

**ASHESI UNIVERSITY**

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

**CAPSTONE PROJECT**

B.Sc. Computer Engineering

##### Alosius Akonteh

**2024**

**ASHESI UNIVERSITY**

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

**CAPSTONE PROJECT**

Capstone project submitted to the Department of Engineering, Ashesi University in partial fulfillment of the requirements for the award of a Bachelor of Science degree in Computer Engineering.

##### Alosius Akonteh

**202**

**DECLARATION**

I hereby declare that this capstone is the result of my own original work and that no part of it has been presented for another degree in this university or elsewhere.

Candidate’s Signature:

Candidate’s Name:

Alosius Akonteh

Date:

31/08/2024

I hereby declare that the preparation and presentation of this capstone were supervised in accordance with the guidelines on supervision of capstone laid down by Ashesi University.

Supervisor’s Signature: ……………………………………………………………………………………………

Supervisor’s Name: ……………………………………………………………………………………………

Date:

…………………………………………………………………………………………

**Acknowledgements**

To my supervisor, Dr Nathan Amanquah, whose encouragement, academic advice, and detailed constructive feedback helped me undertake this project. I am grateful to my colleagues and mentors whose guidance and exhortation have brought me this far. I am most grateful to my God who granted me the strength and grace to undertake this project.

##### Abstract

Machine learning at the edge has grown in popularity due to its low latency and cost and its efficiency for IoT applications. Field-programmable gate arrays (FPGAs) are often used as edge devices to perform inference; however, the development process is not easy to complete quickly due to the required knowledge of hardware description language and Digital System Design. This paper proposes a solution to enable the use of Long Short-Term Memory Recurrent Neural Networks on FPGAs without extensive HDL and Digital System design knowledge by generating the HDL code for users based on their LSTM parameters.

Table of Contents

[Chapter 1 - Introduction 1](#_Toc173279895)

[The Problem 2](#_Toc173279896)

[The Background 3](#_Toc173279897)

[How the Problem will be addressed 4](#_Toc173279898)

[*Overview of Remaining Chapters* 5](#_Toc173279899)

[Chapter 2 – Related Work 6](#_Toc173279900)

[2.1 Neural Networks - A Brief Overview 6](#_Toc173279901)

[2.2 Machine Learning Acceleration - Hardware 9](#_Toc173279902)

[2.3 Machine Learning Optimization - Software 10](#_Toc173279903)

[2.4 Hardware Description Language Generation – Software 11](#_Toc173279904)

[Chapter 3: System Requirements and Design 12](#_Toc173279905)

[3.1 Proposed Solution 12](#_Toc173279906)

[3.2 System Requirements 13](#_Toc173279907)

[3.3 Use Case 14](#_Toc173279908)

[3.4 Operating Environment 15](#_Toc173279909)

[3.5 Design Constraints and Scope 15](#_Toc173279910)

[3.6 Design Overview 15](#_Toc173279911)

[3.7 Number Format 16](#_Toc173279912)

[3.8 Configuration Module 17](#_Toc173279913)

[3.9 Communication Module 17](#_Toc173279914)

[3.10 LSTM Unit Module 18](#_Toc173279915)

[3.9.1 LSTM Cell 20](#_Toc173279916)

[3.9.2 Activation 21](#_Toc173279917)

[3.9.3 Forget Gate 22](#_Toc173279918)

[3.9.4 Input Gate 22](#_Toc173279919)

[3.9.5 Candidate State 23](#_Toc173279920)

[3.9.6 Cell State Update 23](#_Toc173279921)

[3.9.7 Output Gate 24](#_Toc173279922)

[3.9.8 Hidden State Update 24](#_Toc173279923)

[3.9.9 Matrix Multiplier 25](#_Toc173279924)

[3.9.10 Higher Bias Adder 25](#_Toc173279925)

[3.9.11 Adder 25](#_Toc173279926)

[3.9.12 Multiplier 26](#_Toc173279927)

[3.9.13 Element-wise multiplier 26](#_Toc173279928)

[3.9.14 FPGA Controller 27](#_Toc173279929)

[Chapter 4: Implementation 28](#_Toc173279930)

[4.1 Class Hierarchy 30](#_Toc173279931)

[4.1.1 Component Class 30](#_Toc173279932)

[4.1.2 Activation Class 30](#_Toc173279933)

[4.1.3 Gate Class 30](#_Toc173279934)

[4.1.4 LSTM Cell Class 31](#_Toc173279935)

[4.1.5 LSTM Unit Class 31](#_Toc173279936)

[4.1.6 Communication Class 31](#_Toc173279937)

[Chapter 5: Testing 32](#_Toc173279938)

[5.1 Unit Testing 32](#_Toc173279939)

[5.2 Simulation Testing 33](#_Toc173279940)

[5.3 Synthesis Testing 35](#_Toc173279941)

[Chapter 6: Conclusion 38](#_Toc173279942)

[6.1 Limitations 38](#_Toc173279943)

[6.2 Recommendations and Future Work 38](#_Toc173279944)

# 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, and receive results on this data to inform decisions such as turning on an air conditioner based on environmental temperature. 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 must 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. Three main factors are of concern for an IoT system: the data transfer speed from the IoT sensors to the processing device, the processing speed of the processing device, and the security and privacy of transferred data.

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. Also, even when data is not exposed, data transfer patterns raise privacy issues. 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, while removing the security/privacy issues. Previous attempts to address this problem involved using Raspberry Pi boards as edge devices [1] to perform computations and another involved creating a library to enable edge computations on Field Programmable Gate Arrays boards [2] which will be discussed later in this paper. The former consumes quite a bit of power making it less appealing as an alternative. The latter has been implemented successful for Artificial Neural Networks. The goal of this is to create tools for accelerated machine learning at the edge. It 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. The FPGA is configured using a hardware description language. The hardware will be interfaced with the system that collects data and will perform computations necessary for inference. For this work, we will be making use of Long Short-Term Memory Recurrent Neural Networks.

# The Problem

In the artificial intelligence and machine learning industries, the latency of inference-making and input data transfer 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 delay. [3] 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 some IoT edge computing environments, data will be transferred from IoT sensors to a computing unit, in this case, the FPGA board which speeds up computation. This is different from the classical approach to machine inference as this approach requires knowledge of Hardware Description Language to design the inference computation engine making the development process difficult to complete in a short time. A tool to automatically generate the LSTM computation engine in a hardware description language is desirable

## 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 [4] 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 [5]. 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. Also, the parallel processing on FPGAs is much faster than on a conventional CPU. The approach is cheap given that the FPGA 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 users to reprogram the FPGA at will for varying machine learning workloads. The program will be designed for specific algorithms and will be documented to make updating easy with the advent of new machine learning algorithms. The focus of this project is to convert a machine learning algorithm to a hardware description language to run on an FPGA. The algorithm used in this paper is the LSTM which is a neural network. This illustrates the potential for neural network algorithms to run on FPGAs.

#### *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 for 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 learning 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. [6].

Different networks exist namely, fully connected and partially connected [7]. 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. For example, if given the word “John” and asked to guess what comes next, it would be difficult due to the lack of context. On the other hand, it would be easier to guess the next word if the sentence given was “John is a citizen of …” The second example provides context and would narrow the prediction to a place rather. LSTMs capture such context in sequential data to better inform predictions. The context is better captured if the four words are considered (four-time steps back) rather than a single word as in the first example.

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 6 below shows an illustration of an LSTM cell. LSTMs have four gates or compartments through which inputs are passed before predictions are made.

The forget gate helps the LSTM forget information that is no longer relevant to the network. It does this by determining the percentage of the long-term memory of the network to retain based on the current input. A sigmoid activation function is used to keep the input between zero and one to prevent values within the network from growing extremely large as they would influence the computations through which the network learns and adapts its weights and biases to make accurate predictions.

The candidate gate uses the long-term memory and current input to compute a new potential short-term memory. It uses a tanh activation function to scale the input to a range between negative one and positive one. The tanh activation allows the LSTM cell to preserve the sign of the computed values which could be negative. A sigmoid activation would lose the sign if any.

The input gate determines how much of the new potential long-term memory to remember.

It uses a sigmoid activation function just like the forget gate.

The output gate determines how much of the new short-term memory (the output of the LSTM cell) to remember. It uses a sigmoid activation function to compute the percentage of short-term memory to remember.

Thus, the LSTM cell starts off with a previous short and long-term memory. It passes the input and short-term memory to the forget gate which determines how much of the long memory to remember. Simultaneously, it passes these same input and short-term memory to the candidate and input gates which respectively determine the new potential long-term memory and percentage of potential long-term memory to retain. The output gate receives the same input to determine how much of the new short-term memory is needed. The activation functions are sigmoid for gates that compute percentages and tanh for gates that compute values that could be negative. The gates are similar in structure but vary in their weights and biases which are optimized during the training phase of the network. It is important to note that the gates do not rely on one another and thus can perform their computations in parallel.

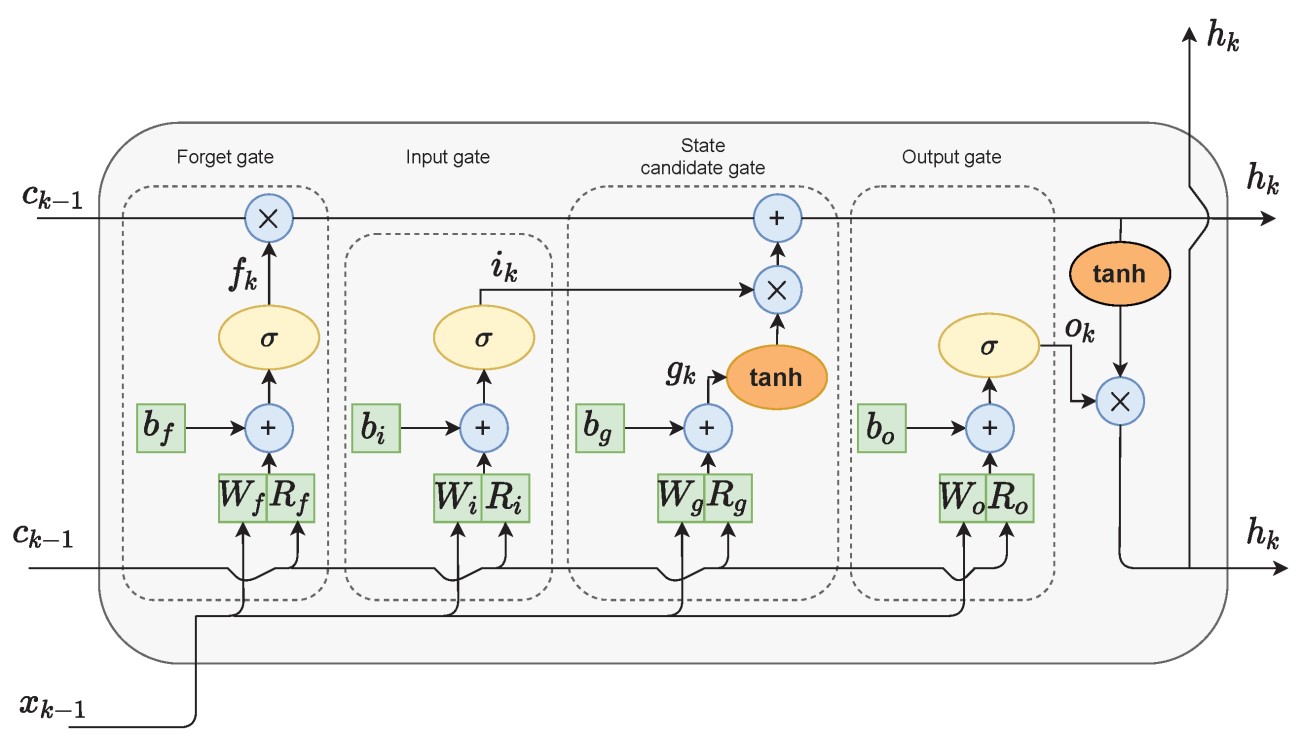


Figure 1 Structure of an LSTM Cell [8]

## 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 [9]. This section explores different machine learning hardware accelerators and compares them to the proposed accelerator. In [10] 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 and an FPGA structure allowing for high computational power. A softcore CPU is one implemented using logical synthesis in a hardware description language. The authors used machine learning algorithms implemented in C/C++ and converted them to VHDL using the Vivado. 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 PC based machine learning. More so, they employ hardware that includes a softcore 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. The work in [11] 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 [12] to convert numerical computation models in TenorFlow 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 [13], 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 [14]. 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

