國立臺灣大學電機資訊學院資訊工程研究所

博士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Doctoral Dissertation

支持任意雙線性乘積之向量神經元

Vector-neuron-based Learning with Arbitrary Bilinear Products

范哲誠
Zhe-Cheng Fan

指導教授：張智星教授
Advisor: Prof. Jyh-Shing Roger Jang

中華民國 108 年 7 月
July, 2019

國立臺灣大學博士學位論文
口試委員會審定書

支持任意雙線性乘積之向量神經元

Vector-neuron-based Learning with Arbitrary Bilinear Products

本論文係范哲誠君（學號 D02922030）在國立臺灣大學資訊工程學系完成之博士學位論文，於民國 108 年 7 月 12 日承下列考試委員審查通過及口試及格，特此證明

口試委員：張智星

（指導教授）

楊奕軒　　

曹昱　　鄭士康

蘇黎

系　主　任　莊永裕

中文摘要

向量值神經網路學習近年來在深度學習領域中逐漸成為一個新主題。傳統上，多筆多維度的訓練資料在放進神經網路訓練之前，會被串接成一筆高維度的向量。然而，這樣串接資料的方式會造成在不同筆資料中，相同維度的資訊關係沒有學習好，進而導致訓練效果不是最佳的。因此，在本論文中，我們提出一個新的神經網路架構，稱為任意雙線性積神經網路。在此架構中，每個神經元處理向量資訊，成為向量神經元，並且可以將不同的任意雙線性積應用在此架構中。

除此之外，我們將向量神經元配合循環群代數的概念，應用在捲積神經網路，提出深度循群環網路。在此架構中，訓練資料，輸入資料、輸出資料、特徵圖、捲積核都是三維的矩陣。我們透過高光譜影像去躁、歌聲分離以及影像修補的實驗來驗證所提出來的架構。實驗結果顯示，我們提出來的架構與傳統的神經網路比較下，擁有較好的效果，同時也驗證在訓練的過程中，向量之間關連性是有被學習到的。

關鍵字：向量神經學習, 向量神經網路, 雙線性積, 任意雙線積神經網路, 倒傳遞, 循環代數, 深度循環群網路

Abstract

Vector-valued neural learning has emerged as a promising direction in deep learning recently. Traditionally, training data for neural networks (NNs) are formulated as a vector of scalars; however, its performance may not be optimal since associations among adjacent scalars are not modeled. In this dissertation, we propose a new vector neural architecture called the Arbitrary BIlinear Product Neural Network (ABIPNN), which processes information as vectors in each neuron, and the feedforward projections are defined using arbitrary bilinear products. Such bilinear products can include circular convolution, seven-dimensional vector product, skew circular convolution, reversed-time circular convolution, or other new products not seen in previous work. Besides, we also employ vector values into convolutional neural networks (CNNs), called deep cyclic group network (DCGN) which uses the cyclic group algebra for convolutional vector-neuron learning. The input to DCGN is a three-way tensor, where the mode-3 dimension corresponds to the dimensionality of the input data. As a proof-of-concept, we apply our proposed network to multispectral image denoising, singing voice separation and image inpainting. Experimental results show that ABIPNN or DCGN obtains substantial improvements when compared to conventional NNs or CNNs, suggesting that associations are learned during training.

Key words: Vector neural learning, vector neural network, bilinear products, arbitrary bilinear product neural networks, backpropagation, deep cyclic group algebra, deep cyclic group networks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Vector neural network

Vector-valued neurons have received much attention lately in different scientific fields, such as communication systems, biological processing, image processing, and audio signal processing [2–6]. Each training sample of these applications can be represented as a multidimensional vector, which can be processed directly by vector-valued neurons. These aforementioned works have shown that vector-valued neurons have good performance in learning, association, and generalization. In the meantime, an increasing number of datasets [7–9] provide multidimensional data suitable for vector-valued neural learning.

In real-valued neural network (NN) learning with multidimensional data, the input is concatenated from a set of vectors and reformulated as a long vector. A neuron takes only one real value as its input and a network is configured to use as many neurons as the dimension of the long vector. But this configuration may not achieve satisfactory

performance for multidimensional problems since associations within each vector are not learned. Therefore, multidimensional vector neurons have received some attention in the literature and have been proposed to address the associations among different dimensions. A vector-valued neuron accepts and represents information as a vector and elements in each vector are processed together as a single unit.

In convolutional neural network learning, advanced network architectures such as batch normalization [10], highway networks [11], and residual networks [12] have been proposed in recent years to improve the performance or training efficiency of deep learning models. Both highway and residual networks reduce the risk of vanishing gradients via gating-based mechanisms or shortcut paths to regularize information flow. There have also been attempts to make more fundamental changes to convolutional neural networks (CNNs). In particular, vector-valued neural learning in CNNs has received increased attention recently, where the inputs, outputs, weights and biases are all extended from real values to vector values.

1.2 Related work

There are several approaches to extend real-valued neurons to vector-valued ones. Two-dimensional complex-valued NNs [13] have been proven to have orthogonal decision boundaries [14, 15] and the ability to solve exclusive-or (XOR) problem [16] using only a single neuron. Another extension to two dimensions is the hyperbolic neural network [17], in which all parameters are hyperbolic numbers. Decision boundaries of hyperbolic neu-

rons are investigated in [18]. An alternative hyperbolic backpropagation algorithm [19] is developed using Wirtinger calculus. Three-dimensional neurons have been proposed in two ways. The first is based on the vector product [20] [21], in which inputs, outputs, weights and biases are all three-dimensional vectors. The second [22] is similar to the first but the weights are now three-dimensional orthogonal matrices. Four-dimensional hypercomplex-valued neurons [20, 23–25] are proposed by using the quaternion algebra. Eight-dimensional octonion-valued neurons [26] represent a generalization of quaternion NNs. More recently, deep complex networks (DCN) [6] have been introduced using complex convolution and complex batch normalization. They extend each neuron from a real value to a two-dimensional vector representing complex numbers, and the model has been evaluated on image classification and music transcription. Deep quaternion networks (DQN) [1, 27] make each neuron a four-dimensional vector representing the so-called quaternions, and the model has been evaluated on automatic speech recognition and image segmentation. Although both DCN and DQN show the promise of vector-neuron learning, the dimensionality of their vector-valued inputs are restricted to two or four only.

1.3 Current limitation of vector neural networks

However, the dimensionality of a vector-valued neuron should not be constrained to a particular number. For instance, in the task of color image inpainting [28], where small corrupted regions are to be restored, the input and output comprise three color channels. In the task of multispectral image denoising [29], different bands of images are stacked

into a tensor. Nonlinear mapping between the noisy tensor and the clean tensor is then performed. For singing voice separation [30], the data structure of a temporal-frequency matrix input is flattened into a high-dimensional vector in the traditional way, and then reshaped as a matrix so that each neuron represents and receives a vector. For EEG-based emotion recognition [8, 31], the human brain wave is filtered into five main frequency bands. Given the above observations, it seems important that the dimensionality should be an arbitrary N. Thus, a good extension of the real-valued neuron is the N-dimensional real-valued neuron [20, 32, 33], which was proposed to have N-dimensional vector inputs and outputs, but N-dimensional orthogonal matrix weights. Nevertheless, we observe that this formulation does not address the case of multiple neurons. Other architectures have also been advanced to extend real-valued neurons. For example, Clifford algebra has been employed to build vector-valued NNs with dimensionality 2^N [34–37]; however, we note that many datasets do not have power-of-two dimensionalities. Matrix-valued NNs [38] have been proposed in which the inputs, outputs, weights, and biases are $N \times N$ square matrices. But it is not always easy to formulate the inputs and outputs like this. Finally, for multiway classification, the tensor-factorized NN [39] integrates Tucker decomposition with neural learning, though its efficiency in regression problems is yet unproven. A similar concept would be the vector neuron models of associative memory [40], which are vector formalisms of Potts-glass NN [41–43] and parametrical NN (PNN) [44], but this concept does not belong to supervised learning since there are no inputs and corresponding targets for training.

1.4 Contribution of the dissertation

In light of the above observations, we propose a new vector-valued NN architecture consisting of N-dimensional vector-valued neurons, where N is an arbitrary positive integer (as shown in Fig. 1.1). For each neuron, the inputs, outputs, weights, and biases are all N-dimensional vectors. Our key observation is that the products used by the aforementioned vector-valued NNs (such as the vector product [21] and quaternion multiplication [23]) are bilinear products (defined in Section 2). This prompts us to propose a general form of bilinear neurons to model the associations between the vector elements. We call it the Arbitrary BIlinear Product Neural Network (ABIPNN). When N equals one, each neuron represents a scalar and the architecture performs matrix multiplications like a conventional deep neural network (DNN). When N is larger than one, each neuron represents a vector and the architecture uses bilinear products for multiplications. In this way, the proposed architecture not only allows for using vectors of arbitrary dimensionality in vector-valued NNs, but also all the vector-valued products that are bilinear.

On the other hand, we also propose a new CNN architecture named Deep Cyclic Group Network (DCGN), which involves the circular convolution from cyclic group algebra [45]. Our inputs, outputs, kernel maps, and feature maps are three-way tensors [46] with the same mode-3 dimension. The mode-3 dimension corresponds to the prescribed dimensionality N of the input data, where N is an arbitrary positive integer. The proposed architecture is shown in Figure 1.3. Besides, weight initialization and batch normalization for vector-valued neurons are investigated. As a consequence, our contributions are

summarized as follows:

- We proposed the ABIPNN model by using arbitrary bilinear products with N-dimensional vector-valued neurons, where N is an arbitrary positive integer.
- We proposed the DCGN model by using N-dimensional vector-valued neurons with cyclic group algebras, where N is also an arbitrary positive integer.
- We derived backpropagation learning algorithm for both ABIPNN and DCGN models, weight initialization, and batch normalization.
- We demonstrated the potential of our architectures with singing voice separation, image inpainting and denoising experiments by using less number of parameters.

1.5 Dissertation organization

The dissertation is organized as follow. Chapter 2 derives feedforward and backpropagation processes using the proposed bilinear neurons. Chapter 3 derives the backpropagation algorithm using circular convolution. Besides, weight initialization and batch normalization for vector-valued neurons are also introduced. Chapter 4 compares the performance between conventional NNs and proposed architectures in singing voice separation, image inpainting, and multispectral image inpainting. Finally, we conclude the work and discuss future work in Chapter 5.

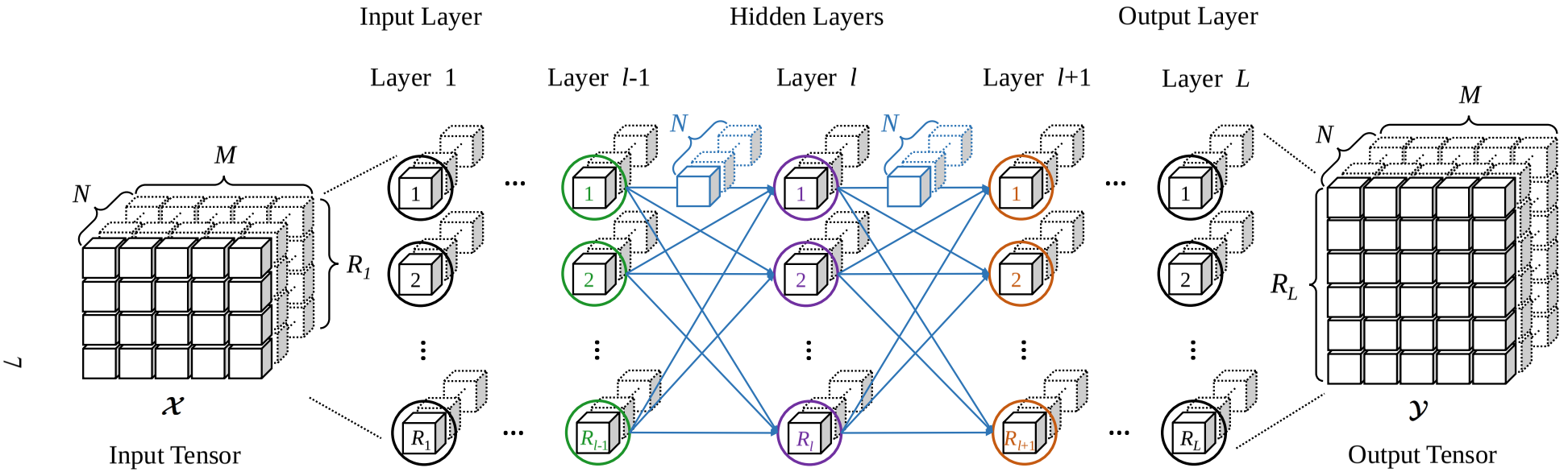


Figure 1.1: Architecture of ABIPNN (best viewed in color), where the inputs, outputs, weights, and biases are all N-dimensional vectors. Cubes in the architecture represent scalars. Blue lines between hidden layers represent weights. M represents the number of training data patterns. R_l represents the dimension at layer l, where $1 \leq l \leq L$. Among them, R_1 and R_L are respectively the dimensions of the input and output layers. From R_2 to R_{L-1} are the dimensions of the hidden layers. $\mathfrak{X}$ and $\mathfrak{Y}$ respectively represent the input and output tensors. When N equals one, each neuron represents a scalar, depicted as a cube with solid lines. This architecture can be viewed as a conventional NN. When $N > 1$, each neuron represents a vector, depicted as the concatenation of cubes connected by the dotted lines.

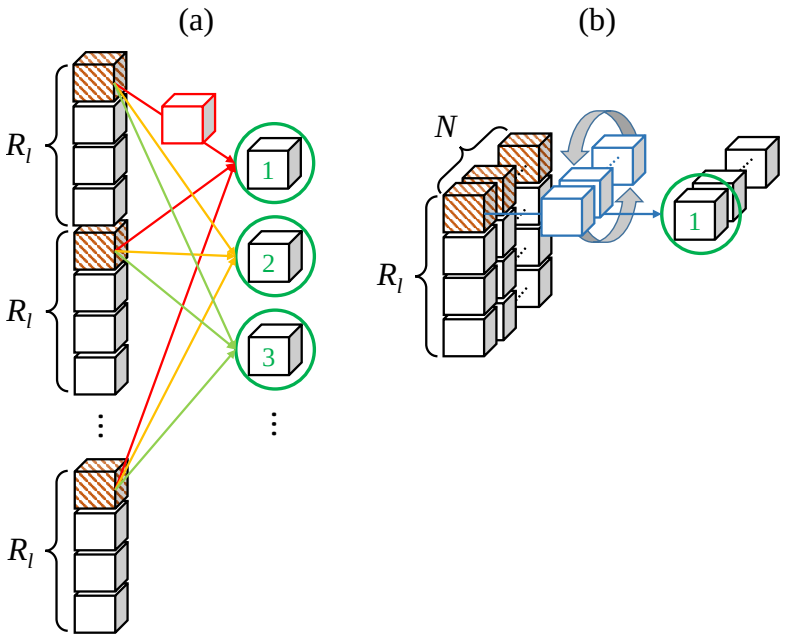


Figure 1.2: Illustration of the operations employed by the (a) conventional NNs and (b) ABIPNNs; best viewed in color. In (a), vectors are concatenated together as a multidimensional input. Each weight stands for a scalar, depicted as a cube. Take the first entry of each vector for example, the associations between the brown cubes are not learned since each hidden neuron receives information by using its group of weights, which are depicted in the same color. Hidden neurons do not share identical weights when performing learning. In (b), vectors are stacked into a $R_l \times 1 \times N$ tensor (or lateral matrix) and brown cubes are concatenated together. Each neuron represents a vector and receives information by using the weight vector, depicted as blue cubes. For example, when we leverage circular convolution as the bilinear product in an ABIPNN, the weight vector can be deemed as a linear kernel mask which captures the associations between brown cubes by rotating itself in the learning process.

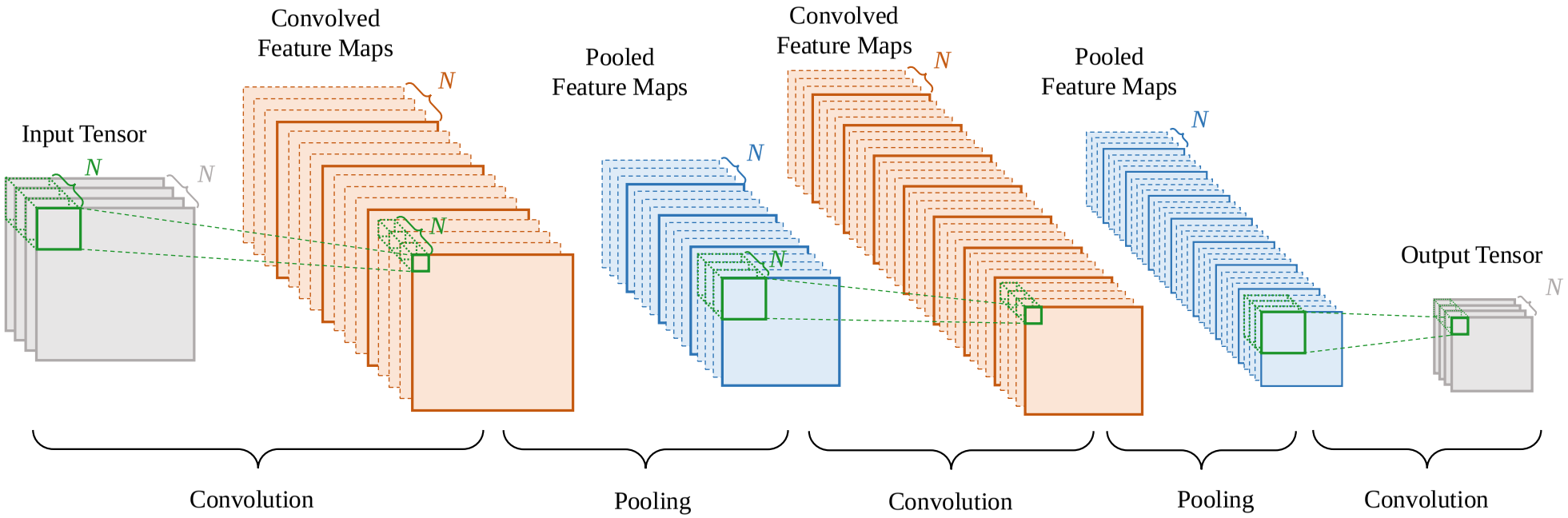


Figure 1.3: Illustration of the proposed DCGN model for regression problems (best viewed in color). Orange and blue squares represent the convolved and pooled feature maps, respectively, and green squares represent the kernel maps. All feature maps across all the layers have the same mode-3 dimension (i.e., N). When $N = 1$, the model reduces to the conventional CNNs, and we will not have those dashed squares. The kernel maps perform convolution with scalar multiplications. When $N > 1$, each feature map and kernel map becomes three-way tensors. It is a regression problem since we have tensor input and tensor output. But, the model can be extended to deal with classification problems, by discarding the last convolutional layer and adding fully-connected layers after DCGN. In such a case, DCGN would perform the function of feature learning.

Chapter 2

Arbitrary Bilinear Product Neural Networks

The idea of ABIPNN is to extend the data type of each neuron from scalars to vectors by replacing multiplications with arbitrary bilinear products. Such products can be used to learn the associations between vector elements in the input (see Fig. 1.2). In the following, tensors are represented by bold uppercase calligraphic letters, matrices by bold uppercase letters, and vectors by bold lowercase letters. Matrix slices of tensors are represented as matrices and vector slices of matrices are represented as vectors. The notational conventions used in this paper are summarized in Table 2.1.

2.1 Generalizing N-D vector neurons with bilinear Products

To begin with, we identify a generalization of all the products used in previous vector-valued NN literature, including vector product [21], quaternion multiplication [23], octonion multiplication [26], and so on. We have been able to confirm that all of these products are bilinear (defined below). Motivated by this important observation, we will extend the vector-valued NN [32] to use any kind of bilinear product between two N-dimensional vectors.

From the inspiration of Chan's work [47], we could assume an N-dimensional space with standard basis $\{\mathbf{e}_1,\ \mathbf{e}_2,\ \ldots,\ \mathbf{e}_N\}$ as column vectors. Let $\mathbf{p} = \sum_{n=1}^{N} p_n \mathbf{e}_n = [p_1,\ p_2,\ \ldots,\ p_N]^T$ and $\mathbf{q} = \sum_{n=1}^{N} q_n \mathbf{e}_n = [q_1,\ q_2,\ \ldots,\ q_N]^T$ be two vectors in this space. A product $\bullet\colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$ is bilinear if and only if $\mathbf{p} \bullet \mathbf{q}$ is linear when we hold one of $\mathbf{p}$ or $\mathbf{q}$ fixed. If $\bullet$ is bilinear, we have:

$$\mathbf{p} \bullet \mathbf{q} = \mathbf{p} \bullet \left(\sum_{n=1}^{N} q_n \mathbf{e}_n \right), \tag{2.1}$$

where we have expanded the second term. Due to bilinearity,

$$\begin{aligned}\mathbf{p} \bullet \mathbf{q} &= \sum_{n=1}^{N} (\mathbf{p} \bullet \mathbf{e}_n)\, q_n \\ &= \left[\mathbf{p} \bullet \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},\ \mathbf{p} \bullet \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix},\ \ldots,\ \mathbf{p} \bullet \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \right] \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{bmatrix} \\ &= [\mathbf{p} \bullet \mathbf{e}_1,\ \mathbf{p} \bullet \mathbf{e}_2,\ \ldots,\ \mathbf{p} \bullet \mathbf{e}_N]\, \mathbf{q}.\end{aligned} \tag{2.2}$$

If we let $[\mathbf{p}]_\bullet = [\mathbf{p} \bullet \mathbf{e}_1,\ \mathbf{p} \bullet \mathbf{e}_2,\ \ldots,\ \mathbf{p} \bullet \mathbf{e}_N]$, then:

$$\mathbf{p} \bullet \mathbf{q} = [\mathbf{p}]_\bullet\, \mathbf{q}. \tag{2.3}$$

Similarly, if we expand the first term, we can write:

$$\begin{aligned}\mathbf{p} \bullet \mathbf{q} &= \left(\sum_{n=1}^{N} p_n \mathbf{e}_n \right) \bullet \mathbf{q} = \sum_{n=1}^{N} p_n\, (\mathbf{e}_n \bullet \mathbf{q}) \\ &= [\mathbf{e}_1 \bullet \mathbf{q},\ \mathbf{e}_2 \bullet \mathbf{q},\ \ldots,\ \mathbf{e}_N \bullet \mathbf{q}]\, \mathbf{p}.\end{aligned} \tag{2.4}$$

Let $[\mathbf{q}]^\dagger_\bullet = [\mathbf{e}_1 \bullet \mathbf{q},\ \mathbf{e}_2 \bullet \mathbf{q},\ \ldots,\ \mathbf{e}_N \bullet \mathbf{q}]$, we also have:

$$\mathbf{p} \bullet \mathbf{q} = [\mathbf{q}]^\dagger_\bullet\, \mathbf{p}. \tag{2.5}$$

Eqs. 2.3 and 2.5 are the two possible matrix representations of the bilinear product $\bullet$. Here we call $[\mathbf{p}]_\bullet$ its matrix representation and $[\mathbf{q}]^\dagger_\bullet$ its transmuted representation (the term *transmuted* is originally used for quaternions [48] but here we extend it for general bilin-

ear products). In what follows, the feedforward and backpropagation processes will be described by using the above notations for bilinear products and their representations. For convenience, in the rest of this paper, if $\mathbf{p}$ or $\mathbf{q}$ are not column vectors, then $\mathbf{p} \bullet \mathbf{q}$ will implicitly reshape them so that the above equations make sense.

