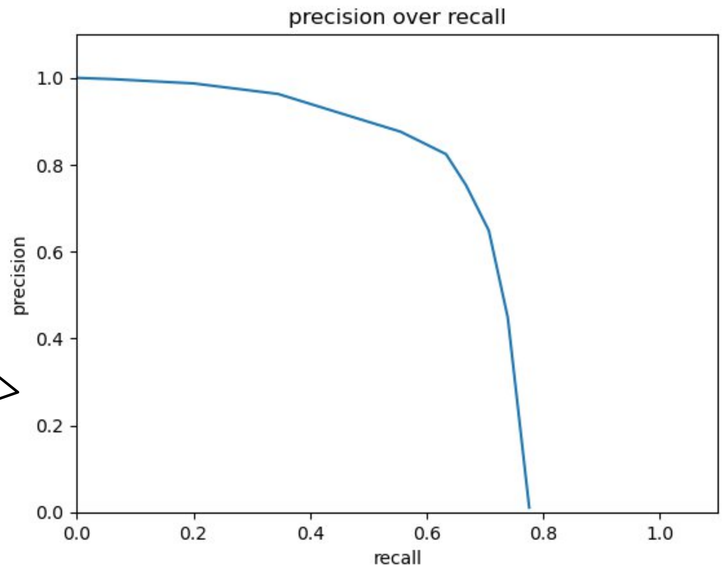
Embedded Machine Learning Lab Challenge

By Valentin & Max

Magnitude Pruning

pretrained model
- adapted for person class
- fine-tuned for 20 epochs

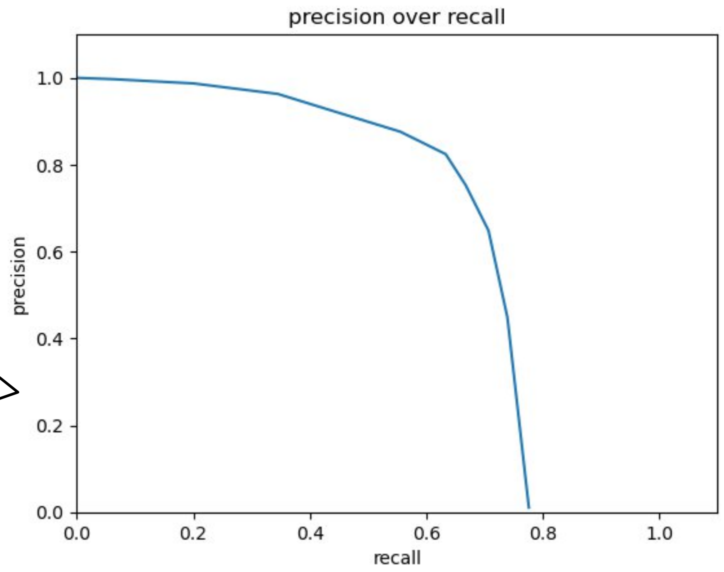
-> 0.65 AP



Magnitude Pruning

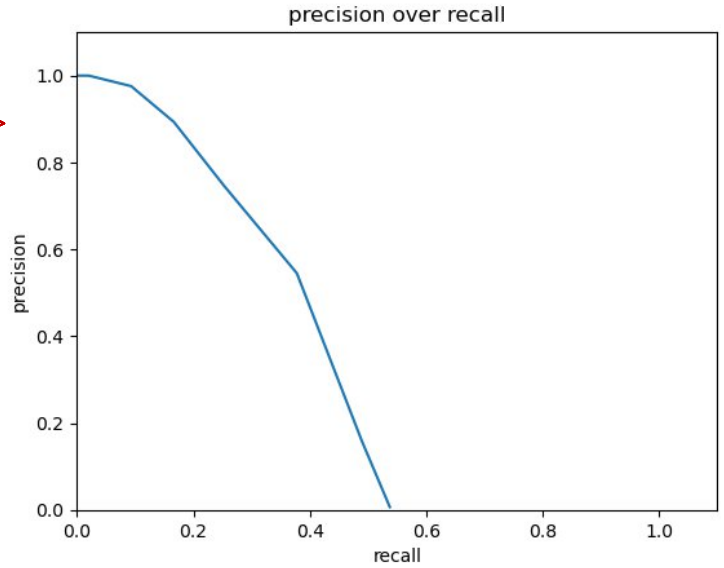
pretrained model
- adapted for person class
- fine-tuned for 20 epochs

