Amplitudes with neural networks

Daniel Maître

IPPP, Durham

CERN, 2nd June 2023

Outline

- Matrix element emulator
  - Tree-level/one-loop
  - Pre-unweighting with emulator
- Integrating with a neural network
  - Application to sector-decomposed amplitudes

Matrix element emulation

Work with Henry Truong
arXiv:2107.06625, arXiv:2302.04005

Motivation

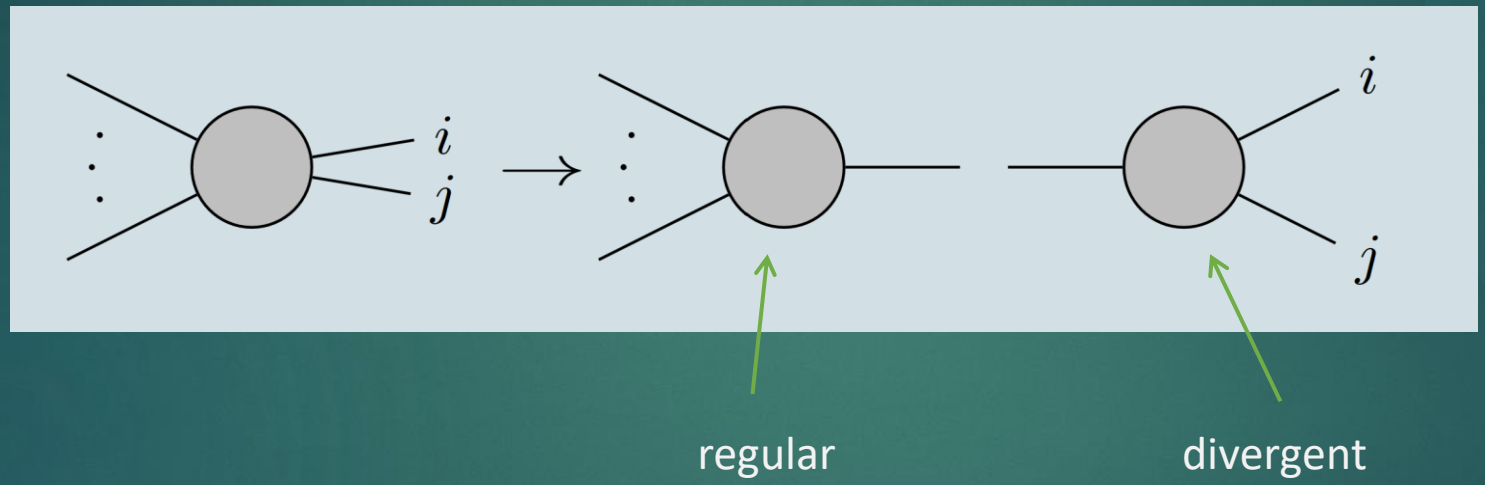
- Matrix elements can take a significant amount of CPU to calculate
  - Typically difficult to run on GPUs
  - Often just to throw away when unweighting
- If precise enough a NN emulation can help:
  - A NN model can easily evaluated on GPU, TPU, etc
  - Can be easily ported from one platform to another
  - Much quicker and parallel evaluation
  - First stage unweighting

Difficulties emulating matrix elements

- Singular behaviour in soft and collinear limits makes a straight-forward emulation difficult:
  - Small change in phase-space input results in dramatic change in ME value
  - Selection of training set/loss can be difficult:
    - The loss can be dominated by singular configurations for loose training cuts
    - The extrapolation becomes unreliable if trained on too tight cuts

Principle

- We know the behaviour of MEs in soft and collinear limits



- Use a NN to predict the regular factor and use the known analytic divergent behaviour
- Use a NLO subtraction-style ansatz to emulate the LO ME

Factorisation-aware emulation

- Write amplitude as an ansatz

$$\langle |\mathcal{M}_{n+1}|^2 \rangle = \sum_{\{ijk\}} C_{ijk} D_{ij,k}$$

- Fit the coefficients

$$L_{\text{MSE}} = \frac{1}{N} \sum_{i=1}^{N} (y_i - p(\vec{x}_i; \theta))^2$$

- Very redundant parametrisation
- Encourage NN to learn factorisation

$$L_{\text{pen}} = J \sum_i \frac{D_i^{-2}}{\sum_j D_j^{-2}} |C_i D_i|$$

$$L = L_{\text{MSE}} + L_{\text{pen}}$$

Factorisation-aware emulation

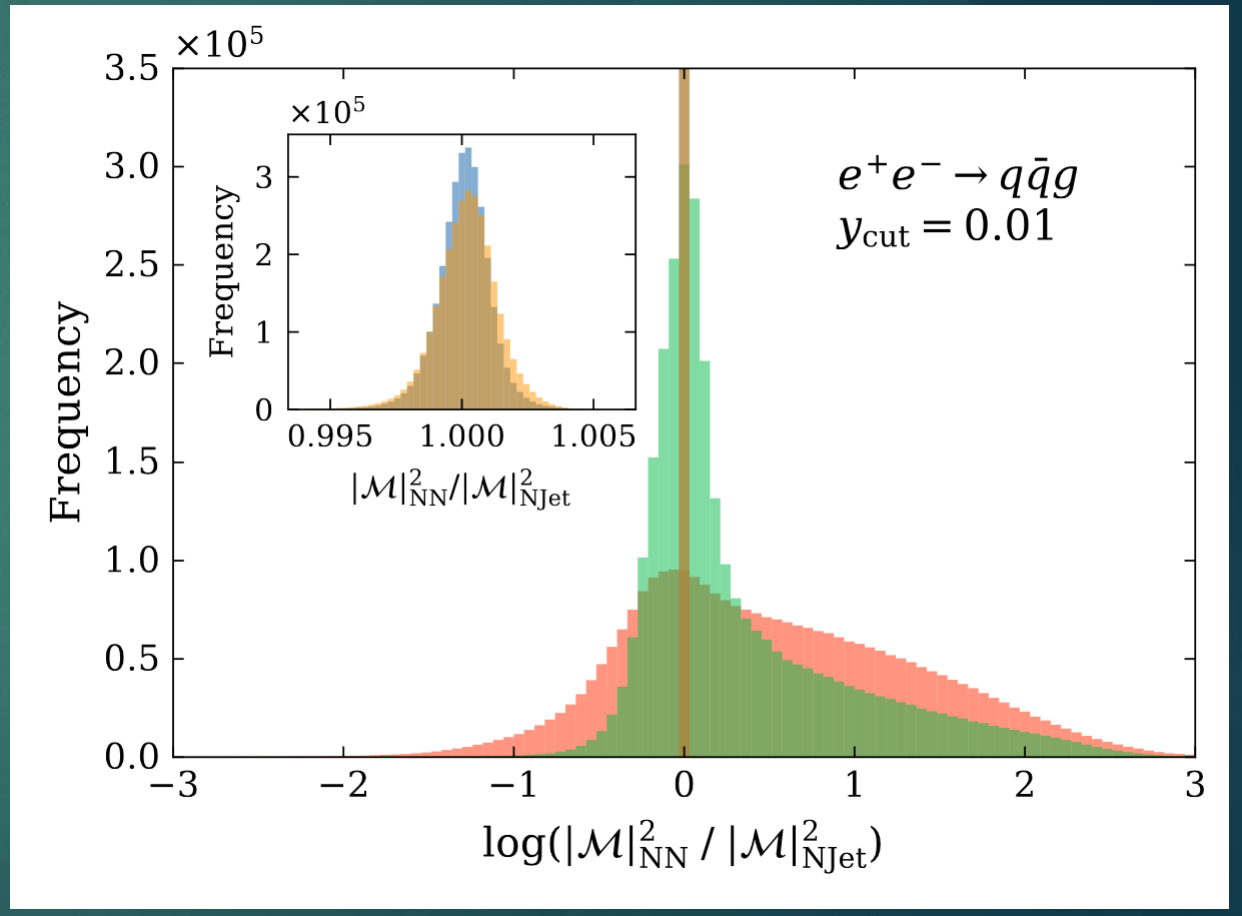
- When approaching a singular limit only the relevant dipole is relevant
- Away from all singularities all terms combine to emulate the matrix element
- At LO we use Catani-Seymour dipoles, at one-loop we used antenna functions
- Azimuthal term added:

$$S_{ij}\sin\left(2\phi_{ij}\right)+C_{ij}\cos\left(2\phi_{ij}\right)$$

- The angle φ_{ij} is the azimuthal angle of the decay particles in the plane perpendicular to the parent particle momentum

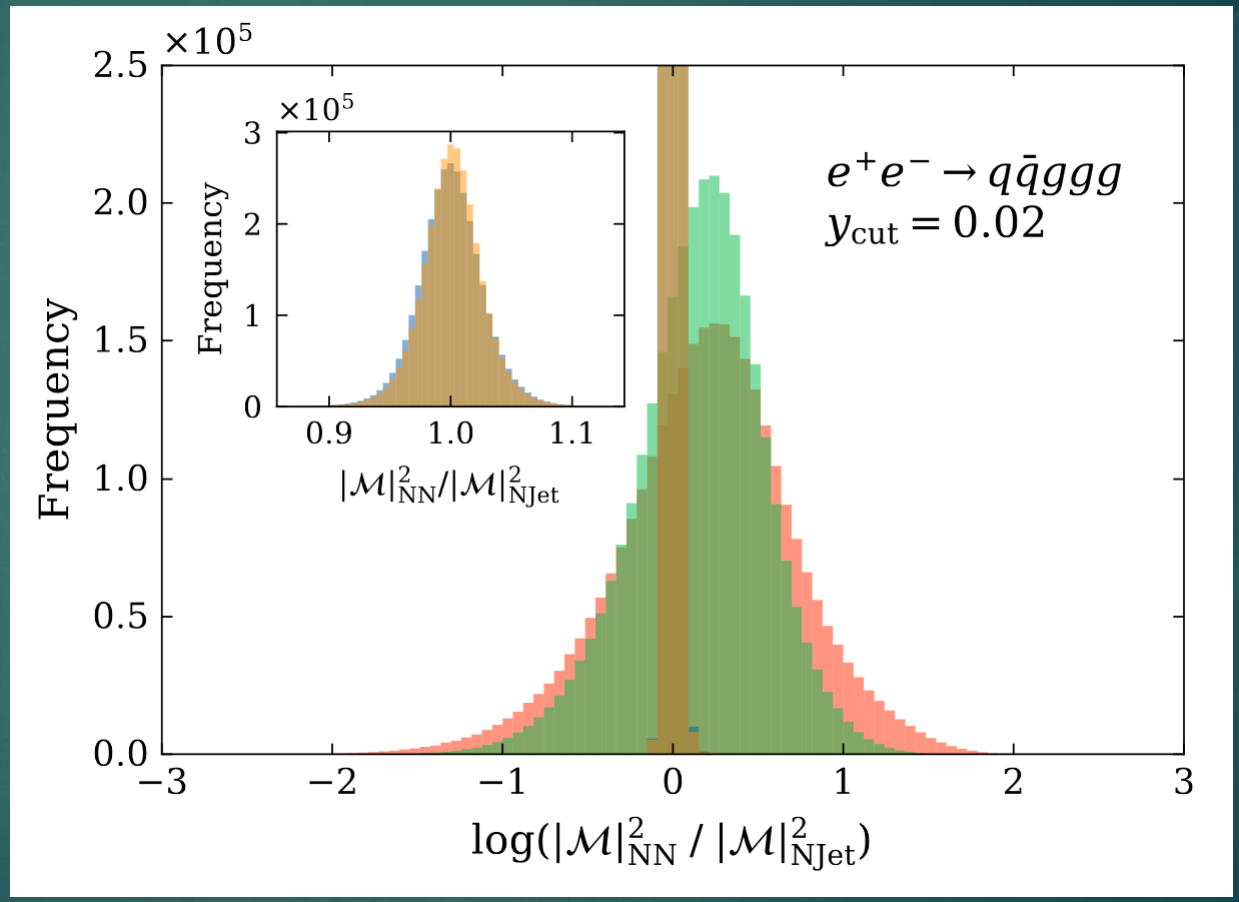
Results

- E+e- annihilation into jets
- Compare with previous attempt [arXiv 2002.07516]
- They used a single NN, or a decomposition where each NN only has to deal with one singularity
- Our results are much more accurate using the same size network and same training set



Results

- 5 jets

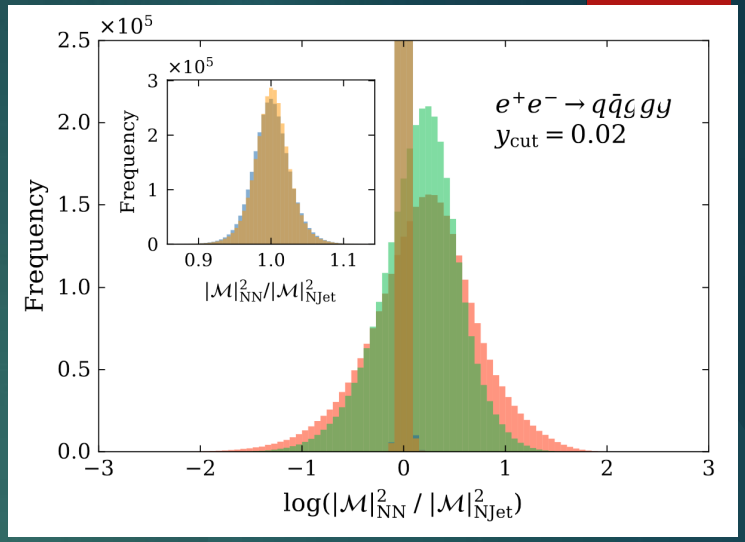


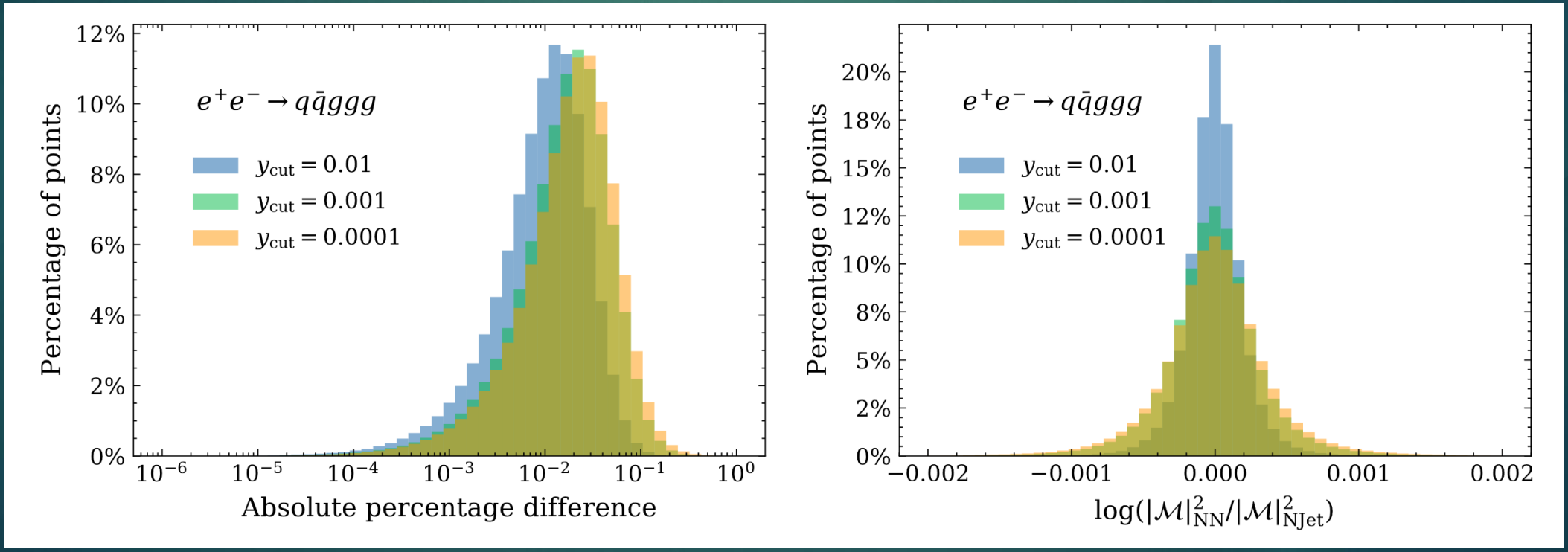
Results

- Accuracy limited by NN size and training set, not the ability of the model to emulate divergent behaviour
- Use larger NN and larger training set
- Train on 3 different training set
  - $y_{cut} = 0.01, 0.001, 0.0001$
- Lower cut means larger range for the matrix element
  - Expect lower precision

