POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Look-Up Table Based Neural Networks For Fast Inference
Auteur: Author:	Moussa Traore
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Traore, M. (2022). Look-Up Table Based Neural Networks For Fast Inference [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10547/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10547/
Directeurs de recherche: Advisors:	J. M. Pierre Langlois, & Jean Pierre David
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Look-Up Table Based Neural Networks For Fast Inference

MOUSSA TRAORE
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Aout 2022

© Moussa Traore, 2022.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Look-Up Table Based Neural Networks For Fast Inference

présenté par **Moussa TRAORE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Guy BOIS, président
Pierre LANGLOIS, membre et directeur de recherche
Jean Pierre DAVID, membre et codirecteur de recherche
Tarek OULD BACHIR, membre

DEDICATION

To my people, that inspired me
not to become someone else,
but to be more thoroughly
myself…

ACKNOWLEDGEMENTS

Leaving my job, and looking for my backpack on the first day back to school is memorable. The excitement I felt, the constant hunger for knowledge and the passion I had for discovery brought me to Professor Langlois and Professor David on a cold winter of January 2020. For the warm welcome to their laboratory, the constant guidance and support I would like to thank Professor Langlois and Professor David. Going through this entire program despite COVID-19 would not have happened without their dedication, their sustained push for improvement and their relentless pursuit of excellency. Thank you.
Working alongside brilliant people with the same dreams and aspirations as me had never felt better. I would like to thank Alexandre, Hossein, Mehdi and Su who have shared ideas and never hesitated to jump on a white board with me for a heated discussion on the next best step.
I would like to thank my mother for never letting me down.
I would like to thank and congratulate my father, Traore Mamadou, for achieving his Doctorate degree while I was working on this thesis. It showed me that no matter a person's age, his success and the respect he receives from his peers, having the humility to learn is something to thrive for. It pushed me to where I am today.
I would like to thank my sister and my brother that put up with my sleepless nights, and my constant stress. Thanks for the support and for believing in me.

RÉSUMÉ

La dernière décénie a connu de fulgurants dévelopements dans le domaine de l'intelligence artificielle, plus précisément de l'apprentissage profond. Bien qu'ayant été un sujet de recherche depuis les années 80, ce domaine ne connait son essor que depuis 2012 lorsqu'un modèle d'apprentissage profond gagne la compétition ImageNet de reconnaissance d'image. Depuis, de nombreux chercheurs se sont penchés sur la question. Plusieurs problèmes majeurs restent encore sans solution, dont celui de la performance de ces modèles et de leur consommation importante d'énergie. Dans le cadre de ce travail, nous explorons l'efficacité de ces modèles dans le contexte des systèmes embarqués qui ont un budget énergétique et une puissance de calculs relativement limités. Pour tenter d'apporter une solution à certains de ces problèmes, nous explorons des techniques de quantification binaire, d'élagage, et des réseaux maître-élève. Généralement, lors de la conception d'un réseau binaire, l'architecture du système sous-jacent n'est pas prise en compte. Cependant, l'exploration récente de l'application des réseaux binaires sur des FPGA a mené à de nouvelles architectures de réseaux pouvant être contenus complètement dans les tables de vérités d'un FPGA. Pour construire ce type de modèles, plusieurs algorithmes d'apprentissage existent. PoET-BiN, l'un de ces algorithmes, est au coeur de notre travail. Dans ce travail, nous commençons par explorer une nouvelle forme d'algorithme visant à compresser les réseaux de neurones binaires. Nous apportons ensuite des améliorations sur le modèle algorithmique de PoET-BiN, dans le but d'améliorer la précision de ses prédictions et explorons comment ce modèle peut être appliqué sur les couches de convolution d'un réseau neural convolutif. Nous démontrons des améliorations en terme de précision d'apprentissage par rapport à l'algorithme original de PoET-BiN sur MNIST. Finalement, nous étudions des améliorations matérielles, notamment en termes d'architecture processeur pouvant améliorer grandement le traitement des prédictions faites par les réseaux de neurones basés sur les tables de vérité. Nous atteignons une une accélération maximale d'un facteur de 2994× lorsque le temps de calcul de notre processeur spécialisé est comparé à celui d'un processeur standard ne possédant pas notre unité de calculs spécialisée.

ABSTRACT

The last decade has seen tremendous developments in the field of artificial intelligence, more specifically deep learning. Although deep learning had been a research subject since the 1980s, the field only took off in 2012 when a deep leaning model won the ImageNet competition. However, several key problems still exist including that of the performance of these models with respect to their energy and power consumption. In this work, we explore the effectiveness of these models in the context of embedded systems that have a constrained energy and power budget. In an attempt to provide a solution to common problems, we explore techniques of binary quantification, pruning and master-networks students. Generally, when designing a binary network, the architecture of the underlying system is not taken into account. But, lately the exploration of the application of binary networks using FPGAs has led to new network architectures that can exclusively be contained in the look-up tables of an FPGA. Several learning algorithms exist to build such models. One of which, PoET-BiN, is at the heart of our work. In this work, we begin by exploring a new form of algorithm aimed at compressing binary neural networks. Then, we bring improvements to the algorithmic model of PoET-BiN, in order to improve the accuracy of the predictions made by PoET-BiN and explore how this model can be applied on the convolutional layers of a convolutional neural network. We demonstrate improvements in terms of learning accuracy compared to the original PoET-BiN algorithm on MNIST. Finally, we study hardware improvements, in particular in terms of processor architecture that can greatly improve the processing speed of these models. We achieve a 2994× speedup when comparing our specialized processor to a standard processor, which does not have our processing unit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
BNN	Binarized Neural Network
BTNN	Binarized Tensor decomposition of Neural Networks
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DNN	Deep Neural Network
FC	Fully Connected
FPGA	Field-Programmable Gate Array
FSP	Flow of Solution Procedure
HOSVD	Higher-Order Singular Value Decomposition
KD	Knowledge Distillation
KL	Kullback–Leibler
LUT	Look-Up Table
MAC	Multiply And Accumulate
MNIST	Modified National Institute of Standards and Technologies
PoET-BiN	Power Efficient Tiny Binary Neurons
RINC	Reduced Input Neural Circuit
SIMD	Single Instruction Multiple Data
SGD	Stochastic Gradient Descent
SOTA	State-Of-The-Art
SVD	Singular Value Decomposition
TT	Tensor-Train
VLIW	Very Long Instruction Word

CHAPTER 1 INTRODUCTION

The adequacy of machine learning algorithms, neural networks (NN) in particular, to process and extract useful information from the abundance of available data in all types of natural, human and industrial activities, has been clearly demonstrated in recent years [1] [2] [3]. More specifically, with recent hardware improvements, Deep learning (DL) algorithms, where several neural network layers are successively stacked in a single neural network, have become the state-of-the-art (SOTA) solution for many data intensive tasks, outperforming many classical learning techniques and expert systems. However, they involve an immense amount of computations and memory to store the model parameters. For example, EfficientNet-L2 has 480 million parameters and GPT-3 has 175 billion parameters [1] [3]. Implementing deep learning algorithms using general purpose architectures such as CPUs poses difficult problems [4] [5]. Deep neural networks (DNNs) are very resource hungry. Due to the von Neumann architecture used by modern CPUs, the sequential loading of data in CPU registers makes the inference process slower than the streaming oriented massively parallel architecture of GPUs [2] [4]. Still, GPUs energy usage tends to be relatively high [5] [6]. This leads to the requirement for more specialized hardware to be able to implement deep learning algorithms and achieve high performance with low energy, especially in the context of embedded devices where the energy budget is limited.
Several researchers are now focusing on developing computer hardware better suited to the realities of deep learning implementation [7]. Application-Specific Integrated Circuit (ASICs) have become the most commonly explored solution, integrating techniques such as in-memory computing, network on-chip, high bandwidth memory and data reuse [5] [7].
The fact that specialized hardware for DNNs depends on the ever changing algorithms and system requirements makes the non-recurring engineering costs of developing ASICs unreasonably high. Consequently, Field-Programmable Gate Arrays (FPGAs) have become increasingly adopted to accelerate machine learning algorithms [8] [9] [10]. They offer a high level of re-configurability making them a preferable solution to ASICs which is desirable in this research area. Jointly with research on hardware accelerators, researchers have been focusing on improving deep learning algorithms, with techniques such as quantization and binarization [11]. Binarized Neural Networks (BNNs) [12] have emerged as a practical solution. Hence, several adaptations to BNNs have been explored. Recently, Look-Up Tables (LUT) have been considered to adapt BNNs to the underlying architecture of FPGAs [13] [14]. In particular, the teacher-student knowledge distillation (KD) approach developed by Chidambaram et al. to build power efficient tiny binary neurons (PoET-BiN) demonstrated

interesting results in the context of image classification with a convolutional neural network (CNN) [13]. However, their approach was subject to inherent limitations that made it unsuitable for a direct application on the convolutional layers of a CNN.
In light of these challenges, we define the main objectives of this thesis to be threefold: explore, enhance and exploit.
In the exploration phase, we look at a different network compression algorithms. Particularly, we combine tensor factorization and binarization and try to derive a LUT-based NN from there.
Therefore, in the enhancement phase, we worked on making PoET-BiN a more suitable candidate for an application on the convolutional layers of a CNN. More specifically, we modified the original work done by Chidambaram et al. to let PoET-BiN's training algorithm account for more features present from the input features space.
Finally, we realized that a regular CPU is ill-suited to exploit LUT-based NNs without an FPGA. Furthermore, even FPGAs with a limited number of LUTs may be ill-suited. Therefore, to exploit LUT-based NNs, we introduce a novel process architecture capable of processing the inference of up to 21 LUTs per clock cycle.
The thesis outlines as follows:

- Chapter 2 introduces key concepts needed to understand the remainder of the thesis.
- Chapter 3 provides a thorough overview of the SOTA in various DNN optimization domains ranging from pruning to knowledge distillation (KD) and tensor factorization.
- Chapter 4 introduces a novel compression approach, binarized tensorized neural networks (BTNN).
- Chapter 5 analyzes the limitation of the PoET-BiN training process and explores various optimizations to the original algorithm.
- Chapter 6 presents a specialized processor architecture designed for the purpose of LUT-based NNs.
- Chapter 7 synthesizes and concludes the thesis. The chapter also describes some limitations encountered in this work and provides some avenues for future works.

CHAPTER 2 BACKGROUND

To better understand the contents of this thesis, a few concepts need to be introduced. In this chapter, we describe how regular NNs learn. We then go over CNNs and BNNs before introducing decision trees and exploring two common learning algorithms, Adaboost and random forests. Finally, we introduce some concepts related to tensors which are used to explore a novel compression technique for neural networks in chapter 4.

2.1 Vanilla neural networks

NNs are able to learn and apply knowledge acquired during a training process. The process relies on partial derivations of a loss function with respect to the parameters of the network and error propagation. In the context of the training algorithm, the network requires a prior mapping of inputs, X, to outputs, Y known as training data. In the following sections, we describe how this algorithm learns patterns. We start by describing the forward propagation of data in a neural network which tries to predict some properties of an input value X.

The known output that is mapped to X, namely Y, is then used to compute the prediction error. This error is then propagated backward in the network and used to update its parameters.
A four-layer neural network is shown in Fig. 2.1. The input is represented by the matrix X and the predicted output of the network by the matrix $\hat{Y}$. The layer-wise parameters of the networks are represented by the matrices W^{L_1}, W^{L_2} and W^{L_3} with dimensions $\{3x4\}$, $\{4x3\}$ and $\{3x1\}$, respectively.

2.1.1 Forward propagation

Forward propagation represents the propagation of values in the network from left to right. The first layer is the input layer and the last layer is the output layer (L_0 and L_3, respectively, in Fig. 2.1). Layers in-between are the hidden layers. L_0's input values are the training examples X. In the upper layers, neurons apply activation functions to their inputs to compute their outputs.

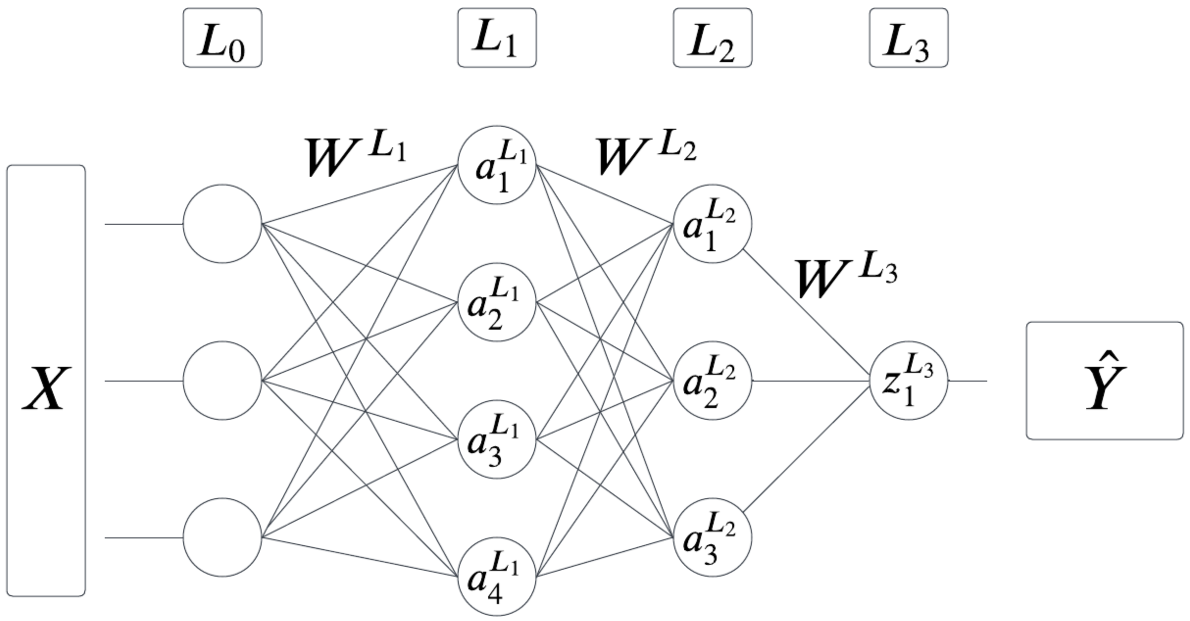


Figure 2.1 Vanilla feed forward neural network with 2 hidden layers.

The output of a neuron a_n^l, where l is the index of the layer and n is index of the specific neuron on that layer, is computed using equations (2.1) and (2.2):

$$z^l = (W^l)^T \times a^{l-1} + b^l, \tag{2.1}$$

$$a_n^l = \sigma(z^l), \tag{2.2}$$

where z^l is the pre-activation vector input of the neuron, σ is the activation function, a^{l-1} the vector containing the outputs of the previous layer, W^l the weight matrix of the current layer and b^l the bias of the current layer. At the end of the forward propagation, we get an estimated value of the property we were trying to predict, $\hat{Y}$.

2.1.2 Back propagation

The weights in the network are learned during training using an optimizer. A popular optimizer is the gradient descent optimizer. To apply gradient descent, we start by forwarding a single data point in the network before computing the estimation error. The error will then be propagated backward in the network to update the weights. A popular error function is the L2 loss function C shown in equation (2.3):

$$C(W_{L_1}, ..., W_L) = \frac{1}{2}\sum ||\hat{y} - y_i||^2 = \frac{1}{2}\sum ||\sigma(W_L^T...\sigma(W_{L_2}^T\sigma(W_{L_1}^T x_i))) - y_i||^2, \tag{2.3}$$

where x_i, y_i are the ith training input and outputs respectively.

Note that C must be a convex function and therefore has a minimum as seen in Fig. 2.2.

The goal of this process is to minimize the error by bringing the estimated value $\hat{Y}$ closer to the actual value Y. The minimum value of the error can be found by finding the right set of parameters W that minimizes the overall loss. When applying gradient descent, we update the weights by moving the error in the direction opposed to the gradient of the loss function. Each weight matrix is updated using equation (2.4):

$$W_{L_x}^{t+1} = W_{L_x}^t - \alpha\vec{\nabla}C(W_{L_x}^t, W_{L_{x-1}}^t, ..., W_{L_1}^t). \tag{2.4}$$

In Fig. 2.2, we observe the loss, C, with respect to the parameters of the network. The initial estimation error is C_0. The target minimum value is C_{min} and the gradients, $\frac{\partial C}{\partial W}$, successively bringing the predicted values closer to the target values by updating the network parameters.

