INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAT005

Thèse de doctorat

Approximate Computing for Embedded Machine Learning

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Electronique et Optoélectronique

Thèse présentée et soutenue à Palaiseau, le 24/03/2021, par

Xuecan YANG

Composition du Jury :

Patrick GIRARD Directeur de Recherches, CNRS	Président
Roselyne CHOTIN Maître de Conférences-HDR, Sorbonne Université (LIP6)	Rapporteur
Jacques-Olivier KLEIN Professeur, Université Paris Saclay	Rapporteur
Fabrice MONTEIRO Professeur, Université de Lorraine	Examinateur
Atillio FIANDROTTI Assistant professor, Université de Turin	Examinateur
Lirida NAVINER Professeur, Télécom Paris (LTCI)	Directrice de thèse
Sumanta CHAUDHURI Maître de conférences, Télécom Paris (LTCI)	Co-encadrant de thèse
Laurence LIKFORMAN SULEM Maître de conférences-HDR, Telecom Paris (LTCI)	Co-encadrant de thèse

Remerciements

J'ai parcouru un long chemin, j'ai vu beaucoup de paysages, et puis je suis venu ici. J'ai rencontré beaucoup de compagnon, j'ai reçu beaucoup d'aide, et puis j'ai beaucoup grandi. J'aime beaucoup le paysage le long du chemin, et je suis reconnaissant pour la compagnie.

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon recherche et qui m'ont aidée lors de la rédaction de cette thèse. Je tiens également à remercier ces moments émouvants et amis qui m'ont encouragé à aller plus loin.

Je voudrais dans un premier temps remercier, mon directeur de thèse Mme. Lirida Naviner De Barros, professeure à Télécom Paris, pour sa patience, sa disponibilité et surtout ses judicieux conseils dans la recherche scientifique. Ses encouragements et ses expériences partagés me donnent de la force, qui me fait marcher courageusement quelles que soient les difficultés que je rencontre.

Je remercie également le co-encadrant de ma thèse, M. Sumanta Chaudhuri, maître de conférences à Télécom Paris. Il a m'accompagné pendant tout mon étude doctorale, même-si pendant le confinement, nous avons gardé une communication fréquente. Cette communication est non seulement fournie beaucoup d'idées nouvelles pour mon travail, mais m'ont également incité à garder le rythme et l'efficace de travail pendant la travaille à distance. Il ne fait aucun doute qu'avec son aide, de nombreuses idées intéressantes peuvent être implémenté et de nombreux travaux précieux peuvent être réalisés.

Un grand merci également à l'autre co-encadrant de ma thèse, Mme. Laurence Likforman-Sulem, maître de conférences-HDR à Télécom Paris. Elle est érudite et sérieuse. Quand je m'arrêts en raison des difficultés, elle peut toujours trouver de nouvelles solutions avec sa richesse de connaissances. Et quand j'avance mon travail en étourdi, elle peut aussi me rappeler prudemment certains détails pour éviter les erreurs.

Merci à Patrick Girard, Roselyne Chotin, Jacques-Olivier Klein, Fabrice Monteiro et à Attilio Fiandrotti pour leur participation à ma soutenance de thèse, leur relecture scrupuleuse du manuscrit et leurs suggestions toujours avisées.

Je n'oublierai pas les amis rencontrés pendant mes études. Jean-Luc, Tarik et tous les autres du groupe SSH, vous êtes les professeurs qui m'ont amené dans le monde informatique et aussi les amis qui m'ont accompagné sur la route de la recherche. Maciel, Elaine, Etienne, Sarah et Soumaya, nous avons des cultures différentes, mais nous pouvons avancer la main dans la main : quel honneur de vous rencontrer et connaitre ! Surtout merci à Elaine et Maciel, ils sont toujours les premiers lecteurs de plusieurs de mes articles et idées. Son regard de néophyte sur le thème m'a été d'une grande aide pour préciser et affiner mon propos. Hao, Yan, Jie, Yunzhi, Zejiang, Wangfan et Jiali les compagnons chinois qui se battaient ensemble à Télécom Paris, nous avons surmonté une difficulté après l'autre ensemble, et le paysage sur cette route est plus pittoresque à cause de vous. Zefeng, nous avons réalisé de nombreux projets intéressants ensemble, je suis très reconnaissant non seulement de l'opportunité et de la plate-forme que vous m'offrez, aussi de l'encouragements et de la confiance depuis que nous nous sommes rencontrés.

Je suis reconnaissant que ma famille sera toujours mon soutien. Je tiens à remercier mes parents. Viens d'une petite ville de Chine, mais je peux étudier et vivre à Paris, l'un des centres de civilisation dans le monde, grâce à le socle solide de mes parents. Même si j'ai été à des milliers de kilomètres de ma ville natale et que je ne puisse pas accompagner mes parents, ils comprennent toujours ma décision. Je tiens également à remercier ma femme. Grâce à ses encouragements, je peux décider de poursuivre ce doctorat. En raison de sa compagnie, je ne serai pas déprimé et abandonnerai lorsque je rencontrerai des difficultés. Sa présence et ses encouragements sont pour moi les piliers fondateurs de ce que je suis et de ce que je fais.

Ma vie de recherche est comme un voyage, qui est long et âpreté. Mais le paysage du voyage est magnifique, et je suis très être honoré de le profiter avec vous.

Abstract

Convolutional Neural Networks (CNNs) have been extensively used in many fields such as image recognition, video processing, and natural language processing. However, CNNs are still computational-intensive and resource-consuming. They are often constrained by the limit performance and memory when deployed on embedded systems. This PhD research project aims at proposing CNNs which are more suitable for embedded systems with low computing resources and memory requirements.

We begin by integrating some of the more recent machine learning techniques (such as dropout, batch normalization and different activation functions) into convolutional neural networks and review the state-of-the-art compression and acceleration methods of CNNs. After this literature review, we propose three methods to accelerate the operation of neural networks: Selective Binarization, Quad-Approx Network and MinConvNets.

Selective Binarization combines layers with different precisions in CNNs to achieve an acceptable speed and accuracy. As well an FPGA based hardware accelerator is proposed for these optimized structures. We train the tiny-YOLO CNN with a drone object detection dataset (DAC-SDC), and it is possible to achieve 1.68x improvement in runtime performance incurring a tolerable 8.99% loss in precision measured by IOU (Intersection over Union).

With the proposed signed PArameterized Clipping acTivation Function (signed PACT) , the CNNs are quantized into 3 bits, and then a loss-less network is established by using approximate multiplier, which is named Quad-Approx Network. An approximate multiplier for 3-bit multiplication also implemented in FPGA is proposed, which can achieve a 1.2x speedup and 5.3x compression when applied to Quad-Approx Networks. In addition to acceleration, what is more valuable is that Quad-Approx shows that CNNs are certain fault tolerance systems, which leads us to propose the MinConvNets.

MinConvNet is a set of multiplication-less CNNs whose multiplications are replaced by approximate operations. MinConvNet can achieve negligible loss of prediction compared to exact image classification networks through transfer learning, meanwhile the multiplication which is more resource consuming to implement is replaced by easier implemented operations. On the one hand, MinConvNets have abandoned the traditional compression idea and proposed a new direction for acceleration of CNNs, that is to say, the approximate method is directly applied on the operators instead of on operands as classic methods. On the other hand, with the proposed criterion in this work, may be evaluate a compression/acceleration method without validation on image-sets in the future, which may greatly save time.

Human is ushering the era of the artificial intelligence. In the meantime, the Internet of Things (IoT) makes our lives more convenient. The works in this thesis have proposed the optimization methods in the aspects of structure, operands and operators, so as to solve one of the important challenges, that is, how to make the resource-consuming CNNs easier deployed on the resource limited platforms. These works bring more complex intelligent algorithms into the edge devices and helps us to create the era of Artificial intelligent Internet of Things (AIoT).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The unprecedented success of machine learning has ushered us into an era of artificial intelligence. *Neural networks*, one of the best algorithms in machine learning, it has been used to solve various problems, such as computer vision and speech recognition. These problems are difficult to solve by traditional rule-based programming. However, there are still many challenges for Neural Networks. The work in this thesis proposes the solutions for the one of the important challenges, that is, how to make the resource consuming CNNs easier deployed on the resource limited platforms.

1.1 Context

Although neural networks exist since the 1940s [4], the significant breakthrough dates back to 2012 when Alexnet [5] surpassed the human programmed approaches for image classification. The major enabling factors behind this success are supposed to be the appearance of large datasets in the cloud and the availability of enormous computing power (e.g GPUs). Since then, there has been a steady growth of customized neural network accelerators research and several architectures have been proposed [6, 7, 8, 9], and more recently TPU (Tensor Processing Units) [10] have been commercialized by Google.

Nowadays, the powerful cloud cluster and its huge computing power have allowed machine learning algorithms to be widely used in many fields such as data mining, securities market analysis, and medical diagnosis. However, with the development of embedded platform such as the Internet of Things, drones, smart cities, and wearable devices, CNNs have more and more frequent been attempted to deploy on embedded systems and mobile terminals. These applications often have real-time constraints and a very tight power budget. Due to the limited computing resource of embedded systems, the CNNs are always pre-trained offline and then implemented in embedded systems for the inference stage. However, most of the neural network, such as Convolutional Neural Network (CNN) is still computationally intensive and resource-consuming in the inference stage due to the convolution calculation requiring a great number of multiply-accumulate (MAC) operations.

For example, AlexNet [5] requires about $1.5G$ multiplication operation in convolutional layers to process a 224×224 image. For more complex tasks, such as object detection, the number of multiplication operations is higher. Multiplication is always more difficult to implement, or time-consuming in computing [11], therefore, several methods are proposed to speed up or compress the network.

Quantization is a class of methods that make use of low-precision arithmetic [12]. [8] proposes half-precision floating point, [10] uses 8-bit quantized weights, and [13, 14] propose the use of binarized weights and feature maps. Compared with the original neural network encoded in 32-bit, the quantized network is smaller, and fewer-bit multiplication may cost fewer computing resources for some special platform.

Pruning networks is another widely discussed method to compress and speed up the neural network. The main idea of pruning is to cut out the redundant neurons in neural networks so that reduces the number of operations. Filter pruning methods that rank filters then remove the less important filters are proposed in [15, 16, 17]. In weights pruning methods [18, 19, 20], the less important weights in each filter are dropped off and set to 0, so that sparse networks can be built and can be accelerated by sparse matrix calculator.

There are some other compression and acceleration techniques, which we will discuss in detail in Chapter 4.

1.2 Problematic

Although lots of excellent works have made great progress in this field, there are still many shortcomings for the compression and acceleration of convolutional neural networks. For example, although the binary networks work well for image classification, they tend to lose precision with more complex tasks, such as object detection which needs to localize the objects in images.

However, more complex tasks are gradually being deployed in embedded systems, such as drones that need to track the object, or auto-driving cars that need to locate pedestrians. Since embedded systems are always with limited resources, compression, and acceleration for these more complex tasks have become an attractive subject. To solve the deficiencies of the current optimization of the neural networks and make it more widely used, the following question have been proposed:

- Is it possible to optimize the existing methods or propose new directions to accelerate and compress CNN, so that CNN requires less resources but can handle more complex tasks?

The works in this thesis attempt to answer this question.

Although neural networks can have several different variations such as multi-layer perceptrons, Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNNs), Long Short Term Memories (LSTM) etc. we limit our scope to accelerating Convolutional Neural Networks (CNNs).

1.3 Contributions

In order to compress and/or accelerate the CNNs, three architecture are proposed: Selective Binarization network, Quad-Approx network, and MinConvNets.

Selective Binarization combines layers of different precisions in CNNs to achieve an acceptable speed and accuracy. The contributions with selective binarization are as follows:

- CNN structure that consists of layers with multiple precisions is introduced. For each convolutional layer, there are three different precision levels to choose from: half-precision floating for feature maps and weights; half for feature maps and binary weights; and binary feature maps and weights. In the binary layer, multiply-accumulate (MAC) operations are converted to addition or/and subtraction or XNOR-popcount logic operations. As well, a new calculator instead of a traditional MAC operator could be used to speed up the calculation.

- In this work, an architecture that is compatible with these three different precisions is proposed. Different precisions can be chosen flexibly for each layer according to different scenarios, then apply the appropriate implementation to accelerate the calculation. To our knowledge, this is the first architecture that proposes such a choice.

- It has been proposed a method of architecture exploration to find the optimum use of binary layers, which is called selective binarization. The aim is to use selective binarization to increase performance and decrease power consumption, within a tolerable precision loss. Then an adapted end-to-end streaming architecture for a deep learning accelerator is proposed.

Quad-Approx networks are structures with the few-bit quantization and approximate operator. The contributions are:

- The PArameterized Clipping acTivation Function (PACT) [21] method is applied to our training process, and propose a Signed PACT method for quantization of fewer-bit signed values.

- Based on this method we show that signed 3-bit quantization is good enough for object detection tasks. It has already been shown by others [14, 22] that binarized networks (1 bit) are good enough for classification tasks.

- An approximate multiplier circuit for 3-bit multiplication is proposed, which can achieve a 1.2x speedup.

- A training method where the multiplier approximations are back annotated to the training process is proposed, which leads to no loss in overall precision.

Finally, the MinConvNets are proposed. But different from the classic methods introduced in section 1.1 which accelerate multiplication or reduce the number of multiplication operations, multiplication is abandoned directly in this work. Multiplication in CNNs is replaced with lighter operations. The main contributions of this work are:

- A set of criteria to measure the degree of approximation between two operations is proposed. Based on these criteria, an approximate operation to multiplication is proposed for error-tolerant systems.

- On top of the proposed approximate operation, an approximate convolutional layer without multiplication is explored to replace the traditional convolutional layer. Then we build and train the CNN by using an approximate convolutional layer, named MinConvNets. The benchmark is applied to MinConvNets to show that lossless accuracy can be achieved in image classification tasks.

- This work shows that in addition to multiplication, other lighter operations can still effectively extract image features. This provides a new direction for compressing or accelerating CNNs in future works.

Selective Binarization is a structural optimization that combines and improves existing compression methods. Quad-Approx Networks quantize the operands in CNNs to solve some difficulties of the existing optimization methods and based on few-bit quantization the approximate operators are applied. The work of MinConvNets uses approximate operators directly and provides a new direction for compressing and accelerating CNNs. These three works answer the question raised in section 1.2 by improving the structure, operands, and operators of CNNs.

These works reduce the resources required by CNNs and makes them easier to be deployed in systems with limited resources. On the one hand, this enriches the usage scenarios of CNN, and on the other hand, it makes the embedded system more intelligent. We have already ushered in the era of artificial intelligence, as well as the era of the Internet of Things. This work can integrate these two fields and help us create the era of Artificial intelligent Internet of Things.

1.4 Organization

The thesis is organized as follows:

- The history of machine learning, and the technologies of convolutional neural networks is presented in Chapter 2.

- Chapter 3 describes the difficulties of CNNs and determines the task and the platform to solve the difficulty.

- The state-of-the-art works are introduced in Chapter 4, while some of these methods are implemented in Chapter 5 to measure the performance.

- Based on this, in Chapters 6, Chapter 7, and Chapter 8, Selective Binarization, Quad-Approx Networks, and MinConvNets are proposed respectively to speed up the CNNs.

- In Chapter 9 the conclusion and the future work are discussed.

Chapter 2

Machine Learning and Convolutional Neural Networks

Machine learning and convolutional neural networks have been proposed and discussed since the last century. After decades of development, the convolutional neural network is already one of the powerful algorithms in the field of machine learning now. Machine learning, especially convolutional neural networks is introduced in this chapter.

First, the history of the development of the neural networks and the basic technologies of convolutional neural networks are introduced. Next, certain classic and important structures of convolutional neural networks are discussed. Then we discuss and compare the widely used neural network frameworks in this chapter. At the end of this chapter, they are summarized.

2.1 Machine Learning

Human beings have been trying to make machines intelligent, that is, artificial intelligence. In the 1950s, peoples believed that if the machine acquired the ability to reason, the machine would be intelligent. For example, *Logic Theorist*, a computer program written in 1956 by A. Newell et al. [23] [24], was the first program deliberately engineered to perform automated reasoning and is called "the first artificial intelligence program". It would ultimately prove 38 of the first 52 theorems in Whitehead and Russell's *Principia Mathematica* and find new and more elegant proofs for some of them [23].

But even if the machine has reasoning ability, it is far from intelligence due to lack of knowledge. Therefore, in the 1970s, the development of artificial intelligence entered a "knowledge period", that is, to summarize human knowledge and teaching it to machines so that machines can gain intelligence. During this period, a great number of expert systems have emerged and achieved a lot of achievements in many fields.

However, on one hand, people gradually realize that it is a gigantic workload project for people to come to their knowledge and then teach it to the computer. On the other hand, the computers operate in accordance with the rules and summarized knowledge set by humans, which means they can never surpass their creators. To solve this problem, a number of scholars thought that the machine could learn knowledge on its own. The concept and method of machine learning have been proposed. Figure 2.1 shows the timeline for the development of artificial intelligence as well as machine learning.

Machine learning is now a huge family involving many algorithms, tasks, and theories. It includes but is not limited to algorithms such as decision trees, neural networks, support vector machines (SVM), and Bayes classifiers. These algorithms attempt to discover the hidden rules from a large amount of data and use them for prediction or classification. More specifically, machine learning can be seen as algorithms to find a mapping function between the inputs (sample data) and the outputs (the desired class), but this function is too complicated to express in a convenient way.

2.1.1 Building a Machine Learning System

The process of obtaining this function is called training a machine learning system. Generally speaking, there are three steps to training a neural network:

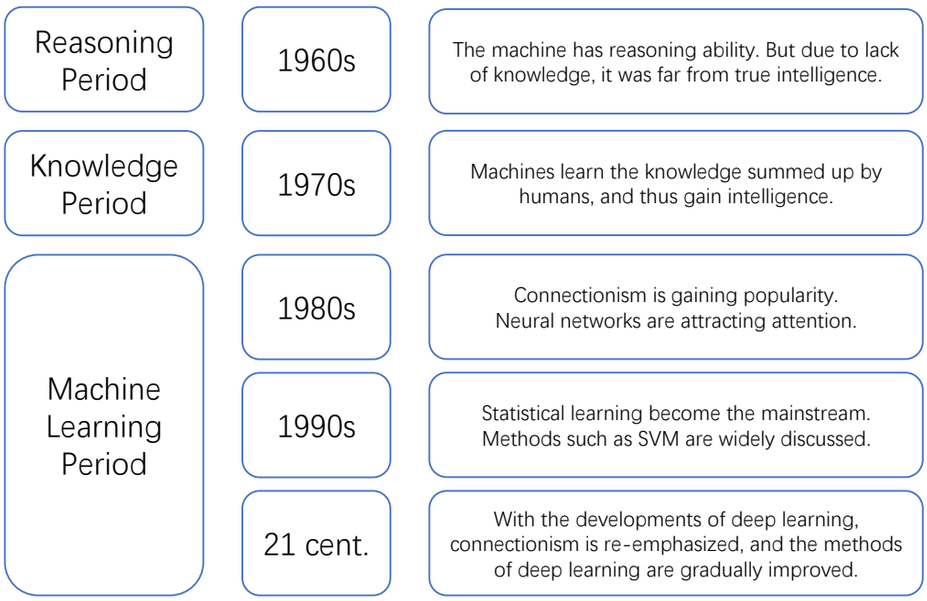


Figure 2.1: The artificial intelligence timeline

Choose a suitable model

The model includes a function model and a dataset for training. Choosing a function model usually depends on the actual problem, and the appropriate model needs to be selected for different tasks. For example, both decision tree and SVM can resolve binary classification tasks, while convolutional neural networks (CNN) usually are more powerful to process images.

Most machine learning algorithms have some parameters to be configured. The performance caused by different parameters is often significantly different. Even if we determine the model used, the parameters of the model need to be constantly adjusted. This adjustment is called parameter tuning.

Another important choice is the training dataset, which should contain a large variety of samples. For example, when we want to train a network that can distinguish cat and dog pictures, the training sample set should contain as many different images of cats and dogs as possible. If the training set contains only the images with black dogs and white cats, then the system may learn to recognize colors instead of animals. Of cause, if the dataset contains only cats and dogs, the trained network will not recognize rabbits and horses.

It should be pointed out that the goal of machine learning is to make the learned function well applicable for the "new samples", not just perform well on training samples. The ability of the learned function to apply to new samples is called generalization. To gain the ability of generalization, the system should try to learn the general characteristics of samples. If a machine learning system relies on the specific characteristics of samples, the generalization ability will be reduced. That is called over-fitting.

Measure the performance of the system

Performance measures can reflect the quality of a model. For a given sample set $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ...(\boldsymbol{x}_m, y_m)\}$, where y_i is the ground truth of sample $\boldsymbol{x}_i$, when measuring the performance of system f, it is necessary to compare the prediction result $f(\boldsymbol{x})$ and ground truth y. For example, mean squared error $E(f, D)$ is the most commonly used performance metric in regression task:

$$E(f, D) = \frac{1}{m}\sum_{i=1}^{m}(f(\boldsymbol{x}_i) - y_i)^2 \tag{2.1}$$

Depending on the task, the measure criteria are different. For example, YOLO introduced in section 2.3.6 can classify images and locate objects in the image. Therefore, measurement criteria for YOLO should evaluate not only the correctness of classification but the accuracy of the location of objects. Some measure criteria for classification and object detection tasks are presented in section 3.2.3.

Find the "best" function

When we build a machine learning system based on the measure criterion, we prefer to pick out the best function to solve our problem. But in fact, it is not easy. For example, some large convolutional neural networks may be nonlinear systems containing billions of parameters, and it is impossible to try these parameters one by one and get the optimal solution. But we can build a loss function of the system, that can be expressed by the performance metric. Then we adjust the parameters through dedicated algorithms to find the minimum value of the loss function. In this way, we transform the problem of selecting parameters into a numerical optimization problem. The next step is to use some optimization methods to find the minimum value of the loss function. For example, the least-squares method can be applied to optimize regression systems, while convolutional neural networks usually use gradient descent algorithms to minimize their loss function and optimize their huge number of parameters.

Based on these steps, a machine learning system can be trained for prediction. For different tasks and methods, the implementations are also different.

2.1.2 Common Concepts and Methods

Next, we will introduce some concepts and methods that are often mentioned when building a machine learning system. These may also be used in later chapters.

Regression, Classification, Structured Learning

These words express the different targets of machine learning systems.

The output of the regression model is the value of a quantitative variable. This model is usually used to predict unknown cases based on well-known cases.

The classification model includes a binary classification and a multiclass classification. The binary classification is the task of classifying the elements of a given set into two groups, such as positive/negative. And multiclass is the problem of classifying instances into one of three or more classes. Based on the different problems and results, the output and measure criteria of binary classification are different. Table 2.1 shows the result of binary classification. In this table, the column ratios are True Positive Rate $TPR = TP/(TP+FN)$, aka sensitivity or recall, with complement the False Negative Rate $FNR = FN/(TP+FN)$; and True Negative Rate $TNR = TN/(TN+FP)$, aka specificity or SPC, with complement False Positive Rate $FPR = FP/(TN+FP)$. The row ratios are Positive Predictive Value $PPV = TP/(TP+FP)$, aka precision, with complement the False Discovery Rate $FDR = FP/(TP+FP)$; and Negative Predictive Value $NPV = TN/(TN+FN)$, with complement the False Omission Rate $FOR = FN/(TN+FN)$. In diagnostic testing, the main ratios used are the true column ratios, i.e., True Positive Rate and True Negative Rate, where they are known as sensitivity and specificity. In informational retrieval, the key ratios are the true positive ratios (row and column), i.e., Positive Predictive Value and True Positive Rate, where they are known as precision and recall.

For multi-classification problems, the measure criteria often change based on the problem. For example, cross-entropy is often used in image classification, and YOLO described in section 2.3.6 uses the L_2 norm of confidence to measure the performance of classification.

As to the structured learning, its output is no longer a fixed-length value. For example, the output of the semantic analysis of the picture is the text description of the picture.

These three categories are not exclusive. For instance, the problem of object detection in the image needs to find the location of the object in the image and classify them. We can transform him into the regression analysis to determine the location and classification information to determine the type of object.

Supervised Learning, Unsupervised Learning, Reinforcement learning

They are three basic machine learning paradigms.

Supervised learning is the machine learning task of learning a function that maps an input to an output based on the example of input-output pairs. [25] It infers a function from labeled training data consisting of a set of training examples [26]. In supervised learning,

Table 2.1: The result of binary classification

	Condition Positive (CP)	Condition Negative (CN)
Test Outcome Positive (OP)	True Positive(TP)	False Positive(FP)
Test Outcome Negative (ON)	False Negative(FN)	True Negative(TN)

each sample is a pair consisting of an input object (typically a vector) and the desired output value (also called the supervisory signal).

A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new samples. Unsupervised learning is a type of machine learning that looks for previously undetected patterns in a dataset with no pre-existing labels and with a minimum of human supervision. In contrast to supervised learning that usually makes use of human-labeled data, unsupervised learning, also known as self organization allows for modeling of probability densities over inputs. [27]

Reinforcement learning focuses on the concept of how software agents ought to take actions in an environment to maximize the notion of cumulative reward. Reinforcement learning differs from supervised learning in not needing labeled input/output pairs to be presented, and there is no need to explicitly correct sub-optimal actions. Instead, the focus is on finding a balance between exploration of uncharted territory and the exploitation of current knowledge. [28]

Hyperparameter

In machine learning, a hyperparameter is a parameter whose value is used to control the learning process. By contrast, the values of other parameters (typically node weights) are derived via training.

Hyperparameters can be divided into model hyperparameters and algorithm hyperparameters. Model parameters cannot be inferred while fitting the machine to the training set because they refer to the model selection task. And algorithm hyperparameters in principle have no influence on the performance of the model but affect the speed and quality of the learning process. An example of a model hyperparameter is the topology and size of a neural network. Examples of algorithm hyperparameters are learning rate and mini-batch size, introduced in section 2.2.

Transfer Learning

Transfer Learning is a research problem that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem [29]. For example, the knowledge gained while learning to recognize cars could apply when trying to recognize trucks. For training a YOLO network in section 2.3.6, convolutional weights pre-trained on ImageNet [30] are used, which uses the transfer learning method. From the perspective of practical standpoint, reusing or transferring information from previously learned tasks for the learning of new tasks has the potential to significantly improve the sample efficiency of a reinforcement learning agent [29].

Deep Learning

Deep learning is a class of machine learning algorithms, which uses multi-layer to progressively extract higher-level features from the raw input [31]. For instance, in image processing, lower layers may identify edges, while higher layers may identify the concepts relevant to a human such as digits or letters or faces.

Data Augmentation

Data augmentation in data analysis are techniques used to increase data volume by adding slightly modified copies of already existing data or newly created synthetic data from existing data. This method is usually used when the training dataset is insufficient. It also helps to reduce over-fitting [32]. Geometric transformations, flipping, color modification, cropping,

rotation, noise injection and random erasing are used to augment the number of images in deep learning.

2.2 Convolutional Neural Network

With the development of computer performance, how to make machines understand images has gradually attracted people's attention. Initially, pattern recognition technology was applied. For instance, templates or patterns could be used describing objects' structure. There has an example, we can distinguish handwritten digits 3 and 8 by detecting whether there is a closed circle. This is artificial intelligence in reasoning and knowledge mode.

Although the pattern recognition has achieved certain results, as mentioned above, manual extracting knowledge is a heavy workload. With the development of machine learning, people prefer to feed images to machines so that machines can analyze and recognize images automatically.

Convolutional Neural Network(CNN), also called ConvNets, is one of the excellent image processing models. Now, CNNs have been widely used in image classification, object detection, video tracking and other fields. In this section, we will give a brief introduction to CNNs.

2.2.1 Computational Primitives

In their most general form, neural networks can be interpreted as computational graphs composed of primitive operations. The computational graphs allow for a rich set of primitive operations and we will describe below the most used and most successful operations.

Matrix-vector Multiplication and Fully-connected Layer

Matrix-vector Multiplication is probably the most widely used primitive for deep learning calculations. It is a linear operation (without nonlinear effects) and is used as an integral part of the most successful neural network architectures.

Most commonly, the vector $x \in \mathbf{R}^n$ represents information previously processed by the neural network and/or unprocessed (input) information. $W \in \mathbf{R}^{m \times n}$ is a trainable matrix, which means its entries are modifiable. The result of this operation is the vector $y \in \mathbf{R}^m$:

$$y = W \times x \tag{2.2}$$

Usually, a vector of biases is added to the multiplication result and the previous equation becomes:

$$y = W \times x + b \tag{2.3}$$

Matrix multiplication is frequently used in CNNs. e.g, Fully-connected layers presented in 2.2.1 in CNNs can be computed with a matrix multiplication followed by a bias offset.

In the context of CNNs, fully connected structures require too many parameters. In CIFAR-10, seen in section 3.2.2, for a color images (3 channels for red blue, and green) with size 32x32, a single fully-connected neuron in the first layer of a Neural Network would already have 32*32*3 = 3072 weights. For a color image with size 200x200, 120K weights are needed. Moreover, we almost certainly would like to have several such neurons, so the parameters would add up quickly.

Spatial Convolution and Convolutional Layer

Convolution is a mathematical operation of two functions (f and g) that produces a third function represents how the shape of one function is modified by the another. It is defined as the integral of two functions where the one is reversed and shifted by another one. As such, it is a particular kind of integral transform:

$$(f \otimes g)(t) = \int_{-\infty}^{+\infty} f(\tau) \cdot g(t - \tau) d\tau \tag{2.4}$$

The pixels of a given input image can be considered as a discrete ternary function $I(w, h, c)$, where w, h, c is the spatial coordinates of the pixel. Then a discrete spatial convolution applied to this ternary function is calculated as:

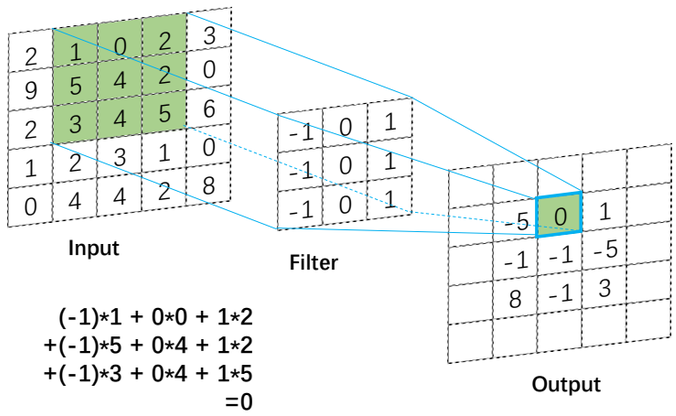


Figure 2.2: An example of convolution for one filter with size=3x3x1

$$g[w,h,c] \otimes I[w,h,c] = \sum_{n_1=1}^{width} \sum_{n_2=1}^{height} \sum_{n_3=1}^{channel} g[n_1,n_2,n_3] \cdot I[w-n_1, h-n_2, c-n_3] \tag{2.5}$$

where $width, height, channel$ describe the size of images.

Traditionally, function g for convolution has the same number of channels to images but the smaller width and height. The size of function g is noted as $f_w \times f_h \times channel$. Usually for convenience, we will associate the function g with the weight filter matrix $W \in \mathbf{R}^{f_w \times f_h \times channel}$, and then the convolution for the image is expressed as:

$$W[w,h,c] \otimes I[w,h,c] = \sum_{l_1=1}^{f_w} \sum_{l_2=1}^{f_w} \sum_{l_3=1}^{channel} W[l_1,l_2,l_3] \cdot I[w+l_1, h+l_2, c+l_3] \tag{2.6}$$

Here, it is observed that the convolution of a pixel is to multiply the points around this pixel by weights then accumulate them. The calculation in Figure 2.2 shows an example that we calculate the convolution of one pixel with a filter with size 3x3x1. For the convolutions of the whole image, the filter is moved to each pixel location. It can be considered as scanning an image by the filter. The convolution of the image is always slide in 2 dimension (width and height), without considering the direction of depth(channel), so the c in Equation 2.6 is usually fixed to 0 and the output of the convolution is a 2 dimension tensor.

The convolution calculation does not look at a pixel isolated, but considers the pixel and its neighborhood as an integrated structure. It is observed that for fixed weights, some pixels can obtain bigger output values by the convolution operation, in other words, these pixels are receptive to the filter.

The output of the convolution is a tensor recording the positions of receptive and non-receptive area, called feature-map. Each filter can extract one feature, and when we need to extract multiple features, more filters each produce a separate 2-dimensional feature-map. We will stack these feature-maps along the depth dimension and produce the output volume. The number of filters we would like to use is called depth. We call the number of filters that we would like to use "depth". It is a hyper-parameter.

The tiny filter will have a limited area of the perception field, therefore multiple layers of filtering can be used, videlicet, send the results of the current filter to the next layer to increase the area of the perception field. Intuitively, the filter will feel several types of visual features such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network.

At this point in time, we can build a convolutional layer. Different from the fully connected layers, the number of weights of the convolutional layer is not related to the size of the image, so the size of the weights will be reduced.

Generally, apart from convolution, the convolutional layers (noted as CONV) usually also use bias calculation, as follows:

$$CONV(X)[w,h,c] = W[w,h,c] \otimes X[w,h,c] + bias \tag{2.7}$$

where $bias \in \mathbf{R}$ is a trainable parameter for each filter.

Non-Linear Function

As mentioned above, we can use deep neural networks to make the types of image recognition more diversified. But if we only use linear layers, such as fully-connected layers and convolutional layers, then we can always find a single-layer linear transformation to replace the deep neural network. Intuitively, no matter how many linear operations are composed, the entire system is no more powerful than a simple linear regression. As such, there is no way to benefit from deep neural networks. Therefore, the non-linear function needs to be inserted, so the deep neural network can establish non-linear relation between input and output.

In neural networks, non-linearity is introduced using the concept of an activation function, which is applied element-wise to the input. It can better perceive and express the image.

Historically, the most popular activation function used to be the *sigmoid* function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.8}$$

Another activation function with a long history is the *tanh* function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.9}$$

Obviously, there are 4 exponential operations in *tanh*. To simplify the operation, $HardTanh$, a function approximated to *tanh* is also used:

$$HardTanh(x) = \begin{cases} 1 & \text{if} \quad x > 1 \\ x & \text{if} \quad -1 < x \leq 1 \\ -1 & \text{if} \quad x \leq -1 \end{cases} \tag{2.10}$$

Currently, one of the most successful activation functions is the Rectified Linear Unit (ReLU) [5]

$$ReLU(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.11}$$

Compare to the *sigmoid* and *tanh*, there is no longer any exponential calculations, which is easier for hardware implementation with limited resources.

Leaky rectified linear units (LReLUs or Leaky) [33] have been found to either match or surpass ReLUs in performance by some authors:

$$Leaky(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.12}$$

α is the scaling factor and is fixed. Parametrized Rectified Linear Units (PReLU) [34] is another rectified activation function, with the same equation as for LReLU. The difference, though, is that α is a trainable parameter (through gradient-based optimization).

We introduced commonly used non-linear functions, which usually follow the convolutional layers or fully connected layers, making these calculations non-linear. Non-linearity makes neural networks, especially deep neural networks, more expressive.

Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Convolutional layers in a CNN architecture. Its function is to gradually reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control over-fitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially. The depth dimension remains unchanged.

The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 down-samples every depth slice in the input by 2 along both width and height, discarding 75% of the activations, and then every Max operation would in this case be taking the maximum value over 4 numbers (little 2x2 region in some depth slice). Given input feature map i, the output feature map o is given by:

$$o[m, n] = Max(i[2 * m, 2 * n], i[2 * m + 1, 2 * n], i[2 * m, 2 * n + 1], i[2 * m + 1, 2 * n + 1]) \tag{2.13}$$

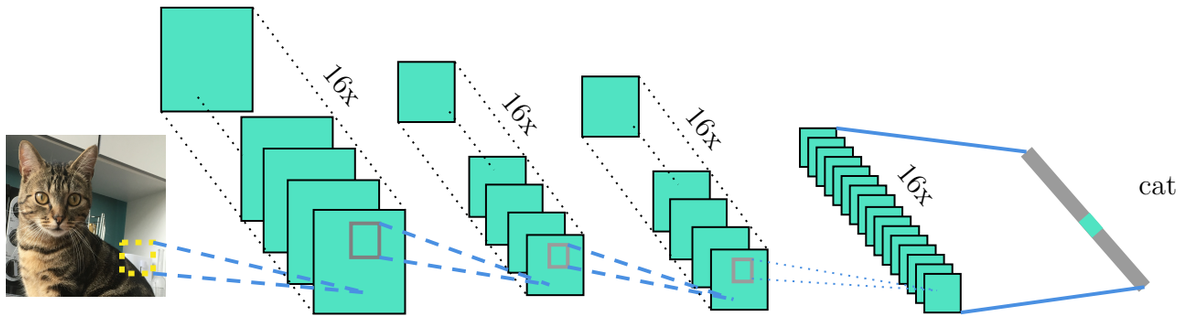


Figure 2.3: A typical CNN architecture for classification

Besides, the pooling units can also perform other functions, such as average operation or even L2-norm. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

In addition to reducing the size, another notable usage of the pooling layer is to set the stride as 1. The main motivation is to provide some robustness to the small images.

2.2.2 Building Convolutional Neural Networks

As we described above, a simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. We use three main types of layers to build CNN architectures: Convolutional Layer(CONV) followed by ReLU, Pooling Layer(POOL), and Fully-Connected Layer(FC).

Figure 2.3 shows an overview of the example CNN architecture for classification. More details are described as follows:

- INPUT $[32 \times 32 \times 3]$ will hold the raw pixel values of the image, in this case, an image of width 32, height 32, and with three color channels R, G, B.
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x16] if we decided to use 16 filters.
- RELU layer will apply an element-wise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the volume unchanged.
- POOL layer will perform a down-sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x16].
- We repeat the CONV-RELU-POOL as steps 2 to 3, but the input is the feature maps of the last layer, as volume [16x16x16]. The output result is in volume such as [8x8x32] if we use 32 filters in this CONV layer.
- FC layer will compute the class scores, resulting in a volume of size [1x1x10], where each of the 10 numbers corresponds to a class score, such as among the 10 categories of Cifar10, an image set presented in section 3.2.2. Just like an ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the neurons in the previous layer.

In this way, CNNs transform the input images layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and others do not. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function without parameters. The parameters in the CONV/FC layers will be trained so that the class scores that the CNN computes are consistent with the labels in the training set for each image.

2.2.3 Building the Loss Function

In the previous sections, we discussed various neural network architectures. Generally, all of these architectures can be represented as parameterized functions $f(x, \theta)$, where by x we denote the inputs and by θ we denote the trainable parameters. We can assume that, initially, the parameters θ are randomly initialized, or are initialized in a particular way, for example, with probability distributions.

Next, we will first introduce loss function L, which is used to quantify the quality of the set of trainable parameters θ. The loss function is minimized by the optimization process, which searches for the corresponding parameters θ. We then briefly discuss optimization in the next section, the process of finding the parameters θ which minimize the loss function L.

As described in section 2.1, performance measures can be used to evaluate the performance of the machine learning system. For image classification, precision and recall introduced in section 3.2.3 may be used to evaluate the performance. However, some common encountered performance measures are often not differentiable, a property that is strongly desirable. For this reason, we need to build a differentiable loss function and optimize the machine learning system according to this loss function.

Taking the architecture in section 2.2.2 for Cifar10 image set as an example, we build the loss function L.

For a given image set such as Cifar10 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), ...(\boldsymbol{x}_m, y_m)\}$, where $\boldsymbol{x}_i$ is the matrix that represents the pixel of i^{th} image in this image set, and y_i marked the category number of this image, we build a CNN architecture:

$$Y = func(\boldsymbol{x}, \theta) \tag{2.14}$$

where the output is a matrix $Y \in R^{10}$. The j^{th} element in the matrix Y represents the predicted possibility that the image belongs to category j, that means $Y(j) = P^{predict}(y_i = j|\boldsymbol{x}_i)$. For a classifier with high accuracy, for example, if $y_i = 4$, then when we input the image $\boldsymbol{x}_i$, in the output of the classifier, the value of the fourth element should be significantly larger.

The loss function called cross-entropy, measuring the distance between Y and y_i is build:

$$L(y_i, Y) = -\sum_{j=1}^{10} (\mathbf{1}_{y_i=j}) log(Y(j)) \tag{2.15}$$

More generally, cross-entropy is defined as following:

$$H(\boldsymbol{y}, p) = -\sum_{j} y_j log(p_j) \tag{2.16}$$

where y is the ground-truth as a one-hot encoding vector including a one at the class position, and zeros elsewhere, p represents the predicted possibility. It indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.

This loss function can be considered as a function of a given data (x, y) and a set of parameters θ. It can be minimized through gradient-based optimization, by adjusting the parameters θ, so that the accuracy of the classifier becomes higher for a given dataset.

2.2.4 Training Convolutional Neural Network

Backward propagation (also called backpropagation) method in conjunction with the optimization algorithm is an iterative method for supervised learning such as CNNs. It starts with an initial set of parameters θ_0, which is iteratively refined so that the loss function is gradually minimized. This is probably the most widely used method for optimizing machine learning models.

The basic process flow of the backward propagation method is shown in Figure 2.4, in which the optimization algorithm is the gradient descent. Actually, the forward propagation phase is the prediction process of the model. When the predicted results of the convolutional neural network do not match the expectation, the backward propagation process is performed to train the network.

Gradient descent method is an iterative optimization algorithm for finding the minimum value of a function. It performs two steps iteratively:

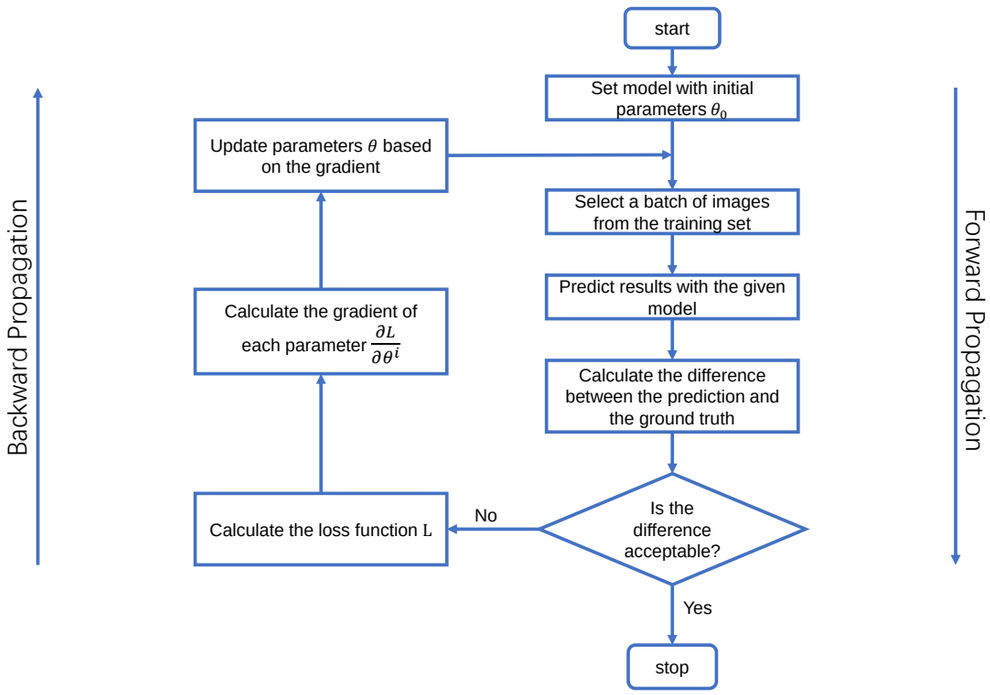


Figure 2.4: The gradient descent training methods for CNNs

1. Compute the slope (gradient) that is the first order derivative of the function at the current point.

   For multi-layer networks, the output of the previous layer is used as the input of the latter layer. In other words, the multi-layer convolution can be regarded as a nested compound function. For nested functions, the direct derivation is complicated to calculate. Hence the chain rule to compute the derivative of a composite function is usually used to compute the gradient of the parameters in the deep networks.

2. Move a small step from the current point to the direction of the gradient descent.

   To update the parameters θ using gradient descent, it must choose a learning rate η. The naive process of updating the parameter θ^i when only one training sample passed through the neural network is expressed as:

$$\theta_{t+1}^i = \theta_t^i - \eta \cdot \frac{\partial L}{\partial \theta_t^i} \tag{2.17}$$

   where t is the index of iteration. For the different size of training sample and the different variants of gradient descent, the specific implementation will be different, that is discussed in below.

A widely used metaphor to describe the gradient descent process is that a blindfolded hiker tries to reach the bottom of a hill by feeling the downward slope at the current location. The core intuition behind this approach (instead of e.g., using random search) is that it can be much easier to make a small improvement in the loss function than to come up with the optimal parameters θ in a single step (as would be necessary for random search).

Depending on the number of training set samples used in each iteration, we get different variants of the gradient descent.

- If a single example is selected randomly at every iteration, we obtain Stochastic Gradient Descent (SGD). The details of SGD can be found in the work [35].
- If a fixed number of examples (e.g., 128) are selected at each iteration, we obtain mini-batch gradient descent (the examples are denoted as a mini-batch).
- When all the examples in the training set are used for each iteration, we obtain batch gradient descent.

The trade-off between selecting more or fewer data points at every iteration goes as follows: using more examples allows for more useful gradients for the optimization process, but they also require more computation. Therefore, mini-batch gradient descent can be considered as a compromise between SGD and batch gradient descent. It is the most commonly used variant of gradient descent in deep learning, because it leads to less noisy gradients but not as computationally intensive as batch gradient descent. As well, code that operates on the mini-batch examples can be parallelized so that the computational cost is not much larger than the cost of processing a single example.

Through the backward propagation and gradient descent methods, the parameters can be updated iteratively. And the given network structure with trained parameters can achieve more accurate predictions with smaller loss function for a specific dataset. For different structures, the minimum value of the loss function is also different. An appreciated structure can bring better prediction results. In the next chapter, we will discuss the classic CNNs structure for different tasks.

2.3 Classic Convolutional Neural Networks

In the previous chapter, we have introduced the computational primitives. CNNs can be regarded as computational graphs composed of these primitive operations. So how to build high-performance computational graphs has become a valuable problem, and it is worth studying. In this chapter, we introduce several classic CNN structures, which have achieved excellent performance for their target tasks. Meanwhile, the methods and ideas proposed along with these structures also provide support for more complex CNN models. It can be said that they are milestones in the development of CNNs.

2.3.1 LeNet

Figure 2.5: The structure of LeNet, where "Conv 5x5 s1, @6, tanh" means a convolutional layer has 6 kernels with size 5x5 and stride 1, and followed by tanh as activation function

LeNet is a convolutional neural network structure proposed by Yann LeCun et al. in 1998. In general, LeNet refers to LeNet-5 and is a simple convolutional neural network.

In 1989, Yann LeCun et al. at Bell Labs first applied the backward propagation algorithm to practical applications and believed that the ability to learn network generalization could be greatly enhanced by providing constraints from the task's domain. He combined a convolutional neural network trained by backward propagation algorithms to read handwritten numbers and successfully applied it in identifying handwritten zip code numbers provided by the US Postal Service. This was the prototype of what later came to be called LeNet [36].

