TensorKrowch: Smooth integration of tensor networks in machine learning

José Ramón Pareja Monturiol[1,2], David Pérez-García[1,2], and Alejandro Pozas-Kerstjens[2]

[1]Departamento de Análisis Matemático, Universidad Complutense de Madrid, 28040 Madrid, Spain
[2]Instituto de Ciencias Matemáticas (CSIC-UAM-UC3M-UCM), 28049 Madrid, Spain

Tensor networks are factorizations of high-dimensional tensors into networks of smaller tensors. They have applications in physics and mathematics, and recently have been proposed as promising machine learning architectures. To ease the integration of tensor networks in machine learning pipelines, we introduce TensorKrowch, an open source Python library built on top of PyTorch. Providing a user-friendly interface, TensorKrowch allows users to construct any tensor network, train it, and integrate it as a layer in more intricate deep learning models. In this paper, we describe the main functionality and basic usage of TensorKrowch, and provide technical details on its building blocks and the optimizations performed to achieve efficient operation.

1 Introduction

Tensor networks are factorizations of high-dimensional tensors into network-like structures composed of smaller tensors. Originating from condensed matter physics and acclaimed for their efficient representation of quantum many-body systems [1–10], these structures have allowed researchers to comprehend the intricate properties of such systems and, additionally, simulate them using classical computers [11–13]. Notably, tensor networks are the most successful method for simulating the results of quantum advantage experiments [14–16]. Furthermore, tensor networks were rediscovered within the numerical linear algebra community [17–19], where the techniques have been adapted to other high-dimensional problems such as numerical integration [20], signal processing [21], or epidemic modelling [22].

With the advent of machine learning and the the quest for expressive yet easy-to-train models, tensor networks have been suggested as promising candidates, due to their ability to parameterize regions of the complex space of size exponential in the number of input features. Since the pioneering works [23, 24] that used simple, 1-dimensional networks known as Matrix Product States (MPS) in the physics literature [4, 25] and as Tensor Trains in the numerical linear algebra literature [18], these have been applied in both supervised and unsupervised learning settings [26–28]. Recent studies have also delved into alternative architectures, including Tree Tensor Networks (TTN) [29, 30] and Projected Entangled Pair States (PEPS) [31, 32].

While there exist indications that tensor network architectures may outperform neural networks in certain scenarios [33], neural networks still hold the upper hand both in versatility and efficiency. However, there exists a growing number of cases where tensor networks seem to provide advantages. First, tensor networks offer a means to compress the matrices used in existing neural networks. This process, known as tensorization, reduces the amount of memory required to store the model, and improves the efficiency of the model in both training and inference [34]. The potential of tensorization has already been explored in several studies [34–36], offering a way to execute complex models in edge computing devices [37]. Second, the large expertise of the community of quantum many-body physics in tensor networks, and their inspiration in real physical systems, allows to better understand questions related to explainability [29, 38, 39]. Third, this expertise can also bring novel features, such as guarantees on privacy that do not compromise on model performance [40]. Finally, another promising research line involves the integration of tensor

network layers with neural network layers. For instance, Ref. [26] proposes using the output of a convolutional neural network, treated as a feature extractor, as the input to four 1-D tensor networks. Remarkably, this straightforward model achieves near-state-of-the-art performance on the FashionMNIST dataset [41].

Therefore, there are several reasons to believe that the integration of tensor networks into deep learning pipelines can enhance the capabilities of current models. Several libraries exist [42–46] that allow to use some concrete tensor network architectures as machine learning models, or that use gradient-based methods to optimize tensor-network ansatzes for quantum many-body calculations. However, from the point of view of machine learning, there is still a need for extensive research to determine, for instance, the situations in which the properties of tensor networks can be maximally leveraged, which are the most effective training methods, optimal architectures (and architectures beyond those used by physicists), and so on. Consequently, there is a demand for user-friendly tools that enable rapid experimentation in this field.

To address this demand, here we introduce TensorKrowch[1] [47], a Python library built on top of PyTorch [48] that aims to bring the full power of tensor networks to machine learning practitioners. TensorKrowch allows to construct any tensor network model using the familiar language and capabilities of PyTorch. The key strength of TensorKrowch lies in defining a solid set of basic components, namely `Nodes` and `Edges`, upon which the entire tensor network can be built. By connecting these `Nodes`, a complete `TensorNetwork` model can be created, that integrates smoothly with other PyTorch modules. Consequently, TensorKrowch leverages the full power of PyTorch, including GPU acceleration, automatic differentiation, compilation to XLA, and easy composition of multi-layer networks. Additionally, TensorKrowch incorporates built-in implementations of widely used tensor networks such as MPS, MPO, TTN and PEPS, methods to tensorize layers of pre-trained neural network models, and tools developed within the context of quantum information theory that enable interpreting the inner workings of the models [29, 38, 39], such as the computation of reduced density matrices and entanglement entropies.

This work is structured as follows: Section 2 introduces tensor networks and their graphical notation. Section 3 presents the library, discussing its underlying philosophy, basic requirements, and main components. Section 4 provides a detailed explanation of the components comprising TensorKrowch, such as `Nodes`, `Edges`, and `TensorNetworks`, and how they are interconnected. Section 5 discusses the operations one can perform between nodes. Section 6 combines all the previously described pieces to guide readers in building their own custom models and training them. Section 7 covers advanced concepts like memory management. Lastly, Section 8 contains additional software information such as future development and contribution guidelines, and Section 9 presents some concluding remarks.

This paper aims to be self-contained, providing a glimpse of the basics of tensor networks for readers from both the machine learning and quantum physics backgrounds, and introducing the fundamental components of TensorKrowch in order to enable readers to create and train their own models. However, it is not intended as a comprehensive tutorial on all the capabilities of the library or a complete description of all its functionality. Such information is available in the TensorKrowch documentation at: https://joserapa98.github.io/tensorkrowch.

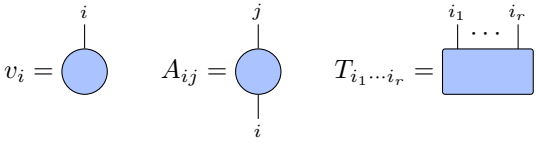
2 Tensor Networks

In this section, we will introduce the concept of a tensor network and the basic operations that are relevant in the context of machine learning. For a more in-depth analysis, we refer the reader to [4, 25, 49, 50].

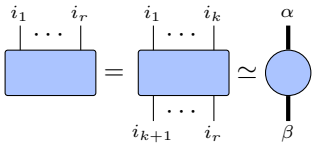
Tensors are an extension of vectors and matrices to higher dimensions. They can be visualized as collections of indexed numbers arranged in multi-dimensional arrays. In general, a rank-r tensor with dimensions $d_1 \times \cdots \times d_r$ belongs to the tensor product vector space $\bigotimes_{i=1}^{r} \mathbb{C}^{d_i} \simeq \mathbb{C}^{\times_{i=1}^{r} d_i}$. Its elements are represented by $T_{i_1 \cdots i_r}$, where each $i_j \in [d_j]$.

The literature on tensor networks also presents a useful and practical graphical notation, where tensors are represented as the nodes of a graph with edges corresponding to their indices. In this notation, vectors, matrices and arbitrary tensors take the following form:

[1] This work describes the library at its latest version upon publication, namely 1.1.4.



In certain areas where tensors are relevant, for instance in geometry or general relativity, subscripts and superscripts are employed to denote indices in covariant or contravariant spaces, and changing the position of the indices requires using the associated metric. However, in the tensor network literature it is customary to obviate all these subtleties, and make no distinctions between sub- and superscript indices. Moreover, it is standard to group and ungroup sets of indices whenever it is convenient. In this way, the same collection of numbers can be arranged in a matrix $A_{ij} \in \mathbb{C}^{d_1} \otimes \mathbb{C}^{d_2}$, or in a vector $A_k \in \mathbb{C}^{d_1 \times d_2}$ by defining $k = (i-1)d_2 + j$.



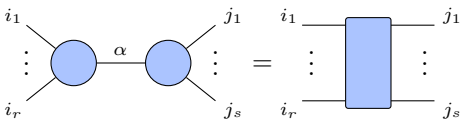
(1)

This flexibility in representation makes it clear that tensors are essentially linear mappings between tensor product spaces, similar to how matrices represent linear mappings between vector spaces. Furthermore, it provides a convenient framework to decompose tensors via matrix factorization algorithms, as we will see shortly below.

