FPGA Deployment of LFADS for Real-time Neuroscience Experiments

Xiaohan Liu

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2023

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

FPGA Deployment of LFADS for Real-time Neuroscience Experiments

Xiaohan Liu

Chair of the Supervisory Committee:
Dr. Scott Hauck
Department of Electrical and Computer Engineering

Neural networks have been widely used in neuroscience experiments to model and analyze neural activities. Sleep spindle, a rare brain signal, is considered to be associated with learning and memory. Currently, a complex neural network model named Multi-block RNN Autoencoders (MREA) has shown satisfactory performance in reconstructing the brain signals and the possibility of revealing the unclear mechanism of how sleep spindles contribute to learning and memory. To develop a real-time system for analyzing sleep spindles, the Field Programmable Gate Array (FPGA) was used to accelerate the model. Because of the substantial size of the MREA, we initially deployed its baseline model, Latent Factor Analysis via Dynamical Systems (LFADS), onto FPGA. The model was translated to hardware descriptions by HLS4ML (High-Level Synthesis for Machine Learning), a framework that converts the traditional machine learning models to HLS models. We deployed the LFADS onto Xilinx U55C by modifying its architecture and implementing the HLS4ML package. The modification of the LFADS architecture, implementation of the HLS4ML, and the on-board performance are discussed in this thesis.

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

In recent years, the interest in utilizing artificial intelligence in a variety of research fields has been increasing. Neural networks have also been widely used to analyze neural signals [1]. Sleep spindle, a particular brain signal, captures the attention of neuroscientists. The sleep spindles are transient low frequency (roughly around 10-15Hz) rare signals that primarily occur during sleep [2]. Recent studies have shown that sleep spindles have demonstrated a co-modulatory relationship with other non-rapid eye movement (NREM) oscillations, such as hippocampal sharp wave-ripples and neocortical slow waves [3]. Both of these two oscillations are associated with learning and memory. Nevertheless, there is still an incomplete understanding of how sleep spindles contribute to learning and memory.

A machine learning model, multi-block RNN autoencoders (MRAE), has shown a remarkable ability to process neural data collected from primate brains [4]. Moreover, MRAE can be further applied to analyze sleep spindles. To analyze sleep spindles with low latency, Field Programmable Gate Arrays (FPGAs) will be used to accelerate the algorithms. The model will be pushed through a framework named HLS4ML [5], a Python package that automatically translates the machine learning models to hardware descriptions. These translated hardware descriptions are applicable for the FPGA deployment. However, due to the complexity of the MRAE architecture, we decided to first deploy the baseline model of MRAE, latent factor analysis via dynamical systems (LFADS), onto FPGA.

This thesis includes the introduction of MREA and LFADS, the advantages of using FPGA to accelerate LFADS, the modifications of LFADS architecture, the implementation of HLS4ML, and the on-board performance of LFADS. Since this work is the first step in analyzing sleep spindles, several future works are also mentioned at the end.

Chapter 2. MRAE AND LFADS

In this chapter, the target model MRAE and its baseline model LFADS will be introduced. Both the MRAE and LFADS follow the variational autoencoder (VAE) architecture. Therefore, it is necessary to understand the VAE and its baseline architecture - autoencoder (AE) first.

2.1 AUTOENCODER AND VARIATIONAL AUTOENCODER

AE is a distinctive type of deep learning architecture. It is designed to compress the inputs to the latent representation and then decode from the latent representation to reconstruct its inputs [6]. Figure 1 illustrates the structure of an AE.

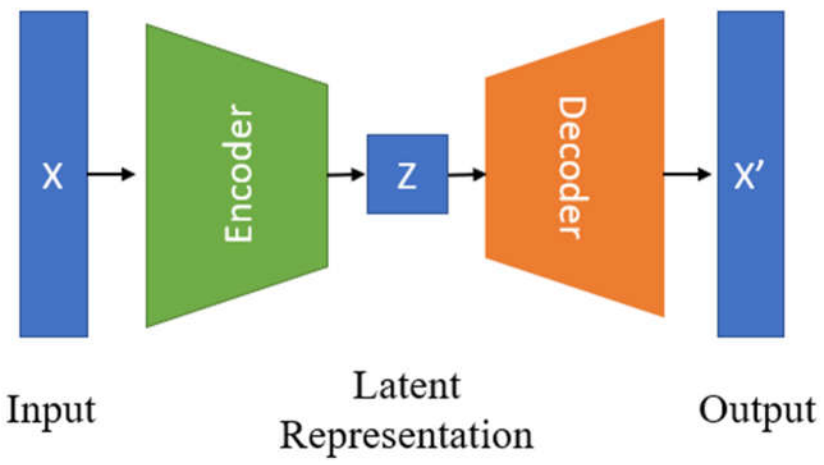


Figure 1. Autoencoder Architecture.

[6] and [7] discuss the architecture of a typical autoencoder.

Encoder: The encoder takes the input X and maps it to a lower-dimensional representation Z through the transformations within its hidden layers.

Latent Representation: The latent space representation Z contains the compressed information of the input X.

Decoder: The decoder mirrors the function of the encoder in reverse. It takes the latent space representation Z and reconstructs the original inputs based on this compressed information. The AE is trained such that the reconstructed output X' is as close to X as possible.

AE has been commonly used in various fields, such as classification, clustering, and anomaly detection [8]. There are various forms of AE, including convolutional AE [9], recurrent AE [10], or MLP AE [11]. Therefore, the implementation of AE is not restricted to the type of neural network layers.

VAE also follows the encoder-decoder structure, but it provides a way to describe the latent space representation using probabilities [12]. The VAE architecture is diagrammed in Figure 2.

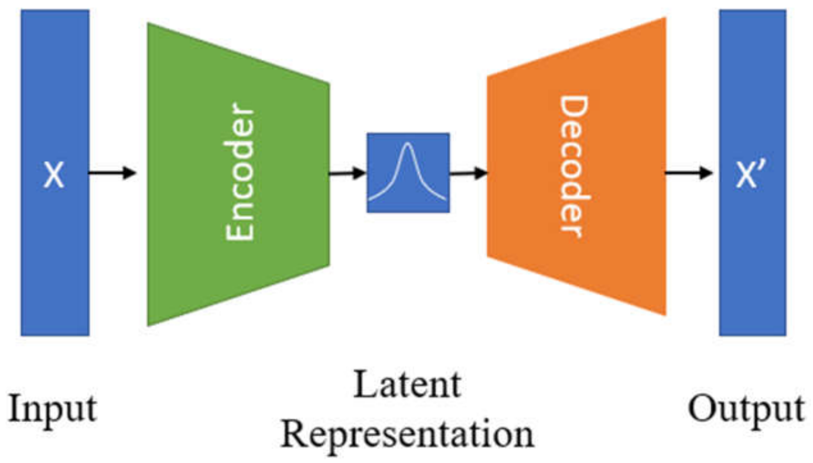


Figure 2. Variational Autoencoder Architecture.

VAE enhances the representation capabilities of traditional AE [8]. Compared to AE, VAE encodes the input X to parameters that can define the latent distribution and then decodes from the samples in this distribution to reconstruct its original input [8]. In our case, both MRAE and LFADS are based on the VAE architecture.

2.2 Multi-Block RNN Autoencoders - MRAE

MRAE was designed by Nolan et al. [4] in 2022. It is designed to directly reconstruct the broadband time-series micro-Electrocorticography (μECoG) data. As its name suggests, the MRAE contains multiple blocks of RNN autoencoder (RAE). The architecture of a single-unit RAE is presented in Figure 3.

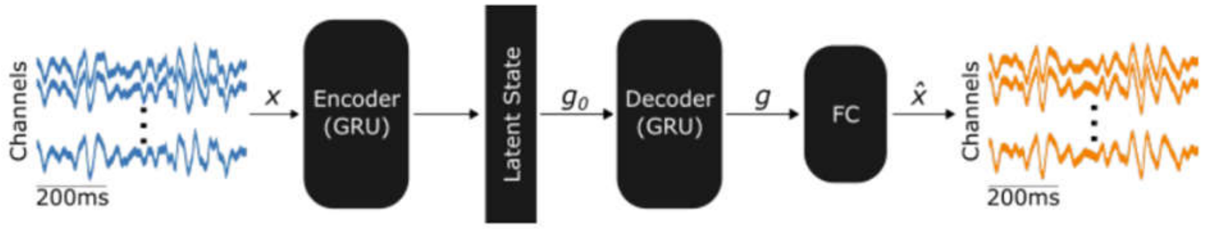


Figure 3. Single-unit RAE Architecture [4].

The single-unit RAE follows the VAE architecture. The RAE encoder and decoder are based on the GRU cell, which is a type of RNN. Different types of RNNs are shown in Figure 4.