Table 2.1: Notational Conventions

$\mathcal{X}$	Input tensor
$\mathbf{X}_m$=$\mathcal{X}_{:m:}$	Lateral slice from input tensor $\mathcal{X}$
$\mathbf{x}$	Vector from input tensor $\mathcal{X}$
$\mathcal{Y}$	Output tensor
$\mathbf{Y}_m$=$\mathcal{Y}_{:m:}$	Lateral slice from output tensor $\mathcal{X}$
$\mathcal{W}^l$	Weight tensor in layer l
$\mathcal{W}^{l+1}$	Weight tensor in layer $l+1$
$\mathbf{w}_{ij}^l = \mathcal{W}_{ij:}^l$	Vector from tensor $\mathcal{W}^l$
$\mathbf{w}_{ki}^{l+1} = \mathcal{W}_{ki:}^{l+1}$	Vector from tensor $\mathcal{W}^{l+1}$
$\mathcal{B}^l$	Bias tensor in layer l
$\mathcal{B}^{l+1}$	Bias tensor in layer $l+1$
$\mathbf{b}_i^l = \mathcal{B}_{im:}^l$	Vector from tensor $\mathcal{B}^l$
$\mathbf{b}_k^{l+1} = \mathcal{B}_{km:}^{l+1}$	Vector from tensor $\mathcal{B}^{l+1}$
$\mathcal{Z}^L$	Input tensor in the layer L
$\mathcal{Z}^l$	Input hidden tensor in layer l
$\mathcal{Z}^{l+1}$	Input hidden tensor in layer $l+1$
$\mathbf{z}_g^L = \mathcal{Z}_{gm:}^L$	Vector from tensor $\mathcal{Z}^L$
$\mathbf{z}_i^l = \mathcal{Z}_{im:}^l$	Vector from tensor $\mathcal{Z}^l$
$\mathbf{z}_k^{l+1} = \mathcal{Z}_{km:}^{l+1}$	Vector from tensor $\mathcal{Z}^{l+1}$
$\mathcal{A}^{l-1}$	Output hidden tensor in layer $l-1$
$\mathcal{A}^l$	Output hidden tensor in layer l
$\mathbf{a}_j^{l-1} = \mathcal{A}_{jm:}^{l-1}$	Vector from tensor $\mathcal{A}^{l-1}$
$\mathbf{a}_i^l = \mathcal{A}_{im:}^l$	Vector from tensor $\mathcal{A}^l$
$\mathcal{D}^{l+1}$	Local gradient tensor in layer $l+1$
$\mathcal{D}^l$	Local gradient tensor in layer l
$\mathbf{d}_j^{l+1} = \mathcal{D}_{im:}^{l+1}$	Vector from tensor $\mathcal{D}^{l+1}$
$\mathbf{d}_i^l = \mathcal{D}_{km:}^l$	Vector from tensor $\mathcal{D}^l$
$\bullet\colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$	Arbitrary bilinear product
$[\cdot]_\bullet$	Matrix representation of $\bullet$ (Eq. 2.3)
$[\cdot]_\bullet^\dagger$	Transmuted representation of $\bullet$ (Eq. 2.5)

2.2 Feedforward process for ABIPNN

As before, we assume the inputs, outputs, weights, and biases are N-dimensional vectors. Suppose we have an L-layer vector-valued NN. For hidden layer l ($1 \leq l \leq L$), the input vector $\mathbf{z}_i^l$ of the hidden neuron i is defined as:

$$\mathbf{z}_i^l = \sum_{j=1}^{R_{l-1}} \mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1} + \mathbf{b}_i^l \,, \tag{2.6}$$

where $\bullet$ denotes an arbitrary bilinear product, $\mathbf{w}_{ij}^l$ stands for the weight vector connecting a neuron j ($1 \leq j \leq R_{l-1}$) in layer $l-1$ to a neuron i ($1 \leq i \leq R_l$) in layer l, and $\mathbf{w}_{ij}^l = \sum_{n=1}^N w_{ijn}^l \mathbf{e}_n = \left[w_{ij1}^l, w_{ij2}^l, \ldots, w_{ijN}^l\right]^T \in \mathbb{R}^N$. The output vector of neuron j in layer $l-1$ is $\mathbf{a}_j^{l-1} = \sum_{n=1}^N a_{jn}^{l-1} \mathbf{e}_n = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, \ldots, a_{jN}^{l-1}\right]^T \in \mathbb{R}^N$, the bias vector of neuron i in layer l is $\mathbf{b}_i^l = \sum_{n=1}^N b_{in}^l \mathbf{e}_n = \left[b_{i1}^l, b_{i2}^l, \ldots, b_{iN}^l\right]^T \in \mathbb{R}^N$. As a result, $\mathbf{z}_i^l = \sum_{n=1}^N z_{in}^l \mathbf{e}_n = \left[z_{i1}^l, z_{i2}^l, \ldots, z_{iN}^l\right]^T \in \mathbb{R}^N$. When l equals 1, the vector $\mathbf{z}$ is simply the input vector $\mathbf{x}$. After $\mathbf{z}_i^l$ is calculated, the output vector $\mathbf{a}_i^l$ of the hidden neuron i is as follows:

$$\mathbf{a}_i^l = \begin{bmatrix} a_{i1}^l \\ a_{i2}^l \\ \vdots \\ a_{iN}^l \end{bmatrix} = \phi(\mathbf{z}_i^l) = \begin{bmatrix} \phi(z_{i1}^l) \\ \phi(z_{i2}^l) \\ \vdots \\ \phi(z_{iN}^l) \end{bmatrix}, \tag{2.7}$$

where ϕ could be any differentiable activation function. Then the output vector of neuron g $(1 \leq g \leq R_L)$ in output layer L is defined as:

$$\mathbf{y}_g^L = \begin{bmatrix} y_{g1}^L \\ y_{g2}^L \\ \vdots \\ y_{gN}^L \end{bmatrix} = \phi(\mathbf{z}_g^L) = \begin{bmatrix} \phi(z_{g1}^L) \\ \phi(z_{g2}^L) \\ \vdots \\ \phi(z_{gN}^L) \end{bmatrix}, \tag{2.8}$$

where $\mathbf{z}_g^L$ is the input vector and $\mathbf{y}_g^L$ is the output vector of a neuron g. The objective of the training process is to estimate the parameters that minimize the cost function, defined as:

$$C(\mathbf{\Theta}) = \sum_{m=1}^{M} loss(\mathbf{Y}_m, f(\mathbf{X}_m; \mathbf{\Theta})) \,, \tag{2.9}$$

where M stands for the number of training data patterns, $\mathbf{\Theta}$ is the set of all training parameters (weights and biases), and $\mathbf{Y}_m$ is the training label related to the input $\boldsymbol{X}_m$. The output $f(\mathbf{X}_m; \mathbf{\Theta})$ is the predicted version of $\mathbf{Y}_m$, made up of $\mathbf{y}_g^L$ mentioned in Eq. 2.9. The $loss(\mathbf{Y}_m, f(\mathbf{X}_m; \mathbf{\Theta}))$ function measures the difference between the predicted results and the training labels.

2.3 Backpropagation algorithm for ABIPNN

The bilinear product is also employed in the process of backpropagation learning. Before we present the error backpropagation process (Algorithm 1), we need to first derive:

1. The gradients of the biases $\mathbf{b}_i^l$.

2. The gradients of the weights $\mathbf{w}_{ij}^l$.

3. The backpropagation of local gradients $\mathbf{d}_k^{l+1} \rightarrow \mathbf{d}_i^l$.

Following Eq. 2.3, the bilinear product can be viewed as matrix-vector multiplication. Hence, Eq. 2.6 can be rewritten as

$$\begin{aligned}
\mathbf{z}_i^l &= \sum_{j=1}^{R_{l-1}} \mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1} + \mathbf{b}_i^l = \sum_{j=1}^{R_{l-1}} \left[\mathbf{w}_{ij}^l\right]_{\bullet} \mathbf{a}_j^{l-1} + \mathbf{b}_i^l \\
&= \sum_{j=1}^{R_{l-1}} \left[\mathbf{w}_{ij}^l \bullet \mathbf{e}_1, \mathbf{w}_{ij}^l \bullet \mathbf{e}_2, \ \ldots, \mathbf{w}_{ij}^l \bullet \mathbf{e}_N\right] \mathbf{a}_j^{l-1} + \mathbf{b}_i^l \\
&= \sum_{j=1}^{R_{l-1}} \left(a_{j1}^{l-1} \mathbf{w}_{ij}^l \bullet \mathbf{e}_1 + \cdots + a_{jN}^{l-1} \mathbf{w}_{ij}^l \bullet \mathbf{e}_N\right) + \mathbf{b}_i^l.
\end{aligned} \tag{2.10}$$

Here, $\mathbf{w}_{ij}^l$ can be formulated as the summation of scalar-vector multiplications:

$$\begin{aligned}
\mathbf{z}_i^l = \sum_{j=1}^{R_{l-1}} &\left(a_{j1}^{l-1} \left(w_{ij1}^l \mathbf{e}_1 + \cdots + w_{ijN}^l \mathbf{e}_N\right) \bullet \mathbf{e}_1 \right. \\
&+ a_{j2}^{l-1} \left(w_{ij1}^l \mathbf{e}_1 + \cdots + w_{ijN}^l \mathbf{e}_N\right) \bullet \mathbf{e}_2 + \ldots \\
&\left. + a_{jN}^{l-1} \left(w_{ij1}^l \mathbf{e}_1 + \cdots + w_{ijN}^l \mathbf{e}_N\right) \bullet \mathbf{e}_N\right) + \mathbf{b}_i^l.
\end{aligned} \tag{2.11}$$

To estimate the first-order partial derivatives of a vector-valued function, we apply the concept of Jacobian matrix, which is a matrix containing all the partial derivatives. For

the vector-valued function $f : \mathbb{R}^N \to \mathbb{R}^M$, the Jacobian matrix is defined as $\left[\frac{\partial f}{\partial \boldsymbol{x}}\right]_{ij} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i \in \mathbb{R}^{N \times M}$. The partial derivative of C with respect to $\mathbf{b}_i^l$ is as follows:

$$\begin{aligned}
\frac{\partial C}{\partial \mathbf{b}_i^l} &= \left[\frac{\partial C}{\partial b_{i1}^l}, \frac{\partial C}{\partial b_{i2}^l}, \ \ldots, \frac{\partial C}{\partial b_{iN}^l}\right] \\
&= \begin{bmatrix}
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial b_{i1}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial b_{i1}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial b_{i1}^l} \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial b_{i2}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial b_{i2}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial b_{i2}^l} \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial b_{i3}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial b_{i3}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial b_{i3}^l} \\
\vdots \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial b_{iN}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial b_{iN}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial b_{iN}^l}
\end{bmatrix}^T,
\end{aligned} \tag{2.12}$$

where $\frac{\partial C}{\partial \mathbf{b}_i^l}$ stands for the gradient vector of $\mathbf{b}_i^l$ belonging to neuron i in layer l. The terms $\frac{\partial C}{\partial z_{in}^l}$ $(1 \leq n \leq N)$ are extracted into an N-dimensional local gradient vector $\frac{\partial C}{\partial \mathbf{z}_i^l}$ and we call it $\mathbf{d}_i^l$ for convenience:

$$\begin{aligned}
\frac{\partial C}{\partial \mathbf{z}_i^l} &= \left[\frac{\partial C}{\partial z_{i1}^l}, \frac{\partial C}{\partial z_{i2}^l}, \ \ldots, \frac{\partial C}{\partial z_{iN}^l}\right] \\
&\triangleq \left[d_{i1}^l, d_{i2}^l, \ \ldots, d_{iN}^l\right] = \mathbf{d}_i^l.
\end{aligned} \tag{2.13}$$

The terms $\left\{\frac{\partial z_{i1}^l}{\partial b_{in}^l}, \frac{\partial z_{i2}^l}{\partial b_{in}^l}, \ldots, \frac{\partial z_{iN}^l}{\partial b_{in}^l}\right\}$ $(1 \leq n \leq N)$ can be obtained by differentiating Eq. 2.11:

$$\begin{aligned}\frac{\partial \mathbf{z}_i^l}{\partial b_{in}^l} &= \left[\frac{\partial z_{i1}^l}{\partial b_{in}^l}, \frac{\partial z_{i2}^l}{\partial b_{in}^l}, \ldots, \frac{\partial z_{iN}^l}{\partial b_{in}^l}\right]^T \\ &= \left[\frac{\partial b_{i1}^l}{\partial b_{in}^l}, \frac{\partial b_{i2}^l}{\partial b_{in}^l}, \ldots, \frac{\partial b_{iN}^l}{\partial b_{in}^l}\right]^T \\ &= \underbrace{\left[0, \ldots, 1, \ldots, 0\right]^T}_{\text{the } n\text{-th entry is 1}}.\end{aligned} \tag{2.14}$$

By combining Eqs. 3.5 and 3.6, the derivative $\frac{\partial C}{\partial b_{in}^l}$ can be formulated as a dot product:

$$\frac{\partial C}{\partial b_{in}^l} = \mathbf{d}_i^l \left[\frac{\partial \mathbf{z}_i^l}{\partial b_{in}^l}\right] = \underbrace{\left[d_{i1}^l, d_{i2}^l, \ldots, d_{iN}^l\right]}_{1\times N} \underbrace{\begin{bmatrix}0\\ \vdots \\ 1 \\ \vdots \\ 0\end{bmatrix}}_{N\times 1} = d_{in}^l. \tag{2.15}$$

Then, each entry of $\frac{\partial C}{\partial \mathbf{b}_i^l}$ can be estimated by the same procedure from Eqs. 3.6 and 2.15.

After each entry is estimated, Eq. 3.3 can be derived as follows:

$$\frac{\partial C}{\partial \mathbf{b}_i^l} = \left[\frac{\partial C}{\partial b_{i1}^l}, \ldots, \frac{\partial C}{\partial b_{iN}^l}\right] = \left[d_{i1}^l, \ldots, d_{iN}^l\right] = \mathbf{d}_i^l. \tag{2.16}$$

As a result, it is evident that the derivative of $\mathbf{b}_i^l$ is completely equal to local gradient vector $\mathbf{d}_i^l$. Next, we derive the partial derivative of C with respect to $\mathbf{w}_{ij}^l$ as follows:

$$\begin{aligned}
\frac{\partial C}{\partial \mathbf{w}_{ij}^l} &= \left[\frac{\partial C}{\partial w_{ij1}^l}, \frac{\partial C}{\partial w_{ij2}^l}, \ldots, \frac{\partial C}{\partial w_{ijN}^l} \right] \\
&= \begin{bmatrix}
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ij1}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ij1}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ij1}^l} \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ij2}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ij2}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ij2}^l} \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ij3}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ij3}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ij3}^l} \\
\vdots \\
\frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ijN}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ijN}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ijN}^l}
\end{bmatrix}^T
\end{aligned} \tag{2.17}$$

where $\frac{\partial C}{\partial \mathbf{w}_{ij}^l}$ stands for the gradient vector of $\mathbf{w}_{ij}^l$ connecting neuron j to neuron i in layer l and it consists of N elements. From Eq. 3.5, here the terms $\left\{ \frac{\partial C}{\partial z_{i1}^l}, \frac{\partial C}{\partial z_{i2}^l}, \ldots, \frac{\partial C}{\partial z_{iN}^l} \right\}$ are extracted as the local gradient vector $\mathbf{d}_i^l$. Likewise, we can obtain the terms $\left\{ \frac{\partial z_{i1}^l}{\partial w_{ijn}^l}, \frac{\partial z_{i2}^l}{\partial w_{ijn}^l}, \ldots, \frac{\partial z_{iN}^l}{\partial w_{ijn}^l} \right\}$ $(1 \leq n \leq N)$ by differentiating Eq. 2.11:

$$\begin{aligned}
\frac{\partial \mathbf{z}_i^l}{\partial w_{ijn}^l} &= \left[\frac{\partial z_{i1}^l}{\partial w_{ijn}^l}, \frac{\partial z_{i2}^l}{\partial w_{ijn}^l}, \ldots, \frac{\partial z_{iN}^l}{\partial w_{ijn}^l} \right]^T \\
&= a_{j1}^{l-1}\mathbf{e}_n \bullet \mathbf{e}_1 + a_{j2}^{l-1}\mathbf{e}_n \bullet \mathbf{e}_2 + \cdots + a_{jN}^{l-1}\mathbf{e}_n \bullet \mathbf{e}_N \\
&= \left[\mathbf{e}_n \bullet \mathbf{e}_1, \mathbf{e}_n \bullet \mathbf{e}_2, \ldots, \mathbf{e}_n \bullet \mathbf{e}_N \right] \left[a_{j1}^{l-1}, \ldots a_{jN}^{l-1} \right]^T \\
&= \left[\mathbf{e}_n \bullet \mathbf{e}_1, \mathbf{e}_n \bullet \mathbf{e}_2, \ldots, \mathbf{e}_n \bullet \mathbf{e}_N \right] \mathbf{a}_j^{l-1},
\end{aligned} \tag{2.18}$$

which is formulated as a matrix-vector multiplication. Each column in the matrix is the bilinear product of two standard bases. Different bilinear products contribute to differ-

ent results. Then, the derivative $\frac{\partial C}{\partial w_{ijn}^l}$ can be formulated as a dot product by combining Eqs. 3.5 and 2.18:

$$\begin{aligned}
\frac{\partial C}{\partial w_{ijn}^l} &= \mathbf{d}_i^l \left[\frac{\partial \mathbf{z}_i^l}{\partial w_{ijn}^l} \right] \\
&= \underbrace{\left[d_{i1}^l, d_{i2}^l, \ \ldots, d_{iN}^l \right]}_{1 \times N} \\
&\qquad \left[\underbrace{\left[\mathbf{e}_n \bullet \mathbf{e}_1, \mathbf{e}_n \bullet \mathbf{e}_2, \ \ldots, \mathbf{e}_n \bullet \mathbf{e}_N \right]}_{N \times N} \underbrace{\mathbf{a}_j^{l-1}}_{N \times 1} \right] \\
&= \mathbf{d}_i^l \left[\sum_{h=1}^{N} a_{jh}^{l-1} \mathbf{e}_n \bullet \mathbf{e}_h \right].
\end{aligned} \tag{2.19}$$

Each entry of $\frac{\partial C}{\partial \mathbf{w}_{ij}^l}$ can be estimated by the same procedure from Eqs. 2.18 and 2.19. After each entry is estimated, Eq. 3.8 can be rewritten as:

$$\begin{aligned}
\frac{\partial C}{\partial \mathbf{w}_{ij}^l} &= \left[\frac{\partial C}{\partial w_{ij1}^l}, \frac{\partial C}{\partial w_{ij2}^l}, \ \ldots, \frac{\partial C}{\partial w_{ijN}^l} \right] \\
&= \left[\mathbf{d}_i^l \left[\sum_{h=1}^{N} a_{jh}^{l-1} \mathbf{e}_1 \bullet \mathbf{e}_h \right], \ldots, \mathbf{d}_i^l \left[\sum_{h=1}^{N} a_{jh}^{l-1} \mathbf{e}_N \bullet \mathbf{e}_h \right] \right] \\
&= \mathbf{d}_i^l \left[\mathbf{e}_1 \bullet \left[\sum_{h=1}^{N} a_{jh}^{l-1} \mathbf{e}_h \right], \ \ldots, \mathbf{e}_N \bullet \left[\sum_{h=1}^{N} a_{jh}^{l-1} \mathbf{e}_h \right] \right],
\end{aligned} \tag{2.20}$$

and referring to Eq. 2.4, we can rewrite Eq. 2.20 into:

$$\begin{aligned}
\frac{\partial C}{\partial \mathbf{w}_{ij}^l} &= \mathbf{d}_i^l \left[\mathbf{e}_1 \bullet \mathbf{a}_j^{l-1}, \mathbf{e}_2 \bullet \mathbf{a}_j^{l-1}, \ \ldots, \mathbf{e}_N \bullet \mathbf{a}_j^{l-1} \right] \\
&= \mathbf{d}_i^l \left[\mathbf{a}_j^{l-1} \right]_{\bullet}^{\dagger},
\end{aligned} \tag{2.21}$$

which is a vector-matrix multiplication with the transmuted representation. After the derivative $\frac{\partial C}{\partial \mathbf{w}_{ij}^{l}}$ is calculated, we derive the local gradient vector $\mathbf{d}_i^l$ from layer $l+1$ by:

$$\begin{aligned}\mathbf{d}_i^l = \frac{\partial C}{\partial \mathbf{z}_i^l} &= \sum_{k=1}^{R_{l+1}} \frac{\partial C}{\partial \mathbf{z}_k^{l+1}} \frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l} \frac{\partial \mathbf{a}_i^l}{\partial \mathbf{z}_i^l} \\ &= \sum_{k=1}^{R_{l+1}} \frac{\partial C}{\partial \mathbf{z}_k^{l+1}} \frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l} \dot{\phi}(\mathbf{z}_i^l),\end{aligned} \tag{2.22}$$

which is a vector-matrix-matrix multiplication, where $\mathbf{z}_k^{l+1}$ stands for an N-dimensional input vector of neuron k in layer $l+1$ and $\mathbf{a}_i^l$ stands for an N-dimensional output vector of neuron i in layer l. The vector $\frac{\partial C}{\partial \mathbf{z}_k^{l+1}}$ is regarded as an N-dimensional local gradient vector, defined as $\mathbf{d}_k^{l+1}$ in layer $l+1$. Referring to Eqs. 3.3 to 3.7, we can see that $\mathbf{d}_k^{l+1}$ can be regarded as the derivative of $\mathbf{b}_k^{l+1}$. The matrix $\dot{\phi}(\mathbf{z}_i^l)$ represents the derivative of the activation function, which is an $N \times N$ diagonal matrix:

$$\dot{\phi}(\mathbf{z}_i^l) = \frac{\partial \mathbf{a}_i^l}{\partial \mathbf{z}_i^l} = \begin{bmatrix} \frac{\partial a_{i1}^l}{\partial z_{i1}^l} & 0 & \dots & 0 \\ 0 & \frac{\partial a_{i2}^l}{\partial z_{i2}^l} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial a_{iN}^l}{\partial z_{iN}^l} \end{bmatrix} = \begin{bmatrix} \dot{\phi}(z_{i1}^l) & 0 & \dots & 0 \\ 0 & \dot{\phi}(z_{i2}^l) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \dot{\phi}(z_{iN}^l) \end{bmatrix}, \tag{2.23}$$

where $\dot{\phi}$ can be an arbitrary differentiable activation function. Next, the matrix $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$ is also a $N \times N$ square matrix:

$$\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l} = \begin{bmatrix} \frac{\partial z_{k1}^{l+1}}{\partial a_{i1}^l} & \frac{\partial z_{k1}^{l+1}}{\partial a_{i2}^l} & \cdots & \frac{\partial z_{k1}^{l+1}}{\partial a_{iN}^l} \\ \frac{\partial z_{k2}^{l+1}}{\partial a_{i1}^l} & \frac{\partial z_{k2}^{l+1}}{\partial a_{i2}^l} & \cdots & \frac{\partial z_{k2}^{l+1}}{\partial a_{iN}^l} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{kN}^{l+1}}{\partial a_{i1}^l} & \frac{\partial z_{kN}^{l+1}}{\partial a_{i2}^l} & \cdots & \frac{\partial z_{kN}^{l+1}}{\partial a_{iN}^l} \end{bmatrix}. \tag{2.24}$$

