Event Generation with Neural Nets

Matthew D. Klimek
Cornell Univ.
Korea Univ.

Work with M. Perelstein 1805.xxxxx

MC4BSM, IPPP Durham, 21 Apr 2018

The general problem of MC integration/generation

A model of particle physics provides predictions in the form of (differential) cross sections.

Comparison of models to data is facilitated by the calculation of total cross sections and the generation of simulated data sets.

Simulated data should be a set of points in phase space (events), distributed according to the probability density function (pdf) specified by the differential cross section.

Since the differential cross sections are typically complicated functions, Monte Carlo techniques are the only feasible way of handling these operations.

The general problem of MC integration/generation

The basic technique is to sample the domain of the function randomly (uniformly in the simplest case) and sum the function values at the N random points. Estimates of the integral and the error are then obtained as:

$$I \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \qquad \sigma^2 \approx \frac{1}{N^2} \left(\sum_{i=1}^{N} f^2(x_i) - \left(\sum_{i=1}^{N} f(x_i) \right)^2 \right)$$

For event generation, random points are drawn and the function is used to decide whether the event should be kept or not (unweighting).

$$\epsilon(x) = \frac{f(x)}{f_{\max}}$$

Each sample requires one evaluation of the function, many of which may be discarded, so the unweighting can become computationally expensive.

Importance Sampling

If the function has large regions of small probability or sharply peaked regions, this introduces a lot of error in the integration, and inefficient event generation, as the important areas will not be sampled often.

This is often the case with physical processes that have strong enhancements in certain kinematic configurations (collinear enhancements, intermediate resonances, etc.).

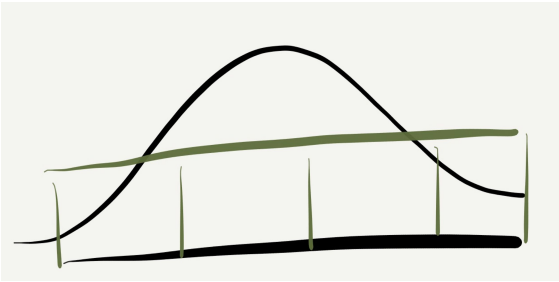
Importance Sampling introduces an algorithm to build a function which:

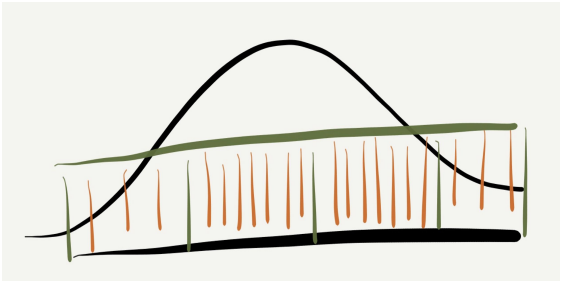
- ➢ is easily sampled, and
- ➢ approximates the target function.
  That is, the function will be sampled most often where it is largest, decreasing integration error and increasing unweighting efficiency.

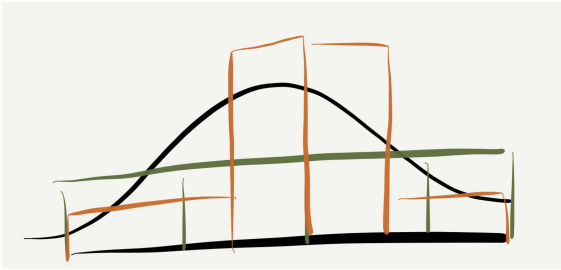
VEGAS

VEGAS (G.P. Lepage, *J.Comp.Phys.* 1978), is an importance sampling technique still in use today (e.g. MadEvent):

- ➢ Approximate the function by a set of bins of containing equal amounts of the integral of the function.
- ➢ To sample the function, simply choose a random bin and then sample uniformly within that bin.
- ➢ Adaptively choose bin edges to best match the function:





