TensorFlow as a compute engine in HEP analyses

Experience with amplitude analyses at LHCb

Anton Poluektov

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

11 May 2020

Introduction

Different patterns of high-performance computing usage in HEP:

- Filtering, histogramming, *etc.* of large datasets.
- Machine learning tasks: jet, cluster reconstruction, particle identification, *etc.*
- Heavy computations on relatively small datasets

↳ Only covering the latter use case.

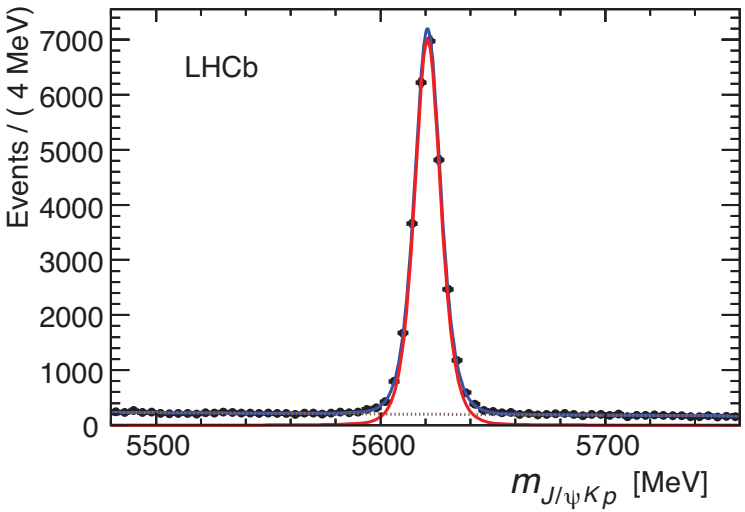
A typical pattern for many analyses in flavour physics.

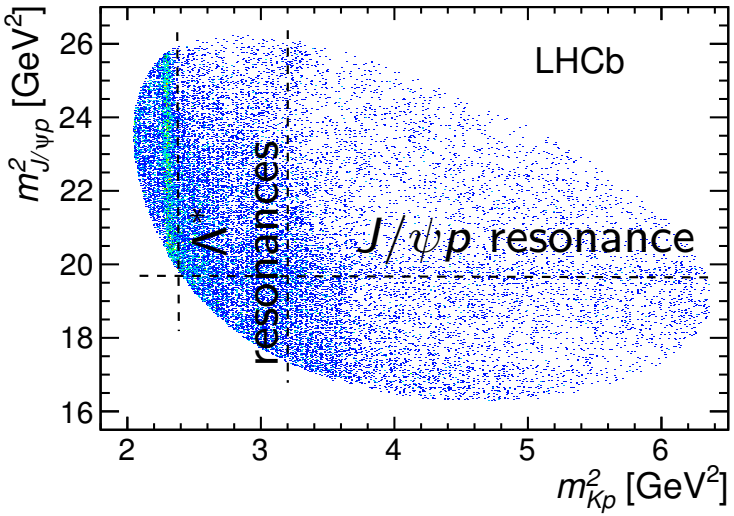
Although *relatively small dataset* has rather broad meaning (10-100 of millions of events for charm datasets at LHCb, expect more after upgrade).

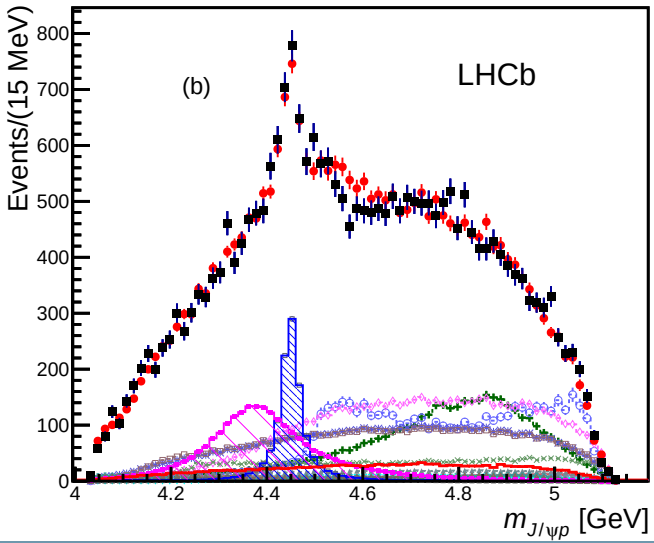
Introduction: amplitude analyses at LHCb

