Performing amplitude fits with `TensorFlow`: LHCb experience

Anton Poluektov

University of Warwick, UK

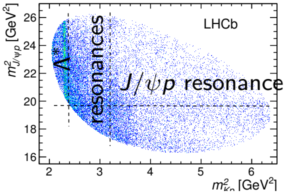
23 May 2017

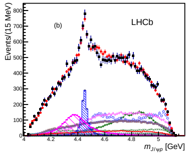
On behalf of LHCb collaboration

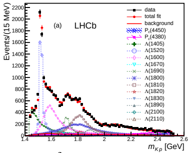
Introduction: amplitude analyses at LHCb

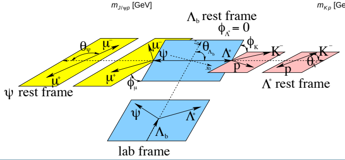
Pentaquark analysis:
[PRL 115 (2015) 072001]

$\sim$ 26000 events, 6D phase space, unbinned ML fit









- Many analyses at LHCb are using amplitude fits:
  - Very powerful analysis technique.
  - Complex unbinned fits with many free parameters over a multidimensional phase space (typically 2–8 dims.)
  - From thousands (rare B decays) to many millions (charm decays) events to fit.
- Several existing frameworks (Laura++, MINT, GooFit) typically limited to a subset of possible analyses (e.g. 2D Dalitz plots).
- Looking at more flexible alternatives to perform amplitude fits efficiently.
- Today: experience with `TensorFlow` library.

Introduction

- Warning for those familiar with `TensorFlow`: this is **not** a talk about machine learning. This is a talk about using `TensorFlow` for maximum likelihood fits (in particular, amplitude fits).

Amplitude analyses

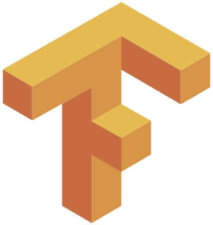
- Large amounts of data
- Complex models
- ... which depend on optimisable parameters
- Optimise by minimising neg. log. likelihood (NLL)
- Need tools which allow
    - Convenient description of models
    - Efficient computations

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

Many software tools are developed for machine learning, could reuse some of them in HEP analyses.

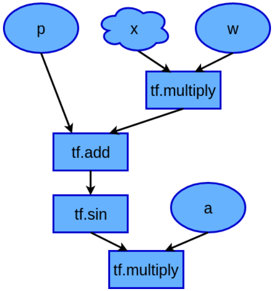
`TensorFlow` library



[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 **computer algebra** 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, mobile CPU, GPU, distributed clusters).
    - Analytic derivatives to speed up gradient descent.
- Has `Python`, `C++` and `Java` front-ends. `Python` is more developed and (IMO) more convenient. Faster development cycle, more compact and readable code.

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`.

Input data can take the form of

- *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: features for amplitude analyses

Built-in optimisers

Built-in minimisation functions OK for ANN training, but not for physics (no uncertainties, likelihood scans). Use `MINUIT` 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 always work as expected, but there is a possibility to *inject* a value of a tensor in `sess.run` using `feed_dict` manually.

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.
- MINUIT interface for minimisation.
- Library of HEP-related functions.

Simplified standalone Dalitz plot generation/fitting script using only TF and ROOT. [DemoDalitzFit.py]

Only around 200 lines of Python, thanks to very compact code, e.g.:

```
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 ))
```

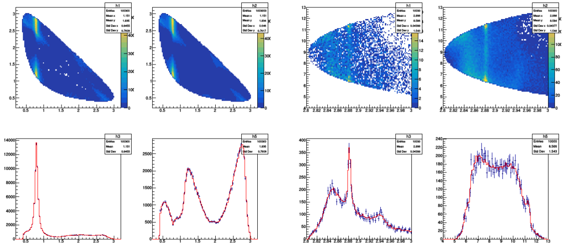
`TensorFlowAnalysis`: structure

Unlike many other amplitude analysis frameworks, `TensorFlowAnalysis` is basically a **collection of standalone functions** for components of the amplitude. These are then glued together in TF itself.
Components of the library are:

- 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

TensorFlowAnalysis: status and plans

Code is functional for conventional 2D Dalitz plots and baryonic 3-body decays like $\Lambda_b^0 \to D^0 p\pi^-$. Examples in [TensorFlowAnalysis/work]



Possible directions of development

- Extending library of function: as needed by the analyses.
- Saving/loading of compiled graphs.
- Optimisations of CPU/memory usage, more intelligent caching.
- Self-documenting feature. Could use Python magic to automatically generate LaTeX description of formulas entering the fit (by replacing the input tensors with special Python objects).