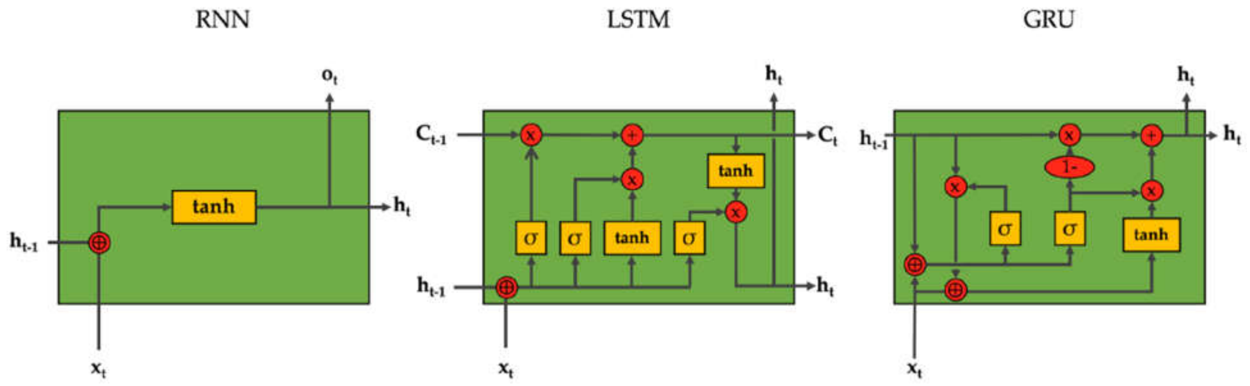


Figure 4. Vanilla RNN, LSTM, GRU Architecture [13].

Compared to a vanilla RNN, GRU has two gates, a reset gate and an update gate, to regulate the computation, which helps to solve the vanishing gradient problem that the vanilla RNN will encounter [14]. Apart from vanilla RNN and GRU, there is also another type of RNN called LSTM. LSTM has more parameters to regulate the computation than GRU. See [14] for a more complete explanation.

Moreover, the encoder of RAE is a bidirectional GRU, which combines two GRU to process the input data in both forward and backward directions. The details of the bidirectional GRU are discussed in Chapter 5.1.

[4] also demonstrated that in the single-unit RAE, the input μECoG signals are passed into the bidirectional GRU encoder, and the encoder estimates the latent distribution parameters. From this distribution, the latent state values are sampled. Then, these values initialize the initial state of the unidirectional decoder GRU. After that, a fully connected layer (FC) processes the output from the decoder to reconstruct the entire input. However, they evidenced that a single-unit RAE was insufficient to precisely reconstruct the broadband signals. Therefore, the MRAE, which is extended from RAE, was proposed. Figure 5 shows the MREA architecture.

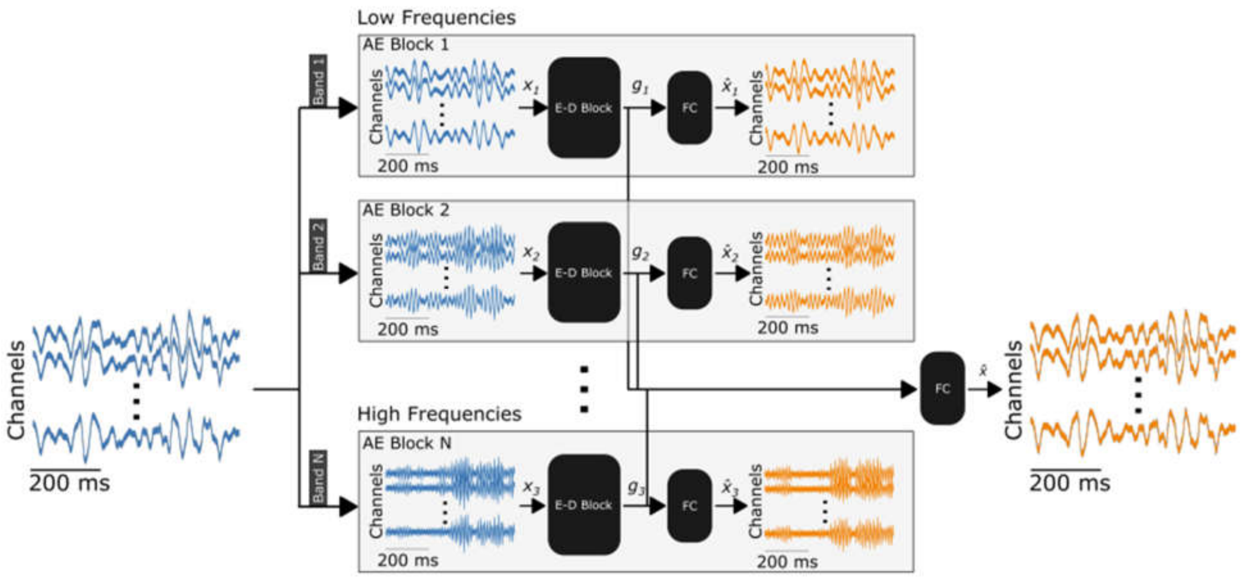


Figure 5. MRAE Architecture [4].

[4] emphasizes that in MRAE, multiple single-unit RAE are grouped into an independent block, and several such blocks are parallelized within MRAE. The input signals are filtered into non-overlapping frequency bands and then processed by independent blocks. The output from the decoder in each block is concatenated and projected by the last fully connected layer to reconstruct the entire input. By increasing both the number of single-unit RAE in each block and the total number of blocks in MRAE, the reconstruction accuracy has increased dramatically. The MRAE with a total of 3 blocks, each containing 512 single-unit RAE, shows the best reconstruction accuracy. See [4] for a complete discussion of the architecture and performance of MRAE.

As mentioned above, MRAE is the candidate for analyzing sleep spindles. However, due to the extremely large size of MRAE, it is hard to be deployed onto FPGA. Its complex architecture will require a large amount of hardware resources to support the inference. Therefore, the baseline model, LFADS, is chosen to be the target of the initial FPGA deployment. LFADS has been developed for years and has shown promising results in modeling brain activities. Furthermore, it is considerably small compared to MRAE, which makes it an ideal candidate for FPGA deployment.

2.3 Latent Factor Analysis via Dynamical Systems - LFADS

Sussillo et al. [15] developed LFADS in 2016. They demonstrated that the LFADS infers smooth dynamics based on the collected neural spiking data to provide a promising way to model complex brain activities.

[15] illustrated that the complexity of the brain presents extreme challenges in developing a universal algorithm to analyze brain signals, and the two most used approaches are feed-forward processing and sequential processing. Feed-forward processing is completely input driven and is

not affected by temporal dynamics. In contrast, sequential processing is mainly based on dynamics. The "computational dynamics" are calculated from the nonlinear system that starts from a specific initial condition to run for a period. Here, temporal dynamics refers to the changes and patterns of neuronal activities that occur in the brain over time, and "computational dynamics" are the stimulations of brain dynamics. These two approaches focus on different aspects and are generally not used together. However, LFADS combines both sequential processing and external input, which takes advantage of both methods in processing complex neural data.

2.3.1 *LFADS Architecture and Basics*

LFADS follows the VAE architecture. A detailed discussion of LFADS architecture is in [15] and [16]. Figure 6 presents the overview of the LFADS architecture.

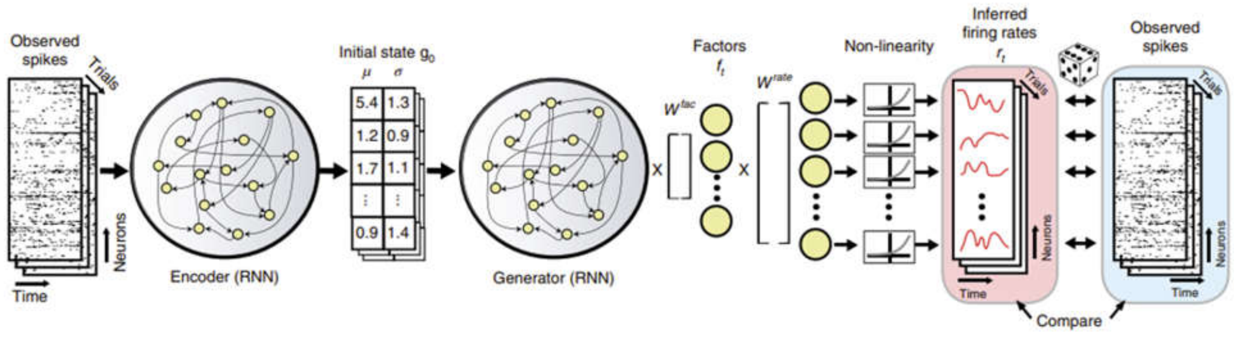


Figure 6. LFADS Architecture [16].

The LFADS model used in this work is sourced from Hurwitz et al. [17]. They reimplemented the original LFADS that is discussed in [15, 16] in Tensorflow2 Keras API. In the following Chapters, "Hurwitz et al.'s LFADS" refers to their reimplemented LFADS. The architecture of Hurwitz et al.'s LFADS is the same as the one shown in Figure 6.

[15] and [16] summarize that the bidirectional GRU encoder compresses the input spiking data to produce a set of mean (μ) and variance (σ). The mean and variance are used to define the Gaussian

distribution. Then the samples from the distribution are used to initialize the initial condition of the unidirectional GRU decoder. Based on this initial condition, the decoder starts to run and produces a set of low-dimensional temporal factors denoted as $\mathbf{f}_t$. These factors are believed to contain the information of the input spiking data. After that, a fully connected layer takes the factors as input and outputs the log firing rates. Finally, the log firing rates are passed into an exponential function to generate the inferred firing rates, $\mathbf{r}_t$, of the input spiking data.

Another notable detail is that the original LFADS has a controller to generate the inferred input for the decoder [16]. Figure 7 shows the LFADS architecture with the controller. However, as discussed in [15], the decoder can also receive no input. In Hurwitz et al.'s LFADS, the controller is not implemented, and the input to the decoder is zero, which is also the architecture used in this work.

