



**ASHESI UNIVERSITY**

**CREATING TOOLS FOR MACHINE LEARNING ACCELERATION AT THE EDGE**

**CAPSTONE PROJECT**

B.Sc. Computer Engineering

**Alosius Akonteh**

**2024**

## **Abstract**

Machine learning at the edge has grown in popularity due to its low latency and cost, coupled with its efficiency for IoT applications. Field Programmable Gate arrays are often used as edge devices to perform inference; however, the development process is not easy to complete quickly due to the knowledge of Hardware Description Language required. This paper proposes a solution to enable Long Short-Term Memory Recurrent Neural Networks as an extension of a previous project to enable the use of machine learning algorithms on embedded systems hardware.

*Keywords:* Machine Learning, Edge Computing, Hardware Description Language, Embedded Systems, Internet of Things, Long Short-Term Memory Recurrent Neural Networks

## Table of Contents

Chapter 1 - Introduction .....	6
The Problem.....	7
The Background .....	8
How the Problem will be addressed .....	9
<i>Overview of Remaining Chapters</i> .....	10
Chapter 2 – Related Work.....	11
2.1 Neural Networks - A Brief Overview .....	11
2.2 Machine Learning Acceleration - Hardware .....	13
2.3 Machine Learning Optimization - Software .....	13
2.4 Hardware Description Language Generation – Software .....	14
Chapter 3: System Requirements and Design.....	16
3.1 Proposed Solution.....	16
System Requirements .....	16
3.3 Use Case.....	18
3.4 Operating Environment .....	18
3.5 Design Constraints and Scope.....	18
3.6 Design Overview .....	19
3.7 Number Format .....	20
3.8 Configuration Module.....	20

3.9 LSTM Network Module .....	20
3.9.1 LSTM Unit or Layer.....	21
3.9.2 Forget Gate .....	21
3.9.3 Input Gate .....	22
3.9.4 Candidate State.....	23
3.9.5 Cell State Update .....	24
3.9.6 Output Gate .....	25
3.9.7 Hidden State Update.....	26
3.9.8 Matrix Multiplier .....	26
3.9.9 Higher Bias Adder .....	27
3.9.10 Adder .....	27
3.9.11 Multiplier .....	28
3.9.12 Element wise multiplier .....	28
3.9.13 Activation .....	28
3.9.14 FPGA Controller .....	29
Chapter 4: Implementation .....	30
4.1 Class Hierarchy.....	30
4.1.1 Component Class .....	31
4.1.2 Activation Class.....	31
4.1.3 Gate Class.....	31
4.1.4 LSTM Cell Class.....	32

4.1.5 LSTM Unit Class.....	32
4.1.6 Model Class.....	32
Chapter 5: Testing .....	33
5.1 Unit Testing .....	33
5.2 Simulation Testing.....	33
5.3 Synthesis Testing.....	34
Chapter 6: Conclusion.....	36
6.1 Limitations .....	36
6.2 Recommendations and Future Work.....	36

## **Chapter 1 - Introduction**

Machine learning is a field that continually spreads its influence into various domains of life. It has contributed tremendously to automating tasks and improving the quality of services by enabling machine learning agents to learn based on data and improve their ability to respond adequately. The Internet of Things (IoT) is a field that involves making electronic devices smart by allowing them to exchange data over a network to enable action. Today, IoT and machine learning work together to solve problems more efficiently. IoT sensors collect data used in the machine learning training process to make predictions more accurate. However, machine learning, unlike human learning, relies heavily on numerical computations involving tons of data and finding similarities or patterns that would serve as the basis for decision-making. At first glance, this would not be a problem considering the computational power of computers today. However, with the rapidly increasing need for accuracy coupled with the advent of big data, using a laptop or desktop with general-purpose central processing units in tandem with IoT devices will be cumbersome and slow because of data transmission latency, and computational latency. Cloud servers are generally used instead of laptop/desktop computers because of their high computational performance. Nonetheless, data needs to be sent over the internet from the IoT sensors to the servers for computation before results are sent to the network of interest for decision-making. Two main factors are considered here for an IoT system: the data transfer speed from the IoT sensors to the processing device and the processing speed of the processing device.

A few concerns arise considering the use of remote servers for computation. The security and privacy of IoT sensor data transferred to cloud servers is not assured, hence, third parties could access the data without the owners' consent. Bandwidth could present a limitation with increasing amounts of data resulting in possible delays in receiving inferences which are generally undesirable. The monetary cost of transferring data using an internet connection and purchasing

cloud storage to store the data before it is processed could exceed the user's budget. Moreover, for Internet of Things (IoT) applications where sensors collect data regularly and actuators respond based on inference from sensor readings, it is desirable to have a computation device of a small size to easily interface with the IoT system or the use of wireless communication for data transfer to the inference center. Using wireless communication leads us back to the latency issue associated with remote servers. The drawbacks associated with the approach described above demand an alternative that would provide smaller latency, portability, lower power requirements, and low cost. Previous attempts to address this problem involved using Raspberry Pi boards as edge devices to perform computations and another involved creating a library to enable edge computations on Field Programmable Gate Arrays boards which will be discussed later in this paper. The former consumes more power making it less appealing as an alternative. The latter has been implemented successfully for Artificial Neural Networks. This project, creating tools for accelerated machine learning at the edge, seeks to provide an alternative for machine learning that does not rely solely on remote servers for computation by using Field Programmable Gate Arrays for computations using a hardware description language. The hardware will be interfaced with the system that collects data and will perform computations necessary for inference-making of Long Short-Term Memory Recurrent Neural Networks.

### **The Problem**

In the artificial intelligence and machine learning industries, the latency of inferencemaking and input data needs to be minimized. Machine inference is therefore best when it happens fast. However, with the large quantity of data generated by sensors and actuators, transmission latency is high when remote servers are used for the computation and interpretation of data for machine learning agents. The use of remote servers to effect computations is called cloud computing. A counterpart that will be used in this paper is edge computing which involves processing data closer

to its collection source. The closer the data is to its collection source, the smaller the transmission source. [1] As a remedy to the transmission latency issue, devices that would enable processing closer to the data source, edge devices, are created. Nonetheless, edge devices are not meant to completely replace cloud servers but rather to provide an alternative that would reduce latency in the machine learning process by processing data close to its collection site. In a typical IoT edge computing environment, data will be transferred from IoT sensors to a computing unit, in this case, the FPGA board. This is different from the classical approach to machine learning as this approach requires knowledge of Hardware Description Language making the development process difficult to complete in a short time. This project addresses this issue by enabling the generation of Hardware Description Code for LSTM inference given LSTM network parameters post-training.

## **The Background**

Other implementations of edge machine learning involve high-end boards like the Nvidia Jetson Nano and Tensor Processing Units. The Jetson nano is a small powerful computer specifically designed for running artificial intelligence applications at the edge. Tensor Processing Units (TPU) are integrated circuits designed to perform large matrix multiplications as is generally the case with machine learning computations. While the jetson nano and TPUs are efficient, they are quite expensive and consume much power making [2] them less desirable. The approach described in this paper utilizes dedicated hardware (FPGA) that provides a good compromise between efficiency, power consumption, and cost. The hardware is programmed using Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). VHDL is a standardized language used to describe and model the behavior and structure of digital systems. VHDL is portable, can describe concurrent processes, and is reusable making it suitable for use in describing machine learning functionalities on hardware. Python is the most widely used programming



language for AI purposes. For simplicity of use, the users will write their machine-learning programs in Python as is commonly the case because of its ease of use, vast array of libraries, and versatility. The solution, a Python library, will cater to the VHDL functionalities required to perform the desired computations by generating VHDL implementations while blending into the common machine-learning workflow.

### **How the Problem will be addressed**

This project involves creating a program that converts Long Short-Term memory Recurrent Neural Networks in Python into VHDL for computation on FPGA (Field Programmable Gate Array). FPGAs are integrated circuits that can be configured after manufacturing. They contain arrays of configurable logic blocks that can be programmed to perform various functions. FPGAs grant flexibility, performance, and parallel processing with low power consumption in many cases [3]. This approach is different in that it will lay a foundation for making machine learning at the edge low on latency and cheap, less power consuming, and would require little new knowledge to use. The approach reduces latency by removing the need to transfer data over the internet to a remote server by becoming the center of computation for the IoT system. The approach is cheap given that the basys3 board used costs about \$200 and there is no need for the purchase of a cloud server subscription for computing. Users will only need to know about machine learning in Python and will not have to learn new libraries to make their algorithms compatible with FPGA. They will only need to input their code into the proposed solution and conversion to VHDL will be handled for them. Machine learning will be more accessible cost-wise by utilizing cheaper processing alternatives and will reduce the latency associated with remote processing. The flexibility of FPGAs will make testing easier by allowing for the configuration of configurable blocks to suit specific workloads and machine learning algorithms. The program will be designed for specific algorithms and will be documented to make updating easy with the advent of new

machine learning algorithms. This project will focus on the neural network approach to machine learning as a foundation for other algorithms.

### ***Overview of Remaining Chapters***

The next chapters will address the rationale for choosing the physical components of the project and programming language, the choice of algorithms for implementation, the logic behind the conversion from Python code to VHDL, and the program testing and identification of strengths and shortcomings.

## **Chapter 2 – Related Work**

This chapter examines the literature in the field of machine learning acceleration and embedded systems to gain knowledge from the methods that have been already explored or are currently used to allow for better decision-making during the system design and implementation.

