Efficient Event Generation with Normalizing Flows

— IML Machine Learning Working Group Meeting —

Claudius Krause

Fermi National Accelerator Laboratory

September 8, 2020

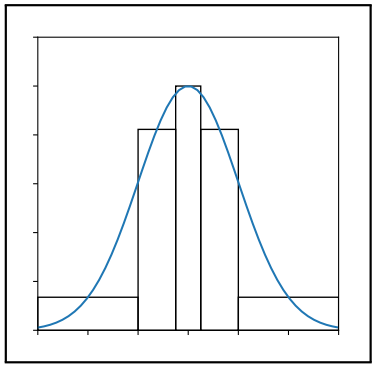
Unterstützt von / Supported by

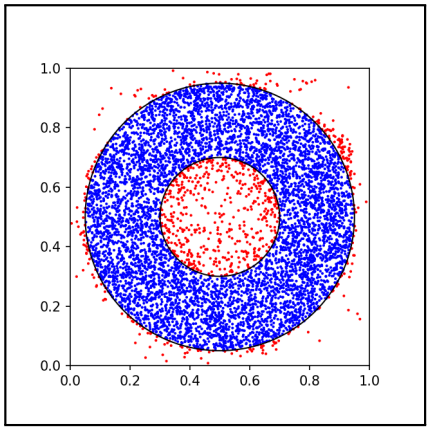
Alexander von Humboldt
Stiftung/Foundation

In collaboration with: Christina Gao, Stefan Höche, Joshua Isaacson, Holger Schulz
arXiv: 2001.05486, ML:ST and arXiv: 2001.10028, PRD
gitlab.com/i-flow/i-flow

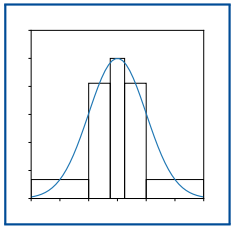
Efficient Event Generation with Normalizing Flows

Part I: Monte Carlo Integration with Importance Sampling





Part II: `i-flow` and its Applications



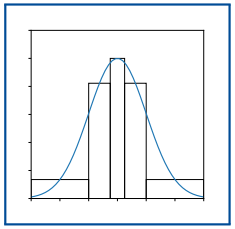
I: Importance Sampling is an efficient tool to estimate an integral.

$$\int_0^1 f(\vec{x})\, d\vec{x} \quad \xrightarrow{\text{MC}} \quad \frac{1}{N}\sum_i f(\vec{x}_i) \qquad \vec{x}_i \ldots \text{uniform}$$

$$= \int_0^1 \frac{f(\vec{x})}{q(\vec{x})}\, q(\vec{x}) d\vec{x} \quad \xrightarrow[\text{importance sampling}]{\text{MC}} \quad \frac{1}{N}\sum_i \frac{f(\vec{x}_i)}{q(\vec{x}_i)} \qquad \vec{x}_i \ldots q(\vec{x})$$

We therefore have to find a $q(\vec{x})$ that approximates the shape of $f(\vec{x})$ and is "easy" enough such that we can sample from its inverse cdf.

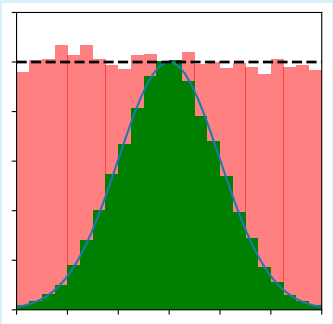
$\Rightarrow$ This $q(\vec{x})$ is also important for event generation!



I: The unweighting efficiency measures the quality of the approximation $q(\vec{x})$.

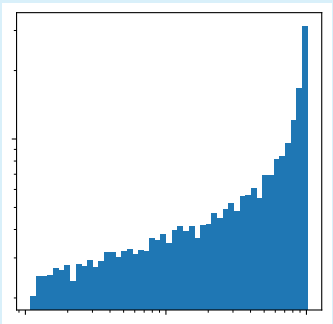
- If $q(\vec{x}) = \text{const.}$, each event $\vec{x}_i$ would require a weight of $f(\vec{x}_i)$ to reproduce the distribution of $f(\vec{x})$. $\Rightarrow$ "Weighted Events"
- If $q(\vec{x}) \propto f(\vec{x})$, all events would have the same weight as the distribution reproduces $f(\vec{x})$ directly. $\Rightarrow$ "Unweighted Events"

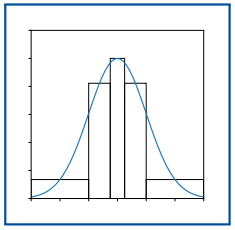
- To unweight, we need to accept/reject each event with probability $\frac{f(\vec{x}_i)}{\max f(\vec{x})}$. The resulting set of kept events is unweighted and reproduces the shape of $f(\vec{x})$.



- The unweighting efficiency η gives the fraction of events that "survives" this procedure.

$$\eta = \frac{\#\text{ accepted events}}{\#\text{ all events}} = \frac{\text{mean } w}{\max w}, \text{ with } w_i = \frac{f(\vec{x}_i)}{q(\vec{x}_i)}.$$

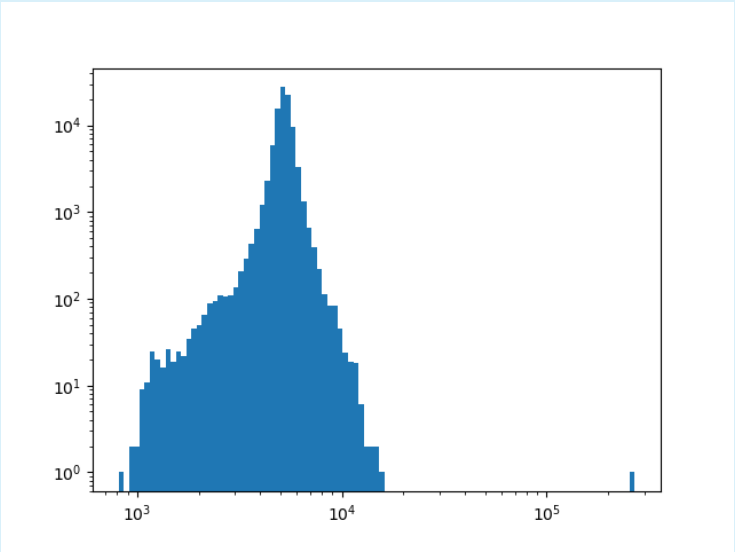




I: The usual definition of unweighting efficiency is unstable if many events are generated.

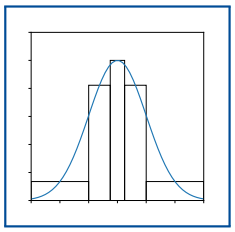
Problems of the old definition:

- The maximum grows with the number of events drawn.
- If more points are drawn than used in training, the chance for outliers increases a lot.
- Generating smaller subsets doesn't work, because we want a globally unweighted set of events.



Our new definition:

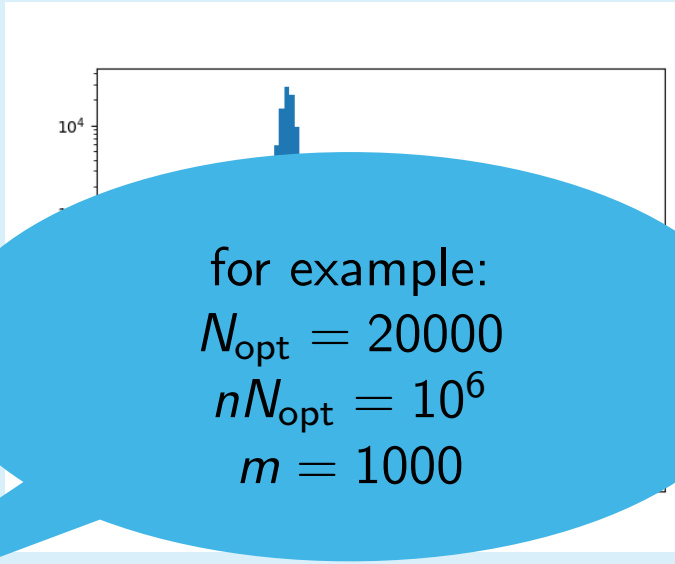
- Assuming we used N_{opt} events during optimization, draw nN_{opt} events.
- Now, select m replicas of N_{opt} events each and find their maximum weight.
- Compute the total maximum as the median of the individual maxima.
- We expect a few overweight events that can either be discarded or included with their weights set to w_{max} (Requiring further control plots!).



I: The usual definition of unweighting efficiency is unstable if many events are generated.

Problems of the old definition:

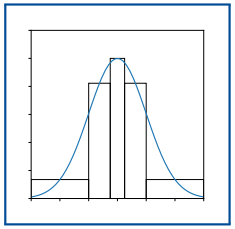
- The maximum grows with the number of events drawn.
- If more points are drawn than used in training, the chance for outliers increases a lot.
- Generating smaller subsets doesn't work, because we want a globally unweighted set of events.



for example:
$N_{\text{opt}} = 20000$
$nN_{\text{opt}} = 10^6$
$m = 1000$

Our new definition:

- Assuming we used N_{opt} events during optimization, draw nN_{opt} events.
- Now, select m replicas of N_{opt} events each and find their maximum weight.
- Compute the total maximum as the median of the individual maxima.
- We expect a few overweight events that can either be discarded or included with their weights set to w_{max} (Requiring further control plots!).

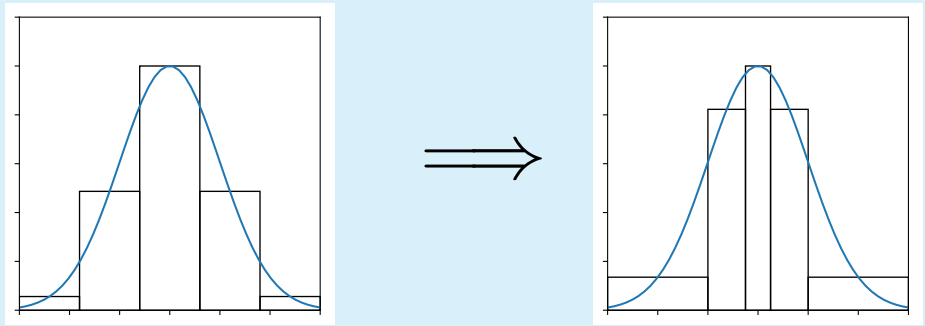


I: The VEGAS algorithm is very efficient.

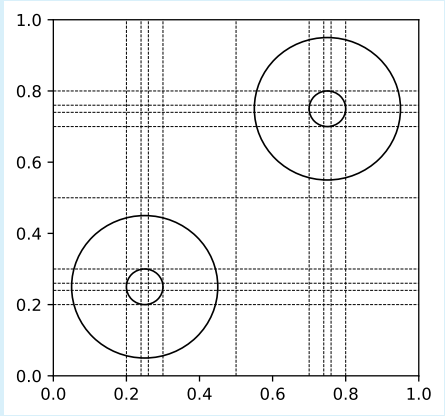
The VEGAS algorithm

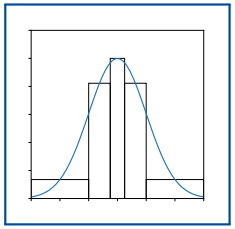
Peter Lepage 1980

- assumes the integrand factorizes and bins the 1-dim projection.
- then adapts the bin edges such that area of each bin is the same.



- It does have problems if the features are not aligned with the coordinate axes.
- The current python implementation also uses stratified sampling.