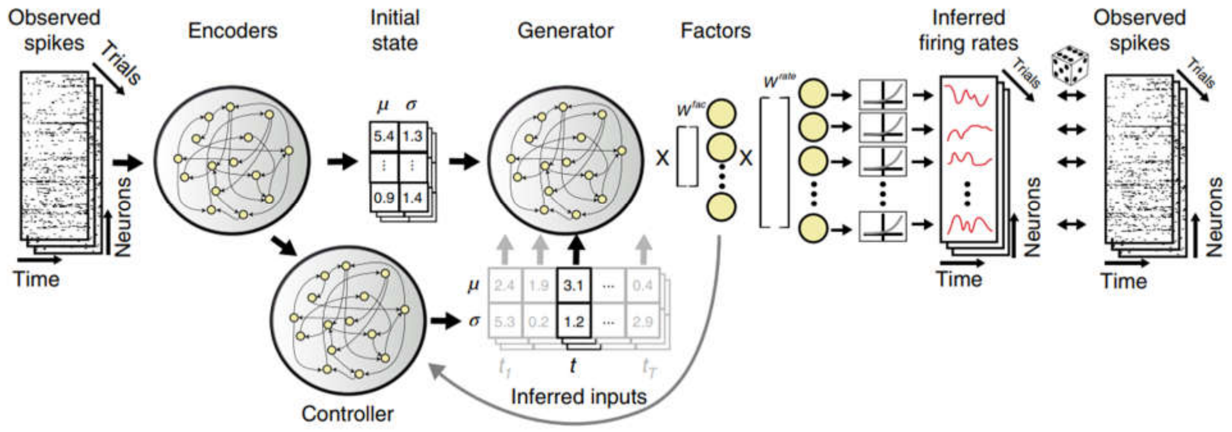


Figure 7. LFADS Architecture with Controller [16].

2.4 Scripts, Dataset, and Evaluation Metrics

Apart from the LFADS model, the scripts that are used to train and evaluate the model are also from Hurwitz et al. [17]. Besides, the dataset used for training, validation, and testing is collected from a monkey reaching experiment [18]. In this work, only a small portion of the entire dataset was used, which only contains a total of 176 trials. We used 136 trials for the training, and the remaining 34 trials were used for validation and testing. Each trial contains information from 70 recording channels, and the data in each channel were sampled into 73 discrete time steps.

Two metrics were used for evaluating LFADS performance in this work. The first one is the negative Poisson log-likelihood, which is also the loss function of the LFADS training. Since the input spiking data is assumed to follow the Poisson process, LFADS minimizes the negative Poisson log-likelihood to reach optimal performance during the training [15]. See [15] for a more detailed discussion. The negative Poisson log-likelihood is calculated by passing both the testing dataset and the generated log firing rates to a Poisson loss function in TensorFlow. One point to note is that the goal of this work is to deploy LFADS onto FPGA, not to optimize LFADS performance. Thus, the performance of Hurwitz et al.'s LFADS can be used as the ground truth. In the following work, we compared the negative Poisson log-likelihood of the modified model with the ground truth to evaluate the performance of the modified model.

The second evaluation metric is the coefficient of determination (R^2). This is calculated by fitting the LFADS factors to the behavioral data. See [17] for a comprehensive discussion. Again, the R^2 score from Hurwitz et al.'s LFADS was used as the ground truth in this work.

Chapter 3. INFERENCE ON HARDWARE

In this chapter, we will first introduce FPGAs and then look at the advantages of using FPGAs to accelerate inference. Finally, we will introduce HLS4ML [5], which is the framework used for translating machine learning models to hardware languages.

3.1 Field-Programmable Gate Arrays – FPGAs

FPGAs are a type of semiconductor device that can be programmed to utilize in different use cases, which include the implementation of high parallelism computation of neural networks [19]. In recent years, FPGAs stand as a prominent choice for accelerating neural network algorithms [20]. When processing a large neural network algorithm, FPGAs typically provide lower latency and higher throughput compared to a typical CPU [21]. On the other hand, FPGA is typically more energy-efficient than a modern GPU for these tasks [19, 21]. Although application specific integrated circuits (ASICs) are more energy efficient than FPGA in processing neural network algorithms, they cost more to produce and take a much longer time to develop [20]. All the above advantages make FPGA a useful device for accelerating neural network inference.

3.2 LFADS Inference on FPGA - Advantages

Since FPGA provides the opportunity of realizing low-latency, low-power, and high throughput model inference, it creates the possibility of developing large-scale real-time experiments. Although deploying LFADS onto FPGA is the first step in interacting with sleep spindles, there are still plenty of other applications for low-latency and high-throughput LFADS. For example, low-latency LFADS can be used to design real-time closed-loop experiments to decode neural activity to control external devices. On top of that, in neural-related experiments, high throughput ML increases the ability to analyze large-scale neural recordings. Experiments that are being done with large-scale neural recordings are becoming more and more common in novel neuroscience experiments [22].

3.3 High-Level Synthesis for Machine Learning (HLS4ML)

HLS4ML is a Python package that translates machine learning models to HLS models, and the translated code is applicable for FPGA deployment [5]. The workflow of HLS4ML is depicted in Figure 8 below.

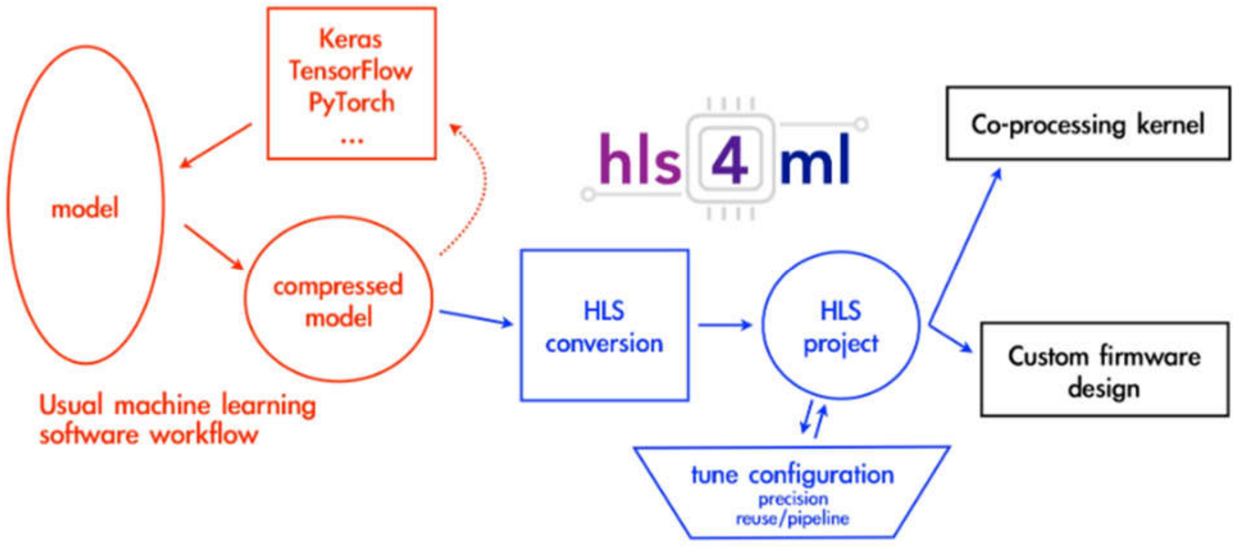


Figure 8. HLS4ML Workflow [5].

[5] indicates that the orange section is the pre-HLS4ML processing, which is to train and tune the model to reach a satisfactory performance in machine learning frameworks, for example, Keras or PyTorch. The blue section represents the flow of the HLS4ML package. HLS4ML scans the trained machine learning model and translates it into the HLS project. The HLS project is synthesizable and can be implemented to deploy onto an FPGA. Besides, HLS4ML also provides the opportunity of tuning multiple parameters to optimize the HLS model performance.

Chapter 4. LFADS MODIFICATION FOR DEPLOYMENT

To push LFADS through HLS4ML and deploy it onto FPGA, several modifications were necessary for the LFADS architecture. As mentioned, HLS4ML can accept models implemented in limited ML frameworks such as Keras or PyTorch. On top of that, all the layers in the model must be the framework built-in layers, for example, Keras.layers.Layer instance, and have been implemented in HLS4ML. Custom models or layers are not supported. Only when a model fulfills all the requirements above can it push through the HLS4ML workflow and generate the hardware descriptions.

To make Hurwitz et al.'s LFADS fulfill the requirements, we removed the custom Gaussian sampling layer from the model and reimplemented it in Keras functional API. We evaluated the performance of the reimplemented model based on the same dataset and proved that the performance is nearly identical to the performance of Hurwitz et al.'s LFADS.

4.1 GAUSSIAN SAMPLING REMOVAL

The Gaussian sampling layer is the only custom layer in Hurwitz et al.'s LFADS. Since implementing a custom layer in HLS4ML will take longer time than implementing a Keras layer, we separated it from LFADS and started a parallel project to implement Gaussian sampling in HLS4ML. Once the Gaussian sampling is implemented in HLS4ML, it will be combined with this work.

As discussed in Chapter 2.3.1, the mean and variance are used to generate the Gaussian samples. Then the Gaussian samples initialize the initial condition of the GRU decoder to generate the decoder output. To remove the Gaussian sampling from LFADS, both the Keras layer for generating the variance and the custom layer for generating Gaussian samples were removed. In this case, the mean values were used to initialize the initial condition of the GRU decoder in the no-Gaussian LFADS.

4.1.1 *No-sampling LFADS Performance*

Next, we trained, validated, and evaluated the no-gaussian LFADS based on the same dataset. The performance comparison is shown in Figure 9.

