A HARDWARE-AWARE NEURAL NETWORK WITH A LOOK-UP TABLE DECOMPOSITION ALGORITHM

XUECHEN ZANG

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF KITAKYUSHU

2021

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Xuechen Zang
July 2021

Acknowledgments

I would like to thank my supervisor, Prof.Nakatake. During the five years from the entrance of master's degree to the graduation of PhD, he always kept teaching me attentively and provided a lot of guidance in the process of research.

I am grateful to Dr.Bo Liu, Dr.Chao Geng, and Dr.Xuncheng Zou. Thanks to their kindness and support during the days and months I spent together in the research lab, I was able to be enlightened and start again many times when I faced difficulties.

Thanks to my mom and dad, they have always been my solid backing, and have tolerated my vulnerabilities during my PhD and given me the strength to persevere.

Thanks to my girlfriend, Yihe. Throughout the long experience of a foreign relationship, she endured the stress and trials that could have been avoided and chose to believe in me and get through it together. With the utmost tenderness, she has always kept my life light and warm.

Finally, despite the bumps in the road, I thank myself for being so fortunate to experience all of this.

Abstract

Neural networks are one of the most rapidly developing machine learning techniques in recent years and have been used with impressive success in a wide range of fields, especially in electronic design automation. Neural networks are not only involved in the design, optimization and implementation of logic circuits to improve efficiency, but have also been successfully implemented on various mobile hardware platforms. However, the high computational cost of neural networks, the large difference in computational accuracy with hardware, and the low structural similarity are often obstacles to be overcome in research.

This thesis presents the knowledge and research development on the cross-application of neural networks and logic circuits, and divides the main related research into three chapters, which are as follows. (1) An experimental procedure on how to implement the logic of multiple lookup tables, a common basic unit in logic circuits, by employing neural networks in a function-fitting manner is presented. The accuracy advantages and applicability of neural networks are demonstrated by comparing them with the traditional machine learning method, polynomial regression, the tuning of neural network parameters and the comparison of results. 2) An approximate decomposition method is introduced, focusing on decomposing a larger size look-up table (LUT) into smaller individuals. A depth-first search divide-and-conquer algorithm is used to search for the best decomposition scheme and generate approximate LUTs with an acceptable error tolerance to make it easier to implement on memory cell-based logic devices. 3) A novel neural network is introduced that incorporates the structural features of recently proposed memory-based programmable logic devices. A hardware-aware structure, sparser connections and a smaller weight matrix are used and can provide near full precision performance and acceptable binary precision performance.

In summary, the cost of implementing bidirectional interaction between neural networks and logic circuits can be effectively reduced by benefiting

from the logic learning capability of neural networks, the decomposition method of large-size LUTs, and the hardware-aware structure of neural networks.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Benefiting from the rapid growth of hardware computing performance in accordance with Moore's Law, many research fields that were once considered to require too much computing power to be applied have been revived with new vigor. Machine learning is one of these fields. Deep learning based on neural networks is one of the most highly regarded technologies.

As the most rapidly developing machine learning techniques in recent years, deep neural network and reinforcement learning have been used in a wide range of fields [SM19] [LAH+21] [PSY+18] [RS20] and have achieved performance as good as or better than that of human experts [SSS+17]. Deep learning and reinforcement learning have been used to optimize the design of logic circuits [HHH+21] and their own implementation on hardware platforms [SCYE17] [YAL20] [CBM+20] [BVM+19], not only at the software level for logical judgments and interactions, but also in the hardware domain, with many impressive results.

In the foreseeable future, the design and optimisation of logic circuits will become even more automated and intelligent thanks to further developments in machine learning technology.

1.1 The Development of Deep Neural Networks

In recent years we have seen exciting breakthroughs in the core problems of machine learning, driven largely by advances in deep neural networks. At the same time, the amount of data collected across a wide range of scientific domains is increasing dramatically in both size and complexity. Overall, this suggests that there are many exciting opportunities for the application of deep learning.

Neural networks are the core technology of deep learning. From classical single-layer perceptrons to multi-layer perceptrons, and then the introduction of activation

functions and back propagation lead to the emergence of standard deep neural networks. Subsequently, the variety of neural networks has become more and more abundant. The first proposed for image classification was LeNet [LBBH98] in 1989. AlexNet [KSH12], proposed in 2012, revived researchers' interest in convolutional neural networks with classification accuracy that were significantly better than other methods at the time. This was followed by VGG [SZ15], which introduced residual learning and a jump connection mechanism by stacking more convolutional layers and smaller convolutional kernels, and ResNets [HZRS15], which strongly addressed the degradation problem caused by the excessive depth of the network. DenseNets [HLvdMW18] differ from standard neural networks in that the network is divided into several 'blocks', with each layer in a 'block' being connected to every other layer. More recent complex models include ResNeXt [XGD+17] and, more recently, EfficientNets [TL20], which has separate scaling factors for network depth, width and spatial resolution of the input image. Other architectures, such as Long Short-Term Memory(LSTM) [HS97] based recurrent neural networks [CGCB15] [GDG+15] [ZSV15], spiking neural networks [TGK+19] and other structures have also been researched and developed individually.

Despite the outstanding learning ability, deep neural networks also often suffer from the trouble of being too large. By representing the network weights with very low precision in order to reduce the storage space and computational complexity occupied by the network. Binarization is the ultimate expression of this idea. As pioneering work on binary deep neural networks, binary neural network(BNN) and XNOR-Net have demonstrated the effectiveness of binarization and enjoy many hardware-friendly features, including memory savings, energy efficiency and significant speed-ups [QGL+20] [SL19].

Driven by the rapid increase in available data and computing resources, deep learning is excelling in many tasks, including speech recognition, image classification, natural language processing and more.

1.2 Deep Learning for Approximate Logic Synthesis

Logical synthesis is an optimisation problem with complex constraints that require exact solutions. Therefore, it is difficult to generate logic synthesis solutions directly using deep learning algorithms, but deep learning can give help with the logic transformation step in logic synthesis. In current synthesis tools, such as ABC [BM10], many logic

transformations exist.

Some previous work has focused on the use of approximation calculations in EDA. One type of research focused on exploring the importance of structural features of specific circuits and their output bits to obtain performance improvements through manual approximations [GMP+11] [KGE11]. As circuits become more complex in terms of functionality, another class of research applying automated methods is emerging, such as heuristics based on Karnaugh graph optimisation principles [SG10], synthesising approximate circuits under a given error constraint [VSK+12], synthesising approximate circuits using Boolean matrix decomposition [HTR18] and so on.

However, deep learning methods have shown great potential for generating high quality solutions to NP-complete problems, saving significant time and resource consumption compared to traditional methods. Compared to traditional methods that typically solve each problem from the beginning, deep learning methods extract high-dimensional features or patterns that can be reused in other related or similar situations, thus avoiding a great deal of potential duplication and waste. Thus, for accelerating the solution of electronic design automation(EDA) problems, there are some studies using deep learning methods to strengthen existing traditional optimization strategies.

LSOracle [NAT+19] relies on DNN to dynamically decide which optimizer should be applied to different parts of the circuit. Another work proposed by Yu et al. [YXM19] design a process of employing a CNN to map a synthesis flow to quality of results (QoR) levels to predict the synthetic streams likely to produce the optimal QoR.

Reinforcement learning is also used for approximate logic synthesis. By modelling the transition between two directed acyclic graphs with the same I/O behaviour as a single action, ALS frameworks based on different algorithms and models has been proposed, such as based on Graph convolutional nerual network [HCS+18], advantage actor critic agent [HHSR19] and Q-learning [PNP19].

1.3 Neural Networks Implementations on Chip

While neural networks have powerful inference capabilities, their high computational complexity and storage footprint pose significant barriers to their implementation. General CPU platforms are not good at performing huge amounts of parallel matrix operations. GPU platforms, while offering high computational power and an easy-to-use development framework, have the disadvantage of being expensive. FPGA and similar memory based programmable logic devices are the next possible competitive

solutions. With specific optimisation measures, overtaking GPU in terms of speed and energy efficiency becomes possible.

Various deep neural networks FPGA-based accelerator designs have been proposed [WGDL21] [SSEM19]. The BNN family of deep neural networks has gained more attention and extended research as lower precision neural networks can run more efficiently on hardware platforms, such as A batch normalization free binary CNN for FPGA [YN17], A accelerator for BNN on FPGA [ZSZ+17] and binary CNN on binary RRAM device [YLC+16]. In addition, researchers have also worked to reduce the cost of their implementation on hardware devices by adjusting the network architecture [GCK19] [SHY20].

Driven by the trend towards the ubiquitous use of deep neural networks, there is an opportunity for networks that are more intimate with the hardware architecture and more cost-effective implementations to become the focus of researchers' attention.

1.4 Organization of the Thesis

The organization of the thesis is as following:

- Chapter 2 introduces the development of neural networks, the basic principles and common optimization methods. One of the most generalized variants, convolutional neural networks, is introduced. In addition, this chapter introduces low-precision binary neural networks that are easier to implement in hardware. Finally, knowledge of programmable logic devices is introduced, including basic FPGAs, look-up tables, and a reconfigurable logic device proposed in recent years.
- Chapter 3 describes the experimental procedure of using neural networks to implement memory unit logic in a function-fitting manner. We compare deep neural networks with a traditional machine learning method, polynomial regression, and verify the accuracy advantages of the neural network approach.
- Chapter 4 presents an approximate decomposition method for decomposing large lookup tables (LUTs) into smaller combinations of lookup tables. The method uses a divide-and-conquer algorithm based on depth-first search to find the best decomposition scheme and generate approximate LUT combinations with acceptable error tolerances. The smaller sub-LUTs obtained after decomposition will be easier to implement on memory cell-based logic devices.

- Chapter 5 introduces a novel neural network that incorporates the structural features of recently proposed memory-based programmable logic devices. Based on hardware-like structure, sparser connections and smaller weight matrices, on common less complex datasets, the novel network can provide near full precision performance and acceptable binary precision performance.
- Chapter 6 summarizes the thesis and describes the combined impact of the research results.

Chapter 2

Preliminary

This chapter presents the necessary prior knowledge covered in the thesis.

- Sec.2.1: the introduction of the development of neural network.
- Sec.2.2: the fundamentals of generic feed-forward neural networks: design origins, model components and principles, non-linear activation functions, back propagation and gradient descent principles for parameter updating.
- Sec.2.3: convolutional neural network.
- Sec.2.4: binary neural networks.
- Sec.2.5: various methods used for neural network optimization such as mini-batch gradient descent, batch normalization, and adaptive learning rates, etc..
- Sec.2.6: field-programmable gate array and memory based reconfigurable logic device.

2.1 Development of Neural Network

Inspired by the nervous system of the human brain inspired by the human brain's nervous system, early neuroscientists constructed a mathematical model that mimics the human brain's nervous system, called an artificial neural network(ANN), or neural network(NN) for short. In the field of machine learning, a neural network is a structural model of a network consisting of many artificial neurons, and the strength of the connections between these artificial neurons is a learnable parameter.

The development of neural networks can be roughly divided into five stages: birth, ice age, revival, trough and rerising.

Birth: In 1943, McCulloch and Pitts first proposed an artificial neural network based on simple logic operations, which was called the MP model, and thus started the

research of artificial neural networks. In 1948, Alan Turing proposed a "B-type Turing machine" that could learn based on Hebbian's law. In 1951, Marvin Minsky, a student of McCulloch and Pitts, built the first neural network machine, SNARC. F. Rosenblatt proposed A neural network model that can simulate human perceptual ability, called Perceptron [Ros58], and proposed a learning algorithm that approximates the human learning process (iteration, trial and error).

In this period, neural networks were mainly used in automatic control and pattern recognition, and achieved remarkable results.

Ice Age: In 1969, Marvin Minsky published his book "Perceptron", which pointed out two key flaws of neural networks: first, perceptrons could not solve XNOR problem; second, computers at that time could not provide the computational power needed to process large neural networks. These assertions raised doubts about perceptron neural networks and led to an "ice age" of research on neural networks.

However, during this period, scholars proposed many useful models and algorithms, a significant portion of which became the cornerstone of later deep learning. In 1974, Paul Werbos proposed the back propagation (BP) algorithm [Wer74], which did not attract much response at that time. In 1980, inspired by the difference in the perceptual range of simple and complex cells in the primary visual cortex of animals, Fukushima proposed a multi-layer neural network with convolution and sub-sampling operations [Fuk80].

revival: The second culmination of research on neural networks was the backpropagation algorithm. In the mid-1980s, a parallel distributed processing (PDP) model became popular. The backpropagation algorithm also gradually became the main learning algorithm for the PDP model. Subsequently, Lecun introduced the back propagation algorithm to CNNs and achieved considerable success in handwritten digit recognition [LBBH98]. The backpropagation algorithm has consequently become the most successful neural network learning algorithm, until today.

Trough: Although neural networks can easily increase the number of layers and neurons to build complex networks, their computational complexity grows with them. The computer performance and data size at that time were not sufficient to support the training of large-scale neural networks. In the mid-1990s, statistical learning theory and machine learning models represented by support vector machines began to emerge. In contrast, the disadvantages of neural networks, such as unclear theoretical foundations, difficulty in optimization, and poor interpretability, became more prominent, and the research on neural networks was once again in the doldrums. Support vector machines

and other simpler methods (e.g., linear classifiers) gradually surpassed neural networks in popularity in the field of machine learning.

Rerising: With the great success of deep neural networks for tasks such as speech recognition [HS06] and image classification [KSH12], neural network-based deep learning is rapidly emerging. In recent years, the computational power of computers has increased dramatically with the spread of massively parallel computing and GPU devices. In addition, the size of data available for machine learning has grown. With the powerful computing power and massive data size, computers have been able to train a large-scale neural network end-to-end.

2.2 Fundamentals of Neural Network

Although the details of how the brain works are still to be explored, it is now generally accepted by researchers that the brain's main unit of computation is the neuron. As the diagram shows, neurons themselves are joined together by a number of elements that enter them (dendrites) and elements that leave them (axons). The neuron receives signals that enter it through the dendrites, performs some sort of computation on these signals, and then produces a signal on the axon. These input and output signals are known as activations. A neuron has an axonal branch that is connected to the dendrites of many other neurons. The connection between a branch of an axon and a dendrite is called a synapse. A key feature of a synapse is that it can scale the signal passing through it (x_i), as shown in Figure 2. This scaling factor can be referred to as a weight (w_i), and the brain is thought to learn by changing the weight associated with a synapse. Thus, different weights lead to different responses to the input. During the learning process, the weights are constantly adjusted, while the way neurons are connected to each other does not change. This feature makes the brain an excellent source of inspiration for machine-learning style algorithms, from which the idea for the design of ANN was derived.

2.2.1 Principles

In a feed forward neural network, each neuron belongs to a different layer. The neurons in each layer can receive signals from the neurons in the previous layer and generate signals for output to the next layer. Layer 0 is called the input layer, the last layer is called the output layer, and the other intermediate layers are called the hidden layers. There is no feedback in the whole network, the signal propagates from the input layer to the output layer in one direction.

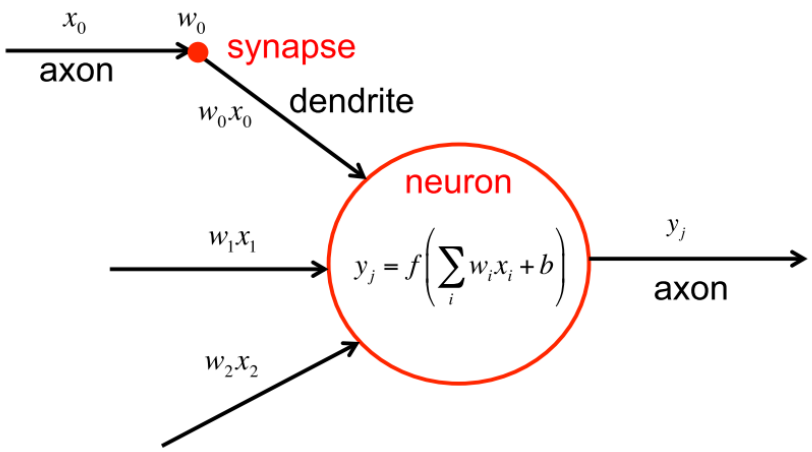


Figure 2.1: Connections of a neuron model. x_i, w_i, $f()$, and b are the activations, weights, non-linear function and bias, respectively. [FLJ]

Fig.2.2a shows a single layer neural network. Neurons in the input layer receive values from the data source and propagate them to neurons in the middle layer of the network, which is also often referred to as 'hidden layer'. There can be one or more hidden layers. After all the hidden layers, the weighted sum is eventually propagated to the output layer, which presents the final output of the network to the user. In the terminology of the neural network field, the output of a neuron is often referred to as the activation, while the synapses are often referred to as the weights.

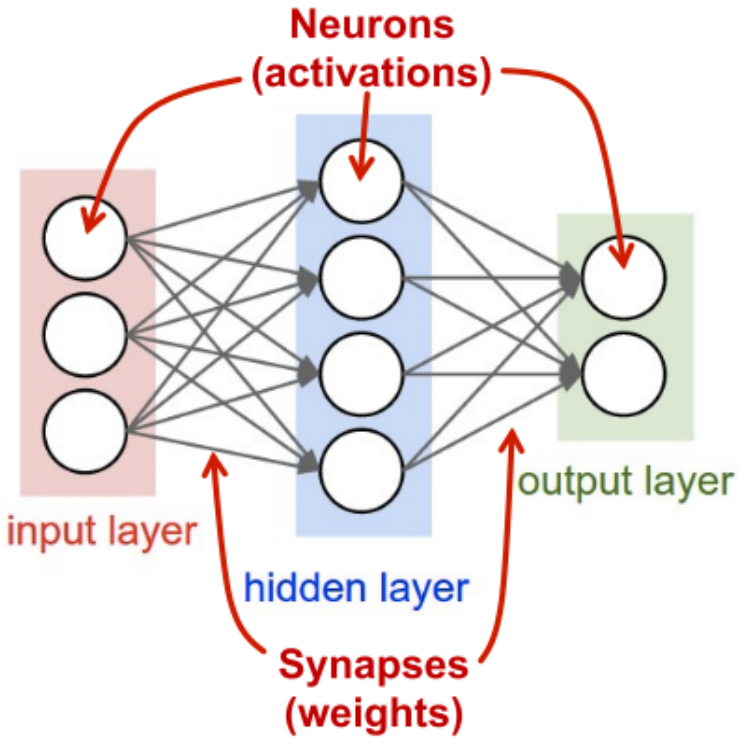
Fig.2.2b shows an example of the computation. It can be expressed as the following equation:

$$\mathbf{z} = \mathbf{w_i} \times \mathbf{a_i} + \mathbf{b} \tag{2.1}$$

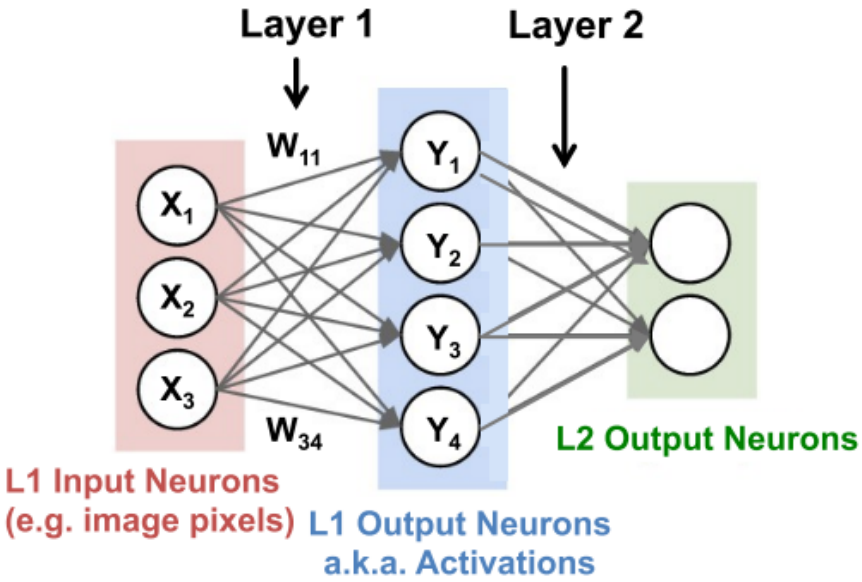
$$\mathbf{a} = \sigma(\mathbf{z}) \tag{2.2}$$

where Eq.2.1 and Eq.2.2 represents the iterative calculation process of neural networks. Variables **z**, **a**, **w** and **b** are called **activation, net activation, weight (matrix) and bias**. In this way, the feed forward neural network can pass the information layer by layer to get the final output.

