Physics and Machine Learning

Jan Pawlowski and Tilman Plehn*

Institut für Theoretische Physik, Universität Heidelberg, Germany

January 27, 2025

Abstract

Physics and modern machine learning have a huge number of connections, through their scientific goals, methods, applications, and theoretical understanding. These notes provide an introduction to modern machine learning, specifically for physicists. As prerequisites they rely on some results from bachelor-level physics lectures, but do not assume any prior knowledge of machine learning. If you are interested in a set of nice tutorials, check it out here.

*plehn@uni-heidelberg.de

1 Basics

1.1 Statistics

Bayes' Theorem Before we start with some basics of machine learning, let use remind ourselves of basics statistics from our first year of physics studies. A formal introduction into statistics starts with the three Kolmogorov Axioms. The first axiom states that a probability for a given outcome is a non-negative real number

$$p(A) \in \mathbf{R} \qquad \text{and} \qquad p(A) \geq 0 \; . \tag{1.1}$$

The second axiom states that the sum of probabilities for all possible outcomes of an experiment is one,

$$\sum_i p(A_i) = 1 \qquad \text{or} \qquad \int dA \; p(A) = 1 \; . \tag{1.2}$$

The third axiom states that probabilities for disjoint outcomes add,

$$p(A \cup B) = p(A) + p(B) \; . \tag{1.3}$$

The logical combination of these outcomes is an 'or'. Alternatively, we can ask the question what the joint probability of two measurements A and B is. Their independence is defined by

$$p(A \cap B) = p(A)p(B) \; . \tag{1.4}$$

If two measurements are not independent, we need to define conditional probabilities. This means we ask what the probability of A is under the condition that we also observed B,

$$p(A|B) := \frac{p(A \cap B)}{p(B)} \quad \Leftrightarrow \quad p(A|B)\, p(B) = p(A \cap B) \; . \tag{1.5}$$

Because $A \cap B = B \cap A$ the inverse conditional probability reads

$$p(A \cap B) = p(B \cap A) \quad \Rightarrow \quad p(B|A) = \frac{p(A \cap B)}{p(A)} \; . \tag{1.6}$$

These formulas can be viewed as definitions or as axioms, extending the Kologorov axioms. They give us Bayes' Theorem, which states that the conditional probability for a theory T give a set of measurements M is

$$\boxed{p(T|M) = \frac{p(M|T)\, p(T)}{p(M)}} \; . \tag{1.7}$$

It comes with a set of definitions: $p(T|M)$ is called posterior probability; if we are interested in $p(M|T)$ as a function of the second argument T, it is called likelihood. The problem with the likelihood is the lack of normalization as a function of T. $p(M)$ is called evidence, and it is typically replaced through the normalization condition

$$\begin{aligned} 1 = \int dT \, p(T|M) &= \frac{1}{p(M)} \int dT \, p(M|T)\, p(T) \\ \Leftrightarrow \qquad p(M) &= \int dT \, p(M|T)\, p(T) \; . \end{aligned} \tag{1.8}$$

Finally, $p(T)$ is called prior probability or just prior. In (1.8) we see that is defines an integration measure over theory space. Bayes' Theorem tells us how to combine two independent measurements. We first make one measurement, then compute its posterior probability, use this as the prior for the second measurement, and compute the combined probability. This is logically clear, but technically complicated.

Information entropy To describe complex systems with the help of probabilities, we need the concept of information entropy. Let us start with a system with 2^N equally probably states,

$$p_j = 2^{-N} \qquad \text{mit} \qquad \sum_j p_j = 2^N \, 2^{-N} = 1 \; . \tag{1.9}$$

To find out in which state the system is we can construct a decision tree. The most efficient algorithm is to step-wise split the possible states into groups of equal size and then ask in which of the two halves the system lives. This way we avoid any element of luck, and we need

$$N = -\log_2 p_j = \frac{\ln p_j}{\ln 2} \tag{1.10}$$

answers $\{0, 1\}$. Now we generalize the system to Ω states with different probabilities p_j. In that case the branches of the decision tree correspond to different probabilities, there will be an element of luck. The number of necessary answers should be the same, but only defined as an expectation value. It defines the information entropy H,

$$\langle -\log_2 p_j \rangle = \boxed{-\frac{1}{\ln 2} \sum_{j=1}^{\Omega} p_j \ln p_j =: \frac{H}{\ln 2}} \tag{1.11}$$

By definition, the information entropy measures (logarithmic) uncertainty in units of $\ln 2$, so-called bits. Uncertainty is the amount of expected information needed to correctly identify the state of the system.

First, we see that for $p_j \in [0, 1]$ each term in the expectation value in (1.11) is positive, so $H \geq 0$. The entropy assumes its smallest value for only one term, $p_j = 1$. Here we know the system perfectly. For two outcomes with $p_1 = p$ and $p_2 = 1 - p$ we can compute the maximum entropy easily,

$$\begin{aligned} H &= -\left[p \ln p + (1-p) \ln(1-p)\right] \\ \frac{dH}{dp} &= -\left[\ln p + \frac{p}{p} - \ln(1-p) - \frac{1-p}{1-p}\right] \\ &= -\ln p + \ln(1-p) = 0 \qquad \Leftrightarrow \qquad p = 1 - p = \frac{1}{2} \; . \end{aligned} \tag{1.12}$$

The information entropy vanishes for $p = 0$ and $p = 1$ and is symmetric around its maximum at $p = 1/2$. For a perfectly understood dataset with only $p(x) = 0$ and $p(x) = 1$ there is no entropy. In general, the entropy is maximal for $\Omega = 1/p$, and its value depends on the number of states,

$$H \leq -\sum_{j=1}^{\Omega} p \ln p = -\ln p = \ln \Omega \; . \tag{1.13}$$

Next, we can ask what happens with the entropy when we split our system in two. If the two systems are not independent, we can compute the entropies in terms of the individual and joint probability $p_{i,j}$,

$$H_1 = -\sum_{i=1}^{\Omega_1} p_i \ln p_i \qquad H_2 = -\sum_{j=1}^{\Omega_2} p_j \ln p_j \qquad H_{12} = -\sum_{i,j} p_{i,j} \ln p_{i,j} \; . \tag{1.14}$$

Because equally distributed systems maximize the entropy, this combined entropy should be smaller than the sum of the independent individual entropies, The difference is the mutual information I_{12}

$$\begin{aligned} I_{12} &= H_1 + H_2 - H_{12} \\ &= -\sum_i p_i \left(\sum_j p_{j|i} \right) \ln p_i - \sum_j \left(\sum_i p_{i|j} \right) p_j \ln p_j + \sum_{i,j} p_{i,j} \ln p_{i,j} \\ &= -\sum_{i,j} \left[p_i p_{j|i} \ln p_i + p_j p_{i|j} \ln p_j - p_{i,j} \ln p_{i,j} \right] \\ &= -\sum_{i,j} p_{i,j} \ln \frac{p_i p_j}{p_{i,j}} \equiv D_{\text{KL}}[p_{i,j}, p_i p_j] \; . \end{aligned} \tag{1.15}$$

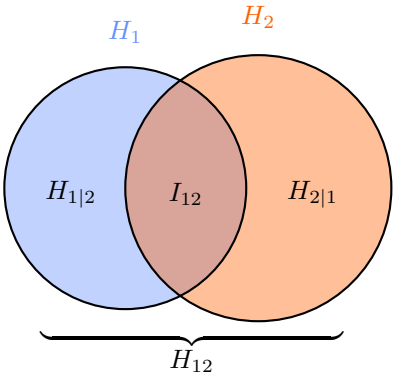


Figure 1: Example of an information diagram.

The relation between the different entropies is illustrated in Figure 1. We can understand the mutual information in terms of clustering algorithms. If we pick a random point in the entire space, what does the association with partition 1 tell us about a possible association with partition 2.

Finally, we can construct the variance of information,

$$\text{VoI}_{12} = H_{1|2} + H_{2|1} = H_1 + H_2 - 2I_{12} \; . \tag{1.16}$$

It compares two partitions, and it vanishes exactly when the two partitions are equal. For finite values it gives a proper distance, *i.e.* positive definite, symmetric, and satisfying the triangle anomaly.

The functional defined in (1.15) is called Kullback–Leibler divergence and compares two probability distributions, evaluated on a dataset corresponding to the first distribution,

$$D_{\text{KL}}[p_a, p_b] = \sum p_a \ln \frac{p_a}{p_b} \neq D_{\text{KL}}[p_b, p_a] \; . \tag{1.17}$$

For continuous distributions it reads

$$\boxed{D_{\text{KL}}[p_a, p_b] = \left\langle \log \frac{p_a}{p_b} \right\rangle_{p_a} \equiv \int dx \; p_a(x) \; \log \frac{p_a(x)}{p_b(x)}} \; . \tag{1.18}$$

It vanishes if two distributions agree everywhere. There is a nice way to show that otherwise the KL-divergence is always positive. The numerator of the logarithm is the negative information entropy for p_a, which means the contribution from the numerator is negative and has a well-defined minimum given in (1.13). The contribution of the denominator is the information entropy for a distribution p_b, which does not match p_a, which is called cross entropy of p_a and p_b. If the two distributions are different, their respective encoding is not optimal, which means the cross entropy is larger than the information entropy, so the KL-divergence is positive. Mathematically, we can use the relation $\log p \leq p - 1$, where the equal sign is true for $p = 1$ to compute

$$\begin{aligned} -D_{\text{KL}}[p_a, p_b] &= \int_{p_a>0} dx \; p_a(x) \; \log \frac{p_b(x)}{p_a(x)} \\ &\leq \int_{p_a>0} dx \; p_a(x) \; \left(\frac{p_b(x)}{p_a(x)} - 1 \right) = \int_{p_a>0} dx \; p_b(x) - 1 \leq 0 \; , \end{aligned} \tag{1.19}$$

using the fact that $p_a(x)$ and $p_b(x)$ are normalized probability distributions. We will come back to the KL-divergence in much more detail later.

Likelihoods and Neyman-Pearson Lemma Combining measurements is easier when we use likelihoods. Let us assume that $p(x)$ gives the probability that an event or phase space configuration x corresponds to some kind of signal, turning the conditional probability or likelihood of (1.7) into

$$p(M|T) \quad \to \quad p(\{x_i\}|T) \; . \tag{1.20}$$

We can split the measurement into a counting experiment leading to n events with s events expected,

$$p_{\text{Poisson}}(n|s) = \frac{s^n}{n!}\, e^{-s} \,, \tag{1.21}$$

and a properly normalized probability for each event,

$$p(x) \in [0,1] \qquad \text{with} \qquad \int dx\, p(x) = 1 \,. \tag{1.22}$$

This way, the likelihood for the set of independent events becomes

$$p(\{x_i\}|T) = \frac{s^n}{n!} e^{-s} \prod_{i=1}^n p(x_i) = \frac{1}{n!} e^{-s} \prod_{i=1}^n [s p(x_i)] \,. \tag{1.23}$$

In this form we see that the combined likelihoods for two independent datasts are simply

$$\begin{aligned} p(\{x_i, x_j\}|T) &= p(\{x_i\}|T)\, p(\{x_j\}|T) \\ &= \frac{s^{n_1+n_2}}{n_1! n_2!} e^{-s_1-s_2} \prod_{i=1}^{n_1+n_2} p(x_i) \,. \end{aligned} \tag{1.24}$$

We can compute the ratio of two likelihoods for the same dataset of n observed events $\{x_i\}$, but two theory hypotheses, for instance signal plus background vs background only,

$$\begin{aligned} \frac{p(\{x_i\}|T_{S+B})}{p(\{x_i\}|T_B)} &= \frac{e^{-s+b}}{e^{-b}} \frac{\prod_{i=1}^n [(s+b) p_{s+b}(x_i)]}{\prod_{i=1}^n [b p_b(x_i)]} \\ &= e^{-s} \prod_{i=1}^n \frac{s p_s(x_i) + b p_b(x_i)}{b p_b(x_i)} \\ \Leftrightarrow \qquad \log \frac{p(\{x_i\}|T_{S+B})}{p(\{x_i\}|T_B)} &= -s + \sum_{i=1}^n \log \frac{s p_s(x_i) + b p_b(x_i)}{b p_b(x_i)} \end{aligned} \tag{1.25}$$

The Neyman-Pearson lemma states that this likelihood ratio is the most powerful test statistics, or test variable, to distinguish two hypotheses. This is defined as the smallest false-negative error for a given false-positive error. For typical physics applications this means it minimizes the mis-identification of a signal for a background fluctuation. The log-likelihood-ratio for independent can be calculated by summing over events, or integrating over phase space in the continuum limit.

Before moving on, let us mention that in the limit of large event counts the Poisson distribution in (1.21) becomes a Gaussian,

$$\boxed{p_{\text{Poisson}}(n|s) \to p_{\text{Gauss}}(n|s) = \frac{1}{\sqrt{2\pi s}} \exp\left(-\frac{(n-s)^2}{2s}\right)} \,, \tag{1.26}$$

a standard form we will use throughout this lecture.

Robust estimators In (1.7) we introduced the likelihood with an abtract definition of a theory T. In (1.25) the two theories were signal-plus-background vs background-only, again without any reference to what makes them different. In general, the theory hypothesis defining a liklihood is decribed by an ntuple of model parameters θ, so we again replace our conventions,

$$p(M|T) \quad \to \quad p(M|\theta) \,. \tag{1.27}$$

To simplify our argument, let us assume that our measurement is not a set of events, but a set of values $f(x)$, for instance counting rates over a binned phase space x. This means we compare a measured set of numbers f_i with the corresponding theory predictions $f(x|\theta)$. If the data f_i and the uncertainties σ_i are statistically distributed, we usually employ a

logarithmic Gaussian likelihood

$$\log p(M|\theta) = \log \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{|f_i - f(x_i|\theta)|^2}{2\sigma_i^2}\right)$$
$$= -\sum_i \frac{|f_i - f(x_i|\theta)|^2}{2\sigma_i^2} + \text{const}(\theta) \; . \tag{1.28}$$

In this form the individual Gaussians have mean f_i, variance σ_i^2, standard deviation σ_i, and a width of the bell curve of $2\sigma_i$. However, in the real world data is usually not distributed statistically, and the tails of distributions are usually less suppressed than the exponential suppression defining the Gaussian.

Given a likelihood, we can estimate the value for a model parameter θ by maximizing the likelihood that it explains the data,

$$\boxed{\hat{\theta} = \text{argmax}_\theta \prod_M p(M|\theta) = \text{argmin}_\theta \sum_M [-\log p(M|\theta)]} \tag{1.29}$$

In the second step we assume that maximizing the likelihood or minimizing the negative log-likelihood gives the same result. We call $\hat{\theta}$ the estimator for the model parameter θ. Removing a θ-dependence by maximizing the function is called profiling, the profile likelihood is the basis of likelihood statistics or inference.

The sensitivity or robustness of an estimator depends on the form of the likelihood. It can be measured by the impact that a chance of a datapoint has on the log-likelihood and is called influence function,

$$\text{IF} = -\frac{d \log p(M|\theta)}{df_i} \tag{1.30}$$

If we already know that Gaussian likelihood does not describe a dataset, for instance in the tails, we can ask what kind of likelihood would lead to a robust estimator. Specifically, we might want to be less sensitive to poorly modelled outliers.

As a simple example, we consider the mean or center $\theta = \mu$ of a one-dimensional likelihood distribution over x. Starting with a one-dimensional Gaussian, we find

$$p(x|\mu) \propto \exp\left(-\frac{(x-\mu)^2}{2\mu}\right)$$
$$\Rightarrow \quad -\log p(x|\mu) = -\frac{(x-\mu)^2}{2\mu} + \cdots$$
$$\Rightarrow \quad \text{IF} = \frac{x-\mu}{\mu} \xrightarrow{x\to\infty} x \; . \tag{1.31}$$

This influence function gives us the robustness of a Gaussian likelihood with extremely suppressed tails. Next, we check a Cauchy or Breit-Wigner distribution centered around zero

$$p(x|\mu) \propto \frac{1}{1+x^2}$$
$$\Rightarrow \quad -\log p(x|\mu) = \log(1+x^2) + \cdots$$
$$\Rightarrow \quad \text{IF} = \frac{2x}{1+x^2} \xrightarrow{x\to\infty} \frac{2}{x} \; . \tag{1.32}$$

Its influence function vanishes for larger x-values, which means the Breit-Wigner distribution is more robust against outliers in the tails than a Gaussian. A more complete set of examples includes a Gaussian and the more robust Laplace, Cauchy (Breit-Wigner) is given in Figure 2.

As a side remark, we can use the estimator as an illustration of frequentist/likelihood vs Bayesian statistics. Everything starts with Bayes theorem, but the two methods ask different questions. For the estimator, the likelihood question in (1.29) determines the value for θ which is the most likely, given the data. For this purpose, it maximizes the likelihood $p(M|\theta)$. If we ask the question what M tells us about possible values of θ, we minimize the negative logarithmic probability,

$$\hat{\theta}_{\text{Bayes}} = \text{argmin}_\theta \sum_M [-\log p(\theta|M)] = \text{argmin}_\theta \sum_M \left[-\log \frac{p(M|\theta)p(\theta)}{p(M)}\right]$$
$$= \text{argmin}_\theta \sum_M [-\log p(M|\theta)p(\theta)] \; . \tag{1.33}$$

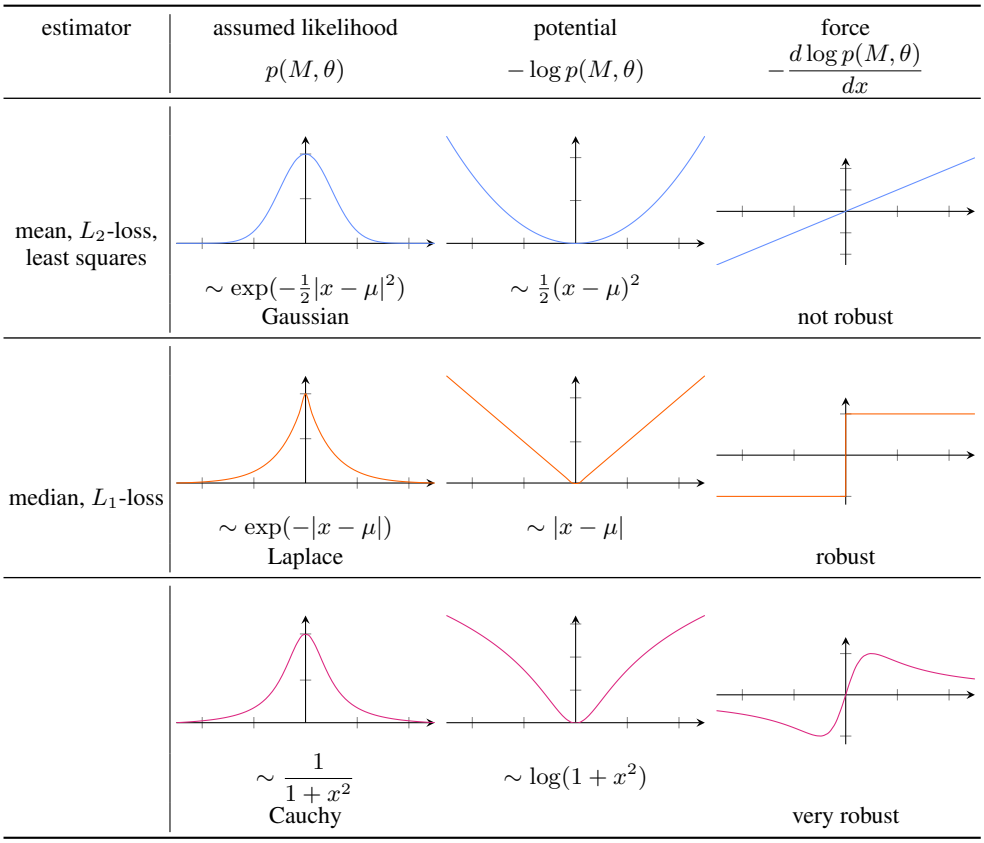


Figure 2: Examples of increasingly robust estimators.

This result differs from (1.29) because we now need a prior to construct the estimator. Similarly, a likelihood approach would get rid of a nuisance parameter or a physics parameter we are currently not interested in by profiling them. This means we identify the maximum likelihood and project this value into the target space. In the Bayesian approach we integrate it out. The prior then defines a θ-space which is needed for such an integration. In terms of the integration measure the profile likelihood would correspond to a delta-distribution prior, which is why we refer to differences between the methods as volume effects. Since the prior is a measure in model space, we have to extract it from other measurements or make an assumption. It is important to always keep in mind, that inference can be done using likelihoods or using the posterior probabilities, and if the two methods give different answers this is because they ask different questions.

1.2 Multivariate analysis

One problem we encounter in any physics analysis is that we want to extract simple numbers, like an energy measurement or a signal probability, from a complex detector output. Historically, this is done by constructing a set of theory-motivated observables and evaluate each of these observables in view of the regression or classification task. If all available information can be extracted from one observable, this observable is called a sufficient statistics, or an optimal observable in particle physics. In reality, this is hardly ever the case. Instead, we are faced with a set of correlated observables, which we need to analyze together.

Let us look at such a set of correlated observables and the problem of multivariate classification. A historic solution is decision tree. Imagine we want to classify an event x_i as signal or background, based on the observables $\mathcal{O}_j$. As a basis for this decision we study a set of training events $\{x_i\}$, histogram them for each $\mathcal{O}_j$, and find the values $\mathcal{O}_{j,\text{split}}$ which define the most successful split between signal and background for each distribution. How is this optimal split defined?

We start with the signal and background probabilities as a function of the signal event count s and the background event

count b,

$$p_S = \frac{s}{s+b} \equiv p \qquad \text{and} \qquad p_B = \frac{b}{s+b} \equiv 1-p \; . \tag{1.34}$$

If we look at a histogrammed normalized observable $\mathcal{O} \in [0 \ldots 1]$ we can compute p and $1-p$ from number of expected signal and background events following (1.34). For instance, we can look at a signal which prefers large $\mathcal{O}$-values and two background distributions to compute these signal and background probabilities $p(\mathcal{O})$.

$$\begin{aligned} s(\mathcal{O}) &= A\mathcal{O} & b(\mathcal{O}) &= A(1-\mathcal{O}) & \Rightarrow \quad p(\mathcal{O}) &= \frac{s(\mathcal{O})}{s(\mathcal{O})+b(\mathcal{O})} = \mathcal{O} \\ s(\mathcal{O}) &= A\mathcal{O} & b(\mathcal{O}) &= B & \Rightarrow \quad p(\mathcal{O}) &= \frac{\mathcal{O}}{\mathcal{O}+\frac{B}{A}} \; . \end{aligned} \tag{1.35}$$

We then declare a single event signal-like when its (signal) probability is $p(\mathcal{O}) > 1/2$, otherwise the event is background-like. This defines the splitting point

$$\mathcal{O}_{j,\text{split}} = \mathcal{O}_j\Big|_{p=1/2} = \begin{cases} \frac{1}{2} \\ \frac{B}{A} \end{cases} \; . \tag{1.36}$$

To organize our multivariate analysis we evaluate the performance of each optimized cut $\mathcal{O}_{j,\text{split}}$, for example to apply the most efficient cut first.

Next, we formalize this condition in terms of statistics — we want to construct the $\mathcal{O}_{j,\text{split}}$ such that they maximize the information entropy defined in (1.11). Because we are not interested in its actual value, we use the natural logarithm

$$H[p] = -\Big[p \log p + (1-p)\log(1-p)\Big] \; , \tag{1.37}$$

Our construction of split values can then be formally written as

$$\mathcal{O}_{j,\text{split}} = \text{argmax}_{\text{ splits}} H[p(\mathcal{O}_j)] \; . \tag{1.38}$$

To build a decision tree out of our observables we first compute the best splitting for each observable individually and then choose the observable with the most successful split. More precisely, for a successful split we want to maximize the difference of the information entropy before the split and the sum of the information entropies after the split. This is called the information gain, and we choose the first observable of our decision tree though

$$\max_j \Big[H_{\text{before split}}[p(\mathcal{O}_j)] - H_{\text{after split},1}[p(\mathcal{O}_j)] - H_{\text{after split},2}[p(\mathcal{O}_j)]\Big] \; . \tag{1.39}$$

Because multivariate analysis using (boosted) decision trees relies on successive split points of pre-defined observables, it cannot be optimal. To do better, we would need to extract $p(\mathcal{O})$ from our dataset and work with it without binning or explicit cuts. This will eventually lead us to classification using neural networks.

A historic illustration for a decision tree used in particle physics is shown in Figure 3. It comes from the first high-visibility application of (boosted) decision trees in particle physics, to identify electron-neutrinos from a beam of muon-neutrinos using the MiniBooNE Cerenkov detector. Each observable defines a so-called node, and the two branches below each node are defined as 'yes' vs 'no' or as 'signal-like' vs 'background-like'. The first node is defined by the observable with the highest information gain among all the optimal splits. The two branches are based on this optimal split value, found by maximizing the cross entropy. Every outgoing branch defines the next node again through the maximum information gain, and its outgoing branches again reflect the optimal split, etc. Finally, the algorithm needs a condition when we stop splitting a branch by defining a node and instead define a so-called leaf, for instance calling all events 'signal' after a certain number of splittings selecting it as 'signal-like'. Such conditions could be that all collected training events are either signal or background, that a branch has too few events to continue, or simply by enforcing a maximum number of branches.

No matter how we define the stopping criterion for constructing a decision tree, there will always be signal events in background leaves and vice versa. We can only guarantee that a tree is completely right for the training sample, if each leaf

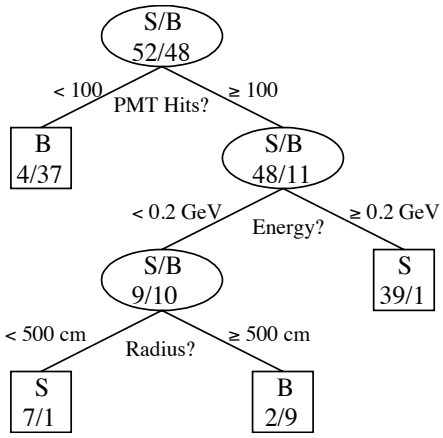


Figure 3: Illustration of a decision tree from an early application in particle physics, selecting electron neutrinos to prove neutrino oscillations. Figure from Ref. [1].

includes one training event. This perfect discrimination obviously does not carry over to an independent test sample, which means our decision tree is overtrained. In general, overtraining means that the performance for instance of a classifier on the training data is so finely tuned that it follows the statistical fluctuations of the training data and does not generalize to the same performance on independent sample of test data.

If we want to further enhance the performance of the decision tree we can focus on the events which are wrongly classified after we define the leaves. For instance, we can add an event weight $w > 1$ to every mis-identified event (we care about) and carry this weight through the calculation of the splitting condition. This is the simple idea behind a boosted decision tree (BDT). Usually, the weights are chosen such that the sum of all events is one. If we construct several independent decision trees, we can also combine their output for the final classifier. It is not obvious that this procedure will improve the tree for a finite number of leaves, and it is not obvious that such a re-weighting will converge to a unique or event improved boosted decision tree, but in practice this method has been shown to be extremely powerful.

Finally, we need to measure the performance of a BDT classification using some kind of success metric. Obviously, a large signal efficiency alone is not sufficient, because the signal-to-background ratio s/b or the Gaussian significance $s/\sqrt{b}$ depend on the signal and background rates. For a simple classification task we can compute four numbers

1. signal efficiency, or true positive rate $\epsilon_S \equiv s^{\text{(S-tagged)}}/s^{\text{(truth)}}$;
2. background efficiency, or true negative rate $b^{\text{(B-tagged)}}/b^{\text{(truth)}}$;
3. background mis-identification rate, or false positive rate $\epsilon_B \equiv b^{\text{(S-tagged)}}/b^{\text{(truth)}}$;
4. signal mis-identification rate, or false negative rate $s^{\text{(B-tagged)}}/s^{\text{(truth)}}$.

If we tag all events we know the normalization conditions $s^{\text{(truth)}} = s^{\text{(S-tagged)}} + s^{\text{(B-tagged)}}$ and correspondingly for $b^{\text{(truth)}}$. The signal efficiency is also called recall or sensitivity in other fields of research. The background mis-identification rate can be re-phrased as background rejection $1 - \epsilon_B$, also referred to as specificity.

Once we have tagged a signal sample we can ask how many of those tagged events are actually signal, defining the

$$\text{purity} = \text{precision} = \frac{s^{\text{(S-tagged)}}}{s^{\text{(S-tagged)}} + b^{\text{(S-tagged)}}} \; . \tag{1.40}$$

Finally, we can ask how many of our decisions are correct and compute the

$$\text{accuracy} = \frac{s^{\text{(S-tagged)}} + b^{\text{(B-tagged)}}}{s^{\text{(truth)}} + b^{\text{(truth)}}} \; , \tag{1.41}$$

reflecting the fraction of correct decisions.

In particle physics we usually measure the success of a classifier in the plane ϵ_S vs ϵ_B, where for the latter we either write $1 - \epsilon_B$ of $1/\epsilon_B$. If a classifier gives us, for example, a continuous measure of signal-ness of an event being signal we can choose different working points by defining a cut on the classifier output. The problem with such any such cut is that we

lose information from all those signal events which almost made it to the signal-tagged sample. If we can construct our classifier such that its output is a probability, we can also weight all events by their signal vs background probability score and keep all events in our analysis.

1.3 Fits and interpolations

We start with the assumption or observation that neural networks are nothing but numerically defined functions. The simplest case, regression networks are scalar or vector fields defined on some space, approximated through parameters θ as $f_\theta(x)$. In physics applications, this space is often some kind of phase space, an important aspect, because many aspects of phase space are interpretable to physicists.

Assuming that we have indirect or implicit access to the truth $f(x)$ in form of a training dataset $(x,f)_j$, we want to construct the approximation

$$\boxed{f_\theta(x) \approx f(x)} \,. \tag{1.42}$$

A set of functional values for a given set of points can be approximated in two ways. First, in a fit we start with a functional form in terms of a small set of parameters which we also refer to as θ.

To determine these network parameters, we use a minimization algorithm on an appropriately defined loss function. More specifically, we maximize the probability for the fit output $f(x_j)$ to correspond to the training points f_j, with uncertainties σ_j. Because we are interested in the θ, we evaluate this probability as a likelihood, for example assuming the Gaussian log-likelihood of (1.28), often referred to as χ^2. For the definition of the best parameters θ we ignore the θ-independent normalization, so the loss function or function we minimize to determine the fit's model parameters is

$$\boxed{\mathcal{L}_\text{fit} = \sum_j \mathcal{L}_j = \sum_j \frac{|f_j - f_\theta(x_j)|^2}{2\sigma_j^2}} \,. \tag{1.43}$$

The fit function is not optimized to go though all or even some of the training data points, $f_\theta(x_j) \neq f_j$. Instead, the log-likelihood loss is a compromise to agree with all training data points within their uncertainties. We can plot the values $\mathcal{L}_j$ for the training data and should find a Gaussian distribution of mean zero and standard deviation one, $\mathcal{N}(\mu=0,\sigma=1)$.

An interesting question arises in cases where we do not know an uncertainty σ_j for each training point, or where such an uncertainty does not make any sense, because we know all training data points to the same precision $\sigma_j = \sigma$. In that case we can still define a fit function, but the loss function becomes a simple mean squared error,

$$\mathcal{L}_\text{fit} = \frac{1}{2\sigma^2} \sum_j |f_j - f_\theta(x_j)|^2 \equiv \frac{1}{2\sigma^2} \text{MSE} \,. \tag{1.44}$$

Again, the prefactor is θ-independent and does not contribute to the definition of the best fit. This simplification means that our MSE fit puts much more weight on deviations for large functional values f_j. This is not what we want, so without control over the uncertainties most machine learning applications will apply a preprocessings to the data,

$$f_j \to \log f_j \qquad \text{or} \qquad f_j \to f_j - \langle f_j \rangle \qquad \text{or} \qquad f_i \to \frac{f_i}{\langle f_i \rangle} \cdots \tag{1.45}$$

In cases where we expect something like a Gaussian distribution a standard scaling would preprocess the data to a mean of zero and a standard deviation of one. In an ideal world, such preprocessings should not affect our results, but in reality they almost always do. The only way to avoid preprocessings is to add information on the deviations, as in the complete likelihood loss in (1.43).

The second way of approximating a set of functional values is interpolation, which ensures $f_\theta(x_j) = f_j$ and is the method of choice for datasets without noise. Between these training data points we choose a linear or polynomial form, the latter defining a so-called spline approximation. It provides an interpolation which is differentiable a certain number of times by matching not only the functional values $f_\theta(x_j) = f_j$, but also the nth derivatives $f_\theta^{(n)}(x \uparrow x_j) = f_\theta^{(n)}(x \downarrow x_j)$. In the machine learning language we can say that the order of our spline defines an implicit bias for our interpolation, because it defines a resolution in x-space where the interpolation works. A linear interpolation is not expected to do well for widely

spaced training points and rapidly changing functional values, while a spline-interpolation should require fewer training points because the spline itself can express a non-trivial functional form.

The main difference between a fit and an interpolation is their respective behavior on unknown dataset. For both, a fit and an interpolation we expect our model $f_\theta(x)$ to describe the training data. To measure the quality of a fit beyond the training data we can compute the loss function $\mathcal{L}$ or the point-wise contributions to the loss $\mathcal{L}_j$ on an independent test dataset. If a fit does not generalize from a training to a test dataset, it is usually because it has learned not only the smooth underlying function, but also the statistical fluctuation of the training data. While a test dataset of the same size will have statistical fluctuations of the same size, they will not be in the same place, which means the loss function evaluated on the training data will be much smaller than the loss function evaluated on the test data. This failure mode is called over-fitting or, more generally, overtraining. For interpolation this overtraining is a feature, because we want to reproduce the training data perfectly. The generalization property is left to choice of the interpolation function.

More systematically, we can define a set of errors which we make when targeting a problem by constructing a fit function through minimizing a loss on a training dataset. First, an approximation error is introduced when we define a fit function, which limits the expressiveness of the network in describing the true function we want to learn. Second, an estimation or generalization error appears when we approximate the true training objective by a combination of loss function and training data set. In practice, these errors are related. A justified limit to the expressiveness of a fit function, or implicit bias, defines useful fits for a given task. In many physics applications we want our fit to be smooth at a given resolution. When defining a good fit, increasing the class of functions the fit represents leads to a smaller approximation error, but increases the generalization error. This is called the bias-variance trade off, and we can control it by limiting or regularizing the expressiveness of the fit function and by ensuring that the loss of an independent test dataset does not increase while training on the training dataset. Finally, any numerical optimization comes with a training error, representing the fact that a fitted function might just live, for instance, in a sufficiently good local minimum of the loss landscape. This error is a numerical problem, which we can solve though more efficient loss minimization. While we can introduce these errors for fits, they will become more relevant for neural networks.

1.4 Neural networks

Next, we introduce a neural network as a numerically defined fit function with a huge number of model parameters θ,

$$f_\theta(x) \approx f(x) \; . \tag{1.46}$$

As mentioned before, we minimize a loss function numerically to determine the neural network parameters θ. This procedure us called network training and requires a training dataset $(x, f)_j$ representing the target function $f(x)$.

We will skip the usual inspiration from biological neurons and instead ask our first question, which is how to describe an unknown function in terms of a large number of model parameters θ without making more assumptions than some kind of smoothness on the relevant scales. For a simple regression task we write the mapping as

$$x \to f_\theta(x) \qquad \text{with} \quad x \in \mathbb{R}^D \quad \text{and} \quad f_\theta \in \mathbb{R} \; . \tag{1.47}$$

The key is to think of this problem in terms of building blocks which we can put together such that simple functions require a small number of modules or building blocks, and model parameters, and complex functions are approximated by a larger number of the same building blocks. We start by defining so-called layers, which in a fully connected or dense network transfer information from all D entries of the vector x defining one layer to all vector entries of the subsequent layer,

$$x \to x^{(1)} \to x^{(2)} \cdots \to x^{(N)} \equiv f_\theta(x) \; . \tag{1.48}$$

Our network consist of N layers, including one input layer $x \to x^{(1)}$, one output layer $x^{(N-1)} \to x^{(N)}$, and $N-2$ hidden layers. If a vector entry $x_j^{(n+1)}$ collects information from all $x_j^{(n)}$, we can try to write each step of this chain as

$$x^{(n-1)} \to x^{(n)} := W^{(n)} x^{(n-1)} + b^{(n)} \; , \tag{1.49}$$

where the $D \times D$ matrix W is referred to as network weights and the D-dimensional vector b as the bias. Both of them are combined into the vector θ. In general, neighboring layers do not need to have the same dimension, which means W does not have to be a diagonal matrix. In our simple regression case we already know that over the layers we need to reduce the width of the network from the input vector dimension D to the output scalar $x^{(N)} = f_\theta(x)$.

Splitting the vector $x^{(n)}$ into its D entries defines the nodes which form our network layer

$$x_i^{(n)} = W_{ij}^{(n)} x_j^{(n-1)} + b_i^{(n)} \qquad\qquad i = 1 \ldots D \,. \tag{1.50}$$

For each node the $D+1$ network parameters are D matrix entries W_{ij} and one bias b_i. If we want to compute the loss function for a given data point (x_j, f_j), we follow the arrows in (1.48), feed each data point x_j through the input layer, go through the following layers one by one, compute the network output $f_\theta(x_j)$, and compare it to f_j though a loss function.

The transformation shown in (1.49) is an affine transformation. Just like linear transformations, affine transformations form a group. This is equivalent to saying that combining affine layers still gives us an affine transformation, just encoded in a slightly more complicated manner. This means our network defined by (1.49) can only describe affine functions, albeit in high-dimensional spaces.

To describe non-affine functions we need to introduce some kind of non-linear structure in our neural network. The simplest implementation of the required nonlinearity is to apply a so-called activation function to each node. Probably the simplest 1-dimensional choice is the so-called rectified linear unit

$$\text{ReLU}(x_j) := \max(0, x_j) = \begin{cases} 0 & x_j \leq 0 \\ x_j & x_j > 0 \end{cases} , \tag{1.51}$$

turning (1.50) into

$$\boxed{x^{(n-1)} \to x^{(n)} := \text{ReLU}\left[W^{(n)} x^{(n-1)} + b^{(n)}\right]} , \tag{1.52}$$

Here we write the ReLU transformation of a vector as the vector of ReLU-transformed elements. This transformation is the same for each node, so all our network parameters are still given by the affine transformations. But now a sufficiently deep network can describe general functions, and combining layers adds complexity, new parameters, and expressivity to our network function $f_\theta(x)$. There are many alternatives to ReLU as the source of non-linearity in the network setup, and depending on our problem they might be helpful, for example by providing a finite gradient over the x-range. However, throughout this lecture we always refer to a standard activation function as ReLU.

This brings us to the second question, namely, how to determine a correct or at least good set of network parameters θ to describe a training dataset $(x, f)_j$. From our fit discussion we know that one way to determine the network parameters is by minimizing a loss function. For simplicity, we think of the MSE loss defined in (1.44) and ignore the normalization $1/(2\sigma^2)$. To minimize the loss we have to compute its derivative with respect to the network parameters. We also ignore the bias for now, so the output layer to the scalar field has the form

$$f_\theta(x) \equiv x_1^{(N)} = \text{ReLU}\left[W_{1k}^{(N)} x_k^{(N-1)}\right] \,. \tag{1.53}$$

For a weight in the last layer we need to compute the derivative

$$\begin{aligned} \frac{d\mathcal{L}}{dW_{1j}^{(N)}} &= \frac{d\left|f - \text{ReLU}[W_{1k}^{(N)} x_k^{(N-1)}]\right|^2}{dW_{1j}^{(N)}} \\ &= \frac{\partial\left|f - \text{ReLU}[W_{1k}^{(N)} x_k^{(N-1)}]\right|^2}{\partial \text{ReLU}[W_{1k}^{(N)} x_k^{(N-1)}]} \frac{\partial \text{ReLU}[W_{1k}^{(N)} x_k^{(N-1)}]}{\partial [W_{1k}^{(N)} x_k^{(N-1)}]} \frac{\partial [W_{1k}^{(N)} x_k^{(N-1)}]}{\partial W_{1j}^{(N)}} \\ &= -2\left|f - \text{ReLU}[W_{1k}^{(N)} x_k^{(N-1)}]\right| \times 1 \times \delta_{jk} x_k^{(N-1)} \\ &\equiv -2\sqrt{\mathcal{L}}\, x_j^{(N-1)} \,, \end{aligned} \tag{1.54}$$

provided $W_{ij}^{(N)} x_j > 0$, otherwise the partial derivative vanishes. This form implies that the derivative of the loss with respect to the weights in the Nth layer is a function of the loss itself and of the previous layer $x^{(N-1)}$. If we ignore the

ReLU derivative in (1.54) and still limit ourselves to the weight matrix in (1.52) we can follow the chain of layers and find

$$\begin{aligned}\frac{d\mathcal{L}}{dW_{ij}^{(n)}} &= \frac{d\left|f-\text{ReLU}[W_{1k}^{(N)}x_k^{(N-1)}]\right|^2}{dW_{ij}^{(n)}} \\ &= \frac{\partial\left|f-\text{ReLU}[W_{1k}^{(N)}x_k^{(N-1)}]\right|^2}{\partial[W_{1k}^{(N)}x_k^{(N-1)}]}\,\frac{\partial[(W^{(N)}\dots W^{(n+1)})_{1k}W_{k\ell}^{(n)}x_\ell^{(n-1)}]}{\partial W_{ij}^{(n)}} \\ &= -2\sqrt{\mathcal{L}}\,\left(W^{(N)}\dots W^{(n+1)}\right)_{1i}x_j^{(n-1)}\,. \end{aligned} \tag{1.55}$$

This means we compute the derivative of the loss with respect to the weights in the reverse direction as the network evaluation shown in (1.48). We have shown this only for the network weights, but it works for the biases the same way. This back-propagation is the crucial step in defining appropriate network parameters by numerically minimizing a loss function. The simple back-propagation might also give a hint to why the chain-like network structure of (1.48) combined with the affine layers of (1.49) have turned out so successful as a high-dimensional representation of arbitrary functions.

The output of the back-propagation in the network training is the expectation value of the derivative of the loss function with respect to a network parameter. We could evaluate this expectation value over the full training dataset. However, especially for large datasets, it becomes impossible to compute this expectation value, so instead we evaluate the same expectation value over a small, randomly chosen subset of the training data. This method is called stochastic gradient descent, and the subsets of the training data are called minibatches or batches

$$\left\langle\frac{\partial\mathcal{L}}{\partial\theta_j}\right\rangle_{\text{minibatch}} \qquad \text{with} \quad \theta_j\in\{b,W\}\,. \tag{1.56}$$

Even through the training data is split into batches and the network training works on these batches, we still follow the progress of the training and the numerical value of the loss as a function of epochs, defined as the number of batch trainings required for the network to evaluate the full training sample.

After showing how to compute the loss function and its derivative with respect to the network parameters, the final question is how we actually do the minimization. For a given network parameter θ_j, we first need to scan over possible values widely, and then tune it precisely to its optimal value. In other words, we first scan the parameter landscape globally, identify the global minimum or at least a local minimum close enough in loss value to the global minimum, and then descend into this minimum. This is a standard task in physics, including particle physics, and compared to many applications of Markov Chains Monte Carlos the standard ML-minimization is not very complicated. We start with the naive iterative optimization in time steps

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \alpha\left\langle\frac{\partial\mathcal{L}^{(t)}}{\partial\theta_j}\right\rangle\,. \tag{1.57}$$

The minus sign means that our optimization walks against the direction of the gradient, and α is the learning rate. From our description above it is clear that the learning rate should not be constant, but should follow a decreasing schedule.

Going back to the three errors introduced in the last section, they can be translated directly to neural networks. The approximation error is less obvious than for the choice of fit function, but also the expressiveness of neural network is limited through the network architecture and the set of hyperparameters. The training error becomes more relevant because we now minimize the loss over an extremely high-dimensional parameter space, where we cannot expect to find the global minimum and will always have to settle for a sufficiently good local minimum. To define a compromise between the approximation and generalization errors we usually divide a ML-related dataset into three parts. The main part is the training data, anything between 60% and 80% of the data. The above-mentioned test data is then 10% to 20% of the complete dataset, and we use it to check how the network generalizes to unseen data, test for overtraining, or measure the network performance. The validation data can be used to re-train the network, optimize the architecture or the different settings of the network. The crucial aspect is that the test data is completely independent of the network training.

1.5 Bayesian networks and likelihood loss

After we have understood how we can construct and train a neural network in complete analogy to a fit, let us discuss what kind of network we actually need for simple physics applications. In a scientific application we are not only interested in

single network output or a set of network parameters θ, but need to include the corresponding uncertainty. Going back to fits, any decent fit approach provides error bands for each of the fit parameters, ideally correlated. With this uncertainty aspect in mind, a nice and systematic approach are so-called Bayesian neural networks. This kind of network is a naming disaster in that there is nothing exclusively Bayesian about them [2], while in particle physics Bayesian has a clear negative connotation. The difference between a deterministic and a Bayesian network is that the latter allow for distributions of network parameters, which then define distributions of the network output and provide central values $f(x)$ as well as uncertainties $\Delta f(x)$ by sampling over θ-space. The corresponding loss function follows a clear statistics logic in terms of a distribution of network parameters.

Let us start with a simple regression task, computing the a scalar transition amplitude as a function phase space points,

$$f_\theta(x) \approx f(x) \equiv A(x) \qquad \text{with} \quad x \in \mathbb{R}^D \; . \tag{1.58}$$

The training data consists of pairs $(x, A)_j$. We define $p(A) \equiv p(A|x)$ as the probability distribution for possible amplitudes at a given phase space point x and omit the argument x from now on. The mean value for the amplitude at the point x is

$$\langle\, A \,\rangle = \int dA\, A\, p(A) \qquad \text{with} \qquad p(A) = \int d\theta\, p(A|\theta)\, p(\theta|x_\text{train}) \; . \tag{1.59}$$

Here, we can think of $p(A|\theta)$ as a single model describing an amplitude through a set of network parameters, while $p(\theta|x_\text{train})$ weights this model by its level of agreement with the training data x_train. We do not know the closed form of $p(\theta|x_\text{train})$, because it includes the training data. Training the network means that we approximate it as a distribution using variational approximation for the integrand in the sense of a distribution and test function

$$p(A) = \int d\theta\; p(A|\theta)\; p(\theta|x_\text{train}) \approx \int d\theta\; p(A|\theta)\; q(\theta|x) \equiv \int d\theta\; p(A|\theta)\; q(\theta) \; . \tag{1.60}$$

As for $p(A)$ we omit the x-dependence of $q(\theta|x)$. This approximation leads us directly to the BNN loss function. We define the variational approximation using the KL-divergence introduced in (1.18),

$$D_\text{KL}[q(\theta), p(\theta|x_\text{train})] = \left\langle \log \frac{q(\theta)}{p(\theta|x_\text{train})} \right\rangle_q = \int d\theta\; q(\theta)\; \log \frac{q(\theta)}{p(\theta|x_\text{train})} \; . \tag{1.61}$$

There are many ways to compare two distributions, defining a problem called optimal transport. We will come back to alternative ways of combining probability densities over high-dimension spaces in Sec. 5. As mentioned above, we do not know $p(\theta|x_\text{train})$, but we can evaluate the likelihood $p(x_\text{train}|\theta)$ using simulations. Using Bayes' theorem we write the KL-divergence as

$$\begin{aligned} D_\text{KL}[q(\theta), p(\theta|x_\text{train})] &= \int d\theta\; q(\theta)\; \log \frac{q(\theta) p(x_\text{train})}{p(\theta) p(x_\text{train}|\theta)} \\ &= D_\text{KL}[q(\theta), p(\theta)] - \int d\theta\; q(\theta)\; \log p(x_\text{train}|\theta) + \log p(x_\text{train}) \int d\theta\; q(\theta) \; . \end{aligned} \tag{1.62}$$

The prior $p(\theta)$ describes the network parameters before training; since it does not really include prior physics or training information we think about it as a hyperparameter which can be chosen to optimize performance and stability. From a practical perspective, a good prior will help the network converge more efficiently, but any prior should give the correct results, and we always need to test the effect of different priors.

The evidence $p(x_\text{train})$ guarantees the correct normalization of $p(\theta|x_\text{train})$ and is usually intractable. If we implement the normalization condition for $q(\theta)$ by construction, we find

$$D_\text{KL}[q(\theta), p(\theta|x_\text{train})] = D_\text{KL}[q(\theta), p(\theta)] - \int d\theta\; q(\theta)\; \log p(x_\text{train}|\theta) + \log p(x_\text{train}) \; . \tag{1.63}$$

The log-evidence in the last term does not depend on θ, which means that it will not be adjusted during training and we can ignore when constructing the loss. However, it ensures that $D_\text{KL}[q(\theta), p(\theta|x_\text{train})]$ can reach its minimum at zero. Alternatively, we can solve the equation for the evidence and find

$$\begin{aligned} \log p(x_\text{train}) &= D_\text{KL}[q(\theta), p(\theta|x_\text{train})] - D_\text{KL}[q(\theta), p(\theta)] + \int d\theta\; q(\theta)\; \log p(x_\text{train}|\theta) \\ &> \int d\theta\; q(\theta)\; \log p(x_\text{train}|\theta) - D_\text{KL}[q(\theta), p(\theta)] \end{aligned} \tag{1.64}$$

This condition is called the evidence lower bound (ELBO), and the evidence reaches this lower bound exactly when our training condition in (1.18) is minimal. Combining all of this, we turn (1.63) or, equivalently, the ELBO into the loss function for a Bayesian network,

$$\boxed{\mathcal{L}_{\text{BNN}} = -\int d\theta\; q(\theta)\;\; \log p(x_{\text{train}}|\theta) + D_{\text{KL}}[q(\theta), p(\theta)]} \; . \tag{1.65}$$

The first term of the BNN loss is a likelihood sampled according to $q(\theta)$, the second enforces a (Gaussian) prior. This Gaussian prior acts on the distribution of network weights. Using an ELBO loss means nothing but minimizing the KL-divergence between the probability $p(\theta|x_{\text{train}})$ and its network approximation $q(\theta)$ and neglecting all terms which do not depend on θ. It results in two terms, a likelihood and a KL-divergence, which we will study in more detail next.

The Bayesian network output is constructed in a non-linear way with a large number of layers, to we can assume that Gaussian weight distributions do not limit us in terms of the uncertainty on the network output. The log-likelihood $\log p(x_{\text{train}}|\theta)$ implicitly includes the sum over all training points.

Before we discuss how we evaluate the Bayesian network in the next section, we want to understand more about the BNN setup and loss. First, let us look at the deterministic limit of our Bayesian network loss. This means we want to look at the loss function of the BNN in the limit

$$q(\theta) = \delta(\theta - \theta_0) \; . \tag{1.66}$$

The easiest way to look at this limit is to first assume a Gaussian form of the network parameter distributions, as given in (1.28)

$$q_{\mu,\sigma}(\theta) = \frac{1}{\sqrt{2\pi}\sigma_q} e^{-(\theta-\mu_q)^2/(2\sigma_q^2)} \; , \tag{1.67}$$

and correspondingly for $p(\theta)$. The KL-divergence of two Gaussians has a closed form,

$$D_{\text{KL}}[q_{\mu,\sigma}(\theta), p_{\mu,\sigma}(\theta)] = \frac{\sigma_q^2 - \sigma_p^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} + \log \frac{\sigma_p}{\sigma_q} \; . \tag{1.68}$$

We can now evaluate this KL-divergence in the limit of $\sigma_q \to 0$ and finite $\mu_q(\theta) \to \theta_0$ as the one remaining θ-dependent parameter,

$$D_{\text{KL}}[q_{\mu,\sigma}(\theta), p_{\mu,\sigma}(\theta)] \to \frac{(\theta_0 - \mu_p)^2}{2\sigma_p^2} + \text{const} \; . \tag{1.69}$$

We can write down the deterministic limit of the Bayesian network loss in (1.65),

$$\boxed{\mathcal{L}_{\text{BNN}} = -\log p(x_{\text{train}}|\theta_0) + \frac{(\theta_0 - \mu_p)^2}{2\sigma_p^2}} \; . \tag{1.70}$$

The first term is again the likelihood defining the correct network parameters, the second ensures that the network parameters do not become too large. Because it include the squares of the network parameters, it is referred to as an L2-regularization. Going back to (1.65), an ELBO loss is a combination of a likelihood loss and a regularization. While for the Bayesian network the prefactor of this regularization term is fixed, we can generalize this idea and apply an L2-regularization to any network with an appropriately chosen pre-factor.

Sampling the likelihood following a distribution of the network parameters, as it happens in the first term of the Bayesian loss in (1.65), is something we can also generalize to deterministic networks. Let us start with a toy model where we sample over network parameters by either including them in the loss computation or not. Such a random sampling between two discrete possible outcomes is described by a Bernoulli distribution. If the two possible outcomes are zero and one, we can write the distribution in terms of the expectation value $\rho \in [0, 1]$,

$$p_{\text{Bernoulli}}(x) = \begin{cases} \rho^x (1-\rho)^{1-x} & x = 0, 1 \\ 0 & \text{else} \; . \end{cases} \tag{1.71}$$

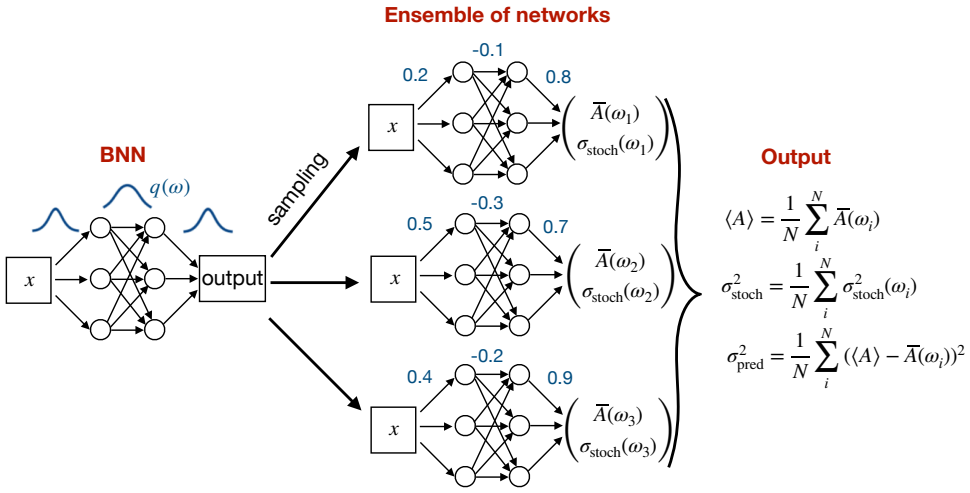


Figure 4: Illustration of a Bayesian network. Figure from Ref. [3].

When we include an event, the network weight is set to θ_0, otherwise it is set to zero, or $\theta = x\theta_0$. We can use the Bernoulli probability as a test function for our integral over the log-likelihood $\log p(x_{\text{train}}|\theta)$ and find

$$\begin{aligned}\mathcal{L}_{\text{BNN}} &= -\int dx \;\; \left[\rho^x(1-\rho)^{1-x}\right] \;\; \log p(x_{\text{train}}|x\theta_0) \\ &= -\rho \log p(x_{\text{train}}|\theta_0)\end{aligned} \tag{1.72}$$

A simple sampling of weights by removing nodes is called dropout and is commonly used to avoid overfitting of networks. For deterministic networks ρ is a free hyperparameter of the network training, while for Bayesian networks this kind of sampling is a key result from the construction of the loss function.

2 Regression

2.1 Amplitude regression

After introducing BNNs using the notation of transition amplitude learning, we still have to extract the mean and the uncertainty for the amplitude A over phase space. In the evaluation step we exchange the two integrals in (1.59) and use the variational approximation to write the mean prediction $\overline{A}$ for a given phase space point as

$$\begin{aligned}\langle A\rangle &= \int dA d\theta \; A \; p(A|\theta) \; p(\theta|x_{\text{train}}) \\ &= \int dA d\theta \; A \; p(A|\theta) \; q(\theta) \\ &\equiv \int d\theta \; q(\theta) \; \overline{A}(\theta) \qquad\qquad \text{with } \theta\text{-dependent mean} \qquad \overline{A}(\theta) = \int dA \; A \; p(A|\theta) \; .\end{aligned} \tag{2.1}$$

We can interpret this formula as a sampling over network parameters, provided we assume uncorrelated variations of the individual network parameters. Corresponding to the definition of the θ-dependent mean $\overline{A}$, the variance of A is

$$\begin{aligned}\sigma^2_{\text{tot}} &= \int dA d\theta \;\; (A - \langle A\rangle)^2 \;\; p(A|\theta) \; q(\theta) \\ &= \int dA d\theta \;\; \left(A^2 - 2A\langle A\rangle + \langle A\rangle^2\right) \;\; p(A|\theta) \; q(\theta) \\ &= \int d\theta \; q(\theta) \left[\int dA \; A^2 \; p(A|\theta) - 2\langle A\rangle \int dA \; A \; p(A|\theta) + \langle A\rangle^2 \int dA \; p(A|\theta)\right]\end{aligned} \tag{2.2}$$

For the three integrals we can generalize the notation for the θ-dependent mean as in (2.1) and write

$$\begin{aligned}
\sigma^2_{\text{tot}} &= \int d\theta\, q(\theta) \left[\overline{A^2}(\theta) - 2\langle A\rangle \overline{A}(\theta) + \langle A\rangle^2\right] \\
&= \int d\theta\, q(\theta) \left[\overline{A^2}(\theta) - \overline{A}(\theta)^2 + \overline{A}(\theta)^2 - 2\langle A\rangle \overline{A}(\theta) + \langle A\rangle^2\right] \\
&= \int d\theta\, q(\theta) \left[\overline{A^2}(\theta) - \overline{A}(\theta)^2 + \big(\overline{A}(\theta) - \langle A\rangle\big)^2\right] \equiv \sigma^2_{\text{stoch}} + \sigma^2_{\text{pred}} \,. \qquad (2.3)
\end{aligned}$$

For this transformation we keep in mind that $\langle A\rangle$ is already integrated over θ and A and can be pulled out of the integrals. This expression defines two contributions to the variance or uncertainty.

First, σ_{pred} is defined in terms of the θ-integrated expectation value $\langle A\rangle$

$$\sigma^2_{\text{pred}} = \int d\theta\, q(\theta) \left[\overline{A}(\theta) - \langle A\rangle\right]^2 . \qquad (2.4)$$

Following from the definition in (2.1), it vanishes in the limit of perfectly narrow network weights, $q(\theta) \to \delta(\theta - \theta_0)$. This limit requires perfect training, so we expect σ_{pred} to decrease with more and better training data. In that sense it represents a statistical uncertainty.

Second, σ_{stoch} is defined without sampling the network parameters,

$$\begin{aligned}
\sigma^2_{\text{stoch}} \equiv \langle \sigma_{\text{stoch}}(\theta)^2\rangle &= \int d\theta\, q(\theta)\, \sigma_{\text{stoch}}(\theta)^2 \\
&= \int d\theta\, q(\theta) \left[\overline{A^2}(\theta) - \overline{A}(\theta)^2\right] . \qquad (2.5)
\end{aligned}$$

It does not vanish for $q(\theta) \to \delta(\theta - \theta_0)$, but it vanishes if the amplitude is arbitrarily well known and well described, characterized by $p(A|\theta) \to \delta(A - A_0)$. While this uncertainty may receive contributions from too little training data, it only approaches a plateau for perfect training. This plateau value can reflect a stochastic training sample, limited expressivity of the network, not-so-smart choices of hyperparameters etc, in the sense of a systematic uncertainty. To avoid mis-understanding we can refer to it as a stochastic or as model-related uncertainty,

$$\sigma_{\text{stoch}} \equiv \sigma_{\text{model}} \equiv \langle \sigma_{\text{model}}(\theta)^2\rangle \,. \qquad (2.6)$$

To understand these two uncertainty measures better, we can look at the θ-dependent network output and read (2.1) and (2.5) as sampling $\overline{A}(\theta)$ and $\sigma_{\text{model}}(\theta)^2$ over a network parameter distribution $q(\theta)$ for each phase space point x

$$\boxed{\text{BNN} : x, \theta \to \begin{pmatrix} \overline{A}(\theta) \\ \sigma_{\text{model}}(\theta) \end{pmatrix}} . \qquad (2.7)$$

If we follow (1.65) and assume $q(\theta)$ to be Gaussian, we now have a network with twice as many parameters as a standard network to describe two outputs. For a given phase space point x we can then compute the three global network predictions $\langle A\rangle$, σ_{model}, and eventually σ_{pred}. Unlike the distribution of the individual network weights $q(\theta)$, the amplitude output is not Gaussian.

