Application Note

Building an Edge AI Application for Automated Retail Scanner on AM6xA MPUs

Reese Grimsley

ABSTRACT

This application note describes the step-by-step process of building an Edge AI application for vision tasks on TI AM6xA processors, using the AM62A and a retail-scanner demo to provide an example of the process. Following this development flow greatly accelerates new designs. This document details the TI Processor-specific steps for training, compiling, and running a model. Additionally, this document provides information about creating a gstreamer pipeline in Linux that leverages hardware accelerators for image processing such that the developer need not program those accelerators manually.

Table of Contents

List of Figures

List of Tables

Trademarks

1 Introduction

The purpose of this document is to describe the tools and process for creating an Edge AI Vision application for Texas Instruments AM6xA Microprocessors. The exemplary application uses the AM62A for an automated retail scanner or checkout system, in which a customer's tray of food items is quickly and automatically recognized using a deep learning neural network, accelerated by TI's C7xMMA deep learning accelerator.

This application note follows the overall development flow as shown in Figure 1-1, starting from a custom dataset and a partially trained model from the TI Model-Zoo [1]. A neural network model is trained with TI-supported tools and compiled to the target device's C7xMMA architecture. This is then used alongside Gstreamer for developing an end-to-end media pipeline, including camera capture, preprocessing, deep learning inference, post-processing, and display to a monitor. The end application is written in Python3, and open source code can be found online on the Texas Instruments Github under the retail-shopping directory [2]. For instructions on how to run the demo, see the README.md.

This application was developed for AM62A, a quad-core microprocessor from Texas Instruments with 2 TOPS of deep learning acceleration. The model is compiled for this architecture, and will need to be recompiled to run on another TI processor to take full advantage of the accelerator. To learn more about this compilation process, see the related document in TI's Edge AI repo [4] and edgeai-tidl-tools [5]. This application was taken to Embedded World 2023 as a demo to showcase the AM62A.

At the time of implementation, TI's Model Composer suite of software within the Edge AI Cloud [6] was not yet available for this processor – those tools simplify neural network model development from data capture to model selection to training to compilation and evaluation as shown in Figure 1-1 up through Model Evaluation. This document follows a more programmatic design flow, which offers greater flexibility that experienced developers may prefer.

Find the concept video for the demo on youtube at the following link: https://www.youtube.com/watch?v=jYJvtoPAW6E.

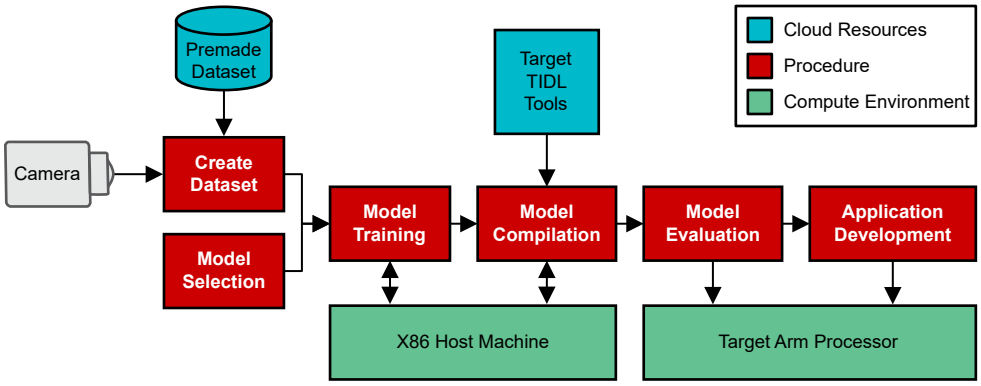

Figure 1-1. Development Flow Diagram

1.1 Intended Audience

This document is intended for readers with beginner or intermediate level deep learning experience, who aim to bring their ideas onto TI processors for vision applications. It is also intended to help readers add practical knowledge to the implementation of such systems; this document intentionally provides additional tips and comments for those with theoretical understanding but less practical expertise.

For those with minimal deep learning knowledge, it is recommended to follow the Fundamental Concepts pages of the Edge AI Academy [4] in the TI Developer Zone.

1.2 Host Machine Information

A host machine, *i.e.,*a laptop or desktop computer, is necessary to handle dataset preparation, neural network training, and neural network compilation. In the making of this application, the host machine was a 64-bit x86 CPU running Ubuntu 18.04 LTS. The machine includes a 512 GB SSD, 16 GB of RAM, and NVIDIA A2000 GPU for training.

Python 3.6 is the dominant programming environment on the host. For guidance on how to setup a Python3 virtual environment with the necessary dependencies, see edgeai-modelmaker.

2 Creating the Dataset

The first step of neural network, a.k.a. "model", creation is to create/curate a dataset. As a data-driven methodology to solving problems, machine learning and deep learning models are only as good as the data these models are trained on. It is ideal to train a model with data that is custom to the final task is designed for.

Public datasets like COCO or Imagenet provide a convenient path to developing and evaluating a deep learning model. There are many of publicly available datasets; many can be found on sites like paperswithcode.com [5]. However, not all public datasets are usable per license terms; they may also have too few high-quality data points. At the same time, custom datasets are time-consuming to create.

In the context of a retail scanner application, the usable and licensable online datasets were not high enough quality to use as-is. Many images were crowd-sourced and poorly labeled, such that trained model had poor performance on both the validation data subset as well as in practice within a well-lit checkout area. Creating a dataset from scratch was necessary.