By increasing the number of hidden layers, neural networks are able to learn higher level features with greater complexity and abstraction than shallow neural networks. For example, in the processing of computer vision tasks, the output of the first few hidden layers is mostly fuzzy lines and edges, corresponding to low-level features in the image; whereas the output of hidden layers closer to the output layer is a more



(a) Neurons and synapses



(b) Compute weighted sum for each layer

Figure 2.2: A simple neural network example [FLJ]

specific set of shapes and forms.

2.2.2 Activation Functions

An important improvement over the traditional perceptron model is the inclusion of a non-linear activation function, which allows the neural network to solve linearly indistinguishable problems that cannot be solved by the perceptron model.

Common activation functions include Sigmoid, hyperbolic tangent function (Tanh), rectified linear unit (ReLU), and leaky ReLU (LReLU). The respective formulas and curves are presented in Fig.2.3.

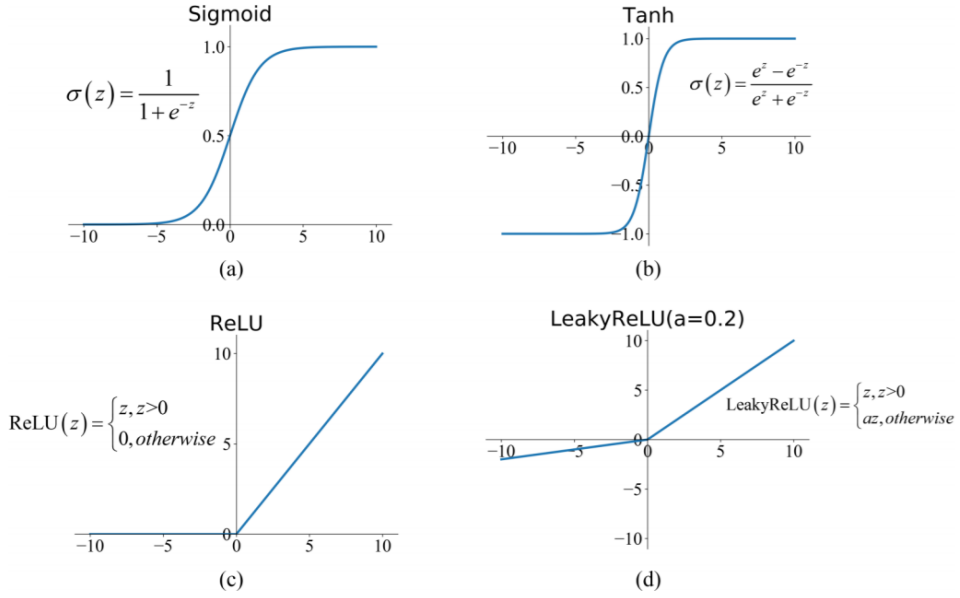


Figure 2.3: Commonly used activation functions: (a) Sigmoid, (b) Tanh, (c) ReLU, and (d) LeakyReLU [FLJ]

The purpose of the activation function is to introduce non-linearity which enables neural networks to tackle very complex non-linear problems. The activation function is generally at the end of each layer in the network and operates on the output activation values that complete the weighted summation to give it a non-linear factor. In most cases, the parameter weight w and bias b are randomly initialized and then iteratively updated according to back-propagated loss using the gradient descent method. The introduction of back propagation and gradient descent method can be obtained in Section 2.2.4.

2.2.3 Loss Functions

A loss function is a non-negative real number function that quantifies the difference between the model prediction and the true label. Several common loss functions are described below. The predicted output is represented as p_i , while the desired output is represented as y_i.

(a)Mean Square Error function: This function is widely used in regression problems. The mathematical expression of the estimated Euclidean loss is Eq.2.3.

$$H(p, y) = \frac{1}{2N} \sum_{i=1}^{N} (p_i - y_i)^2 \tag{2.3}$$

(b)Cross-Entropy Loss Function: The cross-entropy loss function can be used to measure the difference between two probability distributions. The mathematical representation of cross-entropy loss function is Eq.2.4.

$$H(p, y) = -\sum_{i} y_i \log(p_i) \quad \text{where} \quad i \in [1, N] \tag{2.4}$$

(c)Hinge Loss Function: This function is usually used in binary classification problems. Its mathematical formula is Eq.2.5.

$$H(p, y) = \sum_{i=1}^{N} \max(0, m - (2y_i - 1)\, p_i) \tag{2.5}$$

The margin m is commonly set to 1.

2.2.4 Back Propagation based on Gradient Descent

With the addition of a non-linear activation function, the relationship between the weights and the output can no longer be adjusted in a straightforward manner as in the traditional perceptron model. Therefore, the gradient descent method is employed for parameter updating during the learning process of neural networks.

When training a network, each weight value is updated by a gradient descent process. As shown in Eq.2.6, L represents the loss of total results, w^t and w^{t+1} represent the weight value before/after updating and α represents learning rate (an artificially specified factor). The gradient of loss relative to each weight, i.e. the bias of the loss

with respect to the weight, is used to update the weights. This gradient indicates how the weights should change to reduce the loss. When the gradient is positive, indicating that an increase in the weights makes the final loss increase, this weight should be reduced; if the gradient is negative, indicating that a decrease in the weights makes the loss increase, this weight should be increased. This process is repeated to reduce the overall loss.

$$w^{t+1} = w^t - \alpha \frac{\partial L}{\partial w} \tag{2.6}$$

Assuming that each sample is randomly drawn from the real data distribution independently and identically distributed, the optimization objective is to minimize the expected value of the loss function. The gradient descent method is to take a certain number of n samples at a time from a real data distribution of size N, and to approximate the gradient of expected loss by the gradient of empirical loss computed from these samples. When $n = N$, it is called batch gradient descent; when $n = 1$, it is called stochastic gradient descent(SGD); when $1 \leq n \leq N$, it is called mini-batch gradient descent. The latter two methods have much lower computational complexity than the batch gradient descent method because the number of samples collected in a single pass is much smaller. The stochastic gradient descent method is shown in Algorithm1.

Algorithm 1 Stochastic gradient descent

Input: Training set $T = \left\{\left(\boldsymbol{x}^{(n)}, y^{(n)}\right)\right\}_{n=1}^{N}$, Validation set V, learning rate η, loss function $\mathcal{L}$

1: Initialize parameter θ(weight W, bias b)
2: **repeat**
3: Shuffle samples in T randomly
4: **for** $n = 1 \cdots N$ **do**
5: Choose the sample $x^{(n)}, y^{(n)}$
6: $\theta \leftarrow \theta - \eta \frac{\partial \mathcal{L}(\theta; x^{(n)}, y^{(n)})}{\partial \theta}$
7: **end for**
8: **until** Error rate on V stop decreasing.

An effective way to calculate the partial derivative of the gradient is through a process called back propagation. Back propagation is a calculation derived from the chain rule of calculus, where values are passed backwards through the network to calculate how each weight affects the loss.This method has been successfully applied in the training process of neural networks [LBOM12]. The backpropagation process requires intermediate outputs from the network in order to perform the inverse calculation,

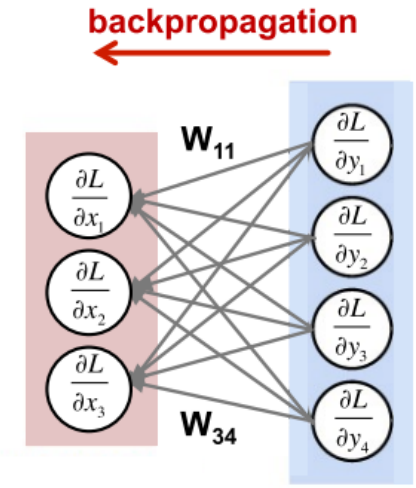
and therefore the training of neural networks has a large data storage requirement compared to other machine learning methods. A simple back propagation computation is shown in Fig.2.4.

The error term in layer l can be calculated from the error term in layer $l+1$, which is called back propagation of loss. The back propagation algorithm means that the loss term of a neuron in layer l is the sum of the weights of the error terms of all the neurons in layer $l+1$ that are connected to that neuron. Then, it multiplies the gradient of the activation function of that neuron. By applying the chain rule, the error term of any neuron in the network can be derived so that the gradient of its weight relative to the loss can be calculated for gradient updating. Alg. 2 gives the stochastic gradient descent training procedure using the back propagation algorithm.

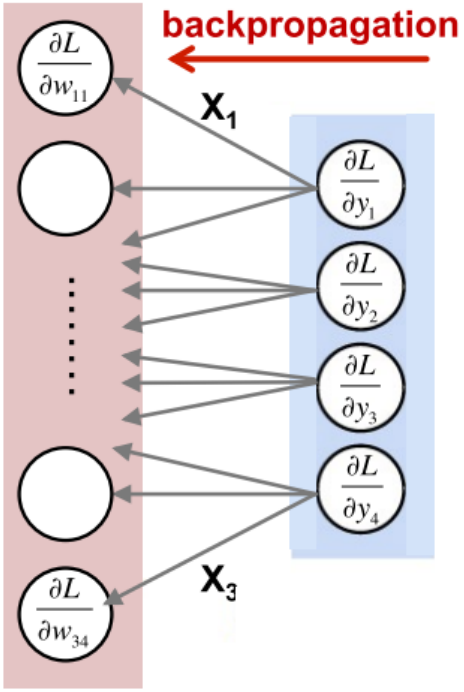
Algorithm 2 Back propagation based on stochastic gradient descent

Input: Training set $T = \left\{\left(x^{(n)}, y^{(n)}\right)\right\}_{n=1}^{N}$, Validation set V, learning rate η, loss function $\mathcal{L}$, Number of network layers L, Regularization factor λ

Initialize parameter θ(weight $\boldsymbol{W}$, bias $\boldsymbol{b}$)
repeat
  Shuffle samples in T randomly
  for $n = 1 \cdots N$ **do**
    Choose the sample $x^{(n)}, y^{(n)}$
    (Feed forward)
    for $l = 1 \cdots L$ **do**
      calculate input $z^{(l)}$ and activation $a^{(l)}$
    end for
    (Back propagation)
    for $l = L \cdots 2$ **do**
      $\frac{\partial \mathcal{L}\left(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)}\right)}{\partial \boldsymbol{W}^{(l)}} = \delta^{(l)} \left(\boldsymbol{a}^{(l-1)}\right)^{\top}$
      $\frac{\partial \mathcal{L}\left(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)}\right)}{\partial \boldsymbol{b}^{(l)}} = \delta^{(l)}$
    end for
    (Update parameters)
    $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \eta \left(\delta^{(l)} \left(\boldsymbol{a}^{(l-1)}\right)^{\top} + \lambda \boldsymbol{W}^{(l)}\right)$
    $\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \eta \delta^{(l)}$
  end for
until Error rate on V stop decreasing.



(a) Compute the gradient of the loss relative to the inputs



(b) Compute the gradient of the loss relative to the weights

Figure 2.4: An example of back propagation through a neural network [FLJ]

2.3 Convolutional Neural Network

When using fully connected feed-forward networks to process images, there is one critical problems: too many parameters. If the input image size is $100 \times 100 \times 3$ (i.e., the image height is 100, the width is 100 and the 3 RGB colour channels), in a fully connected feed-forward network, each neuron in the first hidden layer to the input layer has $100 \times 100 \times 3 = 30,000$ mutually independent connections, each corresponding to a weight parameter. As the number of neurons in the hidden layer increases, the size of the parameters also increases dramatically. This results in a very inefficient training of the entire neural network and is prone to overfitting.

Convolutional Neural Network (CNN or ConvNet) is a deep feed-forward neural network, which is inspired by the Receptive Field mechanism in biology. The receptive field mechanism refers to the property of some neurons in the auditory and visual nervous systems that neurons receive signals only from the stimulus area they innervate.

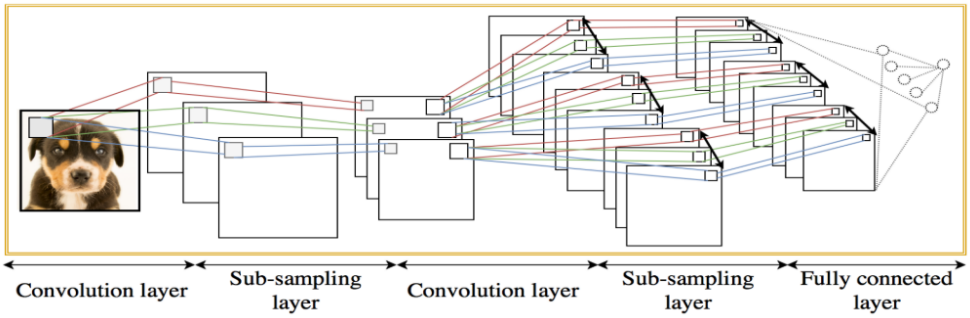


Figure 2.5: An simple CNN framework. [PSY+18]

Current CNNs are generally feed-forward neural networks consisting of a cross-stack of convolutional, convergence and fully connected layers, trained using a back propagation algorithm. Fig.2.5 shows a simple CNN framework. CNNs have structural properties: equivalent representations, sparse interactions, and parameter sharing [GBC16]. These properties allow convolutional networks to have a certain degree of translation, scaling and rotation invariance. Compared to feed-forward neural networks, convolutional neural networks have fewer parameters.

2.3.1 Convolution Operation

Convolution is an important operation in analytical mathematics. In signal processing or image processing, one-dimensional or two-dimensional convolution is often employed. In convolutional neural networks, the main implementation of convolutional computation is to slide a convolutional kernel (also called a filter) over a feature map and

obtain a new set of features by convolutional operations without flipping the kernel. Given an input $\boldsymbol{X} \in \mathbb{R}^{M \times N}$ and a convolution kernel $\boldsymbol{W} \in \mathbb{R}^{U \times V}$, the convolution operation is defined as Eq.2.7:

$$y_{ij} = \sum_{u=1}^{U} \sum_{v=1}^{V} w_{uv} x_{i+u-1, j+v-1} \tag{2.7}$$

Fig.2.6 illustrates the steps of the convolution calculation. The light blue colour represents the area of the input feature where the convolution kernel is sized, and the light green box represents the 2×2 convolution kernel. The resulting product values from multiplying elements in the same position in both are all added up to the output feature value.

In this example, the size of the convolution kernel is 2×2 and the step size is 1, so the final size of the output feature map is smaller than the size of the input feature map by 1. To obtain the feature maps of any size we need, the padding method is available. By padding zeros around the feature map, feature extraction can be made more flexible before performing the convolution operation.

2.3.2 Convolution Layer

The core difference between the convolutional layer and the fully connected layer is that the matrix multiplication operation in the fully connected layer is replaced by the convolutional operation. Similar to the definition in Eq.2.1, the input of l_{th} layer of the convolutional neural network is the result of the convolution of the activation value of $(l-1)_{th}$ layer with the convolution kernel of l_{th} layer, i.e., as shown in Eq.2.8.

$$\boldsymbol{z}^{(l)} = \boldsymbol{w}^{(l)} \otimes \boldsymbol{a}^{(l-1)} + b^{(l)} \tag{2.8}$$

As a result of these properties, the convolutional layer gains two significant advantages over the fully connected layer: 1) sparse connectivity. In a fully connected layer, each neuron in one layer is connected to all neurons in the next layer. In contrast, in a convolutional layer, each neuron is connected to only those neurons in the next layer within a window that does not exceed the size of the convolutional kernel. As a result, the number of weights or connections required between convolutional layers is greatly reduced compared to fully connected layers. 2) Shared weights. Between convolutional layers, weights are not distributed between neurons, but all neurons share the same set of weights in a convolutional kernel, a fact that greatly reduces the required training

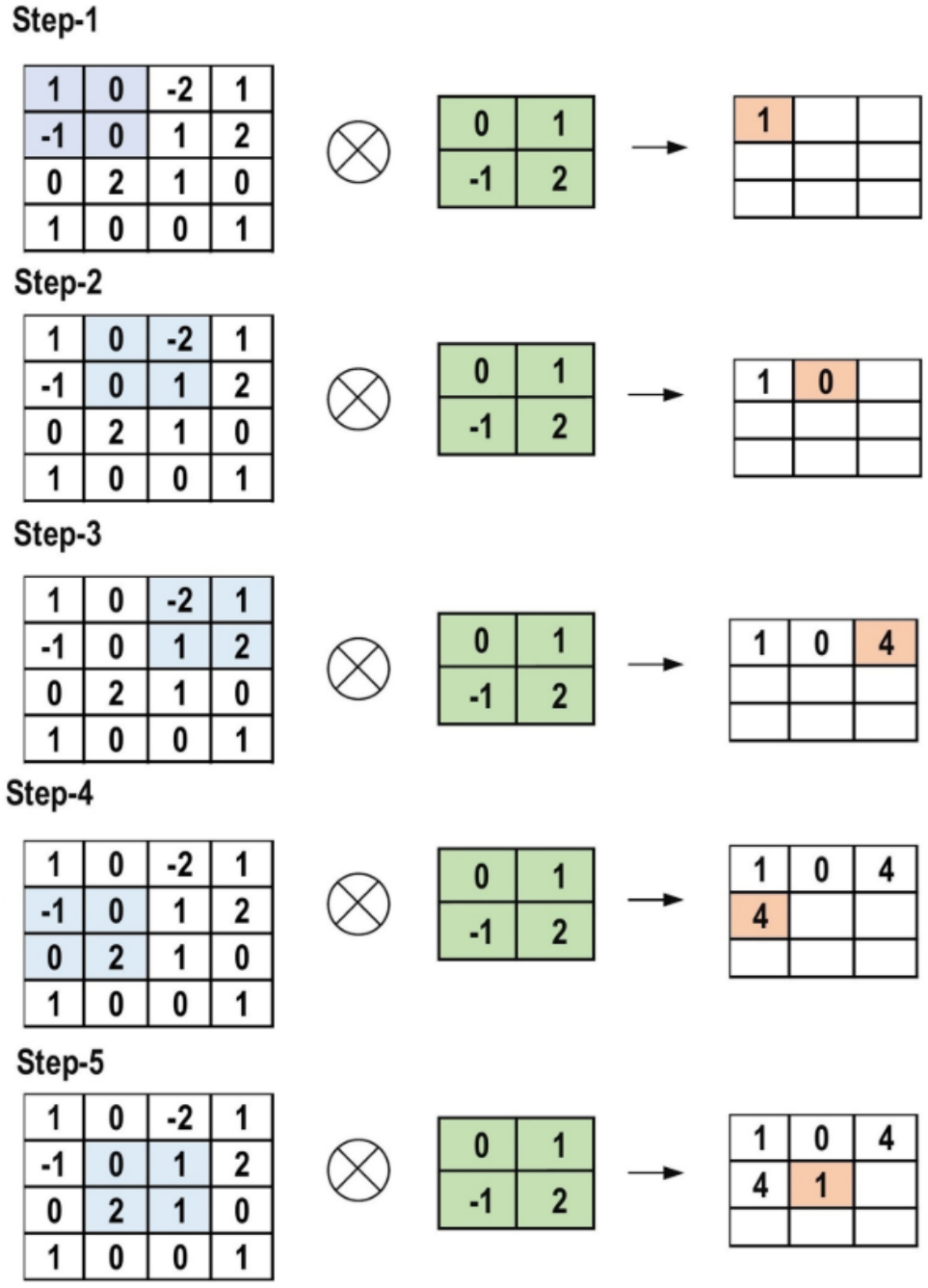


Figure 2.6: Calculations executed at each step of convolutional layer [LAH+21]

time and computational cost.

2.3.3 Pooling Layer

The main task of the pooling layer is to sub-sample the feature map to reduce the number of features and thus the number of parameters, while retaining as much dominant information (or features) as possible. Similar to the convolution operation, the span of the pooling and the size of the filter are assigned before the pooling operation is performed. Of the multiple pooling methods available, the most familiar and frequently utilised pooling methods are maximum pooling and average pooling. Fig.2.7 illustrates these three pooling operations.

