Using recurrent neural networks to optimize dynamical decoupling for quantum memory

Moritz August[*]
Department of Informatics, Technical University of Munich, 85748 Garching, Germany

Xiaotong Ni[†]
Max-Planck Institute for Quantum Optics, 85748 Garching, Germany

(Received 23 September 2016; published 27 January 2017)

We utilize machine learning models that are based on recurrent neural networks to optimize dynamical decoupling (DD) sequences. Dynamical decoupling is a relatively simple technique for suppressing the errors in quantum memory for certain noise models. In numerical simulations, we show that with minimum use of prior knowledge and starting from random sequences, the models are able to improve over time and eventually output DD sequences with performance better than that of the well known DD families. Furthermore, our algorithm is easy to implement in experiments to find solutions tailored to the specific hardware, as it treats the figure of merit as a black box.

DOI: 10.1103/PhysRevA.95.012335

I. INTRODUCTION

A major challenge of quantum information processing (e.g., quantum computation and communication) is to preserve the coherence of quantum states. While in principle we can build a fault-tolerant quantum memory or universal quantum computer once the error rate of the device is below a certain threshold, it is still beyond nowadays experimental capacity to build a decent-size quantum computer. One less explored area is the optimization of implementing a fault-tolerant protocol on a concrete experimental setting. This is often a tedious problem, due to the amount of details in the real devices and the fact that the architectures of both experimental devices and theoretical protocols are still rapidly changing. Thus, an attractive approach is to automatize this optimization task. Apart from convenience, it is conceivable that with less human intuition imposed, the upper bound of the performance will be higher. This has previously been proven to be true in fields such as computer vision where artificial neural network (ANN) models that try to solve tasks without using handcrafted representations of data have overtaken approaches based on human insight in tasks such as image classification and object recognition [1]. Another interesting recent example is the ability of ANNs to learn how to play games on a human or even superhuman level without any or just little prior knowledge about the respective games [2,3].

Automatically optimizing parameters in real (or numerical simulations of) experiments is not a new idea. For example, it has been applied to optimizing the pulse shape of a laser, the parameters of Hamiltonians to achieve certain unitary operations, or the parameters of dynamical decoupling and cold-atom experiments. Most works that attempt to obtain optimal parameters use genetic algorithms [4–8] or (to some degree [9]) local searches such as gradient descent [7,10–14] and the Nelder-Mead simplex method [15–17]. It is argued that by using these optimization methods directly on the experiments, we can avoid the difficulty of modeling the imperfect control and the system-environment interaction. However, one possible weakness of these optimization methods is that they generate new trials only by looking at a fixed number of previous ones and often they need to restart once they reach a local minimum. Thus, in the long run, they do not fully utilize all the data generated by the experiments.

In this work we propose an orthogonal approach, where we try to mimic the structure of good parameters by building a model that approximates the probability distribution of these parameters. After an initial optimization, this model can then be used to efficiently generate new possible trials and can be continuously updated based on new data. In particular, based on the problem we attempt to solve, we choose this model to be a variant of the recurrent neural network (RNN), which makes our approach very similar to the way in which natural languages or handwriting are currently modeled. This ansatz enables us to exploit the models and insights developed by the machine learning community and possibly translate further progress there into the field of quantum control. It is worth pointing out that the machine learning part of this work is purely classical; only the (classical) data are related to quantum time evolution. Among the previous works, the approach by Wigley *et al.* in [18] is the most similar to ours, as they attempt to build a model from the data and utilize the model to perform optimization. Classical machine learning is also used in [19–21] to characterize the error models in quantum error correction and to react accordingly.

To demonstrate the feasibility of using our method to help optimize quantum memory, we consider the problem of automatically learning and optimizing dynamical decoupling sequences (almost) without using any prior knowledge. Dynamical decoupling (DD) [22] is a technique that combats certain noise by applying a sequence of unitary operations on the system (see [23,24] for a review). It has a less stringent requirement compared to general error correction protocols, which allows it to be demonstrated in experiments [15,23,25] in contrast to other methods. Moreover, known classes of good DD sequences have a relatively simple and well-defined structure. Based on the assumption that this holds true also for yet unknown and possibly better classes of sequences, it is

[*]august@in.tum.de
[†]xiaotong.ni@mpq.mpg.de

conceivable that a learning algorithm could eventually sample them without the need of using heavy mathematics.

To clarify, we do *not* attempt to solve the following two questions.

First, what do RNNs try to learn? It is known that RNNs can incorporate both short- and long-range correlation, which is desirable in our case, but it is unclear which one the gradient training method prioritizes. Indeed, it is an ongoing study to understand the behavior of RNNs [26]. Nevertheless, we choose to use RNNs since there are heuristic arguments on the advantage of them compared to similar models [27] and they benefit greatly from modern machine learning libraries and hardware.

Second, what is the optimal machine learning algorithm to find the best DD sequences? It is clear that we cannot claim our algorithm is the best one as there is not much theoretical understanding on RNNs. Indeed, the present authors believe there is much room for improvement, possibly by using better heuristics or taking into account more prior knowledge of DD. However, our work demonstrates that with a general model and a small amount of human effort, we can already achieve nontrivial results for certain problems.

II. BACKGROUND

A. Dynamical decoupling

The majority of dynamical decoupling schemes are designed for error models where the system-environment interaction can be described by a Hamiltonian. We will use $\mathcal{H}_S$ and $\mathcal{H}_B$ to denote the Hilbert space of the system and environment (often called bath), respectively. The difference between system and environment is that the former represents the part of the Hilbert space we can apply the Hamiltonian on and in which we store quantum information. The total noise Hamiltonian is

$$H_0 = H_S \otimes I_B + I_S \otimes H_B + H_{SB}.$$

Without intervention, in general H_0 would eventually destroy the quantum states we store in $\mathcal{H}_S$. To suppress this noise, we could apply a time-dependent Hamiltonian $H_C(t)$ to the system, which makes the total Hamiltonian $H(t) = H_0 + H_C(t)$. In the ideal case, we can control $H_C(t)$ perfectly and reach very high strength (i.e., norm of the Hamiltonian), which allows the ideal pulse

$$V(t) = O\delta(t - t_0).$$

It applies a unitary operator e^{-iO} to the system for an infinitely small duration (we set $\hbar = 1$ in this work). A very simple DD scheme for a qubit (a two-level system S) is the XY_4 sequence: It applies pulses of the Pauli matrices X and Y alternatingly with equal time interval τ_d in between. A complete cycle consists of four pulses $XYXY$, thus the total time period of a cycle is $T_c = 4\tau_d$. In the limit of $\tau_d \to 0$, the qubit can be stored for an arbitrarily long time. The intuition behind DD sequences is the average Hamiltonian theory. Let $U_C(t) = \mathcal{T}\exp\{-i\int_0^t dt' H_C(t')\}$ be the total unitary applied by $H_C(t')$ up to time t. In the interaction picture defined by $U_C(t)$, the dynamics is governed by the Hamiltonian $\tilde{H}(t) = U_C^\dagger(t) H_0 U_C(t)$. If the time interval τ_d between pulses is much smaller than the time scale defined by the norm of $\|H_0\|$, it is reasonable to consider the average of $\tilde{H}(t)$ within a cycle. The zeroth-order average Hamiltonian in T_c (with respect to τ_d) is

$$\bar{H}^{(0)} = \frac{1}{T_c}\int_0^{T_c} dt' U_C^\dagger(t) H_0 U_C(t).$$

For the XY_4 sequences introduced above, it is easy to compute $\bar{H}^{(0)} = \frac{1}{4}\sum_{\sigma\in\{I,X,Y,Z\}} \sigma H_0 \sigma$. Since the mapping $O \to \sum_{\sigma\in\{I,X,Y,Z\}} \sigma O \sigma$ maps any 2×2 matrix to 0, by linearity we know $\bar{H}^{(0)} = 0$.

Here we are going to list several classes of DD sequences. We will first explain how to concatenate two sequences, as most long DD sequences are constructed in this manner. Given two DD sequences $A = P_1 \cdots P_m$ and $B = Q_1 \cdots Q_n$, the concatenated DD sequence $A[B]$ is

$$A[B] = (P_1 Q_1) Q_2 \cdots Q_n (P_2 Q_1) Q_2 \cdots Q_n \\ \cdots (P_m Q_1) Q_2 \cdots Q_n.$$

As an example, when we concatenate the length-2 and length-4 sequences XX and $XYXY$, we obtain $IYXYIYXY$. For convenience, we will use CDD to denote these sequences. Note that originally CDD is used to denote sequences generated solely from recursively concatenating $XYXY$ with itself.

We will use P_i to represent any Pauli matrix X, Y, or Z and for $i \neq j$, $P_i \neq P_j$. The families of DD sequences can then be listed as follows: DD4, length-4 sequences $P_1P_2P_1P_2$; DD8, length-8 sequences $IP_2P_1P_2IP_2P_1P_2$; EDD8, length-8 sequences $P_1P_2P_1P_2P_2P_1P_2P_1$; CDD16, length-16 concatenated sequences; $DD4[DD4]$; CDD32, length-32 concatenated sequences $DD4[DD8]$ and $DD8[DD4]$; and CDD64, length-64 concatenated sequences $DD4[CDD16]$ and $DD8[DD8]$. Longer DD sequences can again be obtained by the concatenation of the ones listed above and in the ideal situation they provide better and better protection against the noise. However, with realistic experimental capability, the performance usually saturates at a certain concatenation level. Since at this moment we are only optimizing short DD sequences, the listed ones are sufficient to provide a baseline for our purpose. One important family we did not include here is the Knill DD (KDD) [28], because it requires the use of non-Pauli gates.

However, we cannot expect these requirements to be met in all real-world experiments. The two major imperfections that are often studied are the flip-angle errors and the finite duration of the pulses. Flip-angle errors arise from not being able to control the strength and time duration of $H_C(t)$ perfectly, thus the intended pulse $V(t) = O\delta(t)$ becomes $V(t) = (1 \pm \epsilon)O\delta(t)$. Also, since zero-width pulses $O\delta(t)$ are experimentally impossible, we must consider finite-width pulses that approximate the ideal ones. In this paper we will only consider the imperfection of finite-width pulses. However, it is straightforward to apply our algorithm to pulses with flip-angle errors.

B. Measure of performance

There are multiple ways to quantify the performance of DD sequences. In practice, we choose different measures to suit the intended applications. Here we use the same measure as in [24,29], which has the advantage of being (initially) state independent and having a closed formula for numerical simulation:

$$D(U,I) = \sqrt{1 - \frac{1}{d_S d_B}\|\mathrm{Tr}_S(U)\|_{\mathrm{Tr}}},$$

where U represents the full evolution operator generated by $H(t)$, d_S and d_B are the dimensions of the system and environment Hilbert space $\mathcal{H}_S$ and $\mathcal{H}_B$, respectively, $\|X\|_{\mathrm{Tr}} = \mathrm{Tr}(\sqrt{X^\dagger X})$ is the trace norm, and $\mathrm{Tr}_S(\cdot)$ is the partial trace over $\mathcal{H}_S$. The smaller $D(U,I)$ is, the better the system preserved its quantum state after the time evolution. For example, the ideal evolution $U = I_S \otimes U_B$ has the corresponding $D(U,I) = 0$.

In experiments, it is very hard to evaluate $D(U,I)$, as we often do not have access to the bath's degree of freedom. Instead, the performance of DD sequences is often gauged by doing process tomography for the whole time duration where DD is applied [25,30]. Although it is a different measure compared to our choice above, the optimization procedure can still be applied as it does not rely on the concrete form of the measure. Moreover, for solid state implementations such as superconducting qubits or quantum dots, a typical run of initialization, applying DD sequences and measurements, can be done on the time scale of 1 ms or much faster. Thus, it is realistic that on the time scale of days we can gather a large data set of DD sequences and their performance, which is needed for our algorithm.

C. Recurrent neural networks

Sequential models are widely used in machine learning for problems with a natural sequential structure, e.g., speech and handwriting recognition, protein secondary structure prediction, etc. For dynamical decoupling, not only do we apply the gates sequentially in the time domain, but also the longer DD sequences are often formed by repetition or concatenation of the short ones. Moreover, once the quantum information of the system is completely mixed into the environment, it is hard to retrieve it again by DD. Thus, an educated guess is that the performance of a DD sequence largely depends on the short subsequences of it, which can be modeled well by the sequential models.

Since our goal is not simply to approximate the distribution of good dynamical decoupling sequences by learning their structure but to sample from the learned distribution to efficiently generate new good sequences, we will further restrict ourselves to the class of generative sequential models. Overall, these models try to solve the following problem: Given $\{x_i\}_{i<t}$, approximate the conditional probability $p(x_t|x_{t-1},\ldots,x_1)$. As a simple example, we can estimate the conditional probability $p(x_t|x_{t-1})$ from a certain data set and use it to generate new sequences.[1] For more sophisticated problems (e.g., natural language or handwriting), it is not enough to only consider the nearest-neighbor correlations as simple models like Markov chains of order one do.

[1]This idea can be at least dated back to Shannon [31], where this model generated "English sentences" like "On ie antsoutinys are t inctore st be s deamy achin d ilonasive tucoowe at . . .."

The long short-term memory (LSTM) network, a variation of the RNN, is a state-of-the-art technique for modeling longer correlations [32] and is comparably easy to train. The core idea of RNNs is that the network maintains an internal state in which it encodes information from previous time steps. This allows the model to, at least theoretically, incorporate all previous time steps into the output for a given time. Some RNNs have even been shown to be Turing complete [33]. In practice, however, RNNs often can only model relatively short sequences correctly due to an inherently unstable optimization process. This is where LSTMs improve over normal RNNs, as they allow for training of much longer sequences in a stable manner. Furthermore, LSTMs, like all ANNs, are based on matrix multiplication and the elementwise application of simple nonlinear functions. This makes them especially efficient to evaluate.

Algorithm 1. Optimization algorithm.

```
Input:  Number of initial models to train: $n$
        Number of models to keep: $k$
        Percentage of data to keep: $p$
        Set of possible topologies: $\mathcal{M}$
        Size of data: $d$
$D \leftarrow$ generateRandomData $(d)$
$D, \langle\varsigma_s\rangle \leftarrow$ keepBestData $(D,p)$
$M \leftarrow$ trainRandomModels $(n,D,\mathcal{M})$
$M \leftarrow$ keepBestKModels $(M,k)$
while $\langle\varsigma_s\rangle$ not converged do
    $M \leftarrow$ trainBestModels $(D)$
    $D \leftarrow$ generateDataFromModels $(M,d)$
    $D, \langle\varsigma_s\rangle \leftarrow$ keepBestData $(D,p)$
end
Output: $\langle\varsigma_s\rangle, D, M$
```

From the machine learning perspective, we treat the problem at hand as a supervised learning problem where we provide the model with examples that it is to reproduce according to some error measure. It is also possible to formulate our problem in the framework of reinforcement learning. However, since we only compute the performance of a whole DD sequence, there is no immediate reward when choosing a gate in the middle of the sequence. Given the length of the sequences we are optimizing, it is likely a reinforcement learning algorithm will need help from certain (un)supervised learning, similar to the way in [3]. A short introduction to machine learning, LSTMs, and their terminology can be found in the Appendixes. More exhaustive discussions can be found in [34–36].

III. ALGORITHM

The algorithm presented in this section is designed with the goal in mind to encode little prior knowledge about the problem into it, in order to make it generally applicable to different imperfections in the experiment. Following this idea, the method is agnostic towards the nature of the considered gates, the noise model, and the measure of performance. To implement this, the algorithm assumes that the individual gates are represented by a unique integer number such that every sequence $s \in \mathcal{G}^{\otimes L_s}$, with $\mathcal{G}$ denoting the set of unique identifiers and L_s being the length of s, and it is provided with a function $f(s)$ to compute the score ς_s of a given sequence s, taking into account the noise model. The optimization problem we want to solve is

$$\min_s f(s) = \min_s \varsigma_s.$$

By assumption, we have no information about f but can efficiently evaluate it. We furthermore assume the set of good sequences to exhibit common structural properties that can be learned well by a machine learning model. So we propose to solve it indirectly by training a generative model $m \in \mathcal{M}$ to approximate the distribution of good sequences, $\mathcal{M}$ being the set of possible models. That means we assume $s_t \sim p_m(s_{t-1}, \ldots ,s_1)$, with s_t being the gate at time t and p_m denoting the distribution learned by m. Then we want to find an optimal m that ideally learns a meaningful representation of the structure of good sequences. In this work we choose the type of model to be the LSTM. We now tackle this surrogate problem by alternatingly solving

$$\max_{m \in \mathcal{M}} \mathcal{L}(m|T),$$

where $\mathcal{L}$ denotes the likelihood and T the training data, and then sampling sequences from the model m to generate a new T consisting of better solutions. The algorithm hence consists of two nested optimization loops, where the inner loop fits a number of LSTMs to the current data while the outer loop uses the output of the inner loop to generate new training data. This scheme of alternatingly fixing the data to optimize the models and consecutively fixing the models to optimize the data resembles the probabilistic model building genetic algorithm [37] and to some extent the expectation-maximization algorithm [38]. The method is shown in Algorithm 1. Partial justification of this heuristic algorithm is given in Appendix C. However, it is easy to see that the algorithm will not always find the global optimum. For example, it is conceivable that for certain problems the second to the 100th best solutions share no common structure with the first one. In that case, it would be unlikely for the machine learning approach to find the optimal one. There is however likely no universal method to bypass this obstruction, as unless we know the best sequences already, it is impossible to verify that they exhibit some structure similar to the training sets. This obstruction seems natural since many optimization problems are believed to be computationally hard. Thus, we should not assume to be able to solve them by the above routine.

We will now explain the most important aspects of the algorithm in more detail.

(a) Choice of LSTMs. The data we want to generate in our application are of sequential nature. This makes employing LSTMs an obvious choice as they pose one of the most powerful models available today for sequential data. Furthermore, the known well-performing families of DD sequences are constructed by nested concatenations of shorter sequences and hence show strong local correlations as well as global structure. Long short-term memories and especially models consisting of multiple layers of LSTMs are known to perform very well on such data and should therefore be able to learn and reproduce this multiscale structure better than simpler and shallow models.

(b) Generation of the initial training data. The size d and the quality, i.e., the percentage p of the initial data to be kept, are the parameters that we can specify. The data are then generated by sampling a gate from the uniform distribution over all gates for each time step. The average score of the initial data can then be used as a baseline to compare against in case no other reference value is available. We would like to point out that in the application considered in this work, an alternative way to generate the initial data might be to use the models trained on shorter sequences. This approach could lead to an initial data set with much higher average score, but at the price of introducing the bias from the previously trained RNNs.

(c) Training of the LSTMs. To reduce the chance of ending up in a bad local optimum, for each training set several different architectures of LSTMs are trained (see Appendix D 2 for detailed description of LSTMs). These models are independently sampled $\mathcal{M}$. More precisely, for the first generation of models, we sample a larger set of n models from $\mathcal{M}$ and train them. We then select the best k models and reuse them for all following generations. While it might introduce some bias to the optimization, this measure drastically reduces the number of models that need to be trained in total. The training problem is defined by assuming a multinoulli distribution over the gates of each time step and minimizing the corresponding negative log-likelihood $-\sum_t \delta_{s_t,i} \log_2 p_{m,i}(s_{t-1}, \ldots ,s_1)$, where i is the index of the correct next gate, $p_{m,i}$ is its predicted probability computed by the LSTM m, and $\delta_{s_t,i} = 1$ if and only if $s_t = i$. This error measure is also known as the cross entropy. To avoid overfitting, we use a version of early stopping where we monitor the average score $\langle \varsigma_s \rangle_{p_m}$ of sequences generated by m and stop training when $\langle \varsigma_s \rangle_{p_m}$ stops improving. We employ the optimizer Adam [39] for robust stochastic optimization.

(d) Selecting the best models. As we employ early stopping based on the average score $\langle \varsigma_s \rangle_{p_m}$, we also rank every trained model m according to this measure. One could argue that ranking the models with respect to their best scores would be a more natural choice. This however might favor models that actually produce bad sequences but have generated a few good sequences only by chance. Using $\langle \varsigma_s \rangle_{p_m}$ is hence a more robust criterion. It would of course be possible to also consider other modes of the p_m, such as the variance or the skewness. These properties could be used to assess the ability of a model to generate diverse and good sequences. We find, however, that the models in our experiments are able to generate new and diverse sequences, thus we only use the average score as a benchmark for selecting models.

(e) Generation of the new training data. The selected models are used to generate d new training data by sampling from p_m. This is done by sampling s_t from $p_i(s_{t-1}, \ldots ,s_1)$ beginning with a random initialization for $t = 1$ and then using s_{t-1} as input for time step t. We combine the generated sequences with the previous training sets, remove any duplicates, and order the sequences by their scores. We then choose the best p percent for the next iteration of the optimization. This procedure ensures a monotonic improvement of the training data. Note that all selected models contribute equally many data to strengthen the diversity of the new training data. A

possible extension would be to apply weighting of the models according to some properties of their learned distributions. Note though that ordering the generated sequences by their score is already a form of implicit weighting of the models.

