MODEL MANAGEMENT SERVICE: A CUSTOM PUS SERVICE FOR FLEXIBLE HANDLING OF MACHINE LEARNING MODELS ON BOARD SPACE SYSTEMS

Karen Scholz and Jan-Gerd Mess

Institute of Space Systems, German Aerospace Center (DLR e.V.), Robert-Hooke-Str. 7, 28359 Bremen, Germany

ABSTRACT

The use of Artificial Intelligence (AI) and Machine Learning (ML) in space missions has become a popular approach to make spacecrafts act more autonomously. Increasing autonomy is required since communication bandwidth and the availability of ground stations is limited and missions get more and more complex and reactive. Many approaches for implementing system autonomy rely on Deep Neural Networks (DNNs) due to their achievements in the past years in several application domains that formerly required human-level intelligence. The basic mathematical operation in DNNs is the matrix multiplication. Weight matrices are consecutively applied to the input data, in order to produce a prediction. DNNs may have multiple thousands or even millions of weights. However, there exists no standardized interface that enables to upload ML models and their weights to embedded systems on spacecraft. Our approach aims to deploy arbitrary ML models including neural networks to space systems using the well-established Packet Utilization Standard (PUS) (cf. ECSS-E-ST-70-41C). Therefore, we extended our Open modUlar sofTware PlatfOrm for SpacecrafT (OUTPOST) which is available as Open Source software with a custom PUS service for dynamically loading and executing trained and validated ML models that were implemented using TensorFlow and TensorFlow Lite by Google.

1. INTRODUCTION

The use of AI and ML in space missions has become a popular approach, to make spacecrafts act more autonomously. Increasing autonomy is required since communication bandwidth and the availability of ground station contact is limited. It is recognized that increasing autonomy also increases reliability, whereas the operational effort is reduced and may also yield detection of science opportunities.
In addition to traditional ML algorithms, DNNs are increasingly applied to space applications [1]. However, these algorithms usually run on ground processing data generated onboard the spacecraft and transmitted via downlink. The basic mathematical operation in DNNs is the matrix multiplication. Weight matrices are consecutively applied to the input data, in order to produce a prediction. During a training phase the weight matrices are adapted, such that the model is suitable for its task. The required training data increases with the number of weights. DNNs may have multiple thousands or even millions of weights, which makes them computationally expensive. For example, MobileNetV2, a deep neural network for image processing tasks developed to run on mobile and embedded devices, has about 3.4 million weights [2].
Several optimization techniques have been developed, in order to reduce the size and computational cost of DNNs [3]. This and the deployment of multi-core processors enables to run DNNs onboard spacecrafts [4]. However, most optimization techniques are applied after training, such that the training process remains computational expensive. Therefore, and to validate the trained models as far as possible, ML models are usually trained and validated on ground and deployed to the embedded system afterwards. However, there exists no standardized interface that enables to load ML models on embedded systems on spacecrafts.
Our approach aims to deploy arbitrary ML models including neural networks to space missions using the Packet Utilization Standard (PUS, cf. ECSS-E-ST-70-41C). Therefore, we extended our Open modUlar sofTware PlatfOrm for SpacecrafT (OUTPOST), which is available as open source software, with a custom PUS service for dynamically loading and executing trained and validated ML models generated by TensorFlow Lite. The custom PUS service enables to upload and remove models from the spacecrafts and update them by other versions. The service further enables to execute models in an event-triggered fashion and makes the prediction results accessible for other components of the flight software.

The paper unfolds as follows:

Section 2 describes approaches and experiments from literature similar to or relevant for our work. In section 3, the general idea behind and the design of our custom PUS service and its workflow using Google's TensorFlow (Lite) is presented. Section 4 is about the service's implementation details in the OUTPOST library. In section 5, we describe the planned future validation and verification efforts to bring the new service into operation. Finally,

section 6 concludes the paper.