### **2.1 Neural Networks - A Brief Overview**

Over the years different machine learning algorithms have been implemented to solve problems that would be too complex to hardcode given the huge number of possible scenarios. Among these is a class of algorithms that have become prevalent in the field of machine due to their flexibility in addressing complex problems and fitting different kinds of patterns in data, called neural networks. Neural networks draw some inspiration from human neurons. Similar to biological neurons, neurons in a machine learning neural network store information and are used to mimic thinking using mathematical computations. Neural networks involve multiple neurons arranged in layers to capture the complexities of the “thinking” process. The layered architecture of a neural network is such that neurons in layer  $n$ , are connected to neurons in the layer after it, layer  $n+1$ , and layers before it, layer  $n-1$  if any. The connection between the neurons in a layer and those in a connected layer carries values called weights that scale the input to a neuron layer to influence complex inferencing. [4].

Different networks exist namely, fully connected and partially connected [5]. In fully connected layers, all neurons in layer  $n$  are connected to and feed their outputs to all neurons in layer  $n+1$ . In the case of recurrent neural networks which this paper will address, the output of each neuron is fed back to the neuron as one of its inputs giving it “memory”. This “memory” is beneficial for solving problems with a sequential connection between data points such as word prediction problems.

### 2.1.1 Long Short-Term Memory (LSTM) Recurrent Neural Networks

LSTMs are a class of recurrent neural networks that handle long and short-term temporal dependencies in input data. They can “remember” the near and distant past to inform their output. Consider a case of speech prediction. To determine which word should come next, the context of which words are present in the sentence is necessary and because the words follow a temporal order (one word comes before the next), it is necessary to account for that. Predicting the next word for the sentence “There is a book” would be inaccurate if only the word “book” was considered as input (just a one-time step back). It is helpful to think of this as being asked to guess the next word someone would say after hearing just a word they spoke. The context is better captured if the four words are considered (four-time steps back).

LSTMs are composed of multiple subparts called gates that process the different calculations involved in making predictions. The different gates include the forget gate, candidate gate, input gate and output gate.

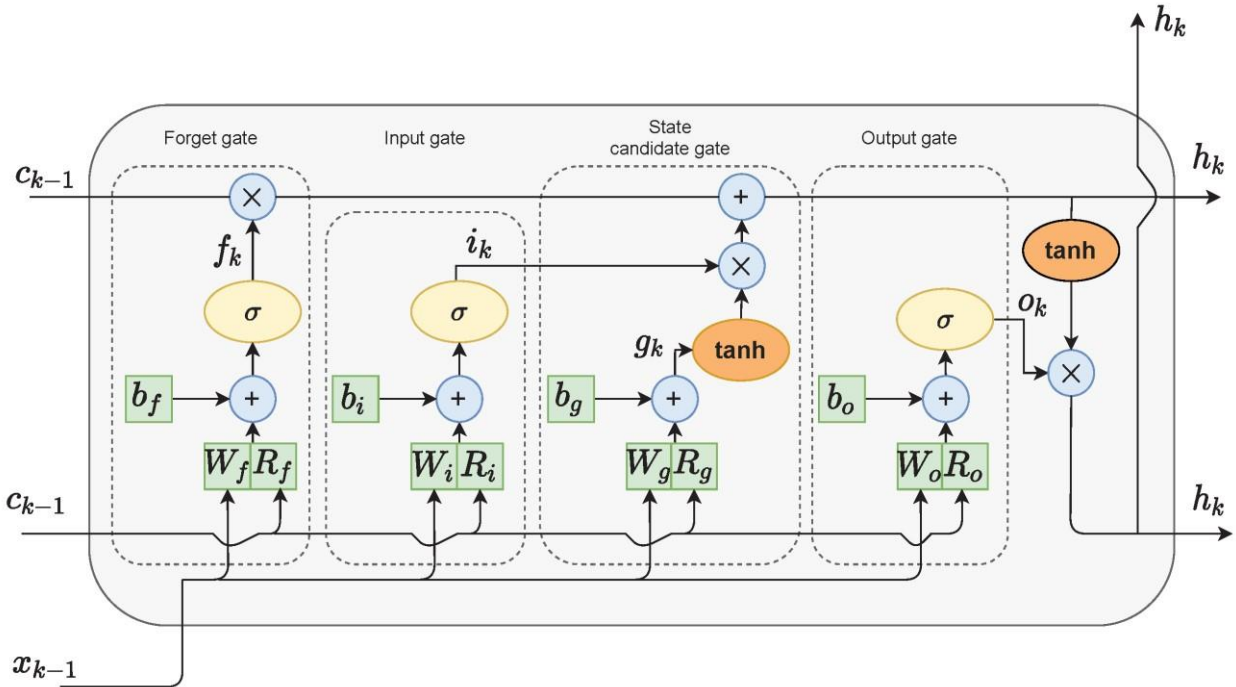


Figure 1 Structure of an LSTM Cell [6]

## **2.2 Machine Learning Acceleration - Hardware**

The hardware used in machine learning can either be general-purpose or specialized hardware. This paper explores Field Programmable Gate Arrays (FPGA) due to their lower cost, power consumption, and latency compared to Central Processing Units and Graphics Processing Units [7]. This section explores different machine learning hardware accelerators and compares them to the proposed accelerator. In [8] the authors utilized the Xilinx ZedBoard for machine learning acceleration of malware detection algorithms for edge devices. The ZedBoard development board combines a softcore CPU, one implemented using logical synthesis in hardware description language and FPGA structure allowing for high computational power. The authors employed C/C++ machine learning algorithms and the Vivado HLS to convert these algorithms to VHDL upon development. This approach differs from the proposed solution in its use of C/C++ machine learning algorithms which are not the most common in the field of machine learning. More so, they employ hardware that includes a CPU and FPGA, a distinction from the singular use of FPGA for this project which will have a major impact on cost. The results of their approach, however, proved that hardware acceleration enhances security by not exposing processed data to a network and provides high efficiency as suggested in chapter one of this paper. [8] attempts to compare the deployment of machine learning algorithms on CPUs, GPUs, and FPGA to compare inference times and rate the efficiency of running inference on FPGA. The results are promising given the high level of parallelism present in FPGAs compared to CPUs and GPUs. The authors used the LeFlow model [9] to convert numerical computation models in TensorFlow to Register Transfer Level (RTL) code for HDL synthesis.

## **2.3 Machine Learning Optimization - Software**

Deploying machine learning on FPGAs requires adjustments to the software that is to be run on them. The resource-constrained nature of hardware makes software optimizations crucial

to ensure maximum utilization of the hardware capabilities. In the context of neural networks, some of such adjustments would be pruning parameters to reduce the number of parameters to be processed [10], analyzing weights and removing the most inconsequential ones, and selecting the right number of layers and neurons for accurate inference. Choosing the adequate type of neural network is a consideration that cannot be overlooked because different neural networks have different computational and memory requirements. For example, long short-term memory (LSTM) neural networks require computations for hidden states and current states and need to store each cell's current and previous state to incorporate the memory component into a neural network [11]. Artificial Neural Networks do not have these requirements and simply perform computations using the known weights and biases obtained from training and the current inputs. LSTMs can capture time dependencies in input data making them suitable for problems where having previous information is necessary. Consider a case where a designer wants to predict the power consumption in a building; having information on previous power consumption and current building occupancy would benefit from an LSTM over an ANN given the time dependency involved.

Though this paper's focus is not on optimizing users' choices in constructing their neural networks, it addresses these ideas to inform users in making more informed decisions to draw the most out of hardware acceleration of machine learning inference to meet their respective functional requirements.

## **2.4 Hardware Description Language Generation – Software**

Some software that allow users to generate HDL code are LeFLoW, which was mentioned earlier, and Vivado HLS. Vivado HLS is not specific to machine learning but works with C/C++ implementations of hardware functionalities to generate corresponding RTL implementation. This implies that users would either implement their LSTM neural networks in C/C++ or find a

way to represent the underlying computations in C/C++ if they hope to work with the software. That would be time-consuming and potentially tedious for complex implementations besides being prone to error.

## **Chapter 3: System Requirements and Design**

This chapter proposes a solution to the issues discussed in the previous chapters by describing the process of creating tools to enable machine learning inference for diverse algorithms on embedded hardware at the network edge, defines the requirements and use cases of the said solution, and lays out a high-level view of the proposed solution.

### **3.1 Proposed Solution**

The paper proposes a framework for enabling the use of machine learning algorithms on FPGA for inference. The hardware required will be a microcontroller coupled with a Field Programmable Gate Array (FPGA) as a hardware accelerator for testing the solution. The microcontroller will be used as an edge IoT device to provide input data for the machine learning model. As discussed in the previous chapter, the FPGA will serve as the computation center at the network edge to run Long Short-Term Memory (LSTM) Recurrent Neural Networks. Every other embedded logic will be performed by the microcontroller. Because this solution focuses primarily on the generation of Very High-Speed Integrated Circuit Hardware Description Language (VHDL) for the FPGA board, the proposed programming language and structure of the conversion program will be given a high-level overview.

Using a microcontroller will overcome the I/O limitations of the FPGA in cases where the inputs cannot be handled by the FPGA. More so, a microcontroller gives additional flexibility to the systems designer using the solution as they will be able to choose their microcontrollers based on their specific requirements.

