ABSTRACT

Title of Dissertation: SPECTRAL METHODS FOR NEURAL NETWORK DESIGNS

Jiahao Su
Doctor of Philosophy, 2022

Dissertation Directed by: Assistant Professor Furong Huang
Department of Computer Science

Neural networks are general-purpose function approximators. Given a problem, engineers or scientists select a hypothesis space of functions with specific properties by designing the network architecture. However, mainstream designs are often ad-hoc, which could suffer from numerous undesired properties. Most prominently, the network architectures are gigantic, where most parameters are redundant while consuming computational resources. Furthermore, the learned networks are sensitive to adversarial perturbation and tend to underestimate the predictive uncertainty. We aim to understand and address these problems using spectral methods — while these undesired properties are hard to interpret from network parameters in the original domain, we could establish their relationship when we represent the parameters in a spectral domain. These relationships allow us to design networks with certified properties via the spectral representation of parameters.

SPECTRAL METHODS FOR NEURAL NETWORK DESIGNS

by

Jiahao Su

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Acknowledgments

Throughout my doctoral reseasrch, I have received a great deal of support and assistance. I owe my gratitude to all the people who have made this dissertation possible.

First and foremost, I would like to thank my advisor, Professor Furong Huang, who helped me in my most difficult time and guided me through my doctoral study. She has given me invaluable opportunities to work on exciting and challenging problems over the last five years. Her research passion has inspired and will continue to inspire me in my career. I am very fortunate and proud to be my advisor's first Ph.D. graduate.

I would also like to thank my intern manager, Professor Anima Anandkumar; my intern mentor and long-term collaborator, Dr. Wonmin Byeon. It is impossible to complete my dissertation without their deep insights into my research.

I am deeply grateful to all my collaborators and colleagues at the University of Maryland. There were memorable moments when I started my doctoral research working with Jingling Li, Xuchen You, and Yanchao Sun. Milan Cvitkovic provided me the initial idea for my first publication at conferences. My collaboration with Shiqi Wang, Xiaoyu Liu, Tahseen Rabbani, Chenghao Deng, and Evan Wang has been highly fruitful.

I owe my deepest thanks to my friends, especially those pursuing their doctoral degrees — their encouragement is crucial for me to finish this dissertation. I would like to express my gratitude to my high school friends Xinshi Chen, Zelun Luo, Siqi Yang, Boyang Zhang, Haici Zhang, Rui Zhang, Yuliang Zou; and my college friends Jinghui Chen, Yang Chen, Shupeng Gui, Zecheng He, Xu Peng, Pan Zhong.

Lastly, I would like to thank my parents for their unconditioned love and support. It would not be possible to start my Ph.D. study without their financial support. Likewise, it would not be possible to continue my research career without their encouragement. Unfortunately, due to the COVID-19 pandemic, I have not seen my parents in the last three years — I miss you, dad and mom.

Table of Contents

Chapter 1: Introduction

Deep learning has achieved unprecedented performance in a wide range of applications, and most notably computer vision [1, 2] and natural language processing [3, 4]. The most crucial factor in these successes is the growing depth of modern neural networks [5, 6], along with enormous effort in developing techniques to learn deep models [7, 8]. However, mainstream architectures and techniques are often ad-hoc, leading to numerous mysterious properties, many of which are undesirable. For example, deep networks typically have more parameters than training examples, where most parameters are redundant while consuming computational resources [9]. Furthermore, the learned models are sensitive to adversarial perturbations [10] and tend to make overconfident predictions [11].

The lack of principled methods attribute to the implicit relationship between network parameters and properties — It is hard to interpret how a parameter impacts the overall properties. In this dissertation, we develop principled network architectures toward addressing this problem using spectral methods — while it is hard to relate network properties to parameters in the original space, we can make the relationship explicit in a carefully chosen spectral domain.

In this chapter, we briefly review the undesirable properties of modern neural networks, including over-parametrization, uncertainty miscalibration, and sensitivity to perturbations. Then we discuss the equivalence between network architecture in deep learning and hypothesis set in machine learning. Our goal of architecture designs is to eliminate undesirable hypotheses from the hypothesis set. The following chapters present how to design network architectures (i.e., choose hypothesis sets) to avoid these undesirable properties.

Network architectures and hypothesis sets. Abstractly, a deep network is an adaptive function that approximates a certain mapping

$$\mathbf{y} = \mathsf{NN}(\mathbf{x}; \theta), \tag{1.1}$$

where $\mathbf{x}, \mathbf{y}$ are the input/output of the network NN, and θ is the network parameters. There are two steps to obtain a particular function.

(1) In the first step, we pick a *network architecture*, which defines the space of all possible mappings:

$$\mathcal{H} = \{\mathsf{NN}(\cdot, \theta)\}_{\theta \in \Theta} \tag{1.2}$$

where $\mathcal{H}$ is known as the hypothesis set, and Θ is the parameter space. When the architecture is known from the context, $\mathcal{H}$ and Θ are two equivalent notions.

(2) In the second step, we choose one mapping $h^\star$ among all candidates $\mathcal{H}$ by minimizing a loss function $\mathcal{L}$:

$$\theta^\star = \arg\min_{\theta \in \Theta} \mathcal{L}(\mathcal{D}, \theta), \tag{1.3}$$

where $\mathcal{D}$ is a dataset. Equivalently, we have $h^\star = \mathsf{NN}(\cdot, \theta^\star)$.

Our research focuses on the first step — we try to choose the hypothesis set $\mathcal{H}$ such

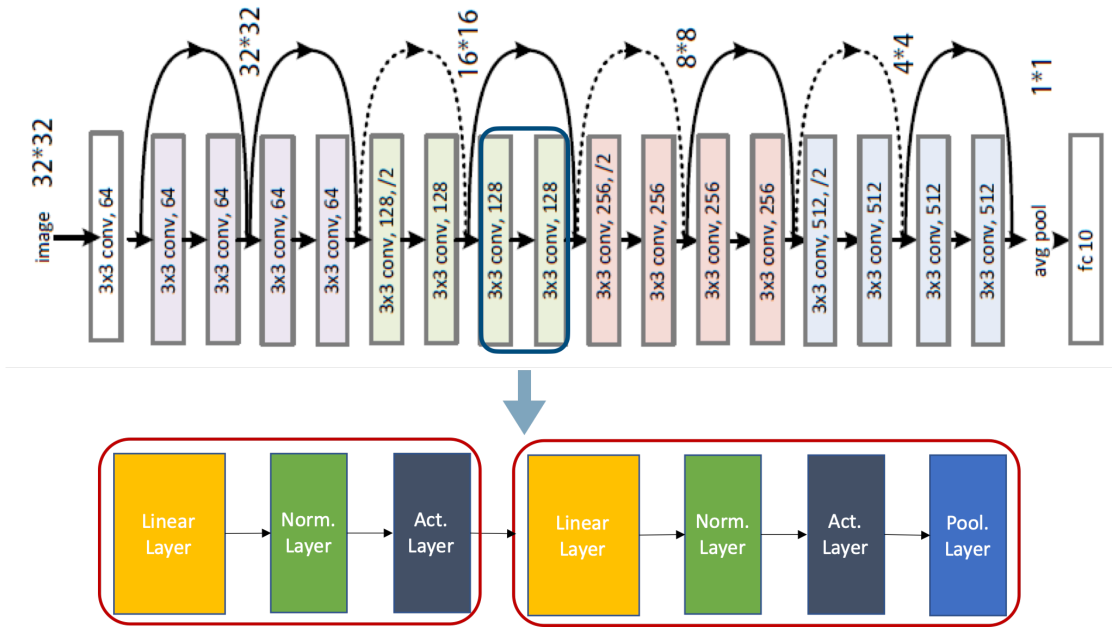


Figure 1.1: **Architecture of a deep residual network [5].**

that we can readily find a desirable model h. Since the relationships between the parameter space and the network properties are generally implicit, we investigate spectral methods $\Theta \to \mathcal{F}$ such that in a transformed space $\mathcal{F}$, the relationships become explicit.

In practice, deep networks' building blocks are usually simple (see Figure 1.1). Typically, a deep network is a cascade of multiple blocks, where each block consists of interleaving linear and nonlinear layers. Despite its simplicity, a composition of simple blocks could lead to many undesirable global properties, including over-parametrization, miscalibrated uncertainties, and sensitivity to perturbations.

Over-parameterization. The increasing performance of modern neural network accredits to the evolving deep architectures [1, 2, 5] (see Figure 1.2). However, their parameters grow with the depth, which in turn leads to an explosive growth of computation, as shown in Figure 1.3. The high demand for computational resources makes these models very difficult to deploy on constrained devices like smartphones or embedded systems.

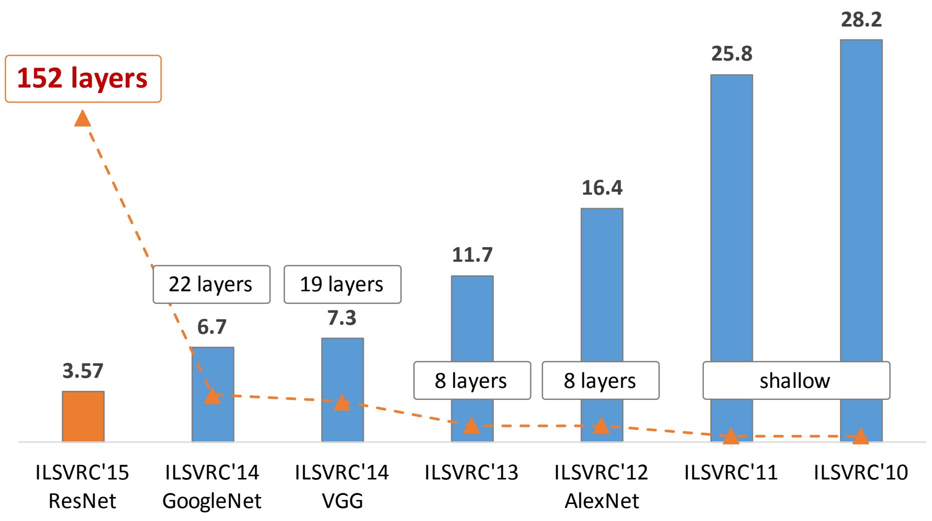


Figure 1.2: **Increasing performance and growing depth** of modern neural networks [5].

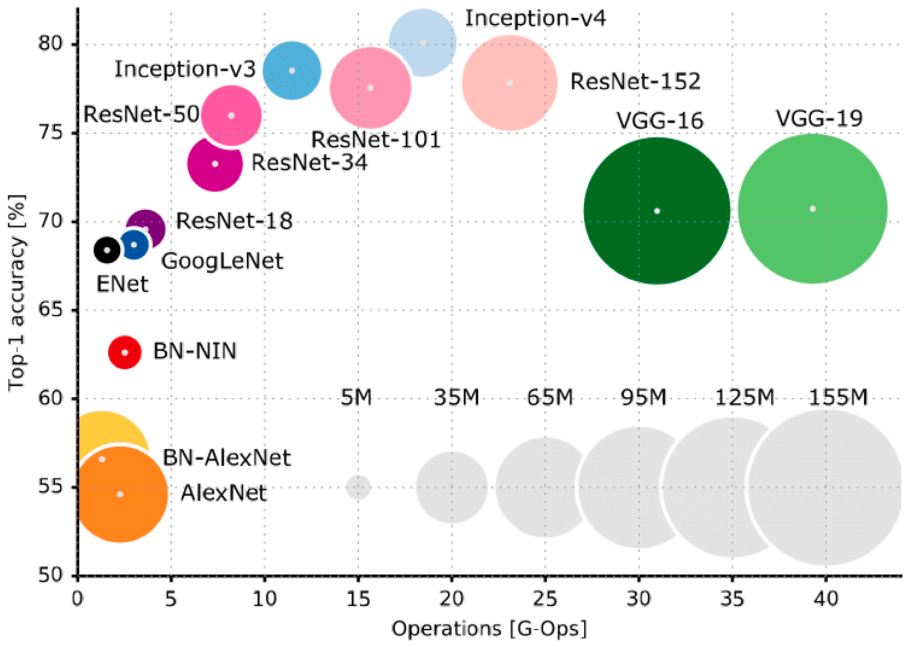
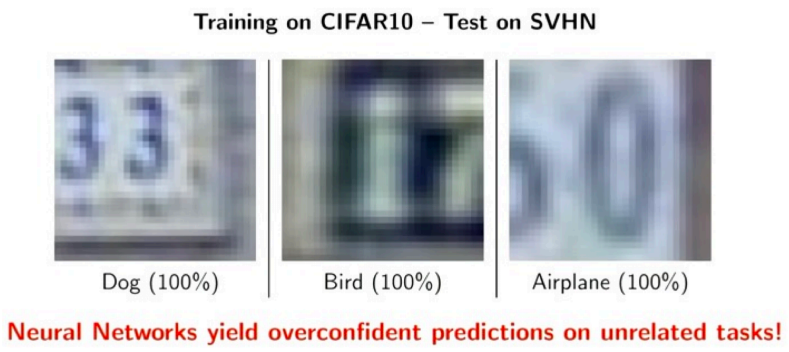
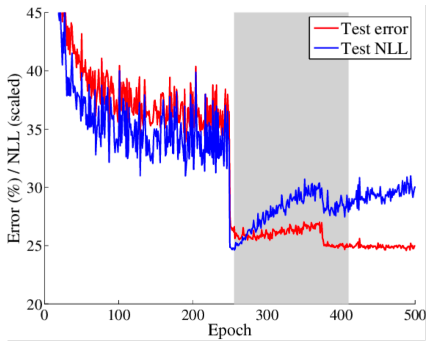


Figure 1.3: **Increasing computational complexity** of modern neural networks [12].

Uncertainty miscalibration. Calibrated prediction, i.e., predicting probability estimates of the true correctness likelihood, is important in uncertainty-sensitive application such as healthcare. However, deep networks are typically poorly calibrated [11]. One mysterious property of these deep networks is that they tend to give over-confident predictions even for out-of-distribution data (c.f. Figure 1.4a). From a different perspective, the negative log-likelihood, which measures the level of uncertainty, is more prone to overfitting than predictive error (c.f. Figure 1.4b).


Training on CIFAR10 – Test on SVHN
Dog (100%)
Bird (100%)
Airplane (100%)
Neural Networks yield overconfident predictions on unrelated tasks!

(a) Overconfident predictions for out-of-distribution data.



(b) Easy overfitting of negative log-likelihood (NLL).

Figure 1.4: **Uncertainty miscalibration** of modern neural networks [11].

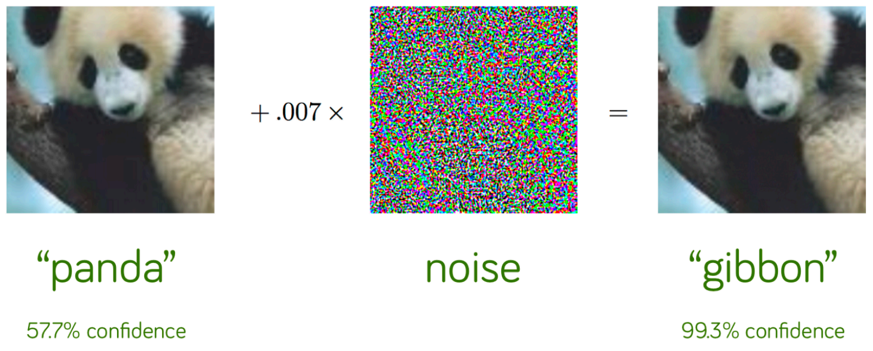


Figure 1.5: Modern neural network is **sensitive to adversarial perturbation** [10].

Sensitivity to perturbations. Deep networks are susceptible to small perturbations – an imperceptible perturbation to the input can flip the prediction completely. The perturbation can be artificial noise added to the input as shown in Figure 1.5 [10], or geometric transforms such as translation, rotation, and scaling as shown in Figure 1.6 [13]. This problem raises concerns about to use of deep networks in security-sensitive applications.

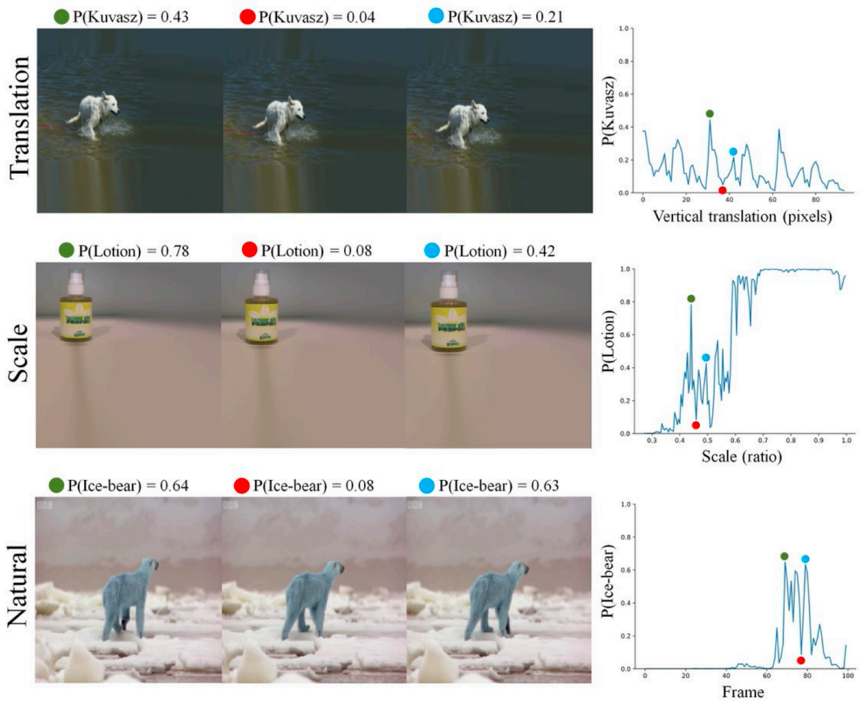


Figure 1.6: Modern neural network is **sensitive to geometric perturbations** [13].

Organization of the dissertation. The rest of the dissertation is organized as follows.

In Chapter 2 and Chapter 3, we introduce tensor representations, which allow for concise descriptions for multilinear operations. In Chapter 2, we present a framework of compact layer designs based on tensor representations. With this framework, we derive a family of compact yet expressive convolutional networks, which maintain high performance after significantly reducing model parameters. In Chapter 3, we develop an automatic library, which can build and efficiently train tensorial neural networks (whose layers adopt a tensor representation). In Chapter 4, we use tensor representations to develop the first higher-order recurrent network for spatio-temporal learning. This model achieves state-of-the-art performance among recent approaches while with the fewest parameters.

In Chapter 5, we introduce Bayesian neural networks, which explicitly characterize uncertainties by learning the posterior distribution of the network parameters. We then propose an efficient sampling-free approach to learn Bayesian quantized networks based on tensor operations — our method nicely bridges the learning of calibrated Bayesian models and compact quantized networks.

In Chapter 6 and Chapter 7, we show that the convolutional layer in neural networks is equivalent to the MIMO filter bank in signal processing. Then, in Chapter 6, we introduce ARMA layers based on IIR filter banks for dense prediction networks such that we can expand their receptive fields economically. Next, in Chapter 7, we will propose orthogonal convolutional layers for robust networks based on paraunitary filter banks such that we can ensure exact orthogonality easily.

We conclude the dissertation in Chapter 8 and discuss potential future works.

Chapter 2: Compact Architecture Designs by Tensor Representations

2.1 Overview

Modern neural networks [1, 2, 6, 14, 15, 16] achieve unprecedented performance on many difficult learning problems at the cost of requiring excessive model parameters for deeper and wider architectures. The vast number of model parameters is a practical obstacle to deploying neural networks on constrained devices, such as smartphones and IoT devices. Thus a fundamental problem in deep learning is to design neural networks with compact architectures that maintain expressive power comparable to large models. Two complementary approaches are common for this purpose: one compresses pre-trained models while preserving their performance as much as possible [9]; the other aims to develop compact neural architectures such as inception modules [16], interleaved group convolutions [17], and bottleneck blocks [15, 18]. Since linear layers (i.e., fully-connected and convolutional layers) comprise almost all parameters and computation, the common goal of both approaches is to reduce the expense by the linear operations.

Motivated by the tensor decomposition of linear layers [19, 20, 21], we propose a framework of tensorial layers that outlines the design space of low-rank factorization — the framework simultaneously allows compression of pre-trained models and exploration of better network architectures. Our proposed tensorial layers extend the linear operations of matrix multiplications (in fully-connected layers) and multi-channel convolutions (in convolutional layers) to multilinear operations with multiple kernels. To characterize these layers, we introduce a novel suite of generalized tensor algebra that extends linear operations on low-order tensors to multilinear ones on higher-order tensors (cf. Section 2.3).

We name a neural network composed of tensorial layers as a *tensorial neural network* (TNN), which by definition generalizes the traditional neural network (NN) — if we restrict the multilinear operations in tensorial layers to matrix multiplications or multi-channel convolutions, the TNN reduces to a traditional NN. Unlike traditional NNs that may flatten the data into low-order tensors (e.g., from videos to frames), TNNs allow for data with arbitrary order. Quite the opposite, TNNs deliberately reshape the data into higher-order tensors and use higher-order weight kernels in each layer. In this higher-order space, TNNs can achieve high expressive power with a smaller number of parameters.

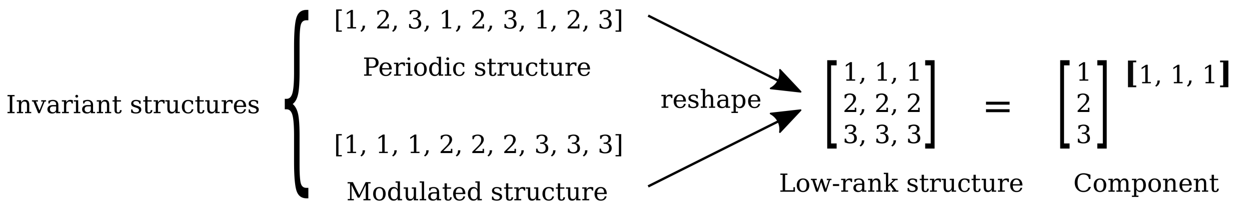


Figure 2.1: **A toy example of invariant structures.** The periodic and modulated structures are exposed by exploiting the low rank structure in the reshaped matrix.

To understand the benefit of higher-order space, we illustrate with a toy example in Figure 2.1. Consider a vector with periodic structure $[1, 2, 3, 1, 2, 3, 1, 2, 3]$ or with modulated structure $[1, 1, 1, 2, 2, 2, 3, 3, 3]$, representing the vector naively requires 9 parameters, which by itself cannot be further compressed by factorization. However, if we reshape the vector into a higher-order object, for instance, a matrix $[1, 1, 1; 2, 2, 2; 3, 3, 3]$. Since all columns of this matrix are the same, we can decompose the rank-1 matrix into an outer product of two vectors without losing information. Therefore, only 6 parameters are needed to represent the original length-9 vector. Intuitively, it is easier to represent higher-order tensors in a factorized form than low-order ones.

To use TNNs in practice, we need to address both *prediction* and *learning* problems in tensorial layers. **(1)** Prediction with a TNN is similar to a traditional NN: its input passes through all layers in a feedforward manner. In a TNN, each layer involves a generalized tensor operation between the *higher-order* input and multiple weight kernels, followed by an activation function such as ReLU. **(2)** To provide a practical solution to the learning problem, we derive efficient backpropagation rules [22] for a broad family of tensorial layers using the newly introduced tensor algebra. We can then efficiently learn TNNs using first-order optimization methods such as stochastic gradient descent (SGD).

Although we could build and train TNNs from scratch, we can also use them to compress pre-trained NNs, as tensorial layers naturally identify both *low-rank* and *invariant* structures in the original kernels of the linear layers (Figure 2.1). Given a pre-trained NN $g^q \in \mathcal{G}^q$ with q parameters, we may compress it to a TNN $h^p \in \mathcal{H}^p$ with p parameters as depicted in Figure 2.7. This process involves two steps: **(1) data tensorization**: reshaping the input into a higher-order tensor; and **(2) knowledge distillation**: mapping a NN to a TNN, using layer-wise data reconstruction.

We demonstrate the expressive power of TNNs by conducting experiments on several benchmark image classification datasets. Our algorithm compresses ResNet-32 on the CIFAR-10 dataset by 10$\times$ with degradation of only 1.92% (achieving an accuracy of 91.28%). Experiments on LeNet-5, VGG, ResNet, and Wide-ResNet consistently verify that our tensorial neural networks outperform the state-of-the-art low-rank architectures under the same compression rate (with 5% test accuracy improvement on CIFAR-10 using sequential knowledge distillation and ImageNet when trained from scratch).

Contributions. In summary, we make the following contributions:

1. We propose a framework of *tensorial layers*, which extends special linear operations in traditional neural networks to general multilinear operations. This results in *tensorial neural networks* (TNNs) with compact architecture designs in higher-order space.

2. We introduce a system of *generalized tensor algebra*, with which we derive efficient prediction and learning in TNNs. In particular, we are the first to derive and analyze backpropagation for generalized tensor operations.

3. We develop an effective algorithm to compress pre-trained models into TNNs, exploiting low-rank and invariant structures in the parameter space.

4. We provide interpretations of famous network architectures with our proposed tensorial layers, explaining why these famous architectures are empirically successful. Our framework provides a principled way to design structured weight matrices/tensors (see examples in Figures 2.8 and 2.9).

2.2 Related Works

Tensor networks are widely used in quantum physics [23], numerical analysis [24], and machine learning [25, 26]. [27, 28] use tensor networks to establish the expressive power of convolutional and recurrent neural networks. Recently, [29] combines tensor networks with genetic algorithms to search for efficient layer designs. Unlike our work, the search space in [29] only includes low-order tensors. Moreover, their method does not apply knowledge distillation to pre-trained models to produce more compact architectures.

Model compression of neural networks. Existing approaches for neural network compression can be roughly grouped into the following categories: *low-rank factorization*, *design of compact filters*, *knowledge distillation*, as well as *pruning, quantization, and encoding.*

1. *Low-rank factorization.* Various factorizations have been proposed to reduce the number of parameters in linear layers. Pioneering works propose to flattening/unfolding the parameters in convolutional layers into matrices (known as *matricization*), followed by dictionary learning or matrix decomposition [30, 31, 32]. Subsequently, [20] and [21] show that it is possible to compress these parameter structures directly using tensor decompositions (e.g., CP or Tucker decomposition [33]). The groundbreaking works [19, 34] demonstrate that the low-order parameter structures can be efficiently compressed via tensor-train decomposition [35] by first reshaping the structures into a higher-order tensor. This idea is later extended in two directions: tensor-train decomposition is used to compress LSTM/GRU layers in recurrent neural networks [36], higher-order recurrent neural networks [37, 38], and 3D convolutional layers [39]; other decompositions are also explored for better compression, such as tensor-ring decomposition [40] and block-term decomposition [41].

2. *Pruning, quantization, and encoding.* The pioneering work [42] proposes a three-step pipeline to compress a pre-trained model by pruning uninformative connections, quantizing remaining weights, and encoding discretized parameters. These ideas are complementary to low-rank factorization — [43] combines pruning with low-rank

factorization, and [44] combines quantization with low-rank factorization.

3. *Knowledge distillation.* This process aims to transfer information from a pre-trained teacher network to a smaller student network. [45, 46] propose to train the student network with the teacher network's logits (the vector before the softmax layer). [47] extends this idea so that the outputs from both networks match *at each layer*, with an affine transformation.

Our approach synergizes two of the above methods: **(1)** it uses knowledge distillation to project a pre-trained network to the set of TNNs with low-rank tensor structures, and **(2)** it exploits low-rank structures, which naturally correspond to compact architecture designs (structured connections) that allow for efficient evaluation using generalized tensor operations. Since other compression methods such as pruning and quantization complement our approach, they may further improve performance augmenting our approach.

2.3 Introduction to Tensor Representations

Notations. Lower case letters (e.g., $\boldsymbol{v}$), upper case letters (e.g., $\boldsymbol{M}$), and calligraphic letters (e.g., $\mathcal{T}$) are used to denote vectors, matrices, and multi-dimensional arrays (tensors), respectively. We say that the array $\mathcal{T} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$ is a *m-order* tensor. Furthermore, the k^{th} coordinate of the entries of $\mathcal{T}$ corresponds to the k^{th} *mode* of $\mathcal{T}$, and I_k is referred to as the *dimension* of $\mathcal{T}$ along mode-k. By fixing all indices of $\mathcal{T}$, except that corresponding to mode-k, we obtain the mode-k *fibers* of $\mathcal{T}$, where the vector $\mathcal{T}_{i_0, \cdots, i_{k-1}, :, i_{k+1}, \cdots, i_{m-1}} \in \mathbb{R}^{I_k}$ denotes the mode-k fiber of $\mathcal{T}$ indexed by $(i_0, \cdots, i_{k-1}, i_{k+1}, \cdots, i_{m-1})$.

Operator	Notation	Definition
mode-(I_k, J_l) Tensor Contraction	$\mathcal{T}^{(1)} = \mathcal{X} \times_{J_l}^{I_k} \mathcal{Y}$	$\mathcal{T}^{(1)}_{i_0,\cdots,i_{k-1},i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}}$ $= \left\langle \mathcal{X}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}}, \mathcal{Y}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}} \right\rangle$ Inner product of mode-k fiber of $\mathcal{X}$ and mode-l fiber of $\mathcal{Y}$
mode-(I_k, J_l) Tensor Convolution	$\mathcal{T}^{(2)} = \mathcal{X} *_{J_l}^{I_k} \mathcal{Y}$	$\mathcal{T}^{(2)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}}$ $= \mathcal{X}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}} * \mathcal{Y}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}}$ Convolution of mode-k fiber of $\mathcal{X}$ and mode-l fiber of $\mathcal{Y}$
mode-(I_k, J_l) Tensor Batch Product	$\mathcal{T}^{(3)} = \mathcal{X} \otimes_{J_l}^{I_k} \mathcal{Y}$	$\mathcal{T}^{(3)}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}}$ $= \mathcal{X}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1}} \ \mathcal{Y}_{j_0,\cdots,j_{l-1},r,j_{l+1},\cdots,j_{n-1}}$ Hadamard product of mode-k fiber of $\mathcal{X}$ and mode-l fiber of $\mathcal{Y}$

Table 2.1: **Summary of tensor operations**. In this table, $\mathcal{X} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}$, $\mathcal{Y} \in \mathbb{R}^{J_0 \times \cdots \times J_{n-1}}$ are input tensors. $\mathcal{T}^{(1)}$, $\mathcal{T}^{(2)}$, $\mathcal{T}^{(3)}$ are the outputs of Mode-(I_k, J_l) tensor contraction, convolution and batch product respectively. Note that both mode-(I_k, J_l) tensor contraction and mode-(I_k, J_l) tensor batch product are legal only if $I_k = J_l$. As a result, $\mathcal{T}^{(1)}$ is an $(m+n-2)$-order tensor, and $\mathcal{T}^{(2)}$, $\mathcal{T}^{(3)}$ are $(m+n-1)$-order tensors.

Tensor diagrams. In Figure 2.2, we introduce *tensor diagrams*, graphical representations of multi-dimensional arrays [23, 24]. In tensor diagrams, each *node* represents an array (scalar, vector, matrix, or higher-order tensor), with its order denoted by the number of *legs* extending from the node. Each leg corresponds to one tensor mode, whose dimension is associated with a positive integer. Notice that tensor diagrams are *ordering-agnostic*, e.g., a matrix $\boldsymbol{M} \in \mathbb{R}^{I \times J}$ and its transpose $\boldsymbol{M}^\top \in \mathbb{R}^{J \times I}$ have the same diagram.

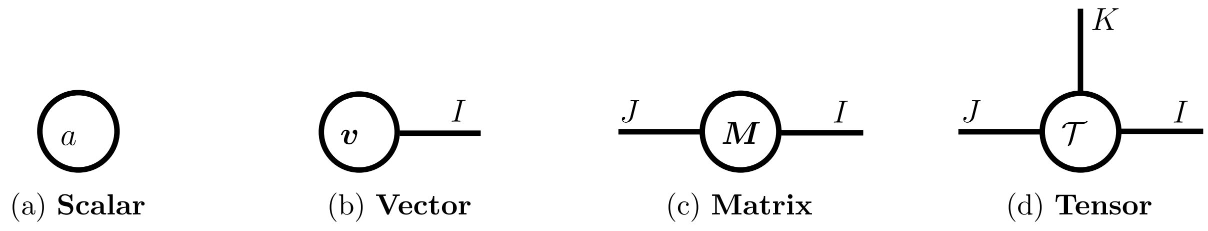
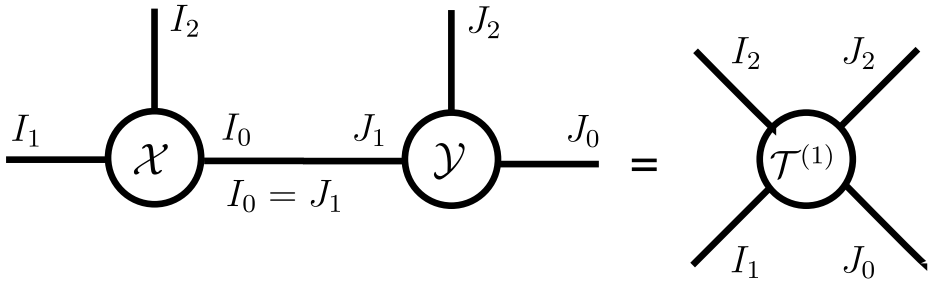
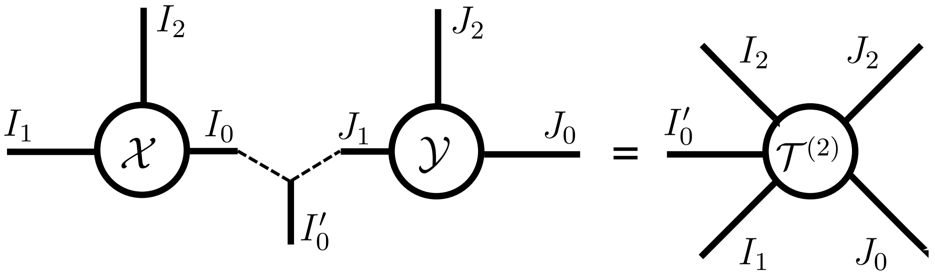


Figure 2.2: **Tensor diagrams** of a scalar $a \in \mathbb{R}$, a vector $\boldsymbol{v} \in \mathbb{R}^I$, a matrix $\boldsymbol{M} \in \mathbb{R}^{I \times J}$, and a 3rd-order tensor $\mathcal{T} \in \mathbb{R}^{I \times J \times K}$.

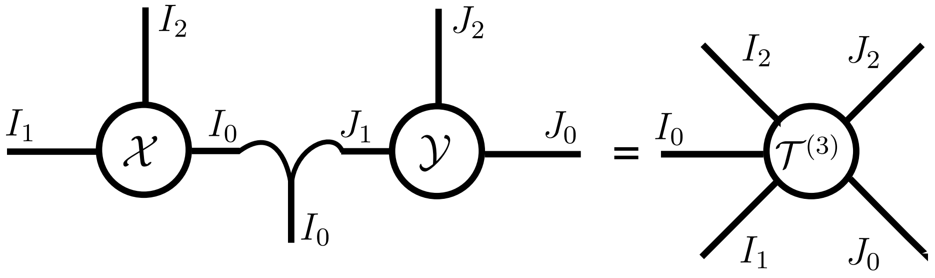
Primitives of tensor operations. In Figure 2.3 and Table 2.1, we introduce the primitives for *generalized tensor operations* on arbitrary-order *tensors*, extending the matrix/vector operations. In tensor diagrams, an operation is a (hyper-)edge that links the legs from



(a) **Mode-(I_0, J_1) tensor contraction**
$\mathcal{X} \times_{J_1}^{I_0} \mathcal{Y} \rightarrow \mathcal{T}^{(1)}$: $\mathcal{T}^{(1)}_{i_1,i_2,j_0,j_2} = \sum_r \mathcal{X}_{r,i_1,i_2} \mathcal{Y}_{j_0,r,j_2}$



(b) **Mode-(I_0, J_1) tensor convolution**
$\mathcal{X} *_{J_1}^{I_0} \mathcal{Y} \rightarrow \mathcal{T}^{(2)}$: $\mathcal{T}^{(2)}_{:,i_1,i_2,j_0,j_2} = \mathcal{X}_{:,i_1,i_2} * \mathcal{Y}_{j_0,:,j_2}$



(c) **Mode-(I_0, J_1) tensor batch product**
$\mathcal{X} \otimes_{J_1}^{I_0} \mathcal{Y} \rightarrow \mathcal{T}^{(3)}$: $\mathcal{T}^{(3)}_{r,i_1,i_2,j_0,j_2} = \mathcal{X}_{r,i_1,i_2} \mathcal{Y}_{j_0,r,j_2}$

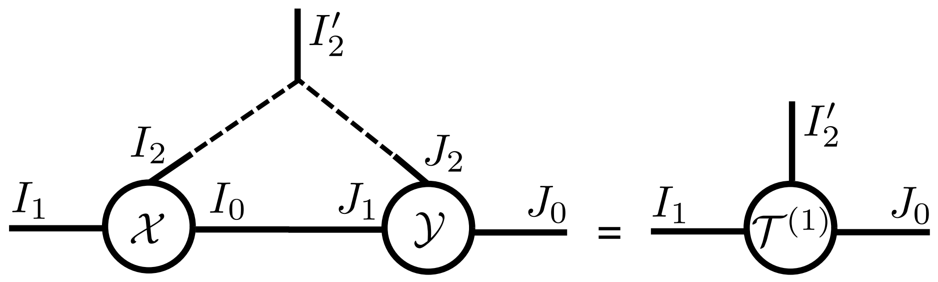
Figure 2.3: **Primitives of generalized tensor operations.** In the diagrams, $\mathcal{X} \in \mathbb{R}^{I_0 \times I_1 \times I_2}$, $\mathcal{Y} \in \mathbb{R}^{J_0 \times J_1 \times J_2}$ are input tensors, and $\mathcal{T}^{(1)} \in \mathbb{R}^{I_1 \times I_2 \times J_0 \times J_2}$, $\mathcal{T}^{(2)} \in \mathbb{R}^{I'_0 \times I_1 \times I_2 \times J_0 \times J_2}$, $\mathcal{T}^{(3)} \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times J_0 \times J_2}$ are output tensors of corresponding operations. Similar definitions apply to general mode-(I_k, J_l) tensor operations.

input tensors, where the edge shape denotes the type of operation: a solid line stands for *tensor contraction*, a dashed line represents *tensor convolution*, and a curved line is for *tensor batch product*. We illustrate these three operations with examples of 3rd-order tensors $\mathcal{X}$ and $\mathcal{Y}$ in Figure 2.3, and define them on higher-order tensors in Table 2.1.

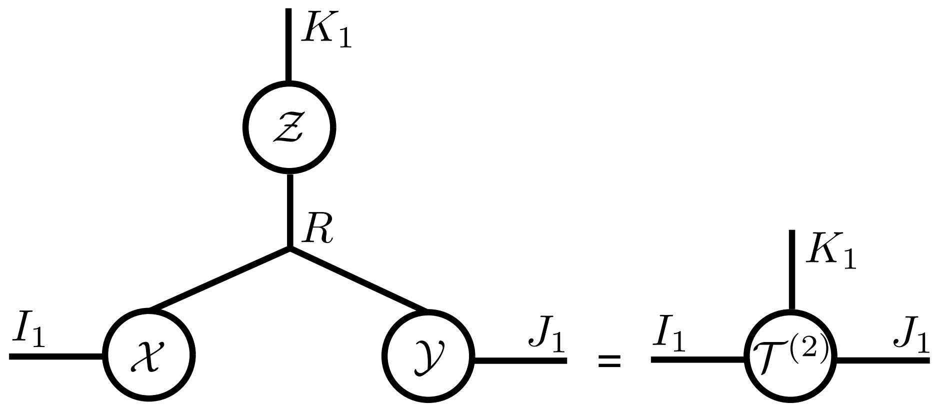
Generalized tensor operations. Generalized tensor operations take two or more tensors as inputs and carry out one or more primitive operations on those tensors. In Figure 2.4, we illustrate three non-primitive generalized tensor operations. We refer to the primitive tensor operations in Figure 2.3 as *single-edge-double-node* operations; similarly, the three generalized tensor operations in Figure 2.4 are called *multi-edge-double-node*, *single-edge-multi-node*, and *multi-edge-multi-node* operations, respectively. Given a generalized tensor operation formed from more than one primitive operation, we may evaluate the primitives in any order to obtain the same result. However, evaluating the primitives in one order may require substantially more floating-point operations (FLOPs) than in another. While it is NP-hard to obtain the best order (that requires the fewest FLOPs) [48], an exhaustive search is practical if the number of input tensors is small [49].

2.4 Tensorial Neural Networks (TNNs)

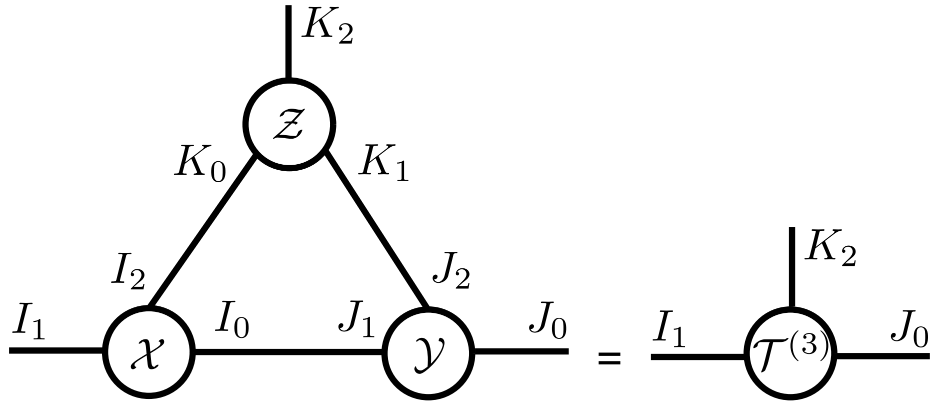
In this section, we introduce Tensorial Neural Networks, a type of neural network whose layers (called *tensorial layers*) are tensor networks. Tensorial layers generalize traditional fully-connected/convolutional layers, as the transformations of these layers are themselves primitive/generalized tensor operations. For example, a fully-connected layer, which involves a matrix-vector product, is equivalent to a contraction (cf. Figure 2.3a), and



(a) **Multi-edges-double-nodes operation**
$\mathcal{X}(\,*^{I_2}_{J_2}\circ\times^{I_0}_{J_1}\,)\mathcal{Y}\to\mathcal{T}^{(1)}$: $\mathcal{T}^{(1)}_{i_1,i_2,j_0,:}=\sum_r \mathcal{X}_{r,i_1,:}*\mathcal{Y}_{j_0,r,:}$



(b) **Single-edge-multi-nodes operation**
$\mathbf{1}\times^R_R(\mathcal{X}\otimes^R_R\mathcal{Y}\otimes^R_R\mathcal{Z})\to\mathcal{T}^{(2)}$: $\mathcal{T}^{(2)}_{i_1,j_1,k_1}=\sum_r \mathcal{X}_{r,i_1}\mathcal{Y}_{r,j_1}\mathcal{Z}_{r,k_1}$



(c) **Multi-edges-multi-nodes operation**
$(\mathcal{X}\times^{I_0}_{J_1}\mathcal{Y})(\,\times^{I_2}_{K_0}\circ\times^{J_2}_{K_1}\,)\mathcal{Z}\to\mathcal{T}^{(3)}$: $\mathcal{T}^{(3)}_{i_1,j_0,k_2}=\sum_{\{r_l\}}\mathcal{X}_{r_0,i_1,r_1}\mathcal{Y}_{j_0,r_0,r_2}\mathcal{Z}_{r_1,r_2,k_2}$

Figure 2.4: **Examples of generalized tensor operations.** The operation in **(a)** is known as a 1D-convolutional layer, and operations **(b) and (c)** are known as CP decomposition and Tensor-Ring decomposition, respectively.

we will see that a convolutional layer is equivalent to a generalized tensor operation (cf. Figure 2.5a). Our primary focus is on tensorial layers that extend the traditional convolutional layer — since a fully-connected layer is simply a 1×1 convolutional layer.

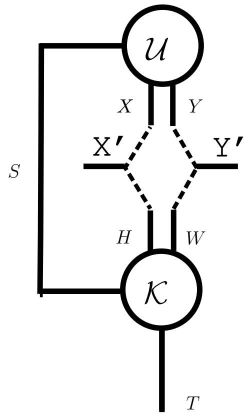
2.4.1 Tensorial Layers versus Traditional Layers

Each convolutional layer in a convolutional neural network (CNN) is given by a compound operation applied to a 3rd-order input tensor and a 4th-order weight tensor (cf. Figure 2.5a). In contrast, each tensorial layer in a TNN corresponds to an arbitrary generalized tensor operation applied to a higher-order input tensor and *multiple* weight tensors (cf. Figure 2.5b). We describe both types of layers in more detail below.

Traditional convolutional layer. A traditional 2D-convolutional layer is parameterized by a 4th-order weight kernel $\mathcal{K} \in \mathbb{R}^{H \times W \times S \times T}$, where H (resp. W) is the height (resp. width) of the filter, and S (resp. T) is the number of input (resp. output) channels. Such a layer maps a 3rd-order input tensor $\mathcal{U} \in \mathbb{R}^{X \times Y \times S}$ to another 3rd-order output tensor $\mathcal{V} \in \mathbb{R}^{X' \times Y' \times T}$, where X (resp. Y) is the height (resp. width) of the input feature maps, and X' (resp. Y') is the height (resp. width) of the output feature maps. We can write this convolutional layer concisely using our generalized tensor operations:

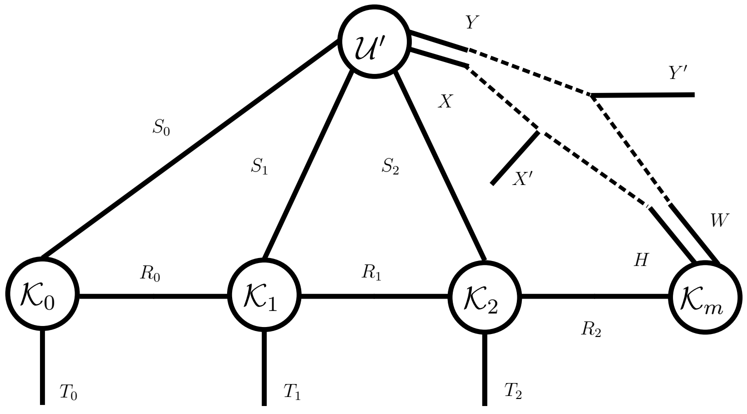
$$\mathcal{V} = \mathcal{U} \left(*_H^X \circ *_W^Y \circ \times_S^S \right) \mathcal{K}. \tag{2.1}$$

Moreover, a convolutional layer involves a multi-edge-double-node operation, where multiple primitive tensor operations apply to different modes. Specifically, there are two tensor convolutions: one on the modes with dimensions X, H, while the other on the modes with dimensions Y, W. A tensor contraction further applies on the modes with dimension S.



$$\mathcal{V} = \mathcal{U}\left(*_H^X \circ *_W^Y \circ \times_S^S\right)\mathcal{K}$$

(a) **Convolutional layer.**



$$\mathcal{U}^{(1)} = \mathcal{U}' \times_{S_0}^{S_0} \mathcal{K}^{(0)}$$
$$\mathcal{U}^{(\ell+1)} = \mathcal{U}^{(\ell)}\left(\times_{R_{\ell-1}}^{R_{\ell-1}} \circ \times_{S_\ell}^{S_\ell}\right)\mathcal{K}^{(\ell)}$$
$$\mathcal{V}' = \mathcal{U}^{(m)}\left(*_H^X \circ *_W^Y \circ \times_{R_{m-1}}^{R_{m-1}}\right)\mathcal{K}^{(m)}$$

(b) **mTT-convolutional layer.**

Figure 2.5: **Comparison between a convolutional layer and a tensorial layer. (a)** The traditional convolutional layer is a building block for CNN; **(b)** The mTT-convolutional layer is a building block for TNN-mTT.

Tensorial layers. Tensorial layers involve applying a generalized tensor operation to an input tensor and multiple weight kernels. We illustrate several tensorial layers in Figure 2.5 and Figure 2.6. In Figure 2.5b, we illustrate a tensorial layer inspired by the tensor-train (TT) layer [35]. We will refer to this layer as a mTT-convolutional layer (the letter 'm' is for 'modified'; this layer is slightly different from the one in [35]). A mTT layer takes an $(m+2)$-order input tensor $\mathcal{U}'$ and returns an output tensor $\mathcal{V}'$ of the same order. This layer has $(m+1)$ kernels $\{\mathcal{K}_i\}_{i=0}^m$ as parameters, in order to preserve the multi-dimensional structure of $\mathcal{U}'$. Mode-i of $\mathcal{U}'$ contracts with its corresponding kernel $\mathcal{K}_i$, and interactions between modes are captured by contractions between adjacent kernels (e.g., $\mathcal{K}_i$

and $\mathcal{K}_{i+1}$). These contractions are crucial for modeling multi-dimensional transformations with high expressive power. Thus, a mTT-convolutional layer enables processing of a higher-order input. We refer to a network with mTT-convolutional layers as a TNN-mTT. In Figures 2.6a to 2.6c, we develop other tensorial layers inspired by tensor-ring (TR), canonical polyadic (CP), and Tucker (TK) tensor decompositions [33, 40]; we refer to the corresponding networks as TNN-mTR, TNN-mCP, and TNN-mTK networks, respectively.

