Implementation of Long Short-Term Memory Neural Networks in High-Level Synthesis Targeting FPGAs

Richa Rao

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Washington

2020

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

Abstract

Field programmable gate arrays (FPGAs) offer flexibility in programmable systems, making them ideal for hardware implementations of machine learning algorithms. The effectiveness of machine learning (ML) methods has been demonstrated successfully in particle physics computations, particularly in Large Hadron Collider (LHC) physics. Their use in FPGA hardware, however, has been restricted due to the complex implementation and significant resource demands. Thus, the need for FPGA resource estimation, as well as a means to simplify ML implementation on FPGAs is being fulfilled by HLS4ML [1] (High-Level Synthesis for Machine Learning). HLS4ML is a framework that translates traditional open-source machine learning package models into HLS, and thus maps neural networks directly onto an FPGA using HLS tools. Facilitating a drastic decrease in firmware development time against traditional VHDL/Verilog algorithms, HLS4ML increases accessibility across the user community. By understanding the mechanism of this framework, we implement a Long Short-Term Memory (LSTM) network targeting an FPGA. We take a Top Tagging LSTM model and translate it to HLS code. Further, using an HLS tool we obtain reports and analyze the overall latency and resources required by the model. The motivation for using LSTMs, its current state of development, and my personal work on the inclusion of this neural network into the HLS4ML framework are explained in this thesis.

CONTENTS

1. Outline

When we think about implementing a neural network on a hardware platform for application in real life scenarios, we realize that it has various aspects to it. There are pre-requisites that need to be understood well to go ahead with the implementation. In sections to come, we will discuss pre-requisites such as a basic understanding of neural networks and their types and the reason we prefer field programmable gate arrays to implement these networks.

Further, we will talk about a framework called HLS4ML that makes the task of putting a neural network onto an FPGA much easier with the help of High-Level Synthesis (HLS) and also discuss what HLS is.

Finally, we look at the applications of this project in the field of physics, post which we delve into the specifics of this thesis and how we can use the knowledge of the abovementioned pre-requisites and apply it to implement a long short-term memory neural network onto an FPGA.

2. Neural Networks

Neural networks can be thought of as a software implementation of the human brain. Similar to how humans learn and perform tasks such as being able to differentiate between a dog and cat, neural networks are also capable of "training" or learning and performing an "inference" or task.

Fig 1a. shows a typical neural network representation.

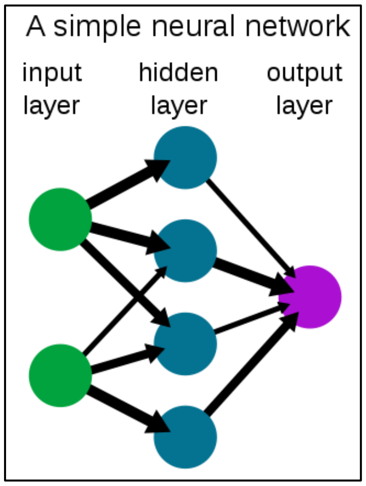


Fig 1a: Neural Network [16]

2.1 Neural Network Components

- Neurons: In Fig 1a, the green, blue, and purple spheres represent “neurons”. They are used to process information that is sent to a neural network.
- Layers: A simple network has at least one input layer, one output layer and one hidden layer. Deeper or more complicated neural networks can have several hidden layers.
- Weights and biases: The arrows shown in Fig 1a represent the weights. These weights represent the relative importance of the connection between neurons. Fig 1b shows the mathematical calculations between inputs, weights, biases and activation functions in a neural network.

output = f (Σ (weight * input) + bias)

f - Activation function
Σ - Summation
* - Multiplication

Fig 1b: Neural Network Mathematics

- Activations: These are activation functions such as sigmoid, tanh, softmax etc. that help keep values within a certain range in the neural network.
  - Sigmoid: Sigmoid Activation function outputs a value between 0 and 1 for any given input (Fig 1c).
  - Tanh: Tanh Activation function outputs a value between -1 and 1 for any given input and has a steeper gradient as compared to sigmoid (Fig 1c).
  - Softmax: Softmax Activation function outputs a probability distribution over the predicted classes in case of a classification neural network model (Fig 1d).

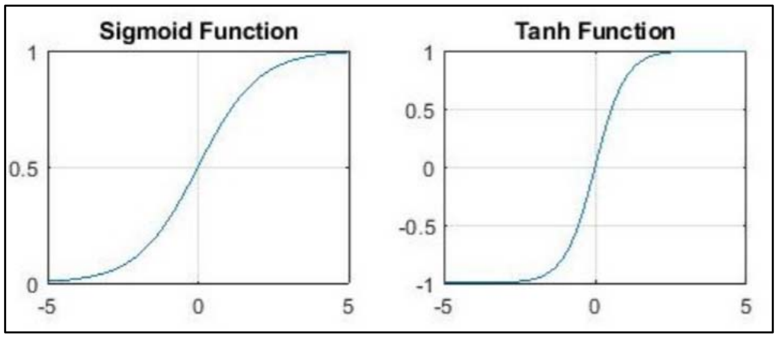


Fig 1c: Sigmoid and Tanh activation functions [17]

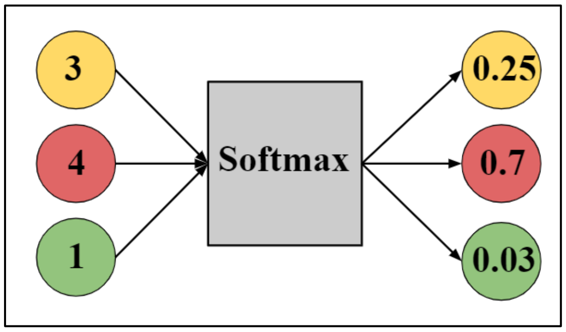


Fig 1d: Softmax activation function

2.2 Training and Inference

One of the most common methods of training a neural network is by providing it with a dataset along with the expected output for each input in the dataset. For example, a dataset consisting of labeled images of dogs and cats, can be given to a network. These images are sent from the input side of the neural network to the output side, passing through all the hidden layers. Once it reaches the output, the obtained value is compared with the expected value (which was made available to the network). An error is calculated which must be minimized in order to help the neural network predict values closer to the expected values. It is important to note that these neural networks are mathematical models and interact with layers within only with numbers.

Backpropagation is an algorithm that is used in training networks. In simple terms, at the end of each forward pass while training, an error/cost function is calculated based on the obtained output vs the expected output. This error is then used to adjust the weights and biases such that for a given input, we receive and output close to the expected output. This process of going back and adjusting the weights and biases is backpropagation and is done to reduce the error as much as possible. Once the network is trained, we can test its prediction accuracy with the process of inference. In the example stated above, we give as input, an unlabeled image of a dog or a cat and check if it is able to predict the right output.

2.3 Types of Neural Networks

There are different types of neural networks that are defined, each with its own strength. Different neural networks can deal with different applications and work with varying datasets. Some take images as inputs, while others prefer a sequence of inputs.

Some examples are:

- Deep Neural Networks (DNN): They have more than one hidden layer
- Recurrent Neural Networks (RNN): They are useful in predicting data that relies on context, for example- text generation.
- Convolutional Neural Networks (CNN): They work very well with images as inputs.

The work in this thesis is primarily based on a certain type of neural network, called the Long Sort-Term Memory neural network which is a modified version of an RNN. In the next section, we will look into the working of these neural networks.

3. Recurrent Neural Networks

Unlike traditional feedforward neural networks, Recurrent Neural Networks have a memory aspect to them. These neural networks can use their internal memory state to process inputs given in the form of a sequence. These are generally preferred in cases where the output depends on a series of inputs. Some common applications include natural language processing, text prediction, and speech recognition.

The term "Recurrent" in this neural network indicates that the same set of tasks are performed recurrently to each element in the input sequence in turn, and the output of each of these tasks plays a part in the overall prediction by the RNN. RNNs are currently the only class of artificial neural networks to possess this feature and are expected to deliver higher prediction accuracy for sequential data or in data where context matters, as in the case of text prediction.

A typical RNN architecture is shown in Fig 2.

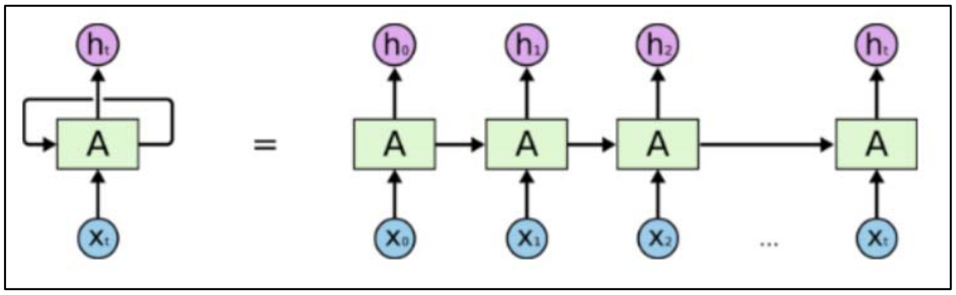


Fig 2: Unrolled RNN [12]

Although the above stated benefits stand true for RNNs in theory, practically vanilla RNNs are limited to looking back only few steps. This is attributed to how vanilla RNNs are trained.

