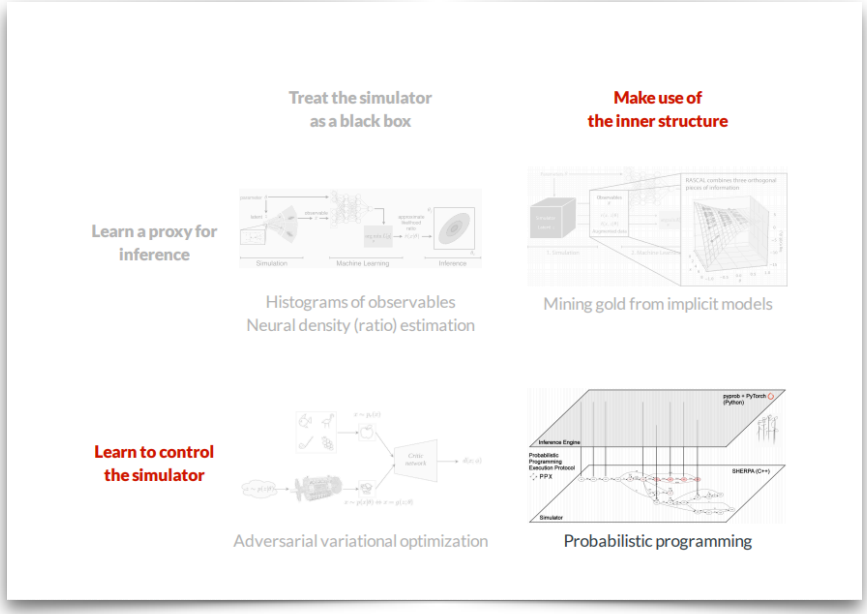
Probabilistic Programming
Modelling
Inference

Lukas Heinrich | MLHEP DESY



... thanks Gilles :)

glouppe 8:00 PM

Oh I didn't realize you were doing this tutorial at the yandex summer school (edited)

I just gave a talk myself

I might have spoiled a bit the probabilistic programming part :)

The world is complex and often things are not fully deterministic

- intrinsically (QM)
- lack of complete information (complex world)

In these situations we try to use

what we can observe

to gain information about

what we want to know.

But we need a language to describe the relationship between them

>> statistical modelling

... and the mathematical toolbox to conclude from observations

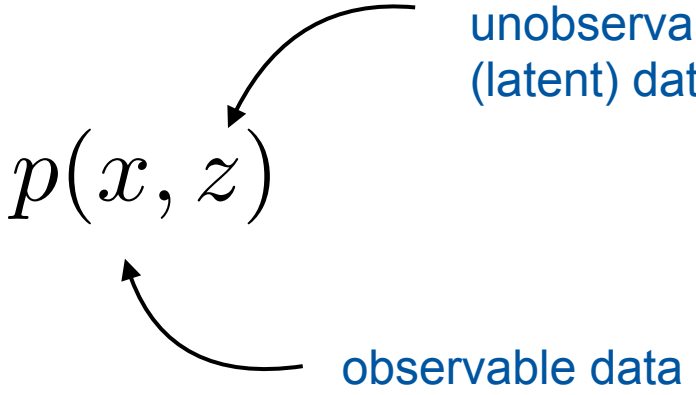
>> statistial inference

Statistical Modelling:

- the data we observe is assumed to be sampled according to a model (probability density function)

Statistical Modelling:

- the data we observe is assumed to be sampled according to a model (probability density function)
- the data space can be extended to include unobservable data, that are random variables but unobservable
  - the latent space



$$p(x) = \int \mathrm{d}z \; p(x, z)$$

Statistical Modelling:

- **there might be parameters of the distribution**

$$p(x, z; \alpha)$$

random variables

not random variables

- **sometimes just fixed choices to pick a specific model from a parametrized family**

Bayesian vs Frequentist Inference

Inference:

statement based on observed data x about other parts of the model

Bayesian View:

target of inference are **unobservable random variables**. Want posterior distribution on them.

Frequentist View:

target of inference are **fixed parameters of the model**. Want (random) estimates of true value.

Bayesian:

- parameters are random variables
- data pdf is conditioned on parameter values
- p(z) consitutes prior belief in how probable various parameters are

$$p(x, z) = p(x|z)p(z)$$

Bayesian:

Inference means:
finding the posterior on (unobservable) random variables, i.e. parameters

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}$$

Result of inference: a probability density

Frequentist:

Parameters are not random variables but fixed constants (i.e. they have true value)

$$p(x) = p(x|\alpha)$$

Inference means: find estimators of e.g. points or sets.

Result of inference: random elements (points, sets) with certain properties (coverage)

First Problem: formulating $p(x, z; \alpha)$

Simple:

data follows a well-known distribution

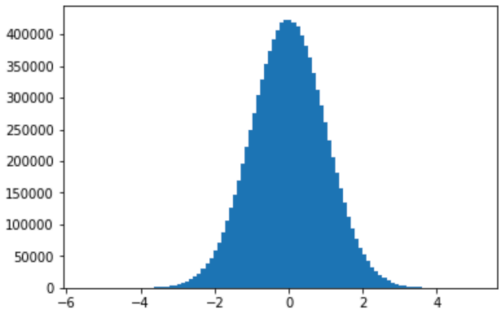
$$x \sim p(x) = \mathcal{N}(\mu, \sigma)$$

Advantages:
- easy to compute
- easy for inference (maybe exact?)

Disadvantages:
- almost never happens

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

```
plt.hist(np.random.normal(0,1, size = int(1e7)), bins = 100);
```



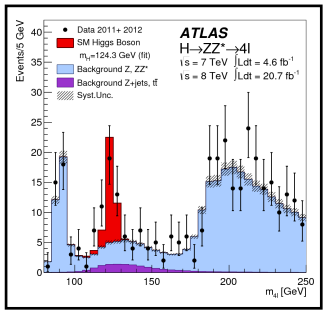
Useful:

can model data within flexible template

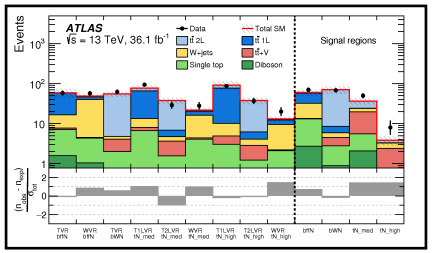
- combine specific fundamental pdfs into larger construct e.g. `HistFactory` or `Combine`

Advantages:
- limited number of building blocks, can be heavily optimized

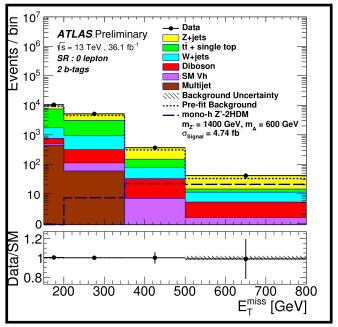
Disadvantages:
- limited number of building blocks, not everything fits the bill (unbinned fits?)



Standard Model



SUSY

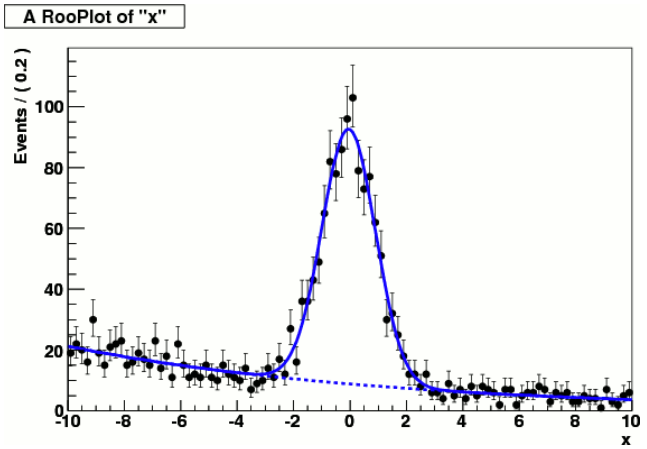


Exotics

$$\mathcal{P}(n_c, x_e, a_p \mid \phi_p, \alpha_p, \gamma_b) = \prod_{c \in \text{channels}} \left[\text{Pois}(n_c|\nu_c) \prod_{e=1}^{n_c} f_c(x_e|\boldsymbol{\alpha}) \right] \cdot G(L_0|\lambda, \Delta_L) \cdot \prod_{p \in \mathbb{S}+\boldsymbol{\Gamma}} f_p(a_p|\alpha_p)$$

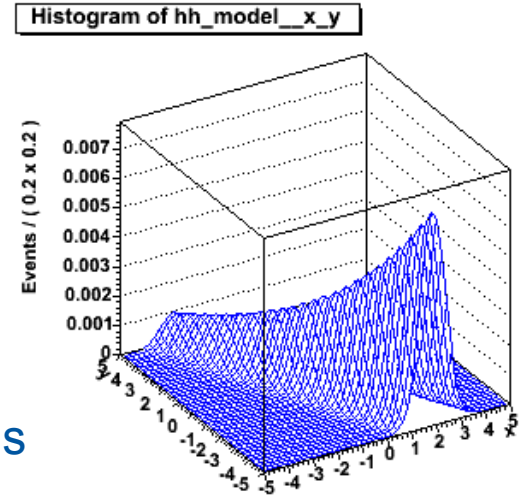
Open World of Statistics:

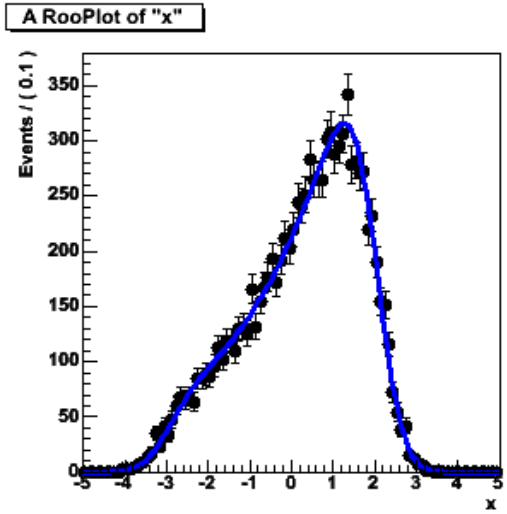
flexible modelling through open-ended composability of building blocks



Advantages:
- powerful modelling

Disadvantages:
- needs much bigger framework / data model
- harder to find common abstractions
- still not completely arbitrary





A more general picture?

A more general picture?
... we already have a good way to describe input and output relations

A more general picture?
... we already have a good way to describe input and output relations

Computer Programs.

