Keras2c: A library for converting Keras neural networks to real-time compatible C

Rory Conlin[a,*], Keith Erickson[b], Joeseph Abbate[c], Egemen Kolemen[a,b,*]

[a] *Department of Mechanical and Aerospace Engineering, Princeton University, Princeton NJ 08544, USA*
[b] *Princeton Plasma Physics Laboratory, Princeton NJ 08544, USA*
[c] *Department of Astrophysical Sciences at Princeton University, Princeton NJ 08544, USA*

Abstract

With the growth of machine learning models and neural networks in measurement and control systems comes the need to deploy these models in a way that is compatible with existing systems. Existing options for deploying neural networks either introduce very high latency, require expensive and time consuming work to integrate into existing code bases, or only support a very limited subset of model types. We have therefore developed a new method called Keras2c, which is a simple library for converting Keras/TensorFlow neural network models into real-time compatible C code. It supports a wide range of Keras layers and model types including multidimensional convolutions, recurrent layers, multi-input/output models, and shared layers. Keras2c re-implements the core components of Keras/TensorFlow required for predictive forward passes through neural networks in pure C, relying only on standard library functions considered safe for real-time use. The core functionality consists of ~ 1500 lines of code, making it lightweight and easy to integrate into existing codebases. Keras2c has been successfully tested in experiments and is currently in use on the plasma control system at the DIII-D National Fusion Facility at General Atomics in San Diego.

1. Motivation

TensorFlow[1] is one of the most popular libraries for developing and training neural networks. It contains a high level Python API called Keras[2] that has gained popularity due to its ease of use and rich feature set. An example of using Keras to make a simple neural net is shown in Listing 1. As the use of machine learning and neural networks grows in the field of diagnostic and control systems [3] [4] [5] [6], one of the central challenges remains how to deploy the resulting

*Corresponding author
Email addresses: wconlin@princeton.edu (Rory Conlin), kerickso@pppl.gov (Keith Erickson), jabbate@princeton.edu (Joeseph Abbate), ekolemen@princeton.edu (Egemen Kolemen)

trained models in a way that can be easily integrated into existing systems, particularly for real-time predictions using machine learning models. Given that most machine learning development traditionally takes place in Python, most deployment schemes involve calling out to a Python process (often running on a distant network connected server) and using the existing Python libraries to pass data through the model [7] [8] [9]. This introduces large latency and is generally not feasible for real-time applications. Existing methods for compiling Python code into C [10] [11] generally require linking in large libraries that are neither deterministic nor thread-safe. Recently, there has been work in methods that allow neural networks to be imported into C/C++ programs without the use of Python such as TorchScript in Pytorch [12] or Frugally Deep [13] for Keras. Both of these libraries resolve some of the limitations of previous methods by not relying on network connections, but in both cases still rely on sizeable external libraries such as Eigen [14] for the underlying computation, and they generally do not result in deterministic behavior and are not safe for real-time use.

```
import tensorflow.keras as keras

model = keras.models.Sequential()
model.add(keras.layers.Conv2D(filters=5, kernel_size=(2,2),
                              padding='same', activation='relu',
                              input_shape=(8,8,2)))
model.add(keras.layers.MaxPooling2D(pool_size=(2, 2), padding="valid"))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(units=8, activation='softmax'))
model.build()

model.compile(optimizer='sgd', loss='mse')
model.fit(x, y, batch_size=32, epochs=10)

predictions = model.predict(test_input)
```

Listing 1: Example of the high level API that Keras provides for building and training neural networks.