For some applications we might want to use this advantage of the BNN loss in (1.70), but without sampling the network parameters θ. In complete analogy to a fit we can then use a deterministic network like in (1.68), described by $q(\theta) = \delta(\theta - \theta_0)$ inserted into the Gaussian BNN loss in (1.70) to define

$$\mathcal{L}_{\text{heteroskedastic}} = \sum_{\text{points } j} \left[\frac{\left|\overline{A}_j(\theta_0) - A_j^{\text{truth}}\right|^2}{2\sigma_{\text{model},j}(\theta_0)^2} + \log \sigma_{\text{model},j}(\theta_0) \right] + \frac{(\theta_0 - \mu_p)^2}{2\sigma_p^2} \,. \qquad (2.8)$$

The interplay between the first two terms works in a way that the first term can be minimized either by reproducing the data and minimizing the numerator, or by maximizing the denominator. The second term penalizes the second strategy, defining a correlated limit of $\overline{A}$ and σ_{model} over phase space. Compared to the full Bayesian network this simplified approach has two disadvantages: first, we implicitly assume that the uncertainty on the amplitudes is Gaussian. Second, σ_{model} only

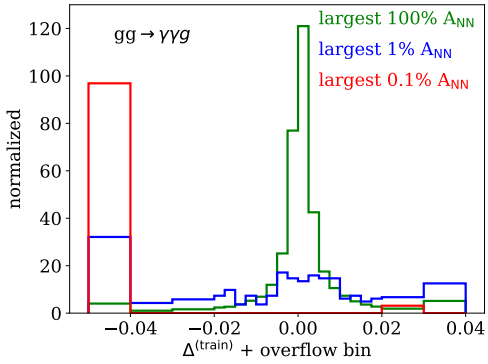


Figure 5: Performance of the BNN in terms of the precision of the generated amplitudes, defined in (2.10) and evaluated on the training (upper) and test datasets. Figure from Ref. [5].

captures noisy amplitude values in the training data. Extracting a statistical uncertainty, as encoded in σ_{pred}, requires us to sample over the weight space. Obviously, this simplification cannot be interpreted as an efficient network ensembling.

For a specific task, let us look at LHC amplitudes for the production of two photons and a jet [4],

$$gg \to \gamma\gamma g(g) \tag{2.9}$$

The corresponding transition amplitude A is a real function over phase space. The jet 4-momenta are identified with the gluon 4-momenta through a jet algorithm. The standard computer program for this calculation is called NJet, and the same amplitudes can also be computed with the event generator Sherpa at one loop. This amplitude has to be calculated at one loop for every phase space point, so we want to train a network once to reproduce its output much faster. Because these amplitude calculations are a key ingredient to the LHC simulation chain, the amplitude network needs to reproduce the correct amplitude distributions including all relevant features and with a reliable uncertainty estimate.

For one gluon in the final state, the most general phase space has $5 \times 4 = 20$ dimensions. We could simplify this phase space by requiring momentum conservation and on-shell particles in the initial and final state, but for this test we leave these effects for the network to be learned. Limiting ourselves to one jet, this means we train a regression network to learn a real number over a 20-dimensional phase space. To estimate the accuracy of the network we can compute the relative deviation between the training or test data with the network amplitudes,

$$\Delta_j^{(\text{train/test})} = \frac{\langle A \rangle_j - A_j^{\text{train/test}}}{A_j^{\text{train/test}}} \tag{2.10}$$

In the upper left panel of Figure 5 we show the performance of a well-trained Bayesian network in reproducing the training dataset. While the majority of phase space points are described very precisely, a problem occurs for the phase space points with the largest amplitudes. The reason for this shortcoming is that there exist small phase space regions where the transition amplitude increase rapidly, by several orders of magnitude. The network fails to learn this behavior in spite of a log-scaling following (1.45), because the training data in these regions is sparse. For an LHC simulation this kind of bias is a serious limitation, because exactly these phase space regions need to be controlled for a reliable rate prediction. This defines our goal of an improved amplitude network training, namely to identify and control outliers in the Δ-distribution of (2.10).

2.2 Numerical integration

The last application of a regression network is the numerical calculation of a given high-dimensional integral, that depends on a large number of external parameters, conveniently stored in a vector $s = (s_1, ..., s_m)$,

$$I(s) = \int_0^1 dx_1 \cdots \int_0^1 dx_D \; f(s;x) \; , \tag{2.11}$$

where x_i are the integration variables and s. This example is mainly based on (part of) the work [6], and one specific and highly relevant physics example is a D-dimensional phase space integral as it occurs in the computation of scattering processes. There, the vector s is given by masses and momentum channels of a given process.

Typically these phase space integrals are computed with Monte-Carlo methods as the dimension of the integrals grow rapidly with the number of the involved particles. The present NN approach is based on the observation that the integrand $f(s;x)$ is typically smooth in both s and x, and hence a combined fit and an analytic integration of the fit function should significantly reduce the numerical work of computing the integral $I(s)$ for general parameters s. It is suggestive that such a fit procedure benefits from the non-linear nature of the optimization in a NN. Such an analysis is certainly beyond the scope of the present example and has also not been undertaken in [6]. Note in this context that it is specifically interesting to look at the scaling of the present approach for large dimensions $d, m \to \infty$.

We proceed with the discussion of the example (2.11). Because the values of the integrand can span a wide numerical range is useful to normalize the integrand, for example by its value at the center of the x-hypercube,

$$f(x;s) \to \frac{f(s;x)}{f(s;\frac{1}{2},\frac{1}{2},,...,\frac{1}{2})} \qquad \Leftrightarrow \qquad I(s) \to \frac{I(s)}{f(s;\frac{1}{2},\frac{1}{2},...,\frac{1}{2})} \; . \tag{2.12}$$

Without going into details, it is also useful to transform the integrand into a form which vanishes at the integration boundaries.

As mentioned in the beginning, we want to cast the integral (2.11) into an analytically solvable form. This requires the computation of the primitive or (indefinite integral) F, defined by

$$\frac{d^D F(s;x)}{dx_1 \dots dx_D} = f(s;x) \; . \tag{2.13}$$

With (2.13), the phase space integral follows by simply evaluating the integration boundaries

$$\begin{aligned}
I(s) &= \int_0^1 dx_1 \cdots \int_0^1 dx_D \; f(s;x) \\
&= \int_0^1 dx_1 \cdots \int_0^1 dx_D \; \frac{d^D F(s;x)}{dx_1 \dots dx_D} \\
&= \int_0^1 dx_1 \cdots \int_0^1 dx_D \; \frac{d}{dx_D} \frac{d^{D-1} F(s;x)}{dx_1 \dots dx_{D-1}} \\
&= \int_0^1 dx_1 \cdots \int_0^1 dx_{D-1} \; \frac{d^{D-1} F(s;x)}{dx_1 \dots dx_{D-1}} \Bigg|_{x_D=0}^{x_D=1} \\
&= \int_0^1 dx_1 \cdots \int_0^1 dx_{D-1} \; \left[\frac{d^{D-1} F(s;x_1,...,x_{D-1},1)}{dx_1 \dots dx_{D-1}} - \frac{d^{D-1} F(s;x_1,...,x_{D-1},0)}{dx_1 \dots dx_{D-1}} \right] \\
&\vdots \\
&= \sum_{x_1,\dots,x_D=0,1} (-1)^{D-\sum x_i} F(s;x) \; .
\end{aligned} \tag{2.14}$$

This leaves us with the task of computing the primitive F of the integrand f, and in most cases of interest including the present phase space example this primitive is not known. In the following we discuss a way of how to reconstruct it and and encode it in a neural network,

$$F_\theta(s;x) \approx F(s;x) \; . \tag{2.15}$$

To begin with, we do not have data for F in order to train a surrogate network for F directly. Instead, a neural network has to use the integrand f. Hence, the idea is to train a surrogate F_θ, such that its Dth derivative matches f. The loss function

is given by the respective mean squared error

$$\mathcal{L}_{\text{MSE}}\left(f(s;x), \frac{dF_\theta(s;x)}{dx_1...dx_D}\right) . \tag{2.16}$$

If the training on the integrand fixes the network weights such that the integrand as well as F are determined by the same network, and F_θ fulfills (2.15), we have directly learned the integral.

A respective neural network can be set up with an activation function on the level of $f_\theta(s;x)$, whose Dth integrals are known. We can also start this consideration with the surrogate F_θ which can be differentiated multiple times with respect to some of its inputs. We accommodate both cases with a differentiable activation function with

$$\frac{d\phi^{(n)}(x)}{dx} = \phi^{(n+1)}(x) \quad \text{for} \quad n = 1, ..., D . \tag{2.17}$$

In (2.17), $\phi^{(n-1)}$ is the primitive of $\phi^{(n)}$. Accordingly, the surrogate $F_\theta(s;x)$ of $F(s;x)$ is build with the activation function $\phi = \phi^{(0)}$, while on the level of $f_\theta(s;x)$ we have the activation function $\phi^{(D)}$ as well as the $\phi^{(n)}$ with $n < D$. In the present case we will finally use the sigmoid function as the activation function either for $\phi^{(D)}$ or for ϕ. If $\phi^{(D)}$ is the sigmoid function, we have

$$\phi^{(D)}(x) = \text{Sigmoid}\,(x) = \frac{1}{1+e^{-x}} , \quad \Rightarrow \quad \phi^{(D-1)}(x) = \log(e^x + 1) , \quad \Rightarrow \quad \phi^{(D-n)}(x) = -\text{Li}_n(-e^x) . \tag{2.18}$$

There exists a fast sum representation of the dilogarithm Li_2 for numerical evaluation. The same set of primitives can be computed for the $\tanh$ activation function. If we choose to the ReLU activation function (1.51) as $\phi^{(D)}$, we are led to

$$\phi^{(D-n)}(x) = \frac{1}{n!} x^n \theta(x) , \tag{2.19}$$

with the Heaviside step function $\theta(x)$.

In turn, if $\phi = \phi^{(0)}$ is the sigmoid function, we have

$$\phi(x) = \text{Sigmoid}\,(x) = \frac{1}{1+e^{-x}} \quad \Rightarrow \quad \phi^{(1)}(x) = (1-\phi)\phi , \quad \Rightarrow \quad \phi^{(2)}(x) = (1-\phi)\phi(1-2\phi) , \tag{2.20}$$

which is readily iterated to general $\phi^{(n)}$.

Both, choosing $\phi^{(D)}$ as the basic activation function as in (2.18) and (2.19), or choosing ϕ as the basic activation function as in (2.20), can be used. In the following we shall use (2.20) for the explicit computation as also done in [6], but the following derivation will not refer to one of these cases. For the computation of the loss function (2.16), we need to compute the derivatives of the fully connected neural network. Following the conventions of (1.48), the input layer for the two input vectors is

$$x_i^{(0)} = \begin{cases} x_i & i \leq D \\ s_{i-D} & i > D \end{cases} . \tag{2.21}$$

For the hidden layers we just replace the ReLU activation function in (1.52) with the sigmoid or its Dth order primitive. However, the following derivation is done for a general activation function ϕ and its derivatives. We start with the output of node i of the layer n,

$$x_i^{(n)} = \phi\left(W_{ij}^{(n)} x_j^{(n-1)} + b_i^{(n)}\right) . \tag{2.22}$$

The scalar output of the network with N layers

$$F_\theta \equiv x^{(N)} = W_j^{(N)} x_j^{(N-1)} + b^{(N-1)} \tag{2.23}$$

can be differentiated, for instance, with respect to x_1,

$$\frac{dF_\theta}{dx_1} = W_j^{(N)} \frac{dx_j^{(N-1)}}{dx_1} = \sum_j W_j^{(N)} \phi^{(1)}\left(W_{jk}^{(N-1)} x_k^{(N-2)} + b_j^{(N-1)}\right) \left[W_{jk}^{(N-1)} \frac{dx_k^{(N-2)}}{dx_1}\right] . \tag{2.24}$$

In (2.24) we have explicitly noted the sum over j, while for the other indices we use the usual summing convention. Equation (2.24) relates the derivative $dx_j^{(N-1)}/dx_1$ to that of $dx_j^{(N-2)}/dx_1$. These derivatives of the activation $x_j^{(i)}$ at the ith level obey the general rule

$$\frac{dx_j^{(i)}}{dx_s} = \phi^{(1)}\left(W_{jk}^{(i)} x_k^{(i-1)} + b_j^{(i)}\right)\left[W_{jk}^{(i)}\,\frac{dx_k^{(i-1)}}{dx_s}\right] , \tag{2.25}$$

which can be read-off from (2.24). The derivative of $x_j^{(i)}$ is proportional to $\phi^{(1)}$ and the derivative of $x_k^{(i-1)}$. Iterating this relation finally terminates with

$$\frac{dx_k^{(0)}}{dx_s} = \delta_{ks} . \tag{2.26}$$

For reducing F_θ to f_θ we have to perform $D-1$ further derivatives of (2.24). The iterative structure is already fully visible within the next derivative with respect to x_2, to wit,

$$\frac{d^2 F_\theta}{dx_1 dx_2} = \sum_j W_j^{(N)}\,\frac{d^2 x_j^{(N-1)}}{dx_1 dx_2} . \tag{2.27}$$

Similarly to the left hand side of (2.24), the left hand side of (2.27) is a special case of the the general rule for the second derivative of an activation $x_j^{(i)}$. This rule follows from the derivative of (2.25) with respect to x_r with

$$\begin{aligned}\frac{d^2 x_j^{(i)}}{dx_r dx_s} =& \phi^{(2)}\left(W_{jk}^{(i)} x_k^{(i-1)} + b_j^{(i)}\right)\left[W_{jk}^{(i)}\frac{dx_k^{(i-1)}}{dx_r}\right]\left[W_{jk}^{(i)}\frac{dx_k^{(i-1)}}{dx_s}\right] \\ &+ \phi^{(1)}\left(W_{jk}^{(i)} x_k^{(i-1)} + b_j^{(i)}\right)\left[W_{jk}^{(i)}\frac{d^2 x_k^{(i-1)}}{dx_r dx_s}\right] ,\end{aligned} \tag{2.28}$$

with the special case for the input layer,

$$\frac{d^2 x_k^{(0)}}{dx_r dx_s} = 0 , \tag{2.29}$$

which readily follows from (2.26).

Now we iterative this procedure up to the derivative w.r.t. x_D for computing the MSE loss in (2.16). This loss can be minimized with respect to the network parameters θ using the usual backpropagation. Because the integrand is known exactly, there is no need to regularize the network, but it cannot hurt either. Also, the numerical generation of integrand values is numerically cheap, which means F_θ an be trained using very large numbers of training data points.

In the original paper, [6], the method is showcased for two integrals, one of them is

$$I_{\text{1L}}(s_{12}, s_{14}, m_H^2, m_t^2) = \int_0^1 dx_1 \int_0^1 dx_2 \int_0^1 dx_3 \frac{1}{F_{\text{1L}}^2} \tag{2.30}$$

with

$$\begin{aligned}F_{\text{1L}} =& m_t^2 + 2x_3 m_t^2 + x_3^2 m_t^2 + 2x_2 m_t^2 - x_2 s_{14} + 2x_2 x_3 m_t^2 - x_2 x_3 m_H^2 + x_2^2 m_t^2 \\ &+ 2x_1 m_t^2 + 2x_1 x_3 m_t^2 - x_1 x_3 s_{12} + 2x_1 x_2 m_t^2 - x_1 x_2 m_H^2 + x_1^2 m_t^2 .\end{aligned} \tag{2.31}$$

Equation (2.30) is needed to compute the LHC rate for Higgs pair production. The accuracy of the estimated integral can be measured in analogue to (2.10),

$$p = \log_{10}\left|\frac{I_{\text{NN}} - I_{\text{truth}}}{I_{\text{truth}}}\right| , \tag{2.32}$$

giving the effective number of digits the estimates gets right. The results for this accuracy are shown in Figure 6. It is based on training an ensemble of eight replicas of the same network, use their average as the central prediction of the integral, and the standard deviation as an uncertainty estimate. The entries in the histogram have different initializations and are trained on different training data. The results for the two different activation functions are similar. The two-loop integral, which we skip in this summary, has a lower accuracy than the one-loop integral, which is to be expected given the larger number of integrations.

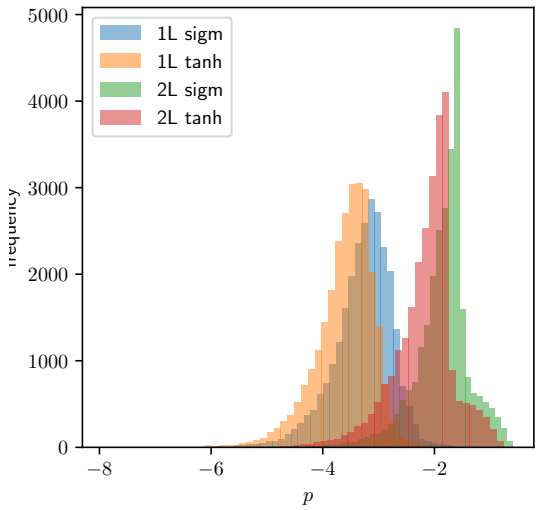


Figure 6: Number of digits accuracy for two integrals, using two different activation functions, one of them defined in (2.30). Figure from Ref. [6].

3 Classification

3.1 Introduction & preparations

For a general discussion, let us start with a data set $\{x\}$, distributed according to the true or data distribution $p_{\text{data}}(x)$. Examples for such a data set is a set of scattering events at the LHC or states/configurations in a statistical system such as an *Ising* spin system e.g. in a solid. An Ising spin can take the values ± 1. This system is the working horse (or guinea pig) of statistical physics.

3.1.1 Spin system

The former setup has already been introduced an explained with simple example. In turn, the latter simple physics system of a spin system on a regular lattice as defined by a solid state system is tailor made for the application and explanation of the current statistics-based approach to Machine Learning as in Section 1.1 and used within first examples in Section 2. Here, we briefly introduce its general setup as is used and required here. For more details we refer beyond the applications in this lecture course we defer the reader to standard statistical physics textbooks.

A statistical system at a given temperature T is defined by its partition function Z, which is given by

$$Z \simeq \text{Tr}\, e^{-\beta H} = \sum_n e^{-\beta E_n}\,, \qquad \text{with} \qquad \beta = \frac{1}{T}\,, \tag{3.1}$$

where we have dropped the normalization of the partition function. In (3.1), H is the Hamiltonian operator of the system at hand and E_n are the energy eigenvalues of the Hamiltonian. The trace in (3.1) sums over all states in the Hilbert space of the system and has been performed with the (normalized) eigenstates $|\varphi_n\rangle$ of the Hamiltonian H with

$$H\,|\varphi_n\rangle = E_n|\varphi_n\rangle\,, \qquad \text{with} \qquad E_n > 0\,, \tag{3.2}$$

assuming a set of discrete eigenstates of the Hamiltonian. It is readily deduced from (3.1), that the probability p_n of a state with the energy E_n is simply

$$p_n = \frac{1}{Z} e^{-\beta E_n}\,, \qquad \Rightarrow \qquad \sum_n p_n = 1\,. \tag{3.3}$$

The expectation value of a general operator $\mathcal{O}$ can be computed with

$$\langle\mathcal{O}\rangle = \frac{1}{Z}\mathrm{Tr}\,\mathcal{O}\,e^{-\beta H} = \frac{1}{Z}\sum_n \langle\varphi_n|\,\mathcal{O}\,|\varphi_n\rangle e^{-\beta E_n}\,, \tag{3.4}$$

where the normalization with $1/Z$ guarantees that $\langle 1\rangle = 1$: the total probability is normalized to unity. Equation (3.4) includes as special cases the entropies and loss functions discussed in the previous Sections Sections 1 and 2.

For an Ising spin system we can also perform the trace as a sum over all spin *configurations* s, which assign the values $s_i \pm 1$ to the different sites i of the given lattice and $E(s)$ is the respective energy. Here i is a d-dimensional vector that labels the sites on the given lattice. Note that for $N = N_1 N_2 \cdots N_d$ lattice sites such a system has exactly 2^N states, where N_μ with $\mu = 1, ..., d$ is the number of layers in each direction. For $H = 0$ all these states would have the same probability as used in the discussion of the information entropy around (1.9). Within this basis, (3.1) takes the form

$$Z \simeq \sum_s \mu(s)\,, \qquad \text{with} \qquad \mu(s) = \langle s|e^{-\beta H}|s\rangle\,, \tag{3.5}$$

where $\mu(s)$ can be understood as a density matrix which takes a diagonal form in this basis. An operator in this system is a function of the configuration s. Its expectation value (3.4) is given by

$$\langle\mathcal{O}\rangle = \frac{1}{Z}\sum_s \mu(s)\,\mathcal{O}(s)\,, \qquad \text{with} \qquad \mathcal{O}(s) = \langle s|\,\mathcal{O}\,|s\rangle\,, \tag{3.6}$$

and Z as defined in (3.5). The original Ising spin system (Ising model) discussed by Lenz and Ising is provided by spins with a next-neighbour interaction on a d-dimensional regular lattice. Its one- and two-dimensional versions admit analytic solutions derived by Ising (one-dimensional) and Onsager (two-dimensional). For the two-dimensional model of infinite extent, the Ising Hamiltonian

$$H = \frac{1}{2}\sum_{i,j} J_{ij}s_i s_j - \sum_i J_i s_i\,, \qquad i = (i_1, i_2)\,, \quad j = (j_1, j_2)\,, \qquad i, j \in \mathbb{Z}^2\,. \tag{3.7}$$

and J_{ij} is only non-vanishing for neighbouring lattice sites: $J_{ij} \neq 0$ for $\|i - j\| = 1$ and $J_{ij} = 0$ for $\|i - j\| \neq 1$. The J_{ij}-term in (3.7) is also referred to as the *hopping* term. If it is uniform, we have

$$J_{ij} = -J\sum_{\mu=1,2}\left(\delta_{i+e_\mu,j} - \delta_{i-e_\mu,j}\right), \qquad \text{with} \qquad e_1 = \begin{pmatrix} 1 & 0 \end{pmatrix}, \quad e_2 = \begin{pmatrix} 0 & 1 \end{pmatrix}, \tag{3.8}$$

where the sign of J is related to physical properties of the system ($J > 0$: ferromagnetic, $J < 0$: antiferromagnetic). The constant J is also called the hopping parameter. In (3.8) we have introduced the two-dimensional Kronecker symbol

$$\delta_{i,j} = \delta_{i_1 j_1}\delta_{i_2 j_2}\,, \tag{3.9}$$

For a constant linear coupling $J_i = h$, the model describes a spin system with a background magnetic field h. The two-dimensional model is easily extended to d-dimensions as well as being readily reduced to one dimensions.

This model, in its local form with next-neighbour interactions but also with non-local interactions, has many application in condensed matter and statistical physics as well as in Machine Learning. It is one of the simplest models that admit phase transitions in dimensions $d \geq 2$. As such it also has many applications and connections to the renormalization group that describes the scale dependence of physics. For simple applications including one for the classification of data sets we introduce a final generalization: instead of considering a variable s_i with discrete values we consider $s_i \to \phi_i \in$ with $\phi_i \in \mathbb{R}$. The respective partition function in d dimensions is given by

$$Z \simeq \int_{\mathbb{R}}\prod_i d\phi_i\, e^{-\beta H(\phi)}\,, \qquad \langle\mathcal{O}\rangle = \frac{1}{Z}\int_{\mathbb{R}}\prod_i d\phi_i\,\mathcal{O}(\phi)\,e^{-\beta H(\phi)}\,. \tag{3.10}$$

in complete analogy with (3.5) and (3.6). For later applications we substitute $\beta H \to S$ and use the specific form

$$S = \sum_{i\in\mathbb{Z}}\left[-2\kappa\sum_{\mu=1}^{d}\phi_i\phi_{i+e_\mu} + (1-2\lambda)\phi_i^2 + \lambda\phi_i^4\right], \tag{3.11}$$

with the hopping parameter κ. Equation (3.11) is a standard form of the action of a scalar lattice field theory. In contradistinction to the Ising model it also includes a higher order term ϕ_i^4 located on each site with the coupling strength λ. Note that such a term cannot be constructed for a spin system with $s_i = \pm 1$. We will use this system or rather its phase structure in Section 3.2 as an example for solving classification tasks with a convolutional neural network (CNN).

3.1.2 Cross entropy

Let us now come back to our discussion of a given data set $\{x\}$ with a data distribution $p_{\text{data}}(x)$. We construct a model approximating the true distribution in terms of the network parameters θ, called

$$p_{\text{model}}(x) \equiv p_{\text{model}}(x|\theta) \; , \tag{3.12}$$

where in the following we shall omit the conditional argument θ. As a function of θ, the probability distribution $p_{\text{model}}(x)$ defines a likelihood, and it should agree with $p_{\text{data}}(x)$. The Neyman-Pearson lemma also tells us that the ratio of the two likelihoods is the most powerful test statistic to distinguish the two underlying hypotheses. One way to compare two distributions is through the KL-divergence (1.18), which vanishes if two distributions agree everywhere. Because the KL-divergence is not symmetric in its two arguments we can evaluate two versions,

$$D_{\text{KL}}[p_{\text{data}}, p_{\text{model}}] = \left\langle \log \frac{p_{\text{data}}}{p_{\text{model}}} \right\rangle_{p_{\text{data}}} \qquad \text{or} \qquad D_{\text{KL}}[p_{\text{model}}, p_{\text{data}}] = \left\langle \log \frac{p_{\text{model}}}{p_{\text{data}}} \right\rangle_{p_{\text{model}}} \; . \tag{3.13}$$

The first is called forward KL-divergence, the second one is the reverse KL-divergence. The difference between them is which of the two distributions we choose to sample the logarithm from, forward is sampled from data, backward is sampled from simulation. Of the two versions we can choose the KL-divergence which suits us better. Since we are working on an existing, well-defined training dataset it makes sense to use the first definition to find the best values of θ and make sure our trained network approximates the training data well,

$$\begin{aligned} D_{\text{KL}}[p_{\text{data}}, p_{\text{model}}] = \left\langle \log \frac{p_{\text{data}}(x)}{p_{\text{model}}(x)} \right\rangle_{p_{\text{data}}} &= \big\langle \log p_{\text{data}}(x) \big\rangle_{p_{\text{data}}} - \big\langle \log p_{\text{model}}(x) \big\rangle_{p_{\text{data}}} \\ &= -\big\langle \log p_{\text{model}}(x) \big\rangle_{p_{\text{data}}} + \text{const}(\theta) \; . \end{aligned} \tag{3.14}$$

We want to maximize the log-likelihood ratio or KL-divergence as a function of the network parameters θ, so we can ignore the second term and instead work with the so-called cross entropy instead,

$$\begin{aligned} H[p_{\text{data}}, p_{\text{model}}] &:= -\big\langle \log p_{\text{model}}(x) \big\rangle_{p_{\text{data}}} \\ &\equiv -\sum_{\{x\}} p_{\text{data}}(x) \log p_{\text{model}}(x) \; . \end{aligned} \tag{3.15}$$

As a probability distribution $p_{\text{model}}(x) \in [0,1]$, so $H[p_{\text{data}}, p_{\text{model}}] > 0$. If we construct $p_{\text{model}}(x)$ by minimizing a KL-divergence or the cross entropy, we refer to the minimized function as the loss function, sometimes also called objective function. The second term in (3.14) only ensures that the target value of the loss function is zero.

If we want to reproduce several distributions simultaneously we can generalize the cross entropy to

$$H[\vec{p}_{\text{data}}, \vec{p}_{\text{model}}] = -\sum_j \big\langle \log p_{\text{model},j}(x|\theta) \big\rangle_{p_{\text{data},j}} \equiv -\sum_j \sum_{\{x\}} p_{\text{data},j}(x) \log p_{\text{model},j}(x|\theta) \; . \tag{3.16}$$

As a sum of individual cross entropies it becomes minimal if each of the $p_{\text{model},j}$ approximates its respective $p_{\text{data},j}$, unless there is some kind of balance to be found between non-perfect approximations. For signal vs background classification, we want to reproduce the signal and the background distributions defined in (1.34) in Section 1.2, giving us the 2-class or 2-label cross entropy

$$\begin{aligned} H[\vec{p}_{\text{data}}, \vec{p}_{\text{model}}] &= -\big\langle \log p_{\text{model}}(x) \big\rangle_{x\sim p_{\text{data}}} - \big\langle \log \left(1 - p_{\text{model}}(x)\right) \big\rangle_{x\sim 1-p_{\text{data}}} \\ &\equiv -\sum_{\{x\}} \Big[p_{\text{data}} \log p_{\text{model}} + (1 - p_{\text{data}}) \log(1 - p_{\text{model}}) \Big] \; . \end{aligned} \tag{3.17}$$

This concludes our discussion of the cross entropy. The main result to remember is the equivalence between minimizing the cross entropy and the KL-divergence defined in (3.14),

$$\text{argmin } H[\vec{p}_{\text{data}}, \vec{p}_{\text{model}}] = \text{argmin} \sum_j D_{\text{KL}}[p_{\text{data},j}, p_{\text{model},j}] \tag{3.18}$$

For our further discussion, we will use the formulation in terms of the KL-divergence, based on the Neyman-Pearson lemma.

3.1.3 Loss functions

Classification tasks are at the heart of essentially every physics experiment, and more generally, at the heart of any statistical physics problem as given by the partition function and action respectively. The introduction of the latter is still 'fresh'; for applications in particle physics let us just define the task, which is to decide if some kind of scattering configuration recorded in the detector corresponds to some kind of signal or the corresponding background. The detector information is abstract and very high-dimensional, so using modern neural networks is an obvious strategy.

We also know from Sections 1.4 and 1.5 that the key ingredient to any neural network and its training is the loss function. We can now put these two things together, implying that for NN-classification we want to learn a signal and a background distribution by minimizing two KL-divergences through our likelihood-ratio loss

$$\begin{aligned}
\mathcal{L}_\text{class} &= \sum_{j=S,B} D_\text{KL}[p_{\text{data},j}, p_{\text{model},j}] \\
&= \Big\langle \log p_{\text{data},S} - \log p_{\text{model},S} \Big\rangle_{p_{\text{data},S}} + \Big\langle \log p_{\text{data},B} - \log p_{\text{model},B} \Big\rangle_{p_{\text{data},B}} \\
&= -\Big\langle \log p_{\text{model},S} \Big\rangle_{p_{\text{data},S}} - \Big\langle \log p_{\text{model},B} \Big\rangle_{p_{\text{data},B}} + \text{const}(\theta) \,.
\end{aligned} \tag{3.19}$$

Dropping the irrelevant constant in the network parameters in the last line of (3.19), we arrive at

$$\boxed{\mathcal{L}_\text{class} = -\sum_{\{x\}} \Big[p_{\text{data},S} \log p_{\text{model},S} + p_{\text{data},B} \log p_{\text{model},B} \Big]} \,. \tag{3.20}$$

From now on we consistently omit the arguments x and θ which we included in Sections 1.4 and 1.5. If we take into account that every jet or event has to either signal or background, $p_S + p_B = 1$, this is just the cross entropy given in (3.17), but derived as a likelihood loss for classification networks. Looking back at (3.13) this simple form of the classification loss tells us that we made the right choice of KL-divergence.

The training procedure is mimicked by a variation that describes the minimization of the loss with respect to θ,

$$\theta_\text{trained} = \text{argmin}_\theta \mathcal{L}_\text{class} = \text{argmin}_\theta \sum_{j=S,B} D_\text{KL}[p_{\text{data},j}(x), p_{\text{model},j}(x|\theta)] \,. \tag{3.21}$$

Replacing $p_B = 1 - p_S$ and a variation with respect to the θ-dependent model distribution yields

$$\begin{aligned}
0 &\stackrel{!}{=} -\frac{\delta}{\delta p_{\text{model},S}} \sum_{\{x\}} \Big[p_{\text{data},S} \log p_{\text{model},S} + (1 - p_{\text{data},S}) \log(1 - p_{\text{model},S}) \Big] \\
&= -\sum_{\{x\}} \left[\frac{p_{\text{data},S}}{p_{\text{model},S}} - \frac{1 - p_{\text{data},S}}{1 - p_{\text{model},S}} \right] \qquad \Leftrightarrow \qquad \boxed{p_{\text{data},S} = p_{\text{model},S}}
\end{aligned} \tag{3.22}$$

If we work under the assumption that a loss function should be some kind of log-probability or log-likelihood, we can ask if our minimized loss function corresponds to some kind of statistical distribution. Again using $p_B = 1 - p_S$ as the only input in addition to the definition of (3.20) we find

$$\begin{aligned}
\mathcal{L}_\text{class} &= -\sum_{\{x\}} \Big[p_{\text{data},S} \log p_{\text{model},S} + (1 - p_{\text{data},S}) \log(1 - p_{\text{model},S}) \Big] \\
&= -\sum_{\{x\}} \log \Big[p_{\text{model},S}^{p_{\text{data},S}} \, (1 - p_{\text{model},S})^{1 - p_{\text{data},S}} \Big] \,.
\end{aligned} \tag{3.23}$$

We can compare the term in the brackets with the Bernoulli distribution in (1.71), which gives the probability distributions for two discrete outcomes. We find that an interpretation in terms of the Bernoulli distribution requires for the outcomes and the expectation value

$$p_{\text{data},S} = x \in \{0, 1\} \qquad \text{and} \qquad p_{\text{model},S} = \rho \tag{3.24}$$

This means that our learned probability distribution has a Bernoulli form and works on signal or background jets and events, encoding the signal vs background expectation value encoded in the trained network.

If we follow this line of argument, our classification network should encode and return a signal probability for a given jet or event, which means the final network layer has to ensure that the network output is $f_\theta(x) \in [0,1]$. For usual networks this is not the case, but we can easily enforce this by replacing the ReLU activation function in the network output layer with a sigmoid function,

$$\text{Sigmoid}\,(x) = \frac{1}{1+e^{-x}} \qquad \Leftrightarrow \qquad \text{Sigmoid}^{-1}(x) \equiv \text{Logit}(x) = \log \frac{x}{1-x}\,. \tag{3.25}$$

The activation function(s) inside the network only work as a source of non-linearity and can be considered just another hyper-parameter of the network. The sigmoid guarantees that the output of the classification network is automatically constrained to a closed interval, so it can be interpreted as a probability without the network having to learn this property. With the loss function of (3.20) and the sigmoid activation of (3.25) we are ready to tackle classification tasks.

The final set of naming conventions we need to introduce is the structure of the training data, for example for classification. If we train a network to distinguish, for instance, light-flavor QCD jets from boosted top quarks, the best training data should be jets for which we know the truth labels. In LHC physics we can produce such datasets using precision simulations, including full detector simulations. We call network training on fully labeled data (fully) supervised learning. The problem with supervised learning at the LHC is that it has to involve Monte Carlo simulations. We will discuss below how we can define dominantly top jet samples. However, no sample is ever 100% pure. This means that training on LHC data will at best start from a relatively pure signal and background samples, for which we also know the composition from simulations and a corresponding analysis. Training a classifier on samples with known signal and background fractions is called weakly supervised learning, and whenever we talk about supervised learning on LHC data we probably mean weakly supervised learning with almost pure samples. An interesting question is how we would optimize a network training between purity and statistics of the training data. Of course, we can find compromises, for instance training a network on a combination of labeled and unlabeled data. This trick is called semi-supervised learning and can increase the training statistics, but there seems to be no good example where this helps at the LHC. One of the reasons might be that training statistics is usually not a problem in LHC applications. Finally, we can train a network without any knowledge about the labels, as we will see towards the end of this section. Here the questions we can ask are different from the usual classification, and we refer to this as unsupervised learning. This category is exciting, because it goes beyond the usual LHC analyses, which tend to be based on likelihood methods, hypothesis testing, and applications of the Neyman-Pearson lemma. Playing with unsupervised learning is the ultimate test of how well we understand a dataset, and we will discuss some promising methods later.

3.2 Convolutional networks

In this Section we introduce the basic architecture of a convolutional neural network (CNN) in Section 3.2.1. Their use for classification tasks illustrates the basic properties of this architectures. We will look at jet images and top tagging in Section 3.2.3 and the identification of phases in a scalar field theory or 'continuous' spin model defined by (3.10) and (3.11) in Section 3.2.2.

3.2.1 Architecture

In both the cases considered the set of data consists our of a set of images, which is very apparent and simple within the example of an Ising-type system, where the image consists out of pixels (sites) with the pixel values ± 1 (black and white). This simple image (excuse the pun) readily generalizes to the (experimental and simulational) images of LHC processes.

The basic idea of convolutional networks is to provide correlations between neighboring pixels, rather than asking the network to learn that pixels 1 and 41 of a spin configuration or jet image lie next to each other. In addition, learning every image pixel independently would require a vast number of network parameters, which typically do not correspond to the actual content of an image. Alternatively, we can try to learn structures and patterns in an image under the assumption that a relatively small number of such patterns encode the information in the image. For example, in the simplest case we may assume that image features are translation-invariant. Indeed, in our statistical physics example this is the case.

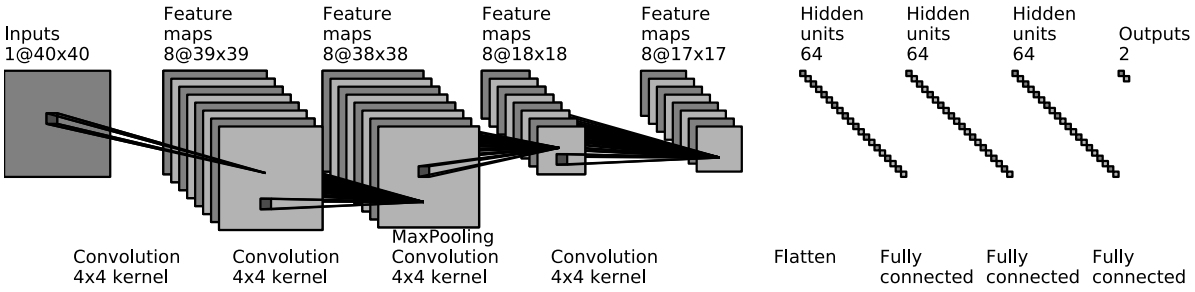


Figure 7: Simple CNN architecture for the analysis of jet images. Figure from Ref. [7], for a more competitive version see Ref. [8].

Then we define a learnable matrix-like filter which applies a convolution and replaces every image pixel with a modified pixel which encodes information about the 2-dimensional neighborhood. This filter is trained on the entire image, which means that it will extract the nature and typical distance of 2-dimensional features. To allow for some self-similarity we can then reduce the resolution of the image pixels and run filters with a different length scale. A sizeable number of network parameters is then generated through so-called feature maps, where we run several filters over the same image. Of course, this works best if the image is not sparse and translation-invariant. For example, an image with diagonal features will lead to a filter which reflects the diagonal structure.

In Figure 7 we illustrate a simple architecture of a convolutional network (CNN), the figures is taken from [7], where CNNs are applied to classify calorimeter images of LHC jets. The network input is the 2-dimensional jet image, $(n \times n)$-dimensional matrix-valued inputs x just like in (1.48), and illustrated in Figure 12. It then uses a set of standard operations:

– Zero padding $(n \times n) \to (n+1 \times n+1)$: It artificially increases the image size by adding zeros to be able to use a filter for the pixels on the boundaries

$$x_{ij} \to \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & x_{ij} & \vdots \\ 0 & \cdots & 0 \end{pmatrix} . \tag{3.26}$$

– Convolution $(n \times n) \to (n \times n)$: To account for locality of the images in more than one dimension and to limit the number of network parameters, we convolute an input image with a learnable filter of size $n_{\text{c-size}} \cdot n_{\text{c-size}}$. These filters play the role of the nodes in (1.50),

$$\boxed{x'_{ij} = \sum_{r,s} W_{rs}\, x_{i+r,j+s} + b \to \text{ReLU}(x'_{ij})} . \tag{3.27}$$

As for any network we also apply a non-linear element, for example the ReLU activation function defined in (1.51).

– Feature maps $n_{\text{f-maps}} \times (n \times n) \to n_{\text{f-maps}} \times (n \times n)$: Because a single learned filter for each convolutional layer defines a small number of network parameters and may also be unreliable in capturing the features correctly, we introduce a set of filters which turn an image into $n_{\text{f-maps}}$ feature maps. The convolutional layer now returns a feature map $x'^{(k)}$ which mixes information from all input maps

$$x'^{(k)}_{ij} = \sum_{l=0}^{n_{\text{f-maps}}-1} \sum_{r,s} W^{(kl)}_{rs}\, x^{(l)}_{i+r,j+s} + b^{(k)} \qquad \text{for} \quad k = 0, ..., n_{\text{f-maps}} - 1 . \tag{3.28}$$

Zero padding and convolutions of a number of feature maps define a convolutional layer. We stack $n_{\text{c-layer}}$ of them. Each $n_{\text{c-block}}$ block keeps the size of the feature maps, unless we use the convolutional layer to slowly reduce their size.

– Pooling $(n \times n) \to (n/p \times n/p)$: We can reduce the size of the feature map through a downsampling algorithm. For pooling we divide the input into patches of fixed size $p \times p$ and assign a single value to each patch, for example a maximum or an average value of the pixels. A set of pooling steps reduces the dimension of the 2-dimensional image representation towards a compact network output. An alternative to pooling are stride convolutions, where the center of the moving convolutional filter skips pixels.

– Flattening $(n \times n) \to (n^2 \times 1)$: Because the classification task requires, two distinct outputs, we have to assume that the 2-dimensional correlations are learned and transform the pixel matrix into a 1-dimensional vector,

$$x = (x_{11}, \ldots, x_{1n}, \ldots, x_{n1}, \ldots, x_{nn}) \ . \tag{3.29}$$

This vector can then be transformed into the usual output of the classification network.

– Fully connected layers $n^2 \to n_{\text{d-node}}$: On the pixel vectors we can use a standard fully connected network as introduced in (1.48) with weights, biases, and ReLU activation,

$$x'_i = \text{ReLU}\left[\sum_{j=0}^{n^2-1} W_{ij} x_j + b_i\right] \ . \tag{3.30}$$

The deep network part of our classifier comes as a number of fully connected layers with a decreasing number of nodes per layer. Finally, we use the classification-specific sigmoid activation of (3.25) in the last layer, providing a 2-dimensional output returning the signal and background probability for a given jet image.

In this CNN structure it is important to remember that the filters are learned globally, so they do not depend on the position of the central pixel. This means the size of the CNN does not scale with the number of pixels in the input image. Second, a CNN with downsampling automatically encodes different resolutions or fields of vision of the filters, so we do not have to tune the filter size to the features we want to extract. One way to increase the expressivity of the network is a larger number of feature maps, where each feature map has access to all feature maps in the previous layer. The number of network parameters then scales like

$$\#_{\text{CNN-parameters}} \sim n^2_{\text{c-size}} \times n_{\text{f-maps}} \times n_{\text{c-layer}} \ll n^2 \ . \tag{3.31}$$

Of course, CNNs can be defined in any number of dimensions, including a 1-dimensional time series where the features are symmetric under time shifts. For a larger number of dimensions the scaling of (3.31) becomes more and more favorable.

3.2.2 Classification of phases in Ising-type models

Now we put the Ising-type model to work, which we have introduced in Section 3.1.1 with the action (3.11) in $d = 2$ dimensions. This model with a continuous field $\phi_i \in \mathbb{R}$ as well as the underlying Ising model with the spin $s_i \in \{-1, 1\}$ allow also for a simple interpretation of the two key operations

(i) Convolution

(ii) Pooling

in the CNN described in the last Section, Section 3.2.1. For that matter we concentrate on the simple Ising spin system with the action (3.7) and the spin on the site i takes the values $s_i = \pm 1$. We consider a square lattice with the length $L = a\,N$ in each direction with the lattice spacing a and $N = 8$, see Figure 9. Alternatively, one may also consider the continuous field $\phi_i \in \mathbb{R}$ and the action (3.11). Both actions, (3.7) or (3.11), can we written in the spirit of the affine transformation in a neural network as

$$S(\phi) = \frac{1}{2}\phi_i W_{ij} \phi_j + b_i \phi_i \,, \tag{3.32}$$

where we have dropped the higher order terms in ϕ_i that are present in (3.11) for the sake of simplicity. The equation of motion (EoM) derived from (3.32) is obtained by a derivative with respect to

$$\frac{\partial S(\phi)}{\partial \phi_i} = W_{ij}\phi_j + b_i = 0 \,, \tag{3.33}$$

which is nothing but the affine transformation (1.49) with $x \to \phi$. Note that a solution of the EoM is a minimum or more generally a saddle point or extremum of the action (3.32). Accordingly, the affine transformation from one layer to the next is tantamount to the the evolution of a field or spin $\phi^{(n)}$ from a given lattice with label n to the next one with label $n + 1$ with a 'drift' force that simply is its equation of motion. Evidently, for a field ϕ or spin configuration s that satisfy the equation of motion, ϕ or s are not transformed at all, the EoM is a fixed point of the transformation and the respective

action is at its minimum. In turn, evolving the field with its EoM force evolves the field towards the EoM. We note that for a given set of lattices (network) with a given set of actions $(W_{ij}^{(n)}(\theta), b_i^{(n)}(\theta))$ the configuration $\phi^{(n)}$ evolves towards a minimum of the sequences of actions.

With the above picture in the back of our mind we now re-analyze the two steps (i) (convolution) and (ii) (pooling):

The first -convolutional- operation (i) can be understood as a two step process:

In the first step each spin s_i or field ϕ_i is mapped to a weighted average of all the spins or fields in the system. For a spin system this reads

$$s_i \to \sum_j W_{ij} s_j + b_j \,, \qquad \text{with} \qquad j = (j_1, j_2) \,, \quad j_1, j_2 = 1, ..., N \,, \tag{3.34}$$

and similarly for the field ϕ_i. Typically, the matrix W is relatively sparse and only spins in a neighbourhood of the lattice site i are summed over. In the case of the Ising spin the map (3.34) has to be combined with a sign function, as the new spin s_i' can only take values ± 1. This limits the choice of the activation function and we are led to

$$s_i' = \text{sign}\left(\sum_j W_{ij} s_j + b_j \right) \,. \tag{3.35}$$

We note in passing that the exceptional case $\sum_j W_{ij} s_j + b_j = 0$ can be avoided by an appropriate choice of W and b. Moreover, in the case of the continuous field ϕ_i, the activation function can be chosen freely. Such a convolution mixes the information of the different sites, in most cases locally. However, at least for an invertible matrix, (3.34) retains all information as we have described a convolutional step with a stride of one. The application of the activation function leads to an information loss as does a convolutional step with a stride larger than one.

The second step (ii), the pooling layer, reduces the information. A simple pooling algorithm on the square lattice under consideration consists of summing over the fundamental squares $\mathcal{C}_{i'}$ as indicated in Figure 9,

$$\phi_{i'}' = \frac{1}{4} \sum_{i \in \mathcal{C}_{i'}} \phi_i \,, \qquad i_1', i_2' = 1, ..., \frac{N}{2} \,, \tag{3.36}$$

assuming for the sake of simplicity that $N/2 \in \mathbb{N}$. In the case of the Ising spin, pooling over four sites is not advisable as the spins may add up to zero. Then, instead of the fundamental squares $\mathcal{C}_{i'}$ we may use fundamental triangles with three spins. These sums are either positive or negative and hence the sign function as in (3.35) can be applied without problems. As already mentioned in the introduction of the CNN in Section 3.2.1, the poling step can be subtituted by a convolutional

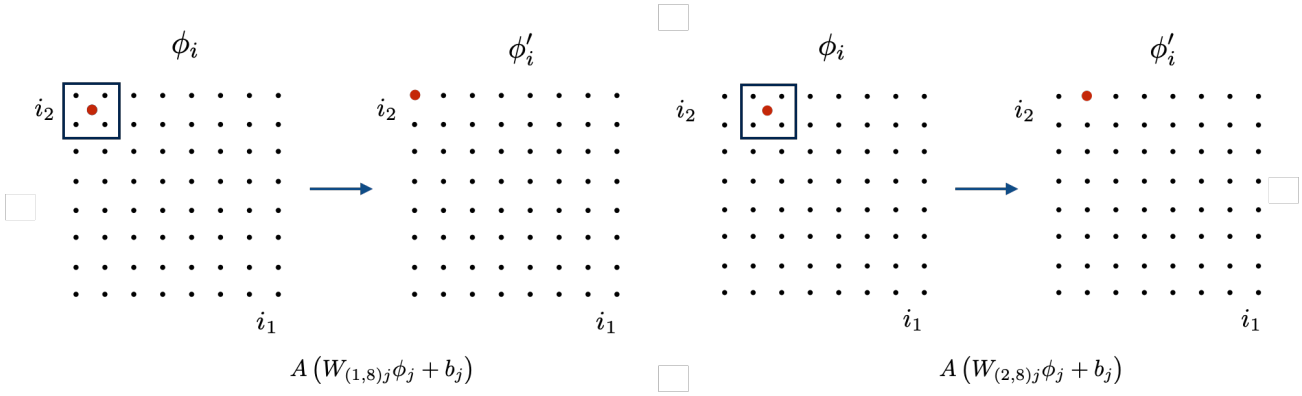


Figure 8: Convolution step in an Ising-type system. We show the map to the new field $\phi_{(1,8)}$ and $\phi_{(2,8)}$ on the sites $i = (1,8), i = (2,8)$. The convolution matrix W is supposedly only taking into account the fundamental squares $\mathcal{C}_i$ with the upper left site i. This local convolution simply averages the spin or field values with the weights W_{ij} with a potential affine part b_j.

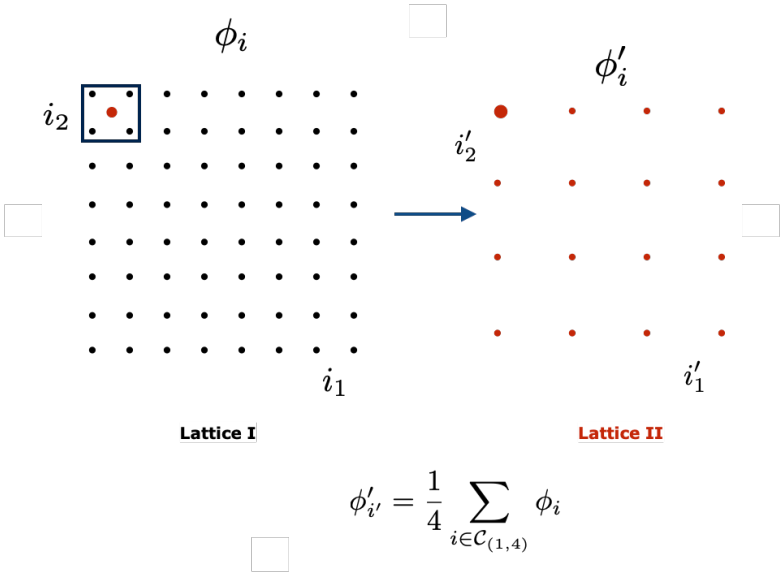


Figure 9: Pooling step in an Ising-type system. We show the averaging of the field on the square lattice with $i_1, i_2 = 1, ..., N$ with $N = 8$ with lattice spacing a and lengths $L_1 = L_2 = N\,a$ in the fundamental square $\mathcal{C}_{(1,4)}$ into the new field $\phi'_{(1,4)}$ living on the new lattice $i'_1, i'_2 = 1, ..., 4$ and lattice spacing $2a$. The lattice length is still the same. In spin systems this is called a block spinning step (coarse graining).

step with a stride larger than one, which is very obvious by comparing Figure 8 with Figure 9. The latter is the former with a stride of two and a fixed 2×2 filter with entries $1/4$ for each filter element.

Evidently, the two steps bear quite some similarities but the second step pools information at the costs of loosing some of it. The pooling transformation is well known from the conceptual and computational background of spin systems (lattice field theory) and is called Kadanoff block spinning. It carries the information of the scale dependence of physics, and is also called a renormalisation group transformation. Finally, we also note that while we have depicted very local convolution and pooling steps, one may also consider non-local convolution and pooling steps.

Within this two-dimensional lattice set-up for an Ising-type system with the action (3.11) we formulate the classification task that will be attacked with a CNN.

To that end we first briefly discuss the non-trivial properties of this model. Apparently, the model has two parameters κ and λ. However, it is only the relative strength which defines the physics or the correlation functions

$$\mathcal{C}_n(i_1, ..., i_n) = \langle \phi_{i_1} \cdots \phi_{i_n} \rangle \tag{3.37}$$

of the system. These are nothing but the *moments* of the underlying probability distribution and the knowledge of all $\mathcal{C}_n$ resolves the distribution and hence the system. For example, for $\kappa = 0 = \lambda$, the distribution is simply a product of Gaußian distributions on the sites. This entails that odd correlations vanish,

$$\mathcal{C}_{2n+1} = 0\,, \tag{3.38a}$$

and the even correlations are products of the $n = 2$ correlation,

$$\mathcal{C}_{2n}(i_1, ..., i_{2n}) = \prod_{m=1}^{n} \mathcal{C}_2(i_{2m-1}, i_{2m}) + \text{cycl. permut.}\,. \tag{3.38b}$$

This result extends to distributions with $\kappa \neq 0$ and $\lambda = 0$. For $\lambda \neq 0$, the distribution is non-Gaußian and (3.38) does not hold anymore.

We close this short introduction with a remark on the information in the above moments $\mathcal{C}_n$ of the distribution. Evidently, even in the case of a non-Gaußian distribution the higher moments carry a trivial information: one part is given by (3.38b), which is nothing but the product of lower order moments. This information can be readily removed, if the right hand

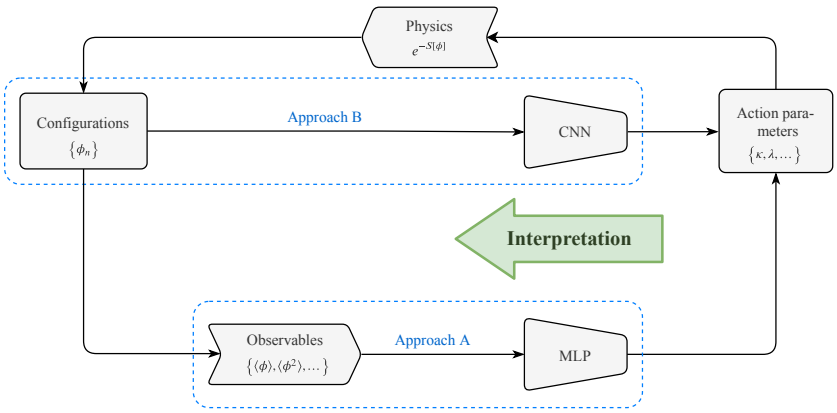


Figure 10: Sketch of a strategy to learn meaningful structures from the simulation data by analysing the networks trained for action parameter inference. Field configurations used for training are either preprocessed into observables for the MLP (Approach A) or directly operated upon with a CNN (Approach B). Obtaining accurate predictions for the parameters indicates approximate cycle consistency in the above diagram, which supports the notion that the networks have successfully identified characteristic features. These can then be extracted in a subsequent interpretation step using LRP.

side of (3.38b) (and further products) are subtracted from the correlation function $\mathcal{C}_{sn}$. Then, these subtracted moments (cumulants) vanish for a Gaußian distribution and only the lowest one $n = 1$ survives. This is nothing but the variance of the distribution.

We proceed with the discussion of the properties of the $d = 2$ Ising-type system. In order to encompass both the Ising model and the model with the continuous field, we concentrate on κ as the tuning parameter. These models feature a phase transition from an ordered or broken phase for large $\kappa > \kappa^*$ to a disordered (symmetric) phase at small $\kappa < \kappa^*$ in the limit $N \to \infty$. In the ordered phase the spins s_i or field values ϕ_i are aligned and hence the expectation value $\langle s_i \rangle$ or $\langle \phi_i \rangle$ does not vanish. In turn, in the disordered phase the spins or fields show a random distribution and the expectation values vanish.

These two phases are characterized by an order parameter and we choose

$$M = \frac{1}{N^2} \left\langle \sum s_i \right\rangle , \qquad \text{with} \qquad M = \begin{cases} 0\,, & \text{for} \quad \kappa < \kappa^* \\ \neq 0\,, & \text{for} \quad \kappa > \kappa^* \end{cases} , \tag{3.39}$$

and similarly for the field ϕ_i. The property (3.39) entails that the set of configurations that are sampled from the distribution $\exp\{-\beta H\}$ or $\exp\{-S\}$ are very different for $\kappa < \kappa^*$ and $\kappa > \kappa^*$.

Hence, this model offers a very simple, well-defined and well-tunable (two bit) classification task: learning the phases of the Ising-type model. We shall set up a CNN for this classification task and use the binary cross entropy as a loss function. A more elaborate version of the model as well as the classification task was used in [9] for not only learning the phases of the model, but also the action parameter κ. This asks for a more refined loss function and more details can be found there. Indeed, this set-up can be used to also learn (novel) order parameters with investigating the layer-wise relevance propagation (LRP) and for that puprose the classification task was accompanied by explainable ML via applying Layer-wise relevance propagation, see Figure 10.

We proceed with a description of the CNN and the simulation set-up used for the above task. A respective ready-to-use code is provided with the material:

First we generate a data sets for both the training of the CNN as well as its application/evaluation. These data are given by a set of configurations $\{\phi\}$ according to the probability distribution of the Ising-type model on an 8×8 lattice. This is a relatively small lattice for physics purposes, but it is tailor made for the present classification task. The generation of the training data and evaluation data can be done with a standard Monte-Carlo algorithm, for read-to-use codes see e.g. [10]. In the available integrated code created by Julian Urban this is done with a Hybrid Monte-Carlo with a leapfrog integrator. We emphazise that these simulation details are only provided for the interested reader, for the classification task the data sets can simply be taken and used.

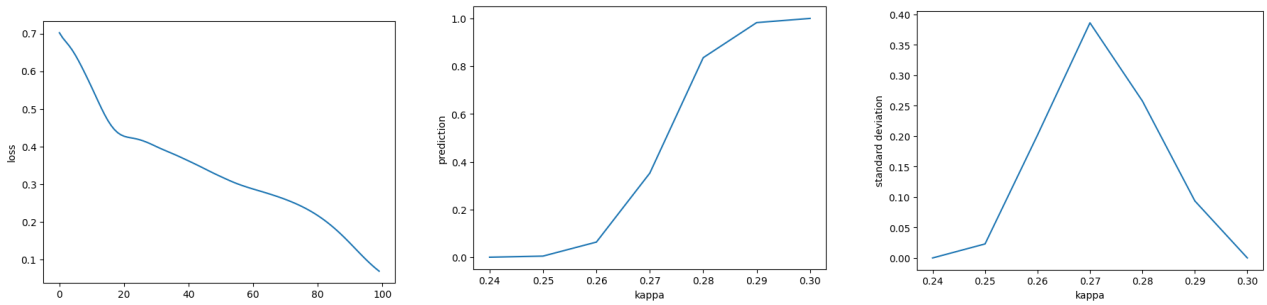


Figure 11: LEFT: evolution of the loss function with the training cycles. MIDDLE: Binary outcome of the classification task as a function of κ: 0 in the symmetric phase and $+1$ in the broken phase. RIGHT: Standard deviation as a function pf κ. Its peak signals the phase transition and this observables can be understood as a susceptibility.

Both, the training samples and evaluation samples contains 10^3 configurations, which is sufficient for the present purpose.

The training samples are generated for a fixed model parameter λ, and κ is chosen in the symmetric and broken phase,

$$\lambda = 0.02\,, \qquad \text{and} \qquad \text{symmetric phase: } \kappa = 0.24\ , \quad \text{broken phase: } \kappa = 0.3\,. \tag{3.40}$$

In turn, the evaluation samples are generated for

$$\kappa = 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30\,. \tag{3.41}$$

These data sets are used to train a CNN with the classification task of learning phases. This very simple task does not require a very elaborate network and we use a CNN with a circular padding (padding_ mode circular). The kernel or filter matrix has the size 3×3, which is relatively big in comparison to the size of the lattice. Similar to Figure 7, 8 feature maps are used and we have two of theses layers. No pooling is applied due to the small size of the lattice. After third convolutional step with no feature maps the lattice is flattened and then the two bit output is generated. Finally we remark that we use the full data set at once for training the CNN as the size of data set and the overall set-up allows for this. Note that for more realistic data sets this is not an option as the training process would be slowed down considerably.

The results for the loss function and the classification task are shown in Figure 11. The evolution of the loss function with the training cycles clearly shows its convergence. Moreover, the κ-dependence of the binary classification outcome (0 for the symmetric phase and $+1$ for the broken phase) can be understood as an order parameter such as (3.39) and shows a rise from zero to unity in the κ-regime of the phase transition. The standard deviation peaks at the phase transition with its critical slowing down of the convergence and can be used to determine the critical $\kappa^* \approx 0.27$. This concludes our example for resolving a classification task with a simple CNN as well as an introduction of spin systems or more generally simple statistical systems as the physical precursors or prototypes for neural networks.

3.2.3 Jet images and top tagging

We observe something in a detector, and we want to know what the detector signal comes from. A simplified version of this task is the question if a detector output corresponds to a signal or a background instance or event. We can ask why we should limit ourselves to a set of theory-defined or high-level observables, or if such a classification could work directly on the detector output channels in a purely data-driven approach.

It turns out that particle physics benefits mostly from image recognition research. The most active field applying such image-based methods is subjet physics. It has, for a while, been a driving field for creative physics and analysis ideas at the LHC. An established subjet physics task like identifying the parton leading to an observed jet, is ideal to develop ML-methods to beat standard methods. The classic, multivariate approach has two weaknesses. First, it only uses information preprocessed into theory-inspired and high-level observables. Next, we need to ask the question is we cannot systematically exploit low-level information about a jet to identify its partonic nature.

For a specific classification task, let us look at a jet and ask the question what kind of particles gave us that final state configuration. This is not trivial, given that jets are complex objects which can come from a light quark or a gluon, but

also from quarks that decay through the electroweak interaction at the hadron level, like charm or bottom quarks. They can also come from a tau lepton, decaying to quarks and a neutrino, or from boosted gauge bosons or Higgs bosons or top quarks. Even when we are looking for apparently simple electrons, we need to be sure that they are not one of the many charged pions which can look like electrons especially in the forward detector. Similarly, photons are not trivial to separate from neutral pions when looking at the electromagnetic calorimeter. Really, the only particle which we can identify fairly reliably at the LHC are muons.

