Design and Implementation of a Convolutional Neural Network Using Tensor-Train Decomposition

by

Junyao Pu

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements of the degree of

Master of Science

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, MB, Canada

Abstract

Neural networks show state-of-the-art performance in different fields. However, this technique suffers a memory consumption issue as we are handling high-dimensional data more and more often. In this thesis, we introduce a new formulation of the convolutional layer and verify a new training algorithm using Bayesian inference. Here we refer to any neural networks with any tensor-train-layers and trained by Bayesian training algorithm as a Bayesian TensorNet (BTN). The BTN provides a compressed network size and simplifies the operation in the neural network forward computation.

We developed a novel tensor-train formulation of a convolutional neural network and trained it with a Bayesian training algorithm for a plant classification problem. We used the idea of representing the fully connected layer given by Novikov, and our novel tensor-train representation for the convolutional layer which is more general and straight than the tensor-train representation given by Garipov. We tested our BTN with a Bayesian training algorithm, which is a algorithm completely different than the backpropagation training algorithm where we do not need to compute any gradient of the network's weights. The training of our BTN was done with a dataset of plant images from the TerraByte project, an academic agriculture project focusing on the machine learning application development in modern digital agriculture. We have tested the training result by achieve a 67% accuracy in the plant classification problem. Currently, the BTN developed here is still computationally expensive. It could benefit from further optimization, graphics processing unit (GPU) acceleration support and new development of neural network architectures. Suggested future work includes the exploration of another numerical integration method and a fair comparison to the backpropagation training algorithm.

Keywords: Machine Learning, Deep learning, Digital Agriculture Bayesian inference, Tensor decomposition, Sequential Monte Carlo method.

Acknowledgment

Above all, I would like to express my very great appreciation to my supervisors, Dr. Sherif Sherif and Dr. Christopher Bidinosti for their excellent guidance, encouragement, and all the support in my master's program. It was a great opportunity to work under their supervision. I would also like to thank my thesis committee members, Dr. Christopher Henry and Dr. Ahmed Ashraf for his expertise and assistance.

Thank you as well to my girlfriend and parents for your continuous support and love.

Dedication

In dedication to my family.

Contents

List of Figures

Chapter 1

Introduction

In recent years, we have witnessed the rise of the artificial neural network (ANN), and the technique has made great success in almost every field [1, 2, 3, 4, 5]. The success of the ANNs is made by algorithmic advances, large amounts of data, and modern computing hardware. However, expensive hardware and long training and inference times tend to prohibit the use of cheap or portable devices [6] and therefore limit the further application of ANNs.

A reduction of the neural network size and fast forward computation is immediately needed to equip this state-of-the-art technology for cheaper or portable devices. One such approach is the tensor-train decomposition method, first introduced by Oseledets in 2011, which decomposes a high order tensor into lower-order tensor cores with fast tensor operations [7]. After that, Novikov and Podoprikhin proposed their tensor-train representation of fully connected layers in 2015 [6]. In the following year, the tensor-train representation of convolutional layers was introduced by Garipov as an extension of Novikov's work [8].

However, the decomposition of both the fully connected layer and convolutional layer is based on a similar idea, which is representing a huge vector or matrix into the tensor-train format. The tensor-train format will be used to operate a matrix-matrix multiplication [6, 8]. This idea of representing the fully connected layer is fine, since the fully connected layer is a matrix-matrix multiplication. The convolutional layer with multiple filters is a high-order tensor performing the correlation operation instead of the matrix-matrix multiplication. Therefore, a more general tensor-train formulation of the convolutional layer with simple tensor operations exists. We developed a more general tensor-train formulation on the convolutional layer. Our novel formulation can deal with tensors of any dimension and is not restricted with the matrix-matrix multiplication. Such a tensor-train formulation would help us to reduce the overall size of the ANN and speed up the ANN training and inference time.

Training a neural network is usually done by using the backpropagation algorithm. The neural network in tensor-train format is not a typical neural network, since the gradient of each layer is not easy to calculate. Therefore, a Bayesian algorithm was used to train and test our tensor-train neural network given by Doucet [9].

The development of a tensor-train neural network would be extremely useful. Such a neural network would be a powerful tool to use on the cheap and portable hardware due to its smaller size and faster forward computation.

1.1 Organization of This Thesis

Developing a new tensor-train formulation of the convolutional neural network requires an understanding of the operation of convolutional neural network, in addition to understanding the tensorization technique and training with a Bayesian algorithm.

In Chapter 2 the theory of convolutional neural network and different types of training algorithms are briefly discussed.

To represent the convolutional neural network in a tensor-train format, one needs to know the tensorization methods. Therefore Chapter 3 discusses the basic tensor operations and tensor decomposition methods.

In Chapter 4 we provide a idea of Bayes' theorem train any neural networks, and discuss some numerical methods for solving the recursive Bayesian solution.

In Chapter 5 we present our novel tensor-train formulation of the convolutional layer and the idea of evaluating the recursive Bayesian solution by using the Sequential Monte Carlo method.

In Chapter 6 we validate our tensorization method and the training algorithm by showing the training results with a plant images classification problem where the plant dataset was used from the TerraByte project. We also analyze the trained model performance with different training noise and different estimator selection. Finally, in Chapter 7 we present our conclusions and suggested future work on the next development of our research.

1.2 Thesis Research Contributions

We developed a novel tensor-train formulation of the convolutional neural network and trained it with a Bayesian training algorithm for a plant classification problem.

A convolutional neural network is mainly made by fully connected layers and convolutional layers. We used the idea of representing the fully connected layer given by Novikov [6]. On the other hand, we used our novel tensor-train representation for the convolutional layer which is more general and straight than the tensor-train representation given by Garipov [8]. Our tensor-train representation can deal with any convolutional layer with arbitrary dimension and operation.

Our novel tensor-train convolutional neural network was trained with a Bayesian training algorithm. We have tested the training result by achieve a 67% accuracy in the plant classification problem. We presented our model inference result with the mean and MAP estimators. We analyzed the training performance of this training algorithm with different training noise.

Chapter 2

Artificial Neural Network

2.1 Applications of Neural Network to Digital Agriculture

Digital agriculture is important to confront the challenges of food production as the need for food explodes as mankind's population increases. Nowadays, integrating ANN into digital agriculture is helping us with crop diseases detection, weed classification, harvesting rate approximation, and soil moisture prediction [10]. These applications would help us farm more efficiently and produce more food for the total agriculture output.

Developing ANN applications in digital agriculture heavily relies on training data. A lack of sufficient training data, both in terms of variety and quantity, is often the bottleneck in the development [11]. To overcome this problem, The University of Winnipeg TerraByte project has developed a robotic system to automatically generate and label a large dataset of plant images for the development of ANN applications for the digital agriculture development.

2.2 The TerraByte Project

The TerraByte is a digital agriculture project led by professor Christopher Bidinosti and professor Christopher Henry at the University of Winnipeg. The project focuses on the lack of sufficient training data both in terms of variety and quantity, which is the bottleneck in the development of modern agriculture machine learning applications. Recently, TerraByte has successfully developed an embedded robotic system called EAGL-I (Embedded Automated Generator of Labeled Images), which is able to automatically generate and label plant images for machine learning applications [11]. The EAGL-I system at the TerraByte project is shown in Fig. 2.1. TerraByte provides a large number of images

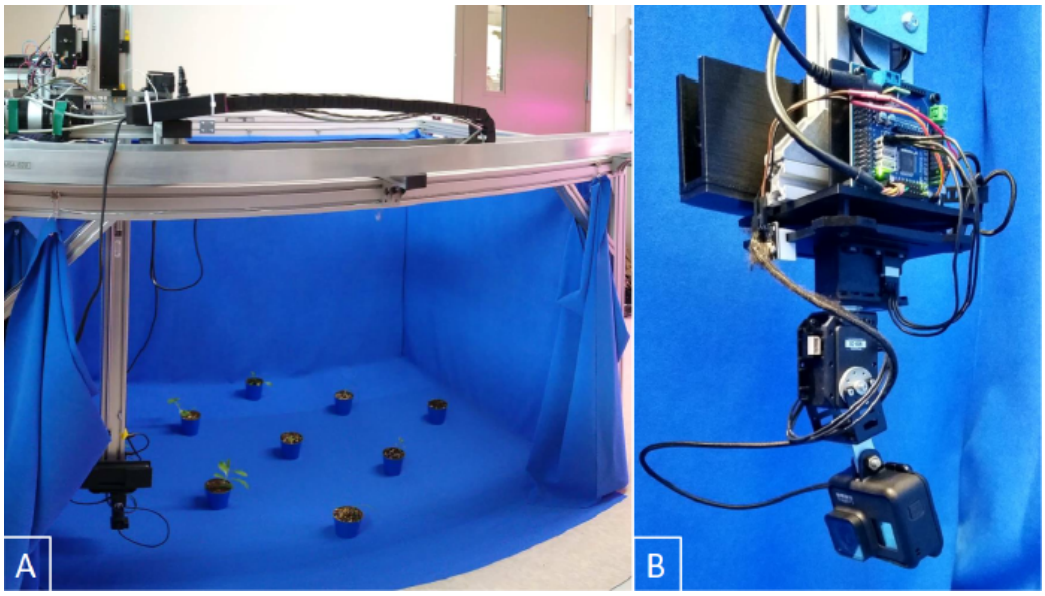

Figure 2.1: Figure A shows the full view of the EAGL-I system, where individual plant are placed inside the EAGL-I system. Pictures of plants from different angles and distances are captured by a GoPro camera. Figure B is a close view of the camera system which can move in 3 dimensions and rotate the camera for a different view of the plant [11].

with labels of various crop plants and weeds found in Manitoba Canada. There include canola, soybean, wheat, foxtail, etc. The datasets include multi-plant images and single plants images in RGB, as well as hyperspectral and 3D point clouds.

In this thesis, we focus on verifying our new tensor-train (TT) format of the convolutional layer in a convolutional neural network. To set up our experiment, we have chosen the TerraByte single plant dataset for a classification problem. The benefits of using this dataset are easy to access to the dataset, high-quality plant images, excellent class labels, and a connection to an important real-world application. Samples of single plant image from the TerraByte dataset are shown in Fig. 2.2

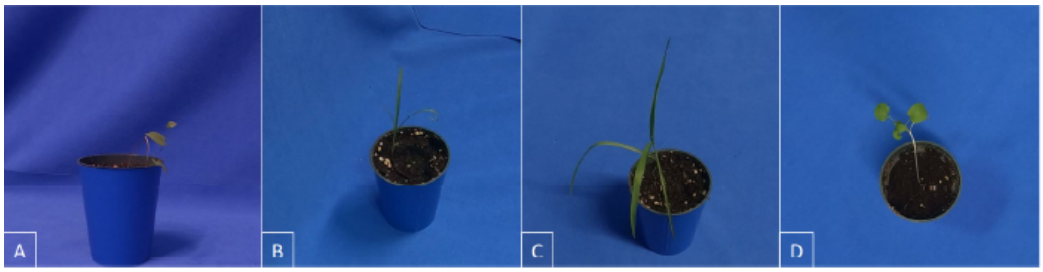

Figure 2.2: The plot shows some plant images captured by the EAGL-I system. Image A is a picture of wild buckwheat from a side-angle view. Images B and C are images of yellow foxtail and barnyard grass from oblique angle views, respectively. Image D is canola from a top view [11].

2.3 Fully Connected Neural Network

The term neural network was first introduced by McCullough and Pitts in 1943 who proposed a computational model called threshold logic [12]. In 1958, Frank proposed the idea of perception which is the first neural network [13]. After a long period after the first introduction of the neural network, Dreyfus adapted the neural network parameter backpropagation with error gradients. This improved the backpropagation algorithm which enabled multiple layers of neural network training, his method became widely used in the neural network community. In the recent decade, the increased computing power from GPUs allowed us to use the larger and more complex neural networks, which enabled us to solve a lot of problems we thought could not be done with the neural network [14]. In the general sense, the neural network includes the fully connected network, convolutional neural network, recurrent network, etc.

The simplest neural network is the fully connected neural network, where all the network layers are all fully connected. The mathematical representation for a fully connected neural network is given by

$$\begin{aligned} l_0 &= x \\ l_n &= h(W_n l_{n-1} + b_n) \\ l_N &= y, \end{aligned} \tag{2.1}$$

where $l_0 = x$ is the neural network input, l_n is the the n^{th} hidden layers, the function $h()$ is the non-linear activation function, and W_n and b_n are weights and bias, respectively. The last layer or the output of a fully connected neural network is $l_N = y$. For an example of an image classification problem, the neural network input $l_0 = x$ is the pixels of the image, W_n and b_n are trainable parameters to optimize the neural network, and the output $l_N = y$ is the predicted class of the input image. A graphical representation of the fully connected neural network is shown in Fig. 2.3

2.3.1 Fully Connected Layer

The fully connected layer is the main building block for the fully connected neural network. The job of fully connected layers is doing a matrix multiplication that computes the next layer's value. The input of a fully connected layer is usually a vector $x \in \mathbb{R}^{(J)}$, and the layer's weight is a matrix $W \in \mathbb{R}^{(I \times J)}$. The layer's weight matrix is first multiplied by the input vector. The computation of a single fully connected layer can be represented mathematically by

$$l = h(Wx + b), \tag{2.2}$$

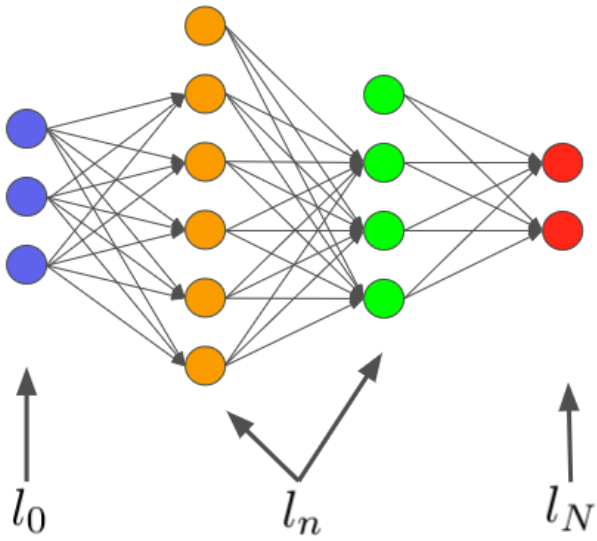

Figure 2.3: A graphical representation of a fully connected neural network. The neural network input is on the left-hand side, the neural network output is on the right-hand side, and the hidden layers are in the between of the input and output layer. In the example of the image classification, the left-hand side neurons are image pixels, and the right-hand side neuron are the classes. These hidden layers are representing the relationship between the input and output in the neural network.

where l is the layer's output, W is the layer's weight matrix, x is the layer's input vector, and b is the bias. A graphical representation of a fully connected layer with all trainable parameters is given in Fig. 2.4

2.3.2 The Activation Function

The activation function $h()$ plays an important role to enable our neural network to deal with non-linear problems. The activation function is usually used after every layer to provide a non-linearity into the neural network. It helps the neural network to solve non-linear problems. After years of development, the most common activation functions are sigmoid function, tanh function, ReLU function, etc. Examples of some activation functions such as ReLU and sigmoid are

ReLU function

$$h(x) = max(0, x). \tag{2.3}$$

Sigmoid function

$$h(x) = \frac{1}{1 + \exp(-x)}. \tag{2.4}$$

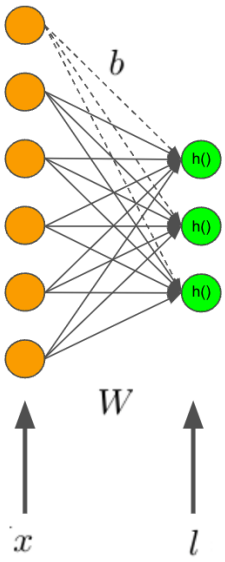

Figure 2.4: A graphical representation of a fully connected layer, the layer's input is on the left-hand side, the layer's output is on the right-hand side. The solid lines are the layer's weight W which is doing the matrix-vector multiplication with the input. The dash lines are the layer's bias which is simply added to the result. After computing the result with the layer's weight and bias, a non-linearity activation function $h()$ is applied to the result.

The choice of the activation function is based on the use of application. Here we also show the graphical representation of a ReLu and sigmoid function in Fig. 2.5.

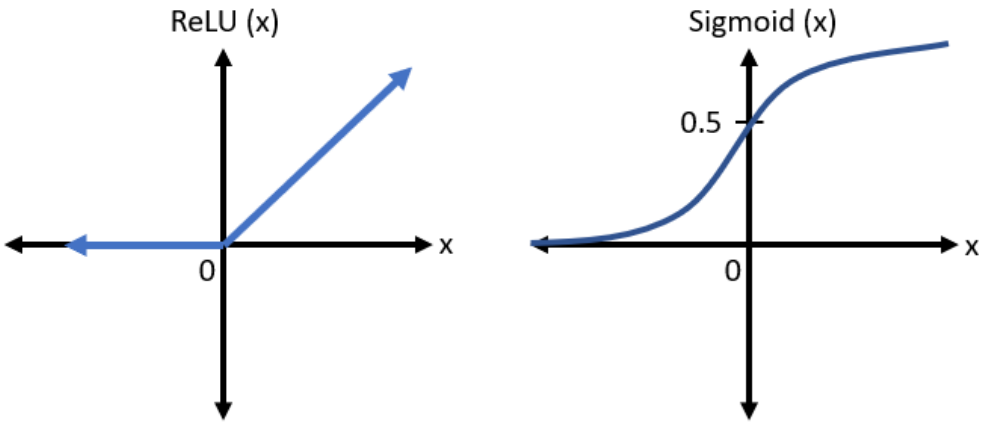

Figure 2.5: Graphical representation of the ReLU on the left-hand side, and Sigmoid activation function on the right-hand side. The ReLU have value 0 for $x \leq 0$ and a linear positive value for $x \geq 0$. On the other hand, the

2.4 Convolutional Neural Network

Convolutional neural networks, also referred to as CNNs, are a particular class of neural networks. CNNs show state-of-the-art performance on image classification, image segmentation, object detection, etc. However, the CNNs are nothing but an extension of the fully connected neural networks, where the CNNs are made with several convolutional layers and a single flatten layer on the top of a fully connected neural network. So, the CNNs can be considered as an input layer, numbers of convolutional layers and a flatten layer to connect to a fully connected neural network.

