A Generative-Model approach to path integrals

J. Rozalén Sarmiento

Supervised by: A. Rios

Departament de Física Quàntica i Astrofísica, Universitat de Barcelona (UB), E08028 Barcelona, Spain

11 July 2022

The Feynman path integral formalism is one of the most elegant approaches to Quantum Mechanics, and it provides an alternative and more intuitive manner of understanding the relation between quantum and classical mechanics. Nevertheless, path integrals have the drawback of being utterly difficult to compute, which is why computational methods tackling this issue are in order. In this work we explore the possibilities that Machine Learning has to offer in such computational scenarios. Inspired by the standard Markov-Chain Monte Carlo approach to path integrals, we design a generative neural network that can infer the path distribution from previously generated paths and can also generate new paths equally distributed. Our method has the fundamental advantage over the Markov Chain technique that, once trained, it can sample random paths efficiently and in parallel. The ML model that we employ is not specifically tailored to solve path integrals, and in fact it can be readily embedded in any setting where learning or sampling from a probability density function is needed.

J. Rozalén Sarmiento: jrozalsa7@alumnes.ub.edu

Acknowledgments

I would first like to thank my advisor Dr. Arnau Rios Huguet for the continuous support and inestimable understanding throughout this project. Also, I want to thank Prof. Bruno Juliá Díaz for some useful advice, and my colleague Joan Gómez Micó for the several discussions on Machine Learning simulations, which have been essential for the development of this thesis. Lastly, I would also like to thank my parents and my partner for hearing me out as I explained every new idea I had.

Contents

1 Introduction

Artificial Neural Networks (ANNs) have become powerful computational tools in the physics domain (1; 2). The manifold of techniques used to approach physical problems with Machine Learning (ML) is growing steadily, and many of these techniques find applications in a wide range of problems. Fields like condensed matter (3) and nuclear physics (4; 5; 6; 7) are benefiting from this, and works like the one in Ref. (8) have already shown unrivalled precision in quantum many-body problems, with applications also in quantum chemistry. This last work uses ML techinques within the variational approach to quantum many-body problems (9), which we have used ourselves in past works to solve a nuclear bound state (10; 11). The fundamental idea is to use an ANN as the wave function ansatz in a variational setting, whence minimization of the energy leads directly to the ground-state wave function.

One ML tool that is gaining momentum is that of random variable generation. In fact, there exist ML settings wherein ANNs are used to learn some probability density function (PDF) which is in turn used to efficiently extract samples (12). Some of the methods in this direction have already been used in physical scenarios like lattice field theory (13) and statistical physics (14), but only one work (at least that we are aware of) has used these ideas in a path integral scenario (15), and it is precisely here that the focus of this work is placed in. In Ref. (15), the harmonic oscillator (HO) and the double-well potential problems are solved within the path integral formalism of quantum mechanics using Recurrent Neural Networks (RNNs).

In a similar vein, here we propose a distinct generative approach to tackle the HO. Rather than aiming at learning the analytical path distribution as in Ref. (15), input paths originally sampled according to the analytical distribution (using standard computational methods) are learned by a neural network. To this end, we use Variational AutoEncoders, a special kind of generative ANNs that can extract relevant features from data and then use them to generate new data with these features. Special attention is payed to the sampling efficiency of our method, which is compared to that of the more standard computational approaches to path integrals. In the last section of this work we briefly discuss the original approach using RNNs and a different training objective.

We divide our work into three main Sections: in section 2 we provide an overview of the path integral formalism. In Section 3 we explain the Markov-Chain Monte Carlo approach that is customarily used in these sorts of problems. This is followed by section 4, wherein we introduce a ML method that shows computational advantage with respect to the standard methods. Lastly, in section 5 we shortly discuss a ML approach to solve path integrals from scratch, similar in spirit to that of section 4 but designed for a different initial setting.

2 The path integral formalism

The path integral (PI) formalism was first developed by Richard Feynman around the year 1948, and was published in the paper of Ref. (16). Feynman derived this formalism from scratch, by analogy with the variational formulation of classical mechanics and without using the standard operator formulation of quantum mechanics. It is a posteriori that he showed that the formalism is equivalent to the standard one of quantum mechanics. Interestingly, we now know that the converse is also valid. This is, we can obtain the PI formalism starting from the operator formalism (17; 18).

The PI formulation has some advantages and some drawbacks when compared to the standard formulation of QM. On the one hand, it establishes a connection with classical mechanics: concepts such as trajectories or *paths* are the main pillar of path integrals, and also the least action principle is recovered (with some nuances). This provides us with a precious intuition of QM that we cannot get from the standard formulation.

On the other hand, for the standard, non-relativistic quantum-mechanical problems the PI formulation is known for making calculations innecessarily complicated. Nevertheless, they become indispensable to the formulation of Quantum Field Theory.

We will now briefly introduce the PI formalism for non-relativistic QM as Feynman did it, for the procedure based on operator quantum mechanics is more easily found in books and also some intuition of the connection with classical mechanics is lost on the way. We follow the discussion of Ref. (19).

In classical mechanics we have a rule to determine which of all the possible paths (or trajectories) $x(t)$ is the one that the system will take to go from point a to point b of spacetime. The rule is given by the functional minimisation of the action, $\delta S = 0$. To rephrase this, the action functional $S[x(t)]$ must be stationary at the correct path. In quantum mechanics the rule changes a bit: now all paths contribute to the probability amplitude of going from a to b, not just the path of stationary action (*aka* classical path). All paths contribute equal magnitudes to the total amplitude, but they contribute at different phases, and the phase of contribution is the action S in units of the quantum action, $\hbar$. Mathematically, if the probability of going from $a = (x_a, t_a)$ to $b = (x_b, t_b)$ is denoted by $P(b, a)$, then in terms of an amplitude $K(b, a)$[1] we have $P(b, a) = |K(b, a)|^2$. This amplitude $K(b, a)$ is the sum of the contributions from all the possible paths that start at a and end at b:

$$K(b,a) = \sum_{\text{paths}} \phi[x(t)] = \sum_{\text{paths}} C e^{iS[x(t)]/\hbar}, \tag{1}$$

where C is a constant. Please note that here the paths are classical trajectories, and the action is the one used in classical mechanics. The only thing that is different is the contribution of each path (along with the non-determinism this entails, of course).

Eq. (1) tells us how classical mechanics arises from quantum mechanics at the macroscopic level. In the classical approximation, the values of the physical magnitudes are within such a scale that the action S is huge in front of $\hbar$. If we now take a path $x(t)$ and apply a perturbation to it that is small in the classical scale but big in the units of $\hbar$, we encounter extremely rapid oscillations in the phase, so that all contributions from the varied paths cancel out and add to (nearly) zero. Therefore, whenever the neighbouring paths have different actions their contribution is null; it is only when the action is an extreme

[1] This quantity is denoted K as in *Kernel*, because we know from standard QM that this is the propagator, i.e., the kernel of the wave function evolution in time. It is also the Green's function associated to the Schrödinger equation.

that the change in S when varying the paths is zero (at first order, at least), and therefore all contributions are in phase. In the classical limit we are considering only the path that is right at the extreme, and this suffices. We can see, then, that the classical trajectory is in fact slightly indefinite: as long as the action variation is within the range of $\hbar$, the neighbouring paths make significant contributions to the amplitude, so their presence is not to be ignored. However, the scale of $\hbar$ is so small that the effect at the macroscopic scale is not noticeable.

A natural question arises when looking at Eq. (1): how is this sum to be interpreted? In fact, there is an infinite number of paths, so the sum is not to be taken literally. It seems logical to use integration, but the problem is then that of choosing an adequate measure: not only do we have an infinite number of paths, but also each path has infinite points. We can start to tackle this problem using the same reasoning as for Riemann integrals: we make a partition of the domain and then take a limit. In our case, we partition a single path by partitioning time into $N+1$ points separated by a width ϵ, so that

$$N\epsilon = t_b - t_a \tag{2}$$
$$\epsilon = t_{i+1} - t_i \tag{3}$$
$$t_0 = t_a \qquad t_N = t_b \tag{4}$$
$$x_0 = x_a \qquad x_N = x_b. \tag{5}$$

Now Eq. (1) becomes:

$$K(b,a) = \int \cdots \int \int \phi[x(t)] dx_1 dx_2 \ldots dx_{N-1}. \tag{6}$$

We do not integrate over x_0 or x_N because these points are fixed. Now, to have a more representative sample of paths we could take the limit of $\epsilon \to 0$, but of course this limit does not exist, since the quantity diverges. This suggests that we need a normalizing factor that depends on ϵ, and here we encounter one of the main formal problems of the PI formalism: there is no known general manner to define such a factor. In Riemman integration, the analogue situation would be the one where we have to sum $N-1$ ordinates $f(x_i)$, the difference being that in this case we know that the normalizing factor is just the width of each step (typically denoted h). In contrast, it can be proven that for all Lagrangians of the form

$$L = \frac{m}{2}\dot{x}^2 - V(x,t) \tag{7}$$

the normalizing factor is A^{-N} (19), where

$$A = \left(\frac{2\pi i\hbar\epsilon}{m}\right)^{1/2}. \tag{8}$$

With this factor the limit exists and we have:

$$K(b,a) = \lim_{\epsilon\to 0} \frac{1}{A} \int \cdots \int \int e^{iS[x(t)]/\hbar} \frac{dx_1}{A} \frac{dx_2}{A} \cdots \frac{dx_{N-1}}{A}. \tag{9}$$

