Improving Numerical Integration and Event Generation with Normalizing Flows

— HET Brown Bag Seminar, University of Michigan —

Claudius Krause

Fermi National Accelerator Laboratory

September 25, 2019

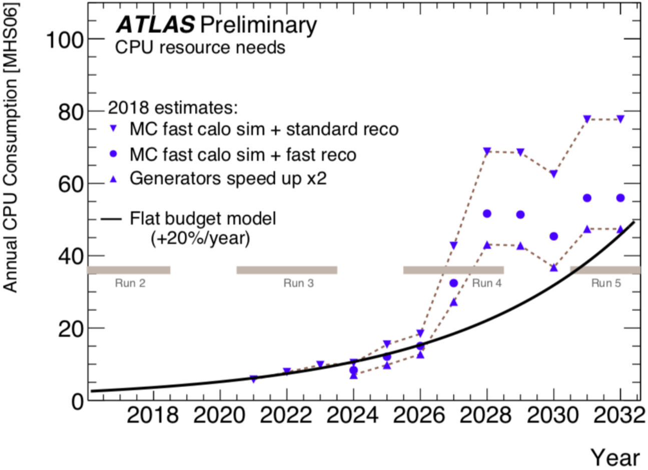
Unterstützt von / Supported by

Fermilab

Alexander von Humboldt
Stiftung/Foundation

In collaboration with: Christina Gao, Stefan Höche, Joshua Isaacson
arXiv: 191x.abcde

Monte Carlo Simulations are increasingly important.

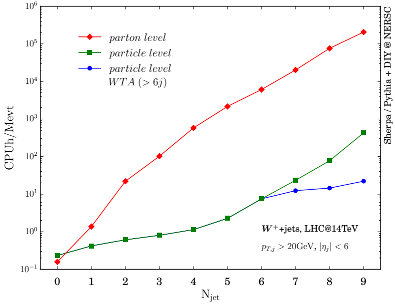


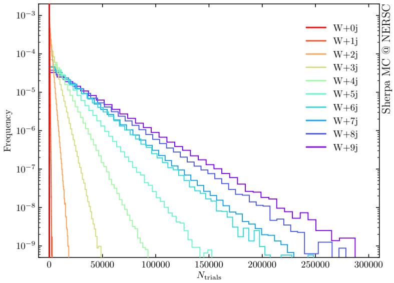
https://twiki.cern.ch/twiki/bin/view/AtlasPublic/ComputingandSoftwarePublicResults

⇒ 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.



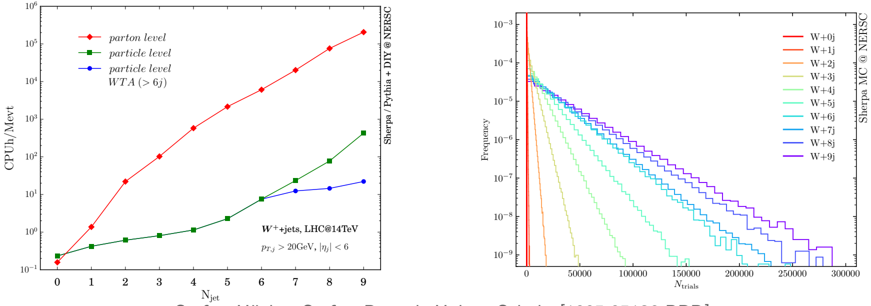


Stefan Höche, Stefan Prestel, Holger Schulz [1905.05120;PRD]

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.