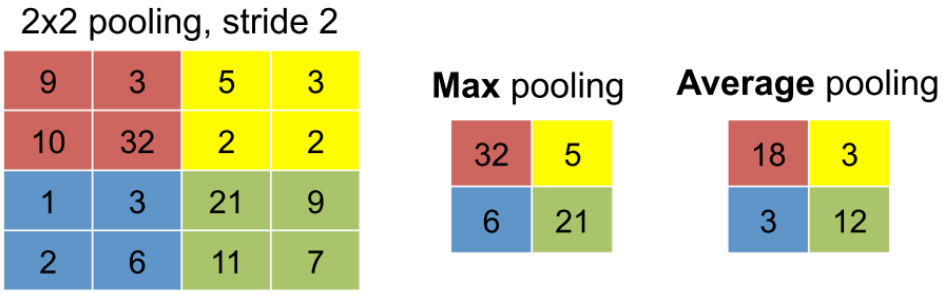


Figure 2.7: Various pooling forms [LAH+21]

2.3.4 Typical Convolutional Network Structure

A typical convolutional network is a cross-stack of convolutional layers, pooling layers, and fully connected layers. A convolutional block contains m consecutive convolutional layers and b pooling layers (m is usually set to $2 \sim 5$ and b to 0 or 1). A convolutional network can have n consecutive convolutional blocks stacked on top of each other, followed by k fully-connected layers (the range of values for n is large; k is typically $0 \sim 2$).

AlexNet [KSH12] is the first modern deep convolutional network model that uses many modern deep convolutional network techniques for the first time, such as using GPUs for parallel training, using ReLU [NH10] as a non-linear activation function, using Dropout [SHK+14] to prevent overfitting, and using data augmentation to improve the accuracy of the model. The structure of AlexNet is shown in Fig.2.8 and consists of five convolutional layers, three convergence layers and three fully connected layers (the last layer is the output layer using the Softmax function). Because the size of the

network exceeded the memory limit of a single GPU at the time, AlexNet splits the network into two halves for parallel training.

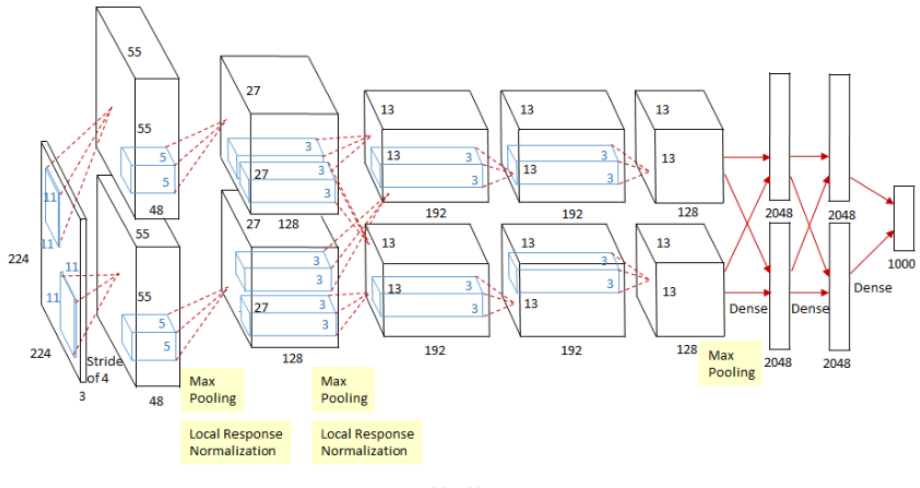


Figure 2.8: The structure of Alexnet [Tsa18]

AlexNet has inspired a great deal of later work as the iconic modern neural network structure. Although it is now no longer good enough for State-of-the-Art, it still plays an important role as a classical model in teaching and low-threshold applications.

2.4 Binary Neural Network

Although deep neural networks have satisfactory performance for many tasks, they rely on high-performance hardware such as GPUs to meet the high demand for storage and computing conditions. In real-world applications, the conditions of the devices are often not sufficient to meet the demand.

In order to reduce the storage and computing threshold of the network, a popular approach is to use a low precision representation of the network weights, which serves to compress the network. The conventional precision for neural networks is FP32, which is a 32-bit floating point number. A common low precision is INT-8, which is an 8-bit fixed-point integer. The most extreme form of low precision processing is binarization. The corresponding neural network is known as a binary neural network(BNN) [CHS+16].

2.4.1 Binarized Weight Values

Since weights in full precision floating point numbers consume too many computational resources, binary neural networks aim to represent weights and activations in 1 bit, i.e. weights and activations in a neural network can only have two possible values, -1 (0) or +1, as the logic of $Sign()$ function defined as Eq.2.9.

$$x^b = \text{Sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.9}$$

2.4.2 Back Propagation in Binary Neural Network

While training a binary neural network, if a gradient descent based back propagation algorithm is to be used to update the parameters, there is no way to directly derive the derivative values and therefore perform gradient descent directly, as the binarization function described above is not differentiable. A solution to this problem is to introduce the technique of straight-through estimators (STE), as shown in Eq.2.10.

$$\text{clip}(x, -1, 1) = \max(-1, \min(1, x)) \tag{2.10}$$

Using STE, the weights can then be gradient updated using an established and proven optimiser such as SDG or Adam. The updated weights are stored in the true weights W_r, while the trained network uses the binarized weights W_b. The other parts are consistent with standard back propagation.

For the optimization of binary neural networks, it is common practice to reduce the quantization error of the weights and activation values. It is a straightforward solution similar to the standard quantization mechanism, i.e., the quantization parameters should be as close as possible to the full precision parameters, with the expectation that the performance of the binary neural network model will be close to the full precision parameters. Out of concern to avoid the problem of W_r differing too much from W_b and thus causing the weights to be difficult to change, BNNs limit the value of W_r to between -1 and +1.

2.4.3 XNOR Operation

When using binarized weights and activation values, the original dot product operation can be reduced to a bitwise operation. The binary values can be either -1 or +1.

If 0/1 is used to encode -1/+1, as shown in Tab.2.1, the multiplication of binary values is identical to XNOR logic. Using XNOR logic-based bit operations instead of dot-multiplication can save significant computational resources in the hardware implementation.

Encoding(Value)		XNOR(Multiply)
0(-1)	0(-1)	1(+1)
0(-1)	1(+1)	0(-1)
1(+1)	0(-1)	0(-1)
1(+1)	1(+1)	1(+1)

Table 2.1: XNOR and multiplication

XNOR is simpler at the bit level than the dot product operation of multiplying and then summing. For a binary encoding generated by XNOR operation, it can be summed by multiplying the number of 1 bits in a set of XNOR products by 2 and subtracting its total number of bits to sum it. The operation of these bits is much simpler than multiplying and summing multi-bit floating point numbers, which can result in shorter operation times and less hardware resource requirements. It is also a clear benefit for the application of neural networks on FPGA-based devices.

2.5 Optimizations of Neural Network

The neural network model is a non-convex function, coupled with the problem of gradient disappearance in deep networks, which makes it difficult to optimize; in addition, deep neural network models generally have more parameters and larger training data, which can lead to lower training efficiency. At present, researchers have developed a number of empirical techniques from extensive practice to improve learning efficiency and obtain a good network model in terms of optimization.

When training deep neural networks, the size of the training data is usually quite large. If in gradient descent each iteration, the gradient over the entire training data has to be computed, which requires more computational resources. In addition the data in a large training set is usually very redundant, and it is not necessary to compute the gradient over the entire training set. Therefore, when training deep neural networks, mini-batch gradient descent is often used.

The main factors affecting the small batch gradient descent method are 1) batch size

2) learning rate and 3) gradient estimation. In order to train deep neural networks more efficiently, based on the standard small-batch gradient descent method, some improvements are also often used to speed up the optimization, such as how to choose the batch size, how to adjust the learning rate, and how to correct the gradient estimation.

2.5.1 Learning Rate Scheduler

The learning rate is an important hyper-parameter when optimizing neural networks. In gradient descent, the value of the learning rate is very critical; if it is too large, it will not converge, and if it is too small, it will converge too slowly. Commonly used learning rate scheduler methods include learning rate decay, learning rate warm-up, periodic learning rate adjustment and some adaptive learning rate adjustment methods such as AdaGrad, RMSprop [GHS], Stochastic Gradient Descent with Warm Restarts(SGDR) [LH17], etc.

In SGDR, the learning rate is increased or decreased periodically according to the cosine function and each period is longer than the previous one, depending on the proportion of the Warm-Up parameter. Its mathematical definition is given in Eq.2.11.

$$\eta_t = \eta_{\min}^i + \frac{1}{2}\left(\eta_{\max}^i - \eta_{\min}^i\right)\left(1 + \cos\left(\frac{T_{cur}}{T_i}\pi\right)\right) \tag{2.11}$$

where $\eta_{\min}^i$ and $\eta_{\max}^i$ are the learning rate ranges. T_{cur} accounts for how many epochs have been performed since the last restart. Therefore, the learning rate is increased or decreased periodically according to the cosine function and each period is longer than the previous one, depending on the proportion of the Warm-Up parameter.

2.5.2 Gradient Estimation

In addition to adjusting the learning rate, a correction to the gradient estimate can be made. In the stochastic (or mini-batch) gradient descent method, if the number of samples selected at a time is relatively small, the loss will decrease in an oscillatory manner. By applying a correction to the gradient to reduce randomness, the speed of optimisation can be improved.

The most general method currently available is Adaptive Moment Estimation (Adam) [KB17], which can be seen as a combination of the momentum method and RMSProp [GHS], allowing the use of both momentum as the direction of parameter updates and

adaptive adjustment of the learning rate.

2.5.3 Parameter Initialization

Parameter learning of neural networks is a non-convex optimization problem. When gradient descent is used to optimize the network parameters, the selection of the initial values of the parameters is very critical, which is related to the optimization efficiency and generalization ability of the network. There are usually three methods of parameter initialization: pretrained initialization, random initialization and fixed value initialization.

Pretraining initialization: Different initial values of parameters converge to different local optimal solutions. Although the losses of these local optima are relatively close to each other in the training set, their generalization ability varies greatly. A good initialization will cause the network to converge to a local optimum with high generalization power. Usually, a model that has been trained on large-scale data can provide a good initial value for the parameters.

Random initialization: One of the simplest and most widely used methods of random initialization is to generate the initial values of the parameters by sampling from a distribution with a fixed mean (usually 0) and variance.

Fixed value initialization: For some special parameters, we can initialize them with a special fixed value based on experience. E.g., the bias is usually initialized with 0.

2.5.4 Batch Normalization

The Batch Normalization (BN) method [IS15] is an effective layer-by-layer normalization method that can normalize any intermediate layer in a neural network. At the first, the motivation for batch normalization was to solve the internal covariance bias problem, but later researchers found that the main advantage of normalisation was that it led to a smoother optimization profile [STIM18].

As shown in Eq.2.12, The distribution of layer input activations (σ, μ) is normalized so that it has zero mean and unit standard deviation. The parameters (γ, β) are learned from training. ϵ is a small constant to avoid numerical problems.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.12}$$

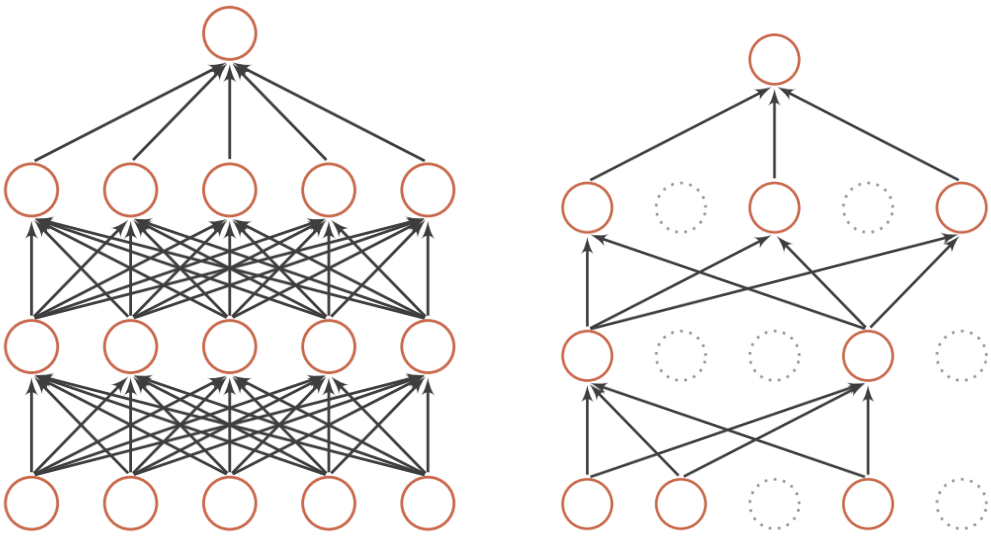
BN can normalize the input distribution of each neural layer to a standard normal

distribution, which allows each neural layer to have better scale invariance to its input. Regardless of the parameter changes in the lower layers, the inputs in the higher layers remain relatively stable. On the other hand, it can make the optimization landscape(the surface shapes of loss functions in high-dimensional space) of the neural network smoother and make the gradient more stable, thus increasing the convergence speed.

2.5.5 Regularization

Regularization is a class of methods that improves generalization by limiting the complexity of the model and thus avoiding overfitting. $L1$ and $L2$ regularization are the most commonly used regularization methods in machine learning. Overfitting of the model on the training dataset is reduced by constraining the $L1$ and $L2$ norms of the parameters.

Another widely used regularisation method is Dropout [SHK+14], where a random number of neurons are dropped (along with their corresponding connected edges) to avoid overfitting when training a deep neural network. Fig.2.9 gives an example of a network after the dropout method has been applied. By applying the Dropout method, the network is not dependent on the output of a particular neuron and the robustness is greatly enhanced



(a) Standard network **(b)** Dropout applied network

Figure 2.9: Dropout example

2.5.6 Sparse Connection

The fully connected layer is the most common component unit in a deep neural network. In a fully connected layer, the output activation consists of a weighted sum of all input activations, i.e. each output is connected to all inputs in the previous layer, a feature that leads to significant storage and arithmetic requirements. The idea of sparse connectivity is to remove some of the connections by setting their weights to zero without undue impact on the final accuracy. The Fig.2.10 shows an example of a fully connected layer compared to a sparsely connected one.

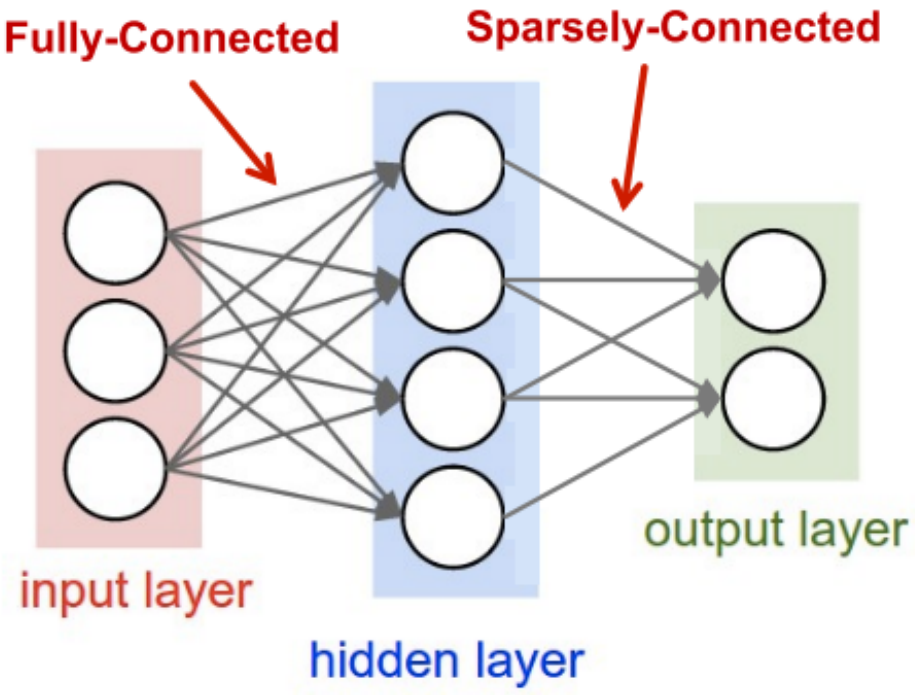


Figure 2.10: Sparse connection [FLJ]

If this idea of sparsity is applied to network structures, structured sparsity occurs by fixing the size of the input window. Keeping the weights of the input window constant over the same single learning process allows all outputs to share the same set of weights, thus effectively reducing the space required to store the weights. The convolutional neural networks are designed with this in mind.

2.5.7 Residual Learning

Residual learning, proposed in [HZRS15], improves the efficiency of information propagation by adding shortcut connections to the non-linear convolutional layers. Unlike standard convolutional neural networks, the basic unit of a residual network is

the residual block, with each residual block containing two convolutional layers, two activation function layers and a shortcut connection, as shown in the Fig.2.11.

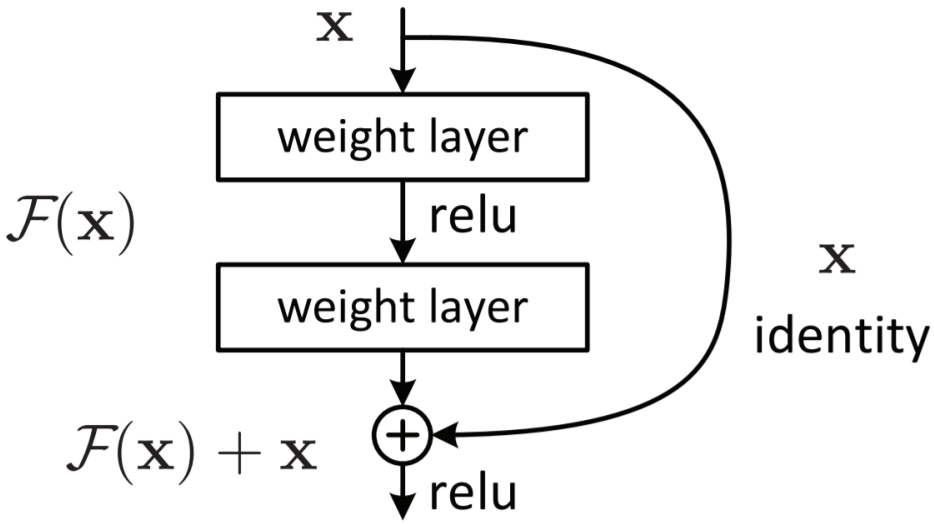


Figure 2.11: Residual block with shortcut connections [HZRS15]

In theory, a nonlinear unit consisting of a neural network has sufficient capacity to approximate the original objective function $\mathcal{F}(x)$ or the residual function $\mathcal{F}(x) - x$, but in practice the latter is easier to learn. The power of residual learning made ResNet the first deep neural network to exceed human-level accuracy in the ImageNet challenge. The idea of residual learning is also not limited to convolutional neural networks, but is commonly used in deep neural networks.

2.6 Programmable Logic Device

2.6.1 Field-Programmable Gate Array

Field-programmable gate array(FPGA) is a programmbale logic device. The biggest advantage of FPGAs over application specific integrated circuits(ASICs) is that they are re-programmable, whereas ASICs are not. This reconfiguration feature makes FPGAs widely suitable for prototyping and rapid verification, thus minimizing serious design errors. Second, FPGAs require only minutes to tens of minutes to reconfigure the internal logic, which is a significant advantage for rapid iteration requirements. Designers can update or modify the logic at any time, which is not possible in an ASIC. However, FPGAs also have some disadvantages, due to their flexible nature, they are larger and slower than their counterparts. But overall, FPGAs are still one of the top choices for lightweight general-purpose development needs.

A standard FPGA has the following components: (I) programming logic blocks, also call configurable logic blocks(CLB), which enable logic functions. (II) routing interconnects, which are used to establish connections between the programmed logic blocks; and (III) input and output blocks(IOB), which are ultimately used to implement the interaction between the internal structure of the FPGA and the external peripherals. Fig.2.12 illustrates a SRAM-based FPGA architecture, where SM and PSB represents switch matrix and programmable switch blocks.