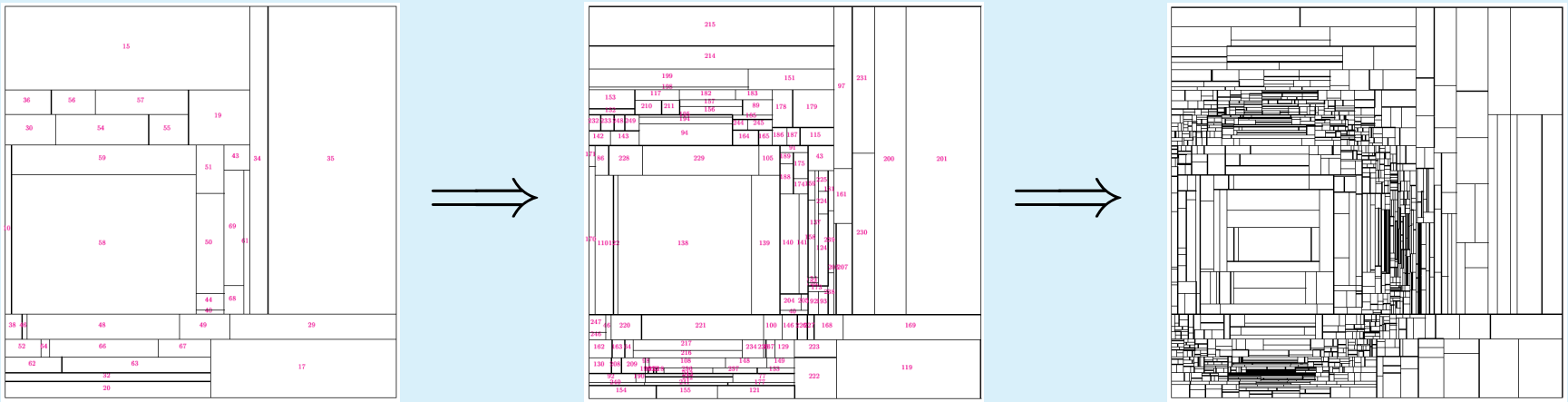


I: The Foam algorithm resolves correlations.

The Foam algorithm

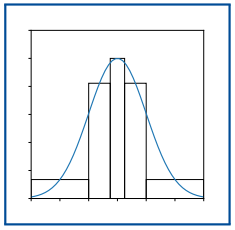
S. Jadach [physics/0203033]

- In the exploration phase, the integration domain is consecutively split into cells.
- In the generation phase, a cell is chosen at random and a point is drawn uniformly from within that cell.



illustrations from ICHEP 2002 slides, S. Jadach

- It captures correlations, but within each cell $q(\vec{x}) = \text{const}$.
- In addition, the exploration phase requires many functional calls.



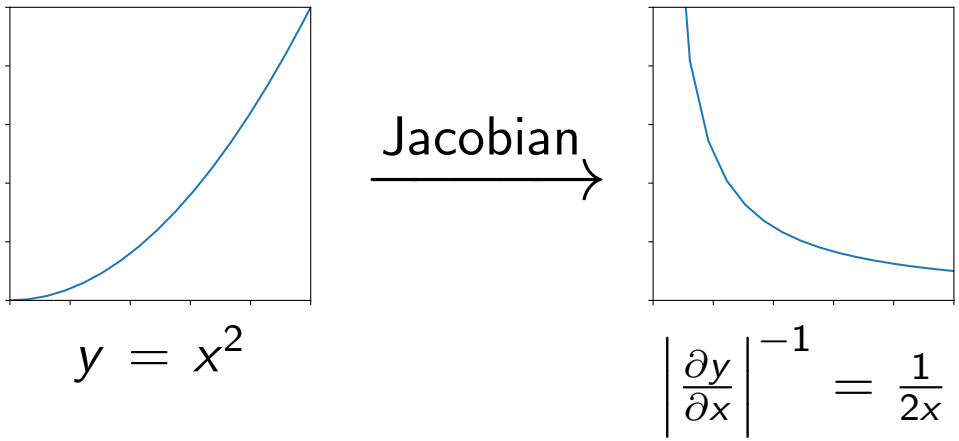
I: Neural Networks can find better $q(\vec{x})$.

We could use the NN as nonlinear coordinate transform:

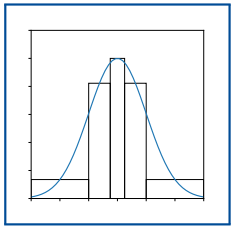
- We use a deep NN with n_{dim} nodes in the first and last layer to map a uniformly distributed $\vec{x}$ to a target $q(\vec{x})$.
- The distribution induced by the map $\vec{y}(\vec{x})$ (=NN) is given by the Jacobian of the map:

$$q(\vec{y}) = q(\vec{y}(\vec{x})) = \left|\frac{\partial \vec{y}}{\partial \vec{x}}\right|^{-1}$$

Klimek/Perelstein [arXiv:1810.11509]



$\Rightarrow$ The Jacobian is needed to evaluate the loss and to sample. However, it scales as $\mathcal{O}(n^3)$ and is too costly for high-dimensional integrands!



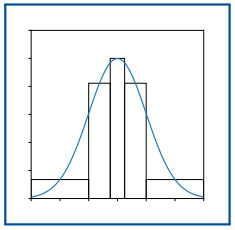
I: Normalizing Flows are numerically cheaper.

A Normalizing Flow:

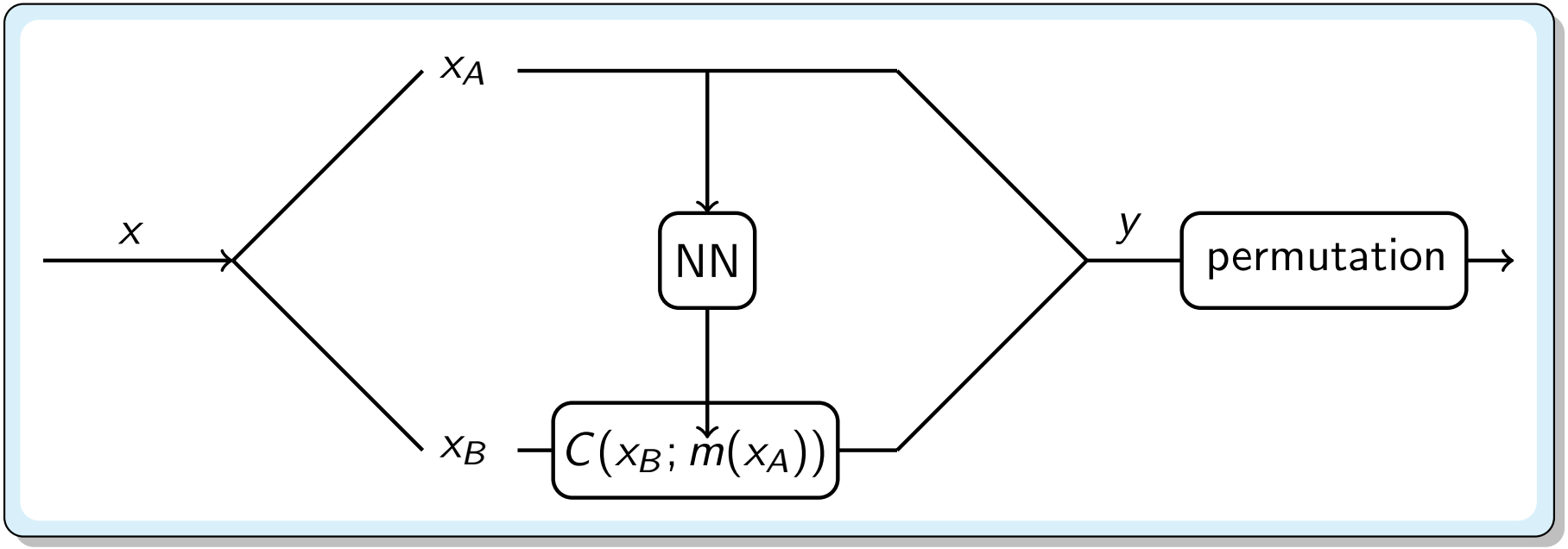
- is a bijective, smooth mapping between two statistical distributions.
- is composed of a series of easy transformations, the *"Coupling Layers"*.
- is still flexible enough to learn complicated distributions.

$\Rightarrow$ The NN does not learn the transformation, but the parameters of a series of easy transformations.

- The idea was introduced as "Nonlinear Independent Component Estimation" (NICE) in Dinh et al. [arXiv:1410.8516].
- In Rezende/Mohamed [arXiv:1505.05770], Normalizing Flows were first discussed with planar and radial flows.
- We follow the ideas of Müller et al. [arXiv:1808.03856], but with the modifications of Durkan et al. [arXiv:1906.04032].



I: The Coupling Layer is the fundamental Building Block.



forward:

$$y_A = x_A$$
$$y_{B,i} = C(x_{B,i}; m(x_A))$$

inverse:

$$x_A = y_A$$
$$x_{B,i} = C^{-1}(y_{B,i}; m(x_A))$$

The C are numerically cheap, invertible, and separable in $x_{B,i}$.

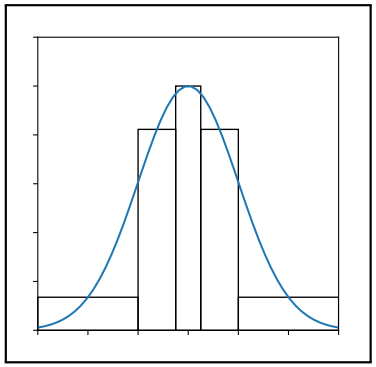
Jacobian:

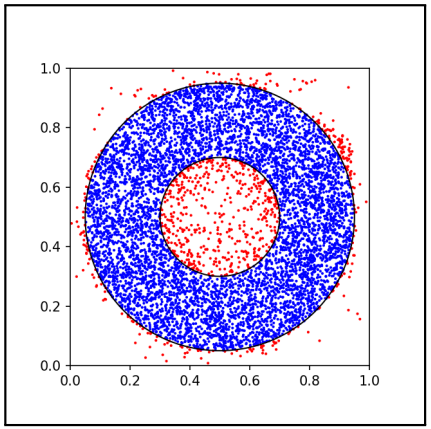
$$\left|\frac{\partial \vec{y}}{\partial \vec{x}}\right| = \begin{vmatrix} 1 & \frac{\partial C}{\partial x_A} \\ 0 & \frac{\partial C}{\partial x_B} \end{vmatrix} = \Pi_i \frac{\partial C(x_{B,i}; m(x_A))}{\partial x_{B,i}}$$

$$\Rightarrow \mathcal{O}(n)$$

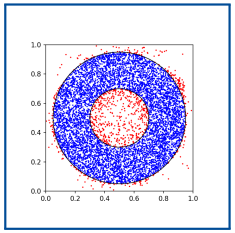
Efficient Event Generation with Normalizing Flows

Part I: Monte Carlo Integration with Importance Sampling





Part II: `i-flow` and its Applications



II: `i-flow`, a flexible Normalizing Flow Implementation.

