Vectorised calculations with `TensorFlow`

GdR InF hands-on project

Anton Poluektov

Aix Marseille Univ, CNRS/IN2P3, CPPM, Marseille, France

12-16 October 2020

Introduction

Heavy computing tasks in flavour physics analyses (where the analyst often has to write their own code):

- Maximum likelihood fits to large datasets and/or complex models
- Large-scale toy MC generation (e.g. feasibility or systematic studies)
- Fast MC w/o full detector simulation
- Parameter scans in combinations of measurements
- ???

In the cases above, we are not dealing individually with each event.

Uniform operations with bulk dataset, boils down to

- Element-wise operations on vectors of data (*map*)
- Reductions of vectors to a single value: summation, integration, maximum etc. (*reduce*)

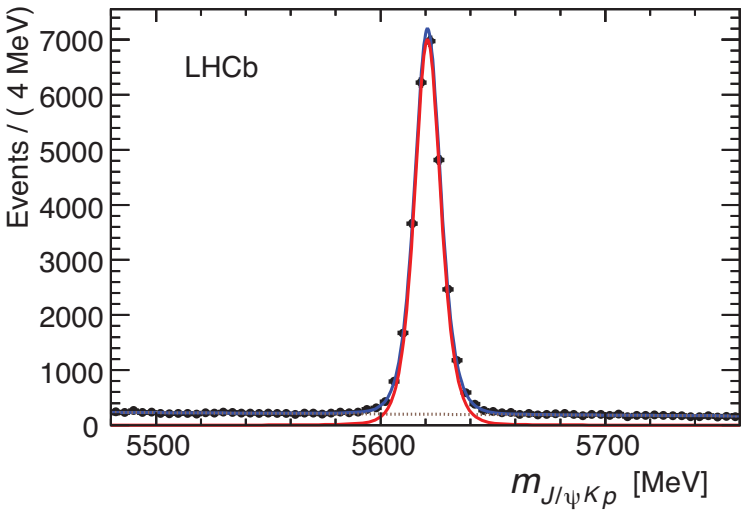
Usually easily vectorisable!

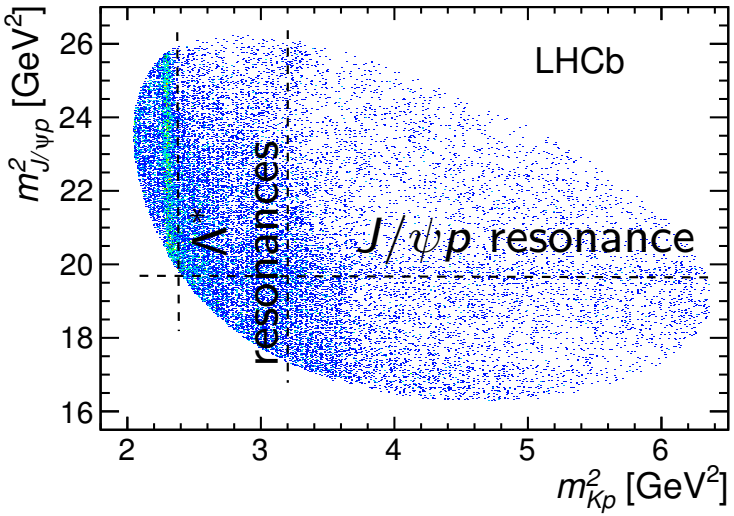
Last but not least: analysis is usually done only once, so quick prototyping of code is essential.

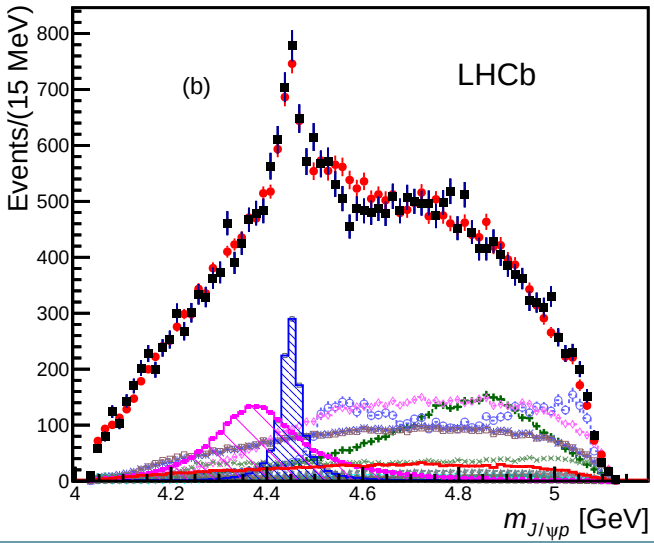
Introduction: amplitude analyses at LHCb

