`i-flow`: Numerical Integration and Event Generation with Normalizing Flows

— ZOOM Theory seminar, NIKHEF —

Claudius Krause

Fermi National Accelerator Laboratory

April 9, 2020

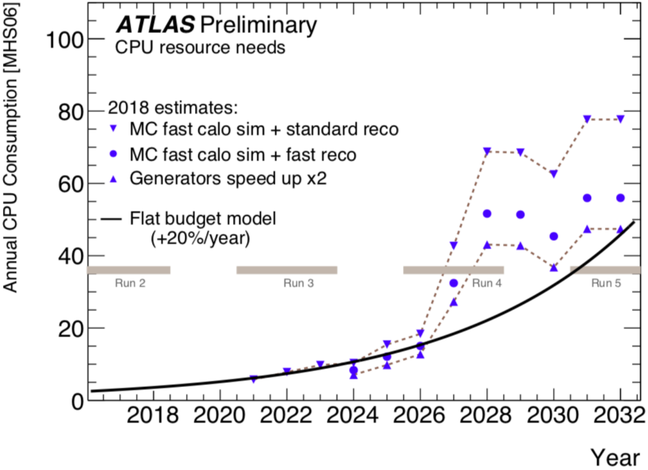
Unterstützt von / Supported by

Alexander von Humboldt Stiftung/Foundation

Fermilab

In collaboration with: Christina Gao, Stefan Höche, Joshua Isaacson, Holger Schulz
arXiv: 2001.05486 and arXiv: 2001.10028

Monte Carlo Simulations are increasingly important.

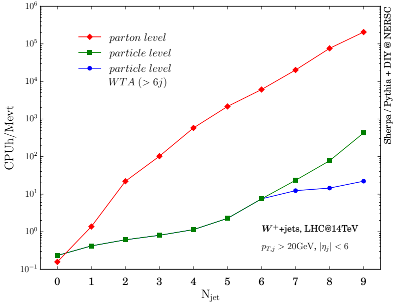


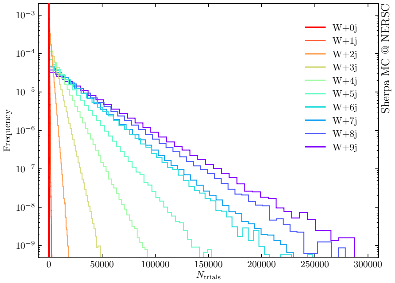
https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ComputingandSoftwarePublicResults

⇒ MC event generation is needed for signal and background predictions.

⇒ The required CPU time will increase in the next years.

Monte Carlo Simulations are increasingly important.



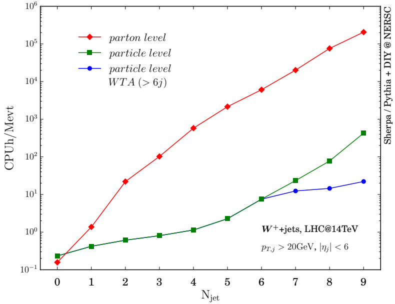


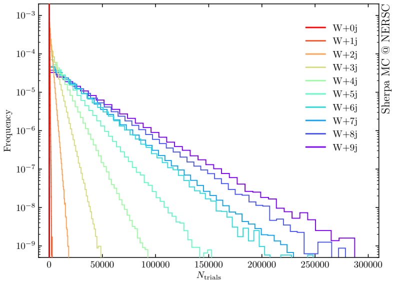
Stefan Höche, Stefan Prestel, Holger Schulz [1905.05120;PRD]

The bottlenecks for evaluating large final state multiplicities are

- a slow evaluation of the matrix element
- a low unweighting efficiency

Monte Carlo Simulations are increasingly important.



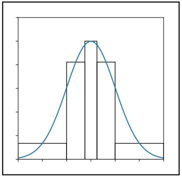


Stefan Höche, Stefan Prestel, Holger Schulz [1905.05120;PRD]

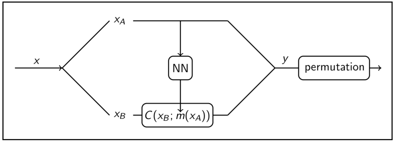
The bottlenecks for evaluating large final state multiplicities are

- a slow evaluation of the matrix element
- a low unweighting efficiency

i-flow: Numerical Integration and Event Generation with Normalizing Flows

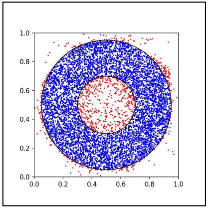


Part I: Monte Carlo Integration and Existing Algorithms



Part II: Machine Learning and Normalizing Flows

Part III: i-flow and its Applications

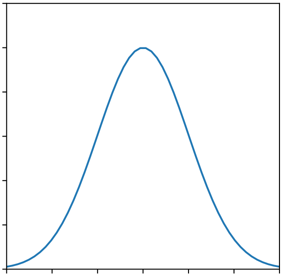


I: There are two problems to be solved...

1)

$$f(\vec{x})$$

$$d\sigma(p_i, \vartheta_i, \varphi_i)$$

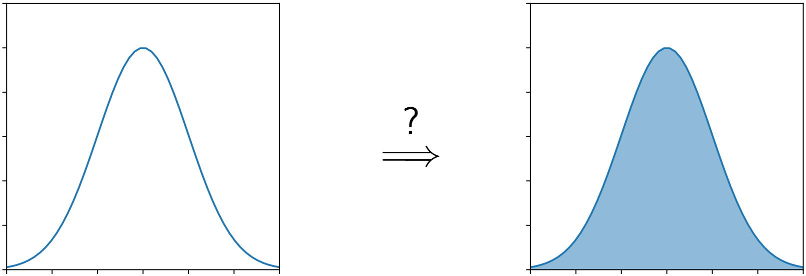


I: There are two problems to be solved...

1)

$$f(\vec{x}) \quad \Rightarrow \quad F = \int f(\vec{x})\, d^D x$$

$$d\sigma(p_i, \vartheta_i, \varphi_i) \quad \Rightarrow \quad \sigma = \int d\sigma(p_i, \vartheta_i, \varphi_i), \quad D = 3n_{\text{final}} - 4 + x$$



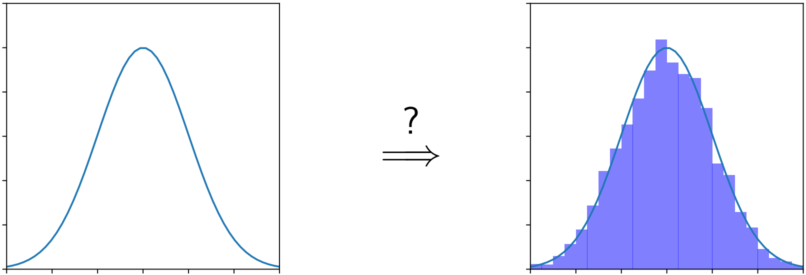
I: There are two problems to be solved...