2. RELATED WORK

Several machine learning models have been deployed for onboard data analysis. In [5], Support Vector Machine (SVM) classifiers were trained in order to detect sulfur on images of Earth's surface by segmenting pixels showing sulfur, ice and rocks. The system was supposed to be deployed on the Earth Observing One (EO-1) spacecraft. The images were obtained from the Hyperion instrument, a hyperspectral imager onboard the EO-1, generating 220 bands of hyperspectral data. Due to limitations regarding the EO-1 processor, at most 12 of the 220 spectral bands were possible to be used for classification purposes. Therefore, feature selection techniques were applied in order to identify the 12 spectral bands leading to the best classifier performance. The authors trained linear SVMs and compared them with SVMs using a gaussian kernel. The best gaussian kernel SVM achieved a mean F-score of 0.93 in comparison to a mean F-score of 0.9 achieved by the linear SVM. However, the estimated runtime of a gaussian kernel SVM onboard the EO-1 spacecraft exceeded the estimated runtime of a linear SVM by a factor of more than 24, rendering the gaussian kernel SVM unsuitable for deployment onboard the EO-1 spacecraft.

A Random Forests classifier was deployed in [6] to the Intelligent Payload Experiment (IPEX) spacecraft, a cubesat, for realtime onboard cloud detection from image data. The classifier was trained on ground on images obtained from a prototype camera onboard a high-altitude balloon, since images from the IPEX cameras were not available during development. The IPEX spacecraft was sent into Low Earth Orbit (LEO) with the pretrained classifier installed. Updating the classifier during the mission was assumed to be infeasible due to the limited uplink bandwidth. In order to reduce computational cost, the classifier was configured to work on every fourth pixel. The remaining pixels were classified by interpolating between the classified pixels. The IPEX spacecraft used the planning sytem CASPER, which manages resources and handles requests in a priority based fashion enabling realtime classifier execution.

According to [1], also deep learning is increasingly used for space data analysis and applications. In [7], a concept for deploying DNNs on an Earth Observation (EO) satellite to classify images based on their content is proposed. The authors state that the classification results might be used to filter out informative images from noisy or meaningless images, in order to reduce downlink bandwidth when images are sent to ground. The images will be obtained by an imager onboard the satellite. A Field Programmable Gate Array (FPGA) shall interface the imager and store the raw images in memory. The images shall then be processed by the classifier enriching the image data with additional metadata based on its content. The authors mention that due to the research in how DNNs can be reduced in size while losing only little prediction performance, it will be possible to train DNNs on ground and transmit them to the spacecraft via uplink. It is planned to implement the proposed concept on board the EO satellite HyperScout 2 which is, in addition to a Central Processing Unit (CPU), equipped with a Graphics Processing Unit (GPU) and a Vision Processing Unit (VPU).

3. BASIC CONCEPTS AND METHODS

The typical workflow for deploying DNN consists of finding the right network structure and/ or cell type for the problem. While some rules of thumb exist, typically, this involves some trial and error. For this, the available data is split into multiple sets for training, testing and validation. The training data is used to train the candidate network. The candidate is then checked for performance using the validation set. This is done to test the generalization capabilities of the model when it has to deal with unknown data.

Once a suitable architecture has been found and all the so-called hyperparameters are tuned, the the performance is checked against the test set. This step is necessary to avoid that the network indirectly *learns* from the validation set through its structure and parameters. After this, the model can be applied and - given it performs adequately - is ready for operation. This workflow is depicted in Figure 1.

TensorFlow by Google [8] provides an accessible way of quickly setting up the necessary infrastructure for the described process. APIs are available for Python and JavaScript as well as - less documented - C++, all of which are, however, not suitable for embedded systems. This is due to the fact that it is optimized for the data-, memory- and runtime-intensive task of training and validation in small to very large networks that are typically executed on special workstations including powerful GPUs instead of conservative space systems.