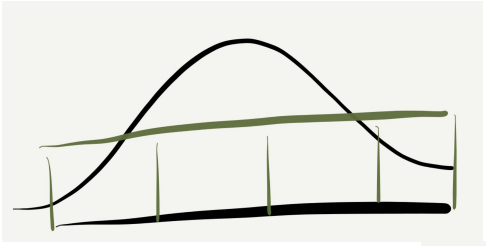


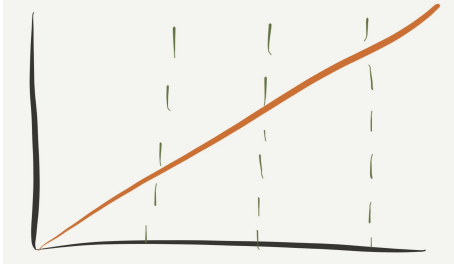
VEGAS as ML

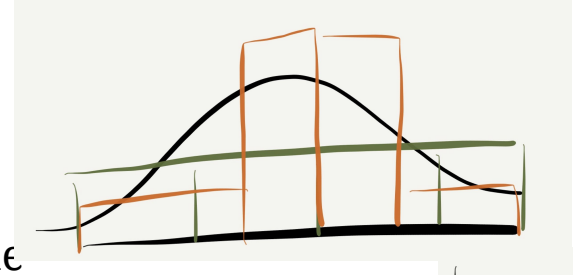
VEGAS is a form of machine learning.

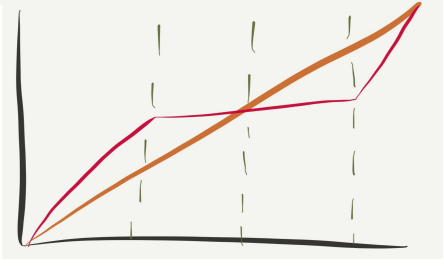
The algorithm goes through a training process where it is allowed to adjust the location of the bin edges, in order to decrease the variance in the number of points that land in each bin.

It builds a map from a sampling space (a space over which points are drawn uniformly) onto the target space (a space where the density of points approximates the desired function) in a piecewise-linear way. **The training adjusts the values of the map at fixed discrete points.**









VEGAS as ML

What if we could extend this to adjust the map at **every** point?

We would need a map, that is defined in terms of some adjustable parameters, that is capable of approximating **any** smooth map.

Universal approximation theorem: Given any continuous function *f(x)* on the *N*-cube, and any $\epsilon > 0$, *f(x)* can be approximated by a function *F(x)*

$$|F(x) - f(x)| < \epsilon$$

where

$$F(x) = \sum_{i=1}^{N} v_i \, A(w_i x_i + b_i)$$

and *A* is a bounded, non-uniform, monotonically increasing function.

This is the basis of artificial neural nets.

General Approach

First suggested by Bendavid 1707.00028, applied only to Gaussian functions, see MC4BSM 2017

The neural net N will be a map from a sampling space $X = \mathbf{R}^d$ to phase space $Y = \mathbf{R}^d$, where d is the appropriate dimensionality of phase space for the process under consideration.

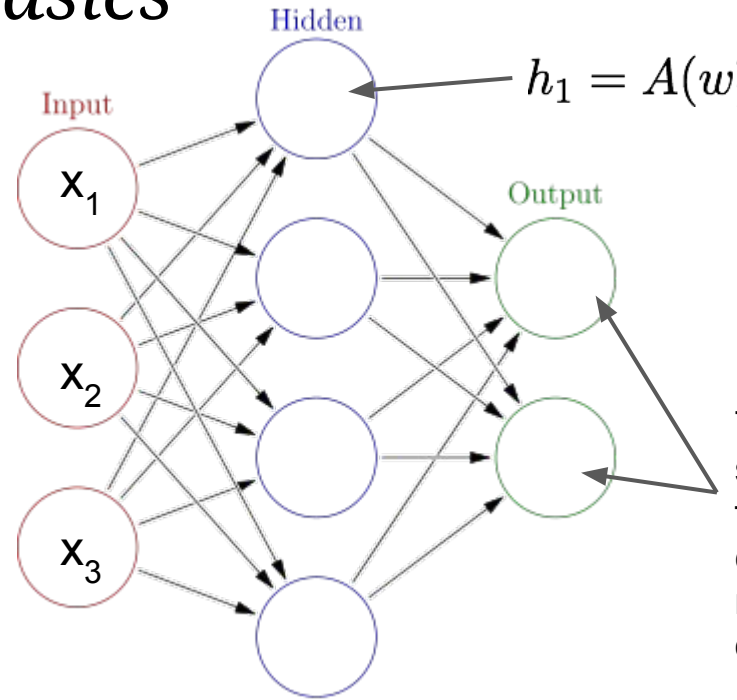
Sample uniformly over X. Then the distribution, which is supposed to approximate the differential cross section, induced on Y will be the inverse of the Jacobian of N: $\mathcal{J}^{-1}(N)|_{x=x(y)} = \left|\frac{\partial N}{\partial x(y)}\right|^{-1}$

Choose a measure of statistical distance between the true differential cross section and *1/*Jac(N). We used the Kullbeck-Leibler divergence, which is zero for two distributions f and g only when $f = g$ and is positive otherwise.

$$D_{KL}(f;g) = \int f(x) \log\left(\frac{f(x)}{g(x)}\right) dx$$

The algorithm should adjust N so that the D_{KL} between *1/*Jac(N) and the differential cross section is minimized (ideally to zero).

