Rapid Prototyping of Deep Learning Models on Radiation Hardened CPUs.

P. Blacker
University of Surrey
Guildford, UK
P.Blacker@Surrey.ac.uk

C. P. Bridges
University of Surrey
Guildford, UK
C.P.Bridges@Surrey.ac.uk

S. Hadfield
University of Surrey
Guildford, UK
S.Hadfield@Surrey.ac.uk

***Abstract*—Interest is increasing in the use of neural networks and deep-learning for on-board processing tasks in the space industry [1]. However development has lagged behind terrestrial applications for several reasons: space qualified computers have significantly less processing power than their terrestrial equivalents, reliability requirements are more stringent than the majority of applications deep-learning is being used for. The long requirements, design and qualification cycles in much of the space industry slows adoption of recent developments.**

GPUs are the first hardware choice for implementing neural networks on terrestrial computers, however no radiation hardened equivalent parts are currently available. Field Programmable Gate Array devices are capable of efficiently implementing neural networks and radiation hardened parts are available, however the process to deploy and validate an inference network is non-trivial and robust tools that automate the process are not available.

We present an open source tool chain that can automatically deploy a trained inference network from the TensorFlow framework directly to the LEON 3, and an industrial case study of the design process used to train and optimise a deep-learning model for this processor. This does not directly change the three challenges described above however it greatly accelerates prototyping and analysis of neural network solutions, allowing these options to be more easily considered than is currently possible.

Future improvements to the tools are identified along with a summary of some of the obstacles to using neural networks and potential solutions to these in the future.

***Index Terms*—LEON, TensorFlow, on-board, Planetary rover, CNN, deep learning, autonomy.**

I. Introduction

Recent advances in the science and application of neural networks have been largely driven by improvements in tools, both hardware and software, used to design and deploy these solutions. Powerful frameworks now exist to design and train network models as well as automated tools to deploy trained inference models on a range of hardware targets. However these tools are aimed at terrestrial hardware; desktop computers, mobile phones, and embedded computers. These targets are significantly more powerful than the current generation of radiation hardened space processors, LEON 3 & RAD 750 [2], [3], meaning these tools are not able to target these processors.

Especially early in the life-cycle of a space mission when low TRL (Technology Readiness Level) feasibility studies are being undertaken, the lack of automated development tools increases the difficulty of evaluating potential deep-learning options. Tools to quickly design and train networks do exist, however analysts also need to determine upper-bounds of the CPU time and memory resources required by the model. At a time of growing interest in the use of deep learning on-board space missions there is a need for tools to make their analysis and implementation on standard space processors as straightforward as possible. It is hoped the TFMin library will enable early analysis of deep-learning solutions during the requirements stage, encouraging their adoption in final spacecraft designs.

TFMin has been developed during our own work investigating the use of deep-learning for processing on-board Mars rovers, and serves to address this gap in the tools which are available. Network design and training has been conducted using the TF (TensorFlow) framework on desktop computers with powerful GPUs, the TFMin library has been used to extend these tools so we can automatically deploy our trained inference networks onto the LEON3 platform and analyse their memory and CPU requirements. This has allowed us to quickly analyse different network topologies and the effects of changes to the processor such as instruction and data cache sizes.

Our work focusses on the LEON family of SparcV8 processors, however the TFMin tool can be used with a wider range of targets, it generates C++11 code which is dependant upon the Eigen linear algebra library [4]. This means it can be deployed on any target platform that has a C++11 compliant compiler including BAE's RAD 750 space processor as well many smaller processors commonly used on nano and micro satellites. During testing small TF models have been successfully deployed to the Teensy 3.2 development board (ARM Cortex-M4)[5], from this we conclude it may be possible to deploy similar models to a wide range of small embedded computers including the low cost ATMEL PA32 radiation hardened processor which has flight heritage or the low cost Vorago rad-hard ARM Cortex-M0 [6]. We intend for TFMin to be a valuable tool for OBDH developers enabling them to quickly generate accurate performance metrics for neural networks on a wide range of processors used in space, as well as generating prototype C++ code which can be used as a starting point for the development of actual flight software.

This paper summarises work on the use of neural network models in space applications and the development of tools to automatically deploy these networks onto a range of hardware

targets. We then introduce the TFMin library and describe its architecture and the internal optimisations that it uses. Finally we describe the terrain evaluation task we have used the library for, as a design and optimisation case study for deep learning solutions on the LEON 3 processor.

