Alexa Rockwell
SUID# 06407300
CS 230 - Deep Learning
Spring 2021

Bounding Box Detection in Embedded Environments

Project Category: Computer Vision

Problem Description

This project aims to customize YOLO to work on a Raspberry Pi 4 (RPi) with optimal accuracy and FPS (frames per second). The goal is to deploy a model that can optimally follow a detected human using an 8-megapixel camera.

To accomplish this, I will use an open-source implementation of YOLO[1] and modify it using techniques found in my literature review. Additionally, due to the perspective of the robot's camera, sometimes the camera may only capture part of a human. The modified YOLO is expected to detect full humans and partial bounding boxes positively. To summarize, this project will attempt to optimize YOLO and find which augmentation on the dataset works best in making a model that can be deployed on the robot. An added level of complexity is deploying the model into a compressed format onto a microcontroller with limited compatibility from a software and hardware standpoint.

There are so many applications for autonomous, spatially aware robots, from delivery services to healthcare, to domestic companionship. Many people can benefit from the aid of a robot in their own home, whether for functional or entertainment purposes.

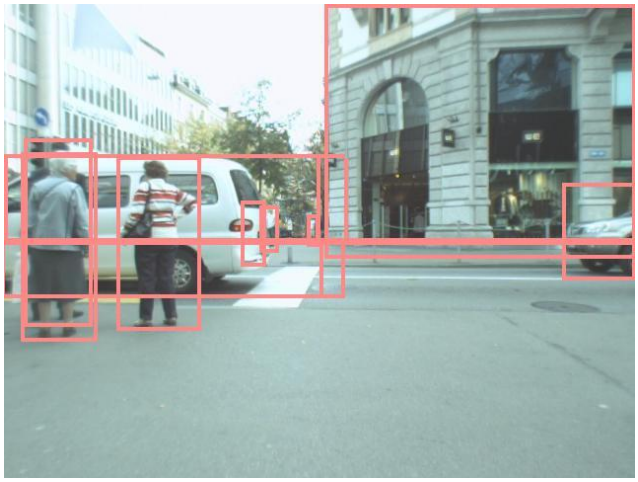
Dataset & Augmentation

Multiple Object Tracking (known as MOT) Benchmark and Challenge has an annotated dataset of frame-by-frame person detection and tracking. The MOT data set contains 5316 frames, with 1920x1080 resolution. Additionally, the data set has 101,305 bounding boxes in total for individually detected humans.

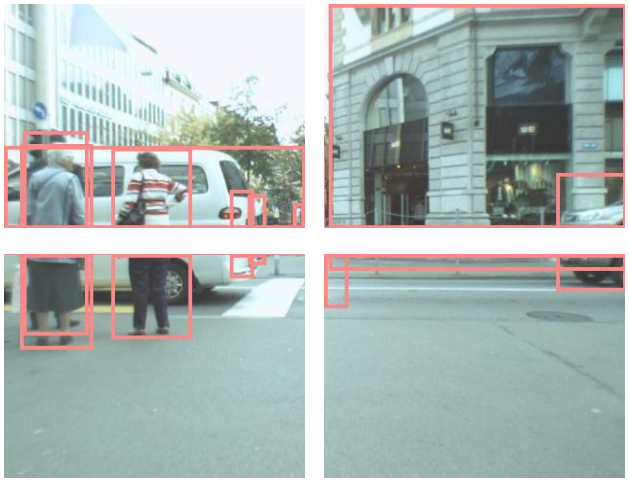
These are a few techniques to apply to the dataset to improve the performance in the robot’s environment. First of which is downsampling the images from the MOT datasets to match the resolution of the 8-megapixel camera. Next, we may segment the dataset, as well as annotated bounding boxes. This results in 21,264 images, with an exploded set of bounding box annotations: approximately 405,220 since I pruned segmented annotations that were too thin or long.

I hypothesized that this augmentation would be critical in improving a model's performance on the robot. It is also projected to have lower accuracy since there's an increase in variance; detecting a whole body is intuitively easier than identifying parts. Another characteristic of the dataset is that many images contain crowds of people individually annotated. I expect this to be a pain point for accuracy since the model may detect the full crowd as a person when using non-max suppression.

I also considered skewing the perspective of the images from MOT since the robot's camera is one foot off the ground. This would ensure the angle of the altered photos would match the robot’s environment, but



Original MOT Ground Truth Annotations



Segmented MOT Ground Truth Annotations

I wasn't able to find time to implement and execute it since the quality of the segmented dataset was more important for the project.