1)

$$f(\vec{x}) \quad \Rightarrow \quad F = \int f(\vec{x})\, d^D x$$

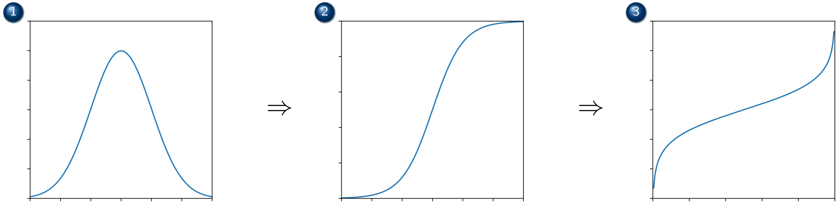
$$d\sigma(p_i, \vartheta_i, \varphi_i) \quad \Rightarrow \quad \sigma = \int d\sigma(p_i, \vartheta_i, \varphi_i), \quad D = 3n_{\text{final}} - 4 + x$$

2) Given a distribution $f(\vec{x})$, how can we sample according to it?



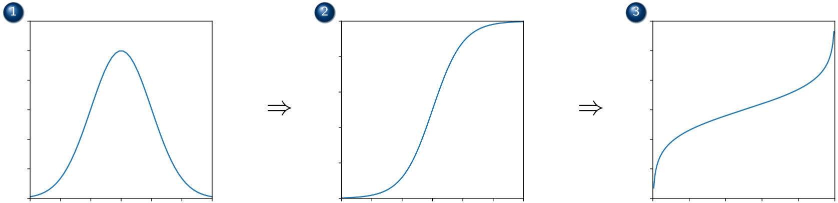
I: . . . but they are closely related.

1. Starting from a pdf, . . .
2. . . . we can integrate it and find its cdf, . . .
3. . . . to finally use its inverse to transform a uniform distribution.

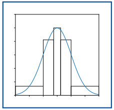


I: . . . but they are closely related.

1. Starting from a pdf, . . .
2. . . . we can integrate it and find its cdf, . . .
3. . . . to finally use its inverse to transform a uniform distribution.



⇒ We need a fast and effective numerical integration!



I: Importance Sampling is very efficient for high-dimensional integration.

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

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

I: Importance Sampling is very efficient for high-dimensional integration.

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

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

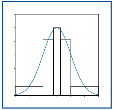
We therefore have to find a $q(x)$ that

- approximates the shape of $f(x)$.
- is “easy” enough such that we can sample from its inverse cdf.

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

- If $q(x) = \text{const.}$, each event x_i would require a weight of $f(x_i)$ to reproduce the distribution of $f(x)$. $\Rightarrow$ "Weighted Events"

- If $q(x) \propto f(x)$, all events would have the same weight as the distribution reproduces $f(x)$ directly. $\Rightarrow$ "Unweighted Events"



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

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

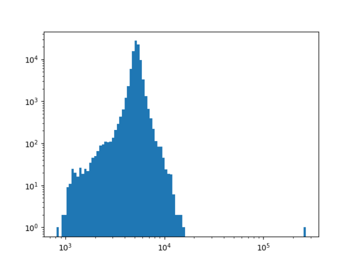
- To unweight, we need to accept/reject each event with probability $\frac{f(x_i)}{\max f(x)}$. The resulting set of kept events is unweighted and reproduces the shape of $f(x)$.
- The unweighting efficiency η gives the fraction of events that “survives” this procedure.

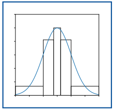
$$\eta = \frac{\#\ \text{accepted events}}{\#\ \text{all events}} = \frac{\text{mean } w}{\max w}, \text{ with } w_i = \frac{p(x_i)}{q(x_i)} = \frac{f(x_i)}{Fq(x_i)}.$$

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.

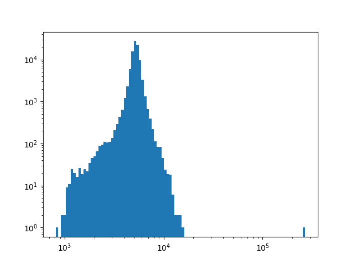




I: The usual definition of unweighting efficiency is unstable if many events are generated.

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.



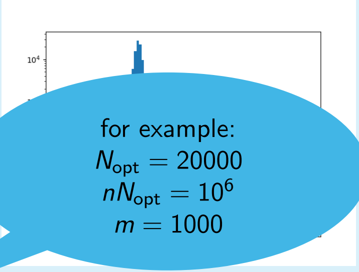
Our new definition:

- 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:

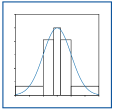
- The maximum grows with the number of events drawn.
- If more points are drawn than used in training, the chance for outliers increases a lot.
- Generating smaller subsets doesn't work, because we want a globally unweighted set of events.



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

Our new definition:

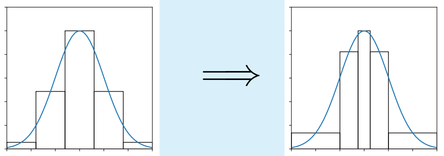
- Assuming we used N_{opt} events during optimization, draw nN_{opt} events.
- Now, select m replicas of N_{opt} events each and find their maximum weight.
- Compute the total maximum as the median of the individual maxima.
- We expect a few overweight events that can either be discarded or included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

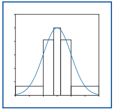


I: The VEGAS algorithm is very efficient.

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.

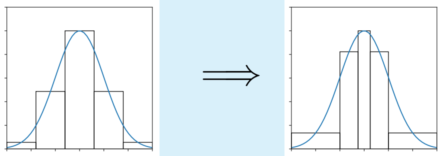




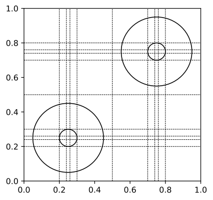
I: The VEGAS algorithm is very efficient.

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.

