Porting ML models to custom AI HW accelerators

DeGirum's PySDK can help developers who wish to port ML models to custom HW

Upcoming hardware (HW) accelerators for edge artificial intelligence (AI) applications promise unprecedented efficiency in power, price, and performance. HW vendors generally couple their HW with software (SW) tools, illustrating applications that run out-of-the-box along with a collection of ready to use machine learning (ML) models.

While such models are sufficient to get started with the development process, application developers eventually want to port their own ML models that are customized and fine-tuned for their use cases. Porting ML models to custom HW poses a lot of challenges to the developers, primarily because the ML engineers involved in the model training might not be aware of the limitations of all the HW options available out there. The fact that not all HW can run all models is exacerbated by the rapid progress in model development where a new state-of-the-art (SOTA) is established every week.

The ML application pipeline

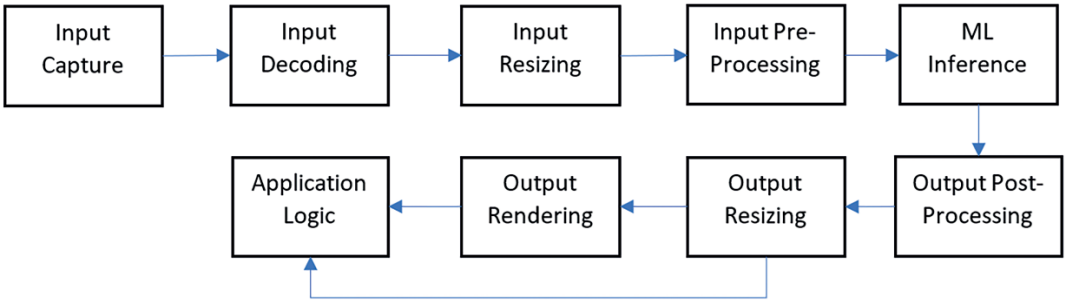
AI HW vendors typically provide model porting tools that convert trained ML models provided in the form of protobuffers or flatbuffers (ONNX, TFLite, etc.) to binaries that can be executed on the HW. They also provide a runtime that executes these model binaries. However, tools to compile a model and run the model binary alone are not sufficient for integrating the ML models to application SW. This is because the pipeline for an AI application is so much more than just ML model inference.

