Crowd Counting using YOLOv3

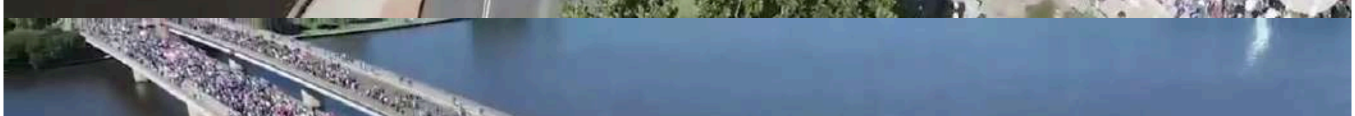
YOLOv3 (You Only Look Once) is an object detection model. Its goal is to identify and classify discrete objects in an image, such as cars, people, or animals, and draw bounding boxes around them. YOLO is designed to detect objects with well-defined boundaries. It works best when the objects are relatively large, clearly separated, and the number of objects is small or moderate.

```
import cv2
import numpy as np
from google.colab.patches import  cv2_imshow

# Load YOLO model
net = cv2.dnn.readNet("yolov3.weights", "yolov3.cfg")
layer_names = net.getLayerNames()
output_layers = [layer_names[i - 1] for i in net.getUnconnectedOutLayers()]

# Load the video
cap = cv2.VideoCapture('/content/Drone Footage of Canberra s HISTORIC Crowd.mp4')

# Time frame from 0:14 to 0:32 corresponds roughly to frames 14*fps to 32*fps
fps = cap.get(cv2.CAP_PROP_FPS)
start_time = 14  # Start time in seconds
end_time = 32    # End time in seconds
start_frame = int(start_time * fps)
end_frame = int(end_time * fps)

frame_count = 0
people_count = []

# Move to the starting frame
cap.set(cv2.CAP_PROP_POS_FRAMES, start_frame)

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    frame_count += 1
    current_frame = start_frame + frame_count

    # Stop if the current frame is beyond the end frame
    if current_frame > end_frame:
        break

    # Prepare the frame for YOLO
    height, width, channels = frame.shape
```

```
    blob = cv2.dnn.blobFromImage(frame, 0.00392, (416, 416), (0, 0, 0), True, crop=F
    net.setInput(blob)
    outs = net.forward(output_layers)

    class_ids = []
    confidences = []
    boxes = []

    # Processing the output
    for out in outs:
        for detection in out:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.5 and class_id == 0:  # Class 0 is 'person' in COCO da
                # Object detected
                center_x = int(detection[0] * width)
                center_y = int(detection[1] * height)
                w = int(detection[2] * width)
                h = int(detection[3] * height)
                # Rectangle coordinates
                x = int(center_x - w / 2)
                y = int(center_y - h / 2)
                boxes.append([x, y, w, h])
                confidences.append(float(confidence))
                class_ids.append(class_id)

    # Apply Non-Maximum Suppression to remove redundant boxes
    indexes = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)

    # Count the number of people
    count = len(indexes)
    people_count.append(count)

    # Draw bounding boxes (Optional, to visualize)
    for i in range(len(boxes)):
        if i in indexes:
            x, y, w, h = boxes[i]
            label = str('Person')
            color = (0, 255, 0)
            cv2.rectangle(frame, (x, y), (x + w, y + h), color, 2)
            cv2.putText(frame, label, (x, y - 10), cv2.FONT_HERSHEY_PLAIN, 1, color,

    # Display the frame (Optional)
    cv2_imshow(frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

```
# Average crowd size
estimated_crowd_size = np.mean(people_count)
print(f"Estimated Crowd Size: {estimated_crowd_size:.2f}")
```

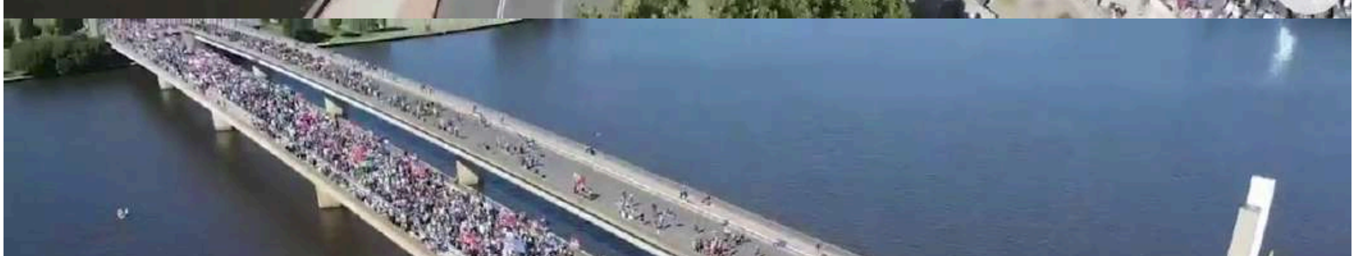

Person
Person




Person
Person
Person

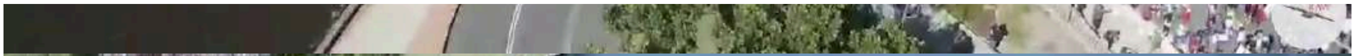

Person


Person




Person


Person




Person


Person


Person

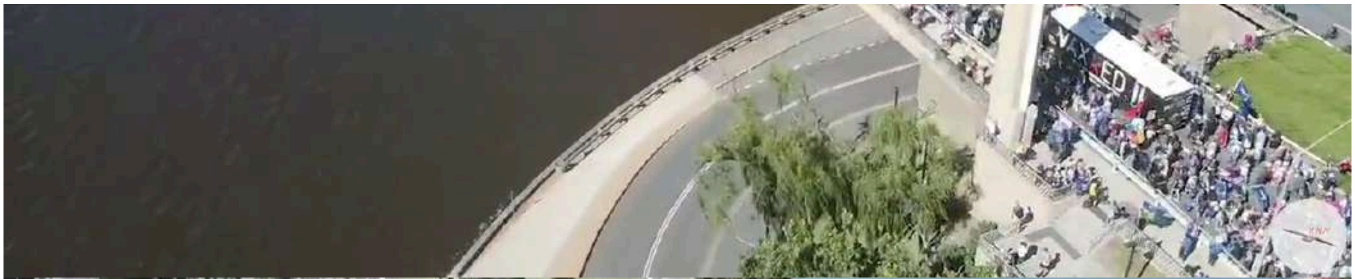

Person


Person


Person

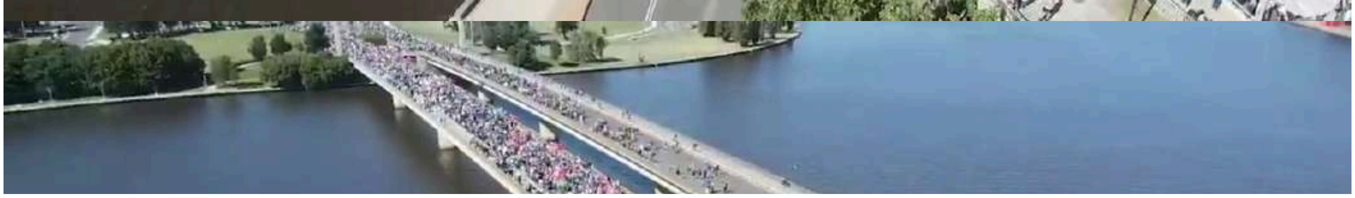

Person


Person



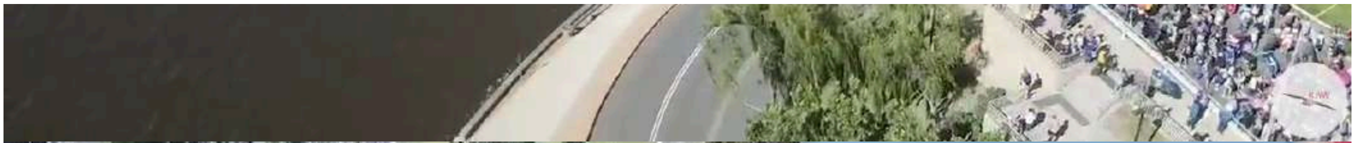

Person


Person




Person


Person




Person


Person
Person


Person

