Use of Machine Learning Techniques for improved Monte Carlo Integration

Josh Bendavid (Caltech/LPC)

Caltech

Jun 16, 2017
IML Meeting

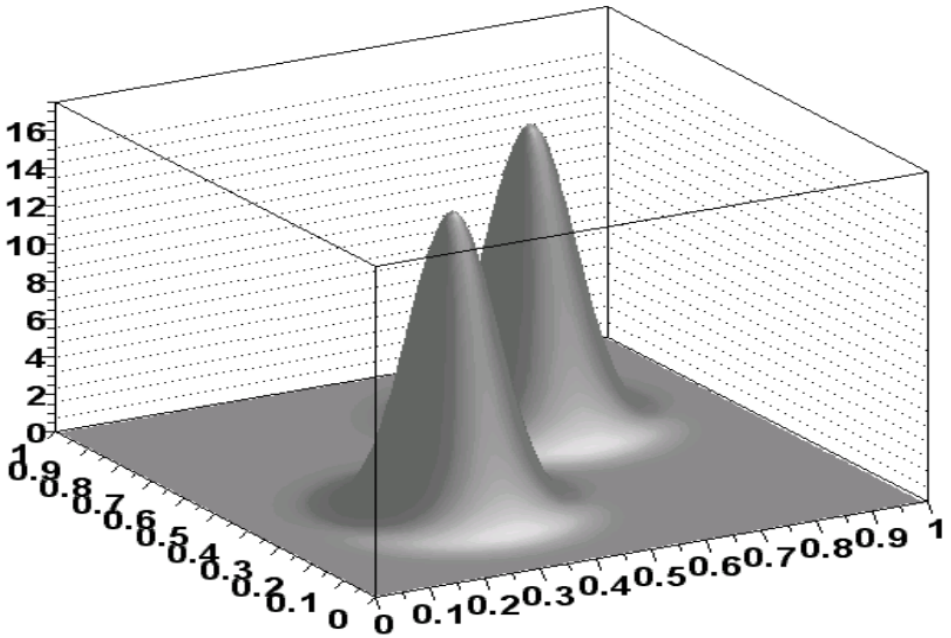
Introduction

- R&D Project Growing out of work on Multivariate Regression: New algorithms for Monte Carlo integration and event generation
- Outline
  - Brief intro to Monte Carlo integration/generation and (non-exhaustive) look at existing algorithms (VEGAS and FOAM)
  - Brief intro on Boosted Decision Trees and their conventional classification/regression applications
  - Adaptation to Monte Carlo integration/generation
  - Preliminary comparisons with existing algorithms
  - **Implementation and performance of Deep Neural Network-based alternative**
- See previous talk on BDT-based algorithm at Argonne/FNAL HPC Generators workshop last fall: `https://indico.cern.ch/event/557731/contributions/2305813/attachments/1342414/2022192/mcgbr-Sept23-2016.pdf`
- (This talk is nearly identical to what was shown a few weeks ago at MC4BSM 2017 at SLAC)

Monte Carlo Integration and Generation

- **Monte Carlo integration:** Given an arbitrary/black box multidimensional function $f(\bar{x})$, find the integral $\int f(\bar{x})d\bar{x}$
- **Monte Carlo generation:** Given an arbitrary/black box multidimensional function $f(\bar{x})$, generate an unweighted set of vectors $\bar{x}$ with a probability density $p(\bar{x}) = f(\bar{x})/\int f(\bar{x})d\bar{x}$
- Typical HEP use case: Given a numerical implementation for a matrix element fully differential in incoming/outgoing four-vectors, compute the total cross section (integral), and generate a set of unweighted events

Monte Carlo Integration and Generation: Example Function



S. Jadach, physics/0203033

- This is the "camel" function from the original VEGAS paper, which can be generalized to N dimensions
- Factorized approach will not work well
- Significant low-density regions which cannot be easily excluded a-priori

Monte Carlo Integration: Brute Force Approach

- Simplest possible algorithm:
  - Randomly sample from a (multidimensional) **Uniform** distribution
  - Integration weight $w = Vf(\bar{x})$ (where V is the total volume of the space)
  - Integral $I = (1/N)\sum w$, $\sigma_I = \sigma_w/\sqrt{N}$
  - Generation: Use simple accept-reject sampling ($\epsilon = w_{max}/ < w >$)
- End result: Huge variance for weights $\rightarrow$ need a huge number of samples to get reasonable numerical precision (and very low unweighting efficiency for generation)

Monte Carlo Integration: Importance Sampling

- General idea, sample from some generating probability density $g(\bar{x})$ instead of uniformly:
  - Integration weight $w = f(\bar{x})/g(\bar{x})$
  - Integral $I = (1/N)\sum w$, $\sigma_I = \sigma_w/\sqrt{N}$
  - Generation: Use accept-reject sampling ($\epsilon = w_{max}/ < w >$)
- Ideal case: $g(\bar{x}) = f(\bar{x})/\int f(\bar{x})d\bar{x} \rightarrow$ try to construct some $g(\bar{x})$ that is easy to sample from **and** well approximates $f(\bar{x})$
- Different considerations for integration (minimize variance) vs generation (balance between variance and maximum weight)

Monte Carlo Integration: Importance Sampling

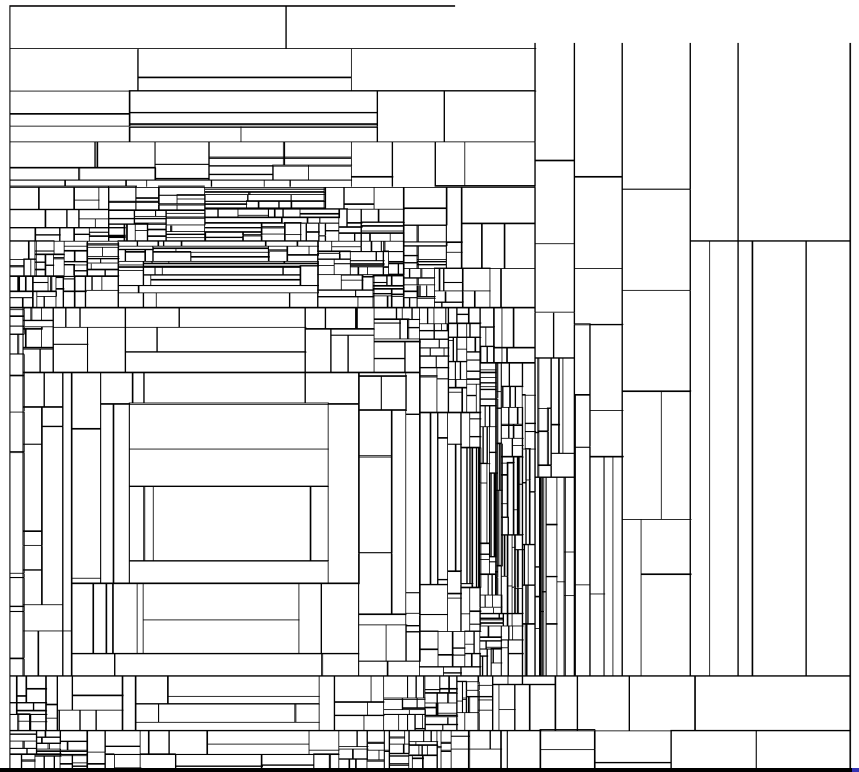
- Typical algorithm divided in two stages
  1. Construct appropriate sampling function $g(\bar{x})$ which approximates $f(\bar{x})$
  2. Generate a large number of events to evaluate the integral with maximum precision (or unweight with maximum efficiency)

VEGAS

- G.P. Lepage, A New Algorithm for Adaptive Multidimensional Integration, Journal of Computational Physics 27, 192-203, (1978)
- Iterative algorithm
- Start from uniform sampling distribution
- At each iteration, build an adaptive-binned histogram to approximate $f(\bar{x})$
- Multidimensional functions are handled as a simple product of one-dimensional histograms
- Building histograms is fast and relatively simple, but for higher-dimensional functions with non-trivial correlations there is a hard limit to the achievable weight variance/unweighting efficiency
- Carefully choosing/transforming integration basis can help (but not always possible)