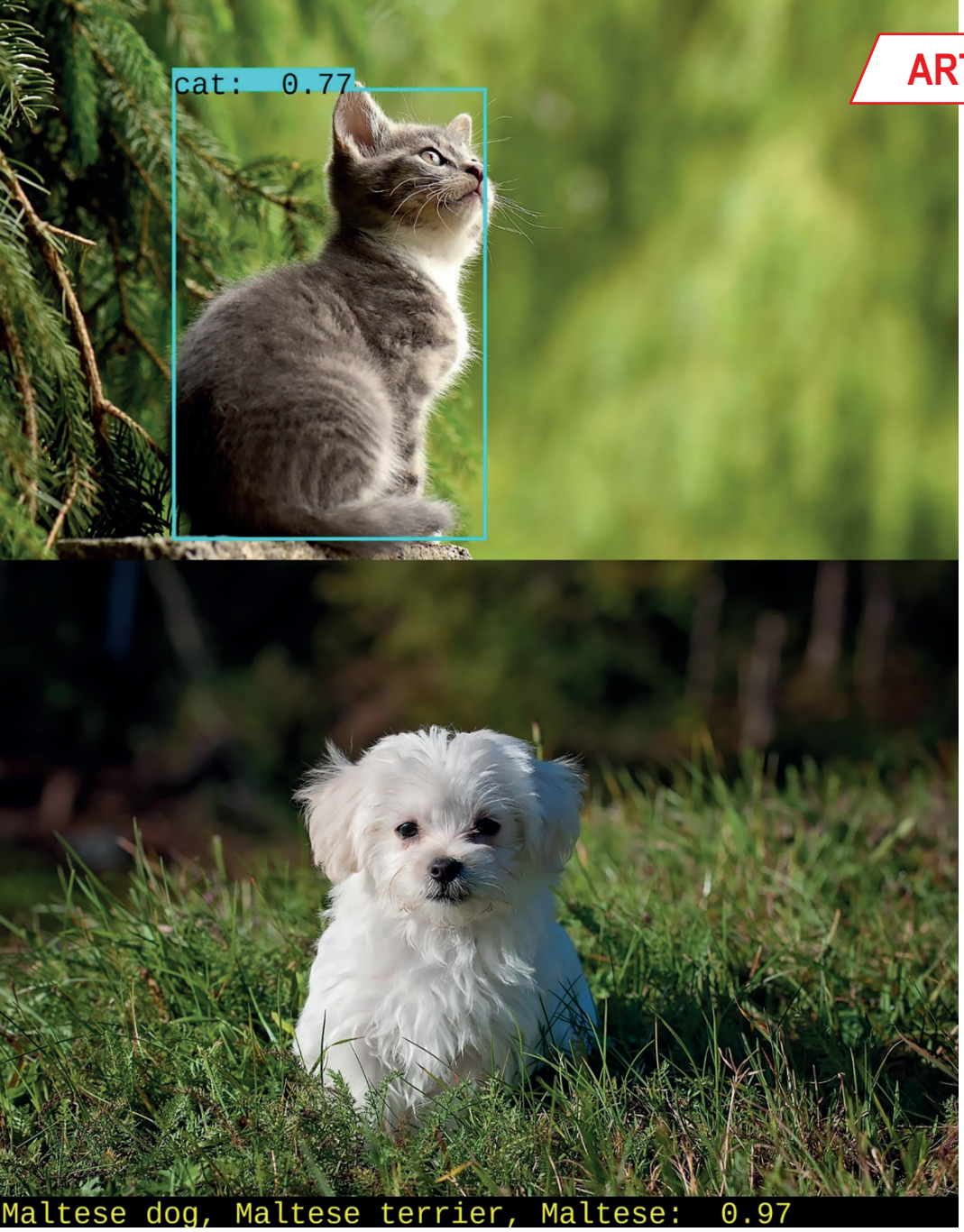
The first step in an AI application is capturing the input (image, voice, text) from the source. In the case of computer vision (CV) applications, the typical sources of input are cameras. Since cameras are used in a wide variety of applications, such as surveillance, quality control, and machine vision to name but a few, the format of the input varies depending on the type of the camera and the application. The incoming input stream can be encoded in H.264 or H.265 or MJPEG formats. Additionally, the resolution of the input can take one of the many possible values, such as VGA (640x480), 720p (1280x720), 1080p (1920x1080) and so on. Hence, the input needs to be decoded before it can be further processed by the ML models.

ML models are trained at some resolution (generally with square sizes such as 224x224, 416x416, and 640x640) which may not match the input resolution of the camera used in the application. The data used for model training comes from different sources and contains images of different sizes. Using square images while training allows the trained models to address different scenarios, as models can be trained at one resolution and deployed at a different

Shashi Kiran Chilappagari, Co-Founder and Chief Architect DeGirum Corp

A typical ML pipeline for a CV application





Visualizing outputs for (a) An object detection model, and (b) an image classification model

resolution. However, this means that the decoded input needs to be resized to the resolution at which the ML model is deployed. The inputs are often normalized before running the inference.

Different ML models provide different information about the input image. Classification models provide a list of most likely labels, whereas detection models provide labels for the objects as well as their location in the image. The location information is provided in the form of bounding boxes. This information is not the direct output of the model; it is obtained by processing the output of the model using methods specific to the model type. Classification models involve sorting the probabilities of the labels to find the most likely labels, whereas detection models need to decode bounding boxes and run a non-max suppression (NMS) algorithm to weed out extra overlapping predictions. The final predictions need to be resized to the original input size and then rendered for visual inspection or sent upstream so that application-specific logic can be applied to the output and proper action can be taken.

Challenges in model porting

Such a deep pipeline poses several challenges to the application developers, some of which are as follows.

