Week 5: Data-based response prediction of metal and composite structures

Artificial Intelligence for Aerospace Engineering

AE-2224II - Q3 2022/3

Giorgio Tosti Balducci

February 17, 2023

This document contains the instructions for the laboratory session of week 5. Please read it page by page, working on the exercises as you go.

In this laboratory, we will deal with typical structural response prediction tasks. Often, obtaining data that describes the behaviour of a structure under loading is an expensive operation. The prime example of this is data obtained from experiments. You will likely have to book facilities, order material, manufacture and/or prepare the specimens, accurately place sensors, etc. On the other hand, in case of computer simulations like finite element (FE) analyses, the economic cost might be lower, but the time expense can still be significative. Think for instance of very accurate fracture analyses on composite panels, maybe with multiple cutouts, stiffeners, etc.

In practice, given few data points, what we can do is to make a(n educated) guess on a model and *regress* to *generalize* on the loading configurations that we don't know. In this lab, we will be doing exactly this. First, we will do regression with simple polynomials and then using multi-layer perceptrons (MLPs), all the while looking at the tactics that ensure good generalization and avoid overfitting.

You will find the files necessary to complete this laboratory in directory `week5/constitutive-regression` of your docker container.

Data: You can find all files in the directory `data/raw_data`. The data you will deal with has two main origins:

1. Experimental testing of a metal bar loaded in tension and undergoing plasticity. This dataset consists of the single file `metal_plasticity.txt`. The first column represents the imposed displacement (input), while the second column is the reaction force of the bar (output).

2. Finite element analyses of a quasi-isotropic composite panel with an elliptic cutout in the middle. The panel is brought past the point of damage initiation and failure. This dataset is spread out over multiple files, according to the nomenclature `EaX_EbY_EsZ.txt`, where X, Y and Z can be positive, negative or zero. Every file represents a different FE simulation and every row in the column is a different load increment. The first three columns are the input and correspond to the *strains* imposed at the edges in the three directions, while the remaining three columns are the output and correspond to the *stress* values averaged over the edges. If you look at the contents of these files, you will notice

that the X, Y and Z values in the file names correspond to the maximum positive or negative strains in the corresponding direction.

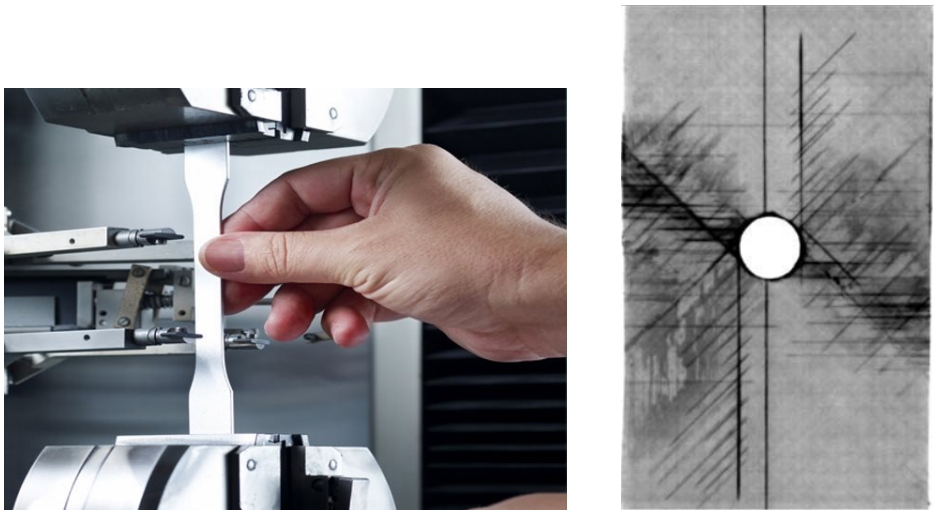

Figure 1: On the left, a tensile test of a metal *dog-bone* specimen. The data in `metal_tensile.txt` were obtained with a similar test. On the right, an open hole composite plate with a complex damage pattern. The data in the `EaX_EbY_EsZ.txt` was obtained from finite element simulation of a similar composite plate, but with an elliptical (rather than circular) cut-out.

Further comment on notation: the three loading directions for the composite panel dataset are named *a*, *b* and *c* in the file names, but in the exercises we will call them 1, 2 and 3 correspondingly.

Aim: The aim of this laboratory session is to practice with two different regression techniques (polynomial regression and MLPs), by using them to predict structural responses based on real-world data.

About the code: Each of the exercises in `week5/constitutive-regression` consists of two files, i.e. `exe_x.py` and `exe_x_utils.py`. Please complete the functions in `exe_x.py` that are *before* the `if __name__ == ´__main__´` line. Each of the functions to complete shows a *docstring* that gives some implementation details and some references and that specifies what the function has to return. Function arguments are provided.

You are not *required* to modify the rest of the `exe_x.py` file or `exe_x_utils.py`. However you should feel free to do so if you want to have a better understanding of it or see how the results change, by changing some lines of code (e.g. modifying the hyperparameters). Our recommendation is to *focus on the functions of the assignment first* and leave the code after `if __name__ == ´__main__´` unchanged. In this way, if your implementation of the functions is correct, the script will give you the results that you expect. Then submit your implementations on WebLab. If you have time and curiosity later, go wild and play with the code!