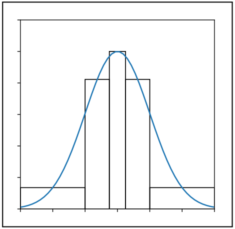
Stefan Höche, Stefan Prestel, Holger Schulz [1905.05120;PRD]

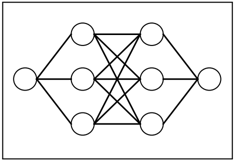
The bottlenecks for evaluating large final state multiplicities are

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

Improving Numerical Integration and Event Generation with Normalizing Flows

Part I: The “traditional” approach



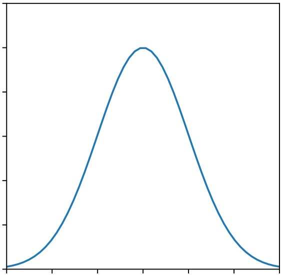


Part II: The Machine Learning approach

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

$f(\vec{x})$

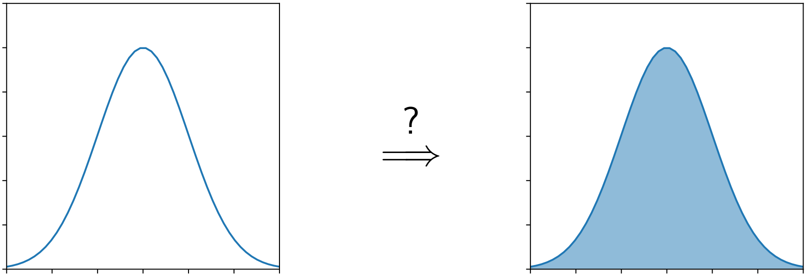
$d\sigma(p_i, \vartheta_i, \varphi_i)$



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

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

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

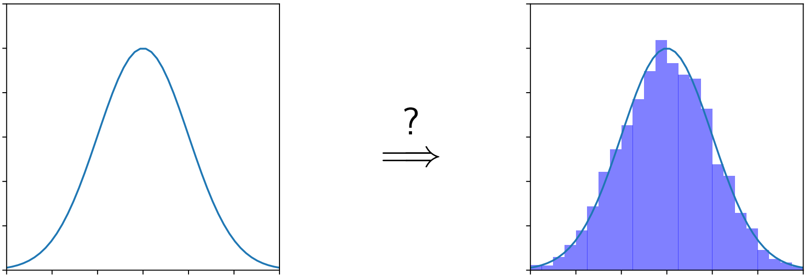


I: There are two problems to be solved...

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

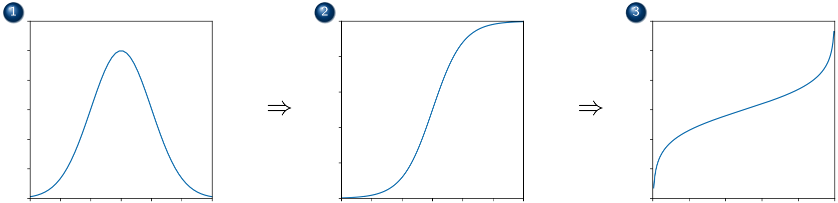
$$d\sigma(p_i, \vartheta_i, \varphi_i) \quad \Rightarrow \quad \sigma = \int d\sigma(p_i, \vartheta_i, \varphi_i), \quad D = 3n_{\mathrm{final}} - 4$$

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



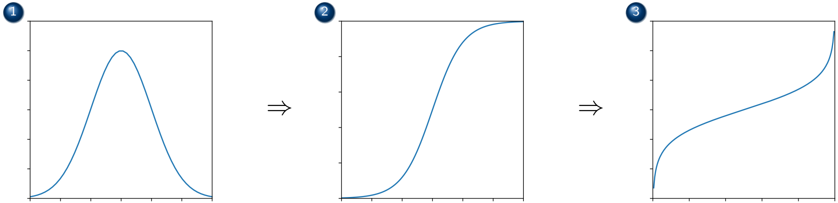
I: . . . but they are closely related.

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.

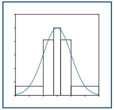


I: . . . but they are closely related.

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.



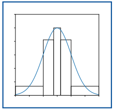
⇒ We need a fast and effective numerical integration!



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

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

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



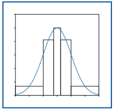
I: Importance Sampling is very efficient for high-dimensional integration.

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

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

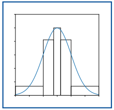
We therefore have to find a $q(x)$ that

- 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 $q(x)$ were constant, each event x_i would require a weight of $f(x_i)$ to reproduce the distribution of $f(x)$. $\Rightarrow$ "Weighted 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 $f(x)$.

