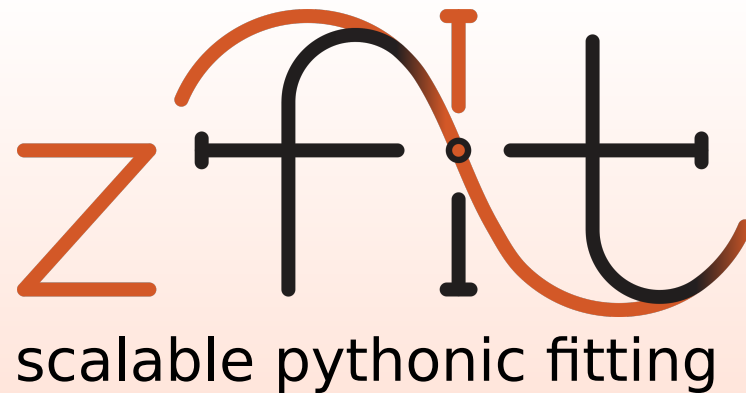


Moving to TF 2.x

PyHEP 2020



Jonas Eschle
Jonas.Eschle@cern.ch



SWISS NATIONAL SCIENCE FOUNDATION



**University of
Zurich**^{UZH}

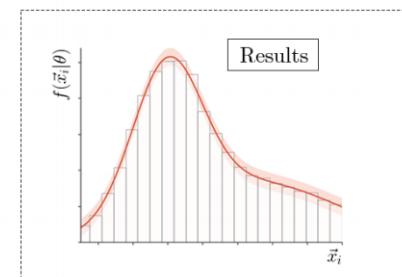
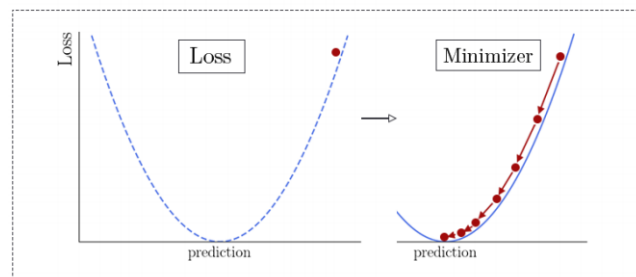
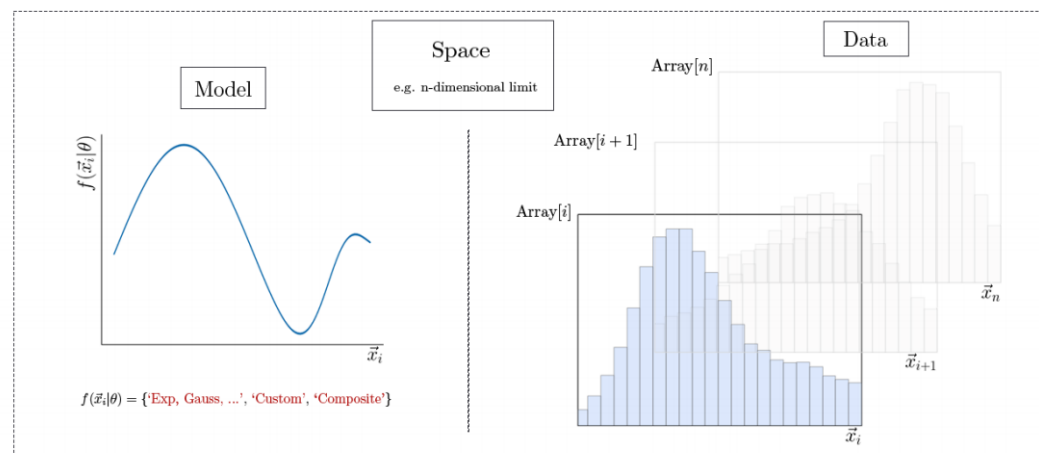
What is zfit



- Model fitting library in pure Python with well defined workflow
- Strong custom model building capability

```
from zfit import z # or directly TensorFlow
```

