A Unified Weight Initialization Paradigm for Tensorial Convolutional Neural Networks

Yu Pan [1] **Zeyong Su** [2] **Ao Liu** [3] **Jingquan Wang** [1] **Nannan Li** [4 5] **Zenglin Xu** [1 6]

Abstract

Tensorial Convolutional Neural Networks (TCNNs) have attracted much research attention for their power in reducing model parameters or enhancing the generalization ability. However, exploration of TCNNs is hindered even from weight initialization methods. To be specific, general initialization methods, such as Xavier or Kaiming initialization, usually fail to generate appropriate weights for TCNNs. Meanwhile, although there are ad-hoc approaches for specific architectures (e.g., Tensor Ring Nets), they are not applicable to TCNNs with other tensor decomposition methods (e.g., CP or Tucker decomposition). To address this problem, we propose a universal weight initialization paradigm, which generalizes Xavier and Kaiming methods and can be widely applicable to arbitrary TCNNs. Specifically, we first present the Reproducing Transformation to convert the backward process in TCNNs to an equivalent convolution process. Then, based on the convolution operators in the forward and backward processes, we build a unified paradigm to control the variance of features and gradients in TCNNs. Thus, we can derive fan-in and fan-out initialization for various TCNNs. We demonstrate that our paradigm can stabilize the training of TCNNs, leading to faster convergence and better results.

[1]Harbin Institute of Technology Shenzhen, Shenzhen, China [2]University of Electronic Science and Technology of China, Chengdu, China [3]Tokyo Institute of Technology, Tokyo, Japan [4]State Key Laboratory for Management and Control of Complex Systems, Institute of Automation, Chinese Academy of Sciences, Beijing, China [5]School of Artificial Intelligence, University of Chinese Academy of Sciences, Beijing, China [6]Pengcheng Laboratory, Shenzhen, China. Correspondence to: Yu Pan <iperryuu@gmail.com>, Zenglin Xu <zenglin@gmail.com>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

1. Introduction

Tensorial Convolutional Neural Networks (TCNNs) are important variants of Convolutional Neural Networks (CNNs). TCNNs usually adopt various tensor decomposition techniques to factorize large convolutional kernels into lower-rank tensor nodes, aiming to reduce the number of parameters. For example, Tensor Ring (TR) is utilized to decompose CNNs (Wang et al., 2018), leading to a high compression rate while maintaining comparably good performance. Tensor Train (TT) was used to improve performance of CNNs for image classification with parameter reduction (Yin et al., 2021). The CP-Higher-Order convolution (CP-HOConv) was proposed to factorize higher-order convolutional neural networks and has achieved the state-of-the-art results in spatio-temporal facial emotion analysis (Kossaifi et al., 2020).

In addition to the advantages in reducing model parameters, TCNNs are promising to be explored as a more general family of CNNs if the corresponding structures can be represented with hypergraphs. A hypergraph is a tensor diagram with a dummy tensor (as illustrated in Figure 1(c)) and a hyperedge (as illustrated in Figure 1(d)). Equipped with the hypergraph representation, TCNNs can include not only factorized CNNs based on tensor decomposition methods (e.g., Tensor Ring (TR) decomposition (Wang et al., 2018), Tensor Train (TT) decomposition (Novikov et al., 2015; Gao et al., 2019; Garipov et al., 2016), CANDECAMP/PARAFAC (CP) decomposition (Lebedev et al., 2015; Pan et al., 2022), Tucker decomposition (Kim et al., 2016; Elhoushi et al., 2019), Block-Term Tucker decomposition (Ye et al., 2018; 2020)), but also traditional CNN variants (e.g., low-rank convolution (Rigamonti et al., 2013; Idelbayev & Carreira-Perpiñán, 2020), factoring convolution (Szegedy et al., 2016), and even the vanilla convolution), since each of them can be represented as a hypergraph.

Despite these merits, TCNNs suffer from unstable training due to inappropriate weight initialization (Wang et al., 2018; Elhoushi et al., 2019). A common and direct initialization method generates weights by sampling from a probability distribution (Pan et al., 2019; Li et al., 2021). Unfortunately, this initialization method is sensitive to the choice of distribution variance; the distribution parameters are usually

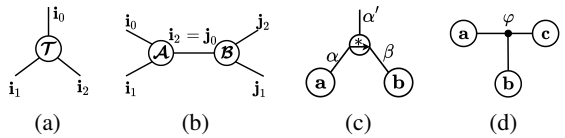


Figure 1. Tensor graphical instances. (a) A tensor $\mathcal{T} \in \mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1 \times \mathbf{i}_2}$; (b) Tensor Contraction; (c) Dummy Tensor; (d) Hyperedge.

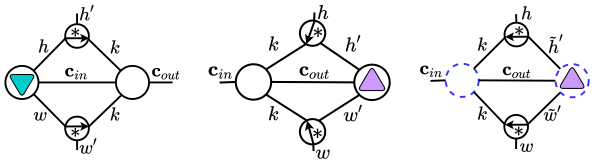


Figure 2. Illustration for vanilla CNN. (a) $\mathbf{G_f}$: A hypergraph forward process formulated as $\mathcal{Y} = \mathcal{X} \circledast \mathcal{C} + \mathbf{b}$, where $\mathcal{C} \in \mathbb{R}^{\mathbf{c}_{in} \times \mathbf{c}_{out} \times k \times k}$ denotes a convolutional kernel, $\mathcal{X} \in \mathbb{R}^{\mathbf{c}_{in} \times h \times w}$ denotes the input feature (Cyan Inverted Triangle), $\mathcal{Y} \in \mathbb{R}^{\mathbf{c}_{out} \times h' \times w'}$ denotes the output feature, $\mathbf{b} \in \mathbb{R}^{\mathbf{c}_{out}}$ represents the bias, and $\circledast$ denotes the convolutional operator. k represents kernel window size, $\mathbf{c}_{in}$ is the input channel, h and w denote height and width of $\mathcal{X}$, $\mathbf{c}_{out}$ is the output channel, h' and w' denote height and width of $\mathcal{Y}$; (b) $\mathbf{G_b}$: A hypergraph backward process derived directly from $\mathbf{G_f}$; (c) $\mathbf{G_{bt}}$: A hypergraph backward process equivalently transformed from $\mathbf{G_b}$ with Reproducing Transformation. $\mathbf{G_{bt}}$ is completely the same as $\mathbf{G_b}$. In (b) and (c), Purple Triangle means input gradient $\Delta\mathcal{Y} \in \mathbb{R}^{\mathbf{c}_{out} \times h' \times w'}$. $\tilde{h}'$ and $\tilde{w}'$ denote height and width of transformed $\Delta\mathcal{Y}$. In dummy tensor represented graphs, convolutional kernel vertex should connect arrow head of the dummy tensor and data-flow should connect the arrow tail. Thus, $\mathbf{G_f}$ and $\mathbf{G_{bt}}$ are convolutions, while $\mathbf{G_b}$ is not.

tuned manually, which is inefficient in practice. Another straightforward method is to borrow some adaptive weight initialization methods widely used in CNNs, such as the Xavier initialization (Glorot & Bengio, 2010) and the Kaiming initialization (He et al., 2015), however, they usually fail to initialize weights at a correct scale for TCNNs (Wang et al., 2020; Chang et al., 2020). In addition, ad-hoc initialization methods, such as modified Xavier methods proposed in (Wang et al., 2018; Chang et al., 2020) or the method of decomposing corresponding CNN weights (Elhoushi et al., 2019), are either designed for specific TCNNs, or dependent on special tensor decomposition methods.

Therefore, it is necessary to design a universal initialization scheme for various TCNN variants. To this end, we propose a unified paradigm that studies TCNNs from their topology. In detail, by extracting a backbone graph (BG) from a convolution hypergraph, we can encode an arbitrary TCNN into an adjacency matrix with the backbone graph and then calculate a suitable initial variance through the adjacency matrix, which can simply initialize any TCNN.

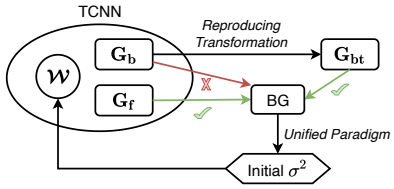


Figure 3. The overall workflow of the proposed unified initialization. A TCNN contains a forward hypergraph $\mathbf{G_f}$ and a backward hypergraph $\mathbf{G_b}$, besides the network weights $\mathcal{W}$. The objective is to achieve an acceptable variance σ^2 for $\mathcal{W}$ in order to keep the magnitude of data-flow stable across layers. To reach the goal, we derive a unified paradigm to calculate the desired σ^2. Note that the paradigm is applicable only to a backbone graph (BG) derived from a convolutional hypergraph. As $\mathbf{G_b}$ cannot be converted into the BG representation, we propose a reproducing transformation to transfer $\mathbf{G_b}$ in a convolutional representation $\mathbf{G_{bt}}$. With $\mathbf{G_f}$ and $\mathbf{G_{bt}}$, we can initialize TCNNs by regulating data-flow variance.

The unified paradigm can be applicable in controlling variance of two data-flow types, i.e., features in the forward process (fan-in mode) and gradients in the backward process (fan-out mode). For the fan-in mode, since the forward hypergraph (namely $\mathbf{G_f}$ in Figure 2(a)) is a dummy based convolution, the unified paradigm can inherently be applied. However, in the fan-out mode, the backward hypergraph (namely $\mathbf{G_b}$ in Figure 2(b)) cannot represent a convolution process due to the conflict with the dummy tensor definition in Section 2.2. To solve this problem, we originally propose the Reproducing Transformation to reproduce $\mathbf{G_b}$ as a convolution hypergraph $\mathbf{G_{bt}}$ shown in Figure 2(c). Through the Reproducing Transformation, the unified paradigm can be applicable to the backward process. The overall working flow is illustrated in Figure 3. In brief, our principal initialization can unify a variety of tensor formats, and meanwhile, fit both forward and backward propagation.

Through extensive experiments on various image classification benchmarks, we demonstrate that our method can produce appropriate initial weights for complicated TCNNs compared with classical initialization methods. Last but not least, we show that our paradigm is intrinsically a generalization of Xavier and relevant methods (Wang et al., 2018; He et al., 2015; Chang et al., 2020), while working more effectively for arbitrary TCNNs.

2. Preliminaries

In this section, we introduce the necessary preliminaries about tensors, and Xavier/Kaiming initialization.

2.1. Tensor Diagram

A tensor diagram mainly consists of two components, a tensor vertex and tensor contraction.

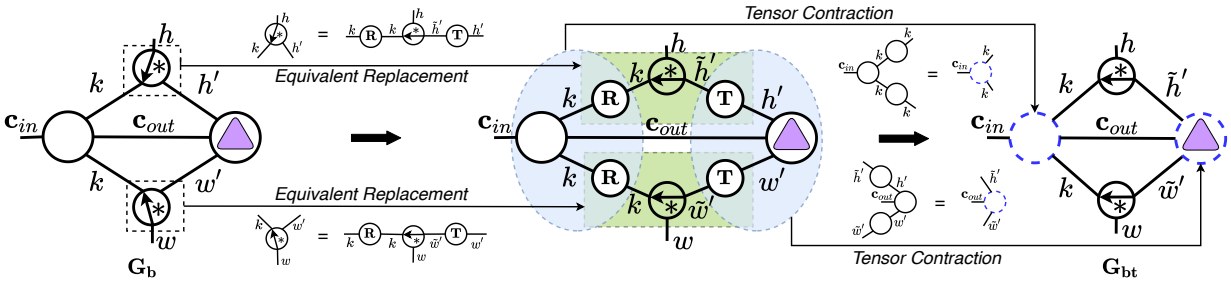


Figure 4. Reproducing Transformation, an equivalent transformation from $\mathbf{G_b}$ to $\mathbf{G_{bt}}$. Notations follow Figure 2. According to Section 2.2, the purple gradient vertex should connect to the arrow tails of dummy tensors, therefore $\mathbf{G_b}$ cannot denote a convolution. To overcome the problem, we design Reproducing Transformation to bond the gradient vertex to arrow tails for the convolution representation. Taking vanilla convolution as an example, Reproducing Transformation utilizes an equivalent replacement with reversal matrix $\mathbf{R}$ and transformation matrix $\mathbf{T}$ (details in Section 3.1) to exchange the arrow tail entry and the extra entry. Then $\mathbf{G_{bt}}$ can be derived through contracting $\mathbf{R}$ and $\mathbf{T}$ with the weight vertex and the gradient vertex, respectively.

Tensor Vertex. A tensor is denoted as a vertex whose order is given by the number of edges connected to it. The integer assigned to each edge denotes the dimension of the corresponding mode. For example, Figure 1(a) shows a 3rd-order tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1 \times \mathbf{i}_2}$.