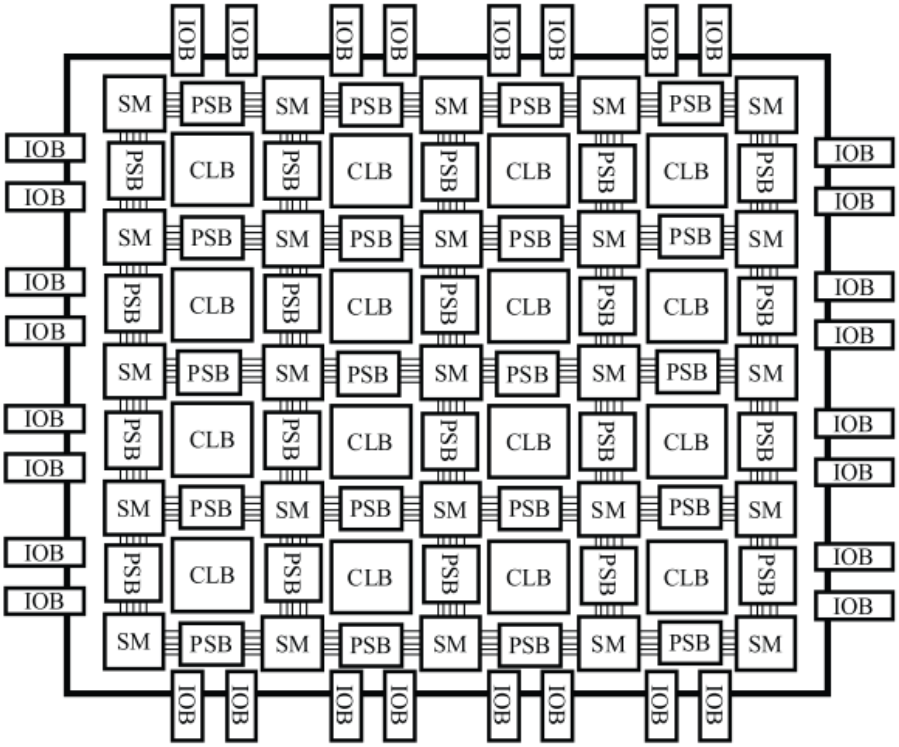


Figure 2.12: An architecture of a normal FPGA

2.6.2 Look-up Table

The look-up table (LUT) is one of the basic components of the CLB and is divided into single-input LUTs and multi-input LUTs. for an LUT with k input pins, the number of bit combinations it can be configured is 2^k. 4-LUT(shown in Fig.2.13) is one of the most commonly used LUTs in FPGAs and as mentioned above, it has 4 input pins and 16 input bit combinations to configure.

The LUT is essentially a Random Access Memory(RAM), and when the user describes a logic circuit through a schematic or hardware description language, the FPGA development software automatically calculates all possible results of the logic circuit

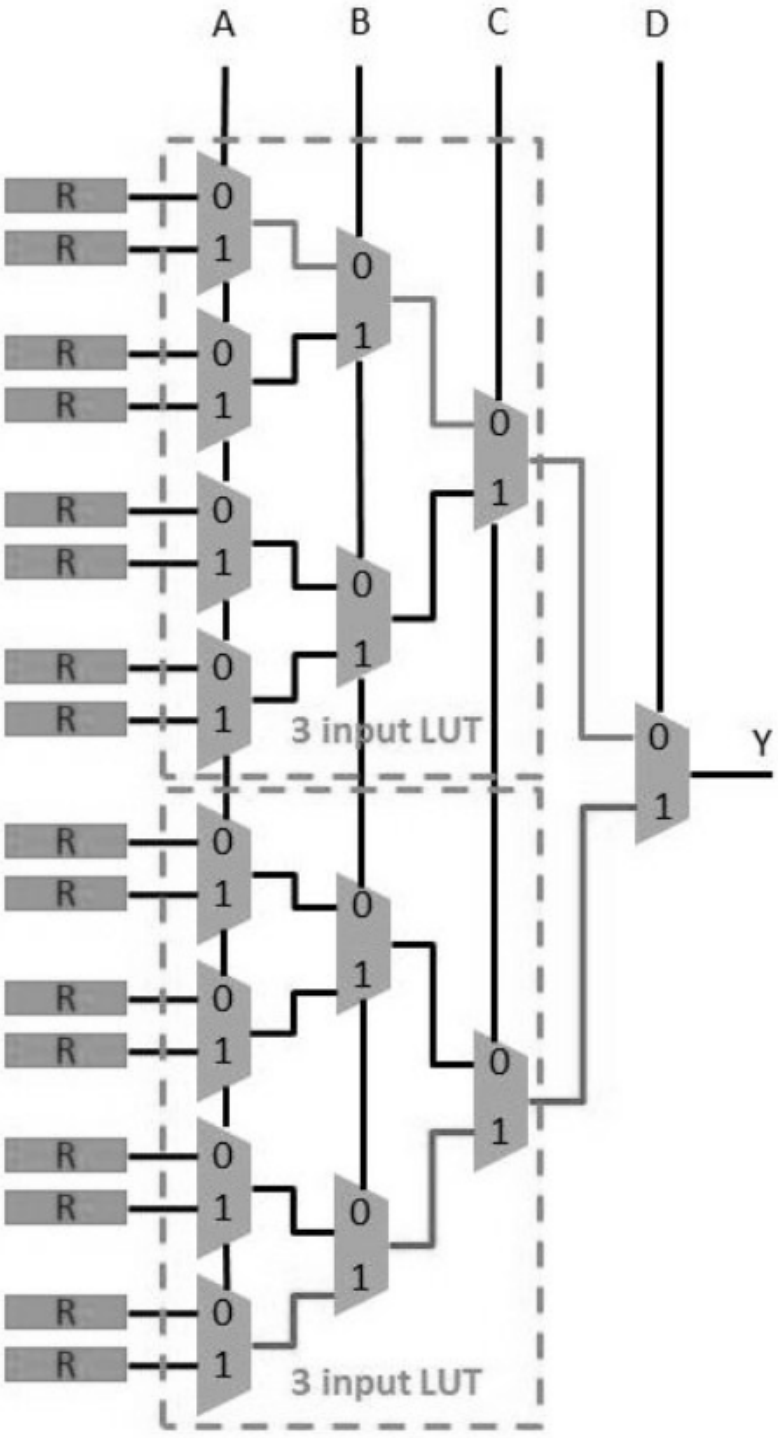


Figure 2.13: A 4-LUT example

and writes the results into the RAM in advance, so that each signal input for logic operation is equivalent to inputting an address for a lookup table, finding out the content corresponding to the address, and then outputting it. By using lookup tables instead of logic gates as the basic unit of CLB, FPGA circuits can provide higher flexibility and integration complexity.

2.6.3 Memory based Reconfigurable Logic Device

In addition to standard FPGAs, there are other kinds of programmable logic devices that use SRAM-based LUTs as the basic unit. One example is a recently proposed memory based reconfigurable logic device (MRLD) [WHT+17], taking multiple look-up tables (MLUT) as the core component structure, has demonstrated attracting benefits such as low production cost, low power and small delay and owns programmable function to implement designed circuit logic. Its structure is shown in Fig. 2.14.

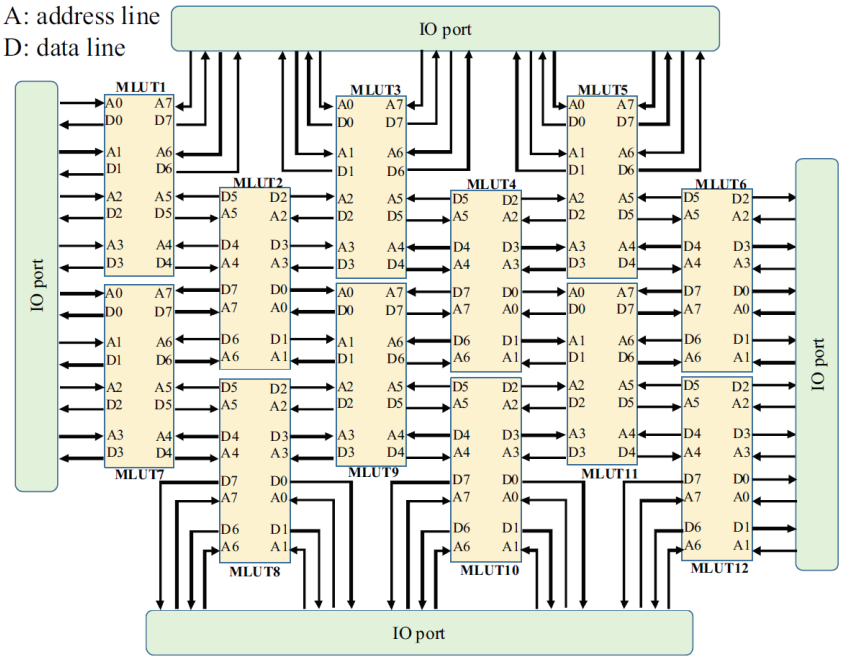


Figure 2.14: The structure of MRLD

MRLD is an MLUT array constructed with a special internal connection structure. As shown in Fig. 2.15, a MLUT block has 4 SRAM with 8 bits for each and one 8-bit output control register and can implement input-and-output relationships under the the number of bits. MLUTs are basic reconfigurable elements that consist of synchronous SRAM and asynchronous SRAM, with address inputs and data outputs forming pairs of interconnects. For an MLUT, the address input comes from the data

outputs of its neighboring MLUTs, and the outputs are connected to the address inputs of these MLUTs. Each MLUT can operate as a independent CLB or a wiring block(like SM or PSB).

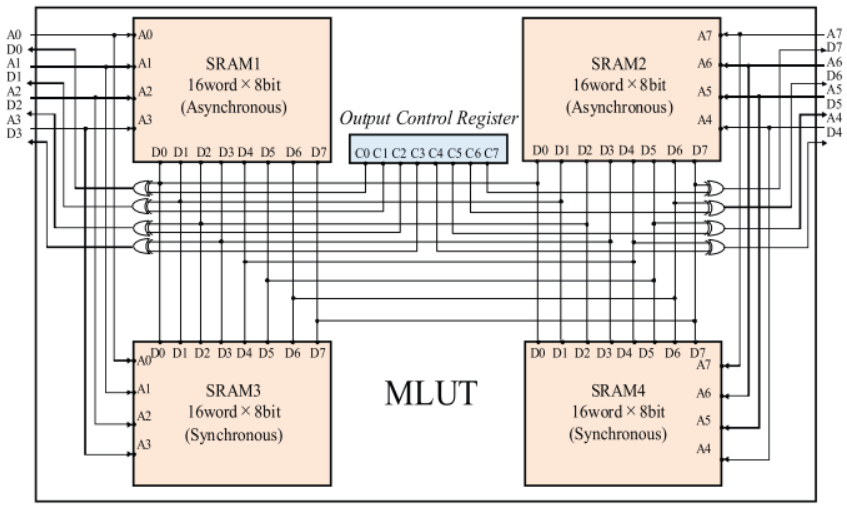


Figure 2.15: Internal schematic of MLUT

Compared to logic and switch block independent FPGAs, the MRLD has the following advantages: Each MLUT can be used as a logic block or wiring element by configuring the corresponding truth table in the SRAM. the address input/data output of the MLUT will be the input/output of the configured logic circuit (or wiring element). Since the logic blocks and wiring elements are configured in the SRAM, there is no longer a need for as many interconnect resources as in an FPGA, making it possible to have high density reconfigurable devices with small latency and low power.

Chapter 3

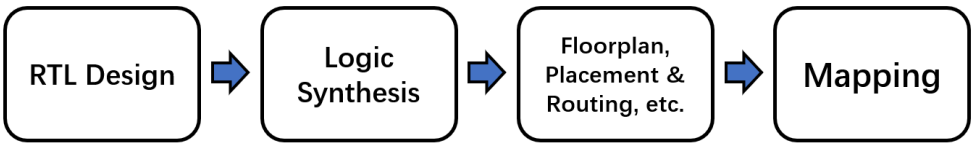
Multiple Look-up Table based Logic Learning by Neural Network

Abstract

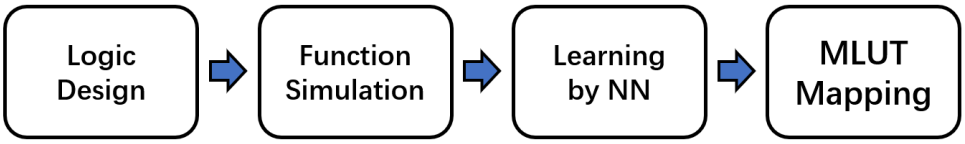
This section presents approximate computing consistent with a memory-based reconfigurable logic device (MRLD). We propose a novel implementation flow how to realize a function of multiple look up table (MLUT) by employing neural network (NN) based machine learning. Like a function fitting, our method implements a logic function induced by a set of input and output. To verify the performance of approximate computing implementation, we compare a general polynomial regression method and a deep neural network. The results suggest relatively a deeper NN is superior on loss value and accuracy rate. The NN models achieve lower symbol error rate (SER) and get considerable loss reduction respectively compared to the polynomial regression.

3.1 Problem Description

A typical hardware implementation contains a logic synthesis, which is the process of converting a high-level description of design into an optimized gate-level representation. Usually, the source of logic synthesis information is hardware description language in RTL level, and target is to generate netlist of circuit. After partitioning, floorplan, placement and routing, the circuit can get mapping with designed logic finally, as shown in Fig.3.1a. A tentative idea is to employ reconfigurable hardware device like FPGA or SRAM logic device which owns programmable function to approach designed circuit logic, as shown in Fig.3.1b.



(a) Typical Logic Synthesis procedure



(b) NN-based MLUT mapping procedure

Figure 3.1: Comparison of NN-based MLUT matching and typical logic synthesis

Assuming that a NN model can learn the input-and-output relationship, under not guaranteed accuracy request, we can implement an approximate function on logic device, as shown in Fig.3.2. Through the function simulation, an input-and-output relationship of circuit logic can be obtained.

Considering the high computation and strong ability of abstracting deep-layer characteristics, it is appropriate to be solved with neural network. Under appropriate parameter configuration, NN is not limited by degrees of polynomial and probably has effective performance on approximate function fitting problem.

3.2 Network Construction

The NN model consists of multiple layers, mainly including a fully connected (FC) layer, a nonlinear activation layer, and an output layer. The fully connected layer performs linear transformation of the input vectors. It is the main component of the NN model. The nonlinear activation layer uses the rectified linear unit RELU function.

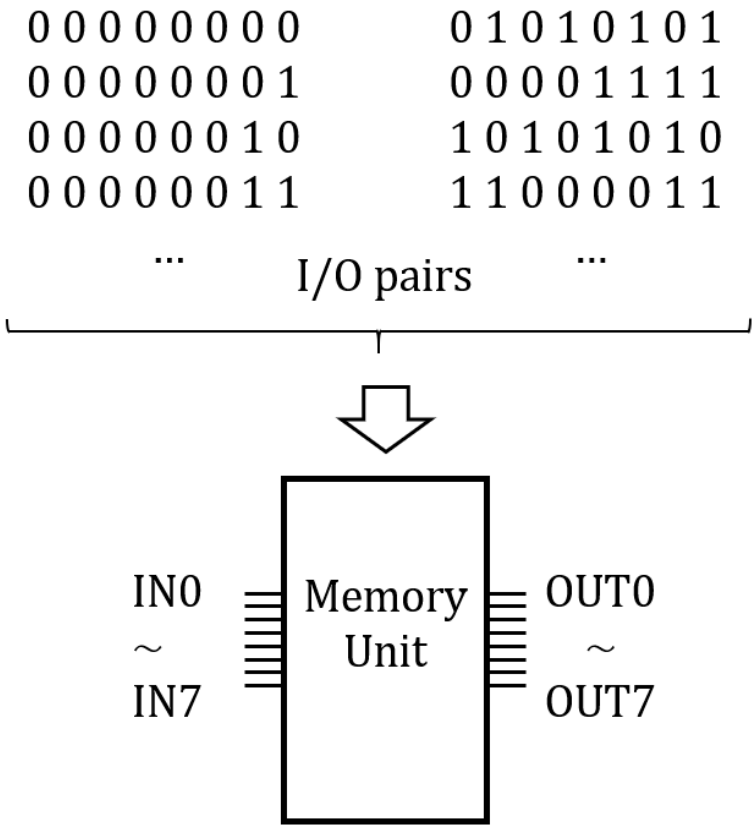


Figure 3.2: Approximate function configuration induced by simulation

The Softmax layer is crucial for implementing the output probability distribution and is often used for a variety of classification problems; whereas for numerical regression problems, the Softmax layer is incompatible with the expected data form. In this paper, the last layer of the NN model is set as the FC layer, which directly outputs the prediction results.

The constructed NN model is shown in Fig.3.3 The size of the input layers is not preset to allow for scaling to different needs. The number of hidden layers is set to 5 and the optimizer uses the Adam optimizer, which is able to achieve a good balance between convergence speed and global optimization. The loss function uses the mean square error (MSE) function.

3.3 Experiments

To check the performance of NN method, an appropriate target function for fitting should be set. In the following experiment, two typical functions are chosen as simulation targets. Function 1 is 3.1, and function 2 is 3.2.

$$f(x) = 2x^3 + 5x^2 + 8x \tag{3.1}$$

$$f(x) = (x^2 + x^{-1}) + \sin x \tag{3.2}$$

Function 1 is a typical polynomial function without a bias while function 2 is a polynomial function based on sin(x). For a polynomial regression (PR) model, it is difficult to fit. In order to show comparison, the fitting results and loss value tables of learning result for function 1 are shown in Fig.3.4 and Tab.3.1, respectively.

Model	Polynomial Regression	Neural Network
Training	$36.82k$	$12.41k$
Testing	$1.39 \times 10^6 k$	$433.38k$

Table 3.1: Loss value of function 1.