Deep learning inference networks have become a common part of terrestrial systems today, large continuously trained models generally reside inside server clusters. Increasingly however frozen inference models are being deployed onto mobile and embedded devices. TF officially supports two frameworks for the deployment of trained inference models, TensorFlow-Lite, and the TensorFlow XLA (Accelerated Linear Algebra) AOT (Ahead of Time) compiler. TensorFlow-Lite is a cut down version of the full TF library aimed at mobile phone grade computers, as such it is still too large for many embedded systems having a library overhead of more than 100MB. This framework has been used by [7] [8] to successfully process actual orbital imagery on a representative flight proven single board computer. It should be noted however that the more powerful Zynq chip (dual core Arm Cortex-A8) used to execute the inference was not rad-hard, it was just protected by the less powerful rad-hard components on the CHREC Space Processor v1 board, limiting this application to lower radiation orbits near Earth or short duration missions. The only rad-hard components with a similar level of computing power are the latest generation of processors (RAD 5500 [9], LEON 4 [10]) which are currently lacking flight heritage and come at a high price. As such TensorFlow-Lite deployment is currently only feasible for LEO missions and High-risk, high-value GEO and deep space missions.

TensorFlow XLA is an LLVM (Low Level Virtual Machine) compiler tool which generates machine code directly from a trained model, the resulting compiled static-libraries can be linked into larger projects with minimal library over heads. This makes it ideally suited to deploy NN (Neural Network) models on radiation hardened processors, due to it's use of the versatile LLVM compiler architecture [11] it is able to produce code for a range of CPU targets. The Sparc V8 architecture of the LEON family of processors is not officially supported by XLA and after some initially promising results the authors failed to successfully integrate it with the Gaisler LLVM compiler [12]. Gaisler's Clang compiler is built on LLVM version 4.0 while the TF XLA compiler generates IR (Intermediate Representation) using features only present in versions 5.0 and above. By taking the LLVM IR produced by the XLA compiler then building it using the Gaisler clang compiler simple fully connected networks were successfully deployed to the LEON 3. However the authors did not manage to compile working versions of large or convolutional networks using this technique. The XLA compiler is considered experimental by the TensorFlow team and it cannot be simply linked to the Gaisler tool chain however this may change in the future. Another drawback of using XLA is the challenge of integration with aerospace V&V (Verification & Validation) practices, since it generates machine code directly its output is difficult to inspect. It also doesn't produce human editable code which could be used as a basis for developing formal flight software, so a different solution is preferable for Aerospace applications.

MatLab's ML (Machine Learning) toolbox provides two options for deploying trained inference models onto embedded systems; MatLabs C coder generates equivalent C++ code while MathWorks HDL (Hardware Definition Language) Coder can deploy models directly to FPGAs (Field Programmable Gate Arrays). Code automatically generated from MatLab has significant runtime library dependencies of a size comparable to TensorFlow-Lite making it appropriate for the same range of hardware targets. MathWorks HDL Coder although targeting lower level hardware than CPUs is an interesting new option for analysing deep-learning models on space qualified hardware. FPGA implementations of DL models have the potential to be of significantly higher performance than CPU implementations. This approach is still an active area of research with open questions that need to be answered.

In addition to these tools there is always the option of manually implementing a NN model on any platform and for smaller networks this may be a practical option. The fully-connected MNIST classification example model [13] included with TFMin is made of seven operations so could be easily hand coded, SqueezeNet on the other hand is made of 93 operations while other deep learning models have many hundreds. Importantly these deeper models are often the most efficient way to solve complex problems and given the need to iterate the design of DL models manual implementation becomes increasingly impractical.

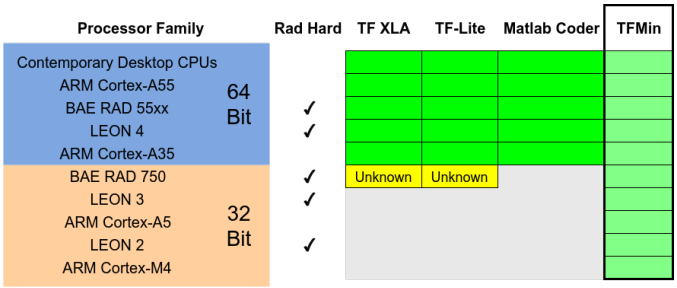

Fig. 1. Common terrestrial and space processors and the tools able to automatically deploy neural networks to them.

Figure 1 shows a range of terrestrial and space qualified CPU processors and which automated NN deployment tools they are supported by. Unsurprisingly recent mobile phone grade processors, which includes the next generation of radiation hardened devices, are well supported, being the main target of these tools. We can see that support for the current generation of radiation hardened devices is lacking, which is critical because these processors are and for a few years will continue to be the most common choice for new space craft. There is a need for a new tool to fill this gap for two reasons, TF XLA compiler does not currently support the LEON family of processors, and even when it does its use does not fit into current coding standards for flight software. We present the TFMin which tool addresses both of these limitations by

avoiding the need for XLA and providing validated C++ code which can then be passed onto a flight software development team.