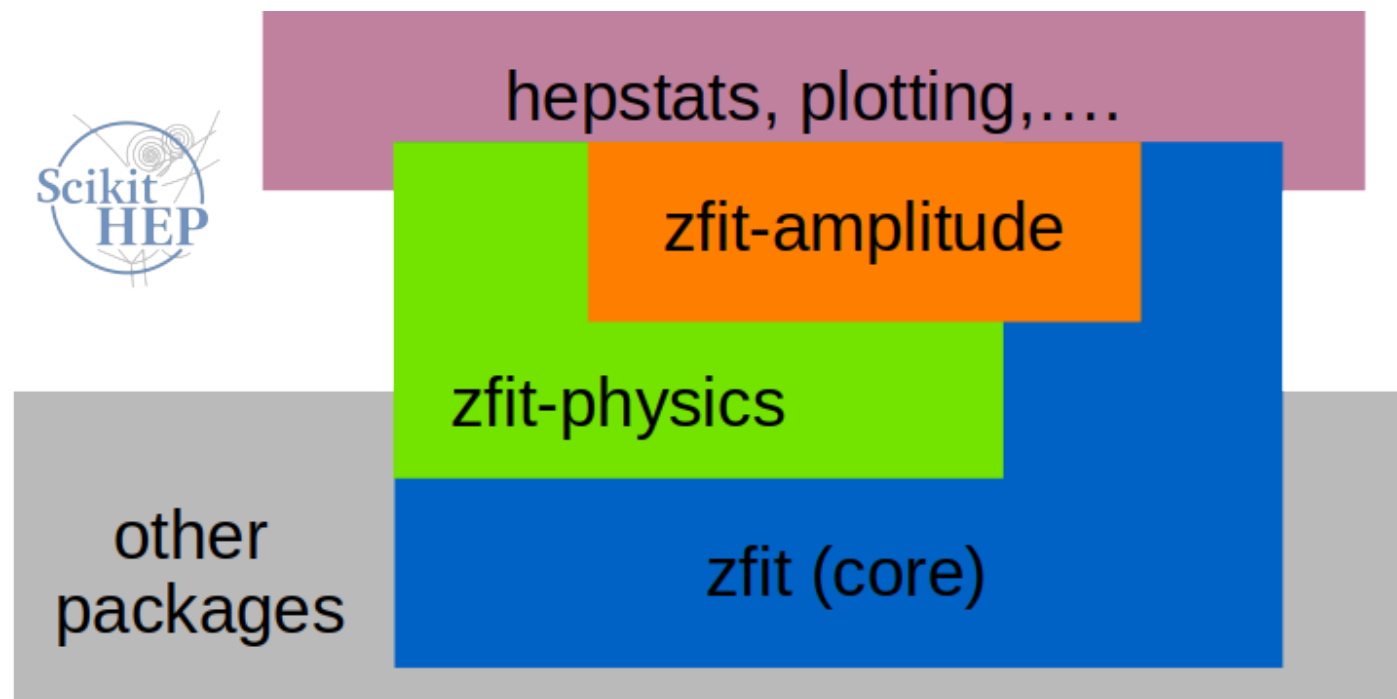
```
class CustomPDF(zfit.pdf.ZPDF):  
    _PARAMS = ['alpha']  
  
    def _unnormalized_pdf(self, x):  
        data = x.unstack x()  
        alpha = self.params['alpha']  
  
        return z.exp(alpha * data)
```



What is zfit: ecosystem





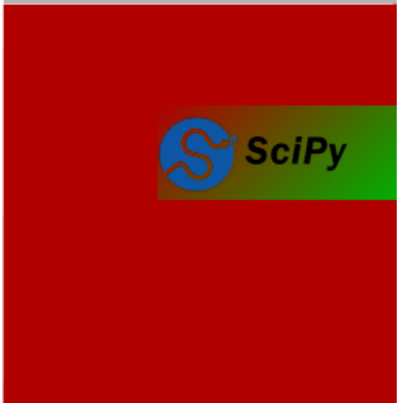





- Strongly embedded into the ecosystem (Scikit-HEP)
- Limited scope:
 - model fitting
 - sampling