As well, the fitting results and loss value tables for function 2 are shown in Fig.3.5 and Tab.3.2, respectively. It can be seen obviously that, under the case of simple polynomial function 1, both models, NN and PR, can learn the function, though NN takes a little advance than PR. For the sin-based function 2, both methods meet

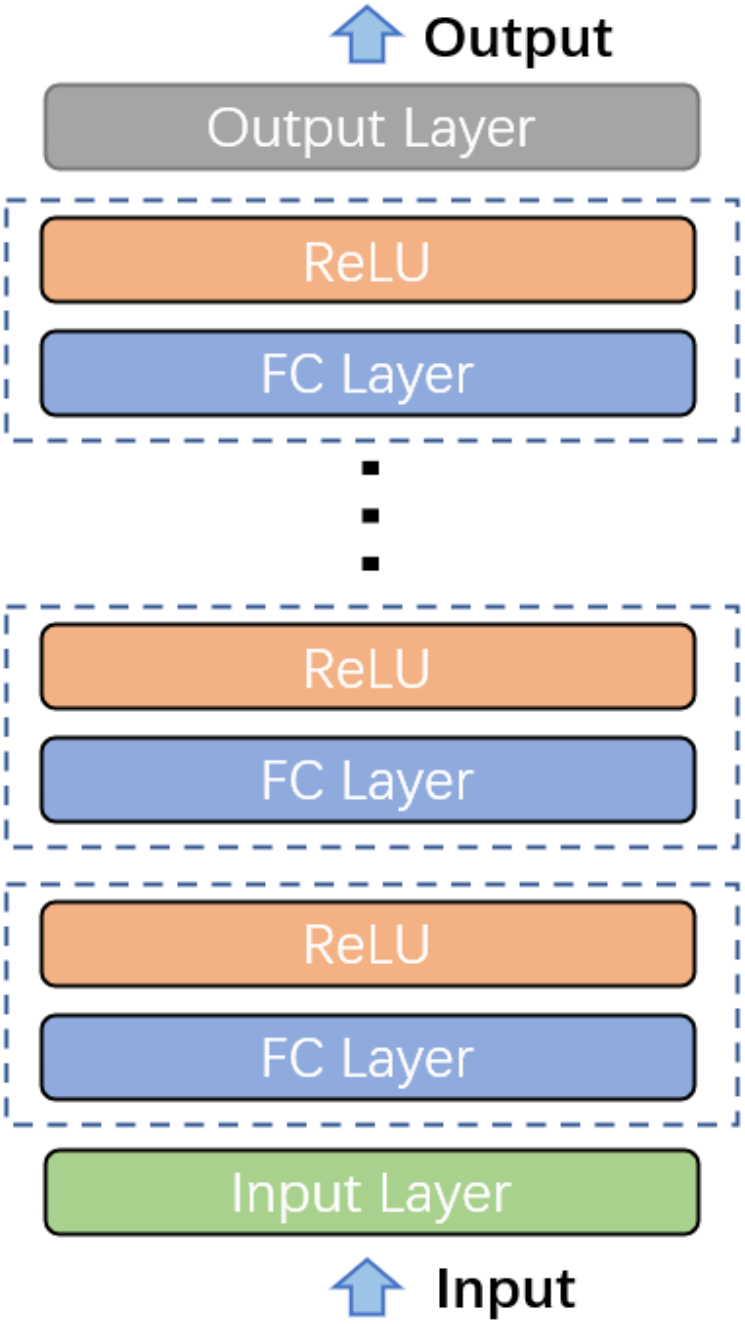


Figure 3.3: Constructed NN model

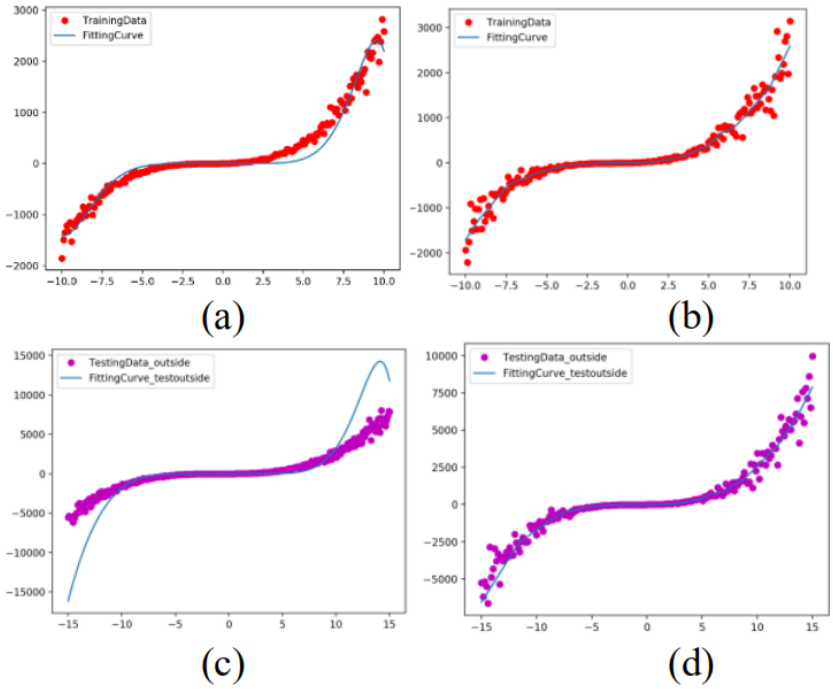


Figure 3.4: Results for function 1; (a) Training results by PR, (b) Testing results by NN, (c) Testing results by PR,(d) Testing results, by NN.

the limitation of the fitting ability on outside range, which is limited by the fitting principle. While in the range of training data, for the new testing data, NN performs much better.

Model	Polynomial Regression	Neural Network
Training	197.536	25.386
Testing	$1.719 \times 10^7 k$	$14k$

Table 3.2: Loss value of function 2.

Although the performance of NN outperforms that of PR model, it still does not satisfy the request well. To reduce the loss and improve the accuracy in further steps, some optimizing measures are essential to be operated.

To enhance the quality of an NN method, a natural idea is expanding the size of network and deepen its depth. A larger network leads to better performance is generally known in area of machine learning, yet with the demerits of slower convergence speed and a higher risk of over-fitting. By testing with various-size networks, it is helpful

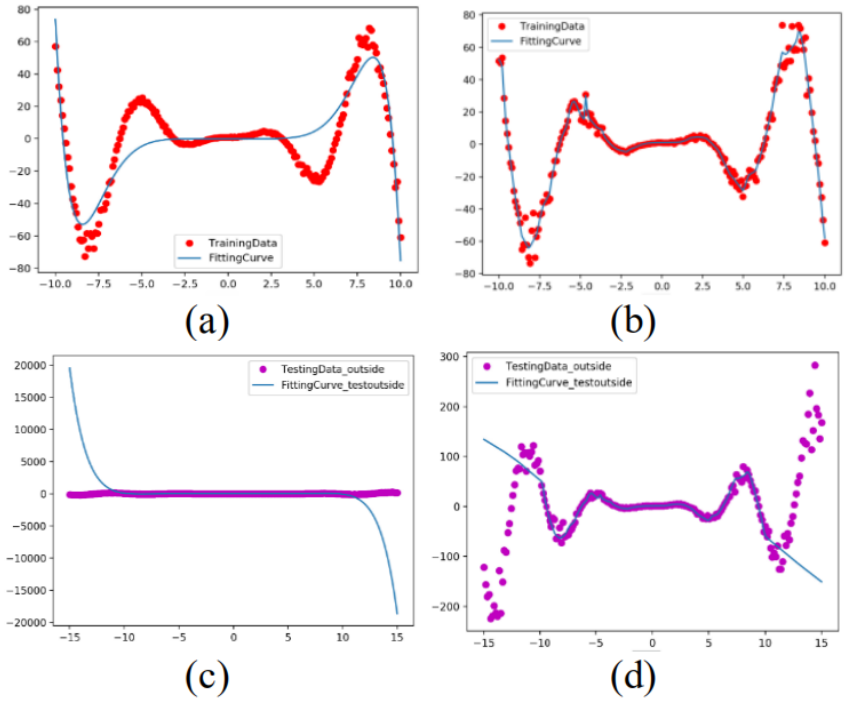


Figure 3.5: Results for function 2; (a) Training results by PR, (b) Training results by NN, (c) Testing results by PR, (d) Testing results, by NN.

to find the optimized size for practical problems. In this chapter, three different size networks are generated, details in Tab.3.3.

	Hidden Layers Amount	Hidden Layers Size
case1-1	5	100
case1-2	20	100
case1-3	20	200

Table 3.3: Size of different networks

Parameter tuning is also an effective method to achieve the optimal convergence point of whole areas. Plus, learning rate is essential for the whole model. Too large learning rate leads to slow the convergence speed even not be able to converge at an optimal point. However, too small learning rate also causes tardy convergence speed and needs more time. For this dilemma, a properly appropriate learning rate is considerable. A big learning rate contributes to accelerate training process but causes negative effect in later period because of oscillation. In contrast, although a small learning rate can avoid oscillation problem but costs much more time. A compromise method is to add

decay to learning rate, such as step decay and exponential decay. The cases with different decay methods are also attached in Tab.3.4.

	Initial learning rate	Decay
case2-1	0.01	N/A
case2-2	0.001	N/A
case2-3	0.001	0.9 EXP

Table 3.4: Learning rate schedulers of different networks

Density of training data points can also cause unexpected influence. One model may behave well when training data points are given sufficiently, but there is still the possibility of worse performance when density of training data decreases. Comprehensive checking of the model performance is an indispensable part in evaluation. Density configurations are shown in Tab.3.5.

	Data points amount
case 3-1	200
case 3-2	500
case 3-3	1000

Table 3.5: Data points density configuration

The fitting results of Tab.3.3 (about different network size) are illustrated in Fig.3.6 The fitting results of Tab.3.4 (about different learning rate) are shown in Fig.3.7. Analogously, the results of Tab.3.5 (about different data density) are demonstrated in Fig. 3.8. Furthermore, loss values of all cases are summarized in Tab.3.6.

Observing the loss value of training/testing, a few key points can be concluded: 1) By increasing the size of network, both testing loss inside and outside decreased (from case 1-1 to case 1-3). However, a deeper NN tends to suffer from the over-fitting and is also hard to accelerate. 2) By adding decay to default learning rate, the testing loss can be reduced effectively (case 2-1 and 2-3). Another noteworthy point is in case 2-2, under the case of too small learning rate, the final results become over-fitting situation and leads to higher testing loss inside. It is important to choose an initial rate and add decay rate cautiously. 3) Along with the increasing of data density, which is same as the amount of data points, the NN model is harder to drop in over-fitting trap and easier to obtain lower testing loss.

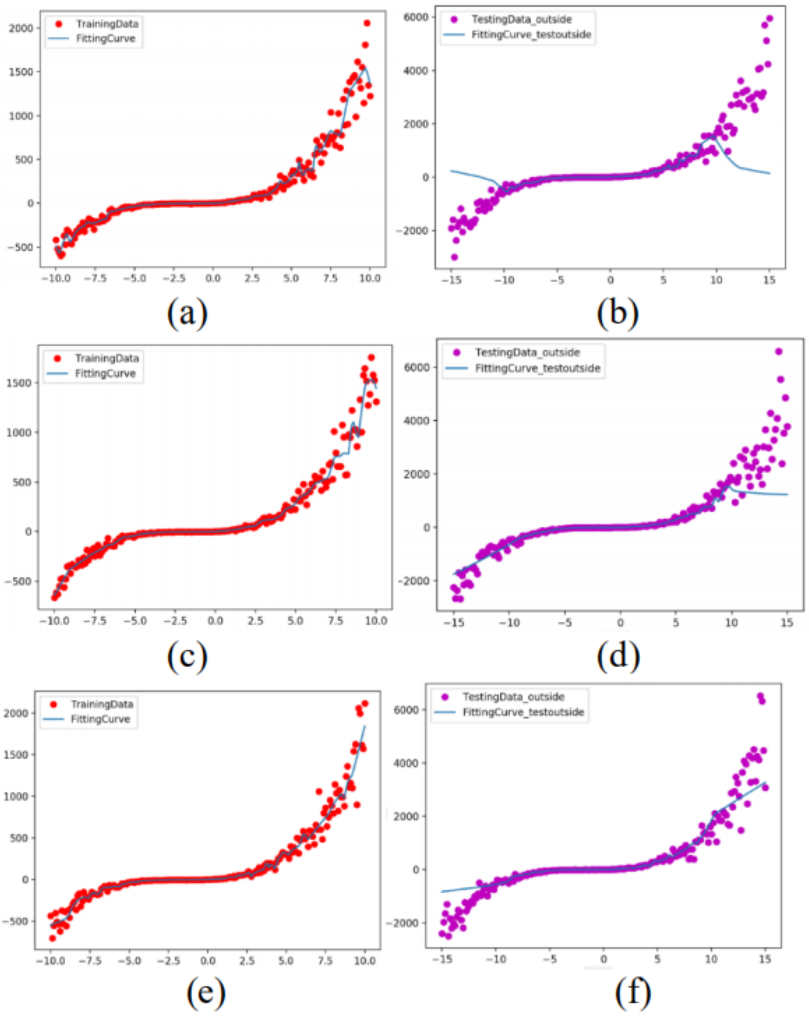


Figure 3.6: Training (red) and testing (purple) result of different network sizes. (a)(b) case 1-1, (c)(d) case 1-2,(e)(f) case 1-3.

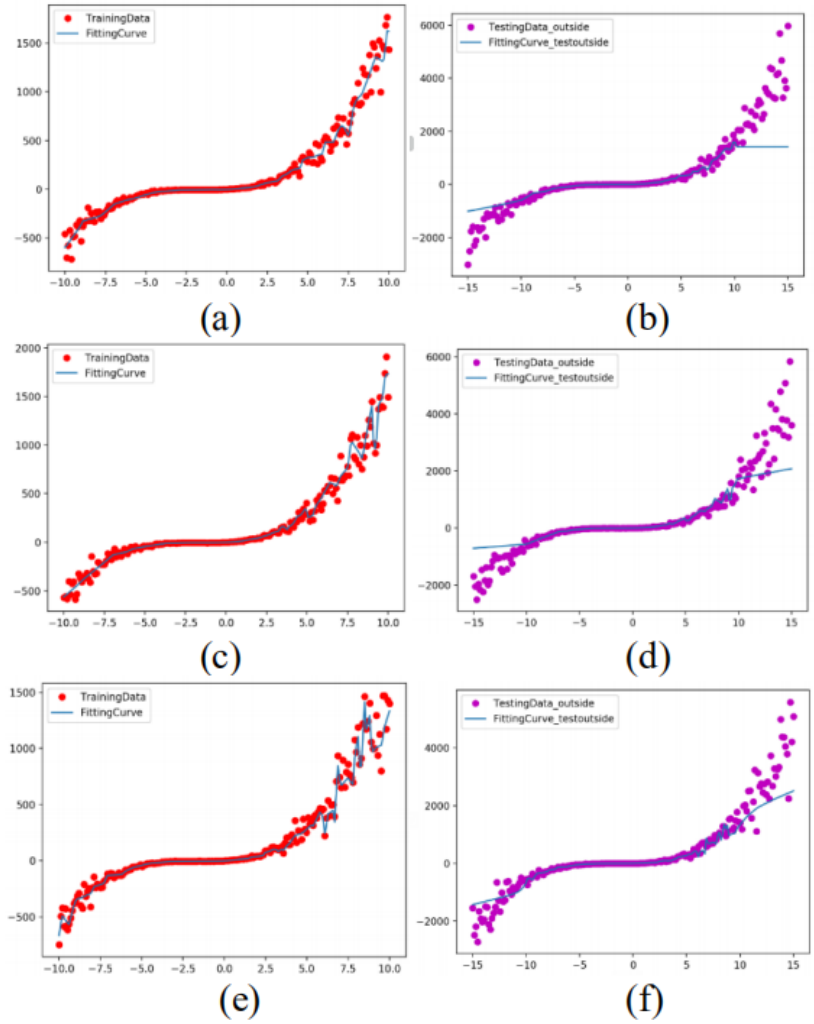


Figure 3.7: Training (red) and testing (purple) result of different network learning rates. (a)(b) case 2-1, (c)(d)case 2-2, (e)(f) case 2-3.

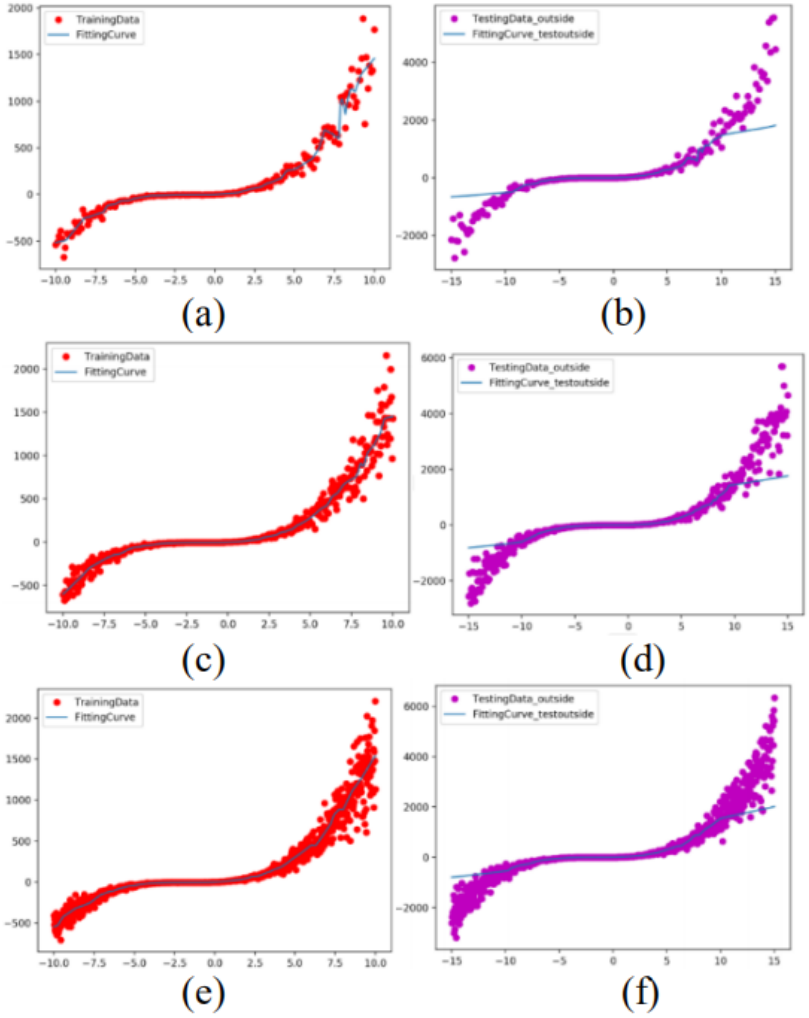


Figure 3.8: Training (red) and testing (purple) result of various data density. (a)(b) case 3-1, (c)(d) case 3-2, (e)(f)case 3-3.

	Training Loss	Testing Loss	Testing Loss (Prediction)
Case 1-1	6.72k	4.38k	1734k
Case 1-2	**4.57k**	2.99k	699k
Case 1-3	8.36k	**2.69k**	**328k**
Case 2-1	4.72k	3.86k	777k
Case 2-2	**3.01k**	6.03k	462k
Case 2-3	3.27k	**3.23k**	**323k**
Case 3-1	**6.42k**	7.93k	633k
Case 3-2	9.09k	4.07k	537k
Case 3-3	10.73k	**2.81k**	**491k**

Table 3.6: Loss of different NN models

3.4 Summary

This chapter proposes an approximate computing experiment with its result for function mapping based on neural network and makes comparison with existing polynomial regression method. Basically, neural network performs better on reduction of loss than polynomial regression method. However, deeper network is also harder to train and easier to drop in the bucket of over-fitting on simple functions.

The uncomplicated structure of the neural network reveals the limitations of making numerical predictions outside the input range. Another obvious drawback is that due to the nature of the neural network itself, it does not learn as well as it could for discrete functions that are not continuous.

Chapter 4

Approximate Decomposition of Multiple Look-up Tables under Acceptable Error Tolerance

Abstract

Approximate calculations are widely used in electronics design automation and optimization to help achieve effective area compression and complexity reduction. For more complex multiple lookup table logic, a common problem is that the size is too large to be deployed directly on the memory cells of configurable logic devices. The subject of this chapter is a novel approximation method based on a depth-first search and partitioning algorithm by setting reserved bits in order to decompose larger size LUTs into combinations of smaller LUTs and finally generate approximate LUT combinations with an acceptable threshold accuracy. The experiments are performed on 4-bit/8-bit multiplier logic.

Unlike traditional logic synthesis-based circuit hardware, logic devices like FPGAs or SRAM-based logic devices have programmable functions to deploy the designed circuit logic. Through functional simulation, the truth table of the circuit logic, i.e., the input and output relationships of the logic, can be obtained, as shown in Fig.4.1(a). Assuming that a memory unit is large enough to store the truth table, the logic can be implemented directly using a single memory unit.

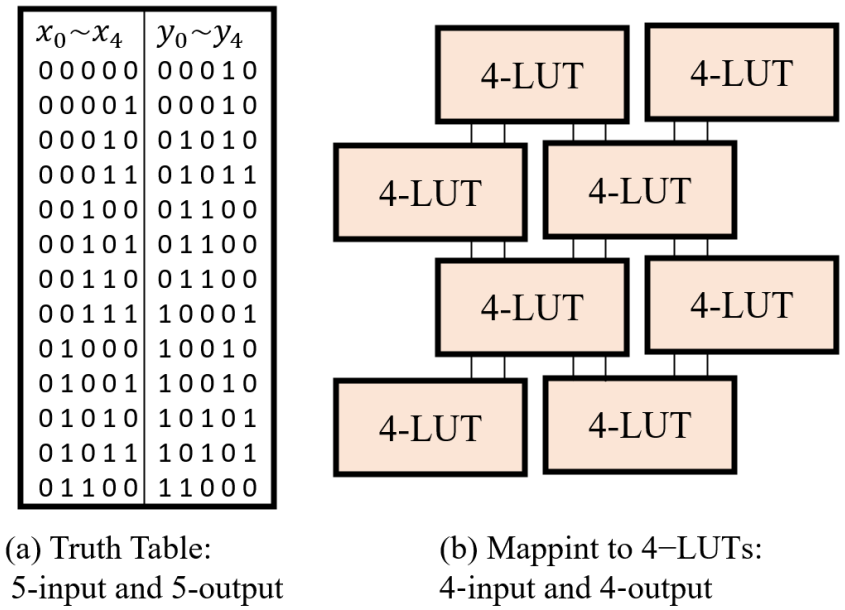


Figure 4.1: Mapping to LUTs

However, in a real reconfigurable hardware device, we have to map the truth table into a set of look-up tables (LUTs), which are interconnected as shown in Fig.4.1(b). An LUT is a form of table that holds predefined information in an array-like entry format that is easily accessible. In Boolean logic, an N-bit LUT can encode an N-input Boolean function by storing the corresponding truth table. In the N-bit case, the LUT has 2^N rows, each corresponding to one possible bit pattern. The input of the Boolean function drives the LUT to access the value of the corresponding output stored in the array.

