Machine Learning and MC Generators

Manuel Szewc
25/06/2025

So far...

A big picture talk about Machine Learning, what it is and what types of problem we apply.

Slightly more involved discussion of Regression and Classification.

Now we can do the Regression, Classification and Unsupervised tutorials.

Next...

Here and in the tutorials we've seen "small" models. This is not only pedagogical: **baseline models are vital**. I don't need to overdo things!

But now, let's check the big guns: tree-based models and neural networks.

Tree-based models

A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space. Tree-based models combine **ensembles of decision trees** in various ways, such as bagging (RandomForests) and boosting (GradientBoosting).

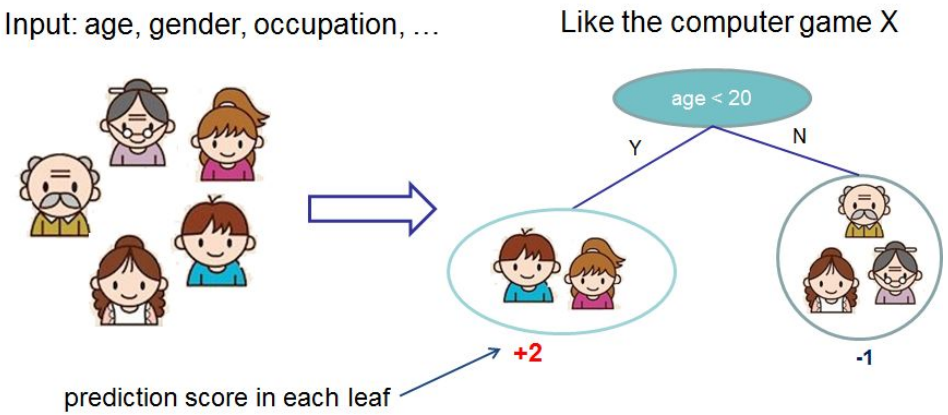


Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models

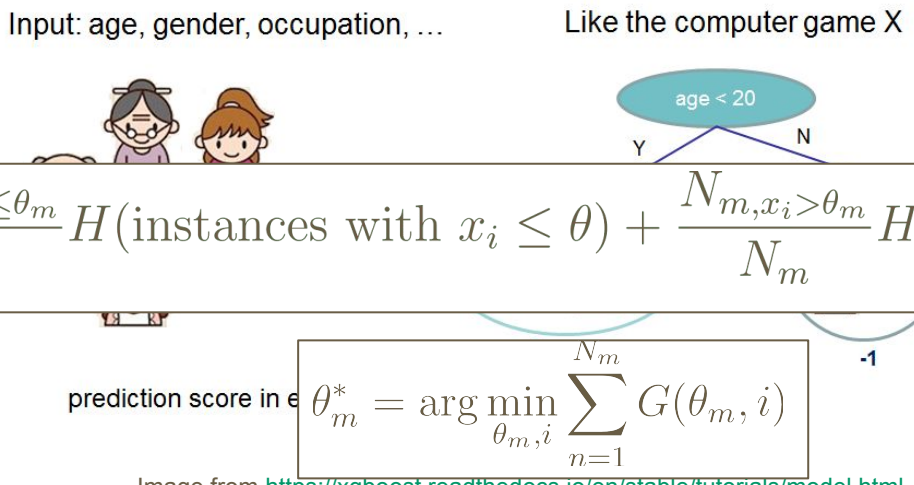
A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space. Tree-based models combine **ensembles of decision trees** in various ways, such as bagging (RandomForests) and boosting (GradientBoosting).

They are incredibly powerful and easy to tune. In many cases they might be more convenient than Neural Networks.

Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models

A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space.



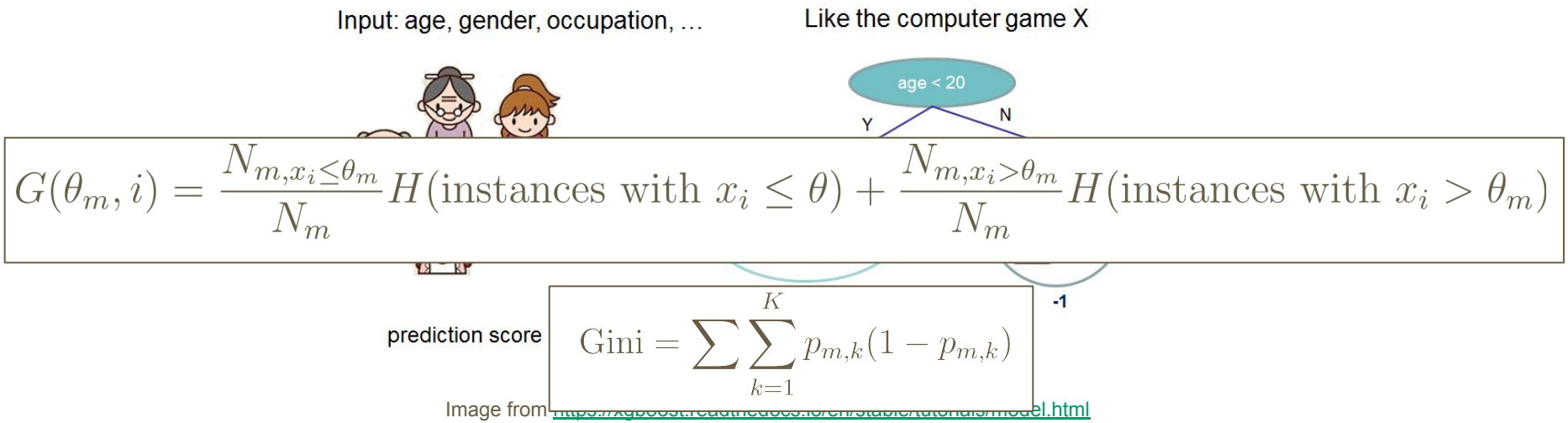
$$G(\theta_m, i) = \frac{N_{m,x_i \leq \theta_m}}{N_m} H(\text{instances with } x_i \leq \theta) + \frac{N_{m,x_i > \theta_m}}{N_m} H(\text{instances with } x_i > \theta_m)$$

$$\theta_m^* = \arg\min_{\theta_m, i} \sum_{n=1}^{N_m} G(\theta_m, i)$$

Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models

A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space.



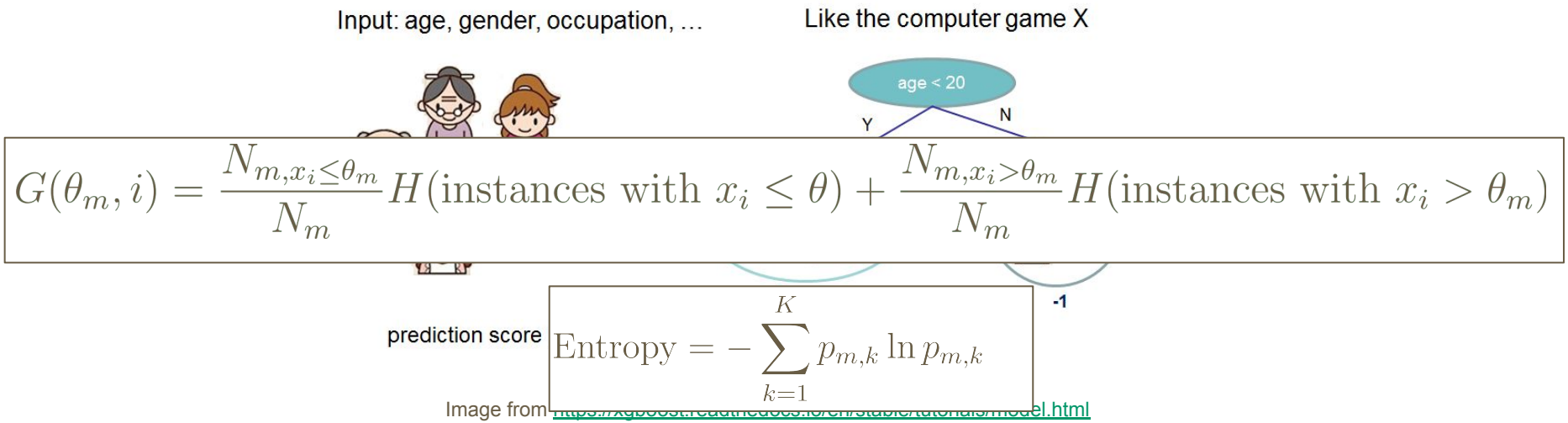
$$G(\theta_m, i) = \frac{N_{m,x_i \leq \theta_m}}{N_m} H(\text{instances with } x_i \leq \theta) + \frac{N_{m,x_i > \theta_m}}{N_m} H(\text{instances with } x_i > \theta_m)$$

$$\text{Gini} = \sum_{k=1}^{K} \sum p_{m,k}(1 - p_{m,k})$$

Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models

A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space.



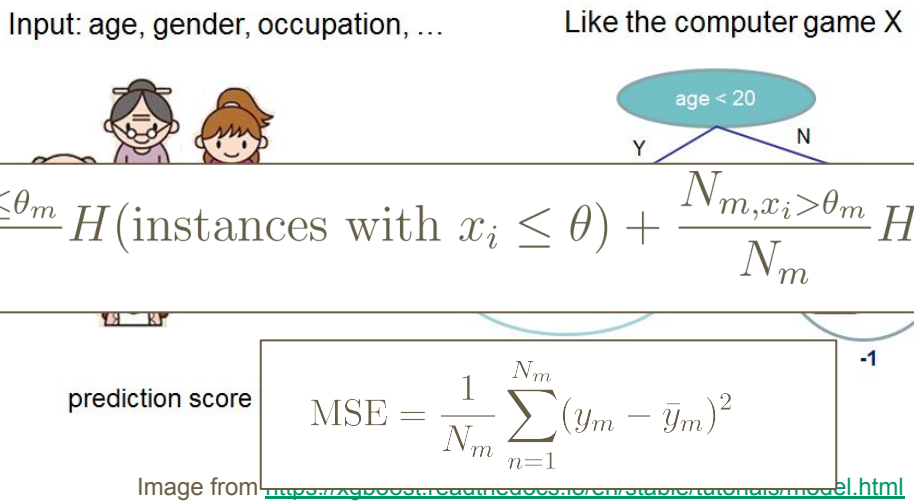
$$G(\theta_m, i) = \frac{N_{m,x_i \leq \theta_m}}{N_m} H(\text{instances with } x_i \leq \theta) + \frac{N_{m,x_i > \theta_m}}{N_m} H(\text{instances with } x_i > \theta_m)$$

$$\text{Entropy} = -\sum_{k=1}^{K} p_{m,k} \ln p_{m,k}$$

Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models

A decision tree is an algorithm that greedily optimizes a given task by performing cuts on the feature space.



$$G(\theta_m, i) = \frac{N_{m,x_i \leq \theta_m}}{N_m} H(\text{instances with } x_i \leq \theta) + \frac{N_{m,x_i > \theta_m}}{N_m} H(\text{instances with } x_i > \theta_m)$$

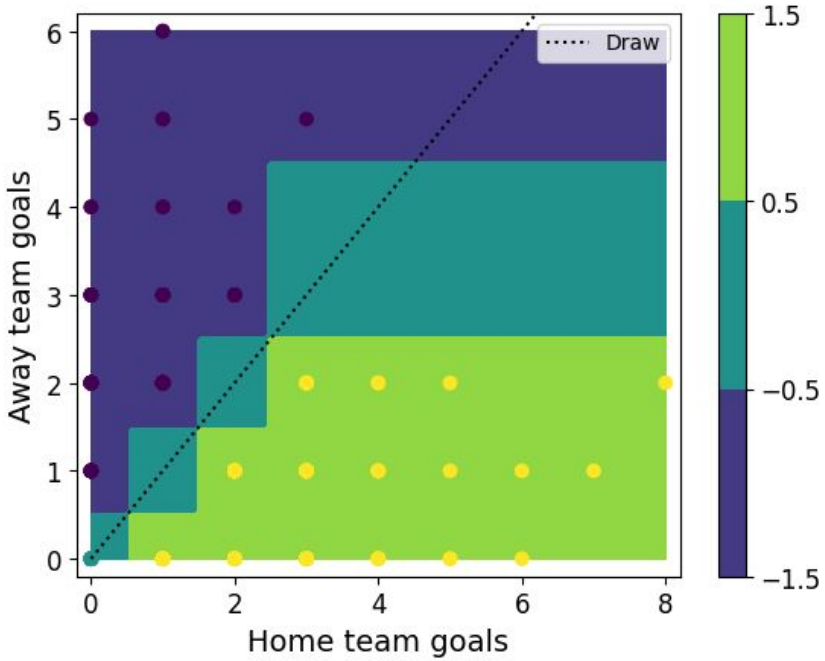
$$\text{MSE} = \frac{1}{N_m} \sum_{n=1}^{N_m} (y_m - \bar{y}_m)^2$$

Image from https://xgboost.readthedocs.io/en/stable/tutorials/model.html

Tree-based models and ensemble methods

Decision trees, when deep enough, are low-bias, high-variance methods → Very flexible, but prone to overfitting

However, because of this they are very amenable to **ensemble methods**

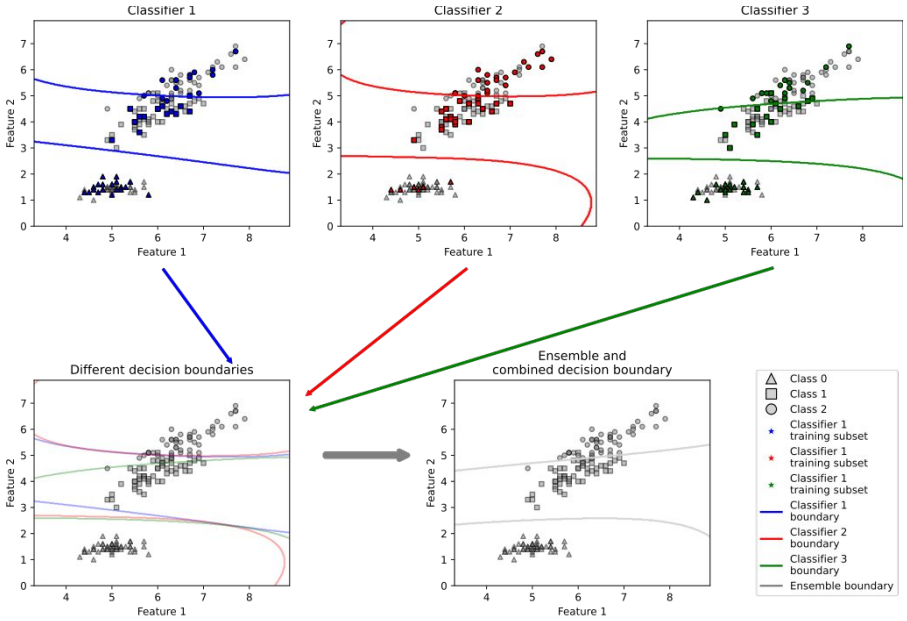


Ensemble methods

Ways to combine different estimators (they don't have to be from the same family!)

- Soft/Hard Voting
- Bagging/Bootstrapping, Pasting
- Boosting
- Stacking

Image from https://commons.wikimedia.org/wiki/File:Combining_multiple_classifiers.svg



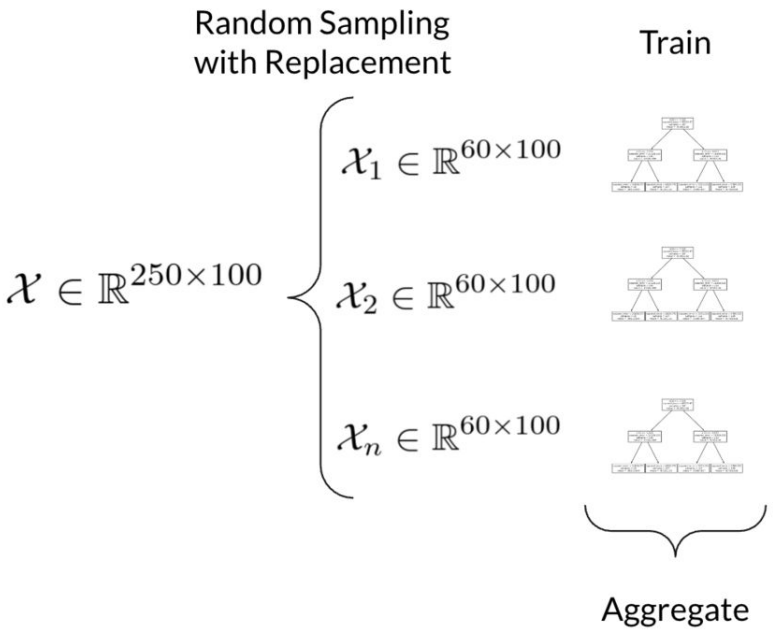
RandomForests

RandomForests are an example of a **bagging algorithm**.

I create a number of **bootstrapped datasets** and train DTs on each one of them (I also select a subset of features at random per split)

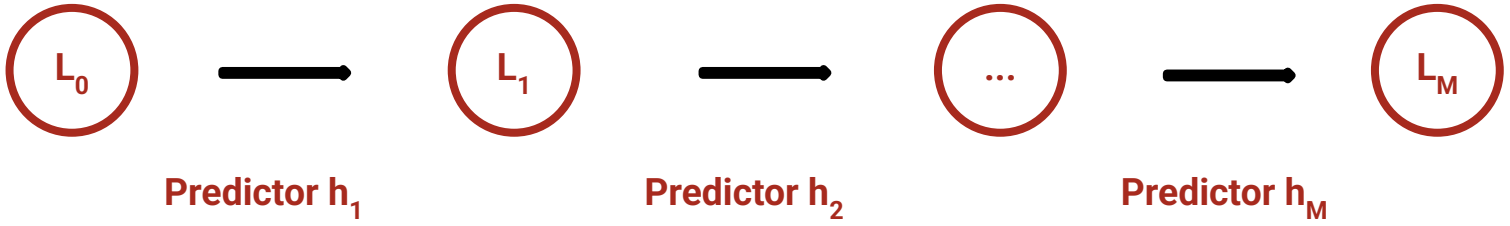
The overall estimator is the average of all DTs

From https://commons.wikimedia.org/wiki/File:Random_Forest_Bagging_Illustration.png



Boosted Decision Trees

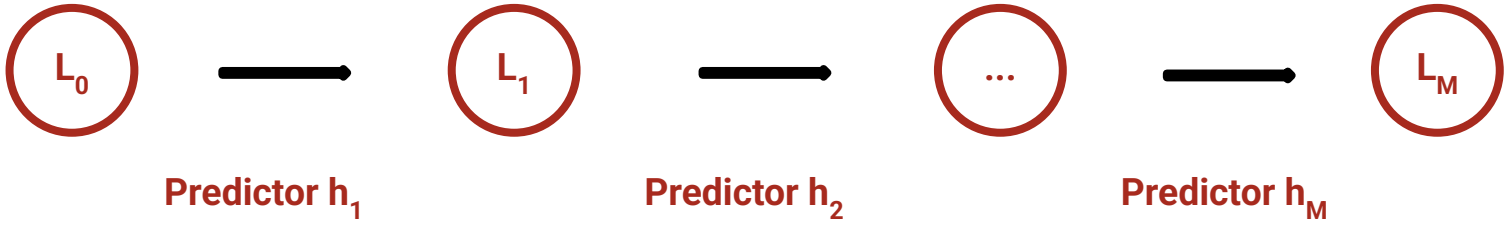
Boosting is a really powerful sequential framework that combines estimators, each one correcting in some way the previous one.



$$y(x_i)=F(h_1(x_i),h_2(x_2),...,h_M(x_M))$$

Boosted Decision Trees

The most popular boosting algorithms are AdaBoosting and GradientBoosting.



$$y(x_i)=F(h_1(x_i),h_2(x_2),\ldots,h_M(x_M))$$

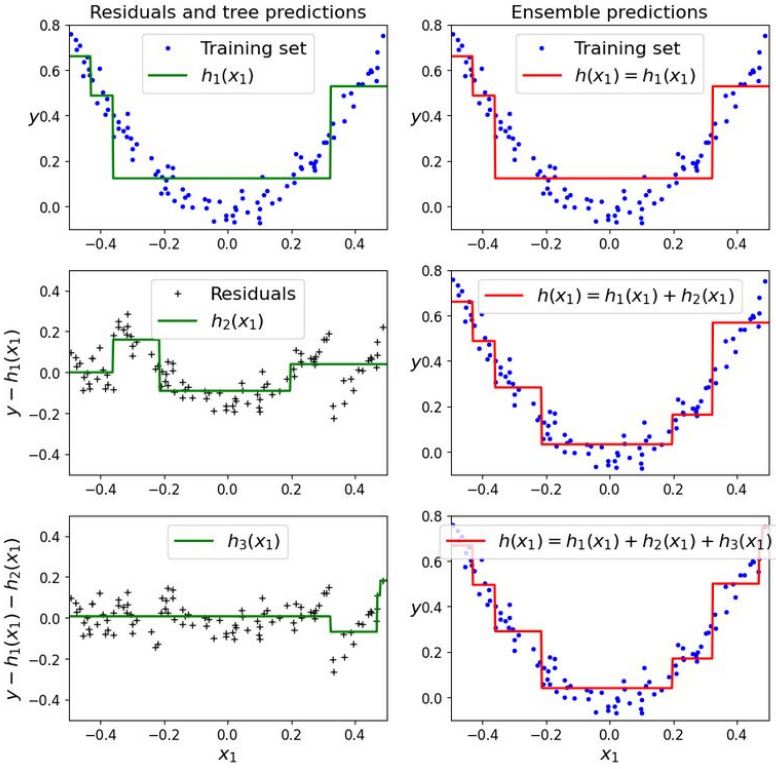
Some intuition

Boosting is great for combining **weak predictors**.

It’s not so easily parallelizable as Bagging/Pasting

It’s greedy. Each step wants to be as good as possible. No global strategy.

It’s implemented in state of the art libraries.



Deep learning

The reason all the fuss is happening. The power of Deep Learning lies in its adaptability and expressivity. These huge models are able to capture non-linear problems with surprising effectiveness.

At the core of the deep learning revolution lie **hardware development**, **big data collection** and **backpropagation**, which renders the loss function minimization possible.

Feed-forward or neural network (NNs)

The classic. A function that takes N numbers and returns K outputs, with the hidden layers and non-linear activation functions providing the power.

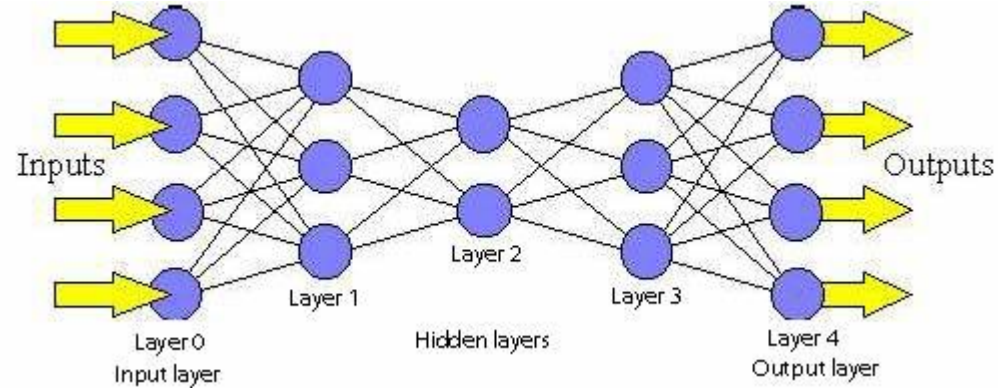


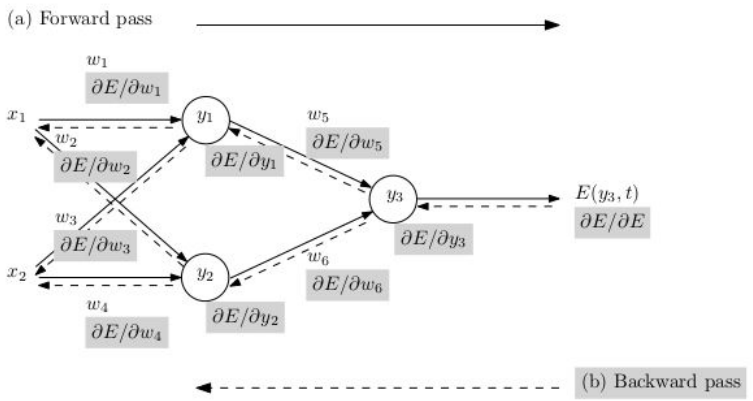
Image from
https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html

Autodifferentiation

Roughly, autodifferentiation (for automatic differentiation if you're not into the whole brevity thing) is the **automatic tracking of derivatives of operations throughout a computation.**

This can be done **forward** (as we go along the operations, we store the gradient) **or backward** (we build a graph and then go back assigning gradients at each step). Autodifferentiation engines are **useful beyond machine learning**, but are **vital for deep learning**.

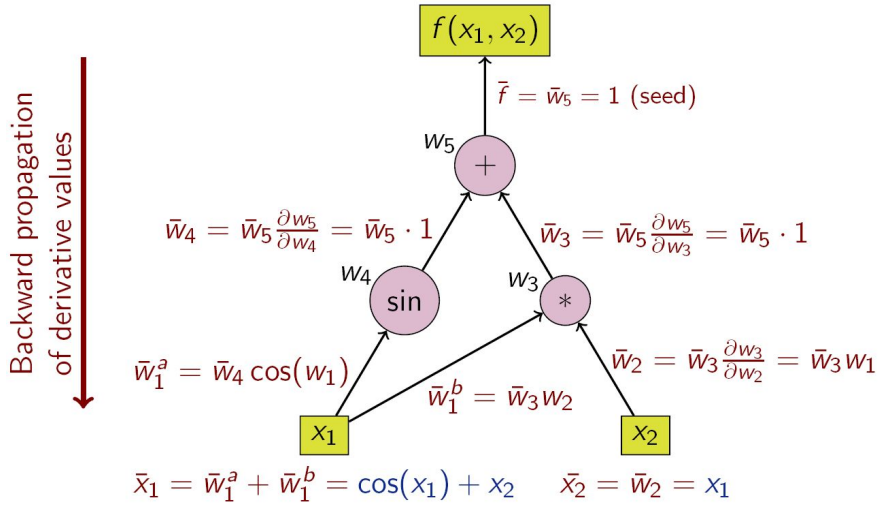
From https://arxiv.org/abs/1502.05767



Backpropagation

How (most) deep learning algorithms compute gradients. We start from the **output layer** and make our way back through the **chain rule** until we reach the **input layer.**

From https://en.wikipedia.org/wiki/File:ReverseaccumulationAD.png



Helping the Universal Approximation thingy

Neural Networks are Universal Function Approximators. We talk more about this in the tutorials.

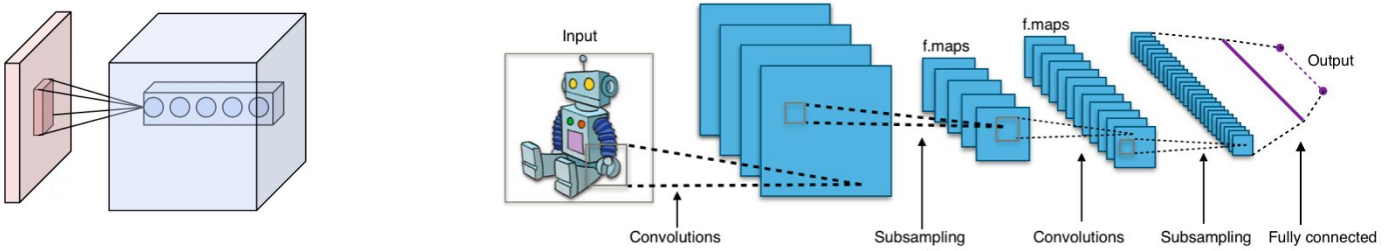
One thing, however, is that we find better performance for finite networks and datasets if we help the neural network in a variety of ways (like initialization, loss function optimizers, etc).

A very important way is to embed the symmetries of the problem into the network architecture. This usually gives us better bang for our buck (in terms of number of parameters, training cost, etc). This is also related to **representation learning**.

Convolutional Neural Networks (CNNs)

The DL revolution started in earnest with Alex-Net, a CNN.

To better deal with images, the learnable parameters are **filters** which are **convoluted** along the specified dimensions.**Translation invariance** for enhanced pattern location



Images from https://en.wikipedia.org/wiki/Convolutional_neural_network

Graph Neural Networks (GNNs)

A more general representation of data. The Neural networks are applied over **graphs** defined by **nodes, features and edges** with a process called **Message passing.** The Neural Network updates each node by looking at its relation to its neighbors and learns an **embedding** of the graph to be used **downstream**.

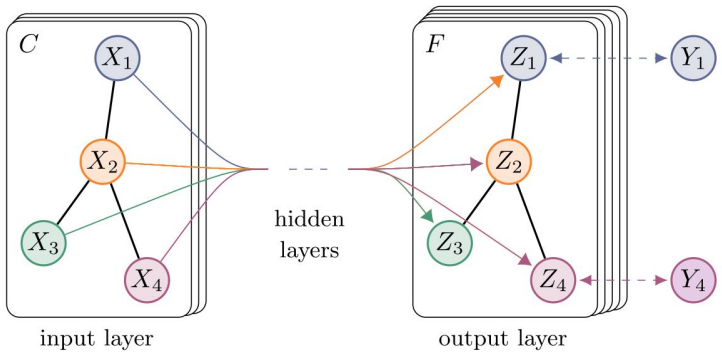
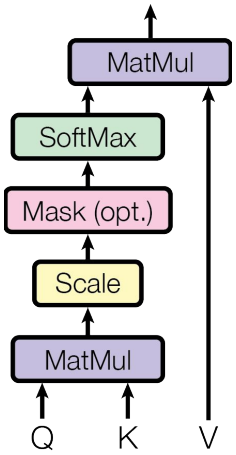
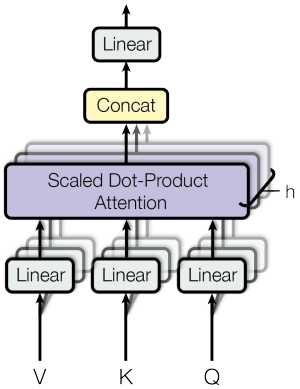


Image from https://arxiv.org/abs/1609.02907

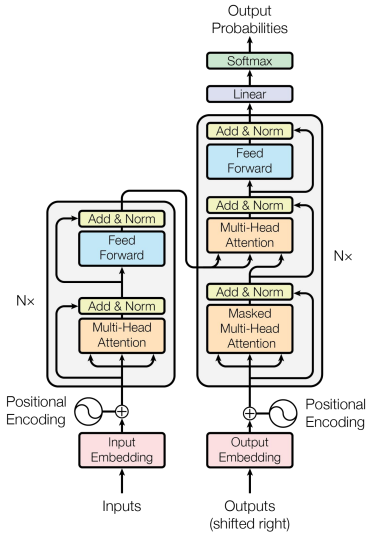
The Attention Mechanism and the Transformer

The Attention mechanism is a more efficient way to **embed an input with context**.





The **transformer** leverages the attention mechanism to optimize the necessary task (embedding, generation, translation, etc...)



Generative models

We have seen Regression and Classification problems in detail. In the tutorials, we also have examples of unsupervised tasks, including density estimation. A related task is **sample generation**.

This is very useful in a variety of ways. e.g. cheaper sample generator, density estimation, learned representations for downstream tasks...

Generative Adversarial Networks (GANs)

More of a philosophy of training for **generative learning** than a specific choice of architecture. We train **two networks, a Generator and a Discriminator** which "fight" each other.