A standard benchmark for jet classification is to separate boosted, hadronically decaying top quarks from QCD jets. Tops are the only quarks which decay through their electroweak interactions before they can form hadrons,

$$t \to bW^+ \to b\ell^+\nu_\ell \qquad \text{or} \qquad t \to bW^+ \to bu\bar{d}\,. \tag{3.42}$$

Most top quarks are produced in pairs and at low transverse momentum. However, even in the Standard Model a fraction of top quarks will be produced at large energies. For heavy resonances, like a new Z' gauge boson decaying to top quarks, the tops will receive transverse momenta around

$$p_{T,t} \sim p_{T,\bar{t}} \sim \frac{m_{Z'}}{2} - m_t\,. \tag{3.43}$$

Before we go into the details of ML-based jet tagging we need to briefly introduce the standard method to define and analyze jets. Acting on calorimeter output in the usual $\Delta\eta$ (longitudinal) vs $\Delta\phi$ (transverse) plane, we usually employ QCD-based algorithms to determine the partonic origin of a given configuration. Such jet algorithms link physical objects, for instance calorimeter towers or particle flow objects, to more or less physical objects, namely partons from the hard process. In that sense, jet algorithms invert the statistical forward processes of QCD splittings, hadronization, and hadron decays. Recombination algorithms try to identify soft or collinear partners among the jet constituents, because we know from QCD that parton splitting are dominantly soft or collinear. To decide if two subjets come from one parton leaving the hard process we have to define a geometric measure capturing the collinear and soft splitting patterns. Such a measure should include the distance

$$R_{ij} = \sqrt{(\Delta\eta_{ij})^2 + (\Delta\phi_{ij})^2} \tag{3.44}$$

and the transverse momentum of one subjet with respect to another or to the beam axis. The three standard measures are

$$\begin{aligned}
&k_T && y_{ij} = \frac{R_{ij}}{R}\,\min\left(p_{T,i}, p_{T,j}\right) && y_{iB} = p_{T,i} \\
&\text{C/A} && y_{ij} = \frac{R_{ij}}{R} && y_{iB} = 1 \\
&\text{anti-}k_T && y_{ij} = \frac{R_{ij}}{R}\,\min\left(p_{T,i}^{-1}, p_{T,j}^{-1}\right) && y_{iB} = p_{T,i}^{-1}\,.
\end{aligned} \tag{3.45}$$

The parameter R only balances competing jet–jet and jet–beam distances. In an exclusive jet algorithm we define two subjets as coming from one jet if $y_{ij} < y_{\text{cut}}$, where y_{cut} is an input resolution parameter. The jet algorithm then consists of the steps

(1) for all combinations of two subjets find $y^{\text{min}} = \min_{ij}(y_{ij}, y_{iB})$
(2a) if $y^{\text{min}} = y_{ij} < y_{\text{cut}}$ merge subjets i and j and their momenta, keep only the new subjet i, go back to (1)
(2b) if $y^{\text{min}} = y_{iB} < y_{\text{cut}}$ remove subjet i, call it beam radiation, go back to (1)
(2c) if $y^{\text{min}} > y_{\text{cut}}$ keep all subjets, call them jets, done

Alternatively, we can give the algorithm the minimum number of physical jets and stop there. As determined by their power dependence on the transverse momenta in (3.45), three standard algorithms start with soft constituents (k_T), purely geometric (Cambridge–Aachen), or hard constituents (anti-k_T) to form a jet. While for the k_T and the C/A algorithms the clustering history has a physical interpretation and can be associated with some kind of time, it is not clear what the clustering for the anti-k_T algorithm means.

There are a few reasons why top tagging has become the *Hello World* of classic or ML-based subjet physics. First, as discussed above, the distinguishing features or top jets are theoretically well defined. Top decays are described by perturbative QCD, and the corresponding mass drop and 3-prong structure can be defined in a theoretically consistent manner without issues with a soft and collinear QCD description. Second, top tagging is a comparably easy task, given

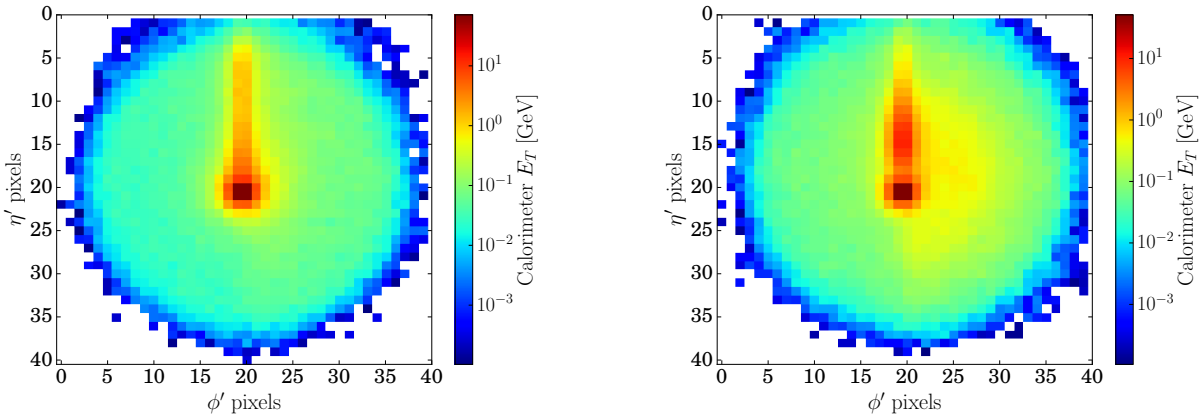


Figure 12: Averaged and preprocessed jet images for a QCD jet (left) and a boosted top decay (right) in the rapidity vs azimuthal plane plane. The preprocessing steps are introduced in the next section. Figures from Ref. [7].

that we can search two mass drops, three prongs, and an additional b-tag within the top jet. Third, in Sec. 3 we mentioned that it is possible to define fairly pure top-jet samples using the boosted production process

$$pp \to t\bar{t} \to (bu\bar{d})\,(\bar{b}\ell^-\bar{\nu}) \qquad \text{with} \qquad p_{T,t} \sim p_{T,\bar{t}} \gtrsim 500\,\text{GeV}\,. \tag{3.46}$$

We trigger on the hard and isolated lepton, reconstruct the leptonically decaying top using the W-mass constraint to replace the unknown longitudinal momentum of the neutrino, and then work with the hadronic recoil jet to, for instance, train or calibrate a classification network on essentially pure samples.

The idea of an image-based top tagger is simple — if we look at the ATLAS or CMS calorimeters from the interaction point, they look like the shell of a barrel, which can be unwrapped into a 2-dimensional plane with the coordinates rapidity $\eta \approx -4.5 \ldots 4.5$ and azimuthal angle $\phi = 0 \ldots 2\pi$, with the distance measure R defined in (3.44). If we encode the transverse energy deposition in the calorimeter cells as color or grey-scale images, we can use standard image-recognition techniques to study jets or events. The network architecture behind the success of ML-image analyses are convolutional networks, and their application to LHC jet images started with Ref. [11]. We show a set of average jet images for QCD jets and boosted top decays in Figure 12. The image resolution of, in this case 40×40 pixels is given by the calorimeter resolution of $0.04 \times 2.25°$ in rapidity vs azimuthal angle. A single jet image looks nothing like these average images. For a single jet image, only 20 to 50 of the 1600 pixels have sizeable p_T-entries, making even jet images sparse at the 1% or 2% level, not even talking about full event images. Nevertheless, we will see that standard network architectures outperform all established methods used in subjet physics.

As always, we can speed up the network training through preprocessing steps. They are based on symmetry properties of the jet images, as we will discuss them in more detail in Sec. 3.3.3. For jet images the preprocessing has already happened in Figure 12. First, we define a central reference point, for instance the dominant energy deposition or some kind of main axis or center of gravity. Second, we can shift the image such that the main axis is in its center. Third, we use the rotational symmetry of a single jet by rotating the image such that the second prong is at 12 o'clock. Finally, we flip the image to ensure the third maximum is in the right half-plane. This is the preprocessing applied to the averaged jet images shown in Figure 12. In addition, we can apply the usual preprocessing steps for the pixel entries from (1.45), plus a unit normalization of the sum of all pixels in an image.

To these jet images we can apply a standard CNN, as illustrated in Figure 7. Before we show the performance of a CNN-based top tagger we can gain some intuition for what is happening inside the trained CNN by looking at the output of the different layers in the case of fully preprocessed images. In Figure 13 we show the difference of the averaged output for 100 signal-like and 100 background-like images. Each row illustrates the output of a convolutional layer. Signal-like red areas are typical for top decays, while blue areas are typical for QCD jets. The feature maps in the first layer consistently capture a well-separated second subjet, and some filters of the later layers also capture a third signal subjet in the right half-plane. While there is no one-to-one correspondence between the location in feature maps of later layers and the pixels in the input image, these feature maps still show that it is possible to see what a CNN learns. One can try a similar analysis for the fully connected network layers, but it turns out that we learn nothing.

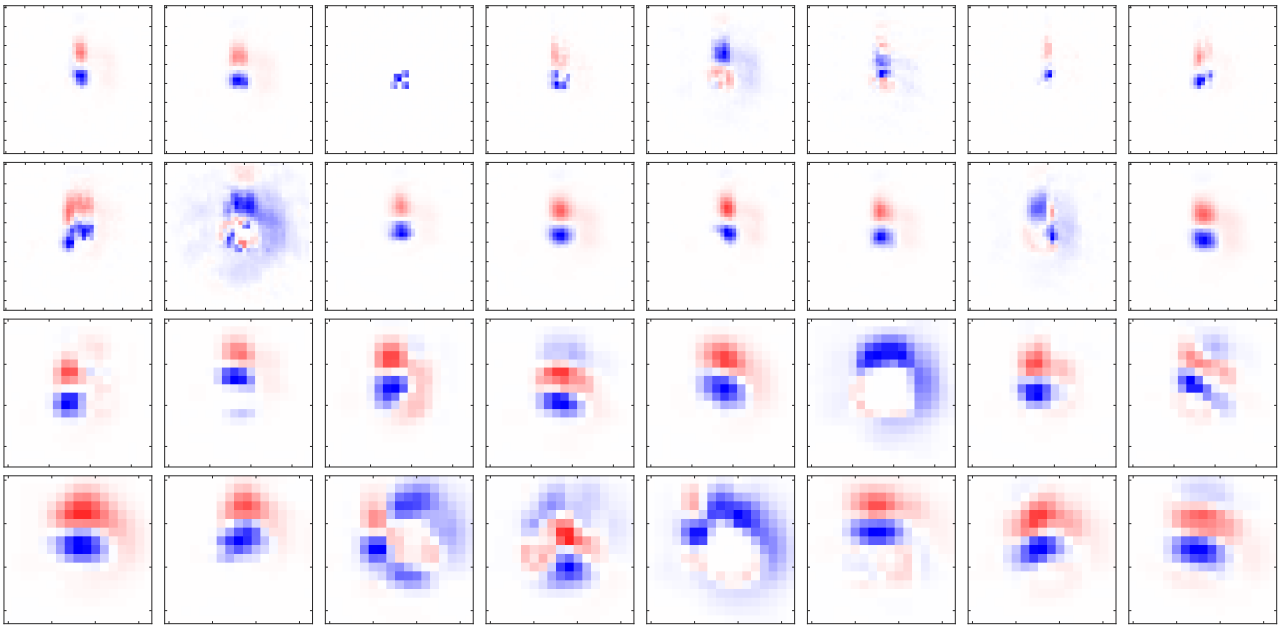


Figure 13: Averaged signal minus background for a simple CNN top tagger. The rows correspond to CNN layers, max-pooling reduces the number of pixels by roughly a factor four. The columns show different feature maps Red areas indicate signal-like regions, blue areas indicate background-like regions. Figure from Ref. [7].

To measure the impact of the pixels of the preprocessed jet image impact on the extracted signal vs background label, we can correlate the deviation of a pixel x_{ij} from its mean value $\bar{x}_{ij}$ with the deviation of the signal probability y from its mean value $\bar{y}$. The correlation for a given set of combined signal and background images is given by the Pearson correlation coefficient

$$r_{ij} = \frac{\sum_{\text{images}} (x_{ij} - \bar{x}_{ij})(y - \bar{y})}{\sqrt{\sum_{\text{images}} (x_{ij} - \bar{x}_{ij})^2} \sqrt{\sum_{\text{images}} (y - \bar{y})^2}} \,. \tag{3.47}$$

Positive values of r_{ij} indicate signal-like pixels. Pixels which carry no information on our classification task will give $r_{ij} \sim 0$. For binary classification, pixels with negative r_{ij} are populated dominantly by background jets. In Figure 14 we show this correlation coefficient for a simple CNN. A large energy deposition in the center leads to classification as background. A secondary energy deposition at 12 o'clock combined with additional energy in the right half-plane means top signal, consistent with Figure 12.

Both, for the CNN and for a traditional BDT tagger we can study signal-like learned patterns in actual signal events by cutting on the output label y. Similarly, we can use background-like events to test if the background patterns are learned as expected. In addition, we can compare the kinematic distributions in both cases to the Monte Carlo truth. In Figure 15 we show two standard distributions, the the fat jet mass m_{fat} and the N-subjettiness ratio τ_3/τ_2. This ratio measures the prongness of the jet in a way that is compatible with perturbative quantum field theory. A small value τ_N indicates consistency with N or fewer substructure axes, so an N-prong decay returns a small ratio τ_N/τ_{N-1}, like τ_2/τ_1 for W-boson or Higgs tagging, or τ_3/τ_2 for top tagging. The CNN and the classic BDT learn essentially the same structures. Their results are even more signal-like than the Monte Carlo truth, because of the stiff cut on y. For the CNN and BDT tagger cases this cut removes events where the signal kinematic features is less pronounced. The BDT curves for the signal are more peaked than the CNN curves because these two high-level observables are BDT inputs, while for the neural network they are derived quantities.

Going back to the CNN motivation, it turns our that preprocessed jet images are not translation-invariant, and they are extremely sparse. This means they are completely different from the kind of images CNNs were developed to analyze. While the CNNs work very well for image-based jet classification this raises the question if there are other, better-suited network architectures for jet taggers.

As common in machine learning, standard datasets are extremely helpful to benchmark the state of the art and develop new ideas. For top tagging, the standard dataset consists of 1M top signal and 1M mixed quark-gluon background jets, produced

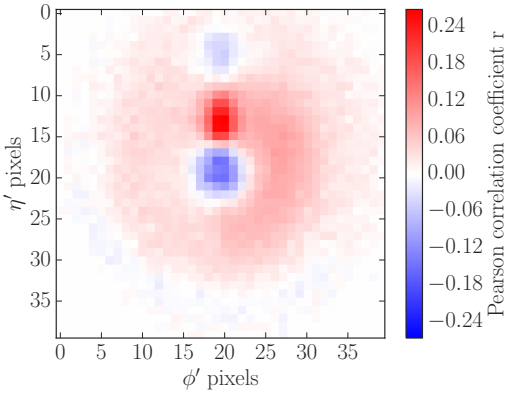


Figure 14: Pearson correlation coefficient for 10,000 signal and background images each. The corresponding jet image is illustrated in Figure 12. Red areas indicate signal-like regions, blue areas indicate background-like regions. Figure from Ref. [7].

with the Pythia8 event generator and the simplified detector simulation Delphes. The are divided into 60% training, 20% validation, and 20% test data. The fat jet is defined through the anti-k_T algorithm with size $R = 0.8$, which means its boundaries in the jet plane are smooth. The dataset only uses the leading jet in each $t\bar{t}$ or di-jet event and requires

$$p_{T,j} = 550 \,....\, 650 \text{ GeV} \qquad \text{and} \qquad |\eta_j| < 2 \,. \tag{3.48}$$

To avoid artifacts, we require a parton-level top and all its decay partons to be within $\Delta R = 0.8$ of the jet axis for the signal jets. The jet constituents are extracted through the Delphes energy-flow algorithm, which combines calorimeter objects with tracking output. The dataset includes the 4-momenta of the leading 200 constituents, which can then be represented as images for the CNN application. Particle information is not included, so b-tagging cannot be added as part of the top tagging, and any quoted performance should not be considered realistic. The dataset is easily accessible as part of a broader physics-related reference datasets [13].

Two competitive versions of the image-base taggers were benchmarked on this dataset, an updated CNN tagger and the standard ResNeXt network. While the toy CNN shown in Figure 7 is built out of four successive convolutional layers, with eight feature maps each, and its competitive counterpart comes with four convolutions layers and 64 feature maps, professional CNNs include 50 or 100 convolutional layers. For networks with this depth a stable training becomes increasingly hard with the standard convolutions defined in (3.28). This issue can be targeted with a residual network, which is built out of convolutional layers combined with skip connections. In the conventions of (1.49) these skip connections come with the additional term

$$x^{(n-1)} \to x^{(n)} = W^{(n)} x^{(n-1)} + b^{(n)} + x^{(\text{earlier layer})} \,, \tag{3.49}$$

where the last term can point to any previous layer. Obviously, we can apply the same structure to the convolutional layer

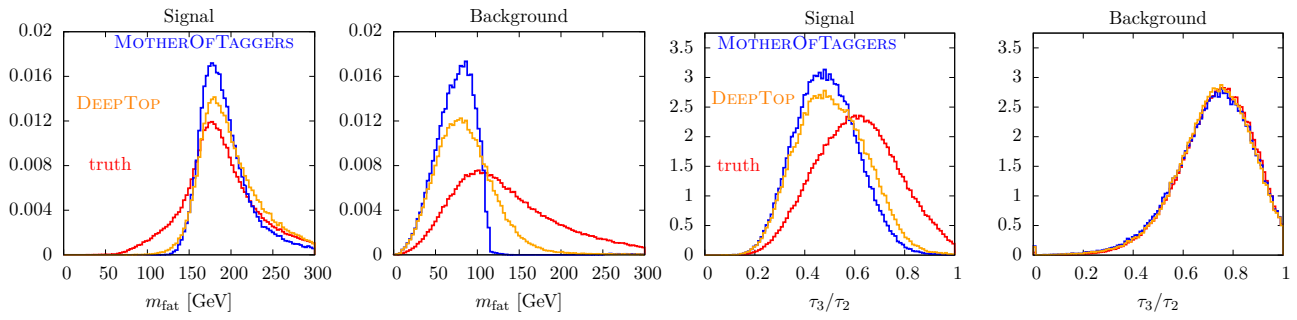


Figure 15: Kinematics observables m_{fat} and τ_3/τ_2 for events correctly determined to be signal or background by the DeepTop CNN and the MotherOfTaggers BDT, as well as Monte Carlo truth. Figure from Ref. [7].

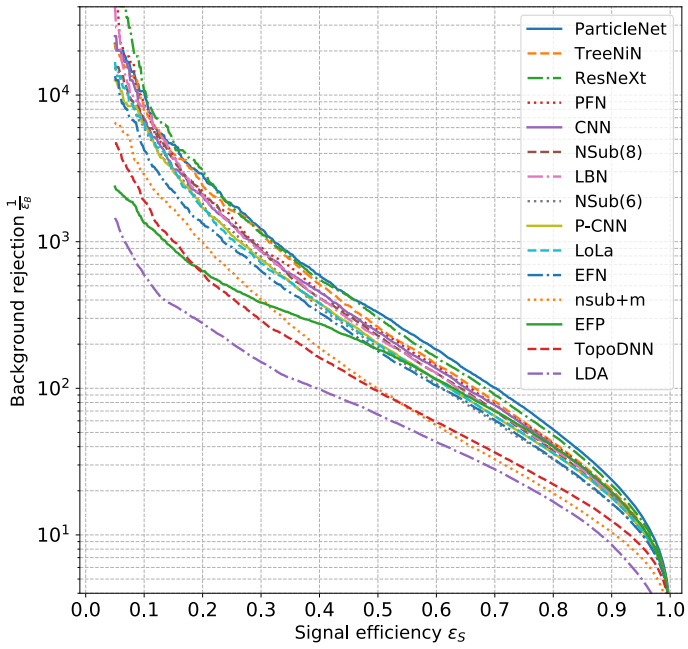


Figure 16: ROC curves for all top-tagging algorithms evaluated on the standard test sample. Figure from Ref. [12].

of (3.27)

$$x_{ij}^{(n)} = \sum_{r,s=0}^{n_{\text{c-size}}-1} W_{rs}^{(n)}\, x_{i+r,j+s}^{(n-1)} + b^{(n)} + x_{ij}^{(\text{earlier layer})} \,. \tag{3.50}$$

Again, we suppress the sum over the feature maps. These skip connections are a standard method to improve the training and stability of very deep networks, and we will come across them again.

In Figure 16 we see that the deep ResNeXt is slightly more powerful than the competitive version of the CNN introduced in Sec. 3.2.1. First, in this study it uses slightly higher resolution with 64×64 pixels. In addition, it is much more complex with 50 layers translating into almost 1.5M parameters, as compared to the 610k parameters of the CNN. Finally, it uses skip connections to train this large number of layers. The sizes of the ResNeXt and the CNN illustrate where some of the power of neural networks are coming from. They can use up to 1.5M network parameters to describe a training dataset consisting of 1M signal and background images each. Each of these sparse calorimeter images includes anything between 20 and 50 interesting active pixels. Depending on the physics question we are asking, the leading 10 pixels might encode most of the information, which translates into 20M training pixels to train 1.5M network parameters. That is quite a complexity, for instance compared to standard fits or boosted decision trees. This complexity also motivates an efficient training, including the back propagation idea, an appropriately chosen loss function, and numerical GPU power. Finally, it explains why some people might be sceptical about the black box nature of neural networks, bringing us back to the question how we can control what networks learn and assign error bars to their output.

3.3 Representing point clouds

In the last section we have argued the case to move from jet images to full event information. Event images work for this purpose, but their increasingly sparse structure cuts into their original motivation. The actual data format behind LHC jets and events are not images, but a set of 4-vectors with additional information on the particle content. These 4-vectors include energy measurements from the calorimeter and momentum measurements from the tracker. The difference between the

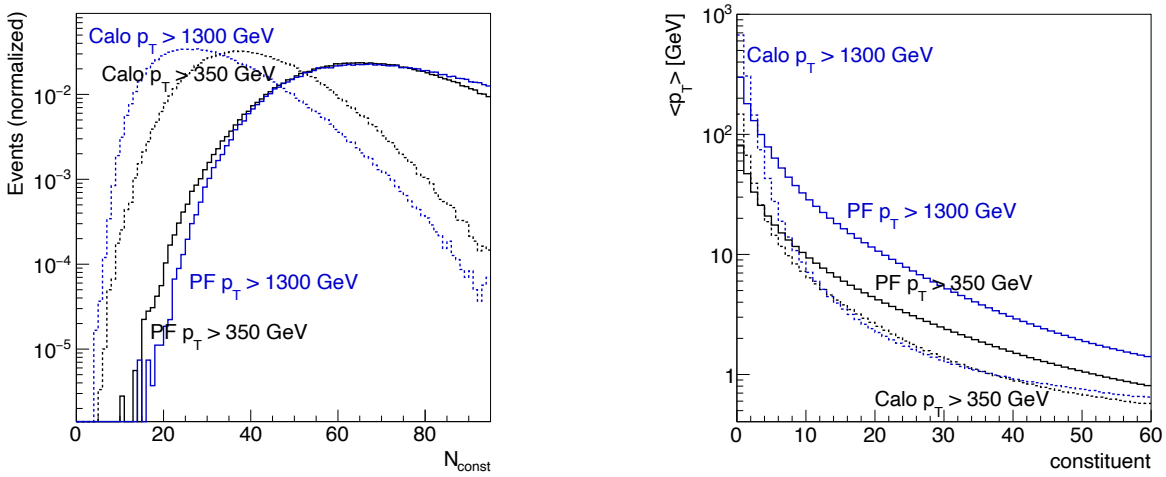


Figure 17: Number of top jet constituents (left) and mean of the transverse momentum (right) of the ranked constituent 4-vectors in (3.51). We show information from jet images (dashed) and from the combined through particle flow (solid). Figure from Ref. [14].

two is that calorimeters observe neutral and charges particles, while tracking provides information on the charged particles with extremely high angular resolution. Calorimeter and tracking information are combined through dedicated particle flow algorithms. Now, we could switch all the way from image recognition to natural language recognition networks, but those do not reflect the main symmetry of 4-vectors or other objects describing LHC collisions, the permutation symmetry. So, before moving to transformers, we will see there are image-based concepts which work extremely well with the LHC data format.

3.3.1 4-Vectors

The basic constituents entering any LHC analysis are a set of C measured 4-vectors sorted by p_T, for example organized as the matrix

$$(k_{\mu,i}) = \begin{pmatrix} k_{0,1} & k_{0,2} & \cdots \\ k_{1,1} & k_{1,2} & \cdots \\ k_{2,1} & k_{2,2} & \cdots \\ k_{3,1} & k_{3,2} & \cdots \end{pmatrix} , \tag{3.51}$$

where for now we ignore additional information on the particle identification. Such a high-dimensional data representation in a general and often unknown space is called a point cloud. If we assume that all constituents are approximately massless, a typical jet image would encode the relative phase space position to the jet axis and the transverse momentum of the constituent,

$$k_{\mu,i} \quad \to \quad \begin{pmatrix} \Delta\eta_i \\ \Delta\phi_i \\ p_{T,i} \end{pmatrix} . \tag{3.52}$$

For the generative networks we will introduce later we implement this transformation as preprocessing, but for the simpler classification network it turns out that we can just work with the 4-vectors given in (3.51).

To illustrate the difference between the image representation and the 4-vector representation we replace our standard top tagging dataset of (3.48) with two distinct samples, corresponding to moderately boosted tops from Standard Model processes and highly boosted tops from resonance searches,

$$p_{T,j} = 350 \ldots 450 \text{ GeV} \qquad \text{and} \qquad p_{T,j} = 1300 \ldots 1400 \text{ GeV} . \tag{3.53}$$

In the left panel of Figure 17 we show the number of calorimeter-based and particle-flow 4-vectors $k_{\mu,i}$ available for our analysis, N_{const}. We see that including tracking information roughly doubles the number of available 4-vectors and

reduces the degradation towards higher boost. In the right panel we show the mean transverse momentum of the p_T-ordered 4-vectors, indicating that the momentum fraction carried by the charged constituents is sizeable. The fact that the calorimeter-based constituents do not get harder for higher boost indicates a serious limitation from their resolution. In practice, we know that the hardest 40 constituents tend to saturate tagging performances, while the remaining entries will typically be much softer than the top decay products and hence carry little signal or background information from the hard process. Comparing this number to the calorimeter-based on particle flow distributions motivates us to go beyond calorimeter images.

As a starting point, we introduce a constituent-based toy tagger which incorporates some basic physics structure. First, we mimic a jet algorithm and multiply the 4-vectors from (3.51) with a matrix C_{ij} and return a set of combined 4-vectors $\tilde{k}_j$ as linear combinations of the input 4-vectors,

$$k_{\mu,i} \xrightarrow{\text{CoLa}} \widetilde{k}_{\mu,j} = k_{\mu,i}\, C_{ij} \; . \tag{3.54}$$

The explicit form of the matrix C defining this combination layer (CoLa) ensures that the $\tilde{k}_j$ include each original momentum k_i as well as a trainable set of $M - C$ linear combinations. These $\tilde{k}_j$ could be analyzed by a standard, fully connected or dense network.

However, we already known that the relevant distance measure between two substructure objects, or any two 4-vectors sufficiently far from the closest black hole, is the Minkowski metric. This motivates a Lorentz layer, which transforms the $\tilde{k}_j$ into the same number of measurement-motivated invariants $\hat{k}_j$,

$$\tilde{k}_j \xrightarrow{\text{LoLa}} \hat{k}_j = \begin{pmatrix} m^2(\tilde{k}_j) \\ p_T(\tilde{k}_j) \\ \square_m w_{jm}^{(E)}\, E(\tilde{k}_m) \end{pmatrix} \; . \tag{3.55}$$

The first two $\hat{k}_j$ map individual $\tilde{k}_j$ onto their invariant mass and transverse momentum, using the Minkowski distance between two four-momenta,

$$d_{jm}^2 = (\tilde{k}_j - \tilde{k}_m)_\mu \; g^{\mu\nu} \; (\tilde{k}_j - \tilde{k}_m)_\nu \; . \tag{3.56}$$

In case the invariant masses and transverse momenta are not sufficient to optimize the classification network, the additional correlator weights w_{jm} are trainable. The third entry in (3.55) constructs a linear combination of all energies, evaluated with one of several possible aggregation functions

$$\square \in \{\text{max}, \text{sum}, \text{mean}, \cdots\} \; . \tag{3.57}$$

A technical challenge related to the Minkowski metric is that it combines two different features: two subjets are Minkowski-close if they are collinear or when one of them is soft ($k_{i,0} \to 0$). Because these two scenarios correspond to different, but possibly overlapping phase space regions, they are hard to learn for the network. To see how the network does and what kind of structures drive the network output, we turn the problem around and ask the question if the Minkowski metric is really the feature distinguishing top decays and QCD jets. This means we define the invariant mass $m(\tilde{k}_j)$ and the distance d_{jm}^2 in (3.55) with a trainable diagonal metric and find

$$\begin{aligned} g = \text{diag}(\quad & 0.99 \pm 0.02, \\ & -1.01 \pm 0.01, -1.01 \pm 0.02, -0.99 \pm 0.02) \; , \end{aligned} \tag{3.58}$$

where the errors are given by five independently trained copies. This means that for top tagging the appropriate space to relate the 4-vector data of (3.54) is defined by the Minkowski metric. Obviously, this is not going to be true for all analysis aspects. For instance, at the event level the rapidities or the scattering angles include valuable information on decay products from heavy resonance compared to the continuum background induced by the form of the parton densities. Still, the LoLa tagger motivates the question how we can combine a data representation as 4-vectors with an appropriate metric for these objects.

In Figure 16 we see that the CoLa-LoLa network does not provide the leading performance. One might speculate that its weakness is that it is a little over-constructed with too much physics bias, with the positive side effect that the network only needs 127k parameters.

Figure 18: Two example representations for the same simple graph.

The fact that we can extract the Minkowski metric as the relevant metric for top-tagging based on 4-vectors leads us to the general concept of graphs. Here we assume that our point-cloud data populates a space for which we can extract some kind of metric or geometry. The optimal metric in this space depends on the task we are training the network to solve. This becomes obvious when we extend the 4-vectors of jet constituents or event objects with entries encoding details from the tracker, like displaced vertices, or particle identification. The question is how we can transform such a point cloud into a structure which allows us to perform the kind of operations we have seen for the CoLa-LoLa tagger, but in an abstract space.

3.3.2 Graph convolutional network

The first problem with the input 4-vectors given in (3.51) is that we do not know which space they live in. The structure that generalizes our CoLa-LoLa approach to an abstract space is, first of all, based on defining nodes, in our case the vectors describing a jet constituent. These nodes have to be connected in some way, defining an edge between each pair of nodes. The LoLa ansatz in (3.55) already assumes that these edges have to go beyond the naive Minkowski metric. The object defined by a set of nodes and their edges is called a graph.

The basic object for analyzing a graph with C nodes is the $C \times C$ adjacency matrix. It encodes the C^2 edges, where we allow for self-interactions of nodes. In the simplest case where we are just interested in the question if two nodes actually define a relevant edge, the adjacency matrix includes zeros and ones. In the graph language such an adjacency matrix defines an undirected — edges do not depend on their direction between two nodes — and unweighted. Even for this simple case this matrix can be useful. Let us look at an example of $C = 5$ nodes with the six edges

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix} . \tag{3.59}$$

This graph is illustrated in Figure 18. Because the graph is undirected, the adjacency matrix is symmetric. Only the first node has a self-interaction, which we count twice because it can be used in two directions. Now we can compute powers of the adjacency matrix, like

$$A^2 = \begin{pmatrix} 6 & 3 & 1 & 1 & 3 \\ 3 & 3 & 0 & 2 & 1 \\ 1 & 0 & 2 & 0 & 2 \\ 1 & 2 & 0 & 2 & 0 \\ 3 & 1 & 2 & 0 & 3 \end{pmatrix} \quad \text{and} \quad A^3 = \begin{pmatrix} 18 & 10 & 4 & 4 & 10 \\ 10 & 4 & 5 & 1 & 8 \\ 4 & 5 & 0 & 4 & 1 \\ 4 & 1 & 4 & 0 & 5 \\ 10 & 8 & 1 & 5 & 4 \end{pmatrix} . \tag{3.60}$$

The matrix A^n encodes the number of different paths of length n which we can take between the two nodes given by the matrix entry. For instance, we can define four length-3 connections between the node and and itself, two different loops each with two directions. Once we have defined a set of edges we can use the existence of an edge, or a non-zero entry in A, to define neighboring nodes, and neighboring nodes is what we need for operations like filter convolutions, the basis of a graph-convolutional network.

Once we have defined our set of nodes and their adjacency matrix, the simplest way to use a filter is to go over all nodes and train a universal filter for their respective neighbors. We already know that our nodes are not just a simple number, but a collection of different features. In that case we can define each node with a feature vector $x_i^{(k)}$ for the nodes $i = 1 \dots C$. In analogy to the feature maps of (3.28) we can then define a filter $W_j^{(kl)}$ where j goes through the neighboring nodes of i and matrix entries match the size of the central and neighboring feature vectors. This means neighboring pixels of (3.28) become nodes with edges, and feature maps turn into feature vectors. A convolutional network working on one node now returns

$$x_i'^{(k)} = \sum_{\text{features } l} \sum_{\text{neighbors } j} W_{ij}^{(kl)}\, x_j^{(l)} \equiv \sum_{\text{features } l} \sum_{\text{nodes } j} W_{ij}^{(kl)} A_{ij}\, x_j^{(l)} \,, \tag{3.61}$$

where in the second form we have used the adjacency matrix to define the neighbors. When using such graph convolutions we can add a normalization factor to this adjacency matrix. Obviously, we also need to apply the usual non-linear activation unction, so the network will for instance return $\text{ReLU}(x_i')$. Finally, for the definition of the universal filter it will matter how we order the neighbors of each node.

Instead of following the convolution explicitly, as in (3.61), we can define a more general transformation than in (3.28), namely a vector-valued function of two feature vectors $x_{i,j}$, and with the same number of dimensions as the feature vector x_i,

$$x_i' = \sum_{\text{neighbors } j} W_\theta(x_i, x_j) \tag{3.62}$$

The sum runs over the neighboring nodes, and we omit the explicit sum in feature space. If we consider the strict form of (3.61) a convolutional prescription, and its extension to $W_{ij}^{(kl)} \to W_{ij}^{(kl)}(x_i, x_j)$ as its generalization, the form in (3.62) is often referred as the most general message passing. Looking at this prescription and comparing it for instance with (3.55), it is not clear why we should sum over the neighboring nodes, so we can define more generally

$$\boxed{x_i' = \square_j\, W_\theta(x_i, x_j)} \,, \tag{3.63}$$

with the aggregation functions $\square_j$ defined in (3.57). The corresponding convolutional layer is called an edge convolution. Just like a convolutional filter, the function W is independent of the node position i, which means it will scale as economically as the CNN. The form of W allows us to implement symmetries like

$$\begin{aligned}
&W_\theta(x_i, x_j) = W_\theta(x_i - x_j) && \text{translation symmetry} \\
&W_\theta(x_i, x_j) = W_\theta(|x_i - x_j|) && \text{rotation symmetry} \\
&W_\theta(x_i, x_j) = W_\theta(x_i, x_i - x_j) && \text{translation symmetry conditional on center.}
\end{aligned} \tag{3.64}$$

The most important symmetry for applications in particle physics is the permutation symmetry of the constituents in a jet or the jets in an event. The edge convolution is symmetric as long as W_θ does not spoil such a symmetry and the aggregation function is chosen like in (3.63).

Going back to the point clouds describing LHC jets or events, we first need to transform the set of possibly extended 4-vectors into a graph with nodes and edges. Obviously, each extended 4-vector of an input jet can become a node described by a feature vector. Edges are defined in terms of an adjacency matrix. We already know that we will modify the adjacency matrix as the initial form of the edges through edge convolutions, so we can just define a reasonably first set of neighbors for each node. We can do that using standard nearest-neighbor algorithms, defining the input to the first edge-convolution layer. When stacking edge convolutions, each layer produces a new set of feature vectors or nodes. This leads to a re-definition of the adjacency matrix after each edge convolution, removing the dependence on the ad-hoc first choice. This architecture is referred to as a Dynamic Graph Convolutional Neural Network (DGCNN). The structure of the edge convolution is shown in the left panel of Figure 19. It combines the nearest-neighbor definition for the graph input with a series of linear edge convolutions and a skip connection as introduced in (3.49).

One reason to introduce the dynamic GCN is that it provides the best jet tagging results in Figure 16. The ParticleNet tagger is based on the third ansatz for the filter function in (3.64) with the simple linear combination

$$\square_j\, W_\theta(x_i, x_j) = \text{mean}_j \Big[\theta_{\text{diff}} \cdot (x_i - x_j) + \theta_{\text{local}} \cdot x_i\Big] \tag{3.65}$$

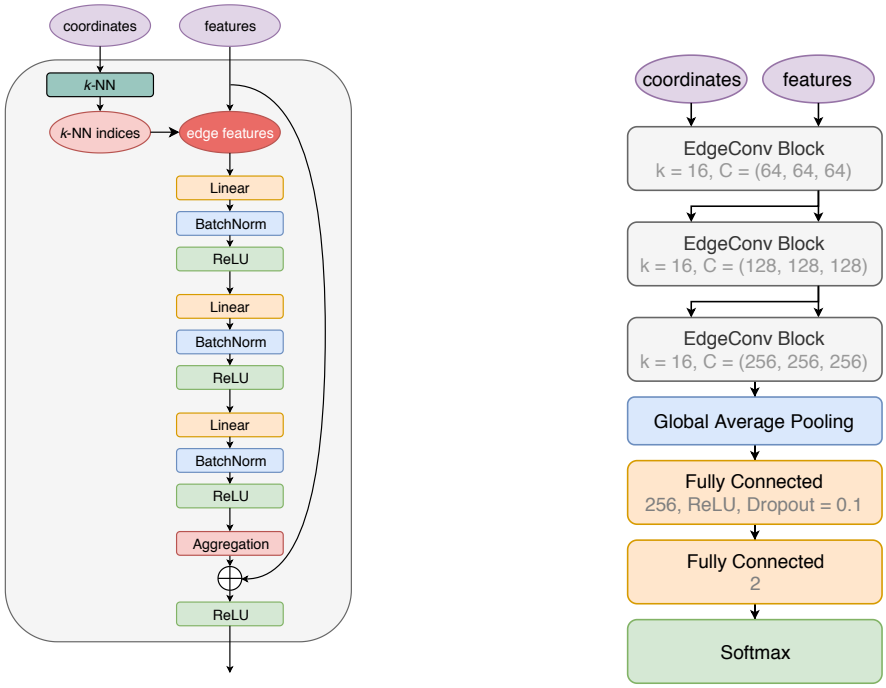


Figure 19: Architectures of an edge convolution block (left) and the ParticleNet implementation for jet tagging. In the right panel, k is the number of nearest neighbors considered and C the number of channels per edge convolution layer. Figure from Ref. [15].

This is the form of the linear convolution referred to in the left panel of Figure 19. Batch normalization is a way to improve the training of deep networks in practice. It evaluates the inputs to a given network layer for a minibatch defined in (1.56) and changes their normalization to mean zero and standard deviation one. It is known to improve the network training, even though there seems to be no good physics or other reason for that improvement.

In the right panel of Figure 19 we show the architecture of the network. After the edge convolutions we need to collect all information in a single vector, which in this case is constructed by average-pooling over all nodes for each of the channels. The softmax activation function of the last layer is the multi-dimensional version of the sigmoid defined in (3.25), required for a classification network. It defines a vector of the same size as its input, but such that all entries are positive and sum to one,

$$\text{Softmax}_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}} \; . \tag{3.66}$$

If we only look at our standard classification setup with two network outputs, where the first gives a signal probability, the softmax function becomes a scalar sigmoid

$$\text{Softmax}_1(x) = \frac{e^{x_1}}{e^{x_1}+e^{x_2}} = \frac{1}{1+e^{x_2-x_1}} \equiv \text{Sigmoid}\,(x_2-x_1) \; , \tag{3.67}$$

for the difference of the inputs, as defined in (3.25). Just as we can use the cross entropy combined with a sigmoid layer to train a network to give us the probability of a binary classification, we can use the multi-class cross entropy combined with the softmax function to train a network for a multi-label classification. Maybe I will show this in an updated version of these notes, but not in the first run.

The fact that for top tagging the GCN outperforms all its competitors indicates that graphs with their permutation invariance are a better representation of jet than images. The number of network parameters is 500k, similar in size to the CNN used for the top-tagging challenge, but leading to the best tagging performance of all architectures.

3.3.3 Transformer

Now that the graph network can construct an appropriate task-dependent space for neighboring nodes, we can tackle the second problem with the input 4-vectors given in (3.51), namely that we do not know how to order permutation-invariant nodes. We can use graphs to solve this problem by removing the adjacency matrix and instead relating all nodes to all nodes, constructing a fully connected graph. A modern alternative, which ensures permutation invariance and can be applied to point clouds, are transformers. Their origin is language analysis, where they provide a structure to analyze how words compose sentences in different languages, where the notion of neighboring works does not really mean anything. Their defining building block is attention, or more specifically, self-attention. It is the natural extension of an adjacency matrix, as shown in (3.59), where a zero means that no information from that node will enter the graph analysis. Self-attention allows an element to assign learned weights to other elements, or, mathematically, a square matrix with an appropriate normalization. These weights define how much 'attention' is placed on those elements whenever our ML-task requires us to define a relation between them.

We motivate the construction of self-attention using a toy model in representation space. The goal of our construction is to describe and then learn some kind of relation between two entries of an input vector or sequence x, and then construct a representation of x_i using these relations.

First, we define an input entry x_i and represent it by a normalized query entry

$$x_i \longrightarrow q = \frac{x_i}{|x|} \,. \tag{3.68}$$

It represents the input x_i as a unit vector in a compact latent space. Next, we need a set of vectors for which we analyse the relations to q, the so-called value vectors v. If they form an orthonormal basis, we can write our input vector as

$$q = \sum_j (q \cdot v_j)\, v_j \equiv \sum_j a_j v_j \qquad \text{with} \qquad a_j = (q \cdot v_j) \,. \tag{3.69}$$

The scalar product a_j represents the strength of the connection between the query vector and a given value vector v_j. However, when implementing this method as a trained network we should not require a normalized basis. Instead, we replace the value basis by another basis of so-called key vectors,

$$q = \sum_j (q \cdot k_j)\, v_j \qquad \text{or} \qquad a_j = (q \cdot k_j) \,. \tag{3.70}$$

As an example, the keys for an orthogonal, but not normalized basis are

$$k_j = \frac{v_j}{v^2} \,. \tag{3.71}$$

Using our freedom in choosing the keys, we then transform the input x_i into a representation z_i, defined as as

$$\boxed{x_i \longrightarrow z_i = \sum_j (q \cdot k_j)\, v_j} \,. \tag{3.72}$$

This representation takes into account the relation to all other entries of our input vector or sequence. Because of the sum over the value basis, it is explicitly permutation invariant. The transformer-encoder is only trained together with the respective network, but we can assume that the network training will construct an orthogonal basis of key and value vectors to make optimal use of the information encoded in the data. The situation for the network is more challenging, when the value representation v_j does not cover the full dimensionality of x_i. In this case, the transformer has to use the downstream training task to decide which information it should remove by constructing scalar products $(q \cdot k_j) \approx 0$, because the dimensionality of the vector k and the maximum index j no longer match.

Let us move on to the proper single-headed self-attention illustrated in Figure 20. Let us assume that we are again analyzing C jet constituents, so the vector x_1 describes the phase space position for the first constituent in an unordered list. If we stick to an image-like representation, the complete vector x will be C copies of the 3-dimensional phase space vector given in (3.52). First, we generalize (3.68) and define a proper query representation of x_1 in a general latent space through a learned weight matrix W^Q,

$$q_1 = W^Q x_1 \,. \tag{3.73}$$

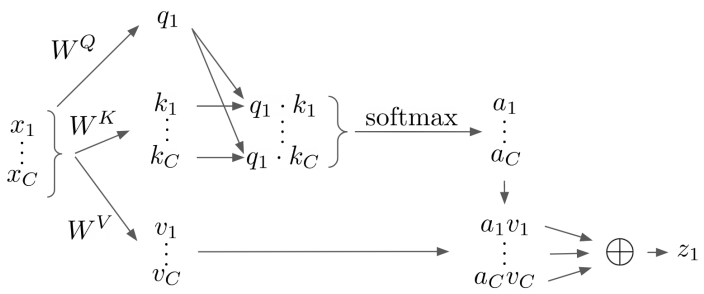


Figure 20: Illustration of single-headed self-attention. Figure from Ref. [16].

If we cover all constituents, W^Q becomes a block-diagonal matrix of size $3C$. To relate all constituents $x_1 \ldots x_C$ to x_1 we define a learned key matrix

$$\begin{pmatrix} k_1 \\ \vdots \\ k_C \end{pmatrix} = W^K \begin{pmatrix} x_1 \\ \vdots \\ x_C \end{pmatrix} \tag{3.74}$$

We use a scalar product to project all keys $k_1 \ldots k_C$ onto q_1. Modulo some details this defines a set of 3-dimensional vectors in analogy to (3.70),

$$\begin{pmatrix} a_1^{(1)} \\ \vdots \\ a_C^{(1)} \end{pmatrix} = \text{Softmax} \begin{pmatrix} (q_1 \cdot k_1) \\ \vdots \\ (q_1 \cdot k_C) \end{pmatrix} \qquad \text{with} \quad a_j^{(1)} \in [0,1] \qquad \sum_j a_j^{(1)} = 1 \; , \tag{3.75}$$

all in reference to x_1. It gives us a quadratic attention matrix, similar to the adjacency matrix (3.59) of a graph,

$$\text{Softmax}\,(q_i \cdot k_j) \equiv a_j^{(i)} \neq a_i^{(j)} \equiv \text{Softmax}\,(q_j \cdot k_i) \; . \tag{3.76}$$

Finally, we transform the complete set of inputs $x_1 \ldots x_C$ into a latent value representation, in analogy to the constrained query form of (3.73), but allowing for full correlations,

$$\begin{pmatrix} v_1 \\ \vdots \\ v_C \end{pmatrix} = W^V \begin{pmatrix} x_1 \\ \vdots \\ x_C \end{pmatrix} \; , \tag{3.77}$$

through the learned matrix W^V. Generalizing from x_1 to x_j gives us the output vector for the transformer-encoder layer in analogy to (3.72)

$$\boxed{z_i = \sum_{j=1}^{3C} a_j^{(i)} v_j = \sum_j \text{Softmax}_j \Big[(W^Q x_i) \cdot (W^K x_j) \Big] \, (W^V x)_j} \; . \tag{3.78}$$

In this formula that the matrices W do not have to be quadratic and can define internal representations Wx with any number of dimensions. The size of W^Q defines the dimension of the output vector z.

A practical problem with the self-attention described above is that each element tends to attend dominantly to itself, which means that in (3.75) the diagonal entries $a_j^{(j)}$ dominate. This numerical problem can be cured by extending the network to multiple heads, which means we perform several self-attention operations in parallel, each with separate learned weight matrices, and concatenate the outputs before applying a final linear layer. This might seem not efficient in computing time, but in practice we can do the full calculation for all constituents, all attention heads, and an entire batch in parallel with tensor operations, so it pays off.

Because a transformer can be viewed as a preprocessor for the jet constituents, enforcing permutation invariance before any kind of NN application, we can combine with it any other preprocessing step. For instance, we know that constituents

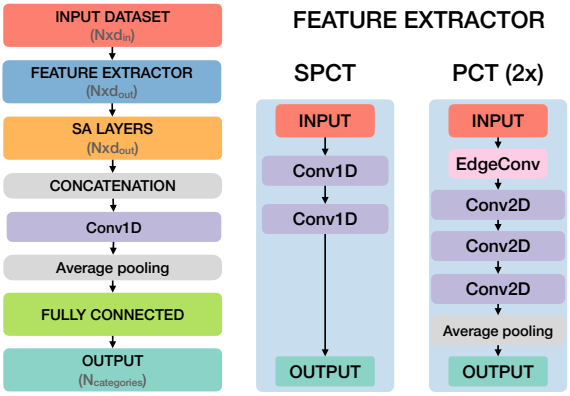


Figure 21: Network architecture and feature extractors for the top tagging application of the point cloud transformer. Figure from Ref. [17].

with small p_T just represent noise, either from QCD or from pileup, and they should have no effect on the physics of the subjet constituents. We can implement this constraint in a IR-safe transformer, where we add a correction factor to (3.75),

$$a_j^{(i)} = \text{Softmax}\left[(q_i \cdot k_j) + \beta \log p_T\right] . \tag{3.79}$$

For small p_T the second contribution ensures that constituents with $p_T \to 0$ do not contribute to the attention weights a_j. In addition, we replace $z_j \to p_{T,j} z_j$ in the transformer output.

The transformer-encoder layer can then be used as part of different network architectures, for example to analyze jets. The output z of a stack of transformer layers can be fed into a classification network directly, or it can be combined with the input features x, in the spirit of the skip connections, to define an offset-attention. In Figure 21 we illustrate the combination of the transformer with two different feature extractors, a set of 1-dimensional convolutional layers running over the features x of each constituent (SPCT), and an edge convolution as introduced in Sec. 3.3.2 (PCT). The edge convolution uses a large number of $k = 20$ nearest neighbors and is followed by 2-dimensional convolutions. The output of the transformer layers is then concatenated to an expanded feature dimension and fed through a fully connected network to provide the usual classification output. This way the SPCT is an transformer-enhanced fully connected network and the PCT combines a simplified GCN structure with a transformer-encoder. The performance of the simple SCPT network matches roughly the standard CNN or LoLa results shown in Figure 16, but with only 7k network parameters. The PCT performs almost as well as the leading ParticleNet architecture, but with only 200k instead of almost 500k network parameters. So while transformers with their different learned matrices appear parameter-intensive, they are actually efficient in reducing the size of the standard networks while ensuring permutation invariance as the key ingredient to successful jet tagging. The challenge of the transformer preprocessing is their long training time.

3.3.4 Deep sets

Motivated by the same argument of permutation invariance as graphs and transformers, another approach to analyzing point cloud data is based on the mathematical observation that we can approximate any observable of 4-vectors using a combination of per-particle mappings and a continuous pooling function,

$$\{k_{\mu,i}\} \to F_\theta \left[\sum_i \phi_\theta(k_{\mu,i})\right] \tag{3.80}$$

where $\phi \in \mathbb{R}^\ell$ is a latent space representation of each input 4-vector or extended particle information. Latent spaces are abstract, intermediate spaces constructed by neural networks, and the power of deep sets lies in complex latent representations which allow us to limit the second step to a sum. By some kind of dedicated requirement, they organize the relevant information. As a simple example, note that a product of two nodes is just their sum in logarithmic space.

As a side remark, the observable F can be turned infrared and collinear safe by replacing $\phi(k_{\mu,i}) \to p_{T,i}\phi$, where ϕ now only depends on the angular information of the k_μ. This is the same strategy as the IR-safe transformer in (3.79).

Such an IR-safe energy flow network (EFN) representation is an additional restriction, which means it will weaken the distinguishing power of the discriminative observable, but it will also make it consistent with perturbative QFT. In Tab. 1 we show a few such representations.

A particle flow network (PFN) implementation of the deep sets architecture is, arguably, the simplest way to analyze point clouds by using two networks. The first network constructs the latent representations respecting the permutation symmetry of the inputs. It is a simple, fully connected network relating the 4-vectors and the particle-ID, for the PFN-ID version, to a per-vector ℓ-dimensional latent representation $\phi_\theta(k_\mu) \in \mathbb{R}^\ell$. The second, also fully connected network, sums the ϕ_θ for all 4-vectors just like the graph aggregation function in (3.63), and feeds them through a fully connected classification network with a softmax or sigmoid activation function as its last layer. The entire classification network is trained though a cross-entropy loss.

For the competitive top tagging results shown in Figure 16 the energy flow network (EFN) and the particle flow network (PFN) only require 82k parameters in the two fully connected networks, with a 256-dimensional latent space. The suppression of QCD jets for a given top tagging efficiency is roughly 20% smaller when we require soft-collinear safety from the EFN rather than using the full information in the PFN.

Just like for the graph network in Sec. 3.3.2 it is also clear how one would add information on the identified particles in a jet or an event. The question is how to best add additional entries to the k_μ introduced in (3.51). In analogy to the preprocessing of jet images, described in Sec. 3.2.1, we would likely use the η and ϕ coordinates relative to the jet axis, combined with the (normalized) transverse momentum p_T. A fourth entry could then be the mass of the observed particle, if non-zero. The charge could simply be a fifth entry added to k_μ. Particle identification would then tell us if a jet constituent is a

$$\gamma, e^\pm, \mu^\pm, \pi^0, \pi^\pm, K_L, K^\pm, n, p \cdots \tag{3.81}$$

One way to encode such categorical data would be to assign a number between zero and one for each of these particles and add a sixth entry to the 4-momentum. This is equivalent to assigning the particle code an integer number and then normalizing this entry of the feature vector. To learn the particle-ID the network then learns the ordering in the corresponding direction.

The problem with this encoding becomes obvious when we remind ourselves that a loss function forms a scalar number out of the feature vector, so the network needs to learn some kind of filter function to extract this information. This means combining different categories into one number is not helping the network. Another problem is a possible bias from the network architecture or loss function, leading to an enhanced sensitivity of the network to larger values of the particle-ID vector. Some ranges in the ID-directions might be preferred by the network, bringing us back to permutation invariance, the theme of this section. Instead, we can encode the particle-ID in a permutation-invariant manner, such that a simple unit vector in all directions can extract the information. Using one-hot encoding the phase space vector of (3.52) becomes

$$k_\mu \quad \to \quad \begin{pmatrix} \Delta\eta \\ \Delta\phi \\ p_T \\ m \\ \delta_{\text{ID}=\gamma} \\ \delta_{\text{ID}=e} \\ \vdots \end{pmatrix} . \tag{3.82}$$

The additional dimensions can only have entries zero and one. This method looks like a waste of dimensions, until we remind ourselves that a high-dimensional feature space is actually a strength of neural networks and that this kind of information is particularly easy to extract and de-correlate from the feature space.

		ϕ	F
mass	m	p^μ	$F(x^\mu) = \sqrt{x^\mu x_\mu}$
multiplicity	n_{PF}	1	$F(x) = x$
momentum dispersion	$p_T D$	(p_T, p_T^2)	$F(x,y) = \sqrt{y}/x$

Table 1: Example for observables decomposed into per-particle maps ϕ and functions F according to (3.80). In the last column, the arguments of F are placeholders for the summed output of ϕ. Table from Ref. [18].

3.3.5 CNNs to transformers and more

After introducing a whole set of network architectures, developed for image or language applications, we can illustrate their differences slightly more systematically. By now, we know to consider our input data as elements of a point cloud $x_{i,j}$, which can be represented as nodes of a graph or similar construction. It is convenient to divide their properties into node features and edge features e_{ij}. A neural network is a trainable function F_θ or ϕ_θ. The symmetry properties of a network are determined by an aggregation function $\square$ as introduced in (3.57), typically a sum or a modification of a sum. Finally, we denote the generic activation function as ReLU.

Convolutional networks, including graph-convolutional networks, are defined by (3.27). We can write this transformation as

$$x'_i = \text{ReLU } F_\theta \left[x_i, \square_{j \in N} c_{ij} \phi_\theta(x_j) \right] \ , \tag{3.83}$$

where constant values c_{ij} imply the crucial weight sharing. The aggregation combines the central node with a neighborhood N. In physics application we often choose this neighborhood small, because physics effects are usually local in an appropriate space.

For self-attention, the basis of transformers defined in Sec. 3.3.3, we replace the c_{ij} by as a general link function of x_i and x_j, with access to the feature structure

$$x'_i = \text{ReLU } F_\theta \left[x_i, \square_{j \in N} a(x_i, x_j, e_{ij}) \phi_\theta(x_j) \right] \ . \tag{3.84}$$

Many transformers cover all nodes instead of a local neighborhood, but this approach is expensive to train and not always required. We refer to them as masked transformers.

Even more generally, we can avoid the factorization into linking relation and the node features and replace it with a general learned function. This defines message passing networks, introduced in (3.62),

$$x'_i = \text{ReLU } F_\theta \left[x_i, \square_{j \in N} \phi_\theta(x_i, x_j, e_{ij}) \right] \ . \tag{3.85}$$

Finally, we can also understand the efficiency gain of the deep sets architecture in (3.80) in this form,

$$x'_i = \text{ReLU } F_\theta \left[\square_j \phi_\theta(x_j) \right] \ , \tag{3.86}$$

Here there are not edges, which means nodes have to be encoded without any notion of an underlying space or metric, and the function F_θ operates on the aggregation of the latent representations of the nodes. Not using edges can simplify the scaling of the network for a large number of objects.

3.4 Symmetries and contrastive learning

After observing the power of networks respecting permutation invariance, we can think about symmetries and neural network architectures more generally. Since Emmy Noether we know that symmetries are the most basic structure in physics, especially in particle physics. LHC physics is defined by an extremely complex symmetry structure, starting with LHC data, the detector geometry, to the relativistic space-time symmetries and local gauge symmetries defining the underlying QFT. If we want to use machine learning we need to embrace these symmetries. For instance, the jet or calorimeter images introduced in Sec. 3.2 are defined in rapidity vs azimuthal angle, observables inspired by Lorentz transformations and by the leading symmetry of the LHC detectors. The preprocessing of jet images exploits the rotation symmetry around the jet axis. The rather trivial case of permutation symmetry is driving all of the network architectures presented in Sec. 3.3. Theoretical invariances under infrared transformations motivated the IR-safe transformer and the energy flow networks.

From a structural perspective, there are two ways symmetries can affect neural networks. First, we call a network equivariant (for mathematicians) of icovariant (for physicists) if for a symmetry operation S and a network output $f_\theta(x)$ we have

$$f_\theta(S(x)) = S(f_\theta(x)) \ . \tag{3.87}$$

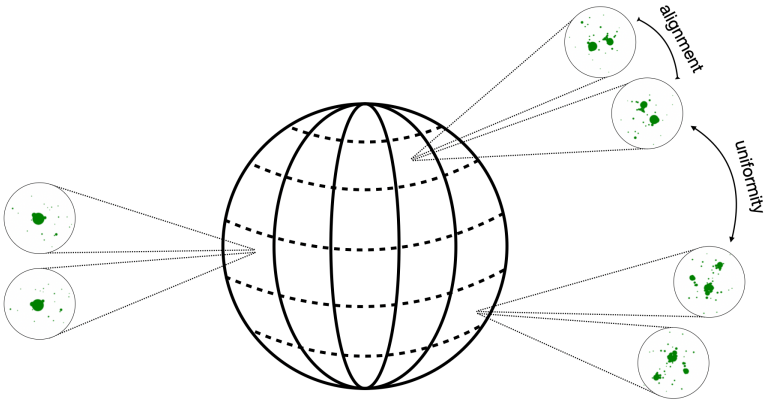


Figure 22: Illustration of the uniformity and alignment concepts behind the contrastive learning. Figure from Ref. [16].

It means we can recover a symmetry operation S on the data x as a symmetry operation of an covariant network output. For example, if we shift all pixels in an input image to a CNN in one direction, the training of the convolutional filters will not change and the feature maps inside the network will just be shifted as well, so the CNN is covariant under translations. We would not want the network to be completely invariant to translations, because the main question in jet tagging is always where features appear relative to a fixed jet axis. An equivalent definition of an covariant network is that for two different inputs with the same output, two transformed inputs also give the same output,

$$f_\theta(x_1) = f_\theta(x_2) \qquad \Rightarrow \qquad f_\theta(S(x_1)) = S(f_\theta(x_1)) = S(f_\theta(x_2)) = f_\theta(S(x_2)) \; . \tag{3.88}$$

A stronger symmetry requirement is an invariant network, namely

$$\boxed{f_\theta(S(x)) = f_\theta(x)} \; . \tag{3.89}$$

The rotation step of the jet image preprocessing in Sec. 3.2.1 ensures the hard way that the classification network is rotation-invariant. Similarly, the goal of the Lorentz layer in (3.55) is to guarantee a Lorentz-invariant definition of the classification network. Finally, the goal of the graph-inspired networks discussed in Sec. 3.3 is to provide a network architecture which guarantees that the classification outcome is permutation invariant.

For particle physics applications, it would be extremely cool to have a training procedure and loss function which ensures that the latent or representation space is invariant to pre-defined symmetries or invariances and still retains discriminative power. This means we could implement any symmetry requirement either through symmetric training data or through symmetric data augmentations. A way to achieve this is contrastive learning of representations (CLR). The goal of such a network is to map, for instance, jets x_i described by their constituents to a latent or representation space,

$$f_\theta : \quad x_i \to z_i \; , \tag{3.90}$$

which is invariant to symmetries and theory-driven augmentations, and remains discriminative for the training and test datasets.