For the example of an image classification problem, let us consider a CNN that has a RBG image $I \in \mathbb{R}^{(I_1 \times I_2 \times C)}$ as the input. Assuming the CNN has only one convolutional layer, and it has M number of filters given by$F \in \mathbb{R}^{(J_1 \times J_2 \times C \times M)}$, those filters with the input RGB images produce the convolutional layer output called the activation map given by $R \in \mathbb{R}^{(I_1 - J_1 + 1) \times (I_2 - J_2 + 1) \times (M)}$. After the operation with the convolutional layer, the activation map will be vectorized by the flatten layer into a vector. The vector will be used as the input to the fully connected neural network, where the fully connected neural network will classify those input from the activation map features into its class or the output of the convolutional neural network. A graphical representation of the CNN with image classification problem is shown in Fig. 2.6

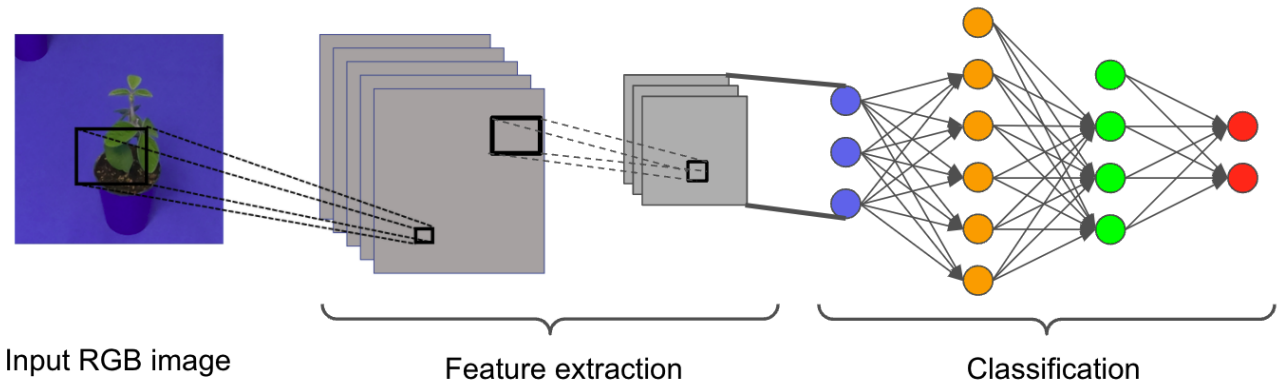

Figure 2.6: The graphical representation of a convolutional neural network for an image classification problem. An RGB image is sent to the convolutional neural network as an input, the image features are extracted by multiple convolutional layers. After the feature extraction, the image features are passed to a fully connected neural network via a flatten layer, these image features are used to classify the input image to its class by the fully connected neural network on the right-hand side.

Therefore, the most important part of the convolutional neural network is the convolutional layers which capture the spatial and color information of the image. In addition, the flatten layer is a bridge

to pass these image features to a fully connected neural network. The classification part is actually done by the fully connected neural network.

2.4.1 Convolutional Layer

The convolutional layer is the key part of the convolutional neural network, allowing improved performance over the fully connected neural network, and giving the ability to capture the image features. The way of the convolutional layer captures image features is using multiple filters via the correlation between the input image and each filter.

Let us consider a simple case, by giving a grayscale image $X \in \mathbb{R}^{(I_1 \times I_2)}$ as input image where I_1 is the image width and l_2 is the image height. Also given a convolutional layer with single filter $Y \in \mathbb{R}^{(J_1 \times J_2)}$ where J_1 and J_2 are the size of each filter. The fully overlapping correlation is the correlation operation but ignores the boundary, the fully overlapping correlation result between the grayscale image and single filter is given by $P^{(m)} \in \mathbb{R}^{(I_1 - J_1 + 1) \times (I_2 - J_2 + 1)}$. Fig 2.7 demonstrates how the fully overlapping operation is implemented on the image and filter.

Now, let us consider that the convolutional layer has M filters given by $Y \in \mathbb{R}^{(J_1 \times J_2 \times M)}$. Every filter is doing the same fully overlapping correlation with the input image. By stacking all results produced by the image and every filter, we end with the convolutional layer's result or the activation map $P \in \mathbb{R}^{(I_1 - J_1 + 1) \times (I_2 - J_2 + 1) \times (M)}$.

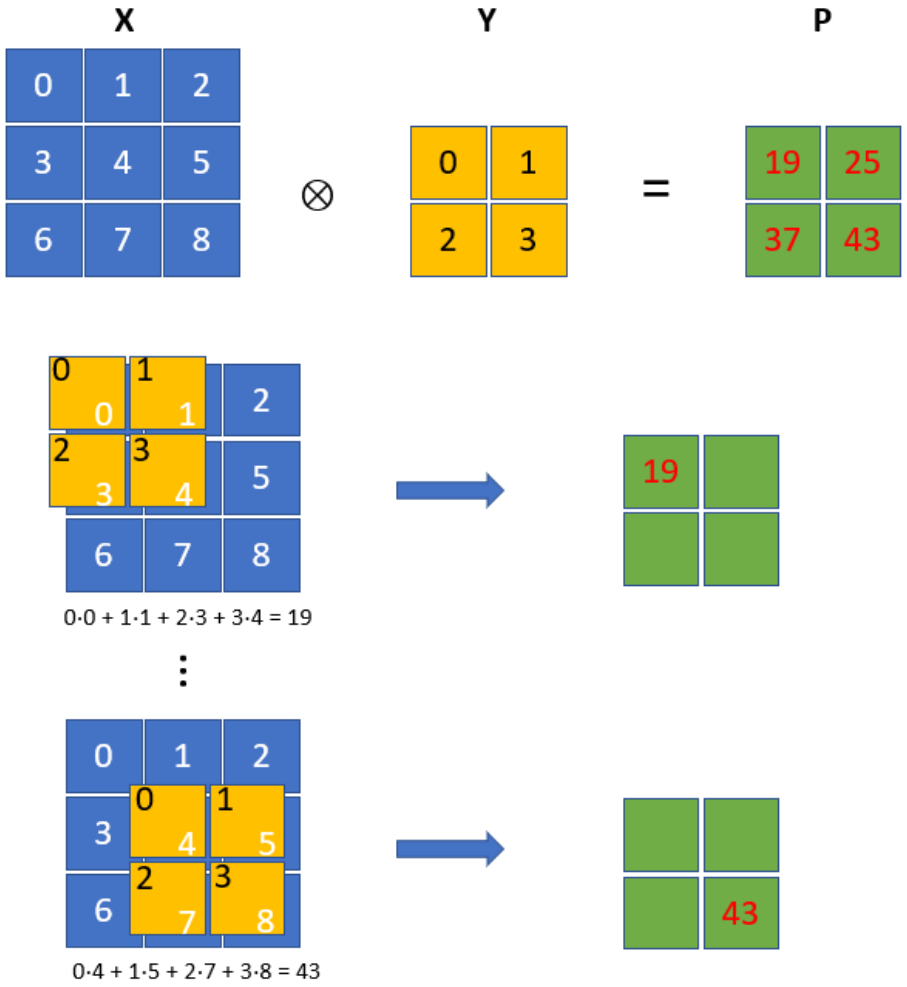

Figure 2.7: Let us assume the blue matrix X is a 3 by 3 image, the yellow matrix Y is a 2 by 2 filter. The fully overlapping correlation result of the image and the filter is the green matrix Y. The first line shows the final result of the fully overlapping correlation, the second line shows how the element $Y(0,0)$ is computed and the last line shows how the element $Y(1,1)$ is computed.

2.4.2 Flatten Layer

We consider the flatten layer as a bridge that connects an activation map and the first fully connected layer. It rearranges the last activation map into a vector that can be accepted by the fully connected layer. For example, the final activation map is given by $R \in \mathbb{R}^{(I_1-J_1+1)\times(I_2-J_2+1)\times(M)}$. we then flatten this 3-rd order tensor into a vector $r \in \mathbb{R}^{(I_1-J_1+1)\cdot(I_2-J_2+1)\cdot(M)}$. After the flatten layer, the elements of the vector r are used by the fully connected neural network for the purpose of the problem, which in our case is used to classify the image to its class. The transformation of the high order tensor into a vector by the flatten layer is demonstrated in Fig.2.8.

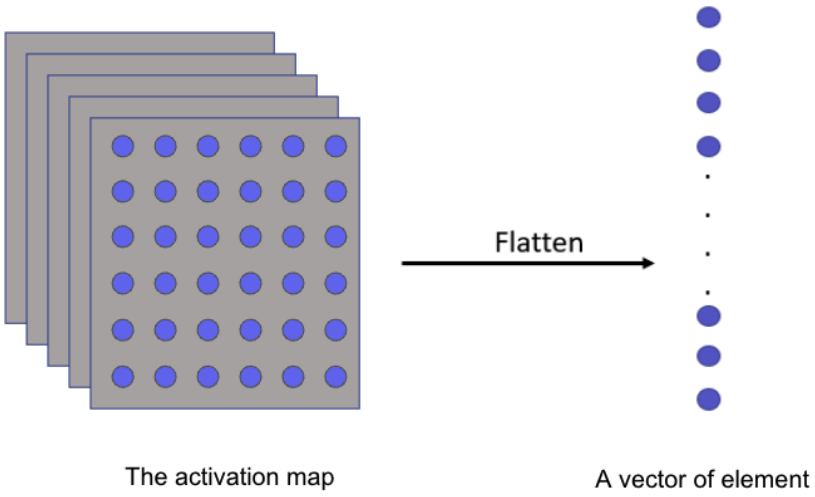

Figure 2.8: Graphical representation of the flatten layer. The flatten layer reshapes the activation map, which is a tensor, into a vector of elements.

2.5 Neural Network Training

Neural network training is a key part of deploying the neural network. Training a neural network means to find the appropriate weights to fit the given training data. It is usually the most time comsuming part in the development of a neural network.

2.5.1 Backpropogation Training Algorithm

The backpropagation or the backward propagation of errors method was first introduced in 1970s. In the last decade, backpropagation became the most common algorithm to train the neural network. This algorithm use the gradient of a loss function to adjust the neural network's weights. The training process of the backpropagation with gradient descent can be represented in the following equation

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta_t} - \alpha \frac{\partial L(\boldsymbol{\theta}, \boldsymbol{X})}{\partial \boldsymbol{\theta}}, \tag{2.5}$$

where $\boldsymbol{\theta_t} = \{\boldsymbol{W}, \boldsymbol{b}\}$ is the neural network weights, including the layer's weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$ at time t. α is a small scalar called the learning rate and the function L is the loss or error function to measure the difference between the prediction and the target. The simplest loss function is the mean squared error which is given by

$$L(\boldsymbol{\theta}, \boldsymbol{X}) = \frac{1}{2N} \sum_{n=1}^{N} (\hat{y_n} - y_n)^2, \tag{2.6}$$

where N is the total number of training pairs, $\hat{y_n}$ is the model prediction of n-th training pair, and y_n is the ground truth of the n-th training pair.

However, not every neural network is trainable with the backpropagation algorithm, especially if it is hard or impossible to compute the gradient of the loss function. Our tensor-train neural network is one such example where the gradient is not easy to compute. Fortunately, a stochastic algorithm such as the Bayesian training algorithm dose not required a gradient. It is a ideal training algorithm for our tensor-train neural network.

2.5.2 Bayesian Training Algorithm

Training a neural network is just fitting the training data with the neural network model. Therefore, training the neural network model can be thought as finding the relationship in the training data. Here we can consider the neural network with current weight as a hypothesis of the relationship in the training data between the input and output. To have a neural network model with good trainable parameters is to have a good hypothesis by giving the relationship in the training data.

Bayes theorem is a useful tool to test the hypotheses. It provides a probabilistic neural network to represent the relationship between the training data and hypothesis as

$$Pr(Hypothesis|Data) = \frac{Pr(Data|Hypothesis)Pr(Hypothesis)}{Pr(Data)}, \tag{2.7}$$

where the probability of the testing hypothesis given by the data is computed by the probability of the data given by the hypothesis is true , the probability of the hypothesis is true and the probability of the data is given. Under the Bayesian inference, we can derive a Bayesian recursive solution to sequentially estimate the neural network's trainable parameters. However, the Bayesian recursive solution can not usually be done analytically, and a numerical method is needed to approximate the solution.

The Monte Carlo (MC) method is a simple and intuitive method to numerically solve high-dimensional problems. One of the MC methods is the Sequential Monte Carlo (SMC) method which allows us to solve a high dimensional problem sequentially. This sequential property fits the need for approximating the Bayesian recursive solution. Therefore, the SMC method is used in this thesis for estimating our Bayesian solution.

2.6 Convolutional Neural Network in Tensor Train Format

2.6.1 Limitations of Modern Convolutional Neural Network

As we are able to solve more and more complex problems with these state-of-the-art techniques, the number of convolutional layers and fully connected layers are increasing the complexity of the problem. In particular, a large amount of memory is needed for modern neural networks, making it hard to use the neural networks on low-end hardware and stopping the further increase of the neural network size [6].

However, the weights of the neural network layers, such as the convolutional layer can be considered as a 4^{th} order tensor, and the fully connected layer is a huge matrix or 2^{nd} order tensor. One solution to reduce the tensor size and simplify the computational operations is using low-rank representation or tensor decomposition, such as the Canonical Polyadic decomposition, the Tucker decomposition, and the tensor-train decomposition. The tensor-train decomposition is chosen in this thesis to overcome these storage and computational operation limitations.

For a convolutional neural network, the network trainable parameters are stored in the fully connected layers and convolutional layers. Decomposing these layers into tensor train format allows us to benefit from storage reduction and simplify the forward computation by using tensor operations on the smaller tensor cores.

2.6.2 Tensor Train Decomposition

Tensor decomposition is a technique to decompose a high-order tensor into numbers of low-order tensors, this reduces the total size of the elements needed to represent a high order tensor and simplify the operation by operating on these low-order tensors instead of the original tensor. For example, to decompose a 3^{rd} order tensor with the tensor-train decomposition, the 3^{rd} order tensor can be represented by 3 smaller tensors. The tensor train format of the 3^{rd} order tensor is shown in Fig. 2.9.

The tensor rrain format provides a lot of tensor operations such as correlation and tensor product, which allows us to use those operations on the smaller tensor core to operate on the original tensor. And the convolutional layer and fully connected layer are basically made by tensors and operate the correlation and matrix multiplications. As a result the tensor train format can help us to improve the convolutional neural network's performance.

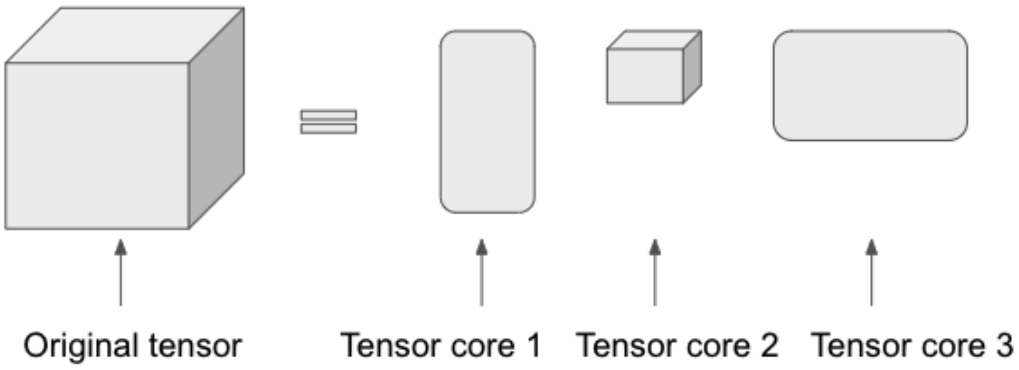

Figure 2.9: Let us consider a 3^{rd} order tensor on the left-hand side. The tensor-train format of the 3^{rd} order tensor is constructed by a tensor core 1, a 2^{nd} order tensor; a tensor core 2, a smaller 3^{rd} order tensor; and tensor core 3, a 2^{nd} order tensor. Overall, the right hand side has fewer elements than the left hand side.

Chapter 3

Tensor Methodology

The name tensor was first introduced by William in 1846. He described something different from what we see as a tensor today. The tensor with today's meaning was given by Woldemar in 1898 [15]. However, the development of tensor analysis really started around 1915 with Einstein's theory of general relativity, and Einstein's general relativity was formulated completely in the language of tensors. In recent years, tensor algebra and tensors are attracting attention due to the huge amount of computational data processing in high dimensions and sizes. The tensor that we see every day, can be just as simple as a vector or a 1^{st} order tensor, a matrix is a 2^{nd} order tensor, etc. Examples of low order tensors from order 0 to 4 are shown in Fig. 3.1.

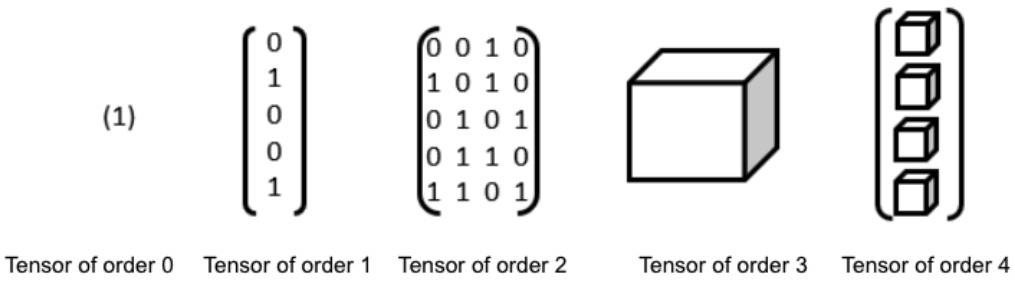

Figure 3.1: Graphical representation of multi-dimensional tensors. The first one is a scalar or tensor of order 0, the second one is a vector or tensor of order 1, the third one is a matrix or tensor of order 2, the fourth one is a tensor of order 3 and the last one is a tensor of order 4.

3.1 Tensor Notation

For convenience and ease in explaining the tensor concept, we first define our notation describing tensors. In this thesis, scalars are denoted by lower non-capital letter, e.g., x, vectors are denoted by bold lower non-capital letter, e.g., $\boldsymbol{x} \in \mathbb{R}^I$; matrices are denoted by bold capital letter, e.g., $\boldsymbol{X} \in \mathbb{R}^{I_1 \times I_2}$; and N^{th}-order tensor is denoted by a bold script letter, e.g.,$\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$. The element of the N^{th}-order tensor is denoted by $\mathcal{X}_{i_1,\ldots,i_n,\ldots,i_N} = \boldsymbol{\mathcal{X}}[i_1, \ldots, i_n, \ldots, i_N]$. Unless otherwise stated, we will use this notation in this thesis.

3.2 Tensor Graphical Representation

The graphical representation shows the tensor shape and tensor operation graphically. When working on high-order tensor operations, the graphical representation helps us to visualize the process. In the tensor graphical representation, we will represent an N^{th}-order tensor as a blue circle with N arms where those arms indicate the number of dimensions. Samples of the tensor graphical representation are shown in Fig. 3.2. In addition, some simple tensor operations like matrix-vector multiplication, matrix-matrix multiplication, and tensor contraction are also shown in Fig. 3.3.

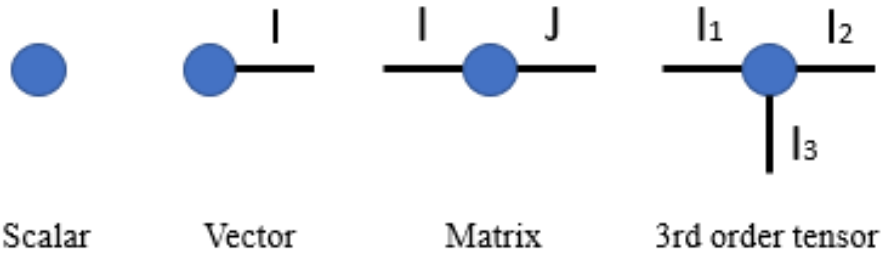

Figure 3.2: Graphical representation of scalar, vector, matrix, and 3rd order tensor, where the number of arms is the number of dimensions of the tensor. This representation can easily identify the order of the tensor [16].