In 1990, their paper again described the application of backward propagation networks in handwritten digit recognition. They only performed minimal preprocessing on the data, and the model was carefully designed for this task and it was highly constrained. The input data consisted of images, each containing a number, and the test results on the postal code digital data provided by the US Postal Service showed that the model had an error rate of only 1% and a rejection rate of about 9% [37].

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	x				x	x	x			x	x	x	x		x	x
1	x	x				x	x	x			x	x	x	x		x
2	x	x	x				x	x	x			x		x	x	x
3		x	x	x			x	x	x	x			x		x	x
4			x	x	x			x	x	x	x		x	x		x
5				x	x	x			x	x	x	x		x	x	x

Table 2.2: The sparse connection of 6 inputs to 16 convolution kernels in C3 layer of LeNet-5

Structure

As a representative of the early convolutional neural network, LeNet possesses the basic units of a convolutional neural network, such as the convolutional layer, pooling layer, and full connection layer, laying a foundation for the future development of convolutional neural networks. As shown in Figure 2.5, the LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier. The detail of the structure is shown below:

- Layer C1 is a convolutional layer with 6 convolution kernels of 5×5 and the size of feature mapping is 28×28, which can prevent the information of the input image from falling out of the boundary of the convolution kernel. For the convolutional layers of LeNet, *tanh* function is followed as the non-linear activation function.
- Layer S2 is the subsampling/pooling layer that outputs 6 feature graphs of size 14×14. Each cell in each feature map is connected to 2x2 neighborhoods in the corresponding feature map in C1. The average pooling layer is applied in LeNet structure.
- Layer C3 is a convolution layer with 16 5×5 convolution kernels. In this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer as shown in Table 2.2. The primary reason is to break the symmetry in the network and keeps the number of connections within reasonable bounds. That is why the number of training parameters in these layers is 1516 instead of 2400 and similarly, the number of connections is 151600 instead of 240000.
- Layer S4 is similar to S2, with a size of 2x2 and an output of 16 5x5 feature graphs.
- Layer C5 is a convolution layer with 120 convolution kernels of size 5x5. Each cell is connected to the 5*5 neighborhood on all 16 feature graphs of S4. Here, since the feature graph size of S4 is also 5x5, the output size of C5 is 1*1. Hence S4 and C5 are completely connected. C5 is labeled as a convolutional layer instead of a fully connected layer because if LeNet-5 input becomes larger and its structure remains unchanged, its output size will be greater than 1x1, that is not a fully connected layer.
- F6 layer is fully connected to C5, and 84 feature graphs are output.
- Finally, there is a fully connected softmax output layer with 10 possible values corresponding to the digits from 0 to 9.

The research of LeNet achieved great success and aroused the interest of scholars in the study of neural networks. While the architectures of the best performing neural networks today are not the same as that of LeNet, the network was the starting point for a lot of neural network architectures, and also brought inspiration to the field.

2.3.2 AlexNet

AlexNet is a convolutional neural network designed by Alex Krizhevsky and published with Ilya Sutskever and Krizhevsky's doctoral advisor Geoffrey Hinton [5].

AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge [30] in September 2012. The network achieved a top-5 error of 15.3%, more than 10.8% points lower than that of the runner up. The primary result of the original paper was that the depth of

the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training [5]. In fact, AlexNet is not the first network running on GPU. However, AlexNet is considered one of the most influential papers published in computer vision, having spurred many more papers published employing CNNs and GPUs to accelerate deep learning [38].

AlexNet contains 11 layers, shown as Figure 2.6. There are five convolutional layers, some of them followed by max-pooling layers, and the last three are fully connected layers. More details of the structure of AlexNet are described in [5]. This paper mainly introduces the main innovations of this study:

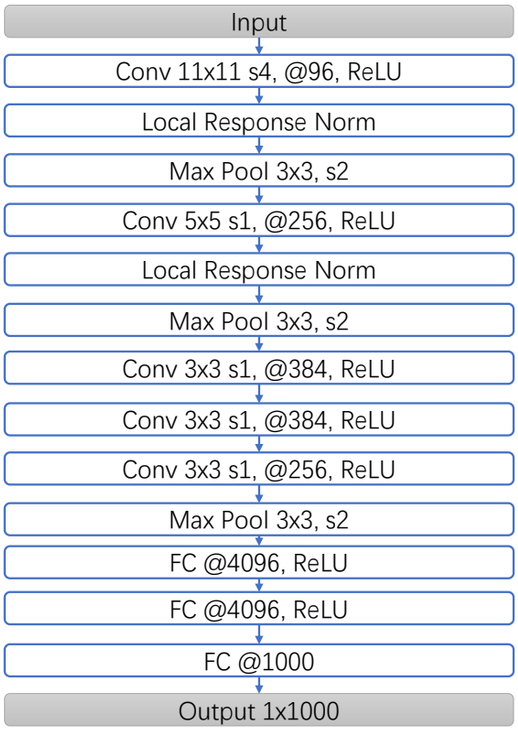


Figure 2.6: The structure of AlexNet, where "Conv 11x11 s4, @96, ReLU" means means a convolutional layer with 96 kernels of size 11x11 and stride 4. ReLU is used as activation function

- It used the non-saturating ReLU activation function, which showed improved training performance over *tanh* and *sigmoid*.
- For propose of avoiding model over-fitting, Data Augmentation to enhance the training images set and Dropout to randomly ignore a number of neurons during training are used.
- Use overlapped maximum pooling on CNN where the step size is smaller than the kernel. Previously, average pooling was commonly used in CNN, and the use of maximum pooling can avoid the blurring effect of average pooling. In the meantime, overlapping effects can enhance the richness of features.
- The LRN (Local Response Normalization) was proposed to create a competition mechanism for the activity of local neurons so that the value with a larger response becomes relatively larger, and other neurons with smaller responses are inhibited. That enhances the generalization of the model.

2.3.3 Inception and GoogLeNet

The first version of GoogLeNet is proposed by Google in the work [1].

Generally speaking, the most direct way to improve network performance is to increase the depth and width of the network, which brings a huge number of parameters. However, a large number of parameters are prone to over-fitting and will greatly increase the amount of calculation. The work [1] proposed the way to solve the above two shortcomings, that is to convert full connections and even general convolutions into sparse connections, as the sparse connections of the real biological nervous system. On the other hand, the work [39] indicates that bloated sparse networks may be simplified without loss of performance.

Previously, in order to break the symmetry of the network and improve the learning ability, traditional networks used random sparse connections, such as the layer C3 of LeNet presented in section 2.3.1. However, the computational efficiency of computer software and hardware for non-uniform sparse data is extremely poor, so the fully connected layer is reused in some network such as AlexNet presented in section 2.3.2, for the purpose of better optimization of parallel operations.

A great number of works show that sparse matrices can be clustered into denser submatrices to improve computing performance. Based on this, the work [1] proposes a structure called Inception, shown as Figure 2.7, that can not only maintain the sparsity of the network structure but also utilize the high computational performance of the dense matrix.

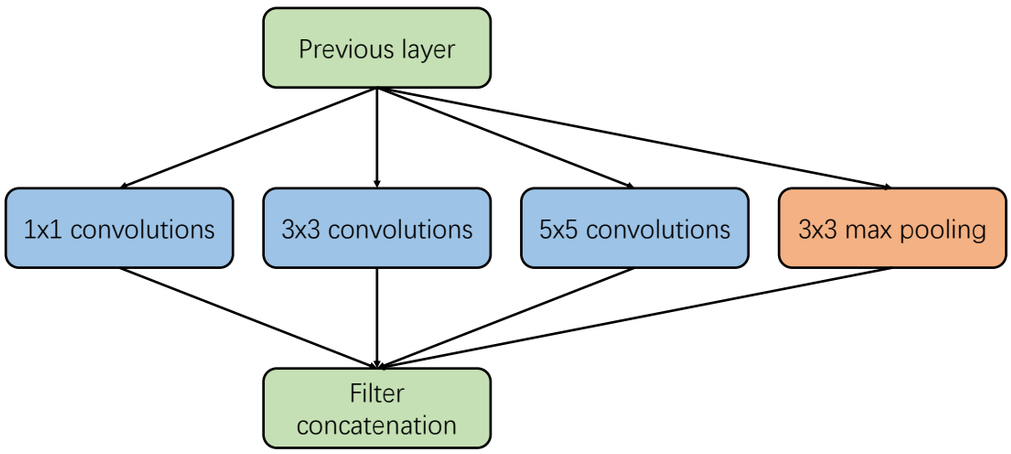


Figure 2.7: The structure of Inception Naive version

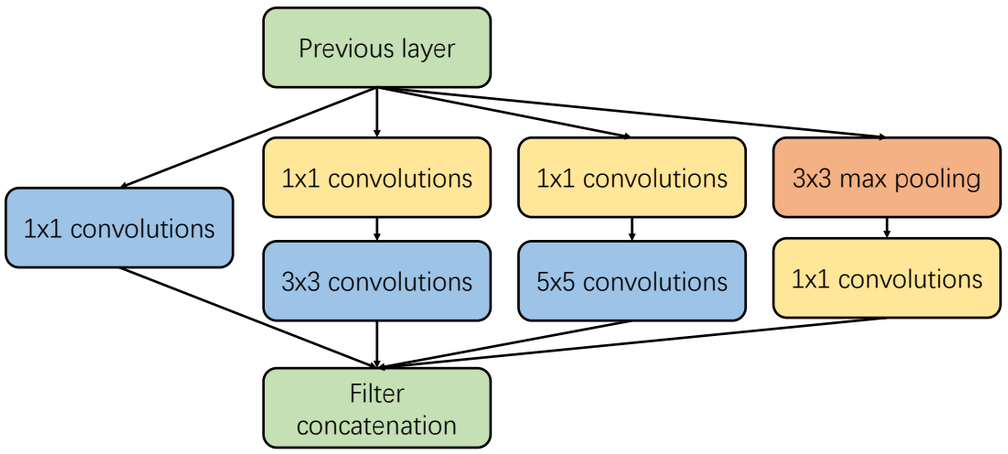


Figure 2.8: The structure of Inception V1

This structure stacks up the commonly used operations in CNNs, which are convolutions with size 1×1, 3×3, 5×5 and pooling operations with size 3×3. By adjusting the padding, the output of each operation has the same size. These outputs are connected as the output of inception. Small filters in inception can extract detailed information of the input, and larger filters can cover most of the input of the receiving layer. The pooling operation can reduce the space size and reduce over-fitting. A ReLU operation is followed each convolutional layer to increase the non-linearity of the network.

In this naive version of inception, the amount of calculation required by the inception, especially for the convolution layer with largest 5×5 filters, is too large. To avoid this case, a 1×1 convolution kernel is added before 3×3, 5×5, and after max pooling, to reduce the thickness of the feature map, which forms the network structure Inception V1, as shown in the Figure 2.8.

Based on the inception module, Google built GoogLeNet, which is the winner of the ImageNet Large Scale Visual Recognition Challenge [30] in 2014. It has significant improvement over ZFNet [40], the winner in 2013, and AlexNet [5], the winner in 2012, and has relatively lower error rate compared with the VGGNet [41], the first runner-up in 2014.

The structure of GoogLeNet is shown as Figure 2.9. The details about GoogLeNet are as follows:

- GoogLeNet uses inception as a module of the structure. The modular structure is more convenient for modification;

- The network eventually replaces fully connected layers with average poolings. This idea comes from work [42]. This replacement can increase the accuracy by 0.6%. However, a fully connected layer is actually added at the end of the network, mainly for flexible adjustment of the output;

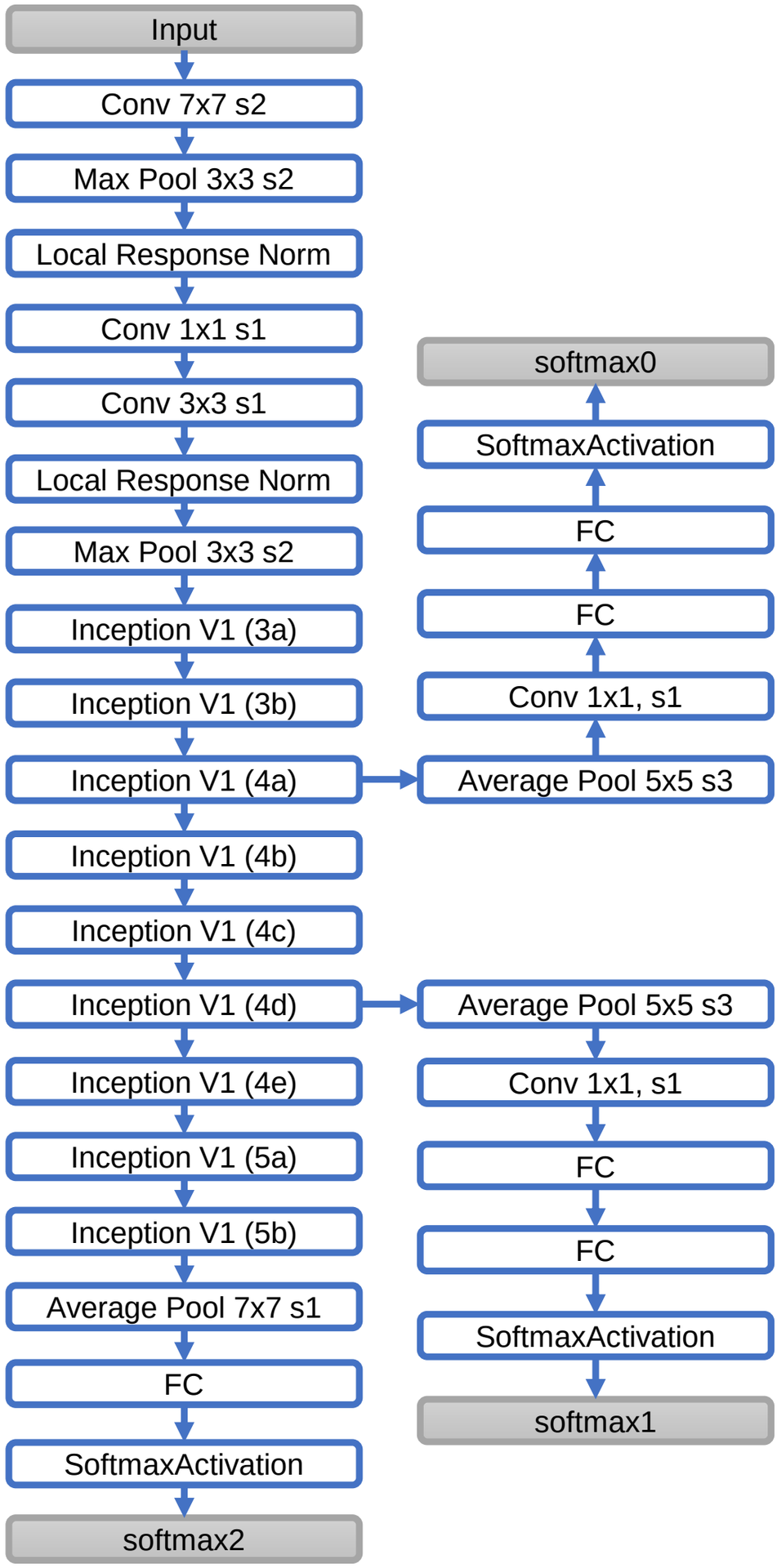


Figure 2.9: The structure of GoogLeNet [1].

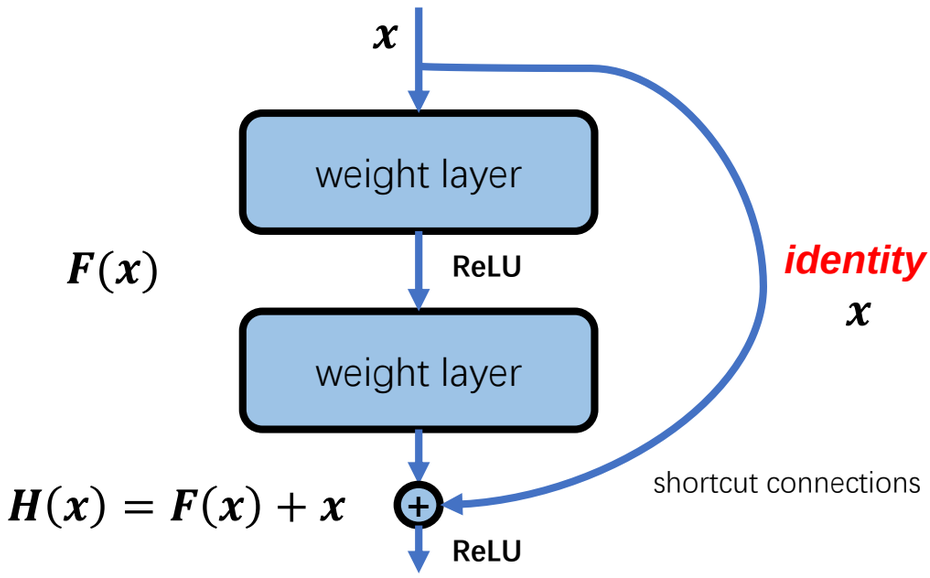


Figure 2.10: A block of the residual network

- Although the full connection is removed, Dropout is still used in the network;
- In order to avoid the disappearance of the gradient, the network adds 2 additional $softmax$ layer to forward the gradient, called auxiliary classifiers. The auxiliary classifier uses the output of a certain middle layer as a classification and adds it to the final classification result according to a small weight (0.3), which is very helpful for the training of the entire network. For well trained networks, these two additional soft-max operations will be removed.

In fact, Google improves the inception in their subsequent work, mainly focused on optimizing the inner structure of the inception module.

The Inception V2 is proposed in the work [43]. In Inception V2, smaller convolution kernels are used to replace a large convolution kernel. For example, two 3x3 convolution kernel is used to replace a 5x5 convolution kernel. A large size convolution kernel can bring a larger receptive field, but it also means that more parameters will be generated. For instance, the 5x5 convolution kernel has 25 parameters, and the 3x3 convolution kernel has 9 parameters. The former is the back 25/9=2.78 times of the person. Hence the GoogLeNet team proposed that a small network composed of two consecutive 3x3 convolutional layers can be used to replace a single 5x5 convolutional layer, which reduces the number of parameters while maintaining the range of the receptive field.

The Inception V3 proposed in the work [44] use the same idea, that a combination of 1x3 and 3x1 convolution kernels was used to replace a 3x3 convolution kernel. Actually, the method of using small convolution kernels is also adopted by other networks such as VGG [41] and SqueezeNet [45], and we also introduce this method in the chapter 4.

The Inception V4 proposed in the work [46] is a combination of the original Inception and ResNet [47] network. The ResNet structure greatly deepens the network depth, and greatly improves the training speed, while the performance is also getting improved. More details of ResNet are presented in section 2.3.4.

2.3.4 ResNet

In the previous chapter, we introduced LeNet, Alexnet and GoogLeNet. Obviously, with the gradual improvement of computing power, such as the wide use of GPUs, deeper and deeper neural networks can be calculated. Although it is generally believed that deeper neural networks can bring richer recognition capabilities, deeper networks also bring problems. For example, the experiments in the work [47] show that with the increase of the numbers of layers in CNNs, the accuracy of the model continues to improve at the beginning. But, in case of the number of layers is continuously increased, the training accuracy and test accuracy decline rapidly. These experiments proved that the deeper network is more difficult to train. In fact, the neural network must continuously propagate the gradient during

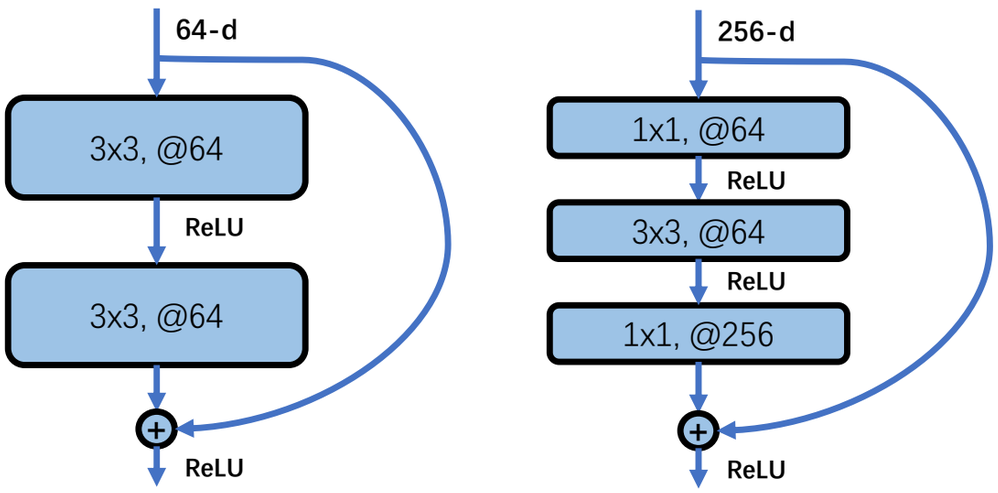


Figure 2.11: The concatenation of the different depth layers in ResNet

the backward propagation. However, due to the chain rule used to calculate the gradient, the difference between the ground-truth and the prediction decreases gradually in backward propagation. Therefore, when the network deepens, the gradient will gradually disappear during the propagation, which makes it impossible to effectively adjust the weight of the previous network layer. Under the circumstances, when the depth of the neural network is continuously increased, the accuracy of the model will first increase and then reach saturation, next, continuing to increase the depth of the network will cause the decrease of accuracy.

To solve this difficulty, then such an assumption is discussed in work [47]: assuming that a relatively shallow network has reached the saturation accuracy, then a deep network is built by adding several identity mapping $y = x$ whose output is equal to the input. The identity mapping increases the depth of the network, but compared to the shallow network, at least the error will not increase, and the accuracy will not get worse in the deep network. The idea of using the identity mapping to directly transfer the output of the shallower layer to the deeper layer is also the inspiration of the famous deep residual network (ResNet), proposed in the work [47].

In the residual network, if it is difficult to train some layers because of the saturation of the number of layers, then the identity mapping will become the main transmission channel in the next learning, that makes the output close to the input, in order to avoid a decrease of accuracy in the deeper layers. An example of a block of residual network composited of several layers is shown as Figure 2.10, where the input of block is x and the expected output is $H(x)$. In this figure, a "shortcut connections" is built to make the input x directly passed to the output as the initial result, and the output result is calculated as $H(x) = F(x) + x$. When $F(x) = 0$, then $H(x) = x$, which is the identity mapping mentioned above. In the case of saturation, the learning goal of this block is changed. It is no longer learning a complete output, but the difference between the target value $H(X)$ and input x, which is called residual calculated as $F(x) = H(x) - x$. Videlicet, the following training goal is to approach the residual result to 0, in this way, as the network gets deeper, the accuracy will not decrease.

Normally, due to the different number of input channels, several residual neural networks cannot be merged directly. Some 1×1 convolutional layers are added to adjust the size, so that convolutions of different sizes can calculate residuals, as shown in Figure 2.11. ResNet is composed of multiple blocks of the residual network as shown in Figure 2.11. More details of the structure of ResNet can be found in [47]

ResNet made a stunning appearance in the ILSVRC 2015 competition. It suddenly increased the network depth to 152 layers and reduced the error rate to 3.57. In terms of image recognition error rate and network depth, it has been greatly improved compared with previous competitions. ResNet won first place in ILSVRC2015. Most importantly, this residual jump structure breaks the traditional neural network's convention, where the $n-1$ layer output can only be used as input to n layers. In ResNet, the output of a certain layer can directly cross several layers as the input of a later layer. It provides a new solution for the problem that the error rate increases when deepening multilayer networks. At this point,

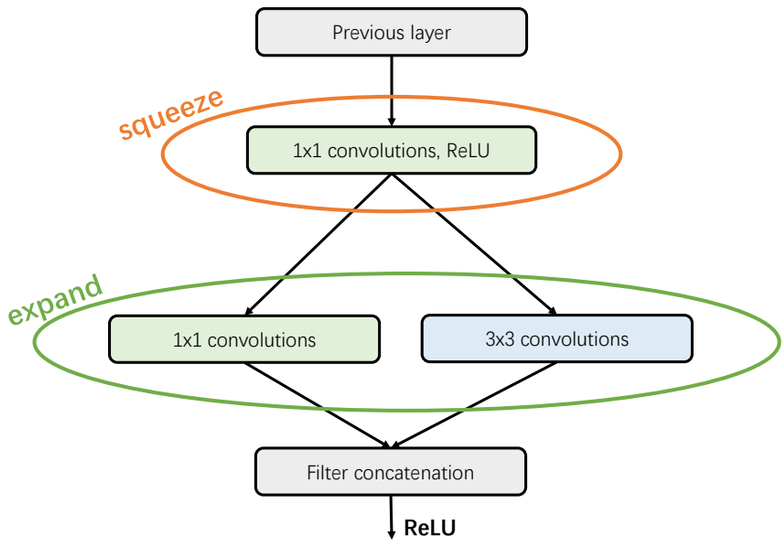


Figure 2.12: The Fire Module in SqueezeNet

the number of layers of the neural network can surpass the previous constraints, reaching dozens of layers, hundreds of layers or even thousands of layers, which provides feasibility for more complex missions such as advanced semantic feature extraction and classification.

2.3.5 SqueezeNet

Figure 2.13: The structure of SqueezeNet

As the neural network continues to deepen, the volume of the parameter of the convolutional neural network is getting increasingly larger, and the calculation time gradually increases. In addition to the accuracy of the prediction, the calculation speed and the volume of the parameters have gradually attracted attention. To compress the network and accelerate the calculation, many methods have been proposed, that will be presented in chapter 4. At the same time, several lighter structures have been proposed. SqueezeNet proposed in [45] is one of the lighten CNN structure.

The fire module shown in Figure 2.12 is the core of SqueezeNet. In the fire module, 1x1 convolution kernels are used as the squeeze part. In the expand part, 1x1 convolution kernels and 3x3 convolution kernels are used to scan the output of the squeeze part. And the numbers of convolution kernels with different sizes are hyperparameters. To make convolution kernels with different sizes generate the same sized output, for large convolutions kernel, a padding operator is needed. Then the outputs generated by convolution kernels with different sizes are concatenated.

The fire module is used to build SqueezeNet. Figure 2.13 shows the global view of SqueezeNet based on AlexNet. We can see that the hidden convolutional layer is replaced by the fire module, and the fully convolutional layer is removed. By using SqueezeNet, compared with AlexNet, it has achieved a 50x compression in model size. In

the meantime, it reaches or exceeds the top-1 and top-5 accuracy of AlexNet.

The fire module uses 1x1 convolutional networks, which proves that small convolutions can replace large convolution kernel convolutions, but at the same time, to keep the receptive field of the module, the 3x3 convolution kernel has not been completely deleted. Interestingly, the building of the fire module is a structural optimization method, which can be used simultaneously with other compression methods introduced in chapter 4.

2.3.6 YOLO and YOLO's family

We have introduced the classic structure and network used for CNNs, and they are milestones in the development of convolutional neural networks. Most of them are proposed only for image classification, or perform well in image classification. However, convolutional neural networks are not only used for classification, but also other complex tasks, such as object detection, object tracking, or semantic analysis. Most of the experiments presented in this thesis are based on object detection systems. Therefore, we introduce CNNs for object detection system in this section.

In comparison with classification, object detection requires more complex output, which not only includes the classification of the objects but also their positions in the input image. There are many excellent object detection systems, such as fast R-CNN [48], SSD [49] and YOLO [2]. YOLO and YOLOv2 adopt the core ideas in other networks. For example, YOLOv2 is inspired by SSD and applies anchors to locate the objects. Hence, we focus on YOLO and its improved version in this section.

YOLO

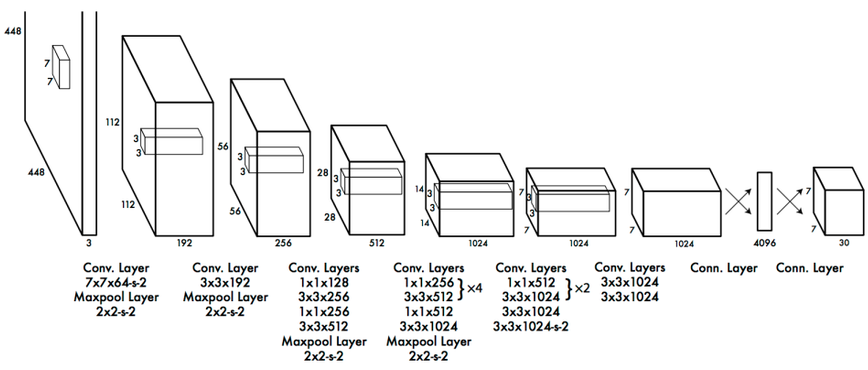


Figure 2.14: The structure of YOLO [2]

You Only Look Once (YOLO)[2] is one of the approaches to object detection. It extracts features from the entire image by using CNN to generate bounding boxes, shown as 2.14, and each bounding box predicts a position of one detected object and the accuracy of the prediction.

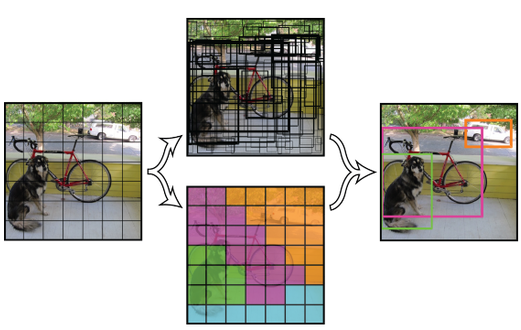


Figure 2.15: Image is divided into SxS grids ($S = 7$ in this figure) [2]

As Figure 2.15 shows, the way YOLO works as if it divides the input image into S x S grids. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. Each grid cell could generate B bounding boxes. Each generated bounding box consists of 5 predictions: x,y,w,h and confidence C. The (x, y) coordinates of the center of the object relative to the grid cell. The weight w and height h predict the size of the box relative to the whole image. And confidence C, defined as $Pr(Object) * IOU^{truth}_{pred}$, presents the probability of object falling into this bounding box and intersection over union(IOU), which is one of the measurement criteria and

introduced in section 3.2.3. Each grid cell also predicts N conditional class probabilities $p_i = Pr(Class_i|Object)$, and N is the number of categories in the image set. In short, the image is divided into S x S grid, each grid has $(B*5+N)$ value (as shown in Fig 2.16, YOLO v1), and the prediction (output) of YOLO's neural network is encoded as an $S*S*(B*5+N)$ tensor. The final prediction will be made by the predictor with the highest IOU and the bounding box.

During training, YOLO optimize a multi-part loss function shown as Equation 2.18, where $\mathbb{1}_i^{obj}$ denotes if an object appears in grid i and $\mathbb{1}_{ij}^{obj}$ denotes that j^{th} bounding box predictor in grid i is responsible for that prediction. The first two parts in the equation penalize the loss from bounding boxes coordinate predictions. The 3^{rd}, 4^{th} and 5^{th} parts in the equation penalize the loss from classification.

$$\begin{aligned}\varphi =&\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{obj}(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2\\&+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{obj}(\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{h_i}-\sqrt{\hat{h}_i})^2\\&+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{obj}(C_i-\hat{C}_i)^2\\&+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{obj}(C_i-\hat{C}_i)^2\\&+\sum_{i=0}^{S^2}\mathbb{1}_{i}^{obj}\sum_{c\in classes}(p_i(c)-\hat{p}_i(c))^2\end{aligned}\tag{2.18}$$

YOLOv2

YOLOv2[50] is a upgraded version of YOLO. Compared to YOLO (version 1), YOLOv2 has the following improvements:

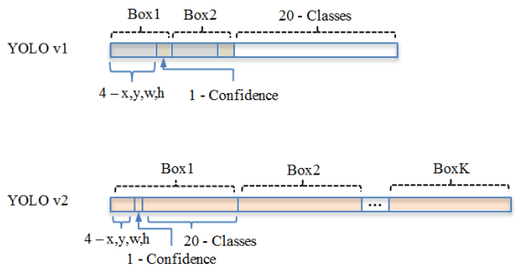


Figure 2.16: Content of the bounding boxes

- Batch Normalization is used to improve the convergence.
- In YOLOv2, fully connected layers are removed, so that using images with the varied resolution is possible. In order to be robust to different resolution of the image, during the training of YOLOv2, the input image will be resized after each 10 batches.
- Anchor boxes are used to predict bounding boxes. The prediction of each bounding boxes is as shown in Figure 2.16. Thus, the prediction (output) is encoded as a $S*S*B*(5+N)$ tensor. Referring to [50], $B = 5$ and S should be odd.

YOLOv2 has raised the speed and accuracy of object detection to a new level through a series of remarkably effective tricks. These tricks are not only extremely effective in YOLOv2, but also have reference value for our other tasks, such as the application of high-resolution transfer learning to semantic segmentation, and the application of multi-scale training to image classification tasks.

Tiny-YOLO

Tiny-YOLO is a tiny version of YOLOv2, shown as Fig 2.17. It uses the same mechanism to process the images and generate predictions, but with a smaller architecture: It contains

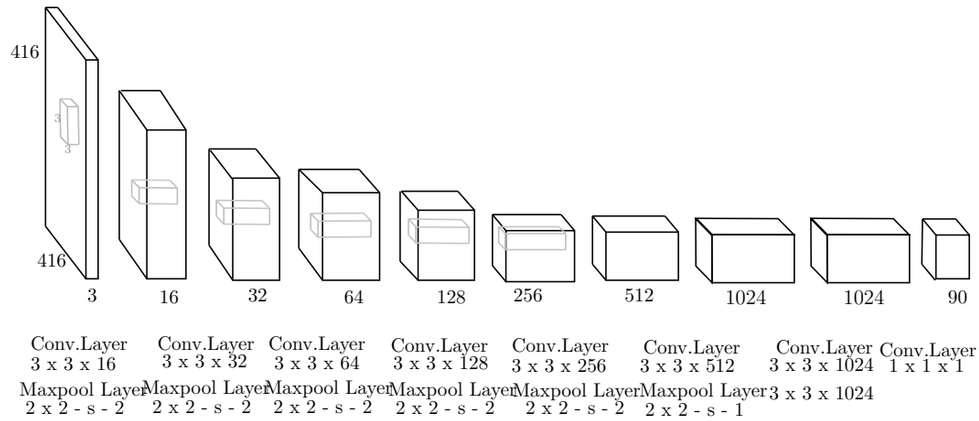


Figure 2.17: The structure of tiny-YOLO

8 convolutional layers before the final output convolutional layer. Each convolutional layer is followed by a batch normalization layer and leaky ReLu activation. (batch normalization layer can be simplified as bias add layer during the inferring). We use S=13, B = 5 and N =13, so the final convolutional layer output a tensor with shape $13 * 13 * 5 * (5 + 13)$.

Due to its tiny size and limited computing resources, tiny-YOLO has become one of the main networks we use. However, due to the limited depth, the accuracy of complex datasets is not as good as the original YOLOv2 network.

YOLOv3 and More

YOLOv3 was proposed in work [51] a few years later after YOLOv2. Compared to version 2, there are several obvious advantages:

- The detection of small objects shows even better performance.
- The detection of dense objects or occluded objects has been improved.
- The generalization ability is better.

This is mainly due to the following updates:

- Multiple predictions layers with different sizes of up-sampling are added to make it more sensitive to the small objects.
- Instead of softmax, logistic loss function is applied to YOLOv2, it brings better results when facing the complex target categories.
- The network is deepened due to the approach as ResNet.
- The router mode is improved for some concatenate layer.

YOLOv3 has achieved amazing results both in speed and accuracy. However, it is worth mentioning that the author stated that because YOLOs are used for military purposes and privacy snooping, he thought it was immoral, and therefore no longer engages in computer vision research. This statement has caused researcher to rethink the moral and ethical issues of artificial intelligence.

2.4 Frameworks for Neural Networks

We introduced some classic neural network structures in section 2.3. Generally speaking, these algorithms can be implemented through different programming languages. For example, Python and MATLAB is a widely used language for scientific computing, that can implement matrix operations relatively easily. Furthermore, the more hardware friendly languages, such as C/C++, and their math library, such as General Matrix Multiply (GeMM) or Basic Linear Algebra Subprograms (BLAS), can be used to build or accelerate the basic structure of convolution. However, most of the operations in neural networks are modular,

and they are composed of some computational primitives, as introduced in chapter 2.2. Common operations can be encapsulated to form machine learning frameworks. Using mature machine learning frameworks can bring some meaningful benefits, such as:

- Since most of the computational paradigms of machine learning are encapsulated in frameworks, more complex networks can be constructed more easily.
- Most of the frameworks have proposed the option that executes in GPU, which allows avoiding rewriting the GPU supported library, such as the code for CUDA [52] or OpenCL [53].
- For conventional machine learning modules, if the backward propagation algorithm is applied for training, the computing for gradients is normally provided by the framework and no longer need to rewrite it.
- Some frameworks provide optimization methods for special devices, such as the compression methods for embedded devices, that makes the neural network more convenient to be transplanted.

In this section, some widely used frameworks are introduced. It should be noted that the frameworks of neural networks are difficult to compare with each other, which means that there is no best or worst framework, but it needs to make choices based on the purpose of tasks. Meanwhile, the framework of the neural network is a commercial technology, they are also developing rapidly, or are gradually eliminated by users. This is just an overview of the framework as of the time of writing.

2.4.1 TensorFlow

TensorFlow [54] is one of the most popular deep learning frameworks today. This is an open-source framework developed and maintained by Google. Many famous groups such as Gmail, Uber, Airbnb, Nvidia are using it. Here we introduce some of its characteristics:

- Python is the most convenient client language for TensorFlow (With the development, Python2 is no longer supported by TensorFlow). However, TensorFlow also provides experimental interactive interfaces for JavaScript, C++, Java, Go, C#, and Julia.
- TensorFlow not only can be executed in a powerful computing cluster but also considers the ability to run models on mobile platforms such as iOS and Android.
- TensorFlow has visualization tools (Tensor Board etc.) that can more intuitively see the structure of neural network graphs and the distribution of data. This facilitates scientific research.
- TensorFlow is still developing and iterating rapidly, so version compatibility issues often occur.
- TensorFlow is very flexible because it provides a wealth of options for building deep learning networks.
- The flexible proposed options also mean that more details need to be considered when using it. Therefore, compared with other frameworks, TensorFlow is relatively more cumbersome to use. In its new version TensorFlow 2.0, since the easy-to-use Keras interface is used as the default interface, this issue has been significantly improved.

Due to its powerful functions, TensorFlow is widely used in the industrial field. As Google continues to optimize and improve the framework, it can be sure that it will become more powerful and easier to use.

2.4.2 PyTorch

PyTorch [55, 56] is a deep learning framework based on Python and its Torch library written in Lua. It was created by Facebook and is currently widely used in academia and industry. As another well-known framework Caffe2 project merged into PyTorch, it is more widely used in academia and industry field.

Based on the Auto-grad, a powerful gradient computing kernel in PyTorch, we can design the network dynamically without having to pre-define a static network diagram to perform calculations. Therefore, the PyTorch library is simple to use. As well, it is compatible with Python library as *NumPy* and *SciPy*, PyTorch has quickly become the mainstream deep learning framework in academia. But at the same time, the shortcomings of PyTorch have also been criticized. It has been mentioned repeatedly by industrial developers that PyTorch is difficult to deploy.

In general, compared to TensorFlow which makes the framework more powerful, PyTorch pays more attention to make the framework simpler to use. As well, TensorFlow is more popular in the industry, and PyTorch is used increasingly in academia. It is difficult to say which one is better. It is interesting to note that with the update, both frameworks are overcoming their shortcomings, which makes the two frameworks more and more similar. It is foreseeable that the frameworks will be easier to use and more powerful in the future.

2.4.3 Darknet

Darknet [57] is an open-source neural network framework written in C and CUDA. It is fast, easy to install, and supports CPU and GPU computation. YOLO's authors developed and used the Darknet framework to release YOLO.

On one hand, there is no complex API or code encapsulation like TensorFlow and PyTorch in Darknet, so it is noticeably light and easy to modify. On the other hand, Darknet does not require the special dependencies. Although it can support for OpenCV, CUDA and CuDNN in GPU, it can still compile and run without these. Hence, Darknet is very suitable for being modified and recompiled. This feature is suitable for studying the underlying layer and can be more convenient to improve and extend it from the underlying layer. It is worth mentioning that since Darknet does not require dependencies, it particularly easy to be migrated, especially quite easy to deploy on embedded systems with few resources.

Since the experiments involved in this paper are mainly to study the impact of changing the underlying computing structure of convolutional neural networks, Darknet has become one of the main applied frameworks.

2.4.4 Other frameworks

In addition to the above-mentioned frameworks, there are other well-known frameworks.

- **Theano** is one of the oldest frameworks. It uses tensors to represent neural network operators, instead of matrices, that is a template for the subsequent frameworks. Unfortunately, the project is no longer updated.

- **Keras** is a very concise framework, which hides a lot of complicated options. It is rather a set of high-level APIs than a framework. Once Keras+Theano was an immensely popular combination. Now, TensorFlow also provides Keras interfaces, so that more users can quickly build the framework.

- **Mxnet** and its high-level API **Gluon** are highly effective in parallel on multiple GPUs and multiple machines, especially in Amazon Web Services (AWS). In 2016, it was chosen by AWS as the official deep learning platform for cloud computing.

- **PaddlePaddle** is one of China's oldest framework developed by Baidu. Although it is not as famous as TensorFlow and PyTorch, relying on excellent performance and rich models, it still occupies a portion of the market share among Chinese developers. Now PaddlePaddle grows up rapidly, and due to Baidu's business promotion, the users have rapidly increased and gradually formed a new deep learning ecosystem.

- **Deeplearning4j**, as his name, is a deep learning framework developed for Java. It is compatible with any JVM language, such as Scala and Kotlin. Since Java is popular in many mobile developments, such as Android, this framework is also often used.

Some of these frameworks are developing rapidly, and some have stopped updating. When choosing these frameworks, we must consider the supported functions, development complexity, development languages, as well as the life cycle of the framework itself. A suitable and efficient deep learning framework according to the characteristics of the deep learning project can achieve a multiplier effect.

2.5 Summary

In this chapter, the history and development of artificial intelligence are introduced. Then, the convolution neural network, one of the most successful algorithms in machine learning, and the classic structure and frameworks for CNN are also discussed.

In this chapter, the history and development of artificial intelligence are introduced. Then, one of the most successful algorithms in machine learning, convolutional neural networks, as well as the classic structure and frameworks for CNN are also discussed.

Through the introduction to the building and different structures of CNN, a global view of the development of CNNs has been constituted. It can be seen that CNN gradually became deeper and more complex from the original LeNet model. At the same time, batch normalization, ResNet, compressed convolution kernel, and other technologies have been proposed and applied. Additionally, the more appreciate accelerators such as GPU has been applied to execute CNN. All the technologies make CNNs more powerful for different tasks.

Nowadays, machine learning is widely used in data mining, computer vision, natural language processing, biometrics, search engines, medical diagnosis, credit card fraud detection, stock market analysis, DNA sequence sequencing, speech and handwriting recognition, strategic games, and robotics field, etc.

Chapter 3

Motivation and Scope

Convolutional neural networks have been widely used in multiple domains. But it has also brought difficulties when applied to some special use case. We introduce these difficulties in this chapter. As well, we introduce the scope in which we work to solve the difficulties.

3.1 The New Challenges of Neural Networks

Neural network technology is more and more widely used in the fields of image recognition, speech recognition, and signal processing. Their recognition accuracy is getting higher and higher, and it has surpassed the traditional algorithm in many fields. In the meantime, the size of the neural network is getting larger and larger, and the network is becoming more and more complicated. Take image recognition as an example. In 2012, when deep learning and convolutional neural networks were just beginning to attract attention, the winner of the ImageNet competition [58] used an 8 layer neural network. Then in 2015, the network with 152 layers was used, and in 2016, it was a network with 1207 layers won the competition. Until now, thousands of layers in deep neural networks are quite common. Complex neural networks have gradually surpassed many traditional algorithms in many fields, but brought many new challenges.

Excessive energy consumption is caused by complex algorithms. The problem of energy supply and heat dissipation due to excessive energy consumption has always been a challenge for large data centers. Fortunately, we can solve it by expanding the scale of equipment and increasing heat dissipation, even if it will bring higher costs. However, for the smaller devices, especially embedded devices which have limited energy and computing resource, the problem is more difficult to solve. Some possible difficulties and challenges are as follows:

- The first is energy consumption. Large-scale neural networks require a lot of calculations and consume a lot of energy. Taking unmanned aerial vehicle (UAV) as an example, it is difficult to say how low power consumption is acceptable, but smaller batteries carried by UAV are always favorable than the bigger ones. Because this will greatly reduce the burden on the UAV's flight system. Therefore, compressing neural networks is an issue worth considering.

- The second is computing resources and computing speed. Taking autonomous driving as an example, we need to determine the color of traffic lights within a specified time to make a decision. But for limited computing resources, an excessively large neural network will be very time-consuming. Then, for time-sensitive systems, accelerating neural networks is an inevitable problem.

- It is necessary to consider the cost of memory and communication. For a large-scale neural network, due to its parameter volume, a large amount of memory is required. For systems with insufficient memory, the weights can be saved in external storage, which brings challenges to the bandwidth of external memory. In addition, as introduced in section 3.3.1, the distributed systems with different accelerators can be used to accelerate machine learning algorithms. This also imposes the requirements on the communication bandwidth between different devices. Larger neural networks will not only spend more time transmitting information but also consume much more energy in transmitting. For all of the above, how to reduce the volume of neural networks is gradually being discussed.

In fact, due to the limited computing resources, embedded devices do not undertake computing tasks in many cases. Many embedded devices are used as sensors, that is, they are only responsible for collecting information, and the information is eventually transmitted to the central server to calculate. However, in this paradigm, on one side the central server needs to receive information from different edge devices and process them concurrently, which brings challenges to the bandwidth and computing power, on the other side, a large number of data streams are transmitted between different devices, which also leads to security risks [59]. Therefore, putting the preliminary calculation of data on the edge devices, thereby reducing the pressure on the transmission network and on the server, is the trend of distributed embedded systems now [59]. A compressed and accelerated neural network suitable for embedded devices is needed.

It should be pointed out that the acceleration to CNNs is usually discussed from two perspectives:

- The acceleration for training phase: that uses different training methods (e.g. SGD [35] and ADAM [60]) to make the trained CNN find the optimal solution faster (faster convergence).
- The acceleration for inferring phase: that focus on speed up the forward propagation and reduce the computing resources used in inference.

Because of the limited computing resource of embedded systems, the CNNs are always pre-trained offline in the cluster and then implemented in embedded systems for the inference stage. Therefore, The acceleration of embedded machine learning discussed in this thesis actually refers to the optimization for the inference phase.

3.2 Tasks and Approaches

The complexity and heaviness of the convolutional neural network is a common problem. There are general proposed compression and acceleration methods for CNNs, which are introduced in chapter 4. However, the same methods applied to different tasks may cause different effects. For example, binary networks proposed in work [22] work well for the classification of image, but when applied to the object detection, it leads to a big loss of accuracy. For these, it is necessary to clarify the basic tasks of convolutional neural networks.