Indices of tensors can be *contracted*. The contraction of indices (of the same or different tensors) consists of a generalization of the scalar product between vectors, this is, the sum of products of the elements in the corresponding axes. For two arbitrary tensors, R and S, of ranks r and s, respectively, the contraction over one of its indices is given by

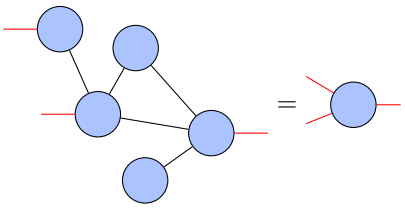
$$\sum_{\alpha=1}^{d} R_{i_1 \ldots i_{m-1} \alpha i_{m+1} \ldots i_r} S_{j_1 \ldots j_{n-1} \alpha j_{n+1} \ldots j_s} = T^{j_1 \ldots j_{n-1} j_{n+1} \ldots j_s}_{i_1 \ldots i_{m-1} i_{m+1} \ldots i_r}, \tag{2}$$

where we have assumed that the dimensions of the indices i_m and j_n are both d. This tensor T is of rank $(r+s-2)$. In the graphical notation, the contraction is represented by connecting the corresponding edges of the tensors:



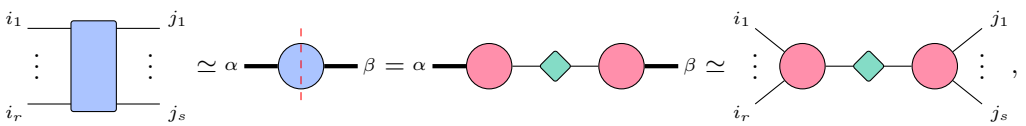
(3)

Contraction enables connecting nodes to form graphs, with the only requirement being that the nodes must have the same dimensions along the connected axes. By contracting all the connected edges in the graph, one can then contract a whole tensor network to obtain a single tensor that preserves the remaining dangling edges:



(4)

Conversely, one may obtain a tensor network by splitting tensors via the various existing decomposition methods [51]. A commonplace method is exploiting grouping and ungrouping of indices to reshape tensors as matrices and applying the singular value decomposition,

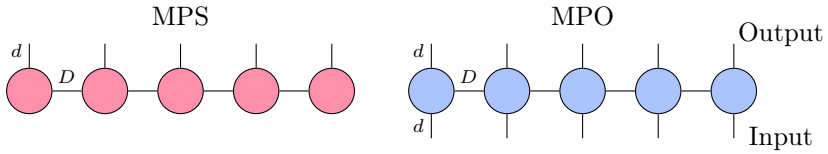


(5)

since this provides, in many cases of relevance, a form of the tensor with desirable properties [4]. In particular, truncations of the singular value decomposition give the best low-rank approximations of the original tensor under the Frobenius and ℓ_2 norms.

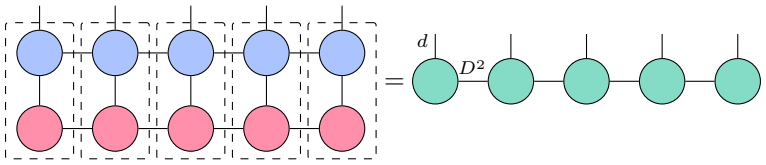
In machine learning, there are two main applications of tensor networks. The first is the decomposition, or approximation, of large tensors in networks with a smaller number of entries. This procedure is known as tensorization [34–36]. Tensorization allows not only to reduce the number of elements to be stored (and thus reducing the storage memory), but also to reduce the computation time. Consider a linear layer within a neural network, where both the input and output dimensions are d^n. The multiplication of the corresponding weight matrix with the input data vector requires $O(d^{2n})$ operations.

On the other hand, consider the same operation between factorizations (or approximations) of the vector and the matrix into tensor networks. For this example, let us use the well-known Matrix Product State (MPS) [4, 25] form for the vector and a Matrix Product Operator (MPO) [52] form for the matrix, both with n nodes, each equipped with dangling edges of dimension d (this is known as the *physical dimension* in the physics literature) and connected to their neighbors via edges of dimension D (this is known as the *bond dimension*). Thus, the MPS requires only $O(ndD^2)$ elements to represent the vector, and the MPO requires $O(nd^2D^2)$ elements to represent the matrix.



(6)

The vector-matrix multiplication is carried out by contracting the MPS with the MPO, connecting all the dangling edges of the MPS to the input edges of the MPO:



(7)

The contraction of these connected edges requires only $O(nd^2D^4)$ operations. Note that the contractions in the boxes in Eq. (7) all involve different tensors and thus can be performed in parallel, thereby giving even greater savings in practice. Thus, representing linear layers in tensorized form presents advantages in terms of, both, memory usage and in the time taken to perform the operations. Similar advantages are also present in the backward pass of automatic differentiation routines [34].

The second main application of tensor networks in machine learning is as architectures themselves. In tensor network architectures, every cell of every tensor is a trainable parameter, and so they implement linear models operating in high-dimensional tensor-product spaces. Taking the example of classification, the contraction of a tensor form of the input with the tensor network gives the corresponding prediction, that is used to compute a loss function whose gradients are used to adjust the model parameters. This approach was pioneered by [23, 24] with MPS, and since has been applied to many different problems [27, 33, 53] and network architectures [29–31, 54].

3 TensorKrowch

3.1 Motivation

The works [23, 24] gave rise to the area of tensor network machine learning, where nowadays many different network architectures are being used for different purposes. However, the common choices for architectures are inherited from the physics community, which has broad experience in tensor networks but is restricted to 1-D and select 2-D architectures. It is for this reason that, currently, there are libraries that are well-integrated into deep learning frameworks but are optimized for specific tensor networks [42] or put their focus on tensorization [43, 44]. On the other hand,

there exist libraries like TensorNetwork [45], that offer more generality in terms of architectures at the cost of more complications at the time of integrating them into machine learning pipelines. Furthermore, the available libraries tend to have a broader focus on physics applications, and thus features that are of interest in machine learning, such as custom parameter initializations, hyperparameter optimization, or even the construction of complicated models, are not considered.

TensorKrowch is developed with the aim of providing a comprehensive framework where one can rapidly prototype tensor network layers as standalone models and integrate them in deep machine learning models. Its main characteristics are the following:

Generality: TensorKrowch allows users to construct any tensor network by creating and connecting nodes. Users can selectively choose which nodes to train and to define arbitrary operations between their components. In addition to this, TensorKrowch has pre-built classes for the most common types of networks.

Ease of use: At the core of TensorKrowch is a Pythonic approach, presenting a simple interface with building blocks and operations that are combined in order to create complex models and training pipelines. These primary objects and operations are described in sections 4 and 5, respectively.

Optimization: While the interface remains simple, numerous optimizations are implemented in the background in order to perform operations efficiently and to reduce redundant computations during training. Section 7 details the optimizations that are performed.

Integration: TensorKrowch is written on top of a well-established deep learning framework, namely PyTorch [48]. This integration enables the creation of `TensorNetwork` models that function like any other PyTorch layer, seamlessly combining with existing components. Consequently, TensorKrowch fully leverages the capabilities of PyTorch, including GPU acceleration, automatic differentiation, and easy composition of multi-layer networks.

Moreover, TensorKrowch can be also used for tensorization purposes, by substituting or approximating dense matrices in deep PyTorch models and applying the built-in matrix factorization techniques.

3.2 Installation and requirements

TensorKrowch is a Python library available on Linux, Mac and Windows operating systems. It can be installed via `pip` with the following command line:

```
pip install tensorkrowch
```

The basic requirement of TensorKrowch is PyTorch [48], which is used as machine learning backend. Additionally, TensorKrowch requires opt_einsum [55], used in the implementation of the `einsum` operation and that allows for the automatic search of good network contraction paths via greedy algorithms.

The source code for TensorKrowch is hosted on GitHub at

https://github.com/joserapa98/tensorkrowch

and is distributed under the MIT License. More details about the software, packaging information, and guidelines for contributing to TensorKrowch are included in Sec. 8.