### **System Requirements**

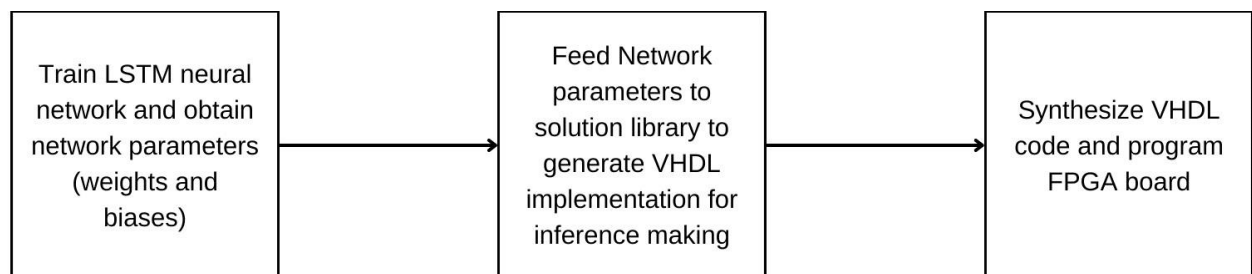
This system is targeted at embedded and IoT systems designers seeking an effective way to run machine learning algorithms with good latency and accuracy performance on edge devices using a combination of a microcontroller and FPGA.



The system should:

- Convert LSTM recurrent neural networks into Hardware Description Language compatible with FPGAs.
- Provide an interface for communicating data between FPGA and microcontroller.
- Be expandable to include other machine learning algorithms.
- Be cost-effective, that is, have low hardware and software costs.
- Work with low latency.
- Be easy to use and upgrade and be modular. The system should provide the user with detailed control over the designs and allow for flexibility in upgrading and modifying to suit specific design constraints.

This solution will be implemented as a library that takes in the parameters of a pretrained LSTM model and generates a VHDL implementation for it. A protocol for data communication between the FPGA and microcontroller will also be implemented for testing purposes. The solution will generally be used in three main steps: the user trains an LSTM to generate model optimal parameters; the conversion stage where the user provides the model parameters and type of algorithm; and the post-processing stage where the user uploads the VHDL version of their model unto the FPGA and uploads the microcontroller code for communicating the data to the FPGA for inference and receiving inference results.



*Figure 2 Block Diagram Illustration of Process*

The library will be implemented in the Python programming language and the microcontroller will be programmed to send inputs to the FPGA and receive outputs from it in C for the testing phase. Python was chosen because it is beginner-friendly and easy to integrate to machine learning workflows which are typically written in Python.

### **3.3 Use Case**

Michael Coleman, a machine learning engineer, and embedded systems designer has collected data with long-term dependencies and trained an LSTM neural network to solve a problem. He wishes to deploy the solution to the field where the solution is needed. Due to his design constraints of size, latency, and efficiency, he cannot deploy the solution on a computer as it will be too big to integrate into the system, increase the latency of the system, consume much power, and be expensive to purchase. Thus, he will need a small yet efficient alternative, the FPGA, to achieve his objectives. He will extract the parameters of his trained LSTM model and feed them to the library to be created to generate Hardware Description Language implementation for use with an FPGA and microcontroller combination.

### **3.4 Operating Environment**

The library targets x86-64 and ARM Desktop PCs with Windows, macOS, and Linux operating systems. The solution will be able to work on any device in CPython, the reference implementation of the Python programming language due to its widely spread use and high compatibility with python code and libraries. However, it is not guaranteed to be compatible directly with other Python implementations.

### **3.5 Design Constraints and Scope**

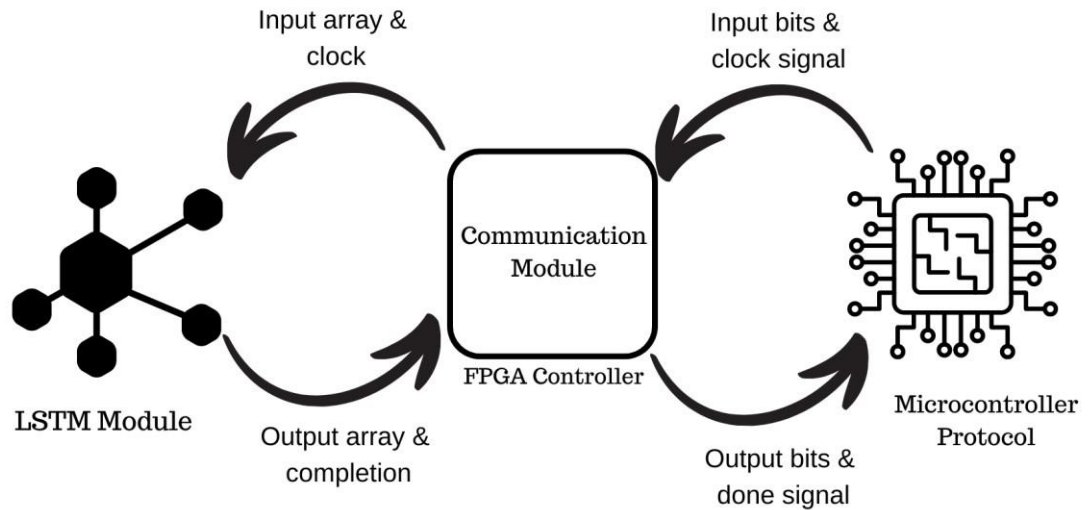
FPGAs and microcontrollers have specific names and mappings that vary from one to the other. Given the abundance of FPGAs and microcontrollers, the library will not generate individual constraint files and I/O mappings. Constraint files map logical elements of a VHDL

design to specific hardware resources on the FPGA board. Thus, users will ensure the compatibility of FPGAs and mapping connections to their microcontrollers. This limitation ensures that only the core logic for the FPGA is developed and allows abstraction regardless of the device used.

The library will be limited to generating HDL code only for fully connected neural networks to simplify the architecture.

### 3.6 Design Overview

The library will generate three high-level sub-systems: The LSTM Network Module, the FPGA Controller, and the Microcontroller Protocol. The LSTM module represents the implementation of the LSTM network in VHDL. The FPGA controller will interface with the microcontroller and move data to and from the LSTM module running on the FPGA. In a nutshell, it will handle I/O operations for the FPGA. The Microcontroller protocol enforces rules for the microcontroller to communicate with the FPGA to pass inputs and receive outputs.



*Figure 3 Subsystems Interactions*

### **3.7 Number Format**

To maximize precision, neural networks are generally trained using floating point numbers because they allow for a wide range of numbers with small increments to be represented by varying the position of the floating point. However, floating point numbers are computationally heavy, especially for resource constrained environments. Using floating point numbers on FPGA will mandate a floating-point unit implementation for every arithmetic operation it performs leading to high memory overhead. Fixed point numbers have lower precision and range but benefit from lighter computation weight.

The system will allow users to choose the number of decimal places of precision to represent the whole number and the numbers will be scaled up by a power of ten to ease computations on the FPGA. No external floating-point library will be used.

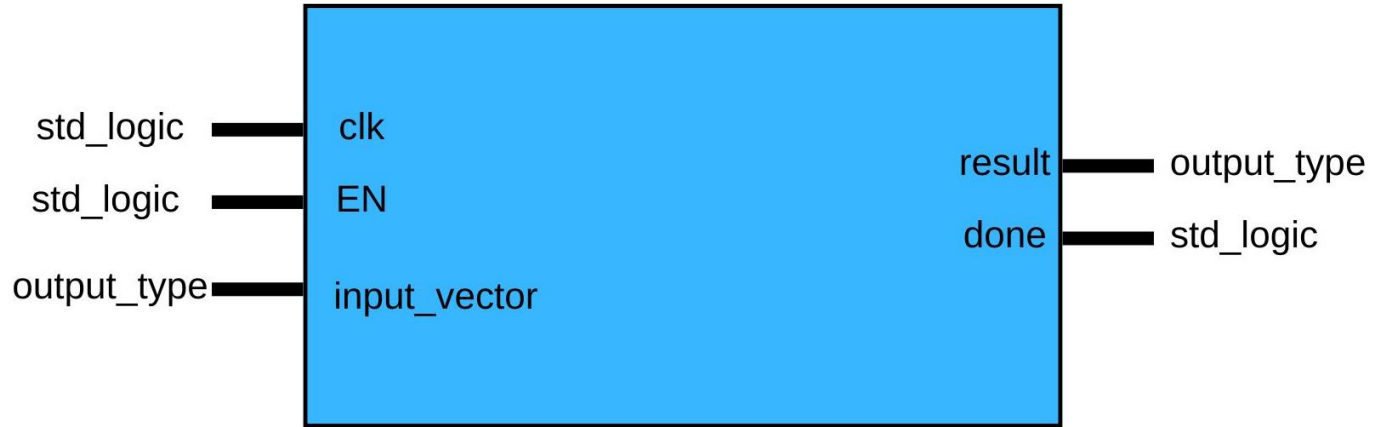
### **3.8 Configuration Module**

The configuration module serves as a reference, storing the details of the system and components that might be shared across the network. It contains specifics for the types used by the system and creates types based on the nature of the inputs.

### **3.9 LSTM Network Module**

This module is responsible for the computation of results using input data. It receives data from the microcontroller and performs computations. It returns a completion signal and the result of the computation.

The LSTM Network module is represented as a component consisting of a layer component, further made up of LSTM cells, which are composed of multipliers, adders, and activation units. This hierarchy is used because of its intuitiveness. Each component represents the components of an LSTM neural network.



*Figure 4 LSTM Network Module*

The LSTM network component accepts the input to the network and outputs the results and a signal that indicates that it is done processing the inputted data. It stores the parameters of the entire network, essentially weights, biases, cell states, and hidden states for every layer. It has the same interface as in Figure 2.

