ContracTN: A Tensor Network Library Designed for Machine Learning

Jacob Miller
Mila, Université de Montréal
Montréal QC, Canada
jmjacobmiller@gmail.com

Guillaume Rabusseau
Mila, Université de Montréal
Montréal QC, Canada
grabus@iro.umontreal.ca

Abstract

Although first developed for the needs of quantum many-body physics and quantum computing, tensor networks (TNs) are increasingly being deployed to solve a wide range of problems in machine learning, optimization, and applied mathematics. Inspired by the distinct implementation challenges of TN methods in these new settings, we present ContracTN, a lightweight Python library for general-purpose TN calculations. Beyond the use of the dense tensor cores supported in standard TN libraries, ContracTN also supports the use of copy tensors, parameter-free objects which allow diverse concepts like batch computation, elementwise multiplication, and summation to be expressed entirely in the language of TN diagrams. The contraction engine of ContracTN also implements a novel form of stabilization, which largely mitigates the issue of numerical overflow arising from the use of low-precision machine learning libraries for TN contraction. Overall, we wish to popularize a collection of methods which have proven invaluable in implementing efficient and robust TN models, in the hope that this can help catalyze the wider adoption of TN methods for problems in machine learning.

1 Introduction

Tensor network (TN) methods were originally developed for the theoretical and computational needs of quantum many-body physics and quantum computing [14]. In this setting, TNs serve as an efficient means of describing strongly correlated states of quantum matter, which enable the straightforward calculation of many physical quantities of interest, chiefly the expected energy of a TN state relative to a problem-dependent Hamiltonian. Although there exist many delicate implementation issues in this setting, the design of TN libraries is generally biased towards the use of densely-represented tensor cores, with a focus on problems that require a relatively small number of contraction operations to reach an answer. Great value is placed in such settings on numerical accuracy, supporting the use of double-precision floating point numbers, whose magnitudes can vary from approximately 10^{-300} to 10^{300}.

More recent applications of TNs to machine learning, optimization, and counting problems make use of much of the theoretical machinery previously developed for quantum problems, but vary significantly in their domain-specific demands. For starters, machine learning models trained on large datasets utilize many repeated computations carried out in a batched fashion, typically within a machine learning library supporting evaluation on a hardware accelerator. The use of single-precision floating point formats is standard here, whose limited dynamic range (approximately 10^{-40} to 10^{40}) is poorly adapted for the exponential growth in magnitude involved in generic TN contraction processes. Non-physics applications of TNs also frequently employ copy tensors[1], hyperdiagonal tensor cores

[1] Copy tensors are also referred to as *hyperedges* [18, 2]

Second Workshop on Quantum Tensor Networks in Machine Learning, 35th Conference on Neural Information Processing Systems (NeurIPS 2021)

which enable copying discrete classical data, and whose correct implementation can lead to significant computational savings relative to a naive treatment as dense tensor cores.

All of these considerations have been instrumental to the design of ContracTN, which takes a distinct approach to TN layout and evaluation. ContracTN permits the use of different types of tensor cores, allowing not only standard densely-represented tensors, but also copy tensors and "template" tensors, where multiple cores arise as duplicates of a single dense parameter tensor. While the latter is useful for recurrent models such as uniform matrix product states [16, 12], we find the former to be invaluable for expressing a range of disparate computational operations in purely TN language. ContracTN utilizes the open-source network analysis library NetworkX [9] to represent the graphical structure of general TN models, with computation facilitated by converting this graphical structure into a single string formatted in the manner of NumPy's multipurpose `einsum` function [11]. This string is then fed to the open-source `einsum` utility opt_einsum [5], which determines an efficient order of contraction for the TN and performs this contraction in a backend-agnostic manner. We use a custom version of the opt_einsum contraction function, which uses a separate scalar quantity to stabilize the norm of intermediate contraction terms. This largely avoids the issue of overflow, at the cost of utilizing a "split" representation of contraction outputs as a rescaled tensor and a scalar value indicated the (logarithm of the) amount of rescaling.

