statistics stuff

want pyhf to be backend agnostic

```
pyhf.tensorlib.sum(...)
```

shim around

```
torch.sum(..)
np.sum(...)
tf.reduce_sum(...)
```

Similarly moving to

```
pyhf.probability.Poisson(rate = tensor)
```

shim around

```
torch.distributions.Poisson(...)
tfp.distributions.Poisson(...)
scipy.stats.poisson(...)
```

tfp and torch.distr have identical API, broadcast semantics etc..

Role of shims:

- **normalize APIs**
- **decouple rest of code from concrete backend**

NumPy PyTorch TensorFlow

pyhf.probability

- depends on agnostic tensorlib to define basic types / API signatures

```
pdf = PdfClass(par_tensor)
logL = pdf.log_prob(par_tensor)
```

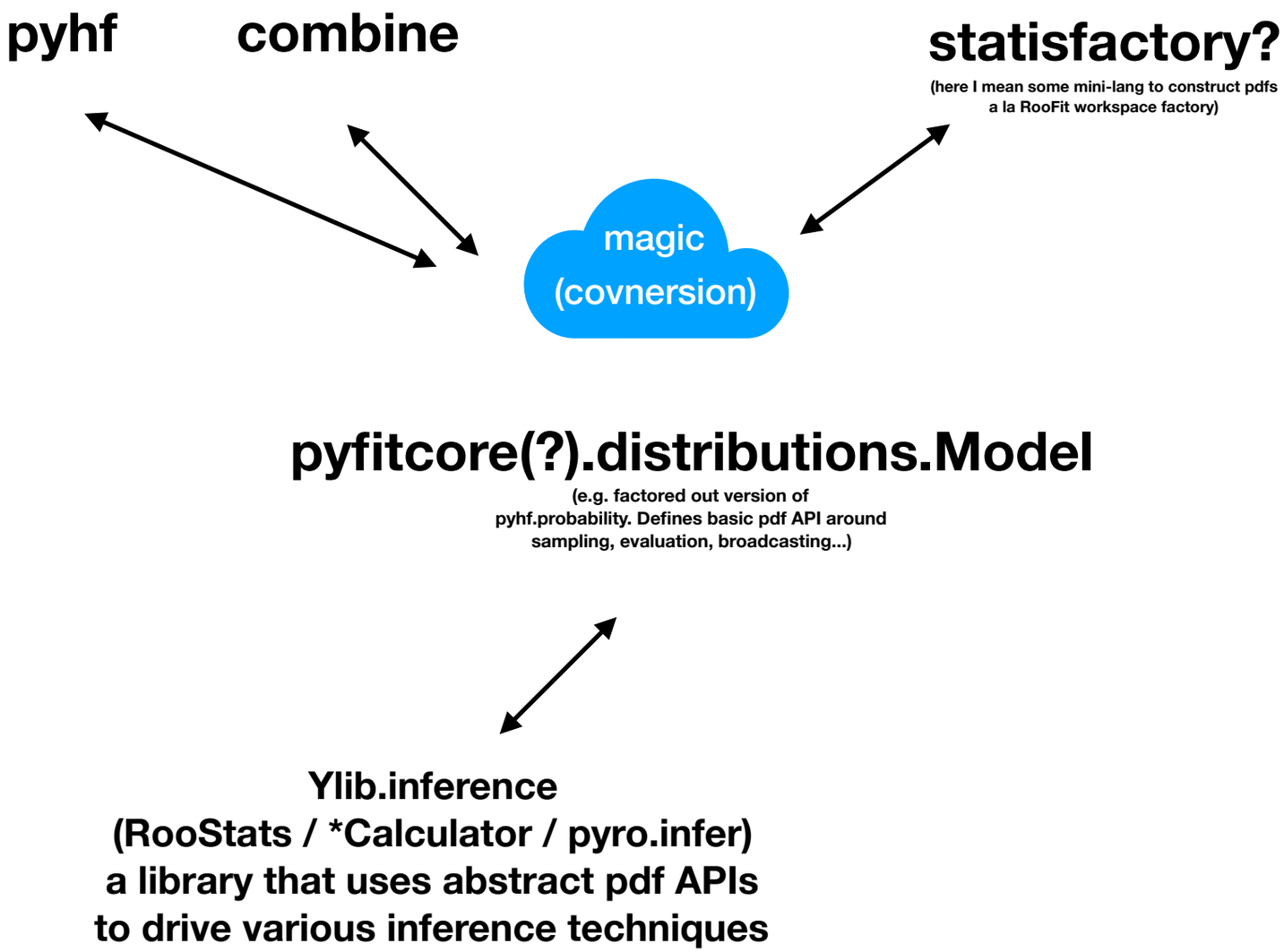
Simultaneous (~RooSimPdf)

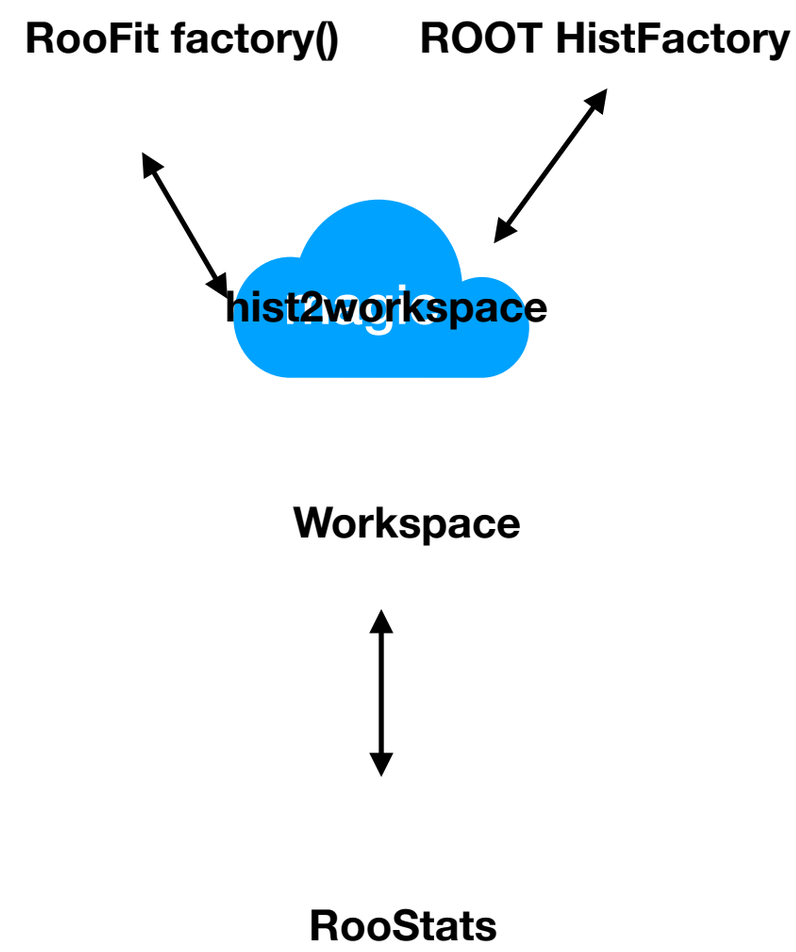
Could imagine abstracting from low-level tensors (ModelConfig discussion)

