Real-time fish detection model on edge device

By

Nguyen Huu Minh Quang 17452
Tra Dang Khoa 17637

Line of main tasks

Quang	Khoa
- Set up aquarium environment - Take images, gathering dataset - Train model - Improve model	- Labelling images - Images augmentation - Model deploying

Table of contents

Introduction

1. Object detection model
2. Dataset
3. Model training
4. Deployment on raspberry 3 and OAK-D
5. Problems and solutions
6. Conclusion

Introduction

a. **Background:** Monitoring fish populations is crucial for ecosystem health and sustainable fisheries. Traditional methods are often inefficient, prompting the exploration of automated solutions using computer vision and edge computing. Deploying these systems on devices like the Raspberry Pi 3 and OAK-D enables real-time, in-situ monitoring, reducing reliance on cloud infrastructure.

b. **Problem Statement:** Deploying complex deep learning models for fish detection in real-world aquatic environments faces challenges: computational constraints of edge devices, environmental variability (lighting, turbidity), complex species identification, and the need for real-time processing.

c. **Project Goal and Objectives:** This project aims to develop and deploy a robust fish detection system on Raspberry Pi 5 and OAK-D, capable of accurately identifying up to six fish species. Objectives include: developing a suitable deep learning model, optimizing it for edge deployment (minimizing size and maximizing speed), achieving real-time performance, evaluating robustness to varying underwater conditions, and comparing performance on both platforms.

d. **Scope:** This project focuses on detecting and classifying six specific fish species (cá vằn, cá sặc, bình tích, buồm đỏ, janitor, neon fish) using the Raspberry Pi 5 and OAK-D. A dataset of underwater images/videos will be used for training and evaluation, with performance measured using metrics like precision, recall, mAP, and FPS. The project will test in controlled aquatic environments and does not include custom hardware development or extensive external system integration.

1. Object detection models

Tra Dang Khoa: testing FOMO model using edge impulse website.
Nguyen Huu Minh Quang: try and test the YOLO model.

In evaluating suitable object detection models for deployment on edge devices, we conduct a comparative analysis between YOLO (You Only Look Once) and FOMO (Faster Objects, More Objects). The below are some comparisons that we have done after using the two models

Feature	**YOLO**	**FOMO**
Approach	One-stage (regression)	Highly optimized for efficiency
Speed	Very fast	Extremely fast
Accuracy	Generally high, but can vary by version	Lower than YOLO, trades accuracy for speed
Resource Use	Moderate to high	Very low
Use cases	Real-time applications, general object detection	Edge devices, resource-constrained environments

After testing the two models, we come to realize that while **FOMO** offered advantages in terms of computational efficiency and suitability for extremely resource-constrained environments, new version of **YOLO** such as **YOLOv5/v8/v10/v11** provided a superior balance of speed and detection accuracy. Given the available computational resources on the Raspberry Pi 5 and OAK-D platforms, and the project's requirement for robust species identification, YOLO was selected as the preferred model for implementation. This decision prioritized achieving a higher level of accuracy in fish detection and classification, while still maintaining acceptable real-time performance on the target hardware.

Additionally FOMO' training time is limited if you don't have a premium account on edge impulse. YOLO is easier to access and train while providing a lot of tutorials for us.

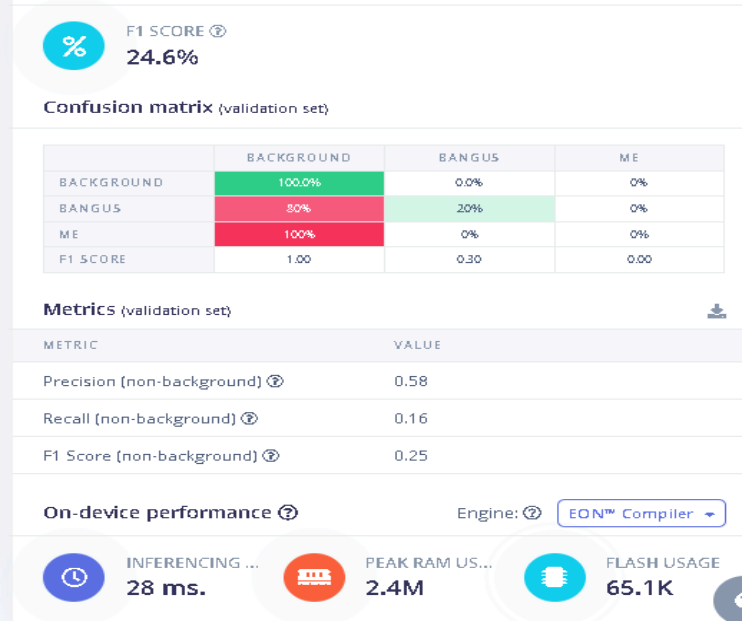


Figure : Accuracy is a problem in FOMO

YOLOv10

YOLOv10 represents a significant leap forward in the realm of object detection, pushing the boundaries of speed and accuracy. This cutting-edge model, developed by researchers at Tsinghua University, introduces a holistic design approach that optimizes various components for both efficiency and performance.

