frontiers | Frontiers in Big Data

TYPE Original Research
PUBLISHED 30 May 2024
DOI 10.3389/fdata.2024.1363978

OPEN ACCESS

EDITED BY
Yanqing Zhang,
Yunnan University, China

REVIEWED BY
Yubai Yuan,
The Pennsylvania State University (PSU), United States
Yuanyuan Ju,
Kunming University of Science and Technology, China

*CORRESPONDENCE
Elizabeth Newman
✉ elizabeth.newman@emory.edu

RECEIVED 31 December 2023
ACCEPTED 29 April 2024
PUBLISHED 30 May 2024

CITATION
Newman E, Horesh L, Avron H and Kilmer ME (2024) Stable tensor neural networks for efficient deep learning.
Front. Big Data 7:1363978.
doi: 10.3389/fdata.2024.1363978

Stable tensor neural networks for efficient deep learning

Elizabeth Newman[1]*, Lior Horesh[2], Haim Avron[3] and Misha E. Kilmer[4]

[1]Department of Mathematics, Emory University, Atlanta, GA, United States, [2]Mathematics and Theoretical Computer Science, IBM TJ Watson Research Center, Yorktown, NY, United States, [3]Department of Applied Mathematics, Tel Aviv University, Tel Aviv-Yafo, Israel, [4]Department of Mathematics, Tufts University, Medford, MA, United States

Learning from complex, multidimensional data has become central to computational mathematics, and among the most successful high-dimensional function approximators are deep neural networks (DNNs). Training DNNs is posed as an optimization problem to learn network weights or parameters that well-approximate a mapping from input to target data. Multiway data or tensors arise naturally in myriad ways in deep learning, in particular as input data and as high-dimensional weights and features extracted by the network, with the latter often being a bottleneck in terms of speed and memory. In this work, we leverage tensor representations and processing to efficiently parameterize DNNs when learning from high-dimensional data. We propose tensor neural networks (t-NNs), a natural extension of traditional fully-connected networks, that can be trained efficiently in a reduced, yet more powerful parameter space. Our t-NNs are built upon matrix-mimetic tensor-tensor products, which retain algebraic properties of matrix multiplication while capturing high-dimensional correlations. Mimeticity enables t-NNs to inherit desirable properties of modern DNN architectures. We exemplify this by extending recent work on stable neural networks, which interpret DNNs as discretizations of differential equations, to our multidimensional framework. We provide empirical evidence of the parametric advantages of t-NNs on dimensionality reduction using autoencoders and classification using fully-connected and stable variants on benchmark imaging datasets MNIST and CIFAR-10.

KEYWORDS

tensor algebra, deep learning, machine learning, image classification, inverse problems

1 Introduction

With the explosion of computing resources, including cloud-based storage and accessible advanced hardware, learning from large-scale, multiway data has become possible. Two distinct fields have emerged as the gold standards for handling multidimensional data: tensor analysis for featurization and compression and deep learning for high-dimensional function approximation. Both deep learning and tensor methods have achieved strong performance in image and video recognition (Vasilescu and Terzopoulos, 2002; Krizhevsky et al., 2012), medical imaging analysis (Omberg et al., 2007; Ronneberger et al., 2015), spatiotemporal weather analysis (Chattopadhyay et al., 2020; Li et al., 2020), and more. This work focuses on leveraging advantages of tensor methods to enhance deep learning design.

Fundamentally, deep learning approximates mappings from (high-dimensional) inputs (e.g., images) to targets (e.g., classes) using deep neural networks (DNNs), which are simply nonlinear, composite functions parameterized by learnable weights. Despite the

success and flexibility of DNNs, the storage and computational costs to design and apply these models can be a significant impediment—there can be millions of network weights and learning requires an immense amount of time and top-of-the-line computational hardware (e.g., GPU clusters).

These computational challenges become bottlenecks for the classic feed-forward neural network, which builds DNNs using dense linear operators (matrices). Such operations uses network weights in an highly inefficient manner, and composing many of these dense matrices can require millions of weights, which is both computationally demanding and can lead to algorithmic problems, such as overfitting. To reduce these inefficiencies, we propose a new type of fully-connected layer that replaces dense linear operators with dense tensor operators. The proposed tensor operators can reduce the number of network weights by an order of magnitude, that leverage the inherent multidimensionality of the input data, and offer the potential for distributed computation. Thus, we call our architecture tensor neural networks (t-NNs).

The foundation of t-NNs is the $\star_M$-product (pronounced "star-M"), a family of tensor-tensor products which induces an algebraic structure on a multidimensional space (Kernfeld et al., 2015). The $\star_M$-framework provably encodes information more efficiently than traditional matrix algorithms (Kilmer et al., 2021) and has had success facial recognition (Hao et al., 2013), tomographic image reconstructions (Soltani et al., 2016; Newman and Kilmer, 2020), video completion (Zhang et al., 2014), image classification (Newman et al., 2018), and solving tensor linear systems (Ma and Molitor, 2022). We call the $\star_M$-product *matrix-mimetic*; that is, familiar notions such as the identity and transpose are well-defined for the multilinear operation. The advantages of processing data multidimensionally including better leveraging inherit multiway structure and reducing the number of learnable network weights by an order of magnitude. The matrix-mimeticity enables the proposed t-NNs to naturally extend familiar deep learning concepts, such as backward propagation and loss functions, and non-trivial architectural designs to tensor space. We propose two additional extensions: tensor-based loss functions and a stable multidimensional framework, motivated by Haber and Ruthotto (2017), that brings topological advantages of featurization.

1.1 Our contributions

Because of the popularity of this area of research, we want to clarify the objectives and contributions of this paper from the outset. Our contributions are the following:

- Tensor algebra and processing for efficient parameterization: we introduce a basic framework for t-NNs, describe the associated tensor algebra, and demonstrate the inherit properties from stable network architectures. We also derive the training algorithm for t-NNs, leveraging matrix-mimeticity for elegant formulations. We show that this tensor parameterization, compared to an equivalent matrix approach, can reduce the number of weights by an order of magnitude.
- Tubal loss functions: our the algebraic structure imposed by the $\star_M$-product is applied end-to-end. This includes defining new loss functions based on the outputs of the t-NN, which are no longer scalars, but the high-dimensional analog called *tubes*. This requires a new definition of *tubal functions*, and opens the door to a wide range of new evaluation metrics. These metrics offer more rigorous requirements to fit the training data, and hence can yield networks that generalize better.
- Stable t-NNs: we demonstrate how matrix-mimeticity preserves of desirable network architecture properties, specifically stability. This will enable the development of deeper, more expressive t-NNs.
- Open-source code: for transparency and to expand the use of t-NNs, we provide open-source at https://github.com/elizabethnewman/tnn.
- Scope: our goal is to explore a new algebraic structure imposed on neural networks and its the advantages over equivalent architectures. This paper serves as the introduction of t-NNs and, similar to the original neural networks, we consider fully-connected layers only. We acknowledge that to obtain state-of-the-art results, we would need tools like convolutional and subsampling layers and significant hyperparameter tuning; however, these are outside the scope of this paper. Convolutional layers apply multiple translation-invariant filters to extract local connections; our t-NNs examine the global structure of the data. Subsampling or pooling layers reduce the dimensionality of our data and hence provide multi-scale features; our t-NNs use no pooling in order to preserve the algebraic structure. We address extensions of t-NNs to convolutional and subsampling layers in the conclusions.

1.2 Organization

This paper is organized as follows. In Section 2, we give a brief outline of related work combining tensors and deep learning. In Section 3, we give the background notation on tensor-tensor products. In Section 4, we formally introduce tensor neural networks (t-NNs) and tubal loss functions. In Section 5, we extend t-NNs to stable architectures and outline a Hamiltonian-inspired architecture. In Section 6, we provide numerical support for using t-NNs over comparable traditional fully-connected neural networks. In Section 7, we discuss future work including implementations for higher-order data and new t-NN designs.

2 Related work

The high dimensional nature of neural network weights has driven the need to reduce the number of weights through structure. Early studies, such as LeCun et al. (1989), demonstrated that neural networks could learn faster from less data and generalize better by removing redundant weights. Following the observation, several works showed that structured weights, such as convolutions (Krizhevsky et al., 2012), low rank weight matrices (Denil et al.,

2013), and Kronecker-structured matrices (Jagtap et al., 2022), could perform well with significantly fewer parameters.

Tensor methods for compression high dimensional data and operators grew in popularity concurrently with the development of structured operators for neural networks. Many popular tensor frameworks are designed to featurize multiway arrays (Tucker, 1966; Carroll and Chang, 1970; Harshman, 1970; de Lathauwer et al., 2000; Kolda and Bader, 2009) or to approximate a given high-dimensional operator (Oseledets, 2011; Cichocki et al., 2016). Because the weights and features of deep neural networks are notoriously high-dimensional, tensorized approaches have gained traction. In Novikov et al. (2015), the authors combine efficient tensor storage and processing schemes with DNN training, resulting up to seven times fewer network weights. This work specifically used the tensor train style of weight storage, which is notable for compression of very high dimensional data, but does not have linear algebraic motivations in this context. Further studies followed, such as Chien and Bao (2018) that used multiway operations to extract features convolutionally. This work computes a Tucker factorization of convolutional features rather than treating tensors as operators. Similar layer contraction approaches, called tensor regression layers, have appeared in works such as in Cao et al. (2017) and Kossaifi et al. (2020). These approaches utilize low-rank Tucker-based factorizations to successfully reduce the number of weights in a network without sacrificing performance. These are more similar in spirit to pooling layers of convolutional neural networks rather than operations that preserve multilinearity. Many more studies have connected tensors and neural networks, and we recommend the survey (Wang et al., 2023) for a more complete history of the intersection of the two fields.

As we eluded to in the previous paragraph, in this work, we take a notably different perspective on tensors. We consider tensors as *multiway operators* and process our layers under this tensor operation. This provides a linear algebraic structure that enables us to extend desirable neural network structure to high dimensions with ease. Because of our strong algebraic foundation, we are able to express forward and backward propagation simply; in comparison, other tensor frameworks require heavy indexing notation. We share and achieve the same goal as other tensor approaches of reducing the number of network weights.

3 Background and preliminaries

To motivate our multidimensional neural network design, we start by introducing our notation and the tensor algebra in which we work. We use MATLAB indexing notation throughout the paper, such as selecting the j-th column of a matrix via $\mathbf{A}(:,j)$ or $\mathbf{A}_{:,j}$.

3.1 Tensor preliminaries

Let $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{m_1 \times m_2 \times n}$ be a real-valued, third-order tensor. Fixing the third-dimension, *frontal slices* $\mathbf{A}^{(k)} \in \mathbb{R}^{m_1 \times m_2}$ are matrices for $k = 1, \ldots, n$. Fixing the second-dimension, *lateral slices* $\vec{\boldsymbol{\mathcal{A}}}_j \in \mathbb{R}^{m_1 \times 1 \times n}$ are matrices oriented along the third dimension for $j = 1, \ldots, m_2$. Fixing the first and second dimensions, *tubes* $\mathbf{a}_{ij} \in \mathbb{R}^{1 \times 1 \times n}$ are vectors oriented along the third dimension for $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_2$. We depict these partitions in Figure 1. While this paper focuses on real-valued, third-order tensors (three indexes), we note all of the presented concepts generalize to higher-order and complex-valued tensors.

We interpret tensors as *t-linear operators* (Kilmer and Martin, 2011; Kernfeld et al., 2015). Through our operator lens, it is possible to define analogous matrix algebraic properties for tensors, such as orthogonality and rank. Thus, this framework has been described as *matrix-mimetic*. We describe the fundamental tools to understand how tensors operate for this paper, and refer the reader to Kilmer et al. (2013, 2021) and Kernfeld et al. (2015) for details about the underlying algebra.

We define a product to apply matrices along the third dimension of a tensor (i.e., along the tubes).

Definition 3.1 (mode-3 product). Given $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{m_1 \times m_2 \times n}$ and $\mathbf{M} \in \mathbb{R}^{\ell \times n}$, the mode-3 product, denoted $\widehat{\boldsymbol{\mathcal{A}}} \equiv \boldsymbol{\mathcal{A}} \times_3 \mathbf{M}$, outputs an $m_1 \times m_2 \times \ell$ tensor with entries

$$\widehat{\boldsymbol{\mathcal{A}}}(i_1, i_2, k) = \sum_{j=1}^{n} \boldsymbol{\mathcal{A}}(i_1, i_2, j)\mathbf{M}(k, j)$$

for $i_1 = 1, \ldots, m_1$, $i_2 = 1, \ldots, m_2$, and $k = 1, \ldots, \ell$.

The mode-3 product can be generalized along any mode; see Kolda and Bader (2009) for details.

Next, we define the facewise product to multiply the frontal slices of two third-order tensors in parallel.