There is still one thing unsolved, and it is the precise form of the paths between each pair of points x_i and x_{i+1}. Although this may not seem as a concern at first sight, recall that even the free particle Lagrangian contains a derivative of position. Even worse, if there was an acceleration term we would find that the velocity is discontinuous. As it turns out, we can unite all the $N+1$ points of our paths with straight lines, as shown in Fig. 1. Then, we can estimate the velocity using the standard rules of computational physics, which are

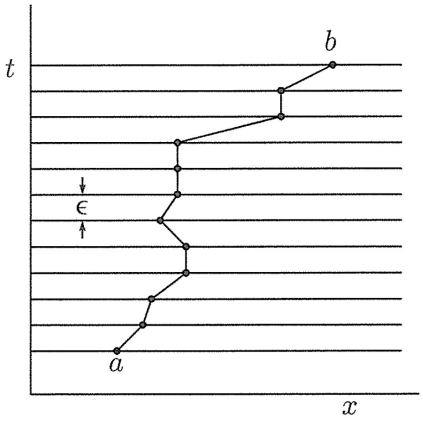

Figure 1: Discretized path.

exact for linear functions. As concerns acceleration, the problem has multiple solutions and, perhaps not so suprisingly, the standard choice is the following:

$$\ddot{x} = \frac{1}{\epsilon^2}(x_{i+1} - 2x_i + x_{i-1}). \tag{10}$$

Nevertheless, there exist some problems where this fix is not adequate, so integration must be redefined. Thus, given that no general procedure is known, the notation used when referring to the summation of Eq. (1) is:

$$K(b,a) = \int_a^b e^{iS[x(t)]/\hbar} \mathcal{D}x(t). \tag{11}$$

This is called a *path integral*. This notation is non-restrictive, in the sense that it is agnostic to the choice of measure and discrete approximation of some quantities.

Path integrals have many exciting properties, but we will not cover any more of those here. Instead, we will end by deriving a useful formula that relates the propagator and the Hamiltonian eigenvectors. In standard QM, the propagator of a particle going from (x_a, t_a) to (x_b, t_b) can be written as:

$$K(x_b, x_a, T) = \langle x_b | e^{-\frac{i}{\hbar}\hat{H}T} | x_a \rangle \,, \tag{12}$$

where we have defined $T \equiv t_b - t_a$. Inserting two identities expressed in the energy eigenvector space in the expression above:

$$\begin{aligned} K(x_b, x_a, T) &= \sum_{n',n} \langle x_b | n' \rangle \langle n' | e^{-\frac{i}{\hbar}\hat{H}T} | n \rangle \langle n | x_a \rangle = \sum_n e^{-\frac{i}{\hbar}E_n T} \langle x_b | n \rangle \langle n | x_a \rangle \\ &= \sum_n e^{-\frac{i}{\hbar}E_n T} \psi_n^*(x_b) \psi_n(x_a). \end{aligned} \tag{13}$$

This is known as the *spectral representation* of the propagator. Although this last exercise is not explicitely related to path integrals, it is one of the few ways in which PI formalism actually appears useful to solve QM problems, and in fact the only reason why we make use of it in our work in the first place; hence its importance here.

3 The standard approach to PI: MCMC

The usual way of computing path integrals is by *Markov-Chain Monte Carlo* or *MCMC* (20; 21; 22). The idea is straightforward: given that a path integral is a high-dimensional integral we can estimate it using the Monte Carlo method. As usual, the properly distributed points can be obtained via the Metropolis algorithm, which makes use of Markov chains. We discuss these ideas in more detail in the next subsections. The concrete choices of parameters are inspired by Ref. (22).

3.1 Constructing the paths

The paths are constructed just as in section 2. The only thing we must bear in mind is that the limit $\epsilon \to 0$ is ignored when computing PIs computationally, and therefore we must choose ϵ small enough in order to have a representative sample of all paths between a and b.[2] For our purposes we have chosen the following parameters:

$$N = 201$$
$$t_a - t_b \equiv T = 100.$$

The step is obviously $\epsilon = T/(N-1)$. We require the initial and final values x_a and x_b to be the same, so that the paths are closed (the reason for this will be clear later in the text). Since we will be generating the paths randomly, a way to do this is by first obtaining $N = 200$ random points, and then adding the last one as $x_N = x_0$. In practice, each path will be a collection of $N = 201$ points, each labelled $x(t_i) \equiv x_i$.

3.2 Approximation of the ground-state density

We want to compute the harmonic oscillator ground-state (GS) density in position space, this is, $|\psi_0(x)|^2$. Since we are using path integrals we start by computing the propagator $K(x', x, T)$. Following Eq. (11), we must start by formulating the Lagrangian:

$$L = \frac{m}{2}\dot{x}^2 - V(x), \tag{14}$$

whence the action is immediately:

$$S(t_b, t_a) = \int_{t_a}^{t_b} L(\dot{x}, x, t)dt. \tag{15}$$

Given that we will be dealing with discretized paths, the Lagrangian and thereby the action will also be discretized. A possible approximation of the action integral for such a (discretized) path $x(t)$ is (23):

$$S[x(t)] \approx \epsilon \sum_{i=0}^{N-1} \left(\frac{m}{2}\dot{x}_i^2 - V(x_i) \right). \tag{16}$$

Now we must specify how the terms in x and $\dot{x}$ are to be interpreted. For our purposes and given that our paths are as that of Fig. 1, it is sufficient to take the following estimates:

$$\dot{x}_i = \frac{x_{i+1} - x_i}{\epsilon}, \quad V(x_i) = \frac{V(x_{i+1}) + V(x_i)}{2}. \tag{17}$$

[2] Sometimes a study is made for decreasingly small ϵ and then some extraploation method is used to simulate the limit $\epsilon \to 0$.

Before continuing any further we need to introduce the concept of *Wick rotation*, which is fundamental for our computational implementation. It consists of the following change of variables: $t \to -i\tau$. The new coordinate τ is usually referred to as *Euclidean time*. In a similar fashion, the action of a path $S[x(\tau)]$ is noted $S_E[x(\tau)]$. Introducing this substitution into Eq. (16), we find:

$$S_E[x(\tau)] \approx i\epsilon \sum_{i=0}^{N-1} \left(\frac{m}{2}\dot{x}_i^2 + V(x_i) \right), \tag{18}$$

where the dot now indicates derivation w.r.t. τ. Also, the PI formula now reads:

$$K(b,a) = \int_a^b e^{-S_E[x(t)]/\hbar} \mathcal{D}x(\tau). \tag{19}$$

We now want access to the GS wave function. To get there, we start at Eq. (13). Then, to obtain the wave function at a point x we can take the propagator of a particle starting and ending at this same point. Indeed,

$$K(x,x,T) = \sum_n e^{-E_n T/\hbar} |\psi_n(x)|^2, \tag{20}$$

where we have maintained the notation for the time interval. Now we can do the following trick: the GS density is, by definition, the state with the lowest energy (E_0), and therefore the term that contributes the most to the sum. In order to further isolate this term we would like the quantity $|T(E_1 - E_0)|$ to be big, $|T(E_1 - E_0)| \gg 1$ (E_1 is the energy of the first excited state). Formally, we can achive this by taking the limit

$$\lim_{T\to\infty} K(x,x,T) = e^{-E_0 T/\hbar} |\psi_0(x)|^2. \tag{21}$$

Eq. (21) holds under the assumption that $E_0 \ll E_1$. Computationally we need not be so drastic with this limit; we might as well pick a value for T that makes the first excited state contribution very small in front of the GS contribution, just as we do in subsection 3.1. Eq. (21) then tells us how to access the particle density via PI. Also, the reason for enforcing $x_N = x_0$ on all paths is now clear: the wave function is related to the diagonal part of the propagator, $K(x,x,T)$, which only involves paths which fulfil $x_N = x_0$.

3.3 Monte Carlo estimation of path integrals

Now that we know how to obtain $|\psi(x)|^2$ we have to address the path integral computationally. The usual approach to doing so is as follows: once the step size ϵ and the Lagrangian (and therefore the measure) are fixed, a PI is just a multi-dimensional integral, and considering the large number of dimensions, the Monte Carlo method presents itself as the best available option.

We need to be careful as concerns the points (paths) used to evaluate the integral, nonetheless, for the integrand is a non-trivial functional of the paths and can therefore present big variations over big time intervals T. For this reason we must obtain paths that are distributed in high accordance with the distribution suggested by the integrand itself. The functional $e^{-S_E[x(\tau)]}$ suggests that the paths that contribute the most are those with minimum Euclidean action, and we can formalize this idea by constructing the following probability density functional:

$$p_E[x(\tau)] = \frac{e^{-S_E[x(\tau)]}}{\int e^{-S_E[x(\tau)]} \mathcal{D}x(\tau)} \equiv \frac{e^{-S_E[x(\tau)]}}{Z}. \tag{22}$$

Then, the Metropolis algorithm can be trivially used to obtain the paths distributed according to $p_E[x(\tau)]$ (a brief description of this algorithm can be found in Appendix A). Note that, while we do not know how to compute Z, we do not need it for the Metropolis algorithm to work. Nevertheless, to obtain the particle density we can do the following trick: given that $|\psi(x)|^2$ is the probability of finding a particle between $x-\Delta x$ and $x+\Delta x$ and that we know all the (most representative) paths that the particle can take, we can estimate $|\psi(x)|^2$ with a histogram by keeping track of the "time" that a particle spends in a given interval, on average. This is,

$$|\psi(x)|^2 = \frac{1}{\Delta x}\sum_i \frac{\theta(\Delta x - |x_i - x|)}{M}, \tag{23}$$

where M is the total number of paths that we have generated, θ is the Heaviside function and Δx is the bin size of the histogram. Note that this allows us to compute the whole $|\psi(x)|^2$ within a single Metropolis sweep, because as opposed to what Eqs. (11) and (21) would suggest, we need not compute a PI for every x.

