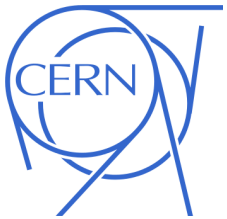
Maximum likelihood fits with TensorFlow

Josh Bendavid



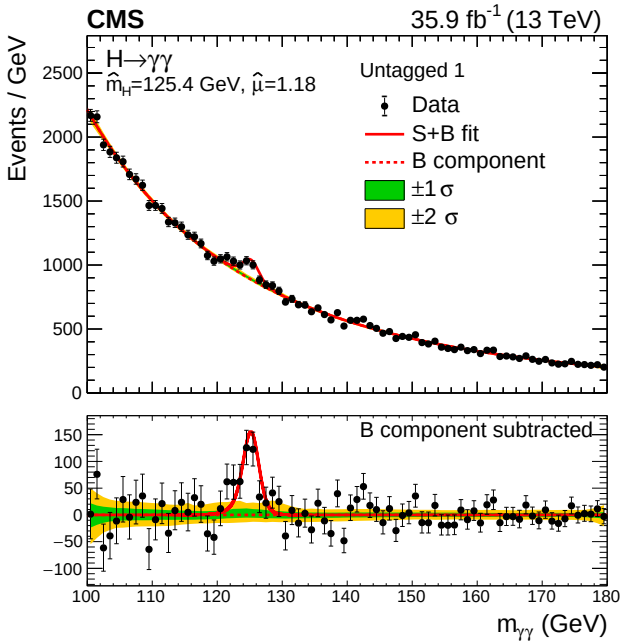
Jun. 4, 2019

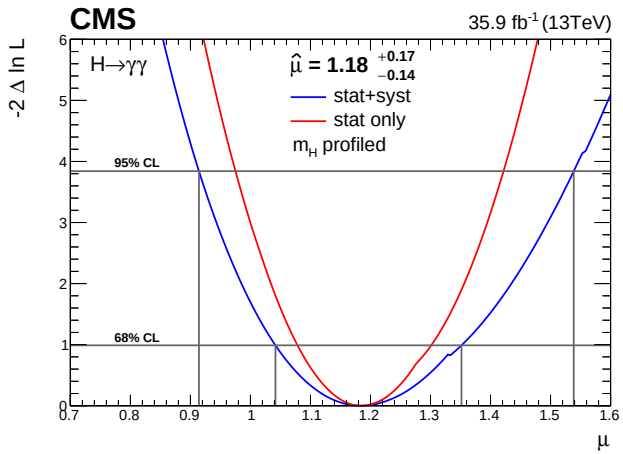
Introduction: Maximum Likelihood Fits

- Common framework for statistical interpretation of HEP data: **Maximum Likelihood Fits**
  - Maximize the joint probability of the data $\vec{x}$ given some parameters of the model $\vec{\theta}$ which may include both **parameters of interest** (POIs) such as production cross sections, particle masses, etc, as well was **nuisance parameters**, e.g. reconstruction efficiency or energy scale allowed to vary within some prior constraint
- Two variants:
  - **Unbinned Maximum Likelihood Fit:** Typically a small number of observables (often 1, rarely more than 3) with a large number of events, evaluate the continuous probability density for each data event: $-\ln L = \sum_{events} p(\vec{x}_{ievent}|\vec{\theta})$
  - **Binned Maximum Likelihood Fit:** Likelihood is evaluated using bin counts in a histogram: $-\ln L = \sum_{bins} p(N_{ibin}|\vec{\theta})$

Introduction: Maximum Likelihood Fits

- **Unbinned Maximum Likelihood Fit:** Typically a small number of observables (often 1, rarely more than 3) with a large number of events, evaluate the continuous probability density for each data event: $-\ln L = \sum_{events} p(\vec{x}_{ievent}|\vec{\theta}) \rightarrow$ **small feature space, many examples**
- **Binned Maximum Likelihood Fit:** Likelihood is evaluated using bin counts in a histogram: $-\ln L = \sum_{bins} p(N_{ibin}|\vec{\theta}) \rightarrow$ **Moderately sized feature space, 1 example**