Results

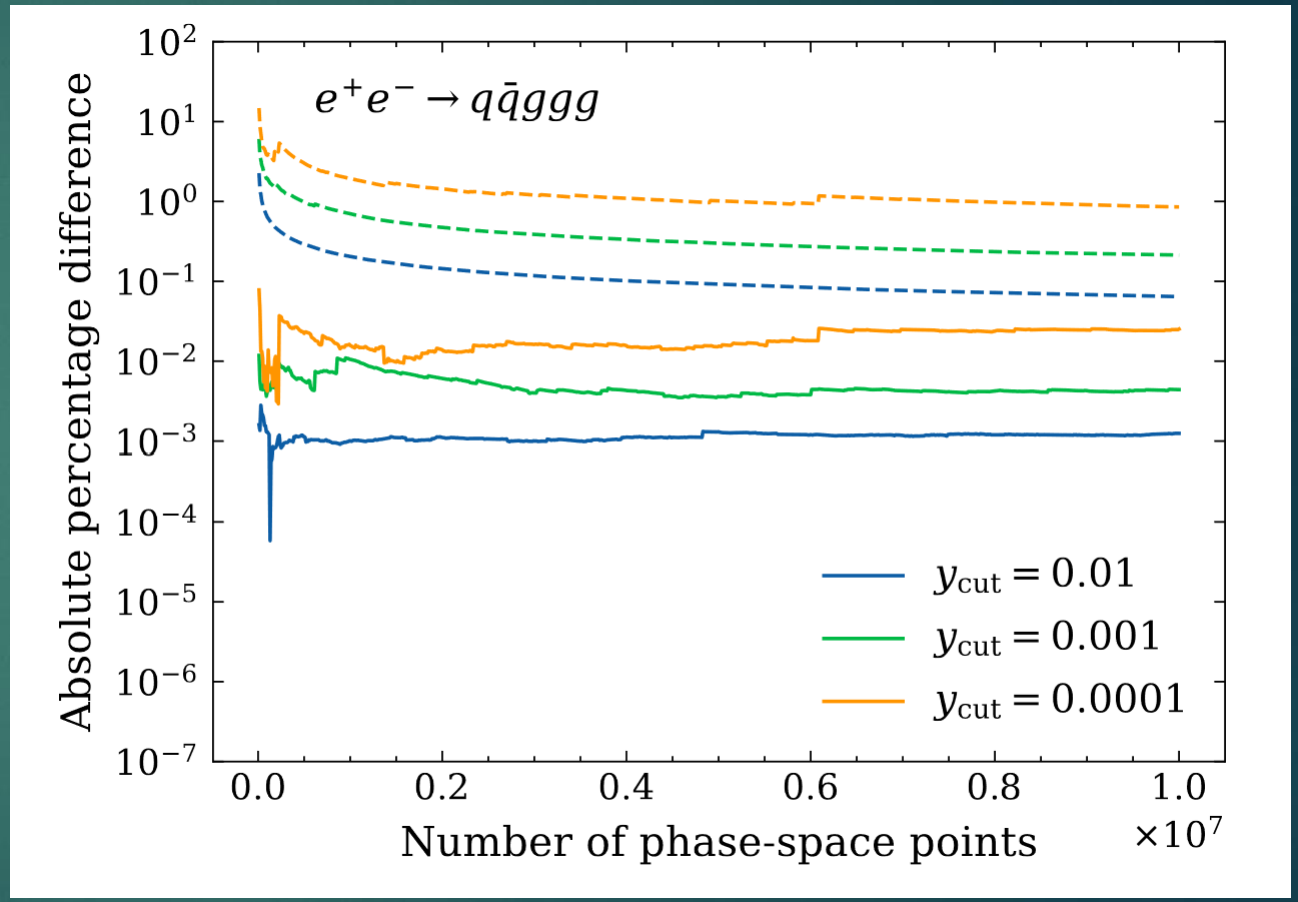
- No discernable bias
- 3-4 digits accuracy





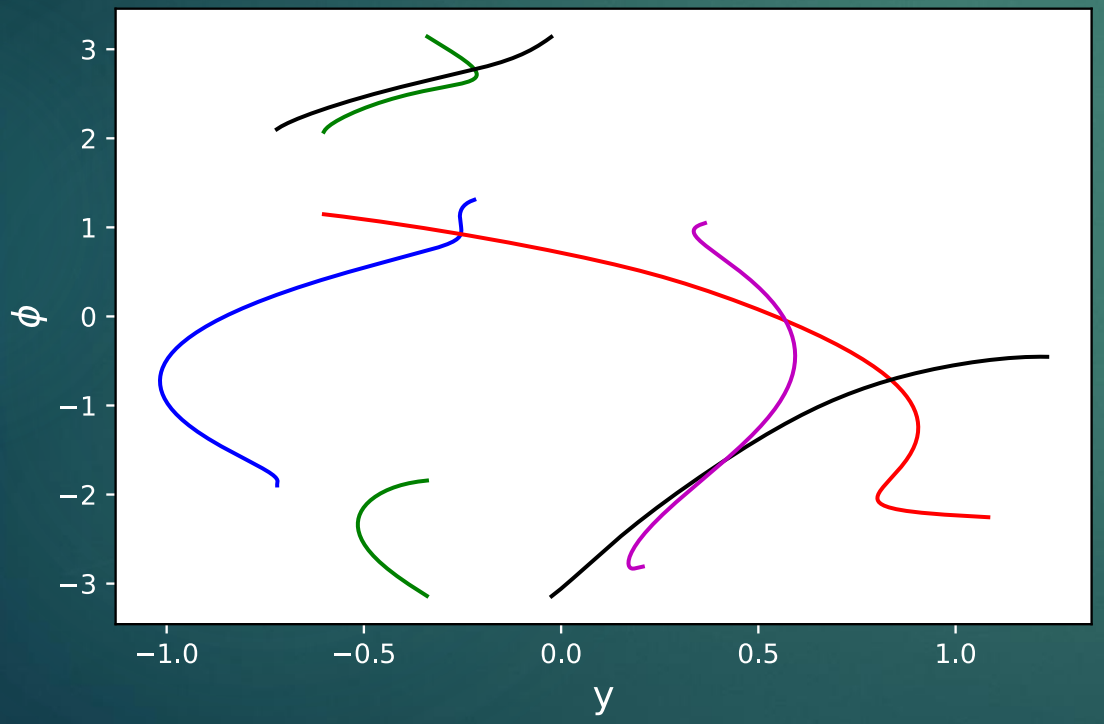
Cross section

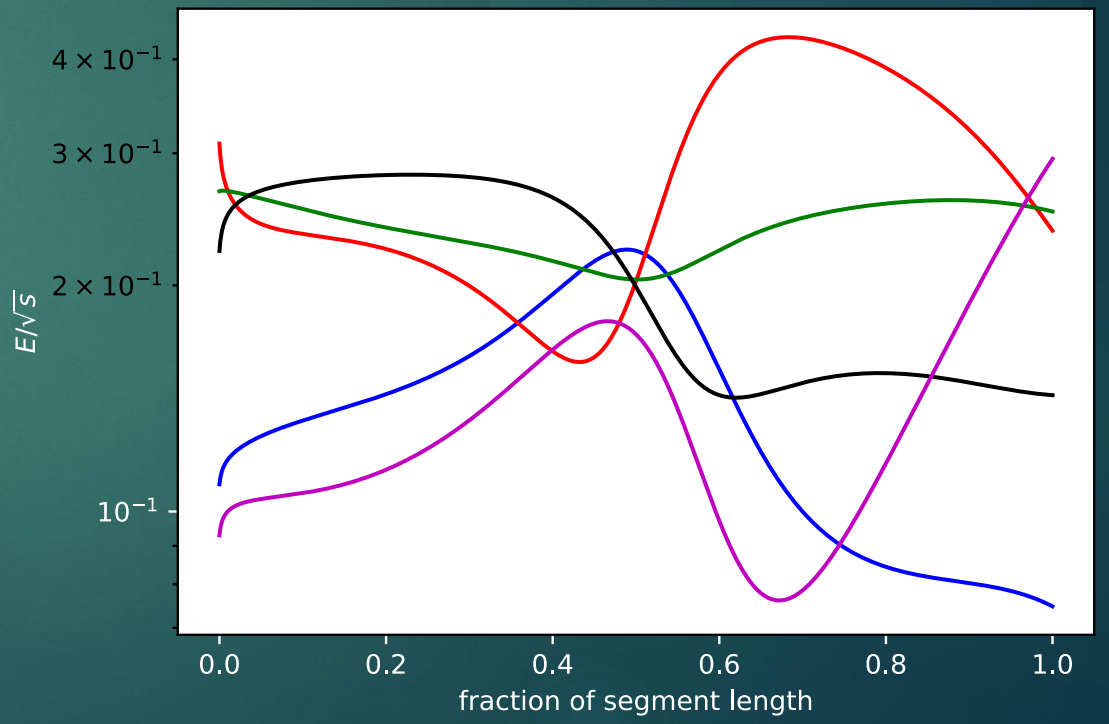
- Calculate the error on the total cross section for the test set
- The error is smaller than the statistical uncertainty
- The model can be used to augment the training set



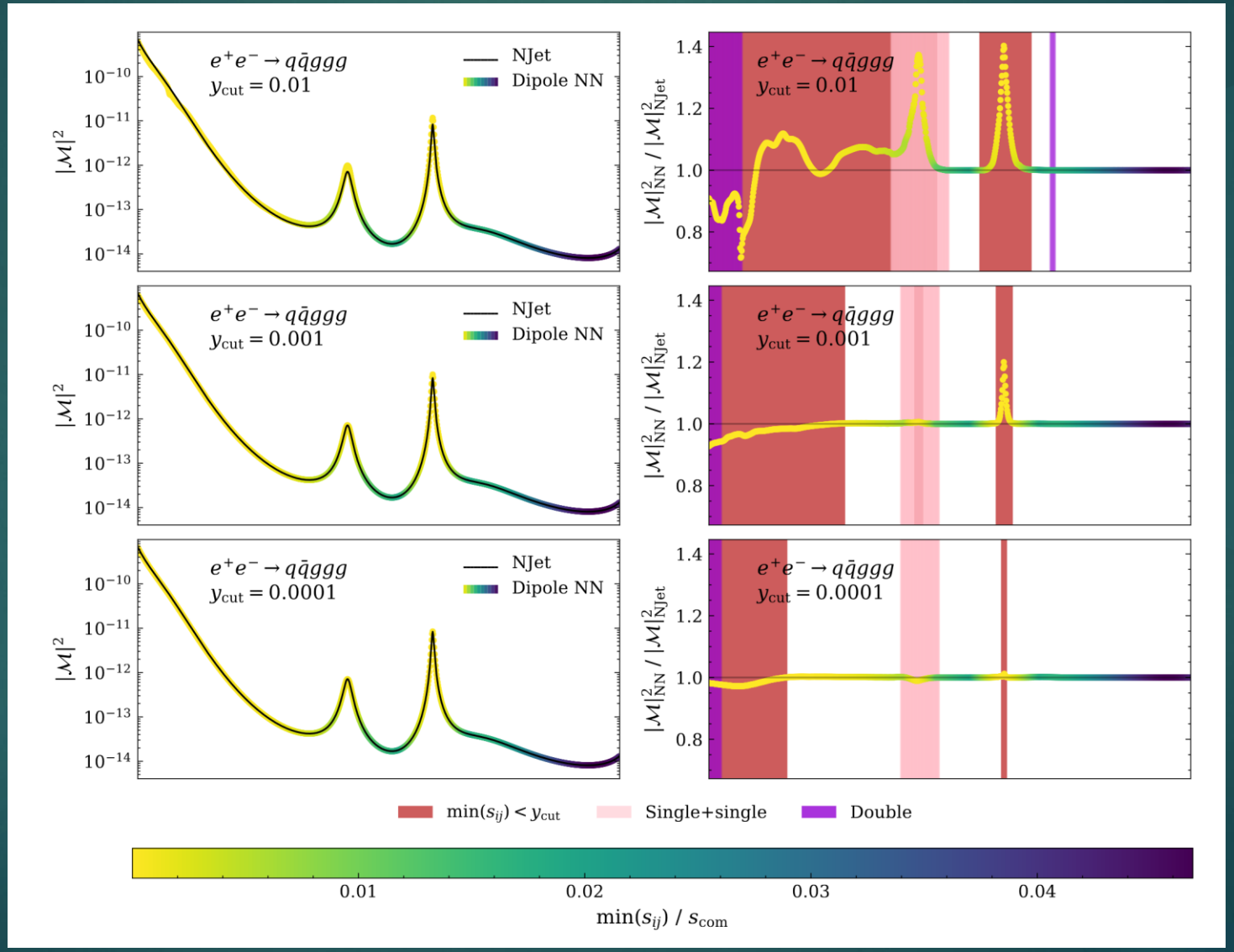
Phase-space trajectories

- RAMBO maps unit hypercube to phase-space uniformly
- Follow mapping along a segment connecting two random points





Phase-space trajectories



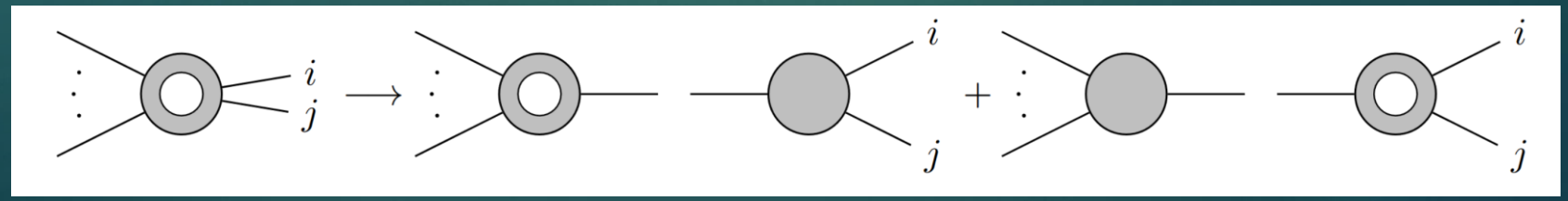
One-loop amplitudes

- Much more CPU intensive to calculate
- Choose to model the *k*-factor

$$k_n = \frac{2\Re\left\{\mathcal{M}^{(n,0)}\mathcal{M}^{(n,1)*}\right\}}{|\mathcal{M}^{(n,0)}|^2} \equiv \frac{|\mathcal{M}^{(n,1)}|^2}{|\mathcal{M}^{(n,0)}|^2}$$

- Factorisation is more complicated, we use antenna factorisation

$$|\mathcal{M}^{(n+1,1)}|^2 \longrightarrow X^0_{ijk}|\mathcal{M}^{(n,1)}|^2 + X^{1,F}_{ijk}|\mathcal{M}^{(n,0)}|^2$$