TensorFlowAnalysis: benchmarks

Benchmark runs (fit time only).

CPU: Intel Core i5-3570 (4 cores), 3.4GHz, 16Gb RAM

GPU: NVidia GeForce 750Ti (640 CUDA cores), 2Gb VRAM

	Iterations	Time, sec			
		No caching		Forced caching	
		CPU	GPU	CPU	GPU
$D^0 \to K_S^0\pi^+\pi^-$, 100k events, 500×500 norm.					
Numerical grad.	2656	484	247	486	272
Analytic grad.	297	69	67	68	70
$D^0 \to K_S^0\pi^+\pi^-$, 1M events, 1000×1000 norm.					
Numerical grad.	2271	3196	1247	3012	1374
Analytic grad.	1146	1678	829	1585	864
$\Lambda_b^0 \to D^0 p\pi^-$, 10k events, 400×400 norm.					
Numerical grad.	7258	1046	302	418	250
Analytic grad.	397	66	67	39	66
$\Lambda_b^0 \to D^0 p\pi^-$, 100k events, 800×800 norm.					
Numerical grad.	6116	3503	400	1802	382
Analytic grad.	280	211	201	116	196

$D^0 \to K_S^0\pi^+\pi^-$ amplitude: isobar model, 18 resonances, 36 free parameters

$\Lambda_b^0 \to D^0 p\pi^-$ amplitude: 3 resonances, 4 nonres amplitudes, 28 free parameters

Forced caching: enforce caching of helicity tensors.

`TensorFlowAnalysis`: problems and limitations

- `TensorFlow` is not readily available at CERN `lxplus`.
  - Installing from binaries on Debian-based systems and Mac is straightforward.
  - With SLC, need to install from source. Tricky, but doable.
- 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.
- Double precision is essential
  - Performance issues with consumer-level GPUs
- In some cases, models run faster on CPU than on GPU
  - Some TF computational kernels are not yet implemented on GPU
  - RAM–VRAM data transfer issues?
- Probably less efficient than dedicated code developed with CUDA/Thrust, but *way* more flexible and easy to hack.

Summary

- Google made a good job providing us a functional framework for doing complicated fits: `TensorFlow`
- Why I think this approach is promising:
  - 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 auxiliary low-level technical stuff in the description of functions.
  - Resulting models very portable and (with minor effort) can work standalone w/o the framework. Should be easy to e.g. share with theorists.
  - Flexible python interface can allow further tricks, e.g. automatic generation of LaTeX documentation or custom code generation.
  - Useful 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 cycle into account, very competitive.
- `TensorFlowAnalysis` package: collection of functions to perform amplitude analysis fits. In active development, used for a few ongoing baryonic decay analyses at LHCb.

BACKUP

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)
- MINUIT seems more suitable. Use it instead, and run TF only for likelihood calculation (custom FCN in python, run Minuit using PyROOT).

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

Subtlety: fit model $f(x)$ enters differently into data and normalisation terms in the likelihood:

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

Thus need to 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.

Since NLL graph is defined separately, it should be 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)
```

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

Examples in `TensorFlowAnalysis/work`

List of example fitting/toy MC scripts in the `master` branch of `TensorFlowAnalysis`

`AngularFit.py`	Fit in 3D angular phase space a la $B^0 \to K^* \mu^+ \mu^-$
`D2KsPiPi.py`	Realistic amplitude for $D^0 \to K^0_S \pi^+ \pi^-$ with 18 resonances, incl. background
`DalitzTF.py`	Simplified amplitude for $D^0 \to K^0_S \pi^+ \pi^-$
`FourBodyToys.py`	Toy MC generation for 4-body $\Lambda^0_b \to p\pi^-\pi^-\pi^+$
`HistInterpolation.py`	Example of using interpolated 2D shape from ROOT histogram (e.g. for efficiency or background)
`Lb2Dppi.py`	$\Lambda^0_b \to D^0 p \pi^-$ amplitude fit in helicity formalism
`Lb2DppiCovariantFit.py`	$\Lambda^0_b \to D^0 p \pi^-$ amplitude fit in convariant formalism
`Lb2DppiCovariantToys.py`	Toy MC generation of resonances in $\Lambda^0_b \to D^0 p \pi^-$ using convariant formalism
`Lc2pKpi.py`	Realistic $\Lambda^+_c \to pK^-\pi^+$ amplitude using helicity formalism. Includes non-uniform efficiency
`Xib2pKK_CP.py`	CPV-allowed combined fit of two Dalitz plots of $\Xi^-_b \to pK^-K^-$