2.1 Collecting Images

Images should be collected in a similar context to how the model will be used in practice on the target embedded processor. For example, a checkout can have a downward facing camera with bright lighting and a neutral colored background. The space can be (mostly)free of additional objects but possibly showing a user's hands, their wallet, smartphone, and so forth.

The camera used to collect images should be good quality, but does not need to be professional quality such as a photographer would use. Most images used during deep learning are scaled to a smaller resolution (< 1 Megapixel: for example, 512x512) than the original input. It can be advantageous to use the same camera that the end product will use, including the same autogain, white balancing, and other tuning settings a camera may require, but it is not explicitly necessary. Using different cameras or settings can improve robustness later in case the camera must change. The retail-scanner application in this document used the end camera (an IMX219 image sensor) and a 1080p USB camera for image capture.

Within the dataset, the goal is to give a machine learning model enough instances of *patterns* that relate to the problem to solve. For example, what a banana looks like such that it can be added to a customer's order. Irrelevant patterns should not consistently exist in the training samples, such as a banana always being near the edge of the image or against a white background. To improve robustness of the model, it is recommended to vary lighting conditions; object orientations, positions, and angles; camera angle/height; and the versions of the objects themselves (for example, bananas at different stages of ripeness) if characteristics can change. If there is not enough variation, then the training data can be *overfit* such that the neural network is extremely effective at recognizing the training data but fails to generalize to new data. The dataset should reflect the images that the end system will see in practice.

Larger datasets are more effective at training deep neural networks than small ones. Large, *high-quality*, datasets are extremely effective. However, a large dataset is time-consuming to collect and label. Starting with a smaller dataset is helpful for diagnosing types of problems that will require more data. A dataset with as small as 50 instances of each class of item is plenty sufficient for a first attempt. As little as 10 can be used for an initial proof of concept. It is important for these tiny datasets that the trained model has already been trained once on a large dataset like imagenet of COCO, which is the case for models trained using TI tools.

The food-detection dataset uses 100-200 instances of each food item for more robustness. This application was intended to demonstrate device capability. While the model did not need to be production worthy, it did need to perform reliably enough to demonstrate in ad hoc settings with reasonable accuracy. This food dataset included realistic fake versions of food or packaged non-perishable foods to make the demo easier to transport and reproduce in new places. This way, the objects' appearances do not change substantially after training. Figure 2-1 is an example image from the food-detection dataset.

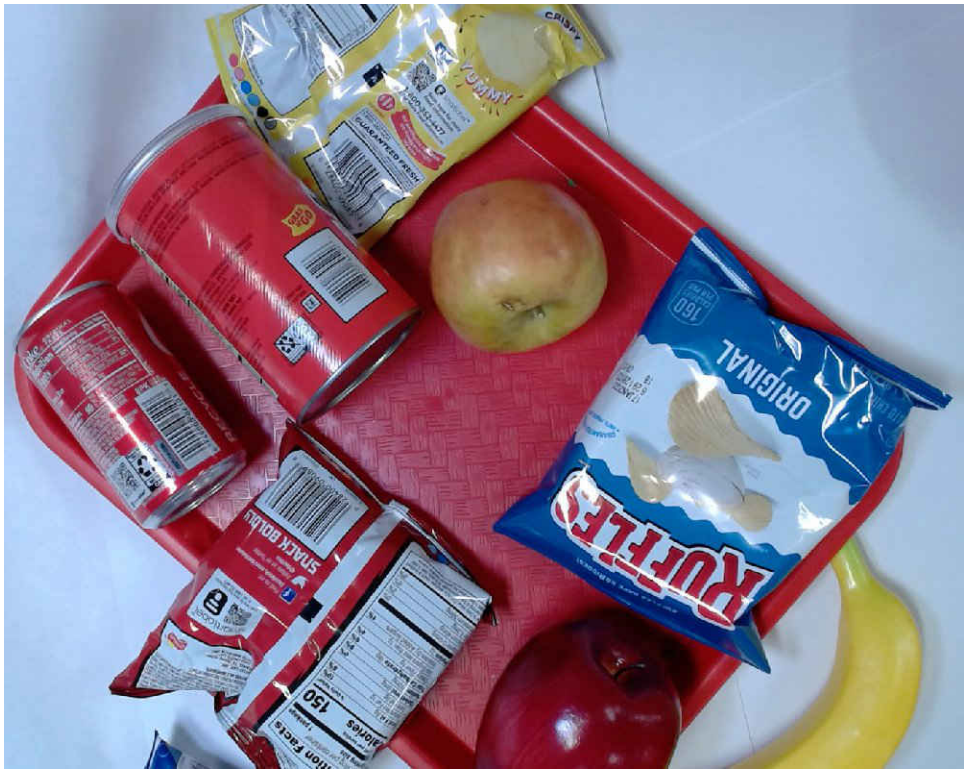

Figure 2-1. An Image From the Food-Recognition Dataset

The dataset can be downloaded from the Github repository for this demo application [2].

2.2 Labelling Images

Deep learning and neural networks are *supervised* machine learning algorithms. They require the data have an associated *label* that informs training what pertinent information is in the data. The type of label depends on the type of problem being solved. An object detection model requires coordinates (typically two 2D points, representing a *bounding* box) and the type of an object; there can be multiple within one image. For the retail-scanner application, this is the right choice because part of the purpose is to recognize multiple things quickly and automatically.