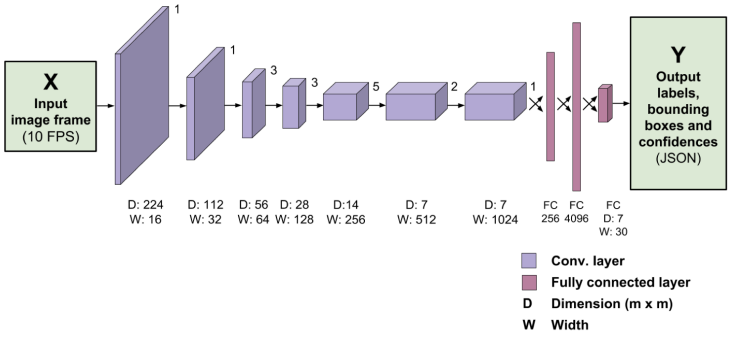
Lastly, I made a critical mistake in choosing MOT as the dataset for this project. When I was investigating inferences, I noticed it contained buildings and cars in the annotations as well, without any sort of classification enum, only bounding box annotations.

Architecture and Hyperparameters

Deployment Challenges

For an object detection project like this one, a YOLO model was the way to go. Picking which open source implementation to use led to the project's downfall. I arbitrarily picked a PyTorch implementation to find out deployment on a Raspberry Pi 4 required wiping the OS to a different version of Debian. Since this was risky and re-configuring the environment might set back my project, I decided to use Tensorflow implementation instead. It was critical to the project's success that a model I would train would be deployable on the Raspberry Pi, so this pivot was necessary.

With the second implementation of YOLO, I trained a model successfully but exporting the model to a TFLite format failed since that functionality wasn't supported on V1. After measuring the inference time of the uncompressed model, it resulted in average performance of 0.667 FPS. Since this was unacceptable, I tried a new open-source implementation of YOLO in Tensorflow, but this time with a version that supported TFLite. Compressing the model to TFLite is also a requirement if I wanted to use the Coral to speed up inference on the Raspberry Pi 4.



Post-Training Quantization on Google Coral

Chatterjee implicitly mentions Post-Training Quantization [2] as the technique used to deploy a YOLO model using a Jetson Nano and reported a 60% decrease in weight size. It is a clever way of converting all floating-point tensors into 8-bit integers. Unfortunately, after much troubleshooting, I couldn't get the Coral dependency packages installed onto the Raspberry Pi 4. Since time was running out, I abandoned this inference optimization method.

Final Approach

After settling on the final implementation of YOLO in Tensorflow, I had the choice to choose between YOLO or TinyYOLO (YOLO24). Since the literature review showed that TinyYOLO was preferable in a constrained environment due to fewer parameters, I used it as the base architecture to train on my custom dataset.

As for hyperparameters, I decided to split train, dev, and test distributions into 96.24%, 1.88%, and 1.88%, respectively. This made sense because I wanted to allocate as much data as possible (5119 images) to the training set. Next, this open-source implementation of YOLO provided hyperparameters such as a batch size of 4, and supported learning rate decay from 1e-4 to 1e-6. The default number of epochs was 100, but I terminated training early at around 70 since the average time to train took about 74 hours and hampered the iterative process.

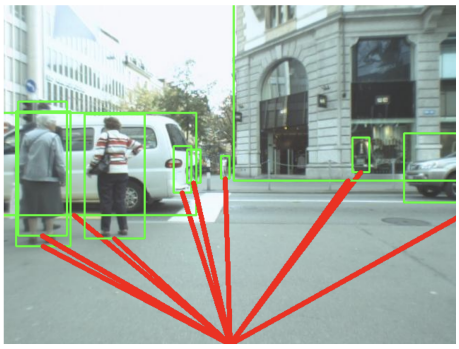
Results

	Pre Trained YOLO			Pre Trained TinyYOLO			Custom Trained TinyYOLO on MOT Dataset		
	Google Colab FPS	Training mAP50	Test mAP50	Google Colab FPS	Training mAP50	Test mAP50	Google Colab FPS	Training mAP50	Test mAP50
Unmodified MOT dataset	5.34	1.027%	2.757%	16.85	0.010%	0.061%	16.88	2.064%	2.045%
Segmented MOT dataset	7.74	0.525%	0.427%	16.84	0.122%	0.216%	16.88	0.702%	0.515%

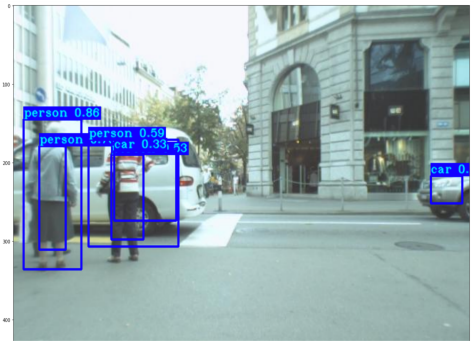
This table will report the FPS measured on Google Colab, but naturally, it's expected to decrease when running inferences on an embedded system. Testing an uncompressed YOLO model inference time in Colab with a NVIDIA Tesla V100 SXM2 takes 26 milliseconds, whereas the Raspberry Pi 4 without any Edge TPUs took approximately 24 seconds. An approximately 1000% increase. This is why exporting Tensorflow models to TFLite and using an Edge TPU is critical in embedded environments.