-> 0.65 AP



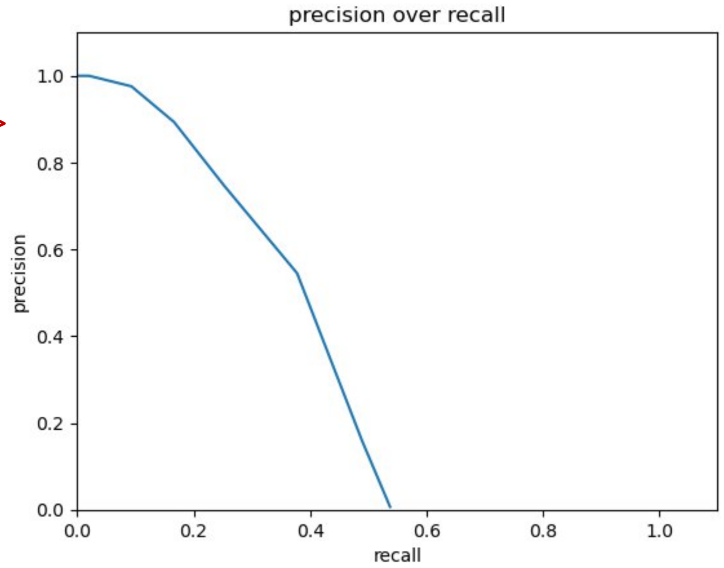
pruning ~40% of params

after magnitude pruning
-> 0.30 AP



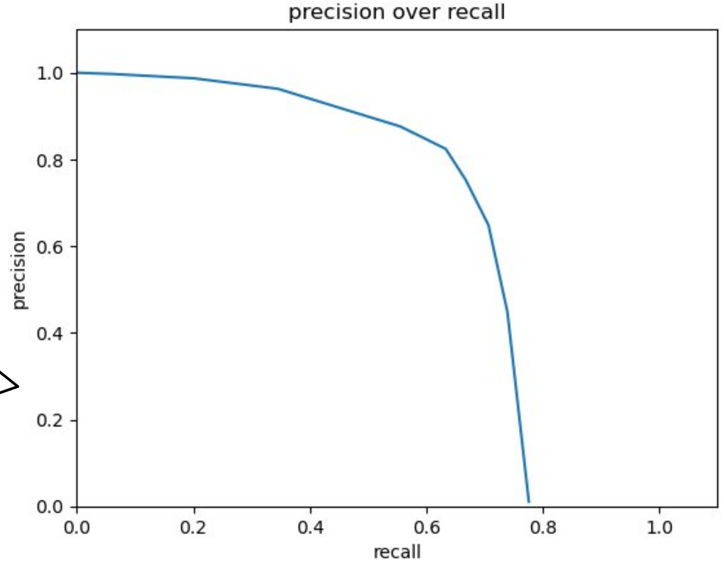
Magnitude Pruning

after magnitude pruning
-> 0.30 AP

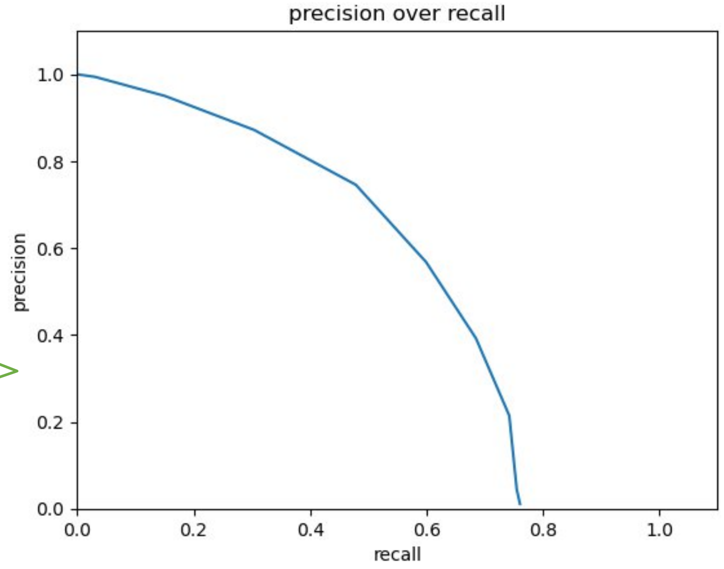


pretrained model
- adapted for person class