K-factor ansatz

$$k_{n+1} \longrightarrow \frac{X^0_{ijk}|\mathcal{M}^{(n,1)}|^2 + X^{1,F}_{ijk}|\mathcal{M}^{(n,0)}|^2}{X^0_{ijk}|\mathcal{M}^{(n,0)}|^2}$$

$$k_{n+1} \longrightarrow \frac{|\mathcal{M}^{(n,1)}|^2}{|\mathcal{M}^{(n,0)}|^2} + \frac{X^{1,F}_{ijk}}{X^0_{ijk}}$$

$$k_{n+1} \longrightarrow k_n + \frac{X^{1,F}_{ijk}}{X^0_{ijk}}$$

$$k_{n+1} = C_0 + \sum_{\{ijk\}} C_{ijk} \frac{X^{1,F}_{ijk}}{X^0_{ijk}}$$

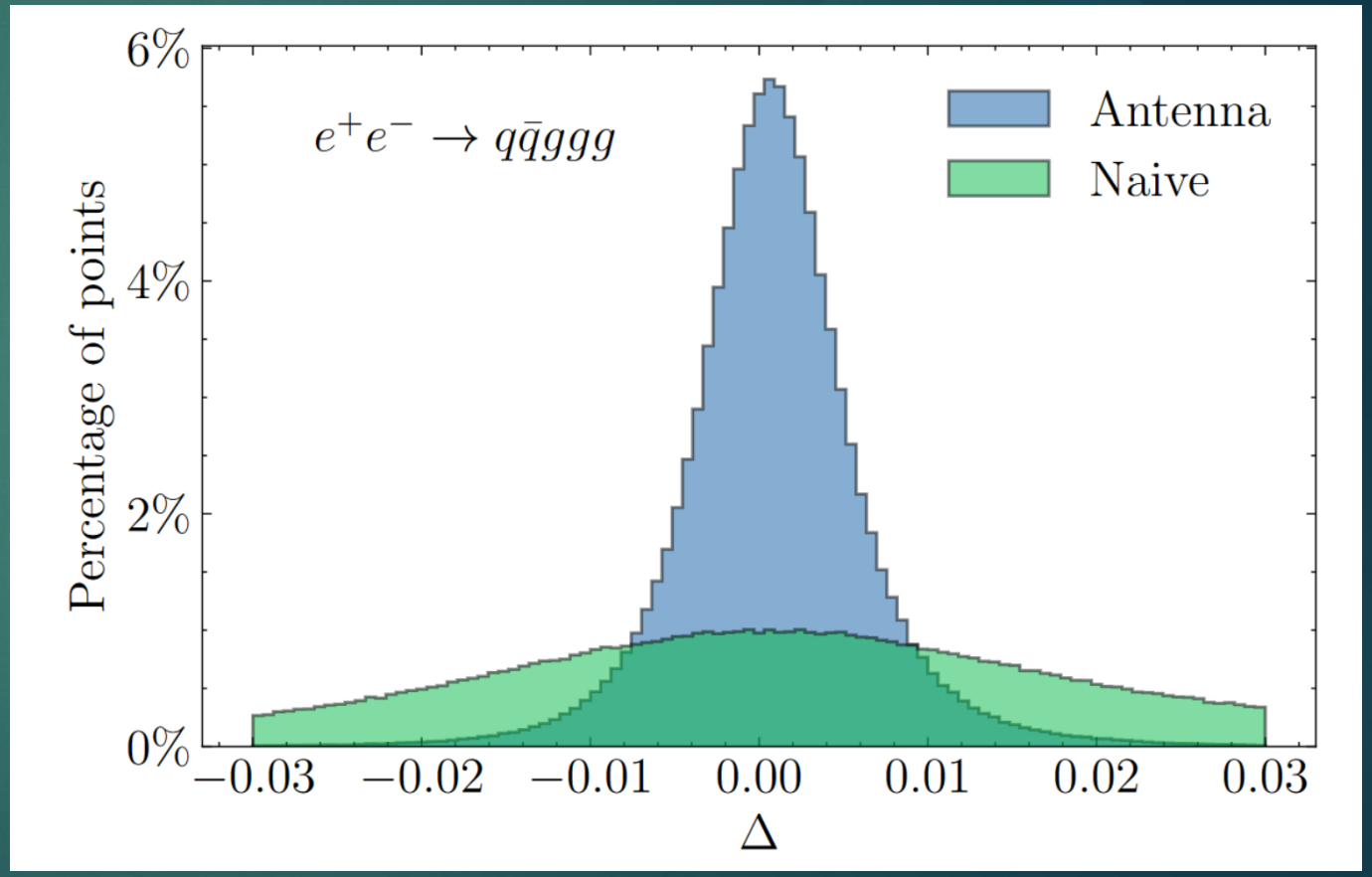
One-loop results

- Evaluate the precision using:

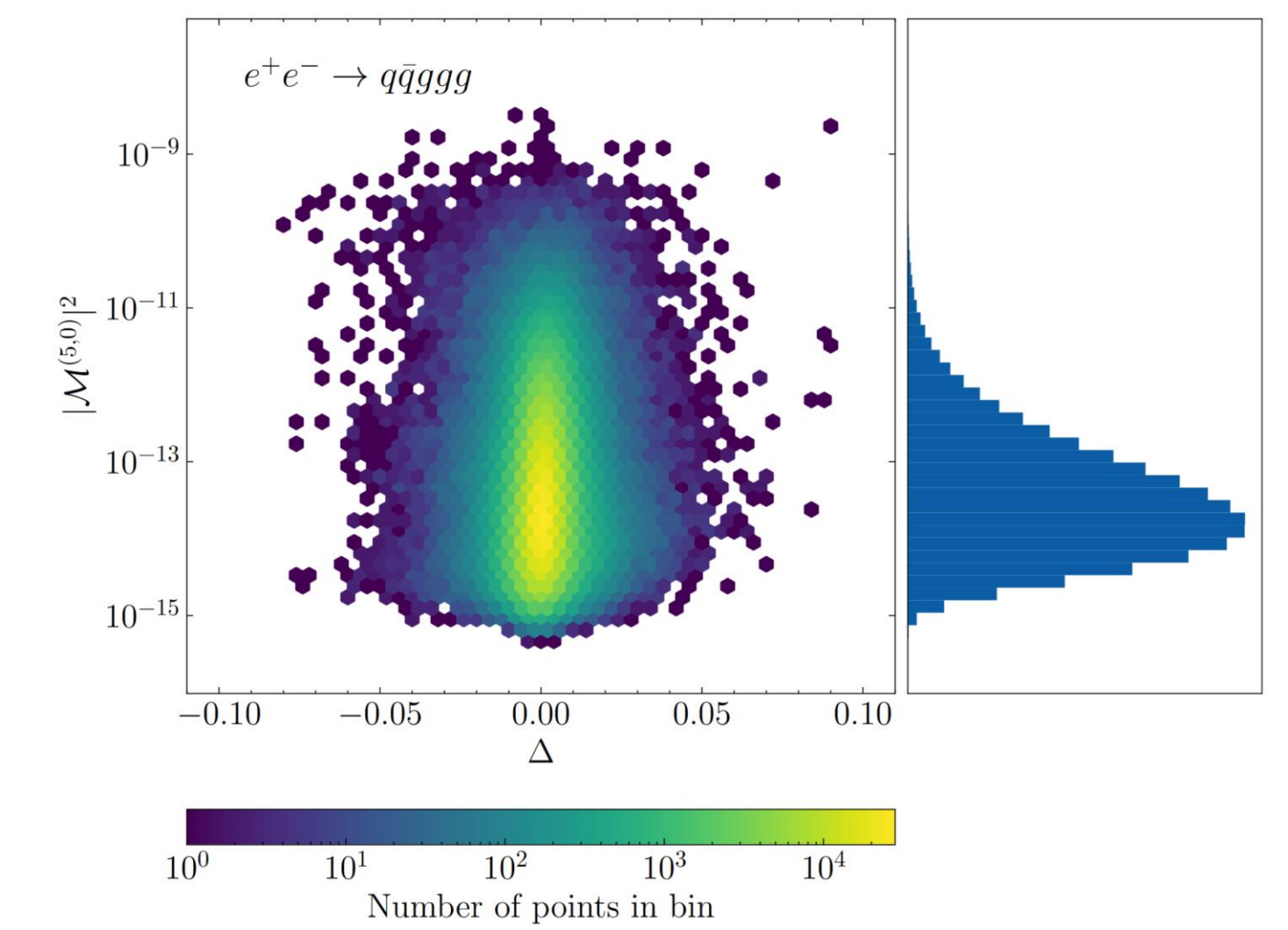
$$k_{\text{true}} - k_{\text{pred}} = \frac{|\mathcal{M}^{(n,1)}|^2_{\text{true}} - |\mathcal{M}^{(n,1)}|^2_{\text{pred}}}{|\mathcal{M}^{(n,0)}|^2_{\text{true}}} = \Delta$$

One-loop results

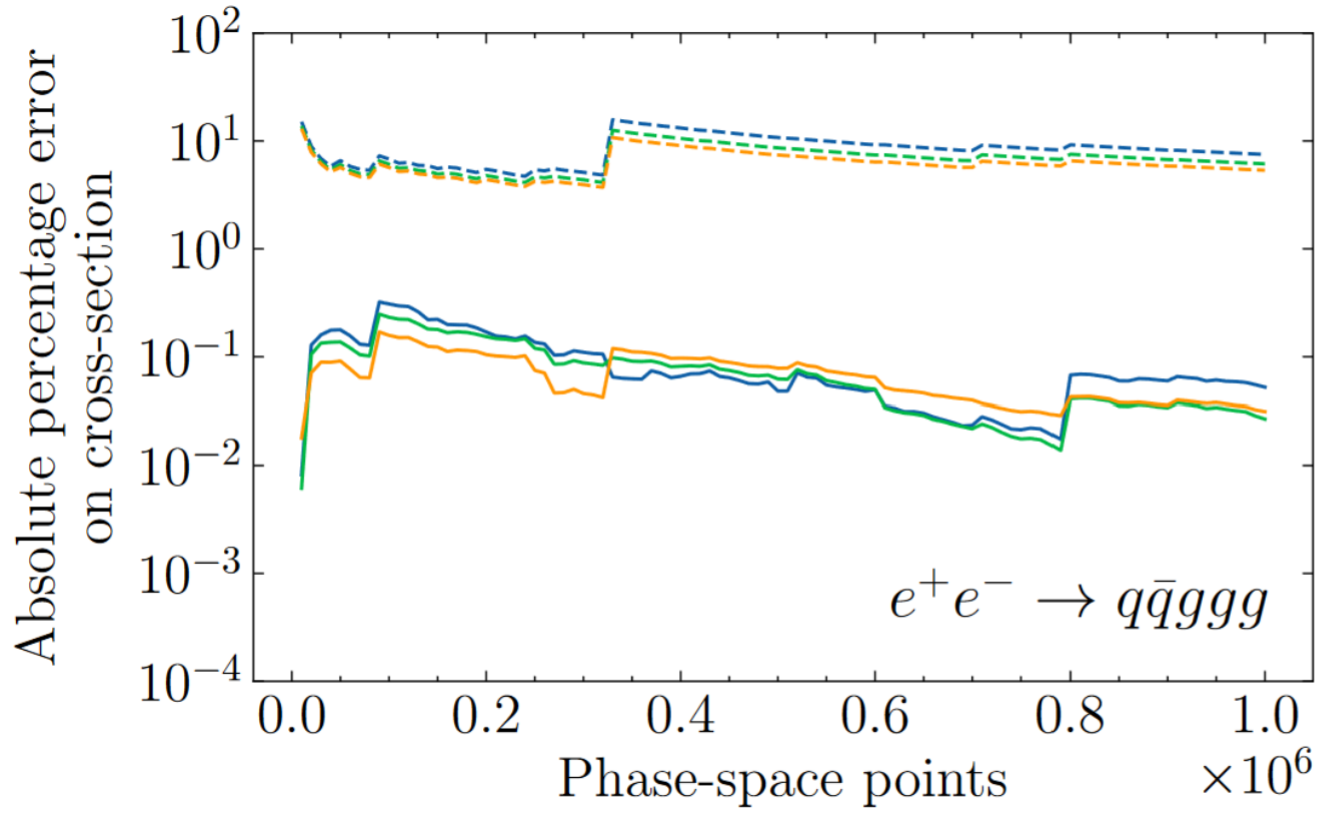
- Compare with a single NN model for the *k*-factor

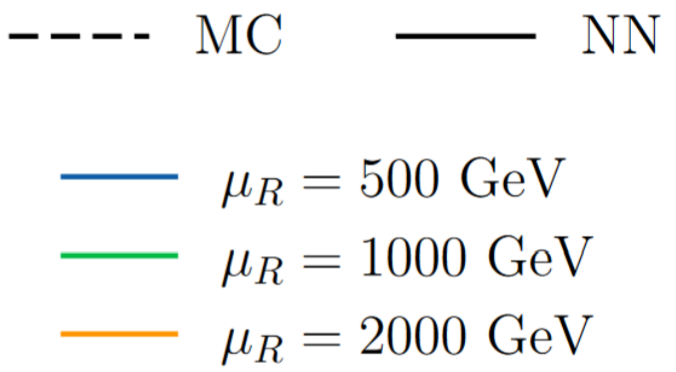


One-loop results



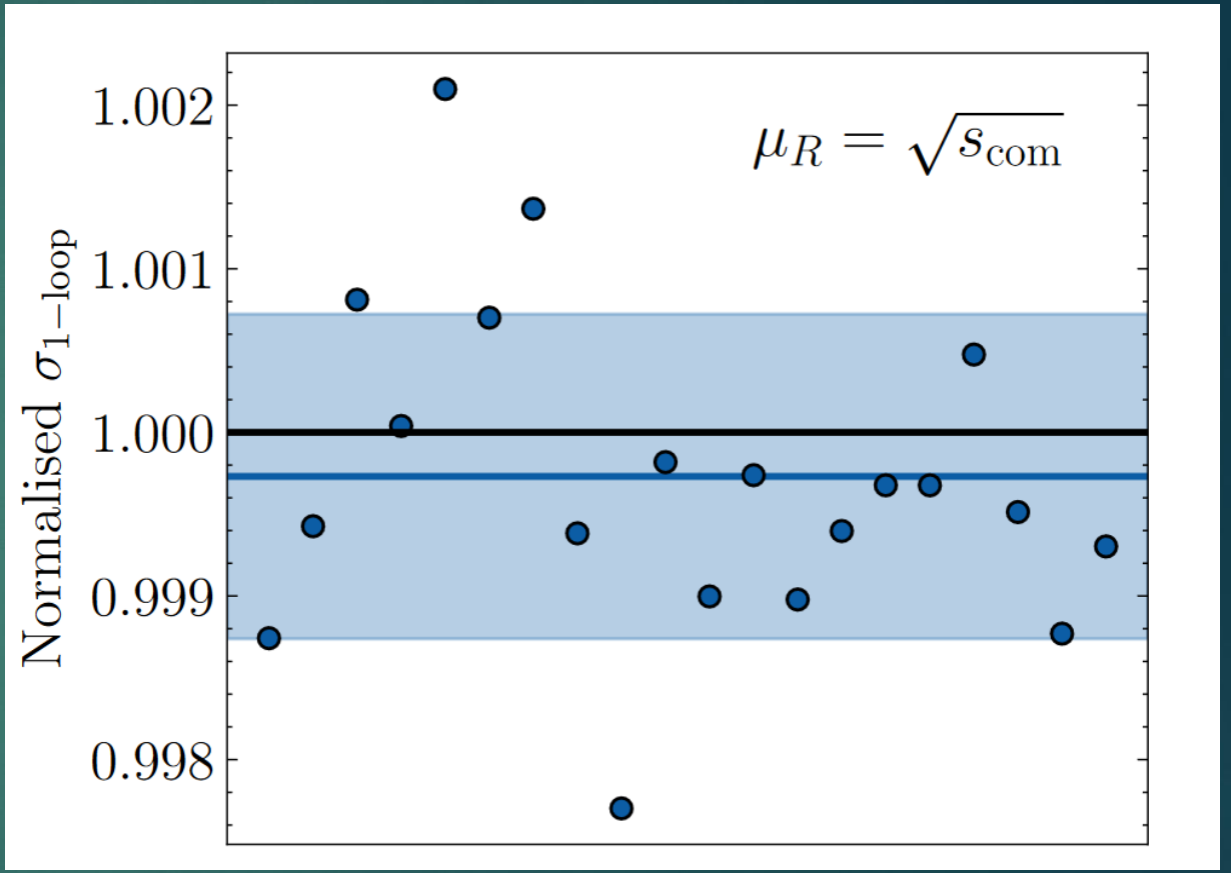
One-loop result





One-loop result

- Use 20 replicas to estimate variance and bias of model
- Statistical MC integration error is of order 10%!
- Can use NN model to augment dataset!
- Use variance as an estimate of the NN error