As usually, our jets x_i are described by n_C massless constituents and their phase space coordinates given in (3.52), so the jet phase space is $3n_C$-dimensional. We first take a batch of jets $\{x_i\}$ from the dataset and apply one or more symmetry-inspired augmentations to each jet. This generate an augmented batch $\{x'_i\}$. We then pair the original and augmented jets into two datasets

$$\begin{aligned} &\text{positive pairs:} && \{(x_i, x'_i)\} \\ &\text{negative pairs:} && \{(x_i, x_j)\} \cup \{(x_i, x'_j)\} \quad \text{for} \quad i \neq j \; . \end{aligned} \tag{3.91}$$

Positive pairs are symmetry-related and negative are not. The goal of the network training is to map positive pairs as close together in representation space as possible, while keeping negative pairs far apart. Simply pushing apart the negative

pairs allows our network to encode any kind of information in their actual position in the latent space. Labels, for example indicating if the jets are QCD or top, are not used in this training strategy, which is referred to as self-supervised training. The trick to achieve this split in the representation space is to replace vectors z_i, which the network outputs, by their normalized counterparts,

$$f_\theta(x_i) = \frac{z_i}{|z_i|} \qquad \text{and} \qquad f_\theta(x_i') = \frac{z_i'}{|z_i'|} \; . \tag{3.92}$$

This way the jets are represented on a compact hypersphere, on which we can define the similarity between two jets as

$$s(z_i, z_j) = \frac{z_i \cdot z_j}{|z_i||z_j|} \in [-1, 1] \; , \tag{3.93}$$

which is just the cosine of the angle between the jets in the latent space. This similarity is not a proper distance metric, but we could instead define an angular distance in terms of the cosine, such that it satisfies the triangle inequality. Based on this similarity we construct a contrastive loss. It can be understood in terms of alignment versus uniformity on the unit hypersphere, illustrated in Figure 22. Starting with negative pairs, a loss term like

$$\boxed{\mathcal{L}_{\text{CLR}} \supset \sum_{j \neq i \in \text{batch}} \left[e^{s(z_i, z_j)} + e^{s(z_i, z_j')} \right]} \tag{3.94}$$

will push them apart, preferring $s \to -1$. On the compact hypersphere they cannot be pushed infinitely far apart. This means our loss will be minimal when the unmatched jets are uniformly distributed. To map the jets to such a uniform distribution in a high-dimensional space, the mapping will identify features to discriminate between them and map them to different points. Next, we want the loss to become minimal when for the positive pairs all jets and their respective augmented counterparts are aligned in the same point, $s \to 1$.

$$\mathcal{L}_{\text{CLR}} \supset - \sum_{i \in \text{batch}} s(z_i, z_i') \tag{3.95}$$

This additional condition induces the invariance with respect to augmentations and symmetries. The combined contrastive loss is given by a sum of the two corresponding conditions

$$\begin{aligned} \mathcal{L}_{\text{CLR}} &= - \sum_{i \in \text{batch}} s(z_i, z_i') + \sum_{i \in \text{batch}} \log \sum_{j \neq i \in \text{batch}} \left[e^{s(z_i, z_j)} + e^{s(z_i, z_j')} \right] \\ &= - \sum_{i \in \text{batch}} \log \frac{e^{s(z_i, z_i')}}{\sum_{j \neq i \in \text{batch}} \left[e^{s(z_i, z_j)} + e^{s(z_i, z_j')} \right]} \; , \end{aligned} \tag{3.96}$$

The first term sums over all positive pairs and reaches its minimum in the alignment limit. The negative pairs contribute to the second term, and the expression in brackets is summed over all negative-pair partners of a given jet. If such a solution were possible, the loss would force all individual distances to their maximum. For the spherical latent space the best the network can achieve is the smallest average s-value for a uniform distribution.

Next, we can apply contrastive learning to LHC jets, to see if the network learns an invariant representation and defines some kind of structure through the uniformity requirement. Before applying symmetry transformations and augmentations we start preprocessing the jets as described in Sec.3.2.1 and ensure that the p_T-weighted centroid is at the origin in the $\eta - \phi$ plane. Now, rotations around the jet axes are a very efficient symmetry we can impose on our representations. We apply them to a batch of jets by rotating each jet by angles sampled from $0 \; ... \; 2\pi$. Such rotations in the $\eta - \phi$ plane are not Lorentz transformations and do not preserve the jet mass, but for narrow jets with $R \lesssim 1$ the corrections to the jet mass can be neglected. As a second symmetry we implement translations in the $\eta - \phi$ plane. Here, all constituents in a jet are shifted by the same random distance, where shifts in each direction are limited to $-1 \; ... \; 1$.

In addition to (approximate) symmetries, we can also use theory-inspired augmentations. QFT tells us that soft gluon radiation is universal and factorizes from the hard physics in the jet splittings. To encode this invariance in the latent representation, we augment our jets by smearing the positions of the soft constituents, in η and ϕ using from a Gaussian distribution centered on the original coordinates

$$\eta' \sim \mathcal{N}\left(\eta, \frac{\Lambda_{\text{soft}}}{p_T}\right) \qquad \text{and} \qquad \phi' \sim \mathcal{N}\left(\phi, \frac{\Lambda_{\text{soft}}}{p_T}\right) \; , \tag{3.97}$$

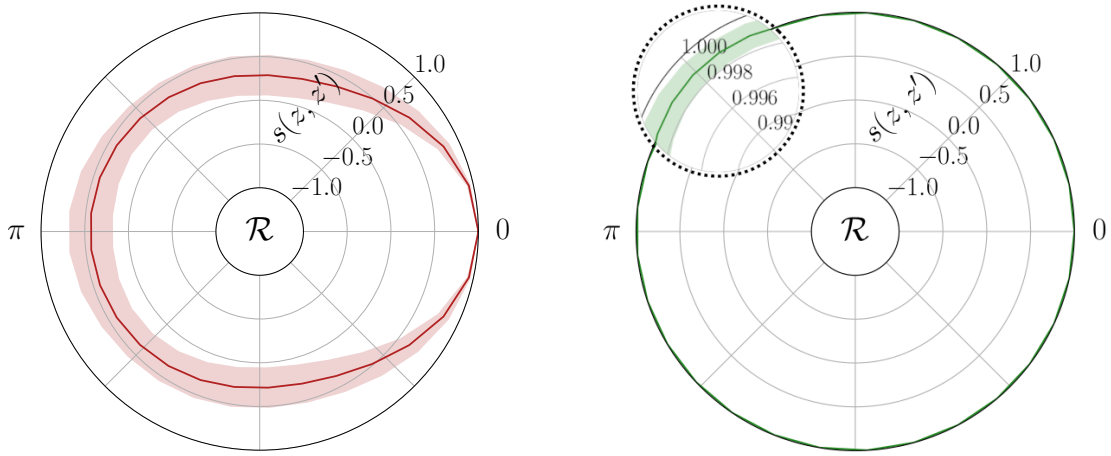


Figure 23: Visualization of the rotational invariance in representation space, where $s(z, z') = 1$ indicates identical representations. We compare JetCLR representations trained without (left) and with (right) rotational transformations. Figure from Ref. [16].

with a p_T-suppression in the variance relative to $\Lambda_{\text{soft}} = 100$ MeV. Secondly, collinear splittings lead to divergences in perturbative QFT. In practice. they are removed through the finite angular resolution of a detector, which cannot resolve two constituents with $p_{T,a}$ and $p_{T,b}$ at vanishing $\Delta R_{ab} \ll 1$. We introduce collinear augmentations by splitting individual constituents such that the total p_T in an infinitesimal region of the detector is unchanged,

$$p_T \to p_{T,a} + p_{T,b} \qquad \text{with} \qquad \eta_a = \eta_b = \eta \qquad \phi_a = \phi_b = \phi \; . \tag{3.98}$$

These soft and collinear augmentations will enforce a learned IR-safety in the jet representation, unlike the modified versions of the transformer or the EFPs.

Finally, we include the permutation symmetry among the constituents, for instance through a transformer-encoder. The combination of contrastive loss and a permutation-invariant network architecture defines the JetCLR approach.

For a test sample of jets we can check if our JetCLR network indeed encodes symmetries. To illustrate the encoded rotation symmetry we show how the representation is invariant to actual rotations of jets. We start with a batch of 100 jets, and produce a set of rotated copies for each jet, with rotation angles evenly spaced in $0 \ldots 2\pi$. We then pass each jet and its rotated copy through the JetCLR network, and calculate their similarity in the latent representation, (3.93). In Figure 23 we show the mean and standard deviation of the similarity as a function of the rotation angle without and with the rotational symmetry included in the JetCLR training. In the left panel the similarity varies between 0.5 and 1.0 as a function of the rotation angle, while in the right panel the JetCLR representation is indeed rotationally invariant. From the scale of the radial axis $s(z, z')$ we see that the representations obtained by training JetCLR with rotations are very similar to the original jets.

Before using the JetCLR construction for an explicit task, we can can analyze the effect of the different symmetries and theory augmentations using a linear classifier test (LCT). For this test we train a linear neural network with a binary cross-entropy loss to distinguish top and QCD jets, while our JetCLR training does not know about these labels. This means

Augmentation	$\epsilon^{-1}(\epsilon_s = 0.5)$	AUC
none	15	0.905
translations	19	0.916
rotations	21	0.930
soft+collinear	89	0.970
combined	181	0.979

Table 2: JetCLR classification results for different symmetries and augmentations and $S/B = 1$. The combined setup includes translation and rotation symmetries, combined with soft and collinear augmentations. Table from Ref. [16].

the LCT tells us if the uniformity condition has encoded some kind of feature which we assume to be correlated with the difference between QCD and top jets. A high AUC from the LCT points to a well-structured latent representation. From first principles, it is not clear which symmetries and augmentations work best for learning representations. In Tab. 2 we summarize the results after applying rotational and translational symmetry transformations and soft+collinear augmentations. It turns our that, individually, the soft+collinear augmentation works best. Translations and rotations are less powerful individually, but the combination defines by far the best-ordered representations.

4 Non-supervised classification

Searches for BSM physics at the LHC traditionally start with a theory hypothesis, such that we can compare the expected signature with the Standard Model background prediction for a given phase space region using likelihood methods. The background hypothesis might be defined through simulation or through an extrapolation from a background into a signal region. This traditional approach has two fundamental problems which we will talk about in this section and which will take us towards a more modern interpretation of LHC searches.

First, we can generalize classification, for example of LHC events, to the situation where our training data is measured data and therefore does not come with event-wise labels. However, a standard assumption of essentially any experimental analysis is that signal features are localized in phase space, which means we can define background regions, where we assume that there is no signal, and signal regions, where there still are background, but accompanied by a sizeable signal fraction. This leads us to classification based on weakly supervised learning.

Second, any searches based on hypothesis testing does not generalize well in model space, because we can never be sure that our model searches actually cover an existing anomaly or sign of physics beyond the Standard Model. We can of course argue that we are performing such a large number of analyses that it is very unlikely that we will miss an anomaly, but this approach is at the very least extremely inefficient. We also need to remind ourselves that ruling out some parameter space in a pre-defined model is not really a lasting result. This means we should find ways to identify for example anomalous jets or events in the most model-independent way. Such a method can be purely data-driven or rely on simulations, but in either case we only work with background data to extract an unknown signal, a method refereed to as unsupervised learning.

4.1 Classification without labels

Until now we have trained classification networks on labelled, pure datasets. Following the example of top tagging in Sec. 3.2.3, such training data can be simulations or actual data which we understand particularly well. The problem is that in most cases we do not understand a LHC dataset well enough to consider it fully labelled. What is much easier is to determine the relative composition of such a dataset with the help of simulations, for instance 80% top jets combined with 20% QCD jets on the one hand and 10% top jets combined with 90% QCD jets on the other.

Let us look to the phase space distributions of signal and background jets or events $p_{S,B}(x)$ again. What we actually observe are not labelled signal and background samples, but two mixed samples with global signal fractions $f_{1,2}$ and background fractions $1-f_{1,2}$ in our two training datasets. The mixed phase space densities $p_{1,2}$ are related to the pure densities as

$$\begin{aligned}
&\begin{pmatrix} p_1(x) \\ p_2(x) \end{pmatrix} = \begin{pmatrix} f_1 & 1-f_1 \\ f_2 & 1-f_2 \end{pmatrix} \begin{pmatrix} p_S(x) \\ p_B(x) \end{pmatrix} \\
\Leftrightarrow \quad &\begin{pmatrix} p_S(x) \\ p_B(x) \end{pmatrix} = \frac{1}{f_1-f_2} \begin{pmatrix} 1-f_2 & f_1-1 \\ -f_2 & f_1 \end{pmatrix} \begin{pmatrix} p_1(x) \\ p_2(x) \end{pmatrix} \\
\Rightarrow \quad &\frac{p_S(x)}{p_B(x)} = \frac{(1-f_2)p_1(x) + (f_1-1)p_2(x)}{-f_2 p_1(x) + f_1 p_2(x)} \; .
\end{aligned} \tag{4.1}$$

The last line implies that, if we know $f_{1,2}$ and can also extract the mixed densities $p_{1,2}(x)$ of the two training datasets, we can compute the signal and background distributions and with them the likelihood ratio as the optimal test statistics.

Next, we remember the Neyman-Pearson theorem and ask the question how the likelihood ratios for signal vs background classification, p_S/p_B, and the separation of the two mixed samples, p_1/p_2, are related. This will lead us to a shortcut in

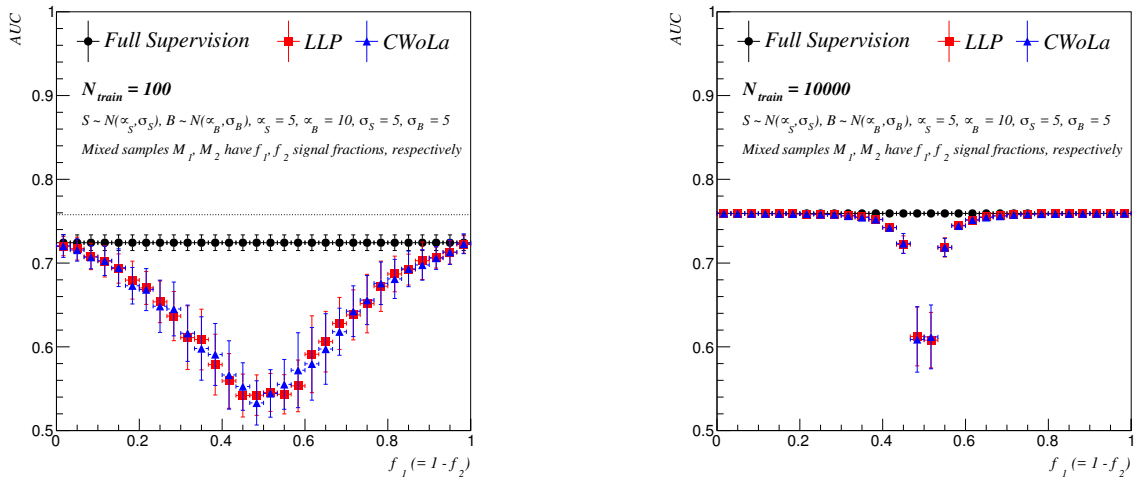


Figure 24: Performance of the CWoLa method for the double-Gaussian toy example as a function of the signal fraction in one of the training datasets for 100 (left) and 10000 (right) training points. Figure from Ref. [19].

our classification task,

$$\frac{p_1(x)}{p_2(x)} = \frac{f_1 p_S(x) + (1-f_1) p_B(x)}{f_2 p_S(x) + (1-f_2) p_B(x)} = \frac{f_1 \dfrac{p_S(x)}{p_B(x)} + 1 - f_1}{f_2 \dfrac{p_S(x)}{p_B(x)} + 1 - f_2}$$

$$\frac{d}{d(p_S/p_B)} \frac{p_1(x)}{p_2(x)} = \frac{f_1 \left[f_2 \dfrac{p_S(x)}{p_B(x)} + 1 - f_2 \right] - f_2 \left[f_1 \dfrac{p_S(x)}{p_B(x)} + 1 - f_1 \right]}{\left[f_2 \dfrac{p_S(x)}{p_B(x)} + 1 - f_2 \right]^2} = \frac{f_1 - f_2}{\left[f_2 \dfrac{p_S(x)}{p_B(x)} + 1 - f_2 \right]^2} \,. \tag{4.2}$$

The sign of this derivative is a global $\text{sign}(f_1 - f_2)$ and does not change if we vary the likelihood ratios. This means the two likelihood ratios are linked through a monotonous function, which means that we can exchange them as a test statistics at no cost. In other words, if we are interested in an optimal classifier we can skip the translation into p_S/p_B and just use the classifier between the two mixed samples, p_1/p_2, instead. This is an attractive option, because it means that we do not need to know $f_{1,2}$ if we are just interested in the likelihood ratio.

While this classification without labels (CWoLa) does not require use to know the signal and background fractions and is therefore, strictly speaking, an unsupervised method, we always work under the assumption that we have a background-dominated and a signal-dominated dataset. Moreover, any such analysis tool needs to be calibrated. If the classification outcome is not a signal or background probability, as discussed in Sec. 3, we need to define a working point for our classifier and determine its signal and background efficiencies. From (4.1) we see that we only need two samples with known signal and background fractions to extract $p_S(x)$ and $p_B(x)$ for any given working point.

We illustrate the unsupervised method using the original toy model of a 1-dimensional, binned observable x and Gaussian signal and background distributions $p_{S,B}(x) = \mathcal{N}(x)$. We have three options to train a classifier on this dataset, which means we can

1. compute the full supervised likelihood ratio $p_S/p_B(x)$ from the truth distributions;
2. use (4.1) to compute the likelihood ratio from $p_{1,2}(x)$ and known label proportions $f_{1,2}$ (LLP);
3. follow the CWoLa method and use $p_1/p_2(x)$ to separate signal and background instead of the two samples.

The AUC values for the three methods are shown in Figure 24 as a function of the signal fraction of one of the two samples, chosen the same as the background fraction for the second sample. The horizontal dashed line indicates the fully-supervised AUC with infinite training statistics. By construction, the AUC for full supervision is independent of f_1. The weakly supervised and unsupervised methods start coinciding with the fully supervised method as long as we stay

away from

$$f_1 \approx f_2 \approx \frac{1}{2} \,. \tag{4.3}$$

We can see the problem with this parameter point in (4.1), where a matrix inversion is not possible. Similarly, in (4.2) the linear dependence of the two likelihood ratios vanishes in the same parameter point.

4.2 Anomaly searches

The main goal of the LHC is to search for new and interesting effects which would require us to modify our underlying theory. If BSM physics is accessible to the LHC, but more elusive than expected, we should complement our hypothesis-based search strategies with more general approaches. For example, we can search for jets or events which stick out by some measure, turning them into prime candidates for a BSM physics signature. If we assume that essentially all events or jets at the LHC are described well by the Standard Model and the corresponding simulations, such an outlier search is equivalent to searching for the most non-SM instances. The difference between these two statements is that the first is based on unlabeled data, while the second refers to simulations and hence pure SM samples. They are equivalent if the data-based approach effectively ignores the outliers in the definition of the background-like sample and its implicitly underlying phase space density.

4.2.1 (Variational) autoencoders

For the practical task of anomaly searches, autoencoders (AEs) are the simplest unsupervised ML-tool. In the AE architecture an encoder compresses the input data into a bottleneck, encoding each instance of the dataset in an abstract space with a dimension much smaller than the dimension of the input data. A decoder then attempts to reconstruct the input data from the information encoded in the bottleneck. This architecture is illustrated in the upper left panel of Figure 25 and works for jets using an image or 4-vector representation. Without the bottleneck the AE could just construct an identity mapping of a jet on itself; with the bottleneck this is not possible, so the network needs to construct a compressed representation of the input. This should work well if the ambient or apparent dimensionality of our data representation is larger than the intrinsic or physical dimensionality of the underlying physics.

The loss function for such an AE can be the MSE defined in (1.44), quantifying the agreement of the pixels in an input jet image x and the average jet image output x', summed over all pixels. Formally, this sum is just an expectation value over a batch of jet images sampled from the usual data distribution $p_{\text{data}}(x)$,

$$\boxed{\mathcal{L}_{\text{MSE}} = \Big\langle |x - x'|^2 \Big\rangle_{p_{\text{data}}}} \,. \tag{4.4}$$

The idea behind the anomaly search is that the AE learns to compress and reconstruct the training data very well, but different test data passed through the AE results in a large loss.

Because AEs do not induce a structure in latent space, we have no choice but to use this reconstruction error or loss also as the anomaly score. This corresponds to a definition of anomalies as an unspecific kind of outliers. Using the latent loss as an anomaly score leads to a conceptual weakness when we switch the standard and anomalous physics hypothesis. For example, QCD jets with a limited underlying physics content of massless parton splittings can be described by a small bottleneck. An AE trained on QCD jets will not be able to describe top-decay jets with their three prongs and massive decays. Turning the problem around, we can train the AE on top jets, in which case it will be able to describe multi-prong topologies as well as frequently occurring single-prong topologies in the top sample. QCD jets are now just particularly simple top jets, which means that they will not lead to a large anomaly score. This bias towards identifying more complex data is also what we expect from the standard use of a bottleneck for data compression.

Moving beyond purely reconstruction-based autoencoders, variational autoencoders (VAEs) add structure to the latent bottleneck space, again illustrated in Figure 25. In the encoding step, a high-dimensional data representation is mapped to a low-dimensional latent distribution, from which the decoder learns to generate the original, high-dimensional objects. The latent bottleneck space then contains structured information which might not be apparent in the high-dimensional input representation. This means the VAE architecture consists of a learnable encoder with the output distribution $z \sim p_\theta^{\text{E}}(z|x)$

mapping the phase space x to the latent space z, and a learnable decoder with the output distribution $x \sim p_\theta^{\text{D}}(x|z)$. The loss combines two terms

$$\boxed{\mathcal{L}_{\text{VAE}} = \Big\langle -\big\langle \log p_\theta^{\text{D}}(x|z)\big\rangle_{p_\theta^{\text{E}}(z|x)} + \beta_{\text{KL}} D_{\text{KL}}[p_\theta^{\text{E}}(z|x), p_{\text{latent}}(z)]\Big\rangle_{p_{\text{data}}}} \,. \tag{4.5}$$

The first terms is the reconstruction loss, where we compute the likelihood of the output of the decoder $p_\theta^{\text{D}}(x|z)$ and given the encoder $p_\theta^{\text{E}}(z|x)$, evaluated on batches sampled from p_{data}. We get back the MSE version serving as the AE reconstruction loss in (4.4) when we approximate the decoder output $p_\theta^{\text{D}}(x|z)$ as a Gaussian with a constant width. The second term is a latent loss, comparing the latent-space distribution from the encoder to a prior $p_{\text{latent}}(z)$, which defines the structure of the latent space. A more systematic deviation of the VAE loss and its link to a maximized likelihood will follow in Sec. 5.1.

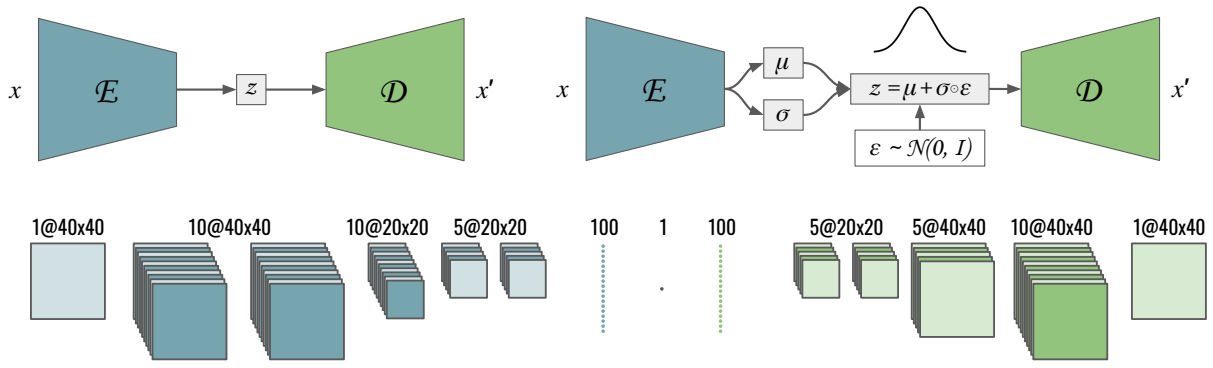
Because of the structured bottleneck we now have a choice of anomaly scores, either based on the reconstruction or the latent space. This way the VAE can avoid the drawbacks of the AE by using an alternative anomaly score to the reconstruction error. We can compare the alternative choices using the standard QCD and top jet-images with a simple preprocessing described in Sec. 3.2.1. To force the network to learn the class the jet belongs to and to be able to visualize this information, we restrict the bottleneck size to one dimension. In the VAE case this gives us a useful probabilistic interpretation, since the mapping to the encoded space is performed by a probability distribution $p_\theta^{\text{E}}(z|x)$. To simplify the training of the classifier, we train on a sample with equal numbers of top and QCD jets. We are interested in three aspects of the VAE classifiers: (i) performance as a top-tagger, (ii) performance as a QCD-tagger, (iii) stable encoding in the latent space. Some results are shown in Figure 26. Compared to the AE results, the regularisation of the VAE latent space generates as relatively stable and structured latent representation, even converging to a representation in which the top jets are clustered slightly away from the QCD jets. On the other hand, the separation in the VAE latent space is clearly not sufficient to provide a competitive anomaly score.

4.2.2 Normalized autoencoder

The problem with AE and VAE applications to anomaly searches leads us to the question what it means for a jet to be anomalous. While the AE only relies on a bottleneck and the compressibility of the jet features, the VAE adds the notion of a properly defined latent space. A normalised autoencoder (NAE) goes one step further and constructs a statistically interpretable latent space, a bridge from an out-of-distribution definition of anomalies to density-based anomalies.

We start by introducing energy-based models (EBMs), a class of models which can estimate probability densities especially well. They introduce a so-called energy function, which is then minimized through the training. This energy function can be chosen as any non-linear mapping of a phase space point to a scalar value,

$$E_\theta(x) : \mathbb{R}^D \to \mathbb{R} \,, \tag{4.6}$$



(a) Encoder and decoder

Figure 25: Architectures used for AE (left) and VAE (right) networks. All convolutions use a 5x5 filter size and all layers use PReLU activations. Downsampling from 40x40 to 20x20 is achieved by average pooling, while upsampling is nearest neighbor interpolation. Figure from Ref. [20].

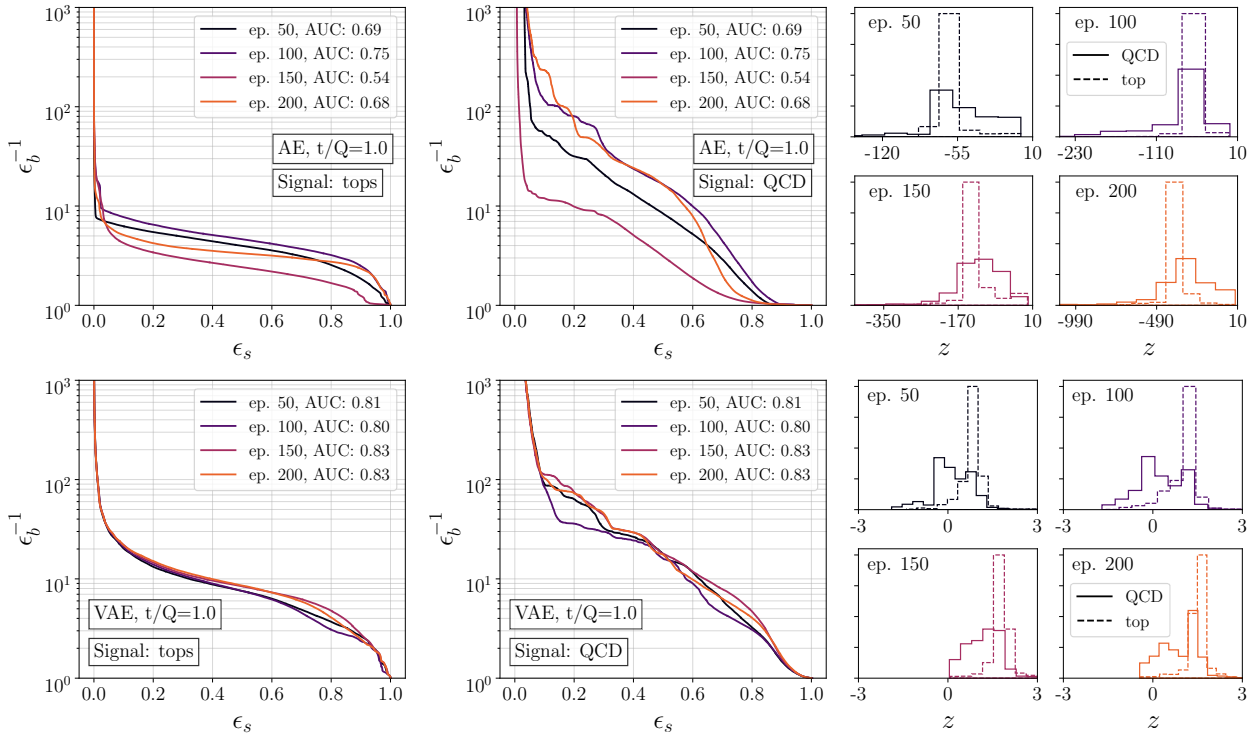


Figure 26: Symmetric performance of the toy-AE (upper) and toy-VAE (lower). In the large panels we show the ROC curves for tagging top and QCD as signals. In the small panels we show the distributions of top and QCD jets in the 1-dimensional latent space. Figure from Ref.[20].

where D is the dimensionality of the phase space. The idea is that the energy summarizes the properties of the system or the minimization in a single real number. The mapping of a complex and often noisy distribution to a single system energy can be motivated from statistical physics.

The EBM uses this energy function to define the loss based on a Boltzmann distribution describing the probability density over phase space

$$p_\theta(x) = \frac{e^{-E_\theta(x)}}{Z_\theta} \qquad \text{with} \qquad Z_\theta = \int_x dx e^{-E_\theta(x)} \; , \tag{4.7}$$

with the partition function Z_θ. The main practical feature of a Boltzmann distribution is that low-energy states have the highest probability. Furthermore, the Boltzmann distribution has two advantages: first, it can easily be integrated, which means we can at least hope to compute Z_θ. Second, it can be singled out as the probability distribution $p_\theta(x)$ which gives the largest entropy

$$S = -\int dx \; p_\theta(x) \log p_\theta(x) \; , \tag{4.8}$$

translating into a large model flexibility. Going beyond our usual likelihood loss, the EBM loss or loss of our normalized AE is the negative logarithmic probability

$$\boxed{\mathcal{L}_\text{NAE} = -\big\langle \log p_\theta(x) \big\rangle_{p_\text{data}} = \big\langle E_\theta(x) + \log Z_\theta \big\rangle_{p_\text{data}}} \; , \tag{4.9}$$

where we define the total loss as the expectation over the per-sample loss. Unlike for the VAE, the loss really minimizes the posterior of the data distribution to be correct, as a function of the network parameters θ; this is nothing but a likelihood loss. The difference between the EBM and typical implementations of likelihood-ratio losses is that we do not use Bayes' theorem and a prior, so the normalization term depends on θ and becomes part of the training.

To train the network we want to minimize the loss in (4.9), so we have to compute the gradient of the full probability,

$$\begin{aligned} -\nabla_\theta \log p_\theta(x) &= \nabla_\theta E_\theta(x) + \nabla_\theta \log Z_\theta \\ &= \nabla_\theta E_\theta(x) + \frac{1}{Z_\theta} \nabla_\theta \int_x dx e^{-E_\theta(x)} \\ &= \nabla_\theta E_\theta(x) - \int_x dx \frac{e^{-E_\theta(x)}}{Z_\theta} \nabla_\theta E_\theta(x) \\ &= \nabla_\theta E_\theta(x) - \Big\langle \nabla_\theta E_\theta(x) \Big\rangle_{p_\theta} . \end{aligned} \tag{4.10}$$

The first term in this expression can be obtained using automatic differentiation from the training sample, while the second term is intractable and must be approximated, as we will see next.

In the actual minimization we evaluate the gradient of the loss as an expectation value over $p_{\text{data}}(x)$. This allows us to rewrite the gradient of the loss as the difference of two energy gradients

$$\Big\langle -\nabla_\theta \log p_\theta(x) \Big\rangle_{p_{\text{data}}} = \Big\langle \nabla_\theta E_\theta(x) \Big\rangle_{p_{\text{data}}} - \Big\langle \nabla_\theta E_\theta(x) \Big\rangle_{p_\theta} . \tag{4.11}$$

The first term samples from the training data, the second from the model. According to the sign of the energy in the loss function, the contribution from the training dataset is referred to as positive energy and the contribution from the model as negative energy. There are three ways to to look at the second term, induced by the normalization constant Z_θ, in the loss gradient: (i) as a normalization which ensures that the loss vanishes for $p_\theta(x) = p_{\text{data}}(x)$, similar to the usual likelihood ratio loss of (3.14); (ii) as a sampling covering both, the data distribution and the model distribution, a little like a forward and reverse KL-divergence; and (iii) as a background structure into the minimization of the likelihood minimization.

One practical way of sampling from $p_\theta(x)$ is to use Markov-Chain Monte Carlo (MCMC). The NAE uses Langevin Markov Chains, where the steps are defined by drifting a random walk towards high probability points according to

$$x_{t+1} = x_t + \lambda_x \nabla_x \log p_\theta(x) + \sigma_x \epsilon_t \qquad \text{with} \qquad \epsilon_t \sim \mathcal{N}(0,1) . \tag{4.12}$$

Here, λ is the step size and σ the noise standard deviation. When $2\lambda = \sigma^2$ the equation resembles Brownian motion and gives exact samples from $p_\theta(x)$ in the limit of $t \to +\infty$ and $\sigma \to 0$.

Because the EBM only constructs a normalized probability loss based on whatever energy function we give it, we can easily upgrade a standard AE with the standard encoder-decoder structure,

$$f_\theta(x) : \mathbb{R}^D \to \mathbb{R}^{D_z} \to \mathbb{R}^D . \tag{4.13}$$

The training minimizes the per-pixel difference between the original input and its mapping, so we upgrade the AE to a probabilistic NAE by using the MSE as the energy function in (4.6)

$$\boxed{E_\theta(x) = \text{MSE} = |x - f_\theta(x)|^2} . \tag{4.14}$$

By using the reconstruction error as the energy, the model will learn to poorly reconstruct inputs not in the training distribution. This way it guarantees the behavior of the model all over phase space, especially in the region close to but not in the training data distribution. We cannot give such a guarantee for a standard AE, which only sees the training distribution and could assign arbitrary reconstruction scores to data outside this distribution.

As usual, we apply the NAE to unsupervised top tagging, after training on QCD jets only, and vice versa. For the latent space dimension we use $D_z = 3$, which allows us to visualize the latent space nicely. Before starting the NAE training we pre-train the network using the standard AE procedure with the standard MSE loss. In the upper panels of Figure 27 we show a projection of the latent space after training the usual AE. In the left panels we train on the simpler QCD background, which means that the latent space has a simple structure. The QCD jets are distributed widely over the low-energy region, while the anomalous top jets cluster slightly away from the QCD jets. This changes when we train on the more complex top jets, as shown in the right panels. The latent MSE-landscape reflects this complex structure with many minima, and top jets spread over most of the sphere.

In a second step we apply the NAE training. We remind ourselves that it also probes the latent space regions where there is no data, by sampling the model. Indeed, we see in the lower panels of Figure 27 that only the regions populated by

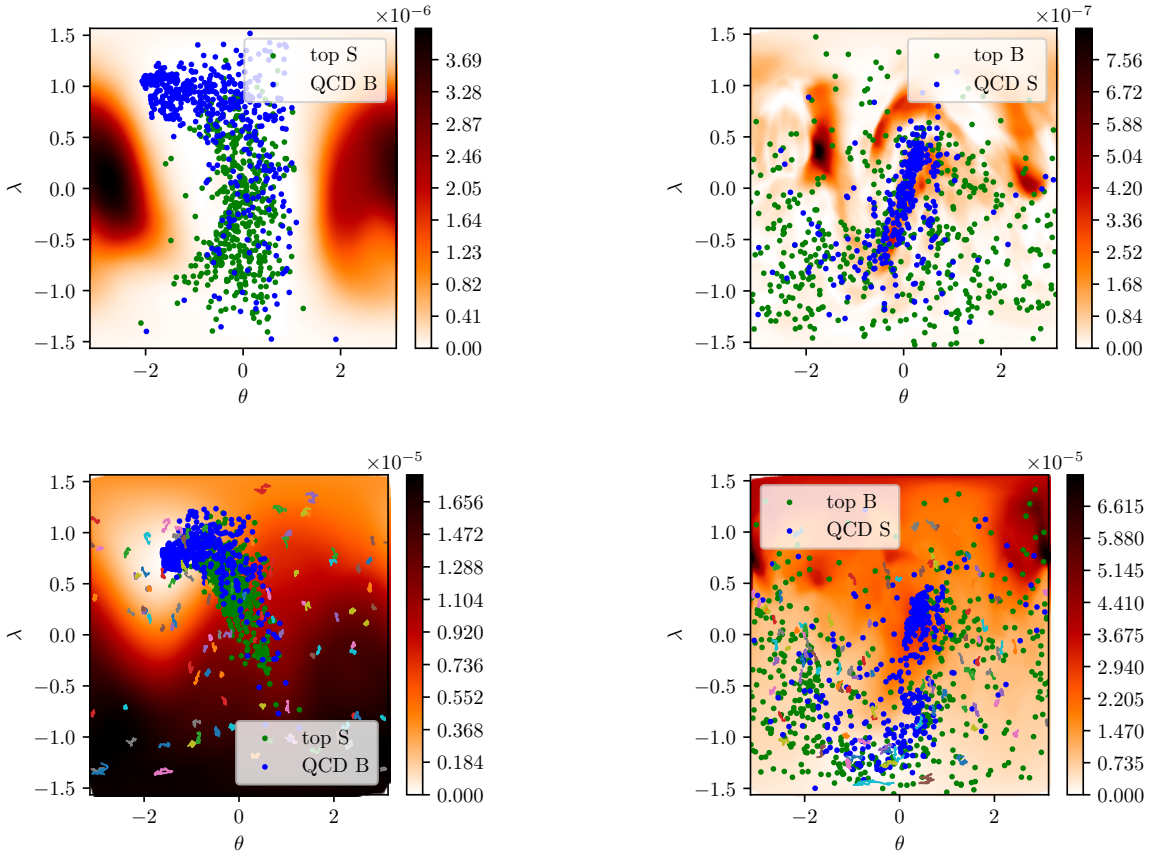


Figure 27: Equirectangular projection of the latent space after pre-training (upper) and after NAE training (lower). The x- and y-axis are the longitude and latitude on the latent sphere. We train on QCD jets (left) and on top jets (right). The lines represent the path of the LMCs in the current iteration. Figure from Ref. [21].

training data remain with a low MSE. The sampling procedure has shaped the decoder manifold to correctly reconstruct only training jet images. For both training directions, the Markov chains move from a uniform distribution to mostly cover the region with low MSE, leading to an improved separation of the respective backgrounds and signals.

To show how the NAE works symmetrically for anomalous tops and for anomalous QCD jets, we can look at the respective MSE distributions. In the left panel of Figure 28 we first see the result after training the NAE on QCD jets. The MSE values for the background are peaked strongly, cut off below $4 \cdot 10^{-5}$ and with a smooth tail towards larger MSE values. The MSE distribution for top jets is peaked at larger values, and again with an unstructured tail into the QCD region. Alternatively, we see what happens when we train on top jets and search for the simpler QCD jets as an anomaly. In the right panel of Figure 28 the background MSE is much broader, with a significant tail also towards small MSE values. The QCD distribution develops two distinct peaks, an expected peak in the tail of the top distribution and an additional peak under the top peak. The fact that the NAE manages to push the QCD jets towards larger MSE values indicates that the NAE works beyond the compressibility ordering of the simple AE. However, the second peak shows that a fraction of QCD jets look just like top jets to the NAE.

4.3 Phase transitions reloaded

In Section 3.2.2 we have used a simple CNN for classifying the phases of the Ising-type model defined in (3.35). The results there can be corroborated using the more elaborate architectures discussed in 3.3. We leave this task to the interested reader and briefly discuss the application of autoencoders, introduced in Section 4.2, to the classification of the ordered and disordered phase in the Ising model with the Hamiltonian (3.7) on a square lattice with length L. Here we follow [22], for a more comprehensive analysis see [23]. We consider a lattice $L \times L$ with $L = Na$ and $N = 28$.

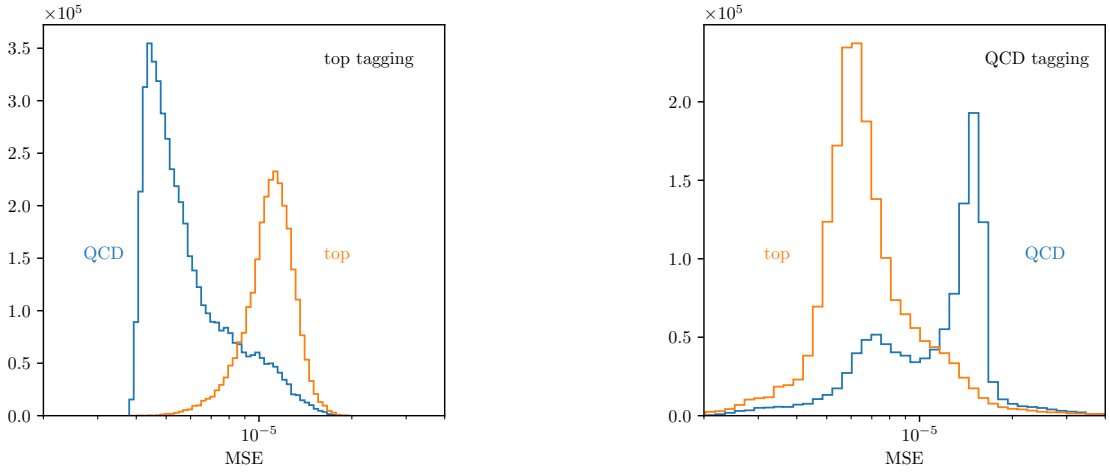


Figure 28: Distribution of the MSE after training on QCD jets (left) and on top jets (right). We show the MSE for QCD jets (blue) and top jets (orange) in both cases. Figure from Ref. [21].

In the following we benchmark the performance of the (V)AEs with that of the principal component analysis (PCA), applied to the present system: we consider the variance of the spin $s = s_1, ..., s_{N^2}$ for a given sample of spin configurations $s^{(1)}, ..., s^{(n)}$,

$$\sum_j (s^{(j)} - \langle s \rangle)^2 \,. \tag{4.15}$$

We project the variance on its largest direction, the first principal component, with the orthogonal linear transformation w,

$$\underset{\|w\|=1}{\text{argmax}} \sum_j \Big((s^{(j)} - \langle s \rangle) \cdot w \Big)^2 \,. \tag{4.16}$$

This transformation can be successively applied to the next largest component. With PCA one constructs a successive diagonalisation of the covariance matrix. Evidently, the principal component is directly related to the (average) expectation value of the spin, the magnetization M defined in (3.39) in Section 3.2.2. We also use kernel PCA, where the spin (data) are first mapped into a kernel space, and in [22] the radial basis function (RBF) kernel was chosen. The correlation between the first principle component and the magnetization M is shown in Figure 29 in the upper panels and constitutes the benchmark result.

In the lower panels we show the results obtained with the standard autoencoder (left) and variational autoencoder (right). Both have one fully connected hidden layer in the encoder and one fully connected hidden layer in the decoder with 256 neurons. The activation function in the final layer is a sigmoid. Figure 29 shows very impressively the impact of the structured bottleneck in the VAE, and the NAE is expected to perform even better (show). Still, both architectures perform worse than the PCA.

Finally, the direct correlation between the latent parameter and the magnetization can be used for a study of the phase diagram of the Ising model. In Figure 30 we show the temperature dependence of the magnetization (order parameter) in comparison to the latent parameter as well as the reconstruction loss. Clearly, the latent parameter also serves as an order parameter.

Seemingly, this simply entails hat VAEs are well-suited for this specific classification task, but do not outperform standard (linear) optimization algorithms. However, this does not take into account the scaling of the algorithms with the size of the data set and the natural applicability of the NNs to non-linear problems.

5 Generation and simulation

In the previous chapters we have discussed mainly classification tasks using modern machine learning, applying all kinds of supervised and unsupervised training. We have seen how this allows us to extract more complete information. Specifi-

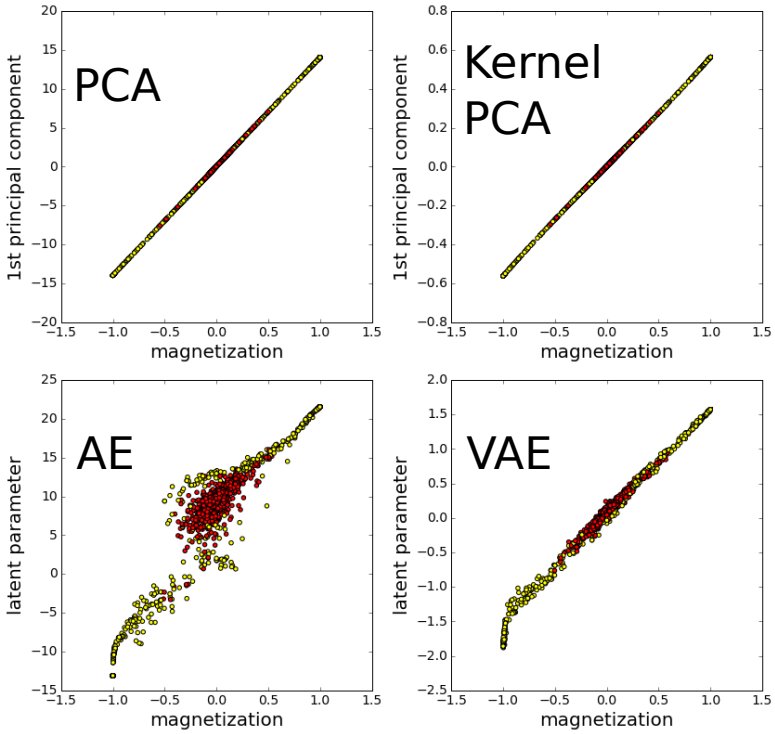


Figure 29: Correlation of 1st principal component or latent parameter versus magnetization for PCA, Kernel PCA, AE and VAE. Res points are in the disordered phase, yellow ones are in ordered one. Figure taken from [22].

cally, it can be used to significantly improve LHC analyses or for detecting phase transitions and suggesting related order parameters or more generically observables.

However, classification is not the same as modern analyses in the sense that particle physics analyses have to be related to some kind of fundamental physics question. This means we can either measure a fundamental parameters of the Standard Model Lagrangian, or we can search for physics beyond the Standard Model. Measuring a Wilson coefficient or coupling of the effective field theory version of the Standard Model is the modern way to unify these two approaches. To interpret LHC measurements in such a theory framework the central tool are simulations — how do we get from a Lagrangian to a prediction for observed LHC events in one of the detectors?

Moreover, in statistical physics or quantum field theory the key task is the generation of the configurations. While this task is readily achieved for the spin systems under consideration so far, very often, physics system that show interesting phenomena are difficult to simulate and are often plagued by (assumed) computational hardness.

The concept which allows us to apply modern machine learning to LHC event generation and simulations as well as the generation of configurations in statistical systems and quantum field theories are generative networks. To train them, we start with a dataset which implicitly encodes a probability density over a physics or phase space. A generative network learns this underlying density and maps it to a latent space from which we can then sample using for example flat or Gaussian random numbers,

$$r \sim p_{\text{latent}}(r) \quad \rightarrow \quad x = f_\theta(r) \sim p_{\text{model}}(x) \approx p_{\text{data}}(x) \; . \tag{5.1}$$

The last step represents the network training, for instance in terms of a variational approximation. A typical latent distribution is the standard multi-dimensional Gaussian,

$$p_{\text{latent}}(r) = \mathcal{N}(0,1) \; . \tag{5.2}$$

Generative networks allow us to produce samples following a learned distribution. The generated data should then have the same form as the training data, in which case the generative network will produce statistically independent samples reproducing the implicit underlying structures of the training data. Because we train on a distribution of events and there

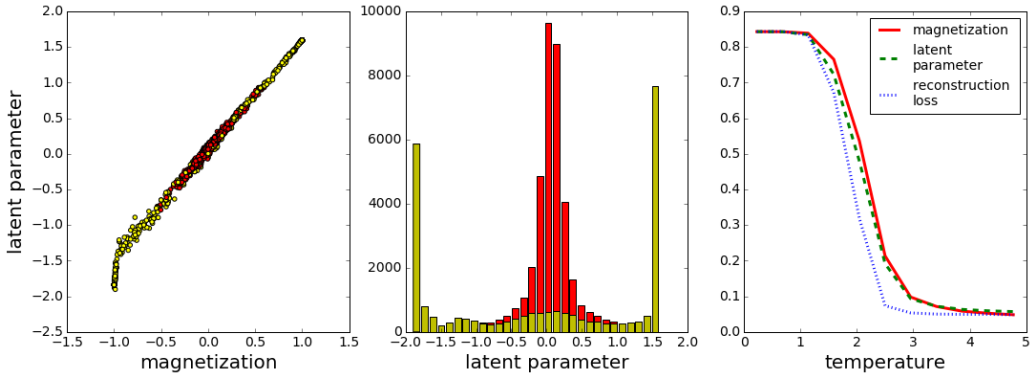


Figure 30: LEFT: Correlation of the latent parameter (VAE) versus the magnetization. MIDDLE: Histogram of the occurrence of latent parameters. The red bars collect the data from the disordered phase, the yellow bars collect the data from the ordered phase. RIGHT: temperature dependence of the magnetization (red, order parameter) in comparison to the latent parameter (green long-dashed) as well as the reconstruction loss (blue, dashed). Figure taken from [22].

are no labels or any other truth information about the learned phase space density, generative network training is considered unsupervised.

If the network is trained to learn a phase space density, we expect generative networks to require us to compare different distributions, for instance the training distributions $p_{\text{data}}(x)$ and the encoded density $p_{\text{model}}(x)$. We already know one way to compare the actual and the modeled phase space density from (3.13). However, the KL-divergence is only one way to compare such a probability distributions and this is part of a much bigger field called optimal transport. We remind ourselves of the definition of the KL-divergence,

$$D_{\text{KL}}[p_{\text{data}}, p_{\text{model}}] = \left\langle \log \frac{p_{\text{data}}(x)}{p_{\text{model}}(x)} \right\rangle_{p_{\text{data}}} = \int dx\ p_{\text{data}}(x)\ \log \frac{p_{\text{data}}(x)}{p_{\text{model}}(x)}\ , \tag{5.3}$$

The KL-divergence between two identical distributions is zero. A crucial disadvantage of this measure is that it is not symmetric, which in the above form means that phase space regions where we do not have data will not contribute to the comparison of the two probability distributions. Two distributions with zero overlap have infinite KL-divergence. If the asymmetric form of the KL-divergence turns out to be a problem, we can easily repair it by introducing the Jensen-Shannon divergence

$$\begin{aligned}
D_{\text{JS}}[p_{\text{data}}, p_{\text{model}}] &= \frac{1}{2}\left[D_{\text{KL}}\left[p_{\text{data}}, \frac{p_{\text{data}}+p_{\text{model}}}{2}\right] + D_{\text{KL}}\left[p_{\text{model}}, \frac{p_{\text{data}}+p_{\text{model}}}{2}\right]\right] \\
&= \frac{1}{2}\left[\int dx\ p_{\text{data}}(x)\ \log \frac{2p_{\text{data}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)} + \int dx\ p_{\text{model}}(x)\ \log \frac{2p_{\text{model}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)}\right] \\
&= \frac{1}{2}\int dx\ \left[p_{\text{data}}(x)\ \log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)} + p_{\text{model}}(x)\ \log \frac{p_{\text{model}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)}\right] + \log 2 \\
&\equiv \frac{1}{2}\left\langle \log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)} \right\rangle_{p_{\text{data}}} + \frac{1}{2}\left\langle \log \frac{p_{\text{model}}(x)}{p_{\text{data}}(x)+p_{\text{model}}(x)} \right\rangle_{p_{\text{model}}} + \log 2\ .
\end{aligned} \tag{5.4}$$

The JS-divergence between two identical distributions also vanishes, but because it samples from both, p_{data} and p_{model}, it will not explode when the distributions have zero overlap. Instead, we find in this limit

$$D_{\text{JS}}[p_{\text{data}}, p_{\text{model}}] \to \frac{1}{2}\int dx\ \left[p_{\text{data}}(x)\ \log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)} + p_{\text{model}}(x)\ \log \frac{p_{\text{model}}(x)}{p_{\text{model}}(x)}\right] + \log 2 = \log 2\ . \tag{5.5}$$

One property, the KL-divergence and the JS-divergence have in common, is that they calculate the difference between two distributions based on the log-ratio of two functional values. Consequently, they reach their maximal values for two distributions with no overlap in x, no matter how the two distributions look. This is counter-intuitive, because a distance measure should notice the difference between two identical and two very different distributions with vanishing overlap,

for instance a two Gaussians vs a Gaussian and a double-Gaussian. This brings us to the next distance measure, which is meant to work horizontally in the sense that it guarantees for example

$$W[p_{\text{data}}(x), p_{\text{model}}(x) = p_{\text{data}}(x-a)] \approx a \; . \tag{5.6}$$

This Wasserstein distance or earth mover distance can be most easily defined for weighted sets of points defining each of the two distributions $p_{\text{data}}(x)$ and $p_{\text{model}}(y)$ in a discretized description

$$M_{\text{data}} = \sum_{i=1}^{N_1} p_{\text{data},i} \delta_{x_i} \qquad \text{and} \qquad M_{\text{model}} = \sum_{j=1}^{N_2} p_{\text{model},j} \delta_{y_j} \; . \tag{5.7}$$

We then define a preserving transport strategy as a matrix relating the two sets, namely

$$\pi_{ij} \geq 0 \qquad \text{with} \qquad \frac{1}{N_2} \sum_j \pi_{ij} = p_{\text{data},i} \qquad \frac{1}{N_1} \sum_i \pi_{ij} = p_{\text{model},j} \; . \tag{5.8}$$

The first normalization condition ensures that all entries in the model distributions j combined with the data entry i reproduce the full data distribution, the second normalization condition works the other way around. We define the distance between the two represented distributions as

$$W[p_{\text{data}}, p_{\text{model}}] = \min_\pi \frac{1}{N_1 N_2} \sum_{i,j} |x_i - y_j| \; \pi_{ij} \; , \tag{5.9}$$

where the minimum condition implies that we choose the best transport strategy. For our example from (5.6) with two identical functions this strategy would give $x_i - y_j = a$. Alternatively, we can write the Wasserstein distance as an expectation value of the distance between two points. This distance has to be sampled over the combined probability distributions and then minimized over the so-called transport plan,

$$W[p_{\text{data}}, p_{\text{model}}] = \min \Big\langle |x-y| \Big\rangle_{p_{\text{data}}(x), p_{\text{model}}(y)} \tag{5.10}$$

From the algorithmic definition we see that computing the Wasserstein distance is expensive and scales poorly with the number of points in our samples. We will see that these three different distances between distributions can be used for different generative networks, to learn and then sample from an underlying phase space distribution.

Finally, if we want to test the performance of a generative network a classifier or discriminator trained to distinguish training data and generated data seems an obvious choice. As a matter of fact, this kind of comparison can already be part of the network training, as we will see in Section 5.2. The reason why we mention this here is that the Neyman-Pearson lemma tells us that in this case the discriminator has to learn the likelihood ratio or a simple variation of it. For a generative network over phase space this means we can extract the scalar field $p_{\text{model}}(x)/p_{\text{data}}(x)$ as the unsupervised counterpart to the agreement between a regression network and its training data from (2.10).

5.1 Variational autoencoders

We studied autoencoders and variational autoencoders already in Section 4.2.1, with the idea to map a physics space onto itself with an additional bottleneck (AE) and an induced latent space structure in this bottleneck (VAE). Looking at their network architecture from a generative network point of view, we can also sample from this latent space with corresponding random numbers, for instance with a multi-dimensional Gaussian distribution. In that case the decoder part of the VAE will generate events corresponding to the properties translated from the input phase space to the latent space through the encoder. This means we already know one simple generative network.

In Section 4.2.1 we introduced the two-term VAE loss function somewhat ad hoc and without any reference to a probability distribution or a stochastic justification. Let us now tackle it a little more systematically. We start by assuming that we know the data x and implicitly its probability distribution $p_{\text{data}}(x)$. We also want to enforce a given latent distribution $p_{\text{latent}}(r)$. In that case the the encoder generates according to the conditional probability $p_{\text{model}}(r|x)$, while the generator is described by $p_{\text{model}}(x|r)$.

We start with the encoder training, which should approximate something like $p_{\text{model}}(r|x) \sim p_{\text{latent}}(r)$. However, this condition cannot be the final word, since it missed the conditional structure. Instead, we need to construct the reference distribution $p(r|x)$ from Bayes' theorem (1.7),

$$p(r|x)\; p_{\text{data}}(x) = p_{\text{model}}(x|r)\; p_{\text{latent}}(r)\;. \tag{5.11}$$

On the left side we have to complete the reference distribution by the data-defined $p_{\text{data}}(x)$, while on the right side we combine the conditional generator with the known latent distribution.

To train the encoder we resort to the variational approximation from Section 1.5, specifically (1.60). The goal is to construct a network function $p_{\text{model}}(r|x)$ which approximates $p(r|x)$, just like in (1.60). As before, we construct this approximation using the KL-divergence of (1.18),

$$\begin{aligned}
D_{\text{KL}}[p_{\text{model}}(r|x), p(r|x)] &= \left\langle \log \frac{p_{\text{model}}(r|x)}{p(r|x)} \right\rangle_{p_{\text{model}}(r|x)} \\
&= \left\langle \log p_{\text{model}}(r|x) - \log \frac{p_{\text{model}}(x|r) p_{\text{latent}}(r)}{p_{\text{data}}(x)} \right\rangle_{p_{\text{model}}(r|x)} \\
&= -\Big\langle \log p_{\text{model}}(x|r) \Big\rangle_{p_{\text{model}}(r|x)} + \Big\langle \log p_{\text{model}}(r|x) - \log p_{\text{latent}}(r) \Big\rangle_{p_{\text{model}}(r|x)} + \log p_{\text{data}}(x) \\
&= -\Big\langle \log p_{\text{model}}(x|r) \Big\rangle_{p_{\text{model}}(r|x)} + D_{\text{KL}}[p_{\text{model}}(r|x), p_{\text{latent}}(r)] + \log p_{\text{data}}(x)\;.
\end{aligned} \tag{5.12}$$

As before, the evidence p_{data} is independent of our network training, which means we can use $D_{\text{KL}}[p_{\text{model}}(r|x), p(r|x)]$ modulo the last term as the VAE loss function. Also denoting that everything is always evaluated on batches of events from the training dataset we reproduce (4.5), namely

$$\boxed{\mathcal{L}_{\text{VAE}} = \Big\langle -\big\langle \log p_{\text{model}}(x|r) \big\rangle_{p_{\text{model}}(r|x)} + \beta_{\text{KL}} D_{\text{KL}}[p_{\text{model}}(r|x), p_{\text{latent}}(r)] \Big\rangle_{p_{\text{data}}}}\;. \tag{5.13}$$

We introduce the parameter β_{KL} to allow for a little more flexibility in balancing the two training tasks which are otherwise linked through Bayes' theorem. Just like the Bayesian network, the VAE loss function derived through the variational approximation includes a KL-divergence as a regularization.

As mentioned before, the structure of the latent space r, from which we sample, is introduced by the prior $p_{\text{latent}}(r) = \mathcal{N}(0,1)$. If we control the r-distribution, we can consider the conditional decoder $p_{\text{model}}(x|r)$ a generative network producing events with the probability we desire.

The problem with such a setup of variational AEs for the generation of data with the data distribution $p(x)$ is the underlying assumption that all features in the data can be compressed can be compressed into a low-dimensional and limited latent space. While such an approach may capture qualitative features of a given probability distribution, such an architecture is bound to lack higher-dimensional structures or simply higher order correlations. In the mix of expressivity and achievable accuracy VAEs are usually not competitive with the other generative network architectures we will discuss next.

An exception might be detector simulations, where the underlying physics of calorimeter showers is simple enough to be encoded in a low-dimensional latent space, while the space of detector output channels is huge.

5.2 Generative Adversarial Networks

In our discussion of the VAE we have seen that generative networks are structurally more complex then regression or classification networks. In the language of probability distributions and likelihood losses, we need a generator or decoder network which relies on a conditional probability $p_{\text{model}}(x|r)$ for the target phase space distribution x given the distribution of the incoming random numbers r. For the VAE we used the variational inference trick to construct this latent representation.

