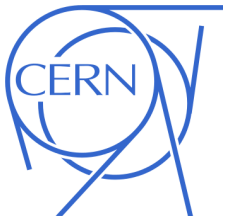
Maximum likelihood fits with TensorFlow

Josh Bendavid, M. Dunser (CERN)
E. Di Marco, M. Cipriani (INFN/Roma)



July 31, 2018

Introduction

- Likelihood for for W helicity/rapidity is quite complex
- With preliminary datacards, muon channel only:
  - $\sim$ 180 M events expected
  - 1444 bins
  - 96 parameters of interest (2 charges x 3 polarization states x 16 $|y|$ bins)
  - 70 nuisance parameters
- Fit was previously attempted with Higgs combination tool:
  - Likelihood constructed/computed with RooFit
  - Minimization with Minuit2
  - Gradient for minimization evaluated numerically with some variation of finite difference method (implemented internally in Minuit)
- Large numbers of events can introduce numerical precision/stability issues
- Major issues with fit convergence, Hessian uncertainties, likelihood scans (standard MINOS implementation $\sim$ completely unusable in this setting)

What is TensorFlow?

- TensorFlow is a library for high performance numerical computation
- Typical workflow:
  - Construct a **computational graph** using TensorFlow library in python
  - Execute graph (transparent-to-user compilation and execution on threaded/vectorized CPU's, GPU's, etc)
- Originally developed at Google for deep learning applications
- **Efficient analytical computation of gradients**, needed for Stochastic Gradient Descent in training of deep neural networks

Likelihood Construction in TensorFlow

- Any template shape fit can be expressed as a many-channel counting experiment
- Negative log-likelihood can be written as

$$L = \sum_{ibin} \left(-n^{\text{obs}}_{ibin} \ln n^{\text{exp}}_{ibin} + n^{\text{exp}}_{ibin} \right) + \frac{1}{2} \sum_{ksyst} \left(\theta_{ksyst} - \theta^{0}_{ksyst} \right)^2 \quad (1)$$

$$n^{\text{exp}}_{ibin} = \sum_{jproc} r_{jproc} n^{\text{exp}}_{ibin,jproc} \prod_{ksyst} \kappa^{\theta_{ksyst}}_{ibin,jproc,ksyst} \quad (2)$$

- $n^{exp}_{ibin,jproc}$ is the expected yield per-bin per-process
- r_{jproc} is the signal strength multiplier per-process
- θ_{ksyst} are the nuisance parameters associated with each systematic uncertainty
- $\kappa_{ibin,jproc,ksyst}$ is the size of the systematic effect per-bin, per-process, per-nuisance
- (The above assumes all shape uncertainties are implemented as log-normal variations on individual bin yields, which is appropriate for e.g. PDF/QCD scale variations, but not for things like momentum scale/resolution variations)

Likelihood Construction in TensorFlow

- Full contents of datacards can be represented by a few numpy arrays:
  - nbin $\times$ nproc 2D tensor for expected yield per-bin per-process
  - nbin $\times$ nproc $\times$ nsyst 3D tensor for κ (actually $ln\kappa$) values parameterizing size of systematic effect from each nuisance parameter on each bin and process (actually two tensors, one each for $\ln \kappa_{up}$ and $\ln \kappa_{down}$ to allow for asymmetric uncertainties)
- POI's and nuisance parameters implemented as TensorFlow Variables
- Full likelihood constructed as TensorFlow computation graph with observed data counts as input
- Some details:
  - Precompute as much as possible with numpy arrays which are compiled into the graph as constants
  - Double precision everywhere
  - Offsetting of likelihood in optimal placement within the graph to minimize precision loss

Minimization

- Minimization in TensorFlow normally done with variations on Stochastic Gradient Descent, appropriate for very large number of parameters in deep learning (10's of thousands to millions)
- For O(100's) of parameters, more appropriate to use second-order minimization techniques
- Hessian can be computed analytically but still slow and not very optimal $\rightarrow$ use quasi-newton methods which approximate hessian from change in gradient between iterations (the MIGRAD algorithm in Minuit/Minuit2 belongs to this class of algorithms, as does BFGS)

Minimization

- **While the likelihood has a global minimum and is well behaved in the vicinity, it is (apparently) NOT convex everywhere in the parameter space**
  - BFGS-type quasi-Newton methods are not appropriate since the Hessian approximation can never capture non-convex features
  - Line search is not a good strategy even with a well-approximated (or exact) Hessian, since this will tend to get stuck or have slow convergence near saddle points/in non-convex regions
  - Major source of non-convexity is the polynomial interpolation of $\ln \kappa$ for asymmetric log normal uncertainties
- Using trust-region based minimizer with SR1 approximation for hessian, as implemented in SciPy (minimal adaptation required for existing TensorFlow-SciPy interface)
  - Bonus: this also supports arbitrary non-linear constraints
  - **Caveat:** Only likelihood and gradient evaluation done in Tensorflow, rest of minimizer is in python/numpy