### **3.9.1 LSTM Unit or Layer**

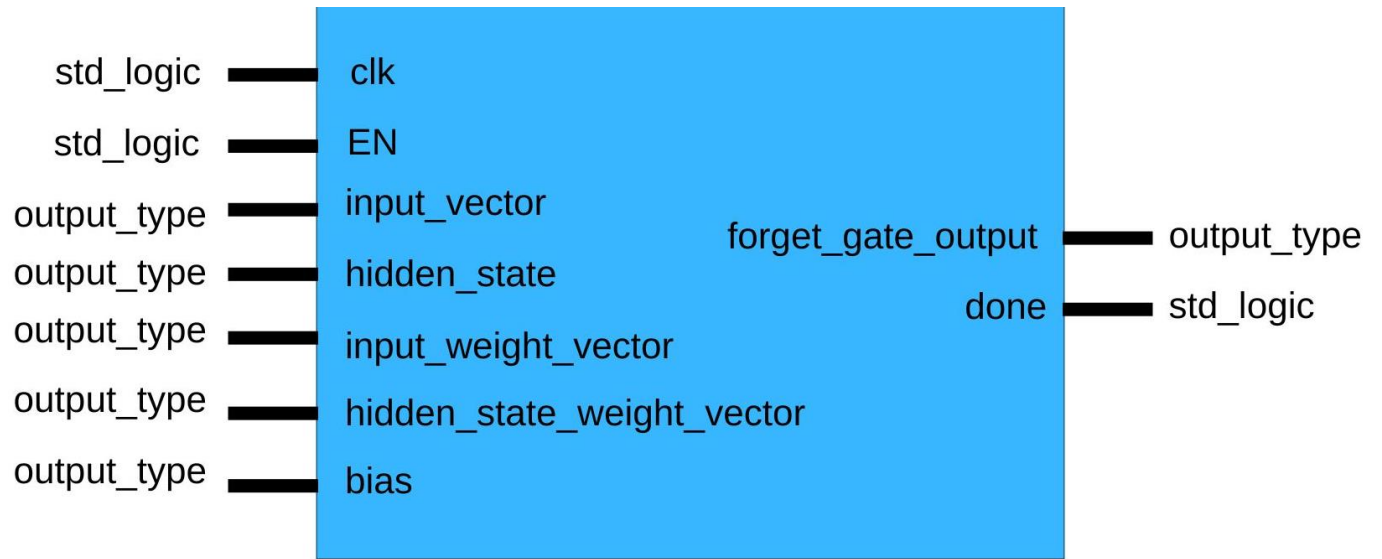
This component represents LSTM units in the network. Each unit has different gates that handle different functions as explained in chapter 2. Each unit serves as a housing for the gates where each computation happens. It has a similar structure to figure 2.

### **3.9.2 Forget Gate**

The forget gate uses the current input and last short-term memory or hidden state with their respective weights to determine how much of the long-term memory to remember or keep for the next computations. It uses the equation:

$$f_t = \sigma(W_i * x_t + W_h * h_{t-1} + b_{i\_f})$$

where  $W_i$  is the weight matrix connecting the input,  $x_t$ , to the forget gate,  $W_h$  is the weight matrix connecting the previous short-term memory to the forget gate,  $h_{t-1}$  and  $b_{i\_f}$  is the bias vector for the forget gate. The sigmoid activation function keeps the values between 0 and 1.



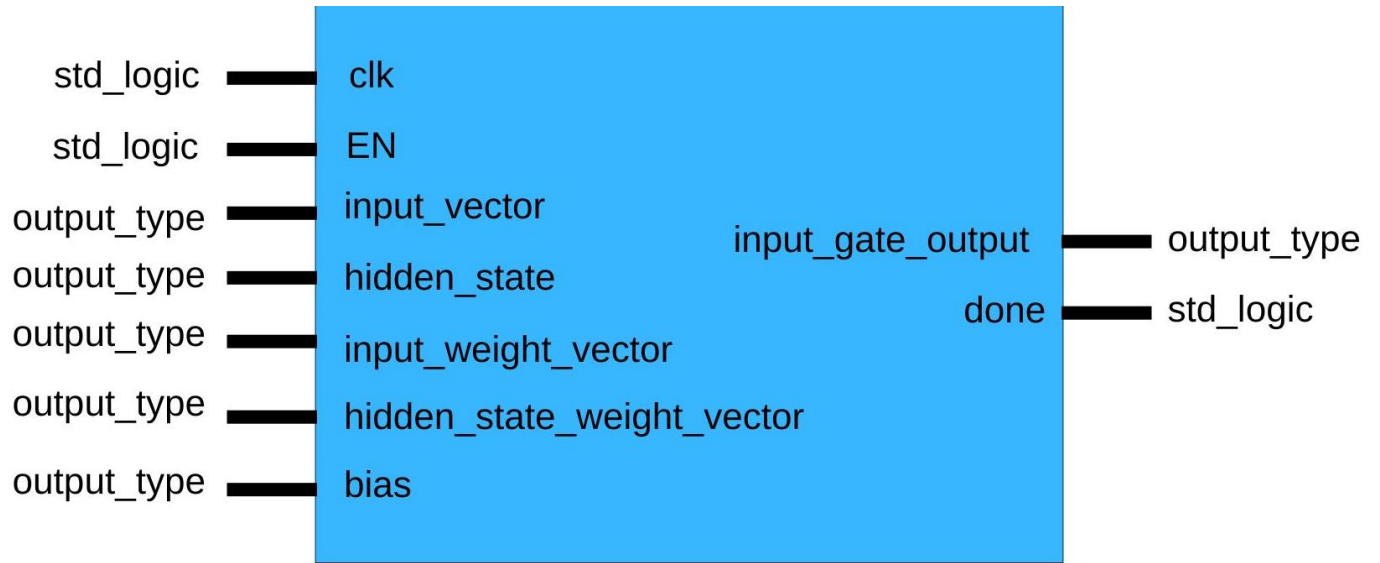
*Figure 5 Forget Gate Interface*

### **3.9.3 Input Gate**

The input gate determines how much of the new candidate cell state should be considered. It has a similar structure to the forget gate. Its result is also passed as input for a sigmoid activation function.

$$i_t = \sigma(W_i * x_t + W_h * h_{t-1} + b_{i_i})$$

where  $W_i$  is the weight matrix connecting the input,  $x_t$ , to the input gate,  $W_h$  is the weight matrix connecting the previous short-term memory to the input gate,  $h_{t-1}$  and  $b_{i_i}$  is the bias vector for the input gate. The sigmoid activation function keeps the values between 0 and 1.



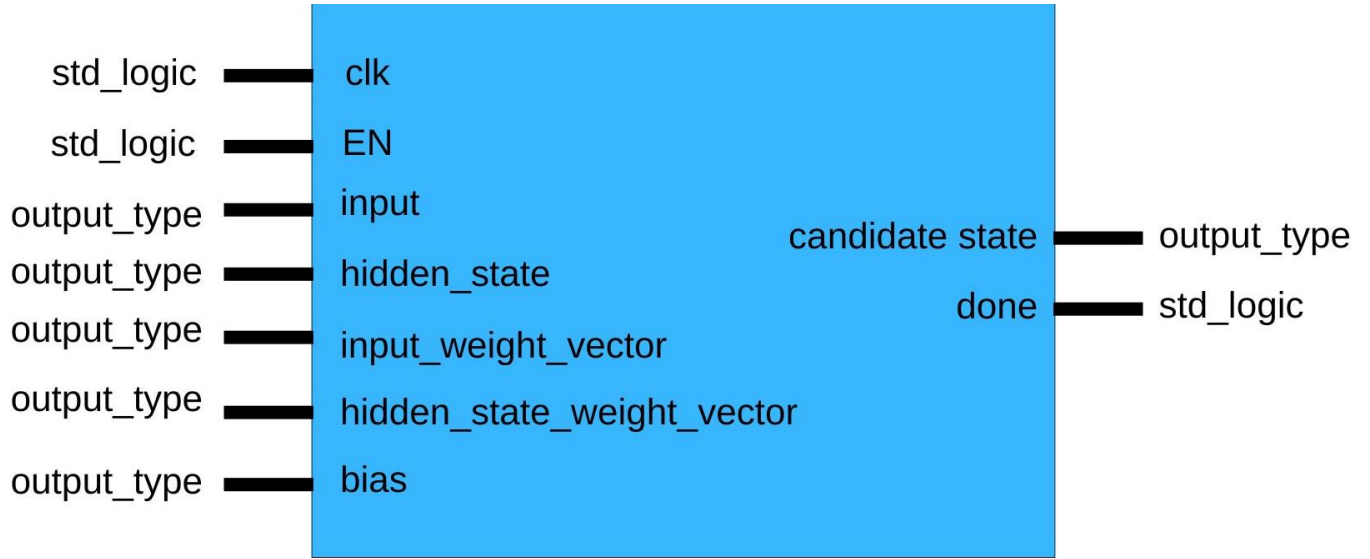
*Figure 6 Input Gate Interface*

### **3.9.4 Candidate State**

This part is often associated with the input gate but for greater modularity, it will be computed as a separate module. The candidate state represents the new information that could potentially be added to the long term memory. It is computed using the current input, previous short-term memory and a tanh activation function. The module won't include the activation function to increase modularity.

$$C_{t\_candidate} = \tanh(W_i * x_t + W_h * h_{t-1} + b_{i\_c})$$

where  $W_i$  is the weight matrix connecting the input,  $x_t$ , to the candidate state,  $W_h$  is the weight matrix connecting the previous short-term memory to the candidate state,  $h_{t-1}$  and  $b_{i\_c}$  is the bias vector for the candidate state. The tanh activation function keeps the values between -1 and 1.