In this project, we will use YOLOv10n, the **'n'** stands for nano, this version is suitable for small edge devices such as raspberry pi that would be used at the end of the project for real-time object detection. Our goal is to detect and classify fishes in real-time through edge device

2. Dataset

(Quang)

At first, our initial effort to acquire a suitable dataset for training the fish detection model involved exploring publicly available online image repositories. While several repositories were identified, a thorough analysis revealed significant limitations for our specific project. These online datasets exhibited considerable heterogeneity, including:

- **Inconsistent Image Quality:** Varying resolutions, lighting conditions, and image compression artifacts were prevalent.
- **Diverse Backgrounds:** Images were captured in diverse environments (aquariums, open water, laboratory settings), introducing unwanted variability.
- **Inconsistent Fish Pose and Appearance:** Fish were depicted in various poses and life stages, potentially hindering the model's ability to learn consistent features.
- **Limited Representation of Target Species:** The available datasets did not always contain a sufficient number of images for all six target species selected for this project.

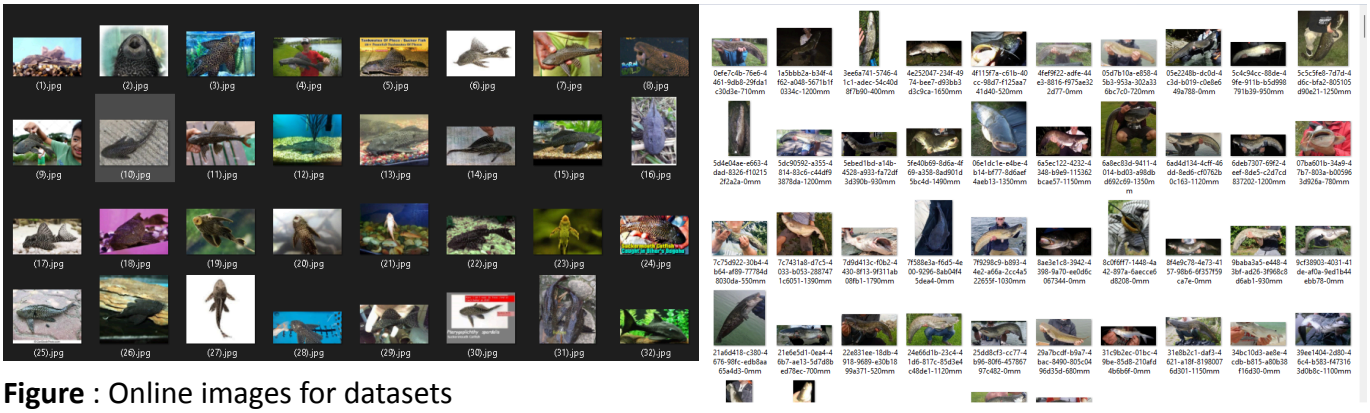


Figure : Online images for datasets

Although up to 875 images have been used, the model still struggling to recognize objects due to unwanted variability and inconsistent fish pose and appearance

Recognizing these limitations, and following guidance from Dr Hien, we selected a custom data acquisition strategy. This approach offered several key advantages:

- **Controlled Environment:** By capturing images in a controlled environment, we were able to standardize lighting, background, and camera settings, minimizing extraneous variability.
- **Targeted Species Acquisition:** We focused specifically on capturing images of the six target species, ensuring sufficient representation for each.
- **Consistent Pose and Appearance:** We could control the fish's pose and capture images at different angles and distances, improving the model's robustness.
- **Data Consistency and Reduced Bias:** Creating our own dataset minimized the risk of introducing biases present in diverse online datasets, leading to a more reliable and generalizable model.

This structured approach to data acquisition, while more resource-intensive, was deemed crucial for developing a robust and accurate fish detection model capable of performing effectively in real-world deployments.

There are six species of fish that we chose for the model to learn the data from, which are cá vằn, cá sặc cảnh, bình tích, cá buồm đỏ, cá lau kiếng (janitor), cá neon