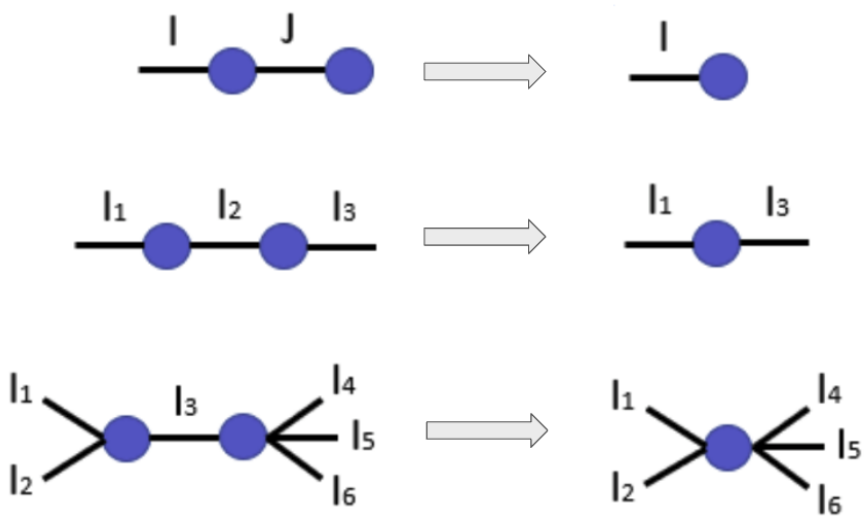

Figure 3.3: Graphical representation of the matrix-vector multiplication on the top, where the matrix is a I by J matrix and the vector has J element. After the matrix-vector multiplication, the result will only remain with one arm. The matrix-matrix multiplication in the middle shows is a I_1 by I_2 matrix multiplied by a I_2 by I_3 matrix. The result will be a I_1 by I_3 matrix. The last is the tensor contraction in the bottom, showing a contraction between a 3^{rd} order tensor and a 4^{th} order tensor. The result is a 5^{th} order tensor [16].

3.3 Basic Tensor Operations

Tensor operations are basic operations in tensor algebra. It can be as simple as a vector-vector multiplication, and it can also be as hard as the tensor product between two higher-order tensors. The reason for tensorizing our data into tensor format is to allow us to implement tensor operations on the data. A tensor operation can be more computationally efficient and easier to perform comparing implementing the same operation on the raw data directly.

3.3.1 Multi-Indices

Since we are dealing with high dimensional tensors with a lot of indices, multi-indices is a useful tool. Multi-indices compute all possible arrangements of the indices, and it used to reorganize the tensor elements after the tensor operation. The multi-indices operation is donated by

$$i = \overline{i_1 i_2 \dots i_N}, \tag{3.1}$$

where $\overline{i_1 i_2 \dots i_N}$ are all the possible combination of the indices $i_1 i_2 \dots i_N$ with $i_n = 1, 2, \dots, I_n$ and $n = 1, 2, \dots, N$. The two most common type of multi-indices are the little-endian and the big-endian. The mathematical representation of these two multi-indices are given as follows [16].

The little-endian is computed as

$$\overline{i_1 i_2 \dots i_N} = i_1 + (i_2 - 1)I_1 + (i_3 - 1)I_1 I_2 + \cdots + (i_N - 1)I_1 \dots I_{N-1}. \tag{3.2}$$

The big-endian is computed as

$$\overline{i_1 i_2 \dots i_N} = i_N + (i_{N-1} - 1)I_N + (i_{N-2} - 1)I_N I_{N-1} + \cdots + (i_i - 1)I_2 \dots I_N. \tag{3.3}$$

All tensor operations are consistent with one of these multi-indices. In this thesis, unless otherwise stated, we will use the little-endian indices.

3.3.2 Matricization

Tensor matricization or tensor flattening is an operation to flatten a tensor into a 2^{nd} order tensor. The most common matricization is the mode-n matricization, where one selects a mode-n and flattens the tensor according to that mode. For example, given an N^{th} order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$, the mode-n matricization of $\boldsymbol{\mathcal{X}}$ is given by

$$\boldsymbol{\mathcal{X}}_{(n)} \in \mathbb{R}^{I_n \times I_1 I_2 \dots I_{n-1} I_{n+1} \dots I_N}, \tag{3.4}$$

where the matrix $\boldsymbol{\mathcal{X}}_{(n)}$ has I_n rows and $I_1 I_2 \dots I_{n-1} I_{n+1} \dots I_N$ columns.

3.3.3 Tensorization

The tensorization of a vector or a matrix is like the reverse process of the tensor matricization. It is an operation to tensorize a vector or a matrix to a higher order tensor. For example, a vector can be tensorized into a matrix, then tensorized again into a 3rd order tensor. A example of the process is shown in Fig. 3.4.

3.3.4 Tensor Product

The tensor product is an operation between two tensors. Given two tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_m \times \cdots \times J_M}$, the tensor product is denoted by $\circ$ and the result $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \circ \boldsymbol{\mathcal{Y}}$ is given by

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \circ \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N \times J_1 \times \cdots \times J_m \times \dots J_M}. \tag{3.5}$$

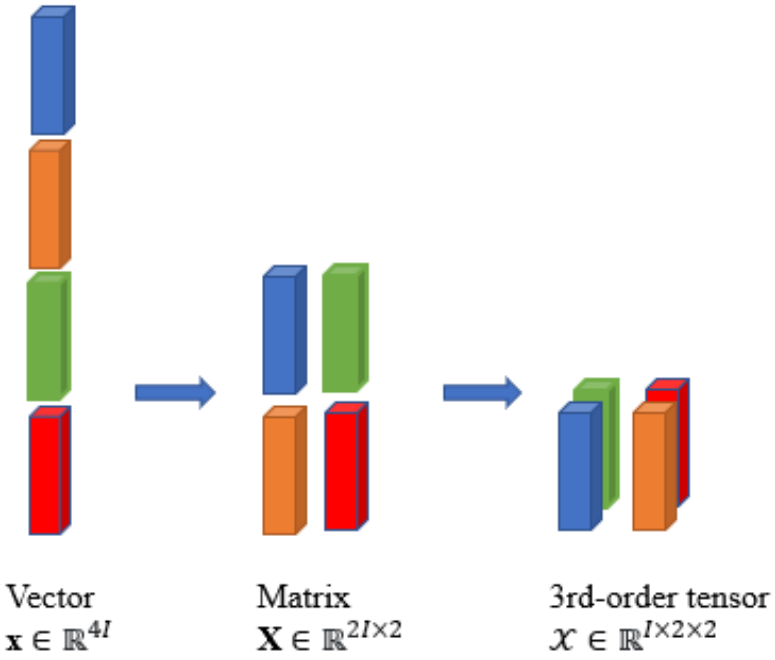

Figure 3.4: Let us consider a vector with $4 \times I$ element where the I is an arbitrary integer. This vector can be reorganized as a matrix or a 3^{rd} order tensor [16]. The process is know as tensorization.

3.3.5 Mode-n Product

The mode-n product or the tensor-times-matrix product is a tensor product between a tensor and a matrix. Given a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times\cdots\times I_n\times\cdots\times I_N}$ and a matrix $\boldsymbol{Y} \in \mathbb{R}^{J\times I_n}$, the mode-n product is denoted by $\times_n$ and the result $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \times_n \boldsymbol{Y}$ is given by

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \times_n \boldsymbol{Y} \in \mathbb{R}^{I_1\times\cdots\times I_{n-1}\times J\times I_{n+1}\times\cdots\times I_N}. \tag{3.6}$$

3.3.6 Contracted Product

The contraction of a tensor is very similar to mode-n product but operates on two tensors instead of a tensor and a matrix. Given two tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1\times\cdots\times I_n\times\cdots\times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1\times\cdots\times J_m\times\cdots\times J_M}$ with common mode $I_n = J_m$, the contraction product between those two tensors is denoted by $\times_n^m$ and the result $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}}\times_n^m\boldsymbol{Y}$ is given by

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}}\times_n^m\boldsymbol{Y} \in \mathbb{R}^{I_1\times\cdots\times I_{n-1}\times I_{n+1}\times\cdots\times I_N\times J_1\times\cdots\times J_{m-1}\times J_{m+1}\times\cdots\times J_M}. \tag{3.7}$$

3.3.7 Convolution

Convolution is one of the most important operations in signal processsing. Given two multi-dimensional tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$, the convolution is denoted by $*$. The operation of $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} * \boldsymbol{\mathcal{Y}}$ is given by

$$\boldsymbol{\mathcal{P}}[i, j, \dots] = \sum_x \sum_y \cdots \sum \boldsymbol{\mathcal{Y}}[x, y, \dots] \boldsymbol{\mathcal{X}}[i - x, j - y, \dots], \tag{3.8}$$

where x and y are index to shitf the tensor $\boldsymbol{\mathcal{X}}$ and tensor $\boldsymbol{\mathcal{Y}}$ along its dimension, i and j are index of the result $\boldsymbol{\mathcal{P}}$ along its dimension The result of convolution has output of $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{(I_1+J_1-1) \times \cdots \times (I_n+J_n-1) \times \cdots \times (I_N+J_N-1)}$.

3.3.8 Partial Mode-n Convolution

The partial (mode-n) convolution is a special tensor operation donate by $\boxed{*}_n$. Given two tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$, the partial mode-n convolution is given by [16]

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \boxed{*}_n \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{(I_1 J_1) \times \cdots \times (I_n + J_n - 1) \times \cdots \times (I_N J_N)}, \tag{3.9}$$

where the subtensor of $\boldsymbol{\mathcal{P}}(f_1, \dots, :, \dots, f_N) = \boldsymbol{\mathcal{X}}(i_1, \dots, :, \dots, i_N) * \boldsymbol{\mathcal{Y}}(j_1, \dots, :, \dots, j_N) \in \mathbb{R}^{I_n + J_n - 1}$ with the multi-indices $f = \overline{ij}$.

3.3.9 Correlation

Correlation is another basic signal processing operation. Given two multi-dimensional tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$, the correlation of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ is a means of comparing how similar they are. The correlation is denoted by $\bigotimes$ and the operation of $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \bigotimes \boldsymbol{\mathcal{Y}}$ is given by

$$\boldsymbol{\mathcal{P}}[i, j, \dots] = \sum_x \sum_y \cdots \sum \boldsymbol{\mathcal{Y}}[x, y, \dots] \boldsymbol{\mathcal{X}}[i + x, j + y, \dots], \tag{3.10}$$

where x and y are index to shitf the tensor $\boldsymbol{\mathcal{X}}$ and tensor $\boldsymbol{\mathcal{Y}}$ along its dimension, i and j are index of the result $\boldsymbol{\mathcal{P}}$ along its dimension. If it is a fully correlation, where we fill the zero padding on the boundary, the result of the correlation is $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{(I_1+J_1-1) \times \cdots \times (I_n+J_n-1) \times \cdots \times (I_N+J_N-1)}$. However, if we use the fully overlapping correlation, where we ignore the boundary, the result is $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{(I_1-J_1+1) \times \cdots \times (I_n-J_n+1) \times \cdots \times (I_N-J_N+1)}$. A simple fully overlapping correlation was shown in Fig.2.7, which is the same fully overlapping correlation in the convolutional layer in the CNNs.

3.3.10 Partial Mode-n Correlation

The partial mode-n fully overlapping correlation is a tensor operation denoted by $\boxed{\otimes}_n$. Given two tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$, the partial mode-n Correlation is given by Cichocki [16]

$$\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \boxed{\otimes}_n \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{(I_1 J_1) \times \cdots \times (I_n - J_n + 1) \times \cdots \times (I_N J_N)}, \tag{3.11}$$

where the subtensor of $\boldsymbol{\mathcal{P}}(f_1, \ldots, :, \ldots, f_N) = \boldsymbol{\mathcal{X}}(i_1, \ldots, :, \ldots, i_N) \bigotimes \boldsymbol{\mathcal{Y}}(j_1, \ldots, :, \ldots, j_N) \in \mathbb{R}^{I_n - J_n + 1}$ with the multi-indices $f = \overline{ij}$.

3.4 Tensor Decompositions

The concept of tensor decomposition was first introduced in 1927 by Hitchcock [15], and the tensor decomposition with a multiway model was given by Cattell in 1944 [17, 18]. Tensor decomposition become popular in the field of chemometrics when Appellof and Davidson first used this technique in 1981 [19]. In the last decade, interest in tensor decomposition is being raising in different fields. For example, in signal processing [20, 21], computer vision [22, 23] and machine learning [6, 8]. In this section, we will discuss the difficulty of the high dimensional computation, and how can we deal with it by using tensor decomposition methods such as the canonical polyadic decomposition, the Tucker decomposition, and tensor train decomposition.

3.4.1 Curse of Dimensionality

When we deal with high dimensional tensors, we will suffer from the curse of dimensionality. The definition of the curse of dimensionality is that the total number of elements of a tensor will increase exponentially as the number of dimensions increases. For example, an N^{th} order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ has a total number of elements $I_1 \times \cdots \times I_n \times \cdots \times I_N$. Since the number of elements increases exponentially with dimension, the storage consumption is getting bigger when we deal with large-scale or high dimension problems. One solution to mitigate the curse of dimensionality in high dimension problems is to use low-rank tensor representations or tensor decomposition.

3.4.2 Rank One Tensor

A rank-one tensor is the simplest form of tensor decomposition, it is a tensor that is represented by a set of vectors through the tensor product. For example, a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ is a rank-one tensor if it can be written as

$$\boldsymbol{\mathcal{A}} = \boldsymbol{a^{(1)}} \circ \cdots \circ \boldsymbol{a^{(n)}} \circ \cdots \circ \boldsymbol{a^{(N)}}, \tag{3.12}$$

where $\boldsymbol{a^{(n)}} \in \mathbb{R}^{I_n}$ are non-zero vectors, and the operation $\circ$ is the tensor product between vectors as defined in Section 3.3.4. A graphical representation of the rank one tensor is shown in Fig 3.5. The rank-one tensor is good for most low-rank tensors. However, when we handle high order tensors we could extend the idea of the rank-one tensor for more complex tensor decompositions such as the canonical polyadic decomposition.

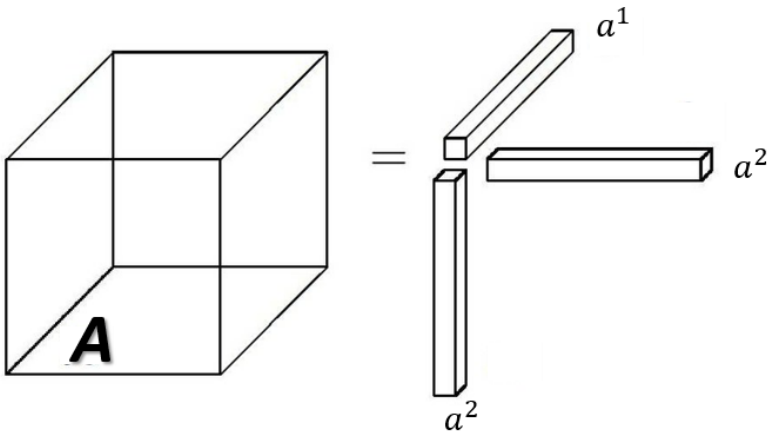

Figure 3.5: A 3rd order tensor $\boldsymbol{\mathcal{A}}$ is a rank one tensor, if we can reconstruct the tensor by using vectors $\boldsymbol{a^{(1)}}$, $\boldsymbol{a^{(2)}}$ and $\boldsymbol{a^{(3)}}$ with a tensor product. In this representation, each vector contains the unique information of the tensor $\boldsymbol{\mathcal{A}}$ along each dimension [24].

3.4.3 The Canonical Polyadic Decomposition

The canonical polyadic decomposition or CPD is a very popular tensor decomposition technique that is using the idea of multi rank-one tensors to represent a high order tensor. The CPD is used to represent a tensor by a summation of a set of rank-one tensors, as shown in Fig 3.14. For example, an N^{th} order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ is decomposed into the CPD format if

$$\boldsymbol{\mathcal{A}} \cong \sum_{r=1}^{R} \lambda_r \boldsymbol{a_r^{(1)}} \circ \cdots \circ \boldsymbol{a_r^{(n)}} \circ \cdots \circ \boldsymbol{a_r^{(N)}}, \tag{3.13}$$

where λ_r is a weighted scalar, $\boldsymbol{a_r^{(n)}}$ are non-zero vectors, and the R is called the canonical rank of the decomposition. The CPD format can also be rewritten as a diagonal N^{th} order tensor core $\boldsymbol{\mathcal{G}}$ which contains those scalars λ_r and a number of factor matrices $\boldsymbol{A}$ made by vectors $\boldsymbol{a_r^{(n)}}$ with mode-n product. It is written as

$$\boldsymbol{\mathcal{A}} \cong \boldsymbol{\mathcal{G}} \times_1 \boldsymbol{A^{(1)}} \cdots \times_n \boldsymbol{A^{(n)}} \cdots \times_N \boldsymbol{A^{(N)}}, \tag{3.14}$$

where the tensor core $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{R \times R \times \cdots \times R}$ is non-zero except on the diagonal, which is made with the weight scalars λ_r. These factor matrices $\boldsymbol{A^{(n)}} = [\boldsymbol{a_1^{(n)}}, \boldsymbol{a_2^{(n)}}, \ldots, \boldsymbol{a_R^{(n)}}] \in \mathbb{R}^{I_n \times R}$ are constructed by vectors.

The benefit of the CPD format is that it only takes $N + \sum_{n=1}^{N}(RI_n)$ memory to store the tensor $\boldsymbol{\mathcal{A}}$. The storage consumption is linear with the number of dimensions N, thus the CPD format can mitigate the curse of dimensionality. However, the canonical format is numerically unstable and may not always exist.