- fine-tuned for 20 epochs

-> 0.65 AP



pruning ~40% of params

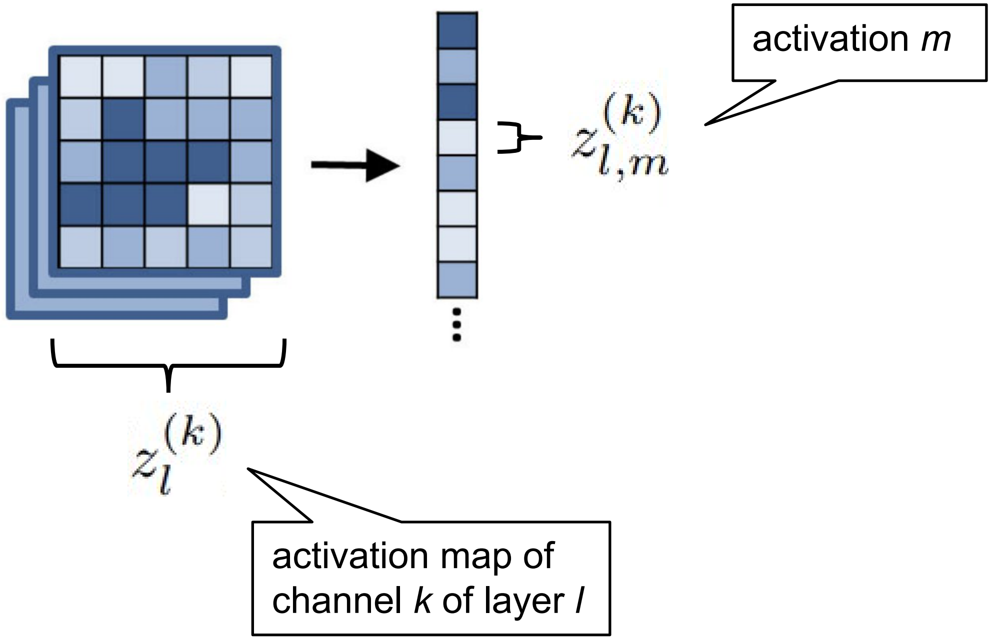


after Taylor pruning
-> 0.51 AP

Taylor Pruning [1]

Channel pruning based on importance

- Defined by approximate change in loss caused by removing channel
- Activations & gradients gathered during forward passes

