On Designing Interfaces to Access Deep Learning Inference Services

Seungwoo Kum, Seungtaek Oh, Jungchul Yeom, Jaewon Moon
Information and Media Research Center
Korea Electronics Technology Institute
Seoul, Republic of Korea
swkum@keti.re.kr, stoh@keti.re.kr, jcyeom@keti.re.kr, jwmoon@keti.re.kr

***Abstract*—** Technologies that are related to the deployment of distribution of deep learning service onto remotes resources are getting more focuses these days. Along with the rapid development of various training technologies, the demands on running deep learning (mostly the inference) on the on-premises resources such as edge devices in factory, warehouse or farm becomes high. Though there are many resources to run an inference as a service, it needs more consideration to deal with various application scenarios from different fields. In this paper, the authors present an DL interface module, which makes it able to access deep-learning based services regardless of the kind of model and service architecture.

Keywords—deep learning; service interface; deployment of deep learning service; edge computing; resource monitoring

I. INTRODUCTION

As more deep learning models are developed with various training technologies, the demands on using the trained model as a service becomes high. [various models and training methods]. The inference service can be organized using the trained method. There are number of tools that can be used to this model-based service, a.k.a model serving. TensorFlow has the TensorFlow Serving [1], which provides methods and APIs to access and predict the model with RESTful or gRPC connection. PyTorch provides similar tools, named as Torch Serve [2], and Nvidia the Triton [3]. With these tools, using trained models for the inference becomes very intuitive.

However, to have an end-to-end service for inference is another story from having a model serving. The pipeline for the end-to-end inference service consists of various components in front of or after the model serving. Suppose that we have a model that is trained for image classification service. To provide the data to the input layer of the model, the input data needs to be resized, changing color order and transformed to an array. After applying the data to the model for the inference, the output data needs to be sorted, and matched to the class table to identify top-1 or top-5 classes. It becomes little bit more complicated when it comes to the data interfaces of the end-to-end inference service. The interfaces vary significantly depending on the use case scenarios of the service. For example, if someone wants to provide a web service for the image classification, the input data (image) will be transferred via HTTP protocol from a web browser along with the request itself, and the result (top-1, top-5 class) are to be returned with a response packet. If the use case is to classify images from a video stream of a video camera, the input will be a device handle to open the stream, and the operation will be rather autonomous than request-response like the previous example. The service will generate analysis output for each incoming frame, and mostly a Pub-Sub protocol such as MQTT seems to the right choice for this one. This demand on high flexibility on service configuration make it difficult to realize an end-to-end service for an inference service. There are a few services and research on end-to-end service pipeline, for example the DeepStream [4] from Nvidia. The model serving technology is using remote procedure call (RPC) technologies such as gRPC for its implementation. The use of RPC framework and serialized Protocol Buffer provides better performance than the request-response service.

In this paper, the authors propose an end-to-end inference service architecture, along with the interfaces to control and manage the service itself. The architecture is designed as a high-level abstraction of various kinds of deep learning-based applications, and interfaces are defined to support various protocols to be used in end-to-end services.

II. RELATED WORKS

A. Model Serving Platform

Most of deep learning platforms has been working on the model service technology. TensorFlow has released TensorFlow Model Serving a few years ago and is quite popular on providing access to the trained models on the cloud. PyTorch has a model serving as well, under the name of TorchServe. Nvidia has Triton server to provide inference on the cloud or edge resources. Though the implementations vary for each service, the concept of model service is similar: to provide access to the inference (prediction) via network interfaces. To this end, those implementations provide ways to register a trained model to the server, and APIs to access the model served. Usually the HTTP protocol is used, and for streamed process of data the Protocol Buffer and RPC protocols are used. Fig.1 depicts the model serving platform.

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2021-0-01578, Smart Farm Platform for high-quality and traceable yield. A multi-purpose DDS based on proximal and remote sensing.).

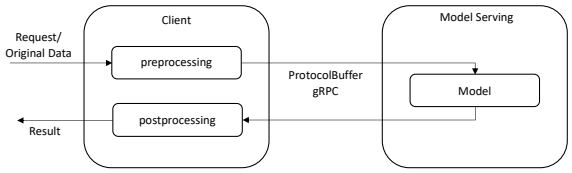

Fig.1 Model Serving Architecture

B. Pipelining DL inference application

Conventionally pipelining refers a lifecycle of developing a model and applying it –data collection, preliminary analysis of data, designing and traing of a model, validation, and feedback. However, the pipeline for DL inference is now gaining focuses from the field, which describes and end-to-end pipeline from the viewpoint of inference service. The focus of this work is to apply trained model to an application. A good example of this DL inference pipelining can be the DeepStream SDK [4] from Nvidia. Nvidia utilizes various technologies for video analysis, and provides SDK to build an end-to-end DL inference application.