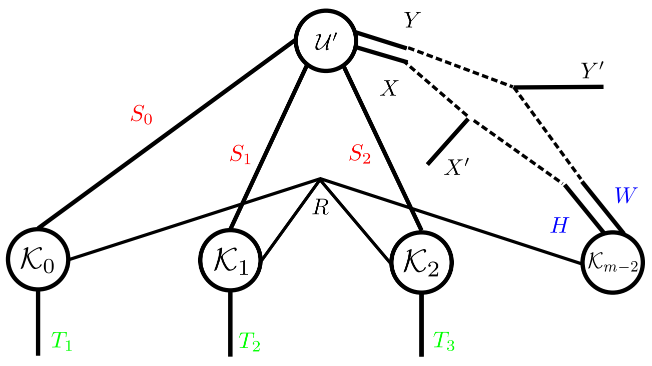
2.4.2 Relationship between NNs and TNNs

Interpretation via generalized tensor decompositions. We can use a tensorial layer to approximate a higher-order linear layer (fully-connected or convolutional). Suppose $\mathcal{U}$, $\mathcal{K}$, and $\mathcal{V}$ in Equation (2.1) are reshaped into higher-order tensors $\mathcal{U}'$, $\mathcal{K}'$, and $\mathcal{V}'$, such that input/output channels are indexed by m modes (i.e., $\mathcal{U}' \in \mathbb{R}^{X \times Y \times S_0 \times \cdots \times S_{m-1}}$, $\mathcal{K}' \in \mathbb{R}^{H \times W \times S_0 \times \cdots \times S_{m-1} \times T_0 \times \cdots \times T_{m-1}}$, and $\mathcal{V}' \in \mathbb{R}^{X' \times Y' \times T_0 \times \cdots \times T_{m-1}}$, where $S = \prod_{i=0}^{m-1} S_i$ and $T = \prod_{i=0}^{m-1} T_i$). We then have the following relationship between $\mathcal{U}'$, $\mathcal{K}'$, and $\mathcal{V}'$:

$$\mathcal{V}' = \mathcal{U}'\big(*_X^H \circ *_Y^W \circ \times_{S_0}^{S_0} \circ \cdots \circ \times_{S_{m+1}}^{S_{m+1}} \big) \mathcal{K}'. \tag{2.2}$$

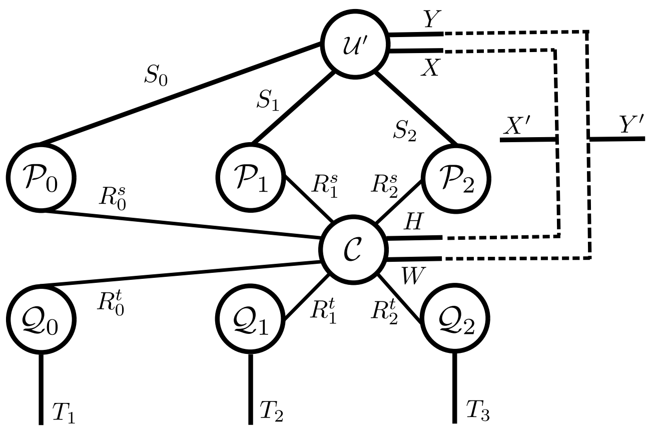
For a TNN-mTT tensorial layer, the kernels $\{\mathcal{K}_i\}_{i=0}^m$ correspond to factors of $\mathcal{K}'$, when $\mathcal{K}'$ can be represented with a *modified tensor-train decomposition*:

$$\mathcal{K}' \triangleq \mathsf{mTT}\big(\{\mathcal{K}_i\}_{i=0}^{m-1}\big) = \mathcal{K}_0 \times_{R_0}^{R_0} \mathcal{K}_1 \times_{R_1}^{R_1} \cdots \times_{R_{m-1}}^{R_{m-1}} \mathcal{K}_m. \tag{2.3}$$



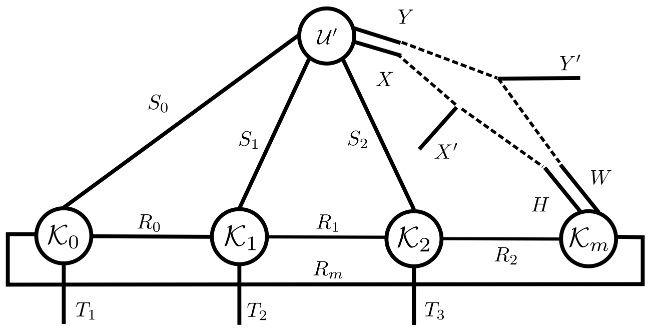
$$\mathcal{U}^{(1)} = \mathcal{U}' \times_{S_0}^{S_0} \mathcal{K}^{(0)}$$
$$\mathcal{U}^{(\ell+1)} = \mathcal{U}^{(\ell)} \left(\otimes_{R}^{R} \circ \times_{S_\ell}^{S_\ell} \right) \mathcal{K}^{(\ell)}$$
$$\mathcal{V}' = \mathcal{U}^{(m)} \left(*_{H}^{X} \circ *_{W}^{Y} \circ \times_{R}^{R} \right) \mathcal{K}^{(m)}$$

(a) **mCP-convolutional layer.**



$$\mathcal{U}^{(0)} = \mathcal{U}' \times_{S_0}^{S_0} \mathcal{P}^{(0)} \cdots \times_{S_{m-1}}^{S_{m-1}} \mathcal{P}^{(m-1)}$$
$$\mathcal{U}^{(1)} = \mathcal{U}^{(0)} \left(*_{H}^{X} \circ *_{W}^{Y} \circ \times_{R_0^s}^{R_0^s} \circ \cdots \circ \times_{R_{m-1}^s}^{R_{m-1}^s} \right) \mathcal{C}$$
$$\mathcal{V}' = \mathcal{U}^{(1)} \times_{R_0^t}^{R_0^t} \mathcal{Q}^{(0)} \cdots \times_{R_{m-1}^t}^{R_{m-1}^t} \mathcal{Q}^{(m-1)}$$

(b) **mTK-convolutional layer.**



$$\mathcal{U}^{(1)} = \mathcal{U}' \times_{S_0}^{S_0} \mathcal{K}^{(0)}$$
$$\mathcal{U}^{(\ell+1)} = \mathcal{U}^{(\ell)} \left(\times_{R_{\ell-1}}^{R_{\ell-1}} \circ \times_{S_\ell}^{S_\ell} \right) \mathcal{K}^{(l)}$$
$$\mathcal{V}' = \mathcal{U}^{(m)} \left(*_{H}^{X} \circ *_{W}^{Y} \circ \times_{R_{m-1}}^{R_{m-1}} \circ \times_{R_m}^{R_m} \right) \mathcal{K}^{(m)}$$

(c) **mTR-convolutional layer.**

Figure 2.6: **Variants of tensorial layers. (a)** The mCP-convolutional layer is a building block for TNN-mCP; **(b)** The mTK-convolutional layer is a building block for TNN-mTK; **(c)** The mTK-convolutional layer is a building block for TNN-mTR.

This motivates us to compress a linear layer into a tensorial layer, and more broadly, compress a traditional NN into a compact TNN. In Section 2.5, we will study relevant compression algorithms in detail.

Hypothesis sets of NNs and TNNs. Suppose the class of traditional NNs and our proposed TNNs share the same architecture (i.e., only the tensor operation in each layer is different). We illustrate the relations between their hypothesis sets in Figure 2.7. Let $\mathcal{G}^q$ and $\mathcal{H}^q$ denote the classes of functions that can be represented by NNs and TNNs, both with at most q parameters. **(1)** *TNNs generalize NNs.* Formally, for any $q > 0$, $\mathcal{G}^q \subseteq \mathcal{H}^q$ holds. **(2)** *NNs can be mapped to TNNs with fewer parameters and thus TNNs can be used for compression of NNs.* Formally, there exists $p \leq q$ such that $\mathcal{H}^p \subseteq \mathcal{G}^q$.

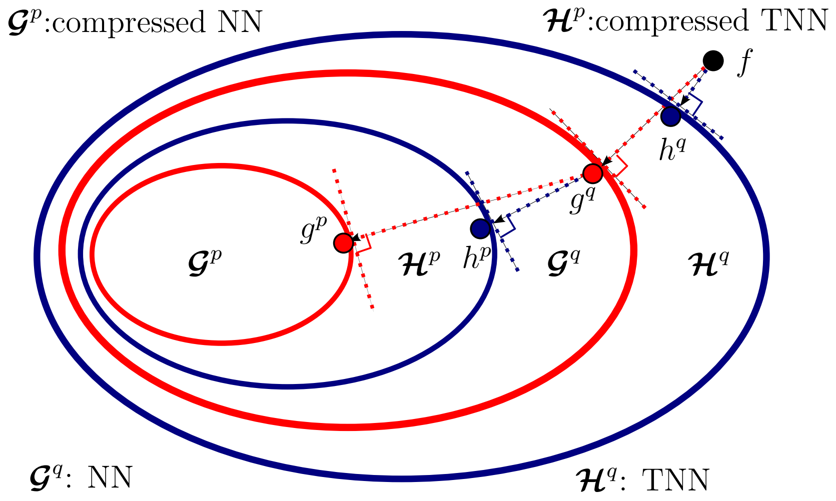

$\boldsymbol{\mathcal{G}}^p$:compressed NN
$\boldsymbol{\mathcal{H}}^p$:compressed TNN
f
h^q
g^q
g^p
h^p
$\boldsymbol{\mathcal{G}}^p$
$\boldsymbol{\mathcal{H}}^p$
$\boldsymbol{\mathcal{G}}^q$
$\boldsymbol{\mathcal{H}}^q$
$\boldsymbol{\mathcal{G}}^q$: NN
$\boldsymbol{\mathcal{H}}^q$: TNN

Figure 2.7: **Relationship between traditional NNs and TNNs.** Suppose the class of NNs and TNNs have the same architecture (i.e., only the tensor operation at each layer is different), and f is the target concept. **(1) Learning** of a NN with q parameters results in g^q that is closest to f in $\mathcal{G}^q$, while learning of a TNN with q parameters results in h^q that is closest to f in $\mathcal{H}^q$. Apparently, h^q is closer to f than g^q, **(2) Compression** of a pre-trained NN $g^q \in \mathcal{G}^q$ to NNs with p parameters ($p \leq q$) results in g^p that is closest to g^q in $\mathcal{G}^p$, while compression of g^q to TNNs with p parameters results in h^p that is closest to g^q in $\mathcal{H}^p$. Apparently, the compressed TNN h^p is closer to g^q than the compressed NN g^p.

2.5 Algorithms for TNNs

In this section, we investigate practical algorithms for TNNs. We first develop prediction and backpropagation algorithms for TNNs, which allows us to train a TNN from scratch. We then consider algorithms that distill a compact TNN from a pre-trained model.

2.5.1 Prediction in TNNs

Prediction with TNN is similar to that of traditional neural networks: the input passes through layers in a feedforward manner. Each layer in a TNN involves applying a generalized tensor operation to the input and multiple weight kernels before applying a nonlinear function such as ReLU. While it is difficult to determine the most efficient order to evaluate the primitives of a generalized tensor operation in general, we derive efficient orders for all TNN architectures introduced in this chapter. For example, we can efficiently evaluate each mTT-convolutional layer as follows:

$$\mathcal{U}_1 = \mathcal{U}' \times_{S_0}^{S_0} \mathcal{K}_0, \tag{2.4a}$$

$$\mathcal{U}_{i+1} = \mathcal{U}_i \big(\times_{R_{i-1}}^{R_{i-1}} \circ \times_{S_i}^{S_i} \big) \mathcal{K}_i, \tag{2.4b}$$

$$\mathcal{V}' = \mathcal{U}_m \big(*_H^X \circ *_W^Y \circ \times_{R_{m-1}}^{R_{m-1}} \big) \mathcal{K}_m. \tag{2.4c}$$

Here $\mathcal{U}'$ is the layer input, and $\mathcal{V}'$ is the output. The tensors $\{\mathcal{U}_i\}_{i=1}^m$ are intermediate results. We provide efficient strategies for performing the forward pass for the other tensorial layers in Figure 2.6, summarize the complexity (the number of FLOPs and amount of parameter storage required) for each forward pass in Table 2.2.

2.5.2 Learning in TNNs

To train a TNN via stochastic gradient descent, we derive backpropagation rules for each tensorial layer displayed in Figures 2.5 and 2.6. To derive such rules, we consider the partial derivatives of some loss function $\mathcal{L}$ with respect to the input ($\partial \mathcal{L} / \partial \mathcal{U}'$) and kernel factors (e.g., $\{\partial \mathcal{L} / \partial \mathcal{K}_i\}_{i=0}^{m}$ in a mTT-convolutional layer), given $\partial \mathcal{L} / \partial \mathcal{V}'$. As previously done for performing a forward pass, we develop efficient strategies for executing backpropagation with each type of tensorial layer. For an mTT-convolutional layer, an efficient strategy for performing backpropagation is

$$\frac{\partial \mathcal{L}}{\partial \mathcal{U}_m} = \frac{\partial \mathcal{L}}{\partial \mathcal{V}'} (*_H^{X'^\top} \circ *_W^{Y'^\top}) \mathcal{K}_m, \tag{2.5a}$$

$$\frac{\partial \mathcal{L}}{\partial \mathcal{K}_m} = \frac{\partial \mathcal{L}}{\partial \mathcal{V}'} (*_X^{X'^\top} \circ *_Y^{Y'^\top} \times_{T_0}^{T_0} \circ \cdots \circ \times_{T_{m-1}}^{T_{m-1}}) \mathcal{U}_m, \tag{2.5b}$$

$$\frac{\partial \mathcal{L}}{\partial \mathcal{U}_i} = \frac{\partial \mathcal{L}}{\partial \mathcal{U}_{i+1}} (\times_{R_i}^{R_i} \circ \times_{T_i}^{T_i}) \mathcal{K}_i, \tag{2.5c}$$

$$\frac{\partial \mathcal{L}}{\partial \mathcal{K}_i} = \frac{\partial \mathcal{L}}{\partial \mathcal{U}_{i+1}} (\times_X^X \circ \times_Y^Y \circ \times_{S_0}^{S_0} \circ \cdots \circ \times_{T_{i-1}}^{T_{i-1}} \circ \times_{S_{i+1}}^{S_{i+1}} \circ \cdots \circ \times_{S_{m-1}}^{S_{m-1}}) \mathcal{U}_i, \tag{2.5d}$$

$$\frac{\partial \mathcal{L}}{\partial \mathcal{U}'} = \frac{\partial \mathcal{L}}{\partial \mathcal{U}_1} (\times_{R_0}^{R_0} \circ \times_{T_0}^{T_0}) \mathcal{K}_0, \tag{2.5e}$$

$$\frac{\partial \mathcal{L}}{\partial \mathcal{K}_0} = \frac{\partial \mathcal{L}}{\partial \mathcal{U}_1} (\times_X^X \circ \times_Y^Y \circ \times_{S_1}^{S_1} \circ \cdots \circ \times_{S_{m-1}}^{S_{m-1}}) \mathcal{U}', \tag{2.5f}$$

where $*^\top$ denotes a transposed convolution. We derive efficient backpropagation strategies for the other tensorial layers in [50] and summarize and their complexities in Table 2.2.

Learning from scratch (Learn-Scratch). We can train any TNN from scratch (referred to as Learning from Scratch, or Learn-Scratch in short), given suitable algorithms for forward and backward passes. Since a TNN is formed by replacing each layer in a

Layer	O(params.)	O(forward ops.)	O(backward ops., input)	O(backward ops., params.)
original	k^2N^2	$k^2N^2D^2$	$k^2N^2D^2$	N^2D^4
mCP	$(mN^{\frac{2}{m}} + k^2)R$	$(mN^{1+\frac{1}{m}} + k^2N)RD^2$	$(mN^{1+\frac{1}{m}} + k^2N)RD^2$	$(mN^{1+\frac{1}{m}} + D^2N)RD^2$
mTK	$(2mN + k^2R^{2m-1})R$	$(2mN + k^2R^{2m-1})RD^2$	$(2mN + k^2R^{2m-1})RD^2$	$(2mN + R^{2m-1}D^2)RD^2$
mTT	$(mN^{\frac{2}{m}}R + k^2)R$	$(mN^{1+\frac{1}{m}}R + k^2N)RD^2$	$(mN^{1+\frac{1}{m}}R + k^2N)RD^2$	$(mN^{1+\frac{1}{m}}R + D^2N)RD^2$
mTR	$(mN^{\frac{2}{m}} + k^2)R^2$	$(mN^{1+\frac{1}{m}} + k^2N)R^2D^2$	$(mN^{1+\frac{1}{m}} + k^2N)R^2D^2$	$(mN^{1+\frac{1}{m}} + D^2N)R^2D^2$

Table 2.2: **The number of parameters and operations required by various tensorial layers.** This table shows the special case when $X = Y = X' = Y' = D$, $S = T = N$, $H = W = k$, and $D \gg k$ (see section 2.4 for notations). *Remark: The number of FLOPs does not accurately reflect the actual running time on GPUs, as the existing CUDA library can not fully utilize the degree of parallelism in general tensor operations.*

traditional NN with a tensorial layer, Learn-Scratch is as straightforward as training a traditional NN but is inefficient if we have a pre-trained reference NN.

2.5.3 Compression via Knowledge Distillation

Suppose we aim to compress a pre-trained neural network $g^q \in \mathcal{G}^q$ to a model with p parameters, where $p \ll q$. As is illustrated in Figure 2.7, $\mathcal{H}^p$ is a broader class of networks than $\mathcal{G}^p$, and hence our goal is to obtain the $h^p \in \mathcal{H}^p$ that is, in some sense, closest to g^q, rather than obtain the analogous $g^p \in \mathcal{G}^p$. We expect that searching for such a h^p yields a network that outperforms the analogous g^p in terms of predictive accuracy. Intuitively, we aim to "project" a pre-trained NN $g \in \mathcal{G}^q$ to a TNN $h^\star \in \mathcal{H}^p$. (Note that we omit the superscripts on g and h to simplify notation.) Denote the input to g as $\mathcal{U}$ and $\mathcal{U}'$ is a reshaped version of $\mathcal{U}$ (so that $\mathcal{U}'$ may be an input for h). our goal is to find $h^\star$ such that

$$h^\star = \arg\min_{h \in \mathcal{H}^p} \mathsf{dist}(h(\mathcal{U}'), g(\mathcal{U})), \tag{2.6}$$

where $\mathsf{dist}(\cdot,\cdot)$ denotes any distance(-like) metric (e.g., the square of the ℓ_2 distance) between the set of network outputs (the logits in classification problems). Solving Equation (2.6) is known as *knowledge distillation*; this process "distills" the knowledge from g and "instills" it into $h^\star$ [46].

Because the class $\mathcal{H}^p$ of TNNs is so vast, in practice, we minimize the objective in Equation (2.6), over a much smaller class of TNNs. Concretely, given the input data $\mathcal{U}$ and $g \in \mathcal{G}^q$, we minimize the objective over the class of TNN-mTTs, TNN-mTRs, TNN-mCPs, and TNN-mTKs, where we assign each of these models a pre-specified number of layers, kernels per layer, and kernel dimensions, with a total of p parameters. Given a model in the class of TNNs selected, let $\{\mathcal{K}_\ell^{(i)}\}_{i=0}^m$ denote the set of $(m+1)$ kernels of the ℓ^{th} layer of that model (replace m with $2m$ for TNN-mTKs). Our goal is to now search for kernels $\{\mathcal{K}_\ell^{(i)}\}_{i,\ell}$ for all L layers in the TNN, such that these kernels can be used to construct the TNN h that is a good approximation to g. Specifically, we aim to solve

$$\{\mathcal{K}_\ell^{(i)\star}\}_{i,\ell} = \arg \min_{\{\mathcal{K}_\ell^{(i)}\}_{i,\ell}} \mathsf{dist}(g(\mathcal{U}), h(\mathcal{U}'; \{\mathcal{K}_\ell^{(i)}\}_{i,\ell})). \tag{2.7}$$

Here, dist denotes a distance metric, which we assume as the squared ℓ_2 distance in this work. In what follows, we discuss three different approaches for solving Equation (2.7).

Layer-wise decomposition (Layer-Decomp). Given the relationship between TNNs and NNs (cf. Section 2.4.2), we may solve Equation (2.7) with the following two steps: **(1)** For each layer (e.g., layer ℓ), we reshape the original kernel $\mathcal{K}^{(\ell)}$ of g into a higher-order tensor $\mathcal{K}'^{(\ell)}$, and **(2)** we solve $\{\mathcal{K}_\ell^{(i)}\}_i$ such that applying corresponding tensor operation to those kernels produces the best approximate of $\mathcal{K}'^{(\ell)}$ (we assume that $\mathcal{K}^{(\ell)}$ is reshaped in a way such that the dimensions of $\mathcal{K}'^{(\ell)}$ match the ones of the approximate). The second step for a mTT-convolutional layer is solving the following optimization problem.

$$\{\mathcal{K}_\ell^{(i)\star}\}_i = \arg \min_{\{\mathcal{K}_\ell^{(i)}\}_i} \|\mathcal{K}'^{(\ell)} - \mathsf{mTT}(\{\mathcal{K}_\ell^{(i)}\}_i)\|^2, \ \forall \ell \in [L], \tag{2.8}$$

where $\mathsf{mTT}(\{\mathcal{K}_\ell^{(i)}\}_i)$ denotes the result of the generalized tensor operation in Figure 2.5b on $\{\mathcal{K}_\ell^{(i)}\}_i$ (we can formulate similar problems for other tensorial layers). Typically, one solves Equation (2.8) via an alternating least squares method [51], as Equation (2.8) reduces to solving a least-squares problem if we fix all but one kernel in each layer. However, such a method typically does not yield accurate solutions to Equation (2.7). Thus, we usually only use it to initialize parameters for more advanced approaches.

End-to-end knowledge distillation (E2E-KD). A classic algorithm of knowledge distillation is to minimize the ℓ_2 distance between the outputs of h and g. Formally, we can formulate the optimization problem as:

$$\{\mathcal{K}_\ell^{(i)\star}\}_{i,\ell} = \arg \min_{\{\mathcal{K}_\ell^{(i)}\}_{i,\ell}} \|h(\mathcal{U}') - g(\mathcal{U})\|^2. \tag{2.9}$$

However, solving the problem has two main drawbacks: **(1)** The backpropagation is very expensive as it requires end-to-end gradients flow in a TNN; **(2)** The optimization is unstable when the factors in all layers are solved simultaneously. To avoid these challenges, we propose decomposing it into a sequence of L sub-problems.

Sequential Knowledge Distillation (Seq-KD). For the ℓ^{th} sub-problem, we obtain the kernels $\{\mathcal{K}_\ell^{(i)}\}_i$ by minimizing the ℓ_2 distance between the intermediate results of the

ℓ^{th} layers of g and h, i.e., $\mathcal{V}^{(\ell)}$ and $\mathcal{V'}^{(\ell)}$,

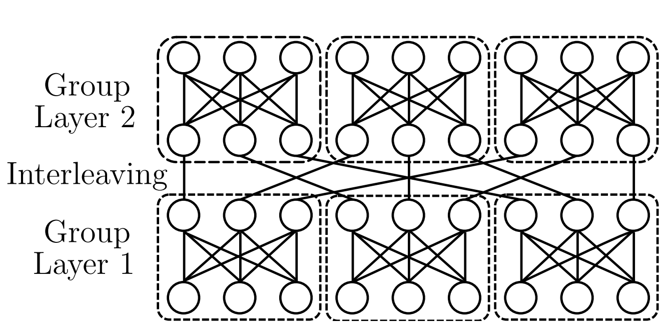
$$\{\mathcal{K}_{\ell}^{(i)\star}\}_i = \arg \min_{\{\mathcal{K}_{\ell}^{(i)}\}_i} \|\mathcal{V}^{(\ell)} - \mathcal{V'}^{(l)}\|^2, \ \forall \ell \in [L]. \tag{2.10}$$

where the input to the ℓ^{th} layer is the output from its previous layer, i.e., $\mathcal{U}^{(\ell)} = \mathcal{V}^{(\ell-1)}$ and $\mathcal{U'}^{(\ell)} = \mathcal{V'}^{(\ell-1)}$. We can use SGD to solve the problem once we derive the backpropagation rule for the general tensor operation used in the ℓ^{th} layer of g. Since ℓ^{th} sub-problem depends on the result from its previous one, the algorithm requires us to solve them *sequentially* from the bottom layers to the upper ones.

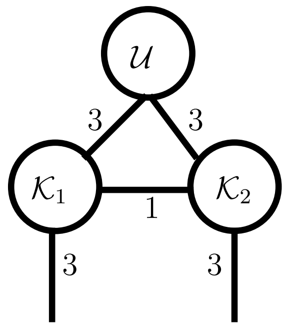
2.6 Interpretation of Existing Compact Architectures

Recent advances in compact architecture designs such as *Inception* [16], *Xception* [52], *interleaved group convolutions* [17], and *bottleneck structures* [15, 18] propose to group multiple primitive operations into modules. We will show that we can express all such modules using the framework of tensorial layers (with minor modifications).

Interleaved group modules. The critical idea in interleaved group modules involves dividing and branching the input into several blocks and constraining each block's connections to avoid computations across blocks. The architectures of tensorial layers utilize a similar strategy: for example, the tensorial layer in Figure 2.8b has the same architecture as the network in Figure 2.8a, where each length-nine input branches into three blocks and connections exist *only* within each block. This idea of grouping operations plays a vital role in the development of Inception [16] and Xception [52].



(a) Neuron Connections.



(b) Diagram.

Figure 2.8: An **interleaved group module** without nonlinearity **(a)** is expressed as a tensorial layer **(b)**.

Bottleneck modules. A bottleneck structure forces a model to adopt a compact representation by constructing a narrow bottleneck (with fewer hidden units) in the middle of each module. Such modules correspond to the low-rank structures used in tensorial layers, as illustrated by the following example with matrices: consider a weight matrix $\boldsymbol{W} \in \mathbb{R}^{S \times S}$, its low-rank decomposition $\boldsymbol{W} = \boldsymbol{P}\boldsymbol{Q}$ (with $\boldsymbol{P} \in \mathbb{R}^{S \times R}$ and $\boldsymbol{Q} \in \mathbb{R}^{R \times S}$). This model requires an input vector $\boldsymbol{u} \in \mathbb{R}^{S}$ to first be multiplied by $\boldsymbol{P}$ and then by $\boldsymbol{Q}$ during a forward pass. Therefore, the input $\boldsymbol{u}$ goes into a low-dimensional space $\mathbb{R}^{R}$ after being multiplied by $\boldsymbol{P}$, resulting in a bottleneck in this two-steps module. In practice, the bottleneck module in [15, 18] can be represented by tensor diagrams (cf. Figure 2.9), whose input with kN channels is first mapped to a structure with N channels by kernel $\mathcal{K}_0$.

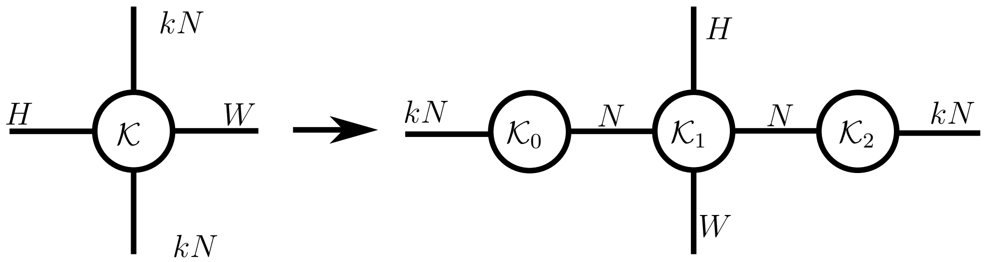


Figure 2.9: A **bottleneck module** without nonlinearity is expressed as a Tucker decomposition of the original layer.

Discussion of compact architecture designs. The two examples above illustrate one way of designing compact tensorial layers. This design process starts with a traditional layer (fully-connected or convolutional), followed by (optional) reshaping and some tensor decomposition of the (reshaped) kernel. Consequently, the original layer transforms into a tensorial layer with a compact structure. We can also design novel architectures from scratch by, for example, using tensor networks as building blocks for other architectures (cf. Section 2.3). One recent attempt that applies this methodology is [37], where *tensor-train networks* are used to introduce multilinear operations to an RNN.

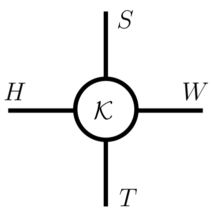
2.7 Experimental Results

We divide this section into two subsections. In Section 2.7.1, we use pre-trained models to evaluate the effectiveness of our compression algorithms (cf. Section 2.5.3). In Section 2.7.2, we demonstrate that our tensorial neural networks can be trained from scratch (i.e., without reference models) on a wide range of datasets and backbone models. In both scenarios, we show that our TNNs maintain high accuracy, even when they utilize significantly fewer parameters than traditional neural networks.

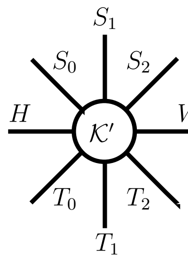
Considerations for TNN experiments. There are three items we consider when designing the experiments with TNNs that follow: **(1) Kernel reshaping.** We refer to an architecture whose kernels are reshaped into higher-order tensors (before performing a low-rank kernel factorization) as a TNN; we refer to an architecture containing factorized kernels but without reshaping as a NN. Although the latter is also a TNN, we still call it a NN, as the resulting architecture (after low-rank factorization) consists only of low-order operations (i.e., matrix multiplications and multi-channel convolutions), as in traditional neural networks. In what follows, we will compare the performance of TNNs to that of NNs. **(2) Types of tensor networks.** Existing NN baselines are networks that do not involve any kernel reshaping and use classical kernel decompositions, e.g., SVD [30, 31], CP [20, 31], and TK [21]. Therefore, we refer to these architectures as NN-SVD, NN-CP, and NN-TK architectures, where the suffix denotes the type of kernel decomposition. As discussed in Section 2.4, we may use kernel reshaping and other types of decompositions to obtain TNNs, which achieve better expressive power than NNs (cf. Figure 2.7). Consequently, we refer to these architectures that involve reshaping kernels as TNN-mCPs, TNN-mTTs, TNN-mTRs, etc. **(3) Training or compression strategy.** We train the above models either via knowledge distillation or from scratch. To distinguish these two strategies, we use the term *compression* for knowledge distillation (i.e., there exists a pre-trained reference network to *compress*). We use the term *TNN-based compression* (TNN-C) to describe the process of training the TNN-mCPs, TNN-mTTs, TNN-mTRs, etc. via knowledge distillation, and the term *low-rank compression* (NN-C) to describe the analogous process for training the NN-SVDs, NN-CPs, NN-TKs, NN-TTs, etc.

2.7.1 Knowledge Distillation

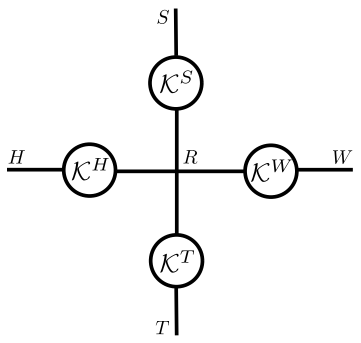
In this subsection, we evaluate different algorithms of knowledge distillation in Section 2.5.3, namely *layer-wise decomposition* (Layer-Decomp), *end-to-end knowledge distillation* (E2E-KD), and *sequential knowledge distillation* (Seq-KD). We conduct extensive experiments on compressing convolutional layers in ResNet-32 for CIFAR10, and we aim



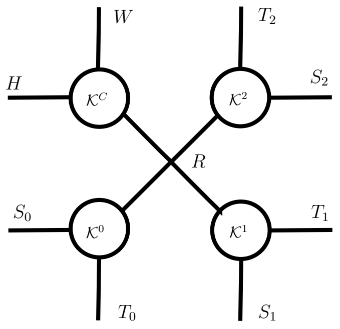
(a) Original kernel.



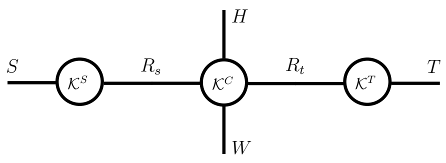
(b) Tensorized kernel.



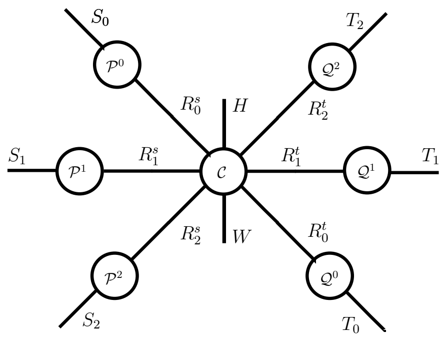
(c) Canonical polyadic (CP)



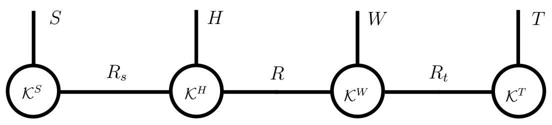
(d) mCP $(m = 3)$



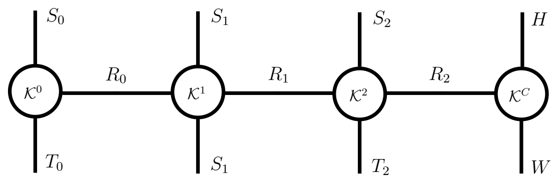
(e) Tucker (TK)



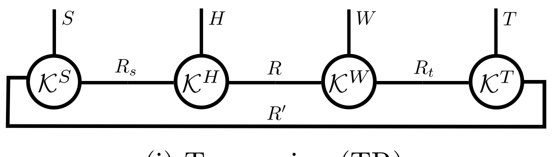
(f) mTK $(m = 3)$



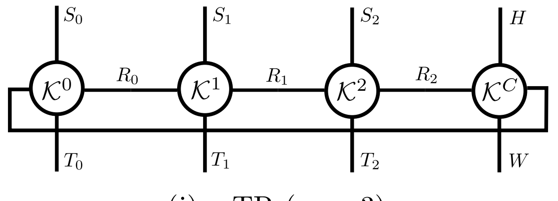
(g) Tensor-train (TT)



(h) mTT $(m = 3)$



(i) Tensor-ring (TR)



(j) mTR $(m = 3)$

Figure 2.10: **Diagrams of tensor decompositions.**

to figure out the best strategy for combining these algorithms.

Experimental Setup. We find that Layer-Decomp is merely better than random guesses in our experiments (see the test errors in Figure 2.11 at the beginning), Therefore, we can only use Layer-Decomp as initialization for E2E-KD and Seq-KD. With both algorithms, all layers are compressed uniformly at the same compression rate *except for the first and last layers*. Therefore, the compression rate is both layer-wise and (approximately) global. (We investigate the non-uniform allocation of all parameters across layers, but empirical results show that uniform assignment performs the best.) For all experiments, we use Adam optimizer with an initial learning rate of 10^{-3}, which decays by 10 every 50 epochs.

Our algorithm achieves 5% higher accuracies than the baselines on CIFAR-10 using ResNet-32. The results from Table 2.3 demonstrate that our TNNs maintain high accuracies even after the pre-trained networks are highly compressed. Given a pre-trained ResNet-32 and compression rate of 10%, the NN-CP with E2E-KD reduces the original accuracy from 93.2% to 86.93%; while the TNN-mCP with Seq-KD maintains the accuracy as 91.28% with the same compression rate — a performance loss of 2% with only 10% of the number of parameters. Furthermore, TNN-C achieves further aggressive compression — a performance loss of 6% with only 2% of the number of parameters. We observe similar trends (higher compression and accuracy) for TNN-mTT. The structure of the mTK decomposition makes TNN-mTK less effective with very high compression, since the decomposition poses a narrow bottleneck, which may lose necessary information. Increasing the network size to 20% of the original provides reasonable performance on

CIFAR-10 for TNN-mTK.

Architect.	Compression rate				Architect.	Compression rate			
	5%	10%	20%	40%		2%	5%	10%	20%
NN-SVD [30, 31]	83.09	87.27	89.58	90.85	TNN-TR [53]†	-	80.80†	-	90.60
NN-CP [20, 31]	84.02	86.93	88.75	88.75	TNN-mCP	85.7	89.86	**91.28**	-
NN-TK [21]	83.57	86.00	88.03	89.35	TNN-TK	61.06	71.34	81.59	87.11
NN-TT [34]*	77.44	82.92	84.13	86.64	TNN-mTT	78.95	84.26	87.89	-

†Cited from [53], the accuracy of 80.8% is achieved by 6.67% compression rate.

*The architecture is proposed as a baseline in [34].

Table 2.3: Test accuracy of ResNet-32 on CIFAR10. We compare **end-to-end knowledge distillation (E2E-KD)** using **low-rank compression (NN-C)** against **sequential knowledge distillation (Seq-KD)** with **TNN-based compression (TNN-C)**. The original ResNet-32 achieves 93.2% test accuracy with 0.46M parameters [5].

TNN-based compression, sequential knowledge distillation, or both? Table 2.3 shows that *TNN-C with Seq-KD* outperforms *NN-C with traditional E2E-KD*. Now we address the following question: is one factor (Seq-KD or TNN-C) primarily responsible for increased performance, or is the benefit due to synergy between the two?

(1) We present the accuracies of different compression methods in Table 2.5. Other than at very high compression rate (5% column in Table 2.5), Seq-KD consistently outperforms E2E-KD. In addition, Seq-KD converges faster than E2E-KD, and Figure 2.11 plots the test error over the number of gradient updates for various compression methods.

(2) We present the effect of different architectures on accuracy in Table 2.4, Table 2.7 and Table 2.8. **(2.1)** First, we compare TNNs with NNs via Seq-KD. Interestingly, as demonstrated in Table 2.4, if TNN-based compression is used, the test accuracy is restored for even very low compression rates[1]. **(2.2)** Second, we compare TNNs with NNs via Learn-Scratch. As demonstrated in Table 2.7 and Table 2.8, TNNs outperform NNs trained using

[1]Note that TNN-mTK remains an exception for aggressive compression due to the extreme bottleneck structure that we previously discussed.

Learn-Scratch under the same number of parameters.

These results confirm that TNNs are more flexible than traditional NNs as TNNs allow exploitation of invariant structures in the original parameters: such structures are exploited by our proposed TNN-based compression (our TNN-C), but not by low-rank compression (NN-C). Therefore, our results show that TNN-C and Seq-KD are symbiotic, and *both* are necessary to obtain high accuracy with significant compression.

Architecture	Compression rate 5%	Compression rate 10%	Architecture	Compression rate 5%	Compression rate 10%
NN-CP [20, 31]	83.19	88.50	TNN-mCP	**89.86**	**91.28**
NN-TK [21]	**80.11**	**86.73**	TNN-mTK	71.34	81.59
NN-TT [34]	80.77	87.08	TNN-mTT	**84.26**	**87.89**

Table 2.4: Test accuracy for ResNet-32 on CIFAR-10. We compare **sequential knowledge distillation (Seq-KD)** against **learning from scratch (Learn-Scratch)** using our **TNNs**. The original ResNet-32 achieves 93.2% accuracy with 0.46M parameters [5].

Architecture	Compression rate							
	5%		10%		20%		40%	
	Seq	E2E	Seq	E2E	Seq	E2E	Seq	E2E
NN-SVD [30, 31]	74.04	**83.09**	85.28	**87.27**	**89.74**	89.58	**91.83**	90.85
NN-CP [20, 31]	83.19	**84.02**	**88.50**	86.93	**90.72**	88.75	**89.75**	88.75
NN-TK [21]	80.11	**83.57**	**86.75**	86.00	**89.55**	88.03	**91.30**	89.35
NN-TT [34]	**80.77**	77.44	**87.08**	82.92	**89.14**	84.13	**91.21**	86.64

Table 2.5: Test accuracy of ResNet-32 on CIFAR-10. We compare **sequential knowledge distillation (Seq-KD)** against **end-to-end knowledge distillation (E2E-KD)** using NN-C. The original ResNet-32 achieves 93.2% accuracy with 0.46M parameters [5].

Convergence rate. Compared to end-to-end knowledge distillation (E2E-KD), an ancillary benefit of sequential knowledge distillation (Seq-KD) is that it is *much* faster and leads to more stable convergence. Figure 2.11 plots compression error over the number of gradient updates for various methods (This experiment is for NN-C with 10% compression rate). There are three salient points: first, Seq-KD has a very high error in the beginning while the "early" blocks are tuned (and the rest of the network is left unchanged to the values after tensor decomposition). However, as the final block is tuned (around 2×10^{11} gradient updates) in the figure, the errors drop to nearly a minimum immediately. In comparison, E2E-KD requires 50–100% more gradient updates to achieve stable performance. Finally, the result also shows that for each block, Seq-KD achieves convergence very quickly (and nearly monotonically), which results in the stair-step pattern since extra tuning of a block does not improve (or appreciably reduce) performance.

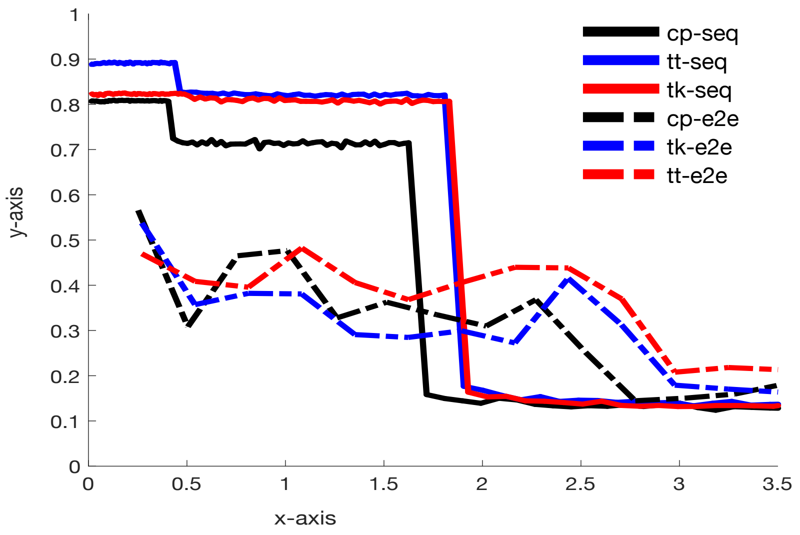


Figure 2.11: Test error curves for **sequential knowledge distillation (Seq-KD)** v.s. **end-to-end knowledge distillation (E2E-KD)** on ResNet-32 for CIFAR-10. Both use **layer-wise decomposition (Layer-Decomp)** for initialization.

2.7.2 Learning from Scratch

While it is beneficial to have a pre-trained model as reference (see Table 2.6 for a comparison), there are scenarios that knowledge distillation is not applicable: **(1)** The pre-trained model is simply not available; **(2)** The model is too deep that a sequential knowledge distillation is too expensive; **(3)** We aim to learn TNNs with even higher expressive power than NNs. In this subsection, we verify the performance of our TNNs when trained from scratch for a wide range of backbone models and datasets.

Architecture	Seq-KD			Learn-Scratch		
	2%	5%	10%	2%	5%	10%
TNN-mCP	85.70	89.86	91.28	81.41	82.12	82.93
TNN-mTK	61.60	71.34	81.59	60.65	61.46	65.75
TNN-mTT	78.95	84.26	87.89	79.95	81.82	83.08

Table 2.6: Test accuracy for ResNet-32 on CIFAR-10. We compare **sequential knowledge distillation (Seq-KD)** against **learning from scratch (Learn-Scratch)** using our **TNNs**. The original ResNet-32 achieves 93.2% accuracy with 0.46M parameters [5].

Wide-ResNet for CIFAR-100. To demonstrate that TNNs are compatible with other backbones (in addition to ResNet), we evaluate our TNNs with Wide-ResNet backbone [14] on the CIFAR-100 dataset. As shown in Table 2.7, our TNNs (in particular TNN-mTT), when trained from scratch, already outperform other state-of-the-art low-rank factorization-based methods.

Compression rate	**0.5%**	**1%**	2%	5%
NN-TT [34]	37.02%	54.65%	52.69%	51.42%
NN-CP [20, 31]	**40.74%**	**58.04%**	56.9%	64.83%
Compression rate	0.33%	**0.5%**	0.66%	**1%**
TNN-mTT	61.67%	**65.36%**	66.82%	**68.83%**

Table 2.7: Test accuracy of Wide-ResNet-28-10 on CIFAR-100. We compare **baseline NNs** against our TNNs by **training all models from from scratch** (i.e., without reference models). The original model achieves 81.25% accuracy with 36.5M parameters [14].

ResNet for ImageNet-2012. To show that our TNNs scale to large datasets, we evaluate their performance on the ImageNet-2012 dataset with a ResNet-50 backbone. The results in Table 2.8 show that our TNNs significantly outperform the low-rank factorization-based methods at each compression rate. Furthermore, our TNNs maintain very high accuracies given less than 10% of the parameters of the original ResNet-50.

Architecture	Compression rate					
	1%	2%	5%	10%	20%	50%
NN-CP	57.86%	64.17%	69.37%	71.52%	72.08%	72.44%
NN-TT	56.82%	62.23%	65.54%	66.21%	66.90%	66.92%
NN-TR	56.59%	62.97%	69.59%	71.61%	73.04%	73.21%
Architecture	Compression rate					
	0.5%	1%	5%	10%	20%	50%
TNN-mCP	72.65%	73.76%	74.03%	75.00%	75.31%	77.31%
TNN-mTT	69.27%	73.04%	73.51%	73.50%	73.87%	74.14%
TNN-mTR	67.49%	73.23%	74.12%	75.01%	75.32%	75.16%

Table 2.8: Top-1 test accuracy of ResNet-50 on ImageNet. We compare **baseline NNs** against **our TNNs**, where all models are **trained from scratch**. The original ResNet-50 model achieves 78.03% Top-1 test accuracy with 25.6M parameters [5].

VGG, ResNet and Wide-ResNet with Full Parameters. While we use TNNs mostly for model compression in this chapter, one remaining question is the performance of TNNs when they have the same number of parameters as the original model. To answer this question, we train TNN-mTT*from scratch* with architectures VGG-16 [2], ResNet-34 [15] and WRN-28-10 [14] on CIFAR-10. As shown in Table 2.9, TNNs (without hyper-parameter optimization) match/outperform their original model (where the hyper-parameters are highly optimized) when their numbers of parameters are the same.

Accuracy	TNN-mTT VGG	NN VGG	TNN-mTT WRN	NN WRN	TNN-mTT ResNet	NN ResNet
Training	100%	100%	100%	100%	100%	100%
Test	93.68%	92.64%	95.09%	95.83%	91.79%	92.49%

Table 2.9: **Performance of TNNs v.s. NNs counterparts on CIFAR-10.** NN stands for the uncompressed model proposed by the original paper. All models are trained from scratch (i.e., without reference models).

2.8 Conclusion

In this chapter, we introduced a new suite of generalized tensor algebra, which provides systematic notations for generalized tensor operations (a.k.a., tensor networks). Based on these generalized tensor operations, we developed a family of tensorial layers, extending existing fully-connected/convolutional layers in traditional neural networks. We constructed tensorial neural networks (TNNs) using tensorial layers as building blocks, and empirically showed that our TNNs maintain high predictive performance even when they contain significantly fewer parameters than traditional neural networks. Our experiments on LeNet-5, VGG, ResNet, and Wide-ResNet consistently verified that our TNNs outperform the state-of-the-art low-rank architectures under the same compression rate.

Chapter 3: AutoTNN: A Framework for Representing and Learning Tensorial Neural Networks

3.1 Overview

Modern neural networks are expressive over various learning problems but at the cost of increased width and depth. State-of-the-art convolutional models, for example, can contain up to several billion parameters [54, 55]. Unfortunately, the size and training costs of such networks are at odds with a rapidly emerging industrial and academic interest in performing learning tasks on low-fidelity hosts such as IoT and mobile devices [56, 57].

An increasingly popular approach for generating compact yet expressive models is to use *tensorial neural network* (TNNs) [20, 50, 58, 59], which factorize each layer's weight tensor into several smaller factors. As a result, each TNN layer is a multilinear operation among its input and weight factors.

One particular method by which a TNN can arise is reshaping a network weight into a higher-order tensor. The reshaped tensor takes a factorized form, such as canonical polyadic (CP), Tucker (TK), and tensor-train (TT) decompositions. Factorization of a reshaped tensor is an efficient way of representing underlying properties such as periodicity and modularity invariances/similarities, which often exist in neural network models [20, 34, 50, 53], and preserved in the factorization. Factorization is especially useful for low-rank, higher-order reshaped weights since we can capture abundant structural information without much representation redundancy. Figure 3.1 depicts this scenario using the example of a vector displaying periodicity.

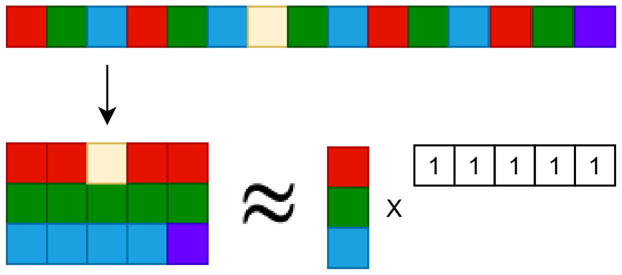


Figure 3.1: **Tensor reshaping.** The figure shows how reshaping can help reduce the number of parameters while preserving critical structural properties. A vector of length 15 displays periodicity in its entries (represented here by repeating colors) except for a few entries. Reshaping it into a 3×5 matrix and then taking a factorized tensor approximation (rank-1 SVD) **a)** reduces the number of parameters to store and **b)** culls out artifacts (non-periodic elements) while representing periodicity sans redundancy.

In addition to the lighter weight, TNNs preserve the same predictive power level over various backbone networks and tasks. However, in contrast to the availability of high-performance libraries for convolutional neural networks (CNNs) (e.g., NVIDIA's cuDNN for GPUs, Intel's MKL for CPUs), efficient library-based solutions are not available for training TNNs, especially those based on CNNs.

In this work, we introduce an open-source library and a suite of meta-algorithms, collectively termed as AutoTNN, to efficiently construct and train TNNs. Given a backbone network, AutoTNN will tensorize its layers, compress the tensorial weights, and conduct end-to-end training on the resulting TNN. In particular, the user does not need to independently develop any part of the TNN architecture.

AutoTNN interprets the forward and backward pass of a TNN as a *generalized* einsum graph/sequence. Here, "generalized" means including convolutions, an operation not supported by any existing einsum implementation but handled by our meta-function conv_einsum. A lack of support for convolution operations in einsum prevented the evaluation of tensorized CNNs via well-known algorithms such as netcon. However, deriving tensor decompositions with convolutions is nontrivial as (a) the factorization involving a convolution is challenging to represent as an einsum string, and (b) backpropagation rules for such structure designs were not derived in any prior work. conv_einsum evaluates these tensor operations with an optimal sequence, rather than naively performing all the tensor operations from left to right, resulting in a vastly reduced/improved number of *floating point operations* (FLOPs) during training and inference, a mechanism we refer to as the *optimal sequencer*.

In totality, AutoTNN addresses computational and memory overhead challenges associated with automated TNN construction and training. In conjunction with conv_einsum, the optimal sequencer, and checkpointing, this work provides a framework and library which significantly improves the efficiency of TNN design and deployment.

Contributions. In summary, our contributions include:

1. We develop an open-source library AutoTNN and framework for designing and training a TNN given a backbone network and specified tensor decomposition.

2. Our algorithm conv_einsum optimizes the training process. In particular, conv_einsum interprets forward and backward passes as sequences of multilinear operations, including (multi-way) convolutions, extending the classical einsum.

3. Furthermore, conv_einsum utilizes the library opt-einsum [60], which generalizes the netcon algorithm [49], to efficiently determine an optimal order of these operations.

