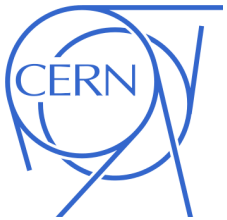
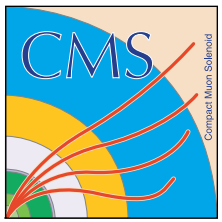
Tensorflow-based Maximum Likelihood fits for High Precision Standard Model Measurements at CMS

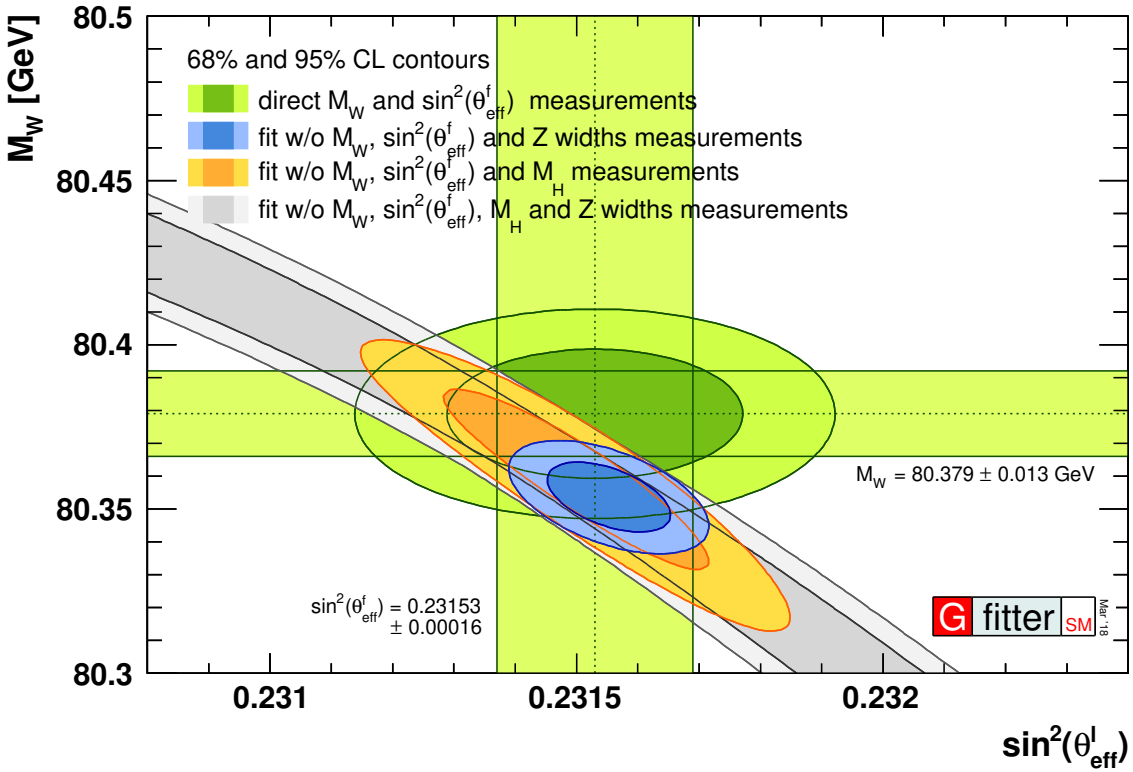
Josh Bendavid (CERN)
on behalf of the CMS collaboration





Jul. 16, 2020
PyHEP 2020

Introduction



- Precision measurements of electroweak parameters provide stringent tests and an opportunity to overconstrain the Standard Model

Introduction

- This talk: Introduction to recent extensive CMS measurement of W production/decay properties: **CMS-PAS-SMP-18-012**
- Deep dive into the technical/statistical aspects of the binned maximum likelihood fit used to extract the results
- Corresponding implementation details

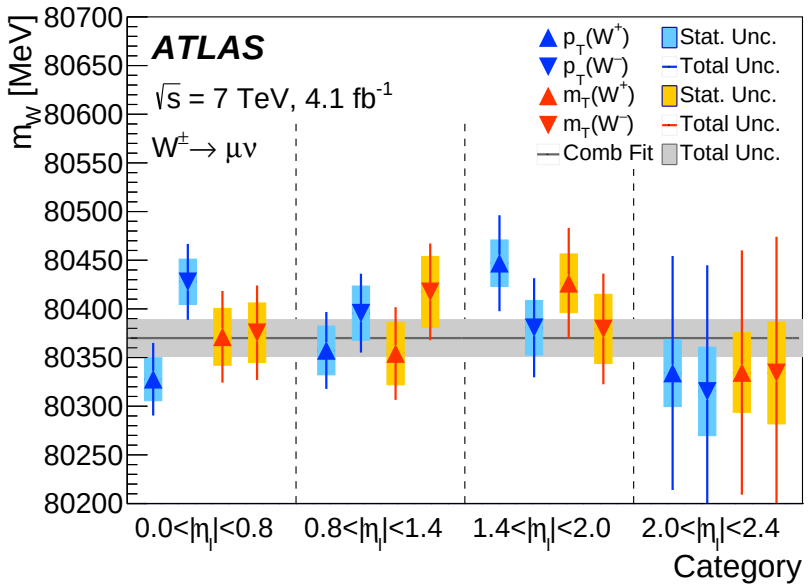
W mass: PDF Uncertainties

Eur. Phys. J. C 78 (2018) 110 (ATLAS)

$m_W = 80370 \pm 7(\text{stat.}) \pm 11(\text{exp.}) \pm 8.3(\text{QCD}) \pm 5.5(\text{EWK}) \pm 9.2(\text{PDF})$ MeV

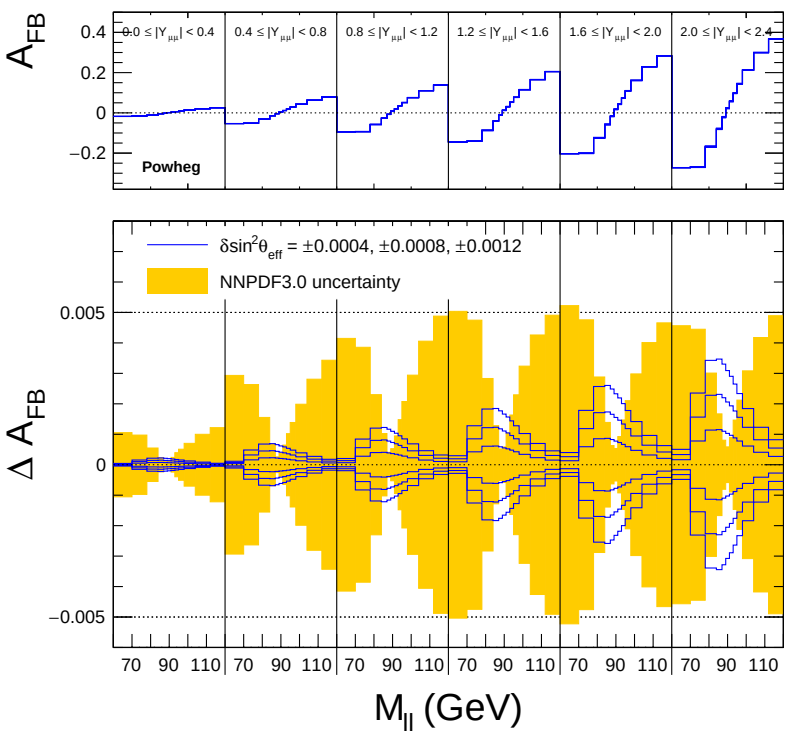
	PDF Uncertainty (MeV)
per $\lvert\eta\rvert$-charge cat.	20-34
per-charge	14-15
full combination	9.2

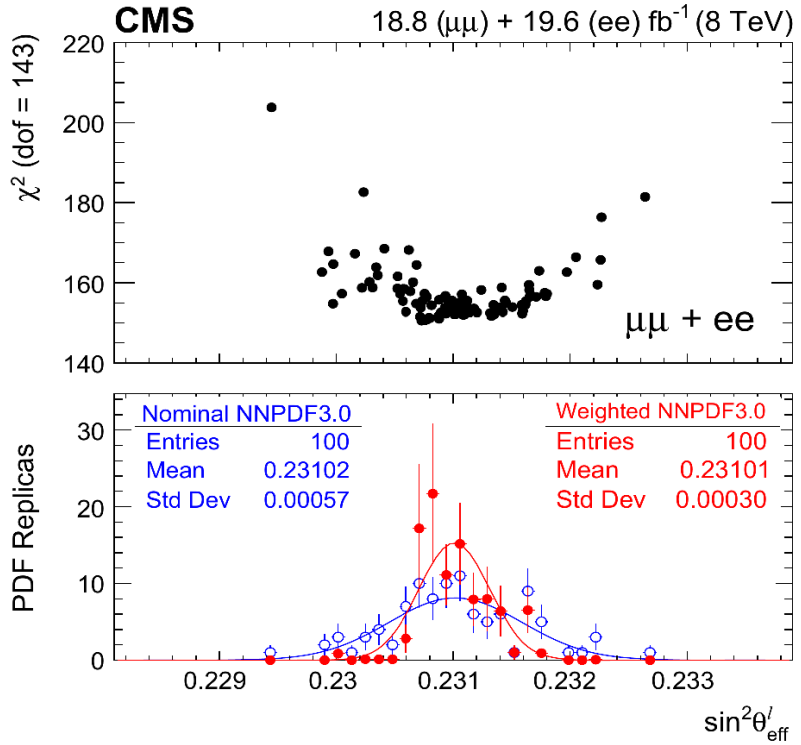
- PDFs are one of the largest sources of uncertainty for m_W measurements at the LHC,
- PDFs determine the W rapidity spectrum and lepton decay angles through W polarization
- Well-defined correlations between phase space regions and processes which are already partly exploited in present measurement to reduce uncertainty. Can be further exploited in the future



In-situ PDF constraints: Weak Mixing Angle Case

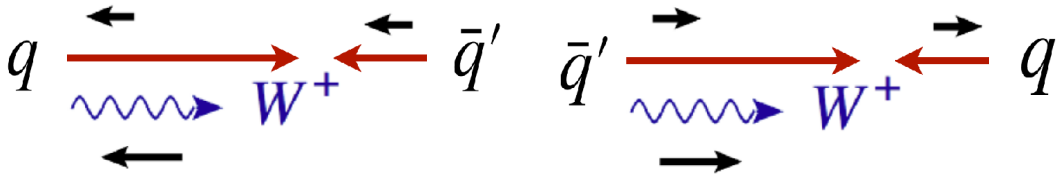
- CMS and ATLAS weak mixing angle measurements exploit in-situ constraints to reduce PDF uncertainties with Bayesian reweighting of Monte Carlo replicas/profiling of nuisance parameters associated with Hessian representation (numerically equivalent in the Gaussian limit)