When it comes to the inference step, which is the most interesting part of the process for deployment, a dedicated library calld TensorFlow Lite allows execution on 32-bit embedded systems. In order to transfer models to an embedded system, standard TensorFlow models can be converted to their TensorFlow Lite equivalent and then serialized using FlatBuffers, a serialization library that was also developed by Google. This step is depicted in the left part of Figure 2. The TensorFlow Lite engine can then perform inference using these models by reading the serialized FlatBuffer, building up the computational tree and executing it using input variables of the embedded system.

For space systems, the problems of upload, on-board management of existing models and the interconnection to other components of the Flight Software (FSW) remain. In order to overcome these, the following section

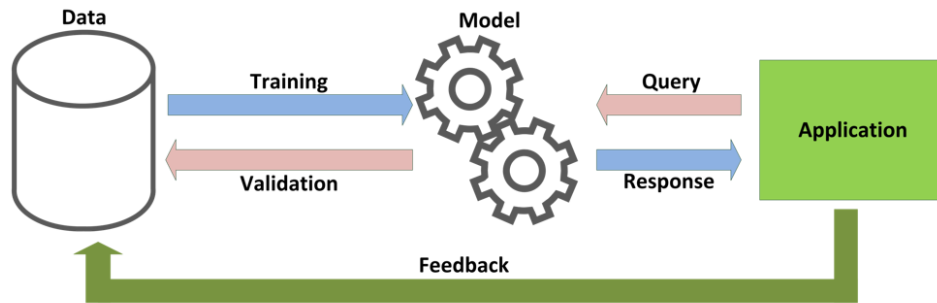

Figure 1. Typical ML process

describes our design of a Model Management Service for on-board handling of arbitrary models using TensorFlow and TensorFlow Lite as an exemplary execution engine.

4. IMPLEMENTATION

The typical process of ML starts with training a created model. According to the literature presented in section 2, models are usually trained on ground, before they are deployed to a spacecraft. As mentioned in [7], this is motivated by practical reasons: On the one hand when training is performed on ground, high-performance processing units such as GPUs can be used, since model training can be computationally expensive, and on the other hand, synthesized training data can be used to overcome the problem of missing real mission data. Additionally, more extensive validation and verification of the resulting model can be conducted before the upload, greatly reducing operational risks.

Figure 2 shows the resulting process of loading and executing an ML model on board a spacecraft using the Model Management Service, which is based on the aforementioned considerations. Currently, the Model Management Service is limited to models generated with TensorFlow Lite, since TensorFlow Lite benefits from the Google's TensorFlow ecosystem, is easily accesible through its Python implementation and directly provides all the necessary tools for creating a serialized representation of the model i.e. the FlatBuffer, which can be transmitted via uplink, while also reducing the required memory space. First, the model is trained and validated on ground using TensorFlow. Afterwards the model is converted using TensorFlow Lite into the TensorFlow Lite FlatBuffer, which is enriched with further model attributes required for serialization and execution on embedded systems. The resulting serialized model representation is then transmitted to the spacecraft by the Model Management Service via uplink. The Model Management Service stores the serialized model representation in memory. Currently, models are held in volatile Random Access Memory (RAM) which is likely to be replaced by storage in a permanent file system at a later stage. When an application queries the model, the serialized model representation is deserialized and a prediction is performed. The process of loading and storing the model onboard the spacecraft as well as the process of triggering model execution and publishing results is described in detail in sections 4.1 and 4.2, respectively.

4.1. Model Management

Onboard the spacecraft, the serialized model representations are stored in memory slots. A memory slot is an abstraction of a preallocated block of memory of fixed size. Since available resources might vary according to the mission, the number of memory slots and their size can be configured with respect to the target system during compile time. Each memory slot is assigned a unique id for identification purposes. When loading a model onboard the spacecraft, the memory slot the model should be stored in, is specified. The Model Management Service ensures that the specified memory slot is available (i.e. not occupied by some other model) and appropriately sized to store the model.

