Solution Document – Beyblade Battle Analysis

Syahvan Alviansyah Diva Ritonga

A. Dataset

The dataset for this project was created using Roboflow, a platform that streamlines the process of building and managing computer vision datasets. The dataset consists of annotated images featuring various beyblades, launchers, and hands in dynamic battle scenarios. The annotated dataset can be accessed and downloaded from here.

B. Model Selection

For this project, YOLOv8 model was selected for detecting beyblades, hands, and launchers due to its real-time performance and high accuracy in object detection tasks. YOLO is known for its ability to detect multiple objects within an image in a single pass, making it particularly suitable for dynamic environments like beyblade battles where various objects are constantly in motion.

The nano version of YOLOv8 was utilized for its speed and efficiency, allowing for fast inference with lower computational requirements. The trained weights for the YOLOv8 nano model can be downloaded from here.

C. Model Peformance

The performance of the YOLOv8 model was evaluated using standard metrics such as Precision, Recall, F1-Score, and Mean Average Precision (mAP). After training and validation, the model achieved the following metrics, which can be seen in the table below.

Table 1. Model peformance metrics.

Class	***Recall***	***Precision***	**mAP@50**	**mAP@50-95**
Beyblade	0.988	0.997	0.995	0.876
Hand	0.873	0.973	0.981	0.843
Launcher	1	0.918	0.995	0.82
All	0.954	0.963	0.99	0.847

The metrics indicate that the model effectively identifies and classifies Beyblades, hands, and launchers with a high degree of accuracy and reliability. However, when tracking objects using algorithms like ByteTrack or BoT-SORT, whether through Ultralyitcs' implementation or with Supervision, the resulting object IDs tend to be unstable and change frequently. This instability poses a challenge for maintaining consistent tracking over time. Therefore, to achieve more stable and reliable IDs, I utilized K-Means Clustering to segment beyblades into two distinct IDs, representing different teams.

D. Logic Behind The Program

The program's logic is structured into four main components: Utilities, Tracker, Assigner, and Battle. Each of these components plays a crucial role in ensuring accurate detection, tracking, and analysis of beyblade battles.

This project begins by reading the input video and initializing a `` `Tracker` `` object using a tracking model to detect and track Beyblade objects in each frame. After obtaining the tracking data, the system adds the object positions and identifies the Beyblade team colors using an `` `Assigner` `` object. Subsequently, a `` `Battle` `` object is utilized to analyze the battle status and collect statistics, such as battle time and the winner, which are saved in a CSV file. The battle results are displayed in the console, and the video frames are annotated to show tracking and battle status before being saved as an output video and winner image. This workflow creates an effective system for analyzing and documenting Beyblade battle results from video footage.

1. Utilities

The utilities include functions for video processing and bounding box handling, which are essential for extracting, saving, and analyzing video data during beyblade battles.

1. Video Processing

```
def read_video(video_path):
    cap = cv2.VideoCapture(video_path)
    frames = []
    while True:
        ret, frame = cap.read()
        if not ret:
            break
        frames.append(frame)
    return frames

def save_video(ouput_video_frames,output_video_path):
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    out = cv2.VideoWriter(output_video_path, fourcc, 30,
(ouput_video_frames[0].shape[1],
ouput_video_frames[0].shape[0]))
    for frame in ouput_video_frames:
        out.write(frame)
    out.release()
```

The `` `read_video` `` function extracts frames from a video file specified by the input path using OpenCV's VideoCapture. It continuously reads each frame until the end of the video is reached, appending each frame to a list. This list of frames is returned for further processing.

The `save_video` function saves processed frames back into a new video file. By initializing a VideoWriter with the XVID codec, it writes each frame from the provided list to the specified output path. This functionality enables users to visualize the analysis results in video format.

