Real-Time, Multithreaded Anomaly Detection System with Autonomous Synchronization & Self-Correction

Brian Przezdziecki

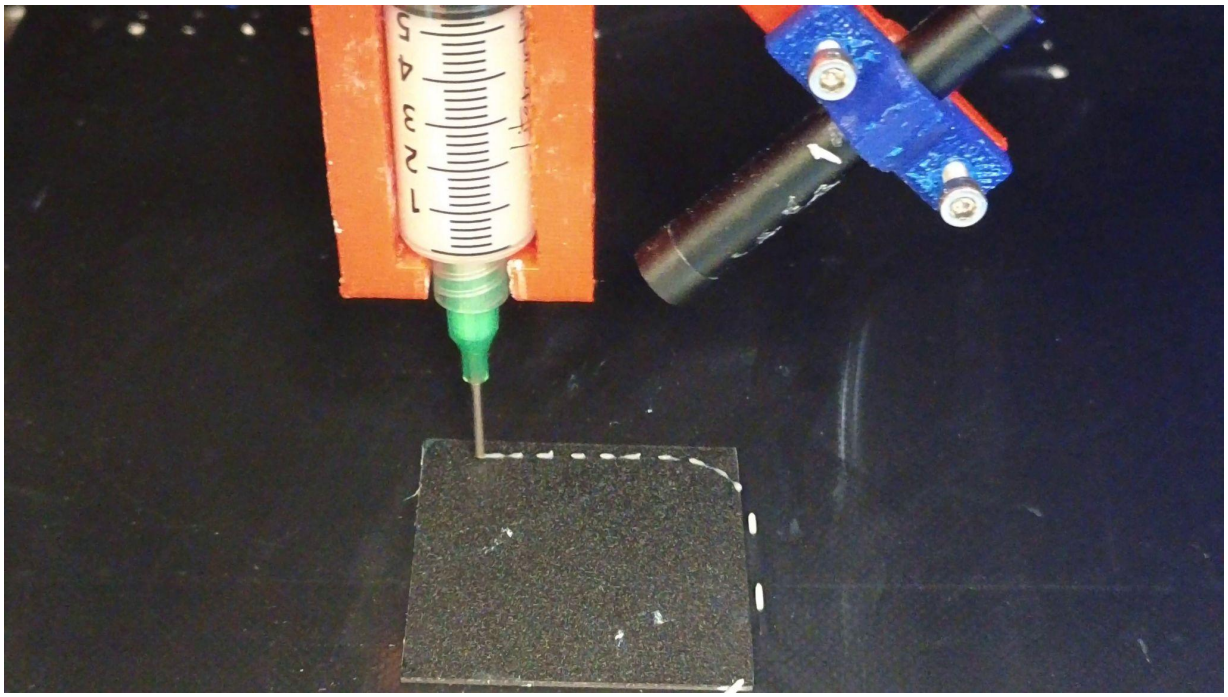
University of Buffalo (SUNY)

Abstract

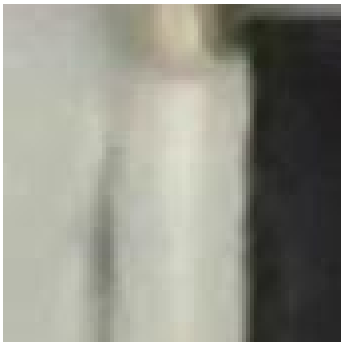
Real-time anomaly detection is crucial during the printing process, especially when looking for issues like over or under extrusion. We employ a MobileNet model to classify these anomalies. Our system has two main components: **Tip Tracking** and **Anomaly Detection**. In the Tip Tracking phase, we estimate the locations of the tips using gcode data. Then, we use the YOLO model to align these predictions with live video footage, ensuring synchronization and correcting any prediction inaccuracies. Given the need for real-time performance on a Raspberry Pi, our system is optimized for computational efficiency. It's designed to operate in real-time, running alongside video feeds, and strikes a balance between accuracy, autonomy, robustness and speed.

Problem

We need to spot errors in 3D printing as they happen. The main mistakes are tied to how much material comes out - too much or too little (Under/Over Extrusion).



