

Efficient Finite Initialization with Partial Norms for Tensorized Neural Networks and Tensor Networks Algorithms

Alejandro Mata Ali*

i3B Ibermatica Fundazioa, Quantum Development Department, Paseo Mikeletegi 5, 20009 Donostia, Spain

Iñigo Perez Delgado[†] and Marina Ristol Roura[‡]
*i3B Ibermatica Fundazioa, Parque Tecnológico de Bizkaia,
 Ibaizabal Bidea, Edif. 501-A, 48160 Derio, Spain*

Aitor Moreno Fdez. de Leceta[§]
*i3B Ibermatica Fundazioa, Unidad de Inteligencia Artificial,
 Avenida de los Huetos, Edificio Azucarera, 01010 Vitoria, Spain*
 (Dated: July 4, 2025)

We present two algorithms to initialize layers of tensorized neural networks and general tensor network algorithms using partial computations of their Frobenius norms and lineal entrywise norms, depending on the type of tensor network involved. The core of this method is the use of the norm of subnetworks of the tensor network in an iterative way, so that we normalize by the finite values of the norms that led to the divergence or zero norm. In addition, the method benefits from the reuse of intermediate calculations. We have also applied it to the Matrix Product State/Tensor Train (MPS/TT) and Matrix Product Operator/Tensor Train Matrix (MPO/TT-M) layers and have seen its scaling versus the number of nodes, bond dimension, and physical dimension. All code is publicly available.

Keywords: Machine learning, Tensor networks, Quantum-inspired

I. INTRODUCTION

Deep neural networks [1] are widely used in machine learning to achieve good results in industrial, research, and various other applications. This good performance has led to its use in more complex contexts, requiring a larger number of parameters. Consequently, various architectures have been utilized to enhance its performance. The greatest example are *Large Language Models* (LLM) [2], which make use of an extreme number of parameters, requiring large devices to use them. The memory requirements are a huge limitation in future applications and the scalability of the current line of progress in artificial intelligence.

With the advent of quantum computing applied to various fields, interest in quantum information compression methods [3] has increased due to the exponential capacity of quantum systems to handle information. This is one of the reasons for the *Quantum Machine Learning* field. However, the current limitation of quantum hardware does not allow one to implement most of the desired algorithms and models that quantum computing offers.

In this context, researchers have explored different classical techniques which could take advantage of the properties of quantum systems, the *quantum-inspired*. This allows some of the advantages of quantum computing without the need of a quantum device to implement

the operations. One of the best known quantum-inspired techniques is the *Tensor networks* [4, 5], which are graphical representations of tensor algebra calculations. Tensor networks have a great capacity to “compress” tensor information by means of representations such as *Matrix Product State/Tensor Trains* (MPS/TT) [6] or *Projected Entangled-Pair States* (PEPS) [7]. This allows us to retain the most important information of the represented tensors with several fewer parameters. This compression has been applied to several machine learning models in various ways, such as decomposing matrices into tensor networks [8, 9]. It has been applied to *neural networks* [10], *convolutional neural networks* [11], *transformers* [12], *spiking neural networks* [13] or LLM [14, 15]. Tensor networks have also been used as the main model, directly training the tensor network itself [16, 17] (Fig. 1).

Our focus of analysis will be on methods in which a large tensor network is generated to model the layer and trained directly, rather than using a trained dense matrix and compressing it. For example, when we try to use *tensorized physics-informed neural networks* [18] or tensor neural networks to solve differential equations [19] for large industrial cases, such as the heat equation of an engine or fluids in a turbine. In this type of cases, an initialization problem is often encountered [20], the explosion or vanishing of the represented tensor values. If we initialize the elements of each tensor of the tensor network with a certain distribution, when we contract the tensor network to obtain the tensor it represents, some of its final elements are too large (infinite) or too small (null) for the computer. This problem may also arise in some quantum machine learning algorithms in which the

* alejandro.mata.ali@gmail.com

† iperezde@ayesa.com

‡ mristol@ayesa.com

§ aitor.moreno@lksnext.com

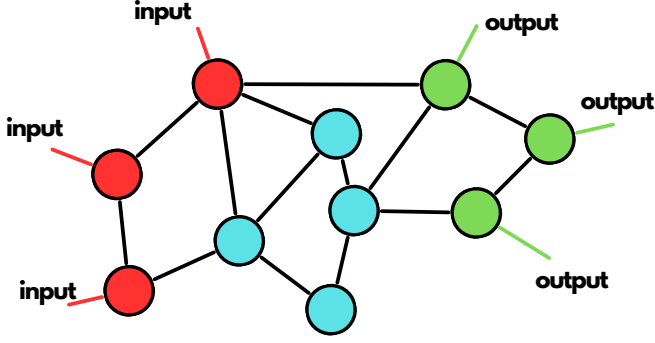


FIG. 1. Arbitrary tensor network layer.

aim is to compress a classical machine learning model.

If we want to eliminate these problems, a first proposal could be to contract the tensor network and eliminate or reduce these elements. However, in very large layers we cannot store all the final tensor elements in memory, so it is not possible in this way. In addition, we may not know how to properly rescale those divergent elements without damaging the other ones. One way is to reinitialize the tensor network by changing a distribution with better hyperparameters, changing the mean and standard deviation, or rescaling them [20]. However, many of these methodologies are not easy to apply in all cases in an efficient way.

In this work, we present two novel algorithms to address this problem in two different scenarios. The first is applicable to tensor networks of general values, which consists of iteratively calculating the *Frobenius norm* for different sections of the tensor network until a condition is met, when we divide all the parameters of the tensor network by the factor calculated in a particular way. This allows us to gradually make the Frobenius norm of the layer tend to the number we want, without having to repeatedly re-initialize. We call this method the *Frobenius Tensor Network Renormalization* (FTNR). The second method applies to tensor networks with all positive values, where the process is analogous to that of the previous method but involves calculating the partial *lineal entry-wise norm* of reduced forms. We call this method the *Lineal Tensor Network Renormalization* (LTNR).

These algorithms are remarkably interesting for hierarchical tree form layers, especially in the *Tensor Train* (TT), *Tensor Train Matrix* (TT-M), and *Projected Entangled Pair States* (PEPS). This can also be used in other methods with tensor networks, such as combinatorial optimization, to determine hyperparameters, and it can be combined with other initialization methods.

