SSC24-WI-04

Implementing a Neural Network Execution Framework in Realistic Space Hardware and Software as a Pseudo On-Orbit Demonstration

Rafael Polanco-Segovia, Piyush M. Mehta
Department of Mechanical, Materials and Aerospace Engineering, West Virginia University(same for all)
1374 Evansdale Drive, Morgantown, WV, 26506, USA(same for all); +1 (304) 293-4821
rp00028@mix.wvu.edu

Scott A. Zemerick
TMC Technologies, NASA Independent Verification and Validation (IV&V), Jon McBride Software Test and Research (JSTAR) Team
2050 Winners Dr, Fairmont, WV, 26554, USA; +1 (304) 816-3600
scott.zemerick@tmctechnologies.com

ABSTRACT

Recent advances in hardware and software technology have made it possible to implement more resource-demanding deep learning algorithms in lighter hardware environments. This creates opportunities to use deep learning for space applications on increasingly lighter and smaller spacecraft. The goal of this work is to demonstrate the viability of implementing a Neural Network Execution Framework (NNEF) that can facilitate a cross-platform and unified deployment of any neural network onboard a spacecraft hardware and flight software. The NNEF generalizes the neural network inference process, regardless of the original framework in which they were created. This allows users to focus on the development of their scientific model architecture and deep learning objectives, rather than being distracted by the implementation process onboard the spacecraft. This framework has been implemented to run inside NASA's core Flight System and on top of a Raspberry Pi 4 board, demonstrating the capability to execute a variety of trained neural networks created in Pytorch and Tensor Flow. This includes a neural-based compression algorithm used to process images from NASA's Solar Dynamics Observatory in a space-like hardware-software configuration. This initial software implementation shows the feasibility of our goal, demonstrating the deployment of deep learning benefits through our framework in a unified way for a broader range of space missions and applications. In addition, for comparison purposes (not for benchmarking), it showed the performance of the networks running in the mentioned hardware-software configuration contrasted with the performance obtained in a regular computer environment.

INTRODUCTION

The purpose of implementing neural networks (NN) on a spacecraft is to utilize deep learning (DL) technology for enhancing new science and engineering applications in space. This enables spacecraft to operate autonomously, reduces communication costs, data volume, and latency, and optimizes the use of computational resources. There is extensive research focusing on the application of NN for DL in solving space science and engineering challenges related to exploration, observation, and operational tasks, such as[1,2,3,4]:

- Analysis of satellite and astronomical imagery
  - Image classification and object detection (asteroids, craters, etc.)
  - Change detection (clouds, space debris, cities growth, etc.)
- Dynamics, navigation, guidance, and control
  - Spacecraft propulsion, inertia parameters, spherical harmonics, and drag coefficients
  - Terrain mapping and landing system
  - Rendezvous and docking operations assistance
  - Detection, tracking, and prediction of object trajectory
- Onboard data processing
  - Reduce transmission bandwidth by data encoding

– Signal processing, spectral analysis, and pattern recognition
– Telemetry analysis

On the one hand, current research is addressing technical challenges such as data availability, compression models, and validation of models for artificial intelligence (AI) capabilities on spacecraft.[2] On the other hand, the research and development dedicated to onboard deployment of DL solutions in the actual constrained hardware-software environment of a spacecraft remains largely unaddressed. Despite the considerable challenge posed by the computational resource requirements for DL, we have made enough progress in dedicated computing hardware platforms that allow DL models to be deployed in space.[2] Any space DL solution should rely on having the appropriate hardware architecture, such as GPUs, FPGAs, and neuromorphic processors, to have the computational power to manage the billions of calculations needed to train and execute scientific-driven and complex NN. However, for space, there is a trade-off between the required computing hardware and the cost and limited power and weight requirements for spacecraft. As in any modern flight system, software also has a huge role in space systems. Efforts have been made to explore the state-of-the-art DL software implementations for embedded systems to integrate unique hardware-software configurations to run DL algorithms for specific spacecraft missions and space applications. For example, one research[5] shows the performance of three different convolutional NN model architectures for computer vision tasks for space applications are verified and validated on a specific hardware architecture to analyze the DL models before deploying them in a real spacecraft. These are Earth hyper-spectral image segmentation, Mars object image classification, and Moon crater image detection. They develop and train the model using the TensorFlow framework. Another research[6] describes a DL testbed flight demonstration conducted on existing flight hardware using TensorFlow Lite for terrestrial-scene image classification (collected in flight by the STP-H5/CSP system). It compares the accuracy, the execution time, and the runtime memory usage, when executing modern pre-trained convolutional neural networks (CNN), such as MobileNet or Inception-ResNet. These efforts demonstrated that it is possible to achieve a reasonable performance executing modern NN on a low space-grade computational resource embedded platform.

However, for deep learning technology to become commonplace for different types of space missions, it is necessary to offer a DL software framework that enables a generic and easier deployment and inference process of any NN models. This DL framework should be suitable for a wide range of hardware architectures and operating systems normally used in spacecraft, including small satellites. Which contributes to DL space applications with limited CPU/GPU power. Also, this framework should be an important component to optimize the memory usage, link throughput, latency, and cost of space communications and science data delivery.

We are aware that to implement a space DL software framework with such characteristics, as the one we are proposing, it is essential to have a flight software framework as a foundation for our software generalization across different platforms. This provides access to the spacecraft hardware infrastructure to interact with (memory management, processors, storage devices, communication devices) and the underlying operating system in an abstract, and homogeneous way. It creates a run-time environment by providing services that are common to most flight applications (telemetry output, command receiving, message bus, file and data transfer, etc.) that simplify the flight software applications development. All of this allows us to focus on developing our DL software framework without worrying about the concrete hardware and operating system implementations.