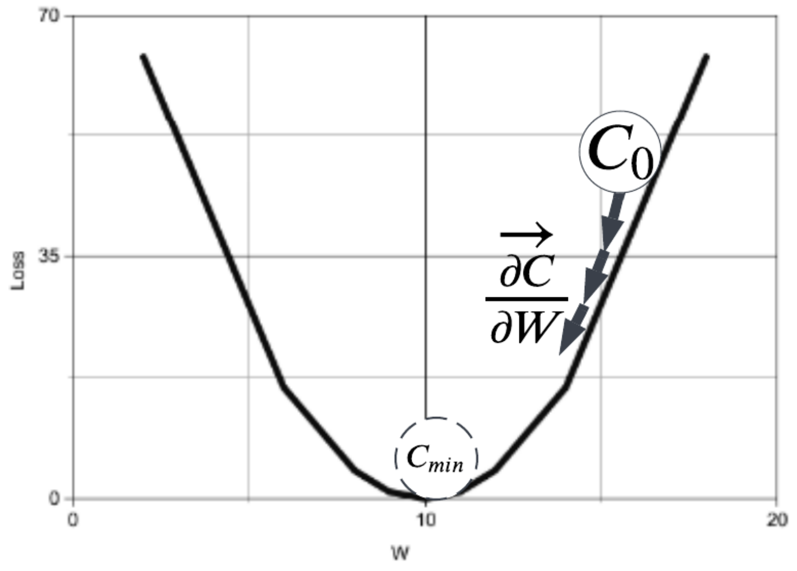


Figure 2.2 Convex loss function and gradient descent.

2.1.3 Classification

The previous section introduced an approach that can be used for regression as-is. However, in the case of a multi-class classification problem, the output layer of the network has multiple neurons. The output of those neurons are called logits. To evaluate the class associated to the input, each logit will contain the probability of the input being of a particular class. In such a case, the number of output neurons equals the number of classes in the dataset. To assign a class to the prediction, the model applies a softmax function, shown in equation 2.5, that associates each logit to a probability of that logit being of a particular class:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{Q} e^{z_j}}, \tag{2.5}$$

where i is the output of the ith neuron on the output layer and Q is the number of classes in the dataset.

2.2 Convolutional neural networks

The fully-connected (FC) layers of a neural network perform well at classifications and estimations. However, they lack in their ability to retrieve spatially related information from input features. Convolutional layers introduce filters that can extract information from the input features [15]. A convolutional layer consists of a learnable set of filters.
A simple filter is a matrix $K \in R^{k_1 \times k_2}$ as shown in Fig. 2.3 in which the set of parameters w_i are learnable during training.

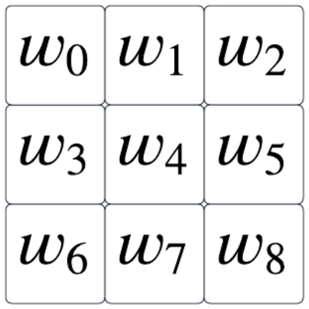


Figure 2.3 Example filter of a CNN.

An output feature map is generated by applying a convolution operation sequentially to the input features while sliding the filter through it. In Fig. 2.4, we can observe this operation where a convolution is applied on an image X with a filter K producing an output feature map Y. An example of the sliding principle shows a convolution on the blue and green areas of the image generating a blue and a green output, respectively.

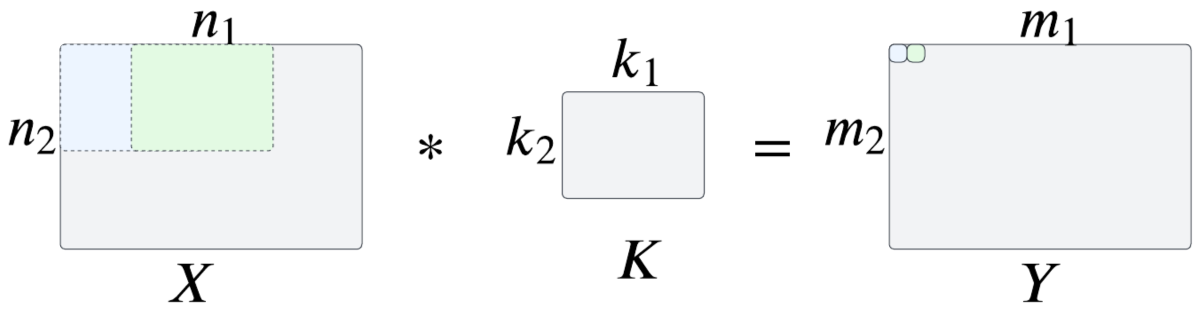


Figure 2.4 Convolution operation on a two dimensional input and a two dimensional filter.

In practice, $n_1 = n_2$, $m_1 = m_2$ and $k_1 = k_2$. Dumoulin et al. found that in this case, the output size m_1, m_2 can be calculated using equation (2.6) [16]:

$$m_1 = \lfloor \frac{n_1 + 2 \times padding - k_1}{stride} \rfloor + 1, \tag{2.6}$$

where the *padding* is the amount of pixels (rows and columns) added to the input image and the *stride* is the amount of pixels by which to move the kernel K in horizontal and vertical directions during the convolution operation. In Fig. 2.4, $padding = 0$ and $stride$ is about half the kernel size. Generally, the inputs and outputs of a convolutional layer and the filter are three dimensional tensors with the third dimension representing different channels.

2.3 Binarized neural networks

In a binarized neural network (BNN), we reduce the available range of values to only 2 bits for the weight matrices and the neuron activations.
There are mainly two approaches to binarize the weight matrices: stochastic and deterministic. In stochastic binarization, the binary value of a floating point input is expressed as a probability density function. This usually leads to better accuracy of the network predictions. However, it requires more resources during training and therefore deterministic binarization

is preferred [12]. Deterministic binarization uses the sign function as shown in equation (2.7).

$$W_b = \begin{cases} -1, & \text{if } W_r < 0 \\ 1, & \text{otherwise} \end{cases} \tag{2.7}$$

W_r and W_b are the real and binary weights, respectively.
To binarize the activations, the hard-tanh activation is used. On an embedded device, switching the representation of -1 to zero helps in applying multiplications using only Boolean operations [17]. More specifically, they can be performed using XNOR operations as shown in Table 2.1.

x_1	x_2	z
0(-1)	0(-1)	1
0(-1)	1	0(-1)
1	0(-1)	0(-1)
1	1	1

Table 2.1 XNOR operation

Furthermore, a pop-count operation is used to count the number of ones in a bit sequence replacing the accumulation. Courbariaux et al. also demonstrated that batch-normalization was costly in terms of operation on an embedded device [18]. They developed a new approach that relies on bit-shifts to compute the batch normalization.
Finally, the computation of the gradient w.r.t. the network parameters need to take binarization into account. The same applies to the activations. To avoid having most of the gradients being equal to zero by derivating the sign function, Courbariaux et al. proposed to use a straight through estimator shown in equation (2.8) [18]:

$$\frac{\partial C}{\partial W_r} = \frac{\partial C}{\partial W_b}. \tag{2.8}$$

Equation (2.8) effectively bypasses the sign function while computing the gradient of the loss function C. For activations, the same straight-through estimator is used, furthermore the saturating effect is taken into account:

$$\frac{\partial C}{\partial a_r} = \frac{\partial C}{\partial a_b} \times \mathbb{1}_{|a| \leq 1}, \tag{2.9}$$

where $\mathbb{1}$ is the indicator function. It is important to note that the gradients are then applied on the real-valued weights which are kept in memory and binarized in the forward pass.

2.4 Decision trees

A decision tree is a graphical learning model in which alternative outcomes are visualizable. In this model, a hierarchical graph is build based on the importance of the input features. The advantage of such a model is that it is easily interpretable. To classify an input, a decision tree partitions the feature space into classes as shown in Fig. 2.5.

A popular algorithm for training decision trees is C4-5 [19]. It builds a decision tree by sequentially selecting input features based on the information gain they provide to the model. However, the computational feasibility of such a decision tree is subject to the dimensionality of the input feature space since the depth of a tree is directly proportional to it [20]. To alleviate this issue and diminish the high variance deeper decision trees suffer from, ensemble learning techniques are explored. There exist three types of ensemble learning techniques: bagging, boosting and stacking [20]. For the purpose of this thesis, we explore bagging with random forests and boosting with Adaboost.

2.4.1 Bagging: Random forests

When building a classifier $H(x)$ for a data-set D that has a certain distribution, P, a problem we may encounter is that we build a classifier that instead of generalizing from the data-set, learns the data-set. This causes high variance in the estimation of unseen data-points. The model overfits. To mitigate this issue, by sampling the original data-set with replacement we can build multiple data-sets D_i, each having their own independent classifier $h_i(x)$. Following this process, we re-combine the multiple weaker classifiers into a strong classifier by averaging their estimations.

$$H(x) = \frac{1}{m}\sum_{i=1}^{m} h_i(x) \tag{2.10}$$

Random forests are a type of bagging algorithm that use decision trees as the weak classifiers. The decision trees are restricted to use k features selected at random to build the trees, which restricts the trees to a specific depth.

2.4.2 Boosting: Adaboost

Boosting is another technique to combine multiple weak learners to build a strong leaner for classification. The general boosting classifier is defined by equation 2.11:

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x), \tag{2.11}$$

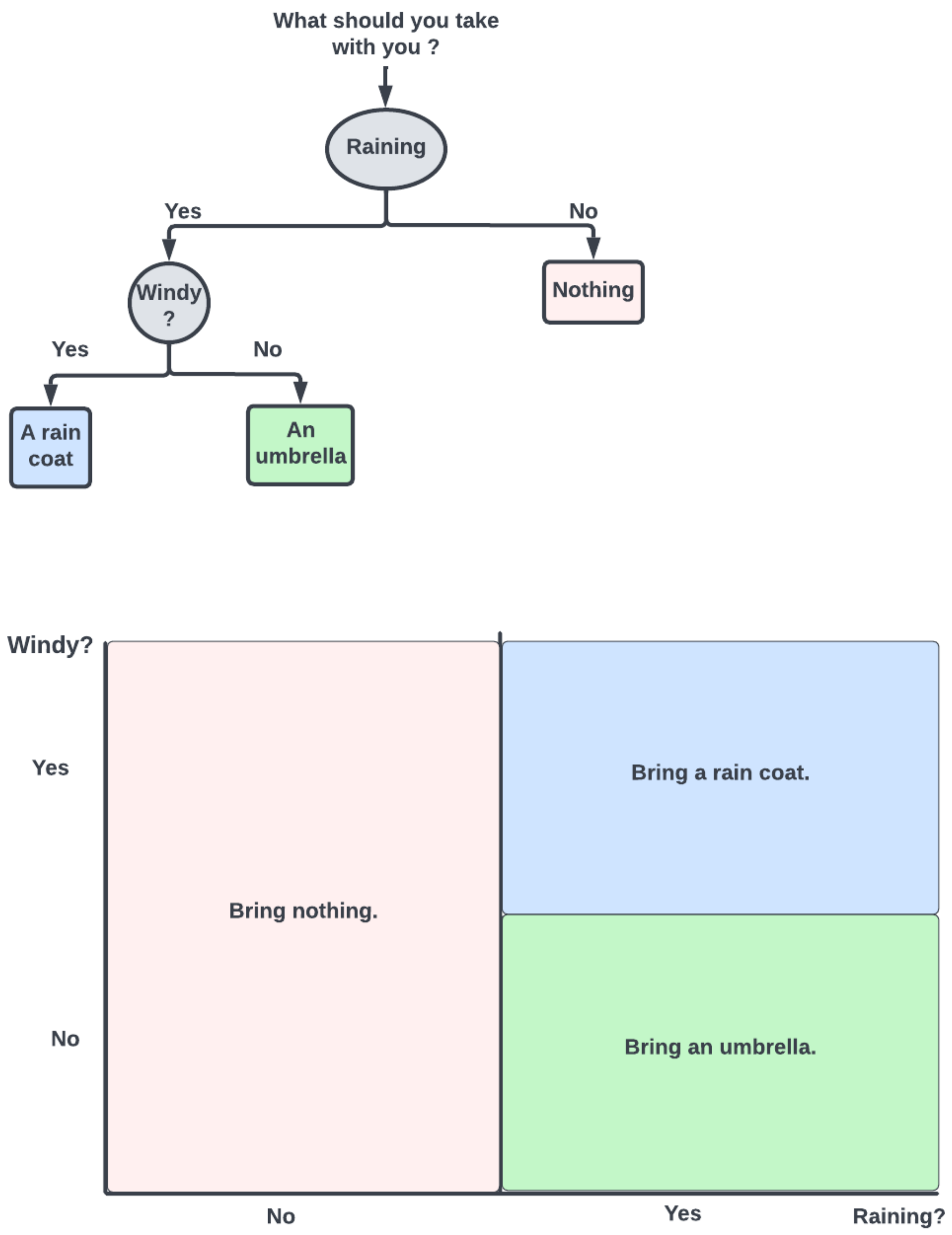


Figure 2.5 Regular decision tree and input features space partitioning.

where $h_t(x)$ is a weak classifier and α_t an adaptable step-size.
The general idea is to aggregate multiple classifiers that each emphasise on the classification of certain examples of the data-set and neglect others. The classifiers are built sequentially

and not in parallel like random forest.
During training, Adaboost builds a strong classifier by iterativally aggregating weaker classifiers. The weaker classifiers are selected based on the output accuracy of the current classifier at time t. Each training example is associated with a weight and each weak classifier is selected based on these weights.
The classification outputs is represented as a vector in the input example space (rather than the input feature space). As an example, imagine a data-set (x_i, y_i) with a single feature x_i and 3 examples: $\{(x_1, y_1 = 1), (x_2, y_2 = -1), (x_3, y_3 = 1)\}$, then the vector $\vec{y} = [y_1, y_2, y_3]$ represents the classified data-set. This is shown in Fig. 2.6.

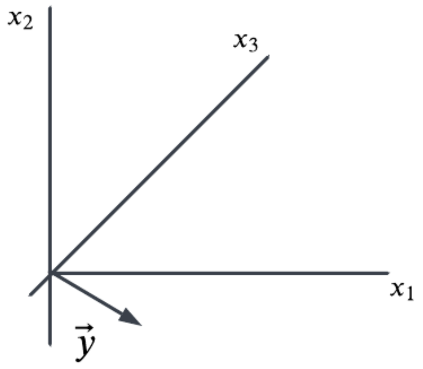


Figure 2.6 Example space.

The prediction vector $\vec{h}_t(x)$ is the vector containing the predictions of every single example $[h(x_1), h(x_2), h(x_3)]$. Note that $y_i \in \{-1, +1\}$ and $h(x_i) \in \{-1, +1\}$.
The loss function in Adaboost is the exponential loss as seen in equation (2.12):

$$l(H) = \sum_i e^{-y_i H_i(x_i)}. \tag{2.12}$$

By minimizing this loss, we move our predicted vector closer to the target. Since this is a sequential process and H is gradually stronger, we define a step size to move towards the minimum of the loss function at each iteration, we compute the next optimal classifier, update the example weights. These are detailed in the following sub-sections.

Next optimal classifier

To find the next best classifier $h_t(x)$ that is added to our sum of classifiers as defined in equation (2.11), we solve the following objective function:

$$h_t = \underset{h \in \Psi}{\arg\min}(l(H + \alpha h)), \tag{2.13}$$

where l is the exponential loss shown in equation (2.12), and Ψ is the set of all valid decision trees, defined by some properties of our model. Using a Taylor approximation, we can solve equation (2.13) and find that the next best classifier is given by equation (2.14):

$$h_t = \underset{h \in \Psi}{\arg\min}(\sum_{h(x_i) \neq y_i} w_i), \tag{2.14}$$

where:

$$w_i = \frac{e^{-y_i H_i(x_i)}}{\sum_j e^{-y_j H_j(x_j)}}. \tag{2.15}$$

Thus, the next best classifier is the classifier that minimizes the weighted error of the training examples.