Another option is rewriting the entire network in C, either from scratch or using an existing library such as mlpack [15], FANN [16], or the existing TensorFlow C/C++ API. However, this is both time consuming and potentially error-prone, and may and require linking the resulting code against large libraries containing millions of lines of code and binaries up to several GB. Additionally, such libraries may be limited in the type of networks supported and be difficult to incorporate into existing Python based machine learning workflows. The release of TensorFlow 2.0 contained a new possibility called "TensorFlow Lite", a reduced library designed to run on mobile and IoT devices. However, TensorFlow Lite only supports a very limited subset of the full Keras API, and still relies on subsets of external libraries such as Eigen or Intel's Math Kernel

Library (MKL) [17] for many mathematical functions for which it is difficult to guarantee deterministic behavior. Therefore, we present a new option, Keras2c, a simple library for converting Keras/TensorFlow neural network models into real-time compatible C code [1], and demonstrate its use on the plasma control system (PCS)[18][19] on the DIII-D National Fusion Facility at General Atomics in San Diego [20].

2. Method

Keras2c is based around the "layer" API of Keras, which treats each layer of a neural network as a function. This makes calculating the forward pass through the network a simple matter of calling the functions in the correct order with the correct inputs. The process of converting a model using Keras2c is shown in Figure 1. The primary functionality can be broken into four primary components: weight and parameter extraction, graph parsing, a small C backend, and automatic testing.

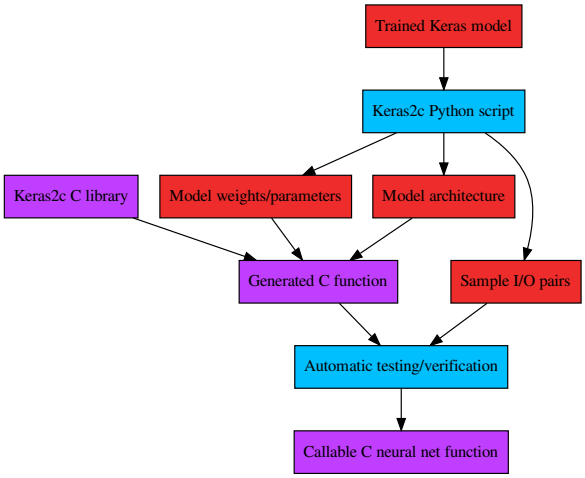

Figure 1: Workflow of converting Keras model to C code with Keras2C

2.1. Weight & Parameter Extraction

The Keras2c Python script takes in a trained Keras model and first iterates through the layers to extract the weights and other parameters. It contains

[1] All Keras2c code, documentation, and examples are available at github.com/f0uriest/keras2c

specialized methods for each type of Keras layer that parse the layer and read in the weights and relevant parameters necessary to perform the forward pass through the network such as activation type, convolution stride and dilation, etc. The parameters are then written to the generated C source file. By default, the weights are written to the file as well to be allocated on the stack using a custom Tensor datatype described in more detail in subsection 2.3.

For larger models, using the stack may be impractical. Therefore, an option exists to write the weights to external files (currently the default is to use comma separated ASCII files, though other formats such as HDF5 or NetCDF could easily be accommodated with minimal changes), which can then be read in at run time and stored on the heap. In such a case, initialization and cleanup functions are automatically generated to allocate the required memory, read in the files, and deallocate memory at the end of computation. Similarly, in some embedded applications it may be preferable to statically allocate all memory at compile time to limit the amount of stack usage. The current version of Keras2c does not support this due to potential issues when multithreading, though it is a feature planned for future versions.

2.2. Graph Parsing

In addition to sequential models, Keras also supports more complex model architectures through its functional API. This allows for models to have multiple inputs and outputs, internal branching and merging, as well as reusing specific layers multiple times in the same model. When using these features, the topology of the neural network will not be a linear stack of layers. Instead, it will be a directed acyclic graph (DAG) with each node as a layer and each edge as a piece of data being passed from one layer to another. Keras2c supports all of these more advanced network types, and it uses a version of Kahn's topological sorting algorithm [21] to flatten the computational graph into a linear sequence. Calling the layers in the corresponding order ensures that the inputs to each layer will have been generated by previous layers before they are called.