Analysis

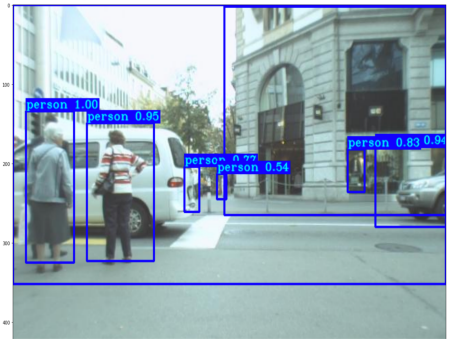
As a baseline, I tested the performance of a pre-trained YOLO and TinyYOLO on the whole and segmented dataset. While the mAP on the COCO dataset was 55% for YOLO, mAP was 1% on the MOT dataset and 0.5% on the segmented MOT dataset. While this is shockingly low, this makes sense once you consider how non-max suppression may be handling large crowds and not successfully detecting people in the distance. This error can be clearly witnessed after comparing the ground truth and detections from the test sample. With more time, I could further evaluate how lowering the IoU threshold may improve performance.



MOT Ground Truth

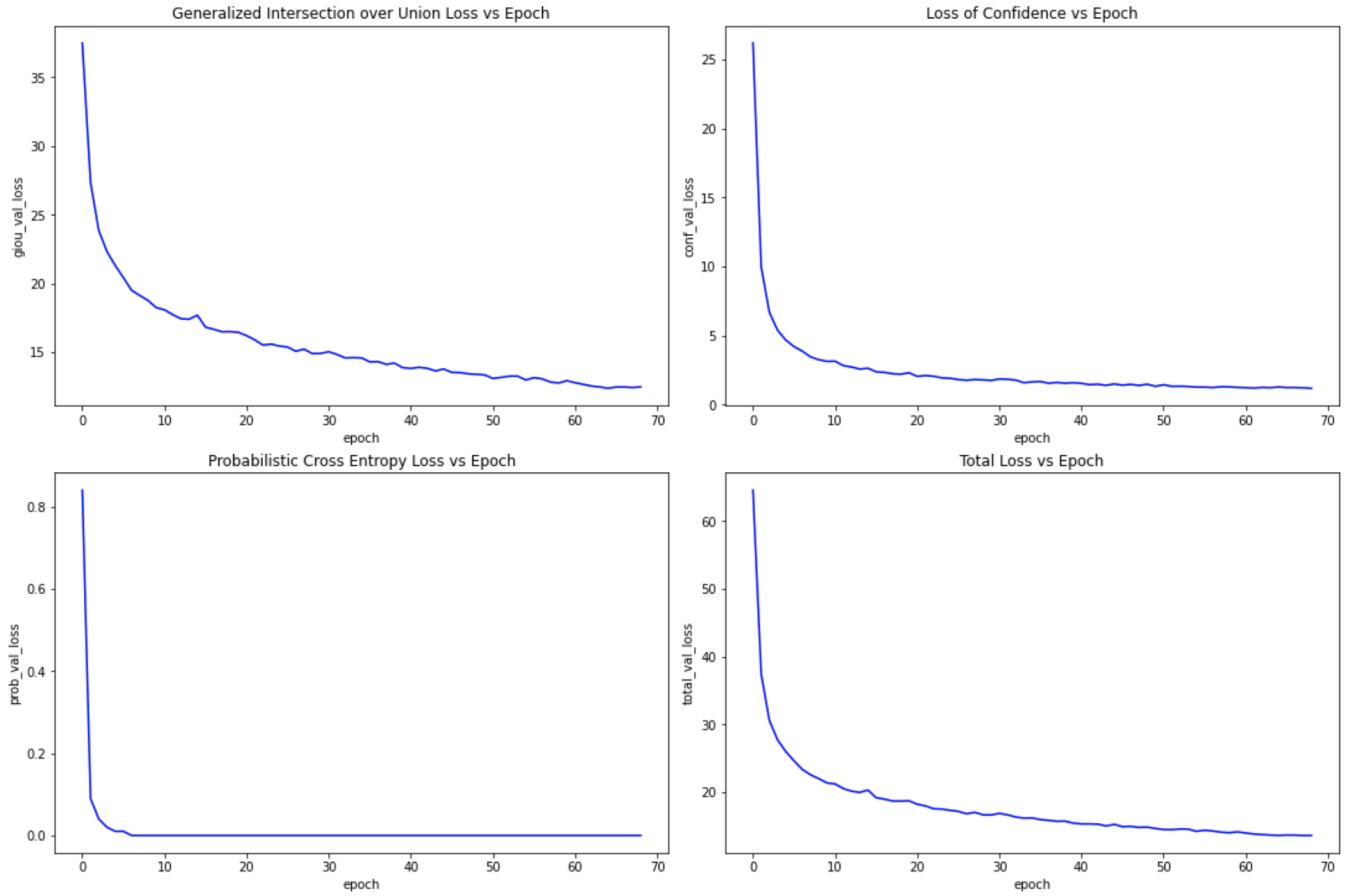


Pre-Trained TinyYOLO Inferences



Custom Trained TinyYOLO Inferences

After comparing performance, I matched the mAP performance of my custom-trained TinyYOLO model to the pre-trained YOLO. The metrics do exhibit signs of high bias and high variance. By choosing TinyYOLO, we do get an increase in FPS, but maybe deploying a YOLO with specific optimizations would reduce bias. Once that's done, then I can explore other regularization techniques if the variance is still high.



Additionally, observe the above statistics gathered from training the model over 70 epochs. They exhibit signs of training loss, converging, and making progress. Finally, after exporting the model to TFLite and deploying it on the Raspberry Pi, with identical python modules, code, and model: it still failed. After investigating with Ayush, we discovered that while the versions are identical, the models were interpreted differently due to the underlying OS architecture (aarch64 vs arm71). Here's one example of a difference I noticed between the two summaries:

Colab Version:

tf.__operators__.getitem_1 (Sli ()	0	tf.compat.v1.shape[0][0]
tf.__operators__.getitem_5 (Sli ()	0	tf.compat.v1.shape_1[0][0]

Raspberry Pi 4 Version:

tf_op_layer_strided_slice_1 (Te [()]	0	tf_op_layer_Shape[0][0]
tf_op_layer_strided_slice_3 (Te [()]	0	tf_op_layer_Shape_1[0][0]

It was responsible for throwing input shape errors on the Raspberry Pi and not the Colab Notebook. After this deadline, I plan on installing aarch64, and restarting this project from scratch to finally achieve a successful deployment.