Unfortunately, labelling is generally a tedious task. If done poorly, model accuracy will suffer. Several tools exist to ease this process. TI's Edge AI Studio is an online cloud tool for labeling and most other model development tasks for TI processors; however, this was not available during the development of the retail-checkout application. An alternative tool for offline/local labeling is “label-studio” and the labeling interface is shown in Figure 2-2. This figure shows an image from the food-recognition dataset. Multiple objects are present and have colored boxes drawn onto them to indicate which class they belong to.

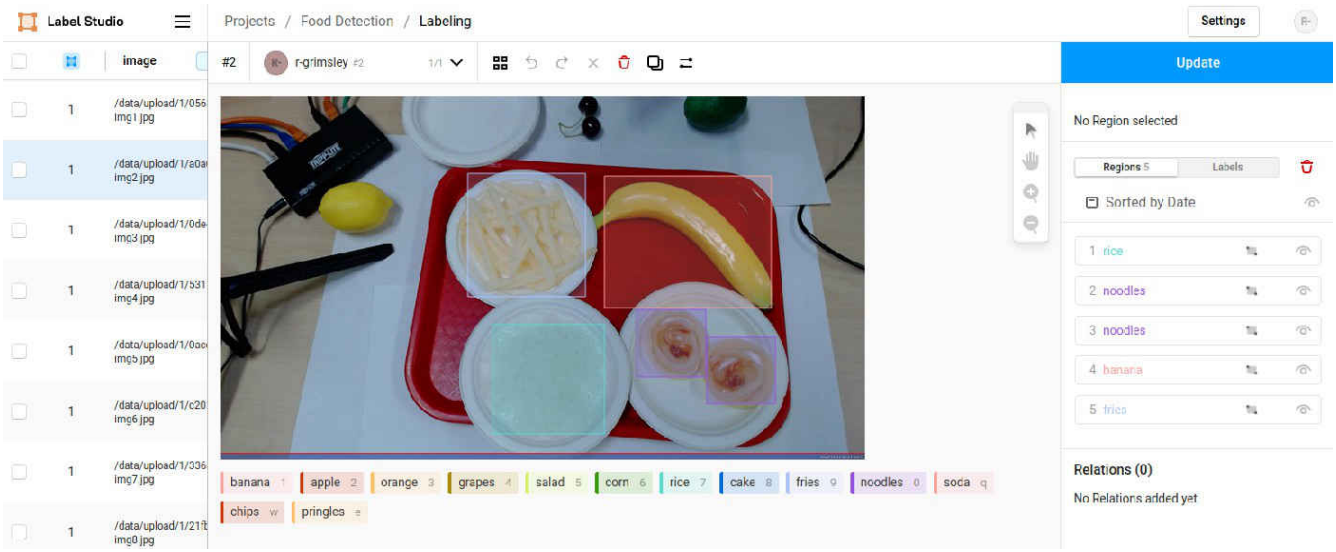

Figure 2-2. Label Studio interface. (Additional objects are intentionally left in the image and unlabeled. Allowing clutter can improve robustness and precision.)

Once all images are labeled, the dataset can be exported in one of several formats. The training tools from TI use COCO JSON format. For this, the output is a ZIP archive with a folder of images and a JSON entitled result.json.

2.3 Augmenting the Dataset (Optional)

"Augmenting a dataset" means to artificially expand it by adding copies of the data with various types of noise and modifications without increasing the data-labelling burden. When done well, augmentation improves model robustness and prevents overfitting. For instance, adding blurring effects can improve robustness when camera focus is not correct; adding rotations to the image can help introduce object orientations that were not part of the original dataset.

If a *train-test split* is being created for the dataset, the split should be done before augmentation so as to not contaminate the testing set. Some augmentations may have little overall impact and thus have a near perfect match between a testing and training image. This would unfairly increase the calculated accuracy of the model.

For the food-recognition model, the list shown in Table 2-1 of augmentations is used.

Table 2-1. List of Augmentations

Size/Orientation transforms	Filters/Localized Effects	Additive/Multiplicative Effects
Perspective Transform	Gaussian blur	Gaussian noise
Flip left-right	Sharpen	Add saturation
Flip up-down	Motion blur	Change color temperature
Shear	Contrast	Multiply hue and saturation
Perspective transform	JPEG compression	
Rotation	Autocontrast	
	Sharpen	

Not all augmentations were applied on every image; a random subset was applied. For each original image, eight additional augmented versions of the image were created. These augmentations were selected based on how likely their effects were to be seen in practice. This is not an exhaustive list of all the available and useful augmentations, but includes more than the standard augmentations of flip, crop, resize, rotate, blur, and add gaussian noise. The imgaug library in Python was used to apply these augmentations and maintain bounding boxes in the process to avoid additional labeling.

Given the type and number of augmentations allowed here, this dataset is considered ‘heavily’ augmented. Heavy augmentation is not recommended for all applications, especially those where small and fine changes in an image could be considered themselves a pattern, for example, defect detection in machine vision applications.

The data_manipulation.py Python3 script within this application's repository includes source code for modifying the images, saving as files, and adding them to the COCO JSON file that describes data labels/annotations. It also includes other functions for reworking the data and labels, for example, combining multiple label-studio training sessions into one large dataset and converting JSON label files into an intermediate format that is easier to manipulate.

3 Selecting a Model

Model selection can happen in parallel with creating a dataset. To get the most performance out of TI’s C7xMMA deep learning accelerator, every layer within the network must be supported on the C7xMMA. Non-supported layers will still work, but may require the CPU runs those layers, which reduces performance.