Definition 3.2 (facewise product). Given $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{m_1 \times \ell \times n}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{\ell \times m_2 \times n}$, the facewise product, denoted $\boldsymbol{\mathcal{C}} \equiv \boldsymbol{\mathcal{A}} \triangle \boldsymbol{\mathcal{B}}$, returns an $m_1 \times m_2 \times n$ tensor where

$$\mathbf{C}^{(k)} = \mathbf{A}^{(k)}\mathbf{B}^{(k)}$$

for $k = 1, \ldots, n$.

Combining Definition 3.1 and Definition 3.2, we define our tensor operation, the $\star_M$-product, as follows:

Definition 3.3 ($\star_M$-product). Given $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{m_1 \times \ell \times n}$, $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{\ell \times m_2 \times n}$, and an invertible $n \times n$ matrix $\mathbf{M}$, the $\star_M$-product outputs an $m_1 \times m_2 \times n$ tensor of the following form:

$$\boldsymbol{\mathcal{A}} \star_M \boldsymbol{\mathcal{B}} = (\widehat{\boldsymbol{\mathcal{A}}} \triangle \widehat{\boldsymbol{\mathcal{B}}}) \times_3 \mathbf{M}^{-1}.$$

where $\widehat{\boldsymbol{\mathcal{X}}} \equiv \boldsymbol{\mathcal{X}} \times_3 \mathbf{M}$.

We say that $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ live in the spatial domain and $\widehat{\boldsymbol{\mathcal{A}}}$ and $\widehat{\boldsymbol{\mathcal{B}}}$ live in the transform domain. We perform the facewise product in the transform domain, then return to the spatial domain by applying $\mathbf{M}^{-1}$ along the tubes. If $\mathbf{M}$ is the identity matrix, the $\star_M$-product is exactly facewise product. If $\mathbf{M}$ were the discrete Fourier transformation matrix (DFT), we obtain the t-product (Kilmer and Martin, 2011). In this case, the frontal slices of $\widehat{\boldsymbol{\mathcal{A}}}$ correspond to different frequencies in the Fourier domain and are therefore decoupled.

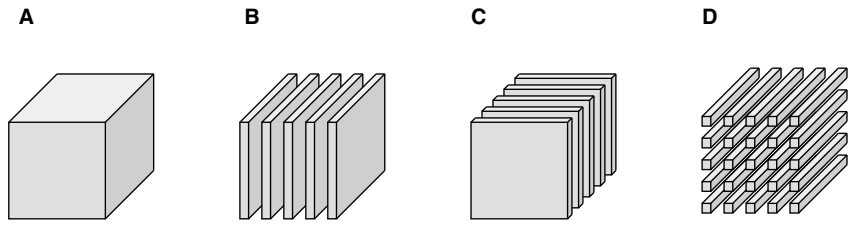


FIGURE 1
Tensor notation. **(A)** $\mathcal{A}$ is a third-order tensor, **(B)** $\vec{\mathcal{A}}_j$ are lateral slices, **(C)** $\mathbf{A}^{(k)}$ are frontal slices, and **(D)** $\mathbf{a}_{ij}$ are tubes.

We can interpret the $\star_M$-product as a block-structured matrix product via

$$\mathcal{A} \star_M \mathcal{B} \equiv \underbrace{(\mathbf{M}^{-1} \otimes \mathbf{I}_{m_1}) \begin{bmatrix} \widehat{\mathbf{A}}^{(1)} & & & \\ & \widehat{\mathbf{A}}^{(2)} & & \\ & & \ddots & \\ & & & \widehat{\mathbf{A}}^{(n)} \end{bmatrix} (\mathbf{M} \otimes \mathbf{I}_\ell)}_{\text{struct}(\mathcal{A})} \underbrace{\begin{bmatrix} \mathbf{B}^{(1)} \\ \mathbf{B}^{(2)} \\ \vdots \\ \mathbf{B}^{(n)} \end{bmatrix}}_{\text{unfold}(\mathcal{B})} \tag{1}$$

where $\otimes$ is the Kronecker product (Petersen and Pedersen, 2012). The block matrix structure, struct($\mathcal{A}$), depends on the choice of transformation, $\mathbf{M}$. We consider two familiar examples: the facewise product ($\mathbf{M} = \mathbf{I}_n$) and the t-product ($\mathbf{M} = \mathbf{F}_n$, the DFT matrix):

$$\mathbf{M} = \mathbf{I}_n \quad \text{struct}(\mathcal{A}) = \text{bdiag}(\mathcal{A}) = \begin{bmatrix} \mathbf{A}^{(1)} & & & \\ & \mathbf{A}^{(2)} & & \\ & & \ddots & \\ & & & \mathbf{A}^{(n)} \end{bmatrix} \tag{2a}$$

$$\mathbf{M} = \mathbf{F}_n \quad \text{struct}(\mathcal{A}) = \text{bcirc}(\mathcal{A}) = \begin{bmatrix} \mathbf{A}^{(1)} & \mathbf{A}^{(n)} & \cdots & \mathbf{A}^{(2)} \\ \mathbf{A}^{(2)} & \mathbf{A}^{(1)} & \cdots & \mathbf{A}^{(3)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{(n)} & \mathbf{A}^{(n-1)} & \cdots & \mathbf{A}^{(1)} \end{bmatrix}. \tag{2b}$$

While we never explicitly form Equation (2), the block structure will be helpful for subsequent analysis.

3.2 Matrix-mimetic tensor algebra

The $\star_M$-product yields a well-defined algebraic structure. Specifically, suppose we have tubes $\mathbf{a}, \mathbf{b} \in \mathbb{R}^{1\times 1\times n}$. Then,

$$\mathbf{a} \star_M \mathbf{b} \equiv \text{vec}(\mathbf{a})^\top \underbrace{\mathbf{M}^\top \text{diag}(\mathbf{M}\,\text{vec}(\mathbf{b}))\mathbf{M}^{-\top}}_{\mathbf{R}[\mathbf{b}]} \tag{3}$$

where $\text{vec} : \mathbb{R}^{1\times 1\times n} \to \mathbb{R}^{n\times 1}$ turns a tube into a column vector. The $\star_M$-product of tubes (3) is equivalent to post-multiplying by the structured matrix, $\mathbf{R}[\mathbf{b}]$. Note that $\mathbf{R}[\cdot]$ implicitly depends on the choice of $\mathbf{M}$, but we omit explicitly writing this dependence for notational simplicity. The tubes form a matrix subalgebra which dictates the algebraic structure imposed on the high-dimensional space (Kernfeld et al., 2015). As a result, the $\star_M$-product is *matrix-mimetic* and yields several familiar concepts.

Definition 3.4 ($\star_M$-identity tube). The identity tube $\mathbf{e} \in \mathbb{R}^{1\times 1\times n}$ under the $\star_M$-product is

$$\mathbf{e} = \mathbf{1} \times \mathbf{M}^{-1},$$

where $\mathbf{1}$ is the $1 \times 1 \times n$ tube containing all ones.

This gives rise to the notion of an identity tensor.

Definition 3.5 ($\star_M$-identity tensor). A tensor $\mathcal{I} \in \mathbb{R}^{m\times m\times n}$ is the identity tensor if $\mathcal{I}(i, i, :) = \mathbf{e}$ for $i = 1, \ldots, m$ where $\mathbf{e}$ is the $\star_M$-identity tube.

Note that if the size of third dimension is equal to one (i.e., $n = 1$), then Definition 3.5 collapses into the identity matrix. This is a hallmark of our $\star_M$-framework and matrix-mimeticity; the product and definitions reduce to the equivalent matrix definitions when the third dimension is removed.

Definition 3.6 ($\star_M$-transpose). Given $\mathcal{A} \in \mathbb{R}^{m_1\times m_2\times n}$, its transpose $\mathcal{B} \equiv \mathcal{A}^\top \in \mathbb{R}^{m_2\times m_1\times n}$ is

$$\widehat{\mathbf{B}}^{(k)} = (\widehat{\mathbf{A}}^{(k)})^H$$

for $k = 1, \ldots, n$.

Note that if our transformation $\mathbf{M}$ is complex-valued, the transpose operator in the transform domain performs the conjugate transpose. However, because we are working with real-valued tensors in the spatial domain, the transpose will be real-valued as well.

4 Tensor neural networks (t-NNs)

In general, neural networks are parameterized mappings from an input space $\mathcal{Y}$ to the target space $\mathcal{C}$. These mappings are composite functions of the form

$$F_{\text{NN}}(\cdot, \boldsymbol{\theta}) \equiv f_d(\cdots f_2(f_1(\cdot, \boldsymbol{\theta}_1), \boldsymbol{\theta}_2)\cdots), \boldsymbol{\theta}_d). \tag{4}$$

Each subfunction $f_j(\cdot, \boldsymbol{\theta}_j)$ for $j = 1, \ldots, d$ is called a *layer*. The goal is to find a good set of weights $\boldsymbol{\theta} \equiv (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_d) \in \Theta$ such that

$F_{\text{NN}}(\mathbf{y}, \boldsymbol{\theta}) \approx c$ for all input-target pairs $(\mathbf{y}, c) \subset \mathcal{D}$. Here, Θ is the parameter space and $\mathcal{D} \subset \mathcal{Y} \times \mathcal{C}$ is the data space.

The most common layer of feed forward neural networks (4) consists of an affine transformation and pointwise nonlinearity of the form

$$\mathbf{y}_j = f_j(\mathbf{y}_{j-1}, \boldsymbol{\theta}_j) = \sigma_j(\mathbf{W}_j \mathbf{y}_{j-1} + \mathbf{b}_j) \tag{5}$$

where $\mathbf{W}_j \in \mathbb{R}^{m_j \times m_{j-1}}$ is a weight matrix, $\mathbf{b}_j \in \mathbb{R}^{m_j}$ is a bias vector, and $\sigma_j : \mathbb{R} \to \mathbb{R}$ is a one-dimensional nonlinear activation function, applied entrywise. In practice, activation functions are monotonic, such as the sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$, or Rectified Linear Unit (ReLU), $\sigma(x) = \max(x, 0)$. We call $\mathbf{y}_j$ the *features* of layer j and $\mathbf{y}_0 \in \mathcal{Y}$ are the input features. Notationally, we use $\boldsymbol{\theta}_j \equiv (\mathbf{W}_j, \mathbf{b}_j)$ to collect all of the learnable weights for layer j.

4.1 Improved parameterization with the $\star_M$-product

When designing a neural network, we seek to balance a simple parameter space with an expressive feature space. However, traditionally fully-connected layers like (5) use parameters in a highly inefficient manner. We propose new tensor fully-connected layers for a more efficient parametrization while still creating a rich feature space. Specifically, we consider the following tensor forward propagation scheme:

$$\vec{\mathcal{Y}}_j = \sigma_j(\mathcal{W}_j \star_M \vec{\mathcal{Y}}_{j-1} + \vec{\mathcal{B}}_j), \tag{6}$$

for $j = 1, \ldots, d$. Suppose our input features $\vec{\mathcal{Y}}_0 = \vec{\mathcal{Y}} \in \mathcal{Y}$ is of size $m_0 \times 1 \times n$. Here, the weight tensor $\mathcal{W}_j$ is of size $m_{j+1} \times m_j \times n$ and the bias $\vec{\mathcal{B}}_j$ is of size $m_{j+1} \times 1 \times n$. The forward propagation through Equation (6) results in a tensor neural network $F_{\text{tnn}}(\cdot, \boldsymbol{\theta})$ where $\boldsymbol{\theta} \equiv (\mathcal{W}_j, \vec{\mathcal{B}}_j)_{j=1}^d$.

Through the illustration in Figure 2, we depict the number of weight parameters required to preserve the size of our feature space using either a dense matrix (Figure 2A) or a dense tensor under the $\star_M$-product (Figure 2B). Using the $\star_M$-algebra, we can reduce the number of weight parameters by a factor of n while maintaining the same number of features (i.e., maintaining a rich feature space). Beyond the parametric advantages, the multilinear $\star_M$-product incorporates the structure of the data into the features. This enables our t-NNs to extract more meaningful features, and hence improve the richness of our feature space.

4.2 The training problem

Training a (tensor) neural network is posed as a stochastic optimization problem given by

$$\min_{\boldsymbol{\theta} \in \Theta} \mathbb{E}\, L(F_{\text{tnn}}(\vec{\mathcal{Y}}, \boldsymbol{\theta}), c) + \lambda R(\boldsymbol{\theta}), \tag{7}$$

where $L : \mathbb{R}^{m_{d+1} \times 1 \times n} \times \mathcal{C} \to \mathbb{R}$ is the loss function that measures the misfit between the network prediction and the true target. The expectation is taken over all input-target pairs $(\vec{\mathcal{Y}}, c) \in \mathcal{D}$. The additional function $R : \Theta \to \mathbb{R}$ regularizes the weights to promote desirable properties (e.g., smoothness), weighted by a regularization parameter $\lambda > 0$.