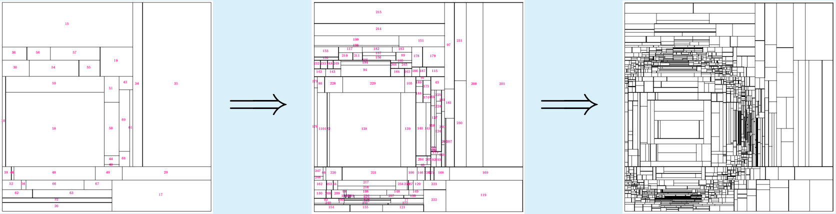


I: The `Foam` algorithm resolves correlations.

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.



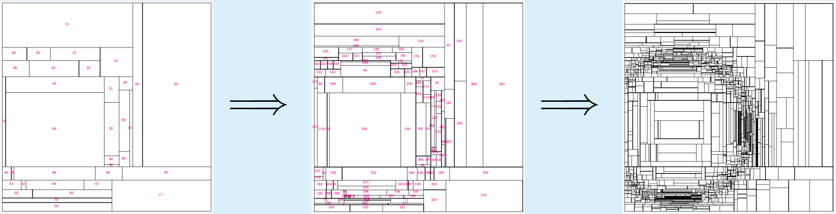
illustrations from ICHEP 2002 slides, S. Jadach

I: The `Foam` algorithm resolves correlations.

The `Foam` algorithm

S. Jadach [physics/0203033]

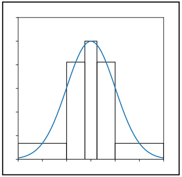
- In the exploration phase, the integration domain is consecutively split into cells.
- In the generation phase, a cell is chosen at random and a point is drawn uniformly from within that cell.



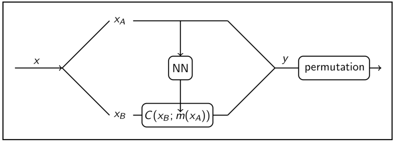
illustrations from ICHEP 2002 slides, S. Jadach

- It captures correlations.
- However, within each cell $q(x) = \text{const}$.

i-flow: Numerical Integration and Event Generation with Normalizing Flows

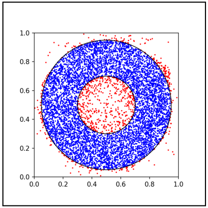


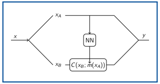
Part I: Monte Carlo Integration and Existing Algorithms



Part II: Machine Learning and Normalizing Flows

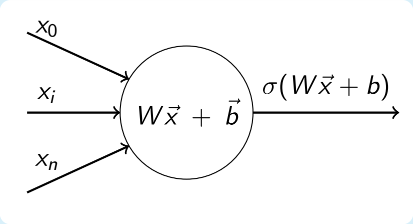
Part III: `i-flow` and its Applications



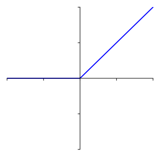


II: Neural Networks are nonlinear functions, inspired by the human brain.

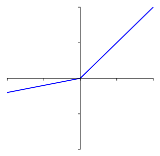
Each neuron transforms the input with a weight W and a bias $\vec{b}$.



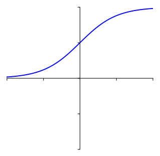
The activation function σ makes it nonlinear.



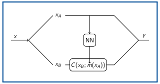
"rectified linear unit (relu)"



"leaky relu"



"sigmoid"



II: There are different approaches to generate events with Machine Learning Techniques.

Generate events directly using GANs.

Bendavid [1707.00028]; Otten et al. [1901.00875]; Hashemi et al. [1901.0528]; Di Sipio et al. [1903.02433]; Butter e l. [1907.03764, 1912.08824]; Carrazza l. [1909.01359]; Ahdida et al. [190 1]

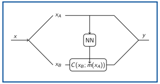
Generative Adversarial Network:
A *generator* and a *discriminator* play a “game”.

- × Need existing event sample to train.
- × Results can be biased if not trained right.

Learn $q(x)$ to improve importance sampling.

Bendavid [1707.00028]; Klimek/Perelstein [1810.11509]; `i-flow` **[this talk, 2001.05486]**

- ✓ Insufficient training just yields high uncertainties, no bias.
- ✓ Events are generated from scratch, no pre-existing set is needed.
- × Resulting set of events still needs to be unweighted.



II: There are different approaches to generate events with Machine Learning Techniques.

Generate events directly using GANs.
Bendavid [1707.00028]; Otten et al. [1901.00875]; Hashemi et al. [1901.05282]; Di Sipio et al. [1903.02433]; Butter et al. [1907.03764, 1912.08824]; Carrazza et al. [1909.01359]; Ahdida et al. [1909.04451]

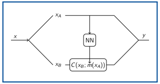
- ✓ Several orders of magnitude faster.
- ✓ Generates unweighted events directly.

- × Need existing event sample to train.
- × Results can be biased if not trained right.

Learn $q(x)$ to improve importance sampling.
Bendavid [1707.00028]; Klimek/Perelstein [1810.11509]; `i-flow` **[this talk, 2001.05486]**

- ✓ Insufficient training just yields high uncertainties, no bias.
- ✓ Events are generated from scratch, no pre-existing set is needed.

- × Resulting set of events still needs to be unweighted.



II: The Loss function quantifies our goal.

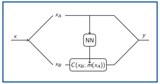
We have different choices:

- Kullback-Leibler (KL) divergence:
  $$D_{KL} = \int p(x) \log \frac{p(x)}{q(x)} dx \qquad \approx \qquad \frac{1}{N} \sum \frac{p(x_i)}{q(x_i)} \log \frac{p(x_i)}{q(x_i)}, \qquad x_i \dots q(x)$$
- Pearson χ^2 divergence:
  $$D_{\chi^2} = \int \frac{(p(x) - q(x))^2}{q(x)} dx \qquad \approx \qquad \frac{1}{N} \sum \frac{p(x_i)^2}{q(x_i)^2} - 1, \qquad x_i \dots q(x)$$
- Exponential divergence:
  $$D_{exp} = \int p(x) \log \left(\frac{p(x)}{q(x)}\right)^2 dx \quad \approx \quad \frac{1}{N} \sum \frac{p(x_i)}{q(x_i)} \log \left(\frac{p(x_i)}{q(x_i)}\right)^2, \quad x_i \dots q(x)$$

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]



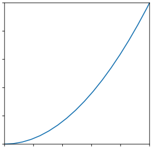
II: Using the NN as coordinate transform is too costly.