II. TFMin - Generating C++ from TensorFlow Models

The TFMin tool automatically converts trained NN models developed using TF into verifiable C++ code which can then be built for any system with a C++11 compliant compiler, including the LEON family of processors used on many geostationary and deep space missions. Being able to quickly deploy and analyse DNNs on computers representative of flight hardware is a valuable tool allowing developers to relate high level network changes to low level performance without the need to manually implement the network model. This conversion is done verbatim so the C++ version is mathematically identical to the TF original, any quantisation and retraining optimisation would need to be done in TF before the model is converted. Additionally unlike the LLVM intermediate representation generated by the XLA AOT compiler, TFMin generates human readable C++ code which can be independently verified and used as a starting point for formal flight code development.

TF is a functional programming tool hiding within a procedural python/C++ framework, neural network models are created as large functions known as flow-graphs. These flow-graphs are networks of functions which transfer n-dimensional tensors between each other, hence the name TensorFlow. The power of TF is its capability to automatically accelerate the evaluation of these flow-graphs using massively parallel GPUs, which is essential for training and in many cases inference of large DNNs (deep neural networks).

Since DNNs are represented as functional flow-graphs they are mobile data structures as opposed to an algorithm that is hard-coded into source code. A python program can procedurally generate a flow-graph however they can also save, load, optimise, and convert them. This capability allows the TFMin library to extend TF with the capability to convert DNN flow-graphs into equivalent C++ code.

When TF evaluates flow-graphs the required output tensors are specified, these outputs are then traced backwards through the network to determine which operations need to be evaluated to compute the solution. Because of this a single flow graph can be used for both inference and training, where evaluating for training accuracy output will execute a batch training run while evaluating for the estimate output will execute an inference run.

TFMin uses the same approach, a flow graph and set of output tensors is provided and the library works out which operations, weights, and inputs are required, ignoring any training or introspection operations. Model weights are written into a C++ header file as array literals, while the required operations from the flow-graph are built into a C++ object as shown in Figure 2. As well as a standard inference method two optional methods can be generated for verification and timing. Verification methods include input data along with the expected results of each operation allowing the generated C++ model to validate itself against the TF original. Timing methods can be used the same way as the standard inference method while additionally providing total and operation level execution times.

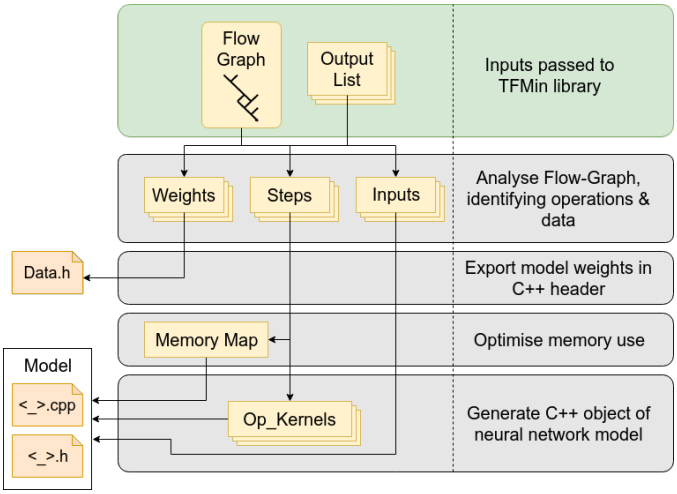

Fig. 2. Architecture of the TFMin TensorFlow to C++ code generator.

TF operations are converted into C++ code using a dictionary of *op_kernels*, these are objects which match themselves to TF operations and generate equivalent code. The list of these *op_kernels* has been designed to be peripheral to the core library so it can easily be extended and optimised by developers to fine tune its code output for their particular projects.

A. Adding TFMin Exporter to a Project

There are two parts to the TFMin library a Python module used to analyse flow graphs and convert them to code, and a C++ header only library containing the virtual base-class that generated model classes are derived from. Integrating the library with an existing TF project in python has been made as simple as possible. Listing 1 shows how an existing TF model can be exported to C++ using only two lines of python code.

Listing 1. Python example exporting a trained model from TensorFlow to C++.

```
import tf_min as tfm

# Setup flow-graph and
# load or train model weights
# ...

gen = tfm.Exporter(flow_graph,
                   ['output0',
                    'output1'])

gen.generate("cnn_model",
             "cnnModel",
             layout='RowMajor')
```

The TF *flow_graph* object and a list of output tensor names is all that is needed, running the code above will generate three files: *cnn_model_data.h* containing the model weights, and *cnn_model.cpp* & *cnn_model.h* which define an object encapsulating the NN model itself.