4.3 Tubal loss (t-loss) functions

The loss function is chosen based on the given task. For regression, we often use mean squared error, and for classification, which is the focus of this paper, we often use cross entropy. Cross entropy loss, related to the Kullback–Leibler (KL) divergence (Kullback and Leibler, 1951), measures the distance between two probability distributions. In practice, we first transform the network outputs into a set of probabilities using exponential normalization. Specifically, we use the softmax function $h : \mathbb{R}^p \to \Delta^p$, defined entrywise as

$$[h(\mathbf{x})]_i = \frac{e^{x_i}}{\sum_{j=1}^p e^{x_j}} \qquad \text{for } i = 1, \ldots, p, \tag{8}$$

where Δ^p is the p-dimensional unit simplex.

To preserve the algebraic integrity of t-NNs, we introduce a tubal variant of the softmax function. Drawing inspiration from Lund (2020), a tubal function, we start by defining tubal functions generally.

Definition 4.1 (tubal function). Given $\mathbf{b} \in \mathbb{R}^{1 \times 1 \times n}$, a *tubal function* $f : \mathbb{R}^{1 \times 1 \times n} \to \mathbb{R}^{1 \times 1 \times n}$ acts on the action of $\mathbf{b}$ under the $\star_M$-product; that is,

$$\underset{\text{tubal function}}{f(\mathbf{b})} \equiv \underset{\text{matrix function}}{f(\mathbf{R}[\mathbf{b}])} \equiv \mathbf{M}^\top \underset{\text{pointwise function}}{f(\mathbf{M}\,\text{vec}(\mathbf{b}))}\, \mathbf{M}^{-\top} \tag{9}$$

In practice, a tubal function is applied pointwise in the transform domain.

We note that the pointwise function in Definition 4.1 is equivalent to applying a matrix function to the eigenvalues of the matrix $\mathbf{R}[\mathbf{b}]$. We provide a visualization of the effects of tubal functions compared to applying an entry-wise function in Example 4.2.

Example 4.2 (Visualizations of tubal functions). Consider the following RGB image $\mathcal{B} \in \mathbb{R}^{150 \times 169 \times 3}$ of a Tufts Community Apeal elephant (TCA, 2023), where 3 is the number of color channels (Figure 3A). We rescale each entry of $\mathcal{B}$ between 0 to 1 and consider the following transformation matrix:

$$\mathbf{M} = \begin{pmatrix} -1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{M}^{-1} = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}. \tag{10}$$

To illustrate the effect of using tubal functions, we compare applying various functions as pointwise functions (i.e., independent of $\mathbf{M}$) and as tubal functions using Equation (9) in Figure 3B.

Tubal functions are able to capture shared patterns among the tubes that entrywise operators ignore. Each choice of tubal function highlights different features of the image, such as the body with $f(x) = \max(x, 0)$ and the heart with $f(x) = \tanh(x)$. In comparison, the entrywise counterparts do not bring new insights or structure. A striking difference occurs for $f(x) = \text{sign}(x)$. The

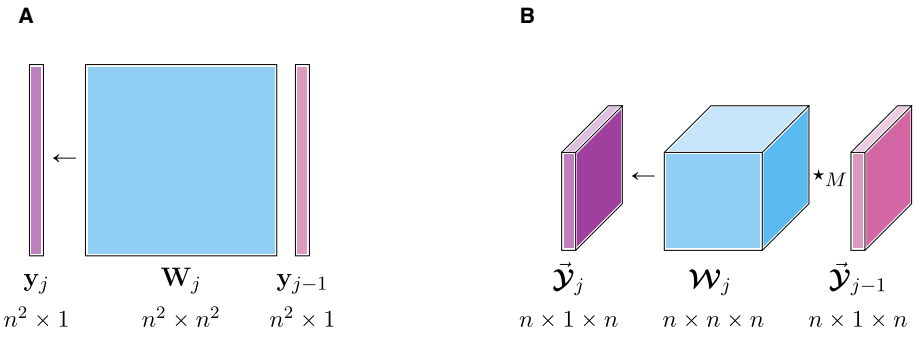


FIGURE 2
Comparison of network parameterizations to preserve the number of features of layer $j-1$ and layer j. The matrix mapping requires n^4 weights in $\mathbf{W}_j$ and tensor mapping requires n^3 weights in $\mathcal{W}_j$. **(A)** Matrix linear mapping. **(B)** Tensor $\star_M$-mapping.

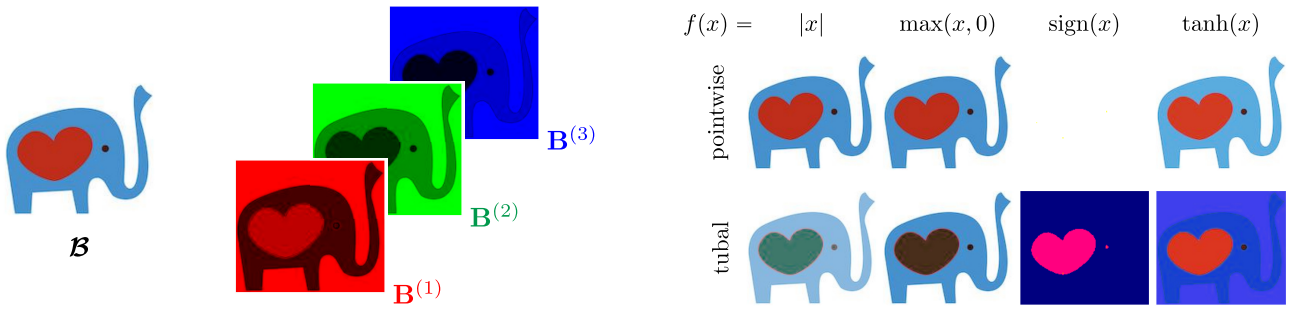


FIGURE 3
Comparison of entry-wise and tubal functions applied to roughly piecewise constant image. **(A)** Image and color channels of elephant image. The elephant is almost piecewise constant with easily distinguishable RGB values. **(B)** Effects of tubal functions under the transformation (10). Images are rescaled between 0 and 1 after applying the respective function.

entrywise operator turns all pixels white because the RGB entries are all nonnegative. In contrast, the tubal equivalent is applied in the transform domain, and hence reveals meaningful features of the image.

Note that the results depend on the chosen tubal function and transform $\mathbf{M}$. We deliberately chose $\mathbf{M}$ to emphasize green and blue channels and diminish the effect of the red channel, enabling easy-to-interpret distinctions between tubal and entrywise functions.

We can now define a tubal softmax function based on the traditional softmax function in (8).

Definition 4.3 (tubal softmax). Consider a lateral slice $\vec{\mathcal{X}} \in \mathbb{R}^{p\times 1\times n}$ as a $p \times 1$ vector of $1 \times 1 \times n$ tubes. Then, the tubal softmax function $h : \mathbb{R}^{p\times 1\times n} \to \Delta^{p\times 1\times n}$ performs the following mapping:

$$[h(\vec{\mathcal{X}})]_i = \left(\sum_{j=1}^{p} \exp(\mathbf{x}_j)\right)^{-1} \star_M \exp(\mathbf{x}_i)$$

for $i = 1, \ldots, p$ where $h(\vec{\mathcal{X}}) \in \mathbb{R}^{p\times 1\times n}$ and the exponential functions are applied as tubal functions. Here, $\Delta^{p\times 1\times n}$ is the tubal-equivalent of p-dimensional unit simplex.

We interpret $h(\vec{\mathcal{X}})$ as a vector of "tubal probabilities" in that the tubes sum to the identity tube

$$\begin{aligned}\textstyle\sum_{i=1}^{p}[h(\vec{\mathcal{X}})]_i &= \textstyle\sum_{i=1}^{p}\left[\left(\sum_{j=1}^{p}\exp(\mathbf{x}_j)\right)^{-1} \star_M \exp(\mathbf{x}_i)\right] \\ &= \left(\textstyle\sum_{j=1}^{p}\exp(\mathbf{x}_j)\right)^{-1} \star_M \left[\textstyle\sum_{i=1}^{p}\exp(\mathbf{x}_i)\right] \\ &= \mathbf{e}.\end{aligned}$$

Through Definition 4.3, we demonstrate the parallels between tubal functions and traditional functions; the similarities are a direct consequence of a matrix-mimetic tensor framework.

The last step to define the tubal cross entropy function that converts the output of tubal function to a scalar. Recall, traditional cross entropy $L_{\text{ce}} : \mathbb{R}^p \times \{1, \ldots, p\} \to \mathbb{R}_+$ for one sample is given by

$$L_{\text{ce}}(\mathbf{x}, c) = -\log[h(\mathbf{x})]_c \tag{11}$$

where $\mathbf{x}$ is the output of the network, c is the corresponding target class, and h is the softmax function. We generalize (11) to a tubal variant as follows:

Definition 4.4 (tubal cross entropy (t-cross entropy)). The *tubal cross entropy function* $L_{\text{tce}} : \Delta^{p\times 1\times n} \times \{1, \ldots, p\} \to \mathbb{R}_+$ is given by

$$L_{\text{tce}}(\vec{\mathcal{X}}, c) = -\| \text{vec}(\log[h(\vec{\mathcal{X}})]_{c,1,:})\|_q$$

where h is the tubal softmax function, log is applied as a tubal function, c is the index corresponding to the target class, and $\|\cdot\|_q$ is a vector norm.

The intuition behind Definition 4.4 is the following. If we have good features $\vec{\mathcal{X}}$, then $[h(\vec{\mathcal{X}})]_{c,1,:} \approx \mathbf{e}$, the identity tube, and the remaining tubes will be closer to $\mathbf{0}$. In the transform domain, $[h(\vec{\mathcal{X}}) \times \mathbf{M}]_{c,1,:} \approx \mathbf{1}$ and the remaining entries will be close to zero. When we apply the log pointwise in the transform domain, the tube $\log[h(\vec{\mathcal{X}}) \times \mathbf{M}]_{c,1,:} \approx \mathbf{0}$ and the remaining entries will be large negative numbers. As a result, L_{tce} is smallest when the $[h(\vec{\mathcal{X}})]_{c,1,:} \approx \mathbf{e}$, as desired.

In practice, if q-norm corresponds to a finite integer, we can instead use $\|\log[h(\vec{\mathcal{X}})]_{c,1,:}\|_q^q$ for t-cross entropy for easier derivative computations. For numerical benefits when training, we consider normalized versions based on the number of tubal entries, e.g., multiply by $1/n$. These suggested modifications should not change performance in theory, but could change preferred training hyperparameters.

4.4 Backward propagation with t-NNs

The workhorse of neural network training is backward propagation (Rumelhart et al., 1986; Bengio et al., 1994; Shalev-Shwartz et al., 2017; Nielsen, 2018), a method to calculate the gradient of the objective function (7) with respect to the weights. With gradient information, one can apply standard stochastic gradient optimization techniques to train.

In the $\star_M$-framework, for an orthogonal transformation $\mathbf{M}$, the backpropagation formulas are analogous to the matrix case. For example, the derivatives of the $\star_M$-product are

$$\frac{\partial}{\partial \vec{\mathcal{Y}}}[\mathcal{W} \star_M \vec{\mathcal{Y}}] = \mathcal{W}^\top \star_M \partial\vec{\mathcal{Y}}$$
$$\text{and} \qquad \frac{\partial}{\partial \mathcal{W}}[\mathcal{W} \star_M \vec{\mathcal{Y}}] = \partial\mathcal{W} \star_M \vec{\mathcal{Y}}^\top \tag{12}$$

where $\partial\mathcal{X}$ indicates a direction or perturbation of the same size as $\mathcal{X}$. For full details on the derivation, we refer the reader to Newman (2019).

The simplicity of the back-propagation formulas (12) is one of the hallmarks of our choice of leveraging a matrix-mimetic tensor framework. Other tensor-based neural network designs (Wang et al., 2023) often require complicated indexing and non-traditional notation which, in addition to being cumbersome, can preclude extending more sophisticated neural network architectures to higher dimensions. The $\star_M$-framework yields derivative formulations that are easy to interpret, implement, and analyze.

5 Stable t-NNs

As the depth (number of layers) of a network increases, gradient-based training is subject to numerical instability such as vanishing or exploding gradient problem (Bengio et al., 1994; Shalev-Shwartz et al., 2017). To avoid these instabilities, one can interpret deep neural networks as discretizations of differential equations (Ee, 2017; Haber and Ruthotto, 2017; Haber et al., 2018) and analyze the stability of forward propagation as well as the well-posedness of the learning problem; i.e., whether the classifying function depend continuously on the initialization of the parameters (Ascher, 2010). By ensuring stability and well-posedness, networks can generalize better to similar data and can classify data more robustly.

We emphasize that the notion of stability is related to the formal numerical analysis definition in the Lyapunov sense (i.e., stability of dynamical systems). This is a property of the model itself and independent of the data. From a statistical and foundational learning theory perspective, neural networks are typically over-parameterized models, which tend to overfit. In this context, stability can promote better generalization by imposing constraints of the structure of the weight matrices, effectively reducing the number of degrees of freedom. The tensorial structure imposes additional constraints and further reduces the number of parameters, which can again lead to better generalization.