3.3 Basic Usage

TensorKrowch provides a set of basic components, namely `Node`, `Edge`, `TensorNetwork`, and variants of these, that can be combined to build trainable models. The usual workflow consists in the following steps: 1) Define the structure of the graph by creating nodes and connecting them; and initialize the tensors within those. 2) Specify which nodes will be used to store the input tensors coming from the training dataset. 3) Define the contraction algorithm to reduce the whole network to a single output tensor, which one can input to any other layer that might follow the tensor network.

To carry out these steps, one needs to know how to create and combine the different building blocks of the model, and how to operate them to contract the network. These topics will be covered in detail in Section 4 and Section 5, respectively.

Once the custom tensor network is defined, the process of training it is analogous to what one would do in vanilla PyTorch. To illustrate this with an example, let us introduce one of the built-in classes provided by TensorKrowch, `MPSLayer`. This class is a variation of the traditional MPS with an additional node that has a dangling edge representing an output dimension. As a result, the `MPSLayer` is contracted into a vector of the specified dimension.

```
import torch
import tensorkrowch as tk

# Instantiate model
mps = tk.models.MPSLayer(n_features=1001,
                         in_dim=2,
                         out_dim=10,
                         bond_dim=10)
```

Tensor network models expect as input a rank-3 tensor with dimensions $b \times n \times d$, being b the size of the batch, n the number of feature vectors, and d their dimension. This tensor, which is a batch of sequences of vectors, represents a tensor network itself, given by the tensor product of all the vectors corresponding to each batch element. These vectors will be placed in specific nodes so that the network can be contracted with these new data.

However, data tensors tend to come in the form of $b \times n$ matrices, requiring a previous transformation to get the proper rank-3 tensor. That is, for each batch element there is a vector of features, that has to be turned into a sequence of feature vectors. To accomplish this, each feature has to be embedded into a vector space. TensorKrowch provides five common embedding functions, namely `unit`, `add_ones`, `poly`, `discretize` and `basis`, the first two being introduced in [23] and [24], respectively.

```
# Shape: batch_size x n_features
data = torch.randn(100, 1000)
# Shape: batch_size x n_features x in_dim=2
data = tk.embeddings.unit(data)

# Shape: batch_size x out_dim=10
labels = torch.randn(100, 10)
```

To reduce repetition of ancillary computations due to dealing with the tensor network structure, TensorKrowch introduces some extra steps that need to be carried out before starting training. These involve setting memory modes and tracing the model; both advanced features will be covered in Section 7.

```
# Set memory modes
mps.auto_stack  = True
mps.auto_unbind = False

# Trace
example = torch.zeros(1, 1000, 2)
# Shape: batch_size=1 x n_features x in_dim=2
mps.trace(example)
```

To be able to train, one needs to set a loss function to minimize and an optimizer to update the parameters of the model according to their gradients. It is important that the optimizer is set after the model has been traced, since the parameters of the model might change during this process.

```
loss_fun  = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mps.parameters(),
                             lr=1e-4,
                             weight_decay=1e-2)
```

Finally, the above ingredients can be put together in the training loop.

```
for epoch in range(n_epochs):
    # Contract tensor network with input data
    scores = mps(data)

    # Compute the loss
    loss = loss_fun(scores, labels)

    # Backpropagate and update parameters
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

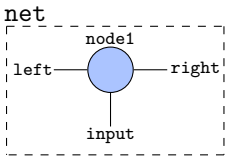
4 The Building Blocks

The main structure of TensorKrowch is similar to that of TensorNetwork [45]. Namely, the main object in TensorKrowch is a `TensorNetwork`, that is populated with `Nodes`. These have `Edges` that are connected according to the desired structure. Creating a `TensorNetwork` is done as follows:

```
import torch
import tensorkrowch as tk
net = tk.TensorNetwork()
```

`Nodes` are the basic elements that make up a `TensorNetwork`. They serve as containers for PyTorch's `torch.Tensor` objects and hold essential information for building, operating, and training the network. Key aspects associated with nodes include shape, tensor, axes, edges, network membership, and successors. Tensors themselves are not actually contained within nodes; instead, they are stored in the shared memory system that is accessible to all nodes in the `TensorNetwork` (see Section 7 for details). To create a `Node` one can specify its name, shape, names for its axes, network, and an initialization method to create a new tensor for it. Together with the code we present a graphical notation to depict not only the tensor network but all the elements that form the `TensorNetwork` object.

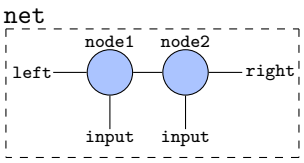
```
node1 = tk.Node(name="node1",
                shape=(2, 5, 2),
                axes_names=
                    ("left", "input", "right"),
                network=net,
                init_method="randn")
```



Specifying a shape automatically creates a set of edges for the `Node`. An `Edge` is nothing more than an object that wraps references to the nodes it connects. Thus it stores information like the nodes it connects, the corresponding nodes' axes it is attached to, its size, etc.

To connect nodes, one can access the desired edges using their names, and connect them with the caret operator:

```
# Equivalent to initialize Node with
# "randn" init_method
node2 = tk.randn(name="node2",
                 shape=(2, 5, 2),
                 axes_names=
                     ("left", "input", "right"),
                 network=net)

node1["right"] ^ node2["left"]
```



It is important to note that the operation `^` does not perform contractions, since it may be done on edges of empty nodes. As it will be shown below, contraction between tensors is performed by using the `@` operator, that contracts tensors along all the edges that connect both nodes.

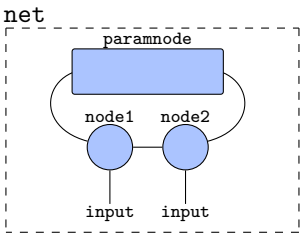
Edges can be designated as *batch* edges if one includes the string `"batch"` in the name of the corresponding axis. This allows for performing batch contractions if the nodes involved in the operation both have batch edges sharing the same name. This functionality is analogous to that of PyTorch functions like, for instance, `torch.bmm`, used to perform batch matrix multiplication.

4.1 Types of nodes

TensorKrowch features several types of nodes, that have different functionalities and goals within the library.

Parameter nodes: In analogy with PyTorch having `torch.Tensor` to denote a tensor of non-trainable entries and `torch.nn.Parameter` for tensors of trainable parameters, TensorKrowch has `Node` to denote containers of tensors with non-trainable entries and `ParamNode` to denote containers of tensors with trainable entries. One can instantiate a `ParamNode` with:

```
paramnode = tk.ParamNode(name="paramnode",
                         shape=(2, 2),
                         axes_names=
                             ("left", "right"),
                         network=net,
                         init_method="randn")

paramnode["left"] ^ node1["left"]
paramnode["right"] ^ node2["right"]
```

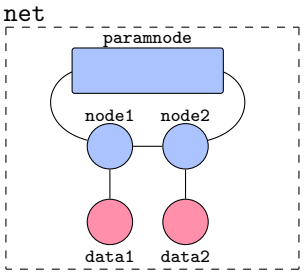


In the graphical notation we will use squares for denoting trainable nodes.

Leaf nodes: These are the nodes that form the `TensorNetwork`, along with the data nodes (see below). Usually, leaf nodes will be the trainable nodes. All leaf nodes contain PyTorch leaf tensors, this is, tensors that are not generated by operations tracked by PyTorch's automatic differentiation engine. By default, all nodes are leaf. In the graphical notation, these are the blue nodes.

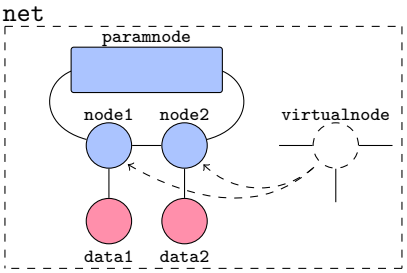
Data nodes: These are nodes that are used to store the tensors coming from input data. It is possible to instantiate data nodes directly by specifying `data=True` in the initialization of `Node`. However, data nodes will be usually created when specifying where they should be put in the network using `set_data_nodes` (see Section 6). Graphically, data nodes will be depicted as red circles.