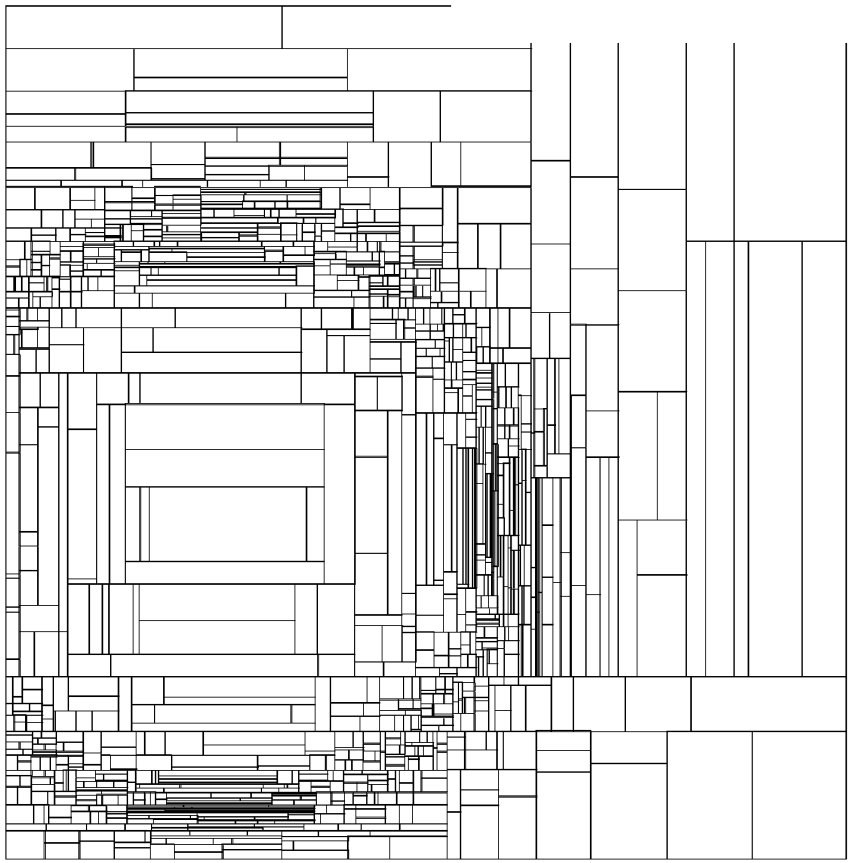
Foam

- S. Jadach, physics/0203033
- Improving on limitations of VEGAS requires true multi-dimensional sampling function
- Foam algorithm based on a single decision tree $\rightarrow$ divide up phase space into hyper-rectangles with optimized boundaries
- Phase-space is sampled uniformly **within each hyper-rectangle** to determine the next binary split until the stopping condition is reached

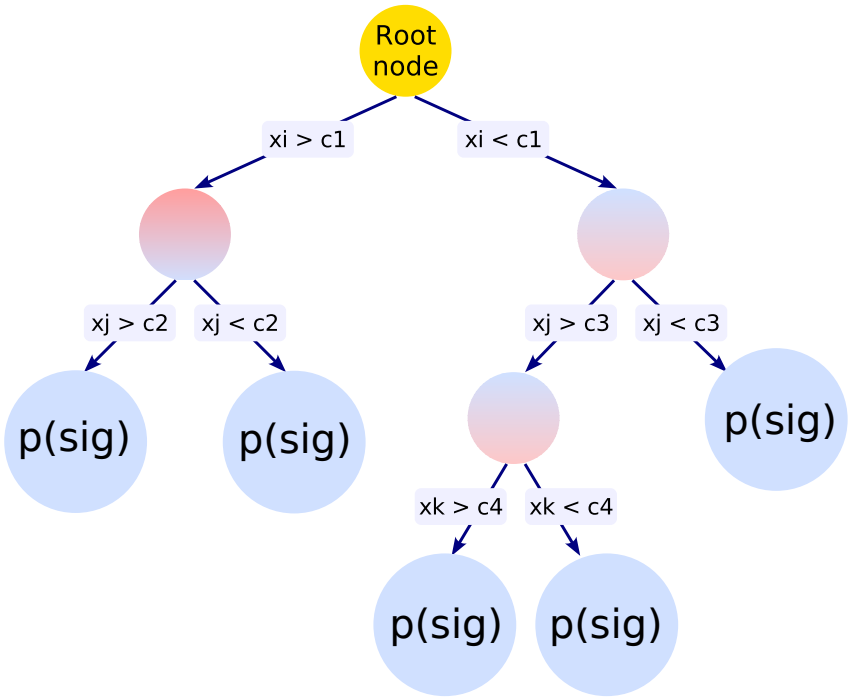


Foam

- Assign a weight to each hyper-rectangle (proportional to estimated integral inside)
- For each event: randomly choose a hyper-rectangle (probability proportional to its weight) then randomly sample within hyper-rectangle



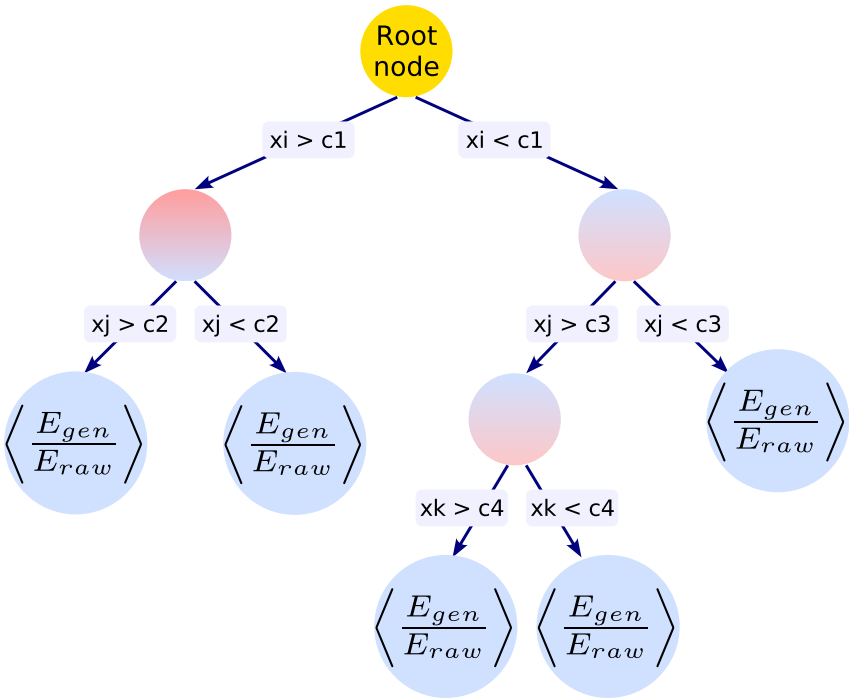
Boosted Decision Trees for Classification



- Decision Tree is a simple structure consisting of a set of connected "nodes"

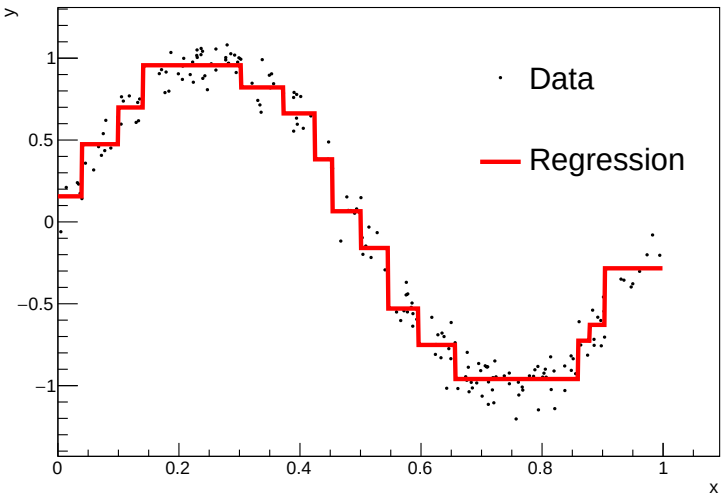
- Intermediate nodes where a variable and cut value is selected to split events into two subsets
- Terminal nodes are assigned a response, in this case the relative signal probability $\frac{\mathcal{L}_s(\vec{x})}{\mathcal{L}_b(\vec{x})}$
- Multidimensional likelihood ratio is therefore approximated by a piecewise-continuous function over the multivariate input space
- **Boosting**: Construct a series of decision trees to improve the overall response

Boosted Decision Trees for Regression

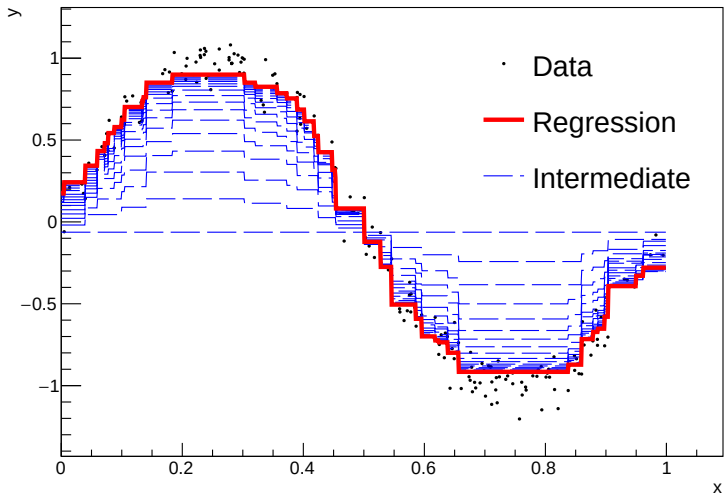


- Boosted Decision Trees can also be used for multivariate regression problem
- Replace log likelihood ratio with generic function $f(\bar{x})$
- Minimize deviation between training sample and regression function
- Decision trees form a series of piecewise continuous approximations for the function $f(\bar{x})$ in the multidimensional input space