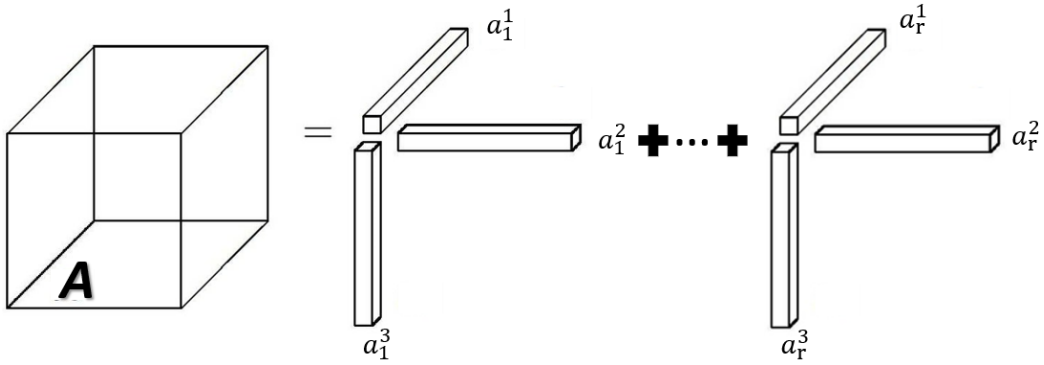

Figure 3.6: The CPD format of a 3rd order tensor that is made with a summation of a numbers of rank-one tensors. Each rank-one tensor contains some information of the original tensor. A we add infinite number of the rank-one tensors, we will have the exact tensor as the original tensor [24]

3.4.4 The Tucker Decomposition

The Tucker decomposition is a more stable technique than the CPD, and always exists for any tensor. The Tucker format of N-th order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ is also given by a summation of rank-one tensors:

$$\boldsymbol{\mathcal{A}} \cong \sum_{r_1=1}^{R_1} \cdots \sum_{r_N=1}^{R_N} \lambda_{r_1,\ldots,r_N} \boldsymbol{a_{r_1}^{(1)}} \circ \cdots \circ \boldsymbol{a_{r_n}^{(n)}} \circ \cdots \circ \boldsymbol{a_{r_N}^{(N)}}, \tag{3.15}$$

where $\lambda_{r_1,\ldots,r_N}$ is the weight scalar, $\boldsymbol{a_{r_n}^{(n)}}$ are non-zero vectors, and the collection $(R_1, \ldots, R_N)$ is called the Tucker ranks. We can rewrite this Tucker format as more general representation as

$$\boldsymbol{\mathcal{A}} \cong \boldsymbol{\mathcal{G}} \times_1 \boldsymbol{A^{(1)}} \cdots \times_n \boldsymbol{A^{(n)}} \cdots \times_N \boldsymbol{A^{(N)}}, \tag{3.16}$$

where $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_N}$ is called the Tucker core, which is made up by the weight scalars $\lambda_{r_1,\ldots,r_N}$. These factor matrices $\boldsymbol{A^{(n)}} = [\boldsymbol{a_1^{(n)}}, \boldsymbol{a_2^{(n)}}, \ldots, \boldsymbol{a_{R_n}^{(n)}}] \in \mathbb{R}^{I_n \times R_n}$ are constructed by these vectors $\boldsymbol{a_{r_n}^{(n)}}$. A graphic representation of the Tucker format with tensor core and factor matrices is shown in Fig 3.7. The CPD format can be seen as a special case of Tucker format with a diagonal tensor core $\boldsymbol{\mathcal{G}}$. The Tucker format is useful since it is always exists as a tensor decomposition. However, it takes $\prod_{n=1}^{N} R_n + \sum_{n=1}^{N}(R_n I_n)$ memory, which does not mitigate the curse of dimensionality. As a result, the Tucker format is only good when the tensor order is small.

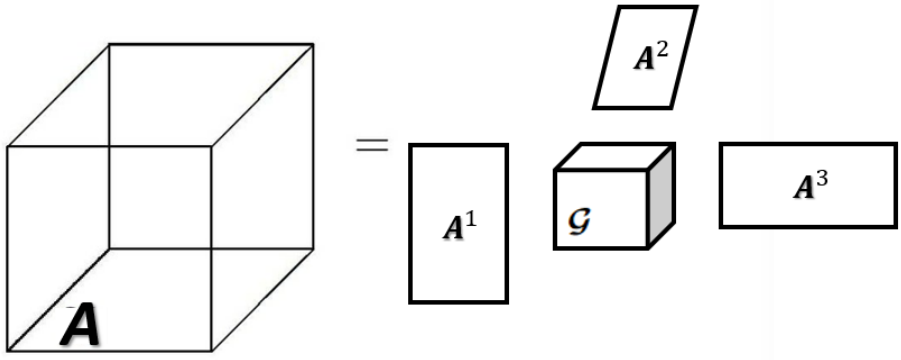

Figure 3.7: A 3-rd order tensor can be represented in the Tucker format with a tensor core with 3 factor matrices. The tensor core contains all the scaler weights which is used to weight the factor matrices [24].

3.4.5 The Hierarchical Tucker Decomposition

Since the Tucker format is not able to mitigate the curse of dimensionality, one solution to overcome this problem is to generalize a tree-structured tensor format called the hierarchical Tucker decomposition or the hierarchical tensor representation [25]. The hierarchical Tucker decomposition splits the set of modes of a tensor in a hierarchical way. In this tree-structured tensor format, a tensor is decomposed into matrices and smaller tensor cores, where those matrices are at the end leaves and the tensor core is like tree branches to connect those leaves with other branches. A graphic representation of the hierarchical Tucker decomposition is shown in Fig 3.8. The hierarchical Tucker format has the advantages of both the canonical format and The Tucker format, where it mitigates the curse of dimensionality and it is numerically stable.

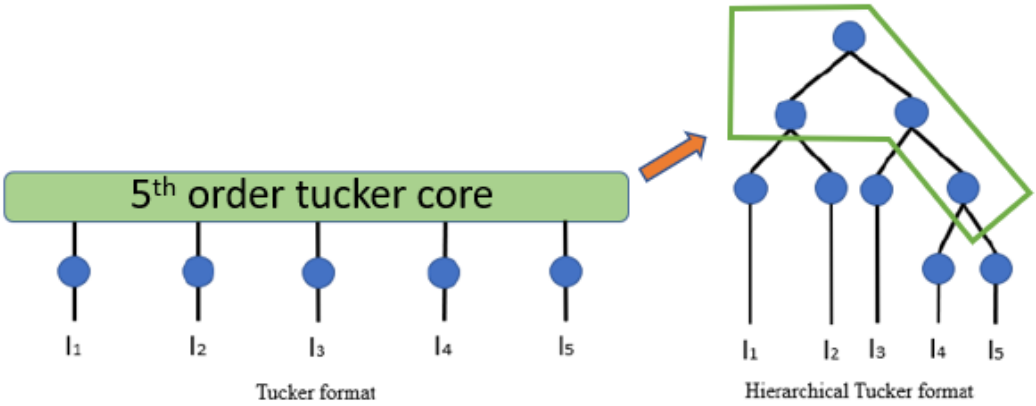

Figure 3.8: Graphical representation of the Tucker format on the left-hand side. The top green bar represents the Tucker tensor core, and the matrices under the green bar are the factor matrices. The Hierarchical Tucker format is represented on the right hand side, which has no high order tensor cores or the green bar. The Hierarchical Tucker format is constructed by 3^{rd} order tensor and 2^{nd} order tensor [25].

3.4.6 The Tensor Train Decomposition

The tensor train (TT) decomposition is a special case of the hierarchical Tucker format where the tree structure only has one main branch as shown in Fig.3.9.

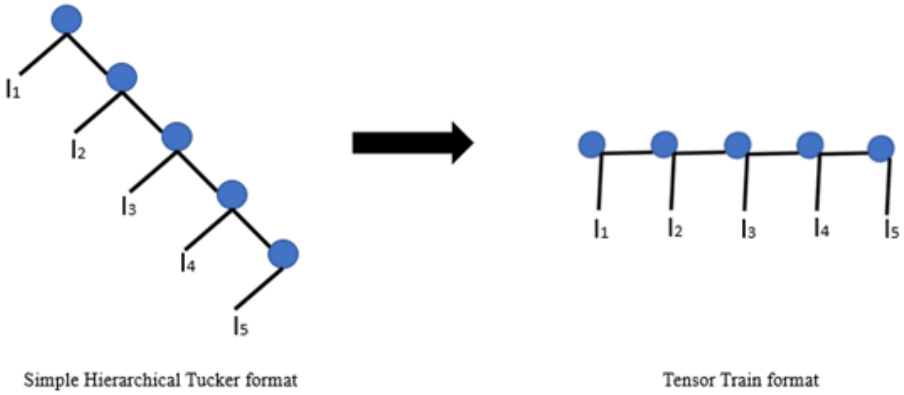

Figure 3.9: Graphical representation of the tensor train format for a 5^{th} order tensor. The tensor train is the simplest hierarchical Tucker format and has only one branch. It is constructed by two 2^{nd} order tensor with three 3^{rd} order tensor to represent the 5^{th} order tensor [25].

The TT format is no different than the canonical format and tucker format, they are all made by a summation of a bunch of rank-one tensors. For example, the TT format of an N-th order tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ is given by

$$\boldsymbol{\mathcal{A}} \cong \sum_{r_1=1}^{R_1} \cdots \sum_{r_{N-1}=1}^{R_{N-1}} \boldsymbol{a}^{(1)}_{\mathbf{1},\boldsymbol{r_1}} \circ \cdots \circ \boldsymbol{a}^{(n)}_{\boldsymbol{r_n},\boldsymbol{r_{n+1}}} \circ \cdots \circ \boldsymbol{a}^{(N)}_{\boldsymbol{r_{N-1}},\mathbf{1}}, \tag{3.17}$$

where $\boldsymbol{a}^{(n)}_{r_n,r_{n+1}} \in \mathbb{R}^{I_n}$ are non-zero vectors, $R_0 = R_N = 1$ by the constraints of the tensor train decomposition, and the collection of $(R_1, \ldots, R_{N-1})$ is called the TT ranks. We can rewrite this TT format as more general representation as

$$\boldsymbol{\mathcal{A}} \cong \boldsymbol{\mathcal{G}}^{(1)} \times^1 \boldsymbol{\mathcal{G}}^{(2)} \ldots \times^1 \boldsymbol{\mathcal{G}}^{(N-1)} \times^1 \boldsymbol{\mathcal{G}}^{(N)}, \tag{3.18}$$

where $\boldsymbol{\mathcal{G}}^{(n)} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ are TT cores with multilinear product $\times^1$. Since $R_0 = R_N = 1$ the first tensor core and the last tensor core can be considered as matrices, where they are made by

$$\boldsymbol{\mathcal{G}}^{(1)} = [\boldsymbol{a}^{(1)}_{1,1}, \ldots, \boldsymbol{a}^{(1)}_{1,r_1}, \ldots, \boldsymbol{a}^{(1)}_{1,R_1}] \in \mathbb{R}^{I_1 \times R_1}, \tag{3.19}$$

$$\boldsymbol{\mathcal{G}}^{(N)} = [\boldsymbol{a}^{(N)}_{1,1}, \ldots, \boldsymbol{a}^{(N)}_{r_N,1}, \ldots, \boldsymbol{a}^{(N)}_{R_N,1}] \in \mathbb{R}^{I_N \times R_{N-1}}. \tag{3.20}$$

The rest of the tensor cores $\boldsymbol{\mathcal{G}}^{(n)}$ in mode-2 matricization are constructed as

$$\boldsymbol{\mathcal{G}}^{(n)}_{(2)} = [\boldsymbol{a}^{(n)}_{1,1}, \boldsymbol{a}^{(n)}_{2,1}, \ldots, \boldsymbol{a}^{(n)}_{r_{n-1},1}, \boldsymbol{a}^{(n)}_{1,2}, \ldots, \boldsymbol{a}^{(n)}_{R_{n-1},R_n}] \in \mathbb{R}^{I_n \times R_{n-1} R_n}. \tag{3.21}$$

A graphically representation of the TT format is shown in Fig 3.10. The storage consumption of TT format of N-th order tensor $\boldsymbol{\mathcal{A}}$ is $\sum_{n=1}^{N}(R_{n-1} I_n R_n)$ which is not increasing exponentially.

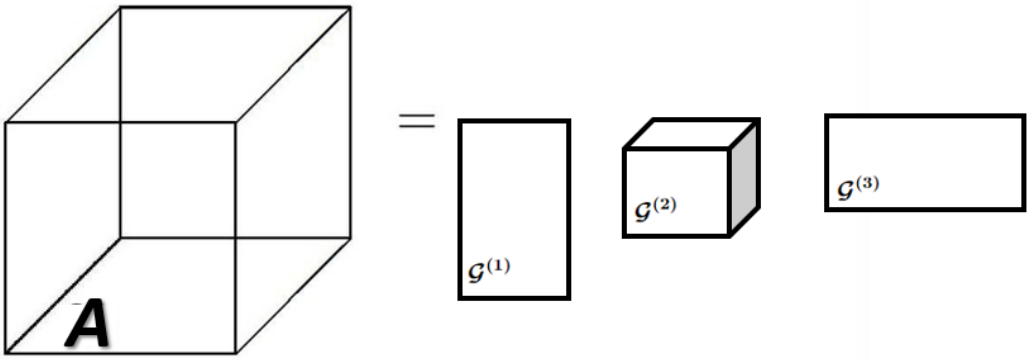

Figure 3.10: The tensor train format of a 3rd order tensor can be represented by 3 tensor cores. The first tensor core is a 2^{nd} order tensor, the second tensor core is a 3^{rd} order tensor, and the last tensor core is a 2^{nd} order tensor [25].

3.4.7 The Quantized Tensor Train Decomposition

Let us consider a big vector or matrix, that we want to decompose with the TT decomposition. However, decomposing a vector or matrix directly into a TT format is the same as the low-rank approximation. If we want to take advantage of all the benefits of the TT format, one needs to use the quantized TT format instead of the general TT format. The idea of the quantized TT format is simple: one just needs to take the vector or matrix and rearrange it into a high order tensor by using the tensorization method. Once they are high order tensors, we now can decompose the high order tensor with Tensor Train decomposition. For example, given a vector $\boldsymbol{V} \in \mathbb{R}^I$ where $I = 2^6$, this vector can be rewritten as a 6^{th} order tensor as $\boldsymbol{\mathcal{V}} \in \mathbb{R}^{2\times2\times2\times2\times2\times2}$. Now, the total number of elements in this tensor is 2^6, but the order of the tensor is 6. So, we can decompose this tensor into TT format with a 6^{th} order tensor instead of a 1^{st} order tensor. The conversion of a vector to a 6^{th} order tensor is shown in Fig 3.11.

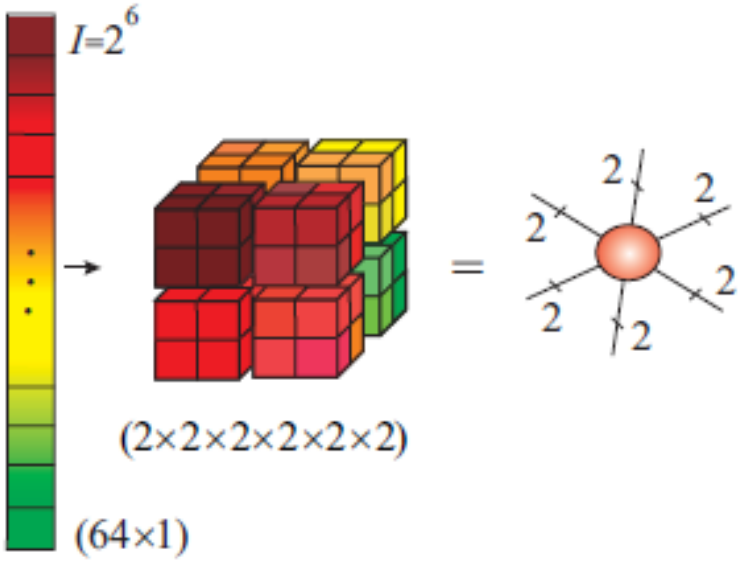

Figure 3.11: The plot shows that initially we have a 1^{st} order tensor $\boldsymbol{V} \in \mathbb{R}^I$ where $I = 2^6$. The 1^{st} order tensor is converted to a 6^{th} order tensor represented in the middle. The graphical representation of the 6^{th} order tensor is shown on the right hand side [26].

Chapter 4

Bayesian Neural Network

The definition of a Bayesian neural network is a stochastic ANN that is trained under Bayesian inference [27]. The main difference between a conventional neural network and the Bayesian neural network (BNN) is that we assume the weights of the conventional neural network are deterministic but those in the BNN are probabilistic. The BNN can be represented mathematically as

$$\boldsymbol{\theta} \sim Pr(\boldsymbol{\theta}), \tag{4.1}$$

$$\boldsymbol{y} = \boldsymbol{NN}(\boldsymbol{\theta}, \boldsymbol{x}) + \boldsymbol{v}, \tag{4.2}$$

where $\boldsymbol{\theta}$ is the neural network parameter sampled from a probability distribution function $Pr(\boldsymbol{\theta})$, $\boldsymbol{NN}()$ is the approximation function of the neural network, $\boldsymbol{x}$ and $\boldsymbol{y}$ are the input and output of the neural network, and $\boldsymbol{v}$ is random noise. For the BNN trained under Bayesian inference, we try to estimate the parameter $\boldsymbol{\theta} = \{\boldsymbol{W}, \boldsymbol{b}\}$ based on the given training data $\boldsymbol{D} = \{\boldsymbol{x}, \boldsymbol{y}\}$, where $\boldsymbol{W}$ and $\boldsymbol{b}$ are the layer weights and bias.

4.1 Derivation of the Recursive Bayesian Solution

BNN training via Bayesian inference is based on Bayes' theorem. Let us consider two events A and B. The joint probability of A and B, or $Pr(A, B)$, is given by

$$Pr(A, B) = Pr(A|B)Pr(B) - Pr(B|A)Pr(A). \tag{4.3}$$

By using Eq 4.3 twice, Bayes' theorem is given by

$$Pr(A|B) = \frac{Pr(B|A)Pr(A)}{Pr(B)}. \tag{4.4}$$

In this thesis, we mainly focus on the parameter estimation of the BNN. The probabilistic information describing the BNN's parameters is given by the joint probability distribution function $Pr(\theta_{0:K}) = Pr(\theta_0, \theta_1, \dots \theta_k, \dots, \theta_K)$, where θ_k is the parameter state at time $t = k$ of the training step. To train the BNN, we are given a set of training data, $D_{1:K} = (D_1, D_2, \dots, D_k, \dots, D_K)$, where every training data D_k is a pair of output and input $D_k = (y_k, x_k)$. We want to fit our BNN with the training data.

The training process can be described based the Bayes' theorem as

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{D_{1:K}}) = \frac{Pr(\boldsymbol{D_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{D_{1:K}})}, \tag{4.5}$$

since the training data $\boldsymbol{D_{1:K}}$ is made by a pair of output and input. Therefore, by substituting $\boldsymbol{D_{1:K} = (y_{1:K}, x_{1:K})}$ into Eq 4.5 we get

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}, \tag{4.6}$$

since $Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}}) = Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})Pr(\boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})$, and the input $\boldsymbol{x}$ is independent of the network parameter $\boldsymbol{\theta}$. Given that $Pr(\boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}}) = Pr(\boldsymbol{x_{1:K}})$, the above equation can be rewritten as

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})Pr(\boldsymbol{x_{1:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}})Pr(\boldsymbol{x_{1:K}})}. \tag{4.7}$$

By cancelling both $Pr(\boldsymbol{x_{1:K}})$ in the numerator and denominator, we end up with following equation

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}})}, \tag{4.8}$$

where the $Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})$ is the likelihood function, $Pr(\boldsymbol{\theta_{0:K}})$ is the prior function and $Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}})$ is the evidence function. Our goal is to compute the posterior function $Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$ which contains the information of how likely our parameter is given to the training data set.