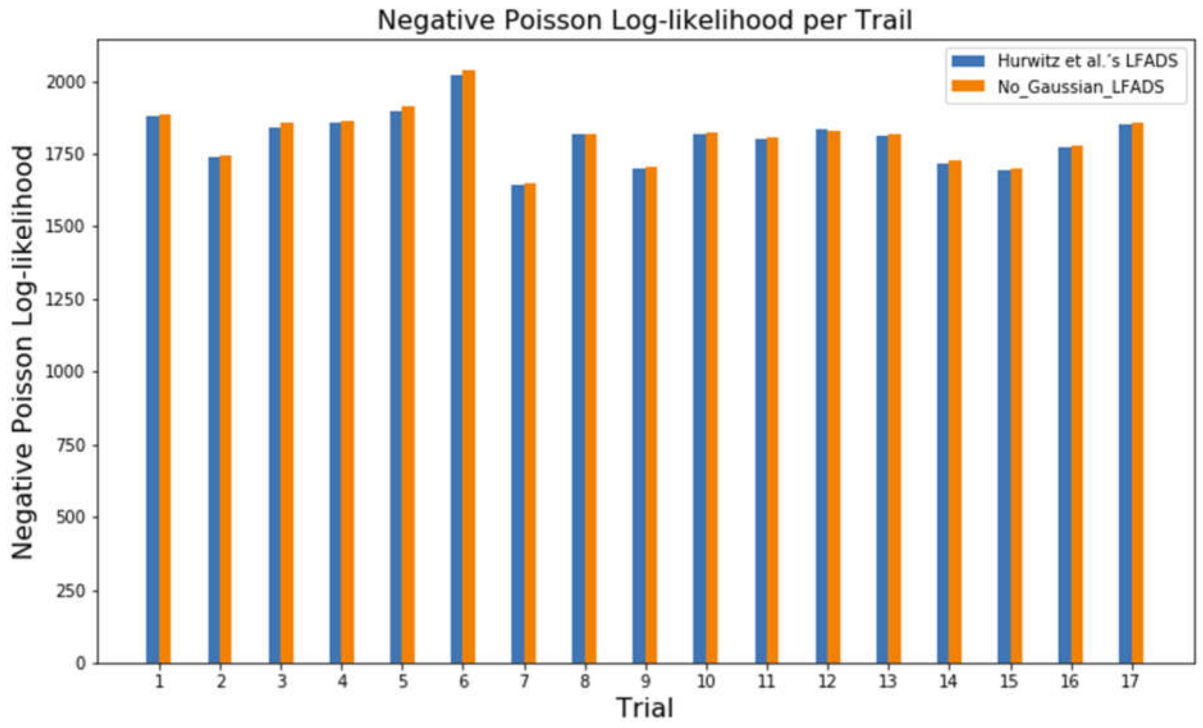
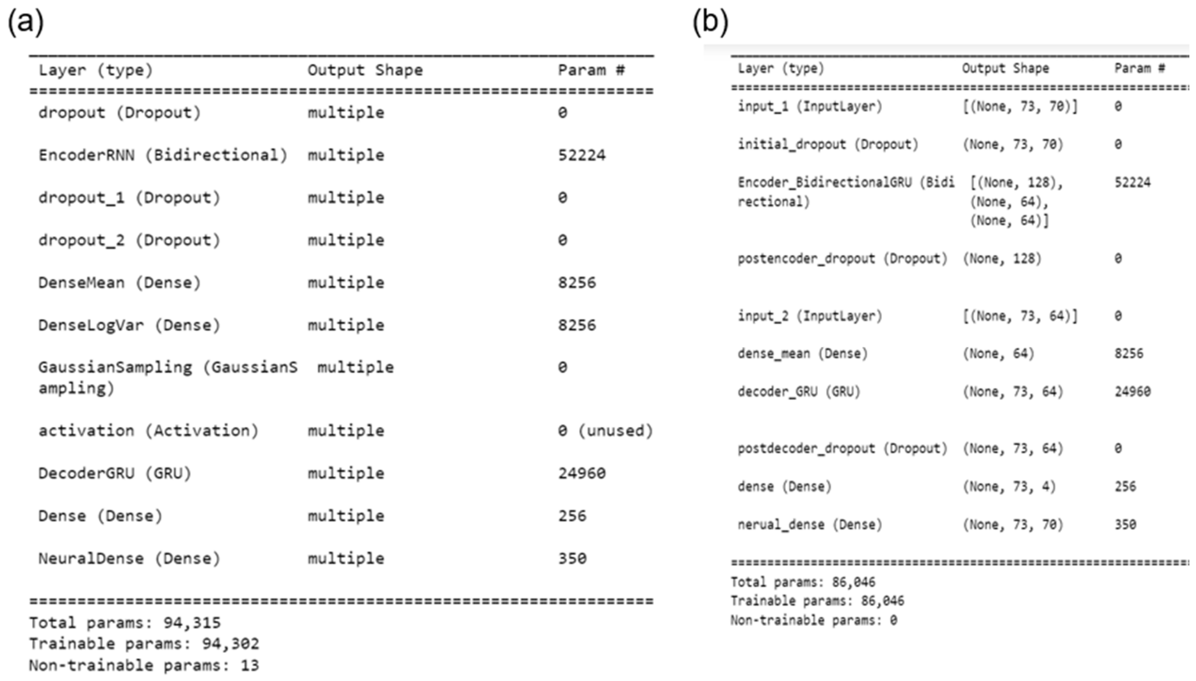


Figure 9. Hurwitz et al.'s LFADS Performance vs No-Gaussian LFADS Performance.

There are a total of 17 trials in the testing dataset. For each trial, the negative Poisson log-likelihood was calculated and compared with the ground truth, which is the negative Poisson log-likelihood of the optimized LFADS. The blue bar and orange bar indicate the performance of Hurwitz et al.'s LFADS and the no-Gaussian LFADS. For all 17 trials, these two models perform nearly identical. Besides, the R^2 score from Hurwitz et al.'s LFADS is 0.90. The no-Gaussian LFADS has a 0.87 R^2 score, which is still in the reasonable range. These results indicate that removing Gaussian sampling from LFADS does not significantly affect the model performance. It is a reasonable approach to accelerate the process of deploying LFADS onto FPGA.

4.2 NO-SAMPLING LFADS IN KERAS FUNCTIONAL API

HLS4ML accepts the model that is implemented in Keras API, but the model has to be based on either Keras functional or sequential API. Hurwitz et al.'s LFADS was implemented on neither of these two. The internal connections are custom defined. Therefore, a reimplementation in either Keras sequential or functional API was necessary. Since LFADS has multiple inputs and outputs, the functional API is a better choice for the reimplementation. We followed the architecture of Hurwitz et al.'s LFADS to reimplement the model except for the layers that are related to the Gaussian sampling. Figure 10 shows the model summary of Hurwitz et al.'s LFADS, as well as the model summary of our reimplemented model.



(a)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dropout (Dropout)            multiple                  0

EncoderRNN (Bidirectional)   multiple                  52224

dropout_1 (Dropout)          multiple                  0

dropout_2 (Dropout)          multiple                  0

DenseMean (Dense)            multiple                  8256

DenseLogVar (Dense)          multiple                  8256

GaussianSampling (GaussianS  multiple                  0
ampling)

activation (Activation)      multiple                  0 (unused)

DecoderGRU (GRU)             multiple                  24960

Dense (Dense)                multiple                  256

NeuralDense (Dense)          multiple                  350

=================================================================
Total params: 94,315
Trainable params: 94,302
Non-trainable params: 13
```

(b)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 73, 70)]          0

initial_dropout (Dropout)    (None, 73, 70)            0

Encoder_BidirectionalGRU (Bidi  [(None, 128),          52224
rectional)                    (None, 64),
                              (None, 64)]

postencoder_dropout (Dropout)  (None, 128)             0


input_2 (InputLayer)         [(None, 73, 64)]          0

dense_mean (Dense)           (None, 64)                8256

decoder_GRU (GRU)            (None, 73, 64)            24960


postdecoder_dropout (Dropout)  (None, 73, 64)          0

dense (Dense)                (None, 73, 4)             256

nerual_dense (Dense)         (None, 73, 70)            350

=================================================================
Total params: 86,046
Trainable params: 86,046
Non-trainable params: 0
```

Figure 10. (a) Hurwitz et al.'s LFADS Model Summary. (b) Keras Functional API LFADS Model Summary.

As mentioned in Chapter 4.1, the dense layer for generating the variance and the layer for generating the Gaussian samples were removed, so there are no such two layers in the reimplemented model. Although the mean values initialize the initial condition of the GRU, the GRU decoder still needs input. Thus, another input layer is added before the dense_mean layer. In Hurwitz et al.'s LFADS, the input to the decoder is inserted by Tensorflow2 functions instead of a Keras layer. To reimplement the entire model within Keras functional API, a Keras input layer is chosen for substituting the Tensorflow2 functions. Besides these layers, all other layers are matched to maintain the correct functionality of the model.

Next, the trained weights and biases were copied to the Keras functional API LFADS. We evaluated the performance of this model based on the same testing dataset, and it performs indicial to the no-Gaussian LFADS, which confirms that the reimplementation in Keras functional API does not affect any of the model performance.

Chapter 5. HLS4ML IMPLEMENTATION FOR LFADS

Besides the modifications of the LFADS architecture, several implementations were also necessary for the HLS4ML to push LFADS through. Implementing a new layer or adding a new feature into HLS4ML requires the developer to create several new files and classes in the current package. The Keras functional API LFADS has one HLS4ML-unsupported layer and one HLS4ML-unsupported feature. In this section, we will introduce the implemented new layer, bidirectional GRU, and the implemented new feature, GRU initial state, based on the Keras configuration and HLS4ML implementation details.