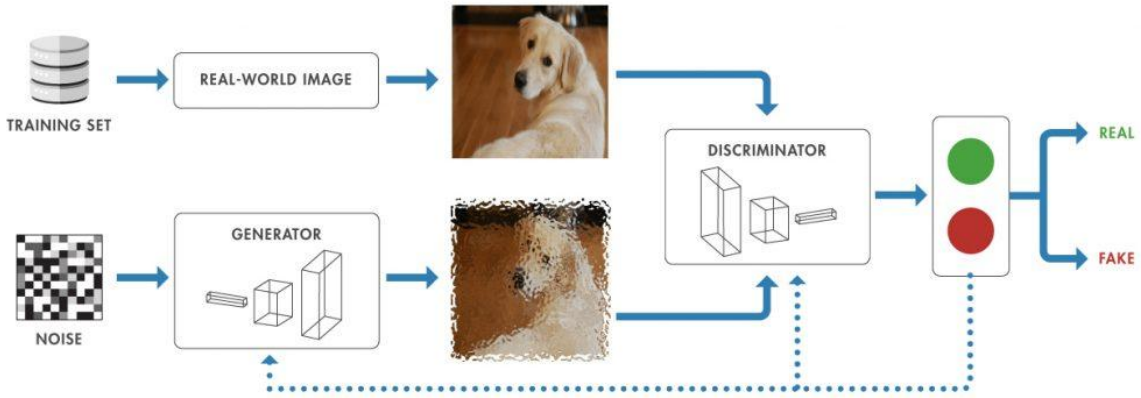


Image from https://blogs.mathworks.com/deep-learning/2021/12/02/synthetic-image-generation-using-gans/

Normalizing Flows (NFs)

A **generative model** that learns how **the data density relates to a simpler base distribution**. We learn the parameters of chosen invertible functions that transform samples from the base distribution to the data distribution.

Less flexible but easier to train + access to the exact likelihood

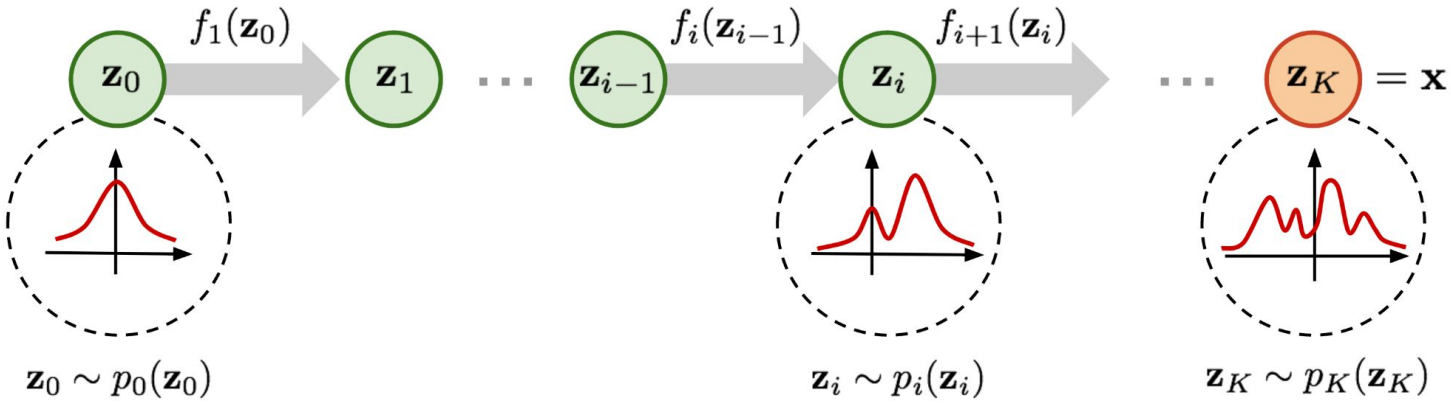
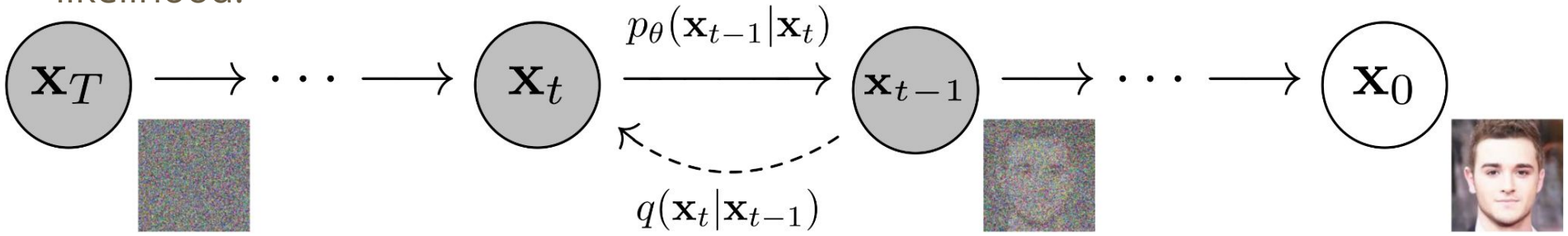


Image from https://lilianweng.github.io/posts/2018-10-13-flow-models/

Score-based and Denoising Diffusion models

Similar to NFs, they relate the distribution of interest to a simple base distribution by **adding noise** and learn how to **invert this process**.

More expensive but more flexible than NFs. They have been shown to produce **the best samples** in several datasets, retaining access to the likelihood.



Most of the state of the art is generative!

As you see, the recent developments have been all about how to get better generative models (with prompts for text / image / stuff generation).

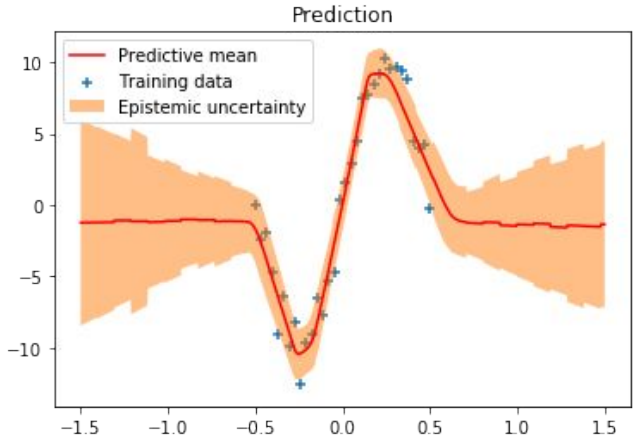
This is awfully convenient for us as well!

Uncertainty estimation in NNs

We have discussed how to define a model that yields certain predictions. However, these models have been deterministic (even if the task is probabilistic such as sampling). Here is a nice review.

For the cases of interest here, the two main methods are **Ensembles** and **Bayesian Neural Networks**.

Image from
http://krasserm.github.io/2019/03/14/bayesian-neural-networks/



Some wisdom regarding ML problems

Most of the work goes into **data collection** and more importantly **problem specification**. The problem selects the algorithms, not viceversa. This is specially true because most state-of-the-art problems are data-intensive and computationally expensive **(training is an art, not a science...)**.

It's always better to start with "basic" algorithms even if only as a baseline against which to compare other models.

Hyperparameters are important and **overfitting** is really deceiving (underfitting can also happen!). Be sure to validate on **unseen data** (cross-validation is a life saver, but expensive), and always evaluate your final models on **more unseen data** (if possible).

ML4MC

ML for MC

ML is useful in general and for a lot of High Energy Physics applications (Living review https://iml-wg.github.io/HEPML-LivingReview/, Snowmass https://inspirehep.net/files/1d860552406c3700aaf4598c7054137f) but what are its specific uses regarding Monte Carlo generators?

A nice review: https://arxiv.org/pdf/2203.07460

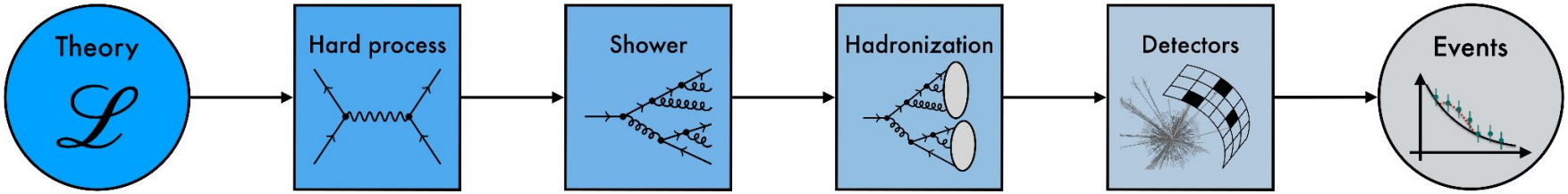


Image by Ramon Winterhalder

ML for phase space sampling

To sample from hard processes f(x)~$|M|^2$ composed of several Feynman diagrams, we need to efficiently sample over the possible phase space.

The **complexity** of f(x) and the **sheer dimensionality of the phase space** make **efficient** sampling and integration challenging.

Current trick is to define **channels** α_i and sample N_i events per-channel, using a **sampling function** $g_i(x)$.

$$I[f] = \int d^D x\, f(x) \quad x \in \mathbb{R}^D$$

$$I[f] = \sum_i I_i \approx \sum_i \frac{1}{N_i} \sum_k \frac{\alpha_i(x_k) f(x_k)}{g_i(x_k)}$$

$$\frac{\sigma^2}{N} = \sum_i \frac{\sigma_i^2}{N_i} \approx \sum_i \frac{1}{N_i} \left(\mathbb{E}[\left(\frac{\alpha_i(x) f(x)}{g_i(x)} \right)^2] - \left(\mathbb{E}[\frac{\alpha_i(x) f(x)}{g_i(x)}] \right)^2 \right)$$

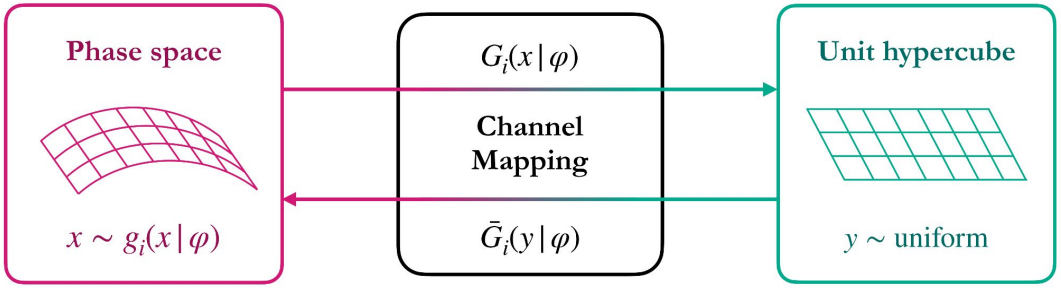
MadNIS (Madgraph-ready Neural Networks for Multi-Channel Importance Sampling)

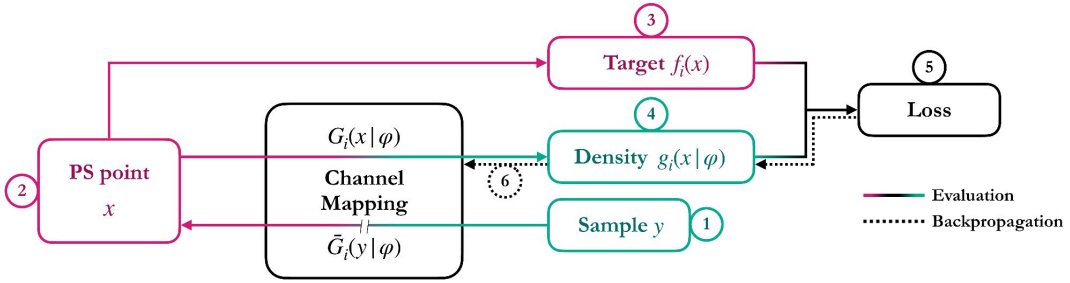
Ultra-fast event generation by promoting the channel weights α and the channel densities g_i to neural networks.

α is a **feed-forward neural network** and g_i is a **normalizing flow**.

Parametric density estimation that combines **online training** and **offline training** to ensure speed and precision.

The goal is to **minimize the variance of the integral estimation** with **smart initialization**, **stratified sampling strategies** and **channel dropping.**





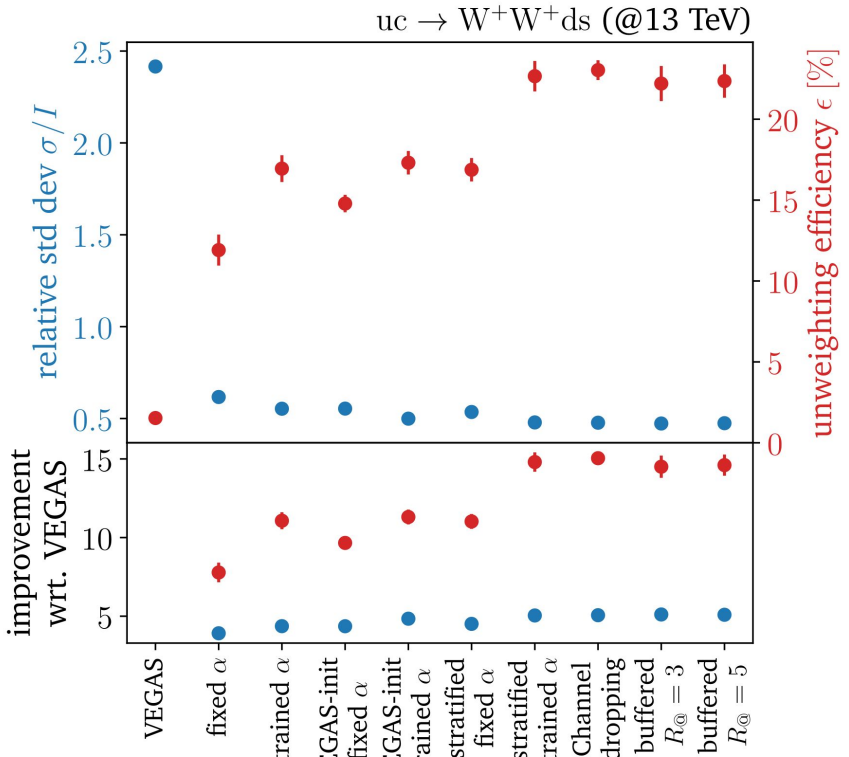
See also 2001.05478 and 2001.05478 and others for similar works

MadNIS

The relevant metrics here are the **variance of the cross-section estimate** (which is also the training loss!) and the **unweighting efficiency**.

Errors are obtained by **ensembling**.

They observe huge improvements with respect to VEGAS in several channels, with tt+jets still challenging.



ML for matrix elements

MEs are challenging to compute. Interpolation techniques are usually implemented to reduce computation time.

To replace exact ME (or integrands) with NN-based surrogate models, precision and speed are **fundamental**.

They need to be trained with **relatively small sample size** to be useful.

See e.g.:

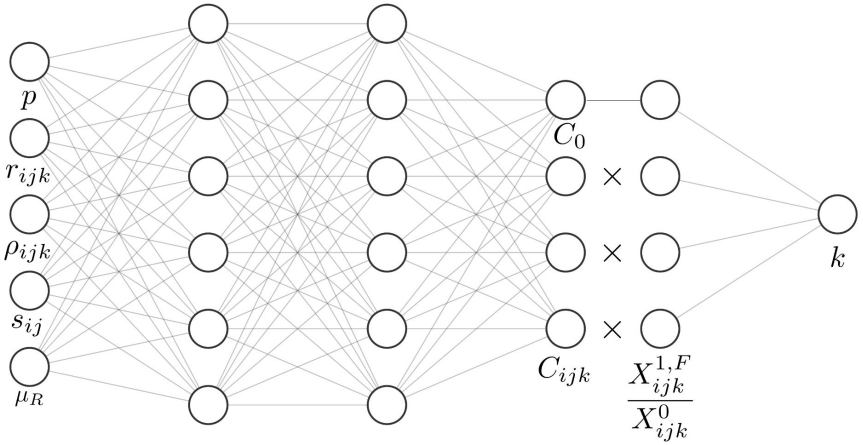
- 2302.04005, a standard **feed-forward neural networks** helped by a **smart parameterization**
- L-GATr, a **Lorentz-Equivariant transformer** without any explicit factorization assumptions
- SYMBA, **transformers** to perform **symbolic regression**

One-loop ME emulation with factorisation awareness

Parameterized in terms of their k-factor with respect to the tree level ME (much cheaper to compute)

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

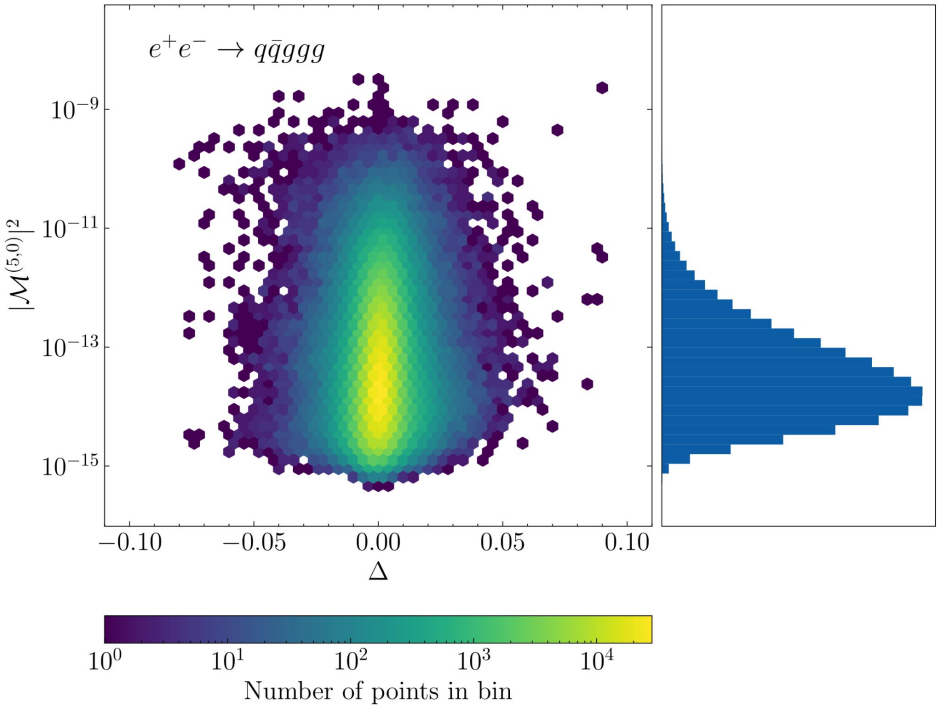
where X_{ijk} are the finite-substracted antenna functions. **C_0 and C_{ijk} are the outputs of a feed-forward neural network.**



One-loop ME emulation with factorisation awareness

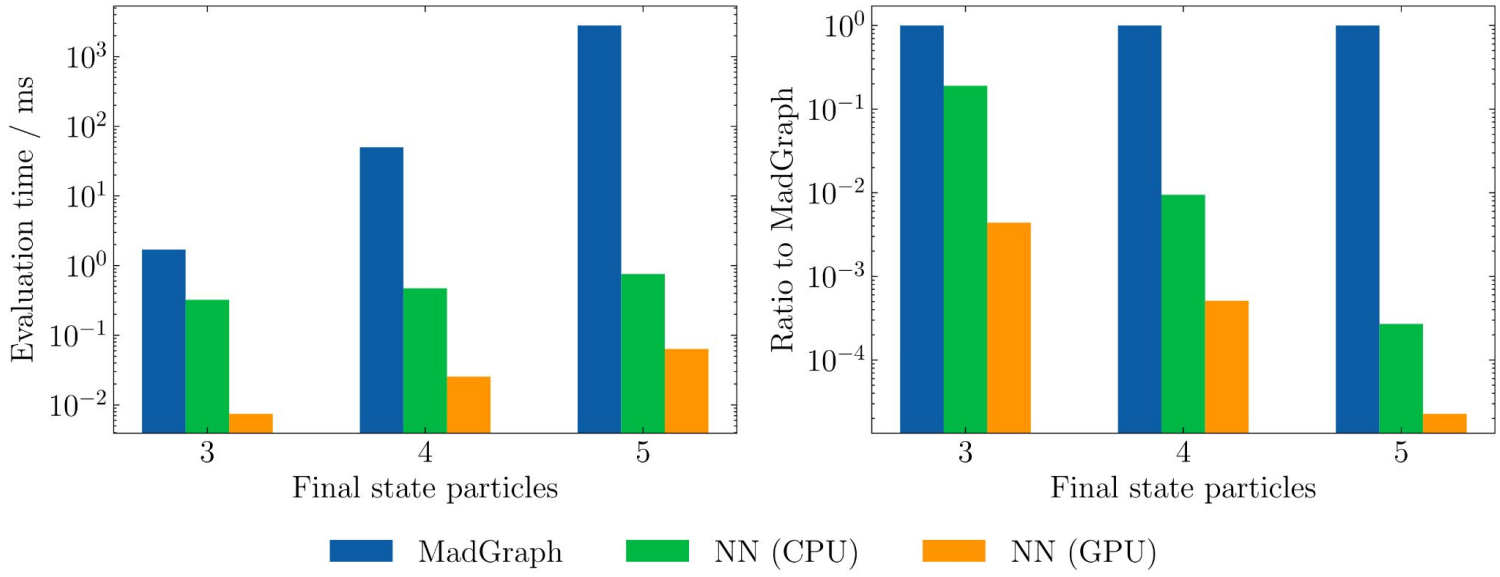
A **weighted MAE loss** gives more importance to the rarer regions of the phase space. The samples consist of 80k train, 20k validation, 1M test → reflects the necessary imbalance.

They also use the **MSE** for the full MEs as a metric, ensembling for uncertainties.



One-loop ME emulation with factorisation awareness

Implementing their model in **ONNX**, they find an improvement in speed.



ML for PDFs (and FFs)

Parton Distribution Functions and Fragmentation Functions are **non-perturbative objects** with well-known theoretical properties (scale evolution, their convolution with the hard process, etc).

PDF determination can be translated to fitting a **Monte Carlo ensemble of fitted functions**, where each replica is obtained by sampling a different dataset with the covariance matrix.

A **probability distribution over functions** that captures how the different theoretical and experimental uncertainties affect the PDF determination.