Step size

To compute the optimal step size α_t, we solve the following objective function:

$$\alpha_t = \underset{\alpha}{\arg\min}(l(H + \alpha h)), \tag{2.16}$$

which solves to:

$$\alpha_t = \frac{1}{2} ln(\frac{1 - \epsilon}{\epsilon}), \tag{2.17}$$

where:

$$\epsilon = \sum_{h(x_i) \neq y_i} w_i. \tag{2.18}$$

Weight update

The final step is to update the weights of each example, w_i under the constraint that $\sum w_i = 1$. The weight update rule is a function of the classification error. If an example was classified its weight value is decreased:

$$w_i^t = \frac{w_i^{t-1} e^{-\alpha_t}}{2\sqrt{\epsilon(1 - \epsilon)}}, \tag{2.19}$$

otherwise it is increased:

$$w_i^t = \frac{w_i^{t-1} e^{\alpha_t}}{2\sqrt{\epsilon(1-\epsilon)}}. \tag{2.20}$$

2.5 Tensors

A tensor is a concept representing a set of values in a specified vector space. A vector, a matrix and a cube are tensors. However, it becomes useful when referring to spaces that are of dimension higher than three. The dimension of a tensor is called its order and this term will be used in the remainder of the thesis.

2.5.1 Rank of a tensor

The rank of a tensor represents the number of linearly independent vectors that constitute the tensor. The other vectors being linearly dependant on the former.

2.5.2 Tensor-Train format

The tensor-train (TT) format is a graphical representation of tensors [21]. It is a convenient notation when working with high order tensors. A tensor $T \in R^{d_1 \times \ldots \times d_n}$ is represented by a circle with n legs as shown in Fig. 2.7. A scalar is simply represented by a circle.

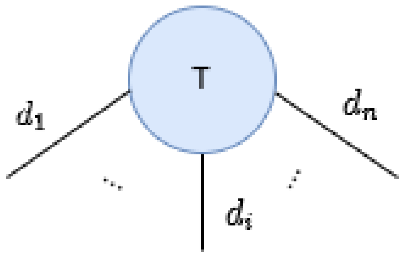


Figure 2.7 Tensor-Train format of a n dimensional tensor.

In Fig. 2.8, we can observe common tensor products represented in their tt format where:

- (a) is the product of two matrices.
- (b) is the product of a matrix and a vector.
- (c) is the inner product of two vectors.

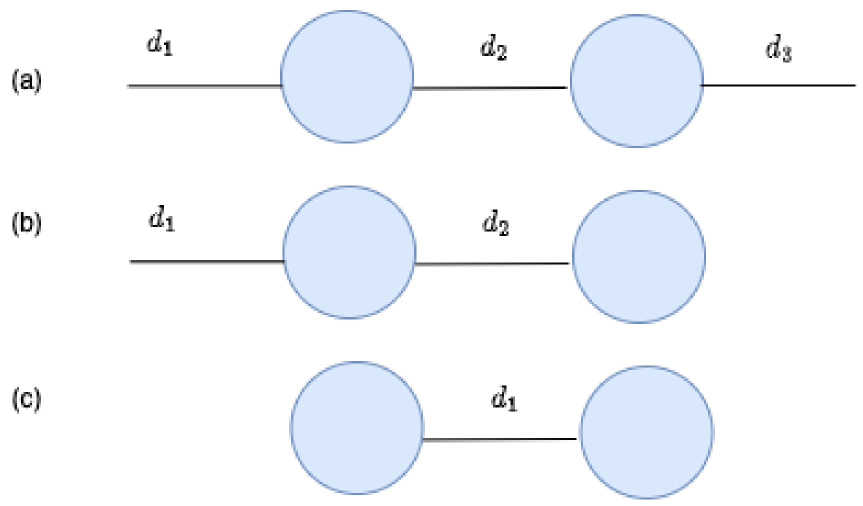


Figure 2.8 Common matrix and vector multiplications in the Tensor-Train format.

2.6 Summary

In this chapter, we have introduced the basic concepts that are necessary for the rest of the thesis to be well understood. We have talked about NNs, CNNs and BNNs. We also introduced decision trees with an emphasis on two common learning methods, bagging and boosting. Finally, we talked about tensors. In the following chapter, we build on top of these basic concepts to present a thorough review of the SOTA in DNN optimization.

CHAPTER 3 LITERATURE REVIEW

Deep Neural Networks (DNNs) are versatile in their requirements across applications. Depending on the desired outcome, their accuracy, latency, throughput, energy and power requirements diverge significantly. With several approaches taken to optimize DNNs, three main optimization types have emerged [22]:

1. Model Optimization
2. System Optimization
3. Joint Model and System Optimization

In the case of model optimization, improvement is made at the algorithmic level of DNNs. They include but are not limited to quantization, pruning and low-rank factorization. The algorithmic modifications can either be hardware aware, meaning that the algorithm is tied to the underlying hardware or hardware agnostic. System optimization works at the deployment level by adapting DNNs to the entire system. When working at this level, the factors taken into consideration are the operating system preemption, the device-server load distribution, and other system dependent variables such as voltage level and energy consumption. These considerations can be statically or dynamically considered to modify the execution of the inference process. To dynamically improve system level processes, one interesting approach is to include an early exit mechanism that outputs a neural network's (NN) prediction and terminates the process once the confidence on the output accuracy is satisfying enough. The junction of these two approaches incorporates model level optimizations and system level optimizations by combining some of the previously mentioned techniques.

In the present work, both of these optimizations types are used making our approach a joint model and system optimization type. In the following sections, different optimization methods that move us steps towards the efficient execution of DNN algorithms on embedded devices are presented.

3.1 Pruning

Pruning compresses and optimizes a neural network inference and training process by removing a set of parameters. Consider a NN model characterized by function $f(x;\theta)$, where θ represents all the parameters of the model. Applying a filter $M \in \{0,1\}^{|\theta|}$ to the model's parameters results in a pruned network such that the model is now defined as $f(x; M \odot \theta)$ which denotes a sparse network [23]. The induced sparsity can either be structured or unstructured. Other distinctive characteristics of pruning algorithms in the literature involve the scoring schemes used to decide which parameters to remove, the number of times the pruning is applied throughout the training process and the way fine tuning is applied in between each pruning iteration.

3.1.1 Structured and unstructured pruning

Unstructured pruning works by removing individual weights from neurons by zeroing them out such that the filter has no inherent structure. Although unstructured pruning ends up reducing memory usage, it leads to a sparse representation of the NN parameters which ultimately does not reduce computation power as the number of multiply and accumulate (MACs) operations remains the same [23]. Another problem with unstructured pruning is the indexing of non-zero parameters that is involved to store the networks parameters after training. Thus, an impact is observed on the computation and memory complexity since all the stored weights now have the index of the parameter in the sparse matrix as a second parameter. The fetching of a single weight then involves a two step operation where the index is fetched followed by the actual parameter [24].

Structured pruning reduces both memory and computation footprint on existing hardware [23]. Instead of removing specific weights, different sets of spatially connected weights, such as all the parameters associated to a specific neuron or a complete filter in a convolutional layer [25] are discarded such that the resulting parameters matrices are structured and the operations can be done using commonly available hardware. However, the compression ratio achieved by unstructured pruning tends to be higher than structured pruning [24].

3.1.2 One-Shot pruning, iterative pruning, fine-tuning

Pruning algorithms can either be One-Shot or iterative. The lottery ticket hypothesis [26] describes an approach to find a pruned network from an initial dense network. Their assumption is that since a pruned network is a smaller version of a larger network, there should be an existing sub-network constructed from the original network that when trained separately should have the same accuracy. Although, the reported results stem from an iterative process, their algorithm describes a One-Shot approach. [27] develop a one-shot pruning scheme that solves an induced structured sparsity problem with a Half-Space Projected Gradient approach. Iterative approaches are more commonly found in the literature. The training of a pruned network is done by gradually compressing the network after each iteration at a certain scheduling rate [23]. To obtain the desired sparsity, the network is first trained then pruned and retrained again to recover the accuracy lost by pruning through a fine-tuning process.

However, the authors of [9] argue that, for structured pruning, a single iteration (One-Shot) has the same outcome as the iterative approach.

3.1.3 Scoring

Different scoring schemes determine the parameters to remove. For structured pruning, the importance of the layer-wise parameters or channel-wise parameters is determined by the magnitude of the l1 or l2 regularizers. Some methods consider the scoring of parameters locally, while others apply it globally without regards to where they reside relative to each other [23].

3.2 Quantization

In Deep Learning, data values (inputs, outputs, activations, gradients, and weights) are usually represented using 32-bit floating point numbers. Quantization was introduced as a means to reduce the size of a network by replacing the traditional floating-point representation of data by other representation strategies that would allow for smaller data sizes. In the literature, 16-bit, 8-bit, 4-bit, 3-bit quantization schemes have been explored [28] [29] [30] [31]. An extreme form of quantization where all values are restricted to binary values has also gained tremendous popularity in recent years [12] [32] [33] [34].

An adequate classification of quantization schemes can be done based on the following factors:

1. Quantization-Aware Training And Post-Training Quantization:
   Quantization-Aware training involves quantizing a model at training time while post-training quantization quantizes pre-trained models [35]. One of the advantages of the later is that it allows for data-free optimization [29] [36] [37]. Depending on the setting, training data might not be available to the model optimizer (i.e. a pre-trained model on cloud). Quantization-Aware training usually renders better performance since the model can be fine-tuned and its hyper-parameters optimized while applying quantization [35].

2. Fixed-Point, Vector And Product Quantization:
   In fixed-point precision quantization, every single data point is quantized to a fixed-point representation, either using scalar values or fixed-point floating point values [30].

Vector quantization on the other hand will derive a vector of values and assign every single data point to their closest representative in that vector using k-means algorithms [38] or other clustering strategies [39] [40]. Product quantization is done by applying vector quantization after partitioning the input matrix into multiple sub-matrices [36] [38]. While most quantization approaches try to minimize the following objective function

$$||W - \hat{W}||_2^2 = \sum_j ||w_j - q(w_j)||_2^2, \tag{3.1}$$

which minimizes the error on the quantized data with regards to the original data, Stock et al. apply a per-layer product quantization where they minimize the error on the output of the layer after quantization instead by minimizing the following objective function [36]:

$$||y - \hat{y}||_2^2 = \sum_j ||x \times (w_j - q(w_j))||_2^2 \tag{3.2}$$

Stock et al. use knowledge distillation techniques (section 3.3) on a trained network to iteratively quantize each layer independently before quantizing the entire network for a final calibration of the generated quantization vectors (codebooks).

3. Single-Precision and Mixed-Precision Quantization:
   Single-precision quantization methods apply the same quantization strategy to all the data points whereas mixed-precision quantization allows for different schemes depending on a per data-point basis [35]. The mixed-precision approach is applied on a per layer or a per channel basis [35] [41]. Nagel et al. adequately explain that mixed-precision quantization methods usually have better performance because they account for the range of the data to compress [37]. As an example, they explain that a layer may have its weights in the range [-0.5, 0.5] while another one may lay in the range [-128,128]. If 8-bit quantization is similarly applied on these ranges, the weights in the former may mostly be assigned a value of zero and thus the network may lose an entire layer's information.

In the above classification of quantization methods, a focus was made on weight quantization. However, quantization is not limited to the weights of neural network. Generally, quantization can be applied on weights, activations and gradients or a combination of them [28]. Applying quantization on weights or activations reduces memory footprint. When applied on weights and activations in the same network, quantization leads to a reduction of computation with the introduction of fixed-point multiplications and additions [28] [30]. Formerly, quantizing was done to optimize the forward propagation of DNNs but recently gradients have also been undergoing this process to optimize training, more specifically, in the case of distributed training on multiple machines [28].

3.3 Knowledge distillation

A popular approach to compressing DNNs is knowledge distillation (KD), where a pair of models work in tandem to transfer knowledge from one, the teacher network, to another, the student network. Two main components play a crucial role in knowledge distillation: the representation of knowledge and the distillation strategy [42].

3.3.1 Knowledge representation

In a NN, knowledge is contained inside the output of the entire model, the outputs of each layer of the model, and the parameters of the model. Gou et al. distinguish three types of knowledge depending on where it is located in a NN: response-based knowledge, feature-based knowledge and relation-based knowledge [42].

- A response-based KD model uses the response of the entire network (logits) as information used by the student network to evaluate the gradient of the loss as shown in (3.3) [43]:
$$\frac{\partial C}{\partial z_i} = \frac{1}{T}(q_i - p_i), \tag{3.3}$$
where z_i is the ith logit of the student network, T is a temperature factor (usually 1), q_i and p_i are the logits associated with the ith class by a the student network and the teacher network, respectively. One can also use the popular Kullback–Leibler (KL) divergence [44]:
$$C = (1 - \lambda)C_{st} - \lambda C_{KL}, \tag{3.4}$$
where λ is an hyper-parameter that serves as a selective trade-off between the student's loss and the KL divergence loss, C_{st} is the student loss and C_{KL} is the KL divergence loss.

- A feature-based KD model uses the intermediate representation of knowledge contained in the output in the hidden layers of a teacher network to train a student network [42]. Particularly, Romero et al. develop a hint-based model where they extract intermediate layer knowledge from the teacher network described as hints to train a student network's guided layers [45]. Then, they recursively train the student network from the first layer up to the guided layer.

- Finally, a relation-based KD algorithm models the relation between pairs of feature maps in a network as a flow of the solution procedure (FSP) matrix where values in the matrix can be computed using equation (3.5) [42] [46]:

  $$G_{i,j}(x;W) = \sum_i \sum_j \frac{F_i^1(x;W) \times F_j^2(x;W)}{h \times w}, \tag{3.5}$$

  where F^1 and F^2 are the input and output feature maps, respectively, i and j are the index in the input and output feature maps, respectively and h and w are the width and height of the feature maps. Furthermore, by representing the relation between feature maps as a correlation matrix, Lee et al. use singular value decomposition (SVD) to capture latent information in the relationship between the layers present in the FSP matrix [47]. Following that information gathering, the student network is trained to capture the knowledge present in the FSP matrix.

3.3.2 Distillation strategy

Gou et al. identify three types of distillation strategies: the offline, online, and self-distillation strategies [42]. They categorize self-distillation as a specialized type of online strategy. The KD models previously described are offline strategies. However, for a larger teacher architecture, offline training is not always convenient since the potential consequent training process of the teacher network is still necessary. Furthermore, Mirzadeh et al. empirically validate that a larger teacher model does not always lead to a better performing student model [44]. They introduce an online training strategy that uses a teacher assistant model that is of a relatively bigger size than the target student network to close the gap in network between a pre-trained teacher and the student network. Zhang et al. introduced a novel type of ensemble learning techniques that leverages the sharing of knowledge between an ensemble of models [48].
Going further, in 2019, Zhang et al. propose a self-distillation strategy where a network passes knowledge in between its own layers [42] [49]. This is done by introducing classifiers at the output of target layers such that the each layer as a teacher and a student in an online training fashion while developing a response-based KD model.

3.4 Tensor-Train decomposition and tensor factorization

Tensor-Train decomposition, first described by Oseledets et al., is one of many tensor factorization techniques [50]. TT decomposition factorizes a high-order tensor of dimension N

into N low-order matrices. More precisely, a tensor $B \in R^{d_1 \times d_2 \times \cdots \times d_N}$ is factorized into N core tensors. In Fig. 3.1, we can visualize the tensor-train format (as described in Section 2.5.2) of N tensors resulting from the decomposition of a tensor of order N.

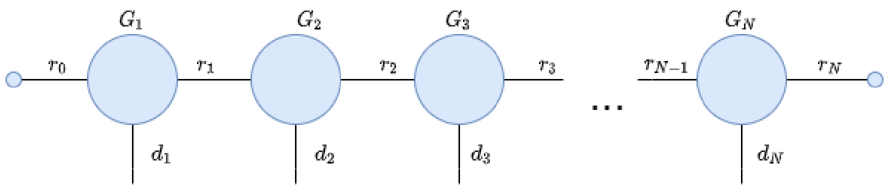


Figure 3.1 Tensor network representation of TT decomposition, G tensors are the cores while r_n indicate TT ranks.