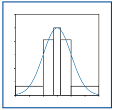


I: The unweighting efficiency measures the quality of the approximation $q(x)$.

- If $q(x)$ were constant, each event x_i would require a weight of $f(x_i)$ to reproduce the distribution of $f(x)$. $\Rightarrow$ "Weighted 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 $f(x)$.

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



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

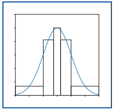
- If $q(x)$ were constant, each event x_i would require a weight of $f(x_i)$ to reproduce the distribution of $f(x)$. $\Rightarrow$ "Weighted Events"

- 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 $f(x)$.

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

We define the $\boxed{\text{Unweighting Efficiency} = \frac{\#\ \text{accepted events}}{\#\ \text{all events}} = \frac{\text{mean } w}{\max w}}$,

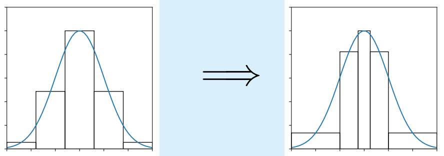
with $w_i = \frac{p(x_i)}{q(x_i)} = \frac{f(x_i)}{Fq(x_i)}$.

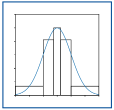


I: The VEGAS algorithm is very efficient.

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.

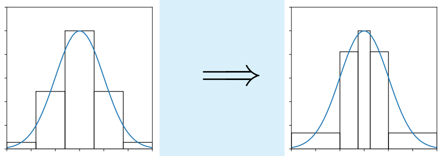




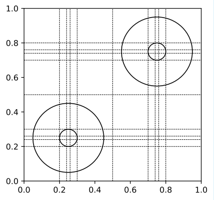
I: The VEGAS algorithm is very efficient.

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.

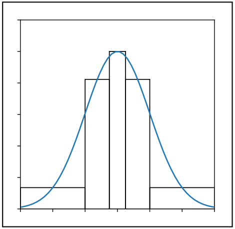


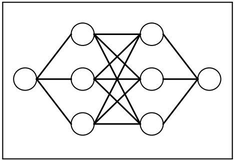
- It does have problems if the features are not aligned with the coordinate axes.
- The current python implementation also uses stratified sampling.



Improving Numerical Integration and Event Generation with Normalizing Flows

Part I: The “traditional” approach

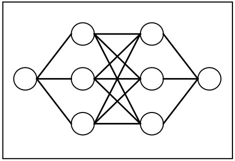


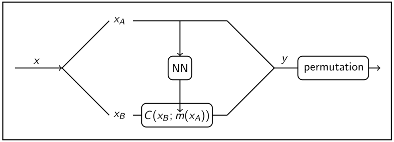


Part II: The Machine Learning approach

Part II: The Machine Learning approach

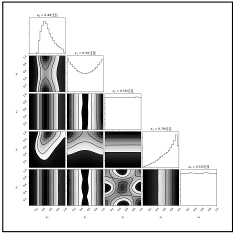
Part II.1: Neural Network Basics





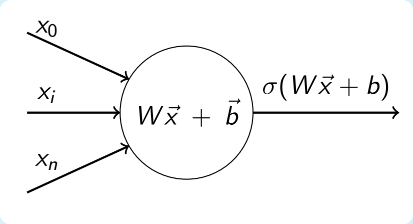
Part II.2: Numerical Integration with Neural Networks

Part II.3: Examples

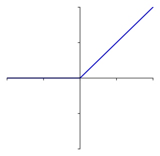


II.1: Neural Networks are nonlinear functions, inspired by the human brain.

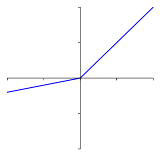
Each neuron transforms the input with a weight W and a bias $\vec{b}$.



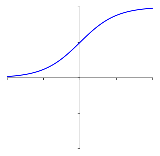
The activation function σ makes it nonlinear.



"rectified linear unit (relu)"



"leaky relu"



"sigmoid"

II.1: The Loss function quantifies our goal.