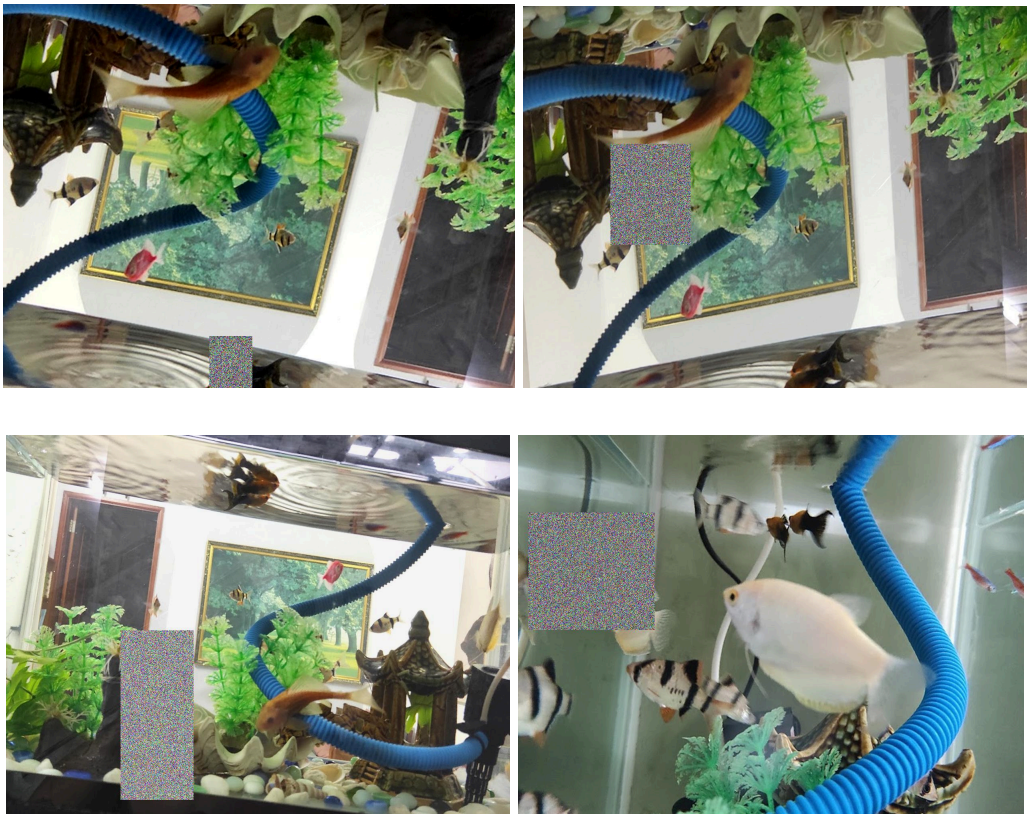
Figure : From Left to right : cá vằn, cá sặc cảnh, bình tích, cá buồm đỏ, cá lau kiếng (janitor), cá neon

<u>Images augmentation (Tra Dang Khoa)</u>

Augmentation only applied to the second version of the dataset and carried out by Tra Dang Khoa.

For the augmentation process, we use augmentor library and google colab for code execution. There are different augmentation techniques such as flip, rotation, etc. These are what we have used:

- Flip (horizontal/vertical) - use this to make model better at detecting different angle
- shear - the same as above
- Rotation - make model more resilient for camera and subjects change
- random crop - better equip model for close up shot
- random erasing - help model to detect the subject that have been obscured by other subject
- random brightness and contrast - help the model less sensitive to light changes
- zoom - make model less sensitive to zooming

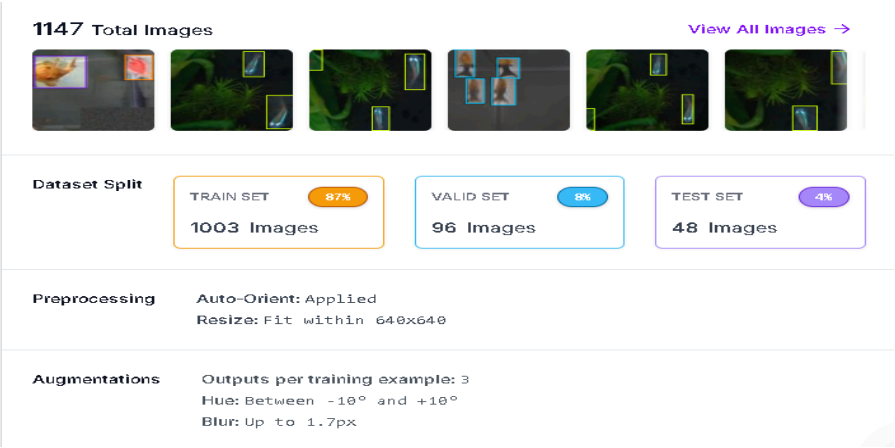


The images augmentation process will be done before labelling as the augmentor library doesn't transform the label of that image. We have to manually label the augmented images. The library also provides unlimited images sampling, meaning that we could artificially increase the size of the dataset. We choose to only sample 500 images because of the reason mentioned earlier.

```
[ ]  pip install Augmentor

Requirement already satisfied: Augmentor in /usr/local/lib/python3.10/dist-packages (0.2.12)
Requirement already satisfied: Pillow>=5.2.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (11.0.0)
Requirement already satisfied: tqdm>=4.9.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (4.66.6)
Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from Augmentor) (1.26.4)

import Augmentor
p = Augmentor.Pipeline("/content/drive/MyDrive/reformed dataset")
p.rotate(probability=0.7, max_left_rotation=10, max_right_rotation=10)
p.zoom(probability=0.5, min_factor=1.1, max_factor=1.5)
p.flip_left_right(probability=0.8)
p.flip_top_bottom(probability=0.3)
p.shear(probability=0.5, max_shear_left=10, max_shear_right=10)
p.random_erasing(probability=0.5, rectangle_area=0.4)
p.random_brightness(probability=0.5, min_factor=0.7, max_factor=1.3)
p.random_contrast(probability=0.5, min_factor=0.7, max_factor=1.3)
p.sample(500)

Initialised with 175 image(s) found.
Output directory set to /content/drive/MyDrive/reformed dataset/output.Processing <PIL.Image.Image image mode=RGB size=4640x3472 at 0x7818067AD660>: 100%
```