Exercise 1: Predicting the structural response of a metal bar under tension with a polynomial function.

In this exercise, we want to use the data in the `metal_tensile.txt` file to fit a polynomial that can predict the reaction of the bar, given any loading condition.

For this exercise, we will make mostly use of the *scikit-learn* library.

Part 0: Data read-out and preparation

Here the data is read, split in a train and test set and scaled. This part is already done for you. However notice what data is being used. We will be excluding the first sample from the data, which is the zero-loading condition. This is because the linear region has not other samples and fitting a global polynomial might be a hard task and a waste of resources. We will focus only on the plasticity region (`X[1:]`, `y[1:]`). In practice, the overall model would then be a piecewise polynomial which is linear in the linear region and of the 'optimal degree' in the plasticity region.

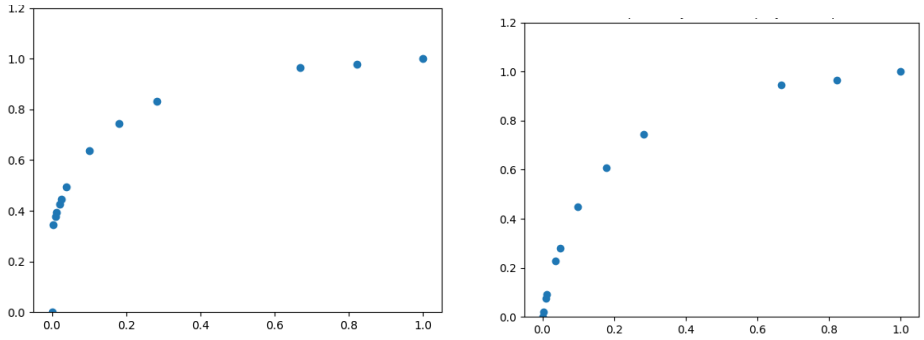

Figure 2: Samples from the metal tensile test. The left figure includes all points, while the right one leaves the first out (only plasticity region).

Part 1: Fit polynomials of different degrees to the plasticity data

Here we will be fitting polynomials of increasing degree to the training data in the least-squares sense. Furthermore we will predict on the test data for each degree and compute the mean squared error (MSE) and the R2 score.

Task: implement the functions

1. `create_ls_regression_model`
2. `fit`
3. `predict`
4. `mse`
5. `rsquared`

Please use Scikit-Learn to complete this task and check the documentation for details.

Question: what do you notice for high polynomial degrees?

Part 2: K-fold cross validation

This dataset is a perfect example of when you have few data and it's hard to take a hold-out set (test set) to see how well your model generalizes [1]. In cases like these, we can do cross-validation (CV), which allows us to do multiple training, each time holding out a different 'piece' of the entire dataset.

In particular, we'll be practicing here with *K-fold* CV. In this algorithm, the dataset is split in K chunks and K different trainings are done. For each training, we take one chunk as a validation set, to evaluate the generalization of the model and use the other $K-1$ chunks for training. After each training, the next chunk will be used as validation set (rest for training) and we proceed until all chunks have been used once for validation.

What we will be doing here is an example of hyperparameter selection, where the hyperparameter is the polynomial degree. For each degree, we will do cross validation and compare the statistics of the validation error at the end.

Task: implement the `kfold_cv` function. In this part of the exercise, you *cannot use the Scikit-Learn modules*, i.e. you will have to implement the function from scratch.

Question: what is the polynomial degree that generalizes best on the plasticity data?

[1] Notice that we *did* take out a test chunk before from our data, but this was necessary for Part 1. In reality, with such a small dataset, we would use it entirely during a cross-validation procedure.

Exercise 2: Fitting the composite panel response with MLP and early stopping

In this exercise, we will consider a single file from the composite panel dataset (`Ea4999_Eb10000_Es10000.txt`) to have a first feeling of the dataset and of how a MLP can fit it. Furthermore, we will implement early stopping as an easy but effective way to avoid overfitting.

For this and the next two exercises, we will make mostly use of the *PyTorch* library.

Part 0: Data read-out, preparation and definition of the model

As before, data reading and preparation has already been done. What is left for you to do is to create a deep neural network (here we will be using this term as synonym to MLP) of fixed width, meaning that every hidden layer has the same number of neurons.

Task: implement the `create_fixed_width_dnn` function.

Part 1: Training the MLP

Here, you will implement a function to train the MLP. The function is given a model, loss function, optimizer training and validation data and returns the training and validation losses during the training epochs. Notice that there are also a number of optional arguments, that are the maximum number of epochs, a tolerance on the training loss and a flag for verbosity. In this function, you will implement a basic stopping criterion, namely that the function should return whenever the training loss goes below tolerance. In the next part of this exercise, we will practice with a more sophisticated stopping technique.

Task: implement the `train_model` function.