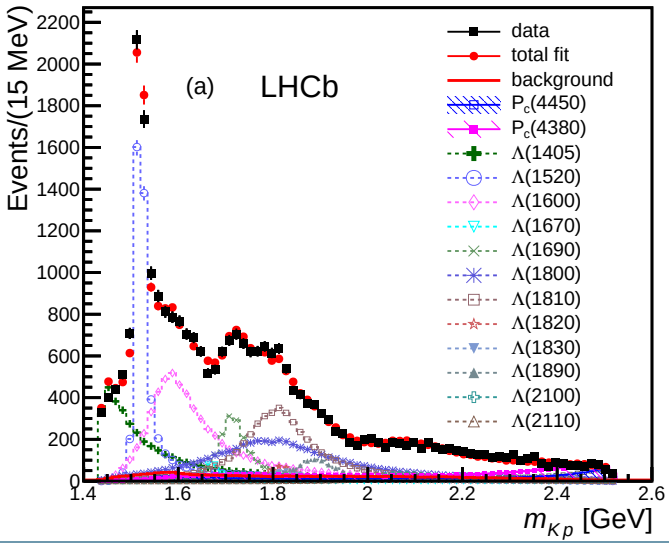
Example: pentaquark discovery: [PRL 115 (2015) 072001]

$\sim$ 26000 events, 6D kinematic phase space, unbinned maximum likelihood fit









Introduction: amplitude analyses at LHCb

Fitting function: a coherent sum of ~ 20 helicity amplitudes

$$|\mathcal{M}|^2 = \sum_{\lambda_{\Lambda_b^0}} \sum_{\lambda_p} \sum_{\Delta\lambda_\mu} \left| \mathcal{M}^{\Lambda^*}_{\lambda_{\Lambda_b^0}, \lambda_p, \Delta\lambda_\mu} + e^{i\,\Delta\lambda_\mu \alpha_\mu} \sum_{\lambda_p^{P_c}} d^{\frac{1}{2}}_{\lambda_p^{P_c}, \lambda_p}(\theta_p)\, \mathcal{M}^{P_c}_{\lambda_{\Lambda_b^0}, \lambda_p^{P_c}, \Delta\lambda_\mu} \right|^2, \quad \leftarrow \text{Decay density}$$

$$\mathcal{M}^{\Lambda^*}_{\lambda_{\Lambda_b^0}, \lambda_p, \Delta\lambda_\mu} \equiv \sum_n \sum_{\lambda_{\Lambda^*}} \sum_{\lambda_\psi} \mathcal{H}^{\Lambda_b^0 \to \Lambda_n^* \psi}_{\lambda_{\Lambda^*}, \lambda_\psi} D^{\frac{1}{2}}_{\lambda_{\Lambda_b^0}, \lambda_{\Lambda^*} - \lambda_\psi}(0, \theta_{\Lambda_b^0}, 0)^* \quad \leftarrow \text{Amplitudes for intermediate resonances}$$

$$\mathcal{H}^{\Lambda_n^* \to Kp}_{\lambda_p, 0} D^{J_{\Lambda_n^*}}_{\lambda_{\Lambda^*}, \lambda_p}(\phi_K, \theta_{\Lambda^*}, 0)^* R_{\Lambda_n^*}(m_{Kp})\, D^1_{\lambda_\psi, \Delta\lambda_\mu}(\phi_\mu, \theta_\psi, 0)^*,$$

$$\mathcal{M}^{P_c}_{\lambda_{\Lambda_b^0}, \lambda_p^{P_c}, \Delta\lambda_\mu^{P_c}} \equiv \sum_j \sum_{\lambda_{P_c}} \sum_{\lambda_\psi^{P_c}} \mathcal{H}^{\Lambda_b^0 \to P_{cj} K}_{\lambda_{P_c}, 0} D^{\frac{1}{2}}_{\lambda_{\Lambda_b^0}, \lambda_{P_c}}(\phi_{P_c}, \theta^{P_c}_{\Lambda_b^0}, 0)^*$$