There are already space flight software frameworks such as Core Flight System[7] (cFS) or F Prime (F'),[8] developed by NASA Goddard Space Flight Center and NASA Jet Propulsion Laboratory, respectively, used in several space flight software-based missions. There is little information on current space missions using any space flight software to execute NN algorithms. Some DL technology test efforts are ongoing. For example, NASA Glenn Research Center explores the use of cFS and specialized hardware to implement cognitive communications capabilities onboard spacecraft to improve the autonomy of space communication.[9] The project emphasizes the development of cognitive decentralized space networks with AI agents optimizing communication link throughput, data routing, and system-wide asset management. They propose future virtualization of cognitive networks using cFS for code reuse across diverse types of missions, working on cFS apps as RTEMS[1] virtual machines. DL models could be executed inside a cloud-centric service

[1]Stands for Real-Time Executive for Multiprocessor Systems. It is a multi-threaded, single-address-space, real-time operating system

architecture in space.

Based on the examples shown, even though these flight software provide a platform to implement mission-specific DL applications, we can affirm that there is still a lack of a DL framework to generalize the implementation of NN models in space missions. Then, to fill this gap, we conceptualized and developed a **Neural Network Execution Framework** that allows the deployment and execution of the inference process of any (or almost any) neural network models for any DL-based space missions. No matter what space flight software, operating system, and hardware configuration are used.

This document is organized as follows: Section 2 depicts the software architecture that was used to model and develop our framework as a generic way for NN deployment on top of any space flight software. Then, section 3 shows a specific framework implementation using cFS, including the component view and its relationship to execute a typical use-case workflow. One subsection is dedicated to describing cFS architecture as the space flight software used for this phase of the NN framework development. Next, section 4 shows a demonstration of how the framework could be used in the inference of several NN, modeled and trained in Pytorch and TensorFlow. The final sections show the demonstration results, conclusions, and future work.

NEURAL NETWORK EXECUTION FRAMEWORK SOFTWARE ARCHITECTURE

A DL software framework involves a platform, libraries, a programming language, and a set of functions that allow us to model, train, test, and deploy NN models with application domains like prediction, image processing and recognition, and much more. Nowadays, the main frameworks to work with DL model architectures are TensorFlow, PyTorch, and Caffe, among others. To execute these NN models in a typical C/C++ space environment, such DL frameworks need to have a way to save their models as Model Scripts (and most of them have it). With this, a spacecraft could execute these model scripts without requiring the complete DL software framework onboard installation. This agrees with and enables the goal of making DL accessible to virtually any spacecraft hardware and software configuration.

Therefore, with all of this in mind, we created our Neural Network Execution Framework (NNEF) with the capacity to execute the inference process on any NN model (saved as a Model Script) for space applications. Its software architecture is implemented as an extension layer made of a group of components on top of space flight software for getting the inputs from science applications and delivering the results through the communications channels (receiving commands, sending telemetry, and transferring files from and to the ground station). Also, with the space flight software we can access the abstracted hardware (i.e., CPU/GPU, memory, storage, etc.) and software resources (i.e., the operating system, basic flight functions, etc.). In summary, performance and execution depend on the hardware, resource abstraction depends on the space flight software, and implementation and deployment generalization are allowed by our NN execution framework. We built this version to integrate the following characteristics:

- Support for the most common and important DL frameworks, as long as they can save their NN models as scripts to be executed in a lightweight environment.
- If one or more specific NN models are implemented in our framework, it allows their execution using the same interface (inputs and outputs), no matter what the DL framework used to make and train that model.
- Allow the execution of trained NN models by loading the corresponding Model Script file, receiving the input dataset from another science application or file system, and delivering the inference output results.
- The NN model training is supported, but is not suggested for small spacecraft due to its computational resource limitations.
- Uploading the NN Model Script file from the ground station, allowing to update it at runtime without reinstalling or rebooting any of the framework software components.
- Integration with the spacecraft command input system to start the inference process of one of the deployed NN models.
- Integration with the spacecraft telemetry output system to send the status of the NN process and the NN execution platform.
- Storage of the inference output data, which could be downloaded to the ground station.
- All of the above should be managed from the ground station through a command-and-telemetry local system.

Figure 1 shows how the current architecture has been implemented.

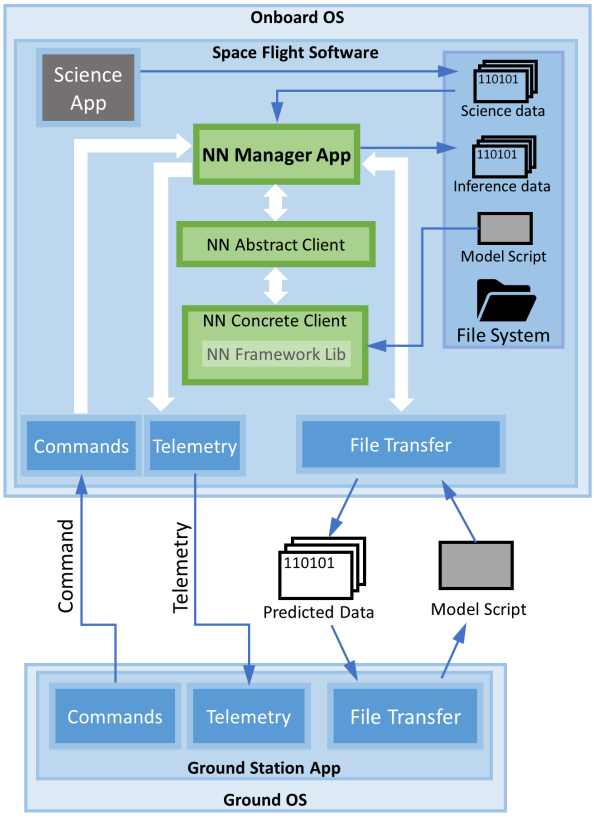

Figure 1: Neural Network Execution Framework. General Software Architecture

The architecture's components are described as follows:

A. An application called **NN Manager** serves as the main interface for commanding the NN execution. This command should contain the NN model the user wishes to execute and all of its parameters, such as the used DL framework Model Script (Python, TensorFlow, etc.), dataset size, and batch size, among others. This component reads the input dataset, which may come from messages or files generated by other onboard science applications, executes the corresponding NN Abstract Client component (see item B.), and manages and saves in the file system the predicted or inferred NN results. Then, space flight software will send them to the ground station. Additionally, the application takes charge of sending its status updates, housekeeping telemetry, and the status of the NN process.

   To do its job, the NN Manager should integrate with the underlying space flight systems, adhering to its standard application architecture and rules. It should receive and send messages from it for handling command inputs and telemetry outputs, and it should interact with it to save and read files from the storage device. It is the responsibility of the space flight software to communicate with the ground station software for command and telemetry interaction and the transfer of files between them.

B. **NN Abstract Client**, consisting of one abstract class, has the primary role of providing a generic and abstract interface for the execution of one specific NN model, having one abstract class for each NN Model type. It follows the abstract factory object-oriented design pattern. This pattern is a well-known creational pattern to instantiate concrete classes to produce concrete products, but the user deals only with the abstract definitions of those classes, represented by this abstract client, ignoring the actual implementations. In this case, the user role is done by the NN Manager, which will interact in the same way with the NN Model, no matter what DL framework was used for its modeling and training.

C. Behind the previous abstract client, a **NN Concrete Client** class is instantiated to load and execute the NN Model Script developed and trained in its specific DL framework. The user should integrate one new concrete client per DL framework Model Script for that NN model type. Here. Each implementation uses its own C/C++ code and additional proprietary framework libraries to train, test, or execute the inference process of the specific NN model.

NN FRAMEWORK IMPLEMENTATION WITH NASA'S cFS

As a starting point and specific implementation, we use NASA's cFS as the base space flight software of our NN execution framework, so it can be integrated into cFS-based missions. But there are other flight software platforms in which we could integrate the framework shortly.

cFS as Our Space Flight Software

Thanks to the platform-independent architecture of cFS and the growing number of space missions that use it,[10] we selected such space flight software

as our first choice to implement our NNEF. Figure 2 shows the cFS dynamic run-time environment and its layered architecture and component-based design.

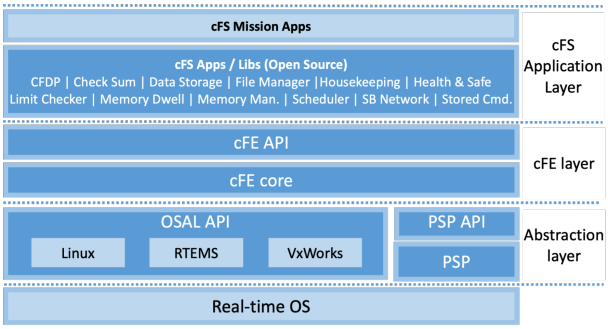

Figure 2: cFS Layered Service Architecture.[7] The Core Flight Executive (cFE) is a portable, platform-independent framework that creates an application runtime environment. Applications provide mission functionality using a combination of cFS community apps and mission-specific apps. The OS Abstraction Layer (OSAL) is a software library that provides a single API to the cFE regardless of the underlying real-time operating system. The Platform Support Package (PSP) is a software library that provides a single API to underlying avionics hardware and board support package.

Implementation Software Architecture

Following the section 2 implementation rules, we integrate our NNEF as a new layer on top of a cFS. This means that our framework classes are now an extension group of cFS applications and libraries, on top of a standard cFS framework.

Figure 3 shows how the NNEF architecture is adapted to cFS.

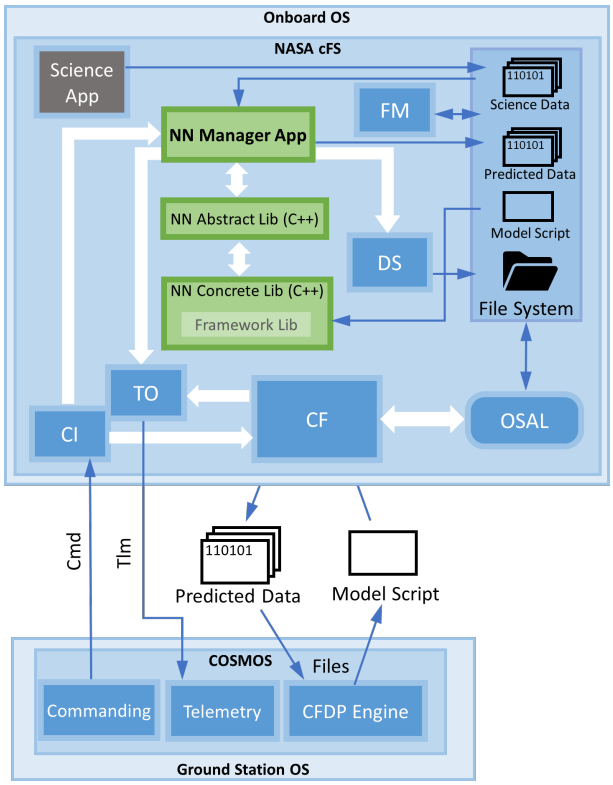

Figure 3: NN Execution Framework and cFS Implementation. Software Architecture

The specific architecture's components are described as follows:

a. NN Manager is implemented as a cFS application, so it adheres to the standard cFS application architecture and rules. As mentioned in item A., for this case, this class integrates with cFS, receiving and sending CCSDS[2] messages from and to the cFE Software Bus for handling interaction with Command Ingestion (CI) and Telemetry Output (TO) components. To save the predicted or inferred NN results for the cFS to send them to the ground station, the NN Manager uses the Data Storage (DS) application, sending the binary results as software bus messages. DS is configured to receive those messages and save them on a storage device such as a solid-state recorder.[11] Also, through cFE Executive Services (ES), the application interacts with the underlying PSP and OSAL layers, as explained in Figure 2

b. The classes described in items B. and C. are implemented as cFS C++ libraries. So, its code is independent of cFE services and cFS applications.

[2]Stands for Consultative Committee for Space Data Systems.[12] Supports CCSDS version 727.0-B-5.
[3]Stands for CCSDS File Delivery Protocol.

c. Operations outside of our NNEF. These operations depend only on the cFS:

c.1 COSMOS ground station software is used for communications between satellites and ground stations. This component sends commands, receives spacecraft telemetry, and supports CFDP[3] for file transfer through its CFDP Engine. cFS can communicate with the ground station software (COSMOS) using CI and TO components.

c.2 With CFDP application (CF) and its handshake with COSMOS CFDP Engine, the user can download the inferred NN result files to the ground station and upload the NN Model Scripts files to be executed in the concrete clients (See item C.). It works by mapping CFDP-compliant PDUs in and out of the cFS's software bus.[13]

c.3 The File Management (FM) application provides onboard file system management services by processing ground commands for copying, moving, renaming, and decompressing files. Creating directories, deleting files and directories. Also providing file and directory informational telemetry messages.[14]

Figure 4 below clarifies the framework execution workflow and how each component is operating:

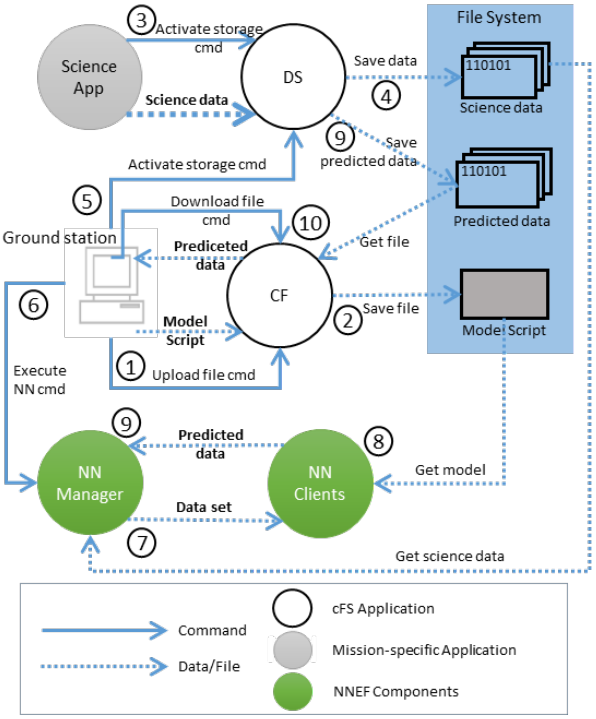

Figure 4: NN Execution Workflow

The next numerated list describes Figure 4 steps:

1. If the NN Model Script hasn't been uploaded yet or there is a need to update it, a ground command is sent to start Model Script file uploading.

2. CF manages the transmission of the file and saves the Model Script in the file system or spacecraft data recorder (SDR).

3. A Science App (or ground station) sends a command to DS that enables the storage of any subsequent science data message.

4. DS will take and save on the file system all available science data messages for which it is configured.

5. A command is sent from the ground station to prepare the DS for the storage of the inferred (predicted) data from the NN that will be produced in the next steps.

6. Ground sends a command to the NN Manager to execute the NN Model. Should indicate the desired NN Model type, the concrete NN model implementation, the data size to be processed, the batch size, and the reserved memory for each one of the inferred elements.

7. NN Manager gets one instance of the corresponding NN abstract and concrete clients (cFS libraries) and sends them the input data in binary form.

8. The concrete NN client acquires the Model Script and loads the trained NN model. Then transform the input data to obtain the dataset (typically in tensor form) and feed the NN model with it. The inferred results are transformed again into binary form and sent back to the NN Manager.

9. NN Manager formats and sends the results to the bus, so that DS saves them in the spacecraft file system.

10. A ground command is sent to CF to download the result data files. CF manages the download transaction to the ground station.