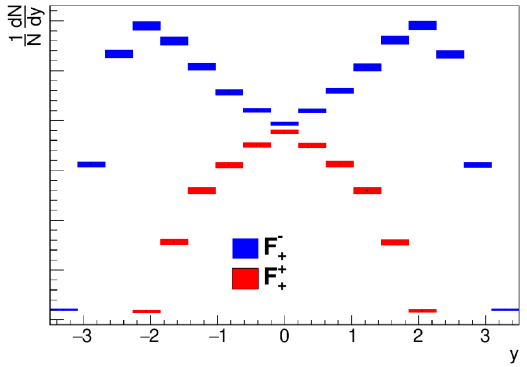


Eur. Phys. J. C (2018) 78: 701 (CMS)

W Helicity/Rapidity at LHC



(a) left-handed W^+ (b) right-handed W^+

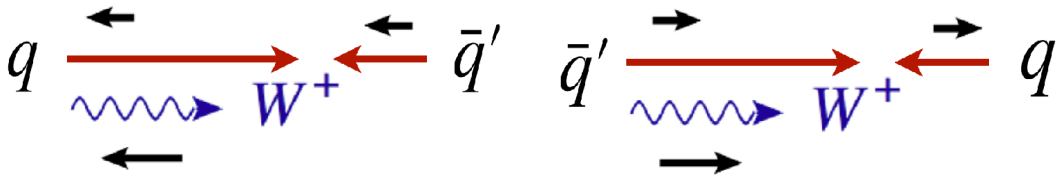


(c) W^+ Rapidity

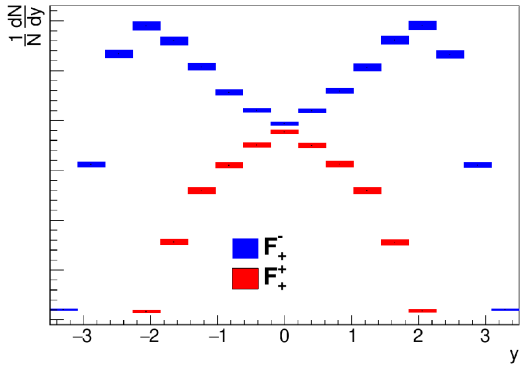
- At tree level:
  - All W production at LHC is $q\bar{q}$ induced
  - Direction of the W relative to the incoming quark determines the helicity
  - Only two helicity amplitudes/polarization states
  - W has zero transverse momentum
  - **Full information on valence quark PDF's in the relevant x range contained in $d\sigma/dy$ broken down into the two helicity states**

JHEP12(2017)130 E. Manca, O. Cerri, N. Foppiani, G. Rolandi

W Helicity/Rapidity at LHC



(a) left-handed W^+ (b) right-handed W^+

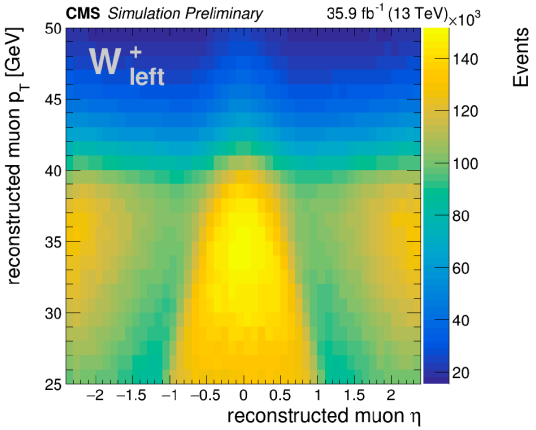


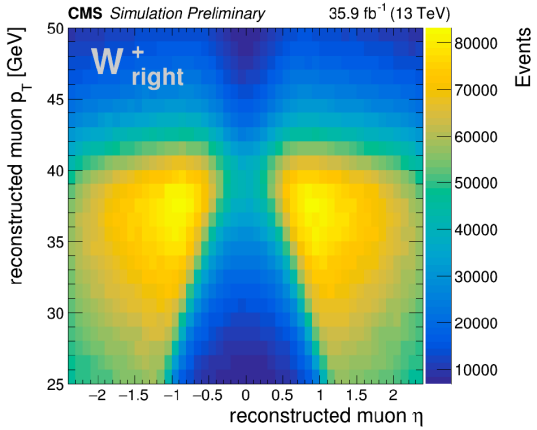
(c) W^+ Rapidity

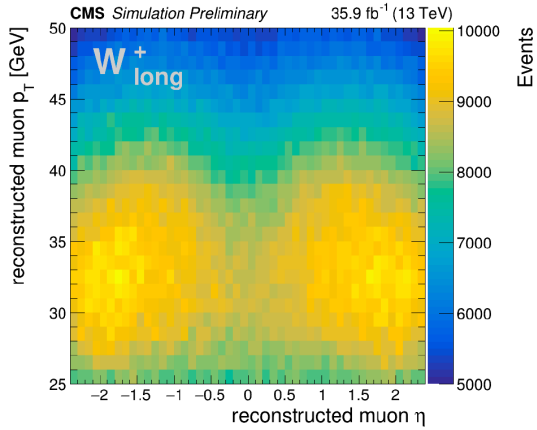
- Direction of incoming quark depends even more on PDF's in pp vs $p\bar{p}$ collisions
- gluon-induced contribution from higher order effects larger and more uncertain (also due to higher E_{cm} compared to Tevatron)

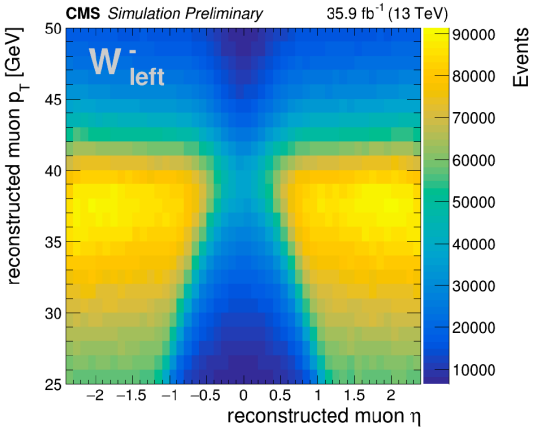
JHEP12(2017)130 E. Manca, O. Cerri, N. Foppiani, G. Rolandi

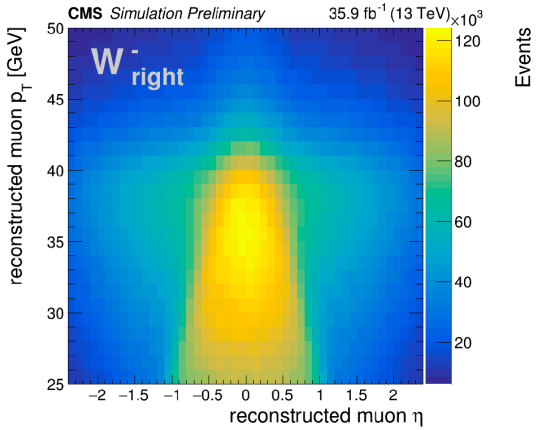
W Helicity/Rapidity at LHC

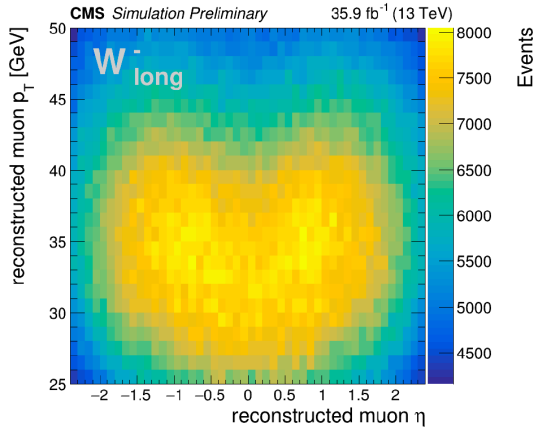






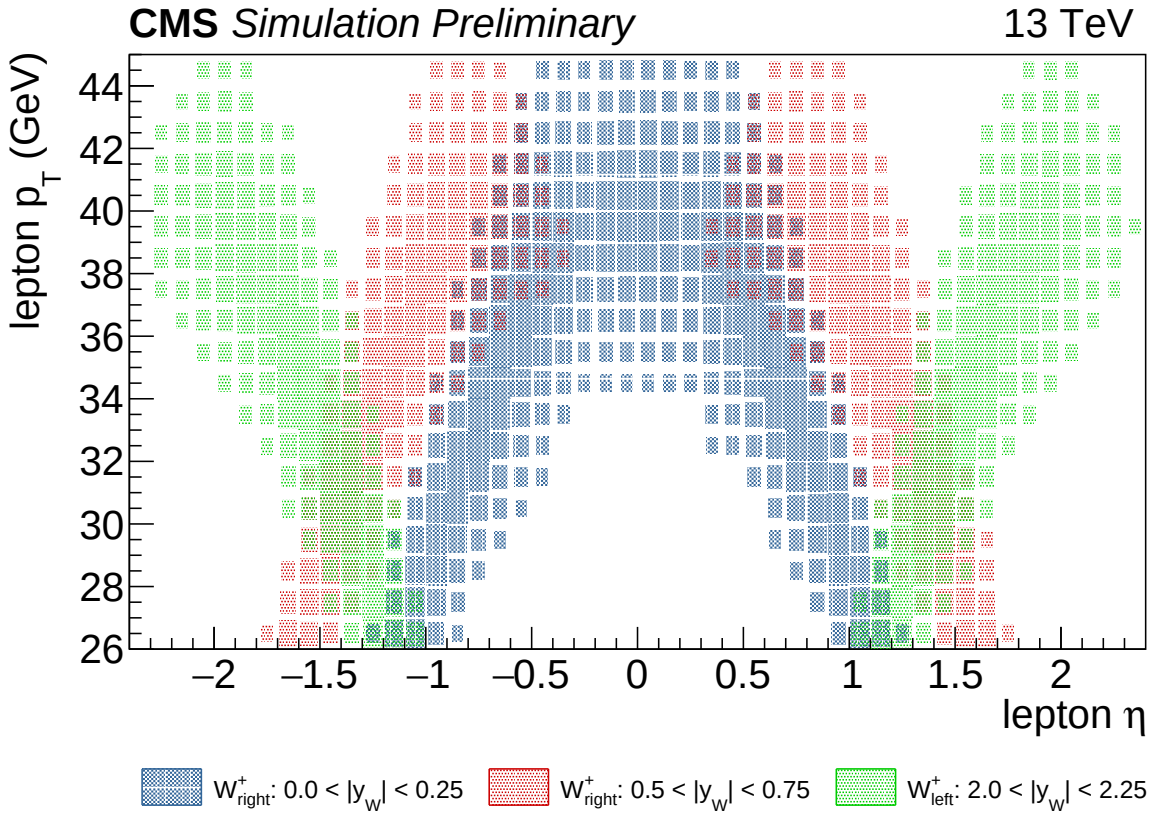






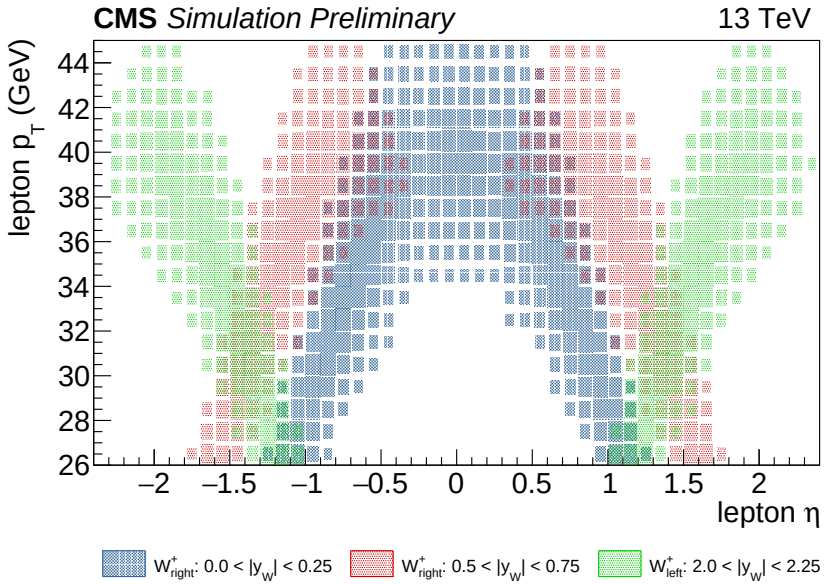
- 2D distribution of charged lepton p_T and η can discriminate between helicity states as well as rapidity of the W

