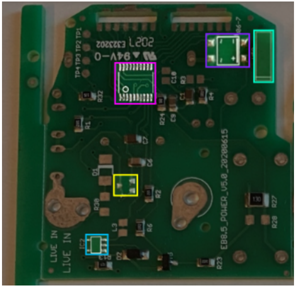
Automated Inspection of PCBs

Smart Bounding Boxes for Live Anomaly Detection

Giulia Ortolani Alessandro Venanzi

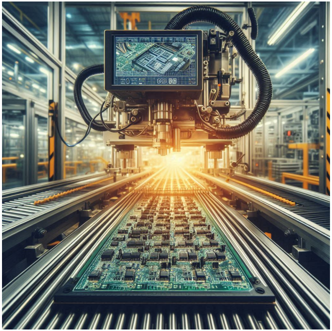
18 June 2024

Project goals



- We wish to leverage state-of-art object detection models such as `YOLOv9` in the context of PCB inspection
- We will analyze the scenario where a fixed camera is posed on top of a conveyor belt used for the production
- Finally, we will try to implement an algorithm that deals with the live stream of images, focusing on adapting the bounding boxes

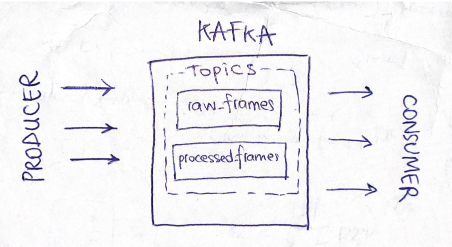
Mathematical framework



- The conveyor belt moves at constant speed $v(t) = v_0$
- Thus the PCB movement form frame to frame is only vertical ($\downarrow$ pixelwise)
- Ideally, its center moves from $(x_c, y_c) \mapsto (x_c, y_c + \Delta y)$
- $y = v(t) \cdot t \implies \Delta y = v_0 \cdot \Delta t$

Kafka topics

We created two Kafka topics to store the raw and the processed frames.



Kafka topics are used to organize and categorize streams of data.

- The topic `raw_frames` will be used to receive the raw frames from our video source;
- The topic `processed_frames` will be used to publish the frames after they have been processed by the model.

Stream processing

- In order to achieve a reasonable scalability and process the stream of frames injected by the camera, we needed a python-friendly environment, possibly built on top of Kafka
- We opted for `Faust`, a Python library that allowed us to build distributed stream processing applications
- This library is designed to be highly scalable and fault-tolerant

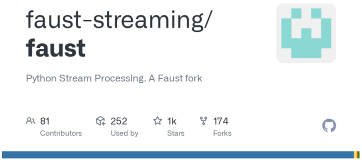
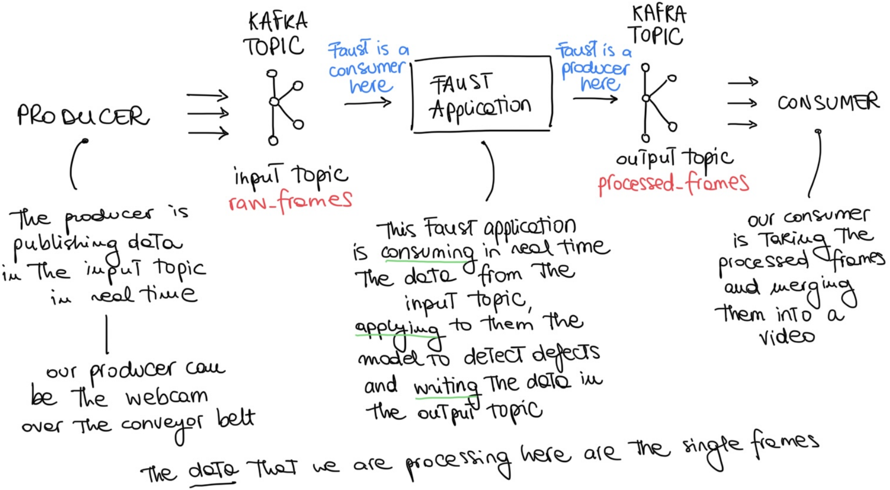


Figure: Faust GitHub repository

How does Faust work?



Faust agent

A **Faust agent** is a high-level abstraction provided by the Faust library for defining and managing stream processing tasks within a Faust application. Here we are defining a Faust agent within the previously instantiated Faust application (`app`):

```
app = faust.App('pcb-defect-detection', broker=KAFKA_BROKER, value_
    serializer='raw')

@app.agent(raw_frames)
```

Faust agent

The agent works as an asynchronous stream processor.

```
# CONSUMER PART
@app.agent(raw_frames)
async def process(frames):
    async for frame in frames:
        frame = np.frombuffer(frame, dtype=np.uint8)
        processed_image = detect_defects(frame)
        # PRODUCER PART
        await processed_frames.send(value=processed_image.tobytes())
```

This agent is responsible for processing raw frames received from a Kafka topic (`raw_frames`), applying the **processing function** and then publishing the processed frames to another Kafka topic (`processed_frames`).

