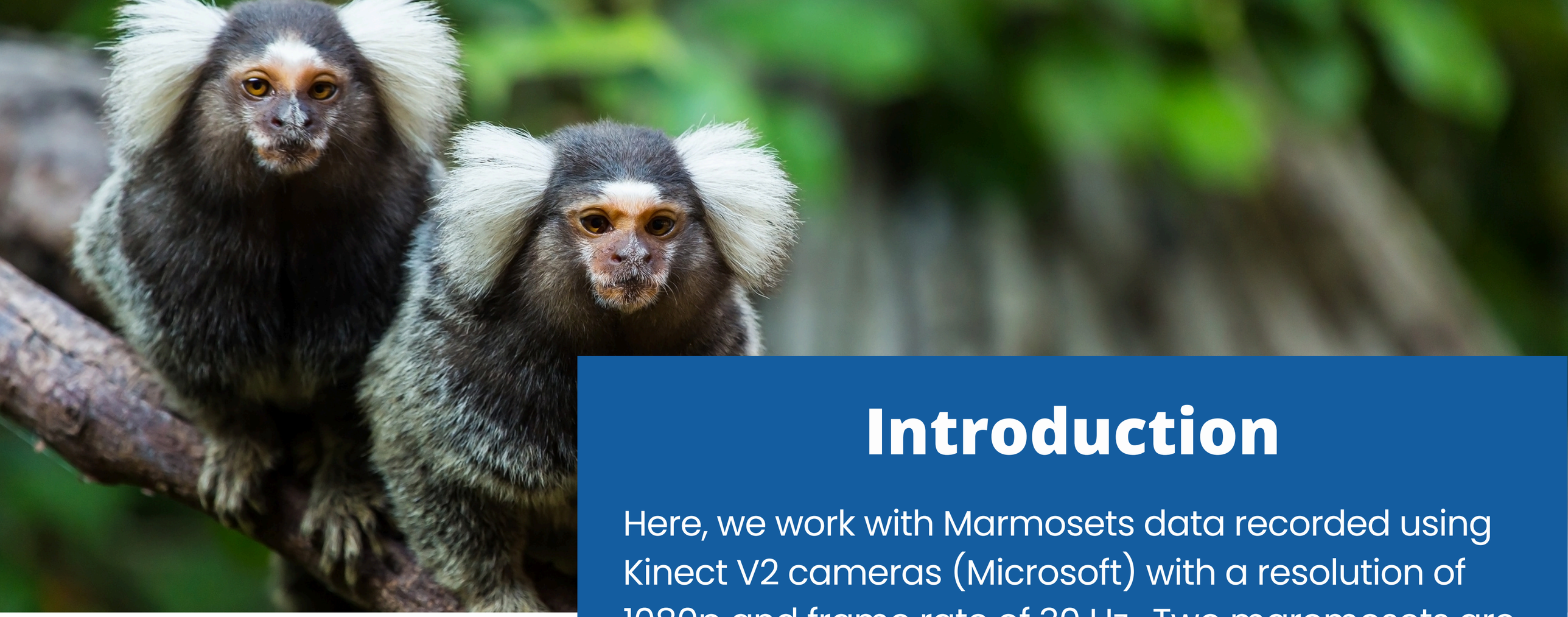
# Pose Estimation and Instance Segmentation Model

**Tanakrit Busarakul 6402008**
**Boonnisa Watcharapathorn 6402010**
**Tharathon Sopithikul 6402050**
**Natthakit Kiattimongkol 6402092**
**Warat Palpai 6402154**



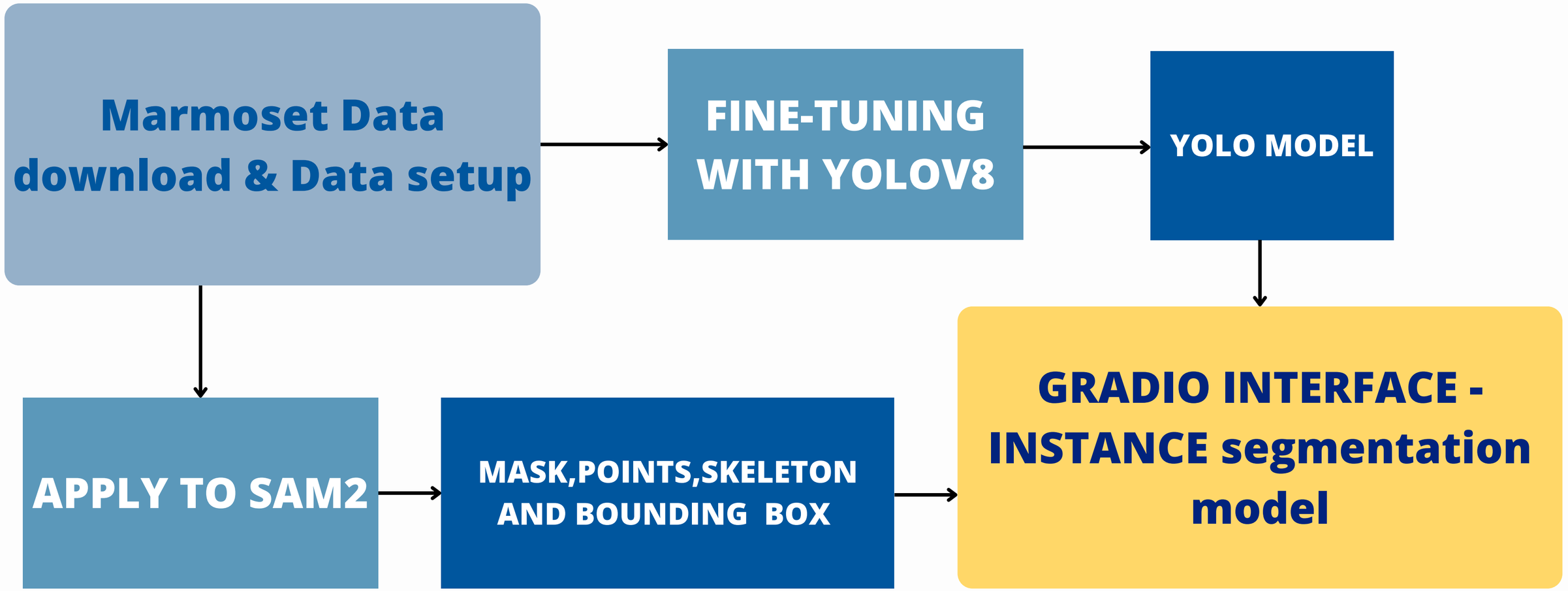
# Introduction

Here, we work with Marmosets data recorded using Kinect V2 cameras (Microsoft) with a resolution of 1080p and frame rate of 30 Hz.  Two maromosets are captured inside a cage and are monitored simultaneously.

# Objectives

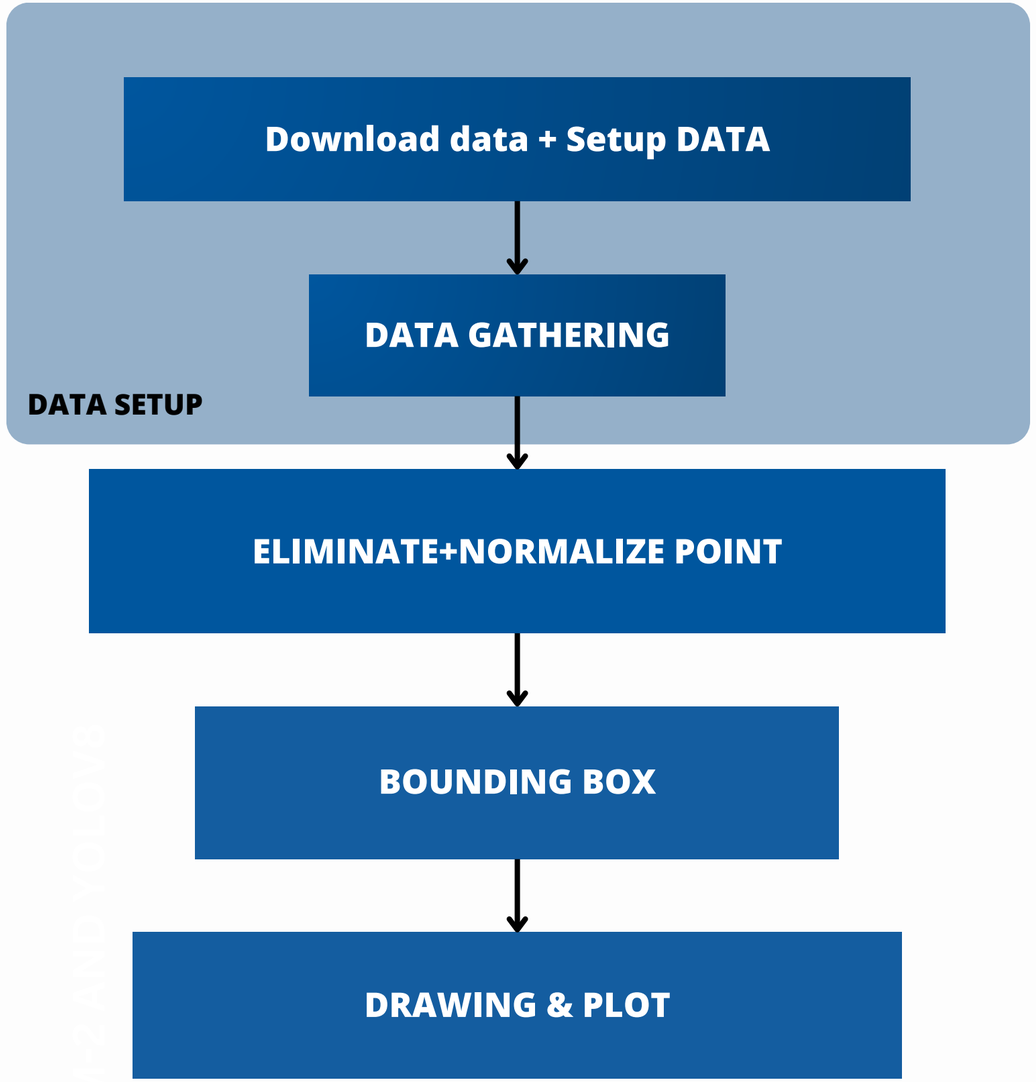
- Use ultralytics to fine-tune pose estimation
- Use SAM to generate mask from each frame with bounding box
- Create a joint pose estimation with instance segmentation model

# Workflow


Marmoset Data download & Data setup
FINE-TUNING WITH YOLOV8
YOLO MODEL
APPLY TO SAM2
MASK,POINTS,SKELETON AND BOUNDING BOX
GRADIO INTERFACE - INSTANCE segmentation model

# Data Cleaning

```
Index(['scorer', 'dlc', 'dlc.1', 'dlc.2', 'dlc.3', 'dlc.4', 'dlc.5', 'dlc.6',
       'dlc.7', 'dlc.8', 'dlc.9', 'dlc.10', 'dlc.11', 'dlc.12', 'dlc.13',
       'dlc.14', 'dlc.15', 'dlc.16', 'dlc.17', 'dlc.18', 'dlc.19', 'dlc.20',
       'dlc.21', 'dlc.22', 'dlc.23', 'dlc.24', 'dlc.25', 'dlc.26', 'dlc.27',
       'dlc.28', 'dlc.29', 'dlc.30', 'dlc.31', 'dlc.32', 'dlc.33', 'dlc.34',
       'dlc.35', 'dlc.36', 'dlc.37', 'dlc.38', 'dlc.39', 'dlc.40', 'dlc.41',
       'dlc.42', 'dlc.43', 'dlc.44', 'dlc.45', 'dlc.46', 'dlc.47', 'dlc.48',
       'dlc.49', 'dlc.50', 'dlc.51', 'dlc.52', 'dlc.53', 'dlc.54', 'dlc.55',
       'dlc.56', 'dlc.57', 'dlc.58', 'dlc.59', 'visibility_first_1',
       'visibility_first_2', 'visibility_first_3', 'visibility_first_4',
       'visibility_first_5', 'visibility_first_6', 'visibility_first_7',
       'visibility_first_8', 'visibility_first_9', 'visibility_first_10',
       'visibility_first_11', 'visibility_first_12', 'visibility_first_13',
       'visibility_first_14', 'visibility_first_15', 'visibility_second_1',
       'visibility_second_2', 'visibility_second_3', 'visibility_second_4',
       'visibility_second_5', 'visibility_second_6', 'visibility_second_7',
       'visibility_second_8', 'visibility_second_9', 'visibility_second_10',
       'visibility_second_11', 'visibility_second_12', 'visibility_second_13',
       'visibility_second_14', 'visibility_second_15'],
      dtype='object')
```



# SETUP : Download dataset+ Setup DATA

**Download dataset**

```python
# Download our training project:
# Import modules for HTTP requests, in-memory bytes, and zip files
import requests
from io import BytesIO
from zipfile import ZipFile
import os

# Create the directory if it doesn't exist
# Add os.makedirs('/content/test', exist_ok=True)  if test folder needed
os.makedirs('/home/badboy-002/Desktop/content', exist_ok=True)
os.makedirs('/home/badboy-002/Desktop/content/train', exist_ok=True)

# Specify URL for downloading the file, and GET request to the URL
url_record = 'https://zenodo.org/api/records/5849371'
response = requests.get(url_record)

# Check if the request was successful (code 200 = OK)
if response.status_code == 200:
    # Get the first file in the JSON response
    file = response.json()['files'][0]
    # Extract the title of the file from the 'key' attribute
    title = file['key']
    print(f"Downloading {title}...")

    # Download the file as a stream
    with requests.get(file['links']['self'], stream=True) as r:
        # Use ZipFile to open the stream and extract its contents
        with ZipFile(BytesIO(r.content)) as zf:
            zf.extractall(path='/home/badboy-002/Desktop/content/train') # Extract

else: # If the request was unsuccessful
    raise ValueError(f'The URL {url_record} could not be reached.')
```