These three files can then be included in a larger C++ application or library project. The interface has again been designed to be as simple as possible, data is passed to and from the model object using raw buffers in the layout specified in the python code above. Listing 2 shows a minimal integration with the SqueezeNet example model; First an Eigen device object is created which defines how the model will be evaluated, in this case a single thread device will be used, multi-threaded devices are also supported. Secondly the model is instantiated, causing required heap memory to be allocated. Thirdly the model is evaluated using the given device, input and output buffers are passed as pointers. Finally heap memory is released when the model instance goes out of scope.

Listing 2. Example C++ project integration with TFMin generated code

```
#include <iostream>
#include "cnn_model.h"

using namespace std;

int main()
{
  float input[154587];
  float est_class[1000];

  Eigen::DefaultDevice device;
  SqueezeNet model;

  cout << "Infering with Model.\n";
  model.eval(device, input, est_class);
  cout << "Inference complete.\n";

  return 0;
}
```

B. Speed Analysis & Optimisation

As well as the standard eval() method shown above two optional methods can be generated to analyse and verify the generated code.

The following section describes how to export additional methods for the generation of per-operation timing information and measurement of a models memory requirements. Full tutorials and examples can be found on the git repository wiki pages [14]. Listing 3 shows how to export these two optional methods, it should be noted that enabling full validation can significantly increase the size of the generated binary since the valid result of every operation needs to be stored.

The timing method performs the same task as *eval()* while additionally reporting the time each operation takes to execute. These results are printed to the terminal and returned in a data structure for automatic recording. The validation method prints either a pass or fail for each operation as the model is being evaluation, terminating in the case of failure. This is especially useful for developers that are optimising or extending the operations supported by the library as described in section II-D

Listing 3. Python example exporting a trained model with timing and validation methods.

```
val_input = [ ... ]

gen.generate("cnn_model",
             "cnnModel",
             validation_inputs =
                {input_placeholder:
                          val_input},
             validation_type='Full',
             timing=True,
             layout='RowMajor')
```

Figure 3 shows the timing results of the convolutional MNIST example network on a 50 MHz LEON 3 [13], this tool can be used to identify which parts of a network topology have the greatest scope for being accelerated. In this simple CNN model the costliest operations are the two convolution filters and the largest fully connected layer as would be expected. These operations predominantly use multiply and accumulate instructions, accessing regions of memory larger than the cache size of the processor. Using the TSim simulator it was found that the cache hit rates were 70% and 82% for the convolution and matrix multiplication operations respectively. These are very poor hit rates and indicate that the speed of execution is being limited by the time taken to access data in RAM. The 4KB data cache size of the simulated LEON processor used is a significant factor in the time taken to calculate these layers. The pattern of memory use also has a large effect on cache hit-rates and therefore performance, using a row-major layout results in this classifier executing in 68% of the time it took to execute the same model using a column-major layout.

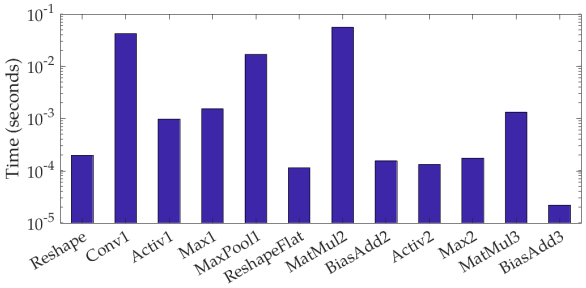

Fig. 3. CPU time per operation of a convolutional MNIST classification model executed on a 50 MHz LEON 3.

Optimisation of Neural network training and inference on different types of hardware is an ongoing research topic, this includes the single or multi core CPU architectures of the LEON family. This work is not trying to push these boundaries, but it is important that the algorithms used were

representative of current CPU implementations so that our performance results are realistic. This was achieved by basing our code on TensorFlow's own open source C++ implementation which use the Eigen linear Algebra library [4]. TFMin could be extended by the development of layer implementations that are optimised for specific hardware, such at the work of Lai et. al.[15] who created a library of optimised ARM Cortex-M NN layer implementations (CMSIS-NN). Combining the automatic code generation framework provided by TFMin with a set of hardware specific layer implementations would combine the fast work-flow of TFMin with state of the art layer implementations.

Using Eigen to perform the fundamental tensor operations allows the library to take advantage of verified and optimised algorithms. The most CPU intensive operation in most DNNs is convolution, TFMin implements this by reshaping the input and filter tensors so that the convolution can be calculated using tensor contraction. The particular contraction used is a matrix multiplication allowing Eigen's optimised GEMM algorithms to be used.

C. Memory Analysis & Optimisation

Measuring memory use on LEON 3 computers is more challenging than on fully featured desktop-systems, however requirements on both types of computer would be expected to be close to each other. For this reason, we recommend using the valgrind tool *massif* to measure the memory use of a DNN model by compiling it for a desktop target and running it natively, it is important to note that both heap and stack memory need to recorded to get an accurate measurement. Taking SqueezeNet v1.1 [16] as an example we can see in Figure 4 that the model requires an approximately constant amount of memory to run with a peak of 7.7 MB.