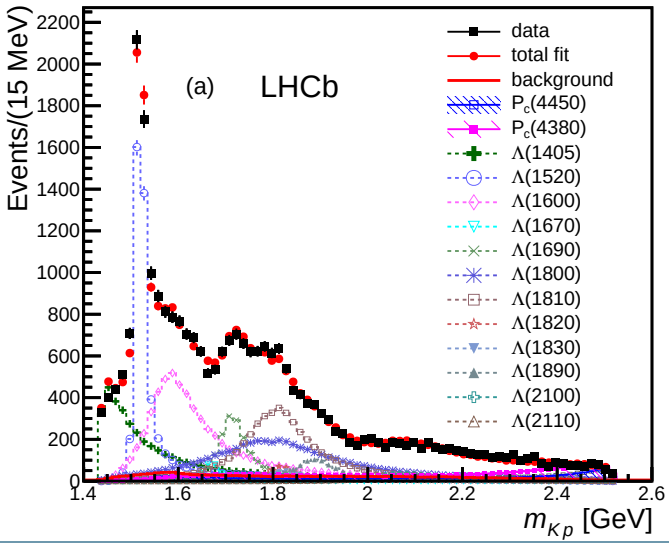
Example: pentaquark discovery: [PRL 115 (2015) 072001]

$\sim$ 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

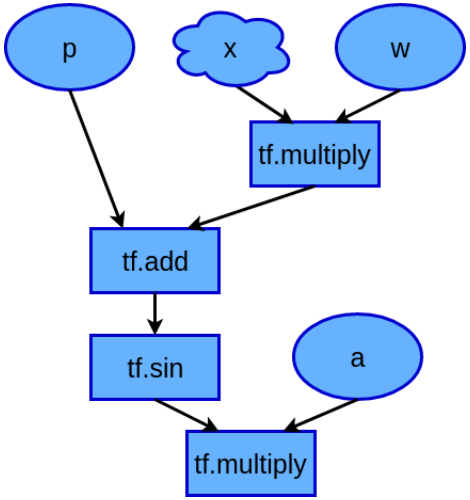
[Tensorflow webpage]
[White Paper]

- "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 mostly applied to `TensorFlow` v1.
- I mostly have experience with TF v1, and the library I'm speaking about is made for this version.
- TF v2 is significantly different:
  - 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.
- More about migration to v2 towards the end of the presentation.

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 *tensors* (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.

Input structures are:

- *Placeholders*: abstract structure which is assigned a value only at execution time. Typically used to feed training data (ML) or data sample to fit to (our case).
- *Variables*: assigned an initial value, can change the value over time. Tunable parameters of the model.

TensorFlow: graph building and execution

To build a graph, you define inputs and TF operations acting on them:

```
import tensorflow as tf

# define input data (x) and model parameters (w,p,a)
x = tf.placeholder( tf.float32, shape = ( None ) )
w = tf.Variable( 1. )
p = tf.Variable( 0. )
a = tf.Variable( 1. )

# Build calculation graph
f = 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 in the TF *session*:

```
# Create TF session and initialise variables
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Run calculation of y by feeding data to tensor x
f_data = sess.run( f, feed_dict = { x : [1., 2., 3., 4.] })

print(y_data)   # [ 0.84147096  0.90929741  0.14112    -0.7568025 ]
```

Input/output in `sess.run` is **numpy arrays**.

TensorFlow: minimisation algorithms

TensorFlow has its own minimisation algorithms:

```
# Placeholder for data
y = tf.placeholder( tf.float32, shape = ( None ) )

# Define chi2 graph using previously defined function f
chi2 = (f-y)**2

# TF optimiser is a graph operation as well
train = tf.train.GradientDescentOptimizer(0.01).minimize( chi2 )

# Run 1000 steps of gradient descent inside TF session
for i in range(1000) :
  sess.run(train, feed_dict = {
            x : [1., 2., 3., 4., 5.],    # Feed data to fit to
            y : [3., 1., 5., 3., 2.] } )
  print(sess.run( [a,w,p] ) )  # Watch how fit parameters evolve
```

- 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 calculation of the graph for analytic gradient of any function (speed up convergence!)