In general, the input and output sizes of a given truth table are much larger than the LUTs of the device. Therefore, decomposing the larger LUT logic into smaller combinations of LUT logic and maintaining a relatively low and acceptable error rate facilitates the efficient use of reconfigurable hardware devices and reduces the difficulty of mapping larger LUT logic.

4.1 Approximate Decomposition for LUT

In this chapter, we provide a new idea to decompose the truth table of a given logic to generate a set of smaller LUTs, keeping the intrinsic logic intact. The basic approach is to decompose a LUT into smaller LUTs that are connected in a cascade. A key idea is the setting of reserved bits, which refers to reserving a portion of the input bits of a relatively preceding sub-LUT for a later LUT in a cascade connection.

4.1.1 Definition of Question

A universal LUT optimization definition is given as follows: for a given LUT and its approximation LUT, the Boolean logic of them are F and F'. The functions of both logics are represented as $Y = F(X)$ and $Y' = F'(X)$ (We label approximation relative variables by the apostrophe, lso in later). For the same input set $X = \{x_0, x_1, \ldots, x_n\}$, there are different output set $Y = \{y_0, y_1, \ldots, y_n\}$ and $Y' = \{y'_0, y'_1, \ldots, y'_n\}$. To evaluate the performance of approximation results, we define the correct rate R_c, which is defined as follows:

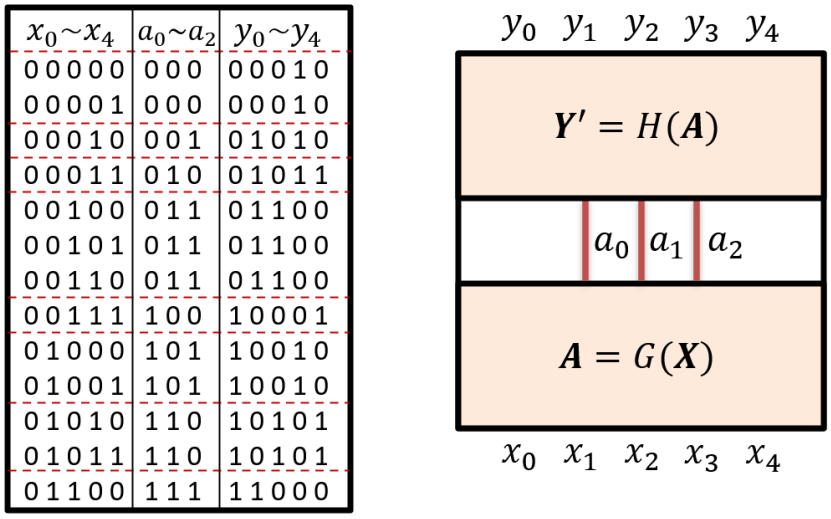
$$C_i = \begin{cases} 0, y_i \neq y'_i \\ 1, y_i = y'_i \end{cases} \tag{4.1}$$

$$R_c = \frac{\sum_{i=1}^{n} C_i}{n} \tag{4.2}$$

For Boolean logic, an N-bit input Boolean truth table can represented by an N-bit LUT completely. While for a lot of actual Boolean logics, some different inputs do not always map different output values but the same ones, which generates repeated output values and provides the space of optimization. By counting the amount of the same outputs individuals to understand their distribution, we can efficiently decompose LUT and generate approximation within an acceptable error tolerance rate. To decompose LUT with large size effectively, we utilize the methodology of block dividing: For each nonrepeat output value, we divide the inputs corresponding to the same output value into the same group and count the amount of output groups. Obviouly, for N-bit LUTs with d non-repeat output values, d is definitely not bigger than 2^N, which reveals potential optimization space.

An the example shown in Fig. 4.2, a decomposed 4-bit LUT are combined of two parts. The first part $A = G(X)$ works like an encoder, which encodes input X to intermediate variables A. And the second part $Y' = H(A)$ works like a decoder,

decoding intermediate variables A to Boolean output Y'. For the whole process, there is $Y' = F'(X) = H(G(X))$.



(a) LUT partition (b) Decomposed LUTs

Figure 4.2: Example: Decompose a 5-bit LUT

4.1.2 Overall Decomposition VS Reserved Bit Decomposition

About how to decompose a LUT, we propose two different ways to compare: overall decomposition (OD) and reserved bit decomposition (RBD). For more intuitive, a 4-bit LUT optimization comparison is showed in Fig. 4.3. The original LUT is of 4-bit inputs and 4-bit outputs, representing it simply as $(4, 4)$. In Fig. 4.3 (a), it has been decomposed to two new LUTs, $(4, 3)$ and $(3, 4)$ respectively. And in Fig. 4.3 (b), LUTs are of size $(3, 2)$ and $(3, 4)$. And the input x_3 is the reserved bit, which connects the sub-LUT, $Y' = H(A)$.

An attempt of decomposing a 4-bit LUT are shown in Fig.4.3 to make a brief explanation. In Fig.4.3(a) using the OD method, the condition for lossless decomposition is that the amount of non-repeat individuals of the output value set Y_1' is not over the maximum possibilities that the digits a_0 a_2 can express, that is, $2^3 = 8$. In Fig.4.3(b) using the RBD method, the condition for lossless decomposition is that for each value (0 or 1) of the reserved bit x_3, the number of non-repeat individuals of output value set Y_2' is not over the maximum possibilities that the digits a_0 a_1 can express $2^2 = 4$. In Fig.4.3(c), the output bits of the first sub-LUT $A = G(X)$ are further reduced on

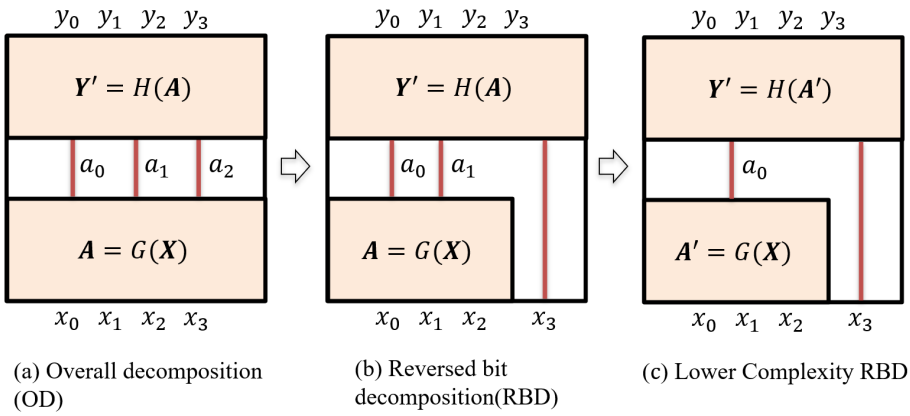


Figure 4.3: Progressive LUT decompositions

the basis of Fig. 4.3(b). In actual situations, this step will be executed if there is still rest optimization space after the RBD is completed. If not, this step will be skipped.

Therefore, it is clear that: If an N-bit LUT can be decomposed losslessly by RBD, it can also be decomposeed losslessly by OD. On the contrary, if an N-bit LUT can be decomposeed losslessly by OD, it may not be able to be decomposed losslessly by RBD. Adding reserved bit reduces rest optimization space. To classify reserved bits does reduce the size of decomposed LUTs, while add restrictions to LUT decomposition.

In a short summary, if the result of OD is within the error tolerance, then further try to use RBD. If the RBD decomposition succeeds to obtain a feasible solution, further attempt will be made to reduce the intermediate transition output bits of the sub-LUT, so as to reduce the number of terminals of the sub-LUT and reduce the subsequent mapping cost.

4.1.3 LUT Decomposition Algorighm

Although it actually imports extra cost, the method of decomposeing larger LUTs to smaller ones make it easier to use reconfigurable hardware device to approach designed complex circuit logic. The decomposition cost can be evaluated from two indicators: one is the total additional size increment of the decomposed LUTs compared to the original LUTs, and another is R_c, the accuracy of the output of the decomposed circuit compared to the original LUTs. For the given and fixed LUTs logic and its decomposition, two factors are related to the size of the extra cost. The amount of non-repeat output values d and configuration of reserved bits. The two factors determine how many bits transition input and output $(a_0 - a_2)$ can be reduced in the

intermediate process of decomposition, the size of the extra LUT size cost and the final accuracy result, R_c.

In OD process, it is not difficult to calculate the R_c from counting the non-repeat output value individuals. Assuming the output value set Y, having d different non-repeat output values. For an N-bit LUT and $2^N > d > 2^{N-1}$, the decomposition loss will be the $(d - 2^{N-1})$ groups which owns the least entries. Thus, R_c is available to be calculated. For RBD, it is unavailable to obtain the R_c of the whole LUT due to the existing of reserved bits. Because it is possible that same output values in the same divided group, while correspond unequal reserved bits values. Therefore, we propose a methodology of divide-and-conquer, dividing the whole LUT into separate blocks and count the total loss to calculate the R_c of the whole LUT. The process of dividing LUT according to reserved bits has been shown in Fig. 4.4.

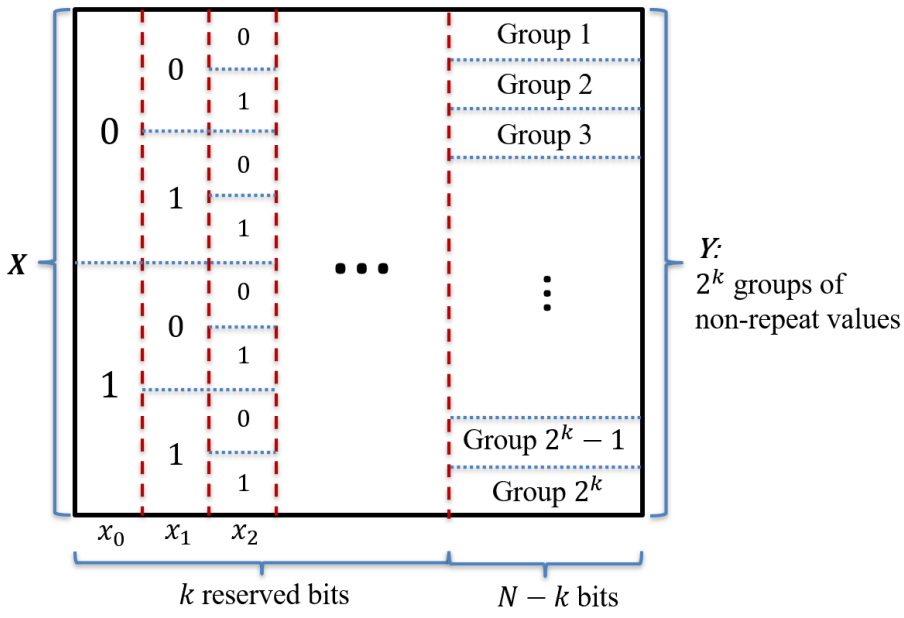


Figure 4.4: LUT decomposition process with reserved bits

Another factor influencing dividing process and final performance is the choice of reserved bits, because different bit patterns have different weight of influence on internal logic. For an N-bit LUT, the choice space of k reserved bit is C_N^k, and the total space $\{C_N^1 + ... + C_N^N\}$. To find optimal solutions, we use depth-first search in the total space. The pseudo Algorithm. 3 is given to explain the procedure.

Algorithm 3 Decomposition process of LUT

Require: Y: the output of N-bit LUT L, R_{cm}: required minimal correct rate
Ensure: RES: LUT decomposition result

```
RES = {}
for k in range(0, N − 1) do
    space = Permutation(C_N^k) /* Search space: the permutation of reserved bit selection */
    length = len(space) /* the length of search space */
    for i in range(0, length) do
        res_i = Decompose(Y, space[i]) /* Decompose LUT L by selected reserved bits */
        R_c = CalculateLoss(res_i) /* Calculate the correct rate R_c */
        if R_c > R_cm then
            RES.append(res_i)
        end if
    end for
end for
return RES
```

4.2 Experimental Results

In this section, we test the LUT decomposition methodology on cases of basic arithmetic multiplier logic, using an 4-bit multiplier logic and a 8-bit multiplier logic respectively. For the 4-bit multiplier logic, the input part consists of two parts $i_1 = x_0x_1x_2x_3 \& i_2 = x_4x_5x_6x_7$, each part as two 4-bit binary numbers. And the output $i_1 \times i_2$ is an 8-bit binary number. The 8-bit multiplier logic is similar and includes two 8-bit input and one 16-bit output. The error rate limitation is set as 30% and the results beyond it will be droped. The operation result of the LUT decomposition methodology under 4-bit and 8-bit multiplier logic are shown in Tab. 4.1.

The LUT decomposition methodology optimizes the two indexes under the error rate tolerance: the amount and selection of reserved bits, and the another is the maximum decomposition accuracy of the remaining part, the sub-LUT. The benefits and additional costs under each approximate solution are also shown in Tab. 4.2. Experimental results show the decomposition of the 4-bit/8-bit multiplier logic and obtain 2-4 bit reduction within error rate of 5.4%-23.4% / 0-19.4%.

Operation time: 4-bit 2.271 s, 8-bit about 44min. Operated under the hardware environment of Intel i7-6700HQ and 16GB RAM.

Case	Method	LUT Size	Reserved Bit	Correct/Total
4-bit MUL	OD	(8,6)+(6,8)	N/A	215/256
	RBD	(7,6)+(7,8)	x_3	242/256
		(6,5)+(7,8)	x_3, x_7	229/256
		(5,4)+(7,8)	x_2, x_3, x_7	196/256
		(4,3)+(7,8)	x_1, x_2, x_3, x_7	183/256
4-bit MUL	OD	(16,14)+(14,16)	N/A	63288/65536
	RBD	(15,14)+(15,16)	x_7	65536/65536
		(15,13)+(14,16)	x_7	52784/65536
		(14,12)+(14,16)	x_7, x_{15}	48711/65536
		(13,12)+(15,16)	x_6, x_7, x_{15}	57611/65536

Table 4.1: 4-bit/8bit multiplier logic decomposition

Case	LUT Size	Method	Total Bit Reduction	Correct Rate	Additional Size Cost
4-bit MUL	(8,6)+(6,8)	OD	2	0.840	25%
	(7,6)+(7,8)	RBD	2	0.946	50%
	(6,5)+(7,8)		3	0.895	25%
	(5,4)+(7,8)		4	0.766	12.5%
	(4,3)+(7,8)		5	0.715	6.25%
8-bit MUL	(16,14)+(14,16)	OD	2	0.966	100%
	(15,14)+(15,16)	RBD	2	1.0	50%
	(13,12)+(15,16)		4	0.879	12.5%
	(15,13)+(14,16)		3	0.806	50%
	(14,12)+(14,16)		4	0.744	25%

Table 4.2: Decomposition Performance Summary

4.3 Summary

We have introduced a divide-and-conquer methodology focusing how decompose a large truth table into smaller LUTs and combine them to the approximation of original truth table. The experiments of universal multiplier logic have been conducted and the result shows that it is available to generate approximate truth tables under an acceptable error tolerance by reasonable selection of reserved bits and decomposition accuracy. This reduces the difficulty of mapping large-scale LUT logic to finite input/output programmable logic circuits effectively.

As the data results shown in Section 4.2, there are multiple approximate LUT decomposing solutions for the same single LUT complexity reduction, and corresponding to different error rate and additional cost of LUT scale selection.

On the other hand, because of the limitation of decomposition and exhaustive search in computational efficiency, when facing large scale LUTs. This will also be one of our future directions of exploring the way to search optimal decomposition solutions of larger size truth tables more efficiently.

Chapter 5

MLUTNet: A Neural Network for Memory based Reconfigurable Logic Device Architecture

Abstract

Neural networks have been widely used and implemented on various hardware platforms, but high computational costs and low similarity of network structures relative to hardware structures are often obstacles to research. In this chapter, we propose a novel neural network in combination with the structural features of a recently proposed memory-based programmable logic device, compare it with the standard structure, and test it on common datasets with full and binary precision, respectively. The experimental results reveal that the new structured network can provide almost consistent full-precision performance and binary-precision performance ranging from 61.0% to 78.8% after using sparser connections and about 50% reduction in the size of the weight matrix.

5.1 introduction

5.1.1 Motivations and Contributions

When deploying neural networks to MRLD-like programmable logic hardware, there are two most critical issues: storage cost and architecture conversion. Some works on limiting the numerical accuracy of neural networks have been presented, but how to structurally reorganize to cut the complexity and storage expense of the network is still a meaningful problem to be solved.

Based on the starting point of solving the above problem, we propose a new neural network based on the MRLD structure named MLUTNet. In MLUTNet, we adopt a network topology similar to the MRLD structure to partition the middle layer of the network in order to reduce the storage expense of neural network deployment on hardware platforms effectively. The contributions of this chapter are as follows.

- We propose MLUTNet, a novel neural network with an atypical structure.MLUTNet combines the advantages of two aspects: the efficient learning performance of neural networks and the similar structure of MRLD, which makes MLUTNet easy to implement on MRLD or other similar logical storage devices without much extra effort.
- We conducted experiments and compared the results with their MLUTNet versions on three different popular datasets using standard and binary neural networks as the baselines, respectively. Compared to a fully connected neural network of the same size, MLUTNet saves over 50% of the weight matrix storage space and also reduces the size of individual weight matrices efficiently, with acceptable performance loss in accuracy on the dataset.
- The method is simple and easy to implement with good scalability; the MLUTNet version of a particular network can be substituted for the original network at no additional cost to meet the need to reduce the size of the network. This is very friendly for subsequent extension studies.

5.1.2 Organization of the chapter

The chapter is organized as following:

- Section 5.2 presents an explanation of the proposed MLUTNet and its operation principle.

- Section 5.3 shows the experimental results and analysis.
- Section 5.4 concludes and summarizes.

5.2 MLUTNet

5.2.1 Network Definition

Assuming an end-to-end approach to train directly on full precision neural networks, i.e., using the input-target data pairs of data as training data and then mapping the trained neural networks to MRLD, we will face many potential problems. 1) The hidden layers of standard neural networks are usually fully connected, the size of individual weight matrix is large, and the number of connections grows exponentially with the size of the weight matrix. From the SRAM storage point of view, it is unrealistic to achieve a reasonable mapping on MRLD without limiting the size of the weight matrix and the number of connections. 2) The parameter storage and arithmetic processes inside the standard neural network are executed at full precision by default. However, most programmable logic devices support limited precision, and the operational expenses increase dramatically when the precision is higher. Therefore, it is important to use a neural network model with limited precision to find a balance between accuracy and cost. 3) The nonlinear activation functions commonly used in neural networks, such as *Sigmoid* and *tanh*, usually require proprietary resources for implementation on programmable logic devices and can only be approximated with limited accuracy. Although the expense of a single operation is not significant, considering that each parameter of the hidden layer weights is involved in the activation operation, a more sensible measure is to switch to other activation functions that are more friendly to hardware mapping.