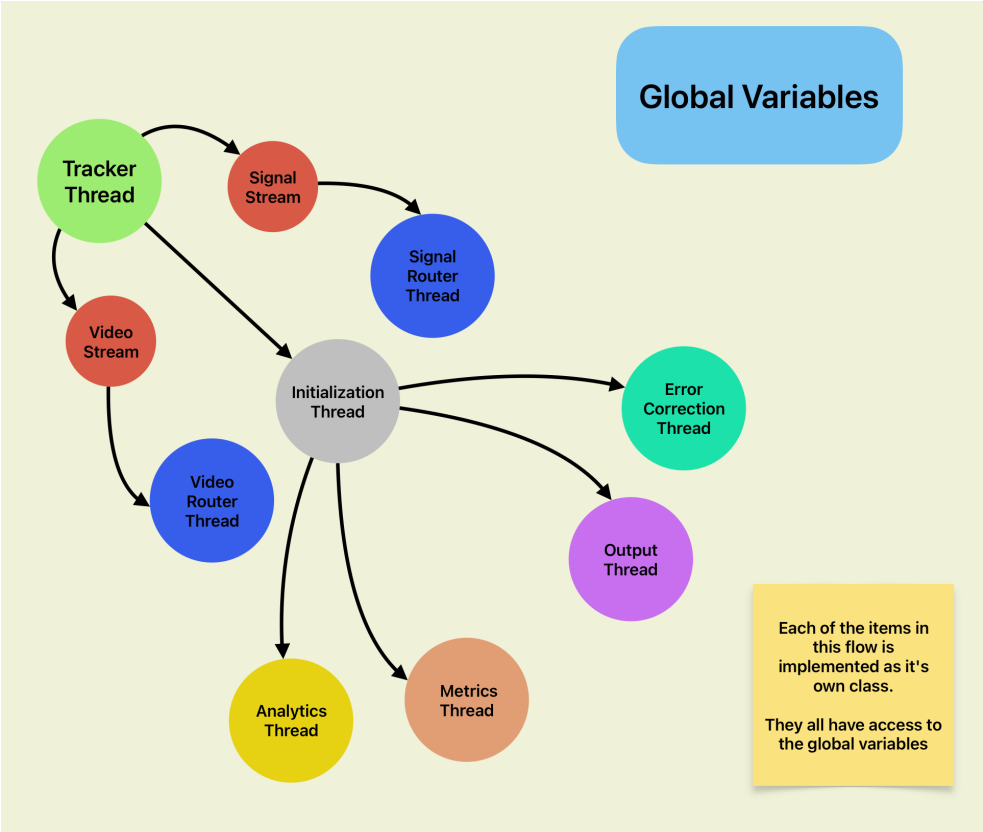
Classes:


Over Normal Under

Another problem arises: Given a frame, the only relevant information is the area around the tip in the direction of recently extruded material. This is why we need to track the tip.

Solution



The given diagram displays how the system is divided into distinct threads and illustrates the process flow. All threads access some shared global variables. The tracker thread initiates and manages all other threads. This system primarily revolves around three key components:

1. **Initialization**: Predicting the path of tip tracking and aligning with the video.
2. **Error Correction**: Correcting errors through a feedback loop.
3. **Analytics**: Identifying anomalies.

For the first two steps, we utilize a fine-tuned Yolov8. For the anomaly identification, we rely on a fine-tuned MobileNetv3 model.

Why we need Tip Tracking & Examples of Video and Anomalies

Challenges

This system is designed with simplicity and versatility in mind. Our goal is to make it readily adoptable for anyone, ensuring it can seamlessly integrate with various 3D printers in diverse settings. There's no need for a precise camera placement, giving users flexibility. The system adjusts to different resolutions, simplifying setup processes.

It's built to work with any printer that operates on g-code, even if the g-code interpretations vary. With edge deployment capabilities, it can run on a Raspberry Pi, streaming in real time. Given that the YOLO model may occasionally err in object detection, our system is made to account for these discrepancies. Moreover, users will benefit from a clear visualization of the tip tracking procedure.

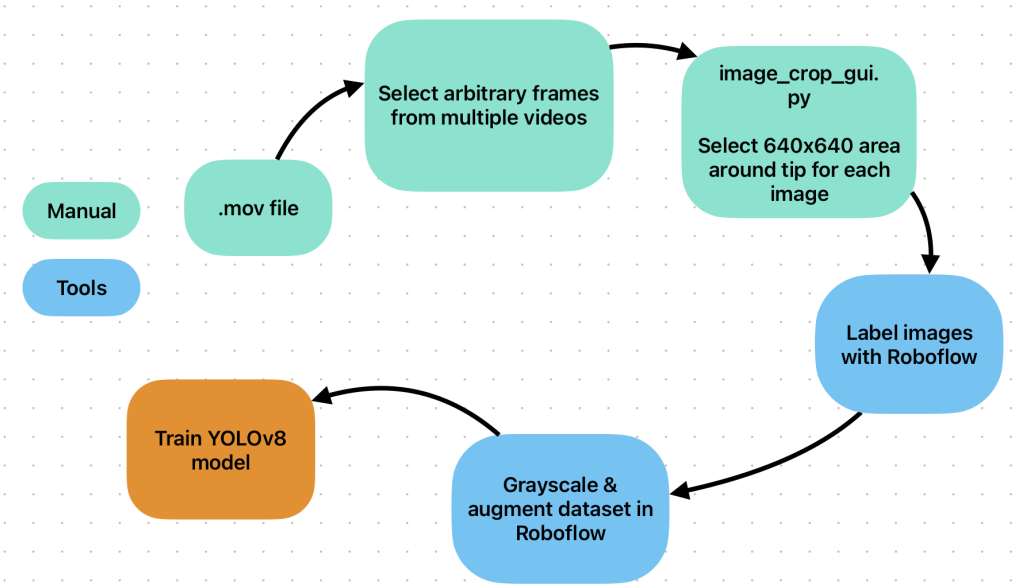
Given these constraints and requirements, the ensuing challenges and our tailored solutions will be elaborated below.