We now illustrate these issues of implementing TN libraries for machine learning with code examples, and the full ContracTN library can be found at `https://github.com/jemisjoky/ContracTN`.

2 Tensor Network Basics

Tensor networks (TNs) are a general formalism for efficiently encoding high-order tensors, which utilize a mixture of graph-theoretic and linear-algebraic machinery in their operation. A generic TN is described by three ingredients:

1. A graph G with node set V and edge set E, whose edges are allowed to connect to either one or two nodes. The former are referred to as "free" or "open" edges, and the latter as "bond" or "closed" edges.
2. A map $D : E \to \mathbb{Z}^+$ associating each edge $e \in E$ of G to a non-negative number D_e, called the "bond dimension" of that edge.
3. A map T associating each node $n \in N$ to a tensor T_n over a field[2] $\mathbb{K}$. If n has degree k, such that it is incident to edges $e_1, e_2, \dots, e_k$, then T_n must be a kth order tensor satisfying $T_n \in \mathbb{K}^{d_{e_1} \times d_{e_2} \times \cdots \times d_{e_k}}$

It is clear from the above that any two tensors $T_n, T_{n'}$ arising from neighboring nodes $n, n' \in N$ have a pair of modes of equal dimension, namely those associated with the edge $e \in E$ joining n and n'. The basic computational operation in a TN is *tensor contraction*, wherein two tensors are multiplied together along these modes of equal dimension, in a manner which generalizes matrix multiplication. This computational operation is paired with a change in the underlying G, where nodes n and n' merge into a new node n'' whose tensor $T_{n''}$ is the output of the tensor contraction.

The tensor T implicitly represented by a TN is that associated with the single node arising from the contraction of all bond edges of a G, with the number of free edges of G giving the order of T. The specific order in which edges are contracted has no impact on the value of T, but has a significant impact on the complexity of computing T. Finding the optimal order of contraction is an NP-hard problem [4], but in practice many algorithms exist that give near-optimal contraction ordering on large-scale problems [17, 6, 8].

3 Copy Tensors

Given a choice of basis $\{f_i\}_{i=1}^d$ for a d-dimensional vector space, one can define a family of *copy tensors* Δ_n, for $n \geq 1$, as

$$\Delta_n = \sum_{i=1}^{d} f_i^{\otimes n} = \sum_{i=1}^{d} f_i \otimes f_i \otimes \cdots \otimes f_i. \tag{1}$$

[2] In general, $\mathbb{K}$ needs only be a semiring, but the case of $\mathbb{K} = \mathbb{R}, \mathbb{C}$ is most common.

