A Flexible Method to Integrate AI Models (PyTorch) into LS-DYNA User Defined Material Subroutines

Cristian Saenz-Betancourt[1,2], Dustin Draper[1], Balazs Fodor[1], Fabian Duddeck[3], Steffen Peldschus[2]

[1]BMW Group, Knorrstr. 147, 80788, Munich, Germany
[2]Ludwig-Maximilians-Universität München, Nussbaumstr. 26, 80336, Munich, Germany
[3]Technical University of Munich, Arcisstr. 21 80333, Munich, Germany

1 Introduction

Using artificial intelligence in mechanics is a rapidly developing research area [1]. New architectures and functionalities are continually released. Efficiently testing the performance of these models within engineering simulations is crucial for determining which designs and features are scalable for industrialization.

Efforts to develop Artificial Neural Networks (ANNs) as material models have been made [2], [3], [4], [5], [6]. Since the end goal of a material model development is to be used in simulations, an interface that enables ANN integration with finite element (FE) software is required. For instance, LS-DYNA is written in Fortran, while AI models are typically developed in Python with libraries like PyTorch.

An ANN is a mathematical function defined by numerous parameters (weights and biases), matrix multiplications, and the evaluation of nonlinear activation functions. While it is possible to implement these operations in Fortran to construct an ANN, doing so is time-consuming. It becomes increasingly impractical with new, more complex architectures and AI capabilities. Moreover, such an approach is relatively inflexible if one wishes to modify the ANN architecture.

To enable an ANN to act as a material model in LS-DYNA, we propose using FTorch [7], a recently developed library that is Free and Open Source (MIT license). FTorch development was motivated by using PyTorch models inside computational climate models that are often written in Fortran. FTorch handles the row-major column-major disparity between the languages, and the use of shared memory is enabled to maximize efficiency during the coupling.

The goal of this paper is to describe the connection between the PyTorch model and LS-DYNA through FTorch and user subroutines, rather than to explain how to design, train, or validate the ANN as a material model. The relevance of this work lies in enabling AI model designers to test online predictions of their models within a commercial FEM software. Evaluating the AI models in LS-DYNA is essential to understand their interaction with the FEM solver, including issues related to stability, ranges of validity, generalization, and time performance.

2 State of the Art

Artificial Neural Networks (ANNs) are universal function approximators. A material model is a function that can be approximated with an ANN by learning the strain–stress relationship from the data [3]. Another application of ANNs in material modeling is to solve small-scale problems within multi-scale frameworks, e.g., in composites [8], [9]. ANN-based material models are attractive for nonlinear and inelastic behavior; for example, classical plasticity models require iterative return-mapping procedures. In contrast, the ANN computes the material state in a single step.

In the context of ANN-based material models, it can be distinguished between ANNs that are fully automatic and discover implicitly material history variables [2] and models that include material history variables as inputs [10]. Both types could be implemented as UMAT for LS-DYNA; however, the ANN architectures differ. If the history variables are known a priori, feed-forward architectures can be used; if not, recurrent architectures are necessary.

The challenge of connecting LS-DYNA with ANNs has been addressed previously [11]. An interprocess communication workflow was introduced in which a dummy subroutine manages communication with

Python, allowing any Python function to serve as a material model, including those developed in PyTorch. This approach requires Python to be running concurrently with LS-DYNA. It offers flexibility in the sense that it allows access to Python in a general manner; however, it is poor in performance compared to a method based on linking libraries.

To the authors' knowledge, the FTorch-LS-DYNA connection has not been previously done and documented. Therefore, we proposed the following workflow.

3 Workflow

First, the FTorch library is installed. Second, the LS-DYNA is linked to FTorch and PyTorch by modifying the Makefile. Third, the architecture of a trained ANN-based material model is presented. Then, a suitable vectorized material subroutine is proposed. Finally, an exemplary simulation is carried out. Fig 2 summarizes the integration workflow. Along this section, snippets with relevant code are included.