Implementation

- Prototype implementation in Higgs combination package, re-using datacard parsing code
- Tensorflow graph for likelihood constructed from standard datacards+histograms for template shape analyses, including shape and normalization uncertainties
- Asimov and random toys supported (bayesian or frequentist treatment of nuisance parameters)
- Hessian uncertainties/covariance matrix computed (analytically from tensorflow graph)
- Profile likelihood scans implemented using constraints on parameter values (allows change of basis/parameterization in principle without reparameterizing POI's)

Implementation

- Direct determination of likelihood contours for MINOS-type uncertainties:
  - For the 1 σ uncertainty along direction Δr, minimize $Lerr = -\Delta r \cdot (x - x_0)$ subject to constraint on negative log-likelihood $L - L_0 = 0.5$
  - This works for n-dimensional contours as well (for 2D contours need to scan in one angle)

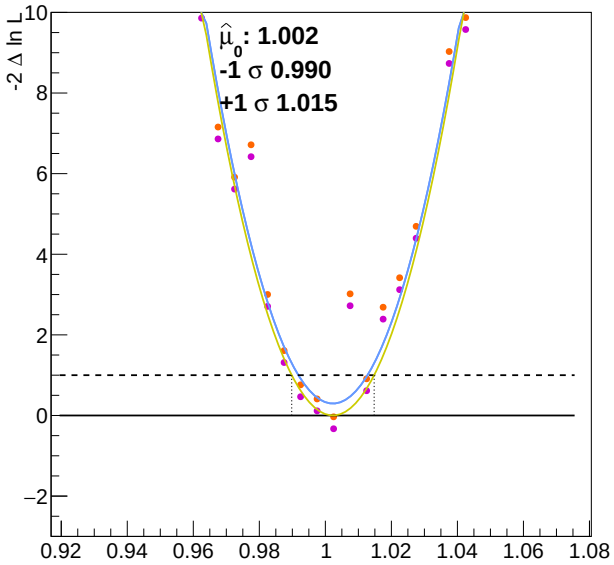
Implementation

- Current version available here:

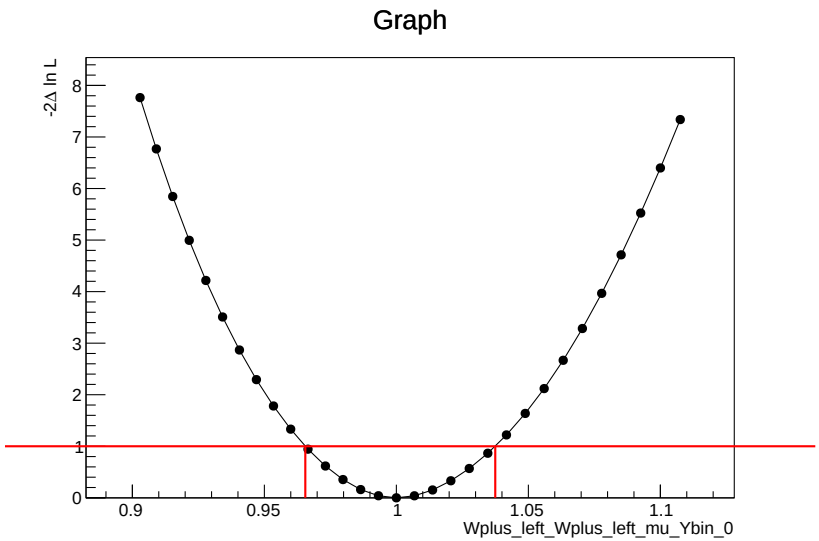
  `https://github.com/bendavid/HiggsAnalysis-CombinedLimit/tree/tensorflowfit_10x`

- Requires SciPy 1.1 (available in CMSSW_10_2_0_pre6 and later)
- Two scripts:
  - **text2tf.py**: Create tensorflow graph from datacards/histograms (outputs ".meta" file containing full graph definition and default values/constants)
  - **combinetf.py**: Run fits/toys/scans with graph

Illustrative Example



(a) Combine w/ GSL-BFGS



(b) TF+SciPy SR1-TR

r = 1.0000 +0.0375 -0.0345

- From current version of W helicity-rapidity cards, for μ^+ channel only
- n.b. vertical lines/asymmetric uncertainties on right are from direct contour determination rather than scan

Some Performance Tests

- Using current W helicity cards (1444 bins, 96 POI's, 70 nuisance parameters)

	Likelihood	Likelihood+Gradient	Hessian
Combine, TR1950X 1 Thread	10ms	830ms	-
TF, TR1950X 1 Thread	70ms	430ms	165s
TF, TR1950X 32 Thread	20ms	71ms	32s
TF, 2x Xeon Silver 4110 32 Thread	17ms	54ms	24s
TF, GTX1080	7ms	13ms	10s
TF, V100	4ms	7ms	8s