5.1 Well-posed learning problem criterion

Consider the residual neural network (He et al., 2016) with tensor operations, given by

$$\vec{\mathcal{Y}}_j = \vec{\mathcal{Y}}_{j-1} + h\sigma(\mathcal{W}_j \star_M \vec{\mathcal{Y}}_{j-1} + \vec{\mathcal{B}}_j) \qquad \text{for } j = 1, \ldots, d, \tag{13}$$

where $\vec{\mathcal{Y}}_j \in \mathbb{R}^{m\times 1\times n}$, $\mathcal{W}_j \in \mathbb{R}^{m\times m\times n}$, and $\vec{\mathcal{B}}_j \in \mathbb{R}^{m\times 1\times n}$. With the addition of the step size h, we can interpret Equation (13) as a forward Euler discretization of the continuous ordinary differential equation (ODE)

$$\frac{d\vec{\mathcal{Y}}(t)}{dt} = \sigma(\mathcal{W}(t) \star_M \vec{\mathcal{Y}}(t) + \vec{\mathcal{B}}(t)) \qquad \text{with} \qquad \vec{\mathcal{Y}}(0) = \vec{\mathcal{Y}}_0 \tag{14}$$

for all $t \in [0, T]$ where T is the final time corresponding to the depth of the discretized network.

The stability of non-autonomous ODEs like (14) depends on the eigenvalues of the Jacobian with repsect to the features. To perform analogous analysis for tensor operators, it is useful to consider the equivalent block matrix version of the $\star_M$-product in Equation (1) where $\mathcal{W} \star_M \vec{\mathcal{Y}} \equiv \text{struct}(\mathcal{W})\,\text{unfold}(\vec{\mathcal{Y}})$. It follows that we can matricize (14) via

$$\frac{d}{dt}\text{unfold}(\vec{\mathcal{Y}}(t)) = \sigma(\text{struct}(\mathcal{W}(t))\,\text{unfold}(\vec{\mathcal{Y}}(t)) + \text{unfold}(\vec{\mathcal{B}}(t))). \tag{15}$$

The Jacobian of the matricized system (15) with respect to $\text{unfold}(\vec{\mathcal{Y}}(t))$ is

$$\mathbf{J}(t) = \text{diag}(\sigma'(\mathbf{x}(t)))\,\text{struct}(\mathcal{W}(t))$$

where $\mathbf{x}(t) = \text{struct}(\mathcal{W}(t))\,\text{unfold}(\vec{\mathcal{Y}}(t)) + \text{unfold}(\vec{\mathcal{B}}(t))$ and $\mathbf{J}(t) \in \mathbb{R}^{mn\times mn}$. In most cases, the activation function σ is non-decreasing, and thus the entries in $\text{diag}(\sigma'(\mathbf{x}(t)))$ are nonnegative. As a result, the stability and well-posedness of the ODE relies on the eigenvalues of $\text{struct}(\mathcal{W}(t))$. As described in Haber and Ruthotto (2017), the learning problem is well-posed if

$$\text{Re}(\lambda_i(\text{struct}(\mathcal{W}(t)))) \approx 0 \qquad \text{for } i = 1, \ldots, mn, \tag{16}$$

where $\lambda_i(\mathbf{A})$ is the i-th eigenvalue of $\mathbf{A}$. This criterion implies that the imaginary part of the eigenvalues drive the dynamics, promoting rotational movement of features. This produces stable forward propagation that avoids features diverging and prevents inputs from distinct classes from converging to indistinguishable points; the latter would lead to ill-posed back propagation and hence an ill-posed learning problem.

We can be more concrete about the eigenvalues because in Equation (1), $\text{struct}(\boldsymbol{\mathcal{W}}(t))$ is block-diagonalized in the transform domain. Thus, we can equivalently write Equation (16) as follows:

$$\text{Re}(\lambda_i(\widehat{\mathbf{W}}^{(k)}(t))) \approx 0 \tag{17}$$

for $i = 1, \ldots, m$ and $k = 1, \ldots, n$. In short, if the eigenvalues of each frontal slice in the transform domain have a real part close to zero, the t-NN learning problem will be well-posed, save one more requirement.

A subtle, yet important requirement for stability is that the weights change gradually over time (i.e., layers). This ensures that small perturbations of the weights yield small perturbations of the features. To promote this desired behavior, we imposed a smoothing regularizer in (discrete) time via

$$R_{\text{smooth}}(\{\boldsymbol{\mathcal{W}}_j, \vec{\boldsymbol{\mathcal{B}}}_j\}_{j=1}^{d}) = \sum_{j=1}^{d-1} \|\boldsymbol{\mathcal{W}}_{j+1} - \boldsymbol{\mathcal{W}}_j\|_F^2 + \|\vec{\boldsymbol{\mathcal{B}}}_{j+1} - \vec{\boldsymbol{\mathcal{B}}}_j\|_F^2. \tag{18}$$

5.2 Hamiltonian-inspired stable t-NNs

While theoretically useful, it is impractical to evaluate the eigenvalues of the weight tensors to satisfy the well-posedness condition (17) as we train a network. Instead, motivated by the presentation in Haber and Ruthotto (2017), we implement a forward propagation that inherently satisfies (17) independent of the weights. This forward propagation scheme is inspired by Hamiltonian dynamics, which we briefly describe and refer to Ascher (2010) and Brooks et al. (2011) for further details. We define a Hamiltonian as follows:

Definition 5.1 (Hamiltonian). Let $\mathbf{y}(t) \in \mathbb{R}^{m_y \times 1}$, $\mathbf{z}(t) \in \mathbb{R}^{m_z \times 1}$, and $t \in [0, T]$. A Hamiltonian $H : \mathbb{R}^{m_y \times 1} \times \mathbb{R}^{m_z \times 1} \times [0, T] \to \mathbb{R}$ is a system governed by the following dynamics:

$$\frac{d\mathbf{y}}{dt} = \nabla_{\mathbf{z}} H(\mathbf{y}(t), \mathbf{z}(t), t) \qquad \text{and} \qquad \frac{d\mathbf{z}}{dt} = -\nabla_{\mathbf{y}} H(\mathbf{y}(t), \mathbf{z}(t), t).$$

Intuitively, the Hamiltonian H describes the total energy of the system with $\mathbf{y}$ as the position and $\mathbf{z}$ as the momentum or velocity. We can separate total energy into potential energy U and kinetic energy T; that is, $H(\mathbf{y}, \mathbf{z}, t) = U(\mathbf{y}) + T(\mathbf{z})$. This separability ensures the Hamiltonian dynamics conserve energy; i.e.,

$$\frac{dH}{dt} = \frac{d\mathbf{y}}{dt} \nabla_{\mathbf{y}} H + \frac{d\mathbf{z}}{dt} \nabla_{\mathbf{z}} H = 0 \tag{19}$$

In terms of neural networks, energy conservation (19) ensures that network features are preserved during forward propagation, thereby avoiding the issue of exploding/vanishing gradients and enabling the use of deeper networks. Additionally, Hamiltonians are symplectic or volume-preserving in the sense that the dynamics are divergence-free. For neural networks, this ensures the distance between features does not change significantly, avoiding converging and diverging behaviors. We also note that Hamiltonians are time-reversible. This ensures that if we have well-posed dynamics during forward propagation, we will have similar dynamics for backward propagation.

5.3 Discretizing Hamiltonians with leapfrog integration

To preserve the benefits of Hamiltonians in the discretized setting, we symmetrize the Hamiltonian (Definition 5.1) and use a leapfrog integration method (Skeel, 1993; Ascher, 2010; Haber and Ruthotto, 2017). For t-NNs, we write the new system in terms of the $\star_M$-product (with slight abuse of notation)

$$\frac{d}{dt}\begin{bmatrix} \vec{\boldsymbol{\mathcal{Y}}}(t) \\ \vec{\boldsymbol{\mathcal{Z}}}(t) \end{bmatrix} = \sigma\left(\begin{bmatrix} \mathbf{0} & \boldsymbol{\mathcal{W}}(t) \\ -\boldsymbol{\mathcal{W}}(t)^\top & \mathbf{0} \end{bmatrix} \star_M \begin{bmatrix} \vec{\boldsymbol{\mathcal{Y}}}(t) \\ \vec{\boldsymbol{\mathcal{Z}}}(t) \end{bmatrix} + \mathbf{b}(t)\right) \tag{20}$$

where $\vec{\boldsymbol{\mathcal{Y}}}(0) = \vec{\boldsymbol{\mathcal{Y}}}_0$ and $\vec{\boldsymbol{\mathcal{Z}}}(0) = \mathbf{0}$. Here, $\vec{\boldsymbol{\mathcal{Y}}}(t)$ indicates the data features and $\vec{\boldsymbol{\mathcal{Z}}}(t)$ is an auxilary variable not related to the data directly. We add the bias tube, $\mathbf{b}(t)$, to each element. The equivalent matricized version of Equation (20) is

$$\frac{d}{dt}\begin{bmatrix} \text{unfold}(\vec{\boldsymbol{\mathcal{Y}}}(t)) \\ \text{unfold}(\vec{\boldsymbol{\mathcal{Z}}}(t)) \end{bmatrix} = \sigma\left(\begin{bmatrix} \mathbf{0} & \text{struct}(\boldsymbol{\mathcal{W}}(t)) \\ -\text{struct}(\boldsymbol{\mathcal{W}}(t))^\top & \mathbf{0} \end{bmatrix} \begin{bmatrix} \text{unfold}(\vec{\boldsymbol{\mathcal{Y}}}(t)) \\ \text{unfold}(\vec{\boldsymbol{\mathcal{Z}}}(t)) \end{bmatrix} + \mathbf{b}(t)\right). \tag{21}$$

The (matricized) system (21) is inherently stable, independent of the weight tensors $\boldsymbol{\mathcal{W}}(t)$, because of the block antisymmetric structure. The eigenvalues of antisymmetric matrices are purely imaginary, which exactly satisfy the stability condition in Equation (17).

We discretize Equation (20) using the leapfrog method, a symplectic integration technique, defined as

$$\vec{\boldsymbol{\mathcal{Z}}}_{j+\frac{1}{2}} = \vec{\boldsymbol{\mathcal{Z}}}_{j-\frac{1}{2}} - h\sigma(\boldsymbol{\mathcal{W}}_{j+1}^\top \star_M \vec{\boldsymbol{\mathcal{Y}}}_j + \mathbf{b}_{j+1}) \tag{22a}$$

$$\vec{\boldsymbol{\mathcal{Y}}}_{j+1} = \vec{\boldsymbol{\mathcal{Y}}}_j + h\sigma(\boldsymbol{\mathcal{W}}_{j+1} \star_M \vec{\boldsymbol{\mathcal{Z}}}_{j+\frac{1}{2}} + \mathbf{b}_{j+1}) \tag{22b}$$

for $j = 0, \ldots, d-1$. We demonstrate the benefits of stable forward propagation (22) in Example 5.2.

Example 5.2 (Trajectories of stable t-NNs). We construct a dataset in $\mathbb{R}^3$ randomly drawn from a multivariate normal distribution with mean of $\mathbf{0}$ and a covariance matrix of $3\mathbf{I}_3$. The points are divided into three classes based on distance to the origin with yellow points inside a sphere with radius $r = 3.5$, green points inside a sphere with radius $R = 5.5$, and purple points outside both spheres. We train with 1200 data points and store the data as $1 \times 1 \times 3$ tubes.

We forward propagate using one of two integrators with weights and biases as $1 \times 1 \times 3$ tubes:

$$\text{Forward Euler} \quad \mathbf{y}_{j+1} = \mathbf{y}_j + h\sigma(\mathbf{w}_{j+1} \star_M \mathbf{y}_j + \mathbf{b}_{j+1}) \tag{23a}$$

$$\text{Leapfrog} \quad \begin{cases} \mathbf{z}_{j+\frac{1}{2}} = \mathbf{z}_{j-\frac{1}{2}} - h\sigma(\mathbf{w}_{j+1}^\top \star_M \mathbf{y}_j + \mathbf{b}_{j+1}) \\ \mathbf{y}_{j+1} = \mathbf{y}_j + h\sigma(\mathbf{w}_{j+1} \star_M \mathbf{z}_{j+\frac{1}{2}} + \mathbf{b}_{j+1}) \end{cases} \tag{23b}$$

We create a network with $d = 32$ layers and use a step size of $h = 2$. We use the discrete cosine transform for $\mathbf{M}$. We train for 50 epochs using Adam (Kingma and Ba, 2015) with a batch size of 10 and a learning rate of $\gamma = 10^{-2}$. To create smoother dynamics, we regularize the weights using Equation (18) with regularization parameter $\lambda = 10^{-4}/h$. We illustrate the dynamics of trained networks in Figure 4.

The dynamics in Figure 4 show the topological benefits of leapfrog integration. The data points exhibit smoother, rotational dynamics to reach the desired linearly-separable final configuration. In comparison, forward Euler propagation significantly changes topology during forward propagation. Such topological changes may yield ill-posed learning problems and poor network generalization.