We have two choices:

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

They give the gradient that is needed for the optimization:

$$\nabla_\theta D_{(KL \text{ or } \chi^2)} \approx -\frac{1}{N} \sum \left(\frac{p(x_i)}{q(x_i)} \right)^{(1 \text{ or } 2)} \nabla_\theta \log q(x_i), \qquad x_i \dots q(x)$$

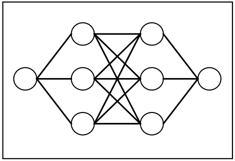
We use the ADAM optimizer for stochastic gradient descent:

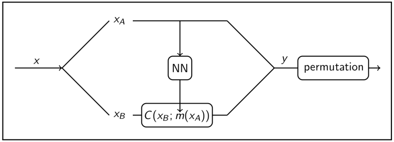
- The learning rate for each parameter is adapted separately, but based on previous iterations.
- This is effective for sparse and noisy functions.

Kingma/Ba [arXiv:1412.6980]

Part II: The Machine Learning approach

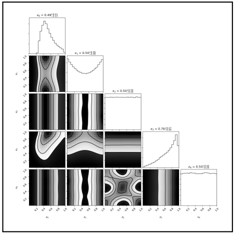
Part II.1: Neural Network Basics





Part II.2: Numerical Integration with Neural Networks

Part II.3: Examples



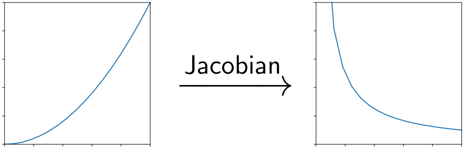
II.2: Using the NN as coordinate transform is too costly.

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]



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

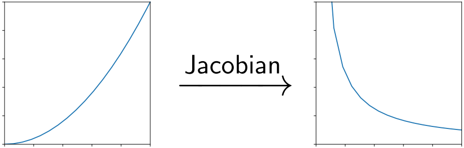
II.2: Using the NN as coordinate transform is too costly.

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, the integral, and to sample. However, it scales as $\mathcal{O}(n^3)$ and is too costly for high-dimensional integrals!

II.2: Normalizing Flows are numerically cheaper.

A Normalizing Flow:

- is a deterministic, 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.2: Normalizing Flows are numerically cheaper.

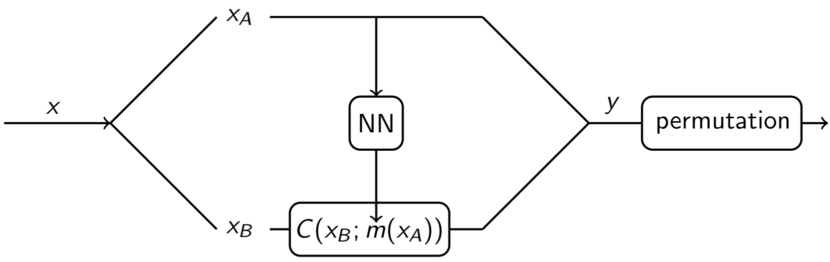
A Normalizing Flow:

- is a deterministic, 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.

⇒ 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.
- Our approach follows the ideas of Müller et al. [arXiv:1808.03856], but with the modifications of Durkan et al. [arXiv:1906.04032].
- Our code uses `TensorFlow 2.0-beta`, `www.tensorflow.org`.

II.2: 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:

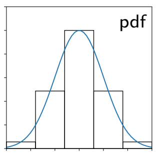
$$\left|\frac{\partial y}{\partial x}\right| = \begin{vmatrix} 1 & \frac{\partial C}{\partial x_A} \\ 0 & \frac{\partial C}{\partial x_B} \end{vmatrix} = \Pi_i \frac{\partial C(x_{B,i}; m(x_A))}{\partial x_{B,i}}$$

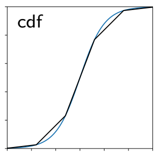
$$\Rightarrow \mathcal{O}(n)$$

II.2: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]



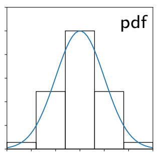


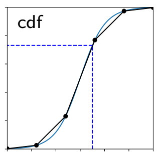
The NN predicts the pdf bin heights Q_i.