Each tensor G_i is composed of matrices of dimensions $r_{i-1} \times r_i$, where $r_0 = r_N = 1$. Therefore, the first and last core tensors are two dimensional tensors. The total number of matrices for a tensor G_i is given by the value of the ith dimension where $i_n \in [1 \ldots d_n]$. The tensors G_i that are not at the extremeties are three dimensional tensors and the ones at the extremeties are two dimensional tensors. Finally, an element of B can be computed from core tensors as follows:

$$W(i_1, i_2, \ldots, i_N) = \underbrace{G_1[i_1]}_{r_0 \times r_1} \times \underbrace{G_2[i_2]}_{r_1 \times r_2} \times \cdots \times \underbrace{G_N[i_N]}_{r_{N-1} \times r_N} \tag{3.6}$$

The collection of ranks $(r_1, \ldots, r_N)$ is called the TT-rank of the tensor. TT-ranks determine the number of parameters and by careful selection, the original tensor can be expressed in its compressed format. The TT-rank is subject to the following restriction: $r_n \leq min(d1 \ldots d_n, d_{n+1} \ldots d_N)$. The low-rank approximation of the original tensor can then be performed by multiplying these low-rank tensors. TT-decomposition reduces the total number of parameters in the original tensor by exploiting correlations in the eigenspace. By thoughtfully selecting the ranks r_n of the resulting core tensors, we can control the final compression ratio.

Other decomposition techniques provide varying degrees of compression with different trade-offs. In Table 3.1, TT-decomposition is compared to two popular decomposition techniques: CP-decomposition [51] and Tucker decomposition [52].

	CP-Decomposition	Tucker-Decomposition	TT-Decomposition
Computing the rank	NP-Hard	Polynomial	Polynomial
Low-Rank Approx.	NP-Hard	Np-Hard but HOSVD is quasi-optimal	quasi-optimal
Set of low-rank tensors	Not closed	Closed	Closed
Number of Parameters	$\sum_{n=1}^{N}(R)$	$R_1 R_2 ... R_N + \sum_{n=1}^{N}(d_n R_n)$	$\mathcal{O}(R^2 \sum_{n=1}^{N}(d_n))$
Unicity	True	False	False

Table 3.1 Side-by-side comparison of different tensor factorization methods.

3.5 LUT-based neural networks

Portions of this section are excerpted from our article presented at NEWCAS 2022 [53], © 2022 IEEE.

Several hardware architectures have adapted the original works of BNNs for a faster and more energy efficient inference [10]. Xilinx introduced FINN [17] that integrates multiple approaches to map binarized neural networks to FPGA more efficiently. In particular, the authors of FINN altered the original BNN inference flow by replacing multiply and accumulate (MAC) operations with XNOR operations and a pop-count operation [34] [54] . Since all the values are binary, instead of accumulating over an entire bitstring that represents a NN layer's response, only the number of ones need to be counted. Pop-count halves the total number of LUTs required. Furthermore, the computation of the batch-norm step was changed to a simple threshold comparison and max-pooling was replaced by a Boolean OR.

Although these algorithms speed up the original algorithm execution time on the FPGA architecture, the intrinsic BNN architecture is not modified. Because the basic building-block of FPGAs are LUTs, a more FPGA-specific type of BNNs are LUTs based neural networks. Inherently, these components can evaluate any Boolean function. Although different LUT size exist, the common 6 input to 1 output LUTs is used as a reference throughout the rest of this section. To leverage the LUTs present in FPGAs, researchers have tried to develop algorithms that would exclusively use LUTs in the inference phase of DNNs.

Nazemi et al. (NullaNet) [14] and Umuroglu et al. (LogicNets) [55] have taken two similar approaches where they implement complete neural networks using only LUTs. They trim a regular neural network by using quantization and reduced fan-in approaches to achieve the ideal case where 6-input neurons produce single outputs. Depending on the requirements, they can use more than a single LUT per neuron. Since the number of utilized LUTs is exponential with respect to the number of inputs, as shown in equation (3.7) [55]:

$$LUTCost(X,Y) = \frac{Y}{3} \times (2^{X-4} - (-1)^{X}), \tag{3.7}$$

where X is the number inputs and Y the number of outputs. Using equation (3.7), it can be observed that six-bit inputs and a single output bit is convenient as this configuration renders exactly one LUT used per neuron.
However when using input-output mappings other than six inputs to one output, the required number of LUTs can be relatively high as the equation suggests. NullaNet uses "don't cares" to ignore input combinations that are not relevant based on the data-set while LogicNets optimizes the architecture by building neural networks that only have the desired input-output mapping. To achieve this, the authors randomly select input connections from previous layers in quantized neural networks. Wang et al. have taken a similar approach to utilize LUT boolean functions as much as possible [56]. However, they differ from NullaNet and LogicNets because they still have data-paths for various components of their architecture (i.e. accumulators).

3.5.1 PoET-BiN

To train this LUT-based NN, Chidambaram et al. [13] enforce a specific input-output mapping, six or eight inputs to one output, where each neuron is allowed six or eight inputs producing a single output. We focus on the six to one input-output mapping for the remainder of this section. Instead of training a conventional neural network, Chidambaram et al. build a network of decision trees using Adaboost and KD techniques. They build a feature-based KD (refer to Section 3.3.1) model by using a BNN as a teacher network. Specifically, the BNN has convolutional layers and a FC classifier. The convolutional layers have multiple configurations. However, the FC layer is engineered such that the last layer of the hidden layers outputs 60 features. Ten groups of six features are then formed and used by the classifier to output ten classes probabilities.
This architecture is depicted in Fig. 3.2. The student network input training data are the feature outputs of the last convolutional layer $X_{student}$ and the target training data are the outputs of the FC hidden layers $Y_{student}$, prior to the teacher output layer. The student is then trained with the extracted features. Specifically, Chidambaram et al. build a reduced input neural circuit (RINC) classifier per output feature $Y_{student}[i]$. sixty are trained in parallel. Finally, the classifier is retrained on the predictions made by the RINC classifiers. The goal of the i^{th} RINC classifier is to select the best combination of input features to predict

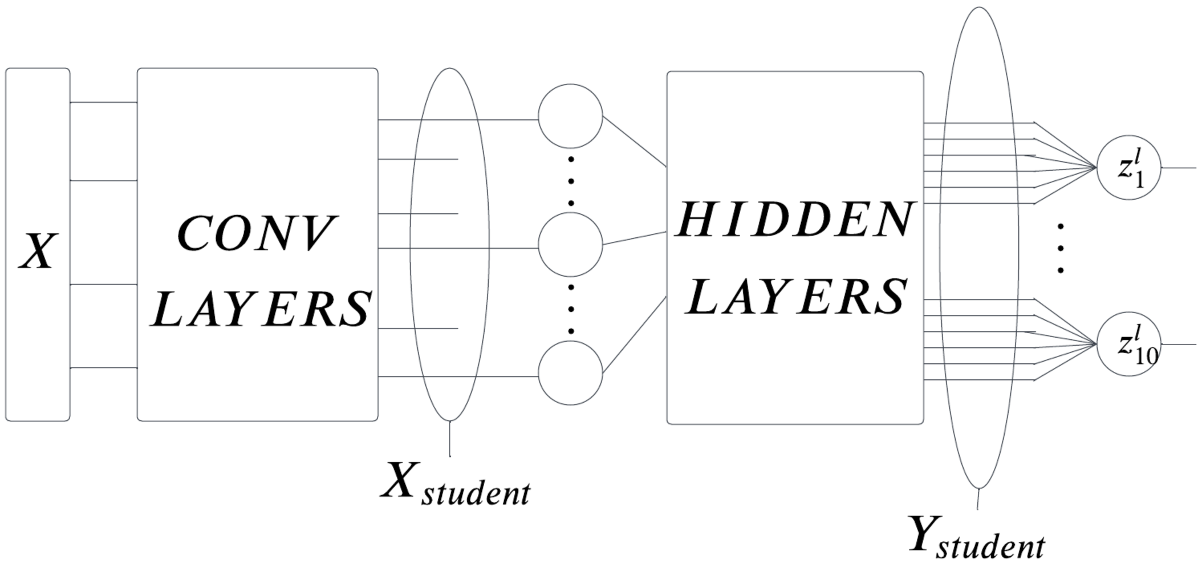


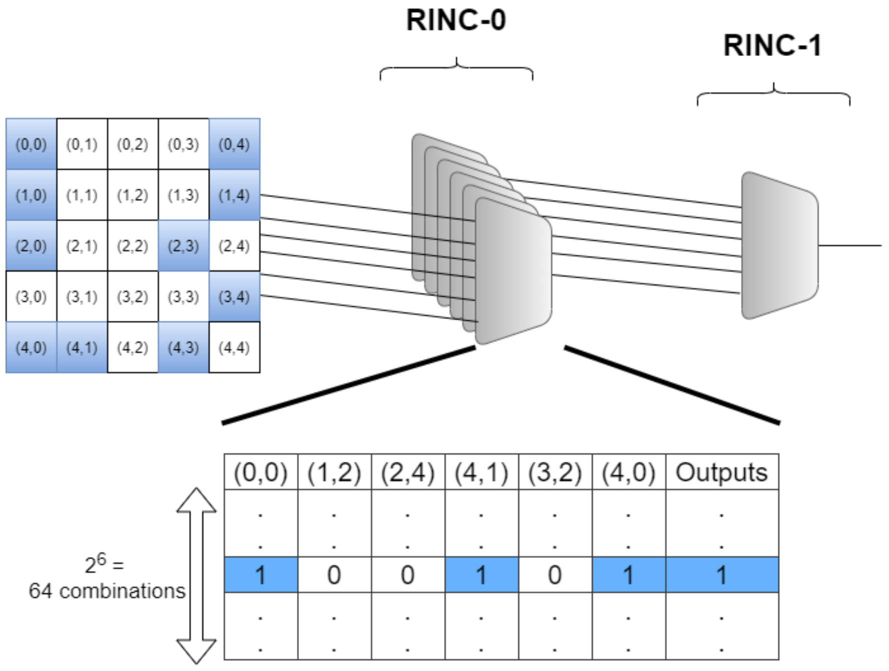
Figure 3.2 PoET-BiN BNN teacher-student architecture.

$Y_{student}[i]$ by using weighted examples. These weights depend on the classification accuracy of each example as per Adaboost. Each weak classifier is a decision tree built by going through the examples and finding the best set of features to make predictions. The decision trees are trained successively and stacked (shown in Fig. 3.3) such that:

- Each RINC is assigned a level where RINC-x indicates the level of the RINC.
- RINC-0s get their inputs from the input feature map $X_{student}$.
- To build a RINC-1, six RINC-0s are built successively, and combined to a RINC-1.
- To build a RINC-2, six RINC-1s are built successively, and combined to a RINC-2.
- The goal is to build a RINC-2.

A limitation of PoET-BiN is that it cannot go deeper than RINC-2, which ultimately constraints the number of usable input features to $6^3 = 216$.

The output assigned for the prediction of a RINC classifier is found by classifying each example with the combination of the selected input features. An exhaustive list of feature combination can be built since the combination of six binary features has exactly 64 possible outcomes. Then, the output is set to be the class that has the highest number of examples that have that exact combination of features for each of the tested combination.



(0,0)	(1,2)	(2,4)	(4,1)	(3,2)	(4,0)	Outputs
.	.	.	.	.	.	.
.	.	.	.	.	.	.
1	0	0	1	0	1	1
.	.	.	.	.	.	.
.	.	.	.	.	.	.

Figure 3.3 Selected inputs for a LUT in PoET-BiN © 2022 IEEE

We observe the representation of a single RINC-1 module in Fig. 3.3. At the RINC-0 level, 6 modules receive their inputs directly from the input image. The RINC-1 level has a single module fed by the outputs of the 6 previous modules at the RINC-0 level. The table shows the LUT content of a single RINC-0 module. The highlighted line is the one that is activated in the particular case of this RINC configuration. The positions of the selected RINC inputs in the feature map are shown at the top of the table. The input at position (0,0) is a 1 and thus is painted in blue whereas at position (1,2) a white background depicts a 0. The output of this particular LUT can be seen in the table.

3.6 Summary

This chapter introduced the SOTA of DNN optimization. In particular, we went over a curated literature review related to NN pruning and quantization, knowledge distillation and finally tensor factorization. These techniques are commonly used to compress DNNs. We then reviewed the state of LUT-based neural networks and formally introduced PoET-BiN which is a building block for the next chapters. In the following chapter, we leverage tensor factorization techniques to compress DNNs in an attempt to derive a new training method for LUT-based NNs.

CHAPTER 4 BTNN : BINARIZED TENSOR DECOMPOSITION OF NEURAL NETWORKS

The present chapter is based on a project done during the course of IFT6760-A (Matrix and Tensor Factorization techniques for Machine Learning) given at the University of Montréal in winter 2020. In particular, the text has been adapted for the matter of this thesis, the results interpretation has been updated and the problem formulation has been revisited.

To build LUT-based neural networks efficiently, multiple algorithms have been introduced in chapters 2 and 3. In this chapter, we visit the compression of a neural network. Such a compression could be used to build a more efficient LUT-based neural network if proven practical. In Section 4.1, we contextualize our approach. Then we visit the tensorization of the FC layers of a neural network with TT decomposition (described in Section 3.4) in Section 4.2. In Section 4.3, we apply binarization to the factorization. We experiment on the MNIST and CIFAR-10 data-sets in Section 4.4 before presenting our results in Section 4.5.

4.1 Introduction

Today, the energy and computational demand of DNNs remain an unresolved issue. Solving the memory storage required to store deep learning parameters inevitably reduces resource usage, ultimately allowing to run the networks on low powered devices. That, along with our goal to run these networks solely using LUTs on an FPGA leads to the exploration of model compression. Indeed, reducing the number of parameters of a neural network reduces the complexity of fitting a neural network inside a network of LUTs using the approach taken by Umuroglu et al. [55].

Courbariaux et al. demonstrated that using binary representation of the weight matrices and the activations of the hidden layers neurons, it is possible to compress the network by a factor of 32 when compared to a 32 bit floating point NN [18]. Furthermore multiplications are replaced by XNOR operations and additions are replaced by pop-count.

Motivated by the redundancy observed in neural network weight matrices, Novikov et al. explored matrix factorization techniques to extract more meaningful parameters out of the original network parameters [57]. The authors explored the tensor train decomposition of the weight matrices of a NN. TT decomposition essentially factorizes a tensor to a low rank approximation by computing different core tensors [50]. They introduce TensorNet and show new state-of-the-art results.

Binarization and TT-decomposition are intended to compress network parameters. In this chapter, we explore the combination of the two compression schemes by applying binarization on the core tensors resulting from the TT decomposition. Similar to the approach taken in this chapter, [58] apply Tucker decomposition on the weights of a neural network and binarize the reconstructed weight matrix. Our approach is different as we focus on the binarization of the core tensors obtained by the TT-decomposition of the weight matrix.

In the following sections, we analyze the effectiveness of binarizing core tensors issued from low-rank TT-decomposition of a neural networks. We show that the compression power of the combination of these two approaches is highly desirable. However, we are not able to learn effectively and suspect conclude that more exploration work is needed to understand how to properly apply the described approach.

4.2 Tensorizing neural networks

Multiple matrix factorization techniques have been applied to DNNs before targeting the compression of the weight matrices [51]. However, TT-decomposition was first used in 2015 by Novikov et al. [57]. Motivated by the huge portion of parameters of a CNN belonging to the fully-connected layers, the compression of FC layers using TT decomposition was desirable.

The weights and biases of a fully-connected layer are matrices and vectors respectively. To work efficiently with weights and biases, they are transformed into high-order tensors. All TT-tensor operations such as summations and products are applicable in the TT-format.

