EPJ Web of Conferences **245**, 06025 (2020) https://doi.org/10.1051/epjconf/202024506025
CHEP 2019

zfit: scalable pythonic fitting

Jonas Eschle[1,*], *Albert* Navarro Puig[1,**], *Rafael* Silva Coutinho[1,***], and *Nicola* Serra[1,****]

[1]University of Zurich

Abstract. Statistical modeling and fitting is a key element in most HEP analyses. This task is usually performed in the C++ based framework ROOT/RooFit. Recently the HEP community started shifting more to the Python language, which the tools above are only loose integrated into, and a lack of stable, native Python based toolkits became clear. We presented zfit, a project that aims at building a fitting ecosystem by providing a carefully designed, stable API and a workflow for libraries to communicate together with an implementation fully integrated into the Python ecosystem. It is built on top of one of the state-of-the-art industry tools, TensorFlow, which is used the main computational backend. zfit provides data loading, extensive model building capabilities, loss creation, minimization and certain error estimation. Each part is also provided with convenient base classes built for customizability and extendability.

1 Introduction

Analysis of data collected from High Energy Physics experiments, such as the one stationed around the Large Hadron Collider, was performed predominantly in a C++ ecosystem. With the recent success of Deep Learning and a general raising popularity of Python, a large ecosystem for data analysis [3, 4] grew. Compared to dedicated toolkits in isolated ecosystem, this is not only used by scientists of a narrow field but shared amongst disciplines and with industry. The HEP community recently shifted more towards the Python language as the preferred tool to use in analysis making use of the open ecosystem. But while many shared tools can be used as-is, there is still a need for a small, dedicated HEP ecosystem built on top of other tools; these efforts are currently mainly guided by the scikit-hep project [5].

Data can usually be described by a parametric shape and the inference of the agreement between observed events and theoretically motivated models is often a crucial step in the whole chain of an analysis. The preferred tool for this task is RooFit [2], which is integrated into the ROOT [1] C++ toolkit. This step, model fitting, is a statistical tool that is used in many fields, in contrast to most other fields though HEP has strong requirements to such a tool. Due to inherent complex models and large data sets, performance becomes critical. Furthermore, many advanced features need to be supported such as custom model building and multidimensional compositions. As other fields often don't require this features, this leads to the fact that there is currently no library well integrated into Python and capable

*e-mail: Jonas.Eschle@cern.ch
**e-mail: albert.puig.navarro@gmail.com
***e-mail: rafael.silva.coutinho@cern.ch
****e-mail: nicola.serra@cern.ch

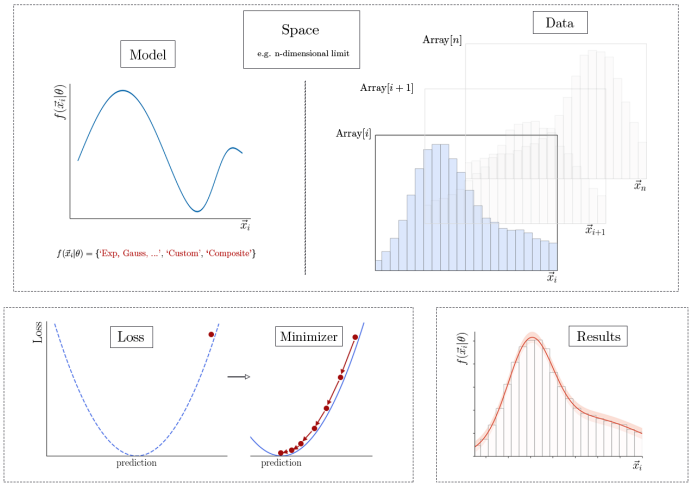

Figure 1. Workflow of zfit. Data is loaded and a model built, each one in the same space. Both together combine to a loss, which is minimized by the minimizer. The result allows to do basic error estimation.

enough of supporting the demands of HEP, including that there is also no standard which could be extended or simply built upon to provide the needs.