Gradient Boosting



(a) Single Tree



(b) Gradient Boosted ($\sim$ 20 trees)

- Decision trees form an additive series of piecewise continuous approximations for the function $f(\bar{x})$ in the multidimensional input space
- Additive series can represent more complex functions than single tree with a given number of nodes
- Trivial example of Sine in 1d with relatively few trees

Boosted Decision Trees for Monte Carlo Integration (aka GBRIntegration)

- Evaluating amplitudes is the critical computational step for phase space integration or unweighting in MC generators
- Number of required phase space points depends on weights variance and/or unweighting efficiency
- **Insight:** Foam is based on a single decision tree, performance of MC integration can be improved by boosting as for classification and regression
- Basic limitation of Foam is that huge number of hyper-rectangles are needed for good performance
- Boosting allows to exploit combinatorics of terminal nodes between different decision trees
- Initial implementation based on GBRLikelihood tool developed for CMS photon energy regression

Boosted Decision Trees for Monte Carlo Integration (aka GBRIntegration)

- Basic Principle: Use a boosted decision tree to **directly** estimate function value $f(\bar{x})$ such that

$$g(\bar{x}) = \sum g_i(\bar{x}) \approx f(\bar{x}) \tag{1}$$

- Where each $g_i(\bar{x})$ is an individual decision tree with some value assigned to each terminal node
- Trivial to compute integral for each tree (and for sum): $\int g_i(\bar{x}) = \sum V_{ij} g_{ij}$, where V_{ij} and g_{ij} are the volume and value assigned to each hyper-rectangle/terminal-node)

Sampling from a Boosted Decision Tree

- Sampling from a Boosted Decision tree is straightforward/efficient:
  - Randomly choose a tree from the series with probability proportional to its integral
  - Randomly choose a terminal node on the tree with probability proportional to its integral
  - Uniformly sample within the hyper-rectangle of the chosen terminal node
- Critical limitation: Any transformation breaks the above logic (ie cannot efficiently sample from $f(\bar{x})$ if $g(\bar{x}) = \sum g_i(\bar{x}) \approx \ln f(\bar{x})$)
- Critical limitation: Only works for **positive-definite** tree values

Constructing the Boosted Decision Tree

- Positive-definite limitation means that slow convergence is required (later trees cannot correct with negative values)
- Train two BDT's in parallel, one for sampling, and one to aid convergence (with transformation $h(\bar{x}) \sim \ln f(\bar{x})$ and no positive-definite constraint)
- BDTs for MC integration constructed iteratively (start with uniform sampling distribution in first iteration with primary and secondary BDT's initialized to a common small value)
  - Sample N events from current secondary (sampling) BDT series $g(\bar{x})$
  - Train tree for primary BDT $h(\bar{x})$
  - Train tree for secondary BDT $g(\bar{x})$
  - Repeat

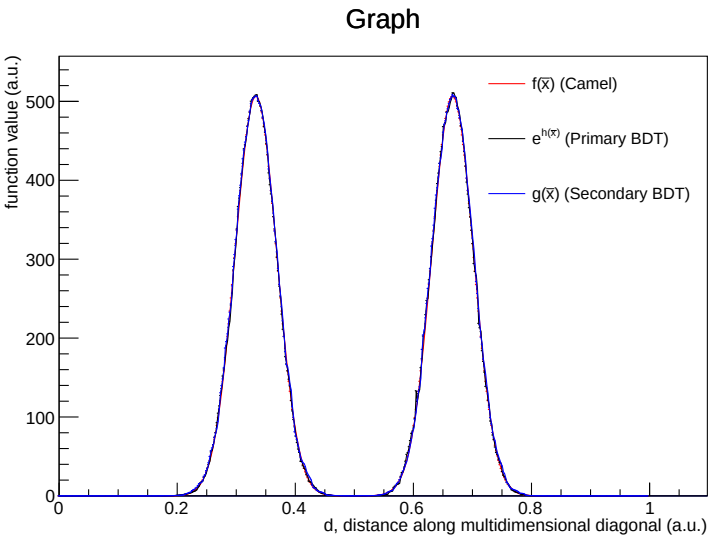
Some results - 4D Camel Function Integration

- Comparing Vegas,Foam, GBRIntegrator for 4-dimensional camel function (since this appears in both VEGAS and Foam papers).
- Given relative weight variance $\sigma_w / < w >$ after training/grid building, relative uncertainty on integral evaluated with N additional events is $\sigma_I / I = \frac{1}{\sqrt{N}} \sigma_w / < w >$

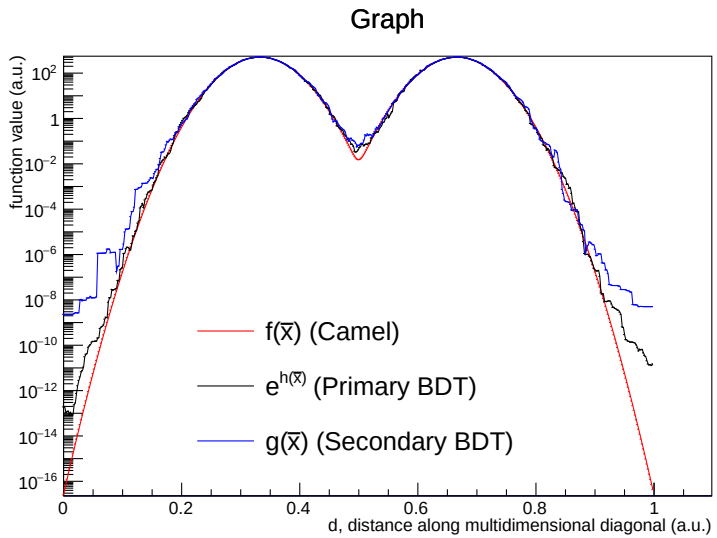
Algorithm	# of Func. Evals	$\sigma_w / < w >$	σ_I / I (2e6 add. evts)
VEGAS	300,000	2.820	$\pm 2.0 \times 10^{-3}$
Foam	3,855,289	0.319	$\pm 2.3 \times 10^{-4}$
GBRIntegrator	300,000	0.082	$\pm 5.8 \times 10^{-5}$
GBRIntegrator (staged)	300,000	0.077	$\pm 5.4 \times 10^{-5}$

- 3x smaller weight variance to foam with 10x less function evaluations
- Substantially improved performance with respect to initial version of GBRIntegrator algorithm (lacking primary/secondary BDT paradigm)
- For this particular function VEGAS performance saturates at relatively poor weight variance

Diagnostic Plots - 4D Camel Function



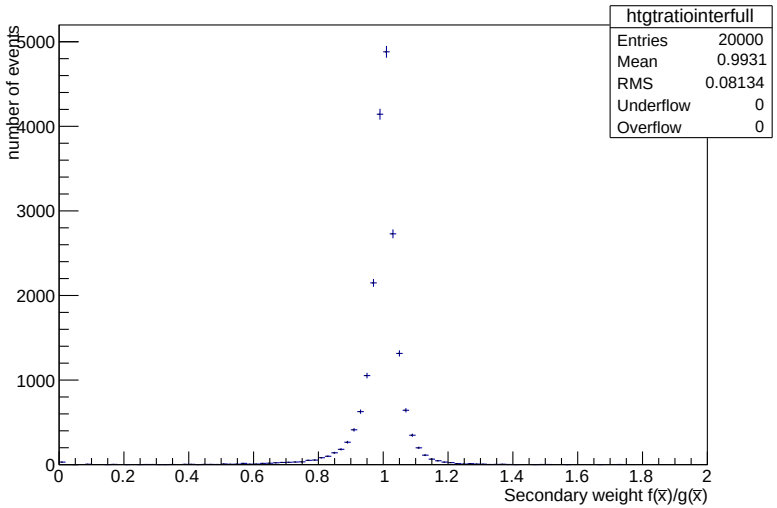
(a) linear



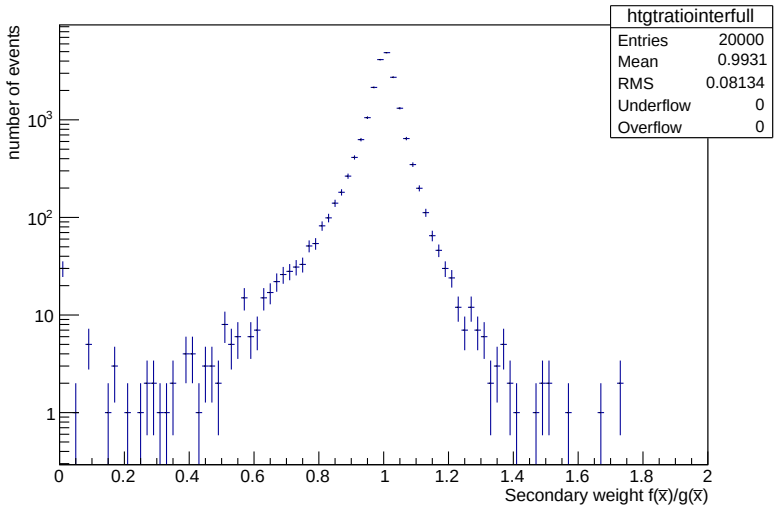
(b) log

- Secondary sampling BDT approximates function slightly worse in very low probability regions (related to initialization values, positive definite constraint during training, and lack of transformation). For this particular case, effect is small. (but this is the reason staged variation achieves slightly better precision)