IV. NUMERICAL RESULTS

A. Noise model and the control Hamiltonian

Throughout the paper we will use the same noise model as in [24]. We consider a 1-qubit system and a 4-qubit bath, namely, $\dim(\mathcal{H}_S) = 2$ and $\dim(\mathcal{H}_B) = 16$. The small dimension of the bath is for faster numerical simulation and there is no reason for us to think that our algorithm would only work for a small bath as the size of the bath enters the algorithm only via the score-computation function. The total noise Hamiltonian consists of (at most) three-body interactions between the system and bath qubits with random strength

$$H_0 = \sum_{\mu\in\{I,X,Y,Z\}} \sigma^\mu \otimes B_\mu, \tag{1}$$

where σ^μ is summed over Pauli matrices on the system qubit and B_μ is given by

$$B_\mu = \sum_{i\neq j}\sum_{\alpha,\beta} c^\mu_{\alpha\beta}\left(\sigma_i^\alpha \otimes \sigma_j^\beta\right),$$

where i,j is summed over indices of the bath qubits and $\sigma_i^{\alpha\,(\beta)}$ is the Pauli matrix on qubit i of the bath. We consider the scenario where the system-bath interaction is much stronger than the pure bath terms. More precisely, we set $c^\mu_{\alpha\beta} \approx 1000 c^I_{\alpha\beta}$ for $\mu \in \{X,Y,Z\}$. Apart from this constraint, the absolute values $|c^\mu_{\alpha\beta}|$ are chosen randomly from a range $[a,b]$, where we set $b \approx 3a$ to avoid too many terms vanishing in (1). The result Hamiltonian has a 2-norm $\|H_0\| = 20.4$.

For the control Hamiltonian, we consider the less explored scenario where the pulse shape have finite width but no switch time between them (100% duty cycle). In other words, the control Hamiltonian is piecewise constant

$$H_C(t) = H_k \quad \text{for} \quad k\tau_d \leqslant t < (k+1)\tau_d,$$

where τ_d is a small time period with respect to the norm of H_0 and $e^{-iH_k\tau_d} \in \{I,X,Y,Z\}$. This is a good toy model for experimental settings whose DD performance is mainly limited by the strength of the control Hamiltonian, but not the speed of shifting between Hamiltonians. Since this regime is less explored in theoretical studies, it is an interesting scenario to explore via machine learning. Another restriction we put on $H_C(t)$ is

$$H_C(t) = -H_C(T-t),$$

where T is the total evolution time. This condition ensures $U_C(T) = \mathcal{T}\exp\{-i\int_0^T dt' H_C(t')\} = I$ and it allows us to apply the same code on the setting where the system has more than one qubit. It is known that this family of symmetric Hamiltonians can remove the first-order terms of τ_d in the average Hamiltonian [22,40]. So, strictly speaking, this should be counted as prior knowledge. However, when we compare the known DD sequences with the numerically found ones, we also use the symmetric version of the known DD sequences. Thus, we perform the comparison on equal terms.

B. Numerical experiments

In the following we present the results of a number of experiments we have conducted to evaluate the performance of our method. We consider sequences consisting of 32, 64, and 128 gates for varying values of τ_d. This translates to having to optimize the distribution of the first 16, 32, and 64 gates, respectively. To compute ς_s, we use the figure of merit D as defined in Sec. II A. Thus, a lower score is better. For $\mathcal{M}$, we consider models with two or three stacked LSTM layers followed by a final softmax layer. The layers comprise 20 to 200 units where layers closer to the input have a higher number of units. We allow for peephole connections and linear projections of the output of every LSTM layer to a lower number dimension [35]. The optimization parameters are also randomly sampled from sets of reasonable values. We choose the step rate to be in $\{10^{-1},10^{-2}\}$ and the batch size to take values in $\{200,500,1000\}$. The parameters specific to the Adam optimizer β_1, β_2, and ϵ, we sample from $\{0.2,0.7,0.9\}$, $\{0.9,0.99,0.999\}$, and $\{10^{-8},10^{-5}\}$, respectively. We perform a truncation of the gradients to 32 time steps in order to counter instabilities in the optimization (see Appendix D 3). As we have stated above, we also employ early stopping in the sense that, for every optimization of a model, we keep the parameters that generate the sequences with the best average score. The algorithm was run until either the best known score was beat or the scores converged, depending on the goal of the respective experiment. We will now briefly list the concrete experiment settings and discuss the results.

(i) Experiment E1: Length 32. In this first experiment, we considered sequences of 32 gates with $\tau_d = 0.002$. We let the algorithm train $n = 30$ models initially and set the number of models to be kept k to 5. We combined the data generated by the LSTMs with the previous training set after each generation and chose the best 10% as the new training data, consisting of 10 000 sequences for each generation. We let every model train for 100 epochs.

(ii) Experiment E2: Length 64. In our next experiment, we tackled a more difficult scenario with 64 gates and a larger $\tau_d = 0.004$. We set $n = 50$ and $k = 5$. Again, we used the best 10% of both generated and previous data as new training data, which consist of a total 10 000 sequences for each training set.

(iii) Experiment E3: Length 128. In the third experiment we tried our method on even longer sequences of 128 gates with τ_d again being 0.004. Due to the very large sequence space, we set the size of the training sets to 20 000, again using the best 10% of sequences generated by the selected models and the previous training set. The number of epochs was increased to 200. We set $n = 30$ and $k = 5$. Here we let the algorithm run until both the average and the best score converged to examine its behavior in long runs.

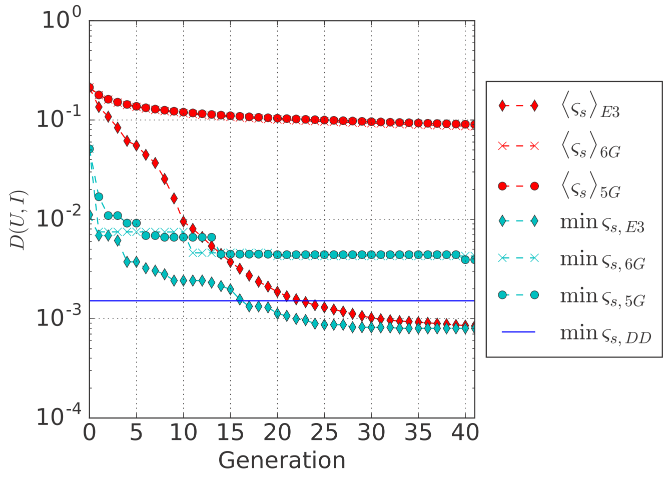
(iv) Experiment E4: Length 32 with random gates. Finally, we tested the performance of Algorithm 1 in the case where we replaced the Pauli gates $\{I,X,Y,Z\}$ with ten randomly chosen gates. More precisely, we chose each gate g_j to be a randomly generated single two-dimensional unitary operator with eigenvalues 1 and -1, i.e., $g_j = U_j^\dagger X U_j$, where U_j is a random unitary. All other parameters were kept as in experiment $E1$.

TABLE I. Comparison of the results obtained in experiments $E1$, $E2$, $E3$, and $E4$ to the best theoretically derived DD families. For each experiment, the average and best score of the last training data and the average score of the best model of the last generation are shown. They are compared to random sequences and the two DD classes that yield the best average and overall best score, respectively. The best results are printed bold.

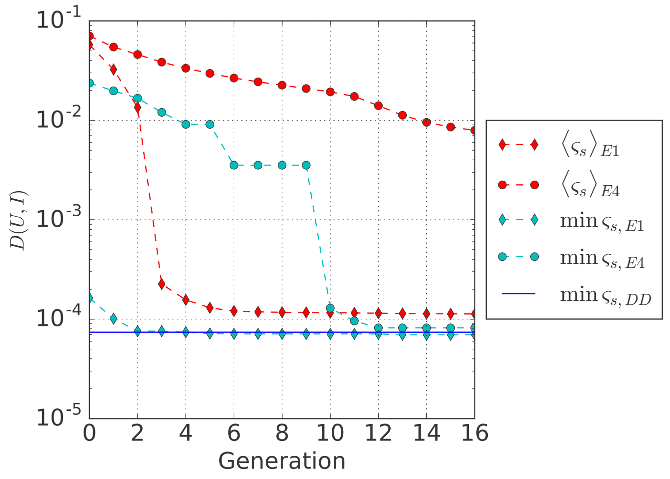
Sequences	$\langle\varsigma_s\rangle$	min ς_s
	Experiment $E2$	
EDD8	0.002398	0.002112
CDD32	0.053250	0.000803
last training set $E2$	**0.000712**	**0.000381**
best model $E2$	0.016692	
random	0.341667	
	Experiment $E3$	
EDD8	0.004793	0.004222
CDD64	0.031547	0.001514
last training set $E3$	**0.000827**	**0.000798**
best model $E3$	0.029341	
random	0.44918	
	Experiments $E1$ and $E4$	
EDD8	0.000151	0.000133
CDD16	0.010699	0.000074
last training set $E1$	**0.000112**	**0.000070**
last training set $E4$	0.007178	0.000082
best model $E1$	0.003089	
random	0.125371	

In Table I we compare the last training data and the best model of the last generation of $E1$–$E4$ against the two DD families that achieve the best average and minimal scores for the given experiment, respectively. We also plot the convergence of the training data of $E3$ and $E1$ with $E4$ in the Figs. 1(a) and 1(b), respectively. In general, the results for $E1$, $E2$, and $E3$ clearly show that our method outperforms DD, achieving a better minimal score of the generated data in a moderate number of iterations and with a relatively small set of models. The results of $E4$ will be discussed below. These findings indicate that our method converges to good local optima and that the models are able to learn a meaningful internal representation of the sequences that allows for efficient sampling of good sequences. There is however a noticeable gap between the scores of the training data and the models. A possible remedy for this could be an increase of the training data size or an adjustment of the model parameters in later stages of the optimization to account for the change in the structure of the data.

To assess the importance of LSTMs for the performance of our algorithm, in experiment $E3$, we also ran a different version of our method where we replaced the LSTMs by simple 5/6-gram models, which only model and generate sequences based on local correlations (see Appendix A 2 for the definition). The convergence plots in Fig. 1(a) show that LSTMs are indeed superior to the simpler models. They are able to improve the average and best scores faster and ultimately let the algorithm converge to a better local optimum. This advantage most likely results from the fact that the LSTM models are able to leverage information about longer-range correlations in the data. These results hence justify our choice of LSTMs as a machine learning model to optimize DD sequences.



(a) Experiment E3



(b) Experiments E1 and E4

FIG. 1. Convergence of the algorithm in (a) $E3$ compared to the case where LSTMs are replaced by 5/6-gram models and (b) $E1$ compared to $E4$ as both consider the same problem setting. In (a) it is clearly visible that LSTMs outperform the n-gram models, while (b) reflects the physical knowledge that the Pauli unitaries are a better choice than random gates. Note that the red (dark) crosses in (a) are almost covered by the red (dark) circles. As a reference, we show the score of the best DD sequence obtained from the known DD classes.

We also compared the results of experiments $E1$ and $E4$ to examine the importance of using the Pauli group as the gate set. Figure 1(b) shows that while for $E1$ the average score quickly becomes very good and the best score exceeds the best known result after a few generations, in $E4$ the average score of the data improves much slower and remains significantly worse than that of $E1$. Although the best score exhibits a much stronger improvement, it eventually converges to a value slightly worse than that of the best theoretical DD sequence and the one found in $E1$. This is expected since with the Pauli group we can achieve first-order decoupling with DD sequences of length 4, which is the shortest. On the other hand, with random unitaries, in general it will take much longer sequences to have approximate first-order decoupling, during which the system and environment can become fairly entangled.

Another interesting aspect to note is the rather strong improvement of the average scores occurring in $E3$ and $E1$ between generations 8 to 10 and 2 to 3, respectively. These jumps can be explained by the known existence of several strictly separate regimes in sequence space that differ strongly in their performance. The results indicate that our algorithm is able to iteratively improve the learned distributions to eventually capture the regime of very good sequences.

In order to verify that sampling the initial training data from the distributions learned for shorter sequences is a viable alternative to uniform sampling, we let the best model obtained in $E2$ generate an initial data set for the problem setting of $E3$. The obtained data were found to have an average score of 0.037 175, which is about one order of magnitude better than the average of the initial training data generated by uniform sampling.

V. CONCLUSION

We have introduced a method for optimizing dynamical decoupling sequences that differs from previous work by the ability to utilize much larger data sets generated during the optimization. Its ability to efficiently generate large sets of good sequences could be used along with other optimization methods to cover their weaknesses or to perform statistical analysis of these sequences. We showed that for certain imperfect control Hamiltonians, our method is able to outperform (almost all) known DD sequences. The little prior knowledge about DD we use is (i) choosing Pauli operators as pulses in the sequences (see experiment $E4$ and its discussion), (ii) choosing specific lengths for the DD sequences, and (iii) enforcing the reversal symmetry, as discussed in Sec. IV A. However, we do not need to initialize the data set in a specific way as in Appendix C 5 a of [24], which actually contains a certain amount of prior knowledge of DD. Also, our method does not fundamentally rely on the prior knowledge stated above. It is conceivable that the use of this prior knowledge can be lifted, at the price of a possibly much slower optimization procedure. For example, the KDD scheme helps to further increase the performance of CDD sequences in some experiments [23]. Thus, an interesting question is when given the freedom of applying non-Pauli gates and choosing variable lengths of the sequences, whether our algorithm could discover a similar strategy. Thus, a possible direction of future research is to see how we can minimize the slowdown when not incorporating any prior knowledge and whether we can obtain good DD sequences with non-Pauli pulses.

While we have applied the algorithm to the case of quantum memory and compared it to dynamical decoupling, it is of general nature. It can in principle be applied to every problem where the optimization of a sequence of gates with respect to some well-defined figure of merit is desired and where it is feasible to evaluate this performance measure for larger numbers of sequences. However, due to the nature of the underlying machine learning model, good results will likely only be obtained for problems whose solution depends strongly on local correlations in the sequences.

ACKNOWLEDGMENTS

We would like to thank Geza Giedke for helpful discussions and comments on the draft. The idea of this paper partially stems from a discussion between Courtney Brell and X.N. about using genetic algorithms to optimize quantum memory. We would also like to thank Peter Wittek for helpful comments. M.A. acknowledges funding by the Elite Network of Bavaria via the doctoral programme "Exploring Quantum Matter."

M.A. and X.N. are contributed equally to this work.

APPENDIX A: ANALYSIS

1. Local correlations of DD sequences

As we suggested earlier, the reason we use RNNs as the probabilistic model is that the performance of dynamical decoupling sequences heavily depends on their local correlations. To illustrate this fact, we can count the frequency of length-2 (see Table II) (or length-3) subsequences from the training set of the 30th generation in experiment 3. We can then compare these statistics to the ones of the sequences generated by the LSTM, which is trained based on the training set. We can see indeed that the percentages match very well. To get more detail about local correlations, we could also count the frequency of length-3 subsequences (see Table III). Note that since the table is based on the data sets in the late stage of the optimization, the distribution of the subsequences are already very polarized. However, we observe the same behavior (the percentages matches well) in other experiments at different stages of the optimization as well. However, RNNs do not only take into account local correlations, as we show in Fig. 1 that they perform better compared to the n-gram models, which we will introduce in the next section.

2. The n-gram models

n-grams are the simplest sequential models that treat the sequences as stationary Markov chains with order $n-1$.

TABLE II. Frequency of length-2 subsequences, from the training set and the set generated by the trained LSTM (given in parentheses) at generation 30 of experiment 3. The total number of subsequences is around 1.2×10^6.

Previous gate \ Next gate	I	X	Y	Z
I	0.00% (0.00%)	0.04% (0.08%)	0.15% (0.68%)	0.02% (0.08%)
X	0.05% (0.22%)	5.38% (5.04%)	30.53% (30.47%)	1.39% (1.26%)
Y	0.07% (0.20%)	30.17% (30.47%)	18.40% (18.61%)	5.84% (5.50%)
Z	0.01% (0.02%)	1.90% (1.68%)	5.75% (5.42%)	0.30% (0.27%)

TABLE III. Frequency of length-3 subsequences started with gate X, from the training set and the set generated by the trained LSTM (given in parentheses) at generation 30 of experiment 3. The total number of subsequences started with X is around 450 000.

Second gate \ Last gate	I	X	Y	Z
I	0.00% (0.00%)	0.02% (0.05%)	0.12% (0.55%)	0.00% (0.01%)
X	0.00% (0.00%)	1.40% (1.22%)	11.99% (11.52%)	0.32% (0.32%)
Y	0.15% (0.47%)	44.79% (45.09%)	33.39% (33.54%)	4.11% (3.85%)
Z	0.01% (0.01%)	2.38% (2.14%)	1.05% (0.98%)	0.28% (0.26%)

Operationally, given a set of sequences, we first estimate the conditional probability distribution

$$p_{x_n,x_{n-1}\cdots x_1} = \Pr(X_t = x_n | X_{t-1} = x_{n-1}, \dots X_{t-n+1} = x_1).$$

Note that we assume the conditional probability is independent of t (hence stationary Markov chain). The estimation is done by counting over the whole set of sequences. The generation of new sequences based on the conditional probability $p_{x_n,x_{n-1}\cdots x_1}$ is straightforward, as we can repeatedly sample from it based on the previous $n-1$ items. This behavior is different compared to that of the RNNs, which have memory units that can store information for an arbitrarily long time in theory.

3. Optimization without reusing data from previous training sets

During the optimization processes in the main text, we always reuse the data from previous training sets, in the sense that we first add the new sequences generated by the models to the training sets and then delete the worst sequences. An interesting question is what will happen if we generate new training sets completely from the trained models. In Fig. 2 we plot the counterpart of Fig. 1(a) with this modification (as well as not deleting duplicated sequences from the training set). We can see that for the LSTMs experiment, the final minimum score gets slightly worse, which is 0.000 874. However, the 5/6-gram experiments actually perform better when not reusing data. While it seems counterintuitive, this can be possibly explained by the fact that in the case of reused data with unique sequences the higher diversity of the data might make it harder for the models to find local correlations, which then in turn slows down the optimization. There is other interesting information contained in the plot. For example, we can see the minimum scores almost always decrease, which implies that the LSTMs are able to learn new information about good sequences in most generations.

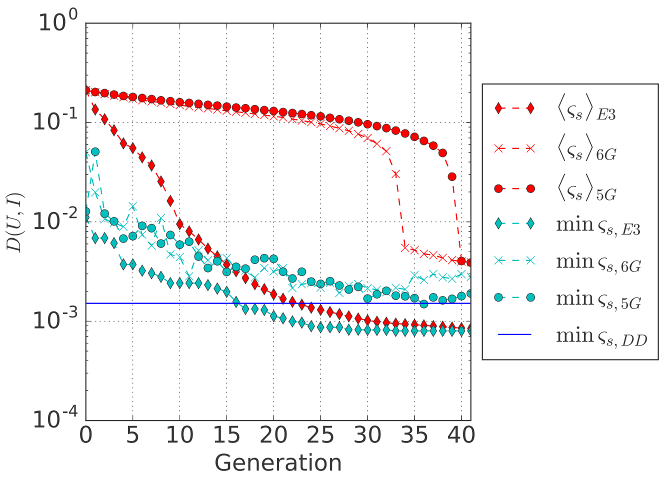


FIG. 2. The 3- and 5/6-gram experiments without data reusage. Otherwise, the experiments are done in the same way as in Fig. 1(a).

4. Performance of the obtained sequences with a larger heat bath

In the main text, all the numerical simulations are done on a randomly generated noise Hamiltonian with the dimension of the bath being $\dim(\mathcal{H}_B) = 16$. The small dimension of the bath is used in order to have a fast simulation. Here we test the performance of some obtained sequences from experiment 2, in the presence of a larger bath with $\dim(\mathcal{H}_B) = 128$. Apart from the change of dimension, the Hamiltonian H_0 is again randomly generated according to the description in Sec. IV A, which has a 2-norm $\|H_0\| = 24.0$. We then computed the scores of the top 500 DD sequences in the last generation of experiment 2. The results are shown in Table IV. While the best score of the obtained sequences is worse than the best score of CDD32, it is clear that, on average, the obtained sequences still work fairly well. This also suggests that our algorithm is potentially capable of adapting to the particular noise Hamiltonian, as the learned sequences outperform known DD families in experiment 2.