*Figure 7 Candidate Cell State Interface*

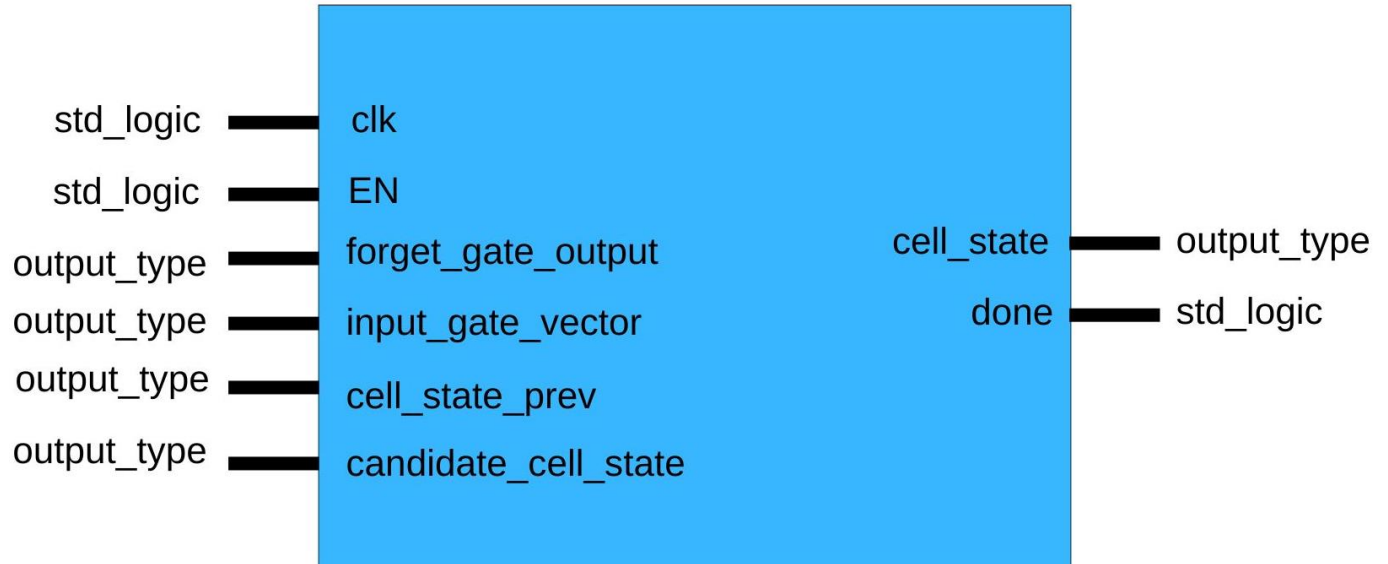
### **3.9.5 Cell State Update**

The cell state represents the long-term memory. It is computed by multiplying the results of the forget and input gates with the long-term memory and candidate state respectively and summing the products together. In simpler terms, it combines the information we have chosen not to forget due to its relevance with the new information we think is relevant.

$$C_t = f_t * C_{t-1} + i_t * C_{t\_candidate}$$

where  $C_t$  is the cell state/long-term memory,  $f_t$  is the amount of the previous long-term memory to remember,  $C_{t-1}$  is the previous long-term memory,  $i_t$  is how much of the new information to be added to memory and  $C_{t\_candidate}$  is the potential information to add to memory.





*Figure 8 Cell State Update Interface*

### 3.9.6 Output Gate

The output gate determines how much of the new long-term memory to include in the new hidden state/short-term memory, the actual output of our LSTM unit. It follows a similar structure to the forget gate.

$$o_t = \sigma(W_i * x_t + W_h * h_{t-1} + b_{i_o})$$

where  $W_i$  is the weight matrix connecting the input,  $x_t$ , to the output gate,  $W_h$  is the weight matrix connecting the previous short-term memory to the output gate,  $h_{t-1}$  and  $b_{i_o}$  is the bias vector for the output gate. The sigmoid activation function keeps the values between 0 and 1.

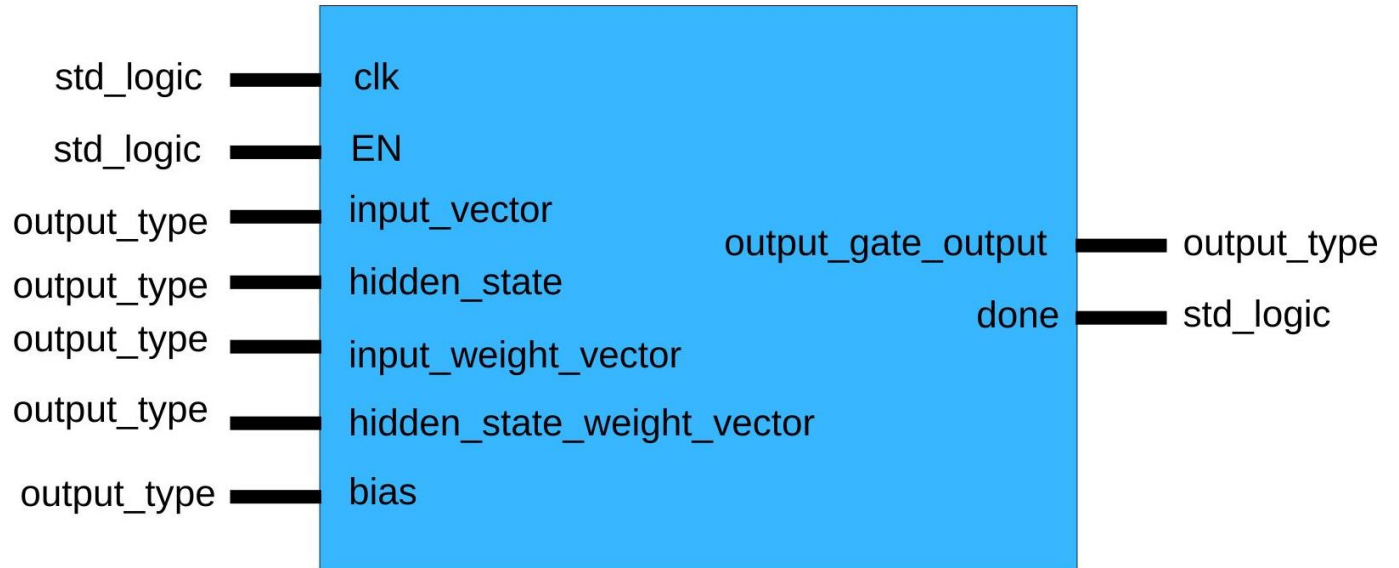


Figure 9 Output Gate Interface

### 3.9.7 Hidden State Update

This block computes the new hidden state by multiplying the output weight vector with the current cell state passed through a tanh activation function.

$$h_t = o_t * \tanh(C_t)$$

where  $h_t$  is the new hidden state,  $o_t$  is the output weight vector from the output weight block passed through a sigmoid function and  $C_t$  is the current cell state.

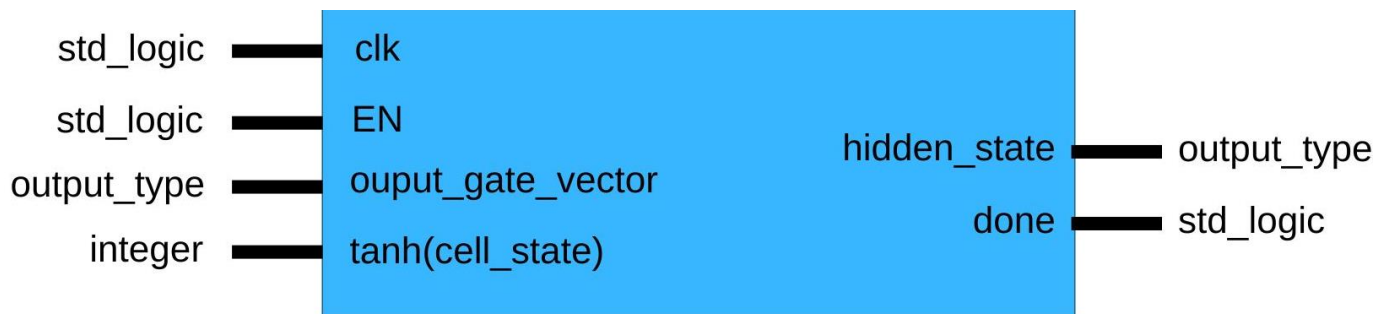
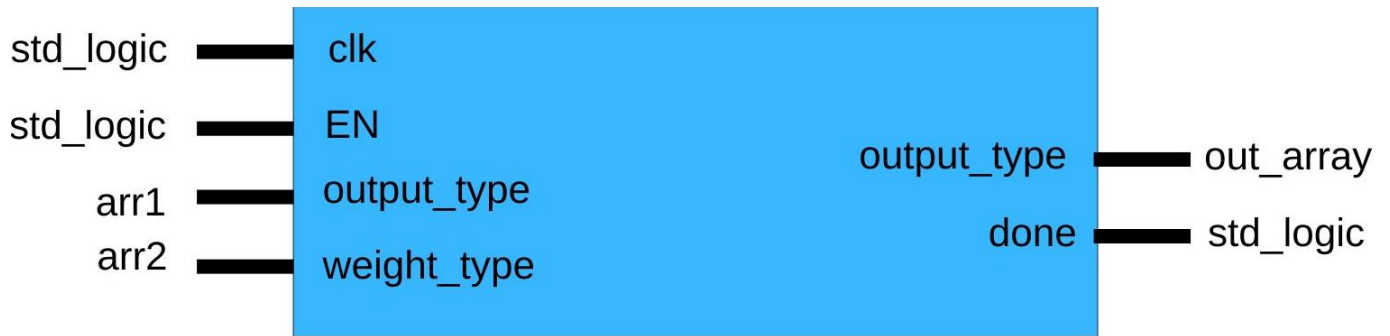