2.3. C Backend

The Keras2c backend implements the core functionality required to calculate the forward pass through each layer of the network. Each layer type supported by Keras is implemented as a function. An example of a fully connected (dense) layer is shown in Listing 2

The fundamental data type `k2c_tensor` (Listing 3) treats any multidimensional tensor as a 1D array (unraveled in row-major order), while preserving knowledge of the tensor's shape for correct indexing.

```c
struct k2c_tensor
{
    float *Array;
    size_t Ndim;
    size_t Numel;
    size_t Shape[K2C_MAX_NDIM];
};
```

Listing 3: Keras2c tensor datatype. "Array" is a pointer to a one dimensional array containing the values of the tensor unwrapped in row major order. "Ndim" is the rank of the tensor (number of dimensions). "Numel" is the total number of elements in the tensor. "Shape" is an array denoting the size of the tensor in each dimension (for example, a rank 2 tensor or matrix would have shape $[N_{rows}, N_{cols}]$). "Numel" is not strictly needed, as it can be computed as the product of the elements in the shape array, but is used to avoid needless repetition of such a calculation.

The full backend contains roughly 1500 lines of code and makes use of only C standard library functions, yet it is able to reproduce nearly every type of operation currently supported by Keras, a full list of which is given in Table 1.

Core Layers	Dense, Activation, Flatten, Input, Reshape, Permute, RepeatVector
Convolution Layers	Convolution (1D/2D/3D, with arbitrary stride/dilation/padding), Cropping (1D/2D/3D), UpSampling (1D/2D/3D), ZeroPadding (1D/2D/3D)
Pooling Layers	MaxPooling (1D/2D/3D), AveragePooling (1D/2D/3D), GlobalMaxPooling (1D/2D/3D), GlobalAveragePooling (1D/2D/3D)
Recurrent Layers	SimpleRNN, GRU, LSTM (statefull or stateless)
Embedding Layers	Embedding
Merge Layers	Add, Subtract, Multiply, Average, Maximum, Minimum, Concatenate, Dot
Normalization Layers	BatchNormalization
Layer Wrappers	TimeDistributed, Bidirectional
Activations	ReLU, tanh, sigmoid, hard sigmoid, exponential, softplus, softmax, softsign, LeakyReLU, PReLU, ELU, ThresholdedReLU

Table 1: Supported layer operations in Keras2c

Unsupported layer types include separable and transposed convolutions, locally connected layers, and recurrent layers with convolutional kernels. The existing framework makes implementing new layers (including the possibility of user defined custom layers) straightforward; the main reason for not implementing these additional layers has been lack of demand from the current user base, though they are planned for inclusion in a future release.

2.4. Automated Testing

As part of the conversion process, Keras2c generates a sequence of randomized inputs to the network and calculates the output of the original Keras/Python network. These input/output pairs are then used to generate a test function that calls the C version of the network with the randomized inputs, compares the output from the Keras2c network to the original Keras/Python network, and verifies that the converted network reproduces the correct behavior to within machine precision.

```
void k2c_dense(k2c_tensor* output, const k2c_tensor* input,
const k2c_tensor* kernel, const k2c_tensor* bias,
k2c_activationType *activation, float fwork[]) {

    if (input->ndim <=2) {
        size_t outrows;
        if (input->ndim>1) {
            outrows = input->shape[0];
        } else {
            outrows = 1;
        }
        const size_t outcols = kernel->shape[1];
        const size_t innerdim = kernel->shape[0];
        const size_t outsize = outrows*outcols;
        k2c_affine_matmul(output->array,input->array,
                          kernel->array,bias->array,
                          outrows,outcols,innerdim);
        activation(output->array,outsize);
    } else {
        const size_t axesA[1] = {input->ndim-1};
        const size_t axesB[1] = {0};
        const size_t naxes = 1;
        const int normalize = 0;

        k2c_dot(output, input, kernel, axesA, axesB,
                naxes, normalize, fwork);
        k2c_bias_add(output, bias);
        activation(output->array, output->numel);
    }
}
```