5.1 LFADS Bidirectioanl GRU – Keras Configuratoin

The keras.layers.Bidirectional is used for the LFADS encoder. It is a wrapper class that can wrap two Keras layers [23]. The general structure of the bidirectional layer is diagrammed in Figure 11.

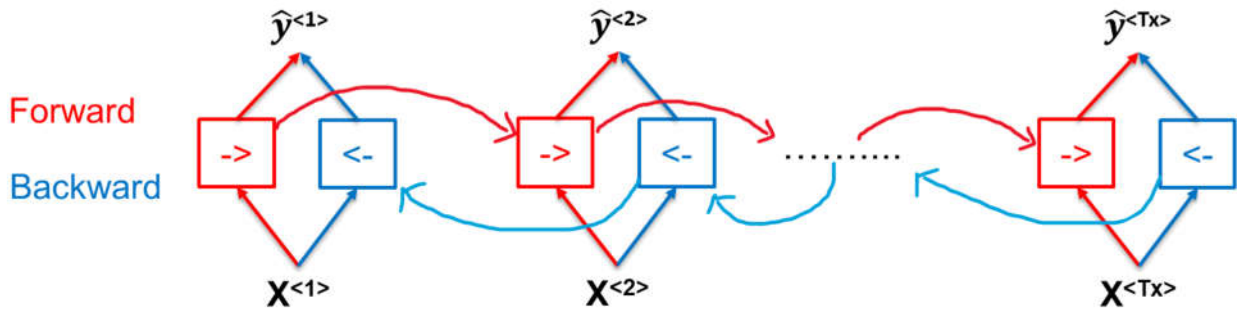
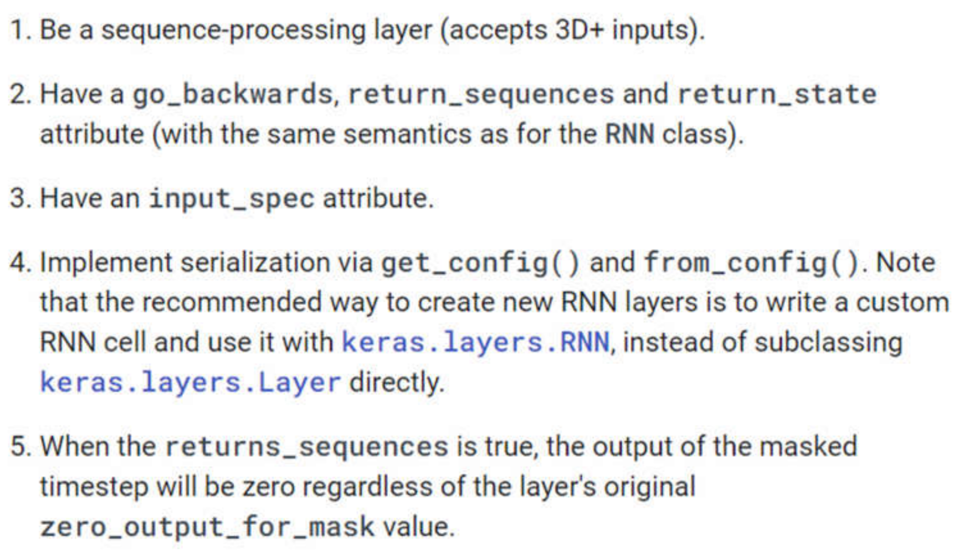


Figure 11. Bidirectional Layer Structure.

In the bidirectional layer, one forward layer processes the input data in the original sequence, and the other backward layer processes the input data in the reversed sequence. Both the forward layer and the backward layer must be the same type of RNNs or layers that fulfill specific criteria [23], shown in Figure 11. [23] presents the below arguments of the bidirectional layer.

Layer: The forward layer, which could be keras.layers.RNN instance, or keras.layers.Layer that fulfills the following criteria, which is shown in Figure 12.



1. Be a sequence-processing layer (accepts 3D+ inputs).
2. Have a `go_backwards`, `return_sequences` and `return_state` attribute (with the same semantics as for the RNN class).
3. Have an `input_spec` attribute.
4. Implement serialization via `get_config()` and `from_config()`. Note that the recommended way to create new RNN layers is to write a custom RNN cell and use it with `keras.layers.RNN`, instead of subclassing `keras.layers.Layer` directly.
5. When the `returns_sequences` is true, the output of the masked timestep will be zero regardless of the layer's original `zero_output_for_mask` value.

Figure 12. Requirements for Wrapped Layer instance [23].

Merge_mode: This "merge_mode" defines how the output from both the forward layer and the backward layer will be combined. A total of 5 choices are provided. Users can choose from "sum," "mul," "concat," "ave," or "None." By default, "concat" is the chosen mode.

Backward layer: The backward layer must match the type of the forward layer and the arguments in the forward layer except for the "go_backwards."

Call arguments: The call arguments of the keras.layers.Bidirectional are the same as those used for the forward and backward layers. There is one call argument that the developer needs to pay attention to, which is the “initial_state” of the RNNs. When passing a list of initial conditions to the bidirectional wrapper, the first half of the list will be passed into the forward layer, and the second half of the list will be passed into the backward layer.

For Hurwitz et al.’s LFADS, the encoder is a bidirectional GRU with “return_state” and “merge_mode” “concat.” The “return_state” argument of a GRU cell provides access to the hidden state value of the last time step. Figure 13 illustrates an example of the “return_state” argument of a GRU cell. In this figure, since the “return_state” argument is true, the hidden state value of the last time step, y_2, is accessed and outputted along with the GRU regular output. It is noteworthy that, in the case of GRU, the hidden state value of the last time step is identical to the GRU regular output.

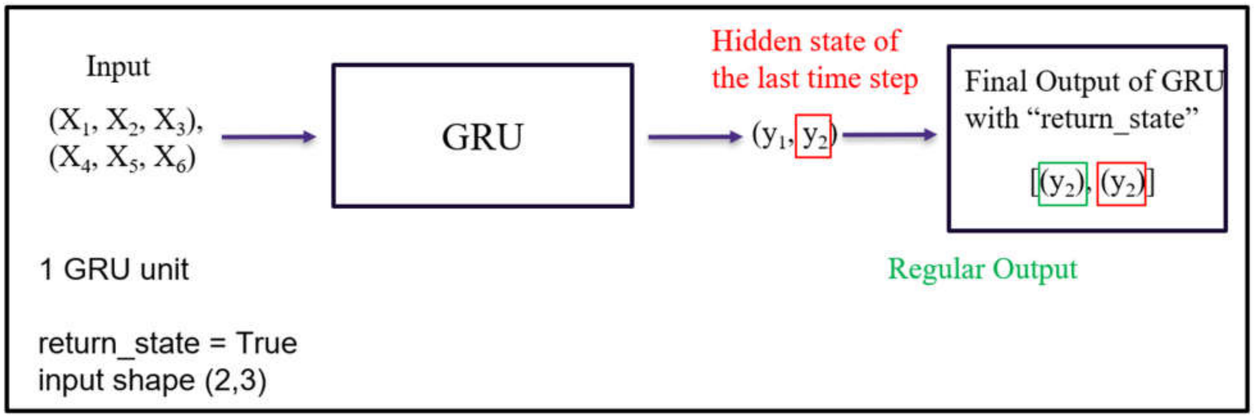

Input
(X_1, X_2, X_3), (X_4, X_5, X_6)
GRU
Hidden state of the last time step
(y_1, y_2)
Final Output of GRU with “return_state”
$[(y_2), (y_2)]$
Regular Output
1 GRU unit
return_state = True
input shape (2,3)

Figure 13. The Logic of the “return_state” Argument.

Figure 14 below demonstrates the logic of the LFADS bidirectional GRU encoder. The keras.layers.Bidirectional takes the input and passes it into the forward GRU. In the meantime, the input is automatically reversed so that the reversed input can be passed into the backward GRU. Since the “return_state” is true for both forward and backward GRU and the “merge_mode” is “concat,” the hidden state value of the last time step from both forward and backward GRU, y_forward$_2$ and y_backward$_2$, are concatenated. All these three values will be outputted. However, only the concatenated value will be passed into the next layer. The y_forward$_2$ and y_backward$_2$ are redundant.