A output neuron of the fully-connected layer of a NN computes the following function:

$$y = Wx + b, \tag{4.1}$$

where $W \in R^{M \times N}$ is the weight matrix, $b \in R^{M}$ is the bias vector and input x and output y as N-dimensional vectors. To further increase the efficiency, the input is also reshaped into a d-dimensional tensor and the linear part of TT-layer can be expressed in tensor form:

$$Y(i_1, ..., i_d) = \sum_{j_1,...,j_d} G_1[i_1, j_1]...G_d[i_d, j_d]X(j_1, ..., j_d) + B(i_1, ..., i_d) \tag{4.2}$$

Another advantage of this method is the compatibility with the gradient descent training algorithms. Novikov et al. show that the gradient of the loss function can be computed efficiently with respect to the cores of W and therefore existing gradient based methods can be easily used for training [57]. The paper shows promising results on CIFAR-10 dataset.

To increase the efficiency, using a low-rank approximation of the core tensors, they are able to compress the weights and biases matrices by very large factors depending on the network architectures while nearly preserving the accuracy of the network. The ranks can then be fine-tuned to get a higher accuracy.

4.3 Binarized TT-decomposition of neural networks

Tensorized NNs provide great reduction in the number of parameters and consequently memory storage. On the other hand, BNNs offer $\approx 32\times$ compression of a NN and the ability to efficiently use this network in the context of a low-powered device [59]. Although BNNs and tensor decomposition methods for DNNs have been studied, TT decomposition of weighs has never been applied to binary networks. Nor has the binarization been applied to the factorized cores resulting from any decomposition. The combination of these algorithms is highly interesting due to the huge compression potential that they offer.

To apply this on the FC layers of a NN, each layer's weight matrix, of shape $W \in R^{[inp]\times[out]}$ is reshaped as illustrated in Fig. 4.1.

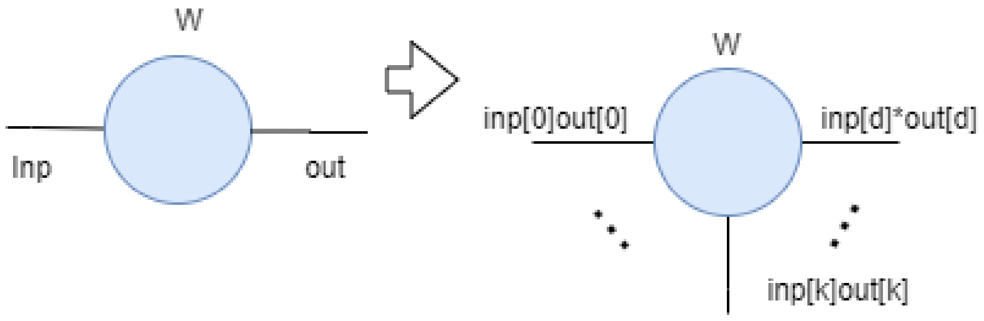


Figure 4.1 Tensor network representation of reshape of the weights matrix W to a tensor

We then perform a TT-decomposition on this d-dimensional tensor as can be seen in Fig. 4.2. Each core is then binarized before being used to compute the output of the layer in the forward pass. In the backward pass, the real-valued weights are kept in memory and gradients update is performed on these real-valued weights. Note that when we multiply binarized cores together the outputs are no longer binary. The batch-normalization fixes this problem. Thus we introduce a batch-normalization layer right after computing the pre-activations. Finally,

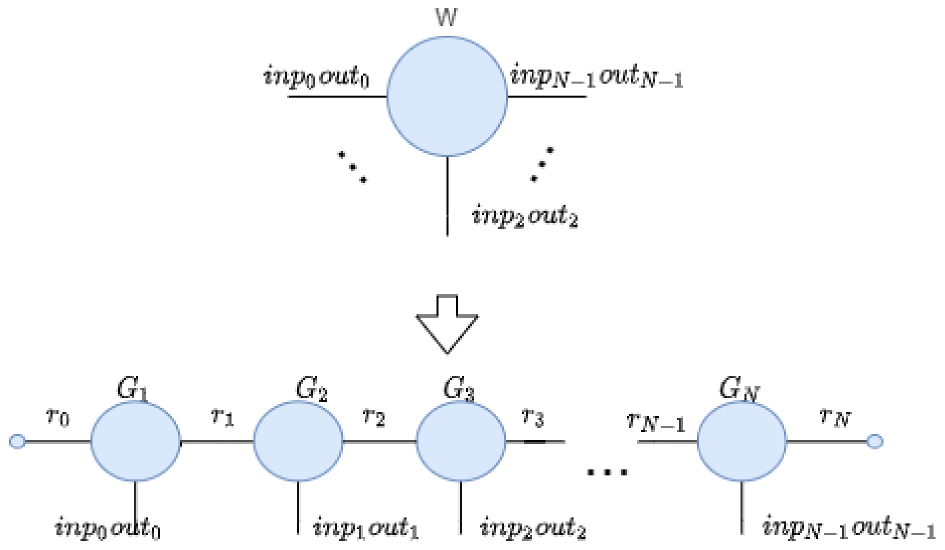


Figure 4.2 TT-decompostion of weight matrix of the weights matrix in tensor network representation

we binarize the activations.

4.4 Experiments

We limit our experiments to FC neural networks without any convolutional layers. Input and hidden layers are replaced by TT-layer. For a non-binarized TT-layer the operations are TT decomposition, batch normalization, ReLu and dropout.

For binarized TT network, the layer consists of binarization of core TT-decomposition tensors, batch normalization, binarization and dropout. Therefore both core tensors and activation maps are binarized. We perform experiments on two well-known data-sets: MNIST and CIFAR-10.

For training, we used a SGD optimizer with cross entropy loss and used exponentially decaying learning rate. The experiments were performed with various TT-ranks to represent different compression ratios and the performance is compared in each case with the non-binarized TT network.

4.4.1 Application to the MNIST dataset

For the MNIST data-set, we used the same architecture as Courbariaux et al. with a 3 layer BNN [18]. The input image size is 28×28 pixels, corresponding to an input vector of size

784. All hidden layers have 4096 neurons and with the same rank. The last layer remains a conventional FC layer without TT-decomposition or binarization. The total number of parameters is thus:

$$\textbf{\#Parameters} = 784 \times 4096 + (4096 \times 4096) \times 2 + 4096 \times 10 = 36806656 \tag{4.3}$$

Since each parameter is a 32 bits floating point value, the total storage size for network weights is :

$$36806656 \times 32 \text{ bits} = 1177812992 \text{ bits} = 147.2 \text{ MB} \tag{4.4}$$

As we can see, this relatively simple neural network already requires considerable amount of memory. Replacing all layers except the output layer with TT-layers, we can achieve some degree of compression. Therefore, we first transform the input parameters into the TT-format with max TT-rank. We then compare it to a network with a TT-decomposition with low-rank. The input images are reshaped to tensors of size $2 \times 2 \times 2 \times 2 \times 7 \times 7 = 784$ and the outputs that are fed to the classification layer to tensors of size $4 \times 4 \times 4 \times 4 \times 4 \times 4 = 4096$.

4.4.2 Application to the CIFAR-10 dataset

For the CIFAR-10 data-set, we were inspired by architecture used in [57]. The input RGB images are of size $32 \times 32 \times 3$ giving 3072 inputs values. The first hidden layer consists of 262144 neurons, while the second and third have 4096 neurons each. Finally, the output layer which will remains unaltered will has 10 neurons. The total number of parameters is thus :

$$\begin{aligned}\textbf{\#Parameters} = \quad & 3072 \times 262144 + (262144 \times 4096) \\ & +(4096 \times 4096) + 4096 \times 10 \\ & = 1895866368\end{aligned} \tag{4.5}$$

With 32 bits floating point values, the storage size is:

$$1895866368 \times 32 \text{ bits} = 6.06^{10} \text{ bits} \approx 7.58 \text{ GB} \tag{4.6}$$

As we can see, this neural network requires a lot of memory for the parameters. By using our approach, we first transform the input parameters into the TT-format with TT-rank capped at 32. We then compare it to a network with a TT-decomposition with very low-rank. The inputs are tensorized to size $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 3$ and the output of first layer is of size $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 8$ and for hidden layers $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4$.

4.5 Results and discussion

In this section, we compare the accuracy of the validation sets of TT-network and binary TT-network for various TT-ranks.

4.5.1 MNIST dataset results

We observe that the accuracy is not greatly affected by the choice of TT-rank. The results can be seen in table 4.1 where BTNNx indicates the TT ranks, compression ratio is the ratio of compression of the number of parameters compared to the uncompressed network and storage value indicates the memory required for weights. Note that Bin. Comp. (Binarized Compression) Ratio is the compression ratio on the binarized TT-cores.
In Table 4.1 TT full rank has best accuracy of 91.1% accuracy while TT-rank 4 decomposition has an accuracy of 88.86%. In this table the number of parameters are sum of calculated number of parameters of layers as for each layer as:

$$\text{no. of parameters for one layer} = \sum_{i=n}^{N} r_{i-1}\, r_i\, inp_i\, out_i \tag{4.7}$$

TT full rank has great compression in terms of number of parameters, while the performance drop is low.

Network	Accuracy	#parameters	Bin. Comp. ratio	Storage
BTNN4	88.86	44240	896.35	163,8 KB
BTNN[Full]	91.1	1,498,160	425.5	169,5 KB

Table 4.1 Experiment results of BTNN on MNIST

In Table 4.1, we observe the difference in the accuracy, the number of parameters, and the conversion ratio when the approximation is done at full-rank, meaning that each dimension of the originally reshaped matrix is used in the decomposition, and when we limit the ranks of the weight tensors to 4. We note that the number of parameters in the approximated version is drastically reduced. Applying binarization further improves the total compression ratio. These values account for the parameters in the hidden layers only where BTNN is applied. The total required storage for our model is shown in the storage column. The proximity of these values reflect the fact that most of the models storage requirements remain in the classification layer. The results presenting the loss over time and the accuracy over time can be seen in Fig. 4.3. In particular, we see that the model is learning effectively for BTNN[Full],

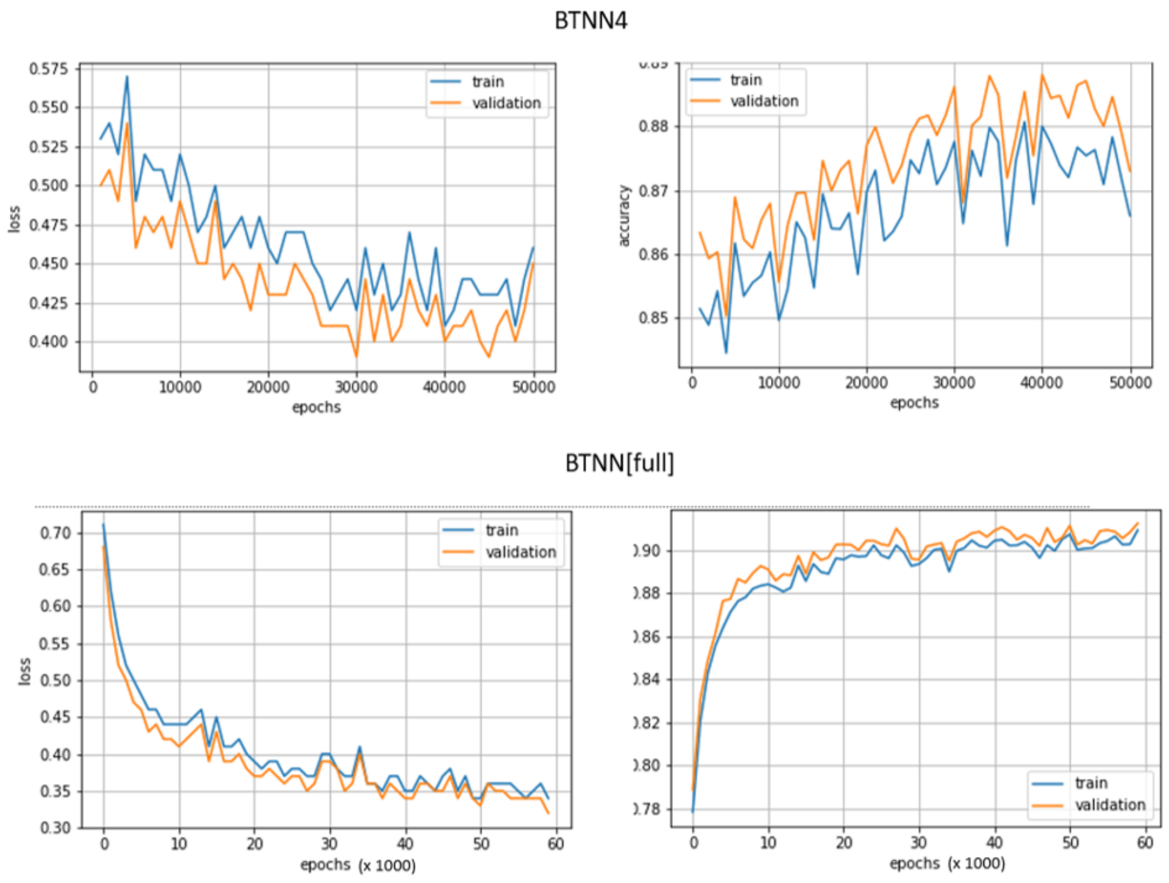


Figure 4.3 Loss and accuracy vs. epoch for the MNIST dataset

the loss is steadily decreasing while the accuracy is steadily increasing. However, in the case of BTNN4, we observe that the network has difficulties generalizing. The decrease in the loss is unstable and the variance the accuracy between successive epochs is high.

4.5.2 CIFAR-10 dataset results

For the CIFAR-10 data-set we considered three different TT ranks: 3, 16 and 32. As shown in Table 4.2, all three perform similarly with an accuracy of ≈ 40%. We also note the huge potential in compression that reducing the TT-ranks offers. We can save around $70KB$ by compressing using BTNN4. However, the limiting factor is the huge drop in accuracy. Applying convolutions prior to the FC layer can potentially help in the learning process. Therefore the results on this data-set are inconclusive. The accuracy on this data-set is low and we observe that the network has difficulties learning. The best accuracy we get for CIFAR-10 is 40%.

Network	Accuracy	#parameters	Bin. Comp. ratio	Storage
BTNN3	40.0	584,978	3,240	146.24 KB
BTNN16	38.2	665,994	55.26	166.48 KB
BTNN32	38.3	864522	42.57	216.13 KB

Table 4.2 Experiment results of BTNN on CIFAR-10, BTNNx indicates the TT ranks

The results presenting the loss over time and the accuracy over time can be seen in Fig. 4.4. For all the explored network architectures, we observe that the network is having difficulties in generalizing which is reflected in the high variance seen in the graphs.

4.6 Conclusion

We combined two existing neural network compression techniques, binarized neural networks and tensorized networks and applied the method to the MNIST and the CIFAR-10 datasets. We applied binarization on the core tensors resulting on the TT-decomposition of the FC layers of a neural network. For MNIST we observed a non-dramatic drop in accuracy when applying a full rank TT-decomposition, and a higher drop for a rank 4 TT-decomposition. For CIFAR-10 we did not observe any indication that the network was effectively learning. With a top accuracy of 40% regardless of the rank of the TT-decomposition, we conclude that the network failed to extract useful patterns out of the training process. We believe that the lack of convolutions in the network makes the learning process tedious for this data-set. Finally, we believe that the described approach compresses networks effectively, and has potential in learning representations. However, it seems like further research needs to be done to evaluate the capacity of the network to extract useful patterns.