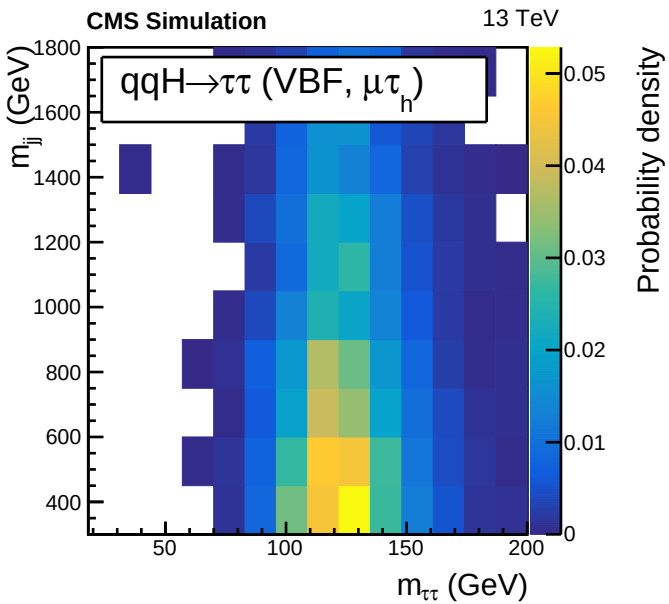


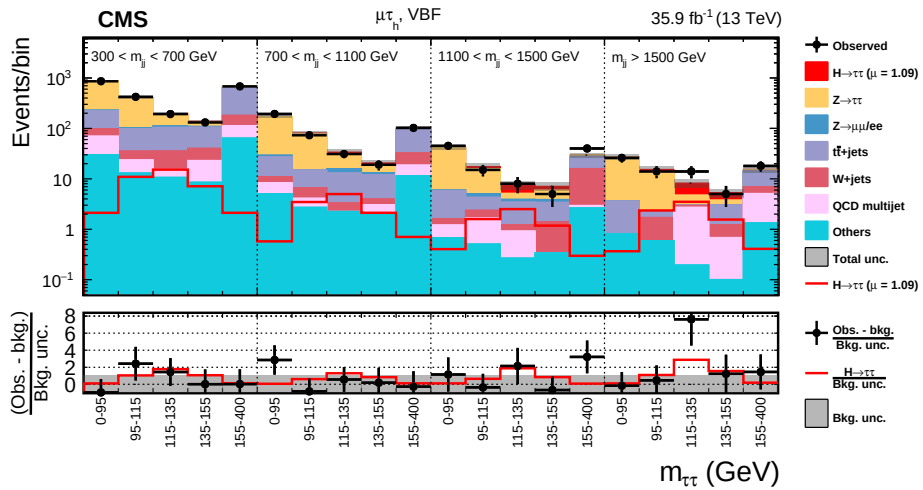


JHEP 1811 (2018)

Introduction: Maximum Likelihood Fits

- Special case: **Binned template fits**: Probability for observing a given number of counts in a given histogram bin is itself encoded in a set of histogram "templates" which are scaled and/or interpolated as a function of the model parameters
- Multi-dimensional histograms can always be "unrolled"





Phys. Lett. B 779 (2018) 283

Introduction

- Discussing a particular test case for binned maximum likelihood fit with templates here related to W cross section measurements:
  - $\sim$ 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

Why TensorFlow?