The main contributions of this work are two efficient algorithms to initialize a tensorized artificial intelligence model, or other tensor network algorithms, without an explosion or vanishing of the parameters, for large layers. This work is structured as follows. First, in Sec. II we will describe properly the main problem we want to

solve, and we will introduce a brief background on existing algorithms to deal with this. Then, in Secs. III and IV, we will introduce our new algorithms for general and positive scenarios, respectively. Additionally, in Sec. V we will perform several experiments to investigate the limitations of the algorithms. Finally, in Sec. VI we will analyze other possible applications of these algorithms.

We created a Python function to run it on an arbitrary PyTorch layer, available in a Jupyter Notebook in the GitHub repository https://github.com/DOKOS-TAYOS/Efficient_Initialization_Tensor_Networks

II. DESCRIPTION OF THE PROBLEM AND BACKGROUND

When we have a tensor network of N nodes, we will find that the elements of the tensor representing the tensor network are given by the sum of a set of values, each given by the product of N elements of the different nodes. If we examine the case of an TT layer, as in Fig. 2.a, the elements of the layer are given as

$$T_{ijklm} = \sum_{nopq} T_{in}^0 T_{njo}^1 T_{ojp}^2 T_{pq}^3 T_{qm}^4. \quad (1)$$

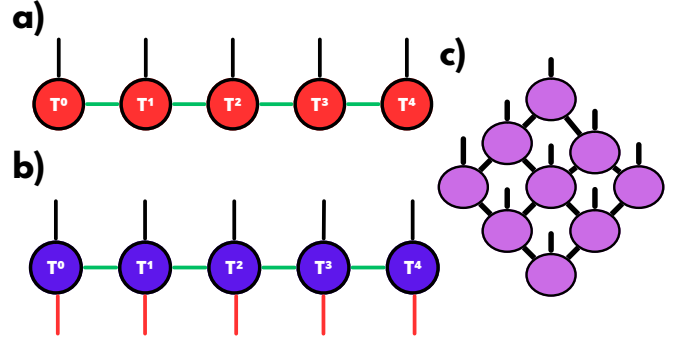


FIG. 2. a) Tensor Train layer with 5 indexes. b) Tensor Train Matrix layer with 10 indexes. c) PEPS layer with 9 indexes.

We see that for 5 indexes in the tensor we have to multiply 5 tensor elements, but in the general case with N indexes we have

$$T_{i_0 i_1 \dots i_{N-1}} = \sum_{j_0 j_1 \dots j_{N-2}} T_{i_0 j_0}^0 T_{j_0 i_1 j_1}^1 \dots T_{j_{N-2} i_{N-1}}^{N-1}, \quad (2)$$

multiplying N elements of the tensors T^i to obtain the element of the tensor T .

To exemplify the problem we want to solve, let us think about this. For a general case with bond dimension b , the dimension of the index that is contracted between each pair of nodes, N nodes, and constant elements of the nodes with value a , we would have the following result for the represented tensor elements

$$T_{i_0 i_1 \dots i_{N-1}} = b^{N-1} a^N. \quad (3)$$

We can see that with $N = 20$ nodes, an element value of $a = 1.5$ and a bond dimension of $b = 10$, the final tensor elements would be $3.3 \cdot 10^{22}$. This is a very large element for a good initialization. This is what we call the “explosion” of the tensor values. On the other hand, if the value of the element was $a = 0.01$, the tensor elements would be 10^{-20} . This is the “vanishing” of the tensor values. However, if we were to divide the values of these tensors by that number, we could arrive at the case where a^N was a number too large or small for our computer to store, and we would get a 0 or infinite in all the renormalized elements.

This problem is exacerbated by the number of nodes in the layer, since each one is a product. Moreover, we cannot simply calculate these tensor elements for cases with many physical indexes, the output indexes. This is because the number of values to be kept in memory increases exponentially with the number of physical indexes. There is also the possibility that, instead of initializing the tensors with a similar distribution, we initialize them with different distributions, making some nodes in the network have smaller elements than other nodes in the network. However, this approach could lead us to problems with model training.

Tensor network architectures often entail a trade-off between expressivity and trainability, strongly influenced by the initial values of tensor cores. For conventional dense networks, random initialization [21] is enough, making use of the *Central Limit Theorem*. However, TN models require more complex strategies due to the multilinear nature of the tensor contractions and the role of bond dimensions, as we noted in the previous section.

The first approach is developed in [22], where they prepare the initialization of a TT for probabilistic models, as a “warm-start”. This technique allows to obtain a model which avoids the causal problems. However, it makes use of the TT-cross approximation [23], so it cannot be easily generalizable. The second approach is to generate unitary matrices from Haar measure distribution [24], because they preserve the norms. However, this has a limitation for general structures where unitarity is not enough, for example PEPS. The third approach is to connect some identity matrices with a small random noise term [25]. However, the authors indicate that this is a use-case specific technique.

III. GENERAL TENSOR NETWORK INITIALIZATION PROTOCOL FTNR

We first address the case of a general real value tensor network, where we have initialized each node with a similar distribution, allowing the elements of the nodes to be on the same scale with respect to each other. Our protocol is based on the use of partial Frobenius norms to normalize the total Frobenius norm of the resulting tensor.

The *Frobenius norm* of a matrix is given by the equa-

tion

$$\|A\|_F = \sqrt{\sum_{ij} |a_{ij}|^2} = \sqrt{\text{Tr}(A^\dagger A)}. \quad (4)$$

In a tensor network, this would be to contract the layer with a copy of itself so that each physical index is connected to the equivalent of its copy. We can see some examples in Fig. 3.

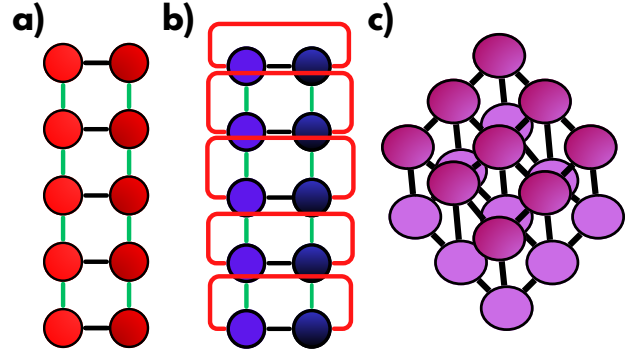


FIG. 3. Square of the Frobenius norm calculated to a) Tensor Train layer. b) Tensor Train Matrix layer. c) PEPS layer.