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 x to a target $q(x)$.
- The distribution induced by the map $y(x)$ (=NN) is given by the Jacobian of the map:

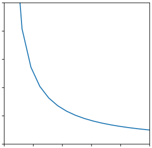
  $q(y) = q(y(x)) = \left|\frac{\partial y}{\partial x}\right|^{-1}$

Klimek/Perelstein [arXiv:1810.11509]

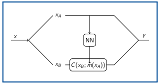


$y = x^2$

Jacobian $\longrightarrow$



$\left|\frac{\partial y}{\partial x}\right|^{-1} = \frac{1}{2x}$

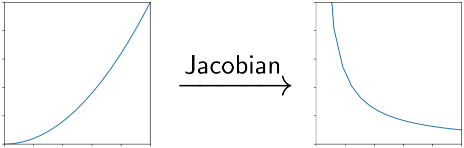


II: Using the NN as coordinate transform is too costly.

We could use the NN as nonlinear coordinate transform:

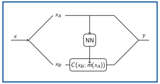
- We use a deep NN with n_{dim} nodes in the first and last layer to map a uniformly distributed x to a target $q(x)$.
- The distribution induced by the map $y(x)$ (=NN) is given by the Jacobian of the map:
  $q(y) = q(y(x)) = \left|\frac{\partial y}{\partial x}\right|^{-1}$

Klimek/Perelstein [arXiv:1810.11509]



$y = x^2$ $\qquad$ $\left|\frac{\partial y}{\partial x}\right|^{-1} = \frac{1}{2x}$

⇒ 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!

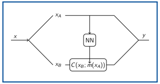


II: Normalizing Flows are numerically cheaper.

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.



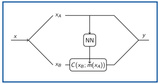
II: Normalizing Flows are numerically cheaper.

A Normalizing Flow:

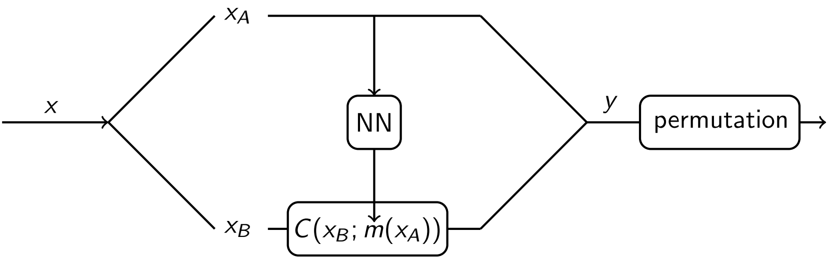
- is a bijective, smooth mapping between two statistical distributions.
- is composed of a series of easy transformations, the *"Coupling Layers"*.
- is still flexible enough to learn complicated distributions.

$\Rightarrow$ 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].



II: 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 $x_{B,i}$.

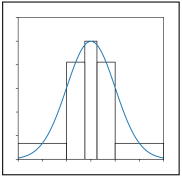
Jacobian:

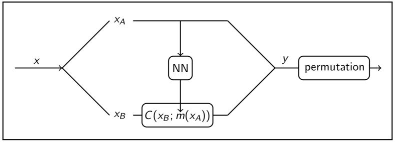
$$\left|\frac{\partial y}{\partial x}\right| = \begin{vmatrix} 1 & \frac{\partial C}{\partial x_A} \\ 0 & \frac{\partial C}{\partial x_B} \end{vmatrix} = \Pi_i \frac{\partial C(x_{B,i}; m(x_A))}{\partial x_{B,i}}$$

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

i-flow: Numerical Integration and Event Generation with Normalizing Flows

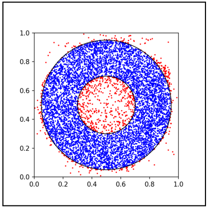
Part I: Monte Carlo Integration and Existing Algorithms

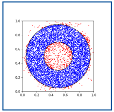




Part II: Machine Learning and Normalizing Flows

Part III: i-flow and its Applications





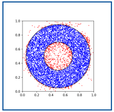
III: Introducing: i-flow.

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

- 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,
- Neural Network architectures,
- Loss functions,
- settings for hyperparameters.



III: Introducing: i-flow.

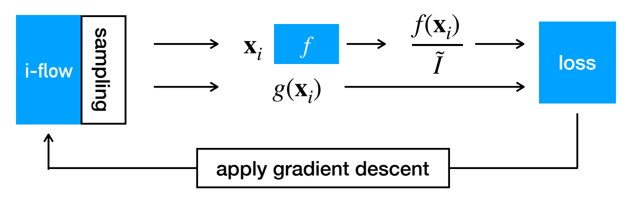
i-flow C. Gao, J. Isaacson, CK [arXiv:2001.05486]

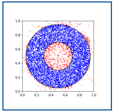
- implements Normalizing Flows in python using TensorFlow 2.0.
- is available at gitlab.com/i-flow/i-flow.

The user can choose different

- transformations in the Coupling Layer,
- Neural Network architectures,
- Loss functions,
- settings for hyperparameters.

How it works:

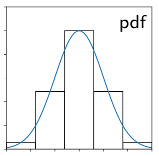


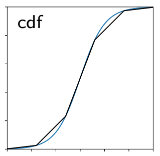


III: The Coupling Function is a piecewise approximation to the cdf.

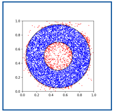
piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]





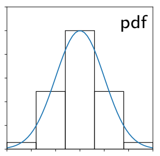
The NN predicts the pdf bin heights Q_i.

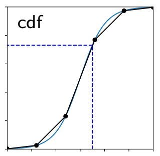


III: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]



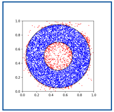


The NN predicts the pdf bin heights Q_i.