A more general picture?
... we already have a good way to describe input and output relations

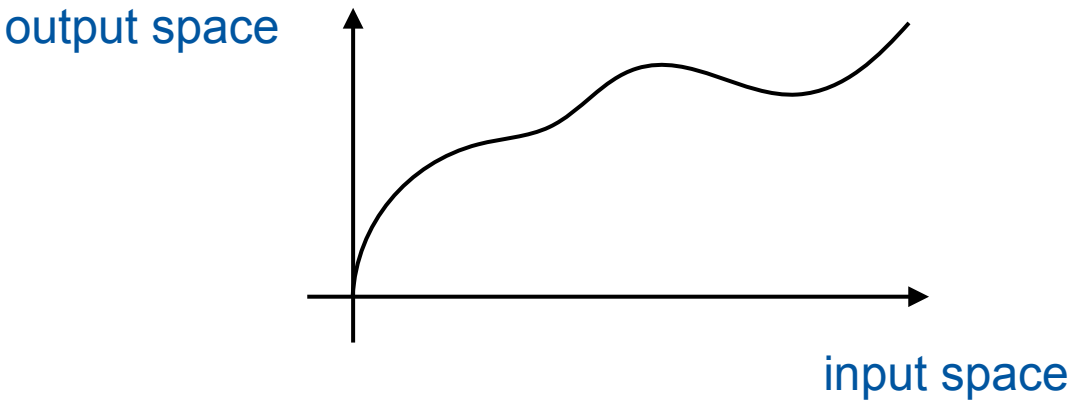
Computer Programs.
Probabilistic Programs.

Programs as functions

prog: y = f(x)

many programs are just deterministic function

- **path through the program is fixed once the input is given**

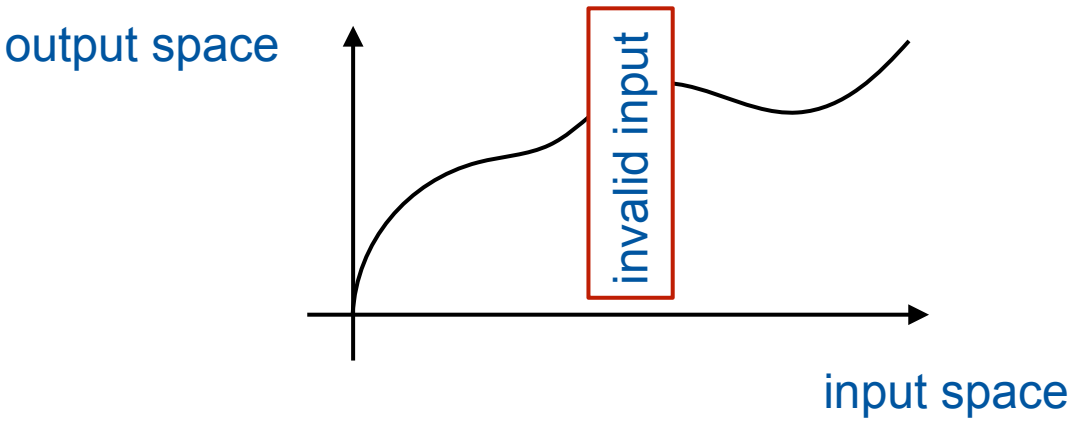


Programs as functions

prog: y = f(x)

not all inputs are valid

- **e.g. consider programs with assertions**
- **given input, the path is fixed but may be an invalid path**



what's the input space?
what are possible outputs?

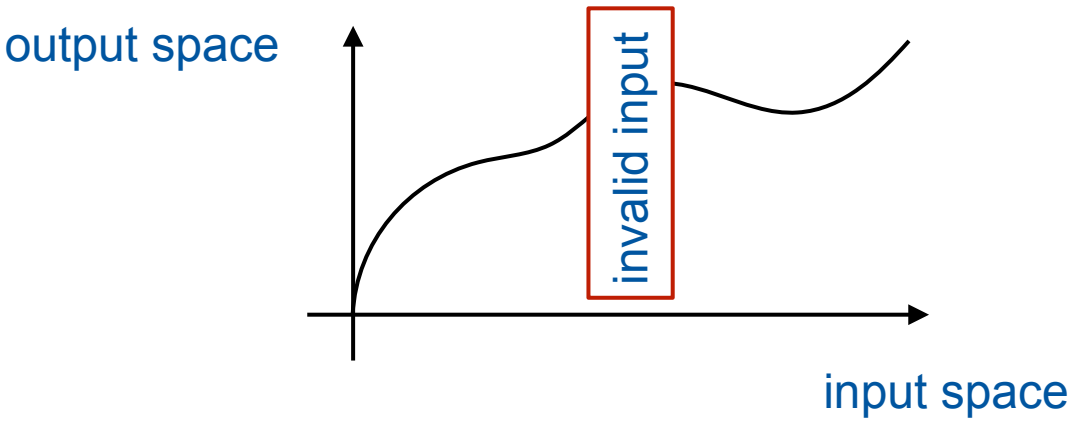
assert.py

Users ▸ lukasheinrich ▸ Code ▸ hamburl_mlhep ▸ ok ▸ assert.py ▸ main

```python
import sys

def function(input_value):
    assert input_value in ['hi','world']
    return_value = len(sys.argv[1])
    return return_value

def main():
    result = function(sys.argv[1])
    print(f'the result is {result}')
    return 0


if __name__ == '__main__':
    main()
```

```
> python assert.py hi
the result is 2
> python assert.py world
the result is 5
> python assert.py hello

Traceback (most recent call last):
  File "assert.py", line 15, in <module>
    main()
  File "assert.py", line 9, in main
    result = function(sys.argv[1])
  File "assert.py", line 4, in function
    assert input_value in ['hi','world']
AssertionError
```

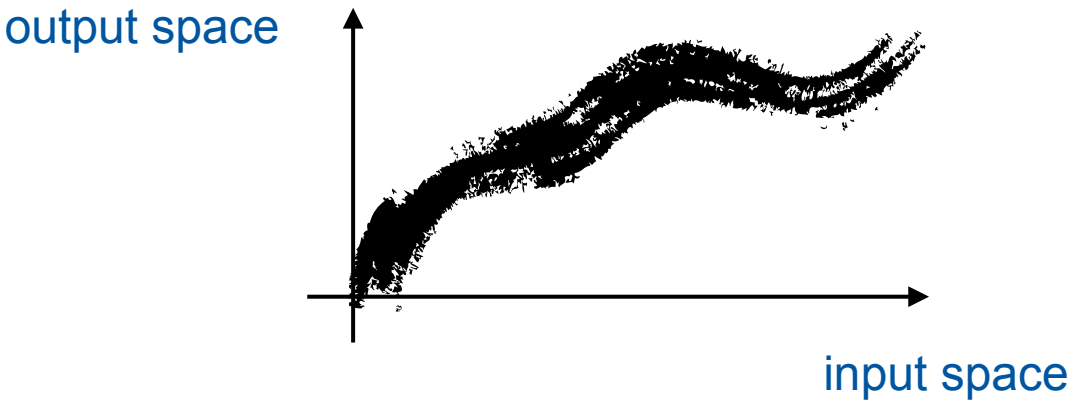
`assert` usage
blocks certain passes through the program



Programs as implicit densities

prog: p(x;input)

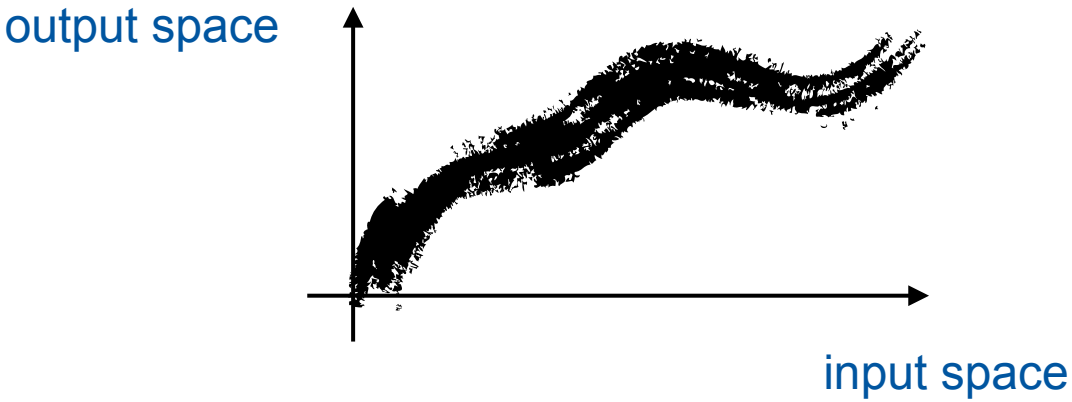
path through the program is not fixed but depends on probabilistic choices



probprog.py

```python
import sys
import random
import math

def function(begin,end):
    latent_value = random.randint(begin,end)
    return_value = math.sqrt(latent_value)
    return return_value

def main():
    result = function(int(sys.argv[1]),int(sys.argv[2]))
    print(f'the result is {result}')
    return 0


if __name__ == '__main__':
    main()
```

```
> for i in `seq 0 3`; do python probprog.py -10 10;done
the result is 3.1622776601683795
Traceback (most recent call last):
  File "probprog.py", line 17, in <module>
    main()
  File "probprog.py", line 11, in main
    result = function(int(sys.argv[1]),int(sys.argv[2]))
  File "probprog.py", line 7, in function
    return_value = math.sqrt(latent_value)
ValueError: math domain error
the result is 2.449489742783178
the result is 2.0
```



A programming language can be made probabilistic by adding a language construct to sample values from distributions

```
z_1 = sample Normal(0,1);
z_2 = sample Normal(0,z_1);
```

sampling functionality either provided by a library or added as a language primitive / keyword (cf. if/else ...)

each `sample` an ERP: Elementary Random Procedure

The Trace:

In the absence of non-stochastic procedure calls the flow through the program is determinstic

$\longleftrightarrow$

The flow through the program is completely defined by the results stochastic procedure calls

Each ERP call: a latent variable z_i

Full pdf: series of conditionals $p(z) = \prod_i p(z_i | z_{<z_i})$

Observations

Not all random numbers of a stochastic program must be latent - some of those variables may be observable!

$$p(z) \rightarrow p(x, z)$$

introduce a way to declare observable random numbers in programs: new keyword!

```
z_1 = sample Normal(0,1);
x_1 = observe Normal(0,1);
```

With observations, the full trace is given by

$$p(x, z) = \prod_j p(x_i | z_{<x_j}) \prod_i p(z_i | z_{<z_i})$$

trace: series of latent program statess

observations

Once we have observations we can ask interesting questions about inference.