As mentioned in section 2.2, backpropagation is a method used to train neural networks. Based on the type of neural network, this algorithm can be modified to best suit the network. In RNNs this training algorithm is called Backpropagation Through Time (BPTT) as it is applied to sequential data and we go back in timesteps to adjust the weights and biases. Conceptually, BPTT works by unrolling all input timesteps. Errors are calculated at each timestep and accumulated. Then the network is rolled back up and the weights are updated. For example, for an input sequence consisting of a thousand timesteps, a single weight update requires calculating thousand errors, one for each timestep. The amount by which the weights get updated is proportional to the partial derivative of the error function with respect to the current weight also called as the gradient. In case of vanilla RNNs, this gradient tends to get smaller over various training iterations and after a point becomes so vanishingly small that it barely updates the weights and the neural networks stops training completely. This is termed as the vanishing gradient problem.

Due to this, vanilla RNNs are rarely used in practical scenarios. To solve the problem of vanishing gradients, the RNN architecture can be modified. Two such architectures are called the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) respectively. In the HLS4ML framework, we have successfully implemented an LSTM and will move on to implement a GRU.

3.1 Types of RNNs

The two types of RNNs that are most widely used are Long Short-Term Memory RNNs (LSTMs) and Gated Recurrent Unit RNNs (GRUs).

Long Short-Term Memory networks – routinely called "LSTMs" – are a special kind of RNN, capable of overcoming the issues that arise due to long term dependencies. LSTMs are designed such that retaining information for prolonged time periods is the default setting for the RNN.

Unlike traditional RNNs, where each cell has an extremely simple structure as shown in Fig 3., LSTMs have gates that can regulate the flow of information. A typical LSTM cell has three gates-forget, input, and output. In Fig 4, these are depicted as the three sigmoid layers.

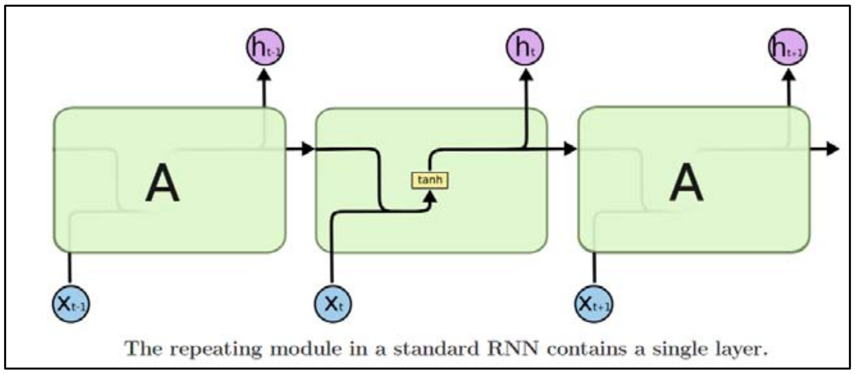
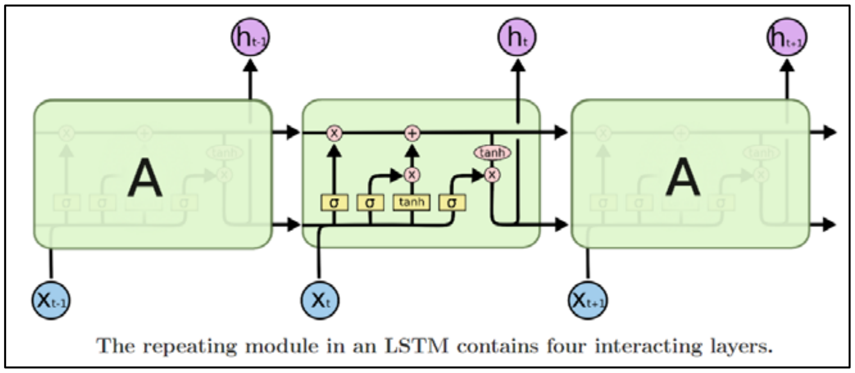

h_{t-1}
h_t
h_{t+1}
A
tanh
A
X_{t-1}
X_t
X_{t+1}
The repeating module in a standard RNN contains a single layer.

Fig 3: RNN Cell [12]



h_{t-1}
h_t
h_{t+1}
A
σ
σ
tanh
σ
tanh
A
X_{t-1}
X_t
X_{t+1}
The repeating module in an LSTM contains four interacting layers.

Fig 4: LSTM Cell [12]

GRUs, like LSTMs, also mitigate the vanishing gradient issue while using fewer parameters when compared to LSTMs. Considered as a variation of LSTMs, GRUs are preferred in certain scenarios.

The number of gates present within the cell of a GRU is reduced to just two (sigmoid layers: r_t and z_t in Fig 5) as opposed to three gates in an LSTM. A typical GRU cell is shown in Fig 5 and details about the two GRU gates is mentioned is section 3.1.2.

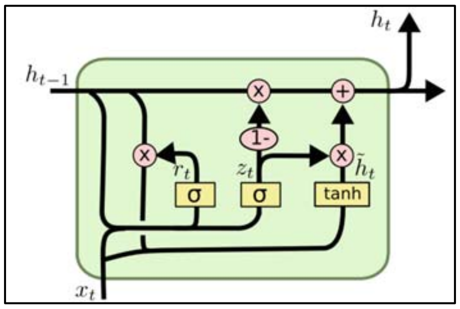


Fig 5: GRU cell [12]

3.1.1 Long Short-Term Memory

This section explains in detail the working of an LSTM cell. As mentioned in section 3.1, an LSTM has 3 gates: Forget gate, Input gate and, Output gate. The input to each of these gates is the same i.e., the input of current timestep (x_t) and the output from the previous timestep (h_{t-1}).

Along with the above inputs, the cell state (C_t) runs through the LSTM cell carrying the memory of the cell. This is the cell state (memory) variable (C_t) that acts like a conveyor belt. It runs straight down the entire chain of LSTM cells, with only some minor linear interactions with the gates. It is crucial that this variable is not confused with the LSTM cell output.

In Fig 6-9[12], the description of the notations are as follows:

- **W_f, W_i, W_C, W_o:** Weight matrices w.r.t gates and cell state
- **b_f, b_i, b_C, b_o:** Biases w.r.t gates and cell state
- **σ:** Sigmoid Activation function outputs a value between 0 and 1 for any given input.
- **tanh:** Tanh Activation function outputs a value between -1 and 1 for any given input and has a steeper gradient as compared to sigmoid.
- ***:** Hadamard product, an operation used to multiple two matrices of the same dimensions. For example, matrix A with dimension [2 X 5] can be multiplied elementwise with matrix B of dimension [2 X 5]. As opposed to matrix multiplication that requires the number of columns in matrix A to be equal to the number of rows in matrix B.

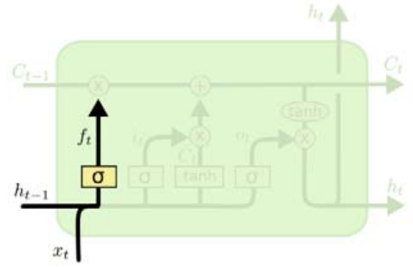
The 3 gates determine the following:

- Information to get rid of at each timestep
- Information to carry to the next timestep
- Output at each timestep

Breaking down the LSTM cell in Fig 4, we can understand the how the above determinations are made.

Information to get rid of at each timestep

Forget Gate (f_t) is responsible in deciding what part of the cell state from the previous timestep (C_{t-1}) must be forgotten. The sigmoid activation, also called as recurrent activation, is used to output values between 0 and 1, where 1 represents "completely keep this" while 0 represents "completely get rid of this." Fig 6 shows the working of the forget gate along with its mathematical equation.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

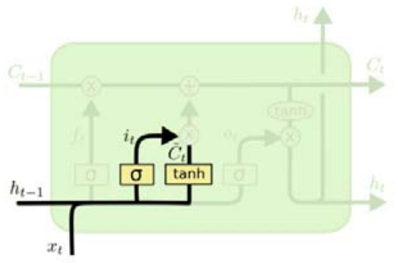
Fig 6: Forget Gate [12]

Information to carry to the next timestep

Input gate is responsible in determining if information should be saved to the cell state or should be left behind.

Now that the data to be eliminated has been taken care of by the forget gate, we need to evaluate what data must be carried to the next timestep. This is done in two parts. The first part involves the input gate (i_t), which is also a sigmoid layer (Fig 7). It determines what data carried by the cell state must be updated and carried forward to the next timestep.