The contraction of this tensor network is equivalent to the Frobenius norm of the matrix it represents. In addition, it can be computed without the need to calculate the elements of the represented matrix, using only the elements of the nodes.

The Frobenius norm is an indicator that serves to regularize layers of a model [26] and gives an estimate of the order of magnitude of the matrix elements. With a smooth distribution of elements around a_{00} , the norm in Eq. (4) will be of the order of $\sqrt{nm} |a_{00}|$ for a matrix $n \times m$.

To avoid the elements of the layer from being too large or too small and therefore having too large or too small outputs in initialization, we will normalize these elements so that the Frobenius norm of the tensor is a number that we choose, for example, $\|A\|_F = 1$. This prevents our highest element from being higher than this value while taking advantage of the localized distribution of values to ensure that the smallest value is not too small.

Still, for a matrix $n \times m$, we will be summing nm values, so we should adjust the norm to be proportional to nm , the size of the problem, and thus do not decrease the magnitude of the values with it.

For this purpose, we define what we call the partial square norm of the tensor network.

A. Partial square norm of the tensor network

For the remainder of the paper, we assume that we can consistently sort the nodes of a tensor network so that they form a single growing network.

Definition 1 (Partial square norm)

Given a tensor network \mathcal{A} of N nodes, and a tensor network \mathcal{A}_n defined by the first n nodes of \mathcal{A} , the partial square norm ${}^{pF}\|\mathcal{A}\|_{n,N}$ at the n nodes of \mathcal{A} is defined as the square of the Frobenius norm of \mathcal{A}_n . That is,

$${}^{pF}\|\mathcal{A}\|_{n,N} = \|\mathcal{A}_n\|_F^2. \quad (5)$$

To get an idea of what this partial square norm is, we exemplify it with a simple case, a tensor train layer. We consider the tensor network in Fig. 4, whose nodes are sorted.

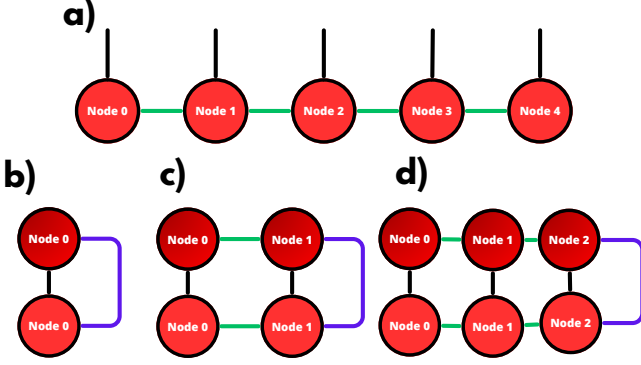


FIG. 4. a) Tensor Train layer with 5 nodes. b) Partial square norm at 1 node. c) Partial square norm at 2 nodes. d) Partial square norm at 3 nodes.

As we can see, in this case we would only have to do the same process as when calculating the total norm of the total tensor network, but stop at step n and contract the bond index of the two final tensors of the chain.

We can see in the following Figs. 5 and 6 how the partial square norm would be for a TT-Matrix layer and for a PEPS layer.

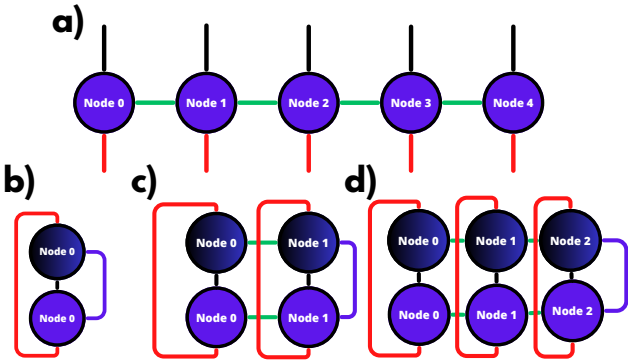


FIG. 5. a) Tensor Train Matrix layer with 5 nodes. b) Partial square norm at 1 node. c) Partial square norm at 2 nodes. d) Partial square norm at 3 nodes.

This calculation can be easily extended to general tensor networks, as long as we have a consistent ordering of the nodes.

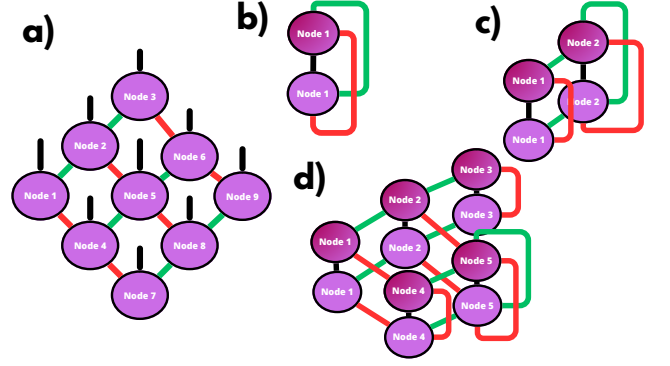


FIG. 6. a) PEPS layer with 9 nodes. b) Partial square norm at 1 node. c) Partial square norm at 2 nodes. d) Partial square norm at 5 nodes.

B. Frobenius Tensor Network Renormalization Initialization protocol

If we have a tensor network \mathcal{A} , representing a $n_A \times m_A$ matrix whose Frobenius norm $\|\mathcal{A}\|_F$ is infinite (larger than our computer can store), zero (smaller than our computer precision), or outside a certain range of values, we will want to normalize the elements of our tensor network so that the norm $\|\mathcal{B}\|_F$ of the new tensor network \mathcal{B} is equal to a given number F . We assume that the tensor represented by this tensor network has a smooth distribution in the values of its elements, due to a smooth distribution also of the elements of the nodes of its tensor network [20]. We also assume $F = n_A m_A$, but we see that another number can be easily chosen. To normalize the norm of the \mathcal{A} tensor with N nodes, we only have to divide the elements of each of its nodes by $\|\mathcal{A}\|_F^{1/N}$.