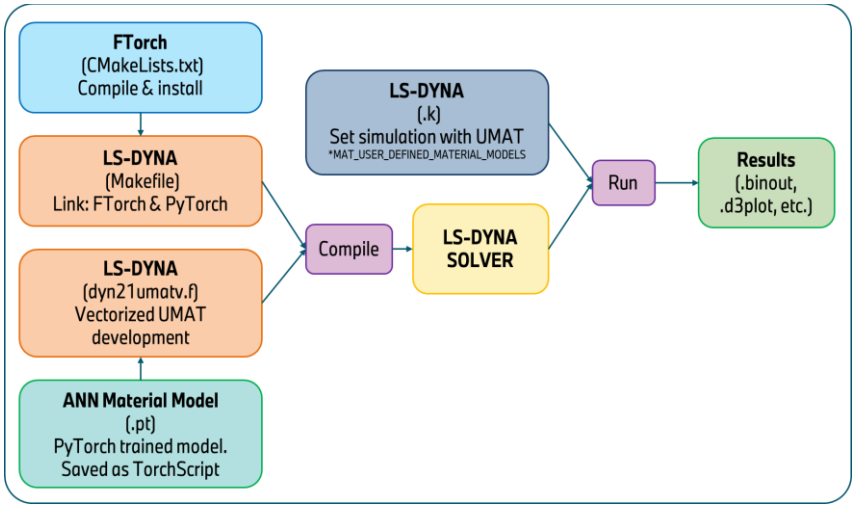

Fig.1: Overview of the method.

3.1 Install FTorch (CMake)

CMake is used to build and install the FTorch library on Linux. A CMake file and installation details are available in the repository documentation (https://cambridge-iccs.github.io/FTorch). The following information can be updated in the CMake file: Fortran compiler, location of PyTorch library, and GPU device, if needed.

3.2 Linking with LS-DYNA (Make)

An LS-DYNA version with access to the user subroutines is used. The Makefile must be modified to include the FTorch and PyTorch libraries. The compilation is initially performed with all default user subroutines to ensure a correct connection.

To include directories of the headers for compilation:

```
-I/home/local/include/ftorch
-I/path/myvenv/lib/python3.10/site-packages/torch/include/torch/csrc/api/include/torch
```

To include the directories of the libraries for linking. The PyTorch CPU-only version is used for this work.

```
-L/home/local/lib64 -lftorch
-L/path/myvenv/lib/python3.10/site-packages/torch/lib -ltorch_cpu
```

After successfully connecting LS-DYNA with FTorch, the UMAT can be written. The UMAT code is dependent on the ANN architecture, specifically on the task of determining the relationship between the ANN variables and the LS-DYNA variables.

3.3 ANN-based Material Model

We show a model for elastoplasticity. It was trained on synthetic data generated from MAT24 for the plane stress case. An elastoplastic material model can be seen as a function that maps the current

known material state together with history variables ($\boldsymbol{\varepsilon}^t, \boldsymbol{\sigma}^t, \boldsymbol{h}^t$), and strain increment ($\Delta\boldsymbol{\varepsilon}^t$) to the updated material state and updated history variables ($\boldsymbol{\varepsilon}^{t+1}, \boldsymbol{\sigma}^{t+1}, \boldsymbol{h}^{t+1}$).

$$\boldsymbol{\sigma}^{t+1},\ \boldsymbol{h}^{t+1} = f(\boldsymbol{\varepsilon}^t, \Delta\boldsymbol{\varepsilon}^t, \boldsymbol{h}^t),$$

$$\boldsymbol{\varepsilon}^{t+1} = \boldsymbol{\varepsilon}^t + \Delta\boldsymbol{\varepsilon}^t.$$

An ANN architecture that fulfills this functionality is the Gated Recurrent Unit (GRU) (Fig. 2). One key advantage is that the history variables of the material are implicitly captured within the hidden state of the GRU, eliminating the need to define them a priori as in classical material models. The chosen hidden state size is thirty. Viewed over an entire simulation, the strain, strain increment, and stress tensor form time series, making recurrent architectures particularly well-suited for this task.

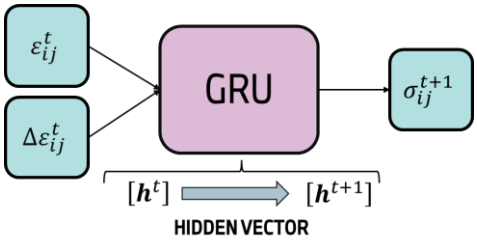

Fig.2: Gated Recurrent Unit (GRU).