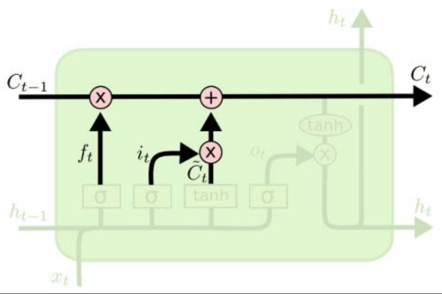
The second part is a tanh layer that creates a vector of new values ($\tilde{C}_t$), that can be added to the current cell state. Tanh activation pushes the values between -1 and 1 and inhibits the data that we do not wish to add to the cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Fig 7. Input Gate and new values added to cell state [12]

Finally, as seen in Fig 8, the information to carry to the next timestep is decided from the outputs of the input gate, new values added to the cell state, forget gate and, the cell state from the previous timestep.



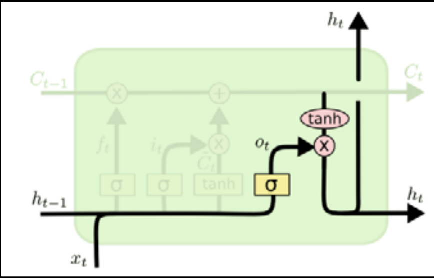
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Fig 8: Cell state computation [12]

<u>Output at each timestep</u>

The Output Gate (o_t) and the Cell state decide the output at each timestep.

The output gate, which is also a sigmoid layer, decides which parts of the cell state we wish to output. Finally, we put the cell state through tanh and multiply it by the output of the sigmoid gate, to determine the hidden state for the next LSTM cell (h_t) [12].



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Fig 9: Output computation [12]

3.1.2 Gated Recurrent Unit

This section explains in detail the working of a GRU cell.

As mentioned in section 3.1, a GRU has 2 gates called the Reset gate (r_t) and the Update gate (z_t). The input to each of these gates is the same: the input of the current timestep (x_t) and the output from the previous timestep (h_{t-1}).

A GRU can be considered as an optimized version of an LSTM where the two GRU gates are used to perform the same function as the three gates of an LSTM.

Fig 10[12] shows the computations involved within a GRU cell.

The 2 gates determine the following:

- Information to get rid of at each timestep
- Information to carry to the next timestep
- Output at each timestep

<u>Information to get rid of at each timestep</u>

To correlate GRU w.r.t LSTM, in Fig 10, h_{t-1} acts as the conveyor belt. This is the output from the previous GRU cell and also holds the memory and carries it through the GRU layer. The reset gate (r_t) decides how much of the memory must be forgotten before going on to the next GRU cell.

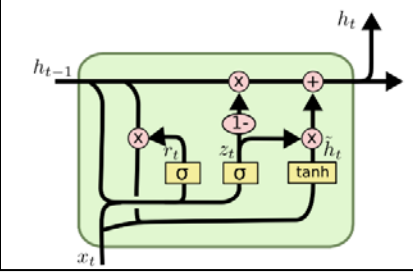
Information to carry to the next timestep

The update Gate (z_t) combines the function of the forget gate and input gate of an LSTM into one and decides how much of the past information must be passed along to the next GRU cell.

Output at each timestep

Finally, using the reset and update gate outputs, the vector h_t is calculated which is the output at the current timestep and also the memory that is carried to the next timestep.

Sigmoid and Tanh activations play a similar role in GRU as in LSTM. In Fig 10, Wz, Wr and W are weight matrices w.r.t the gates and cell state.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Fig 10: GRU cell [12]

Now that we understand how an LSTM processes information and learns the relationship between inputs and outputs, we now must look at which hardware platform would be suitable to carry this network and perform relevant tasks. In the next section we look at how and why FPGAs are well suited for this part.

4. Field Programmable Gate Arrays and Neural networks

Field Programming Gate Arrays are one of the most obvious solutions that come to mind when we consider the implementation of neural networks. Their highly functional and adaptable architecture helps them handle different algorithms in computing, while accommodating memory resources in the same device. FPGAs are programmable devices that offer indisputable advantages in terms of flexibility. In addition, they have low power consumption and high speed making them well-suited to ML applications. With neural networks transforming in many ways and reaching out to more industries, it is useful to have the adaptability FPGAs offer.

In the field of machine learning, FPGAs are preferred for the task of inference as opposed to training as the two differ in their requirements. Training is the process by which a neural network determines a set of weights and biases to best relate a set of inputs to their outputs, and inference is the process of using these weights and biases to predict an output for a certain input. Training is primarily a throughput bound workload and insensitive to latency. On the other hand, inference can be much more latency sensitive [8]. The fact that FPGAs are extremely effective in parallelizing mathematical operations required for an inference comes into play here.

With all the benefits of implementing a machine learning algorithm or neural network on a FPGA stated above, the question now remains, what is the best approach of doing it? Hardware description languages like Verilog and VHDL are at a very low level of abstraction and hence coding for highly complicated neural networks is an extremely tedious job. Thankfully, we do have ways to simplify this using High-Level Synthesis.

4.1 High-Level Synthesis for Machine Learning

High-Level Synthesis (HLS) works at a higher level of abstraction as compared to hardware description languages (HDL), and has the ability to go from complex algorithms written in a high-level language such as SystemC or C/C++ to register transfer level (RTL) HDL such as Verilog or VHDL This enables accurate implementation of the algorithm in RTL without having to write it manually. Additionally, detailed profiles for power and performance are also made available at the end of synthesis.

There are numerous tools that can be utilized to perform the aforementioned HLS process in an accurate manner. With the help of these tools, designers can use a more intuitive and algorithmic programming language and avoid writing HDL from scratch. They handle the micro-architecture and transform HLS code into fully timed RTL implementations, automatically creating cycle-by-cycle detail for hardware implementation. The (RTL) implementations are then used directly in a conventional logic synthesis flow to create a gate-level implementation.

4.2 Why is HLS4ML needed?

HLS4ML is a user-friendly software that aims towards providing an efficient and fast translation of machine learning models to HLS. The software uses the ML models/networks that have been trained using ML packages, such as Tensorflow, PyTorch, Keras etc., as inputs. This input is then translated to provide a hardware description language RTL implementation as output.

HLS4ML also provides the user with flexibility to define parameters within the framework to best suit their needs. Parameters such as precision of the calculations in the model, and parallel/serial implementation of the algorithm with varying levels of pipelining, can be controlled by the user. Fig 11 shows the workflow of the HLS4ML framework, where the blocks in blue are the steps carried out within the software.

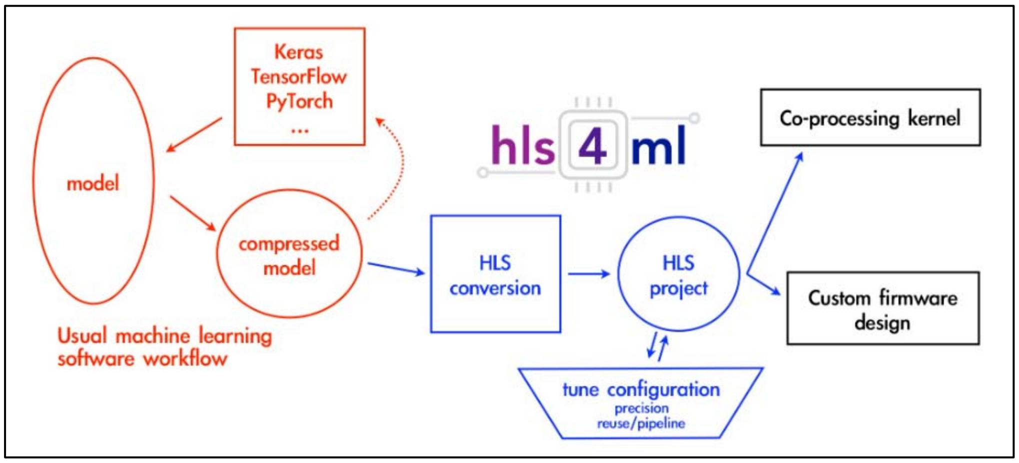


Fig 11. HLS4ML workflow [7]

Utilizing this framework allows the user to greatly reduce the time to obtain results by enabling fast prototyping of the model implementation on an FPGA. As an added benefit, this option of controlling parameters allows the user to adjust the balance between performance, resource utilization and latency requirements.

4.3 HLS4ML framework

Fig 11 shows a typical workflow to translate a model into a FPGA implementation using HLS4ML. This package is used for the automatic translation of a trained neural network that is specified by the model's architecture, weights, and biases, into HLS code [7].

The modules that work together within HLS4ML are described below:

Converter: This module defines each layer of the neural network and creates python objects for it. Layers such as "Conv1D", "Dense", "LSTM" etc. must be defined within this converter to be interpreted by the framework. HLS4ML currently supports four ML packages (Tensorflow, PyTorch, Keras, Onnx); thus, for each of these packages, a separate converter is scripted.