4. We exploit a mechanism referred to as *checkpointing* [61] which can balance the computational cost of a forward or backward pass of a TNN with the memory cost of large intermediate products during evaluation.

5. Our experiments demonstrate the expressiveness of TNNs with AutoTNN over various tasks, including video classification, speech recognition, and image classification.

3.2 Related Works

Low-rank factorization. Various types of low-rank factorization have been proposed to reduce the number of parameters in linear layers. Pioneering works proposed to flatten/unfold the parameters in convolutional layers into matrices (known as *matricization*), followed by (sparse) dictionary learning or matrix decomposition [30, 31, 32]. Subsequently, [20] and [21] showed that it is possible to compress the parameters directly by standard tensor decompositions (in particular, CP decomposition or Tucker decomposition [33]). Further groundbreaking works [19, 34] demonstrated that the low-order weights could be efficiently compressed by the tensor-train decomposition [35] by first reshaping the parameters into higher-order tensors. This paradigm was later extended in two directions: (1) the tensor-train decomposition is used to compress LSTM/GRU layers in recurrent neural networks (RNNs) [36] and higher-order recurrent neural networks [37, 38]; and (2) other decompositions are explored for better compression, such as the tensor-ring decomposition [40] and block-term decomposition [41].

Other compression methods. TNNs belong to the large family of low-rank approximation methods, which complements other compression techniques such as quantization and pruning. Many papers have justified that combining these different lines of the works may render more competitive model compression. For instance, [44] verify that TNNs with quantization achieve SOTA results on 3D tasks, and [43] demonstrate that TNNs with pruning obtain SOTA image classification. This work, however, is chiefly concerned with the automated development and benchmarking of TNNs without any other compression methods. For future work, we will investigate incorporating quantization, pruning, and knowledge distillation techniques directly into AutoTNN.

Existing libraries. Various libraries support tensor operations. **(1) Pytorch** [62] supports specialized tensor operations commonly used in neural networks, including various convolutional layers and the `einsum` function. However, it is non-trivial to implement arbitrary tensor operations optimally. **(2) TensorLy** [63] supports common tensorial operations across various platforms, including Pytorch and TensorFlow. However, the library does not support the construction of arbitrary tensor networks. **(3) NumPy** [64] is a general computation library that has an optimal sequencer in its einsum function, but it does not support GPUs, nor does it support convolutions. **(4) Einops** [65] extends `einsum` to GPUs. To the best of our knowledge, no existing library supports general tensor networks on GPUs, and our work aims to close this gap. **(5) Gnetcon** [66] attempts to extend `einsum` to convolutions but does not support multi-way convolution between a collection of modes with more than two differing dimensions, nor was it fully integrated into an end-to-end training framework. **(6) Tensor Comprehensions (TC)** [67] optimizes tensorial computations through translation of generalized "Einstein" notations of tensorial sequences (including multi-way convolutions). However, TC cannot tensorize a network – the user needs to manually define their network architecture, whereas AutoTNN can automatically tensorize *and* compress a backbone network. Furthermore, defining tensor operation with more than two tensors in the language of TC requires a hand-coded sequence of binary operations. In contrast, our conv_einsum can handle arbitrary tensor networks by faithfully adapting the syntax of einsum.

3.3 Tensor Operations and einsum

In this section, we outline multi-linear operations common to TNNs. Many of these operations are systematically expressible and computable via the popular notational framework and function einsum, first introduced by the Python library NumPy [64]. Therefore, we will first review the essences of tensor operations and their corresponding einsum representations. With these concepts in hand, we then formally introduce TNN.

Notations. We use lower case letters (e.g., $\boldsymbol{v}$) to denote vectors, upper case letters (e.g., $\boldsymbol{M}$) denote matrices, and curly letters (e.g., $\mathcal{T}$) denote tensors. For a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, we refer to a specific entry by the subscript notation $T_{i_1 i_2 \dots i_N}$, where $1 \leq i_n \leq I_n$ for $1 \leq n \leq N$. Furthermore, we refer to N as the order, a particular index n as a mode, and the magnitude of a mode I_n as dimension size. For example, a tensor $\mathcal{T} \in \mathbb{R}^{3 \times 4 \times 5}$ is a 3rd order tensor that has dimension size 4 at its second mode.

3.3.1 Representations of Multilinear Operations

The einsum function allows definitions of multilinear operations via string inputs. In this subsection, we highlight an example of multilinear operation involving three primitive operations (*contraction*, *batch product*, *outer product*). Consider two 3rd-order tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{B \times C \times I}$, $\mathcal{T}^{(2)} \in \mathbb{R}^{A \times C \times J}$, and a multilinear operation between these two tensors:

$$\mathcal{T}_{b,i,j} = \sum_{c=1}^{C} \mathcal{T}^{(1)}_{b,c,i} \, \mathcal{T}^{(2)}_{b,c,j}. \tag{3.1}$$

We can denote the operation above in einsum as:

```
T = einsum("bci,bcj->bij", T1, T2)
```

where the string in the quotation mark precisely specifies the operation, known as an einsum string. In this string, the letter "c" indicates *contraction* since it appears in both inputs but not the output; the letter "b" denotes *batch product* since it appears in both inputs and the output; lastly, the letters "i" and "j" represent *outer product* as they each appears in one of the two inputs and they both appear in the output.

3.3.2 From einsum to conv_einsum

While many popular libraries (e.g., NumPy, TensorFlow, PyTorch) implement einsum, none of them support convolutions, despite that convolution is (multi)linear and ubiquitous in modern neural networks. Therefore, we generalize einsum to a meta-function conv_einsum, which handles convolution as a primitive operation.

Tensor convolution generalizes the convolution on vectors to higher-order tensors. For instance, given two tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{X \times B \times C}$ and $\mathcal{T}^{(2)} \in \mathbb{R}^{L \times D \times E}$, we can define a convolution between the modes with dimension sizes X and L. The operation returns a 5th order tensor $\mathcal{T} \in \mathbb{R}^{X' \times B \times C \times D \times E}$, with its entries calculated as:

$$\mathcal{T}_{:,b,c,d,e} = \mathcal{T}^{(1)}_{:,b,c} * \mathcal{T}^{(2)}_{:,d,e}, \tag{3.2}$$

where $*$ denotes a convolution between two vectors. Note that the dimensions X and L can be different, and the output dimension X' depends on boundary condition (e.g., a standard convolution yields $X' = X + L - 1$). We write Equation (3.2) in conv_einsum as

```
T = conv_einsum("lbc,lde->lbcde|l", T1, T2)
```

In this scheme, the same letter "l" is used for different modes, even if their dimension sizes may differ. Furthermore, the placement of "l" right to the pipe-delimiter indicates that conv_einsum performs convolution on the corresponding modes. Notice that a letter for convolution appears in all inputs, the output, and after the delimiter.

We can use conv_einsum to represent multilinear operations on more than two inputs. For instance, consider three tensors $\mathcal{X} \in \mathbb{R}^{B \times F \times S \times H \times W}$, $\mathcal{K}^{(1)} \in \mathbb{R}^{F \times G \times K \times K}$, $\mathcal{K}^{(2)} \in \mathbb{R}^{S \times T \times K \times K}$, and an operation

$$\mathcal{Y}_{b,g,t,:,:} = \sum_{f=1}^{F} \sum_{s=1}^{S} \mathcal{X}_{b,f,s,:,:} * \mathcal{K}^{(1)}_{f,g,:,:} * \mathcal{K}^{(2)}_{s,t,:,:} \tag{3.3}$$

leads to an output tensor $\mathcal{Y} \in \mathbb{R}^{B \times G \times T \times H' \times W'}$. This is known as *interleaved group convolution* [17]. In conv_einsum, it writes as

```
T = conv_einsum("bfshw,fghw,sthw->bgthw|hw",X, K1, K2)
```

We will discuss how to evaluate a conv_einsum with more than two inputs in Section 3.4.2.

3.3.3 Compact Neural Networks via conv_einsum

In this subsection, we formulate various network layers in terms of conv_einsum.

Standard convolutional networks. We review the standard 2D-convolutional layer in neural networks. Such a layer is parameterized by a 4th order tensor $\mathcal{W} \in \mathbb{R}^{T \times S \times H \times W}$, which maps a 3rd order tensor $\mathcal{X} \in \mathbb{R}^{S \times H' \times W'}$ to a 3rd order tensor $\mathcal{Y} \in \mathbb{R}^{T \times H' \times W'}$:

```
Y = conv_einsum("bshw,tshw->bthw|hw", X, W)
```

Note that a neural network typically computes its inputs in mini-batches, so the conv_einsum string contains an additional letter "b" to index examples in a mini-batch. Since convolutional layers include fully-connected layers as a particular case when $H = W = 1$, we focus on designs of convolutional layers for the remainder of this subsection.

Tensorial neural networks. Convolutional layers motivate the importance and usage of TNNs since their structures benefited from reshaping and tensorial decomposition. Numerous works propose to design *tensorial layers* where the (reshaped) convolution kernel $\mathcal{W}$ is factorized using tensor decompositions [20, 21, 34, 50, 68]. Our proposed conv_einsum can handle these types of designs. Here, we present two representatives of efficient tensorial convolutional layer designs based on the CP decomposition [33].

(1) In a *CP convolutional layer* [20], the kernel $\mathcal{W}$ is factorized into 4 factors $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R \times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R \times S}$, $\boldsymbol{W}^{(3)} \in \mathbb{R}^{R \times H}$, $\boldsymbol{W}^{(4)} \in \mathbb{R}^{R \times W}$ such that

```
W = conv_einsum("rt, rs, rh, rw->tshw", W1, W2, W3, W4)
```

Plugging this decomposition into the 2D-convolutional layer, we obtain the following conv_einsum string for this layer:

```
Y = conv_einsum("bshw, rt, rs, rh, rw->bthw|hw", X, W1, W2, W3, W4)
```

(2) In a *reshaped CP convolutional layer* [50], the convolution kernel $\mathcal{W} \in \mathbb{R}^{T \times S \times H \times W}$ is first reshaped into a higher order tensor $\overline{\mathcal{W}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ such that $T = \prod_{m=1}^{M} T_m$, $S = \prod_{m=1}^{M} S_m$, and then factorized into $(m+1)$ tensors $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R \times T_m \times S_m}$ with $\mathcal{W}^{(0)} \in \mathbb{R}^{R \times H \times W}$. For example, when $M = 3$:

```
W = conv_einsum("r(t1)(s1),r(t2)(s2),r(t3)(s3),rhw"
->(t3)(t2)(t1)(s3)(s2)(s1)", W1, W2, W3, W0)
```

We can write the layer's conv_einsum string as:

```
Y = conv_einsum("b(s1)(s2)(s3)hw,r(t1)(s1),r(t2)(s2),r(t3)(s3),rhw
->n(t1)(t2)(t3)hw", X, W1, W2, W3, W0)
```

For both layers, R is the *rank* of the CP decomposition, which controls the number of parameters (i.e., compression rate) of the layer.

3.4 Algorithms

The last section presents how conv_einsum represents general multilinear operations in neural networks. In this section, we develop a suite of algorithms to efficiently implement conv_einsum. We organize this section as follows: **(1)** In Section 3.4.1, we develop an algorithm to reduce a conv_einsum function with two inputs to a collection of atomic PyTorch operations, which allows us to reuse GPU-optimized utilities in PyTorch to complete the computation. **(2)** In Section 3.4.2, we derive an optimal sequencer which automatically decomposes a conv_einsum function with an arbitrary number of inputs into a sequence of conv_einsum operations with two inputs; and **(3)** In Section 3.4.3, we utilize a *checkpointing* technique to reduce the memory overhead of our implementation further.

3.4.1 Atomic Operations

In the subsection, we show that any conv_einsum function can be realized by GPU-optimized PyTorch utilities, einsum and convNd (e.g., conv1d, conv2d). In particular, any two-inputs conv_einsum with convolution can be realized via a convNd. To understand why this is possible, we analyze the conv_einsum string for the conv1d function:

```
Y = conv_einsum("bsh,tsh->bth|h", X, W)
```

where "t" stands for the output channel, "s" the input channel, "h" the length of features/-filters, and "b" the batch size. Now, we can categorize these letters in terms of primitive operations. **(1)** The letter "h" is a convolution, appearing in both inputs and the output; **(2)** The letter "s" is a contraction, appearing in both inputs but not the output; **(3a)** The letter "t" is an outer product, appearing in the first input and the output; **(3b)** The letter "b" is another outer product, appearing in the second input and the output.

The conv1d function covers almost all mixtures of compound operations in which each operation type appears at most once, which we refer to as an *atomic operation*. However, we have two cases that are not covered: **(4)** A batch product that appears in both inputs and the output; and **(5)** A self-contraction that occurs in only one input. Fortunately, we can readily address these two edge cases. For **(4)**, the function conv1d supports a group-convolution option, which effectively extends to:

```
Y = conv_einsum("gtsh,bgsh->bgth|h", X, W)
```

where "g" stands for the filter group. In terms of tensor operations, it is a batch product, which appears in both inputs and the output. For **(5)**, such a letter can be eliminated by summing over the corresponding index in pre-processing.

Multiple letters with the same operation type. Now we address the scenario where multiple letters have the same operation type. For example, if there are two different letters in a conv_einsum string designated for convolution, we can use conv2d instead of conv1d. Notice that conv2d realizes a conv_einsum such as:

```
Y = conv_einsum("gtshw,bgshw->bgthw|h,w", X, W)
```

where "g" stands for the filter group and "h"/"w" represent the height/width respectively. In principle, we can use a convNd function to compute a conv_einsum function with N letters for convolution (though convNd for $N \geq 4$ requires custom implementation).

For non-convolution letters, all letters with the same type can be merged into one letter by preprocessing (i.e., the corresponding modes reshape into one compound mode), and the letter is converted back to multiple letters by post-processing (i.e., the compound mode reshape back to its corresponding modes).

3.4.2 Optimal Sequencer

Our conv_einsum leverages APIs in the existing open-source opt-einsum library for NumPy [60]. The opt-einsum library can handle determining the FLOPs-optimal evaluation order of tensor networks involving contractions, outer products, and batch products. Our conv_einsum introduces convolution handling to opt-einsum. However, due to the complexity of our convolutional functionality and the core APIs of opt-einsum, we only present a high-level overview of the basis of conv_einsum.

```
A=np.random.rand(4, 7, 9)
B=np.random.rand(10, 5)
C=np.random.rand(5, 4, 2)
D=np.random.rand(6, 8, 9, 2)
print(conv_einsum.contract_path
("ijk,jl,lmq, njpq->ijknp|j", A, B, C, D))
```

(a) **Tensor sequence generation.** A tensor sequence over a collection of tensors $\mathcal{A}, \mathcal{B}, \mathcal{C}$, and $\mathcal{D}$, involving contractions, convolutions, and batch products is analyzed. We store and print the optimal sequence of paths, which is stored in a string array path_info.

```
  Complete sequence:  ijk,jl,lmq,njpq->ijknp|j
     Naive FLOP count:  4.212e+05
 Optimized FLOP count:  2.056e+05
 Largest intermediate:  1.944e+05 elements
--------------------------------------------------------------------------------
                current                                remaining
--------------------------------------------------------------------------------
                lmq,jl->qj                             ijk,njpq,qj->ijknp|j
                qj,njpq->jnp|j                         ijk,jnp->ijknp|j
                jnp,ijk->ijknp|j                       ijknp->ijknp|j
```

(b) **An optimal sequence of paths.** The code, leveraging opt-einsum with our added support for convolutions, displays the optimal sequence of paths for the conv_einsum string submitted in Figure 3.2a. We are also presented with information about the naive left-to-right cost vs the cost of taking the suggested path, along with the size/cost of the largest intermediate.

Figure 3.2: **conv_einsum sample code.** The figure depicts the generation via NumPy of a set of tensors coalesced into one tensor sequence. The sequence is represented as a string and submitted to optimal sequencer of conv_einsum for path analysis. The output of the analysis is presented in Figure 3.2b.

The opt-einsum library relies on the well-known netcon [49] algorithm for determining an optimal order of evaluations. netcon was designed to handle operation sequences in tensor networks. For example, for $\mathcal{A} \in \mathbb{R}^{I \times J \times K}, \mathcal{B} \in \mathbb{R}^{J \times L}, \mathcal{C} \in \mathbb{R}^{L \times M}$, one might be

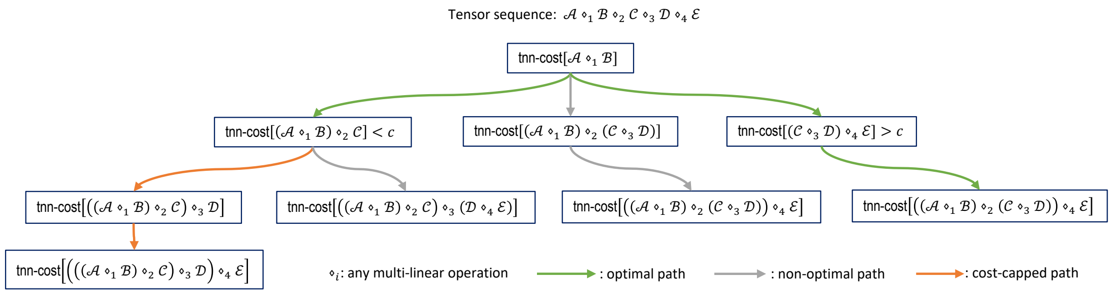

Tensor sequence: $\mathcal{A} \circ_1 \mathcal{B} \circ_2 \mathcal{C} \circ_3 \mathcal{D} \circ_4 \mathcal{E}$
tnn-cost$[\mathcal{A} \circ_1 \mathcal{B}]$
tnn-cost$[(\mathcal{A} \circ_1 \mathcal{B}) \circ_2 \mathcal{C}] < c$
tnn-cost$[(\mathcal{A} \circ_1 \mathcal{B}) \circ_2 (\mathcal{C} \circ_3 \mathcal{D})]$
tnn-cost$[(\mathcal{C} \circ_3 \mathcal{D}) \circ_4 \mathcal{E}] > c$
tnn-cost$[((\mathcal{A} \circ_1 \mathcal{B}) \circ_2 \mathcal{C}) \circ_3 \mathcal{D}]$
tnn-cost$[((\mathcal{A} \circ_1 \mathcal{B}) \circ_2 \mathcal{C}) \circ_3 (\mathcal{D} \circ_4 \mathcal{E})]$
tnn-cost$[((\mathcal{A} \circ_1 \mathcal{B}) \circ_2 (\mathcal{C} \circ_3 \mathcal{D})) \circ_4 \mathcal{E}]$
tnn-cost$[((\mathcal{A} \circ_1 \mathcal{B}) \circ_2 (\mathcal{C} \circ_3 \mathcal{D})) \circ_4 \mathcal{E}]$
tnn-cost$[(((\mathcal{A} \circ_1 \mathcal{B}) \circ_2 \mathcal{C}) \circ_3 \mathcal{D}) \circ_4 \mathcal{E}]$
$\circ_i$: any multi-linear operation
: optimal path
: non-optimal path
: cost-capped path

Figure 3.3: **Optimal sequencer example**. In this figure, conv_einsum deploys the optimal sequencer to analyze the path tree of an abstract, well-defined tensor sequence $\mathcal{A} \circ_1 \mathcal{B} \circ_2 \mathcal{C} \circ_3 \mathcal{D} \circ_4 \mathcal{E}$, where $\circ_i$ for $1 \leq i \leq 4$ is any collection of multi-linear operations, including convolutions, batch products, contractions, and outer products. The tree traversal strategy is a fusion of netcon and our tnn-cost API. The green path indicates a potentially viable optimal path, whereas the red path indicates a path which satisfies the cost cap c at each node. While the red path satisfies the cost cap, it may result in more FLOPs overall compared to the complete green path.

interested in the optimal order of evaluation of the tensor

$$\mathcal{T} = \sum_{l=1}^{L} \sum_{j=1}^{J} \mathcal{A}_{:,j,:} \otimes \mathcal{B}_{j,l} \otimes \mathcal{C}_{l,:}. \tag{3.4}$$

Let us momentarily suppress the index and summation notation of Equation (3.4), i.e., let $(AB) \triangleq \sum_{j=0}^{J-1} \mathcal{A}_{:,j,:}\mathcal{B}_{j,l}$, $(BC) \triangleq \sum_{l=0}^{L-1} \mathcal{B}_{j,l}\mathcal{C}_{l,:}$. The possible paths we may take to arrive at $\mathcal{T}$ include $(AB) \to (AB)C$, or $(BC) \to A(BC)$. Each of these paths has a predictable *contraction cost*, or the number of multiplications/additions (FLOPs), dependent on the dimensions of the tensor modes involved in each intermediate product. We can organize all the possible paths for this example and any contraction sequence in general into a tree. Each node is associated with the contraction cost of forming the product represented by that node. netcon efficiently traverses such path trees and determine the FLOPs-optimal path in a fast manner, even though this is an NP-hard problem in general [48]. Furthermore, netcon supports cost-capping, avoiding traversal down a branch if the resulting intermediate product exceeds some FLOPs cap c.

In particular, netcon is capable of handling all types of multilinear sequences except for those involving convolutions. For example, consider the tensor $\mathcal{T}_{p,:,q,r,t} = \sum_{n=1}^{N} \mathcal{B}_{n,p} \big(\mathcal{A}_{n,:,r} * \mathcal{C}_{:,q}\big) \mathcal{D}_{r,t}$ with a convolution. We may equivalently compute $\mathcal{T}$ as

$$\mathcal{T}_{p,:,q,r,t} = \left(\sum_{n=1}^{N} \left(\mathcal{B}_{n,p} \mathcal{A}_{n,:,r}\right) * \mathcal{C}_{:,q} \right) \mathcal{D}_{r,t} = \sum_{n=1}^{N} \mathcal{B}_{n,p} \left(\left(\mathcal{A}_{n,:,r} \mathcal{D}_{r,t}\right) * \mathcal{C}_{:,q} \right). \tag{3.5}$$

Our conv_einsum extends the netcon paradigm to handle convolutions simply by replacing the contraction cost function with a more general *TNN cost*, which adds the cost (FLOPs-wise) of the convolutions (if present) within an intermediate product at each node. We refer to this generalized scheme as the *optimal sequencer*. Figure 3.3 depicts the optimal sequencer analyzing the path tree an abstract tensor sequence. The action of the optimal sequencer is crucial for trimming the training time of our TNNs.

Modification of the cost model for training. The (netcon-based) opt-einsum sequencer only considers forward computation in tensor networks. However, in a neural network setting, we also need to consider the backpropagation computation. Specifically, given two inputs $\mathcal{T}^{(1)}$, $\mathcal{T}^{(2)}$, which interact through an atomic operation f resulting in an output tensor $\mathcal{T} = f(\mathcal{T}^{(1)}, \mathcal{T}^{(2)})$, the opt-einsum sequencer will calculate the cost of computing $\mathcal{T}$ without any concern for associated backpropagation calculations. However, the backpropagation algorithm needs to compute $\partial\mathcal{L}/\partial\mathcal{T}^{(1)} = g_1(\partial\mathcal{L}/\partial\mathcal{T}, \mathcal{T}^{(2)})$ and $\partial\mathcal{L}/\partial\mathcal{T}^{(2)} = g_2(\mathcal{T}^{(1)}, \partial\mathcal{L}/\partial\mathcal{T})$, where g_1 and g_2 are gradient calculations dependent on f. Therefore, we modify the cost from $\mathsf{cost}(f)$ to $\mathsf{cost}(f) + \mathsf{cost}(g_1) + \mathsf{cost}(g_2)$. For instance, consider f as a standard 2D-convolution, where the operation between the input $\mathcal{T}^{(1)} \in \mathbb{R}^{B\times S\times X\times Y}$ and the weight $\mathcal{T}^{(2)} \in \mathbb{R}^{T\times S\times H\times W}$ leads to the output $\mathcal{T} \in \mathbb{R}^{B\times T\times X'\times Y'}$. We have $\mathsf{cost}(f) = O(BHWXYTS)$ for the forward pass, and $\mathsf{cost}(g_1) = O(BHWX'Y'TS)$, $\mathsf{cost}(g_2) = O(BXYX'Y'TS)$ for the backward pass. In order to achieve optimal scheduling, we modify the cost function of opt-einsum to consider all three costs, which is inherited by the optimal sequencer of conv_einsum.

3.4.3 Checkpointing

TNNs use composite tensor operations (a.k.a. tensor networks) to design compact network layers. However, since we pairwisely evaluate a tensor network, computing a tensor network with N inputs leads to $(N - 1)$ intermediate results. Therefore, if we use an automatic gradient function, we will need to save these $(N - 1)$ intermediates in memory, causing high memory overhead. To avoid storing the intermediate products, we rely on gradient checkpointing [61] which recomputes the gradient during the backward pass rather than saving all intermediate results in memory. We can think of this mechanism as a trade-off between memory and computation. The total memory used by a neural network consists of static and dynamic memory. Static memory depends on the size of the model and some fixed costs built-in by PyTorch, while dynamic memory depends on the computational graph saved in the memory. Usually, when training a neural network, in the forward pass, the model caches all values of the activation neurons and reuses them in the backward pass calculation. Gradient checkpointing avoids any activation caching in the forward pass and thus can effectively relieve any potential memory overflow in that phase.

3.5 Experimental Results

In this section, we compare training and inference run-times of AutoTNN against the prevalent PyTorch. We demonstrate that AutoTNN provides an efficient solution for TNN deployment in different tasks across various domains. We use an NVIDIA GeForce RTX 2080Ti for all tasks.

Tasks. We test AutoTNN on a range of tasks under different network compression rates: **(1)** A classic two-stream convolutional neural network [69] is used for a video classification task, trained on the UCF-101 data set [70]. ResNet-101 [5] was chosen as the ConvNet for both the spatial and temporal streams, pre-trained on ImageNet [71]. The two-stream network is adapted from [69]. **(2)** An Automatic Speech Recognition task using the Conformer architecture [72], which incorporates convolution modules between the attention modules and the feed-forward neural network modules of a Transformer model [4]. We train the model on the LibriSpeech dataset [73]. **(3)** An image classification task trained on the CIFAR10 [74] data set using the classic ResNet-34 [5] architecture. State-of-the-art accuracy requires the usage of knowledge distillation (c.f. [50]).

Baselines. We compare AutoTNN against two baselines across all tasks: PyTorch implementation with and without checkpointing. In particular, we compare the usage of conv_einsum to optimally evaluate forward/backward tensor sequences against PyTorch with and without checkpointing to demonstrate the benefits of conv_einsum.

3.5.1 Accuracy and Memory Results

(1) TNNs demonstrate competitive accuracy under aggressive compression. Table 3.1 shows the test/training accuracy of TNNs using a reshaped CP decomposition of their tensor weights under different compression rates in three machine learning tasks, namely Video Classification (VC), Automatic Speech Recognition (ASR), and Image Classification (IC). A compression rate (CR) of $x\%$ indicates that the size of the TNN model is $x\%$ of the original/baseline model size. As shown in Table 3.1, for instance, a TNN using only 10% size of the original backbone model (i.e., a ResNet-34 with $21M$ parameters) maintains 98% of the baseline performance in an IC task on the CIFAR10 benchmark dataset.

Table 3.1: **TNN performance under various model scales for diverse machine learning tasks.** Automatic Speech Recognition (ASR) on LibriSpeech is measured by Word Error Rate (WER) (the lower, the better). Image classification (IC) on CIFAR10 (results from [50]) is measured by top-1 precision. Video classification (VC) on UCF-101 is measured by top-1 accuracy.

Compression Rate (CR)	IC	ASR	VC
Original	93.2	2.1	88.98
100%	-	2.08	89.00
50%	-	2.29	88.61
20%	-	2.36	88.10
10%	91.28	2.43	87.63
5%	89.86	3.01	86.62
2%	85.70	3.76	86.41

(2) Naive PyTorch implementation of TNNs suffers from high intermediary memory costs during training. AutoTNN significantly reduces these costs. In TNNs, some intermediate data objects can be prohibitively large during training computation, thus challenging to fit into memory. In Table 3.2, we present the maximal batch size allowed

for two large-scale tasks under different compression rates: video classification (VC) and automatic speech recognition (ASR). We observe that if the size of a TNN matches the original backbone model size (i.e., CR=100%), the maximal allowed batch size in a PyTorch implementation is 0 without checkpointing. Even if we compress the model to 1% of the original # of parameters, the maximal allowed batch size is still limited, making the computation infeasible or too slow. The non-optimal evaluation order by PyTorch results in large intermediate objects that do not fit into the memory. As indicated in Table 3.2, even incorporation of checkpointing into PyTorch implementation does not help much with the problem. Although checkpointing sidesteps save any intermediate results in memory arising during forward passes and instead recomputes them in the backward passes, the order of evaluation in the forward pass remains unchanged. If an intermediary computation causes overflow, this issue is likely to persist even in a checkpointing implementation – only the optimal sequencer can help with such a dilemma. On average, the most cost-parsimonious path found by the optimal sequencer will contain smaller intermediate products than a naive left-to-right PyTorch evaluation.

3.5.2 Runtime results

Our conv_einsum implements an optimal sequencer that evaluates a tensorial forward or backward pass in an order which incurs the minimum number of FLOPS. Additionally, conv_einsum uses checkpointing to avoid memory overflow.

(1) The optimal sequencer used in conv_einsum significantly improves the runtime efficiency of training and test in TNNs. In Figures 3.4 to 3.6, we compare the training and

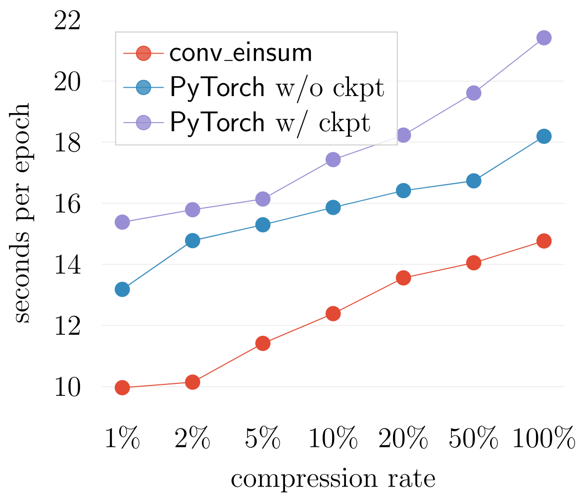
Table 3.2: **Maximum batch size for a speech and video task.** The maximal batch size allowed for data under varying compression rates and using different libraries on **(1)** an automatic speech recognition task on LibriSpeech and **(2)** a video classification task for spatial (S) and temporal (T) streams of a two-stream network on UCF-101. conv_einsum allows for larger batch sizes by efficiently evaluating tensorial forwards and backwards passes. "ckp" means checkpointing.

Automatic speech recognition task

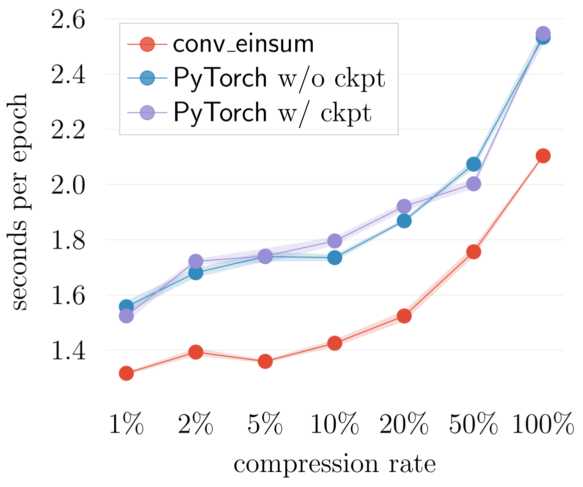
CR	conv_einsum	PyTorch w/ ckpt	PyTorch w/o ckpt
1%	14	8	6
2%	14	8	6
5%	12	6	4
10%	10	4	2
20%	8	2	0
50%	6	2	0
100%	4	1	0

Video classification task

	conv_einsum		PyTorch w/ ckpt		PyTorch w/o ckpt	
CR	S	T	S	T	S	T
1%	20	30	2	4	1	2
2%	20	30	2	4	0	0
5%	20	30	2	4	0	0
10%	18	30	1	2	0	0
20%	16	27	0	0	0	0
50%	12	22	0	0	0	0
100%	4	14	0	0	0	0

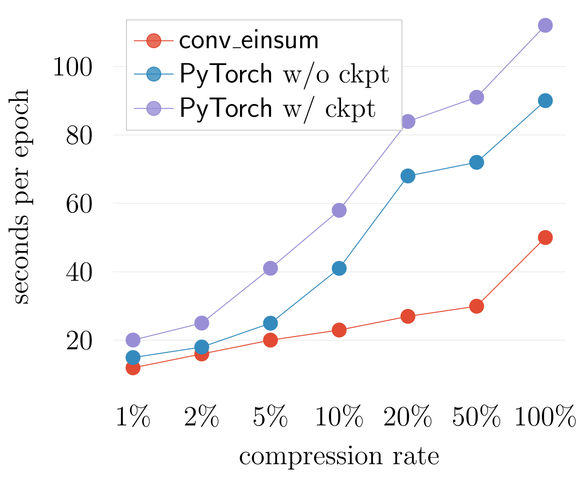


(a) RCP Train

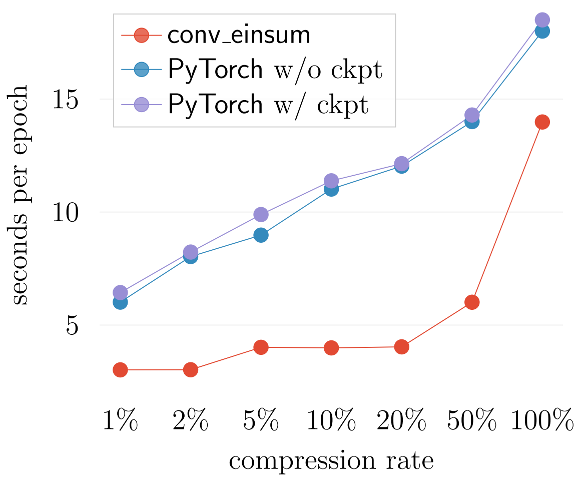


(b) RCP Test

Figure 3.4: **Runtime comparison between conv_einsum and PyTorch for an image classification task.** An RCP-TNN ($R = 3$) is trained on the CIFAR-10 dataset. Runtimes are averaged over 3 random runs, and error bars are denoted by the shaded areas.

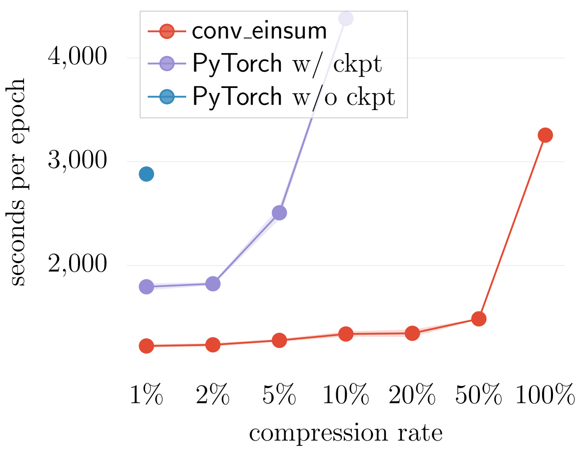


(a) CP Train

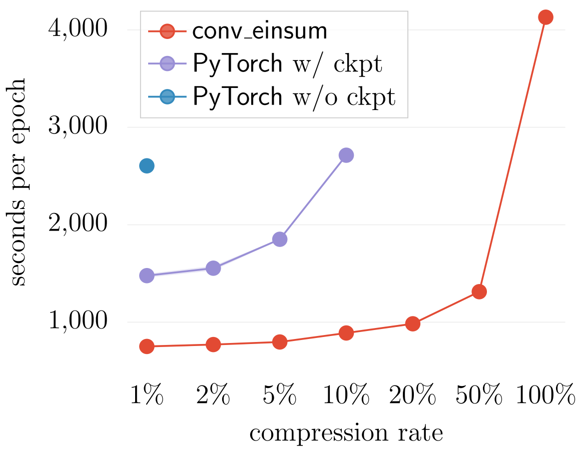


(b) CP Test

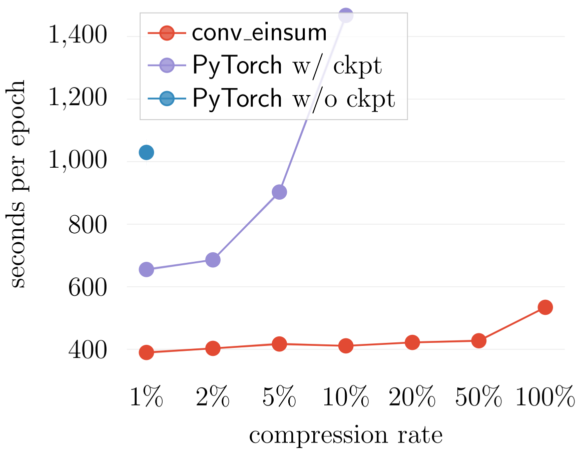
Figure 3.5: **Run-time comparison between conv_einsum and PyTorch for a speech recognition task**. A CP-TNN is trained on the LibriSpeech dataset. Runtimes are averaged over 3 random runs, and error bars are denoted by the shaded areas.



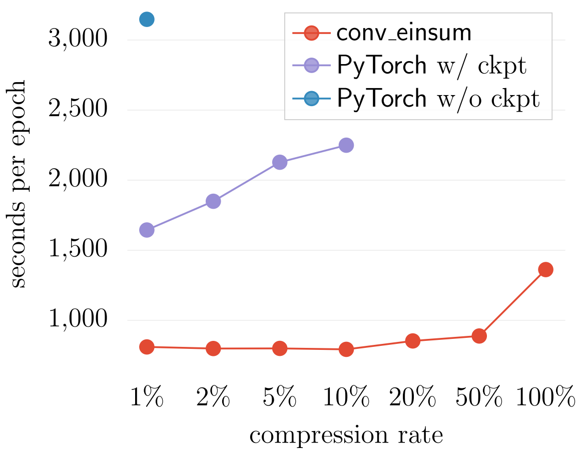
(a) RCP Train (spatial)



(b) RCP Test (spatial)



(c) RCP Train (temporal)



(d) RCP Test (temporal)

Figure 3.6: **Run-time comparison between conv_einsum and PyTorch for a video classification machine learning task** An RCP-TNN ($R = 3$) is trained on the UCF-101 dataset. All tests were run using the maximum allowable batch size. PyTorch w/ checkpointing was only able to run without memory overflow for compression rates 1% - 10% and PyTorch w/o checkpointing was only able to run without memory overflow for compression rate 1%. Runtimes are averaged over 3 random runs, and error bars are denoted by the shaded areas.

test times between conv_einsum and PyTorch (with and w/o checkpointing) implementations over a wide range of model scales and using different forms of tensor decomposition. The IC and VC tasks use RCP decompositions, while the ASR task uses a standard CP decomposition. We observe that conv_einsum universally outperforms the baselines. In the VC task, we use the maximal allowable batch size for each model size, while in the ASR task, we compare implementations using the same batch size. Furthermore, in Table 3.3 we show that conv_einsum outperforms PyTorch in the IC task under different tensor decompositions. When memory requirement is the bottleneck of a task (such as VC), checkpointing helps accelerate the runtime by alleviating potential memory overflows and allowing more batches. On the other hand, when the batch sizes are the same (as in ASR), checkpointing itself trades computational complexity for space, thus increasing the overall runtime. In either scenario, conv_einsum achieves the fastest runtimes in all tasks compared to PyTorch implementations with and without checkpointing.

(2) AutoTNN works for weight tensors with different sizes. AutoTNN serves as a general efficient library solution and is tensor-structure-agnostic. The networks we have experimented with contain weights of vastly differing sizes. As shown in Figures 3.4 to 3.6, AutoTNN exhibits competitive results against existing PyTorch solutions (with or without) checkpointing over various tensor sizes.

(3) AutoTNN works for a variety of tensor forms. conv_einsum is both data-agnostic and structure-agnostic. Given any sequence of multilinear tensor operations, conv_einsum computes the optimal sequence with the least number of FLOPs. As a result, conv_einsum is a universal solution to training any TNN. Table 3.3 shows the results of the image classification task on CIFAR10 using different forms of tensor decomposition. We could

observe that conv_einsum outperforms PyTorch with/without checkpointing in all cases.

Table 3.3: Run-time (seconds per epoch) comparison between conv_einsum and PyTorch implementation of TNNs using different tensor decomposition forms in the image classification task on the CIFAR-10 dataset.

	conv-einsum		PyTorch w/o ckpt		PyTorch w/ ckpt	
Tensor Form	Train	Test	Train	Test	Train	Test
RCP	14	2.14	22	2.57	29	2.61
RTR	8	1.41	16	1.86	16	1.86
RTT	8	1.37	16	1.61	16	1.68
RTK	6	1.34	17	1.47	17	1.54

3.6 Conclusion

In this chapter, we introduce an open-source library, AutoTNN, which can build and efficiently train TNNs. Our AutoTNN is competitive against and, in many cases, superior to the standard PyTorch. For future work, we plan to further accelerate training and test times by incorporating parallel computation paradigms into the intra-layer tensor computations. Since our AutoTNN relies on the PyTorch backend, we can also use tensorRT to accelerate AutoTNN further. Additionally, we will investigate incorporating quantization, pruning, and knowledge distillation techniques directly into AutoTNN.

Chapter 4: Higher-order RNN for Spatio-temporal Learning

4.1 Overview

While computer vision has achieved remarkable successes in many realms, e.g., image classification, many real-life tasks remain out-of-reach for current deep learning systems, such as prediction from complex spatio-temporal data. Such data naturally arises in a wide range of applications such as autonomous driving, robot control [75], visual perception tasks such as action recognition [76] or object tracking [77], and weather prediction [78]. This kind of video understanding problems is challenging, since they require learning spatial-temporal representations that capture both content and dynamics simultaneously.

Learning from (video) sequences. Most state-of-the-art video models are based on recurrent neural networks (RNNs), typically some variations of *Convolutional LSTM* (ConvLSTM) where spatio-temporal information is encoded explicitly in each cell [78, 79, 80, 81]. These RNNs are first-order Markovian models in nature, meaning that the hidden states are updated using information from the previous time step only, resulting in an intrinsic difficulty in capturing long-range temporal correlations.

Incorporating higher-order correlations. For 1D sequence modeling, [82] and [37] proposed higher-order generalizations of RNNs for long-term forecasting problems. Higher-order RNNs explicitly incorporate an extended history of previous states in each update, which requires higher-order tensors to characterize the transition function (instead of a transition matrix as in the first-order RNNs). However, this typically leads to an exponential blow-up in the function's complexity. This problem is more pronounced when generalizing ConvLSTM to higher-order, and no prior work explores these generalizations.

Scaling up with tensor methods. To avoid the exponential blow-up in the complexity of transition function, tensor decompositions [83] have been investigated in higher-order RNNs [37]. Tensor decomposition avoids the exponential growth of model complexity and introduces an information bottleneck that facilitates effective representation learning. This bottleneck restricts the information that is passed on from one sub-system to another in a learning system [84, 85]. Previously, low-rank tensor factorization has been used to improve various deep network architectures [20, 21, 86]. However, its application has not been explored in the context of spatio-temporal LSTMs. The only approach that leveraged tensor factorization for compact higher-order LSTMs [37] considers exclusively sequence forecasting, which does not naturally extend to general spatio-temporal data.

Generalizing ConvLSTM to higher-orders. When extending to higher-order, we aim to design a transition function that can leverage all previous hidden states and satisfies three properties: **(1)** The operations preserve the spatial structure of the hidden states; **(2)** The receptive field increases with time. In other words, the longer the temporal correlation is captured, the larger the spatial context should be; **(3)** Finally, space and time complexities grow at most linearly with the number of times steps. Because previous transition functions in higher-order RNNs were for one-dimensional sequence, when directly extended to spatio-temporal data, they do not satisfy all three properties. A direct extension fails to preserve the spatial stricture or increases the complexity exponentially.

Contributions. In this chapter, we propose a higher-order convolutional LSTM model for complex spatio-temporal data satisfying all three properties. Our model incorporates a long history of states in each update while preserving their spatial structure using convolutional operations. Directly constructing such a model leads to an exponential growth of parameters in both spatial and temporal dimensions. Instead, our model is made computationally tractable via a novel convolutional tensor-train decomposition, which recursively performs a convolutional factorization of the kernels across time. Besides parameter reduction, this factorization introduces an information bottleneck enabling the learning of better representations. As a result, it achieves better results than previous works with only a fraction of the parameters.

We empirically demonstrate our model's performance on several challenging tasks, including early activity recognition and video prediction. We report an absolute increase of 8% accuracy over the state-of-the-art [81] for early activity recognition on the Something-Something v2 dataset. Our model outperforms both 3D-CNN and ConvLSTM by a large margin. We also report a new state-of-the-art multi-step video prediction on both Moving-MNIST-2 and KTH datasets.

4.2 Related Works

Tensor decompositions. Tensor decompositions such as CP, Tucker or Tensor-Train [33, 35], are widely used for dimensionality reduction [25] and learning probabilistic models [83]. These tensor factorization techniques have also been widely used in deep learning to improve performance, speed-up computation, and compress the deep neural networks [19, 20, 21, 50, 58, 87], recurrent networks [36, 88] and Transformers [89]. [36] has proposed tensor-train RNNs to compress both inputs-states and states-states matrices within each cell with TTD by reshaping the matrices into tensors, and showed improvement for video classification.

Departing from prior works that rely on existing tensor decompositions, we propose a novel *convolutional tensor-train decomposition* (CTTD) designed for efficient and compact higher-order convolutional recurrent networks. Unlike [36], we aim to compress higher-order ConvLSTM, rather than first-order fully-connected LSTM. We further propose Convolutional Tensor-Train decomposition to preserve spatial structure after compression.

Spatio-temporal prediction models. Prior prediction models have focused on predicting short-term video [90, 91] or decomposing motion and contents [92, 93, 94, 95]. Many of these works use ConvLSTM as a base module, which deploys 2D convolutions in LSTM to efficiently exploit spatio-temporal information. Some works modified the standard ConvLSTM to better capture spatio-temporal correlations [79, 80]. [91] demonstrated strong performance using a deep ConvLSTM network as a baseline, and we adopt this base architecture in this chapter.

4.3 Background: Convolutional LSTM and Higher-order LSTM

In this section, we briefly review *Long Short-Term Memory* (LSTM), and its generalizations *Convolutional LSTM* (ConvLSTM) for spatio-temporal modeling, and *higher-order LSTM* (d) for learning long-term dynamics.

Long Short-Term Memory [96]**.** LSTM is a first-order Markovian model widely used in 1D sequence learning. At each step, an LSTM cell updates its hidden state $\boldsymbol{h}^{(t)}$ and cell state $\boldsymbol{c}^{(t)}$ using the immediate previous states $\{\boldsymbol{h}^{(t-1)}, \boldsymbol{c}^{(t-1)}\}$ and the current input $\boldsymbol{x}^{(t)}$:

$$[\boldsymbol{i}^{(t)}; \boldsymbol{f}^{(t)}; \tilde{\boldsymbol{c}}^{(t)}; \boldsymbol{o}^{(t)}] = \sigma(\boldsymbol{W}\boldsymbol{x}^{(t)} + \boldsymbol{K}\boldsymbol{h}^{(t-1)}), \tag{4.1a}$$

$$\boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} \circ \boldsymbol{f}^{(t)} + \tilde{\boldsymbol{c}}^{(t)} \circ \boldsymbol{i}^{(t)},\ \boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \sigma(\boldsymbol{c}^{(t)}), \tag{4.1b}$$

where $\sigma(\cdot)$ denotes a $\mathsf{sigmoid}(\cdot)$ applied to the *input gate* $\boldsymbol{i}^{(t)}$, *forget gate* $\boldsymbol{f}^{(t)}$ and *output gate* $\boldsymbol{o}^{(t)}$, and a $\tanh(\cdot)$ applied to the *memory cell* $\tilde{\boldsymbol{c}}^{(t)}$ and *cell state* $\boldsymbol{c}^{(t)}$. $\circ$ denotes element-wise product. LSTMs have two major restrictions: **(a)** only 1D-sequences can be modeled, not spatio-temporal data such as videos; **(b)** they are difficult to capture long-term dynamics as first-order models.

Convolutional LSTM (ConvLSTM) [78, 97]**.** ConvLSTM addresses the limitation **(a)** by extending LSTM to model spatio-temporal structures within each cell, i.e., the states, cell memory, gates and parameters are all encoded as high-dimensional tensors:

$$[\mathcal{I}^{(t)}; \mathcal{F}^{(t)}; \tilde{\mathcal{C}}^{(t)}; \mathcal{O}^{(t)}] = \sigma(\mathcal{W} * \mathcal{X}^{(t)} + \mathcal{K} * \mathcal{H}^{(t-1)}), \tag{4.2}$$

where $*$ defines convolution between states and parameters as in convolutional layers.

Higher-order LSTM [37, 82]. HO-LSTM is a higher-order Markovian generalization of the basic LSTM, which partially addresses the limitation **(b)** in modeling long-term dynamics. Specifically, HO-LSTM explicitly incorporates more previous states in each update, replacing the first step in LSTM by

$$\left[\boldsymbol{i}^{(t)}; \boldsymbol{f}^{(t)}; \tilde{\boldsymbol{c}}^{(t)}; \boldsymbol{o}^{(t)}\right] = \sigma\left(\boldsymbol{W}\boldsymbol{x}^{(t)} + \Phi\left(\boldsymbol{h}^{(t-1)}, \cdots, \boldsymbol{h}^{(t-N)}\right)\right), \tag{4.3}$$

where Φ combines N previous states $\{\boldsymbol{h}^{(t-1)}, \cdots, \boldsymbol{h}^{(t-N)}\}$ and N is the *order* of the HO-LSTM. For example, Φ can be a linear function [82] or a polynomial function [37]:

$$\text{Linear:} \quad \Phi\left(\boldsymbol{h}^{(t-1)}, \cdots, \boldsymbol{h}^{(t-N)};\ \boldsymbol{T}^{(1)}, \cdots, \boldsymbol{T}^{(N)}\right) = \sum\nolimits_{i=1}^{N} \boldsymbol{T}^{(i)}\boldsymbol{h}^{(t-i)}. \tag{4.4}$$

$$\text{Polynomial:} \quad \Phi\left(\boldsymbol{h}^{(t-1)}, \cdots, \boldsymbol{h}^{(t-N)};\ \mathcal{T}\right) = \left\langle \mathcal{T},\ \boldsymbol{h}^{(t-1)} \otimes \cdots \otimes \boldsymbol{h}^{(t-N)} \right\rangle. \tag{4.5}$$

While a linear function requires the numbers of parameters and operations growing linearly in N, a polynomial function has space/computational complexity exponential in N if implemented naively.