Unweighting with NN approximation

Work with Timo Janßen, Stefan Schumann, Frank Siegert and Henry Truong [arXiv:2301.13562]

Unweighting

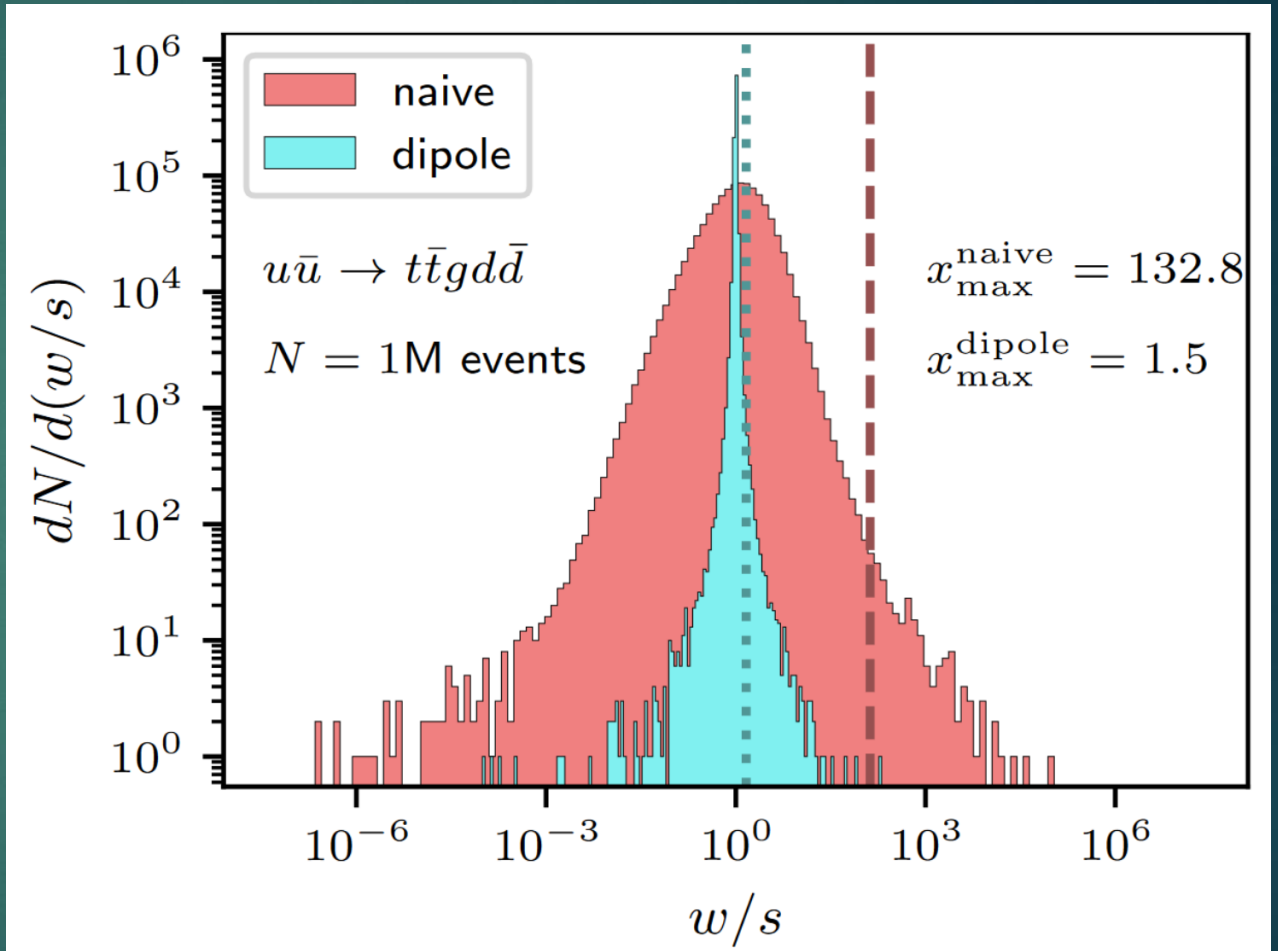
- Unweighting high multiplicity matrix elements can be extremely unefficient
- Can be improved by
  - Generate a set of unweighted values x according to an approximation $g(x)$ that is easy to unweight
  - Unweight this set according to $f(x)/g(x)$
  - If ratio is close to 1 we get to keep many more calculated ME
- Tolerate a small amount of weights above 1

Unweighting

- K. Danziger, T. Janßen, S. Schumann, F. Siegert implemented such a two-stage unweighting in Sherpa and used a NN surrogate [arXiv:2109.11964]
- Z/W +4 jets and $t\bar{t}$+3 jets
- Obtained speed up of up to 10 compared with AMEGIC
- Use our factorisation aware emulator instead of their NN surrogate
  - Needed to implement initial-state and massive dipoles

Unweighting

- First unweight w.r.t NN model s, then correct with true weight w
- Factorisation-aware NN is much more precise
- Up to Z/W +5 jets and $t\bar{t}$+4 jet
- Result in very large efficiency gains (16-350)
- Largest gains for the most complicated processes



(b) Channel $u\bar{u} \to t\bar{t}gd\bar{d}$ $(t\bar{t} + 3 \text{ jets})$.

NN for integration

Work with Roi Santos-Mateos
arXiv:2211.02834

Introduction

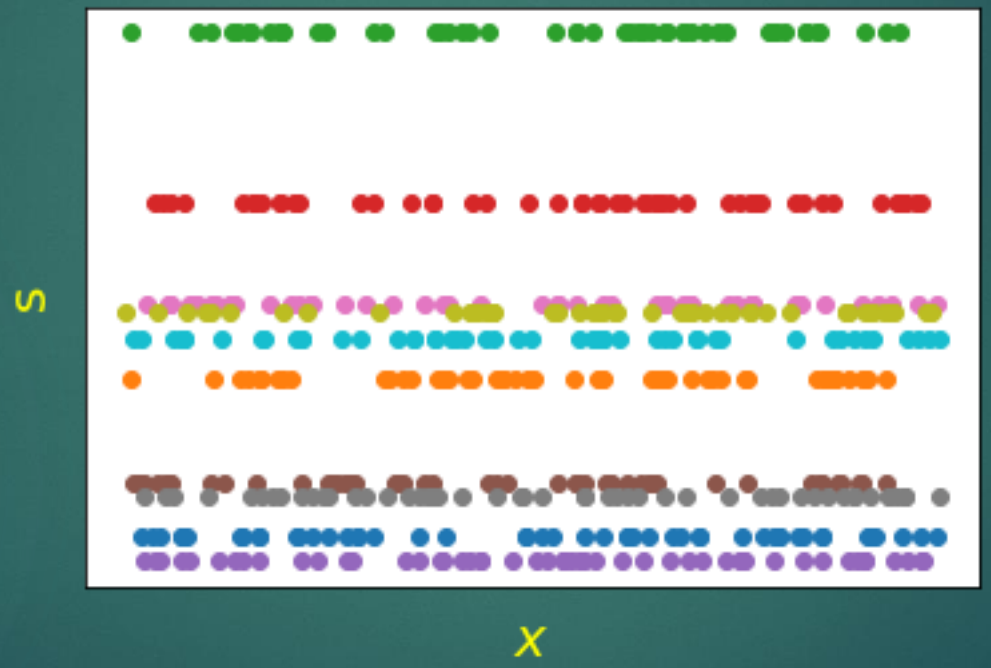
- Consider parametric integrals of the form

$$I(s_1, ..., s_m) = \int_0^1 dx_1 \cdots \int_0^1 dx_k \; f(s_1, ..., s_m; x_1, ..., x_k)$$

- x_i are auxiliary variables and s_i are the parameters
- Example: sector decomposition of loop integrals

Typical solution

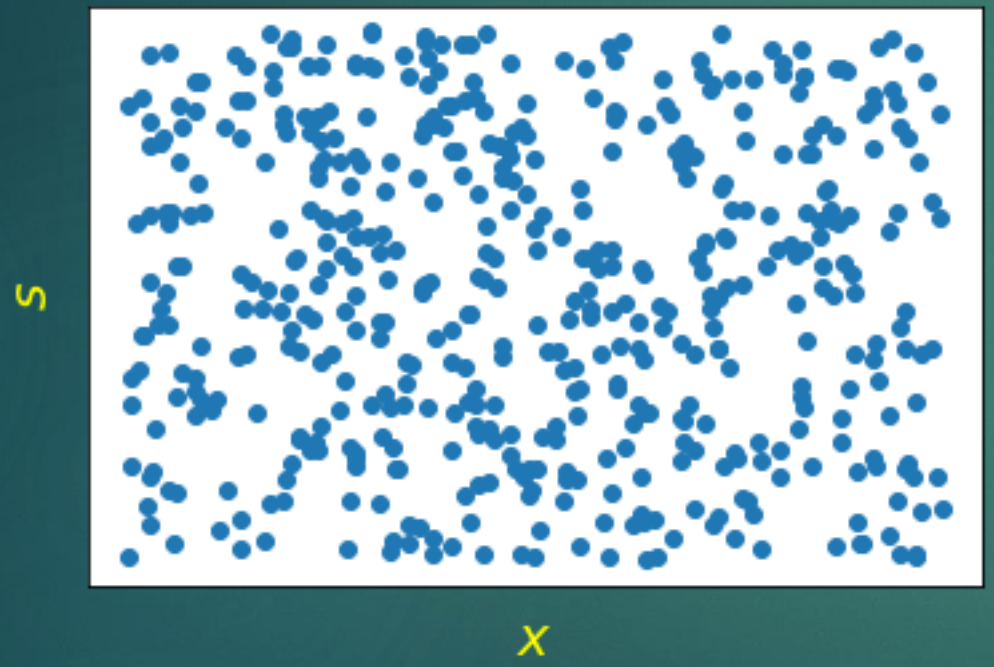
- Monte Carlo integration for each values of the parameters

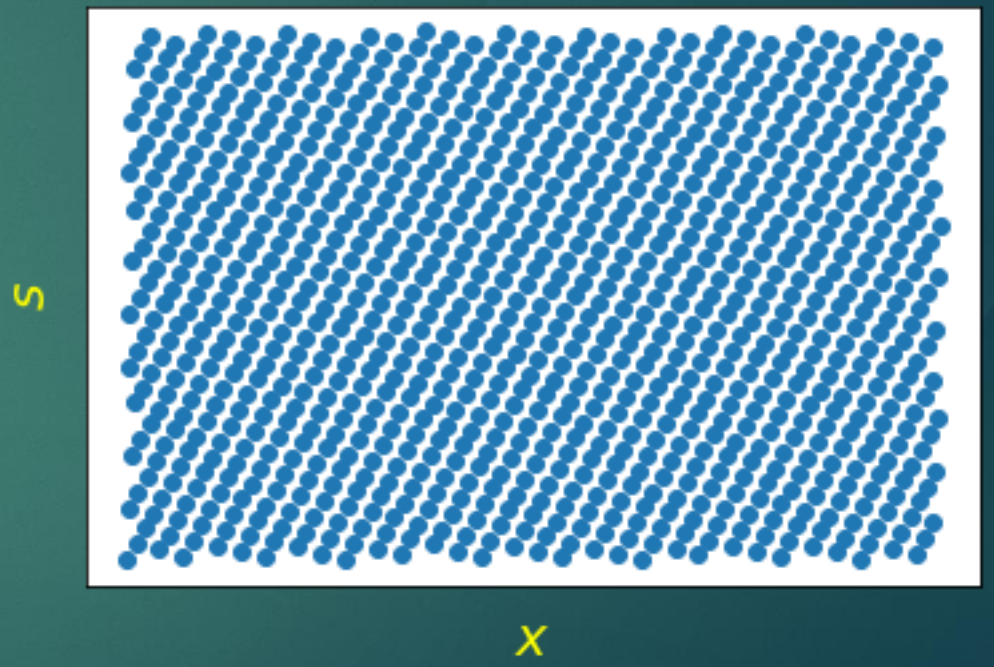


- Each run is independent

Alternative

- Sample the x-s space more uniformly





- Can leverage information on the integrand between separate evaluations

Primitive function

- Suppose we had a function with

$$\frac{d^k F(s_1, ..., s_m; x_1, ..., x_k)}{dx_1 \dots dx_k} = f(s_1, ..., s_m; x_1, ..., x_k)$$

- We can evaluate the integral as

$$I(s_1, ..., s_m) = \sum_{x_1, ..., x_k = 0,1} (-1)^{k - \sum x_i} F(s_1, ..., s_m; x_1, ..., x_k)$$

NN approximation

- We introduce a neural network approximation for the the primitive function

$$\mathcal{N}(s_1, ..., s_m; x_1, ..., x_k)$$

- Train it such that its derivative matches the integrand function

$$L = \text{MSE}\left(f(s_1, ..., s_m; x_1, ..., x_k), \frac{d\mathcal{N}(s_1, ..., s_m; x_1, ..., x_k)}{dx_1 ... dx_k} \right)$$

NN

- Standard network with *L* hidden layers

$$a_i^{(l)} = \phi\left(z_i^{(l)}\right) , \quad z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^l .$$

- Inputs

$$a_i^{(0)} = x_i \text{ for } i \leq k , \quad a_i^{(0)} = s_{i-k} \text{ for } i > k .$$

- output

$$y = \sum_j w_j^{(L+1)} a_j^{(L)} + b^{(L)}$$

Derivatives in the loss function

- The derivative in the loss function contains all the derivatives of the activation function up to degree k

$$\frac{d^p z_i^{(l)}}{dx_1 dx_2 ... dx_p} = \sum_j w_{ij}^{(l)} \frac{d^p a_i^{(l-1)}}{dx_1 dx_2 ... dx_p}$$

$$\begin{aligned}\frac{d^3 a_i^{(l)}}{dx_1 dx_2 dx_3} &= \phi^{(3)}(z_i^{(l)}) \frac{dz_i^{(l)}}{dx_1} \frac{dz_i^{(l)}}{dx_2} \frac{dz_i^{(l)}}{dx_3} \\ &+ \phi''(z_i^{(l)}) \left[\frac{d^2 z_i^{(l)}}{dx_1 dx_3} \frac{dz_i^{(l)}}{dx_2} + \frac{dz_i^{(l)}}{dx_1} \frac{d^2 z_i^{(l)}}{dx_2 dx_3} + \frac{d^2 z_i^{(l)}}{dx_1 dx_2} \frac{dz_i^{(l)}}{dx_3} \right] \\ &+ \phi'(z_i^{(l)}) \frac{d^3 z_i^{(l)}}{dx_1 dx_2 dx_3}\end{aligned}$$