To compute the posterior function, let us first consider expanding following functions.

The likelihood function:

$$\begin{aligned} Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) &= Pr(\boldsymbol{y_K}, \boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) \\ &= Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}). \end{aligned} \tag{4.9}$$

The prior function:

$$\begin{aligned} Pr(\boldsymbol{\theta_{0:K}}) &= Pr(\boldsymbol{\theta_K}, \boldsymbol{\theta_{0:K-1}}) \\ &= Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}})Pr(\boldsymbol{\theta_{0:K-1}}). \end{aligned} \tag{4.10}$$

And the evidence function:

$$\begin{aligned} Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}) &= Pr(\boldsymbol{y_K}, \boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K}}) \\ &= Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}) Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K}}). \end{aligned} \tag{4.11}$$

Since the output $y_{t=k-1}$ at step $t = t-1$ does not depend on the input $x_{t\geq k}$ and the BNN's parameter $\theta_{t\geq k}$, therefore we can write

$$Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) = Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}, \boldsymbol{\theta_{0:K-1}}). \tag{4.12}$$

By using the above simplified equation, we can further simplify the likelihood function Eq 4.9 and the evidence function Eq 4.11 as

$$\begin{aligned} Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) &= Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) \times \\ & \quad Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}, \boldsymbol{\theta_{0:K-1}}), \end{aligned} \tag{4.13}$$

and

$$\begin{aligned} Pr(\boldsymbol{y_{1:K}}|\boldsymbol{x_{1:K}}) &= Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}) \times \\ & \quad Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}). \end{aligned} \tag{4.14}$$

Substituting Eq 4.10, Eq 4.13 and Eq 4.14 back into Eq 4.8, the Bayesian training process can be represented as

$$\begin{aligned} Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) &= \frac{[Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}, \boldsymbol{\theta_{0:K-1}})]}{Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}) Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}})} \times \\ & \quad [Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}}) Pr(\boldsymbol{\theta_{0:K-1}})]. \end{aligned} \tag{4.15}$$

Rearranging these terms, we end up with following equation

$$\begin{aligned} Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) &= \frac{Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}}) Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}})}{Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}})} \times \\ & \quad [\frac{Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}, \boldsymbol{\theta_{0:K-1}}) Pr(\boldsymbol{\theta_{0:K-1}})}{Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}})}], \end{aligned} \tag{4.16}$$

where the last term in the bracket is the previous posterior estimation given by

$$Pr(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) = [\frac{Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}}, \boldsymbol{\theta_{0:K-1}}) Pr(\boldsymbol{\theta_{0:K-1}})}{Pr(\boldsymbol{y_{1:K-1}}|\boldsymbol{x_{1:K-1}})}]. \tag{4.17}$$

Now, we can recursively update our posterior function by using the previous estimation. We will refer to the following equation as the recursive Bayesian solution of the Bayesian training process:

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}}, \boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}})}{Pr(\boldsymbol{y_K}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K}})} \times Pr(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}). \tag{4.18}$$

However, the output $y_{t=k}$ at $t = k$ only depends on the current input x_k and the network parameter $\boldsymbol{\theta_{t=k}}$, and it is also independent of any output $y_{t\leq k-1}$. In addition, the transition of the network parameter has the Markov property which the current neural network's parameter θ_k only depends on the previous parameter θ_{k-1}. With the above assumptions, we can rewrite the Bayesian recursive solution as

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_K}|\boldsymbol{x_K}, \boldsymbol{\theta_K})Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}})}{Pr(\boldsymbol{y_K}|\boldsymbol{x_K})} \times Pr(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}). \tag{4.19}$$

In the network training process, the history of the network parameter evolution in the training process is not important to us. Our interest is the final trained parameter θ_K that fits our training data $\boldsymbol{D_k} = \boldsymbol{(y_k, x_k)}$ with confidence. Moreover, tracking all the parameter update histories is computationally inefficient and more complex. One solution is to integrate those terms off from our recursive Bayesian solution and leave the final probabilistic knowledge of the final trained network parameter θ_K. This can be done by

$$Pr(\boldsymbol{\theta_K}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \int \cdots \int_{\theta_i} Pr(\boldsymbol{\theta_{0:k}}|\boldsymbol{y_{1:k}}, \boldsymbol{x_{1:k}}) d\theta_0 \ldots d\theta_{K-1}. \tag{4.20}$$

We then insert Eq 4.18 into Eq 4.20, and end up with the following equation

$$Pr(\boldsymbol{\theta_K}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_K}|\boldsymbol{x_K}, \boldsymbol{\theta_K})}{Pr(\boldsymbol{y_K}|\boldsymbol{x_K})} \times \int_{\theta} Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{K-1}}) \times [\int \cdots \int_{\theta} Pr(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) d\theta_0 \ldots d\theta_{K-2}] d\theta_{K-1}. \tag{4.21}$$

We can write the bracket term as

$$Pr(\boldsymbol{\theta_{K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) = [\int \cdots \int_{\theta} Pr(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) d\theta_0 \ldots d\theta_{K-2}]. \tag{4.22}$$

Inserting Eq 4.22 back into Eq 4.21, we finally end with the state conditional density of the Bayesian training process given by

$$\begin{aligned} Pr(\boldsymbol{\theta_K}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) &= \frac{Pr(\boldsymbol{y_K}|\boldsymbol{x_K}, \boldsymbol{\theta_K})}{Pr(\boldsymbol{y_K}|\boldsymbol{x_K})} \times \\ & \int_{\theta} Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{K-1}}) Pr(\boldsymbol{\theta_{K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) d\theta_{K-1}, \end{aligned} \tag{4.23}$$

where $Pr(\boldsymbol{y_K}|\boldsymbol{x_K}, \boldsymbol{\theta_K})$ is the likelihood function, $Pr(\boldsymbol{\theta_K}|\boldsymbol{\theta_{0:K-1}})$ is the transition function and $Pr(\boldsymbol{\theta_{K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}})$ is the previous posterior estimation of the network parameter.

It should be noted that the analytical solution for the above equation does not usually exit, or it is hard to compute. However, we can approximate the solutions by using numerical simulations, such as the MC method. In this thesis, the MC method is used to recursively solve the Bayesian solution. Specifically, we will employ the SMC method due to its simple implementation and its ability to simulate the training process sequentially.

4.2 Monte Carlo Integration

In practice, there are many integrals that cannot be solved analytically. Fortunately, MC methods provide a convenient and accurate means to evaluate complex integrals numerically. As noted above in the BNN training, we want to compute the state conditional density of the Bayesian training process. The solution has a series of integrals but cannot be solved analytically. As a result, MC methods are an ideal tool for the BNN training process. Here, we first present how to evaluate a simple integral using MC methods, and then discuss more advanced MC techniques for evaluating the complex integrals of Eq 4.23 such as the SMC method.

Let us first consider approximating a distribution as

$$p(x_{0:t}|y_{1:t}) = \frac{1}{N} \sum_{i=1}^{\infty} \delta_{x_{0:t}^{(i)}}(x_{0:t}), \tag{4.24}$$

where $\delta_{x_{0:t}^{(i)}}$ is the delta-Dirac function, and $x_{0:t}^{(i)}$ are independent and identically distributed random samples from the distribution $p(x_{0:t}|y_{1:t})$.

Now, consider a integral function

$$I = \int g(x_{0:t}) p(x_{0:t}|y_{1:t}) dx_{0:t}. \tag{4.25}$$

Equation 4.25 can be estimated by a MC method as the summation of samples from the probability distribution function $p(x_{0:t}|y_{1:t})$ [9]:

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{\infty} g(x_{0:t}^{(i)}), \tag{4.26}$$

where $x_{0:t}^{(i)}$ is randomly sampled from the target distribution function $p(x_{0:t}|y_{1:t})$, and N is the total number of samples. Numerical methods such as these that rely on random sampling are called Monte Carlo in reference to the famous gambling casino. In the limit $N \to \infty$, $\hat{I} = I$. For finite N, the error for this estimation is given by

$$|e_1| \cong \frac{\sigma_1}{\sqrt{N}}, \tag{4.27}$$

where the variance ${\sigma_1}^2$ is given by [28]

$${\sigma_1}^2 \equiv \int g^2(x_{0:t}) p(x_{0:t}) dx - I^2 \,. \tag{4.28}$$

4.2.1 Importance Sampling

Let us now consider a situation from which the target distribution function $p(x_{0:t}|y_{1:t})$ is difficult to sample. In this case, we evaluate the integral by using the basic MC method above with a new sampling technique known as importance sampling. This approach allows one to sample from a simpler, more convenient, importance function, instead of sampling from the original target distribution. In order to capture the features of $p(x_{0:t}|y_{1:t})$, however, importance weights must be used to statistically weigh how likely the sample would be if it were coming from the target distribution. The idea of Importance sampling is illustrated in Fig. 4.1.

One can rewrite Eq 4.25 as

$$I = \int g(x_{0:t}) p(x_{0:t}|y_{1:t}) dx = \int g(x_{0:t}) \frac{p(x_{0:t}|y_{1:t})}{q(x_{0:t}|y_{1:t})} q(x_{0:t}|y_{1:t}) \, dx_{0:t} \,, \tag{4.29}$$

where $q(x_{0:t}|y_{1:t})$ is the importance function, which is easy to sample from the function $g(x_{0:t})$ and also has to be greater than zero. Letting $w(x_{0:t})$ be the importance weight

$$w(x_{0:t}) = \frac{p(x_{0:t}|y_{1:t})}{q(x_{0:t}|y_{1:t})}. \tag{4.30}$$

Figure 4.1: A target distribution that is hard to sample from represents on the left-hand side. One can use an importance function in the middle, let us consider a uniform distribution that is easy to sample. Therefore, these samples can easily sample from the importance function must include important weight shows on the right-hand side. The important weights are represented by the size of the red spheres, which tells indicates how likely those samples are from the target distribution.

Eq 4.29 can now be rewritten as

$$I = \int [g(x_{0:t})w(x_{0:t})]q(x_{0:t}|y_{1:t})\, dx_{0:t}\,. \tag{4.31}$$

Now, the function in this integral is $[g(x_{0:t})w(x_{0:t})]$ and the distribution function is $q(x_{0:t}|y_{1:t})$. As a result, the MC estimation of this integral can be written as

$$\hat{I} = \frac{1}{N}\sum_{i=1}^{\infty} g(x_{0:t}^{(i)})\tilde{w}(x_t^{(i)}), \tag{4.32}$$

where $x_{0:t}^{(i)}$ are randomly sampled from the importance function $q(x_{0:t}|y_{1:t})$ and $\tilde{w}(x_t^{(i)})$ is the normalized importance weights given by

$$\tilde{w}(x_t^{(i)}) = \frac{w(x_{0:t}^{(i)})}{\sum_{j=1}^{N} w(x_{0:t}^{(j)})}. \tag{4.33}$$

The error of the estimation here is again

$$|e_2| \cong \frac{\sigma_2}{\sqrt{N}}, \tag{4.34}$$

where the variance ${\sigma_2}^2$ is given by [28]

$${\sigma_2}^2 \equiv \int w^2(x_{0:t})q(x_{0:t}|y_{1:t})dx_{0:t} - I^2\,. \tag{4.35}$$

4.3 The Sequential Monte Carlo Method

Let us consider Eq 4.23 again, where we are trying to evaluate this complex equation for the BNN training process. It is a series of integrals that could be evaluated individually using the standard

MC method with importance sampling described above. However, such an approach would be very inefficient, as computational complexity increases at least linearly with dimension [9]. A more suitable evaluation of this kind of equation can be achieved with the sequential Monte Carlo (SMC) method described below, which allows for more efficient use of samples. The graphic representation of the SMC is shown in Fig. 4.2.

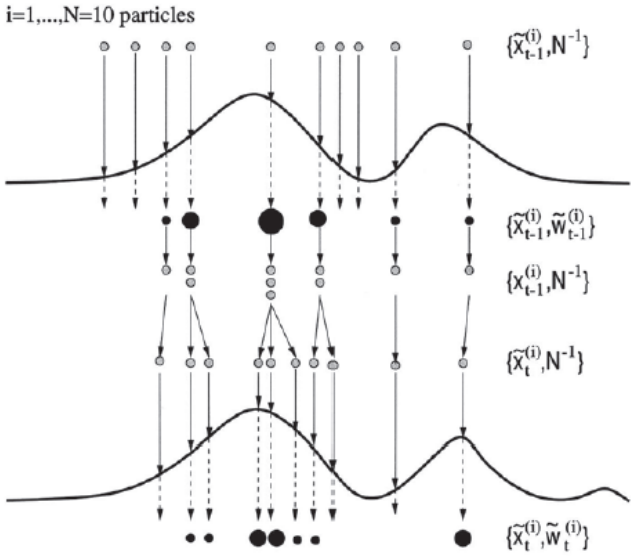

Figure 4.2: This plot shows a simple SMC method to sequentially approximate a distribution function. First, we randomly initialize 10 particles from an importance distribution. The particle will be weighted by the normalized importance weight represented by the size of the circle. Resampling of the particles will be performed to produce more particles with high weight and reduce particles with low weights. After that, these new particles will be used to sequentially approximate the probability distribution [9].

4.3.1 Sequential Importance Sampling

The core idea of SMC is to use what is known as the sequential importance sampling (SIS) technique to evaluate the integrals of Eq 4.23 sequentially [9]. In this case, the numerical samples used to evaluate the previous integral will be reused to evaluate the next integral. In doing so, we can evaluate the integral recursively.

To be able to sequentially sample from the importance function, the importance function $q(x_{0:t}|y_{1:t})$ at time t needs to be a marginal distribution at time $t-1$ the importance function $q(x_{0:t-1}|y_{1:t-1})$ is given by Doucet [9]

$$q(x_{0:t}|y_{1:t}) = p(x_{0:t-1}|y_{1:t-1})q(x_t|x_{0:t-1}|y_{1:t}), \tag{4.36}$$

which is equivalent to

$$q(x_{0:t}|y_{1:t}) = q(x_0)\prod_{k=1}^{t} q(x_t|x_{0:t-1}, y_{1:t}). \tag{4.37}$$

Therefore, one can recursively compute the normalized importance weights by using the previous normalized importance weights as

$$\tilde{w}(x_t^{(i)}) \propto \tilde{w}(x_{t-1}^{(i)})\frac{p(y_t|x_t^{(i)})p(x_t^{(i)}|x_{t-1}^{(i)}))}{q(x_t^{(i)})|q(x_{0:t-1}^{(i)}, y_{i:t})}. \tag{4.38}$$

If we select the prior distribution as the importance distribution, this recursive computation of the normalized importance weight can be simplified as

$$\tilde{w}(x_t^{(i)}) \propto \tilde{w}(x_{t-1}^{(i)})p(y_t|x_t^{(i)}). \tag{4.39}$$

Therefore, we can update the normalized importance weights and reuse these samples to sequentially approximate the probability distribution function. By doing this, we saved the computation of generating new particles and recomputing their importance weights for the estimation.

4.3.2 Resampling Algorithm

The SMC method is a powerful tool for evaluating complex integrals like those from the solution of Eq 4.23. The drawback of the SMC method, however, is an effect known as weight degeneracy, whereby the variance of importance weights increases exponentially as the number of sampling steps [9]. One can fix this problem by using resampling methods. In this section, we will introduce a resampling method and describe how we will implement this method in the training process.

The idea behind resampling is to preserve samples with large weights while discarding those with small weights. The selection probability is based on each sample's normalized weight. Fig. 4.3 illustrates the resampling process. Because we terminate the tracing on those low-weight samples by discarding them earlier, we would expect a reduction in computational time. In addition, by preserving those high-weight samples, one expects an increase in the approximation accuracy.

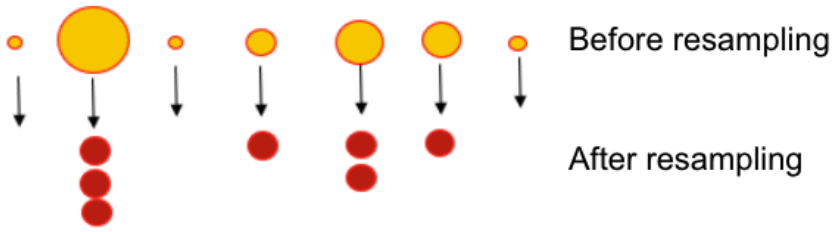

Figure 4.3: Let us consider a set of samples represented by the top spheres, where the size of the sphere is the size of the importance weights. The idea of resampling is to resample the current samples according to the importance weights. The sample with high weight will be reproduced and the small weight sample will be discarded.

There are several algorithms available for resampling method, such as multinomial resampling, systematic resampling, and residual resampling [29]. The simplest resampling method is the multinomial resampling algorithm, since the selection probability of the resampling method depends on the sample's normalized weight. Thus the number of times N_i for each sample in the population set is selected as a binomial distribution Bin(N_i,W_i). For a series of samples, it is distributed according to a multinomial distribution [29]. The multinomial resampling process is illustrated in Fig. 4.4.

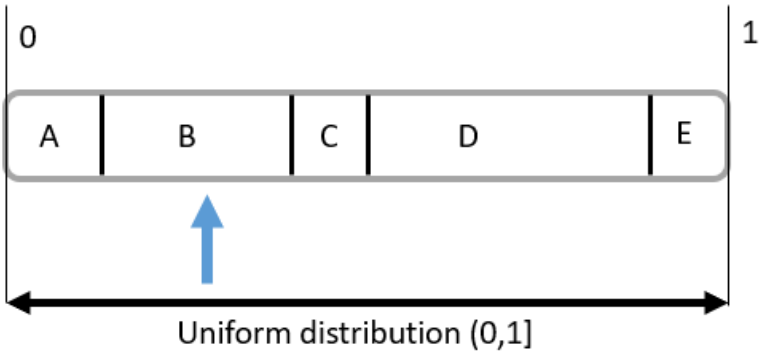

Figure 4.4: Graphical representation of the multinomial resampling process. Samples A B C D and E are stored in a column. The column has space from 0 to 1. These samples occupy the column's space according to their normalized weight. one can select a single sample by randomly generating a number from a uniform distribution $(1, 0]$. The new sample is selected based the number, that mean the sample with large normalized weight will have higher chance to be selected [30].

Chapter 5

Development and Algorithm Implementation

As mentioned previously, any neural network with one or more TT-layers can be referred to as a TensorNet (TN) [6], and a neural network is called a Bayesian neural network (BNN) if it has stochastic network weights and was trained via Bayesian inference [27]. We will therefore refer to any neural network with any TT-layers and trained via Bayesian inference as a Bayesian TensorNet(BTN). In this section, we will present how to represent a CNN into TT-format and train such a network with the TerraByte's plant dataset via Bayesian inference.

5.1 Bayesian TensorNet

In a CNN, the network trainable parameters are stored in the convolutional layers and fully connected layers. To represent the CNN, one needs to represent these layers in TT format.