Notations. To facilitate the reading, we provide a table of notations in Table 4.1.

4.4 Methodology: Convolutional Tensor-Train LSTM

In this section, we detail the challenges and requirements for higher-order ConvLSTM. We then introduce our model and motivate the design of each module by these requirements.

Symbol	**Meaning**	**Value** or **Size**
H	Height of feature map	-
W	Width of feature map	
C_{in}	# of input channels	
C_{out}	# of output channels	
t	Current time step	-
$\mathcal{W}$	Weights for $\mathcal{X}^{(t)}$	$[K \times K \times 4C_{\text{out}} \times C_{\text{in}}]$
$\mathcal{X}^{(t)}$	Input features	$[H \times W \times C_{\text{in}}]$
$\mathcal{H}^{(t)}$	Hidden state	$[H \times W \times C_{\text{out}}]$
$\mathcal{C}^{(t)}$	Cell state	
$\mathcal{I}^{(t)}$	Input gate	
$\mathcal{F}^{(t)}$	Forget gate	
$\tilde{\mathcal{C}}^{(t)}$	Cell memory	
$\mathcal{O}^{(t)}$	Output gate	
Φ	Mapping function for higher-order RNN	-
M	order of higher-order RNN	$M \geq N$
N	Order of CTTD	
K	Initial filter size	$K^{(0)} = K$
$K^{(i)}$	Filter size in $\tilde{\mathcal{K}}^{(i)}$	
$C^{(i)}$	# channels in $\tilde{\mathcal{H}}^{(i)}$	$C^{(0)} = 4C_{\text{out}}$
$\mathcal{G}^{(i)}$	Factors in the CTTD	$[K^{(0)} \times K^{(0)} \times C^{(i)} \times C^{(i-1)}]$
D	Size of sliding window	$D = M - N + 1$
$\mathcal{P}^{(i)}$	Preprocessing kernel	$[D \times K \times K \times C_{\text{out}} \times C^{(i)}]$
$\tilde{\mathcal{H}}^{(i)}$	Pre-processed hidden state	$[H \times W \times C^{(i)}]$
$\mathcal{K}^{(i)}$	Weights for $\tilde{\mathcal{H}}^{(i)}$	$[K^{(i)} \times K^{(i)} \times C^{(i)} \times C^{(0)}]$

Table 4.1: **Table of notations.**

4.4.1 Extending ConvLSTM to Higher-orders

We can express a general higher-order ConvLSTM by combining several previous states when computing the gates for each step:

$$\left[\mathcal{I}^{(t)}; \mathcal{F}^{(t)}; \tilde{\boldsymbol{C}}^{(t)}; \mathcal{O}^{(t)}\right] = \sigma\left(\mathcal{W} * \mathcal{X}^{(t)} + \Phi\left(\mathcal{H}^{(t-1)}, \cdots, \mathcal{H}^{(t-N)}\right)\right). \tag{4.6}$$

1. The operations in Φ preserve the spatial structures in the hidden states $\mathcal{H}^{(t)}$'s, e.g., satisfy translation-equivariant property.

2. The size of the receptive field for $\mathcal{H}^{(t-i)}$ increases with i, the time gap from the current step ($i = 1, 2, \cdots, N$). In other words, the longer temporal correlation captured, the larger the considered spatial context should be.

3. Both space and time complexities grow *at most* linearly with the number of times steps N, i.e., $O(N)$.

Limitations of previous approaches. While it is possible to construct a function Φ by extending the linear function in Equation (4.4) or the polynomial function in Equation (4.5) to the tensor case, none of these extensions satisfy all three properties. While the polynomial function with tensor-train decomposition [37] meets requirement **(3)**, the operations do not preserve the spatial structures in the hidden states. On the other hand, augmenting the linear function with convolutions leads to a function:

$$\Phi\left(\mathcal{H}^{(t-1)}, \cdots, \mathcal{H}^{(t-N)};\ \mathcal{K}^{(1)}, \cdots, \mathcal{K}^{(N)}\right) = \sum\nolimits_{i=1}^{N} \mathcal{K}^{(i)} * \mathcal{H}^{(t-i)} \tag{4.7}$$

which does not satisfy requirement **(2)** if all $\mathcal{K}^{(i)}$ contain filters of the same size K. An immediate remedy is to expand $\mathcal{K}^{(i)}$ such that its filter size $K^{(i)}$ grows linearly in i. However, the function would require $O(N^3)$ space and computational complexity, violating the requirement **(3)**.

4.4.2 Designing an Effective and Efficient Higher-order ConvLSTM

In order to satisfy all three requirements **(1)-(3)** introduced above, and enable efficient learning/inference, we propose a novel *convolutional tensor-train decomposition* (CTTD) that leverages a tensor-train structure [35] to jointly express the convolutional kernels $\{\mathcal{K}^{(1)}, \cdots, \mathcal{K}^{(N)}\}$ in Equation (4.7) as a series of smaller factors $\{\mathcal{G}^{(1)}, \cdots, \mathcal{G}^{(N)}\}$ while maintaining their spatial structures.

Convolutional Tensor-Train module. Concretely, let $\mathcal{K}^{(i)}$ be the i-th kernel in Equation (4.7), of size $[K^{(i)} \times K^{(i)} \times C^{(i)} \times C^{(0)}]$, where $K^{(i)} = i[K^{(1)} - 1] + 1$ is the filter size that increases linearly with i; $K^{(1)}$ is the initial filter size; $C^{(i)}$ is the number of channels in $\mathcal{H}^{(t-i)}$; and $C^{(0)}$ is the number of channels for the output of the function Φ (thus $C^{(0)} = 4 \times C_{\text{out}}$, where C_{out} is the number of channels of the higher-order ConvLSTM). The CTTD factorizes $\mathcal{K}^{(i)}$ using a subset of factors $\{\mathcal{G}^{(1)}, \cdots, \mathcal{G}^{(i)}\}$ up to index i such that

$$\mathcal{K}_i^{(:,:,c_i,c_0)} = \mathsf{CTTD}\left(\{\mathcal{G}^{(j)}\}_{j=1}^{i}\right) \triangleq \sum_{c_{i-1}=1}^{C^{(i-1)}} \cdots \sum_{c_1=1}^{C^{(1)}} \mathcal{G}^{(i)}_{:,:,c_i,c_{i-1}} * \cdots * \mathcal{G}^{(1)}_{:,:,c_1,c_0}, \tag{4.8}$$

where $\mathcal{G}^{(i)}$ has size $[K^{(1)} \times K^{(1)} \times C^{(i)} \times C^{(i-1)}]$. The number of factors N is known as the *order of the decomposition*, and the *ranks of the decomposition* $\{C^{(1)}, \cdots, C^{(N-1)}\}$ are the number of channels of the convolutional kernels.

Notice that the same factors $\{\mathcal{G}^{(1)}, \cdots, \mathcal{G}^{(N)}\}$ is used to construct all convolutional kernels $\{\mathcal{K}^{(1)}, \cdots, \mathcal{K}^{(N)}\}$, such that the total number of parameters grows linearly in N. In fact, the convolutional kernel $\mathcal{K}^{(i+1)}$ can be recursively constructed as $\mathcal{K}^{(i)} = \mathcal{G}^{(i)} * \mathcal{K}^{(i-1)}$ with $\mathcal{K}^{(1)} = \mathcal{G}^{(1)}$ and $\mathcal{K}_i^{(:,:,c_i,c_0)} = \sum_{c_{i-1}} \mathcal{G}_i^{(:,:,c_i,c_{i-1})} * \mathcal{K}_{i-1}^{(:,:,c_{i-1},c_0)}, \forall i \geq 2$. This results in a *convolutional tensor-train module* that we use for function Φ in Equation (4.7):

$$\begin{aligned}\Phi &= \mathsf{CTT}\big(\mathcal{H}^{(t-1)}, \cdots, \mathcal{H}^{(t-N)};\ \mathcal{G}^{(1)}, \cdots, \mathcal{G}^{(N)}\big) \\ &\triangleq \sum_{i=1}^{N} \mathsf{CTTD}\big(\{\mathcal{G}^{(j)}\}_{j=1}^{i}\big) * \mathcal{H}^{(t-i)}.\end{aligned} \tag{4.9}$$

In [38], we show that the computation of Equation (4.9) can be done in linear time $O(N)$, thus the construction of CTT satisfies all requirements **(1)-(3)**.

Preprocessing module. In Equation (4.9), we use the raw hidden states $\mathcal{H}^{(t)}$ as inputs to CTT. This design has two limitations: **(a)** The number of past steps in CTT (i.e., the order of the higher-order ConvLSTM) is equal to the number of factors in CTTD (i.e., the order of the tensor decomposition), which both equal to N. It is prohibitive to use a long history, as a large tensor order leads to gradient vanishing/exploding problem in computing Equation (4.9); **(b)** All the ranks $C^{(i)}$ are equal to the number of channels in $\mathcal{H}^{(t)}$, which prevents the use of lower-ranks to further reduce the model complexity.

To address both issues, we develop a preprocessing module to reduce the number of steps and channels in previous hidden states before CTT. Suppose the number of steps M is no less than the tensor order N (i.e., $M \geq N$), the preprocessing collects the neighboring steps with a sliding window and reduce it into an intermediate result with $C^{(i)}$ channels:

$$\tilde{\mathcal{H}}^{(i)} = \mathcal{P}^{(i)} * \left[\mathcal{H}^{(t-i)}; \cdots ; \mathcal{H}^{(t-i+N-M)}\right] \tag{4.10}$$

where $\mathcal{P}^{(i)}$ represents a convolutional layer that maps the concatenation $[\cdot]$ into $\tilde{\mathcal{H}}^{(i)}$.

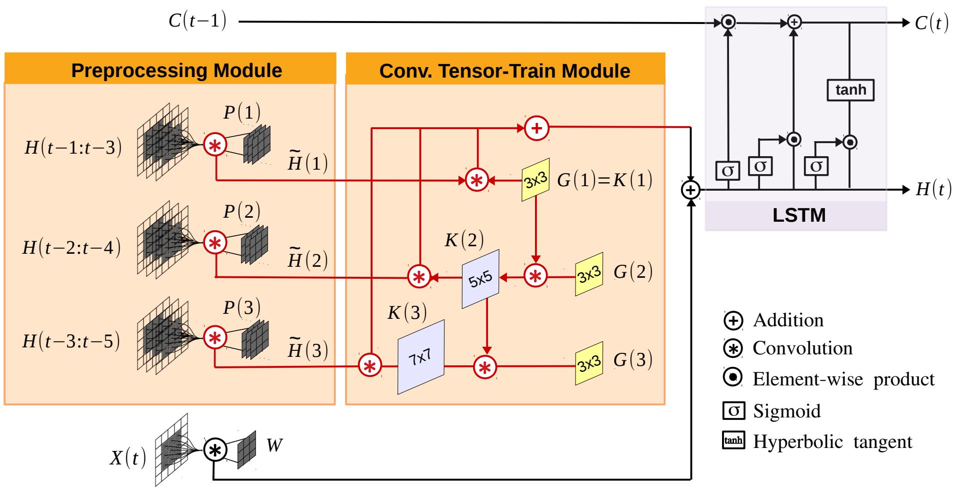


Figure 4.1: **Convolutional Tensor-Train LSTM**. The *preprocessing module* first groups the previous hidden states into overlapping sets with a sliding window and subsequently reduces the number of channels in each group using a convolutional layer. Next, the *convolutional tensor-train module* takes the results, aggregates their spatio-temporal information, and computes the gates for the LSTM update. The diagram visualizes a Conv-TT-LSTM with one channel. When Conv-TT-LSTM has multiple channels, the addition also accumulates the results from all channels.

Convolutional Tensor-Train LSTM. By combining all the above modules, we obtain our proposed Conv-TT-LSTM, illustrated in Figure 4.1 and expressed as:

$$\left[\mathcal{I}^{(t)}; \mathcal{F}^{(t)}; \tilde{\mathcal{C}}^{(t)}; \mathcal{O}^{(t)}\right] = \sigma\left(\mathcal{W} * \mathcal{X}^{(t)} + \mathsf{CTT}\left(\tilde{\mathcal{H}}^{(1)}, \cdots, \tilde{\mathcal{H}}^{(N)};\ \mathcal{G}^{(1)}, \cdots, \mathcal{G}^{(N)}\right)\right) \tag{4.11}$$

This final implementation has several advantages: it drastically reduces the number of parameters and makes the higher-order ConvLSTM even more compact than the first-order ConvLSTM. The low-rank constraint acts as an implicit regularizer, leading to more generalizable models. Finally, the tensor-train structure inherently encodes the correlations resulting from the natural flow of time [37].

4.5 Experimental Results

In this section, we empirically evaluate our approach on two different tasks — video prediction and early activity recognition and find out it outperforms existing methods.

4.5.1 Experimental Setup

Evaluation. For *video prediction*, the model predicts every pixel in the frame. We test our models on the KTH human action dataset [98] with resolution 128×128 and the Moving-MNIST-2 dataset [76] with resolution 64×64. We train all models to predict 10 future frames given 10 input frames and tested to predict $10 - 40$ frames recursively. For *early activity recognition*, we evaluate our approach on the Something-Something V2 dataset. Following [81], we used the subset of 41 categories defined by [99] (Table 7). We train the prediction model to predict the following 10 frames given $25\% - 50\%$ of frames and jointly classify the activity using the learned representations of the prediction model.

Model architecture. In all video prediction experiments, we use 12 recurrent layers. For early activity recognition, we follow the framework in [81]. The prediction model consists of a two-layer 2D-convolutional encoder and decoder with eight recurrent layers between them. The classifier, which contains two 2D-convolutional layers and one fully-connected layer, takes the last recurrent layer's output and returns a label output. We explain the detailed architecture in Section 4.7.2.

Hyper-parameter selection. We validate the hyper-parameters of our Conv-TT-LSTM through a wide grid search on the validation set. Specifically, we consider a base filter size $S = 3, 5$, decomposition order $N = 1, 2, 3, 5$, tensor ranks $C^{(i)} = 4, 8, 16$, and number of hidden states $M = 1, 3, 5$. Section 4.7.3 contains the details of our hyper-parameter search.

Efficient implementation. Two versions of the implementation are available: the original and the optimized version. In the optimized version, we use multi-threading to accelerate our implementation using the NVIDIA apex library [100]. Furthermore, we adopt fused kernels to speed up the Adam optimizer [8] and TorchScript to fuse multiplications and additions. Lastly, we use affinity binding to reduce the communication cost between GPUs and CPUs. These modifications speed up training up to four times. Both versions are available online: `https://github.com/NVlabs/conv-tt-lstm`.

4.5.2 Analysis of Empirical Results

Multi-frame Video prediction: KTH action dataset. First, we test our model on human action videos. In Table 4.2, we report the evaluation on both 20 and 40 frames prediction. Figure 4.3 (right) shows the model comparisons with SSIM v.s. LPIPS and the model size. **(1)** Our model is consistently better than the ConvLSTM baseline for both 20 and 40 frames prediction. **(2)** While our proposed Conv-TT-LSTMs achieve lower SSIM value compared to the state-of-the-art models in 20 frames prediction, they outperform all previous models in LPIPS for both 20 and 40 frames prediction. Figure 4.3 (right) shows a visual comparison of our model, ConvLSTM baseline, PredRNN++ [80], and E3D-LSTM [81]. Our model produces sharper frames and better preserves the human silhouettes, although slight artifacts exist over time (shifting). We believe such artifacts may vanish using a different loss function or additional techniques that help per-pixel motion prediction.

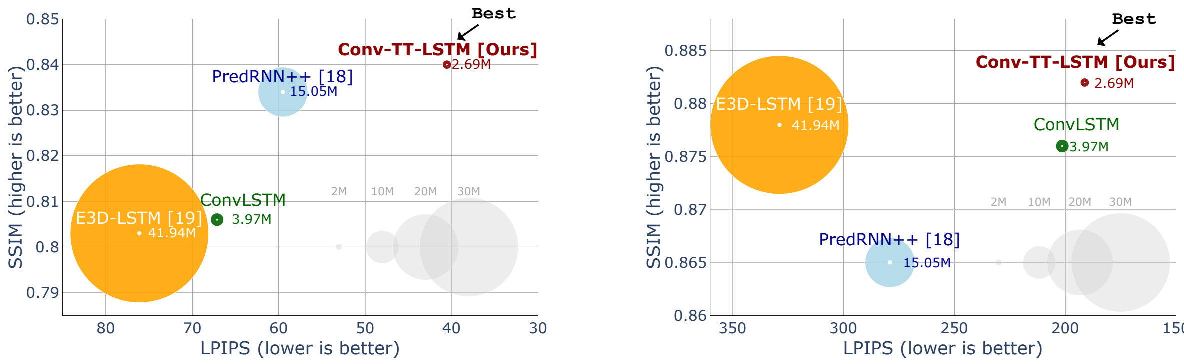


Figure 4.2: **SSIM v.s. LPIPS scores on Moving-MNIST-2 (left) and KTH action datasets (right).** The bubble size is the model size. The higher SSIM scores and lower LPIPS scores, the better quality of predictions. On both datasets and for both metrics, our approach reaches a significantly better performance than other methods while having only a fraction of the parameters.

Multi-frame video prediction: Moving-MNIST-2 dataset. We also evaluate our model on the Moving-MNIST-2 dataset and show that our model predicts the digits almost correctly in terms of structure and motion (See Figure 4.3). Table 4.2 reports the average statistics for 10 and 30 frames prediction, and Figure 4.2 (left) shows the comparisons of SSIM v.s. LPIPS and the model size. Our Conv-TT-LSTM models **(1)** consistently outperform the ConvLSTM baseline for both 10 and 30 frames prediction *with fewer parameters*; **(2)** outperform previous approaches in terms of SSIM and LPIPS (especially on 30 frames prediction), *with less than one fifth of the model parameters.*

We reproduce the PredRNN++ [80] and E3D-LSTM [81] from the source code [101,

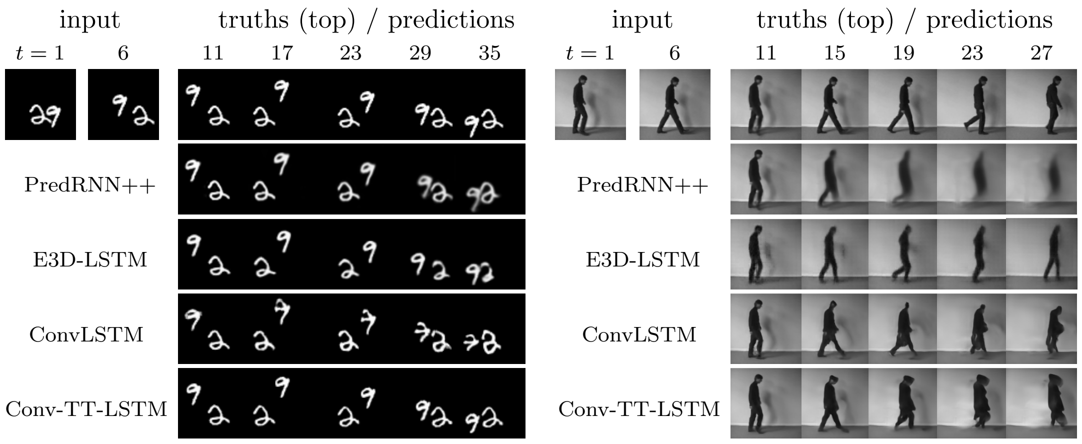


Figure 4.3: **30 frames prediction on Moving-MNIST (left)**, and **20 frame prediction on KTH action datasets (right)** given 10 input frames. The first frames ($t = 1, 11$) are animations. Adobe reader is required to view the animation. Our method generates both semantically plausible and visually crisp images, compared to other approaches.

	Method	**(10 → 20)**			**(10 → 40)**			**Complexities**		
		PSNR	**SSIM**	**LPIPS**	**PSNR**	**SSIM**	**LPIPS**	**# Params.**	**# FLOPS**	**Time(m)**
KTH action	ConvLSTM [78]	23.58	0.712	-	22.85	0.639	-	7.58M	106.6G	-
	MCNET [93]	25.95	0.804	-	-	-	-	-	-	-
	PredRNN++ [80] (retrained [101])	28.62	0.888	228.9	26.94	0.865	279.0	15.05M	-	-
	E3D-LSTM [81] (retrained [102])	27.92	0.893	298.4	26.55	0.878	328.8	41.94M	-	-
	ConvLSTM (baseline)	28.21	0.903	137.1	26.01	0.876	201.3	3.97M	55.83G	28.9
	Conv-TT-LSTM (Ours)	28.36	0.907	133.4	26.11	0.882	191.2	2.69M	37.83G	74.8
	Method	**(10 → 10)**			**(10 → 30)**			**Complexities**		
		MSE	**SSIM**	**LPIPS**	**MSE**	**SSIM**	**LPIPS**	**# Params.**	**# FLOPS**	**Time(m)**
Moving-MNIST	ConvLSTM [78]	25.22	0.713	-	38.13	0.595	-	7.58M	30.32G	-
	VPN [103]	15.65	0.870	-	31.64	0.620	-	-	-	-
	PredRNN++ [80] (retrained [101])	10.29	0.913	59.51	20.53	0.834	139.9	15.05M	-	-
	E3D-LSTM [81] (pretrained [102])	20.23	0.869	76.12	32.37	0.803	150.3	41.94M	-	-
	ConvLSTM (baseline)	18.17	0.882	67.13	33.08	0.806	140.1	3.97M	15.88G	14.8
	Conv-TT-LSTM (Ours)	**12.96**	**0.915**	**40.54**	**25.81**	**0.840**	**90.38**	2.69M	10.76G	29.6

Table 4.2: **Evaluation of multi-steps prediction on the KTH action (top) and Moving-MNIST-2 (bottom) datasets.** Higher PSNR/SSIM and lower MSE/LPIPS values indicate better predictive results. # of FLOPs denotes the multiplications for one-step prediction per sample, and Time(m) represents the clock time (in minutes) required by training the model for one epoch (10,000 samples).

102]. We find that **(1)** PredRNN++ and E3D-LSTM output vague and blurry digits in long-term prediction (especially after 20 steps); **(2)** our Conv-TT-LSTM produces sharp and realistic digits over all steps. An example of visual comparison is shown in Figure 4.3.

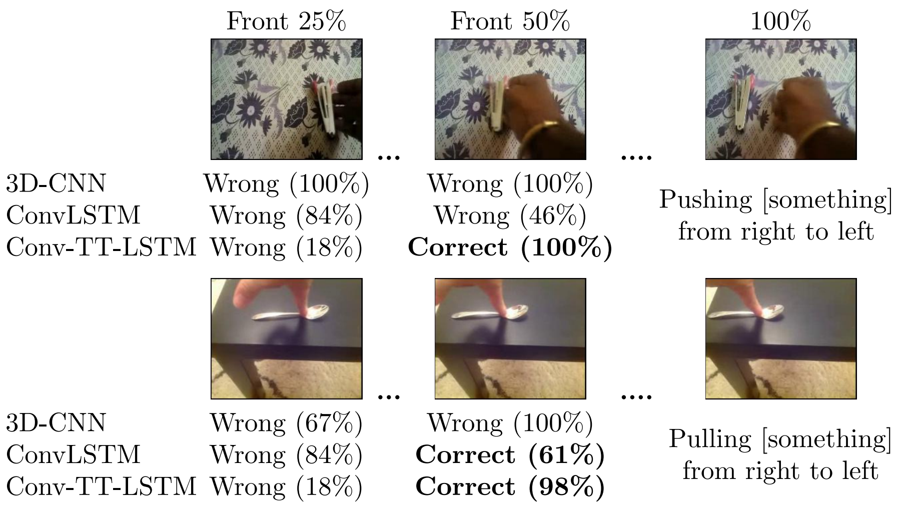


Figure 4.4: **Examples of early activity recognition** on the Something-Something V2 dataset. $(\cdot)$ indicates the confidence of Correct/Wrong prediction.

Model	**Input**	**Dropping**	**Holding**	**MovingLR**	**MovingRL**	**Picking**	**Poking**	**Pouring**	**Putting**	**Showing**	**Tearing**
3D-CNN		8.5	4.7	25.8	32.6	7.5	2.9	1.9	10.3	14.0	14.5
ConvLSTM	25%	8.5	**7.0**	27.4	38.8	16.8	5.9	1.9	12.0	7.0	21.2
Conv-TT-LSTM		**11.5**	4.7	**33.9**	**40.8**	**16.8**	**5.9**	**5.7**	**13.6**	**20.9**	**26.0**
3D-CNN		14.6	11.6	45.2	57.1	16.8	8.8	11.3	17.4	16.3	26.0
ConvLSTM	50%	21.5	7.0	43.5	47.0	15.9	**14.7**	5.7	20.7	16.3	30.8
Conv-TT-LSTM		**24.6**	**11.6**	**56.5**	**57.1**	**27.6**	5.9	**13.2**	**25.5**	**37.2**	**46.2**

Table 4.3: **Per-activity accuracy of early activity recognition on the Something-Something V2 dataset.** We used 41 categories for training. For per-activity evaluation, the 41 categories are grouped into ten similar activities [99]. Our model substantially outperforms 3D-CNN and ConvLSTM on long-term dynamics such as Moving or Tearing while achieving marginal improvement on static activities such as Holding or Pouring.

Early activity recognition: Something-Something V2 dataset. To demonstrate that our Conv-TT-LSTM-based prediction model can learn efficient representations from videos, we evaluate the models on early activity recognition on the Something-Something V2 dataset. In this task, a model only observes a small fraction ($25\% - 50\%$) of frames and learns to predict future frames. Based on the learned representations of the beginning frames, the model predicts the full video's overall activity. Intuitively, the learned representation encodes the future information for frame prediction, and the better the quality of the representations, the higher the classification accuracy. As shown in Table 4.3 and Table 4.4 our Conv-TT-LSTM model consistently outperforms the baseline ConvLSTM and 3D-CNN models as well as E3D-LSTM [81] under different ratio of input frames. Our experimental setup and architecture follow [81].

Model	**Input Ratio**	
	Front 25%	**Front 50%**
3D-CNN*	9.11	10.30
E3D-LSTM* [81]	14.59	22.73
3D-CNN	13.26	20.72
ConvLSTM	15.46	21.97
Conv-TT-LSTM (ours)	**19.53**	**30.05**

Table 4.4: **Early activity recognition on the Something-Something V2 dataset** using 41 categories as [81]. (*) indicates the result by [81].

	MSE($\times 10^{-3}$)	**SSIM**	**LPIPS**
CTTD with 1×1 filters (similar to standard TTD)			
single order	31.52	0.810	148.7
order 3	34.84	0.800	151.2
CTTD with 5×5 filters			
single order	33.08	0.806	140.1
order 3	**28.88**	**0.831**	**104.1**

Table 4.5: **Ablation studies of Conv-TT-LSTM on the Moving-MNIST-2 dataset**. The models are tested for 10 to 30 frames prediction.

4.6 Discussions

In this section, we further justify the importance of our proposed modules, namely the *convolutional tensor-train decomposition* (CTTD) and the *preprocessing module*. We also explain the computational complexity of our model and the difficulties of spatio-temporal learning with Transformer [4].

Importance of encoding higher-order correlations in a convolutional manner. Two key differences between CTTD and existing low-rank decompositions are *higher-order* *decomposition* and *convolutional operations*. To verify their impact, we compare the performance of two ablated models against our CTTD-base model in Table 4.5. The single order means that the higher-order model is replaced with a first-order model (tensor order = 1). By replacing 5×5 filters to 1×1, the convolutions are removed, and the CTTD reduces to a standard tensor-train decomposition. The results show a decrease in performance: the ablated models achieve similar performances of ConvLSTM baseline at best, demonstrating that both higher-order models and convolutional operations are necessary.

Importance of the preprocessing module. There could be other ways to incorporate previous hidden states into the CTT module. One is to reduce the number of channels while keeping the number of steps; the other is to reuse all previous states' concatenation for each input to CTT. Unfortunately, the former fails due to the gradient vanishing/exploding problem. In contrast, the latter has a tube-shaped receptive field that fails to distinguish more recent steps and the ones from the remote history.

Computational complexity. Table 4.2 provides the number of FLOPS for all models. Our Conv-TT-LSTM model has lower computational complexity and fewer parameters than other models under comparison. The efficiency is made possible by a linear algorithm for the CTT module in Equation (4.9), derived in [38]. Notice that a lower FLOPS does not necessarily lead to faster computation since the convolutional tensor-train module is naturally sequential, as shown in Table 4.2.

Transformer for spatio-temporal learning. Transformer [4] is a popular predictive model based on the attention mechanism, which is very successful in natural language processing [104]. However, the Transformer has prohibitive limitations on video understanding due to excessive needs for both memory and computation. While language modeling only involves temporal attention, video understanding requires attention to spatial dimensions as well [105]. Moreover, since the attention mechanism does not preserve the spatial structures by design, Transformer additionally requires auxiliary components, including an autoregressive module and multi-resolution upscaling when applied to spatial data [105, 106, 107]. Our Conv-TT-LSTM incorporates a broad spatio-temporal context, but with a compact, efficient and structure-preserving operator without additional components.

4.7 Supplemental Materials for Empirical Studies

The supplementary material provides implementation details for all experiments and performs additional ablation studies of our model. We demonstrate that our Conv-TT-LSTM model outperforms regular ConvLSTM with varying settings.

4.7.1 Preprocessing Module

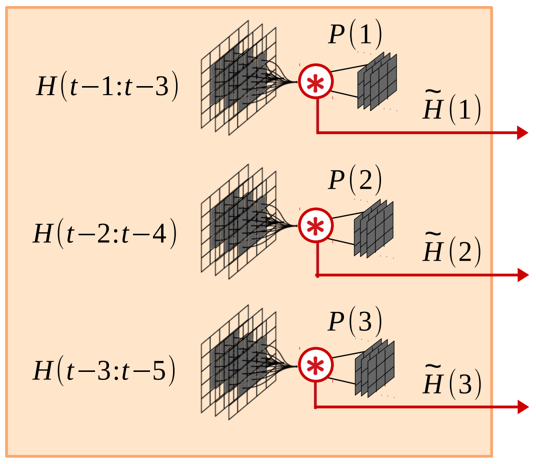
In Section 4.4.2, we use a sliding window to concatenate consecutive states in the preprocessing module (Equation (4.10)). In the discussion (Section 4.6), we argue that other possible approaches are less effective in preserving spatio-temporal structure than our *sliding window approach*. Here, we discuss an alternative approach previously proposed for non-convolutional higher-order RNN [37], which we name as *fixed window approach*. We will compare these two approaches in computational complexity, temporal structure-preserving, and predictive performance.

Fixed window approach. With fixed window approach, M previous steps $\{\mathcal{H}^{(t-1)}, \cdots, \mathcal{H}^{(t-M)}\}$ are first concatenated into a single tensor, which is then repeatedly mapped to N inputs $\{\tilde{\mathcal{H}}^{(1)}, \cdots, \tilde{\mathcal{H}}^{(N)}\}$ to the CTT module.

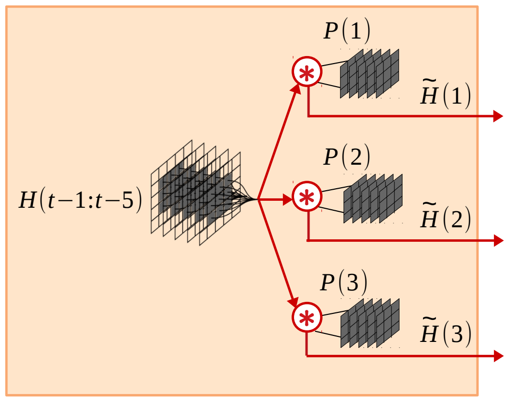
$$\textbf{Fixed Window (FW):} \quad \tilde{\mathcal{H}}^{(i)} = \mathcal{P}^{(i)} * \left[\mathcal{H}^{(t-1)}; \cdots; \mathcal{H}^{(t-N)}\right] \tag{4.12a}$$

$$\textbf{Sliding Window (SW):} \quad \tilde{\mathcal{H}}^{(i)} = \mathcal{P}^{(i)} * \left[\mathcal{H}^{(t-i)}; \cdots; \mathcal{H}^{(t-i+N-M)}\right] \tag{4.12b}$$

For comparison, we list both equations for the fixed window approach and the sliding window approach. We also illustrate these two approaches in Figure 4.5.



(a) Sliding window approach

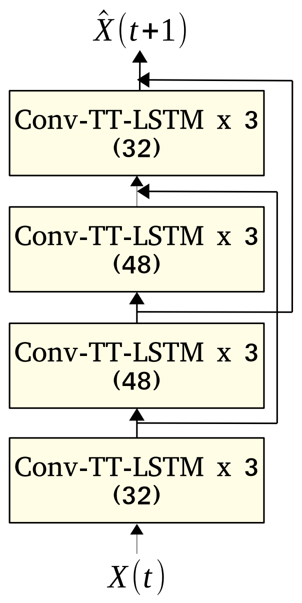


(b) Fixed window approach (alternative)

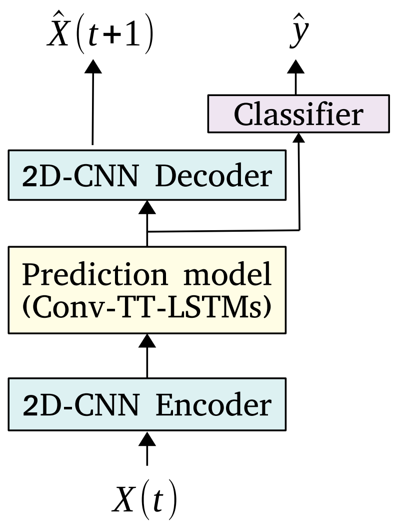
Figure 4.5: **Variations of preprocessing modules.**

4.7.2 Model Architectures

Multi-frame video prediction. All experiments use a 12-layers ConvLSTM / Conv-TT-LSTM with 32 channels for the first and last 3 layers, and 48 channels for the 6 layers in the middle. A convolutional layer is applied on top of all recurrent layers to compute the predicted frames, followed by an extra sigmoid layer for the KTH action dataset. Following [91], we add two skip connections performing concatenation over channels between (3, 9) and (6, 12) layers. An illustration of the network architecture is included in Figure 4.6a. All convolutional kernels are initialized by Xavier's normalized initializer [108] and initial hidden/cell states are initialized as zeros.



(a) Prediction model



(b) Recognition model

Figure 4.6: **Network architectures** for video prediction and early activity recognition.

Early activity recognition. Following [81], the network consists of four modules: a 2D-CNN encoder, a video prediction network, a 2D-CNN decoder and a 3D-CNN classifier, as showed in Figure 4.6b. **(1)** The 2D-CNN encoder has two 2-strided 2D-convolutional layers with 64 channels, which reduce the resolution from 224×224 to 56×56. **(2)** The 2D-CNN decoder contains two 2-strided transposed 2D-convolutional layers with 64 channels, which restore the resolution from 56×56 to 224×224. **(3)** The video prediction network is miniature version of Figure 4.6a, where the number of layers in each block is reduced to 2. In the experiments, we evaluate three realizations of each layer: ConvLSTM, Conv-TT-LSTM or masked 3D-convolutional layer. **(4)** The 3D-CNN classifier takes the last 16 frames from the input and predicts a label for the 41 categories. The classifier contains two 2-strided 3D-convolutional layers with 128 channels, each followed by a 3D-pooling layer. These layers reduce the resolution from 56×56 to 7×7, and the output is fed into a two-layer perceptron with 512 units for a predictive label.

4.7.3 Training, Evaluation, and Hyper-parameters Selection

Training strategy. We argue for a careful choice of learning scheduling and gradient clipping to facilitate training. Specifically, various *learning scheduling techniques*, including learning rate decay, scheduled sampling [109], and curriculum learning with varying weighting factors, are added during training. **(1)** For *video prediction*, we use learning rate decay along with scheduled sampling, where scheduled sampling starts if the model does not improve for a few epochs in terms of validation loss. **(2)** For *early activity recognition*, we combine learning rate decay with weighting factor decay, where the weighting factor decreases linearly $\lambda := \max(\lambda - \epsilon, 0)$ on the plateau. **(3)** We also found *gradient clipping* essential for higher-order models. We train All models with Adam optimizer [8]. In the initial experiments, we found that our models are unstable at a high learning rate $1e^{-3}$, but learn poorly at a low learning rate of $1e^{-4}$. Consequently, we use gradient clipping with a learning rate of $1e^{-3}$, with a clipping value of 1 for all experiments.

Evaluation metrics. We use two traditional metrics, SSIM [110] and MSE (or PSNR), and a recently proposed deep-learning-based metric LPIPS [111], which measures the similarity between features from different layer. Since MSE (or PSNR) is based on pixel-wise difference, it favors vague and blurry predictions — thus, it is not a proper measurement of perceptual similarity. While SSIM was initially proposed to address the problem, [111] shows that their proposed LPIPS metric aligns better with human perception.

Hyper-parameters selection. Table 4.6 summarizes our search values for different hyper-parameters for Conv-TT-LSTM. **(1)** For filter size K, we found models with larger filter size $K = 5$ consistently outperform the ones with $K = 3$. **(2)** For learning rate, we found that our models are unstable at a high learning rate such as 10^{-3}, but learn poorly at a low learning rate 10^{-4}. Consequently, we use gradient clipping with learning rate 10^{-3}, with clipping value 1 for all experiments. **(3)** While the performance typically increases as the order grows, the model suffers gradient instability in training with a high order, e.g., $N = 5$. Therefore, we choose the order $N = 3$ for all Conv-TT-LSTM models. **(4)(5)** For small ranks $C^{(i)}$ and steps M, the performance increases monotonically with $C^{(i)}$ and M. But the performance stays on plateau when we further increase them, therefore we settle down at $C^{(i)} = 8, \forall i$ and $M = 5$ for all experiments.

Filter size K	Learning rate	Order of CTTD N	Ranks of CTTD $\mathcal{C}^{(i)}$	Time steps M
{3, 5}	$\{10^{-4}, 5 \times 10^{-4}, 10^{-3}\}$	$\{1, 2, 3, 5\}$	$\{4, 8, 16\}$	$\{1, 3, 5\}$

Table 4.6: **Hyper-parameters search values** for Conv-TT-LSTM experiments.

4.7.4 Datasets

Moving-MNIST-2 dataset. We generate the Moving-MNIST-2 dataset by moving two digits with size 28×28 in the MNIST dataset within a 64×64 black canvas. These digits are placed at a random initial location, move with constant velocity in the canvas, and bounce when they reach the boundary. Following [80], we generate 10,000 videos for training, 3,000 for validation, and 5,000 for test with default parameters in the generator[1].

KTH action dataset. The KTH action dataset [98] contains videos of 25 individuals performing six types of actions on a simple background. Our experimental setup follows [80], which uses persons 1-16 for training and 17-25 for testing, and we resize each frame to 128×128 pixels. We train all our models to predict 10 frames given 10 input frames. We randomly select 20 contiguous frames from the training videos as a sample and group every 10,000 samples into one epoch to apply the learning strategy, as explained at the beginning of this section.

Something-Something V2 dataset. The Something-Something V2 dataset [99] is a benchmark for activity recognition, which can be download online[2]. Following [81], we use the official subset with 41 categories that contain 55111 training videos and 7518 test videos. The video length ranges between 2 and 6 seconds with 24 frames per second (fps). We reserve 10% of the training videos for validation and use the remaining 90% for optimizing the models.

4.7.5 Ablation Studies

With the ablation studies, we show that our proposed Conv-TT-LSTM consistently improves the performance of ConvLSTM, regardless of the architecture, loss function, and

[1]The Python code for Moving-MNIST-2 generator is publicly available online in [112].
[2]`https://20bn.com/datasets/something-something`

Model		Layers 4	12	Sched. TF	SS	Loss ℓ_1	$\ell_1 + \ell_2$	(10 → 30) MSE	SSIM	LPIPS	Params.
ConvLSTM	-	✓	×	×	✓	×	✓	37.19	0.791	184.2	11.48M
Conv-TT-LSTM	FW							**31.46**	**0.819**	**112.5**	**5.65M**
ConvLSTM	-	×	✓	✓	×	×	✓	33.96	0.805	184.4	3.97M
Conv-TT-LSTM	FW							**30.27**	**0.827**	**118.2**	**2.65M**
ConvLSTM	-	×	✓	×	✓	✓	×	36.95	0.802	135.1	3.97M
Conv-TT-LSTM	FW							**34.84**	**0.807**	**128.4**	**2.65M**
ConvLSTM	-	×	✓	×	✓	×	✓	33.08	0.806	140.1	3.97M
Conv-TT-LSTM	FW							**28.88**	**0.831**	**104.1**	**2.65M**
Conv-TT-LSTM	SW	×	✓	×	✓	×	✓	25.81	**0.840**	**90.38**	2.69M

Table 4.7: **Evaluation of ConvLSTM and our Conv-TT-LSTM under ablated settings.** In this table, FW stands for *fixed window approach*, SW stands for *sliding window approach*; For learning scheduling, TF denotes *teaching forcing* and SS denotes *scheduled sampling*. The experiments show that **(1)** our Conv-TT-LSTM is able to improve upon ConvLSTM under all settings; **(2)** Our current learning approach is optimal in the search space; **(3)** The sliding window approach outperforms the fixed window one under the optimal experimental setting.

learning schedule used. Specifically, we perform three ablation studies on our experimental setting, by **(1)** Reducing the number of layers from 12 layers to 4 layers (same as [78] and [80]); **(2)** Changing the loss function from $\mathcal{L}_1 + \mathcal{L}_2$ to $\mathcal{L}_1$ only; and **(3)** Disabling the scheduled sampling and use teacher forcing during training process. We compare the performance of our proposed Conv-TT-LSTM against the ConvLSTM baseline in these ablated settings, Table 4.7. The results show that our proposed Conv-TT-LSTM consistently outperforms ConvLSTM in all settings, i.e., the Conv-TT-LSTM model improves upon ConvLSTM in a broad range of setups, which is not limited to the specific setting we adopt in this chapter. These ablation studies further show that our configuration is optimal for predictive learning in Moving-MNIST-2 dataset.

4.8 Conclusion

In this chapter, we proposed a fully-convolutional higher-order LSTM model for spatio-temporal data. To make the approach computationally and memory feasible, we proposed a novel convolutional tensor-train decomposition that jointly parameterizes the convolutions and naturally encodes temporal dependencies. The result is a compact model that outperforms prior work on video prediction, including something-something V2, moving-MNIST-2, and KTH action datasets.

Chapter 5: Efficient Learning of Bayesian Quantized Networks

5.1 Overview

A Bayesian approach to deep learning considers the network's parameters as random variables and seeks to infer their posterior distribution given the training data. Models trained this way, called *Bayesian neural networks* (BNNs) [113], in principle have well-calibrated uncertainties when they make predictions, which is essential in scenarios such as active learning and reinforcement learning [114]. Furthermore, the posterior distribution over the model parameters provides valuable information to compress a neural network.

There are three main challenges in using BNNs: **(1) Intractable posterior:** Computing and storing the exact posterior distribution over the network weights is intractable due to the complexity and high-dimensionality of deep networks. **(2) Prediction:** Performing a forward pass (a.k.a. as *probabilistic propagation*) in a BNN to compute a prediction for input cannot be performed exactly since the distribution of hidden activations at each layer is intractable to compute. **(3) Learning:** The classic *evidence lower bound* (ELBO) learning objective for training BNNs is not amenable to backpropagation as the ELBO is not an explicit function of the output of probabilistic propagation.

These challenges are typically addressed either by making simplifying assumptions about the distributions of the parameters and activations, or by using sampling-based approaches, which are expensive and unreliable (likely to overestimate the uncertainties in predictions). Our goal is to propose a **sampling-free** method which uses probabilistic propagation to learn BNNs deterministically.

A seemingly unrelated area of deep learning research is that of *quantized neural networks* (QNNs), which offer advantages of computational and memory efficiency compared to continuous-valued models. QNNs, like BNNs, face challenges in training, though for different reasons: **(4.1)** The non-differentiable activation function is not amenable to backpropagation. **(4.2)** Gradient updates cease to be meaningful since the model parameters in QNNs are coarsely quantized.

In this work, we combine the ideas of BNNs and QNNs in a novel way that addresses the challenges mentioned above **(1)(2)(3)(4)** in training both models. We propose *Bayesian quantized networks* (BQNs), models that (like QNNs) have quantized parameters and activations over which they learn (like BNNs) categorical posterior distributions. BQNs have several appealing properties:

- BQNs solve challenge **(1)** by using categorical distributions for their parameters.
- BQNs can be trained via sampling-free backpropagation and stochastic gradient ascent of a differentiable ELBO, which addresses challenges **(2)**, **(3)**, and **(4)**.
- BQNs leverage efficient tensor operations for probabilistic propagation, further addressing challenge **(2)**. We show the equivalence between probabilistic propagation in BQNs and tensor contractions [33], and introduce a rank-1 CP tensor decomposition (mean-field approximation) that speeds up the forward pass in BQNs.
- BQNs provide a tunable trade-off between computational resource and model com-

plexity: using a fine quantization allows for more complex distribution at the cost of more computation.

- Sampling from BQNs provides an alternative way to obtain deterministic QNNs.

In our experiments, we demonstrate the expressive power of BQNs. We show that BQNs trained using our sampling-free method have much better-calibrated uncertainty compared with the state-of-the-art *Bootstrap ensemble of quantized neural networks* (E-QNN) trained by [115]. More impressively, our trained BQNs achieve comparable log-likelihood against Gaussian *Bayesian neural network* (BNN) trained with *stochastic gradient variational Bayes* (SGVB) [116] (the performance of Gaussian BNNs are expected to be better than BQNs since they allow for continuous random variables). We further verify that BQNs can be easily used to compress (Bayesian) neural networks and obtain deterministic QNNs. Finally, we evaluate the effect of mean-field approximation in BQNs by comparing with its Monte-Carlo realizations, where no approximation is used. We show that our sampling-free probabilistic propagation achieves similar accuracy and log-likelihood — justifying the use of mean-field approximation in BQNs.

Contributions. In summary, we make the following contributions:

1. We propose an alternative *evidence lower bound (ELBO)* for Bayesian neural networks such that optimization of the variational objective is compatible with the backpropagation algorithm.

2. We introduce Bayesian quantized networks (BQNs), establish a duality between BQNs and hierarchical tensor networks, and show prediction a BQN is equivalent

to a series of tensor contractions.

3. We derive a sampling-free approach for both learning and inference in BQNs using probabilistic propagation (analytical inference), achieving better-calibrated uncertainty for the learned models.

4. We develop a set of fast algorithms for efficient learning and prediction for BQNs.

5.2 Bayesian Neural Networks

Notation. We use bold letters such as $\boldsymbol{\theta}$ to denote random variables, and non-bold letters such as θ to denote their realizations. We abbreviate $\mathbf{Pr}[\boldsymbol{\theta} = \theta]$ as $\mathbf{Pr}[\theta]$ and use bold letters in an equation if the equality holds for *arbitrary* realizations. For example, $\mathbf{Pr}[\boldsymbol{x}, \boldsymbol{y}] = \mathbf{Pr}[\boldsymbol{y}|\boldsymbol{x}]\ \mathbf{Pr}[\boldsymbol{x}]$ means $\mathbf{Pr}[\boldsymbol{x} = x, \boldsymbol{y} = y] = \mathbf{Pr}[\boldsymbol{y} = y|\boldsymbol{x} = x]\ \mathbf{Pr}[\boldsymbol{x} = x], \forall x \in \mathcal{X}, y \in \mathcal{Y}$.

Problem setting. Given a dataset $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$ of N data points, we aim to learn a neural network with model parameters $\boldsymbol{\theta}$ that predict the output $y \in \mathcal{Y}$ based on the input $x \in \mathcal{X}$. **(1)** We first solve the **learning problem** to find an *approximate posterior distribution* $Q(\boldsymbol{\theta}; \phi)$ over $\boldsymbol{\theta}$ with parameters ϕ such that $Q(\boldsymbol{\theta}; \phi) \approx \mathbf{Pr}[\boldsymbol{\theta}|\mathcal{D}]$. **(2)** We then solve the **prediction problem** to compute the *predictive distribution* $\mathbf{Pr}[\boldsymbol{y}|x, \mathcal{D}]$ for arbitrary input $\boldsymbol{x} = x$ given $Q(\boldsymbol{\theta}; \phi)$. For notational simplicity, we will omit the conditioning on $\mathcal{D}$ and write $\mathbf{Pr}[\boldsymbol{y}|x, \mathcal{D}]$ as $\mathbf{Pr}[\boldsymbol{y}|x]$ in what follows.

To address the prediction and learning problems in BNNs, we analyze these models in their general form of *probabilistic graphical models*. Let $\boldsymbol{h}^{(l)}$, $\boldsymbol{\theta}^{(l)}$ and $\boldsymbol{h}^{(l+1)}$ denote the inputs, model parameters, and (hidden) outputs of the l-th layer respectively. We assume that **(1)** $\boldsymbol{\theta}^{(l)}$'s are *layer-wise independent*, i.e., $Q(\boldsymbol{\theta};\phi) = \prod_{l=0}^{L-1} Q(\boldsymbol{\theta}^{(l)};\phi^{(l)})$, and $\boldsymbol{h}^{(l)}$ follow the *Markovian property*, i.e., $\mathbf{Pr}[\boldsymbol{h}^{(l+1)}|\boldsymbol{h}^{(:l)},\boldsymbol{\theta}^{(:l)}] = \mathbf{Pr}[\boldsymbol{h}^{(l+1)}|\boldsymbol{h}^{(l)},\boldsymbol{\theta}^{(l)}]$.

5.2.1 The Prediction Problem in BNNs

Computing the predictive distribution $\mathbf{Pr}[\boldsymbol{y}|x,\mathcal{D}]$ in a BNN requires marginalizing over the random variable $\boldsymbol{\theta}$. The hierarchical structure of BNNs allows this marginalization to be computed in multiple steps sequentially. The predictive distribution $\mathbf{Pr}[\boldsymbol{h}^{(l+1)}|x]$ can be obtained from its preceding layer $\mathbf{Pr}[\boldsymbol{h}^{(l)}|x]$:

$$\underbrace{\mathbf{Pr}[\boldsymbol{h}^{(l+1)}|x]}_{P(\boldsymbol{h}^{(l+1)};\psi^{(l+1)})} = \int_{h^{(l)},\theta^{(l)}} \mathbf{Pr}[\boldsymbol{h}^{(l+1)}|h^{(l)},\theta^{(l)}]\ Q(\theta^{(l)};\phi^{(l)})\ \underbrace{\mathbf{Pr}[h^{(l)}|x]}_{P(h^{(l)};\psi^{(l)})}.dh^{(l)}d\theta^{(l)} \tag{5.1}$$

This iterative process to compute the predictive distributions layer-by-layer sequentially is known as *probabilistic propagation* [117, 118, 119]. With this approach, we need to explicitly compute and store each intermediate result $\mathbf{Pr}[\boldsymbol{h}^{(l)}|x]$ in its parameterized form $P(\boldsymbol{h}^{(l)};\psi^{(l)})$ (the conditioning on x is hidden in $\psi^{(l)}$, i.e., $\psi^{(l)}$ is a function of x). Therefore, probabilistic propagation is a deterministic process that computes $\psi^{(l+1)}$ as a function of $\psi^{(l)}$ and $\phi^{(l)}$, denoted as $\psi^{(l+1)} = g^{(l)}(\psi^{(l)},\phi^{(l)})$.