W Helicity/Rapidity at LHC



- 2D distribution of charged lepton p_T and η can discriminate between helicity states as well as rapidity of the W

W Helicity/Rapidity at LHC



- Left and right polarization components can be extracted simultaneously as a function of W rapidity, using only charged lepton kinematics
- Avoids dependence on less precisely measured missing transverse momentum (at the cost of some statistical dilution)
- Avoids circular dependence on PDFs since quark vs anti-quark fraction for each rapidity is **measured**

The Measurement

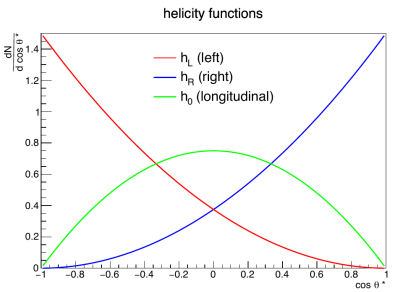
- **Measurement Strategy:**
  - Measure $d\sigma^{\pm}_{L/R}/dy$ (single-differential cross section in terms of boson rapidity, split in charge and left and right helicity) from 2D distribution of charged lepton p_T and η
  - Longitudinal component is partly regularized (constrained to $\pm 30\%$ around predicted value)
- **Alternative Measurement strategy:**
  - Measure double-differential cross section $d\sigma^{\pm}/dp_T d\eta$ in terms of (dressed) charged-lepton kinematics (from 2D distribution of charged lepton p_T and η)
- **Event Selection:**
  - Basic single lepton selection (ID+isolation)
  - $p_T > 26$ (30) GeV for muons(electrons)
  - Additional lepton veto with looser selection to suppress Drell-Yan
  - Cut on $M_T > 40$ GeV to suppress QCD

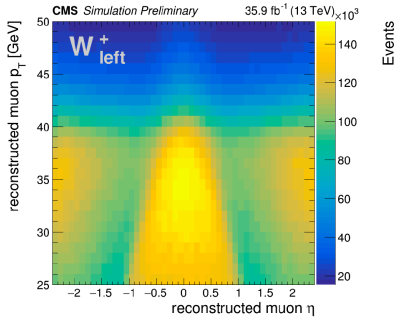
The Measurement

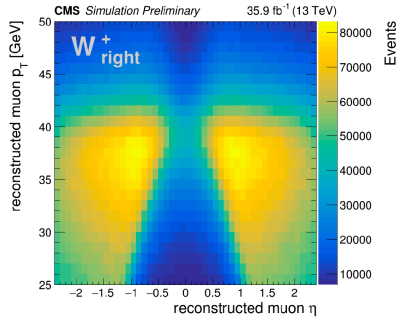
- **Theoretical Ingredients:**
  - Construction of signal templates (W production split in charge, helicity, rapidity)
  - Electroweak background prediction
- **Experimental Ingredients:**
  - Lepton efficiencies
  - Lepton energy/momentum scale
  - QCD Background Estimate
  - Extraction of cross sections

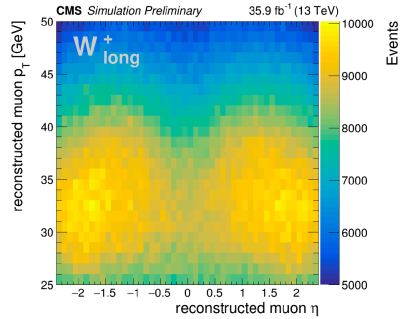
Template Construction: Helicity/Rapidity

- Monte Carlo generators cannot generally produce individual helicity components[1]
- Nominal signal MC for this analysis: MG5_aMC@NLO + Pythia 8 0+1+2 jets NLO (FXFX-merged) w/ NNPDF3.0 NLO (around 700M events with 15% negative weights, about 1/6th the statistical power of the data)
- To construct polarized templates:
  - Bin Monte Carlo in (born) p_T^W, y^W and fit $\cos\theta^*$ distribution in each bin to extract helicity fractions
  - Reweight MC as a function of $p_T^W, y_W, \cos\theta^*$ to construct three individual components for left/right/longitudinal
  - Build two-dimensional templates in (detector-level) charged lepton p_T and η







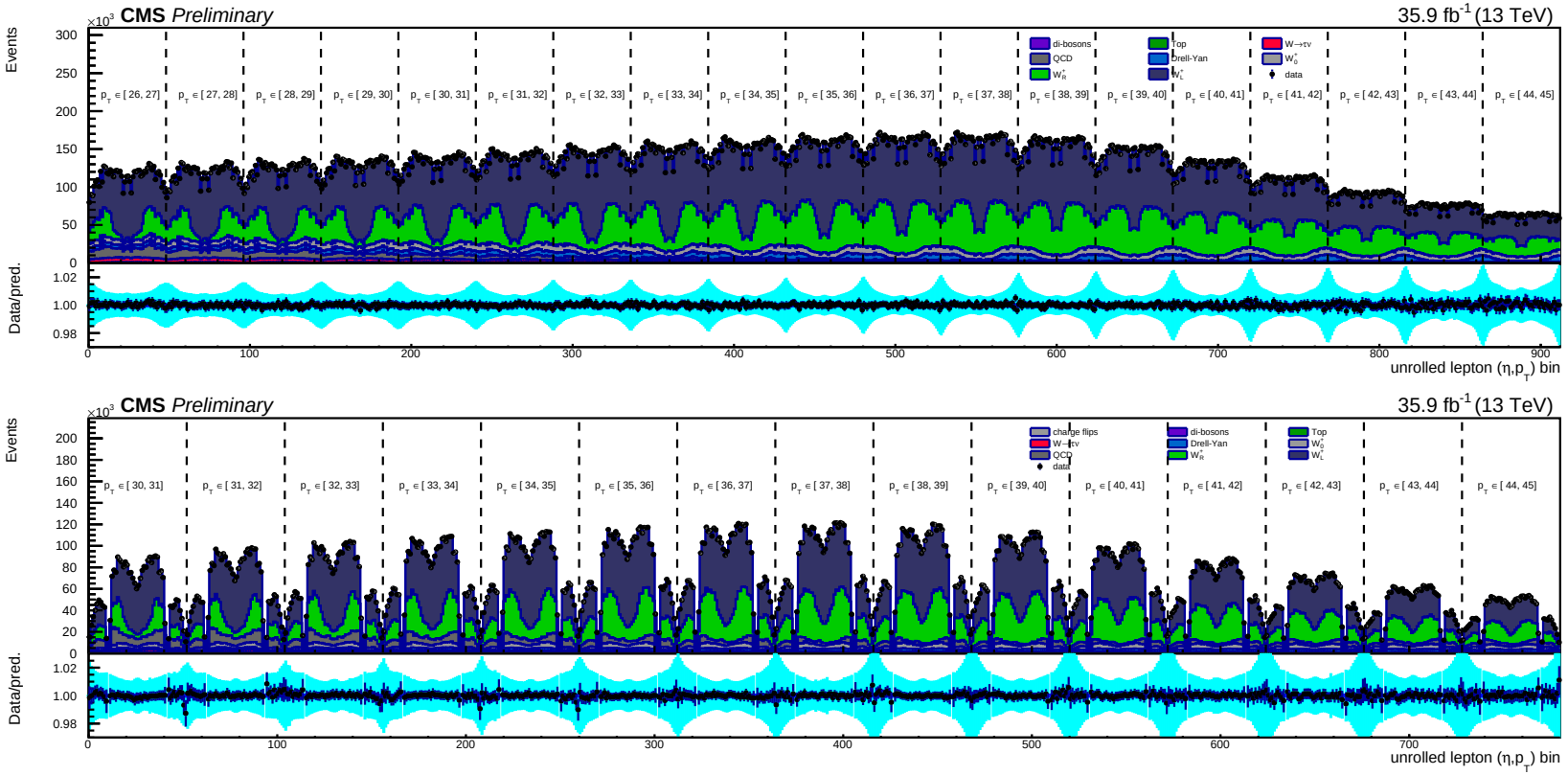


[1] Upcoming verison of madgraph will support this at tree level: arXiv:1912.01725