Tensor Contraction. The inner-product of two tensors on matching modes denotes tensor contraction. As illustrated in Figure 1(b), a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1 \times \mathbf{i}_2}$ and a tensor $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{\mathbf{j}_0 \times \mathbf{j}_1 \times \mathbf{j}_2}$, can contract in the corresponding position, forming a new tensor of $\mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1 \times \mathbf{j}_2 \times \mathbf{j}_3}$, when they have equal dimensions: $\mathbf{i}_2 = \mathbf{j}_0 \triangleq e_0$. The contraction operation can be formulated as

$$(\boldsymbol{\mathcal{A}} \times_2^0 \boldsymbol{\mathcal{B}})_{i_0,i_1,j_2,j_3} = \sum_{m=0}^{e_0-1} \boldsymbol{\mathcal{A}}_{i_0,i_1,m} \boldsymbol{\mathcal{B}}_{m,j_2,j_3}. \quad (1)$$

2.2. Hypergraph

To enhance expressive ability of the tensor diagram in deep models, Hayashi et al. (2019) proposed hypergraph to represent forward process of TCNNs through the dummy tensor and the hyperedge.

Dummy Tensor. A vertex with an arrow symbol denotes a dummy tensor which is able to represent a convolutional operation. As depicted in Figure 1(c), for a dummy tensor $\boldsymbol{\mathcal{P}} \in \{0,1\}^{\alpha \times \alpha' \times \beta}$, α is the arrow tail entry, β is the arrow head entry and α' is the extra entry. Relation among the three entries is formulated as $\boldsymbol{\mathcal{P}}_{j,j',k} = 1$ if $j = sj' + k - p$ and 0 otherwise. Here, s represents the stride size; p denotes the padding size. A vector convolution in Figure 1(c) can be formulated as $\mathbf{c} = \mathbf{a} \times_0^0 \boldsymbol{\mathcal{P}} \times_1^0 \mathbf{b} = \mathbf{a} \circledast \mathbf{b} \in \mathbb{R}^{\alpha'}$, in which $\circledast$ is the convolutional operator. This formulation represents a convolution in which $\mathbf{a}$ means a data-flow and $\mathbf{b}$ denotes a convolutional kernel. For a dummy tensor, convolutional kernel vertex should connect arrow head of the dummy tensor and data-flow should connect the arrow tail.

Hyperedge. A hyperedge φ can connect to more than two tensor vertices. As shown in Figure 1(d), an output of a special case, connecting three vectors through a hyperedge, can be calculated as $y = \sum_{k=0}^{\varphi-1} \mathbf{a}_k \mathbf{b}_k \mathbf{c}_k$. There is usually at most one hyperedge in a hypergraph layer, connecting to all weight vertices (Hayashi et al., 2019). A hyperedge φ of a hypergraph represents summation over sub-structures, the parts without the hyperedge. For such an adding composite structure, we can derive the whole architecture initialization by processing each sub-structure.

2.3. Xavier and Kaiming Initialization

Xavier initialization (Glorot & Bengio, 2010) and Kaiming initialization (He et al., 2015) are widely used in CNNs. They aim to control the variance of features and gradients for stable training. We will introduce them through a vanilla CNN (Figure 2(a)), formulated as $\boldsymbol{\mathcal{Y}} = \boldsymbol{\mathcal{X}} \circledast \boldsymbol{\mathcal{C}} + \mathbf{b}$, where $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{\mathbf{c}_{in} \times \mathbf{c}_{out} \times k \times k}$ denotes a convolutional kernel, $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{\mathbf{c}_{in} \times h \times w}$ denotes the input, $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{\mathbf{c}_{out} \times h' \times w'}$ denotes the output, $\mathbf{b} \in \mathbb{R}^{\mathbf{c}_{out}}$ represents the bias, and $\circledast$ denotes the convolutional operator. k represents kernel window size, $\mathbf{c}_{in}$ is the input channel, h and w denote height and width of $\boldsymbol{\mathcal{X}}$, $\mathbf{c}_{out}$ is the output channel, and h' and w' denote height and width of $\boldsymbol{\mathcal{Y}}$.

Xavier initialization makes the following assumptions: (1) Elements of $\boldsymbol{\mathcal{C}}$, $\boldsymbol{\mathcal{X}}$ and $\mathbf{b}$ all satisfy the i.i.d. condition; (2) $\mathbb{E}(\boldsymbol{\mathcal{C}}) = 0$; (3) $\mathbb{E}(\boldsymbol{\mathcal{X}}) = 0$; and (4) $\mathbf{b} = \mathbf{0}$. There are two modes of Xavier initialization: (1) maintaining the variance of feature $\boldsymbol{\mathcal{X}}$ which is referred to as the fan-in mode: $\sigma^2(\boldsymbol{\mathcal{C}}) = \frac{1}{k^2 \mathbf{c}_{in}}$; (2) maintaining the variance of gradients as the fan-out mode: $\sigma^2(\boldsymbol{\mathcal{C}}) = \frac{1}{k^2 \mathbf{c}_{out}}$. In practice, the harmonic form is preferred: $\sigma^2(\boldsymbol{\mathcal{C}}) = \frac{2}{k^2(\mathbf{c}_{in} + \mathbf{c}_{out})}$.

Kaiming initialization extends the Xavier initialization to incorporate ReLU activation function. In accordance with Assumption (3) of Xavier initialization, Kaiming initializa-

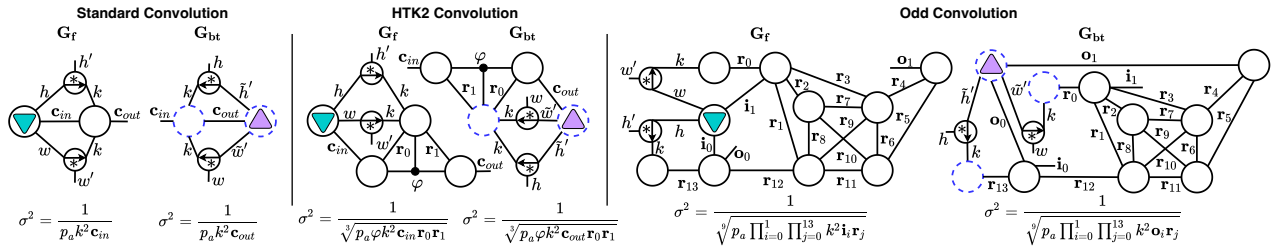


Figure 5. Reproducing Transformation Cases. σ^2 denotes initial variance of each wight vertex. (i) Standard Convolution; It is the most common convolution in CNNs. We observe that the graphical initialization will degenerate to Xavier/Kaiming initialization on the standard convolution, as they have the same weight variance formulation; (ii) Hyper Tucker-2 (HTK2) Convolution; Tucker-2 (TK2) is a classical tensor decomposition, known as the bottleneck structure in ResNet (He et al., 2016). We apply hyperedge to its weight vertices to form the HTK2; (iii) Odd Convolution; We introduce a particularly complicated tensor format (named Odd Tensor here) originally proposed by Li & Sun (2020). Odd Tensor contains 9 vertices and 14 edges. The connection among these vertices is irregular, making weight initialization a complex problem. By connecting all weight vertices with a hyperedge φ, it is flexible to construct HOdd (Graph-in: $\frac{1}{\sqrt[9]{p_a \varphi \prod_{i=0}^{1} \prod_{j=0}^{13} k^2 \mathbf{i}_i \mathbf{r}_j}}$; Graph-out: $\frac{1}{\sqrt[9]{p_a \varphi \prod_{i=0}^{1} \prod_{j=0}^{13} k^2 \mathbf{o}_i \mathbf{r}_j}}$). By successfully training Hyper Odd (HOdd) based networks, we can better demonstrate the potential adaptability of our method to diverse TCNNs.

tion requires the distribution of $\mathcal{C}$ to be symmetric. Similarly, Kaiming initialization also contains two modes: (1) the fan-in mode: $\sigma^2(\mathcal{C}) = \frac{2}{k^2 \mathbf{c}_{in}}$; (2) the fan-out mode: $\sigma^2(\mathcal{C}) = \frac{2}{k^2 \mathbf{c}_{out}}$.

3. Unified Initialization

In this section, we introduce our proposed unified initialization paradigm designed for various TCNNs. We first introduce our Reproducing Transformation, then we demonstrate the derivation of our unified paradigm, and finally we provide a simple exemplar initialization method that can be directly obtained based on the paradigm.

3.1. Reproducing Transformation

We build our unified initialization through derivation on a convolution hypergraph, whereby we can directly achieve the fan-in mode initialization from the forward hypergraph $\mathbf{G_f}$ since it is a natural convolution. However, the backward hypergraph $\mathbf{G_b}$ directly derived from $\mathbf{G_f}$ cannot represent a convolution as elaborated in Figure 2, which hinders the derivation of the fan-out mode. To solve this problem, we build Reproducing Transformation to convert $\mathbf{G_b}$ to a convolution hypergraph $\mathbf{G_{bt}}$. Before presenting the transformation, we first formulate the forward process.

In the forward process of a convolutional layer, we denote the output tensor by $\mathcal{Y}$ and the input tensor by $\mathcal{X}$. Then we have $\mathcal{Y} = \boldsymbol{a}(\boldsymbol{f}(\mathcal{X}, \boldsymbol{\theta})) \triangleq \boldsymbol{g}(\mathcal{X})$, where $\boldsymbol{f}(\cdot)$ means a linear mapping function, $\boldsymbol{\theta}$ denotes parameters of $\boldsymbol{f}(\cdot)$, and $\boldsymbol{a}(\cdot)$ denotes an activation function (usually a ReLU function).

For the backward propagation, $\mathfrak{L}$ denotes the Loss. In this process, we utilize a reversal matrix and a transformation matrix to achieve the equivalent transformation. These two auxiliary matrices will only change element position when they contract with another tensor, which helps calculate the variance of data-flow and weight vertices.

Reversal Matrix. A reversal matrix $\mathbf{R} \in \mathbb{R}^{r \times r}$ is an anti-diagonal matrix, where $\mathbf{R}_{ij} = 1$ when $i + j = r - 1$, $\mathbf{R}_{ij} = 0$ otherwise.

Transformation Matrix. A transformation matrix $\mathbf{T} \in \mathbb{R}^{t \times \tilde{t}}$ is an identity-like matrix, where $\tilde{t} = \varepsilon(t-1) + 1$ and $\varepsilon \in \mathbb{R}^N$ is a coefficient. $\mathbf{T}_{ij} = 1$ when $i = \frac{j}{\varepsilon}$, $\mathbf{T}_{ij} = 0$ otherwise.

With these two matrices, we can derive Theorem 3.1.

Theorem 3.1. *Given a vector* $\mathbf{a} \in \mathbb{R}^{\alpha}$ *and a vector* $\mathbf{b} \in \mathbb{R}^{\beta}$, *let* $\mathbf{y} = \mathbf{a} \circledast \mathbf{b} \in \mathbb{R}^{\alpha'}$, *then* $\Delta \mathbf{a} = \Delta \mathbf{y} \mathbf{T} \circledast \mathbf{R} \mathbf{b}$, *where* $\mathbf{R} \in \mathbb{R}^{\beta \times \beta}$ *denotes a reversal matrix,* $\mathbf{T} \in \mathbb{R}^{\alpha' \times \tilde{\alpha}'}$ *represents a transformation matrix,* $\circledast$ *means convolution operation and* $\Delta \bullet \triangleq \frac{\partial \mathfrak{L}}{\partial \bullet}$ *denotes the gradient.*

The proof of Theorem 3.1 is provided in Appendix A. Theorem 3.1 is corresponding to the equivalent replacement in Figure 4. We implement the Reproducing Transformation by applying the equivalent replacements to the original backward hypergraph $\mathbf{G_b}$, then, we contract $\mathbf{R}$ and $\mathbf{T}$ with the weight vertex and the gradient vertex, respectively. Finally, we can obtain the transformed backward hypergraph $\mathbf{G_{bt}}$ which denotes the backward convolution. We show some Reproducing Transformation cases in Figure 5.