Diagnostic Plots - 4D Camel Function - Integration Weights



(a) linear

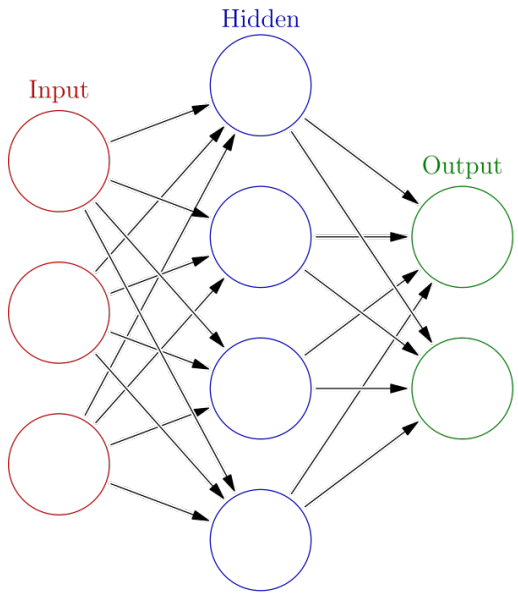


(b) log

- Excellent weight distribution for integration purposes (symmetric, small variance)

Artificial Neural Networks

- Inspired by biology, artificial neural networks comprise one or more layers of artificial neurons with **weight**, **bias**, and **activation function** with many possible architectures for how the neurons/layers are connected
- Already the simple "densely connected" neural network with non-linear activation functions can serve as a universal function approximator in a similar manner to BDT's
- Such neural networks can be trained for classification or regression problems with the appropriate loss function
- Training = finding optimal values for weights and biases to minimize the loss function using some variation of Stochastic Gradient Descent

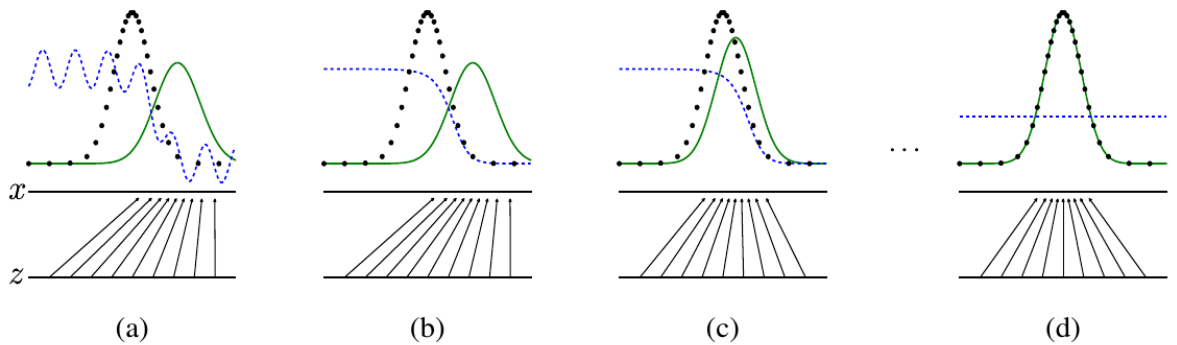


Generative Deep Neural Networks

- Significant recent work on generative deep neural networks in the data science community, with image processing/generation as a common use case
  e.g arXiv:1406.2661
- Some work in this direction in HEP as well e.g. for fast+accurate calorimeter showers (arXiv:1705.02355)
- Typical existing use cases:
  - Have a fixed set of data, or a black box generator
  - Train a generative model to produce samples following the distribution of the training data (or in high dimensional cases such as images, to produce "similar" images to those in the training set)
- Various architectures and training procedures: Variational auto-encoders, auto-regressive models, generative adversarial networks

Generative Adversarial Networks

- Generative adversarial networks train a deep neural network to generate samples starting from a known prior distribution $p(\bar{z})$ which is easy to sample from (e.g. an N-dimensional normal distribution)
- The generative network $\bar{G}$ transforms the input samples to the output space $\bar{x}$, ie $G(\bar{z}) = \bar{x}$
- A discriminator network D (e.g. a standard DNN classifier) is trained to distinguish the generated samples from the training samples
- Training proceeds iteratively such that the D is trained to maximally discriminate and G is trained to minimize the discrimination power of D until the generated samples follow the $\sim$ same distribution as the training set (MINIMAX problem/saddle point, difficult to train)



arXiv:1406.2661

Direct Probability Density Sampling

- For Monte Carlo integration or unweighting, target probability density is known (up to a normalizing constant), but initially samples are not available and cannot be easily generated
- For any given state of the generative network G, and in the special case that the input space $\bar{z}$ and output space $\bar{x}$ have the same dimensionality d, the generating probability density $g(\bar{x})$ can be determined from the sampling prior $p(\bar{z})$ and the jacobian determinant according to

$$p(\bar{z}) = g(\bar{x}) \left|\left| \frac{\partial \bar{G}(\bar{z})}{\partial \bar{z}} \right|\right| \tag{2}$$

- If the function to be integrated $f(\bar{x})$ has a probability density $p_f(\bar{x}) = f(\bar{x})/I_f$, the the KL divergence wrt the generating pdf can be written as

$$D_{KL} = \int g(\bar{x}) \ln \frac{g(\bar{x})}{p_f(\bar{x})} d\bar{x} \tag{3}$$

Direct Probability Density Sampling

- This KL divergence can be approximated numerically from a finite data set sampled from the prior $p(\bar{z})$

$$D_{KL} = \sum_{p(\bar{z})} \left[\ln p(\bar{z}) - \ln \left| \left| \frac{\partial \bar{G}(\bar{z})}{\partial \bar{z}} \right| \right| - \ln f(\bar{x}) \right] + NI_f \qquad (4)$$

  where NI_f is a constant and can be neglected (such that we can proceed without needing to know the integral of f)
- If G is a deep neural network with d inputs and d outputs and suitably continuous activation functions, the above can be used directly as a differentiable loss function in SGD **provided that $f(\bar{x})$ is easily computed and differentiable**
- n.b. the determinant is normally computed from a non-differentiable matrix decomposition, but the derivative can be evaluated from Jacobi's formula according to

$$\frac{\partial}{\partial t} \ln ||A|| = tr \left(A^{-1} \frac{\partial}{\partial t} A \right) \qquad (5)$$

Direct Probability Density Sampling with black box function

- What if the target function $f(\bar{x})$ and/or its derivatives are difficult or expensive to evaluate?
- Solution: Introduce a function approximator $r(\bar{x}) = e^{F(\bar{x})}$ where $F(\bar{x})$ can be e.g. a standard DNN regression together with a (weakly) iterative procedure
  1. Sample from a uniform distribution over the desired integration range for $f(\bar{x})$
  2. Train $F(\bar{x})$ according to loss function $L = \sum (\ln f(\bar{x}) - F(\bar{x}))^2$
  3. Train G according to modified loss function $D_{KL} = \sum_{p(\bar{z})} \left[\ln p(\bar{z}) - \ln \left| \left| \frac{\partial \bar{G}(\bar{z})}{\partial \bar{z}} \right| \right| - F(\bar{x}) \right]$
  4. Replace training set for F with new samples from G
  5. Iterate until convergence
- n.b. this a much more weakly iterative procedure than e.g. GAN's, since there is no saddle point, and F and G can be safely trained to completion at each iteration (no strong equilibrium requirement)

Sampling From Trained Network

- Sampling from trained network is straightforward
- Just draw additional samples from prior $p(\bar{z})$ and run them through generative network G
- Generating pdf value for each generated sample can be computed from $p(\bar{z})$ and Jacobian determinant as during training, and used directly for computing integration weights or for accept-reject sampling with respect to target function $f(\bar{x})$
- (Some modification to the loss functions still needed for optimal unweighting)