We calculate the above derivatives columnwise. The calculations are based on the feed-forward process of $\mathbf{z}_k^{l+1}$:

$$\begin{aligned} \mathbf{z}_k^{l+1} &= \sum_{i=1}^{R_l} \mathbf{w}_{ki}^{l+1} \bullet \mathbf{a}_i^l + \mathbf{b}_k^{l+1} = \sum_{i=1}^{R_l} \mathbf{w}_{ki}^{l+1} \bullet \left(\sum_{n=1}^{N} a_{in}^l \mathbf{e}_n \right) + \mathbf{b}_k^{l+1} \\ &= \sum_{i=1}^{R_l} \mathbf{w}_{ki}^{l+1} \bullet \left[a_{i1}^l \mathbf{e}_1 + a_{i2}^l \mathbf{e}_2 + \cdots + a_{iN}^l \mathbf{e}_N \right] + \mathbf{b}_k^{l+1}. \end{aligned} \tag{2.25}$$

Then, the derivative of the n-th column of Eq. 2.24 is derived by differentiating Eq. 2.25:

$$\begin{aligned} \frac{\partial \mathbf{z}_k^{l+1}}{\partial a_{in}^l} &= \left[\frac{\partial z_{k1}^{l+1}}{\partial a_{in}^l}, \frac{\partial z_{k2}^{l+1}}{\partial a_{in}^l}, \ldots, \frac{\partial z_{kN}^{l+1}}{\partial a_{in}^l} \right]^T \\ &= \mathbf{w}_{ki}^{l+1} \bullet \mathbf{e}_n \\ &= \left(w_{ki1}^{l+1} \mathbf{e}_1 + w_{ki2}^{l+1} \mathbf{e}_2 + \cdots + w_{kiN}^{l+1} \mathbf{e}_N \right) \bullet \mathbf{e}_n \\ &= \sum_{h=1}^{N} w_{kih}^{l+1} \mathbf{e}_h \bullet \mathbf{e}_n \\ &= \underbrace{[\mathbf{e}_1 \bullet \mathbf{e}_n, \mathbf{e}_2 \bullet \mathbf{e}_n, \ \ldots \ , \mathbf{e}_N \bullet \mathbf{e}_n]}_{N \times N} \underbrace{\mathbf{w}_{ki}^{l+1}}_{N \times 1}, \end{aligned} \tag{2.26}$$

resulting in a column vector. Since each column of the matrix $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$ can be estimated by the same procedure mentioned above, the derivatives of the matrix can be derived as:

$$\begin{aligned}\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l} &= \left[\left[\sum_{h=1}^{N} w_{kih}^{l+1}\mathbf{e}_h \bullet \mathbf{e}_1\right], \left[\sum_{h=1}^{N} w_{kih}^{l+1}\mathbf{e}_h \bullet \mathbf{e}_2\right], \right. \\ &\qquad \left. \ldots, \left[\sum_{h=1}^{N} w_{kih}^{l+1}\mathbf{e}_h \bullet \mathbf{e}_N\right]\right] \\ &= \left[\mathbf{w}_{ki}^{l+1} \bullet \mathbf{e}_1, \mathbf{w}_{ki}^{l+1} \bullet \mathbf{e}_2, \ \ldots, \mathbf{w}_{ki}^{l+1} \bullet \mathbf{e}_N\right] \\ &= \left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet},\end{aligned} \tag{2.27}$$

in which the matrix is made up of N vectors. After estimating the matrix $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$, Eq. 3.10 can be rewritten as follows:

$$\mathbf{d}_i^l = \sum_{k=1}^{R_{l+1}} \mathbf{d}_k^{l+1}\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet} \dot{\phi}(\mathbf{z}_i^l), \tag{2.28}$$

which becomes a vector-matrix-vector multiplication. It is evident that local gradient vector $\mathbf{d}_i^l$ in layer l is backpropagated from $\mathbf{d}_k^{l+1}$ in layer $l+1$. From this we see that the local gradient can be inferred from the output layer. In the output layer L, the local gradient vector $\mathbf{d}_g^L$ is defined as:

$$\begin{aligned}\frac{\partial C}{\partial \mathbf{z}_g^L} &= \left[\frac{\partial C}{\partial z_{g1}^L}, \frac{\partial C}{\partial z_{g2}^L}, \ \ldots, \frac{\partial C}{\partial z_{gN}^L}\right] \\ &= \left[\frac{\partial C}{\partial y_{g1}^L}\frac{\partial y_{g1}^L}{\partial z_{g1}^L}, \frac{\partial C}{\partial y_{g2}^L}\frac{\partial y_{g2}^L}{\partial z_{g2}^L}, \ \ldots, \frac{\partial C}{\partial y_{gN}^L}\frac{\partial y_{gN}^L}{\partial z_{gN}^L}\right] \\ &= \left[\frac{\partial C}{\partial y_{g1}^L}\dot{\phi}(z_{g1}^L), \frac{\partial C}{\partial y_{g2}^L}\dot{\phi}(z_{g2}^L), \ \ldots, \frac{\partial C}{\partial y_{gN}^L}\dot{\phi}(z_{gN}^L)\right],\end{aligned} \tag{2.29}$$

Algorithm 1 Arbitrary bilinear product backpropagation

Input: Training inputs $\{\mathbf{X}_m\}_{m=1}^M$
Training targets $\{\mathbf{Y}_m\}_{m=1}^M$
Bilinear product $\bullet$
Activation function ϕ
Output: Parameters $\boldsymbol{\Theta} = \left\{\boldsymbol{\mathcal{W}}^l, \boldsymbol{\mathcal{B}}^l\right\}_{l=1}^L$

while not converged **do**
  for each minibatch $\in \{\mathbf{X}\}_{m=1}^M$ **do**
    Compute $\left\{\boldsymbol{\mathcal{Z}}^l, \boldsymbol{\mathcal{A}}^l\right\}_{l=1}^L$ with Eqs. 2.6–2.8 (feedforward)
    Compute $\left\{\boldsymbol{\mathcal{D}}^l\right\}_{l=1}^L$ with Eqs. 2.28 and 3.12 (backprop)
    Update $\left\{\boldsymbol{\mathcal{W}}^l\right\}_{l=1}^L$ with the gradients from Eq. 2.21
    Update $\left\{\boldsymbol{\mathcal{B}}^l\right\}_{l=1}^L$ with the gradients from Eq. 3.7
  end for
end while

where $\frac{\partial C}{\partial y_{gn}^L}$ $(1 \leq n \leq N)$ is the derivative of the loss function used in the feedforward process, and $\dot{\phi}(z_{gn}^L)$ is the derivative of the activation function. The above process is summarized in Algorithm 1.

2.4 Example: Circular convolution as the bilinear product

In the above, we generalized the vector NNs using arbitrary bilinear products. This algorithm can be realized to train model parameters for scalar neurons or vector neurons from training data based on matrices or tensors. In this subsection, we derive $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ and $\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet$ for circular convolution. The derivation is described between layer $l-1$ and layer l.

Suppose $\mathbf{w}_{ij}^l \bullet \mathbf{a}_{ij}^{l-1}$ stands for the usual circular convolution, $\mathbf{w}_{ij}^l = \left[w_{ij1}^l, w_{ij2}^l, \ldots, w_{ijN}^l\right]^T \in \mathbb{R}^N$ and $\mathbf{a}_j^{l-1} = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, \ldots, a_{jN}^{l-1}\right]^T \in \mathbb{R}^N$ are both N-dimensional vectors, and the matrix representation of $\mathbf{w}_{ij}^l \bullet \mathbf{a}_{ij}^{l-1}$ is:

$$\left[\mathbf{w}_{ij}^l\right]_\bullet = \begin{bmatrix} w_{ij1}^l & w_{ijN}^l & w_{ij(N-1)}^l & \cdots & w_{ij2}^l \\ w_{ij2}^l & w_{ij1}^l & w_{ijN}^l & \cdots & w_{ij3}^l \\ w_{ij3}^l & w_{ij2}^l & w_{ij1}^l & \cdots & w_{ij4}^l \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{ijN}^l & w_{ij(N-1)}^l & w_{ij(N-2)}^l & \cdots & w_{ij1}^l \end{bmatrix}, \tag{2.30}$$

where the weight vector is formulated as an $N \times N$ square matrix with w_{ij1}^l on the main

diagonal. The matrix $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ for Eq. 2.21 is extended as follows:

$$\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger = \begin{bmatrix} a_{j1}^{l-1} & a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & \dots & a_{j2}^{l-1} \\ a_{j2}^{l-1} & a_{j1}^{l-1} & a_{jN}^{l-1} & \dots & a_{j3}^{l-1} \\ a_{j3}^{l-1} & a_{j2}^{l-1} & a_{j1}^{l-1} & \dots & a_{j4}^{l-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & a_{j(N-2)}^{l-1} & \dots & a_{j1}^{l-1} \end{bmatrix}, \tag{2.31}$$

where the permutation of elements in the matrix $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ is identical to the matrix $\left[\mathbf{w}_{ij}^{l}\right]_\bullet$. The matrix $\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet$ for circular convolution is extended as:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet = \begin{bmatrix} w_{ki1}^{l+1} & w_{kiN}^{l+1} & w_{ki(N-1)}^{l+1} & \dots & w_{ki2}^{l+1} \\ w_{ki2}^{l+1} & w_{ki1}^{l+1} & w_{kiN}^{l+1} & \dots & w_{ki3}^{l+1} \\ w_{ki3}^{l+1} & w_{ki2}^{l+1} & w_{ki1}^{l+1} & \dots & w_{ki4}^{l+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{kiN}^{l+1} & w_{ki(N-1)}^{l+1} & w_{ki(N-2)}^{l+1} & \dots & w_{ki1}^{l+1} \end{bmatrix}. \tag{2.32}$$

Note that when N equals two, this architecture becomes the hyperbolic NN [17]. When N is larger than two, this product is also known as polar complex multiplication [47, 49]. Detail investigations of more bilinear products are described in Appendix A.

Chapter 3

Deep Cyclic-group Networks

The peculiarity of CNNs is that it learns input invariance by using kernel maps and outputs feature maps as detected features. Elements in a feature map and a kernel map represent hidden neurons and weights respectively. It uses the same weights for each neuron. Thus, CNNs can be regarded as a kind of DNN architecture using sharing weights for each of hidden neurons in the same layer. In previous work, DCN and DQN extend each neuron and weight to two- and four-dimensions respectively, using complex and quaternion algebras. However, they lack the ability to deal with input data of arbitrary dimensions. The idea of our proposed DCGN is to extend the capacity of each neuron from scalars to vectors with cyclic group algebras. Here, the scalar multiplications in the conventional convolutional procedure are changed into vector multiplications along mode-3 fibers with the help of circular convolutions, which can also be formulated as vector-matrix multiplications with circulant matrices [45, 50]. In other words, our work replaces scalar multiplications with vector multiplications through circulant matrices.

In DCGNs, both the kernel and feature maps are three-way tensors comprising N-dimensional mode-3 fibers. Each fiber in a feature map stands for a N-dimensional vector-valued neuron and its output is also a N-dimensional fiber. The circular convolution is operated between a vector-valued neuron and a fiber coming from a weight kernel map. The convolution procedure is illustrated in Figure 3.1.

In what follows, we firstly derive the learning algorithm from the point of view of a vector neuron, and then describe the overall DCGN model for regression problems and its extension to classification problems. Below, matrices are represented by bold uppercase letters, and vectors by bold lowercase letters. Matrix slices of tensors are represented as matrices and vector slices of matrix are represented as vectors. We provide the simplified version of backpropagation learning algorithm and treat vectors as row vectors. Detailed version will be released along with the code.

3.1 Operation of Circular Convolutions

To perform the learning process between a three-order tensor feature map and a three-order tensor kernel map, we leverage the circular convolution to be an basic operator in the network. Given two N-dimensional vectors, $\mathbf{p} = [p_1, p_2, \ldots, p_N] \in \mathbb{R}^N$ and $\mathbf{q} = [q_1, q_2, \ldots, q_N] \in \mathbb{R}^N$, we can compute the circular convolution between them, denoted as $\mathbf{p} * \mathbf{q}$, with vector-matrix multiplication by using the circulant matrix $[\mathbf{q}]_* = \text{circ}(\mathbf{q})$

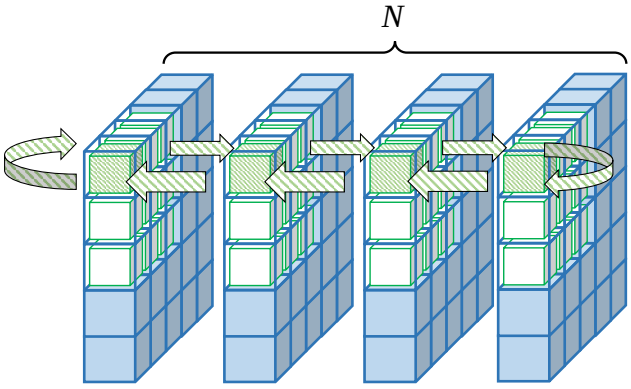


Figure 3.1: Illustration of the convolution procedure when using cyclic group algebra (best viewed in color). Each cube represents a scalar. Collectively, the N matrices of cubes represent a three-way tensor of mode-3 dimension N, each matrix is a frontal slice of the tensor, and each mode-3 fiber is an N-dimensional vector. The feature map and the kernel map are composed of blue cubes and green cubes, respectively. In the convolution procedure, each mode-3 fiber of the kernel map performs circular convolution with a mode-3 fiber of the feature map, by rotating itself along the mode-3 dimension, as the green textured cubes represent. Therefore, via the kernel map we can learn the association between the elements across the N dimensions by using a linear kernel and weighted sum. The kernel map moves along the (mode-1, mode-2) plane, each time at a distance of a stride.

associated to $\mathbf{q}$ as follows:

$$\mathbf{p} * \mathbf{q} = \mathbf{p}[\mathbf{q}]_* = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_N \end{bmatrix}^T \begin{bmatrix} q_1 & q_2 & q_3 & \cdots & q_N \\ q_N & q_1 & q_2 & \cdots & q_{N-1} \\ q_{N-1} & q_N & q_1 & \cdots & q_{N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ q_2 & q_3 & q_4 & \cdots & q_1 \end{bmatrix}. \tag{3.1}$$

We note that $[\mathbf{q}]_* \in \mathbb{R}^{N \times N}$ and $\mathbf{p} * \mathbf{q} \in \mathbb{R}^N$. Algebra of circulant matrices is the vector's isomorphism. The result of $\mathbf{p} * \mathbf{q}$ is identical to the definition of circular convolution operation. In our proposed model, the feedforward and backpropagation processes will be described by using above the formulation and the matrix representation.

3.2 Feedforward process for DCGN

In vector neural learning with cyclic group algebras, the inputs, outputs, weights and biases are all N-dimensional vectors. Specifically, for a network with L layers, the relation between the output of a neuron i in layer l ($1 \leq l \leq L$), $\mathbf{a}_i^l \in \mathbb{R}^N$, and the output of a neuron j in the preceding layer $l-1$, $\mathbf{a}_j^{l-1} \in \mathbb{R}^N$, can be written as:

$$\mathbf{a}_i^l \triangleq \phi(\mathbf{z}_i^l) = \phi\left(\sum_{j=1}^{J^{l-1}} \mathbf{w}_{ij}^l * \mathbf{a}_j^{l-1} + \mathbf{b}_i^l\right) , \tag{3.2}$$

where $\phi(\cdot)$ is a differentiable activation function, $\mathbf{w}_{ij}^l \in \mathbb{R}^N$ the weight vector connecting the two neurons, J^{l-1} the number of neuron in layer $l-1$, and $\mathbf{b}_i^l \in \mathbb{R}^N$ a bias vector. For $l = 1$, we have $J^{l-1} = 1$ and $\mathbf{a}^{l-1} \triangleq \mathbf{x} \in \mathbb{R}^N$, the input data vector of the network. For $l = L$, we have $J^l = 1$, and we define $\mathbf{y} \in \mathbb{R}^N$ as $\mathbf{a}^L$, the output of the network.

3.3 Backpropagation algorithm for DCGN

During the training phase, we are given the target output $\hat{\mathbf{y}} \in \mathbb{R}^N$ for every input data instance, so that we can compare $\hat{\mathbf{y}}$ and $\mathbf{y}$ via some cost function $\epsilon(\mathbf{y}, \hat{\mathbf{y}})$ to know how good the model parameters $\mathbf{\Theta} \triangleq \{\mathbf{w}_{ij}^l, \mathbf{b}_i^l\}$ are. Here, we use $\mathbf{\Theta}$ to denote the collection of all the trainable parameters (i.e., weights and biases) across the L layers. Given the empirical cost C calculated over all the training data instances, we can use backpropagation to update the model parameters $\mathbf{\Theta}$. Finally, given the training data and the cost function $C(\mathbf{\Theta})$, the neural network is used to perform nonlinear mapping between the inputs and

the outputs. The objective of the training process is to estimate parameters minimizing the cost function.

From the previous sections, the feedforward process of the proposed architecture operates with the circular convolution, which is formulated as matrix-vector multiplication. In the backpropagation process, the learning algorithm is derived in the same way. To use backpropagation, we need to take the derivative of C with respect to each parameter $\mathbf{w}_{ij}^l$ or $\mathbf{b}_i^l$. Unlike ordinary CNNs, all the parameters here are vectors. Therefore, in calculating the derivatives we need to compute the Jacobian matrix. For the bias vectors $\mathbf{b}_i^l$, this amounts to calculating the N-dimensional gradient vector:

$$\frac{\partial C}{\partial \mathbf{b}_i^l} = \left[\frac{\partial C}{\partial b_{i1}^l}, \frac{\partial C}{\partial b_{i2}^l}, \ \ldots, \frac{\partial C}{\partial b_{iN}^l} \right], \tag{3.3}$$

where

$$\frac{\partial C}{\partial b_{ik}^l} = \sum_{n=1}^{N} \frac{\partial C}{\partial z_{in}^l} \frac{\partial z_{in}^l}{\partial b_{ik}^l}. \tag{3.4}$$

It can be shown that $\frac{\partial z_{in}^l}{\partial b_{ik}^l} = 1$ only when $n = k$, otherwise it is zero. It will become apparent later that it is more convenient to denote $\frac{\partial C}{\partial z_{in}^l}$ simply as d_{in}^l and accordingly define the *local gradient vector* $\mathbf{d}_i^l = \left[d_{i1}^l, d_{i2}^l, \ldots, d_{iN}^l \right]$ as follows:

$$\mathbf{d}_i^l \triangleq \frac{\partial C}{\partial \mathbf{z}_i^l} = \left[\frac{\partial C}{\partial z_{i1}^l}, \frac{\partial C}{\partial z_{i2}^l}, \ \ldots, \frac{\partial C}{\partial z_{iN}^l} \right]. \tag{3.5}$$

The terms $\left\{ \frac{\partial z_{i1}^l}{\partial b_{ik}^l}, \frac{\partial z_{i2}^l}{\partial b_{ik}^l}, \ \ldots, \frac{\partial z_{iN}^l}{\partial b_{in}^l} \right\}$ $(1 \leq k \leq N)$ can be obtained by differentiating Eq. 3.2:

:

$$\begin{aligned}\frac{\partial \mathbf{z}_i^l}{\partial b_{in}^l} &= \left[\frac{\partial z_{i1}^l}{\partial b_{ik}^l}, \frac{\partial z_{i2}^l}{\partial b_{ik}^l}, \ldots, \frac{\partial z_{iN}^l}{\partial b_{ik}^l}\right]^T \\ &= \left[\frac{\partial b_{i1}^l}{\partial b_{ik}^l}, \frac{\partial b_{i2}^l}{\partial b_{ik}^l}, \ldots, \frac{\partial b_{iN}^l}{\partial b_{ik}^l}\right]^T \\ &= \underbrace{\left[0, \ldots, 1, \ldots, 0\right]^T}_{\text{the } n\text{-th entry is 1}}.\end{aligned} \tag{3.6}$$

It can be shown that $\frac{\partial C}{\partial b_{ik}^l} = d_{ik}^l$ and accordingly,

$$\frac{\partial C}{\partial \mathbf{b}_i^l} = \left[\ldots, \sum_{n=1}^{N} \frac{\partial C}{\partial z_{in}^l} \frac{\partial z_{in}^l}{\partial b_{ik}^l}, \ldots\right] = \mathbf{d}_i^l . \tag{3.7}$$

As a consequence, it is obvious that the derivative of $\mathbf{b}_i^l$ is equal to local gradient vector $\mathbf{d}_i^l$. Next, from the chain rule, we can calculate the derivative of C w.r.t. the weight vector $\mathbf{w}_{ij}^l$ similarly. With some algebra, it can be shown that

$$\begin{aligned}\frac{\partial C}{\partial \mathbf{w}_{ij}^l} &= \left[\frac{\partial C}{\partial w_{ij1}^l}, \frac{\partial C}{\partial w_{ij2}^l}, \ldots, \frac{\partial C}{\partial w_{ijN}^l}\right] \\ &= \begin{bmatrix} \frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ij1}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ij1}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ij1}^l} \\ \frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ij2}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ij2}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ij2}^l} \\ \vdots \\ \frac{\partial C}{\partial z_{i1}^l}\frac{\partial z_{i1}^l}{\partial w_{ijN}^l} + \frac{\partial C}{\partial z_{i2}^l}\frac{\partial z_{i2}^l}{\partial w_{ijN}^l} + \cdots + \frac{\partial C}{\partial z_{iN}^l}\frac{\partial z_{iN}^l}{\partial w_{ijN}^l} \end{bmatrix}^T = \mathbf{d}_i^l \begin{bmatrix} \frac{\partial z_{i1}^l}{\partial w_{ij1}^l} & \frac{\partial z_{i1}^l}{\partial w_{ij2}^l} & \cdots & \frac{\partial z_{iN}^l}{\partial w_{ijN}^l} \\ \frac{\partial z_{i2}^l}{\partial w_{ij1}^l} & \frac{\partial z_{i2}^l}{\partial w_{ij2}^l} & \cdots & \frac{\partial z_{iN}^l}{\partial w_{ijN}^l} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{iN}^l}{\partial w_{ij1}^l} & \frac{\partial z_{iN}^l}{\partial w_{ij2}^l} & \cdots & \frac{\partial z_{iN}^l}{\partial w_{ijN}^l} \end{bmatrix} \\ &= \mathbf{d}_i^l \frac{\partial \mathbf{z}_i^l}{\partial \mathbf{w}_{ij}^l} = \mathbf{d}_i^l [\mathbf{a}_j^{l-1}]_*^T .\end{aligned} \tag{3.8}$$

From Eq. (3.2), we see that the last equality holds because

$$\frac{\partial \mathbf{z}_i^l}{\partial \mathbf{w}_{ij}^l} = \begin{bmatrix} a_{j1}^{l-1} & a_{jN}^{l-1} & \dots & a_{j2}^{l-1} \\ a_{j2}^{l-1} & a_{j1}^{l-1} & \dots & a_{j3}^{l-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & \dots & a_{j1}^{l-1} \end{bmatrix} = [\mathbf{a}_j^{l-1}]_*^T \,, \tag{3.9}$$

where $[\mathbf{a}_j^{l-1}]_*$ is the circulant matrix associated to $\mathbf{a}_j^{l-1}$.