3.2. Unified Paradigm

Here, we will derive a unified paradigm through variance analysis. We first give Proposition 3.2 and Proposition 3.3 to describe the relationship between variance and tensor calculation. Then we introduce Backbone Graph (BG) to illustrate inner production in a hypergraph. At last, we obtain

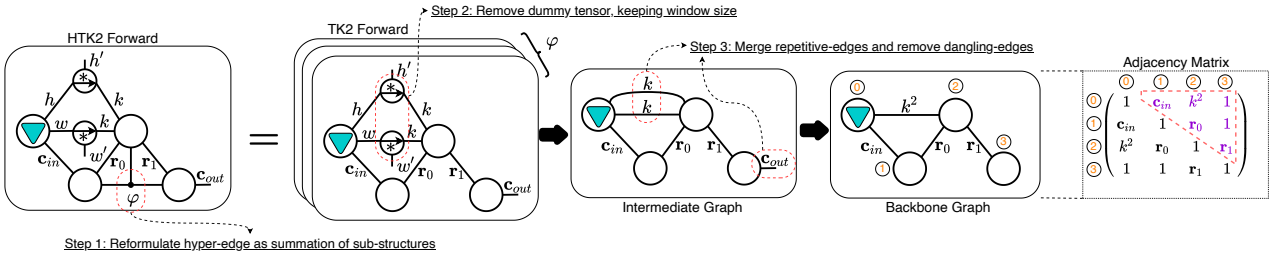


Figure 6. An example of deriving Graph-in mode for Hyper Tucker-2 (HTK2) convolution. Step 1: Since a hyperedge φ indicates adding operation over φ sub-structures (Tucker-2 here), we can derive the whole architecture initialization by processing each sub-structure. Step 2: Since a convolution only calculates on the kernel window, we can remove the dummy tensors by leaving kernel k to derive Intermediate Graph (IG). Step 3: Since elements of IG have same variance, we can further diminish $\mathbf{c}_{out}$ edge while merging repetitive-edges to derive Backbone Graph (BG). Then the initial variance of convolutional weights can be derived as $\frac{1}{\sqrt[3]{\boldsymbol{p_a}\varphi k^2 \mathbf{c}_{in}\mathbf{r}_0\mathbf{r}_1}}$ in terms to the adjacent matrix of BG, where $\boldsymbol{p_a}$ denotes the scale of activation function. Graph-out case is shown in Figure 12 of Appendix.

the paradigm in terms of BG and these two propositions.

Proposition 3.2. *Given tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{\mathbf{i}_0\times\mathbf{i}_1\times\cdots\times\mathbf{i}_{m-1}}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{\mathbf{i}_0\times\mathbf{i}_1\times\cdots\times\mathbf{i}_{m-1}}$, where elements of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ are independent with each other, the variance of their element-wise sum $\boldsymbol{\mathcal{Z}} = \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{Y}}$ is*

$$\sigma^2(\boldsymbol{\mathcal{Z}}) = \sigma^2(\boldsymbol{\mathcal{X}}) + \sigma^2(\boldsymbol{\mathcal{Y}}). \tag{2}$$

Proposition 3.3. *A tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{\mathbf{i}_0\times\mathbf{i}_1\times\cdots\times\mathbf{i}_{m-1}}$ (i.i.d.) and a tensor $\boldsymbol{\mathcal{Y}}$ contract d dimensions ($d \leq \min(m,n)$), where $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{\mathbf{j}_0\times\mathbf{j}_1\times\cdots\times\mathbf{j}_{n-1}}$ is i.i.d. and follows a zero-mean symmetrical distribution. The $\mathbf{x}_t$-th dimension of $\boldsymbol{\mathcal{X}}$ corresponds to the $\mathbf{y}_t$-th dimension of $\boldsymbol{\mathcal{Y}}$, where $\mathbf{x}_t \neq \mathbf{x}_u$ and $\mathbf{y}_t \neq \mathbf{y}_u$ if $t \neq u$, $\mathbf{x}_t \leq m-1$, and $\mathbf{y}_t \leq n-1$. Without loss of generality, let $\mathbf{i}_{\mathbf{x}_t} = \mathbf{j}_{\mathbf{y}_t} = \mathbf{v}_t$, for $t \in \{0,1,\ldots,d-1\}$. The variance of contracted tensor $\boldsymbol{\mathcal{Z}} = \boldsymbol{\mathcal{X}} \times^{\mathbf{y}_0,\mathbf{y}_1,\ldots,\mathbf{y}_{d-1}}_{\mathbf{x}_0,\mathbf{x}_1,\ldots,\mathbf{x}_{d-1}} \boldsymbol{\mathcal{Y}}$ is calculated by*

$$\sigma^2(\boldsymbol{\mathcal{Z}}) = \sigma^2(\boldsymbol{\mathcal{X}})\sigma^2(\boldsymbol{\mathcal{Y}}) \prod_{t=0}^{d-1} \mathbf{v}_t. \tag{3}$$

The proofs of the two propositions are given in Appendix B and C. It is worth mentioning that $\boldsymbol{\mathcal{X}}$ in Proposition 3.3 is hard to satisfy i.i.d, but assuming $\boldsymbol{\mathcal{X}}$ non-i.i.d is still applicable in practice as the empirical elaboration in Appendix H.1.

3.2.1. Backbone Graph

According to Proposition 3.3, variance change depends not only on weight and input, but also on contracted dimension $\mathbf{v}_t$. Therefore, we introduce Backbone Graph (BG) that only contains contracting edges (i.e., contracted dimensions). Figure 6 shows a process to derive BG from a dummy tensor based convolution. An adjacency matrix of τ-vertex BG is defined as $\mathbf{E} \in \mathbb{R}^{\tau\times\tau}$, whose element e_{ij} satisfying $e_{ij} = e_{ji}$ and diagonal element $e_{ii} = 1$, where $i,j \in \{0,1,\ldots,\tau-1\}$. As shown in Figure 6, the adjacency matrix in the tensor diagram is specially designed to fit the calculation of the variance where each element denotes the contraction between two nodes. Thus, $e_{ij} = 1$ means the contracting dimension between node i and node j is equal to 1, suggesting that there is no edge between node i and node j. $\mathbf{E}$ is symmetric and each vertex does not connect to itself. A supergraph denotes an output tensor $\boldsymbol{\mathcal{Y}}$. We use $BG(\mathbf{E})$ to denote the Backbone Graph that comes from $\boldsymbol{\mathcal{Y}}$. $BG(\mathbf{E})$ can be regarded as an element $\boldsymbol{\mathcal{Y}}_*$ of $\boldsymbol{\mathcal{Y}}$.

3.2.2. Derivation for Unified Paradigm

Since $\mathbf{E} \in \mathbb{R}^{\tau\times\tau}$ is symmetric, we consider edges e_{ij} satisfying $i < j$ only. Then based on Proposition 3.3, we present Theorem 3.4 to reveal the scale after the input through a TCNN. The proof of Theorem 3.4 is in Appendix D.

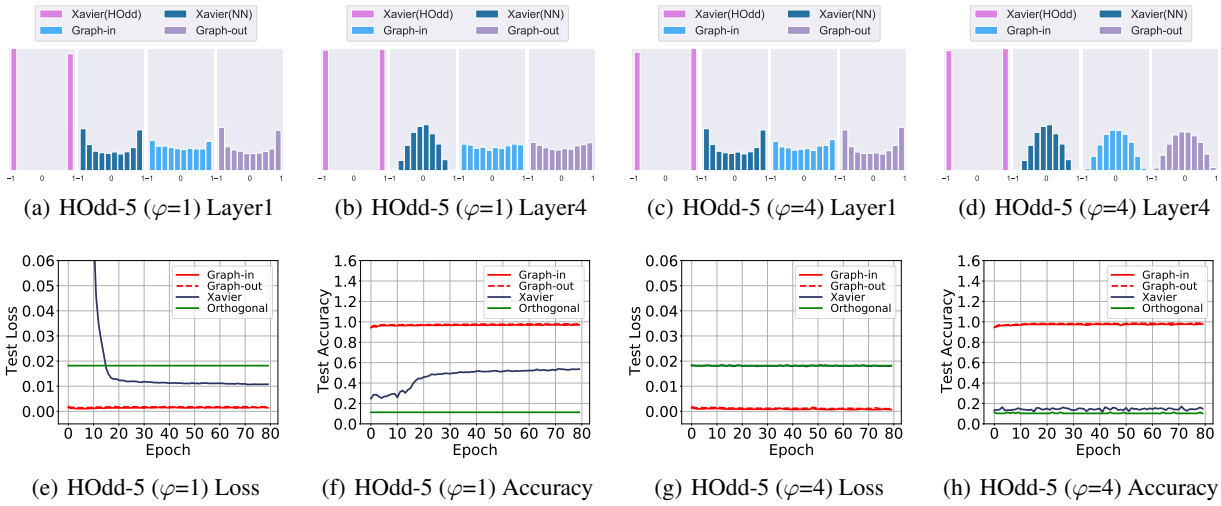
Theorem 3.4. *Assume the input $\boldsymbol{\mathcal{X}}$ contracts with n weight vertices $\{\boldsymbol{\mathcal{W}}^{(i)}\}_{i=0}^{n-1}$. Meanwhile, input variance is $\sigma^2(\boldsymbol{\mathcal{X}})$ and output variance is $\sigma^2(\boldsymbol{\mathcal{Y}})$, then*

$$\sigma^2(\boldsymbol{\mathcal{Y}}) = \sigma^2(\boldsymbol{\mathcal{X}}) \prod_{k=0}^{n-1} \sigma^2(\boldsymbol{\mathcal{W}}^{(k)}) \prod_{i=0}^{n-1}\prod_{j=i+1}^{\tau-1} e_{ij}. \tag{4}$$

Next, considering activation function and a hyperedge φ, variance of final output $\boldsymbol{\mathcal{Y}}_o$ is $\sigma^2(\boldsymbol{\mathcal{Y}}_o) = \boldsymbol{p_a}\varphi\sigma^2(\boldsymbol{\mathcal{Y}})$ according to Proposition 3.2, where $\boldsymbol{a}$ is an activation map, φ means a hyperedge value, and $\boldsymbol{p_a}$ denotes scale caused by activation function. For example, $\boldsymbol{p}_{\text{ReLU}} = \frac{1}{2}$ and $\boldsymbol{p}_{\tanh} = 1$. We set $\sigma^2(\boldsymbol{\mathcal{Y}}_o) = \sigma^2(\boldsymbol{\mathcal{X}})$ to maintain the data-flow variance equal. Thus, We can re-formulate Eq. (4) as

$$\frac{\sigma^2(\boldsymbol{\mathcal{X}})}{\boldsymbol{p_a}\varphi} = \sigma^2(\boldsymbol{\mathcal{X}}) \prod_{k=0}^{n-1} \sigma^2(\boldsymbol{\mathcal{W}}^{(k)}) \prod_{i=0}^{n-1}\prod_{j=i+1}^{\tau-1} e_{ij}. \tag{5}$$

From Eq. (5), We find that $\sigma^2(\boldsymbol{\mathcal{Y}}_o)$ is highly related to φ and edges of BG, and will change exponentially when edge number increases. Notably, Xavier and Kaiming fail since they only consider channel edges and convolutional window



(a) HOdd-5 (φ=1) Layer1 (b) HOdd-5 (φ=1) Layer4 (c) HOdd-5 (φ=4) Layer1 (d) HOdd-5 (φ=4) Layer4

(e) HOdd-5 (φ=1) Loss (f) HOdd-5 (φ=1) Accuracy (g) HOdd-5 (φ=4) Loss (h) HOdd-5 (φ=4) Accuracy

Figure 7. Activation distribution before training and results of the activation propagation analysis. Xavier (NN) and Xavier (HOdd) represent the case of applying Xavier initialization to Linear-5 and HOdd-5 respectively. Graph-in and Graph-out represent the case of applying the proposed initialization to HOdd-5. φ means a hyperedge. Xavier (NN) works since maintaining activation in the unsaturated region of activation function tanh, and Graph(-in/-out) also benefits from this. Under φ=1 and φ=4, activation of Graph(-in/-out) distributes in the unsaturated region, which indicates that Graph(-in/-out) can fit the sophisticated HOdd format and integrate with a hyperedge. Nevertheless, Xavier (HOdd) suffers from activation explosion in the saturated region and fails to train HOdd-5. Orthogonal initialization cannot train HOdd-5 either. By contrast, Graph(-in/out) successfully trains the model and derives relatively good results.

size edges, namely, part of edges. An expository example is in Appendix F.2. As a result, we can derive

$$\prod_{k=0}^{n-1} \sigma^2(\boldsymbol{\mathcal{W}}^{(k)}) = \frac{1}{\boldsymbol{p_a}\varphi \prod_{i=0}^{n-1} \prod_{j=i+1}^{\tau-1} e_{ij}}. \tag{6}$$

If the initialized weight satisfies Eq. (6), then we can attain the same effects as what Xavier and Kaiming achieve, even on multi-vertex tensor graphs. Thus, Eq. (6) can serve as a **unified paradigm** to ensure the effectiveness of weight initialization methods on TCNNs.

3.3. A Simple Initialization Exemplar

To ensure that Eq. (6) holds, there are plenty of choices to set the variance of weight vertices, which indicates potentially numerous weight initialization schemes. To verify the feasibility of our paradigm, we propose an exemplar choice by setting all the variance of weight vertices the same through

$$\sigma^2(\boldsymbol{\mathcal{W}}^{(*)}) = \frac{1}{\sqrt[n]{\boldsymbol{p_a}\varphi \prod_{i=0}^{n-1} \prod_{j=i+1}^{\tau-1} e_{ij}}}. \tag{7}$$

In this way, we can determine a specific weight initialization method, to which we refer as Graph Initialization. It has two modes, **Graph-in** and **Graph-out**, similar to fan-in and fan-out modes of Kaiming initialization.

