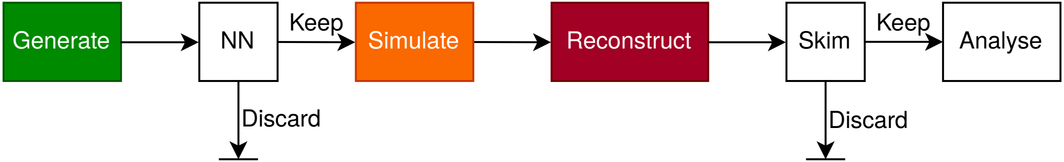
Selective Background Monte Carlo simulation with deep learning

“SmartBKG”

Nikolai Hartmann, Yannick Bross
(based on previous work by James Kahn, Andreas Lindner, Kilian Lieret)

September 22, 2020

Introduction



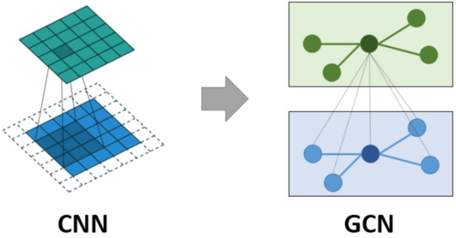
- Event generation takes much less computing time than detector simulation
- Many events discarded (e.g. by skim)
  $\rightarrow$ try to predict which events will be discarded, already after event generation

Status of the project

- Initiated by James thesis
  - models mainly based on CNNs (RNNs, MLPs also tested)
  - train on low-level event record data (MCParticles)
  - Feed graph structure via "decay string"
- At Dresden Deep Learning Hackathon Kilian, James, Andi, Emilio worked out some graph network architectures that are promising
  → See James talk last year and at CHEP
- Yannick is studying bias mitigations
  → more later
- I'm starting to get back into the project

Graph convolutional networks (GCNs)

Aggregate neighboring node features - similar to neighboring pixels in CNNs:

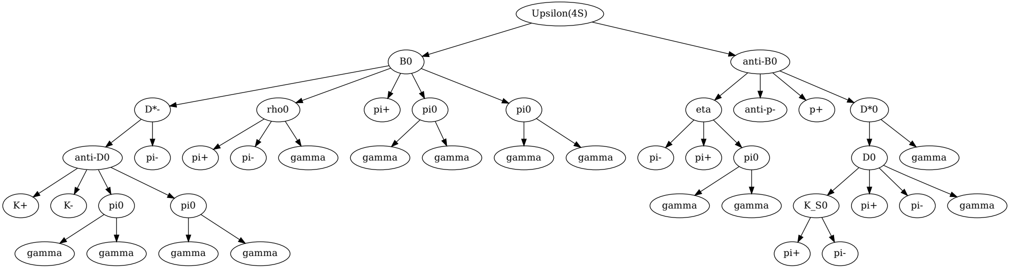


Simple update rule from Kipf & Welling (arXiv:1609.02907):

$H^{(l+1)} = \sigma(GH^lW^l)$ with $G = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$, $\tilde{A} = A + I$ and degree matrix D

$\rightarrow$ in contrast to CNNs no learnable relative weights between nodes
$\rightarrow$ can be added from knowledge about graph structure
(e.g. here: tree decay - can apply learnable weights to parent and daughter edges)

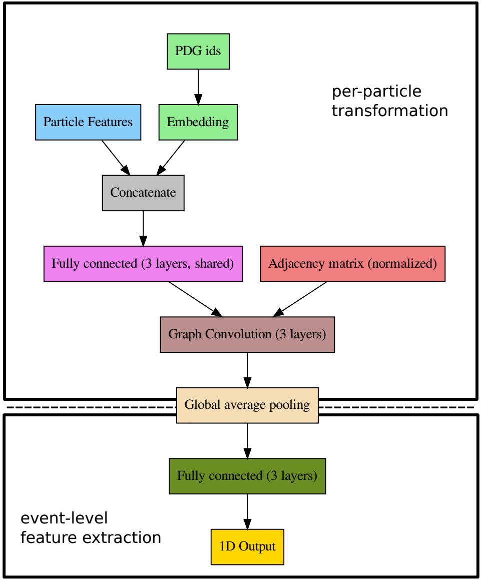
The input features (generator level event record)