Remember `assert`

→ blocked program runs incompatible with Assertion

Generalize to probabilistic programs

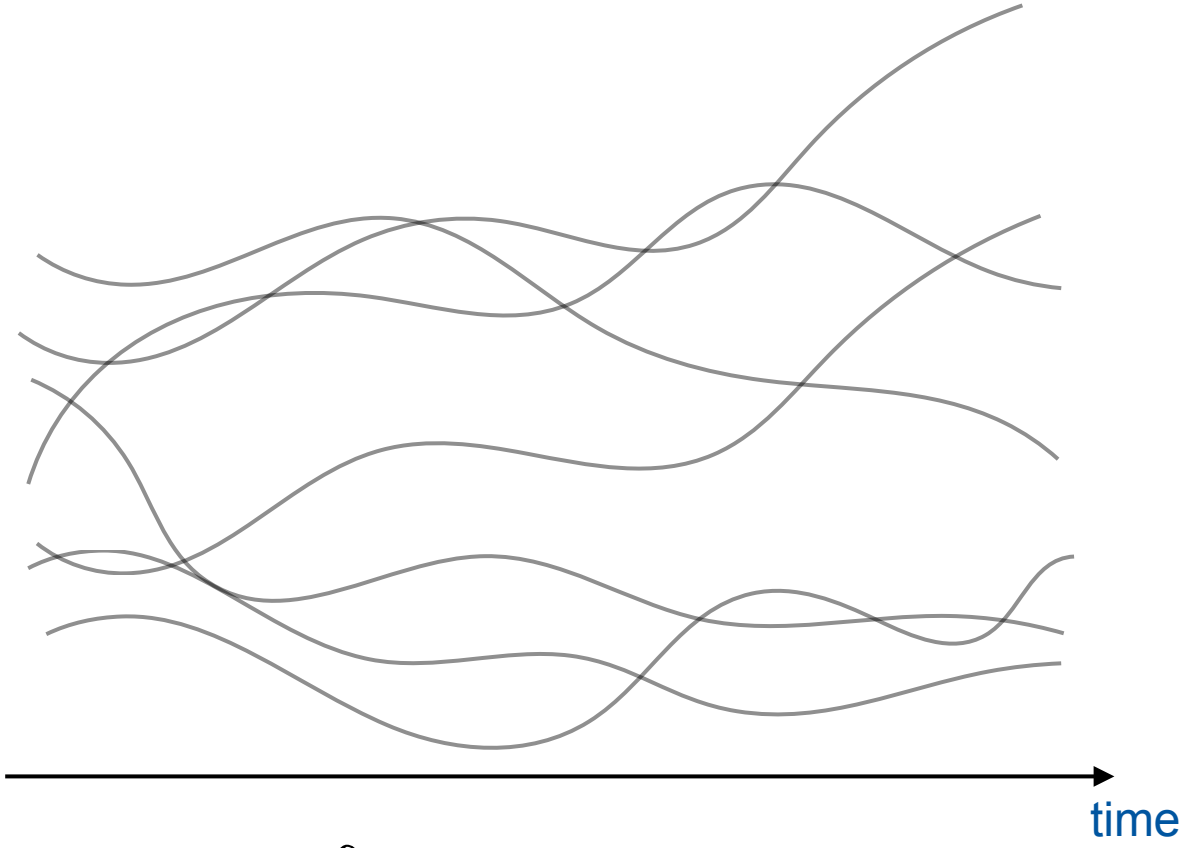
New Keyword `observe` can condition distribution of program runs:

$$p(z) \rightarrow p(z|x)$$

- block paths that are incompatible with observation
- normalize remaining possible paths
  - running program: sampling from the posterior
- (when not conditioned `observe` → `sample`)

$p(z)$

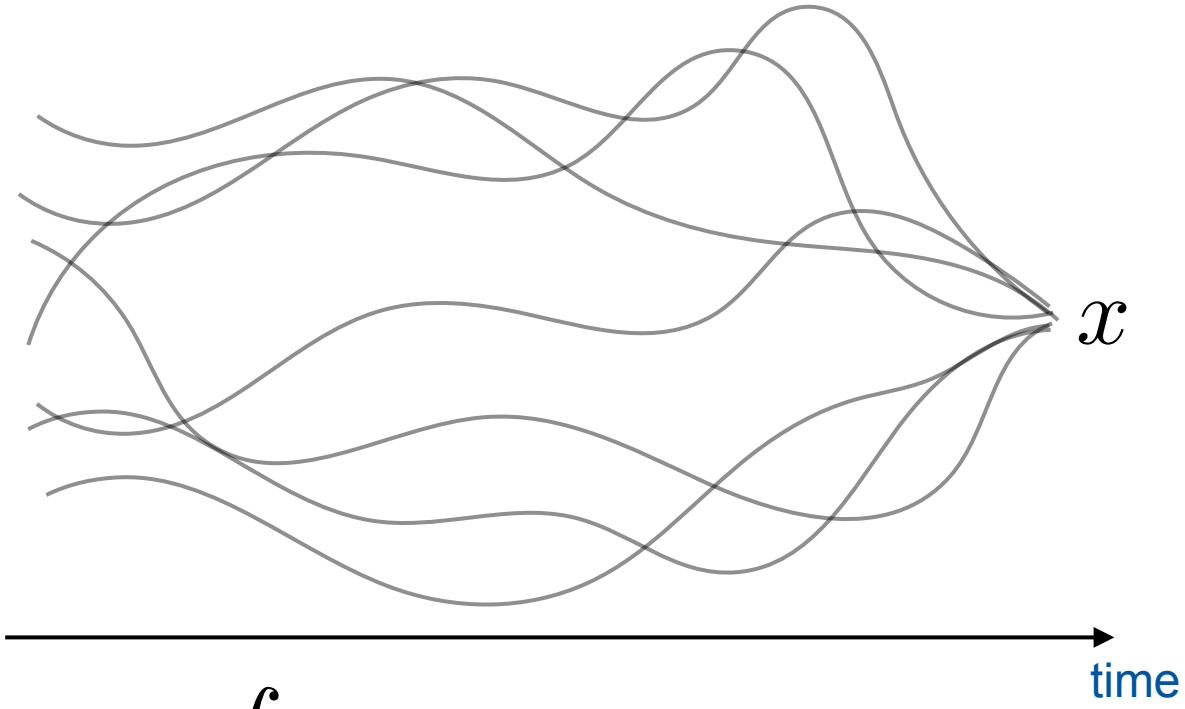
unconditioned probabilistic program



$$\int \mathrm{d}z \; p(z) = 1$$

$p(z|x)$

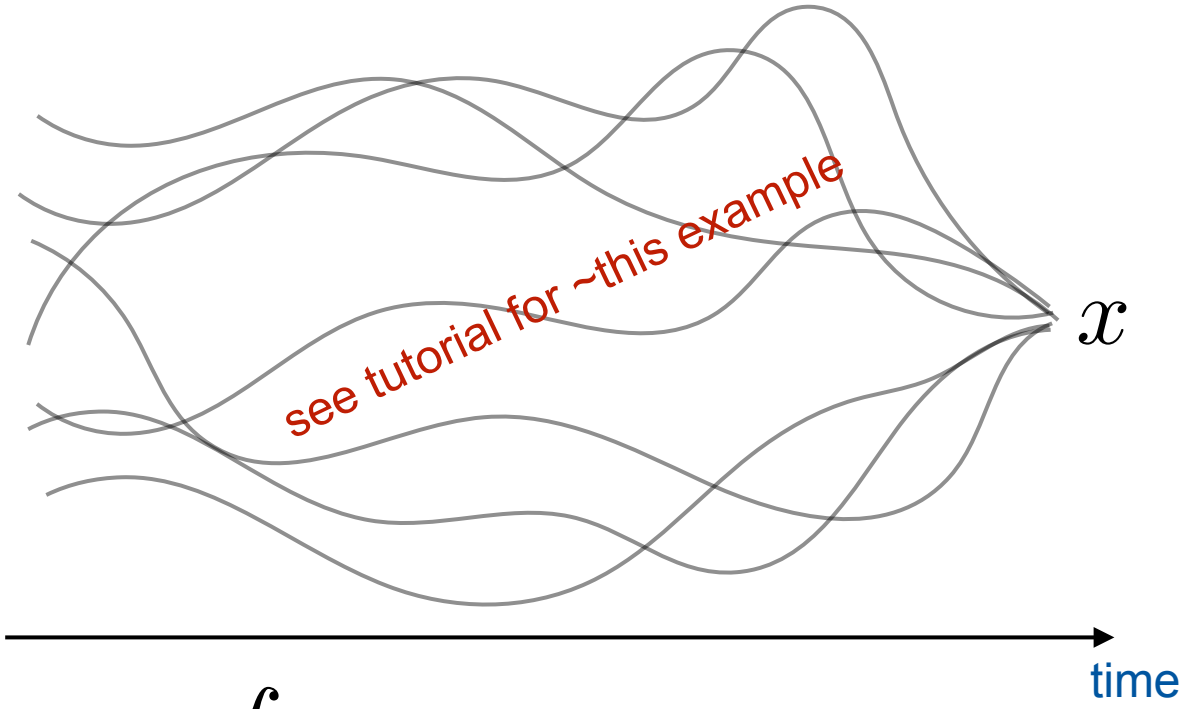
conditioned probabilistic program



$$\int \mathrm{d}z \; p(z|x) = 1$$

$p(z|x)$

conditioned probabilistic program



$$\int \mathrm{d}z \; p(z|x) = 1$$

Recap:

probabilistic programs are defined by a distribution of program traces
= generative model of data x with latent state z

probabilistic programming languages (PPLs) have two additional primitives `sample` and `observe`

PPLs can run programs in conditional or unconditional mode.

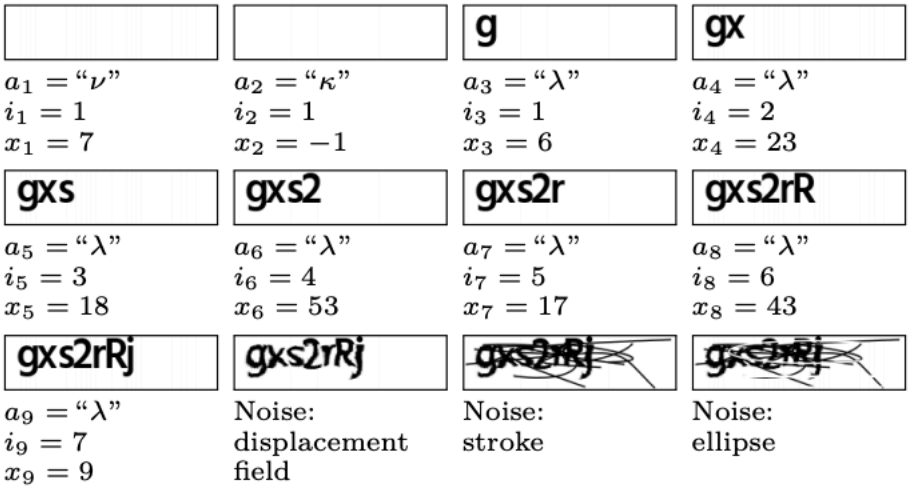
Example: a generative model of Captchas

How to generate captchas?

- create a random string
- render to pixels
- add distortions, etc.

Note:
the true string of the captcha is part of the latent space z !

```
1:  procedure CAPTCHA
2:      ν ~ p(ν)                          ▷ sample number of letters
3:      κ ~ p(κ)                          ▷ sample kerning value
4:  Generate letters:
5:      Λ ← {}
6:      for i = 1, ..., ν do
7:          λ ~ p(λ)                      ▷ sample letter identity
8:          Λ ← append(Λ, λ)
9:  Render:
10:     γ ← render(Λ, κ)
11:     π ~ p(π)                          ▷ sample noise parameters
12:     γ ← noise(γ, π)
13:     return γ
```



Example: a generative model of Captchas

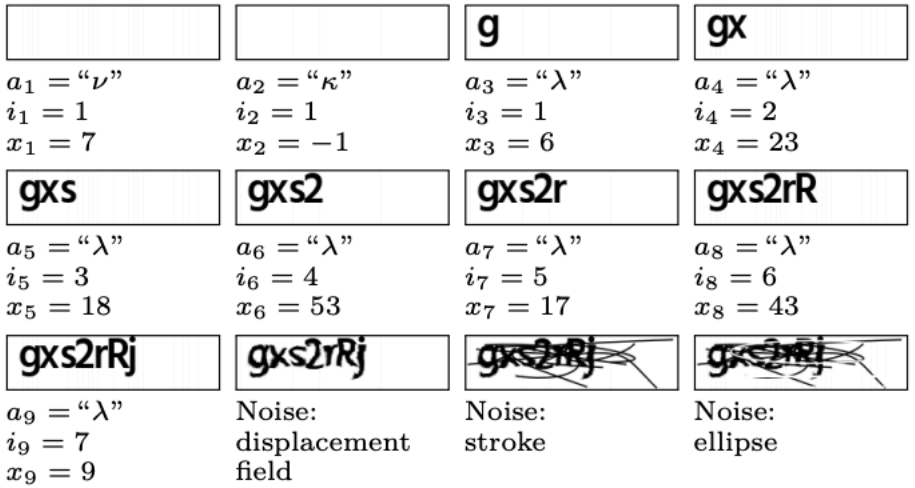
How to generate captchas?