The input, hidden, and output arrays follow the shapes specified by PyTorch GRU with batch_first, a single layer, and a unidirectional configuration. For the selected model, the total strain of the known state and the strain increment are concatenated to form the input vector. The features correspond to the components of the strain and stress.

Input/Output shape: [batch size, sequence length, number of features]
Hidden vector shape: [1, batch size, hidden size]

The GRU can be configured to operate over time steps in multiple ways. It was trained in a many-to-many fashion, predicting complete output time sequences from complete input time sequences. During training, sequences of varying lengths were used. For online prediction within the material subroutines, the GRU is configured for a one-to-one mapping since LS-DYNA advances one time step at a time. Consequently, the hidden state must be preserved across time steps, and the sequence length is effectively one during an ANN online prediction.

A many-to-one approach could be implemented in which strains and stresses across several previous time steps are stored and used to predict the next step. However, this is computationally expensive because all prior information must be stored at each integration point.

To convert the PyTorch model to TorchScript, we employed the pt2ts.py utility from the FTorch repository. TorchScript is a serialized intermediate representation (IR) of the PyTorch model. It can be executed without a Python interpreter. This transformation can be achieved by tracing or scripting; scripting is preferred for completeness, since tracing can yield less reliable results when the model includes conditional branches.

```
# Torch_model: torch.nn.Module
TorchScript_model = torch.jit.script(Torch_model)
TorchScript_model.save('TorchScriptModel.pt')
```

3.4 User Defined Material Subroutine (UMAT)

A vectorized subroutine is implemented instead of the scalar version to achieve better simulation run time. In addition, one can leverage the tensorized structure of ANNs, which are vectorized by default along the batch dimension. By treating integration points as the batch dimension, several integration points can be computed simultaneously, rather than looping over them individually.

The file *dyn21umatv.f* contains ten vectorized material subroutines. The ANN model must be loaded outside of any of the ten subroutines; it is initialized once and invoked within the target subroutine (here,

umat43v). Loading the ANN inside the material subroutine would require reloading it for every time step and for every batch of integration points, which would severely slow the simulation. To address this, a new module was defined within *dyn21umatv.f.* This module contains a new subroutine that loads the TorchScript model onto the device. The subroutine and the model variable in this module are declared public, so they are accessible from any of the ten material subroutines at any time. Details can be found in the following code snippet.

```
c=======================================================================
c     Module for PyTorch model integration in LS-DYNA
c=======================================================================
module modelLoadModule

      ! PyTorch-Fortran interface library
      use ftorch
      implicit none

      ! Make these accessible from outside the module
      public model_init, torchModel

      ! Global PyTorch model variable
      type(torch_model) :: torchModel

      contains

      ! Subroutine to load the PyTorch model
      subroutine model_init()
            ! Load the TorchScript model file
            call torch_model_load(torchModel, '/path/TorchScriptModel.pt')
      end subroutine model_init

end module modelLoadModule
```

In the material subroutine (e.g., *umat43v*), LS-DYNA provides the strain increment (dX) and history variables (hsv). The task consists of updating the history variables and the stress state (sigX) and returning them to the explicit FEM scheme (the outer loop). The index X ranges from 1 to 6. The first three entries correspond to the normal strain increments/stresses, while the last three correspond to the shear components. These are one-dimensional arrays with length nlq which is an LS-DYNA variable for the number of integration points being simultaneously computed.

```
subroutine umat43v(cm,d1,d2,d3,d4,d5,d6,sig1,sig2,sig3,sig4,sig5,sig6,epsps,hsv,lft,llt,dt1siz,capa,
. etype,tt,temps,failels,nlqa,crv,nnpcrv,cma,qmat,elsizv,idelev,reject)
```

For the plane-stress case, the components d5, d6, sig3, sig5, and sig6 are zero-valued. The history variable array hsv is two-dimensional with shape [nlq, nhv], i.e., the number of integration points in a batch by the number of history variables. If the model requires the total strain component, the components (three for the plane stress case) can be stored as history variables (hsv).