```
tfpars = tf.trainable_variables()  # Get all TF variables
grad = tf.gradients(chi2, tfpars)  # Graph for analytic gradient
```

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

Partial execution

In theory, TF should be able to identify which parts of the graph need to be recalculated (after, e.g. changing value of `tf.Variable`), and which can be taken from cache.

In practice, this does not work between `sess.run` calls, but there is a possibility to *inject* a value of a tensor in `sess.run` using `feed_dict` manually.

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 Wignerd(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)
```

TensorFlowAnalysis package

Project in gitlab: [TensorFlowAnalysis].

TF can serve as a framework for maximum likelihood fits (and amplitude fits in particular). Missing features that need to be added:

- ROOT interface to read/write ntuples (use root-numpy or uproot)
- MINUIT interface for minimisation.
- Library of HEP-related functions.

Trying to be as much *functional* as possible: pure functions, stateless objects.

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

def UnbinnedLogLikelihood(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.

TensorFlowAnalysis: structure

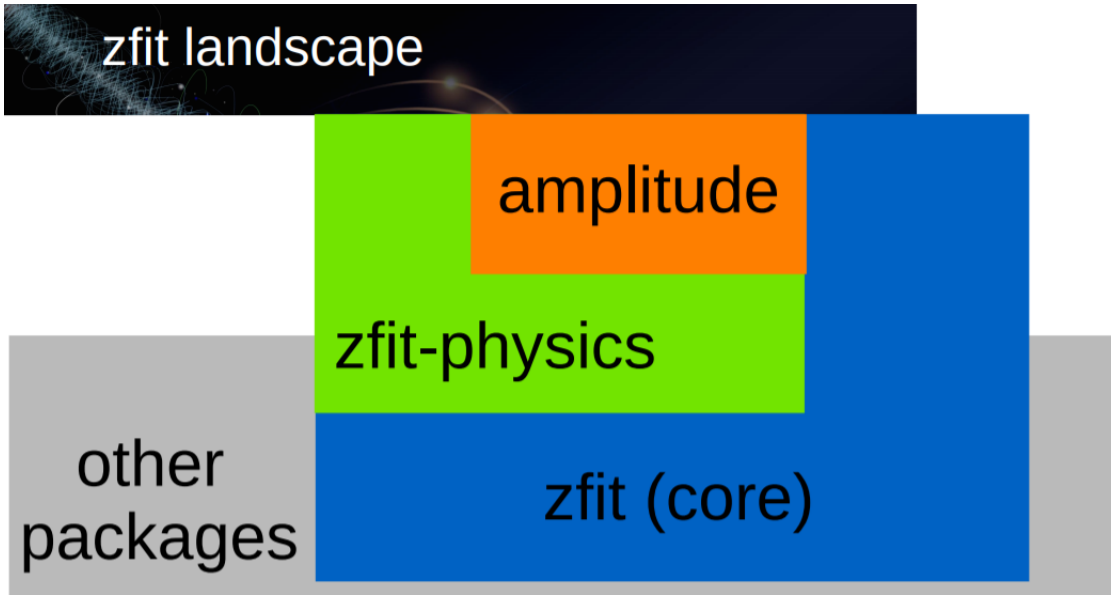
Components of the library:

- 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.
- Fit parameter class: derived from `tf.Variable`, adds range, step size etc. for `MINUIT`
- Interface for `MINUIT`, integration, unbinned log. likelihood
- Functions for toy MC generation, calculation of fit fractions.
- Collection of functions for amplitude description:
  - Lorentz vectors: boosting, rotation
  - Kinematics: two-body breakup momentum, helicity angles
  - Helicity amplitudes, Zemach tensors
  - Dynamics: Breit-Wigner functions, form factors, non-resonant shapes
  - Elements of covariant formalism (polarisation vectors, γ matrices, etc.)
  - Multilinear interpolation of ROOT histograms

zfit

TFA is a low-level library of functions

- LEGO bricks to build your own fitter.