Summary of Uncertainties

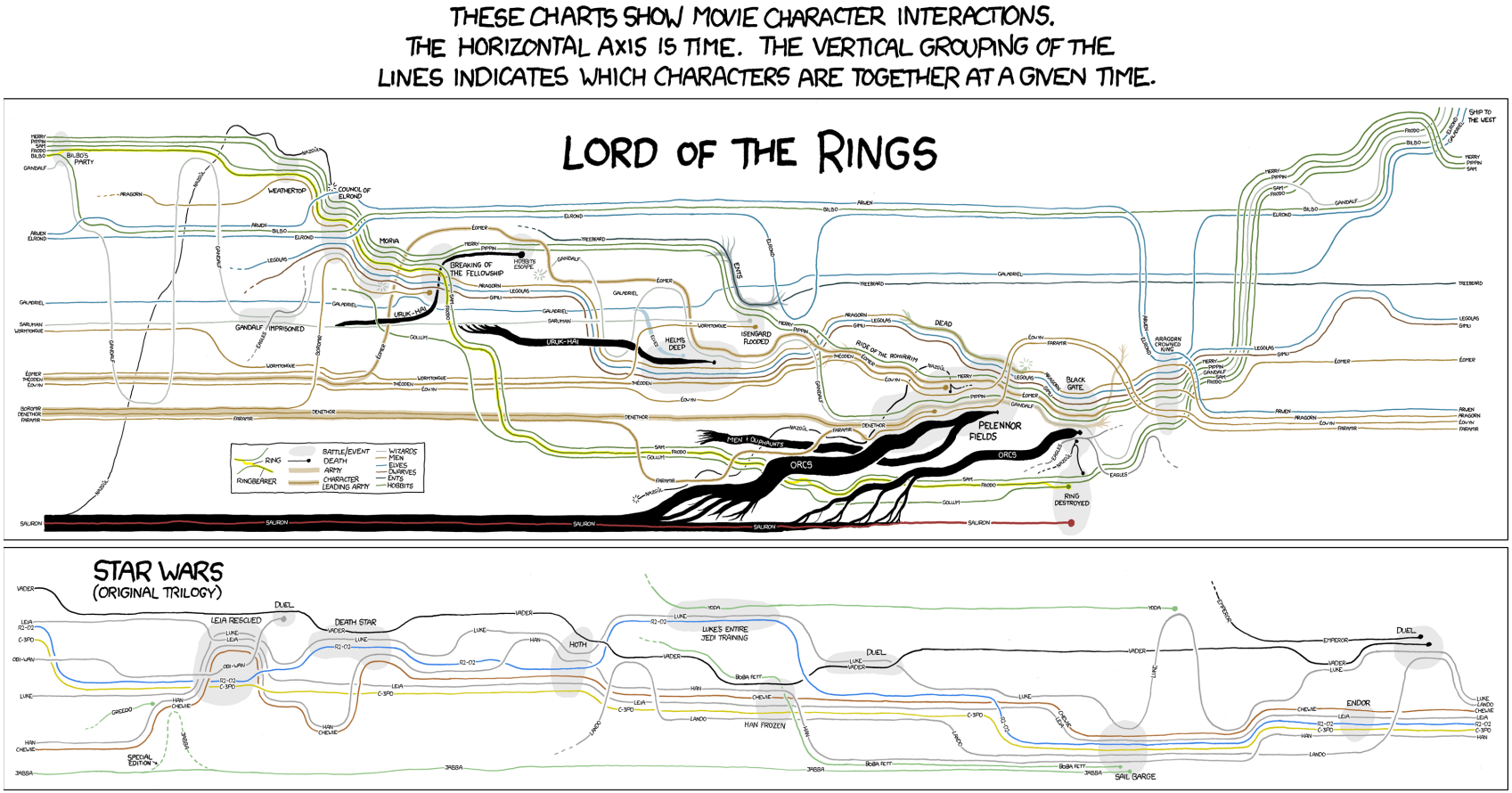
- Extremely detailed breakdown of systematic uncertainties including
  - QCD Renormalization and factorization scales, decorrelated in bins of Wp_T to allow shape variations, decorrelated in W charge and helicity for conservative theory treatment
  - Lepton efficiencies in 48 bins of η, smoothed in p_T, with 3 parameters per η bin for normalization and p_T-dependence variations
  - QCD background estimate and uncertainties (with correlated and de-correlated components vs p_T and η
  - Lepton energy/momentum scale uncertainties: Statistical uncertainties from O(100) nuisance parameters per flavour (from diagonalized bootstrap variations), plus systematic variations

All Together



Post-fit unrolled detector-level plots

All Together



xkcd plot which is approximately as easy to read from my slides

Maximum Likelihood Fit

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

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

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

- Extract cross sections from fully profiled maximum-likelihood fit
- Some numbers for helicity fit:
  - 40 measured cross sections (POI's)
  - 78 total processes
  - 3320 bins
  - 1354 systematics/nuisance parameters
- Some challenges:
  - 10's of thousands of histograms entering the fit
  - gradients in minimization need to be known accurately
  - good convergence behaviour of fit must be maintained
  - time/memory of minimization need to be kept under control
  - uncertainties covariances need to be accurately computed

Maximum Likelihood Fits in 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 and numerically accurate semi-analytical computation of gradients by back-propagation**, needed for Stochastic Gradient Descent in training of deep neural networks
- Currently using Tensorflow 1.x

Likelihood Construction in TensorFlow

- Negative log-likelihood can be written as

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

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

- $n^{exp}_{ibin,jproc}$ is the expected yield per-bin per-process
- μ_{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
- In the above, all uncertainties are implicitly implemented as log-normal variations on individual bin yields with appropriate correlation structure encoded in κ's (asymmetric log-normal uncertainties are allowed, but details not explicitly written out here)

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 a 4D tensor nbin $\times$ nproc $\times$ nsyst $\times$ 2 to accommodate $\ln \kappa_{up}$ 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

- Negative log-likelihood can be written as

$$L = \sum_{ibin} \left(-n^{obs}_{ibin} \ln n^{exp}_{ibin} + n^{exp}_{ibin} \right) + \frac{1}{2} \sum_{ksyst} \left(\theta_{ksyst} - \theta^{0}_{ksyst} \right)^2 \tag{5}$$

$$n^{exp}_{ibin} = \sum_{jproc} \mu_{jproc} n^{exp}_{ibin,jproc} \prod_{ksyst} \kappa^{\theta_{ksyst}}_{ibin,jproc,ksyst} \tag{6}$$

- The above can be computed very efficiently in tensorflow with two tensor contractions (matrix multiplications) plus a small number of element-wise operations

$$\ln r^{syst}_{ibin,jproc} = \ln \prod_{ksyst} \kappa^{\theta_{ksyst}}_{ibin,jproc,ksyst} = \sum_{ksyst} \theta_{ksyst} \ln \kappa_{ibin,jproc,ksyst} \tag{7}$$

$$n^{exp}_{ibin} = \sum_{jproc} \mu_{jproc} n^{exp}_{ibin,jproc} e^{\ln r^{syst}_{ibin,jproc}} \tag{8}$$

(plus some modest additional complexity for asymmetric uncertainties)

Likelihood Construction in TensorFlow

```
276   #matrix encoding effect of nuisance parameters
277   #memory efficient version (do summation together with multiplication in a single tensor contraction step)
278   #this is equivalent to
279   #alpha = tf.reshape(alpha,[-1,1,1])
280   #theta = tf.reshape(theta,[-1,1,1])
281   #logk = logkavg + alpha*logkhalfdiff
282   #logktheta = theta*logk
283   #logsnorm = tf.reduce_sum(logktheta, axis=0)
284
285   mlogk = tf.reshape(logk,[nbinsfull*nproc,2*nsyst])
286   logsnorm = tf.matmul(mlogk,mthetaalpha)
287   logsnorm = tf.reshape(logsnorm,[nbinsfull,nproc])
288
289   snorm = tf.exp(logsnorm)
290
291   #final expected yields per-bin including effect of signal
292   #strengths and nuisance parmeters
293   #memory efficient version (do summation together with multiplication in a single tensor contraction step)
294   #equivalent to (with some reshaping to explicitly match indices)
295   #rnorm = tf.reshape(rnorm,[1,-1])
296   #pnormfull = rnorm*snorm*norm
297   #nexpfull = tf.reduce_sum(pnormfull,axis=-1)
298   snormnorm = snorm*norm
299   nexpfullcentral = tf.matmul(snormnorm, mrnorm)
300   nexpfullcentral = tf.squeeze(nexpfullcentral,-1)
```

github link

- This is the core of the likelihood calculation (calculation of n_{ibin}^{exp})

Likelihood Construction in TensorFlow

```
389 nobsnull = tf.equal(nobs,tf.zeros_like(nobs))
390
391 nexpsafe = tf.where(nobsnull, tf.ones_like(nobs), nexp)
392 lognexp = tf.log(nexpsafe)
393
394 nexpnom = tf.Variable(nexp, trainable=False, name="nexpnom")
395 nexpnomsafe = tf.where(nobsnull, tf.ones_like(nobs), nexpnom)
396 lognexpnom = tf.log(nexpnomsafe)
397
398 #final likelihood computation
399
400 #poisson term
401 lnfull = tf.reduce_sum(-nobs*lognexp + nexp, axis=-1)
402
403 #poisson term with offset to improve numerical precision
404 ln = tf.reduce_sum(-nobs*(lognexp-lognexpnom) + nexp-nexpnom, axis=-1)
405
406 #constraints
407 lc = tf.reduce_sum(constraintweights*0.5*tf.square(theta - theta0))
408
409 l = ln + lc
410 lfull = lnfull + lc
```

github link

- Rest of the likelihood construction is trivial (with some protection for 0-observed bins + offsetting to preserve numerical precision)

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
  - Minuit MIGRAD strategy (add constant to force Hessian positive-definite) is probably suboptimal in such cases

Minimization

- 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
- 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 native tensorflow minimizer 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

Minimizer Implementation

```
110     #n.b. this has a substantially different form from the usual SR 1 update
111     #since we are directly updating the eigenvalue-eigenvector decomposition.
112     #The actual hessian approximation is never stored (but memory requirements
113     #are similar since the full set of eigenvectors is stored)
114     def doSR1Update(ein,UTin,yin,dxin):
115       #compute quantities which define the rank 1 update
116       #and numerical test to determine whether to perform
117       #the update
118       y = tf.reshape(yin,[-1,1])
119       dx = tf.reshape(dxin,[-1,1])
120       ecol = tf.reshape(ein,[-1,1])
121
122       UTdx = tf.matmul(UTin, dx)
123       UTy = tf.matmul(UTin,y)
124       den = tf.matmul(y,dx,transpose_a=True) - tf.matmul(UTdx,ecol*UTdx,transpose_a=True)
125       dyBx =  UTy - ecol*UTdx
126       dyBxnormsq = tf.reduce_sum(tf.square(dyBx))
127       dyBxnorm = tf.sqrt(dyBxnormsq)
128       dxnorm = tf.sqrt(tf.reduce_sum(tf.square(dx)))
129       dennorm = dxnorm*dyBxnorm
130       absden = tf.abs(den)
131       dentest = tf.less(absden,1e-8*dennorm) | tf.equal(tf.reshape(absden,[]),0.)
132       dentest = tf.reshape(dentest,[])
133       dentest = tf.logical_or(dentest,tf.equal(actual_reduction,0.))
134
135
136       def doUpdate():
137         #compute update in the form
138         #B->B + rho zz^T with |z|=1
139         z = dyBx/dyBxnorm
140         signedrho = dyBxnormsq/den
141         signedrho = tf.reshape(signedrho,[])
```

github link

- The next block of code continues on for much longer to implement the SR1 update of the eigendecomposition
- This is not the most efficient algorithm in terms of flops, but vector-parallelizes extremely well

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 with SciPy

	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)

Some Performance Tests: Native TF 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

Performance Tests (pfor)

(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

Another Important Technical Detail

- Running the same kind of likelihood fit to extract directly the double-differential cross section for the charged lepton kinematics has slightly different challenges:
  - 648 measured cross sections (POI's)
  - 655 total processes
  - 2448 bins
  - 1051 systematics/nuisance parameters
- (Slightly different binning and fewer theory nuisances compared to helicity fit)
- nbins $\times$ nproc $\times$ nsyst is **much** larger, but signal templates are actually very sparse due to nearly diagonal response matrix
- Full implementation of likelihood construction using sparse tensors

Sparse Tensor Likelihood Construction

```
251 if sparse:
252   logsnorm = simple_sparse_tensor_dense_matmul(logk_sparse,mthetaalpha)
253   logsnorm = tf.squeeze(logsnorm,-1)
254   snorm = tf.exp(logsnorm)
255
256   snormnorm_sparse = SimpleSparseTensor(norm_sparse.indices, snorm*norm_sparse.values, norm_sparse.dense_shape)
257   nexpfullcentral = simple_sparse_tensor_dense_matmul(snormnorm_sparse,mrnorm)
258   nexpfullcentral = tf.squeeze(nexpfullcentral,-1)
259
```

github link

- This is the equivalent calculation of n^{exp}_{ibin} with sparse tensors
- SimpleSparseTensor and simple_sparse_tensor_dense_matmul are thin wrappers around standard TF implementations to allow int32 indices (to save memory)

