



# Implementation of neural network based constitutive models

DE LA RECHERCHE À L'INDUSTRIE

21 October 2021

M. Duvillard, L. Giraldi, T. Helfer

## Context

## Modelling

- ▶ Constitutive models describe the behaviour of materials in mechanical problems.
- ▶ Include **nonlinearity**, **time-dependency** ( $\sigma(t) = f(\{\varepsilon(s)_{s \leq t}\})$ ) or **heterogeneity**.
- ▶ ex : elasticity, elasto-plasticity, visco-elasticity, finite strain theory, damage, RVEs...
- ▶ **Complex** to model.

## Simulation

- ▶ Could be **computational** expensive.
- ▶ Need to ensure that the global problem is **well-posed**, **solvable** and **consistent**.

## Supervised learning : regression

- ▶ Task : Learning a function from input-output sample.
- ▶ How : Least squares minimization.
- ▶ Holdout validation :
  - training set : fit the model;
  - validation set : tune/select the model;
  - test set : assess the quality of the final model.

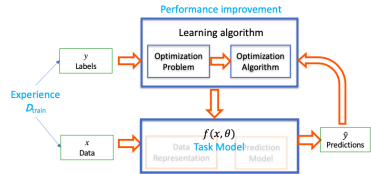


FIGURE 1 – Supervised learning concept.

## Neural networks

- ▶ Universal approximation theorems
- ▶ Deep neural network refers to **multiple layers architecture** allowing to approximate various highly nonlinear functions.
- ▶ Differentiable and trainable.

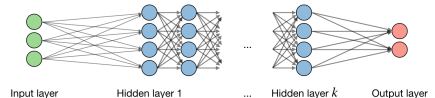


FIGURE 2 – Dense deep neural network architecture.

## Modelling

- ▶ Construct deep neural networks as constitutive models to :
  - identify new behaviours.
  - create surrogate models.
- ▶ We use the PyTorch library.

## Simulation

- ▶ Ensure numerical solver stability criterions by **semi-supervised learning**.
- ▶ Integrating the model in MFront for material point simulation with MTest and finite elements simulations with `mgis.fenics`.

## Steps

1. Determine the type of training examples and input features;
2. Gather a training set **representative** of the real-world use of the function;
3. Run the learning algorithm;
4. Evaluate the prediction of the learned function.

# Application to nonlinear elastic behaviours

- ▶ Depend only on the mechanical state at the time  $t$  :  $\sigma(t) = f(\epsilon(t))$ .
- ▶ Data represent a point in the stress-strain state space : the strain as input and stress as output.
- ▶ Input and output features :

$$\epsilon = \{\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}, \sqrt{2}\epsilon_{xy}, \sqrt{2}\epsilon_{xz}, \sqrt{2}\epsilon_{yz}\}$$

$$\sigma = \{\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sqrt{2}\sigma_{xy}, \sqrt{2}\sigma_{xz}, \sqrt{2}\sigma_{yz}\}$$

- ▶ Representativity : sampling of a subspace of the 6 dimensions strain space.
- ▶ Data could come from different sources : physical tests, micro-mechanical simulations, constitutive equations.
  - Physical tests : still restricted to generate various mechanical state
  - Micro-mechanical simulations and constitutive equations could be generate more easily.
  - Constitutive equations usually use mathematical properties.

- ▶ Applying the method with **synthetic** data generated with suitable available behaviours.
- ▶ A nonlinear elastic behaviour : **Ramberg-Osgood**.

$$\epsilon = \frac{1}{3K} Tr(\sigma) \mathbf{I} + \frac{\sigma_{eq}}{3\mu} \mathbf{n} + \alpha \left( \frac{\sigma_{eq}}{\sigma_0} \right)^n \mathbf{n},$$

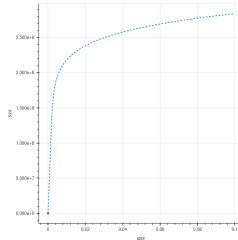


FIGURE 3 – Uniaxial traction load with a Ramberg-Osgood behaviour.



- ▶ Using **random sampling** with a **multivariate normal distribution** in the deviatoric and hydrostatic space.
- ▶ Performing loading paths thanks to the MTest Python API.
- ▶ Simulation of  $10^4$  sequences  $\times$  100 time steps =  $10^6$  points.
- ▶  $\{(\varepsilon, \sigma)_i\}_{i=1}^{10^6}$  divided into training, validation, and test dataset.

