Performing amplitude fits with TensorFlow: LHCb experience

Anton Poluektov
University of Warwick, UK
23 May 2017

On behalf of LHCb collaboration
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.
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 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

\[ f = a \times \text{tf.sin}(w \times x + p) \]

Data are represented by tensors (arrays of arbitrary dimensionality)

- Most of TF operations are vectorised, e.g. \( \text{tf.sin}(x) \) 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.
To build a graph, you define inputs and TF operations acting on them:

```python
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:

```python
# 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**.
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!)

```python
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.
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.:

```python
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 ))
```
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, $\gamma$ 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 $\Lambda^0_b \rightarrow D^0 p\pi^-$. 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).
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

<table>
<thead>
<tr>
<th></th>
<th>Iterations</th>
<th>Time, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No caching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td>(D^0 \rightarrow K_S^0 \pi^+ \pi^-), 100k events, 500 x 500 norm.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical grad.</td>
<td>2656</td>
<td>484</td>
</tr>
<tr>
<td>Analytic grad.</td>
<td>297</td>
<td>69</td>
</tr>
<tr>
<td>(D^0 \rightarrow K_S^0 \pi^+ \pi^-), 1M events, 1000 x 1000 norm.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical grad.</td>
<td>2271</td>
<td>3196</td>
</tr>
<tr>
<td>Analytic grad.</td>
<td>1146</td>
<td>1678</td>
</tr>
<tr>
<td>(\Lambda_b^0 \rightarrow D^0 p\pi^-), 10k events, 400 x 400 norm.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical grad.</td>
<td>7258</td>
<td>1046</td>
</tr>
<tr>
<td>Analytic grad.</td>
<td>397</td>
<td>66</td>
</tr>
<tr>
<td>(\Lambda_b^0 \rightarrow D^0 p\pi^-), 100k events, 800 x 800 norm.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical grad.</td>
<td>6116</td>
<td>3503</td>
</tr>
<tr>
<td>Analytic grad.</td>
<td>280</td>
<td>211</td>
</tr>
</tbody>
</table>

\(D^0 \rightarrow K_S^0 \pi^+ \pi^-\) amplitude: isobar model, 18 resonances, 36 free parameters
\(\Lambda_b^0 \rightarrow D^0 p\pi^-\) amplitude: 3 resonances, 4 nonres amplitudes, 28 free parameters
Forced caching: enforce caching of helicity tensors.
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.
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.
TensorFlow: minimisation algorithms

TensorFlow has its own minimisation algorithms:

```python
# 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).
Experimental data are represented in TensorFlowAnalysis as a 2D tensor
\[ \text{data}[\text{candidate}][\text{variable}] \]
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:

```python
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
```

Anton Poluektov Experience with TensorFlow Analysis Ecosystem Workshop, Amsterdam, 22-24 May 2017
Subtlety: fit model \( f(x) \) enters differently into data and normalisation terms in the likelihood:

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

Thus need to create two graphs for the model as a function of data and normalisation sample placeholders:

\[
\text{model\_data} = \text{model}( \text{phsp\_data\_placeholder} ) \\
\text{model\_norm} = \text{model}( \text{phsp\_norm\_placeholder} )
\]

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

\[
\text{norm\_sample} = \text{sess\_run}( \text{phsp\_RectangularGridSample}(500,500) ) \\
\text{data\_sample} = \text{ReadNTuple}(...)
\]

Create the graph for negative log. likelihood:

\[
\text{norm} = \text{Integral}( \text{model\_norm} ) \\
\text{nll} = \text{UnbinnedNLL}( \text{model\_data}, \text{norm} )
\]

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

\[
\text{result} = \text{RunMinuit}(\text{sess}, \text{nll}, \{ \text{phsp\_data\_placeholder : data\_sample }, \text{phsp\_norm\_placeholder : norm\_sample } \})
\]
Call to

\[
\text{result} = \text{RunMinuit}(\text{sess, nll, ... })
\]

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-enabled fits of two Dalitz plots.

\[
\text{norm} = \text{Integral(model1\_norm)} + \text{Integral(model2\_norm)}
\]
\[
\text{nll} = \text{UnbinnedNLL(model1\_data, norm)} + \text{UnbinnedNLL(model2\_data, norm)}
\]

Example: \([\Xi_b^- \rightarrow pK^-K^- \text{ 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 $B^0 \rightarrow K^* \mu^+ \mu^-$

**D2KsPiPi.py**  Realistic amplitude for $D^0 \rightarrow K_S^0 \pi^+ \pi^-$ with 18 resonances, incl. background

**DalitzTF.py**  Simplified amplitude for $D^0 \rightarrow K_S^0 \pi^+ \pi^-$

**FourBodyToys.py**  Toy MC generation for 4-body $\Lambda_b^0 \rightarrow p\pi^-\pi^-\pi^+$

**HistInterpolation.py**  Example of using interpolated 2D shape from ROOT histogram (e.g. for efficiency or background)

**Lb2Dppi.py**  $\Lambda_b^0 \rightarrow D^0 p\pi^-$ amplitude fit in helicity formalism

**Lb2DppiCovariantFit.py**  $\Lambda_b^0 \rightarrow D^0 p\pi^-$ amplitude fit in convariant formalism

**Lb2DppiCovariantToys.py**  Toy MC generation of resonances in $\Lambda_b^0 \rightarrow D^0 p\pi^-$ using convariant formalism

**Lc2pKpi.py**  Realistic $\Lambda_c^+ \rightarrow pK^-\pi^+$ amplitude using helicity formalism. Includes non-uniform efficiency

**Xib2pKK_CP.py**  CPV-allowed combined fit of two Dalitz plots of $\Xi_b^- \rightarrow pK^-K^-$