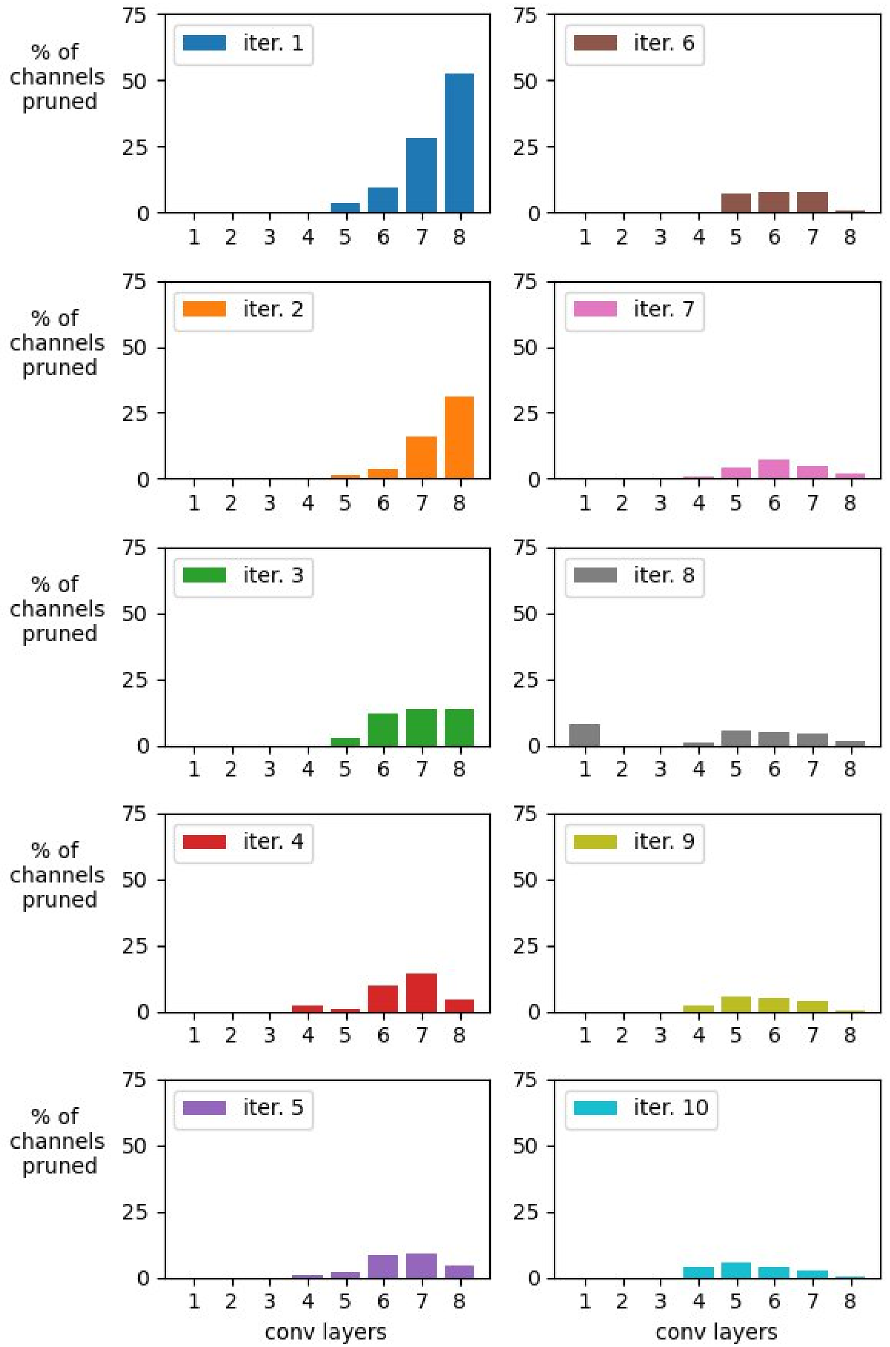


$$\underbrace{\Theta_{TE}(z_l^{(k)}) = \left| \frac{1}{M} \sum_m \frac{\delta C}{\delta z_{l,m}^{(k)}} z_{l,m}^{(k)} \right|}_{\text{importance of channel } k \text{ of layer } l}$$ [1]