Since we cannot divide the elements by 0 or infinity, we use the following logic. If the total norm is infinite (zero), there will exist a value of $n < N$ such that ${}^{pF}\|\mathcal{A}\|_{n,N}$ is finite and non-zero such that ${}^{pF}\|\mathcal{A}\|_{n+1,N}$ is infinite (zero). This is because each step, we add a new node to the partial square norm, we multiply by a new value, so infinity (zero) will appear after a certain number of nodes, being the partial square norm with one node less a valid number to divide by.

The idea is to iteratively normalize the norm little by little so that we eventually achieve full normalization. To avoid having to start the iterative process from the beginning several times, we can take advantage of the fact that, if we normalize \mathcal{A}_n , all the subnetworks $\mathcal{A}_m, \forall m < n$ that compose it will also be normalized. Because of this, each time we normalize, we will not have to start from the network with one node, but we can continue with the network with n nodes.

For certain tensor networks, such as MPS, it is possible to reuse intermediate computations efficiently. For each normalization step, when we compute the partial norm, we first contract all the bond indexes that connect the nodes in the subnetwork and all the physical indexes.

Before contracting the bond indexes connecting nodes that are not present, we store the tensor provisionally. Thus, after renormalization, we can normalize both the individual tensor elements and this provisional tensor so that we do not have to contract the entire tensor network again. Furthermore, we can use this tensor for the next steps of normalization.

We want a tensor network \mathcal{A} with Frobenius norm F , with N nodes, and we set a tolerance range (aF, bF) , with $a \leq 1$, $b \geq 1$. The *Frobenius Tensor Network Renormalization* initialization protocol to follow would be:

1. We initialize the node tensors with some initialization method. We recommend random initialization with a Gaussian distribution of a constant standard deviation (not greater than 0.5) and a constant mean neither too high nor too low and positive.
2. We compute $\|\mathcal{A}\|_F$. If it is finite and non-zero, we divide each element of each node by $\left(\frac{\|\mathcal{A}\|_F}{F}\right)^{1/N}$ and return \mathcal{A} . Otherwise, we continue.
3. We compute ${}^{pF}\|\mathcal{A}\|_{1,N}$.
 - (a) If it is infinite, we divide each element of the nodes of \mathcal{A} by $(10(1 + \xi))^{1/2N}$, being ξ a random number between 0 and 1, and return to Step 2.
 - (b) If it is zero, we multiply each element of the nodes of \mathcal{A} by $(10(1 + \xi))^{1/2N}$ and return to Step 2.
 - (c) Otherwise, we save this value as ${}^{pF}\|\mathcal{A}\|_{1,N}$ and continue.
4. For $n \in [2, N - 1]$, we compute ${}^{pF}\|\mathcal{A}\|_{n,N}$.
 - (a) If it is infinite or zero, we divide each element of the nodes of \mathcal{A} by $({}^{pF}\|\mathcal{A}\|_{n-1,N})^{\frac{1}{2N}}$, and repeat Steps 2 and 4 (from this value of n).
 - (b) If it is finite, but larger than b or smaller than a , we divide each element of the nodes of \mathcal{A} by $\left(\frac{{}^{pF}\|\mathcal{A}\|_{n,N}}{F}\right)^{\frac{1}{2N}}$, and repeat Steps 2 and 4 (from this value of n).
 - (c) Otherwise, we continue.
5. If no partial square norm is outside the range, infinite or zero, we divide each element of the nodes of \mathcal{A} by $\left(\frac{{}^{pF}\|\mathcal{A}\|_{N-1,N}}{F}\right)^{\frac{1}{2N}}$, and repeat steps 2 and 5.

We repeat the cycle until we obtain a valid \mathcal{A} or we reach a stop condition, which entails repeating a certain maximum number of iterations. If we reach this last point, the protocol will have failed and we will have two options. The first is to change the order of the nodes so that other structures are checked. The second is to reinitialize with other hyperparameters in the initialization protocol.

The purpose of using a random factor in the case of divergence in the partial norm with one node is that, not knowing the real value by which we should divide, we rescale by an order of magnitude. However, to avoid possible infinite rescaling loops, we add a variability factor so that we cannot get stuck.

In case the elements of certain tensors of the tensor network have values smaller than or larger than those of the other tensors, there is a possibility that instead of evenly distributing renormalization among the N nodes with the exponent $\frac{1}{2N}$, we allow the other nodes to renormalize differently, equalizing the proportion of the values. We can also try to have as many integer values as possible in the values of the tensor elements by an adjusted renormalization of each tensor, or perform a quantization. These two processes can be performed either before, during, or after finishing the renormalization process, it being advisable to perform it at least at the end.

IV. POSITIVE TENSOR NETWORK INITIALIZATION PROTOCOL LTNR

This case deals with tensor networks in which all elements will be positive, which will allow us to apply computationally less expensive techniques. Our protocol is based on the use of a partial lineal entrywise norm to normalize the total lineal entrywise norm of the resulting tensor.

Definition 2 (Lineal entrywise norm)

The lineal entrywise norm of a matrix A with elements a_{ij} is given by

$$\|A\|_L = \sum_{ij} a_{ij} = \mathbb{1}^T A \mathbb{1}, \quad (6)$$

being $\mathbb{1} = (1, 1, \dots)$ a ones vector.

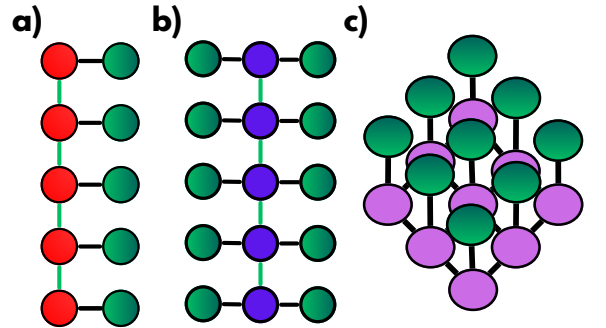


FIG. 7. Lineal entrywise norm calculated to a) Tensor Train layer (red nodes). b) Tensor Train Matrix layer (blue nodes). c) PEPS layer (purple nodes). The green nodes are ones vectors. Green nodes are ones vectors.