It should be noted that CNNs, as well as the compression and acceleration methods for CNNs, can be used in many fields such as sound processing and signal processing, but we mainly study it in image processing. On the one hand, because the image processing is widely used and has a rich dataset, it is quite simple to establish a benchmark. On the other hand, rich tasks with different difficulty also allow having a good performance measure of different CNNs and their methods. In this section, we introduce the basic tasks of convolutional neural networks in the image field and propose our work scope.

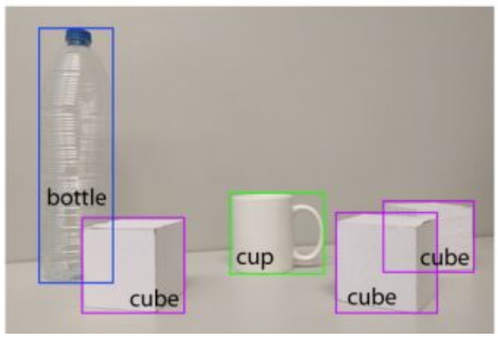
3.2.1 Tasks of Computer Vision

There are many different tasks with varied complexity in the field of computer vision. It is generally believed that the following four tasks are the most basic and discussed issues:

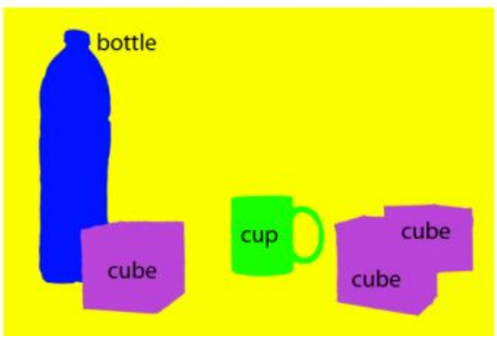
Image Classification is an easier task in the field of computer vision. Given an input image, the aims of image classification task is to determine the category of the image, shown as Figure 3.1a. Each image can contain one or more objects. Most of the classic neural networks introduced in section 2.3 such as LeNet, AlexNet, VGG, GoogLeNet, and ResNet can be used for classification.

Object Localization and Object Detection are two more complicated tasks than classification. These tasks not only aim to classify objects, but also to find out the position and relative size of the objects in the picture. In a general way, these objects are wrapped by bounding box, as shown in Figure 3.1b. The difference between the two tasks is that the number of objects in each image is known for localization but is unknown for detection. The networks R-CNN, SSD, YOLO and their improved versions are the frequently used neural network for these takes.

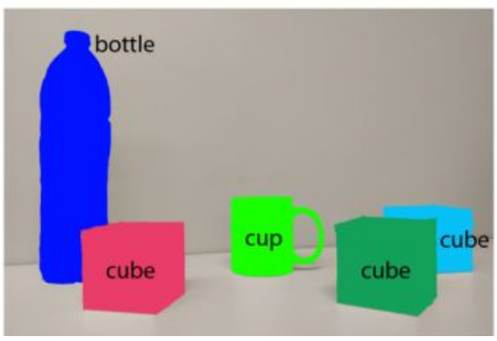
(a) Image Classification



(b) Object Localization and Detection



(c) Semantic Segmentation



(d) Instance Segmentation

Figure 3.1: Examples of basic computer version tasks

Semantic Segmentation is a more advanced task of object detection. Semantic segmentation needs to further determine which object each pixel belongs to, shown as Figure 3.1c. In semantic segmentation, the output result generally has the same size as the input result, which encodes the probabilities that each pixel belongs to. Therefore, convolution-then-deconvolution is the commonly used method for semantic segmentation.

Instance Segmentation distinguish different instances belonging to the same category, which is different from semantic segmentation, shown as Figure 3.1d. For example, when there are multiple cats in the image, the pixels of two cats are predicted to be the category "cat" in semantic segmentation tasks. But for instance segmentation, it needs to distinguish which pixels belong to the first cat and which pixels belong to the second cat.

We introduced the basic tasks in the field of computer vision. There are also some other tasks, such as face recognition, object tracking, and counting of the dense crowd, etc. They are generally considered as a special branch or a combination of basic computer vision tasks, and could usually use common technology to solve problems. There are also more optimized algorithms for this particular task, for example, the Fully-Convolutional Siamese networks proposed in work [61] is a special network for object tracking. Due to the wide variety of the extends and variants of these tasks, we will not go into details here.

In general, for the platform with limited resources, such as edge computing systems or other embedded systems, only relatively naive tasks need to be performed. For example, we do not need a general drone to accurately recognize the outline of a person, but just locate and track. Normally these algorithms commonly used in embedded devices need to be accelerated. Therefore, we focus on these naive tasks, such as classification and object detection. As shown above, we know that classification is also one of the aims for object detection. Hence, the scope of our work is object detection. We will use a CNN-based YOLO architecture and its improved versions. Even though we mainly target applications such as object detection, the methods proposed are generic for CNNs.

3.2.2 Image Sets

For industrial applications, there are special datasets for different tasks. But in the academic field, some commonly used datasets are used to measure the performance of algorithms. We introduce them subsequently. The methods proposed in the next chapter have been applied

to different datasets, in order to prove that the methods have a good ability to generalization.

Image Set for Classification

Classification is one of the basic issues which is easier than object detection. The image for classification usually contains one or more instances in each image, and the labels in the dataset usually contain category information but not location information. We will introduce MNIST, Cifar10/Cifar100, and ImageNet. The number of categories they contain are gradually increasing, and the relative difficulty of classification is gradually increasing.

MNIST [62] is a set of handwritten digits images. It has a training set of 60,000 examples and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Cifar10 [63] , a set of 60000 32x32 color images belonging to 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. This is an image set with small images. It is usually used to test a classifier. **Cifar100** is a similar image set to Cifar10 but contains 100 classes. It is more difficult to classify than Cifar10.

ImageNet [58] is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently, there are over five hundred images per node on average. In the following chapters, we did not use ImageNet to train the network. However, for training a YOLO network, convolutional weights pre-trained on Imagenet are used.

Image Sets for Object Detection

The object detection system needs to pick up the object in images and then classify them. Therefore, the dataset we use should contain different objects, as well, the dataset needs to mark the location and size of different objects in the images. The most used dataset for object detection is MS-COCO (also called COCO-2017) and Pascall VOC. SDC-2018 is also an image set proposed for object detection in 2018. The details of these image sets are as followed:

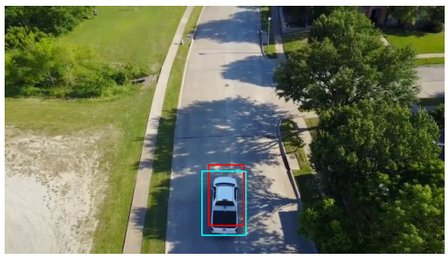
SDC-2018 [64] is an image dataset proposed for the 2018 System Design Contest. This data-set contains 13 class of images and 95 sub-categories. [1] In this data-set, most of the images have one small object occupied 1-2% of the captured images (640x360), which is the main character of UAV-view images provided from DJI company, captured by unmanned aerial vehicles (UAV). In this dataset, there is at most one object in each image. In chapters 5 6 and chapter 7, we use 66K randomly selected images from this dataset for training our proposed networks.

COCO-2017 [65] is a large-scale object detection, segmentation, and captioning dataset. COCO has 80 object categories in 330K images. Each image in COCO contains one or more objects, and 1.5 million object instances are contained in total for this image set. Since the detection of many objects in COCO such as sunglasses, cellphones or chairs is highly dependent on contextual information, it is important that detection datasets contain objects in their natural environments. In the COCO dataset, the author strives to collect images rich in contextual information. In chapter 8, we use 118287 randomly selected images from the dataset for training the networks.

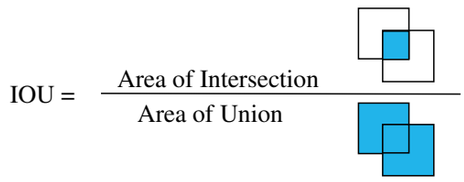
3.2.3 Performance Evaluation

For the different tasks, criteria for measuring network performance are also different. Different measure criteria for classification and object detection are discussed in this section.

[1] The latest version of the data contains only 12 classes of objects, but we still use the original version which contains 13 classes for our experiments.



(a) predicted bounding box (in red) and ground truth (in blue).



(b) IOU between two boxes (prediction and ground truth)

Figure 3.2: An example of object detection and the illustration of IOU.

Measure Criteria for Classification

Classification task consists in determining the category of one object. For a given object, the output of the classification system encodes the probabilities that the object belongs to each category. Based on these probabilities, the two most intuitive criteria are adopted. These two measure criteria are applied in the famous ImageNet ILSVRC competition [30].

Top-1 Error Rate : The frequency that the predicted category with the highest probability is not consistent with the truth label.

Top-5 Error Rate : The frequency that the predicted categories with the top five highest probability does not contain the truth category.

Measure Criteria for Object Localization

Intersection over Union (IOU) is an evaluation metric used to measure the accuracy of an object localization system on a particular dataset. We often see this evaluation metric used in object detection challenges, such as [66] and [64]. Any algorithm that predicts bounding boxes as output can be evaluated using IOU.

More formally, to apply Intersection over Union to evaluate an object detector we need: the ground-truth bounding boxes and the predicted bounding boxes from our model, shown as Figure 3.2a. In this example, the ground-truth bounding boxes are the smallest rectangle that can wrap the object. It is hand-labeled, and it expresses where the image our object is, as well as the size of the object. The predicted bounding box is the output of the object detection system. Just like the ground-truth, it is a rectangle expressing the position and the size of the object but generated by the prediction system.

The IOU is computed as shown in Figure 3.2b. In the numerator, we compute the area of overlap between the predicted bounding box and the ground-truth bounding box. The denominator is the area of the union, or more simply, the area encompassed by both the predicted bounding box and the ground-truth bounding box.

We can see that the IOU considers not only the position of the detected object but its size. If the predicted position and object size are close to the ground-truth, the IOU will be high.

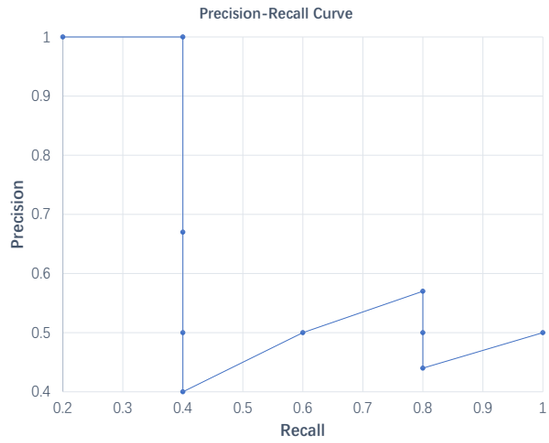
Measure Criteria for Multiple-Object Images

Both the top-x error rate and IOU are measure criteria for a single instance. When there are multiple objects in one image, we can use the average of the IOU or error rate to express the performance of the systems. Meanwhile, for multi-object images, more diverse measure criteria can be applied.

For classification, if the predicted category with the top x highest probability is consistent with the truth label, the prediction is considered as correct in top-x. Otherwise, it is incorrect. For object detection, when the IOU is greater than a given threshold, the object is considered as found correctly, and when IOU is less than the threshold, it is marked as not found correctly. Average Precision (AP) criterion is a more common standard than IOU or top-x when we care more about the correct rate instead of the accuracy of each object.

Rank	Correct?	Precision	Recall
1	True	1.0	0.2
2	True	1.0	0.4
3	False	0.67	0.4
4	False	0.5	0.4
5	False	0.4	0.4
6	True	0.5	0.6
7	True	0.57	0.8
8	False	0.5	0.8
9	False	0.44	0.8
10	True	0.5	1.0

(a) The truth table of the result of object detection.($m = 5$,$n = 10$)



(b) The Precision-Recall Curve.

Figure 3.3: Recall Precision calculation for evaluating ranked object detection

Most of the criteria for multiple-object images are based on the error rate or IOU. Here taking object detection system and IOU as an example, we express other criteria for multiple-object images. The same method can be used for the classification system with the top error rate.

In case that an object detection system finds n objects (some correct and others not) from a batch of images, which contains a total of m objects, if k objects are correct, we can get two indicators:

- $precision = k/n$ represents the proportion of correctly found objects (aka. True Positive).
- $recall = k/m$ represents the proportion of correctly found objects among all the objects in this batch of images.

Figure 3.3 shows an example of these two indicators and demonstrate the calculation of the average precision. In this example, the whole dataset contains 5 objects only ($m = 5$). We collect all the predictions made for objects in all the images and rank them in descending order according to the predicted confidence level. The second column indicates whether the prediction is correct or not. In this example, the prediction is correct if $IOU > 0.5$ ($threshold = 0.5$). Let us take the row with rank #3 and calculate the precision and recall. Precision is the proportion of TP $= 2/3 = 0.67$. The recall is the proportion of TP out of the possible positives $= 2/5 = 0.4$. Recall values increase as we go down the prediction ranking. However, precision has a zigzag pattern, where it goes down with false positives and goes up again with true positives until the recall reaches 100%. Then, the recall-precision curve can be constructed as shown in Figure 3.3. The general definition for the Average Precision (AP) is finding the area under the precision-recall curve above.

$$AP = \int_0^1 p(r)dr \tag{3.1}$$

where p is the precision and r the recall. AP is a criterion considering the precision as well as recall, and the ranks of the detected objects.

The mAP (mean average precision), another frequently used measure criteria, is the average of AP. In some context, we compute the AP for each category separately and average them. But in some context, the AP is calculated for all categories together, in this case, mAP is the same as AP. For example, under the COCO context, there is no difference between AP and mAP [65].

As presented above, we introduced different measure criteria for an object detection system, such as IOU, precision, recall, AP. Generally speaking, these criteria are used to measure the accuracy of prediction for one image. To measure the performance of the network, it is necessary to use the network to process an evaluation dataset containing multiple images, and then calculate the average value of this dataset.

In addition to the accuracy of prediction, other performances may be measured for specific systems. For real-time systems, processing speed is a standard, which can be expressed using

actual calculation time, or it can be inferred with the number of calculations. For embedded systems and edge devices, measure criteria such as power consumption and the utilization of computing resources should also be considered.

In the following, we will optimize object detection systems for embedded devices and edge computing devices with limited resources. Hence, we will comprehensively consider a variety of criteria.

3.3 Runtime Platforms for CNNs

When executing a neural network, a good runtime platform can provide a lot of help, such as simplifying development or accelerating calculations. For example, with the proposition of AlexNet, GPUs which have the powerful ability of parallelism are widely used to calculate convolutional neural networks. In this section, we will briefly analyze the different platforms, and then introduce the platforms used in this thesis.

3.3.1 Platforms and Accelerators for CNNs

The central processing unit (CPU) is the most versatile computer processing core. However, most of the classic CPU is Single Instruction Single Data (SISD) system. For a classic CPU, whether it is a Von Neumann or Harvard structure, it is necessary to read instructions and data multiple times. Through the introduction in chapter 2, there are many repetitive operations in convolutional neural networks, such as repeated multiply-accumulate (MAC) operation. It causes waste to read repeated instructions many times in a SISD system. Therefore, accelerators more suitable for CNNs need to be introduced. In this chapter, three accelerators for CNNs are discussed: GPU, ASIC, and FPGA.

GPU: Single Instruction Multiple Data Accelerator

It is not difficult to see that there are many identical calculations in machine learning. Taking CNNs as an example, it uses a lot of convolution calculations. As the introduction in section 2.2, the convolution can be converted into a multiply-accumulate operation (MAC). So the computer repeats the MAC many times but different input data. A SISD system such as a classic CPU needs to read the same instructions multiple times so that takes a lot of time.

The Graphics Processing Unit (GPU), especially General-Propose GPU (GPGPU), is a Single Instruction Multiple Data (SIMD) system. The GPU will divide complex calculation into millions of separate tasks to solve them at the same time. For the same MAC operation, it reads instructions only once and then performs parallel operations on different data, which undoubtedly increases the speed of operation. This feature makes GPU widely used in the field of machine learning. The combination of CPU + GPU has also become one of the most used accelerator architecture in the field of machine learning. In this combination, the complex and non-repetitive work is placed on the CPU side, such as the reading and saving of network parameters, while the repetitive work is placed on the GPU side, such as the calculation of convolution.

Although the development of GPU components requires special programming languages and tools, such as CUDA and its compiler for Nvidia GPUs, most machine learning frameworks provide encapsulated interfaces for executing CNNs in GPU, which greatly simplifies development. This is another reason for the widespread popularity of GPU accelerators.

ASIC: Powerful but Inflexible System

Since there are many identical operations, for accelerating the CNNs, the Application-Specific Integrated Circuit (ASIC) is another option. The ASICs use a fixed circuit design and is optimized for a specific algorithm, so that the ASIC can be better than GPU in terms of calculation speed and energy consumption.

But ASICs also have shortcomings that cannot be ignored:

- Compared with general-purpose chips such as GPU and CPU, the ASIC chip development process is relatively complex. As well, compared to software compilation, tape-out for ASIC also requires a relatively long time and much more cost. Therefore,

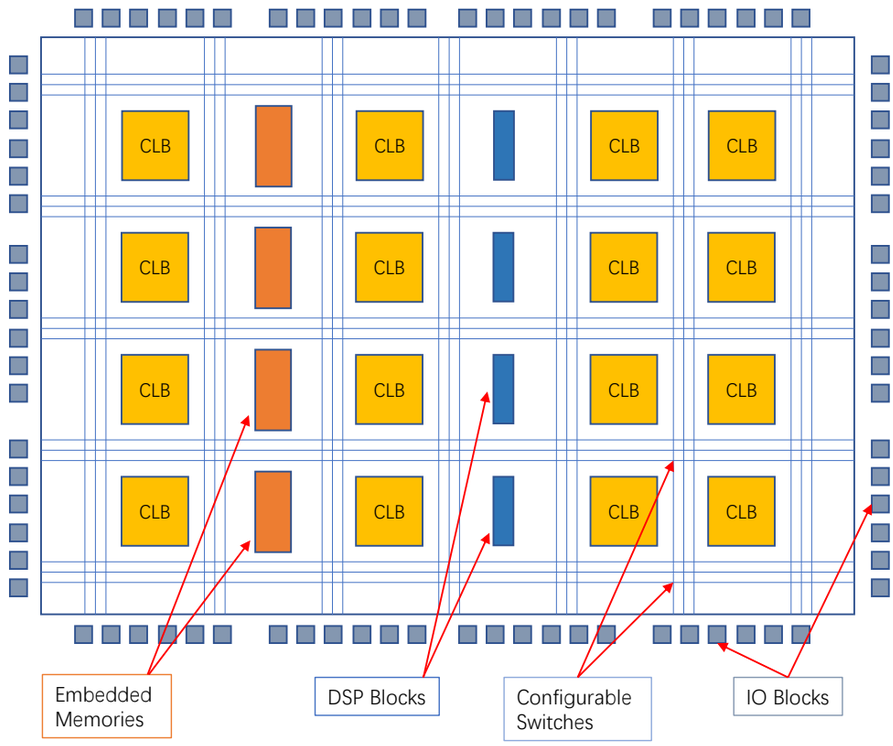


Figure 3.4: Mesh FPGA architecture. For more detail of FPGA architecture, see [3].

if ASIC ships cannot be mass-produced, it will bring excessive development cost and time cost. This is a general weakness of ASIC.

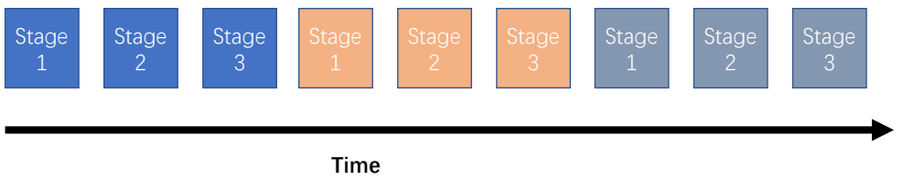
- For the field of machine learning, he has other deficiencies. There are numerous different algorithms in the field of machine learning, and the non-general hardware design is difficult to adapt to all changing algorithms. For example, in embedded devices as mobile phones, we need machine learning algorithms to achieve different functions, such as speech recognition and face recognition. It is difficult to provide a separate ASIC chip for each algorithm. Another difficulty is that the algorithm for machine learning is updated extremely frequently, but ASIC is developed slowly. The ASIC chip for the algorithm has not been completed, and the algorithm is out of date, which is highly likely to happen. For example, at the beginning of my PhD project, YOLO version 2 (YOLOv2) was the latest, but when I wrote a doctoral thesis, YOLO version 4 (YOLOv4) was already available.

The high development cost and inflexibility of ASIC make it not widely used in the field of machine learning.

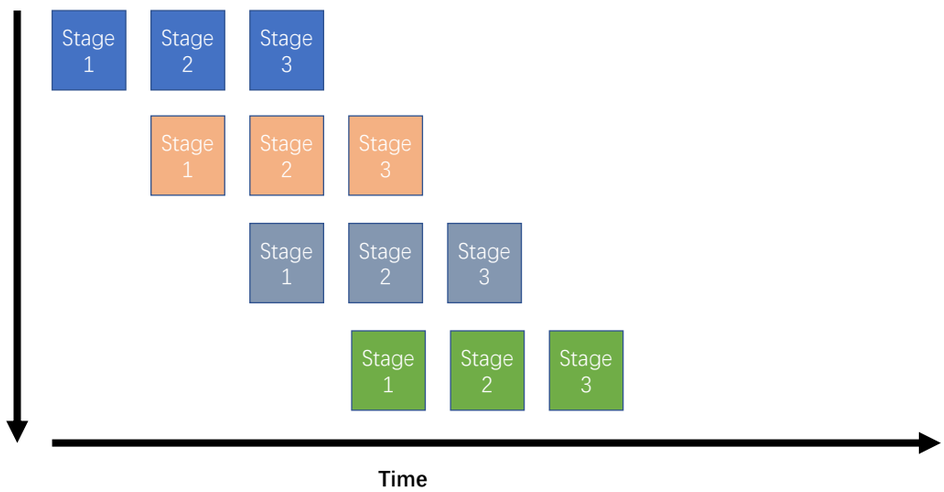
However, it should be noted that what we introduced here is the general situation of ASIC. We should also see some excellent ASICs architecture for CNNs, such as Google's TPU [10] and Eyeriss framework [7]. With the development of ASIC, the mentioned difficulties may gradually be overcome, and a more low-power, high-performance ASIC platform may be proposed to the field of machine learning.

FPGA: Rising Star of CNN Accelerators

The specific hardware can speed up calculations and reduce energy consummation. But fixed circuits such as ASICs are not flexible, which makes them not well used for CNN. We need accelerators which can not only consider the high efficiency and low power consumption as ASIC but also be dynamically updated. Field-Programmable Gate Arrays (FPGAs) are an integrated circuits designed to be able to be reprogrammed to the desired application or functionality requirements after manufacturing. Figure 3.4 shows the mesh FPGA architecture where the matrix of configurable logic blocks (CLBs) can be routed by configurable switch to realize target functions. The components integrated in FPGA are introduced in detail in section 3.3.2. It is observed that the calculation circuits are directly compiled and



(a) Tasks executed without pipeline



(b) Tasks executed with pipeline

Figure 3.5: Pipeline Technology: Pipelining is a form of parallelization in which multiple iterations of a repeated execute concurrently, like an assembly line. Consider the basic processing with three stages and three iterations, shown in Figure 3.5a. A stage is defined as the operations that occur in the repeated execution within one clock cycle. If each stage of this repeated takes one clock cycle to execute, then this loop has a latency of nine cycles. The Figure 3.5b shows the pipelining of the executions from the basic processing. The pipelined processing has a latency of five clock cycles for three iterations (and six cycles for four iterations), But there is no area trade-off. In the example of Figure 3.5b, during the second clock cycle, the calculator for Stage 1 has finished the iteration 1 and it is processing iteration 2. At this clock cycle, the calculator for Stage 2 is processing iteration 1, and Stage 3 is inactive. This repeated processing is pipelined with an initiation interval (II) of 1. An II of 1 means that there is a delay of 1 clock cycle between the beginning of each successive loop iteration. We can estimate the overall latency of the repeated execution of N iterations with the following equation:

$$Latency_e = (N-1) * II + Lantency_i \tag{3.2}$$

where $Latency_e$ is the number of cycles the executions and $Latency_i$ is the number of cycles a single iterations takes to execute.

generated according to the algorithms, that is directly used for calculation, without reading instructions during execution like CPUs or GPUs. Therefore, FPGAs are able to as efficient as integrated circuits, as well, it is flexible due to its ability to be reprogrammed. In fact, due to its efficiency and flexibility, FPGA have already played an important role in many fields, such as communication, control engineering, mathematics, cryptography, and robotic [67, 68]. With the rapid development of machine learning, FPGA has gradually attracted the attention of researchers in this field, and the combination of CPU + FPGA provides new ideas for machine learning.

Another primary benefit of using an FPGA instead of a CPU is that FPGAs use a spatial computing structure. A design can use additional hardware resources in exchange for lower latency, which can take advantage of the spatial compute structure to accelerate the loops by having multiple iterations of a loop executing concurrently, that is, unroll a loop execution. Since the generated hardware computing resources can be reused multiple times, in addition to parallel, FPGA is more convenient to use pipeline technology to accelerate processing, as shown in Figure 3.5. Unlike unrolling computing, pipeline technology just rearranges the allocation of computing resources without requiring more area.

Comparison of Different Accelerators

A few works such as [69, 70] made a quantitative comparison of different accelerators. We present the usual conclusions here. For chips with the same specifications, such as manufacturing process and area cost, three aspects are evaluated:

- In terms of computing capacity, FPGAs are better than CPUs but inferior to GPUs. This means that for the same task, the calculation in FPGA is faster than that in CPU but slower than in GPU.
- In terms of energy power, FPGAs consumed more energy than CPUs but less than GPUs per unit time.
- But considering both energy consumption and computing capacity, performance per watt for FPGA is better than GPU and CPU. In other words, FPGAs are more energy efficient than CPU and GPU.

While CPU + GPU is the mainstream architecture in machine learning research, considering the issue of energy consumption, FPGA is a platform that deserves more investment. For large data centers or large-scale cloud servers, FPGA can significantly reduce energy consumption, which makes operation and maintenance costs lower. Therefore, some famous PaaS and IaaS companies, such as Microsoft, Amazon EC2, as well as Alibaba and Baidu in China are increasingly focusing on FPGAs. Another field that FPGAs have received much attention is embedded systems. Low power computing is always preferred for systems with limited energy.

The experiment described in this Ph.D thesis is based on the above concept. We use GPU as a training platform since it is more convenient for development. But when the neural networks are well trained, they should be deployed into an embedded system. The FPGA platform which is more energy effective is more favorable in our works.

3.3.2 FPGA Platform and Development Tools

As mentioned in section 3.3.1, it is a common practice to use GPU to run machine learning algorithms, and many frameworks also provide the interface for GPU compilation and execution. Therefore, the development with GPU is simple, and we will not introduce more detail here. Relatively speaking, the FPGA development process is more cumbersome. For different manufacturers, the architecture and development processes of FPGA are also different. They are discussed in this section.

Ideally, there are a couple of FPGA vendors. However, most of the market share on FPGAs is divided between Xilinx and Altra. Xilinx is an American technology company, primarily a supplier of programmable logic devices. It is known for inventing the field-programmable gate array (FPGA). Altera Corporation is a Silicon Valley manufacturer of PLDs, reconfigurable complex digital circuits. Since all proposed approaches in this thesis are realized targeting Xilinx FPGAs and tools, this thesis will introduce Xilinx FPGA and the design environment in more detail.

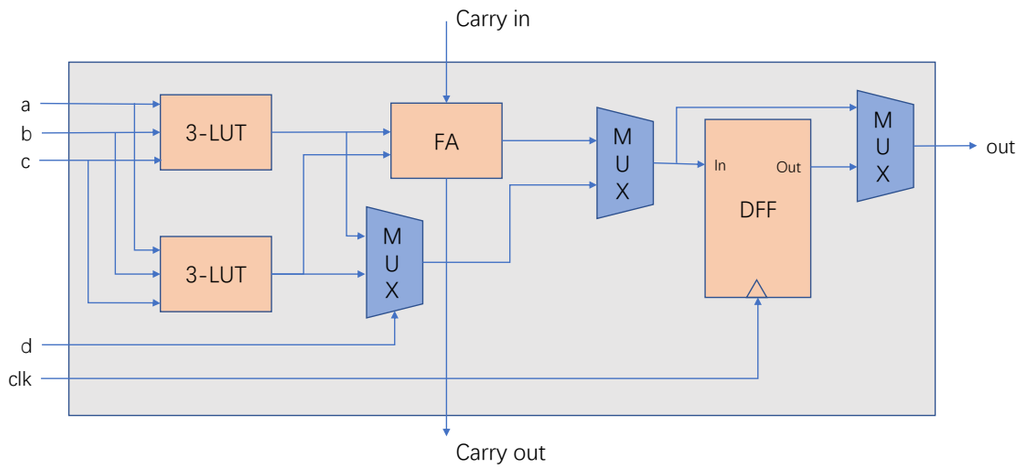


Figure 3.6: The architecture of Configurable Logic Blocks

FPGA Reconfiguration

FPGAs are reprogrammed by generating the configuration bitstream corresponding to the functionality implemented by the user. Ideally, configuration bitstream can be stored in FPGA using various technologies. However, most FPGAs are based on RAM (SRAM). Generally, FPGA reconfigurations are divided as total reconfiguration and partial reconfiguration. In total reconfiguration, the configuration bitstream, containing the FPGA configuration data, provides the information regarding the complete chip and it configures the entire FPGA. In partial reconfiguration, only a portion of the device is reconfigured, while the rest of the hardware mapped on the FPGA can continue to operate transparently with respect to the reconfiguration process. When the reconfiguration is performed while part of the FPGA is running, then it is called run-time reconfiguration.

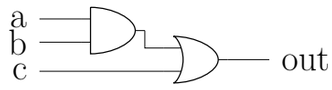
SRAM-based FPGAs SRAM-based FPGA stores logic cell configuration data in the static memory (organized as an array of latches). Since SRAM is volatile and cannot keep data without a power source, such FPGAs must be programmed (configured) upon start. There are two basic modes of programming:

- **Master mode**: when FPGA reads configuration data from an external source, such as an external Flash memory chip.
- **Slave mode**: when FPGA is configured by an external master device, such as a processor. This can be usually done via a dedicated configuration interface or a boundary-scan (JTAG) interface

SRAM-based FPGAs include most chips of Virtex and Spartan families designed by Xilinx, as well as Stratix and Cyclone families designed by Altera.

Configurable Logic Blocks Architecture The configurable logic blocks (CLB) is the backbone of the FPGA to store combinational and sequential functions. By turning on some of the configurable switches within a configurable switch box, shown in Figure 3.4, more complicated computing structure composited of CLBs and other resources can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

A classic CLB consists of m-input Look-Up Tables (LUTs), a full adder (FA), and a D-type flip-flop (DFF) as shown in Figure 3.6. The LUTs are in this figure split into two 3-input LUTs. In normal mode, those are combined into a 4-input LUT through the left mux. In arithmetic mode, their outputs are fed to the FA. The mode selection is programmed into the middle multiplexer. The output can be either synchronous or asynchronous, depending on the programming of the mux to the right, in the figure example. In practice, entire or parts of the FA are put as functions into the LUTs to save space.



(a) Logic operation with 3 inputs

c	b	a	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(b) Truth table

Figure 3.7: A 3-input logic operation

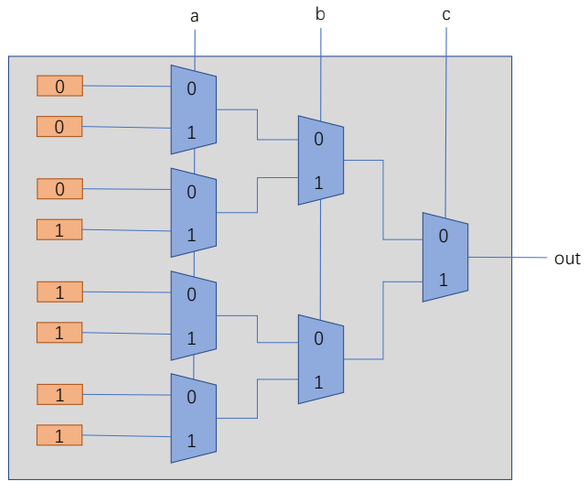


Figure 3.8: A 3-input LUT for the logic operation

Look-Up Table The Look-Up Table (LUT) is used to store the possible outputs of the combinational function for the specified inputs. Then, the inputs will be used as a multiplexer to select the right output during run-time. In general, a LUT is basically a table that determines the output of any given input(s). In the context of combinational logic, it is the truth table. This truth table effectively defines how the combinatorial logic behaves. For example, the logic circuit in Figure 3.7a can be implemented using 3-input LUT. First, the truth table of the circuit is calculated as shown in Figure 3.7b. Then, the 3-input LUT for the circuit will be configured as shown in Figure 3.8.

In fact, the number of input of LUT is different for different FPGA designs, as well as different vendors. With the development of technology, manufacturers have moved to 6-input LUTs in their high-performance parts, claiming increased performance. However, We would like to emphasize is that even the available LUT is m-input, it is possible to implement any combinational function with less than m inputs as well as any combinational logic function with greater than m inputs. When the combinational logic function has fewer inputs than the inputs of the LUT, the higher input will be set to zero and only the fraction of the LUT will be used.

The lookup tables can also be customized based on the requirement. For example, a table built for a complex mathematical function may work much faster than calculating the value by following an algorithm. The table should be stored in RAM or ROM. This brings us to viewing the LUTs simply as memory, where the inputs are the address, and the corresponding outputs are the data stored in the given address.

Other Resources Modern FPGA families expand upon the above capabilities to include higher-level functionality fixed into the silicon. Embedding these common function in silicon reduces the required area and improves the speed of operation of these function. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic, and embedded memories. These cores exist alongside the programmable fabric, but they are built out of transistors instead of LUTs. So they have ASIC level performance and power consumption while not consuming a significant amount of fabric resources, leaving more of the fabric free for the application-specific logic.

In this section, a brief introduction to FPGA is to explain the working mechanism of FPGA, so as to better understand its advantages and disadvantages in machine learning tasks. This is also a key factor in the choice of platform for the follow-up work. For more detailed FPGA structure, see [71].

Xilinx FPGA Design Flow

This section provides an overview of working with the Vivado® Design Suite to create a new design for programming into a Xilinx® device. It provides a brief description of various use

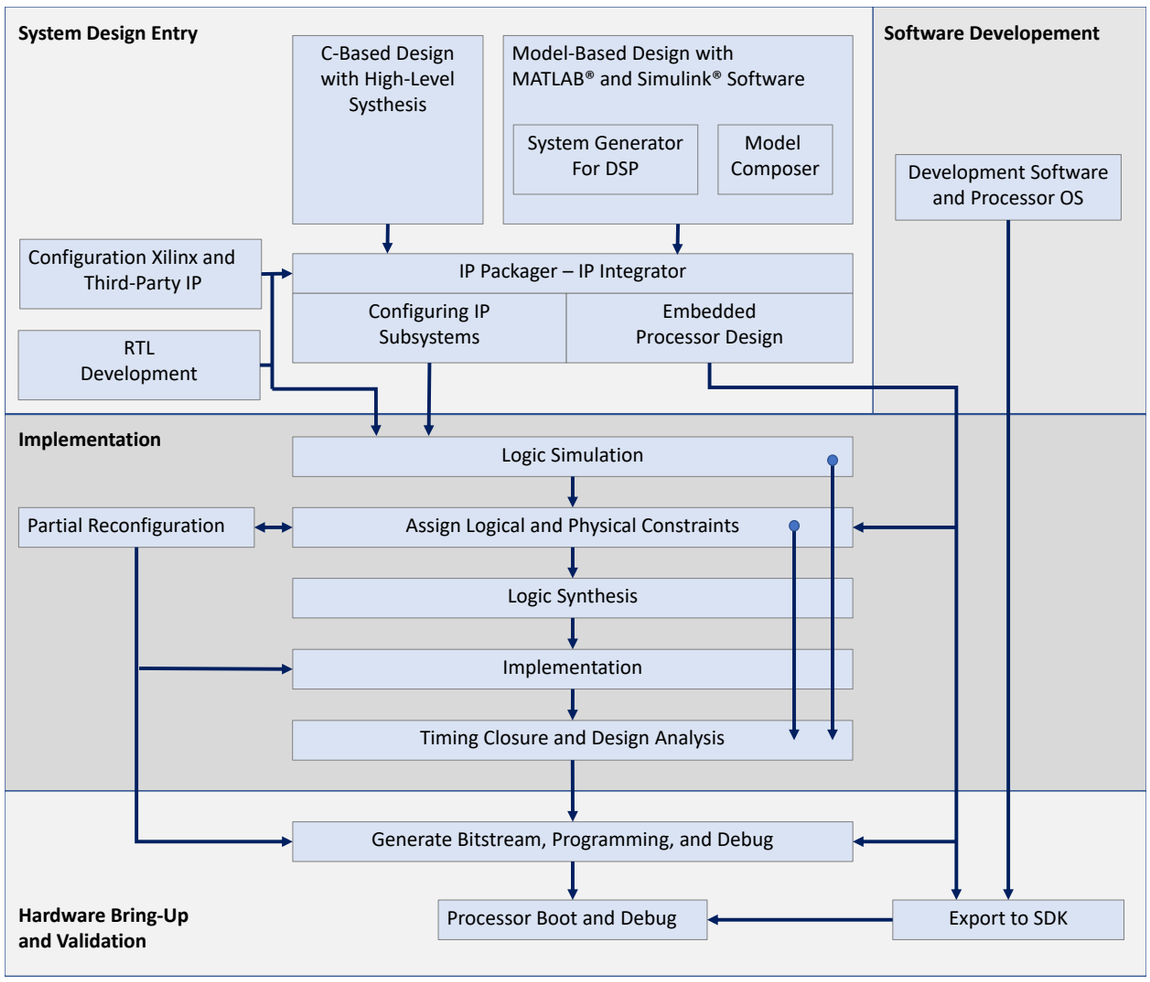


Figure 3.9: Vivado Design Suite High-Level Design Flow

models, design features, and tool options, including preparing, implementing, and managing the design sources and intellectual property (IP) cores.

The Vivado Design Suite offers multiple ways to accomplish the tasks involved in Xilinx FPGA design, implementation, and verification. In addition to the traditional register transfer level (RTL)-to-bitstream FPGA design flow, the Vivado Design Suite provides system level integration flows that focus on intellectual property (IP)-centric design and C-based design. IP can be instantiated, configured, and interactively connected into IP subsystem block designs within the Vivado IP integrator environment. Custom IP and IP block designs can be configured and packaged and made available from the Vivado IP catalog. High-level Synthesis can be leveraged to quickly create and validate complex algorithms in C/C++, synthesize them into RTL and process them through the traditional Vivado RTL flow. Design analysis and verification are enabled at each stage of the flow. Design analysis features include logic simulation, I/O and clock planning, power analysis, constraint definition and timing analysis, design rule checks (DRC), visualization of design logic, analysis and modification of implementation results, programming, and debugging. Figure 3.9 shows the high-level design flow in the Vivado Design Suite, followed by some explications:

- **RTL Design**: RTL source files can be specified to create a project and use these sources for RTL code development, analysis, synthesis, and implementation. Xilinx supplies a library of recommended RTL and constraint templates to ensure RTL and XDC are formed optimally for use with the Vivado Design Suite. Vivado synthesis and implementation support multiple source file types, including Verilog, VHDL, SystemVerilog, and XDC.

- **IP Design and System-Level Design Integration**: The Vivado Design Suite provides an environment to configure, implement, verify, and integrate IP as a standalone module or within the context of the system-level design. IP can include logic, embedded processors, digital signal processing (DSP) modules, or C-based DSP algorithm designs. Custom IP is packaged following the IP-XACT protocol and then made avail-

able through the Vivado IP catalog. The IP catalog provides a quick access to the IP for configuration, instantiation, and validation of IP. Xilinx IP utilizes the AXI4 interconnect standard to enable faster system-level integration. Existing IP can be used in the design either in RTL or netlist format.

- **IP Subsystem Design**: The Vivado IP Integrator environment enables to stitch together various IP into IP subsystems using the AMBA AXI4 interconnect protocol. IP can by interactively configured and connected though a block design style interface and easily connect entire interfaces by drawing DRC-correct connections similar to a schematic. Connecting the IP using standard interfaces saves time over traditional RTL-based connectivity. Connection automation is provided as well as a set of DRCs to ensure proper IP configuration and connectivity. These IP block designs are then validated, packaged, and treated as a single design source. Block designs can be used in a design project or shared in other projects. The IP Integrator environment is the main interface for embedded design and the Xilinx evaluation board interface.

- **I/O and Clock Planning**: The Vivado IDE provides an I/O pin planning environment that enables I/O port assignment either onto specific device package pins or onto internal die pads, and provides tables to let design and analyze package and I/O-related data. Memory interfaces can be assigned interactively into specific I/O banks for optimal data flow. The device and design-related I/O data can be analyzed by using the views and tables available in the Vivado pin planner. The tool also provides I/O DRC and simultaneous switching noise (SSN) analysis commands to validate the I/O assignments.

- **Xilinx Platform Board Support**: In the Vivado Design Suite, an existing Xilinx evaluation platform board can be selected as a target for the design. In the platform board flow, all the IP interfaces implemented on the target board are exposed to enable quick selection and configuration of the IP used in the design. The resulting IP configuration parameters and physical board constraints, such as I/O standard and package pin constraints, are automatically assigned and proliferated throughout the flow. Connection automation enables quick connections to the selected IP.

- **Synthesis**: Vivado synthesis performs a global, or top-down synthesis of the overall RTL design. However, by default, the Vivado Design Suite uses an out-of-context (OOC), or bottom-up design flow to synthesize IP cores from the Xilinx IP Catalog and block designs from the Vivado IP integrator. It can also be chosen to synthesize specific modules of a hierarchical RTL design as OOC modules. This OOC flow lets synthesize, implement, and analyze design modules of a hierarchical design, IP cores, or block designs, out of the context of, or independent of the top-level design. The OOC synthesized netlist is stored and used during top-level implementation to preserve results and reduce runtime. The OOC flow is an efficient technique for supporting hierarchical team design, synthesizing and implementing IP and IP subsystems, and managing modules of large complex designs.

- **Design Analysis and Simulation**: The Vivado Design Suite analyzes, verifies, and modifies the design at each stage of the design process. It can run design rule and design methodology checks, logic simulation, timing, and power analysis to improve circuit performance. This analysis can be run after RTL elaboration, synthesis, and implementation. The Vivado simulator enables to run behavioral and structural logic simulations of the design at different stages of the design flow. The simulator supports Verilog and VHDL mixed-mode simulation, and results can be displayed in a waveform viewer integrated into the Vivado IDE. Third-party simulators can also be used to be integrated into and launched from the Vivado IDE.

- **Placement and Routing**: When the synthesized netlist is available, Vivado implementation provides all the features necessary to optimize, place, and route the netlist onto the available device resources of the target part. Vivado implementation works to satisfy the logical, physical, and timing constraints of the design. For challenging designs, the Vivado IDE also provides advanced floorplanning capabilities to help drive improved implementation results. These include the ability to constrain specific logic

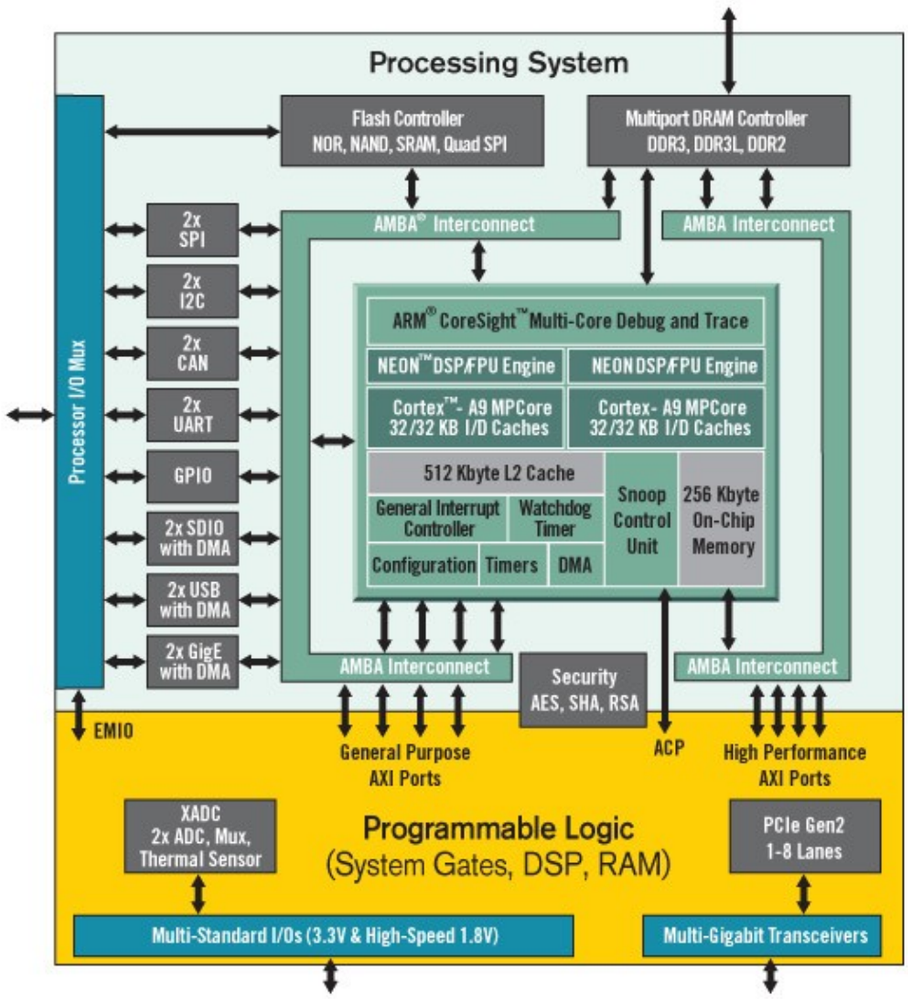


Figure 3.10: Zynq APSoC architecture

into a particular area, or manually placing specific design elements and fixing them for subsequent implementation runs.

- **Hardware Debug and Validation**: Once implementation, the device can be programmed, then analyzed using a Vivado logical analyzer, or within the standalone Vivado Lab Edition environment. Debug signals can be identified in the RTL design, or inserted after synthesis and are processed throughout the flow. Debug cores can be configured and inserted either in RTL, in the synthesized netlist, or in the implemented design using incremental implementation techniques. The nets connected is also replaceable for a debug probe or route internal signals to a package pin for external probing using the Engineering Change Order (ECO) flow.

We have only briefly touched on some of the main steps of Vivado Design Suite and design flow. For a more detailed discussion of the design process, see [72].

PNYQ Framework and PYNQ-Z1

Xilinx® makes Zynq® and Zynq Ultrascale+^{TM} devices, a class of programmable System on Chip (SoC) which integrates a multi-core processor (Dual-core ARM® Cortex®-A9 or Quad-core ARM® Cortex®-A53) and a Field Programmable Gate Array (FPGA) into a single integrated circuit. FPGA, or programmable logic, and microprocessors are complementary technologies for the embedded systems. Each meets distinct requirements for embedded systems that the other cannot perform as well.