An alternative way to learn the underlying density is to combine two networks with adversarial training. Adversarial training means we combine two loss functions

$$\boxed{\mathcal{L}_{\text{adv}} = \mathcal{L}_1 - \lambda \mathcal{L}_2}\;, \tag{5.14}$$

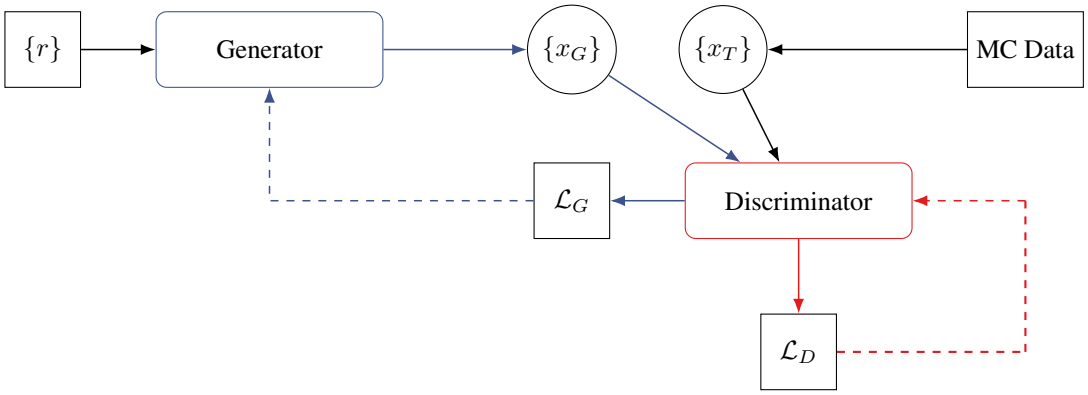


Figure 31: Schematic diagram for a GAN. The input $\{r\}$ describes a batch of random numbers, $\{x\}$ denotes a batch of phase space points sampled either from the generator or the training data.

where the first loss can for example train a classifier and the second term can compute an observable we want to decorrelate. The second network uses the information from the first, classifier network. Because of the negative sign the two sub-networks will now play with each other to find a combined minimum of the loss function. An excellent classifier with small $\mathcal{L}_1$ will use and correctly reproduce the to-be-decorrelation variable, implying also a small $\mathcal{L}_2$. However, for large enough λ the two networks can also work towards a smaller combined loss, where the classifier becomes less ambitious, indicated by a finite $\mathcal{L}_1$, but compensated by an even larger $\mathcal{L}_2$. This balanced gain works best if the classifier is trained as well as possible, but leaving out precisely the aspects which allow for large values of $\mathcal{L}_2$. The two networks playing against each other will then find a compromise where variables entering $\mathcal{L}_2$ are ignored in the classifier training represented by $\mathcal{L}_1$.

Mathematically, the constructive balance or compromise of two players is called a Nash equilibrium. Varying the coupling λ we can strengthen and weaken either of the two sub-networks, a stable Nash equilibrium means that without much tuning of λ the two networks settle into a combined minimum. This does not have to be the case, a combination of two networks can of course be unstable in the sense that depending on the size of λ either $\mathcal{L}_1$ or $\mathcal{L}_2$ wins. Another danger in adversarial training is that the adversary network might force the original network to construct nonsense solutions, which we have not thought about, but which formally minimize the combined loss. In the beginning of Sec. 5 we have observed a mechanism which can lead to such poor solution, where the KL-divergence is insensitive to a massive disagreement between data and network, as long as the distribution we sample from in (5.3) vanishes. In this section we will use adversarial training to construct a generative network.

5.2.1 Architecture

Similar to the VAE structure, the first element of a generative adversarial network (GAN) is the learned generator, just like the VAE decoder

$$p_{\text{model}}(x|r)\Big|_{p_{\text{latent}}(r)=\mathcal{N}(0,1)} . \tag{5.15}$$

Now the latent space r is replaced by a random number generator for r, following some simple Gaussian or flat distribution. The argument x may be a configuration of a statistical theory or quantum field theory, the physical phase space of a jet, a scattering process at the LHC, or a detector output. We remind ourselves that (unweighted) events are nothing but positions in phase space. The difference to the VAE is that we do not train the generator as an inversion or encoder, but use an adversarial loss function like the one shown in (5.14). The GAN architecture is illustrated in Figure 31, and a vanilla GAN is depicted in Figure 32.

We know from Section 3 that it is not hard to train a classification network for statistical theories or jets or events. This means that given a reference dataset $p_{\text{data}}(x)$ and a generated dataset $p_{\text{model}}(x)$ we can train a discriminator or classification network to tell apart the true data and the generated data phase space point by phase space point. This discriminator

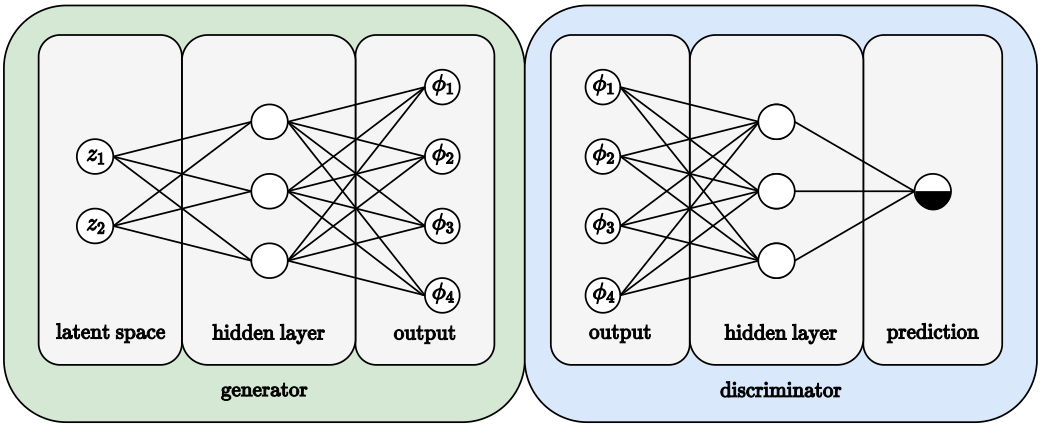


Figure 32: Illustration of the vanilla GAN architecture with fully-connected layers. Neurons are depicted as circles and synapses as lines. The output layer of the discriminator features a single, binary neuron whose state determines the prediction as real or fake. Figure taken from [24].

network is trained to give

$$D(x) = \begin{cases} 0 & \text{generated data} \\ 1 & \text{true data} \end{cases} \tag{5.16}$$

and values in between otherwise. If our discriminator is set up as a proper classification network, its output can be interpreted as the probability of an event being true data. Given a true dataset and a generated dataset, we can train the discriminator to minimize any combination of

$$\big\langle 1 - D(x) \big\rangle_{p_\text{data}} \qquad \text{and} \qquad \big\langle D(x) \big\rangle_{p_\text{model}} . \tag{5.17}$$

For a perfectly trained discriminator both terms will vanish. On the other hand, we know from (3.17) that the loss function for such classification task should be the cross entropy, which motivates the discriminator loss

$$\begin{aligned} \mathcal{L}_D &= \big\langle -\log D(x) \big\rangle_{p_\text{data}} + \big\langle -\log[1 - D(x)] \big\rangle_{p_\text{model}} \\ &= -\int dx \Big[p_\text{data}(x) \log D(x) + p_\text{model}(x) \log(1 - D(x)) \Big] . \end{aligned} \tag{5.18}$$

Comparing this form to the two objectives in (5.17) we simply enhance the sensitivity by replacing $1 - D \to -\log D$. The loss is always positive, and a perfect discriminator will produce zeros for both contributions. From the discriminator loss, we can compute the optimal discriminator output

$$\frac{\delta}{\delta D} \Big[p_\text{data}(x) \log D + p_\text{model}(x) \log(1 - D) \Big] \Big|_{D = D_\text{opt}} = \frac{p_\text{data}(x)}{D_\text{opt}} - \frac{p_\text{model}(x)}{1 - D_\text{opt}} = 0 ,$$

which can be solved for D_opt, to wit,

$$D_\text{opt}(x)\; p_\text{model}(x) = (1 - D_\text{opt}(x))\; p_\text{data}(x) \qquad \Rightarrow \qquad D_\text{opt}(x) = \frac{p_\text{data}(x)}{p_\text{data}(x) + p_\text{model}(x)} , \tag{5.19}$$

assuming that the maximum of the integrand also maximizes the integral because of the positive probability distributions. For (5.19) the loss function has the form of the Jensen-Shannon divergence (5.4).

To train the generator network $p_\text{model}(x|r)$ we now use our adversarial idea. The trained discriminator encodes the agreement of the true and modeled datasets, and all we need to do is evaluate it on the generated dataset

$$\boxed{\mathcal{L}_G = \big\langle -\log D(x) \big\rangle_{p_\text{model}}} . \tag{5.20}$$

This loss will vanish when the discriminator (wrongly) identifies all generated events as true events with $D = 1$.

In our GAN application this discriminator network gets successively re-trained for a fixed true dataset and evolving generated data. In combination, training the discriminator and generator network based on the losses of (5.18) and (5.20) in an alternating fashion forms an adversarial problem which the two networks can solve amicably. The Nash equilibrium between the losses implies, just like in (5.14), that a perfectly trained discriminator cannot tell apart the true and generated samples.

To match the literature, we can merge the two GAN losses (5.18) and (5.20) into one formula after replacing the sampling $x \sim p_{\text{model}}(x)$ with a sampling $r \sim p_{\text{latent}}(r)$ and $x = G(r)$,

$$\begin{aligned} \mathcal{L}_D &= \big\langle -\log D(x) \big\rangle_{p_{\text{data}}} + \big\langle -\log[1 - D(G(r))] \big\rangle_{p_{\text{latent}}} \\ \mathcal{L}_G &= \big\langle -\log D(G(r)) \big\rangle_{p_{\text{latent}}} \;\sim\; \big\langle \log[1 - D(G(r))] \big\rangle_{p_{\text{latent}}} . \end{aligned} \tag{5.21}$$

For the generator loss we use the fact that minimizing $-\log D$ is the same as maximizing $\log D$, which is again the same as minimizing $\log(1 - D)$ in the range $D \in [0, 1]$. The $\sim$ indicates that the two functions will lead to the same result in the minimization, but we will see later that they differ by a finite amount. After modifying the generator loss we can write the two optimizations for the discriminator and generator training as a min-max game

$$\boxed{\min_G \max_D \; \big\langle \log D(x) \big\rangle_{p_{\text{data}}} + \big\langle \log[1 - D(G(r))] \big\rangle_{p_{\text{latent}}}} . \tag{5.22}$$

Finally, we can evaluate the discriminator and generator losses in the limit of the optimally trained discriminator given in (5.19),

$$\begin{aligned} \mathcal{L}_D &\to -\big\langle \log D_{\text{opt}}(x) \big\rangle_{p_{\text{data}}} - \big\langle \log[1 - D_{\text{opt}}(x)] \big\rangle_{p_{\text{model}}} \\ &= -\left\langle \log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{model}}(x)} \right\rangle_{p_{\text{data}}} - \left\langle \log \frac{p_{\text{model}}(x)}{p_{\text{data}}(x) + p_{\text{model}}(x)} \right\rangle_{p_{\text{model}}} \\ &\equiv -2 D_{\text{JS}}[p_{\text{data}}, p_{\text{model}}] + 2 \log 2 , \end{aligned} \tag{5.23}$$

just inserting the definition in (5.4). It shows where the GAN will be superior to the KL-divergence-based VAE, because the JS-divergence is more efficient at detecting a mismatch between generated and training data. For the same optimal discriminator the modified generator loss from (5.21) becomes

$$\begin{aligned} \mathcal{L}_G &\to \big\langle \log(1 - D_{\text{opt}}(x)) \big\rangle_{p_{\text{model}}} \\ &= \left\langle \log \frac{2 p_{\text{model}}(x)}{p_{\text{data}}(x) + p_{\text{model}}(x)} \right\rangle_{p_{\text{model}}} - \log 2 \\ &\equiv D_{\text{KL}} \left[p_{\text{model}}, \frac{p_{\text{data}} + p_{\text{model}}}{2} \right] - \log 2 . \end{aligned} \tag{5.24}$$

For a perfectly trained discriminator and generator the Nash equilibrium is given by

$$p_{\text{data}}(x) = p_{\text{model}}(x) \qquad \Rightarrow \qquad \mathcal{L}_D = 2 \log 2 \qquad \text{and} \qquad \mathcal{L}_G = -\log 2 \tag{5.25}$$

We can find the same result from the original definitions of (5.18) and (5.20), using our correct guess that the perfect discriminator in the Nash equilibrium is constant, namely

$$\begin{aligned} D(x) = \frac{1}{2} \qquad \Rightarrow \qquad \mathcal{L}_D &= -\log \frac{1}{2} - \log \frac{1}{2} = 2 \log 2 \\ \mathcal{L}_G &= -\log \frac{1}{2} = \log 2 . \end{aligned} \tag{5.26}$$

The difference in the value for the generator loss correspond to the respective definitions in (5.21).

The fact that the GAN training searches for a generator minimum given a trained discriminator, which is different from the generator-alone training, leads to the so-called mode collapse. Starting from the generator loss, we see in (5.20) that

it only depends on the discriminator output evaluated for generated data. This means the generator can happily stick to a small number of images or events which look fine to a poorly trained discriminator. In the discriminator loss in (5.18) the second term will, by definition, be happy with this generator output as well. From our discussion of the KL-divergence we know that the first term in the discriminator loss will also be fine if large gradients of $\log D(x)$ only appear in regions where the sampling through the training dataset $p_{\text{data}}(x)$ is poor, which means for example unphysical regions.

After noticing that the JS-divergence of the GAN discriminator loss improves over the KL-divergence-base VAE, we can go one step further and use the Wasserstein distance between the distributions p_{data} and p_{model}. The Wasserstein distance of two non-intersecting distributions grows roughly linearly with their relative distance, leading to a stable gradient. According to the Kantorovich-Rubinstein duality, the Wasserstein distance between the training and generated distributions is given by

$$W(p_{\text{data}}, p_{\text{model}}) = \max_D \left[\big\langle D(x) \big\rangle_{p_{\text{data}}} - \big\langle D(x) \big\rangle_{p_{\text{model}}} \right] . \tag{5.27}$$

For the WGAN the discriminator is also called critic. The definition of the Wasserstein distance involves a maximization in discriminator space, so the discriminator has to be trained multiple times for each generator update. A 1-Lipschitz condition can be enforced through a maximum value of the discriminator weights. It can be be replaced by a gradient penalty, as it is used for regular GANs.

5.3 Applications and limits of GANs

In this Section we briefly discuss applications of GANs to the generation/simulation of data and the limitations for such a use. These limitations also apply to other generative networks and the question of how to overcome them is a highly relevant subject of current research. In Section 5.3.1 we outline the application to the event generation at LHC, and in Section 5.3.2 we describe its use for cutting short out-correlation times in statistical Monte-Carlo simulations.

5.3.1 Event generation

Event generation is at the heart of LHC theory. The standard approach is Monte Carlo simulation, and in this section we will describe how it can be supplemented by a generative network.

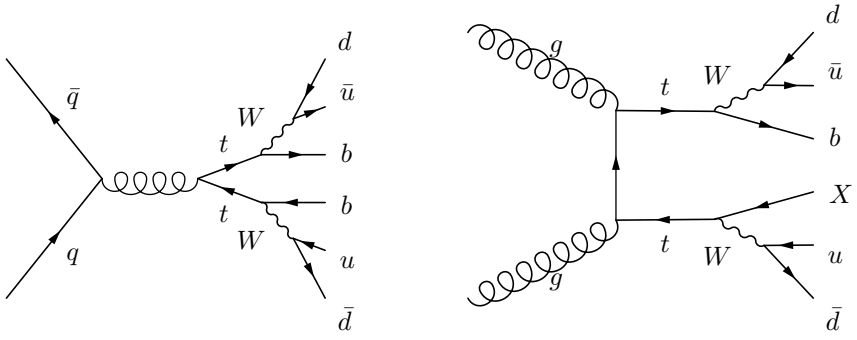
There are several motivations for training a generative network on events:

(i) we can use such a network to efficiently encode and ship standard event samples, rather than re-generating them every time a group needs them.
(ii) we can typically produce several times as many events using a generative network than used for the training
(iii) understanding generative networks for events allows us to test different ML-aspects which can be used for phase space integration and generation
(iv) we can train generative network flexibly at the parton level or at the jet level
(v) finally, we will describe potential applications in the following sections and use generative networks to construct inverse networks

The training dataset for event-generation networks are unweighted events, in other words phase space points whose density represents a probability distribution over phase space. One of the standard reference processes is top pair production including decays,

$$pp \to t^* \bar{t}^* \to (bW^{+*}) \, (\bar{b}W^{-*}) \to (bu\bar{d}) \, (\bar{b}\bar{u}d) \tag{5.28}$$

The star indicates on-shell intermediate particles, described by a Breit-Wigner propagator and shown in the Feynman diagrams



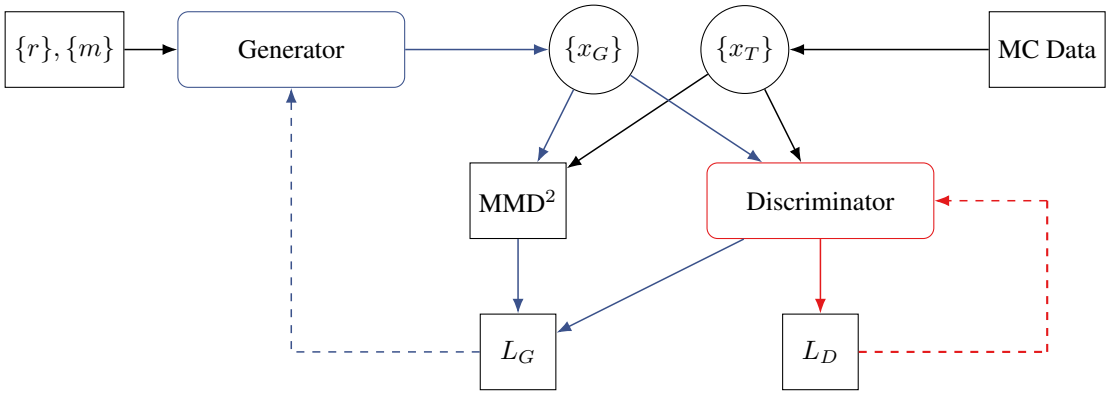


Figure 33: Schematic diagram for an event-generation GAN. It corresponds to the generic GAN architecture in Figure 31, but adds the external masses to the input and the MMD loss defined in (5.32).

For intermediate on-shell particles the denominator of the respective propagator is regularized by extending it into the complex plane, with a finite imaginary part [25]

$$\left| \frac{1}{s - m^2 + im\Gamma} \right|^2 = \frac{1}{(s-m^2)^2 + m^2\Gamma^2} \; . \tag{5.29}$$

By cutting the corresponding self-energy diagrams, Γ can be related to the decay width of the intermediate particle. In the limit $\Gamma \ll m$ we reproduce the factorization into production rate and branching ratio, combined with the on-shell phase space condition

$$\lim_{\Gamma \to 0} \frac{\Gamma_{\text{part}}}{(s-m^2)^2 + m^2\Gamma^2} = \Gamma_{\text{part}} \, \frac{\pi}{\Gamma} \, \delta(s-m^2) = \pi \text{BR}_{\text{part}} \, \delta(s-m^2) \tag{5.30}$$

For a decay coupling g the width scales like $\Gamma \sim mg^2$, so weak-scale electroweak particles have widths in the GeV-range, which means means the Breit-Wigner propagators for top pair production define four sharp features in phase space.

As a first step we ignore additional jet radiation, so the phase space dimensionality of the final state is constant. Each particle in the final state is described by a 4-vector, which means the $t\bar{t}$ phase space has $6 \times 4 = 24$ dimensions. If we are only interested in generating events, we can ignore the detailed kinematics of the initial state, as it will be encoded in the training data. However, we can simplify the phase space because all external particles are on their mass shells and we can compute their energies from their momenta,

$$p^2 = E^2 - m^2 \qquad \Leftrightarrow \qquad E = \sqrt{p^2 + m^2} \; , \tag{5.31}$$

leaving us with an 18-dimensional phase space and six final-state masses as constant input to the network training. Another possible simplification would be to use transverse momentum conservation combined with the fact that the incoming partons have no momentum in the azimuthal plane. We will now use this additional condition in the network training and instead use it to test the accuracy of the network. A symmetry we could use is the global azimuthal angle of the process and replacing the azimuthal angles of all final state particle with an azimuthal angle difference to one reference particle.

The main challenge of training a GAN to learn and produce $t\bar{t}$ events is that the Breit-Wigner propagators strongly constrain four of the 18 phase space dimensions, but those directions are hard to extract from a generic parametrization of the final state. To construct the invariant mass of each of the tops the discriminator and generator have to probe a 9-dimensional part of the phase space, where each direction covers several 100 GeV to reproduce a top mass peak with its width $\Gamma_t = 1.5$ GeV. For a given LHC process and its Feynman diagrams we know which external momenta form a resonance, so we can construct the corresponding invariant mass and give it to the neural network to streamline the comparison between true and generated data. This is much less information than we usually use in Monte Carlo simulations, where we define an efficient phase space mapping from the known masses and widths of every intermediate resonance.

One way to focus the network on a low-dimensional part of the phase space is the maximum mean discrepancy (MMD) combined with a kernel-based method to compare two samples drawn from different distributions. Using one batch of

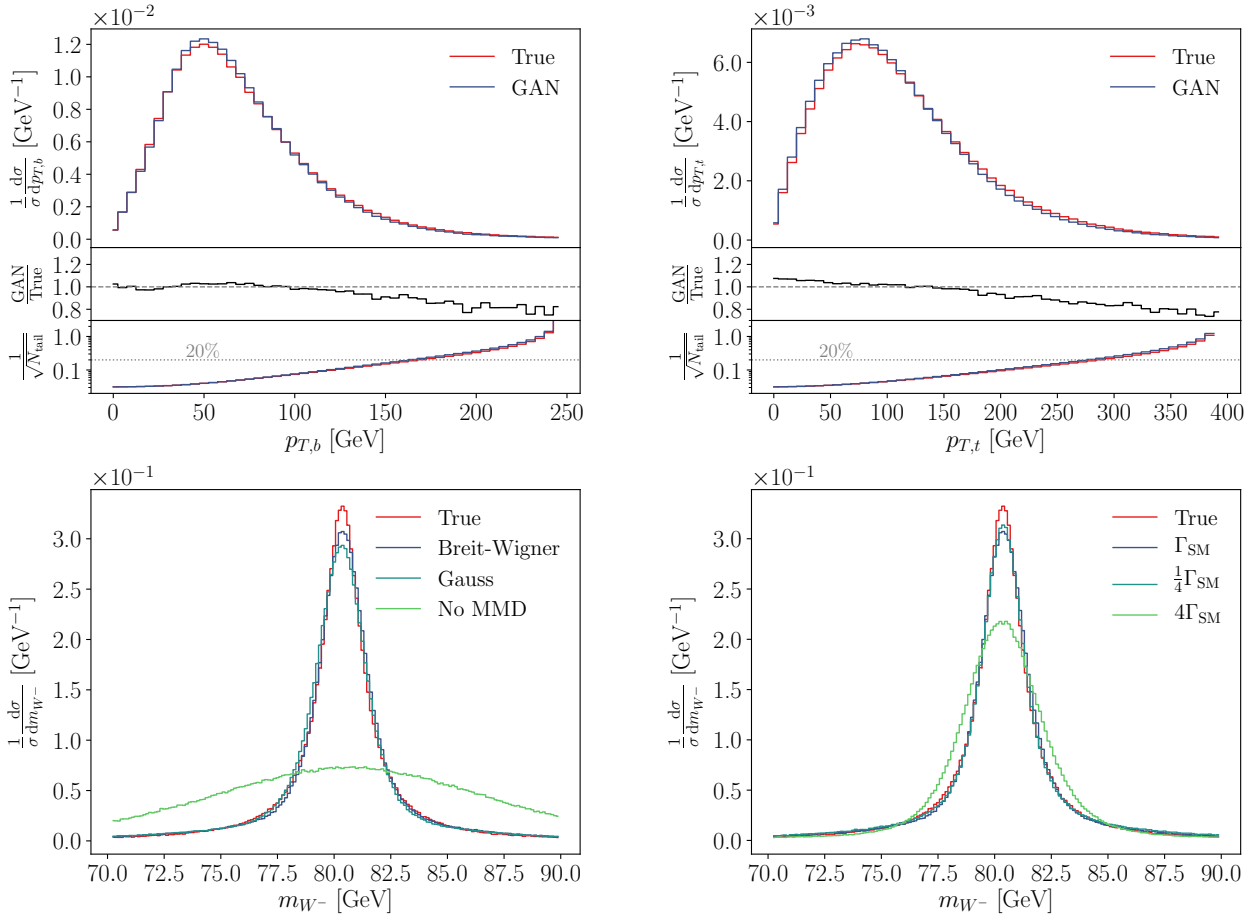


Figure 34: Upper: transverse momentum distributions of the final-state b-quark and the decaying top quark for MC truth and the GAN. The lower panels give the bin-wise ratios and the relative statistic uncertainty on the cumulative number of events in the tail of the distribution for our training batch size. Lower: comparison of different kernel functions and varying widths for reconstructing the invariant W-mass. Figure from Ref.[26].

training data points and one batch of generated data points, it computes a distance between the distributions as

$$\text{MMD}^2(p_\text{data}, p_\text{model}) = \big\langle k(x,x') \big\rangle_{x,x' \sim p_\text{data}} + \big\langle k(y,y') \big\rangle_{y,y' \sim p_\text{model}} - 2 \big\langle k(x,y) \big\rangle_{x \sim p_\text{data}, y \sim p_\text{model}} \; , \tag{5.32}$$

where $k(x,y)$ can be any positive definite, narrow kernel function. Two identical distributions lead to $\text{MMD}(p,p) = 0$ given enough statistics. Inversely, if $\text{MMD}(p_\text{data}, p_\text{model}) = 0$ for randomly sampled batches, the two distributions have to be identical $p_\text{data}(x) = p_\text{model}(x)$. The shape of the kernels determines how local the comparison between the two distributions is evaluated, for instance though a Gaussian kernel with exponentially suppressed tails or a Breit-Wigner with larger tails. The kernel width becomes a resolution hyperparameter of the combined network. We can include the MMD loss to the generator loss of (5.20),

$$\mathcal{L}_G \to \mathcal{L}_G + \lambda_\text{MMD} \, \text{MMD}^2 \; , \tag{5.33}$$

with a properly chosen coupling λ, similar to the parton density loss. The modified GAN setup for event generation is illustrated in Figure 33.

To begin with, we can look at relatively flat distributions like energies, transverse momenta, or angular correlations. In Figure 34 we see that they are learned equally well for final-state and intermediate particles. In the kinematic tails we see that the bin-wise difference of the two distributions increases to around 20%. To understand this effect we estimate the impact of limited training statistics per 1024-event batch through the relative statistical uncertainty on the number of events $N_\text{tail}(p_T)$ in the tail above the quoted p_T value. For the $p_{T,b}$-distribution the GAN starts deviating at the 10% level around

150 GeV. Above this value we expect around 25 events per batch, leading to a relative statistical uncertainty of 20%. The top kinematics is slightly harder to reconstruct, leading to a stronger impact from low statistics.

Next, we can look at sharply peaked kinematic distributions, specifically the invariant masses which we enhance using the MMD loss. In the lower panels of Figure 34 we show the effect of the additional MMD loss on learning the invariant W-mass distribution. Without the MMD in the loss, the GAN barely learns the correct mass value. Adding the MMD loss with default kernel widths of the Standard Model decay widths drastically improves the results. We can also check the sensitivity on the kernel form and width and find hardly any effect from decreasing the kernel width. Increasing the width reduces the resolution and leads to too broad mass peaks.

5.3.2 GAN down autocorrelation times in lattice simulations

The Achilles heel of GANs (and other generative networks) is, that it cannot learn what is not in the data sample it learns from. For example, if we use a GAN to generate 10^3 more data than present in the data sample, it may help us to resolve higher order correlation functions or tails of the distribution, but these results have to be evaluated with great caution.

Instead of using a GAN for the full generative task above, we want to use the GAN for an overrelaxation step within a Standard Monte-Carlo simulation of the lattice action Equation (3.10) as described in [24].

Suppose we wish to sample field configurations from the Boltzmann weight $P(\phi) \propto \exp(-S(\phi))$. A candidate ϕ' is generated from the current configuration ϕ with some a priori selection probability $T_0(\phi'|\phi)$, which we assume to be symmetric. It is then accepted or rejected according to the acceptance probability

$$T_A(\phi'|\phi) = \min\Big(1,\ \exp(-\Delta S)\Big)\,, \qquad \text{with} \qquad \Delta S = S(\phi) - S(\phi')\,. \tag{5.34}$$

For a real scalar field, this is ensured by proposing updates ϕ_i' distributed symmetrically around ϕ, such that $\phi_i' - \phi_i$ is zero on average. Such an update procedure leads to autocorrelations between the sequential configurations $\phi(\tau)$ and $\phi(\tau + t)$, where the discrete variable τ labels the update steps in the algorithm (Markov chain) at work and t is a (discrete) distance to a future one. For general observables X the autocorrelation is captured by the autocorrelation function C_X with

$$C_X(t) = \langle (X_\tau - \langle X_\tau \rangle)\,(X_{\tau+t} - \langle X_{\tau+t} \rangle) \rangle = \langle X_\tau X_{\tau+t} \rangle - \langle X_\tau \rangle \langle X_{\tau+t} \rangle\,. \tag{5.35}$$

Local updating methods based on the Metropolis-Hastings algorithm usually exhibit long autocorrelation times. Significant improvements can be achieved with the HMC method, which we have used to generate the samples for the statistical theory with the action (3.10) in Section 3.2.2. It is based on a molecular dynamics evolution using classical Hamiltonian equations of motion, combined with an accept/reject step. This allows larger steps with reasonable acceptance rates.

We now may try to use a GAN to substitute the direct sampling from the distribution $P(\phi)$. The respective results are depicted in Figure 36: while the GAN-result for the magnetization agrees very well with the data, the distribution does not. Certainly, we can train our GAN better than that, but still, Figure 36 elucidates very clearly the problem of generative networks discussed above.

Now we abandon the task of directly simulating with a GAN and use it instead to speed up the convergence of the direct simulation: A technique used to speed up the motion through configuration space is overrelaxation. It uses the fact that a candidate configuration is automatically accepted in the Metropolis step if $\Delta S = 0$ under the condition that T_0 is symmetric. This can be achieved by performing rotations in group space that leave S unchanged. By itself, overrelaxation is therefore not ergodic, since it moves on the subspace of constant action. Ergodicity is achieved by combining it with a standard Monte Carlo algorithm. The respective workflow is depicted in Figure 35.

However, neither the HMC algorithm nor the overrelaxation method solve critical slowing down. Simply put, the problem occurs whenever the correlation length ξ of a system diverges. Typically, the autocorrelation function $C_X(t)$ decays exponentially,

$$C_X(t) \sim \exp\left(-\frac{t}{\tau_{\text{exp}}}\right)\,. \tag{5.36}$$

Here, τ_{exp} denotes the exponential autocorrelation time. A vanishing correlation length signals a second order phase transition. In the vicinity of the phase transition one expects the autocorrelation time to scale as a power of the correlation length, $\tau_{\text{exp}} \sim \xi^z$. The dynamical critical exponent $z \geq 0$ depends on the type of algorithm used. Close to a critical

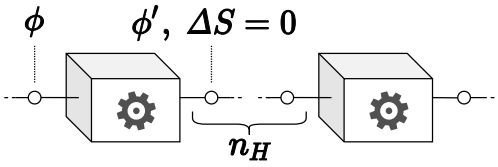


Figure 35: Sketch of the updating procedure on the Markov chain, with boxes depicting the overrelaxation step. Figure taken from [24].

point, the correlation length diverges. This hampers the investigation of critical phenomena as well as extrapolations to the continuum limit, as the latter is given by a second order phase transition.

Now we let loose our vanilla GAN, depicted in Figure 32, on the overrelaxation step. The combined loss function in Figures 31 and 32, Equation (5.22), adapted to the present task, is given by the binary cross entropy $\mathcal{L}_{\text{BCE}}$: the loss per prediction with label $y = G(z)$ is defined as

$$\mathcal{L}_{\text{BCE}} = \langle -G(z)\log(D(G(z))) + (G(z)-1)\log(1-D(G(z)))\rangle \,. \tag{5.37}$$

Equation (5.37) is the loss function taken in [24]. It is a variant of the cross entropy and the multiplication of the logarithms with the label is a common procedure, while not necessary.

Training corresponds to minimizing the loss separately for D and G using opposite labels, respectively. This is achieved by evolving their weights according to a gradient flow equation in an alternating fashion. The gradient of the loss with respect to the weights is calculated using automatic differentiation, and then backpropagated through both networks by the chain rule. In intuitive terms, the optimization objective for the discriminator is to maximize its accuracy for the correct classification of ϕ and $G(z)$ as real or fake. Simultaneously, the generator is trained to produce samples that cause false positive predictions of the discriminator, thereby approximating the true target distribution $P(\phi)$. The two networks play a zero-sum non-cooperative game, and the model is said to converge when they reach so-called Nash equilibrium.

This entails the fundamental difference and possible advantage of this method compared to traditional Monte Carlo sampling. Since $P(z)$ is commonly chosen to be a simple multi-variate uniform or Gaussian distribution, candidate configurations drawn from a properly equilibrated generator are by construction statistically independent. This is because the z are sampled i.i.d. and the resulting field configurations are not calculated as consecutive elements of a traditional Markov chain.

In practice however, one encounters deviations of varying severity from the true target distribution. Also, GANs may not be sufficiently ergodic in order to perform reliable calculations. This becomes apparent when one considers the extreme case of a mode collapse, where the generator learns to produce only one or a very small number of samples largely independent of its prior distribution. Insufficient variation among the GAN output is not punished by the discriminator and can only be checked a posteriori. A number of improved approaches to deal with such issues have since been proposed in the literature [27]. Still, one may question whether GANs can be sufficiently random for the level of rigor required in certain statistical sampling problems. Interestingly, it was shown that they can act as reliable pseudo-random number generators, outperforming several standard, non-cryptographic algorithms [28]. Now we implement GAN as an overrelaxation step, which can then be integrated into any action-based importance sampling algorithm. In this manner, autocorrelations can be reduced substantially while still asymptotically approaching the correct distribution of the theory. Our method of selecting suitable candidate configurations is based on [29], where GANs are proposed as an Ansatz to the more general task of solving inverse problems.

Our approach is implemented with the following procedure (see Figure 35 for a sketch):

- Take a number of HMC steps n_H to obtain a configuration ϕ. In this work, only accepted samples are counted by n_H to facilitate a consistent number of HMC trajectories between GAN overrelaxation steps in the Markov chain.
- Pre-sampling: Sample from the GAN until a configuration $G(z)$ is found that fulfills $|\Delta S| = |S[G(z)] - S[\phi]| \leq \Delta S_{\text{thresh}}$. ΔS_{thresh} is tuned such that $S[\phi]$ and $S[G(z)]$ are close, which accelerates the gradient flow step, but also such that a suitable $G(z)$ can be generated in a reasonable amount of time.
- Gradient flow: Perform a gradient descent evolution of the associated latent variable z using ΔS^2 as loss function, i.e.

$$z' = \text{argmin}_{\,z}(S[G(z)] - S[\phi])^2 \,, \tag{5.38}$$

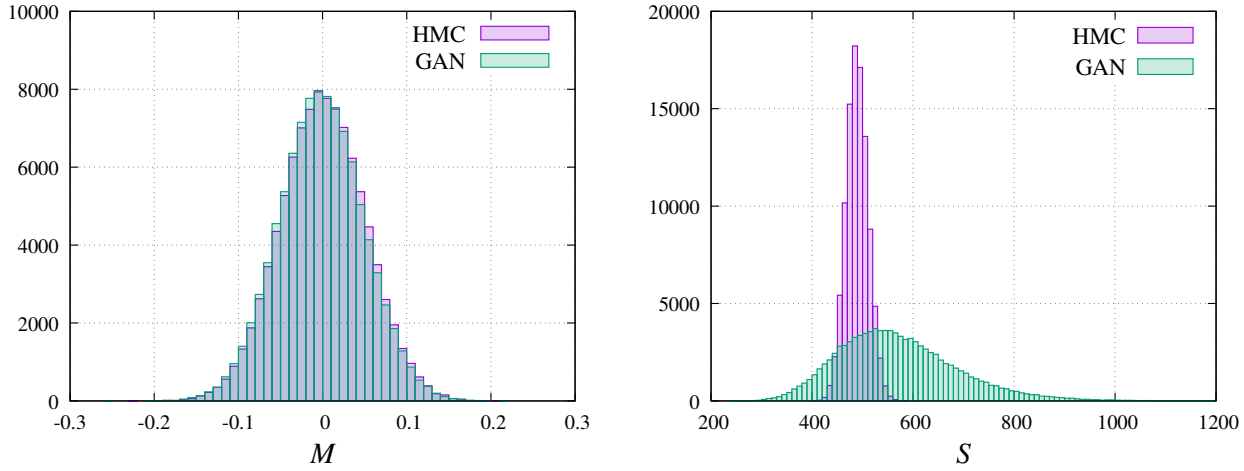


Figure 36: Comparison of magnetization (left) and action (right) distributions with 10^5 samples generated with the HMC algorithm and the GAN trained on 10^3 configurations.

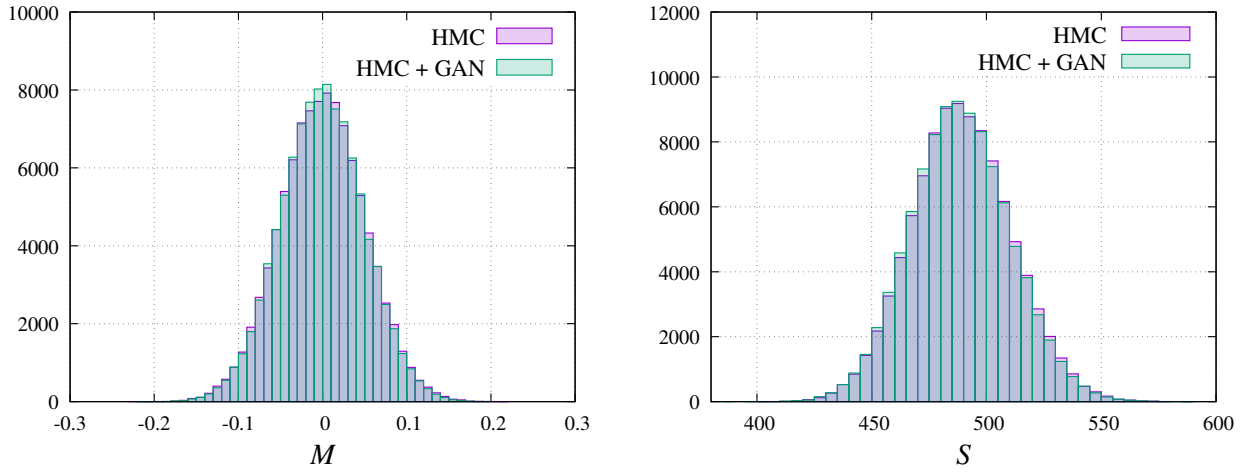


Figure 37: Comparison of magnetization (left) and action (right) distributions with 10^5 samples generated with the baseline HMC and in combination with the GAN overrelaxation step using $n_H = 3$.

by employing a standard discretized gradient flow equation,

$$z'(\tau + \epsilon) = z'(\tau) - \epsilon \frac{\partial \Delta S^2}{\partial z'} \,. \tag{5.39}$$

As mentioned in the last section, gradient flow is also commonly applied to learn the optimal weights of a network. In this context, the finite step size ϵ is called learning rate, and gradients are usually obtained by automatic differentiation. For this work, we use the Adam algorithm [30], a particular variant of stochastic gradient descent.

In this manner, $S[\phi]$ and $S[G(z')]$ can be matched arbitrarily well, down to the available floating point precision. The action values can then be considered effectively equal for all intents and purposes. In principle, this evolution can be performed for any randomly drawn z without the need for repeated sampling until a configuration with $|\Delta S| < \Delta S_{\text{thresh}}$ is found. The additional pre-sampling step simply ensures that the distance in the latent space between the initial value for z and the target z' is already small a priori, which speeds up the evolution and avoids the risk of getting stuck in a local minimum of the loss landscape. The specific choice of ΔS_{thresh}, ϵ as well as the sampling batch size should be determined through a hyperparameter optimization in order to maximize the efficiency.

The distributions of the magentization M and the action S ($\log P$) obtained with our modified sampling algorithm, as well as the aforementioned observables, are consistent with results from the pure HMC simulation, see Figure 37. In particular,

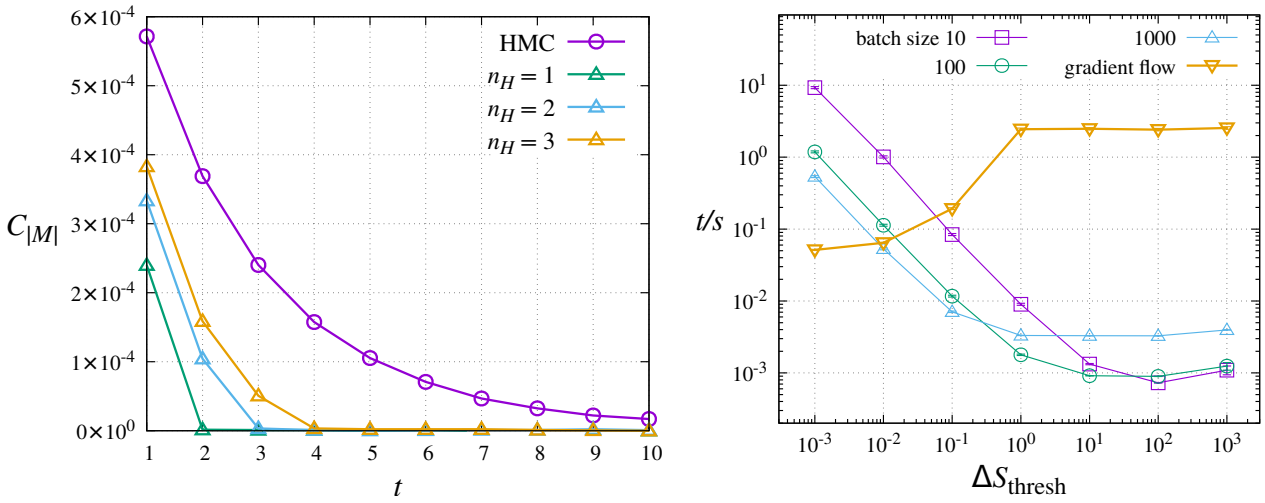


Figure 38: LEFT: Comparison of the autocorrelation functions of $|M|$ for a local Metropolis update, the HMC algorithm and our method using $n_H \in \{1, 2, 3\}$. Results for M also show the same qualitative behavior. RIGHT: Average time given in seconds as a function of ΔS_{thresh}, both for the pre-sampling step using three different batch sizes, as well as the gradient flow step with a learning rate of $\epsilon = 10^{-6}$.

the difference between the action distributions that was seen in Figure 36 has disappeared completely, which indicates that the algorithm correctly reproduces the dynamics of the theory. We can estimate the efficiency gain by comparing the behavior of the associated autocorrelation functions $C_{|M|}(t)$, shown in Figure 38 for $n_H = 1, 2, 3$. We observe a substantial reduction of autocorrelations by our method compared to the HMC baseline, to the extent that $C_{|M|}(t)$ is almost zero at $t = n_H + 1$, i.e. after every GAN overrelaxation step. The small residual autocorrelation observed for $t \geq n_H + 1$ stems solely from the acceptance rate of the intermediate HMC steps and is exactly zero when only accepted samples are taken into account for the computation of $C_{|M|}(t)$.

We now determine for each Markov chain the corresponding integrated autocorrelation time, defined as

$$\tau_{X,\text{int}} = \frac{1}{2} + \frac{1}{C_X(0)} \sum_{t=1}^{T} C_X(t) \,. \tag{5.40}$$

It is generally expected to scale as $\tau_{X,\text{int}} \sim (\xi_X)^z$, where ξ_X is now the correlation length for the observable X and z again denotes the dynamical critical exponent, which depends on the algorithm. We use 10^6 consecutive measurements of $|M|$ to calculate each $C_{|M|}(t)$ and truncate the sum for $\tau_{|M|,\text{int}}$ at $T = 100$. For the HMC baseline, we obtain $\tau_{|M|,\text{int}} \approx 2.29$. Using our modified algorithm, the results for $n_H = 1, 2, 3$ are determined as $\tau_{|M|,\text{int}} \approx 0.75,\ 0.96,\ 1.16$, respectively. This clearly demonstrates that our proposed method could significantly improve the dynamical critical exponents of established sampling algorithms.

In order to facilitate a rough comparison of the computational cost, both the HMC update and the GAN overrelaxation step were implemented using the same framework, the deep learning library PyTorch [31]. All steps of the simulation except for the recording and monitoring functionality are performed on an Nvidia GeForce GTX 1070. The average time for the computation of one (accepted) HMC trajectory was measured to be 42 ms. For the GAN overrelaxation, the average time depends on the aforementioned hyperparameters. In the sampling step, we need to consider the behavior with respect to the batch size and ΔS_{thresh}. Larger batches require more time, but are more likely to contain suitable samples for small values of ΔS_{thresh} and are preferred in this region, since repeatedly sampling smaller batches is less efficient. On the contrary, if ΔS_{thresh} is fixed at a larger value, it is increasingly likely for any given sample to satisfy the criterion, and small batch sizes are sufficient. Figure 38 shows the average time for three different batch sizes as a function of ΔS_{thresh}. Plateaus can be observed at larger and consistent scaling properties at smaller values.

In contrast to the time required for sampling, the gradient flow step is completed faster for smaller ΔS_{thresh} and takes longer at large values, as shown in Figure 38. Hence, one needs to find a trade-off value where neither of the two steps takes a prohibitively long time. The gradient flow also generally depends on the discrete step size or learning rate ϵ, but different choices of this parameter were in our case found to have only a weak effect on the overall time. The optimal order of

magnitude for the hyperparameters was determined to be $\Delta S_{\text{thresh}} = 10^{-2}$ with a batch size of 10^3 and a learning rate of $\epsilon = 10^{-6}$. With these values, the sampling and gradient flow step require on average 53 ms and 64 ms, respectively, yielding a combined time of 117 ms.

5.4 Normalizing flows and invertible networks

After discussing the generative VAE and GAN architectures we remind ourselves that controlling networks and uncertainty estimation are really important for regression and classification networks. An important generic source of uncertainty, $\sigma_{\text{stat}}(x)$, comes with the statistical limitations of the training data. We have already emphasized this fact at the beginning of Section 5.3.1. However, there are at least two further ones that are specific to the task at hand. The first one is the systematic uncertainty, $\sigma_{\text{sys}}(x)$, inherent to our setup. A further one is a theoretical one, $\sigma_{\text{th}}(x)$, that relates to theory shortcomings.

As these errors are specific to the problem, we exemplify them at one of our working horses: in LHC applications we want to know the uncertainties on phase space distributions, e.g. when we rely on simulations for the background p_T-distribution in mono-jet searches for dark matter. If we generate large numbers of events, uncertainties on generated LHC distributions are uncertainties on the accuracy with which our generative network has learned the underlying phase space distribution it then samples from.

In general, there exist at least three sources of uncertainty. First, $\sigma_{\text{stat}}(x)$ arises from statistical limitations of the training data. Two additional terms, $\sigma_{\text{sys}}(x)$ and $\sigma_{\text{th}}(x)$ reflect our ignorance of aspects of the training data, which do not decrease when we increase the amount of training data. If we train on data, a systematic uncertainty could come from a poor calibration of particle energy in certain phase space regions. If we train on Monte Carlo, a theory uncertainty will arise from the treatment of large Sudakov logarithms of the kind $\log(E/m)$ for boosted phase space configurations. Once we know these uncertainties as a function of phase space, we can include them in the network output as additional event entries, for instance supplementing

$$\text{ev} = \begin{pmatrix} \{x_{\mu,j}\} \\ \{p_{\mu,j}\} \end{pmatrix} \quad \longrightarrow \quad \begin{pmatrix} \sigma_{\text{stat}}/p \\ \sigma_{\text{syst}}/p \\ \sigma_{\text{th}}/p \\ \{x_{\mu,j}\} \\ \{p_{\mu,j}\} \end{pmatrix} , \qquad \text{for each particle } j. \tag{5.41}$$

The first challenge is to extract σ_{stat} without binning, which leads us to introduce normalizing flows directly in the Bayesian setup.

5.4.1 Architecture

To model complex densities precisely and sample from them in a controlled manner, we would like to modify the VAE architecture such that the latent space can encode all phase space correlations. We can choose the dimensionality of the latent vector r the same as the dimensionality of the phase space vector x. This gives us the opportunity to define the encoder and decoder as bijective mappings, which means that the encoder and the decoder are really the same network evaluated in opposite directions. This architecture is called a normalizing flow or an invertible neural network (INN),

$$\text{latent } r \sim p_{\text{latent}} \quad \underset{\leftarrow \overline{G}_\theta(x)}{\overset{G_\theta(r) \rightarrow}{\longleftrightarrow}} \quad \text{phase space } x \sim p_{\text{data}} \, , \tag{5.42}$$

where $\overline{G}_\theta(x)$ denotes the inverse transformation to $G_\theta(r)$. Given a sample r from the latent distribution, we can use G to generate a sample from the target distribution. Alternatively, we can use a sample x from the target distribution to compute its density using the inverse direction. A comparison to a standard Bayesian network is depicted in Figure 39.

In terms of the network $G_\theta(r)$ the physical phase space density and the latent density are related as

$$\begin{aligned} & dx \; p_{\text{model}}(x) = dr \; p_{\text{latent}}(r) \\ \Leftrightarrow \qquad & p_{\text{latent}}(r) = p_{\text{model}}(x) \left| \frac{\partial G_\theta(r)}{\partial r} \right| = p_{\text{model}}\big(G_\theta(r)\big) \left| \frac{\partial G_\theta(r)}{\partial r} \right| \\ \Leftrightarrow \qquad & p_{\text{model}}(x) = p_{\text{latent}}(r) \left| \frac{\partial G_\theta(r)}{\partial r} \right|^{-1} = p_{\text{latent}}\big(\overline{G}_\theta(x)\big) \left| \frac{\partial \overline{G}_\theta(x)}{\partial x} \right| \end{aligned} \tag{5.43}$$

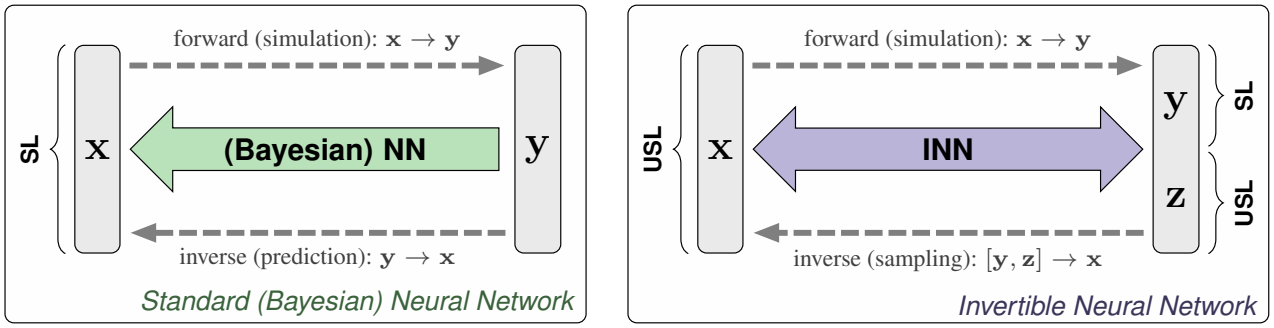


Figure 39: **Abstract comparison of standard approach *(left)* and an INN *(right)*.** The standard direct approach requires a discriminative, supervised loss (**SL**) term between predicted and true $\mathbf{x}$, causing problems when $\mathbf{y} \to \mathbf{x}$ is ambiguous. Our network uses a supervised loss only for the well-defined forward process $\mathbf{x} \to \mathbf{y}$. Generated $\mathbf{x}$ are required to follow the prior $p(\mathbf{x})$ by an unsupervised loss (**USL**), while the latent variables $\mathbf{z}$ are made to follow a Gaussian distribution, also by an unsupervised loss. Figure taken from [32].

For an INN we require the latent distribution p_{latent} to be known and simple enough to allow for efficient sample generation, G_θ to be flexible enough for a non-trivial transformation, and its Jacobian determinant to be efficiently computable. We start by choosing a multivariate Gaussian with mean zero and an identity matrix as the covariance at the distribution p_{latent} in the unbounded latent space. The INN we will use is a special variant of a normalizing flow network, inspired by the RealNVP architecture, which guarantees a

- bijective mapping between latent space and physics space;
- equally fast evaluation in both direction;
- tractable Jacobian, also in both directions.

The construction of G_θ relies on the usual assumption that a chain of simple invertible nonlinear maps gives us a complex map. This means we transform the latent space into phase space with several transformation layers, for which we need to know the Jacobians. For instance, we can use affine coupling layers as building blocks. Here, the input vector r is split in half, $r = (r_1, r_2)$, allowing us to compute the output $x = (x_1, x_2)$ of the layer as

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} r_1 \odot e^{s_2(r_2)} + t_2(r_2) \\ r_2 \odot e^{s_1(x_1)} + t_1(x_1) \end{pmatrix} \quad \Leftrightarrow \quad \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} (x_1 - t_2(r_2)) \odot e^{-s_2(r_2)} \\ (x_2 - t_1(x_1)) \odot e^{-s_1(x_1)} \end{pmatrix}, \tag{5.44}$$

where s_i, t_i $(i = 1, 2)$ are arbitrary functions, and $\odot$ is the element-wise product. In practice each of them will be a small multi-layer network. The Jacobian of the transformation G is an upper triangular matrix, and its determinant is just the product of the diagonal entries.

$$\begin{aligned} \frac{\partial G_\theta(r)}{\partial r} &= \begin{pmatrix} \partial x_1/\partial r_1 & \partial x_1/\partial r_2 \\ \partial x_2/\partial r_1 & \partial x_2/\partial r_2 \end{pmatrix} = \begin{pmatrix} \text{diag}\left(e^{s_2(r_2)}\right) & \text{finite} \\ 0 & \text{diag}\left(e^{s_1(x_1)}\right) \end{pmatrix} \\ \Rightarrow \qquad \left|\frac{\partial G_\theta(r)}{\partial r}\right| &= \prod e^{s_2(r_2)} \prod e^{s_1(x_1)} \,. \end{aligned} \tag{5.45}$$

Such a Jacobian determinant is computationally inexpensive and still allows for complex transformations. We refer to the sequence of coupling layers as $G_\theta(r)$, collecting the parameters of the individual nets s, t into a joint θ.

Given the invertible architecture we proceed to train our network via a maximum likelihood loss, which we already used as the first term of the VAE loss in (5.13). It relies on the assumption that we have access to a dataset which encodes the intractable phase space distribution $p_{\text{data}}(x)$ and want to fit our model distribution $p_{\text{model}}(x)$ via G_θ. The maximum likelihood loss for the INN is

$$\boxed{\mathcal{L}_{\text{INN}} = -\Big\langle \log p_{\text{model}}(x) \Big\rangle_{p_{\text{data}}} = -\left\langle \log p_{\text{latent}}\left(\overline{G}_\theta(x)\right) + \log \left|\frac{\partial \overline{G}_\theta(x)}{\partial x}\right| \right\rangle_{p_{\text{data}}}}. \tag{5.46}$$

The first of the two terms ensures that the latent representation remains, for instance, Gaussian, while the second term constructs the correct transformation to the phase space distribution. Given the structure of $\overline{G}_\theta(x)$ and the latent distribution

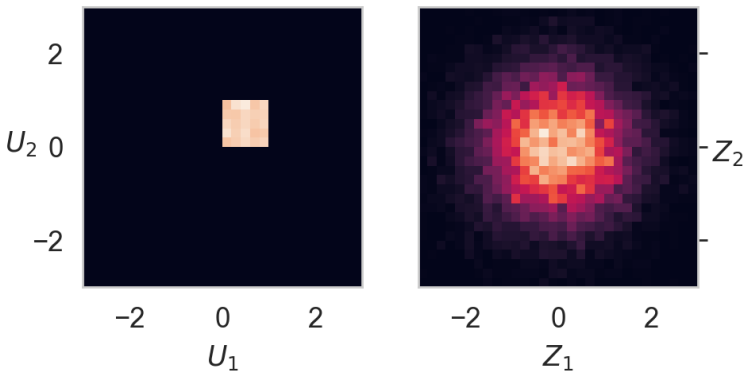


Figure 40: Box-Muller transform (5.50) from two standard uniform distribution $unif(0,1)$ to two uncorrelated unit-variance Gaußian distributions. Figure taken from [33].

p_{latent}, both terms can be computed efficiently. As in (1.18), one can view this maximum likelihood approach as minimizing the KL-divergence between the true but unknown phase space distribution $p_{\text{data}}(x)$ and our approximating distribution $p_{\text{model}}(x)$.

While the INN provides us with a powerful generative model of the underlying data distribution, it does not account for an uncertainty in the network parameters θ. However, because of its bijective nature with the known Jacobian, the INN allows us to meaningfully switch from deterministic sub-networks $s_{1,2}$ and $t_{1,2}$ to their Bayesian counterparts. We recall that we can write the BNN loss function of (1.65) as

$$\mathcal{L}_{\text{BNN}} = -\Big\langle \log p_{\text{model}}(x) \Big\rangle_{\theta\sim q} + D_{\text{KL}}[q(\theta), p(\theta)] \; . \tag{5.47}$$

We now approximate the intractable posterior $p(\theta|x_{\text{train}})$ with a mean-field Gaussian as the variational posterior $q(\theta)$ and then apply Bayes' theorem to train the network on the usual ELBO loss, now for event samples

$$\begin{aligned} \mathcal{L}_{\text{B-INN}} &= -\Big\langle \log p_{\text{model}}(x) \Big\rangle_{\theta\sim q, x\sim p_{\text{data}}} + D_{\text{KL}}[q(\theta), p(\theta)] \\ &= -\left\langle \log p_{\text{latent}}\big(\overline{G}_\theta(x)\big) + \log \left| \frac{\partial \overline{G}_\theta(x)}{\partial x} \right| \right\rangle_{\theta\sim q, x\sim p_{\text{data}}} + D_{\text{KL}}[q(\theta), p(\theta)] \end{aligned} \tag{5.48}$$

By design, the likelihood, the Jacobian, and the KL-divergence can be computed easily.

To generate events using this model and with statistical uncertainties, we remind ourselves how Bayesian network sample over weight space in (2.4), but how to predict a phase space density with a local uncertainty map. In terms of the BNN network outputs analogous to (2.7) this means

$$\begin{aligned} p(x) &= \int d\theta \; q(\theta) \; p_{\text{model}}(x) \\ \sigma^2_{\text{pred}}(x) &= \int d\theta \; q(\theta) \; [p_{\text{model}}(x|\theta) - p(x)]^2 \; . \end{aligned} \tag{5.49}$$

Here x denotes the initial phase space vector from (5.41), and the predictive uncertainty can be identified with the corresponding relative statistical uncertainty σ_{stat}.

5.4.2 Illustration: Box-Muller transform and more

This example is taken from [33] which offers a nice concise hands-on introduction to normalizing flows for lattice field theories.

The Box-Muller transform maps two sets of random numbers U_1, U_2 drawn from the standard uniform distribution $unif(0,1)$ into two uncorrelated unit-variance Gaußian distributions with variables Z_1 and Z_2,

$$Z_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2)\,, \qquad Z_2 = \sqrt{-2\ln U_1}\,(2\pi U_2)\,. \tag{5.50}$$

The sum of the square of the two transformations in (5.50) leads us to

$$Z_1^2 + Z_2^2 = -2\ln U_1\,, \qquad \text{and} \qquad e^{-\frac{Z_1^2+Z_2^2}{2}} = U_1\,. \tag{5.51}$$

In terms of the normalizing flow above the uniform distributions provide the prior distribution

$$r(U_1,U_2) = 1\,, \qquad U_1, U_2 \in [0,1]\,, \tag{5.52}$$

and the model distribution is a normal distribution with unit variance,

$$q(Z_1,Z_2) = \frac{1}{2\pi}\, e^{-\frac{Z_1^2+Z_2^2}{2}}\,. \tag{5.53}$$

The relation between the model distribution and the prior distribution can be determined by

$$dZ_1\, dZ_2\, q(Z_1,Z_2) = dU_1\, dU_2\, r(U_1,U_2)\,, \tag{5.54}$$

leading to

$$\begin{aligned} q(Z_1,Z_2) &= r(U_1,U_2)\left|\det_{kl}\frac{\partial Z_k(U_1,U_2)}{\partial U_l}\right|^{-1} \\ &= 1\times\left|\det\begin{pmatrix} \frac{-1}{U_1\sqrt{-2\ln U_1}}\cos(2\pi U_2) & -2\pi\sqrt{-2\ln U_1}\sin(2\pi U_2) \\ \frac{-1}{U_1\sqrt{-2\ln U_1}}\sin(2\pi U_2) & 2\pi\sqrt{-2\ln U_1}\cos(2\pi U_2)\end{pmatrix}\right|^{-1} = \frac{U_1}{2\pi}\,. \end{aligned} \tag{5.55}$$

It follows from (5.51), that the last line in (5.55) is nothing but the normalized distribution $q(Z_1,Z_2)$.

