Tutorial Machine Learning in Solid Mechanics (Winter term 2025–2026)

Task 1: Feed-forward neural networks

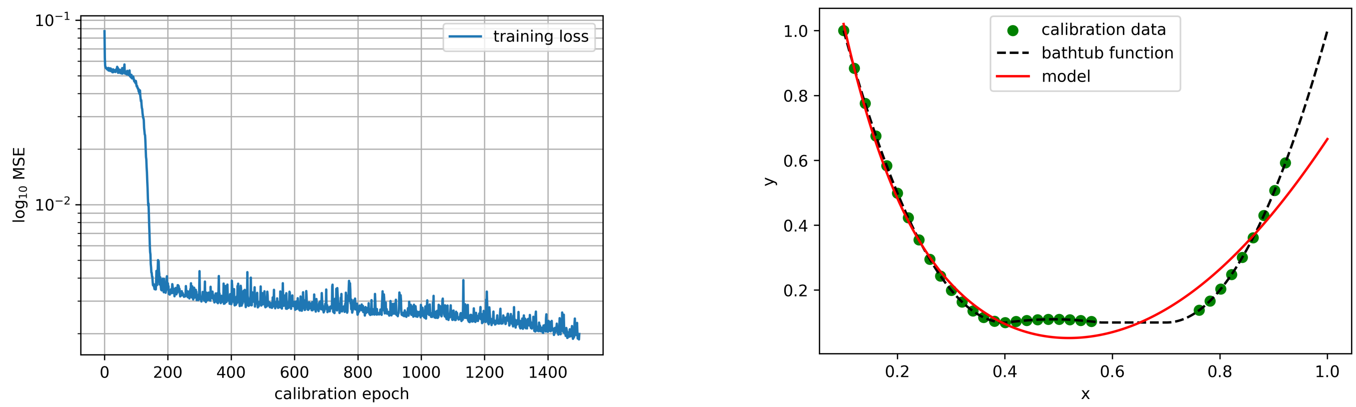
Dominik K. Klein, M.Sc. , Jasper O. Schommartz, M.Sc. , Prof. Dr. Oliver Weeger
October 14, 2025

1 Bathtub function

In the first example, a one-dimensional *bathtub* function, see Fig. 1b, is to be approximated with feed-forward neural networks (FFNNs). This basic example provides an introduction to JAX and some important aspects of neural networks (NNs). Visit the GitHub repository CPShub/TutorialMLinSolidMechanics and go to the folder “ffnn_introduction”. There you find three files: *data*, which generates the data for the *bathtub* function, *models*, where the NN is defined, and the *main* file which executes all tasks. Download the files and run the *main* file. Alternatively, you can run the code on Google Colab. Both the progress of the loss function (here: the Mean Squared Error) over the calibration epochs, see Fig. 1a, as well as the data and the model prediction are visualized, see Fig. 1b. Note that only a part of the overall dataset is used for calibration. For the remaining data, which the model does not see in the calibration process, the NN interpolates and extrapolates, respectively.

1.1 Hyperparameter sweep

The *architecture* of a FFNN is described by *hyperparameters*, i.e., by its number of hidden layers, the number of nodes in each of them and the activation function in the nodes. (Note that you do not have to evaluate all of the combinations of the below mentioned hyperparameter sweep, but only enough to understand the influence of the different hyperparameters.)



(a) MSE of the model calibration

(b) *Bathtub* function and (non-converged) NN model

Figure 1: Hyperparameter sweep

- Vary the number of hidden layers in $[1, 2, 3]$ and the number of nodes in $[4, 8, 16]$.
- Vary the number of epochs in the parameter optimization process in $[500, 1000, 2500, 5000]$ for three different NN architectures.
- Use different activation functions, e.g., *Relu*, *Softplus* and *Sigmoid*.
- What do you observe? Which architecture would you prefer?

1.2 Input convex neural networks

We now assume that the *bathtub* function describes a physical process which we know to be *convex*. Note that the data points introduced above do not describe a convex relationship, in particular for $x \in [0.4, 0.55]$. In practice, it might for instance occur that a dataset, which from a physical perspective should yield a convex relationship, is non-convex due to measurement inaccuracies and uncertainties. This makes it even more important to include the convexity condition in the NN, as otherwise the model would be calibrated to an unphysical behavior.

So far, a standard FFNN architecture was used, which is in general not convex. The next task is to adapt the code in *models* so that the resulting neural network becomes an input convex neural network (ICNN). For this, the following network architecture must be implemented:

- Convex activation functions in the first hidden layer, here: the *Softplus* function, see Fig. 2.
- Convex and non-decreasing activation functions in every subsequent hidden layer, here: the *Softplus* function with non-negative weights.
- A convex and non-decreasing output layer, here: a linear activation function with non-negative weights.

Note that JAX provides simple commands to adapt activation functions.