Two circuit boards, PYNQ-Z1 integrated Zynq devices, and ZCU102 integrated Zynq Ultrascale+^{TM} devices are used in our work. These two development boards can be deployed by the PYNQ framework. The introduction to them is shown in this section.

PYNQ Framework The primary aim of Python Productivity for Zynq (PYNQ) is to make it easier for designers of embedded systems to exploit the unique benefits of Xilinx

devices in their applications. Specifically, PYNQ enables architects, engineers, and programmers who design embedded systems to use Zynq devices, without having to use ASIC-style design tools to design programmable logic circuits.

PYNQ achieves this goal in three ways:

- Programmable logic circuits are presented as hardware libraries called overlays. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. Overlay can be accessed through an application programming interface (API). Creating a new overlay still requires engineers with expertise in designing programmable logic circuits. The key difference, however, is in building once and reusing the paradigm many times. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

- PYNQ uses Python for programming both the embedded processors and the overlays. Python is a "productivity-level" language. To date, C or C++ is the most common, embedded programming language. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually exclusive choices. PYNQ uses CPython which is written in C, and integrates thousands of C libraries, so that can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used. And whenever efficiency dictates, lower-level C code can be used.

- PYNQ is an open-source project that aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. We incorporate the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of the Zynq device. The web server proxy access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors, and Jupyter notebooks. Browser tools are implemented using a combination of JavaScript, HTML and CSS and run on any modern browser.

PYNQ-Z1 The PYNQ-Z1 board is designed to be used with PYNQ framework, which enables embedded programmers to exploit the capabilities of Xilinx Zynq All Programmable SoCs (APSoCs) without having to design programmable logic circuits. Programmable logic circuits are imported as hardware libraries and programmed through their APIs, basically in the same way as software libraries are imported and programmed. To be mentioned, the PYNQ-Z1 board is the first hardware platform for the PYNQ open-source framework.

The Zynq APSoC is divided into two distinct subsystems: The Processing System (PS) and the Programmable Logic (PL). Figure 3.10 shows an overview of the Zynq APSoC architecture, with the PS colored light green and the PL in yellow. Note that the PCIe Gen2 controller and Multi-gigabit transceivers are not available on the Zynq-7020 device.

The PL is nearly identical to a Xilinx 7-series Artix FPGA, except that it contains several dedicated ports and buses that tightly couple it to the PS. The PL also does not contain the same configuration hardware as a typical 7-series FPGA, and it must be configured either directly by the processor or via the JTAG port.

The PS consists of many components, including the Application Processing Unit (APU, which includes 2 Cortex-A9 processors), Advanced Microcontroller Bus Architecture (AMBA) Interconnect, DDR3 Memory controller, and various peripheral controllers with their inputs and outputs multiplexed to 54 dedicated pins (called Multiplexed I/O, or MIO pins). Peripheral controllers that do not have their inputs and outputs connected to MIO pins can instead route their I/O through the PL, via the Extended-MIO (EMIO) interface. The peripheral controllers are connected to the processors as slaves via the AMBA interconnect and contain readable/writable control registers that are addressable in the processors' memory space. The programmable logic is also connected to the interconnect as a slave, and designs can implement multiple cores in the FPGA fabric that each also contains addressable control registers. Furthermore, cores implemented in the PL can trigger interrupts to the processors (connections not shown in Fig. 3) and perform DMA accesses to DDR3 memory.

The software running on the Arm®-A9 CPUs include a web server hosting the Jupyter notebooks design environment, the IPython kernel and packages, Linux, and a base hardware library and API for the FPGA devices.

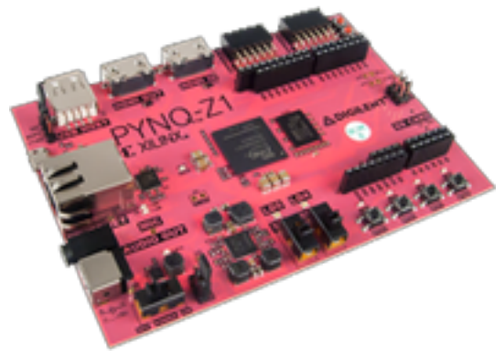


Figure 3.11: PYNQ-Z1

The PYNQ-Z1 natively supports multi-media applications with on-board audio and video interfaces. It is designed to be easily extensible with Pmod, Arduino, and Grove peripherals, as well as general-purpose IO pins.

The PYNQ-Z1 board can be also expanded with USB peripherals including Wi-Fi, Bluetooth, and Webcams.

Next, we enumerate the main interfaces and blocks integrated into PYNQ-Z1. That helps us to understand this development board fully, although we have not used some of them in the next experiments.

- PYNQ XC7Z020-1CLG400C:
  - 650MHz dual-core Cortex-A9 processor
  - DDR3 memory controller with 8 DMA channels and 4 high performance AXI3 slave ports
  - High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
  - Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
  - Programmable from JTAG, Quad-SPI flash, and microSD card
  - Artix-7 family programmable logic:
    - 13,300 logic slices, each with four 6-input LUTs and 8 flip-flops
    - 630 KB of fast block RAM
    - 4 clock management tiles, each with a phase-locked loop (PLL) and mixed-mode clock manager (MMCM)
    - 220 DSP slices
    - On-chip analog-to-digital converter (XADC)
- Memory:
  - 512MB DDR3 with 16-bit bus @ 1050Mbps
  - 16MB Quad-SPI Flash with factory programmed globally unique identifier (48-bit EUI-48/64^{TM} compatible).
  - MicroSD slot
- Power:
  - Powered from USB or any 7V-15V source
- USB and Ethernet:
  - USB-JTAG Programming circuitry
  - USB-UART bridge
  - USB OTG PHY (supports host only)
  - Gigabit Ethernet PHY
- Audio and Video:
  - Electret microphone with pulse density modulated (PDM) output

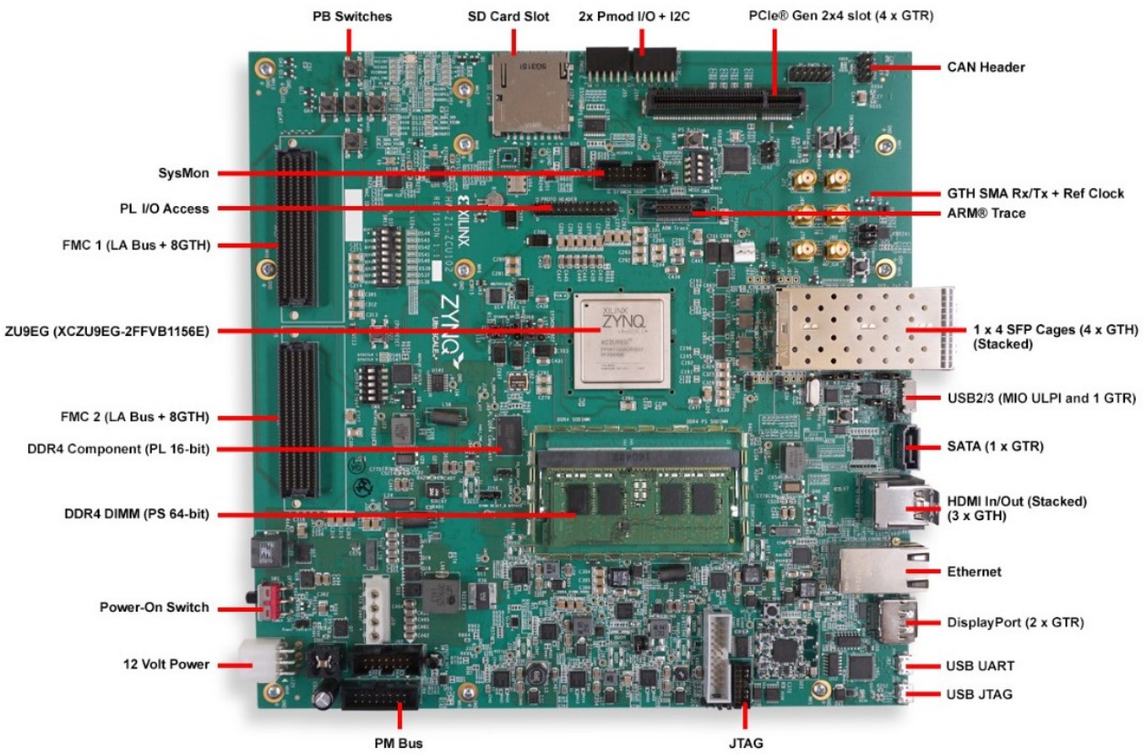


Figure 3.12: The devices and interfaces in ZCU102 board

  - 3.5mm mono audio output jack, pulse-width modulated (PWM) format
  - HDMI sink port (input)
  - HDMI source port (output)

- Switches, push-buttons, and LEDs:
  - 4 push-buttons
  - 2 slide switches
  - 4 LEDs
  - 2 RGB LEDs

- Expansion Connectors:
  - Two standard Pmod ports
    - 16 Total FPGA I/O
  - Arduino/chipKIT Shield connector
    - 49 Total FPGA I/O
    - 6 Single-ended 0-3.3V Analog inputs to XADC
    - 4 Differential 0-1.0V Analog inputs to XADC

We have only briefly mentioned some of the most important characters of PYNQ-Z1. A more detailed introduction regarding the board is provided in [73].

Zynq UltraScale+ MPSoC ZCU102 The ZCU102 Evaluation Kit enables designers to jumpstart designs for automotive, industrial, video, and communications applications. This kit features a Zynq® UltraScale+^{TM} MPSoC with a quad-core Arm® Cortex®-A53, dual-core Cortex-R5F real-time processors, and a Mali^{TM}-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric. The ZCU102 supports all major peripherals and interfaces, enabling development for a wide range of applications.

The FPGA chip *Zynq UltraScale+ XCZU9EG-2FFVB1156 MPSoC* is integrated into this board. The detailed features of this chip are as follows:

- 600k System Logic Cells

- 32.1Mb Memory on Chip
- 2,520 DSP Slices
- 328 usable I/0 Pins.

In the experiments in this Ph.D thesis, we use Vivado and Vivado High Level Syntheses for hardware design, and to generate bitstream finally. Next, bitstream will be loaded into PYNQ-Z1 or ZCU102 and be programmed into FPGA device. The programmable logic part receives data through the interface of shared memory, calculates, and returns the calculation result. The programmable logic part can be called as APIs through the PYNQ framework.

3.4 Summary

In this chapter, we discussed the difficulties encountered with the development of convolutional neural networks. And proposing solutions to these difficulties is the motivation of the next work.

However, due to the widespread use of convolutional neural networks, it is impossible to analyze all scenarios. Hence, we choose the scope of our work in this chapter, which is focusing on the field of computer vision. Furthermore, by discussing the basic tasks of computer vision, we determined image classification and object recognition as benchmark tests for the next works. Based on this, we introduced the commonly used evaluation criteria for these two tasks to build benchmark tests.

Next, the commonly used accelerators for convolutional neural networks are introduced. In addition, and the platforms for our work are also selected. The convolutional neural network is usually trained by CPU+GPU structure. At the same time, the architecture of CPU+FPGA is more favored during deployment and testing. In the end, we discussed the FPGA structure and development process, which helped us better solve the problem.

To sum up, this chapter puts forward the work motivation, determines the work scope and work methods. These set the stage for the next chapters.

Chapter 4

The State-of-the-Art of CNN Compression and Acceleration

CNN-based systems are usually resource-consuming and computational-intensive. When they are integrated into the resource-limited or real-time system, the challenges arise: on one hand, the huge amount of calculation will prolong the calculation time, resulting in some real-time scenarios are not applicable to CNNs. On the other hand, the large volume of weights of the CNNs will also bring challenges to some embedded systems with limited memory. Therefore, we need to compress them, by reducing the size of weight of the CNN, as well as by reducing the amount of calculation.

In this chapter, frequently used methods to compress the CNNs and speed up the calculation are introduced.

4.1 Compact Design

When we talk about compression of CNNs, we often mention many "advanced" methods, such as sparse matrix, or quantization, which will be introduced in the following section. But we often overlook the simplest method, which is to use a simpler design.

4.1.1 The Shallower Deep Neural Networks

Typically, the designers experiment to find out how many hidden layers to use in deep CNNs. Generally speaking, deeper convolutional layers bring higher accuracy. In fact, since the ResNet [47] was proposed, deeper and deeper networks have been used. But these deep networks bring heavy calculations. It is possible to make some adjustments for some special scenarios. For example, if we need a time-sensitive but error-tolerant system, such as a real-time gaming system, or we need more energy-efficient embedded equipment that requires less computation, we can use a shallower CNNs to reduce the amount of calculation and accelerate the system. As well, the same as the number of layers in CNNs, the number of kernels in each convolutional layer can also adjust as needed.

In fact, tiny-YOLO presented in section 2.3.6 is a compressed version of YOLOv2 presented in section 2.3.6. It takes 9 convolutional layers instead of 24 layers as YOLOv2. There is no doubt that this greatly reduces the amount of calculation. Some well-known networks such as VGG [41] also proposed varied depth networks. For example, VGG-16 is composited of 16 layers, but VGG-19 has 19 layers. Obviously, a fewer number of layers reduces the volume of the neural network. But in a general way, it also reduces the accuracy of the results.

We need to admit that this method is essentially a trade-off between precision and computation. For many systems requiring high accuracy, this method is not applicable.

4.1.2 The Smaller Convolutional Kernels

In addition to the number of the convolutional kernels, the size of convolutional kernels in each layer can also be adjusted. As a rule, large convolution kernels bring wider fields of view, so that some networks such as the work [74] use convolution kernels with different sizes to find different patterns. However, the larger convolutional filters often take more resources

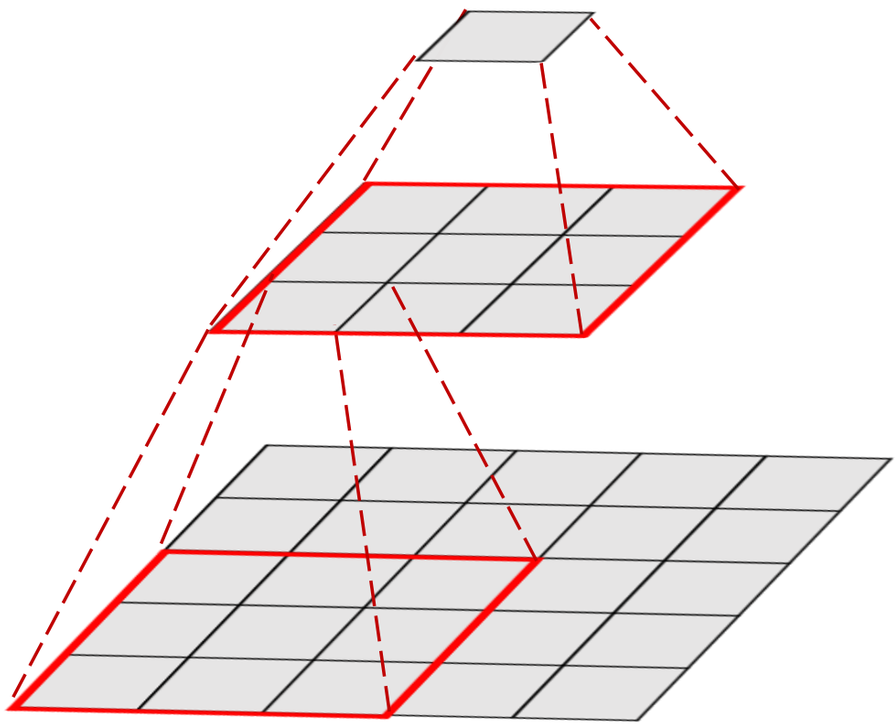


Figure 4.1: The fields of view of the 5×5 kernel replaced by 3×3 kernels

to compute. In order to reduce the amount of calculation, smaller but deeper convolution kernels can be used to replace the larger kernel.

The VGG network [47] used 3×3 kernels to replace the 7×7 kernel in AlexNet. As shown in Figure 4.1, two 3×3 kernels can get the same fields of view as one 5×5 kernel, but only 18 parameters need to be saved, instead of 25 in a 5×5 kernel. Similarly, three 3×3 kernels can be used to replace one 7×7 kernel, but 1.8x compression is achieved. For the Inception V2 proposed in the work [43], it is also introduced to use multiple 3×3 small convolution kernels to replace larger convolution kernels, which significantly reduces the amount of calculation, but the calculation accuracy does not decrease. Furthermore, the Inception V3 proposed in the work [44] also used multiple smaller kernels to replace the larger one. In this work, the combination of 1×3 and 3×1 kernel is used to replace the 3×3 kernel. Interestingly, the smaller kernels with size 1×1 are used to replace some of the 3×3 weights in SqueezeNet [45]. Although the receptive field of some weights has become smaller, the final accuracy in SqueezeNet has not changed significantly.

4.2 Quantization

In general, in fully numerical precision CNN, the data for convolution/multiplication is represented as a 32 bits or 64 bits float-point. However, the higher-precision data will require more memory to store and more bandwidth to transmit. For example, the memory to store a 64-bit floating-point variable can save eight 8-bit integer variables. If we can use lower-precision data for calculations, then we will also reduce the overall size of the network, and for communication channels with the same bandwidth, more data can be transmitted. Meanwhile, for some specific devices, the lower-precision multiplication calculator may use fewer resources, and the calculation will spend less time.

Quantization is a major approach to reduce the numerical precision of CNNs. Through quantization, fewer bits are used to represent high-precision values to achieve the compression of CNN. This section describes quantification approaches.

4.2.1 Using Fixed-point instead of Floating-point

Figure 4.2a shows the format of a general single-precision floating-point variable in IEEE standard 754 [75, 76], where the first bit [31:31] represents the sign, the bits [30:23] encode

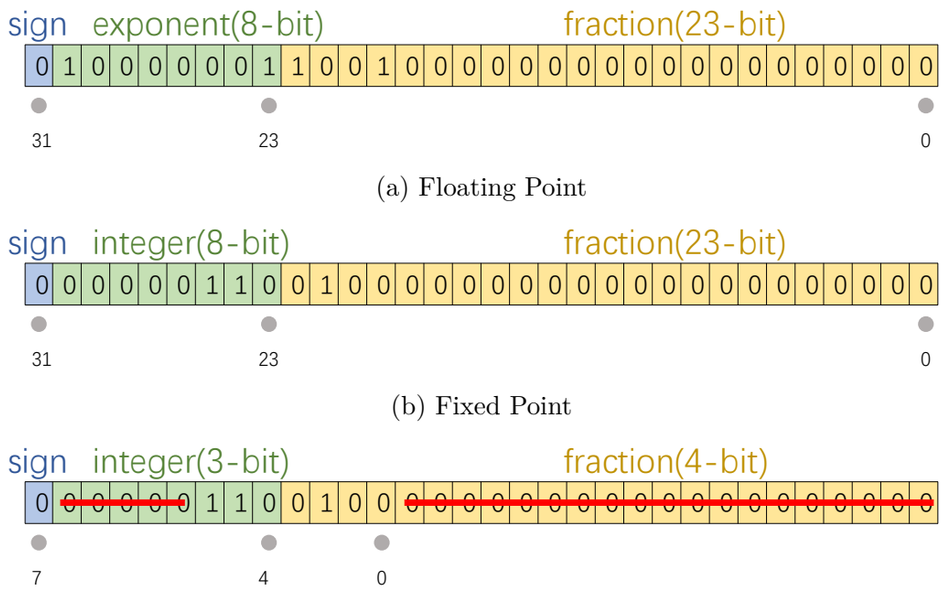


Figure 4.2: The formats of decimals in computers.

the exponent, and the bits [22:0] encode the faction. This expression is similar to scientific notation but uses 2 as the base. Then, the result shown in the Figure 4.2a is calculated as:

- $sign = 0, (+)$
- $fraction = 1 + (2)^{-1} + (2)^{-4} = 1.5625$
- $exponent = -127 + \{10000001\}_{\{2\}} = 2$
- $result = (+1) \times 1.5625 \times 2^2 = 6.25$

Depending on the precision, the floating-point type can also be encoded with 64-bit or 16-bit, which is called double-precision floating-point and half-precision floating-point respectively.

When fewer bits is used to express parameters, fixed-point numbers is more convenient. Figure 4.2b shows the format of a general fixed-point variable. This fixed point number is calculated as:

- $sign = 0, (+)$
- $integet = \{00000110\}_{\{2\}} = 6$
- $fraction = (2)^{-2} = 0.25$
- $result = (+1) \times (6 + 0.25) = 6.25$

In general, since the parameters in networks are not exceptionally large, fewer bits are needed to express the integer part in fixed-point numbers. At the same time, in order to reduce the volume of values, less important bits in the fraction part can be discarded, shown as Figure 4.2c.

In essence, this discarding is a trade-off between precision and volume. However, in practice, the limited discarding of bits has little effect on accuracy. Some works such as [77] proposed methods to determine how many bits should be drop off and how to constitute dynamic fixed-point numbers (DFP).

4.2.2 Using Integers to Represent Floating-point

Unlike fixed-point variables which integrates the integer and fraction parts, the parameters can also be scaled directly to integers without fraction parts. Floating-point parameters can then be represented as integers. In terms of precision, a fixed-point variable that uses 3 bits as an integer and 5 bits as a fraction is equivalent to an 8-bit integer variable.

The method that scaling the variable into an integer can be considered as a linear transformation. In this method, two floating-point variables store the minimum and maximum

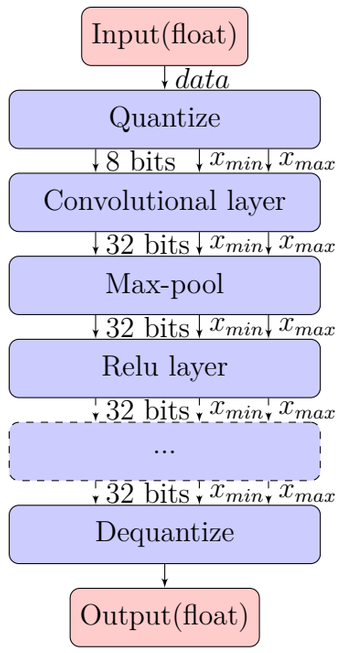


Figure 4.3: In continuous quantization, the quantized output, minimum and maximum float could be used as input for the next layer directly, and the dequantize layer is as the end of all quantized layer.

values of parameters, that correspond to the range of the parameters. Then the floating-point parameters in the ranges are linearly mapped to the quantization interval. An example that quantizes the parameters distributed in [-10.0,30.0] into 8-bit integers whose interval is [0,255] is shown in table 4.1.

Quantized 8 bits	Float 32 bits
0	-10.0 (min)
255	30.0 (max)
128	10.0 (other value)
163	15.5 (other value)

Table 4.1: The representation of float-point numbers

More generally, in the quantization, a float-point is represented as an n bits integer by the followed linear transformation:

$$x_q = (x - x_{min}) \times (\frac{max - min}{x_{max} - x_{min}}) \tag{4.1}$$

where max is the maximum integer expressed by n bits, min is the minimum integer expressed by n bits, x_{max} is the float represented by max, and x_{min} is the float represented by min.

Due to the quantization, the multiplication of floating-point values can be converted to multiplication of integers. We note that

$$\begin{cases} scale &= \dfrac{max - min}{x_{max} - x_{min}} \\ \\ x_0 &= (0 - x_{min}) \times scale \end{cases} \tag{4.2}$$

that shows the quantized value when $x = 0$. Then the multiplication of two float-point values is represented as:

$$prod = x_1 \times x_2 = prod_q / scale_q \tag{4.3}$$

where

$$\begin{cases} prod_q &= (x_{q1} - x_{01}) \times (x_{q2} - x_{02}) \\ \\ scale_q &= scale_1 \times scale_2 \end{cases} \tag{4.4}$$

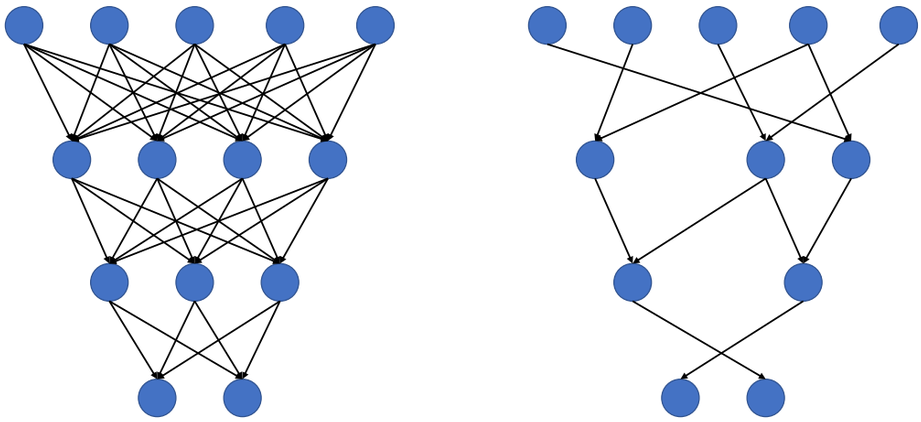


Figure 4.4: Pruning of neural networks

With the definition of operation multiplication, the convolution composited by multiply–accumulate operations can be calculated. To be mentioned that the $scale_q$ is common for all multiplication in one convolutional layer, so that we can calculate only once for one convolutional layer.

Since the $scale_q$ and x_0 are known, for the given minimum input and maximum input, the maximum and minimum of output can be calculated. As Figure 4.3 shown, these parameters can be used as the input of the next quantized layer directly. Hence, for the deep quantized convolutional neural network, the dequantize which generates $prod$ by quantized product $prod_q$ may be only calculated at the output of the quantized network.

Even though the quantize and dequantize layer cost computing resource, they are calculated only once at the beginning or end of the network. And obviously, it is more efficient to multiply 8 bits integers than 32-bit floats. So if the network is deep, this quantization data approach can significantly increase the speed of convolution.

Unlike the fixed-point multiplication which requires the special multipliers, 8-bit integer multipliers are generally integrated on general-purpose chips for integer multiplication. Therefore the quantization by using integers is easier to implement than the quantization by using fixed-point on the general proposed devices. Because of this, the frameworks such as TensorFlow [54] that support GPU execution have supported for the quantization by using 8-bit integers.

4.3 Pruning and Sparse Neural Network

In addition to the compacting network and quantization methods introduced above, pruning networks generating the sparse is another widely discussed method to compress and speed up the neural network. It will be briefly introduced in this chapter.

The main idea of pruning was proposed by some works such as [78, 79, 80] in the 1990s. Since the larger neural networks have been widely used in recent years, pruning methods are also constantly developing. There are many parameters in the large neural network, but some weights do not affect and contribute to the final result, in other words, they are redundant. The basic idea of pruning is to cut out these redundant parameters.

The neural network is usually shown on the left of Figure 4.4: every neuron in the lower layer is connected to the neuron in the upper layer, which means that we must perform a lot of operations. When pruning the neural network, the unimportant connections are deleted. We only need to connect each neuron with several other neurons, as shown in the right of Figure 4.4, which greatly reduces the amount of calculation. This kind of neural network is called a sparse neural network.

In fact, Figure 4.4 is just a schematic of pruning. In practical, there are generally two levels for pruning the convolutional neural network, the structured pruning, and the weight pruning. [81]

4.3.1 Structured Pruning

As introduced in chapter 2.2, there are usually multiple convolution kernels for a certain convolutional layer. Normally, the convolutional kernels are different, and the shapes recognized by different convolution kernels are also different. Different shapes have different importance for the final prediction. Therefore, we can say that different convolution kernels contribute differently to the final result. Then, deleting the convolution kernel that contributes less to the final result can effectively reduce the volume of the neural network.

How to evaluate the contributions of weight is a widely discussed topic. The work [15] discussed the greedy algorithm used to evaluate the quality of each parameter to obtain the optimal solution. This brute force method can get the optimal solution, but it is prohibitively costly to compute. Therefore this work also proposed a new method for ranking neurons based on the first-order Taylor expansion model cost function. The method proposed in [16] is more intuitive. In this work, all the convolutional filters are ranked by the l_1-norm of the weights in the convolution kernel. The convolution filters with the lowest l_1-norm after ranking are discarded to achieve the purpose of pruning. [17] proposed a similar process, rank-then-delete, but with a different index of rank. It uses N particle filters to evaluate the corresponding N convolutional filters. Each particle filter will be assigned a score based on the accuracy of its influence model on the verification dataset, and the convolution kernel with a low score will be discarded to achieve the purpose of pruning. The work [81] also scored to each filter for ranking. But in this work, the scores, named scaling factors, are multiplied to the output of that channel directly. If the score is small, whatever the convolution kernel, the output multiply by the scores is small so that its impact on the final result is also small. In other words, we can evaluate the impact of each kernel during the training, and the network is automatically pruning during the training.

4.3.2 Weight Pruning

Similar to the structured pruning, the weight pruning also discards some unimportant parameters. The difference is that the structured pruning deletes the entire filter, while weight pruning deletes the unimportant connections in each filter.

How to determine the importance of the connections is still an interesting topic. The work [18] proposed to recognize the importance of each connection through training. All connections with importance below a threshold are removed from the network. In [18], there is no guidance for sparsity during training. [19] overcomes this limitation by explicitly imposing sparse constraint over each weight with additional gate variables and achieve high compression rates by pruning connections with zero gate values. The method of discarding the connection whose importance is below the threshold is discussed. Meanwhile, the work [20] discusses how to determine the threshold based on the sensitivity of the connection. These excellent works have provided solutions for weight pruning.

To be mentioned is that for some special networks, they are more sensitive to the size of activations, namely discarding some filters or some connections in the filter will change the output volume, as what happened in work [17], and that will cause trouble for the subsequent calculation. In this case, we can set the discarded convolution kernel or weight value to 0, as done in the works [50, 18]. Some CPU or GPU can skip the multiplication of 0, without calculation, that can also improve the calculation speed of the neural network. In fact, after pruning, the sparse rate (the ratio that weight value is 0) of many deep neural networks is high. Therefore, storage space can be reduced by storing models in a sparse format. This compresses the volume of neural network parameters.

On the other hand, convolution is usually converted into General Matrix Multiply (GeMM) by Image to Matrix method, seen in section 6.3.1. Many libraries such as Basic Linear Algebra Subprograms (BLAS) library proposed by Nvidia provide the optimization for the sparse matrix. That speeds up the calculation of sparse matrices and sparse convolutional layers.

It should be noted that for all the pruning methods introduced in this section, discarding the weight will reduce the accuracy. Therefore, the weights after pruning need to fine-tune. That is, after pruning, it is needed to retrain the network. For some methods, this pruning-training cycle needs to be repeated many iterations.

4.4 Using New Computational Primitives

Quantization and pruning are the most used for improving the performance of CNN. While proposing optimizations for CNNs, researchers have also pointed out that the convolutional layer composed of a large number of MAC is a resource-hungry computational primitive, and easier computational primitives can be used to speed up CNNs. Interestingly, when the fully connected layer is regarded as a resource-consuming calculation, the lighter convolutional layer is a new computational primitive to replace part of the fully connected layer. Now, the convolutional layer is also facing the same problem, so some new computational primitives are proposed. This section takes Hit-or-Miss transform neural networks as example, to show the lighter networks with new computational primitives.

In mathematical morphology, Hit-or-Miss transform is an operation that detects a given pattern in a binary or grayscale image. Let f be a grayscale image, b a structuring element, and $f(x, y)$ the grayscale intensity at a location (x, y). The basic operations of Hit-and-Miss transformation, dilation and erosion (of grayscale image), are defined as:

Definition (Grayscale Dilation [82]) *The grayscale dilation of f by b, denoted as $f \oplus b$, is*

$$(f \oplus b)(x, y) = max\{f(s - x, t - y) + b(x, y) | (s - x), (t - y) \in D_f; (x, y) \in D_b\} \tag{4.5}$$

where D_f and D_b are the domains of f and b, respectively.

Definition (Grayscale Erosion [82]) *The grayscale erosion of f by b, denoted as $f \ominus b$, is*

$$(f \ominus b)(x, y) = min\{f(s + x, t + y) - b(x, y) | (s + x), (t + y) \in D_f; (x, y) \in D_b\} \tag{4.6}$$

where D_f and D_b are the domains of f and b, respectively.

On a side note, there is a parallel between 2-D convolution and dilation/erosion, when sum replaces product and when max/min replaces sum [83]. Based on the dilation and erosion, gray scale hit-or-miss transform is defined as:

Definition (Grayscal Hit-or-Miss transform) *The grayscale hit-or-miss transform is:*

$$f \odot (h, m) = (f \ominus h) - (f \oplus m^r) \tag{4.7}$$

where h is the set associated with the foreground or an object, m is a set associated with the corresponding background, and m^r is the reflection of m, i.e., $m^r(x, y) = m(-x, -y)$.

Let h and m be structuring elements with non-negative weights. The hit and miss structuring elements together define the target pattern with hit indicating the foreground and miss indicating the background. For example, if $h(x, y) > m(x, y)$ at a location (x, y), then that pixel is treated more as foreground than background and vice versa. With Hit-or-Miss transform, the foreground or targeted object can be labeled, just like convolution in convolutional layers. With the Hit-or-Miss transform, a morphology-based neural network is proposed in [83], called Hit-or-Miss transform neural network.

Although the purpose of Hit-or-Miss transform neural network is to find a theoretical explanation of pattern recognition rather than to speed up CNNs, it objectively constructs a deep neural network without convolution. Compared with CNN, only comparison and addition operations are used in Hit-or-Miss neural network, avoiding heavy multiplication operations. Therefore, it can accelerate calculations on certain platforms that do not have multiplication accelerators.

There is no doubt that Hit-or-Miss transform neural network uses a new computational primitive, Hit-or-Miss transform, which makes a large number of multiplications to be replaced, thereby accelerating the network. Unfortunately, in order to improve prediction accuracy, $softmax$ and $softmin$ operations are used to replace max and min in the final version of Hit-or-Miss transform neural network, which may greatly reduce the calculation speed [83]. Meanwhile, Hit-or-Miss transform neural networks have not reached the accuracy of traditional CNNs [83].

In chapter 8, we will also propose a neural network only using comparison and addition, from the perspective of approximate calculation, but its accuracy is higher than that of Hit-or-Miss neural network.

4.5 Others

In addition to the different optimization methods introduced above, there are also some excellent compression and acceleration methods briefly introduced in this section.

4.5.1 Classic Compression Method

When we talk about compression, some classic compression methods can also be used to compress neural networks, for example, the most used lossless compression technology, Huffman coding.

Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol. We can use prefix codes to find the original symbol in the table. The algorithm derives this table from the estimated probability or frequency of occurrence for each possible value of the source symbol, that is, more common symbols are generally represented using fewer bits than less common symbols. In other words, the total length of the message becomes smaller because most words in the message use shorter expressions, videlicet, the message is compressed.

[84] uses the Huffman coding method to compress the neural network. In this work, the pruning network presented in section 4.3 is used to set many weights with small values to 0. Then, the quantization method, presented in section 4.2 is applied to the network. After quantization, the value of weights belongs to in a finite set, instead of an infinite real number set. At this time, the compression problem becomes to compress a given set of symbols (the weights) and their frequency. This is a typical Huffman code problem. Finally, the set of weight values is compiled into a Huffman table according to the frequency of occurrence. This method reduces the size of the weight, and the convolutional neural network is compressed.

This work is one of the most excellent methods proposed in 2015. Although this method has not been widely used in industry, it still shows that some classic compression algorithms in traditional computer science and information theory can play a role in neural network compression

4.5.2 Low-rank Factorization

In a convolutional neural network, the convolutional layers can be converted into matrix multiplication by the Image-to-Matrix method introduced in section 6.3.1, and the computational primitives of the fully connected layer are also matrix multiplication. In that way, the traditional optimization method for matrix multiplication can be used to optimize the calculation of the convolutional neural network. Matrix factorization, which is to decompose a large matrix into multiple small matrices and use these small matrices for calculation, thereby reducing the amount of calculation, is a method for matrix calculation compression and acceleration, as shown in the following:

$$\begin{array}{lll} \text{if} & C & = AB \\ \text{and} & B & = U\Lambda V^T \\ \text{then} & C & = A(U\Lambda V^T) \\ & & = (AU)\Lambda V^T \end{array} \tag{4.8}$$

If we can estimate the parameter matrix with several small matrices, then the output matrix can be obtained by the above formula.

The work [85] proposed to use low-rank matrix decomposition to approximates weight matrix in neural networks with a low-rank matrix using techniques like Singular Value Decomposition (SVD). This method works especially well on fully-connected layers, yielding 3x model-size compression, however without notable speed acceleration, since computing operations on CNN mainly come from convolutional layers. The work [86] also discussed the Canonical Polyadic (CP) decomposition and Batch Normalization (BN) decomposition used to find the low-rank matrix, and compared the compression rate and speed acceleration for a CNN by using these low-rank matrices.

The advantage of the low-rank approximation method is that it does not change the computational primitives in the network and does not require any new operations. The decomposed network is still implemented by convolution operation, so its application is relatively wide. There are various decomposition methods, any matrix or tensor decomposition

method can be used. But generally, the decomposed network needs parameter tuning to ensure the accuracy of the decomposed network model.

The low-rank estimation method has a problem to be solved. For example, it is not clear how many ranks are reserved. Retaining too many ranks can guarantee the accuracy rate, but the acceleration compression effect is not good. Retaining too little ranks, the acceleration compression effect is good, but the accuracy rate is difficult to guarantee. Some works proposed to train a low-rank parameter matrix first, that is, add the evaluation of the parameter matrix rank to the loss function, and then make approximative decomposition of the trained low-rank network. Since many column vectors in the parameter matrix are linear related, so it can keep truly little rank for decomposition.

Other matrix decomposition can also be used to speed up convolutional neural networks. For example, LU decomposition is used to convert the sparse matrix to a triangular matrix, thereby reducing the amount of calculation. These transformation methods for matrices are more about the mathematical direction than the machine learning direction, so we will not introduce them in detail.

4.5.3 Spatial Mapping

Some works try to use fast Fourier transform (FFT) to convert convolution into matrix multiplication, to reduce the amount of calculation. This is an interesting idea. However, this method also brings some difficulties.

- Additional computations for FFT: However, the FFT of weights can be pre-calculated, and the FFT of inputs only needs to be calculated only once, and can be used by all the filters. Therefore, additional computations are not a big challenge for the system.
- Increased memory bandwidth requirements: The complex number generated by the conversion and the larger kernel after the conversion will increase the memory bandwidth requirements.
- The calculation is more complicated: if there is no special calculation unit or instruction, actually a complex multiplication requires four real multiplications and additions.

Since the output of FFT is relatively cost for memory bandwidth and calculation resource, conventional FFT based convolution is cost-effective only for large filters. However, the state-of-the-art convolutional neural networks use small filters, such as 3x3, that means the FFT is not a valuable method.

But similar to the Fourier transform, [87] provides an acceleration algorithm for small convolution kernels. We use an example to introduce the Winograd algorithm.

> Example: Taking a 1-dimensional convolution as an example, the input signal is $d = [d_0, d_1, d_2, d_3]^T$, and the convolution kernel is $g = [g_0, g_1, g_2]^T$, then the convolution can be written as the following matrix multiplication:
>
> $$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} \tag{4.9}$$
>
> For a general matrix multiplication, 6 multiplications and 4 additions is needed:
>
> $$\begin{aligned} r_0 &= (d_0 \cdot g_0) + (d_0 \cdot g_0) + (d_0 \cdot g_0) \\ r_1 &= (d_1 \cdot g_0) + (d_1 \cdot g_0) + (d_1 \cdot g_0) \end{aligned} \tag{4.10}$$
>
> However, the matrix into which the input signal is converted in the convolution operation is not an arbitrary matrix. In this matrix, a great number of repetitive elements are regularly distributed, such as d1 and d2 in the first and second rows. That makes optimization possible. The computation in Winograd algorithm is as following:
>
> $$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m3 \\ m_2 - m3 - m4 \end{bmatrix} \tag{4.11}$$

where

$$\begin{aligned} m1 &= (d_0 - d_2) \cdot g_0 \\ m2 &= (d_1 + d_2) \cdot (\tfrac{g_0+g_1+g_2}{2}) \\ m3 &= (d_2 - d_1) \cdot (\tfrac{g_0-g_1+g_2}{2}) \\ m4 &= (d_1 - d_3) \cdot g_2 \end{aligned} \tag{4.12}$$

For calculate $F(2,3)$, we need 4 addition/subtraction for signal d, 4 addition/subtraction and 2 division in filter g, and 4 multiplication and 4 addition/subtraction for output m.

However, in the inference phase, the values of the convolution kernels are fixed. So the operation on g can be calculated only once before the inference. So the total number of operations required is for d and m. The number of operations in total, that is, 4 multiplications and 8 additions.

Compared with the direct calculation of 6 multiplications and 4 additions, the multiplier decreases, and the number of additions increases. In computers, multiplication is generally slower than the addition. By reducing the number of multiplications and adding a small number of additions, acceleration is achieved.

When we need to deal with convolutions of other sizes, we can also convert it to the calculation with fewer times of multiplications and more times of additions, as shown in work [87]. We know that in hardware, multiplication consumes more resources than addition, so this trade-off between multiplication and addition can reduce resource consumption.

As far as convolution is concerned, the Winograd algorithm is similar to FFT, which first maps the input and filter to a new space through a linear transformation. After a simple operation in that space, it is mapped back to the original space.

Compared with the commonly used paradigm, im2col + GEMM + col2im, the Winograd algorithm achieves speedup by reducing the number of multiplications, but the number of additions will increase accordingly. At the same time, additional transform calculation and storage of the transformation matrix are required. Considering the cost of the addition, transform, and storage, Winograd is generally only suitable for smaller convolution kernels.

4.6 Comparison and Conclusion

In the previous chapter, we discussed that even if machine learning is widely used, it still has some difficulties for embedded systems, such as the challenges for memory bandwidth and computing resources. In this chapter, we introduced some of the state-of-the-art solutions.

The first introduction is the simplest method, which is to use a compact architecture. Most of the architectures of the convolutional neural network are not derived by theory, otherwise, it needs to be obtained after many experiments. In this case, it is undeniable that these convolutional neural networks have certain advantages or even over-fitting for experimental datasets. It is not strange that classifiers for 9000 categories are not suitable for a dataset with 2 categories. Therefore when we apply the CNNs to different tasks, it is always possible to adjust the architecture of the neural network to adapt the target takes or target dataset.

In terms of compression and acceleration, the more discussed methods are quantization and pruning. Quantization is the method that uses fewer bits to re-express more bit values, to speed up the calculations and compress the memory storage. The pruning method evaluates the importance of connections, then deletes unimportant connections to achieve the effect of acceleration and compression. The two directions are independent but not contradictory. The main purpose of pruning is to reduce the total amount of calculations, while quantification makes each calculation smaller or faster. Therefore, I usually describe that pruning reduces the "quantity" of calculation, and quantization is to reduce the "quality".

Section 3.3.1 has already introduced that FPGA and GPU formed the mainstream accelerators for machine learning. These two accelerators have some differences when implementing pruning and quantization methods.

- when we want to quantify a network into m-bit for GPU accelerators, we need to have a special multiplier for m-bit. At present, some GPUs have been equipped with 16-bit or 8-bit quantization multipliers for machine learning, to replace normal 32-bit multiplication calculators. But when we use other bits, such as 3 bits, as will be used

in chapter 7, the GPU will still use 8-bit multipliers for calculations, which cannot take advantage of quantization. Therefore, GPU-based neural networks are usually quantized to 8-bit or 16-bit. It is worth mentioning that FPGAs can better handle bit-level operations so that the processing of different numbers of bits is more flexible than GPUs.

- On the other hand, when it comes to pruning, as described in section 4.3, we usually set the discarded weights to 0 when the size of the neural network is unmodifiable, to generate the sparse networks. For sparse matrices, the powerful library such as BLAS proposed by GPU optimizes the storage and I/O of memory, as well, optimizes some special calculations such as multiplication of zero. The corresponding libraries for FPGA are fewer so that the development is more difficult. Without sparse storage, even for the value 0, it still needs to use 32 bits to store the values for a 32-bit network. That does not down the size of networks. Moreover, the FPGA algorithm is determined after compilation. Due to the pipeline technology, the FPGA needs to perform fixed operations every cycle. That means it will not skip the multiplication of 0. Although some works have added a branch before multiplication to separate zero and non-zero multiplications, these methods have not been widely used. In short, since the application of FPGA in neural networks is still in its infancy, even though some academic works have studied multiplication of sparse matrices for FPGA, in industrial fields, FPGA-based sparse networks are still not widely used compared to GPUs.

Quantization and pruning are the most used optimizations for traditional CNNs, but the method that uses new computational primitives attempts to replace the heavy calculation in traditional CNN. In other words, while quantization and pruning reduce the "quality" and "quantity" of calculation, this method tries to directly "replace" the calculation by an easier operation. This method fundamentally solves the complicated calculation of convolution, because it directly builds neural networks without convolution. This is a very interesting new idea, but unlike other optimization methods, this method has not been widely discussed. This is largely because convolution works extremely well in image processing, and it is difficult to find alternatives that not only bring good accuracy but also do not consume more computing resources.

In this chapter, we also introduced other methods.

Taking Huffman coding as an example, we introduced the application of traditional compression algorithms. This is a capital idea, but it still needs to cooperate with pruning and quantization to transform the neural network problem into an information coding problem.

On the other hand, we get ideas from matrix factorization to decompose a large matrix into multiple small matrices, thereby reducing the amount of calculation. However, in general, matrix factorization for matrix in CNNs is an approximate algorithm. This algorithm also has an impact on the inference accuracy. Therefore, although this method is versatile, it also necessary tunes many times to find a suitable decomposition method and to balance the accuracy and speed.

Finally, we also mentioned the spatial mapping method. In simple terms, this method projects the matrix into a new space, where the calculation is relatively simple. When the calculation is completed, the result is projected back to the original space. This method generally has advantages for certain specific convolutional neural networks, such as FFT mapping works well for large convolution kernels, but Winograd algorithm for small convolution kernels. But, as some new neural networks have been proposed, such as SqueezeNet proposed in [45], they use a lots of 1x1 convolution kernels, which makes these two algorithms not applicable. In short, this is an interesting idea, but there are many limitations in practice.

Researchers have also proposed the optimization to CNN from other perspectives. For instance, since a large amount of data reading and writing is one of the most time-consuming operations in CNN, [88] apply "in-memory" or "near-memory" computing approaches to achieve superior energy-efficiency by avoiding the von-Neumann bottleneck entirely. It is worth mentioning that computing in memory is also one of the main directions of current chip technology development. The spike neural network (SNN) [89] is another optimization from the perspective of neuromorphic. Unlike traditional CNN which processes dense images, SNN only processes impulse signals generated by image changes. This shows a strong ability in the field of time-continuous signal processing such as video. Due to its sparsity, this kind

of network is more energy-efficient. Because of its impulse-sensitive nature like the brain, this kind of network is considered the direction of the next generation of neural networks [90]. Although these excellent works also optimize CNN in a broad sense, they are not the main research direction of this thesis, so we will not introduce them in detail.