We now generalize this example slightly to a simple variant of (5.44). To that end we consider coupling layers with an (differentiable and invertible) function $G(r)$ with $r = (r_1, r_2)$ and $x = G(r)$ with $x = (x_1, x_2)$. Its inverse is given by $G^{-1}(x) = \bar{G}(x) = r$. This leads us to

$$p_{\text{model}}(x) = p_{\text{latent}}\left|\det_{ij}\frac{\partial G_i}{\partial r_j}\right|^{-1}\,. \tag{5.56}$$

Now we consider coupling layers with the affine transformation g

$$x_1' = e^{s(x_2)}x_1\,, \quad x_2' = x_2\,. \qquad \text{Inverse:} \qquad x_1 = e^{-s(x_2')}x_1'\,, \quad x_2 = x_2'\,. \tag{5.57}$$

The transformation (5.57) has a simple Jacobian with a triangular form,

$$\frac{\partial g(x_1,x_2)}{\partial x} = \begin{pmatrix}\frac{\partial x_1'}{\partial x_1} & \frac{\partial x_1'}{\partial x_2} \\ 0 & 1\end{pmatrix}\,, \tag{5.58}$$

and the Jaocbi determinant takes the simple form

$$\left|\det_{ij}\frac{\partial[G(x_1,x_2)]_k i}{\partial x_j}\right| = \prod_i e^{[s(x_2)]_i}\,. \tag{5.59}$$

Equation (5.59) can be easily computed, and its simple form, while still keeping expressivity, is at the root of the advantages of the present architecture.

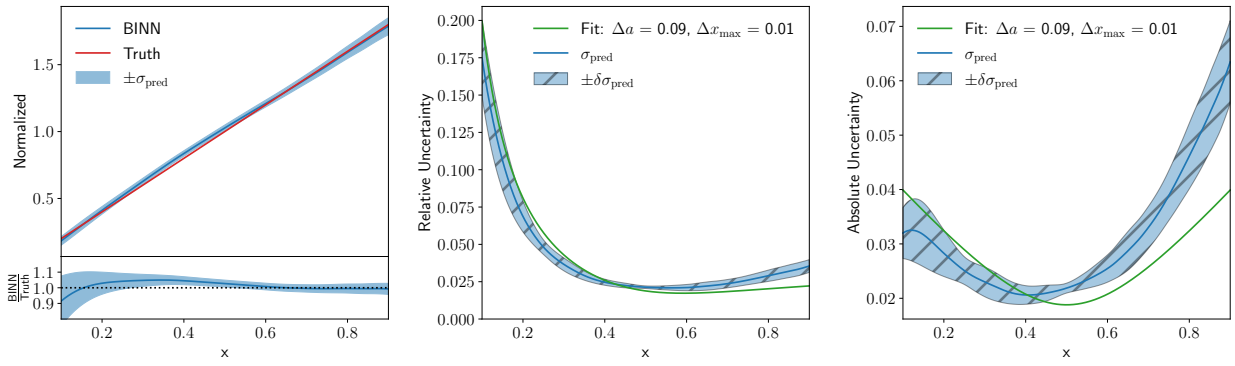


Figure 41: Density and predictive uncertainty distribution for a linear wedge ramp using a B-INN. The uncertainty on σ_{pred} is given by its y-variation. The green curve represents a 2-parameter fit to (5.65). Figure from Ref. [34].

5.4.3 Two-dimensional examples

Let us illustrate Bayesian normalizing flows using a set of 2-dimensional toy models. First, we look a simple 2-dimensional ramp distribution, linear in one direction and flat in the other,

$$p(x,y) = 2x \; . \tag{5.60}$$

The factor two ensures that $p(x,y)$ is normalized. The network input and output consist of unweighted events in the 2-dimensional parameters space, (x,y).

In Figure 41 we show the network prediction, including the predictive uncertainty on the density. Both, the phase space density and the uncertainty are scalar fields defined over phase space, where the phase space density has to be extracted from the distribution of events and the uncertainty is explicitly given for each event or phase space point. For both fields we can trivially average the flat y-distribution. In the left panel we indicate the predictive uncertainty as an error bar around the density estimate, covering the deviation from the true distribution well.

In the central and right panels of Figure 41 we show the relative and absolute predictive uncertainties. The relative uncertainty decreases towards larger x. However, the absolute uncertainty shows a distinctive minimum around $x \approx 0.45$. To understand this minimum we focus on the non-trivial x-coordinate with the linear form

$$p(x) = ax + b \qquad \text{with} \qquad x \in [0,1] \; . \tag{5.61}$$

Because the network learns a density, we can remove b by fixing the normalization,

$$1 = \int_0^1 dx(ax+b) = \frac{a}{2} + b \qquad \Rightarrow \qquad p(x) = a\left(x - \frac{1}{2}\right) + 1 \; . \tag{5.62}$$

Let us now assume that the network acts like a one-parameter fit of a to the density, so we can propagate the uncertainty on the density into an uncertainty on a,

$$\sigma_{\text{pred}} \equiv \Delta p \approx \left| x - \frac{1}{2} \right| \; \Delta a \; . \tag{5.63}$$

The absolute value appears because the uncertainties are defined to be positive, as encoded in the usual quadratic error propagation. The minimum at $x = 1/2$ explains the pattern we see in Figure 41. What this simple approximation cannot explain is that the predictive uncertainty is not symmetric and does not reach zero. However. we can modify our simple ansatz to vary the hard-to-model boundaries and find

$$\begin{aligned} & p(x) = ax + b \qquad \text{with} \qquad x \in [x_{\text{min}}, x_{\text{max}}] \\ \Rightarrow \qquad & p(x) = ax + \frac{1 - \frac{a}{2}(x_{\text{max}}^2 - x_{\text{min}}^2)}{x_{\text{max}} - x_{\text{min}}} \; . \end{aligned} \tag{5.64}$$

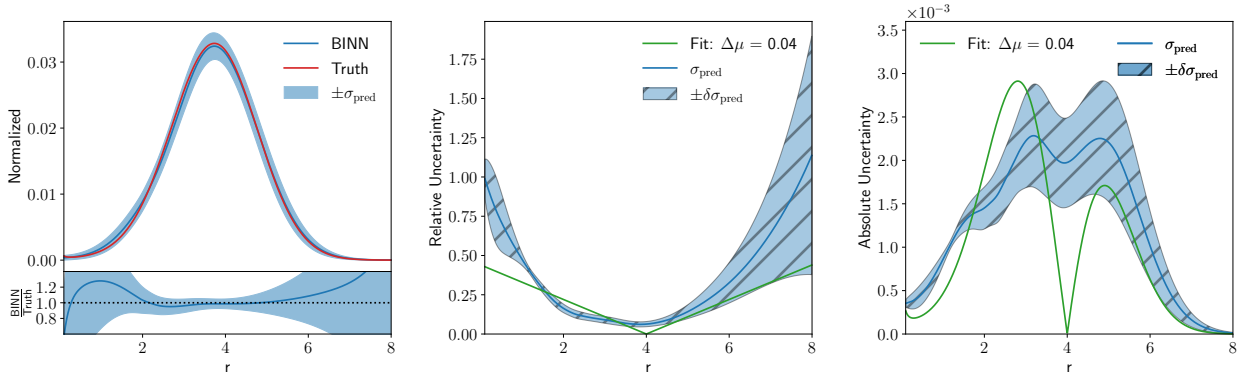


Figure 42: Density and predictive uncertainty distribution for the Gaussian ring. The uncertainty band on σ_{pred} is given by different radial directions. The green curve represents a 2-parameter fit to (5.68). Figure from Ref. [34].

For the corresponding 3-parameter fit we find

$$\sigma_{\text{pred}}^2 \equiv (\Delta p)^2 = \left(x - \frac{1}{2}\right)^2 (\Delta a)^2 + \left(1 + \frac{a}{2}\right)^2 (\Delta x_{\text{max}})^2 + \left(1 - \frac{a}{2}\right)^2 (\Delta x_{\text{min}})^2 \,. \tag{5.65}$$

While the slight shift of the minimum is not explained by this form, it does lift the minimum uncertainty to a finite value. If we evaluate the uncertainty as a function of x we cannot separate the effects from the two boundaries. The green line in Figure 41 gives a 2-parameter fit of Δa and Δx_{max} to the σ_{pred} distribution from the Bayesian INN.

While the linear ramp example could describe how an INN learns a smoothly falling distribution, like p_T of one of the particle in the final state, the more dangerous phase space features are sharp intermediate mass peaks. The corresponding toy model is a 2-dimensional Gaussian ring in terms of polar coordinates,

$$\begin{aligned} & p(r,\phi) = \mathcal{N}(r; \mu = 4, \sigma = 1) \qquad \text{with} \quad \phi \in [0, \pi] \\ \Leftrightarrow \qquad & p(x,y) = \mathcal{N}(\sqrt{x^2+y^2}; \mu = 4, \sigma = 1) \times \frac{1}{\sqrt{x^2+y^2}} \,. \end{aligned} \tag{5.66}$$

where the Jacobian $1/r$ ensures that both probability distributions are correctly normalized. We train the Bayesian INN on Cartesian coordinates, just like for the ramp discussed above. In Figure 42 we show the Cartesian density, evaluated on a line of constant angle. This form includes the Jacobian and leads to a shifted maximum. Again, the uncertainty covers the deviation of the learned from the true density.

Also in Figure 42 we see that the absolute predictive uncertainty shows a dip at the peak position. As before, this leads us to the interpretation in terms of appropriate fit parameters. For the Gaussian radial density we use the mean μ and the width σ, and the corresponding variations of the Cartesian density give us

$$\begin{aligned} \sigma_{\text{pred}} = \Delta p &\supset \left| \frac{d}{d\mu}\, p(x,y) \right| \Delta\mu \\ &= \frac{1}{r}\, \frac{1}{\sqrt{2\pi}\sigma} \left| \frac{d}{d\mu}\, e^{-(r-\mu)^2/(2\sigma^2)} \right| \Delta\mu \qquad (5.67) \\ &= \frac{p(x,y)}{r} \left| \frac{2(r-\mu)}{2\sigma^2} \right| \Delta\mu = \frac{p(x,y)}{r}\, \frac{|r-\mu|}{\sigma^2}\, \Delta\mu \,. \qquad (5.68) \end{aligned}$$

This turns out the dominant uncertainty, and it explains the local minimum at the peak position. Away from the peak, the uncertainty is dominated by the exponential behavior of the Gaussian.

The patterns we observe for the Bayesian INN indicate that our bilinear mapping is constructed very much like a fit. The network first identifies the family of functions which describe the underlying phase space density, and then it adjusts the relevant parameters, like the derivative of a falling function or the peak position of the Gaussian. We emphasize that this result is extracted from a joint network training on the density and the uncertainty on the density, not from a visualization. It also applies to normalizing flows only. Without a tractable Jacobian it is not clear how it can be generalized for example to GANs.

5.4.4 Event generation

As an application of generative normalizing flows, let us look at LHC event, *i.e.* unweighted four-vectors over phase space. In two steps we first need to control that our network has captured all features of the phase space density and only then can we compute the different uncertainties on the encoded density. As for GANs, we use unweighted events as training data, excluding detector effects because they just soften sharp phase space features. The production process

$$pp \to Z_{\mu\mu} + \{1,2,3\} \text{ jets} \tag{5.69}$$

is a challenge for generative networks, because it combines a sharp Z-resonance with the geometric separation of the jets and a variable phase space dimensions. If we assume that the muons are on-shell, but the jets come with a finite invariant mass, the phase space has $6 + n_{\text{jets}} \times 4$ dimensions. Standard cuts for reconstructed jets at the LHC are, as usual

$$p_{T,j} > 20 \text{ GeV} \qquad \text{and} \qquad \Delta R_{jj} > 0.4 \; . \tag{5.70}$$

Strictly speaking, such a hole changes the topology of the phase space and leads to a fundamental mismatch between the latent and phase spaces. However, for small holes we can trust the network to interpolate through the hole and then enforce the hole at the generation stage.

Since we have learned that preprocessing will make it easier for our network to learn the phase space density with high accuracy, we represent each final-state particle by

$$\{ \, p_T, \eta, \Delta\phi, m \, \} \; , \tag{5.71}$$

where we choose the harder of the two muons as the reference for the azimuthal angle and replace the azimuthal angle difference by $\text{atanh}(\Delta\phi/\pi)$, to create an approximately Gaussian distribution. We then use the jet cuts in (5.70) to re-define the transverse momenta as $\tilde{p}_T = \log(p_T - p_{T,\text{min}})$, giving us another approximately Gaussian phase space distribution.

After all of these preprocessing steps, we are left with the challenge of accommodating the variable jet multiplicity. While we will need individual generative networks for each final state multiplicity, we want to keep these networks from having to learning the basic features of their common hard process $pp \to Zj$. Just as for simulating jet radiation, we assume that the kinematics of the hard process depends little on the additional jets. This means our base network gets the one-hot encoded number of jets as condition. Each of the small, additional networks is conditioned on the training observables of the previous networks and on the number of jets. This means we define a likelihood loss for a conditional INN just like (5.46),

$$p_{\text{model}}(x) \;\to\; p_{\text{model}}(x|c,\theta) \qquad \Rightarrow \qquad \mathcal{L}_{\text{cINN}} = -\Big\langle \log p_{\text{model}}(x|c,\theta) \Big\rangle_{p_{\text{data}}} \; , \tag{5.72}$$

where the vector c includes the conditional jet number, for example. While the three networks for the jet multiplicities are trained separately, they form one generator with a given fraction of n-jet events.

To make our INN more expressive with a limited number of layers, we replace the affine coupling blocks of (5.44) with cubic-spline coupling blocks. The coupling layers are combined with random but fixed rotations to ensure interaction between all input variables. In Figure 43 we show a set of kinematic distributions from the high-statistics 1-jet process to the more challenging 3-jet process.

One way to systemically improve and control a precision INN-generator is to combine it with a discriminator. This way we can try to exploit different implicit biases of the discriminator and generator to improve our results. The simplest approach is to train the two networks independently and reweight all events with the discriminator output. This only requires that our discriminator output can be transformed into a probabilistic correction, so we train it by minimizing a cross-entropy loss in (5.21) to extract a probability

$$D(x_i) \to \begin{cases} 0 & \text{generator} \\ 1 & \text{truth/data} \end{cases} \tag{5.73}$$

for each event x_i. For a perfectly generated sample we should get $D(x_i) = 0.5$. The input to the three discriminators, one for each jet multiplicity, are the kinematic observables given in (5.71). In addition, we include a set of challenging kinematic correlations, so the discriminator gets the generated and training events in the form

$$x_i = \{p_{T,j}, \eta_j, \phi_j, M_j\} \cup \{M_{\mu\mu}\} \cup \{\Delta R_{2,3}\} \cup \{\Delta R_{2,4}, \Delta R_{3,4}\} \; . \tag{5.74}$$

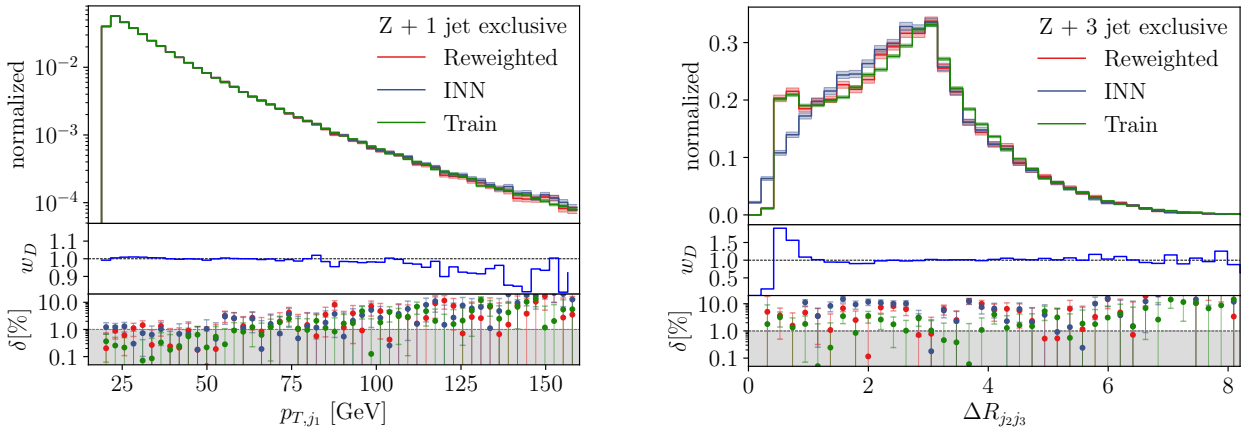


Figure 43: Discriminator-reweighted INN distributions for Z+jets production. The bottom panels show the average correction factor obtained from the discriminator output. Figure from Ref. [35].

If the the discriminator output has a probabilistic interpretation we can compute an event weight

$$w_D(x_i) = \frac{D(x_i)}{1 - D(x_i)} \quad \to \quad \frac{p_{\text{data}}(x_i)}{p_{\text{model}}(x_i)} \; . \tag{5.75}$$

We can see the effect of the additional discriminator in Figure 43. The deviation from truth is defined through a high-statistics version of the training dataset. The reweighted events are post-processed INN events with the average weight per bin shown in the second panel. While for some of the shown distribution a flat dependence $w_D = 1$ indicates that the generator has learned to reproduce the training data as well as the discriminator can tell, our more challenging distributions are significantly improved by the discriminator.

In the top panel of Figure 44 we show the $p_{T,j}$-distribution for $Z + 1$ jet production. We choose the simple 2-body final state to maximize the training statistics and the network's accuracy. In the second panel we show the relative deviation of the reweighted INN-generator and the training data from the high-statistics truth. While the network does not exactly match the precision of the training data, the two are not far apart, especially in the tails where training statistics becomes an issue. In the third panel we show the discriminator weight w_D defined in (5.75). In the tails of the distribution the generator systematically overestimates the true phase space density, leading to a correction $w_D(x) < 1$, before in the really poorly covered phase space regions the network predictions starts to fluctuate.

The fourth panel of Figure 44 shows the uncertainty reported by the Bayesian version of the INN-generator, using the architecture introduced in Sec. 5.4.1. The B-INN slightly overestimates the phase space density in the poorly populated tail, but the learned uncertainty on the phase space density covers this discrepancy reliably.

Moving on to systematic or theory uncertainties, the problem with generative networks and their unsupervised training is that we do not have access to the true phase space density. The network extracts the density itself, which means that any augmentation, like additional noise, will define either a different density or make the same density harder to learn. One way to include data augmentations is by turning the network training from unsupervised to supervised through an the augmentation by a parameter known to the network. This means we augment our data via a nuisance parameter describing the nature and size of a systematic or theory uncertainty, and train the network conditionally on this parameter. The nuisance parameter is added to the event format given in (5.41). As a simple example, we can introduce a theory uncertainty proportional to a transverse momentum, inspired by an electroweak Sudakov logarithm. As a function of the parameter a we shift the unit weights of the training events to

$$w = 1 + a \left(\frac{p_{T,j_1} - 15 \text{ GeV}}{100 \text{ GeV}} \right)^2 \; , \tag{5.76}$$

where the transverse momentum is given in GeV, we account for a threshold at 15 GeV, and we choose a quadratic scaling to enhance the effects of this augmentation. We train the Bayesian INN conditionally on a set of values $a = 0 \ldots 30$, just extending the conditioning of (5.72)

$$p_{\text{model}}(x) \quad \to \quad p_{\text{model}}(x|a, c, \theta) \tag{5.77}$$

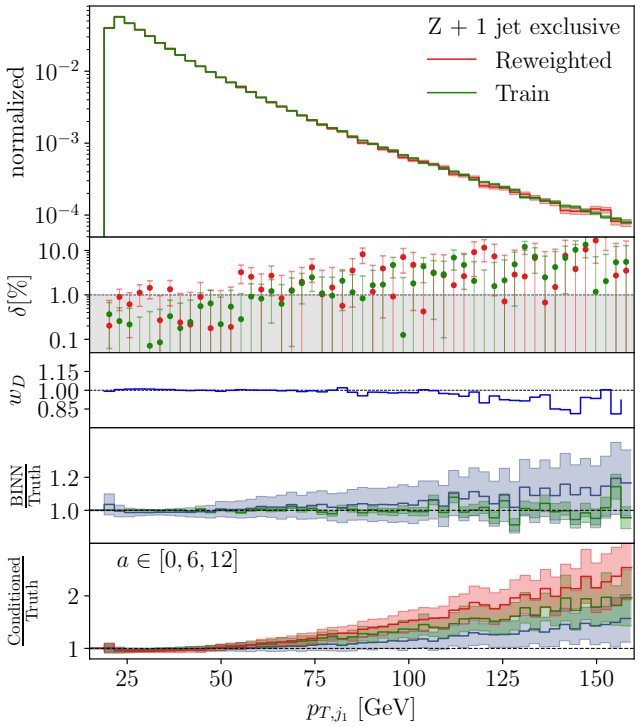


Figure 44: Illustration of uncertainty-controlled INN-generator. We show the reweighted p_{T,j_1}-distribution for the inclusive Z+jets sample, combined with the discriminator D, the B-INN uncertainty, and the sampled systematic uncertainty defined through the data augmentation of (5.76). Figure from Ref. [35].

In the bottom panel of Figure 44 we show generated distributions for different values of a. To incorporate the uncertainty described by the nuisance parameter in the event generation incorporating we sample the a-values for example using a standard Gaussian. In combination, we can cover a whole range of statistical, systematic, and theoretical uncertainties by using precise normalizing flows as generative networks. It is not clear if these flows are the final word on LHC simulations and event generation, but for precision simulations they appear to be the leading generative network architecture.

5.5 Diffusion networks

Diffusion networks are a new brand of generative networks which are similar to normalizing flows, but unlike the INN they are not fully bijective and symmetric. On the positive side, they tend to be more expressive that the relatively contrained normalizing flows. We will look at two distinctly different setups, one based on a discrete time series in Sec. 5.5.1 and another one based on differential equations in Sec. 5.5.2.

5.5.1 Denoising diffusion probabilistic model

Looking at the INN mapping illustrated in Eq. (5.42), we can interpret this basic relation as a time evolution from the phase space distribution to the latent distribution or back,

$$p_{\text{model}}(x_0) \quad \underset{\leftarrow \text{backward}}{\overset{\text{forward} \rightarrow}{\longleftrightarrow}} \quad p_{\text{latent}}(x_T) \; , \tag{5.78}$$

identifying the original parameters $x \to x_0$ and $r \to x_T$ in the spirit of a discrete time series with $t = 0 \ldots T$. The step-wise adding of Gaussian noise defines the forward direction of Denoising Diffusion Probabilistic Models (DDPMs). The task of the reverse, generative process is to to denoise the diffused data.

The forward process, by definition, turns a phase space distribution into Gaussian noise. The multi-dimensional distribution is factorized into independent steps,

$$p(x_1,...,x_T|x_0) = \prod_{t=1}^{T} p(x_t|x_{t-1}) \qquad \text{with} \qquad p(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t) \; . \tag{5.79}$$

Each step describes a conditional probability and adds Gaussian noise with an appropriately chosen variance β_t and mean $\sqrt{1-\beta_t}x_{t-1}$ to generate x_t. Following the original paper we can choose a linear scaling $\beta_t \sim 2\cdot 10^{-2}(t-1)/T$. Naively, we would add noise with mean x_{t-1}, but in that case each step would broaden the distribution, since independent noise sources add their widths in quadrature. To compensate, we add noise with scaled mean $\sqrt{1-\beta_t}x_{t-1}$. In that case we can combine all Gaussian convolutions and arrive at

$$\begin{aligned} p(x_t|x_0) &= \int dx_1 ... dx_{t-1} \prod_{i=1}^{t} p(x_i|x_{i-1}) \\ &= \mathcal{N}(x_t; \sqrt{1-\bar\beta_t}x_0, \bar\beta_t) \qquad \text{with} \qquad 1-\bar\beta_t = \prod_{i=1}^{t}(1-\beta_i) \; . \end{aligned} \tag{5.80}$$

The reverse process in (5.78) starts with a Gaussian and gradually transforms it into the phase-space distribution through the same discrete steps as (5.79). The corresponding generative network approximates each forward step and should produce the correct phase-space distribution

$$\begin{aligned} p_{\text{model}}(x_0) &= \int dx_1 ... dx_T \; p_{\text{latent}}(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \\ \text{with} \qquad p_\theta(x_{t-1}|x_t) &= \mathcal{N}(x_{t-1}; \mu_\theta(x_t,t), \sigma_\theta^2(x_t,t)) \; . \end{aligned} \tag{5.81}$$

Here, μ_θ and σ_θ are learnable parameters describing the individual conditional probability slices $x_t \to x_{t-1}$. It turns out that in numerical practice we can simplify it to $\sigma_\theta^2(x_t,t) \to \sigma_t^2$.

To link the forward and reverse directions, we first apply Bayes' theorem on each slice defined in (5.79) to give us the reverse $p(x_{t-1}|x_t)$. The problem with this inversion is that the full probability distribution $p(x_1,...,x_T|x_0)$ in Eq.(5.79) is conditioned on x_0. With this dependence we can compute the conditioned forward posterior as a Gaussian with an x_0-dependent mean,

$$\begin{aligned} p(x_{t-1}|x_t,x_0) &= \frac{p(x_t|x_{t-1})p(x_{t-1}|x_0)}{p(x_t|x_0)} = \mathcal{N}(x_{t-1}; \hat\mu(x_t,x_0), \hat\beta_t) \\ \text{with} \quad \hat\mu(x_t,x_0) &= \frac{\sqrt{1-\bar\beta_{t-1}}\beta_t}{\bar\beta_t}x_0 + \frac{\sqrt{1-\beta_t}\bar\beta_{t-1}}{\bar\beta_t}x_t \quad \text{and} \quad \hat\beta_t = \frac{\bar\beta_{t-1}}{\bar\beta_t}\beta_t \; . \end{aligned} \tag{5.82}$$

For training the DDPM network we need to just approximate a set of Gaussians, (5.82), with their learned counterparts in (5.81).

The loss function of the diffusion network is the same sampled likelihood as for the INN, given in (5.46), which we can simplify by inserting and then dividing by $p(x_1,...,x_T|x_0)$ following (5.79)

$$\begin{aligned} -\Big\langle \log p_{\text{model}}(x_0) \Big\rangle_{p_{\text{data}}} &= -\int dx_0 \; p_{\text{data}}(x_0) \; \log \left(\int dx_1 ... dx_T \; p_{\text{latent}}(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t) \right) \\ &= -\int dx_0 \; p_{\text{data}}(x_0) \; \log \left(\int dx_1 ... dx_T \; p_{\text{latent}}(x_T) p(x_1,...,x_T|x_0) \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right) \\ &= -\int dx_0 \; p_{\text{data}}(x_0) \; \log \left\langle p_{\text{latent}}(x_T) \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right\rangle_{p(x_1,...,x_T|x_0)} \end{aligned} \tag{5.83}$$

This expression includes a logarithm of an expectation value. There is a standard relation, Jensen's inequality, for convex functions,

$$f(\langle x \rangle) \le \langle f(x) \rangle \; . \tag{5.84}$$

Convex means that if we linearly interpolate between two points of the function, the interpolation lies above the function. This is obviously true for a function around the minimum, where the Taylor series gives a quadaric (or even-power) approximation. We use this inequality for our negative log-likelihood around the minimum. It provides an upper limit to the negative log-likelihood, which we minimize instead,

$$\begin{aligned}
-\Big\langle \log p_{\text{model}}(x_0) \Big\rangle_{p_{\text{data}}} &\leq -\int dx_0\; p_{\text{data}}(x_0) \left\langle \log \left(p_{\text{latent}}(x_T) \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right) \right\rangle_{p(x_1,\ldots,x_T|x_0)} \\
&= -\int dx_0 ... dx_T\; p_{\text{data}}(x_0)\; p(x_1,...,x_T|x_0)\; \log \left(p_{\text{latent}}(x_T) \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right) \\
&\equiv -\int dx_0 ... dx_T\; p(x_0,...,x_T)\; \log \left(p_{\text{latent}}(x_T) \prod_{t=1}^{T} \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right) \\
&= \left\langle -\log p_{\text{latent}}(x_T) - \sum_{t=1}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} \right\rangle_{p(x_0,\ldots,x_T)}
\end{aligned} \tag{5.85}$$

Now we use Bayes' theorem for the individual slices $p_\theta(x_{t-1}|x_t)$ and compare them with the reference form from (5.82),

$$\begin{aligned}
-\Big\langle \log p_{\text{model}}(x_0) \Big\rangle_{p_{\text{data}}} &\leq \left\langle -\log p_{\text{latent}}(x_T) - \sum_{t=2}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{p(x_t|x_{t-1})} - \log \frac{p_\theta(x_0|x_1)}{p(x_1|x_0)} \right\rangle_{p(x_0,\ldots,x_T)} \\
&= \left\langle -\log p_{\text{latent}}(x_T) - \sum_{t=2}^{T} \log \frac{p_\theta(x_{t-1}|x_t) p(x_{t-1}|x_0)}{p(x_{t-1}|x_t,x_0) p(x_t|x_0)} - \log \frac{p_\theta(x_0|x_1)}{p(x_1|x_0)} \right\rangle_{p(x_0,\ldots,x_T)} \\
&= \left\langle -\log p_{\text{latent}}(x_T) - \sum_{t=2}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{p(x_{t-1}|x_t,x_0)} - \log \frac{p(x_1|x_0)}{p(x_T|x_0)} - \log \frac{p_\theta(x_0|x_1)}{p(x_1|x_0)} \right\rangle_{p(x_0,\ldots,x_T)} \\
&= \left\langle -\log \frac{p_{\text{latent}}(x_T)}{p(x_T|x_0)} - \sum_{t=2}^{T} \log \frac{p_\theta(x_{t-1}|x_t)}{p(x_{t-1}|x_t,x_0)} - \log p_\theta(x_0|x_1) \right\rangle_{p(x_0,\ldots,x_T)}
\end{aligned} \tag{5.86}$$

As usual, we ignore terms which do not depend on the network weights θ,

$$\begin{aligned}
\Big\langle \log p_{\text{model}}(x_0) \Big\rangle_{p_{\text{data}}} &\leq \sum_{t=2}^{T} \left\langle \log \frac{p(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)} \right\rangle_{p(x_0,\ldots,x_T)} - \Big\langle \log p_\theta(x_0|x_1) \Big\rangle_{p(x_0,\ldots,x_T)} + \text{const} \\
&= \sum_{t=2}^{T} \int dx_0 ... dx_T\; p(x_0,...,x_T) \log \frac{p(x_{t-1}|x_t,x_0)}{p_\theta(x_{t-1}|x_t)} - \Big\langle \log p_\theta(x_0|x_1) \Big\rangle_{p(x_0,\ldots,x_T)} + \text{const} \\
&= \sum_{t=2}^{T} \Big\langle D_{\text{KL}}[p(x_{t-1}|x_t,x_0), p_\theta(x_{t-1}|x_t)] \Big\rangle_{p(x_0,x_t)} - \Big\langle \log p_\theta(x_0|x_1) \Big\rangle_{p(x_0,\ldots,x_T)} + \text{const} \\
&\approx \sum_{t=2}^{T} \Big\langle D_{\text{KL}}[p(x_{t-1}|x_t,x_0), p_\theta(x_{t-1}|x_t)] \Big\rangle_{p(x_0,x_t)}
\end{aligned} \tag{5.87}$$

The sampling follows $p(x_0,x_t) = p(x_t|x_0)\; p_{\text{data}}(x_0)$. The second sampled term will be numerically negligible compared to the first $T-1$ terms. The KL-divergence compares the two Gaussians from (5.82) and (5.81), with the two means $\mu_\theta(x_t,t)$ and $\hat{\mu}(x_t,x_0)$ and the two standard deviations σ_t^2 and $\hat{\beta}_t$,

$$\boxed{\mathcal{L}_{\text{DDPM}} = \sum_{t=2}^{T} \left\langle \frac{1}{2\sigma_t^2} \left| \hat{\mu} - \mu_\theta \right|^2 \right\rangle_{p(x_0,x_t)}} . \tag{5.88}$$

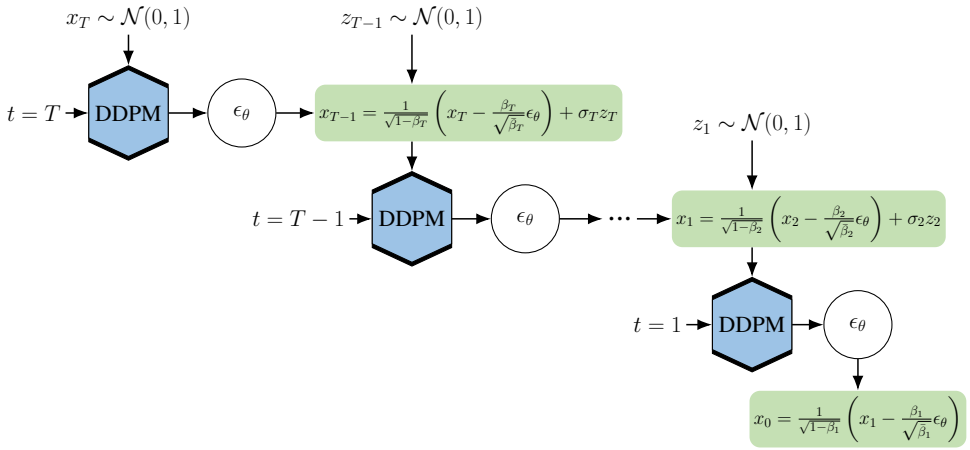


Figure 45: DDPM sampling algorithm, Figure from Ref. [36].

To provide $\hat{\mu}$ we use the reparametrization trick on $x_t(x_0, \epsilon)$ as given in (5.80)

$$\begin{aligned} & x_t(x_0,\epsilon) = \sqrt{1-\bar{\beta}_t}x_0 + \sqrt{\bar{\beta}_t}\epsilon \qquad \text{with} \qquad \epsilon \sim \mathcal{N}(0,1) \\ \Leftrightarrow \qquad & x_0(x_t,\epsilon) = \frac{1}{\sqrt{1-\bar{\beta}_t}}\left(x_t - \sqrt{\bar{\beta}_t}\epsilon\right) , \end{aligned} \tag{5.89}$$

This form we insert into (5.82) to find, after a few simple steps.

$$\hat{\mu}(x_t,\epsilon) = \frac{1}{\sqrt{1-\beta_t}}\left(x_t(x_0,\epsilon) - \frac{\beta_t}{\sqrt{\bar{\beta}_t}}\epsilon\right) . \tag{5.90}$$

The same method can be applied to provide $\mu_\theta(x_t, t) \equiv \hat{\mu}(x_t, \epsilon_\theta)$, in terms of the trained network regression ϵ_θ.

The derivation of the Bayesian INN in Sec. 5.4.1 can just be copied to define a Bayesian DDPM. Its loss follows from Eqs.(5.88) and (5.48), with a sampling over network parameters $\theta \sim q(\theta)$ and the regularization term,

$$\mathcal{L}_{\text{B-DDPM}} = \Big\langle \mathcal{L}_{\text{DDPM}} \Big\rangle_{\theta\sim q} + D_{\text{KL}}[q(\theta), p(\theta)] . \tag{5.91}$$

We turn the deterministic DDPM into the B-DDPM through two steps, (i) swapping the deterministic layers to the corresponding Bayesian layers, and (ii) adding the regularization term to the loss. To evaluate the Bayesian network sample over the network weight distribution.

For the DDPM training we start with a phase-space point $x_0 \sim p_{\text{data}}(x_0)$ drawn from the true phase space distribution and also draw the time step t from a uniform distribution and ϵ from a standard Gaussian. We then use (5.90) to compute the diffused data point after t time steps, x_t. This means the uses many different time steps t for many different phase-space points x_0 to learn the step-wise reversed process. , which is why we use a relatively simple residual dense network architecture, which is trained over many epochs.

The (reverse) DDPM sampling is illustrated in Figure 45. We start with $x_T \sim p_{\text{latent}}(x_T)$, drawn from the Gaussian latent space distribution. Combining the learned ϵ_θ and the drawn Gaussian noise $z_{T-1} \sim \mathcal{N}(0,1)$ we calculate x_{T-1}, assumed to be a slightly less diffused version of x_T. We repeat this sampling until we reach the phase space distribution of x_0. Because the network needs to predict ϵ_θ T times, it is slower than classic generative networks like VAEs, GANs, or INNs.

From Sec. 5.4.1 we know that we can ilustrate the training of a generative network using a simple toy model, for example the linear ramp defined in (5.61). In Figure 46 we show the cooresponding results to the B-INN results from Figure 41. The key results is that the DDPM learns the simple probability distribution with high precision and without a bias. Moreover, the absolute predictive uncertainty on the estimated density has a minimum around $x_2 \sim 0.7$, albeit less pronounced than for the B-INN. This suggests that the DDPM has some similarity to the INN, but its implicit bias on the fitted function and its expressivity are different.

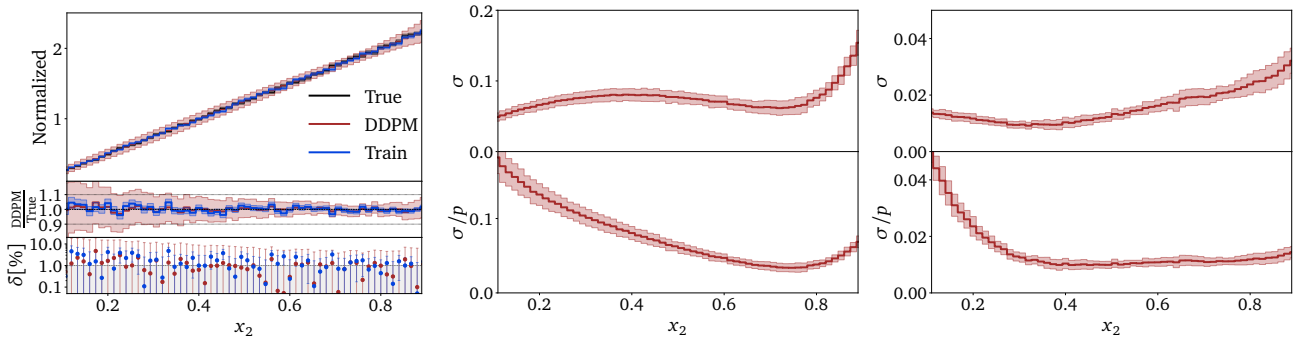


Figure 46: Density and predictive uncertainty distribution for a linear wedge ramp using diffusion networks. We show DDPM results (left and center) and CFM results (right). The uncertainty on σ_{pred} is given by its x_1-variation. Figure from Ref. [36].

5.5.2 Conditional flow matching

An alternative approach to diffusion networks is Conditional Flow Matching (CFM). Like the DDPM, it uses a time evolution to transform a phase space distributions into Gaussian noise, but instead of a discrete chain of conditional probabilities it constructs and solves a continuous ordinary differential equation (ODE)

$$\boxed{\frac{dx(t)}{dt} = v(x(t),t)} \,, \tag{5.92}$$

where $v(x(t),t)$ is called the velocity field. We will learn the velocity field to generate samples by integrating this ODE from $t=1$ to $t=0$. The ODE can be linked to a probability density $p(x,t)$ through the continuity equation

$$\boxed{\frac{\partial p(x,t)}{\partial t} + \nabla_x \left[p(x,t) v(x,t)\right] = 0} \,. \tag{5.93}$$

These two equations are equivalent in that for a given probability density path $p(x,t)$ any velocity field $v(x,t)$ describing the sample-wise evolution (5.92) will be a solution of (5.93), and vice versa. We will use the continuity equation to learn the velocity field $v_\theta(x,t) \approx v(x,t)$ in the network training and then use the ODE with $v_\theta(x,t)$ as a generator.

To realize the diffusion model ansatz of (5.78) we need to express the velocity field in terms of our known density $p(x,t)$ with the boundary conditions

$$p(x,t) \to \begin{cases} p_{\text{data}}(x) & t \to 0 \\ p_{\text{latent}}(x) = \mathcal{N}(x;0,1) & t \to 1 \end{cases} , \tag{5.94}$$

where we use x as the argument of p_{latent}, rather than r, to illustrate that it is an evolved form of the phase space distribuion x. In the forward, diffusion direction, the time evolution goes from a phase space point x_0 to the latent standard Gaussian. In a conditional form we can write a simple linear interpolation

$$\begin{aligned} x(t|x_0) = (1-t)x_0 + tr \to \begin{cases} x_0 & t \to 0 \\ r \sim \mathcal{N}(0,1) & t \to 1 \end{cases} \\ \Leftrightarrow \qquad p(x,t|x_0) = \mathcal{N}(x;(1-t)x_0,t) \,. \end{aligned} \tag{5.95}$$

This conditional time evolution is similar to the DDPM case in (5.80). Formally, we can compute the full probability path from it, ensuring that it fulfills both boundary conditions of (5.94),

$$\begin{aligned} p(x,t) &= \int dx_0 \; p(x,t|x_0) \; p_{\text{data}}(x_0) \\ \text{with} \qquad p(x,0) &= \int dx_0 \; p(x,0|x_0) \; p_{\text{data}}(x_0) = \int dx_0 \; \delta(x-x_0) \; p_{\text{data}}(x_0) = p_{\text{data}}(x) \\ \text{and} \qquad p(x,1) &= \int dx_0 \; p(x,1|x_0) \; p_{\text{data}}(x_0) = \mathcal{N}(x;0,1) \int dx_0 \; p_{\text{data}}(x_0) = \mathcal{N}(x;0,1) \,. \end{aligned} \tag{5.96}$$

For a generative model we need the velocity corresponding to this probability density path. We start with the conditional velocity, associated with $p(x,t|x_0)$, and combine (5.92) and (5.95) to

$$\begin{aligned} v(x(t|x_0),t|x_0) &= \frac{dx(t|x_0)}{dt} \\ &= \frac{d}{dt}\left[(1-t)x_0 + tr\right] = -x_0 + r \ . \end{aligned} \tag{5.97}$$

Our linear interpolation leads to a time-constant velocity, which solves the continuity equation for $p(x,t|x_0)$ because we construct it as a solution to the ODE

$$\frac{\partial p(x,t|x_0)}{\partial t} + \nabla_x \left[p(x,t|x_0)v(x,t|x_0)\right] = 0 \ . \tag{5.98}$$

Just like for the probability paths, (5.96), we can compute the unconditional velocity from its conditional counterpart,

$$\begin{aligned} \frac{\partial p(x,t)}{\partial t} &= \frac{\partial}{\partial t}\int dx_0 \ p(x,t|x_0)p_{\text{data}}(x_0) \\ &= \int dx_0 \ \frac{\partial p(x,t|x_0)}{\partial t} \ p_{\text{data}}(x_0) \\ &= -\int dx_0 \ \nabla_x \left[p(x,t|x_0)v(x,t|x_0)\right] \ p_{\text{data}}(x_0) \\ &= -\nabla_x \left[p(x,t)\int dx_0 \ \frac{p(x,t|x_0)v(x,t|x_0)p_{\text{data}}(x_0)}{p(x,t)}\right] \equiv -\nabla_x \left[p(x,t)v(x,t)\right] \\ \Leftrightarrow \qquad v(x,t) &= \int dx_0 \ \frac{p(x,t|x_0)v(x,t|x_0)p_{\text{data}}(x_0)}{p(x,t)} \ . \end{aligned} \tag{5.99}$$

While the conditional velocity in (5.97) describes a trajectory between a normal distributed and a phase space sample x_0 that is specified in advance, the full velocity in (5.99) evolves samples from p_{data} to p_{latent} and vice versa.

Training the CFM means learning the velocity field in (5.99), a simple regression task, $v(x,t) \approx v_\theta(x,t)$. The straightforward choice is the MSE-loss,

$$\begin{aligned} \mathcal{L}_{\text{FM}} &= \Big\langle \left[v_\theta(x,t) - v(x,t)\right]^2 \Big\rangle_{t,x\sim p(x,t)} \\ &= \Big\langle v_\theta(x,t)^2 \Big\rangle_{t,x\sim p(x,t)} - \Big\langle 2v_\theta(x,t)v(x,t) \Big\rangle_{t,x\sim p(x,t)} + \text{const} \ , \end{aligned} \tag{5.100}$$

where the time is sampled uniformly over $t \in [0,1]$. Again, we can start with the conditional path in (5.95) and calculate the conditional velocity in (5.97) for the MSE loss. We rewrite the above loss in terms of the conditional quantities, so the first term becomes

$$\begin{aligned} \Big\langle v_\theta(x,t)^2 \Big\rangle_{t,x\sim p(x,t)} &= \left\langle \int dx \ p(x,t)v_\theta(x,t)^2 \right\rangle_t \\ &= \left\langle \int dx v_\theta(x,t)^2 \int dx_0 \ p(x,t|x_0)p_{\text{data}}(x_0) \right\rangle_t \\ &= \Big\langle v_\theta(x,t)^2 \Big\rangle_{t,x_0\sim p_{\text{data}},x\sim p(x,t|x_0)} \\ &= \Big\langle v_\theta(x(t|x_0),t)^2 \Big\rangle_{t,x_0\sim p_{\text{data}},r} \end{aligned} \tag{5.101}$$

In the last term we use the simple form of $x(t|x_0)$ given in (5.95), which needs to be sampled over r. Similarly, we rewrite

the second loss term as

$$\begin{aligned}
-2\Big\langle v_\theta(x,t)v(x,t)\Big\rangle_{t,x\sim p(x,t)} &= -2\Bigg\langle \int dx\; p(x,t)v_\theta(x,t)\; \frac{\int dx_0 p(x,t|x_0)v(x,t|x_0)p_{\text{data}}(x_0)}{p(x,t)} \Bigg\rangle_t \\
&= -2\Bigg\langle \int dx dx_0\; v_\theta(x,t)\; v(x,t|x_0)\; p(x,t|x_0)\; p_{\text{data}}(x_0) \Bigg\rangle_t \\
&= -2\Big\langle v_\theta(x,t)\; v(x,t|x_0)\Big\rangle_{t,x_0\sim p_{\text{data}},x\sim p(x,t|x_0)} \\
&= -2\Big\langle v_\theta(x(t|x_0),t)\; v(x(t|x_0),t|x_0)\Big\rangle_{t,x_0\sim p_{\text{data}},r} \; . \qquad (5.102)
\end{aligned}$$

The (conditional) Flow Matching loss of (5.100) then becomes

$$\boxed{\mathcal{L}_{\text{CFM}} = \Big\langle \left[v_\theta(x(t|x_0),t) - v(x(t|x_0),t|x_0)\right]^2 \Big\rangle_{t,x_0\sim p_{\text{data}},r}} \; . \qquad (5.103)$$

We can compute it using the linear ansatz from (5.95) as

$$\mathcal{L}_{\text{CFM}} = \left\langle \left[v_\theta(x(t|x_0),t) - \frac{dx(t|x_0))}{dt}\right]^2 \right\rangle_{t,x_0\sim p_{\text{data}},r} = \Big\langle \left[v_\theta((1-t)x_0+tr,t) - (r-x_0)\right]^2 \Big\rangle_{t,x_0\sim p_{\text{data}},r} \; . \qquad (5.104)$$

As usually, we can turn the CFM into a Bayesian generative network. For the Bayesian INN or the Bayesian DDPM the loss is a sum of the likelihood loss and a KL-divergence regularization, shown in Eqs.(5.48) and (5.91). Unfortunately, the CFM loss in (5.103) is not a likelihood loss. To mimic the usual setup we still modify the CFM loss by switching to Bayesian network layers and adding a KL-regularization,

$$\mathcal{L}_{\text{B-CFM}} = \Big\langle \mathcal{L}_{\text{CFM}} \Big\rangle_{\theta\sim q(\theta)} + cD_{\text{KL}}[q(\theta),p(\theta)]. \qquad (5.105)$$

While for a likelihood loss the factor c is fixed by Bayes' theorem, in this case it is a free hyperparameter. However, we find that the network predictions and their associated uncertainties are very stable when varying it over several orders of magnitude.

To train the CFM we sample a data point $x_0 \sim p_{\text{data}}(x_0)$ and $r \sim \mathcal{N}(0,1)$ as the starting and end points of a trajectory, as well as a time from a uniform distribution. We first compute $x(t|x_0)$ according to (5.95) and then $v(x(t|x_0),t|x_0)$ following (5.97). The point $x(t|x_0)$ and the time t are passed to a neural network which encodes the conditional velocity field

$$v_\theta(x(t|x_0),t) \approx v(x,t|x_0) \; . \qquad (5.106)$$

One property of the training algorithm is that the same network input, a time t and a position $x(t|x_0)$, can be produced by many different trajectories with different conditional velocities. While the network training is based on a wide range of possible trajectories, the CFM loss in (5.103) ensures that sampling over many trajectories returns a well-defined velocity field.

Once the CFM network is trained, the generation of new samples is straightforward. We start by drawing a sample from the latent distribution $r \sim p_{\text{latent}} = \mathcal{N}(0,1)$ and calculate its time evolution by numerically solving the ODE backwards in time from $t=1$ to $t=0$

$$\begin{aligned}
&\frac{d}{dt}x(t) = v_\theta(x(t),t) \qquad \text{with} \quad r = x(t=1) \\
\Rightarrow \qquad &x_0 = r - \int_0^1 v_\theta(x,t)dt \equiv G_\theta(r) \; , \qquad (5.107)
\end{aligned}$$

This generation is fast, as is the CFM training, because we can rely on established ODE solvers. Under mild regularity assumptions this solution defines a bijective transformation between the latent space sample and the phase space sample $G_\theta(x_1)$, similar to the INN case.

Like the INN and unlike the DDPM, the CFM network also allows us to calculate phase space likelihoods. Making use of the continuity equation, (5.93), we can write

$$\begin{aligned}\frac{dp(x,t)}{dt} &= \frac{\partial p(x,t)}{\partial t} + [\nabla_x p(x,t)]\ v(x,t) \\ &= \frac{\partial p(x,t)}{\partial t} + \nabla_x \left[p(x,t) v(x,t)\right] - p(x,t)\left[\nabla_x v(x,t)\right] \\ &= -p(x,t)\ \nabla_x v(x,t)\ . \end{aligned} \tag{5.108}$$

Its solution can be cast in the INN notation from (5.43),

$$\begin{aligned}\frac{p(r,t=1)}{p(x_0,t=0)} &= \exp\left(-\int_0^1 dt\ \nabla_x v(x(t),t)\right) \\ &\equiv \frac{p_{\text{latent}}(G_\theta^{-1}(x_0))}{p_{\text{model}}(x_0)} = \left|\det \frac{\partial G_\theta^{-1}(x_0)}{\partial x_0}\right|^{-1} \\ \Leftrightarrow \qquad \left|\det \frac{\partial G_\theta^{-1}(x_0)}{\partial x_0}\right| &= \exp\left(\int_0^1 dt\ \nabla_x v_\theta(x(t),t)\right)\ . \end{aligned} \tag{5.109}$$

Calculating the Jacobian requires integrating over the divergence of the learned velocity field, which is fast if we use automatic differentiation.

To understand how the generative network learn, we can again use the 2-dimensional linear wedge combined with the Bayesian setup. In the right panel of Figure 46 we show the corresponding distribution. First, we see that the scale of the uncertainty is a factor five below the DDPM uncertainty, which means that the CFM is easily and reliably trained for a small number of dimensions. In addition, we see that the minimum in the absolute uncertainty is even flatter and at $x_2 \sim 0.3$. Again, this is reminiscent of the fit-like INN behavior, but much less pronounced.

An interesting aspect of the CFM is that, unlike the other generative models we have discussed, nothing in our derivation requires the latent distribution to be a Gaussian. Instead of Eq.(5.42) or Eq,(5.78) we can link any two distributions, sample from one and into the second. An interesting LHC application is the generation of events from a narrow phase space into a wider phase space. This kind of problem occurs for collinear or soft or on-shell subtraction terms, where we need to model cancellations between phase spaces with different numbers of particles in the final state. Similarly, we can try to generate off-shell decay configurations from an on-shell or Breit-Wigner approximation. Such off-shell effects are expensive to simulate and numerically suppressed. The advantage of training a generative network to modify the on-shell distributions as compared to training a generative network to simulate off-shell events is that the on-shell events already include the full correlations, so the network only has to lears small modifications rather than the correlations themselves.

Let us look at the production of off-shell top quarks

$$pp \to (be^+\nu_e)\ (\bar{b}\mu^-\nu_\mu)\ . \tag{5.110}$$

The propagators of massive and unstable intermediate particles with mass m include the particle width Γ and have the form

$$\frac{\Gamma_{\text{part}}}{(s-m^2)^2 + m^2\Gamma_{\text{tot}}^2} \overset{\Gamma\to 0}{\longrightarrow} \Gamma_{\text{part}}\ \frac{\pi}{\Gamma_{\text{tot}}}\ \delta(s-m^2) \equiv \pi\ \text{BR}_{\text{part}}\ \delta(s-m^2)\ . \tag{5.111}$$

For vanishing widths, the decay propagator leads to a phase space constraint and the branching ratio BR_{part} into the partial decay channel given by the off-shell process. For small, but finite width we can use this so-called Breit-Wigner propagator to simulate off-shell effects. Far away from the on-shell pole this description is wrong, because Eq.(5.110) does not actually require top quarks to appear in the Feynman diagrams. In Fig. 47 we show the on-shell and off-shell distributions for the reconstructed top mass.

Generalizing Eq.(5.42) the generative task becomes

$$\text{on-shell } x \sim p_{\text{on}}(x) \qquad \longleftrightarrow \qquad \text{off-shell } x \sim p_{\text{model}}(x|\theta) \approx p_{\text{off}}(x)\ . \tag{5.112}$$

The time-dependent probability distributions of Eq.(5.94) become

$$p(x,t) \to \begin{cases} p_{\text{off}}(x) & t \to 0 \\ p_{\text{on}}(x) & t \to 1\ , \end{cases} \tag{5.113}$$

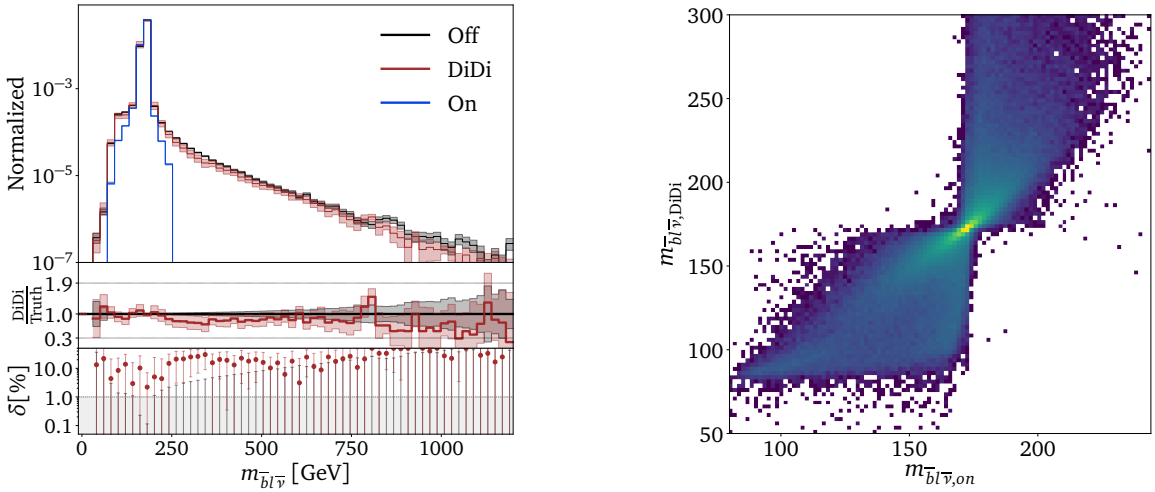


Figure 47: Left: kinematic distributions of on-shell vs off-shell top pair production, compared with the generated off-shell distributions (DiDi). Right: migration plot between on-shell and generated off-shell configurations. Figure from Ref. [37].

and the loss function from Eq.(5.103) is samples from the on-shell and off-shell events,

$$\mathcal{L}_{\mathrm{CFM}} = \left\langle \left[v_\theta((1-t)x_0 + tx_1, t) - (x_1 - x_0) \right]^2 \right\rangle_{t, x_0 \sim p_{\mathrm{off}}, x_1 \sim p_{\mathrm{on}}} . \tag{5.114}$$

For the training, we can use paired events, if available, but we do not have to. This kind of mapping can also be achieved using the Flows-For-Flows setup or the so-called Schrödinger bridge, but as a generalization of the CFM model to Direct Diffusion (DiDi) it is particularly obvious.

In the left panel of Fig. 47 we compare the performance of a simple direct diffusion network to the on-shell base distribution and the off-shell target distribution. The generated off-shell events reproduce the correct distribution in the, apparently, extrapolated phase space region with reasonable precision. Deviations from the target distributions are covered by the error bars of the Bayesian version. As always, the performance of the DiDi generator can be improved through classifier reweighting.

Given that the off-shell extrapolation does not follow a well-defined simulation model, the DiDi training on unmatched event samples has to construct some kind of optimal transport prescription. This is already discussed in some of the original literature on CFM. We can illustrate the learned optimal transport prescription through the correlations between each starting phase space point x_0 and the generated off-shell phase space point x_1. In the right panel of Fig. 47 we show this correlation, for the transverse momentum of the anti-bottom quark. As one might expect, the optimal transport keeps most events close to where they are and avoids moving event from one side of the top mass peak to the other.

6 Inverse problems and inference

If we use a simulation chain in the forward direction, typical analyses starts with a new, theory-inspired hypothesis encoded in a Lagrangian. For every point in the model parameter space we simulate events, compare them to the measured data using likelihood methods, and then discard the model hypothesis. This approach is inefficient for a variety of reasons:

1. Physics hypotheses have free model parameters like masses or couplings, and we usually need to simulate events for each point in model space.
2. There is a limit in electroweak or QCD precision to which we can reasonably include predictions in our standard simulation tools. Beyond this limit we can, for instance, only compute a limited set of kinematic distributions, which excludes these precision prediction from standard analyses.
3. Without a major effort it is impossible to derive competitive limits on a new model by recasting an existing analysis, if the analysis requires the full experimental and theoretical simulation chains.

These three shortcomings point to the same task: we need to invert the simulation chain, apply this inversion to the measured data, and compare hypotheses at the level of the hard scattering. Detector unfolding is a known, but non-standard application. For hadronization and fragmentation an approximate inversion is standard in that we always apply jet algorithms to extract simple parton properties from the complex QCD jets. Removing QCD jets from the hard process is a standard task in any analysis using jets for the hard process and leads to nasty combinatorial backgrounds. Unfolding to parton level is being applied in top physics, assuming that the top decays are correctly described by the Standard Model. Finally, unfolding all the way to the parton-level is called the matrix element method and has been applied to Tevatron signatures, for example single top production. All of these inverse simulation tasks have been tackled with classical methods, and we will see how the can benefit from modern neutral networks.

Inverse problems in particle physics can be illustrated most easily for the case of detector effects. As a one-dimensional binned toy example we look at a parton-level distribution $f_{\text{parton},j}$ which gets transformed into $f_{\text{reco},j}$ at detector or reconstruction level. We can model these detector effects as part of the forward simulation through a response matrix G_{ij},

$$f_{\text{reco},i} = \sum_{j=1}^{N} G_{ij}\ f_{\text{parton},j}\ . \tag{6.1}$$

We postulate the existence of an inversion with the matrix $\overline{G}$ through

$$f_{\text{parton},k} = \sum_{i=1}^{N} \overline{G}_{ki} f_{\text{reco},i} = \sum_{j=1}^{N} \left(\sum_{i=1}^{N} \overline{G}_{ki} G_{ij} \right) f_{\text{parton},j} \quad \text{with} \quad \sum_{i=1}^{N} \overline{G}_{ki} G_{ij} = \delta_{kj}\ . \tag{6.2}$$

If we assume that we know the N^2 entries of G, this form gives us the N^2 conditions to compute its inverse $\overline{G}$. We illustrate this one-dimensional binned case with a toy-smearing matrix

$$G = \begin{pmatrix} 1-\epsilon & \epsilon & 0 \\ \epsilon & 1-2\epsilon & \epsilon \\ 0 & \epsilon & 1-\epsilon \end{pmatrix}\ . \tag{6.3}$$

We can assume $\epsilon \ll 1$, but we do not have to. We look at two input vectors, keeping in mind that in an unfolding problem we typically only have one kinematic distribution to determine the inverse matrix $\overline{G}$,

$$\begin{aligned} f_{\text{parton}} &= n \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \Rightarrow \quad f_{\text{reco}} = f_{\text{parton}}\ , \\ f_{\text{parton}} &= \begin{pmatrix} 1 \\ n \\ 0 \end{pmatrix} \quad \Rightarrow \quad f_{\text{reco}} = f_{\text{parton}} + \epsilon \begin{pmatrix} n-1 \\ -2n+1 \\ n \end{pmatrix}\ . \end{aligned} \tag{6.4}$$

The first example shows how for a symmetric smearing matrix a flat distribution removes all information about the detector effects. This implies that we might end up with a choice of reference process and phase space such that we cannot extract the detector effects from the available data. The second example illustrates that for bin migration from a dominant peak the information from the original f_{parton} gets overwhelmed easily. We can also compute the inverse of the smearing matrix in (6.3)

$$\overline{G} \approx \frac{1}{1-4\epsilon} \begin{pmatrix} 1-3\epsilon & -\epsilon & \epsilon^2 \\ -\epsilon & 1-2\epsilon & -\epsilon \\ \epsilon^2 & -\epsilon & 1-3\epsilon \end{pmatrix}\ , \tag{6.5}$$

where we neglect ϵ^2 relative to the linear terms. The unfolding matrix extends beyond the nearest neighbor bins, which means that local detector effects lead to a global unfolding matrix and unfolding only works well if we understand our entire dataset. The reliance on useful kinematic distributions and the global dependence of the unfolding define the main challenges once we attempt to unfold the full phase space of an LHC process.