3.4 Results

We have computed the ground-state density of a single particle in a 1D HO using the ideas in the subsections above. This is, we have used the Lagrangian

$$L = \frac{m}{2}\dot{x}^2 - \frac{m}{2}w^2x^2 \tag{24}$$

with $m = w = 1$, we have generated $M = 10^4$ paths like those of subsection 3.1 via Metropolis-Hastings and we have then substituted $\Delta x \to \epsilon$ in Eq. (23) to obtain the particle density.

The bottom left panel of Fig. 2 shows a typical particle path obtained via Metropolis. As one can see, the path presents rapid oscillations in time. This figure is also intended to serve as a visual explanation of Eq. (23): the bottom left panel displays vertical lines which represent the different intervals of size Δx. By counting the number of times that the path crosses each of these intervals, storing each number in its corresponding interval and projecting these intervals to the plot above we get the wave function in position space. The rightmost panel shows the evolution of the action as more paths are accepted. Note that here we show only the first $2 \cdot 10^3$ accepted paths.

One possible way to quantify the quality of the histogram fit to the exact GS is to compute the *Kullback-Leibler divergence* or KL divergence, D_{KL}, between the two distributions. The KL divergence can be used as a measure of closeness between two PDFs. Despite this, it is not symmetric and this implies that the two possible orderings of the inputs entail two different meanings: the first argument can be thought of as a reference PDF, and the second argument is an approximation of the reference PDF (see Appendix B for more information on the KL divergence). Therefore, if we call the MCMC GS $q \equiv |\psi_0(x)|^2_{\text{MCMC}}$ and the exact distribution $p \equiv |\psi_0(x)|^2$, the meaningful quantity in our case is $D_{KL}(p||q)$. For reference, the HO GS wave function is given by the following expression:

$$\psi_0(x) = \left(\frac{1}{\pi}\right)^{\frac{1}{4}} e^{-\frac{x^2}{2}}, \tag{25}$$

where we have used $m = \hbar = w = 1$, w being the frequency of the oscillator. These units will be used throughout this text.

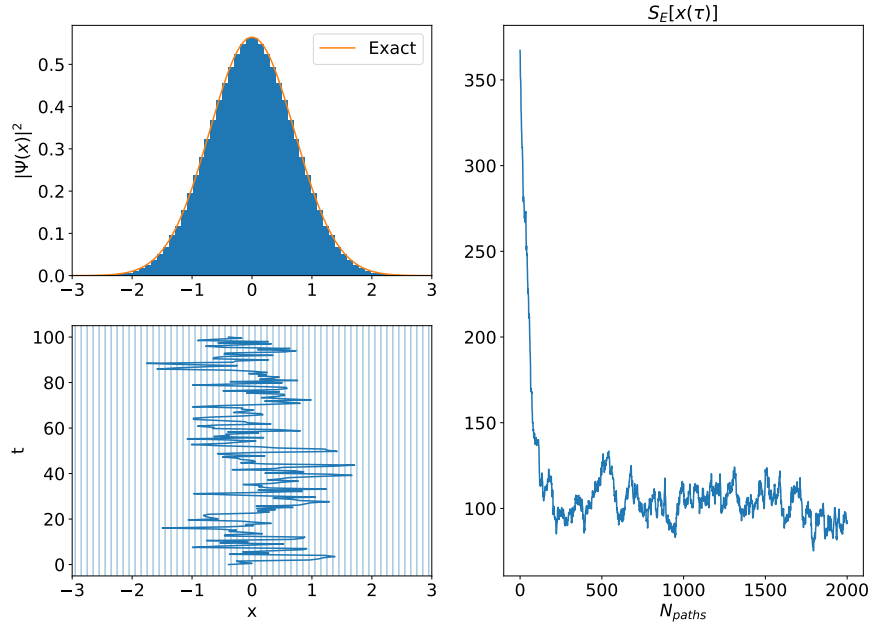

Figure 2: Right panel: evolution of the Euclidean action with the number of paths. Top left panel: histogram of the paths generated with Metropolis (blue) compared to the exact GS density (solid orange). Bottom left panel: a typical particle path in spacetime.

	$M = 400$	$M = 600$	$M = 800$	$M = 1200$	$M = 4000$	$M = 10000$
$D_{KL}(p\|\|q)$	0.0379	0.0293	0.0297	0.0092	0.0030	0.0028

Table 1: KL divergence between the MCMC approximation to the GS density, q, and the exact solution p computed for different number of generated paths. All the numbers have an associated estimation error of 10^{-4} associated to the Simpson method for integral approximation (see main text).

Table 1 shows values of $D_{KL}(p||q)$ computed with different numbers of generated paths. All the values have been computed using the Simpson integration method in 1D. The tendency is clear: the more paths we use to compute the GS density, the better the approximation, which is exactly what one would expect given that the paths are used as points in a Monte Carlo estimation.

4 Sampling paths with a Variational AutoEncoder

In section 3 we have shown how we can obtain the ground-state density of a system with an arbitrary Lagrangian using MCMC. While this method is very general and well-established, the Markov Chain used by the Metropolis algorithm is a big computational bottleneck, mainly due to its sequential nature. For every new path $x^{(i)}(t)$ at iteration i that we might generate, we need the path in the previous iteration $x^{(i-1)}(t)$, and this requires the sampling process to be sequential and therefore, slow. Nevertheless, Machine Learning methods provide us with a setting in which to overcome this very issue, where the sampling process is fully parallelizable. For a quick introduction to ANN training see Appendix C.

In this section we introduce a simple procedure showing how this can be achieved. Then, we apply this procedure to solve the HO in the path integral formalism, just as in section 3, and we compare the GS density thusly obtained against that obtained with MCMC (see Fig. 2). Lastly, we compare the computation time of both methods.

4.1 Parametrizing probability distributions with ANNs

When faced with the problem of sampling from a known PDF $p(\mathbf{x})$, the Metropolis algorithm provides us with a distribution-agnostic procedure to obtain samples distributed according to $p(\mathbf{x})$. However, when $p(\mathbf{x})$ is simple there exist methods for sampling which are faster and more precise than Metropolis. For example, sampling from a uniform or normal distribution is trivial, and most programming languages have built-in methods to sample from these distributions (24; 25). The prime reason why we need Metropolis is because hard problems usually entail hard PDFs for which a simple method does not exist.

A first attempt at overcoming this issue could consist of using parametric approximators of PDFs. For example, Gaussian Mixture Models (GMMs) are known to be universal PDF approximators (26; 27). In other words, there exists a set of parameters with which our GMM is arbitrarily close to the target $p(\mathbf{x})$. A GMM has the following form:

$$q(\mathbf{x}) = \sum_{j=1}^{N_c} \gamma_j \mathcal{N}(\mathbf{x}; \mu_j, \sigma_j^2 \mathbb{I}), \tag{26}$$

where $\gamma_j > 0 \, \forall j$, $\sum_{j=1}^{N_c} \gamma_j = 1$ and $\mathcal{N}$ is the (multivariate) normal distribution. The problem is now reduced to finding $\{\gamma_j, \mu_j, \sigma_j^2\}_{j=1}^{N_c}$, and ANNs provide a natural setting to do this: we can let the GMM parameters be the output of an ANN, and then train the network so that our GMM approximation is close to the target PDF, $p(\mathbf{x})$. In the simplest case we could use single-layer, feed-forward networks to parametrize $\{\gamma_j, \mu_j, \sigma_j^2\}_{j=1}^{N_c}$, and our approximator in Eq. (26) would then read:

$$\begin{aligned} q_\phi(\mathbf{x}) &= \sum_{j=0}^{N_c} \gamma_j \mathcal{N}(\mathbf{x}; \mu_j(\phi, \mathbf{x}), \sigma_j^2 \mathbb{I}(\phi, \mathbf{x})), \text{ where} \\ \gamma &= \text{softmax}(\mathbf{W_4 h} + \mathbf{b_4}) \\ \mu &= \mathbf{W_3 h} + \mathbf{b_3} \\ \sigma^2 &= \exp(\mathbf{W_2 h} + \mathbf{b_2}) \\ h &= \tanh(\mathbf{W_1 x} + \mathbf{b_1}), \end{aligned} \tag{27}$$

and $\phi \equiv \{\mathbf{W_i}, \mathbf{b_i}\}_{i=1}^4$. Here, the softmax makes sure that $\gamma_j > 0 \, \forall j$ and $\sum_{j=1}^{N_c} \gamma_j = 1$, and the exponential is there to enforce positivity. Once the network is trained, since our approximator is made of gaussians, we can sample in a fast and parallel manner.