Chapter 5

Fundamental Experiments

In the previous chapters, the state-of-the-art compression and acceleration algorithms for CNNs are introduced. In this section, some of the introduced technologies are applied to the tiny-YOLO network. The main purpose of the experiments in this chapter is to quantitatively measure these methods and provide the references and foundations of the next work. Therefore, the experimental results are still worth discussing.

The architecture of tiny-YOLO is shown in Fig 2.17. To make a difference with the later modified version of tiny-YOLO, it is named as *tiny-YOLO-v0*.

5.1 Compact Design

Compact Design is a relatively simple optimization method. In this section, we propose several designs of implementation and show the changes these designs bring to accuracy.

5.1.1 The Architecture Designs

Using lower resolution for inferring

The resolution of input images affects the number of multiplications. Using a lower input resolution is an obvious method to reduce the number of calculations.

As presented in section 2.3.6, since tiny-YOLO can detect images with different input resolution, smaller images are also adapted to a network trained by larger images. A higher input resolution that contains more information can be used to train the network. Meanwhile, a lower input resolution that needs less calculation is used to infer the detection.

Combining some layers

In architecture tiny-YOLO-v0, the 8^{th} convolutional layer is the most computationally intensive. It takes images with 1024 channels as input and generates 1024 feature-maps. Then that feature-maps are used as input to generate 90 feature-maps in the 9^{th} convolutional layer. The size of filters in 9^{th} convolutional layer is 1x1, which is used to resize the output feature-maps of 8^{th} layer. What we have done is combining the last two layers of architecture tiny-YOLO-v0. As shown in table 5.1, we use 90 filters with size/stride 3x3/1 to generate 90 feature-maps directly. This architecture is named as tiny-YOLO-v1.

It should be noted that this combination is not equivalent to linear-transformation because of the nonlinear functions *leaky* are applied for the output of 8^{th} convolutional layer in tiny-YOLO-v0 but not for tiny-YOLO-v1. Before the combination, it needs $3.48G\ FLOP$ of MAC to process one image with resolution 416x416. After this combination, the needed capacity is reduced to $2.01G\ FLOP$ of MAC.

The combination of the last two convolutional layers is just an example of designing compact. We could also reduce the number of filters in each convolutional layer to lighten the network.

Using sparse max-pools

Even though varied input resolution for inferring is adapted, the smaller image contains less information than the bigger one, which means using a lower input resolution could lead to

Type	Filters	Size / Stride
Convolutional	16	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	32	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	64	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	128	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	256	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	512	3 x 3 / 1
Maxpool		2 x 2 / 1
Convolutional	1024	3 x 3 / 1
Convolutional	90	**3 x 3** / 1
Detection		

Table 5.1: Tiny-YOLO-v1

Type	Filters	Size / Stride
Convolutional	16	3 x 3 / 1
Maxpool		**3 x 3 / 3**
Convolutional	32	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	64	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	128	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	256	3 x 3 / 1
Maxpool		2 x 2 / 2
Convolutional	512	3 x 3 / 1
Maxpool		2 x 2 / 1
Convolutional	1024	3 x 3 / 1
Convolutional	90	3 x 3 / 1
Detection		

Table 5.2: Tiny-YOLO-v2

a loss of precision of detection. the higher input resolution is still preferred. To use high-resolution inputs but keep fewer calculations for the network, the down-sampling rate can be increased to reduce the amount of computation. In other words, a sparser max-pool with a bigger size and stride could be used, to accelerate the contraction of the neural network.

For example, as shown in table 5.2, the size/stride of the first max-pool is set as 3x3/3. To process an image with resolution 288x288x3, the architecture tiny-YOLO-v1 needs $9.63e8$ multiplications, where tiny-YOLO-v2 only needs $4.48e8$ multiplications. It can be seen that with a sparser max-pool, we can significantly reduce the amount of computation.

The main ideas of the approaches, using lower inputs resolution and using bigger max-pool, are both abandoning part of the information by resizing the images/feature maps. Using a lower input resolution means to discard the less important information of images before entering the detection network, while the max-pool retains important information for the network and drops off the information less important during detecting. It should be noted that these two methods are not contradictory but complementary. Due to the mechanism of YOLO, the size of the generated tensor should be odd. Therefore when we change the size of max-pool, we need to change the size of the input resolution accordingly.

5.1.2 Experiments and Analyzes

Essentially, the modification of architecture is trade between accuracy and speed. What we want is to reduce the loss of accuracy, while increasing the speed as much as possible. For our simulation, the accuracy is shown as IOU, the speed is represented by GFLOPS (Giga FLoat-point Operation Per Second). In this section, we simulate the methods mentioned in the section 5.1.1 and make comparisons. The SDC-2018 image set introduced in section 3.2.2 is used to train and valid the network.

Different Resolutions for tiny-YOLO

IOU: train \ infer	416x416	224x224	160x160
416x416	63.36%	48.31%	33.91%
224x224	47.38%	32.10%	28.19%

Table 5.3: IOU of varied resolutions

As we know, tiny-YOLO can detect the object in images that have different input resolutions than the training image set. So the first question we need to know is which resolution we use for the training and inferring. To evaluate the influence of different input resolutions on the detection accuracy, we simulated the detection with a different input resolution of training and inferring, then compared the IOU. The architecture we used is the tiny-YOLO-v0, and the weights are trained with 193 epochs. The result is shown in table 5.3.

Architecture	Down Sampling	13x13x90 GFLOPS	13x13x90 IOU	7x7x90 GFLOPS	7x7x90 IOU	5x5x90 GFLOPS	5x5x90 IOU
tiny-YOLO-v0	32	17.4	63.36%	5.044	48.31%	2.57	33.91%
tiny-YOLO-v1	32	10.0	59.92%	2.913	46.28%	1.49	38.95%
tiny-YOLO-v2	48	10.5	54.30%	3.048	40.70%	1.56	34.25%

Table 5.4: The performances of different architectures

It can be seen that the higher resolution of training and inferring leads to more accurate detection. In order to assure the speed of calculation, we can not use the highest resolution for inferring, but it is possible to use high resolution for training while a low resolution for inferring.

In addition, it should be noted that YOLOv2 and tiny-YOLO can adapt different input resolution for inferring, because they use varied input resolution during training. If we use a different way to train YOLO, the result may be different.

Varied Architectures of tiny-YOLO

Subsequently, the accuracy and the speed of different architecture presented in section 5.1.1 are compared.

As we know, with the different resolution, accuracy and speed are varied. The images with different resolutions contain different information. Intuitively, it is unfair to compare the performances of network structures by using different size images. However, the architectures have different down-sampling rate ($size_{output}/size_{input}$), Feeding the image with the same size to different architectures may cause the output size to change so that not meet the detection mechanism of tiny-YOLO. Instead, with different architecture and its matching resolution, the generated output tenser can be unified into the same size. In order to ensure the comparability of different architecture and the detection mechanism of tiny-YOLO, the comparison of different architecture is performed under the condition that the generated outputs have the same size.

As the result shown in section 5.1.2, a higher resolution of training makes better detection. So for the test, the resolution brings the output $13x13x90$ is used for training, and the size shown in the table 5.4 is the output size for inferring.

For comparison, we list the down-sampling rate ($size_{output}/size_{input}$) of different architecture. For example, if the size of output tensor is $13 \times 13 \times 90$, with down-sampling is 32, the input $(width, high)$ is calculated as $(13 \times 32, 13 \times 32)$, that to say the resolution is 416x416. The result is shown in table 5.4.

From the table 5.4, it serves to show that with the designing compact (v0 to v1), the GFLOPS needed is reduced by 42.5%, at the same time, the accuracy has not dropped a lot. Even though IOU in architecture tiny-YOLO-v1 has reduced, but it has reduced the amount of calculation significant. However, by modifying the size of the max-pool (v1 to v2), the GFLOPS is increased but the IOU decreases. architecture tiny-YOLO-v2 leads to much more loss of accuracy.

5.2 Quantization of tiny-YOLO

Quantization data by representation can speed up the convolutional network. In architecture tiny-YOLO, the weights of convolution are distributed in the interval $[-1, 1]$. Therefore, a more simplified and efficient method can be applied to represent the float by using 8 bits integer. In our proposed method, the linear transformation from the float-point to integer represented by 8-bit integer is:

$$x_q = int(x \times scale) \tag{5.1}$$

where $scale$ is the ratio of expansion from $[x_{min}, x_{max}]$ to $[-127, 127]$, calculated as:

$$scale = \boldsymbol{min}(abs(\frac{127}{x_{max}}), abs(\frac{-127}{x_{min}})) \tag{5.2}$$

Since quantization is only expansion of the value, it is easily to dequantize by contraction, as followed:

$$x = x_q/scale \tag{5.3}$$

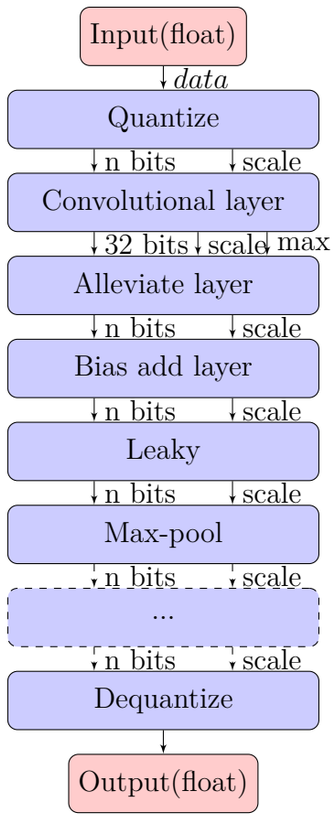


Figure 5.1: The network design for quantization

More generally, it is also possible to increase the number of bits to represent the floating-point, as following definition:

$$scale = \boldsymbol{min}(abs(\frac{2^{n-1}-1}{x_{max}}), abs(\frac{-2^{n-1}-1}{x_{min}})) \tag{5.4}$$

Same as the approach above, the quantized computational layers are used to accelerate the computation, and the inputs and the outputs of these layers are quantized data and its related scales. The quantization and dequantization can be calculated only once at the beginning and the end of the neural network

Subsequently, the computation layer in tiny-YOLO based on this quantization is introduced.

5.2.1 Network Design

The details of the different layers in convolutional neural networks are introduced subsequently.

Convolutional Layer Convolutional layers calculate the convolution 2D of $input$ and $filter$, as

$$conv = input * filter \tag{5.5}$$

where $*$ is the convolution operator. The operands of convolution are constituted by quantized data as:

$$\begin{aligned} input &= input_q / scale_{input} \\ filter &= filter_q / scale_{filter} \end{aligned} \tag{5.6}$$

Therefore, the convolution can be calculated as:

$$\begin{aligned} conv =& (input_q / scale_{input}) * (filter_q / scale_{filter}) \\ =& (input_q * filter_q) / (scale_{input} \times scale_{filter}) \\ =& (conv_q) / scale_{conv} \end{aligned} \tag{5.7}$$

Therefore, the couple $\{conv_q, scale_{conv}\}$ in convolutional layer is defined as:

$$\begin{aligned} conv_q &= input_q * filter_q \\ scale_{conv} &= scale_{input} \times scale_{filter} \end{aligned} \tag{5.8}$$

During calculating the convolution, it is needed to find the maximum value of output matrix $conv_q$, note as max. This value is used to reduce the size of the convolutional result in alleviate layer.

Equation 5.8 shows that the convolution of 32 bits floats by calculating the convolution of 8 bits integers, which achieves 4x compression. To be mentioned that the $scale_{input}$ or $scale_{filter}$ are shared by all the values in inputs or the values in the filter in one convolutional layer, so that we can calculate only once for one convolutional layer.

Alleviate layer The alleviate layer is a special layer for quantized computation. To avoid the overflow of the convolution, we need 32 bits integer as the output of the convolution layer. However, as we have seen, 32 bits integer takes much more bandwidth as well as the computing resource. Thus, we create a alleviate layer to reduce the integer from 32 bits to 8 bits.

What we need to retain are the heaviest 8 bits. Hence, we need to calculate how many bits are used for representing the maximum value max in $conv_q$ matrix, note as x, then drop out the lightest $x - 8$ bits by shifting values.

The output of this layer is defined as:

$$\begin{aligned} output_q &= input_q >> (\lceil log_2(max) \rceil - 8) \\ scale_{conv} &= scale_{input} >> (\lceil log_2(max) \rceil - 8) \end{aligned} \tag{5.9}$$

Bias-add layer Bias add layer is defined as:

$$output^{(k)} = input^{(k)} + bias^{(k)} \tag{5.10}$$

where k is the number of channel of input image or the number of input feature-maps. By using quantized data, it is transformed to:

$$\begin{aligned} output^{(k)} &= input_q^{(k)} / scale_{input} + bias^{(k)} \\ &= \frac{input_q^{(k)} + bias^{(k)} \times scale_{input}}{scale_{input}} \end{aligned} \tag{5.11}$$

Therefore, the quantized bias add layer is defined as:

$$\begin{aligned} output_q^{(k)} &= input_q^{(k)} + bias^{(k)} \times scale_{input} \\ scale_{output} &= scale_{input} \end{aligned} \tag{5.12}$$

The quantized bias-add layer calculates the more multiplication. But only one multiplication should be calculated in each convolutional kernel, which does not bring a big load and it is acceptable.

Leaky ReLU layer Leaky ReLU layer is defined as:

$$output = \begin{cases} input & \text{if } input > 0 \\ 0.1 \times input & \text{otherwise} \end{cases} \tag{5.13}$$

That is represented by quantized data as:

$$output = \begin{cases} \frac{input_q}{scale_{input}} & \text{if } input_q / scale_{input} > 0 \\ 0.1 \times \frac{input_q}{scale_{input}} & \text{otherwise} \end{cases} \tag{5.14}$$

According to the definition and the calculation chain of $scale_{input}$, $scale_{input}$ is always positive. So the output can be represented as:

$$output = \frac{1}{scale_{input}} \times \begin{cases} input_q & \text{if } input_q > 0 \\ 0.1 \times input_q & \text{otherwise} \end{cases} \tag{5.15}$$

Therefore, the quantized leaky ReLU layers is defined as:

$$\begin{aligned} output_q &= \begin{cases} input_q & \text{if } input_q > 0 \\ 0.1 \times input_q & \text{otherwise} \end{cases} \\ scale_{output} &= scale_{input} \end{aligned} \tag{5.16}$$

The quantized leaky layer by this quantization approach takes fewer computation resources than using the quantization in Equation 4.1.

Max-pool Max pooling is realized by applying a max-filter to subregions of the initial representation. For each of the subregions represented by the filter, we will take the maximum value of that subregion and create a new output matrix where each element is the maximum of the original input neutron subregion.

Since *scale* is positive and the same for all the value of the input, a value x is maximum if and only if its quantized integer x_q is maximum. To take the maximum float of a subregion in an input matrix, we just take the maximum integer in that subregion in its quantized matrix.

Therefore, the quantized max-pool is defined as:

$$\begin{aligned} output_q &= maxpool(input_q) \\ scale_{output} &= scale_{input} \end{aligned} \tag{5.17}$$

For the mathematical expression, there is no difference between quantized max-pool layer and the normal max-pool layer. However, the quantized max-pool compares the integers while the normal max-pool compares the floats. They have different hardware implementations.

Compared to the approach in TensorFlow, this approach is only applicable to neural networks whose parameters are concentrated to 0. Such as YOLO, it uses a percentage to express positions information so that the parameters converge to the interval $[-1, 1]$. However, it reduces the amount of computation. For multiplication/convolution, it need not subtract the offset value, the quantized value of 0, that makes multiplication more efficacy. As well, this representation keeps the sign of the original value, so leaky ReLU function is more concise than quantization in TensorFlow.

5.2.2 Experiments and Results

The quantization implementations proposed in section 5.2.1 are applied to the architecture tiny-YOLO-v1 for the benchmark test. The result is shown in table 5.5. Two input resolution, 416x416 and 224x224. are used for the test. The first row in table is the IOU when we used 32 bits float point, as reference for quantized calculation. Then we use different number of bits to represent the float and note the IOU with these different configuration.

Bit \ Reso.	416x416	224x224
32 float	59.92%	46.28%
8 int	50.78%	37.64
9 int	55.49%	41.01%
10 int	60.03%	46.58%
11 int	60.40%	46.96%
12 int	60.06%	46.45%

Table 5.5: The impact of data quantization

We can see that more bits used, the more accurate the results. When we use 10 bits signed integer to calculate, the result is as the same as 32 bits float, but the size of data reduces 68.8%. The accuracy began to decrease when fewer than 10 bits are used. Even if it is reduced to 8 bits, the accuracy of the network is acceptable. But when the used bits are less than 7, the network cannot converge so that cannot be trained.

5.3 Conclusion

In this chapter, the methods which modify the architecture of the network and quantizes the network are applied to the tiny-YOLO network.

It is a good practice to use different structures, which can give a better understanding of the influence of some hyper-parameters of the network, such as the depth of the network or the size of the pooling layer. But this method is more trade-off than optimization. For a specific use scenario or a specific image set, this method can exchange efficiency and precision.

In contrast, quantization brings more significant optimization. Based on the architecture tiny-YOLO-v1 presented in section 5.1, the quantization greatly reduces the size data. But when the number of quantized bits is greater than 8 bits, the accuracy of the network is not reduced or is reduced little.

However, there are still some difficulties for quantization data:

- The first problem is that the alleviate layer takes more computing resources. The alleviate layer needs to know the maximum value of the output of the previous convolutional layer, which means that we need to wait for the entire convolution completed then begin the alleviate layer, therefore pipeline does not work in this situation.

- In some hardware such as FPGA, shift the data with a fixed bit is efficacy. However, as equation 5.9 shown, the number of bits to shift ($log_2(max) - 8$) is variable. The implementation of the dynamic shift operator is a great challenge.

- Another problem is the multiplier. If the multiplier in the accelerator can not handle 10-bit integer multiplication, the 10-bit operand is treated as a 16-bit integer. Even though it is still faster than 32 bits float, we still want to be able to calculate with a more efficient 10-bit multiplication instead of 16-bit.

As seen in the experiment, a network with weight and input less than 8-bit will bring a lot of accuracy loss. But this conclusion is only for the post-training quantization method [12], in which there is no retraining after quantization. We can still compress the network through special training methods. A binary neural network, which is a special quantization network that uses only 1 bit for calculation, it can be built by retraining the network with special methods. The binary network can bring a higher compression rate and faster calculation speed. But it will also bring about the problem of reduced accuracy. We will discuss it in depth in chapter 6.

In chapter 7, another quantization method with less than 8-bit integers is discussed. Different from the method introduced in this chapter, it also needs to retrain the network. The accuracy loss is lower than the binary network, but can deliver a higher compression ratio than the methods described in this chapter.

In total, the experiments in this chapter give the reference for the subsequent works. And in the next two chapters, some solutions are proposed to solve the difficulties in this chapter.

Chapter 6

Selective Binarization Networks

In section 5.2, we introduced the quantization methods in which 8-bit integers are used to re-express floating-point variables. Based on this, we still hope to reduce the number of bits used in the value for calculation. Some works as [22, 91], have proposed the use of binary networks, that is, use 1 bit to re-express the floating-point variables in convolutional neural networks.

When the binary network is used, compared to the original 16-bit floating-point value (half-precision floating-point value), 16x data is transmitted with the same bandwidth. If compared to the original network using 32-bit or 64-bit floating-point (Single-precision or double-precision floating-point) values, the compression rate is higher. Moreover, the arithmetic operations for 1 bit can be simply converted into logical operations, which is very friendly to the hardware.

But as introduction in 5.2, when it is lower than 8 bits, the accuracy will suddenly drop-down. Therefore, we need special training methods for building binary networks. In this chapter, we will introduce how to build a binary network, discuss the challenges for the binary networks, as well, propose the solution for these challenges.

6.1 Binarization Methods

Binarization is a special quantization method in which +1 and -1 are used to re-express the original floating-point value. In an electronic circuit, binary can be expressed by the high and low electrical potential in 1 bit, which is friendly to hardware.

In this section, we will introduce the method of binarization, and how to build a binary neural network.

6.1.1 Binarization Function

In order to constrain a convolution operation $X \otimes W$ to have binary operands, taking weights as an example, the real-valued weights $W \in R^{f_w \times f_h \times C}$ are replaced by binary weights $W^b \in \{+1, -1\}^{f_w \times f_h \times C}$. There are two different binarization functions proposed in work [92].

The first binarization function is deterministic:

$$w^b = Sign(w) = \begin{cases} 1 & \text{if } w \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{6.1}$$

where w^b is the binarized variable and w the real-valued variable. It is very straightforward to implement and works quite well in practice.

The second binarization function is stochastic:

$$w^b = \begin{cases} 1 & \text{with probability} \quad p = \sigma(w) \\ -1 & \text{with probability} \quad 1 - p \end{cases} \tag{6.2}$$

where is the "hard sigmoid" function:

$$\sigma(x) = clip(\frac{x+1}{2}, 0, 1) = max(0, min(1, \frac{x+1}{2})) \tag{6.3}$$

The stochastic binarization is more appealing than the sign function, and some outstanding works such as [93] have also implemented the stochastic binary network in ASIC, and

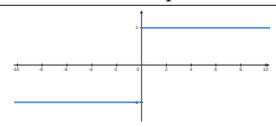
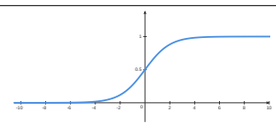
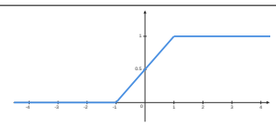
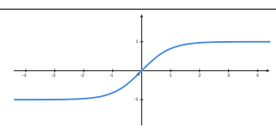
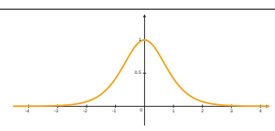
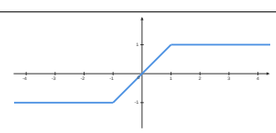
Operation	Function plots	Derivative plots
$Sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$		
$Sigmoid(x) = \frac{1}{1+e^{-x}}$		
$HardSigmoid(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ \frac{x+1}{2} & \text{if } -1 \leq x < 1 \\ 0 & \text{if } x < -1 \end{cases}$		
$Tanh(x) = \frac{e^{-x} - e^{x}}{e^{-x} + e^{x}}$		
$HardTanh(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } -1 < x \leq 1 \\ -1 & \text{if } x \leq -1 \end{cases}$		

Table 6.1: The summary of activation functions used to build a binary network.

based on this, a full binary network in ASIC was established. But in our experiments, we still use the deterministic binarization function which is more easier to implement.

In addition to the binarization function, a real scaling factor $\alpha = \frac{1}{n}||W||_{l1}$ where $n = f_w \times f_h \times C$ is used to approximate the convolution operation as:

$$X \otimes W \approx \alpha(X \oplus W^b) \tag{6.4}$$

where, $\oplus$ indicates a convolution without any multiplication. Since the weight values are constrained to $+1, -1$, the MAC in convolution can be implemented with additions (multiply by $+1$) and subtractions (multiply by -1).

As well, if this binarization function is used to binarize the inputs, a convolution operations can be approximated by:

$$X \otimes W \approx \alpha\beta(X^b \odot W^b) \tag{6.5}$$

where $X^b = Sign(X)$, $\beta = \frac{1}{n}||X||_{l1}$, and $\odot$ indicates a convolution without any algebraic operation, which are replaced by logical operations XNOR-popcount.

6.1.2 Propagating Gradients Through Discretization

We still need to use gradient descent and backward propagation algorithms to train a binary neural network, as introduced in 2.2.4. However, the derivative of the sign function is zero almost everywhere, shown in Table 6.1, making it apparently incompatible with backward propagation. In order to train a binary network, an estimator function of binarization is used to calculate the estimator of the gradient of binarization.

The work [94] studied the question of estimating or propagating gradients through stochastic discrete neurons. They introduced a "straight-through estimator" to cope with this problem. [92] used a similar estimator in a deterministic way, which shown as a Hard Tanh function:

$$HardTanh(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } -1 < x \leq 1 \\ -1 & \text{if } x \leq -1 \end{cases} \tag{6.6}$$

In fact, other estimator such as the $Tanh$ function can also be used to calculate the gradient. But $HardTanh$ is relatively simple which is better than other options. Therefore,

the estimator of gradient for $Sign(x)$ can be calculated by using the estimator function:

$$grad(est(Sign(x))) = grad(HardTanh(x)) = \begin{cases} 0 & \text{if} \quad x > 1 \\ 1 & \text{if} \quad -1 < x \leq 1 \\ 0 & \text{if} \quad x \leq -1 \end{cases} \tag{6.7}$$

More intuitively, we summarize the functions and their derivatives used or mentioned to build binarization functions and networks in this section, as shown in Table 6.1

6.1.3 Building Binarization Networks

Now, by applying the binarization function to the weights and inputs, we can binarize and train neural networks. Then a binary network can be built. In order to distinguish, the network that only the weights are binarized is called binary weights network (BWN), and the network with both the binary weights and binary inputs is called XNOR-Net due to XNOR logic operation replaces multiplication in this network. We use the method proposed in [22] to build a binary Network. The method is shown in Algorithm 1.

Algorithm 1 Training an L-layers CNN with binary weights

Input: A batch of inputs and targets (I, Y), cost function $C(Y, \hat{Y})$, current weight W^t and current learning rate η^t.

Output: updated weight W^{t+1} and updated learning rate η^{t+1}.

1: //**Forward propagation**

2: Binarizing weight filters:

3: **for** $l = 1$ **to** L **do**

4: **for** k^{th} filter in l^{th} layer **do**

5: $\alpha_{lk} = \frac{1}{n}||W^t_{lk}||_{l1}$

6: $W^B_{lk} = sign(W^t_{lk})$

7: $\widetilde{W}_{lk} = \alpha_{lk} W^B_{lk}$

8: $\hat{Y}$ =**BinaryForward**$(I, \widetilde{W}_l)$ // standard forward propagation except that binarized weights are used for convolutions

9: //**Backward propagation**

10: $\frac{\partial C}{\partial \widetilde{W}}$ =**Backward**$(\frac{\partial C}{\partial \hat{Y}}, \widetilde{W})$ // standard backward propagation except that gradients are computed using $\widetilde{W}$ instead of $\widetilde{W^t}$

11: W^{t+1} =**UpdateParamters**$(W^t, \frac{\partial C}{\partial \widetilde{W}}, \eta^t)$ // Any update reuls (e.g., SGD or ADAM)

12: n^{t+1} =**UpdateLearningrate**(η^t, t) // Any learning rate scheduling function

The work [22] also introduced the compression rate for memory and speed up for calculation. In our experiments, the networks with different precision are also applied to the object detection network, tiny-YOLO. The results and comparison are shown in Table 6.2.

It can be seen from the Table 6.2 that binary networks, especially XNOR-Net, have significant advantages in compression rate and calculation speed. At the same time, binary networks also bring acceptable accuracy in classification tasks. However, for object recognition tasks, XNOR-Net showed obvious shortcomings which is a significant decrease in object detection accuracy. In the next section, a solution is proposed for this shortcoming.

6.2 Architecture and Simulation

Many works are discussing binary neural networks. The work [22] uses the XNOR network for classification and achieves good accuracy. In the work [95], it applied BWN to the

	Standard CNN	Quantization (8bit)	BWN	XNOR-Net
Operands	$\mathbb{R}_{32bits} \otimes \mathbb{R}_{32bits}$	$\mathbb{I}_{8bits} \otimes \mathbb{I}_{8bits}$	$\mathbb{R}_{16bits} \otimes \mathbb{B}$	$\mathbb{B} \otimes \mathbb{B}$
Operations	$\times$ +	$\times$ +	+ -	XNOR Bit-count
Memory Saving	1×	4×	$\sim$ 32×	$\sim$ 32×
Computation Saving [22]	1×	-	$\sim$ 2×	$\sim$ 58×
Accuracy on Classification [22]	56.7%	-	56.8%	44.2%
Accuracy on Object Detection	46.28%	37.64%	35.00%	7.00%

Table 6.2: The comparison of standard CNN, quantized CNN (8-bit), BWN, and XNOR-Net. The computation speed and the accuracy on classification are generated by Alexnet applied to ImageNet, and the accuracies on object detection is the IOU of the tiny-YOLO applied to SDC-2018 image set with resolution 224×224.

×	1	-1
1	1	-1
-1	-1	1

(a) Multiply Op.

$\odot$	True	False
True	True	False
False	False	True

(b) XNOR Logic Op.

Table 6.3: The transformation from multiplication to XNOR

YOLO-based object detection system. When binarization is applied for object detection, the accuracy of object positioning will be serious damage. Therefore, the XNOR-net for object detection has not been widely used.

In this section, a method by which most calculations can be binarized within an acceptable loss of accuracy is proposed.

6.2.1 Architecture Exploration

As discussed in section 6.1, XNOR-net and BWN are two types of binary convolutional neural networks which can effectively improve the computing speed and down the size of data. When these binarization methods are applied to only one layer instead of all the network, three different kinds of layers can be found:

- **HH layer**: Both input feature maps and weights are both **H**alf floating-point precision. Compared to neural networks that originally used a single floating point or double floating-point, networks with half floating-point that uses 16-bit to express the real-value are already compressed. But as described in section 4.2, this compression does not require special methods, such as re-express the value or retraining the network, nor does it affect the accuracy of detection, so this compression method can be applied for any neural network, and this widely used compressed network can be seen as the start point of our experiments. For the convenience of the description, the network composited of the HH layer is called HH-ALL in this section.

- **HB layer**: Input feature maps are in **H**alf precision and weights are **B**inary. As mentioned in section 6.1.3, the neural network composed of the HB layer is called BWN. For the convenience of description, it is also called HB-ALL in this section. Compared to HH layers, the weights of HB layers are constraint to $\{+1, -1\}$. And the floating-point multiplication is simplified to multiply by 1 or -1. That means the multiply-accumulate operation (MAC) is reduced to addition (when multiplied by 1 then accumulate) and subtraction (when multiply by -1 then accumulate). Generally speaking, the multiplication requires more time to perform calculations than the addition/subtraction. Since the convolution contains a great number of multiplication operations, replacing them can significantly speed up the calculation. On the other hand, the memory required to store each weight is also smaller, namely, 16x weights can be stored in on-chip memory.

- **BB layer**: Both input features maps and weights are **B**inary. Since all multipliers in this convolution belong to $\{+1, -1\}$, the multiplication is converted to XNOR logic operation, as shown in Table 6.3. Therefore, the MAC operations are reduced to an XNOR-popcount operation, that is, calculate XNOR of operands and count the number of true. For general-purpose calculators, such as GPUs or CPUs, compared to floating-point algebra operations, even addition, and subtraction, logical operations

are faster. For computing circuits such as ASICs or FPGAs, logic operations use fewer resources. This means with the same computing resources, more parallel operations can be set. Also, we know that when we use accelerators, even if the weights can be stored in on-chip memory, the inputs must be sent from the CPU. With 1 bit input, 16x data can be communicated with the same bandwidth.

Each layer has a different rate of computation. As well, the accuracy of the inferring varies.

To speed up the calculation and decreases the resource consummation within a tolerable precision loss, depending on the usage scenario, I propose to mix up these three kinds of layers to build a hybrid system, called selective binarization network. To find a suitable hybrid structure, we deploy the next experiment.

6.2.2 Building the Selective Binarization Networks

As mentioned above, the HH layer should be the start point of our experiment, however, several works as [95, 96] have proposed methods to make the accuracy of *binary weights network* (HB-ALL) close to *floating-point weights network* (HH-ALL). In addition, to reduce the variables in the experiments, and to better study to the impact of the accuracy of each layer, we prefer to fix the weights as binary and change the precision of inputs. Therefore, the whole HB layer network (HB-ALL) is considered as the baseline architecture to conduct the experiments. And then I binarize the feature map of each layer, to speed up the computation with a tolerant loss of accuracy. Since there are no more HH layers in the network, we do not discuss the deployment of the HH layer.

In order to find which layers should be binary, the grid search method is used: It replaces one or two HB convolutional layer(s) by BB layer(s) in sequence. The generated networks are hybrid structures composed of HB and BB layers. Normally the last layer does not take up lots of resources but encodes the final result, so the precision of the last layer is kept as floating-point, which is as an HB layer.

Through the above method, we can build many hybrid networks composed of HB and BB layers and train them. The aim of build these networks is to find the impacts of using binarized layers in different depths. This is an exhaustive method. Therefore, this method can be applied to the small neural network with a shallow depth, for which the requirements for computing resources are relatively small. In the next section, we will conduct experiments based on the small object detection system, a tiny-YOLO network.

The implementation details for each kind of layer are described below:

Building the HB Layer

For the forward propagation in the HB layer, we used the method described in section 6.1.1. The binary weights W^b and the factor α is calculated from the original weights W. Theoretically, the MACs in convolutions are converted to additions/subtractions depends on the W^b. When the convolutions are completed, the values in output are multiplied by α. This is exactly what we do for the inference phase, and we will see the design for the inference phase in section 6.3. But in fact, the calculation speed for the training phase is less important. Therefore, in practice, we do not change the framework Darknet, which directly applies libraries proposed by Nvidia (such as CUDA and CUDNN) to compute convolutions. To build a binary network, we need construct a new matrix,

$$\widetilde{W} = \alpha \cdot W^b \tag{6.8}$$

. $\widetilde{W}$ is used for the normal convolution with the input matrix X to obtain the result of binary convolution, as following:

$$X \otimes W \approx (X \otimes \widetilde{W}) \tag{6.9}$$

To be mentioned, the convolutional layers will be followed by max-pooling and batch normalization layers if necessary. As the same as the forward propagation, the backward propagation computes the gradients by using $\widetilde{W}$ as normal convolutions, then applies these gradients to update the original weights W.

Table 6.4: 12 architectures of mixed tiny-YOLO networks .

Conv Layer	1	2	3	4	5	6	7	8	9
HH-ALL	HH	HH	HH	HH	HH	HH	HH	HH	HH
HB-ALL	HB	HB	HB	HB	HB	HB	HB	HB	HB
BB-1	BB	HB	HB	HB	HB	HB	HB	HB	HB
BB-2	HB	BB	HB	HB	HB	HB	HB	HB	HB
BB-3	HB	HB	BB	HB	HB	HB	HB	HB	HB
BB-4	HB	HB	HB	BB	HB	HB	HB	HB	HB
BB-5	HB	HB	HB	HB	BB	HB	HB	HB	HB
BB-6	HB	HB	HB	HB	HB	BB	HB	HB	HB
BB-7	HB	HB	HB	HB	HB	HB	BB	HB	HB
BB-8	HB	HB	HB	HB	HB	HB	HB	BB	HB
BB-1,2	BB	BB	HB	HB	HB	HB	HB	HB	HB
BB-3,4	HB	HB	BB	BB	HB	HB	HB	HB	HB
BB-5,6	HB	HB	HB	HB	BB	BB	HB	HB	HB
BB-7,8	HB	HB	HB	HB	HB	HB	BB	BB	HB

Building the HH Layer

For the BB layer, we use the same method to build the binary weights W^b and X^b, as well as the factor of weights α. But the factor of inputs *beta* is no longer needed for convolution. The binary convolution is also approximated as:

$$X \otimes W \approx X^b \otimes \widetilde{W} \tag{6.10}$$

where $\widetilde{W} = \alpha \cdot W^b$. The reason for deprecating factor β is that β is calculated by the $l1 - norm$ of the input values, which means that we need to read all the inputs (or all the inputs in one channel) to calculate β. That is time-consuming for the pipeline structure in FPGA. Although the β is discarded, we still consider the scale factor of the inputs. The scale factor of inputs is regarded as a trainable parameter, and it can be integrated into W and its scale factor α. We can simply understand that when a larger input is simplified to ± 1, in order to keep the convolution result constant, W needs to become larger, and that can be achieved through training. More formally, we can describe that the trained weights $W = \beta \cdot \hat{W}$, where β and $\hat{W}$ are the trainable scale factor of inputs and weights used in the convolution introduced in section 6.1.1. For the backward propagation of BB layer, it is the same as HB layers that computing the gradients by using $\widetilde{W}$ but applied the gradients to the original W. For calculate the gradients of input for propagation, the estimator $HardTanh$ presented in section 6.1 is used to estimate the $Sign$ function. This avoids a situation where zero is everywhere.

Updating Parameters

When the forward and backward propagations for all the layers are completed, the parameters such as the weights can be trained by any update rules (e.g., SGD or ADAM). At the end of some training iteration, the hyper-parameters such as the learning rate can also be updated according to the different strategies.

6.2.3 Simulation

For faster training and more convenient on-chip integration, we still take the small network, tiny-YOLO as an example. The image set SDC-2018 and the metric IOU presented in section 3.2 are used to evaluate the performance of proposed methods.

As mentioned earlier, we replace one or two HB layers (except the last layer) with the BB layer in sequence. Since tiny-YOLO has 8 layers that can be replaced, 12 mixed CNN architectures composed of HB and BB layers are generated, as shown in Table 6.4. They are divided into two groups according to the number of BB layers they include (one or two). According to the position of BB layers, the architectures are named as {BB-x} or {BB-x,y}, that means x^{th} or x^{th}&y^{th} layer is set as BB layer.

The training for these networks is launched in a hardware platform composed of 12 IBM Power server compute nodes. Each node has 4 Tesla P100 GPU, with 3584 CUDA cores

inside. We use 3 nodes which include 12 GPU to training 12 mixed tiny YOLO at the same time. Each training task is affined to one exclusive GPU (GPU not shared by others).

Prepossessing Images

Before fed into the network, the images are preprocessed by Darknet. The inputs for tiny-YOLO are images with resolution 416x416, while resolutions of the images in SDC2018 are 1024x768. In order to make the images match the input of tiny-YOLO, we first resize the input image to 416x312, and then merge a 416x52 black rectangle (all colors are 0) on the top and bot of the picture to make the picture size 416x416. Compared to resize to 416 directly, the advantage of resize+merge is that the picture won't be distorted due to the different scaling dimensions in the directions of length and width. The disadvantage is to increase the amount of useless calculation, even if it is a multiplication of zero (with black color), this is still resource-cost and time-consuming.

In addition to the resizing of pictures, we also used image augmentation technology. Image augmentation is one of the data augmentation method that presented in section 2.1.2. Image augmentation technology is to add appropriate noise to the picture, such as adjusting the brightness of the image randomly, as well as rotating or moving the picture randomly, so that similar copies of the original picture are generated. By image augmentation, more different samples can be used for training. This method is usually used when there are not enough images for training. At the same time, because the images are enhanced randomly, image augmentation can also reduce over-fitting for a specific training dataset. For the image augmentation, Darknet adds a variation within the range of ± 0.1 to the hue of images. Darknet also randomly adjusts exposure and saturation of the image by up to a factor of 1.5 in the HSV color space. Tiny-YOLO can process images of different sizes, but we are not studying the effect of different resolutions on images here, so we keep a uniform input size during the training. Rotation and translation of images will change the position and the bounding box of objects. Therefore, rotation and translation are not used in image enhancement.

Training the Networks

For the hyper-parameters throughout training, a batch size of 128, a momentum of 0.9, and a decay of 0.0005 are used. The learning rate is fixed to $1e-5$. The networks were trained up to 100K batches, which is about 190 epochs. The training of 190 epochs of images takes about 220 hours.

The IOU along the training epoch is shown in Figure 6.1. The figure does not contain architecture BB-1 and BB-1,2 which binarizes the input image in the first layer. Since their IOUs are close to 0 until 60 epochs, the training was abandoned. For architecture HB-ALL, 50% IOU can be achieved by continuing the training up to about 500 epochs. But the trainings of other architectures are stopped at 190 epochs because of the limited resource.

It should be noted that YOLO can not only achieve positioning the objects, but also classification. In our experiments, we discussed only IOU, the result of binary object position, but not the effect of classification. That is because binary classification has been widely discussed, such as in the work [22] and [91], the conclusion in our experiments is similar, that is, the binary network does not significantly reduce the accuracy of classification, but it may take longer time to train. Another reason for us to stop studying classification is that our images for experiments are taken by drones. There is only one small object in each image, and every object has a suitable background. Different environments will affect the results of object classification. For example, it recognized a river but said it is a boat and recognized the sky but said it's flying birds. Therefore, this image set can be used for classification, but we do not think it is a good option.

6.2.4 Analyzes and Conclusion

But in these experiments, when the deeper layer (the layer closer to the output) is binarized, the drop in IOU is relatively smaller. I offer some reasonable guesses that will require more experimentation to verify.

- The deeper layers have more filters, so more information can still be detected and retained after binary. In other words, the deeper layer contains more redundancy

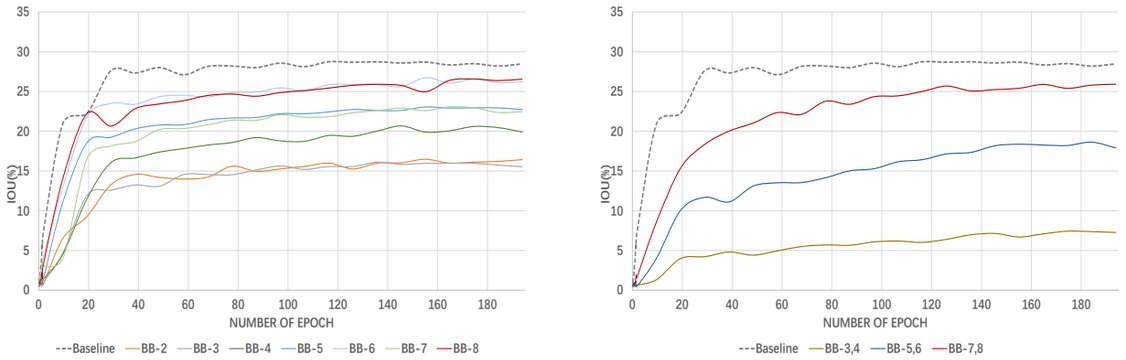


Figure 6.1: IOU along the training of the 12 architecture on the DAC dataset.

information than what we need in the final prediction. In some works of the pruning methods, they are also mentioned that there are more unimportant connections that can be discarded in the deeper than that in the shallower layers.

- We also propose a hypothesis that the binarization used in the shallower layer may throw away too much information, resulting in the deeper layers also unable to complete the detection even though the layer is in higher precision. That is because the deeper layers take the output of the shallower layer as input. When the input image is binary (as architecture BB-1 and BB-1,2), the entire neural network cannot work. This phenomenon can be explained by this hypothesis.

- We have also proposed that the number of layers that are more suitable for being binarized may have a smaller variance or information entropy, so that they can be better assigned to the binary value. Unfortunately, this hypothesis was proved wrong by my experiment, and we are not discussing it.

There may be other more reasonable hypotheses, more experiments are needed to verify them. However, the resources consumed to conduct the experiments are large, so we have not yet conducted all the verification experiments.

According to the experimental results, we can make some reasonable conclusions:

- In the work [95], it is proposed an assumption that the shallower layer of YOLO is mainly responsible for classification, and the deeper is responsible for positioning. So low precision weights are used in the shallower layer and high precision is used in the deeper layer. But, through our experiments, we found that, for the precision of the inputs, the deeper layer should be set with a lower precision, which has less effect on the final result. This is a conclusion to discuss the precision of input, which is not inconsistent with the work [95] who research the precision of weights. But interestingly, we use lower precision in the deeper layers, which has less impact on position accuracy. If we follow the inferences in the work [95], we can take for the shallower layers are responsible for positioning. This is contrary to its assumption. Through the above analysis, I think convolutional neural networks, at least YOLO and its family, are a whole. For a neural network, the role of each connection or neuron in the neural network is still a subject worth exploring, and we cannot easily conclude.

- In total, the proposed compression method is valuable. First, almost all weights become binary. This makes the weights compressed to storage in on-chip memory seen in section 6.3, as well, the multiplication is replaced by addition and subtraction to speed up the calculation. Based on the first optimization, taking BB-7,8 as an example, the IOU of BB-7,8 is close to the baseline HB-ALL. The IOU of HH-ALL can reach 28.47%, while the IOU of BB-7, 8 is 25.91%. The IOU of BB-7,8 is reduced by 8.99 % compared to baseline. This is a loss that cannot be ignored, but it is worth because the 68.8 % of the multiplication is in the 7^{th} and 8^{th} layers, and they are replaced by XNOR operation in architecture BB-7,8. Therefore, this architecture will significantly increase the speed of calculation. More results of execution times for different architecture will be discussed in section 6.3.3.

To sum up, we proposed a compression method that can replace almost all calculations with binary, and 68% of binary calculations are XNOR logical operations, with an 8.99% loss of accuracy. This compression method can be applied to systems that do not require high accuracy but require computational speed, such as drone tracking systems. We continue to propose hardware accelerators in section 6.3, and introduce experimental results based on such hardware in section 6.3.3.

6.3 Hardware Design and Implementation

The architectures presented in section 6.2 reduces the number of multiplications in the network with an acceptable loss of precision. An accelerator is proposed in this section for the architectures presented in section 6.2. The accelerator architecture is suitable for computing one convolutional layer followed by a max-pooling layer if necessary.

6.3.1 Hardware Design

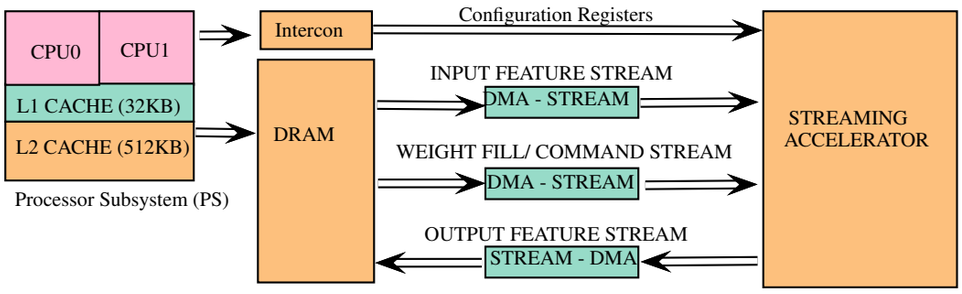


Figure 6.2: The overall of hardware architecture .

In this section, a hardware design is proposed for selective binarization networks.

Figure 6.2 presents the overall system architecture. Before launcher the computations, the configuration that contains the hyper-parameters as image size and convolution depth is written to the related register of the accelerator. The device driver running on the processor handles the tasks of resizing, preprocessing images, pre-loading weights into the accelerator, etc. Then, the image is sent to the accelerator through the input feature stream channel. The images and weights are used to calculate the convolution. When the convolution is calculated, the result is returned and write to the main memory.

From the perspective of the transmitting images and weights, there are still some details to be introduced.

Image-to-Matrix Transformation

The image is a tensor in the three dimension of height, width, and channel. There are usually two formats to transmit it:

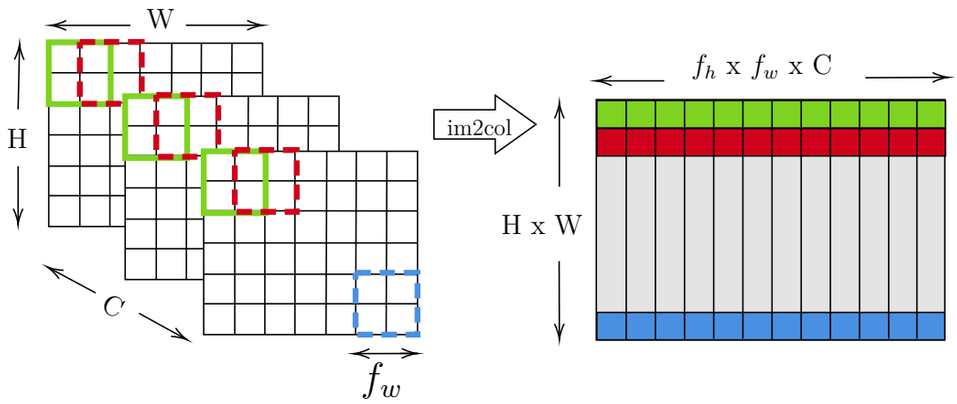


Figure 6.3: Image to Matrix (im2col) Transformation.