- create a random string
- render to pixels
- add distortions, etc.

Note:
the true string of the captcha is part of the latent space z !

```
procedure CAPTCHA
    ν ∼ p(ν)                          ▷ sample number of letters
    κ ∼ p(κ)                          ▷ sample kerning value
Generate letters:
    Λ ← {}
    for i = 1, . . . , ν do
        λ ∼ p(λ)                      ▷ sample letter identity
        Λ ← append(Λ, λ)
Render:
    γ ← render(Λ, κ)
    π ∼ p(π)                          ▷ sample noise parameters
    γ ← noise(γ, π)
    return γ
```

see tutorial for ~this example



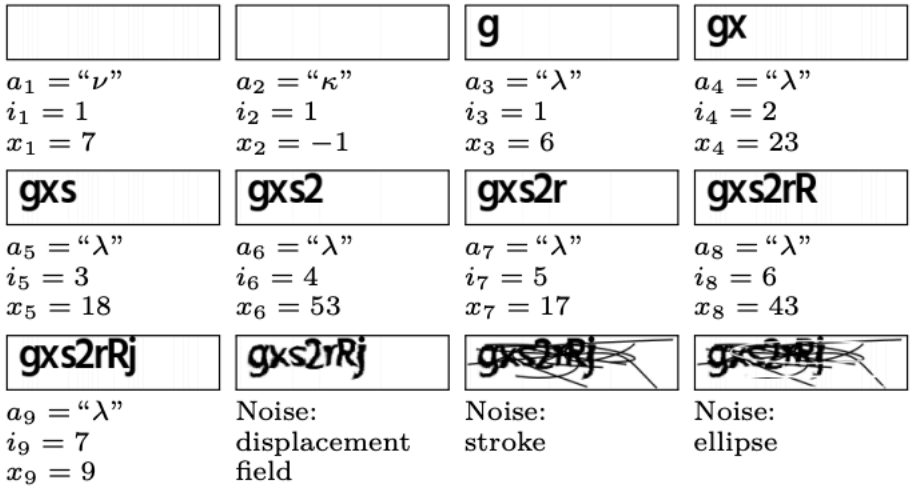
Example: a generative model of Captchas

How to break captchas?

Simple:

- condition on the observed image
- posterior $p(z|x)$ gives distribution over solutions.
- sample from $p(z|x)$, extract solution from latent state

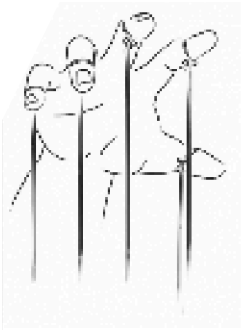
procedure CAPTCHA
  $\nu \sim p(\nu)$ ▷ **sample** number of letters
  $\kappa \sim p(\kappa)$ ▷ **sample** kerning value
Generate letters:
  $\Lambda \leftarrow \{\}$
  for $i = 1, \ldots, \nu$ **do**
    $\lambda \sim p(\lambda)$ ▷ **sample** letter identity
    $\Lambda \leftarrow \text{append}(\Lambda, \lambda)$
Render:
  $\gamma \leftarrow \text{render}(\Lambda, \kappa)$
  $\pi \sim p(\pi)$ ▷ **sample** noise parameters
  $\gamma \leftarrow \text{noise}(\gamma, \pi)$
  return γ



Philosophy of inference using PPL:

if you can generate it, you've understood it.

= (find the right paths through the simulator)



Inference using PPLs compared to Classification(*)

With classification you target a few latent variables

With PPL you target the full trace

Classification on multiple latent variables usually do not know any constraints between them.

With PPL all constraints are coded into the model
Any inference will always obey these constraints

Corollary: "interpretable ML". Inference give you full "posterior story" in a language you understand (i.e. the generative model you wrote)

* assuming you have PPL that can sample efficiently from $p(z|x)$

Example PPLs

Academic:
Church: [arxiv:1206.3255]
Anglican: [arxiv.org:1507.00996]

More aligned with industry tools:

Edward / TF Probability [arxiv:1701.03757] (TensorFlow)
Pyro: [arxiv.org:1810.09538] (PyTorch)

Two Challenges with PPLs

- sampling efficiently from posterior!
- integrating with existing generative models

Two Challenges with PPLs

Sampling from posterior:

Options:

- MCMC (e.g. Metropolis Hastings)
  - Markov chain with trace-trace transitions
  - propose jumps from z $\rightarrow$z', accept / reject

- Variational Inference
  - approximate $p(z|x) \sim q(z, \alpha)$
    find optimal variational parameters
  - when cheap gradients available can use stochastic var. inference (SVI).
    - inference $\rightarrow$ optimization
  - main approach e.g. in Pyro

Two Challenges with PPLs

Sampling from posterior:

Importance Sampling

idea: $$\int \mathrm{d}z\ f(z)p(z|x)dz = \int \mathrm{d}z\ f(z)\frac{p(z|x)}{q(z)}q(z) = \int \mathrm{d}z\ f(z)w(z)q(z)$$

instead of using hard-to-sample w(x) use easy to sample q(x) and reweight.

Only requirement: need to be able to evaluate joint.

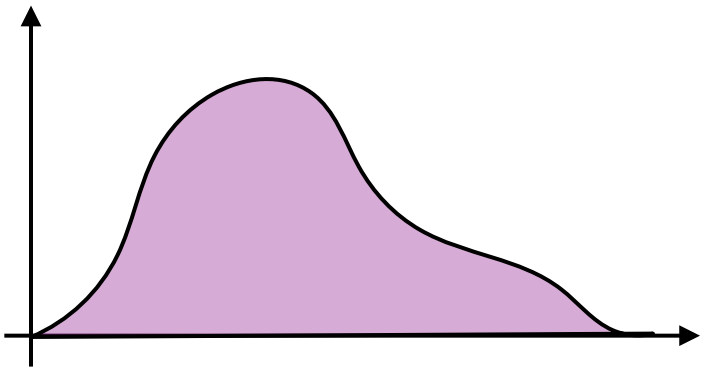
$$w(z_i) = \frac{p(z_i|x_i)}{q(z_i)} = \frac{1}{\sum_i p(z_i|x_i)/q(z_i)}\frac{p(x_i, z_i)}{q(z_i)}$$

Importance Sampling (cont'd):

need to find good proposal

basic requirements:

- same support as p(z)
- sufficiently flexible to model true posterior well,
  → reasonable weights



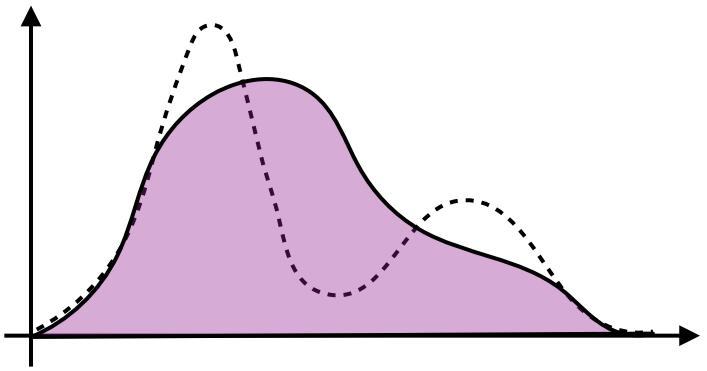
→ use ML to define proposals

Importance Sampling (cont'd):

need to find good proposal

basic requirements:

- same support as p(z)
- sufficiently flexible to model true posterior well,
  → reasonable weights



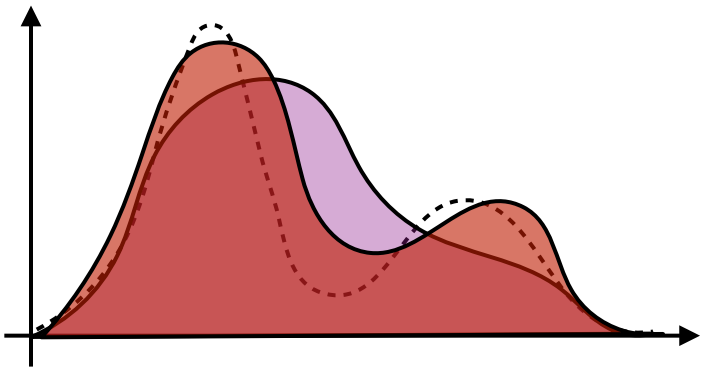
→ use ML to define proposals

Importance Sampling (cont'd):