Figure 10 Hidden State Update Interface

### 3.9.8 Matrix Multiplier

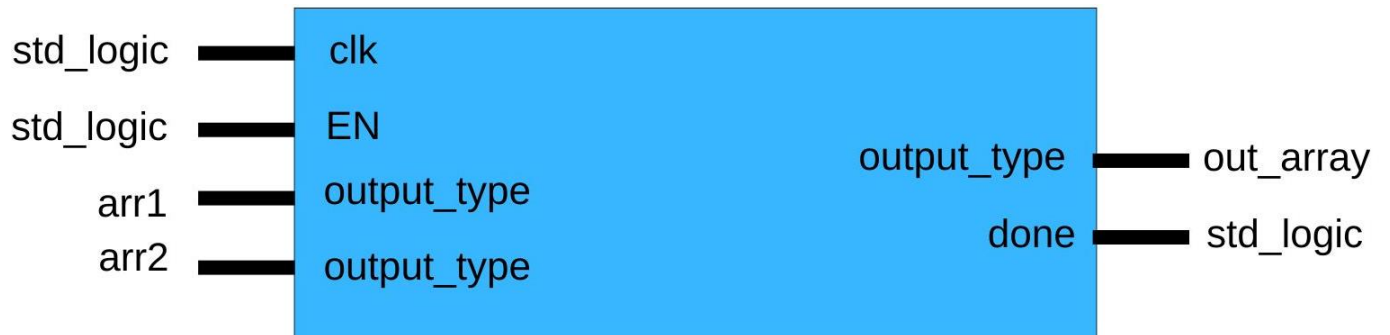
The matrix multiplier module does integer multiplication of two integer arrays and outputs and integer array of the same length as the inputs alongside a completion signal.



*Figure 11 Matrix Multiplier Interface*

### **3.9.9 Higher Bias Adder**

The higher bias adder module performs element wise addition of two matrices and outputs an array of output type and a completion signal. It will add the results from the matrix multipliers of the different gates of the LSTM network.



*Figure 12 Higher Bias Adder Interface*

### **3.9.10 Adder**

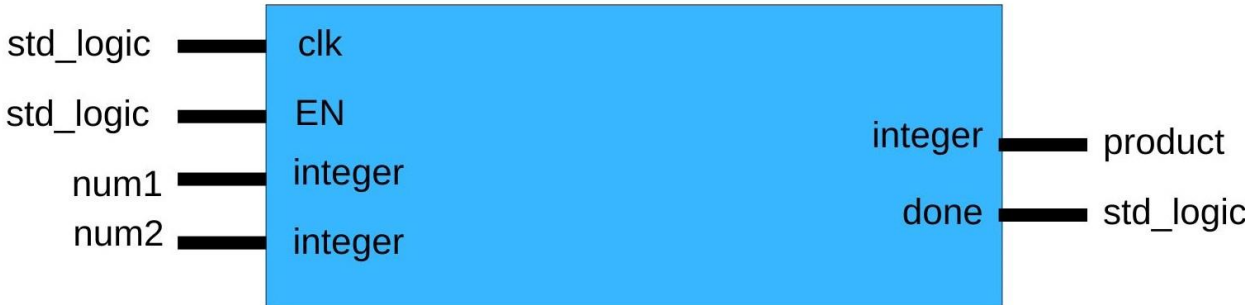
The adder module adds two integers and outputs their sum and a completion signal. This would mainly be used to add the bias terms.



*Figure 13 Adder Interface*

### **3.9.11 Multiplier**

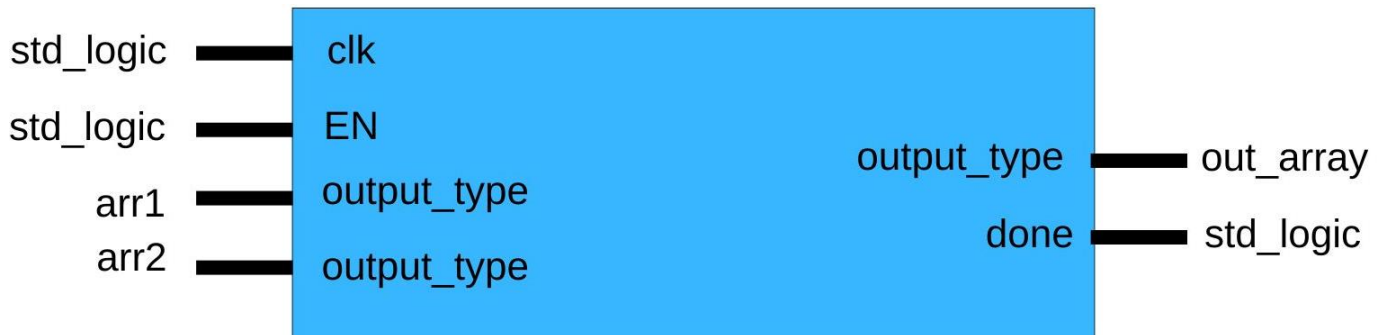
The multiplier module takes in two integers and outputs their product and a completion signal. This component will perform the products from the different gates.



*Figure 14 Multiplier Interface*

### **3.9.12 Element wise multiplier**

The element wise multiplier module performs element wise multiplication of two arrays and returns the product and a completion signal.



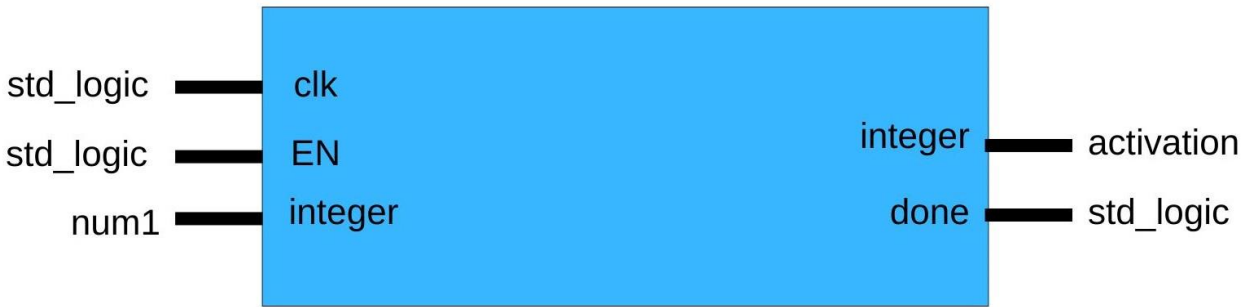
*Figure 15 Element Wise Multiplication Interface*

### **3.9.13 Activation**

The activation module takes in results from individual gates and applies the necessary activation function. It takes an integer input and outputs an approximation of the output activated value and a completion signal. The LSTM layer module uses the completion signal to

determine when all the gates have outputted their values to calculate the final inference. Given the nature of LSTM layers, sigmoid and tanh activations are used to constrain values to a small range.

- Sigmoid:  $f(x) = 1 \div (1 + e^{-x})$
- Tanh:  $f(x) = \tanh(x)$



*Figure 16 Activation Interface*

An input matrix sent to the LSTM network is fed to the respective gates of the network to determine how much of the previous memory to remember and eventually update the long and short-term memories. Each gate uses matrix multipliers and matrix adders to compute their outputs and passes them to the activations whose results are used to calculate the new short-term memory, which is the output of the LSTM network.

#### **3.9.14 FPGA Controller**

The FPGA controller facilitates communication between the microcontroller and the FPGA. It contains an instance of the LSTM network and stores the LSTM network's inputs and outputs. The controller will enable the network after receiving inputs from the microcontroller and send a completion signal to the microcontroller when done. It will also be equipped with a reset button to clear the network memory and a port to receive signals from the microcontroller to execute different tasks: wait, begin computation, and send output.

## Chapter 4: Implementation

This chapter describes the project library and how its different subsystems are implemented. Object-Oriented Programming (OOP) was the approach used in writing the library. OOP is a concept that groups data and actions that can be performed on the data into structures called objects, created from classes that serve as references for the creation of each object. The Object-Oriented Programming paradigm was employed to ensure modularity and scalability, to ease modification and updating, and to allow for different subsystems to be modified independently. Each component of the library is a class that interacts with other classes using inheritance and composition. The library classes and their relationships are shown in the UML diagram below

