Smart Doorbell

Provided By Deeplite

Introduction

Smart Doorbells represent an excellent application for TinyML. Ideally, a video doorbell should record and/or upload video footage of important events that occur in the view of the doorbell and alert the owner when an event has happened. In practice, this is a challenging task since a constant stream of video is too much to constantly upload or store locally. Therefore the doorbell needs to be able to determine, on its own, which video clips should be uploaded or stored by identifying key objects in the video. Additionally, many smart doorbell systems are battery powered and need to last for months at a time and must protect the owners privacy. At this point in the course, you can see how this presents a challenge that requires TinyMLOps. By efficiently running object detection on the device we can greatly improve the usefulness of the product.

Challenge

For many standard computer vision tasks, there are dozens of options for developers and product managers – they can use a pretrained model, or utilize off-the-shelf CNN model repositories to fine-tune on their dataset during the model training phase of development. However, it's important to note that in many cases, such models are over-parameterized and under-optimized for the given use case. When considering what the optimized model architecture would be for a given CV application, it's essential to consider the nature of the data

distribution (how many classes, how each class is represented in the training sample, how much training data is available, how much concept drift is expected) and the required accuracy.

In the context of a smart doorbell device, we have the following requirements:

- 3-classes (person, pet, vehicle)
- >90% accuracy per-class
- Over 1M training samples

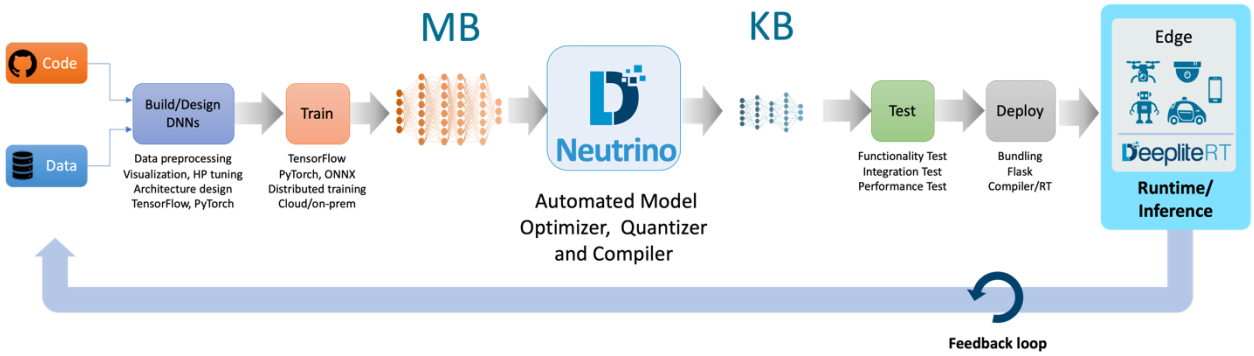
A model pre-trained on the COCO dataset, like Yolov5-l, despite being one of the best models in literature, is not exactly the best fit for this application. COCO has over 80 classes with fewer data samples than the smart doorbell use case. This means the training requirements are different for an application with higher accuracy requirements. Furthermore, it's critical to consider the inference requirements for model optimization. Latency, available memory, CPU usage are factors in creating the best model architecture.

The following are realistic requirements for the smart doorbell in this case.

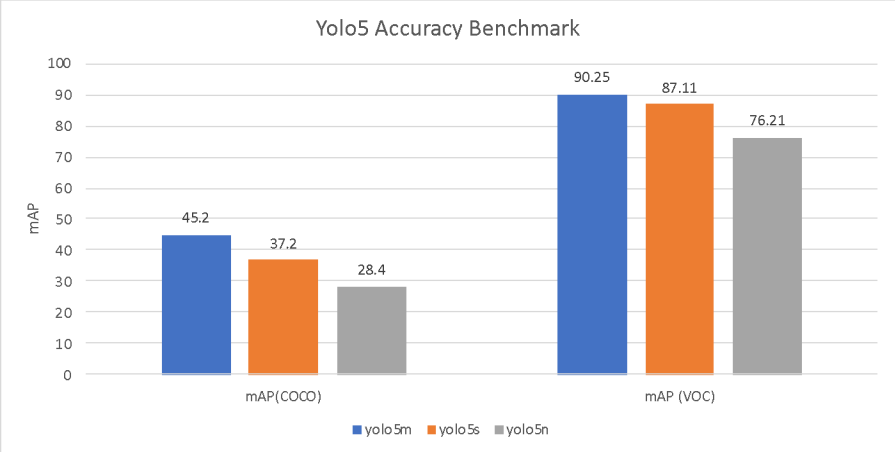
- < 250ms inference latency on 1.2GHz quadcore Arm Cortex-A53 CPU
- < 2MB runtime footprint