Listing 2: Keras2c dense layer example

3. Usage

An example of using Keras2c from within Python to convert a trained model is shown below in Listing 4. Here `my_model` is the Keras model to be converted (or a path to a saved model on disk in HDF5 format) and `"my_converted_model"` is the name that will be used for the generated C function and source files.

```
from keras2c import k2c
k2c(my_model, "my_converted_model", num_tests=10)
```

Listing 4: Using Keras2c to convert a Keras model to C. This will create 3 files, `my_converted_model.c`, `my_converted_model.h`, and `my_converted_model_test_suite.c`

The command shown will generate three files: `my_converted_model.c` containing the main neural net function, `my_converted_model.h` containing the necessary declarations for including the neural net in existing code, and `my_converted_model_test_suite.c` containing sample inputs and outputs and code to run the converted model to ensure accuracy. Compiling and running the test suite will print the maximum error between the original Keras model and the converted Keras2c model over 10 randomly generated input/output pairs, along with the average execution time. The test suite can also serve as a template for how to declare inputs to and outputs from the model, and how to call the model function to make predictions.

4. Benchmarks

Though the current C backend is not designed explicitly for speed, Keras2c has been benchmarked against Python Keras/TensorFlow for single CPU performance, and the generated code has been shown to be significantly faster for small to medium sized models while being competitive against other methods of implementing neural networks in C such as FANN and TensorFlow Lite. Results for several generic network types are shown in Figure 2. They show that for fully connected, 1 dimensional convolutions, and recurrent (LSTM [22]) networks, Keras2c is faster than the standard implementation in Python for models up to $\sim 10^6$ parameters. For 2D convolutions, Keras2c outperforms the Tensorflow backend for models up to 3×10^4 parameters. This scaling is intended only as a rough approximation, and the true behavior will depend strongly on the number and size of each layer, as well as the size of the inputs to the model. For all of these tests, the model was made up of four layers of the specified type, and the size of the kernel in each layer was varied. The dimension of the input was kept at a fixed fraction of the kernel dimension.

We attribute the difference in performance compared to the standard TensorFlow implementation to two primary factors: the overhead inherent in running a python process, and the level of optimization in the standard or "Lite" TensorFlow backend vs the Keras2c backend. The reference TensorFlow implementation is a mix of a high level Python interface and an extensive library of low

level kernels primarily written in C++ [23]. Running the reference TensorFlow introduces many additional layers of abstraction and additional error checking from the Python interpreter before execution passes to the low level C++ kernels. This likely explains why Keras2c is able to outperform the reference implementation for small models, as it completely avoids this additional overhead. The second aspect is the high level of optimization in the reference TensorFlow backend. When running on the CPU, TensorFlow often makes use of highly optimized linear algebra libraries such as Eigen [14] and Intel MKL [17] which offer platform specific tweaks to improve performance. Additionally, many of the lower level operations in TensorFlow are written to optimize performance for medium to large models commonly encountered in large scale machine learning, while similar operations in Keras2c are currently written to optimize for small to medium sized models that have been developed for real-time applications. For example, Keras2c computes a 2D convolution using a "direct" method involving nested for loops. This can be quite fast as long as the size of the input data and the kernels are not too large. On the other hand, TensorFlow unwraps the convolution into a very large matrix-matrix product, which can then be efficiently computed using standard linear algebra routines. This unwrapping requires significant additional memory (often allocated at execution time, which can be difficult in real-time environments), and costs additional time, but for large operations this overhead is offset by better memory access patterns and use of highly optimized matrix-matrix product routines [24].