Inside augmentor.ipynb

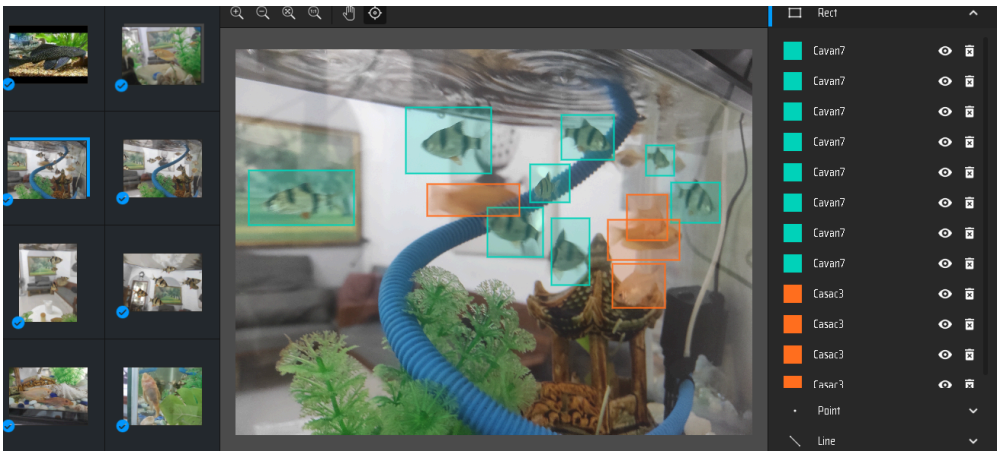
Additionally, we used roboflow website's augmentation (such as exposure and saturation). The benefit of using roboflow augmentation is that they transform the labels as well. However the sampling of roboflow websites is limited to x3.



That's why we used both roboflow and augmentor to help further boost the size of the dataset to more than 1k images.

Labelling images (Khoa + Quang)

After collecting data, for the crucial task of image annotation and labeling, the Makesense.ai platform was utilized. This web-based tool provided a user-friendly interface for efficiently creating bounding boxes around the fish within our acquired image dataset.



For the first version dataset, the labeling process is divided equally for each member to label.

For the second version dataset, all the images was labelled by Tra Dang Khoa

After labelling, Makesense.ai will export a folder which contains images and its label.txt, which will contain the positions of the objects in your images

```
3 0.609375 0.2719665271966527 0.2828125 0.23430962343096234
4 0.565625 0.4351464435146444 0.1375 0.12552301255230125
0 0.2640625 0.6673640167364017 0.1359375 0.22384937238493724
0 0.1375 0.7154811715481172 0.190625 0.25732217573221755
4 0.6015625 0.8138075313807531 0.2125 0.18619246861924685
3 0.8703125 0.8138075313807531 0.259375 0.28661087866108786
```

Objects (fishes) and their position in the image

3. Model Training

(Quang)

Training data is absolutely essential for the development of any effective machine learning model, including our fish detection system. This is because these models learn to recognize patterns and make predictions by being exposed to a large number of labeled examples. In our case, the training data consists of images of fish with corresponding bounding boxes indicating their location and species. By processing this data, the model learns the distinctive visual features of each target species, enabling it to accurately identify and classify fish in new, unseen images. Without a substantial and representative training dataset, the model would be unable to learn these crucial features and would therefore perform poorly in real-world scenarios.

The preparation of our image dataset for model training involved several key steps. First, all annotated images, labeled using Makesense.ai, were organized in a folder named coco128 folder, containing two folders of images and their labels. Yolov10 provides a coco128.yaml file, which will instruct the system what to do with coco128 folder so we do not have to divide the coco128 folder into three category, which are training, validation and testing sets like the older version of yolo anymore

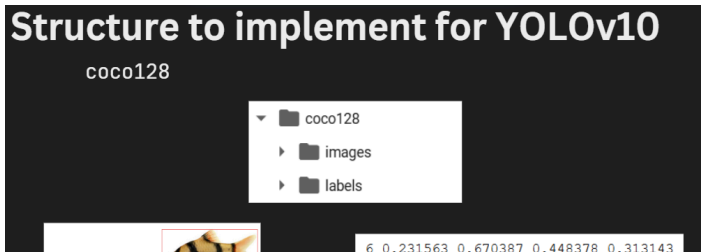
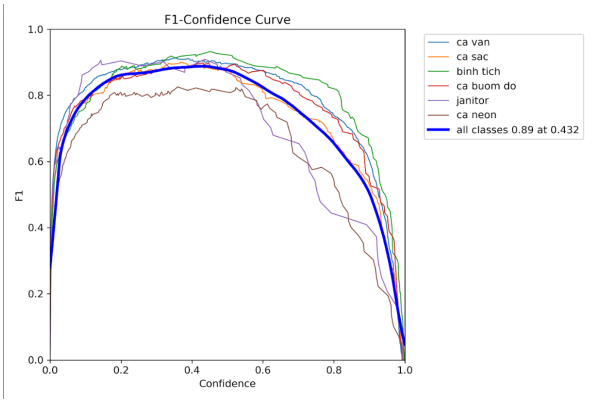
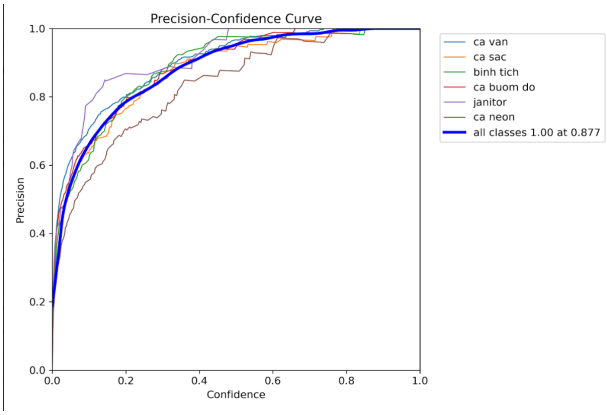