At the top of the material subroutine, the FTorch and the module for loading the model are imported.

```
use ftorch
use modelLoadModule
```

Subsequently, the arrays are declared.

```
! Declare Fortran arrays
real(wp), dimension(nlq,1,6) , target :: in_data
real(wp), dimension(1,nlq,30), target :: h_data
real(wp), dimension(nlq,1,3) , target :: out_data
```

The Torch tensors are declared. Their dimension refers to the number of arrays that they containe, similar to an array of arrays. For instance, the GRU takes the strain and strain increment (*in_data),* and the hidden vector (*h_data*) as input and outputs the stress (*out_data*) and the updated hidden vector (*h_data*).

```
! Declare Torch tensors
type(torch_tensor), dimension(2) :: in_tensors
type(torch_tensor), dimension(2) :: out_tensors
```

Once all variables are declared, the subroutine *model_init()* of the module *modelLoadModule* is called at cycle 0 of the simulation. The public model variable *torchModel* is created and kept available for all other times that the material subroutine is called.

```
if (ncycle.eq.0) then
  call model_init()
endif
```

Usually, the features in ANNs are normalized, e.g., in the range [0,1] or [-1,1] for better training. The normalization function and the inverse of the normalization are straightforward Fortran operations. The LS-DYNA variables are assigned to arrays. The normalization is applied to the input arrays from LS-DYNA before evaluating in the ANN. The inverse normalization is applied to the output arrays before returning the values to the outer loop.

```
! Total strain --> in_data
do k = 1, 3
    in_data(1:nlq, k)=hsv(1:nlq, k) ! ε^t
end do

! Strain increment --> in_data
in_data(1:nlq,4)=d1(1:nlq)          ! Δε^t_xx
in_data(1:nlq,5)=d2(1:nlq)          ! Δε^t_yy
in_data(1:nlq,6)=0.5*d4(1:nlq)      ! Δε^t_xy

! History variables --> h_data
h_data(1:nlq, 1:30) = hsv(1:nlq, 4:33) ! h^t
```

The Torch tensors and arrays are linked by using the function *torch_tensor_from_array_real32_3d.* Other dimensions and types are available in the library. Although the FTorch library is well documented, it is also worth reading the details in its *ftorch.f90* file.

```
! Allocate data in the Torch tensors
call torch_tensor_from_array_real32_3d(in_tensors(1) ,in_data  ,layout ,torch_kCPU) ! [ε^t,Δε^t]
call torch_tensor_from_array_real32_3d(in_tensors(2) ,h_data   ,layout ,torch_kCPU) ! h^t
call torch_tensor_from_array_real32_3d(out_tensors(1),out_data ,layout ,torch_kCPU) ! σ^t+1
call torch_tensor_from_array_real32_3d(out_tensors(2),h_data   ,layout ,torch_kCPU) ! h^t+1
```

At each time step, the model is evaluated by calling the function *torch_model_forward,* which takes the input arrays of *in_tensors* and stores the predictions in the arrays of *out_tensors.*

```
! Evaluate the model:
call torch_model_forward(torchModel, in_tensors, out_tensors)
```

Afterwards, the results are allocated to the LS-DYNA variables. Physical values must be given to sigX, i.e., the inverse normalization is applied to out_data prior to return.

```
! Update stress
sig1(1:nlq) = out_data(1:nlq,1,1) ! σ^t+1_xx
sig2(1:nlq) = out_data(1:nlq,1,2) ! σ^t+1_yy
sig4(1:nlq) = out_data(1:nlq,1,3) ! σ^t+1_xy

! Save total strain
hsv(1:nlq, 1) = hsv(1:nlq, 1) + d1(1:nlq)     ! ε^t+1_xx = ε^t_xx + Δε^t_xx
hsv(1:nlq, 2) = hsv(1:nlq, 2) + d2(1:nlq)     ! ε^t+1_yy = ε^t_yy + Δε^t_yy
hsv(1:nlq, 3) = hsv(1:nlq, 3) + 0.5*d4(1:nlq) ! ε^t+1_xy = ε^t_xy + Δε^t_xy

! Update GRU hidden vector
hsv(1:nlq, 4:33) = h_data(1:nlq,1,:) ! h^t+1
```

All tensors are deleted at the end of the subroutine. For example:

```
! Clean up torch tensors
call torch_tensor_delete(in_tensors(1))
```

Table 1 presents a summary of the variables. The strain increment and total strain are in the same array because they are concatenated as input (specific for this ANN).