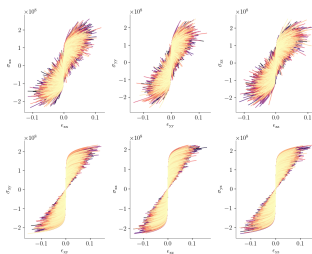


FIGURE 4 – Data representation with linear load sequences.

- Activation : Tanh; Architecture [256, 64, 64, 64, 64]; Loss function : MSE.

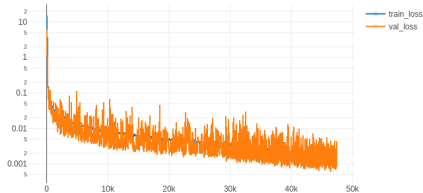


FIGURE 5 – Simultaneous decay of the train and validation loss functions respectively in blue and orange. Avoidance of overfitting.

- The MSE also decreases for **unseen** data (validation set).

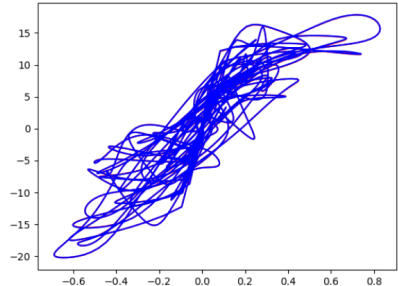


FIGURE 6 –  $(\epsilon_{xx}, \sigma_{xx})$  projection. Reference solution in red perfectly overlaid by the model prediction in blue.

Pros :

- ▶ Successfully train a model with synthetic data.
- ▶ Neural Networks could directly evaluate the derivative with **automatic differentiation**.

Limitations :

- ▶ Ensure solver stability by adding physical constraints.
- ▶ Deal with reduced databases.

3 methods :

1. modifying the data;
  2. the model;
  3. the training;
- by making some assumptions.

- Constraints : symmetry, positive definite properties of the tangent modulus  $\frac{\partial \sigma}{\partial \epsilon}$ .
- Semi-supervised methods : ensure physical constraints on **unlabeled** data  $\epsilon^*$  by penalizing the loss function to include **penalisation**.

$$J_{pen}(\theta) = \frac{1}{N} \sum_{k=1}^N \|\sigma_k - \sigma(\epsilon_k)\|^2 + \sum_i \frac{\lambda_i}{n^*} \sum_{\epsilon^* \in S} \mathcal{P}_i(\epsilon^*, \theta).$$

► Isotropy

- Invariants : 3 strain tensor invariants ( $I_1, I_2, I_3$ ).
- Data augmentation : Application of rotation.

$$\{\boldsymbol{\varepsilon}, \boldsymbol{\sigma}\} \rightarrow \{\{\boldsymbol{\varepsilon}, \boldsymbol{\sigma}\}\}, \{\boldsymbol{Q}^T \boldsymbol{\varepsilon} \boldsymbol{Q}, \boldsymbol{Q}^T \boldsymbol{\sigma} \boldsymbol{Q}\}$$

► Potential prediction - hyperelastic material

$$\boldsymbol{\sigma} = \frac{\partial \psi}{\partial \boldsymbol{\varepsilon}}$$

- Symmetry of the tensor is imposed.

- ▶ Integration step of the PyTorch model in MFfront.
  1. Build/Train/Test the model with PyTorch (Python).
  2. Save the graph, weights and normalization terms of the model with the TorchScript format.
  3. Read the model with the C++ PyTorch API.
  4. Include the PyTorch model in the MFfront file in order to predict the stress  $\sigma$  and the tangent modulus  $\frac{\partial \sigma}{\partial \epsilon}$  (thanks to auto-differentiation).
  5. Build the behaviour file libBehaviour.so .

```
@Include
{
  struct NN
  {
    ...
    torch::jit::script::Module module; // Neural Network
    ...
    inline std::pair<tfel::math::stensor<3u, double>, tfel::math::st2tost2<3u, double>>
    operator()(const tfel::math::stensor<3u, double> &strain)
    {
      std::vector<torch::jit::IValue> inputs; // Initialize the Torch inputs
      mfront2pytorchInputs(strain, inputs); // Convert tfel tensor to Torch

      auto outputs = module.forward(inputs).toTuple(); // Forward evaluation of the model

      torch::Tensor torchStress = outputs->elements()[0].toTensor(); // Stress
      torch::Tensor torchTangentOperator = outputs->elements()[1].toTensor(); // Tangent op.

      // Convert Torch tensors to tfel tensors
      tfel::math::stensor<3u, double> stress;
      tfel::math::st2tost2<3u, double> tangentOperator;
      pytorchOutputs2mfront(torchStress, torchTangentOperator, stress, tangentOperator);

      return {stress, tangentOperator}
    }
  }
}
```

```
@Integrator
{
  // Initialize the NN object (once for all)
  static NN nn("path_to_the_model.pt");
  // Update variables
  std::tie(sig, Dt) = nn(eto + deto);
}
```