We now show how to calculate $\mathbf{d}_i^l$, which requires backpropagation from the upper layer $l+1$ and is therefore more complicated. According to the chain rule, we have

$$\mathbf{d}_i^l \triangleq \frac{\partial C}{\partial \mathbf{z}_i^l} = \sum_{j=1}^{J^{l+1}} \frac{\partial C}{\partial \mathbf{z}_j^{l+1}} \frac{\partial \mathbf{z}_j^{l+1}}{\partial \mathbf{a}_i^l} \frac{\partial \mathbf{a}_i^l}{\partial \mathbf{z}_i^l} \,, \tag{3.10}$$

which involves vector-matrix-matrix multiplications. From Eq. (3.7), we see that the first term in the summation is equal to $\mathbf{d}_j^{l+1}$. From Eq. (3.2), and with some algebra, it can be shown that the middle term is equal to $[\mathbf{w}_{ji}^{l+1}]_* \in \mathbb{R}^{N \times N}$, and the last term is equal to the following $N \times N$ diagonal matrix:

$$\Phi(\mathbf{z}_i^l) \triangleq \frac{\partial \mathbf{a}_i^l}{\partial \mathbf{z}_i^l} = \text{diag}\left(\left[\dot{\phi}(z_{i1}^l),\ \dot{\phi}(z_{i1}^2),\ \dots,\ \dot{\phi}(z_{i1}^N)\right]\right) \,, \tag{3.11}$$

where $\dot{\phi}(\cdot)$ denotes the derivative of the activation function. From the above description, we see that the update of $\mathbf{d}_i^l$ depends on $\mathbf{d}_j^{l+1}$ and we can do this layer-by-layer in the backward direction starting from the last layer. In the output layer L, the local gradient

Algorithm 2 Backpropagation algorithm for updating deep cyclic-group networks

Input: Training inputs $\mathbf{x}$ and related targets $\hat{\mathbf{y}}$; activation function ϕ; cost function ϵ

Output: Parameters $\boldsymbol{\Theta}$ of weights and biases

1: **while** not converged **do**
2: **for each** minibatch **do**
3: Perform feedforward process with Eq. (3.2) to get $\mathbf{z}$
4: Perform backpropagation with Eqs. (3.10) and (3.12) to get local gradient vector $\mathbf{d}$ per neuron per layer
5: Update biases $\mathbf{b}$ with Eq. (3.7)
6: Update weights $\mathbf{w}$ with Eq. (3.8)
7: **end for**
8: **end while**

vector $\mathbf{d}^L$ can be computed by:

$$\begin{aligned}\mathbf{d}^L \triangleq \frac{\partial C}{\partial \mathbf{z}^L} &= \frac{\partial C}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{z}^L} \\ &= \frac{\partial C}{\partial \mathbf{y}} \frac{\partial \mathbf{a}^L}{\partial \mathbf{z}^L} = \frac{\partial C}{\partial \mathbf{y}} \Phi(\mathbf{z}^L)\,,\end{aligned} \tag{3.12}$$

where the last term involves computing the derivative of the cost function $\epsilon(\cdot, \cdot)$.

We summarize the aforementioned process in Algorithm 2. We note that the proposed learning algorithm can be applied to vector neurons with arbitrary dimensions N. If the complexity of CNN is $\mathcal{O}(C)$, then the complexity of DCGN is $\mathcal{O}(NC)$. In other words, the time complexity of DCGN is N times the complexity of CNN. However, if we view N as a constant, the complexity of DCGN and CNN becomes the same.

3.4 Weight initialization

Proper weight initialization can reduce the risk of gradient vanishing or exploding when the architecture of neural networks is deep. Below, we propose a weight initializa-

tion method for DCGN by extending the method proposed by [51] and [52] to DCGNs.

Initially, the weights are all scalars in the conventional CNNs but now all vectors in vector neurons. Since we know that variance is the expectation of the square minus square of the expectation, the equation of weight variance becomes:

$$\begin{aligned} \text{Var}(\mathbf{w}) &= \mathbb{E}\left[\mathbf{w}\mathbf{w}^T\right] - \left(\mathbb{E}\left[\mathbf{w}\right]\right)^2 \\ &= \mathbb{E}\left[|\mathbf{w}|^2\right] - \left(\mathbb{E}\left[\mathbf{w}\right]\right)^2, \end{aligned} \tag{3.13}$$

where $\mathbb{E}\left[\mathbf{w}\right]$ is 0 since the distribution of $\mathbf{w}$ is symmetric around 0. However, estimating $\mathbb{E}\left[|\mathbf{w}|^2\right]$ is not easy to accomplish, so the main idea is to make the output variance equal to the input variance for each neuron. In doing so, we need to estimate the variance of each kernel map $\mathbf{w}$, which can be calculated as [1, 27]:

$$\text{Var}(\mathbf{w}) = \mathbb{E}\left[|\mathbf{w}|^2\right] . \tag{3.14}$$

However, such an estimation is ill-defined for DCGN, since $\mathbf{w}$ is a vector not a scalar. That said, if we further assume w_1, w_2, …, $w_N \sim \mathcal{N}(0, \sigma^2)$ i.i.d., then $\mathbb{E}\left[|\mathbf{w}|^2\right] = \mathbb{E}\left[w_1^2 + w_2^2 + \ldots + w_N^2\right] = \mathbb{E}\left[w_1^2\right] + \mathbb{E}\left[w_2^2\right] + \ldots + \mathbb{E}\left[w_N^2\right] = N\sigma^2$. Thus we can rewrite Eq. (3.14) as:

$$\begin{aligned} \text{Var}(\mathbf{w}) &= \mathbb{E}\left[|\mathbf{w}|^2\right] \\ &= \int_{-\infty}^{\infty} x^2 f\left(x\right) dx \\ &= N\sigma^2, \end{aligned} \tag{3.15}$$

where $f(\cdot)$ stands for N-dimensional distribution with N degree-of-freedom and σ is an estimated parameter of the variance of $\mathbf{w}$. Then, if we use the initialization method proposed by [51], we have:

$$\text{Var}(\mathbf{w}) = \frac{2}{(n_{in} + n_{out})}, \tag{3.16}$$

where n_{in} and n_{out} denote the number of input and output units of that neuron, respectively. Accordingly, we have:

$$\sigma^2 = \frac{2}{N\,(n_{in} + n_{out})}. \tag{3.17}$$

On the other hand, if we use the initialization method proposed by [52], which is designed for networks that use rectified linear units (ReLUs) [53] as the activation function $\phi(\cdot)$, we would have $\text{Var}(\mathbf{w}) = 2/n_{in}$, and $\sigma^2 = \frac{2}{N\,n_{in}}$. Then the variance is written as:

$$\sigma = \sqrt{\frac{2}{N\,n_{in}}}. \tag{3.18}$$

3.5 Batch normalization

Batch normalization [10] is an important technique to stabilize and speed up the training process. The idea is to keep the input of each layer having zero mean and unit variance. The original version is designed for scalar neurons. While [1] have extended the method for DQNs, we present below how to extend it to general vector neurons with arbitrary dimension N. To ensure equal variance in the vector parts, we calculate the variance not

only between different training data instances but also between the N dimensions, leading to the following covariance matrix $\mathbf{V}$:

$$\mathbf{V} = [\text{Cov}(u_i, v_j)]_{\ i=1,\ldots,N\,,\ j=1,\ldots,N}\ , \tag{3.19}$$

where $\text{Cov}(\cdot)$ is the covariance function and u_i, v_j denotes entries of two input vectors $\mathbf{u}$ and $\mathbf{v}$ for that layer. Whitening approach [54], scaling the data by the square root of their variances along each of two-dimensional vectors, is used for complex-valued neurons. It can be completed by multiplying the inverse square root of the 2×2 covariance matrix. However, it is not easy to calculate the inverse square root of a matrix which dimension is more than 2×2. Following [1], Cholesky decomposition is firstly proposed to for covariance in the deep quaternion networks, which is applied to four-dimensional data. In our proposed work, we use Cholesky decomposition on the covariance matrix to learn the shift vector $\beta \in \mathbb{R}^N$ and the transformation matrix $\mathbf{\Gamma} \in \mathbb{R}^{N \times N}$. If the input vector $\tilde{\mathbf{x}}$ has been normalized to mean 0 and variance 1, then batch normalization is done by

$$\text{BN}(\tilde{\mathbf{x}}) = \mathbf{\Gamma}\tilde{\mathbf{x}} + \beta\,. \tag{3.20}$$

We note that $\mathbf{\Gamma}$ is a symmetric matrix, and that its diagonal can be initialized to $1/\sqrt{N}$ to obtain modulus of 1 for the variance of the normalized value. The other terms of $\mathbf{\Gamma}$ and all entries of β are initialized to 0.

3.6 Pooling layers

As shown in Figure 1.3, we can optionally use pooling layers after the convolutional layer. A pooling layer is another type of layers simplifying or summarizing the information from convolved feature maps. It is also called subsampling or downsampling which reduces the dimensionality of each map and produce the downsampled one but retains the important information. The pooling layer is also used if necessary. The pooling methods we employ are max pooling and average pooling, which are also commonly used in CNNs. What the difference is that we propose to calculate the magnitude of each mode-3 fiber in the feature maps, and preserve the one (i.e., a vector) with the largest magnitude for max pooling. For mean pooling, we calculate the average of each element among the mode-3 fibers to generate new three-way feature maps.

3.7 Applications to regression problems

DCGN can be applied to regression problems where we need to learn a nonlinear mapping between tensor inputs and tensor outputs. The inputs and outputs have the same mode-3 dimension, but not necessarily the same mode-1 and mode-2 dimensions. To fit the dimensionality of the output tensor, we need to use only one kernel map for the last convolutional layer of DCGN. For the activation function $\phi(\cdot)$, we can use rectified linear unit (ReLU) or leaky ReLU for all the layers except for the last layer, where we may want to use the sigmoid or tanh function. We can use squared error for the cost function $\epsilon(\mathbf{A}, \mathbf{B}) = \|\mathbf{A} - \mathbf{B}\|_2^2$, where $\mathbf{A}$ and $\mathbf{B}$ denote the groundtruth target and the output tensors

of DCGN, respectively. Such a network is shown in Figure 1.3.

3.8 Applications to classification problems

When DCGN is applied to classification problems, the input is a three-way tensor and the corresponding output is a vector for one-hot representation (i.e., the class labels). We can also use only one kernel map for the last convolutional layer, but this is not mandatory. Moreover, we would add fully-connected layers to the end of DCGN for learning the classifier. We usually use the sigmoid function as the activation function for the last layer of the fully-connected layers, so that we can use the cross entropy as the cost function: $\epsilon(\mathbf{a}, \mathbf{b}) = -\sum[a_i \ln b_i + (1 - a_i) \ln(1 - b_i)]$, where $\mathbf{a}$ and $\mathbf{b}$ are the groundtruth label vector and the predicted one, respectively. In such a case, DCGN performs the function of feature learning.

Chapter 4

Experiments

To validate the effectiveness of ABIPNN, we consider two regression problems that require learning a nonlinear mapping between tensor inputs and tensor outputs. The first is *multispectral image denoising*, which aims to recover the original multispectral image from a noisy input, with N set to 10. The second is *singing voice separation*, which aims to separate the singing voice and the accompaniment from a monaural audio mixture, with N set to 1, 3, 5, 7.

We intend to empirically compare the performance of the conventional DNNs with ABIPNN. In the first experiment, we will investigate their performance using a similar number of parameters. In the second one, the complexity analysis in Chapter 2 will also be verified. For both DNNs and ABIPNN, we use the mean-square error (MSE) as the objective function, and Adam [55] for gradient optimizations. For ABIPNN, we employ circular convolution as our bilinear product. The experiments are performed in Python on a personal computer equipped with an NVIDIA GeForce GTX 1080 Ti GPU and a

memory of 64 GB RAM.

To evaluate the effectiveness of DCGN, we test it on two regression problems. The first one is *color image inpainting*, which aims to reconstruct the deteriorated or lost parts of the original color image. The second one is *multispectral image denoising*, which aims to recover the original multi-band image from a noisy version. For the first problem, the input is a tensor with $N = 3$, the RGB channels. For the second problem, since a multispectral image is composed of multiple greyscale images shooting at different wavelengths (or bands), N is equal to the number of bands. We note that DCGN can deal with multispectral images of arbitrary bands, but DQN can only deal with 4-band multispectral images each time. Therefore, we intend to vary the value of N from 3 to 6 for the second experiment.

As baseline methods, we consider the conventional CNNs and the DQN model proposed by [1], using the code they shared. The goal is to verify the possible advantages of using the vector neurons over the scalar neurons, instead of pursuing high accuracy. Therefore, we opt for simpler model design and aim to compare the models fairly. For all the baseline models and our model, we use only three fully convolutional layers (without any pooling layers) and employ kernel maps with size 3×3. We used ReLU as the activation function, mean squared error (MSE) as the cost function, and Adam [55] for gradient optimization.

For conventional CNNs, we consider three variants of CNN with different number of kernels per layer. CNN_1 has the same number of kernels per layer (i.e., 24) as DCGN and DQN; CNN_2 has more kernels but the total number of parameters is similar to DCGN, and CNN_3 has even more kernels and roughly three times the total number of parameters

as DCGN. DCGN and DQN have more parameters than CNN_1 because their kernel maps are tensors.

The evaluation metrics for both experiments are peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM) [56]. SSIM is a measure of the similarity between two images (i.e., the groundtruth clean one and the recovered one) and its range is $[0, 1]$. Both PSNR and SSIM are the higher the better. We calculate them for each output result and report the average values. All the experiments are performed on a personal computer equipped with an NVIDIA GeForce GTX 1080Ti GPU and 64GB RAM. Our code is written in Python. For reproducible research, we will make the code of the experiments publicly available at https://github.com/zcfan-tw/vectorNNtoolbox.

4.1 Experiment on multispectral image denoising

Multispectral (a.k.a. hyperspectral) imaging systems are usually employed to solve broadband color problems. A multispectral image is composed of a collection of monospectral (or monochrome) images, each of which is captured with a specific wavelength. These monospectral images can be considered as different *bands* of the multispectral image. As different spectral bands may exhibit some mutual associations, we can leverage such associations to enhance the accuracy of image processing applications.

In multispectral image denoising, we are given a noisy version of a multispectral image with N bands, and are asked to recover the clean version (also N bands). In a recent work presented by Zhang *et al.* [29], different supervised multidimensional dictionary learning methods were evaluated on the Columbia multispectral image database [7] for the denoising task. By following their settings, we can compare the performance of our models with these prior arts. These methods include K-TSVD, K-SVD [57], 3D K-SVD [58], LRTA [59], DNMDL [60] and PARAFAC [61].

The database [7] contains 32 real-world scenes and each scene contains 31 monochrome images of size 512×512, captured by varying the wavelength of a camera from 400 nm to 700 nm with a step size of 10 nm. Following Zhang *et al.* [29], we consider images in the "chart and stuffed toy" scene, resize each image to 205×205 and take images of the last 10 bands (i.e., starting from 600 nm), making $N = 10$. Moreover, we divide each image into $8 \times 8 \times 10$ overlapping tensor patches with a hop size of one. We randomly take $10\,000$ tensor patches as the training data and the rest for testing. From the training data,

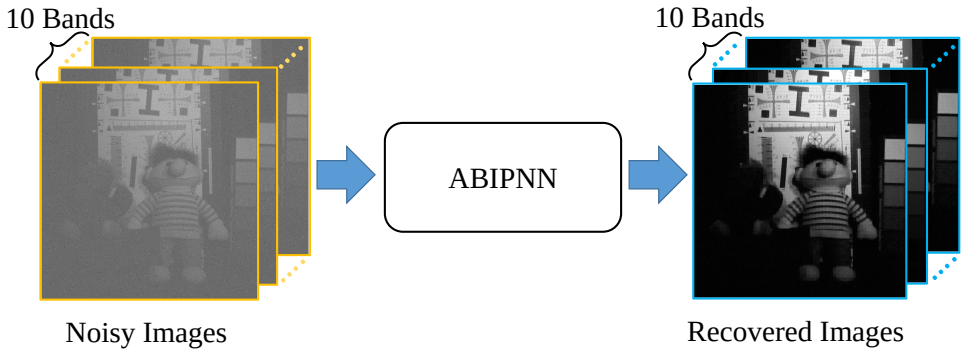


Figure 4.1: Illustration of multispectral image denoising using ABIPNN. Noisy images are contaminated by Gaussian noise. We take the last 10 bands for the experiments.

1000 tensor patches are held out as the validation data. The maximal number of training epochs is set to 3000. We also stop the training process if the validation MSE does not decrease for 100 epochs.

For the noise model, we randomly select a certain number of pixels from each band of an image and add to the pixels Gaussian noise with specific *sigma value*. We refer to the ratio of pixels corrupted per band as the *sparsity* level of the noise. In our experiments, we vary the sigma value from 100 to 200, and the sparsity level from 5% to 15%, to simulate different degrees of corruption. The goal is to recover the corrupted images, as illustrated in Fig. 4.1. As for the objective function in model training, we compute the MSE between the recovered and the original versions of the patches in the training set, across all the 10 bands. As in Zhang *et al.* [29], we measure the performance of denoising in terms of the peak signal-to-noise ratio (PSNR). We calculate the PSNR for each test patch and report the average result.

In our implementation, the ABIPNN consists of 3 hidden layers with 512 neurons in each layer (i.e., $R_2 = \cdots = R_{L-1} = 512$ in Fig. 1.1). Both the input and output layers

have $8 \times 8 = 64$ dimensions (i.e., $R_1 = R_L = 64$ in Fig. 1.1) and the topology is denoted as 64-512-512-512-64. In ABIPNN, each neuron represents a vector with size $N = 10$. Hence, for each patch the 10 bands are processed at the same time. The activation function is sigmoid with a learning rate of 5×10^{-4}. To compare the performance of ABIPNN with conventional DNNs using a similar number of parameters, we consider the following two variants of DNNs as the baselines. The first variant, *DNN-concat*, simply concatenates the 10 bands as a single vector to process them jointly, making $R_1 = R_L = 640$. There are 3 hidden layers with $1,450$ neurons in each layer, so the topology is 640-1450-1450-1450-640, where each neuron represents a scalar. The second variant, *DNN-parallel*, processes the 10 bands separately using 10 DNNs, each with a 64-512-512-512-64 topology. In other words, each DNN is trained for denoising a specific band.

Table 4.4 shows the experimental results. In the upper part of the table, we cite the PSNR values reported in [29], and in the lower part we report the results of our own implementation.[1] Because we follow their experimental settings, the PSNR values for the noisy images (i.e., before denoising) reported in [29] are close to what we observe in our implementation. Besides, Table 4.4 also shows that ABIPNN outperforms all the other methods, including the two DNN baselines, by a large margin across different values of sigma and sparsity. The PSNRs are improved from 12.10–20.92 dB to 29.55–33.92 dB. This result suggests the effectiveness of a neural network for this task. Fig. 4.11 demonstrates the original, noisy, and denoised versions of three images by using ABPINN.

[1]Each result we reported is the average of 10 simulations and the variance of each result is low. For example, when the sparsity is 5% and the sigma is 100, the variance of DNN-concat, DNN-paralled, and ABIPNN is 0.003, 0.014, and 0.011, respectively.

Table 4.1: PSNR (in dB) obtained by different methods for multispectral image denoising, under different sparsity and sigma values

Sparsity	5%	10%	15%	10%	10%
Sigma	100	100	100	150	200
Referenced from [29]					
Noisy Image	20.96	18.18	16.35	14.75	12.10
K-SVD [57]	22.73	22.60	22.49	22.38	22.20
3DK-SVD [58]	22.61	22.53	22.47	22.41	22.20
LRTA [59]	23.54	26.84	26.65	23.90	22.03
DNMDL [60]	24.07	23.73	25.16	17.89	16.83
PARAFAC [61]	27.07	26.86	26.72	26.13	25.24
K-TSVD [29]	27.19	26.98	26.79	26.18	25.44
Our Experiments					
Noisy Image	20.92	18.16	16.35	14.64	12.10
DNN-concat	25.06	24.80	24.93	24.59	24.03
DNN-parallel	30.18	28.88	28.06	27.17	25.88
ABIPNN	33.92	32.47	31.74	31.01	29.55

Among the three NN-based methods, DNN-concat performs the worst, and is even inferior to that of PARAFAC [61] and K-TSVD [29], two non-deep learning based methods. This suggests that concatenating inputs from different bands does not make it easy for an NN to learn the association between bands. In contrast, using multidimensional vector neurons, ABIPNN learns the relations between bands by computing the circular convolution of two vectors: one coming from a hidden node and the other coming from a weight tensor. The vector coming from a weight tensor can be regarded as a linear kernel that captures interactions across bands, which may contribute to enhanced results in denoising.

Figure 4.3(a) displays the changes in MSE values as a function of the training epochs in the training procedure for ABIPNN and DNN-concat. We find that the MSE values

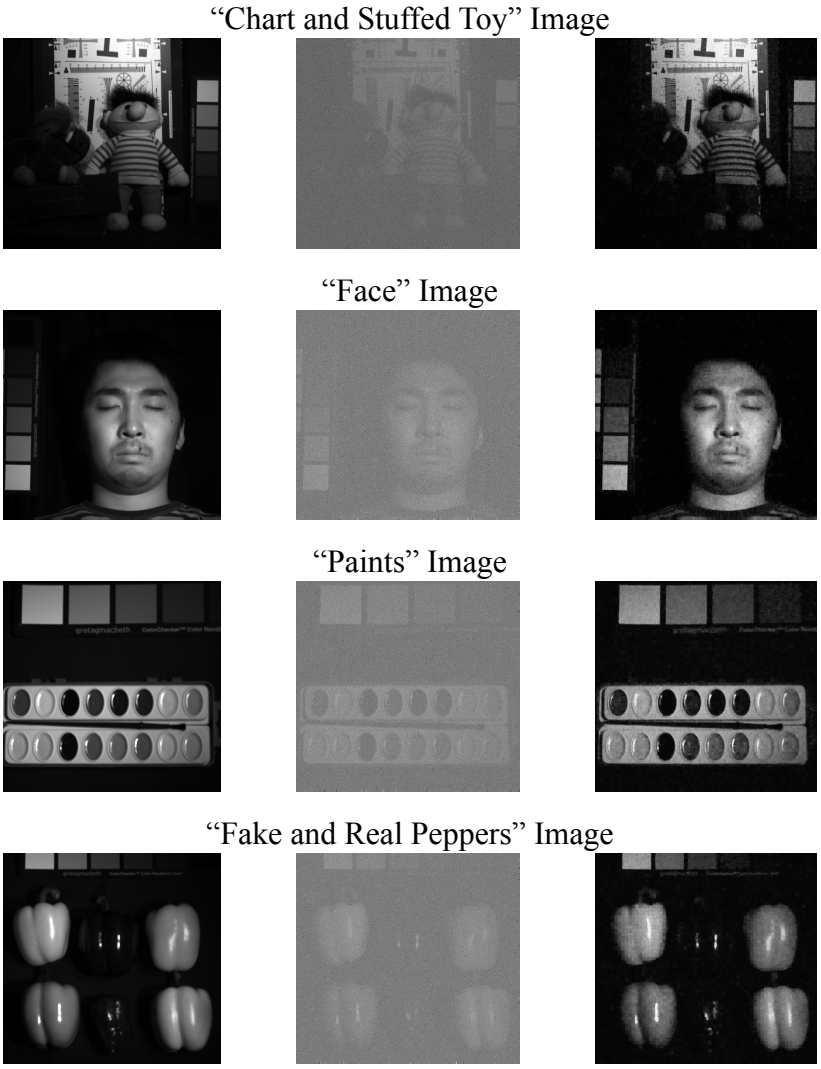


Figure 4.2: Denoised images at the 700 nm band using the proposed ABIPNN method. The sparsity of the noisy pixels is 10% and the sigma value of the additive Gaussian noise is set to 200.

converge to a certain value for both methods, but ABIPNN converges much faster. We conjecture that this is because the error propagation in ABIPNN is not only optimized for each dimension (i.e., band) but also between different dimensions. It is also known that gradient variance reduction helps achieve better convergence in stochastic gradient descent (SGD) [62, 63]. Fig. 4.3(b) shows the changes in gradient variance during SGD training. Here ABIPNN provides lower variance at convergence (after 2000 epochs).