The zfit project was created to fill this gap; to provide a stable ecosystem for fitting in HEP. It focuses only on model fitting and sampling and has three essential goals: defining a workflow and stable API to allow other libraries to be built around; using a state-of-the-art computational backend that keeps up with modern architecture and delivers performance on the level of current C++ tools; providing a strong implementation that is easily extendable by the community using entry points such as pre-defined methods that can be overriden in base classes.

2 Workflow and API

The zfit workflow, illustrated in Fig.1 is divided into five essential parts – data, model, loss, minimization and fit result – which are all designed to be maximally independent and loosely coupled. They define the essential parts of a fit and can be replaced by another library. The features that are provided by the API are restricted to the necessary parts as many things such as plotting and pre-processing can be achieved by using other packages from the Python ecosystem.

Data and its respective weights are stored in a specific data object that abstracts away the origin of the data and the actual format. It has limited functionality and does not do any pre-processing apart from cuts given by the observable.

Models have methods to retrieve the value of the (normalized) function, to integrate over a certain region and to sample from it. They are parametrized by parameters, which can be free variables or built as a composition.

To have the data and models match also in the multidimensional case and to define ranges, there is an object `Space`. This can define arbitrary – or in the simple case rectangular – limits and observables that are then associated with the respective model or data.

The loss combines the data and the model. Typically, this is a negative log-likelihood. It provides methods to retrieve the value as well as the first and second derivative. Further-

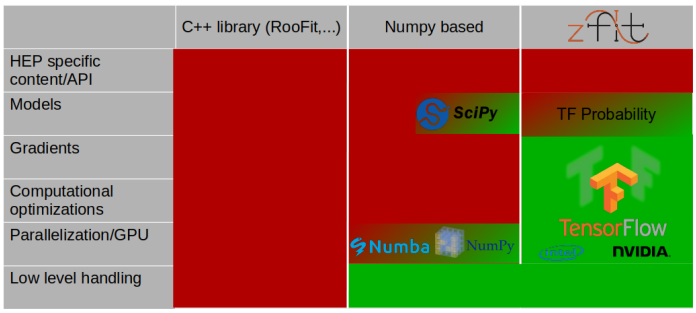

Figure 2. Comparison of the responsibilities of different approaches to build a fitting library. While Numpy and Python provide more convenience than C++ implemented tools, TF takes a lot of the required optimizations and leaves only a small part to a library that builds on top.

more, constraints of parameters such as additional information from other measurements of a desired quantity can be added.

Minimization is done with minimizers that provide a minimal set of methods to minimize a loss. It is usually considered stateless but can contain advanced methods for a more fine-grained minimization procedure.

A `FitResult` object is returned from the minimization that stores the information about the minimum found and provides simple error estimation. More advanced inference is left to tools[6] that are built on top of libraries which implement the zfit interface.

3 Computational Backend

Python is an interpreted language and offers great flexibility which comes at a cost of speed when using its basic building blocks directly to do heavy numerical operations. Therefore, libraries implemented in other languages exists that have a convenient API to define the mathematical operations. The de-facto standard is Numpy [3] and the format is fully supported in zfit. With the success of Deep Learning, more efficient computational libraries that support a wide arrange of optimizations – analytic gradients, automatic parallelization, caching and support for CPUs and GPUs – made it into the Python ecosystem. As flexibility and speed are always a trade-off in computing, these libraries can come with certain restrictions such as a limited set of instructions and additional tuning options.

zfit is mainly built on top of TensorFlow (TF), a Deep Learning framework supported by Google, that uses a static graph to speed up the computations. Using this framework has the advantage of requiring only a minimal set of optimizations to be implemented directly in zfit and to delegate the vast majority of the workload as visualized in Fig.2. This has profound implications: the whole maintenance and need to keep up with the newest developments in computing architecture as well as any future optimizations are factored out. This directly frees up development resources that can be used for work on the HEP specific features. Using TF compared to Numpy comes though with a few extra complications that have to be worked out. But most of these are hidden from the user in zfit and the library offers a similar user experience as the one found in other model fitting libraries. While the vast majority of models can be expressed using TF, there can be limitations. In zfit, there are two ways of using or just invoking standard Python instead. TF provides the possibility to wrap arbitrary functions. zfit supports to switch to numerical gradient calculation, as with arbitrary functions, the automatic gradient calculation of TF is not available anymore. As an even more flexible alternative, zfit

```
normal_np = np.random.normal(loc=2., scale=3., size=10000)

obs = zfit.Space("x", limits=(-2, 3))

mu = zfit.Parameter("mu", 1.2, -4, 6)
sigma = zfit.Parameter("sigma", 1.3, 0.1, 10)
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)

data = zfit.Data.from_numpy(obs=obs, array=normal_np)

nll = zfit.loss.UnbinnedNLL(model=gauss, data=data)

minimizer = zfit.minimize.Minuit()
result = minimizer.minimize(nll)

param_errors = result.error()
```