Example 1

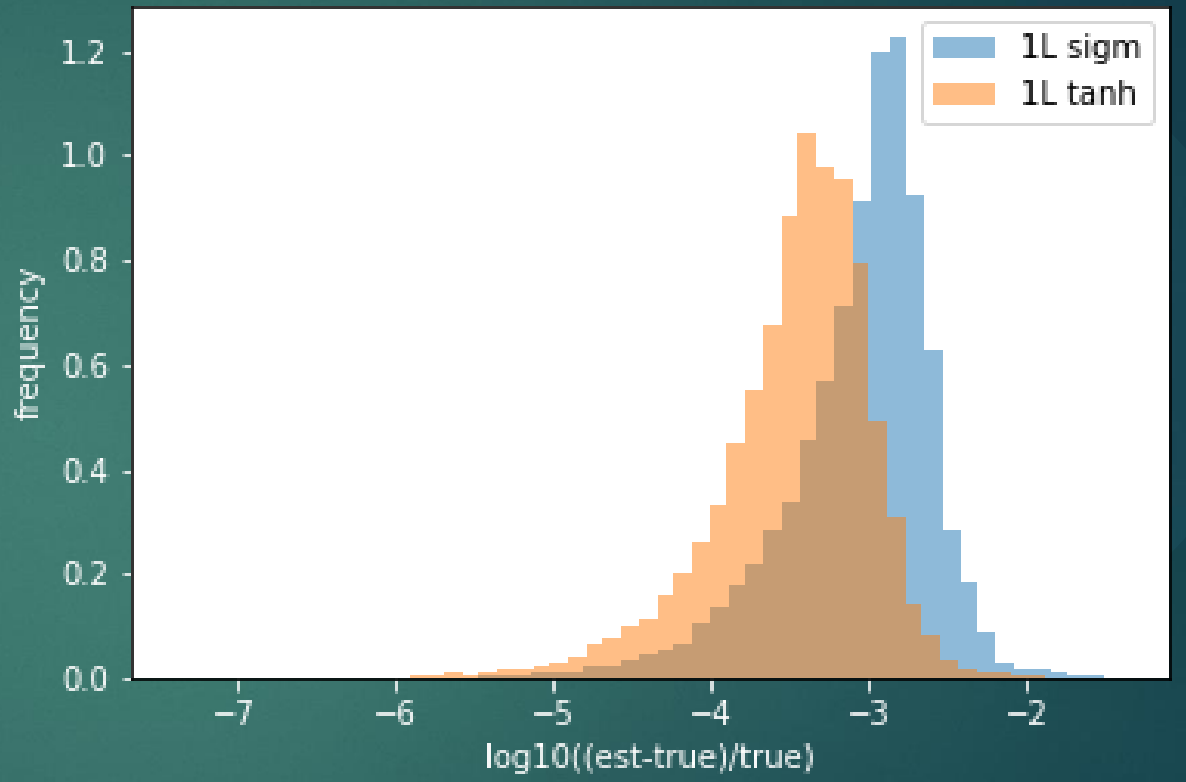
- 1-loop box for gg □ HH
- Four physical parameters: m_t^2, m_H^2, s_{12}, s_{14}
- Three Feynman parameters x_1, x_2, x_3
- 3 sectors generated by pySecDec
- Euclidean region

$$-30 \leq s_{12}/m_t^2 \leq -3 \qquad -30 \leq s_{14}/m_t^2 \leq -3 \qquad -30 \leq m_H^2/m_t^2 \leq -3$$

Result

- 100 nodes
- 4 hidden layers
- 4M x 800 = 3.2B PS points

$$p = \log_{10}\left|\frac{e-t}{t}\right|$$

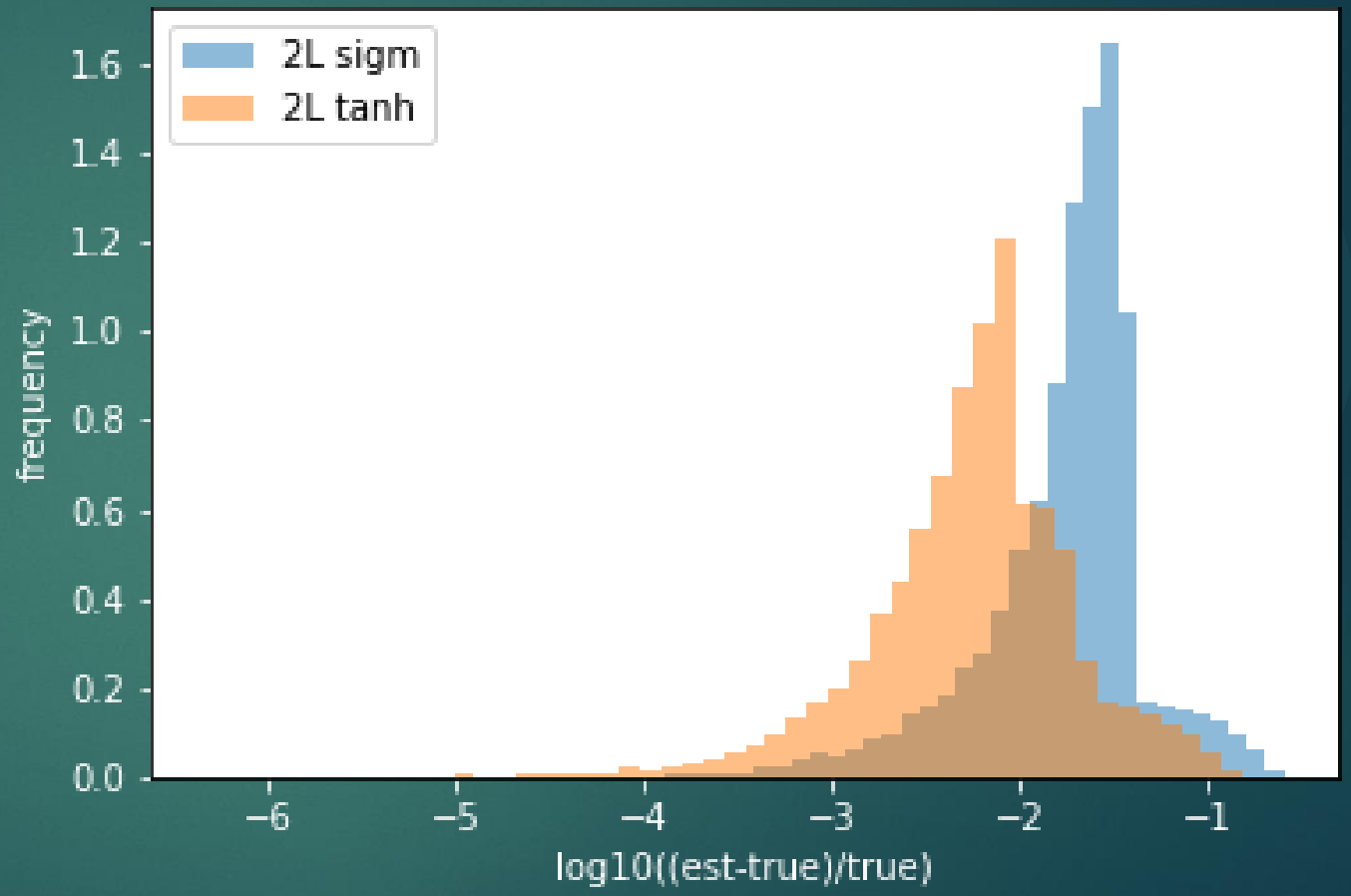


Example 2

- 2-loop box for gg□HH
- Same physical parameters
- 6 Feynman parameters
- 1 of 30 sectors from pySecDec

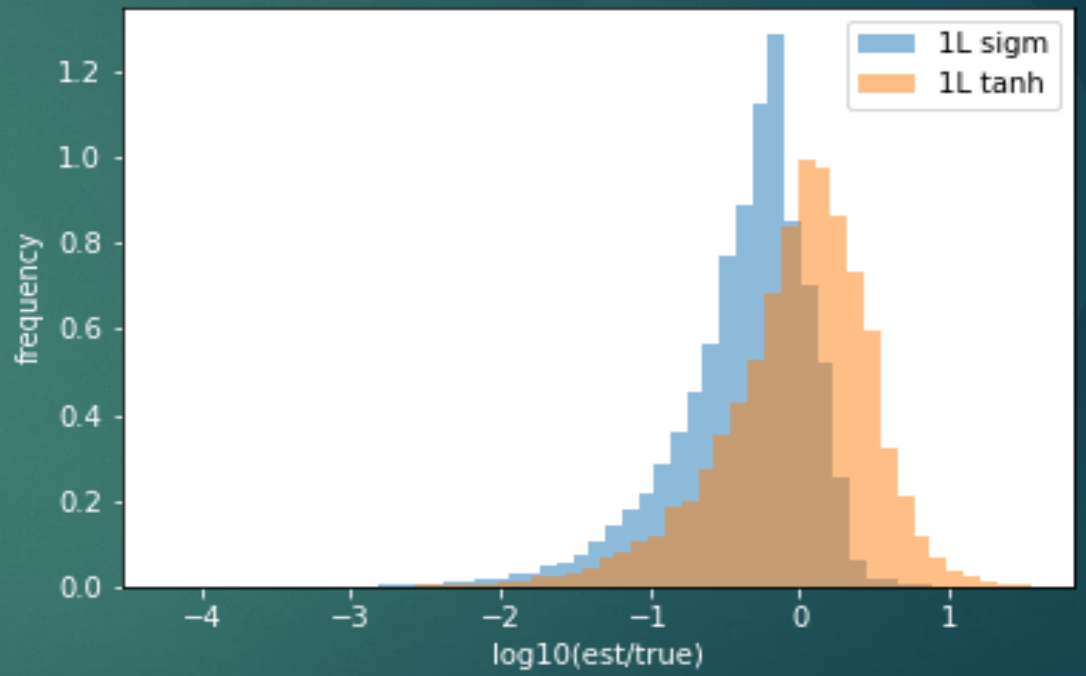
Results

- 30 nodes
- 4 hidden layers
- 800k x 200 PS points



Error estimate

- Use 4 replicas of the network
- Use average as the prediction
- Standard deviation as error estimate



Integration Neural Network

- Not interpolating between integrated values
- MC integration: Approximate integral of the exact integrand
- INN: Exact integral of an approximate integrand

Reducing variance

- Usual subtraction

$$\int_0^1 dx_1 \ldots \int_0^1 dx_k \left(f(x_1, \ldots, x_k) - s(x_1, \ldots, x_k)\right) \quad + \quad S$$

$$S = \int_0^1 dx_1 \ldots \int_0^1 dx_k s(x_1, \ldots, x_k)$$

- Using our neural network

$$\int_0^1 dx_1 \ldots \int_0^1 dx_k \left(f(x_1, \ldots, x_k) - \frac{d^k \mathcal{N}}{dx_1 \ldots dx_k}\right) \quad + \quad \sum_{b_i=0,1} \pm \mathcal{N}(b_i, \ldots, b_k)$$

Reducing variance

- Usual subtraction

$$\int_0^1 dx_1 \dots \int_0^1 dx_k \left(f(x_1, \dots, x_k) - s(x_1, \dots, x_k)\right) \quad + \quad S$$

$$S = \int_0^1 dx_1 \dots \int_0^1 dx_k s(x_1, \dots, x_k)$$

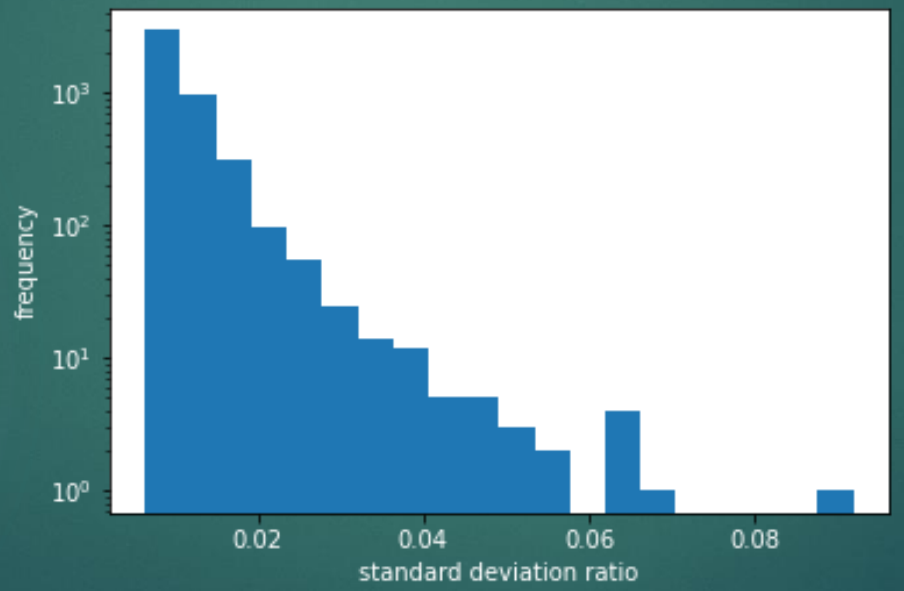
- Using our neural network

$$\int_0^1 dx_1 \dots \int_0^1 dx_k \left(f(x_1, \dots, x_k) - \frac{d^k \mathcal{N}}{dx_1 \dots dx_k}\right) + \sum_{b_i=0,1} \pm \mathcal{N}(b_i, \dots, b_k)$$

Lower variance !

Reduce variance

- 1-Loop example
- Ratio of variance with NN subtraction comparted to without subtraction



- Improvement for all physical parameters!