- Single-threaded CPU calculation of likelihood is 7x **slower** in Tensorflow than in Roofit (to be understood and further optimized)
- Gradient calculation in combine/Minuit is with 2n likelihood evaluations for finite differences (optimized with caching)
- Xeons are lower clocked than Threadripper, but have more memory channels and AVX-512
- Back-propagation calculation of gradients in Tensorflow is much more efficient (in addition to being more accurate and stable)

Some Performance Tests: Minimization

	Minimization		
	L+Gradient	scipy trust-constr	scipy cpu usage
TF, TR1950X 32 Thread	71ms/call	200ms/iteration	2107%
2x Xeon Silver 4110 32 Thread	54ms/call	237ms/iteration	2587%
TF, GTX1080 (+TR1950X)	13ms/call	84 ms/iteration	1081%
TF, V100 (+2x Xeon 4110)	7ms/call	78ms/iteration	1558%

- Each iteration of the SR1 trust-region algorithm requires exactly 1 likelihood+gradient evaluation
- Significant amount of processing power (and CPU bottleneck) in scipy+numpy parts of the minimizer (non-trivial linear algebra)

Further Optimizing Minimization

- Current SR1 trust-region implementation in scipy based on conjugate gradient method for solving the quadratic subproblem $\rightarrow$ large number of inexpensive sub-iterations which don't parallelize well
- Have implemented a quasi-newton trust region minimizer natively in tensorflow (based on algorithm 4.3 in Nocedal and Wright "Numerical Optimization") and adapted from parts of several minimizers in scipy
- Based on iterative near-exact solution of the trust region problem (fewer number of sub-iterations with heavier matrix algebra per iteration possibly better suited to GPU's)
- Some inefficiencies and numerical protections to be added
- Also have a tensorflow implementation of trust region with conjugate gradient methods, closely following "trust-ncg" in scipy, but replacing exact hessian vector products with quasi-newton approximation

Some Performance Tests: Minimization

	Minimization		
	L+Gradient	scipy trust-constr	TF TrustSR1Exact
TF, TR1950X 32 T	71ms/call	200ms/iteration	89ms/iteration
2x Xeon Silver 4110 32 T	54ms/call	237ms/iteration	63ms/iteration
TF, GTX1080 (+TR1950X)	13ms/call	84ms/iteration	55ms/iteration
TF, V100 (+2x Xeon 4110)	7ms/call	78ms/iteration	51ms/iteration

- Substantial reduction of overhead relative to bare likelihood+gradient call
- Relative remaining overhead much larger on GPU
- n.b, this fit converges in about 500 iterations with the TrustSR1Exact algorithm, about 25s/fit with GPU
- Using gradient descent methods available in Tensorflow requires O(10k) iterations

Linear Algebra Aside

- Fastest way to check if a symmetric matrix is positive definite is to attempt the Cholesky decomposition $A = LL^*$ (with L lower-triangular) and check if it succeeds
- In Tensorflow this produces a fatal error in case the matrix is not positive-definite
- Proper solution is to adapt the corresponding tensorflow operation to return the status together with the factorization (which is allowed to be nonsense in case of failure)
- Underlying Eigen implementation already supports this, so should be straightforward
- Current version of TensorFlow-based minimizer avoids this by performing a more expensive eigen-decomposition

Other Optimization Opportunities

- Detailed study of scaling of minimization overhead/performance with number of free parameters is needed
- Most likely there is further room for improvement with better algorithms/ones more suited for GPU's
- Efficiency of specific matrix factorization steps to be carefully checked/profiled
- Batch evaluation of likelihood feasible/useful? (parallel minimization algorithm? Multiple toys in parallel?)
- Implement simpler χ^2/Gaussian approximation to likelihood for high statistics cases?

Memory Consumption

- Not so good: Memory consumption in this example goes from 500MB in combine to 4.5GB in the tensorflow implementation (albeit not increasing with the number of threads)
- Experimental branch with completely refactored storage/loading of large arrays (using hdf5 files with chunked storage and compression):
  - Memory usage on the same test case back down to 700MB
  - CPU/GPU performance to be evaluated
  - Memory usage for very large test case with 78 nuisances, 1122 processes, 4452 bins at about 6.5GB $\rightarrow$ only slightly larger than raw array size in memory for a single copy
  - Sparse tensor representation now available as well $\rightarrow$ above model reduced to 1.0GB memory consumption

Optimizing Memory Consumption

- To optimize memory consumption for graphs with large constants:
  - **Don't** include large constants in the graph definition (there is also a hardcoded 2GB limit in doing so)
  - **Don't** read large numpy arrays from disk (unless using memmapping, but then can't use compression)
  - **Don't** store large constants in tf Variables (because it's apparently impossible to initialize them without having at least a second copy of the contents in memory)