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 still use TFA functions to create custom PDFs

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.

Hardware usage pattern

- **Code development, tests**: any machine (even w/o GPU).
- **Preliminary fits to data, model building**: convenient to have a single high-performance machine with GPU (`Tesla`, multi-GPU server). Allows for a fast try-and-correct cycle (no waiting for batch jobs to start).
- **Large-scale calculations** (toy MC studies, systematic variations); batch cluster (CPU-only or GPU). CERN lxbatch (has GPU machines), `CC-IN2P3 in Lyon for CNRS`.

Beyond TensorFlow

The following is my personal vision:

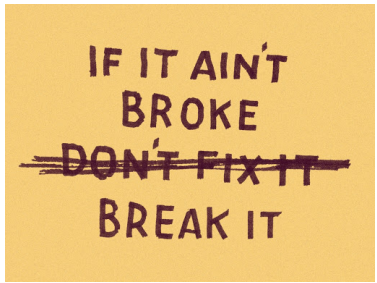
- It's important not to be locked-in with TensorFlow
  - Possible bugs in TF: cross-checks needed.
  - Newer frameworks appearing.
  - Eventual end of support by Google?
- Design the code such that TensorFlow is only one of computational backends. Possible candidates for other backends:
  - pyTorch: 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).

Further developments

- It is clear that TF v1 support should be dropped. Can make some changes at the same time.
- New library that is a successor of TensorFlowAnalysis, but based on TF v2: `AmpliTF`
  - WIP, but used for one early-stage analysis at LHCb
  - Drop ROOT dependency, use `iminuit` for minimisation (or rely on `zfit`)
  - Keep the same philosophy of standalone primitives, functional style
  - Plan to try `jax` instead of TF, make it possible to switch backends?
  - Gradually identify boundaries between existing and planned HEP packages based on TF
    - `zfit`: Choice of different optimisers, general-purpose PDFs, statistical tools, toy MC, etc.
    - `ComPWA/tensorwaves`: higher-level modular amplitude/partial-wave fitting framework with TF backend
    - `AmpliTF`: lower-level primitives (kinematics, dynamics, QFT)



- A `gitter` channel is set up for communication (thanks Stefan Pflueger and Jonas Eschle for initiating this and setting things up).

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.
- Establishing communication between TFA/AmpliTF, zfit and ComPWA/tensorwaves.
  - Other TF-based packages I became aware of: `VegasFlow` (Monte-Carlo techniques), internal BESIII/LHCb library by UCAS (Beijing).
  - Should we organise under the Scikit-HEP umbrella?

Backup

TensorFlowAnalysis: structure of a fitting script

Experimental data are represented in `TensorFlowAnalysis` as a 2D tensor

`data[candidate][variable]`

where inner index corresponds to event/candidate, outer to the phase space variable. E.g. 10000 Dalitz plot points would be represented by a tensor of shape `(10000, 2)`.

In the fitting script, you would start from the definitions of phase space, fit variables and fit model:

```
phsp = DalitzPhaseSpace(ma, mb, mc, md) # Phase space

# Fit parameters
mass = Const(0.770)
width = FitParameter("width", 0.150, 0.1, 0.2, 0.001)
a = Complex( FitParameter("Re(A)", ...), FitParameter("Im(A)", ...) )

def model(x) :          # Fit model as a function of 2D tensor of data
  m2ab = phsp.M2ab(x) # Phase space class provides access to individual
  m2bc = phsp.M2bc(x) #    kinematic variables
  ampl = a*BreitWigner(mass, width, ...)*Zemach(...) + ...
  return Abs(ampl)**2
```

TensorFlowAnalysis: structure of a fitting script

Fit model $f(x)$ enters likelihood via data and normalisation terms:

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

Create two graphs for the model as a function of data and normalisation sample placeholders:

```
model_data = model( phsp.data_placeholder )
model_norm = model( phsp.norm_placeholder )
```

Now can create normalisation sample, and read data e.g.

```
norm_sample = sess.run( phsp.RectangularGridSample(500,500) )
data_sample = ReadNTuple(...)
```