To address the above issues, we use the following measures to improve them. For 1), we import the methods of splitting the hidden layer and neighbor-only connections. Since MRLD devices are composed of multiple MLUTs as basic cells, each MLUT can be regarded as a small SRAM array with data storage or logic wiring or both. We split the originally larger hidden layers according to the MRLD topology, perform the computation and parameter update separately, and merge them into the final result after the computation is completed. And for the connections between the split weight matrices, we utilize neighbor connections to reduce the number of connections and also play a role in preventing overfitting. For 2), we import the idea of low precision neural network, BNN, as the comparative alternative to the full-precision standard

neural network. BNN uses a 1-bit representation for the weights and XNOR-bit operation instead of the usual multiplication operation. Compared with full-precision multiplication, both the software training cost and the hardware computation cost can be effectively reduced. For 3), we use the *ReLU* function [NH10] and *Htanh* functions as activation functions instead of the traditional *Sigmoid* and *tanh*. Benefiting from their simple numerical logic,both can be implemented by linear combinations of logic gates.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{5.1}$$

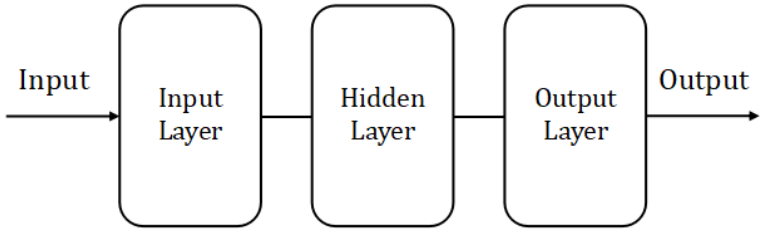
$$\mathrm{Htanh}(x) = \max(-1, \min(1, x)) \tag{5.2}$$

5.2.2 Training and Connection of MLUTNet

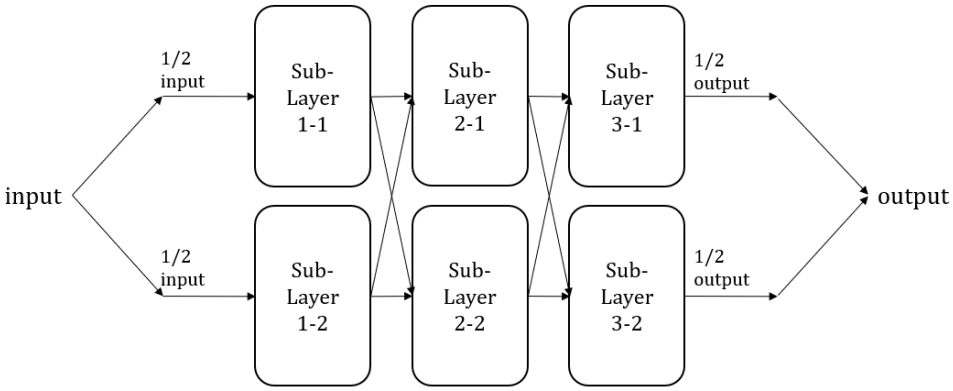
Based on the above analysis, we propose a new artificial neural network structure, MLUTNet. Fig. 5.1 shows an illustration of the conversion and comparison of a single hidden layer neural network with MLUTNet.

In addition to having the learning potential of a standard neural network, the weight matrix separated structure and sparse connection design employed by MLUTNet make it more friendly for subsequent mapping and implementation on MRLD and similar hardware. Based on the trained MLUTNet, the weight matrix can be mapped to MLUT units and wired on the MRLD depending on the network connectivity. If the split matrix still exceeds the unit storage limit of the device, then it needs to be stored separately in multiple memory units while increasing the cost of data exchange in the network implementation. It means that there will be multiple MLUTs that are combined as a larger generalized "Big-MLUT" within which the weight matrix is stored and data is exchanged.

A key point of interest is how each sub-layer should be connected to the next sub-layer between adjacent hidden layers. A plain and natural idea is to use full connectivity in the same way that neurons within a hidden layer are connected to each other. But unfortunately, this approach is not feasible. On the one hand, it is difficult or impossible to fully interconnect the MLUT units storing the weight matrix due to the limitation of the number of connections within the hardware and cost considerations. On the other hand, for MLUTNet, full interconnection between sub-layers does not necessarily enhance performance, but may cause degradation of network performance by reducing the sparsity of the network, as reflected in the experimental data in Section 5.3.



(a) NN

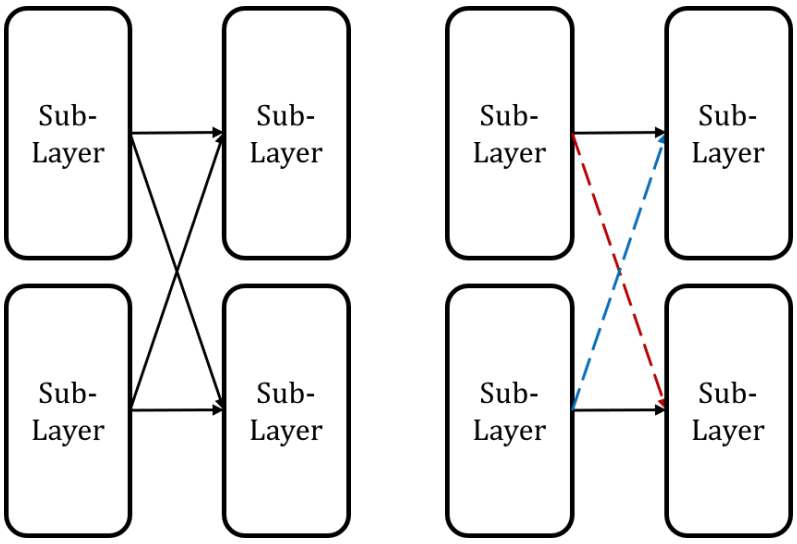


(b) MLUTNet

Figure 5.1: One NN and MLUTNet

Considering the need to fit the structure of MRLD as closely as possible and to reduce the obstacles for subsequent implementations, we use neighborhood connection, as shown in 5.2. The connections between sub-layers will be dropped by some determined logic (e.g., red connection are dropped in the odd number of layers and blue ones in even layers). If a sub-layer has input from more than one sub-layer, the input it receives is summed up as the new input. This connection method has several advantages: firstly, sparse neighbour-only connections make the network structure as similar as possible to the MRLD topology, reducing barriers to subsequent implementation; furthermore, sparse inter-layer connections reduce the possibility of over-fitting; and finally, a smaller number of connections reduces the computational and memory overhead of the network. The disadvantage is that each sub-layer can only affect its neighboring sub-layers, so the network needs to be deep enough to ensure learning capability.

As a result of the above discussion, the network logic of MLUTNet was finalised as follows. Layers of the network are split into multiple sub-layers, and no connection is created between sub-layers of the same layer. The activation function is attached after each sub-layer. During the training process of the network, the training of sub-layers in the same layer is run in parallel. In the input layer, the input data matrix is cut equally along the Y-axis and used as the output of the sub-layers of the input layer, respectively. And at the output layer, the outputs of the two sub-layers are combined along the Y-axis and used as the final output.



(a) Full **(b)** Neighbourhood

Figure 5.2: Full connection and neighbourhood connection

To make the process more understandable, we illustrate the process with a segment of

actual MNIST data going through MLUTNet. As shown in the Figure. 5.3, the test data is a gray-scale image with an initial dimension of 28×28 and the content is a handwritten number whose label is a one-dimensional vector of size 10. This vector is one-hot, i.e., only one element is 1 and the rest is 0. The x-axis coordinate of element 1 represents the content of the image, i.e., which of the numbers 0 to 9 is the image. Suppose we use a batch size of 100 for each training, i.e., the initial dimension of the data is $100 \times 28 \times 28$. After binarizing the data, the size of the input data is 100×784. When entering the input layer, the input data is divided into two matrices of size 100×392. It is then multiplied with the weight matrix of size 392×392 in the hidden layer. In the output layer, we will finally get two matrices of size 100×5 and merge them into the final result of 100×10. For more specific process details, the MLUTNet generic training flow written according to the algorithm format is shown in Algorithm 4.

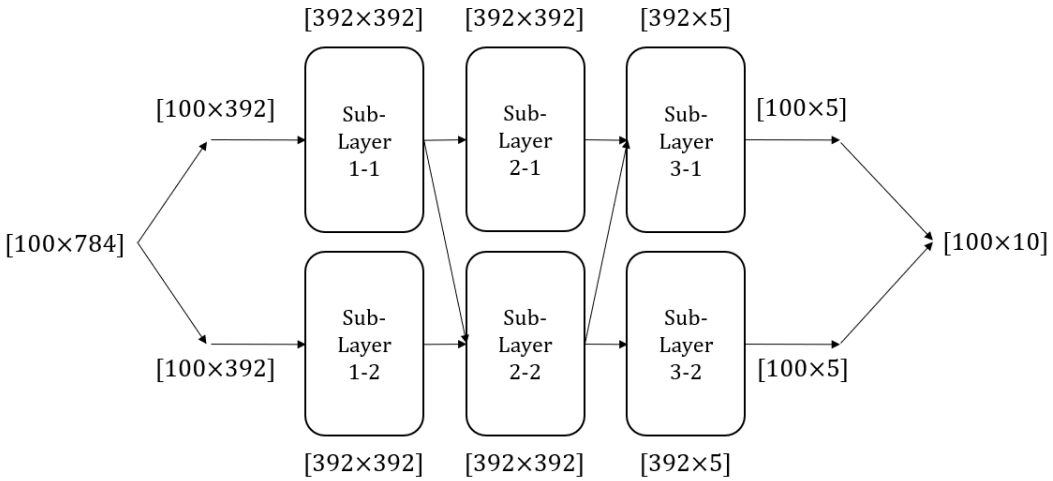


Figure 5.3: Dataflow in MLUTNet on MNIST

5.3 Experimental Results

We configure four types of network models, standard neural network(NN), binary neural network(BNN), MLUTNet neural network(MLUTNet), and binary MLUTNet (B-MLUTNet) neural network. The experimental results are operated under the following hardware environment: Intel i7-6700HQ, NVIDIA GTX 1060 and 16GB RAM. The related codes are implemented by PyTorch framework.

Details configurations about the models are briefly described as follows. The ratio of

Algorithm 4 Algorithm: Train a MLUTNet

Require: the number of network layers L, the number of sub-layers each layer S, weights of network W, activation values a, gradient values g, learning rate η
Ensure: updated weights W^{t+1}, updated learning rate η^{t+1}

{Forward propagation}
for k in range$(1, L)$ **do**
  for m in range$(1, S)$ **do**
    (if Binary) $W_{k,m} \leftarrow Binarize(W_{k,m})$
    $a_{k,m} \leftarrow a^b_{k-1,m} W_{k,m}$
    (if Binary) $a_{k,m} \leftarrow Binarize(a_{k,m})$
  end for
end for
{Backward propagation}
for k in range$(L, 1)$ **do**
  for m in range$(1, S)$ **do**
    $g_{a_{k-1,m}} \leftarrow g_{a_{k,m}} W_{k,m}$
    $g_{W_{k,m}} \leftarrow g^{\top}_{a_{k,m}} a_{k-1,m}$
  end for
end for
{Updating parameters}
for k in range$(1, L)$ **do**
  $W_k^{t+1} \leftarrow Update(W_{k,m}, \eta, g_{W_{k,m}})$
  $\eta^{t+1} \leftarrow Scheduler(\eta)$
end for

training dataset to validation dataset is set as 80%:20%. RMSProp optimizer is used as the optimizer for the experiments. The epochs of the experiments are set to 40. The initial learning rate is 0.001. The learning rate scheduler is set to exponential decay mode and cosine annealing mode respectively, the exponential decay mode halves the learning rate every 10 epochs, the cosine annealing mode period is set to 5, and the "Warm-up" multiplier parameter is set to 2.

5.3.1 Performance on MNIST series datasets

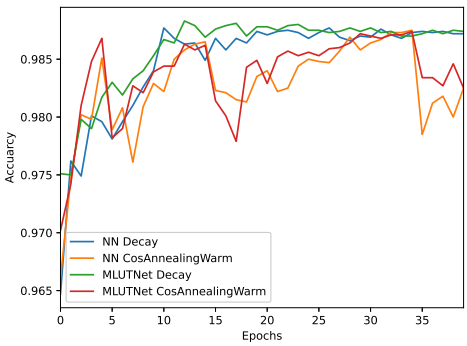
The experiments conducted on three MNIST series datasets: MNIST [LC10], K-MNIST [CBIK+18] and fashion MNIST [XRV17]. In these datasets, the contents of images in datasets are graphics of handwritten numbers, Japanese hiragana characters from ancient books, and fashion items, respectively. The images are gray-scale images with the resolution of 28×28. The accuracy convergence process and the corresponding confusion matrices for each group of experiments are exhibited in Fig.5.4 and Fig.5.5.

By reviewing the experimental results, some conclusions can be revealed. 1) In the full precision case, MLUTNet exhibits comparable performance to the standard NN on all three datasets. With enough epochs, the accuracy of the test dataset for each case converges steadily to approximately the same level. 2) In the binary accuracy case, MLUTNet has different degrees of performance loss compared to the standard BNN. It is the largest in KMNIST with 35.3% and the smallest in MNIST with 20.6%. 3) The performance of the decay scheduler is more stable, but Cosine annealing shows better performance in F-MNIST. The full-accuracy MLUTNet with Cosine annealing scheduler achieves a lead of about 0.71% over the NN with decay scheduler; the binary-precision MLUTNet with Cosine annealing scheduler improves the accuracy by 9.03% over using the decay scheduler.

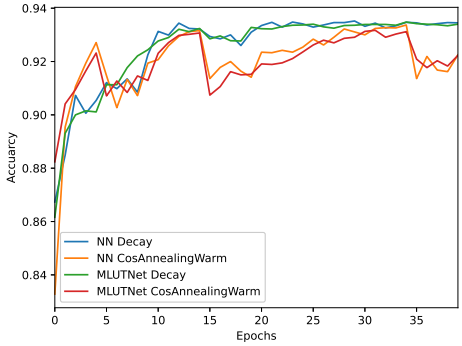
5.3.2 Performance on CIFAR-10 and STL-10 datasets

In the CIFAR-10 dataset, images are divided into 10 categories. Each image is an RGB image with three color channels and a resolution of 32×32. And in the STL-10 dataset, the resolution of the images is further improved to 96×96.

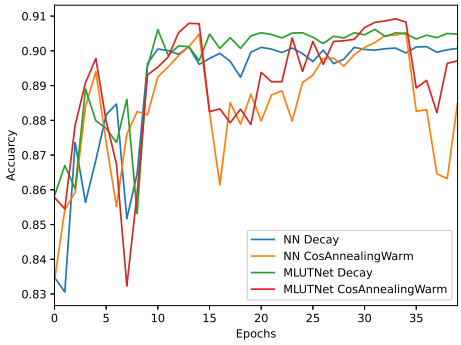
The experimental results on the CIFAR-10 dataset and the STL-10 dataset are shown in Fig. 5.6. At full precision, the NN and MLUTNet achieve about 57% accuracy on the CIFAR-10 dataset and about 45% accuracy on the STL-10 dataset, which is a normal performance for standard structured neural networks. The confusion matrices are shown in Fig.5.7. However, at binary precision, BNN and binary MLUTNet cannot



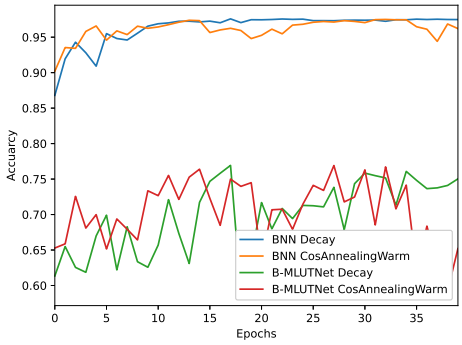
(a) MNIST full-precision



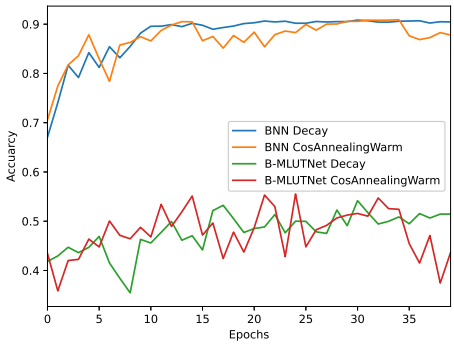
(b) KMNIST full-precision



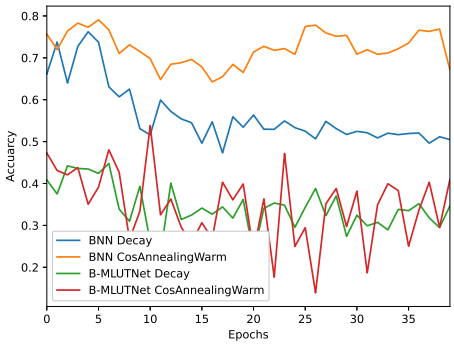
(c) F-MNIST full-precision



(d) MNIST binary-precision

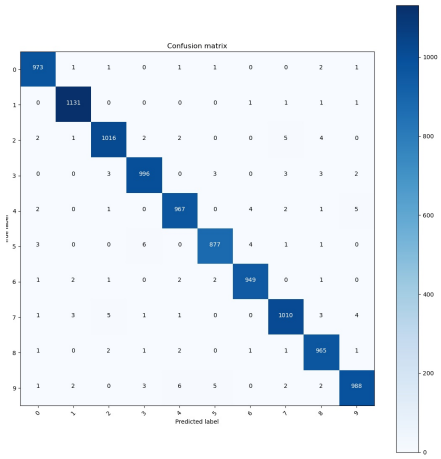


(e) KMNIST binary-precision

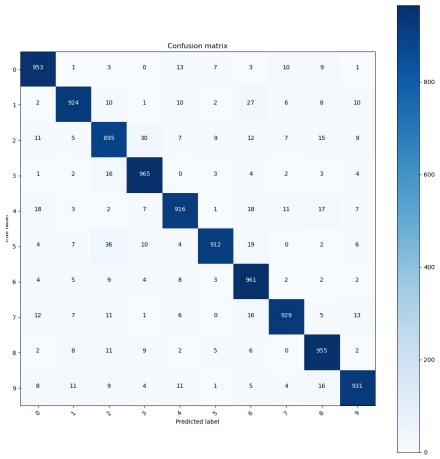


(f) F-MNIST binary-precision

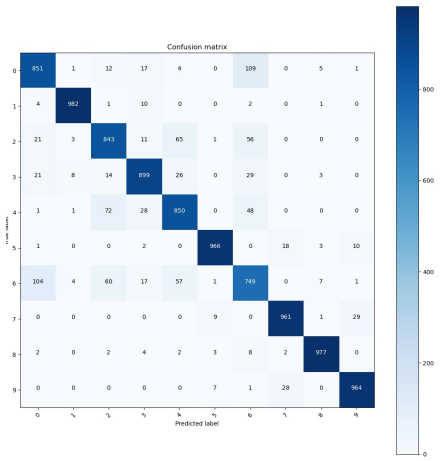
Figure 5.4: Performance on MNIST series datasets