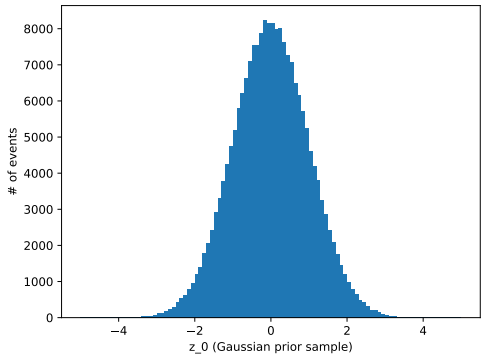
Implementation Details

- Training and generation implemented with Keras+Tensorflow (with numpy for prior sampling and input/output) including custom log determinant tensorflow operation
- Generative model and regression for function approximation implemented as densely connected neural nets with 5 hidden layers of different sizes
- Generative Model:
  - Intermediate layers use a modified tanh activation $(0.7 * tanh(x) + 0.3 * x)$ in order to maintain support over full integration range
  - Output layer uses a sigmoid activation to restrict output to integration range [0,1] (can be trivially shifted/scaled for alternate integration range)
- Regression Model:
  - Intermediate layers use elu activation
  - Output layer uses linear activation (regressing $\ln f(\bar{x})$ so no restriction on output range

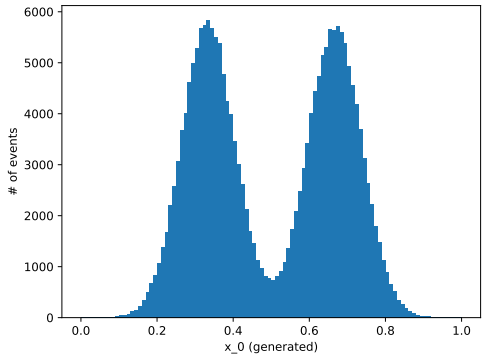
General Considerations on Computational Requirements

- Evaluating models is essentially trivial computationally
- Training models is less trivial, but should be small amount of computation time compared to ME evaluations
- Training and inference can both run on CPU or GPU (but log determinant and matrix inverse operations are only implemented for CPU currently)
- Tensorflow has efficient multi-threading on a single node

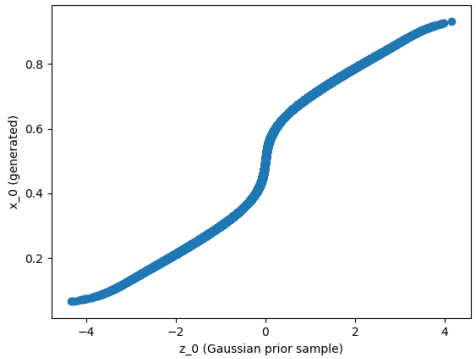
1D Example



(a) Prior



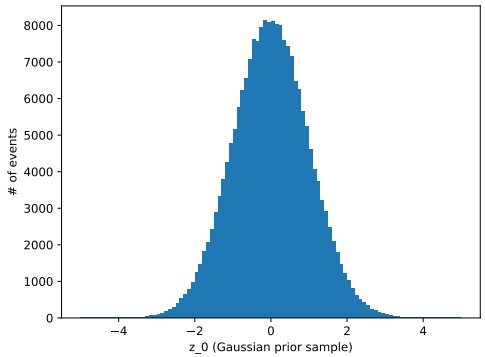
(b) Generated



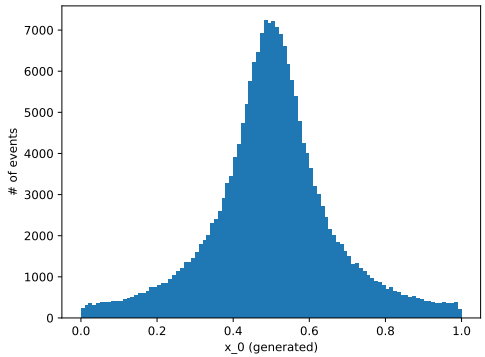
(c) Generated vs Prior

- In 1D the generative network is essential just learning the inverse CDF of the target distribution (numerically)
- Technically the function is $x = \text{CDF}_{p_f}^{-1}(\text{CDF}_p(z))$

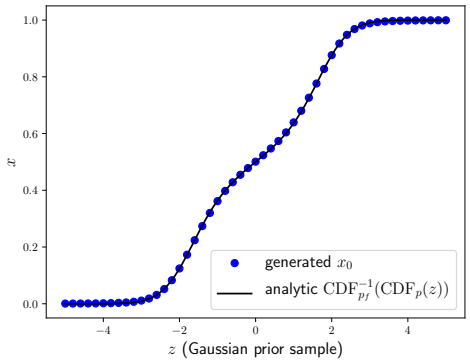
1D Example with Analytic Solution



(a) Prior



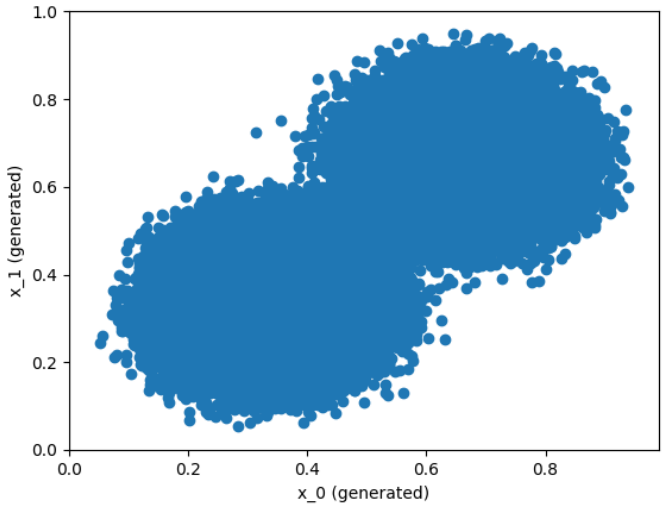
(b) Generated



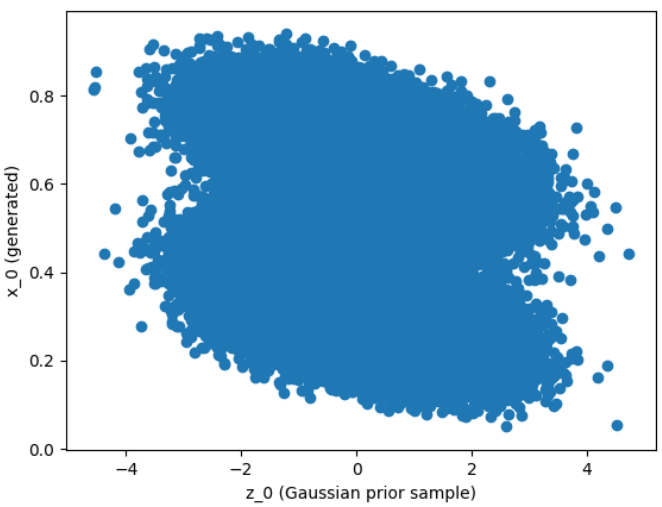
(c) Generated vs Prior

- In 1D the generative network is essential just learning the inverse CDF of the target distribution (numerically)
- Technically the function is $x = \mathrm{CDF}_{p_f}^{-1}(\mathrm{CDF}_p(z))$
- For Cauchy distribution in this example, this can be computed analytically and compared to the trained DNN result

4D Example



(a) Generated (2D Slice)



(b) Generated vs Prior (1D pair)

- The multidimensional case can be considered a generalization of inverse CDF sampling
- This model has 17,220 free parameters

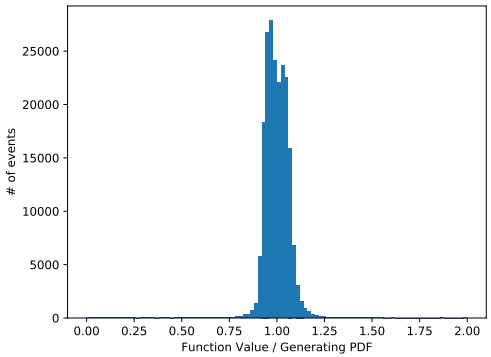
Some results - 4D Camel Function Integration

- Comparing Vegas,Foam, GBRIntegrator, Generative DNN for 4-dimensional camel function (since this appears in both VEGAS and Foam papers).
- Given relative weight variance $\sigma_w / < w >$ after training/grid building, relative uncertainty on integral evaluated with N additional events is $\sigma_I / I = \frac{1}{\sqrt{N}} \sigma_w / < w >$