Create the graph for negative log. likehood:

```
norm = Integral( model_norm )
nll = UnbinnedNLL( model_data, norm )
```

And finally call `MINUIT` feeding the actual data and norm samples to placeholders

```
result = RunMinuit(sess, nll, { phsp.data_placeholder : data_sample ,
                                phsp.norm_placeholder : norm_sample } )
```

TensorFlowAnalysis: structure of a fitting script

Call to

```
result = RunMinuit(sess, nll, ... )
```

internally includes calculation of analytic gradient for NLL. See benchmarks below to get the idea how that helps.

Analyst has full control over how likelihood is constructed and what variables serve as free parameters.

Since NLL graph is defined separately, it is easy to construct custom NLLs for e.g. combined CPV-allowed fits of two Dalitz plots.

```
norm = Integral(model1_norm) + Integral(model2_norm)
nll = UnbinnedNLL(model1_data, norm) + UnbinnedNLL(model2_data, norm)
```

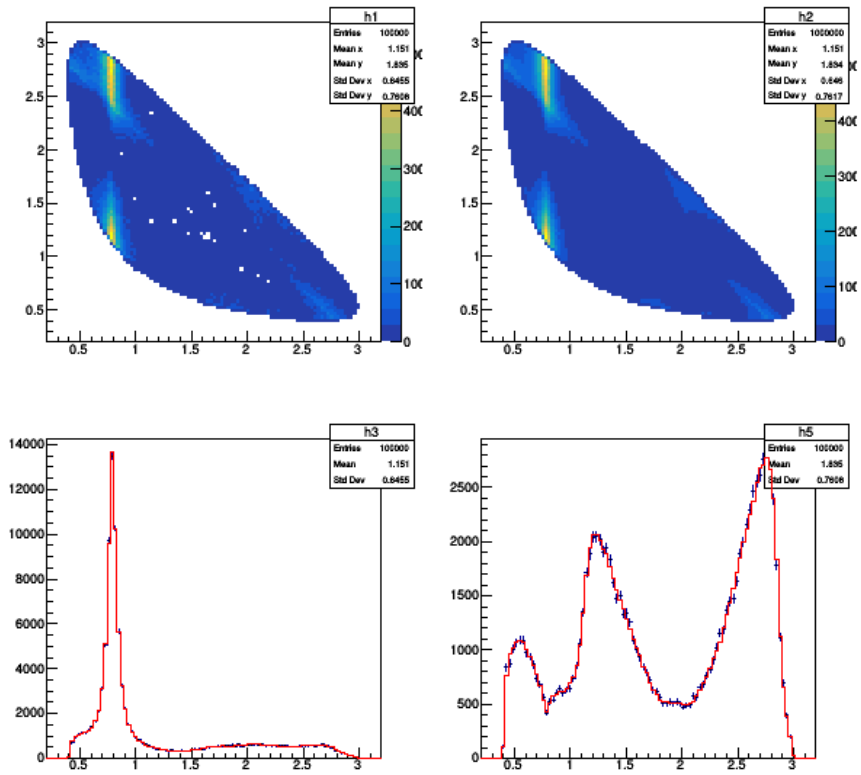
Similarly, complex combinations of fit parameters are easily constructed, e.g. CP-violating amplitudes