Part 2: Include early stopping

In machine learning, the aim is to generalize well on unseen data. Therefore we don't want to continue training if the validation loss starts to increase, because that is likely a sign of overfitting.

In this part of the exercise, you will implement an early-stopping criterion based on the value of the validation loss. The logic (or pseudo-code) is the following

1. Take as input the current validation loss
2. Check if the valdiation loss is lower than the minimum value seen so far
    - if 'yes', update the minimum and continue
    - if 'no', check if the validation loss has increased by more than a tolerance (`min_delta`)
        - if 'yes', increase a counter
            - check if the counter is higher than a predefined 'patience'
                - if 'yes', stop training
                - if 'no', continue
        - if 'no' continue

Task: implement the following functions:

1. `early_stop` in the class 'EarlyStopper'
2. `train_model_early_stop`. To implement this function, please copy the code from `train_model` implemented before and adjust it to include early stopping.

Question: how does early stopping improve the MLP final predictions?

Exercise 3: Learning rate optimization for composite panel data

Also in this exercise, we will be using the composite panel dataset, but look at a different file (`Ea-5000_Eb-5000_Es10000.txt`). In the previous exercise, we optimized to find the weights and biases of a network that could fit (and not overfit) the stress response of the composite data. However, we just assumed some best-guess values for the *hyperparameters*. Can we do any better than guessing?

In this exercise, we will make a random search to find a 'good' learning rate for training. Notice that even though random searching does not exhaustively search the parameter space and doesn't actually find an optimum, it's a very used techinque in practice, both for its efficiency and ease of implementation.

Part 0: Data read-out, preparation and definition of the model

Same initial preparations as before. Here you won't have to implement anything new.

For local testing on vscode: copy the `create_fixed_width_dnn` function implemented earlier.

Part 1: Finding a good learning rate

Here is where you will implement the random search for the learning rate. In a random search, the idea is to sample values of the hyperparameter of choice from a probability distribution. You will use the so-called *log-uniform* distribution, which makes sure that the logarithm of two limit values is sampled uniformly. This is useful, because we want to span over several order of magnitudes that the learning rate can assume. If we were to use a more common linear uniform distribution, we couldn't cover the useful range in a small number of repetitions.

Task: implement the `lr_random_search` function.

Note: the automatic grading of this exercise may well take between 1 and 2 minutes. So don't worry if you have to wait for the results of your test.

For local testing on vscode: copy the `EarlyStopping` class and the `train_model_early_stop` function implemented earlier.

Part 2: Training with optimal learning rate

There is no task to be done here, but you can just run this part of the code and observe how the training and validation losses decrease with the optimal learning rate. To experiment further, try to change the learning rate of some order of magnitude from the optimal value and notice the difference.

Exercise 4: Fitting the entire composite panel dataset with and without batching

Finally, in the last exercise we will be using the entire composite notched panel dataset, i.e. the whole collection of files with names `EaX_EbY_EsZ.txt`. A major change between now and before is the sheer number of samples that we will be using, which is of the order 10^5 even just for the training set.

When dealing with large datsets, *batching* should always be used during training of the model. Batching consists in dividing the dataset in chunks (or *batches*) and update the gradient based only on the data in that batch, rather than the entire dataset. In this way, we don't follow the fastest descent route anymore, which likely means that more epochs will be required before convergence. However, we gain several advantages like less memory requirements, stability of gradient descent and the possibility of processing batches in parallel.

Part 0: Data read-out, preparation and defintion of the DataSet

Most of the data reading and preparation has been done for you, but in this part you're required to perform an extra step. In order to work with batches, PyTorch uses `DataLoader`s, which are iterators that loop over the batches. A `DataLoader` object requires another PyTorch object, namely a `DataSet` as input, as well as other optional arguments, such as the batch size (we will deal with this one later).

Task: implement the class `DatasetExe4`.

Part 1: Training without batching

In this part of the exercise we will use a the entire dataset as done before. However, we want to build a training function that works with dataloaders having a generic batch size, in order to also use it later when we'll split the data in batches[2]. As a starting point, copy the code of `train_model_early_stop` used in the previous exercises and modify for batching (leave the input arguments as they are in the template of `exe_4.py`). Notice that his time you won't have access to the entire training dataset, so your output training loss has to be different than the previous implementations (think, for instance, to the a batch-average per epoch or a rolling average). Furthermore, compute the average wall-clock time over the epochs and return it as a third output of the function.

Task: modify `train_model_early_stop` to work with a dataloader as input and to process batches.

For local testing on vscode: copy the `create_fixed_width_dnn` function and the `early_stop` method in the `EarlyStopper` class, as implemented earlier.

Part 2: Training with batching

In this final part we will fix a batch size and see how this changes the training. From the previouse exercise, you should be able to see both the training and validation losses and the average time per epoch. You won't have any coding tasks in this part of the exercise, but you should notice how batch training compares with training on the full dataset.

Question: how does convergence compare between no-batching and batching? How about the time per epoch?.

[2]Note that in this part of the exercise a single batch is enforced by defining a dataloader with a batch size equal to the size of the dataset