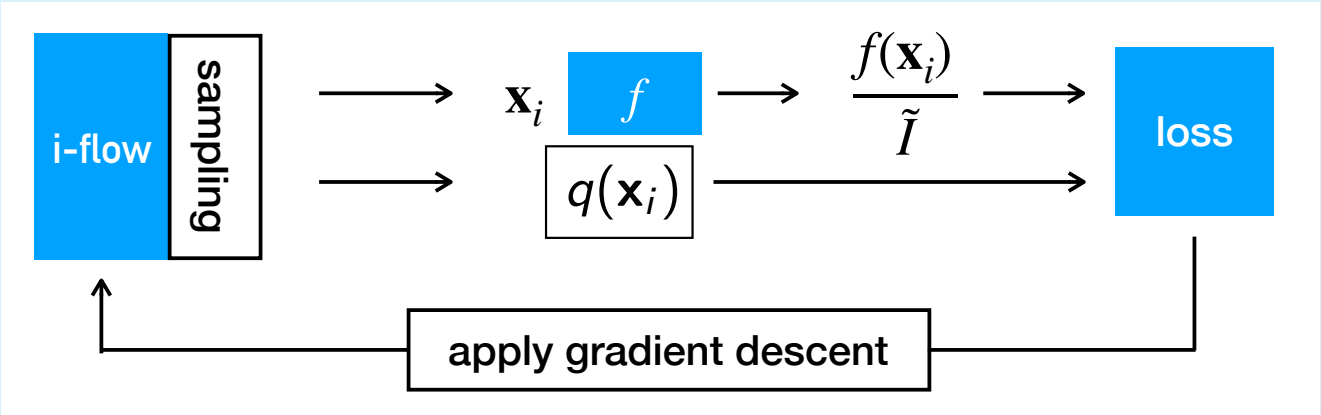
`i-flow` C. Gao, J. Isaacson, CK [arXiv:2001.05486, ML:ST]

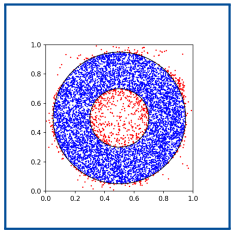
- implements Normalizing Flows in `python` using `TensorFlow 2.0`.
- is available at `gitlab.com/i-flow/i-flow`.

The user can choose different

- Transformations in the Coupling Layer,
- Loss functions,
- Neural Network architectures,
- Settings for hyperparameters.

How it works:

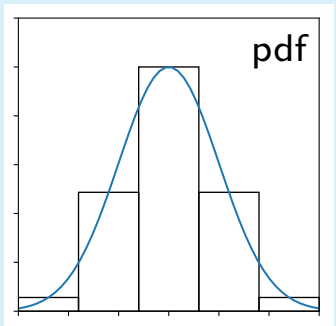


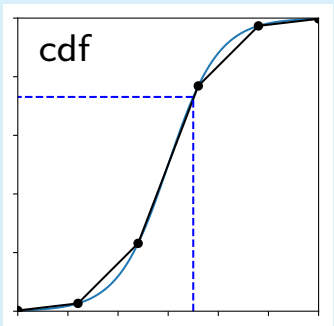


II: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]





$$C = \sum_{k=1}^{b-1} Q_k + \alpha Q_b$$

$$\alpha = \frac{x-(b-1)w}{w}$$

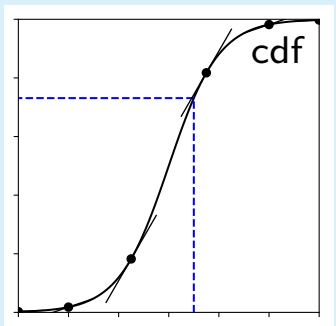
$$\left|\frac{\partial C}{\partial x_B}\right| = \prod_i \frac{Q_{b_i}}{w}$$

The NN predicts the pdf bin heights Q_i.

rational quadratic spline coupling function:

Durkan et al. [arXiv:1906.04032]

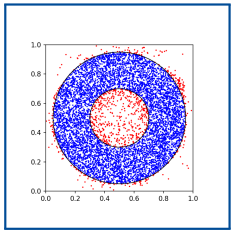
Gregory/Delbourgo [IMA Journal of Numerical Analysis, '82]



$$C = \frac{a_2\alpha^2 + a_1\alpha + a_0}{b_2\alpha^2 + b_1\alpha + b_0}$$

- still rather easy
- more flexible

The NN predicts the cdf bin widths, heights, and derivatives that go in a_i&b_i.



II: The Loss function quantifies our goal.

We have different choices:

- Kullback-Leibler (KL) divergence:

$$D_{KL} = \int p(x) \log \frac{p(x)}{q(x)} dx \qquad \approx \qquad \frac{1}{N} \sum \frac{p(x_i)}{q(x_i)} \log \frac{p(x_i)}{q(x_i)}, \qquad x_i \ldots q(x)$$

- Pearson χ^2 divergence:

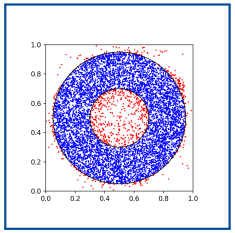
$$D_{\chi^2} = \int \frac{(p(x)-q(x))^2}{q(x)} dx \qquad \approx \qquad \frac{1}{N} \sum \frac{p(x_i)^2}{q(x_i)^2} - 1, \qquad x_i \ldots q(x)$$

- Exponential divergence:

$$D_{exp} = \int p(x) \left(\log \frac{p(x)}{q(x)} \right)^2 dx \qquad \approx \qquad \frac{1}{N} \sum \frac{p(x_i)}{q(x_i)} \left(\log \frac{p(x_i)}{q(x_i)} \right)^2, \qquad x_i \ldots q(x)$$

We use the ADAM optimizer for stochastic gradient descent:

- The learning rate for each parameter is adapted separately, but based on previous iterations.
- This is effective for sparse and noisy functions. Kingma/Ba [arXiv:1412.6980]

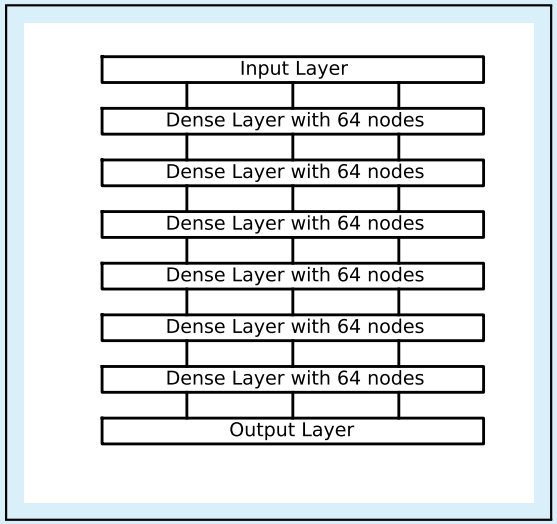


II: There are many hyperparameters to adjust.

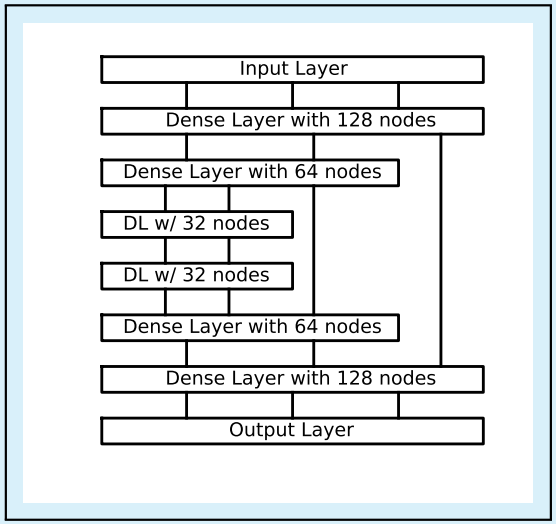
Available Architectures:

Müller et al. [arXiv:1808.03856]

"Fully Connected" Neural Net (NN):

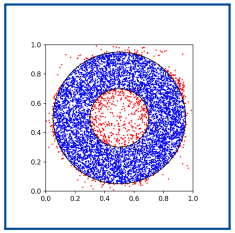


"U-shaped" Neural Net (Unet):



There are additional hyperparameters that can be adjusted:

- learning schedule: schedule function (const., exponential, ...), initial learning rate, decay rate and step size, ...
- training: which loss function, # epochs, # samples per epoch
- normalizing flow specific: # (input/output) bins, how to split dims inside CL, # CLs, which function in the CLs



II: We need $\mathcal{O}(\log n)$ Coupling Layers.

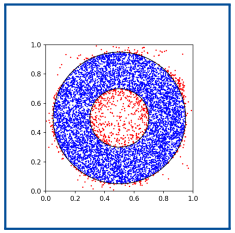
How many Coupling Layers do we need?

- Enough to learn all correlations between the variables.
- As few as possible to have a fast code.

- This depends on the applied permutations and the $x_A - x_B$-splitting:
  (pppttt)↔(tttppp) *vs.* (pppptt)↔(pptttpp)↔(ttpppp)

- More pass-through dimensions (p) means more points required for accurate loss.
- Fewer pass-through dimensions means more CLs needed.

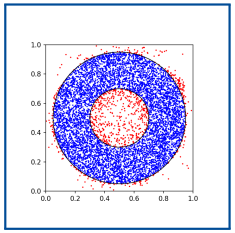
- For $\#p \approx \#t$, we argue that $\boxed{4 \leq \#CLs \leq 2\lceil \log_2 n_{dim} \rceil}$ is sufficient



II: `i-flow` is best at high dimensions.

	Dim	VEGAS	Foam	i-flow
Gaussian	2	**164, 436**	6, 259, 812	2, 310, 000
	4	**631, 874**	24, 094, 679	2, 285, 000
	8	**1, 299, 718**	$> 50, 000, 000$ †	3, 095, 000
	16	**2, 772, 216**	$> 50, 000, 000$ †	7, 230, 000
Camel	2	**421, 475**	5, 619, 646	2, 225, 000
	4	24, 139, 889	21, 821, 075	**8, 220, 000**
	8	$> 50, 000, 000$ †	$> 50, 000, 000$	**19, 460, 000**
	16	993, 294 †	$> 50, 000, 000$ †	32, 145, 000 †
Entangled circles	2	43, 367, 192	**17, 499, 823**	23, 105, 000
Annulus w. cuts	2	4, 981, 080 †	**11, 219, 498**	17, 435, 000
Scalar-top-loop	3	**152, 957**	5, 290, 142	685, 000
Polynomial $\sum_i -x_i^2 + x_i$	18	42, 756, 678	$> 50, 000, 000$	**585, 000**
	54	$> 50, 000, 000$	$> 21, 505, 000$ *	**685, 000**
	96	$> 50, 000, 000$ †	$> 10, 235, 000$ *	**1, 145, 000**

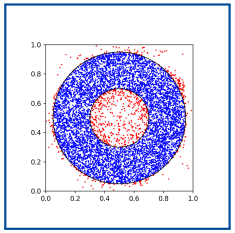
Table: Number of functional calls to reach a total relative uncertainty of 10^{-4} (for the first 11 cases) or 10^{-5} (for the last 3 cases). The integrator with the fewest functional calls is highlighted in boldface. A † indicates that the algorithm did not converge to the true integral value within 5 standard deviations, a * indicates cases where the algorithm ran out of memory before the cut-off was reached.



II: `i-flow` has a high accuracy.

	Dim	VEGAS	(pull)	Foam	(pull)	i-flow	(pull)
Gaussian	2	0.99925(10)	0.7	0.99925(10)	0.6	**0.99919(10)**	**0.1**
	4	0.99861(10)	2.4	**0.99835(10)**	**−0.2**	0.99841(10)	0.4
	8	0.99694(10)	1.9	0.99439(37) †	−6.4	**0.99684(10)**	**0.9**
	16	0.99357(10)	0.6	0.54986(235) †	−188	**0.99354(10)**	**0.4**
Camel	2	0.98175(10)	0.9	0.98163(10)	−0.3	**0.98165(10)**	**−0.1**
	4	0.96345(10)	−2.2	0.96361(10)	−0.5	**0.96365(10)**	**−0.02**
	8	0.92495(28) †	−13	0.92798(19) †	−3.5	**0.92843(9)**	**−2.2**
	16	0.43137(9)	−5001	0.76921(129) †	−72	**0.85940(9)**	**−34**
Ent. circles	2	0.0136798(14)	−3.6	**0.0136838(14)**	**−0.7**	0.0136829(14)	−1.4
Annulus	2	0.509813(51)	−14	0.510559(51)	1.0	**0.510511(51)**	**0.1**
Top-loop $\cdot 10^{10}$	3	1.93711(19)	0.7	**1.93708(19)**	**0.6**	1.93677(19)	−1.0
Polynomial $\sum_i -x_i^2 + x_i$	18	2.99989(3)	−3.6	2.99986(12) †	−1.1	**2.99997(3)**	**−1.1**
	54	8.99972(19) †	−1.5	9.00013(32) ∗	0.4	**9.00001(9)**	**0.2**
	96	0.15547(52) †	−30683	16.0004(3) ∗	1.7	**15.9998(2)**	**−1.2**

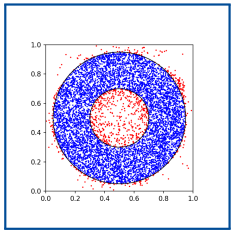
Table: Integral estimate and uncertainty together with their relative deviations ("pull"). A † indicates that the algorithm reached a cut-off of $5 \cdot 10^7$ function calls before the target uncertainty was reached, a $*$ indicates cases where the algorithm ran out of memory before the cut-off was reached.