figure coco128 folder **figure coco128.yaml**

Train model result :

```
Ultralytics YOLOv8.1.34 🚀 Python-3.10.12 torch-2.5.1+cu121 CUDA:0 (NVIDIA A100-SXM4-40GB, 40514MiB)
YOLOv10n summary (fused): 285 layers, 2696756 parameters, 0 gradients, 8.2 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 6/6 [00:0
                   all        175       1433      0.922      0.913      0.971       0.83
                ca van        175        517       0.91      0.914      0.972      0.816
                ca sac        175        303      0.924       0.92      0.978      0.855
             binh tich        175        180      0.977      0.929      0.987      0.855
           ca buom do         175        283      0.916      0.908      0.977      0.836
               janitor        175         46      0.956       0.94      0.972      0.868
               ca neon        175        104      0.852      0.865      0.943      0.749
Speed: 0.1ms preprocess, 0.8ms inference, 0.0ms loss, 0.0ms postprocess per image
Results saved to runs/detect/train
```





The model has been replaced and retrained several times for the purpose of being suitable for the common decision and improving the project overall

Training code

```
[ ] # install library yolov10
    !pip install -q git+https://github.com/THU-MIG/yolov10.git
```

```
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
  Building wheel for ultralytics (pyproject.toml) ... done
```

```
[ ] import os
    HOME = os.getcwd()
    print(HOME)
```

```
/content
```

```
[ ] # Dowload yolov10 nano
    !wget -P /content https://github.com/THU-MIG/yolov10/releases/download/v1.1/yolov10n.pt
```

Show hidden output

```
[ ]
    from ultralytics import YOLOv10

    model = YOLOv10('/content/yolov10n.pt')  # Load model directly from local file

    # Define the path to coco128.yaml file
    yaml_path = '/content/drive/MyDrive/coco128.yaml'

    # Start training
    results = model.train(data=yaml_path, epochs=100)
```

Show hidden output

```
from ultralytics import YOLOv10

# Continue with last best weight 'runs/detect/train/weights/best.pt'
model_path = '/content/drive/MyDrive/newest train /weights/best.pt'
model = YOLOv10(model_path)

# Define the path to coco128.yaml file
yaml_path = '/content/drive/MyDrive/coco128.yaml'

# Resume training for additional epochs
results = model.train(data=yaml_path, epochs=80)
```

4. Deployment on edge devices

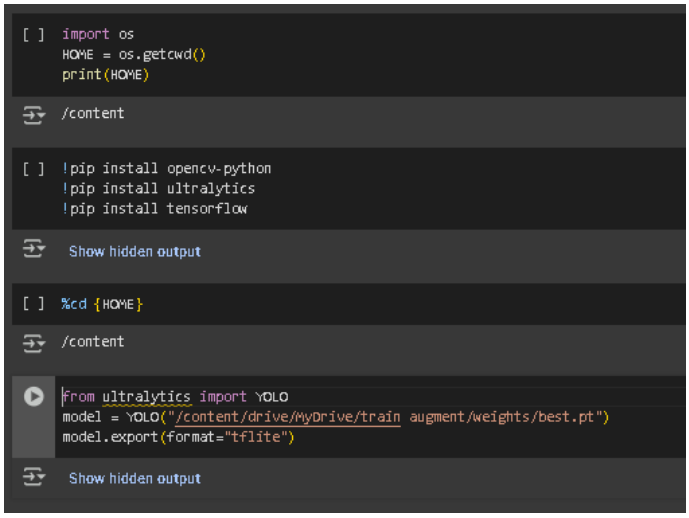
There are two phase deployments. All of them are done by Tra Dang Khoa.

First phase: the first phase is for testing model capability.

Deploying to a laptop. Before the model is deployed. It needs to convert to tflite format because it helps to reduce the computational requirement and increase interfering speed.

The code (both converter and interference) is referenced from a tutorial on youtube: https://www.youtube.com/watch?v=GPag8xLzFCY

The converter is done on google colab



This converter can do int8 quantization on the model as well. Int8 quantization can greatly help to reduce the resource usage and interference speed of the model further. However it made our model detection 10 times worse than before. So we are not using it for now.

Inside the interference code:

```python
import cv2
from ultralytics import YOLO
import pandas as pd
import cvzone


model = YOLO("best_float32.tflite")

def RGB(event, x, y, flags, param):
    if event == cv2.EVENT_MOUSEMOVE:
        point = [x, y]
        print(point)

cv2.namedWindow('RGB')
cv2.setMouseCallback('RGB', RGB)


cap=cv2.VideoCapture(0)
my_file = open("labels.txt", "r")
data = my_file.read()
class_list = data.split("\n")
```

Import necessary libraries like cv2 and cv zone for video processing, Ultralytics for YOLO model. Then create a model path to load the model. Then we create a window for the result. After that use the capture function of cv2 to capture images from the webcam. Load the labels and list classes for the model to detect.