Models from the TI Model-Zoo all consist of supported layers. TI uses many industry standard and state-of-the-art architectures. In some cases, these architectures are slightly modified to be friendlier to acceleration; these models are referred to as ‘lite’ or ‘ti-lite’ models. Note that at this time of this writing, some otherwise-supported layers (for example, Exponential Linear Unit or ELU) were not supported on the AM62A given its recent launch. This means some models within the model-zoo were not yet supported on the AM62A for acceleration.

TI's Edge AI Cloud "Model Analyzer" is a useful tool for selecting a model. This provides a view of model performance so that developers can select a model or architecture based on metrics like performance (for example, inference speed, memory usage) and accuracy (on a standard dataset like COCO). The performance of a model is independent of the dataset it was trained on, but accuracy of a model is related to the dataset it was trained. When comparing accuracy between models, developers should make sure to only compare accuracy between models trained on the same dataset.

For the food-recognition model built for the retail-scanner demo, mobilenetv2SSD was selected. A pretrained model from TI’s model zoo was used as a starting point for transfer learning. This starting model is named “od-8020_onnxrt_coco_edgeai-mmdet_ssd_mobilenetv2_lite_512x512_20201214_model_onnx” within TI tools. There is plenty of information within the model name:

- od: Object Detection neural network
- "8020": a unique number to distinguish models for faster reference when working with many.
- onnxrt: ONNX runtime is the runtime/API to use for this model
- coco: The original version was trained on the COCO dataset, which contains 80+ everyday objects like people, automobiles, bananas(!), home appliances, and so forth.
- edgeai-mmdet: The training framework was edgeai-mmdet, which is TI’s fork of the open source mmdetection tool from OpenMMLab
- ssd: The *head* or task-specific portion of this network, which is used for object detection, is SSD or Single Shot Detection.
- mobilenetv2: The model architecture is uses MobilenetV2 for feature extraction extraction in the initial set of layers in this model. In machine learning jargon, this is the *spine* of the network.
- lite: This model was slightly modified from the original architecture to be friendlier to TI's C7xMMA architecture
- 512x512: Input images are 512x512 resolution
- 2020214: When this model was originally trained on the dataset (COCO). Not all model names include a date string

4 Training the Model

Once a model architecture is selected and a dataset is created, the next step is to retrain the model on a new dataset.

TI provides a suite of easy-to-use tools for model training and compilation. Any training framework can be used to train the deep learning model, so long as it only contains layers supported on the device's accelerator. Experts may wish to use tools they are more familiar with, such as training with PyTorch and exporting to ONNX for compilation. Less experienced users can use TI's tools like Model Composer and edgeai-modelmaker. The steps covered here use edgeai-modelmaker. An advantage to using TI tools is that compilation can be handled automatically at the end of training, such that Section 5 can be entirely skipped.

Edgeai-modelmaker, once setup/installed, uses a separate training framework like edgeai-mmdetection to perform the training itself. This starts from a pretrained model and fine-tunes via transfer learning by resetting the last layer of the network and using a low learning rate. For models supported in model zoo, this downloads the pretrained weights. Note that when using this process, layers before the last layer are *not* frozen and will change during training.

To train a model, the steps are as follows; refer to modelmaker READMEs for the most up-to-date instructions:

- Setup the modelmaker repository, which includes setting up other training frameworks and TI tools. They are held in the same parent directory. There is a setup_all.sh script in the edgeai-modelmaker repo that handles this, including setup for other training frameworks.
  - A virtual python environment using Python 3.6 is recommended.
- Place the training samples / files into "edgeai-modelmaker/data/projects/PROJECT_NAME" .
  - Run modelmaker on one of the example config YAML files to see the structure of these directories. It follows the COCO format with an "annotations" directory (containing one "instances.json" file in COCO format) and a directory of images.
  - The images must all be in the same directory (called images) without any subdirectories. The training framework assumes all image files to be in the same “images” directory.
- Create a config file that points to the project data, selects the model, training parameters, and so forth. For more information, see config_detect_foods.yaml for an example.
- Run the starter script "run_modelmaker.sh" using the config file above as the first argument.
  - See modelmaker’s README for how to set the device target, since processors like the AM62A includes a different C7xMMA deep learning accelerator such that the compilation tools for that target architecture are different that those of TDA4VM, AM68A, and so forth. It relies on a TIDL_TOOLS_PATH environment variable, which the SH script will set if not predefined.

If a GPU is present on the training machine, it is highly recommended to configure it for training, as it provides substantial speedup. The most important component of configuration is ensuring that the CUDA version is consistent with the version Pytorch/torchvision is compiled against. The one shipped with modelmaker at the time of this writing is CUDA 11.3. Correctly setting up these drivers can be a pain point, as there is a relationship between the CUDA version and the NVIDIA display/graphics driver.

If the dataset is fairly small (<10000 images), it is good to start from a network that is pretrained on a large, generic dataset like imagenet1k for image classification or COCO for object detection. Larger networks generally require more samples for training due to the increased number of parameters

For the food-recognition dataset of ~2500 images (after train-test split and augmentations), training took approximately 1.5 hrs on an A2000 GPU (6 GB VRAM) with 12-core CPU, 16GB RAM, and SSD for 30 epochs of training. This dataset reached a mean average precision (mAP; a common object detection accuracy metric) score of 0.68, which is extremely high. This stems from two facts:

- The validation set used by the training framework automatically included post-augmented files, so some validation data was unfairly similar to some training samples, boosting reported accuracy.
- The environment/background is highly controlled to provide consistency. The model performs well within the limited context, but may not generalize well to a totally new setting, such as trying to recognize the objects against a green or blue background. This may or may not important, depending on the use case. [1]

A full evaluation on the final test set was not performed for the retail-scanner model. Rather, a visual inspection was done to assert the model performed reasonably well before moving on to the next stage. Several

iterations of training were performed by varying the number of epochs, degree of augmentation, and variety of augmentations.