From the research, the pipelining of DL inference is more focused on how to distribute or allocate deep learning models [5] so that it can optimize the resource usage [6].

III. ARCHITECTURE AND INTERFACES FOR INTERENCE SERVICE

To design the architecture, the authors have tried to abstract the process of a deep learning process in high level. Though there are different kinds of deep learning inference application, authors were able to abstract the procedure in the following: 1) input data acquisition. This process refers to the loading of the original data to be analysed with the trained model, for example, reading data from file such as csv, mp3, wav, png, mp4. 2) preprocessing of original data. This process refers to the preparation of the data to be fed on the training model by applying various computational methods, for example convert audio to Mel Spectrogram, resizing image, or leveling the input data. 3) inferencing. This process applies the preprocessed data to the trained model. This is basically the same as the model serving. 4) postprocessing of the output. This process interprets the output of the trained model so to make it valuable to the end user. For example, sorting or class matching.

The pseudo state-machine diagram of the abstracted processed are defined and presented in Fig. 2. It can be identified from the state-diagram when the interactions are required for state transition, and the interfaces are defined at those points. The interfaces defined for the architecture is given on table 1. On designing the interfaces, semantics for each argument are designed deliberately to make it able to deal with various application scenarios. On the use cases that the authors collected, it is easy to identify that there are various kinds of protocols and sometimes it needs to deliver the analysis results to multiple destinations simultaneously. For example, if a client wants to receive object detection results to an MQTT topic, and also wants to bounding-boxed video stream on RTSP location, the interfaces can be configured for both destinations.

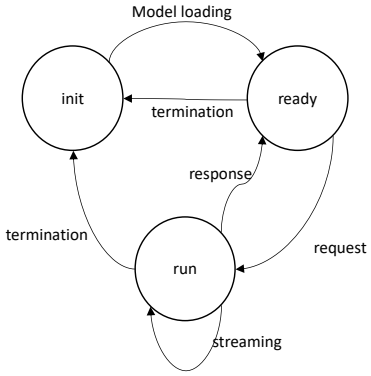

Fig.2 Psuedo-state machine of the proposed architecture

TABLE I. INTERFACE DESCRIPTION

API	Description		
	Method	***Payload***	***Description***
/input	PUT	URL	Configure source location
	GET	-	Returns source location URL
/output	PUT	{"id":"url"}	Configure destination location(s), indexed by the id
	GET	-	Returns destination locations
/compute	PUT	-	Request inference

IV. IMPLEMENTATION

The proposed architecture and interfaces are implemented on selected use cases. For the efficiency and reusability, they are implemented as docker containers. Also, to see that how the proposed work can be applied on various resources, the containers support two different hardware platforms – the ones with Nvidia GPU(s), and the other ones with Nvidia Tegra SoC. The interfaces implementation is based on SWAGGER API and provides GUI to test the interfaces.

Part of the psuedo-UML structure of the implementation is given on Fig. 3. In the figure, there are two major objects, each providing interfaces (Interface), and store the configuration (Config). The Interface provides RESTful API as designed in the section 3, and implanted with Flask and SWAGGER API. The configuration values are stored in the Config object, and the model implementation can interact with those two object to inference service.

Among many use cases of deep learning inference application, two of them that uses Yolo v4 model are selected. The first one is a service that detects objects from the image and returns the bounding-boxed image via HTTP request. The other one is a service that detects objects from a video stream (image object detection service) and returns an RTSP stream with bounding box overlayed on the original stream (video object detection service). Here the request from client is actually an HTTP GET message. Upon receiving the request message on the Interface, it requests inference computation to the model implementation, along with the configured input

data. The result is returned to the Interface object, which encapsulated in the body of HTTP response message.