- Adjacency matrix with daughter - mother connections
- PDG ids
  $\rightarrow$ fed through embedding layer
- p_x, p_y, p_z, E
- vertex position x, y, z
- Production time

Reference architecture

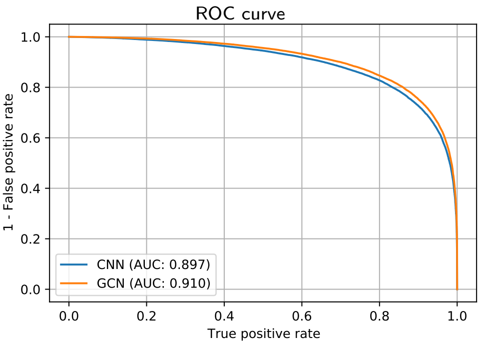
- Adjacency matrix format: Normalized ($D^{-1/2}AD^{-1/2}$), symmetrized (both Mother$\rightarrow$Daughter, Daughter$\rightarrow$Mother edges and self-loops)
- Particle transformation:
  - PDG embeddding, concatenate with other particle features
  - 3 Dense layers, 128 neurons each, relu
  - 3 GCN layers, 128 output features each, relu (also experimented with swapping the order etc.)
- GlobalAveragePooling1D (masking padded particles)
- Final transformation: 3 Dense layers, 128 neurons each, relu
- One output value (for classification, sigmoid)

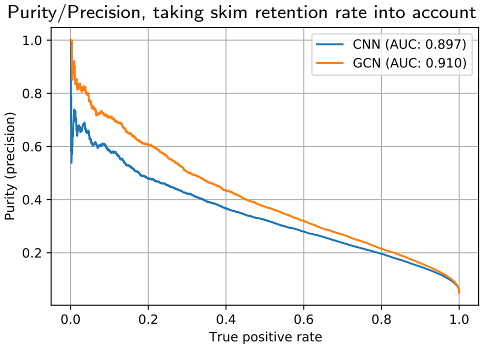

PDG ids
per-particle
transformation
Particle Features
Embedding
Concatenate
Fully connected (3 layers, shared)
Adjacency matrix (normalized)
Graph Convolution (3 layers)
Global average pooling
Fully connected (3 layers)
event-level
feature extraction
1D Output

Dataset and training

- FEI hadronic B0 skim on mixed samples:
  - Full reconstruction of hadronic B0 decays (tag side)
  - Filter on reconstructed candidate (e.g. beam constrained Mass) + event level quantities
- $\approx$ 1M training events (roughly balanced)
- Preprocessing:
  - Particle lists cropped at/padded to 100
    $\rightarrow$ actually works quite well with much less (40 used before)
    $\rightarrow$ mostly crops particles at final stages of decay
  - Adjacency matrix retrieved from one-hot encoding remapped mother particle indices
- Want to provide this (maybe reduced) for IDT ErUM-Data classifier comparison
- Train with batch size 1024
- Binary cross entropy loss
- Stop after no improvement on validation set (20% of training data, wait 10 epochs)

Performance comparison CNN/GCN





→ GCN model slightly better than CNN model, but simpler (no decay strings)

Speedup factor

How many more events can you get with the same computing time?

Depends on:

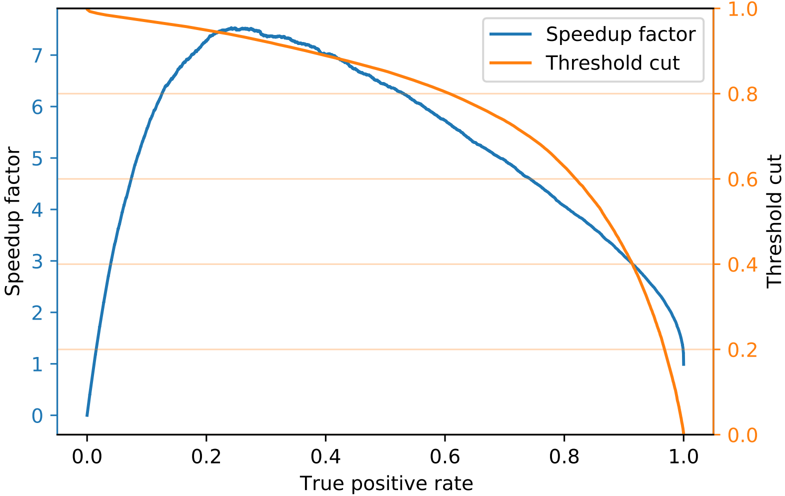
- True positive rate, False positive rate
- Relative processing times between:
  - Simulation + reconstruction
  - Event generation
  - Getting the NN prediction
- Skim retention rate (inital purity)