In terms of physical calculations, there are two evident scenarios where we can apply this methodology. Firstly, any calculation that entails sampling from a hard function can benefit from this approach. In path integrals, for example, we can model the target path distribution, Eq. (22), with an ANN in the aforementioned manner and then train the network to minimise some measure of closeness to the target PDF, such as the Kullback-Leibler divergence. This idea is further discussed in section 5. The second scenario is one where we do not know the exact PDF, but we have some data distributed according to it. For these kind of settings there exist ANN architectures which can learn the distribution of the given data and then generate new data with that same distribution. In this section we will focus on the latter scenario.

4.2 Variational AutoEncoders

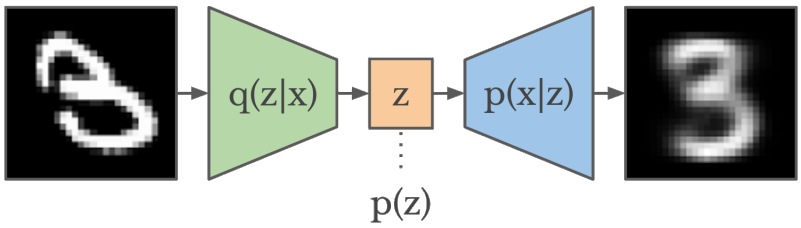

Figure 3: General form of a VAE. Here, a 2D image with the number 3 is passed as an input to the VAE, and the VAE generates a distinct image that also contains the number 3.

Variational AutoEncoders or VAEs are a kind of neural network architecture designed to generate new data that resembles the one that it is fed with during the training process (28). To do this, VAEs work under the assumption that the observable data are generated by some random process involving hidden random variables (also *latent* variables) $\mathbf{z}$ from a parametric family $p_\theta(\mathbf{z})$, so that once these variables are known we can generate new samples of our observable quantity by sampling from the prior PDF $p_\theta(\mathbf{x}|\mathbf{z})$. Intuitively, the latent space must contain information about the dataset, which we can access with the (generally intractable) posterior PDF $p(\mathbf{z}|\mathbf{x})$. In order to approximate $p(\mathbf{z}|\mathbf{x})$ a parametric recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ is used. In coding theory terms, we are *encoding* a sample $\mathbf{x}$ into a latent variable $\mathbf{z}$, and then *decoding* $\mathbf{z}$ to get (a similar version of) $\mathbf{x}$ back, and therefore $q_\phi(\mathbf{z}|\mathbf{x})$ is called *encoder* and $p_\theta(\mathbf{x}|\mathbf{z})$, *decoder*; note that distinct sets of parameters, ϕ and θ, are used for the encoder and the decoder respectively. Fig. 3 illustrates this idea: a 2D image with a number 3 is passed to a VAE, which encodes the information in the latent space, and from this latent representation it generates a similar image.

The learning process of a VAE is aimed at maximizing the marginal likelihood of the dataset $\mathbf{X} = \{\mathbf{x}^{(i)}\}_{i=1}^N$, consisting of i.i.d. random variables. We shall refer to the marginal likelihood of $\mathbf{X}$ as $p_\theta(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)})$.[3] In order to achieve this we use ANNs to parametrize the encoder and the decoder. Then, a lower bound on $\log p_\theta(\mathbf{x}^{(i)})$ can be immediately derived and written as follows (28):

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = -D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})]. \tag{28}$$

Eq. (28) is commonly referred to as the *Evidence Lower Bound* or *ELBO*. The (negative) second term of the r.h.s of Eq. (28) can be thought of as the reconstruction error, this

[3] We are using the same notation for the parameters and the functions of both the PDF of $\mathbf{X}$, $p_\theta(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)})$, and the PDFs of each individual sample $\mathbf{x}^{(i)}$, $p_\theta(\mathbf{x}^{(i)})$. This is because the samples are i.i.d., and therefore the PDF of the joint distribution is the product of the individual PDFs.

is, the error made by attempting to recover $\mathbf{x}^{(i)}$ after having encoded it. The (negative) first term is usually called regularizing term, because it measures the "distance" between the encoder distribution and the actual prior distribution of $\mathbf{z}$ and thus tries to make q_ϕ similar to p_θ. To maximize the ELBO we must jointly learn the parameters θ and ϕ. Once the VAE is trained, we can generate new samples by sampling latent variables, now from the prior distribution $p_\theta(\mathbf{z})$ and using only the decoder. Since the decoder has worked in tandem with the encoder during the traninig, it knows how the important features are encoded or, in other words, where to find them in latent space; this shall prove useful for computing the marginal likelihood with a trained VAE.

The usual approach to computing the ELBO is via Monte Carlo estimation. In fact, Eq. (28) can be trivially rewritten as follows:

$$\mathcal{L}(\theta, \phi, \mathbf{x}^{(i)}) = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log \frac{p_\theta(\mathbf{x}^{(i)}|\mathbf{z})p_\theta(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\right]. \tag{29}$$

The expression above is Monte-Carlo-ready, and the sampling from $q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$ can be made simple with an appropiate choice of the encoder network. In fact, the usual trick here is to use the ANN to parametrize a PDF, as explained in subsection 4.1. For example, in the VAE original paper (see Ref. (28)) a single-layer feed-forward neural network is used to parametrize an isotropic multivariate normal distribution, which we can think of as having $N_c = 1$ in Eq. (26), and this is the case for both the encoder and the decoder. Explicitely, the decoder reads:

$$\begin{aligned} &\log p_\theta(\mathbf{x}|\mathbf{z}) = \log \mathcal{N}(\mathbf{x}; \mu, \boldsymbol{\sigma^2}\mathbb{I}), \text{ where} \\ &\mu = \mathbf{W_3 h} + \mathbf{b_3} \\ &\log \boldsymbol{\sigma^2} = \mathbf{W_2 h} + \mathbf{b_2} \\ &h = \tanh\left(\mathbf{W_1 z} + \mathbf{b_1}\right). \end{aligned} \tag{30}$$

The encoder network can be obtained by swapping $\mathbf{x}$ and $\mathbf{z}$ in Eq. (30). Here, θ makes reference to the set of ANN parameters, $\theta = \{\mathbf{W_1}, \mathbf{b_1}, \mathbf{W_2}, \mathbf{b_2}, \mathbf{W_3}, \mathbf{b_3}\}$. The prior distribution over $\mathbf{z}$ is taken as $p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbb{I})$; note that in this case it does not depend on θ. We shall employ this same parametrization in our experiments.

VAEs are used for a wide range of tasks, such as the generation of human faces, text, speech, handwritten digits, and so on. One popular (and perhaps too enthusiastic) interpretation of the learning process of VAEs is the following: during training, the network forms an abstract internal representation of the given data, just as humans learn abstract ideas. Following this reasoning, this is why we can ask a neural network trained in the MNIST dataset (29), as in Fig. 4, to generate a "number 3" and it generates something which we can interpret as a "3": it has "learned the concept of 3". Thus, in a PI setting like ours, the network can learn the distribution of paths for the PI that is inherent in the training set. However, even in the best possible scenario we cannot know what the learned features will be beforehand.

4.3 Experiments with a simple VAE

We now test the VAE architecture by computing PIs. The idea here is to feed the VAE with (a subset of) the path manifold already generated with MCMC, have it learn the relevant features and then sample new paths in parallel. Please bear in mind that we do not control what the learned features will be, so even though the path distribution is a distinguished feature of the training set we cannot be certain that the encoder-decoder tandem will grasp this. Henceforward, we shall denote the particle paths by $\mathbf{x}$.

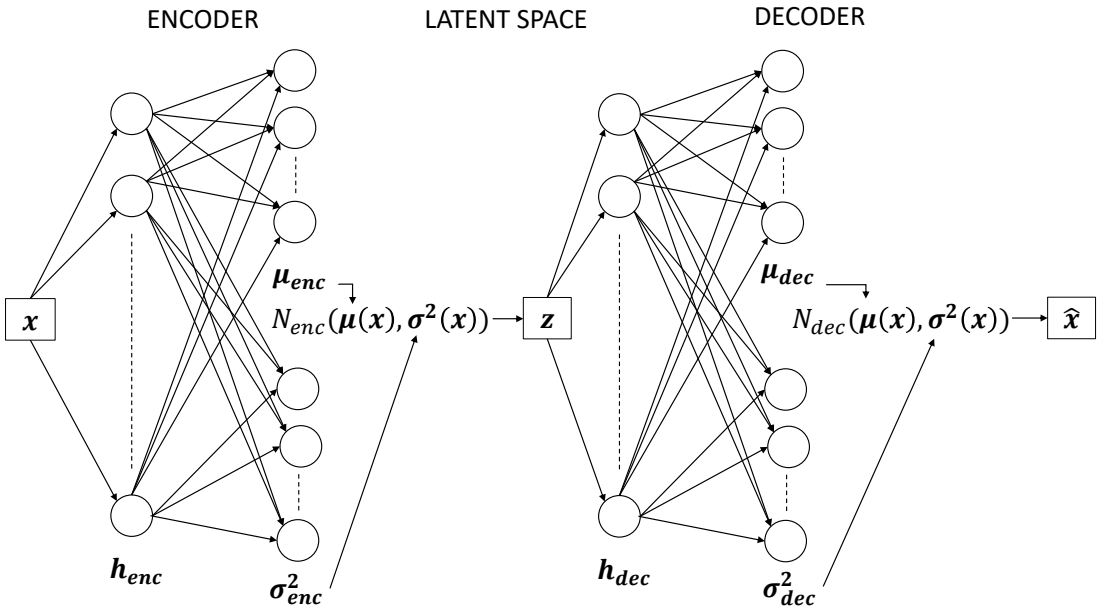

Figure 4: Architecture of the VAE used in our experiments. The encoder network (left) consists of three hidden layers: one with shared parameters and the other two compute μ_{enc} and σ^2_{enc} separately. The decoder network (right) has the same structure. The input ($\mathbf{x}$), latent variable ($\mathbf{z}$) and output ($\hat{\mathbf{x}}$) quantities, although also multi-dimensional, are shown inside black boxes and not as a layer of nodes to stress that they are not the immediate output of a neural network.