```
data1 = tk.Node(name="data1",
                shape=(5,),
                axes_names=("feature",),
                data=True,
                network=net)

data2 = tk.Node(name="data2",
                shape=(5,),
                axes_names=("feature",),
                data=True,
                network=net)

# We can add input data tensors to the nodes
data1.tensor = torch.randn(5)
data2.tensor = torch.randn(5)

node1["input"] ^ data1["feature"]
node2["input"] ^ data2["feature"]
```



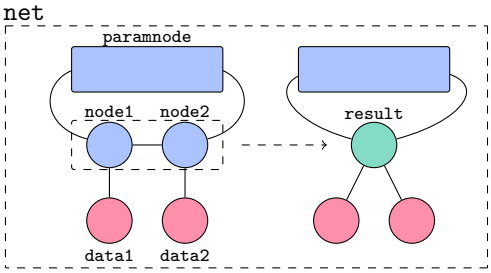
Virtual nodes: These nodes are internal nodes that TensorKrowch uses as shortcuts when one wishes to have the same tensor in different nodes, or to process auxiliary information without adding new leaf nodes to the network. This is useful when defining uniform tensor networks [56], as all their nodes simply use a single tensor stored in a virtual node instead of one tensor per node. Virtual nodes are intended to be mainly internal, but they can be created manually using the argument `virtual=True`. This is needed when defining custom uniform architectures beyond those natively implemented, such as `UMPS`, `UMPO`, `UPEPS` or `UTree`. Virtual nodes are represented by empty nodes with dashed borders.

```
virtual = tk.Node(name="virtualnode",
                  shape=(2, 5, 2),
                  axes_names=
                      ("left", "input", "right"),
                  virtual=True,
                  network=net,
                  init_method="randn")

# Put virtual's tensor into node1 and node2
node1.set_tensor_from(virtual)
node2.set_tensor_from(virtual)
```



Resultant nodes: The result of a contraction of tensors (stored in their respective nodes) is another tensor that must be stored in a new node. These nodes are resultant nodes, which coexist in the `TensorNetwork` object together with the original nodes, inheriting their edges. This allows for subsequent contraction with other neighbouring nodes. Resultant nodes are automatically created upon contracting two nodes on a same network, or when tracing the model on its first run (see Section 5.1). Resultant nodes are displayed as green circles.

```
# Nodes are contracted along their
# connected edges with the @ operator
result = node1 @ node2
```

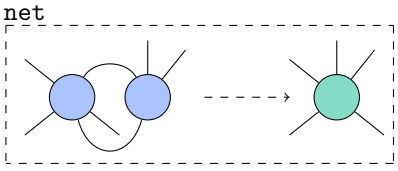


5 Operations between Nodes

Operations between TensorKrowch `Nodes` are instances of a class, `Operation`, that is optimized to avoid unnecessary repeating of basic computations (see Section 7 for details). Operations create resultant nodes that inherit information from the parent nodes. TensorKrowch implements many operations that are extensions of those in vanilla PyTorch, like `permute`, `tprod` (`outer` in PyTorch), `mul`, `div`, `add` and `sub`. The complete list of implemented operations is available in the corresponding page of the documentation. In addition to these, TensorKrowch includes implementations of operations that are inherent to tensor networks. These are:

Contract: Enables contraction of all the edges connecting two nodes, although it is also possible to use it to contract only a selected range of edges. Furthermore, batch contraction is automatically performed if some edge is of batch type (recall, with the string `"batch"` in its name). This is the most basic operation one can use, allowing for the contraction of the whole network just by implementing a contraction path.

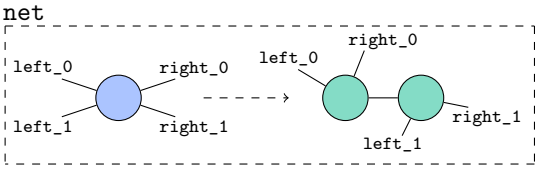
```
node1 = tk.randn(shape=(2, 4, 3, 6, 2))
node2 = tk.randn(shape=(3, 2, 5, 4))

# Edges can also be accessed by index
node1[2] ^ node2[0]
node1[4] ^ node2[1]
# node1 and node2 will belong to the same
# network upon connection

result = node1 @ node2
# result will have all the non-contracted
# edges from node1 and node2
```



Split: As an inverse of the `contract` operation, `split` factorizes the tensor in a node into two, selecting which edges should go to each resultant node. Factorization algorithms include singular value and QR decompositions, with additional functionalities such as selecting the desired amount of singular values to keep, modifying the rank of the resultant nodes accordingly. This can be useful to reduce the bond dimension in cases where it might explode during the contraction algorithm.

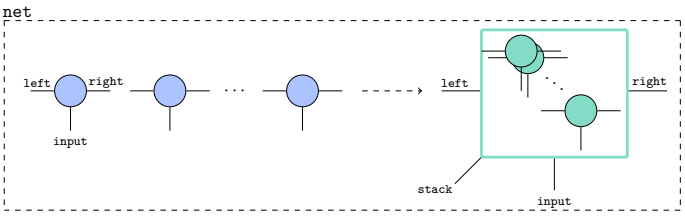
Furthermore, by iterative application of `split`, arbitrary tensors can be decomposed into tensor network formats, which enables defining custom tensorization routines [34–36].

```
node = tk.randn(
    shape=(2, 7, 3, 4),
    axes_names=
        ("left_0", "left_1",
         "right_0", "right_1")
)

node_left, node_right = tk.split(
    node,
    ["left_0", "right_0"],
    ["left_1", "right_1"],
    mode="svd",
    rank=5
)
```



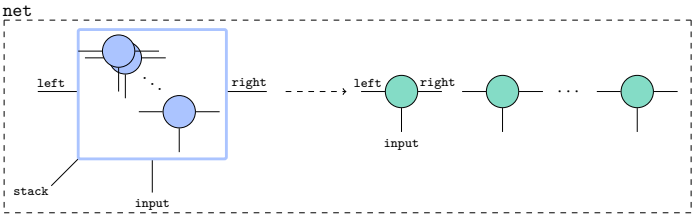
Stack: In many situations, stacking a set of tensors into a larger tensor with one extra dimension speeds up operations by allowing for parallelizing contractions (see Section 7 for more details). Creating such stacks is achieved with `stack`. The node resulting from this operation is a special type of node, namely a `StackNode` or `ParamStackNode`, that is only intended for internal use.

```
net   = tk.TensorNetwork()
nodes = [
    tk.randn(
        shape=(2, 4, 2),
        axes_names=("left",
                    "input",
                    "right"),
        network=net
    )
    for _ in range(n_nodes)
]
stacknode = tk.stack(nodes)
```



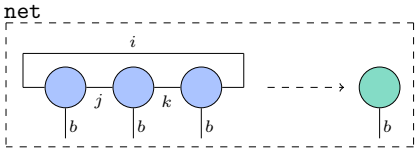
Unbind: As the counterpart of `stack`, this enables one to unbind a stack of resultant nodes, returning them in a list. Only `StackNodes` and `ParamStackNodes` can be unbound.

```
stacknode = tk.stack(nodes)
result = tk.unbind(stacknode)
```



Einsum: Allows for the implementation of complex contractions along sets of edges. Instead of contracting along a connected edge, one can define a contraction path following the Einstein summation convention to contract along several edges at once. This operation uses opt_einsum [55] at its core, making specific checks and simplifications beforehand to adhere to the rules and structures defined in TensorKrowch. For example, indices used twice can only correspond to nodes already connected by an edge in the specified axes. There is a variant of this operation, `stacked_einsum`, which allows to use lists of nodes as inputs, to perform `stack` followed by `einsum` in the same operation.

```
node1 = tk.randn(shape=(10, 15, 100),
                 axes_names=
                     ("left", "right", "batch"))
node2 = tk.randn(shape=(15, 7, 100),
                 axes_names=
                     ("left", "right", "batch"))
node3 = tk.randn(shape=(7, 10, 100),
                 axes_names=
                     ("left", "right", "batch"))

node1["right"] ^ node2["left"]
node2["right"] ^ node3["left"]
node3["right"] ^ node1["left"]

result = tk.einsum("ijb,jkb,kib->b",
                   node1, node2, node3)
```



For a comprehensive explanation of all these operations and the arguments they admit, the reader is referred to the TensorKrowch documentation.

5.1 Trace and Reset

Every operation in TensorKrowch returns a new resultant `Node` that stores the output tensor and inherits the non-contracted edges of its parents. Therefore, in order to reduce the creation of redundant nodes and the amount of memory used for this purpose, it is useful to generate void containers for the resultant tensors. This is achieved with the `trace` operation, which can be called using an example input with batch dimension 1 in order to create all the necessary resultant nodes in the fastest way possible. Further details and explicit comparisons can be found in Section 7.