**Setup Project folder**

```python
import os

# Directories to be created
main_dirs = ['train', 'val']
sub_dirs = ['images', 'labels']

os.makedirs('/home/badboy-002/Desktop/content/project', exist_ok=True)
# Base directory where you want to create these directories
base_dir = '/home/badboy-002/Desktop/content/project'

# Create the directories
for main_dir in main_dirs:
    for sub_dir in sub_dirs:
        os.makedirs(os.path.join(base_dir, main_dir, sub_dir), exist_ok=True)

print("Directories created successfully.")
```

**Check not NaN, and convert to float**

```python
import pandas as pd # Import pandas for handling CSV and DataFrame
from PIL import Image # Import PIL for potential image handling

# Load the CSV file
df = pd.read_csv('/home/badboy-002/Desktop/content/train/marmoset-dlc-2021-05-07/training-datasets/iteration-0/Ur

# Function to convert to float only if not NaN, thus empty cell will remain empty
def convert_to_float(x):
    try:
        return float(x) if pd.notna(x) else x
    except ValueError:
        return x

# Apply the conversion function to the specified part of the DataFrame
for col in df.columns[1:]:  # Columns > 1 (0-based index 2 and beyond)
    df.loc[3:, col] = df.loc[3:, col].apply(convert_to_float) # Apply the conversion from the 4th row onward
```

# Data GATHERING: Acquire x,y,vis for all points EP.1

```python
# Define column ranges for keypoints; column 1-31 = first marmoset, 31-61= second marmoset
keypoints_cols_first = df.columns[1:31]  # Columns for first class keypoints
keypoints_cols_second = df.columns[31:]  # Columns for second class keypoints

# Initialize lists to collect visibility flags [0 = Empty, 1 = Have value]
visibility_first = []
visibility_second = []

# Process rows starting from index 3
for index in range(3, len(df)):
    row = df.iloc[index]  # Select the current row

    # Extract keypoints for the first and second class
    keypoints_first_class = row[keypoints_cols_first]
    keypoints_second_class = row[keypoints_cols_second]

    # Generate visibility flags for every two cells (x, y coord)
    visibility_first_row = [
        int(any(pd.notna([keypoints_first_class.iloc[i], keypoints_first_class.iloc[i + 1]])))
        for i in range(0, len(keypoints_first_class), 2)
    ]
    visibility_second_row = [
        int(any(pd.notna([keypoints_second_class.iloc[i], keypoints_second_class.iloc[i + 1]])))
        for i in range(0, len(keypoints_second_class), 2)
    ]
    # Append visibility flags to lists
    visibility_first.append(visibility_first_row)
    visibility_second.append(visibility_second_row)

    # Replace NaN with 0.0 for normalization and convert to float64
    df.loc[index, keypoints_cols_first] = keypoints_first_class.fillna(0).astype(float)
    df.loc[index, keypoints_cols_second] = keypoints_second_class.fillna(0).astype(float)
```

**Separate Marmoset to ‘B’, ‘W’**

**Check if x,y is visible, and label empty cell**

**Then fill labelled empty cell with 0.0**

# Data GATHERING: Acquire x,y,vis for all points EP.2

```python
# Create visibility columns for rows starting from index 3
visibility_first_df = pd.DataFrame(visibility_first, columns=[f'visibility_first_{i+1}' for i in range(len(visibility_first[0]))])
visibility_second_df = pd.DataFrame(visibility_second, columns=[f'visibility_second_{i+1}' for i in range(len(visibility_second[0]))])

# Create DataFrame for the visibility columns and concatenate
visibility_first_df.index = df.index[3:]  # Align with the original DataFrame starting from index row 3
visibility_second_df.index = df.index[3:]  # Align with the original DataFrame starting from index row 3

# Concatenate visibility columns with the original DataFrame
df = pd.concat([df, visibility_first_df, visibility_second_df], axis=1)

# Display the DataFrame
df
```

**Make dataframe to store the data**

**Check the result df**

| scorer | dlc | dlc.1 | dlc.2 | dlc.3 | dlc.4 | dlc.5 | dlc.6 | dlc.7 | dlc.8 | ... | visibility_second_6 | visibility_second_7 | vis |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| individuals | B | B | B | B | B | B | B | B | B | ... | NaN | NaN | |
| bodyparts | Front | Front | Left | Left | Right | Right | Middle | Middle | Body1 | ... | NaN | NaN | |
| coords | x | y | x | y | x | y | x | y | x | ... | NaN | NaN | |
| labeled-ment1/img000.png | 377.0 | 241.0 | 376.0 | 261.0 | 371.0 | 225.0 | 381.0 | 237.0 | 366.0 | ... | 1.0 | 1.0 | |
| labeled-ment1/img002.png | 179.0 | 252.0 | 140.0 | 289.0 | 226.0 | 279.0 | 181.0 | 287.0 | 267.0 | ... | 1.0 | 1.0 | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| labeled-eo1/img06993.png | 0.0 | 0.0 | 261.0 | 228.0 | 250.0 | 282.0 | 259.0 | 257.0 | 221.0 | ... | 1.0 | 1.0 | |

```python
df.columns
#check the output df: it should list image_id, then visibility
```

```
Index(['scorer', 'dlc', 'dlc.1', 'dlc.2', 'dlc.3', 'dlc.4', 'dlc.5', 'dlc.6',
       'dlc.7', 'dlc.8', 'dlc.9', 'dlc.10', 'dlc.11', 'dlc.12', 'dlc.13',
       'dlc.14', 'dlc.15', 'dlc.16', 'dlc.17', 'dlc.18', 'dlc.19', 'dlc.20',
       'dlc.21', 'dlc.22', 'dlc.23', 'dlc.24', 'dlc.25', 'dlc.26', 'dlc.27',
       'dlc.28', 'dlc.29', 'dlc.30', 'dlc.31', 'dlc.32', 'dlc.33', 'dlc.34',
       'dlc.35', 'dlc.36', 'dlc.37', 'dlc.38', 'dlc.39', 'dlc.40', 'dlc.41',
       'dlc.42', 'dlc.43', 'dlc.44', 'dlc.45', 'dlc.46', 'dlc.47', 'dlc.48',
       'dlc.49', 'dlc.50', 'dlc.51', 'dlc.52', 'dlc.53', 'dlc.54', 'dlc.55',
       'dlc.56', 'dlc.57', 'dlc.58', 'dlc.59', 'visibility_first_1',
       'visibility_first_2', 'visibility_first_3', 'visibility_first_4',
       'visibility_first_5', 'visibility_first_6', 'visibility_first_7',
       'visibility_first_8', 'visibility_first_9', 'visibility_first_10',
       'visibility_first_11', 'visibility_first_12', 'visibility_first_13',
       'visibility_first_14', 'visibility_first_15', 'visibility_second_1',
       'visibility_second_2', 'visibility_second_3', 'visibility_second_4',
       'visibility_second_5', 'visibility_second_6', 'visibility_second_7',
       'visibility_second_8', 'visibility_second_9', 'visibility_second_10',
       'visibility_second_11', 'visibility_second_12', 'visibility_second_13',
       'visibility_second_14', 'visibility_second_15'],
      dtype='object')
```

# Eliminate + Normalize DATA

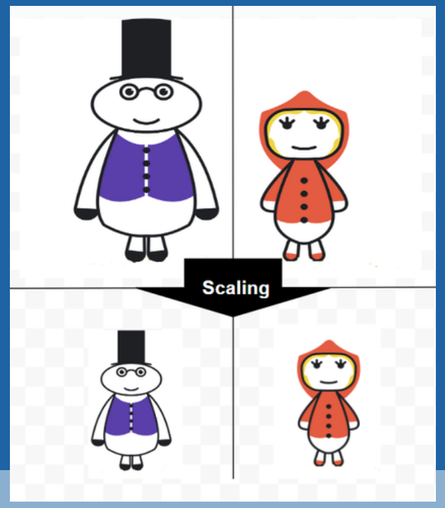
```python
import pandas as pd
from PIL import Image
import os

# Get path of images
img_base_path = '/home/badboy-002/Desktop/content/train/marmoset-dlc-2021-05-07'

# Function to normalize coordinates based on the dimensions of the image
def normalize_coord(x, y, width, height):
    norm_x = x / width # Normalized width
    norm_y = y / height # Normalized height
    return norm_x, norm_y

# Function to process keypoints and visibility
def process_keypoints(image_width, image_height, keypoints_first_class, keypoints_second_class, visibility_first, visibility_second):
    def drop_out_of_bounds(keypoints, visibility):
        updated_keypoints = []
        updated_visibility = []

        for (x, y), v in zip(keypoints, visibility):
            if  v == 0:
                # If visibility is null or zero, keep the point as (0, 0)
                updated_keypoints.append((0, 0))
                updated_visibility.append(0)
            else:
                # Normalize coordinates
                norm_x, norm_y = normalize_coord(x, y, image_width, image_height)
                # Drop points out of bounds
                # Keep keypoints within bounds (0, 1), else set to (0, 0)
                if 0 <= norm_x <= 1 and 0 <= norm_y <= 1:
                    updated_keypoints.append((norm_x, norm_y))
                    updated_visibility.append(v)
                else:
                    updated_keypoints.append((0, 0))
                    updated_visibility.append(0)

        return updated_keypoints, updated_visibility

    # Process first class
    keypoints_norm_first, visibility_first = drop_out_of_bounds(keypoints_first_class, visibility_first)

    # Process second class
    keypoints_norm_second, visibility_second = drop_out_of_bounds(keypoints_second_class, visibility_second)

    return keypoints_norm_first, visibility_first, keypoints_norm_second, visibility_second
```

```python
# Iterate through the DataFrame, skipping the first 3 rows
for index, row in df.iloc[3:].iterrows():
    filename = row[0]  # Get the image filename

    # Load the image to get its dimensions
    image = Image.open(os.path.join(img_base_path, filename))
    image_width, image_height = image.size

    # Extract keypoints and visibility for the first and second class
    keypoints_first_class = row[1:31].values.reshape(-1, 2)     # reshape to 2 column and -1 row (total number of elements) so list 1D of 1-31 => 2* 15 array
    keypoints_second_class = row[31:61].values.reshape(-1, 2)
    visibility_first = row[61:76].values
    visibility_second = row[76:91].values

    # Print progress every 100 rows
    if index%100 == 0:
        print(f"NOW PROCESSING: index{index} - {filename}")

    # Process the keypoints and visibility
    keypoints_norm_first, visibility_first, keypoints_norm_second, visibility_second = process_keypoints(
        image_width, image_height, keypoints_first_class, keypoints_second_class, visibility_first, visibility_second
    )

    # Process first class bounding box for any visible keypoiny
    if any(visibility_first):
        x_coords = [x for (x, y) in keypoints_norm_first if x > 0]
        y_coords = [y for (x, y) in keypoints_norm_first if y > 0]
        x_min_first, y_min_first = min(x_coords), min(y_coords)
        x_max_first, y_max_first = max(x_coords), max(y_coords)
        width_first = x_max_first - x_min_first
        height_first = y_max_first - y_min_first
        x_center_first, y_center_first = x_min_first + width_first / 2, y_min_first + height_first / 2
        width_norm_first, height_norm_first = width_first, height_first
        annotation_first = f"0 {x_center_first} {y_center_first} {width_norm_first} {height_norm_first} " + " ".join([f"{x} {y} {v}" for (x, y), v in zip(keypo
    else:
        annotation_first = None

    # Process second class bounding box for any visible keypoiny
    if any(visibility_second):
        x_coords = [x for (x, y) in keypoints_norm_second if x > 0]
        y_coords = [y for (x, y) in keypoints_norm_second if y > 0]
        x_min_second, y_min_second = min(x_coords), min(y_coords)
        x_max_second, y_max_second = max(x_coords), max(y_coords)
        width_second = x_max_second - x_min_second
        height_second = y_max_second - y_min_second
        x_center_second, y_center_second = x_min_second + width_second / 2, y_min_second + height_second / 2
        width_norm_second, height_norm_second = width_second, height_second
        annotation_second = f"1 {x_center_second} {y_center_second} {width_norm_second} {height_norm_second} " + " ".join([f"{x} {y} {v}" for (x, y), v in zip(
    else:
        annotation_second = None

    # Write annotations to file only if at least one class has non-null annotations
```

# Eliminate + Normalize DATA

## def normalize_coord ()

The normalize_coord() divided the (x,y) with the height and width to set to 0<nor<1 within the plot




```python
import pandas as pd
from PIL import Image
import os

# Get path of images
img_base_path = '/home/badboy-002/Desktop/content/train/marmoset-dlc-2021-05-07'

# Function to normalize coordinates based on the dimensions of the image
def normalize_coord(x, y, width, height):
    norm_x = x / width # Normalized width
    norm_y = y / height # Normalized height
    return norm_x, norm_y
```

```python
# Normalize coordinates
norm_x, norm_y = normalize_coord(x, y, image_width, image_height)
```

# Eliminate + Normalize DATA

## def process_keypoints()

Process for storing normalized x,y , and also empty cell

## def drop_out_of_bounds()

Check and Label out-bound points as (0,0)

```python
# Function to process keypoints and visibility
def process_keypoints(image_width, image_height, keypoints_first_class, keypoints_second_class, visibility_first, visibility_second):
    def drop_out_of_bounds(keypoints, visibility):
        updated_keypoints = []
        updated_visibility = []

        for (x, y), v in zip(keypoints, visibility):
            if  v == 0:
                # If visibility is null or zero, keep the point as (0, 0)
                updated_keypoints.append((0, 0))
                updated_visibility.append(0)
            else:
                # Normalize coordinates
                norm_x, norm_y = normalize_coord(x, y, image_width, image_height)
                # Drop points out of bounds
                # Keep keypoints within bounds (0, 1), else set to (0, 0)
                if 0 <= norm_x <= 1 and 0 <= norm_y <= 1:
                    updated_keypoints.append((norm_x, norm_y))
                    updated_visibility.append(v)
                else:
                    updated_keypoints.append((0, 0))
                    updated_visibility.append(0)

        return updated_keypoints, updated_visibility

    # Process first class
    keypoints_norm_first, visibility_first = drop_out_of_bounds(keypoints_first_class, visibility_first)

    # Process second class
    keypoints_norm_second, visibility_second = drop_out_of_bounds(keypoints_second_class, visibility_second)

    return keypoints_norm_first, visibility_first, keypoints_norm_second, visibility_second
```