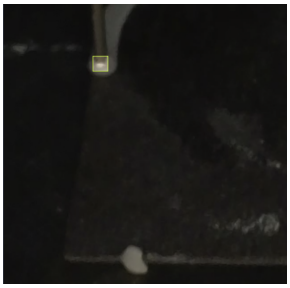
Model Training

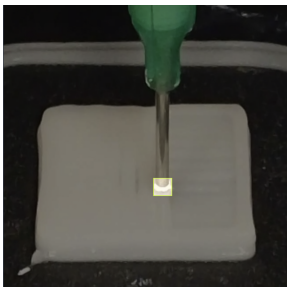
https://github.com/BrianP8701/Anomaly_Classification

Object Detection for Tip Tracking

We utilized the YOLOv8s model, a prominent object detection system, and fine-tuned it through Roboflow, a platform that streamlines the computer vision data process.

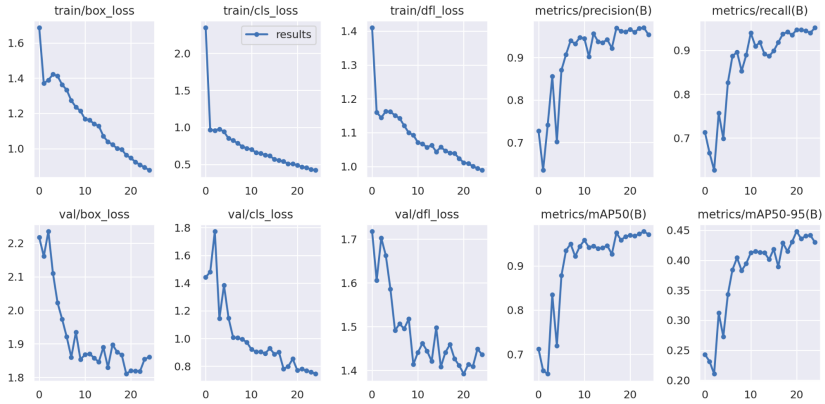






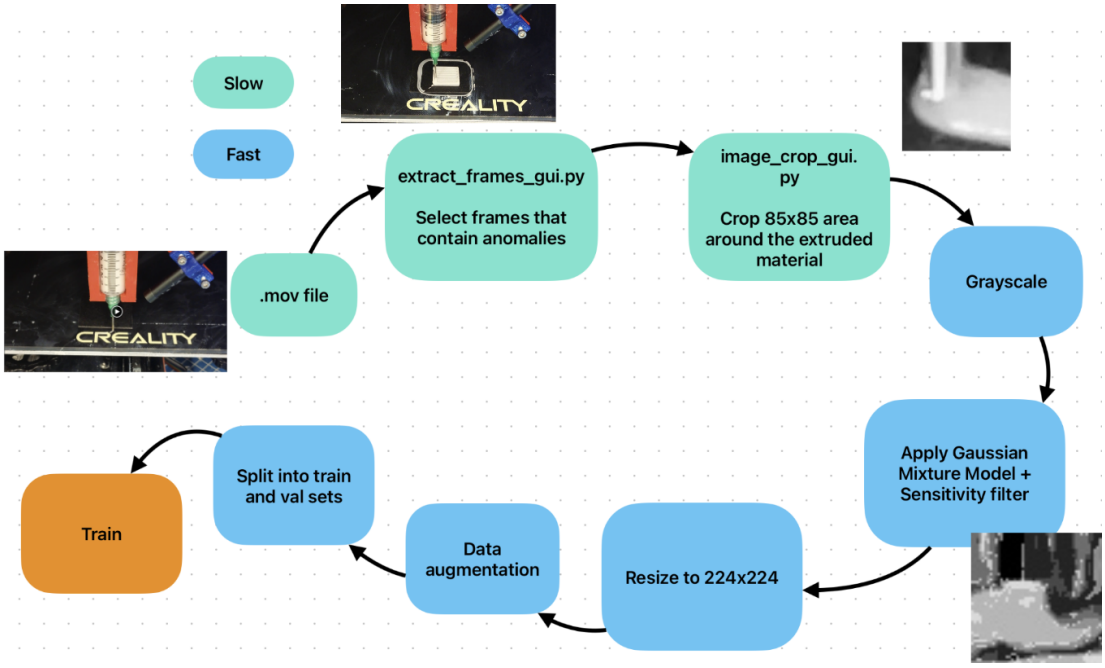
Input Size	Dataset Size	Augmentations	Precision	Recall	MAP50
640x640	2407	1200	0.967	0.931	0.968

We added onto our dataset and retrained multiple times, aiming to fill in gaps and weaknesses of the model, adding null images and wrong detections etc. We applied augmentations of blur, noise and brightness.