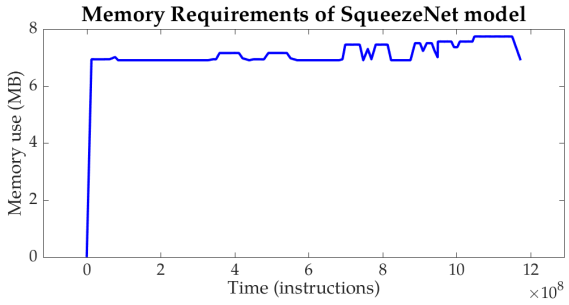

Fig. 4. Combined heap & stack memory use of the TFMin implementation of SqueezeNet v1.1, during an inference operation.

The result shown in Figure 4 includes the memory optimisation step performed by the TFMin library. Memory requirements of most NN models are fixed which means it is possible to pre-calculate an optimal memory use pattern removing the need for dynamic memory management during its evaluation. This optimisation is performed by finding the first assignment and final use of each tensor and the amount of memory needed to store it. Each of these blocks defines a block of memory covering a specific duration in the calculation, a packing algorithm is used to fit them into the smallest fixed sized block of memory, Figure 6. Comparing SqueezeNets optimised pattern against the original pattern, the peak memory requirement has reduced to 20% of the original, Figure 5.

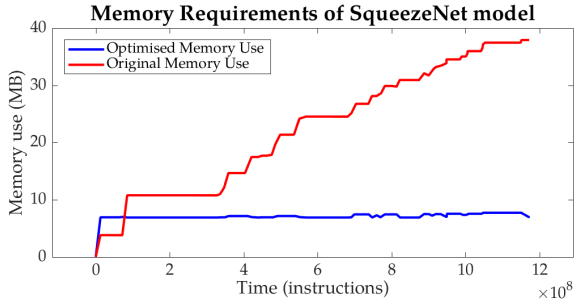

Fig. 5. Combined heap & stack memory use of SqueezeNet v1.1 before and after optimisation.

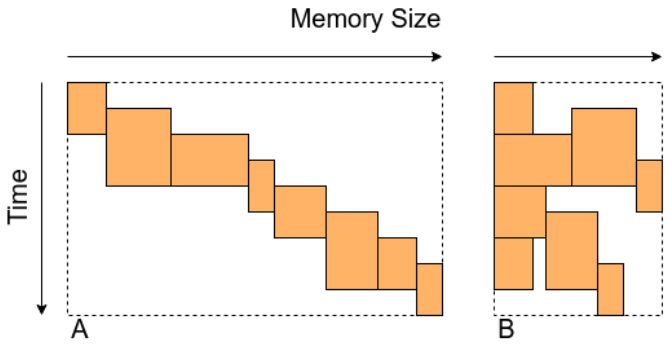

Fig. 6. Tensor memory use patterns before (A) and after optimisation (B).

D. Supported Operations

TFMin uses a set of op_kernel objects which match themselves to TF operations and generate equivalent C++ code, the library currently has 19 op_kernels supporting the most common operations. This is a fraction of the approximately 800 operations available in the full TF framework, but is sufficient to implement all the networks described in this work. It is inevitable that developers will need to extended this set of operations to work with a wider range of network models. An extensible architecture has been used allowing new op_kernels support by adding separate python files to the op_kernels folder.

An example and walk through of this process is included in the *adding_an_operation* tutorial on the git repository wiki [14].

E. Benchmarking CNNs on the LEON 3

Using the analysis techniques described above several well known tasks and network models have been built for the LEON 3, these results provide a valuable guide to what is possible on this generation of radiation hardened processors. Several larger networks were investigated, AlexNet [17] and

TABLE I
EXECUTION SPEED AND MEMORY REQUIREMENTS OF NOTABLE DNN MODELS ON A LEON 3 PROCESSOR.

Model	Peak RAM (KB)	Time (s)
MNIST Dense (float 32)	82.7	0.100
MNIST CNN (float 32)	200	0.121
SqueezeNet v1.1 [16]	7884	41.0

ResNet-50 [18] but were found to be too large to implement on the current LEON 3 development boards available to us.

C++ code generated by TFMin was succesfully built using both the gcc & llvm based compilers provided by Gaisler Aeroflex [12] no significant difference in execution speed was found between the two compilers. All evaluation binaries have been built with maximum optimisation for speed and no debugging symbols. Compiler optimisation is especially important for the Eigen library which has been designed to be built this way, a 20x speed increase was observed between maximum (-O3) and no optimisation.