6 Numerical results

We present two image classification problems to compare tensor linear layers and stable tensor neural networks to comparable matrix versions. Overall, the results show that t-NN trained with tubal loss functions generalize better to unseen data than equivalent matrix networks, and can do so with 20–30 times fewer network weights.

6.1 Experiment setup and hardware

We implement both tensor and matrix frameworks using PyTorch (RRID:SCR_018536) (Paszke et al., 2017). All of the code to reproduce the experiments is provided in https://github.com/elizabethnewman/tnn. All experiments were run on an Exxact server with four RTX A6000 GPUs, each with 48 GB of RAM. Only one GPU was used to generate each result. The results we report are for the networks that yielded the best accuracy on the validation data.

We train all models using the stochastic gradient method Adam (Kingma and Ba, 2015), which uses a default learning rate of 10^{-3}. In most cases, we select hyperparameters and weight initialization based on the default settings in PyTorch. We indicate the few exceptions to this at the start of the corresponding sections. We also pair the stochastic optimizers with a learning rate scheduler that decreases the learning rate by a factor of γ every M steps. In all cases, we used the default $\gamma = 0.9$ and $M = 100$. This is a common practice to ensure convergence of stochastic optimizer in idealized settings (Bottou et al., 2018). We utilize a weight decay parameter in some experiments to reduce overfitting.

For the tensor networks in all experiments, we use the discrete cosine transform for $\mathbf{M}$, which is a close, real-valued version of the discrete Fourier transform (DFT). The DFT matrix corresponds to the t-product, which has been shown to be effective for natural image applications (Kilmer and Martin, 2011; Hao et al., 2013; Newman et al., 2018).

6.2 MNIST dimensionality reduction

The MNIST dataset (LeCun et al., 2010) is composed of 28×28 grayscale images of handwritten digits. We train on 50,000 images and reserve 10,000 for validation. We report test accuracy on 10,000 images not used for training nor validation. For NNs, we vectorize images and store as columns, resulting in a matrix of size $28^2 \times b$ where b is the number of images. For t-NNs, we store the images as lateral slices, resulting in a tensor of size $28 \times b \times 28$.

In this experiment, we train an autoencoder to efficiently represent the high-dimensional MNIST data in a low-dimensional subspace. Autoencoders can be thought of as nonlinear, parameterized extensions of the (truncated) singular value decomposition. Our goal is to solve the (unsrpervised) learning problem

$$\min_{\boldsymbol{\theta}_{\text{enc}}, \boldsymbol{\theta}_{\text{dec}}} \mathbb{E}_{\mathbf{y} \sim \mathcal{Y}} \, \|f_{\text{dec}}(f_{\text{enc}}(\mathbf{y}, \boldsymbol{\theta}_{\text{enc}}), \boldsymbol{\theta}_{\text{dec}}) - \mathbf{y}\|_2^2 \tag{24}$$

where $f_{\text{enc}} : \mathcal{Y} \to \mathcal{Z}$ is the *encoder* and $f_{\text{dec}} : \mathcal{Z} \to \mathcal{Y}$ is the *decoder*. Here, $\mathcal{Z}$ is the latent space that is smaller than the data space; in terms of dimension, we say $\dim(\mathcal{Z}) < \dim(\mathcal{Y})$. Note that the mean squared error (MSE) tubal loss is the same as the MSE loss function when using an orthogonal transformation matrix, as we have done in our experiments. We describe the NN and t-NN autoencoder architectures used in Figure 5 and report results in Figure 6.

We observe that the t-NN outperforms the NN autoencoders with similar numbers of weights with an order of magnitude smaller training and validation loss and test error as well as qualitative improvements of the approximations. The neural network autoencoder with the same feature space dimensions, NN(560,280), performs best in terms of the loss and error metrics, but requires over 20 times more network weights than the t-NN autoencoder. The t-NN layers are able capture spatial correlations more effectively using multilinear operations, resulting in quality approximations with significantly fewer weights.

6.3 MNIST classification

We use the same MNIST dataset as for the autoencoder example. We train for 20 epochs using Adam with a batch size of 32 and a learning rate of 10^{-2}. We add Tikhonov regularization (weight decay) with a regularization parameter of $\lambda = 10^{-4}$. We use the PyTorch defaults for the other optimizer hyperparameters. For the t-cross entropy loss, we use a squared ℓ_2-norm and normalize by the number of entries in the tube.

We compare four different two-layer neural network architectures, described in Table 1. We use either cross entropy loss or t-cross entropy loss, depending on the architecture.

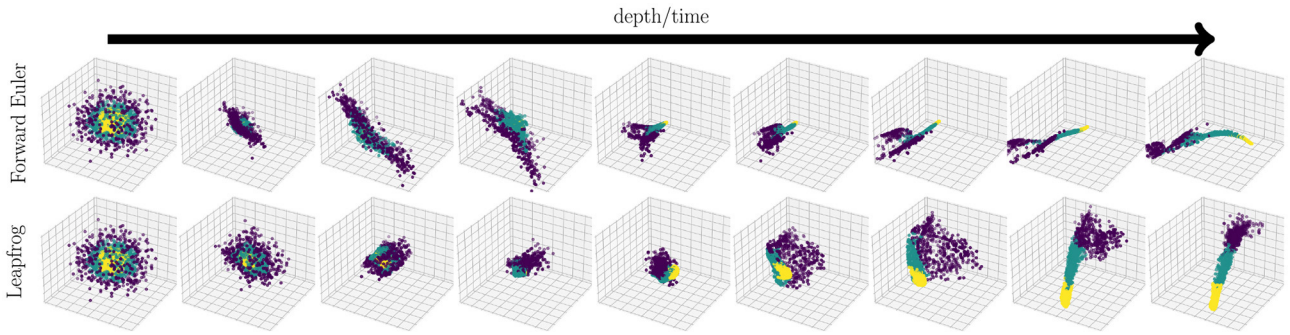
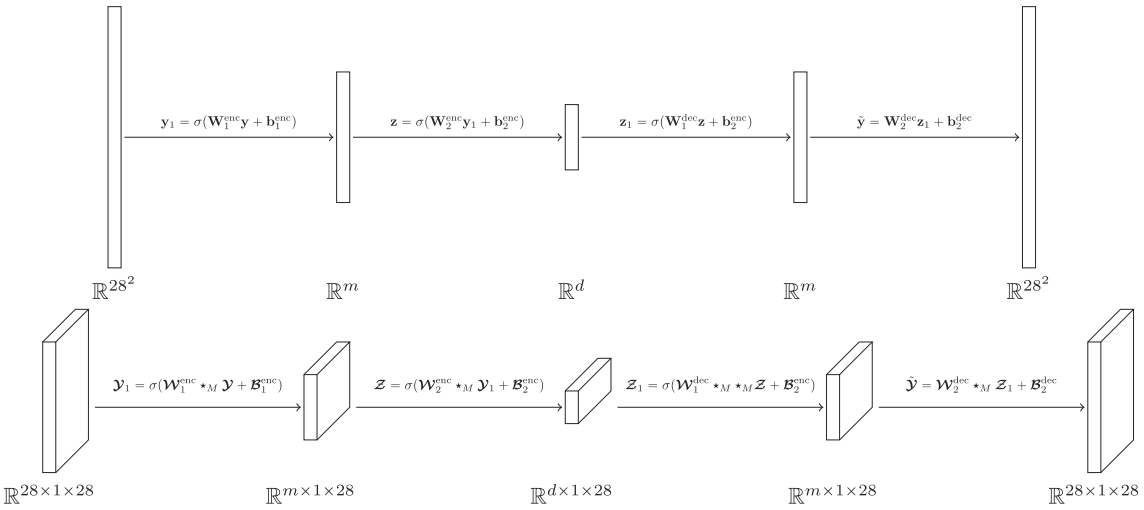


FIGURE 4
Comparison of feature trajectories for stable t-NNs with forward Euler (23b) and leapfrog integration (23a). Each image contains the features at a particularly layer of a trained network (layers $j = 0, 4, \ldots, 32$). The forward Euler network resulted in a test accuracy of 90% and the leapfrog network resulted in a test accuracy of 93.50%. As expected, the leapfrog trajectory is smoother and contains rotational dynamics. The colors are linearly separable at the last layer, indicating good classification performance.



name	$\lvert\theta\rvert$
NN(40,10)	64,394
NN(21,280)	45,794
NN(560,280)	1,193,864
t-NN(20,10)	44,744

FIGURE 5
Description of MNIST autoencoder architectures with $\sigma(x) = \tanh(x)$. **(Top)** Four-layer matrix autoencoder NN(m,d) with first width m and latent space dimension d. **(Middle)** Four-layer tensor autoencoder t-NN(m,d) with first width m and latent space dimension d. For notational simplicity, we omit the vector lateral slice notation for the t-NN. **(Bottom)** Table of networks that we use. We pick sizes relative on the dimensions of t-NN(20,10). The first network NN(40,10) is given more features in the first layer of the encoder. The second network NN(21,280) is given the same number of latent space features and a corresponding width to have roughly the same number of weights as the t-NN. The third network NN(560,280) is given the same number of features on both layers as the t-NN.

We report the convergence and accuracy results in Figure 7 and Table 2, respectively.

The t-NN architecture with cross entropy loss outperforms all networks in terms of test accuracy and accuracy per class. The second-best performing network is the t-NN with t-loss. These results are evidence that the features learned from the tensor linear layer (layer 1) are better than those learned by a dense matrix layer. We further note that matrix network NN with square weights has the same final layer shape as the t-NN with cross entropy loss; the only difference between the networks is the first layer. We depict the learned features of each network that preserves the size of the images in Figure 8. The t-NN features from the first layer contain more structure than the NN features with square weights. This reflects the $\star_M$-operation, which first acts along the tubes (rows of the images in Figure 8). We also observe that the features of NN are more extreme, close to $+1$ and -1, the limits of the range of the activation function $\sigma(x) = \tanh(x)$. In comparison, the features extracted from the t-NN with t-loss offer more variety of entries, but still hits the extreme values often. This demonstrates that t-NNs still produce rich feature spaces and are able to achieve these features with about 20 times fewer weights.

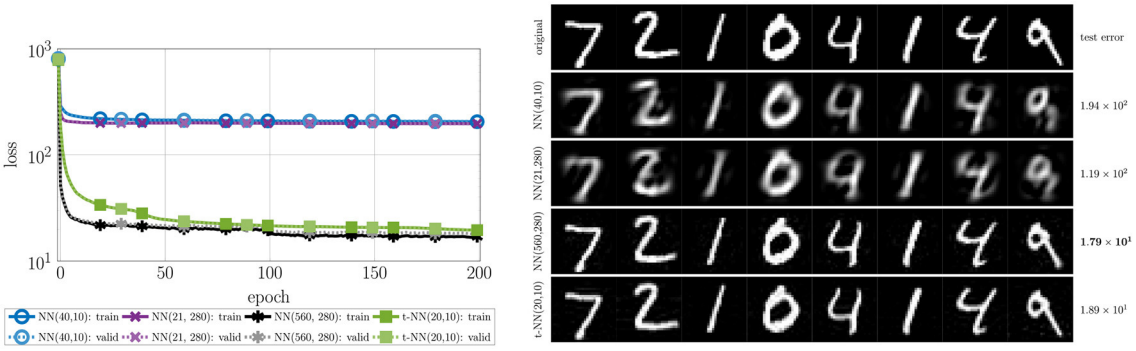


FIGURE 6
(Left) Convergence of the loss for the autoencoder example. The t-NN converges to a lower loss more quickly than the NNs. The validation loss closely follows the training loss, indicating good generalization. **(Right)** Autoencoder approximations to test images. The top row contains the true test images $\mathbf{y}$ and the subsequent rows contains the approximations to the true image $\tilde{\mathbf{y}} = f_{\text{dec}}(f_{\text{end}}(\mathbf{y}, \boldsymbol{\theta}_{\text{enc}}), \boldsymbol{\theta}_{\text{dec}})$ for various autoencoder architectures. To the right of each row, we report the average test error, $\frac{1}{|\mathcal{Y}_{\text{test}}|} \sum_{\mathbf{y} \in \mathcal{Y}_{\text{test}}} \|\mathbf{y} - \tilde{\mathbf{y}}\|_2$. Compared to the first two NN autoencoders, the t-NN produces clearer approximations and a test error an order of magnitude smaller. The autoencoder NN(560,280) does produce the smallest test error, but requires over 20 times more network weights than the t-NN autoencoder.