Graph-in and Graph-out are constructed by applying Eq. (7) on a TCNN's $\mathbf{G_f}$ and $\mathbf{G_{bt}}$, respectively. We take derivation of Graph-in for HTK2 convolution as an instance in Figure 6. After extracting BG from the HTK2 $\mathbf{G_f}$, we can calculate suitable initial variance for weights of HTK2 convolution, by applying Eq. (7) on the BG's adjacency matrix. Graph-out is derived from $\mathbf{G_{bt}}$ exactly the same as Graph-in. We show some Graph Initialization demos in Figure 5.

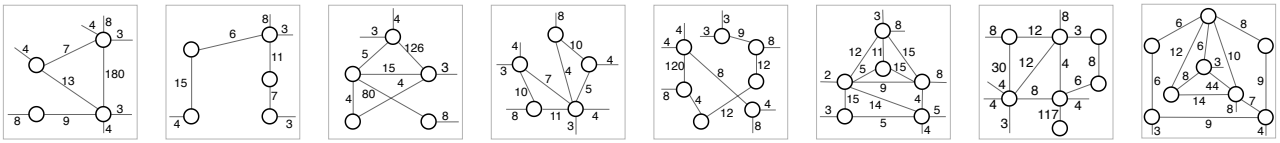
4. Experiment

In this section, we first give an illustrative example on linear layers (i.e., 1×1 convolution) to show the damage of activation amplification. Then, we use randomly generated tensor formats to show statistic results of our Graph(-in/-out) initialization. Next, by experiments on complex HOdd networks, we verify the adaption of the proposed method in complicated situations. Finally, via experiments on ImageNet with random networks, we show that our initialization is suitable for arbitrary TCNNs. We compare it with Xavier or Kaiming when the activation function is tanh or ReLU, respectively. We put details of all experiments in Appendix H, including the batch size, the learning rate, the network architectures and the training machine.

4.1. Evaluation on MNIST

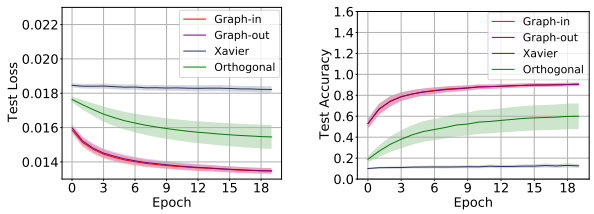
4.1.1. Activation Propagation Analysis

We conduct this experiment on MNIST to show that activation amplification in propagation is the main factor to cause unstable training. In this experiment, the network



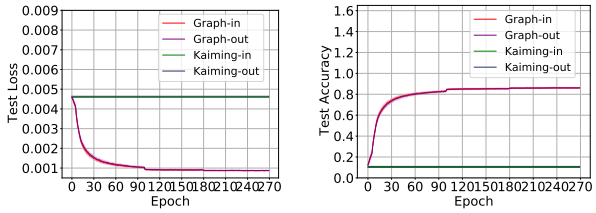
(a) Sam. 1 (b) Sam. 2 (c) Sam. 3 (d) Sam. 4 (e) Sam. 5 (f) Sam. 6 (g) Sam. 7 (h) Sam. 8

Figure 8. Random Tensor Formats. For more clear observation, we draw (a) - (h), in total eight random sub-structure TCNN samples without hyperedges ("Sam." is the abbreviation of "sample"). Notably, tensor nodes are largely different in size. For example, in (g), a large node can be of size $117 \times 8 \times 4 \times 6 \times 4 = 89856$ and a small one is $3 \times 8 = 24$, which indicates the random tensor formats are sophisticated sufficiently.



(a) MNIST Loss (b) MNIST Accuracy

Figure 9. Results of the random tensor format experiment. (a) and (b) draw 150 round results on MNIST. The results show that our method works consistently well.



(a) Cifar10 Loss (b) Cifar10 Accuracy

Figure 10. Results of the Cifar10 experiment. (a) and (b) draw 30 round results on Cifar10. The results show that our method is robust enough to initialize 270 tensor formats.

consists of 5 linear layers (called Linear-5) or 5 HOdd based linear layers (called HOdd-5). Each layer has 500 hidden units. We use Xavier initialization for comparison since the activation function here is a hyperbolic tangent function. For completeness, we also include Orthogonal initialization from PYTORCH (Paszke et al., 2019) for comparison. The training process is optimized by Adam with the learning rate 1e-4. Xavier(NN) and Xavier(HOdd) represent the case of applying Xavier initialization to Linear-5 and HOdd-5, respectively. Graph-in and Graph-out represent the cases of applying the graphical initialization to HOdd-5.

Figure 7 shows distribution of activations when applying Graph Initialization and Xavier initialization. The activation distributions of both Graph-in and Graph-out are similar to these of Xavier(NN). Specifically, the activations of these three cases are mostly distributed in the unsaturated area (-1, 1) of the activation function tanh, which benefits the training. However, Xavier(HOdd) encounters explosion and the activations are distributed mostly in the saturated region around -1 and 1, which shows the limitation of Xavier when applied to HOdd-5. As shown from Figure 7(e) to Figure 7(h), Graph(-in/-out) initialization leads to convergence almost at the beginning of the training, while Xavier initialization costs several epochs to converge, and Orthogonal initialization fails to train the network. In addition, when the hyperedge φ increases from 1 to 4, Xavier loses the ability to train HOdd-5, indicating that considering the hyperedge is necessary for initialization.

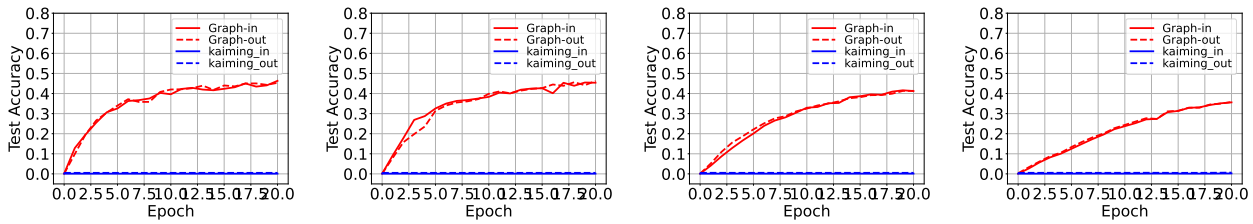
4.1.2. Random Tensor Formats

As a unified paradigm, our graphical initialization has the ability to initialize a variety of TCNNs successfully. In this section, we present the random layer experiment on MNIST to evaluate the generalization ability. To be specific, we design a TCNN consisting of four convolutional layers (referred to as Conv-4) and structure of each layer is generated randomly. Examples of the random layer are shown in Figure 8. Obviously, these examples are quite different in vertex numbers and edge values. In this experiment, we conduct 150 rounds of sub-experiments repeatedly and generate random kernels in every sub-experiment (up to 600 different kernel structures if not considering the circumstance of the same shapes). The activation function here is still a hyperbolic tangent function. Thus we compare four kinds of initialization: Graph(-in/-out), Xavier and Orthogonal. The batch size is 128. The optimizer is Adam with the learning rate 1e-4.

Results are shown in Figure 9. As for Xavier initialization, the network is hard to train and the test accuracy remains low. When using Orthogonal initialization, the network becomes trainable and obtains mediocre test results. However, our initialization Graph(-in/out) achieves the highest accuracy and the smallest loss with a faster speed, which shows great advantage when applied to TCNNs. Meanwhile, due to

Table 1. Top-1 accuracy on Cifar10 and Tiny-ImageNet. Rank-Edge Number means the least number of edges only connected to weight vertices in layers. Random-$*$ denotes randomly generating models. More results are in Appendix 8.

		Cifar10			Tiny-ImageNet		
	Rank-Edge Number	Kaiming (-in/-out)	Graph-in	Graph-out	Kaiming (-in/-out)	Graph-in	Graph-out
Low-Rank	1	0.1	0.8141	0.8163	0.307/0.2776	0.3153	0.3076
Tensor Ring	4	0.1	0.8308	0.8311	0.005	0.2494	0.249
HTK2(φ=4)	2	0.1	0.8638	0.8705	0.005	0.4014	0.4126
HOdd(φ=4)	14	0.1	0.8826	0.8806	0.005	0.5048	0.5045
Random-1	-	0.1	0.8538	0.8483	0.005	0.4965	0.5015
Random-2	-	0.1	0.8801	0.876	0.005	0.5379	0.5356
Random-3	-	0.1	0.8648	0.863	0.005	0.5475	0.5403
Random-4	-	0.1	0.8789	0.8816	0.005	0.5295	0.5306
Random-5	-	0.1	0.8622	0.8644	0.005	0.5444	0.5428



(a) HRand-RN-50 (b) HRand-RN-101 (c) HRand-gMLP-S16 (d) HRand-MLP-Mixer-B16

Figure 11. Top-1 accuracy plots on ImageNet. RN is short for ResNet.

the good performance in these random-structured networks, we believe that the proposed initialization methods will be widely applicable to arbitrary TCNNs.

4.2. Evaluation on Real-world Datasets

4.2.1. Evaluation on Cifar10

In this experiment, we construct experiments on Cifar10. Following Chang et al. (2020), to clearly showing the importance of weight initialization, we adopt the All Convolutional Net (All-Conv) (Springenberg et al., 2015) (containing 9 convolutional layers without Batch Normalization (Ioffe & Szegedy, 2015)) with the SGD optimizer with the learning rate 5e-3, while dropping the normalization tricks and the Adam optimization. We set two cases: (i) training tensorial All-Conv with random tensor formats for 30 round, totally $30 \times 9 = 270$ tensor formats; and (ii) training tensorial All-Conv with common tensor formats (e.g., Tensor Ring and HOdd) for 1 round.

Results of Case (i) are shown in Figure 10, Kaiming(-in/-out) is statistically hard to initialize a random tensorial All-Conv net. By contrast, Graph(-in/-out) performs a flexible adaption in a wide range of tensor formats. We show results of Case (ii) in Table 1. Graph(-in/-out) works fine from the simplest Low-Rank convolution to the most complex HOdd convolution, which indicates good applicability. However, Kaiming(-in/-out) still fails to initialize a common TCNN, even the Low-Rank convolution. Some comparisons with more common tensor formats (e.g., CP and TT) show the same results as Case (ii) in Appendix H.5.3.

In this paper, we focus on a unified method that adapts arbitrary TCNNs. Therefore, a comparison with ad-hoc methods are not the point. For completeness, we put such a comparison with the Cifar10 experiment setting in Appendix H.5.2, in which our method derives comparably good results.

4.2.2. Evaluation on Tiny-ImageNet

Tiny-ImageNet contains a subset of ImageNet's images and is a challenging large-scale dataset. As Kaiming initialization is often applied to ResNet, we also evaluate our initialization for tensorial ResNet-50 on Tiny-ImageNet. The optimizer is SGD with the learning rate 1e-1.

As shown in Table 1, Kaiming initialization almost fails to train all the listed models except the low-rank one, which can be interpreted by Eq. (4), i.e., variance of the output is sensitive to the number of edges. Kaiming-in seems to work for the low-rank ResNet, probably because the Low-rank convolution has much fewer edges than other TCNNs and is very close to a vanilla CNN. Also, batch normalization is used to aid the training, which is not always applicable in memory-constrained scenarios. Besides Kaiming-out still

cannot work even with batch normalization. Based on the above observations, we can see that although Kaiming initialization may work for some extremely simple TCNNs, it fails for complicated TCNNs. By contrast, our method can fit various TCNNs, including randomly generated architectures.

4.2.3. Evaluation on ImageNet

In this section, we employ ResNet (He et al., 2016), and two recent models gMLP (Liu et al., 2021) and MLP-Mixer (Tolstikhin et al., 2021) for validation of our initialization. Tensorial layers are all random layers (termed as HRand). We adopt these models' original settings that tensorial ResNet uses SGD, and tensorial gMLP/MLP-Mixer uses Adam.

As shown in Figure 11, results are highly consistent, namely, Graph(-in/-out) performs well in all three nets (i.e., ResNet, gMLP and MLP-Mixer), while Kaiming fails in all the situations. This phenomenon is reasonable since that data-flow variance will change exponentially when edge number increases. However, Kaiming ignores the inner production in tensor formats, leading to failure. More similar results are given in Appendix H.7. Notably, Graph(-in/-out) always derive similar results, which is hard to be explained as claimed in Chang et al. (2020). In practice, the two modes can be used optionally.

5. Conclusion

In this paper, we present Reproducing Transformation to denote backward process as a convolution, and then derive the unified paradigm to help stabilize arbitrary TCNNs. Based on this variance-control principle, the proposed graphical initialization method can avoid data-flow explosion, and have shown the ability to train diverse TCNNs successfully. In the future, we plan to explore the application of our method to Tensorial Recurrent Neural Networks.