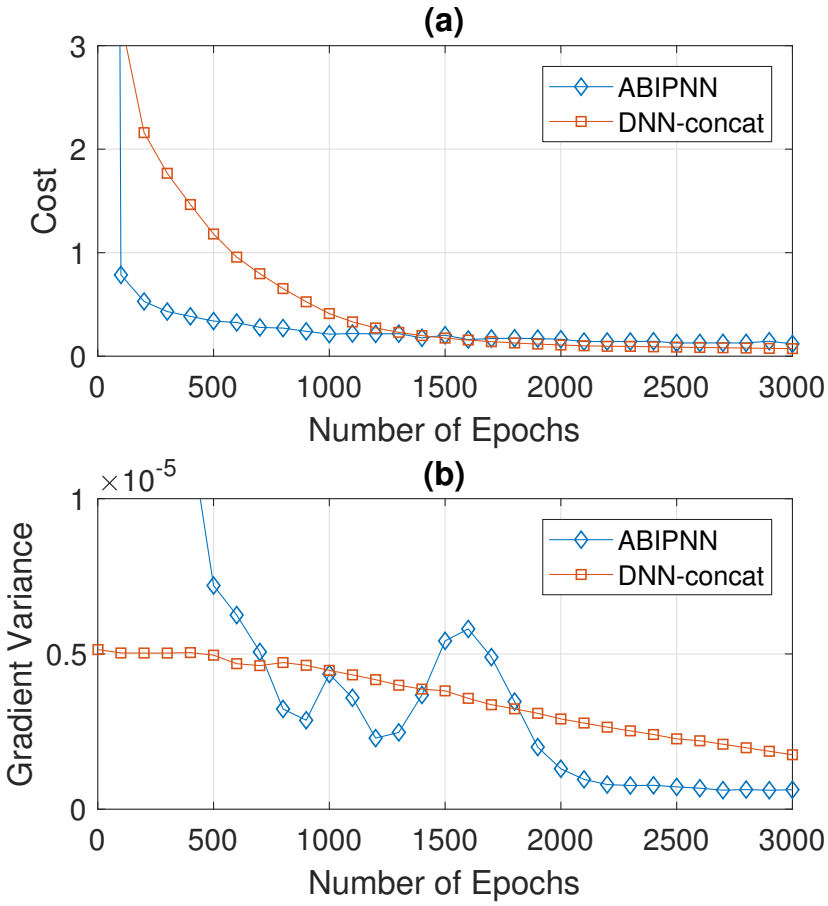


Figure 4.3: (a) Mean square error with Adam and (b) gradient variance with SGD in the training procedure of ABIPNN and DNN-concat, for the case when the sparsity of the noisy pixels is 5% and the sigma value is set to 100.

4.2 Experiment on singing voice separation

Separating the singing voice and the accompaniment from a monaural audio mixture is challenging, because there are more unknowns than equations. It is also a significant preprocessing step in audio signal analysis [64–74]. Several approaches have been proposed previously. Repeating musical structures have been exploited by the REpeating Pattern Extraction Technique [75]. Unsupervised methods such as robust principal component analysis [76–79] assume no labeled data are available and rely on assumptions on the characteristics of the sources for separation. For example, the spectrogram of the accompaniment part is assumed to have lower rank than that of the vocal part. If we are given the original sources of some audio mixtures, we can take these clean sources as the supervisory signal and employ supervised methods such as non-negative matrix factorization (NMF) [80] for source separation. Naturally, supervised methods usually outperform unsupervised methods, as the former can learn from the pairs of mixtures and sources.

Recently, NN-based methods have been introduced to this task [81,82], showing better result than non-NN based methods such as NMF. With the development of deep learning, Mass *et al.* [83] used recurrent neural networks (RNN) to create a clean voice. Huang *et al.* [81] then proposed deep RNN with discriminative training to reconstruct vocals from background music. Training multi-context networks [84, 85] with different inputs combined at layer level was proposed to improve audio separation performance. Deep clustering [86] is also used for music separation. Post-processing with a Wiener filter at the output of neural networks and data augmentation [87] have been proposed to separate

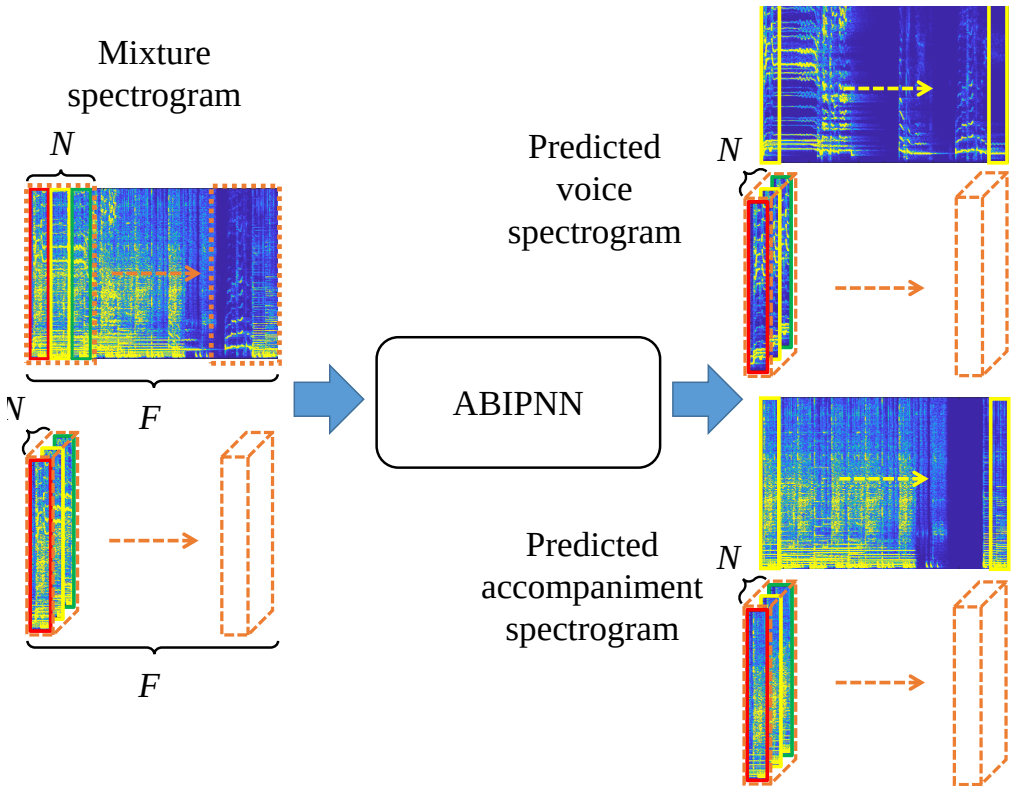


Figure 4.4: Illustration (best seen in color) of singing voice separation using ABIPNN, where T stands for the number of frames. We use red to indicate the previous frames, yellow for the current frames, and green for the subsequent frames. After training, we extract the second dimension (yellow) for soft-time frequency masking.

vocals and instruments. All of these deep learning techniques use multiple non-linear layers to learn the optimal hidden representations from data in a supervised setting.

This can be done by taking the spectrogram of an audio mixture as the input and requiring the network to reproduce the spectrograms of the corresponding two sources at the output. NN works better because they can learn a nonlinear mapping between the input and the outputs. A spectrogram is a 2-D time-frequency representation. It is computed by the short time Fourier transform (STFT), which divides a given time signal into short segments of equal length and then computes the Fourier transform separately on each short segment. We call each short segment a 'frame,' and the result of Fourier transform per frame as a 'spectrum.' The spectrogram considers only the magnitude part of STFT.

A naïve DNN approach for source separation, referred to as *DNN-simple* below, takes the spectrum of each frame of the mixture as input, and estimates the spectra of that frame for the vocal and the accompaniment parts respectively. This is done frame-by-frame, finally leading to the estimated (recovered) spectrograms $\tilde{\mathbf{Y}}_\mathbf{1}$ and $\tilde{\mathbf{Y}}_\mathbf{2}$ of the two sources. Then, we use the Weiner filter to compute the following soft time-frequency mask to smooth the source separation results.

$$\mathbf{m}(f) = \frac{|\tilde{\mathbf{Y}}_\mathbf{1}(f)|}{|\tilde{\mathbf{Y}}_\mathbf{1}(f)| + |\tilde{\mathbf{Y}}_\mathbf{2}(f)|}, \tag{4.1}$$

where $f = 1, 2, ..., F$ denotes different frequency bins. The estimated spectra $\tilde{\mathbf{s}}_\mathbf{1}$ and $\tilde{\mathbf{s}}_\mathbf{2}$, respectively corresponding to vocals and accompaniments, are produced by:

$$\begin{aligned} \tilde{\mathbf{s}}_\mathbf{1}(f) &= \mathbf{m}(f)\mathbf{z}(f), \\ \tilde{\mathbf{s}}_\mathbf{2}(f) &= (1 - \mathbf{m}(f))\mathbf{z}(f), \end{aligned} \tag{4.2}$$

where $\mathbf{z}(f)$ is the magnitude spectra of the input mixture. The estimated spectra $\tilde{\mathbf{s}}_\mathbf{1}$ and $\tilde{\mathbf{s}}_\mathbf{2}$ are finally transformed back to the time-domain by the inverse short time Fourier transform (ISTFT), assuming that the mixture and the two sources share the same phase. Parameters of the NN are learned by using the MSE between the estimated spectra and the groundtruth source spectra.

We can improve the performance of DNN-simple by adding the *temporal context* of each frame to the input [85]. Specifically, in addition to the current frame, we add the spectra of the previous-f and subsequent-f frames to compose a real-valued matrix. For

a conventional NN, we can take the vectorized version of the matrix (which amounts to concatenating the spectra of these $2f + 1$ frames) as the input. We refer to this method as *DNN-concat*. Alternatively, we can view the $2f + 1$ frames as different dimensions and use a vector NN to model the interaction between different frames. Please see Fig. 4.4 for an illustration.

Because ABIPNN can deal with input of arbitrary dimensions, in principle f can take any values. In this experiment, we compare the performance of ABIPNN with the classic vector product neural network (VPNN) [21] (i.e., where $f = 1$ and only two neighboring frames are considered), and the conventional DNNs (i.e., where $f = 1$–3). The major difference between ABIPNN (with $N = 3$) and VPNN is that the former uses circular convolutions.

In our experiments, we use the Demixing Secret Database (DSD100), which was used in the Signal Separation Evaluation Campaign (SiSEC) in 2016 [88, 89]. It is made up of 100 full-track professionally-produced music recordings of different styles. It can be used for training and testing for source separation algorithms, because it includes both the stereophonic mixtures and the original stereo sources. The database is divided into a development set and a test set by the organizers of SiSEC 2016, each consisting of 50 songs. The duration of the songs ranges from 2'22" to 7'20", and the average duration is 4'10". All the songs are sampled at a sampling rate of $44\,100$ Hz. To reduce computational cost, all songs are downsampled to 8000 Hz. For each song, we compute STFT with a 1024-point window and a 512-point hop size. We set the activation function to the rectified linear unit (ReLU) and the learning rate to 3×10^{-5} with exponentially decay reduced by

Table 4.2: Comparison of SDR, SIR, SAR values and computation time (seconds per epoch) obtained by different methods for singing voice separation over the DSD100 data set, under different values of N and number of neurons per layer.

Model	N	Neurons per layer	Number of params	Time (second per epoch)	SDR		SIR		SAR	
					Vocal	Accomp.	Vocal	Accomp.	Vocal	Accomp.
DNN-simple	1	513	1.31M	4.2	4.37	9.98	8.38	12.89	5.81	13.89
DNN-concat$_1$	3	1539	8.68M	7.1	4.63	10.25	8.29	13.34	6.17	14.14
VPNN [30]	3	513	3.95M	9.1	4.64	10.17	8.37	12.97	6.59	14.57
ABIPNN$_1$	3	513	3.95M	9.2	4.67	10.08	8.39	12.76	6.67	14.24
DNN-concat$_2$	5	2565	22.36M	14.1	4.68	10.19	8.86	12.53	6.78	15.07
ABIPNN$_2$	5	513	6.59M	17.5	4.97	10.50	9.45	13.63	6.88	14.78
DNN-concat$_3$	7	3591	42.37M	23.3	4.73	10.30	**10.67**	12.45	6.28	**15.19**
ABIPNN$_3$	7	513	9.22M	30.6	**5.13**	**10.70**	9.41	**13.93**	**7.20**	14.87
DCGN	7	—	7.76K	96.7	5.01	10.18	9.23	13.01	6.83	13.65
CNN	7	—	7.87K	51.2	4.81	9.85	8.95	12.43	6.74	12.71

10% every 300 epochs.

Then we map each t-f unit of the magnitude spectrum to an N-dimensional vector to serve as the input to ABIPNN. Among the training frames, 5% of the tensor frames are randomly picked as the validation data. The training epoch is set to 1000 and the training process is stopped if the MSE of the validation set does not decrease for 20 epochs. The performance is measured in terms of source to distortion ratio (SDR), source to interferences ratio (SIR), and source to artifact ratio (SAR), as calculated by the Source Separation (BSS) Eval Toolbox v3.0 [90].

Table 4.2 shows some details of the evaluated methods and their results. Each model has three hidden layers. In order to obtain the same internal dimensionality corresponding to real-valued neurons of DNN-concat and vector-valued neurons of ABIPNN, we use N times more neurons per layer for DNN-concat than ABIPNN.

In Table 4.2, the two vector NNs (i.e., VPNN and ABIPNN) indeed outperform the conventional DNN methods (i.e., DNN-simple and DNN-concat), demonstrating the effectiveness of considering the interactions between frames. DNN-concat$_1$, VPNN and ABIPNN all outperform DNN-simple by a great margin in SDR. And, in terms of SAR, we see VPNN and ABIPNN perform much better than DNN-concat$_1$, despite that DNN-concat$_1$ has more parameters. This shows that vector neurons can take better advantage of the information provided by the temporal context.

Furthermore, we also compare the computation times of DNN-concat, VPNN and ABIPNN with an NVIDIA 1080 Ti GPU. It is obvious that the computation time of ABIPNN is comparable with DNN-concat given the same value of N.

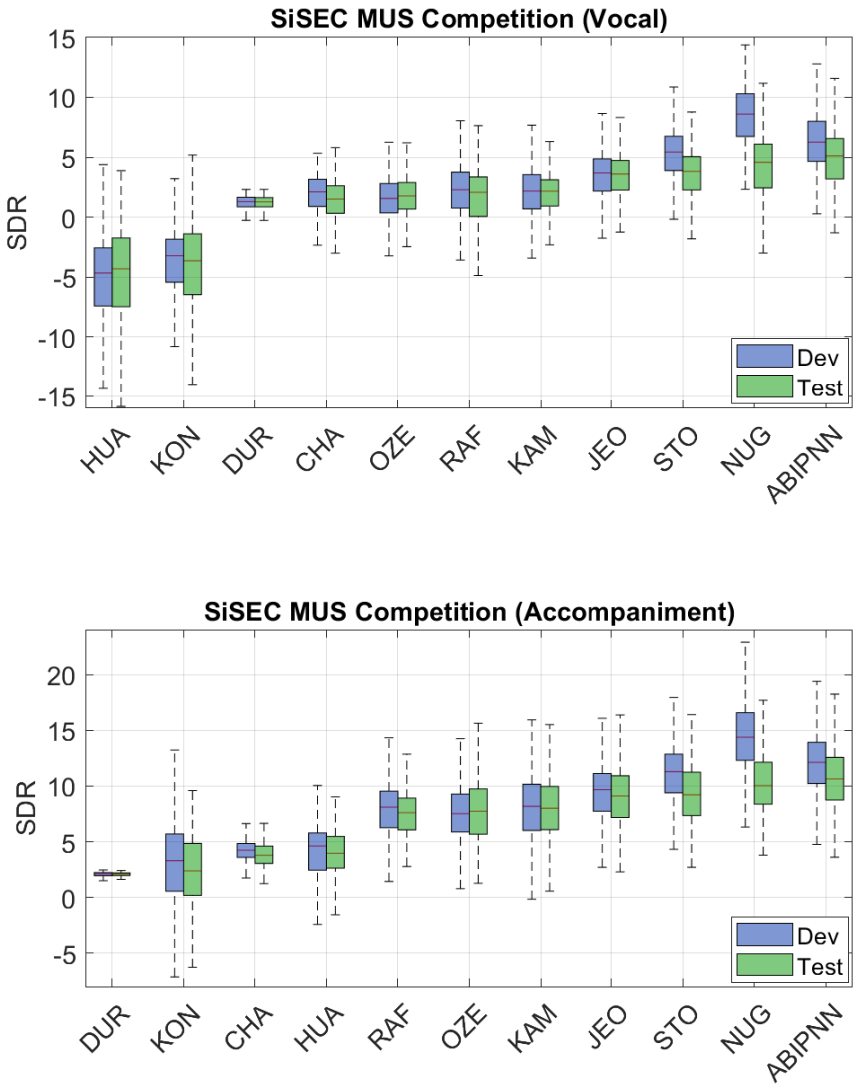


Figure 4.5: Vocal and accompaniment results (in SDR) for the development part and test part of the DSD100 dataset. The methods are sorted by the median SDR for the test part. For the result of ABIPNN, the value of N is set to 7. Please note that we only consider methods that did not use any data augmentation here, for fair comparison.

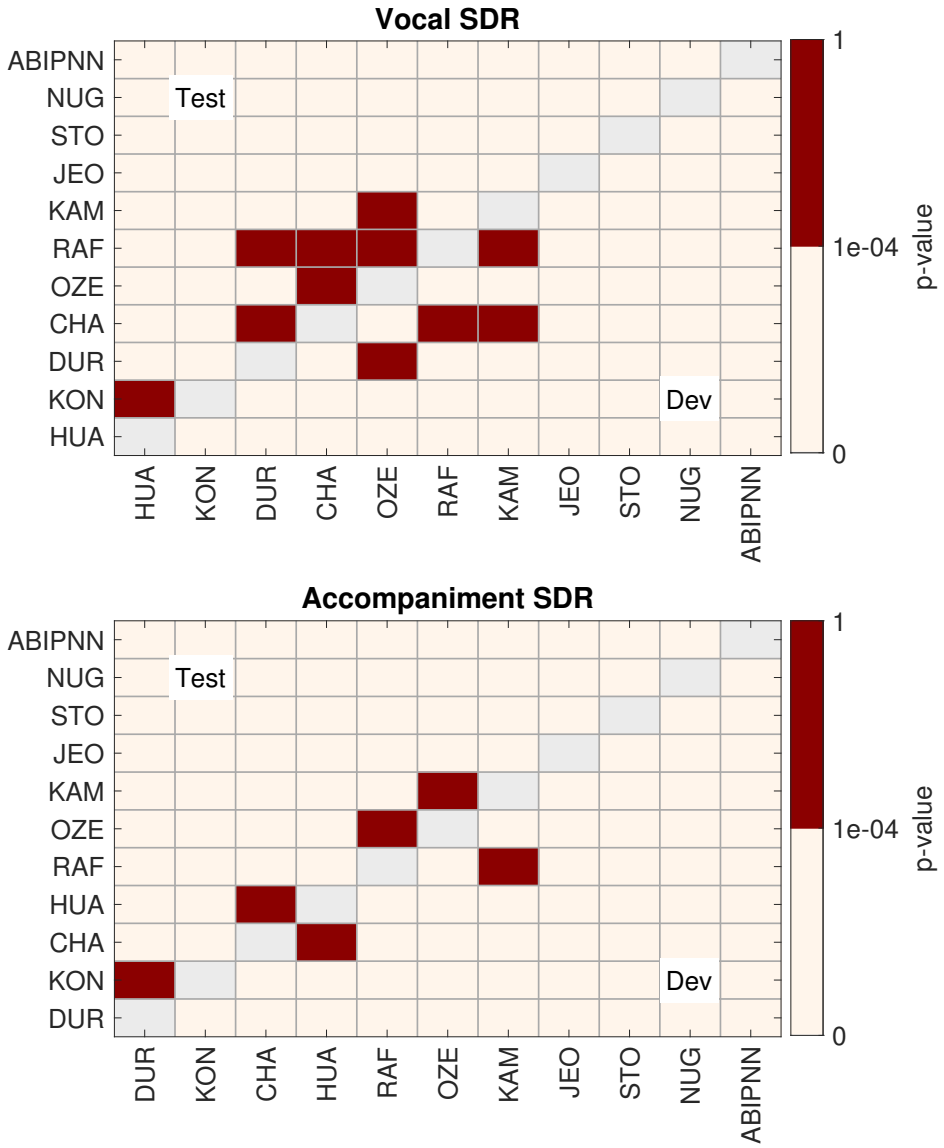


Figure 4.6: Wilcoxon signed-rank test results for SDR vocals and accompaniments. The upper triangle represents the comparison of the test set and the lower triangle is for the development part. For the result of ABIPNN, the value of N is set to 7. Values of $p-\text{value} > 1e-04$ indicate no significant difference between the two group results.