Optimizing Memory

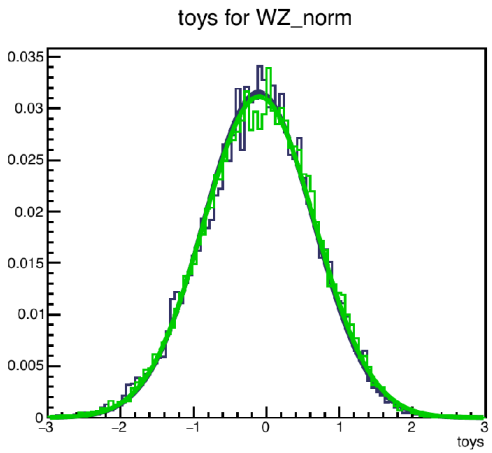
- Adopted solution
  - HDF5 arrays with chunked storage and compression
  - Numpy arrays are stored as flattened HDF5 arrays to allow reading chunk by chunk while preserving the order of the array and maintaining flexibility in choice of chunk size
  - Read chunk by chunk using tf data API with tf py_func to interface with h5py
  - Use batching to reassemble full array into a single tensor, then use the in-memory cache so the read only happens once (reshaping and possible truncation of the overflow from the last batch have near-zero cpu or memory footprint)
  - Text+root histogram conversion has been adapted to write hdf5 arrays instead of a tf graph with in-built constants
- Current (experimental) branch:

  `https://github.com/bendavid/HiggsAnalysis-CombinedLimit/tree/tensorflowfit_h5py_sparse`

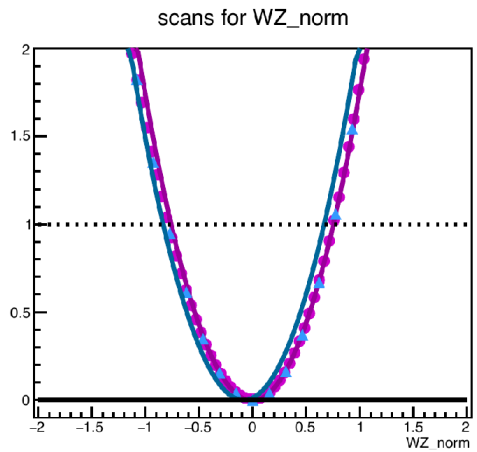
  (text2tf.py script is replaced with text2hdf5.py)
- Optional sparse tensor representation (–sparse option in text2hdf5.py)

Cross-validation with Combine

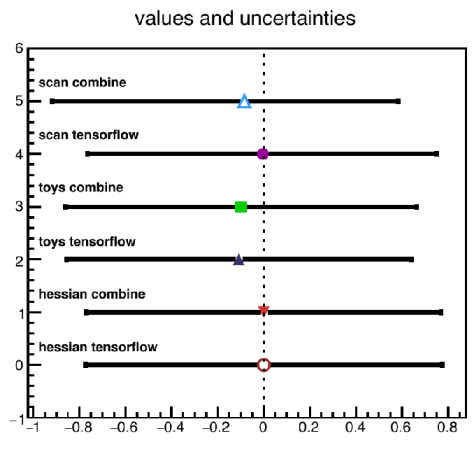
- Direct comparison with combine using the same cards for a simpler example



toys



scans

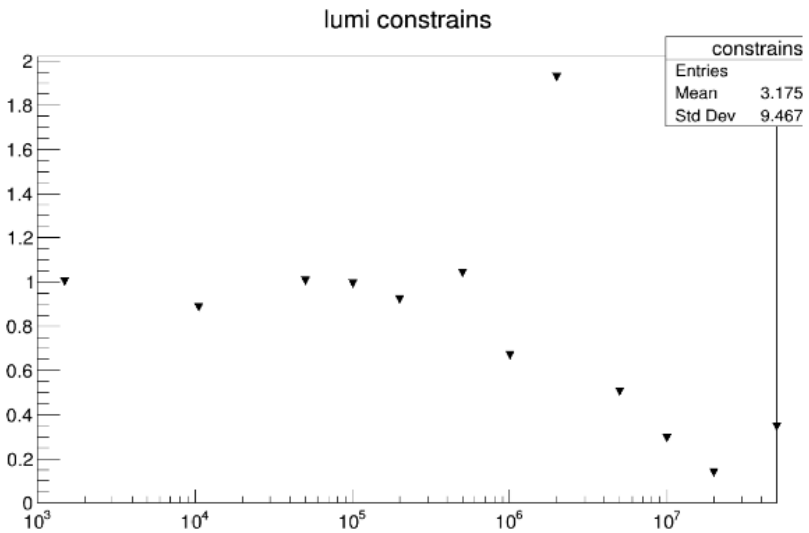


comparison

https://indico.cern.ch/event/735097/contributions/3031878/attachments/1664185/2667224/2018-06-07-wmass.pdf

