Bring Your Own Model (BYOM)

Table of contents

List of Figures

The Holoscan platform is optimized for performing AI inferencing workflows. This section shows how the user can easily modify the `bring_your_own_model` example to create their own AI applications.

In this example we'll cover:

- the usage of `FormatConverterOp`, `InferenceOp`, `SegmentationPostprocessorOp` operators to add AI inference into the workflow
- how to modify the existing code in this example to create an ultrasound segmentation application to visualize the results from a spinal scoliosis segmentation model

Note

The example source code and run instructions can be found in the examples directory on GitHub, or under `/opt/nvidia/holoscan/examples` in the NGC container and the debian package, alongside their executables.

Operators and Workflow

Here is the diagram of the operators and workflow used in the byom.py example.

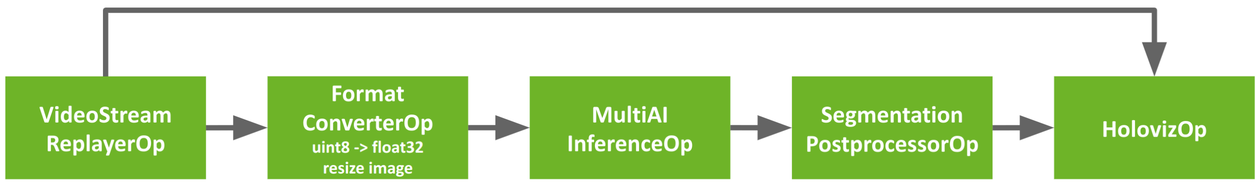

Fig. 10 *The BYOM inference workflow*

The example code already contains the plumbing required to create the pipeline above where the video is loaded by `VideoStreamReplayer` and passed to two branches. The first branch goes directly to `Holoviz` to display the original video. The second branch in this workflow goes through AI inferencing and can be used to generate overlays such as bounding boxes, segmentation masks, or text to add additional information.

This second branch has three operators we haven't yet encountered.

- **Format Converter**: The input video stream goes through a preprocessing stage to convert the tensors to the appropriate shape/format before being fed into the AI model. It is used here to convert the datatype of the image from `uint8` to `float32` and resized to match the model's expectations.

- **Inference**: This operator performs AI inferencing on the input video stream with the provided model. It supports inferencing of multiple input video streams and models.

- **Segmentation Postprocessor**: this postprocessing stage takes the output of inference, either with the final softmax layer (multiclass) or sigmoid (2-class), and emits a tensor with `uint8` values that contain the highest probability class index. The output of the segmentation postprocessor is then fed into the Holoviz visualizer to create the overlay.

Prerequisites

To follow along this example, you can download the ultrasound dataset with the following commands:

> $ wget --content-disposition \ https:*//api.ngc.nvidia.com/v2/resources/nvidia/clara-holoscan/holoscan_ultrasound_sample_data/versions/20220608/zip \ -O holoscan_ultrasound_sample_data_20220608.zip* $ unzip holoscan_ultrasound_sample_data_20220608.zip -d <SDK_ROOT>/data/ultrasound_segmentation

You can also follow along using your own dataset by adjusting the operator parameters based on your input video and model, and converting your video and model to a format that is understood by Holoscan.

Input video

The video stream replayer supports reading video files that are encoded as gxf entities. These files are provided with the ultrasound dataset as the `ultrasound_256x256.gxf_entities` and `ultrasound_256x256.gxf_index` files.

> **Note**
>
> To use your own video data, you can use the `convert_video_to_gxf_entities.py` script (installed in `/opt/nvidia/holoscan/bin` or on GitHub) to encode your video. Note that - using this script - the metadata in the generated GXF tensor files will indicate that the data should be copied to the GPU on read.

Input model

Currently, the inference operators in Holoscan are able to load ONNX models, or TensorRT engine files built for the GPU architecture on which you will be running the model. TensorRT engines are automatically generated from ONNX by the operators when the applications run.

If you are converting your model from PyTorch to ONNX, chances are your input is NCHW and will need to be converted to NHWC. We provide an example transformation script named `graph_surgeon.py`, installed in `/opt/nvidia/holoscan/bin` or available on GitHub. You may need to modify the dimensions as needed before modifying your model.

> **Tip**
>
> To get a better understanding of your model, and if this step is necessary, websites such as netron.app can be used.

Understanding the Application Code

Before modifying the application, let’s look at the existing code to get a better understanding of how it works.

Ingested Tab Module

Next, we look at the operators and their parameters defined in the application yaml file.

Ingested Tab Module

Finally, we define the application and workflow.

Ingested Tab Module

Modifying the Application for Ultrasound Segmentation

To create the ultrasound segmentation application, we need to swap out the input video and model to use the ultrasound files, and adjust the parameters to ensure the input video is resized correctly to the model’s expectations.