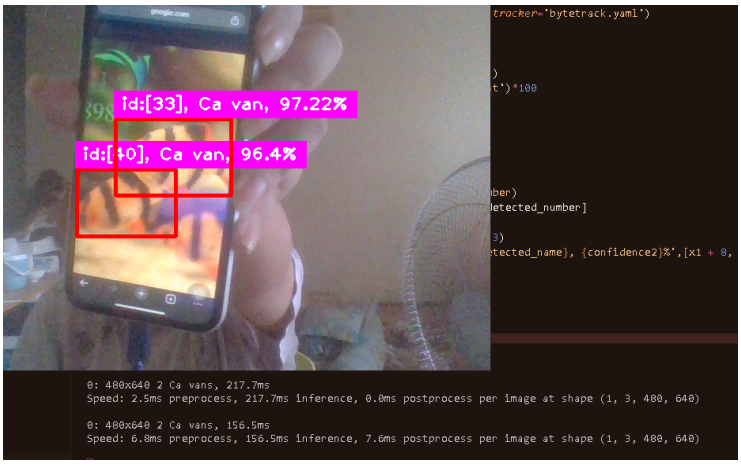
```python
count=0
while True:
    ret,frame = cap.read()
    count += 1
    if count % 3 != 0:
        continue
    if not ret:
       break
    #frame = cv2.resize(frame, (1020, 600))

    results = model(frame, conf=0.3)
    a = results[0].boxes.data
    px = pd.DataFrame(a).astype("float")
    list=[]
    for index, row in px.iterrows():
        x1 = int(row[0])
        y1 = int(row[1])
        x2 = int(row[2])
        y2 = int(row[3])
        d = int(row[5])
        c = class_list[d]
        cvzone.putTextRect(frame,f'{c}',(x1,y1),1,1)
        cv2.rectangle(frame,(x1,y1),(x2,y2),(255,255,0),2)
```

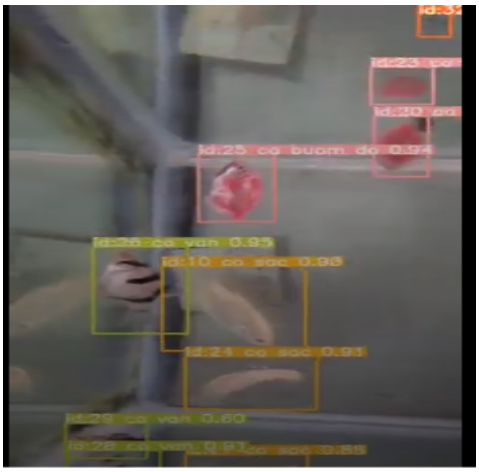
We are going to read the frame that was captured by the webcam (cap.read()). Then throw it to the model to detect at line results. The model will give results such as bounding box coordinate, label, confidence to the cv2. Then cv2 will put these results on the video window.

```
    cv2.imshow("RGB", frame)
    # Break the loop if 'q' is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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

Code for exit the loop.

The result:

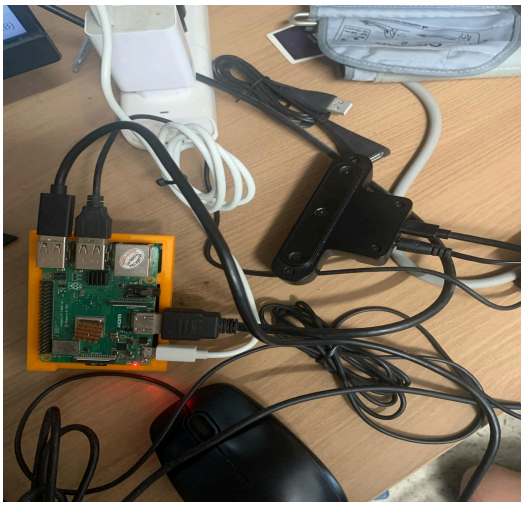




The interference speed is slow but smooth. Detection is still surprisingly good even after converting to tflite format.

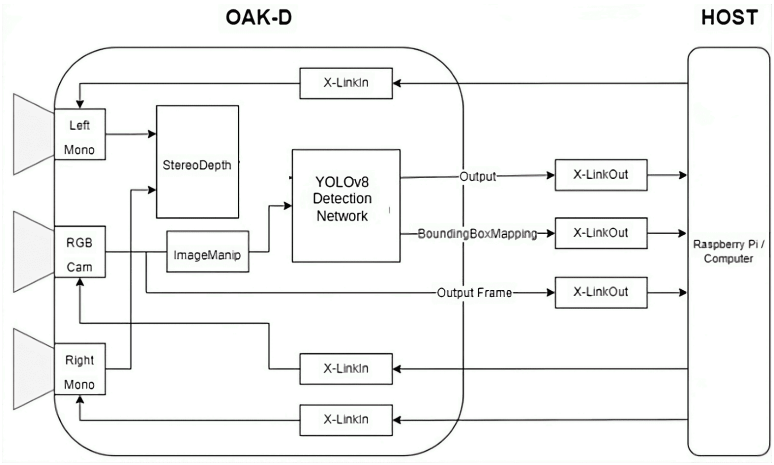
The second and final phase:

In the second phase we deploy on an OAK-D camera using raspberry pi 3 as a host.



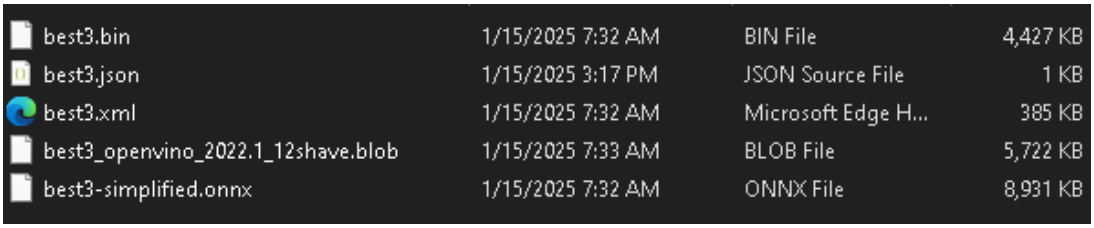
OAK-D camera: this camera is a powerful camera that has notable features such as depth detection, etc. It also has its own powerful processors that can help with the runningML model called Myriad X VPU. So when we run Machine Learning on OKA-D, all the calculations will be mainly done by this camera.

This is general pipeline:



Before the deployment of the model. We converted our model .pt into .blob, .bin, .xml using a depthai tool website (https://tools.luxonis.com/) provided by the camera's manufacturer. On the website you can choose the input image shape and shave. Shave means processing unit of Myriad X VPU inside the camera. The more shaves, the better the performance and interfering speed at a cost of more power usages. When done, the configuration in the .json will be provided. Inside the json file we can change IoU threshold and confidence threshold