- Most important in this context: **Efficient and numerically stable computation of gradients** (using standard backprop)
- Parallelization, use of GPU's etc also interesting

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 loaded into graph via tf data api from h5py arrays on disk
  - Double precision everywhere
  - Offsetting of likelihood in optimal placement within the graph to minimize precision loss

Likelihood Construction in TensorFlow

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

$$L = \sum_{ibin} \left(-n^{\text{obs}}_{ibin} \ln n^{\text{exp}}_{ibin} + n^{\text{exp}}_{ibin} \right) + \frac{1}{2} \sum_{ksyst} \left(\theta_{ksyst} - \theta^{0}_{ksyst} \right)^2 \tag{3}$$

$$n^{\text{exp}}_{ibin} = \sum_{jproc} r_{jproc} n^{\text{exp}}_{ibin,jproc} \prod_{ksyst} \kappa^{\theta_{ksyst}}_{ibin,jproc,ksyst} \tag{4}$$

- Likelihood evaluation reduced to essentially two large tensor contractions (matrix multiplications)
- Both dense and sparse implementations are used as appropriate

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-1000's) of parameters, more appropriate to use second-order minimization techniques
- **Particularity:** Loss function needs to be minimized **exhaustively**. There is a global minimum, and further statistical analysis (determining confidence interevals etc) requires finding it to high accuracy
- 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
- Started with 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

Some Performance Tests

	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

- (1444 bins, 96 POI's, 70 nuisance parameters)
- n.b. these numbers are with an older implementation, all have improved
- 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)
- Best-case speedup is already a factor of 100

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 several variants of quasi-newton trust region minimizers natively in TensorFlow
- Most advanced based on L-SR1 Orthonormal basis minimization (arXiv:1506.07222), including a new non-limited-memory variant with direct update to eigen-decomposition of Hessian
- Hessian-free methods (e.g "trust-krylov" in SciPy) are also interesting since they can be used with exact Hessian-vector products computed efficiently with backprop, but in practice these require many Hessian-vector product evaluations per-iteration

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

- Example here with iterative Cholesky decomposition to solve TR subproblem (a la Nocedal and Wright algo 4.3)
- 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

Updated Performance Tests

(Newer TensorFlow, further optimized, but larger model)

	Likelihood	L+Grad	Hessian	MaxRSS
TF, TR1950X 1 Thread (pfor)	26ms	73ms	7.9s	3000MB
TF, TR1950X 32 Thread (pfor)	39ms	83ms	1.1s	3900MB
TF, GTX1080 (+TR1950X) (loop)	64ms	69ms	3.0s	2900MB
TF, GTX1080 (+TR1950X) (pfor)	64ms	69ms	0.8s	2900MB

- (1824 bins, 101 processes, 96 POI's, 257 nuisance parameters)
- Size of raw arrays is 760MB
- non-pfor hessian calculation failed with "Already exists: Resource" errors without " on CPU

Updated Performance Tests: Large/Sparse Model

	Likelihood	L+Grad	Hessian	MaxRSS
Sparse TF, TR1950X 1 Thread	24ms	40ms	52s	980MB
Sparse TF, TR1950X 32 Thread	40ms	70ms	3.7s	1200MB
Dense TF, TR1950X 1 Thread	245ms	540ms	-	6800MB
Dense TF, TR1950X 32 Thread	237ms	534ms	-	7000MB

- (1296 bins, 655 processes, 648 POI's, 444 nuisance parameters)
- GPU not available with standard build (SparseTensorDenseMatMul)
- Size of raw arrays in dense mode is $6GB$
- pfor for Hessian not available in Sparse case (SparseTensorDenseMatMul not supported)
- Hessian computation in dense mode caused OOM with pfor, and "Already exists: Resource" errors without
- Dense model too big for my GPU