Provoking Bad Behaviour in Minuit

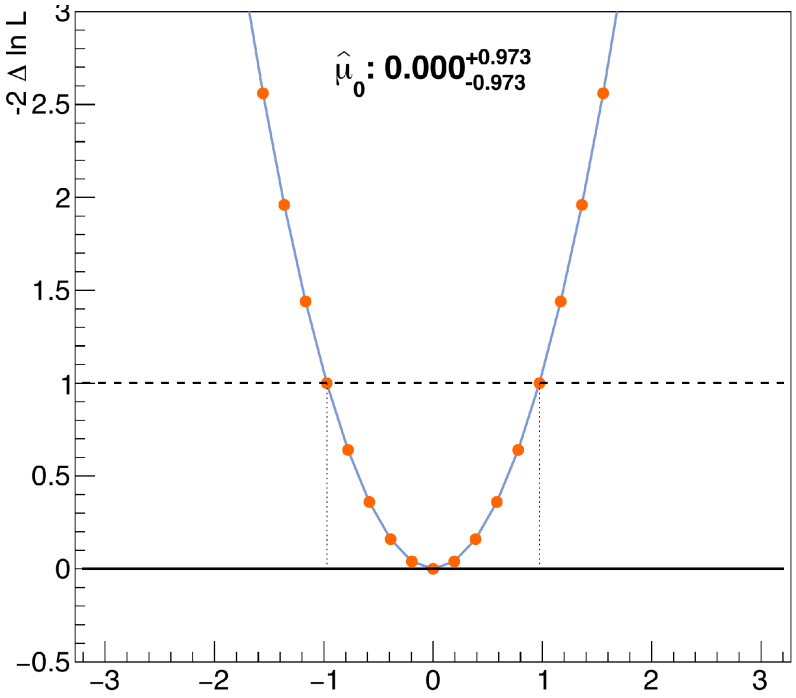
- Artificially small uncertainties from Hesse can appear even in very simple examples with large number of events (in this case, simple counting experiment with one signal process and one lnN systematic uncertainty which should not be constrained at all)
- Some work ongoing from Nick and Andrew to improve these cases



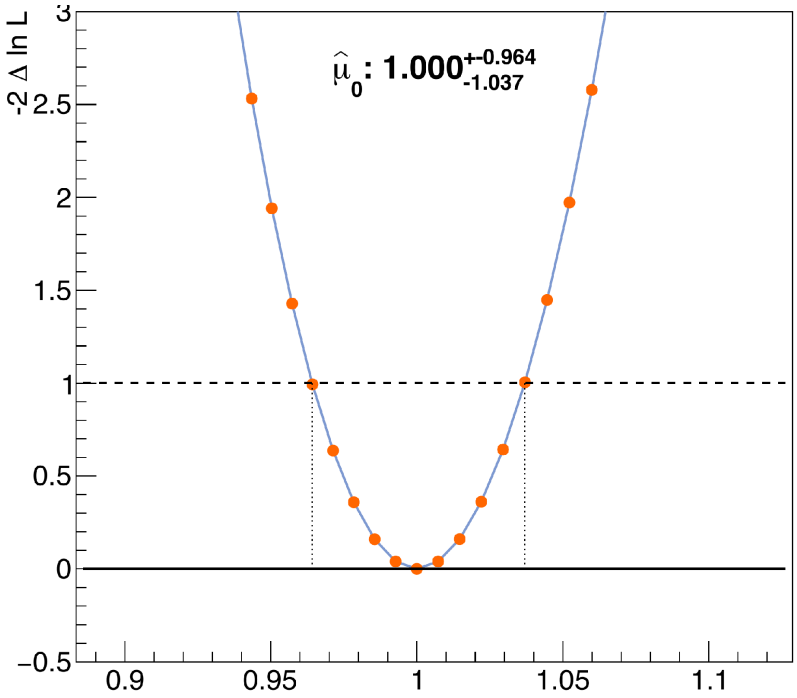
Sample Results in Progress

- W helicity/rapidity measurement foreseen as part of a comprehensive paper on W production at 13 TeV, covering double-differential cross sections in lepton p_T and $|\eta|$, as well as the W production cross section differential in rapidity and decomposed into left, right, and longitudinal polarizations (equivalent to measuring the unpolarized cross section, plus A_0 and A_4 angular cofficients differential in rapidity)
- Many technically challenging fits involved, large numbers of toys eventually needed to validate statistical properties etc
- Showing an assortment of results (all are work-in-progress on the physics side, but things are technically working well)