TABLE 1 **Description of MNIST two-layer network architectures with $\sigma(x) = \tanh(x)$.**

Name	Architecture	Layer 1	Layer 2	$\lvert\theta\rvert$
NN	$\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{y}_0 + \mathbf{b}_1) + \mathbf{b}_2$	$\mathbf{W}_1$: 39×28^2	$\mathbf{W}_2$: 10×39	31,015
		$\mathbf{b}_1$: 39×1	$\mathbf{b}_2$: 10×1	
NN, square	$\mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{y}_0 + \mathbf{b}_1) + \mathbf{b}_2$	$\mathbf{W}_1$: $28^2 \times 28^2$	$\mathbf{W}_2$: 10×28^2	623,290
		$\mathbf{b}_1$: $28^2 \times 1$	$\mathbf{b}_2$: 10×1	
t-NN	$\mathbf{W}_2\,\text{unfold}(\sigma(\boldsymbol{\mathcal{W}}_1 \star_M \vec{\boldsymbol{\mathcal{Y}}}_0 + \vec{\boldsymbol{\mathcal{B}}}_1)) + \mathbf{b}_2$	$\boldsymbol{\mathcal{W}}_1$: $28 \times 28 \times 28$	$\mathbf{W}_2$: 10×28^2	30,586
		$\vec{\boldsymbol{\mathcal{B}}}_1$: $28 \times 1 \times 28$	$\mathbf{b}_2$: 10×1	
t-NN, t-loss	$\boldsymbol{\mathcal{W}}_2 \star_M \sigma(\boldsymbol{\mathcal{W}}_1 \star_M \vec{\boldsymbol{\mathcal{Y}}}_0 + \vec{\boldsymbol{\mathcal{B}}}_1) + \vec{\boldsymbol{\mathcal{B}}}_2$	$\boldsymbol{\mathcal{W}}_1$: $28 \times 28 \times 28$	$\boldsymbol{\mathcal{W}}_2$: $10 \times 28 \times 28$	30,856
		$\vec{\boldsymbol{\mathcal{B}}}_1$: $28 \times 1 \times 28$	$\vec{\boldsymbol{\mathcal{B}}}_2$: $10 \times 1 \times 28$	

The t-NN architectures were chosen to preserve the size of the images. The NN width of 39 is chosen to be as small as possible such that the NN architecture does not have fewer parameters than the t-NN architecture. The final column reports the total number of learnable weights.

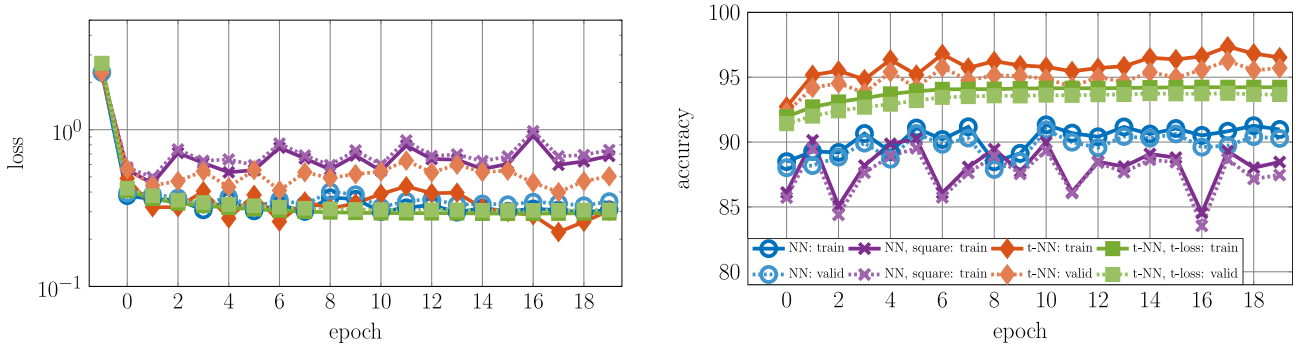


FIGURE 7
Loss and accuracy convergence for MNIST with four different two-layer neural networks. To see the differences clearly, we omit the initial accuracy, which was close to 10% for each network. The t-NN with cross entropy loss (orange diamonds ♦) produces the best training (darker, solid) and validation (lighter, dashed) accuracy. The t-NN with t-loss (green squares ■) performs second best with in terms of accuracy, demonstrating the benefits of tensor operator layers. Despite having the greatest number of weights, the NN with square weights (purple ×'s) performs worst in terms of accuracy.

We note that the t-NN network with t-cross entropy loss performs well, and we gain insight into the beneifts of t-losses from the accuracy per class in Figure 9. We observe that when we use tubal losses, we require high values for many frontal slices. This creates a more rigorous classification requirement and, as we will see in subsequent experiments, can yield networks that

TABLE 2 MNIST training, validation, and test accuracy per class and overall for the four architectures.

		0	1	2	3	4	5	6	7	8	9	Overall
Train	NN	97.05	96.54	90.58	89.73	93.03	88.91	92.36	91.91	87.79	84.57	91.33
	NN, square	96.67	93.85	92.58	84.82	94.84	89.11	97.91	88.76	77.98	85.56	90.24
	t-NN	**98.98**	**99.64**	**97.12**	**97.88**	**98.29**	**96.06**	**98.68**	**97.60**	**93.30**	**95.59**	**97.36**
	t-NN, t-loss	97.28	97.06	95.97	91.09	94.92	92.50	97.26	92.26	91.99	91.61	94.22
Valid	NN	96.90	96.46	89.50	89.79	92.13	89.94	92.28	91.33	88.85	81.02	90.94
	NN, square	94.50	93.36	90.60	84.68	94.99	89.83	**97.36**	89.26	77.81	82.16	89.52
	t-NN	**97.60**	**99.29**	**95.40**	**97.01**	**97.34**	**95.66**	96.75	**97.08**	**92.47**	**93.26**	**96.26**
	t-NN, t-loss	96.20	97.08	94.80	90.94	93.46	93.60	97.26	92.46	92.16	89.21	93.76
Test	NN	98.16	97.18	89.44	92.18	93.08	88.00	90.50	89.49	87.58	85.43	91.20
	NN, square	97.14	94.89	92.44	85.84	95.01	88.79	**97.49**	87.45	78.23	83.35	90.11
	t-NN	**98.67**	**99.38**	**95.25**	**97.03**	**98.37**	**94.96**	97.08	**96.50**	93.74	**94.05**	**96.55**
	t-NN, t-loss	98.16	98.33	95.35	93.66	95.21	91.59	97.18	90.66	**93.94**	91.08	94.57

The t-NN architecture with cross entropy loss consistently produces the highest accuracy. The bolded values indicate the highest accuracy for the class and dataset.

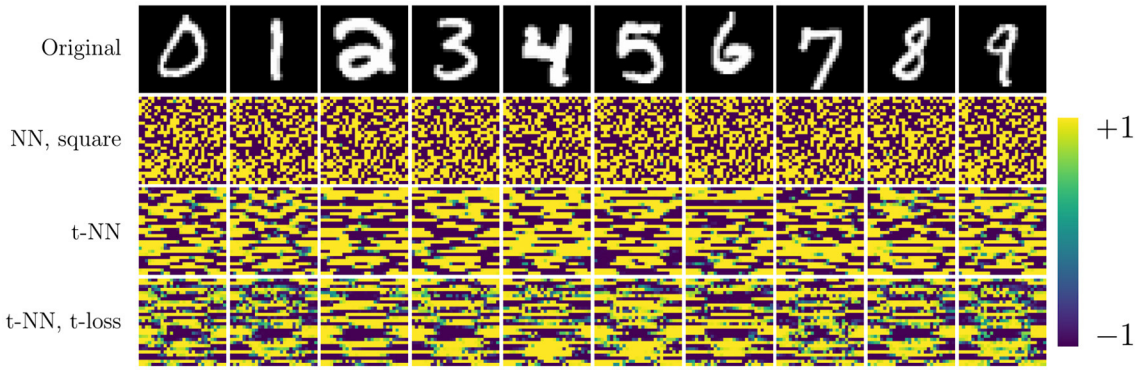


FIGURE 8
Features from first layer of NN, square, t-NN with cross entropy, and t-NN with t-cross entropy networks. Both t-NN features contain more structure because the $\star_M$-operation respects spatial correlations.

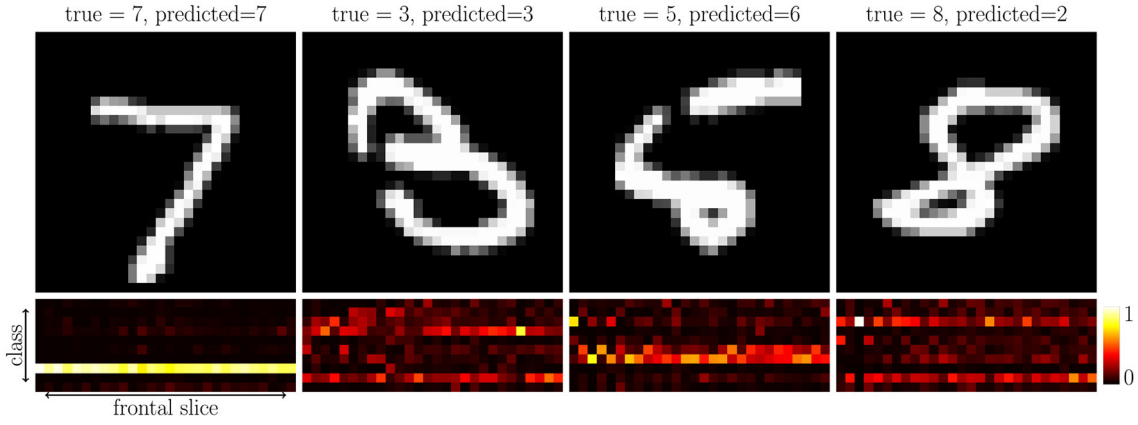


FIGURE 9
Illustration of t-softmax of MNIST test images using t-NN with t-loss. The top row are the test images $\vec{\mathcal{Y}} \in \mathbb{R}^{28\times1\times28}$ and the bottom row are the values of the tubal softmax of the output $F_{\text{tnn}}(\vec{\mathcal{Y}}, \boldsymbol{\theta}) \in \mathbb{R}^{10\times1\times28}$, shown in the transform domain. Each row of the tubal softmax images corresponds to a different class and each column to a different frontal slice. The row with the largest ℓ_2-norm corresponds to the predicted class, where the top row corresponds to class 0 and the bottom row corresponds to class 9. The left two images were predicted correctly and the right two images were predicted incorrectly.

TABLE 3 Description of CIFAR-10 Hamiltonian network architectures with $\sigma(x) = \tanh(x)$.

Name	Hamiltonian layers	Final layer	$\lvert\theta\rvert$
NN	$\mathbf{W}_j$: $(3 \cdot 32^2) \times (3 \cdot 32^2)$	$\mathbf{W}_{d+1}$: 10×3072	4 layers: 37, 779, 470
	b_j: 1×1	$\mathbf{b}_{d+1}$: 10×1	8 layers: 75, 528, 210
t-NN	$\mathbf{W}_j$: $(3 \cdot 32) \times (3 \cdot 32) \times 32$	$\mathbf{W}_{d+1}$: $10 \times (3 \cdot 32^2)$	4 layers: 1, 210, 506
	$\mathbf{b}_j$: $1 \times 1 \times 32$	$\mathbf{b}_{d+1}$: 10×1	8 layers: 2, 390, 282
t-NN, t-loss	$\mathcal{W}_j$: $(3 \cdot 32) \times (3 \cdot 32) \times 32$	$\mathcal{W}_{d+1}$: $10 \times (3 \cdot 32) \times 32$	4 layers: 1, 210, 816
	$\vec{\mathcal{B}}_j$: $1 \times 1 \times 32$	$\vec{\mathcal{B}}_{d+1}$: $10 \times 1 \times 32$	8 layers: 2, 390, 592

The architectures were chosen to preserve the sizes of the CIFAR-10 images. The final column $|\theta|$ reports the total number of learnable weights.

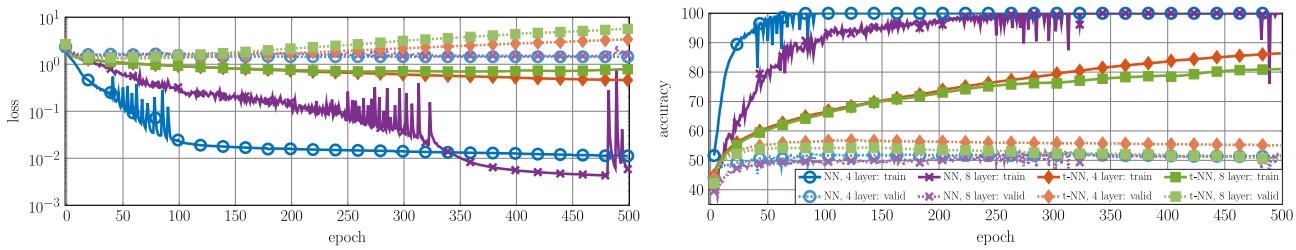


FIGURE 10
Loss and accuracy convergence for CIFAR-10 with different Hamiltonian network depths. We only show the convergence for the t-NN with t-cross entropy loss, which achieved a top validation accuracy of at least 54.37%, compared to the t-NN with cross entropy loss, which topped out at 54.32%. For the accuracy, we start with epoch 5 to highlight the differences between networks.

TABLE 4 CIFAR-10 training, validation, and test accuracy per class and overall for the four architectures.