NN Basics



Each hidden node takes a linear combination of the inputs, specified by the *weights* w_1^i plus a constant *bias* b_1, and transforms it by some non-linear *activation function A.*

The weights and biases together comprise the **parameters** of the net.

The output layer is similar, but its activation function should be chosen to map any real number onto the desired output space.

Designing a neural net consists of choosing the number of layers, hidden nodes, activation functions, loss functions, and training algorithms.

Loss function: gives a measure of how far the output of the net is from the goal when evaluated over some training data with known desired outputs.

Train by adjusting each parameter proportionally to the gradient of the loss function w.r.t. that parameter (gradient descent).

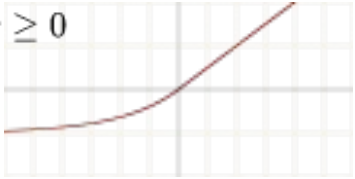
Our basic implementation

- Same number of input and output nodes ($\mathbf{R}^d \rightarrow \mathbf{R}^d$). Number of hidden nodes and hidden layers to be determined by studying performance.
- Loss function will be K-L divergence of the net's Jacobian with respect to the target differential cross section, as described earlier.

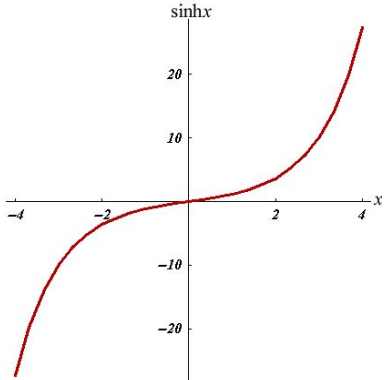
The Jacobian contains derivatives with respect to the net, so the net's components, specifically the activation functions, must be differentiable. (Many common choices are not.) We consider two candidates:

"Exponential Linear Unit" with $\alpha = 1$:

$$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$



or Sinh:



Choice of coordinates

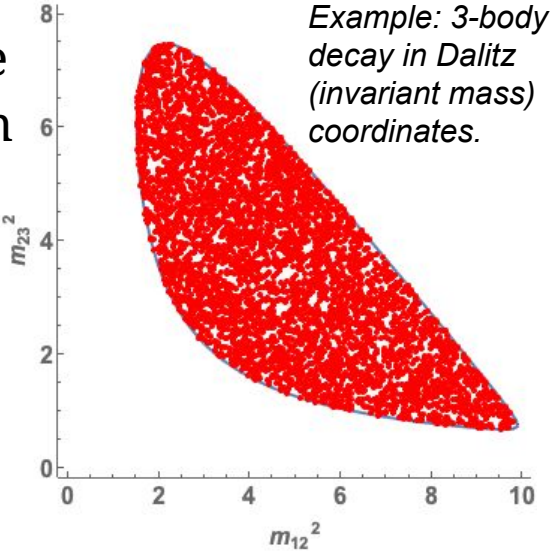
In principle, any choice of coordinates on phase space could be used, but many choices present difficulties in practice.

In many coordinate systems, the physical region of phase space has some non-trivial shape.

The net would need to learn:

- ➢ the correct distribution within the physical region
- ➢ but also where the boundary is, and to not populate anything outside (unphysical events).

However, the net is made of smooth functions so such behavior is not possible.



Example: 3-body decay in Dalitz (invariant mass) coordinates.