Oftentimes, training and validation data generated within a comparable mission is not available during development. The authors in [6] obtained training and validation data from a prototype camera onboard a high-altitude balloon. However, some lightning conditions were not captured in the data leading to a poor model performance on these lightning conditions. This indicates the need to retrain a model during the mission using real mission data. Therefore, the Model Management Service provides an update mechanism, which enables to overwrite a model by another version of the model. A model's version is specified within the model's meta data during upload. This facilitates versioning between ground and space segment and allows for future referencing of decommissioned models.

4.2. Inter-component Communication and Event-triggered Model Execution

In contrast to the simplified Figure 2, there might be more than one application providing input data to a model and

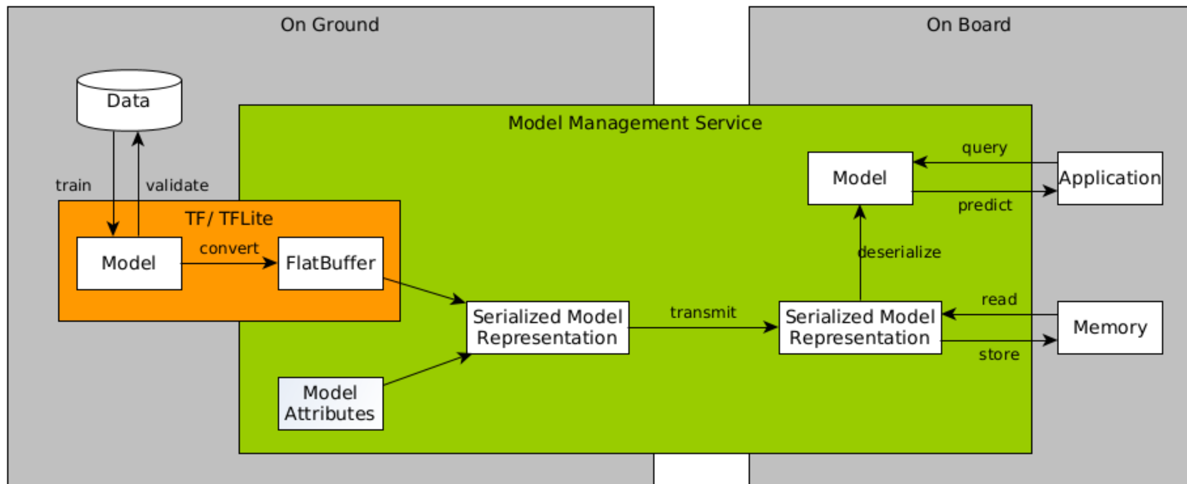

Figure 2. Process of loading and executing an ML model onboard a spacecraft using the Model Management Service

querying for prediction results. Also, the applications providing input data and the applications querying for prediction results might be different ones. Additionally, update rates are likely to vary from parameter to parameter and may not correspond to the desired rate of predictions by an ML model. To ensure a loose coupling between the Model Management Service and the applications generating parameter values or working on the results while still providing the possibility of implementing "sytem-wide" models, the Model Management Service makes use of the OUTPOST *parameter store*. The parameter store can be seen as a database, which assigns an unique identifier to each stored parameter, under which the parameter can be accessed for reading (only the parameter's owner application typically has write access). Due to the parameter store, the exchange of data between the applications and the Model Management Service is reduced to read and write operations. Figure 3 visualizes the processes of how applications and sensors onboard exchange data with the Model Management Service via the parameter store.

First, an application generates new data (status information, sensor readings, etc.) that may, in term, trigger execution of a certain model managed by the Model Management Service. The application writes the generated data to the parameter store. The Model Management Service can now be triggered by mechanisms defined in the PUS, including but not limited to event generation (Event-Action Service) or periodic scheduling. The Model Management Service obtains the serialized model representation from the corresponding memory slot and deserializes the model. It then reads the input data from the parameter store and executes the model. The generated prediction data is written back at fixed ids to the parameter store. The parameter ids are known to the Model Managment Service, since they are comprised in the model attributes. All applications interested in the prediction results are then able to receive them by reading the data from the parameter store (again, through event generation or periodic scheduling).