NNPDF

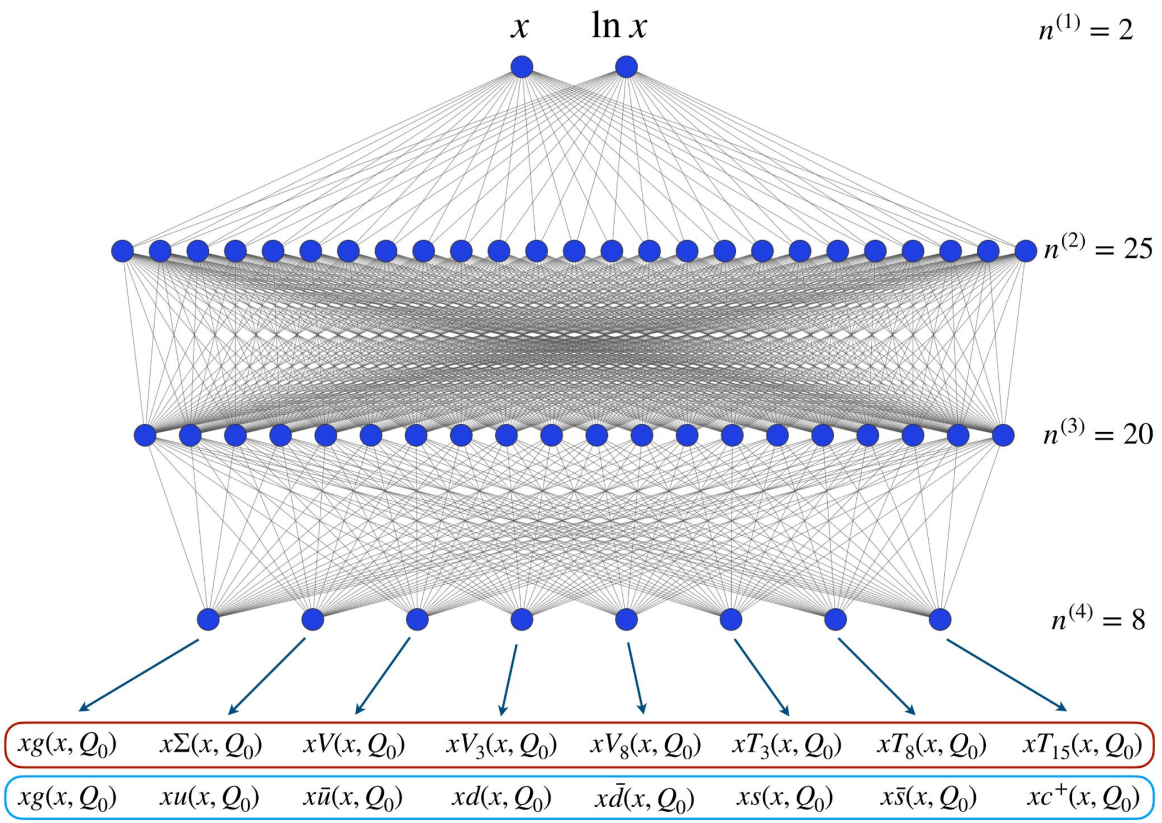
NNPDF4.0 is “the first PDF set to be based on a methodology fully selected through a machine learning algorithm.”: every choice (ML architecture and optimization function) is selected via **automated hyperoptimization**.

NNPDF uses a **feed forward NN** with K outputs that are combined with a preprocessing factor and normalization constants

$$x f_k\left(x, Q_0; \boldsymbol{\theta}\right) = A_k\, x^{1-\alpha_k} (1-x)^{\beta_k} \mathrm{NN}_k(x; \boldsymbol{\theta})\,, \quad k = 1, \ldots, 8\,,$$

where k runs over a specific choice of basis (in NNPDF4.0, evolution or flavor) at a given scale Q_0

NNPDF


x
$\ln x$
$n^{(1)} = 2$
$n^{(2)} = 25$
$n^{(3)} = 20$
$n^{(4)} = 8$
$xg(x, Q_0)$ $x\Sigma(x, Q_0)$ $xV(x, Q_0)$ $xV_3(x, Q_0)$ $xV_8(x, Q_0)$ $xT_3(x, Q_0)$ $xT_8(x, Q_0)$ $xT_{15}(x, Q_0)$
$xg(x, Q_0)$ $xu(x, Q_0)$ $x\bar{u}(x, Q_0)$ $xd(x, Q_0)$ $x\bar{d}(x, Q_0)$ $xs(x, Q_0)$ $x\bar{s}(x, Q_0)$ $xc^+(x, Q_0)$

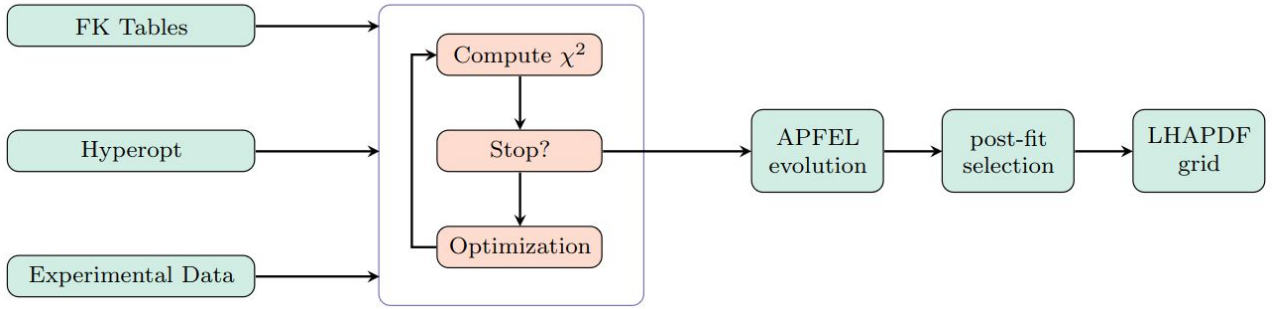
NNPDF

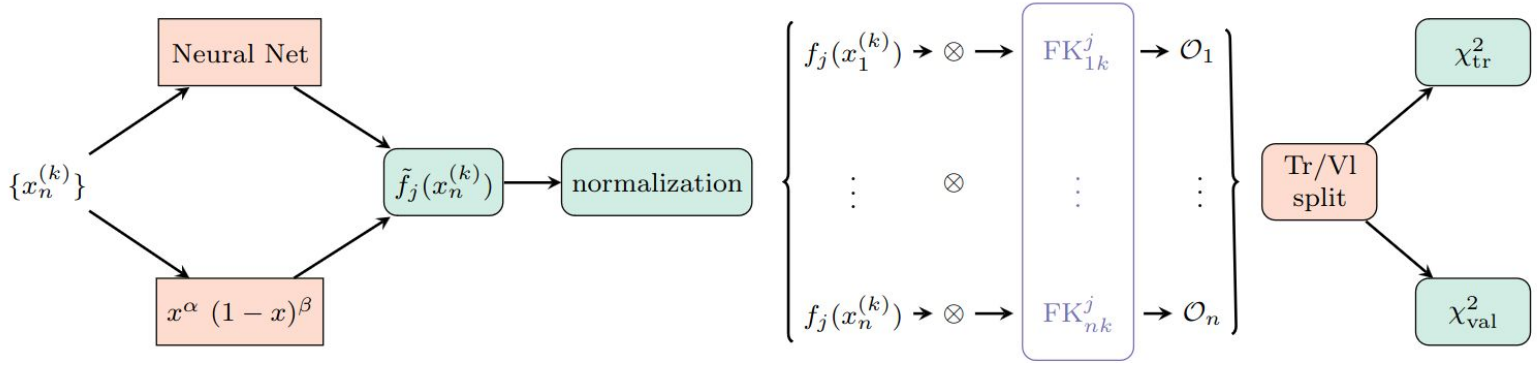
The PDFs must satisfy **physical constraints** (sum rules, positivity constraints and integrability). The first one is imposed during training through the A_k (**smart parameterization**) while the latter two are imposed via Lagrange multipliers in the loss function (**training optimization**).

The loss function is “basically” a **chi-squared** between the **data replica** and the **predicted data** given by the Neural Network. A lot of work actually goes into properly defining consistent datasets, replicas and a weighted loss-function.

NNPDF

The FK-tables are what makes all of this possible: a connection between PDF and observable that allows for **backpropagation**!



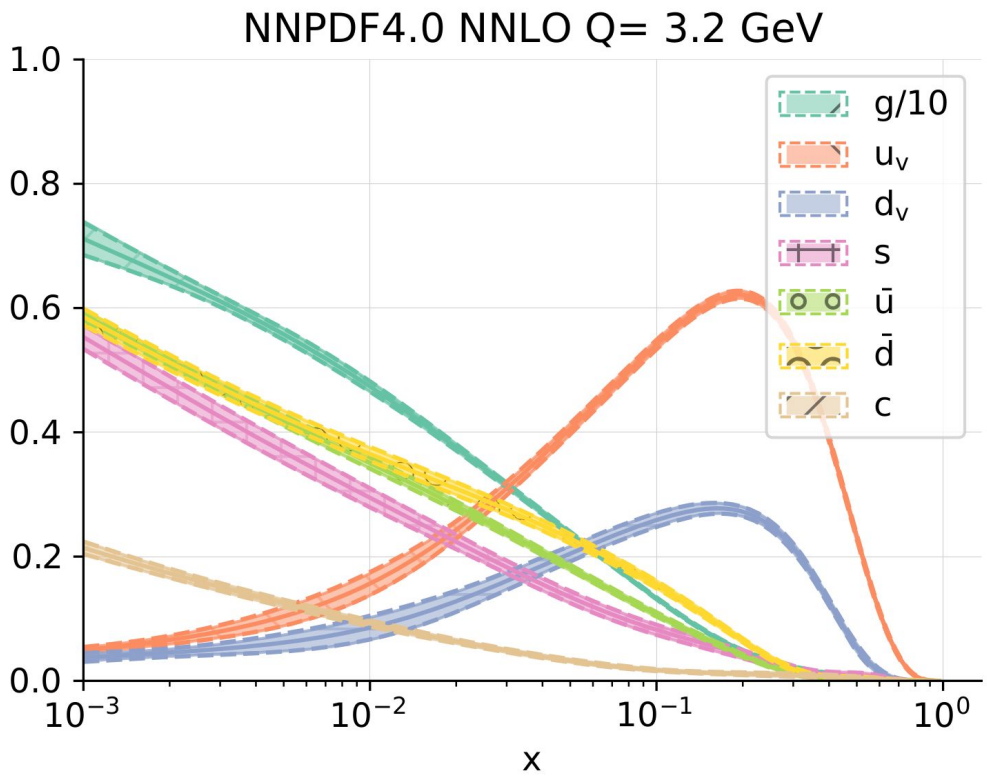


NNPDF

The resulting set of PDFs are implemented in an grid.

I highly recommend reading the validation procedure for the PDF sets.

However, challenges remain regarding the **extrapolation** of the fit into unseen regions.