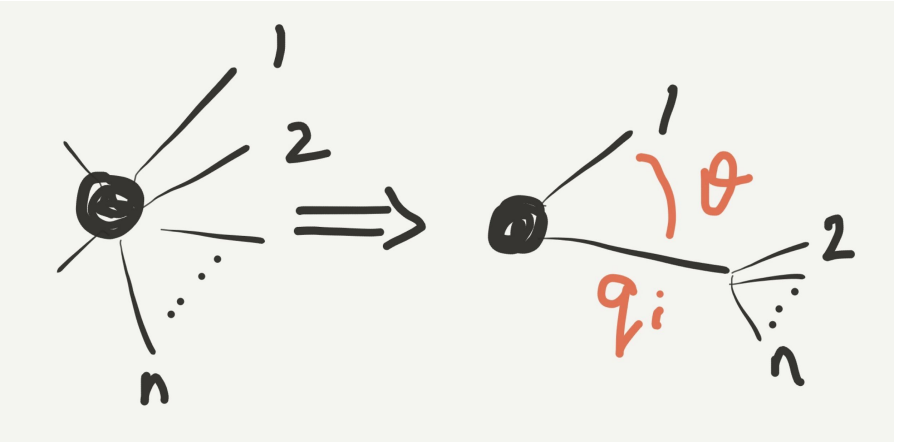
Choice of coordinates

Solution: use coordinates in which the physical region is a unit hypercube, and an output function that maps $\mathbf{R}^n$ onto it. Then all outputs of the net are guaranteed to be physical.

Define $q_i \in [0,1]$ to interpolate between the minimum and maximum possible invariant masses of the system composed of particles {i+1, …, n}.

Rescale all relative angles to the range [0,1].

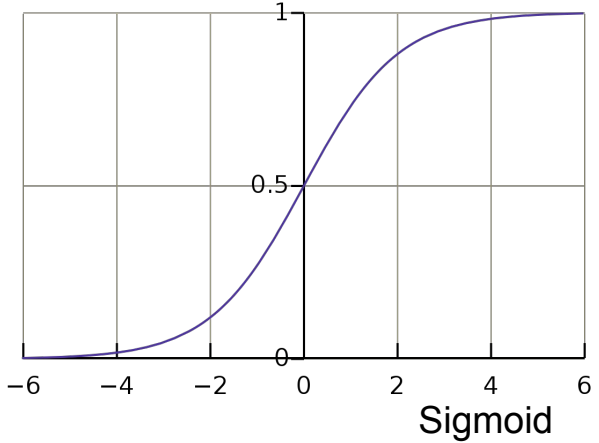
$$q_1 \in \left[\sum_{i=2}^{n} m_i, \sqrt{s} - m_1\right]$$



Output layer considerations

The internal layers of the NN may return any real values, so the output layer should contain a final function that maps onto the unit interval.

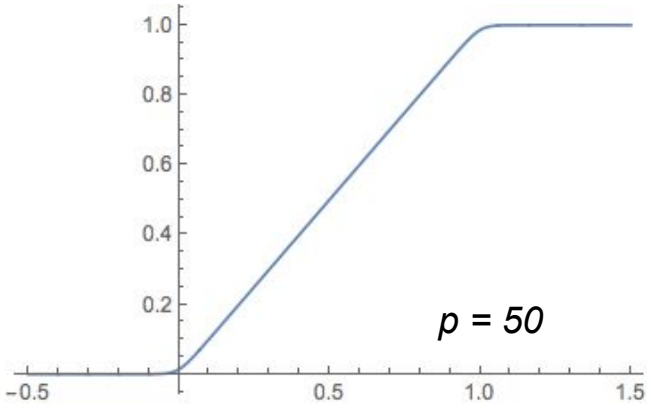
A common choice of output function is the Sigmoid, but it approaches 0 and 1 exponentially slowly, making it very hard to populate the edges of phase space.



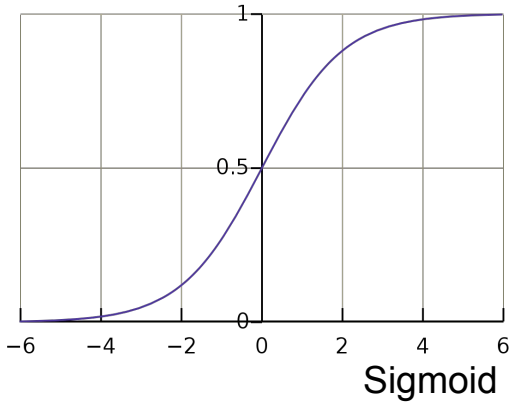
We also investigated a function with faster asymptotic behavior:

$$A(x) = \frac{1}{p} \log\left(\frac{1 + e^{px}}{1 + e^{p(x-1)}}\right)$$

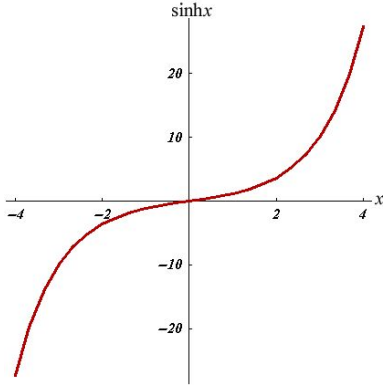
- Always takes values in [0,1]
- Approaches limiting values rapidly
- Approximately linear between [0,1]
- *p* controls how sharp the edges are



