Using TensorFlow for amplitude fits

Adam Morris, Anton Poluektov, Andrea Mauri, Andrea Merli, Abhijit Mathad, Maurizio Martinelli

Using TensorFlow for amplitude fits

The TensorFlowAnalysis package

Adam Morris[1], on behalf of the TensorFlowAnalysis developers

Anton Poluektov[2], Andrea Mauri[3], Andrea Merli[4], Abhijit Mathad[2], Maurizio Martinelli[5], Adam Morris[1]

[1]Aix Marseille Univ, CNRS/IN2P3, CPPM
[2]University of Warwick
[3]Universität Zürich
[4]Università degli Studi e INFN Milano
[5]European Organization for Nuclear Research

PyHEP Workshop, Sofia, 7–8 July, 2018

Overview

1. Amplitude analysis
2. TensorFlow
3. TensorFlowAnalysis
4. Performance

Amplitude analysis

Amplitude analysis: introduction

Amplitude fits:

- Extract information about intermediate states in multi-body decays
- PDFs can be computationally expensive to evaluate
  - Complex models (in both meanings of ‘complex’)
  - Many free parameters
  - Multi-dimensional phase space
  - Often numerically integrated
- Writing fitters can be labour-intensive without the right framework

Used in:

- Hadron spectroscopy
  - Discovery of pentaquarks
- Measurement of CKM parameters
  - *CP* violation
  - γ angle

Amplitude analysis: tools

Amplitude fitting:

- Laura++: https://laura.hepforge.org/
  - C++ with ROOT as its only dependency
  - Powerful tool for Dalitz plot fits
  - Can do time-dependent fits
  - Single-threaded, but many clever optimisations
- MINT: https://twiki.cern.ch/twiki/bin/view/Main/MintTutorial
  - C++ interface
  - Can do 3- and 4-body final states
  - Can be used as a generator in the LHCb simulation package Gauss

Generic GPU-based fitting:

- GooFit: https://github.com/GooFit
  - C++ with python bindings
  - Has a third-party library for amplitude fits
- Ipanema-β: https://gitlab.cern.ch/bsm-fleet/Ipanema
  - Based on pyCUDA
  - HEP-specific functions
  - Lacks amplitude analysis functions

Tool for covariant tensors:

- qft++: https://github.com/jdalseno/qft

Amplitude analysis: tools

Existing frameworks lack functionality and/or flexibility to cover all cases that might be encountered in amplitude anlaysis. Users may spend a lot of time altering the framework itself to suit their needs, *e.g.*:

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

For *n*-body final states with complicated models, we need:

- Speed (of computation)
- Speed (of development)
- Flexibility

Amplitude analysis: similarities with machine learning

Maximum-likelihood fitting (particularly amplitude analysis) is very similar to machine-learning:

- Large amounts of data — many evaluations of the same function
- Complicated models
- Optimisable parameters
- Minimisation (cost function/NLL)
- Both abbreviate to ‘ML’

Many of the challenges faced in amplitude anlaysis have been overcome for machine learning

TensorFlow

TensorFlow: introduction

Open source library developed by Google: https://www.tensorflow.org/

- Primarily a machine learning library, but the core functionality is suitable for other tasks
  - Symbolic mathematics
- High-performance numerical computation using dataflow graphs
  - Calling functions builds a directed graph, which can then be optimised and compiled
- TF can find analytic derivatives of a graph
- Python, C++ and Java interfaces
- Runs on many architectures out-of-the-box, including GPUs

TensorFlow: principles

Functions: symbolic dataflow graphs

- Each node is an operation
- Edges represent the flow of data

Data: tensors (*n*-dimensional arrays)

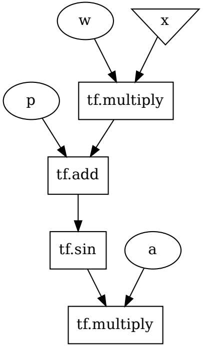
- Input and output of mathematical operations
- Operations are vectorised

Input:

- Placeholders: used to represent data when building dataflow graphs.
- Variables: can change value during a session, *e.g.* fit parameters.

Output:

- Numpy arrays



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

Evaluation:

- Construct a ‘session’
- Run the session by passing a graph and a dict relating placeholders to data samples

TensorFlow: build and execute a graph

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

# 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 f_data # [ 0.84147096, 0.90929741, 0.14112, -0.7568025 ]
```

TensorFlow: features for amplitude analysis

Vectorisation:

- Most functions will calculate element-wise over a tensor
- Ideal for maximum-likelihood fits, where the same function must be evaluated repeatedly for a large number of points

Analytic gradient:

- TF can derive analytic gradients from graphs
- Greatly speed up convergence when passed to a minimiser

Partial execution:

- TF can cache parts of a graph unaffected by changes in parameters
- In practice, this does not work as expected, but one can manually inject the value of a tensor when running a session

Minimisation:

- TF has minimisers for training machine-learning algorithms...
- ... which not particularly suitable for fitting
  - No uncertainties on parameters
  - Cannot do likelihood scans

TensorFlowAnalysis

TensorFlowAnalysis: introduction

TensorFlow alone is almost a suitable framework for amplitude fits. TensorFlowAnalysis (`https://gitlab.cern.ch/poluekt/TensorFlowAnalysis/`) adds some crucial features:

- Read/write ROOT ntuples
- Fit parameter class (extends `tf.Variable`)
- Interface to Minuit
- Toy generation
- Fit fractions
- Functions commonly for calculating amplitudes
  - Kinematics: lorentz vectors, boosts, rotations, two-body momenta, helicity angles...
  - Dynamics: lineshapes, form factors...
  - Helicity amplitudes, *LS* couplings, Zemach tensors...
  - Elements of covariant formalism (polarisation vectors, γ matrices...)
- Phase space classes
  - Check if a datapoint is within the phase space
  - Generate uniform distributions
  - Return/calculate specific variables from a datapoint

TensorFlowAnalysis: fitting

- Simple 2D Dalitz plot model:

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

TensorFlowAnalysis: fitting

- Using MC integration for the normalisation:

```
# Call the model on placeholders to build the dataflow graphs
model_data = model(phsp.data_placeholder)
model_norm = model(norm.data_placeholder)
# Assemble into a negative log-likelihood graph to be minmised
nll = UnbinnedNLL(model_data, Integral(model_norm))
```

- Input data samples are numpy arrays:

```
# Both samples of the form data[event][variable]
data_sample = ReadNTuple(tree, [branches...])
norm_sample = sess.run(phsp.RectangularGridSample(400, 400))
```

- Minimise the NLL with Minuit:

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

TensorFlowAnalysis: fitting

Straightforward to modify the NLL to add functionality, *e.g.*:

- Weighted fit:

```
# Assume the weight is the last element in the list
def event_weight(datapoint, norm = 1.):
  return tf.transpose(datapoint)[-1] * norm
integral = WeightedIntegral(model_norm, event_weight(norm_ph))
weight_correction = sum([dp[-1] for dp in data_sample])
                   /sum([dp[-1]**2 for dp in data_sample])
nll = UnbinnedWeightedNLL(model_data, integral,
                          event_weight(data_ph, norm = weight_correction))
```

- Simultaneous fit:

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

TensorFlowAnalysis: fitting

- Complex combinations of parameters:

```
def HelicityCouplingsFromLS(ja, jb, jc, lb, lc, bls):
  a = 0.
  for ls, b in bls.iteritems():
    # Where b is a Complex(FitParameter(...), FitParameter(...))
    l = ls[0]
    s = ls[1]
    coeff = math.sqrt((l+1)/(ja+1))*Clebsch(jb, lb, jc, -lc, s, lb-lc)
                                   *Clebsch(l, 0, s, lb-lc, ja, lb-lc)
    a += Const(coeff)*b
  return a
```

TensorFlowAnalysis: higher-level features

Some recent features allow the user to quickly build an amplitude model of an *n*-body decay.

The `Particle` class:

- Holds intrinsic properties and mother/daughter relationships
- Useful to quickly define different decay chains within an amplitude model
- Handles rotations and boosts

`HelicityMatrixDecayChain`:

- Takes the head `Particle` of the decay chain and a dict of helicity amplitude parameters
- Builds a dict of matrix elements in the helicity formalism for a specific decay chain

`PHSPGenerator` and `NBody`:

- Construct a phase space object given the mother mass and a list of final-state daughter masses

TensorFlowAnalysis: limitations

Some issues with using TensorFlow for amplitude fits:

- Python 2 only (for now)
- TF not readily available on LXplus
  - Binary distributions available from debian-based distros and Mac
  - Available from pip without machine-specific optimisations
  - Can install from source: tricky (especially with CUDA) but possible.
- Memory usage can be several GB:
  - Especially with anlaytic gradient/large datasets/complicated models
  - Limiting for consumer-grade GPUs
- Double precision essential
  - Limiting for consumer-grade GPUs
- Slow RAM–VRAM transfer
  - Has been mitigated since earlier versions of TFA
- Errors at graph execution time are hard to debug
  - Dedicated debugger: `https://www.tensorflow.org/programmers_guide/debugger`

TensorFlowAnalysis: plans

- Port to python 3
- Expand the library
  - K-matrix formalism
  - Analytical coupled-channel approaches
- Save/load compiled graphs
  - Graph-building can sometimes take longer than minimisation
- Optimisations of CPU and memory usage; better caching
- More symbolic maths
  - Sympy, in particular, works well with TF
- Self-documentation
  - Generate LaTeX description of formulae entering the fit
- Automatic code generation: share standalone models with theorists

Performance

Performance

Benchmark runs (fit time only), compare 2 machines:

- CPU1: Intel Core i5-3570 (4 cores), 3.4GHz, 16 Gb RAM
  GPU1: NVidia GeForce 750Ti (640 CUDA cores), 2 Gb VRAM
- CPU2: Intel Xeon E5-2620 (32 cores), 2.1GHz, 64 Gb RAM
  GPU2: NVidia Quadro p5000 (2560 CUDA cores), 16 Gb VRAM

Two isobar models:

- $D^0 \to K^0_S \pi^+ \pi^-$: 18 resonances, 36 free parameters
- $\Lambda_b \to D^0 p \pi^-$: 3 resonances, 4 non-resonant amplitudes, 28 free parameters

Performance

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

Summary

- TensorFlow is a good basis for an amplitude fitting framework
- High-performance architectures can be exploited without expert knowledge
- Models written in TFA are portable and can, with small effort, work standalone from TF: easy to share with theorists
- Flexibility of TFA allows for rapid and simple development of complicated fits
- TensorFlowAnalysis package: library to perform amplitude analysis fits. In active development, used for a few ongoing baryonic decay analyses at LHCb.