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

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

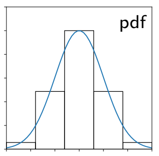
$$\left|\frac{\partial C}{\partial x_B}\right| = \Pi_i \frac{Q_{b_i}}{w}$$

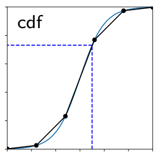


III: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]





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

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

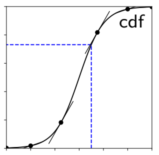
$$\left|\frac{\partial C}{\partial x_B}\right| = \Pi_i \frac{Q_{b_i}}{w}$$

The NN predicts the pdf bin heights Q_i.

rational quadratic spline coupling function:

Durkan et al. [arXiv:1906.04032]

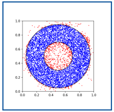
Gregory/Delbourgo [IMA Journal of Numerical Analysis, '82]



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

- still rather easy
- more flexible

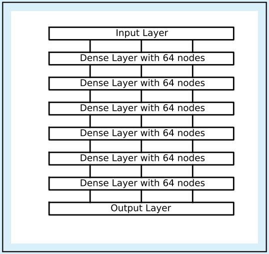
The NN predicts the cdf bin widths, heights, and derivatives that go in $a_i \& b_i$.



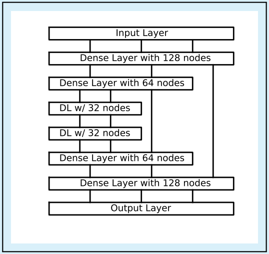
III: There are many hyperparameters to adjust.

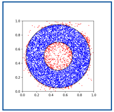
Available Architectures: Müller et al. [arXiv:1808.03856]

"Fully Connected" Neural Net (NN):



"U-shaped" Neural Net (Unet):

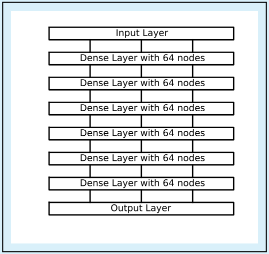




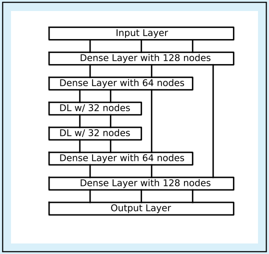
III: There are many hyperparameters to adjust.

Available Architectures: Müller et al. [arXiv:1808.03856]

"Fully Connected" Neural Net (NN):

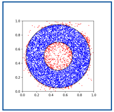


"U-shaped" Neural Net (Unet):



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



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

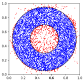
How many Coupling Layers do we need?

- 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)$\leftrightarrow$(tttppp) *vs.* (ppppttt)$\leftrightarrow$(ppttpp)$\leftrightarrow$(ttpppp)

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

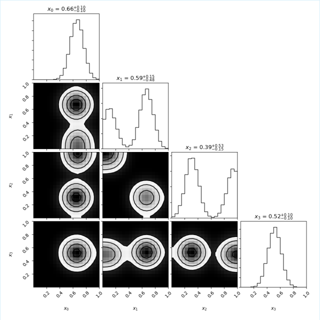
- For $\#p \approx \#t$, we can prove: $\boxed{4 \leq \#CLs \leq 2\lceil \log_2 n_{dim} \rceil}$



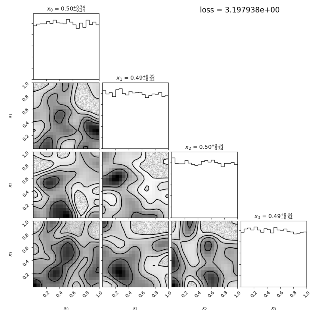
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

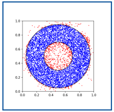
Target Distribution:



Before training:



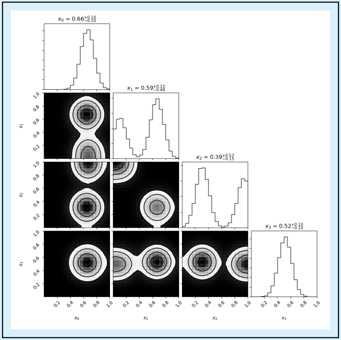
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363



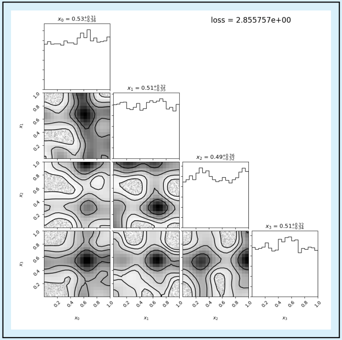
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

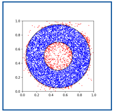
Target Distribution:



After 5 epochs:



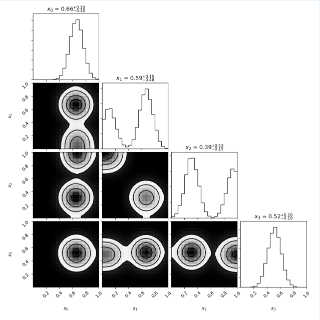
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363



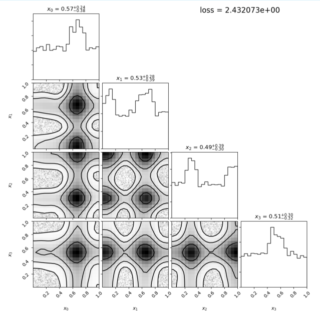
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

Target Distribution:

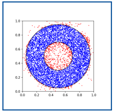


After 10 epochs:



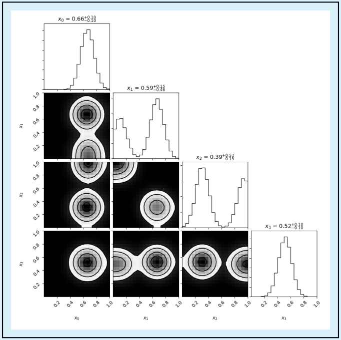
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363

III: The 4-d Camel function illustrates the learning of `i-flow`.