The architecture of the encoder and decoder networks is the one mentioned in subsection 4.1 and is depicted in Fig. 4. Fig. 4 shows the architecture of the VAE that we have used in our experiments. In the leftmost side we find three fully-connected layers: one with shared parameters, one with parameters for μ and one with parameters for σ^2. Following the flow of the arrows, μ, σ^2 are used to parametrize a multivariate normal distribution (the encoder PDF) from which we sample a latent variable $\mathbf{z}$. From here, the rightmost side of the figure is identical to the leftmost side, swapping $\mathbf{x}$ and $\mathbf{z}$. In the end we use the decoder PDF to sample a new path $\hat{\mathbf{x}}$. As concerns the dimensions of the network, the layers that connect $\mathbf{x}(\mathbf{z})$ to $h_{\mathbf{enc}}(h_{\mathbf{dec}})$ have $h = 15$ hidden nodes each. The layers that connect $h_{\mathbf{enc}}(h_{\mathbf{dec}})$ to $\mu_{\mathbf{enc}}(\mu_{\mathbf{dec}})$ and $\sigma_{\mathbf{enc}}(\sigma_{\mathbf{dec}})$ have $h = 3$ each. Note that this implies that the dimension of the latent space is also $s = 3$; the importance of having a low-dimensional latent space lies in that it prevents the VAE from learning the identity. For example, if the latent space had the same dimensionality as the input, the network would then be a bijection and it could learn to make exact copies of the input. Finally, the input and output variables have dimension $N = 20$, i.e., equal to the number of time steps (length of each path). In total, this network has 1111 parameters.

The training consists of taking a fixed subset of $M = 2{\cdot}10^3$ paths from a manifold of 10^4 paths of $N = 20$ points generated with MCMC as the training set, and then passing these to the VAE. At each iteration (epoch) we use stochastic gradient descent techniques (SGD), which provide faster convergence and more importantly, allow the model to fit in our GPU. Broadly, SGD consists in approximating the gradient of the loss over the entire training set, $\nabla_{\theta,\phi}\mathcal{L} = \nabla_{\theta,\phi}\sum_{i=1}^{M}\mathcal{L}_i$, by the gradient of the loss over a smaller subset (or *batch*) of the training set of size b, $\nabla_{\theta,\phi}\sum_{i=1}^{b}\mathcal{L}_i$; the specific batch changes at every iteration. This approximation is intended to entail faster convergence, and also to simply reduce the number of examples that we pass to the network at once. In fact, large models are usually

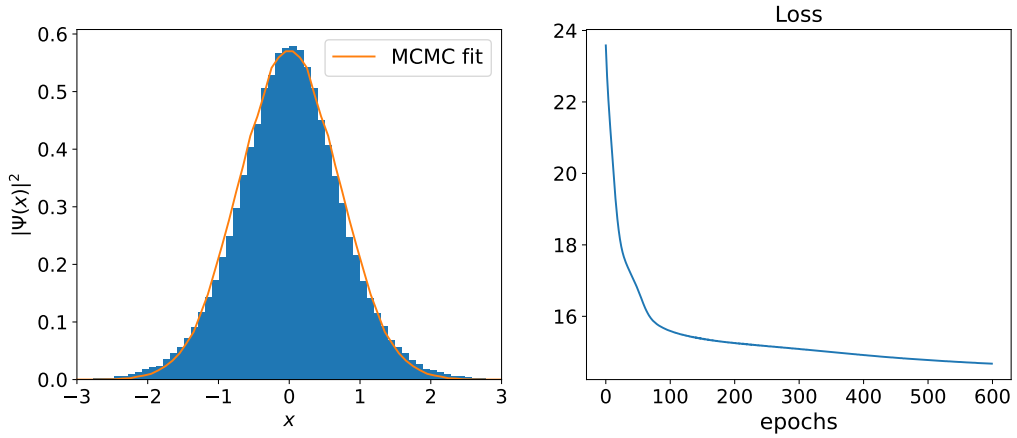

Figure 5: Left panel: histogram of the paths generated with a trained VAE (blue histogram) compared to the MCMC GS (solid orange line). The hyperparameters used for this VAE are the ones mentioned in the main text. Right panel: evolution of the loss function (ELBO) with the number of training epochs.

impossible to fit in a regular computer RAM (or VRAM). In our implementation, we take mini-batches of 150 paths without repetition, and after each batch we update the weights using the Adam optimizer with a learning rate of 0.001 (30). The expectation over the encoder distribution in Eq. (29) requires $\sim 10^3$ samples to yield a reasonably accurate estimate. A typical training is 600 epochs long; here, by epoch we mean a complete iteration through the dataset, which happens every M/batch-size updates of the weights. The code implementation of this project uses the PyTorch library and can be found in Ref. (31).

After the training is complete we can generate new paths by sampling with the decoder, and we do this by sampling latent variables $\mathbf{z}$ from the prior $p(\mathbf{z})$ and then passing them to the decoder. It is at this point where the sampling advantage of this method is evinced: the $\mathbf{z}$ variables can now be sampled in a parallel fashion, and therefore so can the paths. In order to test the accuracy of the generated paths we use them to compute the MCMC ground-state density, in the same vein as in section 3. This is shown in Fig. 5.

Fig. 5 illustrates the GS density computed from the VAE-generated paths and the reference GS density (left panel). Note that the meaningful comparison here is with the density computed with MCMC, since the VAE obtains all the information from the MCMC-generated paths. The right panel shows the loss evolution of the model used to compute the histogram in the left panel. This is a typical mini-batch ELBO minimization, i.e., most of the progress is done in the first few epochs and from there the evolution is rather slow. In fact, this is a known feature of SGD techniques: approximating the full gradient with batches allows for a faster initial convergence. Intuitively, we can think that, when far from the minimum (initially) we are more likely to jump to a point with lower loss, even if the gradient used is not informed of the entired dataset.

We now try to quantify these results. In a similar fashion as in subsection 3.4 we take the KL divergence between the VAE and the MCMC solution as an indicator of the quality of our model. By computing the KL divergence with different hyperparameters we can also determine whether our model is truly functional or not. As a matter of fact, in a functional model one would expect to find a certain kind of correlations between the hyperparameter values and the test metric, and this is precisely what we find. This is explained next.

Table 2 shows values of the KL divergence between the VAE and the MCMC results for the GS density, referred to as $\mathfrak{q}$ and $\mathfrak{p}$ respectively. Different values of the KL divergence

	$h=1$	$h=2$	$h=3$	$h=7$	$h=9$	$h=50$
$D_{KL}(\mathfrak{p}\|\|\mathfrak{q})$	0.0101	0.0086	0.0095	0.0018	0.0020	0.0009

Table 2: KL divergence between the VAE approximation to the GS density, q, and the MCMC solution p shown for different number of encoder/decoder hidden nodes. All the numbers have an associated estimation error of 10^{-4} associated to the Simpson method for integral approximation.

	$M=10$	$M=50$	$M=100$	$M=500$	$M=3000$	$M=4000$
$D_{KL}(\mathfrak{p}\|\|\mathfrak{q})$	0.0925	0.0935	0.0552	0.0043	0.0011	0.0039

Table 3: Similar to Table 2, but here we vary the size of the training set.

correspond to different numbers of nodes in the linear layers h of both the encoder and the decoder (see Fig. 4). We observe the trend that, the more hidden nodes, the lower the value of D_{KL}; this tendency lasts as far as $h \sim 100$, where the D_{KL} values stabilize around 0.0002. This is precisely what one should expect in any ML setting: a model with very few hidden nodes does not possess the same learning capability as one with more nodes, and thereby it performs worsely. Of course, this argument does not hold for high numbers of nodes. If the number of parameters is excessive, the model can become prone to overfitting. In VAEs this reasoning is further conditioned by the latent space dimension, where the same argument can in principle be set forth.

Besides the number of hidden nodes we have also explored the size of the training set and the latent space dimension; the results of these exploration are shown in Tables 3 and 4 respectively. Regarding Table 3, we find that, as M increases, D_{KL} decreases. This behaviour does not extend for all sizes: we can see that from $M = 500$, the KL divergence values stabilize and present small fluctuations, whence we interpret that the model needs at least this many paths to extract significant information out of them. As concerns Table 4 we find that the KL divergence does not seem notably affected by changes in s, even for the lowest values. Although some of these dimensions might seem too low, we believe that either the network has found a compact manner to store information about each path, or the fact that the paths are 1-dimensional and have $N = 20$ points demands no more than such a number of dimensions.