Acknowledgments

This work was partially supported by the National Key Research and Development Program of China (No. 2018AAA0100204), and a key program of fundamental research from Shenzhen Science and Technology Innovation Commission (No. JCYJ20200109113403826).

References

Chang, O., Flokas, L., and Lipson, H. Principled weight initialization for hypernetworks. In *ICLR*. OpenReview.net, 2020.

Elhoushi, M., Tian, Y. H., Chen, Z., Shafiq, F., and Li, J. Y. Accelerating training using tensor decomposition. *CoRR*, abs/1909.05675, 2019.

Gao, Z., Cheng, S., He, R., Xie, Z., Zhao, H., Lu, Z., and Xiang, T. Compressing deep neural networks by matrix product operators. *CoRR*, abs/1904.06194, 2019.

Garipov, T., Podoprikhin, D., Novikov, A., and Vetrov, D. P. Ultimate tensorization: compressing convolutional and FC layers alike. *CoRR*, abs/1611.03214, 2016.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9 of *JMLR Proceedings*, pp. 249–256. JMLR.org, 2010.

Hayashi, K., Yamaguchi, T., Sugawara, Y., and Maeda, S. Exploring unexplored tensor network decompositions for convolutional neural networks. In *NeurIPS*, pp. 5553–5563, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pp. 1026–1034. IEEE Computer Society, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, pp. 770–778. IEEE Computer Society, 2016.

Idelbayev, Y. and Carreira-Perpiñán, M. Á. Low-rank compression of neural nets: Learning the rank of each layer. In *CVPR*, pp. 8046–8056. IEEE, 2020.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 448–456. JMLR.org, 2015.

Kim, Y., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. Compression of deep convolutional neural networks for fast and low power mobile applications. In *ICLR (Poster)*, 2016.

Kossaifi, J., Toisoul, A., Bulat, A., Panagakis, Y., Hospedales, T. M., and Pantic, M. Factorized higher-order cnns with an application to spatio-temporal emotion estimation. In *CVPR*, pp. 6059–6068. IEEE, 2020.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I. V., and Lempitsky, V. S. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *ICLR (Poster)*, 2015.

Li, C. and Sun, Z. Evolutionary topology search for tensor network decomposition. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5947–5957. PMLR, 2020.

Li, N., Pan, Y., Chen, Y., Ding, Z., Zhao, D., and Xu, Z. Heuristic rank selection with progressively searching tensor ring network. *Complex & Intelligent Systems*, pp. 1–15, 2021.

Liu, H., Dai, Z., So, D. R., and Le, Q. V. Pay attention to mlps. *CoRR*, abs/2105.08050, 2021.

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. Tensorizing neural networks. In *NIPS*, pp. 442–450, 2015.

Pan, Y., Xu, J., Wang, M., Ye, J., Wang, F., Bai, K., and Xu, Z. Compressing recurrent neural networks with tensor ring for action recognition. In *AAAI*, pp. 4683–4690. AAAI Press, 2019.

Pan, Y., Wang, M., and Xu, Z. Tednet: A pytorch toolkit for tensor decomposition networks. *Neurocomputing*, 469: 234–238, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8024–8035, 2019.

Rigamonti, R., Sironi, A., Lepetit, V., and Fua, P. Learning separable filters. In *CVPR*, pp. 2754–2761. IEEE Computer Society, 2013.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. Striving for simplicity: The all convolutional net. In *ICLR (Workshop)*, 2015.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. In *CVPR*, pp. 2818–2826. IEEE Computer Society, 2016.

Taki, M. Deep residual networks and weight initialization. *CoRR*, abs/1709.02956, 2017.

Tolstikhin, I. O., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., and Dosovitskiy, A. Mlp-mixer: An all-mlp architecture for vision. *CoRR*, abs/2105.01601, 2021.

Wang, M., Su, Z., Luo, X., Pan, Y., Zheng, S., and Xu, Z. Concatenated tensor networks for deep multi-task learning. In *ICONIP (5)*, volume 1333 of *Communications in Computer and Information Science*, pp. 517–525. Springer, 2020.

Wang, W., Sun, Y., Eriksson, B., Wang, W., and Aggarwal, V. Wide compression: Tensor ring nets. In *CVPR*, pp. 9329–9338. IEEE Computer Society, 2018.

Ye, J., Wang, L., Li, G., Chen, D., Zhe, S., Chu, X., and Xu, Z. Learning compact recurrent neural networks with block-term tensor decomposition. In *CVPR*, pp. 9378–9387. IEEE Computer Society, 2018.

Ye, J., Li, G., Chen, D., Yang, H., Zhe, S., and Xu, Z. Block-term tensor neural networks. *Neural Networks*, 130:11–21, 2020.

Yin, M., Sui, Y., Liao, S., and Yuan, B. Towards efficient tensor decomposition-based DNN model compression with optimization framework. In *CVPR*, pp. 10674–10683. Computer Vision Foundation / IEEE, 2021.

A. Proof of Theorem 3.1

Proof. As defined in Section 2.2, the forward dummy $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{\alpha \times \alpha' \times \beta}$ is defined as $\boldsymbol{\mathcal{P}}_{j,j',k} = 1$, if $j = sj' + k - p$ and otherwise $\boldsymbol{\mathcal{P}}_{j,j',k} = 0$. Then, $j \in \{0, 1, \dots \alpha - 1\}$, $j' \in \{0, 1, \dots, \alpha' - 1\}$ and $k \in \{0, 1, \dots, \beta - 1\}$. s means the stride and p means the padding.

The backward dummy $\boldsymbol{\mathcal{P}}' \in \mathbb{R}^{\alpha \times \tilde{\alpha}' \times \beta}$ is expected as $\boldsymbol{\mathcal{P}}'_{j,\tilde{j}',k} = 1$, if $\tilde{j}' = \tilde{s}j + \tilde{k} - \tilde{p}$ and otherwise $\boldsymbol{\mathcal{P}}'_{j,\tilde{j}',k} = 0$, where $\tilde{s} = 1$, $\tilde{k} = \beta - k - 1$ and $\tilde{p} = \beta - p - 1$.

The transformation matrix $\mathbf{T} \in \mathbb{R}^{\alpha' \times \tilde{\alpha}'}$ has $\tilde{\alpha}' = s\alpha' - s + 1$. The expansion of $\Delta\mathbf{y} \in \mathbb{R}^{\alpha'}$ into $\Delta\tilde{\mathbf{y}} \in \mathbb{R}^{\alpha'}$ can be represented as $\Delta\tilde{\mathbf{y}} = \Delta\mathbf{y}\mathbf{T}$. Here, we give an example of $s = 2$

$$\begin{bmatrix} \Delta\mathbf{y}_0 & \Delta\mathbf{y}_1 & \Delta\mathbf{y}_2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \Delta\mathbf{y}_0 & 0 & \Delta\mathbf{y}_1 & 0 & \Delta\mathbf{y}_2 \end{bmatrix}. \tag{8}$$

Obviously, $\Delta\mathbf{y}_j = \Delta\tilde{\mathbf{y}}_{\tilde{j}} = \Delta\tilde{\mathbf{y}}_{2j}$.

A reversal matrix $\mathbf{R} \in \mathbb{R}^{\beta \times \beta}$ is an anti-diagonal matrix, where $\mathbf{R}_{ij} = 1$ when $i + j = \beta - 1$ and $\mathbf{R}_{ij} = 0$ for other situation. Reverse $\mathbf{b} \in \mathbb{R}^{\beta}$ into $\tilde{\mathbf{b}} = \mathbf{R}\mathbf{b} \in \mathbb{R}^{\beta}$. Here is an example when $\beta = 3$

$$\begin{bmatrix} \mathbf{b}_0 & \mathbf{b}_1 & \mathbf{b}_2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_2 & \mathbf{b}_1 & \mathbf{b}_0 \end{bmatrix}.$$

Demonstrating $\Delta\mathbf{a} = \boldsymbol{dm}(\Delta\tilde{\mathbf{y}}, \tilde{\mathbf{b}})$ is equal to prove

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}' \times_1^1 \mathbf{T} \times_1^0 \mathbf{R}, \tag{9}$$

namely the equvilent replacement in Figure 4.

Contracting $\boldsymbol{\mathcal{P}}'$ with $\mathbf{T}$

$$\hat{\boldsymbol{\mathcal{P}}}' = \boldsymbol{\mathcal{P}}' \times_1^1 \mathbf{T}, \tag{10}$$

where $\hat{\boldsymbol{\mathcal{P}}}' \in \mathbb{R}^{\alpha \times \beta \times \alpha'}$. According to the definition of the transformation matrix, $\tilde{j}' = sj'$, where j' is the index of $\hat{\boldsymbol{\mathcal{P}}}'$ dimension α'. Therefore, a element $\hat{\boldsymbol{\mathcal{P}}}'_{j,k,j'}$ follows

$$sj' = \tilde{s}j + \tilde{k} - \tilde{p}. \tag{11}$$

Contracting $\hat{\boldsymbol{\mathcal{P}}}'$ with $\mathbf{R}$

$$\hat{\boldsymbol{\mathcal{P}}} = \hat{\boldsymbol{\mathcal{P}}}' \times_1^0 \mathbf{R}, \tag{12}$$

where $\hat{\boldsymbol{\mathcal{P}}} \in \mathbb{R}^{\alpha \times \alpha' \times \beta}$. Interacting with the reversal matrix, the index $\hat{k}$ of $\hat{\boldsymbol{\mathcal{P}}}$ dimension β follows $\hat{k} = \beta - \tilde{k} - 1$. Thus, $\hat{k} \equiv k$. Then, $\hat{\boldsymbol{\mathcal{P}}}_{j,j',\hat{k}} = 1$, if $sj' = \tilde{s}j + \beta - 1 - \hat{k} - \tilde{p}$ can also be formulated as $\hat{\boldsymbol{\mathcal{P}}}_{j,j',k} = 1$, if

$$sj' = \tilde{s}j - k + \beta - \tilde{p} - 1. \tag{13}$$

By replacing $\tilde{s}$ and $\tilde{p}$ with 1 and $\beta - p - 1$, we obtain

$$sj' = j - k + p, \tag{14}$$

namely

$$j = sj' + k - p, \tag{15}$$

which indicates that $\hat{\boldsymbol{\mathcal{P}}} \equiv \boldsymbol{\mathcal{P}}$. In conclusion, we demonstrate

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{P}}' \times_1^1 \mathbf{T} \times_1^0 \mathbf{R}. \tag{16}$$

□

B. Proof of Proposition 3.2

Lemma B.1. *Assuming that element in both vector* $\mathbf{a} \in \mathbb{R}^n$ *and* $\mathbf{b} \in \mathbb{R}^n$ *is independent to other elements in the same vector, the variance of vector* $\mathbf{c} = \mathbf{a} + \mathbf{b}$ *is* $\sigma^2(\mathbf{c}) = \sigma^2(\mathbf{a}) + \sigma^2(\mathbf{b})$.

Proof.

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Z}}) &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{Z}})) \\ &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{Y}})) \\ &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{X}}) + \boldsymbol{vec}(\boldsymbol{\mathcal{Y}})),\end{aligned}$$

where $\boldsymbol{vec}(\cdot)$ represents the vectorization operation. According to Lemma B.1, we get:

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Z}}) &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{X}})) + \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{Y}})) \\ &= \sigma^2(\boldsymbol{\mathcal{X}}) + \sigma^2(\boldsymbol{\mathcal{Y}}).\end{aligned}$$

□

C. Proof of Proposition 3.3

Lemma C.1. *Assume that elements of a vector* $\mathbf{a} \in \mathbb{R}^n$ *is i.i.d.. Meanwhile, a vector* $\mathbf{b} \in \mathbb{R}^n$ *(also i.i.d.) follows a symmetrical distribution with zero mean. Then the variance of* $c = \mathbf{a} \odot \mathbf{b}$ *is* $\sigma^2(c) = n\sigma^2(\mathbf{a})\sigma^2(\mathbf{b})$*, where* $\odot$ *denotes the inner-product, and* $\sigma^2(\cdot)$ *denotes the variance.*

Proof. Let $\boldsymbol{\mathcal{X}}^{(\mathbf{x}_*)} \in \mathbb{R}^{\mathbf{i}_{\mathbf{x}_0} \times \mathbf{i}_{\mathbf{x}_1} \times \cdots \times \mathbf{i}_{\mathbf{x}_{d-1}}}$ and $\boldsymbol{\mathcal{Y}}^{(\mathbf{y}_*)} \in \mathbb{R}^{\mathbf{j}_{\mathbf{y}_0} \times \mathbf{j}_{\mathbf{y}_1} \times \cdots \times \mathbf{j}_{\mathbf{y}_{d-1}}}$ denote sub-tensors where subscripts $\mathbf{x}_*$ and $\mathbf{y}_*$ indicate the fixed dimensions. An element of $\boldsymbol{\mathcal{Z}}$ is calculated as $\boldsymbol{\mathcal{Z}}_{\mathbf{x}_*,\mathbf{y}_*} = \boldsymbol{\mathcal{X}}^{(\mathbf{x}_*)} \odot \boldsymbol{\mathcal{Y}}^{(\mathbf{y}_*)}$. The change of variance derivation is shown as following