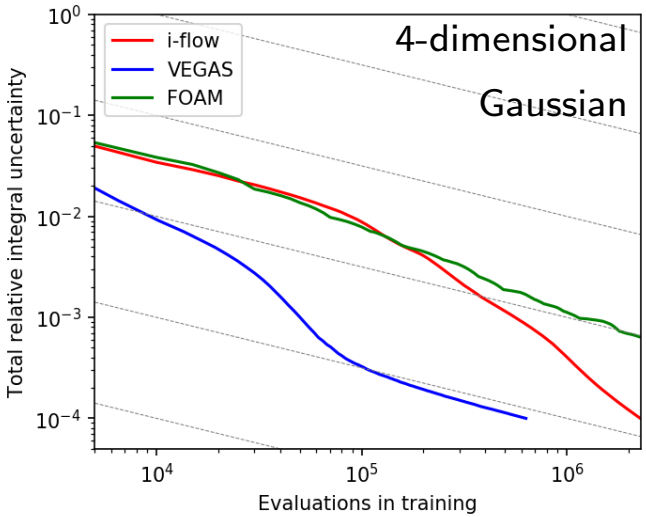
II: `i-flow` adapts well to the integrand.

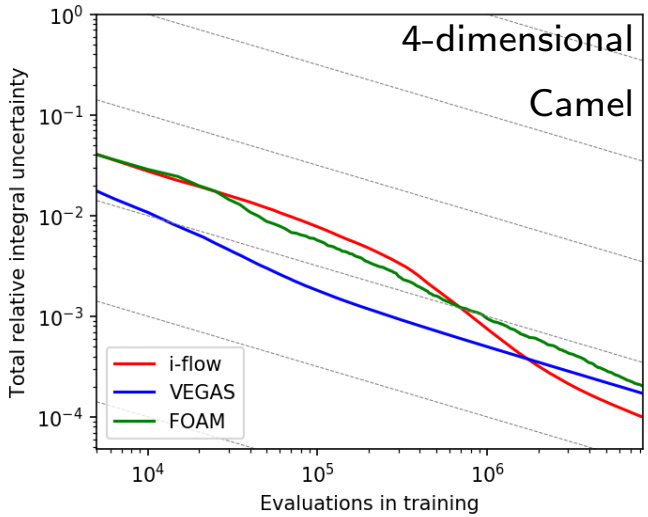
	Dim	VEGAS	Foam	i-flow
Gaussian	2	$\mathbf{7 \cdot 10^{-4}}$	$3 \cdot 10^{-3}$	$2 \cdot 10^{-3}$ $*$
	4	$\mathbf{1.5 \cdot 10^{-3}}$	$3 \cdot 10^{-3}$	$\mathbf{1.5 \cdot 10^{-3}}$ $*$
	8	$2.5 \cdot 10^{-3}$	$3 \cdot 10^{-2}$	$\mathbf{1.5 \cdot 10^{-3}}$ $*$
	16	$3.5 \cdot 10^{-3}$	$2 \cdot 10^{-2}$	$\mathbf{2.5 \cdot 10^{-3}}$ $*$
Camel	2	$\mathbf{2 \cdot 10^{-3}}$	$\mathbf{2 \cdot 10^{-3}}$	$\mathbf{2 \cdot 10^{-3}}$ $*$
	4	$8 \cdot 10^{-3}$	$1 \cdot 10^{-2}$	$\mathbf{4 \cdot 10^{-3}}$
	8	$4 \cdot 10^{-2}$	$1.6 \cdot 10^{-2}$	$\mathbf{5 \cdot 10^{-3}}$
	16	†	$1.5 \cdot 10^{-1}$	$\mathbf{5 \cdot 10^{-3}}$
Entangled circles	2	$1 \cdot 10^{-2}$	$\mathbf{4 \cdot 10^{-3}}$	$5 \cdot 10^{-3}$ $*$
Annulus w. cuts	2	$\mathbf{3 \cdot 10^{-3}}$	$4 \cdot 10^{-3}$ $*$	$5 \cdot 10^{-3}$
Scalar-top-loop	3	$7 \cdot 10^{-4}$	$\mathbf{5 \cdot 10^{-4}}$	$\mathbf{5 \cdot 10^{-4}}$ $*$
Polynomial $\sum_i -x_i^2 + x_i$	18	$1.5 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$ $*$	$\mathbf{8 \cdot 10^{-5}}$ $*$
	54	$3 \cdot 10^{-3}$	$9 \cdot 10^{-4}$ $*$	$\mathbf{8 \cdot 10^{-5}}$ $*$
	96	†	$8 \cdot 10^{-4}$ $*$	$\mathbf{1 \cdot 10^{-4}}$ $*$

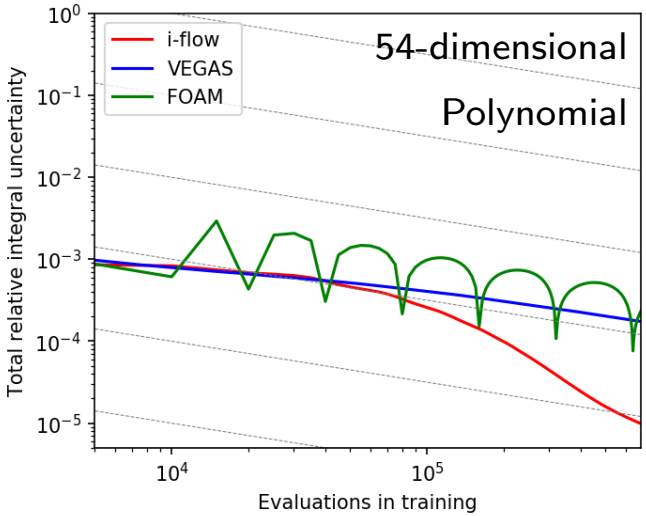
Table: Relative uncertainty on the integral estimate of the last iteration, based on a sample of 5000 points. A $*$ indicates when the value was still decreasing and had not yet converged, a † is in place where the algorithm did not converge to the true integrand.

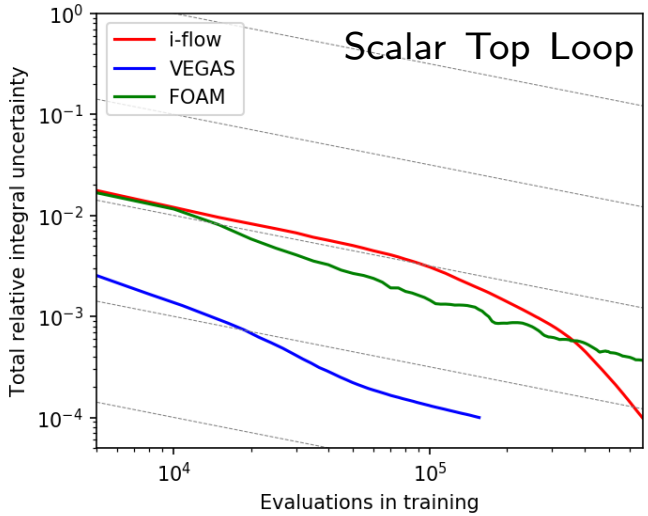


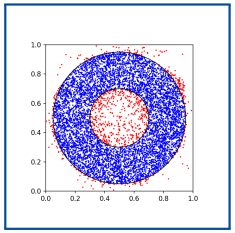
II: `i-flow` adapts better, but needs more iterations.







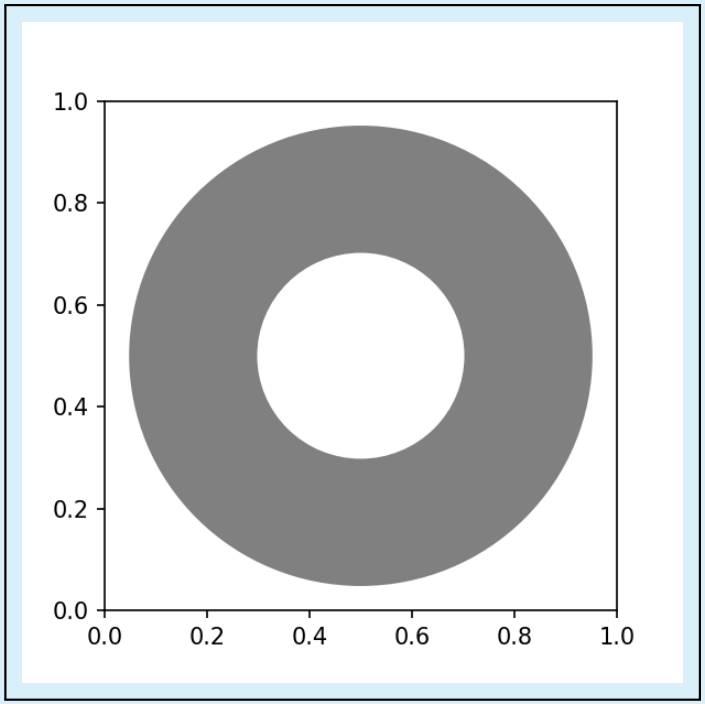




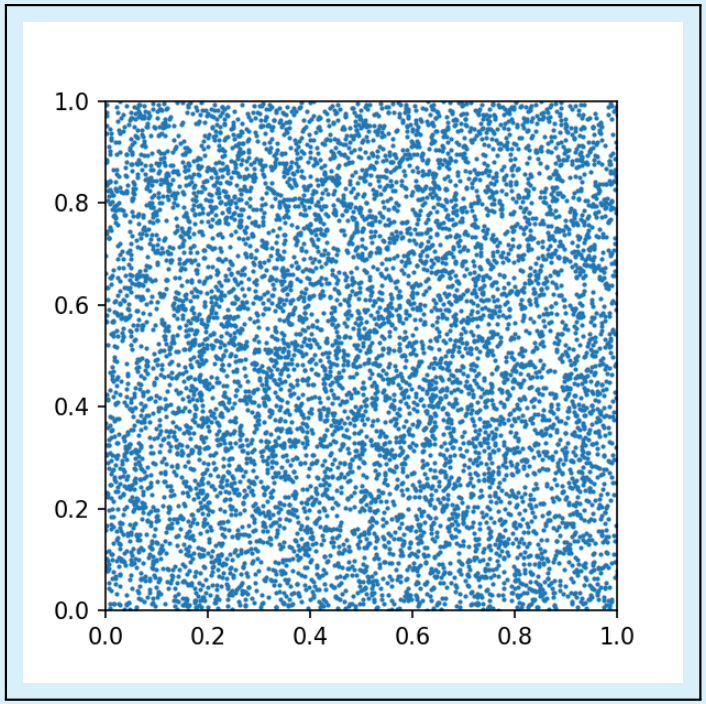
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

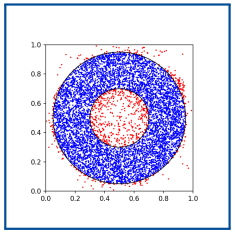
Target Distribution:



Before training:



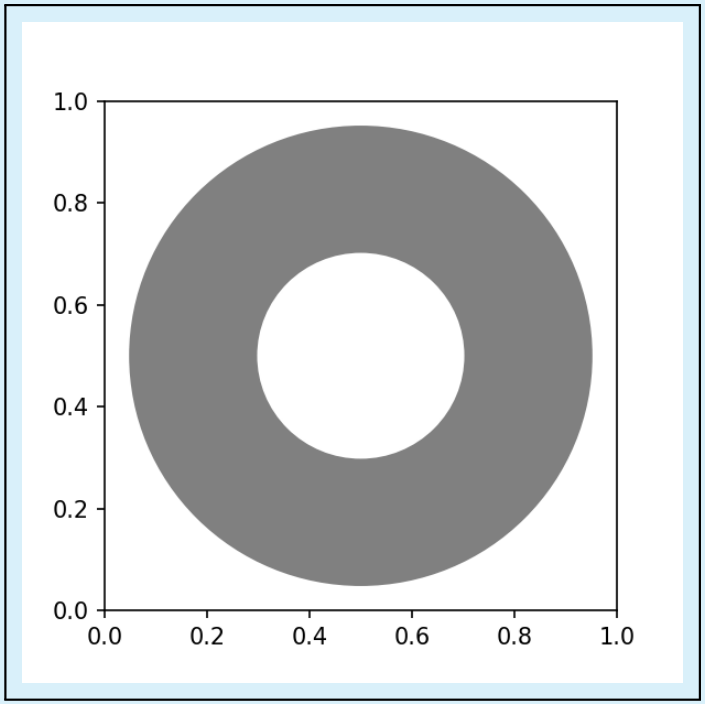
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



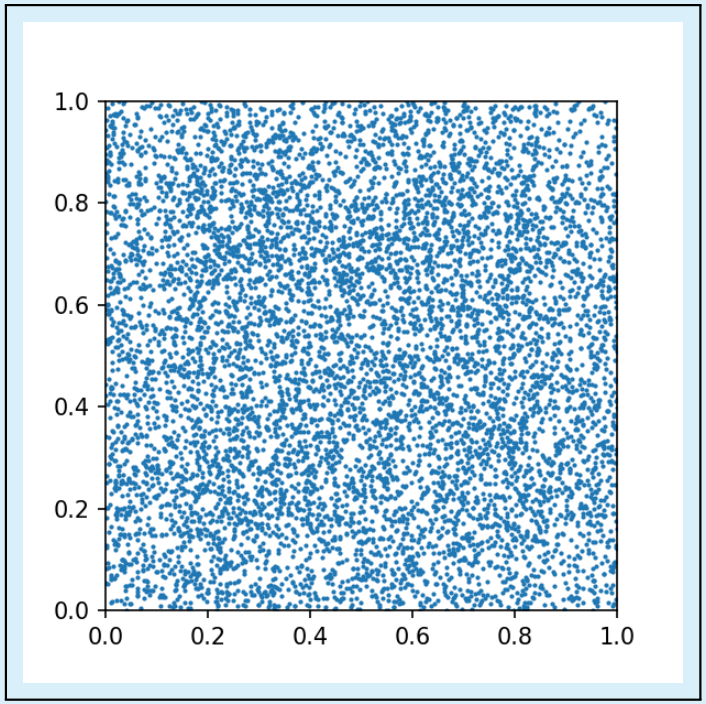
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

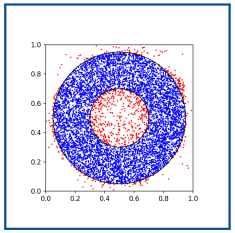
Target Distribution:



After 10 epochs:



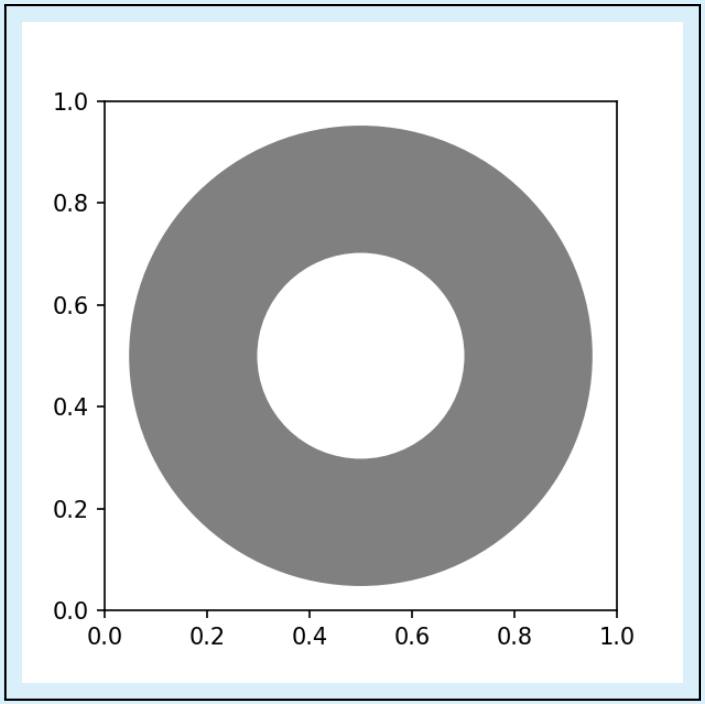
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



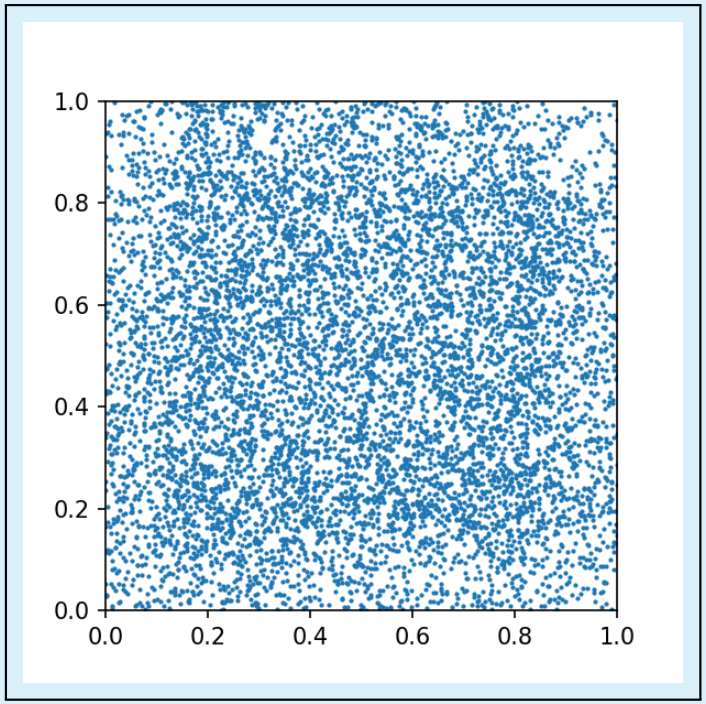
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

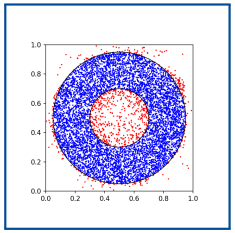
Target Distribution:



After 20 epochs:



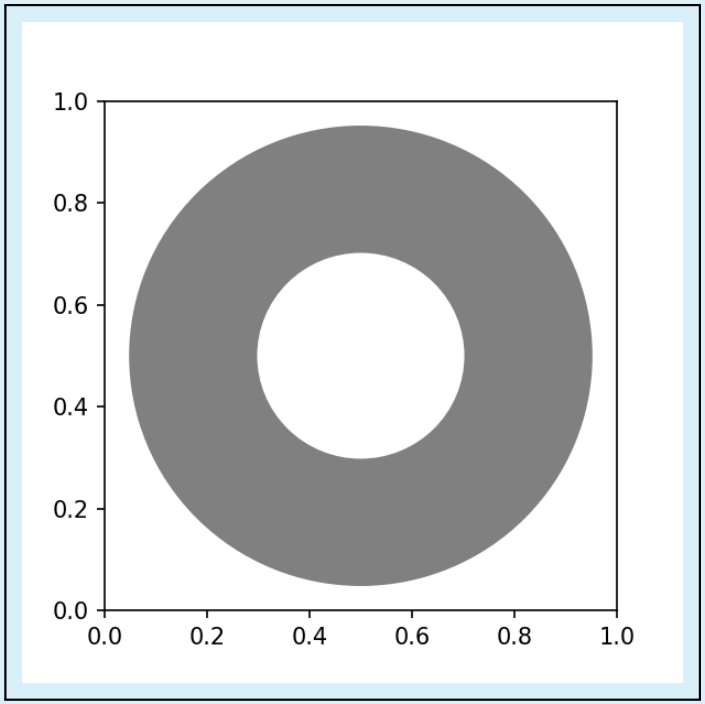
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



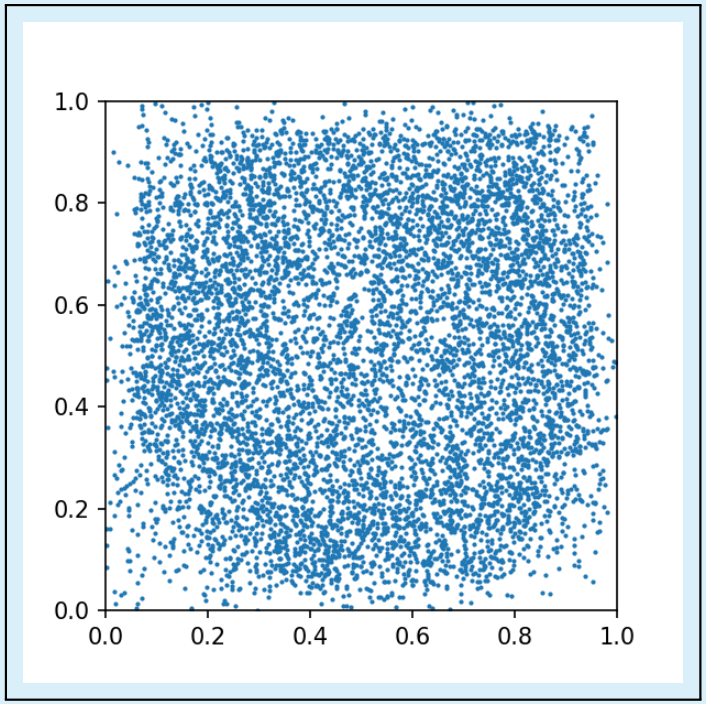
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

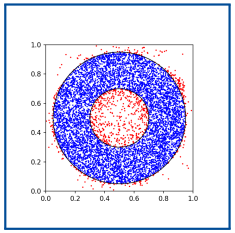
Target Distribution:



After 50 epochs:



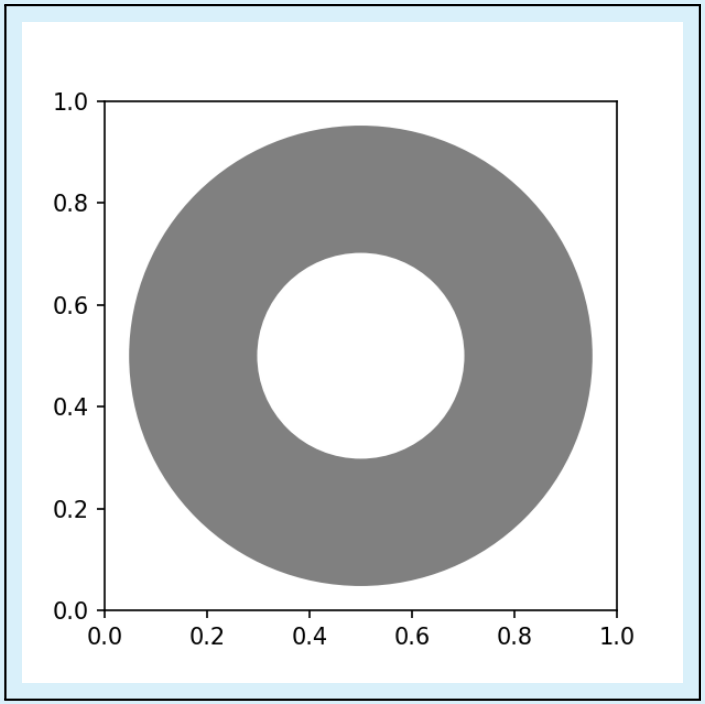
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