II.2: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]





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

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

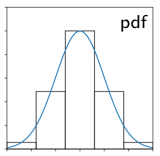
$$\left|\frac{\partial C}{\partial x_B}\right| = \Pi_i \frac{Q_{b_i}}{w}$$

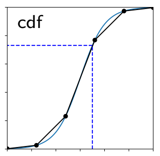
The NN predicts the pdf bin heights Q_i.

II.2: The Coupling Function is a piecewise approximation to the cdf.

piecewise linear coupling function:

Müller et al. [arXiv:1808.03856]





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

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

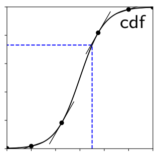
$$\left|\frac{\partial C}{\partial x_B}\right| = \Pi_i \frac{Q_{b_i}}{w}$$

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 $a_i \& b_i$.

II.2: 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.* (ppppt t)$\leftrightarrow$(ppttpp)$\leftrightarrow$(ttpppp)

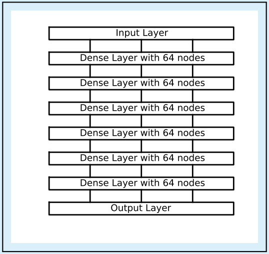
- More pass-through dimensions (p) means more points required for accurate loss.
- Fewer pass-through dimensions means more CLs needed.

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

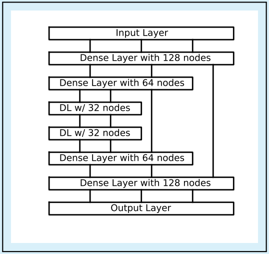
II.2: We utilize different NN architectures.

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

"Fully Connected" Neural Net (NN):



"U-shaped" Neural Net (Unet):

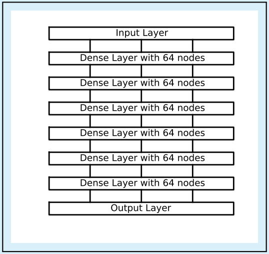


II.2: We utilize different NN architectures.

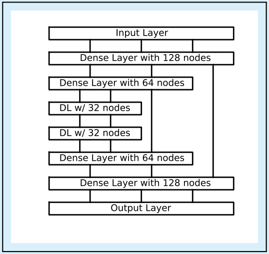
Available Architectures:

Müller et al. [arXiv:1808.03856]

"Fully Connected" Neural Net (NN):



"U-shaped" Neural Net (Unet):



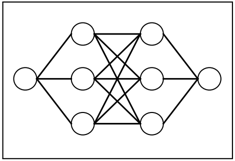
There are different ways to encode the input dimensions x_A.
For example $x_A = (0.2, 0.7)$:

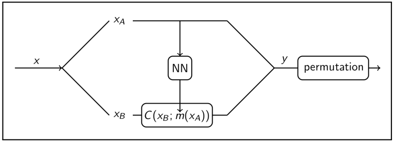
- direct: $x_i = (0.2, 0.7)$
- one-hot (8 bins): $x_i = ((0, 1, 0, 0, 0, 0, 0, 0), (0, 0, 0, 0, 0, 1, 0, 0))$
- one-blob (8 bins): $x_i = ((0.55, 0.99, 0.67, 0.16, 0.01, 0, 0, 0),$
  $(0, 0, 0.01, 0.11, 0.55, 0.99, 0.67, 0.16))$

Müller et al. [arXiv:1808.03856]