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Z}}) &= \sigma^2(\boldsymbol{\mathcal{Z}}_{\mathbf{x}_*,\mathbf{y}_*}) \\ &= \sigma^2(\boldsymbol{\mathcal{X}}^{(\mathbf{x}_*)} \odot \boldsymbol{\mathcal{Y}}^{(\mathbf{y}_*)}) \\ &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{X}}^{(\mathbf{x}_*)}) \odot \boldsymbol{vec}(\boldsymbol{\mathcal{Y}}^{(\mathbf{y}_*)})),\end{aligned}$$

where $\boldsymbol{vec}(\cdot)$ represents the vectorization operation. According to Lemma C.1, we get

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Z}}) &= \sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{X}}^{(\mathbf{x}_*)}))\sigma^2(\boldsymbol{vec}(\boldsymbol{\mathcal{Y}}^{(\mathbf{y}_*)})) \prod_{t=0}^{d-1} \mathbf{v}_t \\ &= \sigma^2(\boldsymbol{\mathcal{X}})\sigma^2(\boldsymbol{\mathcal{Y}}) \prod_{t=0}^{d-1} \mathbf{v}_t.\end{aligned}$$

□

D. Proof of Theorem 3.4

Proof. First of all,

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Y}}) &= \sigma^2(\boldsymbol{\mathcal{Y}}_*) = \sigma^2(BG(\mathbf{E})) \\ &= \sigma^2(\boldsymbol{\mathcal{X}} \times_*^0 \boldsymbol{\mathcal{W}}^{(0)} \times_*^{0,1} \boldsymbol{\mathcal{W}}^{(1)} \cdots \times_*^{0,1,\ldots,n-1} \boldsymbol{\mathcal{W}}^{(n-1)}).\end{aligned} \tag{17}$$

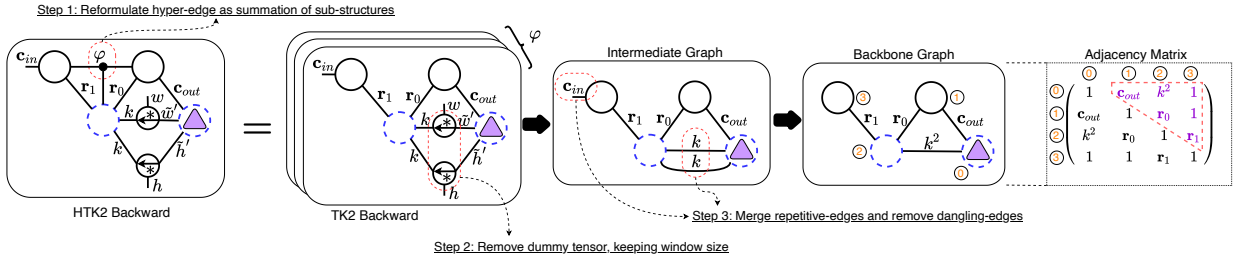


Figure 12. An example of deriving graph-out initialization for Hyper Tucker-2 (HTK2) convolution with the Backbone Graph. By unifying both the forward and backward processes, we can analyze backward propagation similarly to the forward. Then according to the adjacent matrix of Backbone Graph, the initial variance of convolutional weights can be derived as $\frac{1}{\sqrt[3]{\boldsymbol{p_a}\varphi k^2 \mathbf{c}_{out}\mathbf{r}_0\mathbf{r}_1}}$.

According to Proposition 3.3,

$$\begin{aligned}\sigma^2(\boldsymbol{\mathcal{Y}}) &= \sigma^2(\boldsymbol{\mathcal{X}} \times_*^0 \boldsymbol{\mathcal{W}}^{(0)} \times_*^{0,1} \boldsymbol{\mathcal{W}}^{(1)} \dots)\sigma^2(\boldsymbol{\mathcal{W}}^{(n-1)}) \prod_{i=0}^{n-1}\prod_{j=\tau-1}^{\tau-1} e_{ij} \\ &= \sigma^2(\boldsymbol{\mathcal{X}}) \prod_{k=0}^{n-1} \sigma^2(\boldsymbol{\mathcal{W}}^{(k)}) \prod_{i=0}^{n-1}\prod_{j=i+1}^{\tau-1} e_{ij}. \end{aligned} \tag{18}$$

□

E. Tensor Basis

E.1. Tenosr Notation

In this paper, a dth-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1 \cdots \times \mathbf{i}_{d-1}}$ is represented by a boldface Euler script letter. We denote a scalar $s \in \mathbb{R}^1$ by a lowercase letter, a vector $\mathbf{v} \in \mathbb{R}^i$ by a bold lowercase letter and a matrix $\mathbf{M} \in \mathbb{R}^{\mathbf{i}_0 \times \mathbf{i}_1}$ by a bold uppercase letter.

E.2. Dummy Tensor and Hyperedge

Traditional tensor diagram can only describe a tensor structure, which is limited to use in TCNNs. Therefore, to reinforce the representing ability of classical tensor diagram, Hayashi et al. (2019) propose a hypergraph to denote forward computation of convolutional layers by designing the dummy tensor and hyperedge.

Dummy Tensor As depicted in Figure 1(c), a vertex with a star symbol denotes a dummy tensor formulated as

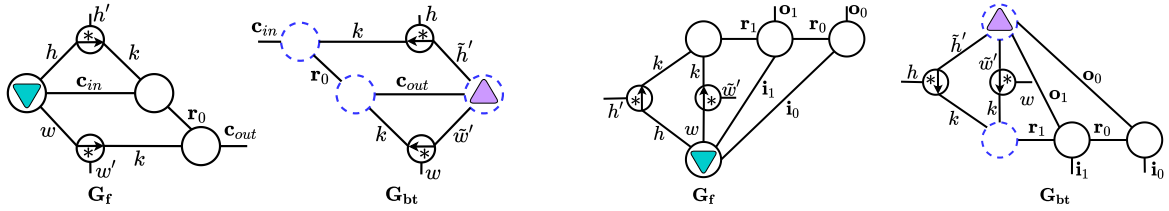
$$\mathbf{y}_{j'} = \sum_{j=0}^{\alpha-1}\sum_{k=0}^{\beta-1} \boldsymbol{\mathcal{P}}_{j,j',k}\mathbf{a}_j\mathbf{b}_k, \tag{19}$$

where $\mathbf{a} \in \mathbb{R}^{\alpha}$, $\mathbf{y} \in \mathbb{R}^{\alpha'}$, $\mathbf{b} \in \mathbb{R}^{\beta}$, $\boldsymbol{dm}$ denotes the operation of dummy tensor, and $\boldsymbol{\mathcal{P}} \in \{0,1\}^{\alpha\times\alpha'\times\beta}$ is a binary tensor with elements defined as $\boldsymbol{\mathcal{P}}_{j,j',k} = 1$ if $j = sj' + k - p$ where s and p represent stride and padding operation respectively, and 0 otherwise.

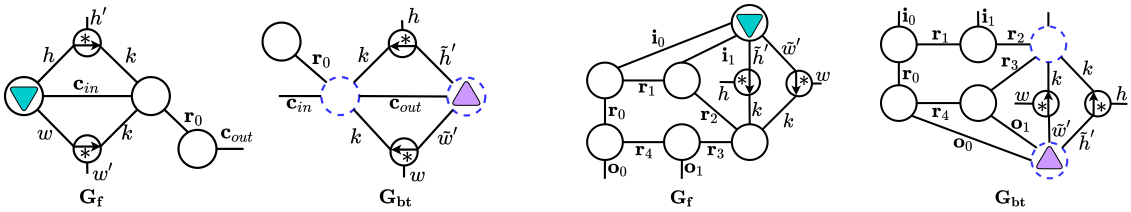
Hyperedge As illustrated in Figure 1(d), an output of a special case, connecting three vectors through a hyperedge, can be calculated

$$y = \sum_{k=0}^{\varphi-1} \mathbf{a}_k\mathbf{b}_k\mathbf{c}_k, \tag{20}$$

where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^{\varphi}$, $y \in \mathbb{R}^1$, $\boldsymbol{he}$ denotes the operation of a hyperedge. Interestingly, a hyperedge is simply equal to a tensor, whose diagonal elements are 1. This tensor indicates the adding operation over several sub-structures (e.g., CNNs). For such an adding composite structure, the key point is the initialization problem of its sub-structures, which is what we focus on in this paper. Hayashi et al. (2019) show that tensor graph can represent arbitrary TCNNs by introducing dummy tensors and hyperedges.



(a) Low-Rank Convolution (b) Tensor Train Convolution



(c) HyperNet Convolution (d) Tensor Ring Convolution

Figure 13. More Reproducing Transformation cases.

F. Model Extension

F.1. Graph-out Derivation for HTK2 Convolution

We draw Graph-out Derivation for HTK2 Convolution in Figure 12.

F.2. Example for Inter Interaction in a Tensor Format

As the most widely used methods for CNNs, Xavier and Kaiming initialization usually perform stable and efficient weight generation. However, they usually fail to produce appropriate weights when applied to TCNNs. Without loss of generality, we show an instance by using Xavier to initialize the HTK2 convolution. By generating each HTK2 node $\mathcal{W}^{(k)}$ with the fan-in variance $\frac{1}{k^2\mathbf{c}_{in}}$, we can calculate the final output variance $\sigma^2(\mathcal{Y}_o)$ according to Eq. (5) as

$$\sigma^2(\mathcal{Y}_o) = \boldsymbol{p_a}\varphi\sigma^2(\mathcal{X})\prod_{k=0}^{n-1}\sigma^2(\mathcal{W}^{(k)})\prod_{i=0}^{n-1}\prod_{j=i+1}^{\tau-1}e_{ij} = \varphi k^2\mathbf{c}_{in}\mathbf{r}_0\mathbf{r}_1\sigma^2(\mathcal{X})\frac{1}{(k^2\mathbf{c}_{in})^3} = \frac{\varphi\mathbf{r}_0\mathbf{r}_1}{(k^2\mathbf{c}_{in})^2}\sigma^2(\mathcal{X}), \tag{21}$$

where $\mathbf{r}_0$ and $\mathbf{r}_1$ denote tensor ranks, and φ means the value of a hyperedge. Obviously, through the output variance scale $\frac{\varphi\mathbf{r}_0\mathbf{r}_1}{(k^2\mathbf{c}_{in})^2}$, when $\mathbf{r}_*$ and φ increase, data-flow will explode easily and cause fatal errors in the training process. However, Xavier and Kaiming initialization only consider $\mathbf{c}_{in}$ and k, leading to fail to train HTK2 convolution. Such problems can also be found in the fan-out mode. To resolve the dilemma, we derive a principle (i.e., Eq. (6)) that generalizes over Xavier to fit multi-node TCNNs, not limited to single-node ones (i.e., Vanilla CNNs).

G. More Reproducing Transformation Cases

As shown in Figure 13, we illustrate some additional Reproducing Transformation cases, including Low-rank (LR) convolution, HyperNet convolution, Tensor Train (TT) convolution and Tensor Ring (TR) convolution. From the similar forward and backward tensor graphs, we show that Graph-out initialization can be calculated as exactly the same as the Graph-in.

H. Details of Experiments

All tensorial network experiments are constructed with the code[1] of tednet (Pan et al., 2022).

[1]Source code at `https://github.com/tnbar/tednet`.

H.1. Applicable Validation for Proposition 3.3

This experiment is implemented to validate Proposition 3.3 can be recursively used under loose conditions (namely non-i.i.d input) in practice. Specifically, Proposition 3.3 requires both $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ i.i.d (denoted as ($\boldsymbol{\mathcal{X}}$: i.i.d, $\boldsymbol{\mathcal{Y}}$: i.i.d)), however, in practice, $\boldsymbol{\mathcal{Y}}$ can easily satisfy the i.i.d condition with an initialization, while $\boldsymbol{\mathcal{X}}$ is hard to follow the i.i.d. condition, which is denoted as ($\boldsymbol{\mathcal{X}}$: non-i.i.d, $\boldsymbol{\mathcal{Y}}$: i.i.d). This situation is also faced by the Xavier and Kaiming initialization methods. However, although Xavier and Kaiming both adopt the ($\boldsymbol{\mathcal{X}}$: non-i.i.d, $\boldsymbol{\mathcal{Y}}$: i.i.d) pattern when initializing networks, they work well in a number of real-world tasks.