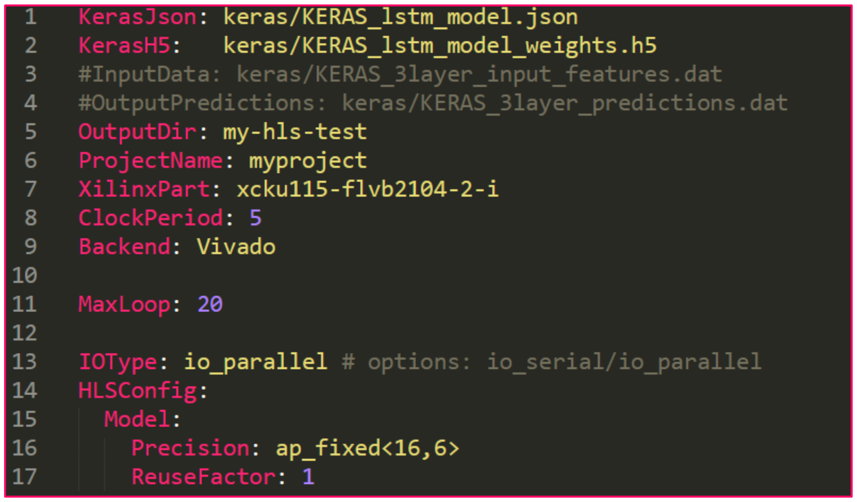
HLS Model: In this module of HLS4ML, each layer is described as a class, and within each class the parameters passed on to the final HLS code are defined. Examples of some parameters are weights, biases, activation etc. This class also defines the output shape of the specified layer based on attributes from the converter.

Templates: This is a module which defines the layer configuration and function templates. Based on the layer name from the HLS Model, the right template is found for each of the layers and pasted into the parameters header file for the final HLS code. This module also contains an include list definition, where a set of header files are associated with each layer type. This enables the right header files to be included in the final code based on the layers that are picked up.

Algorithm: The algorithms (the mathematical blocks of the neural network) are defined as header files, which are mentioned in the include list of their respective layers. The exact calculations that take place between the inputs and weights, based on their sizes, are defined within these header files. There is a separate algorithm defined for each layer type.

Vivado Writer: The Vivado writer consolidates all the above modules together and provides a holistic structure. This module consists of functions that define the contents of the final HLS code. It specifies where the data gets picked up from and added to the final code. For example, there are functions for writing the contents of the HLS testbench, HLS main, header files and also the weights and biases per layer. Additionally, it also writes the build script that builds the Vivado project from C simulation to C/RTL CoSimulation.

To understand the working of HLS4ML framework end-to-end, it is important to know the inputs given to the framework, parameters that can be modified by the user before passing the inputs and the contents of the output folder. As outlined in Fig 12, the input is fed into the framework using a YAML file with the described format. Here, we can see that the trained model is given as a JSON file and its weights as an H5 file. There is the added flexibility of defining parameters like precision and reuse factors. Pertinent details such as clock period and the target FPGA are also provided in this file.



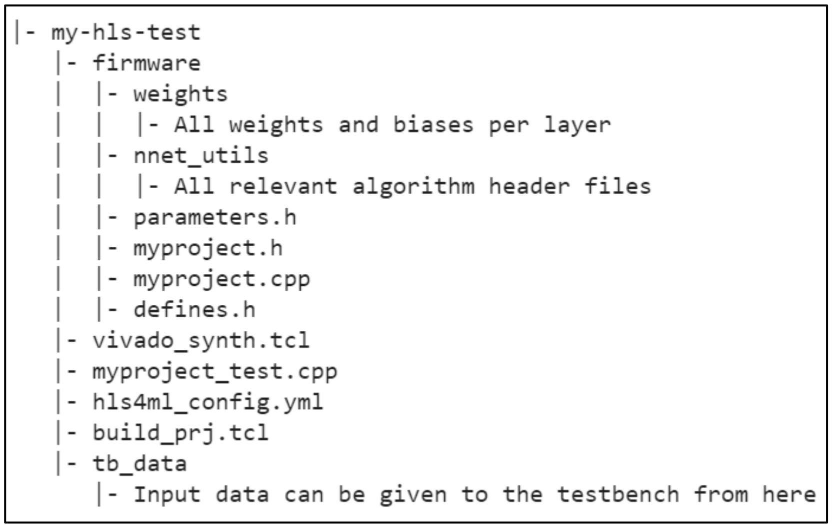
```yaml
KerasJson: keras/KERAS_lstm_model.json
KerasH5:   keras/KERAS_lstm_model_weights.h5
#InputData: keras/KERAS_3layer_input_features.dat
#OutputPredictions: keras/KERAS_3layer_predictions.dat
OutputDir: my-hls-test
ProjectName: myproject
XilinxPart: xcku115-flvb2104-2-i
ClockPeriod: 5
Backend: Vivado

MaxLoop: 20

IOType: io_parallel # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>
    ReuseFactor: 1
```

Fig 12: Sample YAML file

The above mentioned YAML file also names the output directory and project as shown. The contents of the output directory follow the hierarchy as shown in Fig 13.



```
|- my-hls-test
   |- firmware
   |  |- weights
   |  |  |- All weights and biases per layer
   |  |- nnet_utils
   |  |  |- All relevant algorithm header files
   |  |- parameters.h
   |  |- myproject.h
   |  |- myproject.cpp
   |  |- defines.h
   |- vivado_synth.tcl
   |- myproject_test.cpp
   |- hls4ml_config.yml
   |- build_prj.tcl
   |- tb_data
      |- Input data can be given to the testbench from here
```

Fig 13: Output directory hierarchy

4.4 Neural Network models and machine learning packages in HLS4ML

Machine learning packages currently supported by HLS4ML are:

- Keras
- Tensorflow
- PyTorch
- Onnx

The following four types of neural network architectures are either fully supported or currently under test in HLS4ML:

- Fully Connected NNs (multi-layer perceptron)
- Boosted Decision Trees
- Convolutional NNs (1D/2D)
- Recurrent NNs – LSTMs (as provided by my work within this thesis)

The above models are trained on different types of datasets based on the type of neural network. For example, convolutional neural networks work best with an image as input; hence, they are provided with jet images as inputs. On the other hand, recurrent neural networks work best with a sequence of input data, and therefore are provided with lists of particle properties belonging to a single jet.

The latencies of inference of these models are intended to be at most 1μs.

5. Application in Physics

Up until now, we have established how powerful machine learning algorithms can be and what role HLS4ML plays in implementing them on FPGAs. In this section we discuss how ML plays an important role in the field of physics.

The Large Hadron Collider [13], which is the world's largest and most powerful particle accelerator generates particle physics data used to address fundamental open questions in physics. The experiments that take place in the LHC result in a dataset too large for manual interpretation, and machine learning has taken a pivotal role in this field [15]. The utilization of machine learning in physics has broadened the horizon while pushing boundaries on the size of data that can be mined and assessed. One of the applications of machine learning in physics is the task of Top Tagging, which is the discrimination of jets originating from hadronic decays of a top quark from light-flavor and gluon originated jets [4].

Top Tagging:

Within the LHC, when two proton beams collide, they result in the production of jets (Fig 14). A jet is a collection of particles resulting from that collision. The particles associated with a single jet can be studied or evaluated using the features the jet possesses. Based on this evaluation, we can classify the jets into various types. Of these types, five of them are mentioned below:

- quark (q)
- gluon (g)
- W boson (W)
- Z boson (Z)
- top (t)

In the above classification, by "quark" we actually mean light quarks such as up, down, or strange and by "top" we mean the top quark. W, boson, Z boson and gluon are types of forces that exist between quarks.

Moreover, the features associated with each particle are:

- pT: Transverse Momentum
- eta: Pseudo Rapidity
- phi: Azimuthal Angle
- E: Energy

- deltaR: Relative angular distance w.r.t jet axis
- pdgID: Particle identification information

Based on the type of neural network model used to evaluate the jet substructure, some or all of the above features are used to characterize each particle.

Fig 14 shows a cartoon of the proton – proton collision that result in a particle jet. When these jets propagate through a detector, they deposit energy in the hadronic calorimeters. This deposited energy or signals can be reconstructed to form a jet using algorithms and thus be used for research purposes.

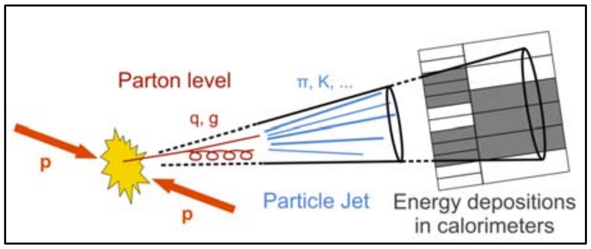


Fig 14. Proton-proton collision and resulting jet [3]

Why top tagging? The top quark is the heaviest known elementary particle [11] and has a correspondingly short lifetime and thus behaves differently as compared to the other quarks (up, down, strange, charm, and bottom). This also means that it decays much before it can combine to form other particles and passes its spin information to its decay products which is why the only way to study the properties of top quarks is through its proxy – top jet. Due to its large mass, it stands out from rest of the quarks and therefore is of keen interest to researchers in the field [11].