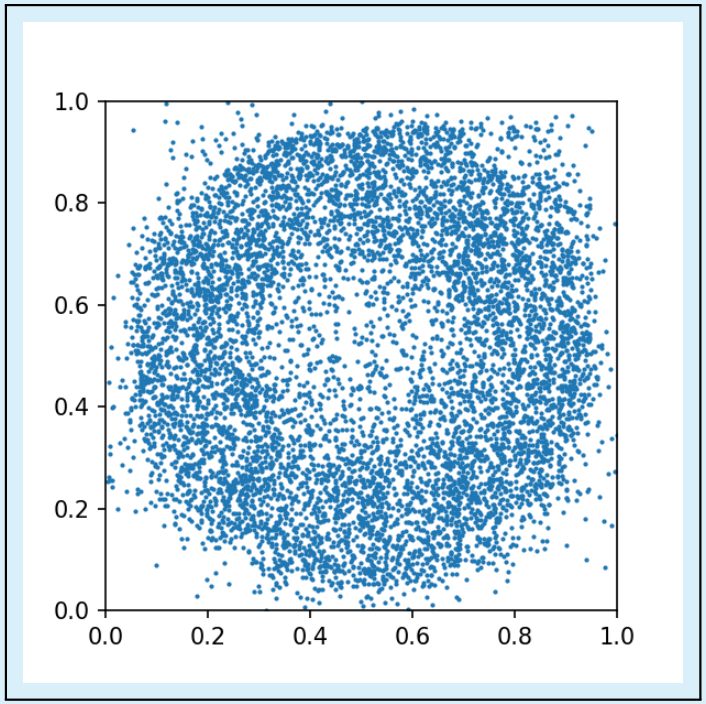
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

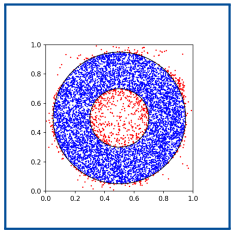
Target Distribution:



After 100 epochs:



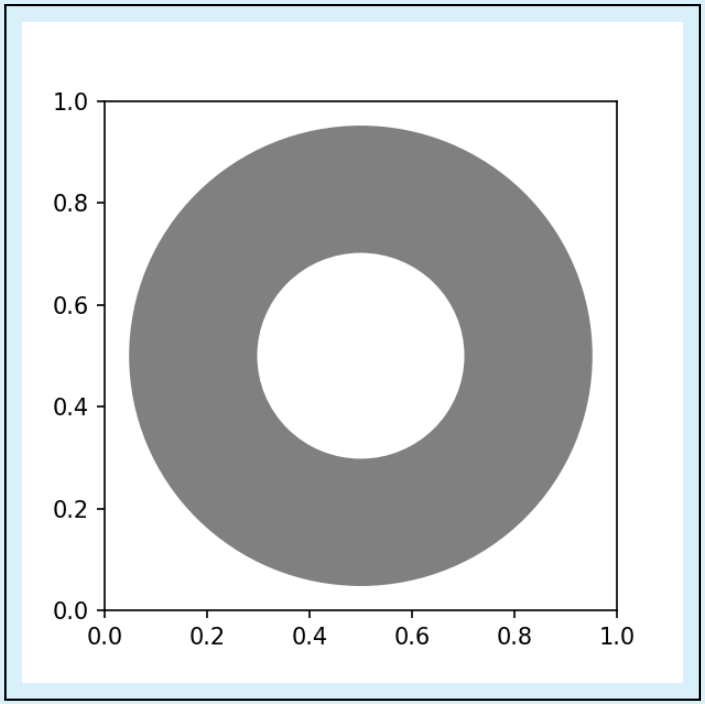
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



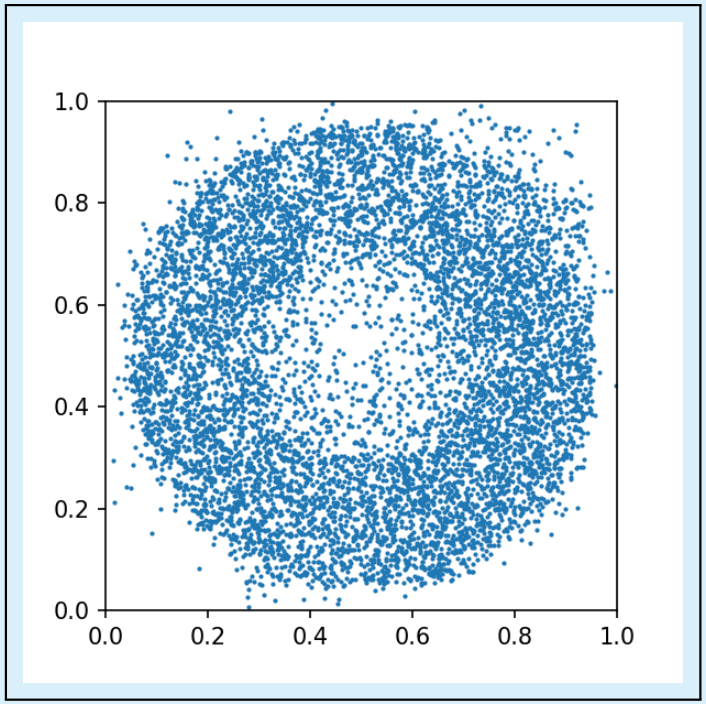
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

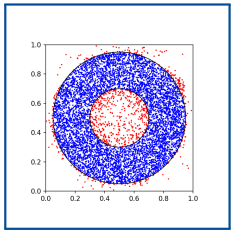
Target Distribution:



After 200 epochs:



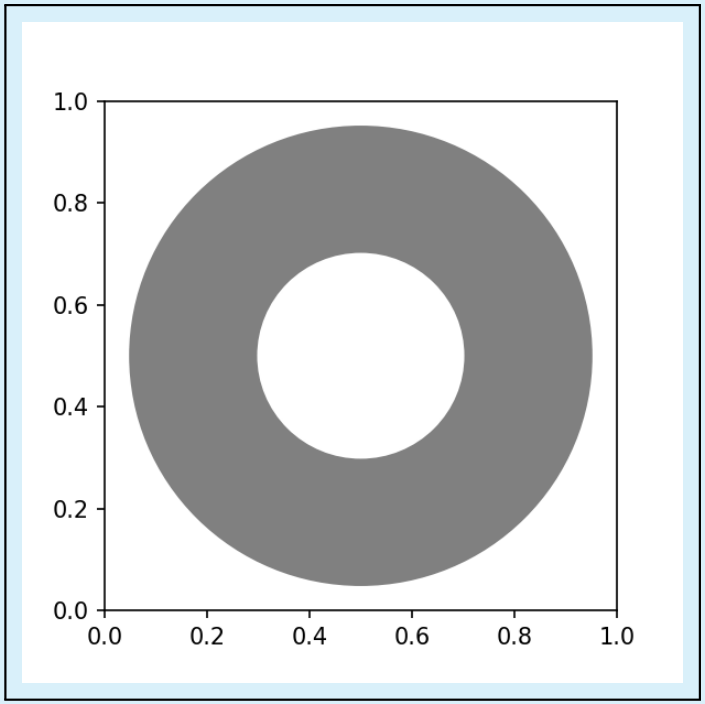
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



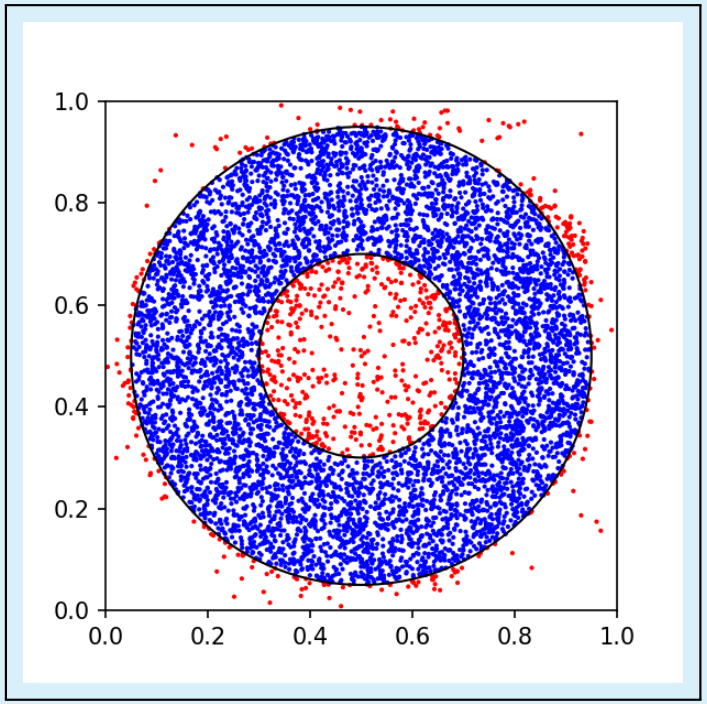
II: `i-flow` also learns hard, non-trivial cuts.

Our test function: a $2d$ annulus function.

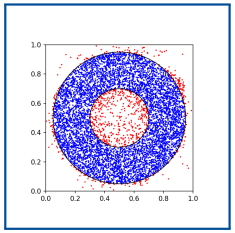
Target Distribution:



Final Distribution (500 epochs):



- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 $\pm$ 0.00018



II: Sherpa needs a high-dimensional integrator.

Sherpa is a Monte Carlo event generator for the **S**imulation of **H**igh-**E**nergy **R**eactions of **PA**rticles. We use Sherpa to

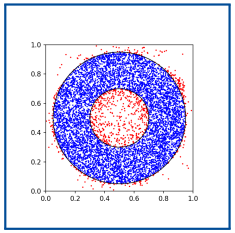
- compute the matrix element of the process.
- map the unit-hypercube of our integration domain to momenta and angles. To improve efficiency, Sherpa uses a recursive multichannel algorithm.

$$\Rightarrow n_{dim} = \underbrace{3n_{final} - 4}_{\text{kinematics}} + \underbrace{n_{final} - 1}_{\text{multichannel}}$$