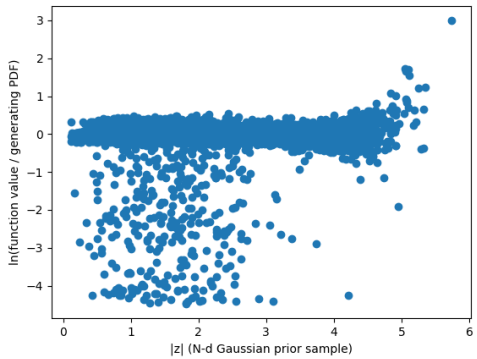
Algorithm	# of Func. Evals	$\sigma_w / < w >$	σ_I / I (2e6 add. evts)
VEGAS	300,000	2.820	$\pm 2.0 \times 10^{-3}$
Foam	3,855,289	0.319	$\pm 2.3 \times 10^{-4}$
GBRIntegrator	300,000	0.082	$\pm 5.8 \times 10^{-5}$
GBRIntegrator (staged)	300,000	0.077	$\pm 5.4 \times 10^{-5}$
Generative DNN	294,912	0.083	$\pm 5.9 \times 10^{-5}$
Generative DNN (staged)	294,912	0.030	$\pm 2.1 \times 10^{-5}$

- 3x smaller weight variance to foam with 10x less function evaluations
- Substantially improved performance with respect to initial version of GBRIntegrator algorithm (lacking primary/secondary BDT paradigm)
- Generative DNN comparable to generative BDT (but Generative DNN + DNN Regression does even better)
- For this particular function VEGAS performance saturates at relatively poor weight variance

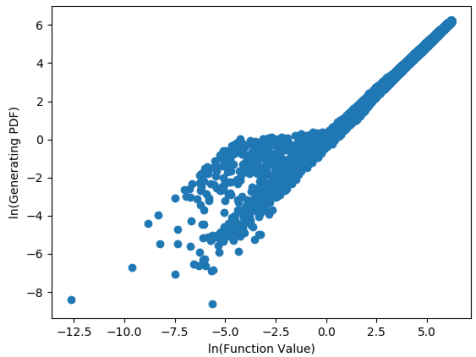
Some Diagnostic Plots - 4D Generative DNN



(a) Integration Weight



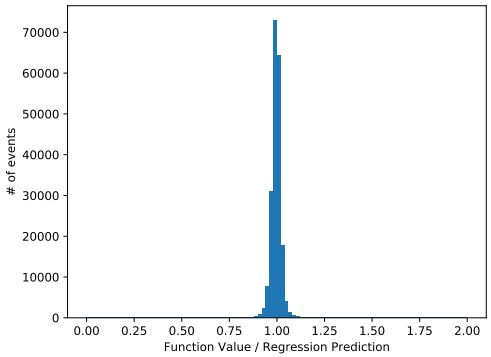
(b) ln W vs $|z|$



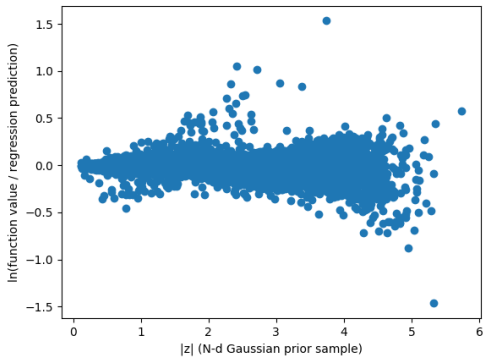
(c) Gen PDF $g(\bar{x})$ vs $f(\bar{x})$

- Reasonably good behaviour
- Some tails in weight distribution in the tails of the prior distribution (biasing the sampling of the prior is straightforward if needed)
- Generating pdf $g(\bar{x})$ tracks the target function $f(\bar{x})$ down to low values, and then sometimes overshoots (not a big issue for either integration or un-weighting)

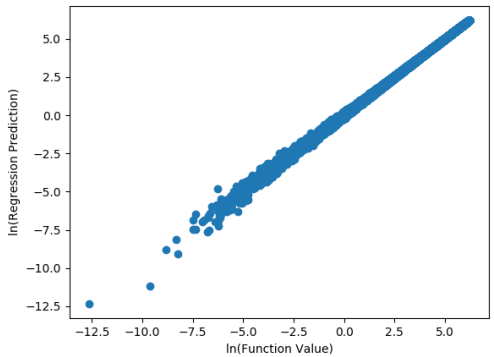
Some Diagnostic Plots - 4D DNN Regression



(a) Integration Weight



(b) $\ln W$ vs $|z|$



(c) Gen PDF $g(\bar{x})$ vs $f(\bar{x})$

- DNN regression better behaved than generative model (easier to train)
- Could use secondary unweighting to sample from DNN regression with reasonable efficiency if desired
- This regression model has 4,417 free parameters

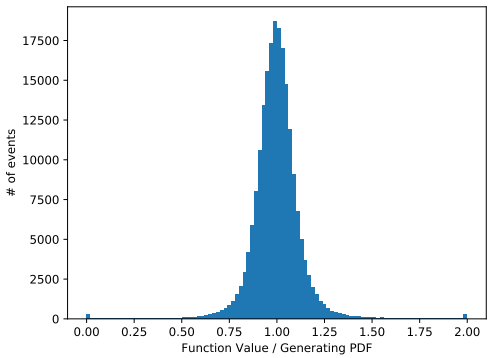
Some results - 9D Camel Function Integration

- Comparing Vegas, GBRIntegrator, Generative DNN for 9-dimensional camel function

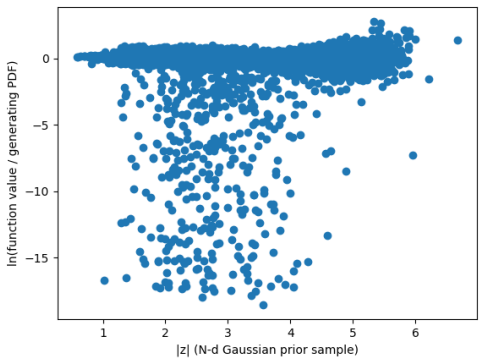
Algorithm	# of Func. Evals	$\sigma_w / < w >$	σ_I / I (2e6 add. evts)
VEGAS	1,500,000	19	$\pm 1.3 \times 10^{-2}$
GBRIntegrator	3,200,000	0.63	$\pm 4.5 \times 10^{-4}$
GBRIntegrator (staged)	3,200,000	0.31	$\pm 2.2 \times 10^{-4}$
Generative DNN	294,912	0.15	$\pm 1.1 \times 10^{-4}$
Generative DNN (staged)	294,912	0.081	$\pm 5.7 \times 10^{-5}$

- 50x smaller weight variance to Vegas with 2x function evaluations
- Larger performance difference between staged and non-staged variations in this case
- DNN approach scales much better with dimensionality ($>$ 100x smaller weight variance than Vegas with 5x **fewer** function evaluations
- For this particular function VEGAS performance saturates at relatively poor weight variance

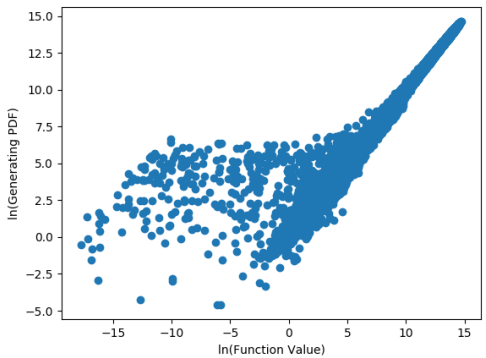
Some Diagnostic Plots - 9D Generative DNN



(a) Integration Weight



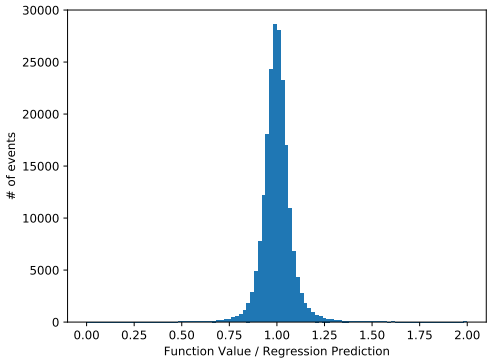
(b) $\ln W$ vs $|z|$



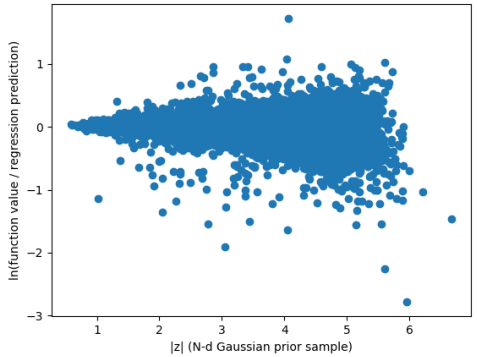
(c) Gen PDF $g(\bar{x})$ vs $f(\bar{x})$

- Qualitatively similar behaviour to $4D$ case
- This generative model has 17,865 free parameters