# BOUNDING BOX

Use X, Y min/max coord to find size of bounding box, and remark as annotation

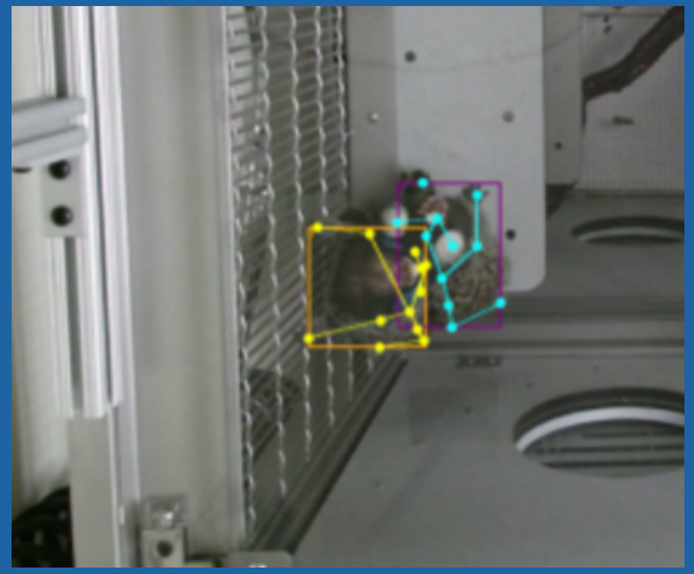
```python
# Process first class bounding box for any visible keypoiny
if any(visibility_first):
    x_coords = [x for (x, y) in keypoints_norm_first if x > 0]
    y_coords = [y for (x, y) in keypoints_norm_first if y > 0]
    x_min_first, y_min_first = min(x_coords), min(y_coords)
    x_max_first, y_max_first = max(x_coords), max(y_coords)
    width_first = x_max_first - x_min_first
    height_first = y_max_first - y_min_first
    x_center_first, y_center_first = x_min_first + width_first / 2, y_min_first + height_first / 2
    width_norm_first, height_norm_first = width_first, height_first
    annotation_first = f"0 {x_center_first} {y_center_first} {width_norm_first} {height_norm_first}
else:
    annotation_first = None
```

```python
# Process second class bounding box for any visible keypoiny
if any(visibility_second):
    x_coords = [x for (x, y) in keypoints_norm_second if x > 0]
    y_coords = [y for (x, y) in keypoints_norm_second if y > 0]
    x_min_second, y_min_second = min(x_coords), min(y_coords)
    x_max_second, y_max_second = max(x_coords), max(y_coords)
    width_second = x_max_second - x_min_second
    height_second = y_max_second - y_min_second
    x_center_second, y_center_second = x_min_second + width_second / 2, y_min_second + height_second / 2
    width_norm_second, height_norm_second = width_second, height_second
    annotation_second = f"1 {x_center_second} {y_center_second} {width_norm_second} {height_norm_second} " + "
else:
    annotation_second = None

# Write annotations to file only if at least one class has non-null annotations
if annotation_first or annotation_second:
    with open(f"{os.path.join(img_base_path, filename).split('.')[0]}.txt", 'w') as f:
        if annotation_first:
            f.write(annotation_first + "\n")
        if annotation_second:
            f.write(annotation_second + "\n")
```

# DRAWING

## def draw_boxes()

Draw a rectangle line from x,y data to a bounding box



```python
# Function to draw bounding boxes on an image
def draw_boxes(image, draw, width, height, annotation_path):
    # Check if the annotation file exists
    if not os.path.exists(annotation_path):
        print(f"Annotation file {annotation_path} does not exist.")
        return

    with open(annotation_path, 'r') as file:
        for line in file:
            parts = line.strip().split()
            # Ensure the line has enough parts to be valid
            if len(parts) < 5:
                continue

            # Parse the bounding box data
            class_id = int(parts[0])
            x_center = float(parts[1]) * width
            y_center = float(parts[2]) * height
            bbox_width = float(parts[3]) * width
            bbox_height = float(parts[4]) * height

            x_min = x_center - (bbox_width / 2)
            y_min = y_center - (bbox_height / 2)
            x_max = x_center + (bbox_width / 2)
            y_max = y_center + (bbox_height / 2)

            color = 'orange' if class_id == 0 else 'purple'

            # Draw the rectangle and class id on the image
            draw.rectangle([x_min, y_min, x_max, y_max], outline=color, width=2)
            draw.text((x_min, y_min), str(class_id), fill=color)
```

# DRAWING

## def draw_keypoints()

Draw x,y data points

```
# Function to draw keypoints on an image
def draw_keypoints(image, draw, keypoints, visibility, color='red', radius=4):
    #print("Keypoints:", keypoints)
    # Draw each keypoint if visibility = 1
    for i in range(len(keypoints)):
        x, y = keypoints[i]
        vis = visibility[i]
        if vis == 1:
            draw.ellipse([x - radius, y - radius, x + radius, y + radius], fill=color, outline=color)
```

## def draw_skeleton()

Label connection line between the dots

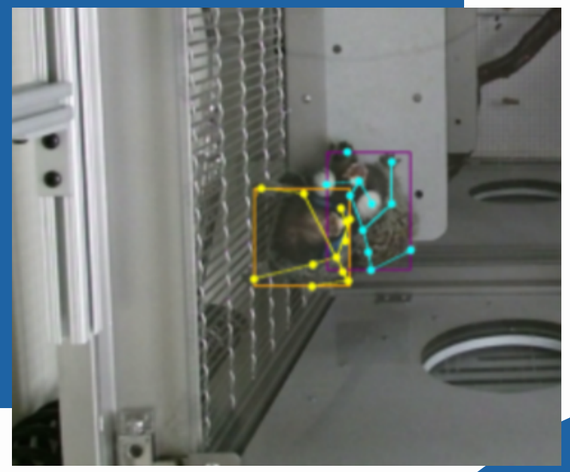
```
# Define skeleton connections
skeleton = [[0,1],[0,2],[0,3],[3,4],[4,5],[5,6],[4,7],[4,9],[7,8],[9,10],[6,11],[6,13],[11,12],[13,14]]

# Function to Draw Skeleton
def draw_skeleton(image, draw, keypoints, visibility, skeleton, color='green', width=2):
    for conn in skeleton:
        start_idx, end_idx = conn
        x_start, y_start = keypoints[start_idx]  # Adjust index for 0-based
        x_end, y_end = keypoints[end_idx]  # Adjust index for 0-based

        # Draw line if both keypoints are visible
        if visibility[start_idx] > 0.0 and visibility[end_idx] > 0.0:
                draw.line([x_start, y_start, x_end, y_end], fill=color, width=width)
```

# DRAWING

## def draw_boxes()

Combine all drawing and data storing variables to display the box and line to show the skeleton of the dataset and the bounding box as a boundary

```
# Main function to process the image
def process_image(image_path, annotation_path):
    image = Image.open(image_path)
    draw = ImageDraw.Draw(image)
    width, height = image.size

    draw_boxes(image, draw, width, height, annotation_path)

    keypoints_first_class, visibility_first_class, keypoints_second_class, visibility_second_class = extract_keypoints_and_visibility(annotation_path, width, height)

    draw_keypoints(image, draw, keypoints_first_class, visibility_first_class, color='yellow')
    draw_keypoints(image, draw, keypoints_second_class, visibility_second_class, color='cyan')

    draw_skeleton(image, draw, keypoints_first_class, visibility_first_class, skeleton, color='yellow')
    draw_skeleton(image, draw, keypoints_second_class, visibility_second_class, skeleton, color='cyan')

    plt.imshow(image)
    plt.axis('off')
    plt.show()

# Run the function
process_image(img_path_selected, annotation_path_selected)
```



# You Only Look Once (YOLO)


SETUP DATA

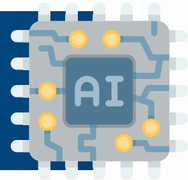
SETUP YAML FILE

INSTALL ULTRALYTICS AND START TRAINING

CHECK THE RESULTS

EXPORT THE BEST MODEL

YOLOMODEL



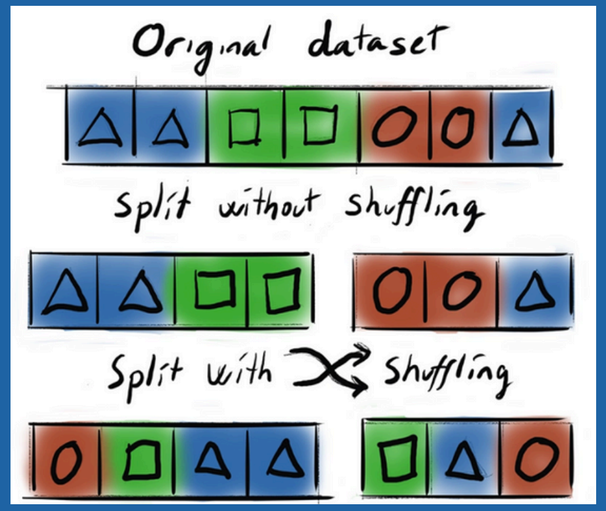
# Split file to train, val

Get the image and labels to the destination folder

```python
import os
import shutil
import random

# Paths for source folders
source_folder1 = '/home/badboy-002/Desktop/content/train/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1'
source_folder2 = '/home/badboy-002/Desktop/content/train/marmoset-dlc-2021-05-07/labeled-data/refinement1'

# Paths for destination folders
train_images_folder = '/home/badboy-002/Desktop/content/project/train/images'
train_labels_folder = '/home/badboy-002/Desktop/content/project/train/labels'
val_images_folder = '/home/badboy-002/Desktop/content/project/val/images'
val_labels_folder = '/home/badboy-002/Desktop/content/project/val/labels'

# Create destination folders if they do not exist
os.makedirs(train_images_folder, exist_ok=True)
os.makedirs(train_labels_folder, exist_ok=True)
os.makedirs(val_images_folder, exist_ok=True)
os.makedirs(val_labels_folder, exist_ok=True)

# Function to list all files in a directory
def list_files(source_folder):
    return [os.path.join(source_folder, f) for f in os.listdir(source_folder) if os.path.isfile(os.path.join(source_fold

# Function to get image and label files from a list of files
def get_image_and_label_files(files):
    images = [f for f in files if f.lower().endswith('.png')]
    labels = [f for f in files if f.lower().endswith('.txt')]
    return images, labels

# Function to copy image and label files to destination folders
def copy_files(images, labels, destination_images_folder, destination_labels_folder):
    for img in images:
        img_filename = os.path.basename(img)
        corresponding_label = img.replace('.png', '.txt')
        # Ensure corresponding label file exists before copying
        if os.path.exists(corresponding_label):
            shutil.copy(img, destination_images_folder)
            shutil.copy(corresponding_label, destination_labels_folder)
```

# def stratify_split()

The `stratify_split()` splits the data into train and validate while maintaining the same proportion of classes across both subsets




```python
# Function to perform stratified split of files into training and validation sets
def stratify_split(source_folder, split_ratio):
    files = list_files(source_folder)
    images, labels = get_image_and_label_files(files)

    # Shuffle the files
    combined = list(zip(images, labels))
    random.shuffle(combined)
    images[:], labels[:] = zip(*combined)

    # Calculate split index
    split_index = int(len(images) * split_ratio)

    # Split into training and validation sets
    train_images = images[:split_index]
    val_images = images[split_index:]
    train_labels = labels[:split_index]
    val_labels = labels[split_index:]

    return train_images, train_labels, val_images, val_labels

# Define split ratio
split_ratio = 0.8  # 80% train, 20% validation

# Perform stratified split for each folder
train_images1, train_labels1, val_images1, val_labels1 = stratify_split(source_folder1, split_ratio)
train_images2, train_labels2, val_images2, val_labels2 = stratify_split(source_folder2, split_ratio)

# Copy files for source_folder1
copy_files(train_images1, train_labels1, train_images_folder, train_labels_folder)
copy_files(val_images1, val_labels1, val_images_folder, val_labels_folder)

# Copy files for source_folder2
copy_files(train_images2, train_labels2, train_images_folder, train_labels_folder)
copy_files(val_images2, val_labels2, val_images_folder, val_labels_folder)
```

**paired image with the correct label and then shuffle**

**split into 80% train 20% validate**

# Create YAML file

A YAML file is a human-readable data serialization format often used for configuration files, data storage, and data exchange between programming languages.

```
# What does YAML mean?
YAML:
    - Y: YAML
    - A: Ain't
    - M: Markup
    - L: Language
```

```python
import yaml
import os

# Define the YAML configuration as a dictionary
yaml_config = {
    'path': '/home/badboy-002/Desktop/content/project',  # Root directory for the dataset
    'train': 'train/images',  # Path to training images (relative to 'path')
    'val': 'val/images',  # Path to validation images (relative to 'path')
    'test': '',  # Path to test images (optional)
    'kpt_shape': [15, 3],  # Number of keypoints, and number of dimensions (x, y, visibility)
    'names': {
        0: 'B',
        1: 'W' # Keypoint class name
    },
    'augmentation': {
        'flip_prob': 0.5,  # Probability of flipping an image horizontally
        'rotation_range': 15,  # Maximum rotation angle in degrees
        'scale_range': [1.0, 1.2],  # Scaling factor range (min, max)
    }
}

# Define the directory and file name
directory = '/home/badboy-002/Desktop/content/project'  # Replace with your directory
file_name = 'yolo_config.yaml'
file_path = os.path.join(directory, file_name)

# Ensure the directory exists
os.makedirs(directory, exist_ok=True)

# Write the YAML configuration to the file
with open(file_path, 'w') as file:
    yaml.dump(yaml_config, file, default_flow_style=False)
```