Reference values (shamelessly rounded from James measurements):

- Ratio between simulation+reconstruction and event generation: ≈ 1000
- Ratio between NN evaluation and event generation: ≈ 10
- Skim retention rate (for FEI skim): ≈ 0.05

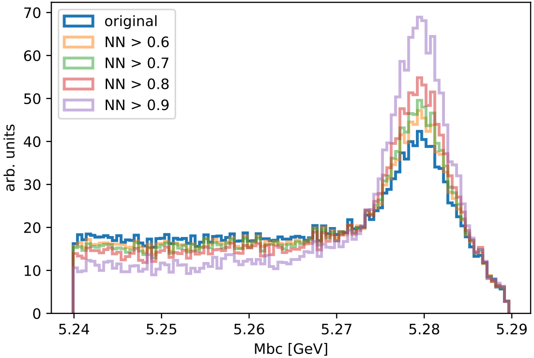
Speedup factor for GCN model

(assuming numbers from previous slide)



Bias

Events that pass skim (True events):



Due to false negative events (True events that we rejected) we can get a bias in the distribution of certain quantities

→ effect strongest for observables that the skim cuts on

Mitigation via distance correlation loss

Master thesis Yannick Bross

Try to mitigate this by adding a loss term that scales with the correlation between the NN output and one or more observables that should not be biased after a selection

$$L_{\text{tot}} = \text{BCE}(y_{\text{True}}, y_{\text{pred}}) + \lambda \cdot \text{dCorr}(x_{\text{decorr}}, y_{\text{pred}})$$

Distance correlation (see Wikipedia for formula)

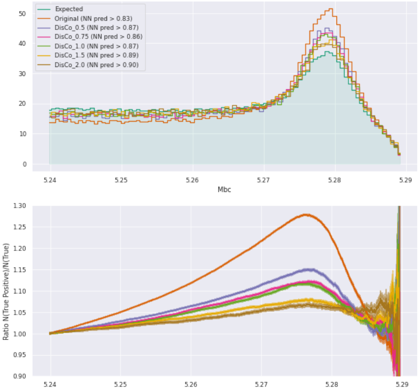
- Sensitive to non-linear correlations
- 0 If and only if there is no correlation between the quantities
- Usage in loss function for particle physics problems inspired by arXiv:2001.05310

→ See `https://github.com/gkasieczka/DisCo` for a tf and pytorch implementation

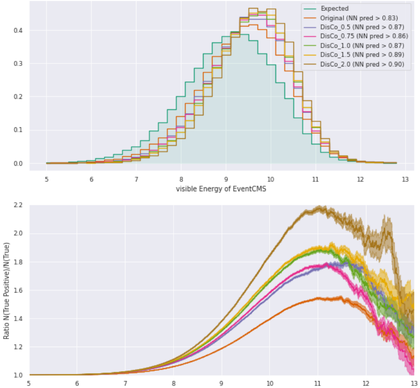
Tuning the relative loss contributions

Master thesis Yannick Bross

Effective for the variable trained on
$\rightarrow$ lower bias for same speedup factor



But: mitigation for one quantity can make bias for others worse

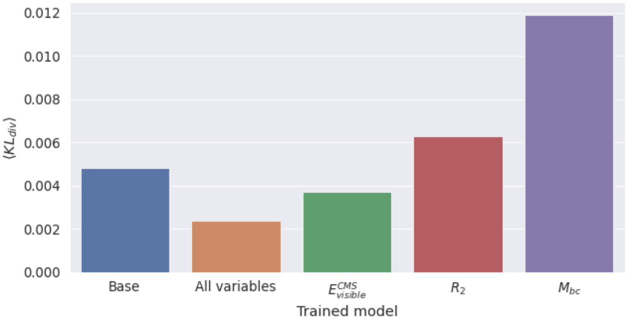


The contribution for the additional loss term can be tuned via a weight λ

$$L_{\text{tot}} = \text{BCE}(y_{\text{True}}, y_{\text{pred}}) + \lambda \cdot \text{dCorr}(x_{\text{decorr}}, y_{\text{pred}})$$