Challenge in probabilistic propagation. If the hidden variables $\boldsymbol{h}^{(l)}$'s are continuous, Equation (5.1) generally can not be evaluated in closed form as it is difficult to find a family of parameterized distributions P for $\boldsymbol{h}^{(l)}$ such that $\boldsymbol{h}^{(l+1)}$ remains in P under the operations of a neural network layer. Therefore most existing methods consider approximations at each layer of probabilistic propagation. In Section 5.2.1, we will show that this issue can be (partly) addressed if we consider the $\boldsymbol{h}^{(l)}$'s to be discrete random variables, as in a BQN.

5.2.2 The Learning Problem in BNNs

Objective function. A standard approach to finding a good approximation $Q(\boldsymbol{\theta}; \phi)$ is *variational inference*, which finds $\phi^\star$ such that the *KL-divergence* from $Q(\boldsymbol{\theta}; \phi)$ to $\mathbf{Pr}[\boldsymbol{\theta}|\mathcal{D}]$ is minimized i.e., $\phi^\star = \arg\min_\phi \mathbf{KL}(Q(\boldsymbol{\theta}; \phi)||\mathbf{Pr}[\boldsymbol{\theta}|\mathcal{D}])$. It is equivalent to maximize the negative KL-divergence, generally known as the *evidence lower bound* (ELBO) $\mathcal{L}(\phi)$:

$$\begin{gathered}\max_\phi \mathcal{L}(\phi) = -\mathbf{KL}(Q(\boldsymbol{\theta}; \phi)||\mathbf{Pr}[\boldsymbol{\theta}|\mathcal{D}]) = \sum_{n=1}^{N} \mathcal{L}_n(\phi) + \mathcal{R}(\phi), \\ \text{where } \mathcal{L}_n(\phi) = \mathbb{E}_Q\left[\log \mathbf{Pr}[y_n|x_n, \boldsymbol{\theta}]\right], \text{ and } \mathcal{R}(\phi) = \mathbb{E}_Q\left[\log\left(\mathbf{Pr}[\boldsymbol{\theta}]\right)\right] + H(Q).\end{gathered} \tag{5.2}$$

Sampling-free probabilistic backpropagation. Optimization in neural networks heavily relies on the gradient-based methods, where the partial derivatives $\partial\mathcal{L}(\phi)/\partial\phi$ of the objective $\mathcal{L}(\phi)$ w.r.t. the parameters ϕ are obtained by *backpropagation*. Formally, if the output produced by a neural network is given by a (sub-)differentiable function $g(\phi)$, and the objective $\mathcal{L}(g(\phi))$ is an *explicit* function of $g(\phi)$ (and not just an explicit function of ϕ), then the partial derivatives can be computed by chain rule:

$$\frac{\partial\mathcal{L}(g(\phi))}{\partial\phi} = \frac{\partial\mathcal{L}(g(\phi))}{\partial g(\phi)} \cdot \frac{\partial g(\phi)}{\partial\phi}. \tag{5.3}$$

The learning problem can then be (approximately) solved by first-order methods, typically *stochastic gradient descent/ascent*. Notice that **(1)** For classification, the function $g(\phi)$ returns the probabilities after the softmax function, not the categorical label; **(2)** An additional regularizer $\mathcal{R}(\phi)$ on the parameters will not cause difficulty in backpropagation, given $\partial\mathcal{R}(\phi)/\partial\phi$ is easily computed.

Challenge in sampling-free probabilistic backpropagation. Learning BNNs is not amenable to standard backpropagation because the ELBO objective function $\mathcal{L}(\phi)$ in Equation (5.4b) is not an *explicit* function of the predictive distribution $g(\phi)$ in Equation (5.4a) (i.e., $\mathcal{L}(\phi)$ can not be easily written as $\mathcal{L}(g(\phi))$).

$$g_n(\phi) = \mathbb{E}_Q\left[\mathbf{Pr}[y_n|x_n, \boldsymbol{\theta}]\right] = \int_\theta \mathbf{Pr}[y_n|x_n, \theta] Q(\theta; \phi) d\theta, \tag{5.4a}$$

$$\mathcal{L}_n(\phi) = \mathbb{E}_Q\left[\log(\mathbf{Pr}[y_n|x_n, \boldsymbol{\theta}])\right] = \int_\theta \log\left(\mathbf{Pr}[y_n|x_n, \theta]\right) Q(\theta; \phi) d\theta. \tag{5.4b}$$

Although $\mathcal{L}_n(\phi)$ is a function of ϕ, it is not an explicit function of $g_n(\phi)$. Consequently, the chain rule in Equation (5.3) on which backpropagation is based is not directly applicable.

5.2.3 Learning in BNNs: An Alternative ELBO

Alternative evidence lower bound. We make learning in BNNs amenable to backpropagation by developing a lower bound $\overline{\mathcal{L}}_n(\phi) \leq \mathcal{L}_n(\phi)$ such that $\partial\overline{\mathcal{L}}_n(\phi)/\partial\phi$ can be obtained by chain rule (i.e., $\overline{\mathcal{L}}_n(\phi)$ is an explicit function of the results from the forward pass.) With $\overline{\mathcal{L}}_n(\phi)$ in hand, we can (approximately) find $\phi^\star$ by maximizing the alternative objective via gradient-based method:

$$\phi^\star = \arg\max_\phi \overline{\mathcal{L}}(\phi) = \arg\max_\phi \left(\mathcal{R}(\phi) + \sum_{n=1}^{N} \overline{\mathcal{L}}_n(\phi)\right). \tag{5.5}$$

Theorem 5.1 provides one such feasible $\overline{\mathcal{L}}_n(\phi)$ which only depends on second last output $\boldsymbol{h}^{(L-1)}$.

Theorem 5.1 (Alternative evidence lower bound). *Define each term $\overline{\mathcal{L}}_n(\phi)$ of $\overline{\mathcal{L}}(\phi)$ in Equation (5.5) as*

$$\overline{\mathcal{L}}_n(\phi) := \mathbb{E}_{\boldsymbol{h}^{(L-1)}\sim P;\ \boldsymbol{\theta}^{(L-1)}\sim Q}\left[\log\left(\mathbf{Pr}[y_n|\boldsymbol{h}^{(L-1)}, \boldsymbol{\theta}^{(L-1)}]\right)\right], \tag{5.6}$$

then $\overline{\mathcal{L}}_n(\phi)$ is a lower bound of $\mathcal{L}_n(\phi)$, i.e., $\overline{\mathcal{L}}_n(\phi) \leq \mathcal{L}_n(\phi)$. The equality $\overline{\mathcal{L}}_n(\phi) = \mathcal{L}_n(\phi)$ holds if $\boldsymbol{h}^{(L-1)}$ is deterministic given input x and all parameters before the last layer $\theta^{(:L-2)}$.

Analytic form of $\overline{\mathcal{L}}_n(\phi)$. While the lower bound in Theorem 5.1 applies to BNNs with *arbitrary* distributions P on hidden variables $\boldsymbol{h}$, Q on model parameters $\boldsymbol{\theta}$, and *any* problem setting (e.g., classification or regression), *in practice* sampling-free probabilistic backpropagation requires that $\overline{\mathcal{L}}_n(\phi)$ can be analytically evaluated (or further lower bounded) in terms of $\phi^{(L-1)}$ and $\theta^{(L-1)}$. This task is nontrivial since it requires redesign of the output layer, i.e., the function of $\mathbf{Pr}[y|\boldsymbol{h}^{(L-1)}, \boldsymbol{\theta}^{(L-1)}]$. For ease of exposition, we only present the classification case in this section — the regression case can be found in [120]. Since $\overline{\mathcal{L}}_n(\phi)$ involves the last layer only, we omit the superscripts/subscripts of $\boldsymbol{h}^{(L-1)}$, $\psi^{(L-1)}$, $\phi^{(L-1)}$, x_n, y_n, and denote them as $\boldsymbol{h}$, ψ, ϕ, x, y for simplicity.

Theorem 5.2 (Analytic form of $\overline{\mathcal{L}}_n(\phi)$ for classification). *Let $\boldsymbol{h} \in \mathbb{R}^K$ (with K the number of classes) be the pre-activations of a softmax layer (a.k.a. logits), and $\phi = s \in \mathbb{R}^+$*

be a scaling factor that adjusts its scale such that

$$\mathbf{Pr}[\boldsymbol{y} = c | \boldsymbol{h}, s] = \exp(\boldsymbol{h}_c / s) / \sum_{k=1}^{K} \exp(\boldsymbol{h}_k / s) \tag{5.7}$$

Suppose the logits $\{\boldsymbol{h}_k\}_{k=1}^{K}$ *are pairwise independent (which holds under mean-field approximation) and* $\boldsymbol{h}_k$ *follows a Gaussian distribution* $\boldsymbol{h}_k \sim \mathcal{N}(\mu_k, \nu_k)$ *(therefore* $\psi = \{\mu_k, \nu_k\}_{k=1}^{K}$*) and* s *is a deterministic parameter. Then* $\overline{\mathcal{L}}_n(\phi)$ *is further lower bounded as*

$$\overline{\mathcal{L}}_n(\phi) \geq \frac{\mu_c}{s} - \log \left(\sum_{k=1}^{K} \exp\left(\frac{\mu_k}{s} + \frac{\nu_k}{2s^2} \right) \right). \tag{5.8}$$

The proof of Theorem 5.2 and its regression counterpart can be found in [120].

5.2.4 Prediction in BNNs: A Tensor Approach

While Section 5.2.3 provides a general solution to learning in BNNs, the solution relies on the ability to perform probabilistic propagation efficiently. To address this, we introduce Bayesian quantized networks (BQNs) — BNNs where both hidden units $\boldsymbol{h}^{(l)}$'s and model parameters $\boldsymbol{\theta}^{(l)}$'s take discrete values — along with a set of novel algorithms for efficient sampling-free probabilistic propagation.

Probabilistic propagation as tensor contractions. For simplicity, we assume activations and model parameters take values from the same set $\mathbb{Q}$, and denote the *degree of quantization* as $D = |\mathbb{Q}|$, (e.g. $\mathbb{Q} = \{-1, 1\}$, $D = 2$).

Lemma 5.3 (Probabilistic propagation in BQNs). *After quantization, each step of* *probabilistic propagation in Equation (5.1) becomes a finite sum instead of an integral:*

$$P(\boldsymbol{h}^{(l+1)}; \psi^{(l+1)}) = \sum\nolimits_{h^{(l)}, \theta^{(l)}} \mathbf{Pr}[\boldsymbol{h}^{(l+1)} | h^{(l)}, \theta^{(l)}] \; Q(\theta^{(l)}; \phi^{(l)}) \; P(h^{(l)}; \psi^{(l)}), \tag{5.9}$$

and a categorically distributed $\boldsymbol{h}^{(l)}$ results in $\boldsymbol{h}^{(l+1)}$ being categorical as well. The equation holds without any assumption on the operation $\mathbf{Pr}[\boldsymbol{h}^{(l+1)} | \boldsymbol{h}^{(l)}, \theta^{(l)}]$ in the neural network.

Note that all distributions in Equation (5.9) are represented in high-order tensors: Suppose there are I input units, J output units, and K model parameters at the l-th layer, then $\boldsymbol{h}^{(l)} \in \mathbb{Q}^I$, $\boldsymbol{\theta}^{(l)} \in \mathbb{Q}^K$, and $\boldsymbol{h}^{(l+1)} \in \mathbb{Q}^J$, and their distributions are characterized by $P(\boldsymbol{h}^{(l)}; \psi^{(l)}) \in \mathbb{R}^{D^I}$, $Q(\boldsymbol{\theta}^{(l)}; \phi^{(l)}) \in \mathbb{R}^{D^K}$, $P(\boldsymbol{h}^{(l+1)}; \psi^{(l+1)}) \in \mathbb{R}^{D^J}$, and $\mathbf{Pr}[\boldsymbol{h}^{(l+1)} | \boldsymbol{h}^{(l)}, \boldsymbol{\theta}^{(l)}]$ $\in \mathbb{R}^{D^J \times D^I \times D^K}$ respectively. Therefore, each step in probabilistic propagation is a *tensor contraction* of three tensors, which establishes the duality between BQNs and hierarchical tensor networks [121].

Since tensor contractions are differentiable w.r.t. all inputs, BQNs thus circumvent the difficulties in training QNNs [122, 123], whose outputs are not differentiable w.r.t. the discrete parameters. This result is not surprising: if we consider learning in QNNs as an *integer programming* (IP) problem, solving its Bayesian counterpart is equivalent to relaxing the problem into a *continuous optimization* problem [124].

Complexity of exact propagation. The computational complexity to evaluate Equation (5.9) exactly is exponential in the number of random variables $O(D^{IJK})$, which is intractable for quantized neural network of any reasonable size. We thus seek approximations to Equation (5.9).

Approximate propagation via tensor decomposition. We propose a principled approximation to reduce the computational complexity in probabilistic propagation in BQNs using tensor *CP decomposition*, which factors an intractable high-order probability tensor into tractable lower-order factors [24]. In this chapter, we consider the simplest *rank*-1 *tensor CP decomposition*, where the joint distributions of P and Q are fully factorized into products of their marginal distributions, thus equivalent to the *mean-field approximation* [125]. With rank-1 CP decomposition on $P(h^{(l)}; \psi^{(l)}), \forall l \in [L]$, the tensor contraction in (5.9) reduces to a standard *Tucker contraction* [33]:

$$P(\boldsymbol{h}_j^{(l+1)}; \psi_j^{(l+1)}) \approx \sum_{h^{(l)}, \theta^{(l)}} \mathbf{Pr}[\boldsymbol{h}_j^{(l+1)} | \theta^{(l)}, h^{(l)}] \prod_k Q(\theta_k^{(l)}; \phi_k^{(l)}) \prod_i P(h_i^{(l)}; \psi_i^{(l)}), \tag{5.10}$$

where each $\psi_i^{(l)}$ or $\phi_k^{(l)}$ parameterizes a single categorical variable. In our implementation, we store the parameters in their log-space, i.e., $Q(\theta_k^{(l)} = \mathbb{Q}(d)) = \exp(\psi_k^{(l)}(d)) / \sum_{q=1}^{D} \exp(\phi_k^{(l)}(q))$.

Fan-in number and the complexity of approximate propagation. In a practical model, for the l-th layer, an output unit $\boldsymbol{h}_j^{(l+1)}$ only (conditionally) depends on a subset of all input units $\{\boldsymbol{h}_i^{(l)}\}$ and model parameters $\{\boldsymbol{\theta}_k^{(h)}\}$ according to the connectivity pattern in the layer. We denote the set of dependent input units and parameters for $\boldsymbol{h}_j^{(l+1)}$ as $\mathcal{I}_j^{(l+1)}$ and $\mathcal{M}_j^{(l+1)}$, and define the *fan-in number* E for the layer as $\max_j \left|\mathcal{I}_j^{(l+1)}\right| + \left|\mathcal{M}_j^{(l+1)}\right|$. The approximate propagation reduces the computational complexity from $O(D^{IJK})$ to $O(JD^E)$, which is linear in the number of output units J if we assume the fan-in number E to be a constant (i.e. E is not proportional to I).

Fast algorithms for approximate propagation. Different types of network layers have different fan-in numbers E, and for those layers with E greater than a small constant, Equation (5.10) is inefficient since the complexity grows exponential in E. Therefore, we devise fast(er) algorithms to further lower the complexity.

- **Small fan-in layers: direct tensor contraction.** If E is small, we implement the approximate propagation through *tensor contraction* in Equation (5.10). The computational complexity is $O(JD^E)$ as discussed previously.

- **Medium fan-in layers: discrete Fourier transform.** If E is medium, we implement approximate propagation through *fast Fourier transform* since summation of discrete random variables is equivalent to convolution between their probability mass function. With the *fast Fourier transform*, the computational complexity is reduced to $O(JE^2D\log(ED))$.

- **Large fan-in layers: Lyapunov central limit theorem.** In a typical linear layer, the fan-in E is large, and a super-quadratic algorithm using fast Fourier transform is still computational expensive. Therefore, we adopt a faster algorithm based on the *Lyapunov central limit theorem.* With CLT, the computational complexity is further reduced to $O(JED)$.

Remarks: Depending on the fan-in numbers E, we adopt CLT for linear layers with sufficiently large E such as *fully connected layers* and *convolutional layers*; DFT for those with medium E such as *average pooling layers* and *depth-wise layers*; and direct tensor contraction for those with small E such as *shortcut layers* and *nonlinear layers.*

5.3 Related Works and Discussions

Probabilistic neural networks and Bayesian neural networks These models consider weights to be random variables and aim to learn their distributions. To further distinguish two families of such models, we call a model *Bayesian neural network* if the distributions are learned using a prior-posterior framework (i.e., via Bayesian inference) [116, 117, 118, 119, 126, 127], and otherwise *probabilistic neural network* [128, 129, 130]. In particular, our work is closely related to *natural-parameters networks (NPN)* [128], which consider both weights and activations to be random variables from exponential family. Since categorical distribution (over quantized values) belongs to the exponential family, our BQN can be interpreted as categorical NPN, but we learn the distributions via Bayesian inference.

For Bayesian neural networks, various types of approaches have been proposed to learn the posterior distribution over model parameters:

(1) Sampling-free assumed density filtering (ADF), including EBP [117] and PBP [118], is an online algorithm that (approximately) updates the posterior distribution by Bayes' rule for each observation. If the model parameters $\boldsymbol{\theta}$ are Gaussian distributed, [131] shows that the Bayes' rule can be computed in analytic form based on $\partial \log(g_n(\phi))/\partial\phi$, and EBP [117] derives a similar rule for Bernoulli parameters in binary classification. Notice that ADF is compatible to backpropagation:

$$\frac{\partial \log(g_n(\phi))}{\partial \phi} = \frac{1}{g_n(\phi)} \cdot \frac{\partial g_n(\phi)}{\partial \phi}, \tag{5.11}$$

assuming $g_n(\phi)$ can be (approximately) computed by sampling-free probabilistic propagation as in Section 5.2.1. However, this approach has two significant limitations: (a) the Bayes' rule needed to be derived case by case, and analytic rules for most common scenarios are not known yet. (b) it is not compatible with modern optimization methods (such as SGD or Adam) as the optimization is solved analytically for each data point, therefore challenging to cope with large-scale models.

(2) Sampling-based variational inference (SVI), formulates an optimization problem and solves it approximately via *stochastic gradient descent (SGD)*. The most popular method among all is, *Stochastic Gradient Variational Bayes (SGVB)*, which approximates $\mathcal{L}_n(\phi)$ by the average of multiple samples [116, 126, 127]. Before each step of learning or prediction, a number of independent samples of the model parameters $\{\theta_s\}_{s=1}^{S}$ are drawn according to the current estimate of Q, i.e. $\theta_s \sim Q$, by which the predictive function $g_n(\phi)$ and the loss $\mathcal{L}_n(\phi)$ can be approximated by

$$g_n(\phi) \approx \frac{1}{S}\sum_{s=1}^{S} \mathbf{Pr}[y_n|x_n, \theta_s] = \frac{1}{S}\sum_{s=1}^{S} f_n(\theta_s), \tag{5.12a}$$

$$\mathcal{L}_n(\phi) \approx \frac{1}{S}\sum_{s=1}^{S} \log\left(\mathbf{Pr}[y_n|x_n, \theta_s]\right) = \frac{1}{S}\sum_{s=1}^{S} \log\left(f_n(\theta_s)\right), \tag{5.12b}$$

where $f_n(\theta) = \mathbf{Pr}[y_n|x_n, \theta]$ denotes the predictive function given a specific realization θ of the model parameters. The gradients of $\mathcal{L}_n(\phi)$ can now be approximated as

$$\frac{\partial \mathcal{L}_n(\phi)}{\partial \phi} \approx \frac{1}{S}\sum_{s=1}^{S} \frac{\partial \mathcal{L}_n(\phi)}{\partial f_n(\theta_s)} \cdot \frac{\partial f_n(\theta_s)}{\partial \theta_s} \cdot \frac{\partial \theta_s}{\partial \phi}. \tag{5.13}$$

This approach has multiple drawbacks: (a) Repeated sampling suffers from high variance, besides being computationally expensive in both learning and prediction phases; (b) While

$g_n(\phi)$ is differentiable w.r.t. ϕ, $f_n(\theta)$ may not be differentiable w.r.t. θ. One such example is quantized neural networks, whose backpropagation is approximated by *straight through estimator* [132]. (3) The partial derivatives $\partial\theta_s/\partial\phi$ are difficult to compute with complicated *reparameterization tricks* [133, 134].

(3) Deterministic variational inference (DVI) Our approach is most similar to [135], which observes that if $\mathbf{Pr}[\boldsymbol{h}^{(l+1)}|\boldsymbol{h}^{(l)}, \boldsymbol{\theta}^{(l)}]$ is a Dirac function, i.e., the underlying model is deterministic,

$$\mathcal{L}_n(\phi) := \mathbb{E}_{\boldsymbol{h}^{(L-1)}\sim P;\ \boldsymbol{\theta}^{(L-1)}\sim Q}\left[\log\left(\mathbf{Pr}[y_n|\boldsymbol{h}^{(L-1)}, \boldsymbol{\theta}^{(L-1)}]\right)\right]. \tag{5.14}$$

Our approach considers a wider scope of problem settings, where the model could be stochastic, i.e. $\mathbf{Pr}[\boldsymbol{h}^{(l+1)}|\boldsymbol{h}^{(l)}, \boldsymbol{\theta}^{(l)}]$ is an arbitrary function. Furthermore, [135] considers the case that all parameters $\boldsymbol{\theta}$ are Gaussian distributed, whose sampling-free probabilistic propagation requires complicated approximation [129].

Quantized neural networks. These models can be categorized into two classes: (1) *Partially quantized networks*, where only weights are discretized [42, 136]; (2) *Fully quantized networks*, where both weights and hidden units are quantized [122, 123, 137, 138, 139]. While both classes provide compact size, low-precision neural network models, fully quantized networks further enjoy fast computation provided by specialized bit-wise operations. In general, quantized neural networks are difficult to train due to their non-differentiability. Gradient descent by backpropagation is approximated by either *straight-through estimators* [132] or *probabilistic methods* [140, 141, 142]. Unlike these papers, we focus on Bayesian learning of fully quantized networks in this chapter. Optimization of quantized neural networks typically requires a dedicated loss function, learning scheduling and initialization. For example, [142] considers pre-training of a continuous-valued neural network as the initialization. Since our approach considers learning from scratch (with a uniform initialization), the performance could be inferior to prior works in terms of absolute accuracy.

Tensor networks and tensorial neural networks *Tensor networks* (TNs) are widely used in numerical analysis [24], quantum physiscs [23], and recently machine learning [25, 26] to model interactions among multi-dimensional random objects. Various tensorial neural networks (TNNs) [50, 143] have been proposed that reduce the size of neural networks by replacing the linear layers with TNs. Recently, [121] points out the duality between probabilistic graphical models (PGMs) and TNs, i.e., there exists a bijection between PGMs and TNs. We advance this line of thinking by connecting hierarchical Bayesian models (e.g. Bayesian neural networks) and hierarchical TNs.

5.4 Experimental Results

5.4.1 Experimental Setup

In this section, we demonstrate the effectiveness of BQNs on the MNIST, Fashion-MNIST, KMNIST and CIFAR10 classification datasets. We evaluate our BQNs with both *multi-layer perceptron* (MLP) and *convolutional neural network* (CNN) models. In training, each image is augmented by a random shift within 2 pixels (with an additional random flipping for CIFAR10), and no augmentation is used in test. In the experiments, we consider a class of quantized neural networks, with both *binary* weights and activations (i.e., $\mathbb{Q} = \{-1, 1\}$) with sign activations $\sigma(\cdot) = \text{sign}(\cdot)$. For BQNs, the distribution parameters ϕ are initialized by Xavier's uniform initializer, and all models are trained by Adam optimizer [8] for 100 epochs (and 300 epochs for CIFAR10) with batch size 100 and initial learning rate 10^{-2}, which decays by 0.98 per epoch.

Training objective of BQNs. To allow for customized level of uncertainty in the learned Bayesian models, we introduce a regularization coefficient λ in the alternative ELBO proposed in Equation (5.5) (i.e., a lower bound of the likelihood), and train the BQNs by maximizing the following objective:

$$\overline{\mathcal{L}}(\phi) = \sum_{n=1}^{N} \overline{\mathcal{L}}_n(\phi) + \lambda \mathcal{R}(\phi) = \lambda \left(1/\lambda \sum_{n=1}^{N} \overline{\mathcal{L}}_n(\phi) + \mathcal{R}(\phi) \right) \tag{5.15}$$

where λ controls the uncertainty level — the importance of the prior over the training set.

Baselines. **(1)** We compare our BQN against the baseline – *Bootstrap ensemble of quantized neural networks* (E-QNN). Each member in the ensemble is trained in a non-Bayesian way [115], and jointly make the prediction by averaging over the logits from all members. Note that [115] is chosen over other QNN training methods as the baseline since it trains QNN from random initialization, thus a fair comparison to our approach. **(2)** To exhibit the effectiveness of our BQN, we further compare against *continuous-valued Bayesian neural network* (abbreviated as BNN) with Gaussian parameters. The model is trained with *stochastic gradient variational Bayes* (SGVB) augmented by *local re-parameterization trick* [116]. Since the BNN allows for continuous parameters (different from BQN with quantized parameters), the predictive error is expected to be lower than BQN.

Evaluation of BQNs. While *0-1 test error* is a popular metric to measure the predictive performance, it is too coarse a metric to assess the uncertainty in decision making (for example, it does not account for how badly the wrong predictions are). Therefore, we mainly use the *negative log-likelihood* (NLL) to measure the predictive performance in the experiments. Once a BQN is trained (i.e., an approximate posterior $Q(\boldsymbol{\theta})$ is learned), we consider three modes to evaluate the behavior of the model: **(1)** *analytic inference (AI)*, **(2)** *Monte Carlo (MC) sampling* and **(3)** *Maximum a Posterior (MAP) estimation*:

1. In analytic inference (AI, i.e., our proposed method), we analytically integrate over $Q(\boldsymbol{\theta})$ to obtain the predictive distribution as in the training phase. Notice that the exact NLL is not accessible with probabilistic propagation (which is why we propose an alternative ELBO in Equation (5.5)), we report *an upper bound* of the NLL.

2. In MC sampling, we draw S sets of model parameters independently from the posterior posterior $\theta_s \sim Q(\boldsymbol{\theta}), \forall s \in [S]$, and the forward propagation is performed as in (non-Bayesian) quantized neural network for each set θ_s, followed by an average over the model outputs. The difference between analytic inference and MC sampling will be used to evaluate **(a)** the effect of mean-field approximation and **(b)** the tightness of the our proposed alternative ELBO.

3. MAP estimation is similar to MC sampling, except that there is only one set of model parameters $\theta^{\star} = \arg\max_{\theta} Q(\theta)$. We exhibit our model's ability to compress a BNN by comparing MAP estimation of our BQN with non-Bayesian QNN.

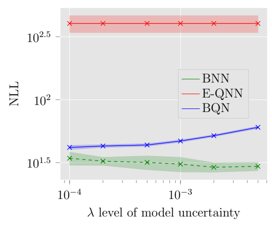
5.4.2 Analysis of Empirical Results

Methods	MNIST		KMNIST	
	NLL (10^{-3})	% Err.	NLL (10^{-3})	% Err.
E-QNN on MLP	546.6±157.9	3.30 ±0.65	2385.6±432.3	17.88±1.86
BQN on MLP	**130.0±3.5**	**2.49±0.08**	**457.7±13.8**	**13.41±0.12**
E-QNN on CNN	425.3±61.8	0.85±0.13	3755.7±465.1	11.49±1.16
BQN on CNN	**41.8±1.6**	**0.85±0.06**	**295.5±1.4**	**9.95±0.15**
Methods	Fashion-MNIST		CIFAR10	
	NLL (10^{-3})	% Err.	NLL (10^{-3})	% Err.
E-QNN on MLP	2529.4±276.7	13.02±0.81	N/A	N/A
BQN on MLP	**417.3±8.1**	**9.99±0.20**	N/A	N/A
E-QNN on CNN	1610.7±158.4	**3.02±0.37**	7989.7 ± 600.2	15.92 ± 0.72
BQN on CNN	**209.5±2.8**	4.65±0.15	**530.6 ± 23.0**	**13.74 ±0.47**

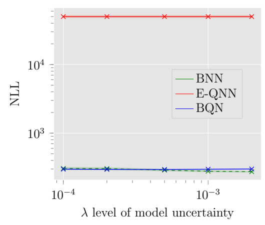
Table 5.1: **Comparison of performance of BQNs against the baseline E-QNN.** Each E-QNN is an ensemble of 10 networks trained individually and make predictions jointly. We report both NLL (which accounts for prediction uncertainty) and 0-1 test error (which doesn't account for prediction uncertainty). All the numbers average over 10 runs with different seeds; the standard deviation is exhibited following the ± sign.

Expressive power and uncertainty calibration in BQNs. We report the performance via all evaluations of our BQN models against the Ensemble-QNN in Table 5.1. **(1)** Compared to E-QNNs, our BQNs have significantly lower NLL and smaller predictive error (except for Fashion-MNIST with architecture CNN). **(2)** As we can observe in Figure 5.1, BQNs impressively achieve comparable NLL to continuous-valued BNN, with slightly higher test error. As our model parameters only take values $\{-1, 1\}$, small degradation in predictive accuracy is expected.

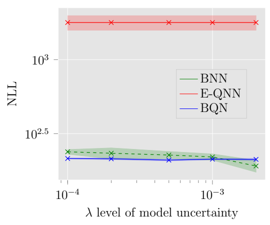
Evaluations of mean-field approximation and tightness of the alternative ELBO. If analytic inference (by probabilistic propagation) were computed exactly, the evaluation metrics would have been equal to the ones with MC sampling (with infinite samples). There-



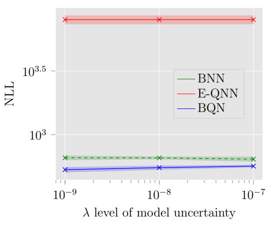
(a) NLL MNIST



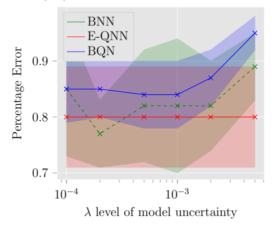
(b) NLL FMNIST



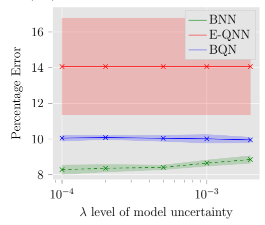
(c) NLL KMNIST



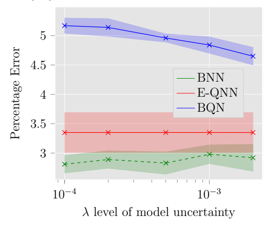
(d) NLL CIFAR10



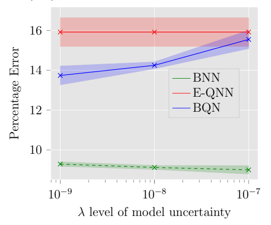
(e) Error MNIST



(f) Error FMNIST

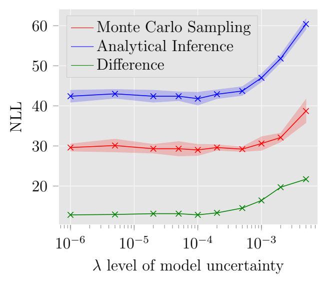


(g) Error KMNIST

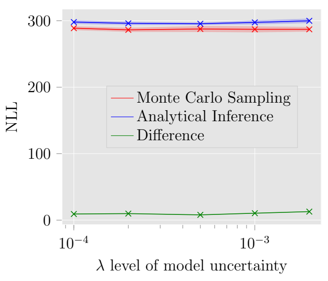


(h) Error CIFAR10

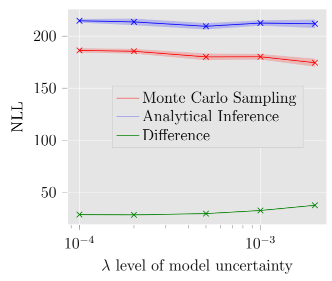
Figure 5.1: **Comparison of the predictive performance** of our BQNs against the E-QNN as well as the non-quantized BNN trained by SGVB on a CNN. Negative log-likelihood (NLL) which accounts for uncertainty and 0-1 test error which doesn't account for uncertainty are displayed.



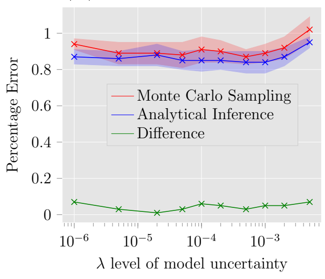
(a) NLL MNIST



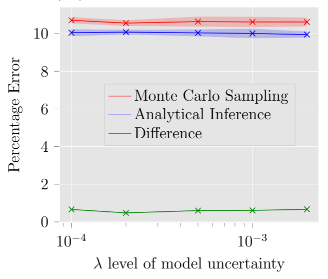
(b) NLL FMNIST



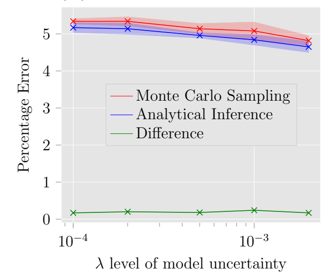
(c) NLL KMNIST



(d) Error MNIST



(e) Error FMNIST



(f) Error KMNIST

Figure 5.2: **Illustration of mean-field approximation and tightness of alternative ELBO on CNNs.** The performance gap between our analytical inference and the Monte Carlo Sampling is displayed.

fore we can evaluate the approximations in probabilistic propagation, namely *mean-field approximation* in Equation (5.10) and *relaxation of the original ELBO* in Equation (5.5), by measuring the gap between analytic inference and MC sampling. As shown in Figure 5.2, such gaps are small for all scenarios, which justifies the approximations we use in BQNs.

To further decouple these two factors of *mean-field approximation* and *relaxation of the original ELBO*, we vary the regularization coefficient λ in the learning objective. **(1)** For $\lambda = 0$ (where the prior term is removed), the models are forced to become deterministic during training. Since the deterministic models do not have mean-field approximation in the forward pass, the gap between analytic inference and MC-sampling reflects the tightness of our alternative ELBO. **(2)** As λ increases, the gaps increases slightly as well, which shows that the mean-field approximation becomes slightly less accurate with higher learned uncertainty in the model.

Methods	MNIST		KMNIST		Fashion-MNIST	
	NLL(10^{-3})	% Err.	NLL(10^{-3})	% Err.	NLL(10^{-3})	% Err.
QNN-MLP	522.4±42.2	4.14±0.25	2019.1±281.2	19.56±1.97	2427.1±193.5	15.67±1.19
BQN-MLP	**137.60±4.40**	**3.69±0.09**	**464.60±12.80**	**14.79±0.21**	**461.30±13.40**	**12.89±0.17**
QNN-CNN	497.4±139.5	1.08±0.2	4734.5±1697.2	14.2±2.29	1878.3±223.8	**3.88±0.33**
BQN-CNN	**30.3±1.6**	**0.92±0.07**	**293.6±4.4**	**10.82±0.37**	**179.1±4.4**	5.00±0.11

Table 5.2: **Deterministic model compression** through direct training of QNN [115] v.s. **MAP estimation** in our proposed BQN. All the numbers are averages over 10 runs with different seeds, the standard deviation is exhibited following the ± sign.

Compression of neural networks via BQNs. One advantage of BQNs over continuous-valued BNNs is that we can obtain deterministic QNNs for free, since a BQN can be interpreted as an ensemble of infinite QNNs (each is a realization of the posterior distribution). **(1)** One simple approach is to set the model parameters to their *MAP estimates*, which compresses a given BQN to 1/64 of its original size (and has the same number of bits as a

Methods	MNIST		KMNIST		Fashion-MNIST	
	NLL(10^{-3})	% Err.	NLL(10^{-3})	% Err.	NLL(10^{-3})	% Err.
E-QNN-MLP	546.60±157.90	3.30 ±0.65	2385.60±432.30	17.88±1.86	2529.40±276.70	13.02±0.81
BQN-MLP	**108.9±2.6**	**2.73±0.09**	**429.50±11.60**	**13.83±0.12**	**385.30±5.10**	**10.81±0.44**
E-QNN-CNN	425.3±61.80	**0.85±0.13**	3755.70±465.10	11.49±1.16	1610.70±158.40	**3.02±0.37**
BQN-CNN	**29.2±0.6**	0.87±0.04	**286.3±2.7**	**10.56±0.14**	**174.5±3.6**	4.82±0.13

Table 5.3: **Bayesian model compression** through direct training of E-QNN v.s. a **Monte-Carlo sampling** on our proposed BQN. Each ensemble consists of 5 quantized neural networks, and for fair comparison we use 5 samples for Monte-Carlo evaluation. All the numbers are averages over 10 runs with different seeds, the standard deviation are exhibited following the ± sign.

single QNN). **(2)** *MC sampling* can be interpreted as another approach to compress a BQN, which reduces the original size to its $S/64$ (as an ensemble of S QNNs). In Table 5.2 and Table 5.3, we compare the models by both approaches to their counterparts (a single QNN for MAP, and E-QNN for MC sampling) trained from scratch as in [115]. Our compressed models outperform their counterparts (in NLL) for both approaches. We attribute this to two factors: **(a)** QNNs are not trained in a Bayesian way; therefore the uncertainty is not well calibrated; and **(b)** Non-differentiable QNNs are unstable to train. Our compression approaches via BQNs simultaneously solve both problems.

5.5 Conclusion

We present a *sampling-free*, *backpropagation-compatible*, *variational* approach for learning Bayesian quantized neural networks (BQNs). We develop a suite of algorithms for efficient inference in BQNs such that our approach scales to large models. We evaluate our BQNs by Monte-Carlo sampling, which proves that our approach can learn a proper posterior distribution on QNNs. Furthermore, we show that our approach can learn (ensemble) QNNs by taking maximum a posterior (or sampling from) the posterior distribution.

Chapter 6: ARMA Nets: Economical Expansion of Receptive Fields

6.1 Overview

Convolutional layers in neural networks have many successful applications for machine learning tasks. Each output neuron encodes an input region of the network measured by the *effective receptive field* (ERF) [144]. A large ERF that allows for sufficient global information is needed to make accurate predictions; however, a simple stack of convolutional layers does not effectively expand ERF. Convolutional neural networks (CNNs) typically encode global information by adding downsampling (pooling) layers, which coarsely aggregate global information. Subsequently, a fully-connected classification layer reduces the entire feature map to an output label. Downsampling and fully-connected layers are suitable for image classification tasks where only a single prediction is needed. But they are less effective, due to potential loss of information, in dense prediction tasks such as semantic segmentation and video prediction, where each pixel requests a prediction. Therefore, it is crucial to introduce mechanisms that enlarge ERF without too much information loss.

Naive approaches to expanding ERF, such as deepening the network or enlarging the filter size, drastically increase the model complexity, which results in expensive computation, difficulty in optimization, and susceptibility to overfitting. Advanced architectures have been proposed to expand ERF, including encoder-decoder networks [145], dilated convolutional networks [146, 147], and non-local networks [148]. However, encoder-decoder networks could lose high-frequency information due to the downsampling layers. Dilated convolutional networks could suffer from the gridding effect while the ERF expansion is limited, and non-local networks are expensive in training and inference.

We introduce a novel *autoregressive-moving-average* (ARMA) layer that enables adaptive receptive field by explicit interconnections among its output neurons. Our ARMA layer realizes these interconnections via extra convolutions on output neurons, on top of the convolutions on input neurons as in a traditional convolutional layer. We provably show that an ARMA network can have arbitrarily large ERF, thus capturing global information, with minimal extra parameters at each layer. Consequently, an ARMA network can flexibly enlarge its ERF to leverage global knowledge without reducing spatial resolution. Moreover, the ARMA networks are independent of the architectures above, including encoder-decoder networks, dilated convolutional networks, and non-local networks.

A significant challenge in ARMA networks lies in the complex computations needed in both forward and backward propagations — simple convolution operations are not applicable since the output neurons are influenced by their neighbors and thus interrelated. Another challenge in ARMA networks is instability — the additional interconnections among the output neurons could recursively amplify the outputs and lead them to infinity. We address both challenges in this chapter.

Contributions. In summary, our contributions include:

1. We introduce a novel ARMA layer that is a plug-and-play module substituting convolution layers in neural networks to allow flexible tuning of their ERF, adapting to

the task requirements, and improving performance in dense prediction problems.

2. We recognize and address the problems of *computation* and *instability* in ARMA layers. **(1)** To reduce computational complexity, we develop FFT-based algorithms for both forward and backward passes; **(2)** To guarantee stability, we propose a *separable ARMA layer* and a re-parameterization mechanism that ensures that the layer operates in a stable region.

3. We successfully apply ARMA layers in ConvLSTM network [78] for pixel-level multi-frame video prediction and U-Net model [145] for medical image segmentation. ARMA networks substantially outperform the corresponding baselines on both tasks, suggesting that our proposed ARMA layer is a general and useful building block for dense prediction problems.

6.2 Related Works

Dilated convolution [149] enlarges the receptive field by upsampling the filter coefficients with zeros. Unlike encoder-decoder structure, dilated convolution preserves the spatial resolution and is thus widely used in dense prediction problems, including semantic segmentation [146, 150, 151], and objection detection [152, 153]. However, dilated convolution creates gridding artifacts if its input contains higher frequency than the upsampling rate [147], and the local information inconsistency hampers the performance of dilated convolutional networks [154]. Such artifacts can be alleviated by additional anti-aliasing layer [147], group interacting layer [154], or spatial pyramid pooling [155].

Deformable convolution [156, 157, 158] allows the filter shape (i.e., locations of the incoming pixels) to be learnable. While the deformable convolution adjusts the filter *shape*, our ARMA layer expands the effective filter *size* adaptively.

Non-local attention network [148] inserts non-local attention blocks between the convolutional layers. A non-local attention block computes a weighted sum of all input neurons for each output neuron, similar to attention mechanism [4]. In practice, non-local attention blocks are computationally expensive, thus they are typically inserted between the layers with low-resolution features. In contrast, our ARMA layers are economical and can be used throughout the network.

Encoder-decoder structured network [145, 150] pairs upsampling and downsampling layers to maintain the resolution, and introduces skip-connection between each pair to preserve the high-frequency information. Since the shortcut bypasses the downsampling/upsampling layers, the network has a small receptive field for high-frequency components. A potential solution is to augment upsampling with non-local attention block [159] or ARMA layer.

Spatial recurrent neural networks [97, 160, 161, 162, 163, 164] are used to expand the receptive field or learn the affinity between neighboring pixels. However, almost all prior works consider nonlinear recurrent neural networks, where the gating mechanism prohibits an efficient parallel algorithm. Quasi-recurrent neural networks [165] partially address the problem by decoupling the linear operations and parallelizing them using convolutions. In contrast, our proposed ARMA layer is a linear recurrent neural network, allowing for efficient evaluation using FFT.

6.3 ARMA Neural Networks

In this section, we introduce a novel *autoregressive-moving-average* (ARMA) layer and analyze its ability to expand *Effective Receptive Field* (ERF) in neural networks. The analysis is further verified by visualizing the ERF with varying network depth and strength of autoregressive coefficients. We then address the problems of computation and stability in the ARMA layer.

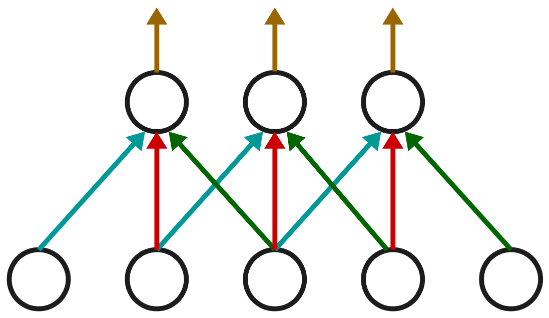
6.3.1 ARMA Layer

A traditional convolutional layer is essentially a *moving-average* model [166]:

$$\mathcal{Y}_{:,:,t} = \sum_{s=1}^{S} \mathcal{W}_{:,:,t,s} * \mathcal{X}_{:,:,s}, \tag{6.1}$$

where the *moving-average coefficients* $\mathcal{W} \in \mathbb{R}^{K_m \times K_m \times T \times S}$, are parameterized by a 4th-order kernel (K_m is the filter size, and S, T are input/output channels), $\mathbf{:}$ denotes all elements from the specified coordinate and $*$ denotes convolution between a channel and a filter.

As motivated in the introduction, we introduce a novel ARMA layer that enables an adaptive receptive field by introducing explicit interconnections among its output neurons, as illustrated in Figure 6.1. Our ARMA layer realizes these interconnections by introducing extra convolutions on the outputs, upon the convolutions on the inputs as in a traditional convolutional layer. As a result, in an ARMA layer, each output neuron can be affected by an input pixel faraway through interconnections among the output neurons, thus receives


(a) Traditional Convolutional Layer.

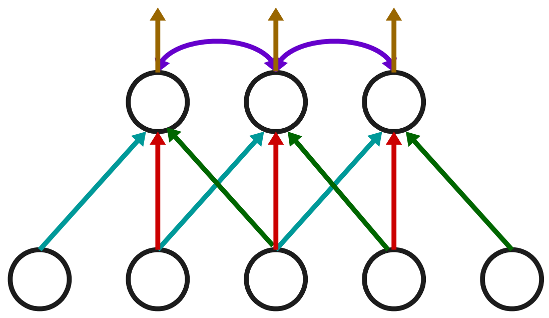

(b) ARMA Layer

Figure 6.1: **Diagrams of 1D convolutional layers.** In **(b)**, the ARMA layer introduces interconnections among output neurons explicitly. Thus, each output neuron can receive information from all input neurons.

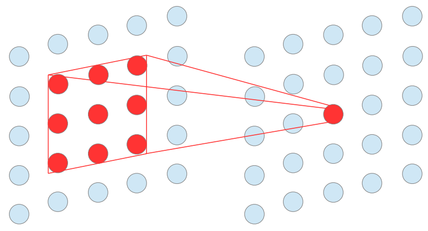
global information. Formally, we define an ARMA layer in Definition 6.1.

Definition 6.1 (**ARMA layer**)**.** *An ARMA layer is parameterized by a moving-average kernel (coefficients)* $\mathcal{W} \in \mathbb{R}^{K_m \times K_m \times S \times T}$ *and an autoregressive kernel (coefficients)* $\mathcal{A} \in \mathbb{R}^{K_a \times K_a \times T}$*. It receives an input* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times S}$ *and returns an output* $\mathcal{Y} \in \mathbb{R}^{I'_1 \times I'_2 \times T}$ *with an ARMA model:*

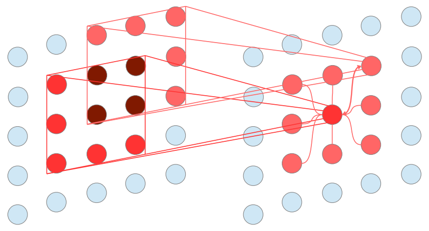
$$\mathcal{A}_{:,:,t} * \mathcal{Y}_{:,:,t} = \sum_{s=1}^{S} \mathcal{W}_{:,:,t,s} * \mathcal{X}_{:,:,s}. \tag{6.2}$$

Remarks: **(1)** The ARMA layer maintains the *shift-invariant* property, since the output interconnections are realized by convolutions. **(2)** The ARMA layer *reduces* to a traditional layer if the autoregressive kernel $\mathcal{A}$ represents an identical mapping. **(3)** The ARMA layer is a *plug-and-play* module that can replace *any* convolutional layer, adding $K_a^2 T$ extra parameters negligible compared to $K_w^2 ST$ parameters in a traditional convolution layer. **(4)** Unlike a traditional layer, computing Equation (6.2) and its backpropagation are nontrivial, which are studied in Section 6.3.3.

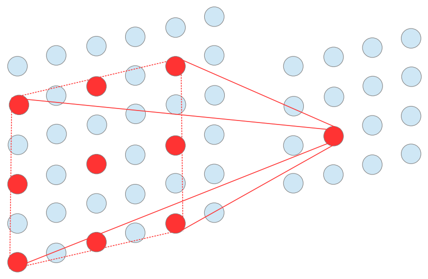
Our ARMA layer is complementary to the methods of dilated convolutional layer,



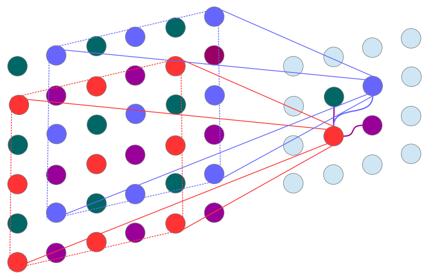
(a) Traditional Convolution Layer.



(b) ARMA Layer.



(c) Dilated Convolutional Layer.



(d) Dilated ARMA Layer.

Figure 6.2: **Diagrams of 2D convolutional layers.** In **(b)**, each output neuron receives its neighbors' receptive field. In **(d)**, ARMA's autoregression fills the gaps created by dilated convolution.

deformable convolutional layer, non-local attention block, as well as encoder-decoder architecture. For instance, a *dilated ARMA layer*, illustrated in Figure 6.2d, removes the gridding effect caused by dilated convolution — the autoregressive kernel plays a role as an anti-aliasing filter.

The motivation of introducing the ARMA layer is to enlarge the effective input region for each network output without increasing the filter size or network depth, thus avoiding the difficulties in training wider or deeper models. As illustrated in Figure 6.2, each output neuron in a traditional convolutional layer (Figure 6.2a) only receives information from a small input region (the filter size). However, an ARMA layer enlarges the small local region to a larger one (Figure 6.2b). It enables an output neuron to receive information from a faraway input neuron through the connections to its neighbors. In Section 6.3.2, we formally introduce the concept of *effective receptive field* (ERF) to characterize the input region size. Moreover, we will show that an ARMA network can have arbitrarily large ERF with a single extra parameter at each layer in Theorem 6.3.