Likelihood Scans

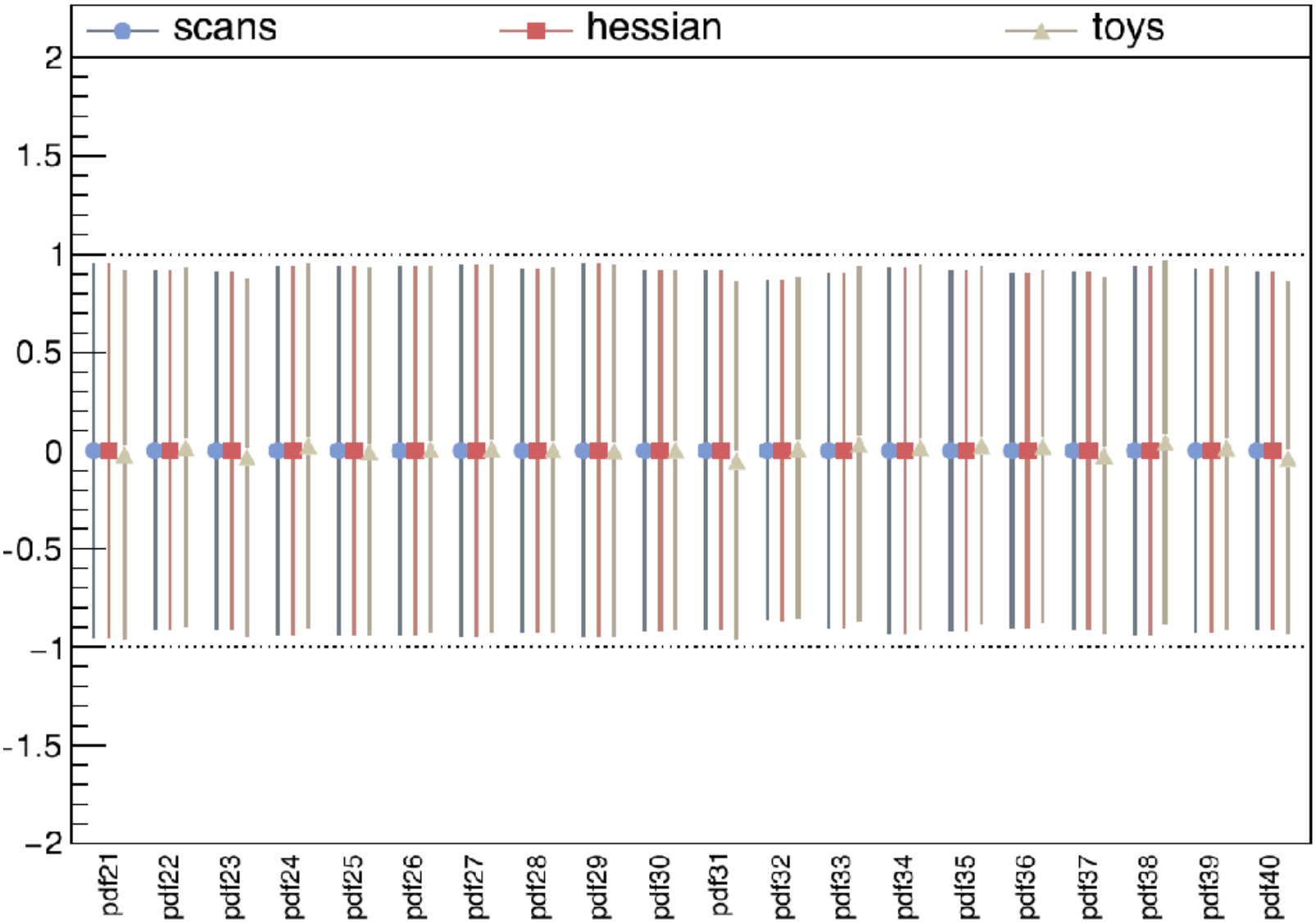


pdf53

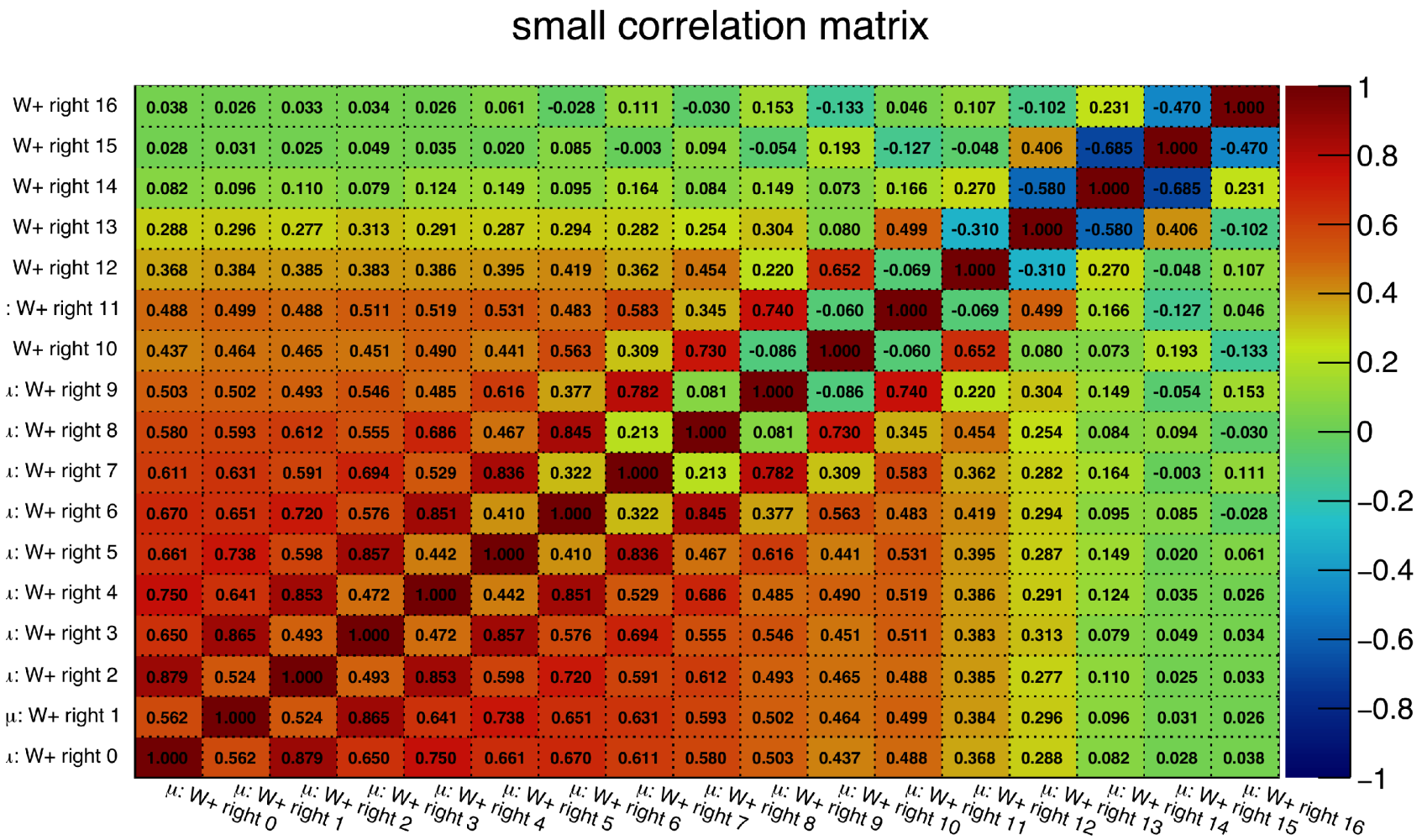


W^+_R Ybin 7

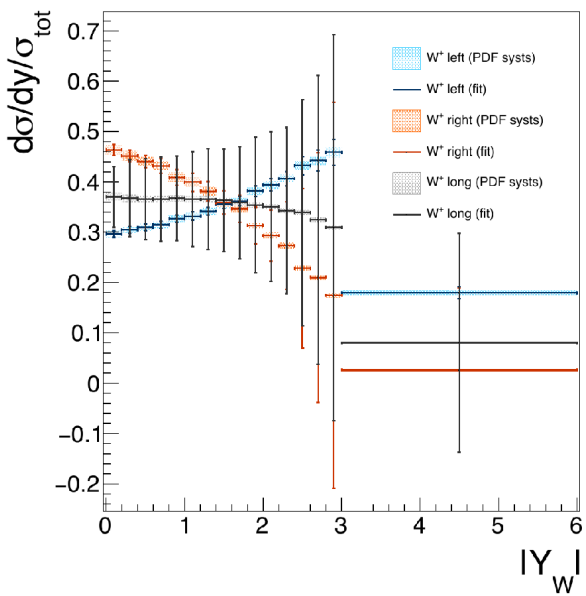
Comparison of Different Uncertainty Methods