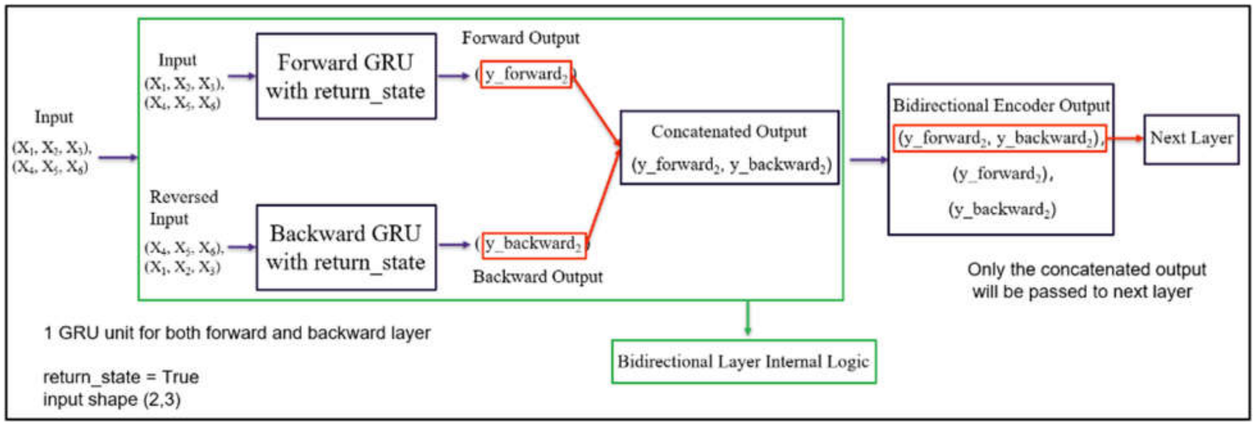

Input
$(X_1, X_2, X_3), (X_4, X_5, X_6)$
Input
$(X_1, X_2, X_3), (X_4, X_5, X_6)$
Forward GRU with return_state
Forward Output
(y_forward$_2$)
Reversed Input
$(X_4, X_5, X_6), (X_1, X_2, X_3)$
Backward GRU with return_state
(y_backward$_2$)
Backward Output
Concatenated Output
(y_forward$_2$, y_backward$_2$)
Bidirectional Encoder Output
(y_forward$_2$, y_backward$_2$),
(y_forward$_2$),
(y_backward$_2$)
Next Layer
Only the concatenated output will be passed to next layer
Bidirectional Layer Internal Logic
1 GRU unit for both forward and backward layer
return_state = True
input shape (2,3)

Figure 14. The Logic of LFADS Bidirectional GRU Encoder.

5.2 Bidirectional Layer – HLS4ML Implementation

In this work, only the unique case was implemented in HLS4ML, which is the bidirectional GRU with "return_state" for both the forward and backward layers and "merge_mode" "concat." This unique combination is the use case of the LFADS encoder. The other two types of RNN, vanilla RNN and LSTM, along with other arguments, can be added to the package by following the same procedure.

Generally, implementing a new Keras layer in HLS4ML follows the typical procedure, which is diagrammed in Figure 15.

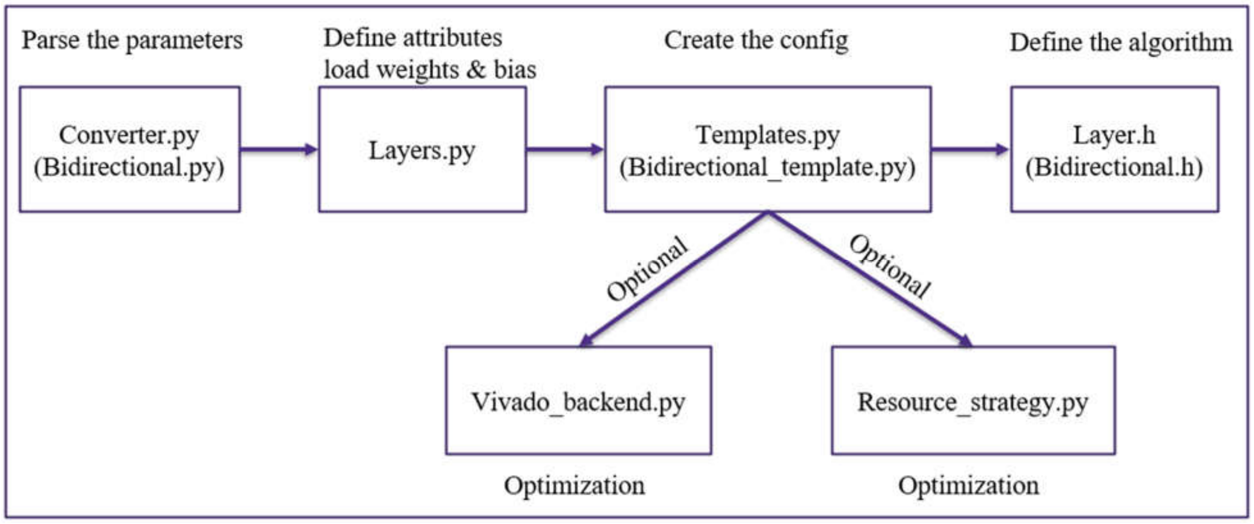


Figure 15. HLS4ML Implementation Flow.

There are four key files that the developer needs to create or modify, which are converter.py, layers.py, templates.py, and layer.h. To support the bidirectional GRU in HLS4ML, bidirectional.py, bidirectional_templates.py, and bidirectional.h were added to the package, and the layers.py were modified.

Bidirectional.py: This script is used to parse the necessary layer parameters from the keras.layers.Bidirectoinal to the HLS4ML. Parameters such as but not limited to input and output shape, “merge_mode,” and “return_state” were included in this file.

Layers.py: A class containing the expected attributes of the bidirectional layer was defined in this file. The expected attributes include choice attribute, weight attribute, type attribute, etc. By executing this file, the expected attributes, as well as the weights and biases of the trained bidirectional layer, can be loaded into HLS4ML.

Bidirectional_templates.py: This template converts the parameters of the bidirectional layer to C++ code during the translation. Since the bidirectional layer wraps two GRU layers, the bidirectional_templates.py was developed based on the recurrent_template.py. Implementations for the activation template and multiplication template were copied from recurrent_template.py.

Bidirectional.h: bidirectional.h is the header file that defines all the calculations that will happen inside the bidirectional GRU. Within this file, two HLS4ML GRU layers are instantiated. One functions as the forward layer, while the other serves as the backward layer. Figure 16 demonstrates an example of the calculation that happens inside the HLS4ML bidirectional GRU.

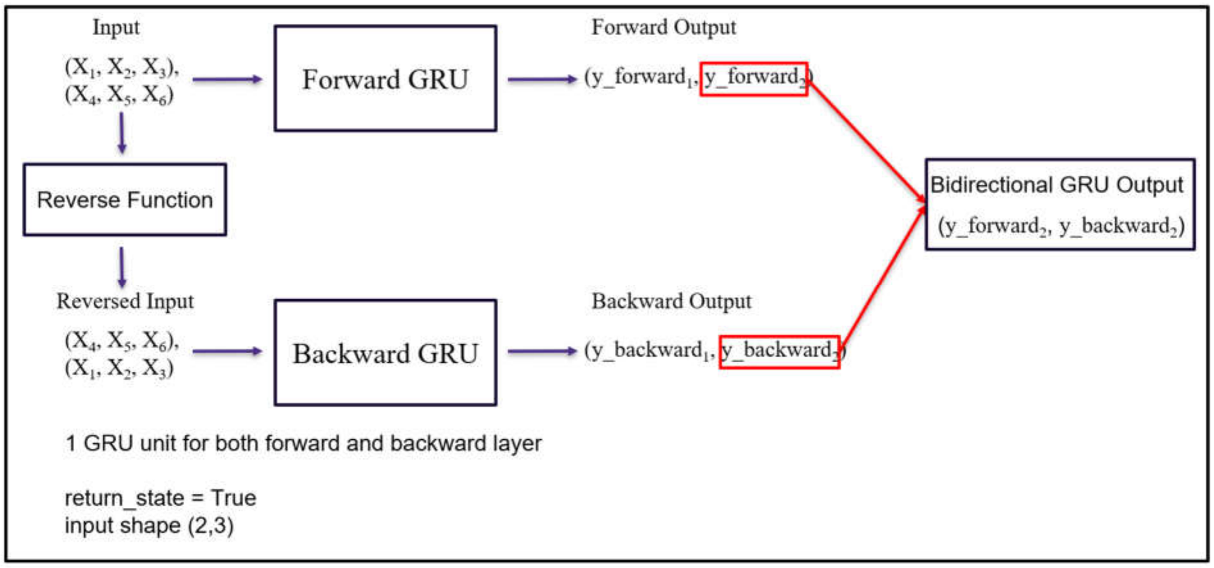

Input
(X_1, X_2, X_3), (X_4, X_5, X_6)
Forward GRU
Forward Output
(y_forward$_1$, y_forward$_2$)
Reverse Function
Bidirectional GRU Output
(y_forward$_2$, y_backward$_2$)
Reversed Input
(X_4, X_5, X_6), (X_1, X_2, X_3)
Backward GRU
Backward Output
(y_backward$_1$, y_backward$_2$)
1 GRU unit for both forward and backward layer
return_state = True
input shape (2,3)

Figure 16. Example of HLS4ML Bidirectional GRU.

Besides the two instantiated HLS4ML GRU, another function was implemented in the bidirectional.h to reverse the input. The original input will first be reversed by this function. Then the original input and the reversed input will be passed into the forward and backward GRU correspondingly. Since the "return_state" is true for both forward and backward GRU, only the hidden state value of the last time step from both the forward GRU and the backward GRU will be concatenated. Moreover, as mentioned in Chapter 5.1, the LFADS encoder passes the concatenated value of the forward and backward GRU to the next layer, so only the concatenated value is reserved for generating the final output.

Vivado_backend.py and resource_strategy.py: These two Python files do not directly affect the HLS4ML bidirectional layer implementation. The calculation inside the bidirectional layer is not defined within these two files, but the optimization of the HLS4ML bidirectional layer is related to these two files. The optimization of the bidirectional layer was implemented, but the performance of the optimization has not been checked yet.

Besides, all the implemented files preserve the space for adding new arguments or RNN instances, such as other types of "merge_mode" or RNN instances. Thus, it will be relatively easy to extend the function of the current HLS4ML bidirectional GRU.

5.3 GRU INITIAL_STATE – KERAS CONFIGURATION

Besides the bidirectional GRU encoder, the other HLS4ML unsupported feature in the Keras functional API LFADS is the call argument, "initial_state," of the unidirectional GRU decoder. This call argument is used to set the initial condition of the GRU before the data processing. The existing logic of the HLS4ML GRU layer can only support zero-initialized conditions. But in LFADS, the initial condition of the decoder GRU is initialized by the mean values.