Outlook

- More work on training
  - Initialisation
  - Training data sequencing
- Size / depth of networks / number of replica
- More complicated examples
  - More integration variables
  - Combined sectors, combined integrals
  - Minkowsky space
- Use as a parametrized Gibbs sampler

Conclusion

- Neural networks are very versatile
- Best results are obtained by injecting physical knowledge
- Reasonable accuracy can be obtained
- If the accuracy is not sufficient, NN can still improve the efficiency of the “honest” method

Backup

INN Training

- Neural network training is similar to standard network but
  - Take care of initialization
  - Can choose our data
    - Random vs qmc grids
    - Size of sample
    - Re-use or generate new data
  - Pick activation function
    - Tanh/sigmoid in *N*: derivatives in Loss
    - Antiderivatives of tanh/sigmoid in *N*: tanh and sigmoid in loss (and lower antiderivatives)

INN Preprocessing

- Korobov transform

$$x = t^2(3-2t)\,, \qquad \int_0^1 dx f(x) = \int_0^1 dt\; 6t(1-t) f(x(t))$$

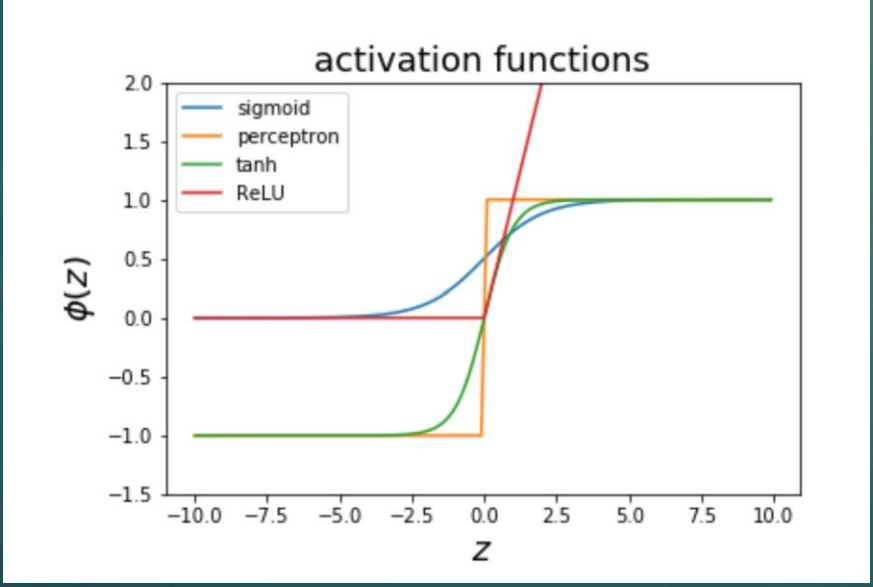
- Remove overall scaling

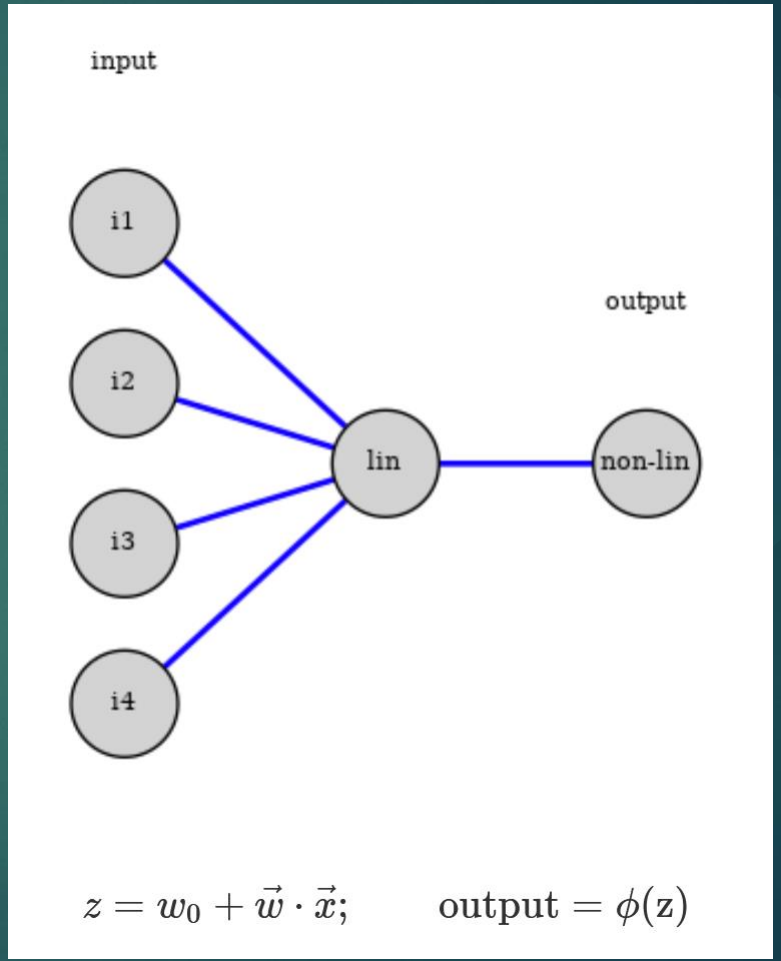
$$f \to \tilde{f}(s_1, ..., s_m; x_1, ..., x_k) \equiv \frac{f(s_1, ..., s_m; x_1, ..., x_k)}{f(s_1, ..., s_m; \frac{1}{2}, \frac{1}{2}, , ..., \frac{1}{2})}$$

- Could be a lot more radical using Normalising Flows to flatten the integrand

Neural Network

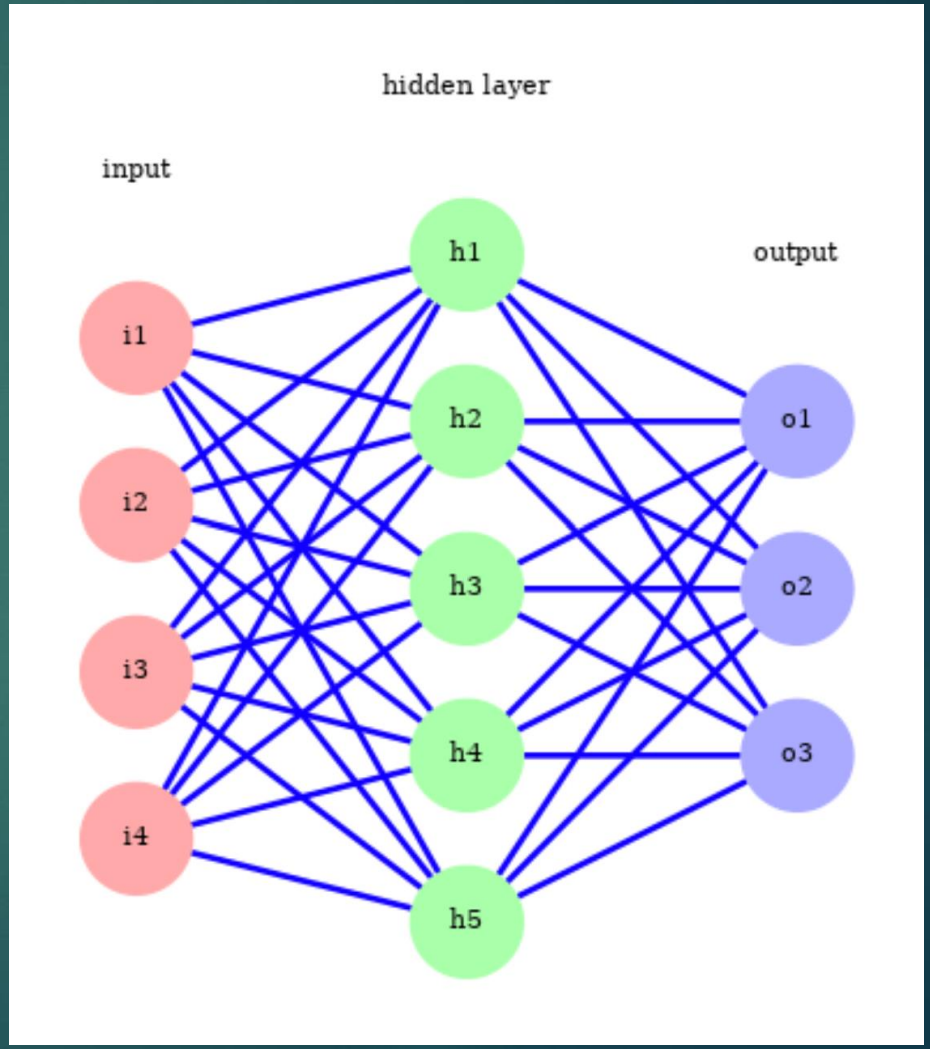
- (A)NN is composed of artificial neurons
- Crucial: the activation function is non-linear





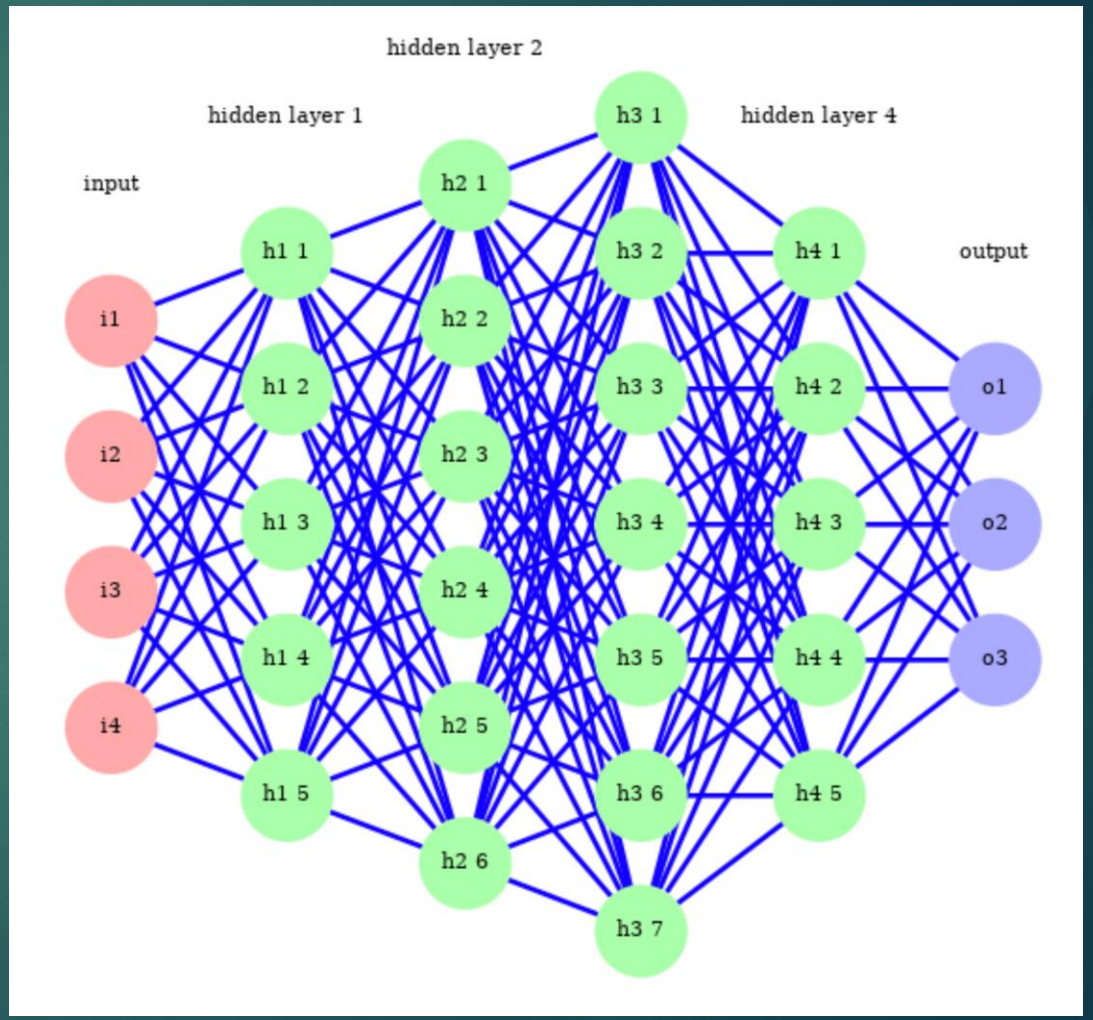
Neural Network

- Neurons are organised in layers
- Each layer of neurones takes the output of the previous layer as its input
- This example is function
$$f: R^4 \rightarrow R^3$$
- How many parameters?
  - 12
  - 43
  - 47
  - 103



Neural Networks

- Networks with many layers are called deep neural networks
- Used to be hard to train
- Now gigantic networks with billions of parameters can be trained (chatGPT-3 has 175 billion parameters)
- Universal approximator

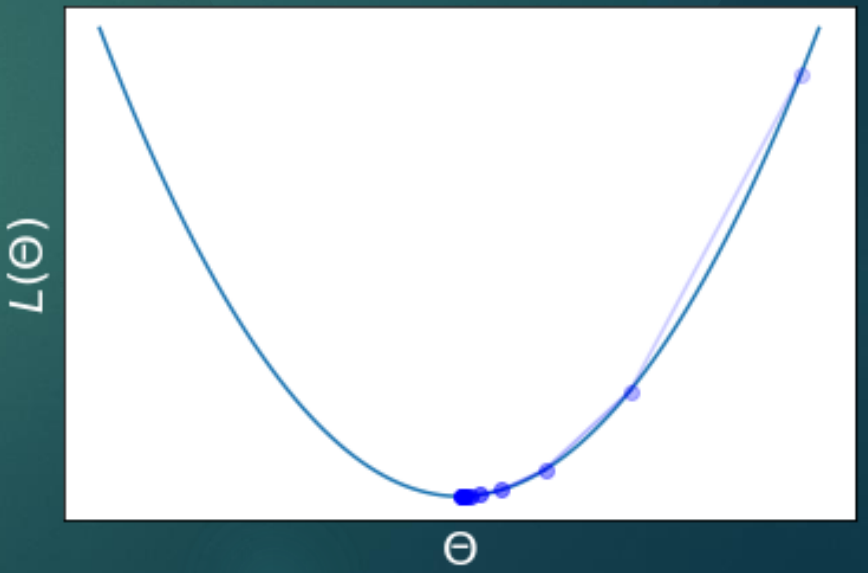


Neural Network Training

- Define an objective function to optimize
- Usually called loss function, to be minimized
- Mean squared error (appropriate when deviation from model are due to Gaussian distributed errors)

$$L_{\mathrm{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - p(\vec{x}_i;\theta))^2$$

- Parameters are optimized, usually using gradient descent
- Frameworks to automatically calculate the gradient exist (PyTorch, Tensorflow, JAX, …)



Neural Network training

- Introducing additional term in the loss function can help enforce other objectives
  - Prevent overfitting
  - Encourage sparse representation
- In this talk:
  - Use factorization
  - Make a NN integrate a function

Bias/variance tradeoff

- The prediction error can be decomposed into two component

$$\left\langle \left(y_{\text{pred}} - y_{\text{true}}\right)^2 \right\rangle = \left\langle \left(y_{\text{pred}} - \bar{y}\right)^2 \right\rangle + \left\langle \left(\bar{y} - y_{\text{true}}\right)^2 \right\rangle$$

Variance Bias

$\bar{y}$ is the expectation of the prediction under repeated fit from different training sets

- Bias ("how wrong are my prediction in average")
- Variance ("how different are my predictions")

Bias/Variance tradeoff

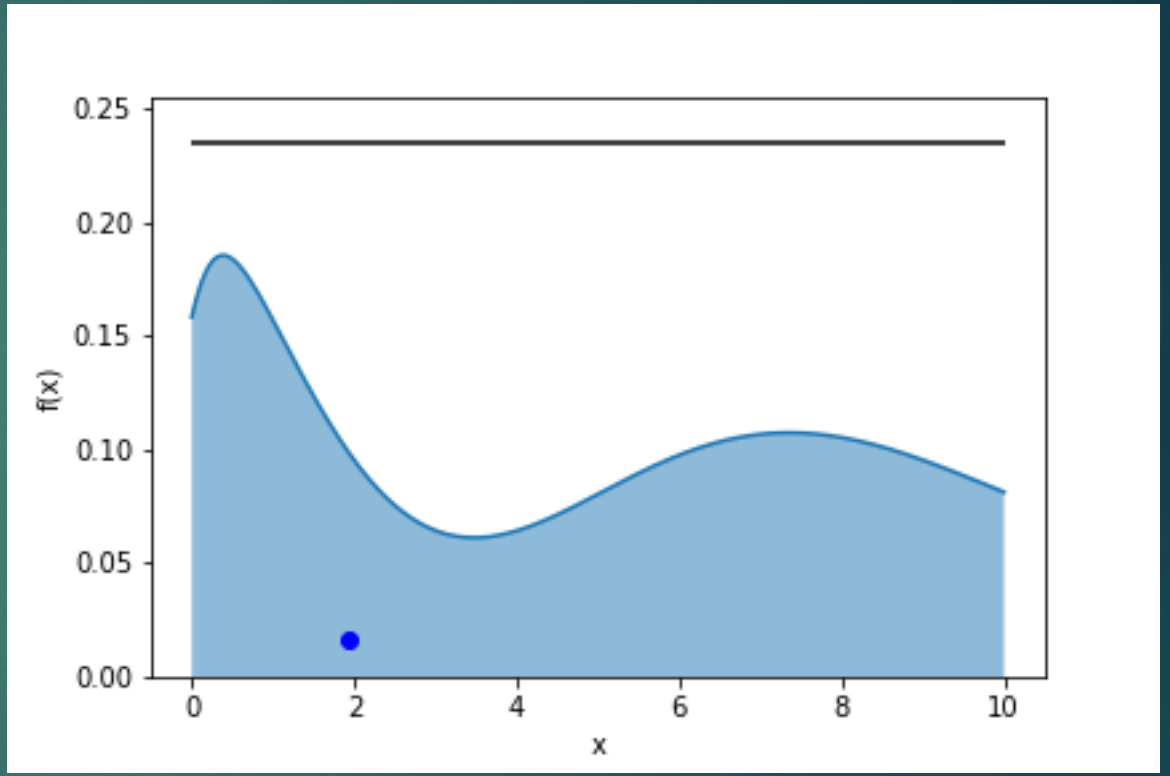
- Underconstrained model
  - Low bias
  - High variance
- Overconstrained model
  - High bias
  - Low variance
- NN tend to have many parameters
  - Estimate prediction error using prediction variance
  - Set of replica of the same NN with different initialisation