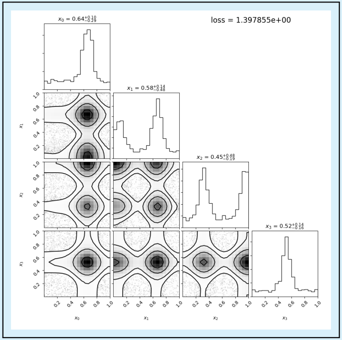


Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

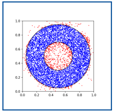
Target Distribution:



After 25 epochs:



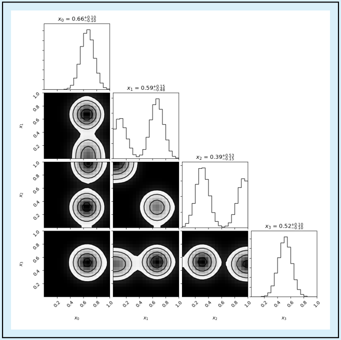
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363



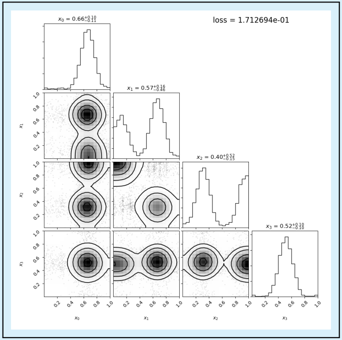
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

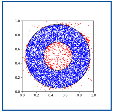
Target Distribution:



After 100 epochs:



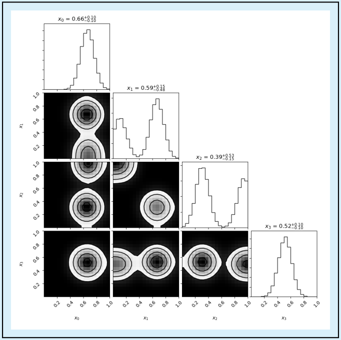
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363



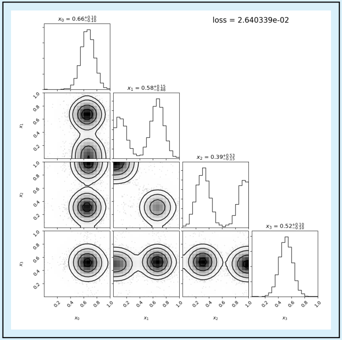
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

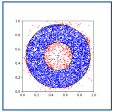
Target Distribution:



After 200 epochs:



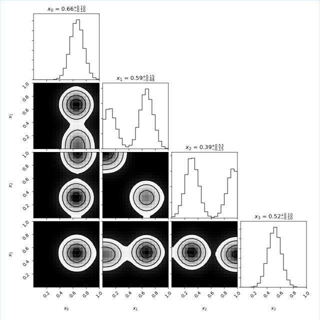
Dim	VEGAS	Foam	i-flow	true value
2	0.98112(89)	0.98169(5)	0.98171(4)	0.98166
4	0.96378(222)	0.96356(30)	0.96389(25)	0.963657
8	0.87752(759)	0.93007(142)	0.92788(44)	0.928635
16	0.43139(25)	0.96498(17337)	0.86153(104)	0.862363



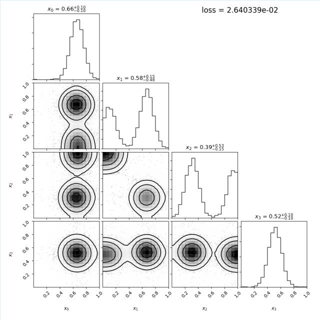
III: The 4-d Camel function illustrates the learning of `i-flow`.

Our test function: 2 Gaussian peaks, randomly placed in a 4d space.

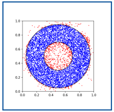
Target Distribution:



After 200 epochs:



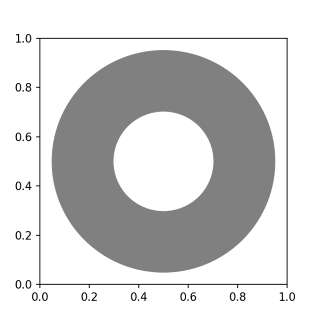
Dim	VEGAS	Foam	i-flow
2	−0.61	0.6	1.25
4	0.06	−0.32	0.93
8	−6.73	1.01	−1.72
16	−1723.89	0.59	−0.8



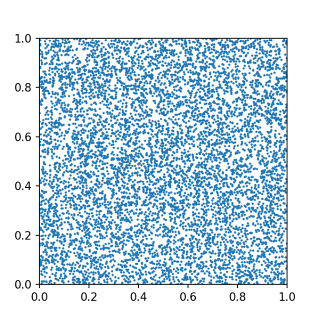
III: `i-flow` also learns hard, non-trivial cuts.

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

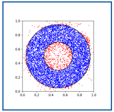
Target Distribution:



Before training:



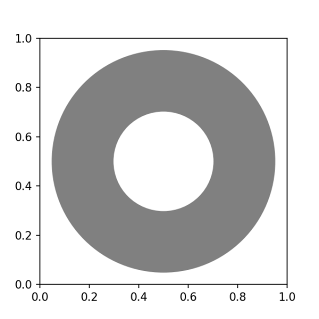
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



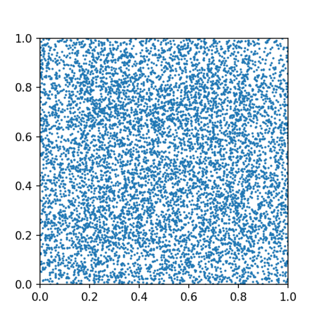
III: `i-flow` also learns hard, non-trivial cuts.

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

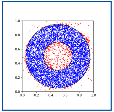
Target Distribution:



After 10 epochs:



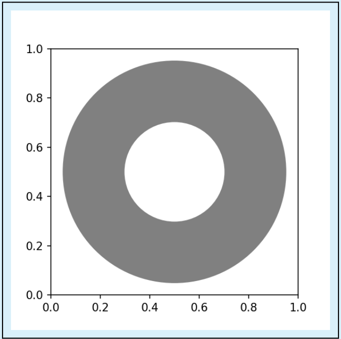
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



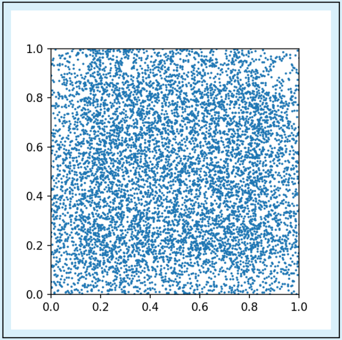
III: `i-flow` also learns hard, non-trivial cuts.

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

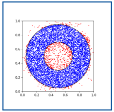
Target Distribution:



After 20 epochs:



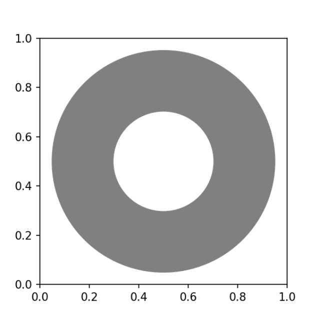
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



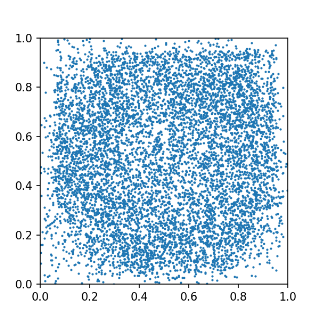
III: i-flow also learns hard, non-trivial cuts.