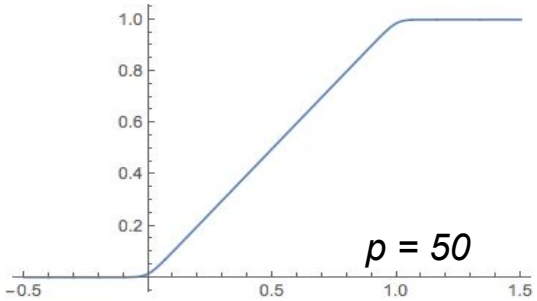
Output layer considerations

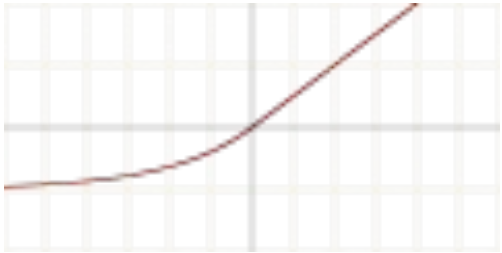


Sigmoid



For the sigmoid output function, we use a sinh activation function. The asymptotically exponential behavior of these functions cancels and allows good reach to the edges 0 and 1.





For our custom activation function which approaches 0 and 1 rapidly, a traditional ELU activation function is sufficient.

Complete setup

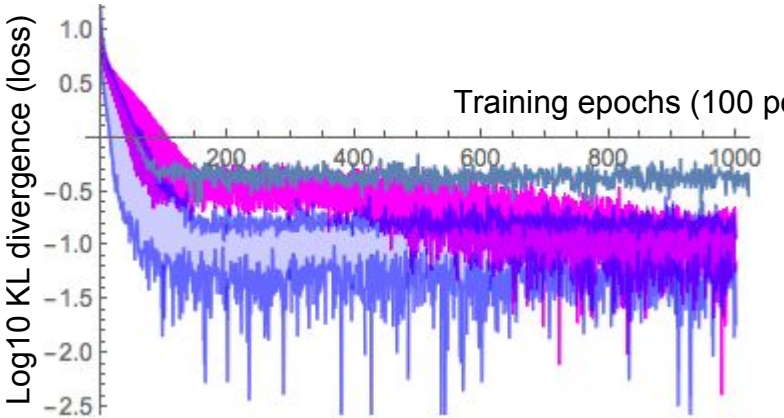
- Implemented in MXNet 1.1.0 with Python 2.7 interface.
- Various numbers of hidden nodes and layers tested, as well as different activation/output functions as described earlier.
- Physics input: simply type analytical expression for the differential cross section into the code, or provide a function that python can call. (Easy to interface with Feynman diagram calculator.)
- Training:
  - Draw a sample of 100 uniform random points.
  - Feed through the NN.
  - Compute the KL divergence between the NN output and the target diff. cross section.
  - Try to minimize KL divergence by adjusting the NN parameters according to the gradient of the KL divergence (gradient descent).
- Train until value of KL divergence stabilizes.
  - (How close to 0 does it get?)

3-body Dalitz, constant matrix element

- 2-dimensional phase space. Parametrize with:
  - m_{23} and θ, the angle between p_2 and p_1 in the m_{23} rest frame.
  - Phase space is flat in θ.
  - Both variables can be shifted/scaled to lie in a unit square.

M = sqrt(s) = 1 GeV
m_1 = 0.1 GeV
m_2 = 0.2 GeV
m_3 = 0.3 GeV

Training results with 3 layers of 64 nodes. Training to stability takes ~1 minute on a very old laptop.