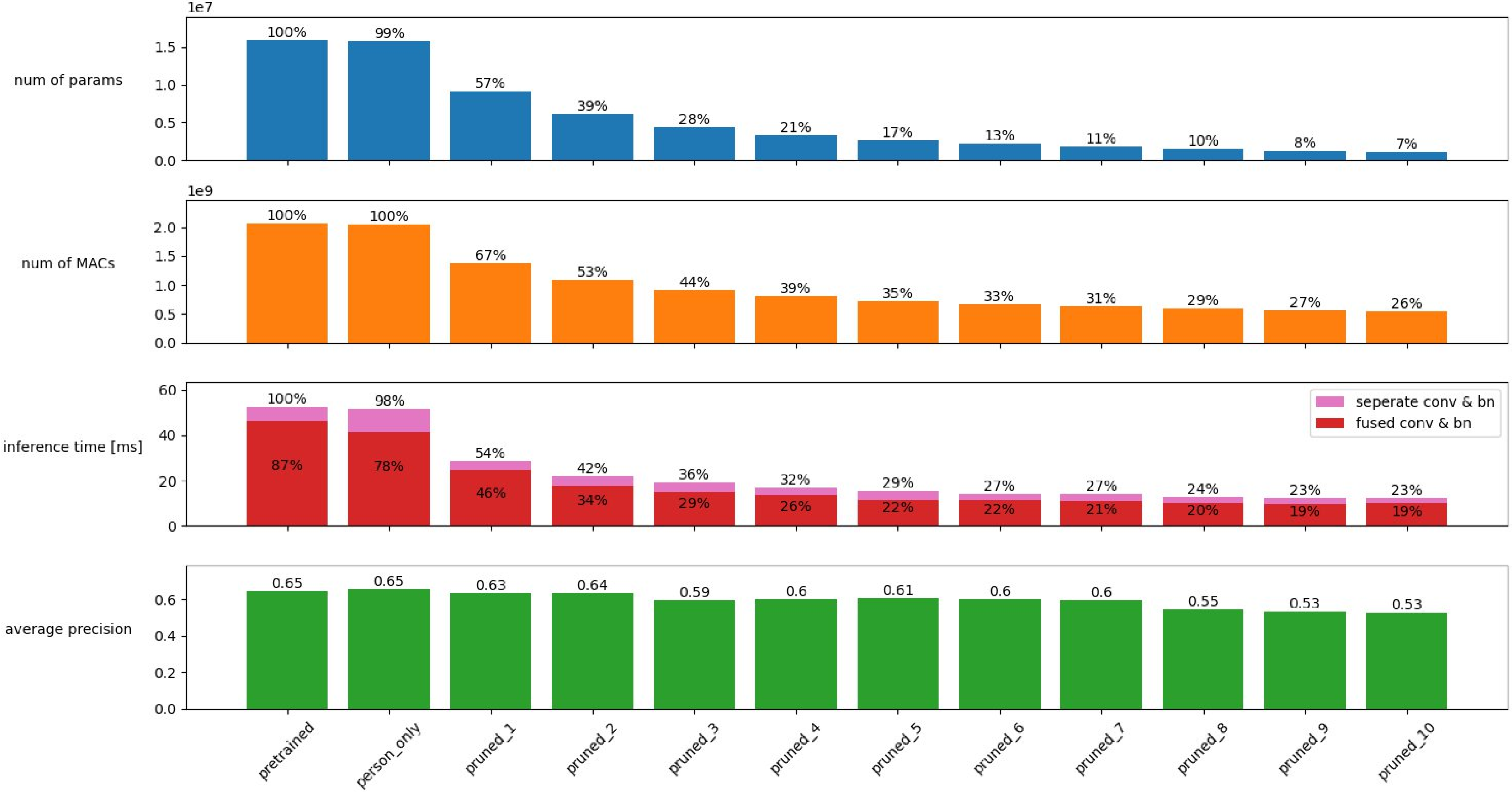
Iterative Pruning

For 10 iterations:

1. Taylor prune *k* least important channels
   - *k* proportional to num of params
2. Fine-tune for 10 epochs to regain performance
   - Very low learning rate
3. Evaluate AP
   - Compare APs over iterations



Pruning Statistics



Live camera flow

Computation steps

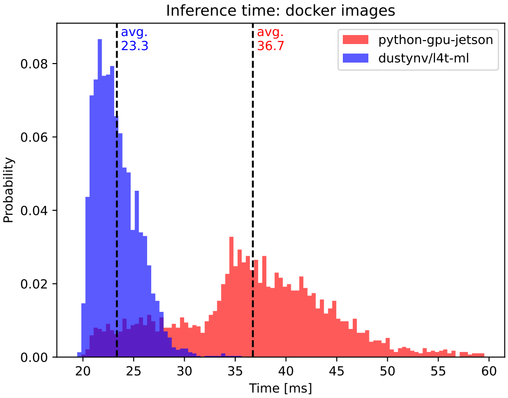
1. capture image with camera (disabled JPEG encode for Jupyter)
2. run model on image
3. filter anchors, apply non-maximum suppression
4. draw bounding boxes and fps
5. compress image to JPEG (OpenCV should use libjpeg-turbo already)
6. show image to user (MJPEG stream)