Including many variables

Master thesis Yannick Bross

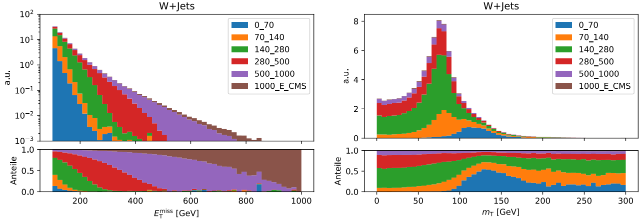


- Distance correlation contribution can in principle be added for arbitrary many variables
- Tried for all ≈ 30 variables considered
- Could reach best reduction in overall bias
  (in terms of average KL divergence of density histograms for these variables)

Experiment-overarching potential

Application for ATLAS? Bachelor theses by Michael Fichtner and Simon Graetz

Example for suboptimal filtering:



- Motivation: Some filters suboptimal for certain selections
  $\rightarrow$ can we improve with ML?
- Tried to apply similar techniques to ATLAS MC
  $\rightarrow$ pp collisions are different
  (number of generator particles $\gg$ 1000 instead of $<$ 100 at Belle II)
- More difficult to extract meaningful information from low-level event-record
- MC generators also more compute-intense
  $\rightarrow$ would need to apply NN in earlier stage

Ideas for the future

- Try to have one metric to summarize performance + bias mitigation
  → maybe statistical error after reweighting a set of distributions to correct remaining bias?
    - James thesis: train NN to distinguish between true and true positive events and reweight according to output histogram
      → stat error from accept/reject sampling
      → alternative: uncertainty on weighted events $\sqrt{\sum w_i^2}$
    - Another approach: BDT reweighting or reweighting according to predicted probability
- James idea: directly optimize for maximum speedup
- Different direction: More "classical filtering"
    - Simulate some rejected events as well, weight up by inverse filter efficiency
    - Can do this in "slices"
    - Continuous version: determine probability to be rejected individually per event (e.g. based on NN output)
      → can i train an NN to directly spit out the "optimal" sampling probability?

→ No biases with this approch, but comes at the price of increased stat error $\sqrt{\sum w_i^2}$

Backup

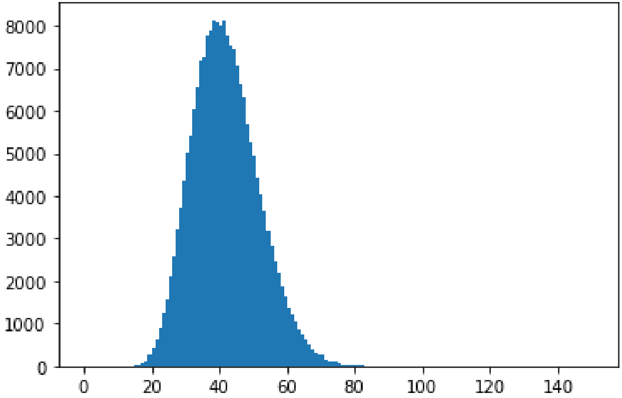
Tuning the relative loss contributions

Studies by Yannick

Bias gets reduced more with higher λ, but comes at cost of lower maximum achievable speedup

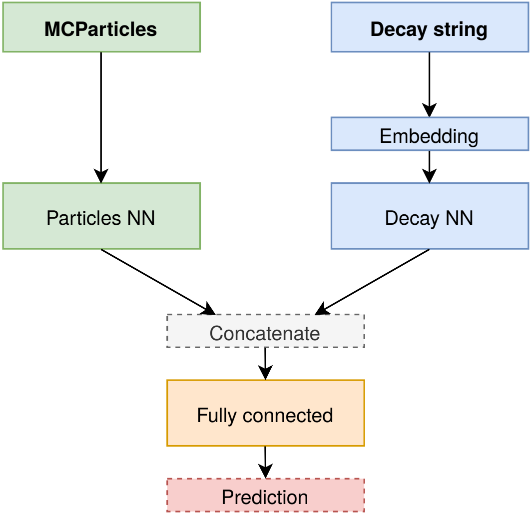
		vECMS	Mbc				
lambda	0	0.5	0.5	0.75	1.0	1.5	2.0
Max Speedup	7.6	7.4	6.7	6.7	6.5	6.4	6.1