Here we conduct a experiment to demonstrate data-flow will keep stable under the ($\boldsymbol{\mathcal{X}}$: non-i.i.d, $\boldsymbol{\mathcal{Y}}$: i.i.d) condition, namely, we show that ($\mathbf{X}$: non-i.i.d, $\mathbf{Y}$: i.i.d) will maintain variance scale (i.e., $\prod_{t=0}^{d-1} v_t$ in Proposition 3.3). In detail, we generate $\mathbf{X} \in \mathbb{R}^{32\times 96}$ and 10 matrices, $\mathbf{W}_1 \in \mathbb{R}^{96\times 200}$, $\mathbf{W}_2 \in \mathbb{R}^{200\times 400}$, $\mathbf{W}_3 \in \mathbb{R}^{400\times 600}$, $\mathbf{W}_4 \in \mathbb{R}^{600\times 800}$, $\mathbf{W}_5 \in \mathbb{R}^{800\times 1000}$, $\mathbf{W}_6 \in \mathbb{R}^{1000\times 800}$, $\mathbf{W}_7 \in \mathbb{R}^{800\times 600}$, $\mathbf{W}_8 \in \mathbb{R}^{600\times 400}$, $\mathbf{W}_9 \in \mathbb{R}^{400\times 200}$ and $\mathbf{W}_{10} \in \mathbb{R}^{200\times 100}$. All these 10 matrices are generated by sampling from $N \sim (0, 1)$. In Table 2, Gaussian means $\mathbf{X}$ are generated by sampling from $N \sim (0, 1)$ and Cifar10 denotes that $\mathbf{X}$ is a Cifar10 image. The production sequence is $(\mathbf{X}\mathbf{W}_1) = \mathbf{X} \times \mathbf{W}_1, (\mathbf{X}\mathbf{W}_1\mathbf{W}_2) = (\mathbf{X}\mathbf{W}_1) \times \mathbf{W}_2, \ldots, (\mathbf{X}\mathbf{W}_1\mathbf{W}_2 \ldots \mathbf{W}_{10}) = (\mathbf{X}\mathbf{W}_1\mathbf{W}_2 \ldots) \times \mathbf{W}_{10}$, where $\times$ denotes a matrix production. According to Proposition 3.3, we calculate the variance scale (i.e., a contracting dimension of $\mathbf{W}_t$ for t-th production according to Proposition 3.3) under ($\mathbf{X}$: i.i.d, $\mathbf{Y}$: i.i.d) as the ground-truth. For example, since $\mathbf{X} \times \mathbf{W}_1$ contracts on dimension 96, the scale is calculated as 96. As for the Gaussian and Cifar10 experiments, we run 500 rounds each.

Results show that when $\mathbf{X}$ is chosen as an i.i.d distribution, its scales are almost around the Ground-Truth, which approximate the ($\mathbf{X}$: i.i.d, $\mathbf{Y}$: i.i.d). For the much more tricky case (i.e., Cifar10), since $\mathbf{X}$ is a realistic image and non-i.i.d, the variance of Scale 1 is a little large, but the mean of Scale 1 is not far away from 96. And other scales do not vary a lot and are close to the Ground-Truth. Therefore, the experiments show that ($\mathbf{X}$: non-i.i.d, $\mathbf{Y}$: i.i.d) can approximate ($\mathbf{X}$: i.i.d, $\mathbf{Y}$: i.i.d) and can be applicable recursively in practice.

Table 2. Scale change in propagation.

Data	Scale 1	Scale 2	Scale 3	Scale 4	Scale 5	Scale 6	Scale 7	Scale 8	Scale 9	Scale 10
Ground-Truth	96	200	400	600	800	1000	800	600	400	200
Gaussian	96.4 ± 1.7	201.5 ± 2.5	398.0 ± 3.8	603.4 ± 5.5	790.4 ± 6	1011.0 ± 8.9	805.3 ± 7.9	601.2 ± 7.0	385.6 ± 7	202.0 ± 5.2
Cifar10	122.0 ± 54.8	200.2 ± 5.9	427.1 ± 12.4	588.2 ± 11.3	803.7 ± 12.8	969.3 ± 18.9	799.4 ± 17.5	567.4 ± 18.3	394.7 ± 19.0	193.1 ± 12.1

H.2. Details of Activation Propagation Analysis

We perform this experiment on MNIST through Linear-5. The structure of Linear-5 is shown in Table 3. The mini-batch size is set to 20 and the learning rate is 1e-4. The $\mathbf{r}_*$ ranks are chosen to be 5. The training process lasts 80 epoch and is optimized by Adam. We use one NVIDIA GTX 1080ti GPU for this experiment.

H.3. Random Generator

We construct a random generator by randomly generating 4-8 vertices, 2-3 input $\mathbf{i}_*$ edges, 2-3 output $\mathbf{o}_*$ edges and uncertain number of $\mathbf{r}_*$ edges. Notably, it is guaranteed that each vertex connects to an edge.

H.4. Details of Random Tensor Format Experiment

In this experiment, each layer of Conv-4 can be constructed by a random generator in Appendix H.3. MNIST is used for training and validation. The batch size is set to 128. The learning rate is 1e-4 and the optimizer is Adam. We train each Conv-4 for 20 epochs on a NVIDIA Tesla V100 GPU.

H.5. Details of Experiment on Cifar10

H.5.1. Details of Experiment on Cifar10 in Main Paper

We conduct this experiment based on the All Convolutional Net model (Chang et al., 2020; Springenberg et al., 2015). By replcaing standard convolution in All Convolutional Net with TCNN convolutions, including Low-rank (LR) convolution, Tensor Ring (TR) convolution, Hyper Tucker-2 (HTK2) convolution, and Hyper Odd (HOdd) convolution, we can derive TCNN based All Convolutional Nets, as shown in Table 4. In addition, we also use random generator in Appendix H.3 to

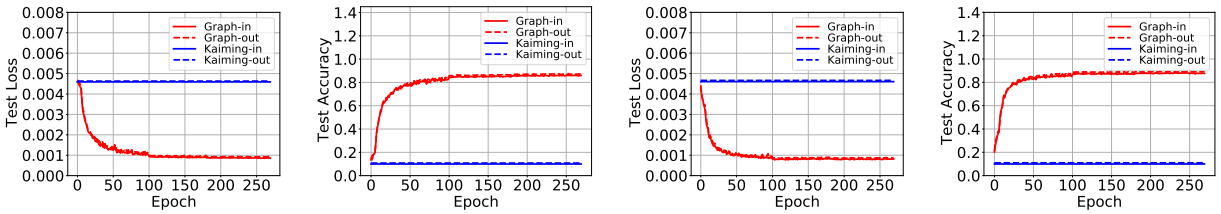
Table 3. Structures of Linear-5 and HOdd-5.

Layer	Linear-5	HOdd–5
linear1	784×500	(28×28)×(20×25)
linear2	500×500	(20×25)×(20×25)
linear3	500×500	(20×25)×(20×25)
linear4	500×500	(20×25)×(20×25)
linear5	500×10	(20×25)×(2×5)

implement 30 different All Convolutional Nets for evaluation. In this experiment, we use Cifar10 dataset for training. The mini-batch size is 128. We choose SGD as the optimizer with learning rate 5e-3, momentum 0.9 and weight decay 5e-4. All $\mathbf{r}_*$ ranks are set to 10. Each network is trained for 270 epochs on an NVIDIA Tesla V100 GPU and the learning rate will be multiplied by a fixed multiplier of 0.2 after 100, 180 and 230 epochs separately. Some results are shown in Figure 14.

Table 4. Architectures of the tensorial All-Conv networks. Window means the convolutional kernel window size. Channels indicate $\mathbf{c}_{in}$ and $\mathbf{c}_{out}$ of a standard convolutional kernel $\mathcal{C} \in \mathbb{R}^{\mathbf{c}_{in} \times \mathbf{c}_{out} \times k \times k}$. The avg pool denotes the average pooling operation.

Layer	Window	Channels	HTK2/Tucker	TR	TT	Low-Rank	HOdd
conv1	3×3	3× 96	(3)× (96)	(3)×(6× 4× 4)	(1× 3× 1)×(6× 4× 4)	(3)× (96)	(1× 3)×(8× 12)
conv2	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv3	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv4	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv5	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv6	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv7	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv8	3×3	96× 96	(96)× (96)	(6× 4× 4)×(6× 4× 4)	(6× 4× 4)×(6× 4× 4)	(96)× (96)	(8× 12)×(8× 12)
conv9	3×3	96× 10 avg pool	(96)× (10) avg pool	(6× 4× 4)×(10) avg pool	(6× 4× 4)×(1× 10× 1) avg pool	(96)× (10) avg pool	(8× 12)×(1× 10) avg pool



(a) HTK2-C4 Loss (b) HTK2-C4 Accuracy (c) HOdd-C4 Loss (d) HOdd-C4 Accuracy

Figure 14. Results on Cifar10.

H.5.2. Comparison with Ad-hoc Initialization

To our best knowledge, our initialization is the first unified method for TCNNs, and there are no other unified initialization methods to be compared. Therefore, we would like to compare with initialization for a specific tensor format (i.e., tensor ring (Wang et al., 2018) and tucker-2 (Elhoushi et al., 2019)), which is hard to extend to other tensor formats. In this experiment, we use Cifar10 as the validation dataset. We adopt All-Conv and ResNet-32 as the base models. Then, tensor ring and tucker-2 are adopted as tensor formats for All-Conv (termed as TR-All-Conv) and ResNet-32 (termed as TK2-ResNet-32), respectively. The experiment is conducted on an NVIDIA Tesla V100 GPU.

TR-All-Conv Setting The mini-batch size is 128. The Optimizer is SGD with learning rate 5e-3, momentum 0.9 and weight decay 5e-4. All $\mathbf{r}_*$ ranks are set to 10.

TK2-ResNet-32 Setting For method of Elhoushi et al. (2019), we use their official code[2] with default setting. For comparison with initial weights, here we decompose the original weight before training. For the proposed graphical method, we set the batch size to 128 and optimizer as SGD with learning rate 0.1, momentum 0.9 and weight decay 5e-4. Shape

[2]Source code at https://github.com/mostafaelhoushi/tensor-decompositions.

of tucker-2 set as the decomposed format from method of Elhoushi et al. (2019). We train each model for 90 epochs with reducing the learning rate through multiplying it by 0.1 after 30 and 60 epochs respectively.

Results are shown in Table 5. Compared the two ad-hoc methods, the proposed graphical performs better than the two ad-hoc initialization. As specially designed methods, the two ad-hoc initialization cannot be applied to other tensor formats. By contrast, our initialization can also be applied to other formats, which indicates a practical advantage of the proposed unified initialization method.

Table 5. Comparison with ad-hoc initialization on Cifar-10.

Initialization (TR-All-Conv)	Accurracy	Initialization (TK2-ResNet-32)	Accurracy
Wang et al. (2018)	0.8307	Elhoushi et al. (2019)	0.8488
Graph-in	0.8308	Graph-in	0.8554
Graph-out	**0.8311**	Graph-out	**0.8654**

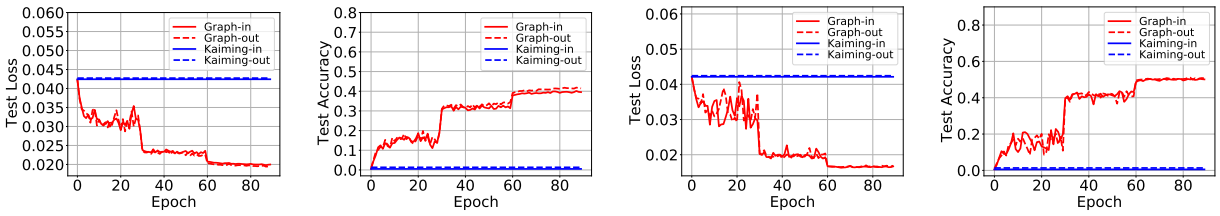
H.5.3. Comparison with Common Tensor Formats

In this section, we validate our initialization method by comparing with CP, Tucker, Tensor Ring (TR) convolution, Tensor Train (TT) convolution, Low-rank (LR) convolution on CIFAR10. To better estimate the performance of our method, we adopt All Convolutional Net structure (Springenberg et al., 2015), which only contains convolutional layers without Batch Normalization (Ioffe & Szegedy, 2015) and residual connection. Optimizer is chosen to be SGD. Reducing normalization tricks and Adam optimization, the training will rely much more on weight initialization. Thus, it will be more clear how our method performs compared with baselines. All $\mathbf{r}_*$ ranks are set to 10 for convenience. The learning rate is set to 5e-3. The experiment is conducted on an NVIDIA Tesla V100 GPU.

In practice, a good initialization should generate weight suitable for diverse models. However, as shown in Table 6, Kaiming-in initialization fails to train all these TCNNs and so does Kaiming-out. On the contrary, our graphical initialization performs well in such a situation. As shown in the figure, our initialization shows an adaptive ability of fitting two completely different TCNN models.

Table 6. Comparison with some common tensor formats on Cifar-10.