5.1 Deep Neural Networks for top tagging

In 2018, a paper was submitted by the HLS4ML group describing the use of the HLS4ML framework for fast inference of deep neural networks (DNN) [1]. This section provides details about the type of inputs given, model architecture, performance of the model and resource estimates.

Input type: The input provided to the network are collimated showers of particles that result from the decay and hadronization of quarks q and gluons g, termed "Jets". These jets can be represented as grayscale images, RGB images, sequences of particles, or a set of physics-inspired high-level features [1]. In the case of DNNs, the input is a set of physics-inspired high-level features.

DNN architecture: Two DNN architectures built on the Keras ML package were used in this paper for different purposes. The first is a fully-connected neural network with three hidden layers used to categorize the five classes of jets (q, g, W, Z, t), while the second is a one-hidden layer model used for top tagging.

Model performance: Receiver Operating Characteristics (ROC) curve and Area under curve (AUC) are the performance metrics used to quantify the model performance (Refer section 6.1.2). Optimal performance with no loss of classification power corresponds to AUC = 1. This metrics can be used to guide tuning, such as deciding the number of integer bits and number of fractional bits (precision parameter). For the DNN model used, the precision was set to <16,6>, where 16 is the bit width and 6 is the integer bits. This implies the fractional bits equate to 10 (Refer Fig 15).

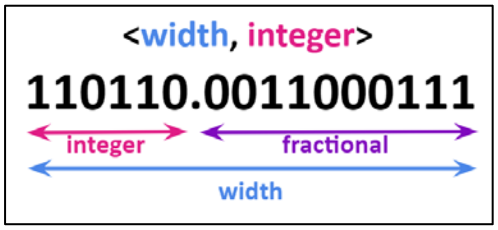


Fig 15: Precision

Resource estimates: Table 1[1] summarizes the resource usage for the uncompressed DNN model.

Table 1: DNN resource estimates

Parameters	**DNN**
FPGA Targeted	xcku115-flvb2104-2-i
Precision	<16,6>
Reuse Factor	1
Latency (ns)	75
Utilization % - DSP [14]	60
Utilization % - Logic (LUT + FF) [14]	13

After the successful implementation of DNNs for the application of top tagging, the HLS4ML team aims towards implementing different types of neural networks for the same application while leveraging the uniqueness of respective neural networks.

The next section discusses my effort towards adding LSTMs into the HLS4ML framework.

6. Long Short-Term Memory RNNs in HLS4ML

To incorporate the LSTM layer into HLS4ML framework, the steps mentioned in section 4.3 must be followed. Details about the model configuration, input data, model performance and resource utilization are mentioned in the subsections below.

6.1 KERAS LSTM model configuration

The KERAS LSTM model that is provided as input to the HLS4ML framework is given in the form of:

- Model: Stored in the form of a JSON file that describes the model layers and activations
- Weights: Consists of the trained weights and biases. Stored in the form of an H5 file.

MODEL:

Fig 16 represents a block diagram of the model with details pertaining to each layer.

As seen, the LSTM model used is basic, with just one LSTM layer. Additionally, the shape of the input is [20 X 6], where 20 is the number of particles provided for each inference and 6 are the features per particle.

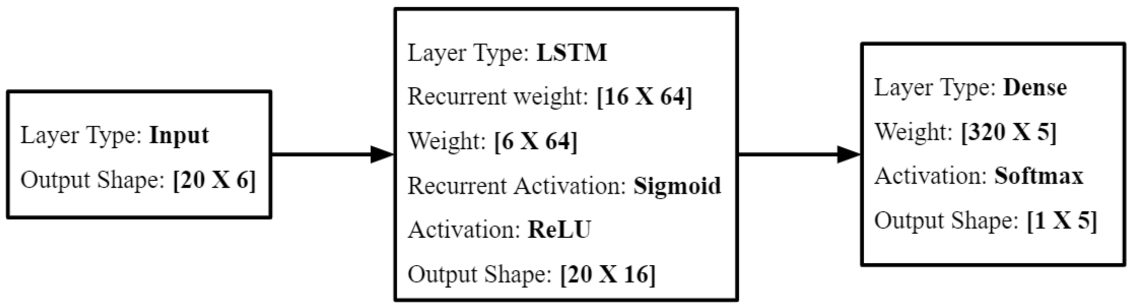


Fig 16: LSTM Model Configuration

WEIGHTS:

Weights from the LSTM layer and Dense layer are saved in the KERAS weights file.

LSTM weights: This layer has two types of trained weights that are stored in KERAS in the form of two big matrices. The first matrix stores the kernel weights and the second matrix stores the recurrent kernel weights. Kernel weights are the weights stored per layer in a neural network and applied to the input at the current timestep; whereas, recurrent kernel weights are special to RNNs and are applied to the output of the previous timestep. The big matrices are divided into 4 equal parts that correspond to the 3 LSTM gates and one cell state for inference computation. Fig 17 explains the order in which these weights are stored along with their shapes.

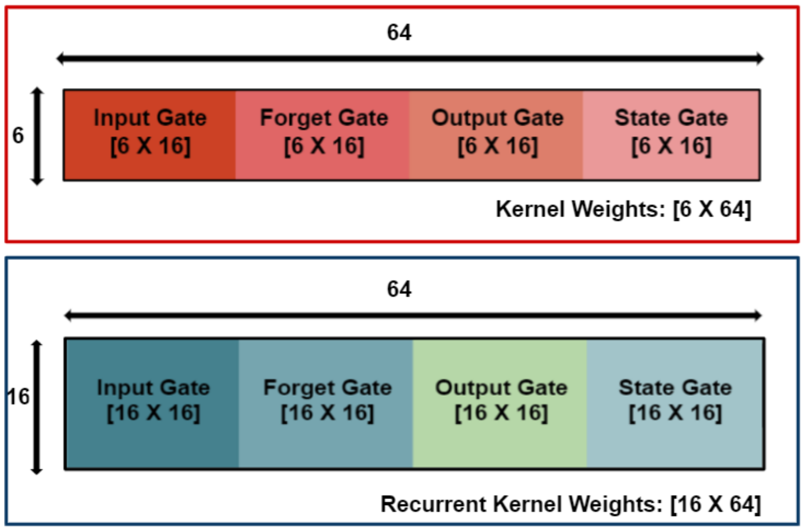


Fig 17: KERAS weight storage

Dense weights: This layer has a single weights matrix with the shape [320 X 5]. Details about how these weights are used in inference are provided in section 6.2.

6.1.1 Dataset used

The dataset used to train the KERAS LSTM model is located at [9]. This consists of particles that are characterized by 6 features. These features are labeled as:

- j1_ptrel
- j1_etarot
- j1_phirot
- j1_erel
- j1_deltaR
- j1_pdgid

These features (also mentioned in section 5) are used to characterize a particle. However, these features are normalized at the data processing stage of training and the relative feature values a.k.a normalized feature values are sent as input to the neural network.

In order to identify an input as a top jet, each of these features have a range of values they fall within. Of the 6 features mentioned above, based on the type of neural network we can use all or some of the features. For example:

- CNN2D: A 2-dimensional convolutional neural network takes only features j1_ptrel, j1_etarot and j1_phirot in the form of a 2-dimensional image as input.
- LSTM: A Long Short-Term Memory network takes all 6 features of all the particles as a sequence of inputs.

The output of the LSTM network has a shape of [1 X 5] as described in section 6.1, where the 5 probabilities correspond to q, g, W, Z and t, which stand for the different types of jets as mentioned in section 5.

A high probability of "t" in the output indicates that the particles given as input had features in the desired range of a top jet.

6.1.2 Model Performance

There are two performance parameters used to measure the performance of a neural network model:

1. Receiver operating characteristics curve: Also known as ROC curve, is often used as a performance metric for classification models in machine learning. This curve plots two parameters (False Positive Rate (FPR) vs True Positive Rate (TPR)) at thresholds ranging from 0.0 to 1.0 (as shown in the x-axis of Fig 18). Fig 18 shows the ROC curve of the KERAS LSTM model, where the TPR is the signal efficiency and FPR is the background efficiency.

    - Signal Efficiency (TPR): Defined as the ratio of the number of top jets identified as top to the total number of top jets.
    - Background Efficiency (FPR): Defined as the ratio of the number of non-top jets identified as top to the total number of non-top jets.

    To understand what a threshold value is, consider an example: a threshold value of 0.7 means that a probability in the range [0.0 – 0.69] is a negative outcome (0) and a probability in [0.7 – 1.0] is a positive outcome (1). These threshold values are w.r.t the probability outcomes of the neural network model under consideration. The curves in Fig 18 are FPR vs TPR at different threshold values. The higher the signal efficiency and the lower the background efficiency means the better the performance of the classification model. For example, in Fig 18, t tagger has a high signal efficiency corresponding to respectively lower background efficiency as compared to other taggers and thus has maximum accuracy of 92.9%.