- All LEON times were generated using a 50 MHz processor with hardware floating point unit and 16 KB data and instruction cache sizes.
- MNIST classifier results can be generated using the free evaluation version of TSIM (LEON simulator) available from Gaisler Aeroflex.
- SqueezeNet result was generating using a Pender GR-XC3S-1500 development board with a Xilinx Spartan FPGA and 64 MB of SDRAM [19].

III. PLANETARY TERRAIN ASSESMENT, USE CASE

A. The Estimation Task

Determining which areas of terrain are safe and unsafe to drive over is an essential function of planetary rovers and terrestrial autonomous ground vehicles [20]. This experiment investigates the use of a deep learning model for this task and compares its accuracy and performance to an existing baseline algorithm. A cost mapping algorithm based upon the GESTALT navigation system used on the MER (Mars Exploration Rovers) [21] is used as this baseline, its input is a set of elevation values from within a rover footprint and it produces a scalar navigation cost metric. These cost values lie between zero and one, where zero is impassible and one is the easiest possible terrain. This algorithm is repeated across the DEM (digital elevation map) to generate a navigation cost map, Figure 7, which is subsequently used by a path planning algorithm to find a safe and efficient route to the chosen destination [20].

A deep learning regression model fits the inputs and outputs of the baseline algorithm, taking a fixed size input matrix and estimating a single scalar. The existing baseline algorithm allows us to automatically generate large sets of training data from any existing DEM, this experiment used a 600 x 600 metre map generated from drone data with an 8cm cell size, taken from the ERGO [22] [23] Morocco field trial. A set of 36 million training cost values were generated from this

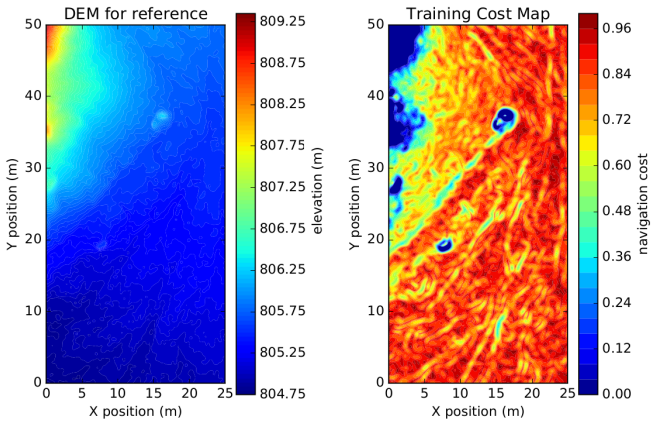

Fig. 7. (left) Digital elevation map taken from the ERGO Morocco dataset, (right) cost map of navigability values.

map which was then augmented using rotations and reflections increasing it eight times to 288 million. A Huber loss [24] function was used with the ADAM optimiser [25] and a variable learning rate to refine the weights of the model during training.

During initial testing a problem was found with this training data, the distribution of navigation costs was not uniform with a bias towards easy terrain. The makes sense when considering that the majority of the map is fairly flat, however this training data bias prevented the model from fitting in areas containing difficult terrain. Splitting the training data into ten sets with evenly distributed navigation costs then selecting samples equally from each of these sets removed this bias, so that during training a uniform distribution of cost values is used. This training methodology, cost function and training data was then used to evaluate a range of CNN topologies.

B. Analysis of DNN Models

We present five different DNN models which were trained as described in Section III-A and compare their performance to a LEON 3 implementation of the baseline terrain assessment algorithm. LEON 3 performance metrics were generated using a 50 MHz processor with hardware floating point unit and 16 kB instruction and data caches. All models were trained on the dataset for 200k steps with a batch size of 100, taking approximately two hours per model using an NVIDIA GTX 1070 GPU.

After considering many different CNN topologies, five models are presented with a range of layer counts and filter sizes. Model A was first to successfully fit to the training data Figure 9, with 5 convolution layers and three fully connected layers, Figure 8 shows the cost map generated using this model and the error between its estimates and the original training data. This network topology was then gradually reduced in size, decreasing the number of layers and reducing the size of the convolutional filters until it was no longer capable of fitting to the training data. Models B to E represent the range of the network topologies that were able to fit to the training data,

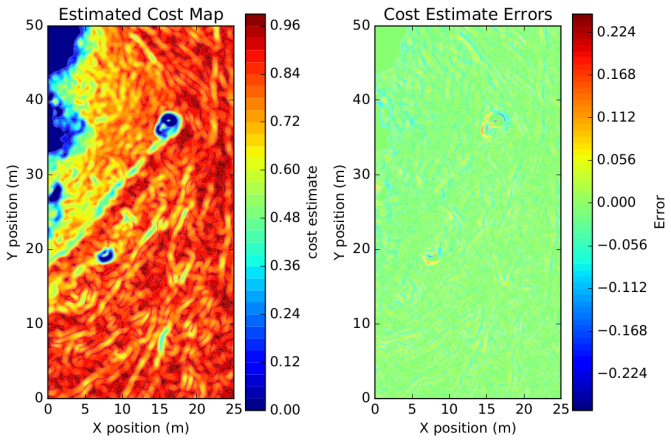

Fig. 8. (left) DNN generated cost map from ERGO DEM data, (right) residual error between DNN model and training data.