APPENDIX B: BEST SEQUENCES

We list here the best sequences we found in experiments 1–3 from the numerical results section. We denote the identity by I and X,Y,Z refer to the respective Pauli matrices. Note that we show only the first half of the complete sequence as the second one is just the first half reversed. In experiment 1 we found $X,Y,X,Z,X,Y,X,Z,Z,X,Y,X,Z,X,Y,X$; in experiment 2, $Z,Z,X,Z,Z,Z,X,Z,Z,X,Z,X,X,X,Z,X,X,X,Z,X,X,Z,X,$

TABLE IV. Comparison between the scores of the top 500 DD sequences in the last generation of experiment 2 and some DD families for the larger bath $\dim(\mathcal{H}_B) = 128$. The best score of the 500 sequences is worse than the best score of CDD32. However, it is clear that, on average, the obtained sequences still work fairly well.

Sequences	$\langle\varsigma\rangle$	$\min\varsigma$
EDD8	0.002781	0.002203
CDD32	0.053753	0.000432
top 500	0.001081	0.000626

X,X,Z,X,Z,Z,X,Z,Z; and in experiment 3, $Z,X,Z,Z,Y,X,Y,Z,Y,X,Y,X,Y,Y,X,Y,Y,Y,Y,X,Y,Y,Y,X,Y,Y,X,Y,X,Y,X,Y,Y,Z,X,Z,Y,Z,X,Z,Y,X,Y,X,X,Y,X,Y,X,Y,X,Y,Y,X,Y,Y,Y,X,Y,X,X,Y,X,X$.

APPENDIX C: COMPARISON OF OPTIMIZATION ALGORITHMS

In this appendix we will give a comparison between several optimization algorithms applied to black-box problems. In other words, the algorithm needs to optimize (minimize) the objective function f only by looking at the values of $f(x)$ (without knowing the concrete formula of it). We are going to look at the following types of algorithms: gradient-based algorithms (when we can access the gradient of f), e.g., Newton's method, variants of gradient descent; Metropolis-Hasting algorithms and its variants, e.g., simulated annealing; and genetic algorithm and its variants, e.g., a probabilistic model building genetic algorithm (PMBGA). The performance of an optimization algorithm depends heavily on the class of the problems it is applied to. (This fact is remotely related to the no free lunch theorem for optimization). Thus, in the following, we will use different objective functions to illustrate the strong and weak points of those algorithms.

1. Gradient-based algorithms

To understand the idea of these algorithms, it is enough to consider $f:\mathbb{R}\rightarrow\mathbb{R}$ defined on a single variable. The simplest gradient descent for finding the minimum of f is the following iterative algorithm: starting from a random number x_0 and successively computing $x_{n+1}=x_n-\alpha f'(x_n)$. Gradient-based algorithms perform well on functions with nonvanishing gradients almost everywhere and very few local minima and likely have a poor performance otherwise. For example, the above algorithm would perform very well on a simple function $f(x)=x^2$, but much worse on the fast oscillating function

$$\begin{aligned} f(x) = {} & \sin(8x)+0.5\sin(4x) \\ & +0.3\sin(2x)+0.1\sin(x). \end{aligned} \tag{C1}$$

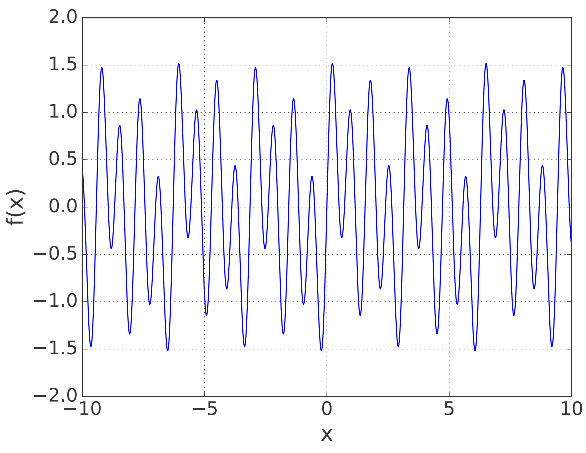


FIG. 3. Plot of the function (C1).

We plot the above function in Fig. 3. It is easy to see that we can construct $f(x)=\sum_{i=1}^{N}a_i\sin(2^i x)$ such that the chance of finding the global minimum is arbitrarily small.

2. Simulated annealing

Simulated annealing (SA) and its variants stem from the Metropolis-Hastings algorithm. The main idea is constructing a family of probability distribution $p(x,T)$ based on the values of the objective function $f(x)$, with the requirement $p(x,0)>0$ only when x is a global minimum of f. Then we repeatedly sample from $p(x,T)$ while slowly decreasing T. In practice, simulated annealing is also an iterative algorithm, i.e., it chooses x_{n+1} based on x_n. Since SA uses the Metropolis-Hastings algorithm as a subroutine, there is a nonzero chance to choose x_{n+1} such that $f(x_{n+1})>f(x_n)$. So, in principle, SA could escape from local minima, which is an advantage compared to gradient descent. Simulated annealing also works for functions with discrete variables. As a trade-off, it is likely to be slower compared to gradient descent when f has very few local minima. Moreover, while SA has the mechanism to escape from local minima, in practice it could work poorly on functions with many local minima and high barriers between them, e.g., the function (C1).

3. Genetic algorithms and beyond

In this section we will assume that f has the form $f:\mathbb{R}^N\rightarrow\mathbb{R}$. A common feature in all versions of genetic algorithms is that they maintain a population of solutions $\{\vec{x}_i, 1\leqslant i\leqslant M\}$, where $\vec{x}_i=(x_{i1},\ldots,x_{iN})$. For the first generation, a number of $M'>M$ solutions is randomly generated, then we pick the $\vec{x}_i$ with the M smallest $f(\vec{x}_i)$ as the population. To generate new potential solutions for new generations, several different operations are introduced. In the original genetic algorithm, the two such operations are crossover and mutation. The effect of the mutation operation on a solution $\vec{x}$ is

$$(x_1,\ldots,x_j,\ldots,x_N)\rightarrow(x_1,\ldots,x_j',\ldots,x_N),$$

where x_j' is a random number. The crossover operation acts on two solutions $\vec{x}$ and $\vec{y}$,

$$(\vec{x},\vec{y})\rightarrow(x_1,\ldots,x_j,y_{j+1},\ldots,y_N),$$

where the position j is picked randomly. Then we can use these two operations to generate M'' new test solutions from the first generation, combine them with the M old solutions, and pick the top M solutions as the population of the second generation. Later generations can be obtained by repeating these steps.

To illustrate the advantage of the (original) genetic algorithm, we can consider the objective function f,

$$f(\vec{x})=\sum_j f_j(x_j).$$

In this case, if $f(\vec{x})$ is (relatively) small, then either $\sum_{j=1}^{k}f_j(x_j)$ or $\sum_{j=k+1}^{N}f_j(x_j)$ is (relatively) small. Thus the crossover operations serve as nonlocal jumps, while the mutation operations help to find local minimum. However, in general, it is not clear for what kind of function f the inclusion of the crossover operations could provide an advantage.

It is easy to construct counterexamples such that the crossover operations deteriorate the performance, such as

$$f(\vec{x}) = f(\vec{x}_a,\vec{x}_b) = \|\vec{x}_a - \vec{x}_b\|,$$

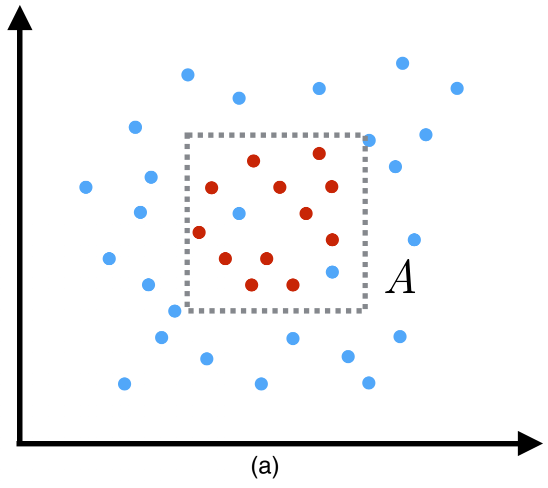
where $\vec{x}_a$ and $\vec{x}_b$ have equal dimension and $\|\cdot\|$ is the Euclidean norm. Clearly, in most cases, the crossover of two good solutions will only produce inferior new solutions.

It turns out that the most important feature of genetic algorithms is the use of a population. In comparison, other optimization methods we mentioned previously only keep track of the last test solution. If we are willing to believe that good solutions of the function f have a certain structure (thus partially dropping the black-box requirement of f), it is possible that we can identify this structure from the solutions in the population and then generate new test solutions. This idea has led to the so-called probabilistic model building genetic algorithm and its variants [37,41]. The optimization algorithm we introduced in the main text is also closely related to this idea.

Instead of going through the details of these algorithms, we will explain the idea using a simple example, as illustrated in Fig. 4. Suppose that we want to minimize a function $f(x,y)$ with two variables defined on a finite region of $\mathbf{R}^2$ and prior knowledge of f allows us to make the hypothesis h that all points $\{(x,y)\}$ with values $f(x,y) < M$ exist in a certain region A [e.g., the square in Fig. 4(a)]. By sampling random points from the domain of the function, we can verify or refute the hypothesis h. For simplicity, we assume that h is satisfied for all sampled points and N of them is inside the region; then the opposite hypothesis of an α fraction of points $\{(x,y)\}$ with values $f(x,y) < M$ existing outside the region A will give the observed data a likelihood of $(1-\alpha)^N$. Thus, we can just optimize f over the region A by ignoring a very small fraction of the good solutions. It is easy to see that we can iterate this process, as long as we can formulate a small number of hypotheses such that one of them will describe the good solutions correctly. Our algorithm in the main text resembles this toy example. However, for functions in high dimension and sophisticated generative models such as RNNs, it is hard to give a mathematical justification like in the above example.

It is natural to concatenate the above process [see Fig. 4(b)]. Let S_0 be the domain of f, and S_1 be the points in region A. By sampling enough points from S_1, we might be able to build a model and sample from a even smaller set S_2 with the good solutions (e.g., find a region $B \subset A$). This way we will introduce a series of sets $\{S_i\}_{i\leqslant K}$ that we can sample from. Assuming that the order of these subsets satisfies $|S_{i+1}| < \frac{1}{2}|S_i|$, then in the ideal scenario the above iterative algorithm would provide an exponential speed-up with respect to K. However, it is worth pointing out that automatically building a model from a data set is, in general, a difficult task (if possible at all).