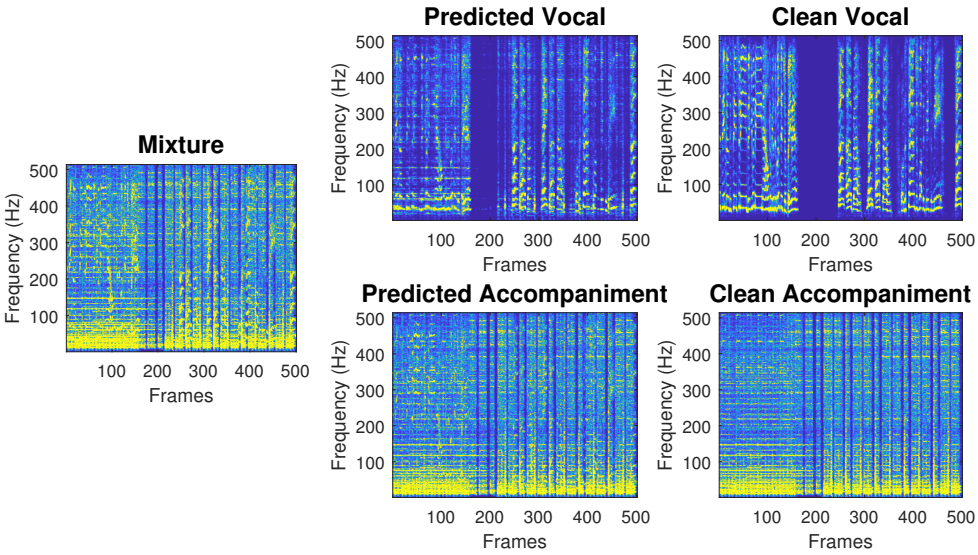
While we can only use $N = 3$ for VPNN, we can use larger N for ABIPNN. From Table 4.2, we see that the performance difference between ABIPNN and DNN-concat increases as N goes larger, despite that the latter architecture has more total parameters. With $N = 7$, the SDR obtained by ABIPNN reaches 5.13 dB for the vocal part, which outperforms DNN-simple by 0.76 dB. As can be seen later from Fig. 4.5, such a performance gap is remarkable.

On the other hand, we also compare the performance between DCGN and CNN by using the similar number of parameters when $N = 7$. Both of them consist of three fully convolutional layers (without any pooling layers) and the number of kernel maps are 24 for DCGN and 61 for CNN, respectively. As shown in Table 4.2, the performances of DCGN and CNN are comparable to ABIPNN by using less number parameters[2]. DCGN performs better than CNN although the computation time of DCGN is longer than CNN.

In Fig. 4.5, we compare the median SDR of the vocal and accompaniment parts of ABIPNN with the methods that have been evaluated for SiSEC 2016 [89], including NUG [91], STO [92], OZE [93], KON [94], KAM [95], JEO [96], HUA [76], DUR [65], CHA [97], and RAF [75]. Among them, CHA is based on CNN, and KON, STO, NUG are based on DNNs. For fair comparison, we only select those submissions that are not trained with augmentation data. Moreover, we show the vocal and accompaniment SDR here instead of other metrics, for saving space (following [89]) and because SDR is usually considered to be more important than the other metrics. It is also a convention in SiSEC

[2]We also try to increase of the number of parameters of DCGN, but it is hard to get a complete training procedure due to the cost of computation time. We will complement the experiments of using DCGN with more parameters in the future when we make the training more scalable to higher values of N.

(a) 009 - Bobby Nobody - Stitch Up



(b) 039 - Swinging Steaks - Lost My Way

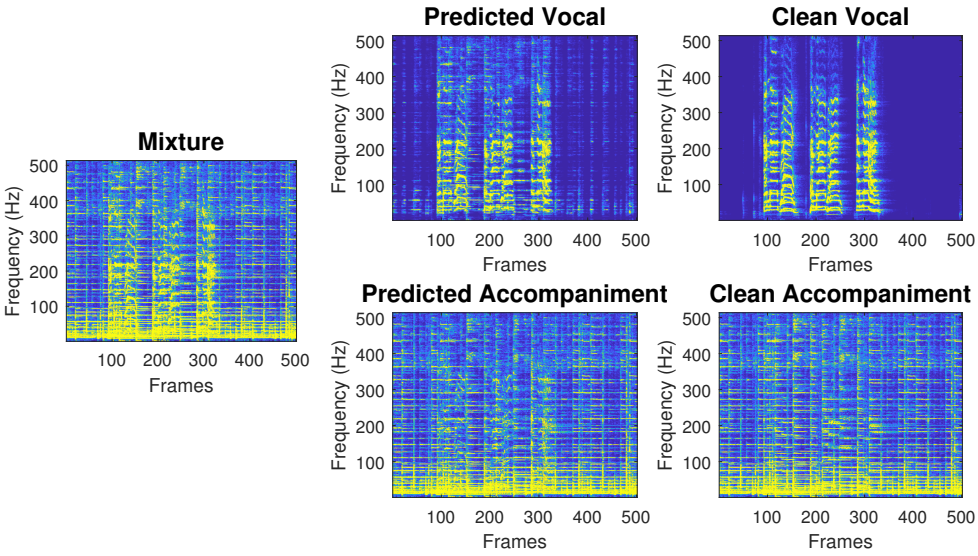


Figure 4.7: Examples of singing voice separation employing ABIPNN on the test part of the DSD100 dataset.

to show the result for both the development and the test sets. From Fig. 4.5, we see that ABIPNN outperforms all the other methods. From the report [89], our results of ABIPNN are comparable to UHL [98], which applies data augmentation and a complicated NN architecture.

In order to show the effect of different methods on SDR values, we follow [99] and apply the Wilcoxon signed-rank test (two-tailed and Bonferroni corrected) for pairwise comparisons. From Fig. 4.6, we can see that the group of results from ABIPNN has significant differences over other methods.

Finally, Fig. 4.7 shows the spectrograms of the input mixture, the separation results by ABIPNN, and the original sources, for two songs randomly picked from the test set of DSD100. We see that the separation results (marked as "predicted vocal" or "predicted accompaniment") resemble the original sources.

4.3 Experiment on color image inpainting

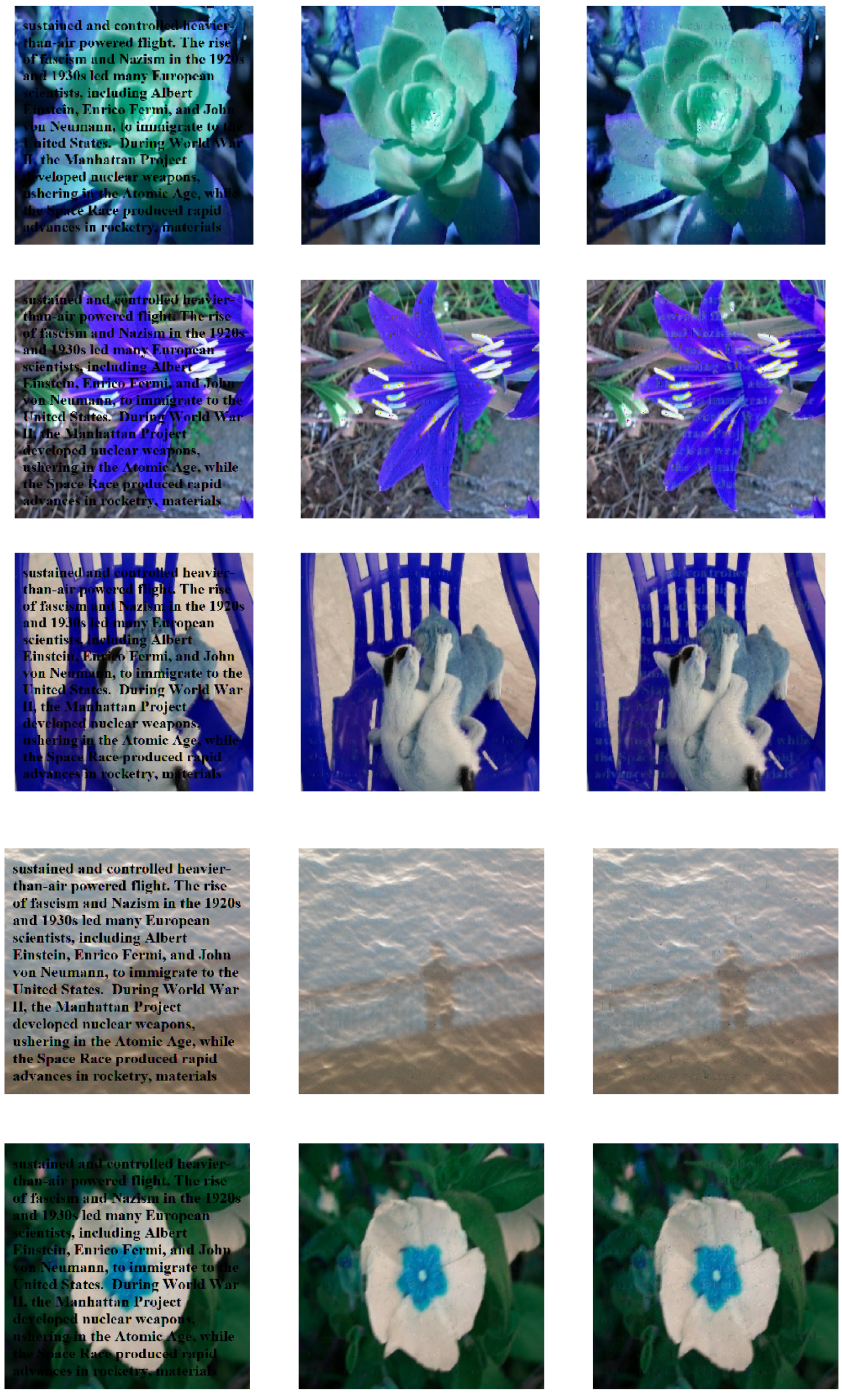
Image inpainting is a task that aims to recover corrupted or missing pixels in an image [100–103], and it is essential in computer vision. Many applications, such as photo editing, image-based rendering and computational photography are based on inpainting algorithms to reconstruct the missing parts. The main challenge of image inpainting can be divided into two categories. The first one is to manipulate an existing image. Some image details or larger regions of a given image should is removed and then we have to create a plausible missing part to complete the image. The target is not to recover a true image but the filled part should looks realistic. The second one is to consider that the given image is locally corrupted. The corruption is small, and sometimes possibly all over the image. The goal is to recover the missing part that are coherent with original given ones. Figure 4.8(a) shows three example images that need to be inpainted. We have to remove the imposed text and fill pixel values that are consistent with the surrounding context.

Many methods have been proposed to process image inpainting. Traditional diffusion-based methods [100, 104, 105] propagate the local information, such as edges and gradients, from the background regions to the missing pixels. Patch-based [106, 107] methods perform by searching for relevant patches from the non-hole regions or other source images of the image in an iterative fashion. Recently, deep learning based methods emerge as a promising scenario for image inpainting. Deep neural networks are proposed for image denoising and inpainting [101, 108]. Then CNNs are used to recover the local corruption [28, 109]. Context encoders proposed by [110] train deep neural networks for

inpainting large holes by using the objective function comprised of reconstruction loss and generative adversarial loss, based on generative adversarial network (GAN). Then, [103] propose both global and local discriminators as adversarial losses to improve the performance of [110]. More recently, contextual attention layer with GAN is proposed by [111] to attend on relevant feature patches for image inpainting.

From above descriptions, more and more inpainting approaches are presented by using deep learning. As a result, we evaluate our proposed method on the color inpainting task ($N = 3$), based on a given locally corrupted image and then recover it to the original one in a supervised setting. We use the images from the LabelMe dataset [112] for this evaluation. The dataset is split into two non-overlapping sets: the training set contains 2,920 images, and the test set 1,133 images. To prevent overfitting, we randomly select from the training set 150 images as the validation set. To reduce the training time, we resize all the images from 2,592×1,944 to 512×512. We contaminate all the images by the same text with variant fonts [109], using the contaminated ones as the model input and the clean ones as the target output. For all the models, we set the maximal number of training epochs to 100, and stop the training process when the validation MSE does not decrease for consecutive 10 epochs. Since the color images only have three dimensions (i.e., RGB), we follow the approach of [1] and set the fourth dimension to all zeros for DQN to provide a fair comparison with the quaternion algebra.

Table 4.3 shows that DCGN and DQN perform generally better than the conventional CNNs, suggesting the benefit of vector neuron learning. While the results of DCGN and DQN are comparable, the result of DCGN is significantly better ($p-$value < 0.01) than



(a) Noisy (b) DCGN (c) CNN_2

Figure 4.8: Examples of three contaminated images, and the result of DCGN and a conventional CNN model (CNN_2) for image inpainting. DCGN performs better than CNN_2 in removing the imposed text, as also shown in Table 4.3.

Table 4.3: The PSNR (in dB) and SSIM obtained by different methods for color image inpainting. 'Kernels' stands for the number of neurons per layer, and 'Parms' the total number of parameters. DQN is based on [1].

Method	N	**Kernels**	**Param.**	**PSNR**	**SSIM**
Input noise	—	—	—	14.23	0.65
CNN_1	3	24	11.7K	29.50	0.90
CNN_2	3	41	32.6K	30.11	0.91
CNN_3	3	72	97.4K	30.86	0.93
DCGN	3	24	32.4K	**31.72**	0.94
DQN [1]	4	24	43.9K	31.62	0.94

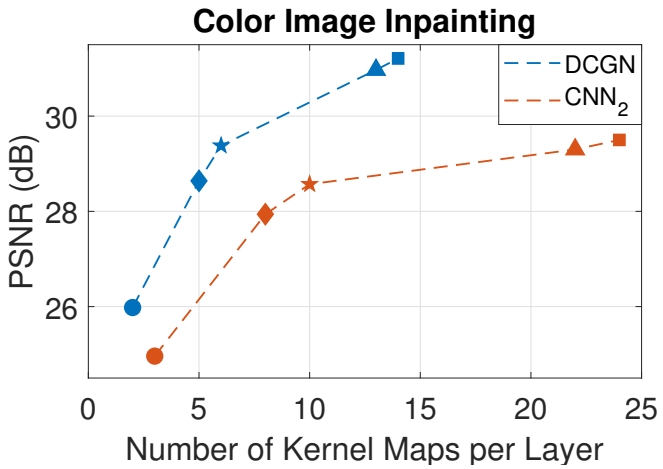


Figure 4.9: The PSNR (in dB) of DCGN and CNN for image inpainting, using different number of kernel maps per layer. We use the same mark to indicate the cases where DCGN and CNN have the same total number of parameters.

even CNN_3 under the paired t-test for both metrics. Moreover, we note that DCGN has fewer total parameters than DQN, yet DCGN obtains the highest mean PSNR 31.72 dB.

Some inpainting results of DCGN and CNN2 are shown in Figures 4.8. Both methods can remove the imposed text, but DCGN performs slightly better. Figure 4.9 further shows that the result of both DCGN and CNN increases as a function of the number of kernels per layer, but DCGN consistently outperforms CNN.

Finally, we show in Figure 4.10 what DCGN might learn for the inpainting task. For this investigation, we use a three-layer DCGN with only five neurons per layer for simplicity. Figure 4.10 shows the output (i.e., feature map) of each neuron for a given image, as well as the direct sum of these outputs per layer. Because the feature maps are all tensors with $N = 3$, they can be viewed as color images. We see that DCGN learns to separate the original image from the imposed text, and that the summation of the feature maps becomes more and more noise-free layer by layer.

4.4 Multispectral image denoising using DCGN

A multispectral (or hyperspectral) image is composed of a collection of monospectral (greyscale) images, each captured with a specific wavelength [113]. These monospectral images can be thought of as in different *bands* of the multispectral image. Since some associations may exist among different bands, we can improve the performance of image processing applications by leveraging such associations [60].

In multispectral image denoising [29, 114], we are given a noisy version of a multi-

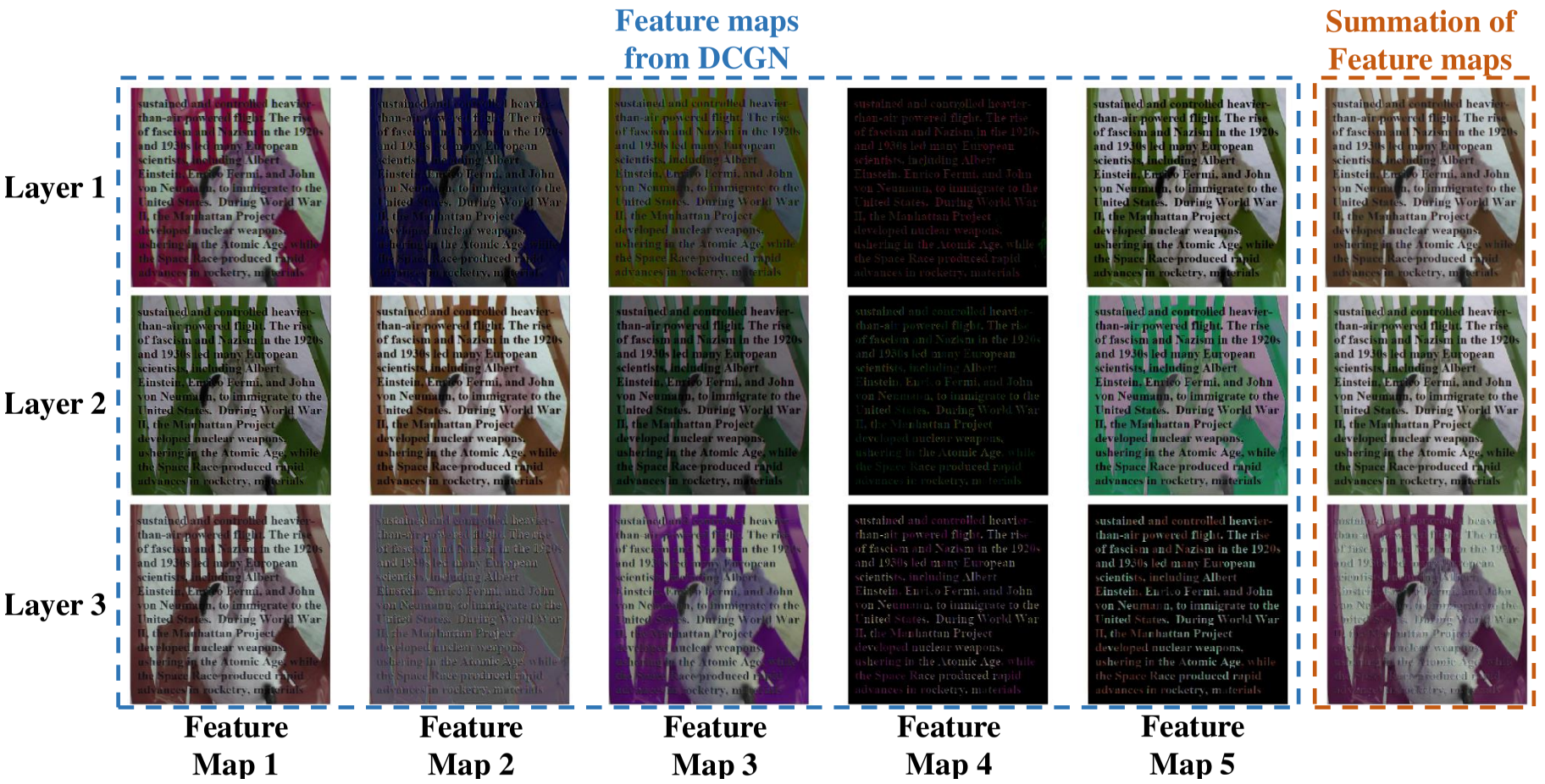


Figure 4.10: Illustration of what DCGN learns. For this illustration, we use five kernels for each of the three layers in DCGN. We show the output of each neuron on the left hand side, and the summation of them on the rightmost column. We can see that DCGN learns to separate the image from the occluding text, and that the summation of the feature maps becomes more and more noise-free layer by layer. We note that the final output of DCGN would actually be a weighted sum of the feature maps.

spectral image with N bands, and are asked to restore the clean version (also N bands). We use the Columbia multispectral image database [7] for this evaluation. It contains 32 real-world scenes and each scene is made up of 31 monochrome images of size 512×512, captured by varying the wavelength of a camera from 400nm to 700nm with a step size of 10nm. We randomly select 16 scenes for the training set, and the rest for the test set. We pick images of the last 3–6 bands to make $N \in [3, 6]$. Moreover, we divide each image into $32 \times 32 \times N$ overlapping tensor patches with hop size being one. Then, we randomly select approximate 20,000 tensor patches as the training data, 1,000 of which are held out as the validation data to prevent overfitting. For each tensor patch, all the N bands are processed at the same time. We set the maximal number of training epochs to 30 and stop the training process if the validation MSE does not decrease for 5 consecutive epochs.

To create a noisy image, we randomly pick a certain ratio of pixels from each band to add Gaussian noise with specific *sigma* value. We refer to this ratio as the *sparsity* below. We vary the sparsity from 5% to 15% and sigma from 100 to 200 to simulate different degrees of corruption. In model training, the MSE is computed between the restored and the original versions of the patches across all the N bands. We vary N from 3 to 6 for DCGN, and set $N = 4$ for CNNs and DQN.

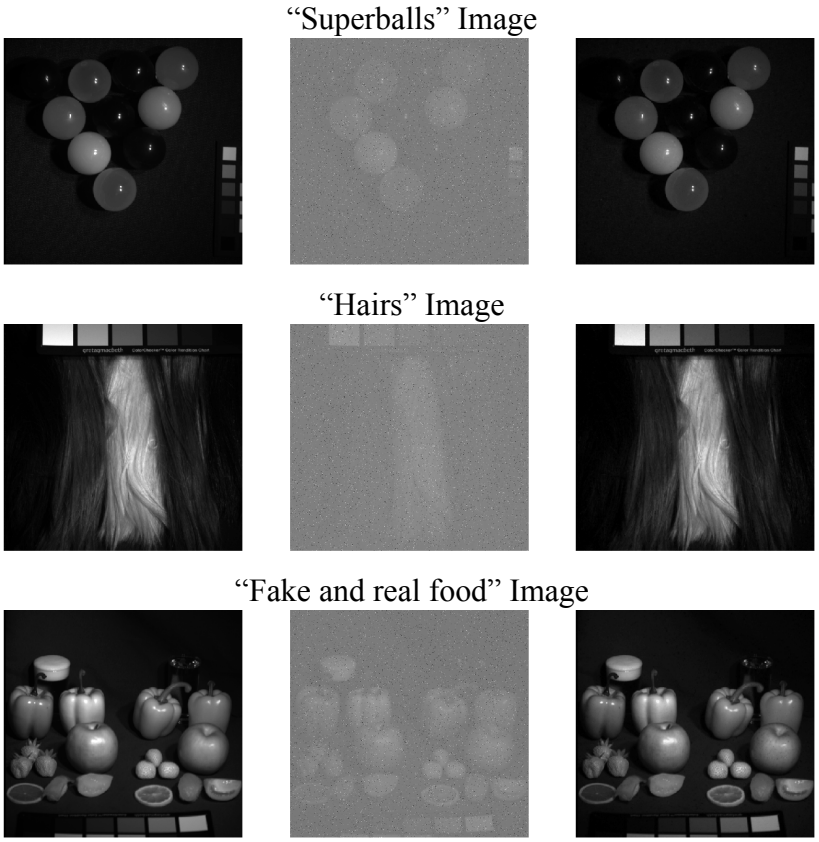
Table 4.4 shows the result of multispectral image denoising. The following observations can be made. First, comparing the result of the two CNNs and DCGN ($N = 4$), we see again DCGN outperforms CNNs for a regression task. The performance difference between DCGN and CNN_2 is significant (p-value<0.01) under the t-test for the first four evaluation scenarios. The fifth evaluation scenario (i.e., 10% sparsity with a sigma of 200)

is the most noisy scenario but DCGN still performs better. Figure 4.11 demonstrates the denoised result of DCGN.