Current unfolding methods follow essentially this simple approach and face three challenges. First, the required binning into histograms is chosen before the analysis and ad-hoc, so it is not optimal. Second, the matrix structure behind correlated observables implies that we can only unfold two or three observables simultaneously. Finally, detector unfolding often misses some features which affect the detector response.

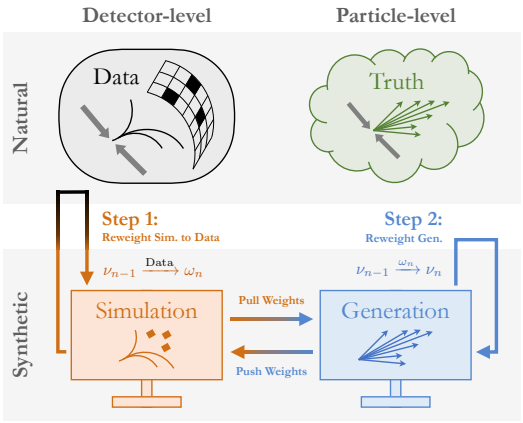


Figure 48: Illustration of the Omnifold method for detector unfolding. The iterations defined in the plot are slightly different from (6.10), where ν_∞ can be defined relative to ν_0 or correspond to the product of many reweightings. Figure from Ref. [38].

6.1 Inversion by reweighting

OmniFold is an ML-based techniques which iteratively unfolds detector effects. It works on high-dimensional phase spaces and does not require histograms or binning. Its reweighting technique is illustrated in Figure 48. The goal is to start from reconstruction-level data and predict the parton-level configuration for a measured configuration. It avoids a direct mapping from reconstruction-level events to parton level events and instead employs an iterative reweighting based on simulated pairs of parton-level and reconstruction-level configurations,

$$(x_\text{parton}, x_\text{reco}) \; . \tag{6.6}$$

These simulated events need to be reweighted at the reconstruction level, to reproduce the measured data. Because of the pairing, this reweighting can be pushed to the parton level.

As a starting point, we assume that the simulated events have finite weights, while the data starts with unit weights,

$$\nu_0(x_\text{parton}) = \nu_0(x_\text{reco})\Big|_\text{model} \qquad \text{and} \qquad \omega_0(x_\text{reco})\Big|_\text{data} = 1 \; . \tag{6.7}$$

The Omnifold algorithm consists of four steps:

1. At reconstruction level it reweights the simulated events to match the measured events. For x_reco the weight ν_0 is corrected to ω_1, such that the original density $p_\text{model}(x_\text{reco}, \nu_0)$ becomes the target density $p_\text{data}(x_\text{reco}, 1)$,

$$\frac{\omega_1(x_\text{reco})}{\nu_0(x_\text{reco})} = \frac{p_\text{data}(x_\text{reco}, 1)}{p_\text{model}(x_\text{reco}, \nu_0)} \; . \tag{6.8}$$

   We include the weights as an explicit argument of the likelihoods.
2. The paired simulation transfers (pulls) this weight to the corresponding parton-level event,

$$\omega_1(x_\text{reco}) \;\to\; \omega_1(x_\text{parton}) \; . \tag{6.9}$$

3. At the parton level we know our target phase space distribution. This introduces a weight ν_1 which replaces ν_0,

$$\frac{\nu_1(x_\text{parton})}{\nu_0(x_\text{parton})} = \frac{p_\text{model}(x_\text{parton}, \omega_1)}{p_\text{model}(x_\text{parton}, \nu_0)} \; . \tag{6.10}$$

4. This new weight we transfer back (push) to the reconstruction level,

$$\nu_1(x_\text{parton}) \;\to\; \nu_1(x_\text{reco}) \; . \tag{6.11}$$

   We then go back to (6.8) and re-iterate the two steps until they have converged.

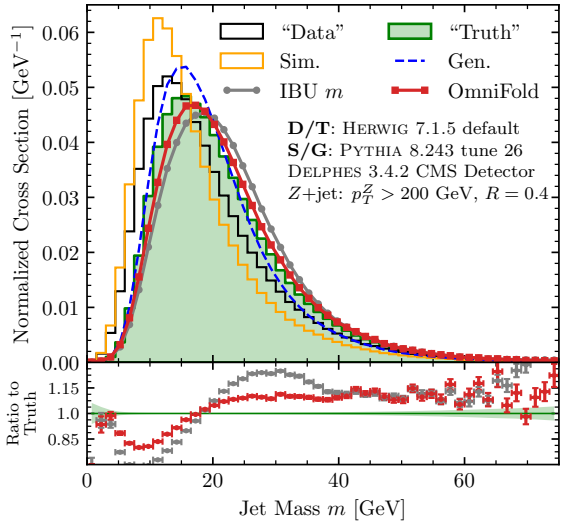


Figure 49: Unfolding results for sample substructure observables, using Herwig jets as data and using Pythia for the simulation. The lower panels include statistical uncertainties on the 1-dimensional distributions. Figure from Ref. [38].

Once the Omnifold algorithm converges, the phase space distribution at the parton level is given by

$$p_{\text{unfold}}(x_{\text{parton}}) = \nu_\infty(x_{\text{parton}})\, p_{\text{model}}(x_{\text{parton}}) \; . \tag{6.12}$$

Because the iteration steps do not require additional simulations and work on fixed, paired datasets, this method is fast and efficient. The ultimate question is how well it describes correlations between phase space points.

To illustrate the performance of Omnifold, we look at jets and represent the actual data, $p_{\text{data}}(x_{\text{reco}})$, with the Herwig event generator. For the simulation of paired jets, $p_{\text{model}}(x_{\text{parton}})$ and $p_{\text{model}}(x_{\text{reco}})$, we use the Pythia event generator. We then use Omnifold can unfold the Herwig data to the parton level, and compare these results with the true Herwig distributions at the parton level

$$p_{\text{unfold}}(x_{\text{parton}}) \;\leftrightarrow\; p_{\text{data}}(x_{\text{parton}}) \; . \tag{6.13}$$

While the event information can be fed into the network in many ways, the Omnifold authors choose to use the deep sets representation of point clouds as introduced in Sec. 3.3.4. For a quantitative test we show the jet mass. The detector effects, described by the fast Delphes simulation can be seen in the difference between the generation curves, $p_{\text{model}}(x_{\text{parton}})$, and the simulation curves, $p_{\text{model}}(x_{\text{reco}})$. We see that the jet mass is significantly reduced by the detector resolution and threshold. The data distributions from Herwig represent $p_{\text{data}}(x_{\text{reco}})$, and we observe a significant difference to the simulation, where Herwig jets have more constituents than Pythia jets.

If we now use the Pythia-trained Omnifold algorithm to unfold the mock data we can compare the results to the green truth curves. The ratio $p_{\text{unfold}}(x_{\text{parton}})/p_{\text{data}}(x_{\text{parton}})$ is shown in the lower panels. We see that, as any network, this deviation is systematic in the bulk and becomes noisy in the kinematic tails. The same is true for the classical, bin-wise iterative Bayesian unfolding (IBU), and the unbinned and multi-dimensional Omnifold method clearly beats the bin-wise method.

6.2 Conditional generative networks

Instead of constructing a backwards mapping for instance from the detector level to the parton level using classifier reweighting, we can also tackle this inverse problem with generative networks, specifically a conditional normalizing flow or cINN. In this section we will describe two applications of this versatile network architecture, one for unfolding and one for inference or measuring model parameters.

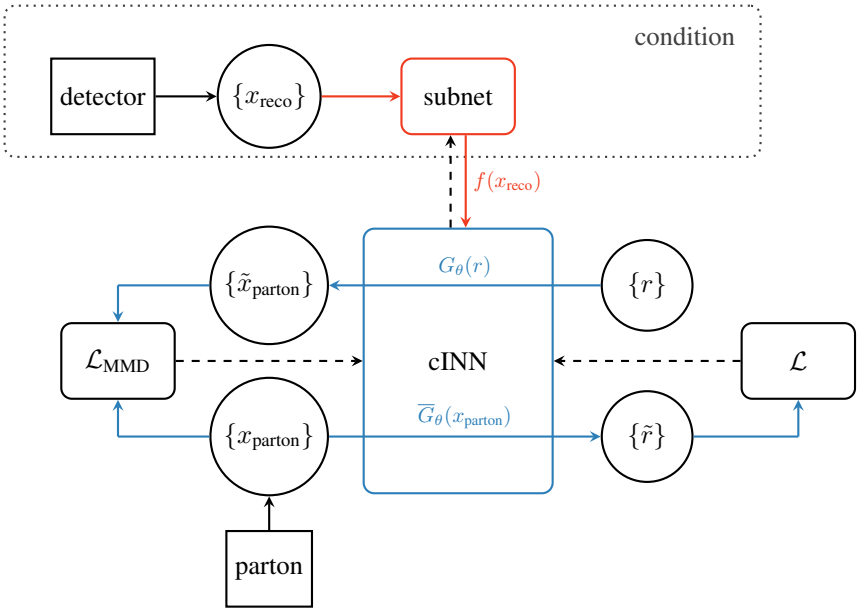


Figure 50: Illustration of the cINN setup. Figure from Ref. [39].

6.2.1 cINN unfolding

The idea of using conditional generative networks for unfolding is to generalize the binned distributions (6.1) to a continuous description of the entire respective phase spaces,

$$\frac{d\sigma}{dx_{\text{reco}}} = \int dx_{\text{parton}}\; G(x_{\text{reco}}, x_{\text{parton}})\; \frac{d\sigma}{dx_{\text{parton}}} \; . \tag{6.14}$$

all in complete analogy to the binned form above. The symmetric form of (6.14) indicates that G and $\overline{G}$ are both defined as distributions for suitable test functions.

Both directions, the forward simulation and the backward unfolding, are stochastic in nature. Based on the statistical picture we should switch from joint probabilities to conditional probabilities using Bayes' theorem. The forward detector simulation is then described by $G(x_{\text{reco}}|x_{\text{parton}})$, whereas $\overline{G}(x_{\text{parton}}|x_{\text{reco}})$ gives the probability of the model x_{parton} being true given the observation x_{reco}. We have to employ Bayes' theorem to relate the two directions

$$\overline{G}(x_{\text{parton}}|x_{\text{reco}}) = \frac{G(x_{\text{reco}}|x_{\text{parton}})\; p(x_{\text{parton}})}{p(x_{\text{reco}})} \; , \tag{6.15}$$

where G and $\overline{G}$ now denote conditional probabilities. For the inverse direction we can consider $p(x_{\text{reco}})$ the evidence and ignore it as part of the normalization, but $p(x_{\text{parton}})$ remains as a model dependence for instance through the choice training data. We will discuss this at the end of this section.

For the generative task we employ an INN to map an set of random numbers to a parton-level phase space with the corresponding dimensionality. To capture the information from the reconstruction-level events we condition the INN on such an event. Trained on a given process the network will now generate probability distributions for parton-level configurations given a reconstruction-level event and an unfolding model. The cINN is still invertible in the sense that it includes a bi-directional training from Gaussian random numbers to parton-level events and back, but the invertible nature is not what we use to invert the detector simulation. We will eventually need show how the conditional INN retains a proper statistical notion of the inversion to parton level phase space. This avoids a major weakness of standard unfolding methods, namely that they only work on large enough event samples condensed to one-dimensional or two-dimensional kinematic distributions, such as a missing transverse energy distribution in mono-jet searches or the rapidities and transverse momenta in top pair production.

The structure of the conditional INN (cINN) is illustrated in Figure 50. We first preprocess the reconstruction-level data by a small subnet, $x_{\text{reco}} \to f(x_{\text{reco}})$ and omit this additional step below. After this preprocessing, the detector information is

passed to the functions s_i and t_i in (5.44), which now depend on the input, the output, and on the fixed condition. The cINN is trained a loss similar to (5.46), but now maximizing the probability distribution for the network weights θ, conditional on x_parton and x_reco always sampled in pairs

$$\boxed{\mathcal{L}_\text{cINN} = -\Big\langle \log p_\text{model}(\theta|x_\text{parton}, x_\text{reco}) \Big\rangle_{p_\text{parton}\sim p_\text{reco}}} \; . \tag{6.16}$$

Next, we need to turn the posterior for the network parameters into a likelihood for the network parameters and evaluate the probability distribution for the event configurations. Because we sample the parton-level and reconstruction-level events as pairs, it does not matter which of the two we consider for the probability. We use Bayes' theorem to turn the probability for θ into a likelihood, with x_parton as the argument,

$$\begin{aligned}
\mathcal{L}_\text{cINN} &= -\left\langle \log \frac{p_\text{model}(x_\text{parton}|x_\text{reco},\theta)\, p_\text{model}(\theta|x_\text{reco})}{p_\text{model}(x_\text{parton}|x_\text{reco})} \right\rangle_{p_\text{parton}\sim p_\text{reco}} \\
&= -\Big\langle \log p_\text{model}(x_\text{parton}|x_\text{reco},\theta) \Big\rangle_{p_\text{parton}\sim p_\text{reco}} - \log p_\text{model}(\theta) + \text{const} \; .
\end{aligned} \tag{6.17}$$

As before, we ignore all terms irrelevant for the minimization. The second term is a simple weight regularization, which we also drop in the following. We now apply the usual coordinate transformation, defined in Figure 50, to introduce the trainable Jacobian,

$$\begin{aligned}
\mathcal{L}_\text{cINN} &= -\Big\langle \log p_\text{model}(x_\text{parton}|x_\text{reco},\theta) \Big\rangle_{p_\text{parton}\sim p_\text{reco}} \\
&= -\left\langle \log p_\text{latent}(\overline{G}_\theta(x_\text{parton}|x_\text{reco})) + \log \left| \frac{\partial \overline{G}_\theta(x_\text{parton}|x_\text{reco})}{\partial x_\text{parton}} \right| \right\rangle_{p_\text{parton}\sim p_\text{reco}} \; .
\end{aligned} \tag{6.18}$$

This is the usual maximum likelihood loss, but conditional on reconstruction-level events and trained on event pairs. As before, we assume that we want to map the parton-level phase space to Gaussian random numbers. In that case the first term becomes

$$\log p_\text{latent}(\overline{G}_\theta(x_\text{parton})) = -\frac{||\overline{G}_\theta(x_\text{parton}))||_2^2}{2} \; . \tag{6.19}$$

We can briefly discuss the symmetry of the problem. If we think of the forward simulation as generating a set of reconstruction-level events for a given detector-level event and using Gaussian random numbers for the sampling, the forward simulation and the unfolding are completely equivalent. The reason for this symmetry is that in the entire argument we never consider individual events, but phase space densities, so our inversion is stochastic and not deterministic.

To see what we can achieve with cINN unfolding we use the hard reference process

$$q\bar{q} \to ZW^\pm \to (\ell^-\ell^+)\,(jj) \; . \tag{6.20}$$

We could use this partonically defined process to test the cINN detector unfolding. However, once we include incoming protons we also need to include QCD jet radiation, so our actual final state is given by the hadronic process

$$pp \to (\ell^-\ell^+)\,(jj) + \{0,1,2\}\,\text{jets} \; . \tag{6.21}$$

All jets are required to pass the basic acceptance cuts

$$p_{T,j} > 25\,\text{GeV} \qquad \text{and} \qquad |\eta_j| < 2.5 \; . \tag{6.22}$$

These cuts regularize the soft and collinear divergences at fixed-order perturbation theory.

The second task of our cINN unfolding will be to determine which of the final-state jets come from the W-decay and which arise from initial-state QCD radiation. Since ISR can lead to harder jets than the W-decay jets, an assignment by $p_{T,j}$ will not solve the jet combinatorics. This unfolding requires us to define a specific hard process with a given number of external jets. We can illustrate this choice using two examples. First, a di-lepton resonance search typically probes the hard process $pp \to \mu^+\mu^- + X$, where X denotes any number of additional, analysis-irrelevant jets. We would invert this measurements

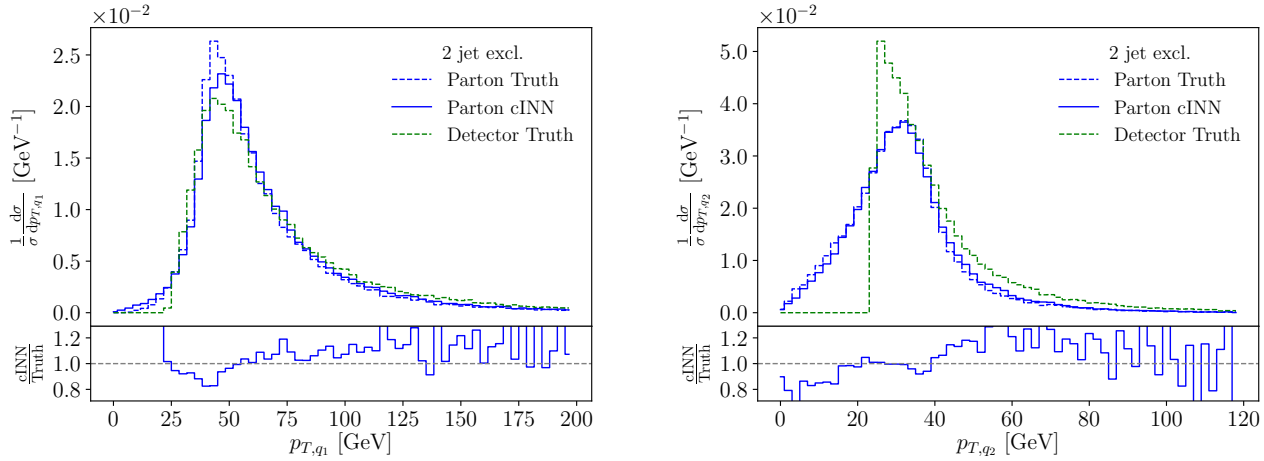


Figure 51: cINNed $p_{T,q}$ distributions. Training and testing events include exactly two jets. The lower panels give the ratio of cINNed to parton-level truth. Figure from Ref. [39].

to the partonic process $pp \to \mu^+\mu^-$. A similar mono-jet analysis instead probes the process $pp \to Z'j(j) + X$, where Z' is a dark matter mediator decaying to two invisible dark matter candidate. Depending on the analysis, the relevant background process to invert is $pp \to Z_{\nu\nu}j$ or $pp \to Z'jj$, where the missing transverse momentum recoils against one or two hard jets. Because our inversion network in trained on matched pairs of simulated events, we implicitly define the appropriate hard process when generating the training data.

In Figure 51 we show the unfolding performance of the cINN, trained and tested on exclusive 2-jet events. The two jets are mapped on the two hard quarks from the W-decay, ordered by p_T. In the left panel we see that the detector effects on the harder decay jets are mild and the unfolding is not particularly challenging. On the right we the effect of the minimum-p_T cut and how the network is able to remove this effect when unfolding to the decay quarks. The crucial new feature of this cINN output is that it provides probability distribution in parton-level phase space for a given reconstruction-level event. By definition of the loss function in (6.17) we can feed a single reconstruction-level event into the network and obtain a probability distribution over parton-level phase space for this single event. Obviously, this guarantees that any kinematic distribution and any correlation between events is unfolded correctly at the sample level.

Next, we can demonstrate how the cINN unfolds a sample of events with a variable number of jets from the hard process and from ISR. First, we show the unfolding results for events with three jets in the upper panels of Figure 52. For three jets in the final state, the combination of detector effects and ISR has a visible effect on the kinematics of the leading quark. This softening is correctly reversed by the unfolding. For the sub-leading quark the problem and the unfolding performance is similar to the exclusive 2-jet case. The situation becomes more interesting when we consider samples with two to four jets all combined. Now the network has to flexibly resolve the combinatorial problem to extract the two W-decay jets from a mixed training sample. In Figure 52 we show a set of unfolded distributions from a variable jet number. Without showing them, it is clear that the $p_{T,j}$-threshold at the detector level is corrected, and the cINN allow for both $p_{T,q}$ values to zero. Next, we see that the comparably flat azimuthal angle difference at the parton level is reproduced to better than 10% over the entire range. Finally, the m_{jj} distribution with a additional MMD loss re-generates the W-mass peak at the parton level almost perfectly. The precision of this unfolding is not any worse than it is for an INN generator. This means that conditional generative networks unfold detector effects and jet radiation for LHC processes very well, even through their network architectures are more complex than the classifiers used in Sec. 6.1.

The problem with unfolding data based on simulated, paired events is that the simulation is based on a model. For example, events describing detector effects are simulated for a given hard process. Formally, this defines the prior $p(x_{\text{parton}})$ in (6.15), which we start from when we simulate the detector in the forward direction. From a physics perspective, it makes sense to assume that detector simulations are at least approximately independent of the hard process, so the inverse simulation should be well defined at the statistical level. If this model dependence is sizeable, we can employ an iterative method to reduce the bias caused by difference between the data we want to unfold and the simulation we use to train the unfolding network. In standard methods it is referred to Bayesian iterative unfolding, and it is simular to the reweighting strategy described in Sec. 6.1.

The iterative IcINN algorithm is illustrated in Fig. 53 and includes three steps:

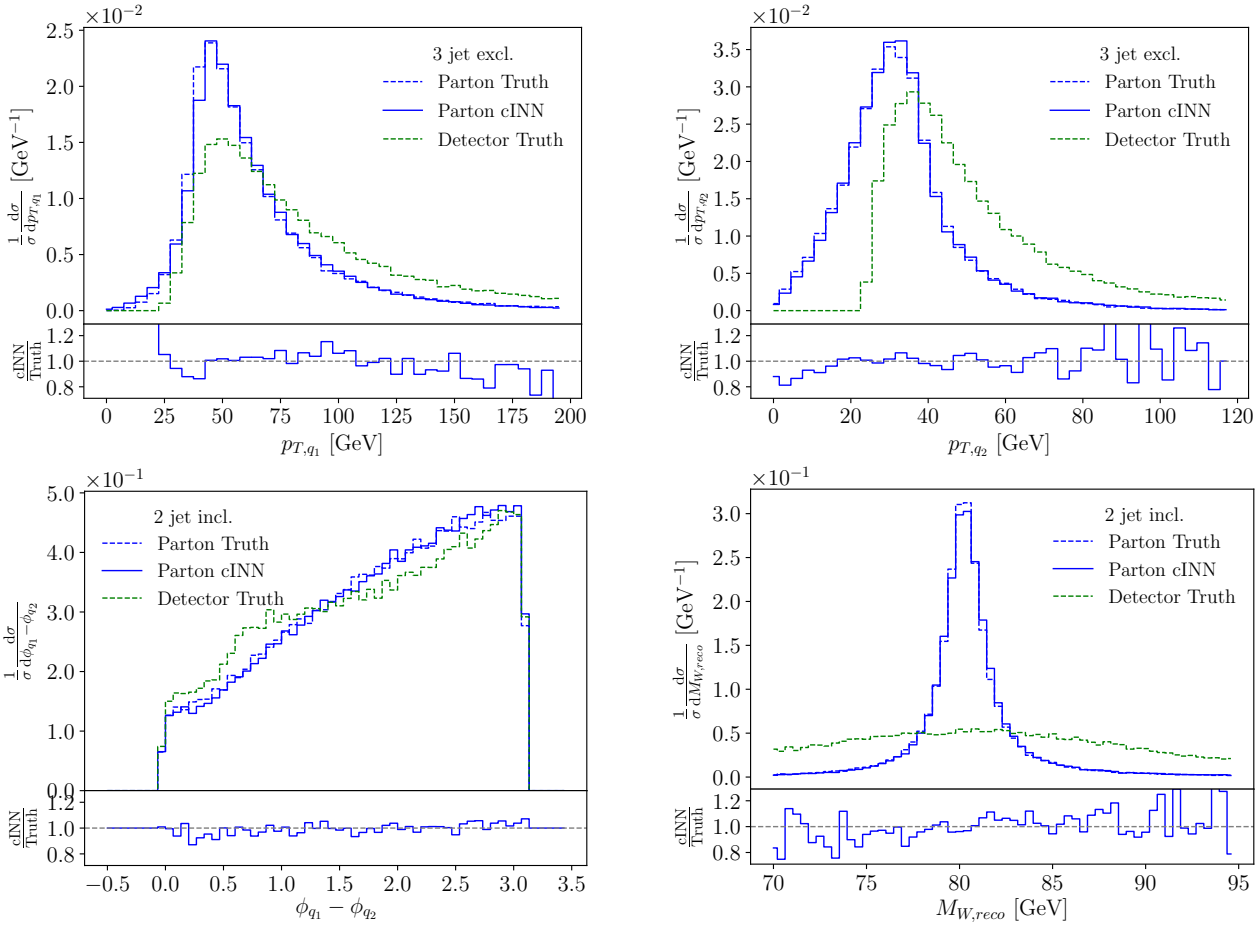


Figure 52: cINNed $p_{T,q}$ and $m_{W,\text{reco}}$ distributions. Training and testing events include exactly three (left) and four (right) jets from the data set including ISR. Figure from Ref. [39].

1. We start with simulated pairs of events before and after detector, train the cINN on these paired events (step 1), and unfold the measured data (step 2). This predicts a phase space distribution encoded in unfolded events over x_parton.
2. Next we train a classifier to learn the ratio between the unfolded measured events and the training data as a function of x_parton. As defined in (5.75) we can use it to reweight the simulated pairs in x_parton to match the measured events (step 3).
3. Because the training data is paired events, we can transfer these weights to x_reco and use these weighted events as new training data for the unfolding cINN (step 1). The training-unfolding-reweighting steps can be repeated until the algorithm converges and the classifier returns a global value of 0.5.

Technically, it turns out to be more efficient of the cINN is not trained from scratch each time. To re-train the cINN on the weighted paired events the loss function in (6.16) is supplemented with learned weights

$$\mathcal{L}_\text{cINN} = -\Big\langle w(x_\text{reco}) \log p_\text{model}(\theta|x_\text{parton}, x_\text{reco}) \Big\rangle_{p_\text{parton}\sim p_\text{reco}} . \tag{6.23}$$

This iterative approach ensures that the unfolding network is trained on events similar to the data we actually unfold, so there is no bias from a difference between training data and measured data. This removes a major systematic uncertainty from unfolded experimental results and it makes it easier to generalize the unfolding network from one hard process to another.

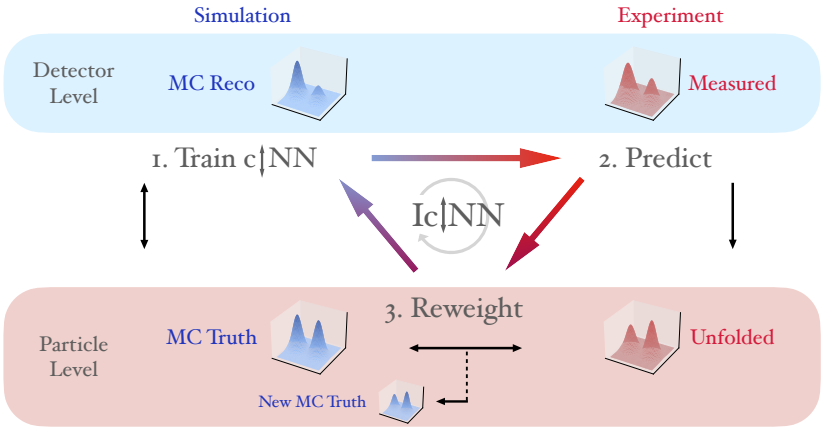


Figure 53: Illustration of the iterative cINN unfolding. Figure from Ref. [40].

7 Physics of Machine Learning

By now we have introduced and applied a large collection of different architectures. While by no means complete, this covers, in terms of underlying structures, a large part of the phase space of existing architectures.

Our approach was very hands-on and is also very common in the field: In view of a given application we use, further develop or devise an architecture that is adapted to the task at hand. The loss function is chosen 'naturally' and modified accordingly to the underlying optimization tasks.

In this last Chapter of the lecture course we shall dive into the physics interpretation of neural networks which also allows us to address the question of 'optimal' loss functions. This Chapter extends the statistical/lattice physics analoga introduced and used in in Sections 3.1 and 5.3.2. There, however, we have used NNs as statistical systems to either generate configurations (events) in a statistical system, here we exploit the interpretation of an NN as a (complex) statistical system to understand and improve its performance for generic ML tasks.

The relation of NNs to statistical systems has been advertised in Sections 3.1 and 5.3.2 at the example of the layerwise propagation of information, reminiscent of the time evolution of a stochastic system in the presence of white or other noise, see 3.2.2. Let us extend the analogy further and formulate the propagation in a network with

$$\partial_\tau \phi = \frac{\partial S(\phi, \theta)}{\partial \phi_i} + \xi_i \,, \tag{7.1}$$

where ϕ is the variable that propagates through the network, ξ_i is an uncorrelated, typically white, noise; S is the underlying Hamiltonian of the statistical field theory and θ comprises the network parameters. Finally, τ is the (discrete) time that describes the layerwise propagation.

In (7.1) we have dropped the activation functions for the sake of simplicity. Clearly, this suggests the interpretation of a given NN as a statistical field theory.

In the lecture course we have studied networks with an action S, that is at most quadratic in the field, leading to an affine transformation of the variable. Non-linearities are introduced by the activation function and the network parameters. The latter carry the information of the probability distribution in the test data set, and the set-up begs the question, which statistical field theory is underlying the combination test data set and ML-task. As mentioned already above, in the following we will investigate this question and the related one of optimising the loss function.

7.1 Bayesian network as a statistical field theory

A nice illustrative example is provided in [41], where the ML-task is given identifying handwritten numbers within a binary classification task, see Figure 54. This task (and similar ones) are evaluated in the limit of *infinitely wide* networks.

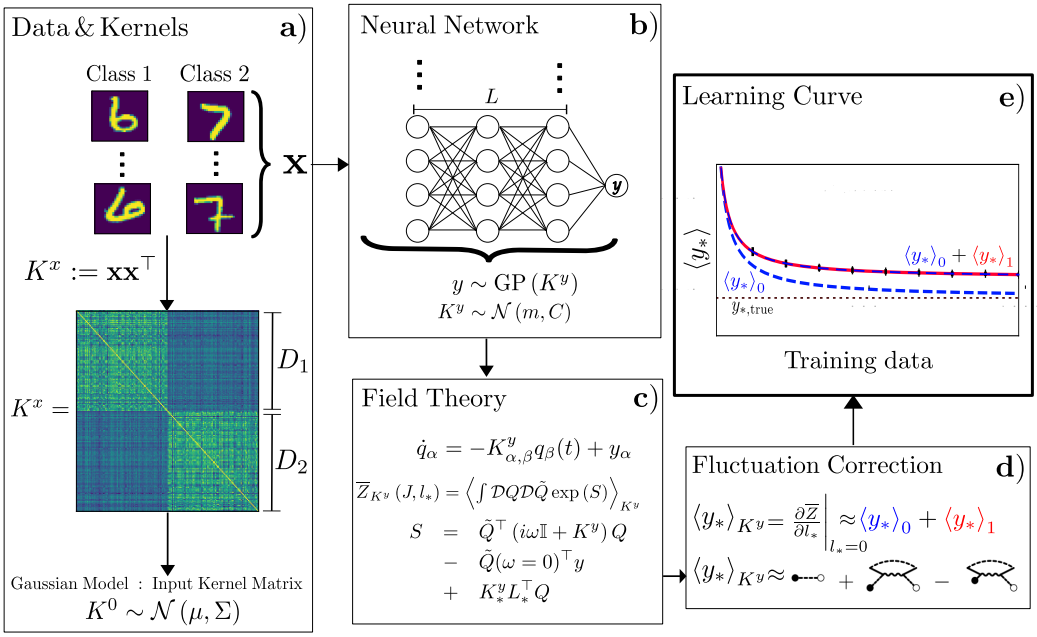


Figure 54: **a)** A binary classification task, such as distinguishing pairs of digits in MNIST, can be described with the help of an overlap matrix K^x that represents similarity across the $c = c_1 + c_2$ images of the training set of two classes, 1 and 2 with D_1 and D_2 samples respectively. Entries of the overlap matrix are heterogeneous. Different drawings of c example patterns each lead to different realizations of the overlap matrix; the matrix is stochastic. We here describe the matrix elements by a correlated multivariate Gaussian.
b) The data is fed through a feed-forward neural network to produce an output y. In the case of infinitely wide hidden layers and under Gaussian priors on the network weights, the output of the network is a Gaussian process with the kernel K^y, which depends on the network architecture and the input kernel K^x.
c) To obtain statistical properties of the posterior distribution, we compute its disorder-averaged moment generating function $\overline{Z}(J, l_*)$ diagrammatically.
d) The leading-order contribution from the homogeneous kernel $\langle y^* \rangle_0$ is corrected by $\langle y^* \rangle_1$ due to the variability of the overlaps; both follow as derivatives of $\overline{Z}(J, l_*)$.
e) Comparing the mean network output on a test point $\langle y^* \rangle$, the zeroth order theory $\langle y_* \rangle_0$ (blue dashed), the first-order approximation in the data-variability $\langle y_* \rangle_{0+1}$ (blue-red dashed) and empirical results (black crosses) as a function of the amount of training data (learning curve) shows how variability in the data set limits the network performance and validates the theory. Figure taken from [41].

In this limit we can draw from central limit theorems which facilitates the considerations. In view of the interpretation as a complex statistical system (or lattice field theory) this limit translates into the thermodynamic limit or infinite volume limit/many body limit.

Specifically, it is shown that in the limit of the Bayesian network used for the classification task can be interpreted as a non-trivial statistical field theory with an action similar to the lattice action (3.11) but with a self interaction term

$$\phi^3 + \phi^4 \,. \tag{7.2}$$

Equation (7.2) describes the non-Gaußian part of the statistical field theory, and can be tested by the higher order moments or cumulants of the system at hand.

7.1.1 Bayesian set-up and Gaußian processes

In order to allow for a simple access to the literature, and in particular [41], we recall some basic properties of Bayesian networks as well as providing a short introduction to Gaußian processes; both within the notation in [41]: The input and output samples, x_α and y_α respectively, of the feed forward neural network are taken from

$$\text{input:} \quad x_\alpha \in \mathbb{R}^{N_{\text{dim}}} \,, \qquad \text{and} \qquad \text{output:} \quad y_\alpha \in \mathbb{R}^{N_{\text{out}}} \,, \tag{7.3}$$

In a slight variation of (1.50) in Section 1.4 we define the network transformations of the feed forward network as

$$h_\alpha^{(l)} = \mathbf{W}^{(l)}\phi^{(l)}\left(h_\alpha^{(l-1)}\right) \quad \text{with} \quad h_\alpha^0 = \mathbf{V}x_\alpha\,, \qquad y_\alpha = \mathbf{U}\phi^{(L+1)}\left(h_\alpha^{(L)}\right)\,, \tag{7.4}$$

where the ϕ's are the activation functions, and

$$\text{read-in weights: } \mathbf{V}\in\mathbb{R}^{N_h\times N_{\text{dim}}}\,, \qquad \text{hidden weights: } \mathbf{W}^{(l)}\in\mathbb{R}^{N_h\times N_h}\,, \qquad \text{read-out weights: } \mathbf{U}\in\mathbb{R}^{N_{\text{out}}\times N_h}\,, \tag{7.5}$$

with the number of the hidden neurons N_h. The layer index l obeys $1\le l\le L$ with the number of layers L. We also assume layer- independent activation functions $\phi^{(l)}=\phi$. The collection of all weights are the model parameters Θ with

$$\Theta = \{\mathbf{V},\mathbf{W}^{(1)},\ldots,\mathbf{W}^{(L)},\mathbf{U}\}\,. \tag{7.6}$$

The choice of an appropriate loss function

$$\mathcal{L}(\Theta,x_\alpha,y_\alpha)\,, \tag{7.7}$$

completes the NN set up. The total data set $\mathcal{D}$ is divided into a training set $\mathcal{D}_{\text{tr}}$ and test set $\mathcal{D}_{\text{test}}$ with

$$\mathcal{D}_{\text{tr}}:\ D_{\text{tr}}=|\mathcal{D}_{\text{tr}}|\,, \qquad \mathcal{D}_{\text{test}}:\ D_{\text{test}}=|\mathcal{D}_{\text{test}}|\,. \tag{7.8}$$

Finally, we add a Gaußian (white) noise term to the process (7.4),

$$y\,(x_\alpha|\Theta) \longrightarrow y\,(x_\alpha|\Theta)+\xi_\alpha\,, \tag{7.9}$$

where the ξ_α are Gaußian independently and identically distributed (iid) with variance σ^2_{reg}. For a detailed analysis of the underlying stochastic processes see e.g. [42]. The regularization with (7.9) leads us to the probability distribution

$$p\,(y|x_\alpha,\Theta) = \langle\delta\,[y_\alpha - y(x_{\text{tr},\alpha}|\Theta)-\xi_\alpha]\rangle_{\xi_\alpha} = \mathcal{N}\left(y_\alpha;\,y(x_\alpha|\Theta),\sigma^2_{\text{reg}}\right)\,, \tag{7.10}$$

To complete the Bayesian set up we use prior, Gaußian, distributions for the weights, to wit,

$$V_{ij}\sim\mathcal{N}\left(0,\sigma_v^2/N_{\text{dim}}\right)\,, \qquad W_{ij}^{(l)}\sim\mathcal{N}\left(0,\sigma_w^2/N_h\right)\,, \qquad U_{ij}\sim\mathcal{N}\left(\sigma_u^2/N_h\right)\,. \tag{7.11}$$

The posterior distribution then follows with Bayes' theorem (1.7),

$$p\,(\Theta|\mathbf{x}_{\text{tr}},\mathbf{y}_{\text{tr}}) = \frac{p\,(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}},\Theta)\;p\,(\Theta)}{p\,(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}})}\,, \tag{7.12}$$

relating it to the likelihood $p\,(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}},\Theta)$ and the model evidence $p\,(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}})$.

Equation (7.12) also allows us to compute the network output y_*, that belongs to an input x_* (in the test sample), $p\,(y_*|x_*,\mathbf{x}_{\text{tr}},\mathbf{y}_{\text{tr}})$. It is obtained by integrating the likelihood over possible parameter sets θ, weighted by the posterior distribution. We obtain

$$p\,(y_*|x_*,\mathbf{x}_{\text{tr}},\mathbf{y}_{\text{tr}}) = \int d\Theta\,p\,(y_*|x_*,\Theta)\,p\,(\Theta|\mathbf{x}_{\text{tr}},\mathbf{y}_{\text{tr}}) = \frac{p\,(y_*,\mathbf{y}_{\text{tr}}|x_*,\mathbf{x}_{\text{tr}})}{p\,(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}})}\,. \tag{7.13}$$

Equation (7.13) represents the learned distribution in terms of the model distribution and the distribution of the joint network outputs for all training points and the test point. This presentation allows us to discuss the underlying statistical theory in the limit of wide networks.

Accordindly we take the limit of wide networks with $N_h\to\infty$. Then the above process approaches a Gaußian process $y\sim\mathcal{N}\,(0,K^y)$, where the covariance $\langle y_\alpha y_\beta\rangle = K^y_{\alpha\beta}$ is also denoted as the kernel. This is beneficial, as the inference for the network output y_* for a test point x_* then also follows a Gaussian distribution with mean and covariance given by

$$\langle y_*\rangle = K^y_{*\alpha}\,(K^y)^{-1}_{\alpha\beta}\;y_{\text{tr},\beta}\,, \qquad \left\langle(y_*-\langle y_*\rangle)^2\right\rangle = K^y_{**}-K^y_{*\alpha}\,(K^y)^{-1}_{\alpha\beta}\;K^y_{\beta *}\,, \tag{7.14}$$

where summation over repeated indices is implied.

This completes our setup which is now used to relate the variability of the data set or rather its training part on the thermodynamic limit or limit of wide networks: what statistical field theory underlies the neural network.

The variability of the data set has to aspects:

(i) For each drawing of D pairs of training samples $(x_{\text{tr},\alpha},y_{\text{tr},\alpha})_{1\le\alpha\le D}$ one obtains a $D\times D$ kernel matrix K^y with heterogeneous entries. This entails that in a single instance of Bayesian inference, the entries of the kernel matrix vary from one entry to the next.

(ii) Each drawing of D training data points and one test data point (x_*,y_*) leads to a different kernel $\{K^y_{\alpha\beta}\}_{1\le\alpha,\beta\le D+1}$, which follows some probabilistic law $K^y\sim p(K^y)$.

7.1.2 Data sets

The training and data sets for this classification task can be taken from the (E)MNIST database ((*E*xtended) *M*odified *N*ational *I*nstitute of *S*tandards and *T*echnology database). This clear-cut standard example is accompanied by an even simpler one with a synthetic binary classification task.

The data set in this case consists of pattern realizations $x_\alpha \in \{-1,1\}^{N_{\text{dim}}}$ with dimension N_{dim} even. We denote the entries $x_{\alpha i}$ of this N_{dim}-dimensional vector for data point α as pixels that randomly take either of two values $x_{\alpha i} \in \{-1,1\}$ with respective probabilities $p(x_{\alpha i}=1)$ and $p(x_{\alpha i}=-1)$ that depend on the class $c(\alpha) \in \{1,2\}$ of the pattern realization and whether the index i is in the left half ($i \leq N_{\text{dim}}/2$) or the right half ($i > N_{\text{dim}}/2$) of the pattern: For class $c(\alpha)=1$ each pixel $x_{\alpha i}$ with $1 \leq i \leq N_{\text{dim}}$ is realized independently as a binary variable as ($c(\alpha)=1$)

$$x_{\alpha i} = \begin{cases} 1 & \text{with } p \\ -1 & \text{with } (1-p) \end{cases} \quad \text{for } i \leq \frac{N_{\text{dim}}}{2}\,, \qquad \text{and} \qquad x_{\alpha i} = \begin{cases} 1 & \text{with } (1-p) \\ -1 & \text{with } p \end{cases} \quad \text{for } i > \frac{N_{\text{dim}}}{2}\,. \tag{7.15}$$

For a pattern x_α in the second class $c(\alpha)=2$ the pixel values are distributed independently of those in the first class with a statistics that equals the negative pixel values of the first class, which is

$$P\left(x_{\alpha i}\right) = P\left(-x_{\beta i}\right)\,, \qquad \text{with} \qquad c(\beta)=1\,, \quad \text{and} \quad c(\alpha)=2\,. \tag{7.16}$$

There are two limiting cases for p which illustrate the construction of the patterns: In the limit $p=1$, each pattern x_α in $c=1$ consists of a vector, where the first $N_{\text{dim}}/2$ pixels have the value $x_{\alpha i}=1$, whereas the second half consists of pixels with the value $x_{\alpha,i}=-1$. The opposite holds for patterns in the second class $c=2$. This limiting case is shown in Figure 55 (right column). In the limit case $p=0.5$ each pixel assumes the value $x_{\alpha,i}=\pm 1$ with equal probability, regardless of the pattern class-membership or the pixel position. Hence one cannot distinguish the class membership of any of the training instances. This limiting case is shown in Figure 55 (left column). If $c(\alpha)=1$ we set $y_{\text{tr},\alpha}=-1$ and for $c(\alpha)=2$ we set $y_{\text{tr},\alpha}=1$.

We now investigate the description of this task in the framework of Bayesian inference. The hidden variables h^0_α (7.4) in the input layer under a Gaussian prior on $V_{ij} \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \sigma_v^2/N_{\text{dim}}\right)$ follow a Gaussian process with kernel $K^{(0)}$ given by

$$K^0_{\alpha\beta} = \langle h^0_\alpha h^0_\beta \rangle_{V \sim \mathcal{N}\left(0, \frac{\sigma_v^2}{N_{\text{dim}}}\right)} = \frac{\sigma_v^2}{N_{\text{dim}}} \sum_{i=1}^{N_{\text{dim}}} x_{\alpha i} x_{\beta i}\,. \tag{7.17}$$

Separability of the two classes is reflected in the structure of this input kernel K^0 as shown in Figure 55: In the cases with $p=0.8$ and $p=1$ one can clearly distinguish blocks; the diagonal blocks represent intra-class overlaps, the off-diagonal blocks inter-class overlaps. This is not the case for $p=0.5$, where no clear block-structure is visible. In the case of $p=0.8$ one can further observe that the blocks are not as clear-cut as in the case $p=1$, but rather noisy, similar to $p=0.5$. This is due to the probabilistic realization of patterns, which induces stochasticity in the blocks of the input kernel K^0 (7.17). To quantify this effect, based on the distribution of the pixel values (7.15) we compute the distribution of the entries of K^0 for the binary classification task. The mean of the overlap elements $\mu_{\alpha\beta}$ and their covariances $\Sigma_{(\alpha\beta)(\gamma\delta)}$ are defined via

$$\mu_{\alpha\beta} = \left\langle K^0_{\alpha\beta} \right\rangle\,, \qquad \Sigma_{(\alpha\beta)(\gamma\delta)} = \left\langle \delta K^0_{\alpha\beta}\, \delta K^0_{\gamma\delta} \right\rangle\,, \qquad \delta K^0_{\alpha\beta} = K^0_{\alpha\beta} - \mu_{\alpha\beta}\,, \tag{7.18}$$

where the expectation value $\langle\cdot\rangle$ is taken over drawings of D training samples each. By construction we have $\mu_{\alpha\beta}=\mu_{\beta\alpha}$. The covariance is further invariant under the exchange of $(\alpha,\beta) \leftrightarrow (\gamma,\delta)$ and, due to the symmetry of $K^0_{\alpha\beta}=K^0_{\beta\alpha}$, also under swapping $\alpha \leftrightarrow \beta$ and $\gamma \leftrightarrow \delta$ separately. In the artificial task-setting, the parameter p, the pattern dimensionality

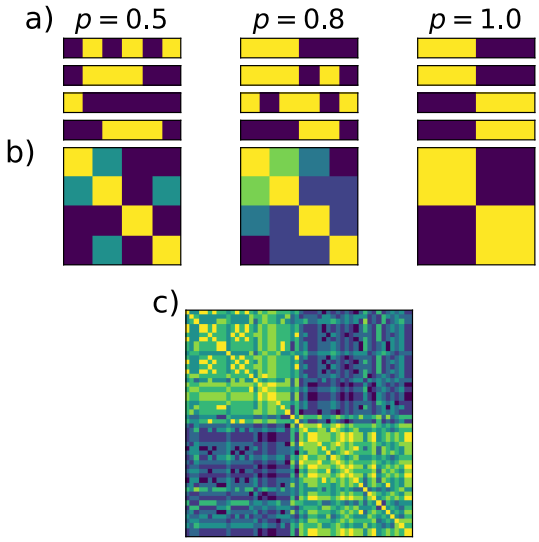


Figure 55: **Synthetic data set.**
a) Two sample vectors $x^{(1\leq\alpha\leq 4)}$ for each of the two classes (upper, lower). The values for the pixels can be $x_i^{(\alpha)} \in \{-1, 1\}$ where yellow indicates $x_i^{(\alpha)} = 1$ and blue $x_i^{(\alpha)} = -1$. The three columns correspond to three different settings of the pixel probability $p \in \{0.5, 0.8, 1\}$.
b) Empirical overlap matrices K^x (7.17) for $D/2 = 2$ patterns of class 1 and $D/2 = 2$ patterns of class 2. The entries of the overlap matrices are $K^x_{\alpha\beta} \in [-1, 1]$. Darker colors indicate $K^x_{\alpha\beta} \approx -1$ and brighter colors correspond to $K^x_{\alpha\beta} \approx 1$.
c) Empirical overlap matrix K^x for the same task as in a), b) with $D = 50$. Other parameters: $N_{\text{dim}} = 6$.

N_{dim}, and the variance $\sigma_v^2/N_{\text{dim}}$ of each read-in weight V_{ij} define the elements of $\mu_{\alpha\beta}$ and $\Sigma_{(\alpha\beta)(\gamma\delta)}$, which read

$$
\begin{aligned}
\mu_{\alpha\beta} &= \sigma_v^2 \begin{cases} 1 & \alpha = \beta \\ u & c_\alpha = c_\beta \\ -u & c_\alpha \neq c_\beta \end{cases}, \\
\Sigma_{(\alpha\beta)(\alpha\beta)} &= \frac{\sigma_v^4}{N_{\text{dim}}}\kappa, \\
\Sigma_{(\alpha\beta)(\alpha\delta)} &= \frac{\sigma_v^4}{N_{\text{dim}}} \begin{cases} \nu & \text{for } \begin{cases} c_\alpha = c_\beta = c_\delta \\ c_\alpha \neq c_\beta = c_\delta \end{cases} \\ -\nu & \text{for } \begin{cases} c_\alpha = c_\beta \neq c_\delta \\ c_\alpha = c_\delta \neq c_\beta \end{cases} \end{cases}, \\
\text{with} \quad \kappa &:= 1 - u^2\,, \\
\nu &:= u\,(1-u)\,, \\
u &:= 4p(p-1) + 1\,.
\end{aligned}
\tag{7.19}
$$

In addition to this, the tensor elements of $\Sigma_{(\alpha\beta)(\gamma\delta)}$ are zero for the following index combinations because we fixed the value of $K^0_{\alpha\alpha}$ by construction:

$$\begin{aligned}
\Sigma_{(\alpha\beta)(\gamma\delta)} &= 0 \quad \text{with} \quad \alpha \neq \beta \neq \gamma \neq \delta\,, \\
\Sigma_{(\alpha\alpha)(\beta\gamma)} &= 0 \quad \text{with} \quad \alpha \neq \beta \neq \gamma\,, \\
\Sigma_{(\alpha\alpha)(\beta\beta)} &= 0 \quad \text{with} \quad \alpha \neq \beta\,, \\
\Sigma_{(\alpha\alpha)(\alpha\beta)} &= 0 \quad \text{with} \quad \alpha \neq \beta\,, \\
\Sigma_{(\alpha\alpha)(\alpha\alpha)} &= 0 \quad \text{with} \quad \alpha \neq \beta\,.
\end{aligned} \tag{7.20}$$

The expressions for $\Sigma_{(\alpha\beta)(\alpha\beta)}$ and $\Sigma_{(\alpha\beta)(\alpha\delta)}$ in (7.19) show that the magnitude of the fluctuations are controlled through the parameter p and the pattern dimensionality N_{dim}: The covariance Σ is suppressed by a factor of $1/N_{\text{dim}}$ compared to the mean values μ. Hence we can use the pattern dimensionality N_{dim} to investigate the influence of the strength of fluctuations. As illustrated in Figure 54, the elements $\Sigma_{(\alpha\beta)(\alpha\beta)}$ denote the variance of individual entries of the kernel, while $\Sigma_{(\alpha\beta)(\alpha\gamma)}$ are covariances of entries across elements of a given row α, visible as horizontal or vertical stripes in the color plot of the kernel.

Equation (7.19) implies, by construction, a Gaussian distribution of the elements $K^0_{\alpha\beta}$ as it only provides the first two cumulants. One can show that the higher-order cumulants of $K^0_{\alpha\beta}$ scale sub-leading in the pattern dimension and are hence suppressed by a factor $\mathcal{O}\left(1/N_{\text{dim}}\right)$ compared to $\Sigma_{(\alpha\beta)(\gamma\delta)}$.

7.2 ML as statistical field theory

In this section we derive the field theoretic formalism which allows us to compute the statistical properties of the inferred network output in Bayesian inference with a stochastic kernel. We show that the resulting process is non-Gaussian and reminiscent of a $\phi^3+\phi^4$-theory. Specifically, we compute the mean of the predictive distribution of this process conditioned on the training data. This is achieved by employing systematic approximations with the help of Feynman diagrams.

Subsequently we show that our results provide an accurate bound on the generalization capabilities of the network. We further discuss the implications of our analytic results for neural architecture search.

7.2.1 Field theoretic description of Bayesian inference

Bayesian inference with stochastic kernels In general, a network implements a map from the inputs x_α to corresponding outputs y_α. In particular a model of the form (7.4) implements a non-linear map $\psi : \mathbb{R}^{N_{\text{dim}}} \to \mathbb{R}^{N_h}$ of the input $x_\alpha \in \mathbb{R}^{N_{\text{dim}}}$ to a hidden state $h_\alpha \in \mathbb{R}^{N_h}$. This map may also involve multiple hidden-layers, biases and non-linear transformations. The read-out weight $\mathbf{U} \in \mathbb{R}^{1\times N_h}$ links the scalar network output $y_\alpha \in \mathbb{R}$ and the transformed inputs $\psi\left(x_\alpha\right)$ with $1 \leq \alpha \leq D_{\text{tot}} = D + D_{\text{test}}$ which yields

$$y_\alpha = \mathbf{U}\,\psi\left(x_\alpha\right) + \xi_\alpha\,, \tag{7.21}$$

where $\xi_\alpha \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2_{\text{reg}})$ is a regularization noise. We assume that the prior on the read-out vector elements is a Gaussian

$$\mathbf{U}_i \overset{\text{iid}}{\sim} \mathcal{N}\left(0, \sigma_u^2/N_h\right). \tag{7.22}$$

The distribution of the set of network outputs $y_{1\leq\alpha\leq D_{\text{tot}}}$ is then in the limit $N_h \to \infty$ a multivariate Gaussian. The kernel matrix of this Gaussian is obtained by taking the expectation value with respect to the read-out vector, which yields

$$\left\langle y_\alpha\, y_\beta \right\rangle_{\mathbf{U}} =: K^y_{\alpha\beta} = \sigma_u^2\, K^\psi_{\alpha\beta} + \delta_{\alpha\beta}\,\sigma^2_{\text{reg}}\,, \tag{7.23}$$

$$K^\psi_{\alpha\beta} = \frac{1}{N_h}\sum_{i=1}^{N_h} \psi_i\left(x_\alpha\right)\,\psi_i\left(x_\beta\right)\,. \tag{7.24}$$

The kernel matrix $K^y_{\alpha\beta}$ describes the covariance of the network's output and hence depends on the kernel matrix $K^\psi_{\alpha\beta}$. The additional term $\delta_{\alpha\beta}\,\sigma^2_{\text{reg}}$ acts as a regularization term, which is also known as a ridge regression or Tikhonov regularization. In the context of neural networks one can motivate the regularizer σ^2_{reg} by using the L^2-regularization in the readout layer.

This is also known as weight decay [43]. Introducing the regularizer σ^2_{reg} is necessary to ensure that one can properly invert the matrix $K^y_{\alpha\beta}$, ensuring that the expressions (7.14) are numerically stable.

Different drawings of sets of training data $\mathcal{D}_{\text{tr}}$ lead to different realizations of kernel matrices K^ψ and K^y. The network output y_α hence follows a multivariate Gaussian with a stochastic kernel matrix K^y. A more formal derivation of the Gaussian statistics, including an argument for its validity in deep neural networks, can be found elsewhere. A consistent derivation using field theoretical methods and corrections in terms for the width of the hidden layer N_h for deep and recurrent networks can also be found elsewhere.

In general, the input kernel matrix K^0 (7.17) and the output kernel matrix K^y are related in a non-trivial fashion, which depends on the specific network architecture at hand. From now on we make an assumption on the stochasticity of K^0 and assume that the input kernel matrix K^0 is distributed according to a multivariate Gaussian

$$K^0 \sim \mathcal{N}\left(\mu,\, \Sigma\right) , \tag{7.25}$$

where μ and Σ are provided in (7.18).

In the limit of large pattern dimensions $N_{\text{dim}} \gg 1$ this assumption is warranted for the kernel matrix K^0. This structure further assumes, that the overlap statistics are unimodal, which is indeed mostly the case for data such as MNIST Furthermore we assume that this property holds for the output kernel matrix K^y as well and that we can find a mapping from the mean μ and covariance Σ of the input kernel to the mean m and covariance C of the output kernel $\left(\mu_{\alpha\beta},\, \Sigma_{(\alpha\beta)(\gamma\delta)}\right) \to \left(m_{\alpha\beta},\, C_{(\alpha\beta)(\gamma\delta)}\right)$ so that K^y is also distributed according to a multivariate Gaussian

$$K^y \sim \mathcal{N}(m,\, C) . \tag{7.26}$$

For each realization $K^y_{\alpha\beta}$, the joint distribution of the network outputs $y_{1\leq\alpha\leq D_{\text{tot}}}$ corresponding to the training and test data points $\mathbf{x}$ follow a multivariate Gaussian

$$p\left(\mathbf{y}|\mathbf{x}\right) \sim \mathcal{N}\big(0, K^y\big) . \tag{7.27}$$

The kernel allows us to compute the conditional probability $p\left(y_*|\mathbf{x}_{\text{tr}}, \mathbf{y}_{\text{tr}}, x_*\right)$, as defined in (7.13), for a test point $(x_*, y_*) \in \mathcal{D}_{\text{test}}$ conditioned on the data from the training set $(\mathbf{x}_{\text{tr}}, \mathbf{y}_{\text{tr}}) \in \mathcal{D}_{\text{tr}}$. This distribution is Gaussian with mean and variance given by (7.14). It is our goal to take into account that K^0 is a stochastic quantity, which depends on the particular draw of the training and test data set $(\mathbf{x}_{\text{tr}}, \mathbf{y}_{\text{tr}}) \in \mathcal{D}_{\text{tr}}, (x_*, y_*) \in \mathcal{D}_{\text{test}}$. The labels $\mathbf{y}_{\text{tr}}, y_*$ are, by construction, deterministic and take either one of the values ± 1. In the following we investigate the dependence of the mean of the predictive distribution on the number of training samples, which we call the learning curve. A common assumption is that this learning curve is rather insensitive to the very realization of the chosen training points. Thus we assume that the learning curve is self-averaging. The mean computed for a single draw of the training data is hence expected to agree well to the average over many such drawings. Under this assumption it is sufficient to compute the data-averaged mean inferred network output, which reduces to computing the disorder-average of the following quantity

$$\langle y_* \rangle_{K^y} = \left\langle K^y_{*\alpha} \left[K^y\right]^{-1}_{\alpha\beta} \right\rangle_{K^y} y_\beta \, . \tag{7.28}$$

To perform the disorder average and to compute perturbative corrections, we will follow these steps

- construct a suitable dynamic moment-generating function $Z(K^y)$,
- propagate the input stochasticity to the network output $K^0_{\alpha\beta} \to K^y_{\alpha\beta}$,
- disorder-average the functional using the model $K^y_{\alpha\beta} \sim \mathcal{N}(m_{\alpha\beta},\, C_{(\alpha\beta)(\gamma\delta)})$,
- and finally perform the computation of perturbative corrections using diagrammatic techniques.

Constructing the dynamic moment generating function Our ultimate goal is to compute learning curves. Therefore we want to evaluate the disorder averaged mean inferred network output (7.28). Both the presence of two correlated random matrices and the fact that one of the matrices appears as an inverse complicate this process. One alternative route is to define the moment-generating function

$$Z(l_*) = \int dy_* \exp(l_* y_*) p\left(y_*|x_*, \mathbf{x}_{\text{tr}}, \mathbf{y}_{\text{tr}}\right) = \frac{\int dy_* \exp(l_* y_*) p\left(y_*, \mathbf{y}_{\text{tr}}|x_*, \mathbf{x}_{\text{tr}}\right)}{p\left(\mathbf{y}_{\text{tr}}|\mathbf{x}_{\text{tr}}\right)} =: \frac{\mathcal{Z}(l_*)}{\mathcal{Z}(0)} , \tag{7.29}$$

with joint Gaussian distributions $p\left(y_*, \mathbf{y}_{\rm tr}|x_*, \mathbf{x}_{\rm tr}\right)$ and $p\left(\mathbf{y}_{\rm tr}|\mathbf{x}_{\rm tr}\right)$ that each can be readily averaged over K^y. Equation (7.28) is then obtained as

$$\langle y_* \rangle_{K^y} = \frac{\partial}{\partial l_*} \left\langle \frac{\mathcal{Z}(l_*)}{\mathcal{Z}(0)} \right\rangle_{K^y} \Bigg|_{l_*=0} . \tag{7.30}$$

A complication of this approach is that the numerator and denominator co-fluctuate. The common route around this problem is to consider the cumulant-generating function

$$W(l_*) = \ln \mathcal{Z}(l_*) \,, \tag{7.31}$$

and to obtain

$$\langle y_* \rangle_{K^y} = \frac{\partial}{\partial l_*} \langle W(l_*) \rangle_{K^y} \,, \tag{7.32}$$

which, however, requires averaging the logarithm. This is commonly done with the replica trick

We here follow a different route to ensure that the disorder-dependent normalization $\mathcal{Z}(0)$ drops out and construct a dynamic moment generating function Our goal is hence to design a dynamic process where a time dependent observable is related to y_*, our mean-inferred network output. We hence define the linear process in the auxiliary variables q_α

$$\frac{\partial q_\alpha(t)}{\partial t} = -K^y_{\alpha\beta}\, q_\beta(t) + y_\alpha \,, \tag{7.33}$$

for $(x_\alpha, y_\alpha) \in \mathcal{D}_{\rm tr}$. From this we see directly that $q_\alpha(t \to \infty) = [K^y]^{-1}_{\alpha\beta}\, y_\beta$ is a fixed point. The fact that $K^y_{\alpha\beta}$ is a covariance matrix ensures that it is positive semi-definite and hence implies the convergence to a fixed point. We can obtain (7.14) $\langle y_* \rangle = K^y_{*\alpha} [K^y]^{-1}_{\alpha\beta}\, y_\beta$ from (7.33) as a linear readout of $q_\alpha(t \to \infty)$ with the matrix $K^y_{*\alpha}$. Using the Martin-Siggia-Rose-deDominicis-Janssen (MSRDJ) formalism, see e.g. [44], one can express this as the first functional derivative of the moment generating function $Z(L_*, K^y)$ in frequency space

$$Z(L_*, K^y) = \int \mathcal{D}\{Q, \tilde{Q}\} \exp\Big(S(Q, \tilde{Q}, L_*)\Big) \,, \tag{7.34}$$

with

$$S(Q, \tilde{Q}, L_*) = \tilde{Q}^\top_\alpha \left(i\omega \mathbb{I} + K^y\right)_{\alpha\beta} Q_\beta - \tilde{Q}^0_\alpha y_\alpha + K^y_{*\alpha} L^\top_* Q_\alpha \,, \tag{7.35}$$

where $\tilde{Q}^\top_\alpha (\cdots) Q_\beta = \frac{1}{2\pi} \int d\omega \tilde{Q}_\alpha(-\omega) (\cdots) Q_\beta(\omega)$ and $\tilde{Q}_\alpha(\omega = 0) = \tilde{Q}^0_\alpha$. As $Z(L_*, K^y)$ is normalized such that $Z(0, K^y) = 1 \quad \forall\, K^y$, we can compute (7.28) by evaluating the functional derivative of the disorder-averaged moment-generating function $\overline{Z}(L_*)$ at $t \to \infty$:

$$\overline{Z}\left(L_*\right) = \left\langle \int \mathcal{D}\{Q, \tilde{Q}\} \exp\Big(S(Q, \tilde{Q}, L_*)\Big) \right\rangle_{K^y} , \qquad \langle y_* \rangle_{K^y} = \lim_{t\to\infty} \int d\omega \exp(i\omega t) \frac{\delta \overline{Z}\left(L_*\right)}{\delta L_*(-\omega)} \Bigg|_{L_*(-\omega)=0} . \tag{7.36}$$

By construction the distribution of the kernel matrix entries $K^y_{\alpha\beta}$ is a multivariate Gaussian (7.25). With eq. (7.36) we finally have arrived at the desired form of a 'statistical' integral.