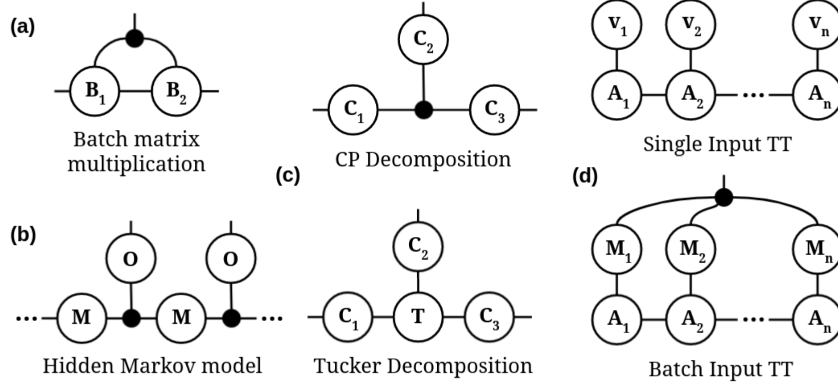

Figure 1: Illustration of the utility of copy tensors for expressing diverse constructions in machine learning. (a) Batch matrix multiplication has a simple TN description using a third-order copy tensor. (b) Hidden Markov models, as with more general probabilistic graphical models, utilize copy tensors to propagate latent states to multiple locations. (c) The CP and Tucker decompositions share the same TN topology, but differ by the placement of a copy vs. a dense tensor in the center. This leads the CP decomposition to have a drastically lower computational cost, an advantage which is lost in libraries which only allow the definition of dense tensor cores. (d) In the use of tensor trains (TT) for supervised learning [21, 13], each n-dimensional input is associated with a separate vector, giving an embedding as an order-n rank-1 tensor. Processing a batch of images as a single TN contraction is achieved by representing each batch of vectors as a matrix, then joining the batch indices of each matrix by a single copy tensor of order $n+1$.

While copy tensors are less commonly used in quantum settings, due to the restrictions of the quantum no-cloning principle [23], they are invaluable for expressing many common constructions in machine learning and applied mathematics within the language of TNs (Figure 1). Although each Δ_n is an nth order tensor, the hyperdiagonal nature of these tensors permits an implementation in software which is significantly more efficient than would be possible with a dense representation of the same tensor.

As a concrete example, consider a TN with $n+1$ cores, consisting of a single copy tensor Δ_{n+1} connected to n vectors v_i of dimension d. It is straightforward to verify from Equation 1 that the output of this contraction is a vector equal to the element-wise (i.e. Hadamard) product of the n vectors, which can be computed in time $\mathcal{O}(nd)$. By comparison, TN libraries which don't support efficient implementations of copy tensors must either use case-by-case methods to handle basis-dependent operations, or else employ a dense representation with memory cost $\mathcal{O}(d^n)$.

We define and contract this TN in the script below using ContracTN, for $n = 100$ and $d = 2$, and find that such a contraction process takes only a few milliseconds. By contrast, a dense representation of the same network is infeasible for any modern computer, owing to the $2^{101} \approx 10^{30}$ elements naively contained in the central copy tensor.

```
import numpy as np
from contractn import TN
tn = TN()

# Add central copy tensor of order 101
copy_node = tn.add_copy_node(101)

# Connect vectors to all but one edge of the copy tensor
for i in range(100):
    vec = np.array([1, 0.99])
    vec_node = tn.add_dense_node(vec)
    # Connect i'th axis of copy_node to 0'th axis of vec_node
    tn.connect_nodes(copy_node, vec_node, i, 0)
```

```
print(tn.contract())  # array([1., 0.36603234])
%time tn.contract()   # 7.0 ms
```

The ability to deal with copy tensors in an efficient manner is a significant asset to a TN library, as it allows concepts such as elementwise multiplication and batch computation to be handled entirely with TN primitives. Although it might seem like implementing such functionality efficiently would require a significant amount of additional program logic, we will see shortly how copy tensors can be seamlessly integrated with dense tensor cores in a fully general fashion.

4 Einsum as a TN Specification Language

One important observation we leverage in the design of ContracTN is the intimate relationship between the structure of TNs and the functionality of NumPy's `einsum` [11], a function specification utility which has inspired similar implementations across many numerical computing libraries. `einsum` utilizes a single string to specify a pattern of contractions between a fixed number of input tensors, yielding a single output tensor. Each nth order tensor is described by a substring of length n, whose component symbols describe the pattern of connectivity between different modes of the input tensors. This string, along with the dense input tensors, are fed to `einsum`, which contracts together all tensors and returns a single output. As a representative example, a batched matrix multiplication operation like that in Figure 1a would be described by the string `abc,acd->abd`.

While each valid `einsum` string specifies a computational operation equivalent to contracting a TN, it is easy to see further that this representation is *complete* for TNs[3]. More precisely, the graphical structure of any TN can be fully described by an `einsum` string, which is unique up to relabeling of symbols and reordering of input terms. Crucially, this completeness applies not just to standard TNs formed from dense tensor cores, but also to those containing copy tensors. In this case, copy tensors don't contribute to the input tensors of `einsum`, but rather serve to identify or sum over indices in a TN.