5. VALIDATION AND VERIFICATION

It is well recognized that validation and/ or verification of neural networks is a crucial topic, especially in safety-critical systems like spacecraft, let alone manned space systems. The demonstrated approach, however, does not only deal with ML models, but also the 3rd-party library TensorFlow (Lite) by Google, that was originally not designed for safety-critical systems. Finally, the self-implemented custom PUS service and the surrounding ecosystem for memory management and model handling need verification. We deal with these entities individually in the following sections.

5.1. Verification of ML models

Due to their potentially large number of (trainable) parameters and blackbox character, neural networks render traditional verification approaches intractable and are especially difficult to validate. Since this paper deals only with the operation and application of ML models in space systems rather than their design and training, laying out a complete process for verification is out of scope. Potential approaches include but are not limited to reachability analysis using simplifying representations [9], exploiting piecewise linearity of certain classes of neural networks and taking insights through approaches like Satisifiability Modulo Theory [10] or over-approximation by reducing the network's size to make it amenable [11].

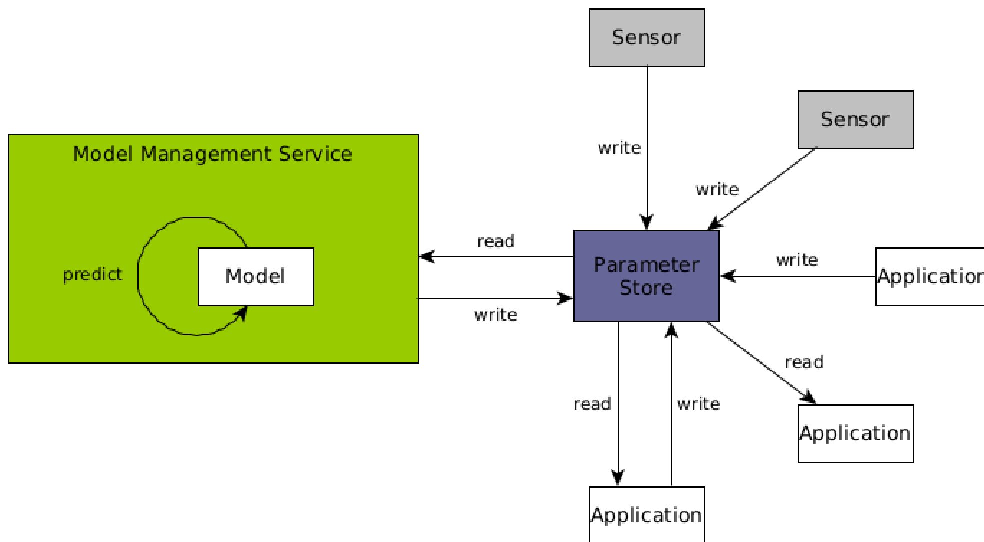

Figure 3. Exchange of data between onboard applications, sensors and the Model Management Service using the parameter store

5.2. Verification of 3rd-party libraries

Verfiation of 3rd-party libraries for safety-critical applications is always causing a lot of effort because it involves working through an unknown codebase (if available) or exhaustive blackbox-testing of the library's API. This is especially true for big libraries involving highly optimized, complex mathematical operations such as TensorFlow (Lite) that were orignally not meant for safety-critical systems.

Again, a complete discussion of the topic is beyond scope and extent of this paper. However, a few approaches include but are not limited to the following:

In [12], an interface-grammar describing the library's behaviour is used as input for a static control, dataflow and alias analysis.
The authors of [13] are applying the Coq proof-assistant - typically used for assistance in mathematical proofs - for reasoning on software correctness.
In [14], a complete framework for software verification is presented that splits the process of verification in the individual steps of pre- and post-condition inference for functions, stub inference for abstraction outside the verification scope and automatic refinement of conditions to avoid false alarms.