Part II: The Machine Learning approach

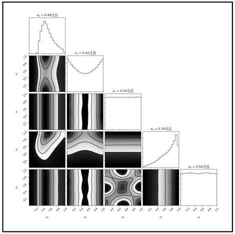
Part II.1: Neural Network Basics





Part II.2: Numerical Integration with Neural Networks

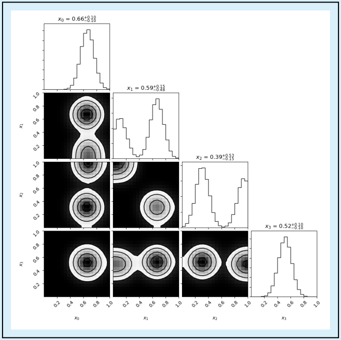
Part II.3: Examples



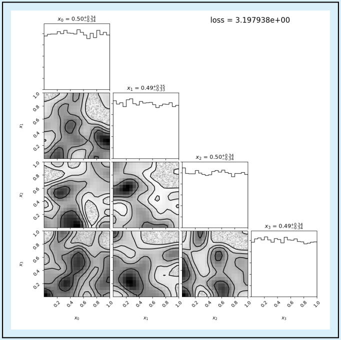
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



Before training:

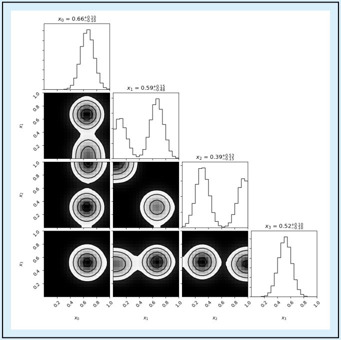


- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 % Untrained efficiency: 0.6 %

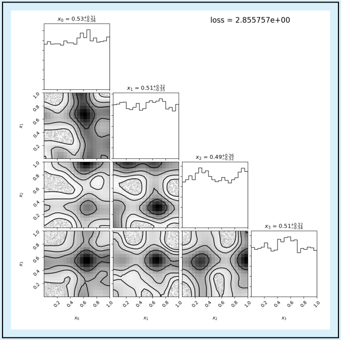
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



After 5 epochs:

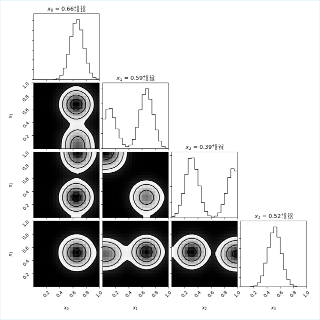


- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 %   Untrained efficiency: 0.6 %

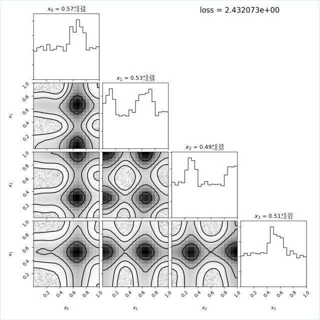
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



After 10 epochs:

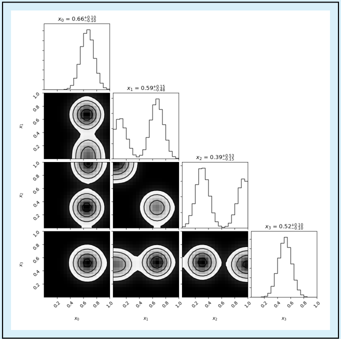


- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 %   Untrained efficiency: 0.6 %

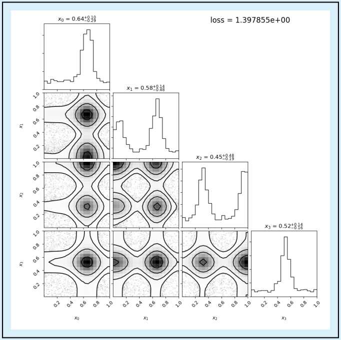
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



After 25 epochs:

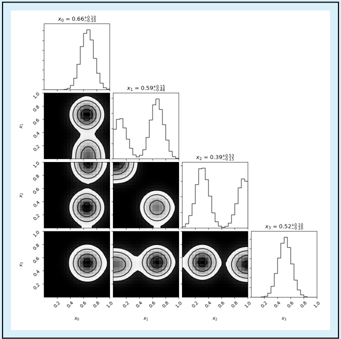


- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 %   Untrained efficiency: 0.6 %