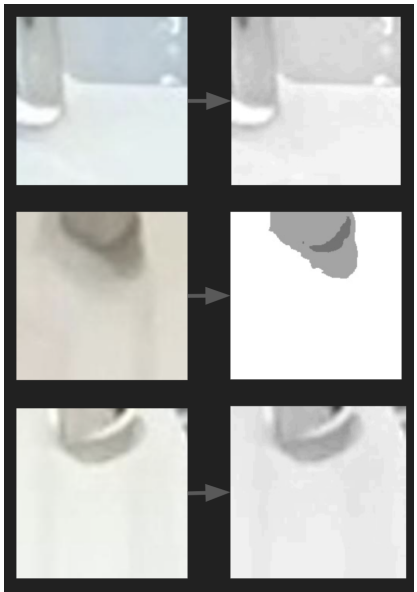


Object Classification for Anomalies

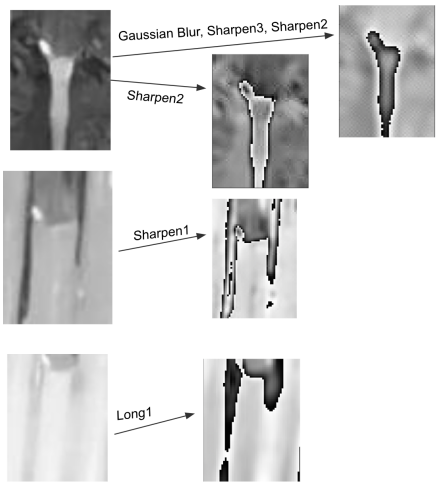
We tested various models, settings, and preprocessing methods to find the best model. Our data is limited because only a few frames in each video show over or under extrusion, making data collection slow. Here's our data pipeline:



Examining the pipeline: I developed several GUIs to address the initial data collection bottleneck. I opted for an 85x85 crop size as it best captured material near the extrusion tip. I used grayscale to reduce dimensions. In some images, edges and material are barely visible. Therefore, a filter was used in preprocessing to enhance these crucial edges.