What is zfit: backend

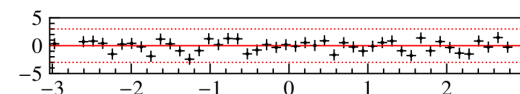
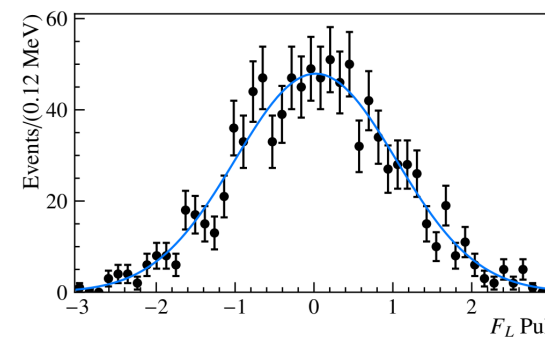
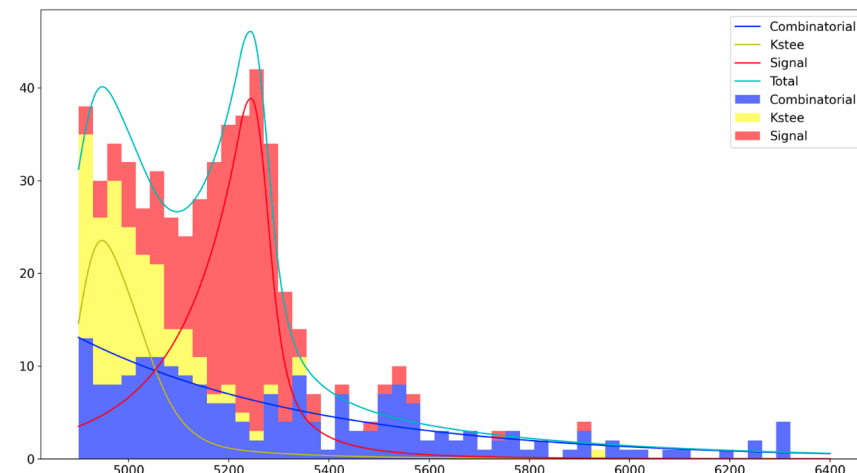


- Uses TensorFlow as the computational backend

	C++ library (RooFit,...)	Numpy based		
HEP specific content/API				
Models				TF Probability
Gradients				
Computational optimizations				
Parallelization/GPU				
Low level handling				

Changes in zfit

- New shapes:
 - ARGUS,
 - Cauchy
 - KDE
- Improved fitting stability and toys
- Arbitrary spaces:
 - Can be combined
 - Arbitrary function, test and filter if inside



Main improvement



- TensorFlow 2.0
- numerical gradients
- Smart JIT compilation cache

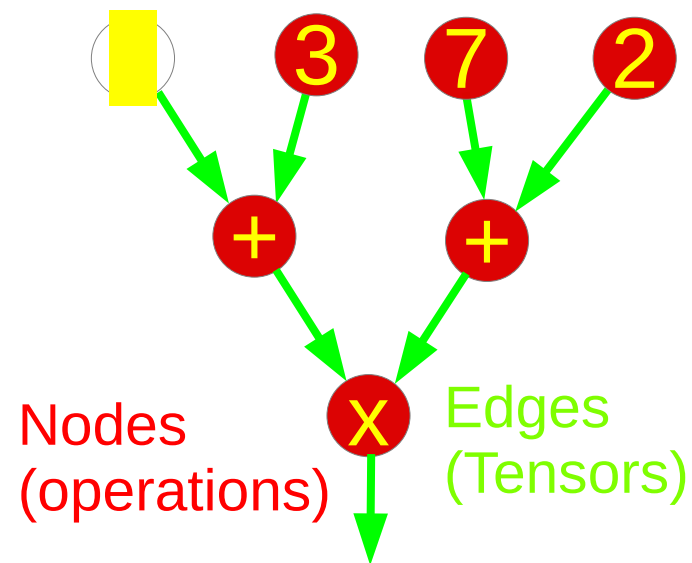
full possible integration into Python & Numpy

Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
for ... in ...:
    rnd = tf.random(...) ← adds new op to graph every time
    rnd_np = run(rnd)
    ...
```



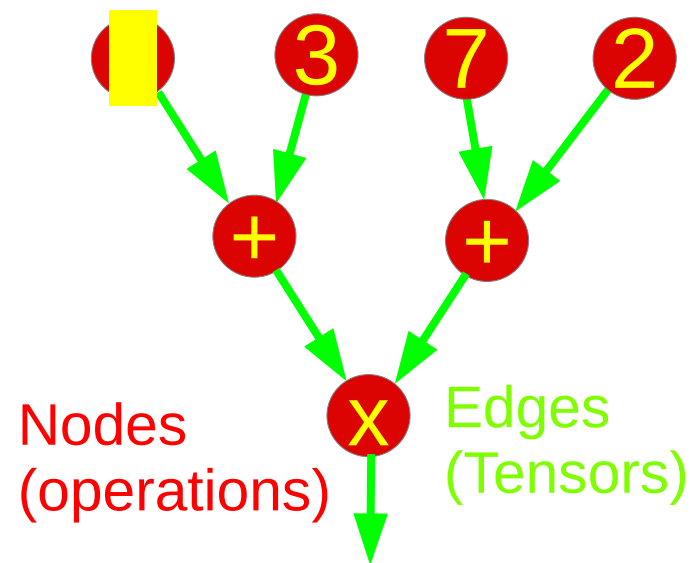
Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
for ... in ...:
    rnd = tf.random(...)
    rnd_np = run(rnd)
    ...
```

← adds new op to graph every time

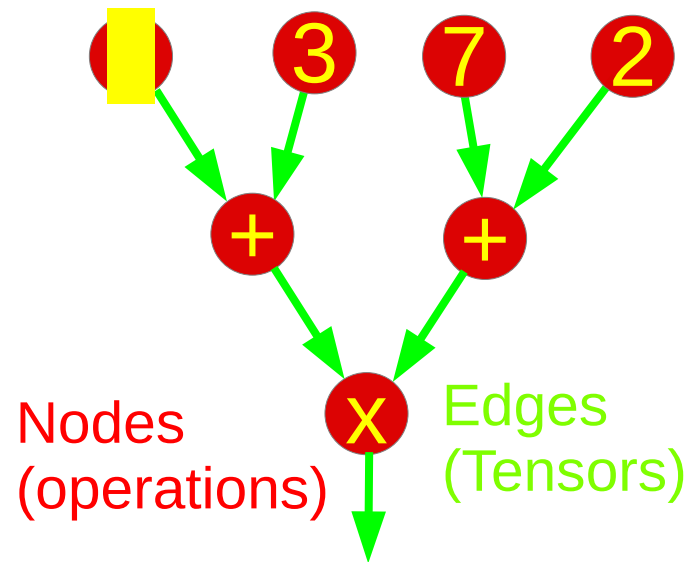


Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
rnd = tf.random(...)
for ... in ...:
    rnd_np = run(rnd)
    ...
```



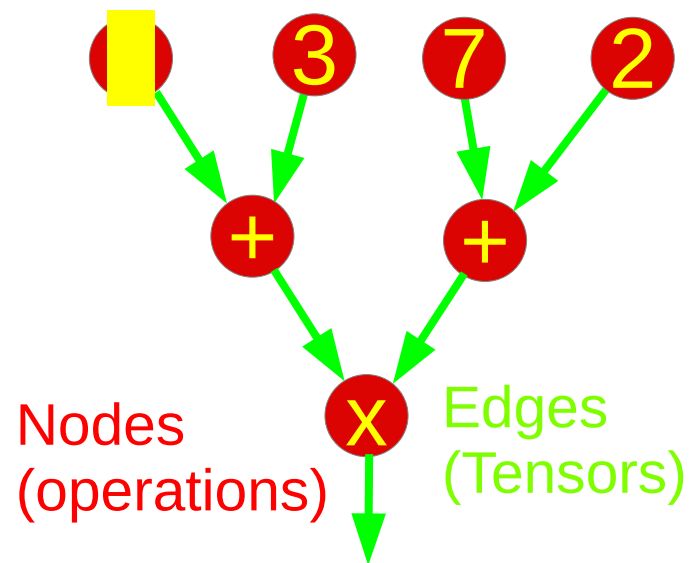
Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
rnd = tf.random(...)
for ... in ...:
    rnd_np = run(rnd)
    ...
```

But inputs? → placeholders!