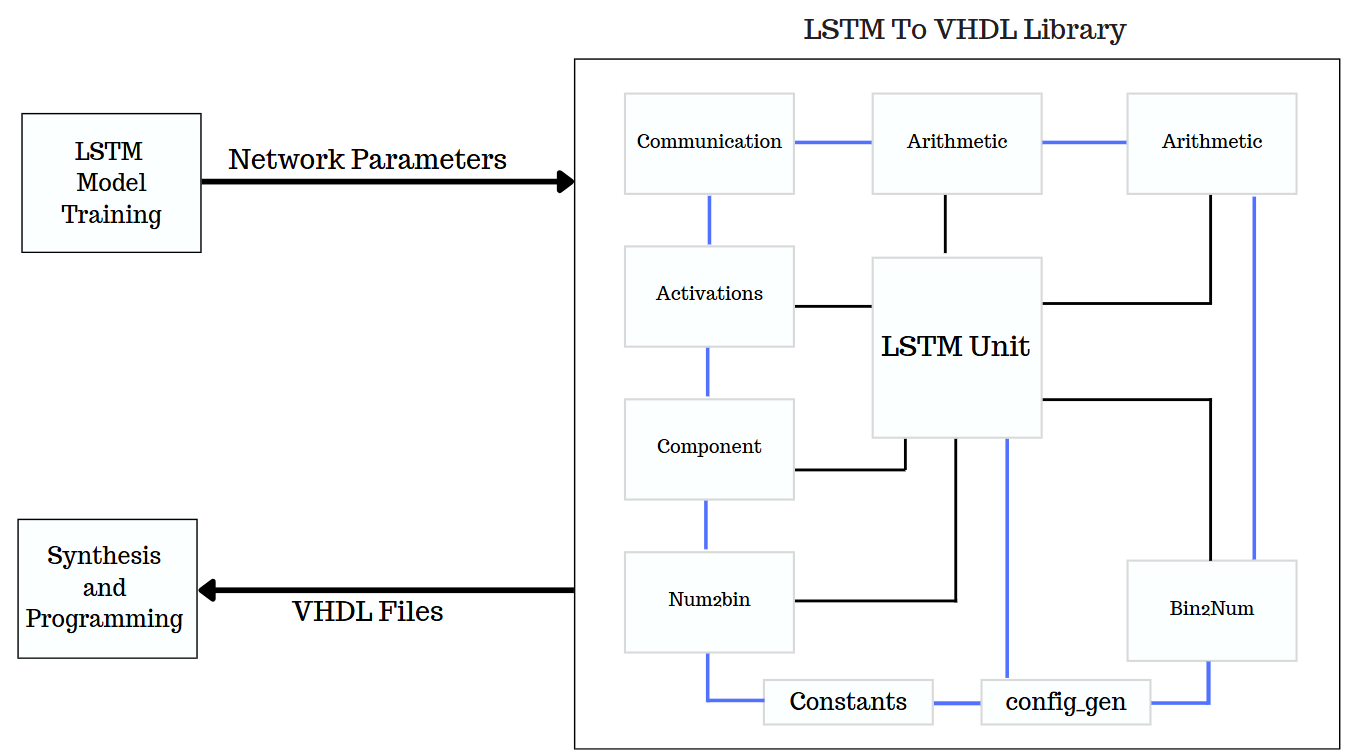


Figure 2 High-Level Overview of Solution

The paper proposes a framework for enabling the use of machine learning algorithms on

FPGA for inference as illustrated in Figure 1. 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.

## 3.2 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 give the user 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 pre-trained 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 shows an illustration of the process.

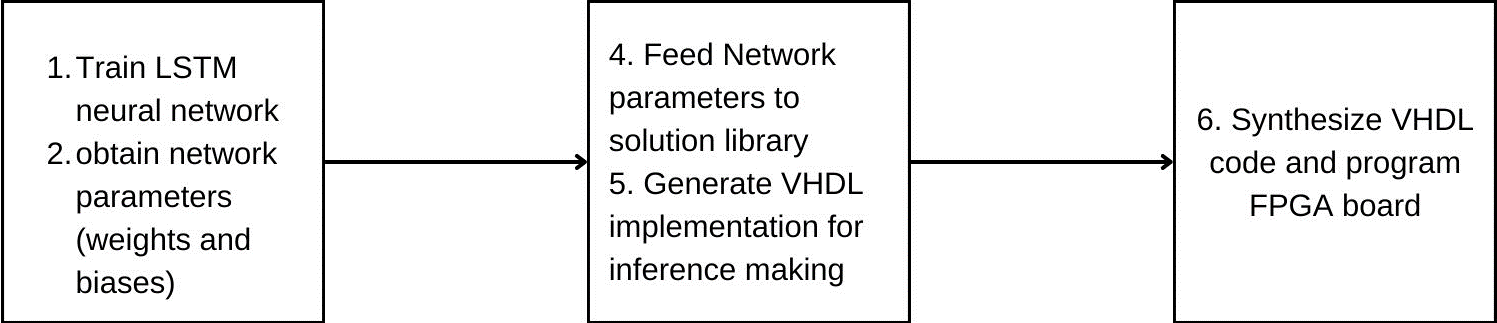


Figure 3 Block Diagram Illustration of Process

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 the microcontroller for the testing phase. Python was chosen because it is beginner-friendly and easy to integrate into 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. The MCU alone would not be desirable due to its lower computation power and inability to perform parallel operations simultaneously. 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 with CPython installed, the reference implementation of the Python programming language due to its wide 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 be responsible for moving 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 will determine when the microcontroller sends and receives data, and will also determine when inference computations begin on the FPGA. Figure 3 shows an illustration of the different subsystems.