Figure 3. Example of a fit to a normal distributed dataset. First, the model and dataset is built, then a negative log-likelihood is created. A minimizer is instantiated and minimizes the loss and returns the result with which the error of the models parameters are estimated.

```
from zfit import z  # or directly TensorFlow


class CustomPDF(zfit.pdf.ZPDF):
    _PARAMS = ['alpha']

    def _unnormalized_pdf(self, x):
        data = x.unstack_x()
        alpha = self.params['alpha']

        return z.exp(alpha * data)
```

Figure 4. Implementation of the shape $exp(\alpha \cdot x)$. The model class inherits from the base class and overrides one of the entry points.

can be run in a mode where TF basically acts like Numpy. While this is less efficient in terms of computation speed, it allows to implement completely arbitrary Python code.

4 Implementation

A simple example of zfit with the workflow implemented and using TF as the backend can be seen in Fig.3. More complex examples will stick to the same workflow. While this allows to use the library for an array of simple to mediocre use-cases, a core element is the extendability of zfit.

While the interface of zfit is built to cope with the most difficult cases, there are base classes that conveniently satisfy the requirements of the API but still allow to override the public methods or steer their behaviour. An example is the implementation of a custom PDF as shown in Fig. 4

The method **_unnormalized_pdf** specifies only the shape without normalization. The latter is also non-trivial, since the model can be normalized over different ranges. The base class implements this logic to automatically integrate the function. This is done analytically if possible in the case of a registered integral, or numerically otherwise. The same applies for sampling from the pdf: if an inverted cumulative distribution function is registered, this

will be used, otherwise an accept-reject procedure is automatically invoked. Furthermore, any of the exposed methods such as **integrate**, **sample** or **pdf** can be completely overridden by implementing the same method with a leading underscore, e.g. **_integrate**. This still preserves more useful functionality that the base class provides, such as input cleaning or the automatic handling of disjoint ranges. With this functionality, it allows to directly implement arbitrary shapes with arbitrary dimensionality. It also leaves room for higher level libraries to provide convenience factories using the basic building blocks of zfit to implementation e.g. complex amplitudes.

5 Conclusion

zfit is a versatile library that fills the gap of a stable model fitting ecosystem in Python for HEP. Built on top of the deep learning framework TensorFlow, it offers great advantages in terms of speed and reduces maintenance effort while preserving the flexibility of Python, if needed. The formalisation into five loosely coupled parts with a stabilizing and well defined API allows libraries to be built on top and zfit to focus on its core capabilities of model fitting and sampling. Through extensive base classes built for convenient customization and extendability – especially the custom model – the library can extend its scope far beyond the usual feature set of common fitting libraries and adapt to the needs of HEP.

References

[1] Brun, R. and Rademakers, F., Nucl. Instrum. Meth. **A389**, 81-86 (1997)

[2] Verkerke, Wouter and Kirkby, David P., eConf **C0303241**, MOLT007 (2003)

[3] Oliphant, Travis E., Comput Sci Eng **9**, 10-20 (2007)

[4] Millman, K. Jarrod and Aivazis, Michael, Comput Sci Eng **13**, 9-12 (2011)

[5] Rodrigues, Eduardo. "The Scikit-HEP Project." Ed. A. Forti et al. EPJ Web of Conferences 214 (2019)

[6] Matthieu Marinangeli and Brian Pollack and Eduardo Rodrigues and Jonas Eschle, Zenodo, DOI: 10.5281/zenodo.3549838 (2019)