4.1 Input Optimization (Optional)

There is an optional parameter in the training YAML configuration file for 'input_optimization' that is related to quantization and compilation. In short, it folds two preprocessing steps into the first few layers of the model to save CPU cycles and DDR bandwidth. Context for this is provided here, as well as necessary background on model quantization, which is a crucial step of compiling the model.

Models are almost universally trained in floating point format for model weights and values. They can be trained with 'quantization awareness' (QAT or quantization aware training) but weights will still be floating point; weights can also be quantized after training (PTQ or post-training quantization). PTQ converts the model into a fixed-point format that neural network accelerators are often optimized for. For TI's architecture, part of compilation is to create a quantization function that maps float32's to integers (8 or 16 bit) – this is sometimes referred to as 'calibration'. This function is universal across the network, so it is important that all float32s are in a similar range so that they map consistently to fixed point without "clipping" very high or low values to the edges of the limited integer range [2].

A common step for quantization is to limit clipping when mapping floats to integers by keeping weights and values in a range between -1 to 1 or 0 to 1. This is made easier in interior layers by using regularization during training. For the model input, images are often in an 8-bit RGB format and are converted to float32 by subtracting a mean and multiplying by a scaler such that values entering the model are in this smaller, consistent range. This is a necessary preprocessing step for the model, but it is performed inefficiently on the CPU.

"Input optimization" is a strategy for making this preprocessing more efficient by including it in the model. The subtract and multiply operations are moved into the model itself by creating two additional layers at the input to do these element-wise operations on the C7xMMA. This reduces CPU usage and DDR usage because the original 8/16-bit integers for RGB pixels can be provided as is, rather than the preprocessed 32-bit floats.

This input_optimization step can also pulled into the model as part of compilation.

5 Compiling the Model

As mentioned in the previous section, compilation can be handled automatically if using edgeai-modelmaker for training. If a model needs compilation on its own, or if the user wishes to be more involved in the compilation and calibration then they can refer to this section. Edgeai-tidl-tools hosts code that serves this purpose. Model compilation must be done on an x86 host machine.

The edgeai-tidl-tools repository offers several options for model compilation, such as Jupyter notebooks and python3 scripts. There are instructions in that repository for compiling custom models. The process using Jupyter notebook can be done on Edge AI Cloud as well using the tools for custom models. Jupyter [7] notebooks are a great tool to show and develop code step-by-step in a modularized form, but the Jupyter kernel can be unstable on long-running or memory-intensive tasks. The Python3 scripts were used instead for the mobilenetv2SSD model compilation due to better runtime stability.

To compile a model, a set of images is passed through the runtime of choice for the model (ONNX for the food-recognition model), during which a TI Deep Learning compilation backend analyzes the intermediate values and formats. As this script runs, "artifacts" are generated in the form of compiled binary files for running the model on the C7xMMA. This includes an "io" file for how CPUs pass information to/from the C7xMMA and a "net" file which encodes the network layers itself.

In this process, the set of images are used to *calibrate* the model for quantization from native 32-bit floating points to fixed-point integers (8 or 16-bit). The tools analyze the range of intermediate values within the model during execution and choose a quantization scheme the best matches that range. Using a larger set of images that better characterizes the breadth of the input scenes are encountered in practice can improve the quality of quantization, but takes longer.

[1] In a retail scanner application, this is ostensibly unimportant given the typically controlled environment. However, a bad actor may attempt to trick the system bringing an oddly colored background, such as a purple piece of paper, such that items on it are difficult to recognize. New backgrounds can be substituted in as another form of augmentation.

[2] Hybrid networks that use more than one level of quantization like 8-bit on some layers and 16-bit on others will have more than one quantization function. Nonetheless, keeping weights in the same range throughout the network will reduce the amount of clipping.

The python scripts underedgeai-tidl-tools/examples/osrt_python/ can handle all steps of model compilation, but are less clear on what is happening with preprocessing, configuration, *etc*. than the more interactive Jupyter notebooks. The onnxrt_ep.py and model_configs.py files required a few changes for compilation to complete:

- There is a list of models to compile. For the food-recognition model, this included 'od-8020_onnxrt_coco_edgeai-mmdet_ssd_mobilenetv2_lite_512x512_20201214_model_onnx'.
- The configuration for the model is held in a large python dictionary from model_configs.py in the above directory. The key is the string from the previous bullet.
  - "model_path" must point to the trained model from the custom dataset.
  - “meta_layers_names_list” must point to the PROTOTXT file path that describes the model architecture.
  - If the two above files are not present, they are downloaded, assuming it is a model architecture that is already supported and has a download at the provided URL.

The TIDL_TOOLS_PATH environment variable in the calling environment should be pointing to the tidl_tools for the target processor. By default, it points to the tools downloaded when setting up edgeai-tidl-tools, during which an SOC variable should have been set for the device to compile to for, *e.g.* SOC=am62a.

Once complete, compiled artifacts are in a directory with the model's full name under edgeai-tidl-tools/model-artifacts.