- CHW that groups the value according to their channels, then read the value in each channel from top to bottom, from left to right. The output is in the format as:

$$[R_{00}, R_{01}, R_{02}...], [G_{00}, G_{01}, G_{02}...], [B_{00}, B_{01}, B_{02}...] \tag{6.11}$$

  where R_{xy} (G_{xy},B_{xy}) means the value in Red (Green, Blue) channel at the x row and y column.

- HWC that groups the value in the same position (height and width), then read the value in each channel. The output is in the format as:

$$[R_{00}, G_{00}, B_{00}...], [R_{01}, G_{01}, B_{01}...], [R_{02}, G_{02}, B_{02}...] \tag{6.12}$$

Considering that it is possible to transmit multiple images one by one, there are formats NCHW and NHWC where N means the number of images.

However, convolutional layer does not read pixels in the expressed order. Convolution is usually read the pixels scanned by the filter. Images steam cannot directly be computed. That means, when the image is read from the memory, it needs to be reordered before sent to computing unit.

In order to reorder the pixels, the images or activations with C channels of dimension ($H \times W \times C$) is converted to a 2D matrix contains $H \times W$ vectors of size ($f_h \times f_w \times C$), where f_h and f_w are the height and weight of filter, respectively, shown in Figure 6.3. Each vector in this matrix contains a set of variables required for multiply-accumulation (MAC) operation in convolution. If we rearrange the filters as a vector of size $f_h \times f_w \times C$, the MAC in convolution can be seen as the dot product of image vector and weight vector. There are $H \times W$ vectors in the converted matrix, and C_{out} filters in one convolutional layer. That means the convolutions in the convolutional layer are converted to a multiplication of the matrices. This transformation is a standard method in all GPU implementations and some accelerators [10], called image-to-matrix or im2col.

According to the system introduced in Figure 6.2, the image-to-matrix transformation can be done in both processor and accelerator. When image-to-matrix is done in the processor part, we will send the converted matrix to the accelerator, that is larger than original images and takes more bandwidth. We can also just send original images and convert them on the accelerator side. For this, more memory is needed in the accelerator part to cache the received images, at least part of the received images.

In this chapter, we place image-to-matrix on the software side. As it can be seen in Figure 6.7, for convolutional layers with large number of channels, the execution time on the processor is not exceedingly high. This is because with large row, the transformation is mainly done in the cache memory. But for smaller rows, the processor has to fetch several non-contiguous lines and spends much more time to access main memory.

Iterations of Weights

The input feature map matrix for a convolutional layer is streamed into the accelerator. When finishing the calculating, the pixels can be discarded. But the weights need to be kept until the last pixel is calculated. Therefore, during the convolution, all the weights involved in the calculation need to be saved in on-ship memory. Due to limited on-chip memory, only a part of the weights matrix can be loaded at one time (32 filters in the experiments) and it is necessary to use several iterations for convolutional layers with a large number of filters. For example, for a convolutional layer with 256 filters, 8 iterations are necessary.

Streaming Accelerator

There is a configurable image-to-matrix transformation stage which is only used for layers with very few channels (But we do not use this part). The input feature stream is then broadcasted to all the convolutional lanes. Since each convolutional lane corresponds to one filter, the N filter weights are distributed to N lanes from the on-chip weight RAM. As it can be noticed, there is a lot of data reuse (often expressed in terms of MAC/data [4]). In this case, the input feature stream has data reuse of N MAC/data and the weight has data reuse of $H \times W$ MAC/data.

The streaming accelerator has N convolutional lanes which perform an inner product between the incoming input feature and weights streams. As shown in Figure 6.5, the stream

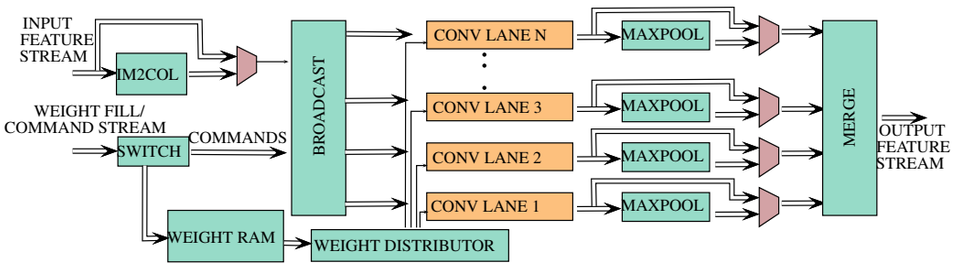


Figure 6.4: The detailed architecture of the streaming accelerator.

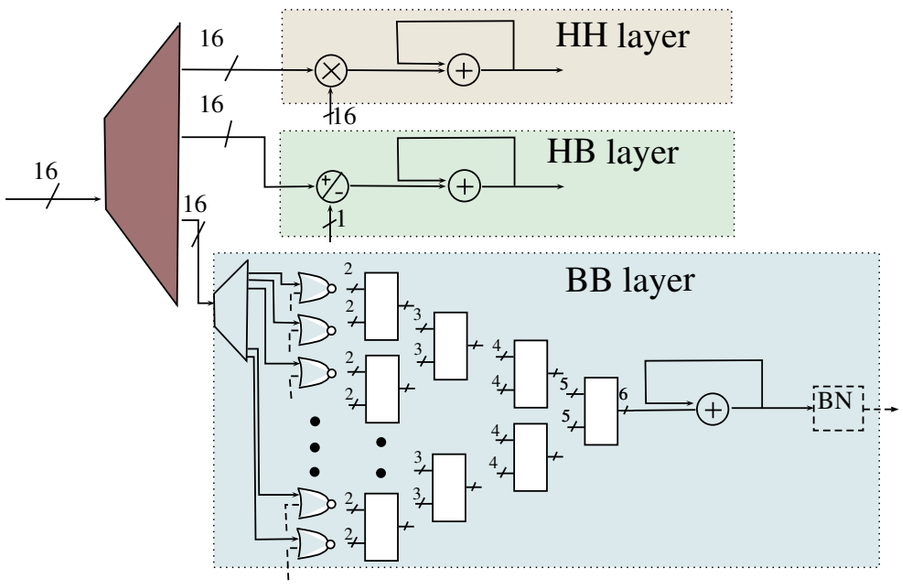


Figure 6.5: The detailed architecture of the convolutional lanes.

computing element performs an inner product of the input streams. The input streams are redirected to one of the three lanes according to the chosen precision. As mentioned earlier, it supports three precision: HH, HB, and BB.

The output of the convolutional lane unit is arranged in the format of HWC. The output stream can be sent to max-pool if configured. However, for the max-pool, taking a filter of size 2x2 as an example, we need to cache 2 rows and 2 columns of output stream in the max-pool for the comparison. Another feasible method is that we calculate the convolution of two adjacent rows at the same time and send them to the max-pool to compare. This method requires one more input channel, but by this method, max-pool only needs to cache two convolution results, which reduces the required memory. We adopted the second method for the experiments in this section.

Finally, the output in HWC format is regrouped and streamed back to main memory. For convolutional layer with C_{out} channels, $\frac{C_{out}}{N}$ iterations are necessary.

6.3.2 Implementation on PYNQ-Z1

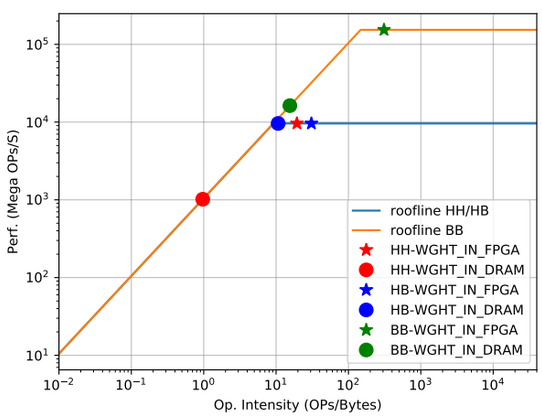
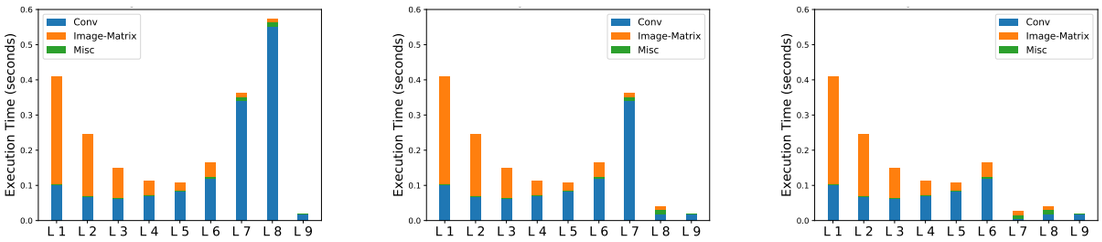


Figure 6.6: The architectural choices and their explanation with roofline.

The experiments here were conducted on the PYNQ-Z1 board [97], which integrates Z7020 FPGA. Table 6.5 shows the resource utilization of my implementation with the Z7020 FPGA. The image-to-matrix transformation part is carried out in the processor.

The main limiting factor is the on-chip memory. Because of the limitation of on-chip memory, only 32 lanes of convolution with half-precision floating-point running at 150 MHz can be implemented. Here a MAC operation is consid-



(a) HB-ALL: IOU:28.47%, time=2.170458s

(b) BB-8: IOU: 26.56%, time=1.64962912s

(c) BB-7,8: IOU: 25.91%, time=1.32284592s

Figure 6.7: The execution time of proposed architectures.The BB-8 and BB-7,8 achieve 1.18x and 1.68x speedup respectively. If considering convolution time only a 2.5x speedup can be achieved.

ered as 2 flops, then the maximum performance of this implementation is 9.6 Gflops/S.

The utilization of Resource is shown as table 6.5.

Table 6.5: The resource utilization in PYNQ-Z1 board with Zynq 7020 FPGA.

Site Type	Used	Available	Util
Slice LUTs	43167	53200	81.14 %
Block RAM Tile	131.5	140	93.93 %
DSPs	64	220	29.09 %

Based on this limitation we have explored other architectural choices:

- Choice Weights_in_FPGA: Feature Maps (FMs) are stored in main memory, and 32 filter weights are stored in FPGA on-chip memory.
- Choice Weights_in_DRAM: Both Feature Maps (FMs) and the weights are stored in the main memory.

With the help of a roofline diagram, Figure 6.6, we show the performance limits of our choices. The roofline for HH and HB are the same, and the roofline for BB is 16x higher performance. We have plotted the points corresponding to the last layer (most compute-intensive) on the roofline. For the HH-Weights_in_FPGA architecture the weights (2 bytes) are stored on-chip, but in practice, it is only possible for the first layer with few weights. The HH-Weights_in_DRAM architecture is limited by memory bandwidth as for each MAC operation we have to stream 66 bytes of data. Both HB-Weights_in_FPGA and HB-Weights_in_DRAM can achieve the performance limit, but we prefer to store the weights on-chip to reduce power consumption. Similarly, BB-Weights_in_FPGA with on-chip storage can achieve the performance limit but BB-Weights_in_DRAM is limited by memory bandwidth.

6.3.3 Experiments and Results

As mentioned in section 6.2, the 68.8% of the multiplication is in the 7^{th} and 8^{th} layers, on the other hand, BB-8 and BB-7,8 get a good IOU shown as Figure 6.1. Therefore, the architectures BB-8 and BB-7,8 are analyzed, which binarize the 7^{th} and 8^{th} layers.

Figure 6.7 shows the detailed execution time for each layer of baseline and BB-8 and BB-7,8 as well as the IOU with 190 epochs images trained. In fact, for architecture BB-7,8 a 1.68x speedup is achievable with an 8.99% loss of IOU.

6.4 Conclusion

In this chapter, we presented methods and architectures to implement CNNs with varied precisions. We proposed an exhaustive search method to retrieve the optimum configuration

of layers, where the precisions can be chosen to be HH, HB, or BB. We also proposed a streaming deep learning architecture where the feature maps are directly streamed to the accelerator and the output stream is stored in the main memory. The advantage of streaming architecture is the pipelined nature of operations where each stage is working in parallel.

This is a subject that can still be studied further. First, the search method for selective binarization should be enhanced from a simple exhaustive search. Next, some of the image-to-matrix transformations can be moved into the accelerator slide, which will considerably reduce the memory bandwidth. It is also interesting to add support for arbitrary precisions.

As a matter of fact, in the next chapter, the implementation of image-to-matrix in the FPGA part is proposed, and a new quantization structure that supports arbitrary precisions and related hardware design is discussed.

Chapter 7

Quad-Approx Networks

In chapter 6 we have proposed the selective binarization method which mixed up HB layers and BB layers. Even though selective binarization improved the speed of calculation significantly, the accuracy of detection is still not as good as the other real-time object detection systems [95].

If HB layers are called as $8+1$ layers since 8 bits for inputs and 1 bit for weights in these layers, BB layers are called as $1+1$ layers. Moreover, in addition to $8+1$ and $1+1$ layers, there are other precisions to choose from. In this chapter, we propose networks composited of $3+3$ layers where 3 bits for both inputs and weights. Although this leads to an increase in the weight size compared to $8+1$ layers, the size of the inputs is greatly reduced compared to 8 bits. And interestingly enough, the weight is usually much smaller than the inputs, and the weights are pre-loaded into the on-chip memory before calculation. Therefore, compressed inputs are more attractive than compressed weights. Due to the reduction of the size of the image, $3+3$ layers still reduce the bandwidth significantly compared to $8+1$ layers proposed in selective binarization.

Obviously, the multiplication replacement proposed in selective binarization cannot be applied in the new proposed layer. In this chapter, we will also propose different calculation units for the $3+3$ layer to process the traditional multiplication.

7.1 Quad and Quad-Approx Network

In this section, we propose a quantization method that uses 2-bits for the value and 1 bit for the sign for both of the inputs images/feature maps and weights for convolutional layers. Moreover, some of the convolutional layers use ReLU as an activation function, and all the output values are positive. In this case, only 2 bits are needed to represent the entire value without the bits for the sign. Only four values $\{0, 1, 2, 3\}$ can be represented by 2 bits, therefore, we call this Quad network.

The works such as [98] have proposed special multipliers for the few-bits integer matrix multiplication. That can be used to implement the Quad network. However, in this section, we propose the method that uses an approximate multiplier which may cause a tolerable error during the calculation, to replace the correct multiplier and to speed up the calculation or reduce the requirement of calculation resource. We call this quad-approx network.

Subsequently, we present how to build a quad-approx network.

7.1.1 PACT for Building a Quad Network

As presented in section 4.2, quantization is a widely used method to compress neural networks. When using quantization, a floating-point value is represented by n bits fixed-point value, or n bits integer value. It can be proved that the two methods are mathematically equivalent. For this point, 2 bits integer is used to represent the value of a floating-point.

The Difficulties of Fewer-bit Quantization

Although quantization has been widely used, the network usually needs to be quantized to 8 bits only in practice. On the one hand, lower-bit quantization requires retraining after quantization. On the other hand, when fewer bits are used, some difficulties shown as the following arise:

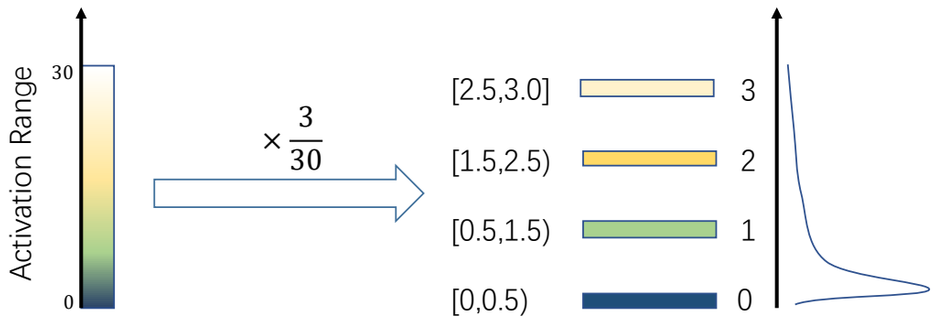


Figure 7.1: Nonuniform distribution of values causes large accuracy losses

- On one side, the activations in the format of floating-point numbers are signed and unbounded, while n-bit integers are always limited. Taking the example of representing floating-point numbers to 8-bit signed integers, the minimum value needs to be represented by -128, while the maximum value is 127. However, in the hidden layer, the activation comes from the calculation result of the previous layer. In other words, to calculate the maximum value and minimum value, we need to wait for all the calculations in the previous layer. This is very time-consuming for a pipeline-based computing system as FPGA.
- On the other side, the distribution of values is not uniform, both for weights and activations. It can often be seen that most of the values in convolutional layers are truly small, but a few are excessively large. When a high dynamic range is used, the floating-point value can be mapped into the integers. But when reduce the quantization bits, and thus reduce the range, many floating-point values will be directly mapped into 0, thus losing the characteristics of the activations and weights. As the example shown in Figure 7.1, the bounded floating-point activation in the interval $[0, 30]$ quantized into 2-bit integers $[0, 3]$, most of the parameters are less than 0.5 and directly quantized to 0. This is particularly problematic when the target bit-precision is 2-bits.

Quantization with Signed PACT

Building on these insights, we use PArameterized Clipping acTivation Function (PACT) presented in the work [21] and the methods presented in [99] to quantize the activation. PACT is an activation quantization scheme in which the activation function has a parameterized clipping level, α. In PACT, the ReLU activation function in CNN is replaced with the following:

$$y = 0.5(\mid x \mid - \mid x - \alpha \mid + \alpha) = \begin{cases} 0, & x \in (-\infty, 0) \\ x, & x \in [0, \alpha) \\ \alpha, & x \in [\alpha, +\infty) \end{cases} \tag{7.1}$$

The output of ReLU is always positive. But some other activation functions such as leaky ReLU used in YOLO may product the negative value as output. Therefore, we propose a different PACT for the signed activation function. It is presented as follows:

$$y = \begin{cases} -\alpha, & x \in (-\infty, -\alpha) \\ x, & x \in [-\alpha, \alpha) \\ \alpha, & x \in [\alpha, +\infty) \end{cases} \tag{7.2}$$

By the PACT method, the activation range is known before the calculation starts, $[-\alpha, +\alpha]$. This is also the quantization range using to quantize data for the next layer without waiting for all calculations completed. At the same time, the PACT method discards excessively large values, so that most smaller values can be relatively scattered in the interval $[-\alpha, +\alpha]$.

Now, 3-bits signed integers will be used to represent floating point numbers in the interval $[-\alpha, +\alpha]$. In fact, that means scaling $[-\alpha, +\alpha]$ to $[-3, +3]$. It should be noted that the range of 3-bit signed integer should be $[-4, 3]$. However, due to the integer is relatively small, the

difference between the scale factor of the positive and negative cannot be ignored. We discard the integer -4 to ensure that the positive and negative values have the same scale factor. The clipped activation output is then linearly quantized to 3-bits for the convolution by the method following:

$$\begin{cases} x_q &= Q(y) = round(y \times \frac{3}{\alpha}) \\ scale_x &= \frac{\alpha}{3} \end{cases} \tag{7.3}$$

where the function *round* returns the nearest integer of a real value. As $y \in [-\alpha, \alpha]$, x_q is an integer in $[-3, 3]$, which can be encoded by 2 bits of value, and 1 bits of sign.

The same strategy is applied as the work in [99] for the value α: every quantized layer in the quad network has its own α, which is shared by all the values in the activation. But different from [99], to find the α for each layer, the hyper-parameters of other layers are kept unchanged, and then the exhaustive method is used to find the integer which gives the best detection accuracy as α.

Similarly, we apply the quantization method to process the weights. But different from inputs, the trained weights are determined before inference. Hence, we can choose an appropriate α for PACT that is related to the values of the weights, instead of an independent value like what to do for the inputs. In fact, we have tried 2 different thresholds to clip the weights: the maximum of weights, and the 2 times the mean of weights. Using maximum value means that quantizes the weights directly without PACT. However, as shown in Figure 7.1, the too-large range caused by maximum value makes the smaller weights tend to zeros, that causes problems while doing quantization. So we use $2 \times \bar{w}$ as the α for PACT, where $\bar{w}$ is the means of the filters in weights, and the α is shared by the values in the same filters. Therefore, the clipped weights w_c is calculated as:

$$w_c = \begin{cases} -2\bar{w}, & w \in (-\infty, -2\bar{w}) \\ w, & w \in [-2\bar{w}, -2\bar{w}) \\ 2\bar{w}, & w \in [2\bar{w}, +\infty) \end{cases} \tag{7.4}$$

And the quantization of weights is shown as the following:

$$\begin{cases} w_q &= round(w_c \times \frac{3}{2\bar{w}}) \\ scale_w &= \frac{2\bar{w}}{3} \end{cases} \tag{7.5}$$

Building the Quad Network

Now, the real-valued of input $X \in R^{W \times H \times C}$ can be replaced by the product of quantized input X^q with $X^q \in \{0, \pm 1, \pm 2 \pm 3\}^{W \times H \times C}$ and its real scaling factor $scale_x$. As well, the real-valued weights $W \in R^{f_w \times f_h \times C}$ can be replaced by the product of quantized weights W^q with $W^q \in \{0, \pm 1, \pm 2 \pm 3\}^{f_w \times f_h \times C}$ and its real scaling factors $scale_w$. It should be noted that the factor is a set of real values. In a convolutional layer, the value of the factor used by each filter is independent.

Then the convolution is approximated as:

$$\begin{aligned} X \otimes W &\approx (scale_x X^q) \otimes (scale_w W^q) \\ &= (scale_x \times scale_w)(X^q \otimes W^q) \\ &= \beta(X^q \otimes W^q) \end{aligned} \tag{7.6}$$

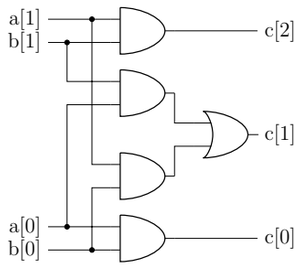
where β is noted as the scale of layer.

All the multiplication in the convolutional layer is replaced by the multiplication between $\{0, \pm 1, \pm 2 \pm 3\}$. We named these networks as *Quad Network*.

7.1.2 Training the Quad Network

The function $y = round(x)$ used in quantization is a discontinued function and the derivative is almost zero everywhere. This means that we can hardly use backward propagation algorithms to train the quad network.

In order to train the quad network, an estimator function of quantization is used to calculate the estimator of the gradient of quantization. This idea is similar to the method of



×	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	7

Figure 7.2: An approximate 2 bits unsigned multiplier.

training a binary neural network introduced in chapter 6. In our experiments, the function $y = x$ is used as the estimator the function $y = round(x)$. And the following estimator for quantization is proposed and used:

$$Est(Q(y)) = y \times \frac{3}{\alpha} \tag{7.7}$$

Therefore, the estimator of gradient of $Q(x)$ is calculated as following:

$$Est(grad(Q(x))) = \frac{Q(y)}{dy}\frac{dy}{dx} = \begin{cases} 0, & x \in (-\infty, -\alpha) \\ \frac{3}{\alpha}, & x \in [-\alpha, \alpha) \\ 0, & x \in [\alpha, +\infty) \end{cases} \tag{7.8}$$

To train a quad network, back-propagation can be applied, where the gradients are computed with respect to the estimated values. And the parameters and the learning rate can be updated by an update rule, e.g., SGD [35] update with momentum or ADAM [60].

In fact, during the training phase, the original floating-point weights are still kept. In each training iteration, the gradient calculated by using quantized weights and quantized inputs but is applied to update the original weights. At the start of the next iteration, the updated original weights are re-quantized again and used for training. For a well-trained neural network, we can directly use the quantized weights for inference without the kept original weights.

7.1.3 Quad-Approx Network

In a quad network, the input for multiplication is represented by 2 bits of value and 1 bit of sign. As mentioned in chapter 6, the bit for the sign can be calculated as the XNOR gate. Hence, we will not discuss the sign of the multiplication. In order to facilitate the discussion, we use sign-and-magnitude code instead of two's complement code to express negative integers, that makes us only consider numerical calculations without discussing the sign of values.

In chapter 6, we used addition/subtraction or XNOR-popcount to replace binary MAC operation in binary convolutional layer. However, since all the operands are represented by 2/3 bits in the quad network, this acceleration is no longer applicable. Even though some works as [98] proposed the multiplier for few bits matrix multiplication, which can implement and speed up the calculation of quad networks, we still want a more streamlined multiplication calculator for the quad network.

We see that excellent neural networks are generalized, that is, they can be applied to images different from the training set. This proves that the neural network is a robust and fault-tolerant system. Based on this phenomenon, we propose a hypothesis: When there are some tolerable errors in the calculator results, the convolutional neural network can still work. The calculator with errors is called an approximate calculator. And the key point of this hypothesis is to choose a suitable approximate calculator. If the approximate calculator takes less resource or faster than the original calculator, it can be used to replace the original calculator, to achieve the result of speeding up and saving resources with a tolerant loss of accuracy.

We proposed an approximate multiplier to replace the original multiplier in this section. We will verify and discuss the effect of these approximate multiplier through experiments in section 7.2.

Figure 7.2 shows one of the approximate multiplier between a[1:0] and b[1:0], the output is encoded in c[2:0]. The calculation results are listed on the right of the image. Compared

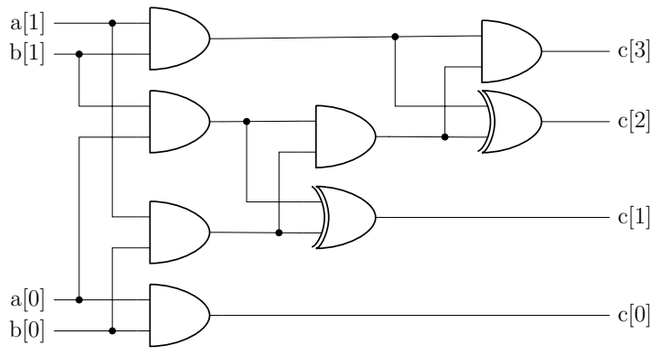


Figure 7.3: One of the standard mulitpler for 2 bit operands

to the exact multiplier, such as the example shown in Figure 7.3, it takes only 5 logic gate, which is far less than the logic gates required by the exact multiplier.

In the approximate multiplication, a mistake occurs when $a = 3$ and $b = 3$, that the result is 7 instead of 9. This mistake may bias the result of the convolutional neural network. Therefore, When the approximate multiplier is applied, we need to retrain the network. The bias caused by the approximate multiplier can be compensated when retraining the network. For example, the bias that the output is less than the actual value can be compensated by increasing the value of the weights during the retrain.

During training neural networks, we still use the derivative of the original multiplication to calculate the gradient. The approximate multiplier will result in a difference between the real gradient and the calculated gradient. However, in some training algorithms, such as stochastic gradient descent, the calculated gradient is also different from the batch gradient. Therefore, this difference caused by an approximate multiplier can be regarded as a stochastic difference. Since the approximate multiplication is still monotonous, the impact of the difference caused by the approximate multiplier on the gradient is limited.

It is observed that the approximate multiplier affects the results of the inference phase but also the training phase of CNNs. However, the impacts are limited and tolerant. At the same time, it can bring some benefits. Part of the benefits of this approximate multiplication are:

- For ASIC implementation, there is only logic calculations, that is much faster than algebraic operation.
- For FPGA implementation, the output of 3 bits signed multiplication is encoded in 4 bits, instead of 6 bits, that saved 33% LUT; the output of 2 bits unsigned multiplication is encoded in 3 bits, instead of 4 bits, that saved 25% LUT.
- Generally, this architecture does not need a special DSP for multiplication. It requires less computational resources for each operator. So it is easier to calculate in parallel than with a traditional multiply operator.

Based on quad networks, approximate multiplication is used to replace original multiplication, and the quad-approx networks can be built.

7.2 Simulation

In the experiments, the tiny-YOLO network is used to detect the single-object detection SDC-2018 image set. The resolution of input images is 416×416. Three steps are needed to train a quad-approx network: 1) selective α_x for each layers; 2) train a quad network; 3) train a quad-approx network.

As mentioned in 7.1, exhaustive method is used to find α_x. In a tiny-YOLO network, the first layer reads the input images, and the last layer generates the prediction result. They need higher precision, so they are kept as fully precision layers. Other convolutional layers are quantized to 3 bits, and the value α_x are shown in the PACT part in table 7.1.

For training the quad network, we use the framework Darknet [57] executed in Nvidia Tesla V100 GPU. For image augmentation, Darknet adds a variation within the range of ± 0.1 to the hue of images. Darknet also randomly adjusts exposure and saturation of the image by up to a factor of 1.5 in the HSV color space. Throughout training a batch size of

Table 7.1: The architecture of tiny-YOLO network and the α for clipping

General					PACT
Type	H x W	C	f_h x f_w / S	N	α_x
Conv.	416 x 416	3	3 x 3 / 1	16	
Maxpool	416 x 416	16	2 x 2 / 2	16	
Conv.	208 x 208	16	3 x 3 / 1	32	54
Maxpool	208 x 208	32	2 x 2 / 2	32	
Conv.	104 x 104	32	3 x 3 / 1	64	27
Maxpool	104 x 104	64	2 x 2 / 2	64	
Conv.	52 x 52	64	3 x 3 / 1	128	18
Maxpool	52 x 52	128	2 x 2 / 2	128	
Conv.	26 x 26	128	3 x 3 / 1	256	12
Maxpool	26 x 26	256	2 x 2 / 2	256	
Conv.	13 x 13	256	3 x 3 / 1	512	7.5
Maxpool	13 x 13	512	2 x 2 / 1	512	
Conv.	13 x 13	512	3 x 3 / 1	1024	6
Conv.	13 x 13	1024	3 x 3 / 1	1024	6
Conv.	13 x 13	1024	1 x 1 / 1	90	

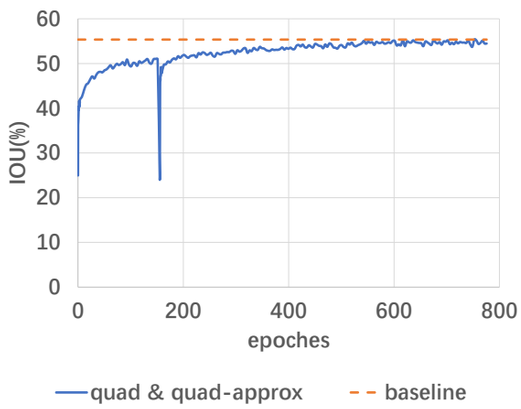


Figure 7.4: IOU of quad network and then quad-approx network, baseline correspondence to original tiny-YOLO.

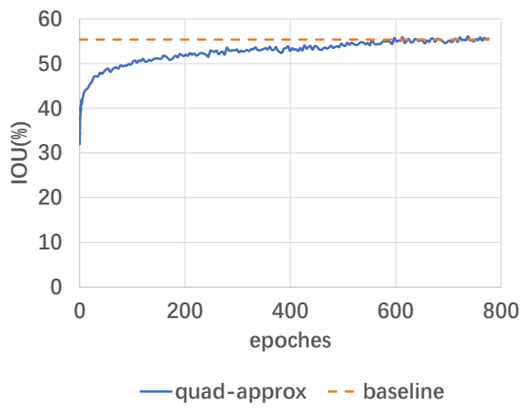


Figure 7.5: IOU of quad-approx network applied at the beginning of training, baseline correspondence to original tiny-YOLO.

128, a momentum of 0.9 and a decay of 0.0005 are used. The learning rate is fixed to $1e-4$. The networks were trained up to 400K batches, which is about 800 epochs.

The IOUs along the training epoch are shown in Figure 7.4. After training with 80K batches (155 epochs), IOU increases slowly. The approximate multiplication is applied to the quad network and a quad-approx network is built. It can be seen in Figure 7.4 that IOU decreases and increases soon at 155 epochs since the exact multiplier is replaced by the approximate multiplier.

In fact, for training quad-approx networks, the step that pre-train a quad network is not necessary. The approximate multiplier can be applied at the beginning of training. Fig. 7.5 shows the IOU along with the training epoch when a quad-approx network is built at the beginning of the training. The results are almost as good as Fig. 7.4. In other words, the approximate multiplier takes less impact. It is observed that the quad-approx network gives a good accuracy of detection even compared to the original tiny-YOLO shown as the reference. In fact, the quad-approx network almost does not cause loss of accuracy in these experiments.

Through these experiments, we also proved that quantization to 3 bits can achieve lossless compression for the CNN based object detection system as tiny-YOLO.

The tiny-YOLO is used in our experiments because this small network is easy to train and easy to implement on hardware, but to prove the generalizability of the method, we applied the quad-approx network to a larger neural network YOLOv2 and other image sets. In some image sets, there are more than one object in one image. Therefore, mAP instead of IOU is used as the evaluation criterion. These mAP are shown in Table 7.2. We can see that the quad network brings a loss of accuracy. YOLOv2 is larger than tiny-YOLO, and

Table 7.2: Quad-Approx Network for YOLOv2 and varied image sets

mAP (%)	YOLOv2	Quad	Quad-Approx
SDC-2018 [64]	90.20	80.52	80.33
COCO-2017 [65]	37.77	30.90	30.82
Pascal VOC [66]	80.07	76.94	76.08

due to limited resources, we did not find the scale factor for each layer, but the scale factor is shared by every three layers. We think this is the main reason for the decrease in accuracy. However, it shows that there is almost no loss of accuracy when applying the approximate multiplier to the network.

7.3 Hardware Design and Implementation

In this chapter, we propose a hardware design adapted to the quad-approx network. The hardware presented in this chapter is completely different from the design in chapter 6. It uses the HLS language to achieve a more complex design. Therefore, we also compare the two hardware designs in this section.

7.3.1 High-level language and Vivado HLS

As shown in Figure 3.9, the Hardware Description Language (HDL), such as Verilog and VHDL can be used to create the representations of a circuit, to then generate the register-transfer level (RTL) circuit designs. In addition, certain high-level languages can also be used to describe the hardware design, such as openCL, SystemC, etc. The code written by the high-level language is then compiled to RTL circuit design. High-level languages require compilers to generate RTL designs, which means that for specific high-level languages, different compilation methods may bring different efficiency and quality. Since high-level languages describe more about the function of the circuit instead of the circuit design, the high-level language is not close to the hardware, and it may lead to more computing resources cost. But high-level languages are easier to write more complex algorithms, as well, it is much easier to be ported between different platforms.

The Vivado High-Level Synthesis (Vivado HLS) compiler enables C, C++, and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado HLS is widely reviewed to increase developer productivity and is confirmed to support C++ classes, templates, functions, and operator overloading. In this chapter, to realize the complex algorithm such as image-to-matrix in the part of the hardware, we use C/C++ and Vivado HLS for hardware design and development.

7.3.2 Hardware Design

The global view of the system is shown as Figure 7.6 Before launching the computation, the configuration that contains the parameters, such as image size and convolution depth is written to the related register of the accelerator. Different from the architecture in chapter 6 who has 2 input stream, there is only one input stream from DRAM to Accelerator. Compared to the architecture presented in Figure 6.2, the architecture for the quad-approx network only has one input stream, that need to pre-write the weights and other factors. The advantage of this design is that takes up less bandwidth. For each calculation, the weights are sent to the accelerator through this channel in advance, and they are stored in the on-chip memory. After the weights are sent, the pre-processed image is also transmitted through the same input channel. In this design, the image-to-matrix is calculated at the part of the accelerator. Hence, the image is sent directly through the NHWC format without reordering. The accelerator reads the stored weights from the on-chip memory, and then performs convolution operations with the image stream from the input channel. The result of the convolution is returned to the main memory through the output stream.

The detailed structure of the accelerator is shown in Figure 7.7. Since the weights and images are transmitted through the same input channel, the accelerator needs to determine the input type according to the configuration. The weights are stored in the block RAM, while the images signal are sent to the im2col module. With im2col operation, the image

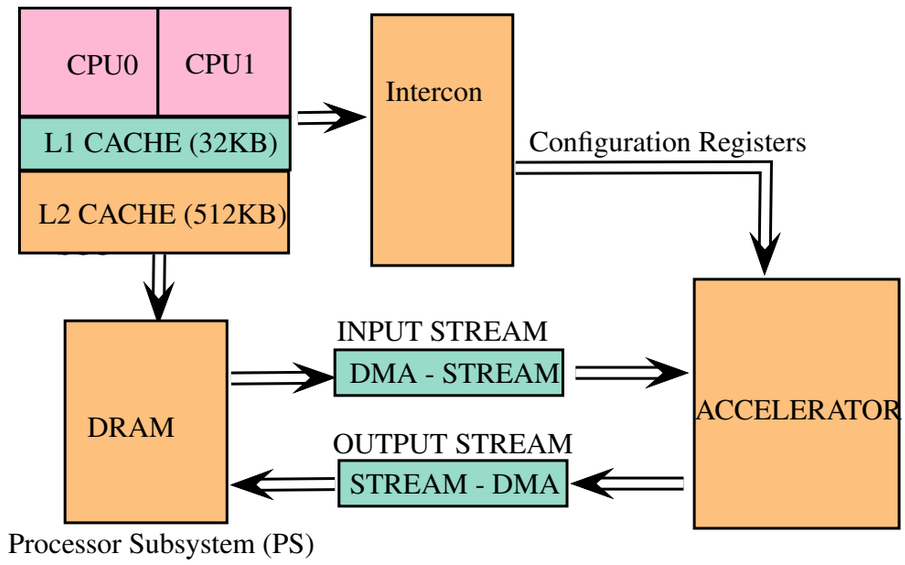


Figure 7.6: The overall of hardware architecture

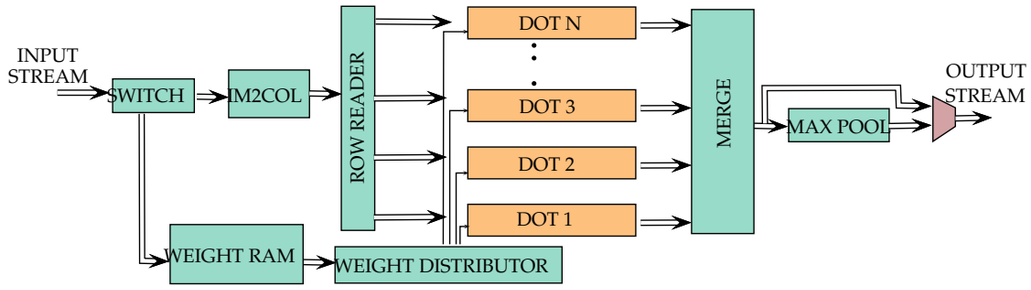


Figure 7.7: The detailed architecture of the Accelerator

is reordered, so that the convolution calculation is converted to matrix multiplication. The matrix multiplication is broken down into the dot-product of multiple vectors. In practice, the image is read row by row, and these rows are assigned to the module for calculating the dot product operation with the corresponding weights. When the dot products are completed, the results are merged and reordered, and the convolution is completed. Then, the pooling module is followed if necessary. Then the calculation result is returned through the output stream. In addition to the introduction of the overall design, some details will be discussed subsequently.

Image-to-Matrix in FPGA Part

The limited memory is the main challenge for Image-to-Matrix modules. We cannot receive and save all the pixels of images because that takes up too much memory. And if the reorder operation starts at the time when all pixels have been received, the waiting time is too long. Therefore, the cache on the accelerator side only holds the minimum pixels required to convert the image. As shown in Figure 7.8, it takes $f_w \times f_w \times C$ pixels to generate the first

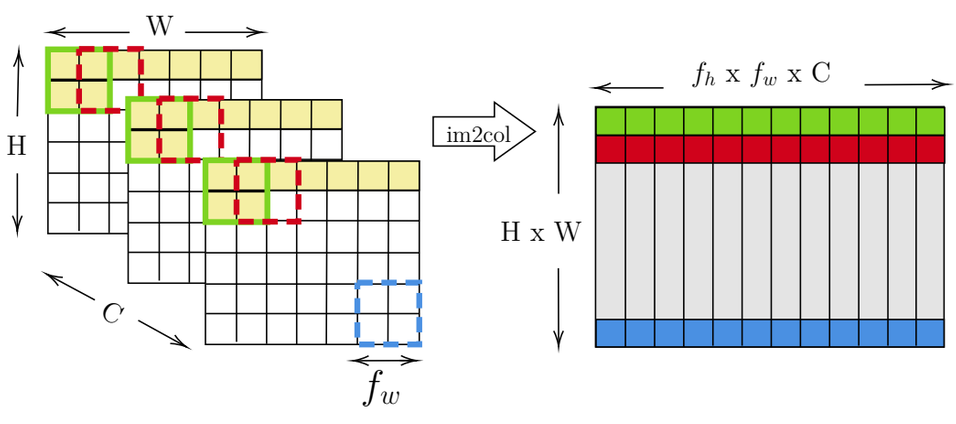


Figure 7.8: Image to Matrix in FPGA Part

row of matrix (the green part). However, since the image is transmitted in NHWC format, $(f_h-1)\times W\times C+f_w\times C$ th pixel (the yellow area) have been transmitted when the last pixel in this row is received. The transmitted pixels are used to generate the next row, so they need to be saved before the first row is completed. Therefore, a cache larger than the size of these received pixels is needed. For convenience, the size of the cache memory is usually set to $h_f \times W \times C$. When these pixels are received, the hardware reads the pixels and reorders them according to different settings, such as filter size and stride. Pixels that are no longer in use should be discarded to free up the cache memory for new pixels.

From the perspective of this design, the Image-to-Matrix is converted to a producer-consumer model with limited cache memory, where the producer receives the pixels and put them into cache one-by-one, and the consumer read the memory according to the convolutional operands' order. Pixels that are no longer consumed will be discarded.

On one hand, this design does not need to save all the pixels of images. On the other hand, since receiving pixels, writing to memory, reordering, and sending to the next module are performed independent, when the memory conflicts are well handled, they can work in parallel and pipeline, thereby increasing the speed of calculation.

Block Matrix Multiplication

Through the Image-to-Matrix module, the images are reordered, and the generated images matrix is used to do the matrix multiplication with weights matrix. To process the matrix multiplication, it is needed to use each column in the weights matrix to multiply each row of images matrix, which is a dot product of vectors.

However, if the weight is too large, or too deep, the vectors will occupy too much memory. But for certain small FPGAs, the too-big buffer brings difficulty to on-chip memory. For resolving this difficulty, the block matrix multiplication is applied in our design. The large matrix is divided into smaller matrices, avoiding the dot product of oversized vectors. For large vectors, the divided matrices can also significantly reduce the required memory. When the matrix is small enough, the matrix can be calculated directly without dividing, which means the number of the block is 1.

In the example shown in Figure 7.9, the images matrix is generated row by row from image-to-matrix module, and finally formed a new 2-D matrix by image-to-matrix module with size $\{W\times H, W_f\times H_f\times C_{in}\}$. Each row of the image matrix is divided into $InputFold$ parts, and there are InP pixels in one part stored in the same buffer. In other words, the size of the block matrix of the image is $1\times InP$. Meanwhile, C_{out} filter kernels are divided into $OutputFold$ groups, and each group has $OutP$ filters. Each filter contains $W_f\times H_f\times C_{in}$ weights as operands of multiplication. These operands are divided into $InputFold$ groups, and there are InP operands in each group. The size of the block matrix of weight kernels is $InP\times OutP$ These divided small matrices are used for block matrix multiplication. The original big multiplication between matrices with sizes $1, f_h\times f_w\times C$ and $f_h\times f_w\times C, C_{out}$ is converted into many small multiplication between matrices with sizes $1, InP$ and $InP, OutP$.

The calculation is performed in the algorithm 2. The loop in line 2-3 and loop in line 6-9 can be combined as a pipeline to speed up the calculation.

Algorithm 2 Block matrix multiplication

Input: A Matrix W that divided as shown in Figure 7.9; A stream in_s that send the Matrix X.
Output: The result of multiplication W · X write into stream out_s

1: **for** $k=1$ **to** $H\times W$ **do**
2: **for** $i=1$ **to** $InputFold$ **do**
3: Read InP values from stream in_s, save into vector $vec[i]$ with size InP
4: **for** $j=1$ **to** $OutputFold$ **do**
5: Set values in vector W to 0, where W is a vector with size $OutP$
6: **for** $i=1$ **to** $InputFold$ **do**
7: Read the block $A_{j,i}$ with size $InP\times OutP$ from memory.
8: Do multiplication and accumulation: $W+=vec[i]\cdot A_{j,i}$.
9: Write the $OutP$ value in vector W into out_s in sequence.

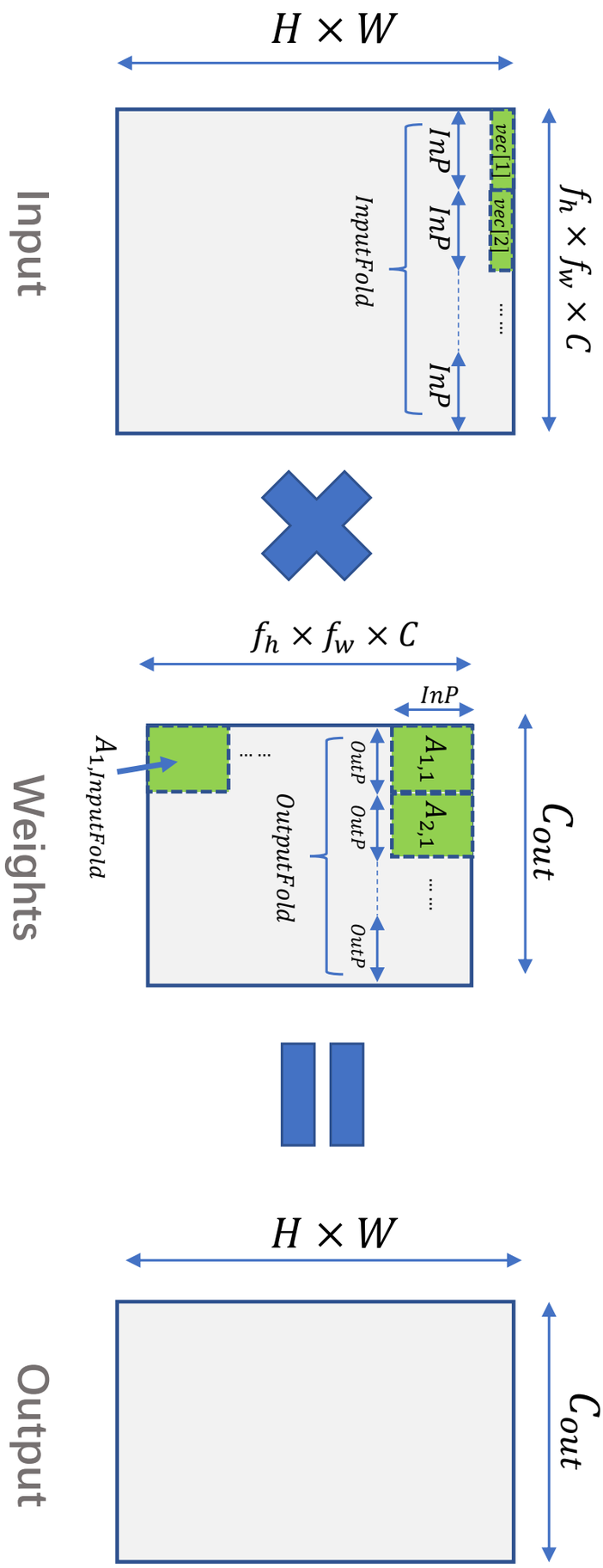


Figure 7.9: Matrix divided into block

Calculation in Parallel

As can be seen from Algorithm2, most of the multiplication and accumulation are calculated by line 8, where a InP vector multiplied by a matrix with size $InP \times OutP$. The matrix can be considered as $OutP$ vectors with size InP, and they do dot product operation with the image vector with size InP. For calculating the dot product of these two vectors, InP multiplication can be processed in parallel. In the proposed hardware design in Figure 7.7, the module for the dot product of 2 vectors with size InP is the minimum calculation unit, named DOT module. Also, there are $OutP$ DOT modules instanced to process the different $OutP$ weight vectors in line 8. Since the DOT instantiation is in parallel, there are $InP \times OutP$ multiplication calculated in parallel in total.