TABLE II
TERRAIN ASSESSMENT MODEL, ACCURACY AND PERFORMANCE RESULTS ON A LEON 3.

	A	B	C	D	E
Weights (kB)	1130	1094	300	154	110
Mean Error	.0194	.0186	0.0198	.0206	.0199
Runtime (s)	8.34	2.03	1.09	0.39	0.18
Baseline ratio	87.6	21.4	11.5	4.1	1.9
Binary (kB)	11371	9807	9496	8868	8815
Memory (kB)	884	595.5	360	286.4	252.3

although there is a slight trend of increasing residual errors as the networks become smaller.

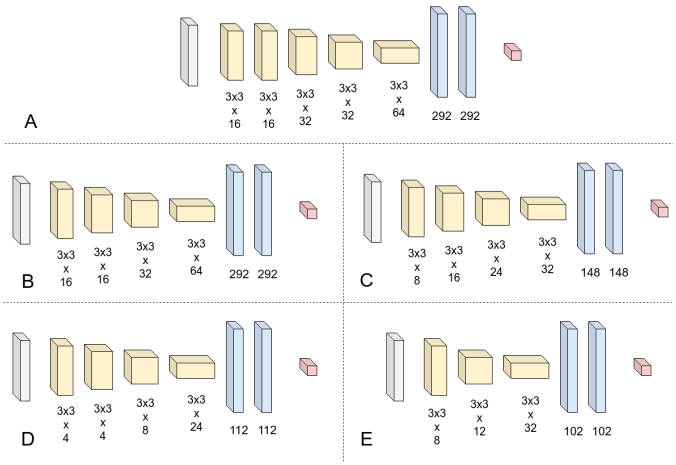

Fig. 9. Terrain traversability estimation models A - E.

The baseline terrain assessment algorithm used calculates three geometric properties which are then combined into the final navigation cost; the slope of the best fit plane, the largest distance between this plane and any point on the terrain, and the greatest height difference between any adjacent DEM cells. The most computationally intensive task is the calculation of the best fit plane, a singular value decomposition (SVD) is performed using the bidiagonal divide-and-conquer algorithm

TABLE III
TERRAIN ASSESSMENT MODEL PERFORMANCE RESULTS ON AN INTEL I7.

	A	B	C	D	E
Time (s)	.0404	.0060	.0040	.0015	.0009
Baseline ratio	336.8	49.9	33.4	12.8	7.3

[26], this produces the principal axes of a best fit ellipsoid, the smallest eigen value will correspond to the axis matching the planes normal. This algorithm takes 95 milliseconds to execute on the LEON 3 CPU described in table II.

C. Use Case Findings

As can be seen in Table II the baseline algorithm is still outperforming the DNN model by a factor of two even in the best cases. This result suggests that deep learning is not capable of estimating navigation cost maps more efficiently than the existing baseline algorithm. Future work will develop this approach using fully convolutional networks which estimate multiple cost map elements at the same time, potentially producing several square metre map in a single pass. Using this technique, it is possible to surpass the performance of the baseline algorithm when larger areas of cost map need to be generated at once.

The value of TFMin to generate accurate LEON 3 timing results for these models is highlighted, when the same comparison between algorithms is made on an Intel i7 CPU, shown in Table III. The relative speeds of the two algorithms on the i7 are significantly different than on the LEON 3, caused by the nature of each algorithm interacting differently with the underlying CPU architecture and resources of each system.

There are many potential reasons for the observed difference in relative performance between the two systems. The i7 level two cache is larger than the entire DNN model, while the LEON 3 has no level two cache and its level one cache cannot store even a single layer of weights. The i7 also supports vectorised FMA (Fused Multiply-Add) operations allowing it to execute significantly more floating point operations per clock cycle than the simpler FPU of the LEON 3. This is why it is important to analyse the performance of an algorithm on the same CPU architecture that it will run on in production, and why TFMin can be a valuable tool for reviewing DNN model performances early in the development cycle.

IV. CONCLUSIONS AND DISCUSSIONS

A new tool has been demonstrated enabling developers to quickly deploy and evaluate TensorFlow models on small resource constrained processors such as the LEON 3. LEON 3 benchmarks for several well known deep learning models [Table I] have been produced using this tool. The workflow and integration of this tool into an existing TensorFlow project has been described, and the methods to extract meaningful analyses from the deployed networks. Finally a simple use-case has been shown which used the TFMin tool to quickly analyse a range of different network models, highlighting why it is important to compare algorithms on target hardware as opposed to the more easily available desktop systems.