- simple wrappers (torch.nn.Parameters)
- more abstract (a la RooDataset, RooStats::ModelConfig)

```python
from . import get_backend
from .tensor.common import TensorViewer


class ForwardMixin(object):
    def log_prob(self, value):
        return self._pdf.log_prob(value)

    def expected_data(self):
        return self._pdf.expected_data()

    def sample(self, sample_shape=()):
        return self._pdf.sample(sample_shape)


class Poisson(ForwardMixin):
    def __init__(self, rate):
        tensorlib, _ = get_backend()
        self.lam = tensorlib.astensor(rate)
        self._pdf = tensorlib.poisson_pdfcls(rate)

    def expected_data(self):
        return self.lam


class Normal(ForwardMixin):
    def __init__(self, loc, scale):
        tensorlib, _ = get_backend()
        self.mu = tensorlib.astensor(loc)
        self.sigma = tensorlib.astensor(scale)
        self._pdf = tensorlib.normal_pdfcls(loc, scale)

    def expected_data(self):
        return self.mu


class Independent(object):
    '''
    A probability density corresponding to the joint
    likelihood of a batch of identically distributed random
     numbers.
    '''

    def __init__(self, batched_pdf, batch_size=None):
        self.batch_size = batch_size
        self._pdf = batched_pdf

    def expected_data(self):
        return self._pdf.expected_data()

    def sample(self, sample_shape=()):
        return self._pdf.sample(sample_shape)

    def log_prob(self, value):
        tensorlib, _ = get_backend()
        result = self._pdf.log_prob(value)
        result = tensorlib.sum(result, axis=-1)
        return result


class Simultaneous(object):
    def __init__(self, pdfobjs, indices):
        self.tv = TensorViewer(indices)
        self.pdfobjs = pdfobjs

    def log_prob(self, data):
        constituent_data = self.tv.split(data)
        pdfvals = [p.log_prob(d) for p, d in zip(self.pdfobjs, constituent_data)]
        return joint_logpdf(pdfvals)

    def expected_data(self):
        tostitch = [p.expected_data() for p in self.pdfobjs]
        return self.tv.stitch(tostitch)

    def sample(self, sample_shape=()):
        return self.tv.stitch([p.sample(sample_shape) for p in self.pdfobjs])


def joint_logpdf(terms):
    tensorlib, _ = get_backend()
    terms = tensorlib.stack(terms)
    result = tensorlib.sum(terms, axis=0)
    return result
```

Libraries


pyhf
combine
statisfactory?
(here I mean some mini-lang to construct pdfs
a la RooFit workspace factory)
magic
(covnersion)
pyfitcore(?).distributions.Model
(e.g. factored out version of
pyhf.probability. Defines basic pdf API around
sampling, evaluation, broadcasting...)
Ylib.inference
(RooStats / *Calculator / pyro.infer)
a library that uses abstract pdf APIs
to drive various inference techniques


RooFit factory()
ROOT HistFactory
magic
hist2workspace
Workspace
RooStats

Libraries

In ROOT Workspace concept a "container" like object that holds

- **pdf objects**
- **info about parameter structure (ModelConfig)**
- **certain parameter snapshots**
- **data structure (GlobalObservables, obsData)**
- **specific data snapshots (Asimov, observed)**

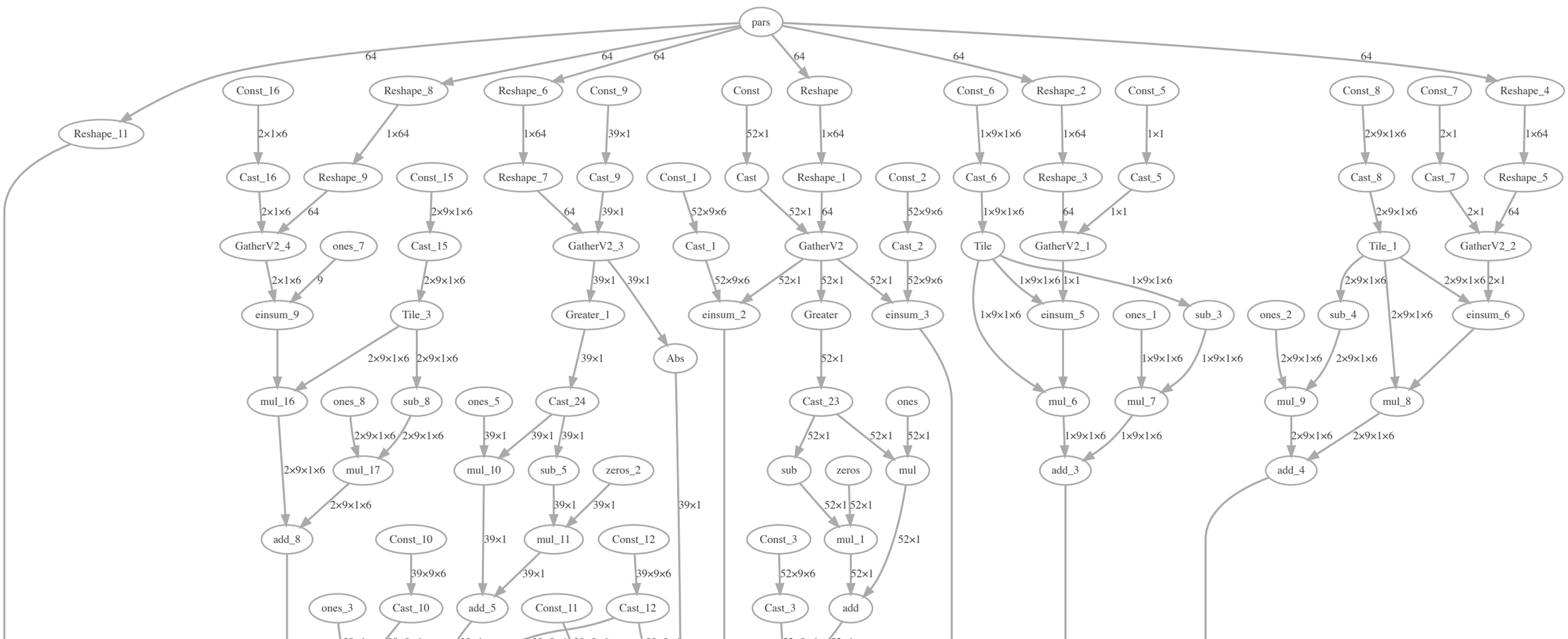
very rudimentary concept in pyhf for now

```
w = pyhf.Workspace(json.load(open('./s+b.json')))
m = w.model() #pars the spec
data = w.data(m)  #full data depends on model details (global obs/auxdata)
pars = m.config.suggested_init() #one of the func of ModelConfig
pdf = m.make_pdf(pars)  # torch.distr-like pdf obj w/ fixed pars
```