Let us now return to our initial goal of modelling the path density. Apart from plotting a histogram of the VAE-generated paths, we can try to estimate the marginal likelihood $p(\mathbf{x})$ to then compare it to the exact distribution.[4] A first approach to computing this quantity would look as follows:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z} = \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[p(\mathbf{x}|\mathbf{z})\right]. \tag{31}$$

Although this expression is technically correct, in practice the distribution $p(\mathbf{x}|\mathbf{z})$ does not have structure in the same region of the latent space as $p(\mathbf{z})$, and this makes the estimation more difficult. Nevertheless, we can use the encoder distribution in an importance sampling scheme, for the encoder's job is precisely to store information about the inputs in concrete regions of the latent space, and it is in those regions where the decoder has the most structure. A better attempt at computing $p(\mathbf{x})$ is therefore:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})\frac{q(\mathbf{z}|\mathbf{x})}{q(\mathbf{z}|\mathbf{x})}d\mathbf{z} = \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] \tag{32}$$

[4]These PDFs are not to be confused with the ones compared in Table 2: $p(\mathbf{x})$ is the probability density of path $\mathbf{x}$, whereas $\mathfrak{p}(x)$ in Table 2 is the squared modulus of a wave function (the GS density).

	$s=1$	$s=3$	$s=5$	$s=10$	$s=14$	$s=20$
$D_{KL}(\mathfrak{p}\|\|\mathfrak{q})$	0.0016	0.0086	0.0045	0.0054	0.0053	0.0041

Table 4: Similar to Table 2, but here we vary the dimension of the latent space.

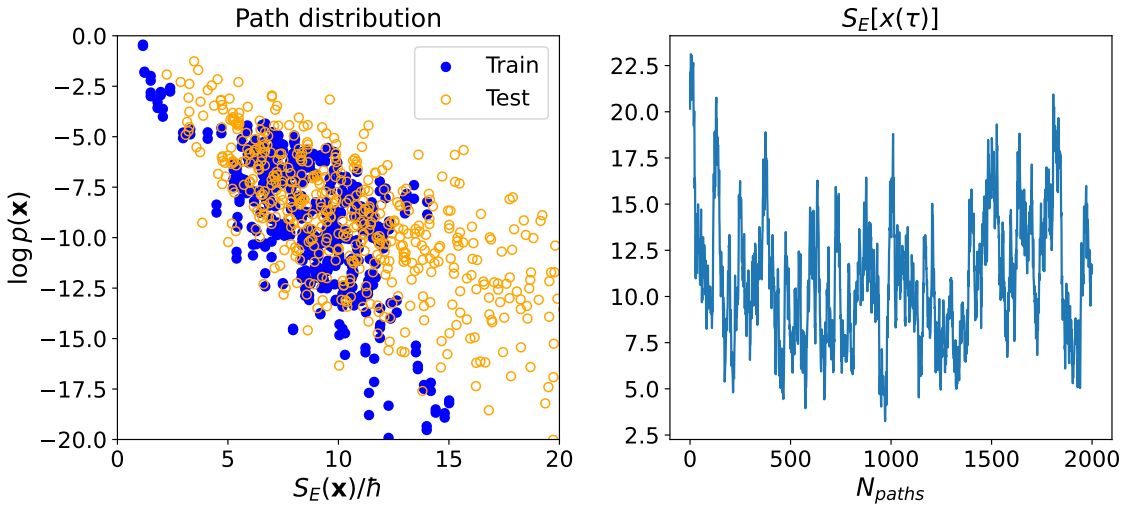

Figure 6: Left panel: each circle represents a path; the filled blue circles represent paths generated with MCMC, whereas the hollow orange circles represent paths sampled by the VAE. In both cases, the probability of each path (vertical axis) is the one learned by the VAE. Right panel: evolution of the action in a MCMC setting where paths with $N = 20$ points are generated. The relation between both panels is explained in the main text.

Equating the marginal $p(\mathbf{x})$ with the exact distribution $p_E(\mathbf{x}) = \exp(-S_E(\mathbf{x})/\hbar)/Z$ and taking logarithms in both sides provides a way to visualize the distance of the VAE-generated paths to $p_E(\mathbf{x})$, and this is shown in Fig. 6.

Fig. 6 shows (left panel) the VAE-learned path distribution for the training set (filled blue circles) and for paths generated with the VAE itself (hollow orange circles). The right panel shows the evolution of the action as a function of the number of accepted paths in a MCMC setting where paths with $N = 20$ points are sampled. In order to judge whether the path distribution in the left panel is good or not we need to look at the right panel first and pay attention to the range of actions within we can find almost all the paths. Then, if we look back at the left panel we realize that the VAE assigns (almost) the highest probabilities to the paths in that same range of actions. Similarly, even though the VAE generates paths that take a wider range of actions, we still find a larger density of paths in that range of actions. The fact that paths with larger actions are also generated is probably because the action is not the only feature that the VAE identifies as "important". Besides, we know that the "ideal" paths would lie on a line with slope -1, and we can see in Fig. 6 how such a line dictates, roughly, the tendency of the plotted paths. We take all of this as a sign that the network has identified the action, and therefore the distribution, as an important property of the paths and that it has learned the distribution to a good extent.

Throughout this section we have explained how to use a generative model to sample data in parallel, but at this point, a quantitative comparison with the standard Metropolis sampling is in order. To this end, we employ both MCMC and VAE to generate different numbers of paths and, during the process, we measure the time it takes for each method to complete the calculation. This is illustrated in Fig. 7.

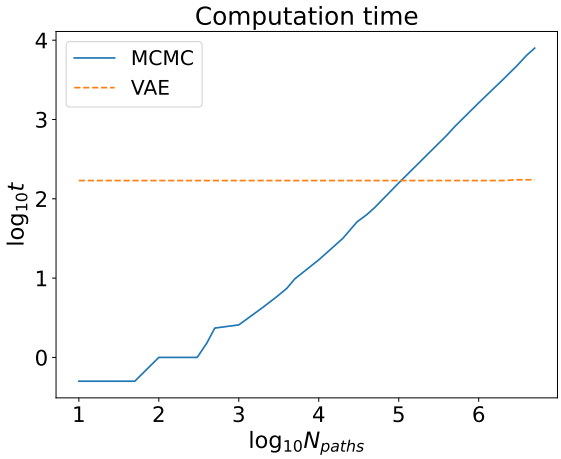

Figure 7: Computation time to generate different numbers of paths with the MCMC algorithm (solid blue) and our VAE method (dashed orange). Note the base 10 logarithm in the axis magnitudes.

Fig. 7 shows the time it takes for both MCMC (solid blue) and our VAE (dashed orange) to sample different numbers of paths. These are total CPU (or GPU) times, and therefore the offsets represent the computation time previous to the generation of the first path. The small offset of the MCMC curve is due to the burn-in period of the Metropolis algorithm, whereas the comparatively bigger offset of the VAE line is due to the training of the network. As one would expect, the time scales linearly with the number of paths for MCMC (the acceptance rate rapidly stabilizes after a few iterations). Contrary to this, the VAE can generate up to 10^6 paths without noticing any difference in computation time, since the sampling occurs in parallel in the GPU. The main overhead in the Machine Learning approach, however, is the training time.

Please note that the VAE training time is greatly dependant on the hyperparameters. For example, the VAE offset in Fig. 7 corresponds to a 170 seconds-long traning, but varying the hyperparameters can potentially help reduce this number. Reducing the training set size, the number of hidden nodes, the number of samples used to evaluate the expectation of Eq. (29) and increasing the learning rate are some examples of techniques we can use to speed up the computation. The key point of our technique, however, is that once the training is done the path generation is immediate, almost independently of the number of paths we wish to generate (we have tested up to 10^6 paths). We believe that the potential of this technique is big. For instance, going back to Fig. 7 we can perform an approximate calculation of the time it would take to generate 10^6 paths from scracth: first we need to generate paths with MCMC, and we have seen how $\sim 10^3$ paths suffices to train a network, so it would take $\sim 10^{0.5}$ sec. We now pass these paths to the VAE, which takes $\sim 10^2$ secs. to train; from here, generating 10^6 paths is immediate, so the total time is of the order 10^2. Thus, even though the VAE might have required MCMC first, it still manages to beat it by a factor of 10^2.

This is not even the kind of settings where VAEs shine the most. In fact, any scenario where the information we have beforehand is not a PDF, like in PIs, but rather data distributed in a certain manner, would not be able to benefit from methods like MCMC; this is the case of most experimental settings. Our method, however, would be perfect for such situations, and we would not even need to generate any data first.

5 Sampling from a generative RNN

In section 4 we have shown how a VAE can be used to infer the path distribution from a set of MCMC-generated paths and then generate new paths equally distributed. Here we introduce a method that also aims at learning the path distribution and generating new paths efficiently, but with the initial assumption that we know the analytical path distribution and do not have (necessarily) previous examples of such paths. This section is based on the work of Ref. (15).

The idea is similar to that of the VAE of section 4. Roughly, the fundamental changes are: (1) we dispose of the encoder network, (2) we set the loss function as the KL divergence between the decoder and the analytical PDF and (3) instead of using a feed-forward ANN as the decoder, we use a RNN. We like to think that (2) makes up for (1). Besides, (3) is related to the fact that we want to process paths, which we can think of as sequential data with long-term correlations. In fact, a RNN is a kind of ANN architecture specifically designed to process sequential data (more information about RNNs can be found in appendix D).