```
{
    "model": {
        "xml": "best3.xml",
        "bin": "best3.bin"
    },
    "nn_config": {
        "output_format": "detection",
        "NN_family": "YOLO",
        "input_size": "640x320",
        "NN_specific_metadata": {
            "classes": 6,
            "coordinates": 4,
            "anchors": [],
            "anchor_masks": {},
            "iou_threshold": 0.5,
            "confidence_threshold": 0.3
        }
    },
    "mappings": {
        "labels": [
            "binh tich",
            "ca buom do",
            "ca neon",
            "ca sac",
            "ca van",
            "janitor"
```



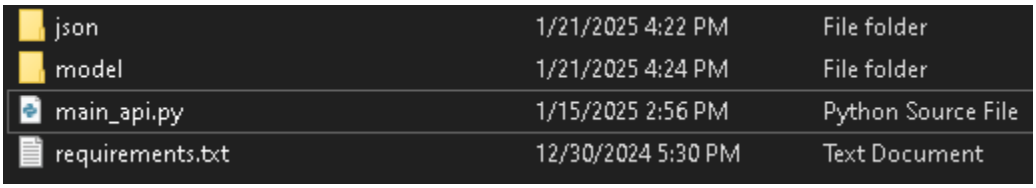
best3.bin	1/15/2025 7:32 AM	BIN File	4,427 KB
best3.json	1/15/2025 3:17 PM	JSON Source File	1 KB
best3.xml	1/15/2025 7:32 AM	Microsoft Edge H...	385 KB
best3_openvino_2022.1_12shave.blob	1/15/2025 7:33 AM	BLOB File	5,722 KB
best3-simplified.onnx	1/15/2025 7:32 AM	ONNX File	8,931 KB

Then for the deployment, we used depthai library and the code provided by this github:

https://github.com/luxonis/depthai-experiments/tree/master/gen2-yolo/device-decoding

And reference: https://learnopencv.com/object-detection-on-edge-device/

The project's folder structure:



json	1/21/2025 4:22 PM	File folder
model	1/21/2025 4:24 PM	File folder
main_api.py	1/15/2025 2:56 PM	Python Source File
requirements.txt	12/30/2024 5:30 PM	Text Document

Json - contain the configuration files

Model - contain all the model file such as .blob, .bin and , .xml

Main_api.py - will deploy the code and run then model the camera

Requirements.txt - list of all libraries necessary to run the main_api.py

Inside the Main_api.py we change the path to our model