Libraries

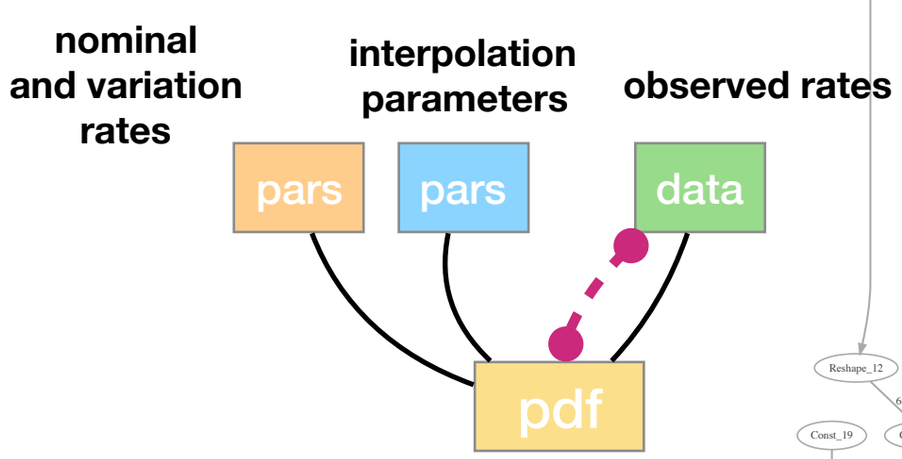
ModelConfig: can translate high-level physics settings to between low-level inputs for pdf

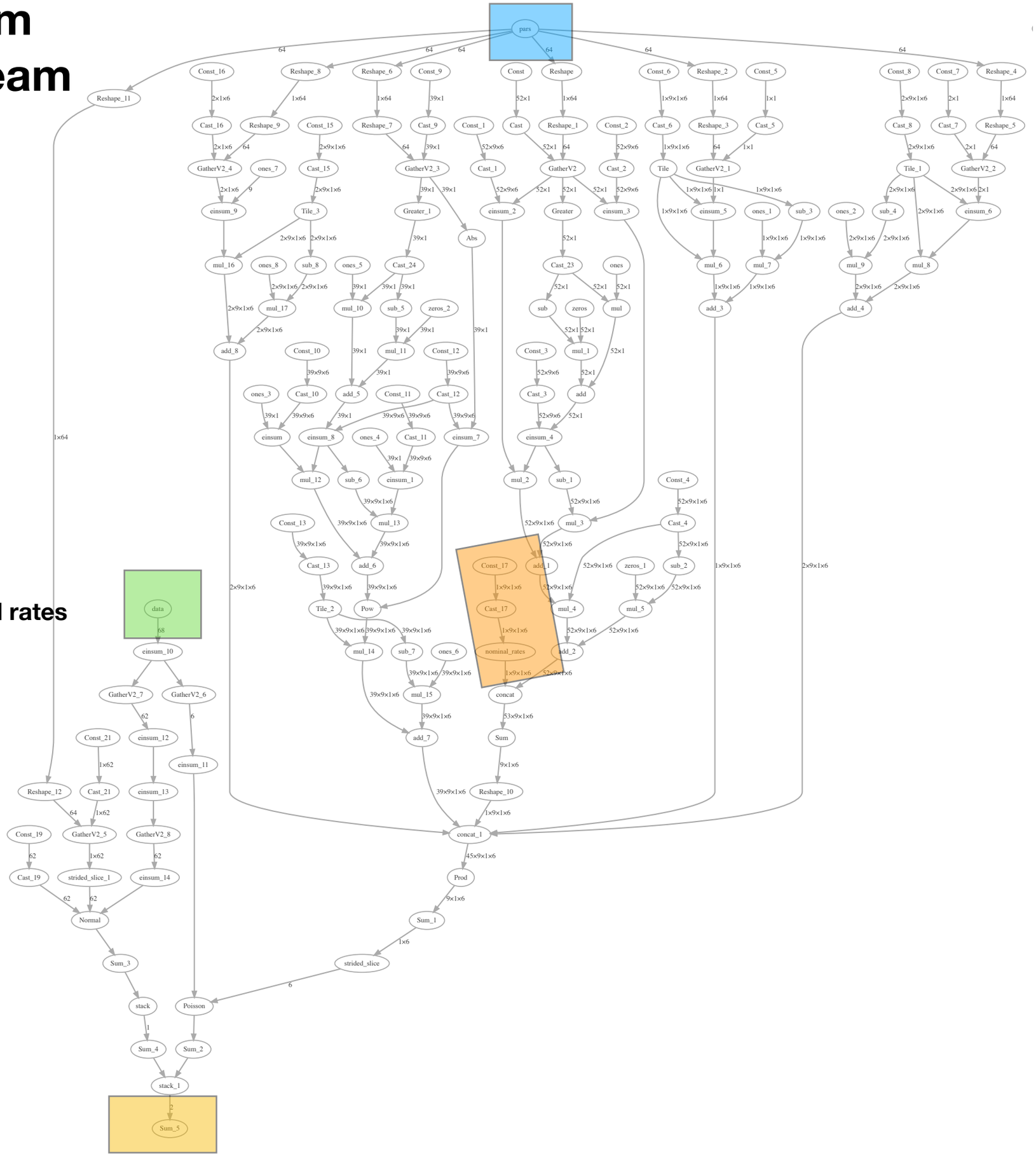
- **give me the POI**
- **give me aprameters but with poi set to X**
- **give me "aux data" associated with given constraint, ...**