This method has the advantage that, as a result of the training process we obtain an estimation of the partition function Z. Indeed, the loss function can be written as follows:

$$\mathcal{L} = \mathbb{E}_{\mathbf{x}\sim q_\phi}\left\{\hbar^{-1}S_E(\mathbf{x}) + \ln q_\phi(\mathbf{x})\right\} + \ln Z, \tag{33}$$

where Z is the partition function and the other variables have the same meaning as in previous sections of the text. Despite the fact that Z appears in the loss function, since it does not depend on the network parameters we can disregard it when computing the loss. By the end of the training, when the loss has reached its minimum value we have direct access to (an estimation of) Z.

There are still some other implementation differences in the approach of Ref. (15). For example, the parametrized PDF of the decoder is a GMM with $N_c > 1$ instead of an isotropic multivariate gaussian distribution. Furthermore, it is not raw RNNs which are used, but a more sophisticated version of these called *Long-Short Term Memory* or LSTM. Broadly, LSTMs have the advantage over vanilla RNNs that they can track long-term dependencies more easily thanks to a technique to avoid vanishing gradients (for more information on LSTMs see Ref. (32)). In the original work, this method is used to solve both the HO and a system with a double-well potential. The results obtained are within reasonable accordance to the exact solutions and, in the HO case, they are similar to ours. For instance, Fig. 3 of Ref. (15) is similar to Fig. 6 of this work; the dissimilarities of the action values stem from the number of time steps of each path.

We have tried this approach ourselves: we have carried out all the coding implementation and we have tried to understand all the moving parts in the explanation of Ref. (15), but at the time of writing this, we have not been quite able to accurately reproduce the original results.

6 Conclusions and future outlook

In this work we explore path integrals from the computational point of view. We explain the standard approach to these problems, this is, using Markov-Chain Monte Carlo to generate paths distributed according to the analytical distribution that appears in the particle propagator, and we use the HO as the test bed for this method. We seize this setting to obtain the ground-state density of the HO. The results obtained, although not novel, are very accurate and they evince the powerfulness of the MCMC method.

We also pinpoint the fundamental performance issue of Markov Chain methods, and we propose a way to overcome this limitation. To this end, we make use of VAEs to learn the path distribution from a given set of MCMC-generated paths and we show how, once the model is trained, generating new paths with the VAE comes at almost no cost. We find that our model is robust and needs no specially tailored architecture to process and extract information from paths. We take this as an indication that these ideas are easily transferable to any probabilistic scenario. Future work might entail applying this ideas to solve other problems. For one, a well-known approach to solving quantum many-body problems is to use a wave function ansatz along with MCMC to iteratively compute the energy of the system (33; 34). Here, we could embed our VAE in the following manner: in the first iteration, after having generated well-distributed points with Metropolis, we train the VAE on these points and, in the subsequent iterations, we replace the Metropolis process with training the VAE on the new points. Another use case for our method might be combining it with *transfer learning*, that is, having a network which has been trained in one task learn a similar one. In physics this might mean first training a network to solve a HO and use this solution as the starting point for a more complicated potential. The hope is that the network will have learned relevant physical properties during the first training and that the second calculation will be able to benefit from that "knowledge".

In the last section we mention a distinct ML technique that can be used to tackle path integral problems. The fundamental idea stems from Ref. (15), and in our work we give an overview of the basic ML methods employed. Even though we have carried out the computational implementation ourselves, at the moment of writing this, valid results are still missing.

Overall, this experience has provided us with a wider background in ML techniques for solving problems in quantum mechanics and, additionally, with some useful ideas of physical calculations in which our VAE approach can be used.

References

[1] Pankaj Mehta M B, Wang C H, Day A G, Richardson C, Fisher C K and Schwab D J 2019 *Physics Reports* **810** 1–124 ISSN 0370-1573 URL https://www.sciencedirect.com/science/article/pii/S0370157319300766

[2] Carleo G, Cirac I, Cranmer K, Daudet L, Schuld M, Tishby N, Vogt-Maranto L and Zdeborová L 2019 *Rev. Mod. Phys.* **91**(4) 045002 URL https://link.aps.org/doi/10.1103/RevModPhys.91.045002

[3] Carleo G and Troyer M 2017 *Science* **355** 602–606 (*Preprint* https://www.science.org/doi/pdf/10.1126/science.aag2302) URL https://www.science.org/doi/abs/10.1126/science.aag2302

[4] Utama R, Piekarewicz J and Prosper H B 2016 *Phys. Rev. C* **93**(1) 014311 URL https://link.aps.org/doi/10.1103/PhysRevC.93.014311

[5] Lasseri R D, Regnier D, Ebran J P and Penon A 2020 *Phys. Rev. Lett.* **124**(16) 162502 URL https://link.aps.org/doi/10.1103/PhysRevLett.124.162502

[6] Niu Z M, Liang H Z, Sun B H, Long W H and Niu Y F 2019 *Phys. Rev. C* **99**(6) 064307 URL https://link.aps.org/doi/10.1103/PhysRevC.99.064307

[7] Wang Z A, Pei J, Liu Y and Qiang Y 2019 *Phys. Rev. Lett.* **123**(12) 122501 URL https://link.aps.org/doi/10.1103/PhysRevLett.123.122501

[8] Pfau D, Spencer J S, Matthews A G D G and Foulkes W M C 2020 *Phys. Rev. Research* **2**(3) 033429 URL https://link.aps.org/doi/10.1103/PhysRevResearch.2.033429

[9] Saito H 2018 *Journal of the Physical Society of Japan* **87** 074002 (*Preprint* arxiv:1804.06521) URL https://doi.org/10.7566/JPSJ.87.074002

[10] Keeble J and Rios A 2020 *Physics Letters B* **809** 135743 URL https://doi.org/10.1016%2Fj.physletb.2020.135743

[11] Rozalén Sarmiento J, Keeble J W T and Rios A 2022 Machine learning the deuteron: new architectures and uncertainty quantification (*Preprint* arxiv:2205.12795) URL https://arxiv.org/abs/2205.12795

[12] Rezende D J and Mohamed S 2015 Variational inference with normalizing flows URL https://arxiv.org/abs/1505.05770

[13] Medvidovic M, Carrasquilla J, Hayward L E and Kulchytskyy B 2020 Generative models for sampling of lattice field theories (*Preprint* arxiv:2021.01442) URL https://arxiv.org/abs/2012.01442

[14] Wu D, Wang L and Zhang P 2019 *Phys. Rev. Lett.* **122**(8) 080602 URL https://link.aps.org/doi/10.1103/PhysRevLett.122.080602

[15] Che Y, Gneiting C and Nori F 2022 *Phys. Rev. B* **105**(21) 214205 URL https://link.aps.org/doi/10.1103/PhysRevB.105.214205

[16] Feynman R P 1948 *Rev. Mod. Phys.* **20**(2) 367–387 URL https://link.aps.org/doi/10.1103/RevModPhys.20.367

[17] Strickland M 2019 *Relativistic Quantum Field Theory, Volume 2* 2053-2571 (Morgan & Claypool Publishers) ISBN 978-1-64327-708-0 URL https://dx.doi.org/10.1088/2053-2571/ab3108

[18] MacKenzie R 2000 Path integral methods and applications URL https://arxiv.org/abs/quant-ph/0004090

[19] Feynman R P 1965 *Quantum Mechanics and Path Integrals* (McGraw-Hill)

[20] Sauer T 2001 The Feynman path goes Monte Carlo *Fluctuating Paths and Fields* (World Scientific) pp 29–42 URL https://doi.org/10.1142%2F9789812811240_0003

[21] Mittal S, Westbroek M J E, King P R and Vvedensky D D 2020 *European Journal of Physics* **41** 055401 URL https://doi.org/10.1088/1361-6404/ab9a66
[22] Elebiary Loren D 2021 Path integral simulation of the harmonic and anharmonic oscillators URL http://hdl.handle.net/2445/176475
[23] Creutz M and Freedman B 1981 *Annals of Physics* **132** 427–462 ISSN 0003-4916 URL https://www.sciencedirect.com/science/article/pii/0003491681900749
[24] Pytorch docs https://pytorch.org/docs/stable/distributions.html#torch.distributions.distribution.Distribution accessed: 2022-07-11
[25] Murphy K P *Probabilistic Machine Learning: Advanced Topics* (MIT Press)
[26] Calcaterra C and Boldt A 2008 Approximating with gaussians (*Preprint* arxiv:0805.3795) URL https://arxiv.org/abs/0805.3795
[27] Plataniotis K N and Hatzinakos D 2001 *Gaussian Mixtures and Their Applications to Signal Processing* (CRC Press)
[28] Kingma D P and Welling M 2013 Auto-encoding variational bayes (*Preprint* arxiv:1312.6114) URL https://arxiv.org/abs/1312.6114
[29] Deng L 2012 *IEEE Signal Processing Magazine* **29** 141–142
[30] Kingma D P and Ba J 2014 Adam: A method for stochastic optimization URL https://arxiv.org/abs/1412.6980
[31] Project github repository https://github.com/javier-rozalen/vfpg.git accessed: 2022-07-11
[32] Sak H, Senior A and Beaufays F 2014 Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition URL https://arxiv.org/abs/1402.1128
[33] Pang T 2016 *An Introduction to Quantum Monte Carlo Methods* 2053-2571 (Morgan Claypool Publishers) ISBN 978-1-6817-4109-3 URL https://dx.doi.org/10.1088/978-1-6817-4109-3
[34] Foulkes W M C, Mitas L, Needs R J and Rajagopal G 2001 *Rev. Mod. Phys.* **73**(1) 33–83 URL https://link.aps.org/doi/10.1103/RevModPhys.73.33
[35] Koonin S and Meredith D 1990 *Computational Physics* (CRC Press) URL https://doi.org/10.1201/9780429494024
[36] Cybenko G 1989 *Math. of Control, Signals, and Systems (MCSS)* **2** ISSN 0932-4194 URL http://dx.doi.org/10.1007/BF02551274

A Metropolis algorithm

The Metropolis algorithm (also Metropolis-Hastings) is a procedure used to obtain points that are distributed according to some known probability density function (PDF). Algorithm 1 shows the steps of this method for generating m points of n dimensions distributed according to a PDF ρ. For a more exhaustive explanation and formal proof of the algorithm see Ref. (35).