**augment to increase the dataset**

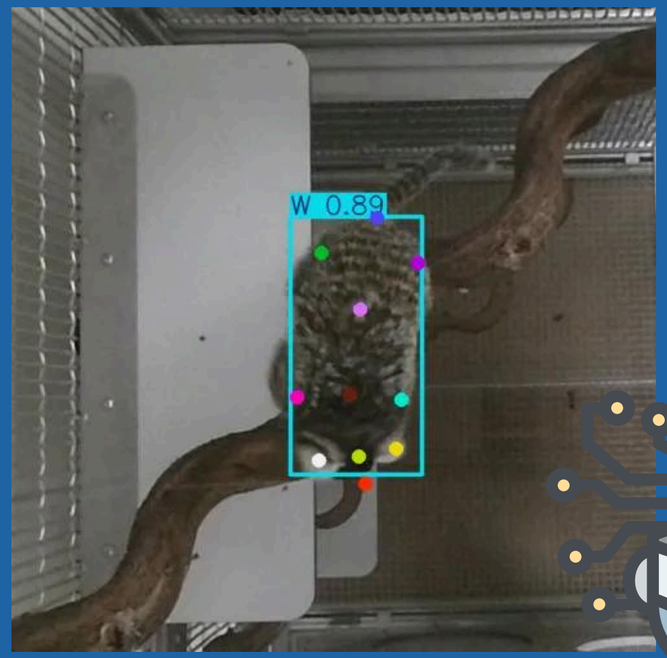
# Resize the image

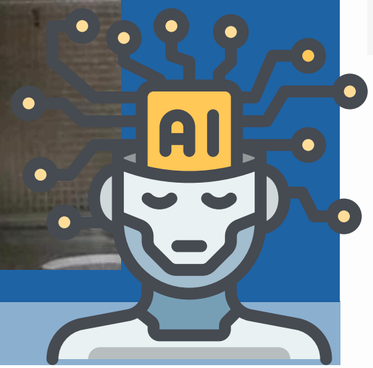
Resize all the images to 512*512 to ensure consistency

```python
from PIL import Image
import os

# Define the list of folders containing images
folders = [
    '/home/badboy-002/Desktop/content/project/train/images',
    '/home/badboy-002/Desktop/content/project/val/images'
]

# Define new size
new_size = (512, 512) # Set new dimensions for the resized images
cnt=1                 # Counter to keep track of the number of processed images
# Process each folder
for folder_path in folders:
    if not os.path.exists(folder_path):
        print(f"Folder {folder_path} does not exist.")
        continue

    # Process each image file in the current folder
    for filename in os.listdir(folder_path):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):  # Check for image file extensions
            file_path = os.path.join(folder_path, filename) # Full path to the image file
            try:
                image = Image.open(file_path)

                # Resize image using default resampling method
                resized_image = image.resize(new_size)  # No resampling filter specified

                # Save resized image, overwriting the original file
                resized_image.save(file_path)

                if cnt%100 == 0 :
                    print(f"Process#{cnt} - Resized and replaced {filename} in {folder_path}.")
                cnt+=1 #counter
            except Exception as e:
                print(f"Error processing {filename} in {folder_path}: {e}")
```

# Train YOLO

Train the pre-trained YOLO model by 120 epochs

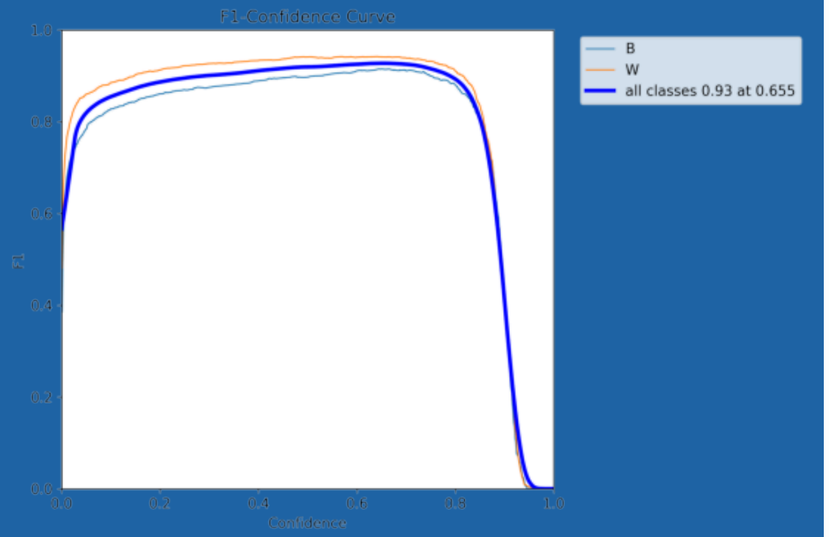



```python
# Install ultralytics package
!pip install ultralytics

# Import the YOLO class from ultralytics
from ultralytics import YOLO    # Import YOLO class for object detection and pose estimation

# Define the path to your YAML configuration file

# Initialize the model with a pre-trained YOLOv8 pose model
model = YOLO("yolov8n-pose.pt")  # You can replace with the specific model variant you need

# Train the model
results = model.train(
    data= '/home/badboy-002/Desktop/content/project/yolo_config.yaml',  # Path to the YAML configuration file
    epochs=120,         # Number of training epochs
    imgsz=512           # Image size
)

# Print results
print("Training completed.")
print(f"Results saved to: {results.save_dir}") # Print the result directory
```

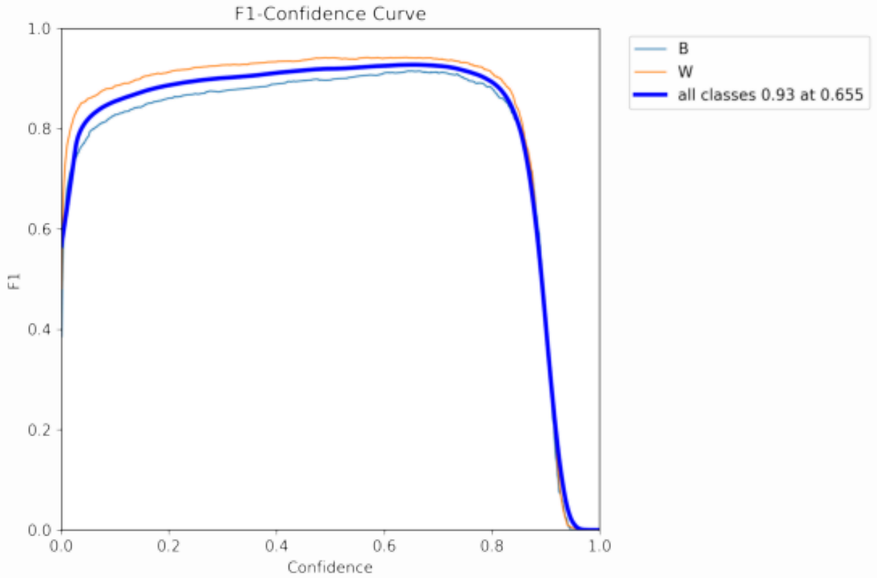
# Plot the results

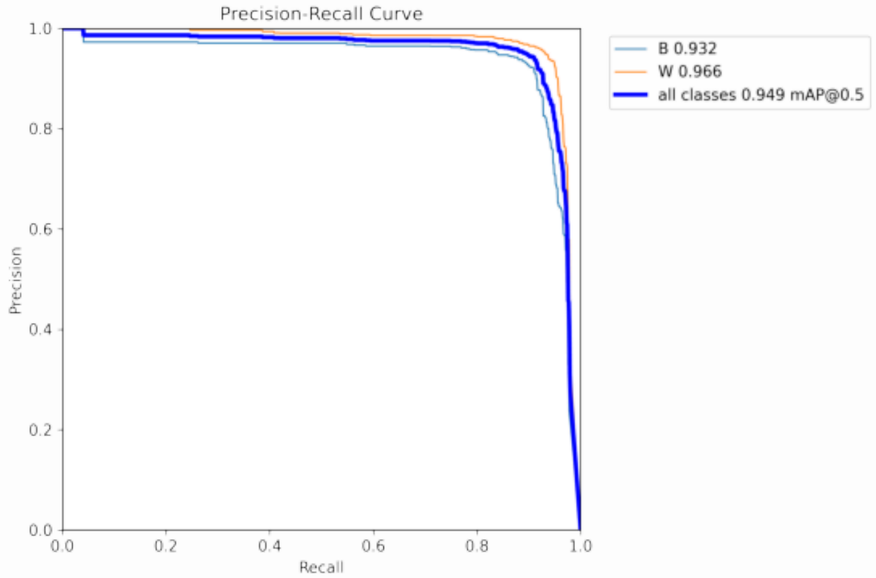
Plot the results using matplotlib

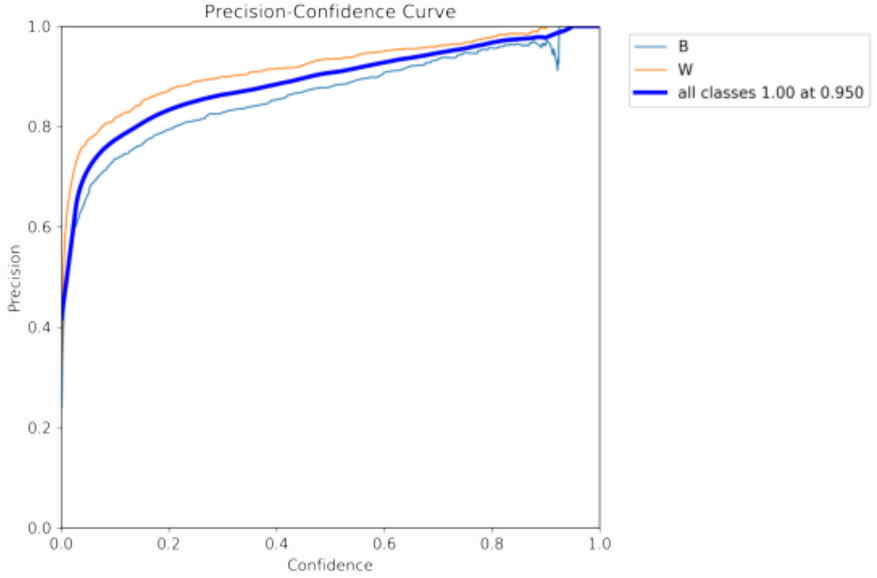


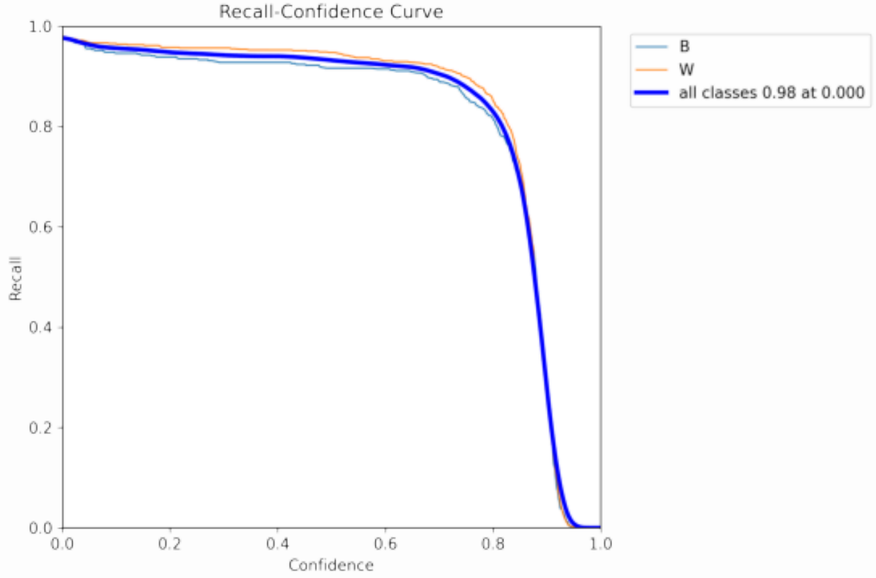

```python
import matplotlib.pyplot as plt
import os
from PIL import Image
import matplotlib.patches as mpatches

# Define the path to the images
image_folder = '/home/badboy-002/Desktop/content/runs/pose/train'  # Replace with your path

# Manually specify the filenames of the images you want to display
selected_images = [
    'BoxF1_curve.png',
    'BoxPR_curve.png',
    'BoxP_curve.png',
    'BoxR_curve.png'
]  # Replace with your filenames

# Set up the figure and axes
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(16,12))

# Iterate over the axes and the selected images
for ax, img_name in zip(axes.flat, selected_images):
    img_path = os.path.join(image_folder, img_name)
    img = Image.open(img_path)
    ax.imshow(img)  # Display the image of the subplot
    ax.axis('off')  # Hide the axes

fig.suptitle('Results of Box Estimation', fontsize=16) # Add a title to the entire figure

plt.tight_layout() # Adjust subplot parameters for a tight fit
plt.show()         # Display the plot
```