Connecting to upstream

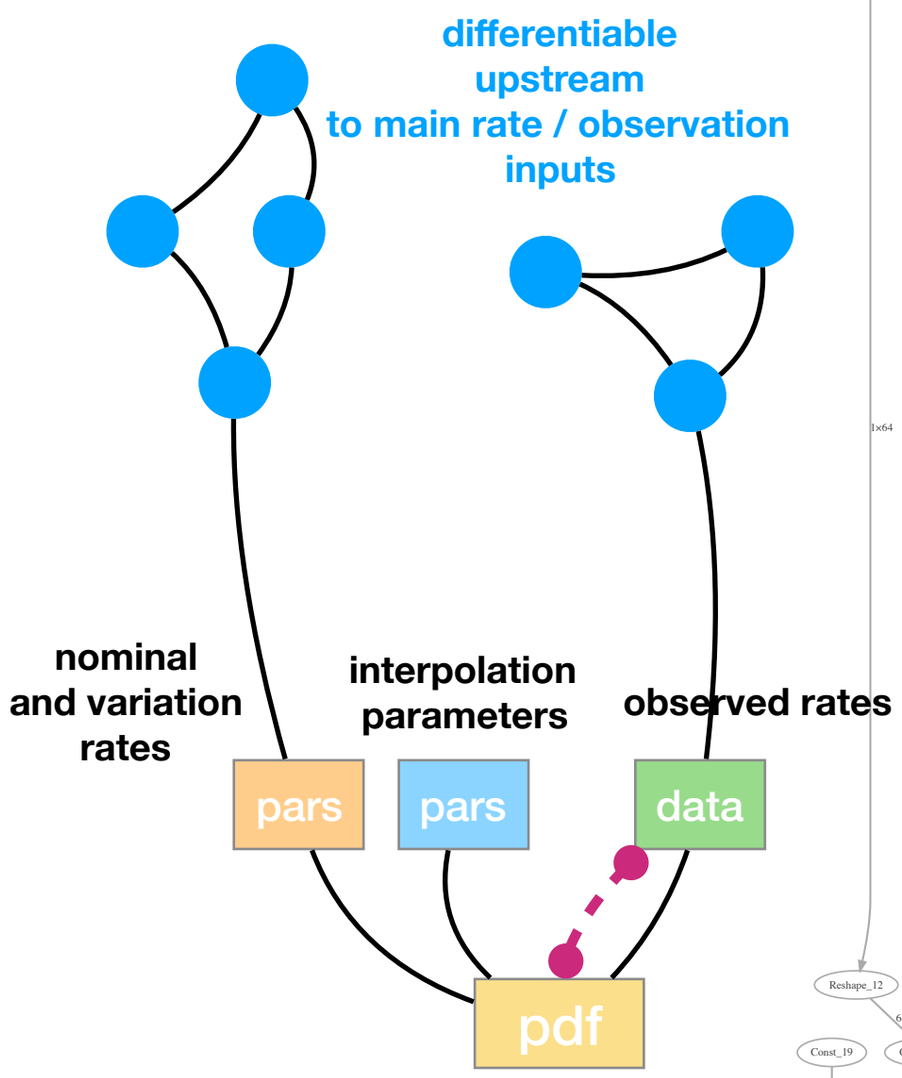
- **Differentiable upstream**

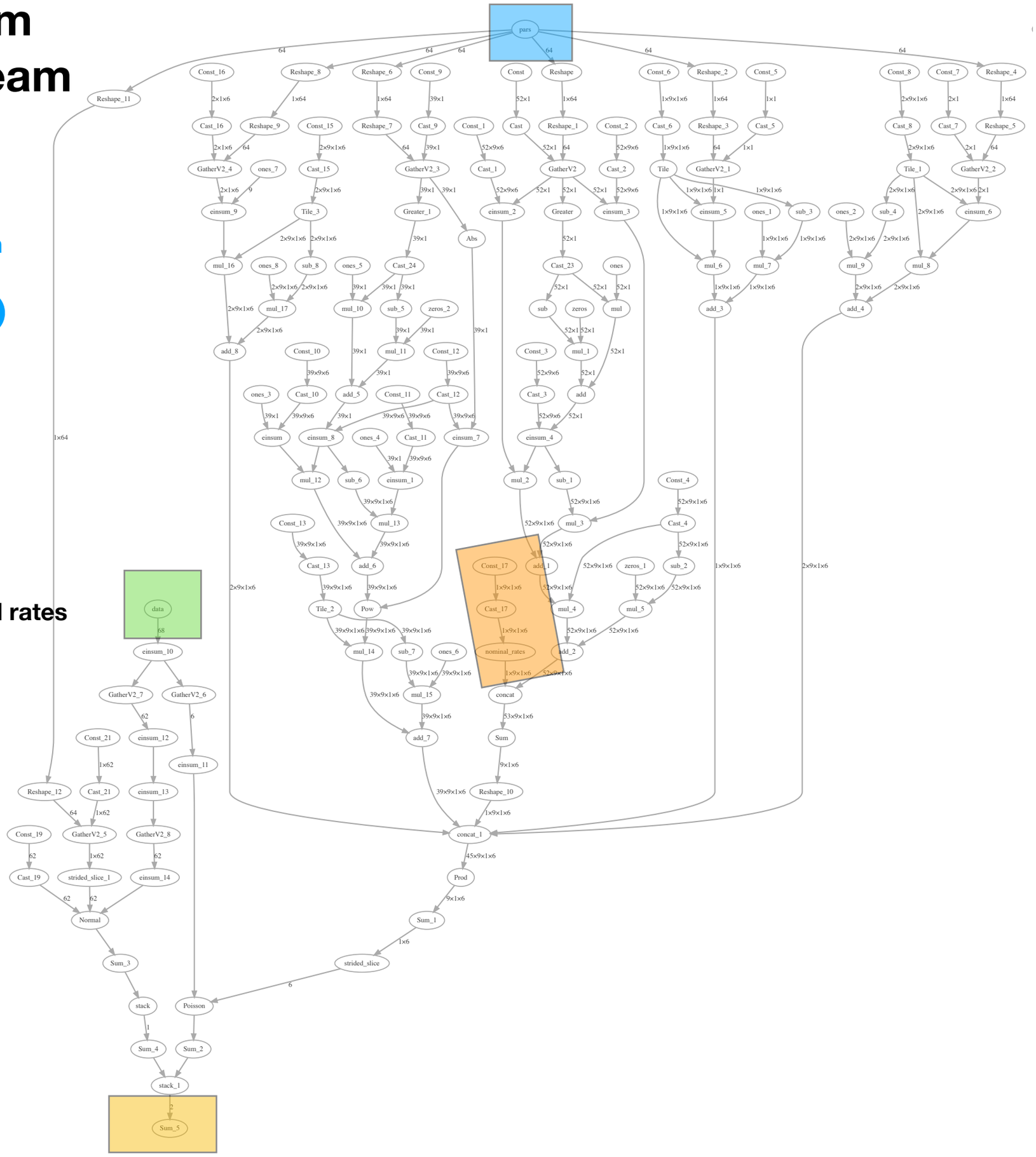




Connecting to upstream

- **Differentiable upstream**





Static Upstream:

~ HistFitter / TRexFitter

Small demo

https://github.com/lukasheinrich/pyhfinput

Uses Regionn Definition uproot/coffea

- formulate
- numexpr

High Impact MiniProject:

TTree::Draw drop-in replacement

JPivarski: ROOT has large test suite to validate against

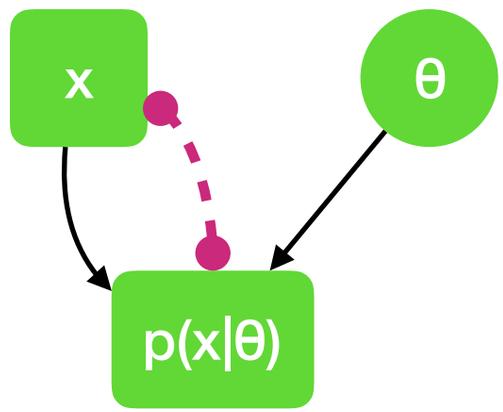
41 lines (37 sloc) | 847 Bytes

```yaml
samples:
- name: 'data'
  variations:
  - name: nominal
    tree: data16
    data: true
    glob: 'exported/data16/*.root'

- name: 'wjets_mc16a'
  variations:
  - name: nominal
    tree: wjets_mc16a_Nom
    glob: 'exported/wjets_mc16a/*.root'
    weight: 'xs_weight * weight'
    lumi: 36207.66
- name: 'zjets_mc16a'
  variations:
  - name: nominal
    tree: zjets_mc16a_Nom
    glob: 'exported/zjets_mc16a/*.root'
    weight: 'xs_weight * weight'
    lumi: 36207.66
- name: 'ttbar_mc16a'
  variations:
  - name: nominal
    tree: ttbar_mc16a_Nom
    glob: 'exported/ttbar_mc16a/*.root'
    weight: 'xs_weight * weight'
    lumi: 36207.66

regions:
- name: regionA
  binning: [-0.5,15.5,17]
  filter: 'n_jet < 7 && n_mu > 0'
  observable: 'n_jet'
- name: regionB
  binning: [-0.5,15.5,17]
  filter: 'n_jet > 8 && n_mu > 0'
  observable: 'n_jet'