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

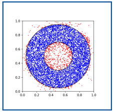
Target Distribution:



After 50 epochs:



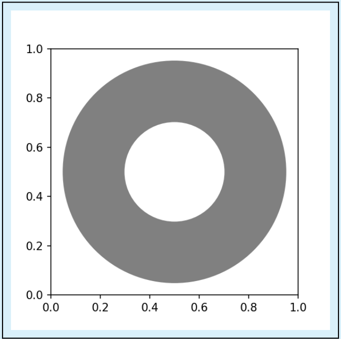
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



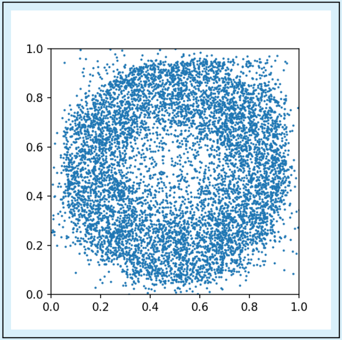
III: `i-flow` also learns hard, non-trivial cuts.

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

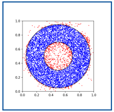
Target Distribution:



After 100 epochs:



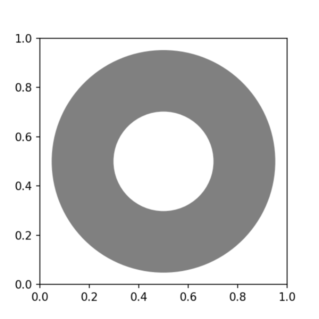
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



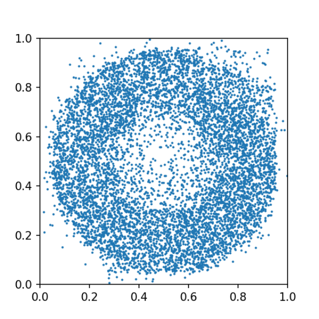
III: i-flow also learns hard, non-trivial cuts.

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

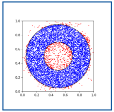
Target Distribution:



After 200 epochs:



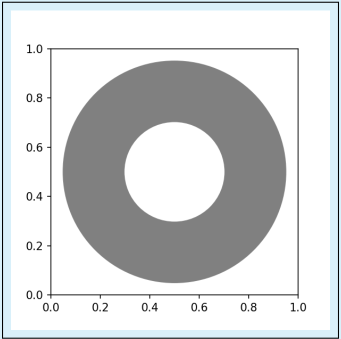
- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



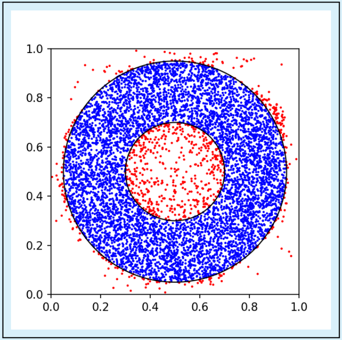
III: `i-flow` also learns hard, non-trivial cuts.

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

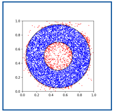
Target Distribution:



Final Distribution (500 epochs):



- Final cut efficiency: 89 % Untrained efficiency: 51 %
- Integral: 0.510508 Estimated integral: 0.51040 ± 0.00018



III: 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

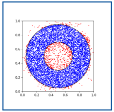
- compute the matrix element of the process.
- map the unit-hypercube of our integration domain to momenta and angles. To improve efficiency, Sherpa uses a recursive multichannel algorithm.

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

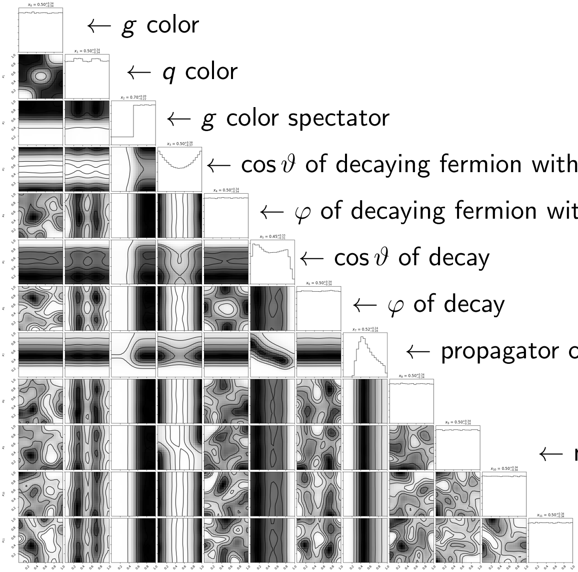
- However, the `COMIX++` ME-generator uses color-sampling, so we should also integrate over final state color configurations. While this improves the efficiency, it is not possible to handle group processes like $W + nj$ with a single flow.