In a tensor network, this would be to contract the layer with a set of nodes of ones vectors. We can see some examples in Fig. 7. The contraction of this tensor network

is equivalent to sum all the elements of the matrix it represents. As in the case of the Frobenius norm, it can be computed without the need to calculate the elements of the represented matrix, using only the elements of the nodes.

In this case, we can interpret this rule in a manner analogous to the Frobenius norm in the general case, since all the elements are positive, we will have that

$$\|A\|_L = \sum_{ij} a_{ij} = \sum_{ij} |a_{ij}| = \sum_{ij} |\sqrt{a_{ij}}|^2 = \|A^{\circ \frac{1}{2}}\|_F. \quad (7)$$

Therefore, for positive value matrices, the lineal entrywise norm is equal to the Frobenius norm of the Hadamard root of the matrix. With a smooth distribution of elements around a_{00} , the norm in Eq. (6) will be of the order of nma_{00} for a $n \times m$ matrix.

As in the general case, the way to avoid values that are too large or too small is to normalize the matrix, but this time with respect to the lineal entrywise norm. Still, for a matrix $n \times m$, we will be summing nm values, so we should adjust the norm to be proportional to nm , the size of the problem, and thus do not decrease the magnitude of the values with it.

For this purpose, we define what we will call the partial lineal norm of the tensor network, analogous to the partial square norm of the Frobenius norm method.

A. Partial lineal norm of the tensor network

Definition 3 (Partial lineal norm)

Given a tensor network \mathcal{A} with N nodes, and the tensor network \mathcal{A}_n defined by the first n nodes of \mathcal{A} , we define ${}^pL\|\mathcal{A}\|_{n,N}$, the partial lineal norm at n nodes of \mathcal{A} as the lineal entrywise norm of \mathcal{A}_n . That is,

$${}^pL\|\mathcal{A}\|_{n,N} = \|\mathcal{A}_n\|_L. \quad (8)$$

To get an idea of what this partial lineal norm is, we will exemplify it with a simple case, a tensor train layer. We will consider the tensor network in Fig. 8, whose nodes are sorted.

As we can see, in this case we would only have to do the same process as when calculating the total norm of the total tensor network, but stop at step n and contract the bond index of the final tensor of the chain with a ones vector.

We can see in the following Fig. 9 and 10 how the partial lineal norm would be for a TT-Matrix layer and for a PEPS layer. This computation can be easily extended to general tensor networks, as long as we have a consistent ordering of the nodes.

B. Lineal Tensor Network Renormalization initialization protocol

If we have a tensor network \mathcal{A} , representing a matrix $n_A \times m_A$ whose lineal entrywise norm $\|\mathcal{A}\|_L$ is infinite,

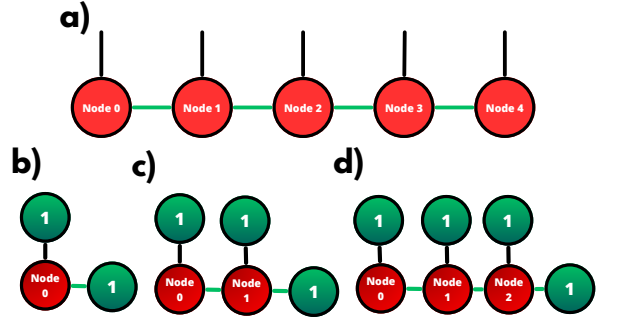


FIG. 8. a) Tensor Train layer with 5 nodes. b) Partial lineal norm at 1 node. c) Partial lineal norm at 2 nodes. d) Partial lineal norm at 3 nodes. The 1 nodes are ones vectors.

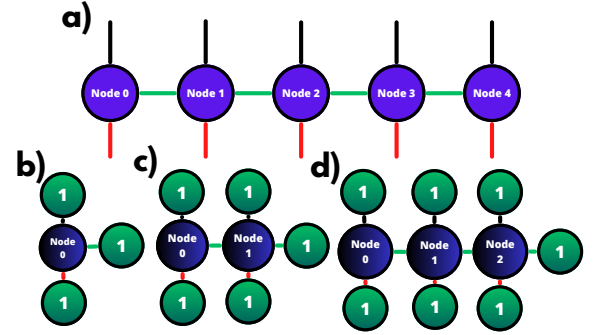


FIG. 9. a) Tensor Train Matrix layer with 5 nodes. b) Partial lineal norm at 1 node. c) Partial lineal norm at 2 nodes. d) Partial lineal norm at 3 nodes. The 1 nodes are ones vectors.

zero or outside a certain range of values, we will want to normalize the elements of our tensor network so that the norm $\|\mathcal{B}\|_L$ of the new tensor network \mathcal{B} is equal to a certain number. We assume that the tensor represented by this tensor network has a smooth distribution in the values of its elements, due to a smooth distribution also of the elements of the nodes of its tensor network [20]. We will also assume $F = n_A m_A$, but we will see that another number can be easily chosen. To normalize the norm of the \mathcal{A} tensor with N nodes, we will only have to divide the elements of each of its nodes by $\|\mathcal{A}\|_L^{1/N}$.

The same reasons as in the previous algorithm led to an analogous algorithm, but with the change from the partial square norm to the partial lineal norm. The idea is to iteratively normalize the norm little by little so that we eventually achieve full normalization. As in the general case, after each normalization, there is no need to recalculate from a single node, and you can continue from the current subnetwork.