Algorithm 1 Metropolis-Hastings

Set $n_{\text{accepted}} = 0$.
Choose an initial random path $\mathbf{x_0}$.
while $n_{\text{accepted}} \leq m$ **do**
  Generate a random vector ξ in the m-dimensional space.
  Compute $\mathbf{y} = \mathbf{x_n} + \delta\xi$ ▷ In the first step, $\mathbf{x_n} = \mathbf{x_0}$
  Compute $r = \rho(\mathbf{y})/\rho(\mathbf{x_n})$
  if $r \geq 1$ **then**
    $\mathbf{x_{n+1}} \leftarrow y$
    $n_{\text{accepted}} \leftarrow n_{\text{accepted}} + 1$
  else if $r \leq 1$ **then**
    Generate $p \in U(0, 1)$
    if $r > p$ **then**
      $\mathbf{x_{n+1}} \leftarrow \mathbf{y}$
      $n_{\text{accepted}} \leftarrow n_{\text{accepted}} + 1$
    else if $r \leq p$ **then**
      $\mathbf{x_{n+1}} \leftarrow \mathbf{x_n}$

B Kullback-Leibler divergence

The Kullback-Leibler divergence D_{KL} (also *relative entropy*) is a divergence in the sense of information theory. In the 1D case it is defined as:

$$D_{KL}(p||q) = \int_{\mathbb{R}} p(x) \log \frac{p(x)}{q(x)} dx, \tag{34}$$

where p and q are PDFs and the logarithm can be taken in any basis, depending on the desired units of information. This quantity has some interesting properties. First, the KL divergence is non-negative, $D_{KL}(p||q) \geq 0$. Second, it serves as a measure of the shared information of p and q, and when $D_{KL}(p||q) = 0$ we say that p and q carry the same information. Sometimes this quantity is referred to as a distance for the sake of simplicity, although it is not a distance in the topological sense (it is not symmetric with respect to its arguments). In fact, swapping the arguments gives different meanings to the overall quantity: the first argument is taken as the truth or *null* hypothesis, whereas the second argument is the alternative hypothesis. For example, in the main text (see subsection 3.4) we pass the analytical distribution as the first argument.

In statistical settings it is common to use the KL divergence as a measure of closeness among functions instead of, say, Mean Squared Error (MSE). This has to do not only with its probabilistic/statistical meaning and suitability for dealing with PDFs, as we have already explained, but also because the magnitudes of PDFs can get very small, and taking logarithms ensures computational stability.

C Artificial Neural Network training

Artificial Neural Networks (ANNs) are a special kind of parametric functions that have the property of being universal approximators of any continuous function (36), i.e., there exists a set of parameters such that the ANN is arbitrarily close to the target function. *Training* an ANN consists of finding these parameters, or at least a set which is good enough. This is achieved by first choosing a *loss function*, i.e., some differentiable function that measures the quality of the ANN predictions, and then using numerical optimization techniques to find the parameters that optimize this loss.

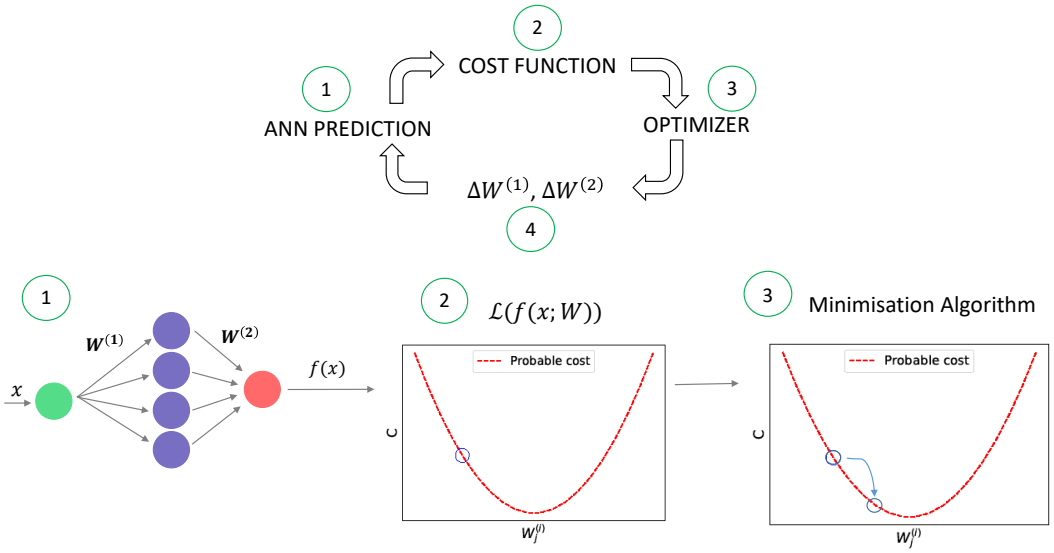

Figure 8: ANN training scheme. Top: flux diagram which indicates the ordered iterative steps of the training process. Bottom: illustrations that represents steps 1 to 3 of the diagram on top.

Fig. 8 illustrates this training process. In step 1 we have an ANN with arbitrary parameters whose outputs have no meaning. We then compute the loss (or cost) $\mathcal{L}$ of these predictions (step 2), which we in turn pass to a minimisation algorithm (or *optimizer*) (step 3). The optimizer then suggests how to change the ANN parameters so that, at the next iteration, the loss of the ANN predictions will be lower than that at the current iteration. This process is repeated until convergence of the loss function is observed. The "learning" in "Machine Learning" refers to this very process.

D Recurrent Neural Networks

In this section of the Appendix we expose the general idea behind RNNs, which are the basis behind LSTMs, the kind of networks used Ref. (15).

Recurrent Neural Networks or *RNNs* are neural networks whose architecture is specifically tailored to learn sequences of data. In these tasks they stand out against other architectures like feed-forward neural networks because they can:

- Take in variable-lenght input.
- Track long-term dependencies (of elements in a sequence).
- Maintain information about order (of elements in a sequence).
- Share parameters accross the sequence.

The architecture of a generic RNN is depicted in Fig. 9. When the network receives an input vector of length N, $\mathbf{x}$, it recursively applies the same transformation f, parametrized by a set of weights and biases W, to all elements of $\mathbf{x}$ until the last element has been inputted to the net. The output of this transformation at each time t (that is, when the input is x_t) is the *internal cell state*, h_t. The input of f_W at time t is not only x_t but also h_{t-1}, and since this is the case at all time steps, each internal cell state is conditioned on all previous internal states. Typically, f_W is taken as a nonlinear function of an affine transformation of the inputs, as in feed-forward ANNs.

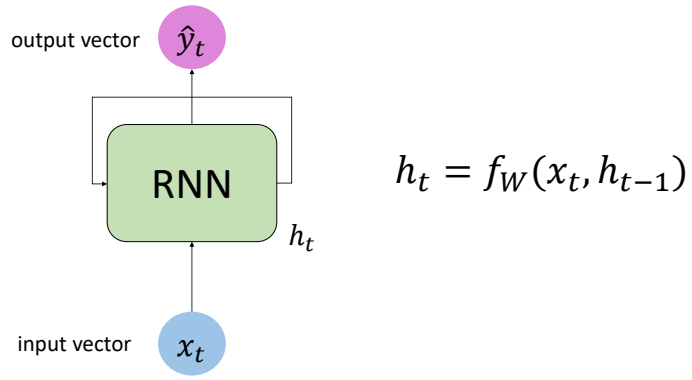

Figure 9: "Rolled" RNN. The parameters W are the same at every time step.

Note that the output of an RNN does not have its dimension fixed by anything. For example, when doing next word prediction we would only compute $\hat{y}_t$ (usually just a linear transformation of h_t) in the last step, but in our work we sample a position coordinate at each time t, whence we obtain an N-dimensional output.

These networks were not used in this form for too long because of a problem that arises due to the architecture itself. In the most general setting where the network output is computed from multiple $\hat{y}_t$, the prediction error is not only backpropagated through each individual $\hat{y}_t$, but also across all time steps[5], and it is not difficult to see how this can lead to exploding and/or vanishing gradients. This often translates into the changes in the net parameters not reaching the first "layers". The solution to this problem was brought by LSTMs (32).

[5]This is known as *Backpropagation Through Time* or BPTT.