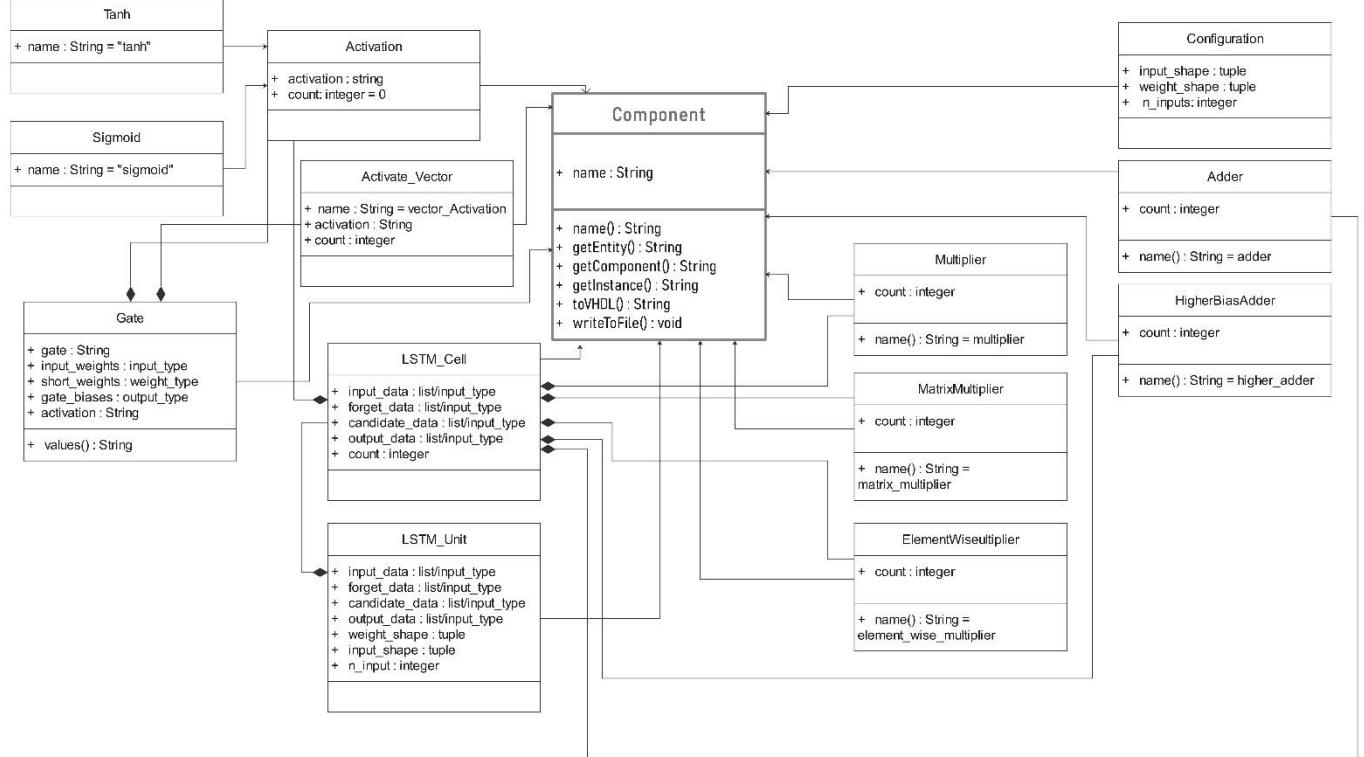


Figure 17 UML Diagram of Library

### 4.1 Class Hierarchy

The following sections will briefly explain how the main classes work and their contributions to the library's functionality.

#### ***4.1.1 Component Class***

The Component class is the base class for all the classes that generate VHDL components. It defines the interface that every class that creates VHDL code would follow. It defines the following attributes:

- Name: Refers to the name of the component and determines the name of the .vhd file created from each interface. Its default value is an empty string in the base class
- getEntity(): Returns the VHDL entity declaration of the current component
- getComponent(): Return the VHDL component declaration of the current component
- getInstance(): Returns the VHDL instance declaration of the current component.
- toVHDL(): Returns the complete VHDL implementation of the current component as a string

Every component in this class is supposed to be implemented in classes that inherit it else a NotImplementedError is raised by python.

#### ***4.1.2 Activation Class***

The Activation Class is the parent class for the sigmoid and tanh activation classes of the library. It defines the entity, component, and instance declarations of the activation function components and lets each activation function class cater only for the toVHDL() functions.

#### ***4.1.3 Gate Class***

The Gate class allows users to create the different gates used in an LSTM cell. The gates have a common structure and vary mainly by their activation functions. Thus, this class takes a gate parameter as a string to determine which activation function and for each gate and what name to give it.

#### ***4.1.4 LSTM Cell Class***

This class creates an LSTM cell by creating the gates and relevant arithmetic components. It allows the user to specify the weights and biases for each gate and calls their `writeToFile()` methods to generate the VHDL implementations for each gate.

#### ***4.1.5 LSTM Unit Class***

This class creates the LSTM network by spawning LSTM cells based on the number of inputs. It provides the inputs, weights, and biases for each LSTM cell object and calls their `writeToFile()` methods to create them. It also generates the configuration file to be used by the `.vhd` components.

#### ***4.1.6 Model Class***

The `.vhd` components use integers for computation. However, inputs to a neural network can be fractions. Thus, the model class allows the users to choose the level of precision they want to use and scales inputs according to produce integers and convert them to binary to be communicated via microcontroller and FPGA pins. The `bin2num` component performs the conversion of the binary input to integers for inferencing and the `num2bin` component converts the output to binary for transmission through FPGA pins. Corresponding `num2bin` and `bin2num` functions perform the same operations on the microcontroller. The model class instantiates the LSTM unit and communication modules.



## Chapter 5: Testing

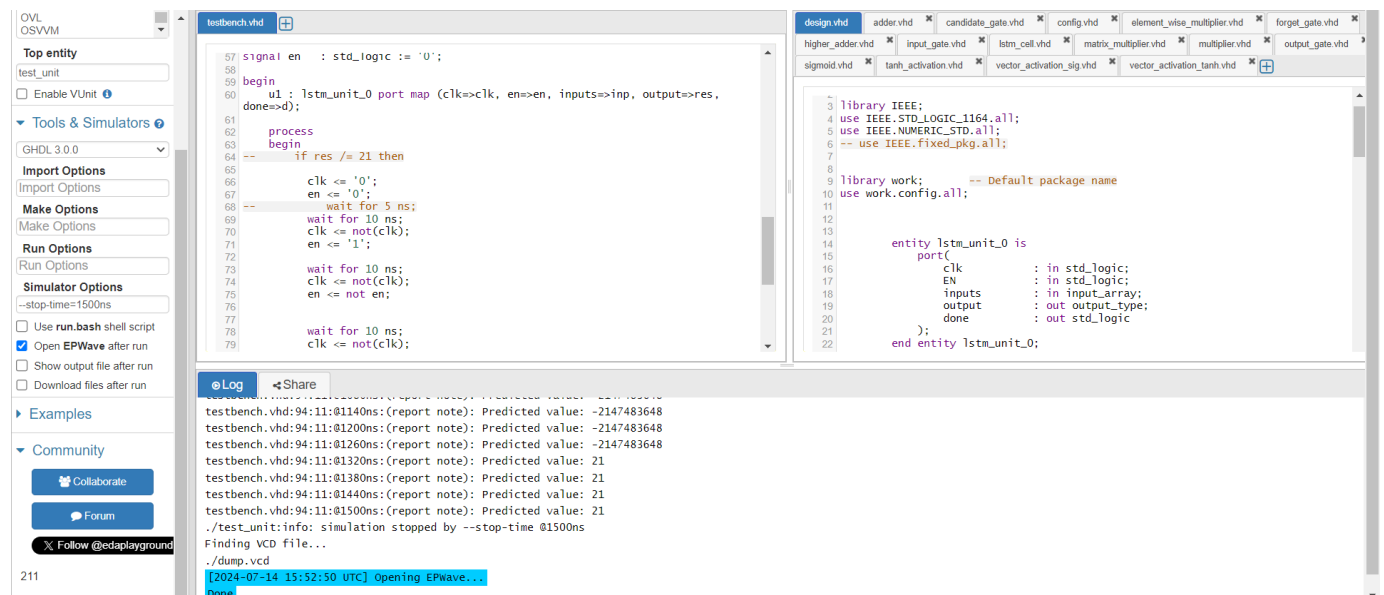
This chapter describes the testing approach for the solution and the results. Testing was carried out to ensure the library worked as expected and that the code it output was correct.

### 5.1 Unit Testing

Unit tests enable designers to perform small test cases on different parts of a system to ensure proper functioning. They also help in verifying that a solution produces expected results given specific inputs. During the development phase, classes were written and tested one at a time to ensure they worked as expected.

### 5.2 Simulation Testing

Simulations were carried out on the outputs of the different classes. Simulations served two primary purposes: to ensure that the VHDL code outputted by the different classes were syntactically and semantically correct. Test benches were handwritten for the generated hardware code and simulated using EDA Playground [12]. The outputs were printed using the report function and studied under predefined inputs to ensure correctness.



```
57 signal en : std_logic := '0';
58
59 begin
60   u1 : lstm_unit_0 port map (clk=>clk, en=>en, inputs=>inp, output=>res,
61     done=>d);
62
63   process
64   begin
65     if res /= 21 then
66       clk <= '0';
67       en <= '0';
68       wait for 5 ns;
69       wait for 10 ns;
70       clk <= not(clk);
71       en <= '1';
72
73       wait for 10 ns;
74       clk <= not(clk);
75       en <= not en;
76
77       wait for 10 ns;
78       clk <= not(clk);
79
80     end process;
```

```
3 library IEEE;
4 use IEEE.STD_LOGIC_1164.all;
5 use IEEE.NUMERIC_STD.all;
6 use IEEE.Fixed_pkg.all;
7
8
9 library work;
10 use work.config.all;
11
12
13
14
15
16 entity lstm_unit_0 is
17   port(
18     clk      : in std_logic;
19     EN       : in std_logic;
20     inputs   : in input_array;
21     output   : out output_type;
22     done     : out std_logic
23   );
24 end entity lstm_unit_0;
```

testbench.vhd:94:11:@1140ns: (report note): Predicted value: -2147483648  
testbench.vhd:94:11:@1200ns: (report note): Predicted value: -2147483648  
testbench.vhd:94:11:@1260ns: (report note): Predicted value: -2147483648  
testbench.vhd:94:11:@1320ns: (report note): Predicted value: 21  
testbench.vhd:94:11:@1380ns: (report note): Predicted value: 21  
testbench.vhd:94:11:@1440ns: (report note): Predicted value: 21  
testbench.vhd:94:11:@1500ns: (report note): Predicted value: 21  
./test\_unit:info: simulation stopped by --stop-time @1500ns  
Finding VCD file...  
./dump.vcd  
[2024-07-14 15:52:50 UTC] Opening EPWave...  
Done