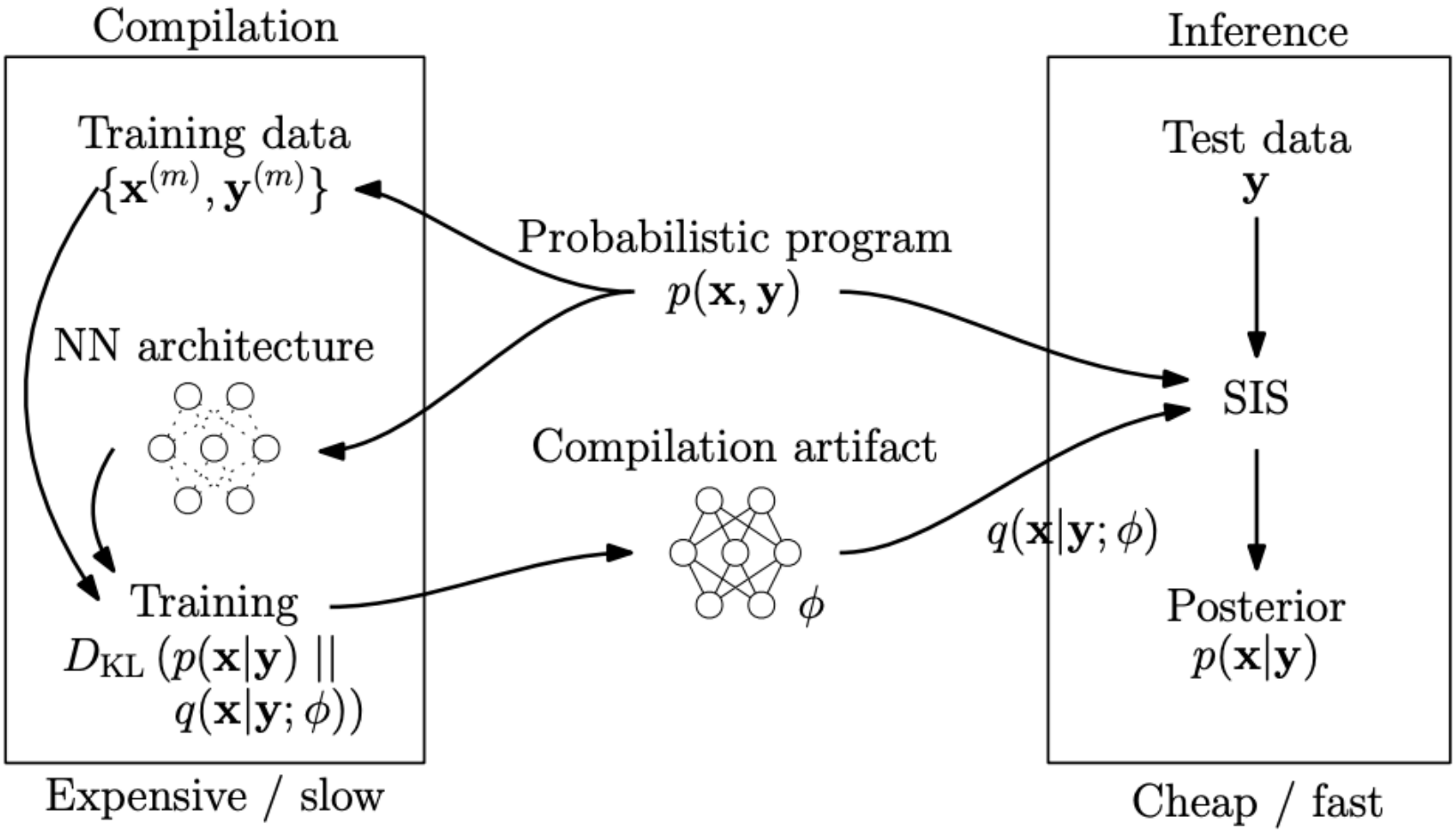
need to find good proposal

basic requirements:

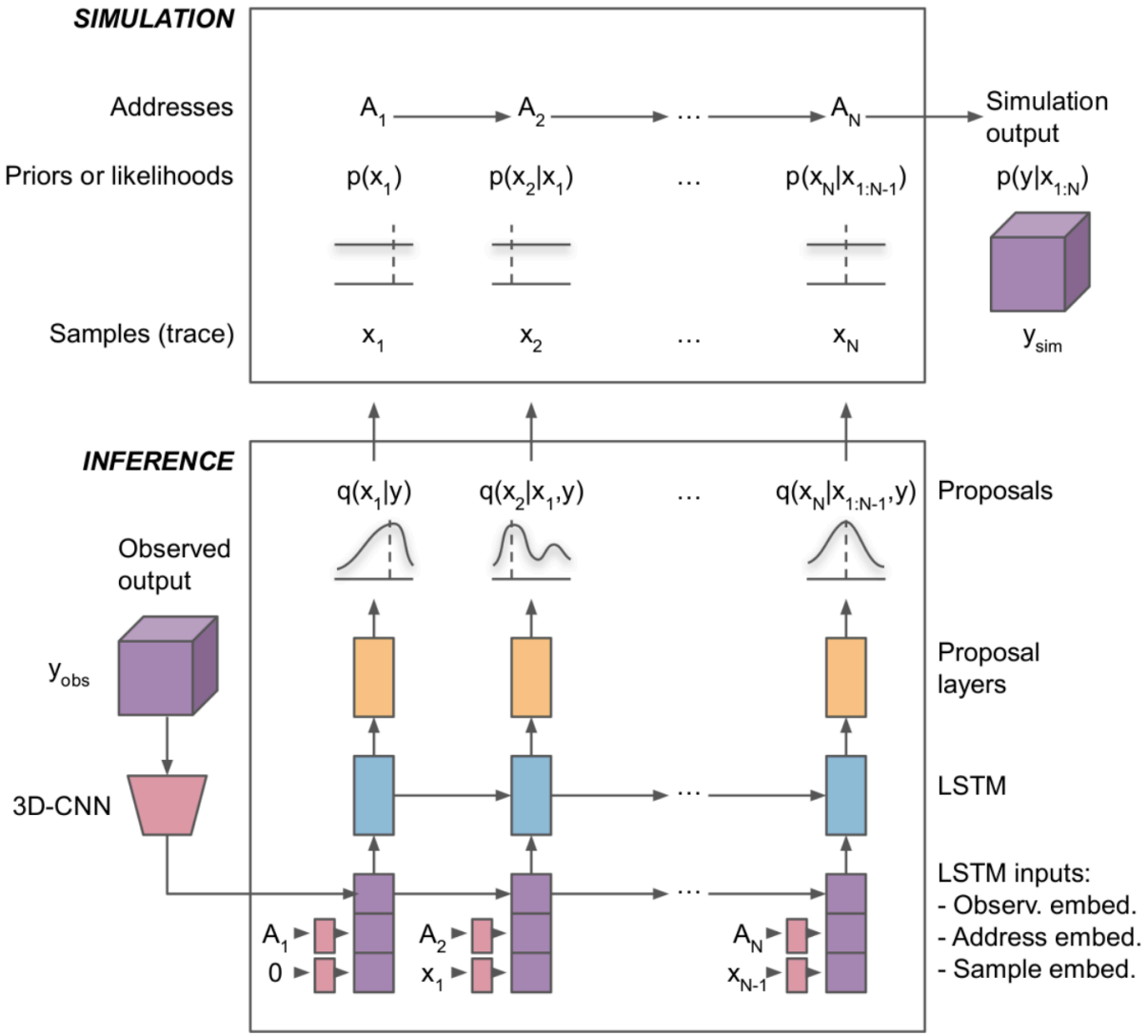
- same support as p(z)
- sufficiently flexible to model true posterior well
  → possibly multimodal...
  → reasonable weights



→ use ML to define good proposals


Compilation
Training data
$\{\mathbf{x}^{(m)}, \mathbf{y}^{(m)}\}$
NN architecture
Training
$D_{\mathrm{KL}}\left(p(\mathbf{x}|\mathbf{y}) \,||\, q(\mathbf{x}|\mathbf{y};\phi)\right)$
Expensive / slow
Probabilistic program
$p(\mathbf{x}, \mathbf{y})$
Compilation artifact
ϕ
Inference
Test data
$\mathbf{y}$
SIS
$q(\mathbf{x}|\mathbf{y};\phi)$
Posterior
$p(\mathbf{x}|\mathbf{y})$
Cheap / fast

Idea: each sample statement is paired with a flexible proposal distribution $q(\eta)$



So where is the particle phyics

ML<u>HEP</u>?

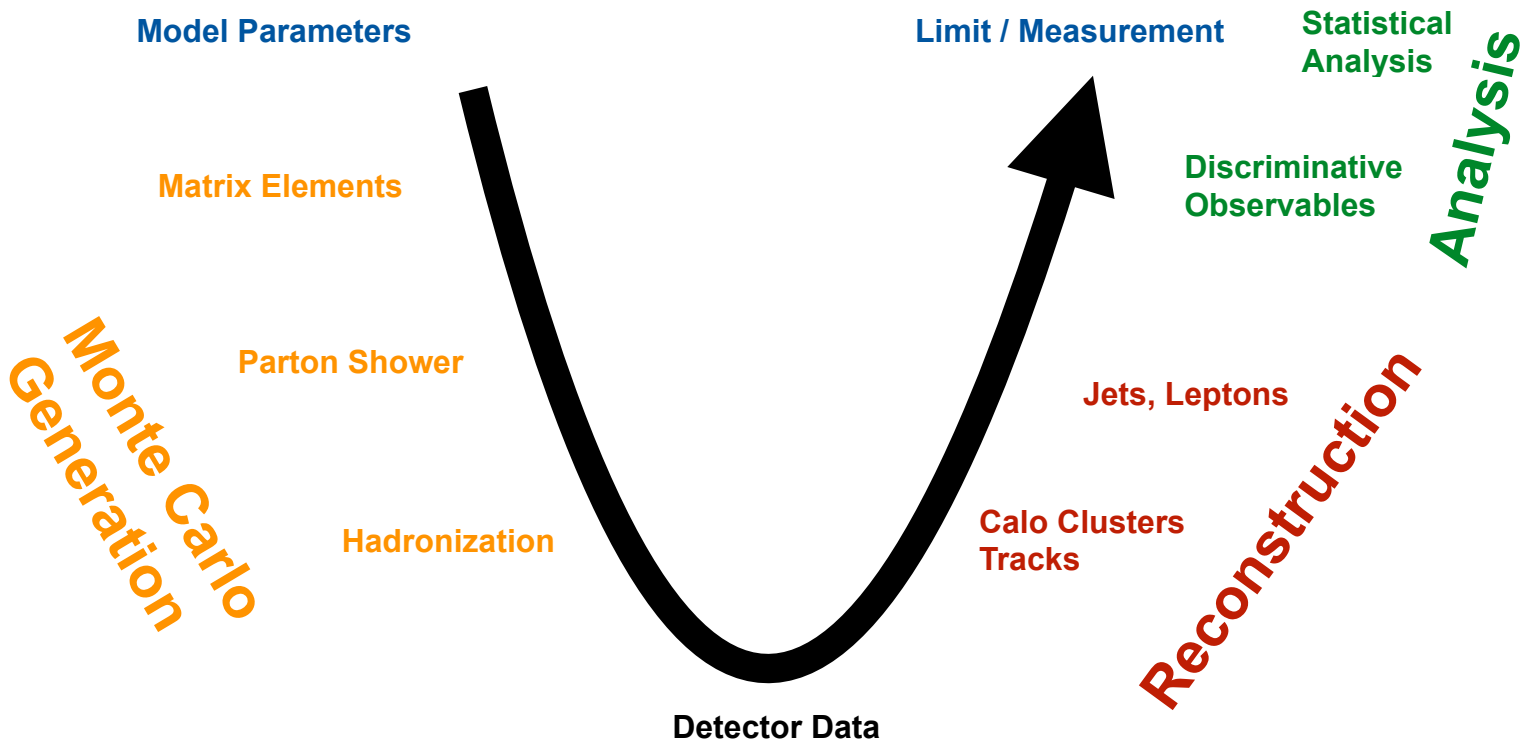
Context:

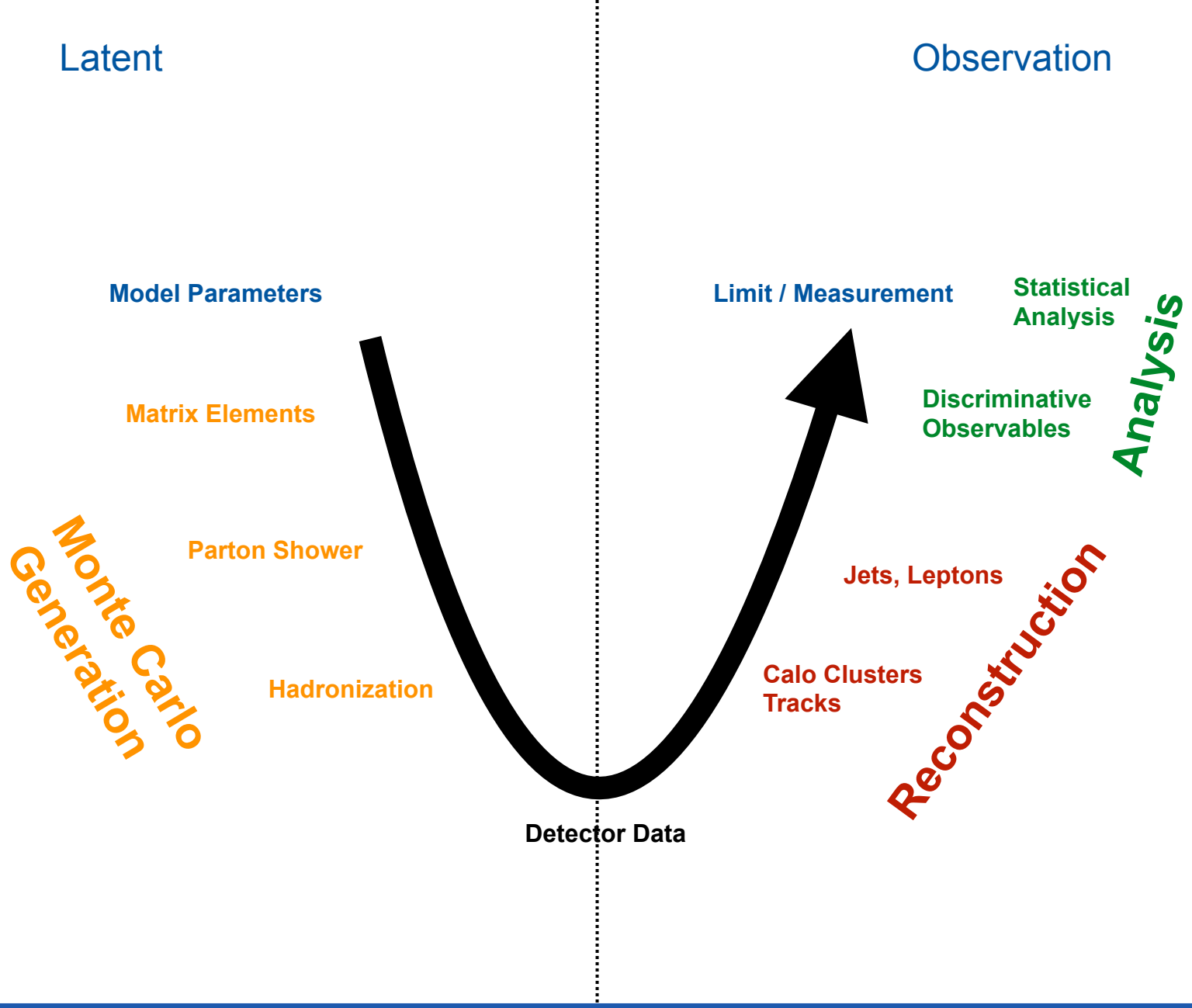
the concept of stochastic generators is very familiar to us

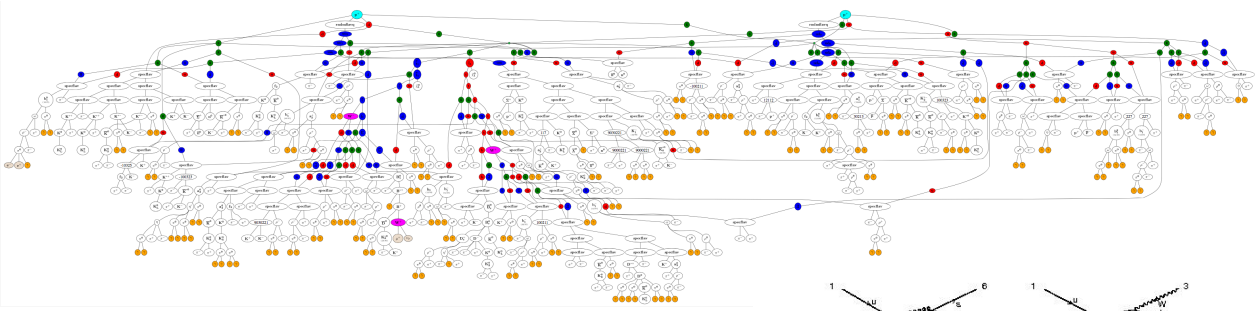
- **Monte Carlo generators**
- **detector generation**
- **Momentum smearing / Transfer functions**

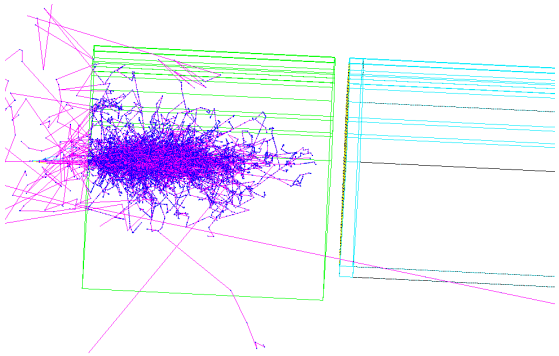
... HEP: the prototypical likelihood free inference problem

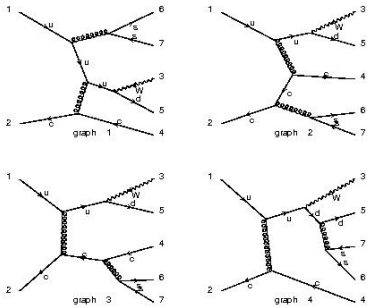
- **a lot of rich latent structure that is hard to learn from scratch**


Latent
Observation
Model Parameters
Limit / Measurement
Statistical Analysis
Analysis
Matrix Elements
Discriminative Observables
Monte Carlo Generation
Parton Shower
Jets, Leptons
Reconstruction
Hadronization
Calo Clusters
Tracks
Detector Data


Latent
Observation
Model Parameters
Matrix Elements
Parton Shower
Hadronization
Monte Carlo Generation
Limit / Measurement
Statistical Analysis
Discriminative Observables
Analysis
Jets, Leptons
Calo Clusters
Tracks
Reconstruction
Detector Data







Madgraph

Pythia

Sherpa

Geant

$$\int \mathrm{d}z_d \mathrm{d}z_h \mathrm{d}z_p \; p(x|z_d)\; p(z_d|z_h)\; p(z_h|z_p)\; p(z_p|\theta)$$

... but they're not implemented in PPLs

- standard programming languages

usually we do not make extensive use of "trace"

- pick individual latent vars ("MC Truth") as labels for calssification tasks

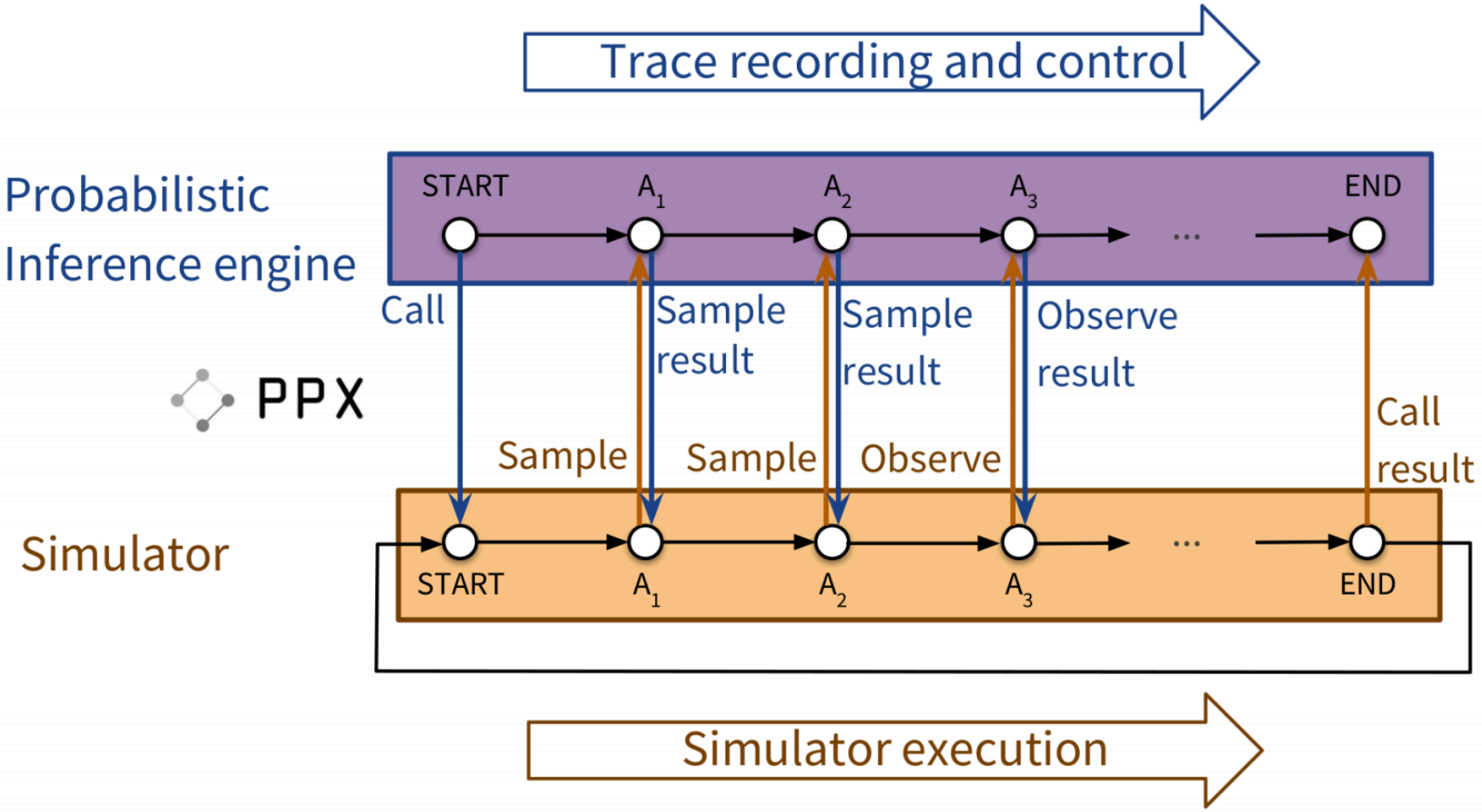
New PPL designed to interface existing simulators:

`pyprob`

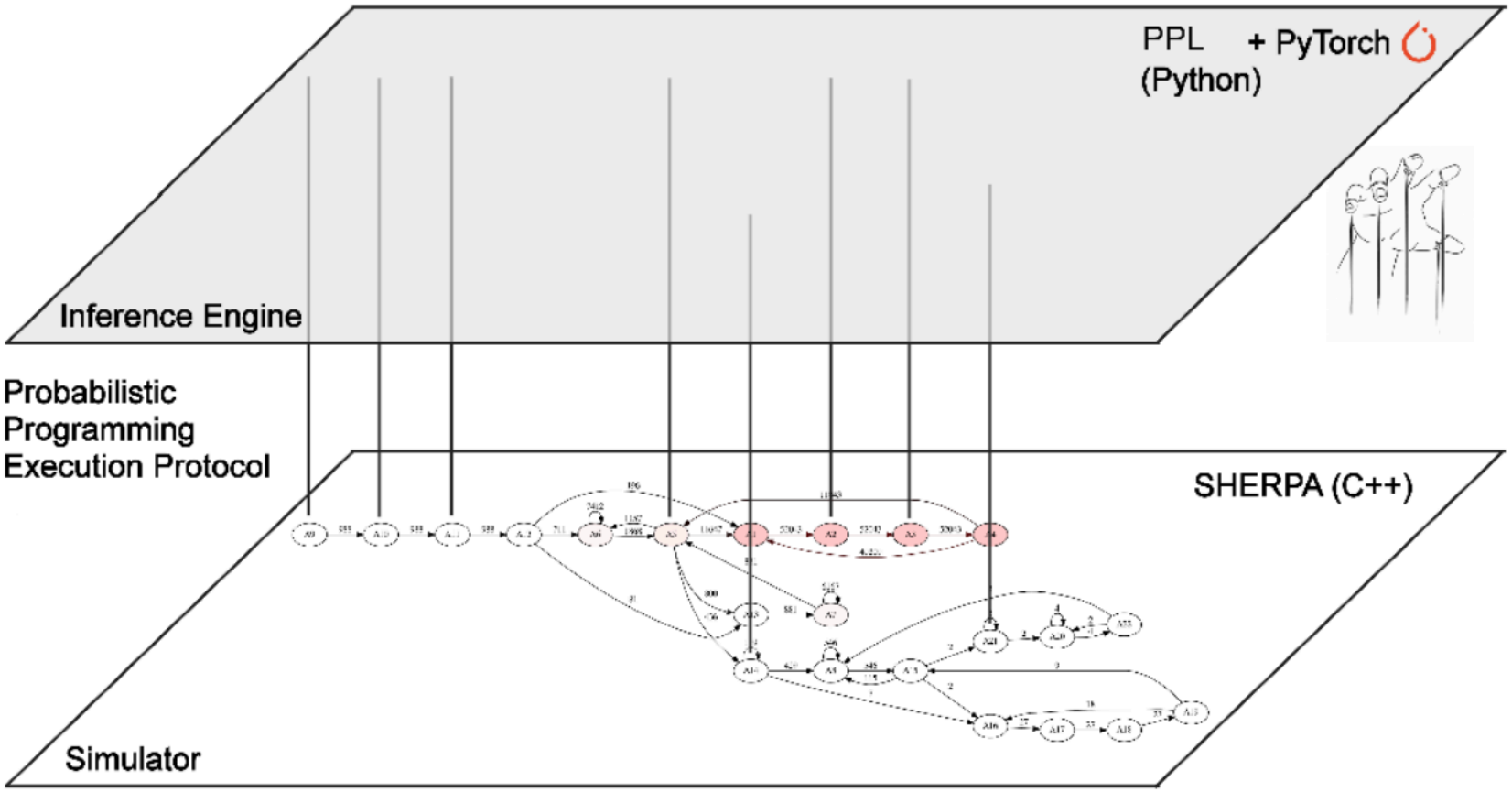
Idea:

intercept RNG calls from existing generators and hook it up to a PPL engine: Instead of random numbers, serve proposals compatible with an observation.

- automatic address ID (cf: other PPLs requiring naming)
- generic host language: comms over message queue

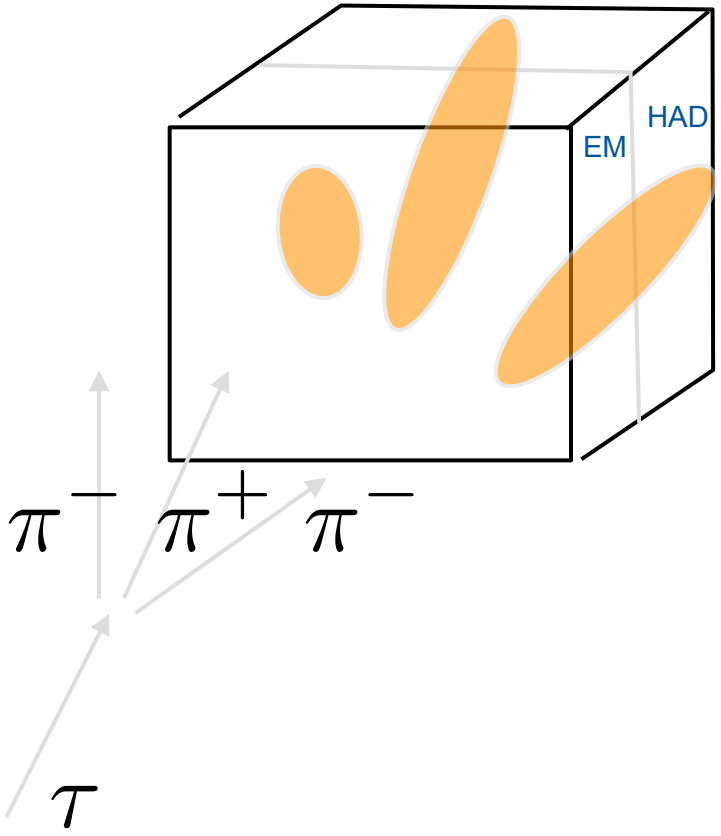

Trace recording and control
Probabilistic
Inference engine
PPX
START
A1
A2
A3
END
Call
Sample result
Sample result
Observe result
Call result
Sample
Sample
Observe
Simulator
START
A1
A2
A3
END
Simulator execution

A toy problem: Hijacking SHERPA


PPL
(Python)
+ PyTorch
Inference Engine
Probabilistic
Programming
Execution Protocol
SHERPA (C++)
Simulator

Setup

SHERPA tau gun + simplified calorimeter

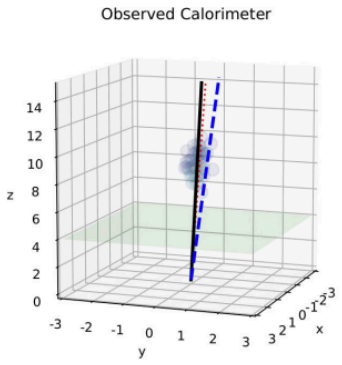


random elements:

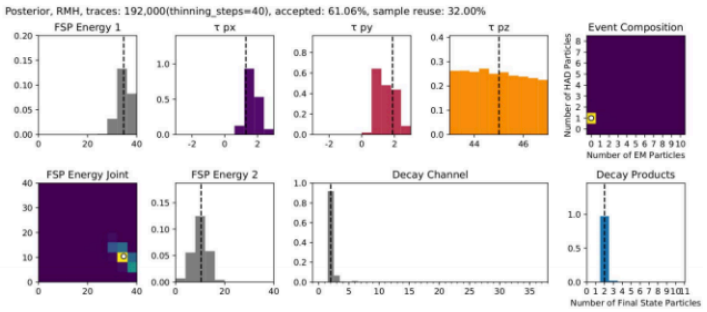
- **decay channel**
- **momentum assignment**
- **shower parameters**

complex structure: event composition, 4-vector conservation, ...

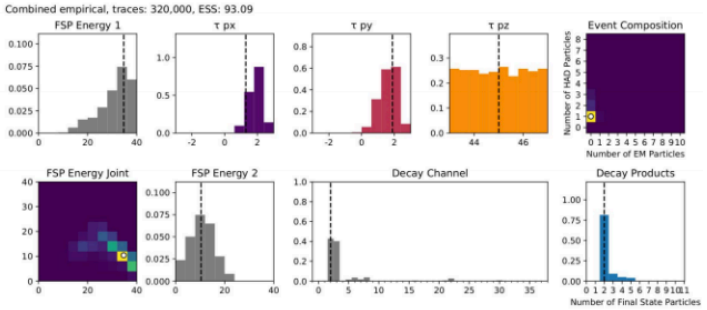
Comparing to true posterior from burned-in MCMC



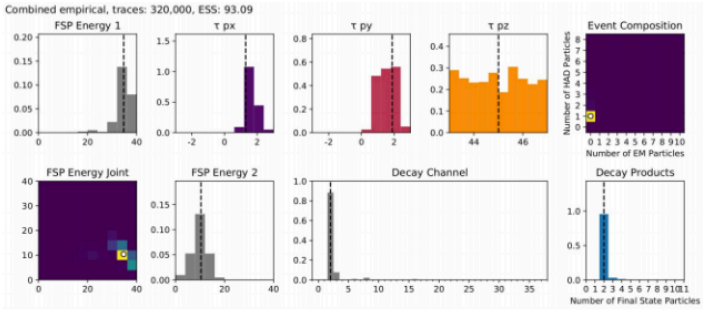
MCMC true posterior
(7.7M single node)



IC proposal
from trained NN

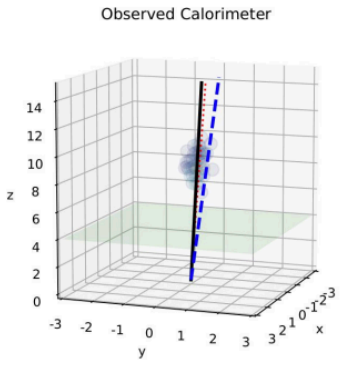


IC posterior
after importance
weighting

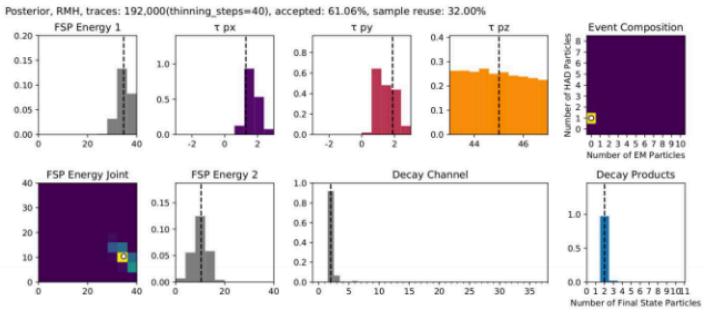


Every generated "posterior" trace is 100% physics compatible

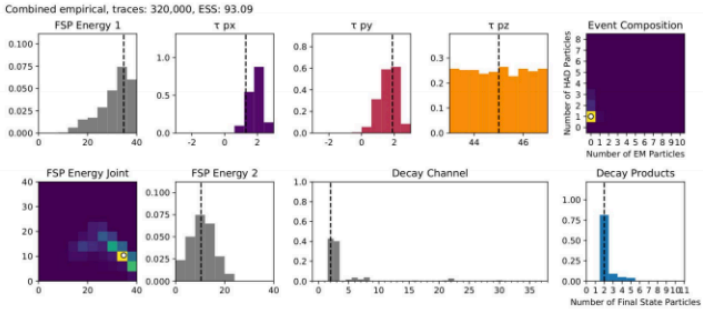
Comparing to true posterior from burned-in MCMC



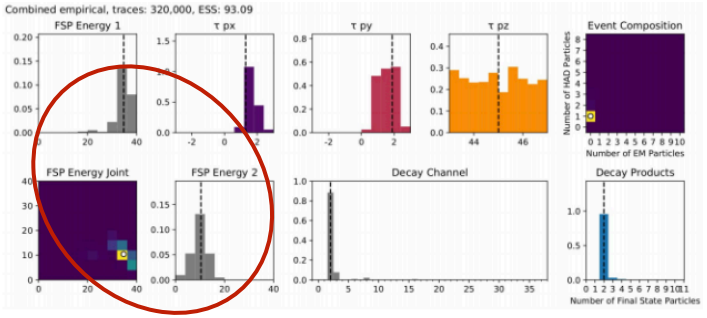
MCMC true posterior
(7.7M single node)