When a sequence of input is processed, the initial condition is the first hidden state of the GRU. Figure 17 shows a detailed GRU architecture. In this example, when the input sequence is going to be processed by the GRU, the "initial_state" will initial the first $\boldsymbol{h_{t-1}}$ to the given values before the processing.

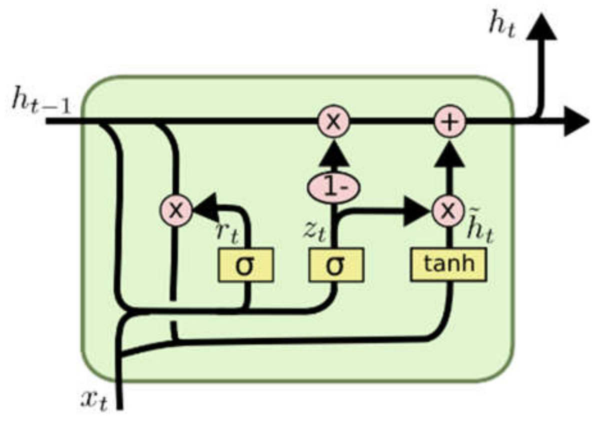


Figure 17. GRU Architecture [14].

5.4 GRU Initial_State – HLS4ML Implementation

The implementation of the “initial_state” followed a similar procedure as the previous implementation. Since there is an existing HLS4ML GRU layer, adding new files is unnecessary. All modifications were made within the existing files. Figure 18 shows the steps of implementing “initial_state” into HLS4ML GRU.

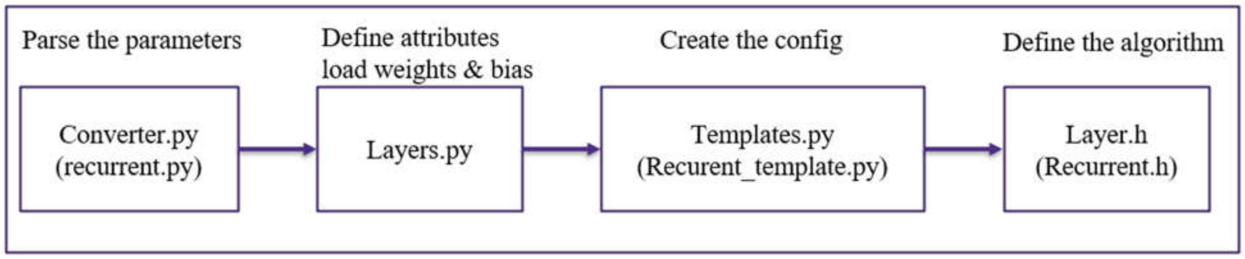


Figure 18. The Implementation Flow of “Initial_state” in HLS4ML GRU.

The “initial_state” was added to recurrent.py, layers.py, and recurrent_template.py. The logic in recurrent.h was modified to initialize the initial condition of the GRU to the given input values other than 0. Previously, the first hidden state of HLS4ML GRU was initialized to 0 within a loop before processing the sequence. After the implementation, the hidden state was able to be initialized to whatever the given values were.

5.5 HLS4ML Implementation - I/O Parallel vs I/O Stream

When implementing the header file for a new layer or a new feature in HLS4ML, the developer can choose to use two types of coding techniques. One is I/O Parallel, and the other is I/O Stream. The HLS4ML implementation described in Chapter 5.1 and Chapter 5.3 was done in I/O Parallel. The key distinction between I/O Parallel and I/O Stream in HLS4ML resides in how they control the data processing. I/O parallel focuses on parallelizing the data within the clock cycle to increase the throughput, while I/O Stream processes the data sequentially to lower the FPGA resource utilization [24].

The conversion from I/O Parallel to I/O Stream for the above implementation was done by Yan-Lun Huang from Dr. Bo-Cheng Lai's group at the National Yang Ming Chiao Tung University. Yan-Lun followed the logic of the I/O Parallel implementation to develop the bidirectional GRU and "initial_state" in the I/O Stream version. Yan-Lun further ensured that the logic and the performance of bidirectional GRU and GRU "initial_state" in the I/O Stream version matches that of the I/O Parallel version.

Chapter 6. LFADS ON FPGA

Next, we will talk about the FPGA deployment of the Keras functional API LFADS. When deploying a model onto FPGA by using the HLS4ML workflow, the developers need to choose the target board, evaluate the HLS model performance, and analyze the latency of the inference and the FPGA resource utilization. In this chapter, we will first show the HLS model performance.

Then we will analyze the latency and resource utilization of the deployment. It is worth noting that the I/O Stream version of LFADS was used for this deployment to lower the FPGA resource utilization.

6.1 KERAS MODEL PERFORMANCE VS HLS MODEL PERFORMANCE

We pushed the Keras functional API LFADS through the implemented HLS4ML. The HLS model was successfully generated. During the translation process, post-training quantization (PTQ) was applied to the HLS model. Figure 19 shows the HLS model performance in different PTQ precision.

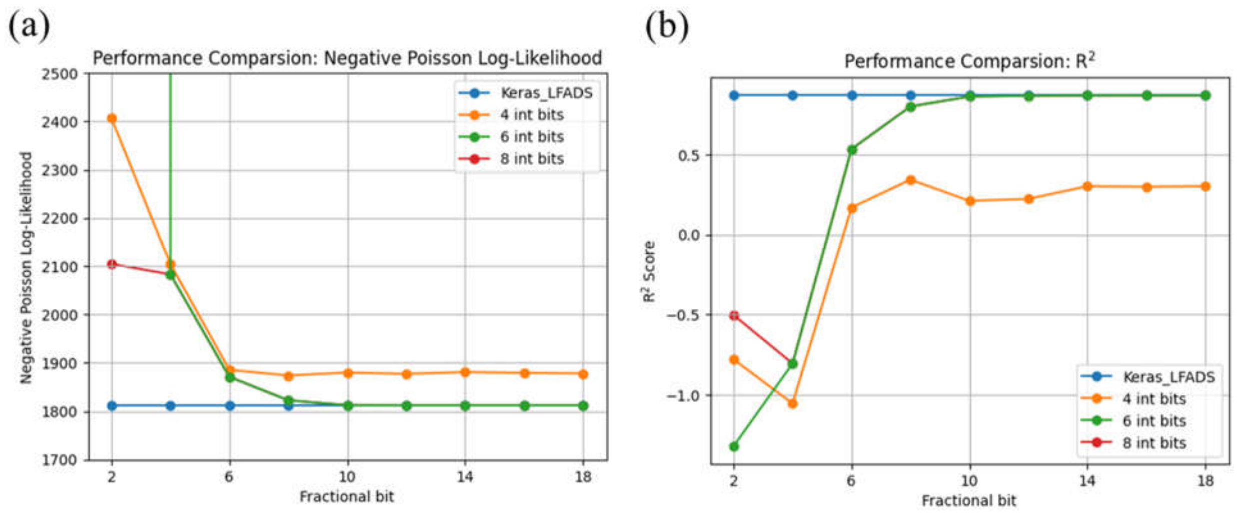


Figure 19. (a) Negative Poisson Log-likelihood of HLS Model in Different Precision. (b) R^2 Score of HLS Model in Different Precision.

In the above figure, the blue line in each figure represents the Keras model performance. It is straightforward to notice that as the fractional bit and integer bit increase, the HLS model performance gets better. At around six integer bits and ten fraction bits, the HLS model performs nearly the same as the Keras model.

We then deployed the HLS model with precision ap_fixed<16,6> onto Xilinx U55C and evaluated the on-board performance. The on-board performance was the same as the HLS model performance depicted in the above figure.

6.2 FPGA Resource Utilization

This chapter will talk about the latency of the inference and the resource utilization of the Xilinx U55C. The configuration of the Xilinx U55C deployment is shown in Table 1.

Table 1: Deployment Configuration.

Target FPGA	Xilinx U55C
Precision	Ap_fixed<16,6>
Frequency	200 MHz
Latency	41.97μs

The precision used for the deployment is ap_fixed<16,6>, which is six integer bits and ten fractional bits. The latency is 41.97μs for running 1 batch of testing data under 200 MHz. Moreover, Table 2 shows the resource utilization of deployment.

Table 2. Xilinx U55C Resource Utilization.

FPGA component	Resource utilization	Resource utilization (%)
BRAM	474	23.51%
DSP	1,869	20.71%
FF	150,882	5.79%
LUT	164,726	12.64%

Table 2 demonstrates that only around one-quarter of the resource on the board was utilized, which indicates that we can deploy several more models onto U55C with ap_fixed<16,6>.

Nevertheless, as mentioned in Chapter 2.2, MRAE has 1536 units of LFADS-like model. Therefore, to deploy MRAE onto FPGA, a further decrease in resource utilization is necessary.

Chapter 7. CONCLUSION

In this thesis, we described the work that has been done in the first two years of deploying MRAE onto FPGA to analyze sleep spindles, which is the FPGA deployment of LFADS. At first, we studied the concepts and architectures of RAE, MRAE, and LFADS. After that, we discussed the modification of the LFADS architecture and the implementation of the HLS4ML. Next, we demonstrated the performance HLS model in different bit precision. Finally, we showed the results of the deployment, including the latency of the inference and the resource utilization of the target FPGA board.