$$\mathcal{H}^{P_{cj} \to \psi p}_{\lambda_\psi^{P_c}, \lambda_p^{P_c}} D^{J_{P_{cj}}}_{\lambda_{P_c}, \lambda_\psi^{P_c} - \lambda_p^{P_c}}(\phi_\psi, \theta_{P_c}, 0)^* R_{P_{cj}}(m_{\psi p})\; D^1_{\lambda_\psi^{P_c}, \Delta\lambda_\mu^{P_c}}(\phi_\mu^{P_c}, \theta_\psi^{P_c}, 0)^*, \quad (4)$$

$$\mathcal{H}^{A \to BC}_{\lambda_B, \lambda_C} = \sum_L \sum_S \sqrt{\tfrac{2L+1}{2J_A+1}} B_{L,S} \left(\begin{array}{cc|c} J_B & J_C & S \\ \lambda_B & -\lambda_C & \lambda_B - \lambda_C \end{array} \right) \times \left(\begin{array}{cc|c} L & S & J_A \\ 0 & \lambda_B - \lambda_C & \lambda_B - \lambda_C \end{array} \right), \quad \leftarrow \text{Complex couplings}$$

$$R_X(m) = B'_{L_X^{\Lambda_b^0}}(p, p_0, d) \left(\frac{p}{M_{\Lambda_b^0}} \right)^{L_{\Lambda_b^0}^X} \mathrm{BW}(m | M_{0X}, \Gamma_{0X})\, B'_{L_X}(q, q_0, d) \left(\frac{q}{M_{0X}} \right)^{L_X} \cdot \quad \leftarrow \text{Dynamical term}$$

$$\mathrm{BW}(m | M_{0X}, \Gamma_{0X}) = \frac{1}{{M_{0X}}^2 - m^2 - i M_{0X} \Gamma(m)},$$

$$\Gamma(m) = \Gamma_{0X} \left(\frac{q}{q_0} \right)^{2L_X + 1} \frac{M_{0X}}{m} B'_{L_X}(q, q_0, d)^2,$$

Angles entering the expressions are functions of 6D decay kinematics (Lorentz boosts, rotations).

Introduction: amplitude analyses at LHCb

Something else to take into account in addition to the theory model:

- Acceptance and backgrounds.
  - Parametrised multidim. density, $\sim$ easy.
- Resolutions, partially reconstructed states.
  - Integration/convolution: expensive computations
- Unbinned maximum likelihood fit.

$$-\ln\mathcal{L} = -\left(\sum \ln f(x_{\text{data}}) - N_{\text{data}} \ln \sum f(x_{\text{norm}})\right)$$

  Easily vectorised (compute PDF values for each data/normalisation point in parallel).

Typically need hundreds/thousands of fits for a single analysis:

- Model building
- Nominal data fit
- Systematic variations
- Toy MC studies

Amplitude analysis tools in LHCb

Writing an amplitude fitting code from scratch is painful and time consuming.
Several frameworks are in use at LHCb:

- **Laura++**
  - A powerful tool for traditional 2D Dalitz plot analyses (including time-dependent)
  - Single-threaded, but many clever optimisations
- **MINT**
  - Can do 3-body as well as 4-body final states
- **GooFit**
  - GPU-based fitter
- **AmpGen**
  - Amplitude analysis extension for `GooFit` (code generation, JIT).
- **Ipanema-β**
  - GPU-based, `python` interface (`pyCUDA`)
- **qft++**
  - Not a fitter itself, but a tool to operate with covariant tensors

... and a lot of private code in use (e.g. based on RooFit).

Amplitude analysis tools in LHCb

The problem with *frameworks* is that they are not flexible enough.

Trying to do something not foreseen in the framework design becomes a pain.

- Non-scalars in the initial/final states
- Complicated relations between fit parameters
- Fitting projections of the full phase space/partially-rec decays

At some point, it becomes easier to write an own framework (that's why there are so many?)

For the analyses that go beyond a readily available frameworks, need a more flexible solution:

- Efficient from the computational point of view
- Tradeoff between person$\times$hours to implement the code vs. CPU$\times$hours to do the actual fits.

Machine learning tools for HEP calculations?

Amplitude analyses

- Large amounts of data
- Complex models
- ... which depend on optimisable parameters
- Optimise by minimising neg. log. likelihood (NLL)
- Need tools which allow
  - Convenient description of models
  - Efficient computations

  and don't require deep low-level hardware knowledge.

Machine learning

- Large amounts of data
- Complex models
- ... which depend on optimisable parameters
- Optimise by minimising cost function
- Need tools which allow
  - Convenient description of models
  - Efficient computations

  and don't require deep low-level hardware knowledge.