		Shape	Array	Torch tensor	LS-DYNA
Strain increment	$\Delta\boldsymbol{\varepsilon}^t$	[nlq, 1, 3]	in_data	in_tensors(1)	d1, d2, d4
Total strain	$\boldsymbol{\varepsilon}^t$	[nlq, 1, 3]			hsv(1:nlq, 1:3)
History variables	$\boldsymbol{h}^t$	[1, nlq, 30]	h_data	in_tensors(2)	hsv(1:nlq, 4:33)
Updated stress	$\boldsymbol{\sigma}^{t+1}$	[nlq, 1, 3]	out_data	out_tensors(1)	sig1, sig2, sig4
Updated history variables	$\boldsymbol{h}^{t+1}$	[1, nlq, 30]	h_data	out_tensors(2)	hsv(1:nlq, 4:33)

Table 1: Summary of variables.

Once the vectorized UMAT is written, LS-DYNA is compiled and ready to use for simulation.

3.5 Simulation.

The model consists of a square that is discretized into nine shell elements. It is driven by displacement with a triangular signal in the x-direction (Fig.3 left). The chosen consistent unit system is ton-mm-s-N. The user material is set in the keyword ***MAT_USER_DEFINED_MATERIAL_MODELS**. MT defines the material subroutine number (41-50); here, 43. IVEC is the vectorized flag, 0 for scalar (*dyn21umats.f)* or 1 for vectorized (*dyn21umatv.f)* subroutines.

LS-DYNA requires the mass density (RO), the bulk modulus, and the shear modulus. The last two are located at the material constant array with length (LMC). IBULK and IG are the corresponding indices in the material constant array. The values are given at PARAM3 and PARAM4, since the IBULK and IG indices are 3 and 4, respectively. The number of history variables is set at NHV. In this example, 33 is chosen. It includes three components of the current known total strain ($\varepsilon_{xx}^t, \varepsilon_{yy}^t, \varepsilon_{xy}^t$) and 30 components of the GRU hidden vector ($\boldsymbol{h}^t$).

```
*KEYWORD
*MAT_USER_DEFINED_MATERIAL_MODELS
$      MID|       RO|       MT|      LMC|      NHV|   IORTHO|    IBULK|       IG|
         1    2.E-9         43        4         33        0         3         4
$    IVECT|    IFAIL|   ITHERM|   IHYPER|     IEOS|     LMCA|
         1         0         0         0         0         0
$   PARAM1|   PARAM2|   PARAM3|   PARAM4|
        0.        0.    12250.   5653.85
```

The results are stored for all integration points in the binary file. Fig.3 (right) shows the typical elastoplastic behavior. It starts with tensile loading, deforms elastically, then plastically with hardening. Afterwards, unloading takes place (elastically). Finally, compressive loading happens with elastic and plastic deformation.

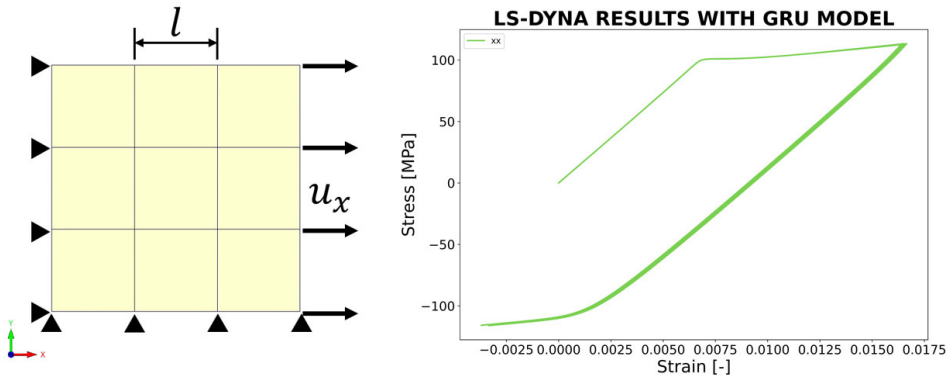

Fig.3: Left: Model. Displacement driven (triangular signal u_x). Right: Simulation results. Elastoplastic behavior in tension and compression. Elastic behavior during unloading.