```

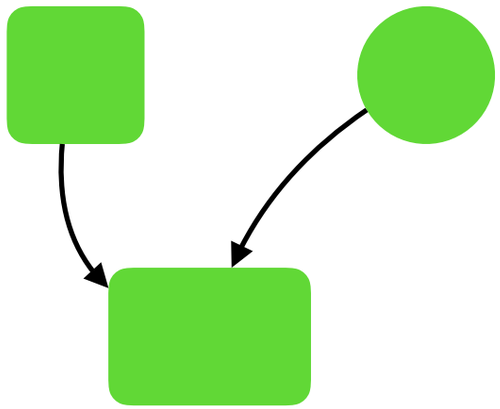
Backup

pdf's in computatioal graphs

$$\int \mathrm{d}x\, f(x|\theta) = 1$$



$$f(x|\theta)$$



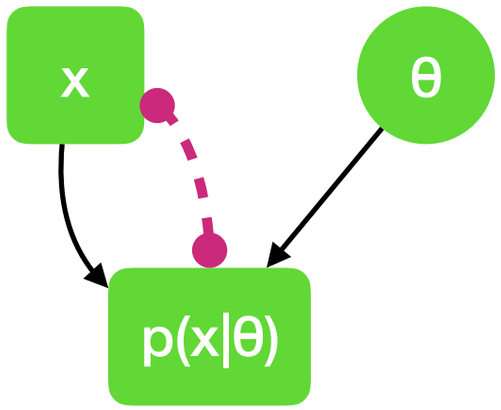
computational
graph

pdf is a
computational graph +
a normalization relationship

Option A: known functions

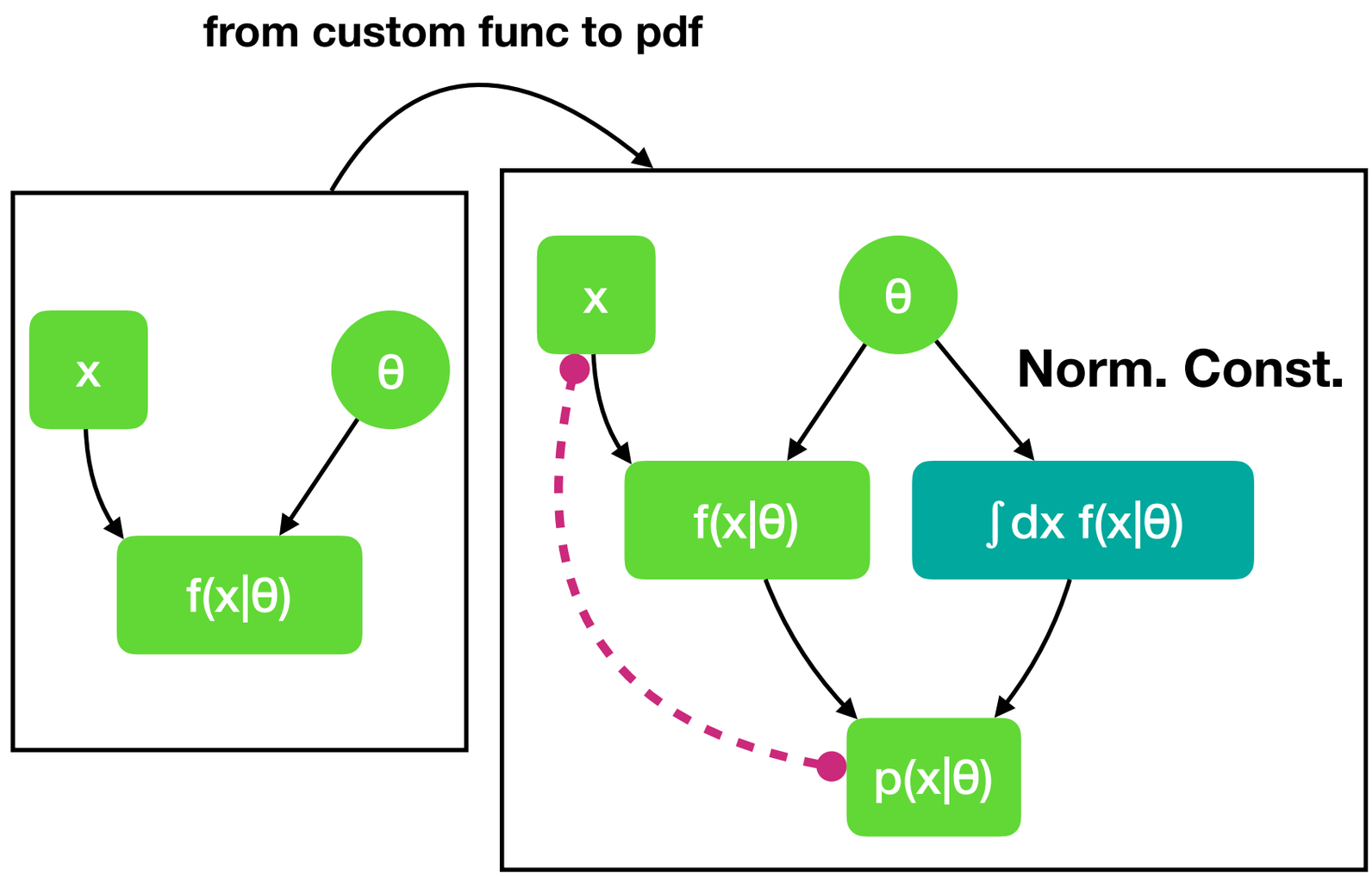
- **Poisson**
- **Normal**
- **...**

here it's enough to just have the graph w/ the correctly normalized formula

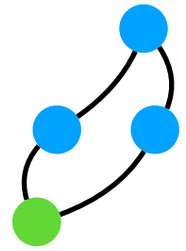


Option B: custom functions

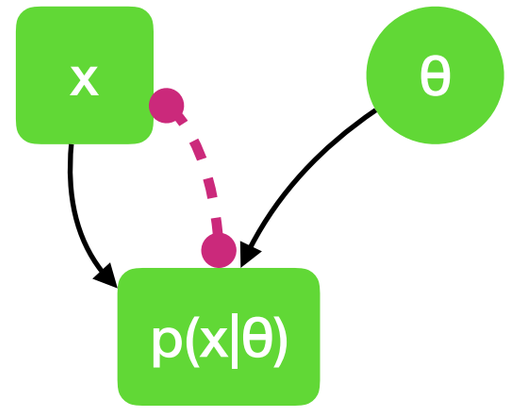
- **need integral**
  - **analytical (as part of graph)**
  - **from numerical integ**



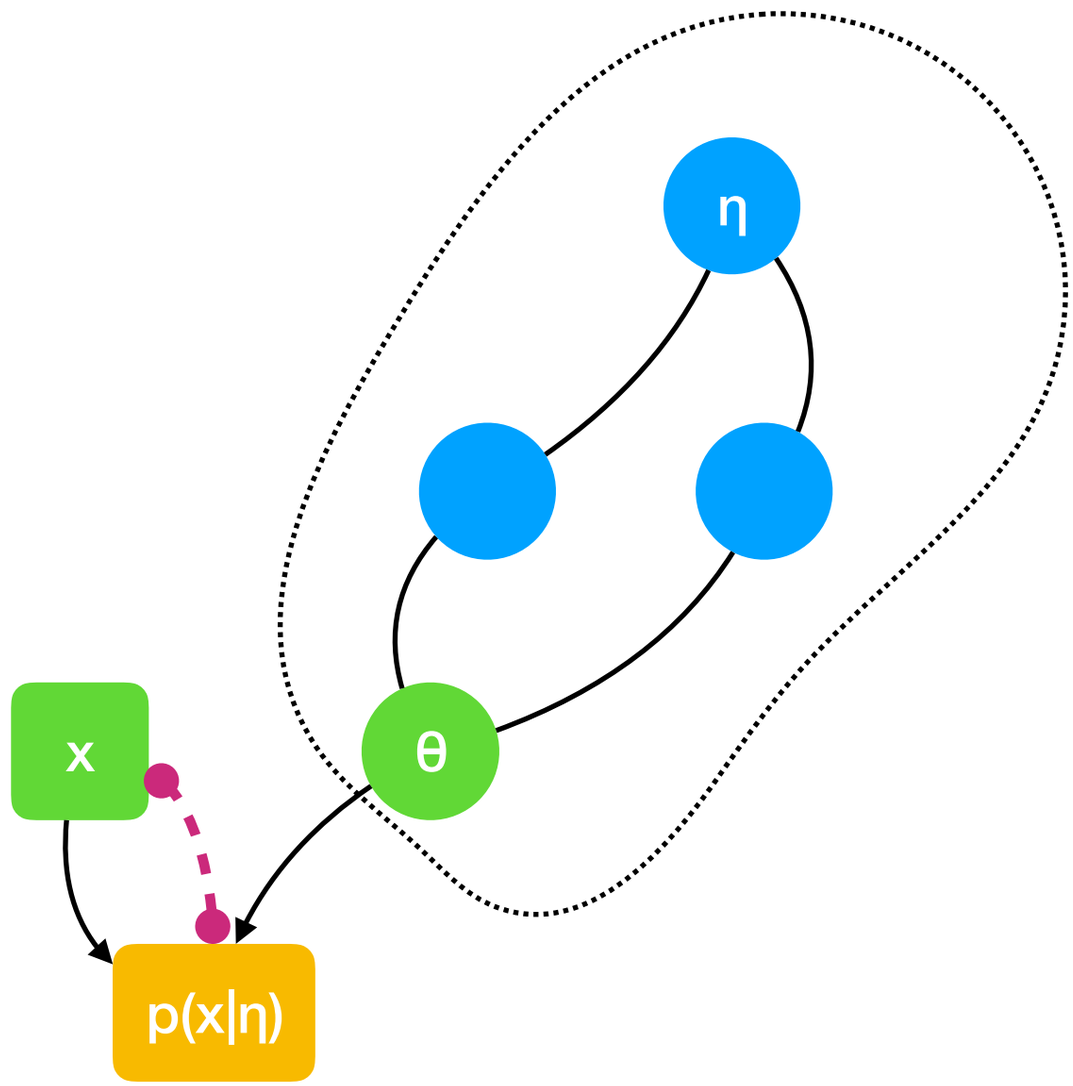
parameter transform



$\theta = f(\eta)$



normalization relationship



Such parametrizations not an issue at all (think custom ctors)

mypdf.from_eta(...)
mypdf.from_theta(...)

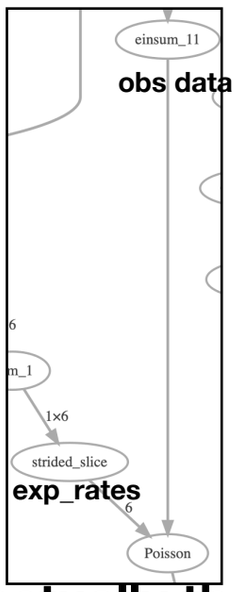
e.g. in pyhf the core pdf at the end is a simple

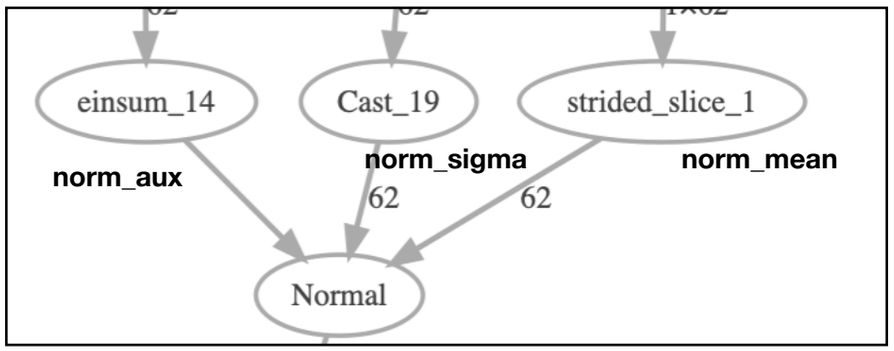
Pois(obs data | expected rates)
main pdf