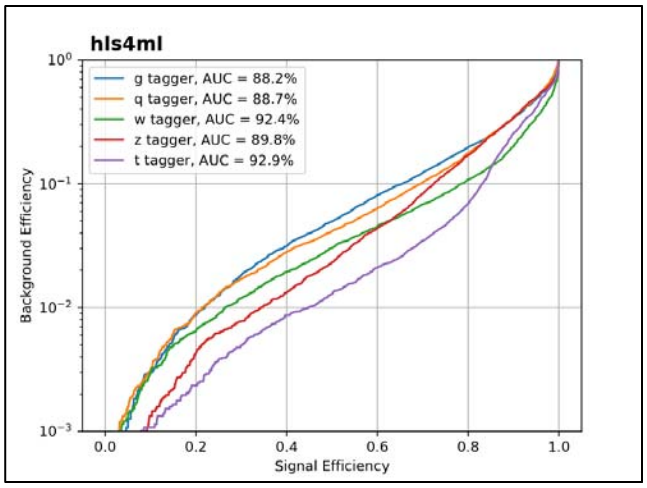


Fig 18: ROC curve

2. Area Under ROC Curve: Also known as AUC, this performance parameter measures the area under the ROC curve. In simple terms, higher the AUC, better the model at predicting 0s as 0s and 1s as 1s.

6.2 LSTM inference process

This section explains in detail the LSTM inference process mathematically with respect to the KERAS LSTM model.

Input layer: For each inference, we provide the data of 20 particles. Each particle data is of shape [1 X 6]. Thus for 20 particles, the shape is 20 [1 X 6] = [20 X 6]. This data is sent to the next layer i.e., LSTM layer.

LSTM layer: As this layer receives 20 particles, we require 20 LSTM cells to perform computation on each of these particles. In short, one LSTM layer uses the same LSTM cell 20 times. The computations in each cell is shown in Fig 19. As seen in this figure, each LSTM cell produces an output (h_t) of shape [1 X 16]. As we have 20 LSTM cells and each cell produces an output of shape [1 X 16], the shape of the output from the LSTM layer will be 20 [1 X 16] = [20 X 16]. This becomes the input to the next layer.

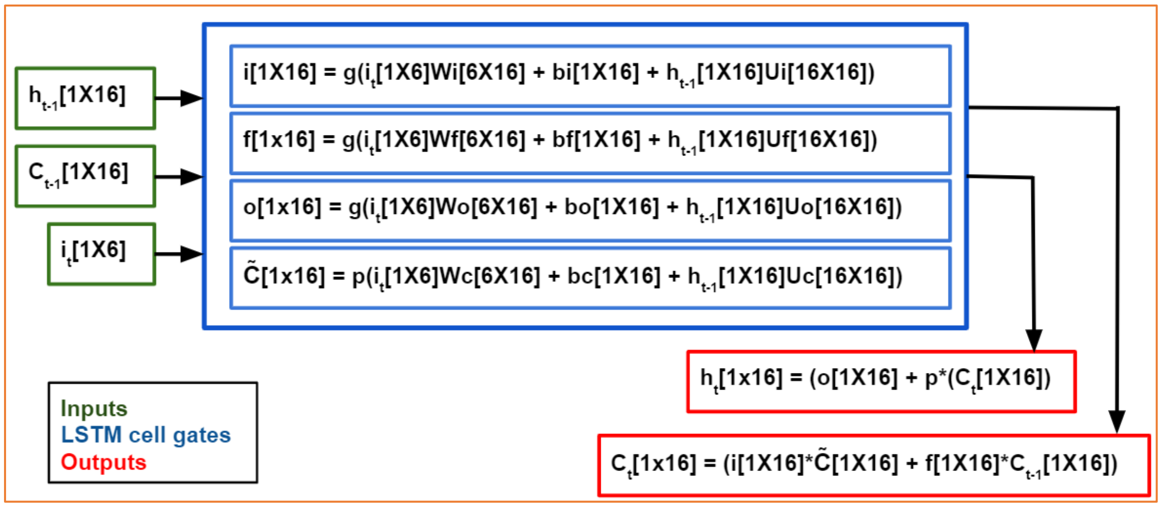

h_{t-1}[1X16]
C_{t-1}[1X16]
i_t[1X6]
i[1X16] = g(i_t[1X6]Wi[6X16] + bi[1X16] + h_{t-1}[1X16]Ui[16X16])
f[1x16] = g(i_t[1X6]Wf[6X16] + bf[1X16] + h_{t-1}[1X16]Uf[16X16])
o[1x16] = g(i_t[1X6]Wo[6X16] + bo[1X16] + h_{t-1}[1X16]Uo[16X16])
C̃[1x16] = p(i_t[1X6]Wc[6X16] + bc[1X16] + h_{t-1}[1X16]Uc[16X16])
h_t[1x16] = (o[1X16] + p*(C_t[1X16])
C_t[1x16] = (i[1X16]*C̃[1X16] + f[1X16]*C_{t-1}[1X16])
Inputs
LSTM cell gates
Outputs

Fig 19: LSTM cell computation

The above figure shows the inputs, outputs, and computation in one LSTM cell along with the shape of each parameter. Table 2 provides the label description for Fig 19.

Table 2: Label description for Fig 19

Label	**Description**
h_{t-1}	Output at time t-1
C_{t-1}	Cell state at time t-1
i_t	Input at time t
i	Input Gate
f	Forget Gate
o	Output Gate
$\tilde{C}_t$	New values added to Cell state
Wi, Wf, Wo, Wc	Kernel weights for each gate and cell state
bi, bf, bo. bc	Kernel biases for each gate and cell state
Ui, Uf, Uo, Uc	Recurrent kernel weights for each gate and cell state
g	Recurrent Activation
p	Kernel Activation
h_t	Output at time t
C_t	Cell state at time t
*	Hadamard product

Dense layer: The LSTM output data of shape [20 X 16] is then flattened to [1 X 320] and sent as input to the Dense layer. This input [1 X 320] is multiplied with the dense weights [320 X 5] to produce an output of shape [1 X 5]. It is then passed through the SoftMax activation that results in the final 5 probabilities.

6.3 Adding recurrent layers into the framework

To add the recurrent layers such as LSTM and GRU into the framework, the modules mentioned in section 4.3 must be modified to include these layers.

Fig 20 shows a block diagram that explains the 4 main modules in the framework that need modifications in order to add a neural network layer.

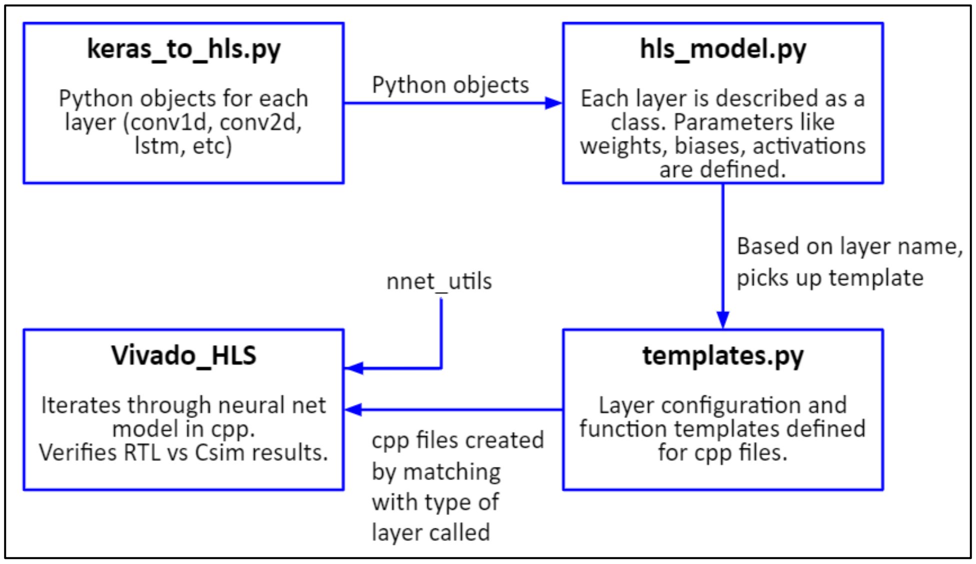

keras_to_hls.py
Python objects for each layer (conv1d, conv2d, lstm, etc)
Python objects
hls_model.py
Each layer is described as a class. Parameters like weights, biases, activations are defined.
Based on layer name, picks up template
nnet_utils
Vivado_HLS
Iterates through neural net model in cpp. Verifies RTL vs Csim results.
templates.py
Layer configuration and function templates defined for cpp files.
cpp files created by matching with type of layer called

Fig 20: HLS4ML setup

keras_to_hls.py: As we are working with a KERAS LSTM model, the KERAS converter needs modification. Here, the LSTM layer is defined based on the layer definition in the Keras documentation [2]. In addition to this, the "recurrent activation", which is specific to RNNs, is also defined as a Python object in this converter and passed on to the HLS model.

hls_model.py: In this script, LSTM is defined as a class. Within this class, the output shape of the LSTM is defined as the product of the number of particles and the state parameters. We also define a recurrent bias array populated with all zeros in order to utilize the dense multiplication algorithm for the LSTM layer. Apart from this, the following parameters are defined: weights, biases, recurrent weights, activation, and recurrent activation. These parameters are made available in the final HLS code.

templates.py: As seen in section 6.1, each LSTM cell performs activation as well as multiplications with weights and adding of biases within it. To incorporate this feature, the LSTM template is defined accordingly. The activation and multiplication templates are called within the LSTM template. This is unique to RNNs as their recurrent behavior require calculations within the cell, as opposed to just from previous layers.

nnet_recurrent.h: This is the header file which describes the calculations within each LSTM cell. Here two LSTM algorithms have been formulated. The first one is an algorithm based on section 6.2 and two new parameters, called the cell output and cell state, are defined. The second algorithm is a modified version of the first, the only variance being that the two new parameters are defined as static arrays. This is done in order to reuse the same matrix multiplication logic again, as opposed to creating a new one for every iteration. This reduces resource utilization and is thus preferred. However, both of the algorithms are defined in terms of a single particle, and thus we require an additional logic that loops over all particles. This looping function is called from within the final HLS code and the subsequent LSTM algorithm is used for computation.