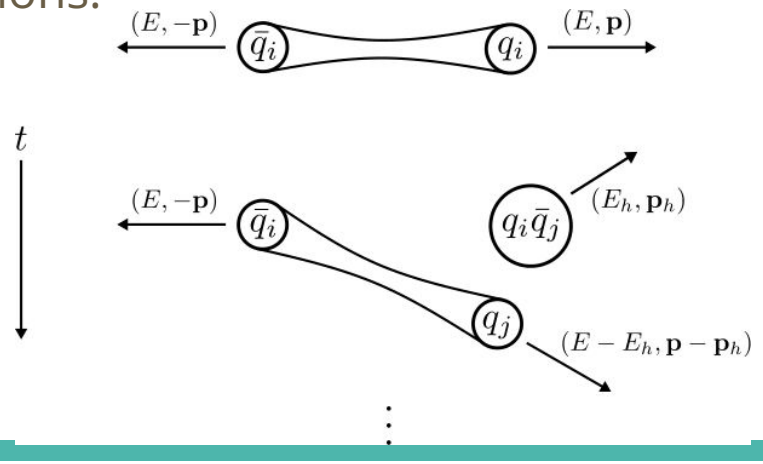


Hadronization

Hadronization is a **inherently non-perturbative process**. We rely on **empirical models** for predictions. It is a more complicated problem that PDFs and FFs because we do not have the differentiable convolution.

There are two main models: the **Lund String model** (Pythia) and the **Cluster model** (Herwig), that rely on different assumptions.

For example, the Lund String Model takes colored singlets and ~20 parameters to produce hadrons as string breaks.

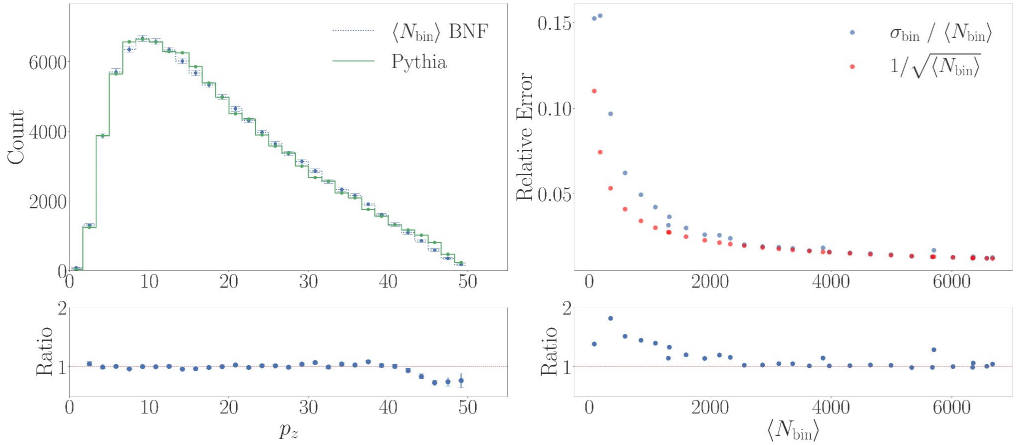


MLHAD

See also HADML using GANs!

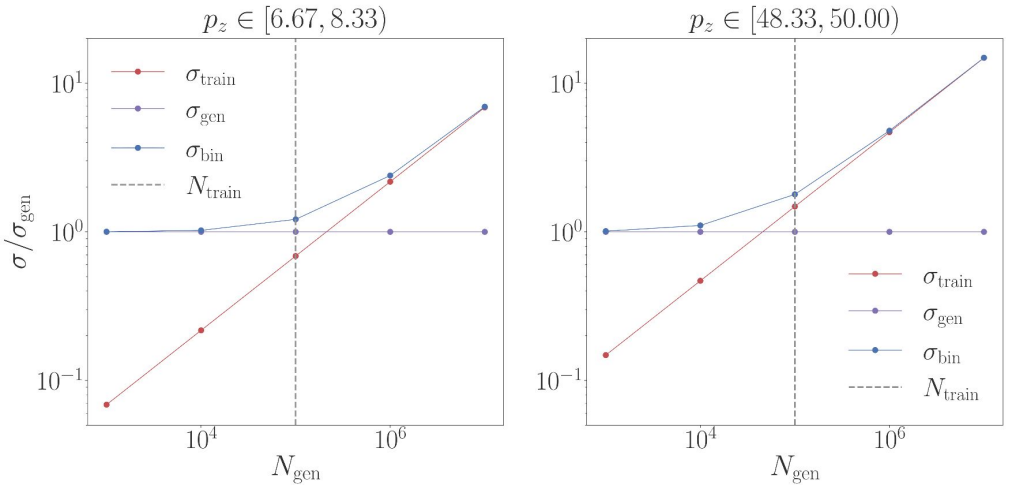
Treat hadronization as a generative process. As a first approximation, use NFs to learn the fragmentation density directly.

By promoting the Normalizing Flows to Bayesian Normalizing Flows, the model also has training uncertainties!



MLHAD

In particular, any generator can generate only so many events before the uncertainty on the model overcomes the statistical uncertainty of the samples.



Detector simulation

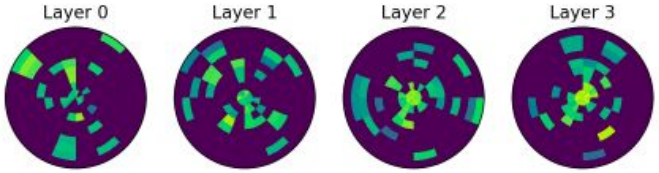
The most expensive stage of the simulation pipeline, both in **time** and in **storage**. Fast lightweight simulation with enough precision could be a potential life-saver!

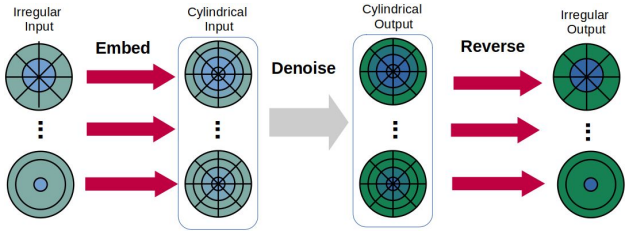
Lot of effort in **calorimeter shower emulation** using **surrogate models**. These include **VAEs**, **GANs**, **Normalizing Flows**, **Transformers**, **GNNs** and **Diffusion models**. Resources such as the **CaloChallenge** or **COCOA** allow to benchmark and compare different models.

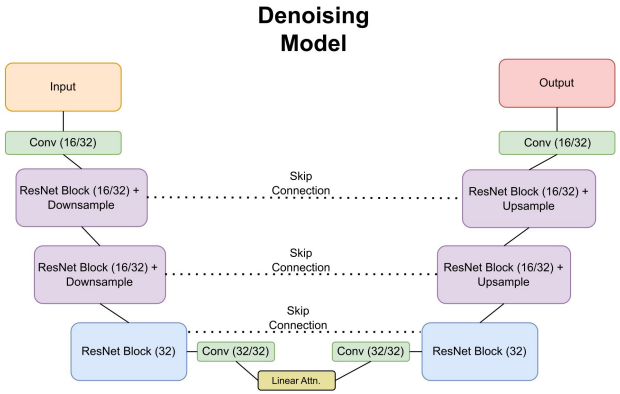
The saving in speed is great if the surrogate models use GPUs. **Precision** is still an **open challenge**. Surrogate models perform well in the bulk but have problems capturing the tails. This motivates hybrid surrogate model - GEANT4 scenarios.

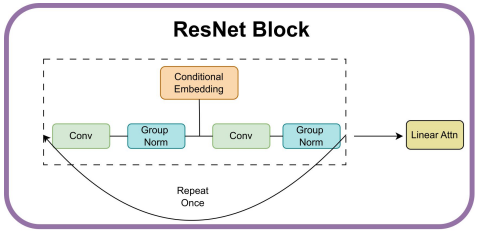
CaloDiffusion

A **denoising diffusion model** that takes explicit advantage of cylindrical symmetries

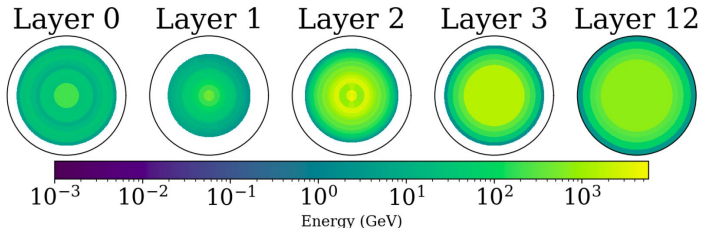








CaloDiffusion



Very good **speed** and **precision**

Dataset	Classifier AUC (low / high) `CaloDiffusion`	`CaloFlow`	`CaloScore v2`
1 (photons)	**0.62** / 0.62	0.70 / **0.55**	0.76 / 0.59
1 (pions)	**0.65** / **0.65**	0.78 / 0.70	- / -
2 (electrons)	**0.56** / **0.56**	0.80 / 0.80	0.60 / 0.62
3 (electrons)	**0.56** / **0.57**	0.91 / 0.95	0.67 / 0.85

TABLE I. The AUC values for a classifier trained to distinguish between `Geant4` and synthetic showers. The first value listed is the AUC for the classifier trained on low-level features and the second is the AUC for the classifier trained on high-level features. The `CaloDiffusion` values are the average of 5 independent classifier trainings. In all cases, the variation in scores was observed to be 0.01 or less. In each row, the bold value is the best AUC value for each classifier type.

Dataset	FPD	KPD
1 (photons)	0.014(1)	0.004(1)
1 (pions)	0.029(1)	0.004(1)
2 (electrons)	0.043(2)	0.0001(2)
3 (electrons)	0.031(2)	0.0001(1)

TABLE II. Additional metrics comparing the agreement between showers generated with `Geant4` and `CaloDiffusion`. The number in parentheses is the uncertainty in the last significant digit as evaluated with the `JETNET` library.

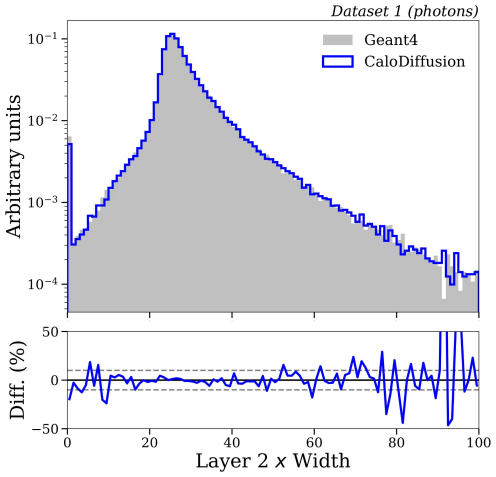
Results are based on a 2.6 GHz Intel E5-2650v2 "Ivy Bridge" 8-Core CPU and an NVIDIA V100 GPU. The time required to generate a shower in `Geant4` depends strongly on the incident energy of the particle. The average over the incident energies used in datasets 2 and 3 is $O(100\text{ s})$ [31].