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

https://sherpa.hepforge.org/



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



← g color

← q color

← g color spectator

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

← φ of decaying fermion with beam

← $\cos\vartheta$ of decay

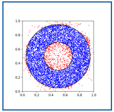
← φ of decay

← propagator of decaying fermion

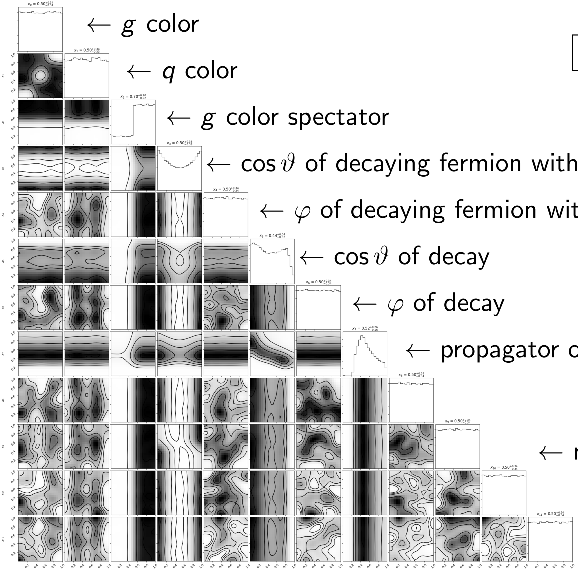
← multichannel

Target distribution

with learning color



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



← g color

← q color

← g color spectator

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

← φ of decaying fermion with beam

← $\cos\vartheta$ of decay

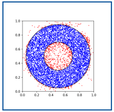
← φ of decay

← propagator of decaying fermion

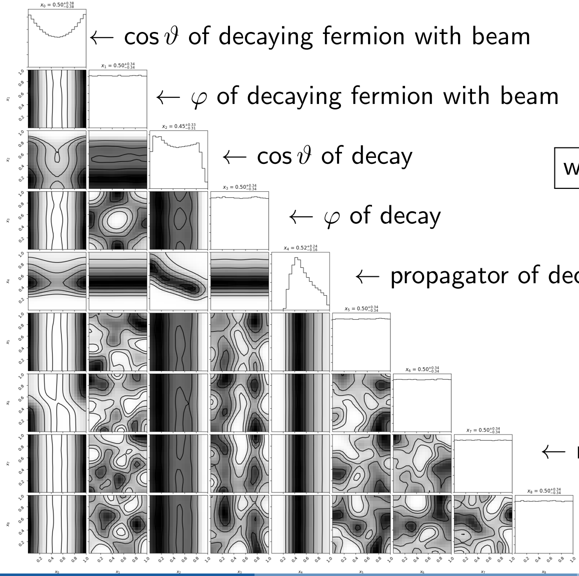
← multichannel

Learned distribution

with learning color



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



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

$\leftarrow$ φ of decaying fermion with beam

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

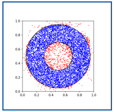
$\leftarrow$ φ of decay

$\leftarrow$ propagator of decaying fermion

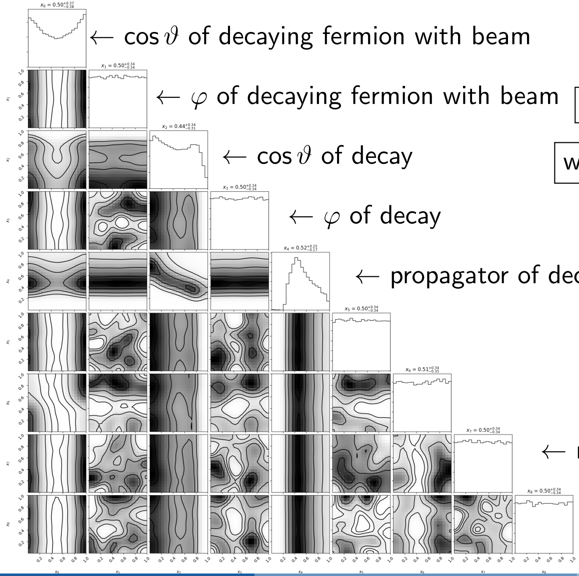
$\leftarrow$ multichannel

Target distribution

without learning color



III: 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

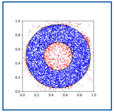
$\leftarrow \varphi$ of decay

$\leftarrow$ propagator of decaying fermion

$\leftarrow$ multichannel

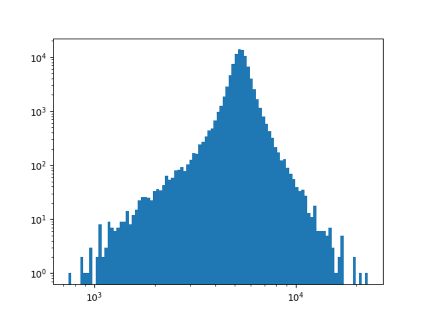
Learned distribution

without learning color



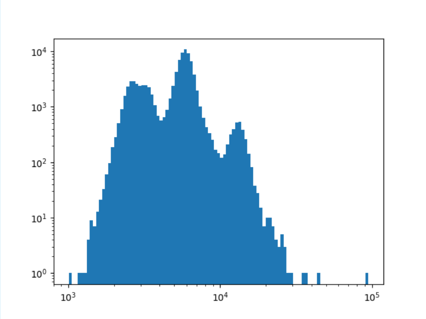
III: Comparing $e^+e^- \to 3j$ with and without learning color.

with learning color

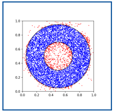


- $\sigma = 4879.8 \pm 5.3$pb
- $\eta_{\text{new}} = 45\%$
- Cut efficiency: 92 %
- 20 overweight events in 100k

without learning color

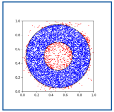


- $\sigma = 4883.5 \pm 8.5$pb
- $\eta_{\text{new}} = 25\%$
- Cut efficiency: 92 %
- 20 overweight events in 100k



III: 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



III: 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

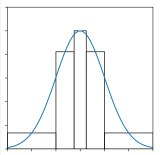
  Winkler et al. [1912.00042]; Tran et al. [1905.10347]
- "learn" the permutations: using 1×1 convolutions

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

  Chen et al. [1604.06174]
- ...

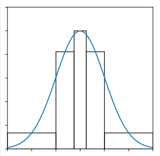
i-flow: Numerical Integration and Event Generation with Normalizing Flows

- I introduced the concepts of numerical integration with Monte Carlo techniques and importance sampling.
- I discussed "traditional" algorithms like, VEGAS or Foam.

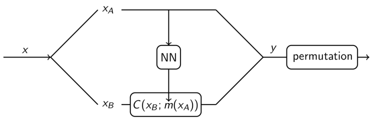


`i-flow`: Numerical Integration and Event Generation with Normalizing Flows

- I introduced the concepts of numerical integration with Monte Carlo techniques and importance sampling.
- I discussed "traditional" algorithms like, `VEGAS` or `Foam`.

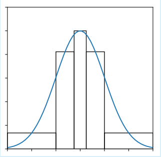


- I introduced Machine Learning and discussed two approaches to event generation: learning $q(x)$ vs. GANs
- I presented the idea of Normalizing Flows.

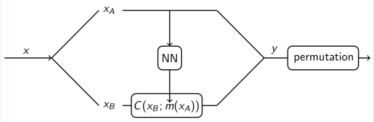


`i-flow`: Numerical Integration and Event Generation with Normalizing Flows

- I introduced the concepts of numerical integration with Monte Carlo techniques and importance sampling.
- I discussed "traditional" algorithms like, `VEGAS` or `Foam`.



- I introduced Machine Learning and discussed two approaches to event generation: learning $q(x)$ vs. GANs
- I presented the idea of Normalizing Flows.



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