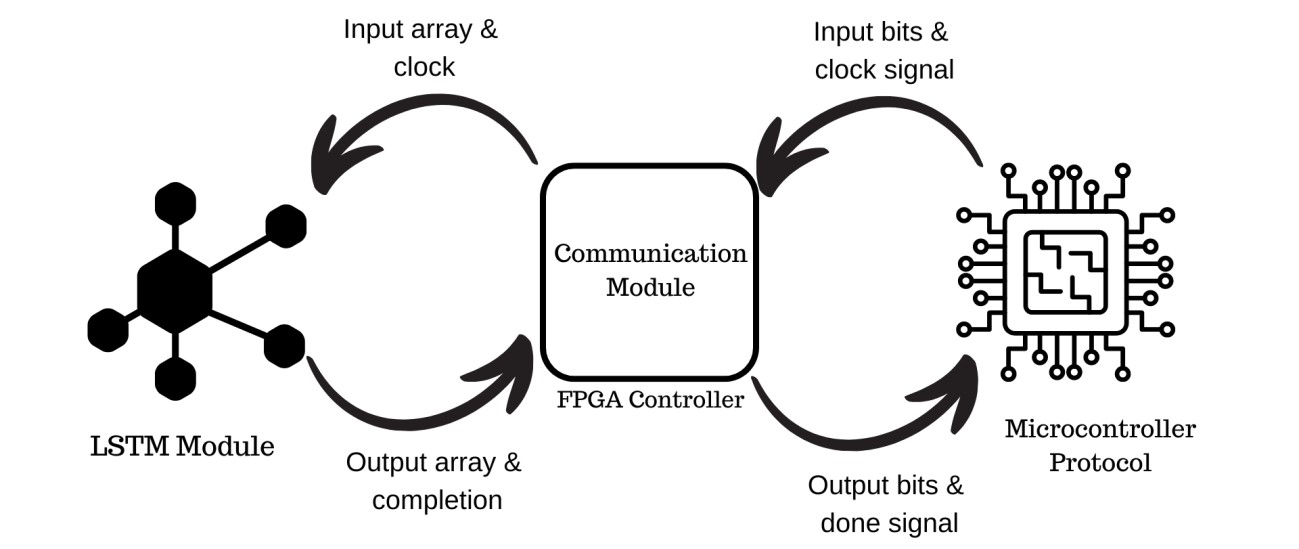


Figure 4 Solution Subsystem Interactions

## 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 data types used by the system and creates types based on the nature of the inputs.

# 3.9 Communication Module

A yellow rectangle with black text

Description automatically generated

Figure 5 Communication Module Interface with 10 bits inputs and outputs diagram generated using Teros HDL

The communication module handles communication between the FPGA and microcontroller unit. It receives control signals and inputs. It receives inputs as bits and converts them to integers before passing them to the LSTM unit for computation. It converts inference results to bits for transmission to the MCU as shown in Figure 4. It receives a two bit control signal to decide which task to perform at a given time as shown in Figure 5.

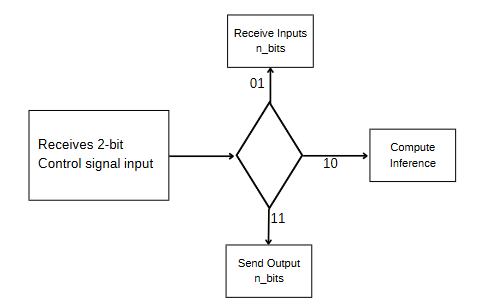


Figure 6 Communication Module Operation Diagram

It houses the binary-to-integer and integer-to-binary units alongside the input capture module that stores all the inputs of the LSTM model until they are all received and ready to be passed to the LSTM Unit for inference. The binary to integer, integer to binary, and input capture interfaces are shown in Figures 6, 7, and 8 respectively.

A yellow and black text

Description automatically generated

Figure 7 Binary to integer conversion interface diagram generated using Teros HDL

A yellow square with black text

Description automatically generated

Figure 8 Input Picker Interface diagram generated using Teros HDL

A yellow square with black text

Description automatically generated

Figure 9 Integer to Binary Conversion Interface diagram generated using Teros HDL

# 3.10 LSTM Unit Module

A yellow rectangle with black text

Description automatically generated

Figure 10 LSTM Unit Interface diagram generated using Teros HDL

This module is responsible for the computation of results using input data. It receives data from the communication module and performs computations. It returns a completion signal and the result of the computation as shown in Figure 9.

A diagram of a computer

Description automatically generated

Figure 11 LSTM Unit Illustration

The LSTM Unit module is represented as a component consisting of LSTM cell components, as shown in Figure 10, 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.

The LSTM unit 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 cell.

### 3.9.1 LSTM Cell

A yellow rectangular sign with black text

Description automatically generated

Figure 12 LSTM Cell Interface diagram generated using Teros HDL

A diagram of a computer

Description automatically generated

Figure 13 LSTM Cell Flow diagram

This component represents LSTM cells in the network. Each cell has different gates, as shown in Figure 12, that handle different functions as explained in Chapter 2. Each cell serves as a housing for the gates where each computation happens. Each LSTM cell receives an input, the previous long and short term memories, as shown in Figure 11, and passes them to the adequate gates for computation. All gates have the structure illustrated in Figure 13 below.

A diagram of a machine

Description automatically generated

Figure 14 Gate Operation Diagram

### 3.9.2 Activation

Every gate has an activation function. 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/(1+𝑒−𝑥)
* Tanh: f(x) = tanh(x)

A yellow rectangle with black letters

Description automatically generated

Figure 15 Activation Function Interface

Due to the exponential nature of sigmoid and tanh activations, the activation functions are implemented as vectors with values within a range and increments specified by the user.

### 3.9.3 Forget Gate

A yellow rectangular sign with black text

Description automatically generated

Figure 16 Forget Gate Interface diagram generated using Teros HDL

The forget gate uses the current input and last short-term memory or hidden state, as shown in Figure 14, 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:

ft = σ(Wi \* xt + Wh \* ht-1 + bi\_f)

where Wi is the weight matrix connecting the input, xt, to the forget gate, Wh is the weight matrix connecting the previous short-term memory to the forget gate, ht-1 and bi\_f is the bias vector for the forget gate. The sigmoid activation function keeps the values between 0 and 1.

### 3.9.4 Input Gate

A yellow rectangle with black text

Description automatically generated

Figure 17 Input Gate Interface diagram generated using Teros HDL

The input gate determines how much of the new candidate cell state should be considered. It has a similar structure to the forget gate. It applies a sigmoid activation function to its result before outputting it.

it = σ(Wi \* xt + Wh \* ht-1 + bi\_i)

where Wi is the weight matrix connecting the input, xt, to the input gate, Wh is the weight matrix connecting the previous short-term memory to the input gate, ht-1 and bi\_i is the bias vector for the input gate. The sigmoid activation function keeps the values between 0 and 1, representing a percentage of the candidate cell state or long-term memory to remember.