Figure 18 Simulation of LSTM Unit in EDA Playground

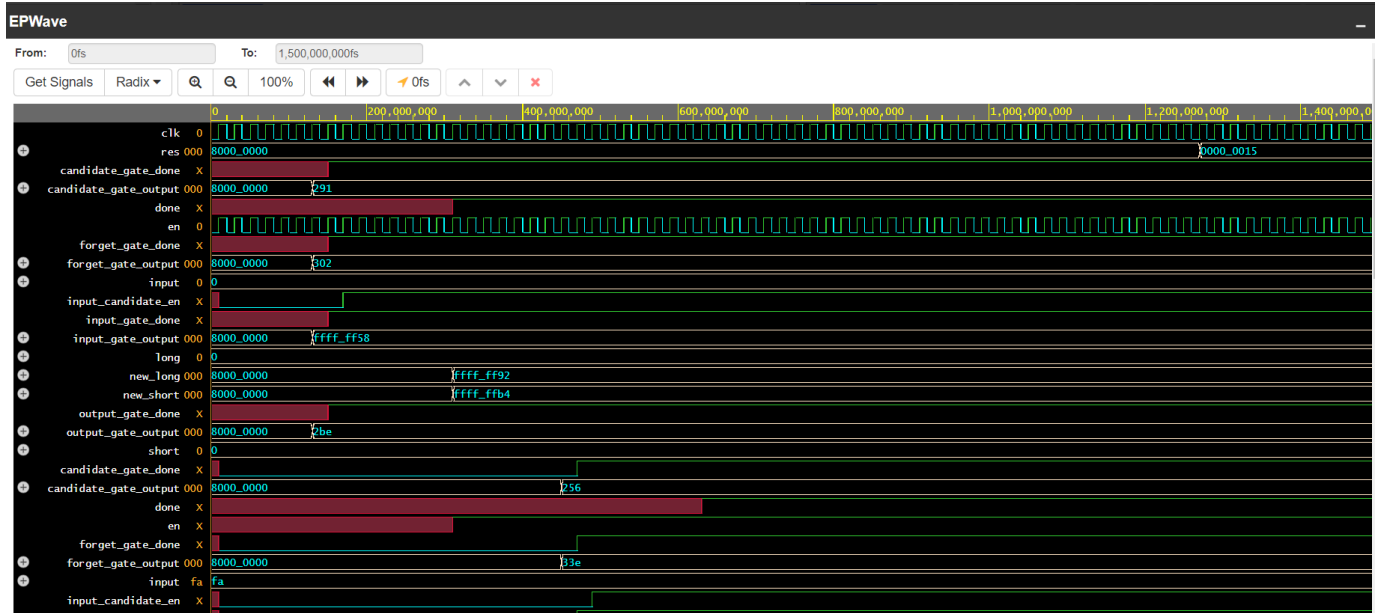


Figure 19 Simulation results for LSTM Unit

### 5.3 Synthesis Testing

Not all designs written in VHDL are synthesizable. To ensure that the designs will work on any FPGA, no vendor-specific libraries or constructs were used. For testing purposes, the design was tested by synthesizing them in vivado 2018.3, with the Xilinx Basys3 FPGA as target device. Below is the result of synthesizing the LSTM Unit component with four inputs.

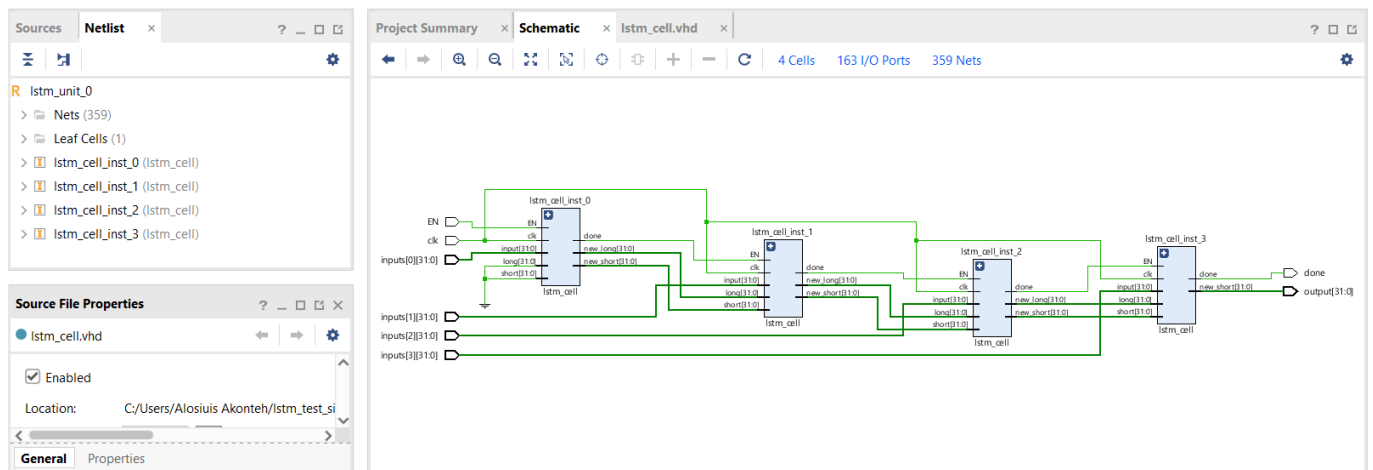


Figure 20 Synthesis Test Results for LSTM Unit and subcomponents

The table below lists all tested components and their synthesis states. The LSTM unit was synthesized with a base case where inputs are non-array types.

<b>Component</b>	<b>Synthesis Test Progress</b>
<b>Adder</b>	Complete
<b>Multiplier</b>	Complete
<b>Higher Bias Adder</b>	Complete
<b>Matrix Multiplier</b>	Complete
<b>Element-wise Multiplier</b>	Complete
<b>Activation</b>	Complete
<b>Vector Activation</b>	Pending
<b>Gate</b>	Complete
<b>LSTM Cell</b>	Complete
<b>LSTM Unit</b>	Complete
<b>Communication Module</b>	Pending

## **Chapter 6: Conclusion**

This paper's objective was to provide a solution to running LSTM recurrent neural networks on FPGA systems. With the library created, designers will be able to easily run their LSTM inferences without having to hard code VHDL components or redesign, they would only format their network parameters as specified in the guide provided in the library.

### **6.1 Limitations**

In the process of creating the library, some limitations encountered were:

- The manual nature of testing made the design iteration slow
- Resource constraints of available hardware made testing of array inputs impossible as the basys3 FPGA is primarily designed for educational purpose

### **6.2 Recommendations and Future Work**

This solution is an addition to a previous project that enabled the use of artificial neural networks on FPGA and provides a good continuation for the implementation of more complex neural networks that build on LSTMs such as convolutional LSTMs and LSTM with attention.

Some future considerations for exploration are:

- The design only allows for regular LSTMs and not bi-LSTMs and can be extended to support other kinds of LSTMs
- The library can be extended to support data type pruning to minimize resource need based on expected network inputs and output ranges if known

## References

- [1] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine Learning at the Network Edge: A survey," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–37, Oct. 2021, doi: 10.1145/3469029.
- [2] N. D. Gundi, T. Shabanian, P. Basu, P. Pandey, S. Roy and K. Chakraborty, "EFFORT: A Comprehensive Technique to Tackle Timing Violations and Improve Energy Efficiency of Near-Threshold Tensor Processing Units," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 10, pp. 1790-1799, Oct. 2021, doi: 10.1109/TVLSI.2021.3106858.
- [3] Lacey, Griffin, Graham W. Taylor and Shawki Areibi. "Deep Learning on FPGAs: Past, Present, and Future." ArXiv abs/1602.04283 (2016). [4] B. K. Chakrabarti, "Neural networks," *Current Science*, vol. 68, no. 2, pp. 153-155, 1995
- [5] M. Isaksson, "Four Common Types of Neural Network Layers," *Towards Data Science*, 6 June 2020. [Online]. Available: <https://towardsdatascience.com/four-common-types-of-neural-networklayers-c0d3bb2a966c>. [Accessed April 2021]
- [6] Zarzycki, K., & Ławryńczuk, M. (2021). LSTM and GRU Neural Networks as Models of Dynamical Processes Used in Predictive Control: A Comparison of Models Developed for Two Chemical Reactors. *Sensors*, 21(16), 5625. <https://doi.org/10.3390/s21165625>
- [7] A. Suresh, B. N. Reddy and C. Renu Madhavi, "Hardware Accelerators for Edge Enabled Machine Learning," 2020 IEEE REGION 10 CONFERENCE (TENCON), Osaka, Japan, 2020, pp. 409-413, doi: 10.1109/TENCON50793.2020.9293918.
- [8] H. Mohammadi Makrani, Z. He, S. Rafatirad and H. Sayadi, "Accelerated Machine Learning for On-Device Hardware-Assisted Cybersecurity in Edge Platforms," 2022 23rd

International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 2022, pp. 77-83, doi: 10.1109/ISQED54688.2022.9806150.

[9] A. Suresh, B. N. Reddy and C. Renu Madhavi, "Hardware Accelerators for Edge Enabled Machine Learning," 2020 IEEE REGION 10 CONFERENCE (TENCON), Osaka, Japan, 2020, pp. 409-413, doi: 10.1109/TENCON50793.2020.9293918.

[10] D. Holanda Noronha, K. Gibson, B. Salehpour and S. J. E. Wilton, "LeFlow: Automatic Compilation of TensorFlow Machine Learning Applications to FPGAs," 2018 International Conference on Field-Programmable Technology (FPT), Naha, Japan, 2018, pp. 393-396, doi: 10.1109/FPT.2018.00082.

[11] S. Han, J. Pool, J. Tran and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," Advances in neural information processing systems, pp. 1135-1143, 2015.

[12] Doulos, Ltd , [Online]. Available: <https://www.edaplayground.com>. [Accessed 2024].