Declarative: *define* computations, execute when needed

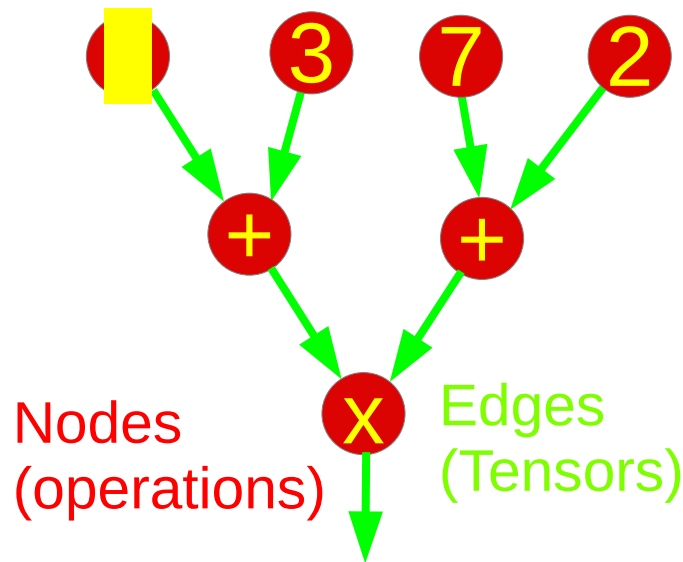
```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
rnd = tf.random(...)
for ... in ...:
    rnd_np = run(rnd)
    ...
```

But inputs? → placeholders!*

*no, actually one of the other four methods!



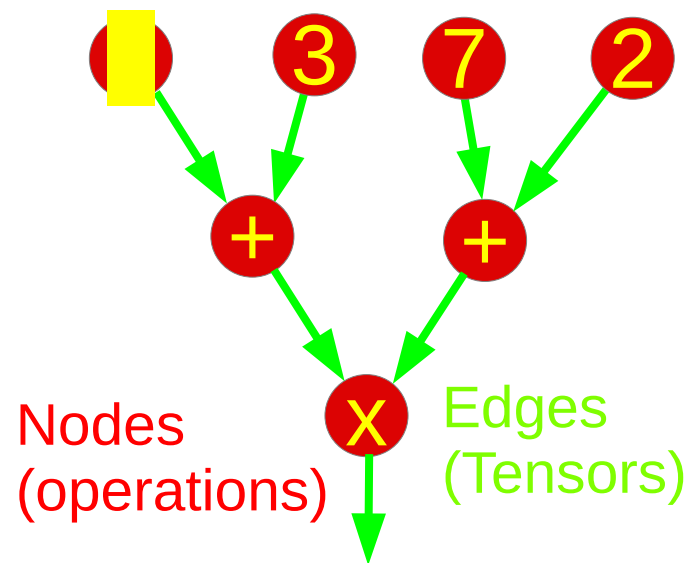
Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

```
rnd = tf.random(...)
for ... in ...:
    rnd_np = run(rnd)
...
```

Did I mention Sessions?



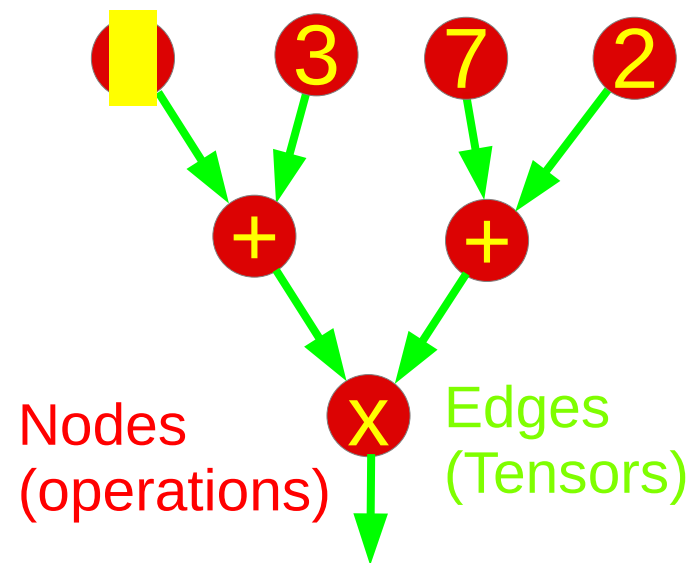
Declarative: *define* computations, execute when needed

```
sum_1 = tf.constant(5) + tf.constant(3)
sum_2 = tf.constant(7) + tf.constant(2)
product = sum_1 * sum_2
```

Product is a symbolic Tensor, we have to "execute" it!