As another concrete example we can consider the objective function (C1) and a routine that looks for the periodicity of the data and then generates new test solutions accordingly. After we go through multiple generations, it is likely that the population would converge to the correct periodic subset that has the minimum $f(x)$.



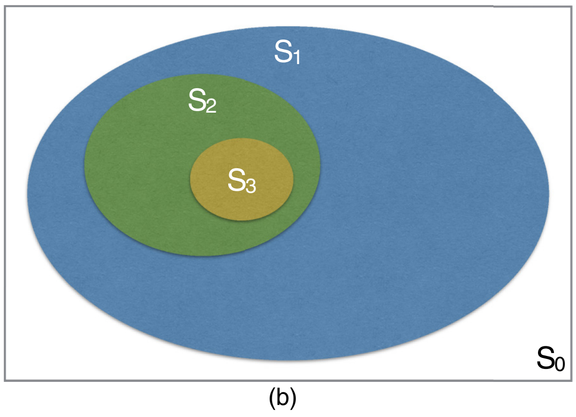


FIG. 4. Outline of our algorithm: (a) demonstrates that if we can model the distribution correctly, then we will be able to sample from good solutions more efficiently [red (darker) points correspond to smaller $f(x,y)$] and (b) illustrates the idea of concatenating the step performed in (a) in order to achieve an exponential speed-up compared to random search.

4. Summary

As seen in the discussion above, each of these optimization methods has its strong and weak points. Thus different methods are chosen depending on the prior knowledge we have on the concrete problems. It should be emphasized that we should not consider these methods as in a pure competition; instead, they can be used in complement with each other. For example, stochastic gradient Langevin dynamics (SGLD) [42] can be viewed as a combination of gradient descent and annealing, and in [43] it is mentioned that inclusion of the deterministic hill climber (discrete version of gradient descent) can lead to a substantial speed-up in the PMBGA.

APPENDIX D: MACHINE LEARNING

This section will give a brief overview over the subfield of machine learning known as supervised learning and introduce a model for time-series data, known as recurrent neural networks. Furthermore, some aspects of the optimization of this class of models will be elaborated on.

1. Supervised learning

The field of machine learning can be divided into three main subfields: supervised learning, unsupervised learning, and reinforcement learning. These branches differ from each other by the way in which the respective models obtain information about the utility of their generated outputs.

In the case of supervised learning, it is assumed that for every input that a model will be trained on, a "supervisor" provides a target, corresponding to the desired output of the model for the given input. These pairs of inputs and desired outputs are then used to make the model learn the general mapping between input and output.

More formally and from a Bayesian perspective, one assumes to have a data set D of size N, consisting of several tuples of independent and identically distributed observations $x \in \mathbf{R}^l$ and corresponding targets $y \in \mathbf{R}^k$ such that

$$D = \left\{(x_i, y_i)\big|_{i=1}^{N}\right\},$$

where x_i and y_i are instances of two random variables X and Y, respectively. These random variables are assumed to be distributed according to some unknown probability distribution p_{gen}, the so-called data-generating distribution

$$X, Y \sim p_{\text{gen}}(X, Y).$$

The goal of any supervised learning method now is to approximate the conditional distribution $p_{\text{gen}}(Y|X)$ in a way that allows for evaluation in some new observation $x_* \notin \{x_i\}|_{i=1}^{N}$. Since p_{gen} is not available, one resorts to fitting the empirical distribution p_{emp} given by D as a surrogate problem.

A typical way of deriving a concrete optimization problem from this is to make an assumption regarding the form of p_{gen} and treating the model at hand as a distribution $p_M(Y|X,\Theta)$ of this kind, parametrized by the parameters of the model Θ that are also often called the weights of the model. Now the fitting of the model can be perceived as a maximum-likelihood problem and hence the supervised learning problem can be formulated as

$$\max_{\Theta} \mathcal{L}(\Theta|D) = \max_{\Theta} \prod_i p_M(y_i|x_i,\Theta),$$

making use of the independent and identically distributed assumption. A commonly employed trick to obtain a more benign optimization problem is to instead optimize the negative log-likelihood. As the logarithm is a monotonic function, this transformation does not change the location of the optimum in the error landscape, but turns the product of probabilities into a sum over the tuples in D. This step then yields a minimization problem, given by

$$\min_{\Theta} -\frac{1}{N} \sum_i \log_2 p_M(y_i|x_i,\Theta),$$

which is also called empirical risk minimization (ERM). These statements of the problem can now be tackled with the optimization methods appropriate for the given model. In the case of the RNN, gradient-based optimization is the state-of-the-art approach and will be explained in Sec. V.

While it is obvious that fitting a model with respect to p_{emp} is identical to fitting it to p_{gen} as long as every tuple in D is only considered once, this is not necessarily true anymore when considering each tuple multiple times. This however is needed by many models in order to fit their parameters to a satisfying degree. In order to prevent the model from learning characteristics of the empirical distribution that are not present in the data-generating distribution, a phenomenon commonly known as overfitting, often some form of regularization, is applied. This may be done by punishing too large parameter values, stopping the training after performance starts to decrease on some holdout data set or by averaging over multiple models. Note that in the Bayesian picture some penalty terms can be perceived as the logarithm of a prior distribution over Θ, hence turning the optimization problem into finding the maximum *a posteriori* parameters.

2. Recurrent neural networks

In this section the recurrent neural network model will be discussed. We will start with an introduction of the standard version of the model and based upon this explain the advanced version of the model employed in this work in a second step.

a. Standard RNN model

In many areas of application, the data can be perceived as, often non-Markovian, discrete time-series data, such that an observation $x_t \in \mathbb{R}^l$ at some time t depends on the previous observations $x_{t-1}, \ldots, x_1$ or with respect to the framework introduced above,

$$X_t \sim p(X_t|X_{t-1}, \ldots, X_1).$$

While Markov chains have been the state-of-the-art approach for this kind of data in recent decades, with the recent rise of artificial neural networks, RNNs [44,45] have also gained momentum and are now generally considered to be the most potent method.

An RNN is defined by the two nonlinear maps $s_t : \mathbb{R}^l \to \mathbb{R}^h$ and $o_t : \mathbb{R}^h \to \mathbb{R}^o$ given by

$$s_t = f_s(Ux_t + Ws_{t-1} + b_s),$$
$$o_t = f_o(Vs_t + b_o),$$

where $U \in \mathbb{R}^{h\times l}$, $W \in \mathbb{R}^{h\times h}$, $V \in \mathbb{R}^{o\times h}$, $b_s \in \mathbb{R}^{1\times h}$, $b_o \in \mathbb{R}^{1\times o}$, and the trainable parameters of the models are constituted by $\Theta = \{U,V,W,b_s,b_o\}$. The nonlinear function f_s is often chosen to be tanh, the rectifier function given by

$$\text{rect}(x) = \max(0,x),$$

or the sigmoid function given by

$$\text{sigm}(x) = \frac{1}{1+e^{-x}}.$$

The function f_o must be chosen according to the distribution that is to be approximated by the model. For the case of a multinoulli distribution as assumed in this work, the corresponding function would be the softmax, defined as

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_k e^{x_k}},$$

the superscripts in this case denoting the single elements of the vector x.

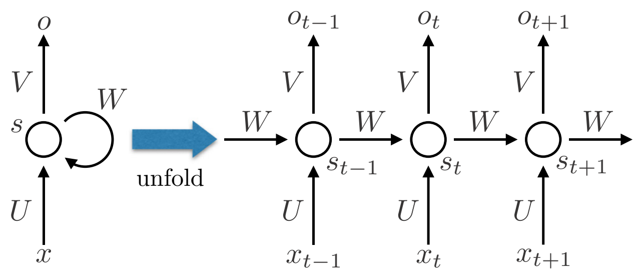


FIG. 5. Standard model of a recurrent neural network shown for three time steps.

The intuition behind this simple model is that it combines its information about the input at a given time step with a memory of the previous inputs, referred to as the state of the network. The precise nature of this combination and the state depends on the weight matrices U and V and the bias vector b_s. The combined information is then used as input of the chosen nonlinear function f_h to generate the next state. From this state, the output o_t is then computed as defined by W, b_o, and f_o. The effect of an RNN acting on the sequence $\{x_t\}$ is illustrated in Fig. 5.

From the above explanation, it is clear that the power of the model depends strongly on the size of the hidden state h. It should however also be noted that another effective way of increasing the expressive power of an RNN is to construct a composition of multiple functions of the form of s_t (see Fig. 6). In the machine learning terminology, the respective functions are called the layers of an artificial neural network and the number of composed functions is referred to as the depth of a network. The layers between the input and the output are referred to as hidden layers. The common intuitive reasoning behind stacking multiple layers is that it will allow the network to learn a hierarchy of concepts, called features, from the initial input data. Thereby, the features are assumed to be of increasing complexity with every layer, as they are based on a linear combination of the features learned by the layer below. Apart from this intuitive reasoning, also more rigorous work on the benefits of using at least one hidden layer between input and output can be found in the literature [46–48]. This ansatz of increasing the power of neural network models via deepening their architecture is publicly known as deep learning and has led to a drastic increase in success of machine learning methods during the past decade. However, having a composition of many state-computing functions of similar size can slow down the optimization process. This is why, when forming such a composition, each pair of functions is often connected via a simple linear projection from the space of the state of the earlier function onto some lower-dimensional space that is then used by the following function. Note that while all the above claims seem natural and lead to a good enough performance for our paper, more benchmarking is needed to really confirm them.

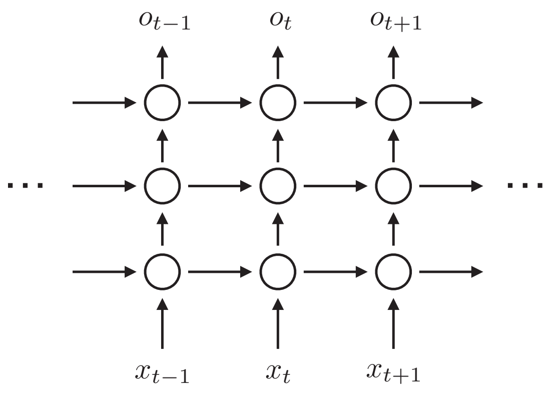


FIG. 6. Illustration of an RNN with three hidden layers.

