    - **CUDA and cuDNN**: For leveraging NVIDIA GPUs to accelerate processing if using deep learning models. Install CUDA and cuDNN from the NVIDIA website, ensuring compatibility with your GPU and TensorFlow/PyTorch versions.

Example Setup Steps

1. **Install Python**:
    - Download and install the latest version of Python from python.org.

2. **Set Up a Virtual Environment**:

```
bash
Copy code
python -m venv pedestrian_detection_env
source pedestrian_detection_env/bin/activate
# On Windows use
`pedestrian_detection_env\Scripts\activate`
```

3. **Install Required Libraries**:

```
bash
Copy code
pip install opencv-python numpy imutils
```

4. **Install OpenCV and Test Installation**:

```
python
Copy code
import cv2
print(cv2.__version__)
```

5. **Configure GPU (if applicable)**:
   - Install NVIDIA drivers, CUDA, and cuDNN.
   - Install TensorFlow or PyTorch with GPU support:

```
bash
Copy code
pip install tensorflow-gpu  # For
TensorFlow
pip install torch torchvision torchaudio
# For PyTorch
```

6. **Write and Run the Code**:
    - Use your preferred IDE to write the pedestrian detection code.
    - Ensure your camera is connected and accessible by OpenCV.

By following these hardware and software requirements, you can set up an efficient environment for real-time pedestrian detection using Python and OpenCV.

Technologies:

- **Programming Languages**: Python

- **Libraries and Frameworks:** OpenCV, NumPy, TensorFlow, Keras, PyTorch, Scikit-learn

- **Development Tools:** Jupyter Notebook, Anaconda, Visual Studio Code

- **Hardware:** Standard PC with a webcam or external camera, optional GPU for acceleration

Testing Techniques:

- **Metrics:** Use metrics such as precision, recall, F1-score, and mean average precision (mAP) to evaluate the performance of the pedestrian detection models.

- **Benchmarking:** Compare the performance of different models on standard benchmark datasets and real-world scenarios.

Project Contribution:

Results:

- Demonstrate the real-time detection capability of the system through video streams or live camera feeds.

- Present the detection accuracy and processing speed of the system under various conditions.

Conclusion:

Summarize the effectiveness of the developed pedestrian detection system, its real-time performance, and potential applications. Discuss possible improvements such as enhancing detection accuracy in crowded scenes, reducing false positives, and extending the system to detect other objects of interest.