NN FRAMEWORK DEMONSTRATION

Chosen Neural Networks

The reader can observe in this document's introduction that many types of NN models have suitable execution onboard a spacecraft. As a proof of concept, we show two types of NN Models:

1. A simple **Convolutional Autoencoder** (CAE) is selected as a NN inference demonstration, with no particular space application, but the reconstruction of handwritten digits image dataset. The same CAE model was implemented, trained, and tested using both Pytorch and Tensorflow frameworks. Figure 5 shows the layered architecture of the selected CAE NN. The Encoder segment captures a reduced representation of the input image, called Latent Space. The Decoder segment tries to reconstruct the image as well as possible. The unsupervised NN is trained to create a latent space, so the differences between the original images and restored ones are minimized. It used 60,000 MNIST[4] images for training and 20,000 images for testing the models.

   Although the model is made of two different sub-models (encoder and decoder), both of them were trained and tested as a whole and unique model. The trained encoder segment is meant to be run on the spacecraft flight software through our NNEF. The encoder reduces the size of the input images (assuming they were created by a previous process or application and saved in the spacecraft file system) and creates the corresponding latent space of each image. They can be downloaded to the ground station. This ensures minimal space and latency in data transmission. Next, the decoder segment will be executed at the ground station to acquire the decoded images back using the corresponding DL framework.

   Both the NN model and training examples, from Pytorch and Tensorflow, are available in the GitHub folder repository `https://t.ly/FX2AO` under the `MNIST_CAE` name. They can be executed using the Jupyter Notebook on Linux environments to observe the training and testing processes and to watch for a comparison between original and reconstructed images (see table 1 for product version information). At the end of each code example, the Model Script that represents the trained encoder model is saved to be uploaded and executed in the spacecraft. The Pytorch encoder Model Script is saved as a single file *TorchScript* format. The Tensorflow encoder Model Script is saved as a multiple file *SavedModel* format. We also created a specific example code to transform and decode the binary output data that represents the encoded images coming from the spacecraft.

2. A **Neural-based Image Compressor** on NASA's Solar Dynamics Observatory (SDO) is selected as a complex NN inference demonstration with a particular space application as the efficient compression applied to solar images that comes from the SDO. This NN Model was developed and trained in Pytorch using specialized quantization and entropy NASA libraries to encode and decode the images.[16] Figure 6 shows the architecture of this NN Model. See the reference for more details about the architecture, training, and results.

   We took a checkpoint of the pre-trained model from `https://t.ly/ds0LW`, and, as in the previous CAE NN Model, only the compression model is executed on the spacecraft flight software through our NNEF. This reduces the size of the input images and creates the corresponding latent space. For testing, we used a 32 SDO images dataset, previously saved in the spacecraft file system.

   The Pytorch code to test this model is available in the GitHub folder repository `https:`

[4]Stands for Modified National Institute of Standards and Technology database of handwritten digits[15]

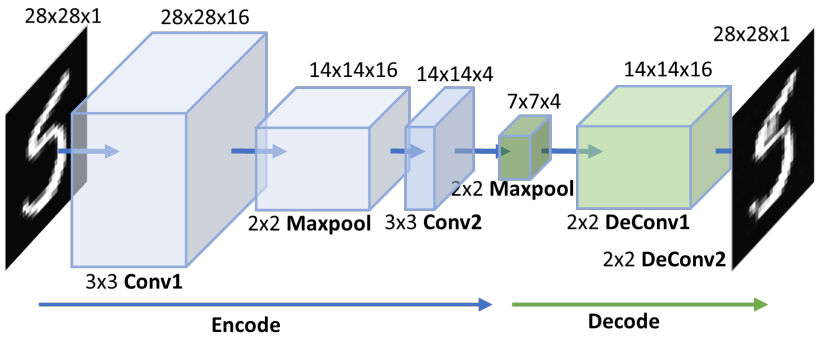

Figure 5: Simple CAE Architecture

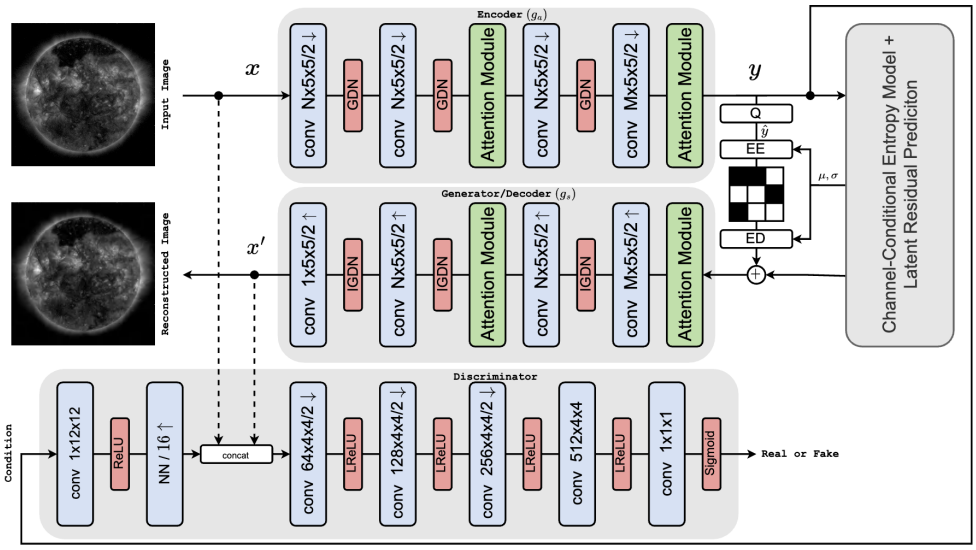

Figure 6: NASA Neural-based Image Compressor. The input image is down-scaled by a factor of 16 to get the latent code and up-sampled in reverse to get the reconstructed image.[16]

//t.ly/FX2AO under the nasa_compressor name. At the end of the code, the Model Script that represents the trained compressor model is saved to be uploaded and executed in the spacecraft. The Pytorch compression Model Script is saved as a single file *TorchScript* format. They can be executed using the Jupiter Notebook under Linux environments to observe the evaluation process and to watch for a comparison between original and reconstructed images (See table 1 for product version information). We also created a specific example code to transform and decompress the binary output data that represents the compressed SDO images coming from the spacecraft.