Now, in the case of supervised learning, one assumes to be in possession of a set of time series $x_1, \ldots, x_n$ that will be used to let the RNN learn to predict series of this kind. The natural way of doing this is to define the pairs $(x_i, y_i) := (x_t, x_{t+1})$. While in principle the model is capable of taking into account all previous time steps, in practice it shows that optimization is only feasible for a relatively short number of steps. This is mainly due to the fact that the gradients that are needed to optimize the parameters of an RNN tend to grow to infinity or zero for higher numbers of steps. This will be discussed more in depth below.

b. Long short-term memory networks

In order to improve upon the standard RNN, Hochreiter and Schmidhuber introduced the long short-term memory network [49], which provides a different way of computing the state of an RNN. Hence the following set of equations can be perceived as a replacement for s_t from the previous section. The main advantage of the approach is that it drastically mitigates the problem of unstable gradients by construction. It is defined by the following set of equations:

$$\begin{aligned}
i_t &= \mathrm{sigm}(U^i x_t + W^i s_{t-1} + b^i),\\
f_t &= \mathrm{sigm}(U^f x_t + W^f s_{t-1} + b^f),\\
o_t &= \mathrm{sigm}(U^o x_t + W^o s_{t-1} + b^o),\\
\tilde{c}_t &= \tanh(U^{\tilde{c}} x_t + W^{\tilde{c}} s_{t-1} + b^{\tilde{c}}),\\
c_t &= c_{t-1} * f_t + \tilde{c}_t * i_t,\\
s_t &= \tanh(c_t) * o_t,
\end{aligned} \tag{D1}$$

where again x_t is the input at time step t, s_{t-1} is the previous state of the network, and c_t is the state of the cell. In addition, $U^i, U^f, U^o, U^{\tilde{c}} \in \mathbb{R}^{h\times l}$, while $W^i, W^f, W^o, W^{\tilde{c}} \in \mathbb{R}^{h\times h}$, $b^i, b^f, b^o, b^{\tilde{c}} \in \mathbb{R}^{1\times h}$, and $*$ denotes the elementwise multiplication.

As it can be seen from the equations, the way in which an LSTM computes the state is a bit more involved. If needed, it may however just be treated as a black box and can be stacked just in the same manner as it was described for the plain RNN model. The general idea of an LSTM is to give the model a higher degree of control over the information that is propagated from one time step to the next. This is achieved by making use of so-called gates that control the information flow to and from the network and cell state. These gates, by taking into account the previous state and the new input, output vectors of values in [0,1] that determine how much information they let through. In the equations given above, i_t is called the input gate, f_t is

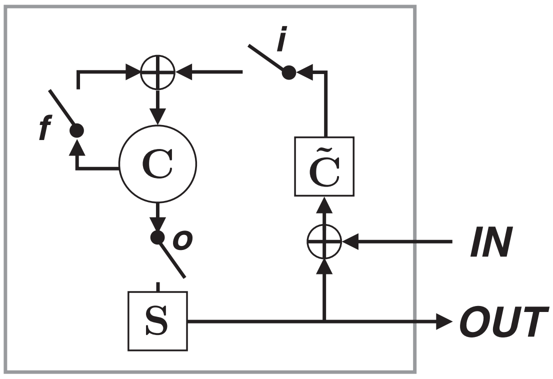


FIG. 7. Long short-term memory model illustrated in a schematic way. In addition to the diagram, the input gate i, the forget gate f, and the output gate o all depend on the current input x_t and previous state of the network s_{t-1}, as described in (D1).

referred to as the forget gate, and o_t denotes the output gate. Now, the mechanism works as follows.

(i) For a given time step t, the new input and previous network state are processed by $\tilde{c}_t$ like for the standard RNN and the output values are squashed to the interval $[-1,1]$ to yield candidate values for the next cell state.

(ii) The input gate i_t determines how to manipulate the information flow from the candidate cell state. Likewise, the forget gate f_t determines how to affect the information flow from the old cell state. The gated previous cell state and the gated input are then added to form the new cell state c_t.

(iii) Finally, the output gate o_t determines what to output from the new cell state. The new cell state is then also projected onto the interval $[-1,1]$ and put through the output gate to become the network state.

The whole process is shown in Fig. 7.

Naturally, there exists a plethora of possibilities to adapt the normal LSTM as explained above. One important enhancement is commonly referred to as peepholes, which allows the gates to incorporate the cell state via an extra term in the sum, in addition to the input and the network state. One other popular possibility introduced in [50] is the use of projection layers between different time steps of LSTM. In this case, we replace s_{t-1} by r_{t-1} in the equations for i_t, f_t, o_t, and $\tilde{c}_t$ and add the simple equation

$$r_t = W^p s_t,$$

where $W^p \in \mathcal{R}^{k\times h}$ is the projection matrix. In this work we have made use of both of these extensions of the normal LSTM. For an exhaustive overview over the known variants of the LSTM, we refer the interested reader to [35].

3. Optimization of RNNs

As the optimization problem described in the beginning of this section cannot be solved analytically for the models considered in this work, gradient-based approaches have established themselves as the state of the art. However, in the case of fitting the parameters of neural network models, three main restrictions need to be accounted for.

(a) The number of parameters for neural network models easily exceeds 100 000 and can for larger architectures go up to several tens or even hundreds of millions. Hence, computing the Hessian (or its inverse) explicitly is not tractable and so one is limited to first-order or approximative second-order methods.

(b) As the error function that is minimized is only a surrogate error function, its global optimum is not necessarily the optimum of the error function one actually wants to minimize.

(c) For many real-world data sets, computing the gradient of the complete sum of the error function over all samples is not feasible. Hence, the sum is normally split up into smaller parts called mini batches and these batches are looped over. A complete loop over D is then called an epoch.

These restrictions have led to the rise of an own subfield of machine learning that is concerned with the parallelization of gradient computations in the mini batch case, the approximation of second-order information, and the formal justification for the splitting up of the error function. All of the currently available methods are nevertheless extensions of the simplest method for gradient-based optimization known as steepest gradient descent: At iteration i in the loop over the batches, the parameters Θ are updated according to

$$\Theta_{i+1} = \Theta_i - \gamma \frac{\partial \mathcal{E}}{\partial \Theta_i},$$

where $\mathcal{E}(D,\Theta)$ is the respective error function and γ is called the step rate. The most straightforward natural adaption is to make γ depend on the iteration and slowly decrease it over time, following the intuition that smaller steps are beneficial the closer one gets to the respective optimum. In addition to that, many methods employ some kind of momentum term [51] or try to approximate second-order information and scale the gradient accordingly [39].

Besides this, the size of the batches also has an influence on the performance of the respective optimization method. In the extreme case where each batch only consists of one sample, the gradient descent method is known to converge almost surely to an optimum under certain constraints [52]. As picking individual samples for optimization can be perceived as sampling from the empirical distribution to approximate the overall gradient, this method is called stochastic gradient descent. Using single data points however is computationally inefficient and by definition leads to heavily oscillating optimization, so it is common practice to resort to larger batches. Following the ERM interpretation, batches B consisting of S_B samples are often used to compute an approximation of the mean gradient over D given by

$$\left\langle \frac{\partial \mathcal{E}}{\partial \Theta_i} \right\rangle_D \approx \left\langle \frac{\partial \mathcal{E}}{\partial \Theta_i} \right\rangle_B = \frac{1}{S_B} \sum_{(x,y)\in B} \frac{\partial \mathcal{E}_{(x,y)}}{\partial \Theta_i},$$

where obviously

$$\lim_{|B|\to|D|} \left\langle \frac{\partial \mathcal{E}}{\partial \Theta_i} \right\rangle_B = \left\langle \frac{\partial \mathcal{E}}{\partial \Theta_i} \right\rangle_D.$$

This interpretation is used, e.g., by the recently proposed algorithm Adam, which has been shown to yield very good local optima while being very robust with respect to noisy gradients and needing comparatively little adjustment of its parameters. We have employed Adam for fitting the models used in this work.

While the approach to optimizing artificial neural networks is well established, this does not change the fact that the optimization problems posed by them are inherently difficult. It is well known that the error landscape becomes less smooth the more layers one adds to a network. This results in error surfaces with large planes where $\frac{\partial\mathcal{E}}{\partial\Theta} \approx 0$ that are followed by short but very steep cliffs. If the step rate is not adapted correctly, the optimization procedure is very likely to get stuck in one these planes or saddle points and to jump away from an optimum in the vicinity of Θ if evaluated on one of the cliffs. The phenomena of the frequent occurrence of very large or very small gradients are referred to in the literature as the exploding gradient or vanishing gradient problem, respectively. To get a better understanding of why these problems exist, it is instructive to examine how the gradients for a given model are obtained.

As has been explained above, multilayer neural network models are a composition of nonlinear functions $\mathbb{R}^{i_k} \to \mathbb{R}^{o_k}$: $x_{k+1} = f_k(W_k x_k + b_k)$, where W_k is the weight matrix, b_k the bias vector, x_0 the input data, and x_K the final output of the network. From this definition it is clear that $o_k = i_{k+1}$. For convenience, we define $y_k \equiv W_k x_k + b_k$. In order to obtain the gradient for a specific W_k or b_k one must obviously make use of the chain rule such that

$$\frac{\partial\mathcal{E}}{\partial W_k} = \frac{\partial\mathcal{E}}{\partial x_{k+1}}\frac{\partial x_{k+1}}{\partial y_k}\frac{\partial y_k}{\partial W_k}$$

$$= \frac{\partial\mathcal{E}}{\partial x_K}\left(\prod_{j=k+1}^{K-1}\frac{\partial x_{j+1}}{\partial x_j}\right)\frac{\partial x_{k+1}}{\partial y_k}\frac{\partial y_k}{\partial W_k}$$

and

$$\frac{\partial\mathcal{E}}{\partial b_k} = \frac{\partial\mathcal{E}}{\partial x_{k+1}}\frac{\partial x_{k+1}}{\partial y_k}\frac{\partial y_k}{\partial b_k}$$

$$= \frac{\partial\mathcal{E}}{\partial x_K}\left(\prod_{j=k+1}^{K-1}\frac{\partial x_{j+1}}{\partial x_j}\right)\frac{\partial x_{k+1}}{\partial y_k}\frac{\partial y_k}{\partial b_k},$$

where $\frac{\partial}{\partial W_k}$ is the shortcut of doing the derivative elementwise:

$$\left[\frac{\partial}{\partial W_k}\right]_{ab} = \frac{\partial}{\partial[W_k]_{ab}}.$$

The same convention applies to $\frac{\partial}{\partial b_k}$. As $\frac{\partial}{\partial W_k}$ and $\frac{\partial}{\partial b_k}$ depend on all the gradients of the later layers, this formulation yields an efficient method of computing the gradients for all layers by starting with the uppermost layer and then descending in the network, always reusing the gradients already computed. Together with the fact that many of the commonly used nonlinearities have an easy closed-form expression of the first derivative, this allows for fully automatic computation of the gradients as it is done in every major deep learning framework. This dynamic programming method of computing the gradients is known in the literature as backpropagation. The vanishing (exploding) gradient problem arises because of the product $\prod_{j=k+1}^{K-1}\frac{\partial x_{j+1}}{\partial x_j}$ in the above equations. For example, if one of the $\frac{\partial x_{j+1}}{\partial x_j} \approx 0$ in the product, then likely we have $\frac{\partial\mathcal{E}}{\partial W_k} \approx 0$, which leads to an ineffective gradient descent. Similarly, if many of the terms $\frac{\partial x_{j+1}}{\partial x_j}$ have large norms, then there is a possibility that $\frac{\partial\mathcal{E}}{\partial W_k}$ becomes too large, which often causes the optimization method to jump out of a local optimum.