One last (optional) item before continuing to using the model in practice: getting class labels aligned between model output and text name. This is only necessary if using edgeai-gst-apps or the tidlpostprocess GST plugin, as they attempt to write text for the recognized object(s) on the image. When modelmaker attempts to compile a model, it creates a dataset.yaml file to describe the dataset, which includes mapping the output class integer to a text label.. This is retrieved from the COCO JSON file that holds labels. The param.yaml file needs to also provide a mapping (key [metric][label_offset_pred]) from the output of the model to the index in dataset.yaml. For the food-recognition model, it is one-to-one since all classes trained on are used. The dataset.yaml is automatically generated by TI’s training frameworks, but the param.yaml may require manual update.

6 Using the Model

The next step is to use the model in practice.

The accuracy reported during training is helpful for determining the effectiveness of the model, but visualizing on realistic input is crucial to be confident the model is performing as expected.

The fastest way to evaluate a model for new input on the target device is to use it within edgeai-gst-apps. This is a valuable proof-of-concept for evaluating more practical accuracy without writing new code. Copy the new directory within “compiled-artifacts” onto the target device and modify a config file like object_detection.yaml (shown in Figure 6-1) to point to this model directory. Ensure that the model is used in the flow at the bottom of the config file. The input can either be a live input like USB camera or a premade video/directory of image files.

```
title: "Food Recognition"
log_level: 2
inputs:
    input0: # 720p USB camera, e.g. Logitech c270 or similar with that resolution capability
        source: /dev/video2
        format: jpeg
        width: 1280
        height: 720
        framerate: 30
    input1: #IMX219 with default settings. Run camera setup script under scripts
        source: /dev/video2
        width: 1920
        height: 1080
        format: rggb
        subdev-id: 2
        framerate: 30
    input2: #IMX219 w/ 1640x1232 RGGB10. Run camera setup script in retail-shopping
        source: /dev/video2
        width: 1640
        height: 1232
        format: rggb10
        subdev-id: 2
        framerate: 30

models:
    model0:
        model_path: /home/root/edgeai-gst-apps-retail-checkout/retail-shopping/food-detection-model-mobv2ssd/
        viz_threshold: 0.6

outputs:
    output0:
        sink: kmssink
        width: 1920
        height: 1080
        overlay-performance: True


flows:
    flow0: [input0,model0,output0,[320,180,1280,720]]
```

Figure 6-1. Example of edgeai-gst-apps config File

A connected display/saved video file looks like Figure 6-2 (or may have performance information overlaid depending on how the output is configured):

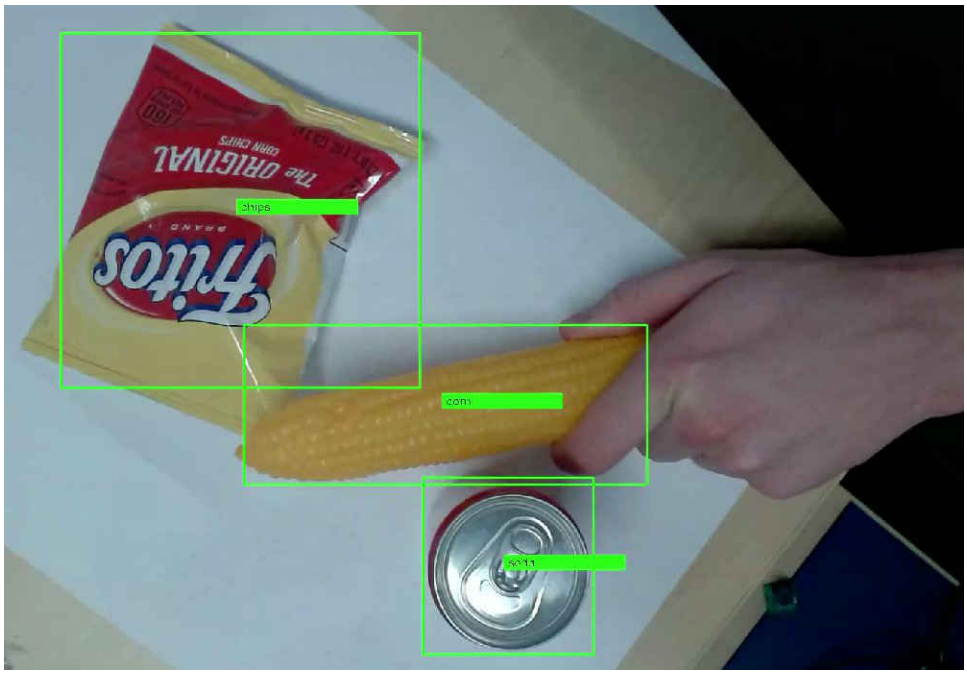

Figure 6-2. Gstreamer Display When Using edgeai-gst-apps for Newly Trained Model

An additional benefit of running the model in this way is that a holistic gstreamer string is printed to the terminal, which is a useful starting point for application development.

7 Building the End Application

Once the model is confirmed to work, an end-to-end application can be built around it.

An end-to-end application means live input from a camera, preprocessing, running the model itself, and doing *something* with the output suited to a real-life use-case. In the retail scanner use-case, that *something* is using the identified objects to populate and display an order for the customer.

To build a data processing pipeline in Linux, gstreamer (GST) is recommended. With the GST plugins used across edgeai-gst-apps, everything but the application code can be done purely with a GST pipeline. GST pipelines allow different stages of processing to happen in a parallel streaming fashion, which improves overall performance as compared to a single-threaded program. TI offers plugins to interact with hardware accelerators for image processing and deep learning. For the end application, GST can use *appsink* and *appsrc* plugins to expose an interface into and out of application code, respectively.