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

- Intermediate configuration with 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)

tf.data implementation for large constants from HDF5

```python
def maketensor(h5dset):

  ndims = len(h5dset.shape)
  if ndims>1:
    raise Exception("Only flat tensors are supported")

  if 'original_shape' in h5dset.attrs:
    shape = h5dset.attrs['original_shape']
  else:
    shape = h5dset.shape

  if h5dset.size == 0:
    return tf.zeros(shape,h5dset.dtype)

  chunksize = h5dset.chunks[0]
  chunkiter = range(0,h5dset.size,chunksize)

  def readChunk(i):
    ielem = chunkiter[i]
    aout = h5dset[ielem:ielem+chunksize]
    if not aout.shape == h5dset.chunks:
      aout.resize(h5dset.chunks)
    return aout

  #calculate number of chunks
  nchunks = len(chunkiter)

  #create tf Dataset which reads one chunk at a time.  The last chunk may need to be padded to match the chunk size.
  #Chunks are then batched together if necessary, and then the padded part is sliced out if necessary
  #n.b. map_and_batch is used instead of the simple dataset batch function because in tf 1.6 this apparently avoids an
  #extra copy in memory

  dset = tf.data.Dataset.range(nchunks)
  dset = dset.map(lambda x: tf.reshape(tf.py_func(readChunk,[x],tf.as_dtype(h5dset.dtype)),h5dset.chunks))
  if nchunks>1:
    dset = dset.apply(tf.contrib.data.map_and_batch(lambda x: x, nchunks))
    if not h5dset.shape[0]%h5dset.chunks[0] == 0:
      paddedshape = (nchunks*chunksize,)
      dset = dset.map(lambda x: tf.reshape(x,paddedshape)[:h5dset.shape[0]])

  dset = dset.map(lambda x: tf.reshape(x,shape))
  dset = dset.cache().repeat(-1)
  atensor = dset.make_one_shot_iterator().get_next()
  return atensor
```

- Read the flattened array in chunks
- Reshape
- batch chunks
- pad and slice last chunk as necessary
- Everything is designed to avoid more than one copy of the full array in memory

github link

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/text2hdf5.py**: Create inputs to 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)

Implementation

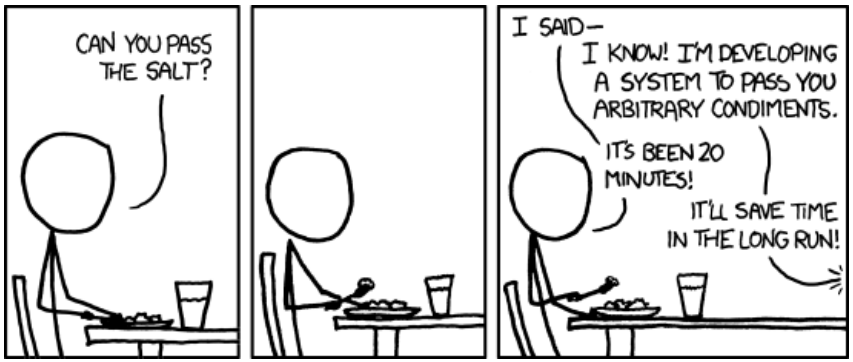
- To first order:

```
#convert txt+root datacard to hdf5 with properly format
text2hdf5.py mydatacard.txt
#run single fit to observed data
combinetf.py mydatacard.hdf5
```

- e.g. with simple example card: `https://github.com/bendavid/HiggsAnalysis-CombinedLimit/blob/dfde59264abe7f5181aec5ff62a3dae7a5ae84ee/data/tutorials/shapes/simple-shapes-TH1.txt`

Making our way back to physics...

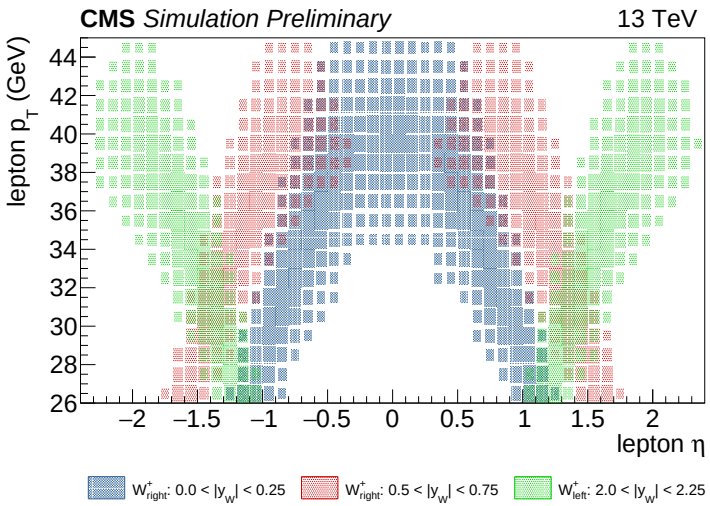
- TLDR: Full fit runs in a few minutes on a 16 core CPU, even faster on GPU
- Current covariance matrix calculation is not fully optimized, takes about 30 mins (explicit tensorflow loop over rows of the hessian)
- Tensorflow pfor (or even migration to Jax) can help substantially with this

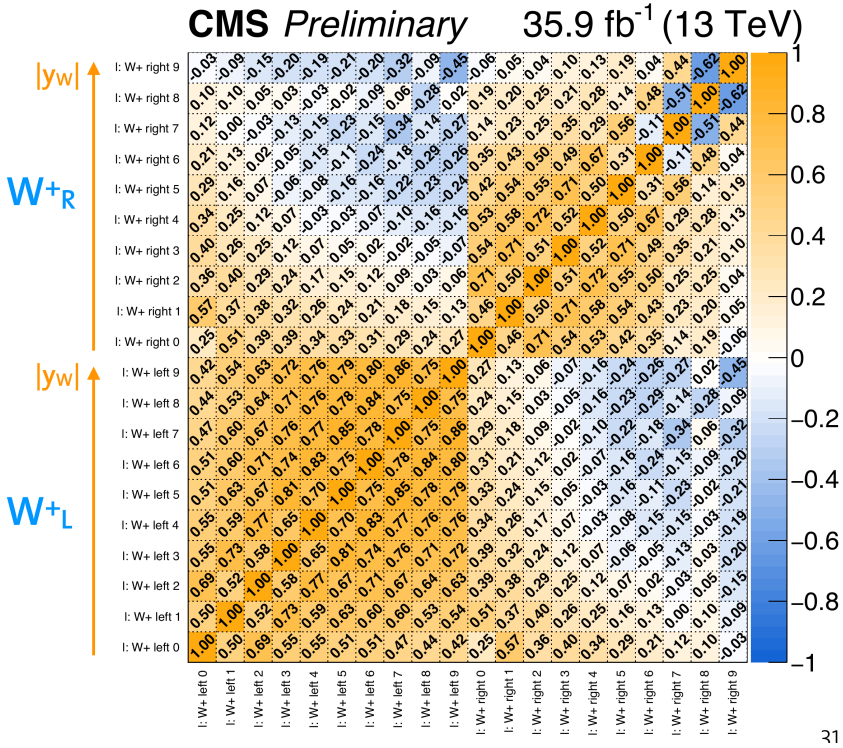


Making our way back to physics...

- POI's in fit are expressed in terms of signal multipliers μ a la Higgs combination fit
- In order to convert back to measured cross sections:
  - Predicted cross section from MC is kept track of (in an additional "masked bin" which does not contribute to the likelihood)
  - Systematic uncertainties which modify the cross section are properly accounted for (e.g. the theory uncertainties)
  - Post-fit cross section is consistently calculated as $\hat{\mu}\sigma(\hat{\bar{\theta}})$ for post-fit values of signal multiplier μ and relevant nuisance parameters $\bar{\theta}$
  - Post fit covariance matrix for measured cross sections calculated using covariance matrix for $(\bar{\mu}, \bar{\theta})$ together with jacobian (all constructed automatically by back-propagation in tensorflow)
- The same "trick" can be used to construct on the fly the full set of normalized cross sections, charge asymmetries, integrated or partially integrated cross sections

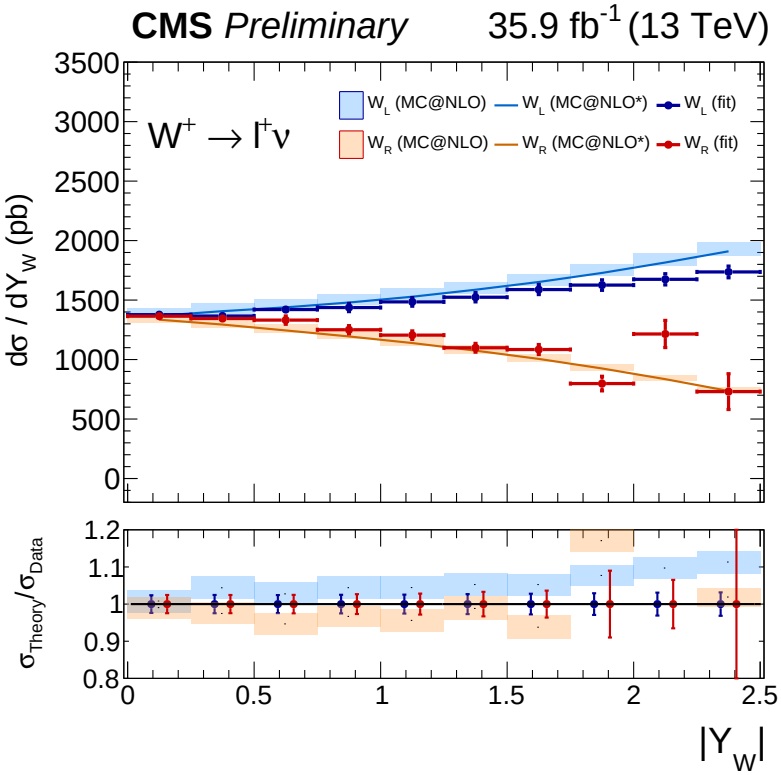
Results: Helicity/Rapidity

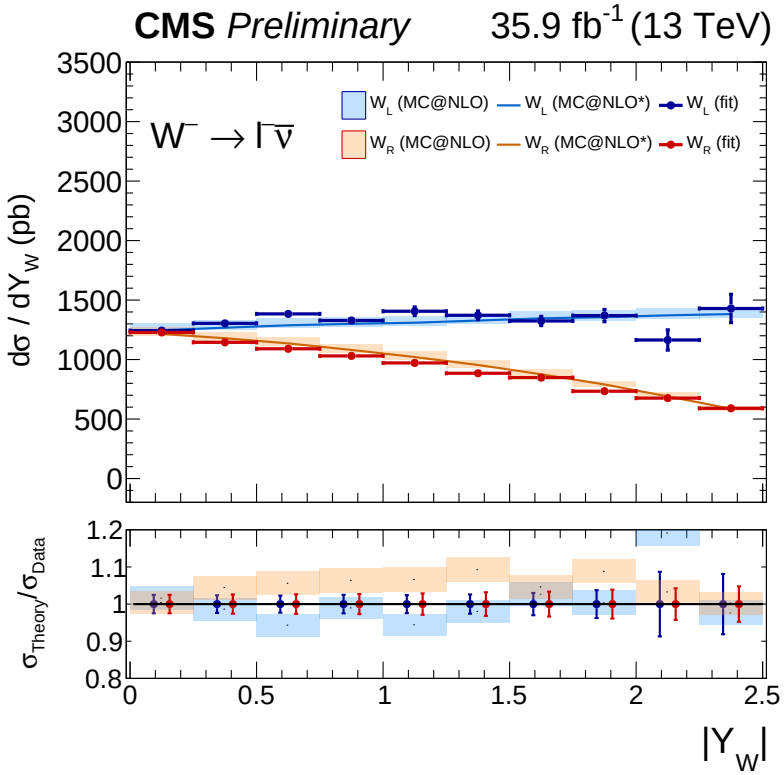




- Overlapping templates produce anti-correlations between neighbouring bins in the helicity fit
- Subset of correlation matrix shown here
- Competition between statistical anti-correlations from neighbouring bins and correlated systematic uncertainties
- The "full" covariance matrix including systematic uncertainties is 1394 x 1394