Summarized:

object, that when executed, takes input, performs calculations and returns output => **a function**



"After years of abusive, around-the-corner, graph-thinking, we forgot how easy things can be..."

Anonymous

why TF2.0 *feels* wrong but is actually right

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

*Your
symbolically
defined
model*

We will solely look at the low level functionality

TL;DR: like Numpy with a JIT like Numba

- By default, behaves like Numpy (but has GPU + CPU kernels)

```
def add_log(x, y):  
    x_sq = tf.math.log(x)  
    y_sq = tf.math.log(y)  
    return x_sq + y_sq
```

Eager mode



- Quite young (not even a year)
- Numpy-like speed or more for larger data
- Experimental: dispatch calls asynchronously
 - Continue execution until result is needed (like PyTorch)
- Allows to replace backend easily
(but when it's clearly needed, this does not come for free)

TL;DR: like Numpy with a JIT (*did someone say Numba?*)

- By default, behaves like Numpy (but has GPU + CPU kernels)
- TensorFlow operations can be compiled together
- Python is only static and doesn't appear in the computation (*autograph later*)

```
def add_log(x, y):  
    x_sq = tf.math.log(x)  
    y_sq = tf.math.log(y)  
    return x_sq + y_sq
```

```
@tf.function(autograph=False)  
def add_log(x, y):  
    x_sq = tf.math.log(x)  
    y_sq = tf.math.log(y)  
    return x_sq + y_sq
```

- Calling the function:
 - "Replaces input by symbolic Tensor"
 - Trace function (record tf.* calls), build graph, optimize
 - Cache graph remembering the input
 - Signature: dtype, (shape); for python objects equality
- Numpy + Numba without Python compilation
=> "no black magic"

```
@tf.function(autograph=False)
def add_log(x, y):
    x_sq = tf.math.log(x)
    y_sq = tf.math.log(y)
    return x_sq + y_sq
```

- Knowledge about dtype, shape, dependencies
- Grappler: graph optimizer
(constant folding, common subexpression elimination, arithmetic simplification,...)
- Parallelization
- Stitches together TF calls

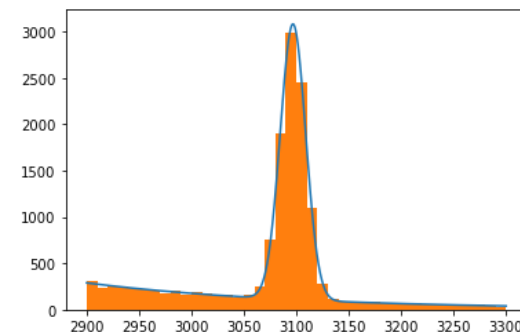
- TF self-contained ecosystem: can use full TF syntax, no Python for dynamic code (but full Python for static)
- TF well designed API, optimized for graphs
- TF lacks good support of converting Python code
- Data processing, conditionals → Numpy & Numba
- Heavy, well defined computing → TensorFlow

zfit benefits



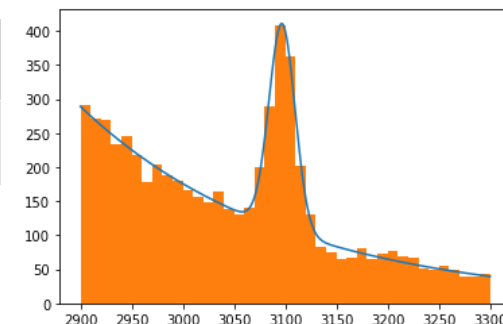
- Thanks to graph supports C++ like speed
- Eager allows to have full Python dynamics

=> the best from two worlds in one



- Example: ~50k events simultaneous fit, background and signal

Library	probit	zfit eager	RooFit	zfit graph
Time (sec)	~ 24	~ 10	~ 0.5	~1.5 (+0.5)



... then why TF?

- No Python compilation and only TF tracing quite powerful
- Very consistent ecosystem!
- Takes care of many low level optimizations and parallelization and a lot of ongoing work there!
- Very well supported in industry