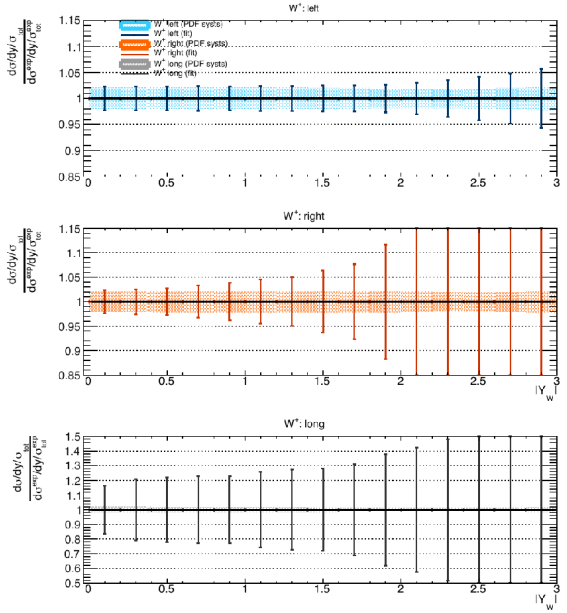


Covariance Matrices



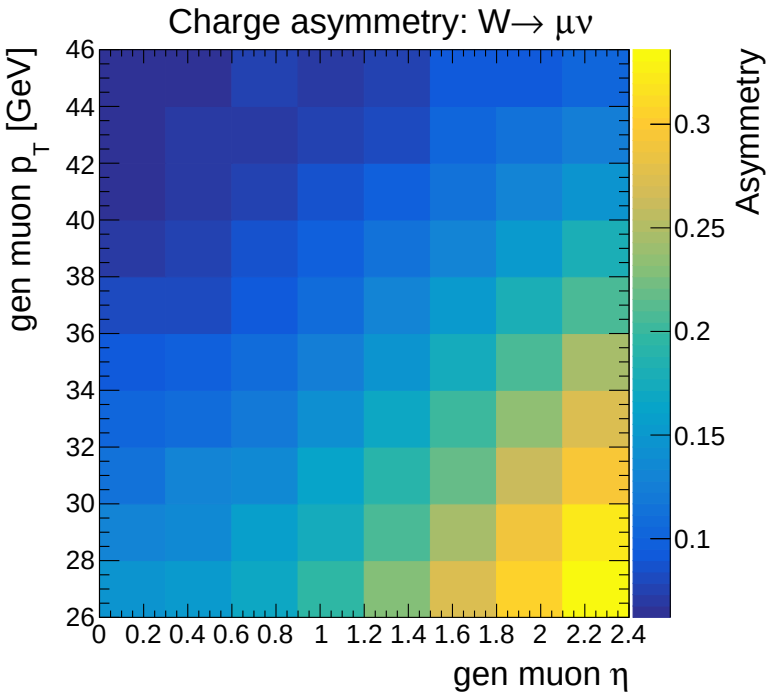
Normalized Cross Sections (Expected)



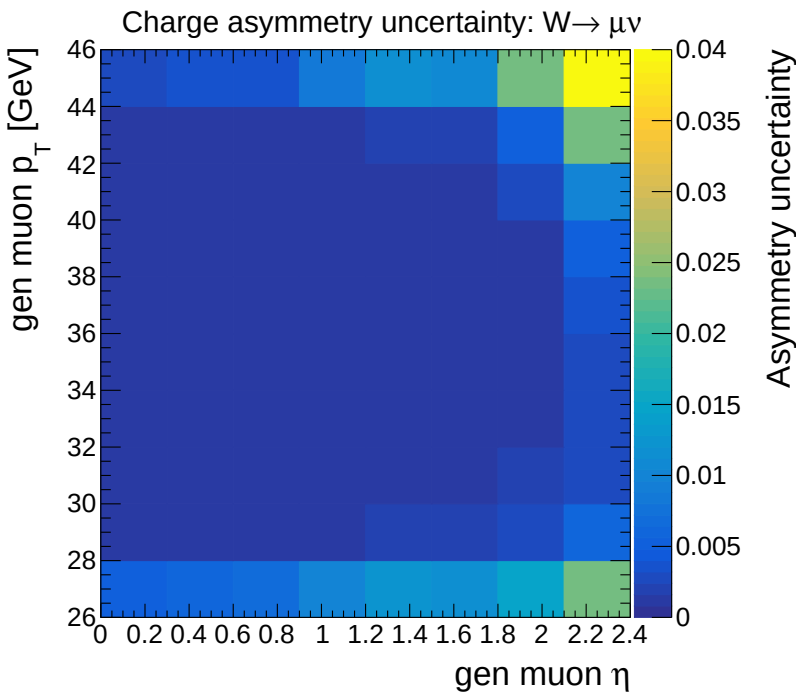


- Note that post-fit cross section is **not** simply signal strength μ times prefit cross section
- Nuisance parameters (e.g. for theory uncertainties may modify the cross section)
- Handled with an implementation of channel masking and normalization tracking following the combine implementation

Double differential lepton Charge Asymmetry (Expected)



(a) Charge Asymm.



(b) Uncertainty

Conclusions

- Construction of likelihood for binned template fits implemented in Tensorflow
- Reasonably stable implementation with basic functionality available, already usable for analysis, with important gains in speed and numerical stability for complex cases
- Additional statistical features to be implemented as needed (e.g. bin-by-bin template uncertainties)
- No plans so far to extend to analytic PDF's (not planning a full re-implementation of RooFit)
- Ongoing studies and work to further understand and optimize performance on GPU's
- Ultimate limits/achievable scale to be further understood