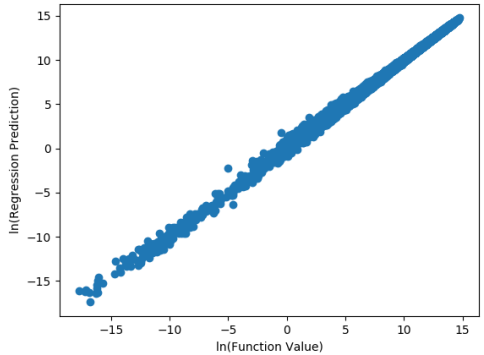
Some Diagnostic Plots - 9D DNN Regression



(a) Integration Weight



(b) $\ln W$ vs $|z|$



(c) Gen PDF $g(\bar{x})$ vs $f(\bar{x})$

- Qualitatively similar behaviour to $4D$ case
- Excellent behaviour over $\sim$ 13 orders of magnitude
- This regression model has 4577 free parameters

Invertibility of Generative DNN Model

- Interesting limitation: Probability density for generative DNN model can **not** be evaluated for an arbitrary phase space point $\bar{x}$, since one needs to know the corresponding point in the prior space $\bar{z}$, and the model is not trivially invertible
- Not a problem for integration or unweighting where all the phase space points are anyways generated by sampling from the prior
- Nevertheless exploring the possibility/requirements to analytically invert such a model, since this might be convenient for diagnostic purposes, and would enable multi-channeling-like extensions

Further Improvements to Generative DNN's

- A number of tunable parameters in terms of number of iterations vs number of phase space points per iteration, re-use of phase space points from early iterations, size and number of layers, optimizer/convergence parameters, etc
- Some potential to explore more sophisticated architectures than simple densely connected networks (autoregressive models, recurrent networks, etc)
- Requirements for narrow integration weight distribution and/or tight upper bound for unweighting are somewhat different than typical machine learning applications in the literature ("generate natural looking images", etc)

BDT vs DNN

- Both approaches are able to encode and efficiently sample from multi-dimensional distributions with non-trivial correlations between dimensions
- Underlying sampling method is entirely different in the two cases (FOAM-based vs inverse-CDF-like)
- For the purpose of integration and unweighting, the generative BDT has quite strict limitations on positive-definite weights/linear mapping to output and a lack of flexibility for the loss function which makes minimization difficult and enforces very slow convergence for good performance
- Generative DNN models are more flexible in this respect and are therefore expected to have better scaling with the number of parameters and dimensionality (already observed for test cases) as well as more room for improvement
- Software infrastructure for training large DNN's is also more widely supported by data scientists and computing industry
- Plan to pursue the DNN-based algorithm and stop work on the BDT's

Conclusions/Todo

- Very promising performance/potential for speedup of MC integration (and eventually generation) with both BDT and DNN-based algorithms
- Further work will focus on DNN's
- Todo:
  - More systematic tests at higher dimensions
  - Tests with real physics examples (integration with Madgraph already in progress)
  - Further optimization or improvements to network architecture and optimization/convergence
  - Implementation for more efficient unweighting (modified loss functions)
  - Understand if/how to best combine with multi-channelling and related techniques (requires a detailed study on the **invertibility** of these generative DNN models)

Conclusions/Todo

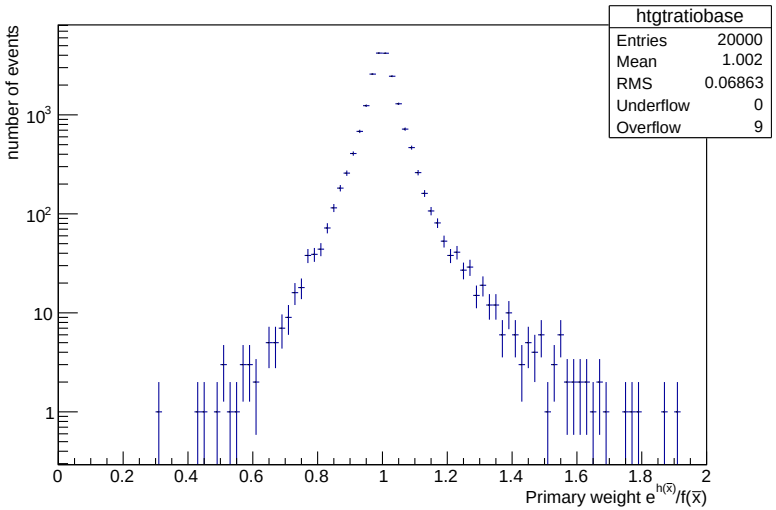
- **Important to keep in mind:** Since the ML models are used for importance sampling and accept-reject sampling, accuracy of integration and proper distribution of generated events does **not** depend on the accuracy of the Machine Learning model, less accurate model only means more samples needed to reach a given integration precision and/or lower unweighting efficiency (just like VEGAS integration grids and similar)

Conclusions/Todo

- Paper and standalone implementation coming very soon (weeks)
- (after some significant scope-creep with respect to previously)
- Implementation has dependencies on Keras, Tensorflow and numpy
- Python interface is most natural, but C/C++/fortran wrappers should be possible
- Once performance and robustness are established for relevant classes of matrix elements, integrate with commonly used generators (Madgraph, Sherpa, Herwig, etc)

Backup

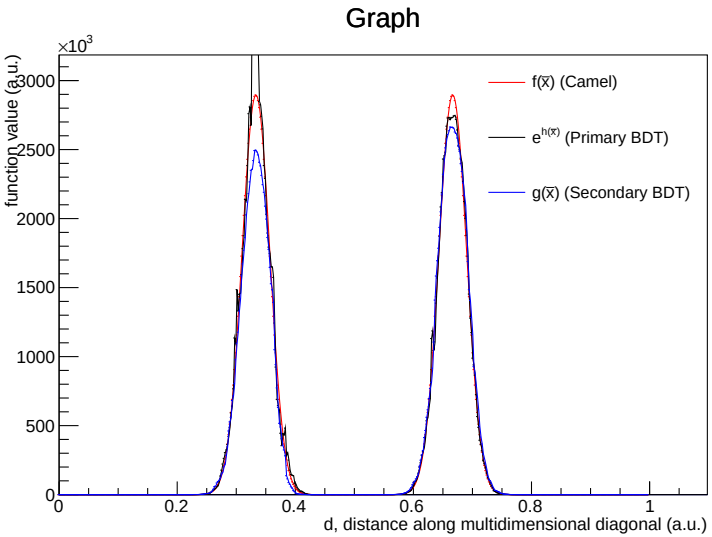
Diagnostic Plots - 4D Camel Function - Integration Weights - Staged case



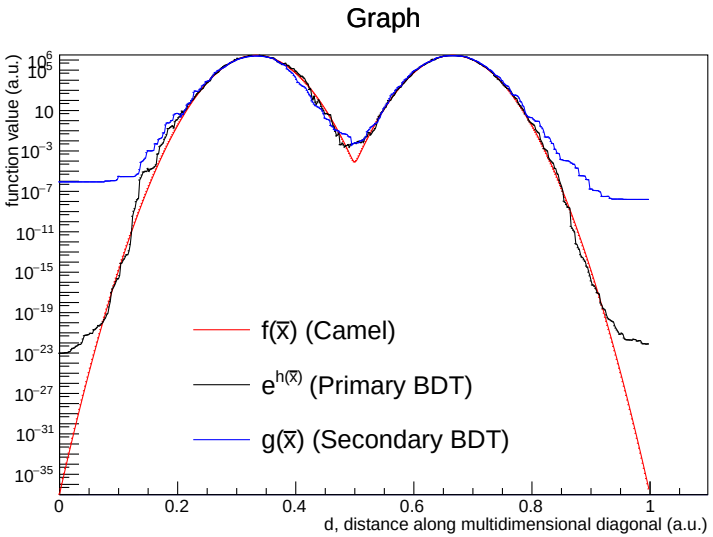
(a) Primary Weight

- Primary weight for integral evaluation, intermediate weight is for intermediate unweighting (primary vs secondary bdt)