In the retail-scanner application (see gst_configs.py), a reference to an appsink is retrieved and used to “pull samples” that hold a raw byte array, representing the chunk of data. This might be an image, a chunk of audio, or the output tensor of a neural network. The structure of the data needs to be known before-hand, such as dimensions and pixel-format. For typical data like images (in GST terms, raw/x-video), the “caps” which describe input and output of a GST plugin give information about the resolution and pixel format.

When first designing the retail scanner demo application, it was most straight-forward to pull input frames from the camera in RGB format at full resolution from the camera. Python application code handled preprocessing, inference, postprocessing, and adding a receipt-like image to the final frame to display. The application borrowed from the main apps_python code, but extra image processing within display.py was slow and had highly variable latency. This worked for flushing all the components and testing functionality, but performance was unacceptable for an interactive application at about 5-6 fps with up to 3 seconds latency. The mobilenetv2SSD model recognizing foods was plenty fast at >60-fps, but the application code on Arm CPU was too slow to keep up with model inference or the 30-fps camera.

7.1 Optimizing the Application With TI’s Gstreamer Plugins

Streaming applications strongly benefit from pipelining such that each component of the application runs in a separate process. This is effective when hardware accelerators and multiple CPU cores are available to allow concurrency. TI’s gstreamer plugins are optimized to leverage the heterogeneous architecture of AM6xA processors and are implemented with zero-buffer copy to reduce memory overhead.

The OpTIFlow plugins (*i.e.,* tiovxdlpreproc, tidlinferer, and tidlpostproc) are instrumental in effective pipelining of an AI application since they allow preprocessing, inference, and postprocessing to happen in parallel. Using the OpTIFlow plugins, the only function that needs to run in dedicated application code is making use of the deep learning output. In the retail-scanner application, this means creating a small image that lists the customer’s order. Figure 7-1 illustrates this pipeline. For the exact gstreamer pipeline and a detailed explanation, see the HOW_ITS_MADE.md readme.

With this pipeline on AM62A (Processor SDK 8.6, E2 revision of the starter-kit EVM), the framerate is 15-18 fps. The main bottleneck is Python application code, as this includes additional memory copies and multiple text-drawing function calls with OpenCV. The deep learning accelerator is loaded 25-30%, quad A53 CPU at 30-40% (avg. across all cores), and 32-bit 3200 MT/s LPDDR4 at 20% load.

Optimizing the pipeline to better offload different tasks of the application represented +200% performance boost in terms of FPS and order of magnitude reduction in latency. Much more can be done to optimize, such as developing in C/C++, creating a more efficient function for writing all text to the frame at once (or removing that portion of the application altogether), or even converting application code into a GST plugin.