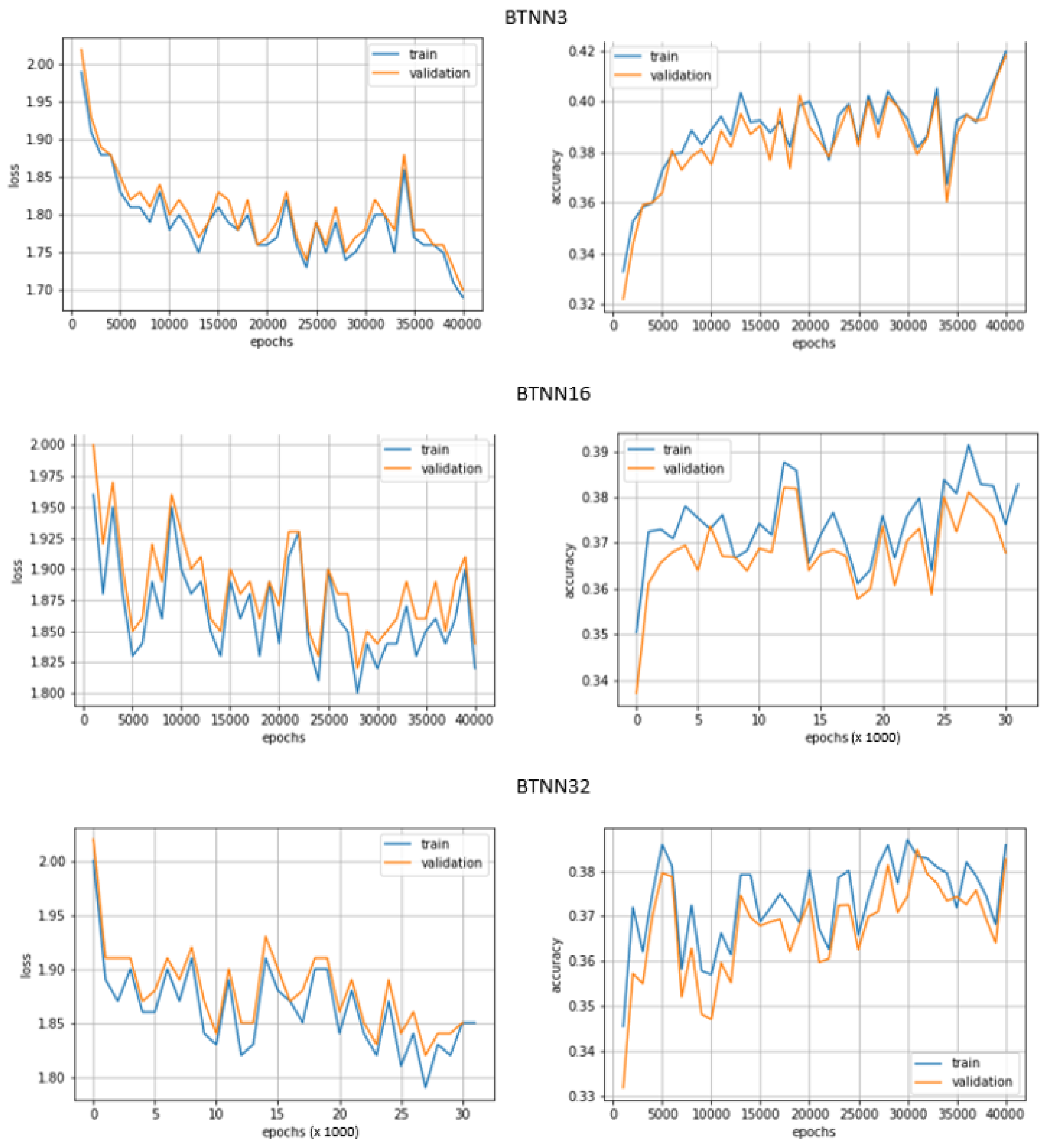


Figure 4.4 Loss and accuracy vs. epoch for the CIFAR-10 dataset

CHAPTER 5 APPLYING POET-BIN TO CONVOLUTIONAL NEURAL NETWORKS

An attempt to compress BNNs and potentially derive a more suitable training algorithm for LUT-based NNs has been presented in the previous chapter. In the current chapter, we focus on improving the original feature-based KD algorithm of PoET-BiN with the goal to apply KD in-between the convolutional layers of a CNN. In particular, we address the inherent problem of the original PoET-BiN algorithm that limits the depth of its stacked decision trees approach to two layers. Then, we evaluate the solution with a deeper PoET-BiN approach. We discover and face an issue we name "the curse of depth" that deeper PoET-BiN models suffer from. Finally, we experiment and validate our enhancements and conclude the chapter.

5.1 PoET-BiN's depth challenge

In Section 3.5.1, we mentioned that the original work of PoET-BiN was limited in the maximum depth of the RINC classifiers. The impact of this constraint is that PoET-BiN cannot aggregate the values of more than $6^3 = 216$ input features to make a prediction. The network architectures that were covered in the original work of PoET-BiN were designed to output 512 features from the convolutional layers. Thus, selecting a maximum of 216 out of 512 features seemed reasonable and the accuracy of the model did not indicate an issue.

When working on replacing convolutional layers with RINC classifiers, Riviello et al. observed that an increase in depth leads to an increase in accuracy when going from RINC-1 to RINC-2 in their testing setting [60]. The considerable increase was due to the number of features considered in the classification. In a RINC-1 classifier, only 36 input features are used compared to 216 for a RINC-2. However, the network architecture that was studied had relatively small convolutions compared to some of the common image centric CNNs such as VGG13 shown in Fig. 5.1.

To further analyze the intuition that a shallow RINC-2 classifier lacks in its capacity to capture enough information from the input features, we compute the number of features that can be extracted from a regular convolutional layer. Consider the CNN architecture, VGG13, shown in Fig. 5.1. With a kernel K of size three, the maximum number of input features to map the outputs of the 4^{th} convolutional layer, $CONV512$, to the outputs of the

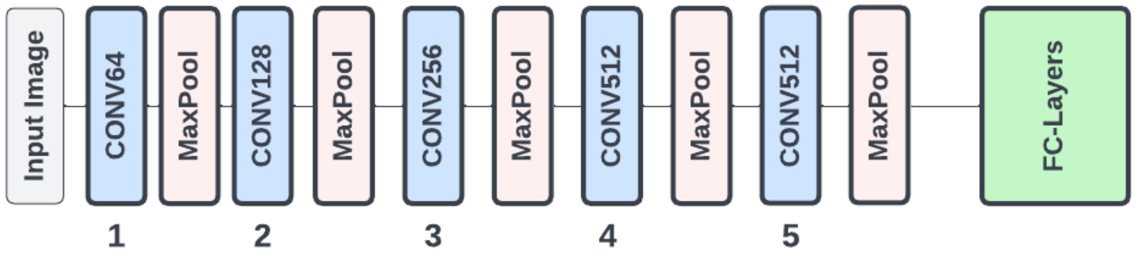


Figure 5.1 VGG13 Architecture

5th convolutional layer, $CONV512$ is given by:

$$512 \times 3 \times 3 = 4608 \tag{5.1}$$

Now the issue is clearer, using only 216 features out of 4096 features can only achieve poor performance.

5.2 Making PoET-BiN deeper

In the remainder of this chapter, we analyze and work at the core of the RINC classifier. In particular, we consider a single RINC-x classifier that classifies the i^{th} output feature from the output feature map, $Y_{student}$, described in Section 3.5.1 and Fig. 3.2. Since improving the classification accuracy of the student network inevitably improves the overall performance, we focus on improving the RINC classifier by increasing its depth.

To address the depth of PoET-BiN, we modify the underlying concept of the original algorithm such that:

- The first RINC level gets its inputs directly from the input features $X_{student}$ (shown in Fig. 3.2).
- Whenever six RINCs of a certain level have been built, they are combined into a RINC of the next level. For example, to build a RINC at layer one, six layer zero RINCs need to be built. The combination of their predictions is used to build a strong RINC-1 classifier. To build a RINC-2 classifier, six RINC-1 classifiers are built and combined to built a stronger classifier.
- We do not impose any restrictions on the maximum depth of the classifiers.

Algorithm 1 outlines our approach to extending the original algorithm of PoET-BiN. In this algorithm, we recursively build a RINC-N classifier, where N is the maximum depth of our classifier. We use the *RincStore* variable to keep a per-layer store of all the RINCs.

Algorithm 1 Recursively build a Rinc-N. classifier.

```
1: procedure RINCN(CurrentDepth, RincStore, weights):
2:     if CurrentDepth = MaxDepth then
3:         done ← True
4:     if current_depth = 0 then
5:         rincN ← RincModule()
6:         rincN.BuildRincLUT()
7:     else
8:         prevLayerModules ← RincStore[CurrentDepth-1]
9:         rincN ← RincModule(prevLayerModules)
10:    rincN.PredictOutLut(train = True)
11:    rincN.PredictOutLut(train = False)
12:    rincN.ComputeAlpha()
13:    weights ← rincN.ComputeWeightsForNextModule()
14:    RincStore[N].append(rincN)
15:    if done = True then
16:        exit
17:    for i ← MaxDepth - 1 to -1 do      ▷ Loop backwards, if six Rinc-i, build Rinc-(i+1)
18:        if layer has six Rinc-i ready then
19:            call RincN(i + 1, RincStore, weights)
20:    call RincN(0, RincStore, weights)
```

The original output error of a single RINC-2 classifier is shown in Fig. 5.2. We plot the accuracy with respect to the number of RINCs. The accuracy being $accuracy = 1 - error$, we can observe that the best prediction accuracy is 96.2%.

We also experiment with a deeper RINC-3 classifier. The result is shown in Fig. 5.3. We observe a disparity between the accuracy on the test set and the accuracy on the training set. This disparity indicates that the network is close to overfitting, which is expected since the total number of features considered now is $6^4 = 1296$ which is over $2\times$ the number of input features.

Two problems stem from this preliminary analysis. First, the training time for a RINC-3 is not sustainable. A RINC-3 classifier can take up to 4 hours to train on a regular CPU. Furthermore, similar to a regular NN, there is no means to stop the training process when the accuracy is plateauing or when the disparity between the accuracies between the train and test sets is too large.

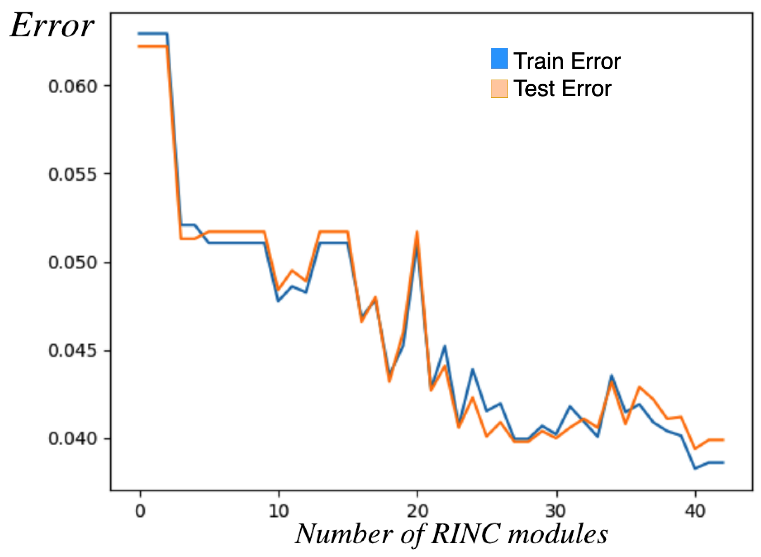


Figure 5.2 Accuracy of a single RINC-2 module
The blue curve shows the accuracy on the training set and the orange curve shows the accuracy on the test set

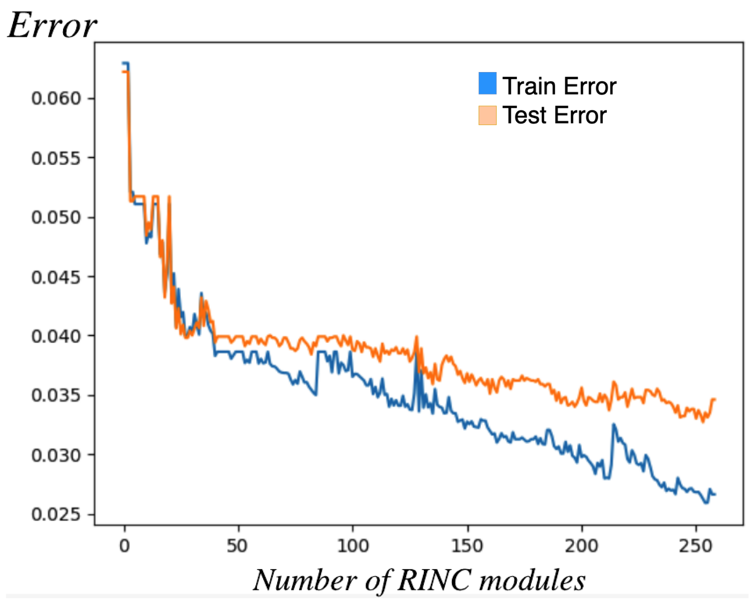


Figure 5.3 Accuracy of a single RINC-3 module
The blue curve shows the accuracy on the training set and the orange curve shows the accuracy on the test set

5.2.1 The curse of depth

We observe that our training time is exponential w.r.t. the number of layers. One solution to that problem is to optimize the time taken to build the RINCs at layer 0. More specifically, we want to reduce the search space for the next best decision tree. Instead of going through the entire feature space to find the next best features, we can introduce randomness in the network. At the time of feature selection, we randomly remove some of the features that were part of the selection space with a probability p. We find that a good trade-off is $p = \frac{1}{2}$.

5.2.2 Early exit

To diminish the overfitting and plateauing issues observed for deeper RINC classifiers, we introduce a variable to account for the maximum acceptable error on the test set. Since we are training a student network which has an indirect effect on the overall classification accuracy, we can optimize the training time and potentially improve the classification accuracy by controlling the maximum tolerable classification error on the test set. Incomplete RINC module are connected whenever that value is reached. For example, if our intent is to build a RINC-2 classifier and we reach the value of early stop before having built a full RINC-2 module, we connect any non-connected RINCs from prior layers into an incomplete RINC-2 module.

5.2.3 Conclusion

In the current chapter, we gave a comprehensive introduction to the inherent limitation of the original training algorithm of PoET-BiN. We proceeded to introduce our solution to that issue and presented an algorithm that mitigates it. While developing and testing the algorithm, we stumbled upon another issue, the curse of depth. To reduce the overall training time, we introduce randomness in the network. Finally, we observe that the deeper the RINC-N classifier architecture, the more likely it is to overfit. In that perspective, we also introduce an early exit parameter that lets the network exit once the maximum acceptable error has been reached on the test set. The work done in this chapter is a building block towards applying PoET-BiN to a full CNN. However, we only tested our approach on a single RINC-N classifier that had the task to make a prediction for the i^{th} output feature. In a regular CNN, thousands of RINC-N classifiers will be used in parallel to make a prediction. The total number of LUTs may reach the limit of the number of LUTs available in an FPGA. In the next chapter, we introduce a processor architecture that does not have this limitation.

CHAPTER 6 ASIP ACCELERATOR FOR LUT-BASED NEURAL NETWORKS INFERENCE

Portions of this chapter are excerpted from our paper presented at NEWCAS 2022 [53].

The goal of this chapter is to present the enhancement of a processor's instruction set with specialized instructions to accelerate the computations required to perform LUT-based neural network inference. This application-specific instruction-set processor (ASIP) reduces latency and increases throughput for LUT-based neural networks, with respect to general purpose CPUs, while maintaining CPU-like generality of processing.

We used Synopsys ASIP Designer [61] to develop our processor's enhanced instruction set. The processor's architecture is based on the TVLIW made available by Synopsys. At full capacity, our developed architecture approximately 2994× faster than the base processor.

The accuracy on different data-sets of all the LUT-based network architectures described in Section 3.5 is proportional to the number of LUTs used. But the number of LUTs is limited by the actual FPGA architecture. To implement larger networks where the number of LUTs required is greater than the available number, Wang et al. suggests to re-use some LUTs through time multiplexing [56]. The approach that we suggest here, however, is to use a general purpose processor with specialized instructions that can compute multiple LUTs outputs per clock cycle. It could be used as an FPGA co-processor or as a standalone processor.

Section 6.1 describes the developed processor and provides an introduction to ASIP Designer from Synopsys. Section 6.2 contains experimental results. Section 6.3 concludes the chapter.