If the quantization method is not applied, $InP \times OutP$ DSPs will be used to calculate the multiplication. If quantized, these DSPs can be replaced by LUTs for a few-bits multiplication. For different FPGAs, according to their different computing resources and memory, InP and $OutP$ can be modified dynamically to change the scale of parallelism, as well as the requirement of computing resources.

Memory Design

The parallel calculation of the block matrix as described above requires a corresponding memory design for weights RAM. The strategy we use is that the weight values for the same DOT module with size InP are stored in the same piece of content. These InP values are read and write together, called an atomic unit of weights. Since $OutP$ DOT modules are instanced for computing in parallel, there are $OutP$ atomic units being read simultaneously. In order to avoid the conflict, $OutP$ RAM blocks are instanced. The $OutP$ atomic units are read from $OutP$ the RAM blocks respectively and distributed to related calculate units, DOT modules. They are broken up and calculated inside the DOT module. It can be simple calculated that there are $InputFold \times OutputFold$ atomic units stored in each RAM block.

In actual operation, the situation will be more complicated: In order to save resources, the instanced RAM block should be used by different layers in a network. But, for memory blocks implemented in FPGAs, the read and write bandwidth (also the size of atomic units) is fixed. The size of an atomic unit is $size_{w\_unit} = WBit \times InP$, where $WBit$ is the number of bits of stored weights value. When the $Wbit$ varied, InP needs to be adjusted accordingly. Taking tiny-YOLO Quad-Approx network presented in section 7.1 as an example, the first and last layers are retained high precision which need at last 8 bits precision, but other layers use 3 bits precision for input images and weights. To maximize the use of memory, the $size_{w\_unit}$ should be the least common multiple of 3 and 8, or an integer multiple of it, for example, $size_{w\_unit} = 48$. Then the corresponding InP for the layer with 3-bits and layer with 8-bits are 16 and 6 respectively. To share the memory, the InP for layers with different bits is constrained. This may reduce the parallel scale of computing units for high-bit precision layers. However, the calculation of the first layer and the last layer is not large. This has little effect on finally calculation speed.

In fact, there is the same problem with $OutP$. Since the bandwidth of output is the same, videlicet, $OutP \times Abit$ should remain the same, where $Abit$ is the number of bit for output. Therefore, when $Abit$ is different, $OutP$ will also adjust accordingly. But since no memory is involved, we can also use redundant methods, that keep maximum bandwidth for different layers.

Packed Activation Function

When the convolution calculation is finished, there are still some operations to follow:

- Multiplied by scale factor β, $y = \beta x$. As shown in equation 7.6, it needs to be multiplied by the scale factor β after the quantization convolution calculation is completed.
- Bias operation or batch normalization. These operations always appear in linear form as $z = ay + b$ where y is the output of convolution with scale factor, a and b are the parameters of bias operator or batch normalization.
- Activation function. Taking leaky Relu as an example, all negative numbers are multiplied by 0.1.

- Quantization for the next layer. For the quad-approx network, the calculation results of the input layer and the hidden layer will be sent to the next layer. Then these data need to be quantified by the PACT method shown in equation 7.2 and 7.3. In our experiments, the clipping factor is noted as α, and it is quantized to [-3,3].

For the above-mentioned operations, they can be packed together, as a format:

$$Q = \begin{cases} 3, & \text{if } ax > (\alpha - b)/\beta \\ 3(a\beta x + b)/\alpha, & \text{if } -b/\beta < ax \leq (\alpha - b)/\beta \\ 0.3(a\beta x + b)/\alpha, & \text{if } (-10\alpha - b)/\beta < ax \leq -b/\beta \\ -3, & \text{if } ax \leq (-10\alpha - b)/\beta \end{cases} \tag{7.9}$$

Even the equation 7.9 seem complicated, β α a and b in this equation are constants. Therefore, it can be simplified to the equation as followed:

$$Q = \begin{cases} 3, & \text{if } ax > S_1 \\ Ax + B, & \text{if } S_2 < ax \leq S_1 \\ 0.1Ax + 0.1B, & \text{if } S_3 < ax \leq S_2 \\ -3, & \text{if } ax \leq S_4 \end{cases} \tag{7.10}$$

where A,B, and S_i can be pre-calculated. This equation packs the four calculations after quantized convolution, therefore called the packed activation function. In fact, for a given convolutional layer, a can also be determined. So the output of the quantized convolution module, x, can be directly distributed into the corresponding calculation module according to the interval, then complete the packed activation function.

Similarly, the parameters of the packed activation function also need to be stored in on-chip-memory. As the memory design presented above, they will be placed in $OutP$ RAM block, and each block stores $C_{out}/OutP$ group of parameters.

7.3.3 Hardware Adaptation

In order to make the neural network more friendly to the hardware, modifications are proposed and applied to the network. In this chapter, we still take tiny-YOLO as an example to introduce these modification methods. It has been proved through experiments that the loss caused by these modifications on the accuracy is negligible, or the network can regain the original accuracy through short retrain.

Leaky ReLU for Hardware

The Leaky ReLU is introduced in section 2.2.2. For Leaky ReLU, when the input is negative, it is multiplied by α. The activation function Leaky ReLU used tiny-YOLO is a special case of Leaky ReLU where $\alpha = 0.1$. We need a floating-point multiplication calculator to calculate the Leaky ReLU function. Even in the packed activation function presented in section 7.3.3, the multiplication of 0.1 is still inevitable.

However, all the inputs of Leaky ReLU are the output of convolution, which are integers. We can use a shift operator to replace floating-point multiplication. First, we adjust the parameters to $\alpha = 0.125$. It is obvious that for positive integers, the multiplication can by replace as the equation followed:

$$int(0.125 \cdot x) = x >> 3 \tag{7.11}$$

However, it does not hold when x is negative. That is because the maximum value of shift operation for negative value is -1 according to the machine code. To process this problem, the Leaky function in tiny-YOLO is modified as equation followed:

$$Leaky(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } -8 < x < 0 \\ 0.125x & \text{otherwise} \end{cases} \tag{7.12}$$

It should be noted that these changes are made on the software side for training tiny-YOLO. According to the Darknet framework used for training, x is represented by a floating-point number. In the hardware side, the implementation is shown as followed:

$$Leaky(x) = \begin{cases} x & \text{if } x \geq 0 \\ x >> 3 & \text{otherwise} \end{cases} \tag{7.13}$$

where x is integer encoding in 16 bits. This change avoids the heavier multiplication on the hardware side but uses faster shift operations to complete Leaky ReLU.

In fact, we have made three changes.

There are three changes for leaky ReLU:

- The parameters α are changed to 0.125. Based on the fault tolerance of CNN, this change hardly affects the accuracy.
- The maximum value of negative leaky is set to -1. This step makes the accuracy slightly lower, but the accuracy quickly rebounds after a short training.
- Since the integer is used instead of floating-point, the shift operator discards the decimal part of leaky. However, as introduced in section 7.3.3, the leaky in tiny-YOLO is implemented as a packed activation function and is quantized into an integer eventually. So this discarding does not bring a decrease in accuracy.

High-precision Layers

For the quad-approx network, quantization is applied for all hidden layers, but two high-precision layers are retained, the first and last layers. In practice, these two high-precision layers are also quantized to 8 bits by symmetry post-trained quantization proposed in work [12]. Compared with the 32-bit original layers, the high precision has achieved 4x compression, which saves bandwidth and memory. At the same time, we can use the 8-bit multiplier instead of 32-bit. Regarding the memory design of 8bit weight, we have already introduced it in the section 7.3.2.

As introduced in the hardware design, each layer is sent to the accelerator and calculated separately. To save resources, only one convolutional layer that can calculate 8-bit multiplication is instanced, which is shared by the first and last layers.

However, we know that an image encoded in RBG format can be represented by an 8-bit unsigned integer, that is, integers from 0 to 255. But the input of the last layer is the output of the hidden layer and encoded in 8-bit signed integers, that contain negative values. To build a general structure for the two layers, we need to build a layer that can handle convolution of 8-bit signed integers, which means that the range of value is $[-128, 127]$. In order to make the images match this range, an offset operator $y = x - 128$ is set during pre-processing of the image, which transforms the unsigned 8-bit integer into signed. This processing will cause a significant decrease in accuracy, but after short retraining, the neural network can correct this linear offset to achieve the original accuracy.

In this subsection, we introduce the methods to make the network adapt to the hardware design. The main idea is to modify the network in software, especially during training, to make it easier to implement in the hardware part. The methods introduced above is not only applicable to tiny-YOLO, but also can be more widely used in the hardware implementation of other CNN networks.

7.3.4 Implementation and Experiment Results

The experiments are conducted in the Zynq UltraScale+ MPSoC ZCU102 board with a target frequency of 400MHz. We compare three different architectures.

- *Arch. Normal* quantized to 8 bits by post-training quantization presented in work [12].
- *Arch. Quad* quantized to 3 bits with exact multiplication.
- *Arch. Quad-Approx* quantized to 3 bits with approximate multiplication.

All architectures have the same structure except for the DOT module that calculates the dot product of the vector. We present the results of synthesis and simulation in Table 7.3.

Quad-Approx network uses LUTs instead of DSP to calculate multiplication. Therefore, more LUTs are used but the utilization of DSPs is reduced. Due to the use of 3bit instead of 16bit for weights, the size of the weights is compressed by 5.3x. The Block RAM in this table shows the total utilization of RAM, including the RAM for weights and for other expressions. Therefore, we can conclude that utilization of RAM has been significantly reduced for Quad

Network Type		Normal(16 bits)	Quad	Quad-Approx
Resources	Slice LUTs	24771	22945	29743
	Block RAM	120	84	84
	DSPs	221	226	82
Performances	Precision	1x	1x	1x
	Cycles for DOT	9	9	4
	Speedup	1x	1x	1.2x
	Compression	1x	5.33x	5.33x

Table 7.3: The utilization and execution time

and Quad-Approx network even normal architecture has already been quantized to 8bit. In addition, since the LUTs are used to calculate the 3bit multiplication, the multiplication in the Quad Approx network costs less time. As well, the MAC operations in Quad Approx networks also take fewer cycles. In syntheses, the DOT module by using DSPs takes 9 cycles while the DOT product in the Quad Approx network takes 4 cycles. For the execution time, the Quad Approx network brings 1.20x speed up.

7.4 Conclusion

It is always a topic worthy of attention to quantize neural networks to few bits. As describe in work [12], conventional quantization can reduce the precision to 8 bits. For fewer-bit quantization, binary network discards the value of activations and/or weights, and directly use the sign function to quantize the network. But the binary system cannot lead to a good detection accuracy in the object detection system. Under the circumstances, we try to retain a part of the value to achieve quantization with few bits.

The signed PACT method we proposed solves the two main problems of the few bit quantization: unbounded activations and uneven weight distribution, so that we can quantize the neural network to fewer bits.

By signed PACT, we can build Quad Network, where the weights and activation of the network are quantized into 3 bits. Compared to the original 16-bit floating-point network, the quad network achieves 5.3x compression of weights and activations. The quad network is a special case of low-bit quantization, but our experiments have proved that using 3 bits can completely perform lossless compression for the object detection system.

The proposed experiments prove that in CNNs, multiplication can be replaced by other approximate operations. Then an approximate multiplier is used to compress and speed up the quantized CNNs. The Quad-Approx networks which take approximate operations bring different benefits, for example, fewer computation resources cost, such as look-up table and DSPs for multiply.

These works are still in progress and various improvements are part of future research. At first, the method that finds the scale factor for PACT in the quad-approx network should be enhanced from a simple exhaustive search. Furthermore, other approximate calculators can be applied. Some other more valuable approximate calculators may bring greater acceleration and structural optimization.

Chapter 8

MinConvNets

In chapter 6, selective binarization is used to speed up the CNNs. With binarized weights, multiplication is converted into addition/subtraction. If the activations are also binary, multiplication is calculated as XNOR gate. In chapter 7, 3-bit quantization is proposed, followed by the approximate calculation for 3 bits multiplication.

Summarizing, the methods introduced in chapter 6 and 7 are quantizing the operands first, then proposing the related accelerated multiplier to speed up the computing. Can we propose a way to directly improve the multiplication without quantization of the operands? In this chapter, we will discuss this issue further.

Firstly, we discuss the approximate calculation of multiplication in section 8.1. In this work, we define similar operations and approximate operations, which can generate results approximate to multiplication under certain constraints, such as the constraints to expected value or variance of multiplication operands. Approximate operations are based on similar operations and have more stringent constraints. The approximate operations can be used to replace the multiplication operations in the error-tolerant systems which meet the constraints.

However, an arbitrary CNN cannot always meet the constraints of approximate operation. Therefore, certain transformations are applied to the general CNN to make them meet the constraints. Then, the approximate operations proposed in the section 8.1 can replace the multiplication operations in the convolutional layers, thereby building CNNs with this approximate convolutional layers, named MinConvNets. Section 8.2 describes the methods to build and train MinConvNets. Then, the proposed architecture is applied to common networks and datasets to analyze the feasibility by the accuracy rate. Section 8.3 focuses on the experimental results of the approximate network, while section 8.4 compares the proposed architecture with other related works. Finally, the conclusion and the future work are discussed in section 8.5.

8.1 Approximate Operations to Multiplication

Similar operation and approximate operation to multiplication are proposed in this section. The similar operation can replace multiplication if followed by a linear transformation, while the approximate operation based on similar operation can directly replace multiplication without transformation.

8.1.1 Similar Operations

Taking a 2D spatial convolution convolution $Z = X \circledast W$ in convolutional layers as an example, the elements in Z are calculated as:

$$z_{u,v} = \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} x_{u+i,v+j} \cdot w_{i,j} \tag{8.1}$$

where (u, v) shows the coordinate of pixels in the image, and f_h, f_w are the height and width of convolutional kernel. If the 2D arrays x and w are unfolded to 1D vectors depends on their coordinate, Equation 8.1 can be transformed into a multiply-accumulate operation as following:

$$z_{u,v} = \sum_{n=1}^{N} \boldsymbol{x}(n) \cdot \boldsymbol{w}(n) \tag{8.2}$$

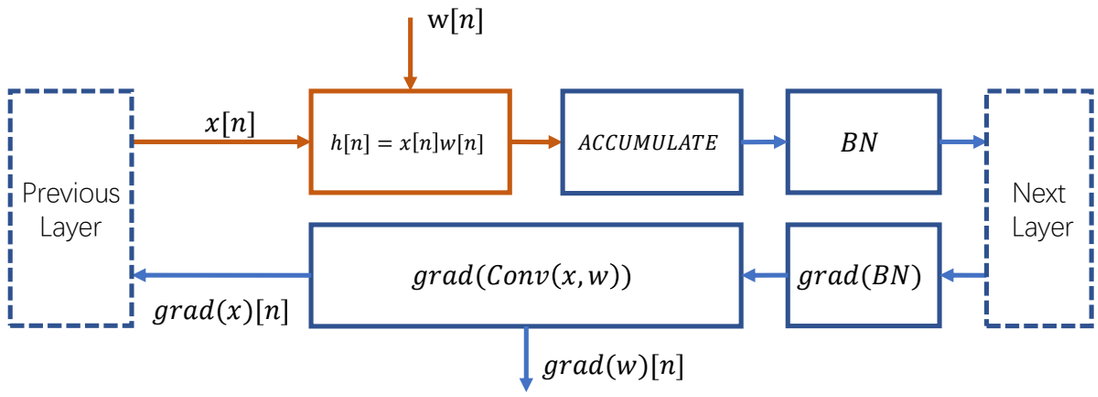


Figure 8.1: The convolutional layer is considered as a system.

where $\boldsymbol{w}$ is the vector of W, $\boldsymbol{x}$ is the vector corresponding to the images pixels scanned by this filter at position (u, v), and N represents the length of the vector calculated as $N = f_w \cdot f_h$. Equation 8.2 can be seen as a system H which takes two spatial signal x and w, then multiply them and generated the signal $h[n] = H(x[n], w[n]) = x[n] \cdot w[n]$ orderly, followed by an accumulation operation. From a perspective of system, this is a module of a convolutional neural network system, shown as Figure 8.1.

If there is another system $g[n] = G(x[n], w[n])$, the dependence of the output signals h and g with the same input signals is an indicator of the similarity between the systems H and G. The most familiar measure of dependence between two quantities is the Pearson product-moment correlation coefficient (PPMCC), commonly called simply "the correlation coefficient". The correlation coefficient between two signals h and g is defined as follows:

$$\rho(h, g) = \frac{cov(h, g)}{\sqrt{var(h)var(g)}} \tag{8.3}$$

where $cov(h, g)$ is the covariance of h and g, and $var(h)$ is the variance of h. The correlation attempts to establish a line of best fit through two signals, and the correlation coefficient indicates the degree of the linear fit of the two signals [100]. In other words, signals with strong correlation can generate approximate signals to each other by linear transformation.

Definition (Similar Operations) *Let two systems take the same input signals, they are similar systems if they can generate strong correlation signals, and the corresponding operations are similar operations.*

For the system shown as Figure 8.1, the multiplication subsystem can be replaced by a similar system followed by a linear transformation, to generate the approximate output. Even though the approximate output brings errors, some works on few-bits quantization as [21, 99] have shown that CNNs are error-tolerant systems and the limited errors can be accepted. It should be noted that in the forward propagation, the approximate output is considered as a result of exact operations with some errors, therefore it is still the exact operations that are used to calculate gradients in backward propagation. The widely used training method SGD [35] uses the approximate gradient to replace the real gradient of the forward propagation, which shows that the operations in forward propagation and the operations used to calculate gradient during backward propagation can be different. Therefore, even if the real gradient of the forward propagation is different from the calculated gradient in the backward propagation in approximate convolution, we believe that errors caused by similar operations have a limited impact on the training of CNNs.

Let μ_h and μ_g be the expected values of the signals $h[n]$ and $g[n]$, the covariance and variances for an input with a convolution kernel size of N are calculated as follows:

$$\begin{cases} cov(h, g) = \sum_{n=1}^{N} (h[n] - \mu_h)(g[n] - \mu_g) \\ \\ var(h) = \sum_{n=1}^{N} (h[n] - \mu_h)(h[n] - \mu_h) \end{cases} \tag{8.4}$$

But what we need is to build approximate systems for arbitrary convolutions instead of for one determined convolution with fixed weights and images. Therefore, x and w can be

Correlation with $h[n] = \|w[n] \cdot x[n]\|$	min-selector $g[n] = \begin{cases} \|w[n]\| & \text{if } \|w[n]\| \leq \|x[n]\| \\ \|x[n]\| & \text{if } \|w[n]\| > \|x[n]\| \end{cases}$	addition $g[n] = \|w[n]\| + \|x[n]\|$	max-selector $g[n] = \begin{cases} \|x[n]\| & \text{if } \|w[n]\| \leq \|x[n]\| \\ \|w[n]\| & \text{if } \|w[n]\| > \|x[n]\| \end{cases}$
$\begin{cases} x \sim N(0,1) \\ w \sim N(0,1) \end{cases}$	0.908	0.882	0.673
$\begin{cases} x \sim N(0,1) \\ w \sim N(0,10) \end{cases}$	0.692	0.683	0.624
$\begin{cases} x \sim N(0,10) \\ w \sim N(0,1) \end{cases}$	0.692	0.683	0.624
$\begin{cases} x \sim U(0,1) \\ w \sim U(0,1) \end{cases}$	0.962	0.926	0.641
$\begin{cases} x \sim U(0,10) \\ w \sim U(0,1) \end{cases}$	0.716	0.717	0.655
$\begin{cases} x \sim U(0,1) \\ w \sim U(0,10) \end{cases}$	0.716	0.717	0.655

Table 8.1: The correlation coefficient of absolute values, where $N(\mu, \sigma)$ means a normal distribution with expected value μ, and variance σ^2, and $U(a, b)$ means a uniform distribution in a closed interval $[a, b]$.

modelled as random variable with a probability distribution, hence g and h are random variable, too. For commonly used image sets, the images are usually independent of each other. If the images for each training iteration are randomly selected, the probability of images used in a new iteration has no relation to the weights trained in the previous iteration. As well, the new images do not affect the probability distribution of the weights trained in the previous iteration, too. Therefore, the weights and the images are considered as independent variables in each iteration. Let $p_x(x)$ and $p_w(w)$ be the probability density functions of x and w respectively, for the two given systems $h = H(x, w)$ and $g = G(x, w)$, the covariance and variances of h and g are calculated as:

$$\begin{cases} cov(h, g) = & \int_\xi \int_\eta (H(\xi, \eta) - \mu_h)(G(\xi, \eta) - \mu_g) \cdot p_x(\xi) p_w(\eta) d\xi d\eta \\ \\ var(h, h) = & \int_\xi \int_\eta (H(\xi, \eta) - \mu_h)(H(\xi, \eta) - \mu_h) \cdot p_x(\xi) p_w(\eta) d\xi d\eta \end{cases} \tag{8.5}$$

where μ_h and μ_g is the expected values of h and g.

Three operators are used to compare the similarities with multiplication: min-selector, addition, and max-selector. In order to maintain the same monotonicity, that magnitudes and signs are treated separately, i.e. all the operations take absolute values of operands only, thereby avoiding negative numbers which make multiplications monotonically decreasing. The signs can be calculated by the XNOR gate independently, as introduced in [22]. According to different operators and their distributions, their similarities are shown in Table 8.1. From this table, it can be seen that:

- The correlation coefficients are all greater than 0.5. In statistics, these signals can be called correlation signals. We believe that this correlation is mainly due to their monotonous increase.
- In case that x and w follow the probability distribution $N(0, 1)$ and $U(0, 1)$, both min-selector and addition show strong correlation, and min-selector is better than addition. Since the calculation is all for absolute value, the variance in normal distribution or interval in uniform distribution can be reflected by the average of the absolute value. We note the above constraint that makes min-selector and multiplication strong correlation as $\mu_{|x|} = \mu_{|w|}$, which means that the expected value of the absolute values of x and w are equal.
- When the absolute values of x and w are more different, the correlation is weaker.

The probability distribution of weights is still a subject under study [101, 102, 103, 104, 105]. But normal and uniform distribution are commonly used initialization weight distributions [106, 34, 47]. Meanwhile, [107] shows that many weights may fit the *t location-scale* distribution. When w follows the above distribution, h and g bring strong correlation with the constraint $\mu_{|x|} = \mu_{|w|}$. In fact, the $\mu_{|x|} = \mu_{|w|}$ is not only a constraint mathematically or experimental, but it also needs to be maintained in the point of view of the convolutional layer in CNN:

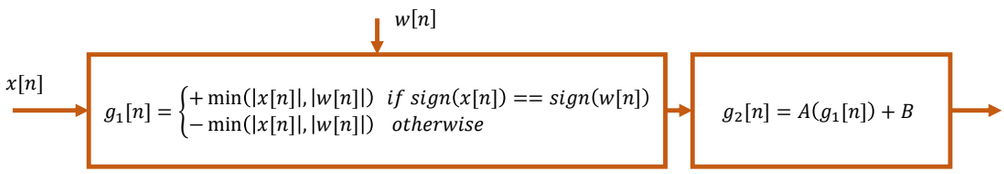


Figure 8.2: The approximate multiplication system constructed by the min-selector.

- If most of the values in $w(n)$ are smaller than $x(n)$, min-selector will highly probably select the value in w, therefore the features in x lost.
- On the contrary, if most of the values in w are larger than values in $x(n)$, no matter what w is, min-selector will highly probably select the value in x, so that w cannot distinguish the pattern in images.

In brief, the constraint $\mu_{|x|} = \mu_{|w|}$ makes w and x comparable, which is a guarantee for w extracting the information in x by using min-selector.

Since multiplication and min-selector are similar operations with the constraint $\mu_{|x|} = \mu_{|w|}$, a subsystem as shown in Figure 8.2 composed of min-selector and followed linear transformation can be built to replace the multiplication in the convolution system. In this subsystem, the $g_1[n]$ takes the minimum absolute value of $x[n]$ and $w[n]$, and then keeps the sign consistent with the multiplication, that is strongly correlated with exact signal $h[n]$. Then, a linear transformation g_2 is followed to generate the result approximate to $h[n]$.

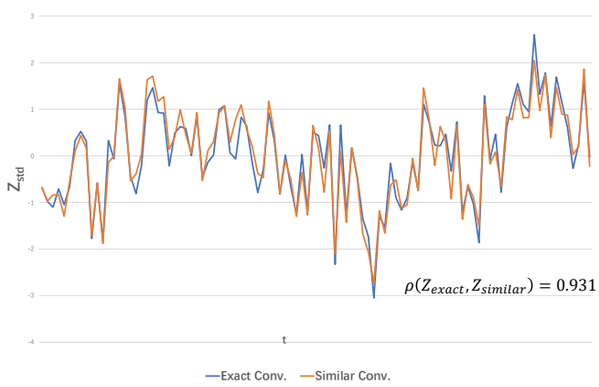


Figure 8.3: The convolution images by exact and approximate operations

We make a comparison of exact and similar one-dimensional convolutions as an example. x and w are randomly generated according to the distribution $x, w \sim U(0, 1)$ and the convolution $Z = conv(x, w)$ is calculated by exact multiplication or similar operations. Since the parameters of the linear transformation are unknown, data standard method *z-scores* shown as Equation 8.6 is applied to both the outputs of exact and similar convolutions to scale data and better compare the two signals.

$$Z_{std} = \frac{Z - \mu_Z}{\sigma(Z)} \tag{8.6}$$

In Equation 8.6, μ_Z and $\sigma(Z)$ are the expected value and the standard deviation of Z. The two standardized signals are shown in Figure 8.3. It is observed that the similar signal has similar trends to the exact signal, and after the data standard method, their values are closed.

It should be noted that, although the data standard method as Equation 8.6 can be applied to both similar and exact convolution, there are problems during training. In fact, the batch normalization layer BN shown in Figure 8.1 is a data standard process by using the average and variance of similar convolutions. However, although the convolution and batch normalization in the forward propagation is replaced by the similar calculations, the result generated is considered as an approximation of the original calculation. So the gradient in the backward propagation is still calculated from original convolution and batch normalization, which means that when training the batch normalization layer, the average and variance of the original convolution is needed. In other words, the average and variance used by the data standard method in propagation are from the similar convolution, but the average and variance of the exact convolutional result are needed for the backward propagation of the data standard method. Therefore, the parameters for training the data standard method cannot be obtained through the forward propagation, as well, we cannot use the data standard method during training a similar network.

8.1.2 Approximate Operations

The similar signal is approximate to the exact signal if followed by a linear transformation as $g_2 = A(g_1) + B$. However, the parameters A and B in this transformation are unknown. To overcome this, we propose two directions:

- The parameters in linear transformation can be found through linear regression or machine learning. Methods for fitting linear models have been discussed in many works [108, 109, 110]. But this means that in every iteration in the training stage, it needs to calculate the exact signal and approximate signal, then perform linear regression calculation to obtain A and B, so this is a costly method. We will not discuss the details.

- Supplementary constraints for similar operations should be proposed so that the linear transformation is not required in the approximate system. The correlation coefficient only reflects the consistency of the changing trend by the degree of linear fitness of two signals, regardless of their difference. In order to make the similar signal replace the exact signal without linear transformation, the difference between the signals should be as small as possible. Next, we discuss the supplementary constraint that makes the difference between two similar operations $g(n)$ and $h(n)$ small.

Definition (Approximate Operations) *Two similar systems are approximate if they have a small difference so that they can replace each other in a fault-tolerant system, and the corresponding operations are approximate operations.*

We use the relative error between the approximate signal and the exact signal as a measure of their difference. For a convolution with determined weights and images, the related error between $h[n]$ and $g[n]$ is calculated as following:

$$L(h, g) = \sum_n \left| \frac{g[n]] - h[n]}{h[n]} \right| \tag{8.7}$$

For arbitrary convolution systems, it is needed to consider the probability distribution of the input signals. Let $p_x(x)$ and $p_w(w)$ be the probability density function of x and w respectively, the difference between signals $h = H(x, w)$ and $g = G(x, w)$ is calculated as :

$$L(h, g) = \int_\xi \int_\eta \left| \frac{H(\xi, \eta) - G(\xi, \eta)}{H(\xi, \eta)} \right| \cdot p_x(\xi) p_w(\eta) d\xi d\eta \tag{8.8}$$

The supplementary constraints need to make $L(h, g)$ as small as possible, where h and g are similar signals. We take the similar signal $h = H(x, w) = |w \cdot x|$ and $g = G(x, w) = min(|w|, |x|)$ as example, where w and x follow normal distributions. In order to ensure the constraint of similar operation $\mu_{|x|} = \mu_{|w|}$, we set that x and w have the same averages and related small variance. Therefore, $w, x \sim N(k, v)$ where $N(k, v)$ means the normal distribution with average k and variance v. With these conditions, $L(h, g)$ is a function about (k, v), and the task of finding the supplementary constraint is transformed into an optimization problem, that is, finding the values of (k, v) which minimize $L(k, v)$.

When $\xi = 0$ or $\eta = 0$ there are singularities in Equation 8.8 because of the division by $H(\xi, \eta) = 0$. Therefore, the points which make $H(\xi, \eta) = 0$ are break-points and are removed from integration interval. In fact, if ξ or η is zero, $H(\xi, \eta) = H(\xi, \eta) = 0$, so there is no error between the two signals. Figure 8.4a shows the the function $L(k)$ with varied variance v. It shows that L achieves a minimum value when k is around 1. Figure 8.4b shows the k that makes $L(k, v)$ minimum with different variances v. It is observed that the value of k is always concentrated around 1 if the variance is less than 1.5. In other words, a quantitative relationship can be obtained: the more values of the operands are concentrated to 1 (or -1 for negative), the more approximate the signals h and g are. In fact, with the application of the batch normalization, the inputs of hidden layers in the CNNs have been standardized, so that we assume x distributed as $x \sim N(0, 1)$. Since the variance v can be changed by simply multiplying by a constant, we study the effect of the variance v on L. Figure 8.4c shows how the value of L changes with variance when $k = 0$. It shows that when $v = 1.57$, L takes the minimum value. For the normal distribution, the average value of the absolute value at $v = 1.57$ can be calculated by its variance, $\mu_{|x|} = 1.25$, that is not far from 1.

$smin(a,b)$	$\|a\| \leq \|b\|$	$\|a\| > \|b\|$
$sign(a) = sign(b)$	$\|a\|$	$\|b\|$
$sign(a) \neq sign(b)$	$-\|a\|$	$-\|b\|$

Table 8.2: The operation $smin(a, b)$.

Based on the above analysis, we propose a constraint of approximation that the absolute value of operands should be concentrated around 1. Considering the constraint of similarity, we mark the constraint with which min-selector is approximate to multiplication as $\mu_{|x|} = \mu_{|w|} = 1$, i.e., the values of the two input parameters are as close as possible, and concentrated to 1 or -1. This is not an exact mathematical condition, but it is the main guide for building a multiplication-less CNN with min-selectors in the next section.

8.2 Building Approximate Networks

In section 8.1, we propose min-selector which is an approximate operation to multiplication with the constraint $\mu_{|x|} = \mu_{|w|} = 1$. However, the multiplication in arbitrary neural networks does not always meet the constraint. Therefore, in this section, we simply transform the convolutional layer to make it meet the constraints, and then use approximate operations to construct the convolutional layer without multiplication.

8.2.1 Building the Approximate Convolution

Next, we proposed the transformation to make an arbitrary multiplication $z = x \cdot w$ meet the approximate constraint $\mu_{|x|} = \mu_{|w|} = 1$. To keep the same monotonicity between multiplication and min-selector, the value and sign of output results are processed separately.

The first step is to set the expected value of the absolute value of the operands to 1:

$$\frac{|z|}{\mu_{|x|}\mu_{|w|}} = \frac{|x|}{\mu_{|x|}} \cdot \frac{|w|}{\mu_{|w|}} \tag{8.9}$$

where $\mu_{|x|}$ and $\mu_{|w|}$ are the expected value of $|x|$ and $|w|$. At this point, the two operands for multiplication are $\frac{|x|}{\mu_{|x|}}$ and $\frac{|w|}{\mu_{|w|}}$, and their expected values are both 1. Then, the min-selector can be used to replace the multiplication to generate an approximate value, as shown below.

$$\frac{|z|}{\mu_{|x|}\mu_{|w|}} \approx min(\frac{|x|}{\mu_{|x|}}, \frac{|w|}{\mu_{|w|}}) \tag{8.10}$$

According to the nature of min-selector and multiplication, the Equation 8.10 is transformed to obtain the following equation:

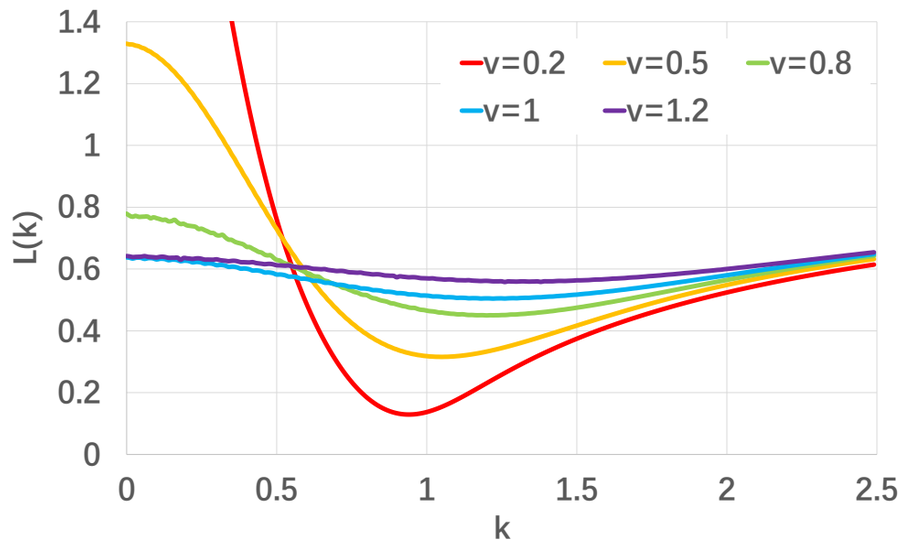
$$\begin{aligned} |z| &\approx \mu_{|x|} \cdot \mu_{|w|} \cdot min(\frac{|x|}{\mu_{|x|}}, \frac{|w|}{\mu_{|w|}}) \\ &= \mu_{|x|} \cdot \mu_{|w|} \cdot (\frac{1}{\mu_{|x|}} \cdot min(|x|, \frac{\mu_{|x|}}{\mu_{|w|}} \cdot |w|)) \\ &= \mu_{|w|} \cdot min(|x|, \frac{\mu_{|x|}}{\mu_{|w|}} \cdot |w|) \end{aligned} \tag{8.11}$$

Since the sign of min-selector should be consistent with multiplication, the approximate calculation for the signed operands is:

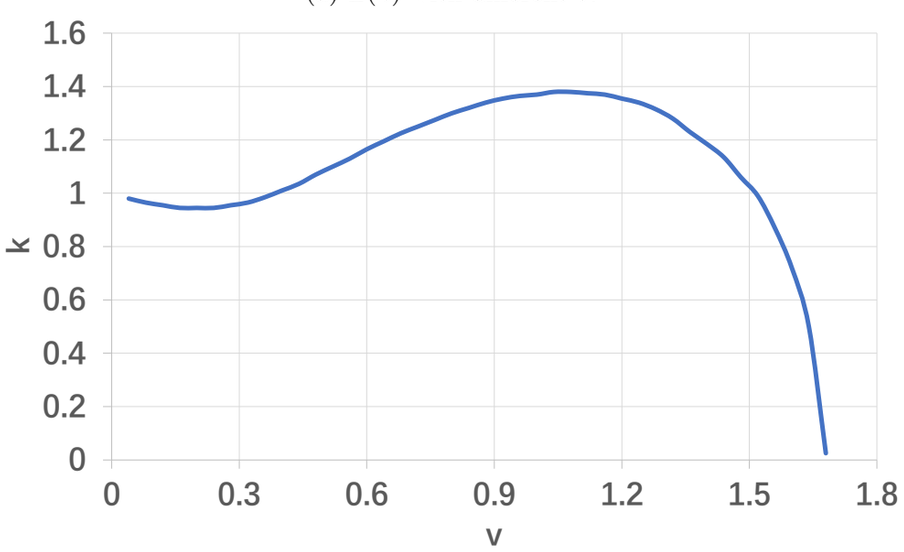
$$z \approx \quad \mu_{|w|} \cdot smin(x, \frac{\mu_{|x|}}{\mu_{|w|}} \cdot w) \tag{8.12}$$

where $smin(a, b)$ is for signed operands and calculated as Table 8.2. It can be seen that $smin(a, b)$ is calculated by comparing the values and signs of operands separately. And there are only four possible outputs according to the comparison, so that can be implemented in a 4-to-1 multiplexer with comparators, which is friendly to some platforms such as FPGA [11].

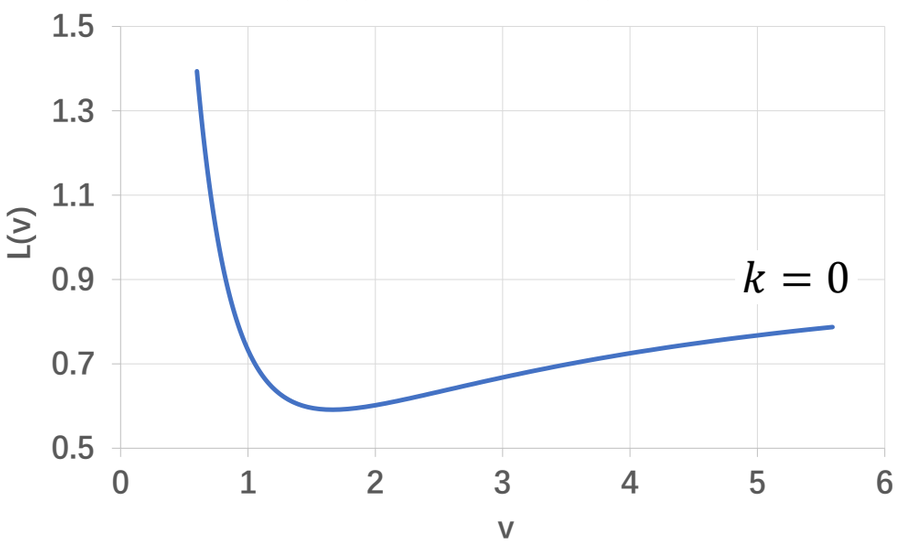
For a well-trained neural network, the weights W and $\mu_{|w|}$ are known. Inspired by [43] which uses moving averages in batch normalization to fix the averages and variances during



(a) $L(k)$ with different v.



(b) $k(v)$ with minimum $L(k)$.



(c) $L(v)$ with $k = 0$.

Figure 8.4: The figures about L, k, v.

inference, the moving $\mu_{|x|}$ obtained through training is used to fix $\mu_{|x|}$ as a constant during inference. Therefore, for a well-trained network, $\mu_{|x|}$ and $\mu_{|w|}$ are known, so we can generate new weights $\widetilde{W}$ where the elements are calculated as:

$$\widetilde{w} = \frac{\mu_{|x|}}{\mu_{|w|}} \cdot w \tag{8.13}$$

which can be directly used for comparison with the inputs without recalculation during the inference.

Then the convolution in Equation 8.1 can be calculated by approximate operations as follows:

$$\begin{aligned} z_{u,v} &\approx \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} (\mu_{|w|} \cdot smin(x_{u+i,v+j}, \widetilde{w}_{i,j})) \\ &= \mu_{|w|} \cdot \sum_{i=1}^{f_h} \sum_{j=1}^{f_w} smin(x_{u+i,v+j}, \widetilde{w}_{i,j}) \end{aligned} \tag{8.14}$$

Although there is still 1 multiplication operation in Equation 8.14, it is much fewer than the $f_h \cdot f_w$ multiplication operations in Equation 8.1. Since many convolutional layers are followed by the bias layer or the batch normalization layer, the multiplication in Equation 8.14 can be integrated into these following layers, thereby constructing an approximate convolutional layer that does not require multiplication.

8.2.2 Training Approximate Convolutional Layers

Based on the approximate convolution constructed in section 8.2.1, we can build an approximate convolution layer. We call the neural network composed of the approximate convolutional layers MinConvNets. Algorithm 3 shows how to train a MinConvNets. Some

Algorithm 3 Training an L-layers CNN with approximate convolutions.

Input: A batch of inputs and targets (X, Y), cost function $cost(Y, \hat{Y})$, current weight W^t and current learning rate η^t.

Output: updated weight W^{t+1} and updated learning rate η^{t+1}.

//**Forward propagation**
for $l = 1$ **to** L **do**
  $\mu^r_{|x_l|} = \gamma \cdot \mu^r_{|x_l|} + (1 - \gamma) \cdot \mu_{|x_l|}$
  $X^c_l = clip(X^t_l, 2\mu_{|x_l|})$
  for k^{th} filter in l^{th} layer **do**
    $W^c_{lk} = clip(W^t_{lk}, 2\mu_{|w_{lk}|})$
    $\widetilde{W}_{lk} = \frac{\mu_{|x_{lk}|}}{\mu_{|w_{lk}|}} \cdot W^c_{lk}$
  $\hat{Y}_l =$**ApproxForward**$(X^c_l, \widetilde{W}_l)$
//**Backward propagation**
$C = cost(Y, \hat{Y}_L)$
for $l = L$ **to** 1 **do**
  $\frac{\partial C}{\partial X^c_l}, \frac{\partial C}{\partial W^c_l} =$**ExactBackward**$(\frac{\partial C}{\partial \hat{Y}_l}, W^c_l, X^c_l)$
  $\frac{\partial C}{\partial W^t} = \frac{\partial C}{\partial W^c_l} \cdot \frac{\partial W^c_l}{\partial W^t_l}$
  $\frac{\partial C}{\partial \hat{Y}_{l-1}} = \frac{\partial C}{\partial X^t} = \frac{\partial C}{\partial X^c_l} \cdot \frac{\partial X^c_l}{\partial X^t_l}$
//**Update Patameters**
$W^{t+1} =$**UpdateParamters**$(W^t, \frac{\partial C}{\partial W^t}, \eta^t)$
$n^{t+1} =$**UpdateLearningrate**(η^t, t)

more detailed introductions are as follows:

- At the beginning of forward propagation, the moving expected value is used to record the average of the absolute value $\mu^r_{|x|}$, where γ is the momentum of moving value. While $\mu_{|x|}$ is used during training, a well trained $\mu^r_{|x|}$ can be used to replace $\mu_{|x|}$ during the inference phase to avoiding calculations.

- X and W are clipped by *clip* function, and $clip(X, \alpha)$ means $\forall x \in X$ the calculation following is applied:

$$clip(x, \alpha) = \begin{cases} \alpha & \text{if } x > \alpha \\ -\alpha & \text{if } x < -\alpha \\ x & \text{otherwise} \end{cases} \tag{8.15}$$

  Correspondingly, the gradient is calculated as follows:

$$grad(clip(x, \alpha)) = \begin{cases} 0 & \text{if } x > \alpha \text{ or } x < -\alpha \\ 1 & \text{otherwise} \end{cases} \tag{8.16}$$

  The main purpose of the *clip* function is to avoid excessive variance caused by excessive values which may reduce the approximation of operations. For a well-trained network, the pre-clipped weights can be used for the inference stage. And the inputs do not need to be clipped during the inference stage, because with the clipped weights, too large input values will not be selected by min-selector. Therefore, the *clip* function is a method to ensure that the network converges, but it does not increase the amount of calculation during the inference stage.

- $ApproxForward$ is a standard forward propagation, except that the matrix multiplication calculated as shown in Equation 8.14. The result of the convolution needs to be multiplied by the constant $\mu_{|w|}$. As mentioned in section 8.2.1, this operation can be done directly or integrated into subsequent layers.

- The $ApproxForward$ uses $\{X^c_l, \widetilde{W}_l\}$ to calculate the approximate convolution of $\{X^c_l, W^c_l\}$. Therefore, it still calculates the gradient of an exact convolution for backward propagation, i.e., $\{X^c_l, W^c_l\}$ are used as inputs instead of $\{X^c_l, \widetilde{W}_l\}$.

- Any update rules (e.g., SGD or ADAM) and learning rate scheduling functions can be applied to update the parameters and learning rate at the end of the algorithm.

8.3 Experiments

In order to compare our multiply-less architecture with standard results, we build different networks and apply them to different datasets.

8.3.1 Networks and Image Sets

LeNet and mini_cifar are used to build approximate networks. LeNet shown in Table 8.3 is a small convolutional network for image classification. In order to measure the performance for negative operands, leaky ReLU [50] instead of ReLU is used in convolutional layers as the activation function. The mini_cifar shown in Table 8.4 is a network deeper than LeNet. There are 6 convolutional layers, but three of them are 1x1 convolutional layers, mainly used to adjust the size of the network.

The image sets used were: MNIST and Cifar10. MNIST [62] is a images-set of handwritten digits with 60,000 examples in the training set and 10,000 examples in the test set. On the other hand, Cifar10 [63] is a data-set of 60000 colored images in 10 classes with 50000 training images and 10000 test images. The details of these image set are introduced in section 3.2.2.

8.3.2 Approximate Networks

The accuracy of different networks and image sets are shown in Table 8.5. In order to evaluate the training speed easily, we recorded the accuracy increasing along with the training epoch, as shown in Figure 8.5, Figure 8.6, and Figure 8.7.

Since MNIST is a relatively simple database, it is usually used to test whether the neural networks work. Although its accuracy is reduced by 0.65% compared with the exact LeNet,

Layer	Size
Input image	28x28x3
Conv.+ leaky ReLU	5x5x32
MaxPool.	2x2
Conv.+ leaky ReLU	5x5x64
Fully Connected+ReLU	1024
drop out	50%
Fully Connected	10

Table 8.3: LeNet network.