		Plane	Car	Bird	Cat	Deer	Dog	Frog	Horse	Ship	Truck	Overall
Train	NN4	**99.95**	**100.00**	**99.95**	**100.00**	**99.98**	**99.97**	**100.00**	**99.97**	**100.00**	**100.00**	**99.98**
	NN8	99.90	99.97	99.80	99.95	99.93	100.00	100.00	100.00	99.93	99.93	99.94
	t-NN4	74.12	79.97	60.44	43.02	59.82	65.27	65.66	88.47	77.14	74.83	68.84
	t-NN8	76.09	81.15	61.12	35.80	69.04	58.83	62.52	91.45	76.94	75.28	68.79
Valid	NN4	54.57	62.77	40.85	29.58	39.36	42.77	58.24	59.52	**70.84**	60.06	51.99
	NN8	**62.08**	65.54	39.76	**33.92**	38.53	43.36	**62.34**	56.62	70.74	59.06	53.30
	t-NN4	59.51	68.32	**49.11**	32.68	49.53	**52.25**	51.41	**73.62**	67.64	**62.86**	**56.83**
	t-NN8	60.43	**66.63**	47.12	23.37	**53.79**	44.82	47.90	73.24	65.33	59.86	54.37
Test	NN4	57.30	64.70	40.60	28.10	38.30	42.10	57.30	58.20	**69.20**	57.90	51.37
	NN8	**62.60**	63.10	40.70	33.20	39.10	44.80	**61.60**	55.30	69.00	55.80	52.52
	t-NN4	61.00	66.90	48.40	**34.40**	48.10	**54.70**	54.50	72.50	67.30	**63.30**	**57.11**
	t-NN8	**62.60**	**67.30**	**48.70**	25.20	**52.50**	46.70	50.20	**72.90**	63.40	60.10	54.96

The t-NN architectures with t-cross entropy loss produce the highest overall validation and test accuracy, the traditional metrics to indicate generalization ability. The bolded values indicate the highest accuracy for the class and dataset.

generalize better. Additionally, the distribution of values in the tubal softmax function is reflective of the predicted class. For the second image (true = 3), the two most likely predicted classes were, in order, 3 and 8. Qualitatively, the particularly handwritten 3 has similarities to the digit 8, and the tubal softmax captures this similarity. For the cases that were incorrectly predicted the digit, the handwritten image had structure emblematic of the predicted class and second most likely predicted class matched the true label.

6.4 CIFAR-10

The CIFAR10 dataset (Krizhevsky and Hinton, 2009) is composed of $32 \times 32 \times 3$ RGB natural images belonging to ten classes. We train on 40,000 images and reserve 10,000 for validation. We report test accuracy on 10,000 images not used for training nor validation. For NNs, we vectorize images and store as columns, resulting in a matrix of size $(3 \cdot 32^2) \times b$ where b is the number of images. For t-NNs, we store the images as lateral slices

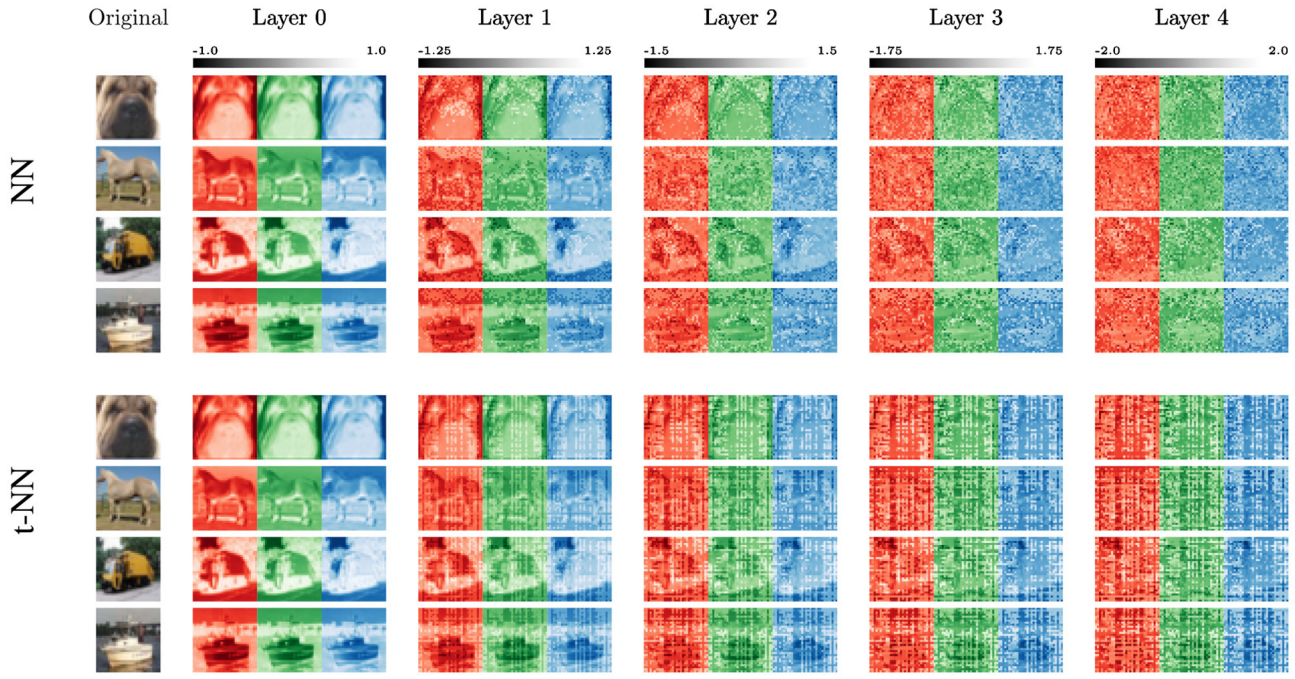


FIGURE 11
Features from the trained 4-layered Hamiltonian networks of both the matrix and tensor parameterized cases for four different training images (top-to-bottom: dog, horse, truck, ship). For the t-NN, we use the better-performing network with t-cross entropy loss. Here, Layer 0 shows the separated color channels of the original images.

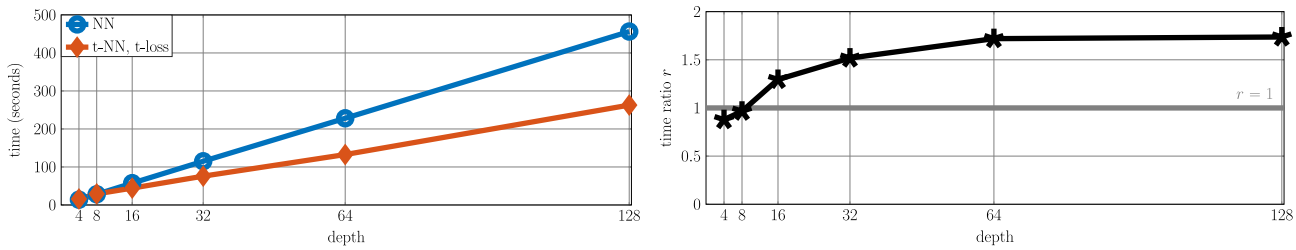
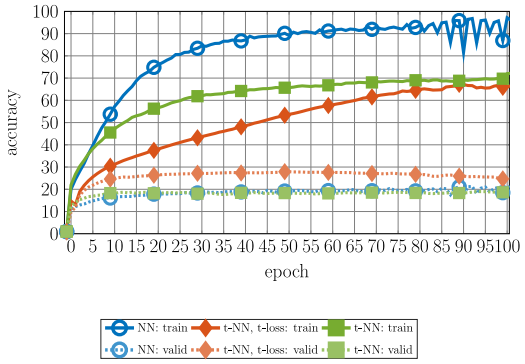


FIGURE 12
(Left) Average time per epoch to train a Hamiltonian network for a fixed batch size. We ran each depth for five epochs and the maximum standard deviation of time was on the order of 10^{-2} relative to the average time. **(Right)** Time ratio average r = NN epoch/t-NN epoch. As the depth of the network grows, the time per epoch for NNs takes almost 1.75 times longer than for t-NNs.

and stack the color channels vertically, resulting in a tensor of size $(3 \cdot 32) \times b \times 32$. We train for 500 epochs using Adam with a batch size of 32. We use a learning rate of 10^{-3} that decays after every 100 epochs by a factor of 0.9. We use the PyTorch defaults for the other optimizer hyperparameters. For the t-cross entropy loss, we use a squared ℓ_2-norm and normalize by the number of entries in the tube.

We compare the performance of the Hamiltonian networks with dense matrix operators and dense tensor operators for various numbers of layers ($d = 4, 8$). In conjunction with the Hamiltonian network, we use the smoothing regularizer (20) with a regularization parameter of $\lambda = 10^{-2}$. We describe the network architectures and number of parameters in Table 3. The NN architectures require more than 30 times the number of weights than the t-NN architectures. We compare the convergence and accuracy results in Figure 10 and Table 4, respectively.

There are several key takeaways from the numerical results. First, the depth of the network did not significantly change performance in this experiment. We state this observation cautiously. We observe this behavior for a certain set of fixed hyperparameters (e.g., step size h, learning rate, regularization parameter λ,...). The interaction of the hyperparameters and performance is complex and a complete ablation study is outside of the scope of this paper. A second takeaway is that the t-NN trained with the tubal loss generalizes better the NN networks and better than t-NNs with cross entropy loss (not shown for simplicity; see Figure 10 for details). This behavior is especially apparent when looking at the test loss in Table 4. The t-NN with four Hamiltonian layers and t-loss performs well overall, obtaining almost 5% better overall test accuracy. In comparison, the matrix NN quickly overfits the training data and thus does not generalize as well. In terms of the test accuracy per class,



		accuracy (best validation)
train	NN	**99.84**
	t-NN	82.02
	t-NN, t-loss	53.29
valid	NN	24.81
	t-NN	20.64
	t-NN, t-loss	**27.91**
test	NN	23.52
	t-NN	19.66
	t-NN, t-loss	**27.75**

FIGURE 13
(Left) Convergence of the accuracy for the CIFAR-100 experiment. To delineate the performance on the validation data, we show the first 100 epochs out of 500 total. The t-NN with t-loss converges to the highest validation accuracy and avoids the generalization gap longest out of all presented networks. **(Right)** Final accuracy for the training, validation, and test data. We report the results using the networks that produced the highest validation accuracy during training.

the t-NN architectures achieve the best performance in all but two classes.

To look into the performance further, we examine the extracted features of Hamiltonian NNs and t-NNs in Figure 11. We see that the NN and t-NN features share similarities. Both features gradually remove the structure of the original image at similar rates. The t-NN architecture achieves this pattern with significantly fewer network weights (over 30 times fewer). The noisy artifacts differ between the two architectures. In particular, we see that the t-NN layers produce blockier artifacts because of the structured $\star_M$-operation.

In the last numerical study in Figure 12, we explore how quickly we can train Hamiltonian t-NNs compared to NNs. In addition to an order of magnitude fewer weights, training t-NNs takes less time than training NNs. As we increase the depth of the networks, we see that each NN epoch takes approximately 1.75 times longer to complete. This performance could potentially be further improved if we optimized the $\star_M$-product, e.g., using fast transforms instead of matrix multiplication.

6.5 CIFAR-100

The CIFAR100 dataset (Krizhevsky and Hinton, 2009) is composed of $32 \times 32 \times 3$ RGB natural images belonging to 100 classes. We train on 40,000 images and reserve 10,000 for validation. We report test accuracy on 10,000 images not used for training nor validation. We use the same setup and training parameters as the CIFAR10 experiment (Section 6.4). For all experiments, we use Hamiltonian networks with a depth of $d = 16$, a step size of $h = 0.25$, and a regularization parameter $\lambda = 10^{-2}$. We report the accuracy results in Figure 13.

Observing the results in Figure 13, the conclusions are the same as in the CIFAR-10 experiment. Specifically, training with the t-NN and t-loss produces the best test accuracy, about a 4% improvement from the comparable NN network. This demonstrates that the benefits of dense tensor operations over dense matrix operations can be realized for more challenging classification problems and motivates further development of these tools to improve state-of-the-art convolutional neural networks and other architectures.

7 Conclusion

We presented tensor neural networks (t-NNs) as a new approach to parameterize fully-connected neural networks. We operate using the $\star_M$-product which can reduce the number of network weights by an order of magnitude while maintaining the same expressiveness. We introduced tubal loss functions that are an algebraically-consistent t-NN architecture. Because the $\star_M$-framework gives rise to a tensor algebra that preserves matrix properties, we extended the notion of stable neural networks to t-NNs, which enable the development of deeper, more expressive networks. Through numerical experiments on benchmark image classification tasks, we demonstrated that t-NNs offer a more efficient parameterization and, when trained with tubal loss functions, can generalize better to unseen data.