```
parser = argparse.ArgumentParser()
parser.add_argument("-m", "--model", help="Provide model name or model path for inference",
                    default='model/best3_openvino_2022.1_12shave.blob', type=str)
parser.add_argument("-c", "--config", help="Provide config path for inference",
                    default='json/best3.json', type=str)
args = parser.parse_args()
```

At first, we chose a 320x320 input image and 8 shaves. Using laptop as host. The result is high FPS but less overall accurate detection. We have to get the camera close to the screen or image to have detection.

Then we used raspberry pi 3 as host. We simply put all our files into USB and transfer to raspberry pi 3. In this one we use 640x640 and 6 shaves and the result is a staggeringly low FPS which is around 3. But the detection is a bit better than 320x320. We don't have to get the camera too close to the subjects. Testing with 8 shaves the overall neuro network FPS reaches to 5 FPS.

5. Problems and Solutions

- **Model's problem**

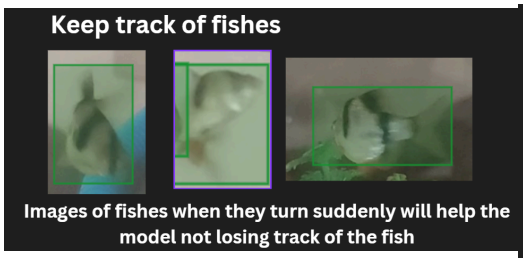
Although the performance of the model in the idealistic environment is good, it still has its problems :

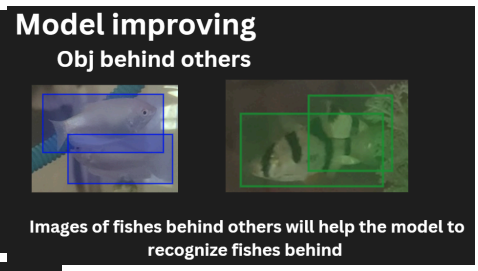
Keep track of fishes, object behind others, different angles and environment

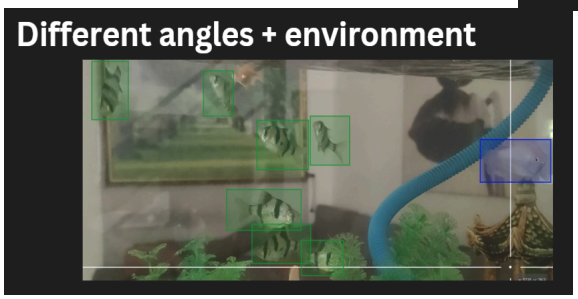
Solution:

We intended to solve the above problems with two kind of approaches

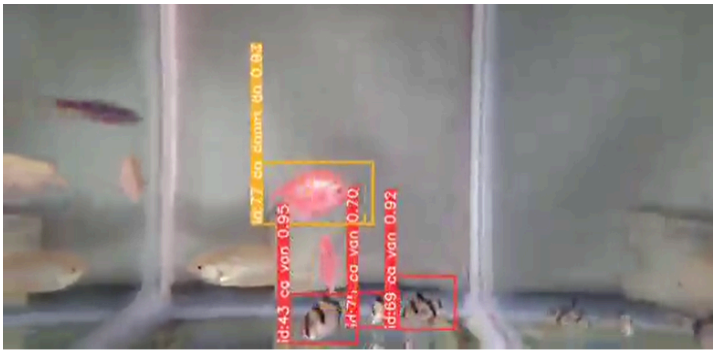
- Taking more images for extra data **(Quang)**
- Augmentation **(Khoa)**

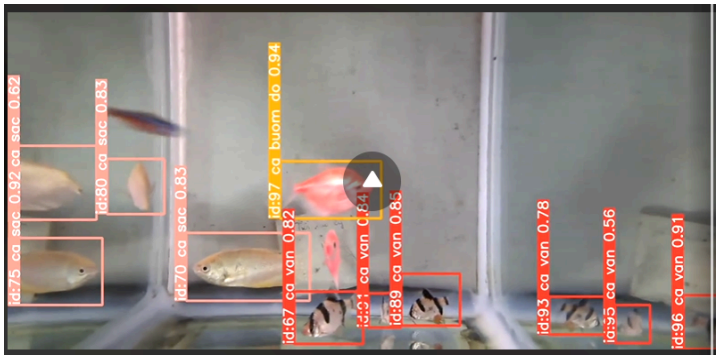






Result





The two are the same frame, but as you can clearly see the difference between the two when the one on the right recognizes more fishes than the left, which proves that the model has been improved significantly.

=> The solutions have proved to be effective, the result has been improved significantly

- **Deployment problems and potential solutions**

We successfully deployed an OAK-D camera using raspberry pi 3 as a host but the FPS is less than what we have desired. Moreover, we still haven't fully learned and used the true power of the camera yet since the time is limited. So in the future, There will be more research and testing to improve the model and fully realize the potential of the OAK-D camera. Furthermore, we could change our focus to a different edge model such as SSD mobilenet.

6. Conclusion

This project successfully developed and implemented a fish detection system using the YOLO object detection model, deployed on edge devices (Raspberry Pi 3 and OAK-D). By prioritizing a custom data acquisition strategy, we addressed the limitations of publicly available datasets and ensured a high-quality, representative training dataset. The use of Makesense.ai streamlined the annotation process, enabling efficient labeling of a large image collection. Careful selection of the YOLOv10n architecture provided a balance between accuracy and computational efficiency, crucial for edge deployment. The trained model demonstrated promising results in identifying and classifying six target fish species, achieving a promising metric on the test set. This work demonstrates the feasibility of deploying real-time fish detection systems on resource-constrained edge devices, offering a valuable tool for ecological monitoring, fisheries management, and aquaculture. Future work will focus on further optimizing the model for improved performance in challenging underwater conditions, exploring deployment in real-world aquatic environments, and investigating the potential for integrating the system with other monitoring tools.