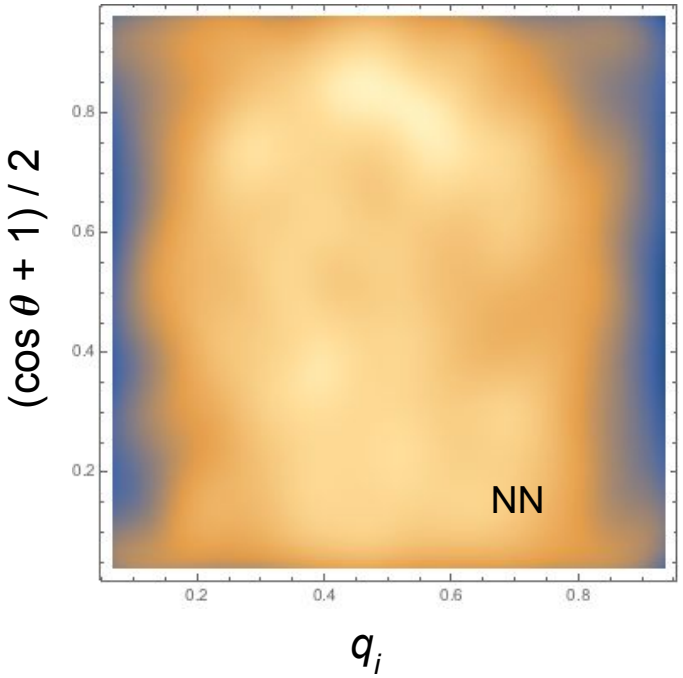


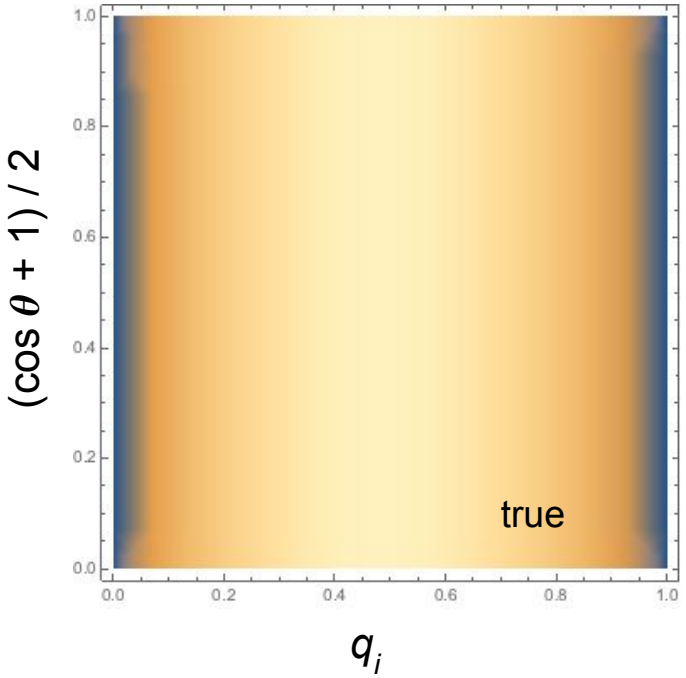
Sinh/Sigmoid
ELU/Custom activation
Bendavid 1707.00028

This is in fact more than is needed for this simple example. 3 layers of 4 nodes, or one layer of 16 nodes is sufficient and training takes ~seconds. See forthcoming paper.

3-body Dalitz

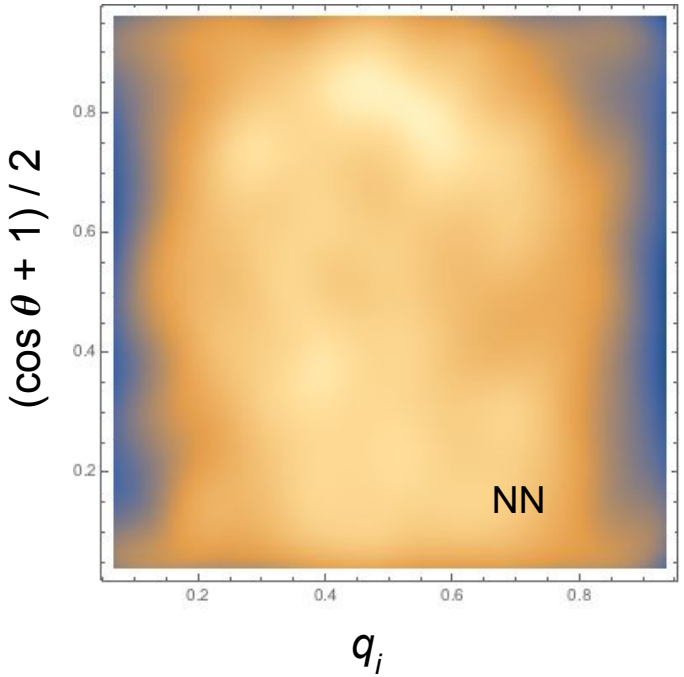
Density of events generated by trained network vs. true distribution:





3-body Dalitz

Is the NN output by itself consistent with the desired true distribution?