Our terrain assessment use case has shown that a conventional regression CNN can accurately reproduce the output of a traditional algorithm, although taking 1.9 times longer to execute. Since only a small improvement in the performance of the CNN model would make it surpass the baseline algorithm, our future work will focus on this area. However if a more efficient deep neural network is found there are still significant challenges to their adoption on-board planetary rovers. Aerospace verification and validation practices do not cover the use of deep neural networks, so we will investigate the use of a novel oversight system to address this gap.

Eight significant deep learning models have been built and analysed on the LEON 3 platform using TFMin, however there are still many improvements that could be made to this tool. Model optimisation and quantisation in TF is available as part of the TFlite module, with many of the standard layer operations only supporting floating point data. During this work experimental micro controller support was added to the TF lite framework, the authors are currently adding LEON support to this tool and will investigate the performance of these implementations as compared to those from TFMin. Although the Eigen tensor operations used are generically optimised, they are not optimised for the LEON3. Producing a set of op_kernels which generate LEON 3 optimised code would be necessary to to achieve the best performance possible on this target. Finally support still needs to be added for additional TensorFlow operations, it is hoped that the open-source community will be able to help with this task.

Airbus deep-learning group is currently investigating the use of this tool, it is hoped that its open-source licence will encourage its use more widely in the space industry.

ACKNOWLEDGMENT

The author would like to thank Matthias Winter for originally suggesting the CNN approach to navigability assessment during his time working on the Exomars rover. We would also like to thank the Airbus Defence and Space AOCS/GNC department at Stevenage for their assistance during this work and the ERGO project and PERASPERA cluster for providing access to the Morocco dataset.

This work has been sponsored by Airbus.

REFERENCES

[1] T. Campbell, "A deep learning approach to autonomous relative terrain navigation," 2017.

[2] G. Aeroflex, "Dual-core leon3-ft sparc v8 processor," http://www.gaisler.com/doc/gr712rc-datasheet.pdf.

[3] R. W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley *et al.*, "The rad750/sup tm/-a radiation hardened powerpc/sup tm/processor for high performance spaceborne applications," in *2001 IEEE Aerospace Conference Proceedings (Cat. No. 01TH8542)*, vol. 5. IEEE, 2001, pp. 2263–2272.

[4] G. Guennebaud, B. Jacob *et al.*, "Eigen," *URl: http://eigen. tuxfamily. org*, 2010.

[5] PJRC, "Teensy 3.2 & 3.1 - new features," https://www.pjrc.com/teensy/teensy31.html.

[6] V. Technologies, "Vorago producs," https://www.voragotech.com/vorago-products.

[7] J. Manning, D. Langerman, B. Ramesh, E. Gretok, C. Wilson, A. George, J. MacKinnon, and G. Crum, "Machine-learning space applications on smallsat platforms with tensorflow," in *Proceedings of the 32nd Annual AIAA/USU Conference on Small Satellites, Logan, UT, USA*, 2018, pp. 4–9.

[8] A. D. George and C. M. Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, 2018.

[9] B. Systems, "Bae systems current processors and single board computers," https://www.baesystems.com/en-us/download-en-us/20190124214317/1434554723043.pdf.

[10] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the ngmp multi-core processor in the space domain," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 175–184.

[11] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[12] J. Gaisler and K. Eisele, "Bcc-bare-c cross-compiler users manual," *Aeroflex Gaisler AB*, 2012.

[13] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[14] P. Blacker, "Github repository of the tfmin code generation tool," https://github.com/PeteBlackerThe3rd/TFMin.

[15] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[16] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[19] Pender, "Pender grxc3s," http://www.pender.ch/products_xc3s.shtml.

[20] M. Winter, C. Barcaly, V. Pereira, R. Lancaster, M. Caceres, N. McManamon, N. Silva, D. Lachat, and M. Campana, "Exomars rover vehicle: detailed description of the gnc system," *ASTRA*, 2015.

[21] M. Maimone, A. Johnson, Y. Cheng, R. Willson, and L. Matthies, "Autonomous navigation results from the mars exploration rover (mer) mission," in *Experimental robotics IX*. Springer, 2006, pp. 3–13.

[22] E. Consortium, "European robotic goal-oientated autonomous controller (ergo)," https://www.h2020-ergo.eu/.

[23] R. Marc, P. Wclewski, and D. Lachat, "Autonomous multi-mode rover navigation for long-range planetary exploration using orbital and locally perceived data," 10 2018.

[24] S. Klanke and H. Ritter, "Variants of unsupervised kernel regression: General cost functions," *Neurocomputing*, vol. 70, no. 7-9, pp. 1289–1303, 2007.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the bidiagonal svd," *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 79–92, 1995.