#1 Replicating Model Accuracy: Developers want to ensure that the accuracy of the originally trained ML model is retained after porting it to custom HW. The model may undergo several optimizations before being compiled for the HW, and the developers need to be confident that such optimizations do not lead to loss of accuracy. However, replicating model accuracy after porting is a challenging task as the accuracy depends on a lot of implementation details listed below:

- Image Backend: The image processing library used to process the inputs. Examples include popular packages such as OpenCV and PIL.
- Interpolation Options: Options used for interpolating images when resizing the images to the size expected by the ML model. Most common options used include (a) nearest neighbor, (b) bilinear, (c) area, (d) bicubic, and (e) lanczos.
- Resize Options: Parameter that specifies how the image is resized to the size expected by the ML model and how the aspect ratio is handled. Common options include (a) stretching, (b) letterboxing, (c) cropping and then resizing, and (d) resizing and then cropping.
- Normalization: In some ML models, the input is normalized using per channel mean and standard deviation parameters before being sent to the ML inference
- Quantization: Model quantization can lead to smaller model size and faster inference (especially on custom HW). However, quantization can impact the model performance.

Developers need a SW stack that can take all these parameters into account to evaluate

the performance of the ported model.

#2 Visualizing model output: To verify that an ML model has been ported successfully, developers need SW to visualize the output of the ML model on some sample inputs. Since different types of models provide different information, a lot of boilerplate code is needed to visualize the model output. Image classification models provide top labels for the image, object detection models provide bounding box information for various detected objects in the image, pose detection models provide key-point information, and sematic segmentation models provide per-pixel class label information. Rendering these different types of information requires developing code specific to handling these cases.

#3 Handling multiple HW options: Consider a scenario where an application developer wants to evaluate multiple HW options and choose the HW best suited for their use case. In such a scenario, they have to develop multiple SW stacks, each one addressing a different HW option. This is due to the fact that the model porting tools—along with the inference runtime libraries—are HW specific. Once they've developed the application, they need to evaluate all the options and then pick the right HW. This leads to a lot of wasted time, effort, and money as bringing up custom HW is a long, expensive, and frustrating experience.

#4 Optimizing Performance: Employing a multistage pipeline in the application has implications on the overall performance as each stage needs to be pipelined efficiently. Different stages may utilize different HW resources (such as video decoders, ML accelerators. etc.), which further complicates the optimization process. In order to utilize the AI HW accelerator resources efficiently, developers need to carefully analyze the various stages and orchestrate the application's execution so that the final application is stable and efficient. Executing each stage of the pipeline in a single synchronous thread will lead to high latency and poor utilization of HW resources.

DeGirum's PySDK

DeGirum's python software development kit (PySDK) is specifically designed to address the challenges faced by application developers. Instead of just providing an inference runtime library that focuses on running the ML model inference, the PySDK provides a simple Model Predict API that handles the pre-processing of the input (including resizing and normalization), the ML inference, and the post-processing of the output (including rendering predictions on original image).

The PySDK also provides a highly efficient Batch Predict API that pipelines all stages of the application, including input capture and decoding as well as running the application logic. The PySDK manages all the ML inference calls to the AI HW and ensures that these calls are scheduled in such a way as to maximize the HW usage. Other highlights of the PySDK include:

• A single JSON file that specifies all the parameters related to input pre-processing, ML inference, and output post-processing. This greatly helps the model developers to replicate model accuracy benchmarks on the ported model.
• A Model Predict API that handles overlaying predictions for different model types such as image classification, object detection, key-point detection, and sematic segmentation. This feature obviates the need to develop boilerplate code for various common use case.
• HW agnostic APIs that ensure that the same code works for multiple HW options. This allows developers to create a unified SW stack that can be used to evaluate multiple HW options and pick the best suited option.

The simplicity of the Model Predict API can be illustrated using the code snippet below:

"DeGirum's python software development kit (PySDK) is specifically designed to address the challenges faced by application developers"

```
import degirum as dg
zoo=dg.connect_model_zoo()
model=zoo.load_model('yolov5_n2x_orca')
res=model(image)
res.image_overlay
```

This code illustrates a YOLOv5 model running on DeGirum's ORCA HW. By changing the model name, the same code can be used to run other types of models. Also, the same code can be used to run across different HW options.

Porting ML models to custom HW poses a lot of challenges to the developers, but DeGirum's PySDK can help smooth the way.

www.degirum.com