As we already mentioned, this NN model originally uses specialized quantization and entropy C/C++ libraries for the compression model, which are linked to the original Pytorch code. However, we needed to change the original Pytorch code, so it was possible to transform the NN model to its Model Script version and recompile such libraries to access them from the Model Script version and the original Pytorch model as well.

Table 1: Product Versions to Train and Test Both NN Models

Framework/Application	Version
Jupyter Notebook (Linux)	7.1.1
Python	3.11.4
Pytorch	2.2.1
Tensor Flow	2.13.0

Neural Networks Implementation

Sticking to the proposed NN Framework software architecture to execute any NN on the spacecraft, shown in 1, specific abstract and concrete components were created to deal with the execution of the chosen NN Models. Figure 7 shows these components for both Pytorch and TensorFlow CAE NN and the related elements. Also, in Figure 8, it is shown the software components for the Pytorch NASA Neural-based compressor NN.

Following the workflow enumerated steps in Figure 4, for each tested NN Model, the specific NN components and their role inside the workflow are described.

For the Convolutional Autoencoder NN Model (check with Figure 7),

- In steps 1 and 2, the Encoder NN Model Scripts are uploaded to spacecraft. The TorchScript Encoder NN and SavedModel Encoder NN script files come from the Pytorch and TensorFlow frameworks, respectively.

- For step 6, the user indicates to the NN Manager App that he wants to execute the MNIST Encoder part of the CAE NN Model. Also, he indicates if he prefers to use the Phyton or the TensorFlow version. Also, he should set the appropriate dataset size, batch size, and inferred element memory reserved size.

- For steps 7 and 8, the abstract client would be **MINST Encoder Lib**, which is responsible for obtaining and executing the corresponding concrete Encoder client according to the desired DL framework to use. Two concrete Encoder clients were implemented:

  1. **Encoder Torch Client** runs the Pytorch Encoder version, loading the model from the TorchScript Encoder NN script file. This implementation uses the LibTorch (a C++ Pytorch API) auxiliary library (see https://pytorch.org/cppdocs/frontend.html) to execute the NN from C++.
  2. **Encoder TensorFlow Client** runs the TensorFlow Encoder version, loading the model from the SavedModel Encoder NN script file. This implementation uses the LibTensorFlow for C auxiliary library from the TensorFlow official site (see https://www.tensorflow.org/install/lang_c) and CppFlow C++ lib (see https://serizba.github.io/cppflow/index.html) as a high-level wrapper to execute the NN from C++.

- After all workflow steps are complete, the ground station receives the binary output data. Then, the data is transformed and delivered to the decoder part of the corresponding CAE NN Model, which reconstructs a close approximation of the original images.

For the Neural-based SDO Image Compressor NN Model (check with Figure 8),

- In steps 1 and 2, the SDO Encoder NN Model Script is uploaded to the spacecraft. The TorchScript SDO Encoder NN script file comes from the Pytorch framework.