Optimizing Memory Consumption

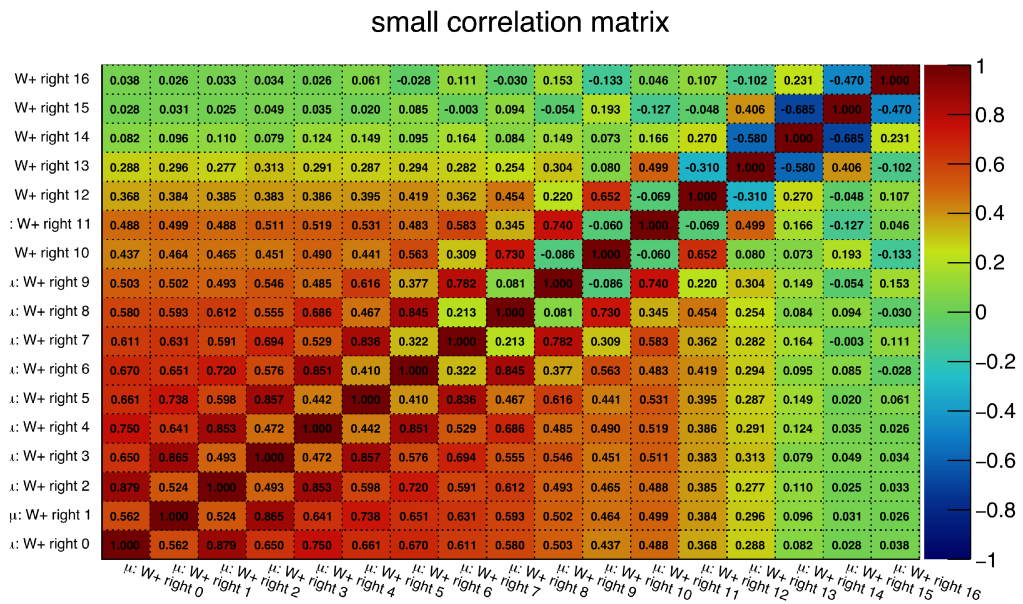
- This type of model has a peculiar feature of very large constants (3-tensor representing systematic variations on templates can be several GB especially in dense mode with larger numbers of processes and systematic variations)
- To optimize memory consumption for graphs with large constants:
  - **Don't** 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
- (Avoiding a second copy in memory took some patience and was not obvious how to achieve)

Covariance Matrices

- Irrespective of minimization algorithm, often want to compute covariance matrix at the end for interpreting uncertainties $\rightarrow$ compute Hessian and invert it
- New vectorized pfor construction gives large speedups for this (so much that full second-order minimization methods are even feasible in some cases)



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

Some Technical Feedback

- From first tests pfor works very well for Hessian computation in this context
- A few things which would be nice to have:
  - GPU support for double-precision in SparseTensorDenseMatMul (trivial, just some missing macro calls)
  - Vectorized pfor support for SparseTensorDenseMatMul (for Hessian computation)
  - int32 support for SparseTensor indices
  - More streamlined/memory efficient handling of large constants
  - Option to return status codes for matrix decomposition instead of throwing an exception (particularly for Cholesky decomposition, where underlying Eigen code already supports it $\rightarrow$ very useful for steering behaviour depending on positive-definiteness of a matrix)
  - Would be nice if tf.where could suppress NaN's from non-selected branches
- Have not moved to Tensorflow 2.0 yet (and no very strong need for eager execution, though it could simplify implementation of minimization algorithms)

Implementation

- Code lives here: `https://github.com/bendavid/HiggsAnalysis-CombinedLimit/tree/tensorflowfit` (not very streamlined for the moment, since the priority has been on a particular set of physics analyses in progress with it, and currently somewhat intertwined with existing CMS fitting tools)
- Two scripts:
  - **scripts/text2tf.py**: Create tensorflow graph from datacards/ROOT histograms (outputs hdf5 file containing flattened arrays for large constant tensors)
  - **scripts/combinetf.py**: Construct graph, load constant arrays into tensors, run fits/toys/scans with graph
- Some interesting bits related to reading hdf5 arrays, some sparse tensor operations, and minimization in python area
- Second order minimizers will be interesting to contribute upstream (and some work already on L-SR1 algorithms for more conventional deep learning applications, e.g arXiv:1807.00251)

Other Related Work

- Some related efforts by others in parallel:
  - pyhf: `https://github.com/diana-hep/pyhf`
  - ZFit: `https://github.com/zfit/zfit`

Other Use Cases

- Another interesting use case: Monte Carlo Phase space integration
- Technically closer to traditional machine learning use cases with generative models
- e.g arXiv:1707.00028

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, especially with respect to tradeoffs of different minimization algorithms
- Ultimate limits/achievable scale to be further understood