Our work opens the door to several natural extensions. First, we note that while this paper focused on imaging benchmark problems in machine learning, the $\star_M$-framework can be applied to many data sets, including dynamic graphs (Malik et al., 2021), longitudinal omics data (Mor et al., 2022), and functional magnetic resonance imaging (fMRI) (Keegan et al., 2022). Second, we could use tensor parameterizations to improve convolutional neural networks (CNNs), just as we used t-NN layers to improve fully-connected networks. CNNs are state-of-the-art for image classification and rely on convolution operations. The $\star_M$-product is, in some sense, a convolution based on the transformation $\mathbf{M}$; in fact, when $\mathbf{M}$ is the discrete Fourier transform, the result is a circulant convolution. A t-CNN could offer more efficient parameterization and a greater range of convolutional features that could increase the expressibility of the network. Third, we could extend the use of tubal loss functions to any network architecture. Tubal loss functions offer more stringent requirements to fitting

data which can mitigate overfitting. Additionally, tubal loss functions foster a new level of flexibility to evaluate performance, such as various norms to transform tubal probabilities into scalars and new measures of accuracy per frontal slice. Fourth, we can consider learning the operator $\mathbf{M}$ based on the data or allowing the operator to evolve with the layers. Lastly, we can explore methods to improve t-NN efficiency on CPUs and GPUs by exploiting the parallelize of the $\star_M$-products.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found at: https://pytorch.org/vision/main/datasets.html and https://github.com/elizabethnewman/tnn.

Author contributions

EN: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing—original draft, Writing—review & editing. LH: Conceptualization, Data curation, Funding acquisition, Writing—original draft, Writing—review & editing. HA: Conceptualization, Data curation, Writing—original draft, Writing—review & editing. MK: Funding acquisition, Writing—original draft, Writing—review & editing.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This work was partially funded by the Exploratory Science program at IBM. EN's work was partially supported by the National Science Foundation (NSF) under grants [DMS-2309751]. HA's work was partially funded an IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Acknowledgments

We thank Dr. Eldad Haber and Dr. Lars Ruthotto for providing additional insight on their work which helped us generalize their ideas to our high-dimensional framework.

Conflict of interest

LH is employed by IBM.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Ascher, U. M. (2010). *Numerical Methods for Evolutionary Differential Equations*. Philadelphia, PA: SIAM.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* 5, 157–166. doi: 10.1109/72.279181

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *SIAM Rev.* 60, 223–311. doi: 10.1137/16M1080173

Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L. (2011). *Handbook of Markov Chain Monte Carlo*. Boca Raton, FL: Chapman and Hall/CRC. doi: 10.1201/b10905

Cao, X., Rabusseau, G., and Pineau, J. (2017). Tensor regression networks with various low-rank tensor approximations. *arXiv* [Preprint]. arxiv:1712.09520. doi: 10.48550/arxiv.1712.09520

Carroll, J. D., and Chang, J.-J. (1970). Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart-Young" decomposition. *Psychometrika* 35, 283–319. doi: 10.1007/BF02310791

Chattopadhyay, A., Hassanzadeh, P., and Pasha, S. (2020). Predicting clustered weather patterns: a test case for applications of convolutional neural networks to spatio-temporal climate data. *Sci. Rep.* 10:1317. doi: 10.1038/s41598-020-57897-9

Chien, J.-T., and Bao, Y.-T. (2018). Tensor-factorized neural networks. *IEEE Trans. Neural Netw.* 29, 1998–2011. doi: 10.1109/TNNLS.2017.2690379

Cichocki, A., Lee, N., Oseledets, I., Phan, A.-H., Zhao, Q., Mandic, D. P., et al. (2016). Tensor networks for dimensionality reduction and large-scale optimization: part 1 low-rank tensor decompositions. *Found. Trends Mach. Learn.* 9, 249–429. doi: 10.1561/2200000059

de Lathauwer, L., de Moor, B., and Vandewalle, J. (2000). A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.* 21, 1253–1278. doi: 10.1137/S0895479896305696

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Frietas, N. (2013). "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems 26*, eds. C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Curran Associates, Inc.), 2148–2156. Available online at: http://papers.nips.cc/paper/5025-predicting-parameters-in-deep-learning.pdf (accessed December 18, 2023).

Ee, W. (2017). A proposal on machine learning via dynamical systems. *Comm. Math. Stat.* 5, 1–11. doi: 10.1007/s40304-017-0103-z

Haber, E., and Ruthotto, L. (2017). Stable architectures for deep neural networks. *Inverse Probl.* 34:1. doi: 10.1088/1361-6420/aa9a90

Haber, E., Ruthotto, L., Holtham, E., and Jun, S.-H. (2018). Learning across scales–multiscale methods for convolution neural networks. *Proc. AAAI Conf. Artif. Intell.* 32, 3142–3148. doi: 10.1609/aaai.v32i1.11680

Hao, N., Kilmer, M. E., Braman, K., and Hoover, R. C. (2013). Facial recognition using tensor-tensor decompositions. *SIAM J. Imaging Sci.* 6, 437–463. doi: 10.1137/110842570

Harshman, R. A. (1970). "Foundations of the parafac procedure: models and conditions for an "explanatory" multimodal factor analysis," in *UCLA Working Papers in Phonetics*, 16, 1–84. (University Microfilms, Ann Arbor, Michigan, No. 10,085).

He, K., Zhang, X., Ren, S., and Sun, J. (2016). "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV: IEEE), 770–778. doi: 10.1109/CVPR.2016.90

Jagtap, A. D., Shin, Y., Kawaguchi, K., and Karniadakis, G. E. (2022). Deep kronecker neural networks: a general framework for neural networks with adaptive activation functions. *Neurocomputing* 468, 165–180. doi: 10.1016/j.neucom.2021.10.036

Keegan, K., Vishwanath, T., and Xu, Y. (2022). A tensor SVD-based classification algorithm applied to fmri data. *SIAM Undergrad. Res. Online* 15, 270–294. doi: 10.1137/21S1456522

Kernfeld, E., Kilmer, M., and Aeron, S. (2015). Tensor-tensor products with invertible linear transforms. *Linear Algebra Appl.* 485, 545–570. doi: 10.1016/j.laa.2015.07.021

Kilmer, M. E., Braman, K., Hao, N., and Hoover, R. C. (2013). Third-order tensors as operators on matrices: a theoretical and computational framework with applications in imaging. *SIAM J. Matrix Anal. Appl.* 34, 148–172. doi: 10.1137/110837711

Kilmer, M. E., Horesh, L., Avron, H., and Newman, E. (2021). Tensor-tensor algebra for optimal representation and compression of multiway data. *Proc. Natl. Acad. Sci. USA*. 118:e2015851118. doi: 10.1073/pnas.2015851118

Kilmer, M. E., and Martin, C. D. (2011). Factorization strategies for third-order tensors. *Linear Algebra Appl.* 435, 641–658. doi: 10.1016/j.laa.2010.09.020

Kingma, D. P., and Ba, J. (2015). "Adam: a method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, May 7-9, 2015, Conference Track Proceedings*, eds. Y. Bengio, and Y. LeCun (San Diego, CA).

Kolda, T. G., and Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Rev.* 51, 455–500. doi: 10.1137/07070111X

Kossaifi, J., Lipton, Z. C., Kolbeinsson, A., Khanna, A., Furlanello, T., Anandkumar, A., et al. (2020). Tensor regression networks. *J. Mach. Learn. Res.* 21, 1–21. doi: 10.48550/arXiv.1707.08308

Krizhevsky, A., and Hinton, G. (2009). *Learning multiple layers of features from tiny images*. Technical report, University of Toronto. Available online at: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems, Vol. 25*, eds. F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger (Red Hook, NY: Curran Associates, Inc.) 1091–1105. Available online at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Kullback, S., and Leibler, R. A. (1951). On information and sufficiency. *Ann. Math. Stat.* 22, 79–86. doi: 10.1214/aoms/1177729694

LeCun, Y., Cortes, C., and Burges, C. J. C. (1998). *The MNIST Database of Handwritten Digits*. New York, NY. Available online at: http://yann.lecun.com/exdb/mnist/

LeCun, Y., Denker, J., and Solla, S. (1989). "Optimal brain damage," in *Advances in Neural Information Processing Systems, Volume 2*, ed. D. Touretzky (Cambridge, MA: Morgan-Kaufmann).

Li, D., Yu, Z., Wu, F., Luo, W., Hu, Y., Yuan, L., et al. (2020). The tensor-based feature analysis of spatiotemporal field data with heterogeneity. *Earth Space Sci.* 7:e2019EA001037. doi: 10.1029/2019EA001037

Lund, K. (2020). The tensor t- function: a definition for functions of third-order tensors. *Numer. Linear Algebra Appl.* 27:e2288. doi: 10.1002/nla.2288

Ma, A., and Molitor, D. (2022). Randomized Kaczmarz for tensor linear systems. *BIT Numer. Math.* 62, 171–194. doi: 10.1007/s10543-021-00877-w

Malik, O. A., Ubaru, S., Horesh, L., Kilmer, M. E., and Avron, H. (2021). "Dynamic graph convolutional networks using the tensor M-product," in *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)* (Philadelphia, PA), 729–737. doi: 10.1137/1.9781611976700.82

Mor, U., Cohen, Y., Valdés-Mas, R., Kviatcovsky, D., Elinav, E., Avron, H., et al. (2022). Dimensionality reduction of longitudinal omics data using modern tensor factorizations. *PLoS Comput. Biol.* 18, 1–18. doi: 10.1371/journal.pcbi.1010212

Newman, E. (2019). *A Step in the Right Dimension: Tensor Algebra and Applications* [PhD thesis]. Medford, MA: Tufts University.

Newman, E., Kilmer, M., and Horesh, L. (2018). "Image classification using local tensor singular value decompositions," in *2017 IEEE 7th International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP)* (Curacao: IEEE). doi: 10.1109/CAMSAP.2017.8313137

Newman, E., and Kilmer, M. E. (2020). Nonnegative tensor patch dictionary approaches for image compression and deblurring applications. *SIAM J. Imaging Sci.* 13, 1084–1112. doi: 10.1137/19M1297026

Nielsen, M. A. (2018). *Neural Networks and Deep Learning*. San Francisco, CA: Determination Press.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. (2015). "Tensorizing neural networks," in *Advances in Neural Information Processing Systems 28*, eds. C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (Curran Associates, Inc.), 442–450. Available online at: http://papers.nips.cc/paper/5787-tensorizing-neural-networks.pdf (accessed December 20, 2023).

Omberg, L., Golub, G. H., and Alter, O. (2007). A tensor higher-order singular value decomposition for integrative analysis of dna microarray data from different studies. *Proc. Nat. Acad. Sci.* 104, 18371–18376. doi: 10.1073/pnas.0709146104

Oseledets, I. V. (2011). Tensor-train decomposition. *SIAM J. Sci. Comput.* 33, 2295–2317. doi: 10.1137/090752286

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., et al. (2017). "Automatic differentiation in pytorch," in *NIPS-W* (Long Beach, CA).

Petersen, K. B., and Pedersen, M. S. (2012). *The Matrix Cookbook*. Lyngby: Technical University of Denmark.

Ronneberger, O., Fischer, P., and Brox, T. (2015). "U-net: convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2015*, eds. N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi (Cham: Springer International Publishing), 234–241. doi: 10.1007/978-3-319-24574-4_28

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature* 323, 533–536. doi: 10.1038/323533a0

Shalev-Shwartz, S., Shamir, O., and Shammah, S. (2017). "Failures of gradient-based deep learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70, ICML'17*, 3067–3075. Available online at: https://JMLR.org (accessed December 12, 2023).

Skeel, R. D. (1993). Variable step size destabilizes the stromer/leapfrog/verlet method. *BIT* 33, 172–175. doi: 10.1007/BF01990352

Soltani, S., Kilmer, M. E., and Hansen, P. C. (2016). A tensor-based dictionary learning approach to tomographic image reconstruction. *BIT Numer. Math.* 56, 1425–1454. doi: 10.1007/s10543-016-0607-z

TCA (2023). *Tufts community appeal*. Available online at: https://communityrelations.tufts.edu/engage-us/faculty-staff-students/tufts-community-appeal-tca (accessed December 20, 2023).

Tucker, L. R. (1966). Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 279–311. doi: 10.1007/BF02289464

Vasilescu, M. A. O., and Terzopoulos, D. (2002). "Multilinear analysis of image ensembles: tensorfaces," in *Computer Vision – ECCV 2002*, eds. A. Heyden, G. Sparr, M. Nielsen, and P. Johansen (Berlin: Springer Berlin Heidelberg), 447–460. doi: 10.1007/3-540-47969-4_30

Wang, M., Pan, Y., Xu, Z., Yang, X., Li, G., Cichocki, A., et al. (2023). *Tensor networks meet neural networks: A survey and future perspectives*. arXiv, Cornell University, Ithaca, NY.

Zhang, Z., Ely, G., Aeron, S., Hao, N., and Kilmer, M. (2014). "Novel methods for multilinear data completion and denoising based on tensor-SVD," in *2014 IEEE Conference on Computer Vision and Pattern Recognition* (Columbus, OH: IEEE). doi: 10.1109/CVPR.2014.485