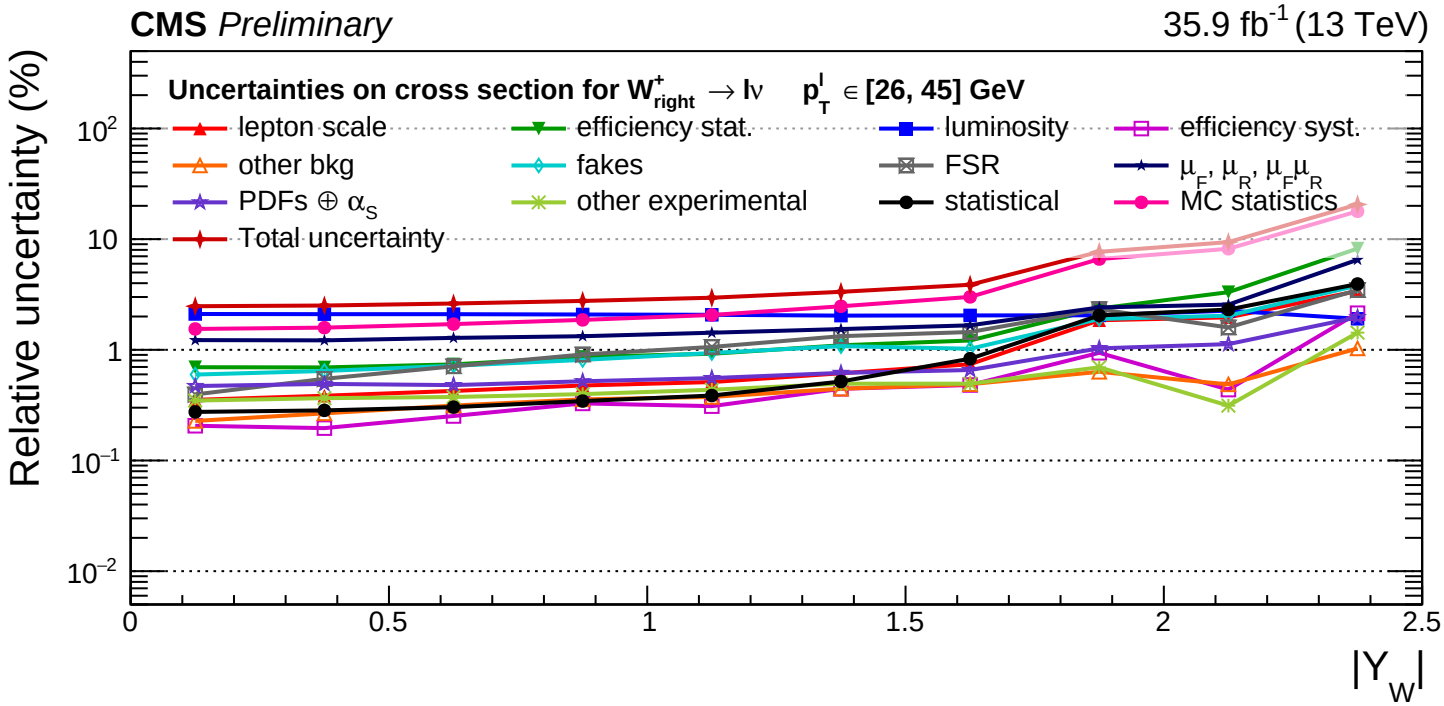
Results: Helicity/Rapidity: Polarized Cross Sections





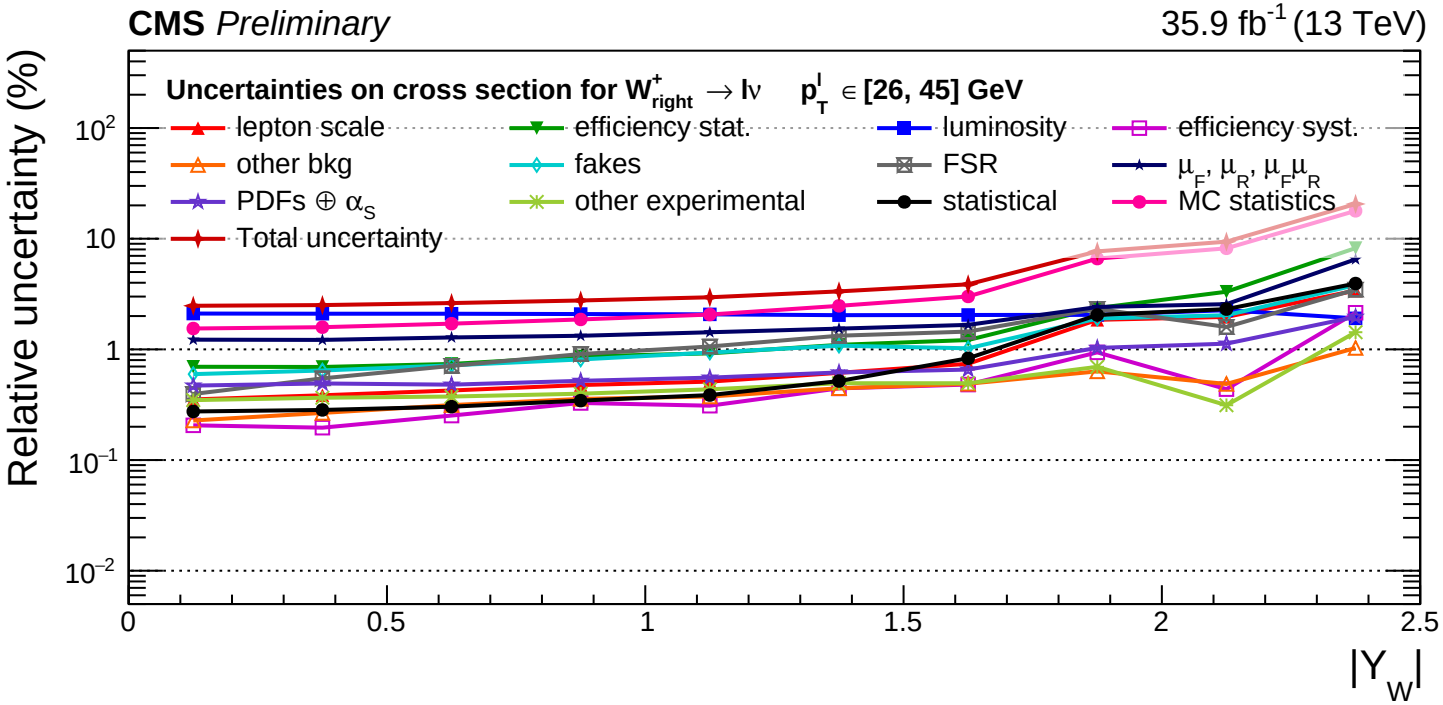
- Correlated component of the uncertainty dominated by luminosity
- "oscillations" due to statistical (+MC stat) anti-correlations

Uncertainty Breakdown : Impacts



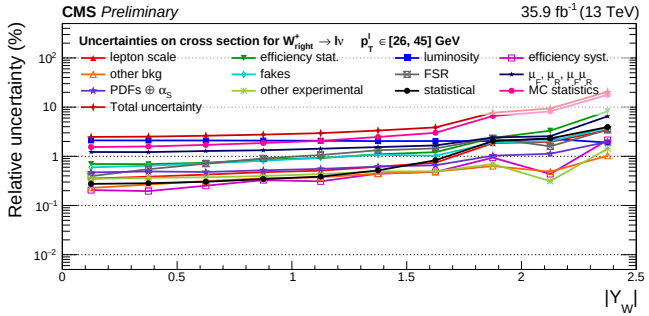
- Correlated component of the uncertainty dominated by luminosity
- For this particular helicity-charge combination, data/MC statistical uncertainties become relevant at high rapidity (falling cross section and reduced acceptance from forward lepton emission)