- Solution of a mechanical problem with Finite Elements Simulation :
  - Using of the `MFrontNonlinearMaterial` and `MFrontNonlinearProblem` classes of the `mgis.fenics` module.
  - Uniaxial traction of a holed plate.

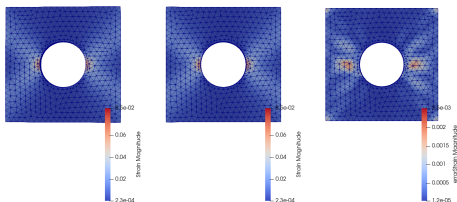


FIGURE 7 – Reference solution (left), model prediction (center), error representation of the strain field (right).

Field	Displacement $\mathbf{u}$	Strain $\varepsilon$	Stress $\sigma$
Relative error (%)	3.5	2.4	3.5

TABLE 1 – Errors on the holed plate trial.

- ▶ Use of neural network in order to predict constitutive behaviours.
- ▶ Introduce the concept of semi-supervised learning to ensure criterions.
- ▶ Integration of the model in mechanical simulation with MFront interface.

## What has not been shown...

- ▶ Dealing with time-dependency with Recurrent Neural Networks

- ▶ Up to now :
  - Un-noised data ;
  - Training the model on already known behaviours.
- ▶ Perspectives :
  - Adding noise in the training data (experimental noise);
  - Generating data with Representative Volume Elements (RVEs) : include heterogeneity, unknown behaviour and without mathematical formalism.

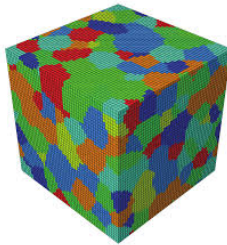


FIGURE 8 – Exemple of a Representative Volume Element of a polycrystal. R. A. RubioSarra, 2019)

$$\mathcal{P}_{sym}(\epsilon^*, \theta) = \left\| \frac{\partial \sigma}{\partial \epsilon}(\epsilon^*) - \frac{\partial \sigma}{\partial \epsilon}^T(\epsilon^*) \right\|_F^2.$$

$$(x_1, x_2, \dots, x_{n-1}, x_n) \text{ iid. } \mathcal{N}(0, 1), \quad \mathcal{P}_{dp2}(\epsilon^*, \theta) = \frac{1}{N} \sum_{i=1}^N \max \left( 0, -x_i^T \frac{\partial \sigma}{\partial \epsilon}(\epsilon^*) x_i \right).$$

- Take into account the loading history :  $\sigma(t) = f(\{\varepsilon(s)_{s \leq t}\})$ .
- Use of **recurrent** neural networks (RNN) : Introduction of a **hidden state** and **shared weights**.
- Link to the internal variables formalism :

$$h^{t+1} = P(\varepsilon^{t+1}, h^t; \psi, \phi),$$

$$\sigma^{t+1} = \hat{\sigma}(\varepsilon^{t+1}, h^{t+1}; \psi),$$

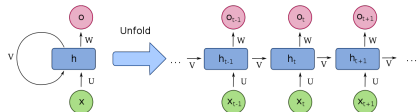


FIGURE 9 – Recurrent neural networks architecture.

- Data sequences generation for the RNN model training.
- Thermodynamic principal : Prediction of a free-energy potential and positivity of the dissipation rate using penalisation.

```
// Allocate space to store hidden variables
@StateVariable real h[100];
@Integrator
{
    // Initialize the model (once for all)
    static NN nn("path_to_the_model.pt");

    // Update stress, tangent modulus & hidden variables
    auto result = nn(eto + deto, h);
    sig = result.stress;
    h = result.hiddenState;
    Dt = result.tangentOperator;
}
```