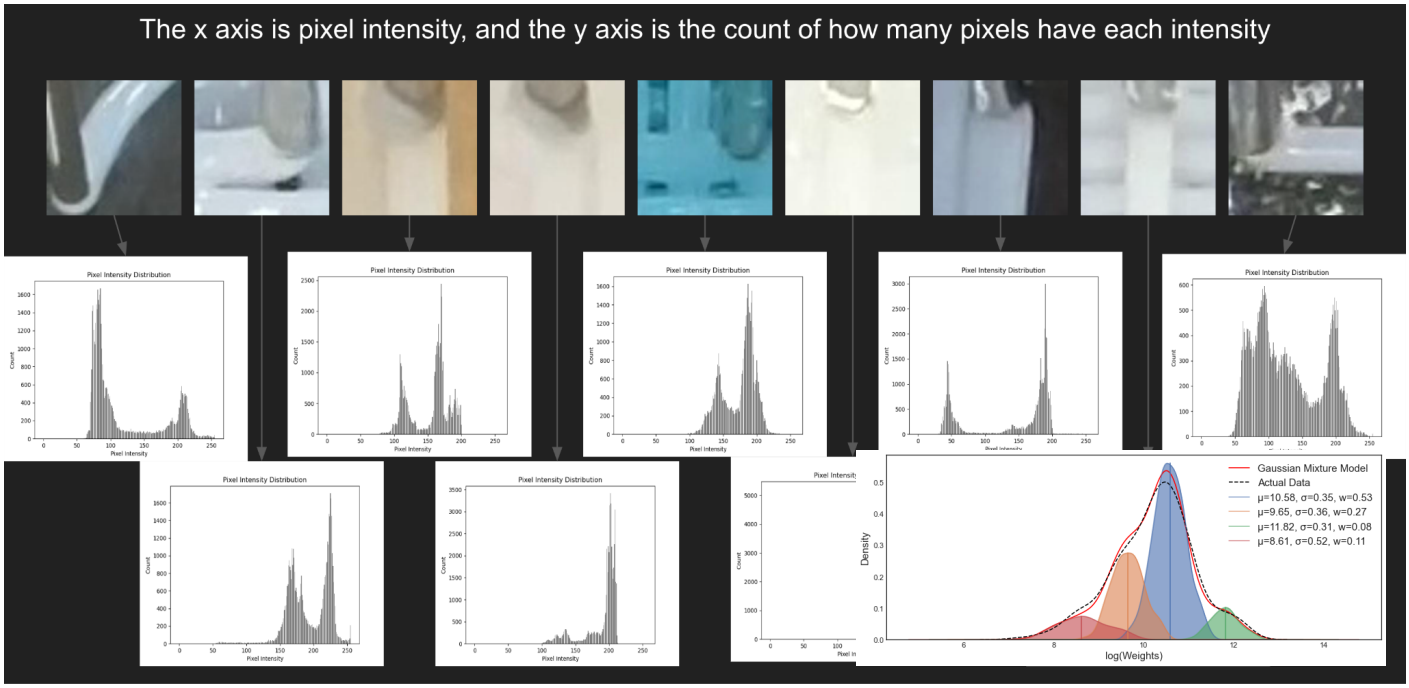


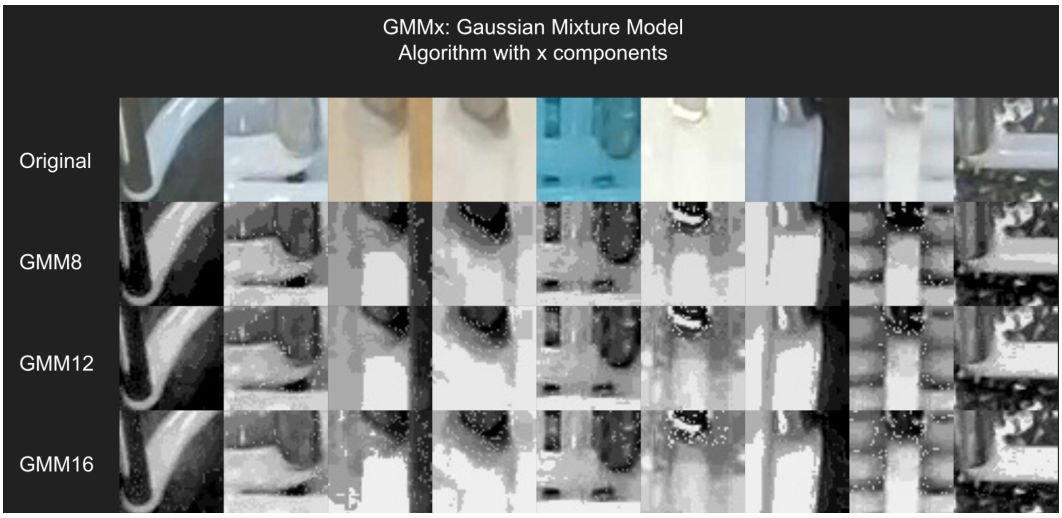
I initially attempted a filter that segmented the image into distinct color intensities, using the original image's color intensity standard deviation. While somewhat effective, it missed the subtlest edges.



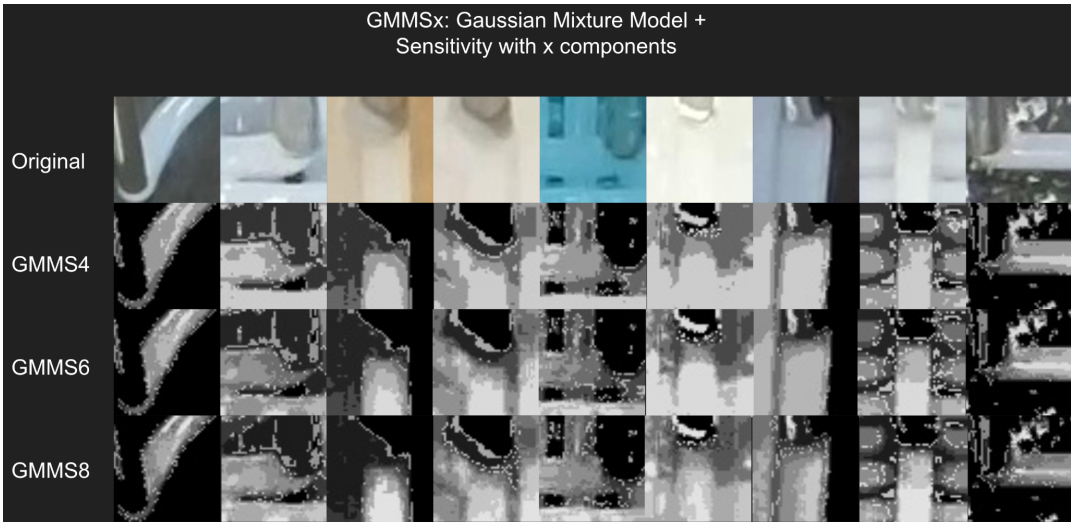
Drawing inspiration from the convolutional operations in CNNs, I experimented with various kernels and matrix filter combinations on the images. However, no single combination consistently worked across all images. I tested standard filters such as the Non-local Means Denoising from cv2 and the Unsharp Mask from PIL. Yet, neither succeeded in enhancing the faint edges.

An observation was made, each image features a limited set of objects and hues: material, tip, bed, shadows, and gleam. Each color intensity in the image uniquely follows a normal distribution with its distinct mean and variance. Hence, Gaussian Mixture Models are apt for this task.