### 3.9.5 Candidate State

A yellow rectangle with black text

Description automatically generated

Figure 18 Candidate Gate Interface diagram generated using Teros HDL

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, as shown in Figure 16, and applies a tanh activation function to its output. The tanh activation is used to preserve the notion of sign in the inputs rather than the sigmoid function which represents percentages in this application. The module will make use of the activation function implemented separately as a component to increase modularity.

Ct\_candidate = tanh(Wi \* xt + Wh \* ht-1 + bi\_c)

where Wi is the weight matrix connecting the input, xt, to the candidate state, Wh is the weight matrix connecting the previous short-term memory to the candidate state, ht-1 and bi\_c is the bias vector for the candidate state. The tanh activation function keeps the values between -1 and 1.

### 3.9.6 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.

Ct = ft \* Ct-1 + it \* Ct\_candidate

where Ct is the cell state/long-term memory, ft is the amount of the previous long-term memory to remember, Ct-1 is the previous long-term memory, it is how much of the new information to be added to memory and Ct\_candiate is the potential information to add to memory.

### 3.9.7 Output Gate

A yellow rectangle with black text

Description automatically generated

Figure 19 Output Gate Interface diagram generated using Teros HDL

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 as shown in Figure 17 and performs the computation below.

ot = σ(Wi \* xt + Wh \* ht-1 + bi\_o)

where Wi is the weight matrix connecting the input, xt, to the output gate, Wh is the weight matrix connecting the previous short-term memory to the output gate, ht-1 and bi\_o is the bias vector for the output gate. The sigmoid activation function keeps the values between 0 and 1.

### 3.9.8 Hidden State Update

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

ht = ot \* tanh(Ct)

where ht is the new hidden state, ot is the output weight vector from the output weight block passed through a sigmoid function and Ct is the current cell state.

### 3.9.9 Matrix Multiplier

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

A yellow rectangular sign with black text

Description automatically generated

Figure 20 Matrix Multiplier Interface diagram generated using Teros HDL

### 3.9.10 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.

A yellow rectangular sign with black letters

Description automatically generated

Figure 21 Higher Bias Adder Interface diagram generated using Teros HDL

### 3.9.11 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.

A yellow rectangular sign with black text

Description automatically generated

Figure 22 Adder Interface generated using Teros HDL

### 3.9.12 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.

A yellow rectangular sign with black letters

Description automatically generated

Figure 23 Multiplier Interface generated using Teros HDL

### 3.9.13 Element-wise multiplier

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

A yellow rectangular sign with black letters

Description automatically generated

Figure 24 Element-wise Multiplication Interface diagram generated using Teros HDL

### 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. The library is implemented in Python and when given approximate LSTM parameters, it generates the VHDL model for the LSTM network. The VHDL files will be able to run on FPGAs. 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

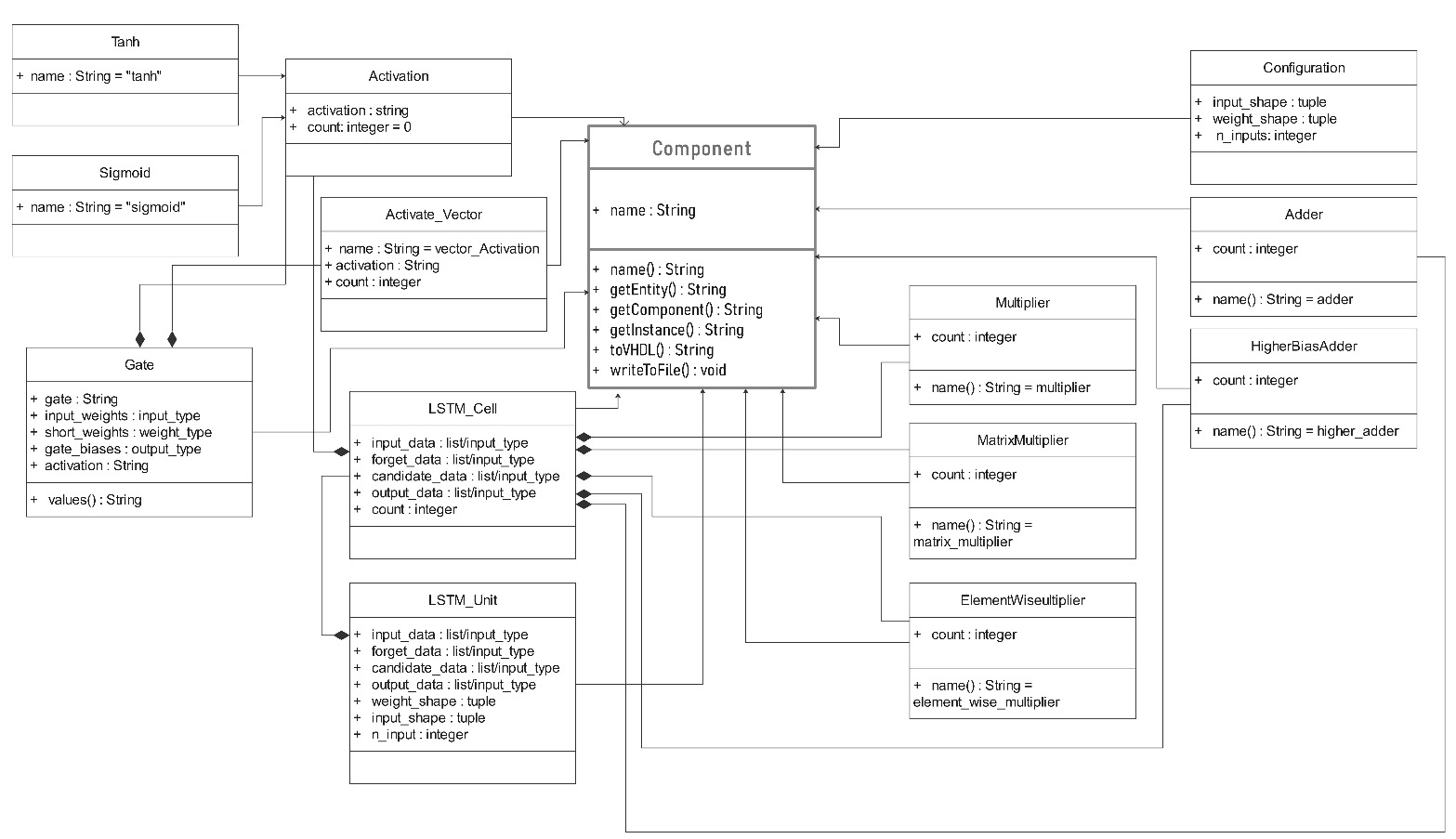


Figure 25 UML Diagram of Library

## 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 NotImplentedError 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() method.