We want a tensor network \mathcal{A} with lineal entrywise norm F , with N nodes, and we set a tolerance range (aF, bF) , with $a \leq 1$, $b \geq 1$. The *Lineal Tensor Network Renormalization* initialization protocol to follow would be:

1. We initialize the node tensors with some initializa-

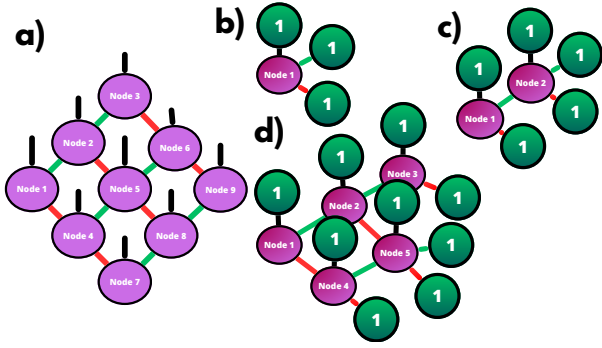


FIG. 10. a) PEPS layer with 9 nodes. b) Partial linear norm at 1 node. c) Partial linear norm at 2 nodes. d) Partial linear norm at 5 nodes. The 1 nodes are ones vectors.

tion method. We recommend random initialization with a Gaussian distribution of a constant standard deviation (not greater than 0.5) and a constant mean neither too high nor too low and positive.

2. We compute $\|\mathcal{A}\|_L$. If it is finite and non-zero, we divide each element of each node by $\left(\frac{\|\mathcal{A}\|_L}{F}\right)^{1/N}$ and return \mathcal{A} . Otherwise, we continue.
3. We compute ${}^{pL}\|\mathcal{A}\|_{1,N}$.
 - (a) If it is infinite, we divide each element of the nodes of \mathcal{A} by $(10(1+\xi))^{1/N}$, being ξ a random number between 0 and 1, and return to Step 2.
 - (b) If it is zero, we multiply each element of the nodes of \mathcal{A} by $(10(1+\xi))^{1/N}$ and return to Step 2.
 - (c) Otherwise, we save this value as ${}^{pL}\|\mathcal{A}\|_{1,N}$ and continue.
4. For $n \in [2, N-1]$, we compute ${}^{pL}\|\mathcal{A}\|_{n,N}$.
 - (a) If it is infinite or zero, we divide each element of the nodes of \mathcal{A} by $({}^{pL}\|\mathcal{A}\|_{n-1,N})^{\frac{1}{N}}$, and repeat Steps 2 and 4 (from this value of n).
 - (b) If it is finite, but larger than b or smaller than a , we divide each element of the nodes of \mathcal{A} by $\left(\frac{{}^{pL}\|\mathcal{A}\|_{n,N}}{F}\right)^{\frac{1}{N}}$, and repeat Steps 2 and 4 (from this value of n).
 - (c) Otherwise, we continue.
5. If no partial lineal norm is outside the range, infinite or zero, we divide each element of the nodes of \mathcal{A} by $\left(\frac{{}^{pL}\|\mathcal{A}\|_{N-1,N}}{F}\right)^{\frac{1}{N}}$, and repeat steps 2 and 5.

As in the general case, we repeat the cycle until we reach a stop condition, which will be to have repeated a certain maximum number of iterations. If we reach that point, the protocol will have failed, and we will have the same two options as before.

V. EXPERIMENTS

In this section, we will perform several experiments with both algorithms. We will check the scaling of the number of steps needed to normalize the tensor network, considering one step each time we have to partially normalize because we have found an infinity or zero in the partial norms. We tested for the TT layer and the TT matrix layer with N nodes, uniform physical dimensions p and bond dimensions b . We use a value of 1 for the mean and a value of 0.5 for the standard deviation. We choose $F = p^N$. For the TT layer, this is the number of elements we have, and for the TT-matrix layer, it is its square root, a number we take for convergence purposes. Our tolerance range is $(10^{-3}F, 10^3F)$.

We first test the performance of the normalization with the Frobenius norm. First, we check the number of steps versus the number N of the nodes of the tensor network, from 2 to 34, for different values of p with $b = 12$ in Fig. 11. Then, we check the number of steps against p for the same value of $N = 25$ and $b = 10$ in Fig. 12. Finally, we check the number of steps against b with the same value of $N = 25$ and $p = 15$ in Fig. 13.

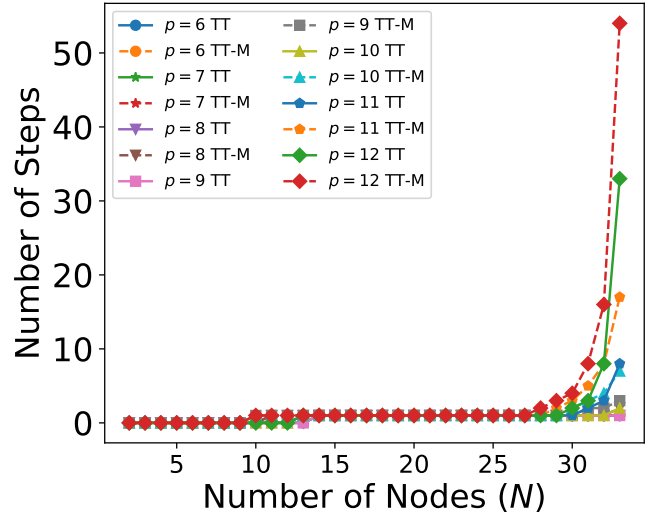


FIG. 11. Number of steps vs N for p from 6 to 12 for the TT layer and TT-Matrix layer with fixed $b = 10$ for the Frobenius method.

We can observe that for all cases, for $N < 10$ no steps are needed, while there is a region up to $N = 27$ where only one step is needed. For a larger number of nodes, the number of steps increases exponentially with N , increasing faster for larger p . Against p , we see that for $p < 15$ only one step is needed, while for larger values, there is another exponential growth. Relative to b , no substantial dependence is observed. The algorithm should need more steps for larger cases, but we have not enough memory to compute them. For all checks, the TT-Matrix requires a larger number of steps.

Now, we test the linear algorithm, with the same con-

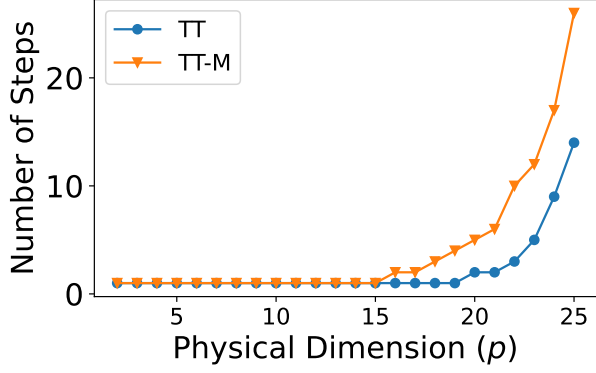


FIG. 12. Number of steps vs p with fixed $N = 25$ and $b = 10$ for the TT layer and the TT-Matrix layer for the Frobenius method.