2. Bounding Box Processing

```
def get_center_of_bbox(bbox):
    x1,y1,x2,y2 = bbox
    return int((x1+x2)/2),int((y1+y2)/2)

def is_overlapping(bbox1, bbox2):
    if bbox1 and bbox2:
        x1_min, y1_min, x1_max, y1_max = bbox1
        x2_min, y2_min, x2_max, y2_max = bbox2

        x_inter_min = max(x1_min, x2_min)
        y_inter_min = max(y1_min, y2_min)
        x_inter_max = min(x1_max, x2_max)
        y_inter_max = min(y1_max, y2_max)

        width_inter = max(0, x_inter_max - x_inter_min)
        height_inter = max(0, y_inter_max - y_inter_min)
        area_inter = width_inter * height_inter

        if area_inter == 0:
            return False

        area_obj1 = (x1_max - x1_min) * (y1_max - y1_min)
        area_obj2 = (x2_max - x2_min) * (y2_max - y2_min)
        area_union = area_obj1 + area_obj2 - area_inter

        IoU = area_inter / area_union

        return IoU > 0
    else:
        return False
```

The `get_center_of_bbox` function calculates the center point of a bounding box defined by its coordinates (x1, y1, x2, y2). This center point is essential for tracking the position of objects.

The `is_overlapping` function checks if two bounding boxes overlap by calculating their Intersection over Union (IoU). If the IoU is greater than zero, it indicates an interaction between the objects, which is vital for tracking and detecting launched beyblades, collisions during beyblade battles, and when a beyblade is picked up by hand. This function helps maintain continuity in object tracking across frames.

2. **Tracker**
   The Tracker component of the program is responsible for maintaining the state and identity of each detected object (beyblades, hands, launchers) throughout the video.

```
class Tracker:
    def __init__(self, model_path):
        self.model = YOLO(model_path)
        self.tracker = sv.ByteTrack()

    def add_position_to_tracks(self,tracks):
        for object, object_tracks in tracks.items():
            for frame_num, track in enumerate(object_tracks):
                for track_id, track_info in track.items():
                    bbox = track_info['bbox']
                    position = get_center_of_bbox(bbox)
                    tracks[object][frame_num][track_id]['positio
n'] = position

    def detect_frames(self, frames):
        batch_size=20
        detections = []
        for i in range(0,len(frames),batch_size):
            detections_batch =
self.model.predict(frames[i:i+batch_size],conf=0.1)
            detections += detections_batch
        return detections

    def get_object_tracks(self, frames, read_from_stub=False,
stub_path=None):

        if read_from_stub and stub_path is not None and
os.path.exists(stub_path):
            with open(stub_path,'rb') as f:
                tracks = pickle.load(f)
            return tracks
        detections = self.detect_frames(frames)

        tracks={
            "Beyblade":[],
            "Hand":[],
            "Launcher":[]
        }

        for frame_num, detection in enumerate(detections):
            cls_names = detection.names
            cls_names_inv = {v:k for k,v in cls_names.items()}

            # Covert to supervision Detection format
            detection_supervision =
sv.Detections.from_ultralytics(detection)

            # Track Objects
            detection_with_tracks =
self.tracker.update_with_detections(detection_supervision)

            tracks["Beyblade"].append({})
            tracks["Hand"].append({})
            tracks["Launcher"].append({})

            for frame_detection in detection_with_tracks:
                bbox = frame_detection[0].tolist()
                cls_id = frame_detection[3]
                track_id = frame_detection[4]

                if cls_id == cls_names_inv['Beyblade']:
                    tracks["Beyblade"][frame_num][track_id] =
{"bbox":bbox}

            for frame_detection in detection_supervision:
                bbox = frame_detection[0].tolist()
                cls_id = frame_detection[3]

                if cls_id == cls_names_inv['Hand']:
                    tracks["Hand"][frame_num][1] = {"bbox":bbox}
                if cls_id == cls_names_inv['Launcher']:
                    tracks["Launcher"][frame_num][1] =
{"bbox":bbox}

        if stub_path is not None:
            with open(stub_path,'wb') as f:
                pickle.dump(tracks,f)

        return tracks

    def draw_triangle(self,frame,bbox,color,track_id=None):
        y = int(bbox[1])
        x,_ = get_center_of_bbox(bbox)

        triangle_points = np.array([
            [x,y],
            [x-10,y-20],
            [x+10,y-20],
        ])
        cv2.drawContours(frame, [triangle_points],0,color,
cv2.FILLED)
        cv2.drawContours(frame, [triangle_points],0,(0,0,0), 2)

        rectangle_width = 140
        x1_rect = x - rectangle_width//2
        x2_rect = x + rectangle_width//2
        y1_rect = y - 50
        y2_rect = y - 30

        if track_id is not None:
            cv2.rectangle(frame,
                           (int(x1_rect),int(y1_rect) ),
                           (int(x2_rect),int(y2_rect)),
                           color,
                           cv2.FILLED)

            x1_text = x1_rect + 10

            cv2.putText(
                frame,
                f"Beyblade #{track_id}",
                (int(x1_text),int(y1_rect+15)),
                cv2.FONT_HERSHEY_SIMPLEX,
                0.6,
                (255,255,255),
                2
            )

        return frame

    def draw_annotations(self, video_frames, tracks):
        output_video_frames= []
        for frame_num, frame in enumerate(video_frames):
            beyblade_dict = tracks["Beyblade"][frame_num]

            # Draw Beyblade
            for track_id, beyblade in beyblade_dict.items():
                color = beyblade.get("beyblade_color",(0,0,255))
                team = beyblade.get("team",1)
                bbox = beyblade['bbox']

                frame =
self.draw_triangle(frame,bbox,color,team)

            output_video_frames.append(frame)

        return output_video_frames
```