Although typical implementations of `einsum` are limited in the number of allowed symbols and their ability to optimize over the contraction order of input tensors, ContracTN makes use of the excellent opt_einsum library [5], an `einsum` utility well-adapted to the needs of large TNs. Contraction in ContracTN is handled by simply converting the network structure of a given TN into a single `einsum` string, and then relying on opt_einsum to find a contraction order and compute the output. This has the advantage of disentangling the issues of network layout and computation inherent to TNs, while also enabling ContracTN to be used with any deep learning library supported by opt_einsum, such as PyTorch [15], TensorFlow [1], and JAX [7]. We illustrate this conversion of network structure to `einsum`-style strings in the following example, where two TNs describing 3rd-order CP and Tucker decompositions (c.f. Figure 1c) are given.

```
# Initialize TNs for 3rd-order Tucker and CP decompositions
cp = TN()
tucker = TN()

# Add central "hub" cores
cp_hub = cp.add_copy_node(3)
tucker_hub = tucker.add_dense_node(np.ones((4, 4, 4)))

# Connect each hub to three factor matrices
for i in range(3):
    mat = np.eye(4, 10)
    cp_mat = cp.add_dense_node(mat)
    tucker_mat = tucker.add_dense_node(mat)
    cp.connect_nodes(cp_hub, cp_mat, i, 0)
    tucker.connect_nodes(tucker_hub, tucker_mat, i, 0)
```

[3] Although this completeness doesn't seem to have been explicitly mentioned in the literature, it should be obvious to researchers familiar with TNs and the use of `einsum`. For an illustration of the versatility of `einsum`, see the in-depth blog posts [3] and [19].

```
print(cp.einsum_str)      # "ac,ad,ae->cde"
print(tucker.einsum_str)  # "abc,ae,bf,cg->efg"
```

Observe that even though both TNs contain 4 component tensor cores, only the dense cores are represented in the `einsum` string, with the copy tensor in the CP decomposition serving simply to link together the left indices of its factor matrices.

5 Stabilizing TN Contractions

One recurring problem in implementing large TN models is the large range of magnitudes occurring during contraction, an issue which comes from the multiplicative nature of TN contraction. In particular, given a TN containing n core tensors A_i which contract together to give an output tensor T, rescaling each core A_i by a scalar factor of κ leads to a rescaled output tensor $\kappa^n T$. For large values of n (e.g. $n \sim 100 - 1000$), this leads to a computational process which is extremely sensitive to small changes in the core tensor values, leading to difficulties in initializing and adjusting core tensors weights during learning. This problem is made worse by the tendency of deep learning libraries to favor low-precision floating point formats, so that for a large TN, changing the magnitude of core tensors by just 20% can change an output that underflows to 0 into one that overflows to infinity.

A standard method for handling this issue is to convert the TN core tensors into some type of canonical form [22, 20] after any modification in the core tensor values, ensuring that the TN describes a tensor T with ℓ_2-norm $\|T\|_2 = 1$. While this approach is usually sufficient for computing expectation values in the setting of quantum many-body physics, it has several notable disadvantages: (a) Convenient canonical forms don't exist for arbitrary TNs, and converting core tensors into a canonical form typically use algorithms specialized to one graph structure; (b) Conversion to canonical form is computationally expensive, and in many settings has a higher asymptotic cost than the actual TN contraction itself[4]; (c) When the TN cores are being used for several distinct tasks, it isn't always possible to find a single canonical form which stabilizes the output used for both tasks.