Insights and Discussions

Aside from the deployment process, data preparation took the most time. Segmenting the dataset required a lot of trial and error to validate and execute. I've noticed Google uses segmented images as their version of CAPTCHA. This project successfully demonstrates how top-of-the-line models have yet to detect parts/segments of a class.

To get an idea of how the final product of the proposed methodology would behave, I used MobileNet on the Raspberry Pi 4 to detect and approach a person. The robot was even subject to drift since the motors weren't calibrated and someone with no mechanical engineering experience designed it. Correcting a robot's trajectory to account for the drift is a project of its own, and it would be interesting to see how Q-learning would correct its trajectory.

The biggest takeaway was that this project had too many complex components to complete over a semester. If I had a perfectly working Raspberry Pi with an identical development environment to Google Colab, I would have time to explore how to improve the TinyYOLO model on segmented data. I still think it was valuable to go through these exercises since it's a significant obstacle the current industry faces when applying research into a final product.

References

[1] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, 2016, pp. 779–788.

[2] Sayantan Chatterjee, Faheem H. Zunjani, Souvik Sen, Gora C. Nandi: “Real-Time Object Detection and Recognition on Low-Compute Humanoid Robots using Deep Learning”, 2020;

[3] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” CoRR, vol. abs/1311.2524, 2013. [Online]. Available: http://arxiv.org/abs/1311.2524

[4] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” CoRR, vol. abs/1804.02767, 2018.

[5] J. Ma, L. Chen, and Z. Gao, Hardware Implementation and Optimiza- tion of Tiny-YOLO Network

[6] Generalized Intersection over Union, giou.stanford.edu/.

PythonLessons, TensorFlow-2.x-YOLOv3, (2021), GitHub repository, https://github.com/pythonlessons/TensorFlow-2.x-YOLOv3

Singh, Aishwarya. “Selecting the Right Bounding Box Using Non-Max Suppression (with Implementation).” Analytics Vidhya, 4 Aug. 2020,