6.1 Specialized processor

Although multiple training algorithms were introduced in the previous section, the rigidity imposed by the direct mapping of BNNs to LUTs leads to structurally constrained neural network architectures. Algorithm 2 outlines the operations required to compute the output of such neurons. Compared to the computation on an FPGA which takes a single clock cycle to compute the prediction, this algorithm is inefficient. In the algorithm, each neuron is represented by the set of $\mathbf{M}$ input features and the output configuration corresponding to the 2^M different output values that it could output based on the combination of its inputs. The binary inputs and the LUT configurations are stored in the data memory (DM). Thus, the

Algorithm 2 Computes the output of neuron @neuron_ref and stores the result in the input array @output_offset

```
1: procedure PREDICT(neuron_ref, output_offset)
2:     LoadInputs(IOMem)                                        ▷ Binary Inputs
3:     LutConfiguration = LUTConfigurations[neuron_ref]
4:     Address = 0
5:     for i ← 0 to M-1 do                      ▷ M is the number of input per LUT
6:         inputValue = FindInput(LutConfiguration.positions[i], IOMem)   ▷ Binary Value
7:         Address = Address | (inputValue << i)
8:     output = LutConfiguration.LUT[Address]
9:     IOMem[output_offset] = output
```

first step is to retrieve the binary inputs array and the LUT configurations array from the DM. Then from all the LUT configurations, the specific LUT representing the neuron is selected. The M different input features positions are encoded in the configuration. Furthermore, the 2^M possible binary outputs of the LUT are also encoded in the configuration. Next, the M binary inputs are concatenated to form an integer value which is used as an address to retrieve the LUT output. In Fig. 3.3, it can be observed that the 6 combined binary inputs led to address 37 (100101 = 37). The LUT configuration at that specific address is 1. Finally, that output feature is stored in the binary inputs array. Another LUT in the following layer uses the output as input. The process induces multiple cycles for a single inference. In a real world setting with a relatively large number of LUTs, the inference process of the whole network is slow as reported in 6.2.

Multiple specialized instructions are added to improve the inference speed of this algorithm. In the following subsections, ASIP Designer from Synopsys is first introduced. The original unoptimized processor architecture is then presented. Finally, the added specialized instructions as well as the different hardware components are presented.

6.1.1 ASIP Designer

ASIP Designer is a tool to design an ASIP processor [61]. It generates a register transfer level (RTL) design of the processor that can be used to program an FPGA, a compiler, an assembler, a linker that outputs assembly instructions and the program's byte-code for that processor from a C++ source code, and a simulator to execute a program without leaving the ASIP Designer's environment. All of these are generated using a single code base in nML.

6.1.2 Base processor: TVLIW

ASIP Designer has a collection of example processors with different characteristics. The developed processor is derived from the TVLIW [62], a 64-bit Very Long Instruction Word (VLIW) processor. Having longer instruction words allows for the execution of the inference of multiple LUTs by using multiple slots of instructions. These slots can run independent instructions in parallel. Each slot that executes an instruction requires its own independent data-path and processing unit. The TVLIW has a 32-bits data memory (DM) and an instruction set architecture (ISA) that consists of the same regular instructions found in MIPS processors.

However, using this ISA, the base processor has limitations when it comes to computing the inference of LUT-based neural networks. When running Algorithm 2 on this VLIW processor, the number of instructions required per neuron causes unsatisfactory delays for a single prediction.

6.1.3 Added hardware components

To adapt the processor described in Section 6.1 for a fast inference of Algorithm 2, specialized instructions and additional resources were added. The processor's architecture is first presented as these instructions heavily depend on it. The following components are added to the TVLIW base processor:

- An input-output register file, named IOMem, with 4096 fields and a data width of 1 bit to store binary inputs and prediction outputs of each neurons. The input data is loaded and the output data is produced in this register file. The latter will eventually become an input to the next level's neurons.
- Eight data memories with 1 read port and 1 write port each. As shown in Fig. 6.1, they contain the LUT configurations aligned on 256 bits of which only the 136 least significant bits are relevant and they store LUT information as follows:
  - Out of the 136 bits, 72 MSBs contain references for the neuron's input addresses (6 inputs per neuron). These inputs are stored in the IOMem memory that is addressable using 12 bits. Hence, 12 bits $\times$ 6 bits = 72 bits.
  - The remaining 64 LSBs of the 136 bits are the outputs possibilities of a 6 inputs LUT as previously described in Fig. 3.3. Depending on the combination of the values stored in the IOMem, we select a single bit among the 64 possible outputs making that bit the output assigned to this LUT. We then store that bit in the

IOMem. It will eventually be used as an input for a LUT at the next level or the output of the model.

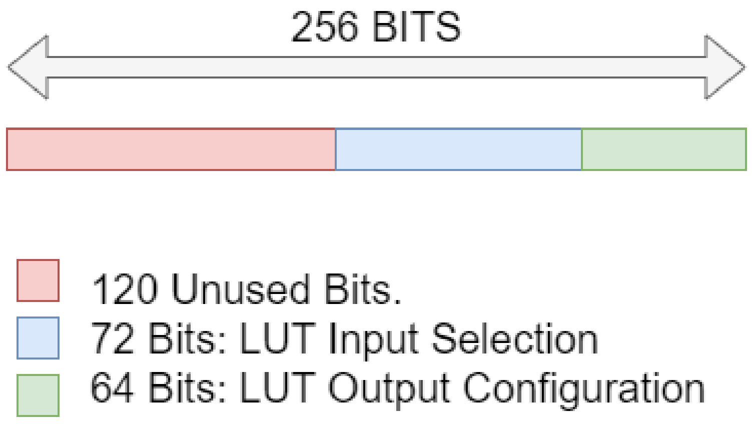

256 BITS
120 Unused Bits.
72 Bits: LUT Input Selection
64 Bits: LUT Output Configuration

Figure 6.1 Data Alignment in Specialized Data Memories

The processor described in Fig. 6.2 contains all the resources and data paths from the base TVLIW processor and the different resources and data-paths needed for the added specialized instructions. It is divided in 5 pipeline stages. It is also relevant to note that Fig. 6.2 showcases a smaller version of the final processor design. It only has 2 slots of the predict specialized instruction described hereafter, whereas the finalized processor contains up to 21 slots.

6.1.4 Specialized instructions

The base processor instruction width has been extended from 32 bits to 512 bits to parallelize our instructions as much as possible. At full capacity, the built processor accelerator can execute the predictions of 21 LUTs in parallel using 21 individual data paths and functional units that we refer to as slots. Furthermore, there are also instructions to load and read data from the IOMem and different data memories described above. Overall 4 instructions were added:

Instruction to store in Data Memories

$$DMX[" \ addr \ "] = " \ dataH0 \ " \ \& \ " \ dataL1 \ " \ \& \ " \ dataL0$$

The total number of data memories is equivalent to the number of slots available to execute the predictions which depends on the specific configuration of the processor. They store all

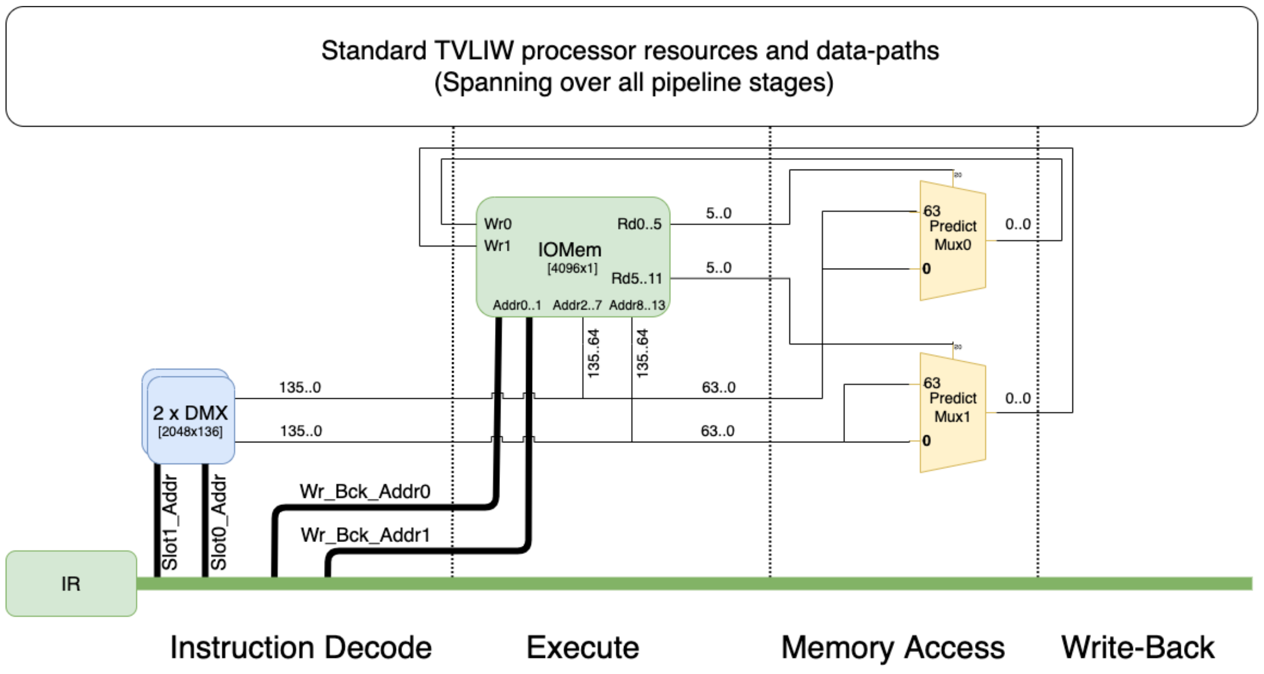


Figure 6.2 Overall Processor Architecture © 2022 IEEE

of the LUTs configurations. Each DM is aligned on 256 bits. By default, ASIP Designer can use integers with length equal or smaller to 64 bits inside the processors instructions. Therefore, we use 4 groups of 64 bits integers in a single instruction to fill in 256-bits per data address. The first 120 bits remain unused. The last 136 bits must contain relevant data. For this reason, we can break down the 256-bits of data into 3 groups of 64-bits, the 4^{th} group remaining unused. The 3^{rd} group, dataH0 shown in assembly instruction 6.1.4, uses only 12 bits out of 64. The first, dataL0, and second group, dataL1, both contain 64 bits of relevant data. The breakdown of this binary sequence is show in Fig. 6.3.

Since a single instruction uses 3 bits $\times$ 64 bits $=$ 192 bits, we can parallelize 2 DM store instructions in a single 512-bits instruction word. Doing so, we use 2 $\times$ 192 bits $=$ 384 bits of the 512-bits instruction word.

Instruction to store in IOMem register

$$IOMem[" \ addr \ "] = " \ value \ "$$

The IOMem is a register file that has 4096 single bit registers. Each of these registers is individually addressable. The input image is loaded in this register using a specialized instruction. 8 slots handle the process in parallel. Each slot moves a single bit into the addressed register. The instruction arguments are passed immediately through the instruction word.

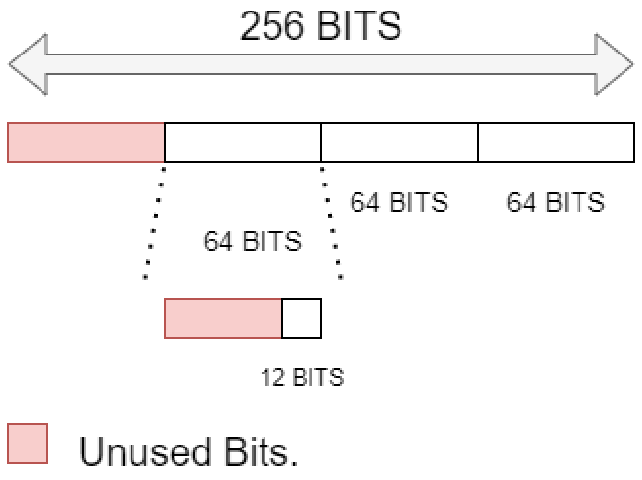


Figure 6.3 Data Memory Store Instruction Breakdown

Instruction to predict LUT output

$$IOMem["outputAddr"] = predict(DMX["lutNumber\ "])$$

The predict instruction is at the heart of the LUT-based inference engine. It can run 8 predict instructions in parallel, each producing 1 LUT output per clock cycle with a latency of 5 clock cycles. A single predict instruction has a width of 24 bits and is composed of:

- lutNumber: 12 bits representing the LUT positions in the connected data memory.
- outputAddr: 12 bits representing the position of the output bit to be placed in the IOMem register.

In the simpler case of two slots that evaluate two LUT outputs, we start by storing their configuration in each individual data memory. Thus, each data memory holds the configuration of one LUT. Then, we write the input image data inside the IOMem registry. Finally, we execute 8 predictions in parallel producing the output of the 8 LUTs that are placed back in the IOMem register and can be read using another instruction.

6.2 Evaluation

We benchmarked our processor using the approach taken by the authors of POET-BIN [13] to train a LUT-based neural network on the MNIST data-set. We extracted the outputs of the convolutional layer and used them as inputs to the POET-BIN algorithm which outputs the LUT configurations. We used these configurations to generate an assembly program that

Table 6.1 LUTs, BRAM, Flip-Flops, F7 Mutiplexers, F8 Mutiplexers Count and number of cycles of predict instruction With regards to the number of Slots of the predict Instructions © 2022 IEEE

#Slots	#LUTs	#Flip-Flops	#BRAM	#F7 Muxes	#F8 Muxes	#Cycles Predict
0	5374	1419	8	479	128	368345
2	144250	8155	40	10575	3886	1290
3	160748	9231	56	14198	5535	860
4	176572	9507	72	17191	7215	645
8	245120	10224	136	30327	13751	323

would load the different memories and registers of the processor. To ensure the correctness of our inference engine, we validated that the outputs of our processor were the same as the outputs of the POET-BIN algorithm when running on a regular processor. We measure our performance against a base TVLIW processor with no specialized instructions. We benchmarked multiple processor configurations with 2, 3, 4 and 8 slots of predict execution. We used the RTL design generated by ASIP Designer to synthesize and implement our processor on an FPGA to produce the benchmark results. We report the FPGA synthesis results in terms of the number of LUTs used, the number of flip-flops used, the number of F7 multiplexers used, the number of F8 multiplexers used and the number of BRAMS used. On the other hand, using ASIP Designer's simulation tools, we executed different programs written in C++ and assembly to retrieve metrics regarding the execution of the inference algorithm. We express these results in terms of the cycle count.

Table 6.1 contains all the retrieved data from our experiences. From the table, we see that the number of resources (LUTs, FF, BRAMS, F7, F8 Muxes) increases while the number of slots increases. A sharp increase in resource usage can be seen, for example the total LUTs usage rises from 5374 LUTs to 144250 LUTs, as soon as the specialized instructions is introduced even with only 2 slots. Along the same line, the number of prediction instructions decreases with the number of slots. After a sharp drop in the number of cycles taken to execute the inference algorithm when introducing our specialized instructions (from 0 slots at 368345 cycles to 2 slots at 1290), the number of cycles taken decreases with hyperbolic relationship with respect to the number of slots. With a 512-bits processor, up to 21 slots can be executed at full capacity since there are 24 bits per instructions. After testing for up to 8 slots, From the gathered data and the graphs 6.4 and 6.5, we can infer the number of cycles that would be taken by any particular configuration by using the observed hyperbolic relation. The equation of the hyperbole is

$$y = \frac{z}{x}, \tag{6.1}$$

where y is the number of cycles, x the number of slots, and z the total number of LUTs in the neural networks.

In our particular experiment, 2580 LUTs are used. By using equation (6.1), we deduce that at maximum capacity, i.e. 21 slots (= 512/24), 123 instructions would be used to complete the inference of this entire network. This renders a speed-up of 2994x when compared to the base processor that uses 368345 instructions to produce the same result. Figure 6.4 and Figure 6.5 help to visually distinguish the discussed relationships. All the developed architectures run on an FPGA at a maximum frequency of 125 MHz and consume 3W of power as estimated by Vivado HLS synthesis tool.