Chapter 8. FUTURE WORK

The future goal is to deploy MRAE onto FPGA. To achieve this, one of the most important tasks is to integrate the Gaussian sampling layer with the current no-Gaussian LFADS. Since the MRAE follows the VAE structure, the support of the entire VAE architecture in HLS4ML is necessary.

Another challenge we face is the high resource utilization of the FPGA. The current resource utilization is still too high to deploy a large model. Compared to LFADS, the MRAE is extremely large. To deploy a large model like this, several possible methods can be considered.

1. Apply quantization-aware training (QAT) on the model. Unlike PTQ, QAT allows the user to train the model with varied bit precision. This provides the opportunity to use fewer bits for the FPGA deployment while still maintaining reasonable performance. Currently, Chi-Jui Chen and Lin-Chi Yang from Dr. Lai's group are working on applying the QAT on LFADS and implementing both the quantized GRU and quantized bidirectional GRU in HLS4ML. Once these studies are finished, the FPGA resource utilization can be largely reduced.
2. Simplify the MRAE architecture. The current MRAE model is used to precisely reconstruct the broadband μECoG signal. However, sleep spindles are low-frequency brain signals. To interact with sleep spindles, a simplified MRAE that only aims to reconstruct the signals in the low-frequency band may be enough.

Chapter 9. REFERENCE

[1] G. R. Yang and X.-J. Wang, “Artificial Neural Networks for Neuroscientists: A Primer,” *Neuron*, vol. 107, no. 6, pp. 1048–1070, Sep. 2020, doi: 10.1016/j.neuron.2020.09.005.

[2] L. M. J. Fernandez and A. Lüthi, “Sleep Spindles: Mechanisms and Functions,” *Physiological Reviews*, vol. 100, no. 2, pp. 805–868, Apr. 2020, doi: 10.1152/physrev.00042.2018.

[3] A. Peyrache and J. Seibt, “A mechanism for learning with sleep spindles,” *Phil. Trans. R. Soc. B*, vol. 375, no. 1799, p. 20190230, May 2020, doi: 10.1098/rstb.2019.0230.

[4] M. Nolan, B. Pesaran, E. Shlizerman, and A. Orsborn, “Multi-block RNN Autoencoders Enable Broadband ECoG Signal Reconstruction,” Neuroscience, preprint, Sep. 2022. doi: 10.1101/2022.09.07.507004.

[5] J. Duarte *et al.*, “Fast inference of deep neural networks in FPGAs for particle physics,” *J. Inst.*, vol. 13, no. 07, pp. P07027–P07027, Jul. 2018, doi: 10.1088/1748-0221/13/07/P07027.

[6] Q. V. Le, “A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks”.

[7] G. E. Hinton and R. R. Salakhutdinov, “Reducing the Dimensionality of Data with Neural Networks,” *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006, doi: 10.1126/science.1127647.

[8] D. Bank, N. Koenigstein, and R. Giryes, “Autoencoders.” arXiv, Apr. 03, 2021. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/2003.05991

[9] Y. Zhang, “A Better Autoencoder for Image: Convolutional Autoencoder”.

[10] T. Kieu, B. Yang, C. Guo, and C. S. Jensen, “Outlier Detection for Time Series with Recurrent Autoencoder Ensembles,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Macao, China: International Joint Conferences on Artificial Intelligence Organization, Aug. 2019, pp. 2725–2732. doi: 10.24963/ijcai.2019/378.

[11] J. Feng and Z.-H. Zhou, “AutoEncoder by Forest,” *AAAI*, vol. 32, no. 1, Apr. 2018, doi: 10.1609/aaai.v32i1.11732.

[12] C. Doersch, “Tutorial on Variational Autoencoders.” arXiv, Jan. 03, 2021. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/1606.05908

[13] M. Sit, B. Demiray, Z. Xiang, G. Ewing, Y. Sermet, and I. Demir, “A Comprehensive Review of Deep Learning Applications in Hydrology and Water Resources,” Engineering, preprint, Jun. 2020. doi: 10.31223/OSF.IO/XS36G.

[14] R. Rao, “Implementation of Long Short-Term Memory Neural Networks in High-Level Synthesis Targeting FPGAs”.

[15] D. Sussillo, R. Jozefowicz, L. F. Abbott, and C. Pandarinath, “LFADS - Latent Factor Analysis via Dynamical Systems.” arXiv, Aug. 22, 2016. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/1608.06315

[16] C. Pandarinath *et al.*, “Inferring single-trial neural population dynamics using sequential auto-encoders,” *Nat Methods*, vol. 15, no. 10, pp. 805–815, Oct. 2018, doi: 10.1038/s41592-018-0109-9.

[17] C. Hurwitz *et al.*, “Targeted Neural Dynamical Modeling.” arXiv, Oct. 27, 2021. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/2110.14853

[18] M. G. Perich, J. A. Gallego, and L. E. Miller, "A Neural Population Mechanism for Rapid Learning," *Neuron*, vol. 100, no. 4, pp. 964-976.e7, Nov. 2018, doi: 10.1016/j.neuron.2018.09.030.

[19] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A Survey of FPGA-Based Neural Network Accelerator." arXiv, Dec. 06, 2018. Accessed: Aug. 03, 2023. [Online]. Available: http://arxiv.org/abs/1712.08934

[20] C. Wang and Z. Luo, "A Review of the Optimal Design of Neural Networks Based on FPGA," *Applied Sciences*, vol. 12, no. 21, p. 10771, Oct. 2022, doi: 10.3390/app122110771.

[21] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, Xi'an, China: IEEE, Dec. 2016, pp. 77–84. doi: 10.1109/FPT.2016.7929192.

[22] J. P. Cunningham and B. M. Yu, "Dimensionality reduction for large-scale neural recordings," *Nat Neurosci*, vol. 17, no. 11, pp. 1500–1509, Nov. 2014, doi: 10.1038/nn.3776.

[23] "tf.keras.layers.Bidirectional | TensorFlow v2.13.0," TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional (accessed Aug. 03, 2023).

[24] T. Aarrestad et al., "Fast convolutional neural networks on FPGAs with hls4ml," *Mach. Learn.: Sci. Technol.*, vol. 2, no. 4, p. 045015, Dec. 2021, doi: 10.1088/2632-2153/ac0ea1.

Chapter 10. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Foundation under Grant No. OAC-2117997.

First and foremost, my sincere appreciation goes to both of my advisors, Dr. Scott Hauck and Dr. Shih-Chieh Hsu. Thank you for all the support and advice you provided during my time in graduate school. I would not have been able to finish my graduate study without your constant support and advice. I'm sincerely grateful to be a part of the ACME lab and the A3D3 institute. The time I spent in the ACME lab and A3D3 institute is one of the most precious memories of my life.

A big thank you to Dr. Elham E Khoda for helping me understand the HLS4ML framework and make plans for the project when I was confused about the direction. My sincere thanks go to Michael Nolan for walking me through the concepts of MRAE and LFADS, as well as the basic concepts of neuroscience. Special thanks to Aidan Yokuda for working with me to get familiar with the hardware device and understand the system setup. I am appreciative of the assistance from Zhixing "Ethan" Jiang. Thank you for discussing the HLS4ML implementation details with me and providing me with guidance for the HLS4ML implementation. I'm also grateful for the help from Vladimir Loncar. Thank you for providing advisable solutions for HLS4ML-related questions. I extend my thanks to Dr. Amy Orsborn, Dr. Eli Shelizerman, Dr. Leo Scholl, Trung Le, and Lauren Peterson. Thank you all for providing me with help in understanding the related concepts neuroscience area.

Many thanks to Dr. Bo-Cheng Lai, Chi-Jui Chen, Lin-Chi Yang, and Yan-Lun Huang. Thank you all for your contributions in the past several months. There would not have been such significant progress without your help. It is my sincere pleasure to work with you all.

Thank you to the researchers in both the HLS4ML community and the A3D3 institute who have helped me and worked with me during my graduate study. Collaborating with the talented people in both these two organizations has inspired my personal growth.

To my dear wife, Yujue Wang, I would like to express my deepest appreciation. Thank you for always being on my side. You always let me pursue my interests and give me confidence when I face the moment of self-doubt. I could not achieve this today without your support.

To my dear family, my parents Li Liu and Haibo Wang, my grandparents Changli Wang and Yuxiu Ying, my uncle Haitao Wang, my parents-in-law Juncheng Wang and Yawei Wang. Thanks for believing me in the past few years and giving me constant support when I question myself. I feel incredibly fortunate to have my loving family.

Thank you to all my friends and my family who have provided support in both my academic study and personal life over the past few years.

Thank you all. My gratitude goes beyond words.

APPENDIX

Appendix A: HLS4ML Layer Implementation Tutorial

The tutorial for implementing a new Keras layer in HLS4ML - https://github.com/uw-acme/acme-lab-documentation/tree/main/hls4ml_layer_implementation_tutorial. In this tutorial, the implementation flow with examples is described in detail. Developers can follow the tutorial to implement the unsupported layers in HLS4ML.

Appendix B: Github Repositories

Tensorflow 2 Keras LFADS - https://github.com/XLSMALL/TF2Keras-LFADs

This repository contains the scripts that we used to reimplement the Keras functional API LFADS and also the dataset that is introduced in chapter 2.4. It is forked from https://github.com/HennigLab/tndm, which was developed by Hurwitz et al.

HLS4ML for LFADS - https://github.com/XLSMALL/hls4ml/tree/LFADs

This repository contains the implemented HLS4ML package. The implementation is what we have discussed in this thesis, bidirectional GRU and the GRU “initial_state,” which are necessary for LFADS deployment.