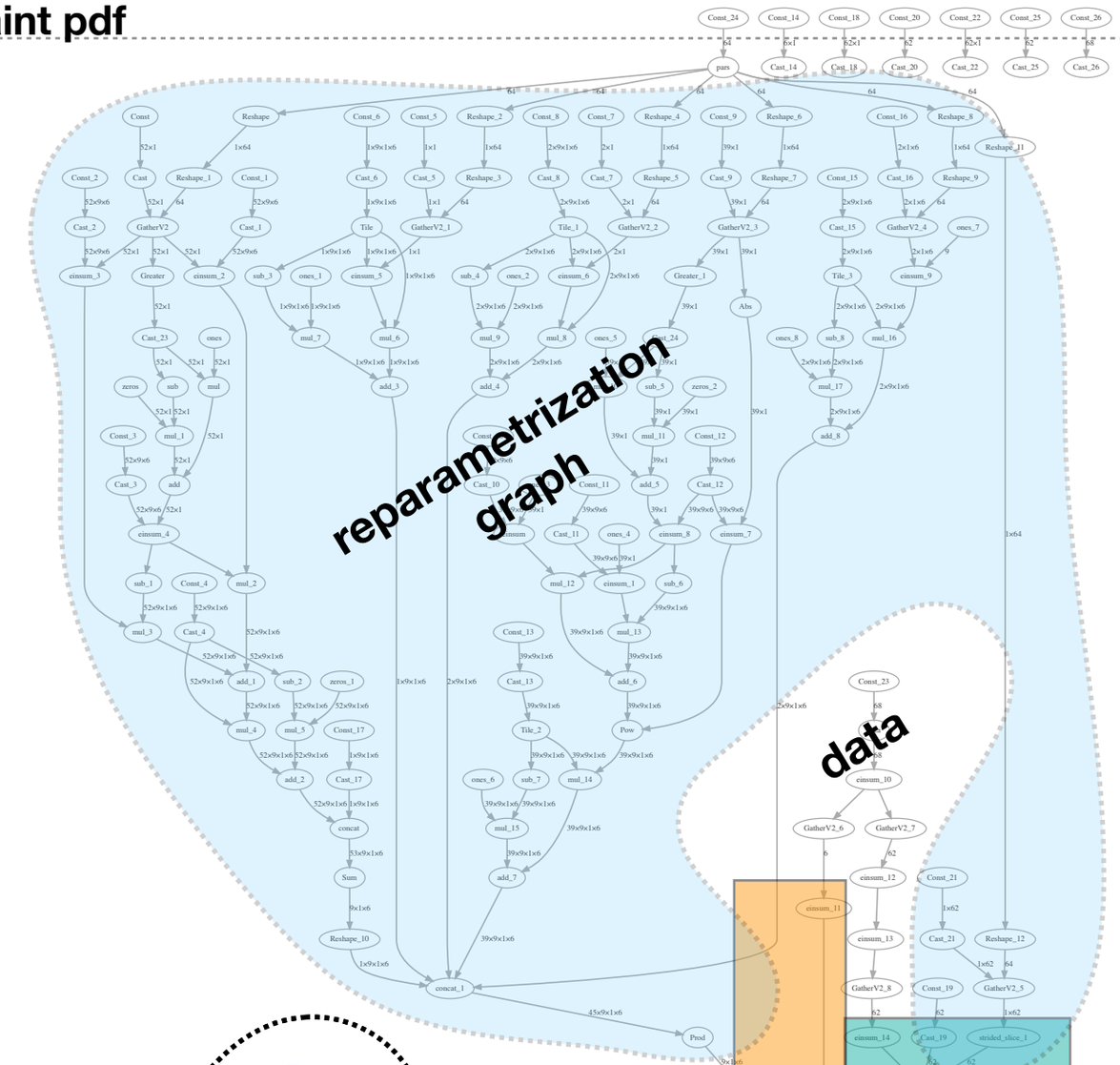
Pois(pois_aux| pois_rates)
poisson constraint

Norm(norm_aux | norm_mean, norm_sigma)
gaussian constraint

constraint pdf

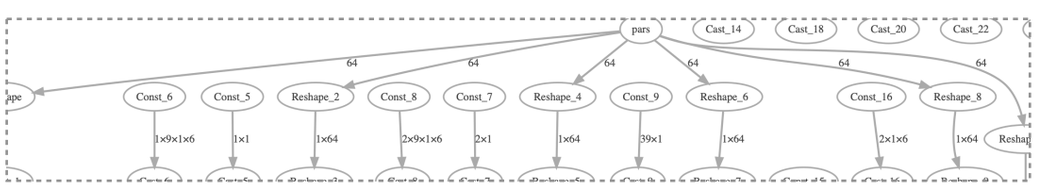




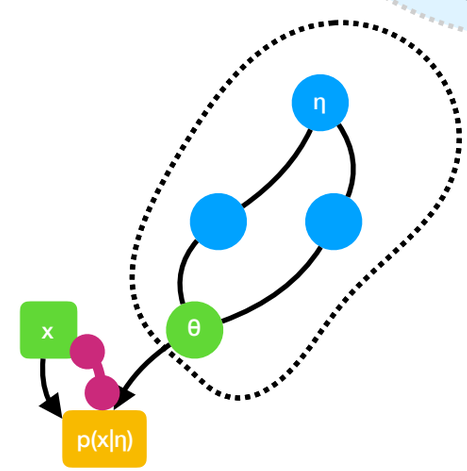


but actually the parameters of those core pdfs are functions of our "physics" parameters

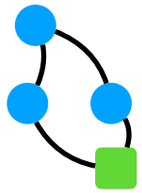
expected rates = expected rates(pars)
pois_rates = pois_rates(pars)
norm_mean = norm_mean(pars)
norm_sigma = norm_sigma(pars)



