# POLAR BACKGROUND PREDICTION

DOCUMENTATION: INDUSTRIAL REPORT

**Stéphane Liem NGUYEN**

**Nicolas PRODUIT**

**Computer Science Department**

**Department of Nuclear and Corpuscular physics**

**University of Geneva**

August 3, 2023

**Contents**

**ABSTRACT**

This industrial report is for documentation purposes and should at least cover how the project is structured, how to use it, and information on the tools we've used.

***Keywords*** POLAR data · Regression · Anomaly Detection

## 1 Introduction

This project aims at first constructing a model representing normal behaviour and then use it to detect abnormal behaviour, Gamma Ray Bursts (GRBs).

This normal behaviour, in which there shouldn't be any Gamma Ray Bursts (GRBs), will be called as the *background*.

The data that we use was captured by the POLAR detector which was mounted on the Tiangong-2 spacelab. However, as our examples are not all labelled as normal/abnormal, the model that we construct based on a subset of this data is therefore not a true model of normal behavior. The model that we construct is therefore polluted by unknown GRBs or also called anomalies, outliers or abnormal behaviors.

More precisely, our model should predict well photon rates when there's no GRBs and predict badly photon rates when there's an actual GRB. One way of defining how bad the prediction should be in order to be considered as an outlier or anomaly is to place a threshold on the residuals, i.e. the difference between the target and the prediction.

We tried different models such as a linear regression and a fully connected feedforward ANN (MLP) to predict photon rates from different measurements such as: magnetic field, latitude, longitude, time since South Atlantic Anomaly.

Although this report won't cover the full extent of what we've tried and won't explain the linear regression model, we will explain how the data is processed before being fed to our model for training.

## 2 Project structure

The project is composed of at least these files/folders:

- `./checkpoints`: configuration files, PyTorch checkpoints (model weights, optimizer etc.) for different runs
- `./config`: configuration file(s)
- `./data`: datasets in ROOT-CERN format or other (e.g. `.pkl`, `.csv`)
- `./logbook`: markdown file explaining some details of what we've done each week
- `./notebooks`: jupyter notebooks and python scripts used at the beginning of the project for data exploration/visualization and for creating basic models to predict photon rates (using for example Scikit-learn Multilayer perceptron). Also contains jupyter notebooks related to PyTorch models for results visualization.

- `README.md`
- `requirements.txt`
- `./results`
- `./src`: Python scripts for processing the data, training our model, visualizing our predictions, and other useful tasks

The `README.md` explains what is needed to quickly get started (installation, training phase, visualization of results). However, some details might be omitted there and are therefore explained in this report.

For the sake of completeness, we also explain again what is explained in the `README.md`.

## 3 Installation

- Install PyTorch

```
pip3 install torch torchvision torchaudio\
--index-url https://download.pytorch.org/whl/cu118
```

- And other dependencies from `requirements.txt`:

```
pip3 install -r requirements.txt
```

The code was developed for Python `3.10.12` and `3.10.6` and with `torch 2.0.1` and `torchvision 0.15.2`.

## 4 Usage

1. Place your ROOT-CERN file into the `./data` folder.
2. Change the `./config/trainer.yaml` config file with the correct filename under dataset.filename

- `python src/main.py` to run the training phase
- `python src/main.py wandb.mode=disabled` to run the training phase without using weights and biases
- `python src/visualizer.py` to load pretrained model, plot loss and predicted photon rates for validation set.

You can change the `./config/trainer.yaml` if you want a different model architecture, different hyperparameters, features etc.

### 4.1 Configuration file

The following example[1] of `./config/trainer.yaml` configuration file is nearly self-explanatory about how to specify a different model architecture, different hyperparameters, features etc.:

```yaml
wandb:
  project: POLAR-background-prediction
  mode: online
wandb_watch: True  # log in w&b model & criterion
common:
  seed: 42
  n_epochs: 200
  device: cuda
dataset:
  filename: data/nf1rate.root
  save_format: null  # won't save in .pkl nor .csv
  # new_columns: []
  new_columns:  # not necessarily used in the features or targets
    - rate_err[0]
    - rate[0]/rate_err[0]
  feature_names:
    - unix_time
    - glat
    - glon
    - altitude
    - temperature
    - fe_cosmic
    - raz
    - decz
    - rax
    - decx
    - is_orbit_up
    - time_since_saa
    # - crabarf
    # - sun
```

[1] not necessarily the one we use

```yaml
    # - sun_spot
    - B_r
    - B_theta
    - B_phi
  target_names:
    - rate[0]/rate_err[0]  # had to specify in new_columns
  filter_conditions:
    - rate[0]/rate_err[0] > 20
  split:
    type: periodical
    periodicity: 200  # # of samples for the periodicity, for type periodical
  train:
    size: 0.6
    batch_size: 200
    shuffle: True
  val:
    size: 0.2
    batch_size: 200
  test:
    size: 0.2
    batch_size: 200

model:
  type: MLP
  inner_activation_fct: ReLU
  output_activation_fct: null  # identity
  hidden_layer_sizes:
    - 100
    - 100

optimizer:
  hyperparams:
    # For adam optimizer
    lr: 1e-3
    betas:
      - 0.9
```

```
    - 0.999
  eps: 1e-08
```

However, we still provide below some textual explanation for what the different parts mean:

`common`:

- `seed`: seed for random number generator used by PyTorch, random, and numpy.
- `n_epochs`: number of epochs (iterations through the whole training set). That's the only end condition for the moment. **TODO: maybe change this with a stagnation end condition ?**
- `device`: device on which PyTorch tensors are and on which operations on them are performed.

`dataset`:

- `filename`: path to our dataset, it can be in ROOT-CERN format or another supported format (e.g. .pkl or .csv). The behavior changes depending on the format. It's only in root format that our script uses `new_columns` and `filter_conditions`[2] but the other formats will load directly (e.g. no filtering) the data as Pandas DataFrame and work with it under the hood.
- `save_format`: saves or not the Pandas DataFrame containing the dataset in some format. It is only used when filename is in ROOT-CERN format.

  Possible choices of `save_format`:

  - `null`: if don't want to save processed dataset
  - `pkl`: saves processed dataset in Pickle format (not split into train, validation and test set yet.)
  - `csv`: saves processed dataset in CSV format (not split into train, validation and test set yet.)

  **TODO: I didn't try if it actually works with the CSV format**
- `new_columns`: creates new columns in the Pandas DataFrame based on existing column names. These new columns can be used for filtering out examples or as target or feature for our model. Order can matter.
- `feature_names`: name of the input features that are used in our model.

[2] In other words, it's only in root format that it goes through the processing steps of creating new columns and filtering out rows/examples

- `target_names`: name of the target(s)/output(s) that are used in our model (e.g. photon rates).
- `filter_conditions`: filter based on existing columns. It can be used to filter out known GRBs based on their `unix_time`. Order can matter.
- `split`: how to split the dataset into subsets such as train, validation, and test set.

  Possible choices of `split.type`:

  - `random_split`[3]: uses PyTorch `random_split` to obtain a training, validation and test set (and PyTorch DataLoaders).
  - `periodical`: **TODO explain this**.

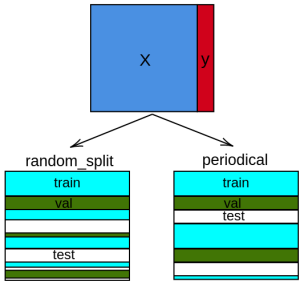


Figure 1: Diagram intuitively explaining the possible choices of `split.type`

- `[train|val|test].size`: The proportion of the data that goes into the training set, validation set and test set are specified by their `size` argument.
- `[train|val|test].batch_size`: mini-batch size used in the training, validation and test PyTorch DataLoaders. For the training phase for example, the model "uses" **TODO: find a better verb** `train.batch_size` examples to update its weights instead of the whole training set which could slow down the iterative process.

  Therefore, the iterative process cannot take true gradient steps towards a local minima. **TODO: make this more clear ?**
- Recall that the split is not saved in `save_format` format so every time we initialize our model (see `./src/trainer.py`, in the `init_datasets` method), it splits the dataset based on the what we wrote in the YAML configuration file.

[3]Actually, you can write anything and it will be `random_split` by default

`model`:

- `type`: choose which model to use

  Possible choice(s):

  - `MLP`: Multilayer perceptron (fully connected feedforward artificial neural network)

  Depending `model.type`, we can have different hyperparameters specifying for the architecture.