```
Estimated Crowd Size: 0.27
```

YoloV3 has a harder time detecting small, distant objects (e.g., people far away) because its bounding box detectors work better with objects that occupy a significant portion of the image. In crowd scenes, many people can appear as very small, distant figures, making it difficult for YOLOv3 to detect them reliably.

YOLOv3 is optimized for real-time object detection. It is fast and can process frames quickly, but it sacrifices accuracy in dense or overlapping scenarios, especially when the number of objects increases. It is designed for tasks like autonomous driving, general object detection, and real-time monitoring.

Crowd Counting using CSRNet Model

CSRNet (Convolutional Neural Network for Crowd Counting) is a crowd density estimation model. Instead of detecting individual objects with bounding boxes, CSRNet generates density maps that estimate how many people are present in various parts of the image. It is specialized for crowd counting, especially in dense scenes, where people overlap and are closely packed together.

CSRNet is designed specifically to estimate the number of people in dense crowds, while YOLOv3 is designed to detect individual objects, which becomes difficult when people are packed closely together.

Excels in dense crowd situations because it doesn't rely on individual detection. Instead, CSRNet produces a density map where each pixel represents a "contribution" to the total count, allowing it to estimate the number of people even when they overlap or are partially occluded.

YOLOv3 struggles with high-density, overlapping crowds because it is focused on detecting distinct objects. CSRNet, on the other hand, handles overlapping and occluded people well because it focuses on generating a density map for estimation rather than individual detection.

```
import torch.nn as nn
from torchvision import models

class CSRNet(nn.Module):
    def __init__(self, load_weights=False):
        super(CSRNet, self).__init__()
```

```
        self.seen = 0
        self.frontend_feat = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512
        self.backend_feat  = [512, 512, 512,256,128,64]
        self.frontend = make_layers(self.frontend_feat)
        self.backend = make_layers(self.backend_feat,in_channels = 512,dilation =
        self.output_layer = nn.Conv2d(64, 1, kernel_size=1)
        if not load_weights:
            mod = models.vgg16(pretrained = True)
            self._initialize_weights()
            for i in range(len(self.frontend.state_dict().items())):
                list(self.frontend.state_dict().items())[i][1].data[:] = list(mod
    def forward(self,x):
        x = self.frontend(x)
        x = self.backend(x)
        x = self.output_layer(x)
        return x
    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.normal_(m.weight, std=0.01)
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)


def make_layers(cfg, in_channels = 3,batch_norm=False,dilation = False):
    if dilation:
        d_rate = 2
    else:
        d_rate = 1
    layers = []
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=d_rate,dila
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

```
import h5py
import scipy.io as io
import PIL.Image as Image
import numpy as np
from matplotlib import pyplot as plt, cm as c
from scipy.ndimage.filters import gaussian_filter
import scipy
import torchvision.transforms.functional as F
import torch
from torchvision import transforms
```

```
<ipython-input-26-815c7c4b4df0>:6: DeprecationWarning: Please import `gaussian
  from scipy.ndimage.filters import gaussian_filter
```

```
model = CSRNet()

checkpoint = torch.load('weights.pth', map_location="cpu")
model.load_state_dict(checkpoint)
model.eval()

transform=transforms.Compose([
                       transforms.ToTensor(),transforms.Normalize(mean=[0.485, 0.
                                     std=[0.229, 0.224, 0.225]),
                   ])
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: Use
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: Use
  warnings.warn(msg)
<ipython-input-27-5c863132dbcc>:3: FutureWarning: You are using `torch.load` w
  checkpoint = torch.load('weights.pth', map_location="cpu")
```

```
def process_video(video_path):
    cap = cv2.VideoCapture(video_path)

    # Get FPS and calculate frame range for 0:14s to 0:32s
    fps = cap.get(cv2.CAP_PROP_FPS)
    start_time = 14  # 14 seconds
    end_time = 32    # 32 seconds
    duration = end_time - start_time  # Duration in seconds

    # Total number of frames in the time range
```

```
    total_frames = int(fps * duration)

    # Select 18 evenly spaced frames from the time range
    frames_to_capture = np.linspace(0, total_frames, 18, endpoint=False, dtype=in

    # Set the video to the start time
    start_frame = int(fps * start_time)
    cap.set(cv2.CAP_PROP_POS_FRAMES, start_frame)

    total_crowd_count = 0
    frame_count = 0

    for frame_index in frames_to_capture:
        # Move to the desired frame
        cap.set(cv2.CAP_PROP_POS_FRAMES, start_frame + frame_index)
        ret, frame = cap.read()

        if not ret:
            break

        frame_count += 1

        # Convert frame to PIL Image
        pil_img = Image.fromarray(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB))

        # Preprocess the image
        img = transform(pil_img).unsqueeze(0)

        # Make the prediction
        with torch.no_grad():
            output = model(img)
            predicted_count = int(output.detach().cpu().sum().numpy())

        # Accumulate the total crowd count
        total_crowd_count += predicted_count

        # Convert the density map to a numpy array for visualization
        density_map = output.squeeze(0).squeeze(0).cpu().numpy()

        # Normalize the density map for visualization
        density_map_normalized = (density_map - density_map.min()) / (density_map
        density_map_colored = cm.jet(density_map_normalized)[:, :, :3]  # Apply c

        # Resize the density map to match the original frame size
        density_map_resized = cv2.resize(density_map_colored, (frame.shape[1], fr
```

```
        # Overlay the density map on the original frame
        overlay = cv2.addWeighted(frame, 0.6, (density_map_resized * 255).astype(

        # Show the crowd count on the frame
        cv2.putText(overlay, f"Crowd Count: {predicted_count}", (10, 30), cv2.FON

        # Show the frame with the density map overlay
        cv2_imshow(overlay)

        # Break the loop if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

    # Output the total crowd count for the 18 frames
    print(f"Total Crowd Count for 18 frames between 0.14s and 0.32s: {total_crowd
    avg_crowd_count = total_crowd_count / frame_count
    print(f"Average Crowd Count: {avg_crowd_count:.2f}")

# Example usage:
process_video('/content/Drone Footage of Canberra s HISTORIC Crowd.mp4')
```