Note. This Faust agent process incoming messages asynchronously (`async` keyword). This means that it can handle multiple messages concurrently without blocking the execution of other tasks.
The asynchronous behavior allows an efficient utilization of system resources and supports high throughput processing.

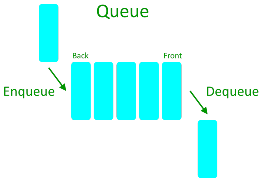
```
async def process_frames(frames):
    async for frame in frames:
```

Algorithm of the processing function

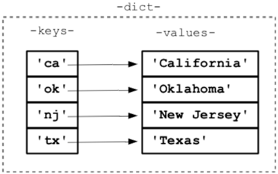
for frame in video **do**
    cur $\leftarrow$ yolo.predict()
    for match in matches(cur, prev) **do**
        if confidence $\geq$ threshold **then**
            $x_c \leftarrow \frac{1}{k}\sum_{i=0}^{k} x_i$
            $y_c \leftarrow \frac{1}{2}(y_c + y_{c,t-1} + \tilde{v}_0 \cdot t)$
            frame.addBB(x_c, y_c,id)
        end if
    end for
    $\tilde{v}_0$.update(cur)
    prev.pop()
    prev.append(cur)
    output frame
end for

Design choices

- To store the k previous predictions, we used a queue: it granted us $O(1)$ for both the insertion and the deletion since our update follows the FIFO philosophy



- For what concerns the matches, we opted for a hash map (i, j) such that looping through all the matches yields $O(n)$, where n is the number of predicted defections at time t



Evaluation

- Given that the model has 99.5% accuracy, we are allowed to use the median predicted x_c as ground truth
- We then studied the distribution of $\varepsilon = med(x_c^{det}) - x_c^{det}$
- From the plot it is clear that with our model, the shift along the x-axis is imperceptible

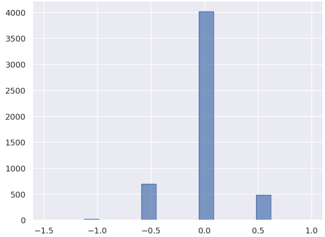


Figure: Histogram of the errors on the x-axis

Evaluation

- For what concerns y_c, we studied the quantity

$$\varepsilon = \lfloor \hat{v}_0 \cdot \Delta t \rfloor + y_{c,t-\Delta t} - y_{c,t}$$

- Where the estimate $\hat{v}_0$ is robust since we computed it after the whole video inference was completed
- In this case, the distribution is centered in 0 and almost every value is zero, meaning that our estimate is coherent with the speed of the conveyor belt

Issues

- When dealing with high FPS video, one may need to "cut" some frames to keep the latency low and deal with the
- When working with a non-ideal camera, we need to perform a geometrical transformation, exploiting a *PerspectiveTransformer*, to keep track of the x-coordinate of the bounding box
- A slight variation from the ideal conditions produce a significant amount of noise and highly affect the tracking

Issues

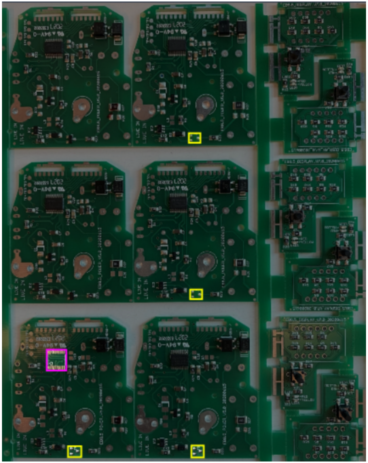


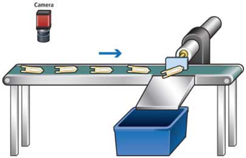
Figure: Low quality image with small defects

Comments

- Providing non-rectangular bounding boxes can enhance the tracking performance: for instance, a simple parallelogram can take into account with video rotation with higher accuracy
- Inference on HD videos is characterized by higher recall
- Fine-tuning the model with more images and in particular small bounding boxes can boost increase the recall on low quality videos
- A tiny amount of noise in the frames does not affect the model performance

Conclusions

- Overall, we think that such a framework can be implemented in production, with some clever choices in the camera setting and a proper Kafka-based stream processing pipeline
- This can yield a significant speed-up in the industrial process of PCB inspection



References

- https://inference.roboflow.com/
- https://universe.roboflow.com/uni-4sdfm/pcb-defects