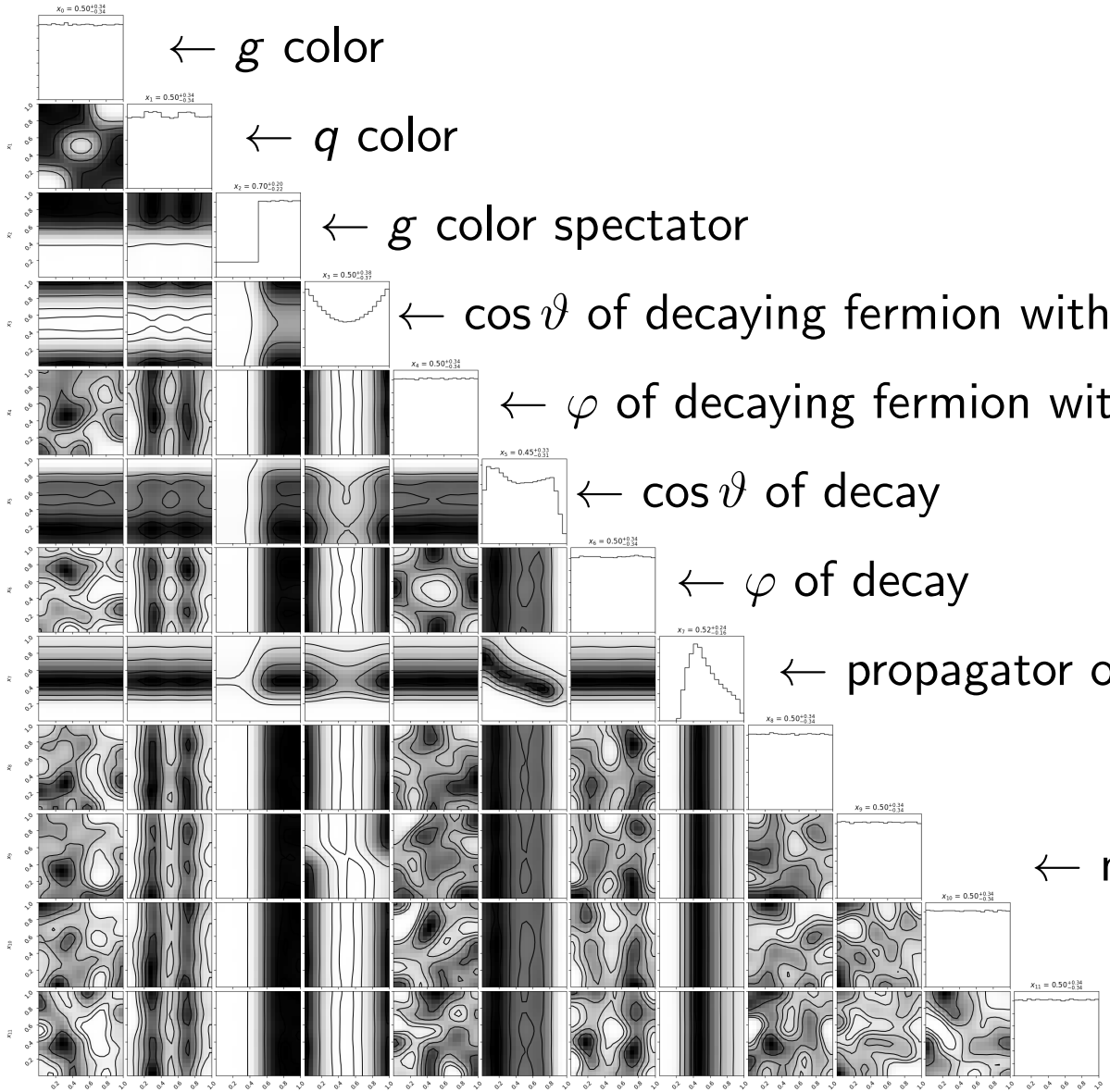
- However, the `COMIX++` ME-generator uses color-sampling, so we should also integrate over final state color configurations. While this improves the efficiency, it is not possible to handle group processes like $W + nj$ with a single flow.

$$\Rightarrow n_{dim} = 4n_{final} - 4 + 2n_{color}$$

https://sherpa.hepforge.org/



II: An easy example: $e^+e^- \to 3j$.



$\leftarrow$ g color

$\leftarrow$ q color

$\leftarrow$ g color spectator

$\leftarrow$ $\cos\vartheta$ of decaying fermion with beam

$\leftarrow$ φ of decaying fermion with beam

$\leftarrow$ $\cos\vartheta$ of decay

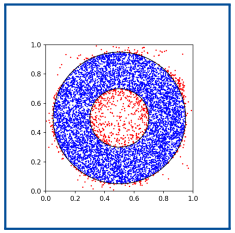
$\leftarrow$ φ of decay

$\leftarrow$ propagator of decaying fermion

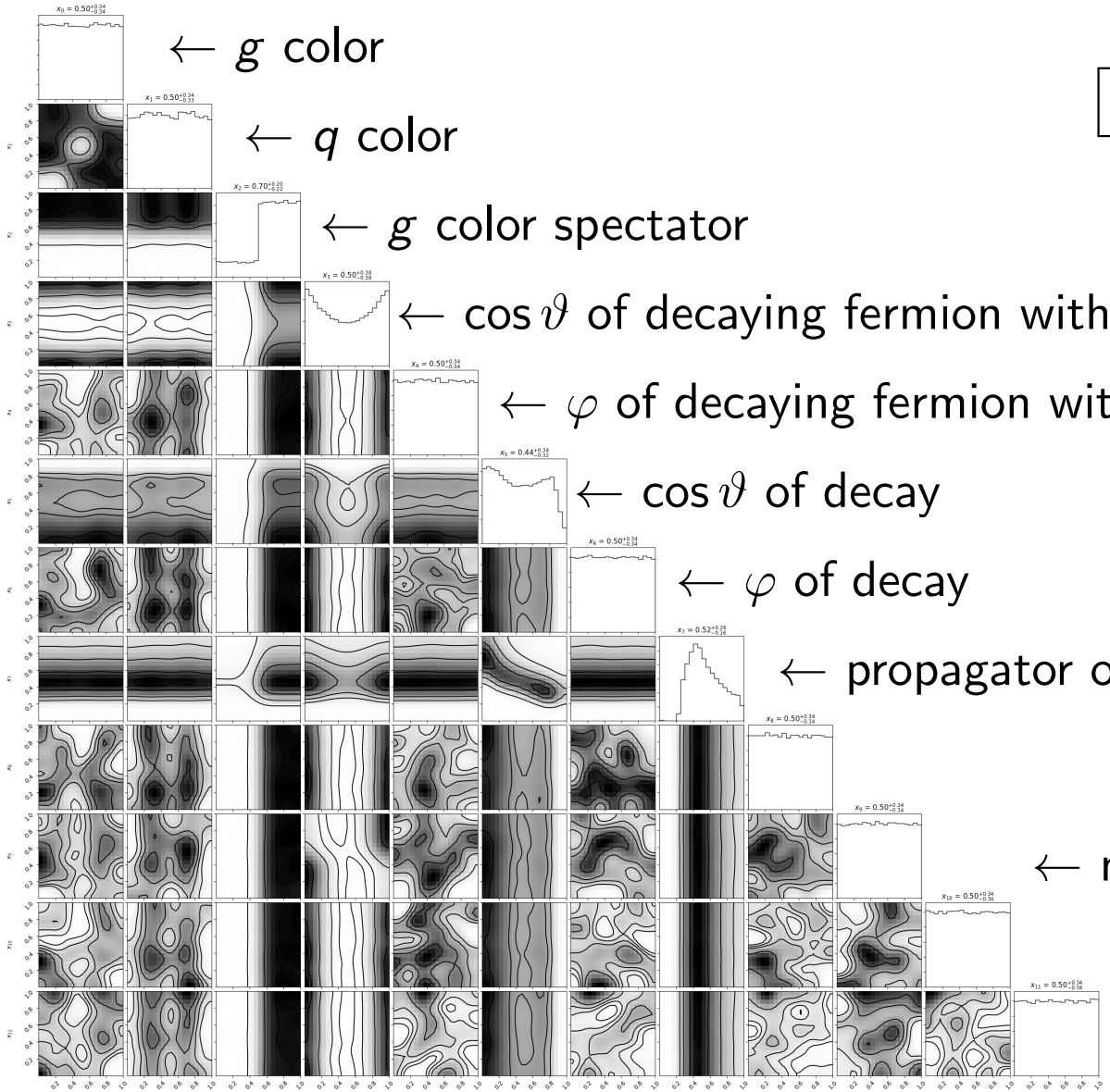
$\leftarrow$ multichannel

Target distribution

with learning color



II: An easy example: $e^+e^- \to 3j$.



$\leftarrow$ g color

$\leftarrow$ q color

$\leftarrow$ g color spectator

$\leftarrow$ $\cos\vartheta$ of decaying fermion with beam

$\leftarrow$ φ of decaying fermion with beam

$\leftarrow$ $\cos\vartheta$ of decay

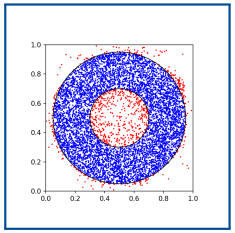
$\leftarrow$ φ of decay

$\leftarrow$ propagator of decaying fermion

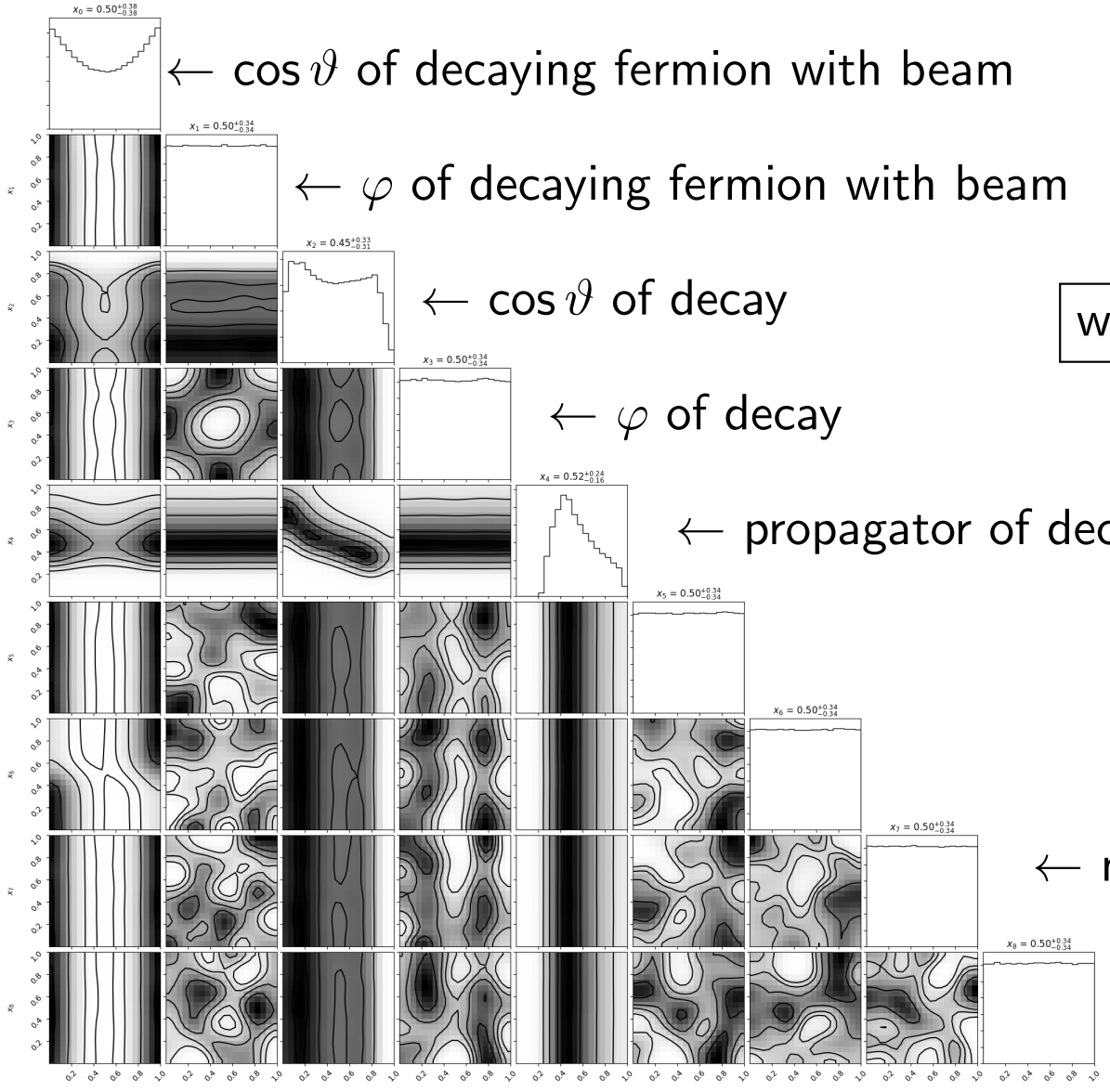
$\leftarrow$ multichannel

Learned distribution

with learning color



II: An easy example: $e^+e^- \to 3j$.