The `__init__` function initializes the tracker by loading the YOLO model for detecting objects and setting up the ByteTrack tracker to maintain object identities across frames. These tools are essential for identifying and following Beyblades, hands, and launchers during the video analysis.

The `add_position_to_tracks` function calculates the center position of each bounding box for tracked objects, adding this information to the tracking data. This positional information is later utilized for tasks such as determining whether an object is located within a predefined polygonal area.

The `detect_frames` function processes the video frames in batches, using the YOLO model to generate object detections. By grouping frames into batches of 20, the detection process becomes more efficient, reducing the computational load when handling large video files.

The `get_object_tracks` function is responsible for detecting objects and tracking them across video frames. It either reads precomputed tracking data from a stub file or processes the frames using YOLO. It then organizes the detected objects (beyblades, hands, and launchers) into a tracking structure, assigning track IDs to maintain consistency across frames.

The `draw_triangle` function visualizes a triangle marker above each detected beyblade, showing its track ID and team color. This graphical representation helps users easily identify individual beyblades during the analysis by visually distinguishing them in the video.

Lastly, the `draw_annotations` function applies all visual markers and labels to each video frame. It extracts the tracked beyblade data and overlays triangles and team information on the frames, generating a final output video with clear visualizations for the detected objects.

3. **Assigner**
The Assigner component is responsible for classifying detected Beyblades into two teams using K-Means clustering.

```
class Assigner:
    def __init__(self):
        self.beyblade_colors = {}
        self.beyblade_assigner_dict = {}

    def get_clustering_model(self,image):
        # Reshape the image to 2D array
        image_2d = image.reshape(-1,3)

        # Preform K-means with 2 clusters
        kmeans = KMeans(n_clusters=2, init="k-means++",n_init=1)
        kmeans.fit(image_2d)

        return kmeans

    def get_beyblade_color(self,frame,bbox):
        image =
frame[int(bbox[1]):int(bbox[3]),int(bbox[0]):int(bbox[2])]

        # Get Clustering model
        kmeans = self.get_clustering_model(image)

        # Get the cluster labels for each pixel
        labels = kmeans.labels_

        # Reshape the labels to the image shape
        clustered_image =
labels.reshape(image.shape[0],image.shape[1])

        # Get the beyblade cluster
        corner_clusters =
[clustered_image[0,0],clustered_image[0,-1],clustered_image[-
1,0],clustered_image[-1,-1]]
        non_beyblade_cluster =
max(set(corner_clusters),key=corner_clusters.count)
        beyblade_cluster = 1 - non_beyblade_cluster

        beyblade_color =
kmeans.cluster_centers_[beyblade_cluster]

        return beyblade_color

    def assign_beyblade_color(self,frame, beyblade_detections):

        beyblade_colors = []
        for _, beyblade_detection in
beyblade_detections.items():
            bbox = beyblade_detection["bbox"]
            beyblade_color
=  self.get_beyblade_color(frame,bbox)
            beyblade_colors.append(beyblade_color)

        kmeans = KMeans(n_clusters=2, init="k-
means++",n_init=10)
        kmeans.fit(beyblade_colors)

        self.kmeans = kmeans

        self.beyblade_colors[1] = kmeans.cluster_centers_[0]
        self.beyblade_colors[2] = kmeans.cluster_centers_[1]

    def get_beyblade_team(self,frame,beyblade_bbox,beyblade_id):
        if beyblade_id in self.beyblade_assigner_dict:
            return self.beyblade_assigner_dict[beyblade_id]
        beyblade_color = 
self.get_beyblade_color(frame,beyblade_bbox)

        beyblade_id = 
self.kmeans.predict(beyblade_color.reshape(1,-1))[0]
        beyblade_id+=1

        self.beyblade_assigner_dict[beyblade_id] = beyblade_id

        return beyblade_id
```