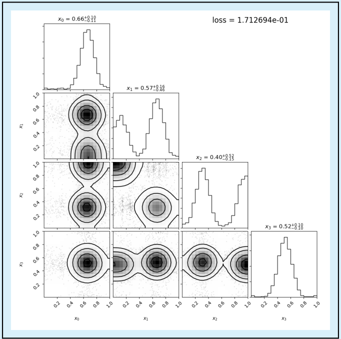
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



After 100 epochs:

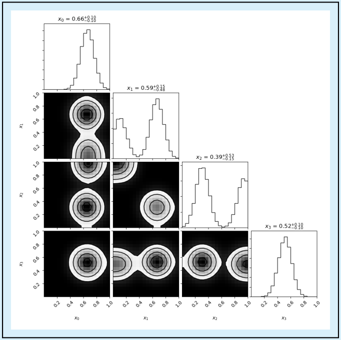


- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 % Untrained efficiency: 0.6 %

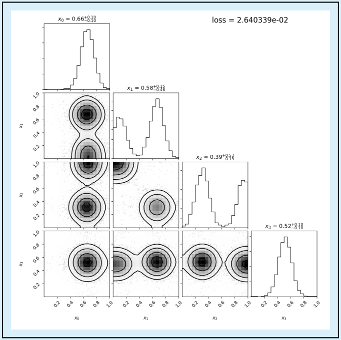
II.3: The 4-d Camel function illustrates the learning of the NN.

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

Target Distribution:



After 200 epochs:



- Final Integral: 0.0063339(41)
- `VEGAS` plain: 0.0063349(92)
- `VEGAS` full: 0.0063326(21)
- Trained efficiency: 14.8 %   Untrained efficiency: 0.6 %

II.3: 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

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

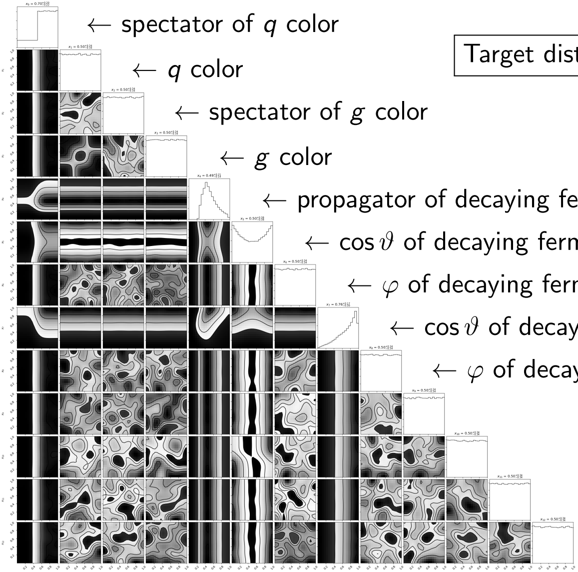
$$\Rightarrow \quad n_{dim} = \underbrace{3n_{final} - 4}_{\text{kinematics}} + \underbrace{n_{final} - 1}_{\text{multichannel}}$$

- compute the matrix element of the process. The `COMIX++` ME-generator uses color-sampling, so we need to integrate over final state color configurations, too.

$$\boxed{\Rightarrow n_{dim} = 4n_{final} - 3 + 2(n_{color})}$$

https://sherpa.hepforge.org/

II.3: Already in $e^+e^- \to 3j$ we are more effective.