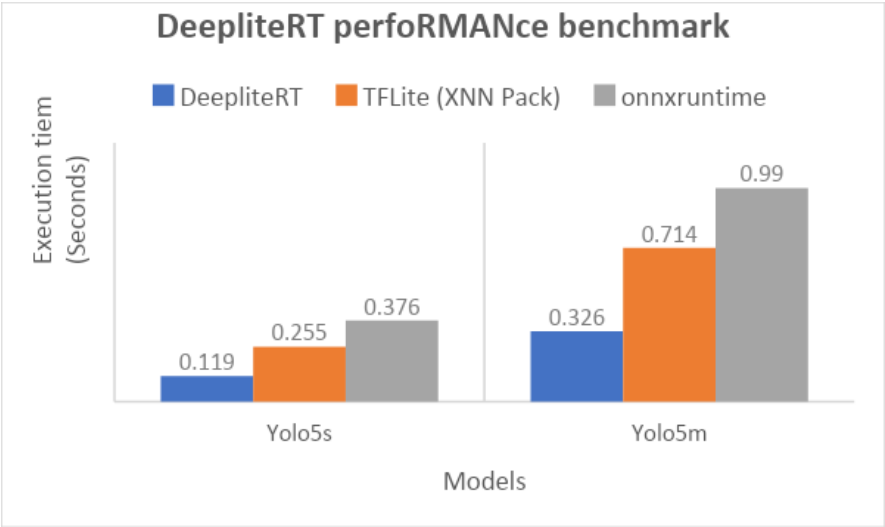
Model Optimization

TinyMLOps tools can automate the process of optimizing models to meet specific constraints. For example, Deeplite's Neutrino, shown below, is a design space exploration and low-bit precision quantization engine, which can combine both training and inference requirements into the model optimization process.



Unlike naïve weight or channel pruning or post-training quantization to INT8, combining multiple optimization techniques in a single optimization pipeline allows end users to factor training requirements (like accuracy) with inference requirements (like latency) to rapidly converge on an ideal model architecture that is ready for production deployment. In this case, the Neutrino engine found a compressed Yolov5-s model with 2bit quantized weights and activations that met accuracy criteria, inference latency (119ms vs 250ms) and provided sufficient memory savings (15.5x model compression vs FP32 and only 1.8MB overhead with DeepliteRT).





Now that the model has been optimized to meet the training and inference requirements, the next step is to convert the model into a format that can be run efficiently on the target device.