IC proposal
from trained NN

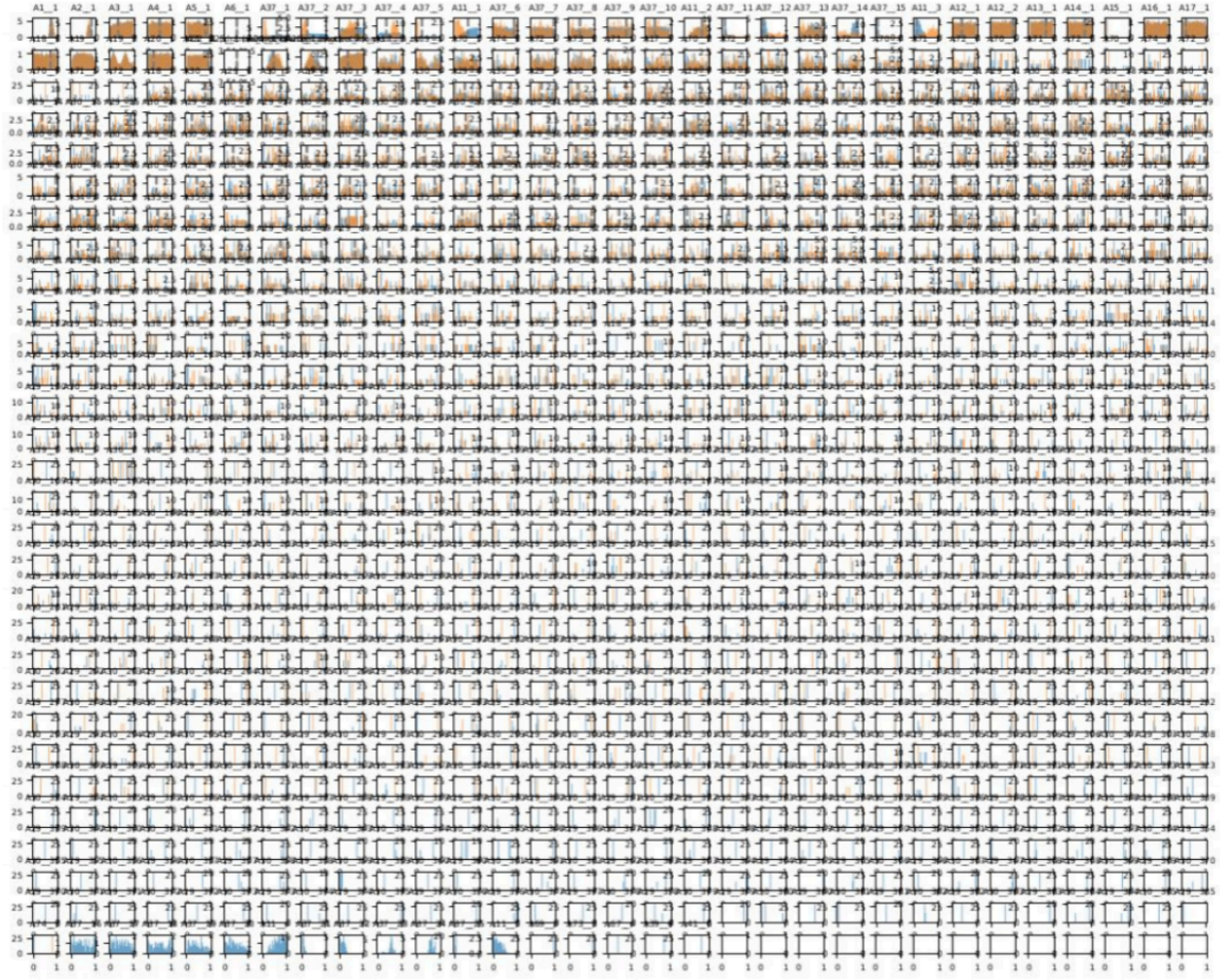


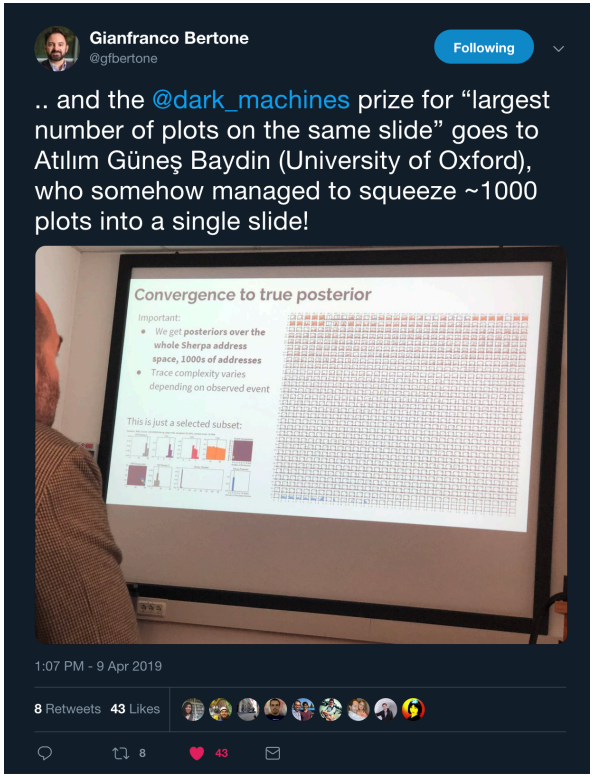
IC posterior
after importance
weighting



Every generated "posterior" trace is 100% physics compatible

Comparing to true posterior from burned-in MCMC

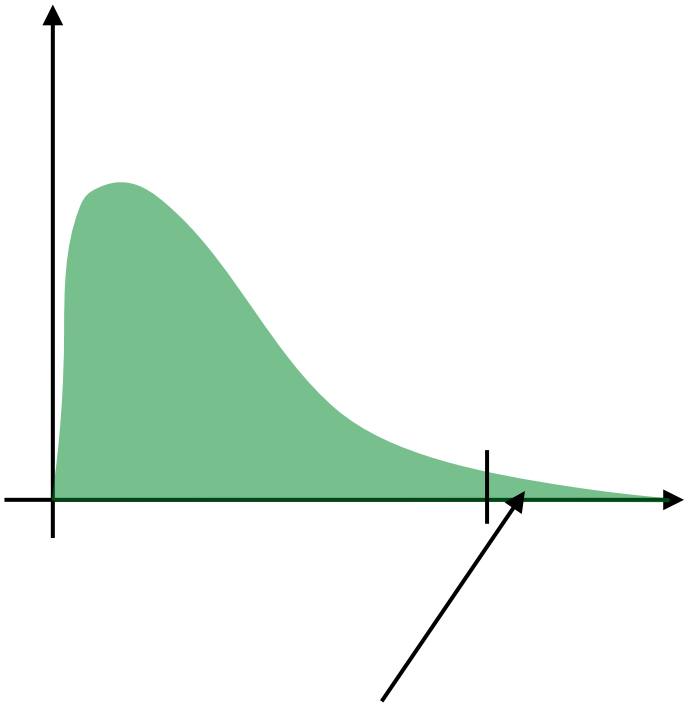




Other Use Cases

Anomaly analysis / complex conditions

Scenario:

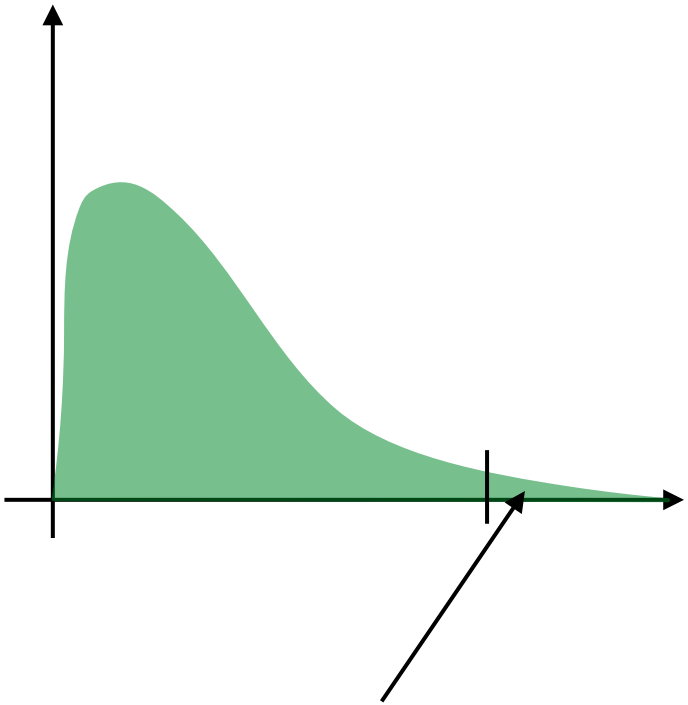


what type of events populate this tail?

Often hard to tell, need to generate huge MC to have sufficient statistics?
Can we do better?

Anomaly analysis / complex conditions

Scenario:

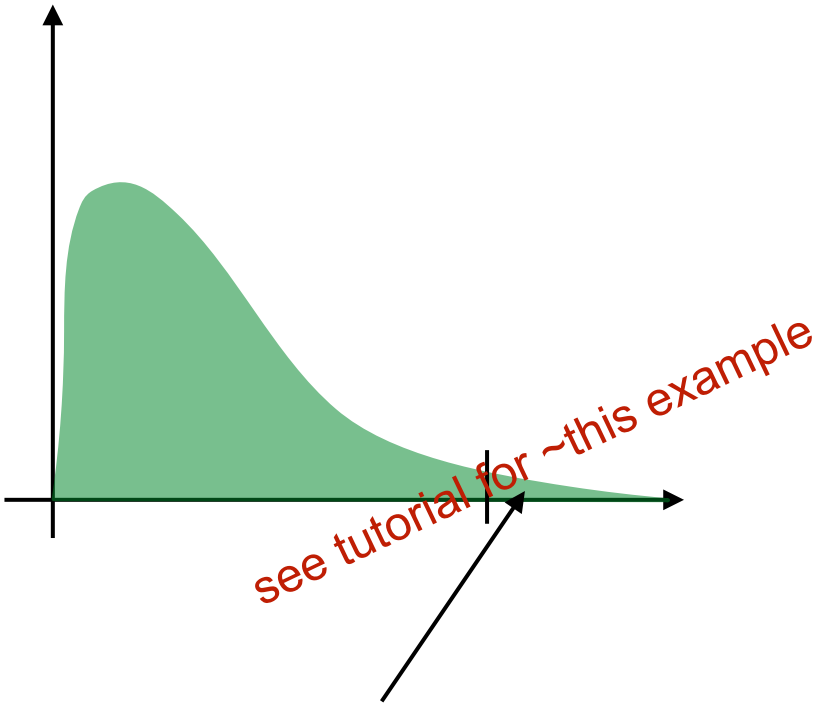


what type of events populate this tail?

use probprog to generate specifically p(z| is in tail) efficiently

Anomaly analysis / complex conditions

Scenario:



what type of events populate this tail?

use probprog to generate specifically p(z| is in tail) efficiently

Summary

probabilistic programming: universal (turing) complete way to build probability models using extended prog. langs.

use internal structure to automate inference

use ML to run inference efficiently

new PPLs can interface seamlessly with existing simulators. Do not need to re-write.