# Results of Box Estimation

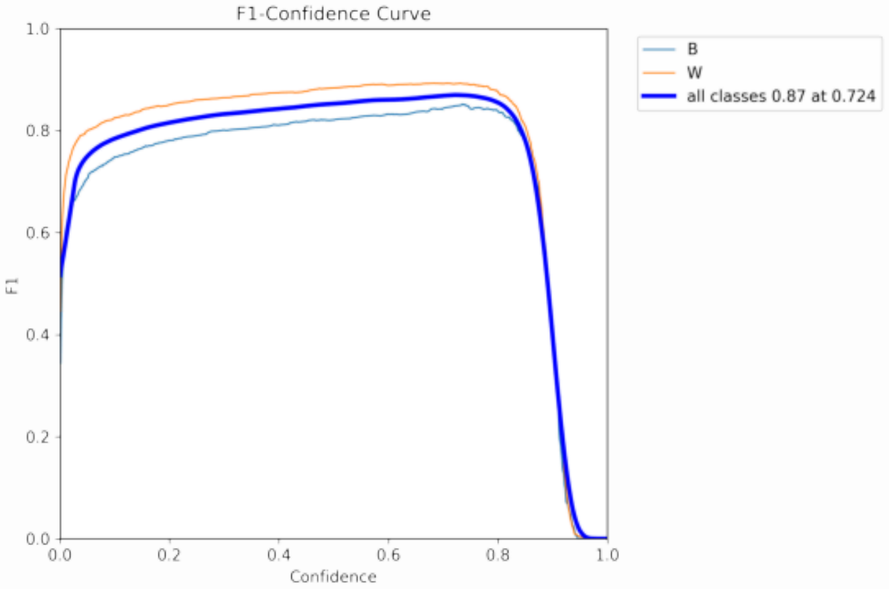


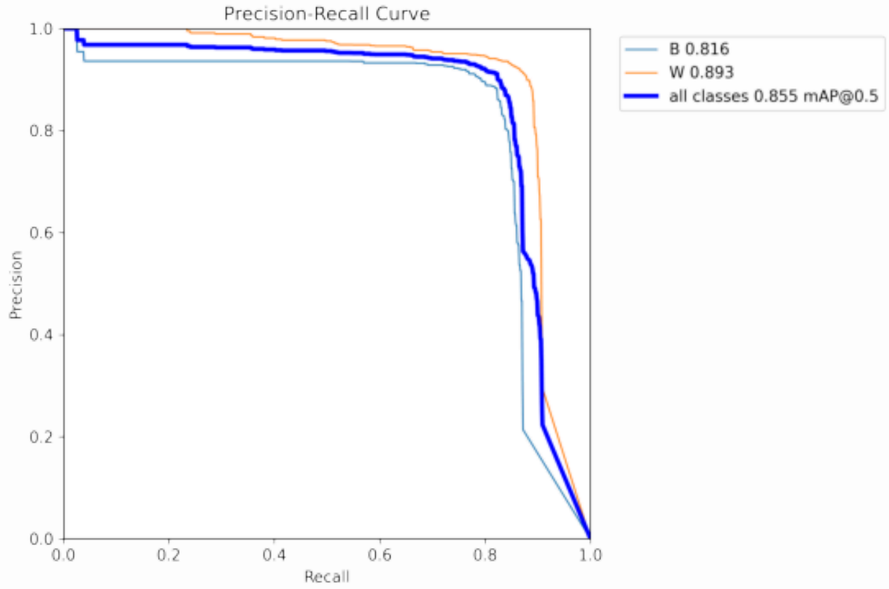



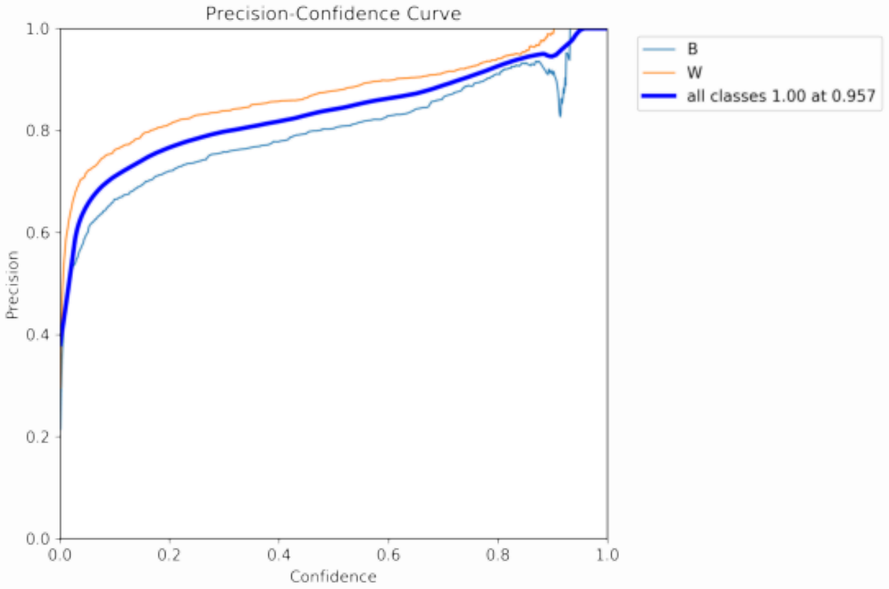



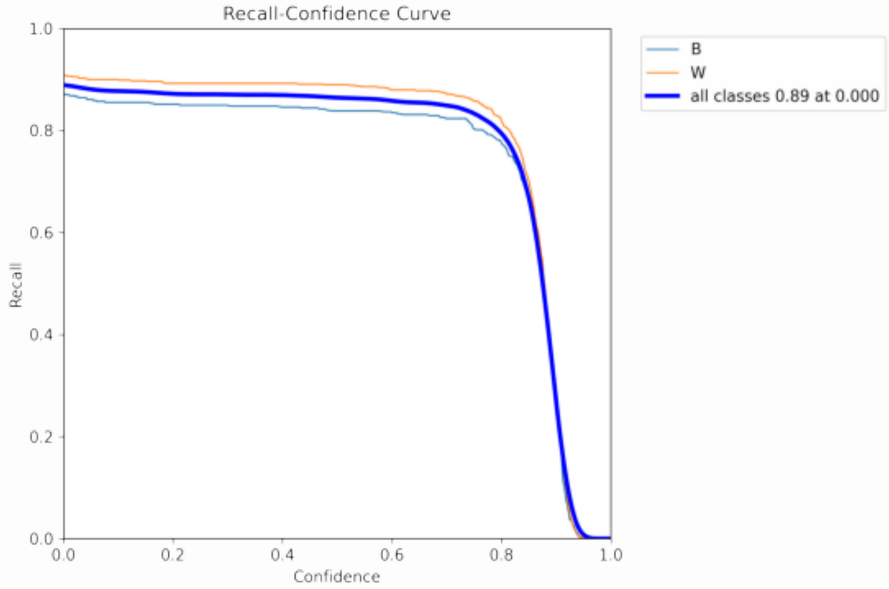



# Results of Pose Estimation



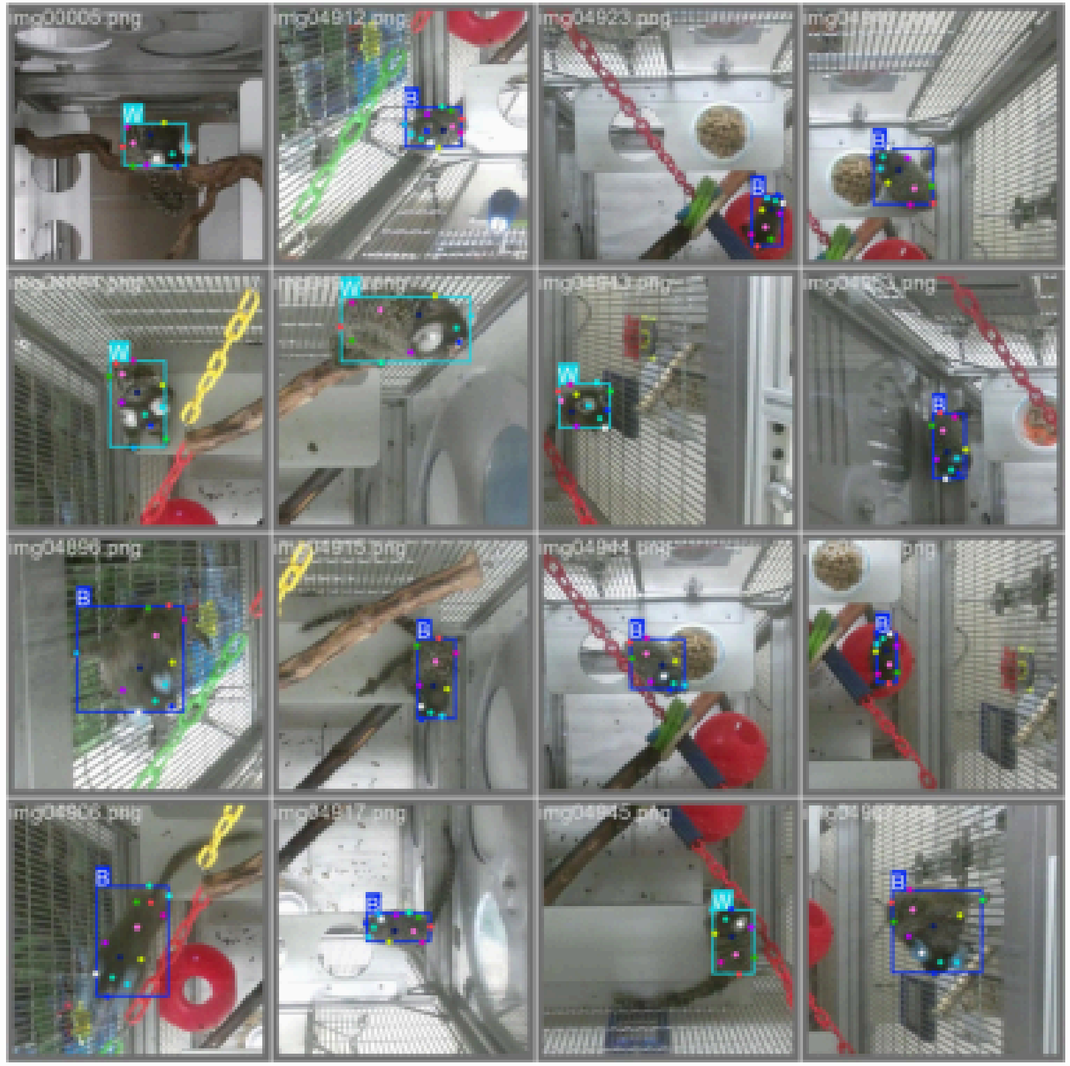



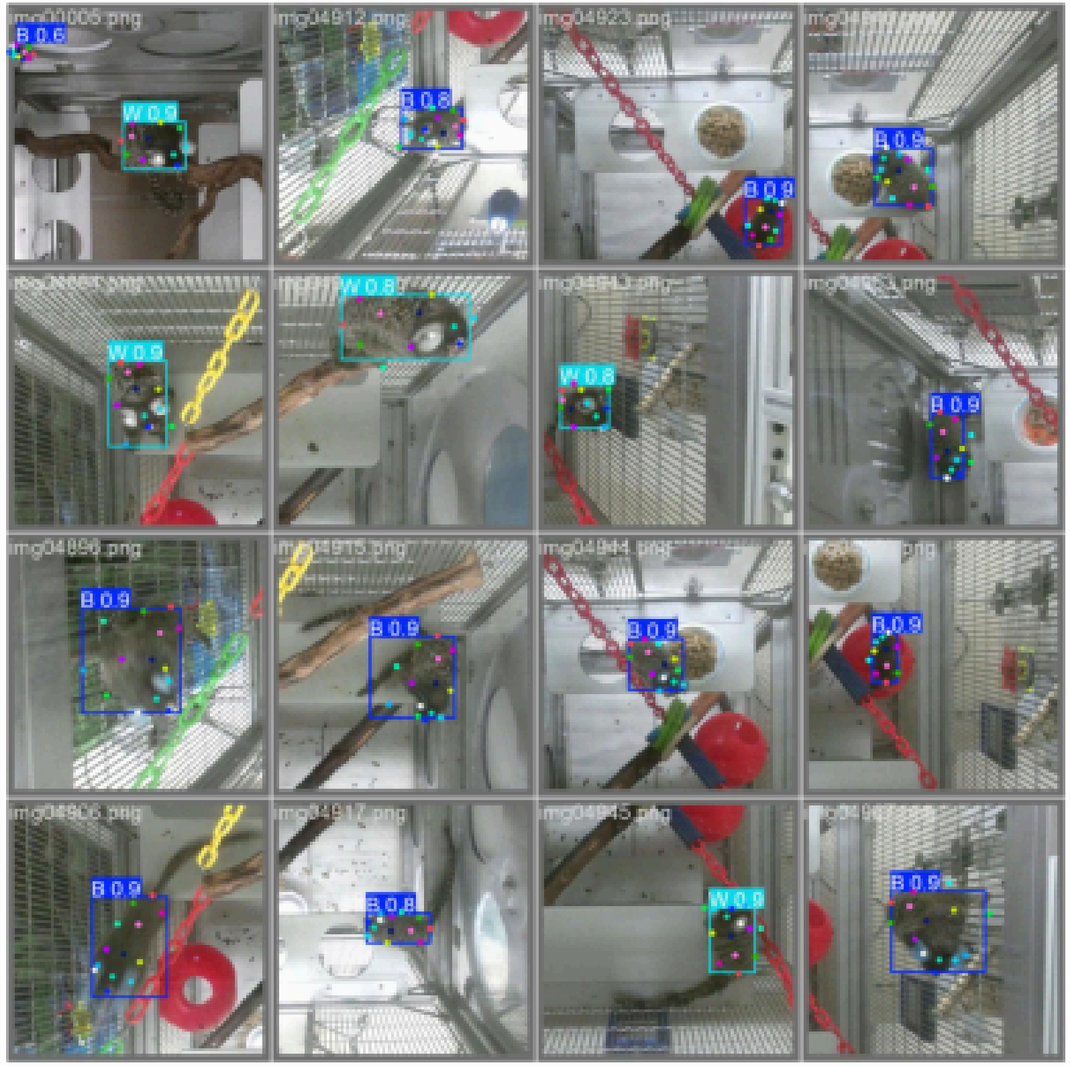



# Comparison of labels and predictions

labels



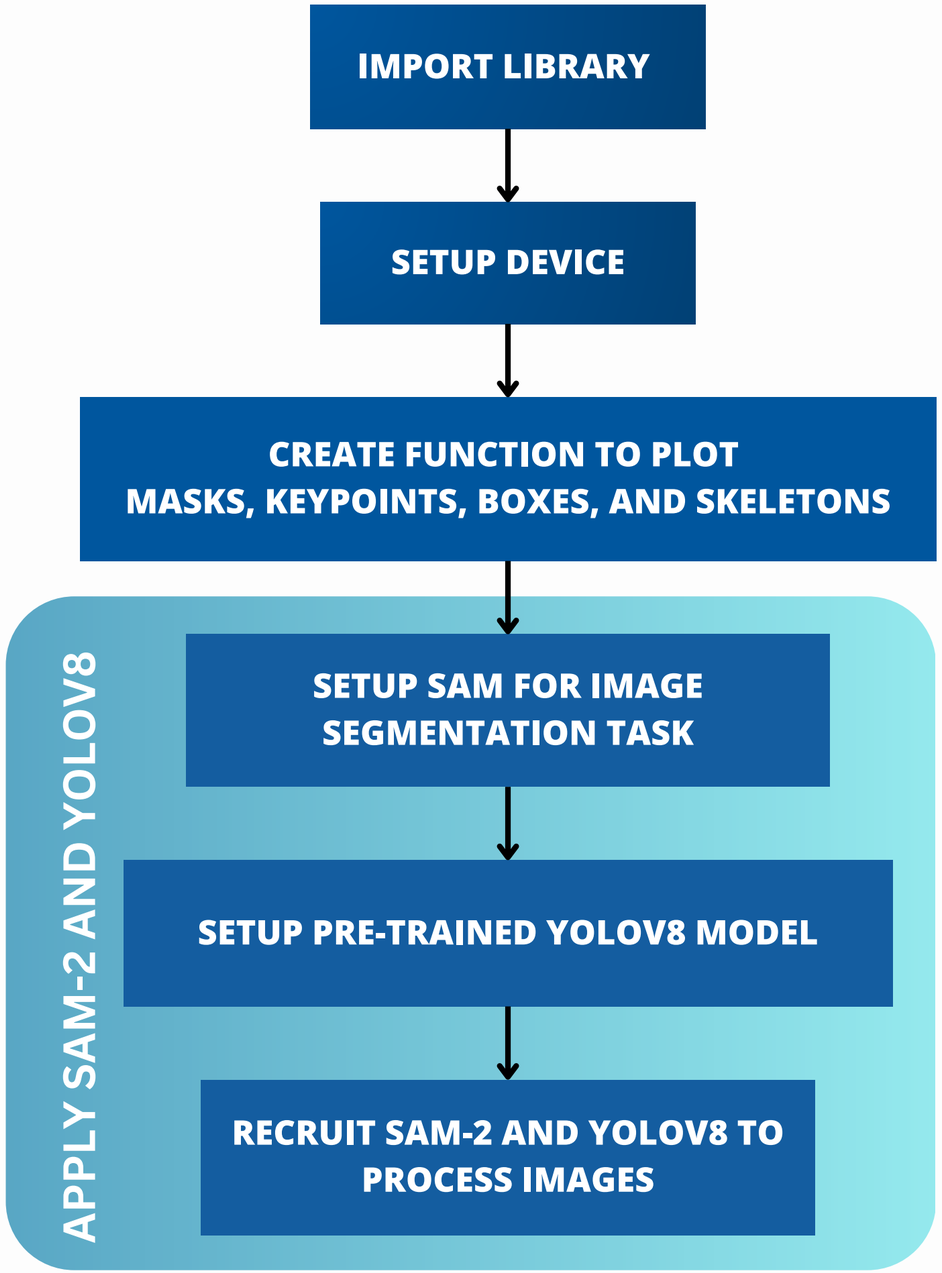
predictions



# Export the best trained YOLO model to use with SAM

`yolomodel.pt`

# Segment Anything Model (SAM)



# Setup

```
# Importing PyTorch and Torchvision libraries
import torch
import torchvision

# Printing the versions of PyTorch and Torchvision
print("PyTorch version:", torch.__version__)
print("Torchvision version:", torchvision.__version__)

# Checking if CUDA (GPU support) is available on the system
print("CUDA is available:", torch.cuda.is_available())

# Importing sys module for executing system commands
import sys

# Installing OpenCV and Matplotlib using pip
!{sys.executable} -m pip install opencv-python matplotlib

# Installing the Segment Anything model from the repository using pip
!{sys.executable} -m pip install
'git+https://github.com/facebookresearch/segment-anything-2.git'

# Creating a directory for storing model checkpoints
!mkdir -p ../checkpoints/

# Downloading the pre-trained SAM-2 model checkpoint into the checkpoints
directory
!wget -P ../checkpoints/
https://dl.fbaipublicfiles.com/segment_anything_2/072824/sam2_hiera_large.pt
```

```
import os
# Setting an environment variable to enable fallback to CPU for unsupported
operations when using Apple MPS (Metal Performance Shaders)
os.environ["PYTORCH_ENABLE_MPS_FALLBACK"] = "1"

# Importing NumPy library for numerical operations
import numpy as np

# Importing PyTorch for tensor operations
import torch

# Importing Matplotlib for plotting images and data
import matplotlib.pyplot as plt

# Importing the Image module from the Python Imaging Library (PIL) for image
manipulation
from PIL import Image

# Importing OpenCV for image processing
import cv2

# Installing Gradio, a library for building web-based interfaces
!pip install gradio

# Importing Gradio for creating user interfaces
```

```
# Select the device for computation (GPU, MPS, or CPU)
if torch.cuda.is_available():
    # If CUDA is available, use the GPU (CUDA)
    device = torch.device("cuda")
elif torch.backends.mps.is_available():
    # If MPS (Metal Performance Shaders) is available (on Apple Silicon), use
MPS
    device = torch.device("mps")
else:
    # Otherwise, fall back to CPU
    device = torch.device("cpu")

# Print the selected device
print(f"using device: {device}")

# If the device is CUDA (NVIDIA GPU)
if device.type == "cuda":
    # Enable automatic casting to bfloat16 precision for faster computation
    torch.autocast("cuda", dtype=torch.bfloat16).__enter__()

    # If the GPU is from the Ampere architecture (major version 8 or higher),
enable TensorFloat-32 (TF32) for better performance
    # This feature allows faster matrix multiplications with slightly reduced
precision
    if torch.cuda.get_device_properties(0).major >= 8:
        torch.backends.cuda.matmul.allow_tf32 = True
        torch.backends.cudnn.allow_tf32 = True

# If the device is MPS (Apple GPU)
elif device.type == "mps":
    # Print a message warning about preliminary support for MPS in PyTorch,
and possible performance or output differences
    print(

        "\nSupport for MPS devices is preliminary. SAM 2 is trained with CUDA
and might "
        "give numerically different outputs and sometimes degraded performance
on MPS. "
        "See e.g. https://github.com/pytorch/pytorch/issues/84936 for a
discussion."
    )
```

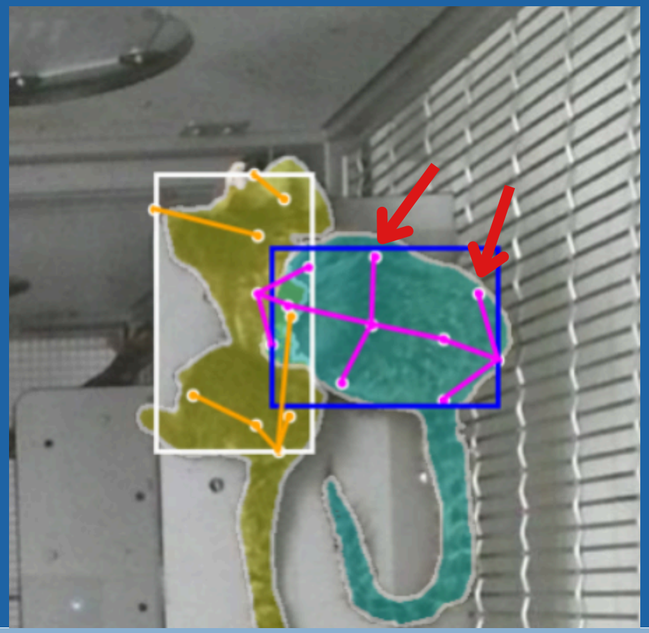
# def show_mask()

The `show_mask()` function used to overlays a segmentation mask