Layer	Size
Input image	28x28x3
Conv.+ leaky ReLU	3x3x32
MaxPool.	2x2
Conv.+ leaky ReLU	1x1x16
Conv.+ leaky ReLU	3x3x64
MaxPool.	2x2
Conv.+ leaky ReLU	1x1x32
Conv.+ leaky ReLU	3x3x128
Conv.+ leaky ReLU	1x1x64
Fully Connected+ReLU	1024
drop out	50%
Fully Connected	10

Table 8.4: mini_cifar network.

Architecture		LeNet-MNIST	LeNet-Cifar10	mini_cifar-Cifar10
Standard Network		99.06%	75.26%	77.30%
Approximate	170 epoch	98.42%		
	512 epoch		64.18%	71.46%
	2048 epoch		65.54%	72.89%
Transfer Learning	512 epoch		74.92%	77.01%
	1024 epoch		75.10%	77.26%

Table 8.5: Accuracy for different architectures and image sets.

the approximate LeNet achieves good accuracy. It proves that the approximate network can converge and be used for image classification.

Compared with MNIST, Cifar10 is more complex. Two networks are applied to Cifar10, with exact or approximate convolution. The accuracies of different networks are shown in Table 8.5. It can be seen that for more complex datasets, approximate networks have brought errors compared with exact networks. Figure 8.6 and Figure 8.7 show the accuracy along with training. After 512 epoch, the accuracies of standard networks have already stopped increasing, while the approximate networks are still slowly rising. On the one hand, this reflects the slower training of the approximate network. On the other hand, it also shows that the accuracy of the approximate network can be better with more training. In fact, we continue training the approximate networks until the 2048 epoch, in which the increase is much slower, the results are shown in Table 8.5. Continued training brings a related better result, but it did not change our conclusion, that is, compared with the exact network, the training of approximate network is slower, and depending on the dataset and network structure, the approximate network will bring different degrees of error.

8.3.3 Transfer Learning

Since MinConvNets are approximate to exact networks, we believe that the weights which have been well-trained on the exact networks can accelerate the convergence of the approximate network. Therefore, the transfer learning method is applied to train the approximate network.

The weights trained after 512 epoch in the standard network are used as the initial weights of the approximate network. Then the top-1 accuracy along with the training epochs are shown in Figure 8.6 and Figure 8.7. It can be seen from these figures that transfer learning can speed up the training of approximate networks. After 512 epoch training, the accuracy levels of the approximate network are almost the same as the exact networks. In addition, as shown in Table 8.5, with more training, such as 1024 epochs, the accuracy is closer to the standard network. In other words, there is negligible loss of accuracy in the approximate CNNs.

8.4 Comparison

Quantization reduces the consumption of each multiplication while pruning reduces the number of multiplications. However, but both are still necessary to calculate the multiplication.

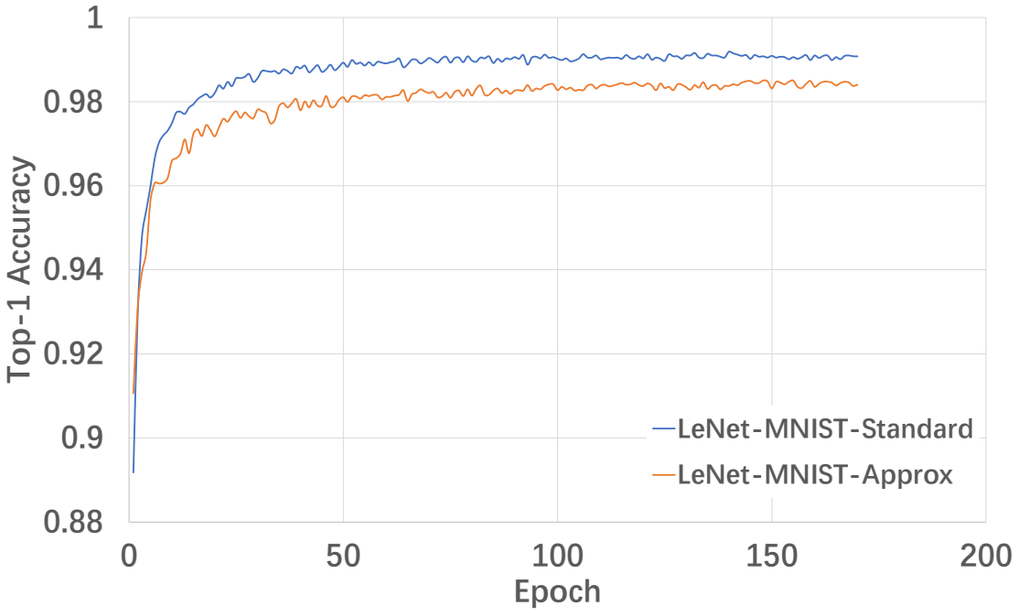


Figure 8.5: Top-1 accuracy of LeNet applied to MNIST.

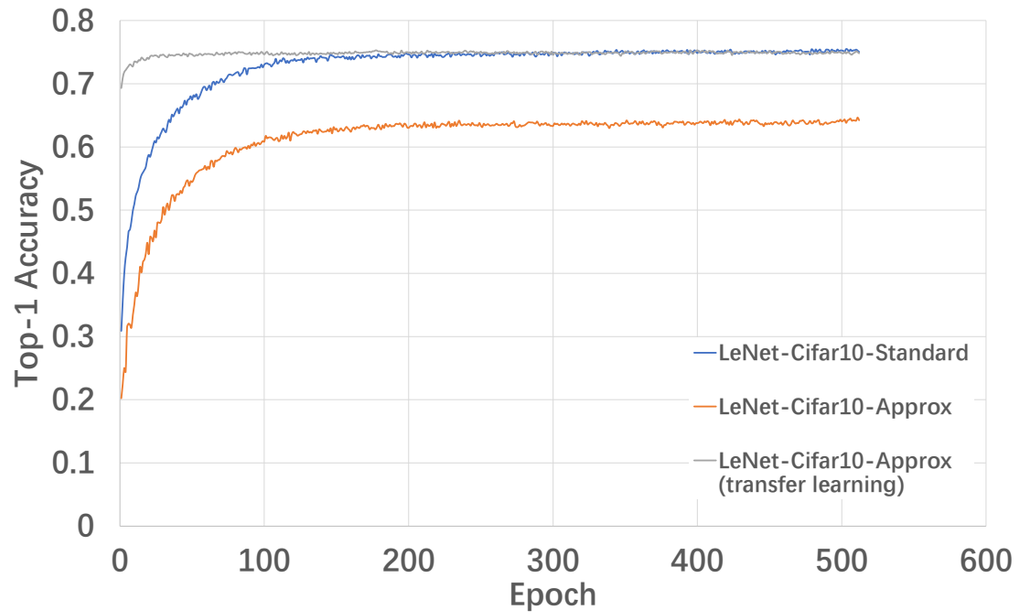


Figure 8.6: Top-1 accuracy of LeNet applied to Cifar10.

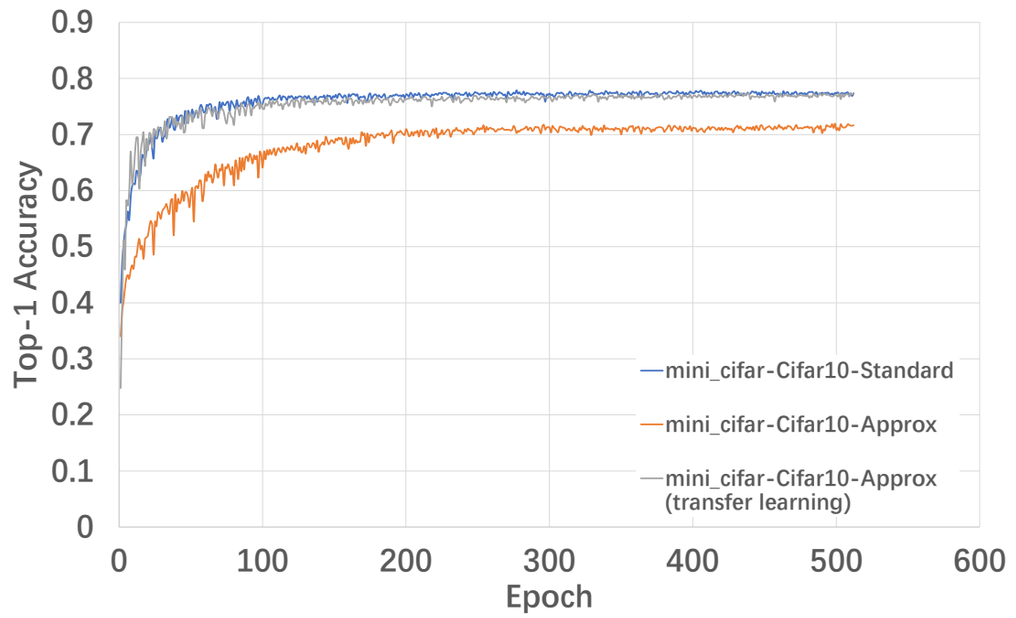


Figure 8.7: Top-1 accuracy of mini-cifar applied to Cifar10.

Different from these methods, we abandon multiplication directly. The comparison operation which is less time-consuming and easier implemented is proposed in our works to replace multiplication in CNNs.

Some works have also proposed multiplication-less networks. In [111], the weight is quantized to 1 bit, which allows multiplication to be converted into addition/subtraction. Furthermore, when the activation is also quantized to 1 bit, as introduced in [22, 14], multiplication is converted to XNOR logic gate. These methods fix one or more multipliers to ± 1, thereby simplifying the calculation. Moreover, there are also some networks that break away from multiplication. [83] uses the hit-and-miss transformation composed of comparison and addition to construct a network. Our work also uses comparison and addition operations, but the network with hit-or-miss transformation has brought a considerable loss of accuracy, while the networks in our work have not lost accuracy.

8.5 Conclusion

In order to speed up the CNNs, we reinterpret CNNs from the perspective of signals and systems. and propose a new structure named MinConvNets. In the convolutional layers of MinConvNets, the multiplication is approximated by an operation based on a minimum comparator, which is easier to be implemented or faster to be calculated. MinConvNets bring negligible loss of accuracy in the benchmark test.

Furthermore, the MinConvNets have shown that operations with errors can also generate good global inference results for CNNs, so that multiplication can be replaced by some simpler operation. This work shows that other operations can also extract features from images and provides a new research direction for the acceleration technologies for neural networks.

This work is still in progress and various improvements are parts of future research. First, the main contribution of this work is to propose a new architecture, but do not measure the runtime performance. The proposed approximate operation can be implemented in different platforms, and it is necessary to measure the runtime performance on these platforms. Next, other approximate calculators can be applied. Some other more valuable approximate calculators may bring greater acceleration and structural optimization. Furthermore, MinConvNets are still based on original neural networks, such as using the same training methods as original ones. Since other operations can also extract features, it is possible to get out of the original framework and propose a lighter image processing system in the future.

Chapter 9

Conclusions and Future work

CNN is one of the best machine learning algorithms. However, the huge parameter volume and complex calculations have brought challenges when CNNs are deployed on embedded systems. To overcome these shortcomings, the primary works described in this thesis put forward the optimization to convolutional neural networks on top of the related literature. On the one hand, the parameters of the neural network are compressed to make it less requiring on memory. On the other hand, the approximation applied to structure, operands, and operators, which makes the calculation faster on real-time platforms with limited computing resources. Next, we summarize these works and then provide a more speculative perspective on the subjects discussed in this thesis.

In Chapter 2, we introduced the history of machine learning, especially with the development of convolutional neural networks. The convolutional neural network has experienced rapid development several times. Until now, many classic CNN models are proposed. With the progress of technology, more accurate models, or models capable of handling more complex tasks have been proposed, but the excellent design of these classic models provides a reference for follow-up work. Meanwhile, a richer development framework also makes today's machine learning easier to develop and use. At the end of this thesis, we are incredibly pleased to say that artificial intelligence is ushering in a new era with the efforts of predecessors, while later generations will continue to write this story.

In Chapter 3, we discussed the new challenges of convolutional neural networks. For the embedded systems which have limited resources, the huge amount of calculation in CNNs brings difficulties on the calculation resource and communication bandwidth. According to the application scenario, we focused on the tasks as image classification and object detection, which are more likely to be deployed on the embedded platform. As well, we also introduced the runtime platforms for CNNs and determined our work scope after evaluating their performance. Chapter 4 has shown some state-of-the-art solutions for the introduced difficulties and Chapter 5 has implemented some of these solutions. Through analysis and experiment, we found that even if there is a lot of excellent works in this field, the technologies still have many shortcomings.

In Chapter 6, we proposed Selective Binarization. Naive binarization is a classical method that is widely discussed, but we have proposed a new possibility, that is, mixing binary and fully-precision layers. This method greatly reduces the amount of calculation thus improves the calculation speed. However, the essence of this method is the exchange of performance and accuracy, especially for more complicated tasks, such as object detection systems. it is a good choice for error-tolerant real-time systems. Nevertheless, for systems that require high precision, this method still has limitations.

Different from Selective Binarization, Quad-Approx proposed in Chapter 8 brings a lossless compression for object detection. Fewer-bit quantization is realized through proposed signed PACT methods. Then an approximate multiplier for 3 bits signed operands is applied to speed up the calculation. The contributions of this work are not only in compression and acceleration but also shows that the CNNs can accept incorrect calculation.

Since CNNs are fault-tolerant systems, approximate calculations can be applied to CNNs without compression by classic methods. In Chapter 8, we proposed a criterion to measure the approximation of two operations, and based on this criterion, an approximate operation to multiplication based on the comparator is proposed. Then, approximate convolutional layers in which the multiplication are replaced by the approximate operation have been built. And MinConvNets are constructed by approximate convolutional layers. Since the comparator is easier to be implemented than multiplication in the varied platform, MinCon-

vNets lighten the convolutional neural network.

The methods proposed in this thesis give answers to the question proposed in section 1.2. Selective Binarization optimize existing binary methods, Quad-Approx Network proposed a novel approximate computing method based on low-bit quantization, and MinConvNets proposed a new perspective to improve CNNs. Overall, these works reduce the resources required by CNNs and make them easier to be deployed on systems with limited resources, which bring a wider range of usage scenarios for CNNs and also make the embedded system more intelligent.

Throughout the whole thesis, the optimization method we discussed do something similar, that is, simplify the calculation within a tolerant loss of accuracy through different approximations methods: binarization, quantization, approximate operators, etc. Proposing more new methods is always valuable, but the criterion proposed in Chapter 8 seems to be more meaningful. This criterion can evaluate the approximate methods not only we have proposed and but also introduced in the state-of-the-art, whether the approximate computing is applied to the structure, the operands, or the operators. A satisfactory criterion can be applied to various approximation methods then measure the approximate level of different methods used in CNNs, so that the loss of accuracy by using the approximate methods can be predicted before the validation in the dataset. This greatly reduces the time required to optimize the CNNs. Therefore, an extremely attractive work in the future is to use the proposed criterion to evaluate the proposed approximation approach, not only in this works but also in the other classic methods as quantization and pruning, then to modify the criterion to better fit the experimental results, and finally to propose a more universal law. I believe that this will provide a new and useful tool for the approximate calculation of neural networks.

Appendix A

Scientific Production

The work carried out during this PhD led to the preparation of the following articles:

Published papers:

- Xuecan YANG, Sumanta Chaudhuri, Lirida Naviner, Laurence Likforman-Sulem, *"Quad-Approx CNNs for Embedded Object Detection Systems"*, The 27th IEEE International Conference on Electronics Circuits and Systems (ICECS 2020), Virtual Conference, Nov 2020
- Xuecan YANG, Sumanta Chaudhuri, Lirida Naviner, Laurence Likforman-Sulem, *"Accelerating CNNs on FPGAs with Selective Binarization"*, Journées Nationales du Réseau Doctoral en Micro-nanoélectronique (JNRDM 2019), Montpelier, Jun 2019
- Xuecan YANG, Sumanta Chaudhuri, Lirida Naviner, Laurence Likforman-Sulem, *"A streaming deep learning accelerator with selective binarization"*, Emerging Deep Learning Accelerators workshop (EDLA 2019), Valencia, Jan 2019
- Xuecan YANG, Sumanta Chaudhuri, Lirida Naviner, Laurence Likforman-Sulem, *"Object Detection with Embedded Machine Learning"*, International Workshop on Machine Learning & Artificial Intelligence, Paris, Sep 2018

In preparation paper:

- *"MinConvNets: A new class of multiplication-less Neural Networks"*,
- *Approximate operators applied to object detection systems*

Appendix B

Résumé étendu en Français

B.1 Introduction

Les réseaux neuronaux (NN) existent depuis les années 1940s, et les réseaux de neurones convolutifs (CNNs) ont également été utilisés avec LeNet [37]. Aujourd'hui, les réseaux de neurones convolutifs ont été largement utilisés dans de nombreux domaines tels que la reconnaissance d'image, le traitement vidéo et le traitement du langage naturel. Mais en revenant sur l'historique de développement des réseaux de neurones convolutifs, la percée majeure est présenté à 2012 quand le CNN AlexNet [5] a été proposé pour la classification d'images. Les principaux facteurs favorables à ce succès sont censés être l'apparition de grands ensembles de données dans le cloud et la disponibilité d'une énorme puissance de calcul (par exemple, sur GPU). C'est aussi parce que ces plates-formes de puissance de calcul sont largement utilisées que les réseaux de neurones deviennent de plus en plus grands [41, 1, 49, 2, 47].

En fait, les CNN sont toujours gourmands en calculs et en ressources, car ces calculs nécessitent un grand volume d'opérations de multiplication et d'accumulation (MAC). Mais dans certains cas, le cloud et le calculateur avec puissance n'est pas toujours disponible. Par exemples, on souhaite effectuer des opérations CNN sur des appareils informatiques de périphérie IoT. Donc les CNNs sont souvent limités par les performances et la mémoire limitées lorsqu'ils sont déployés sur des systèmes embarqués. Même si des services sur cloud sont disponibles, les CNNs plus légers, plus rapides et économes en énergie sont toujours plus favorables.

En considérant ces problèmes, ce projet de recherche doctorale vise à proposer des CNNs à faibles besoins en ressources informatiques et en mémoire, qui sont plus adaptés aux systèmes embarqués.

L'algorithme de réseau de neurones se compose généralement de deux phases, l'apprentissage et l'inférence. Les méthodes proposées dans cette thèse sont principalement utilisées pour accélérer la phase inférence. Les principale tâches de recherche sont la classification d'images et la détection d'objets dans les images, qui sont plus susceptibles d'être déployées sur des systèmes embarqués. Mais les méthodes d'optimisation proposés peuvent également être utilisées dans d'autres tâches réalisé par CNN.

B.2 State de l'Art d'Accélération et de Compression

Comme les CNNs deviennent de plus en plus complexes, certaines techniques d'optimisation sont progressivement valorisées. Les contributions de ces technologies d'optimisation sont d'accélérer la vitesse de calcul de CNN, ou de réduire la taille de CNN. Ils seront brièvement présentés dans ce chapitre.

B.2.1 Design Compact

Comparée à de nombreuses méthodes "avancées", la méthode "l'utilisation d'un design compact" est souvent négligée. Cependant, l'utilisation d'un design compact est en effet l'une des méthodes les plus efficaces et les plus simples à réaliser.

D'une part, les structures de CNN bien connues et largement utilisées sont généralement conçues sur certains ensembles de données spécifiques. Mais la tâche réelle peut être moins compliquée que ces ensembles de données. Sur cette base, une réduction appropriée de la

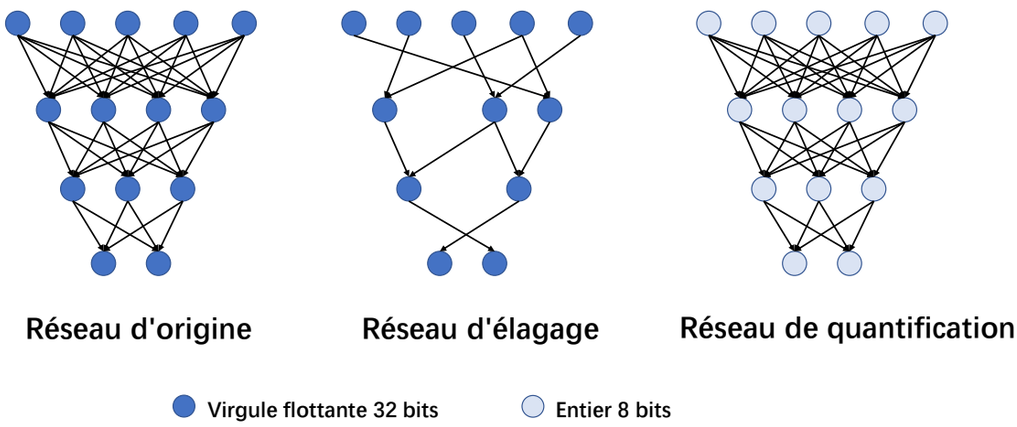


Figure B.1: Réseau d'élagage et Réseau de quantification

complexité de certains réseaux peut entraîner des performances de coût plus élevées. D'autre part, l'utilisation d'une série de plusieurs noyaux de convolution plus petits pour remplacer les plus grands peut non seulement réduire la quantité de calcul, mais également apporter plus de non-linéarité au réseau.

B.2.2 Réseaux de Quantification

La quantification est l'une des méthodes d'accélération les plus couramment utilisées dans le domaine industriel. L'idée principale de la quantification est d'utiliser les valeurs avec moins de nombres de bits pour remplacer des variables à virgule flottante plus précises.

Par exemple, des virgule flottante 16 bits et des entiers 8 bits dans TensorFlow [54] sont fournis pour remplacer les variables à virgule flottante 32 bits, et les variables à virgule fixe qui est plus court que virgule flottant peut apporter des calculs plus efficaces dans un FPGA [77].

En allant plus loin, certains travaillent à utilisent seulement 1 bit pour construire le réseau. Ces réseaux appelé réseau binaire convertit la multiplication en opérations logiques XNOR, accélérant ainsi le calcul. Mais ce type de réseau perd beaucoup de précision dans la tâche de la détection d'objets, et une solution va être proposés dans le chapitre suivant.

B.2.3 Réseau d'Elagage

Les CNNs sont généralement redondants. Par conséquent, la suppression de connections sans importance peut accélérer le calcul du réseau avec peu de perte de précision. Cette méthode s'appelle l'élagage.

Le réseau d'élagage est généralement divisé en deux catégories, l'élagage de la structure et l'élagage du poids. L'élagage de la structure consiste généralement à supprimer directement la connexion peu importante, et qui sont les filtres qui contribuent le moins dans une couche de convolution. Certains travaux sont proposés pour évaluer la contribution des filtres [15, 16, 81]. L'élagage du poids ne change pas la structure du réseau, mais les poids qui sont plus petits que le seuil sont directement mis à 0 grâce à cette méthode. Certains travaux discutent de la façon de mettre à 0 et la façon de trouver le seuil de l'élagage [18, 19, 20].

Certaines autres solutions sont également discutées dans cette thèse [84, 85, 86, 87, 87, 83]. Mais l'élagage et la quantification représenté dans la figure B.1 sont les deux directions les plus discutées en plus du design de la structure. La quantification est également le point de départ des travaux de cette thèse.

B.3 Selective Binarization

Le réseau binaire peut convertir la multiplication en opération logique XNOR, ce qui est plus convivial pour le matériel. Mais pour des tâches plus complexes, telles que la détection

d'objets, le réseau binaire entraînera une perte de précision. Par conséquent, nous proposons une structure hybride de CNN, qui est composée de couche HH, couche HB et couche BB.

- HH: des couches de convolution où l'activation et le poids sont à virgule flottante demi-précision.
- HB: des couches de convolution où l'activation est en virgule flottante demi-précision, mais le poids est binaire.
- BB: des couches de convolution où l'activation et le poids sont binaires.

Dans cette structure, une partie du CNN est binaire, et d'autres parties conservent toujours une précision en virgule flottante.

Évidemment, le binaire apporte une vitesse de calcul plus rapide, mais il entraîne également une perte de précision. Autrement dit, c'est un échange entre la vitesse et la précision. Et ce travail montre à travers des expériences que le taux d'échange est différent lorsque le binaire est utilisé dans différentes positions. Nous prenons tiny-YOLO comme exemple pour montrer la structure hybride avec le taux d'échange diffèrent en utilisant la binarisation dans la position différente. Nous appelons cette structure le Sélective Binarisation. La Sélective Binarisation combine des couches avec différentes précisions dans les CNNs pour obtenir une vitesse et une précision acceptables.

De plus, un accélérateur basé sur FPGA est proposé pour ces structures optimisées. Par rapport à l'accélérateur du réseau original, l'accélérateur pour Sélective Binarisation peut traiter différentes structures.

- HH: Des multiplicateurs et des additionneurs sont utilisé pour les multiplications-et-accumulations (MACs).
- HB: Les valeurs de poids $w \in \{\pm 1\}$. Ainsi, les MACs sont convertis en addition (si $w = 1$) ou soustraction (si $w = -1$).
- BB: XNOR porte logique sont utilisés pour multiplication binaire. Dans le même temps, comme 1bit utilise moins de bande passante et XNOR porte logique utilise moins de ressources, couche BB peut être mieux organisée en parallèle, de sorte que sa vitesse de calcul est plus rapide.

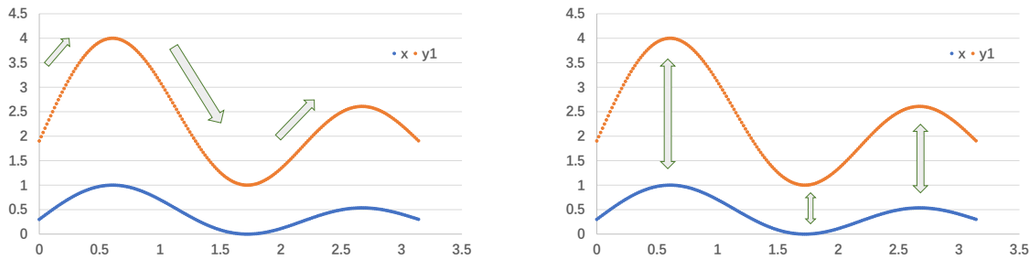
Enfin, nous utiliserons le binaire pour les 7ème et 8ème couches de tiny-YOLO Network, puis on l'applique sur des images de détection d'objet de drone (DAC-SDC). Il est possible d'obtenir une amélioration de 1,68x des performances ce qui entraîne une perte de précision tolérable de 8,99 % mesurée par IOU (Intersection over Union).

B.4 Quad-Approx Networks

Bien que le binaire accélère les calculs, il entraîne toujours une perte de précision dans les tâches complexes. Dans ce cas, nous pouvons utiliser 2 bits ou plus pour la quantification. Différent du binaire utilisant uniquement des symboles (positive ou négative) pour quantifier les paramètres, quand il s'agit de la quantification des valeurs, certains problèmes se posent.

- Les paramètres du réseau sont non-uniforme distribués. Dans la quantification d'entiers traditionnelle, des paramètres relativement petits peuvent être mappés à des entiers relativement petits. Mais dans la quantification à faible bit, de nombreux petits paramètres sont directement mappés à 0. Cela a entraîné la perte de nombreuses informations.
- La quantification traditionnelle nécessite le calcul de la plage d'activation, ce qui signifie qu'il faut attendre la fin de tous les calculs pour commencer la quantification. Cela viole évidemment le principe de la conception du pipeline, et réduit la vitesse de calcul.

L'approche PArameterized Clipping acTivation Function signé (PACT signé) est proposée dans notre travail. Avec PACT signé, les valeurs trop grandes sont directement saturées et mappé à paramètre configuré (seuil paramètre). De cette manière, les activations sont limitées à une plage claire. Dans le même temps, étant donné que des nombres trop grands sont supprimés, les petites valeurs peuvent être mappées à des valeurs non nulles. Avec



(a) La tendance des changements

(b) La distance des valeurs

Figure B.2: L'approximation de deux signaux aléatoires

PACT signé, CNN peut être quantifié en utilisant 2 bits (non-signé) ou 3 bits (signée), ce qui permet au moins une 10.6x compression par rapport aux réseaux 32 bits.

Bien qu'il existe de nombreuses calculatrices de multiplication à faible bit, ces multiplicateurs sont encore compliqués. Par conséquent, nous proposons d'utiliser des calculatrices approximatives au lieu de calculatrices précises. Ce calcul approximatif utilise moins de ressources pour calculer la multiplication à 2 bits, mais donne des résultats inexacts : le résultat de 3 fois 3 est 7 au lieu de 9. Bien que certains calculs soient inexacts, après un entraînement avec la méthode proposée, il n'y a pas de perte de précision dans l'inférence finale pour des tâches telles que la classification et la détection d'objets. Nous avons construit un réseau quantitatif qui utilise moins de ressources de calcul, mais ne réduit pas la précision. Ce réseau s'appelle Quad-Approx Network.

Un 3 bits multiplicateur approximatif implémenté dans FPGA est proposé, qui peut atteindre une accélération de 1,2x et une compression de 5,3x lorsqu'il est appliqué aux Quad-Approx Networks.

En plus de l'accélération, il est plus précieux que Quad-Approx montre que les CNNs sont des systèmes de tolérance aux pannes, ce qui nous conduit à proposer les MinConvNets.

B.5 MonConvNets

Le calcul de multiplication nécessite souvent des ressources spéciales, telles que Processeur de signal numérique (DSP), sinon le calcul sera très lent. L'une des principales raisons du calcul lourd de CNN est qu'il existe de nombreuses multiplications. Par conséquent, nous continuons à adopter l'idée Quad-Approx et à utiliser des calculs de multiplication approximative qui est plus légers mais inexacts au lieu de calculs de multiplication précis.

Nous voulons trouver des opérateurs approximatifs de multiplication, mais il est souvent difficile de définir l'approximation des opérateurs. Afin d'évaluer l'approximation des deux opérateurs, nous modélisons les opérations à deux systèmes qui ont les mêmes signaux entrés. Lorsque le signal de sortie de ces deux systèmes sont plus approximatives, les deux opérateurs sont plus approximatifs.

Il y a nombreuses méthodes pour évaluer l'approximation des signaux. Nous proposons ici deux critères d'évaluation :

- La tendance des changements illustrée à la Figure B.2a : des signaux approximatives devraient avoir des tendances de changement similaires, qui est mesuré par le coefficient de cohérence dans notre proposition.

- La distance des valeurs illustrée à la Figure B.2b : la valeur absolue de deux signaux approximatifs doit être aussi proche que possible. Nous utilisons l'erreur relative pour le mesurer.

Nous espérons trouver une opérations *op* approximative à multiplication : lorsque des variables aléatoires ξ, η sont données comme entrées de système, sa sortie $y_i = op(\xi, \eta)$ est approximative à la sortie de la multiplication $x = \xi \cdot \eta$. C'est-à-dire qu'elle a une tendance similaire et une distance relativement petite entre le signal y_i et z. Après l'analyse, nous avons constaté que min-selector $y_{min} = min(\xi, \eta)$ est approximative à la multiplication dans certaines contraintes. Nous déformons le CNN d'origine pour respecter ces contraintes, puis utilisons les opérateurs approximatifs pour remplacer les multiplications dans CNN et puis établir MinConvNet.

MinConvNet peut obtenir une perte de prédiction négligeable par rapport aux réseaux de classification d'image exacte grâce à l'apprentissage par transfert, tandis que la multiplication difficile à mettre en œuvre est remplacée par des opérations plus faciles à implémenter. D'une part, MinConvNets a abandonné l'idée de compression traditionnelle et a proposé une nouvelle direction pour l'accélération des CNNs. D'une autre part, avec le critère proposé dans ce travail, il peut être possible d'évaluer une méthode de compression ou accélération sans validation sur des ensembles d'images dans le futur, ce qui peut grandement faire gagner du temps. Par conséquent, même si MinConvNet n'a pas été implémenté dans une plate-forme d'exécution telle que la Selective Binarization et les Quad-Approx Networks, c'est toujours un travail très intéressant.

B.6 Conclusion

D'une part, l'humain inaugure l'ère de l'intelligence artificielle. D'un autre côté, l'Internet des Objets (IoT) nous facilite la vie. Cependant, dans de nombreux cas, les réseaux d'intelligence artificielle sont très complexes et lourds, ce qui pose des défis aux applications d'IoT, car l'IoT est généralement mis en œuvre sur des plates-formes embarquées avec des ressources limitées, telles qu'une mémoire limitée et des ressources informatiques limitées.

Les travaux de cette thèse ont proposé les méthodes d'optimisation dans les aspects système, opérandes et opérateurs, afin de répondre à l'une des questions importantes des CNN, à savoir comment faciliter le déploiement des CNN consommateurs de ressources sur les plates-formes à ressources limitées. Ces travaux apportent des algorithmes intelligents plus complexes dans les appareils de périphérie et nous aident à créer l'ère de l'Internet des objets artificiel et intelligent (AIoT).

Bibliography

[1] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[3] Adrien Blanchardon, Roselyne Chotin-Avot, and Habib Mehrez. Générateur d'architecture de fpga. In *Colloque GDR SOC-SIP*, pages 1–3, 2012.

[4] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

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

[6] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*, pages 109–116, June 2011.

[7] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[9] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks. *CoRR*, abs/1807.07928, 2018.

[10] Norman P. Jouppi et. al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[11] Jean-Pierre Deschamps, Gustavo D Sutter, and Enrique Cantó. *Guide to FPGA implementation of arithmetic functions*, volume 149. Springer Science & Business Media, 2012.

[12] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[13] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017.

[14] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[15] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[16] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–18, 2017.

[18] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[19] Suraj Srinivas, Akshayvarun Subramanya, and R Venkatesh Babu. Training sparse neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 138–145, 2017.

[20] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. Learning sparse neural networks via sensitivity-driven regularization. In *Advances in neural information processing systems*, pages 3878–3888, 2018.

[21] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[22] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[23] Pamela McCorduck and Cli Cfe. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press, 2004.

[24] Daniel Crevier. *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, Inc., 1993.

[25] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[26] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[27] Geoffrey E Hinton, Terrence Joseph Sejnowski, Tomaso A Poggio, et al. *Unsupervised learning: foundations of neural computation*. MIT press, 1999.

[28] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[29] Jeremy West, Dan Ventura, and Sean Warnick. Spring research presentation: A theoretical foundation for inductive transfer. *Brigham Young University, College of Physical and Mathematical Sciences*, 1(08), 2007.

[30] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[31] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[32] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, 2019.

[33] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[35] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[36] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[37] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a backpropagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[38] Adit Deshpande. The 9 deep learning papers you need to know about (understanding cnns part 3). *adeshpande3. github. io. Retrieved*, pages 12–04, 2018.

[39] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. In *International Conference on Machine Learning*, pages 584–592, 2014.

[40] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

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

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

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

[44] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[45] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[46] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016.

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

[48] Ross Girshick. Fast r-cnn. *arXiv preprint arXiv:1504.08083*, 2015.

[49] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[50] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

[51] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[52] Nvidia cuda home page, 2020.

[53] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Opencl as a unified programming model for heterogeneous cpu/gpu clusters. *ACM SIGPLAN Notices*, 47(8):299–300, 2012.

[54] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[55] Serdar Yegulalp. Facebook brings gpu-powered machine learning to python. *InfoWorld*, 19, 2017.

[56] Nikhil Ketkar. Introduction to pytorch. In *Deep learning with python*, pages 195–208. Springer, 2017.

[57] Darknet: Open Source Neural Networks in C, Last accessed 13/01/2019.

[58] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[59] Junior Dongo, Ludovic Foltete, Charif Mahmoudi, and Fabrice Mourlin. Distributed edge solution for iot based building management system with ndn. In *2019 Global Information Infrastructure and Networking Symposium (GIIS)*, pages 1–5. IEEE, 2019.

[60] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[61] Luca Bertinetto, Jack Valmadre, Joao F Henriques, Andrea Vedaldi, and Philip HS Torr. Fully-convolutional siamese networks for object tracking. In *European conference on computer vision*, pages 850–865. Springer, 2016.

[62] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[63] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[64] Xiaowei Xu, Xinyi Zhang, Bei Yu, X Sharon Hu, Christopher Rowen, Jingtong Hu, and Yiyu Shi. Dac-sdc low power object detection challenge for uav applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[65] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[66] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

[67] Johannes Romoth, Mario Porrmann, and Ulrich Rückert. Survey of fpga applications in the period 2000–2015 (technical report). 2017.

[68] Khadija Bousmar, Fabrice Monteiro, Zineb Habbas, Sofiene Dellagi, and Abbas Dandache. A new fpga-based dpll algorithm to improve sat solvers. In *2015 27th International Conference on Microelectronics (ICM)*, pages 287–290. IEEE, 2015.

[69] Karl Pauwels, Matteo Tomasi, Javier Diaz Alonso, Eduardo Ros, and Marc M Van Hulle. A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers*, 61(7):999–1012, 2011.

[70] Pooja Jawandhiya. Hardware design for machine learning. *International Journal of Artificial Intelligence and Applications (IJAIA)*, 9(1):63–84, 2018.

[71] Umer Farooq, Zied Marrakchi, and Habib Mehrez. Fpga architectures: An overview. *Tree-based heterogeneous FPGA architectures*, pages 7–48, 2012.

[72] Vivado design flow overview, 2020.

[73] Pynq z1 reference manual, 2020.

[74] Yingying Zhang, Desen Zhou, Siqin Chen, Shenghua Gao, and Yi Ma. Single-image crowd counting via multi-column convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 589–597, 2016.

[75] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[76] Roselyne Chotin and Habib Mehrez. A floating-point unit using stochastic arithmetic compliant with the ieee-754 standard. In *9th International Conference on Electronics, Circuits and Systems*, volume 2, pages 603–606. IEEE, 2002.

[77] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[78] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[79] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[80] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[81] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[82] Rafael C Gonzalez and Richard E Woods. Digital image processing. 2002. *Google Scholar Google Scholar Digital Library Digital Library*, 2007.

[83] Muhammad Aminul Islam, Bryce Murray, Andrew Buck, Derek T Anderson, Grant J Scott, Mihail Popescu, and James Keller. Extending the morphological hit-or-miss transform to deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

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

[85] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

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

[87] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[88] Axel Laborieux, Marc Bocquet, Tifenn Hirtzlin, J-O Klein, L Herrera Diez, Etienne Nowak, Elisa Vianello, J-M Portal, and Damien Querlioz. Low power in-memory implementation of ternary neural networks with resistive ram-based synapse. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 136–140. IEEE, 2020.

[89] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity*. Cambridge university press, 2002.

[90] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[91] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

[92] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[93] Tifenn Hirtzlin, Bogdan Penkovsky, Marc Bocquet, Jacques-Olivier Klein, Jean-Michel Portal, and Damien Querlioz. Stochastic computing for hardware implementation of binarized neural networks. *IEEE Access*, 7:76394–76403, 2019.

[94] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[95] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 31–40. ACM, 2018.

[96] Jiaolong Xu, Peng Wang, Heng Yang, and Antonio M López. Training a binary weight object detector by knowledge transfer for autonomous driving. *arXiv preprint arXiv:1804.06332*, 2018.

[97] Zynq-7000 AP SoC Family Product Tables and Product Selection Guide - zynq-7000-product-selection-guide.pdf, September 2017.

[98] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Sjalander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *Field Programmable Logic and Applications (FPL), 2018 28th International Conference on*, FPL '18, 2018.

[99] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and efficient 2-bit quantized neural networks. In *Proceedings of the 2nd SysML Conference*, volume 2019, 2019.

[100] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for research*, volume 512. John Wiley & Sons, 2011.

[101] I Bellido and Emile Fiesler. Do backpropagation trained neural networks have normal weight distributions? In *International Conference on Artificial Neural Networks*, pages 772–775. Springer, 1993.

[102] Marcus Gallagher and Tom Downs. Weight space learning trajectory visualization. In *Proc. Eighth Australian Conference on Neural Networks, Melbourne*, pages 55–59, 1997.

[103] Jinwook Go and Chulhee Lee. Analyzing weight distribution of neural networks. In *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No. 99CH36339)*, volume 2, pages 1154–1157. IEEE, 1999.

[104] Lior Deutsch, Erik Nijkamp, and Yu Yang. A generative model for sampling high-performance and diverse weights for neural networks. *arXiv preprint arXiv:1905.02898*, 2019.

[105] Neale Ratzlaff and Li Fuxin. Hypergan: A generative model for diverse, performant neural networks. *arXiv preprint arXiv:1901.11058*, 2019.

[106] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[107] Muhammad Atta Othman Ahmed. Trained neural networks ensembles weight connections analysis. In *International Conference on Advanced Machine Learning Technologies and Applications*, pages 242–251. Springer, 2018.

[108] Mansfield Merriman. *A List of Writings Relating to the Method of Least Squares: With Historical and Critical Notes*, volume 4. Academy, 1877.

[109] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[110] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[111] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. A lightweight yolov2: A binarized CNN with A parallel support vector regression for an FPGA. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*, pages 31–40, 2018.

[112] Saralees Nadarajah and Samuel Kotz. Exact distribution of the max/min of two gaussian random variables. *IEEE Transactions on very large scale integration (VLSI) systems*, 16(2):210–212, 2008.

[113] Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009.

[114] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[115] Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[116] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[117] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[118] Peng Li and David J Lilja. Using stochastic computing to implement digital image processing algorithms. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 154–161. IEEE, 2011.

[119] H. Yonekawa and H. Nakahara. On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an fpga. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 98–105, May 2017.

[120] FPGA-based neural network inference, Last accessed 13/01/2019.

[121] Zynq UltraScale+ MPSoC Product Tables and Product Selection Guide - zynq-ultrascale-plus-product-selection-guide.pdf, September 2017.

[122] DAC-HDC-2018, September 2017.

[123] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-bnn: Binarized neural network on fpga. *Neurocomputing*, 275:1072–1086, 2018.

[124] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy. Scalable high-performance architecture for convolutional ternary neural networks on fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, September 2017.

[125] Luca Bertinetto, Jack Valmadre, João F. Henriques, Andrea Vedaldi, and Philip H. S. Torr. Fully-convolutional siamese networks for object tracking. *CoRR*, abs/1606.09549, 2016.

[126] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[127] David Held, Sebastian Thrun, and Silvio Savarese. Learning to track at 100 FPS with deep regression networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, pages 749–765, 2016.

[128] Laurence Likforman Xuecan Yang, Sumanta Chaudhuri and Lirida Naviner. A streaming deep learning accelerator with selective binarization. In *EDLA 2019 WORKSHOPS*, January 2019.

[129] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[130] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[131] Miguel de Prado, Maurizio Denna, Luca Benini, and Nuria Pazos. Quenn: Quantization engine for low-power neural networks. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 36–44. ACM, 2018.

[132] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *arXiv preprint arXiv:1703.03073*, 2017.

[133] Wenjia Meng, Zonghua Gu, Ming Zhang, and Zhaohui Wu. Two-bit networks for deep learning on resource-constrained embedded devices. *arXiv preprint arXiv:1701.00485*, 2017.

[134] Yaman Umuroglu, Davide Conficconi, Lahiru Rasnayake, Thomas Preusser, and Magnus Sjalander. Optimizing bit-serial matrix multiplication for reconfigurable computing. *ACM Transactions on Reconfigurable Technology and Systems*, 2019.

[135] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 1–7, 2016.

[136] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 145–150, 2016.

[137] Thommen George Karimpanal and Roland Bouffanais. Self-organizing maps for storage and transfer of knowledge in reinforcement learning. *Adaptive Behavior*, 27(2):111–126, 2019.

[138] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

ECOLE
DOCTORALE

Titre : Calcul Approximatif pour l'Apprentissage Automatique Embarqué

Mots clés : Calcul Approximatif, Apprentissage Automatique, Système Embarqué

Résumé : Les réseaux de neurones convolutifs (CNN) ont été largement utilisés dans de nombreux domaines tels que la reconnaissance d'image, le traitement vidéo et le traitement du langage naturel. Cependant, les CNN sont toujours gourmands en calculs et en ressources. Ils sont souvent limités par les performances et la mémoire limitées lorsqu'ils sont déployés sur des systèmes embarqués. Ce projet de recherche doctorale vise à proposer des CNNs à faibles besoins en ressources informatiques et en mémoire, qui sont plus adaptés aux systèmes embarqués.
En plus de la revue de la littérature, trois méthodes pour accélérer les CNNs sont proposées : Selective Binarisation, Quad-Approx Networks et MinConvNets :
La Selective Binarisation combine des couches avec différentes précisions dans les CNNs pour obtenir une vitesse et une précision acceptables. De plus, un accélérateur basé sur FPGA est proposé pour ces structures optimisées.
Avec le PArameterized Clipping acTivation Function signé proposé (signed PACT), les CNN sont quantizées en 3 bits, puis le multiplicateur approximatif est utilisé pour construire un réseau sans perte les précisions de détection, appelé Quad-Approx Network. En plus de l'accélération, il est plus précieux que Quad-Approx montre que les CNN sont des systèmes de tolérance aux pannes, ce qui nous conduit à proposer les MinConvNets.
MinConvNet est un ensemble de CNN sans multiplication dont la multiplication est remplacée par une opération approximative. MinConvNet peut obtenir une perte de prédiction négligeable par rapport aux réseaux de classification d'image exacte grâce à l'apprentissage par transfert, tandis que la multiplication difficile à mettre en œuvre est remplacée par des opérations plus faciles à implémenter.
D'une part, l'humain inaugure l'ère de l'intelligence artificielle. D'un autre côté, l'Internet des objets (IoT) nous facilite la vie. Ces travaux apportent des algorithmes intelligents plus complexes dans les appareils de périphérie et nous aident à créer l'ère de l'Internet des objets artificiel et intelligent (AIoT).

Title : Approximate Computing for Embedded Machine Learning

Keywords : Approximate Computing, Machine Learning, Embedded System

Abstract : Convolutional Neural Networks (CNNs) have been extensively used in many fields such as image recognition, video processing, and natural language processing. However, CNNs are still computational-intensive and resource-consuming. They are often constrained by the limit performance and memory when deployed on embedded systems. This PhD research project aims at proposing CNNs which are more suitable for embedded systems with low computing resources and memory requirements.
Based on literature review, we propose three methods to accelerate the operation of neural networks : Selective Binarization, Quad-Approx Network and MinConvNets.
Selective Binarization combines layers with different precisions in CNNs to achieve an acceptable speed and accuracy. As well an FPGA based hardware accelerator is proposed for these optimized structures.
With the proposed signed PArameterized Clipping acTivation Function (signed PACT), the CNNs are quantized into 3 bits, and then a loss-less network is established by using approximate multiplier, which is named Quad-Approx Network. In addition to acceleration, what is more valuable is that Quad-Approx shows that CNNs are certain fault tolerance systems, which leads us to propose the MinConvNets.
MinConvNet is a set of multiplication-less CNNs whose multiplications are replaced by approximate operations. MinConvNet can achieve negligible loss of prediction compared to exact image classification networks through transfer learning, meanwhile the multiplication which is more resource consuming to implement is replaced by easier implemented operations.
Human is ushering the era of the artificial intelligence. In the meantime, the Internet of Things (IoT) makes our lives more convenient. These works bring more complex intelligent algorithms into the edge devices and helps us to create the era of Artificial intelligent Internet of Things (AIoT).

Institut Polytechnique de Paris
91120 Palaiseau, France