Number of primary MCParticles per event



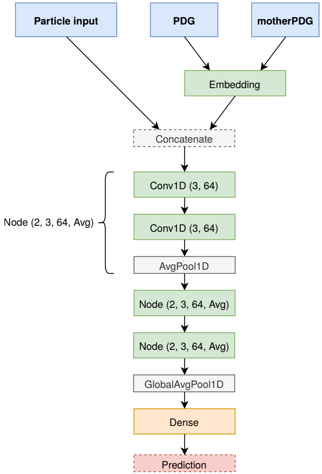
Vanilla CNN + wide CNN architecture (old ref.)

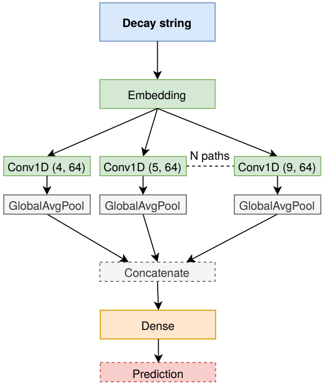
Graphics from James



Vanilla CNN + wide CNN architecture (old ref.)

Graphics from James, these studies: only one “Node” block for Vanilla CNN





Loss for maximum speedup

Preliminary/Experimental! Do not use without understanding everything it does wrong!

```
def inverse_speedup(
    y_true,
    y_pred,
    skim_retention=0.05,
    ratio_gen_simrec=0.001,
    ratio_nn_gen=10,
    from_logits=True
):
    if from_logits:
        y_pred = tf.keras.activations.sigmoid(y_pred)
    tpr = (
        tf.reduce_sum(y_pred[y_true==1], axis=0)
        / tf.reduce_sum(y_true, axis=0)
    )
    y_false = tf.cast(y_true == 0, tf.float32)
    fpr = (
        tf.reduce_sum(y_pred[y_true==0], axis=0)
        / tf.reduce_sum(y_false, axis=0)
    )
    r = skim_retention
    pp = tpr / (tpr + (1 - r) / r * fpr) # purity with NN
    p = skim_retention # initial purity
    f_gr = ratio_gen_simrec
    f_ng = ratio_nn_gen
    return inv_speedup = (
        p / pp + p * (f_gr * (f_ng + 1.0)) * (1.0 / tpr + (1.0 / pp - 1.0) / fpr)
    ) / (1.0 + f_gr)
```

Loss for minimum effective uncertainty (sampling method)

Preliminary/Experimental! Do not use without understanding everything it does wrong!

```
def effective_uncertainty(y_true, filter_prob, class_weights=[1, 0.05], from_logits=True):
    if from_logits:
        filter_prob = tf.keras.activations.sigmoid(filter_prob)

    effective_selected_true = tf.reduce_sum(filter_prob[y_true==1]) * class_weights[1]
    effective_selected_false = tf.reduce_sum(filter_prob[y_true==0]) * class_weights[0]
    filter_eff = (
        (effective_selected_true + effective_selected_false)
        / (
            tf.reduce_sum(tf.cast(y_true==1, tf.float32)) * class_weights[1]
            + tf.reduce_sum(tf.cast(y_true==0, tf.float32)) * class_weights[0]
        )
    )

    weights = 1. / filter_prob

    # sum(w) = sum(p * 1 / p) = N
    # sum(w**2) = sum(p * (1 / p) ** 2) = sum(1 / p)
    sumw = tf.reduce_sum(tf.cast(y_true==1, tf.float32))
    sumw2 = tf.reduce_sum(weights[y_true==1])
    # effective sample size (sample size with same relative uncertainty)
    neff = (sumw ** 2) / sumw2

    neff *= class_weights[1]
    # i could have simulated this factor of more events, due to filtering
    neff /= filter_eff

    return tf.sqrt(neff) / neff
```