5.1.1 Correlation Operation in Tensor Train Format

Since we are working on the CNN, the correlation is a important operation especially for the fully overlapping correlation. Given two tensors $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$, we want to compute $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \bigotimes \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{(I_1 - J_1 + 1) \times \cdots \times (I_n - J_n + 1) \times \cdots \times (I_N - J_N + 1)}$ with fully overlapping correlation. To do this in the TT format, let us first write $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$ and $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{J_1 \times \cdots \times J_n \times \cdots \times J_N}$ in TT format with TT rank $(R_0 = 1, R_1, \ldots R_{N-1}, R_N = 1)$ and $(Q_0 = 1, Q_1, \ldots Q_{N-1}, Q_N = 1)$ as follows:

$$\boldsymbol{\mathcal{X}} \cong \boldsymbol{\mathcal{G}^{(1)}} \times^1 \boldsymbol{\mathcal{G}^{(2)}} \ldots \times^1 \boldsymbol{\mathcal{G}^{(N-1)}} \times^1 \boldsymbol{\mathcal{G}^{(N)}}, \tag{5.1}$$

where $\boldsymbol{\mathcal{G}^{(n)}} \in \mathbb{R}^{R_{n-1} \times I_n \times R_n}$ with $(R_0 = 1, R_1, \dots R_{N-1}, R_N = 1)$, and

$$\boldsymbol{\mathcal{Y}} \cong \boldsymbol{\mathcal{H}^{(1)}} \times^1 \boldsymbol{\mathcal{H}^{(2)}} \dots \times^1 \boldsymbol{\mathcal{H}^{(N-1)}} \times^1 \boldsymbol{\mathcal{H}^{(N)}}, \tag{5.2}$$

where $\boldsymbol{\mathcal{H}^{(n)}} \in \mathbb{R}^{Q_{n-1} \times J_n \times Q_n}$ with $(Q_0 = 1, Q_1, \dots Q_{N-1}, Q_N = 1)$. Fig. 5.1 shows the graphical transformation from an N^{th} order tensor into N number of small tensor cores.

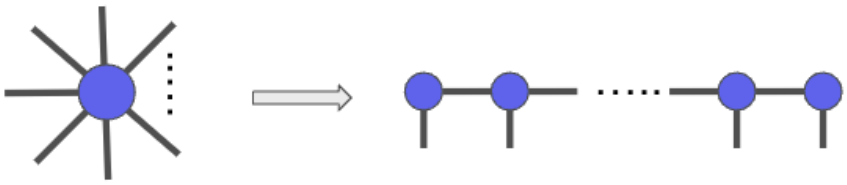

Figure 5.1: Let us consider an N^{th} order tensor such as $\boldsymbol{\mathcal{X}}$ represented on the left hand side with N arms. the N^{th} order tensor can be represent by using N tensor cores on the right hand side. The first and the last tensor cores are 2^{nd} order tensors with only two arms. The rest of them are 3^{rd} order tensors with three arms.

For computing $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \bigotimes \boldsymbol{\mathcal{Y}}$, the TT format of $\boldsymbol{\mathcal{P}}$ can be calculated TT core by another one TT core with partial mode-2 correlation, such as

$$\boldsymbol{\mathcal{U}^{(n)}} = \boldsymbol{\mathcal{G}^{(n)}} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}^{(n)}} \in \mathbb{R}^{(R_{n-1}Q_{n-1}) \times (I_n - J_n + 1) \times (R_n Q_n)}, \tag{5.3}$$

where $n = 1, 2, \dots N$. Therefore,

$$\boldsymbol{\mathcal{P}} \cong \boldsymbol{\mathcal{U}^{(1)}} \times^1 \boldsymbol{\mathcal{U}^{(2)}} \dots \times^1 \boldsymbol{\mathcal{U}^{(N-1)}} \times^1 \boldsymbol{\mathcal{U}^{(N)}}. \tag{5.4}$$

Once, we have those TT cores ready, the result of the fully overlapping correlation $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{X}} \bigotimes \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{(I_1 - J_1 + 1) \times \dots \times (I_n - J_n + 1) \times \dots \times (I_N - J_N + 1)}$ can be reconstructed by the TT cores $\boldsymbol{\mathcal{U}^{(1)}}, \boldsymbol{\mathcal{U}^{(2)}}, \dots, \boldsymbol{\mathcal{U}^{(N)}}$. The computation of $\boldsymbol{\mathcal{P}}$ is illustrated in Fig. 5.2.

5.1.2 Convolutional Layer in Tensor Train Format

For a convolutional layer in the convolutional nerual network, let us first consider a batch of input images as a 4th order tensor $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{I_1 \times I_2 \times C \times N}$, and a batch of filters as another 4th order tensor $\boldsymbol{\mathcal{F}} \in \mathbb{R}^{J_1 \times J_2 \times C \times M}$. We can represent these 4th orders tensor by two 5th orders tensor with an empty dimension such as $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{I_1 \times I_2 \times C \times N \times 1}$ and $\boldsymbol{\mathcal{F}} \in \mathbb{R}^{J_1 \times J_2 \times C \times 1 \times M}$. As such, they are basically the same tensors but with an additional dimension. We write tensors $\boldsymbol{\mathcal{I}}$ and $\boldsymbol{\mathcal{J}}$ into TT format representation with TT ranks $(R_0, R_1, R_2, R_3, R_4, R_5)$ and $(Q_0, Q_1, Q_2, Q_3, Q_4, Q_5)$ where the first and the last TT rank are 1, so that $R_0 = R_5 = Q_0 = Q_5 = 1$. The TT format of the batch of image $\boldsymbol{\mathcal{I}}$ is then given by

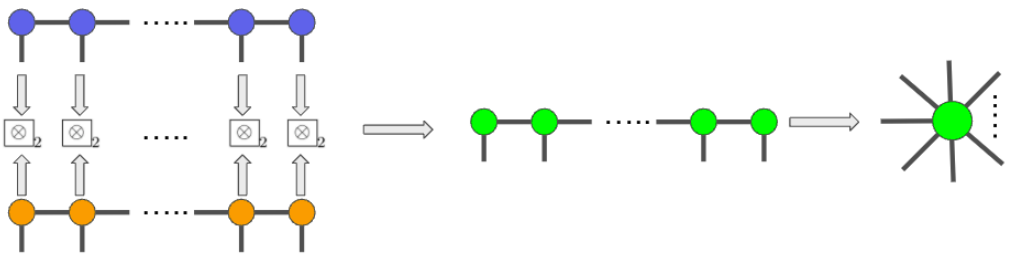

Figure 5.2: On the left hand side, each tensor core $\boldsymbol{\mathcal{G}^{(n)}}$ is used to implement the partial mode-2 correlation with corresponding tensor core $\boldsymbol{\mathcal{H}^{(n)}}$. The result is shows in the middle, which is a TT format constructed by the results of the partial mode-2 correlatio or tensor cores $\boldsymbol{\mathcal{U}^{(n)}}$. The final result is on the right hand side using the tensor cores $\boldsymbol{\mathcal{U}^{(n)}}$ to reconstruct the high order tensor $\boldsymbol{\mathcal{P}}$.

$$\boldsymbol{\mathcal{I}} \cong \boldsymbol{\mathcal{G}^{(1)}} \times^1 \boldsymbol{\mathcal{G}^{(2)}} \times^1 \boldsymbol{\mathcal{G}^{(3)}} \times^1 \boldsymbol{\mathcal{G}^{(4)}} \times^1 \boldsymbol{\mathcal{G}^{(5)}}, \tag{5.5}$$

where

$$\begin{aligned} \boldsymbol{\mathcal{G}^{(1)}} &\in \mathbb{R}^{1\times I_1\times R_1} \\ \boldsymbol{\mathcal{G}^{(2)}} &\in \mathbb{R}^{R_1\times I_2\times R_2} \\ \boldsymbol{\mathcal{G}^{(3)}} &\in \mathbb{R}^{R_2\times C\times R_3} \\ \boldsymbol{\mathcal{G}^{(4)}} &\in \mathbb{R}^{R_3\times N\times R_4} \\ \boldsymbol{\mathcal{G}^{(5)}} &\in \mathbb{R}^{R_4\times 1\times 1}. \end{aligned} \tag{5.6}$$

Similarly, The TT format of the batch of filters $\boldsymbol{\mathcal{F}}$ is given by

$$\boldsymbol{\mathcal{F}} \cong \boldsymbol{\mathcal{H}^{(1)}} \times^1 \boldsymbol{\mathcal{H}^{(2)}} \times^1 \boldsymbol{\mathcal{H}^{(3)}} \times^1 \boldsymbol{\mathcal{H}^{(4)}} \times^1 \boldsymbol{\mathcal{H}^{(5)}}, \tag{5.7}$$

where

$$\begin{aligned} \boldsymbol{\mathcal{H}^{(1)}} &\in \mathbb{R}^{1\times J_1\times Q_1} \\ \boldsymbol{\mathcal{H}^{(2)}} &\in \mathbb{R}^{Q_1\times J_2\times Q_2} \\ \boldsymbol{\mathcal{H}^{(3)}} &\in \mathbb{R}^{Q_2\times C\times Q_3} \\ \boldsymbol{\mathcal{H}^{(4)}} &\in \mathbb{R}^{Q_3\times 1\times Q_4} \\ \boldsymbol{\mathcal{H}^{(5)}} &\in \mathbb{R}^{Q_4\times M\times 1}. \end{aligned} \tag{5.8}$$

To compute the fully overlapping correlation in a normal convolutional layer $\boldsymbol{\mathcal{P}} = \boldsymbol{\mathcal{I}} \bigotimes \boldsymbol{\mathcal{F}}$, we can just simply implement the partial mode-2 correlation between each TT cores as

$$\begin{aligned}
\boldsymbol{\mathcal{U}}^{(1)} &= \boldsymbol{\mathcal{G}}^{(1)} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}}^{(1)} \in \mathbb{R}^{(1)\times(I_1-J_1+1)\times(R_1Q_1)} \\
\boldsymbol{\mathcal{U}}^{(2)} &= \boldsymbol{\mathcal{G}}^{(2)} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}}^{(2)} \in \mathbb{R}^{(R_1Q_1)\times(I_2-J_2+1)\times(R_2Q_2)} \\
\boldsymbol{\mathcal{U}}^{(3)} &= \boldsymbol{\mathcal{G}}^{(3)} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}}^{(3)} \in \mathbb{R}^{(R_2Q_2)\times(1)\times(R_3Q_3)} \\
\boldsymbol{\mathcal{U}}^{(4)} &= \boldsymbol{\mathcal{G}}^{(4)} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}}^{(4)} \in \mathbb{R}^{(R_3Q_3)\times(N)\times(R_4Q_4)} \\
\boldsymbol{\mathcal{U}}^{(5)} &= \boldsymbol{\mathcal{G}}^{(5)} \boxed{\otimes}_2 \boldsymbol{\mathcal{H}}^{(5)} \in \mathbb{R}^{(R_4Q_4)\times(M)\times(1)}.
\end{aligned} \tag{5.9}$$

The convolutional layer's result $\boldsymbol{\mathcal{P}}$ is recomstructed using these TT cores from above as

$$\boldsymbol{\mathcal{P}} \cong \boldsymbol{\mathcal{U}}^{(1)} \times^1 \boldsymbol{\mathcal{U}}^{(2)} \times^1 \boldsymbol{\mathcal{U}}^{(3)} \times^1 \boldsymbol{\mathcal{U}}^{(4)} \times^1 \boldsymbol{\mathcal{U}}^{(5)}, \tag{5.10}$$

where $\boldsymbol{\mathcal{P}} \in \mathbb{R}^{((I_1-J_1+1))\times(I_2-J_2+1)\times 1\times N\times M}$ is the result or the activation map of the convolutional layer with N images, and $\boldsymbol{\mathcal{P}}(:,:,:,n,:)$ is the result of the convolutional layer with n^{th} image.

5.1.3 Fully Connected Layer in Tensor Train Format

To convert a conventional CNN into a BTN and train with our Bayesian training algorithm, we need not only convert the convolutional layer into TT format but also the fully connected layer. However, our work was mainly focused on the formulation of the convolutional layer in the TT format and verifying the training of the BTN with our new TT convolutional layer formulation. Therefore, the conversion of TT format for fully connected layer was adopted from Noviko [6]. His idea is based on the vector-matrix multiplication in the TT format, since the fully connected layer is just a vector-matrix multiplication. However, the decomposition on a big vector or matrix by using the TT decomposition is the same as the low-rank approximation as we discussed in Section 3.4.7, so he used the quantized tensor train format to convert the big vector and matrix into TT format and then performs the vector-matrix multiplication on these tensor cores.

Overall, the benefit to representing the CNN in this manner is to reduce the storage consumption and fast forward computation, since these high order tensors are represented by smaller tensor cores and the tensor operation is done on these tensor cores. However, the conversion between the original tensor and the decomposed tensor will increase the complexity of the neural network, since we need to switch between the original representation and the decomposed representation. Unless we develop new neural network architecture that is designed specifically for the TensorNet, we must suffer the conversion issue.

5.1.4 Evaluating the Recursive Bayesian Solution

The conversion issue leads to a problem when training this TensorNet with the standard backpropagation algorithm, where every time we want to update the layer's weights with the gradient, we need to compute the gradient and run into the tensor decomposition conversion issue. Fortunately, the TensorNet has the advantage of fast-forward computation. We could train the TensorNet using Bayesian inference as we referred to as BTN, since it only requires the forward computation result to test the hypothesis. By training the BTN we could ignore the gradient computation on these trainable parameters and avoid the conversion issue.

To train the BTN, let us consider approximating of the posterior function in Eq 4.19 by the following MC approximation

$$\hat{Pr}(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{1}{N}\sum_{i=1}^{N}\delta(\boldsymbol{\theta_{0:K}} - \boldsymbol{\theta_{0:K}^{(i)}}), \tag{5.11}$$

where $\boldsymbol{\theta_{0:K}^{(i)}}$ are samples from posterior distribution function $Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$. We can approximate any expectation of the form

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \int f_k(\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})d\boldsymbol{\theta_{0:K}}, \tag{5.12}$$

by MC simulation as

$$E[f_k(\boldsymbol{\theta_{0:K}})] \approx \frac{1}{N}\sum_{i=1}^{N} f_k(\boldsymbol{\theta_{0:K}^{(i)}}), \tag{5.13}$$

where $\boldsymbol{\theta_{0:K}^{(i)}}$ is sampled from the posterior density function $Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$.

However, it is very common that we are not able to sample directly from the posterior function $Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$. But we can sample from a proposal distribution $q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$ over the target distribution $Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$ as we discussed in Section 4.2.1:

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \int f_k(\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})\frac{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}d\boldsymbol{\theta_{0:K}}. \tag{5.14}$$

Applying the Bayes' theorem

$$Pr(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \frac{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}, \tag{5.15}$$

and we can rewrite the above equation as

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \int f_k(\boldsymbol{\theta_{0:K}}) \frac{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} \frac{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} d\boldsymbol{\theta_{0:K}}. \tag{5.16}$$

Rearranging the above equation gives

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \int f_k(\boldsymbol{\theta_{0:K}}) \frac{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} \frac{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} d\boldsymbol{\theta_{0:K}}. \tag{5.17}$$

If we now let the unnormalized importance ratio be

$$w_k(\boldsymbol{\theta_{0:K}}) = \frac{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}})Pr(\boldsymbol{\theta_{0:K}})}{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}, \tag{5.18}$$

insert this equation back into Eq 5.17, we get

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \int f_k(\boldsymbol{\theta_{0:K}}) w_k(\boldsymbol{\theta_{0:K}}) \frac{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} d\boldsymbol{\theta_{0:K}}. \tag{5.19}$$

We are able to remove the normalized constant $Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$ to simplify our equation by doing the following steps. Fisrt, we write

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \frac{1}{Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} \int f_k(\boldsymbol{\theta_{0:K}}) w_k(\boldsymbol{\theta_{0:K}}) q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) d\boldsymbol{\theta_{0:K}}. \tag{5.20}$$

Then, recognizing that

$$Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \int Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}}) Pr(\boldsymbol{\theta_{0:K}}) d\boldsymbol{\theta_{0:K}}, \tag{5.21}$$

we can write

$$Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \int Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}}) Pr(\boldsymbol{\theta_{0:K}}) \frac{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}{q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})} d\boldsymbol{\theta_{0:K}}. \tag{5.22}$$

By simplify the equation with the unnormalized importance ratio on Eq 5.18

$$Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = \int w_k(\boldsymbol{\theta_{0:K}}) q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) d\boldsymbol{\theta_{0:K}}. \tag{5.23}$$

Now, we can rewrte Eq 5.20 as

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \frac{\int f_k(\boldsymbol{\theta_{0:K}}) w_k(\boldsymbol{\theta_{0:K}}) q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) d\boldsymbol{\theta_{0:K}}}{\int w_k(\boldsymbol{\theta_{0:K}}) q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) d\boldsymbol{\theta_{0:K}}}. \tag{5.24}$$

Finally, recognizing the numerator and denominator of Eq 5.20 have the same form, we can write

$$E[f_k(\boldsymbol{\theta_{0:K}})] = \frac{E_q[w_k(\boldsymbol{\theta_{0:K}}) f_k(\boldsymbol{\theta_{0:K}})]}{E_q[w_k(\boldsymbol{\theta_{0:K}})]}. \tag{5.25}$$

Both the numerator and denominator can now be approximated by MC approximation by sampling from the proposal distribution $q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})$. This is written as

$$E[f_k(\boldsymbol{\theta_{0:K}})] \approx \frac{\frac{1}{N}\sum_{i=1}^{N} w_k(\boldsymbol{\theta}_{\boldsymbol{0:K}}^{(i)}) f_k(\boldsymbol{\theta}_{\boldsymbol{0:K}}^{(i)})}{\frac{1}{N}\sum_{i=1}^{N} w_k(\boldsymbol{\theta}_{\boldsymbol{0:K}}^{(i)})}. \tag{5.26}$$

Letting $\tilde{w}^{(i)}$ be the normalized importance ratio given by

$$\tilde{w_k}^{(i)} = \frac{{w_k}^{(i)}}{\sum_{i=1}^{N} {w_k}^{(i)}}, \tag{5.27}$$

we can further simplify Eq 5.26 with the normalized importance as

$$E[f_k(\boldsymbol{\theta_{0:K}})] \approx \sum_{i=1}^{N} \tilde{w_k}(\boldsymbol{\theta}_{\boldsymbol{0:K}}^{(i)}) f_k(\boldsymbol{\theta}_{\boldsymbol{0:K}}^{(i)}). \tag{5.28}$$

To approximate our solution sequentially without modifying our samples $\boldsymbol{\theta_{0:k-1}}$, we need to select a proposal distribution like the following

$$q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = q(\boldsymbol{\theta_0}) \prod_{j=1}^{k} q(\boldsymbol{\theta_j}|\boldsymbol{\theta_{0:j-1}}, \boldsymbol{y_{1:j}}, \boldsymbol{x_{1:j}}), \tag{5.29}$$

or

$$q(\boldsymbol{\theta_{0:K}}|\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}) = q(\boldsymbol{\theta_{0:K-1}}|\boldsymbol{y_{1:K-1}}, \boldsymbol{x_{1:K-1}}) q(\boldsymbol{\theta_k}|\boldsymbol{\theta_{0:K-1}}, \boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}). \tag{5.30}$$