Unweighting

- Matrix element generators calculate weights for a given phase-space configuration
- If is often more useful to have a set of configuration where
  - each element has equal weight
  - Relative likelihoods are reflected in relative frequencies in the set
- Often used before detector simulation

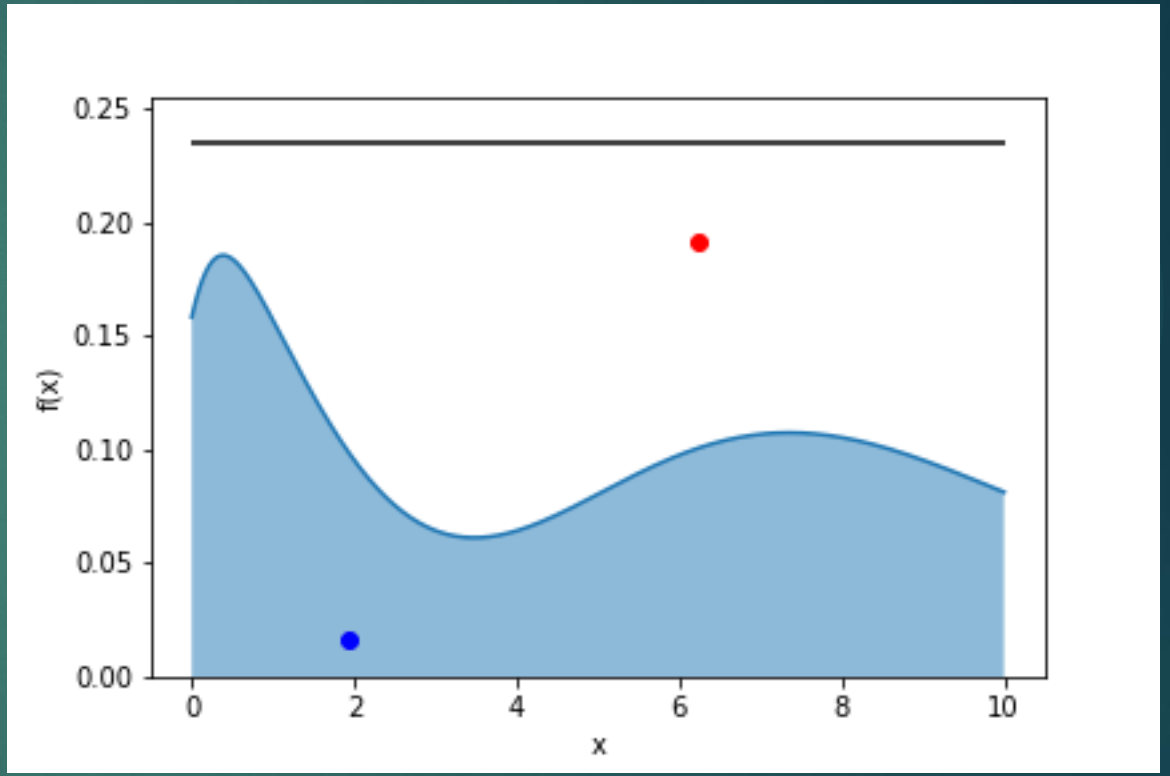
Unweighting

- For complicated distribution we use “hit and miss” method
- Set a maximum value m for the function
- For a new point x evaluate $f(x)$ and keep x in the set if $f(x) > m\,r$ where r is a random number between 0 and 1



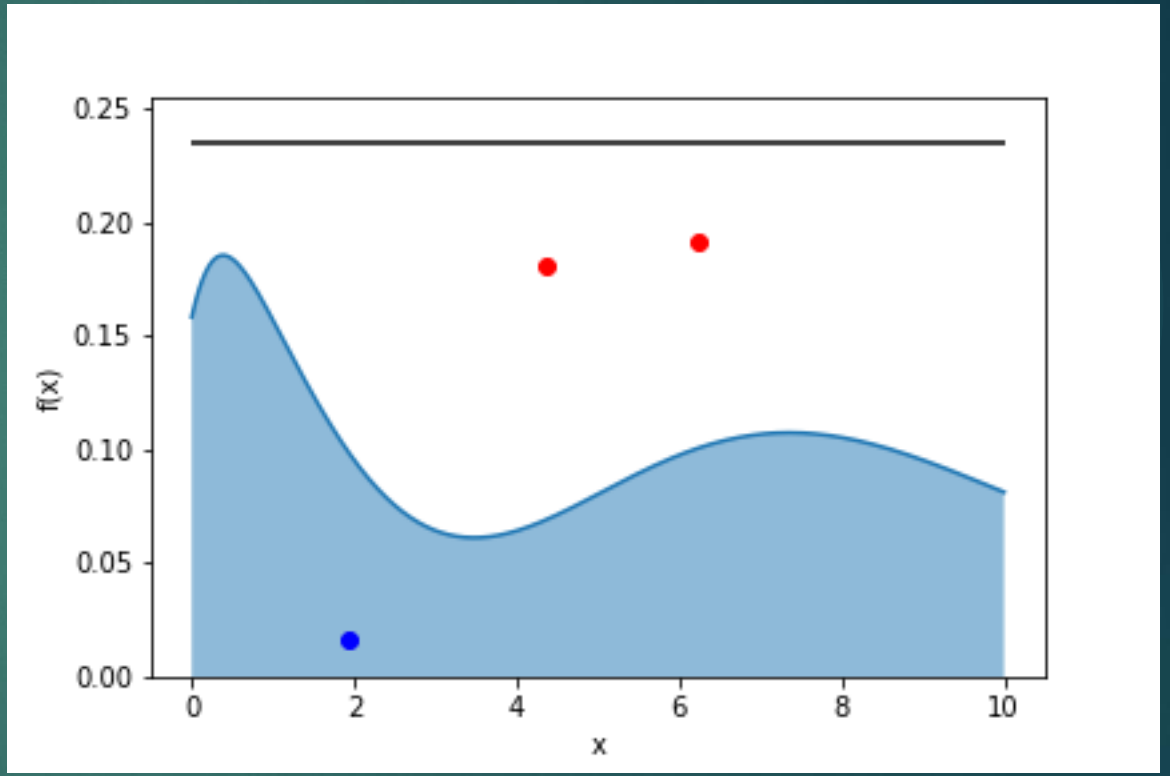
Unweighting

- For complicated distribution we use “hit and miss” method
- Set a maximum value m for the function
- For a new point x evaluate $f(x)$ and keep x in the set if $f(x) > m\,r$ where r is a random number between 0 and 1



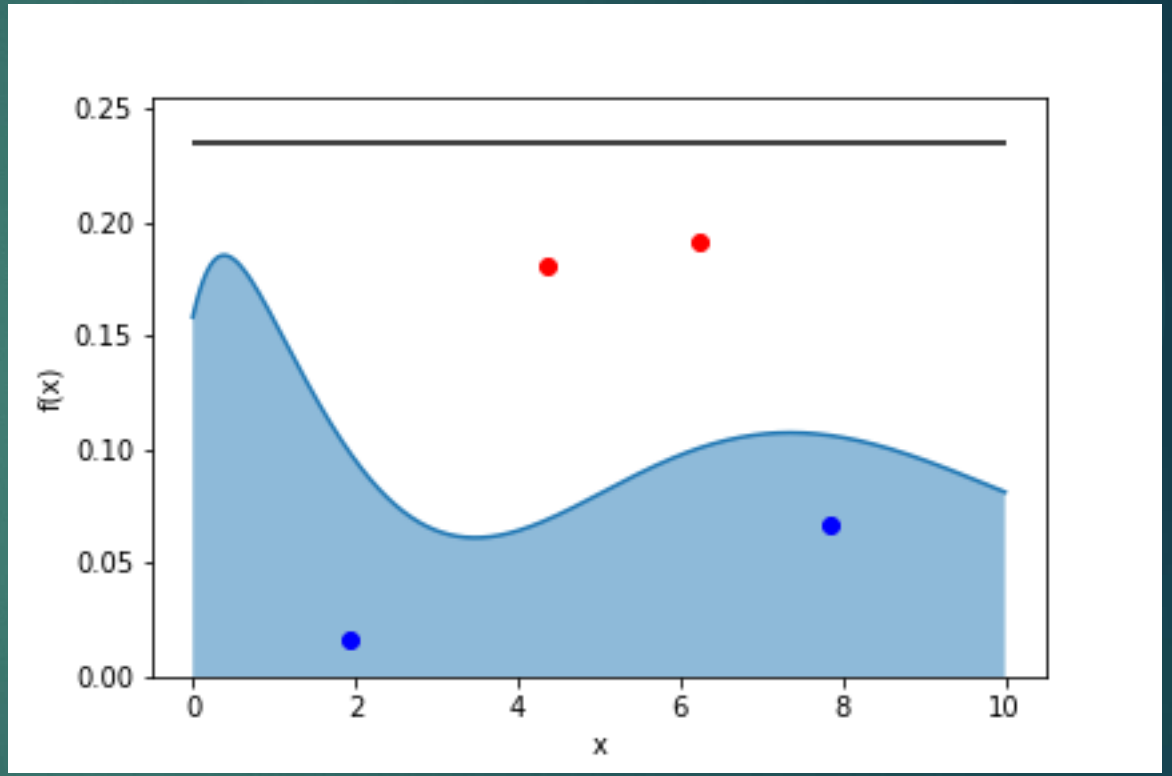
Unweighting

- For complicated distribution we use “hit and miss” method
- Set a maximum value m for the function
- For a new point x evaluate $f(x)$ and keep x in the set if $f(x) > m\,r$ where r is a random number between 0 and 1



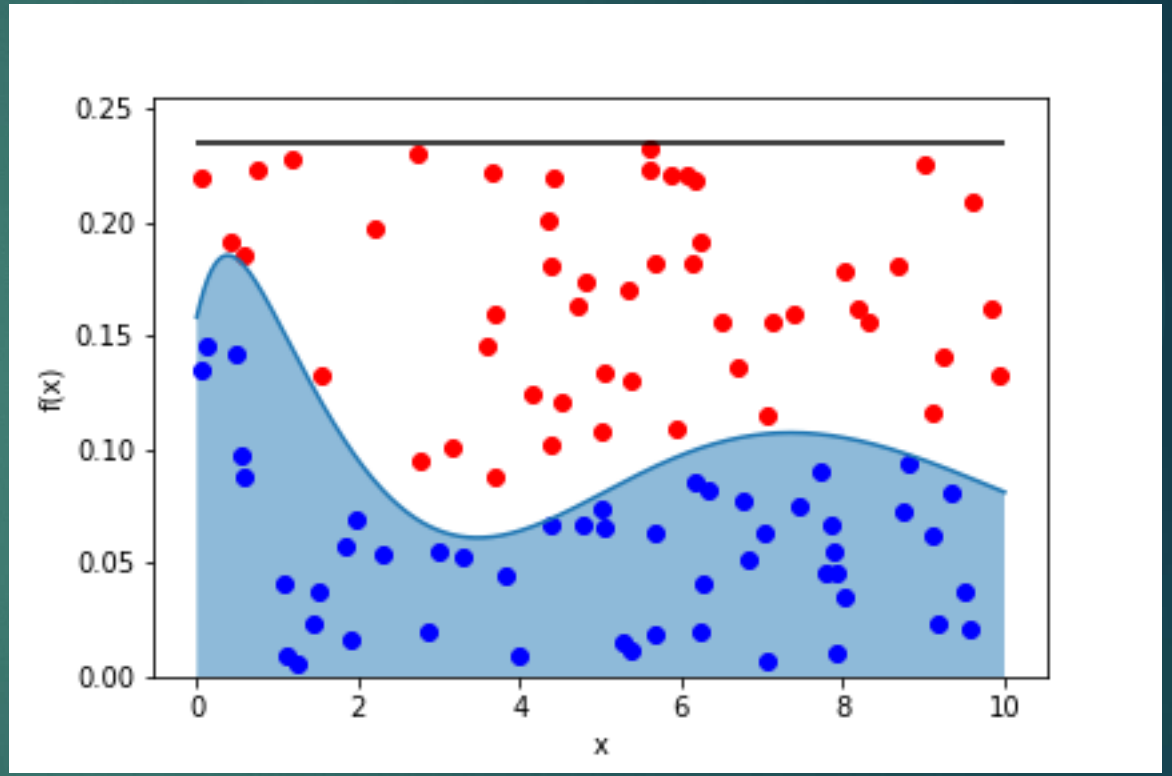
Unweighting

- For complicated distribution we use “hit and miss” method
- Set a maximum value m for the function
- For a new point x evaluate $f(x)$ and keep x in the set if $f(x) > m\,r$ where r is a random number between 0 and 1



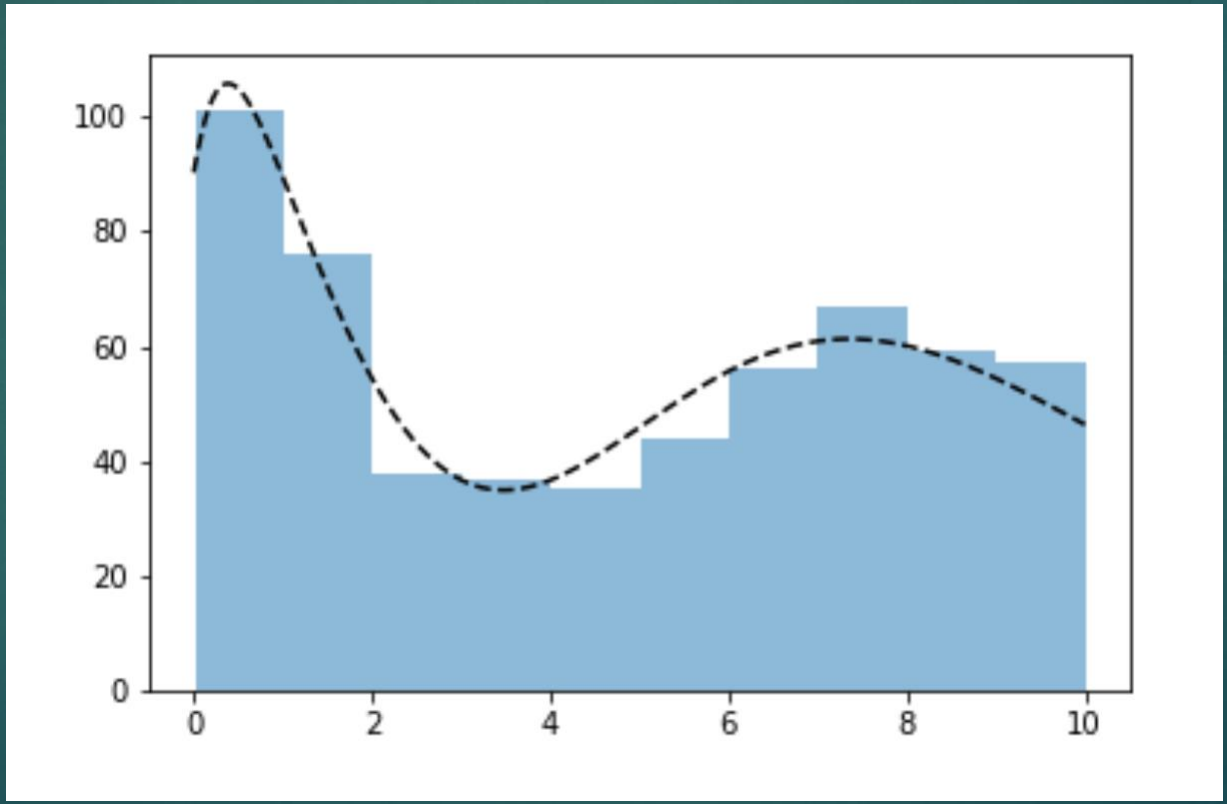
Unweighting

- For complicated distribution we use “hit and miss” method
- Set a maximum value m for the function
- For a new point x evaluate $f(x)$ and keep x in the set if $f(x) > m\,r$ where r is a random number between 0 and 1
- Unweighting efficiency is the ratio of the number of kept point over the total number of points