6.4 Verification of KERAS model vs C simulation

The HLS model of a neural network that is generated when a trained model is passed through the HLS4ML framework must be verified against its KERAS model to ensure correct and efficient conversion by the framework. This is done by providing the same input values to both the HLS and KERAS models and checking whether the outputs from both of these models match.

A Python code was written that loads the KERAS LSTM model configuration (JSON) and weights (H5) and provides the prediction output for any dummy input given. To obtain the output prediction from the HLS model, the same dummy input was hardcoded into the HLS testbench and a C simulation was run using Vivado HLS. As part of the verification, multiple dummy inputs were given and the predictions from both the models were compared.

6.5 LSTM model on FPGA – Resource Utilization

Vivado HLS generates a synthesis report that provides details on performance as well as area of the RTL design. In this section, we analyze the resources utilized on the targeted FPGA and understand the factors that affect it. The FPGA resources required by the model depend on multiple aspects. Some of them are:

- Quantization: This is the precision chosen for the inputs, weights, and biases. The quantization applied to this model was a fixed-point precision <16,6>. This value is chosen keeping in mind the trade-off between model performance and resource utilization. A higher precision can enhance model performance; however, it can also considerably increase the amount of FPGA resources.
- Parallelization: This parameter decides the number of times a given multiplier is reused within a layer computation. If the "Reuse factor" equals 1, it means that the system is completely parallel.

Based on the above dependencies, the following firmware implementation metrics are monitored for the LSTM network implementation:

- Resources: This includes DSPs, FFs and LUTs
- Latency: Time it takes to compute the entire network
- Initiation interval: Time before a new set of inputs can be accepted.

As mentioned in section 6.3, two different algorithms were tested for the LSTM implementation. The results for both these algorithms are given in Table 3.

Table 3: Resource Utilization

Parameters	**LSTM Algorithm**	**LSTM Static Algorithm**
FPGA Targeted	xcku115-flvb2104-2-i	xcku115-flvb2104-2-i
Precision	<16,6>	<16,6>
Reuse Factor	1	1
Latency (μs)	1.35	1.35
Initiation interval (ns)	5	1350
Utilization (%) – DSPs	326	43
Utilization (%) – FFs	48	6
Utilization (%) – LUTs	104	10
Utilization (%) – BRAM	11	~0

As seen in Table 3, the resource utilization in the LSTM Algorithm implementation is fairly large. One way to reduce the resources being used is to stop the pipeline and use static arrays. The LSTM static algorithm does exactly this. The static array reuses the same matrix multiplication algorithm repeatedly.

The DSP utilization is reduced by ~86%, FF utilization is reduced by ~87.5%, LUT utilization is reduced by ~90% and BRAM utilization is reduced by ~100%. However, as there is no pipeline, the initiation interval has increased from 5ns to 1.35μs.

Another point to note is that, ideally, the HLS4ML framework works towards generating networks that operate within a 1μs latency. As the KERAS LSTM model used has an input sequence of 20 particles, a latency higher than 1μs is expected due to limitations in the hls4ml framework and would require serious optimizations to ensure a smaller latency. However, reducing the input sequence to 10 particles would solve this issue.

6.6 KERAS DNN Top Tagging model vs KERAS LSTM Top Tagging model

In this section, we will compare the Keras DNN Top Tagging model discussed in section 5.1 with the Keras LSTM static Top Tagging model discussed in section 6. The comparison is based on the model performance (using ROC curves) and overall resource utilization.

DNN vs LSTM Model Performance

Fig 21 shows the ROC curves of both the top tagging models. Looking at the accuracy numbers, we can see that the performance of the DNN model is slightly better than the LSTM model.

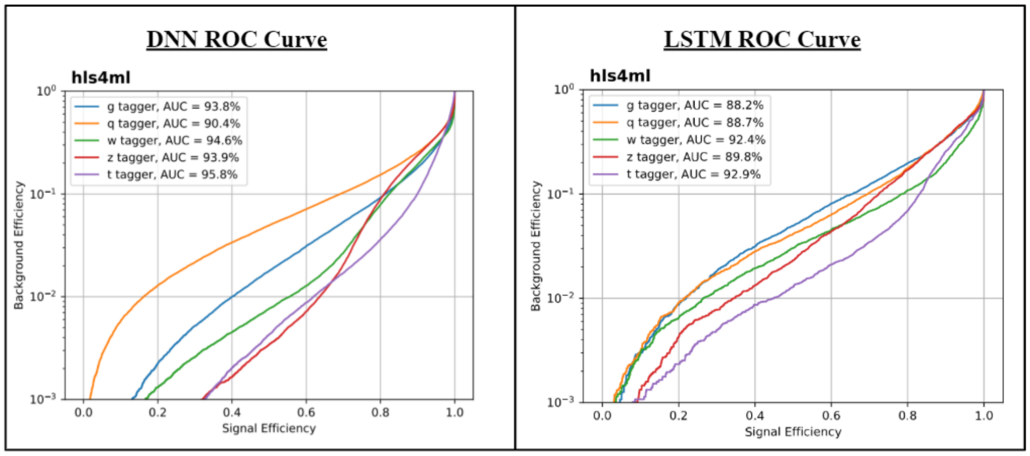


Fig 21: DNN vs LSTM ROC Curves

DNN vs LSTM Resource Utilization

Table 4 shows the resource utilization numbers for DNN vs LSTM model targeted on a specific FPGA.

Table 4: DNN vs LSTM Resource Utilization

Parameters	**DNN**	**LSTM Static**
Input particles	16	20
FPGA Targeted	xcku115-flvb2104-2-i	xcku115-flvb2104-2-i
Precision	<16,6>	<16,6>
Reuse Factor	1	1
Latency (ns)	75	1350
Utilization (%) – DSPs	60	43
Utilization (%) – FFs + LUTs	13	8

7. Conclusion

In this thesis, we described the steps taken to implement a Long Short-Term Memory neural network into the HLS4ML framework. The modifications made to the framework are for a specific ML package called Keras. To start with, we considered a Top tagging trained LSTM model that takes a sequence of 20 particles as input and performs inference. We studied the framework in detail such that the information that can be used to add any new layer into HLS4ML. Further, we understood the basics of recurrent neural networks and two of its modified versions – LSTM and GRU.

Specifics about the LSTM model used, and mathematical calculations involved in each inference with respect to the data were studied. Once the LSTM was successfully implemented, we went on with a two-step verification process. The first one involved the verification between the KERAS LSTM model versus the HLS model generated by the framework. The second verification was performed by the HLS tool – Vivado HLS, that confirmed the model functionality between HLS and RTL. Finally, we analyzed the reports generated to understand the overall resource utilization and latency per inference and discussed the possible causes of the obtained results.

The code for the implementation is located on GitHub and further details are mentioned in Appendix A.

8. Next Steps

While the implementation of the LSTMs into the framework is functional and has been extensively tested on the KERAS LSTM Top tagging model to receive the latency and resource utilization of the targeted FPGA, this is just the beginning in terms of exploring the future of LSTM models in the field of particle physics. As mentioned in section 6.5, that a model with 20 particles is bound to yield a latency greater than 1μs owing to the limitations of the framework, the immediate next step would be to test the latency and mapping out the resource utilization for a model that takes in 10 particles as input.

Further, training a new LSTM model with a deeper architecture and performing inference using the framework will help identify not just if the prediction accuracy is considerably better, but also analyze the FPGA resources required by the model.

Finally, another class of RNNs, called the GRU (refer section 3.1.2), can be added to the framework following an implementation similar to the LSTM.

9. References

[1] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N. and Wu, Z., 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07), pp.P07027-P07027.

[2] Team, K., 2020. *Keras Documentation: Keras API Reference*. [online] Keras.io. Available at: <https://keras.io/api/> [Accessed 16 May 2020].

[3] Cms.cern. 2020. *Jets At CMS And The Determination Of Their Energy Scale | CMS Experiment*. [online] Available at: <https://cms.cern/news/jets-cms-and-determination-their-energy-scale> [Accessed 16 May 2020].

