Approach Document

Dataset Collection and Preparation

At first, I started looking for publicly available datasets for this project but couldn't find anything promising. Most of the datasets were focused on detecting Pepsi or Coke bottles instead of just the logos. Therefore, I shifted my approach to finding datasets specifically for individual Pepsi and Coca-Cola logos. I found 3-4 datasets, so I combined all the images from these datasets.

The next step was working on the labels. Primarily, the datasets for both classes had 0 indexing as they were individual databases. I wrote a Python script to modify the labels, changing the indexing from 0 to 1 for Coca-Cola and leaving Pepsi labels as they were. This script helped standardize the labels across the combined dataset.

```
def process_files(folder_path):
    for filename in os.listdir(folder_path):
        if filename.endswith(".txt"):
            file_path = os.path.join(folder_path, filename)
            with open(file_path, 'r') as file:
                lines = file.readlines()
            with open(file_path, 'w') as file:
                for line in lines:
                    parts = line.strip().split()
                    if parts[0] == '0':
                        parts[0] = '1'
                    file.write(" ".join(parts) + "\n")
```

However, the dataset still had some issues. Some images in the Pepsi dataset also contained Coca-Cola logos that were not labeled, which could potentially lower the precision of the model. To address this, I manually cleaned the dataset by removing these problematic images to ensure better accuracy.

Training

Once the dataset was ready, it was time for training. I decided to use the YOLOv8.2 model due to its superior performance and also i have worked on some projects like `Forest-Fire Detection` (https://github.com/NeuralNoble/fire-detection-system and `Fire-arm detection` (https://github.com/NeuralNoble/threat_detection) . Initially, I chose the larger YOLOv8l model, but I dropped it because of its slow convergence and slow inference speed for video processing. Instead, I opted for the YOLOv8n (nano) model.

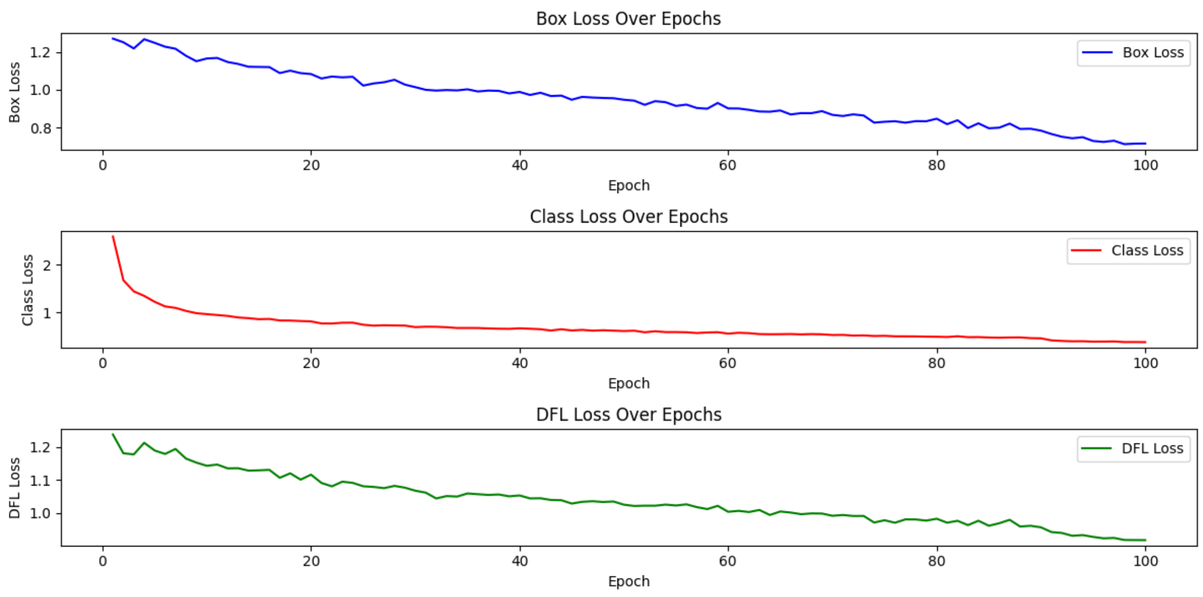
I uploaded the dataset to my Google Drive and mounted it in Google Colab for training the model. Initially, I chose batch gradient descent, but it converged very slowly, causing me to run out of compute units in Collab. Consequently, I switched to mini-batch gradient descent with a batch size of 16. However, the small batch size made the data noisy. Ultimately, I settled on a batch size of 32 and 100 epochs. This choice allowed the model to converge faster while introducing a bit of noise to improve generalization.

I also considered augmenting the data, but since most images in the dataset already covered all orientations, additional augmentation reduced model performance. After 100 epochs, the model summary (fused) 168 layers, 3,006,038 parameters, 0 gradients, and 8.1 GFLOPs.