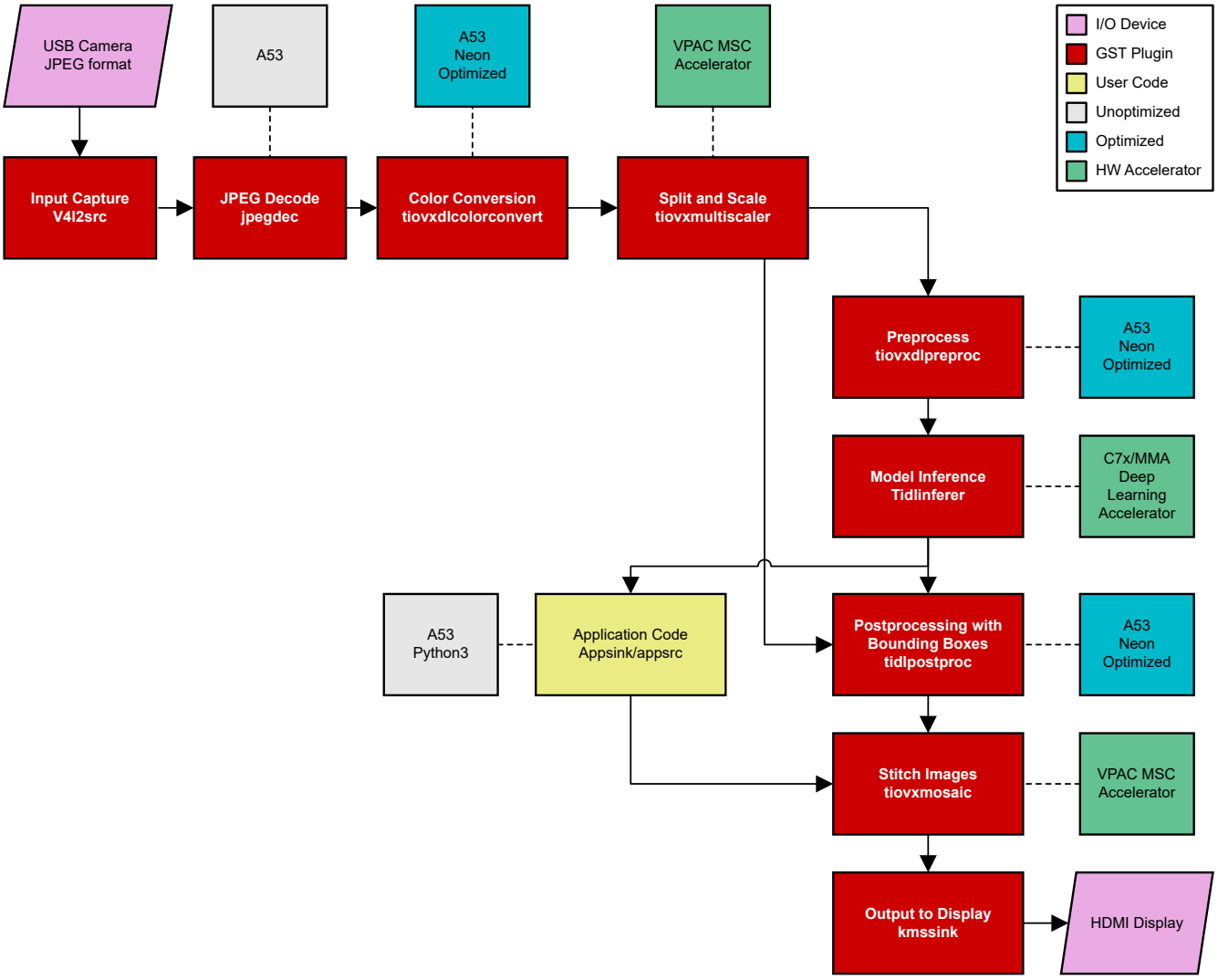

Figure 7-1. Block Diagram of gstreamer pipeline for the Retail-Scanner Application

7.2 Using a Raw MIPI-CSI2 Camera

TI's Edge AI Processors, e.g. AM62A, TDA4VM, AM68A, include an ISP and additional hardware accelerators for scaling images (multiscaler MSC) and dewarping distortions from wide-angle lens (Lens Distortion Correction LDC). The ISP, MSC, and LDC are all TI-designed IP within the Vision Processor Accelerator (VPAC). This section will describe how an IMX219 image sensor was used for this application.

Note

The retail-scanner demo also works with USB cameras in 720 and 1080p. When running, use the -c tag with either usb-720p or usb-1080p for 720 or 1080p USB input, resp. The C270 and C920 logitech USB cameras have both been verified to work. The version of the GST pipeline uses USB input to simplify until this point in the document.

An ISP (Image Signal Processor) is necessary to take the raw output of an image sensor and process it into a more typical format like RGB. Raw output sensors are cheaper and external ISP's (including integrated ones like TI's) are more readily tunable than ISP's built into the image sensor. The ISP can be tuned based on lighting conditions, lens effects, auto-exposure, and so forth. TI has a tuning tool for the integrated ISP and supporting documentation.

The IMX219 is a low-cost sensor that is supported out-of-box, and can capture at up to 8 MP at 15 fps or 2 MP at 30 fps (2MP at 60fps is possible but requires a linux patch that is linked in SDK documentation).

Supporting a sensor includes several components, all three of which are available for the IMX219:

- A driver, ideally on that is upstreamed to the mainline Linx kernel and actively maintained
- ISP tuning
- A hardware module for evaluation

The retail-scanner demo required a 2-megapixel input at 30 fps and at least 70° field-of-view. Using this particular sensor posed several challenges for this use case; these challenges and their resolutions are described below:

- The driver has limited support for different resolutions and formats
  - The typical 16:9 aspect ratio 1920x1080 (2MP) resolution does not have full field of view – it is a cropping the full 8MP frame, and thus appears artificially zoomed in. The result is far below the 70° FOV requirement.
  - Full FOV with 4:3 aspect ratio is possible with 1640x1232 (half resolution in width and height; still 2 MP). However, the upstream driver only works as-is with 10-bit pixel values (RGGB10). This mode is used for the retail-scanner application, and the setup_cameras.sh script within the demo repository is modified to set this configuration using the linux "media-ctl" command-line tool.
  - 60 fps is not supported at 2MP resolutions - it runs at 30 fps. This is a result of the upstream driver. The Edge AI processors from TI are capable of full 60 fps for this sensor, but it requires a patch to the linux kernel, rebuild for the driver module, and installation on the target.
- Different resolutions may require changes to the ISP files (typically under /opt/imaging/imx219) – this required the "DCC" files to be regenerated. That process is not covered in this document.
  - The 8.6 Edge AI SDK now includes ISP configuration files for all resolution and bit-depths that are possible with the upstream IMX219 driver.

8 Summary

This document described the step-by-step process of building an Edge AI application for vision tasks on TI AM6xA processors, using the AM62A and a retail-scanner demo to provide an example of the process. Following this development flow can greatly accelerate new designs. This document has detailed the TI Processor-specific steps for training, compiling, and running a model, as well as creating a gstreamer pipeline in Linux that leverages hardware accelerators for image processing, without requiring the user to program those accelerators manually. References are provided to point developers to the tools, code, and further guidance to develop the applications that leverage the complexity of the processor without introducing additional design complexity.

9 References

1. Texas Instruments, "Edge AI Marketplace - Retail Checkout," April 2023. [Online]. Available: https://github.com/TexasInstruments/edgeai-gst-apps-retail-checkout.
2. Texas Instruments, "edgeai-gst-apps-retail-checkout," [Online]. Available: https://github.com/TexasInstruments/edgeai-gst-apps-retail-checkout/blob/main/retail-shopping/README.md. [Accessed 18 4 2023].
3. Texas Instruments, "TI Edge AI - Model Development," [Online]. Available: https://github.com/TexasInstruments/edgeai/blob/master/readme_models.md.
4. Texas Instruments, "Edge AI Academy," [Online]. Available: https://dev.ti.com/tirex/explore/node?node=A__ABufwRlVEGy83u-FZ.kg5Q__EDGEAI-ACADEMY__ZKnFr2N__LATEST.
5. "Papers with code - Image Datasets," [Online]. Available: https://paperswithcode.com/datasets?mod=images.
6. Texas Instruments, "Edge AI Studio," [Online]. Available: https://dev.ti.com/edgeaistudio/.
7. "Jupyter Notebook," [Online]. Available: https://jupyter.org/.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265