The `__init__` function initializes the `Assigner` class by creating two dictionaries: one to store the colors of detected Beyblades and another to map Beyblade IDs to their respective teams. This setup ensures that each Beyblade can be consistently assigned to a team during the battle.

The `get_clustering_model` function performs K-Means clustering on a cropped image of the Beyblade. It reshapes the image into a 2D array where each pixel is a data point, and then applies K-Means clustering to group the pixels into two clusters, which helps differentiate the Beyblade from the background.

The `get_beyblade_color` function extracts the color of a Beyblade by isolating its bounding box from the frame, applying K-Means clustering, and determining which cluster represents the Beyblade. It uses the cluster centers to identify the dominant color of the Beyblade for later assignment to a team.

The `assign_beyblade_color` function runs K-Means clustering on the colors of detected Beyblades within the frame. It assigns two cluster centers, each representing the average color of the Beyblades on opposing teams. These clusters are stored to assign colors to each Beyblade for future frames.

The `get_beyblade_team` function assigns each Beyblade to a team based on its color. If a Beyblade's team is already known, it returns the previously assigned team. Otherwise, it predicts the team using the K-Means clustering model and updates the `beyblade_assigner_dict` to store the result for future reference.

4. **Battle**
   The Battle component handles the core logic of analyzing the Beyblade battle. It processes the detected and tracked data to derive key battle metrics.

```
class Battle:
    def __init__(self):
        self.vertices =
np.array([(779,0),(175,400),(245,1080),(1750,1080),(1820,400),(1
250,0)])
        self.start_battle_time = None
        self.end_battle_time = None
        self.beyblade_time = {1:0,2:0}
        self.battle_time = 0
        self.winner = None
        self.winner_bbox = None
        self.winner_frame_num = None
        self.total_collision = 0

    def add_beyblade_status(self,tracks,video_frames):
        prev_gray = None
        for frame_num, beyblade_track in
enumerate(tracks['Beyblade']):
            frame_gray = cv2.cvtColor(video_frames[frame_num],
cv2.COLOR_BGR2GRAY)
            for beyblade_id, track in beyblade_track.items():
                position = track['position']
                tracks['Beyblade'][frame_num][beyblade_id]['insi
de_polygon'] = cv2.pointPolygonTest(self.vertices, position,
False) >= 0

                hand_bbox = tracks["Hand"][frame_num].get(1,
{}).get('bbox', None)
                launcher_bbox =
tracks["Launcher"][frame_num].get(1, {}).get('bbox', None)
                IoU_hand = is_overlapping(track['bbox'],
hand_bbox)
                IoU_launcher = is_overlapping(track['bbox'],
launcher_bbox)
                if IoU_hand or IoU_launcher:
                    tracks['Beyblade'][frame_num][beyblade_id]['
is_taken'] = True
                else:
                    tracks['Beyblade'][frame_num][beyblade_id]['
is_taken'] = False

                if prev_gray is not None:
                    # Optical Flow Farneback
                    flow =
cv2.calcOpticalFlowFarneback(prev_gray, frame_gray, None, 0.5,
3, 15, 3, 5, 1.2, 0)
                    magnitude = np.sqrt(flow[..., 0]**2 +
flow[..., 1]**2)
                    x_min = max(0, int(position[0]) - 5)
                    x_max = min(magnitude.shape[1],
int(position[0]) + 5)
                    y_min = max(0, int(position[1]) - 5)
                    y_max = min(magnitude.shape[0],
int(position[1]) + 5)
                    movement = np.mean(magnitude[y_min:y_max,
x_min:x_max])
                    if movement > 1.0:
                        tracks['Beyblade'][frame_num][beyblade_i
d]['is_rotating'] = True
                    else:
                        tracks['Beyblade'][frame_num][beyblade_i
d]['is_rotating'] = False
                else:
                    tracks['Beyblade'][frame_num][beyblade_id]['
is_rotating'] = False
            prev_gray = frame_gray

    def check_battle(self,frame_num,beyblade_track):
        beyblade_inside_polygon = []
        for beyblade_id, track in beyblade_track.items():
            inside_polygon = track['inside_polygon']
            taken = track['is_taken']
            rotate = track['is_rotating']
            if inside_polygon and not taken and rotate:
                team = track['team']
                beyblade_inside_polygon.append(team)
                if self.start_battle_time:
                    time_now = frame_num / 30
                    self.beyblade_time[team] = time_now -
self.start_battle_time
                if self.winner == team:
                    self.winner_bbox = track['bbox']
                    self.winner_frame_num = frame_num
        if len(beyblade_inside_polygon) >= 2:
            if not self.start_battle_time:
                self.start_battle_time = frame_num / 30
                return 1, self.battle_time, self.beyblade_time
            else:
                self.end_battle_time = frame_num / 30
                self.battle_time = self.end_battle_time - 
self.start_battle_time
                return 1, self.battle_time, self.beyblade_time
        elif len(beyblade_inside_polygon) == 1 and 
self.start_battle_time:
            self.winner = beyblade_inside_polygon[0]
            return 2, self.battle_time, self.beyblade_time
        else:
            if self.start_battle_time and self.winner:
                return 2, self.battle_time, self.beyblade_time
            else:
                return 0, self.battle_time, self.beyblade_time

    def get_battle_stat(self, tracks, battle_log_path):
        battle_stat = {
            'battle_status':[],
            'battle_time':[],
            'beyblade1_time':[],
            'beyblade2_time':[],
            'collision':[],
            'total_collision':[]
        }

        battle_log = pd.DataFrame(columns=["frame_num", 
"battle_status", "collision"])

        for frame_num, beyblade_track in 
enumerate(tracks['Beyblade']):
            battle_stat["battle_status"].append({})
            battle_stat["battle_time"].append({})
            battle_stat["beyblade1_time"].append({})
            battle_stat["beyblade2_time"].append({})
            battle_stat["collision"].append({})
            battle_stat["total_collision"].append({})
            battle_status, battle_time, beyblade_time =
self.check_battle(frame_num,beyblade_track)
            battle_stat['battle_status'][frame_num] =
battle_status
            battle_stat['battle_time'][frame_num] = battle_time
            battle_stat['beyblade1_time'][frame_num] =
beyblade_time[1]
            battle_stat['beyblade2_time'][frame_num] =
beyblade_time[2]

            beyblade_bbox = []
            for beyblade_id, track in beyblade_track.items():
                beyblade_bbox.append(track['bbox'])
            if len(beyblade_bbox) == 2:
                IoU_beyblade = is_overlapping(beyblade_bbox[0],
beyblade_bbox[1])
                battle_stat['collision'][frame_num] =
IoU_beyblade
                if IoU_beyblade:
                    self.total_collision += 1
                battle_stat['total_collision'][frame_num] =
self.total_collision
            else:
                battle_stat['total_collision'][frame_num] =
self.total_collision
                battle_stat['collision'][frame_num] = False

            battle_log_data =
[frame_num,battle_status,battle_stat['collision'][frame_num]]
            battle_log_series = pd.Series(battle_log_data,
index=battle_log.columns)
            battle_log = pd.concat([battle_log,
battle_log_series.to_frame().T], ignore_index=True)

        battle_log.to_csv(battle_log_path, index=False)

        return battle_stat

    def draw_stat(self, video_frames, battle_stat, tracks):
        output_video_frames= []
        for frame_num, frame in enumerate(video_frames):
            cv2.polylines(frame, [self.vertices], isClosed=True,
color=(255, 0, 0), thickness=2)

            battle_status =
battle_stat["battle_status"][frame_num]
            battle_time = battle_stat["battle_time"][frame_num]
            beyblade1_time =
battle_stat["beyblade1_time"][frame_num]
            beyblade2_time =
battle_stat["beyblade2_time"][frame_num]
            total_collision =
battle_stat["total_collision"][frame_num]

            if battle_status == 0:
                frame = cv2.putText(frame,f"Waiting
Opponent...",(50,50), cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),3)
            else:
                frame = cv2.putText(frame,f"Battle Time:
{battle_time:.2f} s",(50,50),
cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),3)
                frame = cv2.putText(frame,f"Beyblade 1 Time:
{beyblade1_time:.2f} s",(50,90),
cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),3)
                frame = cv2.putText(frame,f"Beyblade 2 Time:
{beyblade2_time:.2f} s",(50,130),
cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),3)
                frame = cv2.putText(frame,f"Total Collision:
{total_collision}",(50,170),
cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),3)

            if frame_num == self.winner_frame_num:
                x1, y1, x2, y2 = [int(value) for value in
self.winner_bbox]
                winner_img = frame[y1:y2, x1:x2]

            output_video_frames.append(frame)

        return output_video_frames, winner_img
```