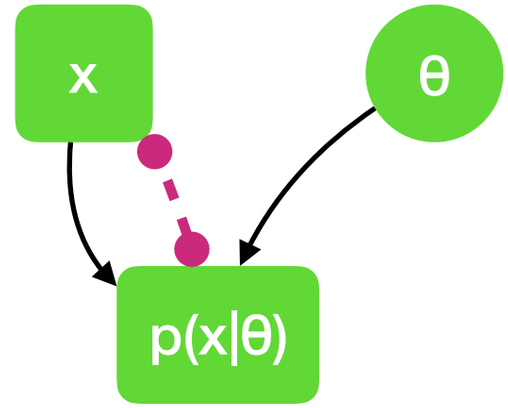
+ large graph



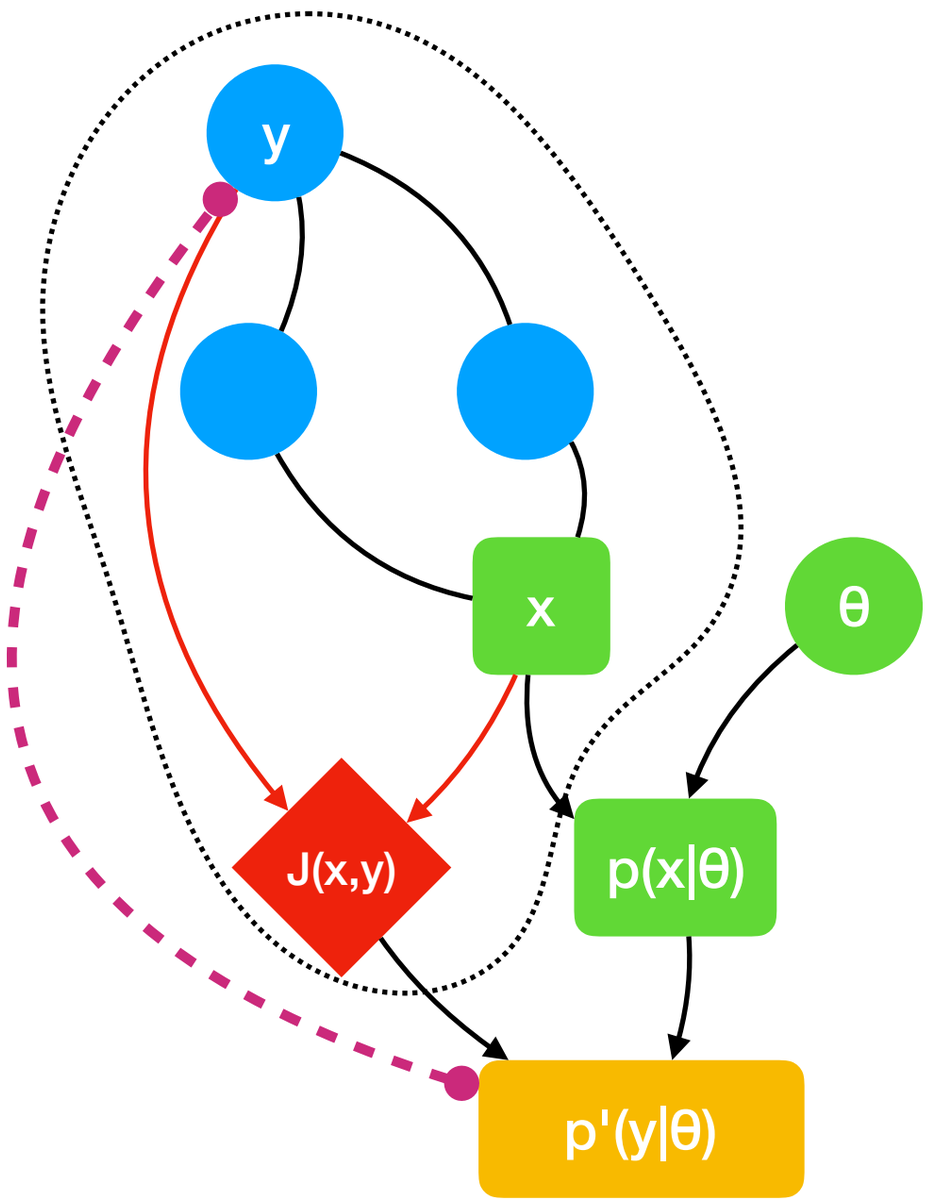
data transform



x = f(y)



Option 1

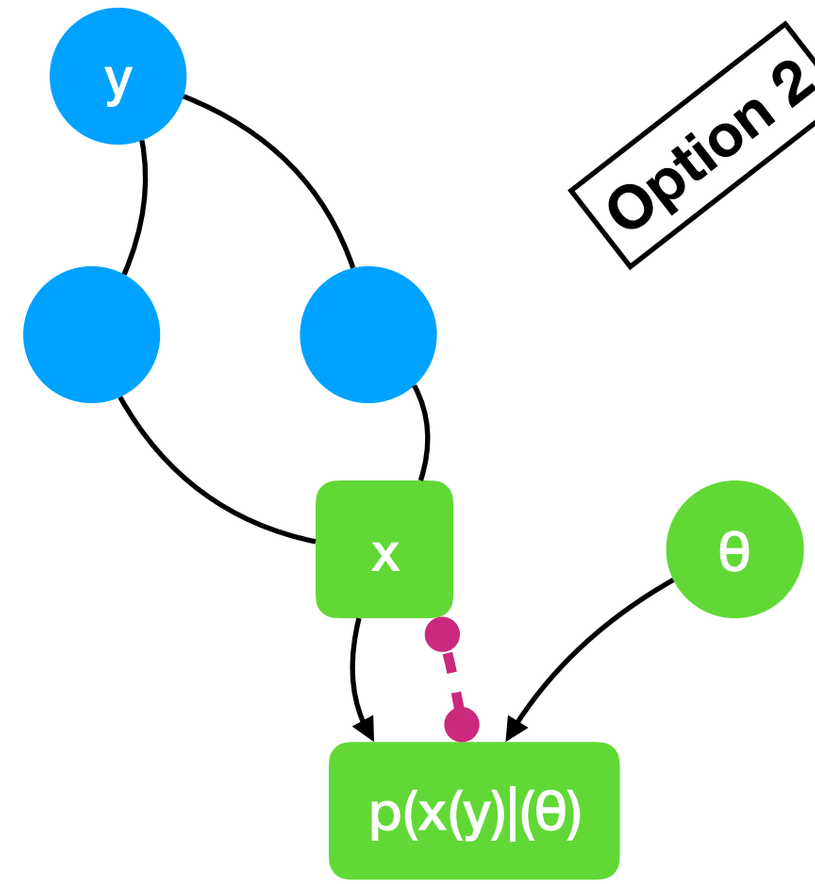


if we want new pdf to be normalized we need Jacobian factor

note: new norm. rel

more complicated to correctly have this this be p'(y|θ)

Option 2



old pdf still normalized wrt x, but x computed on the fly
keep old norm. rel

.. on-the-fly calc part of same diff graph.

- joint pdf: neither tesorflow prob / torch have primitives for it

$$p(d|\theta) = p(d_A, d_B|\theta_A, \theta_B) = p(d_A, \theta_A) \cdot p(d_B, \theta_B)$$

- when generating (e.g. toys), need to be able to **create joint data from individual data** ("stitching")

$$(d_A, d_B) \to d$$

- when evaluating need to be able to **split data into individual pieces** ("projection")

$$d \to (d_A, d_B)$$

Note: could by convention be simple concat/slice, but doesn't

- joint pdf

$$p(d|\theta) = p(d_A, d_B|\theta_A, \theta_B) = p(d_A, \theta_A) \cdot p(d_B, \theta_B)$$

- when instantiating with p(·|θ) need to be able to derive individual parameters from joint parameters:

$$\theta \to (\theta_A, \theta_B)$$

- can be functions that are **not purely projection** on subspace
  - mix between projection/splitting and reparametrization

$$\theta \to (\theta_A(\theta), \theta_B(\theta))$$