Since the BTN parameters evolved in the training process is a Markov process, where the current BNN's parameters only depend on the previous parameters, we can write

$$Pr(\boldsymbol{\theta_{0:K}}) = Pr(\boldsymbol{\theta_0}) \prod_{j=1}^{k} Pr(\boldsymbol{\theta_j}|\boldsymbol{\theta_{j-1}}). \tag{5.31}$$

The network's output is conditionally independent of the network parameter, therefore we can rewrite this equation as

$$Pr(\boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}}|\boldsymbol{\theta_{0:K}}) = \prod_{j=1}^{k} Pr(\boldsymbol{y_j}, \boldsymbol{x_j}|\boldsymbol{\theta_j}). \tag{5.32}$$

By using Eqs. 5.30, 5.31 and 5.32, we can sequentially compute the importance ratio in the Eq 5.18 by

$$w_k = w_{k-1} \frac{Pr(\boldsymbol{y_k}, \boldsymbol{x_k}|\boldsymbol{\theta_k}) Pr(\boldsymbol{\theta_k}|\boldsymbol{\theta_{k-1}})}{q(\boldsymbol{\theta_k}|\boldsymbol{\theta_{0:K-1}}, \boldsymbol{y_{1:K}}, \boldsymbol{x_{1:K}})}. \tag{5.33}$$

We are able to update our importance ratio based on Eq 5.33, and the estimation of the solution can be computed with the normalized importance ratio as

$$E[f_k(\boldsymbol{\theta_{0:K}})] \approx \sum_{i=1}^{N} \tilde{q_k}^{(i)} f_k(\boldsymbol{\theta}_{\mathbf{0:K}}^{(i)}). \tag{5.34}$$

At this stage, the immediate question is which proposal function we should use in our SMC design. For the convenient implementation of the training algorithm, we have decided in the work to use this most popular choice of the proposal distribution, which is the transition function from the state conditional density of the Bayesian training process:

$$q(\boldsymbol{\theta_k}|\boldsymbol{\theta_{0:k-1}}, \boldsymbol{y_{1:k}}, \boldsymbol{x_{1:k}}) = Pr(\boldsymbol{\theta_k}|\boldsymbol{\theta_{k-1}}) \tag{5.35}$$

As a result, our importance ratio is based on the choice of the proposal distribution as we discussed in Section 4.2.1:

$$w_k = w_{k-1} \frac{Pr(\boldsymbol{y_k}, \boldsymbol{x_k}|\boldsymbol{\theta_k}) Pr(\boldsymbol{\theta_k}|\boldsymbol{\theta_{k-1}})}{Pr(\boldsymbol{\theta_k}|\boldsymbol{\theta_{k-1}})}, \tag{5.36}$$

$$w_k = w_{k-1} Pr(\boldsymbol{y_k}, \boldsymbol{x_k}|\boldsymbol{\theta_k}) \tag{5.37}$$

where the likelihood function $Pr(\boldsymbol{y_k}, \boldsymbol{x_k}|\boldsymbol{\theta_k})$ is easy to compute and will be used to update the importance ratio sequentially.

5.2 Bayesian TensorNet Training Workflow

In this thesis, we trained our BTN for a classification problem with a dataset of single-plant images. The BTN was constructed with a single TT-convolutional layer, one flatten layer, one TT-full connected layer, and one fully connected layer. The training of BTN is basically approximated by the recursive Bayesian solution by using the SMC method.

The training workflow of our experiment is shown in Fig 5.3. We start by creating the training data and selecting the architecture of the conventional neural network, then we decompose the convolutional and fully connected layers into TT format by using the TT decomposition technique. To approximate the recursive Bayesian solution via SMC method, samples used in the SMC method are predicted and updated via the transition and measurement equation. These prediction and update steps are sequential in evaluating the recursive solution. These samples construct the representation of the posterior function we are approximating. When we have the information of the posterior function, we can compute our estimator, such as mean, median maximum a posteriori (MAP), etc. Some important components of the training workflow will be discussed in the following subsections.

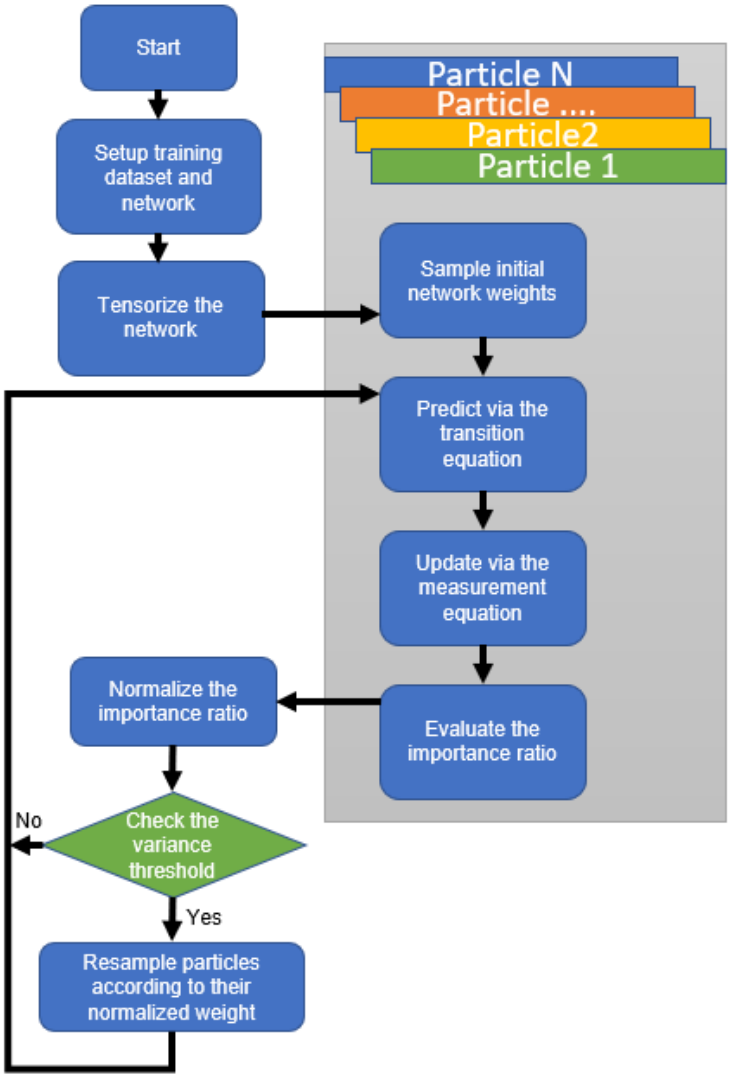

Figure 5.3: The diagram shows the main components in our BTN training. The left top part is the pre-processing where we set up the training dataset and neural network in the TT format. The right-hand side in the gray box is the main body of the SMC method to simulate and optimize our BTN's trainable parameters. The left bottom part is an extension of our SMC method, where we use the resampling method to improve the simulation or training.

5.2.1 Preparing the Training Dataset

The BTN input is RGB single-plant images, where a single plant is in a blue poy surrounded by a blue background. Therefore, we can focus on classifying the plant type and ignoring the background noise. Plant images with different classes are saved in different folders (canola, wheat, and soy). A python script was written to construct a CSV file with the image file name and class label from each

folder. The CSV file is then used in our TensorFlow script for loading those training images and labels. Examples of images are shown in Fig 5.4.

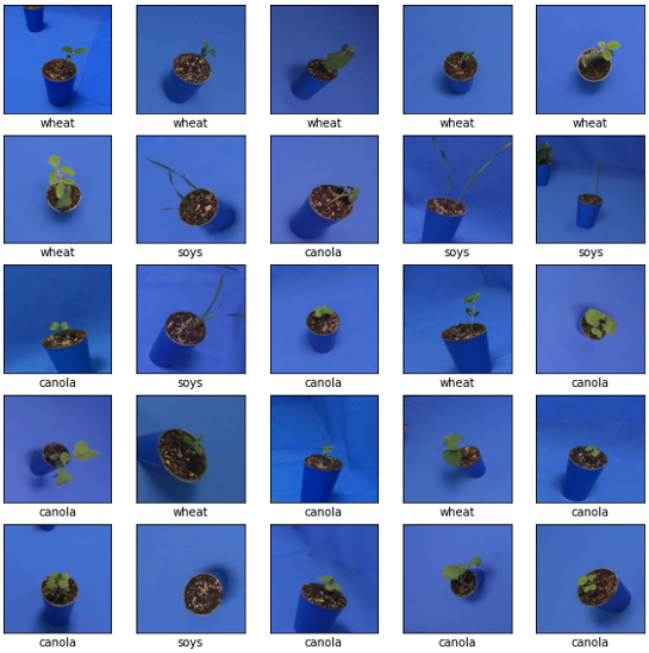

Figure 5.4: A single plant dataset from TerraByte with 3 classes: canola, wheat and soy bean. A total of 10000 images was used in our training with 3522 canola images, 3711 wheat images, and 2731 soy images. These image are resized to 64 by 64 for better illustration and training.

Before we start the training, we need to first implement pre-processing of the raw images that were first resized to 64×64 to have them all with the same resolution. They were normalized with the ImageNet normalization mean and variance. Finally the training images were rearranged into batches representing by a 4^{th} order tensor such as $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{64\times64\times3\times N}$.

Since our first layer is a TT-convolutional layer in TT-format, we need to send the batch of images in TT-format as well. The 4^{th} order tensor will be rearranged as a 5^{th} order tensor $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{64\times64\times3\times N\times1}$ by adding an empty dimension. This 5^{th} order tensor was decomposed into a TT format representing by 5 tensor cores and TT ranks $(R_0, R_1, R_2, R_3, R_4, R_5)$ as

$$\begin{aligned}
\boldsymbol{\mathcal{G}}^{(1)} &\in \mathbb{R}^{1\times64\times R_1} \\
\boldsymbol{\mathcal{G}}^{(2)} &\in \mathbb{R}^{R_1\times64\times R_2} \\
\boldsymbol{\mathcal{G}}^{(3)} &\in \mathbb{R}^{R_2\times3\times R_3} \\
\boldsymbol{\mathcal{G}}^{(4)} &\in \mathbb{R}^{R_3\times N\times R_4} \\
\boldsymbol{\mathcal{G}}^{(5)} &\in \mathbb{R}^{R_4\times1\times1}.
\end{aligned} \tag{5.38}$$

The five tensor cores become the input we send to our BTN for training. One can represent the input images by using fewer elements than the actual size of the input images, therefore the bandwidth of the neural network is smaller than the original one.

5.2.2 Neural Network in Tensor Train Format

The CNN we tested was constructed by one convolutional layer with two fully connected layers. Usually, the convolutional layer is the bottleneck for the forward and backward computation, where doing the correlation between two high order tensors is computationally expensive. On the other hand, the fully connected layer is the bottleneck from the storage point of view, since it stores a huge matrix for the linear operation in the neural network. Simplification of the convolutional layer computation and fully connected layer storage will be needed to enable the use of the neural network in a cheaper or portable device.

In our experiment, the convolutional layer is represented in TT format by using the idea of Section 5.1.2. For these two fully connected layers, we will only represent the first fully connected layer into TT-format by using the idea of Section 5.1.3. the second fully connected layer will stay in the regular format. We chose to do this because the second fully connected layer is the last layer of the BTN, and the size of this layer is not big. A diagram of the BTN is shown in Fig 5.5.

The convolutional layer has thirty-two 3×3 filters and is a 4^{th} order tensor. However, we want to rewrite it as a 5^{th} order tensor given by $\boldsymbol{\mathcal{F}} \in \mathbb{R}^{3\times3\times3\times1\times32}$ by adding an empty dimension, alowing it to be represented in TT format with five tensor cores as the formulation in Section 5.1.2. The five tensor cores with TT ranks $(Q_0, Q_1, Q_2, Q_3, Q_4, Q_5)$ is given by

$$\begin{aligned}
\boldsymbol{\mathcal{H}}^{(\mathbf{1})} &\in \mathbb{R}^{1\times3\times Q_1} \\
\boldsymbol{\mathcal{H}}^{(\mathbf{2})} &\in \mathbb{R}^{Q_1\times3\times Q_2} \\
\boldsymbol{\mathcal{H}}^{(\mathbf{3})} &\in \mathbb{R}^{Q_2\times3\times Q_3} \\
\boldsymbol{\mathcal{H}}^{(\mathbf{4})} &\in \mathbb{R}^{Q_3\times1\times Q_4} \\
\boldsymbol{\mathcal{H}}^{(\mathbf{5})} &\in \mathbb{R}^{Q_4\times32\times1}.
\end{aligned} \tag{5.39}$$

The method to represent the fully connected layer into a TT-format is exactly the same as the method Alexander proposed [6], where they tensorize the matrix into a high order tensor and represented the high order tensor into QTT format. For more detailed implementation, refer to Novikov's paper [6].

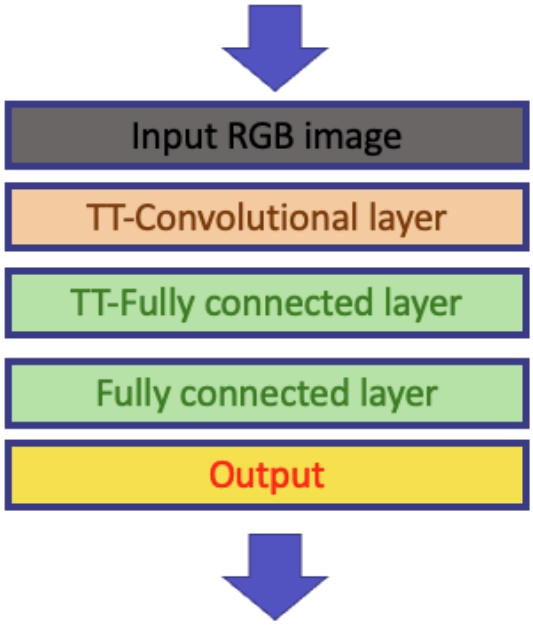

Figure 5.5: A simple CNN with RGB image as input. The CNN is made by one convolutional layer in TT format (refered to as TT-Convolutional layer), one fully connected layer in TT format (refered to as TT-Fully connected layer), and a regular fully connected layer. A softmax function is used on the output of the fully connected layer to compute the output.

5.2.3 Prediction Step

The prediction step is an important part of our training process, where the previous BTN parameters are predicted based on Eq 5.35. We randomly predict the next BTN parameters by using Gaussian noise, which can be mathematically represented as

$$\boldsymbol{\theta}_{\boldsymbol{k}}^{(i)} = \boldsymbol{\theta}_{\boldsymbol{k-1}}^{(i)} + \boldsymbol{n}_{\boldsymbol{k-1}}^{(i)}, \tag{5.40}$$

where $n_{k-1}^{(i)}$ is sample from a Gaussian distribution $N(0, var_n)$. The predicted parameter of the BTN is basically searching in the parameter space for an optimized parameter set.

5.2.4 Update Step

After the prediction step, we have a BTN with a new set of predicted parameters. However, we do not know how good the random prediction was. We need a way to guide us toward a better parameter set. Here, we employ an update step which is based on the Eq 5.37. The equation was used to update our importance ratio. The updated importance ratio is the information indicating how good our prediction

is. If the prediction is good, the importance ratio of that prediction sample will be greater than those of bad prediction.

5.2.5 Variance Threshold

After the prediction and update steps on the BTN's parameters, the importance ratio will be normalized. We want to check the variance threshold to ensure we need to implement the resampling method to prevent particle variance degeneracy of the SMC method, as discussed in Section 4.3.2. To do this, we used a variance threshold to check if the implementation of resampling is required. The variance threshold used here is given by Doucet [31]:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N} (\tilde{w_k}^{(i)})^2}, \tag{5.41}$$

where N_{eff} is the variance threshold, and $\tilde{w_k}^{(i)}$ is the normalized important ratio for each particle at $t = k$. If the variance of the normalized particle is great than the N_{eff}, we need to perform the resampling on all particles. Otherwise, we skip the resampling step and perform the next training step.

5.2.6 Resampling Implementation

Every time the normalized importance ratio's variance is above the variance threshold, we will perform the resampling algorithm. In our training, the normalized ratio of the particle is used as the selection probability in the resampling algorithm, where the normalized importance ratio is given by

$$W_i' = \frac{W_i}{\sum_{i=1}^{N} W_i}, \tag{5.42}$$

and the variance of the normalized importance ratio is given by

$$\sigma^2 = \frac{\sum_{i=1}^{N} (W_i - \overline{W})^2}{N}, \tag{5.43}$$

where the $\overline{W}$ is the mean of importance ratio. The calculated variance is compared with a variance threshold N_{eff} to decide if resampling should be implemented or not.

5.3 TensorFlow and T3F Library

All the implementation was done on the TensorFlow platform. TensorFlow was created by Google Brain and is an open-source library for machine learning [32]. TensorFlow provides easy access for building

machine learning applications. We chose TensorFlow as our programming language because TensorFlow allows the speed-up of the construction of machine learning models, so we do not have to build them from scratch. The machine learning model can be generated in just a few lines instead of hours of work.

TensorFlow is an excellent programming language for doing machine learning research, However, TensorFlow was developed mainly for backpropagation training algorithm, which is a totally different training algorithm than our Bayesian training algorithm in this thesis. As a result, we were not able to use the full benefits from TensorFlow, such as GPUs support and fast computation algorithm. The TensorFlow code in our project was mainly used for the basic model construction and forward computation pipeline with our tensor operation.

T3F is a library built on top of TensorFlow for researchers working with the Tensor Train decomposition created by Novikov and Izmailov [33]. This library gives researchers easy access to Tensor Train decomposition with the TensorFlow API. It also provides basic tensor operations for tensor computations such as tensor addition, multiplication, and tensor product. The T3F library was used in our project to represent the convolutional layer and fully connected layer into the TT format.

Chapter 6

Training Performance Comparision and Analysis

We implemented our BTN training on an Intel i9-10980XE CPU with 3.0GHz and 256GB RAM. We trained our BTN with the TerraByte plant images for a classification problem. We tested our BTN with various prediction noises for searching better training performance. A different estimator was used to verify the BTN's training and inference. In the following results, we will show two BTN trainings with different Gaussian noise and compare their performance on the BTN inference.

However, it is not only the prediction noise that can affect our BTN performance, since the BTN's trainable parameters are probabilistic instead of deterministic. Therefore, choosing an estimator is also a key to having a good model. In our experiment, we have found the mean and the maximum a posteriori (MAP) estimator has the best performance compared with others. In the following result, we will use the mean and the MAP as our estimator to test our BTN performance.

6.1 Training Preformance with Small Prediction Noise

In this experiment, we set up our prediction noise with Gaussian noise and the standard deviation as 0.001. The training was done with the TerraByte single plant dataset where 10000 images were used. 5000 images for the training step and the other 5000 for the validation step. We sequentially approximated the posterior function by using the SMC method. The mean estimator of the posterior function was computed every 100 training steps, and the estimator was used to compute the training loss against the training step plot. The training loss against the training step plot is shown in Fig. 6.1. The plot shows we were able to train our BTN, and we reduced the loss value from 1.10 to 0.81 in 10000

training steps. However, due to the lack of GPU acceleration support on our new algorithm, the total training time took approximately 73 hours on a single Intel i9-10980XE CPU. We expect this training speed will improve with better programming coding and hardware support.