Slicing: Relation filter efficiency - sampling probability

Suppose we want that a fraction f of a total number N_{tot} of events ends up in a certain slice that has a filter efficiency of ϵ. Then the number of events that will be generated for that slice is given by

$$fN_{\text{tot}} = \epsilon p N_{\text{tot}}$$

with the sampling probability p. Consequently the filter efficiency ϵ is given by

$$\epsilon = \frac{f}{p}$$

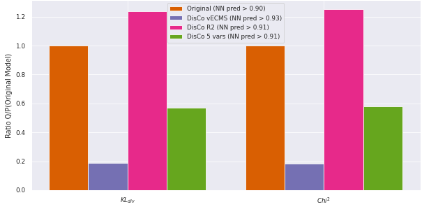
Since (up to luminosity/cross section normalization) each event is weighted by $\frac{\epsilon}{N}$ this corresponds to a weight with the inverse sampling probability and an overall normalization $\frac{1}{N_{\text{tot}}}$

$$w = \frac{\epsilon}{N} = \frac{f}{pfN_{\text{tot}}} = \frac{1}{pN_{\text{tot}}}$$

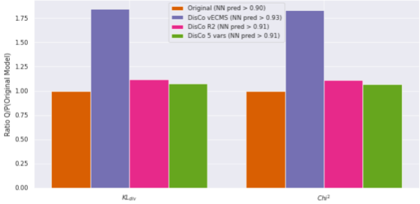
Metrics to evaluate bias

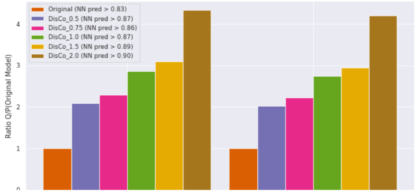
Studies by Yannick

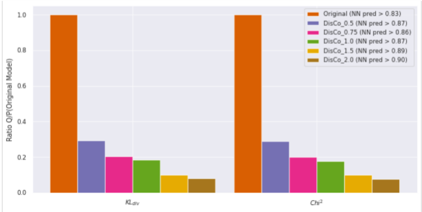
visible Energy of Event CMS



Mbc



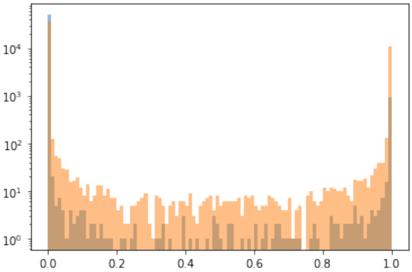




Good quantities for measuring bias:
Relative difference in KL divergence or Mean squared error of histograms

First try: Directly optimize speedup

NN output:



- Also here: instead of fixed cut, use NN output as sampling probability
- makes defining loss easy, e.g. number of true positives $= \sum_{i \in \text{true events}} \text{output}_i$

  $\rightarrow$ "NN decides which events to select"
- Minimize inverse speedup
  $\rightarrow$ achieved speedup of 5.6
  (compared to ≈ 7.5 with original method of training with BCE and optimize cut)
  $\rightarrow$ investigate more

Intermezzo: Filtering at ATLAS

"Classical filtering"

Compose each sample into a certain set of orthogonal "slices" with respective filter efficiencies ϵ_{filter}. If the filter of one samples lets N events pass through, the corresponding weight for each event in that sample is

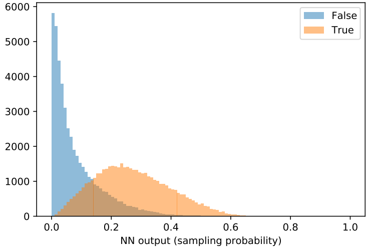
$$w = \frac{\sigma_{\text{sample}} \cdot \epsilon_{\text{filter}}}{N} \int L \mathrm{d}t$$

That is equivalent to having several slices where each of them has a probability p to let an event through which consequently means to scale the events of that slice up by a factor of $\frac{1}{p}$.

This can be generalised to the continuous case where i assign each event a probability p (e.g. based on the NN output) to let it through and afterwards weight it up by a factor of $\frac{1}{p}$. The optimization problem: How to find the best assignment of p to the events?

First try 2: Optimize for lowest effective uncertainty

NN output:



- Define loss by "effective uncertainty in skim (true) selection":
  - Effective sample size: $\frac{(\sum w_i)^2}{\sum w_i^2}$
    (unweighted sample size that would have the same relative uncertainty)
  - Define weights by inverse NN output
  - Scale up effective sample size by inverse filter efficiency
    "i could have simulated this factor of more events"
  - Currently neglecting finite processing time for event generation + evaluation of NN
- First try: Decrease relative uncertainty by 30% (or: effective speedup factor 2)

Advantage of that method: No Bias