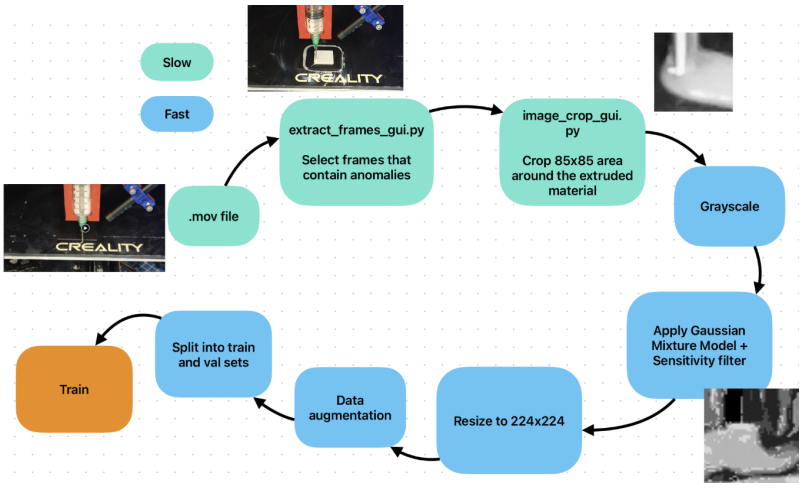




Using Gaussian Mixture Models, we can sharply define even the subtlest edges. I tested various component numbers, indicating the assumed Gaussians in the mixture. Next, we prioritize sensitivity around brighter colors since it's challenging to discern edges when material accumulates. This focus ignores the darker bed, which isn't of concern. This final refinement resulted in our preprocessing step, dubbed GMMS.



Through experimentation, GMMS6 is found to perform the best when training models.



Back to examining the pipeline, the images are resized to 224x224, suitable for EfficientNet, MobileNet, and ResNet, which have trained on the 224x224-sized ImageNet dataset, making them adept at recognizing features of this scale. Post-resizing, the images undergo augmentation and are split into training, validation, and test datasets.

Dataset Summary:

Under: 436 images | Normal: 463 images | Over: 446 images

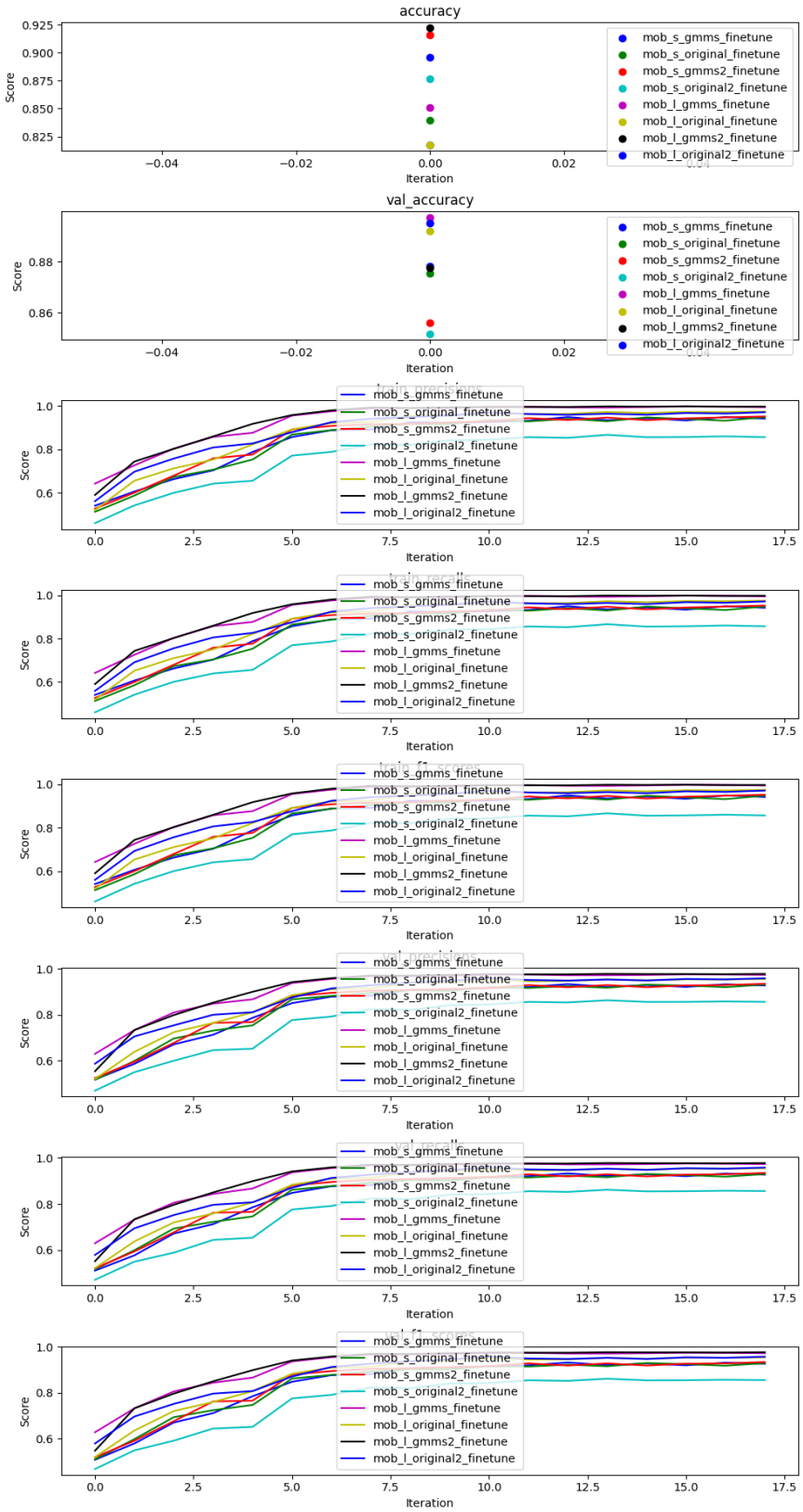
Extrusion Classification Dataset

Augmentations for each class include rotations, brightness, saturation, contrast, hue adjustments, and flips (both horizontal and vertical) creating 60 new images per class. The data distribution is 75% for training, 15% for validation, and 10% for testing.