Uncertainty Breakdown : Impacts: Technical Details



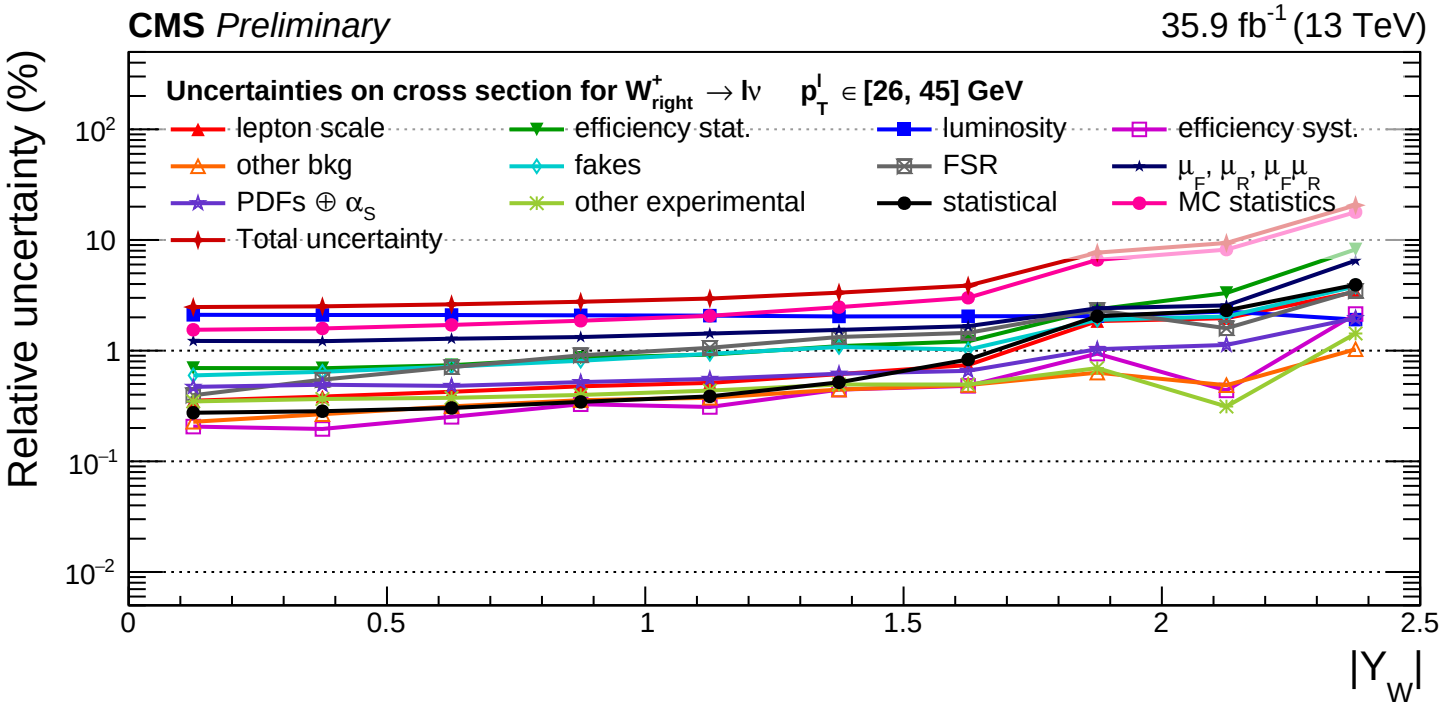
- Procedure for defining uncertainty breakdown for profile likelihood fits not uniquely defined
- "statistical" impact is stat-only uncertainty (covariance matrix with no systematics, but post-fit nuisance values)
- "MC Statistics" evaluated with Barlow-Beeston lite procedure as part of likelihood construction, impact taken as difference in quadrature (including systematics) with vs without MC stat uncertainties

Uncertainty Breakdown : Impacts: Technical Details



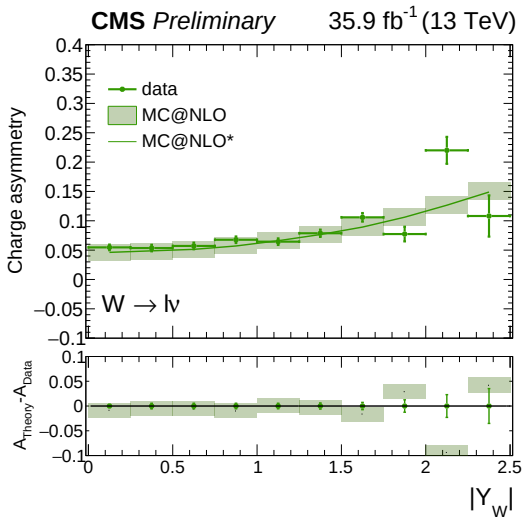
- "Impact" for a single nuisance parameter defined as Cov(nuisance,POI)/σ(nuisance) from post-fit covariance matrix
- In the Gaussian limit, this is the equivalent of moving a nuisance parameter by $\pm 1\sigma$ of its post-fit uncertainty, and re-running the fit with everything else profiled (this is how impacts for single nuisances are defined in the Higgs combination)
- In the presence of correlations, the impacts do not necessarily sum in quadrature to the total uncertainty
- An alternative would be to successively add systematic contributions (this guarantees uncertainties sum in quadrature to the total, but depends on an arbitrary ordering of uncertainties)

Uncertainty Breakdown : Impacts: Technical Details

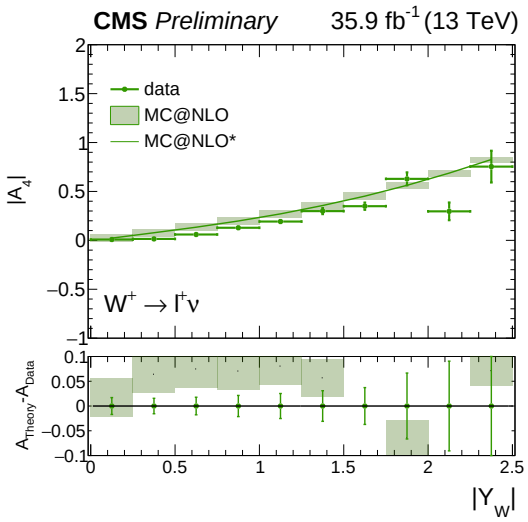


- Impacts are generalized to groups of nuisances by diagonalizing nuisances within a group from the post-fit covariance and computing total impact
- If strongly correlated uncertainties are grouped together, this reduces the effect of correlations on the breakdown
- Impacts may still not sum in quadrature to total due to residual correlations between groups

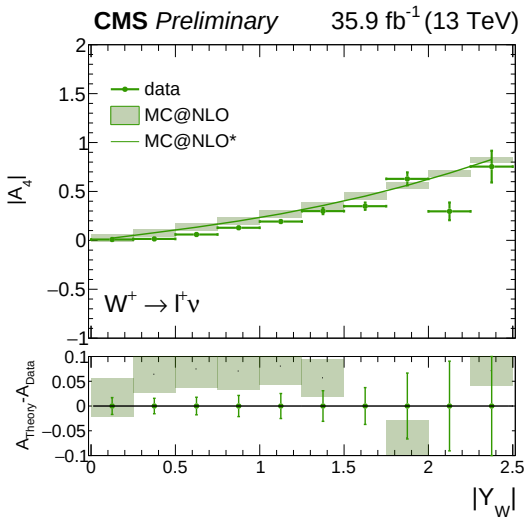
Results: Helicity/Rapidity: Asymmetries



(a) Unpol. Charge Asym.



(b) A_4^+



(c) A_4^-

- For charge and polarization asymmetries, correlated experimental systematics largely cancel, and predictions become dominated by PDF uncertainties
- Constraining power for PDFs (and possible tensions) become manifest
- Charge asymmetry is comparable to W charge asymmetry from Tevatron, but again, relevant helicity fractions are freely profiled

Conclusions

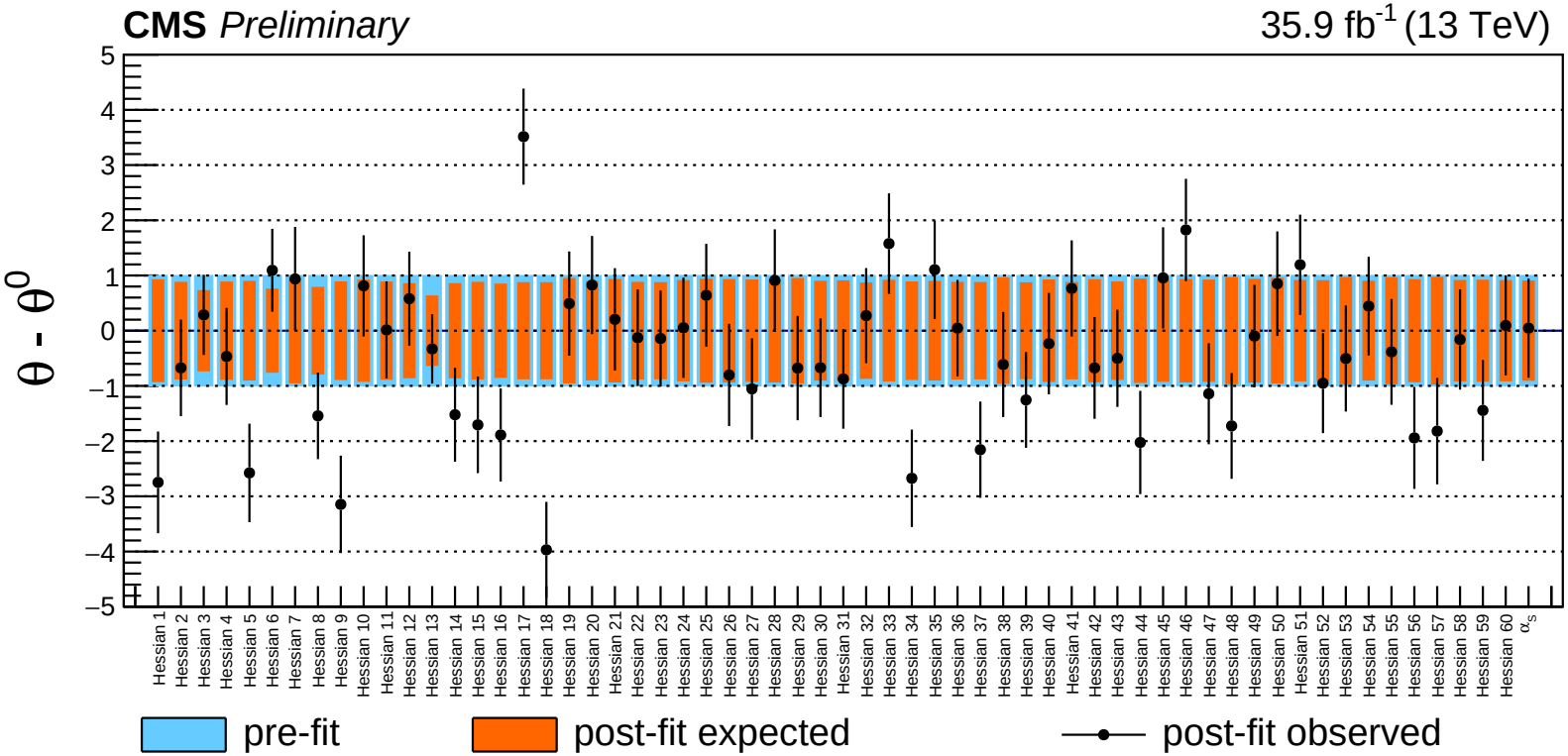
- W helicity measurement provides polarized differential cross section measurements of W production, with significant potential for PDF constraints, and is an important precursor for future m_W measurements
- Analysis is designed to allow in-situ constraints of PDFs with maximum statistical power considering experimental and other theory uncertainties
- Statistical interpretation required the development of new tools and algorithms based on TensorFlow (also used by some other CMS analyses)
- Substantial gains in speed and numerical stability with respect to traditional tools
- Enables more sophisticated and accurate modelling of systematic uncertainties which are crucial for future measurements