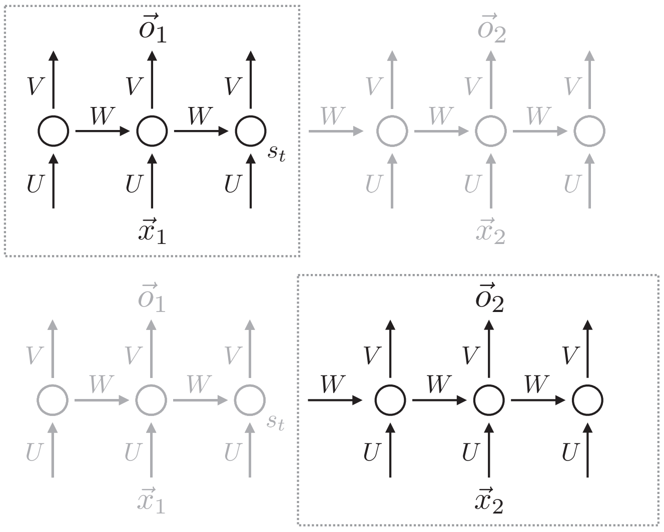


FIG. 8. Illustration of how we truncate the gradient computation for long sequences. Here we divide the sequences into two halves. As the first step, we compute the gradient of the error function $\mathcal{E}(\vec{x}_1,\vec{o}_1)$ with respect to the parameters U,V,W, while ignoring the other half of the network. In the second step, we compute the gradient of $\mathcal{E}(\vec{x}_2,\vec{o}_2)$, while treating the final state of the network s_t of the first half as a constant. The final gradients are approximated by the sums of these two constituents. Thus, we are able to avoid the instability of computing gradients, but still capture the correlation between two halves, since we feed the final network state s_t into the second half.

In the case of an RNN as defined in Sec. V, the above generic equations for the derivative become a little more involved, as in addition to the term for possibly multiple stacked layers, a term accounting for states of previous times has to be added. Nevertheless, at the heart of the problem, it is still about computing derivatives of composite functions. This slightly more involved backpropagation method is known as backpropagation through time and can also be fully automatized. Similar to the multilayer neural network models mentioned above, the gradient computation of RNNs also has these instability issues. As can be seen from Fig. 5, the same matrix W is used in all time steps of an RNN. Thus, a tiny change of W could affect the output o_t drastically when the time step t gets big. In other words, the derivative of the error function $\mathcal{E}$ with respect to W could again become very large or very small in certain situations. To deal with this issue, we could truncate the number of time steps during the computation, as described in Fig. 8. More discussion on this topic can be found in Sec. 3.2 of [32].

APPENDIX E: TECHNICAL ASPECTS

For the implementation of this work, we have made use of PYTHON with the numerical libraries NumPy, SciPy, and TensorFlow [53–55]. All experiments were run on single workstations with up to eight threads and a GeForce Titan X. The runtime of the experiments varied, depending on the optimization parameters, from a few hours to days.

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, in *Proceedings of the Conference on Advances in Neural Information Processing Systems 25*, edited by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (MIT Press, Cambridge, MA, 2012), pp. 1097–1105.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, arXiv:1312.5602.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Nature (London) **529**, 484 (2016).

[4] R. S. Judson and H. Rabitz, Phys. Rev. Lett. **68**, 1500 (1992).

[5] U. Las Heras, U. Alvarez-Rodriguez, E. Solano, and M. Sanz, Phys. Rev. Lett. **116**, 230504 (2016).

[6] I. Geisel, K. Cordes, J. Mahnke, S. Jöllenbeck, J. Ostermann, J. Arlt, W. Ertmer, and C. Klempt, Appl. Phys. Lett. **102**, 214105 (2013).

[7] Ł. Pawela and P. Sadowski, Quantum Inf. Process. **15**, 1937 (2016).

[8] M. Grace, C. Brif, H. Rabitz, I. A. Walmsley, R. L. Kosut, and D. A. Lidar, J. Phys. B **40**, S103 (2007).

[9] M. Krenn, M. Malik, R. Fickler, R. Lapkiewicz, and A. Zeilinger, Phys. Rev. Lett. **116**, 090405 (2016).

[10] C. Chen, L.-C. Wang, and Y. Wang, Sci. World J. **2013**, 869285 (2013).

[11] D. Dong, C. Chen, B. Qi, I. R. Petersen, and F. Nori, Sci. Rep. **5**, 7873 (2015).

[12] L. Banchi, N. Pancotti, and S. Bose, npj Quantum Inf. **2**, 16019 (2016).

[13] S. Machnes, U. Sander, S. J. Glaser, P. de Fouquières, A. Gruslys, S. Schirmer, and T. Schulte-Herbrüggen, Phys. Rev. A **84**, 022305 (2011).

[14] D. J. Egger and F. K. Wilhelm, Phys. Rev. Lett. **112**, 240503 (2014).

[15] M. J. Biercuk, H. Uys, A. P. VanDevender, N. Shiga, W. M. Itano, and J. J. Bollinger, Nature (London) **458**, 996 (2009).

[16] P. Doria, T. Calarco, and S. Montangero, Phys. Rev. Lett. **106**, 190501 (2011).

[17] J. Kelly *et al.*, Phys. Rev. Lett. **112**, 240504 (2014).

[18] P. B. Wigley, P. J. Everitt, A. van den Hengel, J. W. Bastian, M. A. Sooriyabandara, G. D. McDonald, K. S. Hardman, C. D. Quinlivan, P. Manju, C. C. N. Kuhn, I. R. Petersen, A. N. Luiten, J. J. Hope, N. P. Robins, and M. R. Hish, Sci. Rep. **6**, 25890 (2016).

[19] J. Combes, C. Ferrie, C. Cesare, M. Tiersch, G. J. Milburn, H. J. Briegel, and C. M. Caves, arXiv:1405.5656.

[20] D. Orsucci, M. Tiersch, and H. J. Briegel, Phys. Rev. A **93**, 042303 (2016).

[21] M. Tiersch, E. J. Ganahl, and H. J. Briegel, Sci. Rep. **5**, 12874 (2015).

[22] L. Viola, E. Knill, and S. Lloyd, Phys. Rev. Lett. **82**, 2417 (1999).

[23] A. M. Souza, G. A. Alvarez, and D. Suter, Phys. Rev. Lett. **106**, 240501 (2011).

[24] G. Quiroz and D. A. Lidar, Phys. Rev. A **88**, 052306 (2013).

[25] G. de Lange, Z. H. Wang, D. Ristè, V. V. Dobrovitski, and R. Hanson, Science **330**, 60 (2010).

[26] A. Karpathy, J. Johnson, and L. Fei-Fei, arXiv:1506.02078.

[27] Z. C. Lipton, J. Berkowitz, and C. Elkan, arXiv:1506.00019.

[28] C. A. Ryan, J. S. Hodges, and D. G. Cory, Phys. Rev. Lett. **105**, 200402 (2010).

[29] M. D. Grace, J. Dominy, R. L. Kosut, C. Brif, and H. Rabitz, New J. Phys. **12**, 015001 (2010).

[30] A. M. Souza, G. A. Álvarez, and D. Suter, Philos. Trans. R. Soc. A **370**, 4748 (2012).

[31] C. Shannon, Bell Syst. Tech. J. **27**, 379 (1948).

[32] A. Graves, arXiv:1308.0850.

[33] J. B. Pollack, On connectionist models of natural language processing, Ph.D. thesis, New Mexico State University, 1987.

[34] M. A. Nielsen, *Neural Network and Deep Learning* (Determination Press, 2015).

[35] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, arXiv:1503.04069.

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016).

[37] M. Pelikan, D. E. Goldberg, and F. G. Lobo, Comput. Optim. Appl. **21**, 5 (2002).

[38] A. P. Dempster, N. M. Laird, and D. B. Rubin, J. R. Stat. Soc. Ser. B **39**, 1 (1977).

[39] D. Kingma and J. Ba, arXiv:1412.6980.

[40] A. M. Souza, G. A. Álvarez, and D. Suter, Phys. Rev. A **85**, 032306 (2012).

[41] M. Pelikan, in *Hierarchical Bayesian Optimization Algorithm* (Springer, Berlin, 2005), pp. 31–48.

[42] M. Welling and Y. W. Teh, in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (ICML, Bellevue, 2011), pp. 681–688.

[43] M. Pelikan and A. K. Hartmann, in *Scalable Optimization via Probabilistic Modeling* (Springer, Berlin, 2006), pp. 333–349.

[44] R. J. Williams and D. Zipser, Neural Comput. **1**, 270 (1989).

[45] P. J. Werbos, Proc. IEEE **78**, 1550 (1990).

[46] Y. Bengio and Y. LeCun, in *Large-Scale Kernel Machines*, edited by L. Bottou, O. Chapelle, D. DeCoste, and J. Weston (MIT Press, Cambridge, MA, 2007), Chap. 14.

[47] K. Hornik, M. Stinchcombe, and H. White, Neural Networks **2**, 359 (1989).

[48] K. Hornik, Neural Networks **4**, 251 (1991).

[49] S. Hochreiter and J. Schmidhuber, Neural Comput. **9**, 1735 (1997).

[50] H. Sak, A. Senior, and F. Beaufays, arXiv:1402.1128.

[51] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (ICML, Bellevue, 2013), pp. 1139–1147.

[52] D. Saad, *On-line Learning in Neural Networks* (Cambridge University Press, Cambridge, 2009), Vol. 17.

[53] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, Comput. Sci. Eng. **13**, 22 (2011).

[54] E. Jones, T. Oliphant, and P. Peterson (unpublished).

[55] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, software available from TensorFlow.org.