Disorder averaged moment generating function To compute the disorder averaged mean-inferred network output (7.28) we need to compute the disorder average of the dynamic moment generating function $\overline{Z}(L_*) = \langle Z(L_*, K^y) \rangle_{K^y}$ and its functional derivative at $L_*(\omega) = 0$. Due to the linear appearance of K^y in the action (7.35) and the Gaussian distribution for K^y (7.26) we perform the disorder average directly and obtain the action

$$\overline{Z}\left(L_*\right) = \langle Z(L_*, K^y) \rangle_{K^y} := \int \mathcal{D}\{Q, \tilde{Q}\} \exp\Big(\overline{S}(Q, \tilde{Q}, L_*)\Big) \,, \tag{7.37}$$

with the action

$$\overline{S}(Q,\tilde{Q},L_*) = \tilde{Q}_\alpha^\top \left[i\omega \mathbb{I}_{\alpha\beta} + m_{\alpha\beta}\right] Q_\beta - \tilde{Q}_\alpha^0 y_\alpha + m_{*\alpha} L_*^\top Q_\alpha + \frac{1}{2} C_{(\alpha\beta)(\gamma\delta)} \tilde{Q}_\alpha^\top Q_\beta \tilde{Q}_\gamma^\top Q_\delta + C_{(*\alpha)(\beta\gamma)} L_*^\top Q_\alpha \tilde{Q}_\beta^\top Q_\gamma + \mathcal{O}\left(L_*^2\right) , \tag{7.38}$$

with $\tilde{Q}^0 := \tilde{Q}(\omega = 0)$ As we ultimately aim to obtain corrections for the mean inferred network output $\langle y_* \rangle_{K^y}$, we utilize the action in (7.38) and established results from field theory to derive the leading order terms as well as perturbative corrections diagrammatically. The presence of the variance and covariance terms in (7.38) introduces corrective factors, which cannot appear in the zeroth-order approximation, which corresponds to the homogeneous kernel that neglects fluctuations in K^y by setting $C_{(\alpha\beta)(\gamma\delta)} = 0$. This provides us with the tools to derive an asymptotic bound for the mean inferred network output $\langle y^* \rangle$ in the case of an infinitely large training data set. This bound is directly controlled by the variability in the data.

In summary we have found a representation of the Bayesian network in terms of a statistical theory which features cubic and quartic self-interaction terms, if dropping higher order terms in the source L. A detailed analysis is beyond the scope of this lecture course and we refer to the original work [41] for further reading.

8 Learning trivializing flows for statistical theories

In the final Chapter of the lecture course we discuss the application of normalizing flows to the task of learning the probability measure of a statistical theory. This allows us to use the network for the generation of a set of configurations the given statistical theory.

We resort to our standard guinea pig, the scalar ϕ^4 theory introduced in Section 3.1.1, see (3.10) and (3.11). For classification tasks in this theory see Sections 3.2.2 and 4.3. Applying normalizing flows as introduced in Section 5.4 to a given (training) set of this theory maps its distribution

$$p(\phi) = \frac{1}{Z} e^{-S[\phi]} , \qquad \text{with} \qquad S[\phi] = \sum_x \left[-\beta \sum_{\mu=1}^{2} \phi_{x+e_\mu} \phi_x + \phi_x^2 + \lambda(\phi_x^2 - 1)^2 \right] , \tag{8.1}$$

to a normal distribution. Here Z is the partition function (3.10). The action S in (8.1) is that in (3.10) with $x = (i_1, i_2)$ and $\beta = 2\kappa$ up to an additional constant term $\lambda \sum_x$, proportional to the volume. Evidently, this term drops out for normalized correlation functions.

We consider the same observables as before: The first is the magnetization M defined in (3.39),

$$M = \frac{1}{V} \sum_x \langle \phi_x \rangle , \tag{8.2}$$

and the lattice volume is given by $V = L^2$, $L = N$. The second observable is the the two-point correlation function,

$$G(y) = \frac{1}{V} \sum_x \langle (\phi_{x+y} - \langle\phi\rangle)(\phi_x - \langle\phi\rangle) \rangle = \frac{1}{V} \sum_x \langle \phi_{x+y} \phi_x \rangle - \langle \phi \rangle^2 , \tag{8.3}$$

or variants thereof. The expectation value $\langle\phi\rangle = \langle\phi_x\rangle$. It is position independent due to translation invariance. The correlation length can be extracted from the spatially-averaged two-point function,

$$\sum_{y_1=0}^{L-1} G(y_1, y_2) \propto \cosh\left(\frac{y_2 - L/2}{\xi} \right) , \tag{8.4}$$

at sufficiently large Euclidean time separations y_2 (It corresponds to the inverse mass of the lightest mode in the spectrum of the theory).

We can also measure the one-point susceptibility,

$$\chi_0 \equiv G(0) = \frac{1}{V} \sum_x \left[\langle \phi_x^2 \rangle - \langle \phi \rangle^2 \right] . \tag{8.5}$$

This sets the stage for the discussion of trivializing flows in statistical theories in the next chapter, Section 8.1.

8.1 Trivializing flows

Trivializing flow in lattice or statistical theories have been introduced to map the task of sampling with the *target* distribution $p(\phi)$ into one of a theory with a normal distribution. This task is implemented via a change of variables

$$\tilde{\phi} = \mathcal{F}^{-1}(\phi)\,, \tag{8.6}$$

in (3.10). We are led to

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int \prod_x d\phi_x \, \mathcal{O}(\phi)\, e^{-S[\phi]} = \frac{1}{\mathcal{Z}} \int \prod_x d\tilde{\phi}_x \, \mathcal{O}(\mathcal{F}(\tilde{\phi})) e^{-S[\tilde{\phi}]}\,, \tag{8.7}$$

with The Jacobian J of the variable transformation $\phi \to \tilde{\phi}$ and the action S_{eff},

$$S_{\text{eff}}(\tilde{\phi}) = S[\mathcal{F}(\tilde{\phi})] + \log\det J[\mathcal{F}(\tilde{\phi})]\,, \qquad \text{and} \qquad \int \prod_x d\phi_x = \int \prod_x d\tilde{\phi}_x \, \det J[\mathcal{F}]\,, \tag{8.8}$$

If $\mathcal{F}$ is chosen such that the action for the transformed field, $S_{\text{eff}}[\tilde{\phi}]$ is Gaußian, the transformation $\mathcal{F}$ is called a *trivializing map*.

A trivializing flow, introduced in [45], is a flow

$$\dot{\phi}_t \equiv T[t, \phi_t]\,, \tag{8.9}$$

with the flow time $t \in [0, 1]$. The initial condition at $t = 0$ is given by

$$\phi = \phi_0\,, \tag{8.10}$$

and a trivializing flow obeys

$$\tilde{\phi} = \mathcal{F}^{-1}(\phi) = \phi_1\,. \tag{8.11}$$

Though not known in closed form, the kernel T of the trivializing flow can be expressed as a power series in the flow time t. In practice, this power series was truncated at leading order and the flow integrated numerically, resulting in an approximate trivializing map where the effective action in (8.8) is still interacting in general. Nevertheless, S_{eff} ought to be easier to sample than S.

The algorithm introduced in [45] is essentially the HMC algorithm applied to the flowed field variables $\tilde{\phi}$. This algorithm was tested for the CP^{N-1} model in [46] for the task of reducing the topology freezing [47]. The conclusion of the study was that, although there was a small improvement in the proportionality factor, the overall scaling of the computational cost towards the continuum did not change with respect to standard HMC.

8.2 Flow HMC (FHMC)

It is clear from the above that the idea behind trivilizing flows is the same as that being normalizing flows as discussed in Section 5.4 (More accurately, it is the other was around). This suggests to adapt normalizing flows to the present setup, and the following analysis is taken from [48].

Normalizing flows have been introduced in statistical and lattice theories in [49], for a review see [33]. Starting from an initial set of configurations $\{z_i\}_{i=1}^N$ drawn from a probability distribution where sampling is cost-efficient,

$$z_i \sim r(z), \tag{8.12}$$

a transformation $\phi = f^{-1}(z)$ is applied so that the transformed configurations $\{\phi_i\}_{i=1}^N \equiv \{f^{-1}(z_i)\}_{i=1}^N$ follow the new probability distribution

$$p_f(\phi) = r\big(f(\phi)\big) \left| \det \frac{\partial f(\phi)}{\partial \phi} \right|. \tag{8.13}$$

The probability density p_f is called the *model distribution*. The transformation f is implemented via NNs with a set of trainable parameters $\{\theta_i\}$ which have been optimized so that $p_f(\phi)$ is as close as possible to the target distribution $p(\phi) = e^{-S(\phi)}/\mathcal{Z}$, i.e. the distribution of the theory we are interested in. The determinant of this transformation can be easily computed if the network architecture consists of coupling layers with a checkerboard pattern, as explained in [49, 33]. Normalizing Flows are therefore NN parametrisations of trivializing maps.

Ideally, the NNs would be trained such that the Kullback-Leibler (KL) divergence between the model and the target distribution,

$$D_{\mathrm{KL}}(p_f||p) = \int \prod_x d\phi_x \; p_f(\phi) \log \frac{p_f(\phi)}{p(\phi)} \,, \tag{8.14}$$

is minimised. However, the partition function $\mathcal{Z}$ appearing in $p(\phi)$ is generally not known, so in practice one minimizes a shifted KL divergence,

$$L(\theta) \equiv D_{\mathrm{KL}}(p_f \,||\, p) - \log \mathcal{Z} = \int \prod_x d\phi_x \, p_f(\phi) \left[\log p_f(\phi) + S(\phi)\right] \,. \tag{8.15}$$

This loss function can be stochastically estimated by drawing samples $\phi_i \sim p_f(\phi)$ from the model; there is no requirement to have a set of existing training data. Since f is differentiable, the loss can be minimized using standard gradient-based optimization algorithms such as stochastic gradient descent and ADAM [50], which we used here. The absolute minimum of the loss function occurs when $L(\theta) = -\log \mathcal{Z}$, where $p_f = p$. In practice this minimum is unlikely to be achieved due to both the limited *expressivity* of the model and a finite amount of training, but one expects to have an approximate trivializing map at the end of the training, i.e. $p_f \approx p$.

The Normalizing Flow model generates configurations that are distributed according to p_f, not p. To achieve unbiased sampling from p, in [49, 33] the model is embedded in a Metropolis–Hastings (MH) algorithm where p_f serves as the proposal distribution. Since the proposals are statistically independent, the only source of autocorrelations are rejections. However, training models to achieve a fixed low MH rejection rate can become prohibitively expensive for large systems and long correlation lengths [51].

In contrast, in this work we propose to use the trained model as an approximation to the trivializing map in the implementation of the trivializing flow algorithm described in Section 8.1. Thus we identify

$$\mathcal{F} = f^{-1} \,, \tag{8.16}$$

and

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_x d\phi_x \, \mathcal{O}\left(f^{-1}(\tilde{\phi})\right) \, e^{-S[f^{-1}(\tilde{\phi})] + \log\det J[f^{-1}(\tilde{\phi})]} \equiv \frac{1}{\mathcal{Z}} \int \prod_x d\phi_x \, \mathcal{O}\left(f^{-1}(\tilde{\phi})\right) \, e^{-S_{\mathrm{eff}}[\tilde{\phi}]} \,, \tag{8.17}$$

where we have defined the new action S_{eff} in the transformed coordinates to be

$$S_{\mathrm{eff}}(\tilde{\phi}) \equiv S(f^{-1}(\tilde{\phi})) - \log\det J[f^{-1}(\tilde{\phi})] \,. \tag{8.18}$$

If the new probability distribution, $\exp\{-S_{\mathrm{eff}}(\tilde{\phi})\}$ is easier to sample than $\exp\{-S(\phi)\}$ then performing HMC with the new variables, $\tilde{\phi}$, would result in a Markov chain $\{\tilde{\phi}_i\}_{i=1}^N$ with lower autocorrelation times for the observable $\mathcal{O}$.

We will refer to this algorithm as Flow HMC (FHMC), and its workflow is as follows:

1. Train the network f by minimising the KL divergence in Equation (8.15).
2. Run the HMC algorithm to build a Markov chain of configurations using the action S_{eff},
$$\{\tilde{\phi}_1, \; \tilde{\phi}_2, \; \tilde{\phi}_3, \; \ldots, \; \tilde{\phi}_N\} \sim e^{-S_{\mathrm{eff}}(\tilde{\phi})} \,. \tag{8.19}$$
3. Apply the inverse transformation f^{-1} to every configuration in the Markov chain to undo the variable transformation. This way we obtain a Markov chain of configurations following the target probability distribution $p(\phi) = \exp\{-S[\phi]\}$,
$$\{f^{-1}(\tilde{\phi}_1), \; f^{-1}(\tilde{\phi}_2), \; f^{-1}(\tilde{\phi}_3), \; \ldots, \; f^{-1}(\tilde{\phi}_N)\} = \{\phi_1, \; \phi_2, \; \phi_3, \; \ldots, \; \phi_N\} \sim e^{-S(\phi)} \,. \tag{8.20}$$

Note that the HMC acceptance in step 2 can be made arbitrarily high by increasing the number of integration steps in the molecular dynamics evolution of HMC. Contrary to what happens in the algorithm suggested in Reference [49], this acceptance does not measure how well f^{-1} approximates a trivializing map; the relevant question is whether this algorithm improves the autocorrelation of HMC.

The motivation behind this setup is that a Normalizing Flow parametrized by NNs ought to be better able to approximate a trivializing map than the leading-order approximation of the flow equation introduced and tested in [45, 46].

Algorithm 1 HMC

function HMC(ϕ, S)
  $\pi \leftarrow$ GENERATEMOMENTA
  $(\phi', \pi') \leftarrow$ LEAPFROG(ϕ, π, S)
  $\phi_{\text{new}} \leftarrow$ ACCEPTREJECT($(\phi', \pi'), (\phi, \pi), S$)
  return ϕ_{new}
end function

8.3 Network architecture and training

We focus on keeping the training costs negligible with respect to the cost of producing configurations with FHMC. The most intuitive choices that we took for this optimization are:

1. Use Convolutional Neural Networks (CNNs) instead of fully connected networks. The action S in (8.1) has translational symmetry, so the network f should apply the same transformation to ϕ_x for all x. CNNs respect this translational symmetry, and also require less parameters than fully connected networks.
2. Tune the number of layers and kernel sizes of the CNNs so that the footprint of the network f is not much bigger than the correlation length ξ. Two-point correlation functions will generally decay with $\sim e^{-|x-y|/\xi}$, so the transformation of ϕ_x should not depend on ϕ_y if $|x-y| \gg \xi$. Also, limiting the number of layers reduces the number of trainable parameters.
3. Enforce f to satisfy $f(-\phi) = -f(\phi)$. The action S in (8.1) is invariant under $\phi \to -\phi$, so enforcing equivariance under this symmetry should optimize training costs.

Following [49, 33], we partition the lattice using a checkerboard pattern, so that each field configuration can be split as $\phi = \{\phi^A, \phi^B\}$, where ϕ^A and ϕ^B collectively denote the field variables belonging to one or the other partition. We then construct the transformation f as a composition of n layers,

$$f(\phi) = g^{(1)}(g^{(2)}(\dots g^{(n)}(\phi)\dots)), \tag{8.21}$$

where each layer $g^{(i)}$ does an affine transformation to a set of the field variables, $\{\phi^A, \phi^B\}$, organized in a checkerboard pattern, such as

$$\phi^A_x = \begin{cases} \phi_x & \text{if } x_1 + x_2 \text{ odd} \\ 0 & \text{otherwise} \end{cases}, \qquad \phi^B_x = \begin{cases} 0 & \text{if } x_1 + x_2 \text{ odd} \\ \phi_x & \text{otherwise} \end{cases}, \tag{8.22}$$

where $x = (x_1, x_2)$. In the affine transformation

$$g^{(i)}(\{\phi^A, \phi^B\}) = \{\phi^A, \phi^B \odot e^{|s^{(i)}(\phi^A)|} + t^{(i)}(\phi^A)\}\,, \tag{8.23}$$

the partition ϕ^A remains unchanged and only the field variables ϕ^B are updated. $s(\phi)$ and $t(\phi)$ are CNNs with kernel size k. To make this transformation equivariant under $\phi \to -\phi$ we enforce $f(-\phi) = -f(\phi)$ by using a tanh activation function and no bias for the CNNs (see Sec. III.F of [51]). The checkerboard pattern ensures that the Jacobian matrix has a triangular form so its determinant can be easily computed, and reads

$$\left|\det \frac{\partial g^{(i)}(\phi)}{\partial \phi}\right| = \prod_{\{x | \phi^B_x = \phi_x\}} e^{|s^{(i)}_x(\phi^A)|}\,, \tag{8.24}$$

where the product runs over the lattice points where the partition $\phi^B_x = \phi_x$. An example of the action of a CNN with only 1 layer and a tanh activation function over a lattice field would be

$$s^{(i)}_x(\phi^A) = \tanh\left[\sum_{y \in \left[-\frac{k-1}{2}, \frac{k-1}{2}\right]^2} w^{(i)}(y)\phi^A_{x-y}\,,\right], \tag{8.25}$$

L	6	8	10	12	14	16	18	20	40	80
β	0.537	0.576	0.601	0.616	0.626	0.634	0.641	0.645	0.667	0.677
λ	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Table 3: Studied parameters of the action in (8.1). β has been chosen so that $\xi = L/4$, so the continuum limit is taken in the direction of increasing L.

where $w^{(i)}(y) \equiv w^{(i)}_{y+(k+1)/2}$ is the weight matrix of size $k \times k$ of the CNN s of kernel size k. Choosing the same functional form for $t^{(i)}_x(\phi^A)$, it is easy to check that the transformation in Equation (8.23) is equivariant under $\{\phi^A, \phi^B\} \to \{-\phi^A, -\phi^B\}$.

Two different affine layers with alternate checkerboard patterns are necessary to transform the whole set of lattice points, and we will denote such a pair of layers as a *coupling* layer.[1]

In this work we studied network architectures with $N_l = 1$ affine coupling layers, while the CNNs, s and t, have kernel size k and no hidden layers. The output configuration is rescaled with an additional trainable parameter. Finally, independent normal distributions are used as the prior distributions r in Equation (8.12).

8.4 FHMC implementation

The main focus of this work is the scaling of the autocorrelation times of the magnetization, τ_M, and one-point susceptibilities, τ_{χ_0} and $\tau_{\chi_{0,t}}$. Using local update algorithms such as HMC, these autocorrelation times are expected to scale as [52]

$$\tau \sim \xi^2. \tag{8.26}$$

We will benchmark the scaling of the autocorrelation times in the FHMC algorithm against those in standard HMC.

For a scalar field theory, the HMC equations of motion read

$$\dot{\phi}_x = \pi_x, \qquad \dot{\pi}_x = -\nabla_{\phi_x} S[\phi], \tag{8.27}$$

where the force for the momenta π follows from the derivative of the action in (8.1),

$$\begin{aligned} F_x \equiv -\nabla_x S[\phi] = \beta \sum_{\mu=\pm 1}^{\pm 2} \phi_{x+e_\mu} \\ + 2\phi_x \left(2\lambda \left(1 - \phi_x^2\right) - 1\right). \end{aligned} \tag{8.28}$$

In our simulations we used a leapfrog integration scheme with a single time scale, and the step size of the integration was tuned to obtain acceptances of approximately 90%. A pseudocode of an HMC implementation is depicted in Algorithm 1: the HMC function receives as input a configuration, ϕ, and the action of the target theory, S; after generating random momenta, the leapfrog function performs the molecular dynamics step and a configuration, ϕ_{new}, is chosen between the evolved field, ϕ', and the old field, ϕ, with the usual MH accept–reject step.

The proposed FHMC algorithm is in essence the HMC algorithm with the transformed action in Equation (8.18), which arises from the change of variables $\tilde{\phi} = f(\phi)$ in Equation (8.17). The new Hamilton equations of motion now include derivatives with respect to the new variables $\tilde{\phi}$,

$$\dot{\tilde{\phi}}_x = \pi_x, \qquad \dot{\pi}_x = -\nabla_{\tilde{\phi}_x} S_{\text{eff}}[\tilde{\phi}]. \tag{8.29}$$

The basic implementation is sketched in Algorithm 2. The main differences with respect to standard HMC in Algorithm 1 are line 2, where we transform from the variables ϕ to the variables $\tilde{\phi}$ using the trained network f; and line 6, where we undo the change of variables to obtain the new configuration ϕ_{new} from $\tilde{\phi}_{\text{new}}$.

[1] See Reference [33] for an example of an actual implementation of all these concepts.

Algorithm 2 FHMC

1: **function** FHMC($\phi, \tilde{S}, f$)
2: $\tilde{\phi} \leftarrow f(\phi)$
3: $\pi \leftarrow$ GENERATEMOMENTA
4: $(\tilde{\phi}', \pi') \leftarrow$ LEAPFROG($\tilde{\phi}, \pi, \tilde{S}$)
5: $\tilde{\phi}_{\text{new}} \leftarrow$ ACCEPTREJECT($(\tilde{\phi}', \pi'), (\tilde{\phi}, \pi), \tilde{S}$)
6: $\phi_{\text{new}} \leftarrow f^{-1}(\tilde{\phi}_{\text{new}})$
7: **return** ϕ_{new}
8: **end function**

Note that the molecular dynamics evolution and the accept–reject step, lines 4 and 5, are applied to the transformed field variables $\tilde{\phi}$ with the new action S_{eff}. Irrespective of the transformation f, the acceptance rate can be made arbitrarily high by reducing numerical errors in the integration of the equations of motion in Equation (8.29). This means that we will always be able to tune the FHMC acceptance to approximately 90% by increasing the number of integration steps, even for a poorly trained Normalizing Flow.

Note also that now the evaluation of the force $\tilde{F}_x \equiv -\nabla_{\phi_x} S_{\text{eff}}[\tilde{\phi}]$ requires computing the derivative of $\log\det J[f^{-1}(\tilde{\phi})]$. This cannot be written analytically for an arbitrary network, and we used PyTorch's automatic differentiation methods for its evaluation.

8.5 Compilation of results

This part is taken directly from [48]. We are interested in the scaling of the cost towards the continuum limit. Following the analysis in [51], we tuned the coupling β so that the correlation length satisfies

$$\xi \approx \frac{L}{4}. \tag{8.30}$$

The continuum limit is therefore approached in the direction of increasing L. In Table 3 we summarise the parameters used in our simulations of FHMC and standard HMC. Results obtained with both HMC and FHMC can be found in tables Table 5 and Table 6.

8.5.1 Minimal network

An obvious strategy to reduce training costs is to build networks with few parameters to train. Using $N_l = 1$ coupling layer would suffice to transform the whole lattice, while a kernel size $k = 3$ for the CNNs, which would couple only nearest neighbour, is the smallest that can be used.[2] Such a network has only 37 trainable parameters in total,[3] and is the most minimal network that we will consider.

It is interesting to study whether such a simple network can learn physics of a target theory with a non-trivial correlation length. In Figure 56 (left) we plot the evolution of the KL divergence during the training of a network with such minimal architecture, where the target theory has parameters $\beta = 0.641$, $\lambda = 0.5$, lattice size $L = 18$ and correlation length $\xi = L/4 = 4.5$. Since the network has very few parameters, the KL divergence reaches saturation after only $\mathcal{O}(100)$ iterations.

Once the network is trained, one can use it as a variable transformation for the FHMC algorithm as sketched in Algorithm 2. In Figure 56 (right) we show the magnetizations of a slice of 4000 configurations of Markov chains coming from FHMC and HMC simulations, yielding as autocorrelation times

$$\tau_{M,\text{FHMC}} = 74.4(3), \quad \tau_{M,\text{HMC}} = 100.4(2). \tag{8.31}$$

[2] The transformation with $k = 1$ being a trivial rescaling.

[3] A CNN with 1 layer has k^2 parameters. The transformation in Equation (8.23) with N_l affine coupling layers has $4 \times N_l$ different CNNs, and therefore $4 \times N_l \times k^2$ parameters. Since we also add a global rescaling parameter as a final layer of our network, this architecture has $N_p = 4 \times N_l \times k^2 + 1$ parameters.

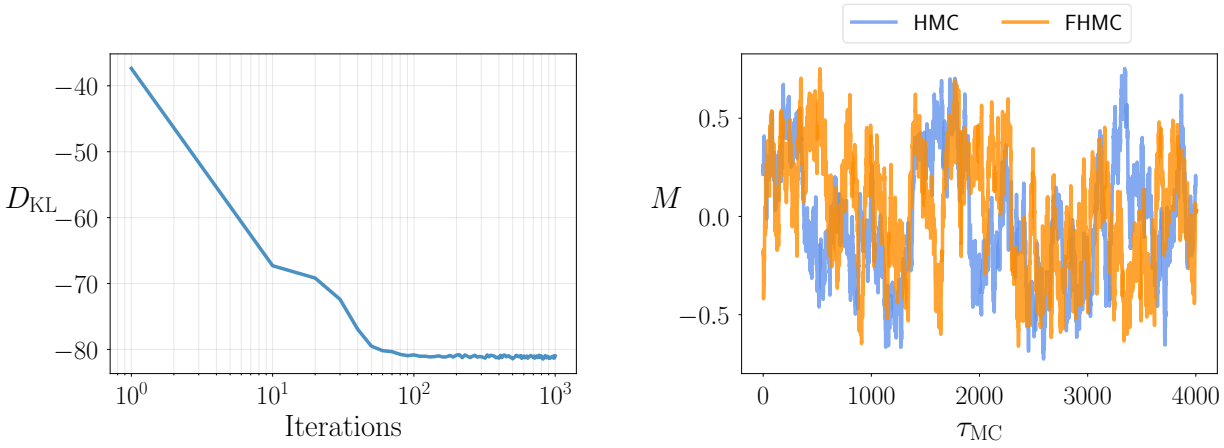


Figure 56: (Left) History of the KL divergence during the training from independent Gaussians to a theory with parameters $\beta = 0.641$, $\lambda = 0.5$, lattice size $L = 18$ and $\xi = L/4$. (Right) History of the magnetization for a simulation with HMC (blue) and FHMC (orange).

L	β	Acc. at L	Acc. at $2L$
3	0.537	0.3	0.2
4	0.576	0.04	0.001
5	0.601	0.002	0.00003
6	0.616	0.002	0.000007
7	0.626	0.0001	$< 10^{-7}$
8	0.634	0.0001	-
9	0.641	0.00007	-
10	0.645	0.00004	-

Table 4: MH acceptances of networks trained at lattice size L and used to sample a theory with coupling β at lattice sizes L and $2L$.

All the results with this minimal network can be found in Table 6. showing that FHMC leads to smaller autocorrelations compared to standard HMC, especially as the continuum limit is approached. This fact seems to indicate that a network with few parameters can indeed learn transformations with relevant physical information.

A measure of the closeness of the distributions p_f and p is given by the MH acceptance[4] when sampling p with configurations drawn directly from p_f. Focusing on the first three columns of Table 4, it is clear that the acceptances are low and decrease towards the continuum limit, in spite of the fact that the autocorrelation time of FHMC is better than that of HMC. This is because the networks used have a very reduced set of parameters and therefore limited expressivity. The map defined via the trained networks is not very accurate in generating the probability distribution of the target theory. However FHMC, i.e. a molecular dynamics evolution using flowed variables, yields a clear gain in the autocorrelation times.

8.5.2 Infinite volume limit

An important advantage of using a translationally-invariant network architecture, such as the one containing CNNs, is that they can be trained at a small lattice size L and then used in a larger lattice $L' > L$. Note that doing this in the approach of [49, 33, 51] would not be viable since it would lead to an exponential decrease in the MH acceptance due to the extensive character of the action.

In the last column of Table 4 we show the MH acceptance using networks with the minimal architecture of Section 8.5.1 trained at lattice size L when the target theory has lattice size $2L$. One can see that the acceptances are significantly lower than those obtained with the target theory at size L. However, the acceptance of the FHMC algorithm (Algorithm 2) can be kept arbitrarily high by increasing the number of integration steps of the Hamilton equations, so reusing the networks

[4]Not to be confused with the HMC and FHMC acceptances, which are tuned to 90% in this work.

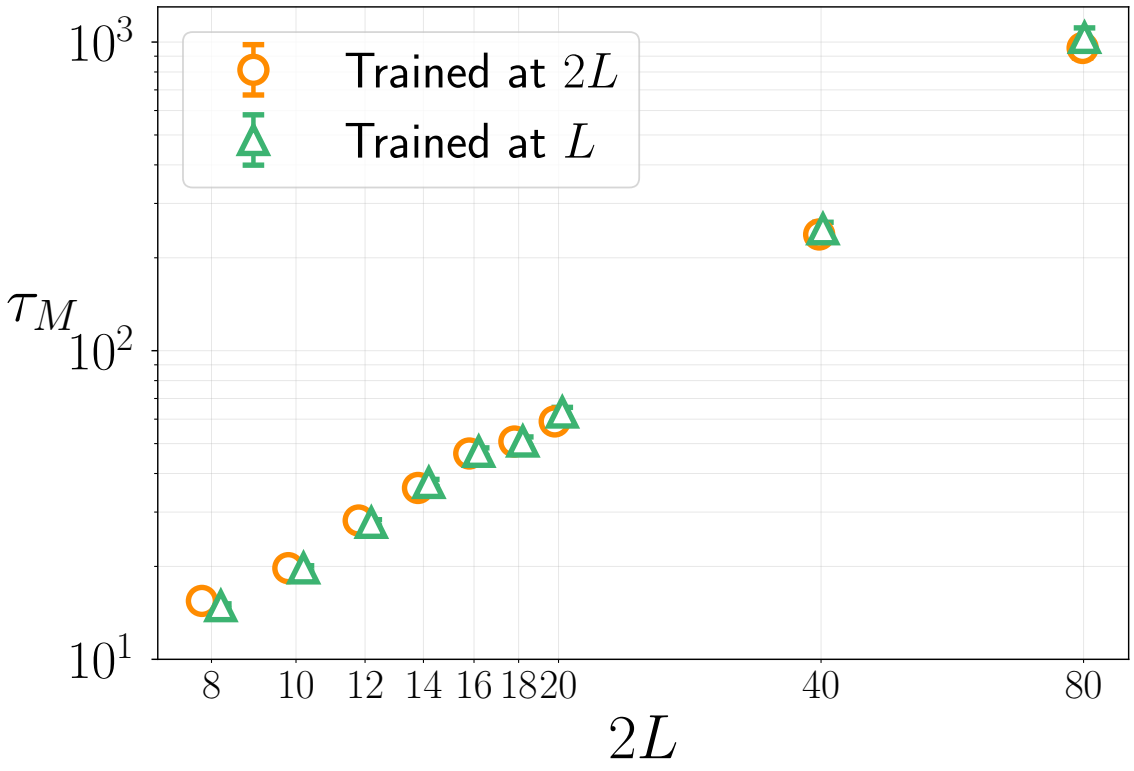


Figure 57: Autocorrelation time of the magnetization at lattice size $2L$ using FHMC. In circles, the networks used were trained at lattice size $2L$; in triangles, they were trained at L and used at $2L$.

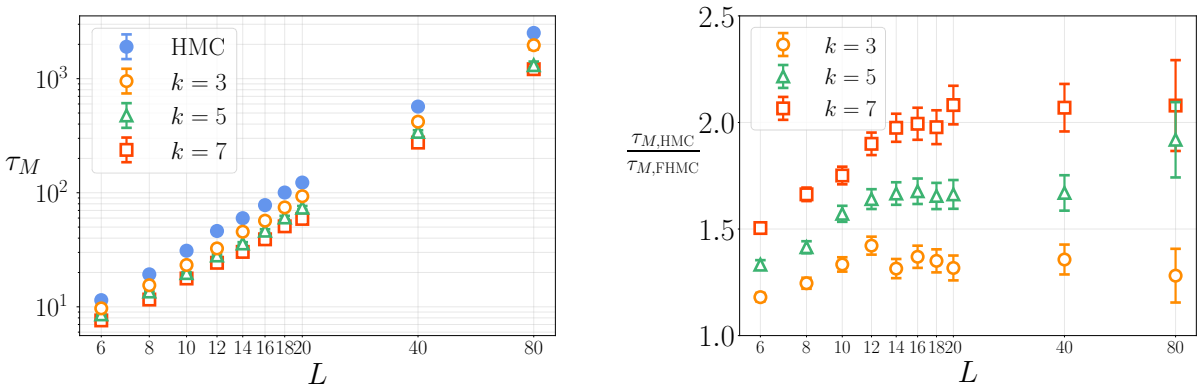


Figure 58: (Left) Scaling of the autocorrelation time of the magnetization towards the continuum for HMC (filled, blue circles) and FHMC with kernel sizes $k = 3, 5, 7$ (open circles, triangles and squares). (Right) Scaling of the ratio of autocorrelation times of the magnetization of HMC with respect to FHMC.

for higher volumes does not pose any problem. The reduced MH acceptance does not translate into a change in the autocorrelation time.

In Figure 57 we compare the autocorrelation times for the magnetization of networks trained at L and reused at $2L$ with the ones of networks trained directly at $2L$. Since they agree within statistical significance, this indicates that the relevant physical information is already learned at small volumes and reinforces the intuition that the training does not need to be done at a lattice sizes larger than ξ.

8.5.3 Continuum limit scaling with fixed architecture

Finally we want to determine whether the computational cost of FHMC has a better scaling than the standard HMC as we approach the continuum. First we will consider a fixed network architecture as we scale L. We have trained a different network for each of the lattice sizes in Table 3, with $N_l = 1$ and kernel sizes $k = 3, 5, 7$. The cost of the training is in all cases negligible with respect to the costs of the FHMC, and the integration step of the leapfrog scheme is tuned to have an acceptance rate of approximately 90% for every simulation, as we do for HMC.

In Figure 58 (left) we show the autocorrelation times of the magnetization for both HMC (filled, blue circles) and FHMC with kernel sizes $k = 3, 5, 7$ (open circles, triangles and squares). One can see that the autocorrelations in FHMC are lower than the ones of HMC, and decrease as the kernel size of the CNNs is increased.[5]

In order to study the scaling towards the continuum limit, we plot the ratio of autocorrelation times for HMC versus FHMC in Figure 58 (right) for the three values of the kernel size. Although for the coarser lattices the ratio increases towards the continuum, it seems to saturate within the range of lattice spacings explored, indicating that the cost scaling of both algorithms is the same. The same behaviour is observed for the one-point susceptibility (8.5).

8.6 Continuum limit scaling with $k \sim \xi$

As we take the continuum limit the correlation length increases in lattice units. If the footprint is chosen to scale with ξ, the convolution implemented by the network covers the same physical region. Using our architecture, this can be done by adding more coupling layers or increasing the kernel size, k. More concretely, a kernel size k couples $(k-1)/2$ nearest neighbours; therefore, since we have no hidden layers, if we have N_l coupling layers the network will couple $N_l(k-1)$ nearest neighbours.

In Figure 59 (left) we show again the scaling of the autocorrelation times of the magnetization, but now the networks used for the FHMC algorithm have $N_l(k-1) \approx \xi$. Particularly, all networks of the plot have $N_l = 1$ coupling layers, so only

[5] Note that an accurate cost comparison should include the overhead of computing the force via automatic differentiation. We have not tried to optimize this step and therefore postpone a detailed cost comparison to future work.

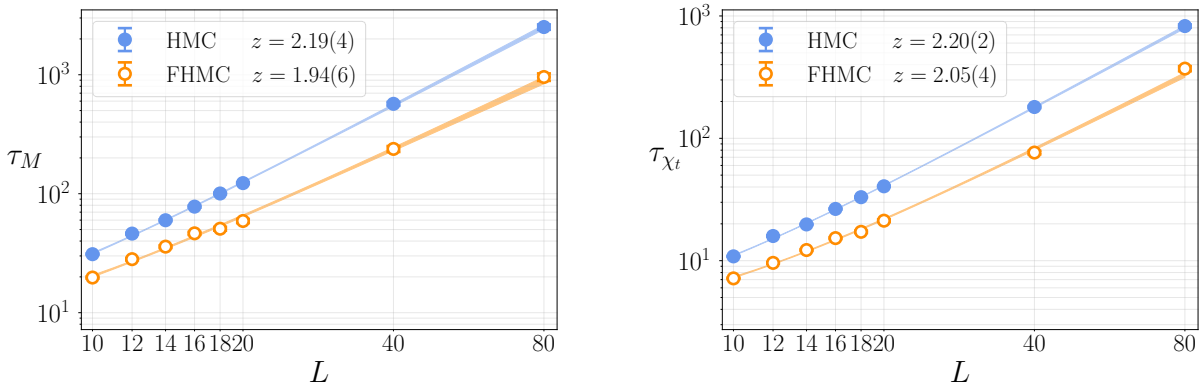


Figure 59: (Left) Scaling of the autocorrelation time of the magnetization for HMC and FHMC with a network with $k \sim \xi$. (Right) Same for the flowed one-point susceptibility. The fits correspond to the best fit function of Equation (8.32).

the kernel size varies: for $L = 10$ to $L = 16$ the networks have $k = 5$; for $L = 18$ and $L = 20$, $k = 7$; for $L = 40$, $k = 11$; and for $L = 80$, $k = 21$.

The curves of the plot correspond to fits to the function

$$\tau = a\xi^z + b, \tag{8.32}$$

yielding as result

$$z_{M,\text{HMC}} = 2.19(4), \quad z_{M,\text{FHMC}} = 1.94(6). \tag{8.33}$$

Thus keeping the physical footprint size constant seems to yield a slight improvement in the scaling towards the continuum.[6] It is also interesting to see that the same happens with the smeared susceptibility in Figure 59 (right). The latter is a non-local observable that has been measured in smeared configurations with a smoothing radius $\sqrt{4t} = \xi$.

This slight improvement is in agreement with the fact that for a fixed network architecture the continuum scaling remains the same as HMC, while increasing the kernel size improves the global factor of the autocorrelations, as was seen in Figure 58.

It is important to note that increasing the kernel size of the network implies increasing the number of parameters in the training, and also the number of operations to compute the force in the molecular dynamics evolution using automatic differentiation. Particularly, the number of parameters of our networks is given by

$$N_{\text{params}} = 4k^2 N_l + 1. \tag{8.34}$$

In Figure 60 we show the time needed to compute the force on a lattice with fixed length $L = 320$ as a function of the number of network parameters (keeping $N_l = 1$ and varying k in the interval $k \in [3, 161]$). Since the computing time seems to scale linearly with the number of parameters, if k scales with the correlation length ξ then there is an additional term proportional to ξ^2 in the FHMC cost.

According to this estimate, FHMC would not reduce the asymptotic simulation cost of a ϕ^4 theory with respect to HMC. However, the implementation of the FHMC force does not require the integration of the flow equation in Equation (8.9), unlike in [46]. Knowing that FHMC already reduces autocorrelation times with minimal architectures, it could probably be used to reduce simulation costs in Lattice QCD with minimal implementation effort.

Finally, reference values from the simulations of HMC and FHMC with $k = 3$ and $N_l = 1$ can be found in Tables 5 and 6, respectively. Autocorrelation times for the unflowed and flowed one-point susceptibilities, χ_0 and $\chi_{0,t}$, are displayed in Figure 61, showing a similar behaviour to the autocorrelation time of the magnetization showed in Figure 58.

[6] As a word of caution, the errors in the results of Equation (8.33) shall not be interpreted as Gaussian; the error of τ_{int} involves a sum over the four-point autocorrelation function, whose computation is usually approximated

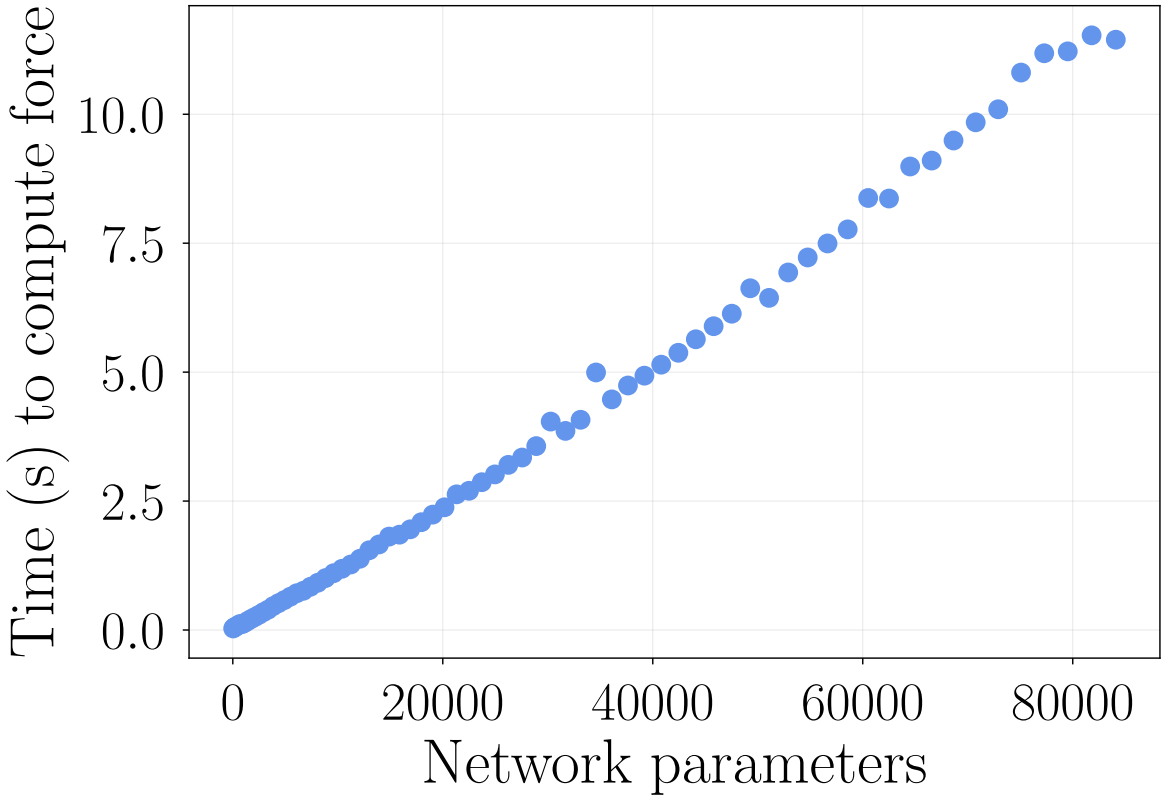


Figure 60: Dependence of the time required to evaluate the force on the number of parameters in the network for a lattice with $L = 320$.

L	τ_M	$\lvert M\rvert$	$\tau_{\lvert M\rvert}$	χ_0	τ_{χ_0}	$\chi_{0,t}$	$\tau_{\chi_{0,t}}$	# confs.	# steps	acc.
6	11.439(94)	0.27545(12)	4.218(22)	0.604082(45)	1.3407(41)	0.168701(79)	3.953(20)	20×10^6	5	0.91
8	19.26(20)	0.26282(15)	7.087(47)	0.636218(46)	2.0910(79)	0.147630(92)	6.696(43)	20×10^6	6	0.91
10	31.03(40)	0.25907(19)	11.518(95)	0.663980(49)	3.284(15)	0.13964(11)	10.845(87)	20×10^6	6	0.88
12	46.21(73)	0.25362(22)	16.89(17)	0.684141(51)	4.583(25)	0.13247(13)	15.90(15)	20×10^6	6	0.86
14	59.8(11)	0.24714(24)	21.09(23)	0.699527(49)	5.367(31)	0.12576(13)	19.78(21)	20×10^6	8	0.91
16	77.9(15)	0.24368(28)	28.36(35)	0.713385(51)	6.930(45)	0.12175(15)	26.48(32)	20×10^6	8	0.89
18	100.4(16)	0.24436(22)	35.02(35)	0.727030(37)	8.493(44)	0.12087(12)	33.03(32)	40×10^6	10	0.92
20	122.8(22)	0.23818(23)	43.30(48)	0.735209(36)	10.011(56)	0.11562(12)	40.55(44)	40×10^6	10	0.91
40	570(21)	0.22642(46)	191.7(42)	0.792768(41)	36.88(38)	0.10217(22)	180.3(39)	40×10^6	14	0.91
80	2518(130)	0.20803(65)	888(29)	0.829350(31)	134.3(18)	0.08642(29)	827(26)	80×10^6	18	0.88

Table 5: HMC results.

L	τ_M	$\lvert M\rvert$	$\tau_{\lvert M\rvert}$	χ_0	τ_{χ_0}	$\chi_{0,t}$	$\tau_{\chi_{0,t}}$	# confs.	# steps	acc.
6	9.69(14)	0.27561(23)	3.686(34)	0.604119(96)	1.5298(95)	0.16877(15)	3.627(34)	5×10^6	8	0.98
8	15.46(28)	0.26278(27)	5.739(65)	0.636246(94)	2.225(16)	0.14760(17)	5.605(63)	5×10^6	8	0.97
10	23.26(50)	0.25913(32)	8.53(12)	0.663968(95)	3.062(26)	0.13964(19)	8.25(11)	5×10^6	8	0.95
12	32.50(82)	0.25402(38)	12.14(20)	0.684240(96)	4.047(39)	0.13271(22)	11.60(18)	5×10^6	8	0.94
14	45.5(13)	0.24735(42)	15.84(29)	0.699595(94)	4.907(52)	0.12593(23)	15.07(27)	5×10^6	10	0.95
16	56.9(18)	0.24376(47)	20.37(41)	0.713427(94)	5.972(69)	0.12176(25)	19.35(38)	5×10^6	10	0.94
18	74.4(27)	0.24472(53)	26.40(60)	0.727035(96)	7.389(95)	0.12098(28)	24.97(56)	5×10^6	10	0.93
20	93.2(38)	0.23807(57)	32.28(81)	0.735258(95)	8.47(12)	0.11566(30)	30.27(74)	5×10^6	10	0.92
40	420(16)	0.22596(48)	143.5(33)	0.792708(43)	28.99(32)	0.10204(23)	134.6(30)	30×10^6	15	0.89
80	1965(165)	0.2077(11)	611(30)	0.829372(51)	97.3(21)	0.08638(46)	563(27)	20×10^6	25	0.87

Table 6: FHMC results with a network with $N_l = 1$ and $k = 3$.

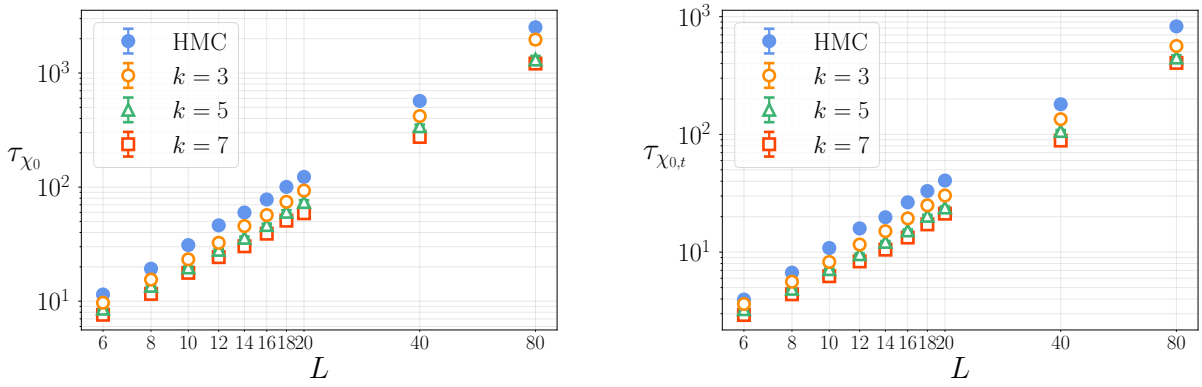


Figure 61: Scaling of the autocorrelation time of the unflowed (left) and flowed (right) one-point susceptibility towards the continuum for HMC (filled, blue circles) and FHMC with kernel sizes $k = 3, 5, 7$ (open circles, triangles and squares).

Schedule

Date	#	Chapter	Tilman	Jan	comments
10/17	1	Basics	⊗		
10/19	1	BTDs & fits	⊗		
10/24	1	NNs & likelihood loss	⊗		
10/26	2	Amplitude regression	⊗		
10/31	2	Integration	H&N	⊗	
11/02	3.1	Classification	H&N	⊗	
11/07	3.2	CNNs	ML4Jets	⊗	
11/09	3.2	CNNs	ML4Jets	⊗	
11/14	3.2	Top tagging	⊗		
11/16	3.3	Point clouds & graphs	⊗		
11/21	3.3	Transformer	⊗		
11/23	3.3	Deep sets & relations	⊗		
11/28	3.4	Symmetries & CLR	⊗		
11/30	4.1-2	Weakly and unsupervised	⊗		
12/05	4.3	More on autoencoders		⊗	
12/07	5.1	Generative & VAE		⊗	
12/12	5.2	GANs	Glühwein-WS	⊗	
12/14	5.3	NF	Glühwein-WS	⊗	
12/19	5.3	NF applications		⊗	
01/09		Diffusion models DDPM	⊗		
01/11		Diffusion models CFM	⊗		
01/16		Unfolding	⊗		
01/18		Generative Unfolding	⊗		
01/23		Physics of NNs		⊗	
01/25		Physics of NNs		⊗	
01/30		Physics of NNs		⊗	
02/01		Physics of NNs		⊗	

References

[1] B. P. Roe, H.-J. Yang, J. Zhu, Y. Liu, I. Stancu, and G. McGregor, *Boosted decision trees, an alternative to artificial neural networks*, Nucl. Instrum. Meth. A **543** (2005) 2-3, 577, arXiv:physics/0408124.

[2] Y. Gal, *Uncertainty in Deep Learning*. PhD thesis, Cambridge, 2016.

[3] G. Kasieczka, M. Luchmann, F. Otterpohl, and T. Plehn, *Per-Object Systematics using Deep-Learned Calibration*, SciPost Phys. **9** (2020) 089, arXiv:2003.11099 [hep-ph].

[4] J. Aylett-Bullock, S. Badger, and R. Moodie, *Optimising simulations for diphoton production at hadron colliders using amplitude neural networks*, JHEP **08** (2021) 066, arXiv:2106.09474 [hep-ph].

[5] S. Badger, A. Butter, M. Luchmann, S. Pitz, and T. Plehn, *Loop Amplitudes from Precision Networks*, SciPost Phys. Core **6** (2023) 034, arXiv:2206.14831 [hep-ph].

[6] D. Maître and R. Santos-Mateos, *Multi-variable integration with a neural network*, JHEP **03** (2023) 221, arXiv:2211.02834 [hep-ph].

[7] G. Kasieczka, T. Plehn, M. Russell, and T. Schell, *Deep-learning Top Taggers or The End of QCD?*, JHEP **05** (2017) 006, arXiv:1701.08784 [hep-ph].

[8] S. Macaluso and D. Shih, *Pulling Out All the Tops with Computer Vision and Deep Learning*, JHEP **10** (2018) 121, arXiv:1803.00107 [hep-ph].

[9] S. Blücher, L. Kades, J. M. Pawlowski, N. Strodthoff, and J. M. Urban, *Towards novel insights in lattice field theory with explainable machine learning*, Phys. Rev. D **101** (2020) 9, 094507, arXiv:2003.01504 [hep-lat].

[10] *Example codes Lattice Field Theories*, 2022.

[11] L. de Oliveira, M. Kagan, L. Mackey, B. Nachman, and A. Schwartzman, *Jet-images — deep learning edition*, JHEP **07** (2016) 069, arXiv:1511.05190 [hep-ph].

[12] A. Butter *et al.*, *The Machine Learning landscape of top taggers*, SciPost Phys. **7** (2019) 014, arXiv:1902.09914 [hep-ph].

[13] L. Benato *et al.*, *Shared Data and Algorithms for Deep Learning in Fundamental Physics*, Comput. Softw. Big Sci. **6** (2022) 1, 9, arXiv:2107.00656 [cs.LG].

[14] A. Butter, G. Kasieczka, T. Plehn, and M. Russell, *Deep-learned Top Tagging with a Lorentz Layer*, SciPost Phys. **5** (2018) 3, 028, arXiv:1707.08966 [hep-ph].

[15] H. Qu and L. Gouskos, *ParticleNet: Jet Tagging via Particle Clouds*, Phys. Rev. D **101** (2020) 5, 056019, arXiv:1902.08570 [hep-ph].

[16] B. M. Dillon, G. Kasieczka, H. Olischlager, T. Plehn, P. Sorrenson, and L. Vogel, *Symmetries, safety, and self-supervision*, SciPost Phys. **12** (2022) 6, 188, arXiv:2108.04253 [hep-ph].

[17] V. Mikuni and F. Canelli, *Point cloud transformers applied to collider physics*, Mach. Learn. Sci. Tech. **2** (2021) 3, 035027, arXiv:2102.05073 [physics.data-an].

[18] P. T. Komiske, E. M. Metodiev, and J. Thaler, *Energy Flow Networks: Deep Sets for Particle Jets*, JHEP **01** (2019) 121, arXiv:1810.05165 [hep-ph].

[19] E. M. Metodiev, B. Nachman, and J. Thaler, *Classification without labels: Learning from mixed samples in high energy physics*, JHEP **10** (2017) 174, arXiv:1708.02949 [hep-ph].

[20] B. M. Dillon, T. Plehn, C. Sauer, and P. Sorrenson, *Better Latent Spaces for Better Autoencoders*, SciPost Phys. **11** (2021) 061, arXiv:2104.08291 [hep-ph].

[21] B. M. Dillon, L. Favaro, T. Plehn, P. Sorrenson, and M. Krämer, *A Normalized Autoencoder for LHC Triggers*, arXiv:2206.14225 [hep-ph].

[22] S. J. Wetzel, *Unsupervised learning of phase transitions: From principal component analysis to variational autoencoders*, Physical Review E **96** (Aug., 2017) .

[23] A. Dawid *et al.*, *Modern applications of machine learning in quantum sciences*, 4, 2022. arXiv:2204.04198 [quant-ph].

[24] J. M. Pawlowski and J. M. Urban, *Reducing Autocorrelation Times in Lattice Simulations with Generative Adversarial Networks*, Mach. Learn. Sci. Tech. **1** (2020) 045011, arXiv:1811.03533 [hep-lat].

[25] T. Plehn, *Lectures on LHC Physics*, Lect. Notes Phys. **844** (2012) 1, arXiv:0910.4182 [hep-ph].

[26] A. Butter, T. Plehn, and R. Winterhalder, *How to GAN LHC Events*, SciPost Phys. **7** (2019) 6, 075, arXiv:1907.03764 [hep-ph].

[27] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, *A review on generative adversarial networks: Algorithms, theory, and applications*, 2020.

[28] M. D. Bernardi, M. Khouzani, and P. Malacaria, *Pseudo-random number generation using generative adversarial networks*, 2018.

[29] R. Anirudh, J. J. Thiagarajan, B. Kailkhura, and T. Bremer, *An unsupervised approach to solving inverse problems using generative adversarial networks*, 2018.

[30] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 [cs.LG].

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, eds., pp. 8024–8035. Curran Associates, Inc., 2019. arXiv:1912.01703 [cs.LG].

[32] L. Ardizzone, J. Kruse, S. Wirkert, D. Rahner, E. W. Pellegrini, R. S. Klessen, L. Maier-Hein, C. Rother, and U. Köthe, *Analyzing Inverse Problems with Invertible Neural Networks*, arXiv:1808.04730 [cs.LG].

[33] M. S. Albergo, D. Boyda, D. C. Hackett, G. Kanwar, K. Cranmer, S. Racanière, D. J. Rezende, and P. E. Shanahan, *Introduction to Normalizing Flows for Lattice Field Theory*, arXiv:2101.08176 [hep-lat].

[34] M. Bellagente, M. Haussmann, M. Luchmann, and T. Plehn, *Understanding Event-Generation Networks via Uncertainties*, SciPost Phys. **13** (2022) 1, 003, arXiv:2104.04543 [hep-ph].

[35] A. Butter, T. Heimel, S. Hummerich, T. Krebs, T. Plehn, A. Rousselot, and S. Vent, *Generative networks for precision enthusiasts*, SciPost Phys. **14** (2023) 4, 078, arXiv:2110.13632 [hep-ph].

[36] A. Butter, N. Huetsch, S. P. Schweitzer, T. Plehn, P. Sorrenson, and J. Spinner, *Jet Diffusion versus JetGPT – Modern Networks for the LHC*, arXiv:2305.10475 [hep-ph].

[37] A. Butter, T. Jezo, M. Klasen, M. Kuschick, S. Palacios Schweitzer, and T. Plehn, *Kicking it Off(-shell) with Direct Diffusion*, arXiv:2311.17175 [hep-ph].

[38] A. Andreassen, P. T. Komiske, E. M. Metodiev, B. Nachman, and J. Thaler, *OmniFold: A Method to Simultaneously Unfold All Observables*, Phys. Rev. Lett. **124** (2020) 18, 182001, arXiv:1911.09107 [hep-ph].

[39] M. Bellagente, A. Butter, G. Kasieczka, T. Plehn, A. Rousselot, R. Winterhalder, L. Ardizzone, and U. Köthe, *Invertible Networks or Partons to Detector and Back Again*, SciPost Phys. **9** (2020) 074, arXiv:2006.06685 [hep-ph].

[40] M. Backes, A. Butter, M. Dunford, and B. Malaescu, *An unfolding method based on conditional Invertible Neural Networks (cINN) using iterative training*, arXiv:2212.08674 [hep-ph].

[41] J. Lindner, D. Dahmen, M. Krämer, and M. Helias, *A theory of data variability in neural network bayesian inference*, 2023.

[42] L. Kades and J. M. Pawlowski, *Discrete Langevin machine: Bridging the gap between thermodynamic and neuromorphic systems*, Phys. Rev. E **101** (2020) 6, 063304, arXiv:1901.05214 [cs.NE].

[43] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[44] M. Helias and D. Dahmen, *Statistical field theory for neural networks*, Lect. Notes Phys. **970** (2020) pp., arXiv:1901.10416 [cond-mat.dis-nn].

[45] M. Luscher, *Trivializing maps, the Wilson flow and the HMC algorithm*, Commun. Math. Phys. **293** (2010) 899, arXiv:0907.5491 [hep-lat].

[46] G. P. Engel and S. Schaefer, *Testing trivializing maps in the Hybrid Monte Carlo algorithm*, Comput. Phys. Commun. **182** (2011) 2107, arXiv:1102.1852 [hep-lat].

[47] L. Del Debbio, G. M. Manca, and E. Vicari, *Critical slowing down of topological modes*, Phys. Lett. B **594** (2004) 315, arXiv:hep-lat/0403001.

[48] D. Albandea, L. Del Debbio, P. Hernández, R. Kenway, J. Marsh Rossney, and A. Ramos, *Learning trivializing flows*, Eur. Phys. J. C **83** (2023) 7, 676, arXiv:2302.08408 [hep-lat].

[49] M. S. Albergo, G. Kanwar, and P. E. Shanahan, *Flow-based generative models for Markov chain Monte Carlo in lattice field theory*, Phys. Rev. D **100** (2019) 3, 034515, arXiv:1904.12072 [hep-lat].

[50] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[51] L. Del Debbio, J. M. Rossney, and M. Wilson, *Efficient modeling of trivializing maps for lattice $\phi 4$ theory using normalizing flows: A first look at scalability*, Phys. Rev. D **104** (2021) 9, 094507, arXiv:2105.12481 [hep-lat].

[52] L. Baulieu and D. Zwanziger, *QCD(4) from a five-dimensional point of view*, Nucl. Phys. B **581** (2000) 604, arXiv:hep-th/9909006.