Training Summary:

Fine tuning, model checkpointing, 4 batches, ~12 epochs, learning rate decay, early stopping. learning_rate=0.01, momentum=0.9, step_size=5, gamma=0.1

After trials of various models, settings, hyperparameters and preprocessing steps, the combination of the described preprocessing and training procedures outlined above yielded the highest accuracy. The standout was a fine tuned MobileNetv3Large model, which not only won in accuracy but also had superior computational efficiency.



To the left shows one of the comparison rounds, where we identified our best performing model. Here we compare 3 different settings and their combinations: Fine tuning MobileNetv3Small vs MobileNetv3Large. Using no GMM preprocessing vs using GMM preprocessing vs using GMMS preprocessing with additional sensitivity to more intense pixels.

The results are fitting, showing that indeed using GMMS with extra sensitivity is beneficial, and training the larger model results in better performance. The training and validation metrics are nearly indistinguishable indicating that it is not overfitting.

Best performing model accuracy:

0.9025974025974026

Implementation

In this section I will describe the design choices and details of the system.

https://github.com/BrianP8701/STREAM.AI

Input

1. G-Code:

- Instruction set for 3D print.
- Simple and sequential: it runs from start to finish.
- Specifies the top speed at any moment and provides movement coordinates.
- The printer moves directly from one point to the next without deviation.

2. Video Stream:

- Live footage of the 3D printer bed.
- The angle of the camera relative to the printer is expected to be roughly consistent.

3. Signal Stream:

- Managed by the SKR board, which controls the 3D printer.
- A hardware connection links the SKR board to our Raspberry Pi.
- The Raspberry Pi gets a signal every time a movement completes. This is the signal, and consists of the bed position and time. [time, x, y, z]
- This data streams in real time, parallel to the printing process.

Initialization

First and foremost, the system, starting in the main Tracker Thread, begins receiving and routing frames and signals through their corresponding routers. These routers will pass the data to the correct objects and threads through the shared global variable space based on the current state of the system (Initializing or Tracking).

The duty of the initialization phase is to autonomously adapt to the camera position. It primarily performs these tasks:

1. **Make initial predictions:**

```
G1 F480
G1 X84.947 Y80.391 E4.77753
G1 X87.711 Y79.388 E9.55561
G1 X90.000 Y79.143 E13.29648
G1 X110.000 Y79.143 E45.79648
G1 X112.913 Y79.542 E50.57431
G1 X115.613 Y80.707 E55.35281
G1 X117.901 Y82.554 E60.13107
```

   a. To transform G-code into predictions for every moment in time, we first break down the G-code into a sequence of motion instructions between corners (or waypoints) where there are changes in direction or speed.
   b. For each of these motions, we calculate the optimal speed profile considering constraints like maximum speeds and acceleration. We need to look at the future 2 moves as well, to determine what the final speed at the end of the motion should be.
   c. Given the optimal speed profiles for each motion, we'll calculate how long each move should take using kinematics. One problem cannot be solved analytically, and needs to be solved iteratively. The iterative method tweaks the distance over which we accelerate or decelerate, attempting to converge on the desired final speed without exceeding the machine's constraints.
   d. After computing the speed profile, the script then divides the motion into frames (based on the given frames-per-second parameter) and predicts the position and angle of the machine's tip for each frame. By the end of this process, we have a detailed account of where the machine should be and at what angle for every single frame of the motion.

2. **Derive millimeter to pixel ratio:**

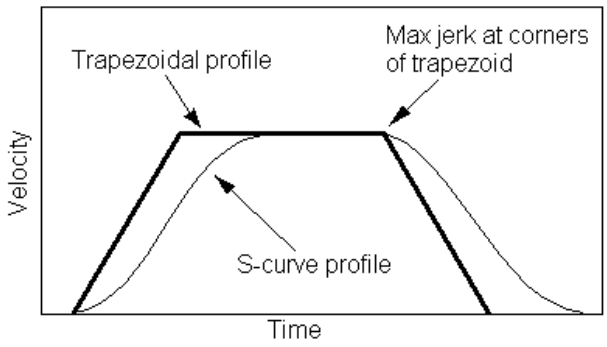
   For robustness, the system will autonomously map bed predictions to screen predictions without requiring users to enter any configuring variables. By waiting for two signals and observing their millimeter disparity on the bed, we can correlate this with the pixel difference YOLO detects, facilitating the necessary conversion. We slide a 640x640 window across the image to run YOLO.