6.3.2 Effective Receptive Field

Effective receptive field (ERF) measures the contribution of each input pixel to an output neuron [144]. In this subsection, we analyze the ERF of an L-layers network with ARMA layers v.s. traditional convolutional layers. Formally, consider an output at (i_1, i_2), the impact from an input pixel at $(i_1 - p_1, i_2 - p_2)$ (i.e L layers and (p_1, p_2) pixels away) is measured by the gradient amplitude $g(i_1, i_2, p_1, p_2) = \left|\partial \mathcal{Y}^{(L)}_{i_1,i_2,t} / \partial \mathcal{X}^{(1)}_{i_1-p_1,i_2-p_2,s}\right|$ (where superscripts index the layers), i.e., how an output neuron changes as an input pixel perturbs.

Definition 6.2 (Effective receptive field, ERF). *Consider an L-layers network with an S-channels input $\mathcal{X}^{(1)} \in \mathbb{R}^{I_1 \times I_2 \times S}$ and a T-channels output $\mathcal{Y}^{(L)} \in \mathbb{R}^{I_1 \times I_2 \times T}$, its effective receptive field is defined as the empirical distribution of the gradient maps: $ERF(p_1, p_2) = 1/(I_1 I_2 S T) \cdot \sum_{s,t,i_1,i_2} [g(i_1, i_2, p_1, p_2) / \sum_{j_1,j_2} g(j_1, j_2, p_1, p_2)]$, To measure the size of the ERF, we define its radius $r(ERF)$ as the standard deviation of the empirical distribution:*

$$r\left(ERF\right)^2 = \sum_{p_1,p_2} \left(p_1^2 + p_2^2\right) ERF\left(p_1, p_2\right) - \left[\sum_{p_1,p_2} \sqrt{p_1^2 + p_2^2}\; ERF(p_1, p_2)\right]^2. \tag{6.3}$$

Notice that the ERF simultaneously depends on the model parameters and a specified input to the network, i.e., it is both *model-dependent* and *data-dependent*. Therefore, it is generally intractable to compute the ERF analytically for any practical neural network. We follow [144] to estimate the radius with a simplified linear network. The paper empirically verifies that such an estimation is accurate and can be used to guide filter designs.

Theorem 6.3 (ERF radius of a linear ARMA network). *Consider an L-layers linear network, where the ℓ^{th} layer computes $y_i^{(\ell)} - a^{(\ell)} y_{i-1}^{(\ell)} = \sum_{p=0}^{K^{(\ell)}-1} [(1 - a^{(\ell)})/K^{(\ell)}] \cdot y_{i-d^{(\ell)}p}^{(\ell-1)}$ (i.e., the moving-average coefficients are uniform with length $K^{(\ell)}$ and dilation $d^{(\ell)}$, and the autoregressive coefficients $\boldsymbol{a}^{(\ell)} = \{1, -a^{(\ell)}\}$ has length 2). Suppose $0 \leq a^{(\ell)} < 1, \forall \ell \in [L]$, the ERF radius of the network is*

$$r(ERF)_{ARMA}^2 = \sum_{\ell=1}^{L} \left[\frac{d^{(\ell)^2}\left(K^{(\ell)^2} - 1\right)}{12} + \frac{a^{(\ell)}}{\left(1 - a^{(\ell)}\right)^2} \right]. \tag{6.4}$$

We prove Theorem 6.3 in [167]. If the coefficients for all layers are identical, e.g., $K^{(\ell)} = K, d^{(\ell)} = d, a^{(\ell)} = a$, the radius reduces to $r(\mathrm{ERF})_{\mathrm{ARMA}} = \sqrt{L} \cdot \sqrt{d^2(K^2-1)/12 + a/(1-a)^2}$. Moreover, if $a = 0, d = 1$, the ARMA layers reduce to traditional convolutional layers, and the ERF of the linear CNN has radius $r(\mathrm{ERF})_{\mathrm{CNN}} = \sqrt{L} \cdot \sqrt{(K^2-1)/12}$ as shown in [144].

Remarks: **(1) Compared with a (dilated) CNN, an ARMA network can have arbitrarily large ERF with an extra parameter a at each layer.** When the autoregressive coefficient a is large (e.g., $a > 1 - 1/(dK)$), the second term $a/(1-a)^2$ dominates the radius, and the ERF is substantially larger than that of a CNN. In particular, the radius tends to infinity as a approaches 1. **(2) An ARMA network can adaptively adjust its ERF through learnable parameter a.** As a gets smaller (e.g., $a < 1 - 1/(dK)$), the second term is comparable to or smaller than the first term, and the effect of expanded ERF diminishes. In particular, if $a = 0$, an ARMA network reduces to a CNN.

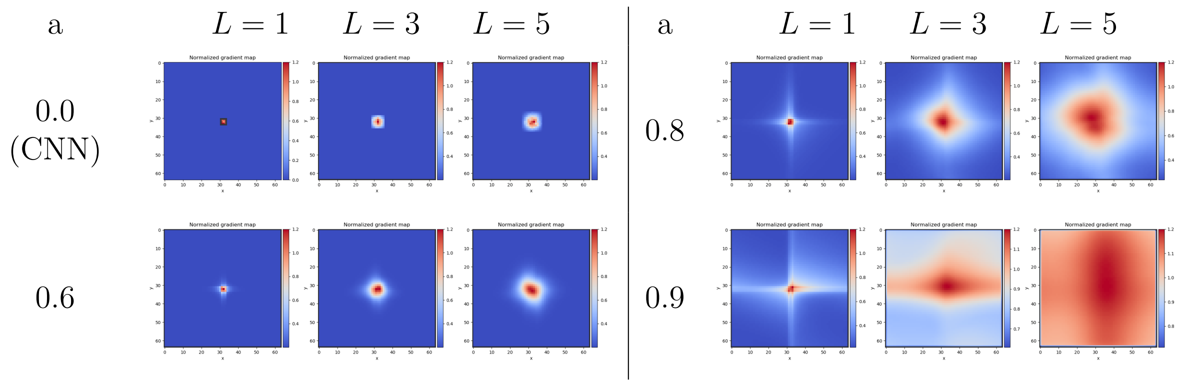


Figure 6.3: **Visualization of the effective receptive field of linear ARMA networks** under different network depth $L = 1, 3, 5$ and different magnitude of the autoregressive coefficients $a = 0.0, 0.6, 0.8, 0.9$.

Visualization of the ERF. We analytically show in Theorem 6.3 that the ERF radius increases with the network depth and autoregressive coefficients' magnitude. We now verify our analysis by simulating linear ARMA networks with varying depths and autoregressive coefficients' magnitude. As shown in Figure 6.3, the ERF radius increases as the autoregressive coefficients get larger. When the autoregressive coefficients are zeros, an ARMA network reduces to a traditional convolutional network. The simulation results also indicate that ARMA's ability to expand the ERF increases with the network depth. In conclusion, an ARMA network can have a large ERF even when the network is shallow, and its ability to expand the ERF increases when the network gets deeper.

6.3.3 Prediction and Learning of ARMA Layers

In an ARMA layer, each neuron is influenced by its neighbors from all directions (see Figure 6.2b). As a result, no neuron could be evaluated alone before evaluating any other neighboring neurons. To compute Equation (6.2), we need to solve a set of linear equations to obtain all values simultaneously. **(1)** However, the standard solver using

Gaussian elimination is too expensive to be practical, and therefore we need to seek a more efficient solution. **(2)** Furthermore, the solver for linear equations typically does not support automatic differetiation, and we have to derive the backward equations analytically. **(3)** Finally, we also need to devise an efficient algorithm to compute the backpropagation equations efficiently. In the section, we address these aforementioned problems.

Decomposition of an ARMA Layer. We decompose the ARMA layer in Equation (6.2) into a moving-average (MA) layer and an autoregressive (AR) layer:

$$\text{MA Layer:} \quad \mathcal{T}_{:,:,t} = \sum_{s=1}^{S} \mathcal{W}_{:,:,t,s} * \mathcal{X}_{:,:,s}, \tag{6.5a}$$

$$\text{AR Layer:} \quad \mathcal{A}_{:,:,t} * \mathcal{Y}_{:,:,t} = \mathcal{T}_{:,:,t}, \tag{6.5b}$$

where $\mathcal{T} \in \mathbb{R}^{I_1' \times I_2' \times T}$ is the intermediate result.

Difficulty in computing the AR layer. While the MA layer in Equation (6.5a) is simply a traditional convolutional layer defined in Equation (6.1), it is nontrivial to solve the AR layer in Equation (6.5b). In principle, the linear equations in the AR layer can be solved in time cubic in dimension $O((I_1^2 + I_2^2)I_1 I_2 T)$ using Gaussian elimination. However, this is too expensive in practice.

Computing the AR layer in frequency domain. We propose to use the frequency-domain division to solve the *deconvolution* problem in the AR layer [168]. Since the convolution in the spatial domain leads to an element-wise product in the frequency domain, we first transform $\mathcal{A}, \mathcal{T}$ into their frequency representations $\widetilde{\mathcal{A}}, \widetilde{\mathcal{T}}$, with which we compute $\widetilde{\mathcal{Y}}$

Layer	# params.	# FLOPs	$r(ERF)^2$
Conv.	$K_w^2 C^2$	$O(I^2 K_w^2 C^2)$	$O(LK_w^2)$
ARMA	$K_w^2 C^2 + K_a^2 C$	$O(K_w^2 I^2 C^2 + I^2 \log(I) C)$	$O\big(LK_w^2 + La/(1-a)^2\big)$

Table 6.1: **Comparison between traditional convolutional layer and ARMA layer.** An ARMA layer achieves large gain of the ERF radius through small overhead of extra parameters and FLOPs. Through a single extra parameter a (thus $K_a = 2$), the ERF radius can be arbitrarily large. For simplicity, we assume all heights and widths are equal $I_1 = I_2 = I'_1 = I'_2 = I$, and the input and output channels are identical $S = T = C$.

(the frequency representation of $\mathcal{Y}$) with the element-wise division. Then, we reconstruct the output $\mathcal{Y}$ by an inverse Fourier transform of $\widetilde{\mathcal{Y}}$.

Computational overhead. ARMA trades small overhead of extra parameters and computation for a large gain of ERF radius, as shown in Table 6.1. With the *Fast Fourier Transform* (FFT) algorithm, the FLOPS required by the extra autoregressive layer is $O(\log(\max(I_1, I_2)) I_1 I_2 T)$. Importantly, compared with non-local attention block [148], the extra computation introduced in an ARMA layer is smaller; a non-local attention block requires $O(I_1^2 I_2^2 T)$ FLOPS.

Backpropagation. Deriving the backpropagation for an ARMA layer is nontrivial; though the rule for the MA layer is conventional, that of the AR layer is not. In Theorem 6.4, we show that the backpropagation of an AR layer can be computed as two ARMA models.

Theorem 6.4 (**Backpropagation of an AR layer**). *Given an AR layer* $\mathcal{A}_{:,:,t} * \mathcal{Y}_{:,:,t} = \mathcal{T}_{:,:,t}$ *and its output gradient* $\partial \mathcal{L}/\partial \mathcal{Y}$*, the gradients* $\{\partial \mathcal{L}/\partial \mathcal{A}, \partial \mathcal{L}/\partial \mathcal{X}\}$ *can be obtained by* *two ARMA models:*

$$\mathcal{A}^{\top}_{:,:,t} * \frac{\partial \mathcal{L}}{\partial \mathcal{A}_{:,:,t}} = -\mathcal{Y}^{\top}_{:,:,t} * \frac{\partial \mathcal{L}}{\partial \mathcal{Y}_{:,:,t}}, \tag{6.6a}$$

$$\mathcal{A}^{\top}_{:,:,t} * \frac{\partial \mathcal{L}}{\partial \mathcal{T}_{:,:,t}} = \frac{\partial \mathcal{L}}{\partial \mathcal{Y}_{:,:,t}}, \tag{6.6b}$$

where $\mathcal{A}^{\top}_{:,:,t}$, $\mathcal{Y}^{\top}_{:,:,t}$ are the transposed images of $\mathcal{A}_{:,:,t}$, $\mathcal{Y}_{:,:,t}$ (e.g., $\mathcal{A}^{\top}_{i_1,i_2,t} = \mathcal{A}_{-i_1,-i_2,t}$.

We prove Theorem 6.4 in [167]. Since the backpropagation is characterized by ARMA models, it can be evaluated efficiently using the FFT algorithm similar to Equation (6.5).

6.3.4 Stability of ARMA Layers

An ARMA model with arbitrary coefficients is not always stable. For example, the model $y_i - a y_{i-1} = x_i$ is unstable if $|a| > 1$: Consider an input $\boldsymbol{x}$ with $x_0 = 1$ and $x_i = 0, \forall i \neq 0$, the output $\boldsymbol{y}$ will recursively amplify itself as $y_0 = 1, y_1 = a, \cdots, y_i = a^i$ and diverge to infinity.

Stability constraints for an ARMA layer. Therefore, the key to the stability of an ARMA layer is to constrain its autoregressive coefficients, which prevents the output from repeatedly amplifying itself. To derive the constraints, we propose a special design, *separable ARMA layer* inspired by *separable filters* [168].

Definition 6.5 (Separable ARMA layer). *A separable ARMA layer is parameterized by a moving-average kernel $\mathcal{W} \in \mathbb{R}^{K_w \times K_w \times S \times T}$ and $T \times Q$ sets of autoregressive filters $\{(f^{(q)}_{:,t}, g^{(q)}_{:,t})_{q=1}^{Q}\}_{t=1}^{T}$,. It takes an input $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times S}$ and returns an output $\mathcal{Y} \in \mathbb{R}^{I'_1 \times I'_2 \times T}$ as*

$$\left(f^{(1)}_{:,t} * \cdots * f^{(Q)}_{:,t}\right) \otimes \left(g^{(1)}_{:,t} * \cdots * g^{(Q)}_{:,t}\right) * \mathcal{Y}_{:,:,t} = \sum_{s=1}^{S} \mathcal{W}_{:,:,t,s} * \mathcal{X}_{:,:,s} \tag{6.7}$$

where the filters $f^{(q)}_{:,t}, g^{(q)}_{:,t} \in \mathbb{R}^3$, and $\otimes$ denotes outer product of two 1D-filters.

Remarks: Each autoregressive filter $\mathcal{A}_{:,:,t}$ is designed to be separable, i.e., $\mathcal{A}_{:,:,t} = F_{:,t} \otimes G_{:,t}$, thus it can be characterized by two 1D-filters $F_{:,t}$ and $G_{:,t}$. By the fundamental theorem of algebra [169], any 1D-filter can be represented as a composition of length-3 filters. Therefore, $F_{:,t}$ and $G_{:,t}$ can further be factorized as $F_{:,t} = f^{(1)}_{:,t} * f^{(2)}_{:,t} \cdots * f^{(Q)}_{:,t}$ and $G_{:,t} = g^{(1)}_{:,t} * g^{(2)}_{:,t} \cdots * g^{(Q)}_{:,t}$. In summary, each $\mathcal{A}_{:,:,t}$ is characterized by Q sets of length-3 autoregressive filters $(f^{(q)}_{:,t}, g^{(q)}_{:,t})_{q=1}^{Q}$.

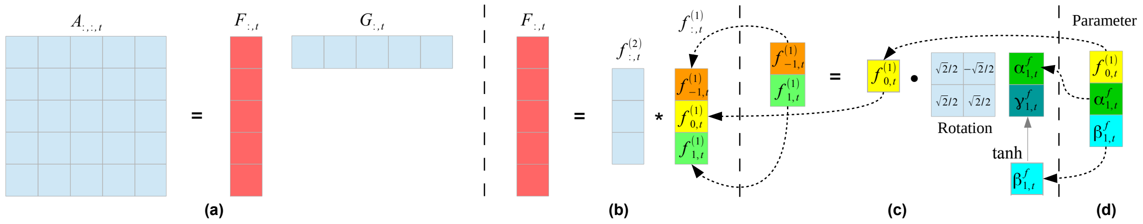


Figure 6.4: **Illustration of the re-parametrization mechanism.** For each channel t, **(a)** the two-dimensional filter $\mathcal{A}_{:,:,t}$ is parameterized through an outer product of two 1D-filters $F_{:,t}$ and $G_{:,t}$; **(b)** $F_{:,t}$ is parameterized through a convolution of $f^{(1)}_{:,t} * \cdots * f^{(Q)}_{:,t}$, and similarly $G_{:,t}$ as a convolution of $g^{(1)}_{:,t} * \cdots * g^{(Q)}_{:,t}$; **(c)** we re-parameterize each constrained $(f^{(q)}_{-1,t}, f^{(q)}_{1,t})$ to unconstrained $(\alpha^f_{q,t}, \beta^f_{q,t})$, and similarly $(g^{(q)}_{-1,t}, g^{(q)}_{1,t})$ to $(\alpha^g_{q,t}, \beta^g_{q,t})$; **(d)** final parameters for unconstrained optimization are $(f^{(q)}_{0,t}, \alpha^f_{q,t}, \beta^f_{q,t}, g^{(q)}_{0,t} \alpha^g_{q,t}, \beta^g_{q,t})_{q=1}^{Q}$.

Theorem 6.6 (**Constraints for a stable separable ARMA layer**)**.** *A sufficient condition for the separable ARMA layer (Definition 6.5) to be stable (i.e., output be bounded* *for any bounded input) is:*

$$\left|f^{(q)}_{-1,t} + f^{(q)}_{1,t}\right| < f^{(q)}_{0,t}, \ \left|g^{(q)}_{-1,t} + g^{(q)}_{1,t}\right| < g^{(q)}_{0,t}, \ \forall q \in [Q], t \in [T]. \tag{6.8}$$

We prove Theorem 6.6 in [167], which uses the standard techniques of linear system theory using Z-transform.

Ensuring stability via re-parameterization. In principle, the constraints for stability in an ARMA layer (as in Theorem 6.6) could be enforced through constraints in optimization. However, a constrained optimization algorithm, such as projected gradient descent [170], is more expensive as it requires an extra projection step. Moreover, it is more difficult to achieve convergence. In order to avoid these challenges, we introduce a *re-parameterization* mechanism to remove constraints needed to guarantee stability.

Theorem 6.7 (Re-parameterization)**.** *For a separable ARMA layer in Definition 6.5, if we re-parameterize each tuple* $(f^{(q)}_{-1,t}, f^{(q)}_{1,t}, g^{(q)}_{-1,t}, g^{(q)}_{1,t})$ *as learnable parameters* $(\alpha^f_{q,t}, \beta^f_{q,t}, \alpha^g_{q,t}, \beta^g_{q,t})$*:*

$$\begin{pmatrix} f^{(q)}_{-1,t} & g^{(q)}_{-1,t} \\ f^{(q)}_{1,t} & g^{(q)}_{1,t} \end{pmatrix} = \begin{pmatrix} f^{(q)}_{0,t} & 0 \\ 0 & g^{(q)}_{0,t} \end{pmatrix} \begin{pmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix} \begin{pmatrix} \alpha^f_{q,t} & \alpha^g_{q,t} \\ \tanh(\beta^f_{q,t}) & \tanh(\beta^g_{q,t}) \end{pmatrix} \tag{6.9}$$

then the layer is stable for ***unconstrained*** $\{(f^{(q)}_{0,t}, \alpha^f_{q,t}, \beta^f_{q,t}, g^{(q)}_{0,t}, \alpha^g_{q,t}, \beta^g_{q,t})_{q=1}^{Q}\}_{t=1}^{T}$.

In practice, we set $f^{(q)}_{0,t} = g^{(q)}_{0,t} = 1$ (since the scale can be learned by the moving-average kernel), and only store and optimize over each tuple $(\alpha^f_{q,t}, \beta^f_{q,t}, \alpha^g_{q,t}, \beta^g_{q,t})$. In other words, each autoregressive filter $\mathcal{A}_{:,:,t}$ is constructed from $(\alpha^f_{q,t}, \beta^f_{q,t}, \alpha^g_{q,t}, \beta^g_{q,t})_{q=1}^{Q}$ on the fly during training or inference.

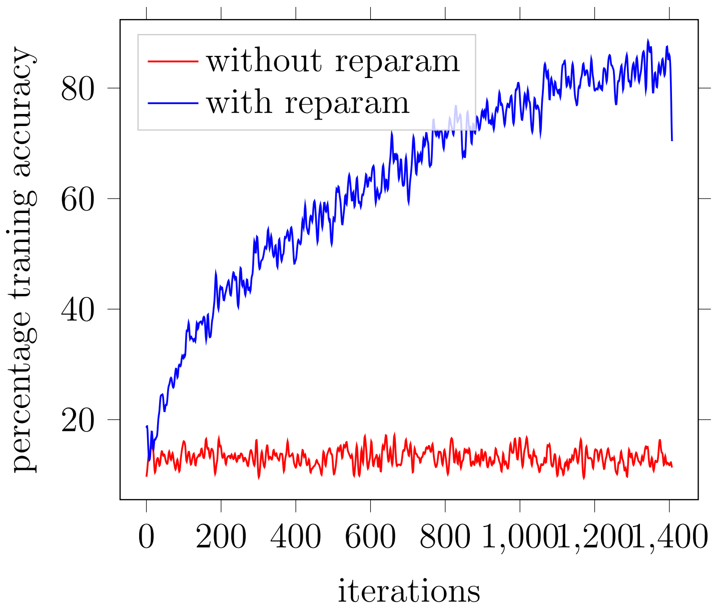


Figure 6.5: **Learning curves with/without re-parameterization mechanism.** The ARMA network with VGG-11 backbone is trained on the CIFAR-10 dataset.

Experimental demonstration of re-parameterization. To verify that such mechanism is essential for stable training, we train a VGG-11 network [2] on the CIFAR-10 dataset, where all convolutional layers are replaced by ARMA layers with autoregressive coefficients initialized as zeros. We compare the learning curves using the re-parameterization v.s. not using the re-parameterization in Figure 6.5. As we can see, the training quickly converges under our proposed re-parameterization mechanism with which the stability of the network is guaranteed. However, without the re-parameterization mechanism, a naive training of the ARMA network never converges and gets NaN error quickly. The experiment thus verifies that the theory in Theorem 6.7 is effective in guaranteeing stability.

6.4 Experimental Results

We apply our ARMA networks on two dense prediction problems – pixel-level video prediction and semantic segmentation to demonstrate the effectiveness of ARMA networks. **(1)** We incorporate our ARMA layers in U-Nets [145, 171] for semantic segmentation, and in the ConvLSTM network [78, 91] for video prediction. **We show that the resulted ARMA U-Net and ARMA-LSTM models uniformly outperform the baselines on both tasks.** **(2)** We then interpret the varying performance of ARMA networks on different tasks by visualizing the histograms of the learned autoregressive coefficients. We include the detailed setups (datasets, model architectures, training strategies, and evaluation metrics) and visualization in section 6.5 for reproducibility purposes.

Semantic segmentation on biomedical medical images. We evaluate our ARMA U-Net on the lesion segmentation task in the ISIC 2018 challenge [172], comparing against a baseline U-Net [145] and non-local U-Net [171] (U-Net with non-local attention blocks).

Model	params.	ACC	SE	SP	PC	F1	JS
U-Net [145]	3.453M	0.946 ±0.003	0.884 ±0.019	**0.977 ±0.005**	0.857 ±0.020	0.842 ±0.009	0.754 ±0.011
NL U-Net [171]	4.403M	0.945 ±0.003	0.877 ±0.017	0.973 ±0.004	0.844 ±0.014	0.831 ±0.012	0.741 ±0.013
ARMA U-Net	3.455M	0.955 ±0.003	0.896 ±0.011	0.972 ±0.005	**0.873 ±0.011**	0.861 ±0.007	0.780 ±0.009
NL ARMA U-Net	4.405M	**0.960 ±0.002**	**0.909 ±0.009**	0.968 ±0.004	0.870 ±0.011	**0.870 ±0.006**	**0.790 ±0.008**

Table 6.2: **Semantic segmentation on ISIC dataset.** For all metrics (ACC, SE, SP, PC, F1 and JS), higher values indicates better performance. The reported numbers are an average of 10 runs with different seeds.

ARMA networks outperform both baselines in almost all metrics. As shown in Table 6.2, our (non-local) ARMA U-Net outperforms both U-Net and non-local U-Net except for specificity (SP). Furthermore, we find that the synergy of non-local attention and ARMA layers achieves the best results among all.

Pixel-level video prediction. We evaluate our ARMA-LSTM network on the Moving-MNIST-2 dataset [173] with different moving velocities, comparing against the baseline Conv-LSTM network [78, 91] and its augmentation using dilated convolutions and non-local attention blocks [148]. As shown in the visualizations in subsection 6.5.2, the dilated ARMA-LSTM does not have gridding artifacts as in dilated Conv-LSTM; that is, *ARMA removes the gridding artifacts.*

Model	MA	AR	dil.	params.	original speed		2X speed		3X speed	
					PSNR	SSIM	PSNR	SSIM	PSNR	SSIM
Conv-LSTM (size 3)	3	1	1	0.887M	18.24	0.867	16.62	0.827	15.81	0.810
Conv-LSTM (size 5)	5	1	1	2.462M	19.58	0.901	17.61	0.856	16.99	0.841
Dilated Conv-LSTM	3	2	2	0.887M	19.16	0.893	17.92	0.858	17.48	0.846
Dilated ARMA-LSTM	3	3	2	0.893M	**19.72**	**0.904**	18.05	0.870	17.65	0.855
ARMA-LSTM (size 3)	3	2	1	0.893M	**19.72**	0.899	**18.73**	**0.881**	**18.13**	**0.869**

Table 6.3: **Multi-frame video prediction on the Moving-MNIST-2 dataset** with three different speeds. MA and AR denote the size of moving-average and autoregressive kernels, respectively, and dil. the dilation in the moving-average kernel. Higher PSNR and SSIM indicate better performance, and all results are average over 10 predicted frames.

ARMA networks outperform non-local blocks: As shown in Table 6.4, our ARMA-LSTM with kernel sizes 3×3 outperforms the Conv-LSTM networks augmented by non-local blocks. However, the non-local mechanism does not always improve the baselines or our models. When both ARMA-LSTM and Conv-LSTM are combined with non-local blocks, our model achieves better performance compared to the non-ARMA baselines.

Model	MA	AR	dil.	Original PSNR	Original SSIM	Non-local PSNR	Non-local SSIM
Conv-LSTM	3	1	1	18.24	0.867	19.45	0.895
Conv-LSTM	5	1	1	19.58	**0.901**	19.18	0.891
ARMA-LSTM	3	3	1	**19.72**	0.899	**19.62**	**0.897**

Table 6.4: **Comparison with non-local networks on video prediction.** The original networks are the same as in Table 6.3. Each non-local network additionally inserts two non-local blocks in the corresponding base network.

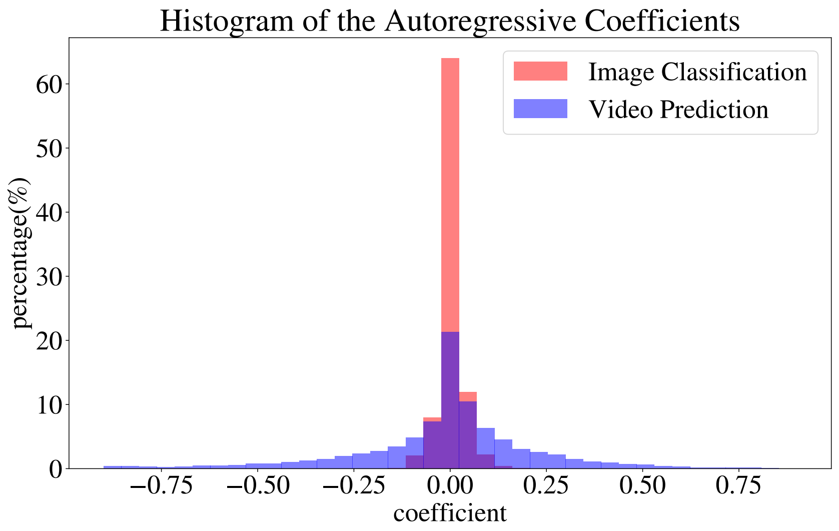


Figure 6.6: **Histogram of the autoregressive coefficients** in ARMA networks.

Interpretation by autoregressive coefficients. Figure 6.6 compares the histograms of the trained autoregressive coefficients between video prediction and image classification to explain why ARMA networks achieve impressive performance in dense prediction, (subsection 6.5.4 demonstrates the performance of ARMA networks in image classification with baselines VGG and ResNet.)

1. The histograms demonstrate how ARMA networks adaptively learn autoregressive coefficients according to the tasks. As motivated in the introduction, dense prediction such as video prediction requires each layer to have a large receptive field to capture

global information.

2. The large autoregressive coefficients in the video prediction model suggest that the overall ERF is significantly expanded. In the image classification model, global information is already aggregated by pooling (downsampling) layers and a fully-connected classification layer. Therefore, the ARMA layers automatically learn nearly zero autoregressive coefficients.

6.5 Supplemental Materials for Empirical Studies

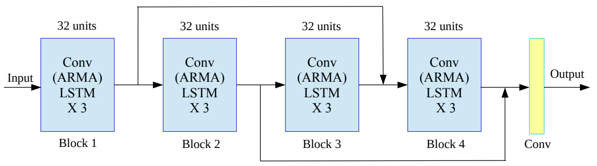
In this section, we explain detailed setups (datasets, model architectures, learning strategies, and evaluation metrics) of all experiments, and provide additional visualizations of the results.

6.5.1 Visualization of Effective Receptive Fields

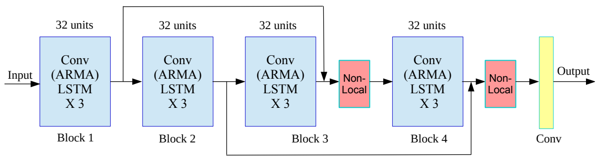
In the simulations in Section 6.3.2, all linear networks have 32 channels and 64×64 feature size at each layer. The filter size for both moving-average coefficients and autoregressive coefficients is set to 3×3: each moving-average kernel $\mathcal{W}$ is initialized using Xavier's method, while the autoregressive kernel $\mathcal{A}$ is initialized randomly within a stable region $-a \leq f^{(q)}_{-1,t} + f^{(q)}_{1,t} \leq 0, -a \leq g^{(q)}_{-1,t} + g^{(q)}_{1,t} \leq 0, \forall t \in [T], q \in [Q]$ (see Section 6.3.4 for details). We compute each heat map in Figure 6.3 as an average of 32 gradient maps from different channels.

6.5.2 Multi-frame Video Prediction

Datasets and metrics. We generate the Moving-MNIST-2 dataset by moving two digits of size 28×28 drawn from the MNIST dataset within a 64×64 black canvas [173]. Each digit starts from a random initial location, moves with constant velocity in the canvas, and bounces when they reach the boundary independently. In addition to the default velocity in the public generator [173], we increase the velocity to $2\times$ and $3\times$ to test all models on videos with stronger motions. For each velocity, we generate $10,000$ videos for the training set, $3,000$ for the validation set, and $5,000$ for the test set, each of which contains 20 frames. We train all models to predict the next 10 frames given 10 input frames, and we evaluate their performance based on the metrics of *mean square error* (MSE), *peak signal-noise ratio* (PSNR), and *structure similarity* (SSIM) [110].



(a) Conv-LSTM.



(b) Non-Local Conv-LSTM.

Figure 6.7: **Backbone architectures for video prediction.**

Model architectures. **(1) Baselines.** The backbone architecture consists of a stack of 12 Conv-LSTM modules, and each module contains 32 units (channels). Following [91], two skip connections that perform channel concatenation are added between (3, 9) and (6, 12) module. An additional traditional convolutional layer is applied on top of all recurrent layers to compute the predicted frames. The backbone architecture is illustrated in Figure 6.7a. In the baseline networks, we consider three different convolutions at each layer: **(a)** Traditional convolution with filter size 3×3; **(b)** Traditional convolution with filter size 5×5; and **(c)** 2-dilated convolution with filter size 3×3. **(2) ARMA networks.** Our ARMA networks use the same backbone architecture as baselines and replace their convolutional layers with ARMA layers. For all ARMA models, we set the filter size for both moving-average and autoregressive parts to 3×3. In the ARMA networks, we consider two different convolutions each layer: **(a)** The moving-average part is a traditional convolution; **(b)** We further consider using 2-dilated convolution in the moving-average part. **(3) Non-local networks.** In non-local networks, we additionally insert two non-local blocks in the backbone architecture, as illustrated in Figure 6.7b. In each non-local block, we use embedded Gaussian as the non-local operation [148], and we replace the batch normalization by instance normalization that is compatible with recurrent neural networks. In non-local networks, we consider three types of convolutions at each layer: **(a)(b)** Traditional convolutions with filter size 3×3 and 5×5; **(c)** ARMA layer with 3×3 moving-average and autoregressive filters.

Visualization of the predictions. We visualize the predictions by different models under three moving velocities in Figure 6.8, Figure 6.9, and Figure 6.10. Notice that autoregression removes the gridding artifacts by dilated convolutions — since each neuron receives information from all pixels in a local region (Figure 6.2d), adjacent neurons no longer receive from separate sets of pixels. Moreover, for videos with a higher speed, the advantage of our ARMA layer is more pronounced as expected due to ARMA's ability to expand the ERF.

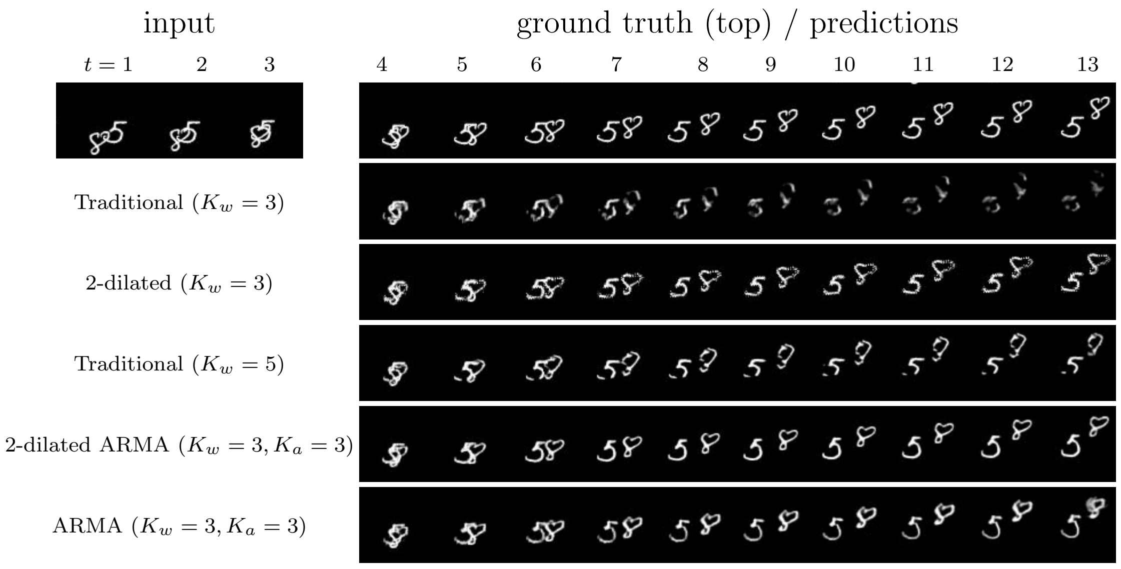


Figure 6.8: **Prediction on Moving-MNIST-2 (original speed)**. The first row contains the last 3 input frames and 10 ground-truth frames for models to predict.

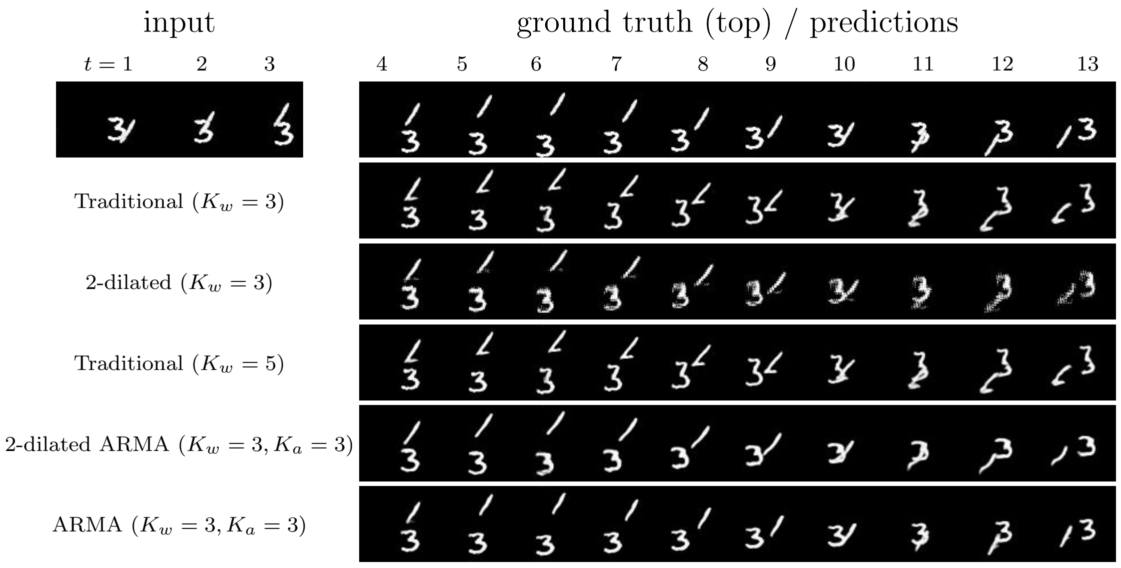


Figure 6.9: **Prediction on Moving-MNIST-2 (2× speed)**. The first row contains the last 3 input frames and 10 ground-truth frames for models to predict.

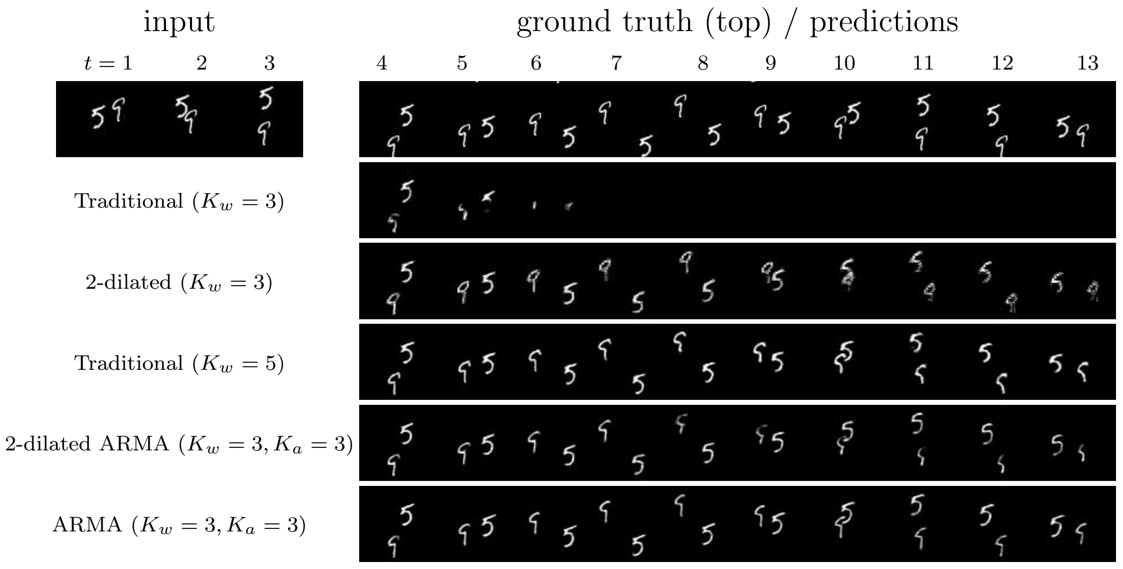
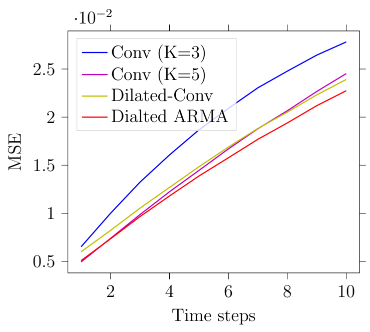
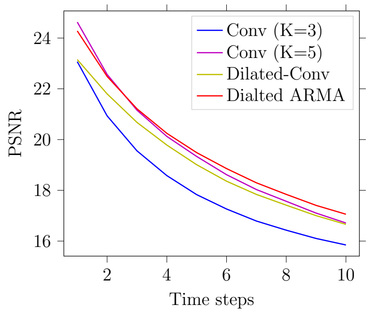


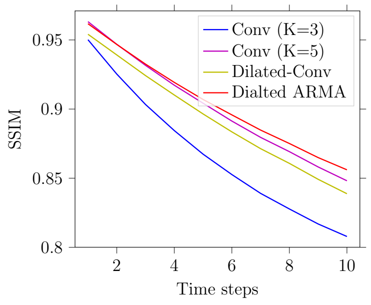
Figure 6.10: **Prediction on Moving-MNIST-2 (3× speed)**. The first row contains the last 3 input frames and 10 ground-truth frames for models to predict.



(a) MSE

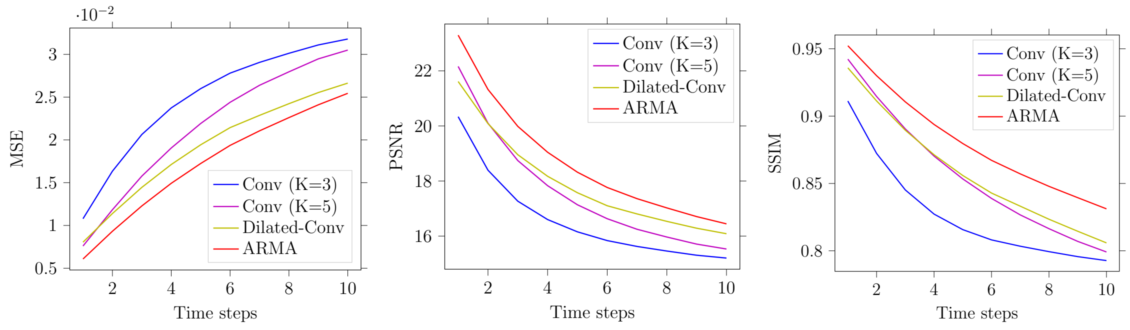


(b) PSNR



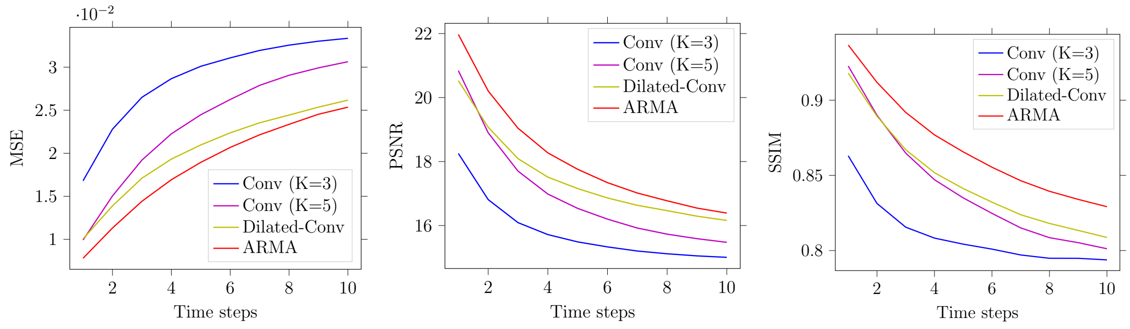
(c) SSIM

Figure 6.11: **Per-frame performance comparison** of our ARMA and our dilated ARMA networks v.s. the Conv-LSTM, dilated Conv-LSTM baselines for Moving-MNIST-2 (original speed). Lower MSE (in 10^{-3}) and higher PSNR/SSIM indicate better performance.



(a) MSE (b) PSNR (c) SSIM

Figure 6.12: **Per-frame performance comparison** of our ARMA and our dilated ARMA networks v.s. the Conv-LSTM, dilated Conv-LSTM baselines for Moving-MNIST-2 (2$\times$ speed). Lower MSE (in 10^{-3}) and higher PSNR/SSIM indicate better performance.

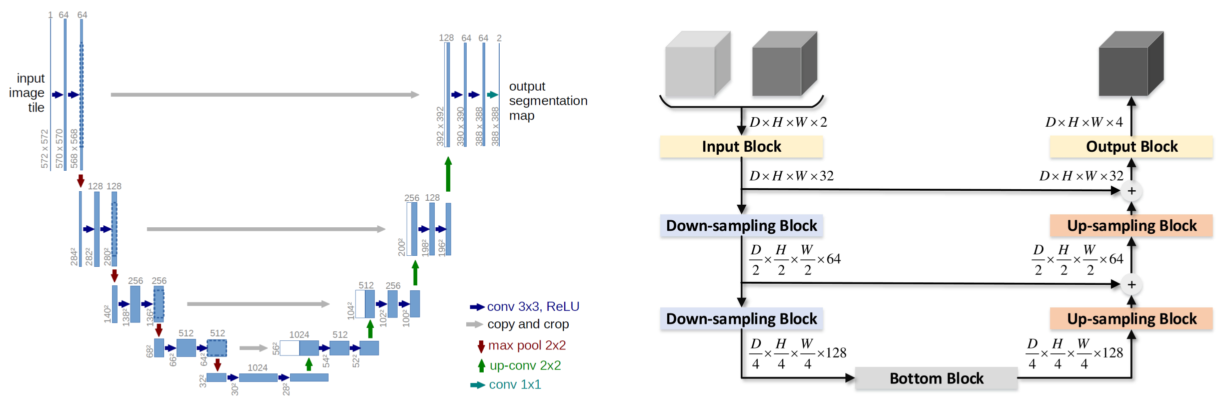


(a) MSE (b) PSNR (c) SSIM

Figure 6.13: **Per-frame performance comparison** of our ARMA and our dilated ARMA networks v.s. the Conv-LSTM, dilated Conv-LSTM baselines for Moving-MNIST-2 (3$\times$ speed). Lower MSE (in 10^{-3}) and higher PSNR/SSIM indicate better performance.

6.5.3 Medical Image Segmentation

Dataset and metrics. For all experiments, we use a dataset from ISIC 2018: Skin Lesion Analysis Towards Melanoma Detection [174], which can be downloaded online[1]. In this task, a model aims to predict a binary mask that indicates the location of the primary skin lesion for each input image. The dataset consists of 2594 images, and we resize each image to 224×224. We split the dataset into a training set, validation set and test set with ratios of 0.7, 0.1, and 0.2, respectively. All models are evaluated using the following metrics: $AC = (TP+TN)/(TP+TN+FP+FN)$, $SE = TP/(TP+FN)$, $SP = TN/(TN+FP)$, $PC = TP/(TP+FP)$, $F1 = 2PC \cdot SE/(PC+SE)$ and $JS = |GT \cap SR|/|GT \cup SR|$, where TP stands for true positive, TN for true negative, FP for false positive, FN for false negative, GT for ground truth mask and SR for predictive mask.



(a) U-Net architecture.

(b) Non-local U-Net architecture.

Figure 6.14: **Backbone architectures for semantic segmentation.**

Model architectures. **(1) Baselines.** We use U-Net [145] and non-local U-Net [171] as baseline models. U-Net has a contracting path to capture context, and a symmetric

[1]https://challenge2018.isic-archive.com/task1/training/

expanding path enables precise localization. The network architecture is illustrated in Figure 6.14a. Non-local U-Net further contains global aggregation blocks based on the self-attention operator to aggregate global information without a deep encoder for biomedical image segmentation, as illustrated in Figure 6.14b. **(2) Our architectures.** We replace all traditional convolution layers with ARMA layers in U-Net and non-local U-Net.

Training strategy. All models are trained using Adam optimizer [8] with binary cross entropy (BCE) loss. For the initial learning rate, we search from 10^{-2} to 10^{-5} and choose 10^{-3} for U-Net and 10^{-2} for non-local U-Net. The learning rate is decayed by 0.98 every epoch. During training, each image is randomly augmented by rotation, cropping, shifting, color jitter, and normalization following the public source code `https://github.com/LeeJunHyun/Image_Segmentation`.

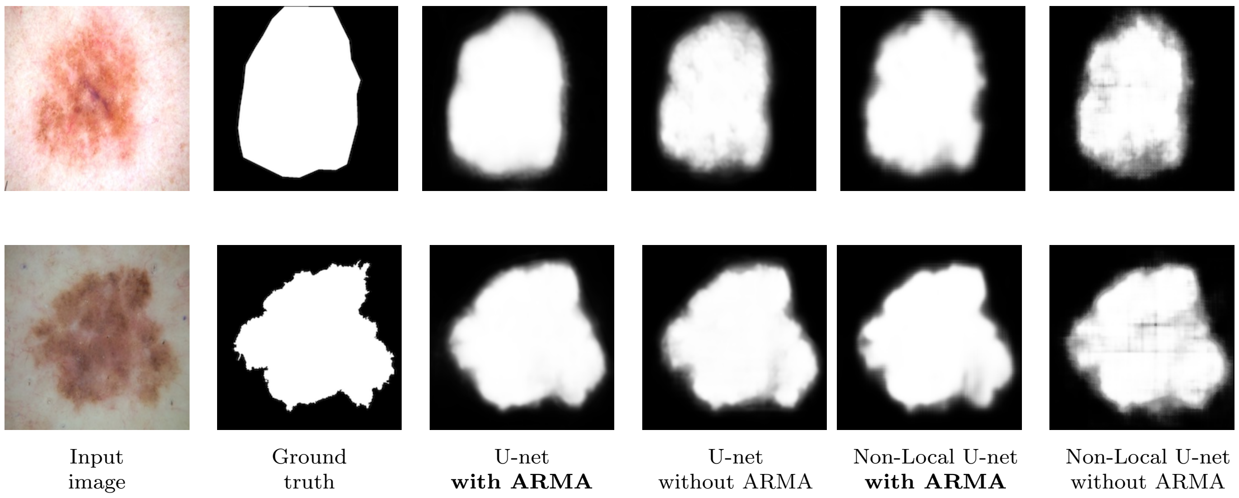


Figure 6.15: **Predictive results of U-Nets with/without ARMA layers.**

6.5.4 Image Classification

Model architectures and datasets. We replace the traditional convolutional layers by ARMA layers in three benchmarking architectures for image classification: AlexNet [1], VGG-11 [2], and ResNet-18 [5, 15]. We apply our proposed ARMA networks on CIFAR10 and CIFAR100 datasets. Both datasets have 50000 training examples and 10000 test examples, and we use 5000 examples from the training set for validation (and leave 45000 examples for training).

Training strategy. We train all models using cross-entropy loss and SGD optimizer with batch size 128, learning rate 0.1, weight decay 0.0005 and momentum 0.9. For CIFAR10, the models are trained for 300 epochs and we half the learning rate every 30 epochs. For CIFAR100, the models are trained for 200 epochs and we divide the learning rate by 5 at the 60th, 120th, 160th epochs.