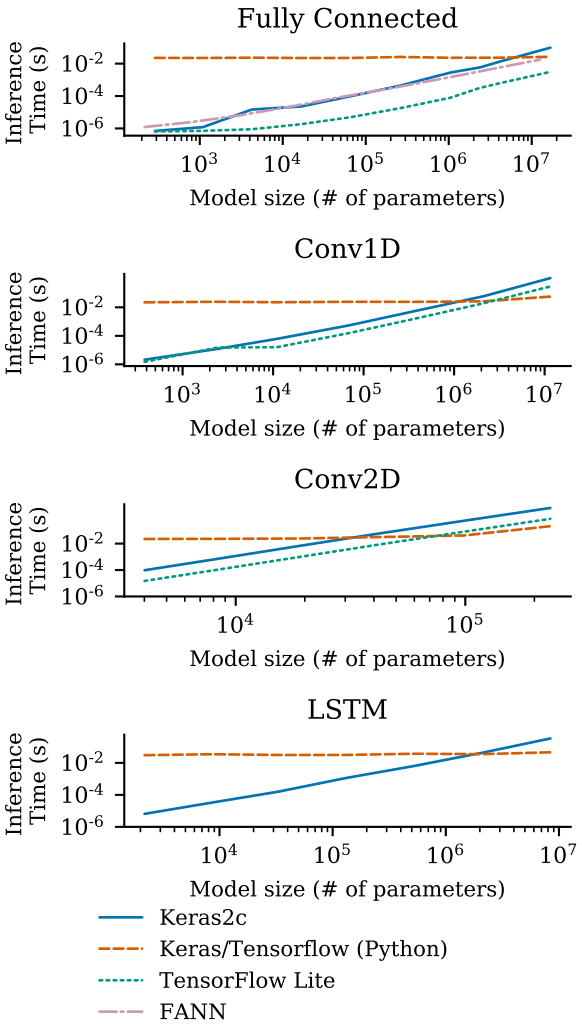

Figure 2: Benchmarking results, Keras2c vs Keras/TensorFlow in Python, TensorFlow Lite, and FANN (note that FANN only supports fully connected networks and TensorFlow Lite does not yet support recurrent networks). The vertical axis shows the inference time (ie time to compute a single forward pass through the model), while the horizontal axis shows the size of the model as measured by the total number of trainiable parameters. Each model was made up of four layers of the specified type, and the size of each layer was varied. All tests conducted on Intel Xeon E5-2630 v3 @2.40 GHz, single threaded, 32GB RAM. Keras2c compiled with ICC v19.0.4 with -O3 optimization. TensorFlow v2.3.0

5. Real-Time Applications

While not designed explicitly for speed, Keras2c was designed with real-time applications in mind, so efforts were made to ensure the generated code is deterministic and thread-safe, segmenting all non-deterministic behavior such as system calls to dedicated initialization and cleanup routines. The core of the neural net function is fully deterministic. All functions are re-entrant, and all mutable data is explicitly passed into and out of each function, allowing multiple calls to the neural net to be safely executed in parallel on multiple threads. The one exception is for "stateful" recurrent layers which require maintaining an internal state between function calls, though modification to make this thread-safe as well would be minimal and depend on the particular multithreading method used.

Keras2c has been used to deploy neural networks on the plasma control system (PCS) at the DIII-D tokamak at the National Fusion Facility operated by General Atomics in San Diego [20], where neural networks are used to predict the evolution of the plasma state [25] and the onset of dangerous instabilities using FRNN[26] and MLDA[27]. Future work will utilize neural networks and Keras2c to speed up calculations required for control of the plasma divertor [28] and pedestal [29]. The PCS consists of a software framework running on a collection of dedicated GNU/Linux real-time computers connected via an Infiniband QDR interface[30]. It operates on microsecond timescales to acquire data from sensors and diagnostics, calculate monitoring and feedback algorithms on those inputs, and output control commands to actuators on the tokamak device. There are approximately 50 different algorithms that run on varying periodics, and 3 of these now apply machine learning through the Keras2c framework to both analyze and control the plasma state.