3. **Synchronize predictions with video:**

   To initiate the process, we align our screen predictions using the timestamp of the first YOLO-signal inference from the video.

Predictions are imperfect due to accumulated small tracking errors. Reasons include:

1. Merlin firmware in our 3D printer employs Beziet's 6th order function for movement extrapolation, while our algorithm uses a simpler 2nd order function. However, our model considers acceleration.
2. Some printers adjust movement factors, like speed and acceleration.
3. Minor deviations arise when the printer halts.
4. Merlin Firmware has intricate mechanisms for gcode interpretation.



Instead of replicating Merlin's exact approach, we designed a straightforward interpreter with real-time error feedback loops, enhancing system robustness and compatibility with various 3D printers.

Error Correction

Predictions face two error types: Temporal and Spatial, addressed by the ErrorCorrection thread.

1. Spatial Error: Mainly arises from incorrect ratios or YOLO inference. These are rare, typically minor initial tracking deviations. For demonstration, an exaggerated error was intentionally introduced in the Spatial Error Correction video.

Measure	Detect	Correct
Everytime a signal arrives, we run YOLO to see where the tip is. We compare this to our prediction and measure the difference in pixels.	When there is a consistent spatial error of the same magnitude and direction for sufficient time.	Shift all predictions by spatial error.

2. Temporal Error: This was a significant challenge. Without input signals, gauging temporal error was tough. The ultimate solution simply involved immediate measurement and correction

of temporal discrepancies. This error stems from varying interpretations of gcode, especially differing accelerations.

Measure	Detect	Correct
Everytime a signal arrives, we compare the time of the signal, compared to our predictions. We measure error in seconds.	We don’t look for any consistent errors. When there is an error, we simply perform a correction.	Pause or skip ahead to realign.

Analytics

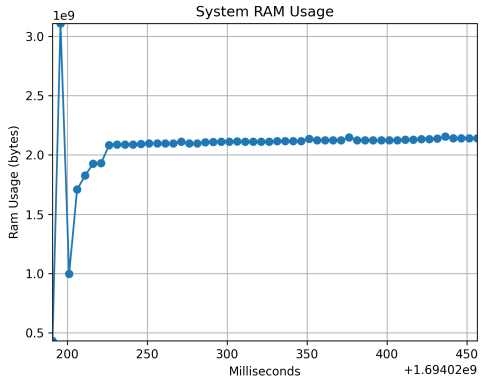
Incorporating a real-time feedback loop (autonomous corrections) results in near-flawless tip tracking. Using YOLO isn't constant; it's employed during signals and error measurements. This enables efficient pinpointing of the recently extruded material around the tip—our target. Given our knowledge from the gcode about the printer's movement direction, we crop accordingly. This cropped image then enters our data preprocessing pipeline and undergoes classification by the tailored MobileNet model, determining whether extrusion is under, normal, or over.

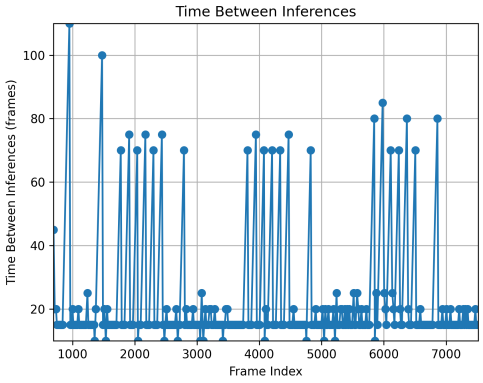
Metrics

To evaluate our system, we'll focus on its computational efficiency, while the accuracy is derived from the MobileNet model's test performance. We'll assess:

- **RAM Usage**: We'll monitor RAM consumption over time.
- **Inference Intervals**: Given the adaptive nature of our MobileNet model, observing the intervals between successive inferences provides a measure of system speed. The model refrains from inferring with a full buffer.

These metrics will be gauged on both my device and a Raspberry Pi 4, with respective specifications provided below.





Deployment to Raspberry Pi

Raspberry Pi Metrics

Future Improvements

1. Processor (Chipset) Details:
   - Model: Apple M1 Pro
   - Total Number of Cores: 14

2. Memory:
   - Size: 16 GB
   - Type: LPDDR5
   - Manufacturer: Hynix

3. Graphics/GPU:
   - Chipset Model: Apple M1 Pro
   - Metal Support: Metal 3

4. Display:
   - Type: Built-in Liquid Retina XDR Display
   - Resolution: 3024 x 1964 Retina

5. Storage:
   - Type: SSD (APPLE SSD AP0512R)
   - Capacity: 500.28 GB
   - Protocol: Apple Fabric (indicating it's an NVMe SSD)
   - Free Space: 120.4 GB

6. File System:
   - Type: APFS

https://qengineering.eu/install-opencv-on-raspberry-pi.html
https://qengineering.eu/install-pytorch-on-raspberry-pi-4.html
https://gist.github.com/wenig/8bab88dede5c838660dd05b8e5b2e23b
https://onnxruntime.ai/docs/tutorials/iot-edge/rasp-pi-cv.html