Results. The experimental results are summarized in Table 6.5. Our results show that ARMA models achieve comparable or slightly better results than the benchmarking architectures. Replacing the traditional convolutional layer with our proposed ARMA layer slightly boosts VGG-11 and ResNet-18 by 0.01%-0.1% in terms of accuracy. **Since image classifications tasks do not require convolutional layers to have large receptive fields, the learned autoregressive coefficients concentrate around 0, as shown in Figure 6.6.** Consequently, ARMA networks effectively reduce to traditional convolutional neural networks and therefore achieve comparable results.

Dataset	AlexNet		VGG-11		ResNet-18	
	Conv.	ARMA	Conv.	ARMA	Conv.	ARMA
CIFAR10	86.30 ± 0.29	85.67 ± 0.19	91.57 ± 0.59	91.57 ± 0.73	95.01 ± 0.15	95.07 ± 0.13
CIFAR100	58.99 ± 0.37	57.43 ± 0.24	68.25 ± 0.11	68.36 ± 1.67	73.71 ± 0.23	73.72 ± 0.52

Table 6.5: **Image classification on CIFAR10 and CIFAR100.** We report accuracy (%) and standard deviations from 10 runs with different seeds.

6.6 Conclusion

In this chapter, we propose a novel ARMA layer capable of expanding a network's effective receptive field adaptively. Compared to a traditional convolutional layer, an ARMA layer has additional interconnections among output neurons realized by convolutions. We address the computation and instability problems in ARMA layers, thus allows ARMA layers to replace traditional convolutional layers with minimal extra cost. We show empirically that ARMA networks consistently improve performance on dense prediction tasks.

Our model is closely related to techniques in signal processing and machine learning. First, an ARMA layer is equivalent to an IIR filter bank in signal processing [175]. It thus opens the opportunity for users to improve ARMA layers using filter bank theory. Alternatively, we can interpret the autoregressive (AR) part as a learnable *spectral normalization* [176] following the moving-average (MA) part. Following this relationship, we can further investigate how to better learn the AR part according to the spectrum of the MA part. Finally, the ARMA layer is a *linear recurrent neural network*, where the recurrent propagations are over the spatial domain (section 6.2). Therefore, an interesting future direction is to use ARMA layers to accelerate the computation in RNNs.

Chapter 7: Scaling-up Diverse Orthogonal Convolutional Networks by a Paraunitary Framework

7.1 Overview

Convolutional neural networks, whose deployment has witnessed extensive empirical success, still exhibit limitations that are not thoroughly studied. Firstly, deep convolutional networks are in general difficult to learn, and their high performance heavily relies on techniques that are not fully understood, such as skip-connections [5], batch normalization [7], specialized initialization [108]. Secondly, they are sensitive to imperceptible perturbations, including adversarial attacks [10] and geometric transformations [13]. Finally, a precise characterization of their generalizability is still under active investigation [177, 178].

Orthogonal networks, which have a "flat" spectrum with all singular values of each linear layer being 1 (thus the output norm $\|\mathbf{y}\|$ equals the input norm $\|\mathbf{x}\|, \forall \mathbf{x}$), alleviate all problems above. As shown in recent works, by enforcing orthogonality in neural networks, we obtain **(1) easier optimization** [179, 180]: since each orthogonal layer preserves the gradient norm during backpropagation, an orthogonal network is free from gradient vanishing/ exploding problems; **(2) robustness against perturbation** [181, 182, 183]: since each orthogonal layer is 1-Lipschitz, an orthogonal network will not amplify any input perturbation to flip the output prediction; **(3) better generalizability** [178]: a network's generalization error is positively related to the standard deviation of each linear layer's singular values, thus encouraging orthogonality lowers the network's generalization error.

Despite the benefits, enforcing orthogonality in convolutional networks is challenging. To avoid strict constraint, orthogonal initialization (dynamical isometry) [184, 185, 186] and orthogonal regularization [180, 187] are opted for the gradient vanishing/exploding problems. However, as they do not enforce *strict* orthogonality (and Lipschitzness), these methods are unsuitable for applications that require strict Lipschitzness, such as adversarial robustness [181] and residual flows [188].

Our goal is to enforce exact orthogonality in state-of-the-art deep convolutional networks without expensive overhead. We identify three main challenges. **Challenge I: Achieving exact orthogonality throughout training.** Prior works such as *orthogonal regularization* [187] and *reshaped kernel orthogonality* [189, 190], while enjoying algorithmic simplicity, fail to meet the requirement of exact orthogonality. Also, note that enforcing the constraint during training is necessary since a post-training orthogonalization can substantially alter the weight values and jeopardize the performance. **Challenge II: Avoiding expensive computations.** An efficient algorithm is crucial for scalability to large networks. Existing work based on projected gradient descent [191], however, requires expensive projection after each update. For instance, the projection step in [191] computes an SVD and flattens the spectrum to enforce orthogonality, which costs $O(\text{size}(\text{feature}) \cdot \text{channels}^3)$ for a convolutional layer. **Challenge III: Scaling-up to state-of-the-art deep convolutional networks.** There are many variants to the standard convolutional layer essential for state-of-the-art networks, including dilated, strided, group convolutions. However, none of the existing methods proposes mechanisms to orthogonalize these variants. The lack of techniques, as a result, limits the broad applications of orthogonal convolutional layers to state-of-the-art networks.

We resolve **challenges I, II & III** by proposing a parameterization of orthogonal convolutions. First, using the convolution theorem [192] (spatial convolution is equivalent to spectral product), we reduce the problem of designing orthogonal convolutions to constructing unitary matrices for all frequencies, i.e., paraunitary systems [175]. Further using a complete factorization theorem of paraunitary systems, we obtain a parameterization in the spatial domain for *all* orthogonal 1D-convolutions and separable 2D-convolutions (no work achieves a complete parameterization of all orthogonal 2D-convolutions), which attains high expressiveness, computational/memory efficiency, and exact orthogonality. Since no previous approach achieved exact orthogonality, we are the first to show how essential exact orthogonality is in different types of networks — we observe that exact orthogonality in deeper architectures (with > 10 layers) is beneficial to obtain robust performance.

Furthermore, we unify orthogonal convolution variants (dilated, strided, and group convolutions) as paraunitary systems, allowing us to orthogonalize these variants using the same parameterization for standard convolutions. (No previous work presents mechanisms for learning these variants' orthogonal versions.) Since these variants are crucial in advanced architectures, our work makes it possible to generate the orthogonal counterparts of these architectures, allowing us to investigate their performance, which was not possible before. Combined with our study in skip-connection and initialization, we scale orthogonal networks to deep architectures, including ResNet and ShuffleNet, substantially outperforming their shallower counterparts (Section 7.6). Finally, we show how to deploy our orthogonal networks in Residual Flow [188], a flow-based generative model that requires strict Lipschitzness (Section 7.7.3).

Contributions. In summary, we have made the following contributions:

1. We establish the equivalence between orthogonal convolutions in the spatial domain and paraunitary systems in the spectral domain. Therefore, we interpret and unify all existing approaches as implicit designs of paraunitary systems.

2. Based on a factorization theorem of paraunitary systems, we propose a complete parameterization of orthogonal 1D-convolutions and separable 2D-convolutions, which attains high expressiveness, exact orthogonality (machine-epsilon), and computational/memory efficiency ($<$ 50% memory of previous methods). These features are crucial in learning deep orthogonal networks with state-of-the-art performance.

3. We prove that orthogonality for various convolutional layers (strided, dilated, group) are also entirely characterized by paraunitary systems. Consequently, our parameterization easily extends to these variants, ensuring their exact orthogonality, completeness, and efficiency.

4. We study the design considerations (choices of skip connection, initialization, depth, width, kernel size) for orthogonal networks and show that orthogonal networks can scale to deep architectures (e.g., ResNet, ShuffleNet).

7.2 Related Works

Dynamical isometry [184, 185, 186, 193] aims to address the gradient vanishing/exploding problems in deep vanilla networks with *orthogonal initialization*. These works focus on understanding the interplay between initialization methods and various nonlinear activations. However, these approaches do not guarantee orthogonality (and Lipschitzness) after training, thus are unsuitable for applications that require strict Lipschitz bounds, such as adversarial robustness [181, 182] and residual flows [188, 194].

Learning orthogonality has three typical families of methods: *regularization*, *parameterization* (i.e., mapping unconstrained parameters to the feasible set with a surjective function), *projected gradient descent* (PGD) / *Riemannian gradient descent* (RGD). While the regularization approach is approximate, the latter two learn exact orthogonality. Among these approaches, PGD/RGD requires modification of the optimizer, whereas the other two are compatible with standard optimizersx. **(1)** For **orthogonal matrices**, various regularization methods are proposed in [195] and [196]. Alternatively, numerous parameterization methods exist, including *Householder reflections* [197], *Given rotations* [198], *Cayley transform* [199], *matrix exponential* [200], and *algorithmic unrolling* [181, 201]. Lastly, [189] propose PGD via *singular value clipping*, and [202, 203] consider RGD. **(2)** For **orthogonal convolutions**, some existing works learn orthogonality for the *flattened matrix* [189, 190, 196] or each *output channel* [204]. However, these methods do not lead to orthogonality (norm preserving) of the operation as a whole. [191] propose to use PGD via *singular value clipping and masking* — however, singular value decomposition is expensive, and masking can lead to approximate orthogonality. To the best of our knowledge, there is no accurate PGD or RGD for orthogonal convolutions. Alternatively, recent works adopt parameterizations, using *block convolutions* [182], *Cayley transform* [183], or *convolution* *exponential* [205].

Paraunitary systems are extensively studied in filter banks and wavelets [175, 206, 207]. Classic theory shows that 1D-paraunitary systems are completely characterized by a spectral factorization (see Chapter 14 of [175] or Chapter 5 of [206]), but not all multi-dimensional (MD) paraunitary systems admit a factorized form (see Chapter 8 of [207]). While the complete characterization of MD-paraunitary systems is known in theory (which requires solving a system of nonlinear equations) [208, 209], most practical constructions use separable paraunitary systems [207] and special classes of non-separable paraunitary systems [210]. The equivalence between orthogonal convolutions and paraunitary systems thus opens the opportunities to apply these classic theories in designing orthogonal convolutions.

7.3 Designing Orthogonal Convolutions via Paraunitary Systems

Designing an orthogonal convolutional layer $\{\boldsymbol{h}_{t,s} : \mathbf{y}_t = \mathbf{h}_{t,s} * \mathbf{x}_s\}_{t=1,s=1}^{T,S}$ (s, t index input/output channels) in the spatial domain is challenging. In terms of matrix-vector product, a convolutional layer has a block-circulant weight matrix, with its $(t,s)^{\text{th}}$ block $\mathsf{Cir}\left(h_{t,s}\right) \in \mathbb{R}^{N \times N}$ as:

$$\mathsf{Cir}\left(h_{t,s}\right) = \begin{bmatrix} h_{t,s}[1] & h_{t,s}[N] & \cdots & h_{t,s}[2] \\ h_{t,s}[2] & h_{t,s}[1] & h_{t,s}[N] & \cdots \\ \vdots & \ddots & \ddots & \vdots \\ h_{t,s}[N] & \cdots & h_{t,s}[2] & h_{t,s}[1] \end{bmatrix}. \tag{7.1}$$

Therefore, the layer is orthogonal if block-circulant matrix $[\mathsf{Cir}\,(h_{t,s})]_{t=1,s=1}^{T,S}$ is orthogonal. However, it is not obvious how to enforce orthogonality in a block-circulant matrix.

7.3.1 Achieving Orthogonal Convolutions by Paraunitary Systems

We propose a novel design of orthogonal convolutions from a spectral perspective, motivated by the *convolution theorem* (Theorem 7.1). For simplicity, we group the entries at the same locations a vector/matrix, e.g., we denote $\{x_s[n]\}_{s=1}^{S}$ as $\mathbf{x}[n] \in \mathbb{R}^S$ and $\{h_{t,s}[n]\}_{t=1,s=1}^{T,S}$ as $\mathbf{h}[n] \in \mathbb{R}^{T \times S}$.

Theorem 7.1 (Convolution theorem [192]). *A convolution layer* $\mathbf{h}$*:* $\mathbf{y}[i] = \sum_n \mathbf{h}[n]\mathbf{x}[i-n]$ *in the spatial domain is equivalent to matrix-vector products in the spectral domain, i.e.,* $\mathbf{Y}(z) = \mathbf{H}(z)\mathbf{X}(z), \forall z \in \mathbb{C}$*. Here,* $\mathbf{X}(z) = \sum_{n=0}^{N-1} \mathbf{x}[n]z^{-n}$*,* $\mathbf{Y}(z) = \sum_{n=0}^{N-1} \mathbf{y}[n]z^{-n}$*,* $\mathbf{H}(z) = \sum_{n=-\underline{L}}^{\overline{L}} \mathbf{h}[n]z^{-n}$ *denote the* z*-transforms of input, output, kernel respectively, where* N *is the length of* $\mathbf{x}, \mathbf{y}$ *and* $[-\underline{L}, \overline{L}]$ *is the span of the filter* $\mathbf{h}$*.*

The convolution theorem states that a convolution layer is a matrix-vector product in the spectral domain. If the *transfer matrix* $\mathbf{H}(z)$ is unitary at $z = e^{\mathsf{j}\omega}$ for all frequencies $\forall \omega \in [0, 2\pi)$ (j is the imaginary unit), the layer $\mathbf{h}$ is orthogonal.

As our major novelty, we design orthogonal convolutions via construction of unitary transfer matrix $\mathbf{H}(e^{\mathsf{j}\omega})$ at all frequencies $\omega \in [0, 2\pi)$, known as a *paraunitary system* [175]. The *paraunitary theorem* (Theorem 7.2) shows that a convolutional layer is orthogonal in the spatial domain if and only if it is paraunitary in the spectral domain.

Theorem 7.2 (Paraunitary theorem). *A standard convolutional layer is orthogonal if* *and only if its transfer matrix* $\mathbf{H}(z)$ *is paraunitary, i.e.,*

$$\mathbf{H}(z)^\top \mathbf{H}(z) = \boldsymbol{I},\ \forall |z| = 1 \Longleftrightarrow \mathbf{H}(e^{\mathrm{j}\omega})^\top \mathbf{H}(e^{\mathrm{j}\omega}) = \boldsymbol{I},\ \forall \omega \in \mathbb{R}. \tag{7.2}$$

In other words, the transfer matrix $\mathbf{H}(e^{\mathrm{j}\omega})$ *is unitary for all frequencies* $\omega \in \mathbb{R}$.

Benefits through paraunitary systems. **(1)** The spectral representation simplifies the designs of orthogonal convolutions, which avoids analysis of block-circulant matrices. **(2)** Since paraunitary systems are necessary and sufficient for orthogonal convolutions, it is *impossible* to find an orthogonal convolution whose transfer matrix is *not* paraunitary. **(3)** There exists a complete factorization of paraunitary systems: any paraunitary $\mathbf{H}(z)$ is a product of multiple factors in the spectral domain (Equation (7.3a)). **(4)** Since spectral multiplications correspond to spatial convolutions, *any* orthogonal convolution can be realized as cascaded convolutions, each parameterized by an orthogonal matrix (Equation (7.3b)). **(5)** There are mature methods that parameterize orthogonal matrices via unconstrained parameters. Consequently, we can learn orthogonal convolutions using standard optimizers on a model parameterized via our design.

Interpretation of existing methods. Since paraunitary system is necessary and sufficient for orthogonal convolution, all existing approaches, including *singular value clipping and masking (SVCM)* [191], *block convolution orthogonal parameterization (BCOP)* [182], *Cayley Convolution (CayleyConv)* [183], *skew orthogonal convolution (SOC) [205]* construct paraunitary systems implicitly (see [211] for interpretations).

7.3.2 Parameterization of Paraunitary Systems

After reducing the problem of orthogonal convolutions to paraunitary systems, we are left with how to realize paraunitary systems. To address this, we use a complete factorization to realize any paraunitary system as shown in the following theorem.

Theorem 7.3 (Complete factorization for 1D-paraunitary systems). *Suppose that a paraunitary system* $\mathbf{H}(z)$ *is finite-length, i.e., it can be written as* $\sum_n \mathbf{h}[n]z^{-n}$ *for some sequence* $\{\mathbf{h}[n], n \in [-\underline{L}, \overline{L}]\}$*, then it can be factorized in the following form:*

$$\mathbf{H}(z) = \mathbf{V}(z; \boldsymbol{U}^{(-\underline{L})}) \cdots \mathbf{V}(z; \boldsymbol{U}^{(-1)})\mathbf{Q}\mathbf{V}(z^{-1}; \boldsymbol{U}^{(1)}) \cdots \mathbf{V}(z^{-1}; \boldsymbol{U}^{(\overline{L})}), \tag{7.3a}$$

$$\text{where } \mathbf{V}(z; \boldsymbol{U}^{(\ell)}) = \boldsymbol{I} - \boldsymbol{U}^{(\ell)}{\boldsymbol{U}^{(\ell)}}^\top + \boldsymbol{U}^{(\ell)}{\boldsymbol{U}^{(\ell)}}^\top z, \forall \ell \in \{-\underline{L}, \cdots, -1\} \cup \{1, \cdots, \overline{L}\}. \tag{7.3b}$$

Here $\boldsymbol{Q}$ *is an orthogonal matrix, and* $\boldsymbol{U}^{(\ell)}$ *is a column-orthogonal matrix. Consequently, the paraunitary system* $\mathbf{H}(z)$ *is parameterized by* $\underline{L} + \overline{L} + 1$ *(column-)orthogonal matrices* $\boldsymbol{Q}$ *and* $\boldsymbol{U}^{(\ell)}$*'s.*

As spectral multiplications are equivalent to spatial convolutions, the *complete* spectral factorization of paraunitary systems in Equation (7.3a) allows us to parameterize *any* orthogonal convolution in the spatial domain as cascaded convolutions of $\mathbf{V}(z; \boldsymbol{U}^{(\ell)})$'s spatial counterparts and the orthogonal matrix $\boldsymbol{Q}$.

Model design in the spatial domain. Following Equation (7.3), we obtain a *complete design of orthogonal 1D-convolutions*: using *learnable (column)-orthogonal matrices* $(\{\boldsymbol{U}^{(\ell)}\}_{\ell=-\underline{L}}^{-1}, \boldsymbol{Q}, \{\boldsymbol{U}^{(\ell)}\}_{\ell=1}^{\overline{L}})$, we parameterize a size $(\underline{L} + \overline{L} + 1)$ convolution as cascaded convolutions of the following filters in the spatial domain

$$\left\{\left[\boldsymbol{I} - \boldsymbol{U}^{(\ell)}\boldsymbol{U}^{(\ell)\top},\ \boldsymbol{U}^{(\ell)}\boldsymbol{U}^{(\ell)\top}\right]\right\}_{\ell=-\underline{L}}^{-1}, \boldsymbol{Q},\ \left\{\left[\boldsymbol{U}^{(\ell)}\boldsymbol{U}^{(\ell)\top}, \boldsymbol{I} - \boldsymbol{U}^{(\ell)}\boldsymbol{U}^{(\ell)\top}\right]\right\}_{\ell=1}^{\overline{L}}. \tag{7.4}$$

Figure 7.1 visualizes our design of orthogonal convolution layers; each block denotes a convolution and the filter coefficients are displayed in each block. In practice, we compose all $(\underline{L} + \overline{L} + 1)$ filters into one for orthogonal convolution, which not only increases the computational parallelism but also avoids storing intermediate outputs between filters.

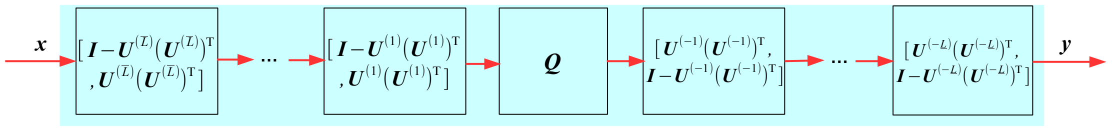

$\boldsymbol{x}$
$[\boldsymbol{I}-\boldsymbol{U}^{(\overline{L})}(\boldsymbol{U}^{(\overline{L})})^{\mathrm{T}}, \boldsymbol{U}^{(\overline{L})}(\boldsymbol{U}^{(\overline{L})})^{\mathrm{T}}]$
...
$[\boldsymbol{I}-\boldsymbol{U}^{(1)}(\boldsymbol{U}^{(1)})^{\mathrm{T}}, \boldsymbol{U}^{(1)}(\boldsymbol{U}^{(1)})^{\mathrm{T}}]$
$\boldsymbol{Q}$
$[\boldsymbol{U}^{(-1)}(\boldsymbol{U}^{(-1)})^{\mathrm{T}}, \boldsymbol{I}-\boldsymbol{U}^{(-1)}(\boldsymbol{U}^{(-1)})^{\mathrm{T}}]$
...
$[\boldsymbol{U}^{(-\underline{L})}(\boldsymbol{U}^{(-\underline{L})})^{\mathrm{T}}, \boldsymbol{I}-\boldsymbol{U}^{(-\underline{L})}(\boldsymbol{U}^{(-\underline{L})})^{\mathrm{T}}]$
$\boldsymbol{y}$

Figure 7.1: **Complete design of 1D orthogonal convolution as a cascade of convolutions.** The filter coefficients are depicted in each block, where $\boldsymbol{Q}$ is orthogonal and $\boldsymbol{U}^{(\ell)}$'s are column-orthogonal.

With a complete factorization of paraunitary systems, we reduce the problem of designing orthogonal convolutions to the one for orthogonal matrices.

Parameterization for orthogonal matrices. There are various parameterizations for orthogonal matrices, including the *Björck orthogonalization* [181, 182], the *Cayley transform* [199, 212], and the *exponential map* [200]. We follow the GeoTorch library [1], which adopts a modified version of exponential map due to its efficiency, exactness, and completeness. The exponential map is a *surjective* mapping from a skew-symmetry matrix $\boldsymbol{A}$ to a special orthogonal matrix $\boldsymbol{U}$ (i.e., $\det(\boldsymbol{U}) = 1$) with $\boldsymbol{U} = \exp(\boldsymbol{A}) = \boldsymbol{I} + \boldsymbol{A} + \boldsymbol{A}^2/2 + \cdots$, which is computed up to machine-precision [213]. To parameterize all orthogonal matrices,

[1]`https://github.com/Lezcano/geotorch`

GeoTorch introduces an orthogonal matrix $\boldsymbol{V}$ in $\boldsymbol{U} = \boldsymbol{V} \exp(\boldsymbol{A})$, where $\boldsymbol{V}$ is (randomly) generated at initialization and fixed during training.

Finally, observe that the upper-triangle entries uniquely determine a skew-symmetric matrix. We now have an end-to-end pipeline in Figure 7.2, which parameterizes orthogonal convolutions by unconstrained upper-triangle entries in skew-symmetric matrices.

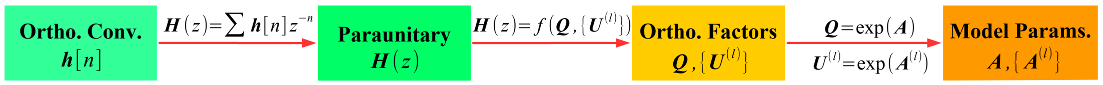


Figure 7.2: **SC-Fac: A pipeline for designing orthogonal convolutional layer. (1)** An orthogonal convolution $\mathbf{h}[n]$ is equivalent a paraunitary system $\mathbf{H}(z)$ in the spectral domain (Theorem 7.1). **(2)** The paraunitary system $\mathbf{H}(z)$ is multiplications of factors characterized by (column-)orthogonal matrices ($\{\boldsymbol{U}^{(\ell)}\}_{\ell=-\underline{L}}^{-1}, \boldsymbol{Q}, \{\boldsymbol{U}^{(\ell)}\}_{\ell=1}^{\overline{L}}$) (Equation (7.3), Theorem 7.2). **(3)** These orthogonal matrices are parameterized by skew-symmetric matrices using exponential map.

7.3.3 Separable Orthogonal 2D-Convolutions

As 2D-convolutions are widely used in convolutional networks, we extend our orthogonal 1D-convolution to the 2D version. Analog to the 1D case, a 2D-convolutional layer is orthogonal if and only if its transfer matrix $\mathbf{H}(z_1, z_2)$ is paraunitary ($\mathbf{H}(z_1, z_2)$ is unitary $\forall |z_1| = 1, |z_2| = 1$).

Construction of orthogonal 2D-convolutions. Using two orthogonal 1D-convolutions, we can readily obtain a complete design of *separable orthogonal 2D-convolutions*, where $\mathbf{H}(z_1, z_2) = \mathbf{H_1}(z_1)\mathbf{H_2}(z_2)$ is a product of two 1D-paraunitary systems. As a result, we can parameterize a separable orthogonal 2D-convolution with filter size $(\overline{L}_1 + \underline{L}_1 + 1) \times (\overline{L}_2 + \underline{L}_2 + 1)$ as a convolution of two orthogonal 1D-convolutions with learnable (column-)orthogonal matrices ($\{\boldsymbol{U}_1^{(\ell)}\}_{\ell=-\underline{L}_1}^{-1}, \boldsymbol{Q}_1, \{\boldsymbol{U}_1^{(\ell)}\}_{\ell=1}^{\overline{L}_1}$) and ($\{\boldsymbol{U}_2^{(\ell)}\}_{\ell=-\underline{L}_2}^{-1}, \boldsymbol{Q}_2, \{\boldsymbol{U}_2^{(\ell)}\}_{\ell=1}^{\overline{L}_2}$). Since our method relies on separability and complete factorization for 1D-paraunitary systems, we call it *Separable Complete Factorization (SC-Fac)*. (See Section 7.7.1 for pseudo code).

Benefits compared to other methods. **(1) Easier analysis.** While BCOP [203] and our SC-Fac are complete in 1D case, none of them is complete in 2D case. However, since separability reduces the design to the 1D case, it makes the analysis of various types of convolutional layers easier, as we will see in Section 7.4. **(2) Efficient inference.** Note that CayleyConv [183] and SOC [205] define the convolution implicitly (as an infinite-length filter), the coefficients for $\mathbf{H}(z_1, z_2)$ can not be saved for repeated inference. In contrast, SC-Fac has the same inference expense as a normal convolutional layer after a one-time computation of the coefficients. **(3) Efficient training.** As shown in [211], SC-Fac also has the lowest computational complexity among all approaches. **(4) Exact orthogonality.** Lastly, as shown in Table 7.2 (Section 7.6.1), SC-Fac achieves exact orthogonality (up to machine-precision), while previous approaches are approximate to a varying degree.

7.4 Unifying Orthogonal Convolutions Variants as Paraunitary Systems

Various convolutional layers (strided, dilated, and group convolution) are widely used in neural networks. However, it is not apparent how to enforce their orthogonality, as the convolution theorem (Theorem 7.1) only holds for *standard* convolutions. Previous approaches only deal with standard convolutions [182, 183, 191], thus orthogonality for state-of-the-art architectures has never been studied before.

We address this limitation by modifying convolution theorem for each variant of convolution layer, which allows us to design these variants using paraunitary systems.

Table 7.1: **Various types of convolutions.** In the following table, we present the modified Z-transforms, $\underline{\mathbf{Y}}(z)$, $\underline{\mathbf{H}}(z)$, and $\underline{\mathbf{X}}(z)$ for each type of convolution such that $\underline{\mathbf{Y}}(z) = \underline{\mathbf{H}}(z)\underline{\mathbf{X}}(z)$ holds. For dilated and strided convolutions, $\mathbf{X}^{r|R}(z)$ is the *(r,R)-polyphase component* of $\mathbf{X}(z)$, with which we define $\mathbf{X}^{[R]}(z) \triangleq [\mathbf{X}^{0|R}(z)^\top, \dots, \mathbf{X}^{R-1|R}(z)^\top]^\top$ and $\widetilde{\mathbf{X}}^{[R]}(z) = [\mathbf{X}^{-0|R}(z), \dots, \mathbf{X}^{-(R-1)|R}(z)]$. For group convolution, $\mathbf{h}^g$ is the filter for the g^{th} group with $\mathbf{H}^g(z)$ being its Z-transform. We stack matrices from different groups into block-diagonal matrices: $\mathbf{h}^{\{G\}}[n] = \mathsf{blkdiag}\left(\{\mathbf{h}^g[z]\}\right)$, $\mathbf{H}^{\{G\}}(z) = \mathsf{blkdiag}\left(\{\mathbf{h}^g[z]\}\right)$.

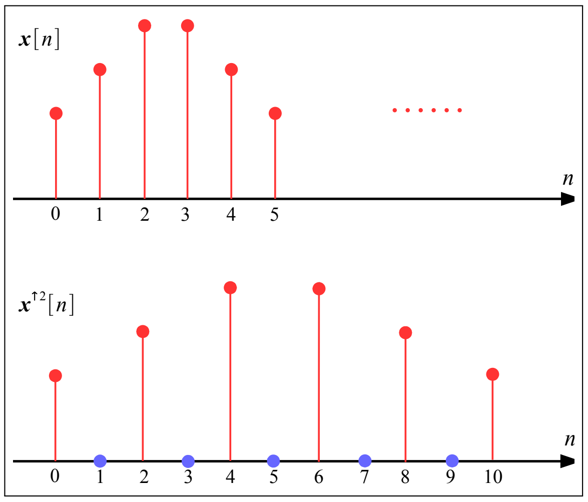
Type	Spatial Representation	Spectral Representation $\underline{\mathbf{Y}}(z)$	$\underline{\mathbf{H}}(z)$	$\underline{\mathbf{X}}(z)$
Standard	$\mathbf{y}[i] = \sum_{n\in\mathbb{Z}} \mathbf{h}[n]\mathbf{x}[i-n]$	$\mathbf{Y}(z)$	$\mathbf{H}(z)$	$\mathbf{X}(z)$
R-Dilated	$\mathbf{y}[i] = \sum_{n\in\mathbb{Z}} \mathbf{h}^{\uparrow R}[n]\mathbf{x}[i-n]$	$\mathbf{Y}(z)$	$\mathbf{H}(z^R)$	$\mathbf{X}(z)$
$\downarrow R$-Strided	$\mathbf{y}[i] = \sum_{n\in\mathbb{Z}} \mathbf{h}[n]\mathbf{x}[Ri-n]$	$\mathbf{Y}(z)$	$\widetilde{\mathbf{H}}^{[R]}(z)$	$\mathbf{X}^{[R]}(z)$
$\uparrow R$-Strided	$\mathbf{y}[i] = \sum_{n\in\mathbb{Z}} \mathbf{h}[n]\mathbf{x}^{\uparrow R}[i-n]$	$\mathbf{Y}^{[R]}(z)$	$\mathbf{H}^{[R]}(z)$	$\mathbf{X}(z)$
G-Group	$\mathbf{y}[i] = \sum_{n\in\mathbb{Z}} \mathbf{h}^{\{G\}}[n]\mathbf{x}[i-n]$	$\mathbf{Y}(z)$	$\mathbf{H}^{\{G\}}(z)$	$\mathbf{X}(z)$

Theorem 7.1 (**Convolution and paraunitary theorems for various convolutions**)**.** *Strided, dilated, and group convolutions can be unified in the spectral domain as* $\underline{\mathbf{Y}}(z) = \underline{\mathbf{H}}(z)\underline{\mathbf{X}}(z)$*, where* $\underline{\mathbf{Y}}(z)$*,* $\underline{\mathbf{H}}(z)$*,* $\underline{\mathbf{X}}(z)$ *are modified Z-transforms of* $\mathbf{y}$*,* $\mathbf{h}$*,* $\mathbf{x}$*. We instantiate the equation for various convolutions in Table 7.1. Furthermore, a convolution is orthogonal if and only if* $\underline{\mathbf{H}}(z)$ *is paraunitary.*

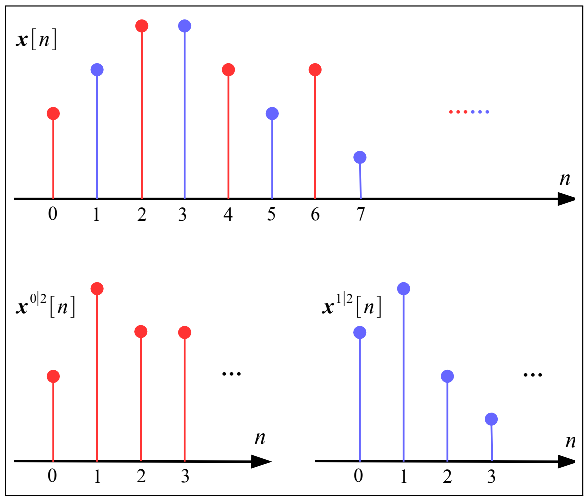
In Table 7.1, we formulate strided, dilated, and group convolutions in the spatial domain, interpreting them as up-sampled or down-sampled variants of a standard convolution. Now, we introduce the concept of up-sampling and down-sampling precisely below.

Given a sequence $\mathbf{x}$, we introduce its *up-sampled sequence* $\mathbf{x}^{\uparrow R}$ with sampling rate R as $\mathbf{x}^{\uparrow R}[n] \triangleq \mathbf{x}[n/R]$ for $n \equiv 0 \pmod R$. On the other hand, its *(r,R)-polyphase component* $\mathbf{x}^{r|R}$ indicates the r-th down-sampled sequence with sampling rate R, defined as $\mathbf{x}^{r|R}[n] \triangleq \mathbf{x}[nR + r]$. We illustrated an example of $\mathbf{x}^{\uparrow R}$ and $\mathbf{x}^{r|R}$ in Figure 7.3 when sampling rate $R = 2$. The Z-transforms of $\mathbf{x}^{\uparrow R}$, $\mathbf{x}^{r|R}$ are denoted as $\mathbf{X}^{\uparrow R}(z)$, $\mathbf{X}^{r|R}(z)$ respectively.

Now we are ready to interpret convolution variants. *(1) Strided convolution* is used to



(a) Up-sample



(b) Down-sample

Figure 7.3: **Up/down sampling.** In **(a)**, the sequence $\mathbf{x}[n]$ is up-sampled into another sequence $\mathbf{x}^{\uparrow 2}[n]$. In **(b)**, the sequence $\mathbf{x}[n]$ is down-sampled into two sequences $\mathbf{x}^{0|2}[n]$ with even entries (red) and $\mathbf{x}^{1|2}[n]$ with odd entries (blue).

adjust the feature resolution: a strided convolution ($\downarrow R$-strided) decreases the resolution by down-sampling after a standard convolution, while a transposed strided convolutional layer ($\uparrow R$-strided) increases the resolution by up-sampling before a standard convolution. *(2) Dilated convolution* increases the receptive field of a convolution without extra parameters: an R-dilated convolution up-samples its filters before convolution with the input. *(3) Group convolution* reduces the parameters and computations, thus widely used by efficient architectures: a G-group convolution divides the input/output channels into G groups and restricts the connections within each group. According to Theorem 7.1, a convolution is orthogonal if and only if its modified Z-transform $\underline{\mathbf{H}}(z)$ is paraunitary.

7.5 Learning Deep Orthogonal Networks with Lipschitz Bounds

In this section, we switch our focus from layer design to network design. In particular, we aim to study how to scale-up deep orthogonal networks with Lipschitz bounds.

Lipschitz networks [181, 182, 183], whose Lipschitz bounds are imposed by their architectures, are proposed as competitive candidates to guarantee robustness in deep learning. A Lipschitz network consists of *orthogonal layers* and *GroupSort activations* — both are 1-Lipschitz and gradient norm preserving. Given a Lipschitz constant L, a Lipschitz network f can compute a certified radius for each input from its output margin. Formally, denote the output margin of an input $\mathbf{x}$ with label c as

$$\mathcal{M}_f(x) \triangleq \max(0, f(\mathbf{x})_c - \max_{i \neq c} f(\mathbf{x})_i), \tag{7.5}$$

i.e., the difference between the correct logit and the second largest logit. Then the output is robust to perturbation such that $f(\mathbf{x} + \epsilon) = f(\mathbf{x}) = c, \forall \epsilon : \|\epsilon\| < \mathcal{M}_f(\mathbf{x})/\sqrt{2}L$.

Despite the benefit, existing architectures for Lipschitz networks remain simple and shallow, and a Lipschitz network is typically an interleaving cascade of orthogonal layers and GroupSort activations [182]. More advanced architectures, such as ResNet and ShuffleNet, are still out of reach. While orthogonal layers supposedly substitute the role of *batch normalization* [180, 184, 186], other critical factors, including *skip-connections* [5, 15] and *proper initialization* [108] are lacking. In this section, we explore skip-connections and initialization methods toward addressing this problem.

Skip-connections. Two classes of skip-connections are widely used in deep networks, one based on *addition* and another on *concatenation*. The addition-based one is proposed in ResNet [5], and adopted in SE-Net [214] and EffcientNet [215], while the concatenation-based one is proposed in flow-based generative models [216, 217, 218], and adopted in

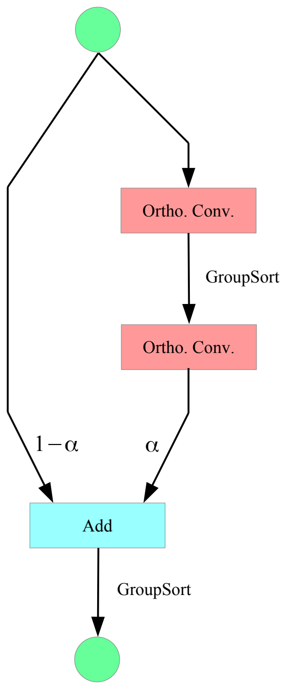
DenseNet [6] and ShuffleNet [219, 220]. In what follows, we propose Lipschitz skip-connections with these two mechanisms, illustrated in Figure 7.4.

Proposition 7.1 (Lipschitzness of residual blocks). *Suppose f^1, f^2 are L-Lipschitz and $\alpha \in [0,1]$ is a learnable scalar, then an additive residual block $f : f(\mathbf{x}) \triangleq \alpha f^1(\mathbf{x}) + (1-\alpha) f^2(\mathbf{x})$ is L-Lipschitz. Alternatively, suppose g^1, g^2 are L-Lipschitz and $\boldsymbol{P}$ is a permutation, then a concatenative residual block $g : g(\mathbf{x}) \triangleq \boldsymbol{P}\left[g^1(\mathbf{x}^1); g^2(\mathbf{x}^2)\right]$ is L-Lipschitz, where $[\cdot;\cdot]$ denotes channel concatenation, and $\mathbf{x}$ is split into $\mathbf{x}_1$ and $\mathbf{x}_2$, i.e., $\mathbf{x} = [\mathbf{x}_1, \mathbf{x_2}]$.*

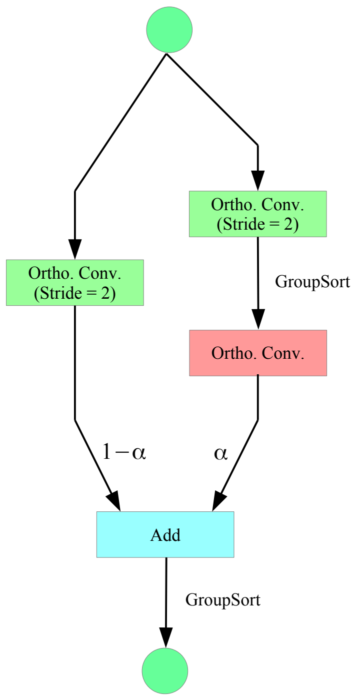
Initialization. Proper initialization is crucial in training deep networks [5, 108]. Various methods are proposed to initialize orthogonal matrices, including the uniform and torus initialization, for orthogonal RNNs [199, 200, 221]. However, initialization of orthogonal convolutions was not systematically studied, and all previous approaches inherit the initialization from the underlying parameterization [182, 183]. In Proposition 7.2, we study the condition when a paraunitary system (in the form of Equation (7.3)) reduces to an orthogonal matrix. This reduction allows us to apply the initialization methods for orthogonal matrices (e.g., uniform, torus) to orthogonal convolutions.

Proposition 7.2 (Reduction of a paraunitary system to an orthogonal matrix). *Suppose a paraunitary system $\mathbf{H}(z)$ takes the complete factorization in Equation (7.3), and assume $\underline{L} = \overline{L}$ with $\boldsymbol{U}^{(-\ell)} = \boldsymbol{Q}\boldsymbol{U}^{(\ell)}$ for all ℓ, then the paraunitary matrix $\mathbf{H}(z)$ reduces to an orthogonal matrix $\boldsymbol{Q}$,*

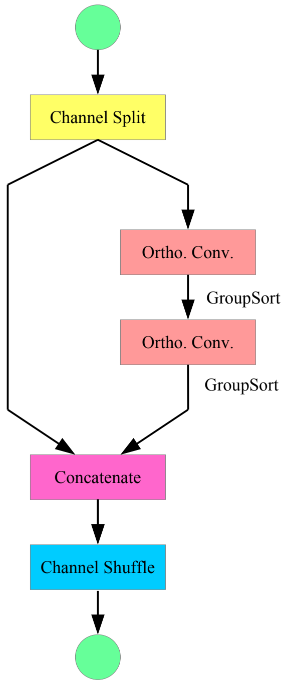
$$\mathbf{H}(z) = \mathbf{V}(z; \boldsymbol{U}^{(-\underline{L})}) \cdots \mathbf{V}(z; \boldsymbol{U}^{(-1)}) \boldsymbol{Q} \mathbf{V}(z^{-1}; \boldsymbol{U}^{(1)}) \cdots \mathbf{V}(z^{-1}; \boldsymbol{U}^{(\overline{L})}) = \boldsymbol{Q}. \tag{7.6}$$



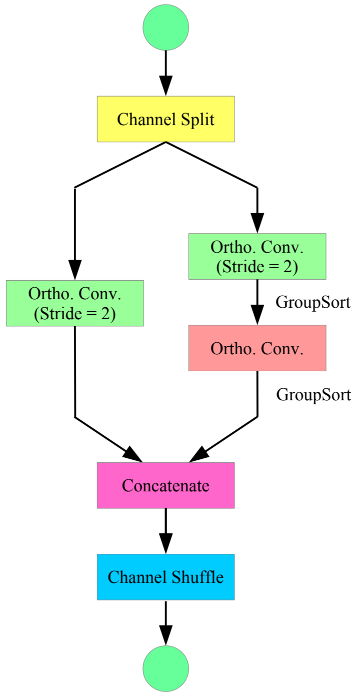
(a) Additive block.



(b) Strided additive block.



(c) Shuffling block.



(d) Strided shuffling block

Figure 7.4: **Variants of residual blocks.** In our experiments, we combine **(a) & (b)** to construct an *orthogonal ResNet*, and **(c) & (d)** to construct an *orthogonal ShuffleNet*. In Proposition 7.1, we prove the Lipschitzness of these building blocks. Since composition of Lipschitz functions is still Lipschitz, it implies that a network constructed by these building blocks is also Lipschitz.

In the experiments, we will evaluate the impact of different choices of skip-connections and initialization methods to the performance of deep Lipschitz networks.

7.6 Experimental Results

In the experiments, we achieve the following goals. **(1)** We demonstrate in Section 7.6.1 that our separable complete factorization (SC-Fac) achieves precise orthogonality (up to machine-precision), resulting in more accurate orthogonal designs than previous ones [182, 183, 191]. **(2)** Despite the differences in preciseness, we show in Section 7.6.2 that different realizations of paraunitary systems only have a minor impact on the adversarial robustness of Lipschitz networks. **(3)** Due to the versatility of our convolutional layers and architectures, in Section 7.6.3, we explore the best strategy to scale Lipschitz networks to wider/deeper architectures. **(4)** In Section 7.7, we further demonstrate in a successful application of orthogonal convolutions in residual flows [188]. Training details are provided in Section 7.7.2.

7.6.1 Exact Orthogonality

We evaluate the orthogonality of our SC-Fac layer verse previous approaches, including CayleyConv [183], BCOP [182], SVCM [191], RKO [190], OSSN [176]. Our experiments are based on a convolutional layer with 64 input channels and 16×16 input size. We orthogonalize the layer using each approach, and evaluate it with Gaussian inputs. For our SC-Fac layer, We initialize all orthogonal matrices uniformly, while we use built-in initialization for others. We evaluate the difference between 1 and the ratio of the output norm to the input norm — a layer is exactly orthogonal if the number is close to 0.

Table 7.2: **(Left) Orthogonality evaluation of different designs for standard convolution.** The number $\|\mathsf{Conv}(\mathbf{x})\|/\|\mathbf{x}\|-1$ indicates the difference between the output and input norms of a layer. A layer is more precisely orthogonal if the number is closer to 0. As shown, our SC-Fac achieves orders of magnitude more orthogonal on standard convolution. **(Right) Orthogonality evaluation of our SC-Fac design for various convolutions.** The numbers $\|\mathsf{Conv}(\mathbf{x})\|/\|\mathbf{x}\|-1$ displayed are in the magnitude of 10^{-8}. As shown, our SC-Fac layers achieve machine epsilon orthogonality on variants of convolution.

Conv.	$\|\mathsf{Conv}(\mathbf{x})\|/\|\mathbf{x}\|-1$
SC-Fac	$(\mathbf{+3.14 \pm 7.38}) \times \mathbf{10^{-8}}$
CayleyConv	$(+2.88 \pm 1.90) \times 10^{-4}$
BCOP	$(+2.59 \pm 6.14) \times 10^{-3}$
SVCM	$-0.429 \pm 3.31 \times 10^{-3}$
RKO	$-0.666 \pm 1.74 \times 10^{-3}$
OSSN	$-0.422 \pm 3.44 \times 10^{-3}$

Type \ Groups		1	4	16
R-Dilated	1	$+3.14 \pm 7.38$	$+1.94 \pm 6.87$	$+1.44 \pm 6.29$
	2	$+3.65 \pm 7.87$	$+1.41 \pm 6.77$	$+1.02 \pm 6.46$
	4	$+3.18 \pm 7.46$	$+1.79 \pm 6.87$	$+1.54 \pm 6.21$
$\downarrow R$-Strided	2	-4.69 ± 5.10	$+4.38 \pm 6.30$	$+1.79 \pm 5.78$
	4	$+10.39 \pm 5.15$	$+6.35 \pm 6.04$	$+3.05 \pm 5.79$
$\uparrow R$-Strided	2	$+3.67 \pm 7.96$	$+1.38 \pm 6.70$	$+1.43 \pm 6.23$
	4	$+3.86 \pm 7.09$	$+1.12 \pm 6.81$	N/A

(1) Standard convolution. We show in Table 7.2 (Left) that our SC-Fac is orders of magnitude more precise than all other approaches. The SC-Fac layer is in fact exactly orthogonal up to machine epsilon, which is $2^{-24} \approx 5.96 \times 10^{-8}$ for 32-bits floats. While RKO and OSSN are known not to be orthogonal, we surprisingly find that SVCM is far from orthogonal due to its masking step. **(2) Convolutions variants.** In Section 7.4, we show that various orthogonal convolutions can be constructed using paraunitary systems. We verify our theory in Table 7.2 (Right): SC-Fac layers are exactly orthogonal (up to machine precision) for all types.

7.6.2 Adversarial Robustness

In this subsection, we evaluate the adversarial robustness of Lipschitz networks. Following the setup in [183], we adopt KW-Large, ResNet9, WideResNet10-10 as the backbone architectures, and evaluate their robust accuracy on CIFAR-10 with different designs of orthogonal convolutions. We extensively perform a hyper-parameter search and choose the best hyper-parameters for each approach based on the robust accuracy. The details of the hyper-parameter search is in Section 7.7.2. We run each model with 5 different seeds and report the best accuracy.

Table 7.3: **(Left) Certified robustness for convolutional networks** (without input normalization). We use KW-Large introduced by [222]. The results for RKO, OSSN, and SVCM are produced by [183]. **(Right) Practical robustness for residual networks** (with input normalization). For 22 layers, the width of SC-Fac is multiplied with 10, CayleyConv with 6, and BCOP and RKO with 8. We are unable to scale CayleyConv, BCOP, and RKO due to memory constraint. As shown, deeper architectures perform better than shallow ones for all approaches, and our SC-Fac has a clear advantage.

		KW-Large					
ϵ	Test Acc.	SC-Fac	Cayley	BCOP	RKO	OSSN	SVCM
0	Clean	74.69	75.57	74.81	74.47	71.69	72.43
$\frac{36}{255}$	Certified	58.68	59.03	58.83	57.50	55.71	52.11
	PGD	67.72	67.78	67.47	68.32	65.13	66.43

		ResNet9				WideResNet10-10				WideResNet22-max			
ϵ	Test Acc.	SC-Fac	Cayley	BCOP	RKO	SC-Fac	Cayley	BCOP	RKO	SC-Fac	Cayley	BCOP	RKO
0	Clean	82.19	**84.26**	83.20	84.07	84.09	82.99	84.29	**84.51**	**87.82**	85.85	84.50	84.55
$\frac{36}{255}$	PGD	71.21	73.47	73.05	**75.03**	74.29	76.02	74.60	**77.14**	**76.46**	74.81	75.00	76.41

(1) Certified robustness. Following [182], we use the raw images (without normalization) for network input to achieve the best certified accuracy. As shown in Table 7.3 (Top), different realizations of paraunitary systems, SC-Fac, CayleyConv and BCOP have comparable performance — CayleyConv is $< 1\%$ better in clean accuracy, but the difference in robust accuracy are negligible.

(2) Practical robustness. [183] shows that the certified accuracy is too conservative, and it is possible to increase the practical robustness (against PGD attacks) with a standard input normalization. Notice that the normalization increases the Lipschitz bound, thus lower the certified accuracy. Our experiments in Table 7.3 (Bottom) are based on ResNet9, WideResNet10-10 [183] and a deeper WideResNet22. For the shallow architectures (ResNet9, WideResNet10-10), our SC-Fac, CayleyConv, and BCOP again achieve comparable performance — CayleyConv is slightly ahead in robust accuracy. **For the**

deeper architecture, our SC-Fac has a clear advantage in both clean and robustness accuracy, and the clean accuracy to only 5% lower than a traditional ResNet 32 trained with batch normalization. Surprisingly, we find that RKO also performs well in robust accuracy while not exactly orthogonal. In summary, our experiments show that various paraunitary realizations provide different impacts on certified and practical robustness. While exact orthogonality provides tight Lipschitz bound, there is a trade-off between the exact orthogonality and the practical robustness (especially with the shallow architectures).

7.6.3 Scaling-up Deep Orthogonal Networks with Lipschitz Bounds

All previous Lipschitz networks [182, 183] only consider shallow architectures (≤ 10 layers). In this subsection, we investigate various factors to scale Lipschitz networks to deeper architectures: skip-connection, depth/width, receptive field, and down-sampling.