```
# Shape: batch_size x n_features x feature_dim
data = torch.randn(100, 20, 5)
example = torch.zeros(1, 20, 5)

net.trace(example)
```

The inverse of the `trace` function is `reset`. This function deletes all the resultant nodes created during training, resetting the network to its initial state. This is useful when one wants to make changes to the structure of the network, to switch on/off the memory modes, or to save a trained model (otherwise, calling `torch.save(net.state_dict())` will save a *traced* model, whose parameters might not be exactly the same as the ones in the original model, due to internal optimizations).

```
# After training
net.reset()

# Save model
torch.save(net.state_dict(), "net.pt")

# Load model
new_net = TensorNetwork()
new_net.load_state_dict(torch.load("net.pt"))
```

6 Building and training tensor network models

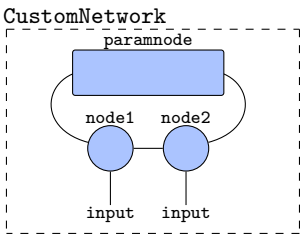
Sections 4 and 5 showed how one can create nodes belonging to a `TensorNetwork`, and operate them to contract the network. However, although this functionality might be useful for experimentation, the main usage of TensorKrowch will be to define custom models as subclasses of `TensorNetwork`. This allows, for instance, to instantiate tensor networks that work as any other PyTorch layer.

The workflow to define custom tensor networks is similar to how one defines custom layers in PyTorch. There, one needs to subclass `torch.nn.Module`, to define the parameters and architecture of the layer in the `__init__` method, and to specify how input data is processed by the layer in the `forward` method.

Similarly, for defining a custom tensor network in TensorKrowch one needs to subclass `TensorNetwork`, overriding the following methods:

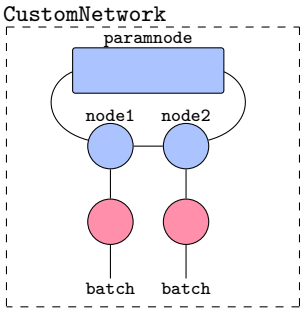
`__init__`: Defines the graph of the tensor network and initializes the tensors of the nodes.

```
class CustomNetwork(tk.TensorNetwork):

    def __init__(self):
        super().__init__(name="CustomNetwork")

        self.node1 = tk.randn(
            shape=(2, 5, 2),
            axes_names=("left", "input", "right"),
            name="node1",
            network=self
        )
        self.node2 = tk.randn(
            shape=(2, 5, 2),
            axes_names=("left", "input", "right"),
            name="node2",
            network=self
        )
        self.paramnode = tk.randn(
            shape=(2, 2),
            axes_names=("left", "right"),
            name="paramnode",
            network=self,
            param_node=True
        )

        self.node1["right"] ^ self.node2["left"]
        self.paramnode["left"] ^ self.node1["left"]
        self.paramnode["right"] ^ self.node2["right"]
```



`set_data_nodes` (optional): Creates the data nodes where the data tensors will be placed. Usually, it will just select the edges to which the data nodes should be connected, and call the parent method.

```
class CustomNetwork(tk.TensorNetwork):

    # ...

    def set_data_nodes(self):
        # Collect edges to which data
        # nodes will be connected
        input_edges = [self.node1["input"],
                       self.node2["input"]]

        # Define number of batch indices
        # for the input
        num_batch_edges = 1

        # Call parent method
        super().set_data_nodes(input_edges,
                               num_batch_edges)
```

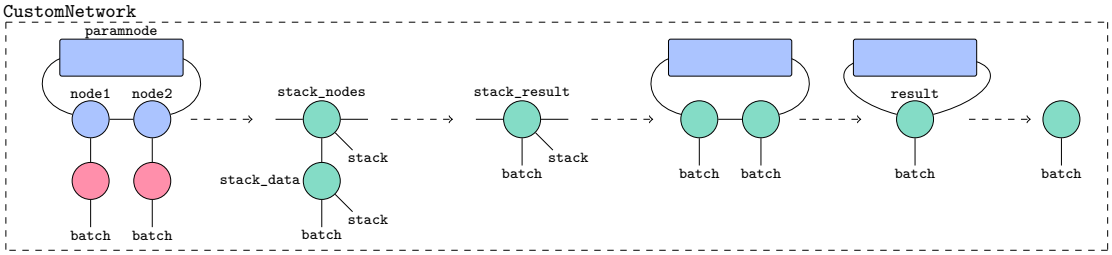


`add_data` (optional): Places input data into the previously specified data nodes. Commonly, all data nodes will have the same shape, namely $b \times d$, being b the batch size and d the feature dimension. Assuming there are n of these nodes (typically one per feature), the input to `add_data` must be a tensor of dimension $b \times n \times d$. On default operation, the input tensor will be then unbound at its second axis, delivering each slice of shape $b \times d$ to each of the data nodes. However, this method can be overridden to customize how data is set into the data nodes.

`contract`: Very much like the `forward` method in PyTorch, this is the main method that describes how the components of the network are combined. In contrast to vanilla PyTorch, however, in a TensorKrowch `TensorNetwork` the `forward` method shall not be overriden, since its goal is to just call `set_data_nodes`, if needed, `add_data`, and `contract`, and then it will return the tensor corresponding to the last resultant node. Instead, one should customize the `contract` method. TensorKrowch does not implement algorithms for searching optimal contraction paths [57]. Thus,

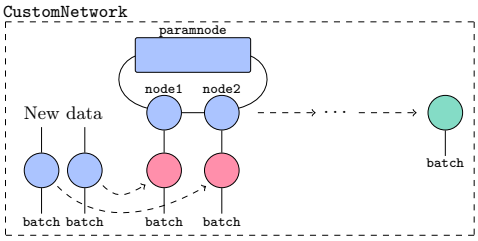
one must specify custom contraction algorithms for each user-defined tensor network, via `einsum` (recall Section 5) or by any other means. As will be detailed in Section 7, the order in which TensorKrowch `Operations` appear in the algorithm is significant, being mandatory that the last `Operation` is the one returning the final node.

```
class CustomNetwork(tk.TensorNetwork):

    # ...

    def contract(self):
        stack_nodes = tk.stack([self.node1, self.node2])
        stack_data  = tk.stack(list(self.data_nodes.values()))

        # Stacks need to be reconnected before contraction
        stack_nodes["input"] ^ stack_data["feature"]
        stack_result = stack_nodes @ stack_data

        stack_result = tk.unbind(stack_result)

        result = stack_result[0] @ stack_result[1]

        # Last operation must return the output node
        result @= self.paramnode
        return result
```



With the subclass correctly defined, one can now instantiate the custom network and feed it with new input data tensors:

```
net = CustomNetwork()

# Pass data to the model
# Shape:
#    batch_size x n_features x feature_dim
data = torch.randn(100, 2, 5)
result = net(data)
```



As mentioned in the beginning of the section, creating a custom network as a subclass of `TensorNetwork` makes the integration of tensor network layers within PyTorch models straightforward:

```
import torch.nn as nn

# Combine built-in custom TN layer with PyTorch
   layers
model = nn.Sequential(
    tk.models.MPSLayer(n_features=100,
                       in_dim=2,
                       out_dim=10,
                       bond_dim=5),
    nn.ReLU(),
    nn.Linear(10, 10)
)
```

The last codeblock contains a built-in class, `MPSLayer`, that readily implements a MPS architecture. Similar implementations are available for MPOs, PEPS and TTNs, both uniform and non-uniform. This direct integration also enables to efficiently define tensorized neural network

models through the interleaving of MPOs, or simply `MPSLayers`, with the common nonlinearities of neural networks.

In general, tensor networks have gauge symmetries, i.e., several collections of parameters describe the exact same final tensor. In certain tensor network architectures there exist ways to define a preferred set of parameters. This is known as choosing a *canonical form* [4, 58], and is desirable in some physics applications. In machine learning, canonical forms have been associated to benefits in terms of privacy preservation [40]. TensorKrowch allows, in the built-in MPS and TTN implementations, to compute canonical forms using the function `canonicalize` (or `canonicalize_univocal` for the univocal canonical form described in [40]).

6.1 Optimization

The provided code enables users to create custom `TensorNetwork` models. To train them, different approaches can be followed. In PyTorch, it is customary to pass the model parameters to native optimizers that implement different gradient descent algorithms. Gradients are automatically computed via the automatic differentiation engine, which tracks all operations in which model parameters are involved and computes gradients by applying the chain rule. Since `TensorNetwork` models are defined as subclasses of `torch.nn.Module`, and the `ParamNodes` within these are of type `torch.nn.Parameter`, this type of gradient-based optimizations are effortlessly implemented to train the `ParamNodes`. This approach enables the implementation of models similar to the one presented in [24], where all MPS nodes are optimized at the same time.

This, however, is not the only way to train tensor networks in TensorKrowch. With the capability to parameterize and de-parameterize `Nodes` and `ParamNodes`, respectively, and assign different tensors to specific `Nodes`, various optimization schemes can be explored. For instance, DMRG-like approaches [23] can be easily implemented by freezing all MPS nodes, except for a trainable block that can traverse the matrix chain.

The TensorKrowch documentation includes examples illustrating how to train MPS models in different fashions, including the one in [24], as well as DMRG approaches as explained in [23]. Besides those, implementations of hybrid tensorial neural network models [26] and tensorized neural networks [34] can be found.

7 Time and memory optimizations

Operating tensor networks requires a careful handling of memory, since the memory requirement may vary drastically with the contraction path. In addition to this, it is always desirable to have fast and efficient operations in machine learning pipelines. To ensure efficient memory utilization, TensorKrowch employs a memory management scheme where nodes do not possess their own memory. Instead, the memory is stored within the `TensorNetwork` object, and nodes are just pointers to the corresponding addresses in this shared memory. This design choice enables memory sharing among all elements of the model, facilitating operations and allowing nodes to utilize tensors from other nodes. However, it is important to note that memory sharing is limited to elements within the same object, meaning that nodes created in different `TensorNetwork`s will not share memory. By adopting this memory management approach, TensorKrowch incorporates a range of optimizations that effectively reduce both time and memory overheads:

7.1 Operations

Tensor network operations in TensorKrowch are not simple functions, but rather instances of a class, `Operation`, that is designed to minimize redundant steps during training. Each node operation consists of two functions: one that is executed the first time the operation is called, and one that is executed in every subsequent call with the same arguments. Although these functions are similar, the former makes extra computations regarding the creation of the resultant nodes and some auxiliary operations that yield the same result in every call. For instance, when contracting two nodes, tensors are typically permuted first; how this permutation is carried out is always the same, in spite of the fact that the tensors themselves may be different.

Furthermore, to keep track of repeated calls to an `Operation`, a new object is created during the first run: `Successor`. This is a class intended for internal use that acts as a cache memory, storing the arguments that were used to call the operation, some hints regarding the auxiliary tensor-network-related computations, and a reference to the resultant nodes created. Hence, once an operation has been called, both the parent and children nodes are determined, and only their tensors will change in further contractions. This enables to reduce all the code of the contraction algorithm, which may include plain Python code to collect parent nodes, into a sequence of calls to TensorKrowch `Operations`. Because of this simplification, the order in which these operations are called is relevant. Consequently, the last operation must always be the one returning the final node that corresponds to the contraction of the entire network.

These two optimizations (having different functions for different calls, and exploiting cache memory) break the whole contraction into a set of basic tensor operations that are computed sequentially, thus improving the efficiency of the training process.

7.2 Trace

In Section 5.1 the `trace` operation was introduced as a means of keeping heavy auxiliary operations involved in the first run of the contraction out of the training loop. Tracing the model not only saves time, but also saves memory. While tracing a `TensorNetwork` model, a new memory is created to keep track of which nodes are involved in each operation of the contraction algorithm. This enables to free up the memory of data or resultant nodes that have already taken part in some operation but are not going to be needed any more in the contraction. An explicit example of the difference in memory usage that tracing the model produces can be found in Figure 1.

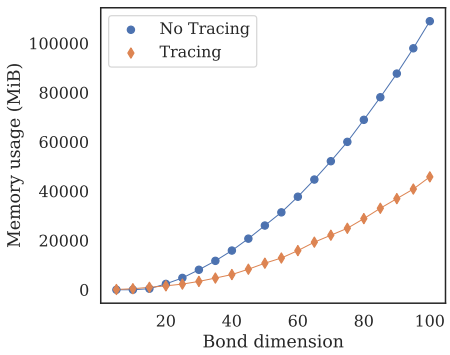


Figure 1: Comparison of the maximum memory usage of one contraction of the built-in `MPSLayer` tensor network, when the model is traced or not, using different bond dimensions. Contraction is performed in a training regime: 1) an example data tensor is passed through the model, 2) gradients are computed via backpropagation, and 3) parameters are updated according to the gradients. All the contractions are computed in CPU using a batch size of 500, with both memory modes `auto_stack` and `auto_unbind` set to `True`, both contraction arguments `inline_input` and `inline_mats` also set to `True`, and the following arguments for the `MPSLayer` model: `n_features`=1000, `in_dim`=2, `out_dim`=10. All the experiments were run on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM.

