FlexServe: Deployment of PyTorch Models as Flexible REST Endpoints

Edward Verenich, *Clarkson University and Air Force Research Laboratory;* Alvaro Velasquez, *Air Force Research Laboratory;* M. G. Sarwar Murshed and Faraz Hussain, *Clarkson University*

https://www.usenix.org/conference/opml20/presentation/verenich

This paper is included in the Proceedings of the 2020 USENIX Conference on Operational Machine Learning.

July 28–August 7, 2020

978-1-939133-15-1

FlexServe: Deployment of PyTorch Models as Flexible REST Endpoints

Edward Verenich[§‡], Alvaro Velasquez[‡], M. G. Sarwar Murshed[§], and Faraz Hussain[§]

[§]Clarkson University, Potsdam, NY
{verenie,murshem,fhussain}@clarkson.edu

[‡]Air Force Research Laboratory, Rome, NY
{edward.verenich.2,alvaro.velasquez.1}@us.af.mil

Abstract

The integration of artificial intelligence capabilities into modern software systems is increasingly being simplified through the use of cloud-based machine learning services and representational state transfer architecture design. However, insufficient information regarding underlying model provenance and the lack of control over model evolution serve as an impediment to more widespread adoption of these services in operational environments which have strict security requirements. Furthermore, although tools such as TensorFlow Serving allow models to be deployed as RESTful endpoints, they require the error-prone process of converting the PyTorch models into static computational graphs needed by TensorFlow. To enable rapid deployments of PyTorch models without the need for intermediate transformations, we have developed FlexServe, a simple library to deploy multi-model ensembles with flexible batching.

1 Introduction

The use of machine learning (ML) capabilities, such as visual inference and classification, in operational software systems continues to increase. A common approach for incorporating ML functionality into a larger operational system is to isolate it behind a microservice accessible through a REpresentational State Transfer (REST) protocol [1]. This architecture separates the complexity of ML components from the rest of the application while making the capabilities more accessible to software developers through well-defined interfaces.

Multiple commercial vendors offer such capabilities as cloud services as described by Cummaudo et al. in their assessment of using such services in larger software systems [2]. They found a lack of consistency across services for the same data points, and also the unpredictable evolution of models, resulting in temporal inconsistency of a given service for identical data points. This behavior is due to the underlying classification models and their evolution as they are trained on different and additional data points by their vendors, providing insufficient information to the consuming system developer regarding the provenance of the model. Lack of control over the underlying models prevents many operational systems from consuming inference output from these services.

A better approach for preserving the benefits of a REST architecture while maintaining control of all aspects of model behavior is to deploy them as RESTful endpoints, thereby exposing them to the rest of the system. A popular approach for serving ML models as REST services is TensorFlow Serving [3]. However, serving PyTorch's dynamic graph through Tensor Flow Serving requires transforming PyTorch models to an intermediate representation such as Open Neural Network Exchange (ONNX) [4] which in turn is converted to a static graph compatible with TensorFlow Serving. This two-step conversion often fails and is difficult to debug because not all PyTorch features are supported by ONNX, making the train-test-deploy pipeline through TensorFlow Serving at best slow and at worst impossible for some PyTorch models. Another solution is to use the KFServing Kubernetes library [5][1], but that requires the deployment of Kubernetes, as KFServing is a serverless library and depends on a separate ingress gateway deployed on Kubernetes. Although this is a promising good solution, its deployment options are not lightweight when compared to TensorFlow Serving and Kubernetes itself can be complex to configure and manage [6].

To enable PyTorch model deployments in a manner similar to TensorFlow model deployments with TensorFlow Serving, *we have developed FlexServe, a simple REST service deployment module* that provides the following additional capabilities which are commonly needed in operational environments: (i) the deployment of multiple models behind a single endpoint, (ii) the ability to share a single GPU memory across multiple models, and (iii) the capacity to perform inference using flexible batch sizes.

In the remainder of the paper, we give an overview of FlexServe and demonstrate its advantages in scenarios such as those outlined above.

[1] KFServing is currently in beta status.