### 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 Communication 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.

Once the user trains their LSTM model, they would extract the weights and biases of the respective gates and pass them to the communication module as a list of dictionaries. The communication class will also require them to specify the input and weight. If the inputs are integers, the shapes will be [1, 1]. Finally, the user would input the number of inputs the LSTM expects, the number of decimal places with which the system would work, the input resolution (number of bits), and the estimated input range for the activation functions.

# 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.

The library is used by creating an object of it and passing the parameters below:

* *nbits*: The bit resolution of the inputs and outputs; integer type
* *input\_range*: A list or tuple for the expected input range for the activation functions
* *accuracy*: The increment between values in the input range. It influences the accuracy of activation functions
* *dp*: The number of decimal places of accuracy for the inputs and outputs of the network
* *forget\_data*: A dictionary with the input weights, short memory weights, and gate biases. The parameters are either array type or numeric.
* *input\_data*: A dictionary with the input weights, short memory weights, and gate biases. The parameters are either array type or numeric.
* *candidate\_data*: A dictionary with the input weights, short memory weights, and gate biases. The parameters are either array type or numeric.
* *output\_data*: A dictionary with the input weights, short memory weights, and gate biases. The parameters are either array type or numeric.
* *n\_inputs*: The number of inputs or time steps of the network; integer type
* *input\_shape*: The dimensions of the input; list or tuple type
* *weight\_shape*: The dimensions of the weights of the network; list or tuple type
* *input\_range*: The estimated range of inputs and outputs; list or tuple type. This will be used to generate the activation functions

A sample use of the library is provided below:



Figure 26 Library Object Creation and parameters

## 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 codes outputted by the different classes were syntactically and semantically correct. Test benches were handwritten for the generated hardware code and simulated using EDA Playground [15]. The outputs were printed using the inbuilt VHDL report function and studied under predefined inputs to ensure correctness.

Figure 18 shows the VHDL implementation and testing of a network with inputs (0, 0.25, 0.5, 1.0). Below is the code for using the library in this case.

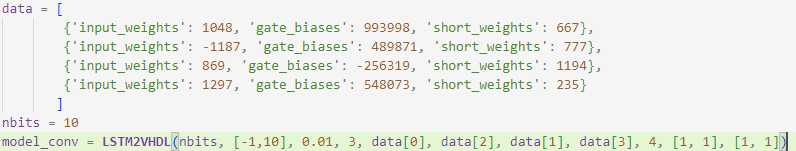


Figure 27 Sample use of library with inputs from pretrained LSTM model

With an expected prediction of 0, the console outputs a 7 which is with computations scaled by a power of 3 and a step of 0.01 for the activation function inputs, the decimal places chosen by the user. Thus, the actual prediction is 0.07 which is close to the obtained value and will be even closer with a higher number of decimal places and more fine-tuned precision for the activation functions.

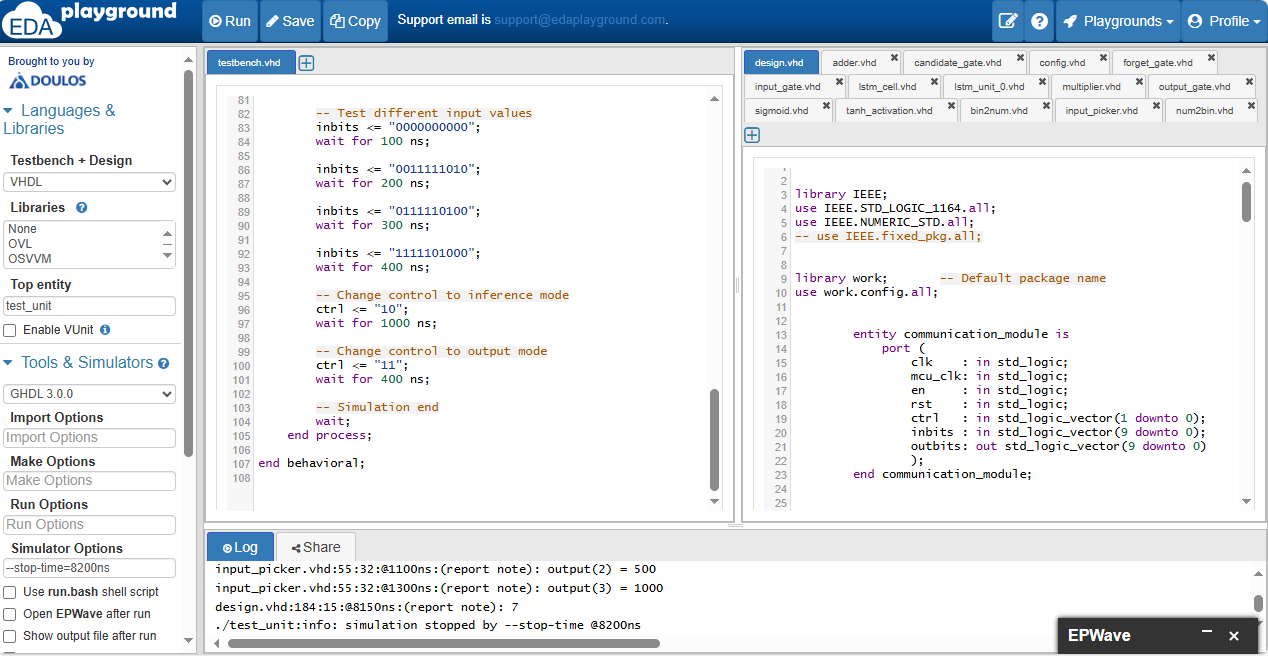


Figure 28 Simulation of LSTM Unit in EDA Playground

Figure 28 shows the results of testing the network in Figure 17. The run time is about 8.2 microseconds which is significantly faster than relying on remote servers for which data transfer could take times in the magnitudes of millimeters even for small inputs as the ones described above.