7.3 Memory Modes

Additionally, there are two modes that can change how tensor networks utilize their memory. These modes, `auto_stack` and `auto_unbind`, allow the `stack` and `unbind` operations, respectively, to reuse information stored in other parts of the memory instead of recalculating it in every contraction. This helps accelerating both, training and inference. To illustrate how these modes affect the efficiency of the tensor network model, Figure 2 presents a comparison of running times for the built-in `MPSLayer` class, when these modes are activated or deactivated.

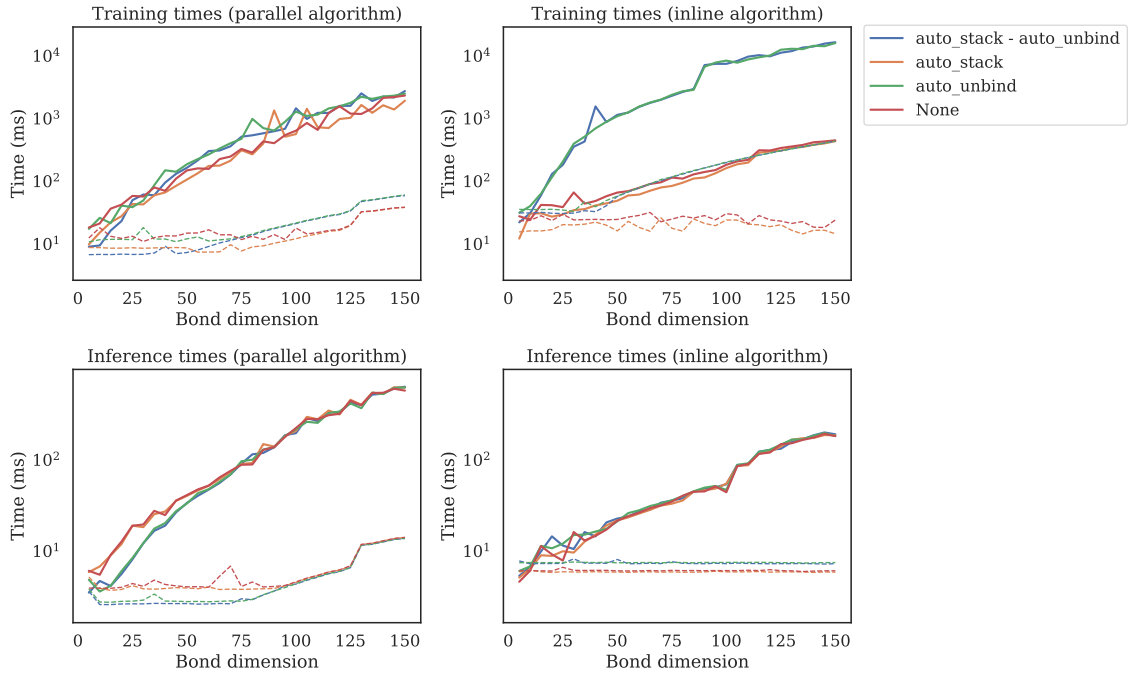


Figure 2: Comparison of running times of one contraction of the built-in `MPSLayer` tensor network, using different bond dimensions. Contraction is performed in different regimes: training/inference, parallel/inline algorithm, CPU/GPU execution, and using different combinations for the options `auto_stack` and `auto_unbind`. For training, 1) an example data tensor is passed through the model, 2) gradients are computed via backpropagation and 3) parameters are updated according to the gradients. For inference, only the example data tensor is passed to perform one contraction of the model. Parallel and inline refer to the two possible algorithms that can be used to contract `MPSLayer`, specified by the argument `inline_mats`. When `inline_mats = True` (inline), the matrices of the MPS are contracted sequentially; when `inline_mats = False` (parallel), matrices are iteratively stacked, contracted in parallel and unbound until the chain of matrices is reduced to a single one. The argument `inline_input` is always set to `False`, indicating that the contraction of the MPS nodes with the respective input nodes is performed in parallel. Solid lines represent CPU times, dashed lines represent GPU times. All the contractions are computed using a batch size of 100 and the following arguments for the `MPSLayer` model: `n_features`=100, `in_dim`=2, `out_dim`=10. All the experiments were run on an Intel Xeon CPU E5-2620 v4 with 256GB of RAM and an NVIDIA GeForce RTX 3090.

8 Additional library information

8.1 Limitations and future extensions

Tensor networks have shown promise in various fields and possess valuable properties for machine learning. However, their application in this domain is relatively recent, leaving much to be explored in terms of best practices such as initialization techniques, optimizers, and architectures. In fact, although TensorKrowch allows the creation of any tensor network, some limitations exist for using certain graphs as models. For example, contracting PEPS is known to be #P-hard [32], and finding optimal contraction paths in arbitrary graphs is also #P-hard [59]. Therefore, TensorKrowch is presented as a tool that enables rapid prototyping to explore the best techniques to be used in this domain.

We have discussed the various optimizations carried out by TensorKrowch to avoid redundant calculations as much as possible, ensuring both efficiency and simplicity. However, for scenarios where efficiency is paramount, such as quantum computer simulation [14, 15] or tensorization of neural networks for edge computing [60], there may be libraries with better resource management. Nevertheless, TensorKrowch remains useful even in those cases, as it can be part of a preliminary prototyping step.

Since TensorKrowch has a strong focus on machine learning, the priority has been to implement tensor networks understood as architectures that parameterize intricate families of functions.

Alternative perspectives, such as considering tensor networks as quantum states, have assumed a secondary role. Despite this, given the importance that the quantum information perspective can have in machine learning explainability [29, 38, 39], TensorKrowch supports tensors with complex coefficients and implements the calculation of reduced density matrices and entanglement entropy. Building more elaborated explainability tools, and exploiting TensorKrowch for quantum simulation (for instance, by leveraging Pytorch's automatic differentiation engine for calculating ground-state energies), will be the first steps to be addressed in the future.

On the other hand, other interesting operations include multiple tensor decompositions known in the numerical linear algebra community [51]. TensorKrowch allows for automatic splitting of tensors appearing in machine learning (such as those appearing in linear and convolutional layers) into MPO forms via singular value decomposition. Other tensorizations that are relevant in the machine learning literature, such as the canonical polyadic [61] and the Tucker decompositions [62], are not yet natively implemented. This is another upcoming objective to develop.

Finally, there are numerous other methods that will progressively be implemented, such as allowing for the modification of the number of edges a node has through a reshape operation, or incorporating visual feedback for observing the constructed graph, akin to the graphical notation used in this paper.

8.2 Documentation for TensorKrowch

The documentation for TensorKrowch is available online at https://joserapa98.github.io/tensorkrowch. It consists of a comprehensive user's guide and an API glossary. The user's guide provides detailed information that expands upon the topics covered in this paper. It includes more information on the installation and in-depth tutorials with examples ranging from the basic usage of `Nodes` and `Edges` to building advanced hybrid neural-tensor networks like the one discussed in [26]. The API glossary is automatically generated from the docstrings (formatted comments to code objects), containing detailed information about the public functions and classes defined in TensorKrowch.

8.3 Contribution guidelines

We welcome contributions to TensorKrowch from the wider communities interested in integrating tensor networks within machine learning frameworks, and in quantum information theory. Contributions can include feedback about the library, feature requests, bug fixes, or code contributions via pull requests.

Feedback and feature requests can be done by opening an issue on the TensorKrowch GitHub repository [47]. Bug fixes and other pull requests can be done by forking the TensorKrowch source code, making changes, and then opening a pull request to the TensorKrowch GitHub repository. Pull requests are peer-reviewed by TensorKrowch's core developers to provide feedback and/or request changes.

Contributors are expected to adhere to TensorKrowch development practices including style guidelines and unit tests. Tests are written with the PyTest Python framework and are implemented outside the module. To test installation or changes, one can download the source code from the repository, and use standard PyTest functions. For example, executing the following in a Unix terminal in the test folder runs all the tests:

```
python -m pytest -v
```

9 Concluding remarks

Machine learning research relies heavily on rapid prototyping and iteration. By building on top of PyTorch, TensorKrowch enables these features for machine learning architectures based on tensor networks. With it, it is possible to use, off the shelf, standard tensor networks (MPS, MPOs, PEPS and TTNs) either as standalone architectures or as layers in deep networks, as well as defining custom networks. In the latter case, the user has access to customizing fine details of how the network processes input data, such as specifying which parts of the input are sent to which nodes,

or fully customizing the contraction path. Our aim is that TensorKrowch contributes to the wide adoption of tensor networks by the machine learning community, and that allows to go beyond (and helps advancing) the body of knowledge generated by the communities working in quantum information theory and quantum many-body physics.

In this work we have described the main logic behind TensorKrowch, as well as the broad families of cases in which it can be used. We have also detailed the features of its building blocks, and many optimizations that are performed behind the scenes in order to obtain a fast and efficient operation. Further information on these topics, as well as end-to-end examples of use with state-of-the-art tensor-network and hybrid architectures in standard datasets can be found in the documentation website. We strongly encourage any willing user to contribute to the development of the library via its repository [47].

Acknowledgments

This work is supported by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (CEX2019-000904-S, CEX2019-000904-S-20-4, PRE2020-091917 and PID2020-113523GB-I00), Comunidad de Madrid (QUITEMAD-CM P2018/TCS-4342), Universidad Complutense de Madrid (FEI-EU-22-06), and the CSIC Quantum Technologies Platform PTI-001. This work has been financially supported by the Ministry for Digital Transformation and of Civil Service of the Spanish Government through the QUANTUM ENIA project call - Quantum Spain project, and by the European Union through the Recovery, Transformation and Resilience Plan – NextGenerationEU within the framework of the Digital Spain 2026 Agenda.

References

[1] M. Fannes, B. Nachtergaele, and R. F. Werner, Commun. Math. Phys. **144**, 443 (1992).
[2] S. R. White, Phys. Rev. Lett. **69**, 2863 (1992).
[3] G. Vidal, Phys. Rev. Lett. **91**, 147902 (2003), arXiv:quant-ph/0301063.
[4] D. Pérez-García, F. Verstraete, M. M. Wolf, and J. I. Cirac, Quantum Inf. Comput. **7**, 401 (2007), arXiv:quant-ph/0608197.
[5] G. Vidal, Phys. Rev. Lett. **99**, 220405 (2007), arXiv:cond-mat/0512165.
[6] G. Vidal, Phys. Rev. Lett. **101**, 110501 (2008), arXiv:quant-ph/0610099.
[7] G. Evenbly and G. Vidal, Phys. Rev. B **79**, 144108 (2009), arXiv:0707.1454.
[8] Y.-Y. Shi, L.-M. Duan, and G. Vidal, Phys. Rev. A **74**, 022320 (2006), arXiv:quant-ph/0511070.
[9] L. Tagliacozzo, G. Evenbly, and G. Vidal, Phys. Rev. B **80**, 235127 (2009), arXiv:0903.5017.
[10] V. Murg, F. Verstraete, Ö. Legeza, and R. M. Noack, Phys. Rev. B **82**, 205105 (2010), arXiv:1006.3095.
[11] K. Hémery, F. Pollmann, and D. J. Luitz, Phys. Rev. B **100**, 104303 (2019), arXiv:1901.05793.
[12] S.-H. Lin, M. P. Zaletel, and F. Pollmann, Phys. Rev. B **106**, 245102 (2022), arXiv:1908.07545.
[13] T. Soejima, K. Siva, N. Bultinck, S. Chatterjee, F. Pollmann, and M. P. Zaletel, Phys. Rev. B **101**, 085117 (2020), arXiv:1908.07545.
[14] F. Pan, K. Chen, and P. Zhang, Phys. Rev. Lett. **129**, 090502 (2022), arXiv:2111.03011.
[15] C. Oh, M. Liu, Y. Alexeev, B. Fefferman, and L. Jiang, Tensor network algorithm for simulating experimental Gaussian boson sampling (2023), arXiv:2306.03709.
[16] S. Sánchez-Ramírez, J. Conejero, F. Lordan, A. Queralt, T. Cortes, R. M. Badia, and A. García-Saez, in *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)* (2021) pp. 1–8, arXiv:2201.06620.
[17] U. Schollwöck, Rev. Mod. Phys. **77**, 259 (2005), arXiv:cond-mat/0409292.
[18] I. V. Oseledets, SIAM J. Sci. Comput. **33**, 2295 (2011).
[19] I. V. Oseledets, Comput. Method Appl. Math. **11**, 382 (2011).
[20] S. V. Dolgov and D. V. Savostyanov, Comput. Phys. Commun. **246**, 106869 (2020), arXiv:1903.11554.
[21] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos, IEEE Trans. Signal Process. **65**, 3551 (2017), arXiv:1607.01668.

[22] S. V. Dolgov and D. V. Savostyanov, Tensor product approach to modelling epidemics on networks (2022), arXiv:2209.03756.
[23] E. Stoudenmire and D. J. Schwab, in *Advances in Neural Information Processing Systems*, Vol. 29, edited by D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Curran Associates, Inc., 2016) pp. 4799–4807, arXiv:1605.05775.
[24] A. Novikov, M. Trofimov, and I. V. Oseledets, Bull. Pol. Acad. Sci. Tech. Sci. **66**, 789 (2018), arXiv:1605.03795.
[25] J. I. Cirac, D. Pérez-García, N. Schuch, and F. Verstraete, Rev. Mod. Phys. **93**, 045003 (2021), arXiv:2011.12127.
[26] I. Glasser, N. Pancotti, and J. I. Cirac, IEEE Access **8**, 68169 (2020), arXiv:1806.05964.
[27] J. Miller, G. Rabusseau, and J. Terilla, in *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 130, edited by A. Banerjee and K. Fukumizu (PMLR, 2021) pp. 3079–3087, arXiv:2003.01039.
[28] J. Lopez-Piqueres, J. Chen, and A. Perdomo-Ortiz, Mach. Learn.: Sci. Technol. **4**, 035009 (2023), arXiv:2211.09121.
[29] D. Liu, S.-J. Ran, P. Wittek, C. Peng, R. Blázquez García, G. Su, and M. Lewenstein, New J. Phys. **21**, 073059 (2019), arXiv:1710.04833.
[30] S. Cheng, L. Wang, T. Xiang, and P. Zhang, Phys. Rev. B **99**, 155131 (2019), arXiv:1901.02217.
[31] T. Vieijra, L. Vanderstraeten, and F. Verstraete, Generative modeling with projected entangled-pair states (2022), arXiv:2202.08177.
[32] F. Verstraete, M. M. Wolf, D. Pérez-García, and J. I. Cirac, Phys. Rev. Lett. **96**, 220601 (2006), arXiv:quant-ph/0601075.
[33] J. Wang, C. Roberts, G. Vidal, and S. Leichenauer, Anomaly detection with tensor networks (2020), arXiv:2006.02516.
[34] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, in *Advances in Neural Information Processing Systems*, Vol. 28, edited by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Curran Associates, Inc., 2015) arXiv:1509.06569.
[35] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, Speeding-up convolutional neural networks using fine-tuned CP-decomposition (2014), arXiv:1412.6553.
[36] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song, in *Advances in Neural Information Processing Systems*, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) arXiv:1906.09777.
[37] K. Zhang, C. Hawkins, X. Zhang, C. Hao, and Z. Zhang, in *Proceedings of the ICLR 2021 Workshop of Hardware Aware Efficient Training* (2021) arXiv:2104.03420.
[38] J. Tangpanitanon, C. Mangkang, P. Bhadola, Y. Minato, D. G. Angelakis, and T. Chotibut, New J. Phys. **24**, 053032 (2022), arXiv:2112.08628.
[39] B. Aizpurua, S. Palmer, and R. Orús, Tensor networks for explainable machine learning in cybersecurity (2024), arXiv:2401.00867.
[40] A. Pozas-Kerstjens, S. Hernández-Santana, J. R. Pareja Monturiol, M. Castrillón López, G. Scarpa, C. E. González-Guillén, and D. Pérez-García, Physics solutions for machine learning privacy leaks (2022), arXiv:2202.12319.
[41] H. Xiao, K. Rasul, and R. Vollgraf, Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms (2017), arXiv:1708.07747.
[42] J. Miller, TorchMPS (2019), `https://github.com/jemisjoky/TorchMPS`.
[43] M. Usvyatsov, R. Ballester-Ripoll, and K. Schindler, J. Mach. Learn. Res. **23**, 1 (2022), arXiv:2206.11128.
[44] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, J. Mach. Learn. Res. **20**, 1 (2019), arXiv:1610.09555.
[45] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, TensorNetwork: A library for physics and machine learning (2019), arXiv:1905.01330.
[46] J. Gray, J. Open Source Softw. **3**, 819 (2018).
[47] J. R. Pareja Monturiol, D. Pérez-García, and A. Pozas-Kerstjens, TensorKrowch (2023), `https://github.com/joserapa98/tensorkrowch`.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, in *Advances in Neural Information Processing Systems*, Vol. 32, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) arXiv:1912.01703.
[49] R. Orús, Ann. Phys. **349**, 117 (2014), arXiv:1306.2164.
[50] J. C. Bridgeman and C. T. Chubb, J. Phys. A: Math. Theor. **50**, 223001 (2017), arXiv:1603.03039.
[51] T. G. Kolda and B. W. Bader, SIAM Rev. **51**, 455 (2009).
[52] B. Pirvu, V. Murg, J. I. Cirac, and F. Verstraete, New J. Phys. **12**, 025012 (2010), arXiv:0804.3976.
[53] J. Martyn, G. Vidal, C. Roberts, and S. Leichenauer, Entanglement and tensor networks for supervised image classification (2020), arXiv:2007.06082.
[54] J. A. Reyes and E. M. Stoudenmire, Mach. Learn.: Sci. Technol. **2**, 035036 (2021), arXiv:2001.08286.
[55] D. G. A. Smith and J. Gray, J. Open Source Softw. **3**, 753 (2018).
[56] V. Zauner-Stauber, L. Vanderstraeten, M. T. Fishman, F. Verstraete, and J. Haegeman, Phys. Rev. B **97**, 045145 (2018), arXiv:1701.07035.
[57] J. Gray and S. Kourtis, Quantum **5**, 410 (2021), arXiv:2002.01935.
[58] A. Acuaviva, V. Makam, H. Nieuwboer, D. Pérez-García, F. Sittner, M. Walter, and F. Witteveen, in *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)* (2023) pp. 328–362, arXiv:2209.14358.
[59] C. Damm, M. Holzer, and P. McKenzie, Comput. Complex. **11**, 54 (2002).
[60] C. Hawkins, X. Liu, and Z. Zhang, SIAM J. Math. Data Sci. **4**, 46 (2022), arXiv:2010.08689.
[61] N. Kargas and N. D. Sidiropoulos, IEEE Trans. Signal Process. **69**, 1097 (2021).
[62] Y. Liu and M. K. Ng, Knowl.-Based Syst. **241**, 108171 (2022).