Fig.4 displays the strains and stresses as time series. In Fig.4 (left), ε_{xx} follows the triangular signal of the displacement boundary condition. ε_{yy} shows the Poisson ratio effect and the change of slope when plastic deformation occurs. ε_{xy} remains closed zero, as expected. In Fig.4 (right), σ_{xx} shows the tensile and compressive stresses. σ_{yy}, σ_{xy} remain close to zero, as expected in a uniaxial loading scenario.

A relevant consideration for online ANN prediction is that the sequence length is determined by the total simulation time and the time step, in accordance with the CFL condition. This illustrates a scenario in which LS-DYNA governs how the ANN is used, and the model's ability to adapt to different time steps is hence tested. In Fig.4, the strain time series diverge toward the end of the simulation; this is due to the stability of the ANN over long sequences. Moreover, after the loading-unloading transition (at half simulation time), the stresses $(\sigma_{yy}, \sigma_{xy})$ do not precisely meet the zero-stress condition for the uniaxial loading case. This has implications while solving the equilibrium for the next steps.

The testing of an ANN as a material model in Python without an explicit FEM scheme is equivalent to testing it in a simulation in which all degrees of freedom have a Dirichlet boundary condition. This means that the strain increment is known a priori at all time steps. That situation is completely stable in LS-DYNA. However, as it is usual in FEM simulations, many elements contain free nodes, in which the strain increments are computed from the previous states, i.e., the previous ANN predictions. This can generate stability issues and error propagation.

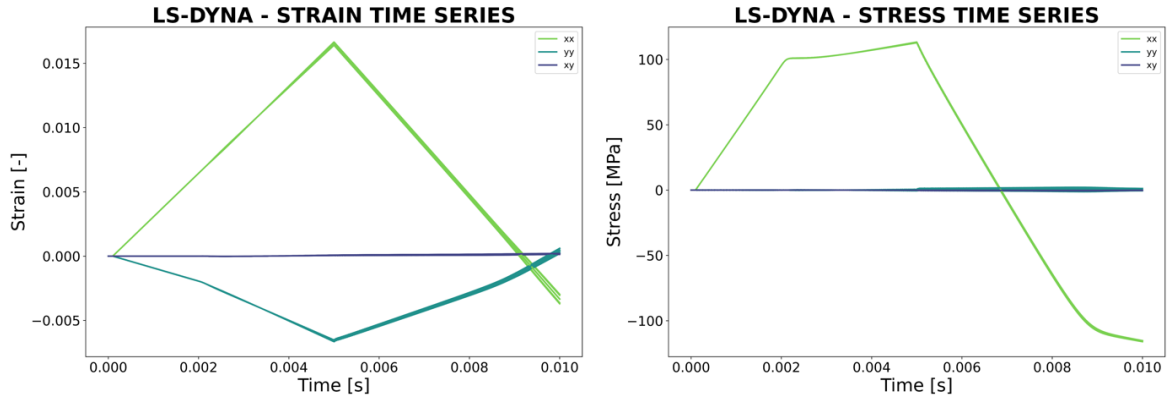

Fig.4: Simulation results. Time series at all integration points. Read from binout file.

Some strategies to improve the stability of the simulation are implemented. A constant time step is selected during the total simulation time (**`*CONTROL_TIMESTEP`**). It allows the total sequence length for the GRU to be controlled.

Considering that the ANNs are approximators of a function, there is not always a smooth response. Oscillations can be present, especially in transition zones, e.g., elasticity-plasticity or loading-unloading. The material model oscillations can interact with the oscillations in the explicit FEM scheme. Therefore, bulk viscosity (**`*CONTROL_BULK_VISCOSITY`**) is included to damp them. This requires further research.

4 Summary and Outlook

FTorch is an open-source library that enables the use of AI material models within Fortran-based codes, such as LS-DYNA, via User Defined Material Subroutines (UMAT). To maximize simulation time performance, a vectorized subroutine is recommended to leverage the parallel data layout inherent to ANN computations.

An example with a GRU-based material model coupled to LS-DYNA was shown. The trained model was saved as TorchScript, a format that FTorch can read. The simulation included elasticity, plasticity with hardening, and unloading. The presence of free nodes in the simulation poses challenges for offline-trained ANN models because of effects from time discretization and error propagation. By leveraging the FTorch connection, AI models can be evaluated (online) under realistic simulation scenarios, allowing for the understanding of the interaction of AI and FEM for better AI model design and training.