$\leftarrow \cos\vartheta$ of decaying fermion with beam

$\leftarrow \varphi$ of decaying fermion with beam

$\leftarrow \cos\vartheta$ of decay

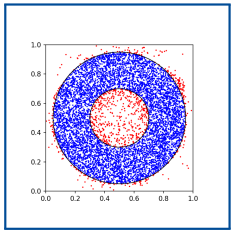
$\leftarrow \varphi$ of decay

$\leftarrow$ propagator of decaying fermion

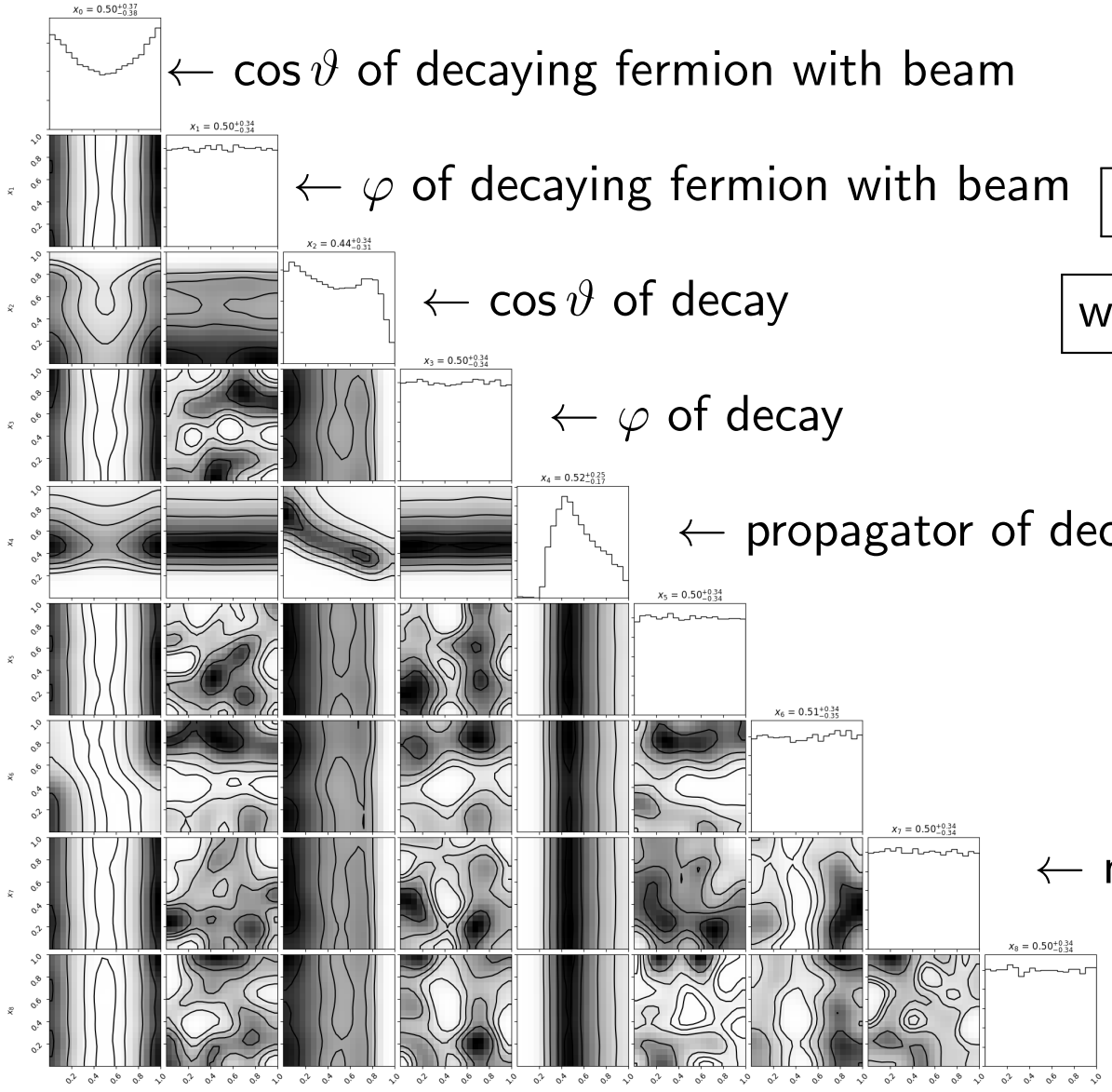
$\leftarrow$ multichannel

Target distribution

without learning color



II: An easy example: $e^+e^- \to 3j$.



$\leftarrow \cos\vartheta$ of decaying fermion with beam

$\leftarrow \varphi$ of decaying fermion with beam

$\leftarrow \cos\vartheta$ of decay

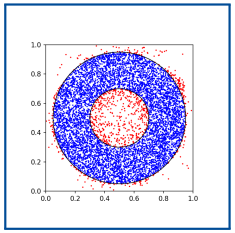
$\leftarrow \varphi$ of decay

$\leftarrow$ propagator of decaying fermion

$\leftarrow$ multichannel

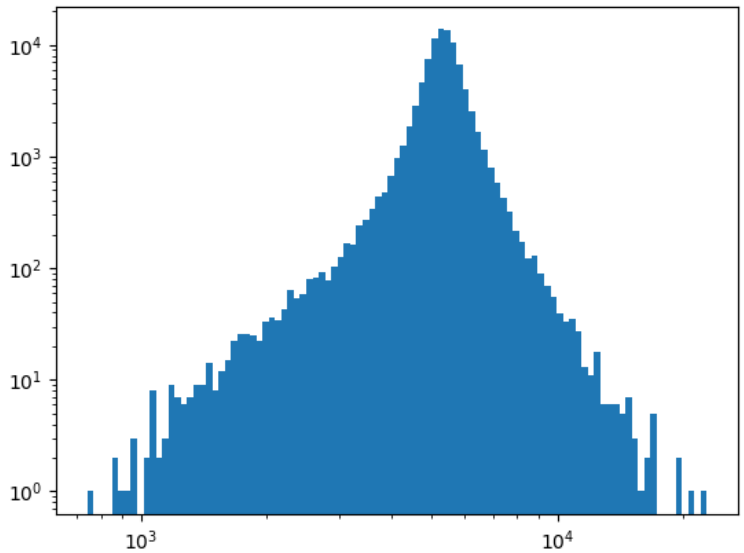
Learned distribution

without learning color



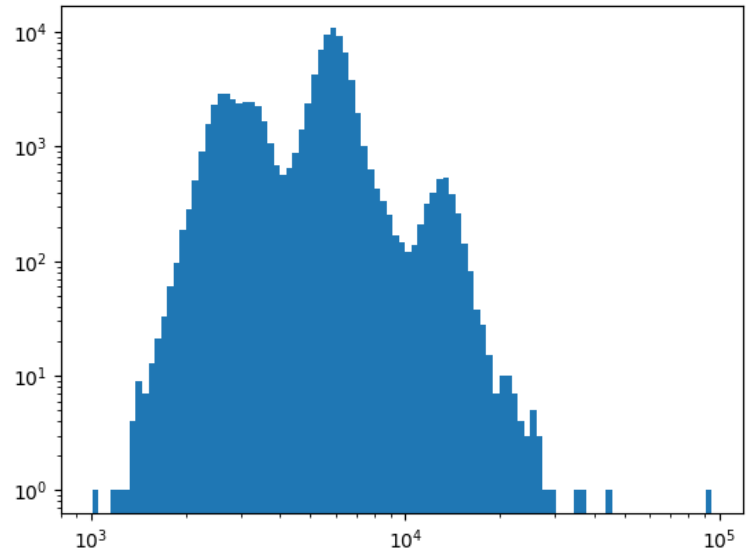
II: Comparing $e^+e^- \to 3j$ with and without learning color.

with learning color

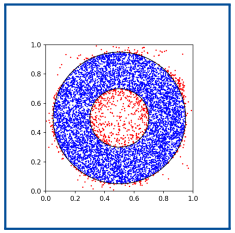


- $\sigma = 4879.8 \pm 5.3$pb
- $\eta_{\text{new}} = 45\%$
- Cut efficiency: 92 %
- 20 overweight events in 100k

without learning color



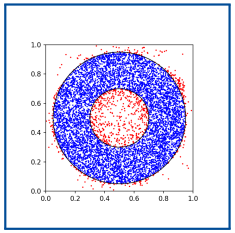
- $\sigma = 4883.5 \pm 8.5$pb
- $\eta_{\text{new}} = 25\%$
- Cut efficiency: 92 %
- 20 overweight events in 100k



II: High Multiplicities are still difficult to learn.

unweighting efficiency $\langle w \rangle / w_{\max}$		LO QCD $n=0$	$n=1$	$n=2$	$n=3$
$W^+ + n$ jets	Sherpa	$2.8 \cdot 10^{-1}$	$3.8 \cdot 10^{-2}$	$7.5 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$
	`i-flow`	$6.1 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$	$1.0 \cdot 10^{-2}$	$1.8 \cdot 10^{-3}$
	Gain	2.2	3.3	1.4	1.2
$W^- + n$ jets	Sherpa	$2.9 \cdot 10^{-1}$	$4.0 \cdot 10^{-2}$	$7.7 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$
	`i-flow`	$7.0 \cdot 10^{-1}$	$1.5 \cdot 10^{-1}$	$1.1 \cdot 10^{-2}$	$2.2 \cdot 10^{-3}$
	Gain	2.4	3.3	1.4	1.1
$Z + n$ jets	Sherpa	$3.1 \cdot 10^{-1}$	$3.6 \cdot 10^{-2}$	$1.5 \cdot 10^{-2}$	$4.7 \cdot 10^{-3}$
	`i-flow`	$3.8 \cdot 10^{-1}$	$1.0 \cdot 10^{-1}$	$1.4 \cdot 10^{-2}$	$2.4 \cdot 10^{-3}$
	Gain	1.2	2.9	0.91	0.51

C. Gao, S. Höche, J. Isaacson, CK, H. Schulz [arXiv:2001.10028, PRD]



II: There are numerous ways to improve `i-flow` in the near future.

- adjust hyperparameters
- use a CNN in the CL
- introduce Conditional Normalizing Flows or Discrete Flows to improve the multichannel or color sampling

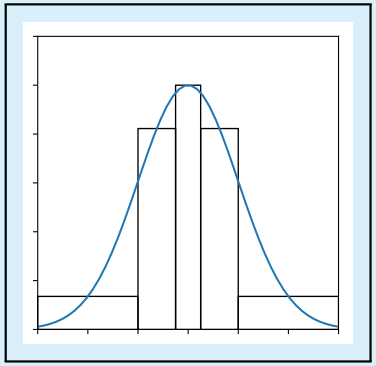
  Winkler et al. [1912.00042]; Tran et al. [1905.10347]
- "learn" the permutations: using 1×1 convolutions

  Kingma/Dhariwal [1807.03039]
- improve memory consumption with checkpointing

  Chen et al. [1604.06174]
- ...

Efficient Event Generation with Normalizing Flows
— i-flow —

- I introduced numerical integration with Monte Carlo techniques and importance sampling.
- I discussed "traditional" algorithms like VEGAS or Foam, and the Machine Learning approach using Normalizing Flows.



- I presented i-flow, our python implementation of Normalizing Flows and showed its performance in test functions. $\Rightarrow$ [2001.05486, ML:ST]
- I showed results for $pp \to W + nj$ with Sherpa. $\Rightarrow$ [2001.10028, *PRD*]