  For instance, for `MLP`:

- `inner_activation_fct`: activation function used for the hidden layers.

  Possible choice(s):

  - `ReLU`: Rectified Linear Unit $\max(0, x) = x^+$

- `output_activation_fct`: activation function for the last output layer (by default the identity function)
- `hidden_layer_sizes`: number of neurons for each hidden layer. We can specify a lot more than just 2 hidden layers by adding more rows.

`optimizer.hyperparams`: Adam optimizer's hyper-parameters. See PyTorch Adam's documentation for what can be changed.

**TODO: add schema explaining graphically what the data goes through and what happens to the dataframe ? and maybe where the code is situated**

## 5 Dataset

### 5.1 Create Pandas DataFrame from ROOT-CERN file and potentially save it

We show, in Figure 2, a flowchart of how the Pandas DataFrame is created/saved from the ROOT-CERN file. Note that some rows/examples could be unexplicitely filtered using `create_new_columns` (near point **2.**), e.g. if we perform division by zero when creating new columns, examples for which the operation is invalid are ignored/not kept. Note that `self.dataset_full` is a PyTorch Dataset.

### 5.2 Features and targets

Using `dataset.feature_names` and `dataset.target_names`, we can restrict our attention only to specific columns (see columns in red and blue from Figure 3). Note however

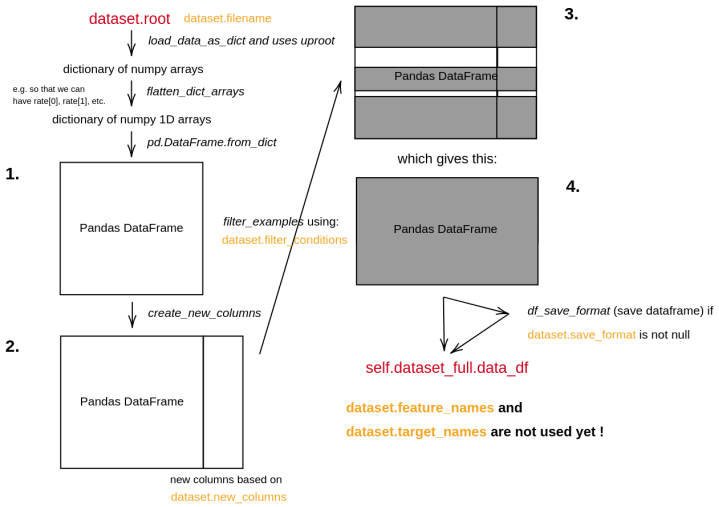


Figure 2: Pandas DataFrame created/saved from the ROOT-CERN file. In orange, we have some arguments we've specified in the YAML configuration file.

that retrieving an element/example from the PyTorch Dataset would apply some transformation on the feature and target values. In our case, we only transform the features by centering and reducing them based on the training mean vector and training standard deviation vector. Therefore, this transformation and retrieval do not make sense prior to splitting the data into training, validation and test set.

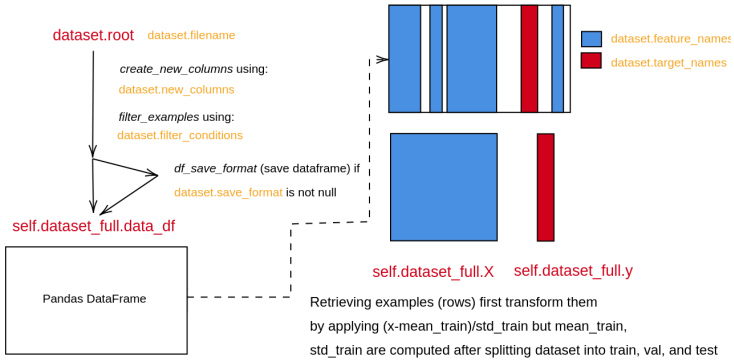


Figure 3: Features and targets from Pandas DataFrame

## 6 PyTorch

## 7 Weights and Biases

## 8 Hydra

## 9 Temporaray section for credits

- Code and project structure: https://github.com/eloialonso/iris
- 55 GRBs: https://www.researchgate.net/publication/326811280_Overview_of_the_GRB_observation_by_POLAR

## References