An important example of (c) is probabilistic modeling with Born machines [10], where the elements $T_{x_1,\dots,x_n}$ and norm $\|T\|_2$ of a tensor T are used jointly to encode a probability distribution as $\Pr(x_1, \dots, x_n) = |T_{x_1,\dots,x_n}|^2 / \|T\|_2^2$. Given the huge difference in magnitude between these two quantities (typically, $\Pr(x_1, \dots, x_n) = \exp(-\mathcal{O}(n))$), any fixed rescaling of core tensor values which stabilizes the value of one quantity will necessarily lead the value of the other to overflow or underflow. While similar issues arise in deep neural networks, in this case numerical stability can be achieved through the use of dynamical normalization methods, such as batch and layer normalization, which aren't easily expressable in terms of TN primitives.

ContracTN uses a simple form of dynamical rescaling to address this issue, which largely solves the problem of overflow and underflow in TN contraction. The idea behind this technique is to work with a "split" representation of a tensor T as $T = \exp(c) \cdot \hat{T}$, where $\hat{T}$ is a rescaled version of T whose elements satisfy $|T_{x_1,\dots,x_k}| \sim 1$, and c is a real scalar quantity giving the logarithm of the rescaling factor relating T and $\hat{T}$. This condition on the rescaled tensor $\hat{T}$ can always be maintained at each step of the contraction process, by use of the equality:

$$T = \exp(c) \cdot \hat{T} = \exp(c + \ln(\kappa)) \cdot \hat{T}/\kappa, \quad \forall \kappa > 0. \tag{2}$$

ContracTN uses a slight modification of the opt_einsum `contract` function, which utilizes Equation 2 to maintain numerical stability throughout the computation (Algorithm 1). This modification entails negligible additional computational costs, yet is able to accurately return contraction results which would otherwise diverge. Given that this split format might be unfamiliar to users, the contraction utilities of ContracTN all accept a Boolean argument `split_format`, with the default behavior of `split_format=False` corresponding to standard TN contraction. We illustrate the use of this

[4] A representative example is supervised learning with tensor trains [21, 13], where an n-core TT with bond dimension D requires time $\mathcal{O}(nD^2)$ to generate a prediction for a single input, and time $\mathcal{O}(nD^3)$ to convert all TT cores into canonical form.

Algorithm 1 Stable contraction algorithm for TN with n cores

```
function STABLE_CONTRACT(cores = [T_1, ..., T_n], c_pairs = [(i_1, j_1), ..., (i_{n-1}, j_{n-1})])
    c = 0.                                         ▷ c stores the log rescale factor
    for (i, j) in c_pairs do                       ▷ c_pairs specifies contraction order
        R = cores[i], S = cores[j]                 ▷ The two tensors to contract together
        T = contract(R, S)                         ▷ contract equivalent to tensordot
        η = ‖T‖_1                                  ▷ The choice of norm isn't important
        κ = η/numel(T)                             ▷ numel gives number of elements in T
        c = c + ln(κ), T = T/κ                     ▷ Rescaling utilizing Equation 2
        cores.append(T)
    T̂ = cores[0]                                   ▷ By now, cores only has one element
    return (T̂, c)                                  ▷ Gives output in split format
```

stabilized contraction in the following simple example of multiplying a sequence of 1000 3×3 matrices with an input vector[5].

```
# Initialize connected chain of one vector and 1000 3x3 matrices
tn = TN()
prev_node = tn.add_dense_node(np.ones((3,)))
for i in range(1000):
    mat_node = tn.add_dense_node(np.ones((3, 3)))
    tn.connect_nodes(prev_node, mat_node, -1, 0)
    prev_node = mat_node

print(tn.contract())
    # [inf inf inf]
print(tn.contract(split_format=True))
    # (array([1., 1., 1.]), array(1098.61228867))
```

The appropriate use of this split format output depends on the problem at hand. For tasks which only rely on the relative magnitude of different components of the output, the first term can be used as-is, with the second scalar output discarded. For probabilistic modeling in Born machines using a negative log likelihood (NLL) loss, the scalar output will represent an additive contribution to this loss, which is typically significantly larger than the contributions from the first tensor output. No matter how it is used, the slight additional complexity of this split output is worth the benefit of removing the most common sources of underflow and overflow, with the additional log-scale register allowing the representation of tensors whose elements have magnitudes in the range of approximately $10^{-10^{40}}$ to $10^{10^{40}}$.