Second, Table 4.4 indicates that the performance of DQN [1] is also much better than CNN_1. However, it is only comparable to CNN_2, which uses more kernels per layer but the same overall total number of parameters. Comparing the result of DCGN ($N = 4$) and DQN, it seems DCGN is more effective, possibly owing to the use of cyclic group algebras.

Third, comparing the result of different DCGN models in Table 4.4, we see that the performance in PSNR improves as a function of N, except for the two noisier scenarios. This nicely demonstrates the importance of having a vector neural learning model that can accommodate arbitrary data dimensions. The dependency across different data dimensions may increase when we have more dimensions, and DCGN can exploit such dependency to improve the result of a machine learning task.

Finally, we also employ ABIPNN on this task by using the similar number of parameters when $N = 6$. The architecture of ABIPNN is made up of three hidden layers and each hidden layer consists 30 neurons. As shown in Table 4.2, the performance of DCGN outperforms ABIPNN when $N = 6$. It shows that DCGN has an better ability to learn features by using less parameters when comparing to ABIPNN.



(a) Original (b) Noisy (c) Denoised

Figure 4.11: Examples of three noisy images (at 700nm band) and the multispectral image denoising result by DCGN with $N = 4$. The sparsity of the noisy pixels is 10% and the sigma value of the additive Gaussian noise is set to 200.

Table 4.4: PSNR (in dB) and SSIM obtained by different methods for multispectral image denoising under different sparsity (in %) and sigma values. We can use different number of bands for denoising, leading to different data dimensions N.

Method	N	**Kernels**	**Param.**	**(5%, 100)**		**(10%, 100)**		**(15%, 100)**		**(10%, 150)**		**(10%, 200)**	
				PSNR	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**
Input noise	—	—	—	21.23	0.72	18.35	0.59	16.68	0.50	14.81	0.53	12.30	0.52
CNN_1	4	24	12.2K	42.07	0.97	40.62	0.95	39.06	0.94	38.97	0.95	39.24	0.94
CNN_2	4	48	45.1K	45.07	0.98	42.18	0.97	41.02	0.96	42.07	0.97	42.01	0.97
CNN_3	4	83	130.2K	43.82	0.97	41.71	0.97	41.57	0.97	43.04	0.97	41.49	0.96
DCGN ($N = 3$)	3	24	32.4K	48.24	0.99	46.00	0.98	45.02	0.98	46.97	0.98	44.93	0.98
DCGN ($N = 4$)	4	24	43.3K	48.61	0.99	46.58	0.98	45.42	0.98	46.02	0.98	43.62	0.97
DCGN ($N = 5$)	5	24	54.0K	48.96	0.99	46.97	0.98	45.55	0.98	44.60	0.98	45.03	0.98
DCGN ($N = 6$)	6	24	64.8K	50.26	0.99	47.19	0.98	45.63	0.98	45.45	0.98	46.31	0.98
ABIPNN	6	–	65.3K	46.12	0.96	45.65	0.95	44.23	0.95	43.86	0.94	43.01	0.94
DQN [1]	4	24	43.5K	45.15	0.98	43.20	0.97	42.29	0.97	42.61	0.97	41.57	0.97

Chapter 5

Conclusions and Discussions

In this dissertation, we proposes a novel vector-valued neuron which employs arbitrary bilinear products for feedforward and backpropagation. The proposed architecture generalizes and extends all existing vector-valued neurons and is useful for datasets where each training sample is a multidimensional vector. Through bilinear products, the vector neural network captures the associations among different entries in the same position in each vector. The model can be trained efficiently by using vector error backpropagation through the Adam algorithm. Experimental results on multispectral denoising and singing voice separation show that our proposed model performs better than conventional NNs by using less number of parameters. On the other hand, we have also presented a new type of CNN architecture consisting of N-dimensional vector-valued neurons endowed with the cyclic group algebra, where N can be an arbitrary positive integer. In the convolution procedure, scalar multiplications in conventional CNNs are changed to vector multiplications through circulant matrices. During the feedforward and backpropagation, the

circular convolution is formulated as a vector-matrix multiplication with a circulant matrix. The learning process is performed with a backpropagation algorithm and optimized through Adam. Experimental results on color image inpainting and multispectral denoising show that the proposed model exhibit better performance than conventional CNNs and deep quaternion networks.

Future work involves three directions. First, we observe that the training time of the proposed network grows quadratically as a function of the dimensionality N. We make the training more scalable to higher values of N. Next, we would like to test it on classification problems, and datasets with even larger dimensions, such as EEG-based emotion recognition. Finally, since our proposed architecture could operates with arbitrary biliner products, it is similar to the concept of object orientation, where each neuron could be seen an object encapsulating all bilinear products and operating with other neurons. In the future, we will extend the proposed architecture to graph neural networks to investigate more powerful models.

Appendix A

Bilinear Products for ABIPNN

In Chapter 2, we use circular convolution as the bilinear product in ABIPNN and briefly derive the results of matrices $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ and $\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet$ used in the backpropagation process. In this section, we will derive some more bilinear products in detail. The feed-forward connection is still described between layer $l-1$ and layer l.

A.1 Vector product

Here $\mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1}$ stands for the usual vector product, $\mathbf{w}_{ij}^l = \left[w_{ij1}^l, w_{ij2}^l, w_{ij3}^l\right]^T \in \mathbb{R}^3$, and the input $\mathbf{a}_j^{l-1} = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, a_{j3}^{l-1}\right]^T \in \mathbb{R}^3$ are all three-dimensional vectors $(N = 3)$.

The matrix representation of $\mathbf{w}_{ij}^{l} \bullet \mathbf{a}_{j}^{l-1}$ is:

$$\left[\mathbf{w}_{ij}^{l}\right]_{\bullet} = \begin{bmatrix} 0 & -w_{ij3}^{l} & w_{ij2}^{l} \\ w_{ij3}^{l} & 0 & -w_{ij1}^{l} \\ -w_{ij2}^{l} & w_{ij1}^{l} & 0 \end{bmatrix}, \tag{A.1}$$

where the weight vector is formulated as a 3×3 square matrix with zeros on the main diagonal. When calculating $\frac{\partial C}{\partial \mathbf{w}_{ij}^{l}}$, the matrix $\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger}$ in Eq. 2.21 can be extended as follows:

$$\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger} = \begin{bmatrix} 0 & a_{j3}^{l-1} & -a_{j2}^{l-1} \\ -a_{j3}^{l-1} & 0 & a_{j1}^{l-1} \\ a_{j2}^{l-1} & -a_{j1}^{l-1} & 0 \end{bmatrix}. \tag{A.2}$$

Then, we calculate $\frac{\partial \mathbf{z}_{k}^{l+1}}{\partial \mathbf{a}_{i}^{l}}$ in Eq. 2.27 using the following matrix:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet} = \begin{bmatrix} 0 & -w_{ki3}^{l+1} & w_{ki2}^{l+1} \\ w_{ki3}^{l+1} & 0 & -w_{ki1}^{l+1} \\ -w_{ki2}^{l+1} & w_{ki1}^{l+1} & 0 \end{bmatrix}. \tag{A.3}$$

ABIPNN with this product is identical to the three-dimensional vector product neural network [21].

A.2 Quaternion multiplication

Quaternions are four-dimensional numbers $a+bi+cj+dk$ with the multiplication rule $i^2 = j^2 = k^2 = ijk = -1$. Here $\mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1}$ stands for quaternion multiplication, the weight $\mathbf{w}_{ij}^l = \left[w_{ij1}^l, w_{ij2}^l, w_{ij3}^l, w_{ij4}^l\right]^T \in \mathbb{R}^4$ and the input $\mathbf{a}_j^{l-1} = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, a_{j3}^{l-1}, a_{j4}^{l-1}\right]^T \in \mathbb{R}^4$ are all four-dimensional vectors $(N = 4)$. The matrix representation of $\mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1}$ is:

$$\left[\mathbf{w}_{ij}^l\right]_\bullet = \begin{bmatrix} w_{ij1}^l & -w_{ij2}^l & -w_{ij3}^l & -w_{ij4}^l \\ w_{ij2}^l & w_{ij1}^l & -w_{ij4}^l & w_{ij3}^l \\ w_{ij3}^l & w_{ij4}^l & w_{ij1}^l & -w_{ij2}^l \\ w_{ij4}^l & -w_{ij3}^l & w_{ij2}^l & w_{ij1}^l \end{bmatrix}, \tag{A.4}$$

where the weight vector is formulated as a 4×4 square matrix with w_{ij1}^l on the main diagonal. The matrix $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ for the calculation of $\frac{\partial C}{\partial \mathbf{w}_{ij}^l}$ in Eq. 2.21 can be extended as follows:

$$\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger = \begin{bmatrix} a_{j1}^{l-1} & -a_{j2}^{l-1} & -a_{j3}^{l-1} & -a_{j4}^{l-1} \\ a_{j2}^{l-1} & a_{j1}^{l-1} & a_{j4}^{l-1} & -a_{j3}^{l-1} \\ a_{j3}^{l-1} & -a_{j4}^{l-1} & a_{j1}^{l-1} & a_{j2}^{l-1} \\ a_{j4}^{l-1} & a_{j3}^{l-1} & -a_{j2}^{l-1} & a_{j1}^{l-1} \end{bmatrix}. \tag{A.5}$$

After that, we can use the following matrix $\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet}$ when computing $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$ in Eq. 2.27:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet} = \begin{bmatrix} w_{ki1}^{l+1} & -w_{ki2}^{l+1} & -w_{ki3}^{l+1} & -w_{ki4}^{l+1} \\ w_{ki2}^{l+1} & w_{ki1}^{l+1} & -w_{ki4}^{l+1} & w_{ki3}^{l+1} \\ w_{ki3}^{l+1} & w_{ki4}^{l+1} & w_{ki1}^{l+1} & -w_{ki2}^{l+1} \\ w_{ki4}^{l+1} & -w_{ki3}^{l+1} & w_{ki2}^{l+1} & w_{ki1}^{l+1} \end{bmatrix}. \tag{A.6}$$

With quaternion multiplication, the ABIPNN is equivalent to the quaternion-valued neural network [23].

A.3 Seven-Dimensional Vector Product

The seven-dimensional vector product is defined as per [115]. Here $\mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1}$ stands for the seven-dimensional vector product, $\mathbf{w}_{ij}^l = \left[w_{ij1}^l, w_{ij2}^l, \ldots, w_{ij7}^l\right]^T \in \mathbb{R}^7$ and $\mathbf{a}_j^{l-1} = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, \ldots, a_{j7}^{l-1}\right]^T \in \mathbb{R}^7$ are both seven-dimensional vectors $(N = 7)$, and the matrix

representation of $\mathbf{w}_{ij}^{l} \bullet \mathbf{a}_{j}^{l}$ is:

$$\left[\mathbf{w}_{ij}^{l}\right]_{\bullet} = \begin{bmatrix} 0 & -w_{ij4}^{l} & -w_{ij7}^{l} & w_{ij2}^{l} & -w_{ij6}^{l} & w_{ij5}^{l} & w_{ij3}^{l} \\ w_{ij4}^{l} & 0 & -w_{ij5}^{l} & -w_{ij1}^{l} & w_{ij3}^{l} & -w_{ij7}^{l} & w_{ij6}^{l} \\ w_{ij7}^{l} & w_{ij5}^{l} & 0 & -w_{ij6}^{l} & -w_{ij2}^{l} & w_{ij4}^{l} & -w_{ij1}^{l} \\ -w_{ij2}^{l} & w_{ij1}^{l} & w_{ij6}^{l} & 0 & -w_{ij7}^{l} & -w_{ij3}^{l} & w_{ij5}^{l} \\ w_{ij6}^{l} & -w_{ij3}^{l} & w_{ij2}^{l} & w_{ij7}^{l} & 0 & -w_{ij1}^{l} & -w_{ij4}^{l} \\ -w_{ij5}^{l} & w_{ij7}^{l} & -w_{ij4}^{l} & w_{ij3}^{l} & w_{ij1}^{l} & 0 & -w_{ij2}^{l} \\ -w_{ij3}^{l} & -w_{ij6}^{l} & w_{ij1}^{l} & -w_{ij5}^{l} & w_{ij4}^{l} & w_{ij2}^{l} & 0 \end{bmatrix}, \tag{A.7}$$

where the weight vector is formulated as a 7×7 square matrix in a similar way as the vector product one. The matrix $\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger}$ in Eq. 2.21 becomes:

$$\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger} = \begin{bmatrix} 0 & a_{j4}^{l-1} & a_{j7}^{l-1} & -a_{j2}^{l-1} & a_{j6}^{l-1} & -a_{j5}^{l-1} & -a_{j3}^{l-1} \\ -a_{j4}^{l-1} & 0 & a_{i5}^{l-1} & a_{j1}^{l-1} & -a_{j3}^{l-1} & a_{j7}^{l-1} & -a_{j6}^{l-1} \\ -a_{j7}^{l-1} & -a_{j5}^{l-1} & 0 & a_{j6}^{l-1} & a_{j2}^{l-1} & -a_{j4}^{l-1} & a_{j1}^{l-1} \\ a_{j2}^{l-1} & -a_{j1}^{l-1} & -a_{j6}^{l-1} & 0 & a_{j7}^{l-1} & a_{j3}^{l-1} & -a_{j5}^{l-1} \\ -a_{j6}^{l-1} & a_{j3}^{l-1} & -a_{j2}^{l-1} & -a_{j7}^{l-1} & 0 & a_{j1}^{l-1} & a_{j4}^{l-1} \\ a_{j5}^{l-1} & -a_{j7}^{l-1} & a_{j4}^{l-1} & -a_{j3}^{l-1} & -a_{j1}^{l-1} & 0 & a_{j2}^{l-1} \\ a_{j3}^{l-1} & a_{j6}^{l-1} & -a_{j1}^{l-1} & a_{j5}^{l-1} & -a_{j4}^{l-1} & -a_{j2}^{l-1} & 0 \end{bmatrix}, \tag{A.8}$$

where the signs are flipped when compared to the matrix $\left[\mathbf{w}_{ij}^{l}\right]_{\bullet}$ above. The matrix $\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet}$ for the seven-dimensional vector product is then extended as:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_{\bullet} = \begin{bmatrix} 0 & -w_{ki4}^{l+1} & -w_{ki7}^{l+1} & w_{ki2}^{l+1} & -w_{ki6}^{l+1} & w_{ki5}^{l+1} & w_{ki3}^{l+1} \\ w_{ki4}^{l+1} & 0 & -w_{ki5}^{l+1} & -w_{ki1}^{l+1} & w_{ki3}^{l+1} & -w_{ki7}^{l+1} & w_{ki6}^{l+1} \\ w_{ki7}^{l+1} & w_{ki5}^{l+1} & 0 & -w_{ki6}^{l+1} & -w_{ki2}^{l+1} & w_{ki4}^{l+1} & -w_{ki1}^{l+1} \\ -w_{ki2}^{l+1} & w_{ki1}^{l+1} & w_{ki6}^{l+1} & 0 & -w_{ki7}^{l+1} & -w_{ki3}^{l+1} & w_{ki5}^{l+1} \\ w_{ki6}^{l+1} & -w_{ki3}^{l+1} & w_{ki2}^{l+1} & w_{ki7}^{l+1} & 0 & -w_{ki1}^{l+1} & -w_{ki4}^{l+1} \\ -w_{ki5}^{l+1} & w_{ki7}^{l+1} & -w_{ki4}^{l+1} & w_{ki3}^{l+1} & w_{ki1}^{l+1} & 0 & -w_{ki2}^{l+1} \\ -w_{ki3}^{l+1} & -w_{ki6}^{l+1} & w_{ki1}^{l+1} & -w_{ki5}^{l+1} & w_{ki4}^{l+1} & w_{ki2}^{l+1} & 0 \end{bmatrix}. \tag{A.9}$$

A.4 Circular convolution

Here $\mathbf{w}_{ij}^{l} \bullet \mathbf{a}_{ij}^{l-1}$ stands for the usual circular convolution, $\mathbf{w}_{ij}^{l} = \left[w_{ij1}^{l}, w_{ij2}^{l}, \ldots, w_{ijN}^{l}\right]^{T} \in \mathbb{R}^{N}$ and $\mathbf{a}_{j}^{l-1} = \left[a_{j1}^{l-1}, a_{j2}^{l-1}, \ldots, a_{jN}^{l-1}\right]^{T} \in \mathbb{R}^{N}$ are both N-dimensional vectors, and the matrix representation of $\mathbf{w}_{ij}^{l} \bullet \mathbf{a}_{ij}^{l-1}$ is:

$$\left[\mathbf{w}_{ij}^{l}\right]_{\bullet} = \begin{bmatrix} w_{ij1}^{l} & w_{ijN}^{l} & w_{ij(N-1)}^{l} & \ldots & w_{ij2}^{l} \\ w_{ij2}^{l} & w_{ij1}^{l} & w_{ijN}^{l} & \ldots & w_{ij3}^{l} \\ w_{ij3}^{l} & w_{ij2}^{l} & w_{ij1}^{l} & \ldots & w_{ij4}^{l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{ijN}^{l} & w_{ij(N-1)}^{l} & w_{ij(N-2)}^{l} & \ldots & w_{ij1}^{l} \end{bmatrix}, \tag{A.10}$$

where the weight vector is formulated as an $N \times N$ square matrix with w_{ij1}^{l} on the main diagonal. The matrix $\mathbf{A}_{j}^{l-1}$ for Eq. 2.21 is extended as follows:

$$\mathbf{A}_{j}^{l-1} = \begin{bmatrix} a_{j1}^{l-1} & a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & \dots & a_{j2}^{l-1} \\ a_{j2}^{l-1} & a_{j1}^{l-1} & a_{jN}^{l-1} & \dots & a_{j3}^{l-1} \\ a_{j3}^{l-1} & a_{j2}^{l-1} & a_{j1}^{l-1} & \dots & a_{j4}^{l-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & a_{j(N-2)}^{l-1} & \dots & a_{j1}^{l-1} \end{bmatrix}, \tag{A.11}$$

where the permutation of elements in the matrix $\mathbf{A}_{j}^{l-1}$ is identical to the matrix $\left[\mathbf{w}_{ij}^{l}\right]_{\bullet}$.

The matrix $\mathbf{W}_{ki}^{l+1}$ for circular convolution is extended as:

$$\mathbf{W}_{ki}^{l+1} = \begin{bmatrix} w_{ki1}^{l+1} & w_{kiN}^{l+1} & w_{ki(N-1)}^{l+1} & \dots & w_{ki2}^{l+1} \\ w_{ki2}^{l+1} & w_{ki1}^{l+1} & w_{kiN}^{l+1} & \dots & w_{ki3}^{l+1} \\ w_{ki3}^{l+1} & w_{ki2}^{l+1} & w_{ki1}^{l+1} & \dots & w_{ki4}^{l+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{kiN}^{l+1} & w_{ki(N-1)}^{l+1} & w_{ki(N-2)}^{l+1} & \dots & w_{ki1}^{l+1} \end{bmatrix}. \tag{A.12}$$

A.5 Skew circular convolution

The skew-circular convolution is obtained by replacing the circulant matrix in Eq. A. 10 by a skew-circulant one [50]. The matrix representation of $\mathbf{w}_{ij}^{l} \bullet \mathbf{a}_{j}^{l-1}$ then becomes:

$$\left[\mathbf{w}_{ij}^{l}\right]_{\bullet} = \begin{bmatrix} w_{ij1}^{l} & -w_{ijN}^{l} & -w_{ij(N-1)}^{l} & \dots & -w_{ij2}^{l} \\ w_{ij2}^{l} & w_{ij1}^{l} & -w_{ijN}^{l} & \dots & -w_{ij3}^{l} \\ w_{ij3}^{l} & w_{ij2}^{l} & w_{ij1}^{l} & \dots & -w_{ij4}^{l} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{ijN}^{l} & w_{ij(N-1)}^{l} & w_{ij(N-2)}^{l} & \dots & w_{ij1}^{l} \end{bmatrix}, \tag{A.13}$$

where the weight vector is formulated as an $N \times N$ square matrix with w_{ij1}^{l} on the main diagonal and the upper triangular part of the weight matrix is multiplied by minus one. When we compute $\frac{\partial C}{\partial \mathbf{w}_{ij}^{l}}$ in Eq. 2.21, $\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger}$ is extended as follows:

$$\left[\mathbf{a}_{j}^{l-1}\right]_{\bullet}^{\dagger} = \begin{bmatrix} a_{j1}^{l-1} & -a_{jN}^{l-1} & -a_{j(N-1)}^{l-1} & \dots & -a_{j2}^{l-1} \\ a_{j2}^{l-1} & a_{j1}^{l-1} & -a_{jN}^{l-1} & \dots & -a_{j3}^{l-1} \\ a_{j3}^{l-1} & a_{j2}^{l-1} & a_{j1}^{l-1} & \dots & -a_{j4}^{l-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & a_{j(N-2)}^{l-1} & \dots & a_{j1}^{l-1} \end{bmatrix}. \tag{A.14}$$

When $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$ is computed, the matrix $\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet$ in Eq. 2.27 is extended for skew circular convolution as follows:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet = \begin{bmatrix} w_{ki1}^{l+1} & -w_{kiN}^{l+1} & -w_{ki(N-1)}^{l+1} & \dots & -w_{ki2}^{l+1} \\ w_{ki2}^{l+1} & w_{ki1}^{l+1} & -w_{kiN}^{l+1} & \dots & -w_{ki3}^{l+1} \\ w_{ki3}^{l+1} & w_{ki2}^{l+1} & w_{ki1}^{l+1} & \dots & -w_{ki4}^{l+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{kiN}^{l+1} & w_{ki(N-1)}^{l+1} & w_{ki(N-2)}^{l+1} & \dots & -w_{ki1}^{l+1} \end{bmatrix}. \tag{A.15}$$

If $N = 2$, then ABIPNN becomes a complex-valued neural network [13]. For even $N \geq 2$, the skew circular convolution is also known as planar complex multiplication [49].