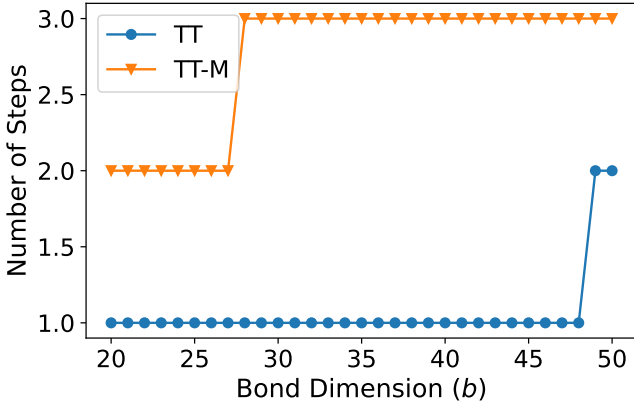


FIG. 13. Number of steps vs b with fixed $N = 25$ and $p = 15$ for the TT layer and the TT-Matrix layer for the Frobenius method.

figurations as in the Frobenius experiment. In Fig. 14, we can observe a similar tendency as in the Frobenius case. All instances of $N < 13$ need no steps and for $N < 30$ only one step is enough to normalize the matrix. In the following region, an exponential amount of steps are needed, but not as much as in the Frobenius case. Only the $p = 12$ TT-matrix instance needs more steps. In Fig. 15, we can observe the same behavior as in the Frobenius case, but with less required steps. Finally, in Fig. 16, there is no clear dependence for the number of steps and b , probably for the same reasons as before.

As we can see, both algorithms perform notably well in a wide range of possible instance sizes. Moreover, the lineal algorithm performs better, probably due to the smoother scaling of the lineal entrywise norm of the tensor, which scales linearly while Frobenius norm scales quadratically in the interior of the square root.

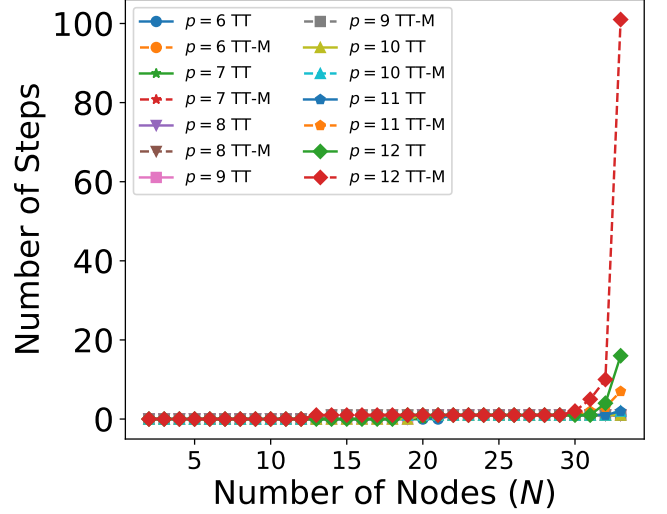


FIG. 14. Number of steps vs N for p from 6 to 12 for the TT layer and TT-Matrix layer with fixed $b = 10$ for the lineal method.

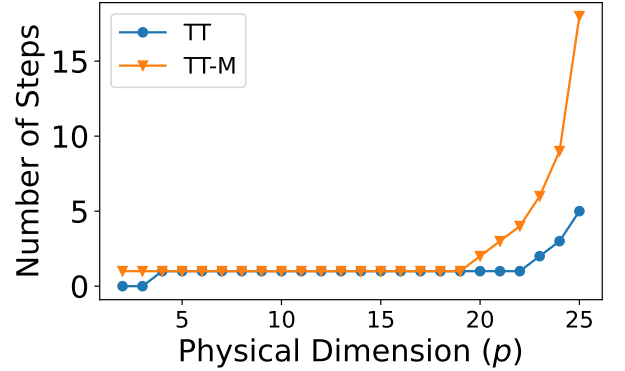


FIG. 15. Number of steps vs p with fixed $N = 25$ and $b = 10$ for the TT layer and the TT-Matrix layer for the lineal method.

VI. OTHER APPLICATIONS

So far we have seen the application of tensorized neural networks, but this method can be useful for more fields. Every algorithm or application that needs to contract a tensor network and the non-zero elements of the tensors that form it are of the same order of magnitude, we can benefit from these methods. This can be helpful in cases where we do not want the absolute scale of the final tensor elements but we want to observe a relative scale between them.

An example would be the simulation of imaginary time evolution processes where we want to see which is the state with minimum energy and not the energy it has. However, this energy could be recovered if we performed the method, saved the scale factor by which we multiply

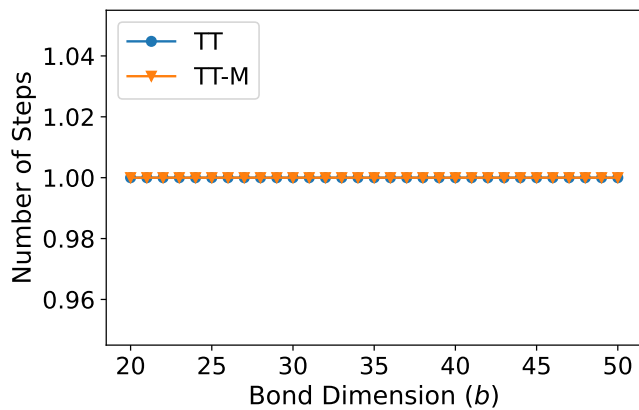


FIG. 16. Number of steps vs b with fixed $N = 25$ and $p = 15$ for the TT layer and the TT-Matrix layer for the lineal method.

the elements of the tensor network, and multiplied the values of the resulting tensor network by this factor. This can be interesting because the different factors can be multiplied keeping their order of magnitude apart, so that we do not have overflows.

VII. CONCLUSIONS

We have developed two methods to successfully initialize layers of tensorized neural networks using partial

computations of their Frobenius norms and linear entry-wise norms, depending on the type of tensor network involved. We have exposed a way to take advantage of intermediate calculations for various types of tensor network to optimize this process. We have also applied it to different layers and seen its scaling versus the number of nodes, bond dimension, and physical dimension.