We can reuse the tools developed by a much broader ML community for our needs.

TensorFlow framework

[Tensorflow webpage]
[White Paper]

- "TensorFlow is an open source software library for numerical computation using data flow graphs." Released by Google in October 2015.
- Uses **declarative programming** paradigm: instead of actually running calculations, you describe what you want to calculate (*computational graph*)
- TF can then do various operations with your graph, such as:
  - Optimisation (e.g. caching data, *common subgraph elimination* to avoid calculating same thing many times).
  - Compilation for various architectures (multicore, multithreaded CPU, GPU, distributed clusters, mobile platforms).
  - Analytic derivatives to speed up gradient descent.
- Front-ends for several languages. `Python` is the most natural. Faster development cycle, more compact and readable code.

TensorFlow v1 vs. v2

- What is said below is applied to TensorFlow v2.
- TF v2 is significantly different from v1:
  - The distinction between *declaration* and *execution* is less expressed ("Eager mode")
  - Easier to debug (e.g. can print out intermediate results), but more difficult to figure out what happens under the hood.
- Earlier I have presented the usage of TF v1 in several LHCb meetings. Now moving to v2.
- TensorFlowAnalysis library based on v1 is used in a few analyses.
- New libraries to be used with v2 we will try in this project: AmpliTF and TFA2.

This tutorial

I will demonstrate how one can use `TensorFlow` in a few typical cases of flavour physics analyses.

[Homepage of the project in GitHub]

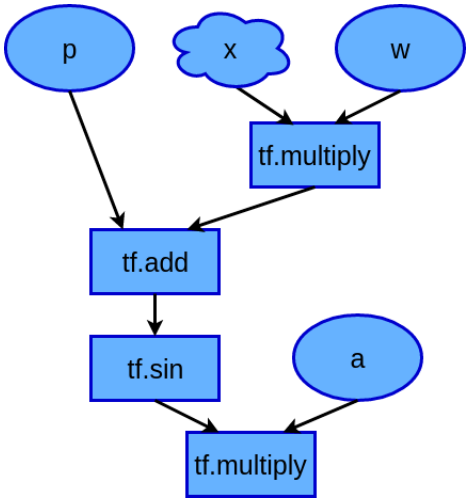
I propose to install `TensorFlow` and all dependencies via `Conda` package manager. [Installation instructions]

After installing (and after this presentation), you can go through the examples one-by-one. Run them yourself and read the explanations on the documentation page.

The examples will gradually introduce the features of `TensorFlow` and HEP extensions (`AmpliTF` and `TFA2`).

Disclaimer: This is the first time I'm giving this tutorial! Expect typos, errors in the code, missing details, bad structure, ... Feedback is very welcome!

TensorFlow: basic structures



TF represents calculations in the form of directional *data flow graph*.

- Nodes: operations
- Edges: data flow

```
f = a*tf.sin(w*x + p)
```

Data are represented by *TF tensors* (rectangular arrays of arbitrary dimensionality)

- Most of TF operations are **vectorised**, e.g. `tf.sin(x)` will calculate element-wise $\sin x_i$ for each element x_i of multidimensional tensor `x`.
- Useful for ML fits, need to calculate same function for each point of large dataset.
- Data can be placed in RAM or VRAM (on GPU)
- Tensors: constant (e.g. training data) or variable (trainable parameters).

TensorFlow: graph building and execution

[hello_world.py in TFA2]

The graph is built from Python function with `@tf.function` decorator:

```
import tensorflow as tf

a = tf.Variable(1., trainable = True)
w = tf.Variable(1., trainable = True)
p = tf.Variable(0., trainable = True)

@tf.function
def f(x):
  return a*tf.sin(w*x + p)
```

(note that calculation graph is described using TF building blocks. Can't use existing libraries directly)

Nothing is executed at this stage. The actual calculation runs once the function `f(x)` is called with TF tensor as an input:

```
x = tf.constant([0., 1., 2., 3., 4.])
print(f(x))  # tf.Tensor([ ... ], shape=(5,), dtype=float32)
```

Tensors can be created fron numpy arrays, and converted back to numpy array with a `t.numpy()` call.