Figure 3 shows the performance of the Keras2c generated code on a recent experiment on DIII-D. The network was designed to predict the plasma state, consisting of a total of 45,485 parameters in 30 convolutional layers, 2 LSTM layers, and a wide variety of merging and splitting layers. The timing shown is for 1 forward pass through the network and includes the time required to acquire the needed input data from other processes and convert the input data into the correct format and shape [2]. The mean time is 1651.6 μs, and the worst case jitter (full range, min to max) is 23.3 μs (1.4% of the mean), with an RMS value of $\pm$3.75 μs. Running the inference in the Python version of Keras/TensorFlow (not shown) is significantly slower, with a mean time of 48 ms with worst case jitter of 12.5 ms (26% of the mean time).

[2] The additional time required for such data pre-processing is negligible compared to the computations involved in the network itself, but efforts to isolate it introduced additional latency, resulting in worse performance overall. The time shown including pre-processing can be thought of as an upper bound on the time for the inference alone.

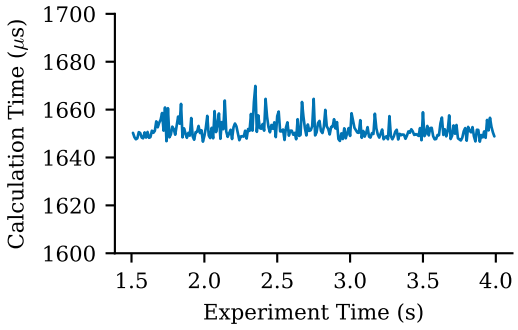

Figure 3: Example timing from a neural network running on the DIII-D control system. The time shown is for 1 forward pass through the network, and also includes the time needed to gather and prepare input data. The network contains 30 convolutional layers, 2 LSTM layers, and a number of merging and splitting layers, with a total of 45,485 parameters. Test performed on Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, compiled with ICC 2018.

6. Discussion

As discussed in section 4, the backend code as currently implemented is not optimized for execution speed, especially for large models. Convolutions are currently implemented using a direct calculation as opposed to the "im2col + GEMM" approach more commonly used in deep learning libraries [24]. Planned future work will include implementing this and other modifications to improve calculation speed and better support the larger model sizes becoming common in state of the art research. Work is also underway to allow use of Intel's Math Kernel Library for backend operations.

7. Conclusion

Keras2c allows for the straightforward and simple conversion of Keras neural networks to pure C code, in a form that can be easily deployed to real-time control systems, or anywhere that a C executable can be run. By relying only on C standard library functions, it avoids any complicated dependencies that can make deploying any program a challenge. The generated code is designed to be human readable and editable, to allow for custom modifications and extensions if necessary. Despite not making use of advanced numerical libraries, it is still competitive with the reference TensorFlow implementation for small to medium model sizes and more importantly has been designed from the start for low-latency deterministic behavior which has been demonstrated in its use on the plasma control system of the DIII-D tokamak at the National Fusion Facility in San Diego. Planned future improvements will aim to improve on this performance for larger models while maintaining a lean structure and limited dependencies to ease integration into existing codebases and systems.

8. Acknowledgments

The authors thank Mitchell Clement, Ge Dong, and Mark D. Boyer for their help in beta testing and bug fixing. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Fusion Energy Sciences, using the DIII-D National Fusion Facility, a DOE Office of Science user facility, under Awards DE-FC02-04ER54698, DE-SC0015878, DE-AR0001166, and Field Work Proposal No. 1903

Notice: This manuscript is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Fusion Energy Sciences, and has been authored by Princeton University under Contract Number DE-AC02-09CH11466 with the U.S. Department of Energy. The publisher, by accepting the article for publication acknowledges, that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