Diagnostic Plots - 9D Camel Function with BDT



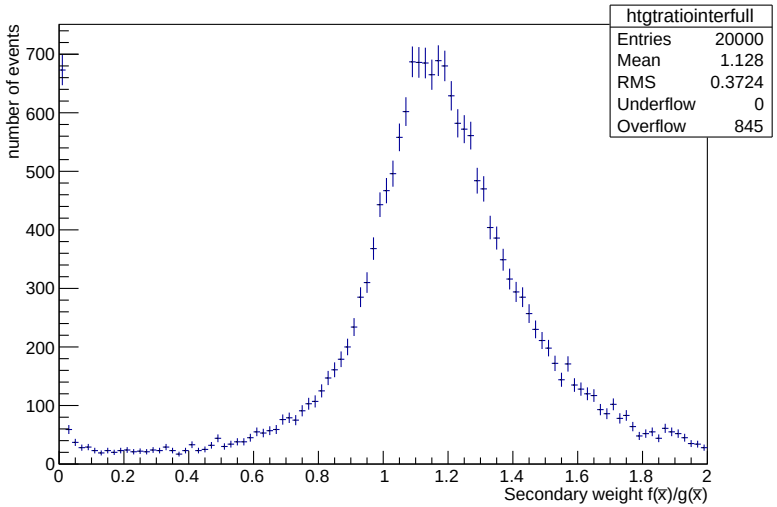
(a) linear



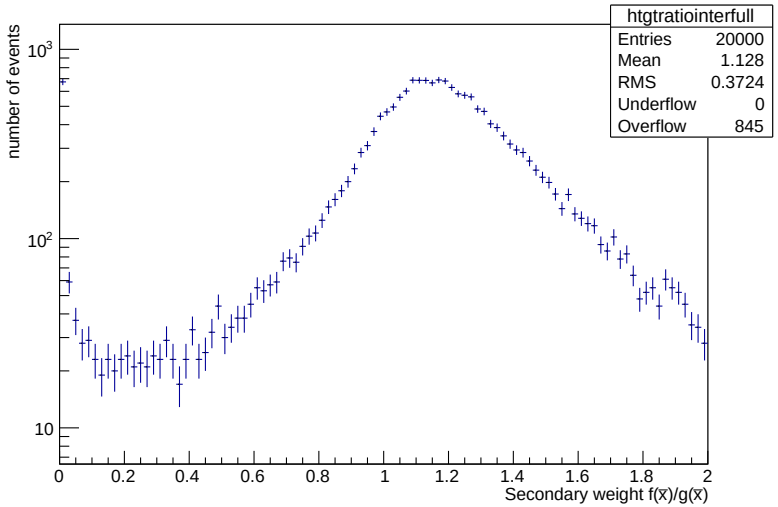
(b) log

- Deficiencies of secondary sampling BDT with respect to primary BDT are larger in higher dimensional case

Diagnostic Plots - 9D Camel Function - BDT Integration Weights



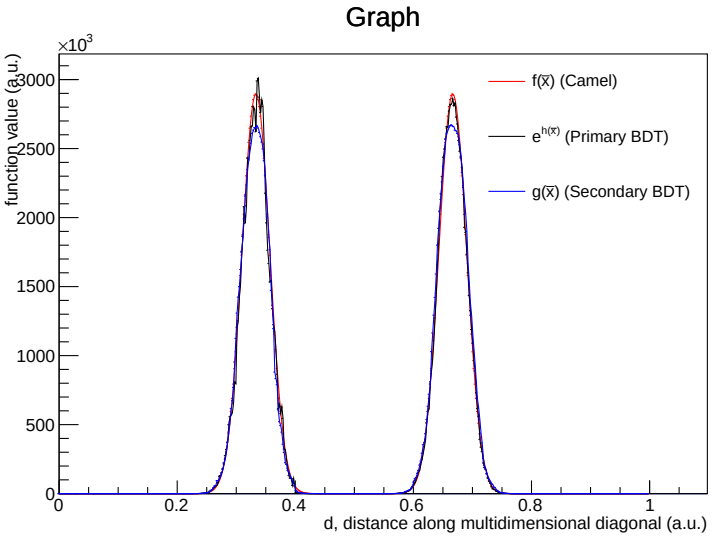
(a) linear



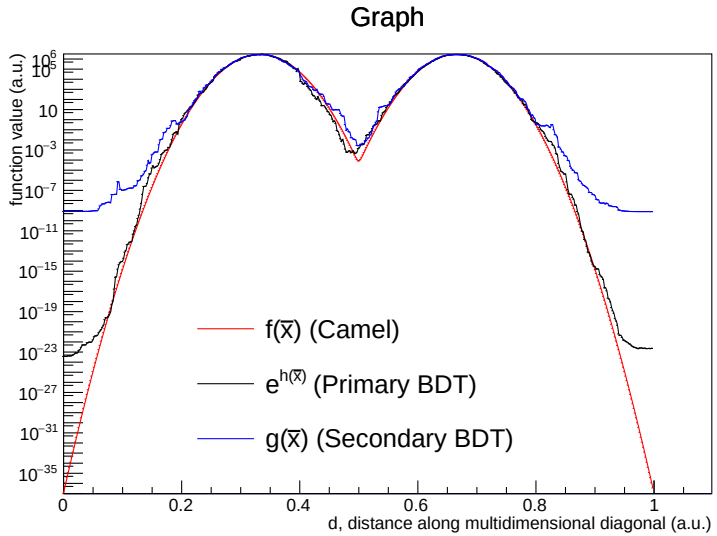
(b) log

- Visible also in the weight distribution
- (Checking staged case now)

Diagnostic Plots - 9D Camel Function - Staged BDT Case



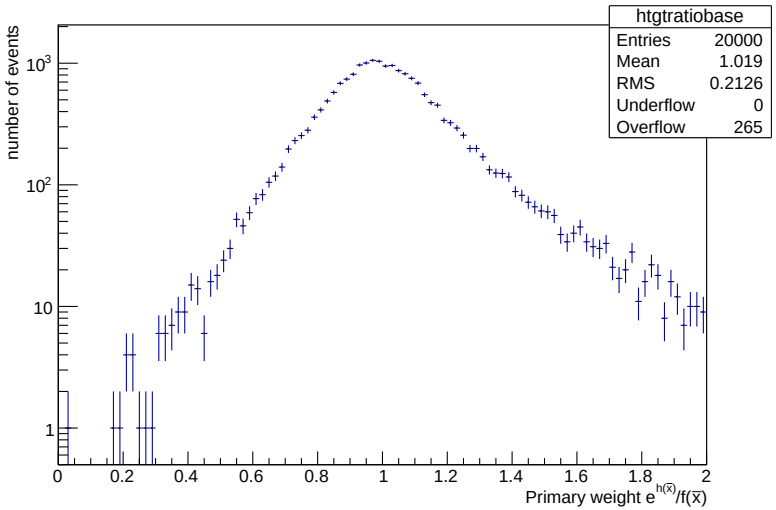
(a) linear



(b) log

- Deficiencies of secondary sampling BDT with respect to primary BDT are larger in higher dimensional case

Diagnostic Plots - 9D Camel Function - Integration Weights - Staged BDT case



(a) Primary Weight

- Staged approach corrects obvious issues with weights in 9D case

BDT Output Transformation

- If not for the sampling limitations, would prefer to use transformation such that $h(\bar{x}) = \sum h_i(\bar{x}) \approx \ln f(\bar{x})$ (then $f(\bar{x}) \approx e^{h(\bar{x})}$)
- This ensures that prediction for $f(\bar{x})$ is positive-definite, even when $h_i(\bar{x})$ are not
- This form of transformation also has mathematical conveniences for minimization

BDT Loss Function

- For variance reduction, would like to directly minimize $\sum(f(\bar{x})/g(\bar{x}) - 1)^2$ or similar, but this is not mathematically convenient
- Alternate loss function $\sum(\ln f(\bar{x}) - \ln g(\bar{x}))^2$ is equivalent to second order
- With transformed output this reduces to the simplest possible parabolic loss function $k\sum(\ln f(\bar{x}) - h(\bar{x}))^2$
- This can also be interpreted as the negative log-likelihood for a log-normal distribution in $f(\bar{x})$:

$$p(f(\bar{x})|h(\bar{x}),\sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{\frac{(\ln f(\bar{x}) - h(\bar{x}))^2}{2\sigma^2}} \text{ with } k = \frac{1}{2\sigma^2}$$

Secondary BDT

- In order to be able to easily sample from the resulting output, construct also a secondary BDT subject to sampling limitations
- **Problem:** Without any transformation, loss function $\sum(\ln f(\bar{x}) - \ln g(\bar{x}))^2$ is very difficult (has non-convex regions)
- **Problem:** Since each tree must be positive definite, results from early trees are "locked in" and cannot be compensated by later trees $\rightarrow$ must ensure slow convergence for good results
- **Solution:** Train secondary BDT to approximate primary BDT output $e^{h(\bar{x})}$ rather than $f(\bar{x})$ directly. If both sets of trees are trained in parallel then this ensures slow convergence.
- **Solution:** Use modified loss function at i_{th} iteration $L_i = k\sum(\ln(e^{h(\bar{x})} - g_{i-1}(\bar{x})) - \ln \Delta g_i(\bar{x}))^2$
- Can be interpreted as fitting the mean of a log-normal distribution with respect to the **residuals** after the previous iterations
- Requires some numerical protections (minimum value of e.g. e^{-24} for ln argument $e^{h(\bar{x})} - g_{i-1}(\bar{x})$)