```python
# Function to display a mask on the image
def show_mask(mask, cls, ax, borders=True):
    # Set color based on class (cls). Class 1 uses yellow, otherwise cyan.
    if cls == 1.0:
        color = np.array([255/255, 255/255, 0/255, 0.3])  # Yellow with transparency
    else:
        color = np.array([0/255, 255/255, 255/255, 0.3])  # Cyan with transparency

    # Get height and width from the mask's dimensions
    h, w = mask.shape[-2:]

    # Convert mask to uint8 format for proper image processing
    mask = mask.astype(np.uint8)

    # Reshape the mask and color to create a colored mask image
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)

    # If borders are enabled, find and draw contours around the mask
    if borders:
        import cv2
        # Find contours in the mask
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

        # Smooth contours by approximating polygons
        contours = [cv2.approxPolyDP(contour, epsilon=0.01, closed=True) for contour in contours]

        # Draw the contours on the mask image
        mask_image = cv2.drawContours(mask_image, contours, -1, (1, 1, 1, 0.5), thickness=2)

    # Display the mask on the given axis (ax)
    ax.imshow(mask_image)
```

# def show_points()

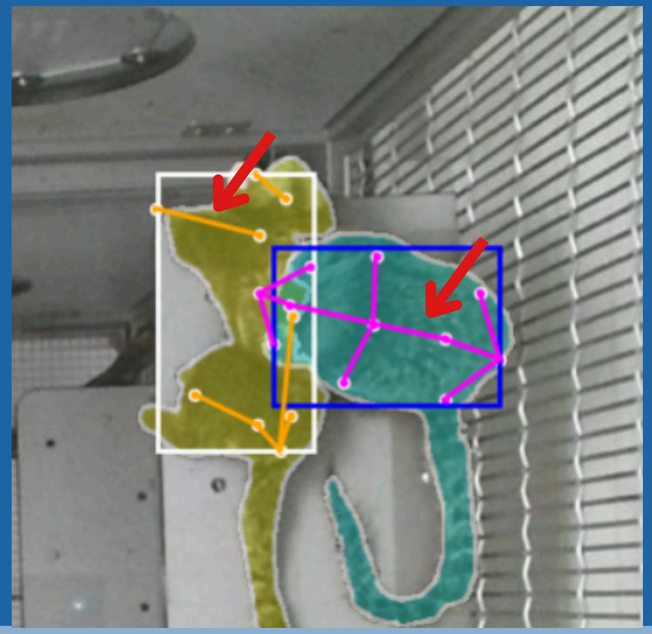
The `show_points()` function used for visualizing keypoints on an image



```python
# Function to display keypoints as points on the image
def show_points(coords, cls, ax, marker_size=20):
    # Convert coordinates to a NumPy array
    coords = np.array(coords)

    # Set color based on class (cls). Orange for class 1, magenta otherwise.
    if cls == 1.0:
        color = 'orange'
    else:
        color = 'magenta'

    # Loop through each point and plot it
    for pt in coords:
        x_coord = pt[0]
        y_coord = pt[1]

        # Only plot points with valid coordinates (greater than 0)
        if x_coord > 0 and y_coord > 0:
            ax.scatter(x_coord, y_coord, color=color, marker='o', s=marker_size, edgecolor='white', linewidth=1)
```

# def show_skeletons()

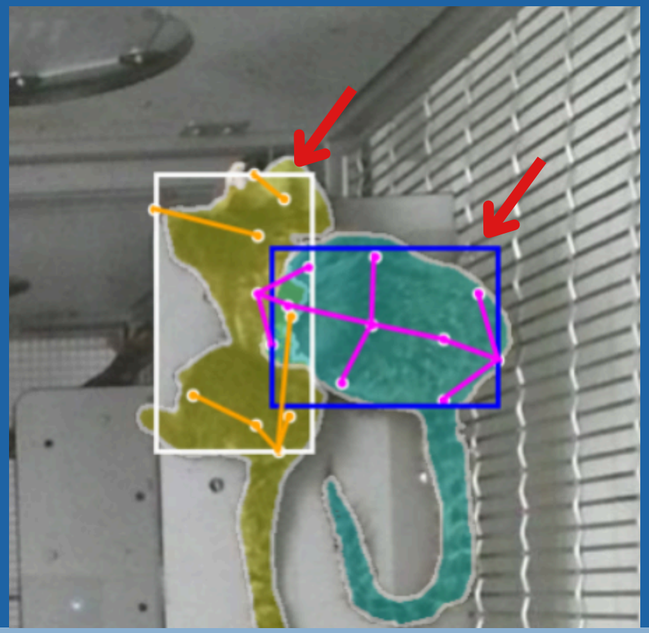
The `show_skeleton()` draws lines between keypoints



```python
# Define the skeleton structure (connections between keypoints)
skeleton = [[0, 1], [0, 2], [0, 3], [3, 4], [4, 5], [5, 6], [4, 7], [4, 9], [7, 8], [9, 10], [6, 11], [6, 13], [11, 12], [13, 14]]

# Function to draw skeleton lines between keypoints
def show_skeletons(keypoints, ax, cls, color='orange', width=2):
    # Iterate through the skeleton connections
    for conn in skeleton:
        start_idx, end_idx = conn
        x_start, y_start = keypoints[start_idx]  # Get coordinates of the start keypoint
        x_end, y_end = keypoints[end_idx]  # Get coordinates of the end keypoint

        # Set color based on class (cls). Orange for class 1, magenta otherwise.
        if cls == 1.0:
            color = 'orange'
        else:
            color = 'magenta'

        # Plot line connecting the two keypoints only if their coordinates are valid
        if x_start > 0 and y_start > 0 and x_end > 0 and y_end > 0:
            ax.plot([x_start, x_end], [y_start, y_end], color=color, linewidth=width)
```

# def show_box()

The `show_box()` display bounding box on image



```python
# Function to display a bounding box on the image
def show_box(box, cls, ax):
    # Get the coordinates of the box (x0, y0, width, height)
    x0, y0 = box[0], box[1]
    w, h = box[2] - box[0], box[3] - box[1]

    # Set color based on class (cls). White for class 1, blue otherwise.
    if cls == 1.0:
        color = 'white'
    else:
        color = 'blue'

    # Draw the box as a rectangle on the image
    ax.add_patch(plt.Rectangle((x0, y0), w, h, edgecolor=color, facecolor=(0, 0, 0, 0), lw=2))
```