TensorFlow: minimisation algorithms

TensorFlow has its own minimisation algorithms:

```
# Training data
y = tf.constant([1., 2., 3., 4., 5.])

@tf.function
def chi2() :
    return tf.reduce_sum((f(x)-y)**2)

# Create optimiser
from tensorflow.python.training import gradient_descent
opt = gradient_descent.GradientDescentOptimizer(0.001)

# Run 1000 steps of gradient descent
for _ in range(1000) :
    print(a.numpy(), w.numpy(), p.numpy(), chi2().numpy())
    opt.minimize(chi2)
```

- Built-in minimisation functions seem to be OK for ANN training, but not for physics (no uncertainties, likelihood scans, check for global minimum)
- MINUIT seems more suitable. Use it instead, and run TF only for likelihood calculation (custom FCN in python, run Minuit using PyROOT).

Analytic gradient

Extremely useful feature of TF is automatic building of the graph for analytic gradient of any function (speed up convergence!)

In the example above, we can obtain the vector of derivatives for the function `chi2()` over the fit parameters `[a,w,p]` as follows:

```
with tf.GradientTape() as gt:
  grad = gt.gradient(chi2(), [a,w,p])
print(grad)
```

Analytic gradient is calculated internally in the built-in optimizers, but can be called explicitly and passed to `MINUIT`.

Interface with sympy

sympy is a symbolic algebra system for python. Consider it as mathematica with python interface. Free and open-source.

sympy has many extensions for physics calculations
See. e.g. sympy.physics module.

Recent versions of sympy can *generate code* for TensorFlow. Avoid re-implementing functions missing in TF. E.g. create TF tree for Wigner *d* function:

```
def wigner_d(theta, j, m1, m2) :
  """
  Calculate Wigner small-d function. Needs sympy.
    theta : angle
    j : spin
    m1 and m2 : spin projections
  """
  from sympy.abc import x
  from sympy.utilities.lambdify import lambdify
  from sympy.physics.quantum.spin import Rotation as Wigner
  d = Wigner.d(j, m1, m2, x).doit().evalf()
  return lambdify(x, d, "tensorflow")(theta)
```

AmpliTF package

Project in `github`: [AmpliTF].

TF can serve as a framework for complex calculations in flavour physics.
`AmpliTF` implements a library of HEP-related functions.
Trying to be as much *functional* as possible: pure functions, stateless objects.

```
def relativistic_breit_wigner(m2, mres, wres) :
  return 1./complex(mres**2-m2, -mres*wres)

def unbinned_log_likelihood(pdf, data_sample, integ_sample) :
  norm = tf.reduce_sum(pdf(integ_sample))
  return -tf.reduce_sum(tf.log(pdf(data_sample)/norm ))
```

Avoid complicated structure of classes:

- Primitives are standalone and can be reused in e.g. other libraries
- Easier for external developers to contribute

Primitives are glued together in TF itself.

AmpliTF: structure

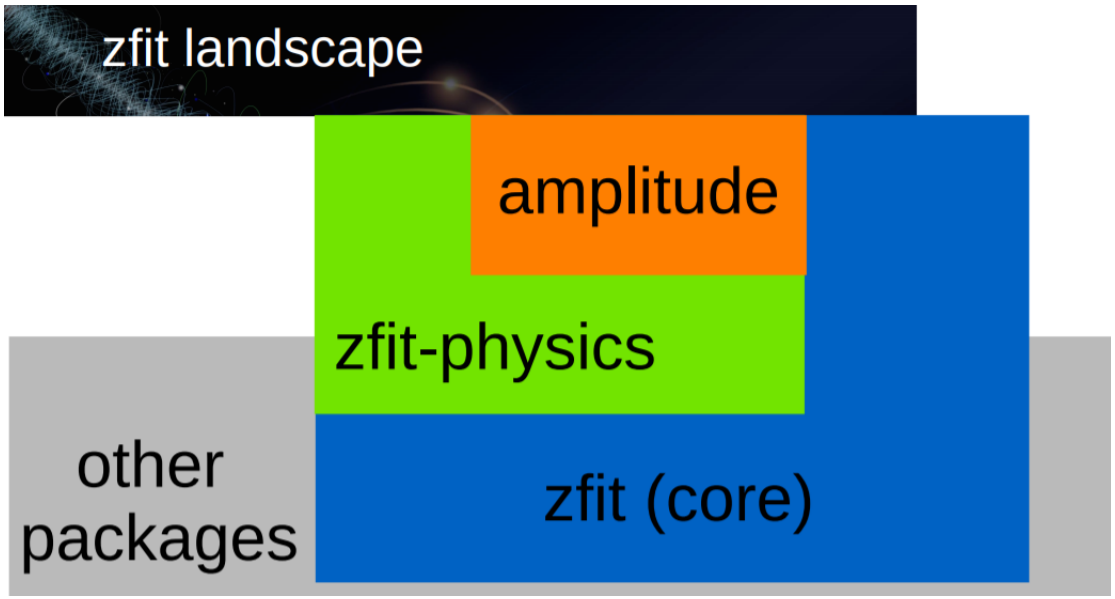
Components of the library:

- Phase space classes (Dalitz plot, four-body, baryonic 3-body, angular etc.): provide functions to check if variable is inside the phase space, to generate uniform distributions etc.
- Collection of functions for amplitude description:
  - Lorentz vectors: boosting, rotation
  - Kinematics: two-body breakup momentum, helicity angles
  - Helicity amplitudes, Zemach tensors
  - Dynamics: Breit-Wigner, Flatte, LASS etc. functions, form factors, non-resonant shapes

zfit

TFA is a low-level library of functions

- LEGO bricks to build your own programs to do fits/toy MC etc.



More high-level package: `zfit` [github].

- The project to use TensorFlow for generic fitting (a-la RooFit).
- Hide TensorFlow technicalities from the user
- Choice of fitters, integration techniques
- No ROOT dependencies (`iminuit` for fitting, `uproot` for tuples)
- Can use AmpliTF functions to create custom PDFs

`ComPWA` is a common amplitude analysis framework develped for PANDA. [paper], [github] (Mainz and Bochum groups).

- High-level framework (hides low-level math from the user, based on isobar description in xml files).
- Originally written in C++/boost, with python interface (`pycompwa`).
- Consider moving to TF. Prototype is called `tensorwaves`: [github]

`tensorwaves` is using `AmpliTF` for kinematic/dynamical functions.

TensorFlow: issues

- TF is heavy (distribution size, loading time)
  - E.g. impacts performance if the large number of quick and simple fits has to be done.
- Memory usage: can easily exceed a few Gb of RAM for large datasets (charm) or complicated models.
  - Especially with analytic gradient
  - Limiting factor with consumer-level GPU.
  - `Tesla V100` works great, but $8000...
- Double precision performance is essential
  - Single precision not sufficient except for simplest models, poor convergence.
  - Again, look for high-end GPU cards for performance.
- Results are not 100% reproducible between different GPUs and CPU
  - Subtle differences in FP implementation?
  - Minimisation can go different ways and even converge to different minima
- Less efficient than dedicated code developed with e.g. CUDA/`Thrust`, but *way* more flexible and easy to hack.

Beyond TensorFlow

The following is my personal vision:

- It's important not to be locked-in with TensorFlow
  - Possible bugs in TF: cross-checks needed.
  - Newer frameworks appearing.
  - Eventual end of support by Google?
- Design the code such that TensorFlow is only one of computational backends. Possible candidates for other backends:
  - pyTorch: probably not the best option (maths not as well developed, no complex maths).
  - Pure numpy (of course, w/o automatic differentiation)
  - JAX: more recent ML tool by Google
    - numpy replacement with GPU/TPU support and autograd.
    - Much lighter than TF: ideal fit for us?

  numpy and TF interfaces are similar enough that this should be easy.

  Approach already used in pyhf (pure Python implementation of HistFactory).

TFA2 package

[GitHub repository for TFA2]

While `AmpliTF` is a package that tries to be as simple as possible and maximally decoupled from `TensorFlow`, `TFA2` is more tightly bound to `TensorFlow`:

- Interface between TF functions and `imimuit` for minimisation
- Routines for toy MC generation with rejection sampling
- Plotting using `matplotlib`, LHCb publication style
- Simple ROOT I/O helper functions using `uproot`
- Multidim. density estimation using ANNs (once we are already in ML ecosystem!)

[Demo scripts] and [their documentation] are also included in `TFA2` repository.

Summary

- `TensorFlow` is gaining popularity as a general-purpose compute engine in HEP
  - Can utilise modern computing architectures (mutithreaded, massively-parallel, distributed) without deep knowledge of their structure.
  - Interesting optimisation options, e.g. analytic derivatives help a lot for fits to converge faster.
  - Transparent structure of code. Only essence of things, no low-level stuff.
  - Fast development cycle with Python backend.
  - Training value for students who will leave HEP for industry.
- As any generic solution, possibly not as optimal as specially designed tool. But taking development time into account, very competitive.