Initialization	CP	Tucker	Tensor Ring	Tensor Train	Low-Rank
Kaiming(-in/-out)	0.1	0.1	0.1	0.1	0.1
Graph-in	0.7823	0.7775	0.8308	0.8276	0.8141
Graph-out	0.767	0.7709	0.8311	0.8341	0.8163



(a) HTK2 (φ=4) Loss (b) HTK2 (φ=4) Accuracy (c) HOdd (φ=4) Loss (d) HOdd (φ=4) Accuracy

Figure 15. Results on Tiny-ImageNet.

H.6. Details of Experiment on Tiny-ImageNet

In this experiment, we employ tensorial ResNet (Taki, 2017) to validate the performance of our initialization on Tiny-ImageNet. Details of HTK2 and HOdd based ResNet-50 are shown in Table 7. Values of HTK2 and HOdd hyperedges set to 4. And we also generate some random architectures to demonstrate the ability of our model. We set the batch size to 128 and optimizer as SGD with learning rate 1e-1, momentum 0.9 and weight decay 5e-4. All $\mathbf{r}_*$ ranks are still 10 except HOdd

Table 7. Structures of HTK2 and HOdd based ResNet. k represents convolutional kernel window. • n denotes n same residual blocks. When depth is set to 50, $\{\mathbf{U}_i\}_{i=1}^4$ are 2, 3, 5, 2. And $\{\mathbf{U}_i\}_{i=1}^4$ of 101-depth are set to 2, 3, 22, 2.

Layer	HTK2	HOdd
Pre	$k7$, (3)×(64)	$k7$, (1×3)×(8×8)
Unit 1	$\begin{bmatrix} k1, & (64)\times(64) \\ k3, & (64)\times(64) \\ k1, & (64)\times(256) \\ k1, & (256)\times(64) \\ k3, & (64)\times(64) \\ k1, & (64)\times(256) \end{bmatrix}$ • 1 • $\mathbf{U}_1$	$\begin{bmatrix} k1, & (8\times8)\times(8\times8) \\ k3, & (8\times8)\times(8\times8) \\ k1, & (8\times8)\times(32\times8) \\ k1, & (32\times8)\times(8\times8) \\ k3, & (8\times8)\times(8\times8) \\ k1, & (8\times8)\times(32\times8) \end{bmatrix}$ • 1 • $\mathbf{U}_1$
Unit 2	$\begin{bmatrix} k1, & (256)\times(128) \\ k3, & (128)\times(128) \\ k1, & (128)\times(512) \\ k1, & (512)\times(128) \\ k3, & (128)\times(128) \\ k1, & (128)\times(512) \end{bmatrix}$ • 1 • $\mathbf{U}_2$	$\begin{bmatrix} k1, & (32\times8)\times(8\times16) \\ k3, & (8\times16)\times(8\times16) \\ k1, & (8\times16)\times(32\times16) \\ k1, & (32\times16)\times(8\times16) \\ k3, & (8\times16)\times(8\times16) \\ k1, & (8\times16)\times(32\times16) \end{bmatrix}$ • 1 • $\mathbf{U}_2$
Unit 3	$\begin{bmatrix} k1, & (512)\times(256) \\ k3, & (256)\times(256) \\ k1, & (256)\times(1024) \\ k1, & (1024)\times(256) \\ k3, & (256)\times(256) \\ k1, & (256)\times(1024) \end{bmatrix}$ • 1 • $\mathbf{U}_3$	$\begin{bmatrix} k1, & (32\times16)\times(16\times16) \\ k3, & (16\times16)\times(16\times16) \\ k1, & (16\times16)\times(64\times16) \\ k1, & (64\times16)\times(16\times16) \\ k3, & (16\times16)\times(16\times16) \\ k1, & (16\times16)\times(64\times16) \end{bmatrix}$ • 1 • $\mathbf{U}_3$
Unit 4	$\begin{bmatrix} k1, & (1024)\times(512) \\ k3, & (512)\times(512) \\ k1, & (512)\times(2048) \\ k1, & (2048)\times(512) \\ k3, & (512)\times(512) \\ k1, & (512)\times(2048) \end{bmatrix}$ • 1 • $\mathbf{U}_4$	$\begin{bmatrix} k1, & (64\times16)\times(16\times32) \\ k3, & (16\times32)\times(16\times32) \\ k1, & (16\times32)\times(64\times32) \\ k1, & (64\times32)\times(16\times32) \\ k3, & (16\times32)\times(16\times32) \\ k1, & (16\times32)\times(64\times32) \end{bmatrix}$ • 1 • $\mathbf{U}_4$
	avg pool	avg pool
FC	$(2048)\times(200)$	$(64\times32)\times(10\times20)$

which set to 5 for faster training. We train each model for 90 epochs with reducing the learning rate through multiplying it by 0.1 after 30 and 60 epochs respectively. We use one NVIDIA Tesla V100 GPU for training. Some results are shown in Figure 15.

H.7. Details of Experiment on ImageNet

Lastly, to be more convincing, we validate our initialization on ImageNet.

Tensorial ResNet Setting We construct tensorial ResNet by replacing all convolutional layers of ResNet with tensor layers. In this experiment, we train HOdd (hyperedge set to 4 and $\mathbf{r}_*$ is 5) and randomly generating ResNet for 20 epochs. Structure of HOdd ResNet is similar to Table 7. Depths are set to 50 and 101. Similarly, we use SGD to optimize parameters with learning rate 0.1, momentum 0.9 and weight decay 5e-4. Mini-batch size is set to 512. We use four NVIDIA Tesla V100 GPUs for training.

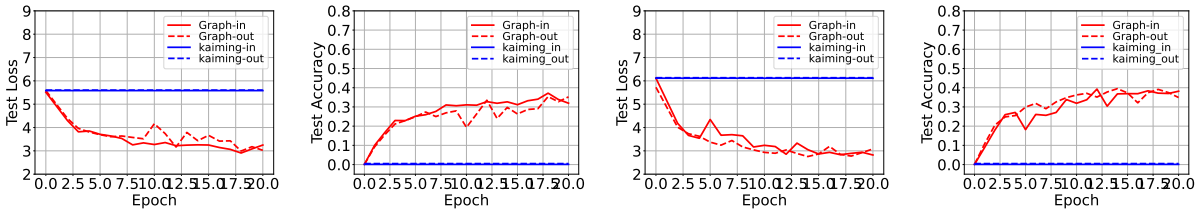
Tensorial gMLP/MLP-Mixer Setting We construct tensorial gMLP/MLP-Mixer by replacing all linear layers of ResNet with tensor layers. In this experiment, we train randomly generating tensor layers for 20 epochs. Training stratey follows setting of Liu et al. (2021). Data augmentation set to AutoAugment. Input resolution of ImageNet is 224×224. Batch-size is 1024. We use Cutmix-Mixup with switch probability 0.5. Cutmix α is 1.0. Mixup α is 0.8. Label smoothing is 0.1. Learning rate is 1e-3 before trainig. Cosine function is adopted as learning rate decay. Optimizer is AdamW with $\epsilon = 1e-6$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. Weight decay is 0.05. We use four NVIDIA Tesla A100 GPUs for training.

Results of HOdd-ResNet, HRand-ResNet, HRand-gMLP and HRand-MLP-Mixer are shown in Figure 16, 17, 18 and 18, respectively. In these figures, our graphical initialization is suitable for all the models while Kaiming initialization fails in all the situations, which demonstrates Graph(-in/-out) algorithm is sufficiently robust and effective. Worth to mention, test loss of tensorial ResNet explodes to NaN (not a number) from the beginning, and the test loss explosion of tensorial gMLP/MLP-Mixer is not severe like so. The difference is caused by the optimizer. ResNet is trained with SGD, and

Table 8. Top-1 accuracy on Cifar10 and Tiny-ImageNet. Rank-Edge Number means the least number of edges only connected to weight vertices in layers. Random-$*$ denotes randomly generating models.

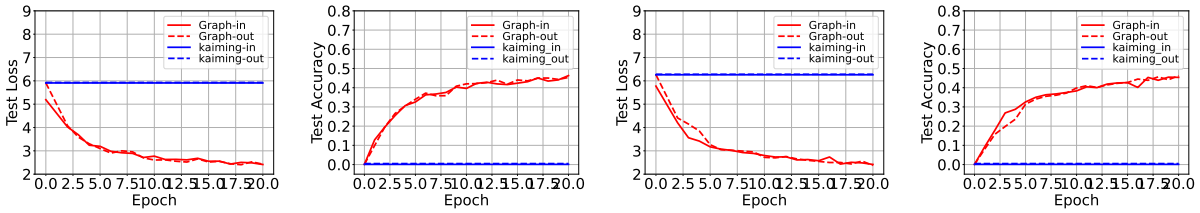
		Cifar10			Tiny-ImageNet		
	Rank-Edge Number	Kaiming (-in/-out)	Graph-in	Graph-out	Kaiming (-in/-out)	Graph-in	Graph-out
Low-Rank	1	0.1	0.8141	0.8163	0.307/0.2776	0.3153	0.3076
Tensor Ring	4	0.1	0.8308	0.8311	0.005	0.2494	0.249
HTK2 (φ=4)	2	0.1	0.8638	0.8705	0.005	0.4014	0.4126
HOdd (φ=4)	14	0.1	0.8826	0.8806	0.005	0.5048	0.5045
Random-1	-	0.1	0.8538	0.8483	0.005	0.4965	0.5015
Random-2	-	0.1	0.8801	0.876	0.005	0.5379	0.5356
Random-3	-	0.1	0.8648	0.863	0.005	0.5475	0.5403
Random-4	-	0.1	0.8789	0.8816	0.005	0.5295	0.5306
Random-5	-	0.1	0.8622	0.8644	0.005	0.5444	0.5428
Random-6	-	0.1	0.8735	0.8721	0.005	0.5452	0.5446
Random-7	-	0.1	0.8601	0.8558	0.005	0.5328	0.5394
Random-8	-	0.1	0.8589	0.8561	0.005	0.5269	0.5291

gMLP/MLP-Mixer is trained with AdamW that has the desirable property of being invariant to the scale of the gradients. However, even through AdamW is so powerful that test loss can return to an acceptable level, it still fails to train a model with the unsuitable initialization, as the test accuracy has not increased. Therefore, our graphical initialization that is adaptive to all the situations, is necessary for TCNNs as a key component of training.



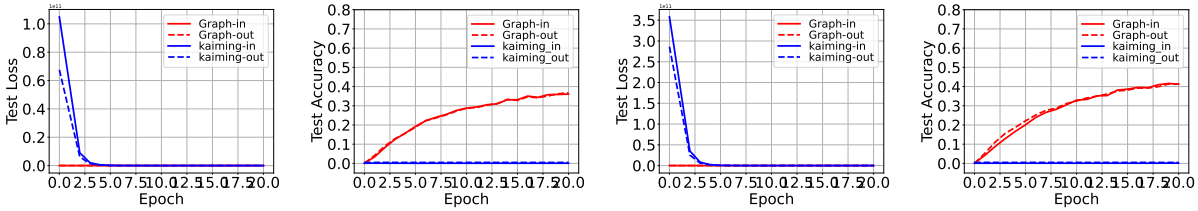
(a) HOdd-RN-50 Loss (b) HOdd-RN-50 Accuracy (c) HOdd-RN-101 Loss (d) HOdd-RN-101 Accuracy

Figure 16. Results of HOdd-RN on ImageNet. RN is short for ResNet.



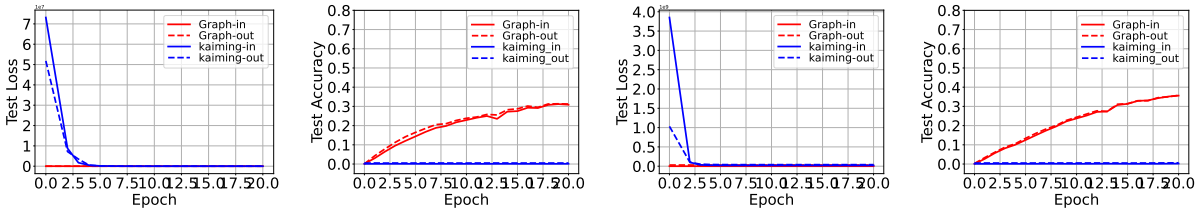
(a) HRand-RN-50 Loss (b) HRand-RN-50 Accuracy (c) HRand-RN-101 Loss (d) HRand-RN-101 Accuracy

Figure 17. Results of HRand-RN on ImageNet. RN is short for ResNet.



(a) Hg-Ti16 Loss (b) Hg-Ti16 Accuracy (c) Hg-S16 Loss (d) Hg-S16 Accuracy

Figure 18. Results of HRand-gMLP (Hg) on ImageNet.



(a) HMM-S16 Loss (b) HMM-S16 Accuracy (c) HMM-B16 Loss (d) HMM-B16 Accuracy

Figure 19. Results of HRand-MLP-Mixer (HMM) on ImageNet.