6 Conclusion

We have introduced a new TN library which utilizes several important design principles, chiefly: (a) The use of copy tensors as a first-class ingredient within TN models; (b) The separation of network layout and computational concerns, with the former simply converting the graphical structure into a string which can be passed to an appropriately efficient and robust `einsum` function for contraction; (c) The use of a split format for tensors as a means of mitigating the numerical overflow and underflow problems common in applying TNs to difficult real-world problems. We include the current code for ContracTN, including all scripts given above, in the supplemental material, and hope that its design can serve as a source of inspiration for current and future libraries for general-purpose TN computations.

[5] We leave it as an exercise to the reader to verify that the scalar quantity output as the second component of the stabilized contraction is indeed the correct value, namely $1000 \cdot \ln(3)$.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] R. Bailly, F. Denis, and G. Rabusseau. Recognizable series on hypergraphs. In *International Conference on Language and Automata Theory and Applications*, pages 639–651. Springer, 2015.

[3] O. Bilaniuk. Einstein summation in numpy. `https://obilaniu6266h16.wordpress.com/2016/02/04/einstein-summation-in-numpy/`, 2016.

[4] L. Chi-Chung, P. Sadayappan, and R. Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7(02):157–168, 1997.

[5] G. Daniel, J. Gray, et al. Opt_einsum-a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

[6] J. M. Dudek, L. Duenas-Osorio, and M. Y. Vardi. Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *arXiv preprint arXiv:1908.04381*, 2019.

[7] R. Frostig, M. J. Johnson, and C. Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[8] J. Gray and S. Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021.

[9] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[10] Z.-Y. Han, J. Wang, H. Fan, L. Wang, and P. Zhang. Unsupervised generative modeling using matrix product states. *Phys. Rev. X*, 8:031012, 2018. doi: 10.1103/PhysRevX.8.031012. URL `https://link.aps.org/doi/10.1103/PhysRevX.8.031012`.

[11] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with NumPy. *Nature*, 585 (7825):357–362, 2020.

[12] J. Miller, G. Rabusseau, and J. Terilla. Tensor networks for probabilistic sequence modeling. In *International Conference on Artificial Intelligence and Statistics*, 2021.

[13] A. Novikov, M. Trofimov, and I. Oseledets. Exponential machines. In *International Conference on Learning Representations*, 2017.

[14] R. Oruś. A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States. *Annals Phys.*, 349:117–158, 2014. doi: 10.1016/j.aop.2014.06.013.

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[16] V. Pestun, J. Terilla, and Y. Vlassopoulos. Language as a matrix product state. *arXiv preprint arXiv:1711.01416*, 2017.

[17] R. N. Pfeifer, J. Haegeman, and F. Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.

[18] E. Robeva and A. Seigal. Duality of Graphical Models and Tensor Networks. *Information and Inference: A Journal of the IMA*, 8(2):273–288, 06 2018. ISSN 2049-8772. doi: 10.1093/imaiai/iay009. URL `https://doi.org/10.1093/imaiai/iay009`.

[19] T. Rocktäschel. Einsum is all you need - einstein summation in deep learning. `https://rockt.github.io/2018/04/30/einsum`, 2018.

[20] Y.-Y. Shi, L.-M. Duan, and G. Vidal. Classical simulation of quantum many-body systems with a tree tensor network. *Physical review a*, 74(2):022320, 2006.

[21] E. Stoudenmire and D. J. Schwab. Supervised learning with tensor networks. *Advances in Neural Information Processing Systems*, pages 4806–4814, 2016.

[22] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical review letters*, 91(14):147902, 2003.

[23] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886): 802–803, 1982.