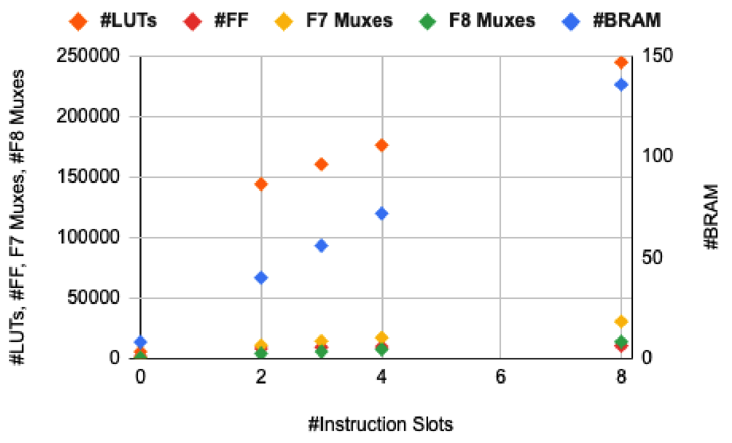


Figure 6.4 Resource usage with respect to the number of slots © 2022 IEEE

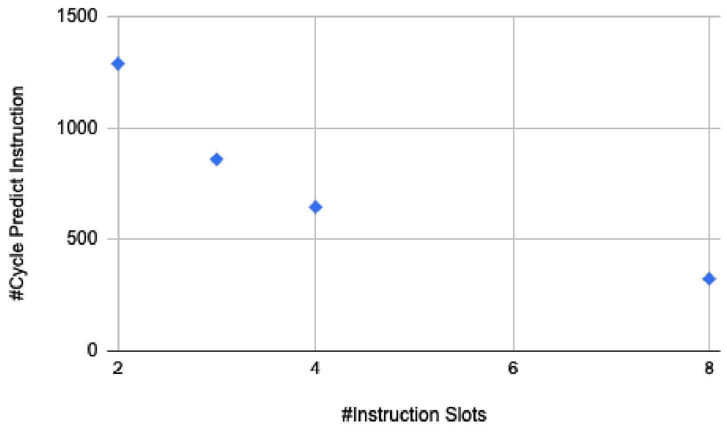


Figure 6.5 Cycle count for predictions with respect to the number of slots © 2022 IEEE

6.3 Conclusion

We have successfully enhanced a processor's instruction set to accommodate for a set of neural networks architectures where each neuron has six binary inputs and a single binary output. The developed processor accelerates the inference process by a factor of 2994x and eliminates some of the work done when considering the sequential architecture of general purpose processors. Our test setup effectively replaced the fully-connected layers of a CNN by RINC modules and ran the inference process on our accelerator. The current architecture is limited by the size of the IOMem register and the available DMs. We have also expressed our results in terms of their implementation on an FPGA. It would be interesting to see an ASIC with this configuration to see the actual energy consumption.

CHAPTER 7 CONCLUSION

This chapter synthesizes the work done in this thesis. We go over the work done in relation to the three phases announced in the introduction: exploration, enhancement and exploitation. We follow by presenting the limitations of our work and open avenues for future research.

7.1 Summary of works

In the present thesis, we started with the goal to optimize LUT-based NNs to apply them to CNNs, particularly to the convolutional layers of a CNN. In this regard, we centered our research around the work done by Chidambaram et al. where the authors introduced PoET-BiN [13]. Before diving in PoET-BiN, we started in an exploration phase where we looked into deriving a compression algorithm that could potentially be used to build LUT-based NNs. We started by going through a thorough literature review, where we identified multiple compression algorithms that had potential in building LUT-based NNs. We discovered that the TT-decomposition of a tensor and binarization of NNs had never been used together and realized that we could potentially leverage their combination to compress a DNN by a very large factor. Even though our empirical analysis of the solution validates a huge compression ratio, we found that our results are inconclusive and the generalization capacity of the produced network architecture was poor. Following that discovery, we turned to the enhancement of PoET-BiN in Chapter 5. We analyzed the limitation of the original PoET-BiN model. We found that the maximum depth of the RINC classifiers were a limitation to the approach and designed an algorithm that was free of those limitations. However, the depth of the RINC classifier introduced a computation limitation as the time complexity was exponential with respect to the depth of the RINC classifier. We also found that depending on the actual size of the input feature space, overfitting could potentially be an issue. To solve "the curse of depth" issue, we introduced randomness in the network by randomly removing some of the potential features for the selection space before building RINC-0 classifiers. To solve the overfitting issue and speed up the learning process as well, we introduced an early stop mechanism with a maximum accuracy error parameter that stops the training process once the value was reached. Finally, we thought about the exploitation of the model in a real world setting. We realized that the total number of LUTs needed for its direct application on the convolutional layers of a CNN had inherent limitations due to the limited number of LUTs available in an FPGA. We developed an ASIP processor that could potentially be used as an FPGA co-processor or a standalone general purpose CPU to process the inference of

LUT-based NNs. We tested and demonstrated a 2994× speedup in the inference time when comparing our processor to a base processor without our specialized instructions. Overall, we have explored multiple solutions to map the convolutional layers of a CNN into LUTs. We then picked and enhanced PoET-BiN for that purpose. Finally, we developed a processor architecture that can be used to process the high number of LUTs that are necessary in the context of convolutions.

7.2 Limitations

In Chapter 4, we can see that the compressed binarized neural networks still manages to perform classifications. Despite a huge compression factor, the impact on the accuracy is negligible on the MNIST dataset. On CIFAR-10, however, the network does not demonstrate any learning and renders poor performance regardless of the TT-ranks of the TT-decomposition. We suspect that a convolutional layer to extract spatial features might be helpful. Another limitation is that the values inside the core weight tensors (resulting from the TT-decomposition) are indeed binary. However, after reconstructing the full weight by multiplying all the core tensors, the values are no longer binary. In Chapter 5, the biggest limitation we faced was the lack of a framework that successively replaces the layers of a CNN by going forward in the layers and retraining the upper layers after replacement. The training process is tedious and needs improvement, especially in the case of a deeper network. Finally, the ASIP processor developed in Chapter 6 suffers from the limitation that the processor has been only optimized for the inference of six inputs to one output LUTs. Other works have shown that different LUT sizes can have better accuracy depending on the task at hand [63]. A second limitation is that a VLIW processor is used while a vector processor might have been more adequate. One main advantage of this approach is that the compiler generated by ASIP Designer could be used alongside our processor without worrying about the re-organization of instructions that happens when a VLIW processor.

7.3 Future research

Future works should take into account the limitations we described in this chapter. In particular, future works would explore the development of a more robust BTNN where the values of the reconstructed weight matrices are strictly binary. Another potential future work would leverage the enhancement we have made to the original PoET-BiN model to develop a training framework that would sequentially replace layers of a CNN, starting from the input image onward. Lastly, the ASIP processor we have developed could be modified to accept different LUTs input sizes. We also recommend dropping the TVLIW and making the processor a SIMD processor.

REFERENCES

[1] T. B. Brown *et al.*, “Language models are few-shot learners,” *arXiv*, 5 2020. [Online]. Available: http://arxiv.org/abs/2005.14165

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks.” [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[3] M. Tan and Q. V. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 10 691–10 700, 5 2019. [Online]. Available: http://arxiv.org/abs/1905.11946

[4] A. A. Awan, H. Subramoni, and D. K. Panda, “An in-depth performance characterization of cpu- and gpu-based dnn training on modern architectures,” *Proceedings of MLHPC 2017: Machine Learning in HPC Environments - Held in conjunction with SC 2017: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 11 2017.

[5] Y. Chen *et al.*, “A survey of accelerator architectures for deep neural networks,” *Engineering*, vol. 6, pp. 264–274, 3 2020.

[6] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate cpu vs. gpu performance without the answer,” in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011, pp. 134–144.

[7] R. Zhao *et al.*, “Hardware compilation of deep neural networks: An overview,” in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.

[8] J. Duarte *et al.*, “Fast inference of deep neural networks in fpgas for particle physics,” *Journal of Instrumentation*, vol. 13, no. 07, p. P07027–P07027, Jul 2018. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/13/07/P07027

[9] Z. Liu *et al.*, “Rethinking the value of network pruning,” *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[10] H. Qin *et al.*, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, 3 2020. [Online]. Available: http://arxiv.org/abs/2004.03333http://dx.doi.org/10.1016/j.patcog.2020.107281

[11] H. Li *et al.*, “Training quantized nets: A deeper understanding,” vol. 2017-December. Neural information processing systems foundation, 2017, pp. 5812–5822.

[12] M. Courbariaux *et al.*, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[13] S. Chidambaram, J. M. P. Langlois, and J. P. David, “Poet-bin: Power efficient tiny binary neurons,” 2 2020. [Online]. Available: https://arxiv.org/abs/2002.09794

[14] M. Nazemi, G. Pasandi, and M. Pedram, “Nullanet: Training deep neural networks for reduced-memory-access inference,” 7 2018. [Online]. Available: http://arxiv.org/abs/1807.08716

[15] J. Zhang and J. Zhang, “An analysis of cnn feature extractor based on kl divergence,” *https://doi.org/10.1142/S0219467818500171*, vol. 18, 7 2018.

[16] V. Dumoulin, F. Visin, and G. E. P. Box, “A guide to convolution arithmetic for deep learning,” 2016. [Online]. Available: http://ethanschoonover.com/solarized

[17] Y. Umuroglu *et al.*, “Finn: A framework for fast, scalable binarized neural network inference,” *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[18] M. Courbariaux *et al.*, “Binarized neural networks: Training neural networks with low precision weights and activations,” *arXiv:1602.02830*, 2016.

[19] J. R. Quinlan, M. K. Publishers, and S. L. Salzberg, “C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993,” *Machine Learning 1994 16:3*, vol. 16, pp. 235–240, 9 1994. [Online]. Available: https://link.springer.com/article/10.1007/BF00993309

[20] A. Criminisi *et al.*, “Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning,” *Foundations and Trends R in Computer Graphics and Vision*, vol. 7, pp. 81–227, 2012.

[21] U. O. Montreal, “Ift6760a: Lecture 9 tensor decompositions-part 1,” 2022. [Online]. Available: https://www-labs.iro.umontreal.ca/~grabus/courses/ift6760_files/lecture-9.pdf

[22] S. I. Venieris *et al.*, “How to reach real-time ai on consumer devices? solutions for programmable and custom architectures,” in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 93–100.

[23] D. Blalock *et al.*, “What is the state of neural network pruning?” 2020. [Online]. Available: https://arxiv.org/pdf/2003.03033.pdf

[24] S. Anwar, K. Hwang, and W. Sung, “Structured pruning of deep convolutional neural networks,” *J. Emerg. Technol. Comput. Syst*, vol. 13, no. 3, 2017. [Online]. Available: http://dx.doi.org/10.1145/3005348

[25] H. Li *et al.*, “Pruning filters for efficient convnets,” *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[26] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” in *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[27] T. Chen *et al.*, “Only Train Once: A One-Shot Neural Network Training And Pruning Framework,” 2021. [Online]. Available: http://arxiv.org/abs/2107.07467

[28] Y. Guo, “A Survey on Methods and Theories of Quantized Neural Networks,” 2018. [Online]. Available: http://arxiv.org/abs/1808.04752

[29] P. Nayak, D. Zhang, and S. Chai, “Bit efficient quantization for deep neural networks,” 2019. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1910/1910.04877.pdf

[30] R. Goyal *et al.*, “Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms,” Feb. 2021. [Online]. Available: http://arxiv.org/abs/2102.02147

[31] R. Banner, Y. Nahshan, and D. Soudry, “Post training 4-bit quantization of convolutional networks for rapid-deployment,” in *Advances in Neural Information Processing Systems*, vol. 32, 2019. [Online]. Available: https://github.com/submission2019/cnn-quantization.

[32] I. Hubara *et al.*, “Quantized neural networks: Training neural networks with low precision weights and activations,” *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018.

[33] M. Courbariaux, Y. Bengio, and J. P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems*, vol. 2015-Janua, 2015, pp. 3123–3131. [Online]. Available: https://github.com/MatthieuCourbariaux/BinaryConnect

[34] M. Rastegari *et al.*, “XNOR-net: Imagenet classification using binary convolutional neural networks,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9908 LNCS, 2016, pp. 525–542. [Online]. Available: https://arxiv.org/pdf/1603.05279.pdf

[35] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” 2018. [Online]. Available: http://arxiv.org/abs/1806.08342

[36] P. Stock *et al.*, “And the Bit Goes Down: Revisiting the Quantization of Neural Networks,” 2019. [Online]. Available: http://arxiv.org/abs/1907.05686

[37] M. Nagel *et al.*, “Data-free quantization through weight equalization and bias correction,” in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-Octob, 2019, pp. 1325–1334. [Online]. Available: https://arxiv.org/pdf/1906.04721.pdf

[38] Y. Gong *et al.*, “Compressing Deep Convolutional Networks using Vector Quantization,” 2014. [Online]. Available: http://arxiv.org/abs/1412.6115

[39] Y. Choi, M. El-Khamy, and J. Lee, “Towards the limit of network quantization,” *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[40] E. Park, J. Ahn, and S. Yoo, “Weighted-entropy-based quantization for deep neural networks,” in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, 2017, pp. 7197–7205.

[41] K. Wang *et al.*, “HAQ: Hardware-aware automated quantization with mixed precision,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, 2019, pp. 8604–8612.

[42] J. Gou *et al.*, “Knowledge distillation: A survey,” *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 6 2021.

[43] G. Hinton and J. Dean, “Distilling the knowledge in a neural network,” *ArXiv*, 2015. [Online]. Available: https://arxiv.org/pdf/1503.02531.pdf

[44] S. I. Mirzadeh *et al.*, "Improved knowledge distillation via teacher assistant," 2019. [Online]. Available: www.aaai.org

[45] A. Romero *et al.*, "Fitnets: Hints for thin deep nets." International Conference on Learning Representations, ICLR, 2015.

[46] J. Yim *et al.*, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-January, pp. 7130–7138, 11 2017.

[47] S. H. Lee, "Self-supervised knowledge distillation using singular value decomposition," 2018. [Online]. Available: https://arxiv.org/pdf/1807.06819.pdf

[48] Y. Zhang *et al.*, "Deep mutual learning," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4320–4328, 12 2018.

[49] L. Zhang *et al.*, "Be your own teacher: Improve the performance of convolutional neural networks via self distillation," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 3712–3721, 10 2019.

[50] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011. [Online]. Available: https://doi.org/10.1137/090752286

[51] S. Rabanser, O. Shchur, and S. Günnemann, "Introduction to tensor decompositions and their applications in machine learning." *CoRR*, vol. abs/1711.10781, 2017. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1711.html#abs-1711-10781

[52] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Journal of Mathematics and Physics*, vol. 6, pp. 164–189, 4 1927. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/sapm192761164https://onlinelibrary.wiley.com/doi/abs/10.1002/sapm192761164https://onlinelibrary.wiley.com/doi/10.1002/sapm192761164

[53] M. Traore, J. M. Pierre Langlois, and J. Pierre David, "Asip accelerator for lut-based neural networks inference," in *2022 20th IEEE NEWCAS Conference (NEWCAS)*, 2022, pp. 524–528.

[54] M. Kim and P. Smaragdis, "Bitwise neural networks for efficient single-channel source separation," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2018-April, pp. 701–705, 9 2018.

[55] Y. Umuroglu *et al.*, “Logicnets: Co-designed neural networks and circuits for extreme-throughput applications,” *Proceedings - 30th International Conference on Field-Programmable Logic and Applications, FPL 2020*, 2020.

[56] E. Wang *et al.*, “Lutnet: Rethinking inference in fpga soft logic,” *Proceedings - 27th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019*, 2019.

[57] A. Novikov *et al.*, “Tensorizing neural networks,” in *Advances in Neural Information Processing Systems*, C. Cortes *et al.*, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/6855456e2fe46a9d49d3d3af4f57443d-Paper.pdf

[58] A. Bulat *et al.*, “Matrix and tensor decompositions for training binary neural networks,” 2019. [Online]. Available: https://arxiv.org/abs/1904.07852

[59] Y. Wang *et al.*, “Sub-bit neural networks: Learning to compress and accelerate binary neural networks,” 2021. [Online]. Available: https://github.com/yikaiw/SNN

[60] A. Riviello, “Binary neural networks for keyword spotting tasks,” 2020. [Online]. Available: https://publications.polymtl.ca/5449/

[61] *ASIP Designer Overview Manual*, Synopsys, Mountain View, CA, 2019.

[62] *ASIP Designer Tvliw Core Processor Manual*, Synopsys, Mountain View, CA, 2019.

[63] S. Chatterjee, “Learning and memorization,” in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 755–763. [Online]. Available: https://proceedings.mlr.press/v80/chatterjee18a.html