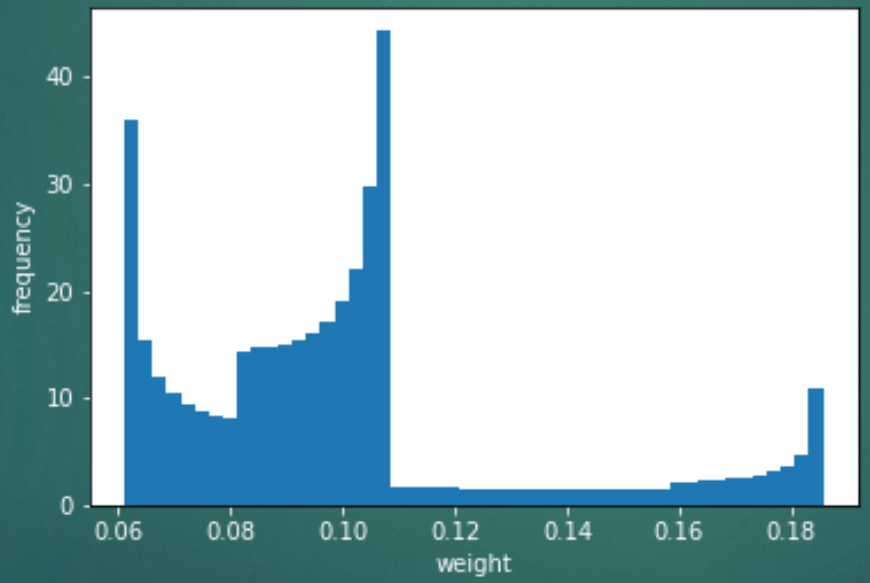
Unweighting

- Points we kept have the right distribution



Unweighting

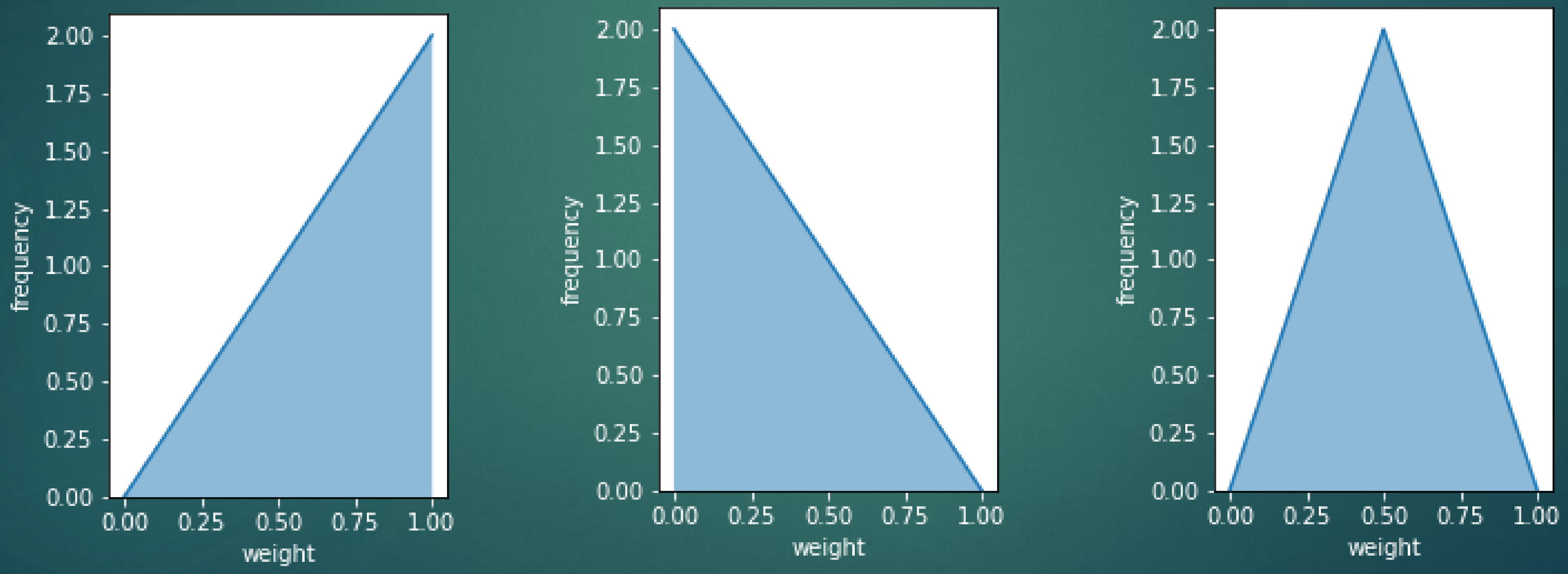
- If m is set too high we reject many events
- We need to calculate f for all attempts



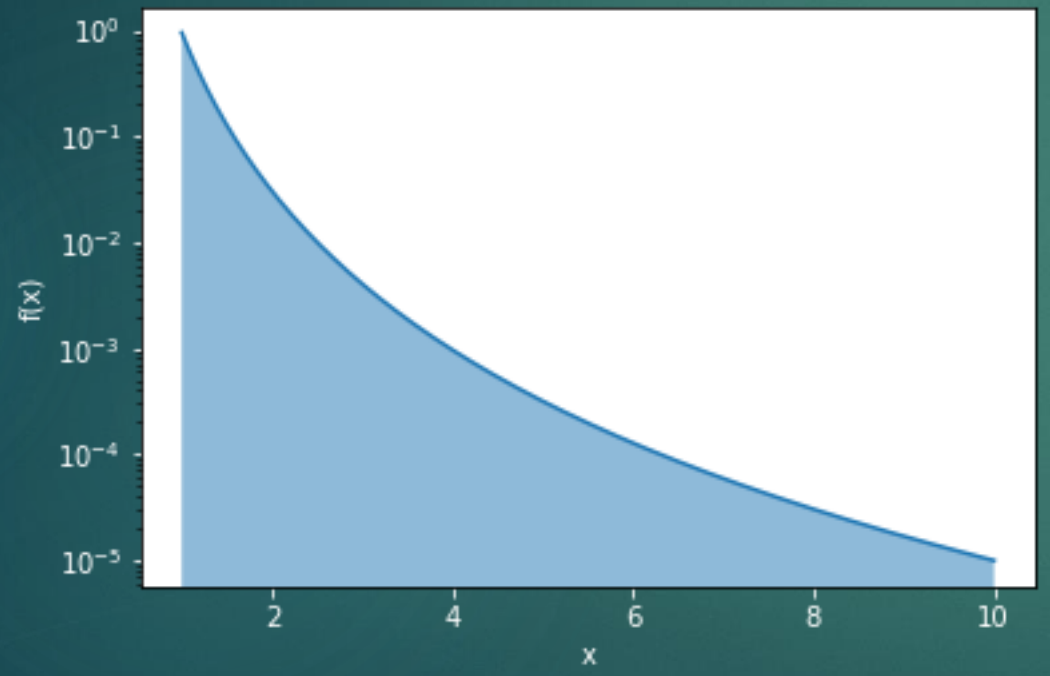
- Large spreads in weights lead to low efficiency

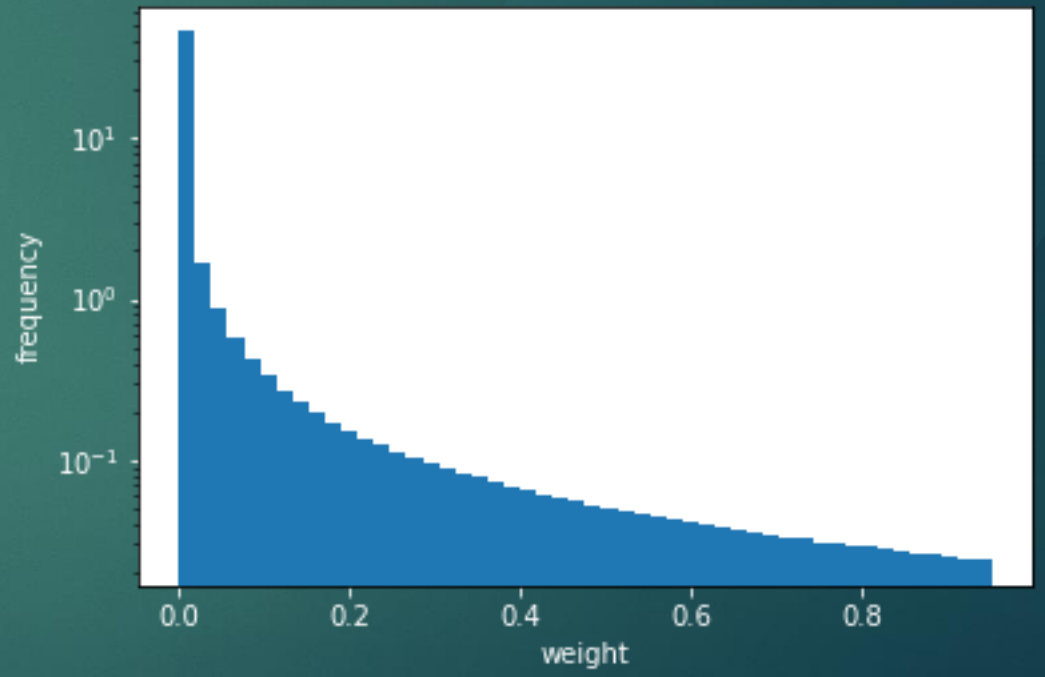
Unweighting

- Which distribution of weights has the largest unweighting efficiency?



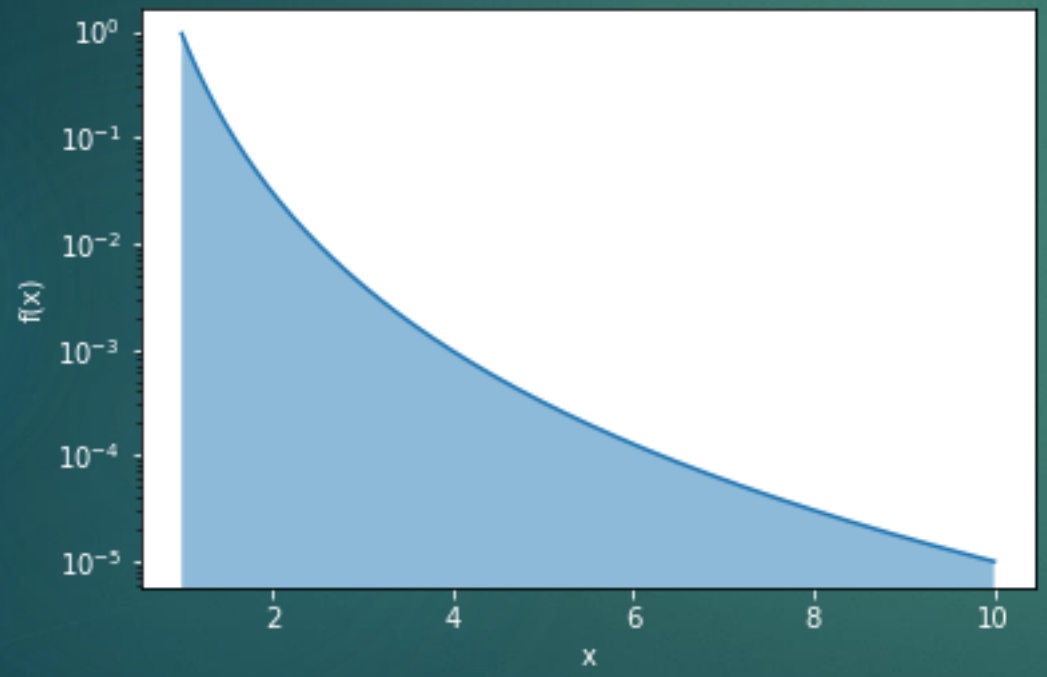
Unweighting

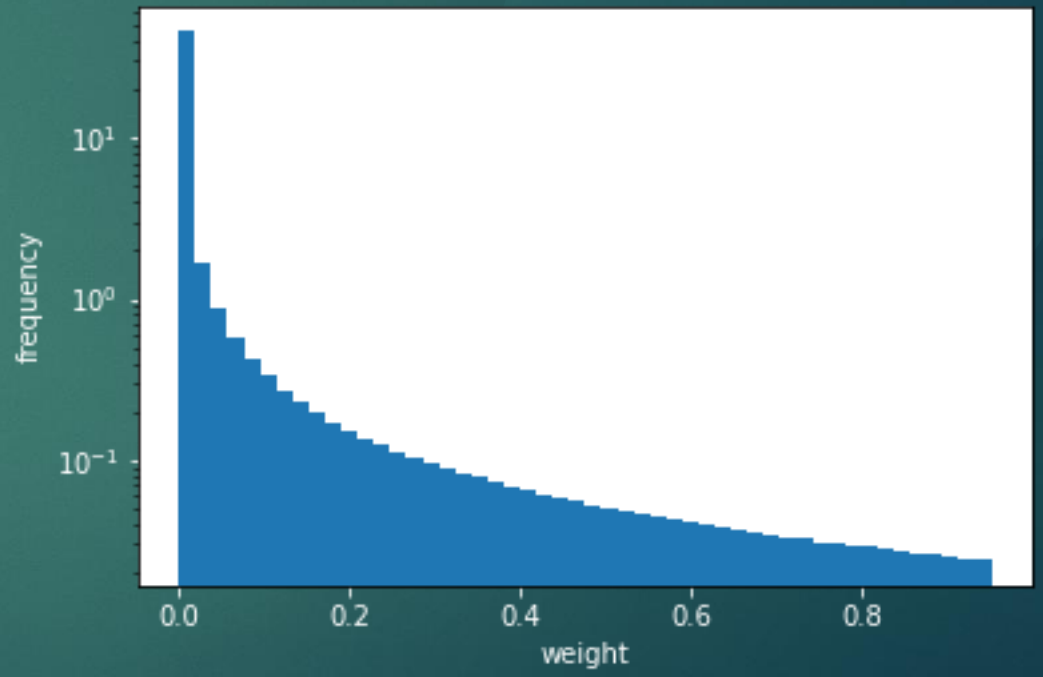




Unweighting

- Bad scenario:





Unweighting

- Can unweight in stages
- Generate a set of unweighted values x according to an approximation $g(x)$ that is easy to unweight
- Unweight this set according to $\frac{f(x)}{g(x)}$