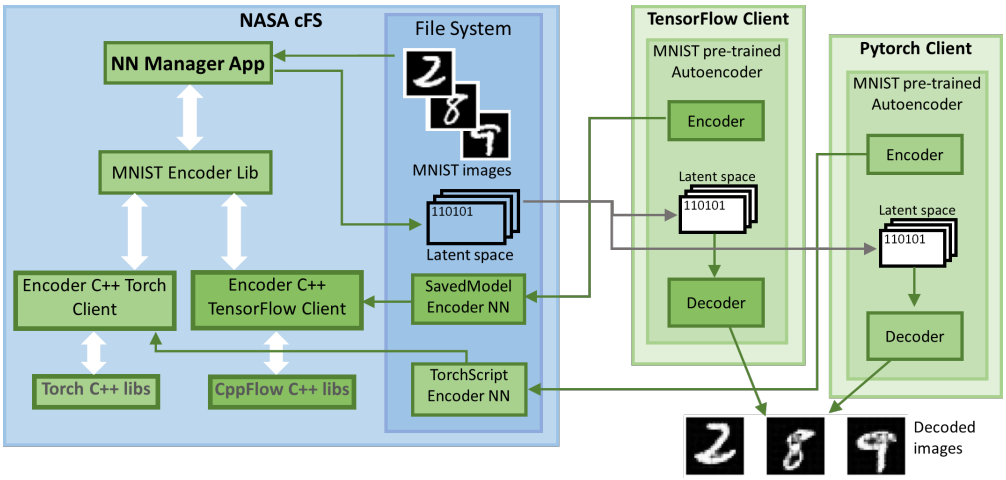

Figure 7: Framework Software Architecture for a Simple CEA NN

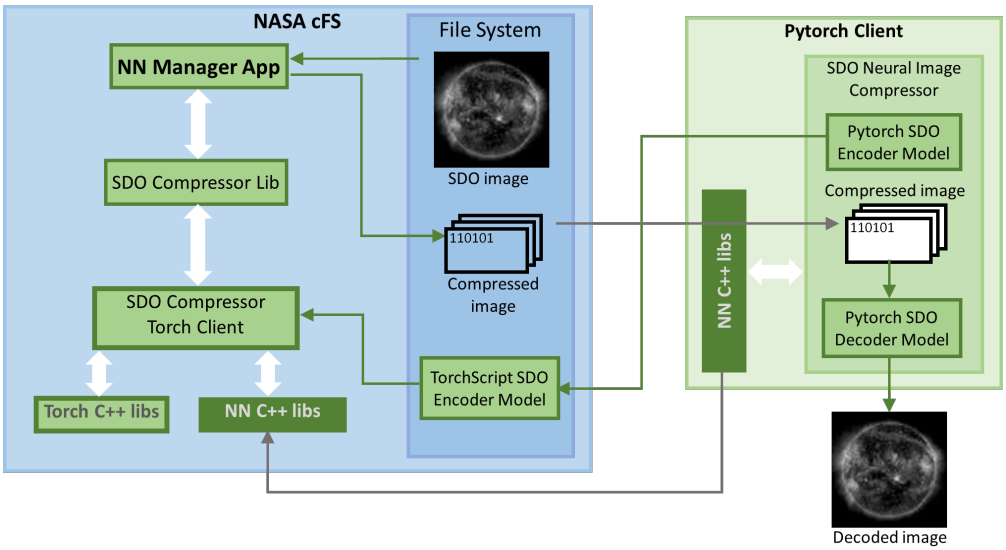

Figure 8: Framework Software Architecture for the NASA SDO Compressor NN

- For step 6, the user indicates to the NN Manager App that he wants to execute the SDO Compressor part of the NN Model. Also, he should set the appropriate dataset size, batch size, and inferred element memory reserved size.

- For steps 7 and 8, the abstract client would be **SDO Compressor Lib**, which is responsible for obtaining and executing the corresponding concrete SDO Compressor client according to the desired DL framework to use. Only one concrete SDO Compressor client was implemented: **SDO Compressor Torch Client** runs the Pytorch version, loading the model from the TorchScript SDO Encoder Model script file. This implementation uses the LibTorch (a C++ Pytorch API) auxiliary library (see `https://pytorch.org/cppdocs/frontend.html`) to execute the NN from C++. As we mentioned before, additional specialized C++ libraries are used inside the TorchScript SDO Encoder Model.

- After all workflow steps are complete, the ground station receives the binary output data. Then, the data is transformed and delivered to the SDO Decoder part of the SDO Neural-based NN Model, which reconstructs a close approximation of the original images.

Neural Networks Deployment and Execution

To demonstrate that the framework is suitable to execute all the NN Models running the NNEF in a realistic on-orbit hardware-software configuration, we set up a Raspberry Pi board, with a Linux Ubuntu OS. Also, we installed our NNEF on top of a basic cFS system distribution, including FM, DS, and CF additional apps, and we deployed all the shown NN models with the corresponding C/C++ clients and libraries. LibTorch and CppFlowLib are integrated into the operating system and dynamically loaded through the *cmake* configuration files. COSMOS application is executed on a regular computer to emulate the ground station. It was necessary to create the COSMOS configuration files to target the cFS commands and telemetry system. Also, to integrate all necessary setup for the CFDP engine and its PDU communications with cFS's CF app, and create the new commands and telemetry files for our new NN Manager App. The next table (Table 2) shows all computer devices and operating system specifications.

Table 2: Hardware and Operating Systems (OS) for Demonstration

Emulated Spacecraft specs	
Board:	Raspberry Pi 4 - Model B
Processor:	64-bit quad-core Cortex-A72
Memory:	8 GB RAM
Storage:	32 GB
OS:	Linux Ubuntu Desktop for RPi 23.10 (64-bit)
Emulated Ground Station specs	
Processor:	Intel(R) Core(TM) i7-10875H × 2
Memory:	8 GB RAM
Storage:	108 GB
OS:	Linux Ubuntu 23.04 (64-bit)

All NN Execution Framework code, the integrated NN Model applications and components, and the specific integration with cFS are available in the GitHub folder repository `https://t.ly/FX2AO` under the `NNEF` name. Table 3 describes all software applications or frameworks and their versions used to run our NNEF demonstration.

Table 3: NN Execution Framework Software Environment

Framework/Application	**Version**
Emulated Spacecraft	
NASA cFS bundle	draco-rc5
cFS: CFDP App	draco-rc5
cFS: DS App	draco-rc5
cFS: FM App	draco-rc5
LibTorch (C++ Pytorch API)	2.1.0
Libtensorflow (C TensorFlow API)	2.14.0
CppFlow (C++ TensorFlow API)	2.0.0
Emulated Ground Station	
Docker Engine	26.0.1
OpenC3 COSMOS	5.0.6

RESULTS AND CONCLUSIONS

Table 4: Encoder NN Performance Comparative

NN Model	Regular computer		Raspberry Pi	
	Latency (sec/image)	Throughput (images/sec)	Latency (sec/image)	Throughput (images/sec)
TF MNIST Encoder	3.3×10^{-3}	4,900	4.87×10^{-4}	2,051
PT MNIST Encoder	3.5×10^{-4}	6,500	5.13×10^{-4}	1,951
PT SDO Neural Compressor	1.67	0.79	5.02	0.20

PT: Pytorch, TF: Tensor Flow

1. Our NN Execution Framework was able to execute the inference process of all deployed NN models onboard a cFS-based spacecraft using a unified way to execute it, receiving the input data set and predicting the results. We demonstrate in this work that it is possible to integrate simple and complex TensorFlow and Pytorch-trained networks inside NNEF, and all of that can run on top of a space-like small hardware-software configuration.