Here are the performance metrics:

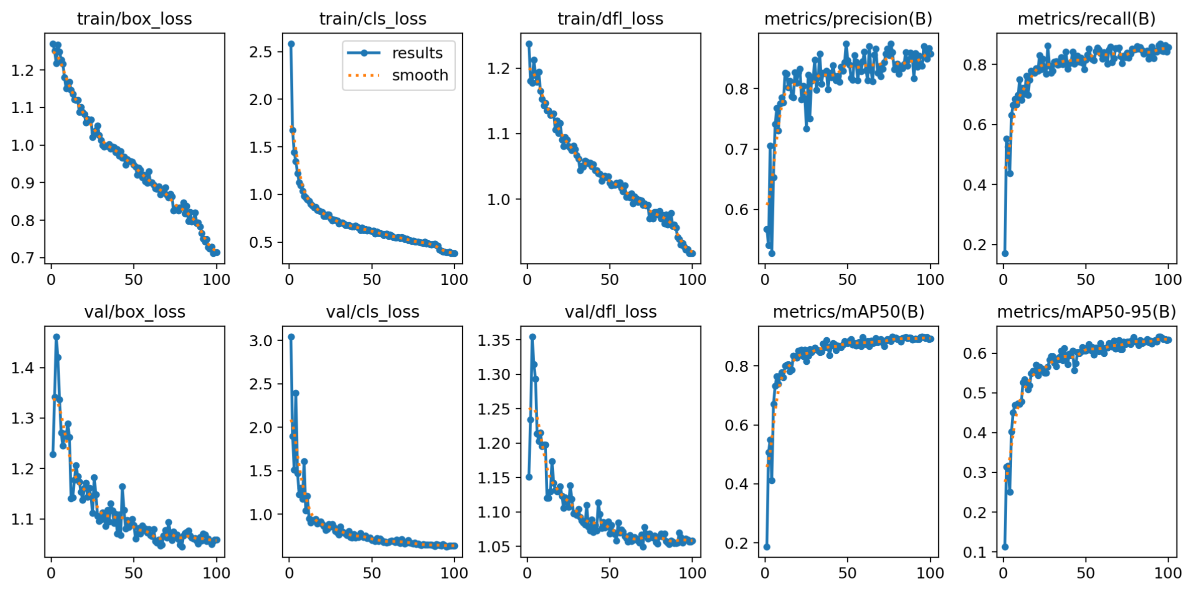
- Class: All
  - Precision: 0.838
  - Recall: 0.87
  - mAP@50: 0.899
  - mAP@50-95: 0.64

The loss was still decreasing, indicating that additional training epochs could further improve the model's performance.



Analysis

- **Overall Performance**: The model demonstrates strong performance with high precision and recall, particularly in detecting Pepsi logos.
- **Class Imbalance Impact**: Pepsi instances have higher precision and recall compared to Coca-Cola, which might indicate that the model performs better with Pepsi images.
- **Inference Speed**: The model is efficient, with a total processing time of 7.8ms per image (sum of preprocessing, inference, and post-processing times).



Pipeline

After my model was ready i started working on pipeline(`pipelinev2.py`) . It involves extracting frames and timestamps, processing detection results to compute logo size and distance from the frame center, and saving structured data to a JSON file.

Steps

1. **Frame Extraction and Timestamps**
    - Utilizes the `av` library to decode frames and capture timestamps from the input video file.
    - Frames are extracted as RGB images and timestamps are recorded for each frame.
2. **YOLO Model Loading**
    - Loads my custom-trained YOLO model using the `ultralytics` library.
    - The model is initialized and moved to either GPU or CPU based on device availability.
3. **Logo Detection**
    - Applies the custom trained YOLO model to predict bounding boxes and labels for each frame.
    - Returns detection results including coordinates and confidence scores for detected objects.
4. **Processing Detections**
    - Computes the center of each frame using its dimensions (height, width).
    - For each detected logo, calculates:
        - **Timestamp**: When the logo was detected in the video.
        - **Size**: Diagonal length of the logo's bounding box as a measure of its spatial extent.
        - **Distance from Center**: Euclidean distance from the logo's center to the frame center.
    - Rounds these values to two decimal places for clarity.

5. **Data Organization**
    - Structures processed information into a dictionary format:
        - `"Pepsi_info"`: List of dictionaries containing details for each detected Pepsi logo.
        - `"CocaCola_info"`: List of dictionaries containing details for each detected Coca-Cola logo.
6. **Output to JSON**
    - Saves the structured data into a JSON file (`output.json`) for further analysis or visualization.

```
info = {
"timestamp": round(float(timestamp), 2),
 "size": round(float(bbox_size), 2),
"distance_from_center": round(float(distance_from_center), 2)
}
```

Deployment and WebApp

Once the entire pipeline was ready, I created a user interface (UI) for the pipeline using Streamlit. The UI allows users to upload a video file and receive an output JSON file that contains the detection results. The JSON file includes timestamps of logo detections, the size of the bounding box diagonals, and the distances from the center of the frame.

In the WebApp:

1. **Model Caching**: To improve performance, the YOLO model is cached in the browser, reducing loading times for subsequent video uploads.
2. **Video Upload**: Users can upload their video files directly through the Streamlit interface.
3. **Processing**: The uploaded video is processed through the pipeline, extracting frames, detecting logos, and calculating the necessary metrics.
4. **Output**: Users can download the resulting JSON file, which contains detailed detection information:

- **Timestamps**: When each logo was detected.
- **Bounding Box Size**: The diagonal length of each detected logo's bounding box.
- **Distance from Center**: The Euclidean distance from the logo's center to the center of the frame.

This deployment with Streamlit ensures an easy-to-use and interactive experience for end-users, enabling efficient logo detection in video files with minimal effort. The inclusion of model caching in the browser further enhances the usability by providing faster processing times.