# def save_mask()

The `save_mask()` function used for overlay masks onto an image and save the resulted file

```python
# Function to save the mask, keypoints, and boxes on the image, and save the result as an image file
def save_masks(image, masks, scores, classes, points, boxes, borders=True):
    # Create the output directory ('content/outputs') if it doesn't already exist
    output_dir = "content/outputs"
    os.makedirs(output_dir, exist_ok=True)

    # Display the base image
    plt.imshow(image)

    # Loop through masks, scores, classes, points, and boxes, and draw each on the image
    for i, (mask, score, cls, point, box) in enumerate(zip(masks, scores, classes, points, boxes)):
        show_mask(mask, cls, plt.gca(), borders=borders)  # Show mask
        if points is not None:
            show_points(point, cls, plt.gca())  # Show points
            show_skeletons(point, plt.gca(), cls)  # Show skeleton
        if boxes is not None:
            show_box(box, cls, plt.gca())  # Show box

    plt.axis('off')  # Hide axis for a clean image

    # Save the resulting image in the 'content/outputs' directory
    img_file = os.path.join(output_dir, "output_image.png")
    plt.savefig(img_file, bbox_inches='tight', pad_inches=0)  # Save the figure
    plt.close()  # Close the plot to free up memory

    # Open and return the saved image
    img = Image.open(img_file)
    return img
```

**save image exclude extra whit space**

# Create SAM2ImagePredictor

```
# Import required functions and classes from the SAM-2 model package
from sam2.build_sam import build_sam2  # Function to build the SAM-2 model
from sam2.sam2_image_predictor import SAM2ImagePredictor  # Class to handle image predictions with

# Set the path to the pre-trained SAM-2 model checkpoint
sam2_checkpoint = "../checkpoints/sam2_hiera_large.pt"

# Specify the configuration file for the SAM-2 model architecture
model_cfg = "sam2_hiera_l.yaml"

# Build the SAM-2 model using the configuration and the checkpoint file
# 'device' specifies whether to load the model on CPU, CUDA, or MPS
sam2_model = build_sam2(model_cfg, sam2_checkpoint, device=device)

# Create an image predictor object using the built SAM-2 model
predictor = SAM2ImagePredictor(sam2_model)
```

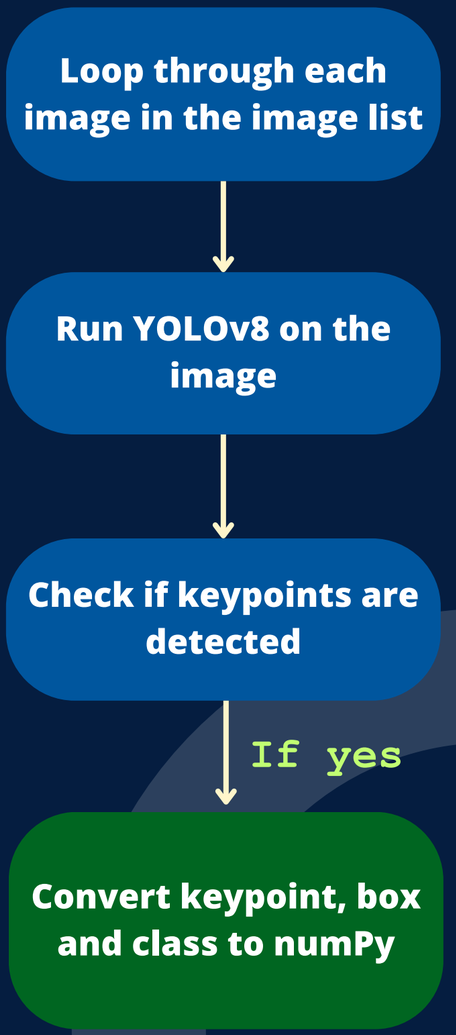
# Import *Yolomodel.pt* from YOLOv8

```
# Install the 'ultralytics' library which contains YOLOv8 functionalities
!pip install ultralytics

# Import the YOLO class from the 'ultralytics' library
from ultralytics import YOLO

# Import Google's Colab 'drive' module to mount Google Drive
from google.colab import drive

# Mount Google Drive to access files stored in your Drive from the Colab environment
drive.mount('/content/drive')

# Load a pre-trained YOLOv8 model from the specified path in Google Drive
# "/content/drive/My Drive/best.pt" refers to the location where your trained YOLOv8 model weights are stored
yolo_model = YOLO("/content/drive/My Drive/best.pt")
```

# Recruit sam-2 and yolov8 to process

## def segment_images()

```python
# Function to segment images using YOLOv8 and SAM-2
def segment_images(image_list):
    # Loop through each image path in the provided list
    for image_path in image_list:
        # Run YOLOv8 model on the image
        results = yolo_model([image_path])
        result = results[0]
        print(f"Processing {image_path}")

        # Check if keypoints are detected in the YOLOv8 results
        if result.keypoints is not None and len(result.keypoints) > 0:
            # Move tensors (keypoints, boxes, classes) from GPU to CPU and convert to NumPy arrays
            keypoints = result.keypoints.xy.cpu().numpy()
            boxes = result.boxes.xyxy.cpu().numpy()  # Bounding boxes
            classes = result.boxes.cls.cpu().numpy()  # Class labels
```



# Recruit sam-2 and yolov8 to process

```
# Initialize lists to store masks and scores
all_masks = []
all_scores = []

# Loop through each detection's keypoints, bounding box, and class
for keypoint, box, cls in zip(keypoints, boxes, classes):
    points = []  # List to store valid keypoints for SAM-2

    # Filter keypoints to only include those inside the bounding box
    for kp in keypoint:
        if kp[0] > 0.0 and kp[1] > 0.0:  # Ensure the keypoint is valid (non-zero)
            if kp[0] < box[2] and kp[0] > box[0] and kp[1] > box[1] and kp[1] < box[3]:
                points.append([kp[0], kp[1]])  # Append valid keypoint coordinates

    # Create labels for SAM-2 (all are '1' as they're valid points)
    labels = [1] * len(points)

    # Open the image using PIL and set it as input for the SAM-2 predictor
    image = Image.open(image_path)
    predictor.set_image(image)

    # Predict masks using SAM-2 with the given points
    masks, scores, logits = predictor.predict(
        point_coords=points,  # Coordinates of the keypoints
        point_labels=labels,  # Labels for the keypoints (1 = foreground)
        multimask_output=True,  # Generate multiple mask outputs
    )
```

## Process keypoint, box and class

**Convert keypoint, box and class to numPy**

**Loop through each detected object**

- Filter valid keypoint
- Create label for SAM2 [1]
- Set image for SAM2 to predict

# Recruit sam-2 and yolov8 to process

```
            # Sort the masks and scores based on scores in descending order
            sorted_ind = np.argsort(scores)[::-1]
            masks = masks[sorted_ind]
            scores = scores[sorted_ind]
            logits = logits[sorted_ind]

            # Select the best mask for further refinement
            mask_input = logits[np.argmax(scores), :, :]

            # Refine the best mask prediction using SAM-2
            masks, scores, logits = predictor.predict(
                point_coords=points,  # Coordinates of the keypoints
                point_labels=labels,  # Labels for the keypoints
                mask_input=mask_input[None, :, :],  # Best mask from previous step
                multimask_output=False,  # Produce a single refined mask
            )

            # Add the refined masks and scores to the list
            for (mask, score) in zip(masks, scores):
                all_masks.append(mask)
                all_scores.append(score)

        # Save and display the segmented image with masks, keypoints, and bounding boxes
        output_img = save_masks(image, all_masks, all_scores, classes, keypoints, boxes)

        # Display the output image using matplotlib
        plt.imshow(output_img)
        plt.axis('off')  # Turn off the axis display for clarity
        plt.show()
    else:
        print(f"No keypoints found for {image_path}")  # If no keypoints are detected, print this message
```

```
# List of image paths to process
images_dir = [
    '/content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img01961.png',
    '/content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img03064.png',
    '/content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img01693.png',
    '/content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img06627.png',
    '/content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img02357.png'
]

# Call the function to process and segment the images
segment_images(images_dir)
```

**Loop through each detected object**

- Filter valid keypoint
- Create label for SAM2 [1]
- Set image for SAM2 to predict

---

  - Predict initial mask
    - Sort mask and select the best one
  - Refine the best mask using SAM2
- Save masks and scores

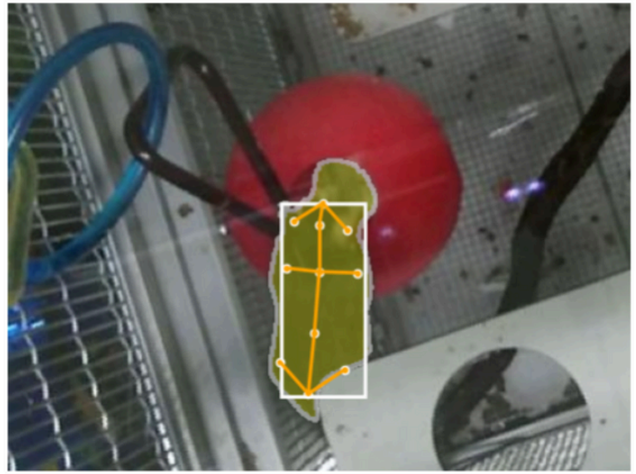
**Save and display the segmented image**

**Check if keypoints are detected**

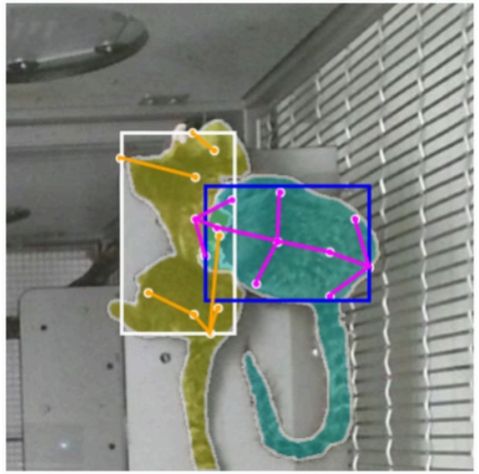
**No: Print "No keypoints found" message**

# Results

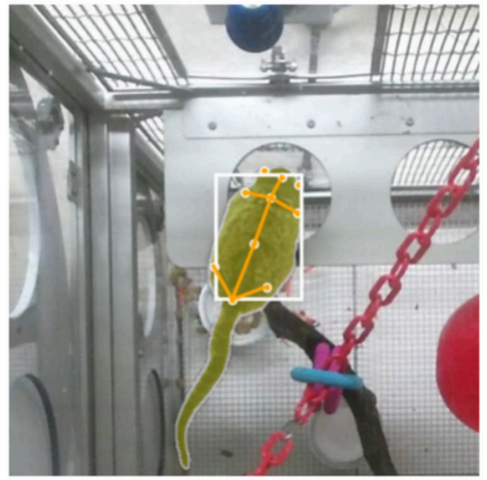
0: 384x512 1 W, 47.9ms
Speed: 2.3ms preprocess, 47.9ms inference, 2.7ms postprocess per image at shape (1, 3, 384, 512)
Processing /content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img01961.png



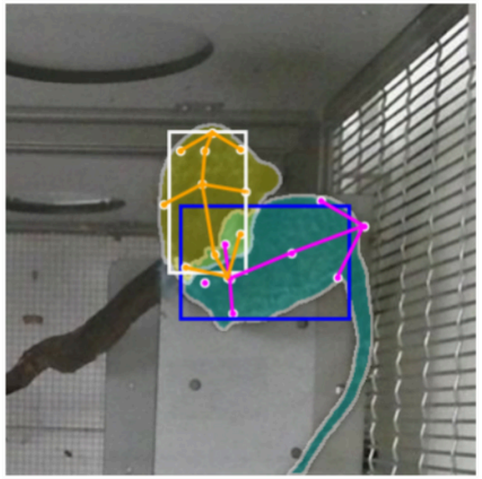
0: 512x512 1 B, 1 W, 30.5ms
Speed: 2.7ms preprocess, 30.5ms inference, 2.4ms postprocess per image at shape (1, 3, 512, 512)
Processing /content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img03064.png



0: 512x512 1 W, 36.7ms
Speed: 2.9ms preprocess, 36.7ms inference, 1.8ms postprocess per image at shape (1, 3, 512, 512)
Processing /content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img01693.png



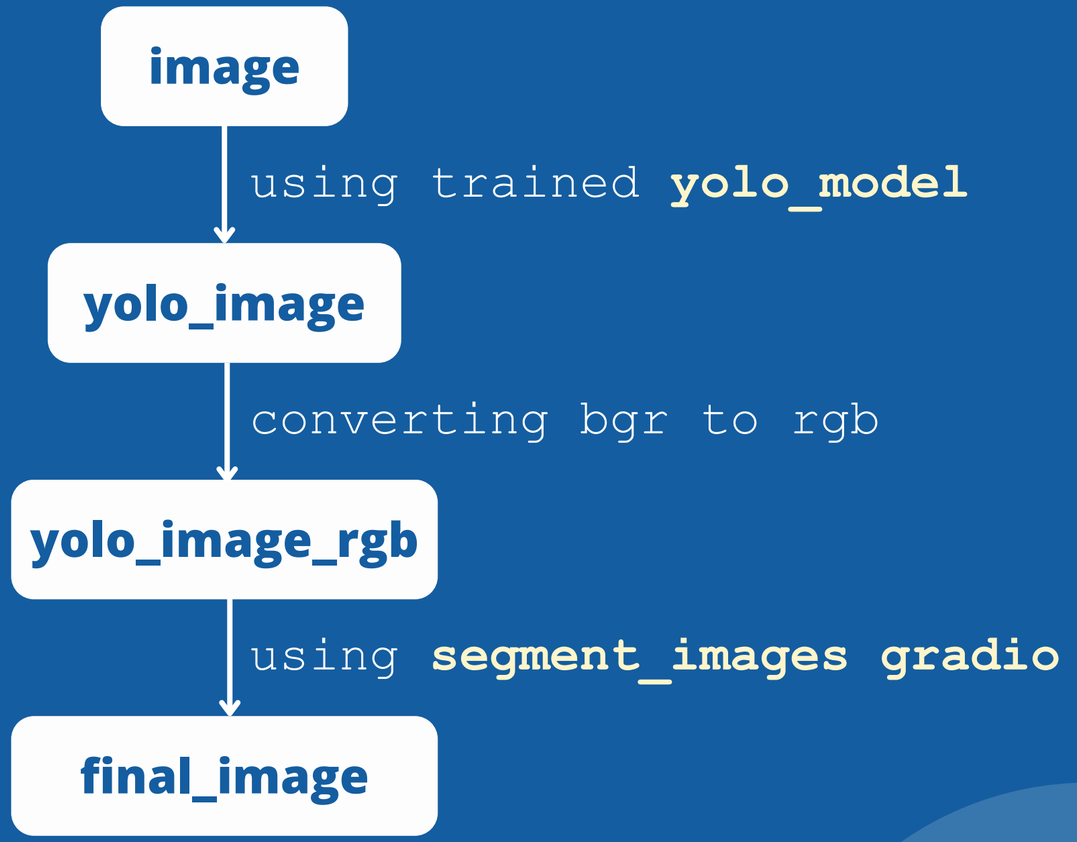
0: 512x512 1 B, 1 W, 34.6ms
Speed: 2.7ms preprocess, 34.6ms inference, 1.9ms postprocess per image at shape (1, 3, 512, 512)
Processing /content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img06627.png