5.3. Planned IOV of the custom PUS service

In order to bring the Model Management Service into orbit, we follow a series of steps for graceful validation, not only tthe service but also ML in general. Therefore, we first deploy the system in a minimal example to development boards, potentially STM32 or Zynq based on ARM Cortex-M and Cortex-A processors or our own Software Development Model (SDM) (i.e. breadboard) based on Leon3. On successful execution of these tests, the service can be integrated in the FSW of our Eu:CROPIS compact satellite. Using the satellite's Ground Reference Model (GRM), the service can safely be tested for stability, usability as well as performance in the context of operational FSW. Finally, the service can be uploaded to orbit to the Eu:CROPIS satellite currently in its extended mission for In-Orbit Verification (IOV). It should be noted that no actual control shall given to any ML model but rather use them as an observer of the system state (or rather subsets thereof) looking for anomalies and reporting its findings to ground.

6. CONCLUSION

In this paper, we have discussed a custom PUS service for spacecraft that handles ML models by providing capabilities for upload, on-board memory management, versioning but also their inference on on-board parameters that can be triggered both automatically as well as manually. The service is implemented in DLR's software library OUTPOST and consequently integrates well with existing FSW as well as future developments.

While leaving out evidently important topics such as verification of 3rd party libraries and ML models in general for now, we are convinced that this work paves the way for applying methods of AI and ML on-board spacecraft by providing a "standardized" way of handling models and providing access to the software system.

REFERENCES

[1] Dario Izzo, Marcus Märtens, and Binfeng Pan. A survey on artificial intelligence trends in spacecraft guidance dynamics and control. *Astrodynamics*, 3(4):287–299, 2019.

[2] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[3] Nesma M. Rezk, Madhura Purnaprajna, Tomas Nordström, and Zain Ul-Abdin. Recurrent Neural Networks: An Embedded Computing Perspective. *IEEE Access*, 8:57967–57996, 2020.

[4] Vivek Kothari, Edgar Liberis, and Nicholas D. Lane. The Final Frontier: Deep Learning in Space. In *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, pages 45—49, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Lukas Mandrake, Umaa Rebbapragada, Kiri L. Wagstaff, David Thompson, Steve Chien, Daniel Tran, Robert T. Pappalardo, Damhnait Gleeson, and Rebecca Castaño. Surface Sulfur Detection via Remote Sensing and Onboard Classification. *ACM Trans. Intell. Syst. Technol.*, 3(4), 2012.

[6] Alphan Altinok, David R Thompson, Benjamin Bornstein, Steve A Chien, Joshua Doubleday, and John Bellardo. Real-Time Orbital Image Analysis Using Decision Forests, with a Deployment Onboard the IPEX Spacecraft. *Journal of Field Robotics*, 33(2):187–204, 2016.

[7] Gianluca Furano, Antonis Tavoularis, and Marco Rovatti. AI in space: applications examples and challenges. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020.

[8] TensorFlow official web page. `http://www.tensorflow.org`. Accessed: 2021-06-13.

[9] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *Computer Aided Verification*, pages 3–17, Cham, 2020. Springer International Publishing.

[10] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and M. Pawan Kumar. A unified view of piecewise linear neural network verification. *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018.

[11] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. An abstraction-based framework for neural network verification. In *Computer Aided Verification*, pages 43–65. Springer International Publishing, 2020.

[12] Marcus Mews and Steffen Helke. Towards static modular software verification. In Stefan Jähnichen, Bernhard Rumpe, and Holger Schlingloff, editors, *Software Engineering 2012. Workshopband*, pages 147–153. Gesellschaft für Informatik e.V., 2012.

[13] Christine Paulin-Mohring. *Introduction to the Coq Proof-Assistant for Practical Software Verification*, pages 45–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[14] Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Naoto Maeda, Hiroki Tokuoka, Takashi Imoto, and Yoshiaki Miyazaki. Dc2: A framework for scalable, scope-bounded software verification. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 133–142, 2011.