9. References

1. Abadi, M. *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015. https://www.tensorflow.org/.
2. Chollet, F. *et al. Keras* https://keras.io. 2015.
3. Hunt, K. J., Sbarbaro, D, Żbikowski, R & Gawthrop, P. J. Neural networks for control systems—a survey. *Automatica* **28,** 1083–1112 (1992).
4. Jin, L., Li, S., Yu, J. & He, J. Robot manipulator control using neural networks: A survey. *Neurocomputing* **285,** 23–34 (2018).
5. Liu, W. *et al.* A survey of deep neural network architectures and their applications. *Neurocomputing* **234,** 11–26 (2017).
6. Abiodun, O. I. *et al.* State-of-the-art in artificial neural network applications: A survey. *Heliyon* **4,** e00938 (2018).
7. *Amazon SageMaker* https://aws.amazon.com/sagemaker/ (2020).
8. *Oracle GraphPipe* https://oracle.github.io/graphpipe/ (2020).
9. Bai, J., Lu, F., Zhang, K., *et al. ONNX: Open Neural Network Exchange* https://github.com/onnx/onnx. 2019.
10. Behnel, S. *et al.* Cython: The Best of Both Worlds. *Computing in Science Engineering* **13,** 31 –39. ISSN: 1521-9615 (2011).
11. Hayen, K. *Nuitka* https://nuitka.net/ (2020).
12. Paszke, A. *et al.* Automatic differentiation in PyTorch (2017).
13. Hermann, T. *Frugally Deep* https://github.com/Dobiasd/frugally-deep. 2020.

14. Guennebaud, G., Jacob, B., *et al. Eigen v3* http://eigen.tuxfamily.org. 2010.

15. Curtin, R. R. *et al.* mlpack 3: a fast, flexible machine learning library. *Journal of Open Source Software* **3,** 726 (2018).

16. Nissen, S. *Implementation of a Fast Artificial Neural Network Library (fann)* tech. rep. http://fann.sf.net (Department of Computer Science University of Copenhagen (DIKU), 2003).

17. Intel. *Intel Math Kernal Library (MKL)* software.intel.com/mkl. 2020.

18. Ferron, J. R., Penaflor, B., Walker, M. L., Moller, J. & Butner, D. Flexible software architecture for tokamak discharge control systems. *Proceedings - Symposium on Fusion Engineering* **2,** 870–873 (1995).

19. Hyatt, A. W. *et al.* Physics operations with the DIII-D plasma control system. *IEEE Transactions on Plasma Science* **38,** 434–440 (2010).

20. Luxon, J. L. A design retrospective of the DIII-D tokamak. *Nuclear Fusion* **42,** 614–633 (2002).

21. Kahn, A. B. Topological sorting of large networks. *Communications of the ACM* **5,** 558–562 (1962).

22. Gers, F. A., Schmidhuber, J. & Cummins, F. Learning to forget: Continual prediction with LSTM (1999).

23. TensorFlow. *TensorFlow Source* https://github.com/tensorflow/tensorflow. 2020.

24. Chetlur, S. *et al.* cuDNN: Efficient Primitives for Deep Learning. *arXiv,* 1–9. arXiv: 1410.0759. http://arxiv.org/abs/1410.0759 (2014).

25. Abbate J., C. R. & Kolemen, E. Fully Data-Driven Profile Prediction for DIII-D. *Nuclear Fusion* (In Review).

26. Kates-Harbeck, J., Svyatkovskiy, A. & Tang, W. Predicting disruptive instabilities in controlled fusion plasmas through deep learning. *Nature* **568,** 526–531 (2019).

27. Fu, Y. *et al.* Machine learning control for disruption and tearing mode avoidance. *Physics of Plasmas* **27** (2020).

28. Kolemen, E. *et al.* Initial development of the DIII-D snowflake divertor control. *Nuclear Fusion* **58,** 066007. ISSN: 17414326 (2018).

29. Laggner, F. *et al.* Real-time pedestal optimization and ELM control with 3D fields and gas flows on DIII-D. *Nuclear Fusion* **60,** 076004. ISSN: 0029-5515 (2020).

30. Penaflor, B. *et al.* Extending the capabilities of the DIII-D Plasma Control System for worldwide fusion research collaborations. *Fusion engineering and design* **84,** 1484–1487 (2009).