0: 512x512 1 B, 1 W, 36.0ms
Speed: 2.7ms preprocess, 36.0ms inference, 2.2ms postprocess per image at shape (1, 3, 512, 512)
Processing /content/drive/My Drive/marmoset-dlc-2021-05-07/labeled-data/reachingvideo1/img02357.png

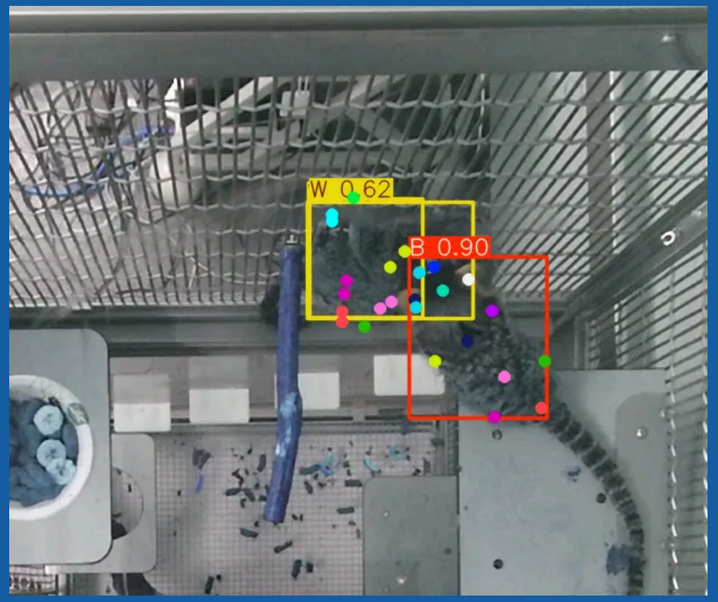
# def predict_image(img)

```python
# Define the function for prediction
def predict_image(img):
    # Run YOLO model predictions
    results = yolo_model.predict(
        source=img,
        show_labels=True,
        show_conf=True,
    )

    if results:
        # Get the YOLO prediction image and convert to RGB
        yolo_image = results[0].plot()
        yolo_image_rgb = cv2.cvtColor(yolo_image, cv2.COLOR_BGR2RGB)

        # Perform segmentation
        final_image = segment_images_gradio(img, results[0], yolo_image_rgb)

        return final_image  # Return as PIL image for Gradio display
    else:
        # Convert original image to PIL if no results
        return img  # Assuming img is already a PIL image
```



image


yolo_image_rgb




yolo_image


final_image

# def segment_images_gradio(image, result, yoloimage)

```python
def segment_images_gradio(image, result, yoloimage):
    # Check if keypoints are present in the results
    if result.keypoints is not None and len(result.keypoints) > 0:
        keypoints = result.keypoints.xy.cpu().numpy()
        boxes = result.boxes.xyxy.cpu().numpy()
        classes = result.boxes.cls.cpu().numpy()
        all_masks = []
        all_scores = []

        # Process each keypoint, box, and class
        for keypoint, box, cls in zip(keypoints, boxes, classes):
            points = [kp[:2] for kp in keypoint if kp[0] > 0.0 and kp[1] > 0.0 and box[0] < kp[0] < box[2] and box[1] < kp[1] < box[3]]
            labels = [1] * len(points)

            predictor.set_image(image)

            # Predict masks
            masks, scores, logits = predictor.predict(
                point_coords=points,
                point_labels=labels,
                multimask_output=True,
            )
```

collect keypoints, bounding boxes, classes

points

using SAM2 model

masks

Because the YOLO model can predict keypoint and already plot them in yolo_image_rgb → unnecessary to collect all points

# def segment_images_gradio(image, result, yoloimage)

```
        # Sort masks and scores by score
        sorted_ind = np.argsort(scores)[::-1]
        masks = masks[sorted_ind]
        scores = scores[sorted_ind]
        logits = logits[sorted_ind]

        # Select the best mask for further refinement
        mask_input = logits[np.argmax(scores), :, :]

        # Refine the mask
        masks, scores, logits = predictor.predict(
            point_coords=points,
            point_labels=labels,
            mask_input=mask_input[None, :, :],
            multimask_output=False,
        )

        # Append results
        for (mask, score) in zip(masks, scores):
            all_masks.append(mask)
            all_scores.append(score)

    # Overlay masks on YOLO image
    output_img = overlay_masks(yoloimage, all_masks, all_scores, classes, keypoints, boxes)
    return output_img
else:
    return yoloimage
```

**yolo_image rgb**

**keypoints from YOLO model**

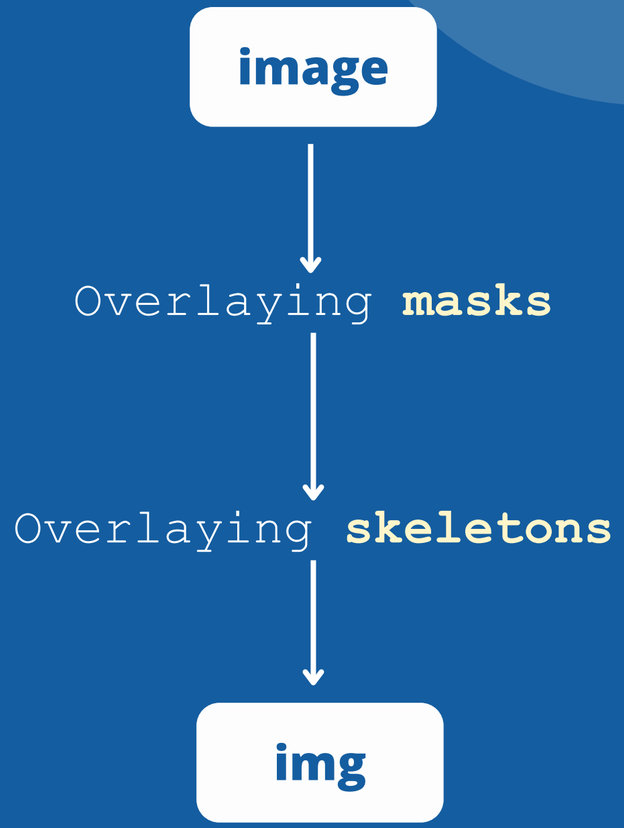
**array of masks from SAM2 model**

using **overlay_mask** function to draw **masks** and **skeletons**

**output_img**

# def overlay_masks(image, masks, scores, classes, points, boxes, borders=True)

```python
def overlay_masks(image, masks, scores, classes, points, boxes, borders=True):
    # Create 'content/outputs' directory if it doesn't exist
    output_dir = "content/outputs"
    os.makedirs(output_dir, exist_ok=True)

    # Plot the image and masks
    fig, ax = plt.subplots()
    ax.imshow(image)

    for i, (mask, score, cls, point, box) in enumerate(zip(masks, scores, classes, points, boxes)):
        show_mask_overlay(mask, cls, ax, borders=borders)
        if points is not None:
            show_skeletons_overlay(point, ax, cls)

    ax.axis('off')

    # Save the image in the 'content/outputs' directory
    img_file = os.path.join(output_dir, "output_image.png")
    plt.savefig(img_file, bbox_inches='tight', pad_inches=0)
    plt.close()  # Close the plot to free up memory

    # Load and return the saved image as PIL Image
    img = Image.open(img_file)
    return img
```



# def show_mask_overlay and show_skeleton_overlay

```python
def show_mask_overlay(mask, cls, ax, borders=True):
    # Define color based on class
    color = np.array([255/255, 255/255, 0/255, 0.3]) if cls > 0.0 else np.array([0/255, 255/255, 255/255, 0.3])

    # Process the mask
    h, w = mask.shape[-2:]
    mask = mask.astype(np.uint8)
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)

    if borders:
        contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        contours = [cv2.approxPolyDP(contour, epsilon=0.01, closed=True) for contour in contours]
        mask_image = cv2.drawContours(mask_image, contours, -1, (1, 1, 1, 0.5), thickness=2)

    ax.imshow(mask_image)
```

```python
def show_skeletons_overlay(keypoints, ax, cls, color='orange', width=2):
    skeleton = [[0,1],[0,2],[0,3],[3,4],[4,5],[5,6],[4,7],[4,9],[7,8],[9,10],[6,11],[6,13],[11,12],[13,14]]
    for conn in skeleton:
        start_idx, end_idx = conn
        x_start, y_start = keypoints[start_idx]
        x_end, y_end = keypoints[end_idx]
        color = 'orange' if cls == 1.0 else 'magenta'
        if x_start > 0 and y_start > 0 and x_end > 0 and y_end > 0:
            ax.plot([x_start, x_end], [y_start, y_end], color=color, linewidth=width)
```

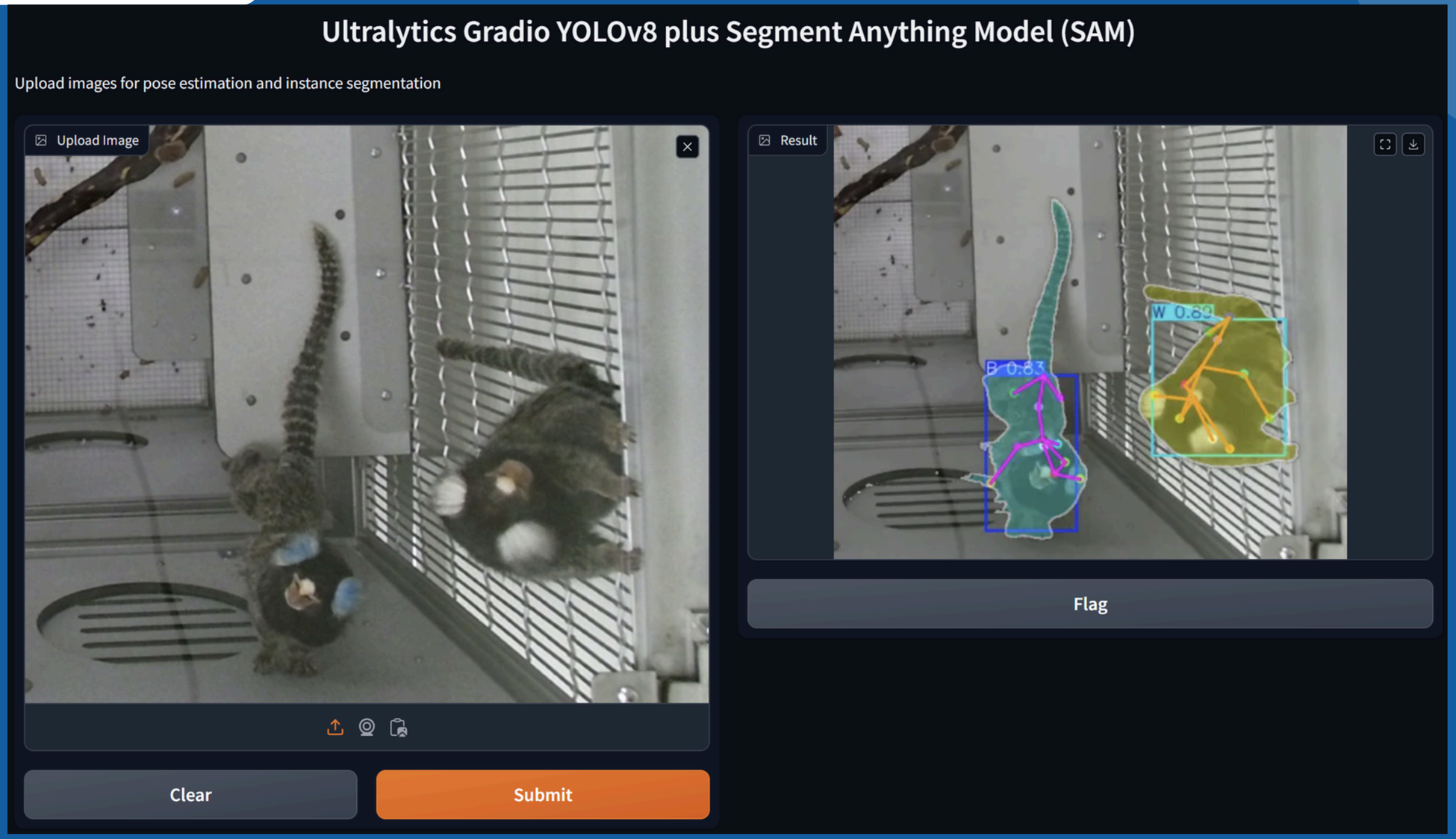
# Launch Gradio

```python
# Create Gradio interface
iface = gr.Interface(
    fn=predict_image,
    inputs=[
        gr.Image(type="pil", label="Upload Image")
    ],
    outputs=gr.Image(type="pil", label="Result"),  # Use gr.Image for PIL image output
    title="Ultralytics Gradio YOLOv8 plus Segment Anything Model (SAM)",
    description="Upload images for pose estimation and instance segmentation",
)

# Launch the Gradio interface
iface.launch()
```

**Using predict_image function to generate output from input**

**LAUNCH**

# Launch Gradio



# Pose Estimation and Instance Segmentation Model

**Tanakrit Busarakul 6402008**
**Boonnisa Watcharapathorn 6402010**
**Tharathon Sopithikul 6402050**
**Natthakit Kiattimongkol 6402092**
**Warat Palpai 6402154**