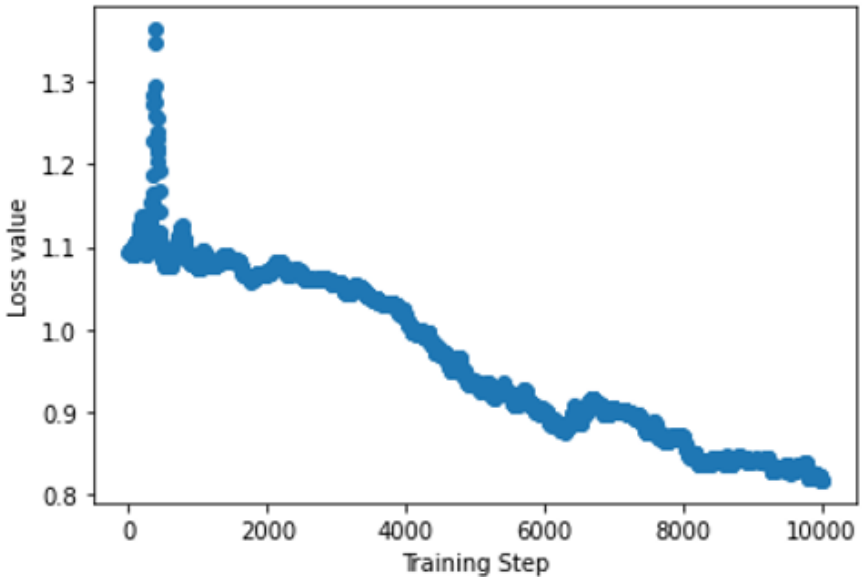

Figure 6.1: In this plot, the trianing loss function was the Tensorflow built-in loss function SparseCategorical-Crossentropy. The loss value was computed every 100 trianing step when the mean estimator is avaliable. The BTN with predition noise N(0,std = 0.001) was able to reduce the loss value from 1.10 to 0.81 in 10000 steps.

After the BTN finishes the 10000 training steps, it will stop the training and compute the final mean estimator and MAP estimator of the posterior. The performance of the BTN was tested with the mean estimator and the MAP estimator. A confusion matrix plot was used to present the BTN prediction accuracy on the validation images. Fig. 6.2 and 6.3 show the confusion matrices for the mean and MAP estimator respectively.

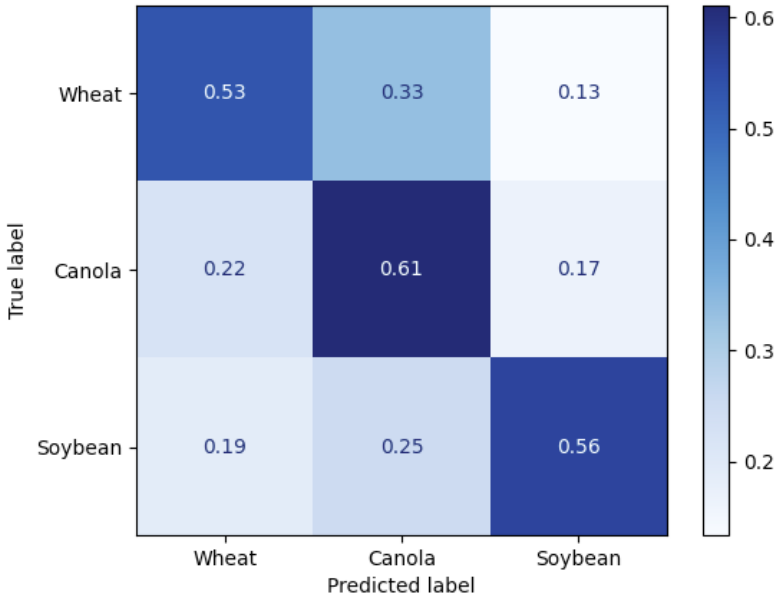

Figure 6.2: The plot shows the trained BTN with the mean estimator. The folowing accuracies were achieved: 0.53 on the wheat image prediction; 0.61 on the canola image prediction and 0.56 on the soybean image prediction. As shown in the legend, higher prediction levels are indicated by a darker colour.

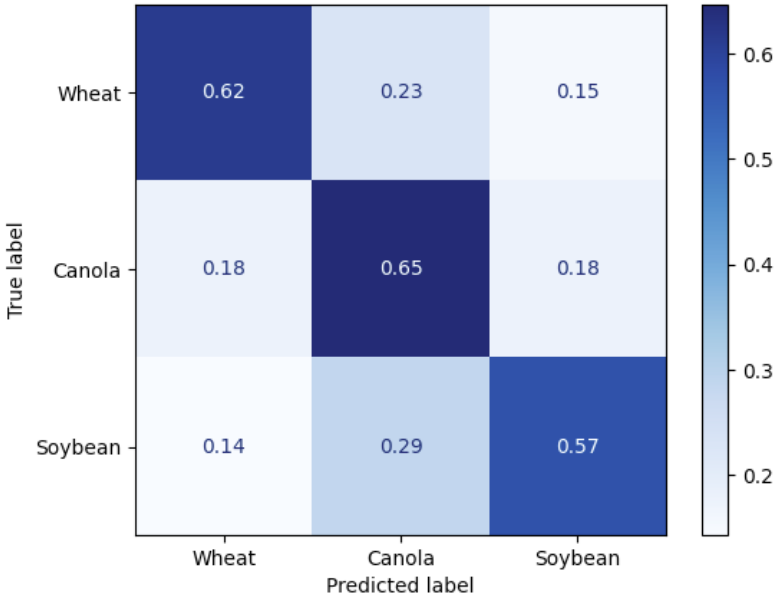

Figure 6.3: The plot shows the trained BTN with the MAP estimator. The folowing accuracies were achieved: 0.62 on the wheat image prediction; 0.65 on the canola image prediction and 0.57 on the soybean image prediction. As shown in the legend, higher prediction levels are indicated by a darker colour.

The BTN was successfully trained with our Bayesian training algorithm without computing any gradient.

6.2 Training Preformance with Large Prediction Noise

The BTN trained with prediction noise N(0, std = 0.001) shows a good prediction result after the 10000 training steps. If we choose a proper prediction noise, we could reduce the loss value further and improve our BTN's performance. Here we present the training result with the prediction noise N(0, std = 0.005) with the same training setup and training dataset as in Section 6.1. The training loss against the training step plot is shown in Fig 6.4. We were able to reduce the loss value from 1.10 to 0.68 in 10000 training steps. The total training time took approximately 75 hours on a single Intel i9-10980XE CPU.

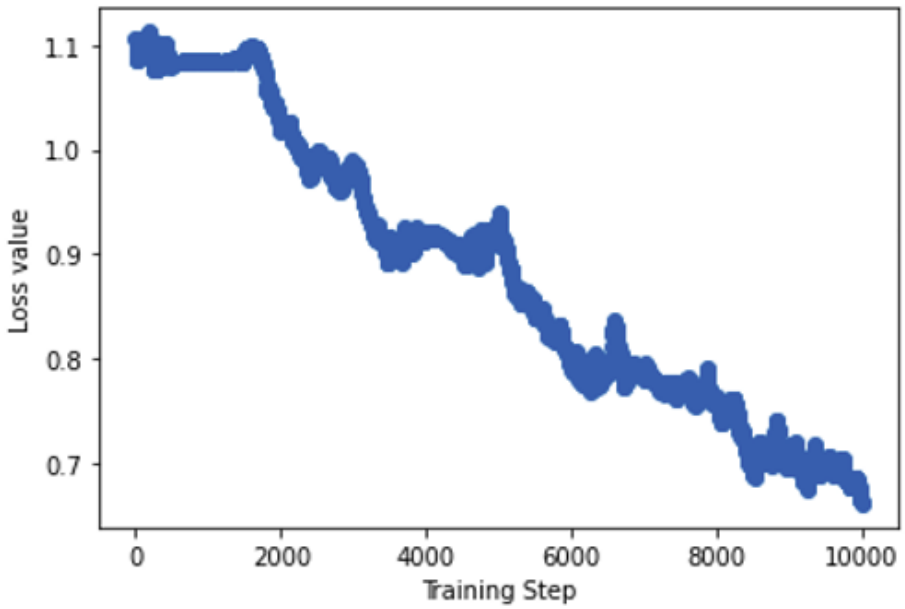

Figure 6.4: In this plot, the trianing loss function was the Tensorflow build-in loss function SparseCategorical-Crossentropy. The loss value was computed every 100 trianing step when the mean estimator is avaliable. The BTN with predition noise N(0,std = 0.005) was able to reduce the loss value from 1.10 to 0.68 in 10000 steps.

By doing the same process, we trained the BTN with prediction noise N(0, std = 0.005) with 10000 steps. The mean and MAP estimator were computed after the training. The performance of the BTN was tested with the mean estimator and the MAP estimator. A confusion matrix plot was used to present the BTN prediction accuracy on the validation images. Fig. 6.5 and 6.6 is the confusion matrix for the mean and MAP estimator respectively.

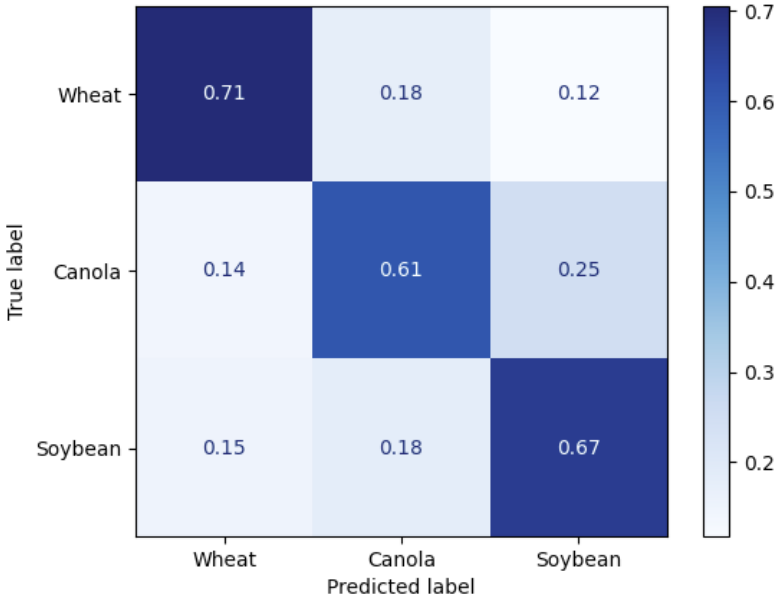

Figure 6.5: The plot shows the trained BTN with the mean estimator. The folowing accuracies were achieved: 0.70 on the wheat image prediction; 0.61 on the canola image prediction and 0.67 on the soybean image prediction. As shown in the legend, higher prediction levels are indicated by a darker colour.

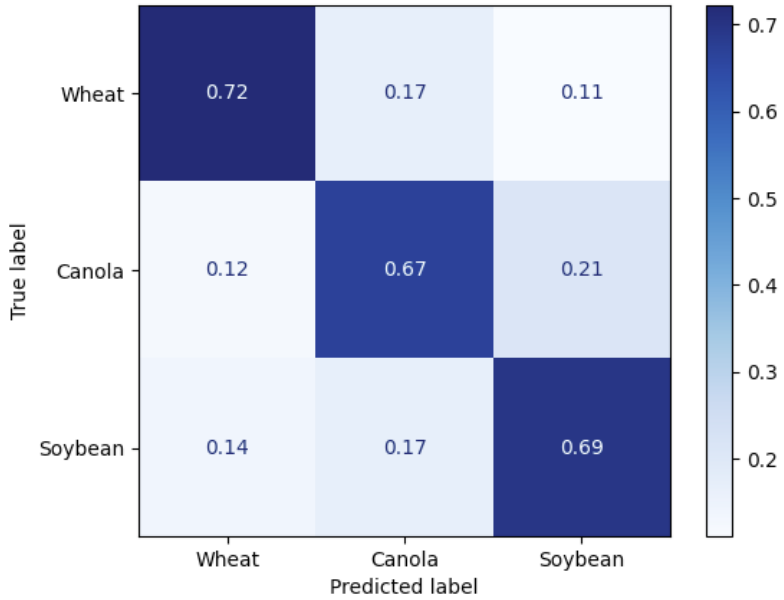

Figure 6.6: The plot shows the trained BTN with the MAP estimator. The folowing accuracies were achieved: 0.72 on the wheat image prediction; 0.67 on the canola image prediction and 0.69 on the soybean image prediction. As shown in the legend, higher prediction levels are indicated by a darker colour.

We increased the prediction noise to further reduce the loss value and improved our BTN performance in the same 10000 training steps. However, there is a limit to using a bigger prediction noise. After a lot of training experiments, we found the BTN with the single plant dataset is only able to train with Gaussian noise N(0, std<0.01). If the prediction noise has a standard deviation greater than 0.01, the BTN will not be able to converge. On the other hand, if we use a very small prediction noise, it will slow down training convergence. Therefore, the role of the prediction noise in our Bayesian training algorithm is like the learning rate in the backpropagation algorithm. It is a hyperparameter in the training process and one needs to be careful with the choice of noise level.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we developed a novel tensor-train formulation of a convolutional neural network and trained it with a Bayesian training algorithm for a plant classification problem. We represented the fully connected layer with the same idea given by Novikov [6]. We also used our novel tensor-train representation for the convolutional layer. In our experiment with the TerraByte plant dataset, we successfully trained the BTN for a classification problem with an average 67% accuracy on the validation dataset. We also found the MAP estimator has the best performance in this plant dataset classification experiment. We discussed the property of the prediction noise in our training algorithm and found our prediction noise cannot exceed N(0,std>0.01).

By using the tensorization method on the neural network, we could reduce the network's size and improve the inference time. The digital agriculture development usually needs portable and power-saving hardware to implement the neural network for various applications in the outdoor field. This work could be valuable to digital agriculture to integrate the state-of-art technology, since our neural network required a smaller memory size and a simpler forward computation on the model inference.

7.2 Future Work and Discussion

Still more research that needs to be done on the BTN to make the BTN more efficient and practical. Since there is no GPU acceleration support on the BTN, further optimization and GPU parallelization are required to speed up the training process with our new training algorithm. A fair comparison between our training algorithm and the backpropagation training algorithm is also needed. To compare

both training algorithms fairly, one needs to formulate the backpropagation training algorithm on the TT-layers and train the TensorNet on the same hardware support. The most important thing is that we need more development of the new neural network architecture specifically for the BTN, since most of the neural network architecture is built based on the regular tensor representation and backpropagation training algorithm.

The approximation by using other numerical algorithm could also help us to solve the recursive Bayesian solution. In our thesis we used the SMC method. Exploring other Monte Carlo methods to estimate the posterior function could include Reversible-jump Markov chain Monte Carlo and Resample-move Sequential Monte Carlo method.

Finally, replacing the TT decomposition by using other tensor decomposition method, such as more general hierachical Tucker decomposition to further reduce the total number of CNN parameters and improve the inference time.

Bibliography

[1] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *MICCAI*, 2015.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84 – 90, 2012.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[5] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.

[6] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *NIPS*, 2015.

[7] I. Oseledets, "Tensor-train decomposition," *SIAM J. Sci. Comput.*, vol. 33, pp. 2295–2317, 2011.

[8] T. Garipov, D. Podoprikhin, A. Novikov, and D. P. Vetrov, "Ultimate tensorization: compressing convolutional and fc layers alike," *ArXiv*, vol. abs/1611.03214, 2016.

[9] A. Doucet, N. de Freitas, and N. J. Gordon, "Sequential monte carlo methods in practice," in *Statistics for Engineering and Information Science*, 2001.

[10] H. collaboration V. Andreev, M. Arratia, A. Baghdasaryan, A. Baty, and K. Begzsuren, "Measurement of lepton-jet correlation in deep-inelastic scattering with the h1 detector using machine learning for unfolding," 2021.

[11] M. A. Beck, C.-Y. Liu, C. P. Bidinosti, C. J. Henry, C. M. Godee, and M. Ajmani, "An embedded system for the automated generation of labeled plant images to enable machine learning applications in agriculture," *PLoS ONE*, vol. 15, 2020.

[12] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biology*, vol. 52, pp. 99–115, 1990.

[13] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65 6, pp. 386–408, 1958.

[14] I. J. Goodfellow, Y. Bengio, and A. C. Courville, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.

[15] F. L. Hitchcock., "The expression of a tensor or a polyadic as a sum of products," 1927.

[16] A. Cichocki, N. Lee, I. Oseledets, A. Phan, Q. Zhao, and D. P. Mandic, "Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges part 1," *ArXiv*, vol. abs/1609.00893, 2016.

[17] R. B. Cattell, ""parallel proportional profiles" and other principles for determining the choice of factors by rotation," *Psychometrika*, vol. 9, pp. 267–283, 1944.

[18] R. B. Cattell, "The three basic factor-analytic research designs-their interrelations and derivatives.," *Psychological bulletin*, vol. 49 5, pp. 499–520, 1952.

[19] C. J. Appellof and E. R. Davidson, "Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents," *Analytical Chemistry*, vol. 53, pp. 2053–2056, 1981.

[20] L. D. Lathauwer and A. de Baynast, "Blind deconvolution of ds-cdma signals by means of decomposition in rank-$(1,l,l)$ terms," *IEEE Transactions on Signal Processing*, vol. 56, pp. 1562–1571, 2008.

[21] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, pp. 1253–1278, 2000.

[22] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear analysis of image ensembles: Tensorfaces," in *ECCV*, 2002.

[23] M. A. O. Vasilescu and D. Terzopoulos, "Multilinear image analysis for facial recognition," *Object recognition supported by user interaction for service robots*, vol. 2, pp. 511–514 vol.2, 2002.

[24] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, pp. 455–500, 2009.

[25] P. Gelß, "The tensor-train format and its applications," 2017.

[26] A. Cichocki, "Tensor networks for big data analytics and large-scale optimization problems," *ArXiv*, vol. abs/1407.3124, 2014.

[27] L. V. Jospin, W. L. Buntine, F. Boussaid, H. Laga, and Bennamoun, "Hands-on bayesian neural networks - a tutorial for deep learning users," *ArXiv*, vol. abs/2007.06823, 2020.

[28] W. L. Dunn, "Exploring monte carlo methods," 2011.

[29] J. V. Candy, "Bayesian signal processing: Classical, modern and particle filtering methods," 2009.

[30] T. Li, M. Bolic, and P. M. Djurić, "Resampling methods for particle filtering," 2015.

[31] A. Doucet and A. M. Johansen, "A tutorial on particle filtering and smoothing: Fifteen years later," 2008.

[32] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "Tensorflow: A system for large-scale machine learning," *ArXiv*, vol. abs/1605.08695, 2016.

[33] A. Novikov, P. Izmailov, V. Khrulkov, M. Figurnov, and I. Oseledets, "Tensor train decomposition on tensorflow (t3f)," *J. Mach. Learn. Res.*, vol. 21, pp. 30:1–30:7, 2020.