- Calibrate a standard FFNN and an ICNN to the data of the bathtub function.
- Compare the results for the general FFNN and the ICNN. What do you observe? Which implications has this on the interpolation of data and the model prediction?
- Can you use other activation functions to create an ICNN?

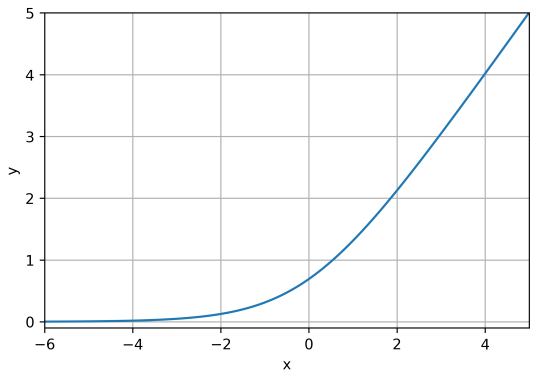


Figure 2: *Softplus* activation function $f(x) = \log(1 + e^x)$

2 Two-dimensional functions

While JAX provides a variety of tools, sometimes the implementation of custom layers is required. This was already done in the previous section: In the module *models*, the class “_x_to_y” generates a custom layer which uses a FFNN to map a scalar-valued input to a scalar-valued output. Before adapting the code, examine the following:

- Which dimensions do the inputs / outputs of this model have?
- How would they change for a two-dimensional input?
- In which parts of the code are dimensions of the input / output defined?

2.1 Data generation

Generate data with the functions

$$\begin{aligned} f_1(\boldsymbol{x}) &= x^2 - y^2\,, \\ f_2(\boldsymbol{x}) &= x^2 + 0.5y^2\,, \end{aligned} \tag{1}$$

and the derivative

$$\frac{\partial f_2(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{pmatrix} 2x \\ y \end{pmatrix}\,, \tag{2}$$

where $\boldsymbol{x} = (x, y)$. Use $x, y \in [-4, 4]$ with 20 equidistant points in each direction.

2.2 Trainable custom layer

Implement a FFNN which maps a two-dimensional input to a scalar-valued output by adapting the code of Sec. 1. This FFNN will in general not be convex. In the next step, adapt the model so that it becomes an ICNN. Calibrate both the convex and the non-convex NN to the data of f_1 and f_2 generated in Sec. 2.1.

2.3 Sobolev training

In Sec. 2.2, a model was implemented which maps a two-dimensional input to a one-dimensional output using an ICNN. In the next step, adapt the model so that it has two outputs: both the output of the ICNN as well as the derivative of the ICNN. For a given ICNN with input $\boldsymbol{x}$ and parameters $\boldsymbol{p}$, this equals the two outputs of the model

$$\begin{aligned} \text{Output}_1 &= \text{ICNN}(\boldsymbol{x};\, \boldsymbol{p}) \in \mathbb{R}\,, \\ \text{Output}_2 &= \frac{\partial\, \text{ICNN}(\boldsymbol{x};\, \boldsymbol{p})}{\partial \boldsymbol{x}} \in \mathbb{R}^2\,. \end{aligned} \tag{3}$$

For the derivative of the ICNN, implement a differential layer using JAXs “GradientTape” function. Note that the ICNN has to placed within the GradientTape function so that JAX can apply automatic differentiation correctly.

The resulting model can be calibrated both on the original output of the ICNN (meaning Output_1) and on the derivative of the ICNN (meaning Output_2). Calibrating a NN on its gradients is usually referred to as *Sobolev training*. Apply the following calibration strategies:

- Calibration on f_2.
- Calibration on f_2 and its gradient.
- Calibration only on the gradient.

Evaluate the NN-model prediction for both f_2 and its gradient for all three strategies.

2.4 Hyperelasticity

In Sec. 2.3, a NN model was implemented that employs a Sobolev training strategy, i.e., the NN is calibrated through it’s gradients. Apply this model to a simple 1D material modeling problem. There, the aim is to find a relation between the strain $\varepsilon \in \mathbb{R}_{>0}$ and the stress $\sigma \in \mathbb{R}$ it evokes in a solid body. In hyperelasticity, this relationship is established via en energy potential $W = W(\varepsilon)$, from which the stress follows as

$$\sigma = \frac{\partial W(\varepsilon)}{\partial \varepsilon}\,. \tag{4}$$

The potential W can be represented with a NN, i.e., $W = \text{NN}(\varepsilon)$. This NN potential can be calibrated on the stress

$$\sigma^{\text{NN}} = \frac{\partial \text{NN}(\varepsilon)}{\partial \varepsilon}\,, \tag{5}$$

by employing Sobolev training. Investigate the following NN models:

- The NN potential is a standard FFNN
- The NN potential is an ICNN

The models are to be calibrated on experimental data of soft polymers. For each group, a different dataset will be provided. Investigate both interpolation and extrapolation with the NN models described above.