[4] Egan, S., Fedorko, W., Lister, A., Pearkes, J. and Gay, C., 2017. [online] Available at: <https://arxiv.org/pdf/1711.09059.pdf> [Accessed 16 May 2020].

[5] Medium. 2020. *Understanding RNN And LSTM*. [online] Available at: <https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e> [Accessed 16 May 2020].

[6] En.wikipedia.org. 2020. *High-Level Synthesis*. [online] Available at: <https://en.wikipedia.org/wiki/High-level_synthesis> [Accessed 16 May 2020].

[7] Fastmachinelearning.org. 2020. *HLS4ML · Gitbook*. [online] Available at: <https://fastmachinelearning.org/hls4ml/> [Accessed 16 May 2020].

[8] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, 2018, pp. 1-14, doi: 10.1109/ISCA.2018.00012.

[9] Cernbox.cern.ch. 2020. *Cernbox*. [online] Available at: <https://cernbox.cern.ch/index.php/s/AgzB93y3ac0yuId> [Accessed 16 May 2020].

[10] Xilinx.com. 2020. [online] Available at: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf> [Accessed 16 May 2020].

[11] Gallinaro, M., 2013. Top quark physics: A tool for discoveries. *Journal of Physics: Conference Series*, 447, p.012012.

[12] Colah.github.io. 2020. *Understanding LSTM Networks -- Colah's Blog*. [online] Available at: <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> [Accessed 16 May 2020].

[13] CERN Accelerating science. (n.d.). Retrieved from https://home.cern/science/accelerators/large-hadron-collider

[14] Hep.ucl.ac.uk. 2020. [online] Available at: <https://www.hep.ucl.ac.uk/seminars/slides/11-01-2019-Ngadiuba_UCLSeminar_HLS4ML.pdf> [Accessed 27 May 2020].

[15] 2020. [online] Available at: <https://www.nature.com/articles/s41586-018-0361-2> [Accessed 29 May 2020].

[16] En.wikipedia.org. 2020. *Neural Network*. [online] Available at: <https://en.wikipedia.org/wiki/Neural_network> [Accessed 29 May 2020].

[17] Nwankpa, Chigozie Enyinna, et al. *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*. 8 Nov. 2018, arxiv.org/pdf/1811.03378.pdf.

10. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1934360.

The work presented in this document is a result of exuberant MS journey that involved numerous individuals contributing to its culmination.

First and foremost, I would like to thank my advisors, Scott Hauck and Shih-Chieh Hsu who have been pivotal since the beginning. Their constant support and advice have helped me both in my professional and personal growth. The opportunity to work at CERN is one of my most cherished memories and I owe it to my advisors. I am grateful for having them as my guides during this tumultuous journey.

A big shout out to the HLS4ML team where I was a part of something bigger than myself. The work being done, and the individuals have inspired me to work hard and achieve what I have achieved today. The learnings and experience from the team will last a long time to come.

I am grateful to Nhan Tran for onboarding me onto the team and introducing me to the project at the early stages that has developed so beautifully today. Thank you to Vladimir Loncar for get me accustomed to the HLS4ML framework and bearing with my constant queries that facilitated the integration of my work into the framework. My gratitude towards Philip Harris for helping me understand the base of my work: RNNs and LSTMs which acted as the foundation for my work presented herein.

I cannot be profuse enough in thanking my parents Shrinivas Naidu and Radhika Naidu for always believing in me and keeping me on my toes with their words of encouragement and support.

My undying love and graciousness for my sister, Varsha Rao, for always being by my side through the highs and the lows; picking me up, cheering me on and giving me the emotional support to get through all the tough times that I encountered.

Thank you to the rest of my family and friends who have been there, directly, or indirectly, guiding, supporting, and assisting me through a journey that was just as memorable as it was difficult.

The words of support, the advices that came as one-liners, the time you all have invested in me along with the effort, have all amalgamated to be my rock and my guiding light. My gratitude to you all is unfathomable and cannot be expressed in words.

Appendices

Appendix A: LSTM in HLS4ML

Relevant repositories:

- https://github.com/richarao/hls4ml/tree/keras-lstm
- https://github.com/hls-fpga-machine-learning/hls4ml/tree/recursive2

Presentations:

- https://indico.cern.ch/event/919468/
- https://indico.cern.ch/event/900197/
- https://indico.cern.ch/event/889243/

Appendix B: KERAS Training

In the work presented in my thesis, the KERAS LSTM model used for the application of top tagging was a trained model already saved in the HLS4ML repository. To ensure the model being used was as per specifications and trained on the right dataset, I performed the training of a new KERAS LSTM model. This training also came in handy in the final testing phase of the framework to ensure it could efficiently translate LSTM models with different configurations as well.

The location of the repository used for training was:

https://github.com/hls-fpga-machine-learning/keras-training

Below are some of the performance evaluation plots generated for this model. Fig 22 shows the ROC curves, Fig 23 shows the accuracy between training and validation dataset and Fig 24 shows the normalized confusion matrix.

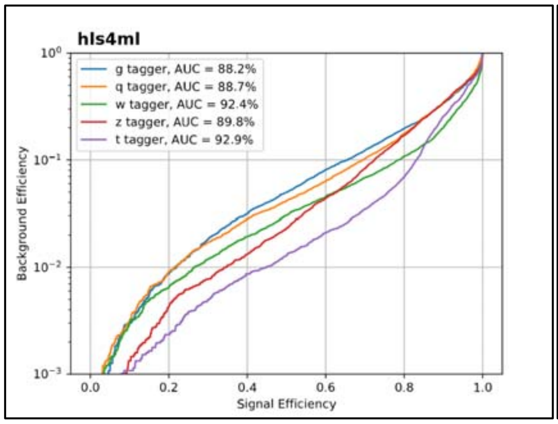


Fig 22: ROC Curves

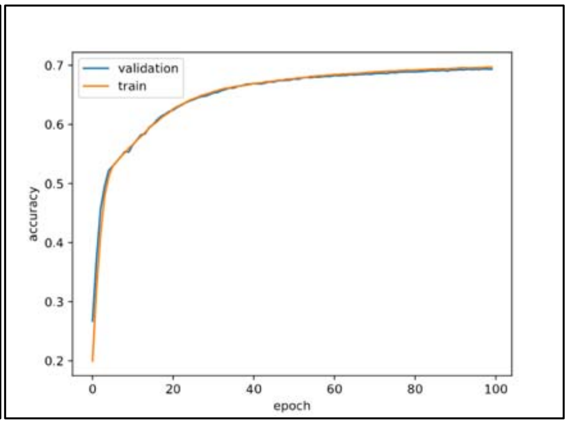


Fig 23: Accuracy

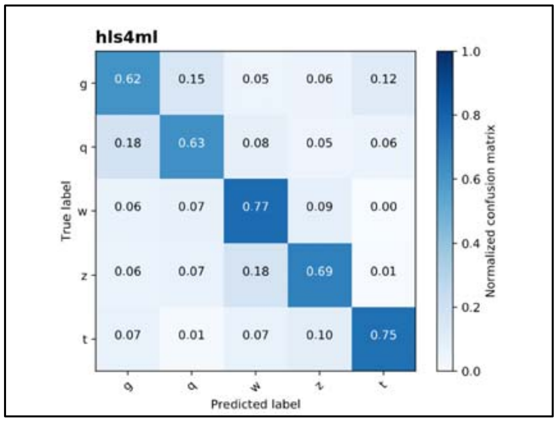


Fig 24: Confusion Matrix

Appendix C: Automation of Data Acquisition Testing

I spent my summer of 2019 interning at the European Organization for Nuclear Research (CERN), where I worked on a project entirely separate from my work on the HLS4ML framework. During my 3 months there, I worked on a project that focused on the automation of data acquisition testing.

Data acquisition is one of the most important tasks conducted at CERN during particle collisions. There are detectors present in different layers to capture this data. These detectors have firmware that is upgraded with time and needs to be tested to verify if the data being collected is accurate. This data is then passed through the post analysis phase.

I created a testing framework that would automatically perform checks on the current firmware and produce reports which could be used for the post analysis in case of any errors.

When data acquisition is performed manually, it could take a couple days as the checks use multiple tools. By automating the testing procedure, the functionality of the firmware can be determined within few minutes. Hence, it saves time taken to determine the error, which then could be used to analyze the cause of error instead.

Some of the checks conducted were:

- Fragment check
- BUSY signal check
- TIMEOUT signal check
- RodMon Checks
- ROD Registers
- BOC Registers
- FW/SW loading check

The DAQ Testing document that explains each of the above checks is located at:

https://gitlab.cern.ch/atlas-pixel/docs/pixeliblTestingnote

Presentations given during the internship:

- Project Introduction (23rd Jul): *https://indico.cern.ch/event/803420/*
- Update ppt (20th Aug): *https://indico.cern.ch/event/803422/*
- Final ppt (9th Sep): *https://indico.cern.ch/event/846497/*