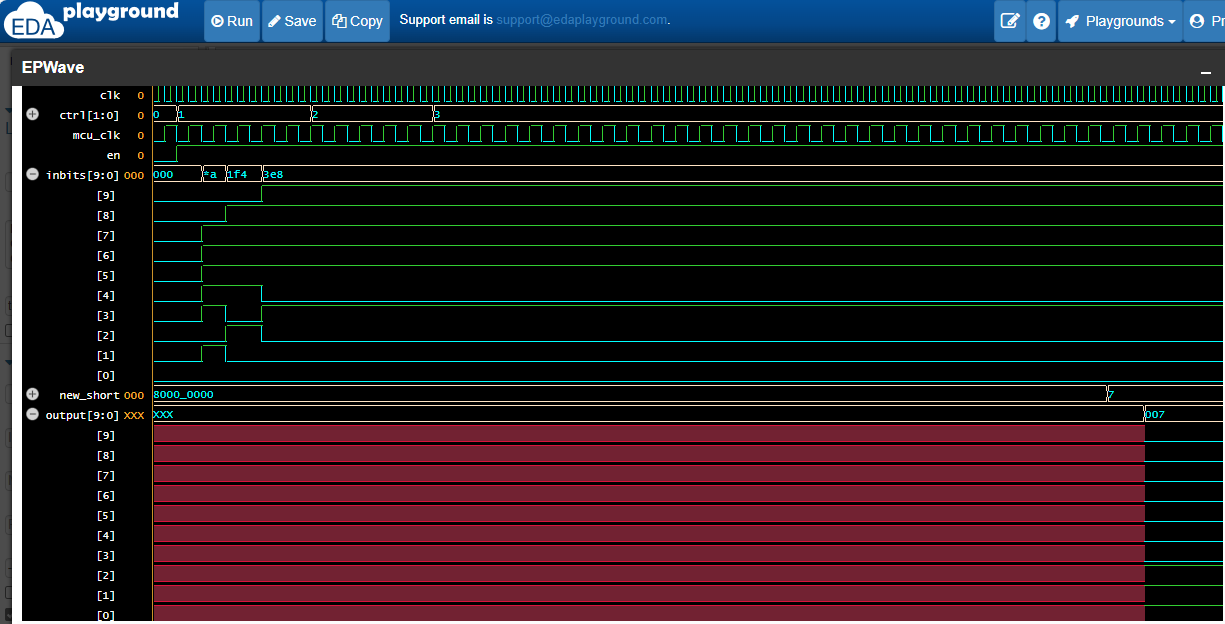


Figure 29 Simulation results for LSTM Unit

Figure 19 shows the signal diagram. The inputs are received using the *mcu\_clk* signal which indicates that the inputs can be received at the microcontrollers pace even though it is slower than the FPGA without needing to synchronize their clocks.

The *ctrl* signal determines which operation to be performed. The ouput is converted to bits and sent to the output bit vector to be sent to the microcontroller for decision making.

Inference generation was also tested with matrix parameters as shown.

A screenshot of a computer program

Description automatically generated

Figure 30 Sample LSTM Parameters and inputs

The expected output is [0.25518572 0.07367425] and Figure 31 shows the testbench and results from the generated LSTM Unit in VHDL with a precision of 0.01 for the activation function and 3 decimal places for every other quantity. With matrix parameters, the matrix multiplier, element-wise multiplier, and higher adder components are generated instead of the regular multipliers and adders. Also, the vector activation components are included to activate matrix inputs. The predicted values from the generated LSTM are 247 and 70 which are values scaled by a 10 raised to the number of decimal places specified. Thus, the predictions are [0.247 0.07] which are close to the actual values. Accuracy can be increased by modifying the accuracy and decimal places parameters.

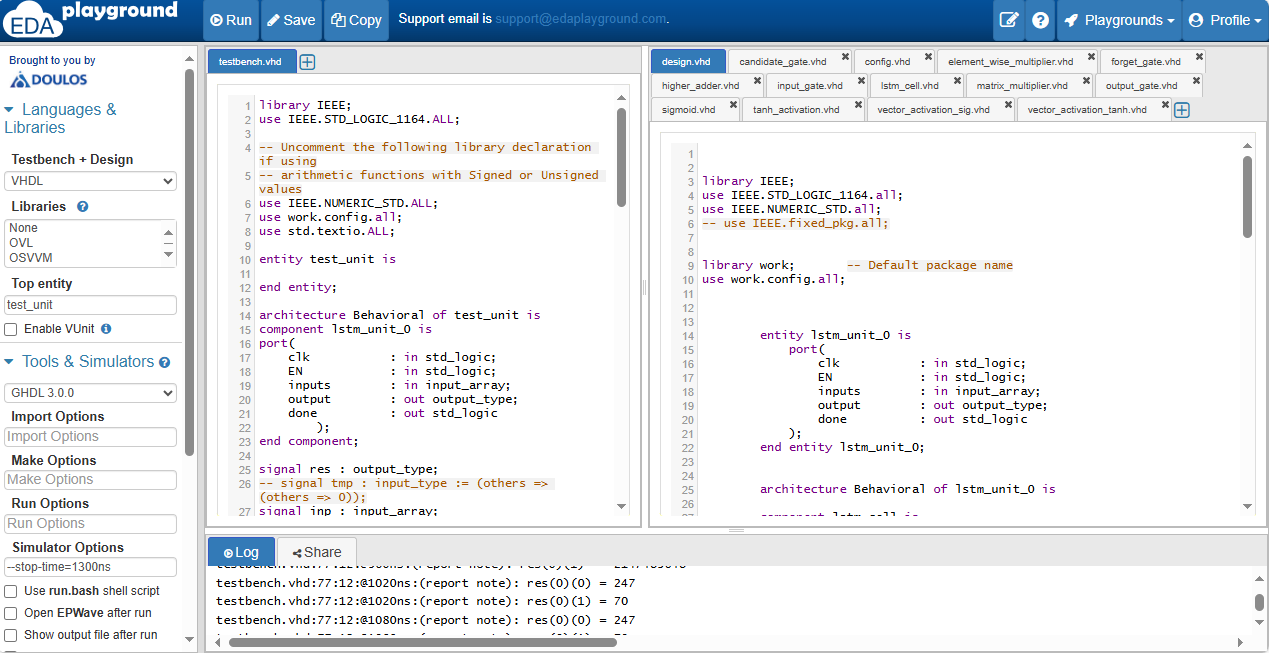


Figure 31 Testbench for LSTM with matrix parameters

## 5.3 Synthesis Testing

Not all designs written in VHDL are synthesizable. The VHDL components were synthesized in Vivado 2018.3 to ensure they would be work on hardware when constrained with input/outputs on FPGA. To ensure that the designs would 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.