2. We were able to send the inferred results files to the ground station. All results were correctly decoded and compared in their corresponding DL framework.

3. The performance of the inference process of all NN Models onboard a Raspberry Pi is shown in Table 4. And is compared with the performance we would get with the ground station regular computer specifications, but using the corresponding original DL Framework to execute exactly the same NN Models. As expected, we can see the performance in small hardware-software configurations is lower, but it is enough to execute NN Models with real-life complexity for space missions.

4. The MNIST Encoder works much faster than the SDO Compressor because the latter processes greater images and performs more complex and resource-demanding mathematical operations. For small hardware, the MNIST Encoder throughput is approximately 2000 images/sec. and with the SDO Neural Compressor, we can generate compressed representations at a rate of 1 image every 5 seconds.

Therefore, it could be concluded that the NNEF is a suitable framework to deploy and execute a large number of NN-based space applications, at least with less or equal complexity than the neural-based SDO image compression NN shown here. Facilitating the deployment and execution of the inference process using cFS as space flight software, that allows us to be agnostic of what operating system and hardware configuration are used. Moreover, demonstrating that this configuration can run in a small hardware-software setup, similar to the one currently used in real space missions.

FUTURE WORK

For the next stages of our project, we will be working on:

- Integrate and test more examples and more types of NN models to be applied in realistic space applications to ensure a broader NNEF DL applicability.
- Integrate the NNEF with more Space Flight systems, such as F Prime and others, to ensure we can integrate the NNEF into more space mission types and ensure cross-platform compatibility.
- Continue testing on several space-type hardware boards to integrate our NNEF with more complex hardware platforms, such as real-time OS and FPGAs.

References

[1] Stefano Silvestrini and Michèle Lavagna. Deep learning and artificial neural networks for spacecraft dynamics, navigation and control. *Drones*, 6(10):270, 2022. doi:10.3390/drones6100270.

[2] 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, 2020. doi:10.48550/arxiv.2001.10362.

[3] Corey OMeara, Leonard Schlag, and Martin Wickler. Applications of deep learning neural networks to satellite telemetry monitoring. In *2018 spaceops conference*, page 2558, 2018.

[4] Jianing Song, Duarte Rondao, and Nabil Aouf. Deep learning-based spacecraft relative navigation methods: A survey. *Acta Astronautica*, 191:22–40, 2022.

[5] Maciej Ziaja, Piotr Bosowski, Michal Myller, Grzegorz Gajoch, Michal Gumiela, Jennifer Protich, Katherine Borda, Dhivya Jayaraman, Renata Dividino, and Jakub Nalepa. Benchmarking deep learning for on-board space applications. *Remote Sensing*, 13(19):3981, 2021. doi:10.3390/rs13193981.

[6] Jacob Manning, David Langerman, Barath Ramesh, Evan Gretok, Christopher Wilson, Alan George, James MacKinnon, and Gary Crum. Machine-learning space applications on smallsat platforms with Tensorflow. 2018. https://digitalcommons.usu.edu/smallsat/2018/all2018/458/.

[7] Elizabeth Jean Timmons. Core Flight System (cFS) Training-cFS Caelum. Technical report, Goddard Space Flight Center, NASA, 2021. https://ntrs.nasa.gov/api/citations/20210022378/downloads/NASATM20205000691REV2Final.pdf.

[8] Jeffrey Levison. F prime flight software overview for small scale systems. Technical report, Pasadena, CA: Jet Propulsion Laboratory, NASA, 2018. https://hdl.handle.net/2014/49128.

[9] David Chelmins, Janette Briones, Joseph Downey, Gilbert Clark, and Adam Gannon. Cognitive communications for NASA space systems. In *Advances in Communications Satellite Systems: Proceedings of the 37th International Communications Satellite Systems Conference (ICSSC-2019)*. IET, 2021. doi:10.1049/pbte095e_ch16.

[10] Rich Landau. Evolution of nasa's core flight system (cfs) for the mission of tomorrow. In *Software for the NASA Science Mission Directorate Workshop 2024*, 2024. https://ntrs.nasa.gov/api/citations/20240005103/downloads/cFS_SMD_2024_FULL.pdf.

[11] GitHub - nasa/DS: The Core Flight System (cFS) Data Storage (DS) application. https://github.com/nasa/DS.

[12] CCSDS File Delivery Protocol (CFDP) - Part 1: Introduction and Overview, Green Book. Technical report, Consultative Committee for Space Data Systems, 2021. https://public.ccsds.org/Pubs/720x1g4.pdf.

[13] GitHub - nasa/CF: The Core Flight System (cFS) CFDP (CF) application. https://github.com/nasa/CF.

[14] GitHub.com - nasa/FM: The Core Flight System (cFS) File Manager (FM) application. https://github.com/nasa/FM.

[15] Li Deng. The MNIST database of handwritten digit images for machine learning research [Best of the Web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. doi:10.1109/MSP.2012.2211477.

[16] Ali Zafari, Atefeh Khoshkhahtinat, Piyush M Mehta, Nasser M Nasrabadi, Barbara J Thompson, Daniel Da Silva, and Michael SF Kirk. Attention-based generative neural image compression on solar dynamics observatory. In *2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 198–205. IEEE, 2022.