Tutorial Machine Learning in Solid Mechanics (Winter term 2022–2023)

TECHNISCHE UNIVERSITÄT DARMSTADT

CPS Cyber-Physical Simulation TU Darmstadt

Task 1: Feed-Forward Neural Networks

Dominik K. Klein, M.Sc. , Yusuf Elbadry, M.Sc.
October 26, 2022

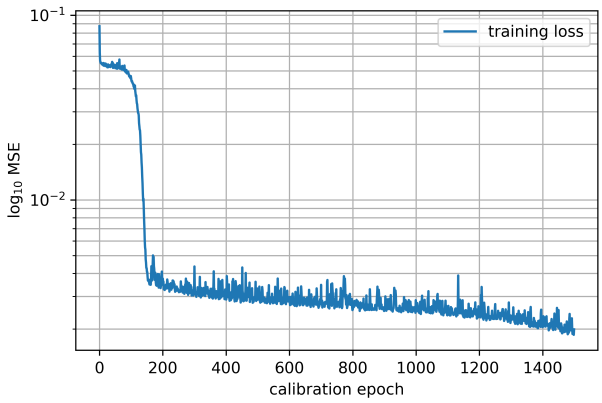
1 Bathtub function

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 TensorFlow and some important aspects of neural networks (NNs). Visit the GitHub repository CPShub/TutorialMLinSolidMechanics and go to the folder "01_FFNNs". 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. 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.

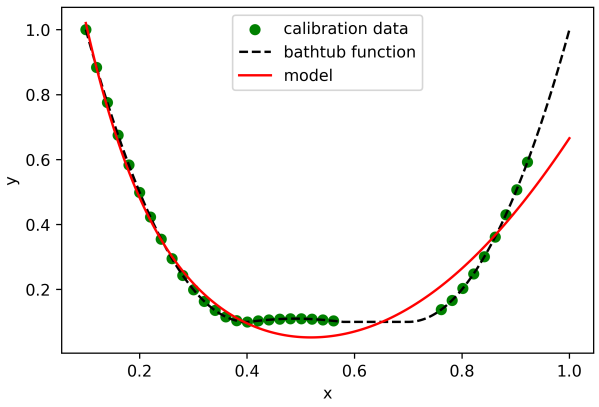
1.1 Hyperparameter sweep

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.)

- Vary the number of hidden layers in $[1, 2, 3]$ and the number of nodes in $[4, 8, 16]$.



(a) MSE of the model calibration



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

Figure 1: Hyperparameter sweep

- 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 datapoints introduced above don't describe a convex relationship, in particular for $x \in [0.4, 0.55]$. This might also occur in practice: A dataset which from a physical perspective should yield a convex relationship might become non-convex due to inaccuracies in some measurement. 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 TensorFlow provides simple commands to adapt activation functions.

Compare the results for the non-convex 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?

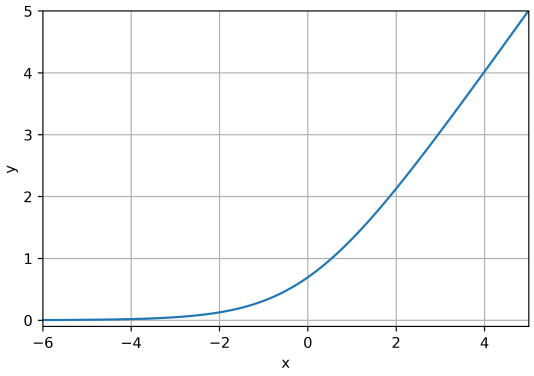


Figure 2: *Softplus* activation function $f(x) = \log(1 + e^x)$

2 Two-dimensional functions

While TensorFlow 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. This is a trainable FFNN, meaning that it can be calibrated to some given dataset (in our case the *bathtub* function). 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 Non-trainable custom layer

In the first step, we use the framework that TensorFlow provides to implement *non-trainable* layers and use them for non-trainable models. Strictly speaking, this is not a FFNN anymore. However, non-trainable layers are a commonly applied method in physics-augmented NNs. Here, we will use non-trainable models to generate data. Implement two different models consisting of only one non-trainable layer each, which map a two-dimensional input to a scalar output according to:

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

Then, generate data for $x, y \in [-4, 4]$ with 20 equidistant points in each direction.

2.2 Trainable custom layer

Implement a trainable 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 generated in Sec. 2.1.

2.3 Sobolev training

Adapt the non-trainable model of Sec. 2.1 for f_2 so that it provides not only the value of f_2 as output, but also its gradient, i.e.

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

Then, regenerate the data as described in Sec. 2.1, now for both the functions and its gradient.

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 TensorFlows “GradientTape” function. Note that the ICNN has to placed within the GradientTape function so that TensorFlow 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.