The `__init__` function initializes the `Battle` class, defining the vertices of the battle arena, tracking the start and end times of the battle, and setting up counters for each Beyblade's time spent battling. It also initializes variables for tracking the winner and the total number of collisions that occur during the battle.

The `add_beyblade_status` function processes the status of Beyblades for each frame in a video. It converts each frame to grayscale for optical flow analysis and checks whether each Beyblade is inside the defined polygon (battle arena). It also assesses whether a Beyblade is currently being launched based on its bounding box's overlap with the bounding boxes of the hand and launcher. Optical flow is calculated to determine if the Beyblade is rotating based on movement magnitude.

The `check_battle` function evaluates the current state of the battle. It collects information about which Beyblades are inside the polygon, whether they are currently being launched, and if they are rotating. If two Beyblades are inside the arena and in motion, it starts the battle timer. If one Beyblade remains inside while the other leaves, it declares the remaining Beyblade as the winner. The function returns different statuses indicating the state of the battle.

The `get_battle_stat` function compiles battle statistics over each frame, including the battle status, battle times for each Beyblade, and collision counts. It uses the `check_battle` method to determine the current state of the battle for each frame and logs collision events between Beyblades. The statistics are then saved into a CSV file for future analysis.

The `draw_stat` function visualizes the battle statistics on the video frames. It overlays text displaying the battle status, the time each Beyblade has been active, and the total number of collisions. It highlights the winning Beyblade by capturing its bounding box if the current frame indicates a win. The modified frames are collected and returned for further processing or display.

E. Result

The project generates several outputs that document the results of the Beyblade battles:

1. **output_video.avi**: This video file contains the annotated video frames, showcasing the tracked Beyblades, their movements, and battle statistics overlayed throughout the duration of the battle. The output video can be accessed and downloaded from here.
2. **battle_results.csv**: This CSV file summarizes the final battle statistics, including the total battle time, the winner of the battle, the time each Beyblade spent in the arena, the remaining time difference between the two Beyblades, and the total number of collisions that occurred during the battle.
3. **battle_log.csv**: This file logs the battle statistics for each frame, providing a detailed account of the battle dynamics over time. It includes information on the battle status and collision occurrences, allowing for an in-depth analysis of the battle sequence.
4. **winner.jpg**: This image captures the winning Beyblade, providing a visual representation of the victor in the battle.

These outputs serve as comprehensive documentation of the battle's progression and outcomes, allowing for further analysis and review.

F. Future Improvement or Recommendation

One notable limitation of the current approach to team assignment using K-Means clustering is its inability to differentiate between Beyblades of the same color. In scenarios where both competing Beyblades share identical colors, K-Means fails to accurately cluster them, leading to confusion in team assignments. To overcome this limitation, a different technique should be considered, one that can handle situations where the objects are visually similar.