Jupyter notebook unreliable: kernel crashes, unresponsive, ...
→ **run on a custom web server inside docker**

Docker container

- provided image outdated (2021)
- use "l4t-ml" from dustynv (11/2023)
  `https://github.com/dusty-nv/jetson-containers`
- chose matching NVIDIA JetPack/L4T
- newer libraries, **faster inference**
- unchanged performance on other parts

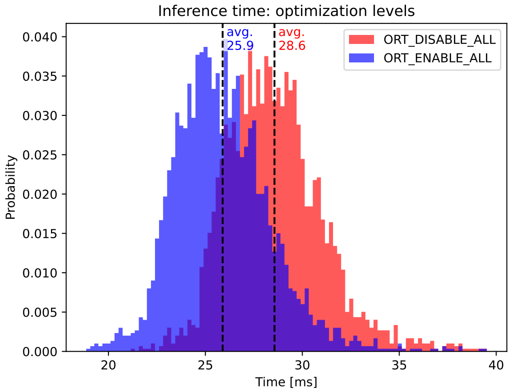
- simple web server with Flask
- low resource usage
- has endpoint serving MJPEG stream
- UI to start/stop camera



ONNX runtime

- use for inference on GPU
- supposedly faster than PyTorch
- represents model computations as graph
- use optimization `ORT_ENABLE_ALL`
  $\rightarrow$ fuses conv and batch-norm layers

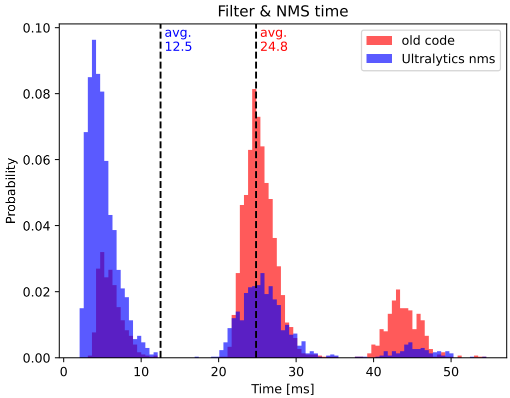
- Quantization
  - int8 $\rightarrow$ not supported by GPU
  - float16 $\rightarrow$ no speed up



Filter boxes and NMS

- original code: slow, pure Python impl.
- use `non_max_suppression()` from Ultralytics library

- licensed as AGPL v3
- everything combined in one function
- based on native `torchvision.ops.nms()`



Pipelining

- split work between GPU and CPU
- parallelize steps
- as Python `threading.Thread`
  $\rightarrow$ GIL mostly no issue
- connected by queues, length limited

- **Input**: image from camera callback
- **Output**: byte stream for HTTP response

Stage 1 *on GPU*

model inference only

$\downarrow$

Stage 2 *on CPU*

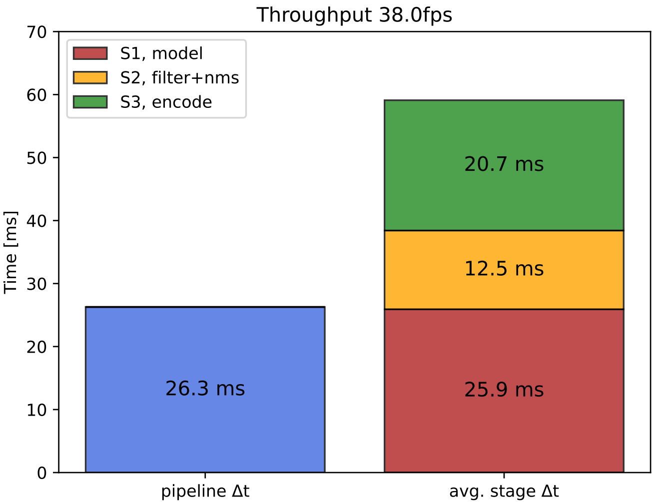
filter boxes, non-max suppression

$\downarrow$

Stage 3 *on CPU*

encode JPEG, draw boxes/fps

Pipelining: results



References

[1] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, & Jan Kautz. (2017). Pruning Convolutional Neural Networks for Resource Efficient Inference.