We will need to modify the python and yaml files to change our application to the ultrasound segmentation application.

Ingested Tab Module

The above changes are enough to update the byom example to the ultrasound segmentation application.

In general, when deploying your own AI models, you will need to consider the operators in the second branch. This example uses a pretty typical AI workflow:

- **Input**: This could be a video on disk, an input stream from a capture device, or other data stream.
- **Preprocessing**: You may need to preprocess the input stream to convert tensors into the shape and format that is expected by your AI model (e.g., converting datatype and resizing).
- **Inference**: Your model will need to be in onnx or trt format.
- **Postprocessing**: An operator that postprocesses the output of the model to a format that can be readily used by downstream operators.

- **Output**: The postprocessed stream can be displayed or used by other downstream operators.

The Holoscan SDK comes with a number of built-in operators that you can use to configure your own workflow. If needed, you can write your own custom operators or visit Holohub for additional implementations and ideas for operators.

Running the Application

After modifying the application as instructed above, running the application should bring up the ultrasound video with a segmentation mask overlay similar to the image below.

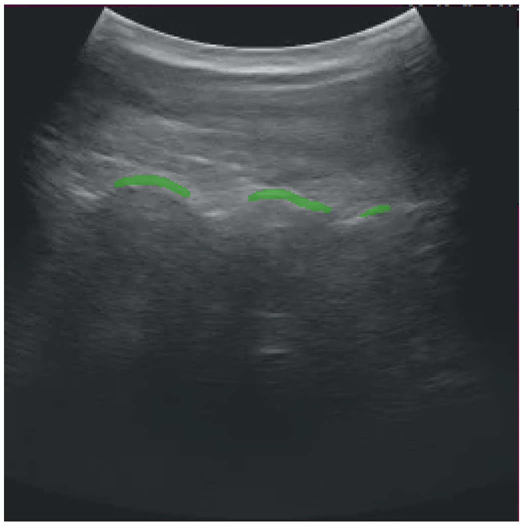

Fig. 11 *Ultrasound Segmentation*

Note

If you run the byom.py application without modification and are using the debian installation, you may run into the following error message:

```
[error] Error in Inference Manager ... TRT Inference: failed to build TRT engine file.
```

In this case, modifying the write permissions for the model directory should help (use with caution):

```
sudo chmod a+w
/opt/nvidia/holoscan/examples/bring_your_own_model/model
```

Customizing the Inference Operator

The builtin `InferenceOp` operator provides the functionality of the Inference. This operator has a `receivers` port that can connect to any number of upstream ports to allow for multiai inferencing, and one `transmitter` port to send results downstream. Below is a description of some of the operator's parameters and a general guidance on how to use them.

- `backend`: if the input models are in `tensorrt engine file` format, select `trt` as the backend. If the input models are in `onnx` format select either `trt` or `onnx` as the backend.

- `allocator`: Can be passed to this operator to specify how the output tensors are allocated.

- `model_path_map`: contains dictionary keys with unique strings that refer to each model. The values are set to the path to the model files on disk. All models must be either in `onnx` or in `tensorrt engine file` format. The Holoscan Inference Module will do the `onnx` to `tensorrt` model conversion if the TensorRT engine files do not exist.

- `pre_processor_map`: this dictionary should contain the same keys as `model_path_map`, mapping to the output tensor name for each model.

- `inference_map`: this dictionary should contain the same keys as `model_path_map`, mapping to the output tensor name for each model.

- `enable_fp16`: Boolean variable indicating if half-precision should be used to speed up inferencing. The default value is False, and uses single-precision (32-bit fp) values.

- `input_on_cuda`: indicates whether input tensors are on device or host

- `output_on_cuda`: indicates whether output tensors are on device or host
- `transmit_on_cuda`: if True, it means the data transmission from the inference will be on **Device**, otherwise it means the data transmission from the inference will be on **Host**

Common Pitfalls Deploying New Models

Color Channel Order

It is important to know what channel order your model expects. This may be indicated by the training data, pre-training transformations performed at training, or the expected inference format used in your application.

For example, if your inference data is RGB, but your model expects BGR, you will need to add the following to your segmentation_preprocessor in the yaml file: `out_channel_order: [2,1,0]`.

Normalizing Your Data

Similarly, default scaling for streaming data is `[0,1]`, but dependent on how your model was trained, you may be expecting `[0,255]`.

For the above case you would add the following to your segmentation_preprocessor in the yaml file:

`scale_min: 0.0` `scale_max: 255.0`

Network Output Type

Models often have different output types such as `Sigmoid`, `Softmax`, or perhaps something else, and you may need to examine the last few layers of your model to determine which applies to your case.

As in the case of our ultrasound segmentation example above, we added the following in our yaml file: `network_output_type: softmax`