Model Conversion

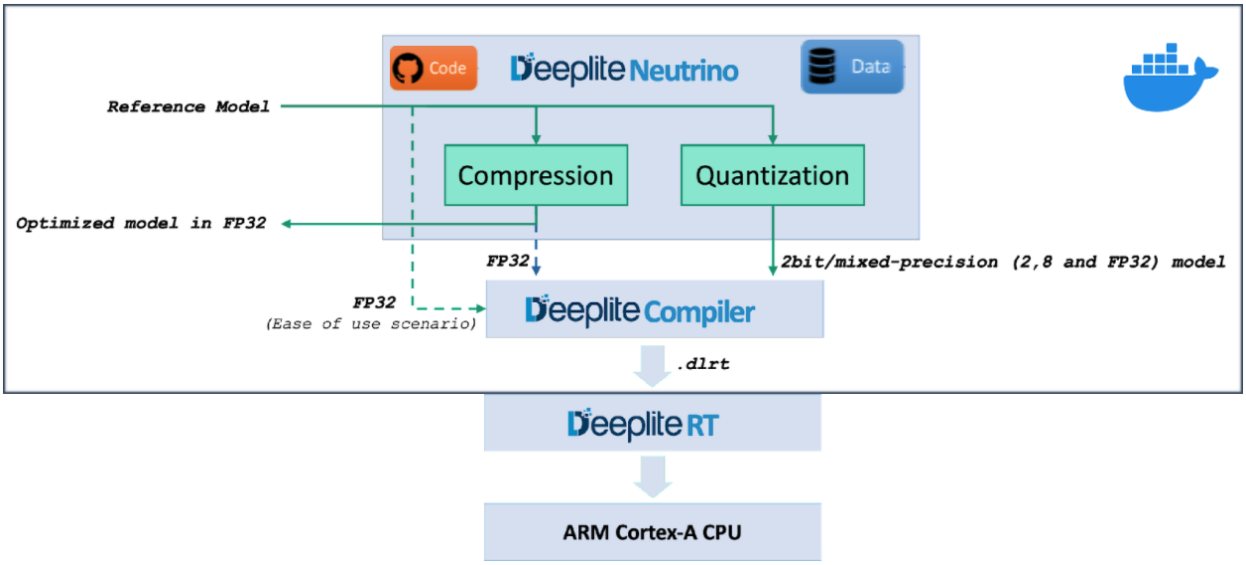
The model conversion process involves two basic inputs – the output from model optimization and the target HW for deployment. The various dependencies, versions and AI frameworks for model development - which largely focus on high-level abstraction and python programmability – must be reconciled with low level system requirements and optimization for the target hardware. When dealing with edge AI and tinyML, the latter can be a significant engineering challenge given the harsh constraints we are dealing with ($1-5 chips, 100milliwatt power budget, < 1GOPS). Unlike cloud environments common for training, containerization and VMs are less common when we are dealing with edge AI deployments that will also require continuous model update and data collection once in production.

In order to avoid the engineering cost of converting the model, model conversion tools exist that automate and optimize the process. For example, the Deeplite compiler allows the automatic conversion of an optimized model to a format that supports Arm based CPUs and runs with

optimal inference speed. In this case, the model needs to be converted in a proprietary model format called dlrt. This format supports the 2 bit precision, runs on the Linux OS and Arm CPU already installed on the smart doorbell and requires the Deeplite runtime (DeepliteRT) to be installed on the end device to run AI model inference.

Model Deployment

Now that the model has been optimized and converted for inference on the Arm Cortex-A SoC it can be deployed. The first step is to ensure DeepliteRT is installed over-the-air using the device IP address and Linux OS to infer the dlrt model on-device. The entire package is provided as a docker image to the end-user for ease-of-use when automating the model conversion process. Then the model can be sent to the device and periodically updated over-the-air as well.



Putting it All Together

Using a TinyMLOps stack for conversion and deployment can substantially reduce the engineering hours required. The tighter the constraints on the application, the more difficult the conversion process becomes, as optimizations add complexity. However, new TinyMLOps tooling improves the process and enables TinyML developers to scale their prototypes across thousands of endpoint devices. In this case study, what typically takes years of manual development was reduced to only 4 months of prototyping and testing before deploying deep learning-enabled person, pet and vehicle detection models on over 100,000 smart doorbells.

Additional Resources

- AAAI 2021 paper on DeepliteNeutrino
- Deeplite Neutrino
- PyTorch Dev Day 2021 poster on DLRT
- DeepliteRT Video
- https://github.com/Deeplite

Forum Questions

- What are the factors that impact model conversion?
- How do model conversion concerns impact other stages in TinyML development?
- How does TinyMLOps improve model conversion?
- What are the primary constraints of smart doorbell applications?
- What additional features might be added to improve the usefulness of a smart doorbell?