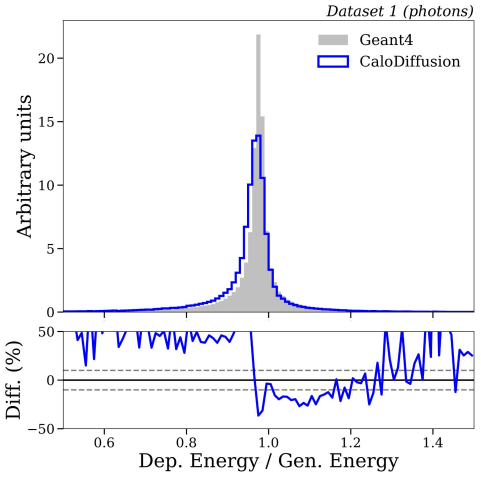
Dataset	Batch Size	Time/Shower [s] CPU	GPU
1 (photons)	1	9.4	6.3
(368 voxels)	10	2.0	0.6
	100	1.0	0.1
1 (pions)	1	9.8	6.4
(533 voxels)	10	2.0	0.6
	100	1.0	0.1
2 (electrons)	1	14.8	6.2
(6.5K voxels)	10	4.6	0.6
	100	4.0	0.2
3 (electrons)	1	52.7	7.1
(40.5K voxels)	10	44.1	2.6
	100	-	2.0

TABLE III. The shower generation time for `CaloDiffusion` on CPU and GPU for various batch sizes.

CaloDiffusion

Most of the explored variables are well-reproduced. However, **global features** and **outliers** are not.





ML to improve generators

Any generator, be it a Monte carlo simulator or a an end-to-end ML generative model, is an imperfect representation of data that needs to be **tuned**.

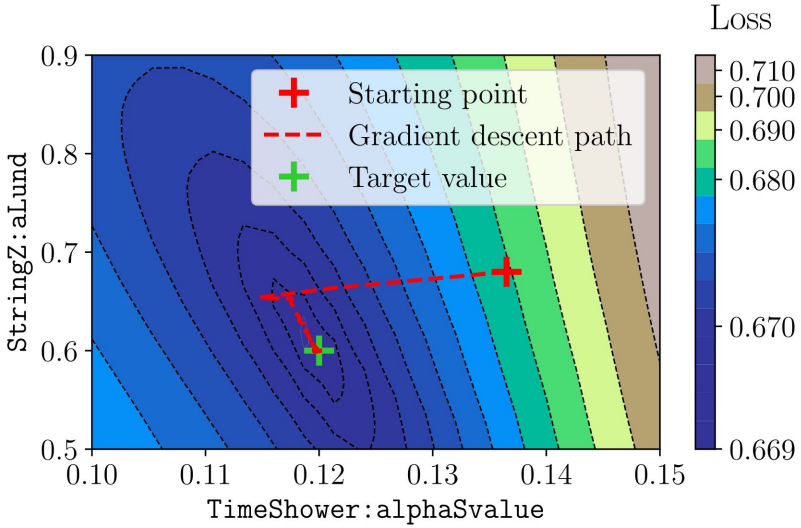
Easier said than done: optimization is very expensive. But ML can help!

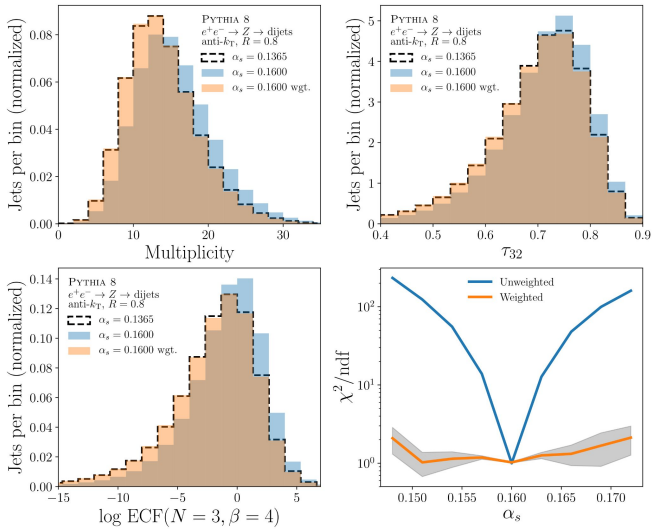
If parametric, **tuning** provides best fit values and hopefully uncertainties.

An alternative is to reweight your samples to better match the data in one region. This has the drawback of **reducing the statistical power of your samples**.

DCTR and DCTRGAN

Deep neural networks using **C**lassification for **T**uning and **R**eweighting. A **classifier** is trained to reweight between different parameter values/refine the GAN noise sampling.





DCTR and DCTRGAN

Deep neural networks using **C**lassification for **T**uning and **R**eweighting. A **classifier** is trained to reweight between different parameter values/refine the GAN noise sampling.

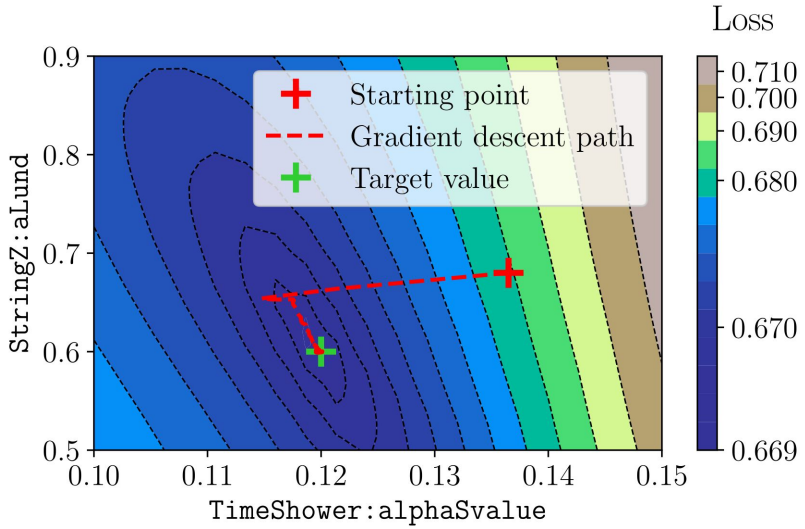


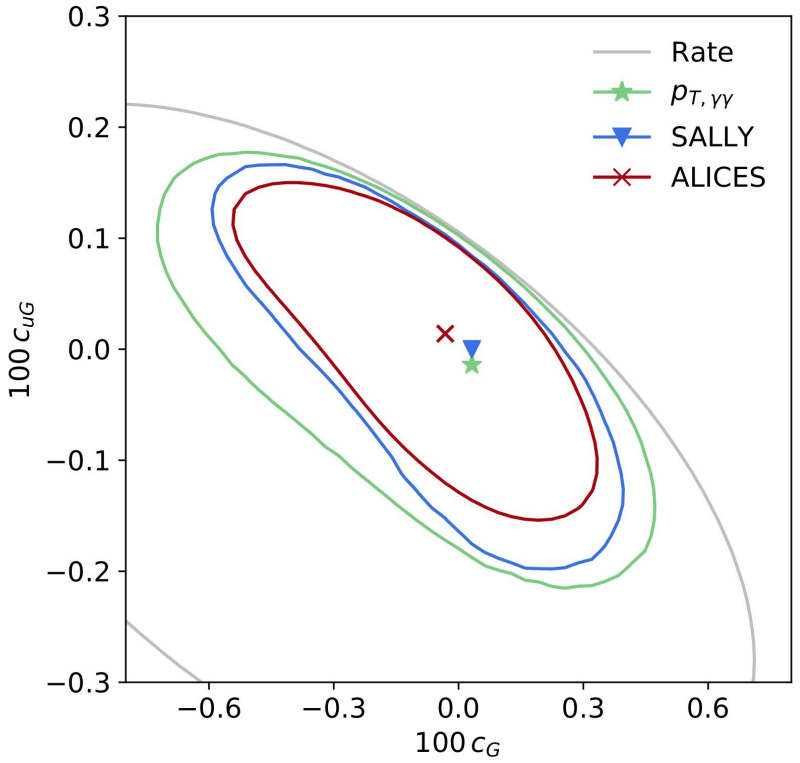
TABLE II. Simultaneous fit for three parameters. The top row shows the results for the validation fit where we knew the target parameters, and the bottom row is the blinded fit. The reported numbers are the mean and standard deviation over 20 runs with different model initializations.

	Parameter	Target value	Fit value
Val.	TimeShower:alphaSvalue	0.1200	0.1195 ± 0.0022
	StringZ:aLund	0.6000	0.6276 ± 0.0373
	StringFlav:probStoUD	0.1200	0.1203 ± 0.0071
Blinded	TimeShower:alphaSvalue	0.1700	0.1707 ± 0.0022
	StringZ:aLund	0.7500	0.7425 ± 0.0453
	StringFlav:probStoUD	0.1400	0.1422 ± 0.0065

Inference using ML and MC event generators

Likelihood-free or **simulation based inference** methods, such as MadMiner, take advantage of **forward models** to learn **optimal test statistics** from samples obtained by scanning over the relevant parameter space.

These methods are usually **computationally expensive** because of sample generation and training.

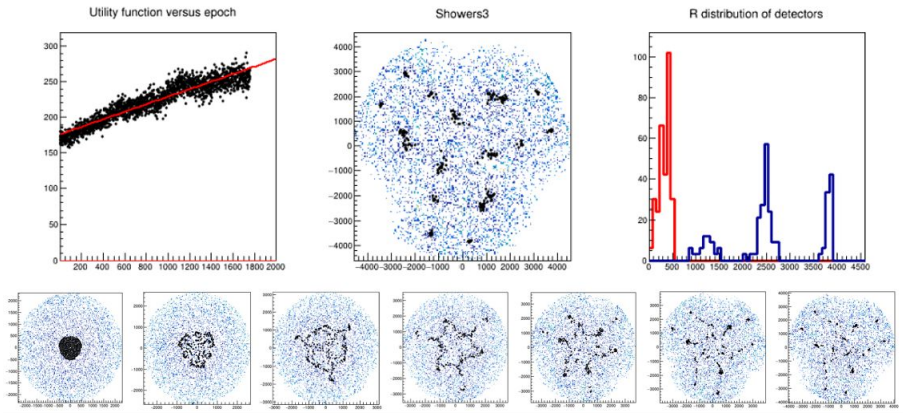


Differential programming

Surrogate models and event reweighting are useful tools to “add” a gradient to MC generators.

Another way of doing this would be to have fully differentiable generators which would allow to optimise even detector construction (see MODE collaboration)

Image from https://arxiv.org/abs/2310.01857



All in all…

ML has been successfully applied in MC relevant areas, mainly for **surrogate models**. However, many challenges remain. Specifically, the **trade-off between speed and precision**; trustworthiness of **uncertainties** and **extrapolation**.

A lesson that I take from these applications: **domain knowledge** is **vital**. The most important lessons regarding ML for MC **lie in the MC part**.