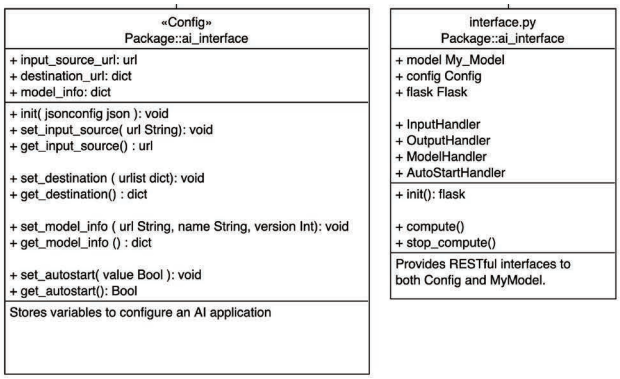

Fig. 3 Psuedo-UML presentation of Interface

For the video object detection service, it needs more consideration since it is not able to return the streamed object via HTTP request. Actually, there are a few methods to return video streaming or streaming objects with HTTP such as MJPEG over HTTP, however the encoding is not very efficient and effective in resource handling, so it is not suitable for the production-level service. Thus, the authors applied the RTSP streaming. On receiving the inference request, the model implementation spawns the thread that process inference of each image that is capture from the incoming video stream. And then instead of returning those messages with HTTP response, it creates a subprocess that encodes video stream to an RTSP stream and deliver the result to the subprocess. The URL of RTSP service is returned as a response of HTTP request. The client can access the encoded stream with an RTSP client such as VLC. The encoding and streaming subprocess is implemented with ffmpeg. The flow is depicted in Fig. 4.

The main difference between the model serving approach and the proposed architecture is whether the pre- and post-processing is done within the same process of the inference. In the proposed architecture the process is the same while model serving is not.

V. Conclusion

In this paper, the authors have proposed an architecture and interfaces for accessing inference service implementation. The service architecture reflects high level abstraction of various kinds of inference applications, and as an outcome, four processes were identified. The states and interaction are identified between those process to define the interfaces that are essential to configure and control a deep learning inference application. Also, the interfaces are designed sophisticatedly to deal with various protocols and data formats. The implementation is based on Docker container, to guarantee platform independence, and the four implemented containers shows that uniform interfaces can be applied to control multiple containers. This providing uniform set of interfaces for different kinds of implementations can help controlling large scale deployment and management of AI service with resource orchestration tools such as Kubernetes. The authors will investigate how to improve the interfaces for the trained model management, and also to apply the deployment of target containers on large scale cluster.

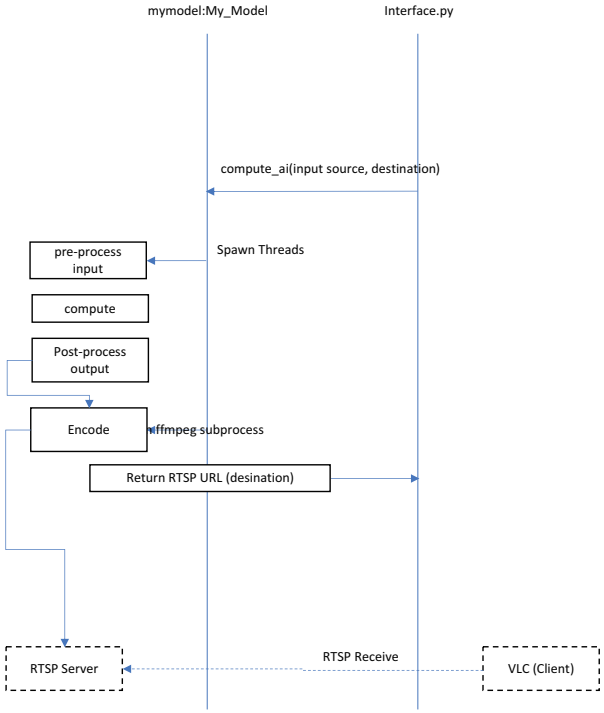

Fig.4 Flow of RTSP streaming of inference result.

References

[1] TensorFlow Serving, https://www.tensorflow.org/tfx/guide/serving

[2] TorchServer, https://pytorch.org/serve/

[3] Triton Server, https://developer.nvidia.com/nvidia-triton-inference-server

[4] DeepStream, https://developer.nvidia.com/deepstream-sdk

[5] Goel, A. et al. “Efficient Computer Vision on Edge Devices with Pipeline-Parallel Hierarchical Neural Networks,” 2022 27th Asia South Pac Des Automation Conf Asp-dac,532–537 (2022).

[6] Jeong, E., Kim, J. & Ha, S. “TensorRT-based Framework and Optimization Methodology for Deep Learning Inference on Jetson Boards,” ACM Transactions on Embedded Computing Systmes, preprints, doi:10.1145/3508391.