Crowd Count: 277


Crowd Count: 258

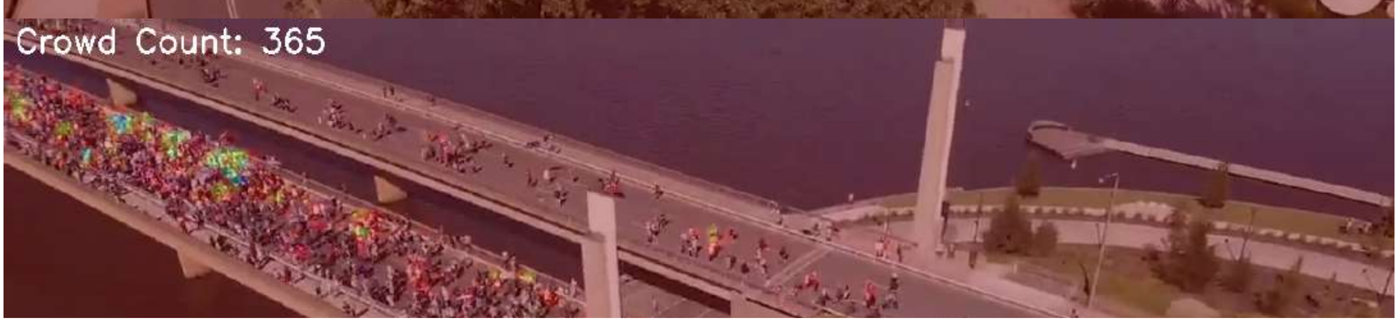

Crowd Count: 367
Crowd Count: 334


Crowd Count: 383


Crowd Count: 280


Crowd Count: 343


Crowd Count: 360


Crowd Count: 365

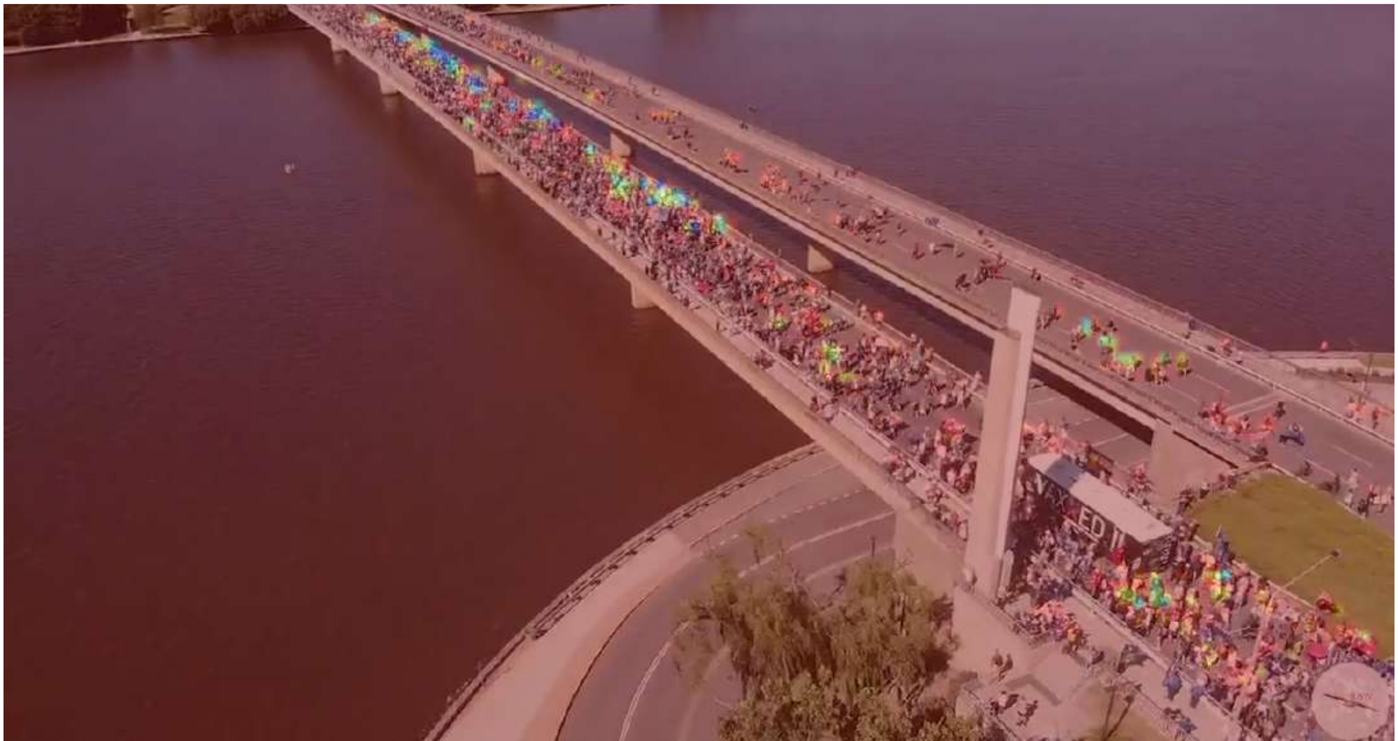

Crowd Count: 246


Crowd Count: 231


Crowd Count: 172


Crowd Count: 141


Crowd Count: 144




Crowd Count: 107


Crowd Count: 113

```
Total Crowd Count for 18 frames between 0.14s and 0.32s: 4728
Average Crowd Count: 262.67
```

Outputs a density map. In a crowd density map, each pixel contributes a fractional count to the total crowd size, making it very effective for counting people in images where individuals are not easily separable, such as in large or dense crowds.

CSRNet is designed to output density maps that work better for crowd counting tasks, while YOLOv3's bounding box output is not well suited for estimating the number of people in crowded scenes.

Summary of Key Reasons Why CSRNet Works Better for Crowd Counting:

Crowd density map: CSRNet is designed to produce a density map, making it far more effective for estimating the number of people in densely packed scenes. Scale and occlusion handling: CSRNet handles varying scales (people far away or close) and occlusions better than YOLOv3. Not reliant on bounding boxes: CSRNet doesn't need to detect each individual person as a distinct object, whereas YOLOv3 struggles with overlapping or occluded objects. Task-specific design: CSRNet is specifically designed for crowd counting, while YOLOv3 is a general object detector, making CSRNet more accurate in this specialized task. In conclusion, CSRNet is better suited for crowd counting tasks, particularly in dense crowds, due to its specialized architecture that focuses on generating density maps rather than relying on object detection like YOLOv3.