$\leftarrow$ spectator of q color

$\leftarrow$ q color

$\leftarrow$ spectator of g color

$\leftarrow$ g color

$\leftarrow$ propagator of decaying fermion

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

$\leftarrow$ φ of decaying fermion with beam

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

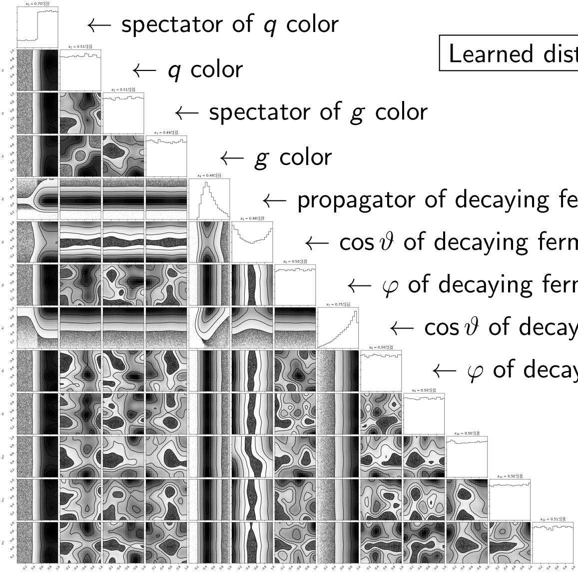
$\leftarrow$ φ of decay

$\leftarrow$ multichannel

Target distribution

II.3: Already in $e^+e^- \to 3j$ we are more effective.

Learned distribution



$\leftarrow$ spectator of q color

$\leftarrow$ q color

$\leftarrow$ spectator of g color

$\leftarrow$ g color

$\leftarrow$ propagator of decaying fermion

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

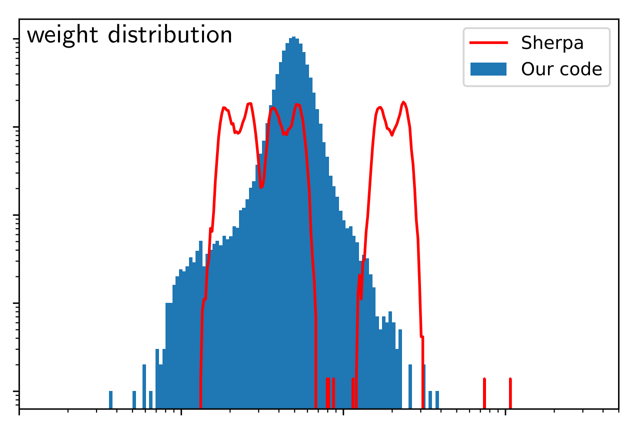
$\leftarrow$ φ of decaying fermion with beam

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

$\leftarrow$ φ of decay

$\leftarrow$ multichannel

II.3: Already in $e^+e^- \to 3j$ we are more effective.

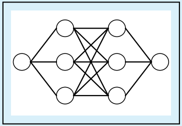


$\sigma_{\text{our code}} = 4887.1 \pm 4.6\text{pb}$ $\sigma_{\text{Sherpa}} = 4877.0 \pm 17.7\text{pb}$

unweighting efficiency $= 12.9\%$ unweighting efficiency $= 2.8\%$

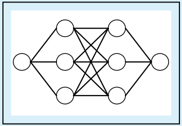
Improving Numerical Integration and Event Generation with Normalizing Flows

- I summarized the concepts of numerical integration and the "traditional" VEGAS algorithm.
- I introduced Neural Networks as versatile nonlinear functions.

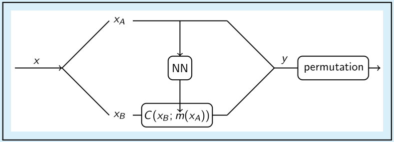


Improving Numerical Integration and Event Generation with Normalizing Flows

- I summarized the concepts of numerical integration and the "traditional" VEGAS algorithm.
- I introduced Neural Networks as versatile nonlinear functions.

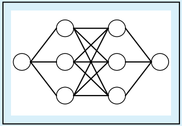


- I presented the idea of Normalizing Flows.
- I discussed their superiority for large integration dimensions.

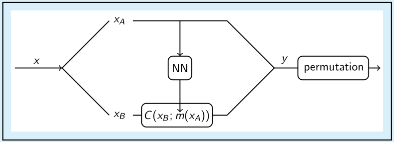


Improving Numerical Integration and Event Generation with Normalizing Flows

- I summarized the concepts of numerical integration and the "traditional" VEGAS algorithm.
- I introduced Neural Networks as versatile nonlinear functions.



- I presented the idea of Normalizing Flows.
- I discussed their superiority for large integration dimensions.



- I showed the results of two different examples
- In $e^+e^- \to 3j$, we "beat" Sherpa by roughly a factor of 5.