Backup

Assessing Potential for PDF constraints

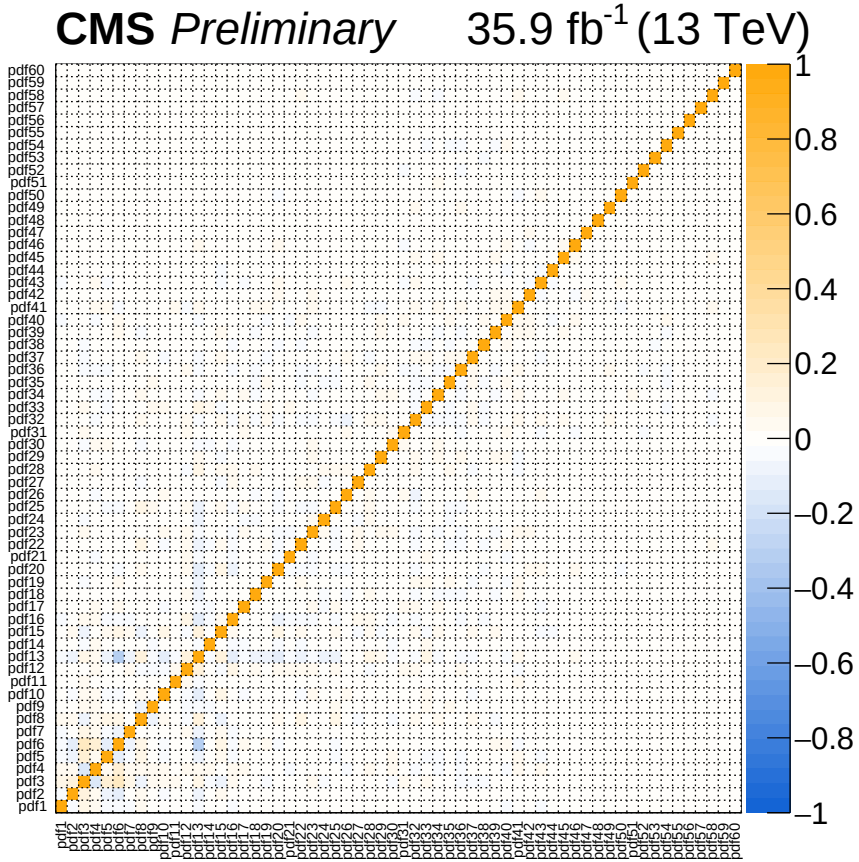
- PDF uncertainties enter measured cross sections only as very small residual acceptance effects, however they are fully implemented as part of the fit
- Can re-run the likelihood fit with cross sections fixed to prediction within theory uncertainties
- Nuisances associated with PDF uncertainties can then be considered the parameters of interest
- Technically this is done in the context of the helicity/rapidity fit, which has slightly more conservative theory uncertainties due to the de-correlation of scale variations by helicity
- This is closely related to PDF profiling from unfolded cross sections, eg. in ATLAS 7TeV W/Z paper, with some important differences:
  - Profiling is done at **detector level**
  - QCD accuracy is (in this case) limited to NLO
  - Re-summation corrections are included through in this case MC@NLO matching to parton shower
  - PDF sets are limited to weights available in MC (in this case only NNPDF 3.0 NLO)
- Standard caveats about interpretation of results far from initial values still apply

Assessing Potential for PDF constraints



- Significant constraints on some eigenvectors, and some significant pulls/tension
- $\chi^2/\text{dof} = 117/61$

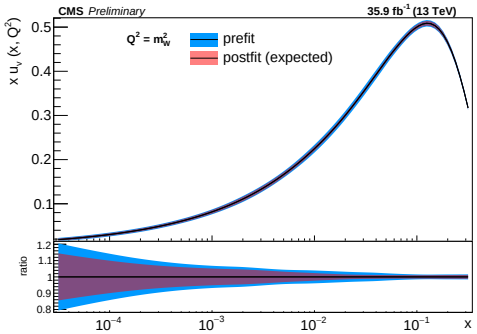
Assessing Potential for PDF constraints



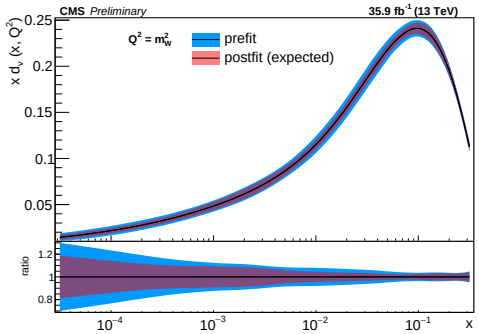
- Correlation matrix doesn't look very interesting at first glance, but strongest anti-correlations correspond to the two most constrained eigenvectors, and this is relevant

Assessing Potential for PDF constraints (expected)

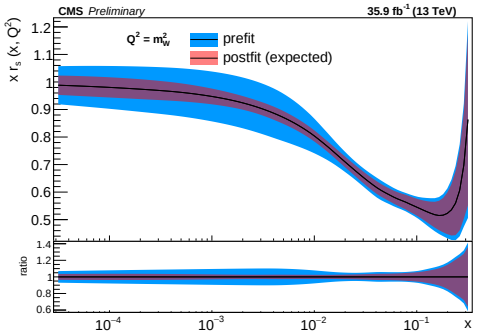
- Post-fit nuisance values and covariance matrix can be used to recompute central value and uncertainties for parton distributions in LHAPDF with dramatic expected constraints



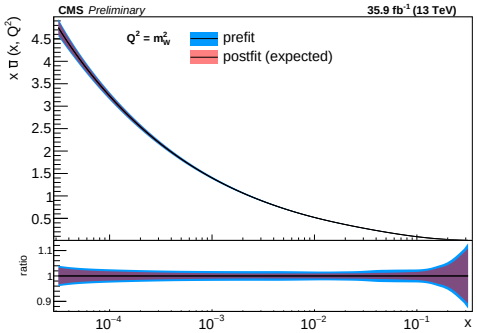
(a) u_V



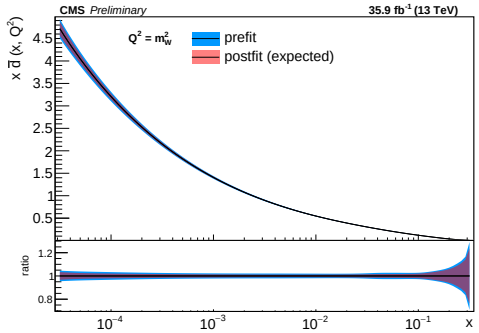
(b) d_V



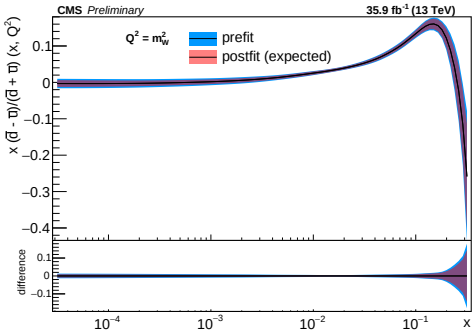
(c) $r_s = (s + \bar{s})/(\bar{u} + \bar{d})$



(d) $\bar{u}$

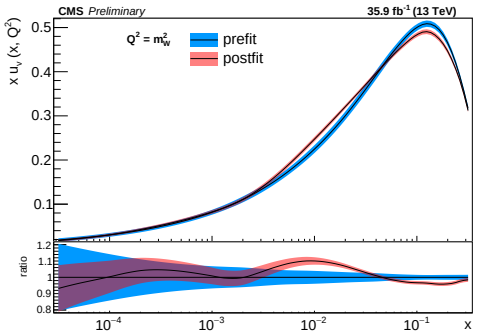


(e) $\bar{d}$

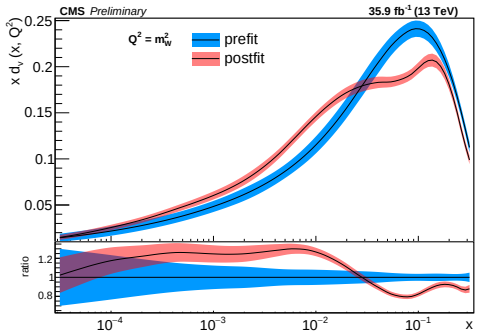


(f) $(\bar{d} - \bar{u})/(\bar{d} + \bar{u})$

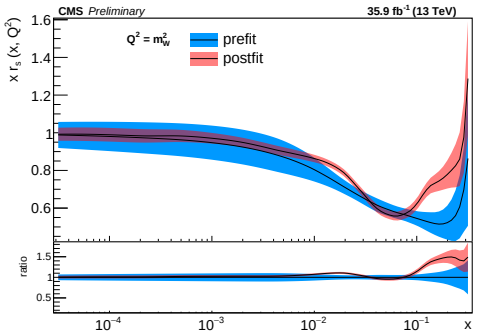
Assessing Potential for PDF constraints (observed)



(a) u_V



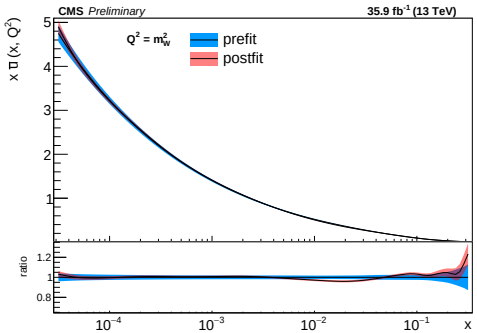
(b) d_V



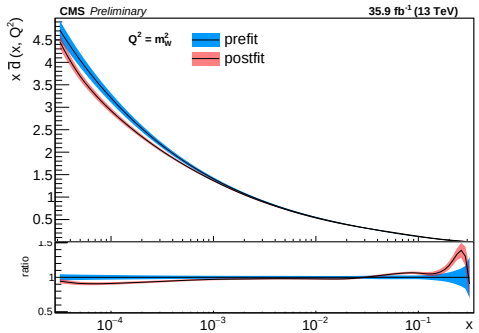
(c) $r_s = (s + \bar{s})/(\bar{u} + \bar{d})$

- Oscillations in part related to anti-correlations between different x values arising from the statistical uncertainties in the measurement
- Still clear tensions with the input PDF set
- Can be related to limitations of profiling at this level of constraint, limitations of NLO accuracy, limitations of the input PDF set/underlying parameterization (e.g. NNPDF3.0 uses perturbatively generated charm, but NNPDF3.1 fits the charm distribution from data)

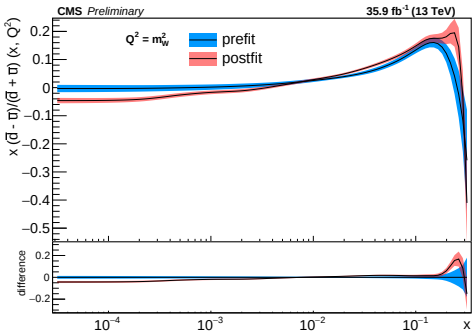
Assessing Potential for PDF constraints (observed)



(a) $\bar{u}$



(b) $\bar{d}$



(c) $(\bar{d} - \bar{u})/(\bar{d} + \bar{u})$

- Oscillations in part related to anti-correlations between different x values arising from the statistical uncertainties in the measurement
- Still clear tensions with the input PDF set
- Can be related to limitations of profiling at this level of constraint, limitations of NLO accuracy, limitations of the input PDF set/underlying parameterization (e.g. NNPDF3.0 uses perturbatively generated charm, but NNPDF3.1 fits the charm distribution from data)

Assessing Potential for PDF constraints

- A much more detailed study of the implications of this data for PDFs is clearly needed (NNLO accuracy, wider range of PDF sets, full QCD analysis, etc)
- Some non-trivial challenges in producing suitable predictions
- Releasing results from detector level profiling allows a comparison point when interpreting unfolded cross sections in the future