No: p-value ~ 10^{-4}

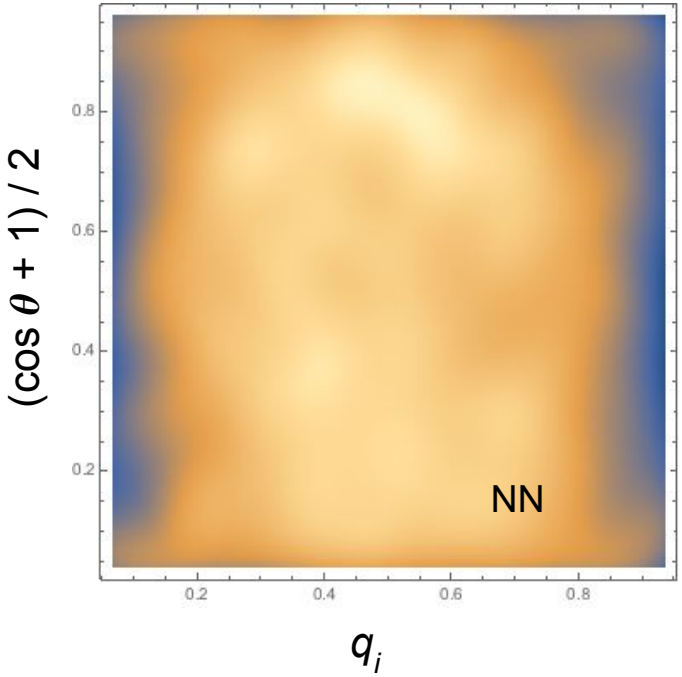
But a NN is a universal approximator. What happened?

The parameter space of the NN is very high dimensional, and there are many local minima of the loss function (KL divergence.)

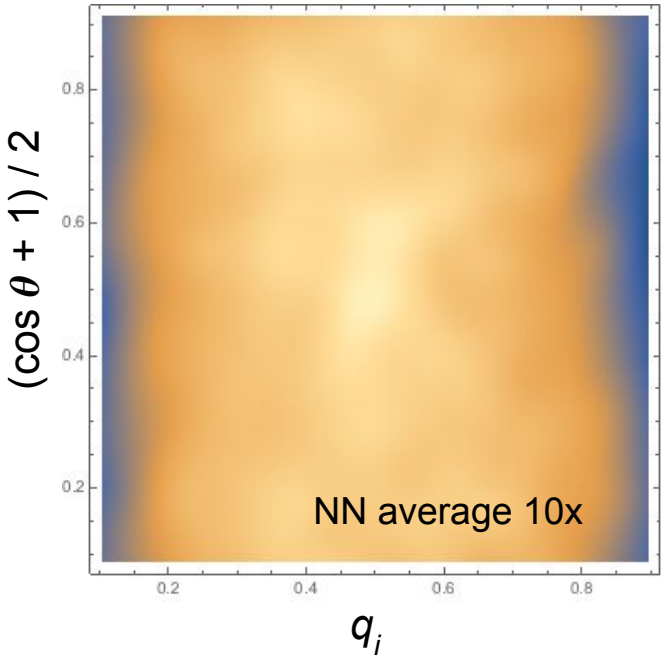
In general, one lands in a local minimum which is a good, but not great, approximation to the desired function.

3-body Dalitz

What to do about errors from "pretty good" false minima?



- Unweighting: NN is already pretty good so unweighting is quite efficient
- Average: each false minimum is a ~random deviation from the right answer. Train multiple times with different random seeds and combine results:

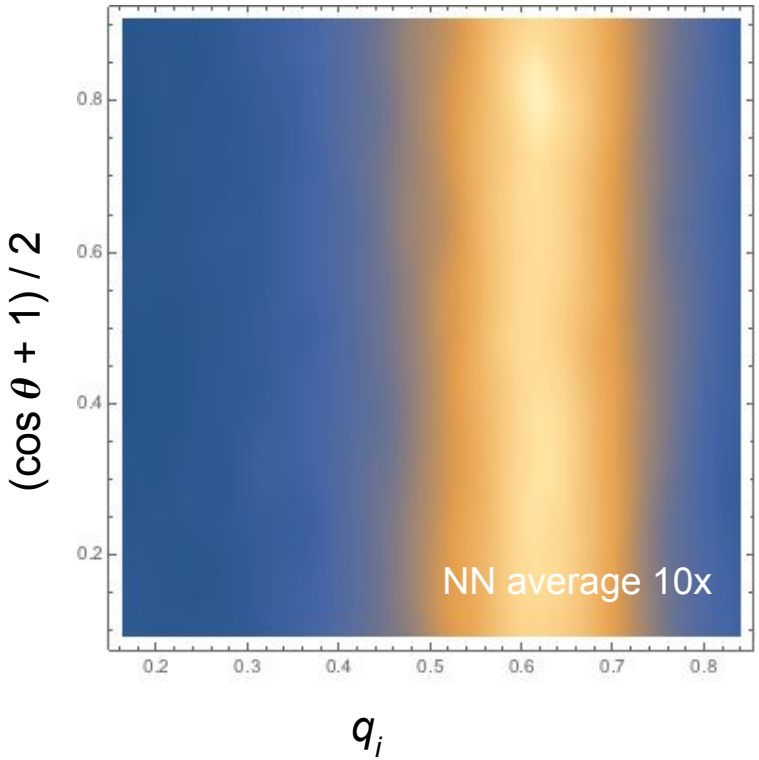


p = ~0.5 with no unweighting (100% efficiency)

3-body Dalitz with intermediate resonance

We can include a matrix element along with phase space:

3-body decay via an intermediate resonance with mass 0.75 GeV.

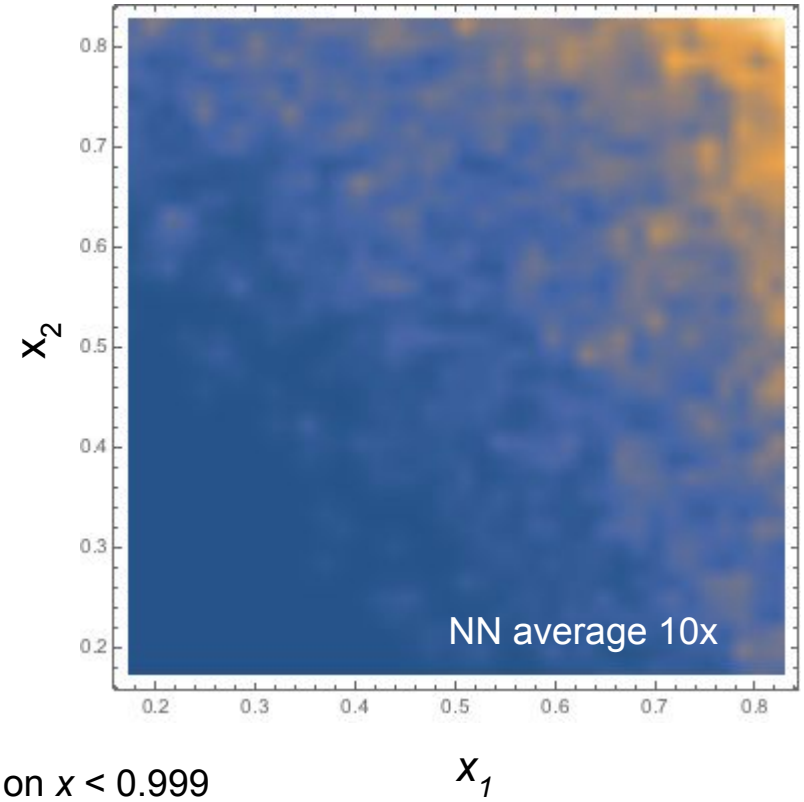


p = ~0.5

qqg

The NN also has no trouble handling singularities such as are present in *qqg* production.

$$\frac{d\sigma}{dx_1 dx_2} \propto \frac{x_1^2 + x_2^2}{(1-x_1)(1-x_2)}$$



Cut on $x < 0.999$

Parameter Scans

- Training must be performed for each parameter point.
- For these simple cases, training is cheap but can we do better?
- Yes: it is not necessary to initialize a new net with random weights for each point in the parameter scan
- After training for one point the NN has learned the “basic physics”
- It is very cheap to train *from* one parameter point *to* a nearby point
- Scans can be performed step-by-step very efficiently

Summary, and questions for the audience

- Neural Networks allow a continuum implementation of the classic VEGAS phase space integrator/generator.
- Proper choices of network architecture allow typical physical scenarios to be handled accurately and quickly

- Questions:
  - What would you accept as a fair measure of performance to compare the NN method with classic methods?
  - What are some examples of MC generation where the phase space generation is the bottleneck (so we can try those)?