FTorch also offers the functionality of keeping the gradients, which can be used for automatic differentiation. It could be attractive to the material models based on potentials (e.g., Helmholtz free energy in hyperelasticity) which compute the stresses by derivation. FTorch is currently being expanded; for example, the PyTorch optimizer can be called from Fortran. This implies that the ANN could be

updated while running an LS-DYNA simulation. It is helpful because of the learning from the real-time discretization.

This work was implemented for explicit simulations. Additionally, FTorch can also be applied to implicit simulations, where the ANN would be tasked with predicting the components of the consistent tangent stiffness matrix. Furthermore, FTorch can be used in other interfaces with potential to be enhanced by AI, such as equations of state, cohesive models, friction models, etc.

The flexibility of **`*MAT_USER_DEFINED_MATERIAL_MODELS`** could be increased. It requires inputs such as the bulk and shear moduli (for the time step, contact, etc.). However, an ANN-based material model contains nonphysical parameters (weights and biases). If a purely ANN material model is employed, the bulk and shear moduli would be unknown, necessitating a workaround to define or estimate these quantities within the material subroutine.

5 References

[1] L. Herrmann and S. Kollmannsberger, “Deep learning in computational mechanics: a review,” *Comput. Mech.*, vol. 74, no. 2, pp. 281–331, Aug. 2024, doi: 10.1007/s00466-023-02434-4.

[2] R. Lourenço, A. Tariq, P. Georgieva, A. Andrade-Campos, and B. Deliktaş, “On the use of physics-based constraints and validation KPI for data-driven elastoplastic constitutive modelling,” *Comput. Methods Appl. Mech. Eng.*, vol. 437, p. 117743, Mar. 2025, doi: 10.1016/j.cma.2025.117743.

[3] C. Bonatti and D. Mohr, “One for all: Universal material model based on minimal state-space neural networks,” *Sci. Adv.*, vol. 7, no. 26, p. eabf3658, Jun. 2021, doi: 10.1126/sciadv.abf3658.

[4] M. Rosenkranz, K. A. Kalina, J. Brummund, and M. Kästner, “A comparative study on different neural network architectures to model inelasticity,” *Int. J. Numer. Methods Eng.*, vol. 124, no. 21, pp. 4802–4840, Nov. 2023, doi: 10.1002/nme.7319.

[5] P. Böhringer *et al.*, “A strategy to train machine learning material models for finite element simulations on data acquirable from physical experiments,” *Comput. Methods Appl. Mech. Eng.*, vol. 406, p. 115894, Mar. 2023, doi: 10.1016/j.cma.2023.115894.

[6] J. N. Fuhg *et al.*, “A review on data-driven constitutive laws for solids.” arXiv, May 06, 2024. Accessed: May 11, 2024. [Online]. Available: http://arxiv.org/abs/2405.03658

[7] J. Atkinson *et al.*, “FTorch: a library for coupling PyTorch models to Fortran,” *J. Open Source Softw.*, vol. 10, no. 107, p. 7602, Mar. 2025, doi: 10.21105/joss.07602.

[8] Y. Zhou and S. J. Semnani, “A machine learning based multi-scale finite element framework for nonlinear composite materials,” *Eng. Comput.*, Apr. 2025, doi: 10.1007/s00366-025-02121-3.

[9] L. Wu, V. D. Nguyen, N. G. Kilingar, and L. Noels, “A recurrent neural network-accelerated multi-scale model for elasto-plastic heterogeneous materials subjected to random cyclic and non-proportional loading paths,” *Comput. Methods Appl. Mech. Eng.*, vol. 369, p. 113234, Sep. 2020, doi: 10.1016/j.cma.2020.113234.

[10] M. B. Gorji and D. Mohr, “Towards neural network models for describing the large deformation behavior of sheet metal,” *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 651, no. 1, p. 012102, Nov. 2019, doi: 10.1088/1757-899X/651/1/012102.

[11] J. Sprave, T. Erhart, and A. Haufe, “An Interprocess Communication based Integration of AI User Materials into LS-DYNA,” 2023.