Master's Thesis

A Study on Multi-Exit Deep Neural Network for Real-time Processing

1810209 LI NAQIN

Supervisor : Professor Kiyofumi Tanaka
Main Examiner : Professor Kiyofumi Tanaka
Examiners : Professor Mineo Kaneko
Professor Yasushi Inoguchi
Associate Professor Yuto Lim

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
[Information Science]

January, 2020

Abstract

The deep neural network has been wildly used in various fields in recent years, since it has excellent performance and is easy to use in classification and prediction tasks. It is also applied in real-time systems, such as self-driving and object tracking systems, etc. However, the problem of it is the huge amount of multiplication and accumulation operations make a neural network task become a heavy task in real-time systems. This may bring a huge impact on the performance of the system in the real-time property. Because of this, most of the real-time systems which include the neural network task must make the compromise to the accuracy of the result of the neural network task and the real-time requirement of the system.

BranchyNet [13] and MSDNet [2] were proposed in 2017. These two neural networks are different from the typical neural network model. They have more than one exit-point instead of only one exit-point. Different exit-points are inserted into different places in the neural network model. Since the insertion locations in the model are different from each other, the execution times from input-point to each exit-point are also different. The later exit-point will take more time to finish compared with earlier exit-points. Each of the exit-points in the multiple exit-point models gives the output of the classification result on one input picture.

In Chapter 2, we briefly introduce the CNN(Convolutional neural network), and how the feature map flows as the input and output data between each layer. For the BranchyNet, it was proposed for the fast inference via finishing its execution from earlier exit-points if the system already has enough confidence with the generated classification results. One of our proposals utilizes a BranchyNet model based on the VGG-16 for Cifar-10 dataset. This VGG-16 based BranchyNet has 3 exit-points. From the entry of the model to the first exit-point "EXIT1", it has 9 layers. In addition, it has 12 layers from entry-point to second exit-point "EXIT2", and 16 layers from entry-point to the third exit-point "EXIT3". The accuracy for "EXIT1" on Cifar-10 is 87.22%, for "EXIT2" is 88.51%, and for "EXIT3" is 88.57%. MSDNet, which was proposed for the similar purpose to the BranchyNet considers the impact of the accuracy of the final exit-point by inserting early exit-points into the model. The MSDNet for Cifar-10 has 24 layers, and an exit-point

is inserted after every 2 layers. The accuracy for "EXIT1" in MSDNet is 84.45%, for "EXIT2" is 86.55%, "EXIT3" is 87.95%, "EXIT4" is 88.93%, "EXIT5" is 90.11%, "EXIT6" is 90.17%, "EXIT7" is 90.28%, "EXIT8" is 90.55%, "EXIT9" is 90.6%, "EXIT10" is 90.61%, and "EXIT11" is 90.77%.

In Chapter 3, we regard the neural network task which is implemented with the multiple exit-points model in real-time systems as the imprecise computation [9] task. We show three scheduling methods for real-time systems that have one neural network task in their taskset. The first scheduling method is the typical real-time scheduling algorithm with the single exit-point neural network task. The second method "IC" includes the neural network task with a multiple exit-points model and regard this task as the imprecise computation task. In this scheduling, we decide the exit-point to finish the execution of the neural network based on the system load, instead of choosing by the confidence of classification results of each exit-point. If the time resource that the neural network task received from the system is long enough to exit from the later exit-points, then the system uses the classification result of the later exit-point. The later exit-point will give a result with higher accuracy. The third method "SIC" is a server-based imprecise computation scheduling method. It is proposed as an enhanced imprecise computation scheduling. With "SIC", the response time of each exit-point will be improved. Since "SIC" is a server-based method, we present a way to decide the server's priority in the systems, a way to compute the budget of the server, and when this server will be released to the system.

In Chapter 4, we show experiments to evaluate these 3 scheduling methods. We compare the accuracy of the classification result of the task among a single exit-point model, multiple exit-points model with "IC", and multiple exit-points model with "SIC". In addition, we compare the response time of each exit-point in "IC" and "SIC". In experiments, we implemented the multiple exit-points model for neural network task with the VGG-16 based BranchyNet and the MSDNet model for Cifar-10 dataset. The basic scheduling algorithms we used are EDF(earliest deadline first) and RM(rate-monotonic).

In Chapter 5, we implemented the binarized VGG-16 based BranchyNet on FPGA with VHDL. We show the structure of the neural network model and the way we are using it to implement the batch normalization operation. Then, we compare the execution result between the binarized BranchyNet and its software implementation, for checking the correctness of the hardware implementation. Furthermore, we show the accuracy and execution of it.

In Chapter 6, we draw a conclusion on our research. From our research, we can know the efficiency of applying the multiple exit-points neural network models and treating the neural network task as the imprecise computation task. It can improve the accuracy of the neural network task without making other tasks miss their deadline.

Acknowledgments

First of all, I would like to give my sincere gratitude to my supervisor, Professor Kiyofumi Tanaka. Professor Tanaka has been helping me through my master student life, and even before I become a master student. I cannot come to study in Japan if professor Tanaka didn't accept me as a master student. And in my master student life, Professor Tanaka has given me sufficient suggestion and guidance, whatever in research or train me consider and talking, writing as a scientific student.

My gratitude goes to JAIST for giving me the tuition fee reduction every time, so that I don't have too much pressure for saving the tuition fee.

I would like to thanks my lab-mate, who have been giving me advice, whenever I have problem about my life in Japan. With their help, I can live in Japan in a good way.

Last, I would like to thank to my parents, within this 3-years. They give me a lot of support, mentally and financially. I got a lot of encourage from them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In the past of few years, Neural Network has become the most popular and state-of-art in deep learning field, since it can achieve a high accuracy by extracting and learning features from data by itself. Because of this advantage, it is also applied on real-time systems, for example the self-driving system and object-tracking system.

In most of the condition, when the neural network task is integrated in the system, system developers desire to get a higher accuracy from the neural network task, and it requires relatively larger models and spends longer time to execute it. According to this, if the system utilization is already high enough, a neural network with a longer execution time may overload the system utilization, and cause one or more tasks to miss their deadline.

For handling the overload condition, imprecise computation [9] was proposed. In the imprecise computation model, it separates a task into mandatory phase and optional phase. The mandatory phase takes less execution time but gives a less acceptable result. The optional phase takes longer execution time than mandatory phase to finish , but a relatively higher precision(or accuracy) than the result from mandatory phase will be produced.

In 2017, BranchyNet [13] was proposed by Surat Teerapittayanon and MSDNet [2] was proposed by Gao Huang. These two neural network models all considered the resource-limited and energy-sensitive application, and they all give the solution for fast exit from the execution of neural network by designing several early exit-points in the model. This means several exit-points are

inserted in different places in the network model. Some of the exit-points are inserted in the early layers, so that they have a short path, and take shorter execution time to finish even though they give a lower accuracy.

In the neural network model like BranchyNet and MSDNet we mentioned above, the shortest path can be regarded as the mandatory phase and the other exit-points inserted in the later layers can be regarded as the optional phase in the imprecise computation.

Binarized Neural Networks were proposed [1]. Such neural network model uses two values (+1 and -1) to represent the values of weight parameters instead of using floating point values as a normal neural network model. By simplifying the weight parameters to +1 and -1, Binarized Neural Network will get a relatively lower accuracy than a normal floating point implementation, but decrease the utilization of memory. Since it uses only +1 and -1, if we use 1 to represent +1 and 0 to represent -1, then we can improve the execution time by using XNOR operation instead of floating point multiplication to implement the basic calculation of neural network.

For the speed and energy-efficiency of neural network, a Binarized Neural Network accelerator for BranchyNet model which runs on FPGA is considered.

1.2 Objectives

In this research, our target is to use a larger deep neural network model in real-time system even in overloaded situations, without making other tasks miss their deadline. The following things are the objectives.

- Train a BranchyNet and a MSDNet as a neural network task in the real-time system.
- Implement a real-time scheduling simulator and two scheduling algorithms which are Earliest Deadline First(EDF) and Rate Monotonic(RM).
- Run the designed neural network task in the scheduling simulator as an imprecise computation task and observe the accuracy when the utilization of system and utilization of neural network task are changed.
- Implement a Binarized version of BranchyNet on FPGA.

1.3 Outline

The rest of the thesis is organized as follow:

- Chapter 2 describes the basic knowledge of Convolutional Neural Network, the BranchyNet, and MSDNet.
- Chapter 3 proposes 3 scheduling strategies.
- Chapter 4 shows the scheduling result with the three different scheduling strategies.
- Chapter 5 describes the Binarized implementation of BranchyNet.
- Chapter 6 Concludes this research and presents future work.

Chapter 2

Related Work

2.1 Convolutional Neural Network

Convolutional Neural Network [8] is a class of feedforward neural networks. It is wildly used now in various fields including computer vision, audio processing, and natural language processing and etc because of its outstanding performance of prediction or classification accuracy. A typical convolutional neural network consists of more than one convolution layer which is applying convolution operation on input data for automatically extracting features. After one convolution layer, a pooling layer (Max pooling, Average pooling, or Global pooling) follows as an option. The pooling layer is for reducing the dimension and preventing the overfitting to the training data. A fully connected layer is commonly applied in the very last layer as a classifier to generate the output of the prediction or classification result. Figure 2.1 shows an image of the structure of the convolutional neural network.

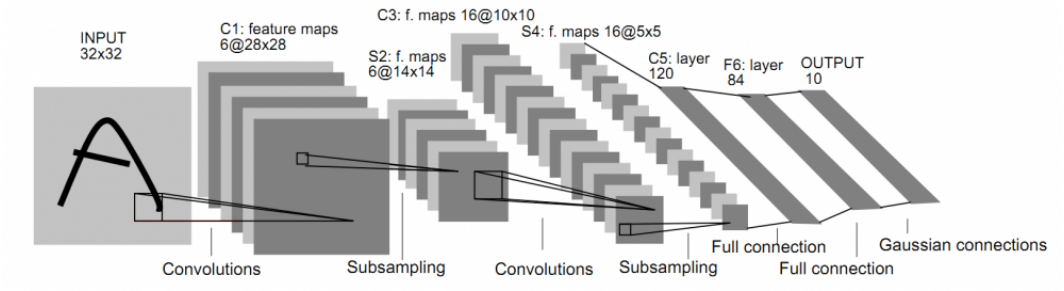


Figure 2.1: The structure of the Convolutional Neural Network [8].

In this figure, it includes 5 parts: input data, subsampling, convolutional operations, full connection layer, and output.

The input data is a raw picture. Input pictures can be a monochrome picture or a 3-channel RGB picture. And the first convolution layer will compute its output feature map from this input picture.

For the subsampling and convolutional operations, convolution layers with tanh activation function are for the convolution operations and pooling layers are for the subsampling.

Before the output data from the last convolution layer is input to the first full connection layer, it will be flattened, because as shown in the figure, whatever the input or output feature maps for convolution and subsampling layer are 2-dimensional data, but the full connection layer only accepts 1-dimensional input. After the last full connection layer, the classification result is produced.

2.2 BranchyNet

2.2.1 Introduction

Considering the high resource consumption for Convolutional neural network, it is still a problem to apply it on resource-restrictive and energy-sensitive application. The most straightforward solution for solving this problem is the fast inference which tries to finish the execution of the convolutional neural network in the early-stage instead of finishing processing layer by layer of the complete model. According to this, the idea of inserting some early exit-points to a complete Convolutional neural network model is designed. One of the simple designs of BranchyNet [13] which is based on AlexNet [6] is shown in Figure 2.2.

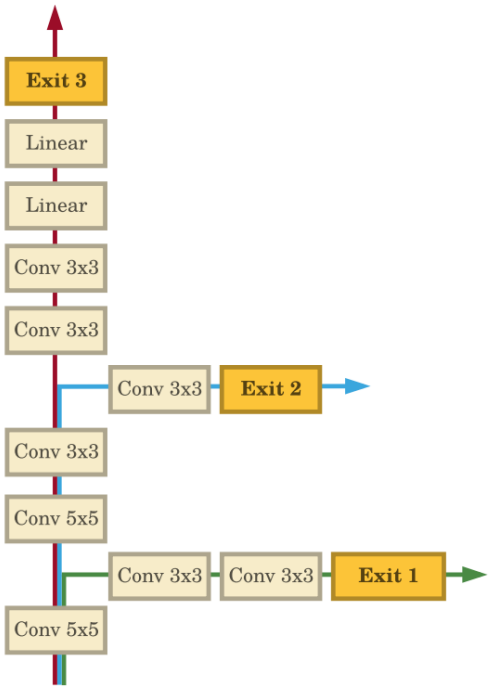


Figure 2.2: An BranchyNet implementation based on the AlexNet [13].

The origin AlexNet has 8 layers, 5 of them are implemented with a convolution layer, and the rest of the 3 layers are fully connected layers. In the figure above, the original AlexNet structure is from the convolution layer at the bottom as the entry to the "EXIT3" as the exit, and we also call it the main branch. Two side branches are inserted on the main branch. One is inserted after the first convolution layer. Its execution path is from the first convolution layer until the "EXIT1". The second one is inserted after the third convolution layer. It is from the first convolution layer until the "EXIT2".

In this network structure, the classifier of "EXIT3" is a full connection layer. For "EXIT2" and "EXIT1", each of them consists of 1 max-pooling layer and 1 full connection layer as the final layer.

BranchyNet is proposed for fast inference, which means if the condition is met, It tries to exit from the execution of BranchyNet via early exit-point as early as possible. According to this, a procedure or mechanism for making the decision to exit from early exit-point is necessary. In their paper, a "confidence" is defined for doing such duty. The "confidence" is calculated by entropy. The following formula calculates the entropy:

$$entropy(\boldsymbol{y}) = \sum_{c \epsilon \boldsymbol{C}} y_c log y_c,$$

where $\boldsymbol{y}$ is the vector of classification result from an exit-point which is a set of values, containing the possibility that input data belongs to the corresponding class, and $\boldsymbol{C}$ is a set of possible classes.

A procedure named BRANCHYNETFASTINFERENCE is defined as Algorithm 1.

Algorithm 1: BRANCHYNETFASTINFERENCE

```
for n = 1, 2, ... N do
    z = f_exit_n(x)
    ŷ = softmax(z)
    e ← entropy(ŷ)
    if e < T_n then
        return MaxIndex(ŷ)
end
return MaxIndex(ŷ)
```

In the procedure above, $\boldsymbol{x}$ is one input data. Assume there are N exit-points in the neural network model. In each iteration, one classification result from exit-point $exit_n$ is marked with $\boldsymbol{z}$. Then it calls the function of softmax and entropy until we get a value e. In the final step of this for-loop, a comparison is performed: if the entropy of the current classification result is less than T_n. The definition of T_n is the threshold for the n-th exit-point. It is checked whether the entropy of a prediction result from an exit-point is less than its threshold. For example, if the result of i-th exit-point is less than T_i, then they think this prediction result is confident enough to finish the execution, so exit the execution of this neural network model from this exit-point. The details of calculating T_n is also showed in the repository of BranchyNet [7].

2.2.2 VGG-16 Based BranchyNet

In our research, we modified one BranchyNet based on VGG-16 [12] and the dataset for training and testing is Cifar-10 [5]. The basic VGG-16 consists of 13 convolution layers and 3 fully connected layers. The details for its structure are shown in the "C" column in Figure 2.3.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (32×32 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.3: Baseline of VGG-16 in the C column [12].

After the modification, the structure of the BranchyNet based on VGG-16 is showed in Figure 2.4.

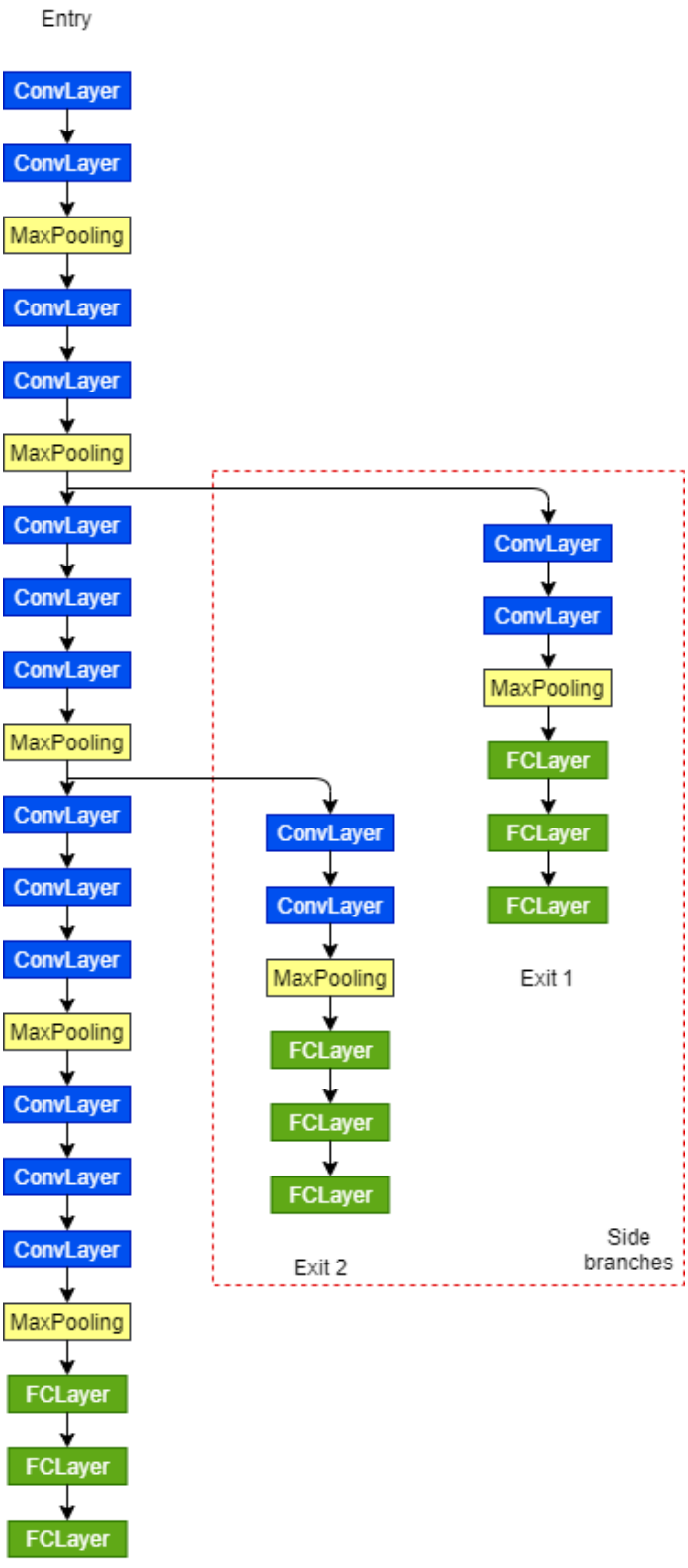


Figure 2.4: VGG-16 based BranchyNet [12].

In the VGG-16 based BranchyNet, 2 side-branches are inserted. Branch 1 is from "Entry" to the "Exit 1". Similarly, branch 2 is from "Entry" to the "Exit2". Each of these two side branches consists of 2 convolution layers, 1 max-pooling layer, and 3 full connection layers.

Accuracy for the evaluation data in Cifar-10 dataset for all of the exit-points in this model is showed in Figure 2.5. According to the evaluation result, the accuracy of the later exit-point is relatively higher than the earlier exit-point.

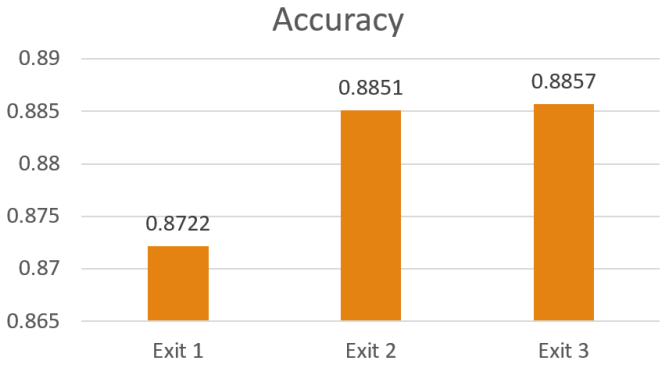


Figure 2.5: Accuracy of VGG-16 based BranchyNet [12].

2.2.3 Training BranchyNet

In this subsection, we show the training procedure of BranchyNet. Assume the input is x, and a function f_{exit_n} generates the output from the n-th exit-point, and y is the classification result.

$$y = f_{exit_n}(x). \tag{2.1}$$

Then apply the softmax operation on y to get $\hat{y}$. C in this formula is the set of all the possible labels. The Formula2.2 is

$$\hat{y} = softmax(y) = \frac{exp(y)}{\sum_{c \in C} exp(y_c)}. \tag{2.2}$$

When the one-hot ground-truth label vector is z, the loss function to calculate the error of this classification result y is as Formula2.3

$$L(\hat{y}, z) = -\frac{1}{|C|} \sum_{c \in C} z_c log \hat{y}_c. \tag{2.3}$$

The formula to calculate the error of BranchyNet is Formula2.4

$$L_{BranchyNet}(\hat{y}, z) = \sum_{n=1}^{N} w_n L(\hat{y}_{exit_n}, z), \tag{2.4}$$

where N is the total number of exit-points, and w_n represents the weight of loss value of n-th exit-point in the total loss value. The sum of weight values of all exit-points should be 1.0.

The gradient of each weight parameter in the BranchyNet is calculated based on loss value $L_{BranchyNet}(\hat{y}, z)$. The optimizer we are using is Adam.

2.3 MSDNet

2.3.1 Introduction

The full name of MSDNet [2] is Multi-Scale Dense Networks. The MSDNet is proposed for the situation of limited computation resource in the platform, which is quite similar to the BranchyNet. But the main difference between MSDNet and BranchyNet is that MSDNet considers the effect of insertion of early exit-point to the accuracy of the final classifier. The accuracy of the final exit-point is reduced because of the inserted exit-point. This happens because the early exit-point makes the features generated by early convolution layers optimized for the short-term, but for long-terms, the final classifier requires some "high-quality" features which might have no high contribution to the early exit-point but are good for the latest exit-point, and this effect is more obvious when the early-exit point is appended to the earlier exit-point.

For the reason above, the MSDNet is implemented based on the DenseNet [3]. The DenseNet connects all of the feature maps generated by previous layers and uses them as the input for the next layer, so in the later layers, it will receive the feature map with "high-quality" features and the feature maps which are good for the early exit-point. The connection is shown in Figure 2.6.

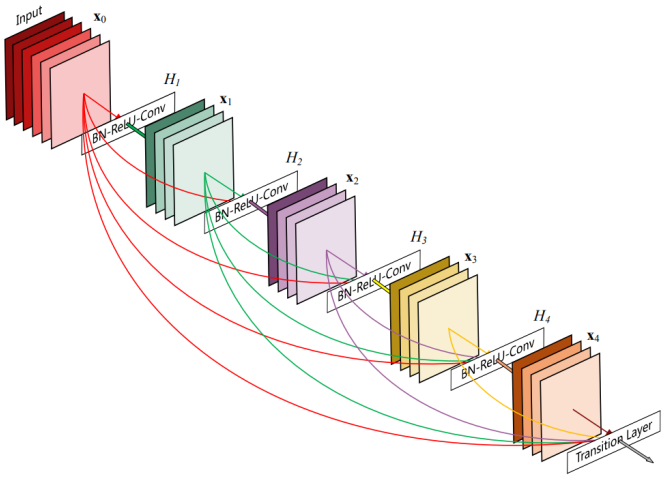


Figure 2.6: Example of structure for DenseNet.

In this figure, x_i is the feature maps generated by H_i,

$$x_i = H_i([x_0, x_1, \ldots, x_{i-1}]).$$

When $i = 0$, then:

$$x_0 = input\ data.$$

When $i = 1$, then:

$$x_1 = H_1(x_0).$$

The structure of MSDNet has multiple MSDNetlayer, each MSDNetlayer consists of multiple scales of feature maps as in Figure 2.7. In this figure, the horizontal axis is the depth of the network, and the vertical is the scale of the network of each layer. For the first MSDNetlayer, the feature map of layer 1 and scale 1 is generated first, then the feature maps of scale 2 are generated based on scale 1, and scale 3 is calculated based on scale 2 in layer 1 with the stride convolution operation. The stride convolution operation means the stride for this convolution operation is 2, and relatively there is a regular convolution operation defined with a stride of 1.

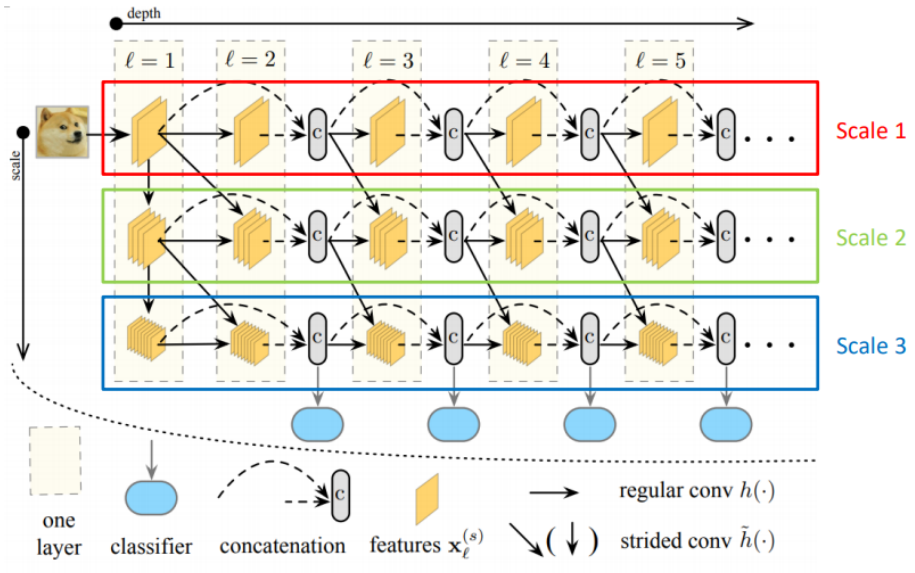


Figure 2.7: Example of structure for MSDNetlayer.

The input data for the first MSDNetlayer is a picture. For the later layers, the input of the convolution layer $H_{i,j}$ for getting feature map $x_{i,j}$ in layer i with scale j is the concatenation result of the feature maps from previous layers with the same scale, which means $[x_{0,j}, x_{1,j}, \ldots, x_{i-1,j}, x_{i-1,j-1}]$. So the formula is

$$x_{i,j} = H_{i,j}([x_{0,j}, x_{1,j}, \ldots, x_{i-1,j}, x_{i-1,j-1}]).$$

The exception is the first scale in each layer. The input feature map in each layer in scale 1 is

$$x_{i,1} = H_{i,1}([x_{0,1}, x_{1,1}, \ldots, x_{i-1,j}]).$$

The concatenation operation is rarely found in other neural network structure. An example to simply explain the concatenation operation is as follows. $\boldsymbol{a}$ is defined as $\boldsymbol{a} = [3.0, 4.0, 5.0, 6.0]$ and $\boldsymbol{b}$ is defiend as $\boldsymbol{b} = [1.0, 2.0, 7.0]$, then after the concatenation we get a set $\boldsymbol{c} = [\boldsymbol{a}, \boldsymbol{b}] = [3.0, 4.0, 5.0, 6.0, 1.0, 2.0, 7.0]$.

2.3.2 MSDNet for Cifar-10

The structure of MSDNet for Cifar-10 dataset has 24 MSDNetlayers and 11 classifiers. Layers 1 to 8 have 3 scales in each layer, layers 9 to 16 have 2 scales

and after layer 16, there is only one scale in each layer. 11 classifiers are inserted in the model; the first classifier is inserted after the fourth MSDNetlayer. After the fifth layer, 1 classifier is inserted after every 2 layers. Each classifier consists of 2 convolution layers and 1 fully connected layer.

Feature map size is 32x32 in scale 1, 16x16 in scale 2, and 8x8 in scale 3.

The structure is shown in Figure 2.8.

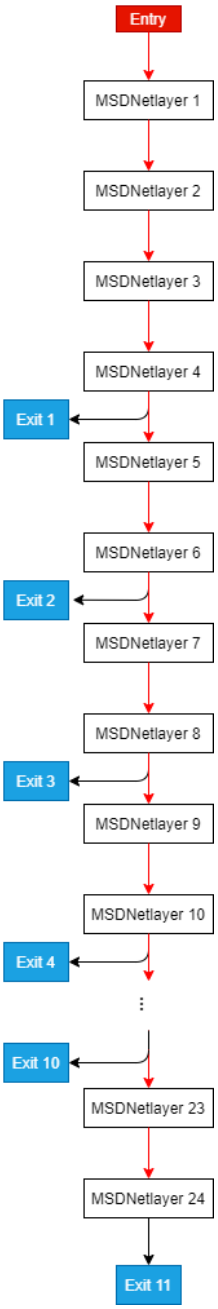


Figure 2.8: Structure for MSDNetlayer on Cifar-10.

The evaluation accuracy for MSDNet on Cifar-10 is showed in Figure 2.9.

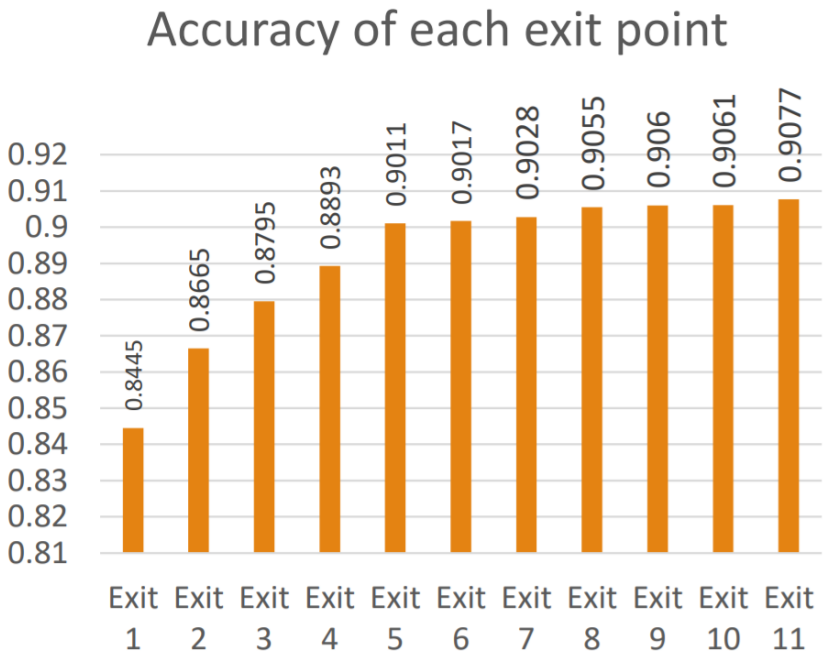


Figure 2.9: Structure for MSDNetlayer on Cifar-10.

2.3.3 Training MSDNet

For training the MSDNet model, when we use f_k to represent the k-th classifier in the model, the loss function on this classifier is denoted as $L(f_k)$. We are using the cross-entropy as the loss function. The optimization target is to minimize the loss value $L_{MSDNet}(x, y)$ calculated with Formula 2.5.

$$L_{MSDNet}(x, D) = \frac{1}{|D|} \sum_{(x,z)\in D} \sum_{k} w_k L(f_k), \tag{2.5}$$

where D is the training set, since we are using the batch training method. For each time, multiple data will be input and computed by the neural network. w_k is the weight for k-th exit-point.

Chapter 3

Scheduling Strategy

In this chapter, I show 3 scheduling strategies for scheduling the neural network task in the system. The first strategy is scheduling with a single exit-point neural network task model. The second one is a scheduling strategy that schedules the neural network model with multiple exit-point as the imprecise computation task. I call it "IC" as the simplified name. In the third strategy we use a server-based mechanism to schedule the neural network task with multiple exit-point as the imprecise computation task. We simply call it "SIC".

3.1 Basic concept for real-time scheduling

Before I start to show these 3 scheduling methods, some basic concepts about real-time scheduling will be introduced. For each real-time system, it has some computation resource, and a taskset consists of some tasks for utilizing the computation resource. In our research, we assume the computation resource is a single-core CPU. All of the task τ_i in this taskset τ, including neural network task, are periodic task and each of τ_i has its own worst-case execution time $WCET_i$, the relative deadline D_i and the period T_i. Only one neural network task is executing in the system.

The worst-case execution time, $WCET_i$ is the longest execution time taken for finishing one job instance of a task. But in some cases, a job instance of a task will go to execute a branch which requires less execution time than the branch with the longest execution time, according to the system condition or input data of that moment. One thing that should be noticed is that the system designer cannot predict the exact execution time spent by this task

because it is decided dynamically.

The relative deadline D_i means the longest time interval for finishing the job of task τ_i should be less than or equal to D_i. For example, assume a job instance j_i of task τ_i is released at time t then the j_i should be finished before time $t + D_i$. If j_i is finished later than $t + D_i$, then the calculation result of j_i is meaningless. We call this the deadline miss.

The period T_i means for each system time T_i passed, one job instance of task τ_i will be released into system. The task τ_i's worst case execution time $WCET_i$ should be less than or equal to its relative deadline D_i and period $T_i (WCET_i \leq D_i \leq T_i)$. In our problem, we consider the period and relative deadline are equal$(D_i = T_i)$.

Besides these 3 basic factors, utilization U_i for task τ_i is calculated by the formula

$$U_{\tau_i} = \frac{WCET_i}{T_i}.$$

For each task, its utilization should be less than or equal to 1.0. Otherwise, it means its worst-case execution time $WCET_i$ is longer than the period T_i , and a new job instance of τ_i will be released before the old one is finished even if processor is only running this task. No feasible execution strategy exists for this task on this processor.

Besides the utilization defined for individual task, a total utilization of U is defined as

$$U = \sum_{\tau \in \tau} U_\tau,$$

where τ is the taskset. From this formula, we can know the definition of total utilization of U of the system is the summation of all of the tasks in the taskset. If a taskset with a total utilization U larger than the upper-bound of the scheduling algorithm [1], then there will be no scheduling strategy for this task set without causing any deadline miss.

[1]The upper-bound of schedulability depends on the scheduling algorithm. For example, the upper-bound is 1.0 for earliest deadline first, while 0.69 for rate-monotonic.

3.2 Imprecise computation

Imprecise computation [9] was proposed for handling the overload condition in a real-time computation system, so it is focusing on solving the timing faults in a real-time system instead of the quality of the computation result from a real-time task. In the imprecise computation, a task is separated into the mandatory phase and optional phase. The mandatory phase will give the lowest accuracy or precision of the computation result but utilize the least execution time compared with finishing the task until the optional phase. When the task finished the optional phase, it will get a relatively higher precision result. And according to the system conditions, the precision of computation result can be determined at different levels, so there can be two or more optional phases for the task in an imprecise computation model.

3.3 Example of task set

For helping me to explain these 3 scheduling methods, I will use a taskset example shown in Table 3.1. In this taskset, there are normal task τ_1, τ_2 and τ_3, and a neural network task named τ_{NN}. In the first scheduling method, τ_{NN} is a task with a single exit-point neural network model, and the worst-case execution time is 2 system time units. But in the "IC" and "SIC", τ_{NN} is a neural network task which is running with a multiple exit-point(to be specific, 2 exit-points in the example). In this condition, since we need to guarantee the shortest path of the neural network should be finished, we define worst-case execution time $WCET_{NN}$ for it of 2, and there is a later exit-point as the optional one which requires one more system time to exit from it.

	WCET	T	U
τ_1	2	4	$\frac{1}{2}$
τ_2	2	8	$\frac{1}{4}$
τ_3	2	16	$\frac{1}{8}$
τ_{NN}	2	16	$\frac{1}{8}$

Table 3.1: Table of taskset.

When the system is running with the task set above, the actual execution time $AET_{i,j}$ for j-th job instance of task τ_i is generated as

$AET_{1,1} = 1 \quad AET_{1,2} = 2 \quad AET_{1,3} = 1 \quad AET_{1,4} = 2$
$AET_{2,1} = 1 \quad AET_{2,2} = 2$
$AET_{3,1} = 2$
$AET_{NN,1} = 2$

Actually, for the task τ_{NN}, its $AET_{NN,j}$ should be $WCET_{NN}$ all the time, because as we know, once the structure of the neural network is fixed, its execution time should be a constant.

3.4 Scheduling with single exit-point model

The first scheduling strategy is the schedule with a single exit-point neural network model. When the system designers considered using the single exit-pint model in the real-time system, they must be aware of the total system utilization, which is a hard requirement for the taskset running on the system. The largest model that can be used in the system must guarantee that when the system is running, no task can miss its deadline(the total utilization of system load is less than the upper-bound of the applied scheduling algorithm), so the procedure for choosing the model is compromised with the worst-case execution time calculated as:

$$UPPER - BOUND\_WCET = (U - U_{oth})T_{NN},$$

where the U_{oth} is the total utilization of the tasks in taskset except the τ_{NN}, and the period T_{NN} should satisfy the required FPS(frame per second, some of the self-driving systems require up to the 63 FPS), $T_{nn} = \frac{1sec}{FPS}$. Any of the models which requires longer execution time than the calculated $UPPER - BOUND\_WCET$ cannot be applied. Other tasks in taskset will not always run for their worst-case execution time. So the CPU might be free at some times because of this.

A scehduling exmaple is shown in Figure 3.1. In this example, the scheduling algorithm is the EDF, and the utilization of this taskset is the upper-bound of EDF. But, there is some empty slot appear at 11-12 and 14-16, which happens since the actual executed time $AET_{1,1}$, $AET_{1,3}$ and $AET_{2,1}$ are not equal to their worst-case execution time.

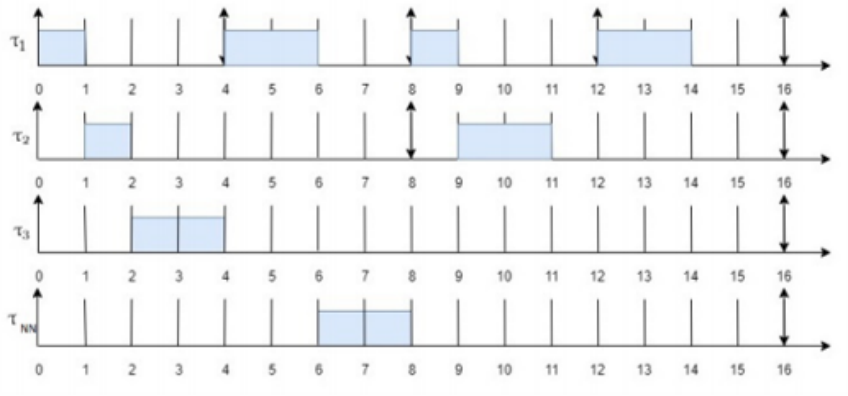


Figure 3.1: Schedule example with single exit-point model.

3.5 Scheduling with multiple exit-point model

3.5.1 Neural network model changed to multiple exit-point model

With the scheduling example in the previous section, we noticed even the calculated theoretical total utilization for the taskset is the upper-bound of the EDF 1.0. There are some empty slots not utilized by any task. If those empty slots can be utilized by neural network task, we can use a larger model to get a higher accuracy for classification. But as we mentioned before, the empty slot happens dynamically and cannot be predicted by the developer, so it becomes a problem.

In the second scheduling strategy "IC", we apply the neural network model with multiple exit-points. When we are using the multiple exit-point models, since each exit point is inserted into a different place in the model, finishing the execution of the neural network from different exit-point requires different execution times. If it exits from the later exit-point, it will spend a longer time than the earlier exit-point but the later exit-point gives higher accuracy. We think this behavior is similar to the concept of imprecise computation in the real-time computation field if we regard the shortest path exit from the earliest exit-point as the mandatory phase, for example the "EXIT1" in VGG-16 based BranchyNet model, and the later exit-point "EXIT2" as optional phase 1 and "EXIT3" as optional phase 2.

After we applied the imprecise computation model for the neural network

model, the worst-case execution time for the neural network task has become the execution time required by the mandatory phase. The reason for making this design is when the developer is doing the off-line analysis on taskset, he or she has to make sure that all of the tasks will finish their execution before their deadline, and neural network task has to finish its mandatory phase to give an acceptable result. The optional phase is regarded as a kind of "reward" for the system condition with enough empty slots, where those empty slots are got from the early finish of other tasks.

3.5.2 Scheduling for imprecise computation

In the scheduling, all of the tasks will be scheduled based on their priority in the system including the neural network task. The neural network task will be scheduled with only two conditions:

1. It is the task which has the highest priority in the system at that time.
2. There is some empty slots which are not utilized by any task and the neural network task did not miss its deadline, and did not finish until its last optional phase.

3.5.3 Example of IC-EDF

Assume that in the previous task example, the neural network task is implemented with the model which has 2 exit-points like shown in Figure 3.2. This model is based on the model we ever used in the example of single exit-point and now we call the original exit-point as "EXIT1", while a long branch with "EXIT2" is inserted into it for higher accuracy, which will ask for more execution time. After this operation, the worst-case execution time is the execution time for exiting from "EXIT1", that is 2, while the additional execution time for finishing the inserted longer branch up to "EXIT2" is 1.

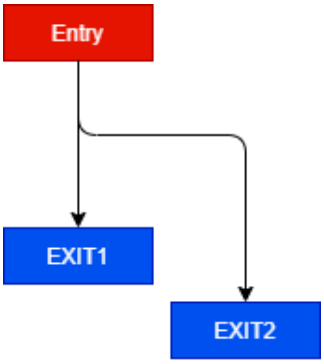


Figure 3.2: Image of model for IC-EDF

The scheduling result will be like the Figure 3.3.

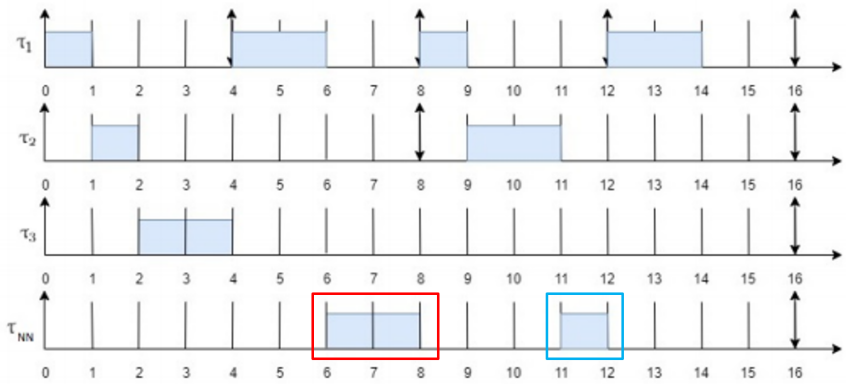


Figure 3.3: Image of model for IC-EDF

In the scheduling result, τ_{NN} is chosen to run based on its priority, and after that execution, τ_{NN} finishes its mandatory phase(marked with red frame) and gets a relatively rough classification result. At system time 11 to 12, the τ_{NN} is scheduled again because it is the second condition, and this time the τ_{NN} finishes its optional phase(marked with blue frame) and gets a relatively higher accuracy than the result from "EXIT1".

3.6 Improved scheduling strategy with multiple exit-point model

In the previous section, all of the tasks are scheduled based on their priority, and some of the tasks did not spend all of their determined worst-case execution

time. The reason why we can let the neural network task finish until the later exit-point to get a high accuracy is some of the job instances of the task do not take all of its worst-case execution time. For example, $j_{1,1}$ is the first job instance of τ_1, but it only spends 1 system time unit although system preserved 2 system time unit for it, and the $j_{2,1}$ can be executed earlier and an empty slot is got later. The empty slot happens because of the early finish of $j_{1,1}$. If we can run the τ_{NN} once any of the job instances finished early, an earlier response time of a job instance of τ_{NN} will be achieved.

3.6.1 Server for imprecise computation

For making this possible, we designed a server mechanism that is scheduled in the system background. To define a server mechanism, we need to consider the following questions:

1. when this server is released,
2. the priority for this server,
3. the budget for this server, and
4. what this server will do.

For question 1, the server is released at system time t, when a job instance $j_{i,j}$ finishes its running and the actual executed time $AET_{i,j} < WCET_i$.

For question 2, once the server appears, it will have the highest priority in the system. Before it consumes all of its budgets, no job can be scheduled.

For question 3, the budget for the server is calculated dynamically by the system. Developers can not predict how much the server can run in the off-line analysis. If the budget for the server is exhausted, the task with the highest priority is scheduled. The budget will be updated when a job instance is finished earlier. It is calculated as below.

$$BDG = WCET_i - AET_{i,j},$$

where the BDG is the budget for the server, and $AET_{i,j}$ is the actual execution time for $j_{i,j}$ which is finished at that time.

For question 4, whenever the server is released to the system, it will run the τ_{NN} if it did not miss its deadline and did not finish until the last exit-point.

If τ_{NN} finished until the last exit-point but the server still has some budget, it releases the computation resource.

3.6.2 Scheduling condition for neural network task

With this server-based mechanism, the neural network task will be scheduled to run in 2 conditions.

1. The neural network task is the task with highest priority at that time.
2. A server is released and neural network task has not finished until the last exit-point.

3.6.3 Example of SIC-EDF

The scheduling result for "SIC-EDF" is shown in Figure 3.4. The basic scheduling algorithm is the EDF, and the server mechanism is implemented based on this algorithm. Taskset example is the same as the task example in the previous sub-section.

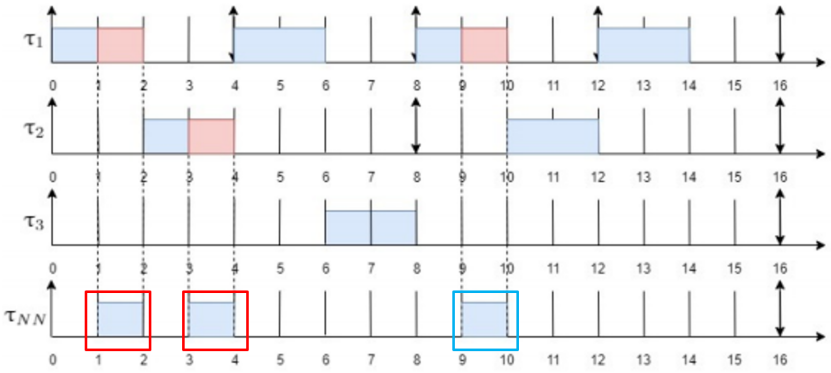


Figure 3.4: Image of model for SIC-EDF.

At $t = 0$, all of the job of tasks are released. At $t = 1$, $j_{1,1}$ is finished and the $AET_{1,1} = 1$. Since $AET_{1,1} < WCET_1$, so the server is released into system, and BDG at $t = 1$ is 1. Since server has the highest priority once released into the system, so it will utilize the processor and run the neural network task now. At $t = 2$, the budget for server is become 0, so it releases the processor to other tasks. At $t = 3$, $j_{2,1}$ is finished and $AET_{2,1} < WCET_2$, so server is

released and $BDG = WCET_2 - AET_{2,1} = 1$. The server will run the τ_{NN} until $t = 4$.

From the figure, we can know the τ_{NN}'s mandatory phase(marked with red frame) is executed in time slot 1-2 and 3-4 and finished at $t = 4$. Compared with the IC-EDF which is finished at $t = 8$, the response time of "EXIT1" for SIC-EDF is earlier than for IC-EDF. The response time of "EXIT2" for SIC-EDF is also earlier than the response time of "EXIT2" for IC-EDF.

Chapter 4

Evaluation

In this chapter, we show the experiment result of applying the single exit-point and the multiple exit-point with "IC" and "SIC". The VGG-16 based BranchyNet and MSDNet for Cifar-10 introduced in chapter 2 are used as the multiple exit-points model in the experiment. Accuracy of neural network task is the factor for comparing the single exit-point model and the multiple exit-points model in different scheduling strategy "IC" and "SIC". And another comparison is performed between the "IC" and "SIC". The response time is the factor when we are comparing these two strategies.

4.1 Experimental setup

4.1.1 Taskset profile

Tasksets for experiments are generated randomly. Each taskset has two parameters, total utilization U and the utilization for neural network task U_{NN}. For the generated taskset τ, it is

$$\tau = \tau_{oth} \cup \tau_{NN},$$

where τ_{oth} are tasks other than neural network task in the taskset.

When we calculate the total utilization, we used formula:

$$U = \sum_{\tau \in \tau} U_{\tau},$$

as shown in chapter 3.

For each combination of U and U_{NN}, there are 10 tasksets. The range for U and U_{NN} for taskset is decided as follows:

1. when the scheduling algorithm is the EDF, then U is from 0.7 to 1.0
2. when the scheduling algorithm is the RM, then U is from 0.5 to 0.7
3. the U_{NN} is from 0.1 to $U - 0.1$

Since we used the specific neural network model, the execution times of all phases for τ_{NN} are fixed, and they are collected by using the real execution time unit on CPU(Intel(R) Xeon(R) Silver 4116).

The execution times from entry-point to each exit-point for VGG-16 based BranchyNet are shown in Table 4.1.

exit-point	time (millisecond)
1	11
2	22
3	34

Table 4.1: Execution time required by each exit-point for BranchyNet.

The execution times from entry-point to each exit-point for MSDNet for Cifar-10 are shown in Table 4.2.

exit-point	time(millisecond)
1	27
2	52
3	79
4	114
5	133
6	149
7	164
8	179
9	186
10	193
11	201

Table 4.2: Execution time required by each exit-point for MSDNet.

4.1.2 Scheduling simulator

For the scheduling simulator, the basic scheduling algorithm we used is Rate-Monotonic or the EDF,

The actual execution time for each job is generated on-line by the scheduler simulator with the formula:

$$AET_{i,j} = RAND(\frac{WCET_i}{2}, WCET_i),$$

where the $RAND(a, b)$ is generating the random integer in range a to b.

4.2 Result for VGG-16 Based BranchyNet

4.2.1 Based on EDF

The experiment results for the VGG-16 based Branchynet with the EDF are in Figure 4.1, which shows the accuracy of the neural network task in different scheduling methods.

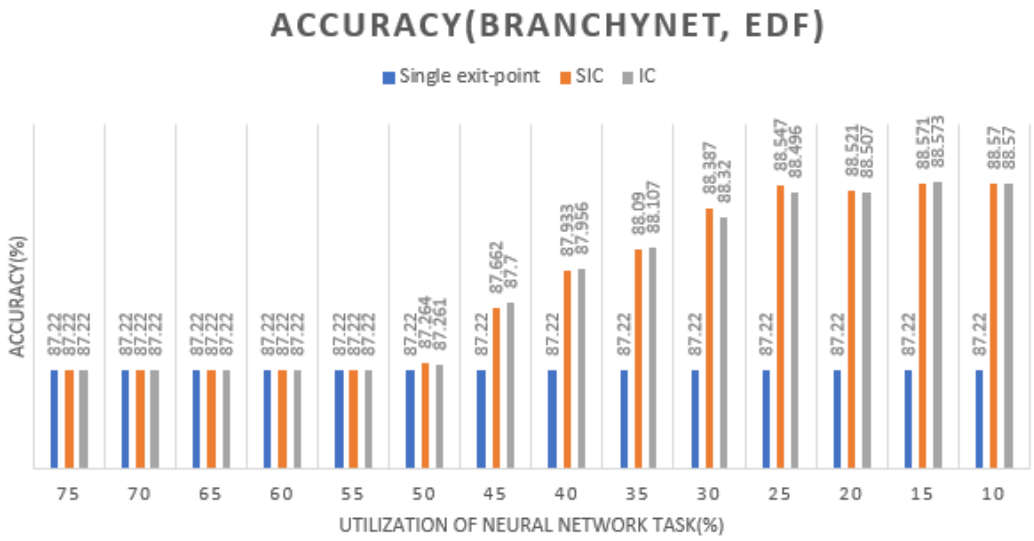


Figure 4.1: Accuracy of τ_{NN} for single exit-point, "IC" and "SIC", with the EDF.

The result we showed is when the U is 85%. The x-axis is the utilization of neural network task. The y-axis is the accuracy of the classification result from the neural network task.

According to Figure 4.1, we can conclude that the accuracy of the single exit-point model is constant. And for the "IC" and "SIC", if the U is fixed and U_{NN} is decreased, then the accuracy of the classification result of τ_{NN} is increased. This is because in the "IC" and "SIC", the execution time from entry to each exit-point is a constant. And less utilization for τ_{NN} means a longer period for it, and therefore more chance to get the empty slot happens due to the other tasks which finished early.

The next shown in Figure 4.2 is about the response time of each exit-point in "IC" and "SIC". The results in Figure 4.2 are when $U = 85\%$ and $U_{NN} = 10\%$. According to this figure, the response time of each exit-point in "SIC" is shorter than in the "IC".

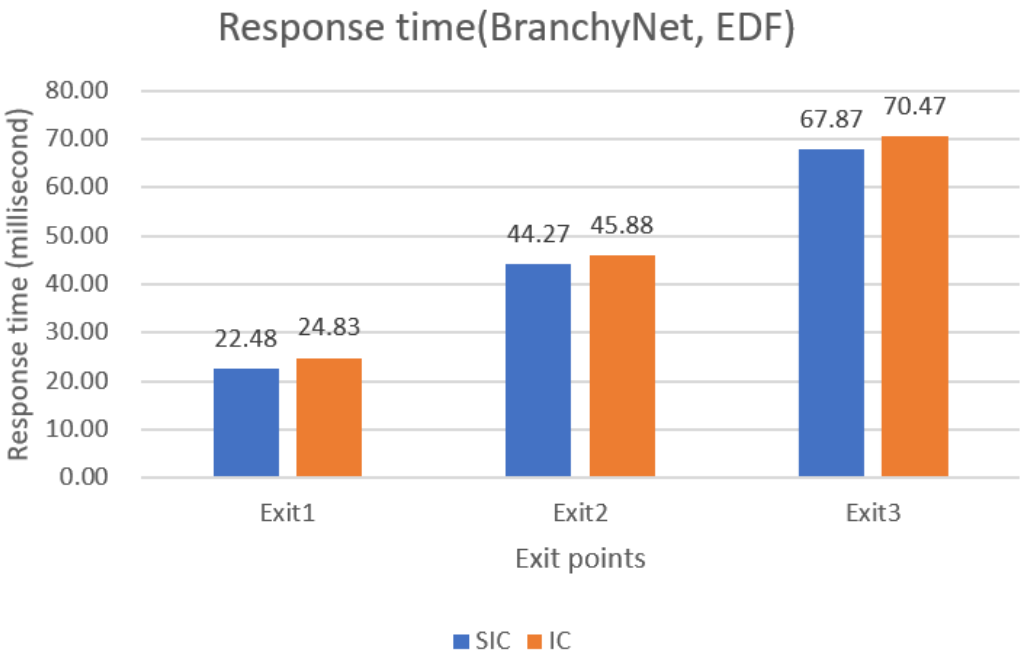


Figure 4.2: Comparison of response time between "IC" and "SIC", with the EDF.

4.2.2 Based on RM

An experiment similar to the previous subsection was conducted where the basic scheduling algorithm is RM. The result in Figure 4.3 is generated when $U = 65\%$.

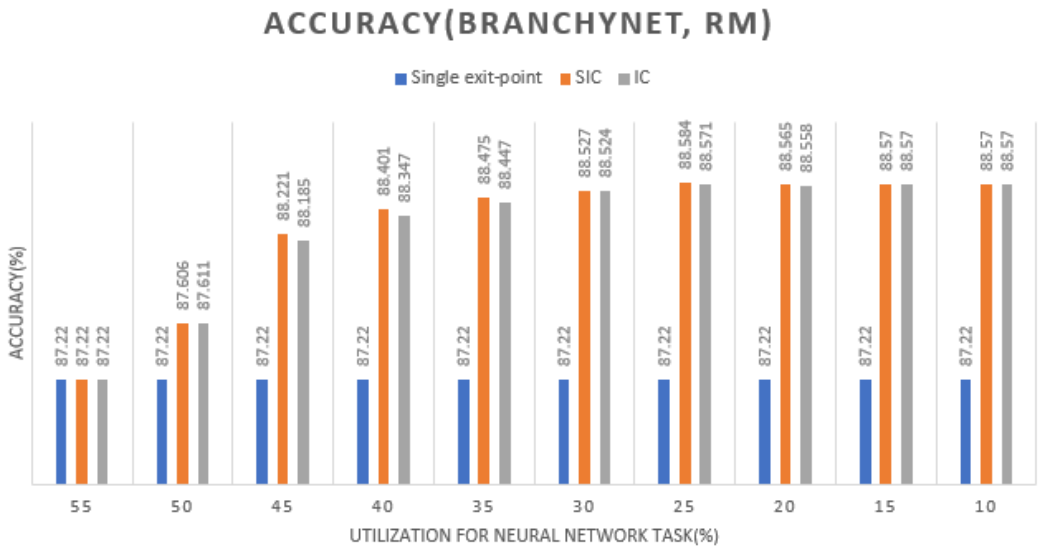


Figure 4.3: Accuracy of τ_{NN} for single exit-point, "IC" and "SIC", with the RM.

Figure 4.4 is the comparison of the response times of each exit-point when $U = 65\%$ and $U_{NN} = 10\%$. From this figure, we can get the same conclusion as when we use the EDF as the basic scheduling algorithm, that is the response time of each exit-point in "SIC" is shorter than in the "IC".

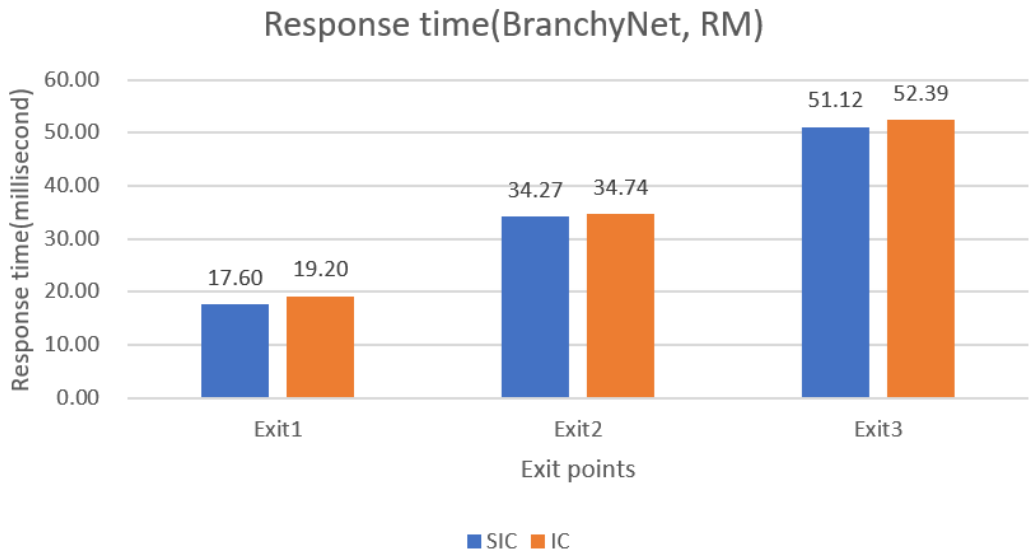


Figure 4.4: Comparison of response time between "IC" and "SIC", with the RM.

4.3 Result for MSDNet for Cifar-10

4.3.1 Based on EDF

The accuracy of τ_{NN} which is implemented with MSDNet where the basic scheduling algorithm is the EDF is shown in Figure 4.5. In this experiment, $U = 85\%$.

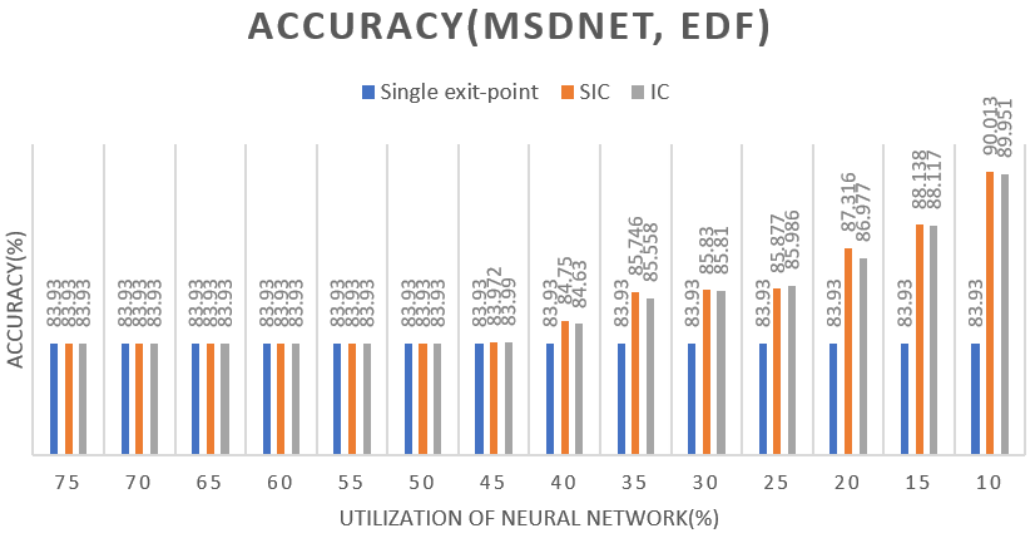


Figure 4.5: Comparison of accuracy, with the EDF.

In Figure 4.5, the x-axis is the U_{NN} and the y-axis is the accuracy of the classification result.

The comparison of response time is shown in Figure 4.6. This result is when $U = 70\%$ and $U_{NN} = 10\%$. Response time is 0 on some exit-points, which means no exit from that exit-point in scheduling. From this figure, we can see the response time of each exit-point in "SIC" is shorter than in "IC".

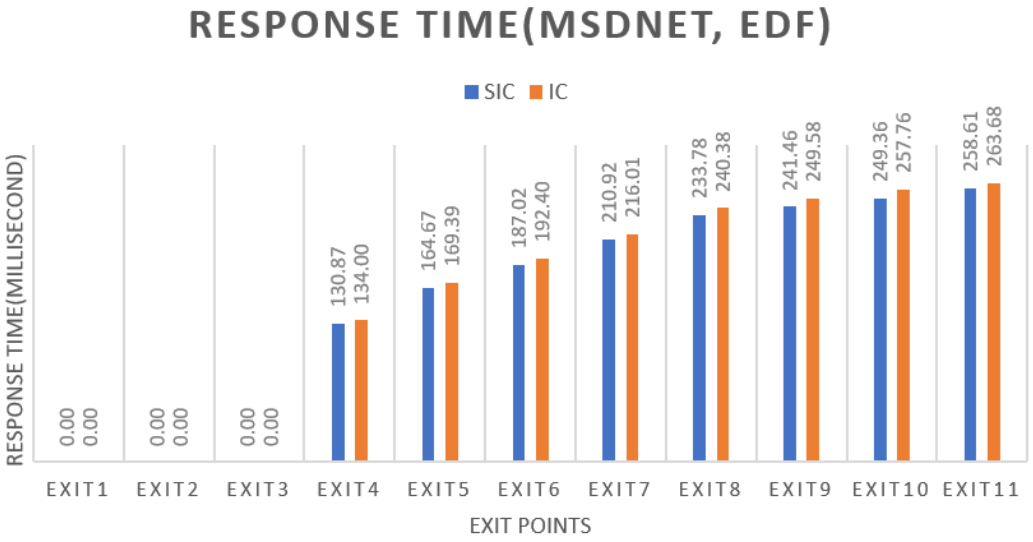


Figure 4.6: Comparison of response time, with the EDF.

4.3.2 Based on RM

When the basic scheduling algorithm is RM, we got a similar result which is shown in Figure 4.7. The U is 65% and U_{NN} is from 55% until 10%.

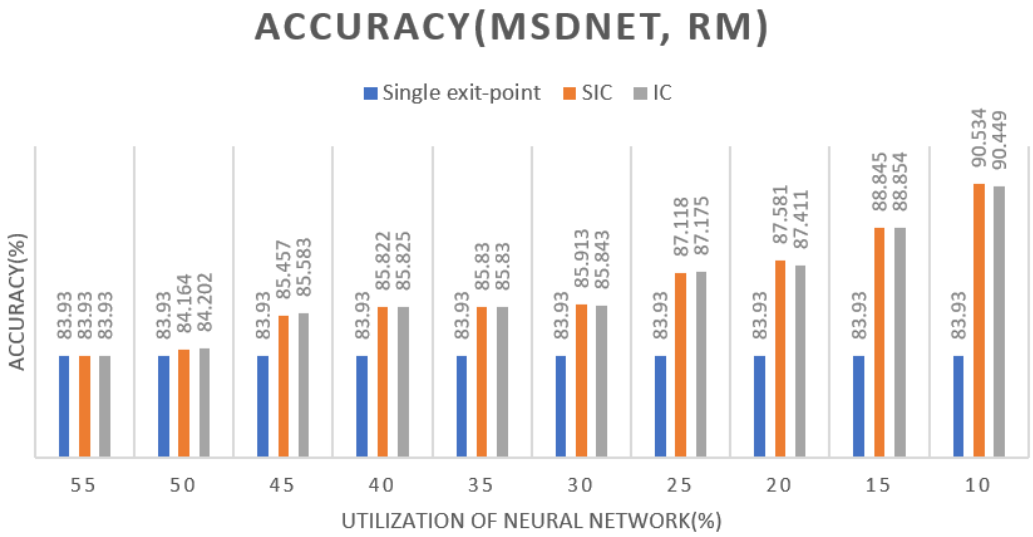


Figure 4.7: Comparison of accuracy, with the RM.

The response time comparison is in the condition of $U = 50\%$ and $U_{NN} = 10\%$.

Similar trends to the previous subsections can be found.

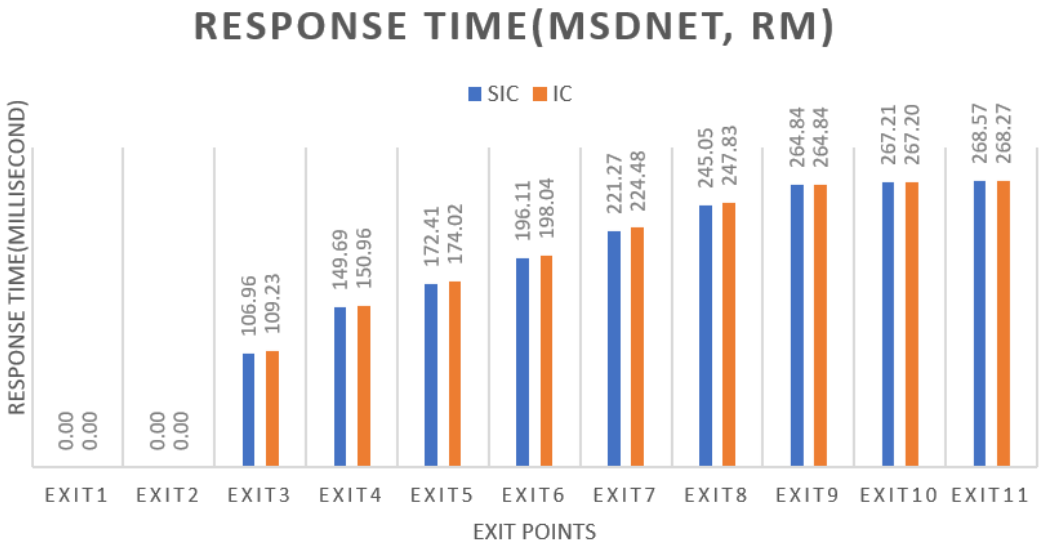


Figure 4.8: Comparison of response time, with the RM.

Chapter 5

Implementation of Hardware BranchyNet

In this chapter, we show the implementation of binarized BranchyNet in hardware on an FPGA board with the hardware description language VHDL. The network structure that we implemented is the VGG-16 based BranchyNet that we introduced in Chapter 2. It has 18 convolutional layers and 9 full connection layers. The BNN(binarized neural network [1]) uses '1' and '0' as the weight parameters, and the values in feature maps generated from each layer are also represented by a binary value. In BNN, the multiplication operations used in the typical neural network are replaced by the XNOR operation, and the accumulation of multiplication result is also replaced by the popcount operation. The XNOR and popcount operations can be simply implemented with a few logic gates, so the performance and resource usage of BNN are much better than the typical neural network.

The architecture of the hardware implementation referred to the implementation by Guo Jiajun [4]. The only difference is the implementation of batch normalization. We referred to Manuele [11]'s paper for batch normalization.

When we use the formal way to do the batch normalization in the inference phase, we perform the operation like Formula 5.1. Assume the input data of batch normalization is X, and there are m values in X. Formula 5.1 as follows.

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i,$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2,$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

$$y_i = BatchNorm_{\gamma,\beta}(x_i) = \gamma\hat{x}_i + \beta. \tag{5.1}$$

y_i is the i-th output from the batch normalization, and γ and β are the parameter which are trained by the backpropagation algorithm [10].

In Manuele [11]'s paper, the calculation of batch normalization in the inference phase uses Formula 5.2. In the formula, $\varphi(m, x, y)$ is the popcount result of the pixel [x, y] in m-th channel. $thresh(m)$ in the formula is the calculated threshold.

$$pixel(m,x,y) = \begin{cases} \varphi(m,x,y) \geq thresh(m) & if\ \gamma \geq 0, \\ \varphi(m,x,y) \geq thresh(m) & if\ \gamma < 0, \\ 1 & if\ \gamma = 0\ and\ \beta \geq 0, \\ 0 & if\ \gamma = 0\ and\ \beta < 0. \end{cases} \tag{5.2}$$

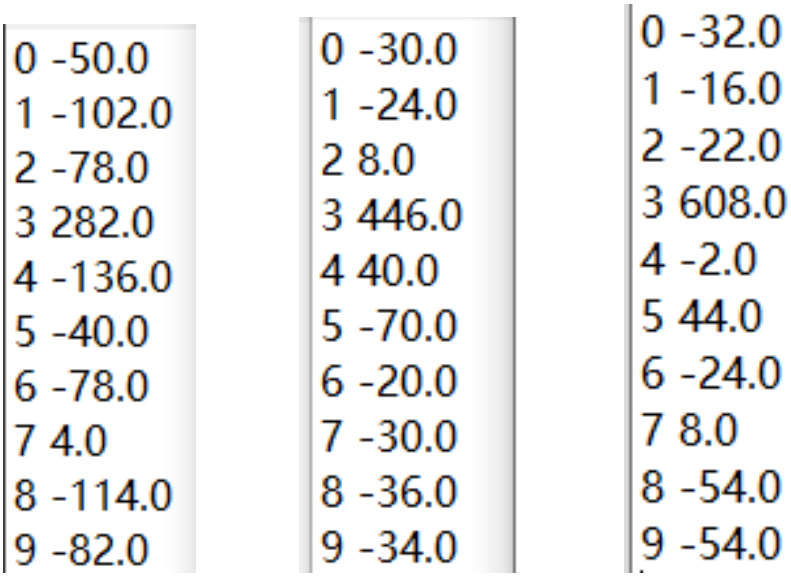
The $pixel(m, x, y)$ is the calculated pixel located at coordinate [x, y] in m-th channel of the feature map.

The formula used for calculating the threshold $thresh(m)$ is Formula 5.3:

$$thresh(m) = \begin{cases} \lfloor \mu_m - \beta \cdot \sigma/\gamma \rfloor & if\ \gamma > 0, \\ \lceil \mu_m - \beta \cdot \sigma/\gamma \rceil & if\ \gamma < 0. \end{cases} \tag{5.3}$$

By applying Formula 5.2, we can get the binarized result with only comparison operation. This operation does not utilize too much LUTs on FPGA, but bring less impact to the performance compare with the formal batch normalization calculation.

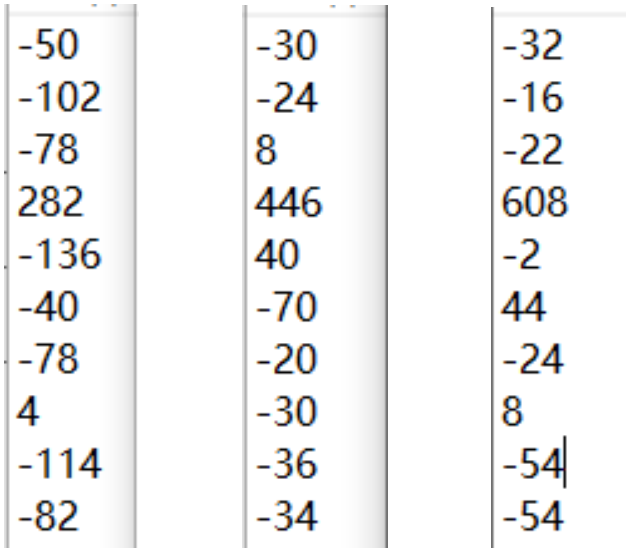
The execution result from the software is in Figure 5.1. Figure 5.1(a) is the output of "EXIT1", Figure 5.1(b) is the output of "EXIT2" and 5.1(c) is output of "EXIT3". In Figure 5.1, the left side is the class number, and on the right side is the output of the final layer of BranchyNet. In figure 5.1(a), this result is from "EXIT1". The largest number of output in this figure is 282.0, and the corresponding class is class 3, so the classification result is class 3. Similar results can be seen in Figure 5.1(b) and Figure 5.1(c).



(a) EXIT1 (b) EXIT2 (c) EXIT3

Figure 5.1: Output from software.

The result from hardware implementation is in Figure 5.2. The first line of output from hardware corresponds to class 0, the second line corresponds to class 1, and so on for the rest of the lines of the classification result.



(a) EXIT1 (b) EXIT2 (c) EXIT3

Figure 5.2: Output from hardware.

By comparing the execution result of the last layer of the network, we can know the BranchyNet in hardware implementation is the same as the output of software implementation. The hardware implementation of FPGA is correct.

The accuracy for each exit-point of the hardware implementation is shown in Table 5.1.

Exit point	Accuracy
Exit1	0.7947
Exit2	0.8091
Exit3	0.8111

Table 5.1: Accuracy for each exit-point

Here is the execution time for each exit-point in Table 5.2, where the unit of time is in seconds.

Exit point	Time(sec)
Exit1	0.04068265
Exit2	0.040690265
Exit3	0.04751765

Table 5.2: Execution time for each exit-point

FPGA device we choose for running this BranchyNet is xcvu440-flagb2377-1-l(Xilinx Ultra Scale series). The detail of the resource usage is in Table 5.3.

Resource	Available	Utilized	Utilization(%)
LUTs	2532960	1857461	73.33
Registers	5065920	374341	7.39
CARRY8	316620	876	0.28
F7 Muxes	1266480	170875	13.49
F8 Muxes	633240	38091	6.02
BRAM36	2520	1151	46.15
BRAM18	5040	24	0.48

Table 5.3: Available resource and utilization.

Chapter 6

Conclusion and Future work

6.1 Conclusion

The use of CNN will be increased even in the computation resource critical environment.

In our research, we used the untypical CNN, i.e., BranchyNet and MSDNet as the neural network model to implement the neural network task in a real-time system, and considered the scheduling of neural network tasks from the real-time computation view, instead of using the typical way. We found that when we apply the neural network model which has multiple exit-point and regard it as an imprecise computation task, it will give us a higher accuracy than the typical CNN model. And within the scheduling method in "IC" and "SIC", we found the "SIC"(the server-based imprecise computation) method gives a shorter response time than IC.

Besides the scheduling method for the multiple exit-point, we also implemented the BranchyNet on FPGA with VHDL with the consideration of the hardware implementation being faster than the software implementation on general-purpose CPU.

6.2 Future work

For future work, the implementation of BranchyNet on FPGA. Our implementation is slower than other's, so as one of the future work, refine the implementation of the hardware implementation to increase the speed and reduce the resource utilization.

Bibliography

[1] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[2] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense convolutional networks for efficient prediction. *CoRR*, abs/1703.09844, 2017.

[3] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[4] Guo Jiajun. Study on lightweight deep neural network with quantization and ensemble methods. 03 2019.

[5] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012.

[7] kuanglab. branchynet, 2019.

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

[9] Jane Liu, Wei-Kuan Shih, Kwei-Jay Lin, Riccardo Bettati, and Jen-Yao Chung. Imprecise computations. *Proceedings of the IEEE*, 82:83 – 94, 02 1994.

[10] David Rumelhart, Geoffrey Hinton, and Ronald Williams. Learning representations by back-propagating errors. *Nature*, 323:533 – 536, 10 1986.

[11] Manuele Rusci, Lukas Cavigelli, and Luca Benini. Design automation for binarized neural networks: A quantum leap opportunity? *CoRR*, abs/1712.01743, 2017.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[13] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. *CoRR*, abs/1709.01686, 2017.