Figure 1: FlexServe architecture consists of an ensemble module which loads N models into a shared memory space. Inference output of each model is combined in a single response and returned to the requesting client as a JSON response object. The Flask application invokes the *fmodels* module, which encompasses the ensemble of models, and exposes RESTful endpoints through the Web Service Gateway Interface. Variable batch sizes provide for maximum efficiency and flexibility as clients are not restricted to a fixed batch size of samples to send to the inference service. Additional efficiency is achieved through the use of the shared memory, better utilizing available GPUs and requiring only one data transformation for all models in the ensemble.

2 Approach and Use Cases

FlexServe is implemented as a Python module encompassed in a Flask [7] application. We chose the lightweight Gunicorn [8] application server as the Web-Server Gateway Interface (WSGI) HTTP server to be used within the Flask application. This WSGI enables us to forward requests to web applications using the Python programming language. Figure 1 shows the high-level FlexServe architecture and its interaction with consuming applications.

2.1 Multiple models, single endpoint

Running ensembles of models is a common way to improve classification accuracy [9], but it can also be used to adjust the number of false negatives of the ensemble dynamically. Consider an ensemble of n models trained to recognize the presence of a specific object. By using different architectures, the ensemble model takes advantage of different inductive biases that perform better at different geometric variations of the target object. Then, combining its inference outputs according to the sensitivity policy of the consuming application, ensemble sensitivity can be adjusted dynamically. For example, let $y \in \{0,1\}$ be a binary output (0=absent, 1=present) of a classifier and let y' be the combined output. Then for maximum sensitivity the policy is $y' = y_1 \parallel y_2 \parallel \ldots \parallel y_n$, meaning that when a single model detects the target, the final ensemble output is positive identification. Different sensitivity policies can be employed by the client as needed.

2.2 Share a single device

Deployed models vary in size, but are often significantly smaller than the memory available on hardware accelerators such as GPUs. Loading multiple models in the same device memory brings down the cost and provides for more efficient inference. FlexServe allows multiple models to be loaded as part of the ensemble and performs multi-model inference on a single *forward* call of *nn.Module*, thereby removing the additional data transformation calls associated with competing methods. Scaling horizontally to multiple CPU cores is also possible through the use of Gunicorn workers.

2.3 Varying batch size

FlexServe accepts varying batch sizes of image samples and returns a combined result of the form *'model y_i': ['class','class', ...,'class']* for every model y_i. This functionality can be used in many applications. For example, to perform time series tracking from conventional image sensors or inexpensive web cameras by taking images at various time intervals and sending these chronological batches of images to FlexServe. As an object moves through the field of view of the sensor, a series of images is produced that can be used to infer movement of an object through the surveillance sector when more sophisticated object trackers are not available and video feeds are too costly to transmit. This places the computation and power burden on the Flask server as opposed to the potentially energy-constrained consumer which is only interested in the inference results of the ensemble model.

3 Conclusion

Commercial cloud services offer a convenient way of introducing ML capabilities to software systems through the use of a REST architecture. However, lack of control over underlying models and insufficient information regarding model provenance & evolution limit their use in operational environments. Existing solutions for deploying models as RESTful services are not robust enough to work with PyTorch models in many environments. FlexServe is a lightweight solution that provides deployment functionality similar to TensorFlow Serving without intermediate conversions to a static computational graph.

Availability

The FlexServe deployment module is available in a public repository (https://github.com/verenie/flexserve) which provides details showing how FlexServe can be used to deploy an ensemble model and also describes its limitations with respect to the REST interface.

Acknowledgments

The authors acknowledge support from the StreamlinedML program (Notice ID FA875017S7006) of the Air Force Research Laboratory Information Directorate (EV) and startup funds from Clarkson University (FH).

References

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 2503–2511, Cambridge, MA, USA, 2015. MIT Press.

[2] Alex Cummaudo, Rajesh Vasa, John C. Grundy, Mohamed Abdelrazek, and Andrew Cain. Losing Confidence in Quality: Unspoken Evolution of Computer Vision Services. *CoRR*, abs/1906.07328, 2019.

[3] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ML Serving. *arXiv preprint arXiv:1712.06139*, 2017.

[4] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[5] KFServing. https://github.com/kubeflow/kfserving. Accessed: 2020-02-25.

[6] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.

[7] Flask. https://github.com/pallets/flask. Accessed: 2020-02-25.

[8] Gunicorn. https://gunicorn.org/. Accessed: 2020-02-25.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.