$$a_{\pm} = (\rho_{CPC} \pm \rho_{CPV}) e^{i(\delta_{CPC} \pm \delta_{CPV})}$$

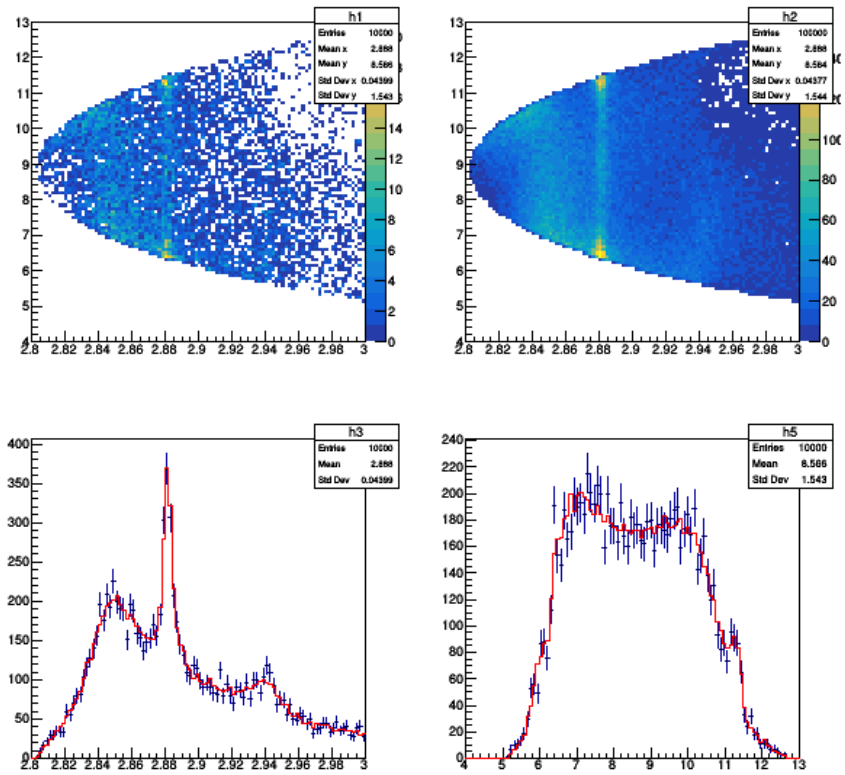
Example: [$\Xi_b^- \to pK^-K^-$ CPV-enabled toy MC]

TensorFlowAnalysis: examples

Isobar models implemented with helicity formalism and "simple" line shapes (Breit-Wigner, Gounaris-Sakurai, Flatté, LASS, Dabba, etc.).
Examples in [TensorFlowAnalysis/work]



Traditional Dalitz plot
$D^0 \to K^0_S \pi^+ \pi^-$



Baryonic
$\Lambda^0_b \to D^0 p \pi^-$

CPU profiling

Profiling feature allows to identify bottlenecks in execution speed.

Breakup of operations by CPU core (32-core Xeon).

TensorFlowAnalysis: benchmarks

Benchmark runs (fit time only), compare 2 machines.
CPU1: Intel Core i5-3570 (4 cores @ 3.4GHz, 16Gb RAM)
GPU1: NVidia GeForce 750Ti (640 CUDA cores @ 1020MHz, 2Gb VRAM, 88Gb/s, 40 Gflops DP)
CPU2: Intel Xeon E5-2620 (32 cores @ 2.1GHz, 64Gb RAM)
GPU2: NVidia Quadro p5000 (2560 cores @ 1600MHz, 16Gb VRAM, 320Gb/s BW, 280 Gflops DP)
GPU3: NVidia K20X (2688 cores @ 732MHz, 6Gb VRAM, 250Gb/s BW, 1300 Gflops DP)

	Iterations	Time, sec				
		CPU1	GPU1	CPU2	GPU2	GPU3
$D^0 \to K^0_S\pi^+\pi^-$, 100k events, 500×500 norm.						
Numerical grad.	2731	488	250	113	59	82
Analytic grad.	297	68	36	18	12	19
$D^0 \to K^0_S\pi^+\pi^-$, 1M events, 1000×1000 norm.						
Numerical grad.	2571	3393	1351	937	306	378
Analytic grad.	1149	1587	633	440	148	180
$\Lambda^0_b \to D^0 p\pi^-$, 10k events, 400×400 norm.						
Numerical grad.	9283	434	280	162	157	278
Analytic grad.	425	33	23	18	21	32
$\Lambda^0_b \to D^0 p\pi^-$, 100k events, 800×800 norm.						
Numerical grad.	6179	910	632	435	266	364
Analytic grad.	390	133	62	126	32	45

$D^0 \to K^0_S\pi^+\pi^-$ amplitude: isobar model, 18 resonances, 36 free parameters
$\Lambda^0_b \to D^0 p\pi^-$ amplitude: 3 resonances, 4 nonres amplitudes, 28 free parameters