(1) Types of skip-connections. Conventional wisdom suggests that skip-connections mainly address gradient vanishing/exploding problems; thus, they are not needed for orthogonal networks. To understand their role, we perform an experiment that trains deep Lipschitz networks without skip-connection and with additive/concatenative skip-connections (see Section 7.5). As shown in Table 7.4 (left), the network with additive skip-connection substantially outperforms the other two, and the one without skip-connections performs the worst. Thus, we empirically show that additive skip-connection is crucial in deep Lipschitz networks.

(2) Depth and width. Exact orthogonality is criticized for harming the expressive power of neural networks. We show that the loss of expressive power can be compensated by increasing the network depth/width. In Table 7.3 (Bottom) and Table 7.6 (Section 7.7.2), we observe that deeper/wider architectures increase the clean/robust accuracy.

(3) Initialization methods. We try different initialization methods, including identical, permutation, uniform, and torus [199, 221]. We find that identical initialization works the best for deep Lipschitz networks (> 10 layers), while all methods perform similarly in shallow networks as shown in Table 7.5 (Section 7.7.2).

(4) Receptive field and down-sampling. Previous works [182, 183] use larger kernel size and no stride for Lipschitz networks. In Table 7.4 (Right), we perform a study on the effects of kernel/dilation size and down-sampling types for the orthogonal convolutions. We find that an average pooling as down-sampling consistently outperforms strided convolutions. Furthermore, a larger kernel size helps to boost the performance.

Table 7.4: **(Left) Comparisons of various skip connection types** on WideResNet22-10 (kernel size equals 5). **(Right) Comparisons of various receptive field and down-sampling types** on WideResNet10-10. The symbols ✓, ✗ indicate whether average pooling or strided convolution is used for down-sampling. For "slim" in strided convolution, we set kernel_size = stride; and for for "wide", kernel_size = stride * kernel_size' (where kernel_ size' is the kernel size for the main branch.

Skip type	Test Acc.	
	Clean	PGD
ConvNet (w/o skip)	69.59	59.22
ShuffleNet (concat)	75.21	66.00
ResNet (add)	**87.82**	**76.46**

Receptive Field		Down-Sampling		Test Acc.	
Kernel	Dilation	Pool	Stride	Clean	PGD
3	1	✗	slim	80.70	68.81
3	1	✗	wide	82.36	70.36
3	1	✓	✗	84.54	71.71
3	2	✓	✗	81.53	70.07
5	1	✓	✗	**84.09**	**74.29**
5	2	✓	✗	81.28	70.58

(5) Run-time and memory comparison. We find that previously proposed orthogonal convolutions such as CayleyConv, BCOP, and RKO require more GPU memory and computation time than SC-Fac. Therefore, we could not to scale them due to memory constraints (for 22 and 32 layers using Tesla V100 32G). In order to scale up Lipschitz networks, economical implementation of orthogonal convolution is crucial. As shown in Figure 7.5, for deep and wide architectures, our SC-Fac is the most computationally and memory efficient method and the only method that scales to a width increase of 10 on WideResNet22. Missing numbers in Figure 7.5 and Table 7.6 (Section 7.7.2) are due to the large memory requirement.

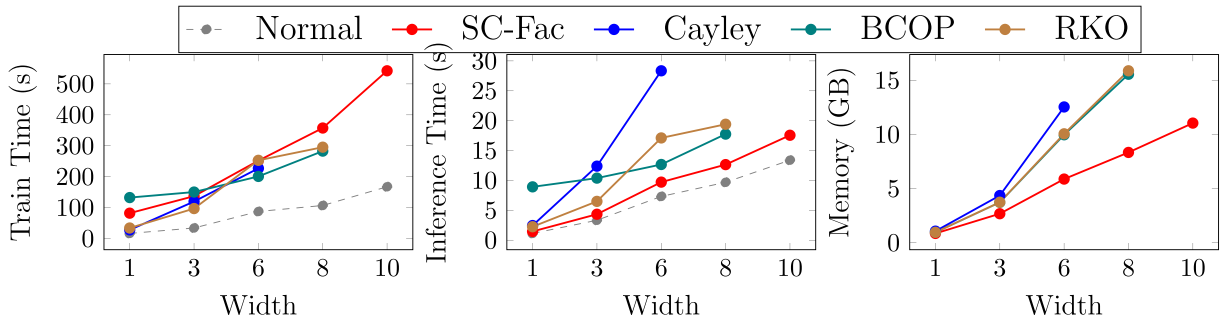


Figure 7.5: **Run-time and memory comparison** using WideResNet22 on Tesla V100 32G. x-axis indicates the width factor (channels = base_channels × factor). Our SC-Fac is the most computationally and memory-efficient for wide architectures and is the only method that scales to width factor to 10 on WideResNet22. We also compare with an ordinary network with regular convolutions and ReLU activations. Note that SC-Fac has the same inference speed as a regular convolution — the overhead is from the GroupSort activations.

In summary, additive skip-connections are still essential for deep orthogonal networks. Due to the orthogonal constraints, it is helpful to increase the depth/width of the network. However, this significantly increases the memory requirement; thus, a cheap implementation (like SC-Fac) is desirable. Finally, we find that a larger kernel size and down-sampling based on average pooling is helpful, unlike standard practices in deep networks.

7.7 Supplementary Materials

7.7.1 Pseudo Code for SC-Fac Algorithm

Algorithm 1: Separable Complete Factorization (SC-Fac)

```
Input: Number of channels $C$, kernel size $K = 2L + 1$, and
       Skew-symmetric matrices $\{A_d^{(\ell)}\}$ with $A_d^{(\ell)} \in \mathbb{R}^{C \times C}, \ell \in [-L, L]$, $d \in \{1, 2\}$.
Output: A paraunitary system $\mathcal{H} \in \mathbb{R}^{C \times C \times K \times K}$.
Initialization: Sample $N_d^{(\ell)}$ from $\{1, \cdots, C\}$ uniformly $\forall \ell \in [-L, L]$, $d \in \{1, 2\}$
/* Iterate for vertical/horizontal dimensions */
for $d = 1$ to $2$ do
    /* 1) Compute orthogonal matrices from skew-symmetric matrices */
    /* Iterate for filter locations */
    for $\ell = -L$ to $L$ do
        if $\ell = 0$ then
            $Q_d \leftarrow \mathsf{matrix\_exp}(A_d^{(0)})$ // use matrix_exp() in GeoTorch [223]
        else
            $U_d^{(\ell)} \leftarrow \mathsf{select}(\mathsf{matrix\_exp}(A_d^{(\ell)}), \mathsf{cols} = N_d^{(\ell)})$ // selects the first cols
                columns of the matrix
        end if
    end for
    /* 2) Compose 1D paraunitary systems from orthogonal matrices */
    $\mathcal{H}_d \leftarrow Q_d$
    for $\ell = 1$ to $L$ do
        $\mathcal{H}_d \leftarrow \mathsf{conv1d}(\mathcal{H}_d, \left[ U_d^{(\ell)} {U_d^{(\ell)}}^\top, \; \boldsymbol{I} - U_d^{(\ell)} {U_d^{(\ell)}}^\top \right])$
        $\mathcal{H}_d \leftarrow \mathsf{conv1d}(\left[ \boldsymbol{I} - U_d^{(-\ell)} {U_d^{(-\ell)}}^\top, \; U_d^{(-\ell)} {U_d^{(-\ell)}}^\top \right], \mathcal{H}_d)$
    end for
end for
/* 3) Compose a 2D paraunitary systems from two 1D paraunitary */
$\mathcal{H} \leftarrow \mathsf{Compose}(\mathcal{H}_1, \mathcal{H}_2)$ // i.e., $\mathcal{H}_{:,:,i,j} = (\mathcal{H}_2)_{:,:,j} (\mathcal{H}_1)_{:,:,i}$ where the 1D
    paraunitary systems $\mathcal{H}_1$ and $\mathcal{H}_2$ are of size $C \times C \times K$
return $\mathcal{H}$
```

We include the pseudo-code for *separable complete factorization* (Section 7.3) in Algorithm 1 and *diverse orthogonal convolutions* (Section 7.4) in Algorithm 2.

The pseudo-code in Algorithm 1 consists of three parts: **(1)** First, we obtain orthog-

Algorithm 2: Construct Diverse Orthogonal Convolutions from Paraunitary Systems

```
Input: Number of base channels $C$, kernel size $K = R(2L+1)$,
       stride $R$, dilation $D$, number of groups $G$
Output: An orthogonal kernel $\mathcal{W} \in \mathbb{R}^{T \times S \times K \times K}$
Set $K' \leftarrow K/R$, number of input channels $S \leftarrow GC/R^2$ and output channels
  $T \leftarrow GC$
for $g = 0$ to $G-1$ do
    /* 1) Construct orthogonal convolutions from paraunitary systems */
    Initialize skew-symmetric matrices $\{\{A_d^{(\ell,g)}\}_{\ell=-L}^{L}\}_{d=1}^{2}$ for the current $g$
    $\mathcal{H}^g \leftarrow$ Algorithm 1: SC-Fac$(C, K', \{\{A_d^{(\ell,g)}\}_{\ell=-L}^{L}\}_{d=1}^{2})$
    $\mathcal{H}^g \leftarrow \mathsf{reshape}(\mathcal{H}^g, (C, C, K', K') \to (C/R^2, C, K, K))$
end for
/* 2) Concatenate orthogonal convolutions from different groups */
$\mathcal{W} \leftarrow \mathsf{concatenate}(\{\mathcal{H}^g\}_{g=0}^{G-1},\ \mathsf{dim} = 0)$
return $\mathcal{W}$ (where the filter for input channel $s$ and output channel $t$ is
  $\mathcal{W}_{t,s,:,:} \in \mathbb{R}^{K \times K}$)
```

onal matrices from skew-symmetric matrices using matrix exponential. We use GeoTorch library [223] for the function matrix_exp in our implementation; **(2)** Subsequently, we construct two 1D paraunitary systems using these orthogonal matrices; and **(3)** Lastly, we compose two 1D paraunitary systems to obtain one 2D paraunitary systems. The pseudo-code in Algorithm 2 consists of two parts: **(4)** First, we reshape each paraunitary system into an orthogonal convolution depending on the stride; and **(5)** second, we concatenate the orthogonal kernels for different groups and return the output. ‘

7.7.2 Setups and Additional Results for Empirical Studies

Network architectures. For fair comparisons, we follow the architectures by [183] for KW-Large, ResNet9, WideResNet10-10 (i.e., shallow networks). We set the group size for GroupSort activations as 2 in all experiments. For networks deeper than 10 layers, we implement their architectures modifying from the Pytorch official implementation of ResNet.

It is crucial to replace the global pooling before fully-connected layers with an average pooling with a window size of 4. For the average pooling, we multiply the output with the window size to maintain its 1-Lipschitzness. Other architectures, including ShuffleNet and plain convolutional network (ConvNet), are further modified from the ResNet, where only the skip-connections are changed or removed. We use the widen factor to indicate the channel number: we set the number of channels at each layer as base channels multiplied by the widen factor. The base channels are $16, 32, 64$ for three groups of residual blocks. More details of the ResNet architecture can be found in the official PyTorch implementation.[2]

Learning strategies. We use the CIFAR-10 dataset for all our experiments. We normalize all input images to $[0, 1]$ followed by standard augmentation, including random cropping and horizontal flipping. We use the Adam optimizer with a maximum learning rate of 10^{-2} coupled with a piece-wise triangular learning rate scheduler. We initialize all our SC-Fac layers as permutation matrices: **(1)** we select the number of columns for each pair $\boldsymbol{U}^{(\ell)}, \boldsymbol{U}^{(-\ell)}$ uniformly from $\{1, \cdots, T\}$ at initialization (the number is fixed during training); **(2)** for $\ell > 0$, we sample the entries in $\boldsymbol{U}^{(\ell)}$ uniformly with respect to the Haar measure; and **(3)** for $\ell < 0$, we set $\boldsymbol{U}^{(-\ell)} = \boldsymbol{Q}\boldsymbol{U}^{(\ell)}$ according to Proposition 7.2.

Multi-class hinge loss. Following previous works on Lipschitz networks [181, 182, 183], we adopt the multi-class hinge loss in training. For each model, we perform a grid search on different margins $\epsilon_0 \in \{1 \times 10^{-3}, 2 \times 10^{-3}, 5 \times 10^{-3}, 1 \times 10^{-2}, 2 \times 10^{-2}, 5 \times 10^{-2}, 0.1, 0.2, 0.5\}$ and report the best performance in terms of robust accuracy. Notice that the margin ϵ_0 controls the trade-off between clean and robust accuracy, as shown in Figure 7.6.

[2] `https://github.com/pytorch/vision/blob/master/torchvision/models/`

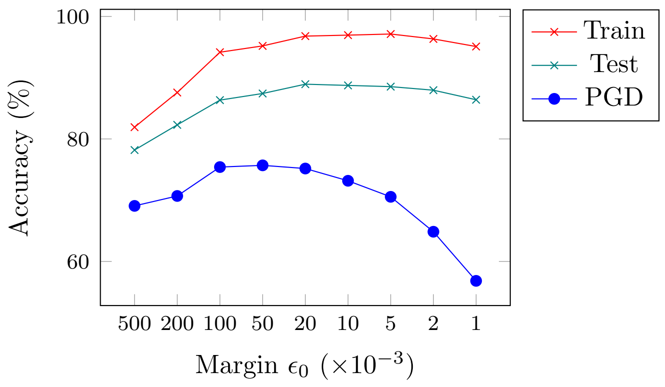


Figure 7.6: **Effect of the Lipschitz margin ϵ_0 for WideResNet22-10**. It shows a trade-off between clean and robust accuracy with different margins for multi-class hinge loss. As shown, the training and test accuracy become higher with larger margin, but the robust accuracy decreases after $\epsilon_0 = 0.1$.

Table 7.5: **Effect of initialization methods** on WideResNet (kernel size 5).

Initialization	WideResNet10-10		WideResNet22-10	
	Clean (%)	PGD (%)	Clean (%)	PGD (%)
uniform	**83.58**	73.20	87.55	75.71
torus	82.40	72.50	**88.12**	75.43
permutation	83.18	73.16	87.82	**76.46**
identical	83.29	**73.49**	87.82	75.49

Initialization methods. In Proposition 7.2, we show how to initialize our orthogonal convolutional layers as orthogonal matrices. In Table 7.5, we perform a study on different initialization methods, including identical, permutation, uniform, and torus [199, 221]. We find that permutation works the best for WideResNet22-10, while all methods are similar in shallower WideResNet10-10. Therefore, we use permutation for all other experiments.

Network depth and width. Exact orthogonality is criticized for harming the expressive power of neural networks, and we find that increasing network depth/width can partially compensate for such loss. In Table 7.6, we perform a study on the impact of network depth/width on the predictive performance. As shown, deeper/wider architectures con-

Table 7.6: **Comparison of different depth and width** on WideResNet (kernel size 5). Some numbers are missing due to the large memory requirement (on Tesla V100 32G). The notation width factor indicates (channels = base channels × factor).

	10 layers									
Width	1	3	6	8	10	1	3	6	8	10
	Clean (%)					PGD with $\epsilon = 36/255$ (%)				
Ours	79.96	84.17	84.96	84.61	84.09	65.92	69.70	72.18	72.51	74.29
Cayley	77.88	82.14	82.56	**85.53**	85.01	66.65	73.06	74.33	75.66	76.13
RKO	81.37	83.55	84.67	85.18	84.62	70.55	74.44	76.41	76.65	**77.02**
	22 layers									
Width	1	3	6	8	10	1	3	6	8	10
	Clean (%)					PGD with $\epsilon = 36/255$ (%)				
Ours	79.90	82.22	87.21	**88.10**	87.82	67.95	70.88	74.30	75.12	**76.46**
Cayley	79.11	84.82	85.85	-	-	69.79	65.61	74.81	-	-
RKO	82.71	84.19	84.33	84.55	-	72.40	74.36	75.66	76.41	-
	34 layers									
Width	1	3	6	8	10	1	3	6	8	10
	Clean (%)					PGD with $\epsilon = 36/255$ (%)				
Ours	81.24	88.17	**88.92**	-	-	69.21	71.85	**75.09**	-	-
Cayley	82.46	84.29	-	-	-	71.27	74.73	-	-	-
RKO	81.51	83.24	83.92	-	-	71.38	73.84	75.03	-	-

sistently improve both the clean and robust accuracy for our implementation. However, the best robust accuracy is achieved by a 22-layer network since we can afford a wide architecture for 34-layer architecture.

7.7.3 Orthogonal Convolutions for Residual Flows

In this subsection, we first review the class of *flow-based generative models* [224, 225]. We focus on *invertible residual network* [194], a flow-based model that relies on Lipschitz residual block, and its extended version *Residual Flow* [188]. We then show how to construct improved Residual Flow using our orthogonal convolutions.

Flow-based models. Given an observable vector $\boldsymbol{x} \in \mathbb{R}^D$ and a latent vector $\boldsymbol{z} \in \mathbb{R}^D$, we define a bijective mapping $f : \mathbb{R}^D \to \mathbb{R}^D$ from the latent vector $\boldsymbol{z}$ to an observation $\boldsymbol{x} = f(\boldsymbol{z})$. We further define the inverse of f as $F = f^{-1}$, with which we represent the likelihood of $\boldsymbol{x}$ by the one of $\boldsymbol{z}$ as:

$$\ln p_X(\boldsymbol{x}) = \ln p_Z(\boldsymbol{z}) + \ln|\det \boldsymbol{J}_F(\boldsymbol{x})|, \tag{7.7}$$

where p_X is the data distribution, p_Z is the base distribution (usually a normal distribution), and $\boldsymbol{J}_F(x)$ is the Jacobian of F at $\boldsymbol{x}$. In practice, the bijective mapping f is composed by a sequence of K bijective mapping such that $f = f_K \circ \cdots \circ f_1$, where each f_k is named as a *flow*. Since the inverse mapping $F = F_1 \circ \cdots F_K$ transforms the data distribution p_X into a normal distribution p_Z, flow-based models are also known as *normalizing flows*. Accordingly, we rewrite Equation (7.7) as:

$$\ln p_X(\boldsymbol{x}) = \ln p_Z(\boldsymbol{z}) + \sum_{k=1}^{K} \ln|\det \boldsymbol{J}_{F_k}(\boldsymbol{x})|, \tag{7.8}$$

In a flow-based model, we require efficient computations of **(a)** each bijective mapping f_k, **(b)** its inverse mapping $F_k = f_k^{-1}$, and **(c)** the corresponding log-determinant $\ln|\det \boldsymbol{J}_F(\cdot)|$.

Invertible residual networks (i-ResNets). [194] proposes a flow-based model based on residual network (ResNet). Note that a block in ResNet is defined as $F(\boldsymbol{x}) = \boldsymbol{x} + g(\boldsymbol{x})$, where g is a convolutional network. In [194], the authors prove that F is a bijective mapping if g is 1-Lipschitz, and its inverse mapping can be computed by *fixed-point iterations*:

$$\boldsymbol{x}_{k+1} = \boldsymbol{y} - g(\boldsymbol{x}_k), \tag{7.9}$$

where $\boldsymbol{y} = g(\boldsymbol{x})$ is the output of F and the initialization of the iterative algorithm is $\boldsymbol{x}_0 := \boldsymbol{y}$. From the Banach fixed-point theorem, we have

$$\|\boldsymbol{x} - \boldsymbol{x}_k\|_2 = \frac{\mathrm{Lip}(g)^k}{1 - \mathrm{Lip}(g)} \|\boldsymbol{x}_1 - \boldsymbol{x}_0\|, \tag{7.10}$$

i.e., the convergence rate is exponential in the number of iterations and smaller Lipschitz constant yields faster convergence. Moreover, the log-determinant can be computed as:

$$\begin{aligned}
\ln p_X(\boldsymbol{x}) &= \ln p_Z(\boldsymbol{z}) + \mathsf{tr}\left(\ln\left(\boldsymbol{I} + \boldsymbol{J}_g(\boldsymbol{x})\right)\right) &(7.11)\\
&= \ln p_Z(\boldsymbol{z}) + \mathsf{tr}\left(\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \left[\boldsymbol{J}_g(\boldsymbol{x})\right]^k\right), &(7.12)
\end{aligned}$$

where the infinite sum is approximated by truncation and the trace is efficiently estimated using the *Hutchinson trace estimator* $\mathsf{tr}(\boldsymbol{A}) = \mathbb{E}_{\boldsymbol{v}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{I})}[\boldsymbol{v}^\top \boldsymbol{A}\boldsymbol{v}]$.

To constrain the Lipschitz constant, i-ResNet uses *spectral normalization* on each linear layer. Moreover, to improve optimization stability, i-ResNet changes the activation function from ReLU to ELU, ensuring nonlinear activations have continuous derivatives.

As summarized in [194], there are two remaining problems in this model: **(1)** The estimator of the log-determinant is biased and inefficient; **(2)** Designing and learning networks with a Lipschitz constraint are challenging — one needs to constrain each linear layer

Table 7.7: **Comparisons of various flow-based models on the MNIST dataset.** We report the performance in bits per dimension (bpm), where a smaller number indicates a better performance.

Model	MNIST
Glow [218]	1.05
FFJORD [226]	0.99
i-ResNet [194]	1.05
Residual Flow [188]	0.97
SC-Fac Residual Flow (Ours)	0.896

in the block instead of being able to control the Lipschitz constant of a block.

Residual Flow. [188] addresses **problem {(1)** by proposing an unbiased *Russian roulette estimator* for Equation (7.12):

$$\mathsf{tr}\left(\ln\left(\boldsymbol{I}+\boldsymbol{J}_g(\boldsymbol{x})\right)\right)=\mathbb{E}_{n,\boldsymbol{v}}\left[\sum_{k=1}^{n}\frac{(-1)^k}{k}\frac{\boldsymbol{v}\left[\boldsymbol{J}g(\boldsymbol{x})^k\right]\boldsymbol{v}}{\mathbb{P}(N\geq k)}\right], \tag{7.13}$$

where $n\sim p(N)$ and $\boldsymbol{v}\sim\mathcal{N}(\boldsymbol{0},\boldsymbol{I})$. Residual Flow further changes the activation from ELU to LipSwish. The LipSwich activation avoids derivative saturation, which occurs when the second derivative is zero in a large region. However, **problem (2)** remains unresolved.

Residual flows with orthogonal convolutions. We propose to address **problem (2)** by replacing spectral normalization by orthogonal convolution (SC-Fac). Note that orthogonal convolutions directly control the Lipschitz constant of a ResNet block. We keep all other components unchanged — in particular, we use LipSwish instead of GroupSort, as GroupSort suffers from derivative saturation. We experiment our model on MNIST dataset, and Table 7.7 shows that our model substantially improve the performance over the original Residual Flow. We display some images generated by our model in Figure 7.7.

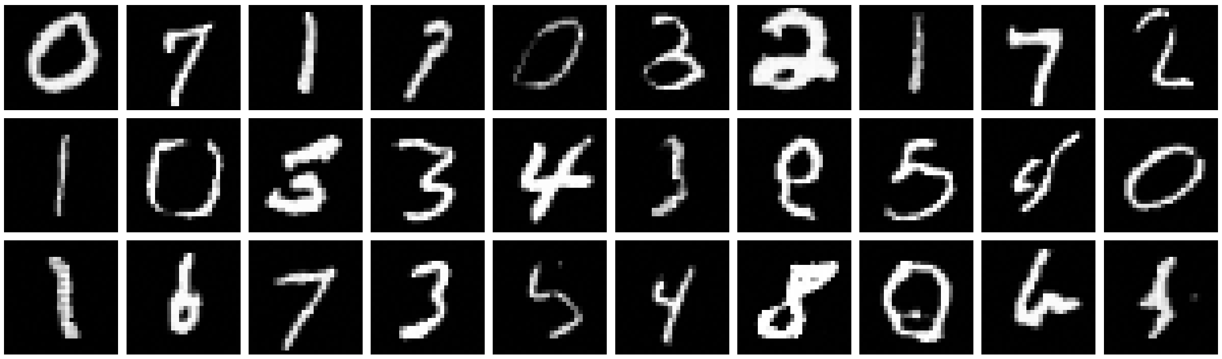


Figure 7.7: Random samples from SC-Fac Residual Flow trained on MNIST.

7.8 Conclusion

In this chapter, we present a paraunitary framework for orthogonal convolutions. Specifically, we establish the equivalence between orthogonal convolutions in the spatial domain and paraunitary systems in the spectral domain. Therefore, any design for orthogonal convolutions is implicitly constructing paraunitary systems. We further show that the orthogonality for variants of convolution (strided, dilated, and group convolutions) is also fully characterized by paraunitary systems. In summary, paraunitary systems are all we need to ensure orthogonality for diverse types of convolutions.

Based on the complete factorization of 1D paraunitary systems, we develop the first exact and complete design of separable orthogonal 2D-convolutions. Our versatile design allows us to study the design principles for orthogonal convolutional networks. Consequently, we scale orthogonal networks to deeper architectures, substantially outperforming their shallower counterparts. In our experiments, we observe that exact orthogonality plays a crucial role in learning deep Lipschitz networks. In the future, we plan to investigate other use cases that exact orthogonality is essential.

Chapter 8: Conclusion

This dissertation investigates spectral methods to design neural networks with desirable properties (or without undesirable properties). We divide the dissertation into three modules.

In the first module, we apply tensor representations to interpret and design multilinear operations. In Chapter 2, we introduced a framework to design compact convolutional layers, which maintain high expressive power but with significantly fewer parameters. Then, in Chapter 3, we develop an automatic library that can effectively denote tensor representations, with which we can efficiently learn multilinear operations. And lastly, in Chapter 4, we propose the first higher-order recurrent layer for spatio-temporal learning, which achieves the state-of-the-art in long-term prediction tasks. There are two potential directions to further advance multilinear operations in neural networks. It will be interesting to understand and design attention modules from the perspective of tensor representations, as attention mechanisms are gaining popularity over convolutions and recurrences in recent years. We also plan to develop fused algorithms to further accelerate tensor representation evaluation, which avoids the serial computation of binary operations.

In the second module (Chapter 5), we develop an efficient scheme to model uncertainties using Bayesian quantized networks. Using this scheme, we convert the intractable computation in Bayesian neural networks into tractable tensor operations and develop fast algorithms for these operations. On the other hand, learning a quantized neural network is an integer programming problem since the weights only take discrete values. However, the problem becomes continuous since the probability vector is continuous. Therefore, our scheme also provides an alternative way to learn compact quantized networks. In the future, it will be interesting to investigate the statistical properties of our design and other approaches for learning Bayesian neural networks. Since a quantitive analysis of the trade-off between bias and variance is lacking, different methods are comparable in their final performance. Therefore, it is desirable to characterize their properties such that we know their applicability from the context and further improve these approaches.

In the last module, we show that a convolutional layer is equivalent to a *multi-input multi-output* (MIMO) filter bank, which allows us to specify the properties of convolutional layers using filter bank theory. In Chapter 6, we propose ARMA layers based on *infinite impulse response* (IIR) filter banks such that we can economically expand the receptive field of a convolutional layer. And in Chapter 7, we propose orthogonal convolutions based on *paraunitary* filter banks such that we can easily ensure their exact orthogonality. In this module, we have mainly focused on the properties of convolutional layers per se. As a future direction, it will be interesting to investigate the interplay between the properties of convolutional layers and network architectures. For instance, it will be interesting to understand the interactions between ARMA layers and self-attention modules in expanding receptive fields. As another example, we plan to investigate how orthogonal convolutions collaborate with other components in constructing invertible neural networks.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[3] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[6] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[9] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017.

[10] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[11] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

[12] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[13] Aharon Azulay and Yair Weiss. Why do deep convolutional networks generalize so poorly to small image transformations? *Journal of Machine Learning Research*, 20(184):1–25, 2019.

[14] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *British Machine Vision Conference 2016*. British Machine Vision Association, 2016.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[16] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[17] Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved group convolutions. In *Proceedings of the IEEE international conference on computer vision*, pages 4373–4382, 2017.

[18] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[19] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. *Advances in neural information processing systems*, 28, 2015.

[20] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[21] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[23] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[24] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[25] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, and Danilo P Mandic. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9(4-5):249–429, 2016.

[26] Andrzej Cichocki, Anh-Huy Phan, Qibin Zhao, Namgil Lee, Ivan Oseledets, Masashi Sugiyama, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning*, 9(6):431–673, 2017.

[27] Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. In *International Conference on Machine Learning*, pages 955–963, 2016.

[28] Valentin Khrulkov, Alexander Novikov, and Ivan Oseledets. Expressive power of recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[29] Kohei Hayashi, Taiki Yamaguchi, Yohei Sugawara, and Shin-ichi Maeda. Exploring unexplored tensor network decompositions for convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[30] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[31] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *Advances in neural information processing systems*, 27, 2014.

[32] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1984–1992, 2015.

[33] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[34] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike. *arXiv preprint arXiv:1611.03214*, 2016.

[35] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[36] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *International Conference on Machine Learning*, pages 3891–3900. PMLR, 2017.

[37] Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. Long-term forecasting using tensor-train RNNs. *arXiv preprint arXiv:1711.00073*, 2017.

[38] Jiahao Su, Wonmin Byeon, Jean Kossaifi, Furong Huang, Jan Kautz, and Anima Anandkumar. Convolutional tensor-train LSTM for spatio-temporal learning. *Advances in Neural Information Processing Systems*, 33:13714–13726, 2020.

[39] Dingheng Wang, Guangshe Zhao, Guoqi Li, Lei Deng, and Yang Wu. Compressing 3d-CNNs based on tensor train decomposition. *Neural Networks*, 131:215–230, 2020.

[40] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[41] Jinmian Ye, Guangxi Li, Di Chen, Haiqin Yang, Shandian Zhe, and Zenglin Xu. Block-term tensor neural networks. *Neural Networks*, 2020.

[42] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[43] Saurabh Goyal, Anamitra Roy Choudhury, and Vivek Sharma. Compression of deep neural networks by combining pruning and low rank decomposition. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 952–958. IEEE, 2019.

[44] Donghyun Lee, Dingheng Wang, Yukuan Yang, Lei Deng, Guangshe Zhao, and Guoqi Li. QTT-net: Quantized tensor train neural networks for 3d object and video recognition. *Neural Networks*, 2021.

[45] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *Advances in neural information processing systems*, 27, 2014.

[46] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[47] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[48] Chi-Chung Lam, P Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reductions for parallel execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

[49] Robert NC Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.

[50] Jiahao Su, Jingling Li, Xiaoyu Liu, Teresa Ranadive, Christopher Coley, Tai-Ching Tuan, and Furong Huang. Compact neural architecture designs by tensor representations. *Frontiers in Artificial Intelligence*, 5, 2022.

[51] Pierre Comon, Xavier Luciani, and André LF De Almeida. Tensor decompositions, alternating least squares and other tales. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 23(7-8):393–405, 2009.

[52] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[53] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. Wide compression: Tensor ring nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9329–9338, 2018.

[54] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, 2020.

[55] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[56] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[57] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[58] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 26–32, 2017.

[59] Jean Kossaifi, Zachary C Lipton, Arinbjorn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *Journal of Machine Learning Research*, 21(123):1–21, 2020.

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

[61] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[63] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019.

[64] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[65] Alex Rogozhnikov and Cristian Garcia. Einops. https://github.com/arogozhnikov/einops, 2020.

[66] Alexander Reustle, Tahseen Rabbani, and Furong Huang. Fast GPU convolution for CP-decomposed tensorial neural networks. In *Proceedings of SAI Intelligent Systems Conference*, pages 468–487. Springer, 2020.

[67] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[68] Po-An Wang and Chi-Jen Lu. Tensor decomposition via simultaneous power iteration. In *International Conference on Machine Learning*, pages 3665–3673, 2017.

[69] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. *Advances in neural information processing systems*, 27, 2014.

[70] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[71] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[72] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *Proc. Interspeech 2020*, pages 5036–5040, 2020.

[73] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[74] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, University of Toronto*, 2009.

[75] Chelsea Finn and Sergey Levine. Deep visual foresight for planning robot motion. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2786–2793. IEEE, 2017.

[76] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhudinov. Unsupervised learning of video representations using LSTMs. In *International conference on machine learning*, pages 843–852. PMLR, 2015.

[77] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social LSTM: Human trajectory prediction in crowded spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 961–971, 2016.

[78] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.

[79] Yunbo Wang, Mingsheng Long, Jianmin Wang, Zhifeng Gao, and Philip S Yu. Predrnn: Recurrent neural networks for predictive learning using spatiotemporal LSTMs. *Advances in neural information processing systems*, 30, 2017.

[80] Yunbo Wang, Zhifeng Gao, Mingsheng Long, Jianmin Wang, and S Yu Philip. Predrnn++: Towards a resolution of the deep-in-time dilemma in spatiotemporal predictive learning. In *International Conference on Machine Learning*, pages 5123–5132. PMLR, 2018.

[81] Yunbo Wang, Lu Jiang, Ming-Hsuan Yang, Li-Jia Li, Mingsheng Long, and Li Fei-Fei. Eidetic 3d LSTM: A model for video prediction and beyond. In *International conference on learning representations*, 2018.

[82] Rohollah Soltani and Hui Jiang. Higher order recurrent neural networks. *arXiv preprint arXiv:1605.00064*, 2016.

[83] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *The Journal of Machine Learning Research*, 15(1):2773–2832, 2014.

[84] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW)*, pages 1–5. IEEE, 2015.

[85] Alessandro Achille and Stefano Soatto. Emergence of invariance and disentanglement in deep representations. *The Journal of Machine Learning Research*, 19(1):1947–1980, 2018.

[86] Jean Kossaifi, Adrian Bulat, Georgios Tzimiropoulos, and Maja Pantic. T-net: Parametrizing fully convolutional nets with a single high-order tensor. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7822–7831, 2019.

[87] Yongxin Yang and Timothy Hospedales. Deep multi-task representation learning: A tensor factorisation approach. *arXiv preprint arXiv:1605.06391*, 2016.

[88] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 4451–4458. IEEE, 2017.

[89] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. *Advances in Neural Information Processing Systems*, 32, 2019.

[90] William Lotter, Gabriel Kreiman, and David Cox. Deep predictive coding networks for video prediction and unsupervised learning. *arXiv preprint arXiv:1605.08104*, 2016.

[91] Wonmin Byeon, Qin Wang, Rupesh Kumar Srivastava, and Petros Koumoutsakos. Contextvp: Fully context-aware video prediction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 753–769, 2018.

[92] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. *Advances in neural information processing systems*, 29, 2016.

[93] Ruben Villegas, Jimei Yang, Seunghoon Hong, Xunyu Lin, and Honglak Lee. Decomposing motion and content for natural video sequence prediction. *arXiv preprint arXiv:1706.08033*, 2017.

[94] Emily L Denton et al. Unsupervised learning of disentangled representations from video. *Advances in neural information processing systems*, 30, 2017.

[95] Jun-Ting Hsieh, Bingbin Liu, De-An Huang, Li F Fei-Fei, and Juan Carlos Niebles. Learning to decompose and disentangle representations for video prediction. *Advances in neural information processing systems*, 31, 2018.

[96] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[97] Marijn F Stollenga, Wonmin Byeon, Marcus Liwicki, and Juergen Schmidhuber. Parallel multi-dimensional LSTM, with application to fast biomedical volumetric image segmentation. *Advances in neural information processing systems*, 28, 2015.

[98] Christian Schuldt, Ivan Laptev, and Barbara Caputo. Recognizing human actions: a local SVM approach. In *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, volume 3, pages 32–36. IEEE, 2004.

[99] Raghav Goyal, Samira Ebrahimi Kahou, Vincent Michalski, Joanna Materzynska, Susanne Westphal, Heuna Kim, Valentin Haenel, Ingo Fruend, Peter Yianilos, Moritz Mueller-Freitag, et al. The" something something" video database for learning and evaluating visual common sense. In *Proceedings of the IEEE international conference on computer vision*, pages 5842–5850, 2017.

[100] Github repo. `https://github.com/NVIDIA/apex`, 2018. [Online; accessed 20-October-2020].

[101] Github repo. `https://github.com/Yunbo426/predrnn-pp`, 2019. [Online; accessed 20-October-2020].

[102] Github repo. `https://github.com/google/e3d_lstm`, 2019. [Online; accessed 20-October-2020].

[103] Nal Kalchbrenner, Aäron Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Video pixel networks. In *International Conference on Machine Learning*, pages 1771–1779. PMLR, 2017.

[104] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

[105] Dirk Weissenborn, Oscar Täckström, and Jakob Uszkoreit. Scaling autoregressive video models. In *International Conference on Learning Representations*, 2019.

[106] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[107] Jacob Menick and Nal Kalchbrenner. Generating high fidelity images with subscale pixel networks and multidimensional upscaling. In *International Conference on Learning Representations*, 2018.

[108] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[109] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems*, 28, 2015.

[110] Zhou Wang, Alan C Bovik, Hamid R Sheikh, Eero P Simoncelli, et al. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[111] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–595, 2018.

[112] Github repo. `https://github.com/jthsieh/DDPAE-video-prediction/blob/master/data/moving_mnist.py`, 2018. [Online; accessed 20-October-2020].

[113] Hao Wang and Dit-Yan Yeung. Towards bayesian deep learning: A framework and some existing methods. *IEEE Transactions on Knowledge and Data Engineering*, 28(12):3395–3408, 2016.

[114] Yarin Gal. *Uncertainty in deep learning.* PhD thesis, University of Cambridge, 2016.

[115] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

[116] Kumar Shridhar, Felix Laumann, and Marcus Liwicki. A comprehensive guide to bayesian convolutional neural network with variational inference. *arXiv preprint arXiv:1901.02731*, 2019.

[117] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. *Advances in neural information processing systems*, 27, 2014.

[118] José Miguel Hernández-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869, 2015.

[119] Soumya Ghosh, Francesco Maria Delle Fave, and Jonathan Yedidia. Assumed density filtering methods for learning bayesian neural networks. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[120] Jiahao Su, Milan Cvitkovic, and Furong Huang. Sampling-free learning of bayesian quantized neural networks. In *International Conference on Learning Representations*, 2019.

[121] Elina Robeva and Anna Seigal. Duality of graphical models and tensor networks. *Information and Inference: A Journal of the IMA*, 2017.

[122] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in neural information processing systems*, 28, 2015.

[123] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[124] David P Williamson and David B Shmoys. *The design of approximation algorithms.* Cambridge university press, 2011.

[125] Martin J Wainwright, Michael I Jordan, et al. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305, 2008.

[126] Alex Graves. Practical variational inference for neural networks. *Advances in neural information processing systems*, 24, 2011.

[127] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[128] Hao Wang, Xingjian Shi, and Dit-Yan Yeung. Natural-parameter networks: A class of probabilistic neural networks. *Advances in neural information processing systems*, 29, 2016.

[129] Alexander Shekhovtsov and Boris Flach. Feed-forward propagation in probabilistic neural networks with categorical and max layers. In *International conference on learning representations*, 2018.

[130] Jochen Gast and Stefan Roth. Lightweight probabilistic deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3369–3378, 2018.

[131] Thomas Peter Minka. *A family of algorithms for approximate Bayesian inference.* PhD thesis, Massachusetts Institute of Technology, 2001.

[132] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[133] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

[134] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[135] Anqi Wu, Sebastian Nowozin, Edward Meeds, Richard E Turner, José Miguel Hernández-Lobato, and Alexander L Gaunt. Deterministic variational inference for robust bayesian neural networks. In *International Conference on Learning Representations*, 2018.

[136] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[137] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[138] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[139] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[140] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. *Advances in neural information processing systems*, 28, 2015.

[141] Oran Shayer, Dan Levi, and Ethan Fetaya. Learning discrete weights using the local reparameterization trick. In *International Conference on Learning Representations*, 2018.

[142] Jorn WT Peters and Max Welling. Probabilistic binary neural networks. *arXiv preprint arXiv:1809.03368*, 2018.

[143] Elizabeth Newman, Lior Horesh, Haim Avron, and Misha Kilmer. Stable tensor neural networks for rapid deep learning. *arXiv preprint arXiv:1811.06569*, 2018.

[144] Wenjie Luo, Yujia Li, Raquel Urtasun, and Richard Zemel. Understanding the effective receptive field in deep convolutional neural networks. In *Advances in neural information processing systems*, pages 4898–4906, 2016.

[145] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[146] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[147] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. Dilated residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 472–480, 2017.

[148] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7794–7803, 2018.

[149] Matthias Holschneider, Richard Kronland-Martinet, Jean Morlet, and Ph Tchamitchian. A real-time algorithm for signal analysis with the help of the wavelet transform. In *Wavelets*, pages 286–297. Springer, 1990.

[150] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[151] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[152] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

[153] Yanghao Li, Yuntao Chen, Naiyan Wang, and Zhaoxiang Zhang. Scale-aware trident networks for object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 6054–6063, 2019.

[154] Zhengyang Wang and Shuiwang Ji. Smoothed dilated convolutions for improved dense prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2486–2495, 2018.

[155] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[156] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.

[157] Yunho Jeon and Junmo Kim. Active convolution: Learning the shape of convolution for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4201–4209, 2017.

[158] Xizhou Zhu, Han Hu, Stephen Lin, and Jifeng Dai. Deformable convnets v2: More deformable, better results. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9308–9316, 2019.

[159] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al. Attention u-net: Learning where to look for the pancreas. *arXiv preprint arXiv:1804.03999*, 2018.

[160] Wonmin Byeon, Thomas M Breuel, Federico Raue, and Marcus Liwicki. Scene labeling with LSTM recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3547–3555, 2015.

[161] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[162] Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *arXiv preprint arXiv:1507.01526*, 2015.

[163] Sifei Liu, Jinshan Pan, and Ming-Hsuan Yang. Learning recursive filters for low-level vision via a hybrid neural network. In *European Conference on Computer Vision*, pages 560–576. Springer, 2016.

[164] Sifei Liu, Shalini De Mello, Jinwei Gu, Guangyu Zhong, Ming-Hsuan Yang, and Jan Kautz. Learning affinity via spatial propagation networks. In *Advances in Neural Information Processing Systems*, pages 1520–1530, 2017.

[165] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

[166] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[167] Jiahao Su, Shiqi Wang, and Furong Huang. ARMA nets: Expanding receptive field for dense prediction. *Advances in Neural Information Processing Systems*, 33:17696–17707, 2020.

[168] Jae S Lim. *Two-dimensional signal and image processing*. Prentice-Hall, Inc., 1990.

[169] Alan V Oppenheim, John R Buck, and Ronald W Schafer. *Discrete-time signal processing*. Prentice-Hall, 2014.

[170] Dimitri P Bertsekas and Athena Scientific. *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[171] Zhengyang Wang, Na Zou, Dinggang Shen, and Shuiwang Ji. Non-local u-nets for biomedical image segmentation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6315–6322, 2020.

[172] Webpage. `https://challenge2018.isic-archive.com`. [Online; accessed 29-May-2020].

[173] Github repo. `https://github.com/jthsieh/DDPAE-video-prediction/blob/master/data/moving_mnist.py`. [Online; accessed 05-Jan-2020].

[174] Noel CF Codella, David Gutman, M Emre Celebi, Brian Helba, Michael A Marchetti, Stephen W Dusza, Aadi Kalloo, Konstantinos Liopyris, Nabin Mishra, Harald Kittler, et al. Skin lesion analysis toward melanoma detection: A challenge at the 2017 international symposium on biomedical imaging (ISBI), hosted by the international skin imaging collaboration (ISIC). In *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*, pages 168–172. IEEE, 2018.

[175] PP Vaidyanathan. *Multirate systems and filter banks*. Prentice-Hall, Inc., 1993.

[176] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. In *International Conference on Learning Representations*, 2018.

[177] Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. A PAC-bayesian approach to spectrally-normalized margin bounds for neural networks. In *International Conference on Learning Representations*, 2018.

[178] Kui Jia, Shuai Li, Yuxin Wen, Tongliang Liu, and Dacheng Tao. Orthogonal deep neural networks. *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[179] Jiong Zhang, Qi Lei, and Inderjit Dhillon. Stabilizing gradients for deep neural networks via efficient SVD parameterization. In *International Conference on Machine Learning*, pages 5806–5814. PMLR, 2018.

[180] Haozhi Qi, Chong You, Xiaolong Wang, Yi Ma, and Jitendra Malik. Deep isometric learning for visual recognition. In *International Conference on Machine Learning*, pages 7824–7835. PMLR, 2020.

[181] Cem Anil, James Lucas, and Roger Grosse. Sorting out lipschitz function approximation. In *International Conference on Machine Learning*, pages 291–301. PMLR, 2019.

[182] Qiyang Li, Saminul Haque, Cem Anil, James Lucas, Roger B Grosse, and Jörn-Henrik Jacobsen. Preventing gradient attenuation in lipschitz constrained convolutional networks. In *Advances in neural information processing systems*, pages 15390–15402, 2019.

[183] Asher Trockman and J Zico Kolter. Orthogonalizing convolutional layers with the cayley transform. In *International Conference on Learning Representations*, 2021.

[184] Jeffrey Pennington, Samuel Schoenholz, and Surya Ganguli. Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems*, 30, 2017.

[185] Jeffrey Pennington, Samuel Schoenholz, and Surya Ganguli. The emergence of spectral universality in deep networks. In *International Conference on Artificial Intelligence and Statistics*, pages 1924–1932. PMLR, 2018.

[186] Lechao Xiao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel Schoenholz, and Jeffrey Pennington. Dynamical isometry and a mean field theory of CNNs: How to train 10,000-layer vanilla convolutional neural networks. In *International Conference on Machine Learning*, pages 5393–5402. PMLR, 2018.

[187] Jiayun Wang, Yubei Chen, Rudrasis Chakraborty, and Stella X Yu. Orthogonal convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11505–11515, 2020.

[188] Ricky TQ Chen, Jens Behrmann, David K Duvenaud, and Jörn-Henrik Jacobsen. Residual flows for invertible generative modeling. *Advances in Neural Information Processing Systems*, 32, 2019.

[189] Kui Jia, Dacheng Tao, Shenghua Gao, and Xiangmin Xu. Improving training of deep neural networks via singular value bounding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4344–4352, 2017.

[190] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval networks: Improving robustness to adversarial examples. In *International Conference on Machine Learning*, pages 854–863. PMLR, 2017.

[191] Hanie Sedghi, Vineet Gupta, and Philip M Long. The singular values of convolutional layers. In *International Conference on Learning Representations*, 2019.

[192] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2nd Ed.)*. Prentice-Hall, Inc., 1996.

[193] Minmin Chen, Jeffrey Pennington, and Samuel Schoenholz. Dynamical isometry and a mean field theory of RNNs: Gating enables signal propagation in recurrent neural networks. In *International Conference on Machine Learning*, pages 873–882. PMLR, 2018.

[194] Jens Behrmann, Will Grathwohl, Ricky TQ Chen, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2019.

[195] Di Xie, Jiang Xiong, and Shiliang Pu. All you need is beyond a good init: Exploring better solution for training extremely deep convolutional neural networks with orthonormality and modulation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6176–6185, 2017.

[196] Nitin Bansal, Xiaohan Chen, and Zhangyang Wang. Can we gain more from orthogonality regularizations in training deep networks? *Advances in Neural Information Processing Systems*, 31, 2018.

[197] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *International Conference on Machine Learning*, pages 2401–2409. PMLR, 2017.

[198] Victor Dorobantu, Per Andre Stromhaug, and Jess Renteria. Dizzyrnn: Reparameterizing recurrent neural networks for norm-preserving backpropagation. *arXiv preprint arXiv:1612.04035*, 2016.

[199] Kyle Helfrich, Devin Willmott, and Qiang Ye. Orthogonal recurrent neural networks with scaled cayley transform. In *International Conference on Machine Learning*, pages 1969–1978. PMLR, 2018.

[200] Mario Lezcano-Casado and David Martınez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *International Conference on Machine Learning*, pages 3794–3803. PMLR, 2019.

[201] Lei Huang, Li Liu, Fan Zhu, Diwen Wan, Zehuan Yuan, Bo Li, and Ling Shao. Controllable orthogonalization in training DNNs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6429–6438, 2020.

[202] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *International Conference on Machine Learning*, pages 3570–3578, 2017.

[203] Jun Li, Fuxin Li, and Sinisa Todorovic. Efficient riemannian optimization on the stiefel manifold via the cayley transform. In *International Conference on Learning Representations*, 2019.

[204] Sheng Liu, Xiao Li, Yuexiang Zhai, Chong You, Zhihui Zhu, Carlos Fernandez-Granda, and Qing Qu. Convolutional normalization: Improving deep convolutional network robustness and training. *Advances in Neural Information Processing Systems*, 34, 2021.

[205] Sahil Singla and Soheil Feizi. Skew orthogonal convolutions. In *International Conference on Machine Learning*, pages 9756–9766. PMLR, 2021.

[206] Gilbert Strang and Truong Nguyen. *Wavelets and filter banks*. SIAM, 1996.

[207] Yuan-Pei Lin and PP Vaidyanathan. Theory and design of two-dimensional filter banks: A review. *Multidimensional Systems and Signal Processing*, 7(3-4):263–330, 1996.

[208] Shankar Venkataraman and Bernard C Levy. A comparison of design methods for 2d FIR orthogonal perfect reconstruction filter banks. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(8):525–536, 1995.

[209] Jianping Zhou. *Multidimensional multirate systems: Characterization, design, and applications*. University of Illinois at Urbana-Champaign, 2005.

[210] Barry Hurley and Ted Hurley. Paraunitary matrices. *arXiv preprint arXiv:1205.0703*, 2012.

[211] Jiahao Su, Wonmin Byeon, and Furong Huang. Scaling-up diverse orthogonal convolutional networks with a paraunitary framework. *arXiv preprint arXiv:2106.09121*, 2021.

[212] Kehelwala DG Maduranga, Kyle E Helfrich, and Qiang Ye. Complex unitary recurrent neural networks using scaled cayley transform. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4528–4535, 2019.

[213] Nicholas J Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM review*, 51(4):747–764, 2009.

[214] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[215] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[216] Laurent Dinh, David Krueger, and Yoshua Bengio. NICE: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[217] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. *arXiv preprint arXiv:1605.08803*, 2016.

[218] Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

[219] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[220] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[221] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. In *International Conference on Machine Learning*, pages 2034–2042. PMLR, 2016.

[222] Eric Wong, Frank Schmidt, Jan Hendrik Metzen, and J Zico Kolter. Scaling provable adversarial defenses. *Advances in Neural Information Processing Systems*, 31, 2018.

[223] Mario Lezcano Casado. Trivializations for gradient-based optimization on manifolds. *Advances in Neural Information Processing Systems*, 32, 2019.

[224] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021.

[225] Ivan Kobyzev, Simon Prince, and Marcus Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[226] Will Grathwohl, Ricky TQ Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. FFJORD: Free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*, 2018.