A.6 Reverse-time circular convolution

The reversed-time circular convolution is obtained by flipping the circulant matrix upside down (see also [116]). The matrix representation of $\mathbf{w}_{ij}^l \bullet \mathbf{a}_j^{l-1}$ becomes:

$$\left[\mathbf{w}_{ij}^l\right]_\bullet = \begin{bmatrix} w_{ijN}^l & w_{ij(N-1)}^l & w_{ij(N-2)}^l & \dots & w_{ij1}^l \\ w_{ij(N-1)}^l & w_{ij(N-2)}^l & w_{ij(N-3)}^l & \dots & w_{ijN}^l \\ w_{ij(N-2)}^l & w_{ij(N-3)}^l & w_{ij(N-4)}^l & \dots & w_{ij(N-1)}^l \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{ij1}^l & w_{ijN}^l & w_{ij(N-1)}^l & \dots & w_{ij2}^l \end{bmatrix}, \tag{A.16}$$

where the weight vector is formulated as an $N \times N$ square matrix which is the same as the weight matrix for circular convolution rotated by 90 degrees. The matrix $\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger$ in Eq. 2.21 is extended as follows:

$$\left[\mathbf{a}_j^{l-1}\right]_\bullet^\dagger = \begin{bmatrix} a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & a_{j(N-2)}^{l-1} & \cdots & a_{j1}^{l-1} \\ a_{j(N-1)}^{l-1} & a_{j(N-2)}^{l-1} & a_{j(N-3)}^{l-1} & \cdots & a_{jN}^{l-1} \\ a_{j(N-2)}^{l-1} & a_{2(N-3)}^{l-1} & a_{j(N-4)}^{l-1} & \cdots & a_{j(N-1)}^{l-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{j1}^{l-1} & a_{jN}^{l-1} & a_{j(N-1)}^{l-1} & \cdots & a_{j2}^{l-1} \end{bmatrix}, \tag{A.17}$$

Then, the derivative $\frac{\partial \mathbf{z}_k^{l+1}}{\partial \mathbf{a}_i^l}$ in Eq. 2.27 is extended for reverse-time circular convolution by the following:

$$\left[\mathbf{w}_{ki}^{l+1}\right]_\bullet = \begin{bmatrix} w_{kiN}^{l-1} & w_{ki(N-1)}^{l-1} & w_{ki(N-2)}^{l-1} & \cdots & w_{ki1}^{l-1} \\ w_{ki(N-1)}^{l-1} & w_{ki(N-2)}^{l-1} & w_{ki(N-3)}^{l-1} & \cdots & w_{kiN}^{l-1} \\ w_{ki(N-2)}^{l-1} & w_{ki(N-3)}^{l-1} & w_{ki(N-4)}^{l-1} & \cdots & w_{ki(N-1)}^{l-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{ki1}^{l-1} & w_{kiN}^{l-1} & w_{ki(N-1)}^{l-1} & \cdots & w_{ki2}^{l-1} \end{bmatrix}. \tag{A.18}$$

Bibliography

[1] Chase Gaudet and Anthony Maida. Deep quaternion networks. *arXiv preprint arXiv: 1712.04604*, 2017. [Online]. Source code available at: https://github.com/gaudetcj/DeepQuaternionNetworks.

[2] D. P. Mandic and V. S. L. Goh. *Complex Valued Nonlinear Adaptive Filters: Noncircularity, Widely Linear and Neural Models*. Wiley, New York, NY, 2009.

[3] T. Nitta. *Complex-Valued Neural Networks: Utilizing High-Dimensional Parameters*. Information Science Reference, Hershey, PA, 2009.

[4] A. Hirose. *Complex-Valued Neural Networks*. Springer, Berlin, 2013.

[5] A. Hirose. *Complex-Valued Neural Networks: Advances and Applications*. John Wiley and Sons, New York, 2013.

[6] C. Trabelsi, O. Bilaniuk, Y. Zhang, D. Serdyuk, S. Subramanian, J.-F. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. J Pal. Deep complex networks. In *Proc. Int. Conf. Learn. Representations (ICLR)*, 2018.

[7] F. Yasuma, T. Mitsunaga, D. Iso, and S. K. Nayar. Generalized assorted pixel camera: postcapture control of resolution, dynamic range, and spectrum. *IEEE Trans. Image Process.*, 19(9):2241–2253, Mar. 2010. [Online]. Available: http://www1.cs.columbia.edu/CAVE/databases/multispectral/.

[8] S. Koelstra, C. Mühl, M. Soleymani, J.-S. Lee, A. Yazdani, T. Ebrahimi, T. Pun, A. Nijholt, and I. Patras. DEAP: A database for emotion analysis; using physiological signals. *IEEE Trans. Affective Comput.*, 3(1):18–31, 2012.

[9] E. Vincent, S. Watanabe, J. Barker, and R. Marxer. The third CHiME speech separation and recognition challenge: Dataset, task and baselines. In *Proc. IEEE Workshop Automat. Speech Recognition and Understanding (ASRU)*, pages 504–511, 2015.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv: 1502.03167*, 2015.

[11] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Proc. Advances in Neural Information Processing Systems*, pages 2377–2385, 2015.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning

for image recognition. In *Proc. IEEE Conf. Computer Vision Pattern Recognition*, pages 770–778, 2016.

[13] T. Nitta. An extension of the back-propagation algorithm to complex numbers. *Neural Netw.*, 10(8):1391–1415, 1997.

[14] T. Nitta. An analysis of the fundamental structure of complex-valued neurons. *Neural Process. Lett.*, 12(3):239–246, 2000.

[15] T. Nitta. Orthogonality of decision boundaries in complex-valued neural networks. *Neural Comput.*, 16(1):73–97, Jan. 2004.

[16] T. Nitta. Solving the XOR problem and the detection of symmetry using a single complex-valued neuron. *Neural Netw.*, 16(8):1101–1105, 2003.

[17] S. Buchholz and G. Sommer. A hyperbolic multilayer perceptron. In *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, pages 129–133, 2000.

[18] T. Nitta and S. Buchholz. On the decision boundaries of hyperbolic neurons. In *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, pages 2974–2980, 2008.

[19] T. Nitta and Y. Kuroe. Hyperbolic gradient operator and hyperbolic back-propagation learning algorithms. *IEEE Trans. Neural Netw. Learn. Syst.*, (99):1–14, Mar. 2017.

[20] Bipin Kumar Tripathi. *High Dimensional Neurocomputing*. Springer, New Delhi, India, 2015.

[21] T. Nitta. A backpropagation algorithm for neural networks based an 3D vector product. In *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, pages 589–592, 1993.

[22] T. Nitta. Three-dimensional vector valued neural network and its generalization ability. In *Neural Information Processing – Letters and Reviews*, pages 237–242, 2006.

[23] P. Arena, L. Fortuna, L. Occhipinti, and M.G. Xibilia. Neural networks for quaternion-valued function approximation. In *Proc. IEEE Int. Symp. Circuits and Syst.*, pages 307–310, 1994.

[24] P. Arena, L. Fortuna, G. Muscato, and M.G. Xibilia. MLP in quaternion algebra. In *Neural Networks in Multidimensional Domains*, volume 234, pages 49–75. Springer-Verlag London, London, 1998.

[25] T. Nitta. A quaternary version of the back-propagation algorithm. In *Proc. IEEE Int. Conf. Neural Netw.*, pages 2753–2756, 1995.

[26] C.-A. Popa. Octonion-valued neural networks. In *Proc. Int. Conf. Artificial Neural Netw.*, pages 435–443, 2016.

[27] Titouan Parcollet, Ying Zhang, Mohamed Morchid, Chiheb Trabelsi, Georges Linarès, Renato De Mori, and Yoshua Bengio. Quaternion convolutional neural networks for end-to-end automatic speech recognition. *arXiv preprint arXiv: 1806.07789*, 2018.

[28] David Eigen, Dilip Krishnan, and Rob Fergus. Restoring an image taken through a window covered with dirt or rain. In *Proc. IEEE Int. Conf. Computer Vision*, pages 633–640, 2013.

[29] Z. Zhang and S. Aeron. Denoising and completion of 3D data via multidimensional dictionary learning. In *Proc. Int. Joint Conf. Artificial Intelligence (IJCAI)*, pages 2371–2377, 2016.

[30] Z.-C. Fan, T.-S. T. Chan, Y.-H. Yang, and J.-S. R. Jang. Music signal process using vector product neural network. In *Proc. Int. Workshop Deep Learning for Music*, 2017.

[31] S. Jirayucharoensak, S. Pan-Ngum, and P. Israsena. EEG-based emotion recognition using deep learning network with principal component based covariate shift adaptation. *The Scientific World J.*, 2014, 2014.

[32] T. Nitta. N-dimensional vector neuron. In *Proc. Int. Joint Conf. Artificial Intelligence (IJCAI)*, pages 2–7, 2007.

[33] T. Nitta. N□dimensional vector neuron and its application to the N□bit parity problem. In *Complex-valued neural networks: Advances and applications*, pages 59–74. John Wiley and Sons, New York, 2013.

[34] J. K. Pearson and D. L. Bisset. Back propagation in a Clifford algebra. In *Proc. IEEE Int. Conf. Neural Netw.*, volume 2, pages 413–416, 1992.

[35] J. K. Pearson and D. L. Bisset. Neural networks in the Clifford domain. In *Proc. IEEE Int. Conf. Neural Netw.*, volume 3, pages 1465–1469, 1994.

[36] E. J. Bayro-Corrochano. Geometric neural computing. *IEEE Trans. Neural Netw.*, 12(5):968–986, 2001.

[37] Sven Buchholz and Gerald Sommer. Clifford algebra multilayer perceptrons. In *Geometric Computing with Clifford Algebras: Theoretical Foundations and Applications in Computer Vision and Robotics*, pages 315–334. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[38] C.-A. Popa. Matrix-valued neural networks. In *Mendel 2015*, pages 245–255, Cham, 2015. Springer International Publishing.

[39] J.-T. Chien and Y.-T. Bao. Tensor-factorized neural networks. *IEEE Trans. Neural Netw. Learn. Syst.*, (99):1–14, Apr. 2017.

[40] Boris V Kryzhanovsky, Leonid B Litinskii, and Andrey L Mikaelian. Vector-neuron models of associative memory. In *Proc. IEEE Int. Joint Conf. Neural Netw.*, volume 2, pages 909–914. IEEE, 2004.

[41] Ido Kanter. Potts-glass models of neural networks. *Physical Review A*, 37(7):2739, 1988.

[42] Désiré Bollé, Patrick Dupont, and Jort van Mourik. Stability properties of Potts neural networks with biased patterns and low loading. *Journal of Physics A: Mathematical and General*, 24(5):1065, 1991.

[43] Désiré Bollé, Patrick Dupont, and J Huyghebaert. Thermodynamic properties of the Q-state Potts-glass neural network. *Phys. Rew. A*, 45(6):4194, 1992.

[44] Boris V Kryzhanovsky, Leonid B Litinskii, and Anatoly Fonarev. An effective associative memory for pattern recognition. In *Int. Symposium Intelligent Data Anal.*, pages 179–186. Springer, 2003.

[45] Douglas R. Farenick. *Algebras of Linear Transformations*. Springer-Verlag, 2001.

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

[47] T.-S. T. Chan and Y.-H. Yang. Polar n-complex and n-bicomplex singular value decomposition and principal component pursuit. *IEEE Trans. Signal Process.*, 64(24): 6533–6544, 2016.

[48] Todd A. Ell. On systems of linear quaternion functions. *arXiv preprint arXiv:math/0702084*, 2007.

[49] S. Olariu. *Complex Numbers in N Dimensions*. Elsevier, Amsterdam, 2002.

[50] P. J. Davis. *Circulant Matrices*. American Mathematical Society; 2 edition, 2012.

[51] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proc. Int. Conf. Artificial Intelligence and Statistics*, pages 249–256, 2010.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proc. IEEE Int. Conf. Computer Vision*, pages 1026–1034, 2015.

[53] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proc. Int. Conf. Machine Learning*, pages 807–814, 2010.

[54] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, João Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio,

and Christopher J Pal. Deep complex networks. *Proc. Int. Conf. Learning Representations*, 2017.

[55] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[56] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Processing*, 13(4):600–612, 2004.

[57] M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Trans. Signal Process.*, 54(11):4311–4322, Nov. 2006.

[58] M. Elad and M. Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Trans. Image Process.*, 15(12):3736–3745, Dec. 2006.

[59] N. Renard, S. Bourennane, and J. Blanc-Talon. Denoising and dimensionality reduction using multilinear tools for hyperspectral images. *IEEE Trans. Geosci. Remote Sens.*, 5(2):138–142, Apr. 2008.

[60] Y. Peng, D. Meng, Z. Xu, C. Gao, Y. Yang, and B. Zhang. Decomposable nonlocal tensor dictionary learning for multispectral image denoising. In *Proc. IEEE Conf.*

Comput. Vision Pattern Recognition (CVPR), pages 2949–2956, 2014.

[61] X. Liu, S. Bourennane, and C. Fossati. Denoising of hyperspectral images using the parafac model and statistical performance analysis. *IEEE Trans. Geosci. Remote Sens.*, 50(10):3717–3724, Oct. 2012.

[62] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances Neural Info. Process. Systems (NIPS)*, pages 315–323, 2013.

[63] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alexander J Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances Neural Info. Process. Systems (NIPS)*, pages 2647–2655, 2015.

[64] J. Serrà, E. Gómez, and P. Herrera. *Audio Cover Song Identification and Similarity: Background, Approaches, Evaluation, and Beyond*, pages 307–332. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[65] J. L. Durrieu, B. David, and G. Richard. A musically motivated mid-level representation for pitch estimation and musical audio source separation. *IEEE J. Sel. Topics Signal Process.*, 5(6):1180–1191, Oct. 2011.

[66] J. Salamon, G. Peeters, and A. Röbel. Statistical characterisation of melodic pitch contours and its application for melody extraction. In *Int. Soc. for Music Info. Retrieval Conf.*, Porto, Portugal, Oct. 2012.

[67] J. Salamon and J. Urbano. Current challenges in the evaluation of predominant melody extraction algorithms. In *Int. Soc. for Music Info. Retrieval Conf.*, Porto, Portugal, Oct. 2012.

[68] J. Salamon, E. Gómez, D. P. W. Ellis, and G. Richard. Melody extraction from polyphonic music signals: Approaches, applications and challenges. *IEEE Signal Processing Magazine*, In Press (2013).

[69] C. L. Hsu, D. Wang, J.-S. R. Jang, and K. Hu. A tandem algorithm for singing pitch extraction and voice separation from music accompaniment. *IEEE/ACM Trans. Audio, Speech, Language Process.*, 20(5):1482–1491, July 2012.

[70] S. Uhlich, F. Giron, and Y. Mitsufuji. Deep neural network based instrument extraction from music. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, pages 2135–2139, 2015.

[71] Rachel M Bittner, Justin Salamon, Slim Essid, and Juan Pablo Bello. Melody extraction by contour classification. In *Int. Soc. for Music Info. Retrieval Conf.*, pages 500–506, 2015.

[72] Z.-C. Fan, J.-S. R. Jang, and C.-L. Lu. Singing voice separation and pitch extraction from monaural polyphonic audio music via DNN and adaptive pitch tracking. In *Proc. IEEE Int. Conf. Multimedia Big Data. (BigMM)*, pages 178–185, 2016.

[73] Y. Ikemiya, K. Itoyama, and K. Yoshii. Singing voice separation and vocal f0 estimation based on mutual combination of robust principal component analysis and subharmonic summation. *IEEE/ACM Trans. on Audio, Speech, and Language Processing*, 24(11):2084–2095, Nov 2016.

[74] Justin Salamon. Pitch analysis for active music discovery. In *Machine Learning for Music Discovery workshop, International Conference on Machine Learning (ICML)*, 2016.

[75] Z. Rafii and B. Pardo. REpeating Pattern Extraction Technique (REPET): A simple method for music/voice separation. *IEEE Trans. Audio, Speech, Language Process.*, 21(1):73–84, Jan 2013.

[76] P.-S. Huang, S. D. Chen, P. Smaragdis, and M. Hasegawa-Johnson. Singing-voice separation from monaural recordings using robust principal component analysis. In *Proc. IEEE Int. Conf. Acoust., Speech and Signal Process. (ICASSP)*, pages 57–60, 2012.

[77] Y.-H. Yang. Low-rank representation of both singing voice and music accompani-

ment via learned dictionaries. In *Proc. Int. Soc. Music Info. Retrieval Conf. (ISMIR)*, pages 427–432, 2013.

[78] P. Sprechmann, A. M. Bronstein, and G. Sapiro. Real-time online singing voice separation from monaural recordings using robust low-rank modeling. In *Proc. Int. Soc. Music Info. Retrieval Conf. (ISMIR)*, pages 67–72, 2012.

[79] T.-S. Chan, T.-C. Yeh, Z.-C. Fan, H.-W. Chen, L. Su, Y.-H. Yang, and R. Jang. Vocal activity informed singing voice separation with the iKala dataset. In *Proc. IEEE Int. Conf. Acoust., Speech and Signal Process. (ICASSP)*, pages 718–722, 2015.

[80] D. D. Lee and H. S. Seung. Learning the parts of objects by nonnegative matrix factorization. *Nature*, 401:788–791, 1999.

[81] P.-S. Huang, M. Kim, M. Hasegawa-Johnson, and P. Smaragdis. Singing-voice separation from monaural recordings using deep recurrent neural networks. In *Proc. Int. Soc. Music Info. Retrieval Conf. (ISMIR)*, pages 477–482, 2014.

[82] J. R. Hershey, Z. Chen, J. Le Roux, and S. Watanabe. Deep clustering: Discriminative embeddings for segmentation and separation. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, pages 31–35, 2016.

[83] A. L. Maas, Q. V. Le, T. M. O'Neil, O. Vinyals, P. Nguyen, and A. Y. Ng. Recurrent neural networks for noise reduction in robust ASR. In *Proc. Interspeech*, pages 22–

25, 2012.

[84] X.-L. Zhang and D. Wang. Multi-resolution stacking for speech separation based on boosted dnn. In *Proc. Interspeech*, pages 1745–1749, 2015.

[85] Xiao-Lei Zhang and DeLiang Wang. A deep ensemble learning method for monaural speech separation. *IEEE Trans. Audio, Speech, Language Process.*, 24(5):967–977, 2016.

[86] Y. Luo, Z. Chen, J. R. Hershey, J. L. Roux, and N. Mesgarani. Deep clustering and conventional networks for music separation: Stronger together. In *Proc. IEEE Int. Conf. Acoust., Speech and Signal Process. (ICASSP)*, pages 61–65, 2017.

[87] S. Uhlich, M. Porcu, F. Giron, M. Enenkl, T. Kemp, N. Takahashi, and Y. Mitsufuji. Imporving music source separation based on deep neural networs through data augmentation and network blenging. In *Proc. IEEE Int. Conf. Acoust., Speech and Signal Process. (ICASSP)*, pages 261–265, 2017.

[88] SiSEC MUS Homepage, 2016. [Online] https://sisec.inria.fr/sisec-2016/2016-professionally-produced-music-recordings/.

[89] Antoine Liutkus, Fabian-Robert Stöter, Zafar Rafii, Daichi Kitamura, Bertrand Rivet, Nobutaka Ito, Nobutaka Ono, and Julie Fontecave. The 2016 signal separation evaluation campaign. In *Proc. Int. Conf. Latent Variable Anal. Signal Sep-*

aration, pages 323–332. Springer, 2017. [Online]. Available: https://www.sisec17.audiolabs-erlangen.de/.

[90] E. Vincent, R. Gribonval, and C. Fevotte. Performance measurement in blind audio source separation. *IEEE Trans. Audio, Speech, Language Process.*, 14(4):1462–1469, July 2006.

[91] Aditya Arie Nugraha, Antoine Liutkus, and Emmanuel Vincent. Multichannel audio source separation with deep neural networks. *IEEE/ACM Trans. Audio, Speech, Language Process.*, 24(9):1652–1664, 2016.

[92] F.-R. Stöter, A. Liutkus, R. Badeau, B. Edler, and P. Magron. Common fate model for unison source separation. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, pages 126–130, 2016.

[93] Alexey Ozerov, Emmanuel Vincent, and Frédéric Bimbot. A general flexible framework for the handling of prior information in audio source separation. *IEEE Trans. Audio, Speech, Language Process.*, 20(4):1118–1133, 2012.

[94] Po-Sen Huang, Minje Kim, Mark Hasegawa-Johnson, and Paris Smaragdis. Joint optimization of masks and deep recurrent neural networks for monaural source separation. *IEEE/ACM Trans. Audio, Speech, Language Process.*, 23(12):2136–2147, 2015.

[95] Antoine Liutkus, Derry Fitzgerald, and Zafar Rafii. Scalable audio separation with light kernel additive modelling. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, pages 76–80. 2015.

[96] Il-Young Jeong and Kyogu Lee. Singing voice separation using rpca with weighted l_ {1} -norm. In *Proc. Int. Conf. Latent Variable Anal. Signal Separation*, pages 553–562. 2017.

[97] Pritish Chandna, Marius Miron, Jordi Janer, and Emilia Gómez. Monoaural audio source separation using deep convolutional neural networks. In *Proc. Int. Conf. Latent Variable Anal. Signal Separation*, pages 258–266. Springer, 2017.

[98] S. Uhlich, M. Porcu, F. Giron, M. Enenkl, T. Kemp, N. Takahashi, and Y. Mitsufuji. Improving music source separation based on deep neural networks through data augmentation and network blending. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, pages 261–265, 2017.

[99] Andrew JR Simpson, Gerard Roma, Emad M Grais, Russell D Mason, Chris Hummersone, Antoine Liutkus, and Mark D Plumbley. Evaluation of audio source separation models using hypothesis-driven non-parametric statistical methods. In *2016 24th European Signal Processing Conference (EUSIPCO)*, pages 1763–1767. IEEE, 2016.

[100] Marcelo Bertalmio, Guillermo Sapiro, Vincent Caselles, and Coloma Ballester. Image inpainting. In *Proc. Annual Conf. Computer Graphics and Interactive Techniques*, pages 417–424, 2000.

[101] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Proc. Advances in Neural Information Processing Systems*, pages 341–349, 2012.

[102] C. Guillemot and O. Le Meur. Image inpainting: Overview and recent advances. *IEEE Signal Processing Magazine*, 31(1):127–144, 2014.

[103] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Globally and locally consistent image completion. *ACM Trans. Graphics*, 36(4):107, 2017.

[104] Tony F Chan and Jianhong Shen. Non-texture inpainting by curvature-driven diffusions (CDD). 2001.

[105] Simon Masnou and J. M. Morel. Level lines based disocclusion. In *Int. Conf. Imgae Processing*, pages 259–263, 1998.

[106] Alexei A Efros and William T Freeman. Image quilting for texture synthesis and transfer. In *Proc. Annual Conf. Computer Graphics and Interactive Techniques*, pages 341–346, 2001.

[107] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. In *ACM Trans. Graphics*, volume 24, pages 795–802, 2005.

[108] Rolf Köhler, Christian Schuler, Bernhard Schölkopf, and Stefan Harmeling. Mask-specific inpainting with deep neural networks. In *Proc. German Conf. Pattern Recognition*, pages 523–534, 2014.

[109] Jimmy SJ Ren, Li Xu, Qiong Yan, and Wenxiu Sun. Shepard convolutional neural networks. In *Advances in Neural Information Processing Systems 28*, pages 901–909. 2015.

[110] Deepak Pathak, Philipp Krahenbuhl, Jeff Donahue, Trevor Darrell, and Alexei A Efros. Context encoders: Feature learning by inpainting. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 2536–2544, 2016.

[111] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, and Thomas S Huang. Generative image inpainting with contextual attention. *arXiv preprint arXiv: 1801.07892*, 2018.

[112] Bryan C Russell, Antonio Torralba, Kevin P Murphy, and William T Freeman. Labelme: a database and web-based tool for image annotation. *Int. J. Computer Vision*, 77(1-3):157–173, 2008.

[113] Peter D. Burns and Roy S. Berns. Analysis multispectral image capture. In *Proc. Color and Imaging Conf.*, pages 19–22, 1996.

[114] Weiying Xie and Yunsong Li. Hyperspectral imagery denoising by deep learning with trainable nonlinearity function. *IEEE Geoscience and Remote Sensing Letters*, 14(11):1963–1967, 2017.

[115] P. Lounesto. *Clifford Algebras and Spinors*. Cambridge University Press, Cambridge, 2001.

[116] S. R. Powell and P. M. Chau. Time reversed filtering in real-time. In *Proc. IEEE Int. Symp. Circuits Syst.*, pages 1239–1243, 1990.