The main limitation of these methods is the need for the values of the represented tensor elements to follow a smooth distribution, which can be limiting in certain cases. This limitation must be overcome in order to generalize the scope of application of this kind of algorithms, allowing new tensor network algorithms to be possible.

A possible future line of research could be to investigate how to reduce the number of steps to be performed. Another could be to study the scaling of complexity with increasing size of each of the different types of existing layers. We could also apply it to the methods mentioned in Sec. VI, for example in combinatorial optimization [27] to determine the appropriate decay factor and adapt it to quantum machine learning layers.

ACKNOWLEDGMENTS

The research leading to this paper has received funding from the Q4Real project (Quantum Computing for Real Industries), HAZITEK 2022, no. ZE-2022/00033.

-
- [1] M. H. M. Noor and A. O. Ige, A survey on deep learning and state-of-the-art applications (2024), arXiv:2403.17561.
 - [2] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, Llama: Open and efficient foundation language models (2023), arXiv:2302.13971 [cs.CL].
 - [3] V. Belis, P. Odagiu, M. Grossi, F. Reiter, G. Dissertori, and S. Vallecorsa, Guided quantum compression for high dimensional data classification, *Machine Learning: Science and Technology* **5**, 035010 (2024).
 - [4] J. Biamonte and V. Bergholm, Tensor networks in a nutshell (2017), arXiv:1708.00006.
 - [5] R. Orús, A practical introduction to tensor networks: Matrix product states and projected entangled pair states, *Annals of Physics* **349**, 117–158 (2014).
 - [6] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac, Matrix product state representations (2007), arXiv:quant-ph/0608197 [quant-ph].
 - [7] V. M. F. Verstraete and J. Cirac, Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems, *Advances in Physics* **57**, 143 (2008), <https://doi.org/10.1080/14789940801912366>.
 - [8] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, Tensorizing neural networks (2015), arXiv:1509.06569.
 - [9] Z.-F. Gao, S. Cheng, R.-Q. He, Z. Y. Xie, H.-H. Zhao, Z.-Y. Lu, and T. Xiang, Compressing deep neural networks by matrix product operators, *Phys. Rev. Res.* **2**, 023300 (2020).
 - [10] Y. Qing, K. Li, P.-F. Zhou, and S.-J. Ran, Compressing neural networks using tensor networks with exponentially fewer variational parameters, *Intelligent Computing* **4**, 10.34133/icomputing.0123 (2025).
 - [11] S. Singh, S. S. Jahromi, and R. Orus, Tensor network compressibility of convolutional models (2024), arXiv:2403.14379 [cs.CV].
 - [12] H. Li, J. Zhao, H. Huo, S. Fang, J. Chen, L. Yao, and Y. Hua, T3srs: Tensor train transformer for compressing sequential recommender systems, *Expert Systems with Applications* **238**, 122260 (2024).
 - [13] D. Lee, R. Yin, Y. Kim, A. Moitra, Y. Li, and P. Panda, Tt-snn: Tensor train decomposition for efficient spiking neural network training (2024), arXiv:2401.08001 [cs.NE].
 - [14] B. Aizpurua, S. S. Jahromi, S. Singh, and R. Orus, Quantum large language models via tensor network disentanglers (2024), arXiv:2410.17397 [quant-ph].
 - [15] A. Tomut, S. S. Jahromi, A. Sarkar, U. Kurt, S. Singh, F. Ishtiaq, C. Muñoz, P. S. Bajaj, A. Elborady, G. del

- Bimbo, M. Alizadeh, D. Montero, P. Martin-Ramiro, M. Ibrahim, O. T. Alaoui, J. Malcolm, S. Mugel, and R. Orus, Compactifai: Extreme compression of large language models using quantum-inspired tensor networks (2024), arXiv:2401.14109 [cs.CL].
- [16] J. Qi, C.-H. H. Yang, P.-Y. Chen, and J. Tejedor, Exploiting low-rank tensor-train deep neural networks based on riemannian gradient descent with illustrations of speech processing (2022), arXiv:2203.06031.
- [17] P. Blagoveschensky and A. H. Phan, Deep convolutional tensor network (2020), arXiv:2005.14506.
- [18] S. K. Vemuri, T. Büchner, J. Niebling, and J. Denzler, Functional tensor decompositions for physics-informed neural networks (2024), arXiv:2408.13101 [cs.LG].
- [19] R. Patel, C.-W. Hsing, S. Sahin, S. S. Jahromi, S. Palmer, S. Sharma, C. Michel, V. Porte, M. Abid, S. Aubert, P. Castellani, C.-G. Lee, S. Mugel, and R. Orus, Quantum-inspired tensor neural networks for partial differential equations (2022), arXiv:2208.02235.
- [20] F. Barratt, J. Dborin, and L. Wright, Improvements to gradient descent methods for quantum tensor network machine learning (2022), arXiv:2203.03366.
- [21] X. Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Y. W. Teh and M. Titterton (PMLR, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.
- [22] X. Tang, Y. Khoo, and L. Ying, Initialization and training of matrix product state probabilistic models (2025), arXiv:2505.06419 [math.NA].
- [23] I. Oseledets and E. Tyrtyshnikov, Tt-cross approximation for multidimensional arrays, *Linear Algebra and its Applications* **432**, 70 (2010).
- [24] F. Mezzadri, How to generate random matrices from the classical compact groups (2007), arXiv:math-ph/0609050 [math-ph].
- [25] E. Puljak, S. Sanchez-Ramirez, S. Masot-Llima, J. Vallès-Muns, A. Garcia-Saez, and M. Pierini, tn4ml: Tensor network training and customization for machine learning (2025), arXiv:2502.13090 [cs.LG].
- [26] J. Wang, C. Roberts, G. Vidal, and S. Leichenauer, Anomaly detection with tensor networks (2020), arXiv:2006.02516.
- [27] T. Hao, X. Huang, C. Jia, and C. Peng, A quantum-inspired tensor network algorithm for constrained combinatorial optimization problems, *Frontiers in Physics* **10**, 10.3389/fphy.2022.906590 (2022).