# 1 Overview

In this exercise sheet, you will train fully connected neural networks, from tabular data. We will perform this using the PyTorch, but without relying on fastai.

## 1.1 Data Loader and Cleanup

Out first step is to implement the data loading mechanism.

- The function `extract_numpy_from_df`. It accepts a Pandas dataframe and some indication which columns should be input and output. It returns a tuple $(X, Y)$, where $X$ is the input columns and $Y$ is the output columns, both encoded as numpy array in float32 data format.
- The function `clean_data` deletes any defective rows in the data.

a) Argue if Unit Tests are required for an implementation of the functions.

- `extract_numpy_from_df`: Test if the function returns the correct array for different positions of input and output columns.
- `clean_data`: Test if there are still a sufficient amount of data points left after the function has been called.

b) Implement the functions as Python code.

c) Implement the functions above.

d) Consider the clean data function. What happens of the data contains many defective rows, or defective columns? Make an argument, to what extend the clean data should silently handle corrections, when it should adapt, or when it should fail.

- If the data contains many defective rows, the function should fail, because the data is not usable. You can handle this by checking the amount of nan values in the data and if it is above a certain threshold, the function should raise an error.

I needed the following time to complete the task:

## 1.2 Data Encoding

- All input columns are numerical and should be normalized. The function `normalize` accepts a numpy array and normalizes each column.
- The output is categorical and should be hot-1 encoded. The function `hot_1_encode` accepts a numpy array and a dictionary, that maps each value in the numpy array to a column index. For our dataset it has the form `codes = { 'category_1' : 0, 'category_2' : 1, 'category_3' : 2 }`.

a) Argue what Tests are required for an implementation of the functions.

- `normalize`: Test normalization on an array where all elements are the same
- `normalize`: Test if the function normalizes the array correctly
- `hot_1_encode`: Test if the function returns the correct array for different values of the input array

b) Implement the tests.

c) Implement the functions as Python code.

I needed the following time to complete the task:

## 1.3 Splitting, Randomization, Mini Batching

In the following, we will prepare the data for training:

- The available data points will be randomly split in a ratio of 80-20 in training data and validation data
- The learning takes place in mini batches. Each minibatch consists of a number (32 for a start) of items. The entire minibatch is represented as numpy array of dimensions ($n\_columns \times batch\_size$. Input and output are represented as separate numpy arrays.

a) Argue what Tests are required for an implementation of the functions.
   - `split_data`: Test if the function returns the correct amount of data points for training and validation and test if there are no duplicates in the data
   - `mini_batch`: Test if the function works as expected on a small dataset

b) Implement the tests.

b) Implement the functions as Python code.

I needed the following time to complete the task:

## 1.4 Training Loop

The signature and some source code is already present in the jupyter lab. In summary, a single epoch consists of the following steps:

- Determine the number of iterations (size of training data set / batch size).
- For each iteration:
- assempe a torch tensor for $X$ and $Y$ for that specific mini batch
- delete the accumulated gradients in the optimizer
- compute $\hat{Y}$ by calling model.forward
- compute the loss to compare $\hat{Y}$ and $Y$
- call `loss.backward()` and `optimizer.step()`

If you get stuck, you can google for "PyTorch Training Loop".

I needed the following time to complete the task:

## 1.5 Optimal Training Parameters

Now, we experimentally determine optimal training parameters. You will need to organize your source code to allow for the following experiments. Consider that a network is initialized with random values only when the model is created. If training is interrupted and continued, weights and biases are preserved.

a) Find an optimal learning rate. Argue which rates you tested and how you determined an optimal value.
   - I tested learning rates from 0.1 to 0.0001 in logaritmic steps.

b) Find an optimal learning batch size. Argue which batch sizes you tested and how you determined an optimal value.

- I tested batch sizes from 32 to 124 in steps of x2 the previous value.

b) Find an optimal network architecture (number of layers and nuber of features for each layer). Argue which architectures you tested and how you determined an optimal setup.

- I tested a very simple architecture with 2 layers and few nodes, medium complexity with 2 layers and more nodes, and 3 layers and lots of nodes

I needed the following time to complete the task:

## 1.6 Adaptation to Iris Dataset

Finally, we will adapt the source code to work with the Iris dataset that we used in previous exercises.

a) Analyse, which parts of your source code will need to changed. Describe in your own words an implementation plan for the adaptation.

- I have to load another dataset, and decrease batch size, the rest stays the same. i test less hyperpar to reduce training time.

b) Make the necessary adaptations and solve the classification problem on the Iris dataset.

c) Make a comparative study to compare the performance of the neural network with your decision tree. How do they compare for the MECS dataset and for the Iris dataset?

- The neural network is much more flexible and can learn more complex patterns. The decision tree is faster to train and easier to interpret.

I needed the following time to complete the task: