A Modified Minimal Gated Unit and Its FPGA Implementation

by

Tong Zhu

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Tong Zhu, 2020
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

A Modified Minimal Gated Unit and Its FPGA Implementation

by

Tong Zhu

Supervisory Committee

Dr. Xiaodai Dong, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Mihai Sima, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Recurrent neural networks (RNNs) are versatile structures used in a variety of sequence data-related applications. The two most popular proposals are long short-term memory (LSTM) and gated recurrent unit (GRU) networks. Towards the goal of building a simpler and more efficient network, minimal gated unit (MGU) has appeared and shown quite promising results. In this project, we present a simple and improved MGU model, MGU_1, implemented on scalable field programmable gate arrays (FPGA). Experiments with various sequence data show that the MGU_1 has better accuracy compared to the MGU. The accelerator implemented on FPGA accelerates the inference phase utilizing the model trained on our indoor localization data set. It has two layers of MGU_1 and each has 32 hidden units. The accelerator can achieve 142 MHz and 60 GOPS on the Xilinx XC7Z020 FPGA and outperforms the Intel i5-5350U based software solution by two orders of magnitude.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

1.1 Motivation 1
1.2 Contributions 2
1.3 Organization 3

2 Background and Indoor Localization **4**

2.1 Supervised Learning 4
2.2 Recurrent Neural Networks 4
2.2.1 LSTM 6
2.2.2 GRU 8
2.2.3 MGU 10
2.3 Indoor Localization 10

3 MGU Optimization and Experimental Results **14**

3.1 Optimization 14
3.2 Experimental Results 16
3.2.1 Fashion MNIST 16
3.2.2 LibriSpeech 18
3.2.3 PTB 18
3.2.4 RSSI data set 21

4 FPGA Implementation **23**
4.1 FPGA Platform 23
4.2 Related Work 26
4.3 Architecture 26
4.3.1 Top Level Architecture 26
4.3.2 The Sub-Modules 30
4.3.3 PE 31
4.3.4 Activation module 33
4.3.5 Timing Schematic 39
4.4 Implementation and Result 40
4.4.1 Dynamic fixed point 40
4.4.2 Implementation 41
4.4.3 Implementation Result 42

5 Conclusions and Future Work **46**

Bibliography **48**

List of Tables

Table 4.1 DSP Modes 32
Table 4.2 Tanh approximation equations 35
Table 4.3 Resource Utilization of activation module 39
Table 4.4 Resource utilization of the accelerator 42
Table 4.5 Performance Comparison 45

List of Figures

2.1 Vanilla RNN 5
2.2 LSTM 7
2.3 Illustration of (a) GRU [1] and (b) MGU 9
(a) GRU 9
(b) MGU 9
2.4 RSSI RNN model 11
2.5 RSSI RNN model optimized 12
2.6 Task schedule after optimization 13

3.1 Optimized model MGU_1 15
3.2 Learning curves for training and validation on Fashion MNIST data set 17
(a) Training 17
(b) Validation 17
3.3 Learning curves for training and validation on LibriSpeech data set . 19
(a) Training 19
(b) Validation 19
3.4 Learning curves for training and validation on PTB data set 20
(a) Training 20
(b) Validation 20
3.5 Learning curves for training and validation on RSSI data set 22
(a) Training 22
(b) Validation 22

4.1 A Zybo Z7-20 FPGA board, taken from [2] 25
4.2 Architecture of one layer MGU_1 accelerator 28
4.3 Data path of one layer MGU_1 accelerator 29
4.4 Architecture of the input module 30
4.5 Sequence of matrix multiplication 30
4.6 State machine of the forget module 32

4.7 Architecture of one PE 33
4.8 Piecewise linear approximation of tanh function 34
4.9 Error introduced by PWL and LUT methods 36
4.10 Pipeline structure of the activation module 37
4.11 Pipeline timing diagram of activation module 37
4.12 Configurable pair of DSP slices and Activation modules 38
4.13 Timing schematic for MGU_1 39
4.14 Dynamic fixed point numbers 40
4.15 Illustration of the program flowchart 43
4.16 The ground truth trajectory and the predicted trajectory 44

Chapter 1

Introduction

1.1 Motivation

In this era of information explosion, we obtain a great volume of data each day. Deep neural networks (DNNs) is one of the tools to process this massive data for various applications. Multiple layer neural networks [3, 4] with a nonpolynomial activation function can be used as a universal approximator for any other functions. DNNs are currently the basis of many artificial intelligence applications. These DNNs are deployed in various applications ranging from self-driving cars, cancer detection to complex games. In many fields, DNNs can surpass human accuracy. The outstanding performance of the DNN comes from its ability to use statistical learning methods to extract features from raw sensory data and to obtain an effective representation of the input space from a large amount of data. This is different from previous methods that used manual feature extraction or expert design rules.

Recurrent neural network (RNN) is very successful in processing time-series data [5, 6, 7]. However, the vanilla RNN has very poor performance on sequence-based tasks with long-term dependencies. Hochreiter and Schmidhuber [8] proposed LSTM in 1997 to deal with this issue. LSTM is a recurrent neural network with sophisticated recurrent hidden units, which leads to more parameters and more training time. Cho et al. [1] proposed a gated recurrent unit (GRU) in 2014, and evaluation in [9] and [10] both show that the simplified structure GRU can reach similar accuracy as the LSTM or even outperform it in some problems.

In this project, we aim to implement an RNN accelerator for inference on an embedded field programmable gate arrays (FPGA) in indoor localization tasks instead of

the central processing unit (CPU) or the graphics processing unit (GPU). Most CPUs consist of large caches (high-speed buffer memory) and control units to ensure the orderly execution of instructions. However, there are not many parts responsible for massive data operations in parallel. Compared with the small number of arithmetic-logic units in a CPU, GPU is a huge computing matrix as a whole. A GPU has thousands of computing cores to support parallel computing in deep learning, greatly speeding up the training process. But the hardware structure is fixed and does not have programmability. If the deep learning algorithm changes significantly, the GPU cannot flexibly configure the hardware structure. Moreover, the energy consumption is very high. An FPGA has more parallelism than a CPU and more flexibility and smaller power consumption when comparing to a GPU. FPGAs are programmable hence they can realize the status quo structures of the neural network efficiently. Also, it is easy to integrate pre-processing algorithm modules into the accelerator while keeping relatively low power consumption and high performance.

The FPGA implementation of the RNN is subject to computational complexity and memory resource constraints. An embedded platform may not be able to support a complex neural network owing to limited resources. Therefore, we want a simple RNN structure while achieving similar accuracy. Recently, a minimal gated unit (MGU) was proposed by Zhou et al. [11] based on the GRU that only keeps the forget gate. This is a simplified version of the LSTM and GRU structures. The MGU has only roughly half the network size of that of the LSTM, and 67% of that of the GRU. As a result, an MGU needs less training and inference time.

1.2 Contributions

With fewer parameters and a simpler structure, GRUs can achieve similar performance to LSTMs in most applications. MGUs, however, have underperformed LTSMs and GRUs both in terms of convergence time and accuracy in a lot of scenarios. Therefore,

- in this report, we optimize the structure of the MGU into a modified model MGU_1. It not only has fewer parameters when comparing with the GRU but also has improved performance at the same level as the GRU in our experiment.

- A pipelined and scalable architecture of the MGU_1 accelerator for the FPGA is proposed. The accelerator is applied to the inference stage. In this architecture,

we use dynamic fixed-point numbers to reduce resource requirements. Moreover, a technique to share the FPGA resources among the needed operations is implemented. Then we carefully design an orchestrated pipeline operation for controlling the data flow. At last, we create a four-stage pipelined activation module to maximize the scalability and reusability. The number of activation modules can be changed considering the number of neurons and FGPA resources. In general, the proposed architecture has the flexibility of adjusting its resource usage to deploy on different FPGAs to meet a variety of needs. The accelerator can achieve 142 MHz and 60 GOPS on the Xilinx XC7Z020 FPGA and outperforms the Intel i5-5350U based software solution by two orders of magnitude.

1.3 Organization

The rest of this report is organized as follows. In the second chapter, we introduce the data flow in both the GRU and the MGU and the background of the indoor localization model. In Chapter 3, we propose an optimized MGU model - MGU_1 and evaluate all three models separately on four different data sets - the fashion MNIST data set, the LibriSpeech data set, the Penn TreeBank (PTB) data set and the received signal strength indicator (RSSI) data set. Then in Chapter 4, we train the MGU_1 model on the RSSI data set, implement the model on the FPGA, and compare the results using other platforms. Chapter 5 concludes the paper and addresses future work.

Chapter 2

Background and Indoor Localization

2.1 Supervised Learning

Supervised learning is a method in machine learning. It can learn from the labeled training data set to establish a model and then infer new instances based on this model [12]. The training set is composed of a series of training examples, and each training example is composed of an input object (usually a vector) and expected output. The output of the function can be a continuous value (called regression analysis) or predict a classification label (called classification).

Supervised machine learning has two stages, namely the training stage and the inference stage. The training uses the training data set and trains the model to match predictions with labels. After the training phase is completed, the inference phase (also called the test phase) will test the performance of the model. We use new data set to test the performance to ensure that the model can handle new data in real applications.

2.2 Recurrent Neural Networks

The output of RNNs is based on the information learned from earlier time steps, which gives it merit on sequential data tasks [13]. We use the subscript t to represent different positions of the input sequence, and use h_t to represent the hidden layer state vector at time t, and x_t to represent the input at time t. The hidden layer state

vector at the moment t depends on the current input x_t and the hidden layer state vector at the previous time step h_{t-1}:

$$h_t = f(x_t, h_{t-1}), \tag{2.1}$$

where f is a nonlinear mapping function. A common approach is to calculate the linear product of x_t and h_{t-1} and then apply a nonlinear activation function, written as

$$h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h), \tag{2.2}$$

where W_{xh} and W_{hh} are parameter matrices, and tanh is used as the activation function.

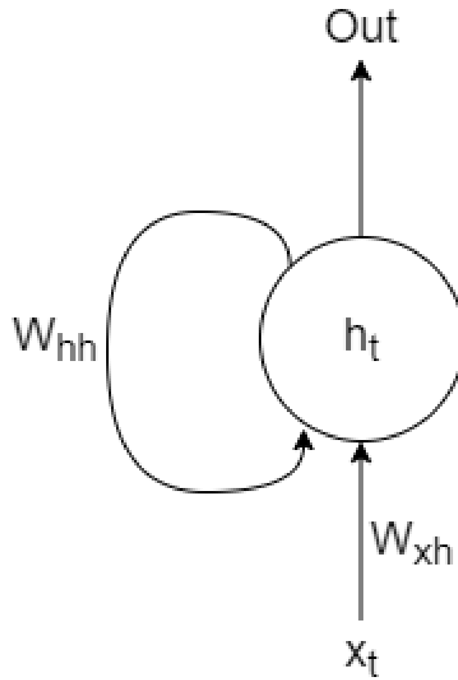


Fig. 2.1: Vanilla RNN

As we can see from Fig. 2.1, the calculation is carried out from bottom to top. The whole calculation includes three steps: input x_t and h_{t-1}, multiply W_{xh} and W_{hh} respectively, add, and undergo tanh nonlinear transformation. We can think that h_t stores the memory in the network, which records the input information $x_1, ..., x_t$ till time step t.

Nonetheless, the vanilla RNNs are not good at long sequential tasks due to vanish-

ing or exploding gradients [14]. If we simplify the network by making the activation functions linear and consider a univariate version, we get a simplified time-related component of back-propagation from [14] as

$$\frac{\partial h_T}{\partial h_1} = \omega^{T-1}, \tag{2.3}$$

where h_T and h_1 are the hidden layer states at time step T and time step 1 respectively and ω is the weight value. It can easily explode or vanish when T is large (long-term) unless ω is very close to 1. For instance, if $\omega = 1.1$ and $T = 100$, we get $\partial h_T / \partial h_1 = 12527.8$, whereas if $\omega = 0.9$ and $T = 100$, we get $\partial h_T / \partial h_1 = 0.00003$.

2.2.1 LSTM

LSTM is a special kind of RNN, which can deal with the problem of gradient disappearance and gradient explosion during long sequence training [8]. LSTM is also to decide what history information to keep and what to drop. Compared with the vanilla RNN, the classic LSTM outputs the hidden layer state h_t and the memory of the current unit c_t, and adds three gates based on RNN, which are:

- Forget gate f_t: Control how much the memory of the previous moment is forgotten.
- Input gate i_t: Control how much information can be input effectively at the current moment.
- Output gate o_t: Control how much the memory output at the current moment.

We use the equations of LSTM in [8] but without peep-hole connections:

$$f_t = \sigma(W_{xf} x_t + W_{hf} h_{t-1} + b_f), \tag{2.4}$$
$$i_t = \sigma(W_{xi} x_t + W_{hi} h_{t-1} + b_i), \tag{2.5}$$
$$o_t = \sigma(W_{xo} x_t + W_{ho} h_{t-1} + b_o), \tag{2.6}$$
$$\tilde{c}_t = tanh(W_{xc} x_t + W_{hc} h_{t-1} + b_c), \tag{2.7}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \tag{2.8}$$
$$h_t = o_t \odot tanh(c_t), \tag{2.9}$$

where σ is the logistic sigmoid function, and $\odot$ means component-wise product. Next, long-term and short-term memory needs to calculate the candidate memory cells $\tilde{c}_t$. Its calculation is similar to the three gates described above but uses the hyperbolic tangent function (tanh) which has a range of values in [-1,1] as the activation function.

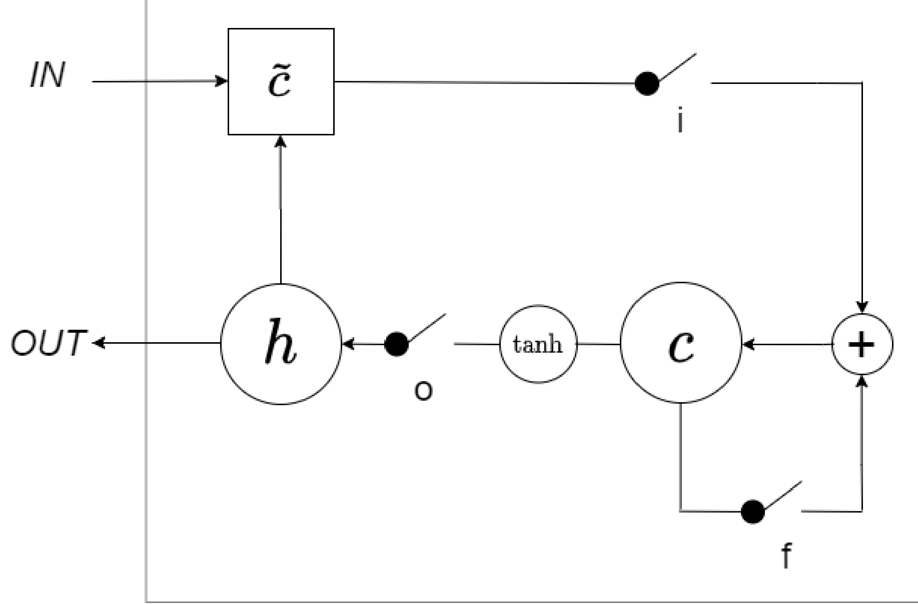


Fig. 2.2: LSTM

As illustrated in Fig. 2.2, the calculation of the current time step memory cell c_t combines the memory cell's information of the earlier time step and the current time step. The forget gate controls whether the information in the memory cell c_{t-1} of the previous time step is passed to the current time step, and the input gate controls how the input x_t of the current time step flows into the memory cell of the current time step through the candidate memory cell $\tilde{c}_t$. If the forget gate is always 1 and the input gate is always 0, the past memory cells will always be saved and passed to the current time step. The output gate controls how the memory of the current unit c_t flows into the hidden layer state h_t.

In short, LSTM can perform better in longer sequences than ordinary RNN [9]. The path containing the forget gate is the key to long-term temporal dependencies [15]. If the input and output gates are off (the value is 0), we can get the only part where gradients flow through time from [15] as

$$\frac{\partial c_T}{\partial c_1} = \prod_{k=1}^{T-1} f_k, \tag{2.10}$$

where c_T and c_1 are the memory cell states at time step T and time step 1 respectively. If the forget gate is on ($f_* = 1$), it will pass the memory cell gradients through unchanged. Therefore, the LSTM architecture is resistant to exploding and vanishing gradients, although these two phenomena are still possible mathematically.

2.2.2 GRU

GRU has fewer parameters and a simpler structure comparing to LSTM but keeps the performance at the same level in most application cases [1, 9]. It is characterized by the following equations [1, 9]:

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z), \tag{2.11}$$

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r), \tag{2.12}$$

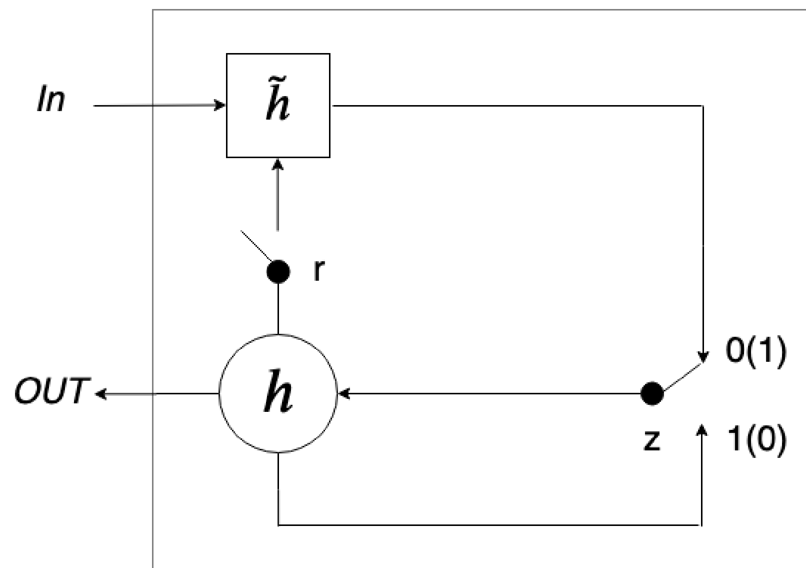
$$\tilde{h_t} = tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h), \tag{2.13}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \quad or \quad h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h_t}, \tag{2.14}$$

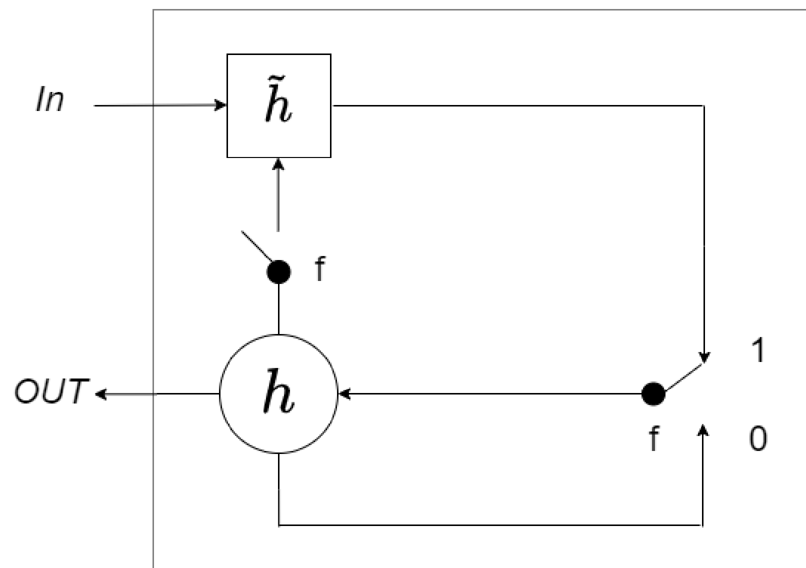
where z_t and r_t are vectors represent the update and reset gates, while h_t represents the state vector for the current hidden state. The candidate hidden state $\tilde{h_t}$ is processed with a hyperbolic tangent. A layer of the network consists of the hidden states and gates to control whether to remember or forget inputs (i.e., x_t), and outputs of the previous time step (i.e., h_{t-1}). Weight matrices (i.e., W_{x*} and W_{h*}) and the bias vectors (i.e., b_*) are to determine each gate and candidate hidden state. As in Eq. (2.14), the current hidden state vector h_t is a linear interpolation between the previous hidden state h_{t-1} and the current candidate hidden state $\tilde{h_t}$. Note that there are two types of the final equation.

The discussion in Chung et al. [9] gives an impression on the function of each gate. Fig. 2.3(a) illustrates the data flow in the GRU. There are two types of switches in the figure. One is the reset gates which work like the normal switch (0 is off). Another is the update gate, similar to a single-pole double-throw switch (SPDT), which connects to either the upper branch or the lower branch when the value is close to 0 or 1. Similar to the forget gate in the LSTM, the update gate and the related path can mitigate the problem of exploding and vanishing gradients.

- Reset gate r: the reset gate is used to control the effect of the previous hidden state h on the current input x. If the previous hidden state is not important



(a) GRU



(b) MGU

Fig. 2.3: Illustration of (a) GRU [1] and (b) MGU

to the current input, the reset gate can be turned off (r close to 0) so that the unit acts as if it is starting a new input sequence.

- Update gate z: the update gate is used to decide whether to ignore the current input x. The update gate can judge whether the current input is important for the expression of the current hidden state and decide how much to update its hidden state.

2.2.3 MGU

MGU is introduced to further lower complexity and maintain comparable accuracy. There are two gates in the GRU to control the data flow. Since the reset gate and the update gate both control the portion of input x and the previous hidden state in the current hidden state, there must be a correlation and redundancy between these two gates. Micro et al. [16] have proved the correlation by cross-correlation. Using the same value for the reset gate and update gate, therefore, is a natural operation. The MGU has only one gate - the forget gate (f_t) - to represent the two gates above as below:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f), \tag{2.15}$$

$$\tilde{h_t} = tanh(W_{xh}x_t + W_{hh}(f_t \odot h_{t-1}) + b_h), \tag{2.16}$$

$$h_t = (1 - f_t) \odot h_{t-1} + f_t \odot \tilde{h_t}. \tag{2.17}$$

The data flow is illustrated in Fig. 2.3(b). Because the structure is nearly the same as the GRU, the MGU can still be resistant to exploding and vanishing gradients. The MGU model has 33% fewer parameters than the GRU model. Therefore, as reported in [11], the MGU model was found to be faster in training over the GRU model in the data sets investigated. In [11] the two models have similar performance in 4 experiments.

2.3 Indoor Localization

RSSI, an optional part of the wireless transmission layer, has been used to determine the quality of the link and whether to increase the strength of the broadcast transmission. It can be used to measure the distance between a signal point and a receiving

point based on the strength of the received signal and then performs positioning calculations based on the corresponding data. Another way of utilizing RSSI data for indoor localization is to consider the RSSI data as fingerprints for each discrete spatial point to discriminate between locations [17]. This data set and the neural network model are based on the fingerprinting method. There are two main processes used in [18] to optimize RSSI data for neural networks:

- The RSSI measurements fluctuate from time to time due to dynamically changing environments. The outliers are filtered out by an iterative recursive weighted average filter which has the form of a low pass filter.
- Random training trajectories are generated based on the constraints of maximum distance a user can move within the sample interval.

Algorithms used for indoor localization have challenges such as spatial ambiguity, RSSI instability, and RSSI short collecting time per location [19]. Because of the sequential correlation of the RSSI data, RNN is adopted to address these issues. The temporal information can be used to distinguish ambiguous locations.

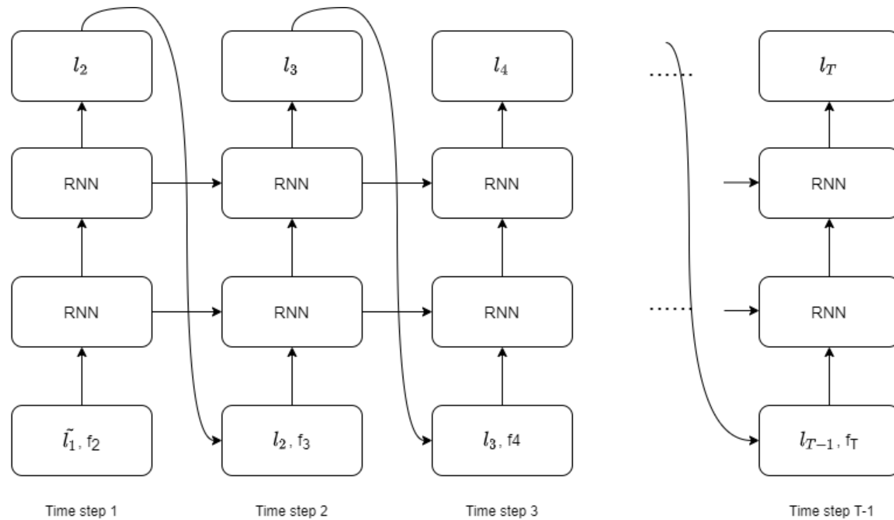


Fig. 2.4: RSSI RNN model

The RNN model used in [18] is illustrated in Fig. 2.4. Input location l in each time step is either the ground truth location (start point, $\tilde{l_1}$) or the predict location from the previous time step $(l_2, l_3, ...)$. The length of a trajectory T defines the memory length which significantly impacts the performance of the RNN model. This model takes in multiple RSSI readings (f_i) and previously predicted locations for the input

and produces multiple locations for the output. For this work, we will use the MGU as the RNN structure to get a fast and simple network. However, for the hardware design, we want to modularize the RNN structure to reach high clock frequency, high throughput, and scalability. If each time step needs the final product from the previous time step, there will be less parallelism. Therefore, we modify the model that all the time steps in one bundle use the same input location data as the first one as shown in Fig. 2.5. We use two layers of MGUs for better performance and the output is a full connection layer. It has a similar performance with the original model in Fig. 2.4.

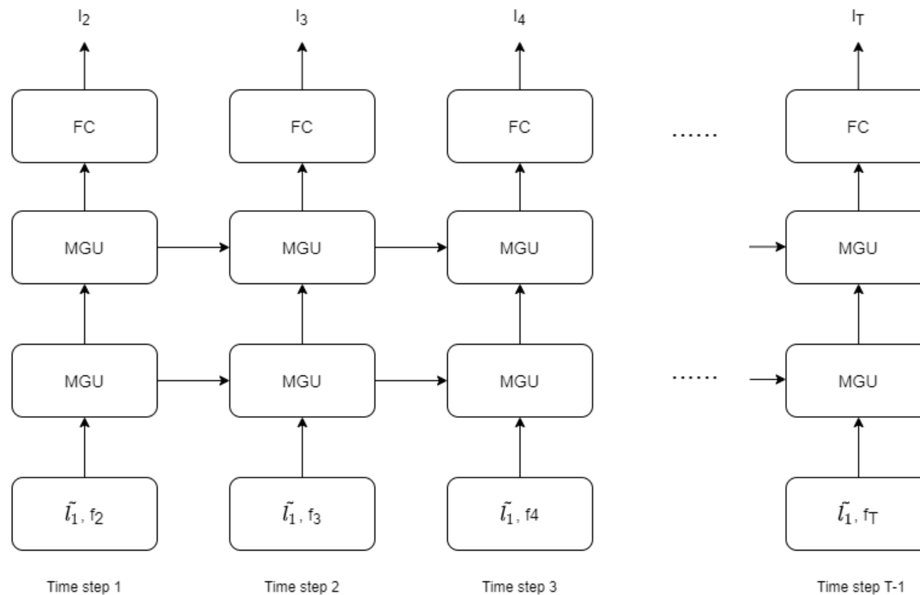


Fig. 2.5: RSSI RNN model optimized

We separate the input module and other parts of the MGU as a way of modularization. If we consider a toy example as shown in Fig. 2.6, the optimized structure needs only 3 time sections to finish 2 time steps comparing to the normal structure of 4 time sections. The efficiency is improved since the input module does not need to wait for the end of other calculations to start the next operation.

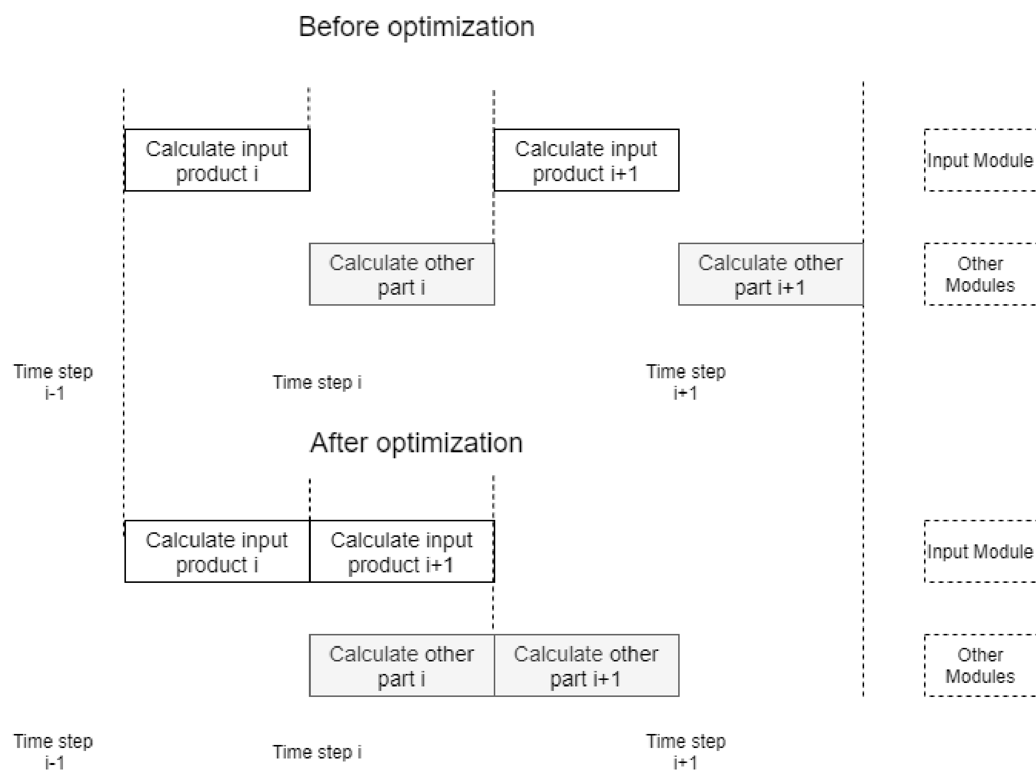


Fig. 2.6: Task schedule after optimization

Chapter 3

MGU Optimization and Experimental Results

3.1 Optimization

In MGU model, when the forget gate f_t is off ($f_t = 0$), the previous hidden state will be cut off from flowing to the candidate hidden state. As a result, the candidate hidden state ($\tilde{h_t}$) only contains the product of the input x. This indicates that the previous hidden state value is not important. For example, if the input x is a word, it might be about to start a new chapter and should be memorized.

For the hidden state (h_t) calculation, when the forget gate f_t is off, the previous hidden state (h_{t-1}) will be put through, and the candidate hidden state ($\tilde{h_t}$) will all drop off. This indicates that the current input is not important. In this case, if the input x is a word, it might be meaningless and should be ignored.

There are conflicts between the reset gate and the update gate when they use the same value f in Eqs. (2.16) and (2.17). We can avoid the conflict in MGU by modifying the Eq. (2.17) to

$$h_t = f_t \odot h_{t-1} + (1 - f_t) \odot \tilde{h_t}. \tag{3.1}$$

We call this modified model MGU_1 which has solved the major conflict in MGU. In this way, when f_t is large, the hidden node output depends more on the previous hidden layer value, while when f_t is small, the output is consistently more heavily influenced by the input x_t.

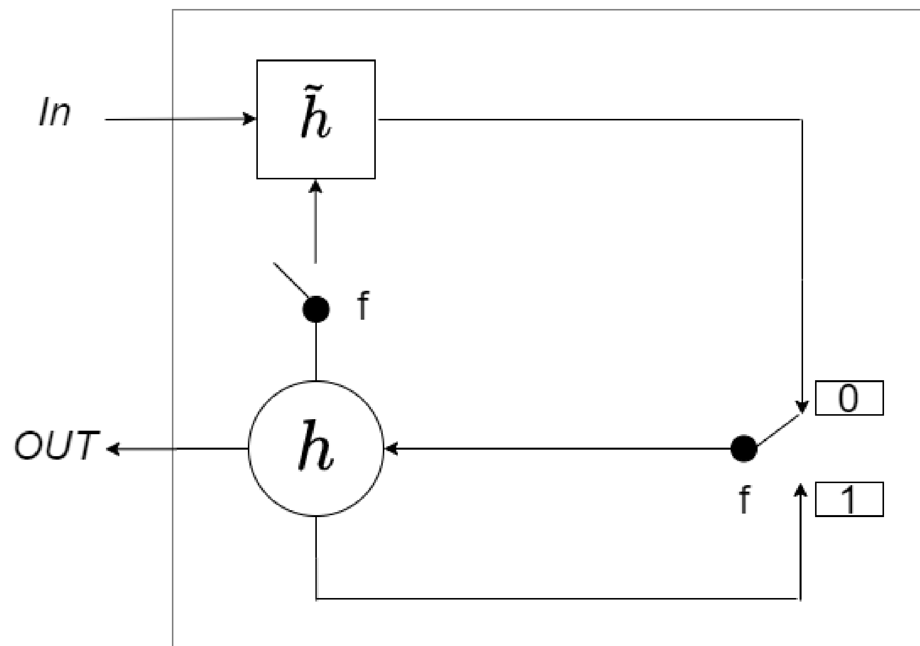


Fig. 3.1: Optimized model MGU_1

3.2 Experimental Results

We evaluate the performance of the MGU_1 by using four data sets. First, the fashion MNIST data set, which is an image classification task, is presented in Section 3.2.1. Then the LibriSpeech data set is used as a speech recognition problem in Section 3.2.2. Also, we use the PTB data set as an advanced language modeling problem to evaluate the performance in section 3.2.3. Finally, we will use the RSSI data set [18] for indoor localization in Section 3.2.4.

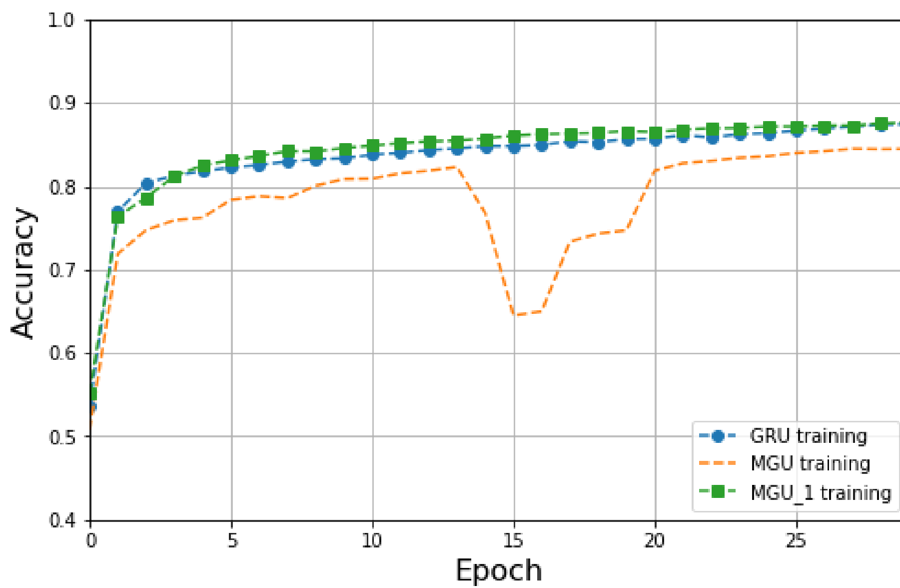
GRU has shown high performance in a great variety of fields, hence we choose it as a baseline. For each task, we train three different recurrent neural networks, each having either GRU units (GRU-RNN, see Section 2.2.2), MGU units (MGU-RNN, see Section 2.2.3), or MGU_1 units. These three neural networks are set to have the same number of hidden units in each task. All RNNs are implemented in Keras and Tensorflow. We keep all models sufficiently small to avoid overfitting.

3.2.1 Fashion MNIST

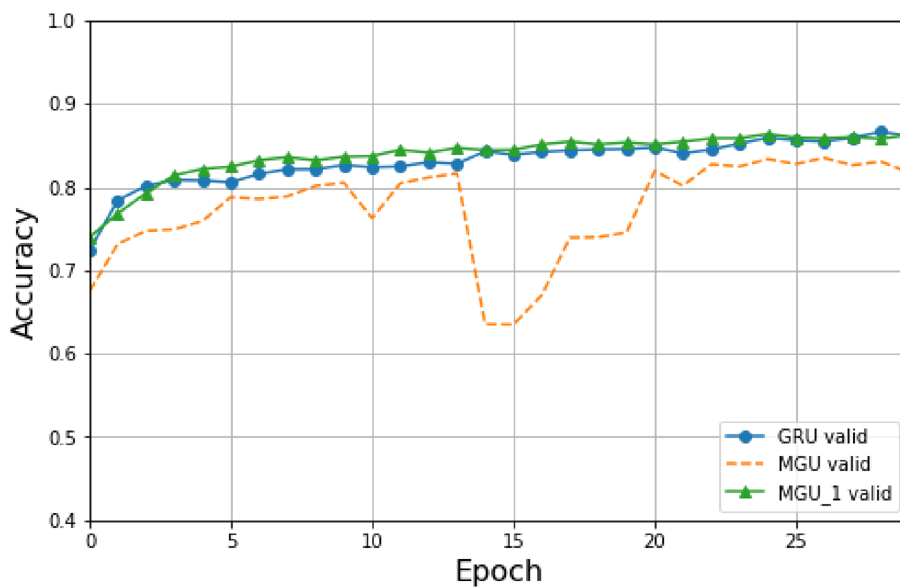
Fashion MNIST [20] is an image data set that replaces the MNIST handwritten digit set. It is provided by the research department of Zalando, a German fashion technology company. It covers pictures of 70,000 different products from 10 categories. The size, format, and training/validating set division of Fashion MNIST is exactly the same as the original MNIST. The training data set has 60000 28x28 grayscale pictures. And there are 10000 more for validation.

We treat each row (28 pixels) as a single input for one time step. Hence, a whole image contains 28 time steps. For this task, we use 64 units and the batch size is 256. Fig. 3.2 shows the accuracy of the 3 models, where the x-axis is the epoch and the y-axis is the accuracy.

The performance of the GRU and the MGU_1 are almost the same, and the MGU falls behind. Moreover, the MGU does not converge as fast as the other two models and has some notable spikes. The MGU's accuracy is lower than that of the GRU and the MGU_1 after 30 epochs. For this task, the MGU and MGU_1 layers both have 11,904 parameters, while the GRU layer has 17,856 parameters. Hence the MGU and the MGU_1 networks need less time for training.



(a) Training



(b) Validation

Fig. 3.2: Learning curves for training and validation on Fashion MNIST data set

3.2.2 LibriSpeech

The LibriSpeech data set is an audiobook data set containing text and speech. It is a corpus of about 1000 hours of 16kHz reading English speech written by Vassil Panayotov [21]. The data comes from the reading audiobooks of the LibriVox project and has been carefully subdivided and aligned. We choose the "dev-clean" subset as our training data set and the "test-clean" subset as our validation data set. The model for this experiment is structured based on a github project [1] and it contains 4 layers - the input layer, the RNN layer, the time distributed layer, and the softmax layer. Moreover, we use the connectionist temporal classification (CTC) [22] as the loss function.

The rank of performance of three models on the LibriSpeech data set is similar to that on the Fashion MNIST problem as shown in Fig. 3.3. For this task, we use 200 hidden units and the batch size is 20. The MGU and MGU_1 layers both have 144,800 parameters in RNN layer, while the GRU layer has 217,800 parameters.

3.2.3 PTB

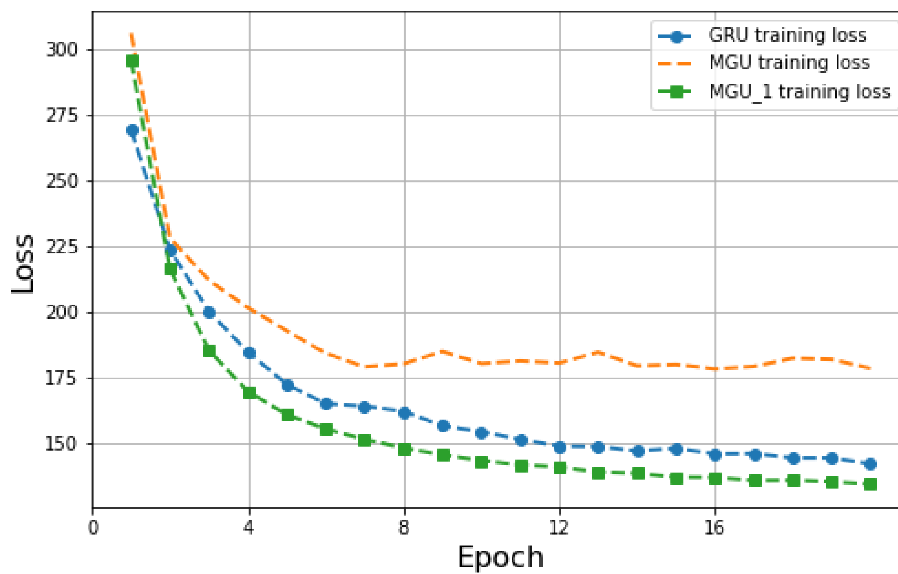
The PTB data set [23] is widely used in natural language processing (NLP) research. We use this data set for the word prediction task and consider the model in [24] [2] as a reference. The size of the vocabulary is 10,000. It has 929k words for training and 73k words for validation. The model contains 2 layers of RNN and each has 200 hidden units. The time steps and batch size are set to 20 and 20 separately. A dropout layer is used and set the drop out rate to 0.2. Then a fully connected layer predicts one of the 10,000 words at the output. As a direct comparison, we show the cross-entropy as the loss function in Fig. 3.4. The x-axis is the epoch number.

The performance of the GRU and the MGU_1 is still better than MGU on this data set. MGU's loss is higher than that of the GRU and the MGU_1 after 20 epochs. For this task, one MGU or MGU_1 layer has 160,400 parameters, while one GRU layer has 241,200 parameters.

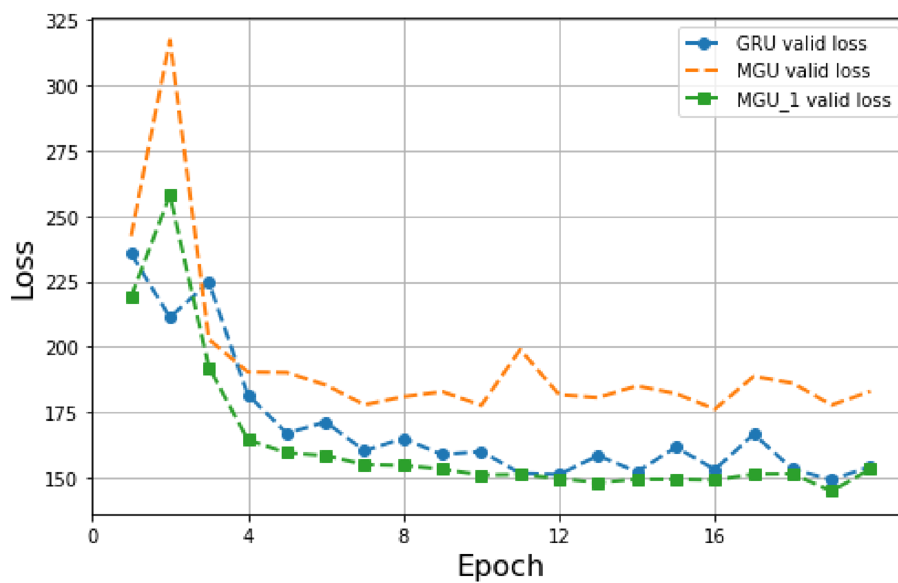
---

[1] https://github.com/udacity/AIND-VUI-Capstone

[2] https://github.com/wojzaremba/lstm

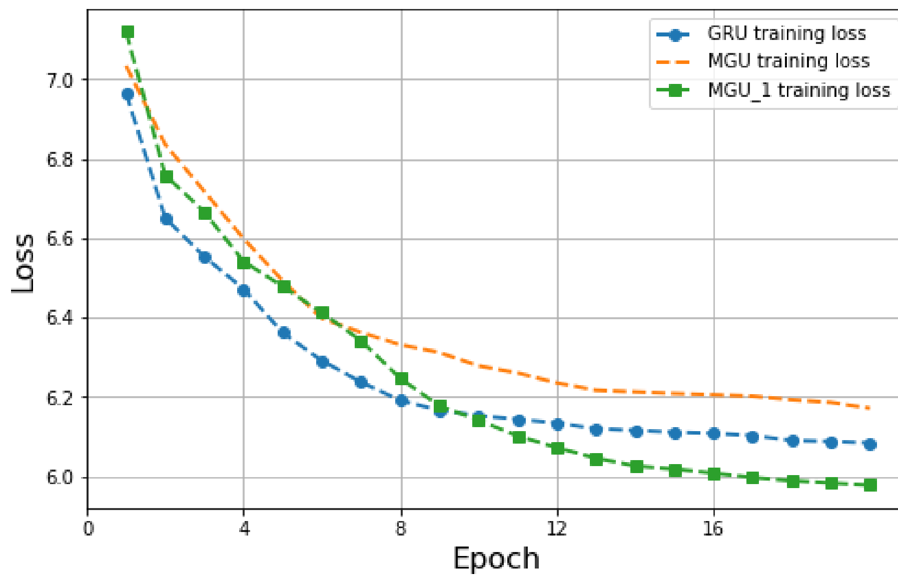


(a) Training

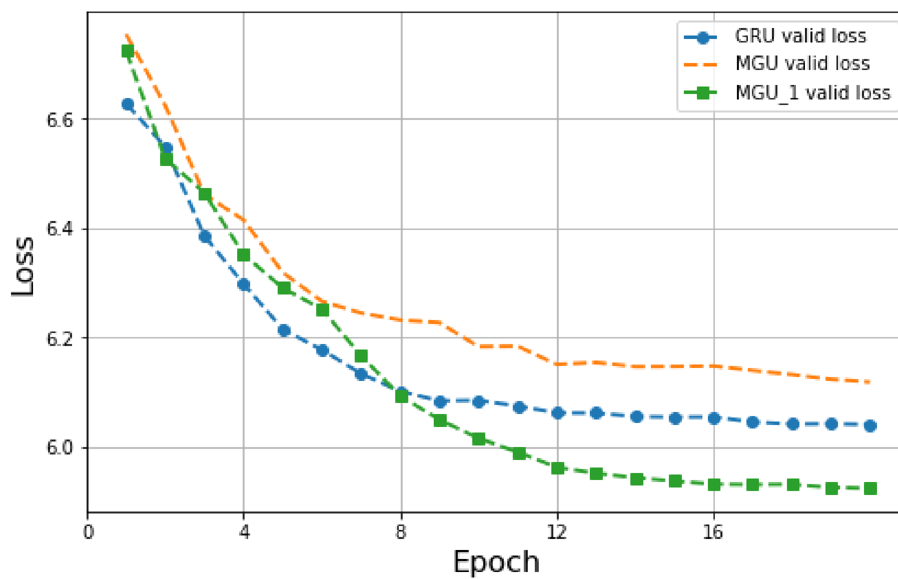


(b) Validation

Fig. 3.3: Learning curves for training and validation on LibriSpeech data set



(a) Training



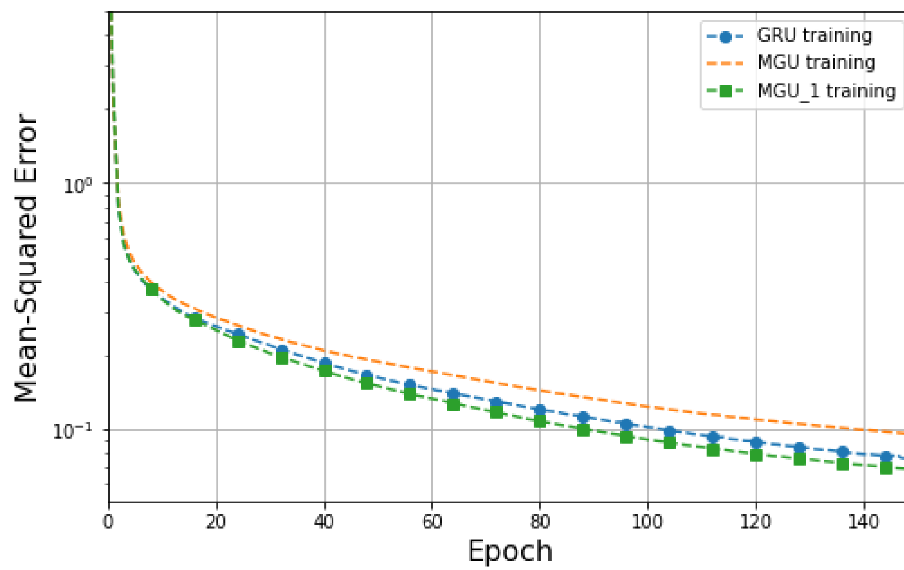
(b) Validation

Fig. 3.4: Learning curves for training and validation on PTB data set

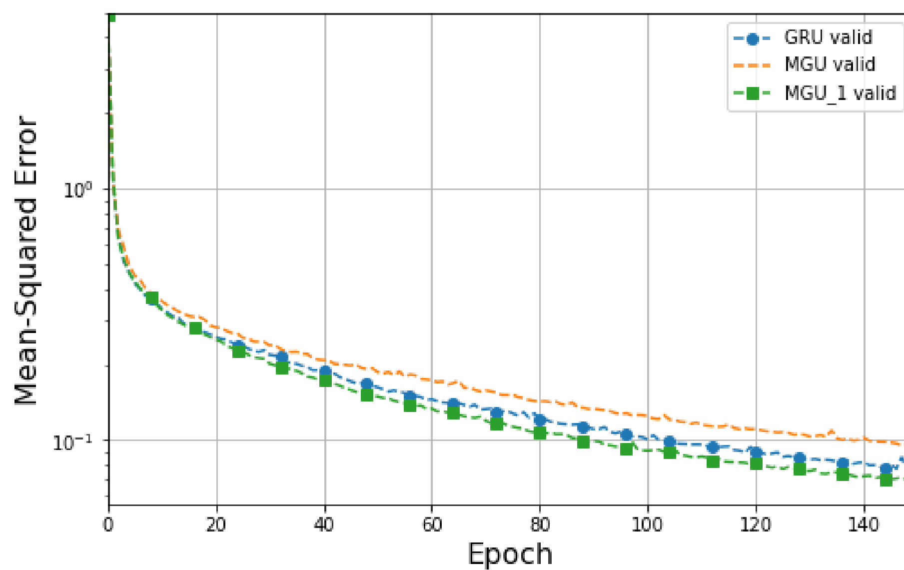
3.2.4 RSSI data set

We obtained the RSSI data set by the localization system proposed in [18]. There are a total of 365,000 randomly generated training trajectories. Each trajectory contains 9 location data and each data has 11 RSSI readings and the ground truth of the first location. We use 270000 trajectories as the training data set and another 30,000 trajectories as the validation data set. Our proposed model is illustrated in Fig. 2.5. For the model, we have 2 layers of MGUs and each has 32 units. The input is the same ground truth combined with RSSI readings. For example, for the first input, $\tilde{l_1}$ is the ground truth of the first location, and $f2$ is the RSSI reading of the second location. The output $l2$ is the prediction of the second location. For all the 9 time steps we use the same ground truth location but with different RSSI readings.

The rank of performance of the three models on RSSI data set is similar to the previous data sets. As shown in Fig. 3.5, the y-axis is in log scale. The MGU or MGU_1 model has 7,170 parameters, while the GRU Model has 10,914 parameters.



(a) Training



(b) Validation

Fig. 3.5: Learning curves for training and validation on RSSI data set

Chapter 4

FPGA Implementation

4.1 FPGA Platform

FPGA is playing an important role in the data sampling and processing industry due to its highly parallel architecture, low power consumption, and flexibility of custom algorithms. In the field of artificial intelligence, to implement custom neural networks and achieve low power consumption, FGPA is widely adopted for inference in a variety of tasks, such as autonomous driving [25] and automatic speech language recognition [26].

The Xilinx Zynq-7000 Scalable Processing Platform [27] tightly integrates the dual ARM Cortex-A9 MPCore processor system with programmable logic and hard IP peripherals, providing a perfect combination of flexibility, configurability, and performance. Using a 28 nm manufacturing process, each product of the Zynq-7000 embedded processing platform series uses a dual-core ARM Cortex-A9 MPCore processing system with NEON and a double-precision floating-point engine. The system is hard-wired to include L1, L2 cache, memory controller, and common peripherals. Although FPGA manufacturers have previously introduced devices with hard-core or soft-core processors, the unique feature of Zynq-7000 is that it is controlled by an ARM processor system rather than a programmable logic element. In other words, the processing system can boot (before the FPGA logic) and run various operating systems independent of the programmable logic at boot time. In this way, designers can program the processing system and configure programmable logic as needed.

In addition to selecting the widely used and popular ARM processor system, an important architectural decision was to use high-bandwidth AMBA Advanced Exten-

sion Interface (AXI) interconnects between the processing system and programmable logic. In this way, it can support the multi-gigabit data transmission between the ARM dual-core Cortex-A9 MPCore processing subsystem and the programmable logic with lower power consumption, thereby eliminating the control, data, I/O and memory faced performance bottleneck.

In Xilinx products, users can configure programmable logic and connect it to the ARM core through the AXI interconnect module to expand the performance and functional range of the processor system. Xilinx and the ARM ecosystem provide a large number of soft AMBA interface IP cores for designers to use in FPGA programmable logic. Designers can use them to build any custom functions required by their target applications.

We choose Zybo Z7-20 [2] as our FGPA board as shown in Fig. 4.1 which is built around the Xilinx XC7Z020 chip. It contains a 667 MHz dual-core Cortex-A9 processor and 1 GB DDR3L with a 32-bit bus. Also, the resource list is 53,200 look-up tables (LUTs), 106,400 flip-flops, 630KB block RAM (BRAM), and 220 digital signal processing slices (DSPs).

- Configurable logic block (CLB): CLB is the main logic resources for implementing sequential as well as combinational circuits. Each CLB has 8 LUTs and 16 flip-flops. Therefore, the combination of many CLBs and the global clock can realize complex digital functions. There are two kinds of slices in CLB, one is called SLICEL and the other is called SLICEM. In addition to basic functions, SLICEM can realize the functions of RAM and shift registers.

- BRAM: There are two types of RAM resources in FPGA. One is distributed RAM. Distributed RAM is implemented by multi-level LUTs cascade. A distributed RAM may be realized by LUTs that are far apart, hence the name distributed. Distributed RAM can make use of abundant and flexible LUT resources. It is suitable for occasions where low latency is not required. The other type of RAM, the block RAM, is a dedicated RAM block resource added to FPGA in addition to logic resources. Compared with distributed RAM, BRAM is specially placed and routed, so that it has high operating speed and low latency.

- DSPs: DSP48E1 is a digital signal processing logic unit included in the Zynq-7000 series. The DSP48E1 can be used to implement different arithmetic operations. For example, multiply accumulator or single-step/multi-step counter.

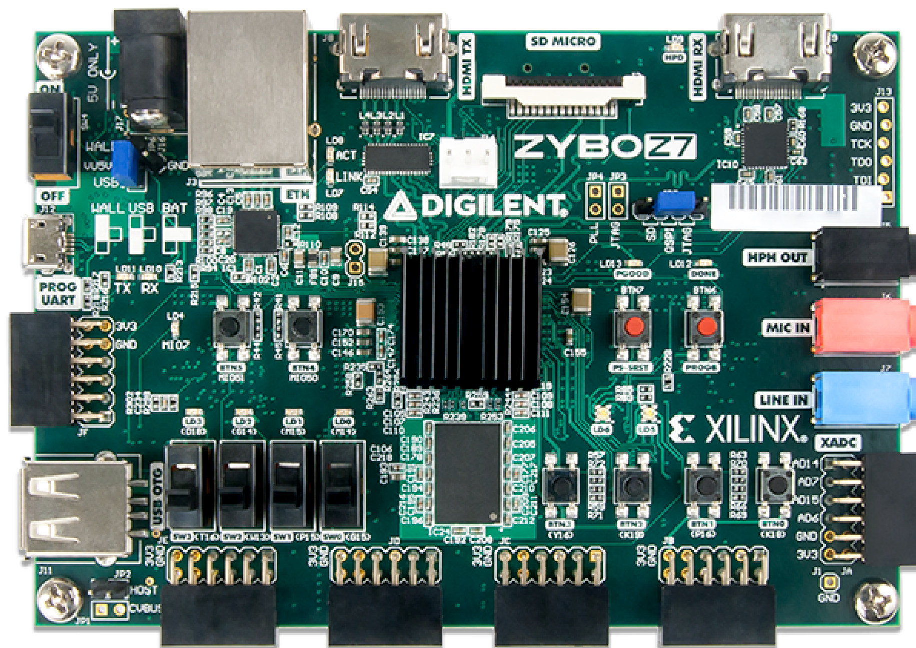


Fig. 4.1: A Zybo Z7-20 FPGA board, taken from [2]

It can also be used to perform various logic operations (for example, AND, OR, and XOR operations). Multiple DSP48E1 slices can be cascaded to perform complex functions. For example, implementing complex multipliers or n-order FIR filters without using additional resources.

Xilinx provides a software development kit for embedded software application projects. Linux application developers can take full advantage of the two Cortex-A9 CPU cores in the Zynq-7000 device to achieve the highest performance in symmetric multi-processor mode.

4.2 Related Work

Co-processors for accelerating computing of CNNs and RNNs have been implemented on FPGAs in recent years. An FPGA implementation that focuses on vanilla RNN is described by [28]. The approach divides the feed-forward phase into two stages: from the input to the hidden layer and from the hidden layer to the output. This architecture also unfolds the RNN model into several timesteps B (previous timesteps) and computes them in parallel. Another FPGA implementation with two layers of LSTM and 128 hidden units was proposed in [29]. It does not unfold the LSTM along with the time domain but uses a single module that consumes the input x and the previous hidden state h_{t-1} at the same time. Moreover, to improve the performance of LSTM, a new architecture was proposed in [30]. It fine-tunes the pipelined structure to have low-power and high-speed features. This implementation can map the architecture onto different types of FPGAs by adjusting the parameters.

Our proposed FPGA implementation of MGU_1 has low-power consumption and high performance as the previous architectures. It can also be mapped to other FPGA platforms by changing the number of processing elements (PEs) in each module. Furthermore, we use dynamic fixed-point numbers and look-up table (LUT) based activation modules to achieve less accuracy loss than previous implementations.

4.3 Architecture

4.3.1 Top Level Architecture

The MGU_1 model has been slightly tuned for FPGA to achieve high efficiency. In [9] there is a variant of Eq. (2.13), to follow which we can modify Eq. (2.16) as below:

$$\tilde{h_t} = tanh(W_{xh}x_t + f_t \odot (W_{hh}h_{t-1}) + b_h). \quad (4.1)$$

The performance is similar to the original model. But in this way, $h_{t-1}W_{hh}$ can be calculated without f_t. Since matrix multiplication takes more cycles than element-wise multiplication in general, the efficiency of the structure is improved. To maximize the parallel ability of the circuit, we further divide the Eqs. (2.15), (4.1), and (3.1) into small parts similar to the approach in [29] as

$$xf_t = W_{xf}x_t + b_f, \quad (4.2)$$

$$xh_t = W_{xh}x_t + b_h, \quad (4.3)$$

$$f_t^* = xf_t + W_{hf}h_{t-1}, \quad (4.4)$$

$$f_t = \sigma(f_t^*), \quad (4.5)$$

$$\tilde{h_t}^* = xh_t + f_t \odot (W_{hh}h_{t-1}), \quad (4.6)$$

$$\tilde{h_t} = tanh(\tilde{h_t}^*), \quad (4.7)$$

$$h_t = f_t \odot h_{t-1} + (1 - f_t) \odot \tilde{h_t}. \quad (4.8)$$

The overall architecture of the proposed MGU_1 accelerator is shown in Fig. 4.2. It is divided into 5 modules: input module, forget module, hidden state module, activation module, and output module. For each new time step, the forget module relies on the output of the input module, and the hidden state module needs products from both the input module and forget module to generate the result. The input module takes the input vectors, generates the product of xf_t and xh_t and sends them to the FIFOs. It only depends on the input value, hence it can operate when the inputs are ready. The forget module calculates the f_t^* value and sends it to the activation module. The hidden state module is responsible for $\tilde{h_t}^*$ and the final activation value h_t. We use registers and interrupt to communicate with the ARM processor. The registers should be set before starting the accelerator. After calculation, an interrupt will be generated to inform the CPU to fetch the data from the output buffer. The data path of the accelerator is shown in Fig. 4.3. The multiply-accumulate (MAC) array is responsible for matrix multiplication. The activation module is to calculate the sigmoid and tanh fuctions. Notably, the input module product is buffered so that the fluctuation is smoothed.

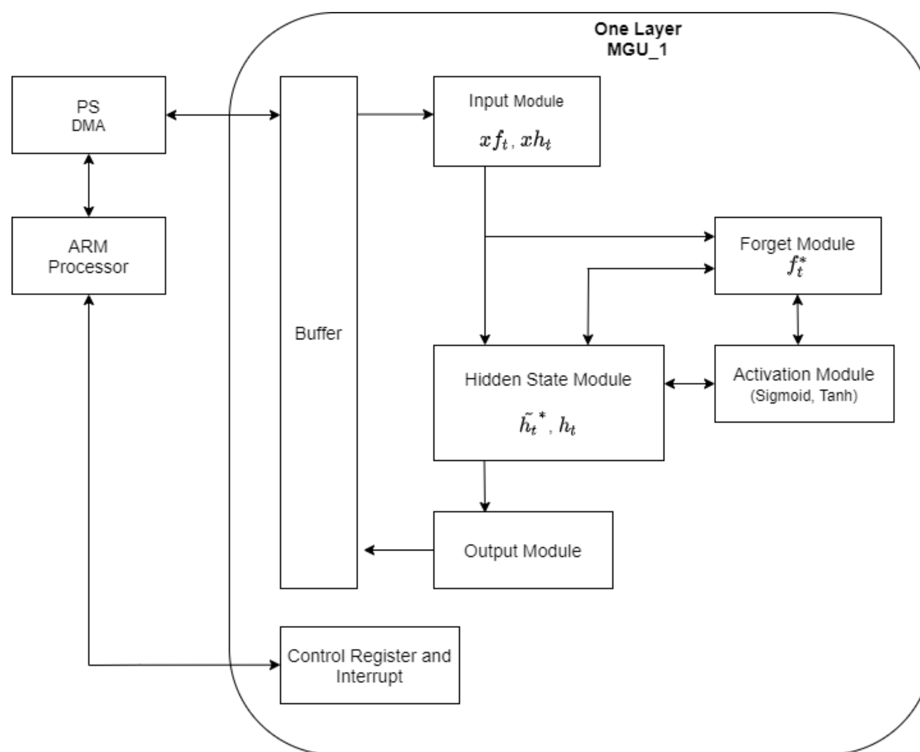


Fig. 4.2: Architecture of one layer MGU_1 accelerator

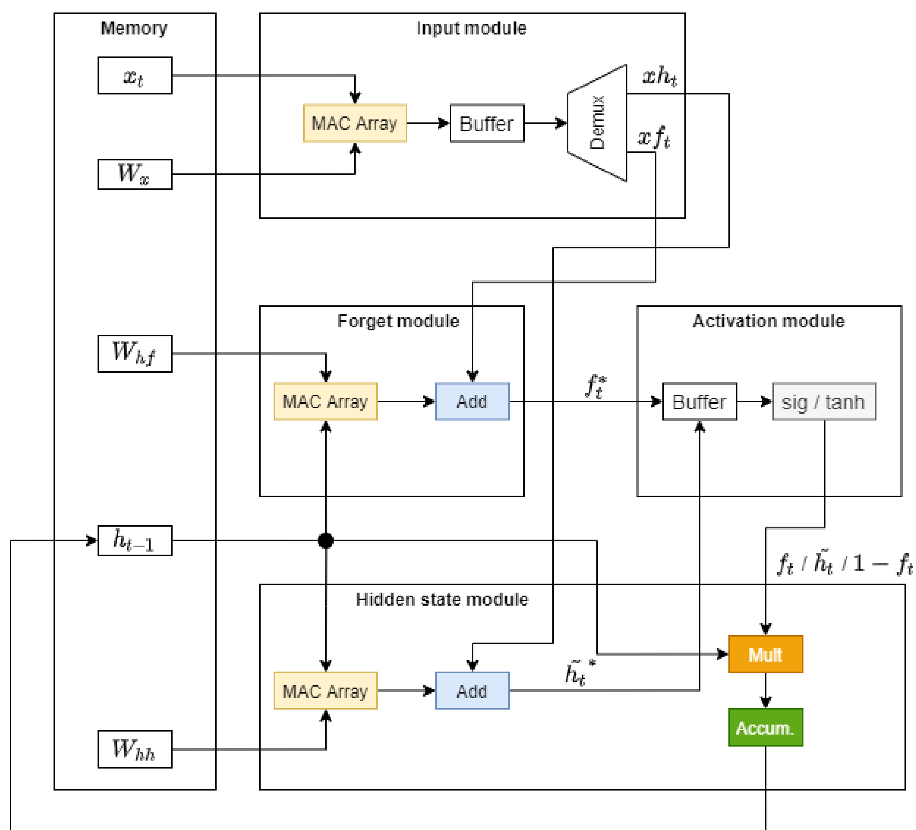


Fig. 4.3: Data path of one layer MGU_1 accelerator

4.3.2 The Sub-Modules

The input module, the forget module, and the hidden state module are three main computational modules in our architecture. They have a similar structure, hence we will take the input module as an example. As illustrated in Fig. 4.4, the input and the output of the input module are buffered to avoid data loss when fluctuation. We use block RAM to store weights and bias since it is an on-chip RAM and faster than off-chip RAM. Two PEs are used to perform the matrix multiplication. The input and output buffers are implemented by FIFOs which can inform the other modules when they are empty or full.

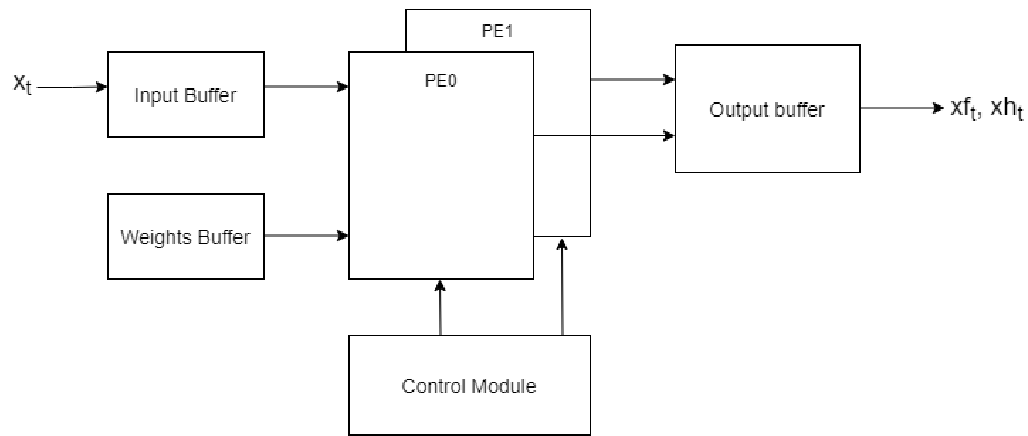


Fig. 4.4: Architecture of the input module

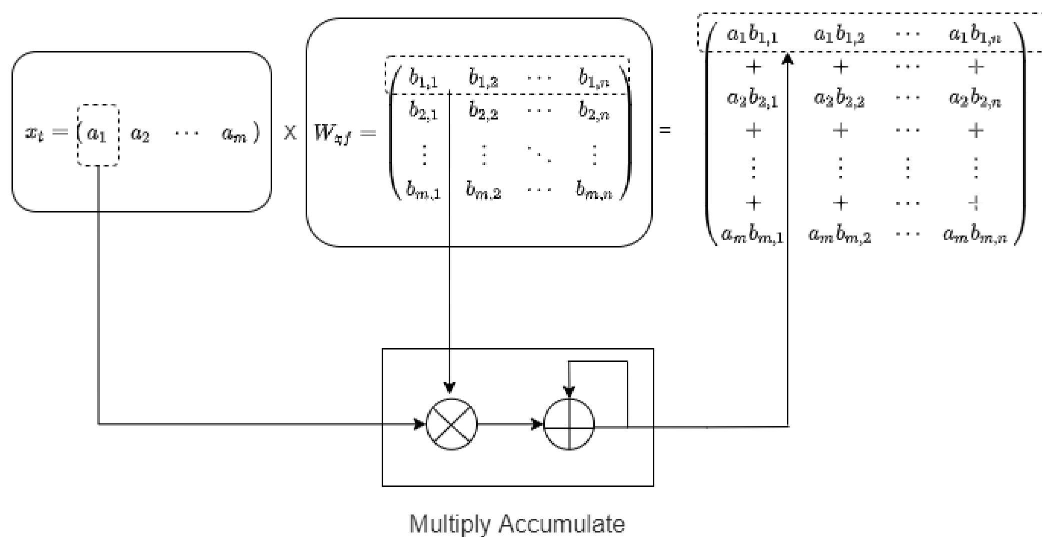


Fig. 4.5: Sequence of matrix multiplication

The sequence of the matrix multiplication is illustrated in Fig. 4.5. The input dimension is m, and n is the number of hidden states. For each cycle, one feature of the input is selected, also one row of the weights matrix is sent to the PE. Then we have the one piece of the product of each hidden state. For example, at first, the input element a_1 and the first row $[b_{1,1}, ..., b_{1,n}]$ are the input of the MAC module. The product we get is $[a_1b_{1,1}, ..., a_1b_{1,n}]$. Next input will be a_2 and $[b_{2,1}, ..., b_{2,n}]$, and the accumulate output will be $[a_1b_{1,1} + a_2b_{2,1}, ..., a_1b_{1,n} + a_2b_{2,n}]$. And the process will continue till the last input element. In total, it needs m cycles to finish the computation if we have an equal number of DSP slices and hidden states. If the DSP number is half of the hidden states, then we have to send half of the row of the weights matrix to the PE and the total cycles used are $2m$.

All the main modules have a control logic that uses a state machine to control the data flow. Also, the state machine controls the timing between these modules. Let's take the state machine of forget module as an example. As shown in Fig. 4.6, the state machine is the control of the flow of Eq. (4.4). When the start signal is set to 1 and the input product (xf_t) is ready, the state machine will jump from IDLE mode to adder mode which adds the xf_t. Since this is the first step and the initial value of h_{t-1} is 0, the calculation of f_t^* in Eq. (4.4) for the first step is done and the request is sent to the activation module. If this is not the last step, the state machine will wait for h_{t-1} ready and then calculate $W_{hf}h_{t-1}$ for the next step. After the calculation, if xf_t is ready from the input module, the state machine will directly transfer to the adder state. Otherwise, the state machine will go to idle mode to wait for the xf_t product ready.

4.3.3 PE

The matrix multiplication and element-wise multiplication are both a series of multiplication and addition, therefore we can use the DSP48E1 module from Xilinx XC7Z020 FPGA as MAC unit to carry out all the calculations.

The DSP slice can have several operating modes. In this structure, we'll use three of them as in Table 4.1. They can be switched by giving different cmd signal values. A, B, and C are input values. P is the feedback of the accumulated value. If we use $A * B + C$ as the initial mode, we can set the input C as the bias input. In this way, we can save one additional cycle for adding the bias. After the first cycle, we have to change the mode input to $2'b00$ to start the accumulation. Moreover, for the

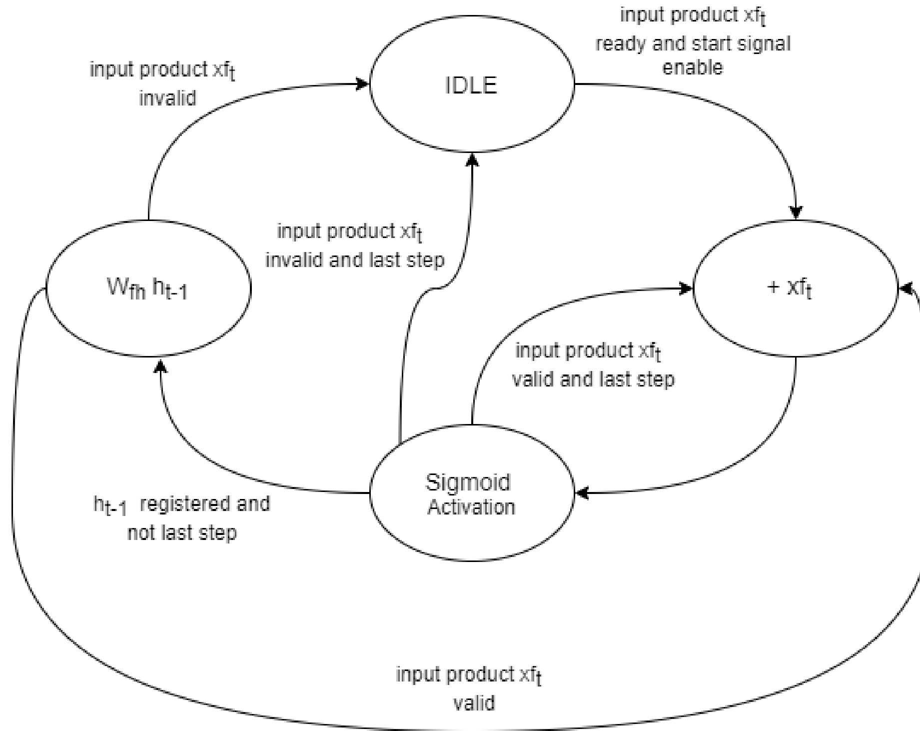


Fig. 4.6: State machine of the forget module

Table 4.1: DSP Modes

cmd[1:0]	**Operation**
2'b00	$A * B + C$
2'b01	$A * B$
2'b10	$A * B + P$

element-wise multiplication, we can reuse the PE by setting the cmd signal to $2'b01$ and $2'b10$ in sequence. Since the element-wise multiplication only needs two cycles, a dedicated module is a waste of resources.

The PE's structure which contains 16 DSP units is shown in Fig. 4.7. For each calculation cycle, the input value is the same for each DSP, while the weights are different. The PE is the fundamental block of the input module, forget module, and the hidden state module. It is scalable and reusable. For example, if our model has 32 hidden units, we can use 2 PEs for each module. If we want to have 64 hidden units, 4 PEs can be used.

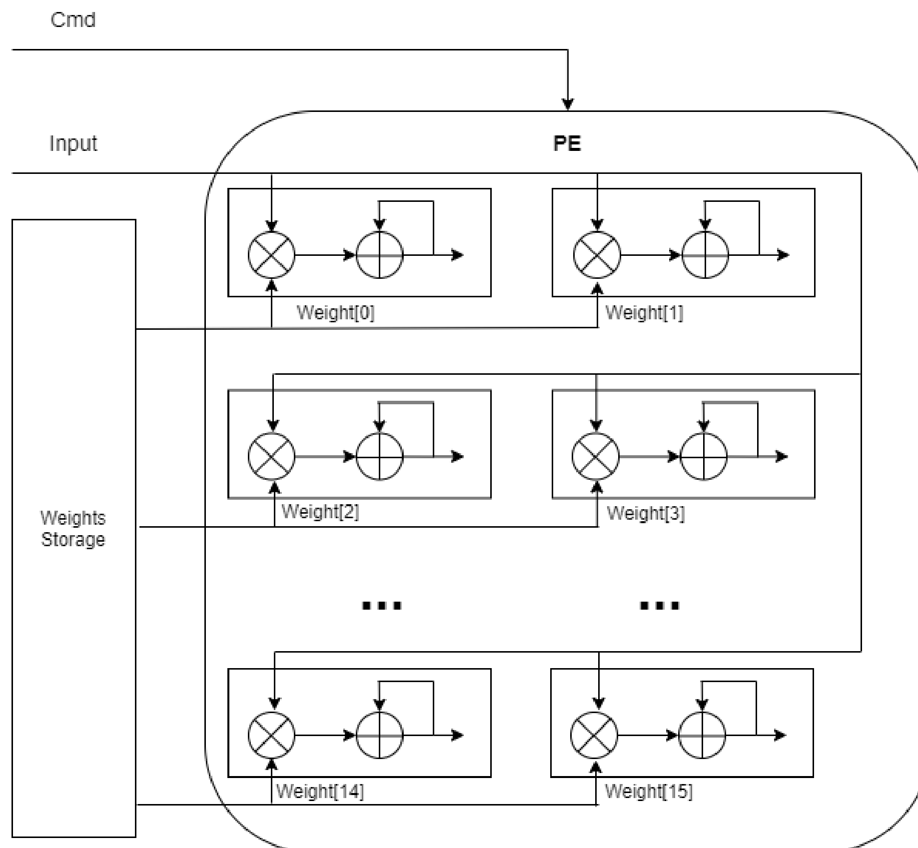


Fig. 4.7: Architecture of one PE

4.3.4 Activation module

The activation module performs sigmoid or tanh functions to generate output f_t and $\tilde{h}_t$, which are written to the storage space in the activation module.

- Sigmoid Function: Sigmoid function is a common S-shaped function, also known as the sigmoid growth curve. Because of its properties such as single increase, the sigmoid function is often used as the threshold function of neural networks to map variables to 0 and 1 intervals. In other words, the sigmoid function is equivalent to compressing a real number to between 0 and 1. When the input is a very large positive number, the output will approach 1, and when the input is a very small negative number, the output will approach 0. In RNN structure, it works like a switch to control the data flow.

- Tanh Function: The shape of the tanh function is similar to that of the sigmoid function. The difference is that the range of the tanh function is $[-1, 1]$, and it pasts the origin, which helps to avoid bias in the gradients. Therefore, it is often used as the activation function for the hidden layer.

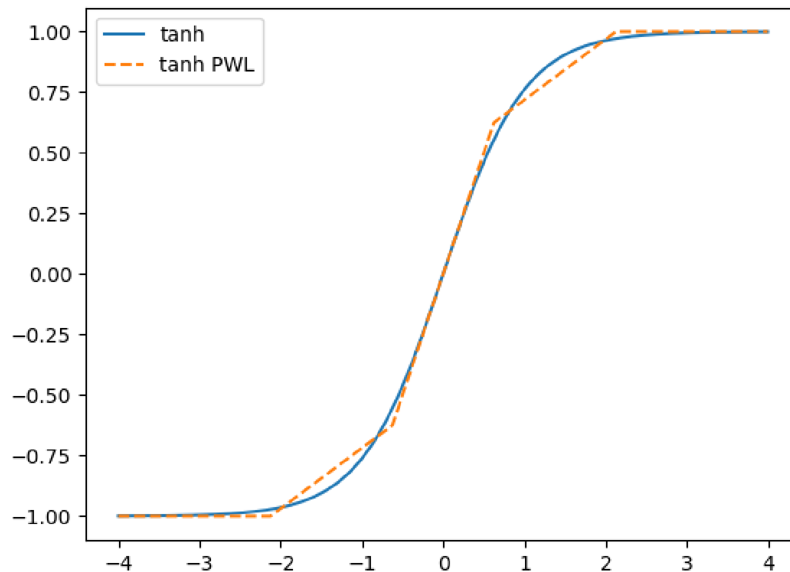


Fig. 4.8: Piecewise linear approximation of tanh function

There are several ways to realize activation functions, including piecewise linear (PWL) approximation in [31, 32], and LUTs in [33, 34]. The PWL approximation scheme uses a series of linear segments to approximate the function. The number and location of these segments are chosen to minimize errors, processing time, and area. This method usually requires several clock cycles and the use of an adder. Fig.

4.8 shows the PWL approximation of the tanh function with five line segments. The breakpoints and equations are listed in Table 4.2.

Table 4.2: Tanh approximation equations

Equations	**Conditions**
1	$x >= 2.125$
$0.625 * x$	$\lvert x \rvert <= 0.625$
$0.25 * x + 0.46875$	$0.625 < x < 2.125$
$0.25 * x - 0.46875$	$-2.125 < x < -0.625$
-1	$x <= -2.125$

We obtain a modified curve with the gradient of each linear segment expressed as a power of two. In this way, the multipliers can be replaced with shifters. The implementation of PWL approximation requires fewer resources than most other methods. However, it comes with the drawbacks of high absolute error. Hence it can not be used in scenarios that need high accuracy. The LUT implementation is more versatile. The output value of the activation function of a certain precision is pre-stored in the look-up table. The input of the look-up table is the input value of the activation function of a certain precision. The comparison of these two methods is shown in Fig. 4.9 with 10 bits in the decimal part. The error of the LUT method is smaller and more coherent.

To balance the speed, accuracy and area, we choose LUT for the activation functions. Since the sigmoid function has the nearly odd property and tanh is rotational symmetry about the origin:

$$\sigma(-x) = 1 - \sigma(x), tanh(-x) = -tanh(x). \tag{4.9}$$

Therefore, the size of the LUT can be reduced to half the desired space. Both the $\sigma(x)$ and $1 - \sigma(x)$ will be the output to the activation module since they are both in Eq. (4.8). The activation module has 4 stages as in Fig. 4.10.

- The first stage: cache the request command and data.
- The second stage: calculate the absolute value of the input.
- The third stage: send the data as input to LUT.

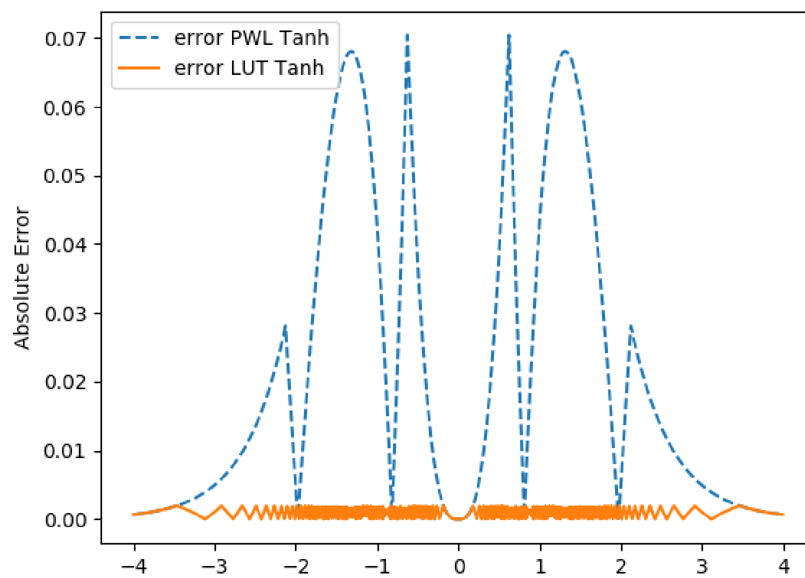


Fig. 4.9: Error introduced by PWL and LUT methods

- The fourth stage: reconstruct the data by Eq. (4.9) if the input is a negative number.

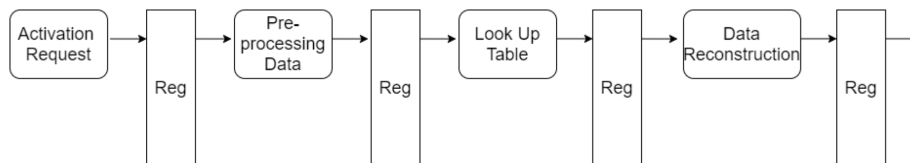


Fig. 4.10: Pipeline structure of the activation module

Since we might use activation modules for multi-layer RNN structures, there are chances where requests from different layers generate at the same time. Here we set the request from higher layers (close to the output) with higher priority. The request from lower layers will have to wait for one or more cycles. The timing pipeline for the conflict is shown in Fig. 4.11. When layer 1 and layer 2 modules request for activation at the same cycle, the request from layer 1 will be delayed for 1 cycle to avoid conflict. This is convenient and efficient since the 1 cycle has a small impact on the overall performance and the reuse of the activation module saves a lot of resources.

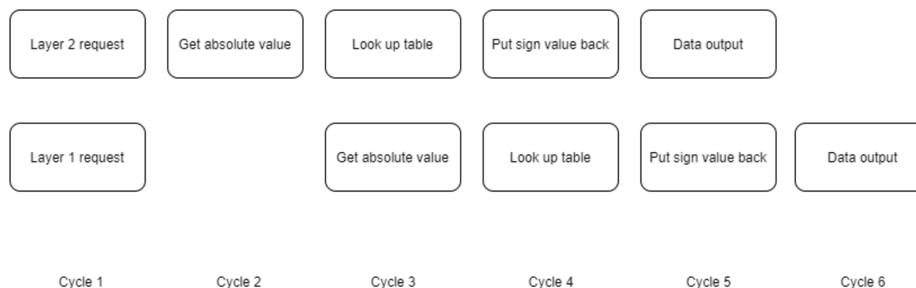


Fig. 4.11: Pipeline timing diagram of activation module

Moreover, the number of activation modules is configurable to pair with the number of DSP slices. One activation module can connect to any number of DPS slices. As illustrated in Fig. 4.12, one activation module can be connected to two DSP slices or four DSP slices. Because the activation module is pipelined, the four DSP slices connection only requires two more cycles comparing to the two DSP slices connection. The extra cycles can be ignored in most of the applications while the connection saves massive resources.

There are two options to synthesis the activation module on FPGA - using distributed RAM or block RAM. This is a great advantage for FPGAs with limited resources. RNNs with a large number of parameters require a great amount of block

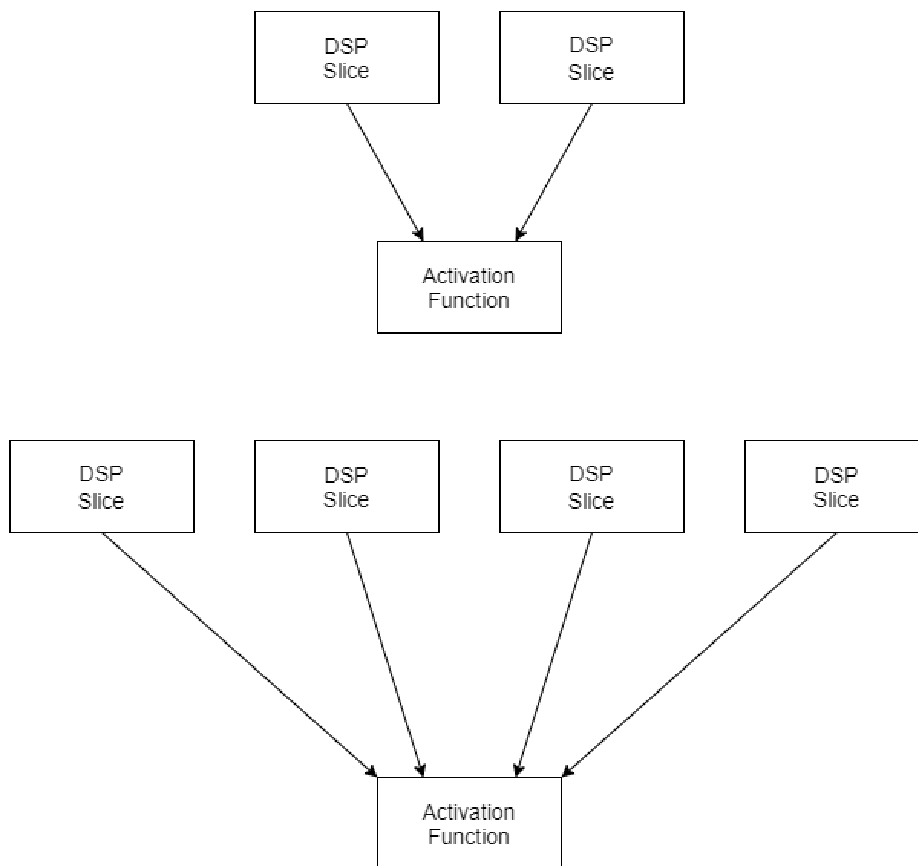


Fig. 4.12: Configurable pair of DSP slices and Activation modules

RAM, hence a synthesis option to use distributed RAM for the activation module can make possible a large number of activation functions when block RAM resources are running out. One example is illustrated in Table 4.3. We synthesis one activation function module which has 11 bits input and 11 bits output with different options. One uses the block RAM and the other uses the distributed RAM.

Table 4.3: Resource Utilization of activation module

Resource	**BRAM**	**DSP**	**FF**	**LUT**
Implemented with block RAM	1	0	93	138
Implemented with distributed RAM	0	0	113	276

4.3.5 Timing Schematic

The timing schematic is illustrated in Fig. 4.13. We use DSP48E1 for the calculation in each module, hence there will be a 3 cycle delay for each matrix multiplication. If we have a P dimension input vector, the calculation of xf_t or xh_t will take P+3 cycles. Similarly, if the MGU_1 has N hidden units, the calculation of f_t^* in Eq. (4.4) will take N+4 cycles. The extra cycle is used for the addition. For the forget module, it needs xf_t to finish the calculation. For the hidden state module, it needs f_t and xh_t to get the final hidden state h_t.

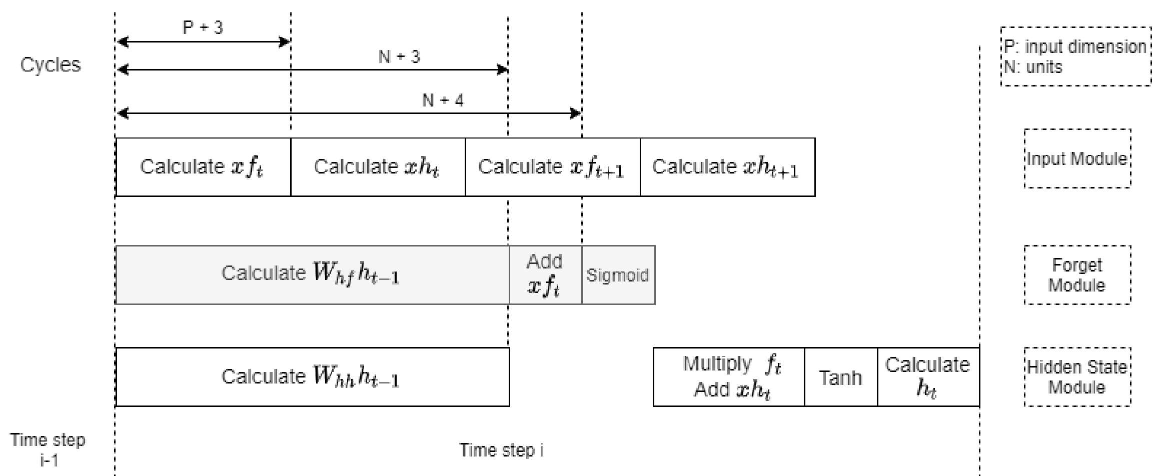


Fig. 4.13: Timing schematic for MGU_1

4.4 Implementation and Result

4.4.1 Dynamic fixed point

If we merely use the fixed-point number to quantize from the floating-point numbers, the result will not be good due to a huge loss. We adopt the dynamic fixed point in [35] for each calculation which keeps an acceptable inference accuracy comparing with the floating-point data format. In different modules, the outputs are the result of thousands of accumulations, thus the weights are much smaller than the module outputs. It is not possible to use fixed numbers to cover a wide dynamic range.

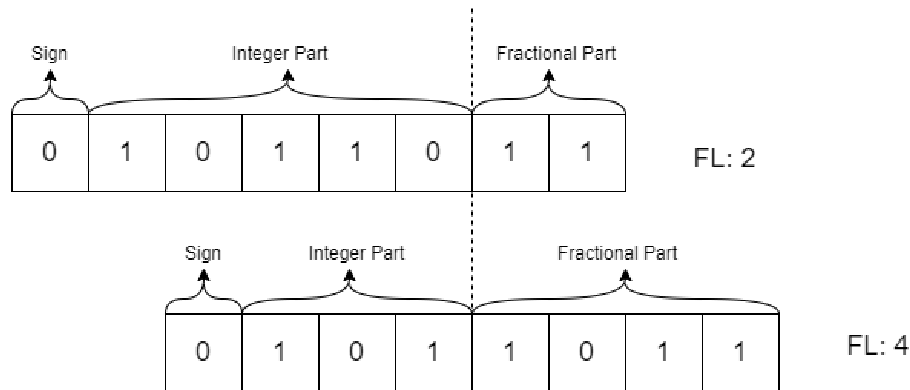


Fig. 4.14: Dynamic fixed point numbers

The data format can be seen in Fig. 4.14. In the structure, we have a single bit to denote the sign bit. Two variables bw and fl represent the bit width and the fractional part length. In this way, the weights and the output in each module can have different fl values to achieve a high dynamic range. Each number can be represented as

$$(-1)^s \times 2^{-fl} \times \sum_{i=0}^{bw-2} 2^i x_i, \tag{4.10}$$

where x denotes the mantissa bits. The intermediate values in a model have different ranges. Each network layer is split into two groups: one for the layer outputs, one for the layer weights. This allows us to better cover the dynamic range of both layer outputs and weights, as weights are normally significantly smaller. On the FPGA side, it is possible to implement dynamic fixed-point arithmetic by using bit truncation.

We can quantize data from floating-point numbers to fixed point numbers by the equation as

$$round(rd \times 2^{fl}), \tag{4.11}$$

where rd denotes real data before quantization. The precision loss is less than 2^{-fl}. One of the problems left is to determine the fl. We can solve the problem by using the equation:

$$fl = bw - ceiling(log_2(max(abs(data)) + 1)) - 1, \tag{4.12}$$

where the ceiling function maps input value x to the least integer greater than or equal to x.

4.4.2 Implementation

The Zybo Z7-20 FPGA board includes software programmability based on embedded processors and hardware programmability of FPGAs. This technology allows us to perform critical analysis and hardware acceleration when integrating CPU, DSP, and mixed-signal functions on a single device. We use RTL to design the structure. Although there are high-level synthesis tools to design the structure, we still choose the traditional RTL language to implement the accelerator to achieve high performance and area efficiency.

The neural network model used is given in Fig. 2.5. It has two layers of MGU_1 and each has 32 units. The input dimension is 13. The model is trained on GPU and weights initial file (COE) is generated by Python. Since our architecture is scalable, we can extend the structure in Figure 4.2 to have 2 layers by adding another input module, forget module, and hidden state module. We will only use one activation module for two layers because it takes up a lot of areas. Moreover, our pipeline structure for the activation module is efficient. If both layers request for activation, the first layer's request will be delayed for one clock cycle which can be ignored considering the whole processing time. The LUT in the activation function has an 11-bit data input and 11-bit data output.

The direct memory access (DMA) controller is in charge of data movement. The DMA controller copies data from one address space to another address space and provides high-speed data transfer without using the CPU which is quite weak in FPGA. The DMA controller uses interrupts to handle the data without using the CPU which is quite weak in FPGA. The DMA controller in XC7Z020 consists of an

instruction acceleration engine, an AXI Master data interface, an AXI APB register access interface, a peripheral request interface that can be connected to the PL, a data buffer FIFO, and a control and status generation unit. The DMA controller has eight channels, four of which are responsible for data handling on the central interconnect storage unit; the other four data channels are for peripheral requests. Each DMA channel executes its instructions and has its independent thread, which does not affect each other. The instruction execution engine has its independent cache line.

The software flowchart is illustrated in Fig. 4.15. Two DMA channels are used. One is for moving data from dynamic random-access memory (DRAM) to MGU_1 accelerator's data buffer. Another is used to move data from the data buffer to DRAM. The start location is stored in register in MGU_1 module as the ground truth of the starting point. Interrupt is used to communicate between the CPU and MGU_1 accelerator.

4.4.3 Implementation Result

The MGU_1 accelerator is running at 142 MHz with power consumption at 2.471 W. The whole model has 7,170 parameters. The resource utilization is presented in Table 4.4. The input value, weights, and bias are all int16 format. The number of the time steps for the test is 9 and the total predict location is 171 in the test data set. The first input location of the first time step bunch is the ground truth location. Then the output of the ninth predicted location will output to the next bunch of time steps as the ground truth and so on and so forth.

Table 4.4: Resource utilization of the accelerator

Resource	**BRAM**	**DSP**	**Flip-flop**	**LUT**
Used	54.5	208	25164	20230
Total	140	220	106400	53200
Utilization	38.9%	94.5%	23.6%	38.0%

The test data is a square-shaped corridor shown in Fig. 4.16. The predicted data is sent from FPGA to a computer through the UART port. The predicted trajectory is close to the ground truth. There are some fluctuations in the corner area but the model can correct itself.

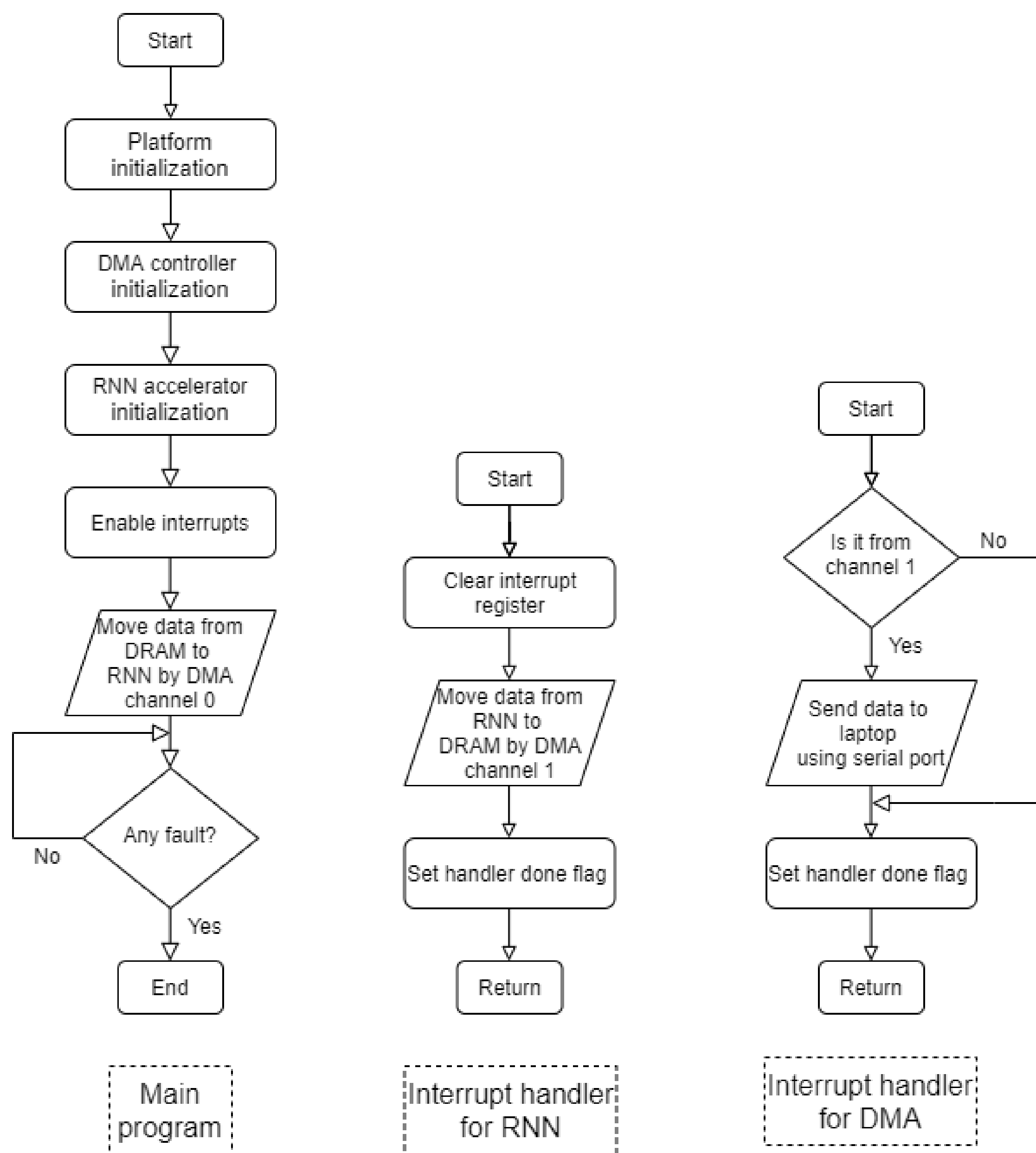


Fig. 4.15: Illustration of the program flowchart

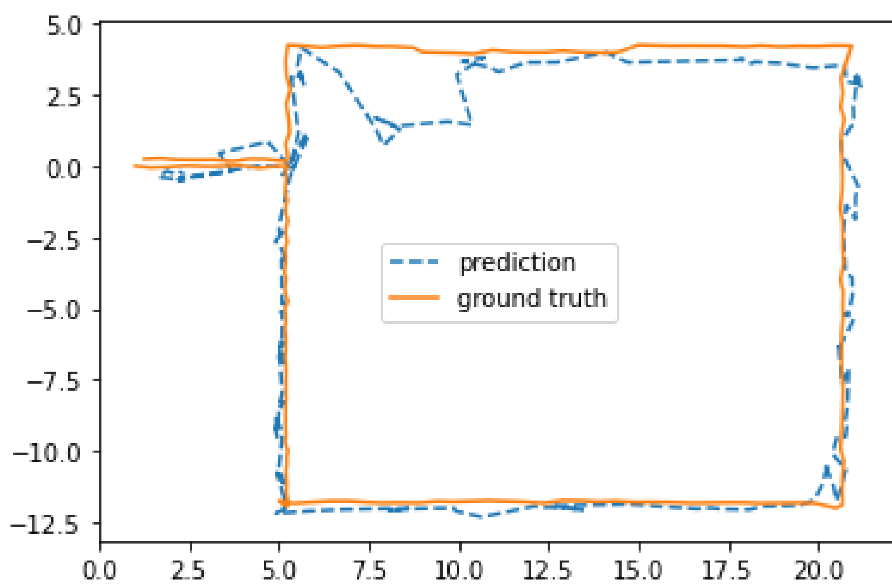


Fig. 4.16: The ground truth trajectory and the predicted trajectory

We compare the result of XC7Z020 with Intel i5-5350U in Table 4.5. The latency is based on the time used for calculating 9 time steps. The code running on the Intel i5-5350U platform is written in Python. The XC7Z020 is faster than i5-5350U by two orders of magnitude. Although the data format is different, the accuracy is close between the two platforms.

Table 4.5: Performance Comparison

Platform	**Format**	**Latency (us)**	**Average Error (m)**
i5-5350U	float64	1344	0.97±0.78
XC7Z020	int16	6	0.96±0.78

Chapter 5

Conclusions and Future Work

In this report, we have proposed a modified model of the MGU. We optimize the structure of the MGU and experiment on 4 data sets. The result is consistent with our expectations. The accuracy of the modified model MGU_1 is at the same level as the GRU and outperforms the MGU model while the MGU_1 model is smaller compared to the GRU model. Then we implement an accelerator on a Xilinx XC7Z020 FPGA based on the MGU_1 model. This accelerator's structure is scalable and flexible and can be used on both simple and complex networks. Besides, we use the LUT method rather than PWL method to realize the activation module to achieve better accuracy. The number of this activation module in the accelerator can be configured to get a performance/resource balance. Moreover, since FPGA is running at a low clock speed, the whole system has a low power consumption. The comparison results show that our design outperforms software implementations by a huge amount, and it achieves similar accuracy. Overall, this project has achieved our two main goals:

- Find a simple RNN model with high accuracy
- Design a scalable and low power consumption architecture on FPGA

There are several opportunities for further research, such as to combine the MGU and GRU structure. Since in [16] the cross-correlation is not 1, by adding 10% independent parameters to forget gate rather than using the same parameter value in the hidden state calculation, the performance may improve in more complicated scenarios. Also, the pruning method in [26] could be implemented to maximize performance and efficiency. In hardware design, model compression helps saving the computation

as well as the memory footprint, which means lower latency and better energy efficiency. In some practical cases, 90% of the parameters can be pruned away without hurting word error rate [26]. To implement this pruned structure, we need the adaptive hardware to implement a special data format and a corresponding computational flow which is a specialty of FPGA.

Bibliography

[1] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[2] Zybo Z7 reference manual. `https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual`. Accessed: 2020-11-20.

[3] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[4] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861 – 867, 1993.

[5] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2, 1999.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112. Curran Associates, Inc., 2014.

[7] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.

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

[9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[10] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. *Journal of Machine Learning Research*, 2015.

[11] Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234, 2016.

[12] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning: with applications in R*. Springer, 2013.

[13] Ian oodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[15] Justin Simon Bayer. *Learning sequence representations*. Dissertation, Technische Universität München, München, 2015.

[16] M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio. Light gated recurrent units for speech recognition. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2):92–102, 2018.

[17] C. Liu, D. Fang, Z. Yang, H. Jiang, X. Chen, W. Wang, T. Xing, and L. Cai. Rss distribution-based passive localization and its application in sensor networks. *IEEE Transactions on Wireless Communications*, 15(4):2883–2895, 2016.

[18] M. T. Hoang, B. Yuen, X. Dong, T. Lu, R. Westendorp, and K. Reddy. Recurrent neural networks for accurate rssi indoor localization. *IEEE Internet of Things Journal*, 6(6):10639–10651, 2019.

[19] M. T. Hoang, Y. Zhu, B. Yuen, T. Reese, X. Dong, T. Lu, R. Westendorp, and M. Xie. A soft range limited k-nearest neighbors algorithm for indoor localization enhancement. *IEEE Sensors Journal*, 18(24):10208–10216, 2018.

[20] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[21] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[22] Alex Graves, Santiago Fernández, Faustino J. Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In William W. Cohen and Andrew W. Moore, editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 369–376. ACM, 2006.

[23] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, June 1993.

[24] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[25] Shaoshan Liu, Jie Tang, Chao Wang, Quan Wang, and Jean-Luc Gaudiot. Implementing a cloud platform for autonomous driving. *arXiv preprint arXiv:1704.02696*, 2017.

[26] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. Ese: Efficient speech recognition engine with sparse lstm on fpga. In Jonathan W. Greene and Jason Helge Anderson, editors, *FPGA*, pages 75–84. ACM, 2017.

[27] Zynq technical reference manual. `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. Accessed: 2020-11-20.

[28] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu. FPGA acceleration of recurrent neural network based language model. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 111–118, 2015.

[29] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on fpga. *arXiv preprint arXiv:1511.05552*, 2015.

[30] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram. Polar: A pipelined/overlapped fpga-based lstm accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(3):838–842, 2020.

[31] H. Amin, K. M. Curtis, and B. R. Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings - Circuits, Devices and Systems*, 144(6):313–317, 1997.

[32] Alin Tisan, Stefan Oniga, Daniel Mic, and Attila Buchman. Digital implementation of the sigmoid function for fpga circuits. *Acta Technica Napocensis Electronics and Telecommunications*, 50(2):6, 2009.

[33] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi. Efficient hardware implementation of the hyperbolic tangent sigmoid function. In *2009 IEEE International Symposium on Circuits and Systems*, pages 2117–2120, 2009.

[34] K. Leboeuf, A. H. Namin, R. Muscedere, H. Wu, and M. Ahmadi. High speed vlsi implementation of the hyperbolic tangent sigmoid function. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, volume 1, pages 1070–1073, 2008.

[35] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2015.