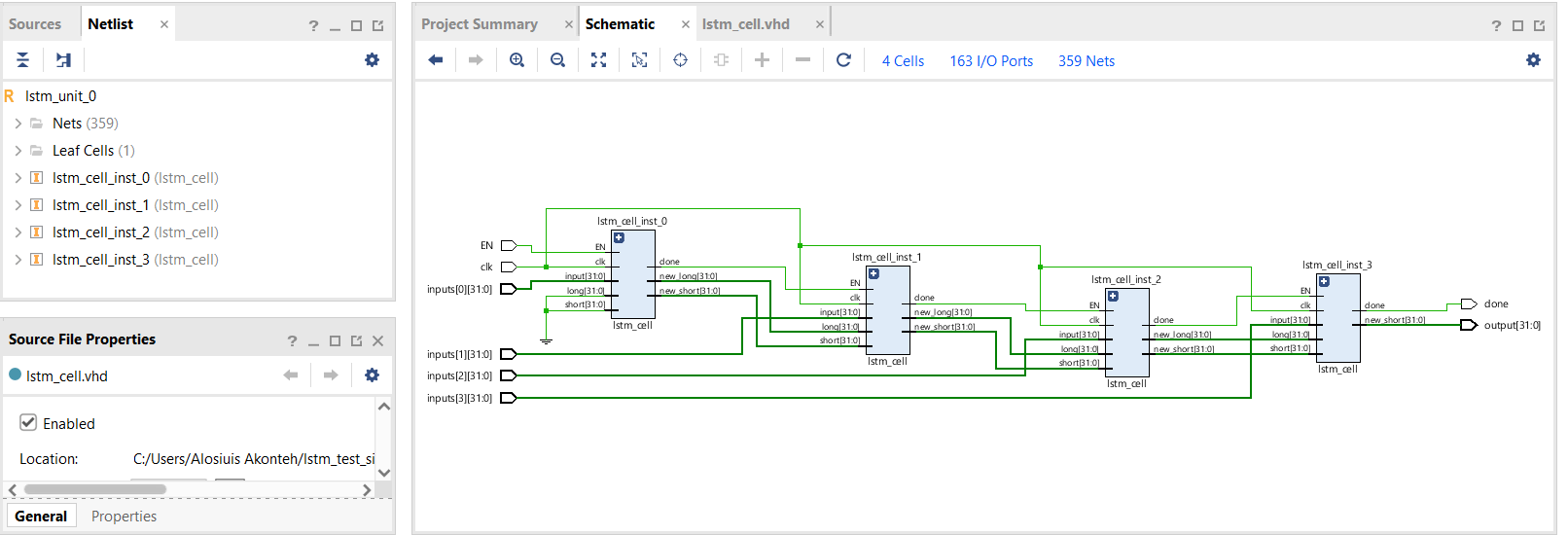
****

Figure 32 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 arrays and when inputs are non-array types.

|  |  |
| --- | --- |
| Component | Synthesis Test Progress |
| Adder | Complete |
| Multiplier | Complete |
| Higher Bias Adder | Complete |
| Matrix Multiplier | Complete |
| Element-wise Multiplier | Complete |
| Activation | Complete |
| Vector Activation | Complete |
| Gate | Complete |
| LSTM Cell | Complete |
| LSTM Unit | Complete |
| Binary to Integer | Complete |
| Integer to Binary | Complete |
| Input picker | Complete |
| Communication Module | Complete |

# Chapter 6: Conclusion

This paper**’s** objective was to provide a solution torunning 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 and the code will be generated for them. The only task left will be matching port declarations to the configuration file of the specific FPGA under use.

## 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 on the as the basys3 FPGA which is primarily designed for educational purpose. However, testbench tests were successful.

## 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. R. Mohandas, M. Bhattacharya, M. Penica, K. Van Camp and M. J. Hayes, "On the use of Deep Learning Enabled Face Mask Detection For Access/Egress Control Using TensorFlow Lite Based Edge Deployment on a Raspberry Pi," 2021 32nd Irish Signals and Systems Conference (ISSC), Athlone, Ireland, 2021, pp. 1-6, doi: 10.1109/ISSC52156.2021.9467841
2. Akotey, I. J. A. (2021, May 1). *Edge computing and machine learning on embedded systems*. https://hdl.handle.net/20.500.11988/858
3. 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.

1. 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.

1. Lacey, Griffin, Graham W. Taylor and Shawki Areibi. “Deep Learning on FPGAs: Past, Present, and Future.” ArXiv abs/1602.04283 (2016).
2. B. K. Chakrabarti, "Neural networks," Current Science, vol. 68, no. 2, pp. 153-155, 1995
3. 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]
4. K. Zarzycki and M. Ławryńczuk, “LSTM and GRU neural networks as models of dynamical processes used in predictive control: A comparison of models developed for two chemical reactors,” *Sensors*, vol. 21, no. 16, p. 5625, Aug. 2021, doi: 10.3390/s21165625.
5. 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.
6. 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.
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. 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.
9. 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.
10. P. Chimsuwan, P. Supnithi, W. Phakphisut and L. M. Min Myint, "Construction of LSTM model for total electron content (TEC) prediction in Thailand," *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, Chiang Mai, Thailand, 2021, pp. 276-279, doi: 10.1109/ECTI-CON51831.2021.9454881.
11. Doulos, Ltd , [Online]. Available: https://www.edaplayground.com. [Accessed 2024].