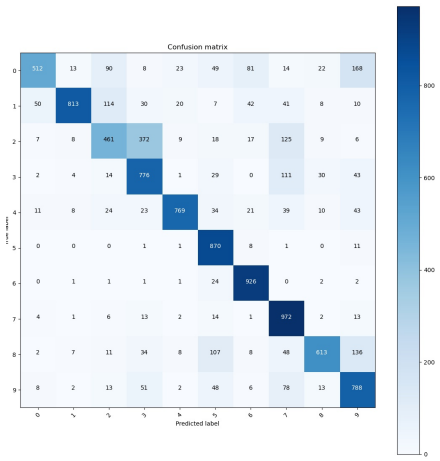
(a) MNIST MLUTNet



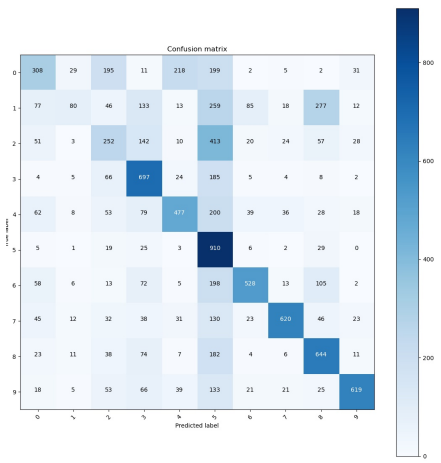
(b) KMNIST MLUTNet



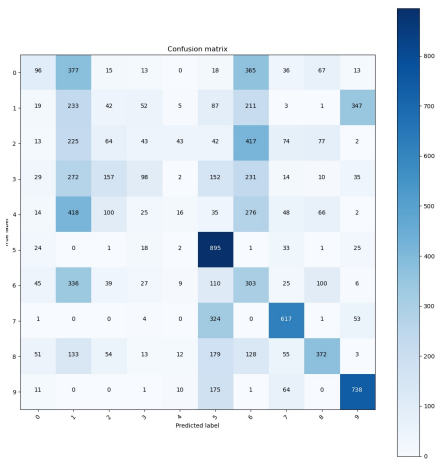
(c) F-MNIST MLUTNet



(d) MNIST B-MLUTNet



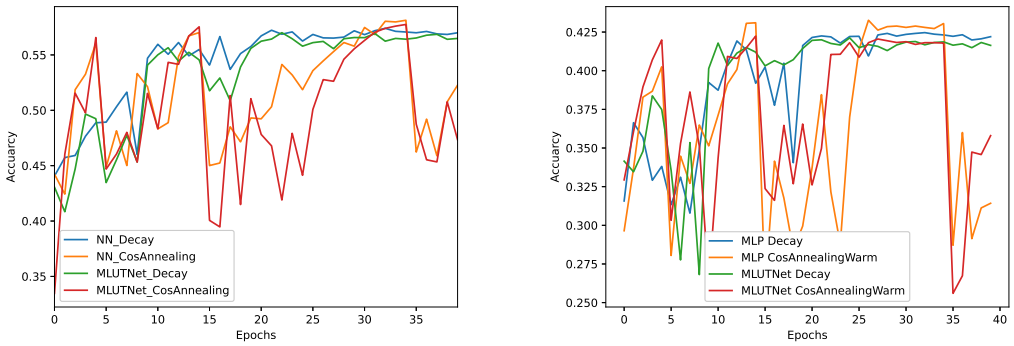
(e) MNIST B-MLUTNet



(f) MNIST B-MLUTNet

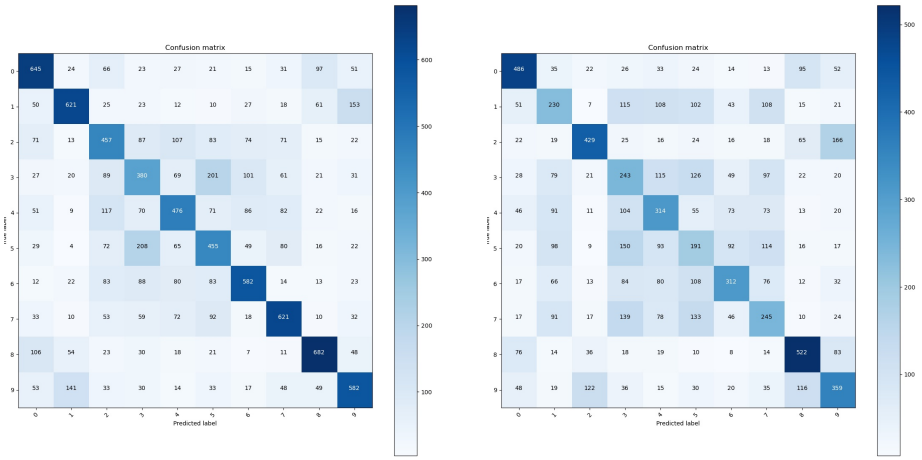
Figure 5.5: Confusion matrices of MLUTNet

learn and converge smoothly due to the limited structural complexity.



(a) CIFAR-10 **(b)** STL-10

Figure 5.6: Performance on CIFAR-10 and STL-10 dataset



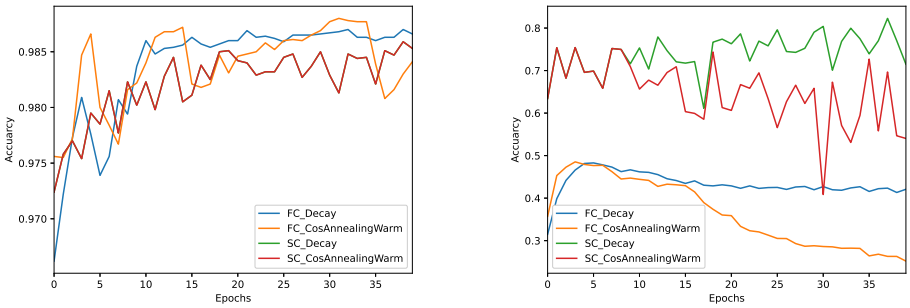
(a) CIFAR-10 MLUTNet **(b)** STL-10 MLUTNet

Figure 5.7: Performance on CIFAR-10 and STL-10 dataset

5.3.3 Sparsely Connection Verification

We conducted a set of comparative experiments regarding the way neighboring sub-layers are connected in MLUTNet. In the experiments, MLUTNet and Binary-MLUTNet are connected according to fully connected and sparsely connected, respectively, and tested under the same dataset, and the results are shown in Fig.5.8.

The fully connected MLUTNet does not show any significant advantage while increasing the computational effort, while the fully connected B-MLUTNet causes a significant decrease in accuracy instead. We speculate that this is because the sparsely connected sub-layers somehow avoids premature overfitting.



(a) On MLUTNet **(b)** On Binary MLUTNet

Figure 5.8: Comparison of fully connection and sparsely connection

5.3.4 Results Summary

A summary of the optimal results of different accuracy models under each model is shown in Tab.5.1. The run-time of models are shown in Fig.5.9.

	NN	MLUTNet	BNN	B-MLUTNet
MNIST	0.9877	0.9883	0.9759	0.7693
K-MNIST	0.9347	0.9348	0.909	0.5552
F-MNIST	0.9053	0.9092	0.7909	0.5382
CIFAR-10	0.5813	0.5775	N/A	N/A
STL-10	0.4305	0.4186	N/A	N/A

Table 5.1: Optimal accuracy performance

On all four datasets, using the results of the standard NN as the baseline, Fig. 5.10 shows the correct rate performance ratio of MLUTNet. The results show that the final accuracy performance of MLUTNet in full-precision is comparable to that of the standard NN model, and the size of the former's weight matrix is about half of that of the latter; in binary-precision, depending on the dataset, respectively, the accuracy performance of MLUTNet ranges from 61% to 78.8% of that of the standard BNN model.

The experimental results demonstrate the effectiveness of MLUTNet. Under full-precision, MLUTNet achieves comparable performance to standard NN using a smaller scale weight matrix and fewer inter-layer connections; under binary-precision, MLUT-Net suffers from the accuracy impact caused by the reduced weight matrix, and the final accuracy is reduced compared to standard BNN.

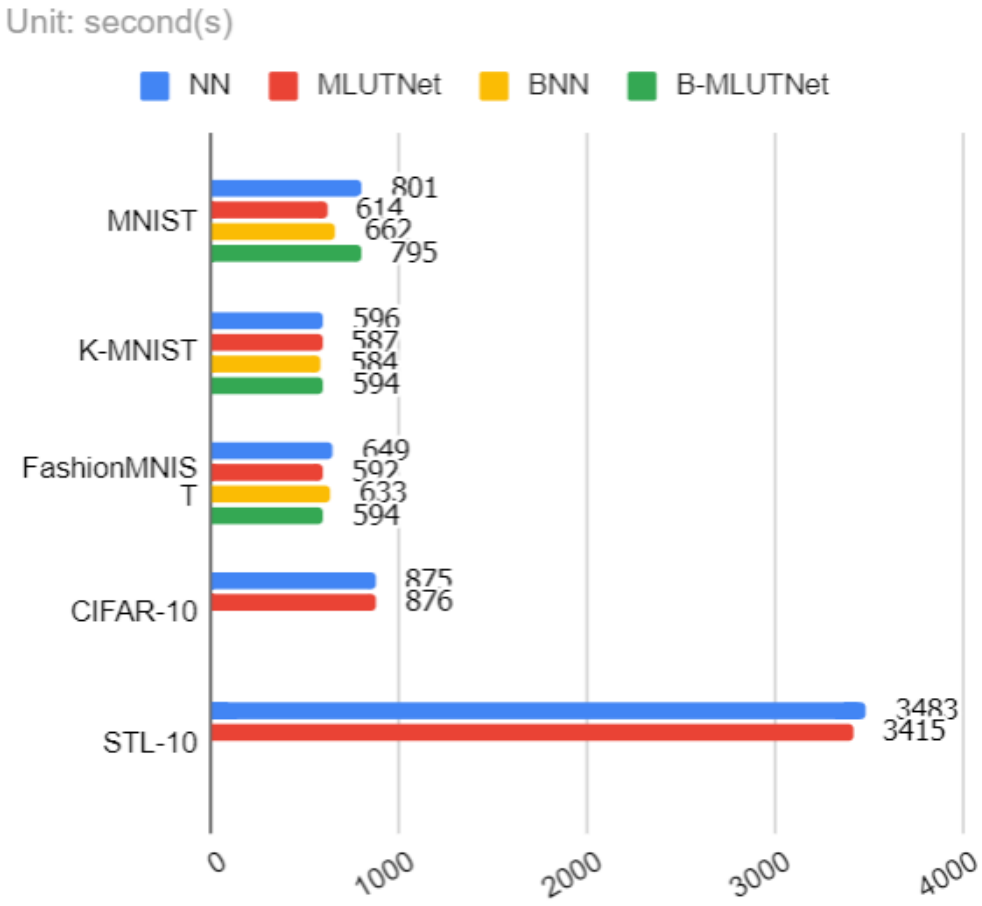


Figure 5.9: Models operation time

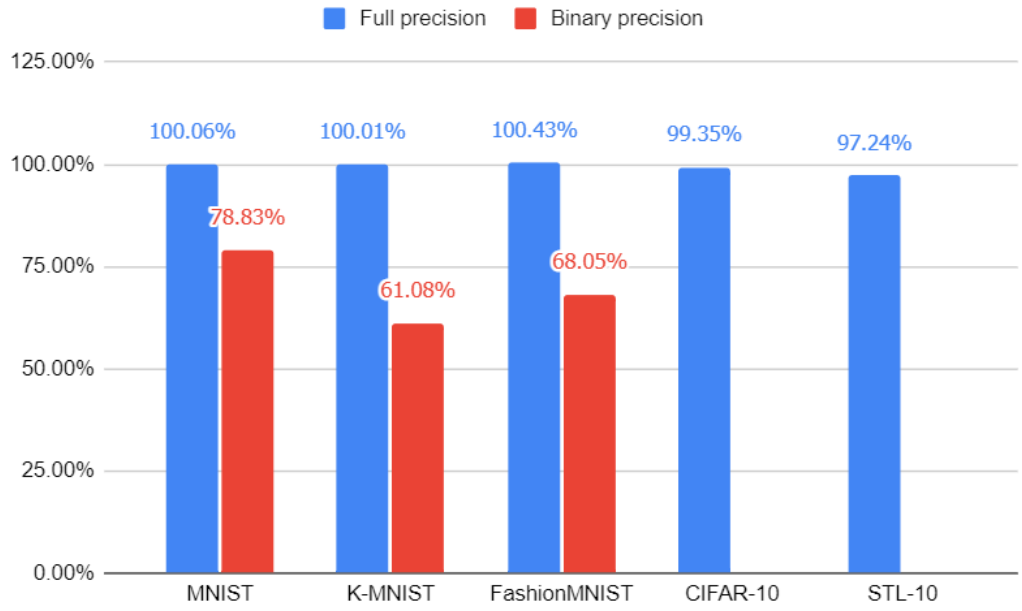


Figure 5.10: Correct Rate Performance Ratio

5.4 Summary

In this chapter, we propose MLUTNet, an innovative neural network structure, by combining the structural features of MRLD devices. In MLUTNet, we import and effectively utilize measures such as binarized weights, scale reduction of the weight matrix, and sparse connection to significantly reduce the computing expense and porting cost of the network and maintain a relatively acceptable performance.

The experimental results reveal that MLUTNet achieves essentially equivalent or slightly better accuracy on test datasets. Moreover, on binary-precision networks, although cutting the weight matrix brings a greater additional precision impact, MLUTNet also achieves 78.8%, 61.0%, and 68.0% of the performance of the standard NN model on the three datasets of the MNIST series, respectively.

In Summary, we have verified the effectiveness of MLUTNet in full-precision on general datasets, and further improving the performance and stability through optimization measures will be the focus of our next step. In addition, in this chapter, due to the limitation of the standard NN's own structure, it cannot learn and converge smoothly on the CIFAR-10 dataset with binarization accuracy. Therefore, further extension of the MLUTNet structure to convolutional neural networks will also be a target of our future research.

Chapter 6

Conclusion

Overall, with the help of customization of neural networks, appropriate algorithmic optimization and improvement of the structural adaptation of the networks, this thesis conducts a study on the reduction of barriers to the implementation of bidirectional logic within a framework based on the cross-application of MRLD hardware and neural networks.

To reach this goal, many research and experimental methods have been applied. These include building networks using programming languages and deep learning frameworks, constructing learning target logic functions, preprocessing data for training, tuning and visualization of network parameters, and underlying customization of basic hidden layer units. Through the research approach as described above, meaningful findings were derived and demonstrated the effectiveness of neural networks for learning multiple lookup table logic, the effectiveness of large-size LUT decomposition algorithms for reducing the complexity of individual LUTs, and the excellent performance of the hardware-inspired neural network MLUTNet, respectively. Overall, the findings are able to strongly support the conclusions in the thesis.

The research included in this thesis together form a framework based on MRLD hardware and neural network learning and implementation, the core components of which are hardware-aware neural network and a look-up table decomposition algorithm. with the help of customization of neural network With the help of customization of the neural network, appropriate algorithmic optimization and improvement of the structural adaptation of the network, the barrier of bidirectional flow of logic between the MRLD hardware and the neural network is reduced. This is a new alternative route for researchers who wish to implement approximate logic at low cost.

In addition to the above stated advantages, the research content and the proposed results of this thesis also contain the following limitations: (1) In the proposed process of learning approximate target multiple lookup table logic using neural networks,

the selection of the target function is more limited by the continuity of the function values. (2) In the proposed larger size LUT decomposition algorithm, the maximum depth of decomposition is very strongly correlated with the internal logic of the LUT, which leads to significant performance limitations (3) For the hardware- aware neural network MLUTNet, its performance is similar to that of the standard NN model at full precision, but significant performance degradation occurs at binarized precision, and the network structure that mimics the hardware structure is clearly the The network structure mimicking the hardware structure is obviously the main reason for this result.

Finally, there are still many issues that need further research. Such as exploring the learning effect of neural networks on more general discrete logic or even Boolean functions, thus further lowering the threshold of logic implementation; the extension and modification of large-size LUT decomposition algorithms to other common logics; the extension of MLUTNet structure to convolutional neural networks, and whether MLUTNet based on convolutional neural networks is sufficient to overcome the problem of degradation of binarization accuracy due to its own specificity. Together, these issues constitute the goal of further research in the future.

References

[BM10] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BVM+19] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges. *ACM Journal on Emerging Technologies in Computing Systems*, 15(2):1–35, Jun 2019.

[CBIK+18] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep learning for classical japanese literature, 2018.

[CBM+20] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7), 2020.

[CGCB15] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. Gated feedback recurrent neural networks, 2015.

[CHS+16] 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, 2016.

[FLJ] A. Karpathy F.F. Li and J. Johnson. Stanford cs class cs231n: Convolutional neural networks for visual recognition.

[Fuk80] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GCK19] Roshan Gopalakrishnan, Yansong Chua, and Ashish Jith Sreejith Kumar. Hardware-friendly neural network architecture for neuromorphic computing, 2019.

[GDG+15] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation, 2015.

[GHS] Nitish Srivastava Geoffrey Hinton and Kevin Swersky. Neural networks for machine learning, lecture 6a, overview of mini-batch gradient descent.

[GMP+11] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. Impact: Imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 409–414, 2011.

[HCS+18] Winston Haaswijk, Edo Collins, Benoit Seguin, Mathias Soeken, Frédéric Kaplan, Sabine Süsstrunk, and Giovanni De Micheli. Deep learning for logic optimization algorithms. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2018.

[HHH+21] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, Xuefei Ning, Yuzhe Ma, Haoyu Yang, Bei Yu, Huazhong Yang, and Yu Wang. Machine learning for electronic design automation: A survey, 2021.

[HHSR19] Abdelrahman Hosny, Soheil Hashemi, Mohamed Shalan, and Sherief Reda. Drills: Deep reinforcement learning for logic synthesis, 2019.

[HLvdMW18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[HS06] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[HTR18] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. Blasys: Approximate logic synthesis using boolean matrix factorization. In *2018 55th*

ACM/ESDA/IEEE Design Automation Conference (DAC), pages 1–6, 2018.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KGE11] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th Internatioal Conference on VLSI Design*, pages 346–351, 2011.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[LAH+21] Jinglan Zhang Laith Alzubaidi, Amjad J. Humaidi, et al. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8(53), 2021.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBOM12] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient backprop*, pages 9–48. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag, 2012. Copyright: Copyright 2021 Elsevier B.V., All rights reserved.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[LH17] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts, 2017.

[NAT+19] Walter Lau Neto, Max Austin, Scott Temple, Luca Amaru, Xifan Tang, and Pierre-Emmanuel Gaillardon. Lsoracle: a logic synthesis framework driven by artificial intelligence: Invited paper. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2019.

[NH10] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[PNP19] Ghasem Pasandi, Shahin Nazarian, and Massoud Pedram. Approximate logic synthesis: A reinforcement learning-based technology mapping approach, 2019.

[PSY+18] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5), September 2018.

[QGL+20] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, 2020.

[Ros58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[RS20] Maithra Raghu and Eric Schmidt. A survey of deep learning for scientific discovery, 2020.

[SCYE17] 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.

[SG10] Doochul Shin and Sandeep K. Gupta. Approximate logic synthesis for error tolerant applications. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 957–960, 2010.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural

networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[SHY20] Jiong Si, Sarah Harris, and Evangelos Yfantis. Neural networks on an fpga and hardware-friendly activation functions. *Journal of Computer and Communications*, 08:251–277, 01 2020.

[SL19] Taylor Simons and Dah-Jye Lee. A review of binarized neural networks. *Electronics (Switzerland)*, 8(6), 2019.

[SM19] Ajay Shrestha and Ausif Mahmood. Review of deep learning algorithms and architectures. *IEEE Access*, 7:53040–53065, 2019.

[SSEM19] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. FPGA-Based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2019.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[STIM18] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Mądry. How does batch normalization help optimization? In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2488–2498, Red Hook, NY, USA, 2018. Curran Associates Inc.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[TGK+19] Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, Mar 2019.

[TL20] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[Tsa18] Sik-Ho Tsang. Review: Alexnet, caffenet — winner of ilsvrc 2012 (image classification), 2018.

[VSK+12] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. Salsa: Systematic logic synthesis of approximate circuits. In *DAC Design Automation Conference 2012*, pages 796–801, 2012.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

[WGDL21] Ran Wu, Xinmin Guo, Jian Du, and Junbao Li. Accelerating neural network inference on FPGA-based platforms—a survey. *Electronics (Switzerland)*, 10(9):1–25, 2021.

[WHT+17] Senling Wang, Yoshinobu Higami, Hiroshi Takahashi, Masayuki Sato, Mitsunori Katsu, and Shoichi Sekiguchi. Testing of interconnect defects in memory based reconfigurable logic device (mrld). In *2017 IEEE 26th Asian Test Symposium (ATS)*, pages 17–22, 2017.

[XGD+17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017.

[XRV17] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[YAL20] Huo Yingge, Imran Ali, and Kang-Yoon Lee. Deep neural networks on chip - a survey. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 589–592, 2020.

[YLC+16] Shimeng Yu, Zhiwei Li, Pai-Yu Chen, Huaqiang Wu, Bin Gao, Deli Wang, Wei Wu, and He Qian. Binary neural network with 16 mb rram macro chip for classification and online training. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 16.2.1–16.2.4, 2016.

[YN17] Haruyoshi Yonekawa and Hiroki 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, 2017.

[YXM19] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. Developing synthesis flows without human knowledge, 2019.

[ZSV15] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization, 2015.

[ZSZ+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 15–24, New York, NY, USA, 2017. Association for Computing Machinery.