UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Efficient Implementation of Recurrent Neural Network Accelerators

VIDA ABDOLZADEH

TUTOR: PROF. NICOLA PETRA

COORDINATOR: PROF. DANIELE RICCIO

XXXII CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Acknowledgments

I would like to thank my family, especially my mother, to whom I dedicate the entire thesis work. She has provided me through moral and emotional support in my life.

I thank Prof. Petra for the opportunity given to me and for his support during the thesis period, he has always been available to guide me in the development of the activities and ready to respond to my doubts about the project development.

I want also to thank the remaining professors of Electronics Group for their priceless lessons. I am also grateful to my friends who have supported me along the way.

Finally, I would like to thank the friends of the Electronics Group at DIETI, Antonio, Darjn Michele, Luca for their friendship and nice time spent together.

“One, remember to look up at the stars and not down at your feet. Two, never give up work. Work gives you meaning, and purpose and life is empty without it. Three, if you are lucky enough to find love, remember it is there and don't throw it away."

(Stephen Hawking)

Abstract

In this dissertation we propose an accelerator for the implementation of Long Short-Term Memory layer in Recurrent Neural Networks. We analyze the effect of quantization on the accuracy of the network and we derive an architecture that improves the throughput and latency of the accelerator. The proposed technique only requires one training process, hence reducing the design time. We present implementation results of the proposed accelerator. The performance compares favorably with other solutions presented in Literature.
The goal of this thesis is to choose which circuit is better in terms of precision, area and timing. In addition, to verify that the chosen circuit works perfectly as activation functions, it is converted in Vivado HLS using C and then integrated in an LSTM Layer. A Speech recognition application has been used to test the system. The results are compared with the ones computed using the same Layer in Matlab to obtain the accuracy and to decide if the precision of the Non-Linear functions is sufficient.

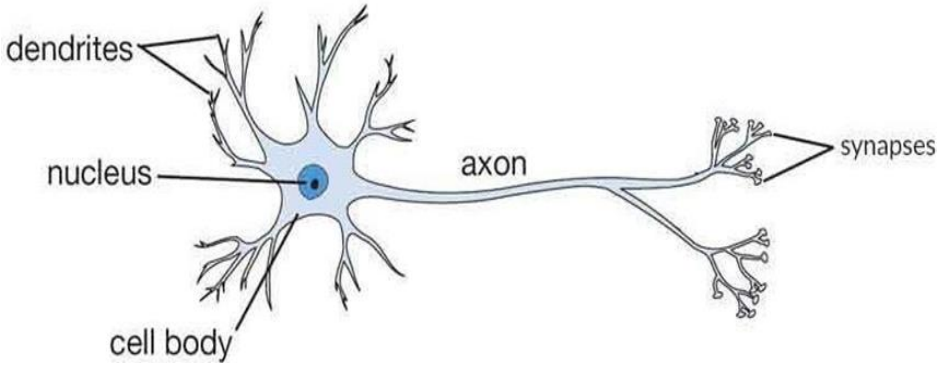
Chapter 1

State of Art

In this Chapter will be introduced the first idea of Neural Networks, starting from the basic model to the latest architectures.

1.1 Biological Neural Network

A Biological Neural Network is a series of interconnected neurons whose activation define a recognizable linear pathway. The interface through which neurons interact with their neighbors usually consists of several axon terminals connected via synapse to dendrites on other neurons. If the sum of the input signals into one neuron overcome a threshold, the neuron sends an action at the axon and transmits this electrical signal along the axon.



The first study of neuronal training was presented by Hebb [1] in 1949, while neuroscientists McCulloch and Pitts [2, 3] showed theoretically that networks of artificial neurons could implement logical, arithmetic and symbolic functions. These studies inspired the Artificial Neural Networks.

1.2 Artificial Neural Network

Artificial Neural Networks (ANNs) are mathematical models inspired by biological neural networks that constitute animals' brain. These models take shape in computing systems that learn tasks by considering examples, generally without a simplified version of biological animals' neurons, and the connection between them it's called **Synapse**.
In a Synapse, an artificial neuron can transmit a signal to another artificial neuron and, this one, can process it and send to another neuron link to it. The signal, usually, is a real a number and the output of each artificial neuron is the result of a non-linear function of the sum of its inputs.
A weight is assigned to classify the strength between the neurons, and its value decreases or increases during the learning process. A threshold is used to guarantee that a signal is sent only when it's sufficiently strong, in this way, a lot of resources are saved.
ANNs are usually organized in **layers**, as it's possible to see in Figure 1.

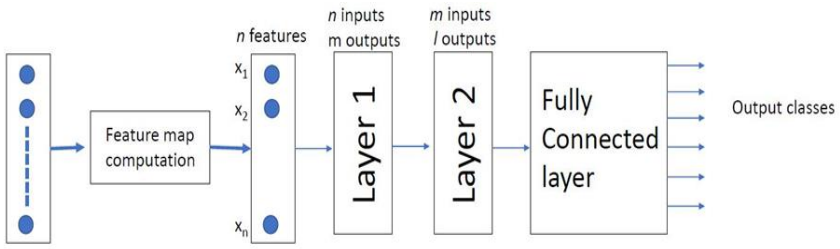


Fig 1. A simple structure of an ANN

The input of the ANN is a vector data obtain from the pixel's image or sample audio. The vector is sent to a Feature Map Computation block that provides the extraction of the Feature Map if the input is an image. If the input is a sample audio, the Feature Map Computation block will

give N **Cepstrum**, where a cepstrum is the result of the Inverse Fourier Transformation of the logarithm of the signal spectrum.

The Features (or Cepstrum) are sent to the first layer. A layer is usually composed of M linear functions, so its output will be a vector of M elements.
The number of layers it's choice according to the application, and this problem is solved during the design part of an ANN.
The last layer is called Fully Connected Layer (Figure 2), because it's connected to all the output of the previous one, and its task is to calculate the result [4].

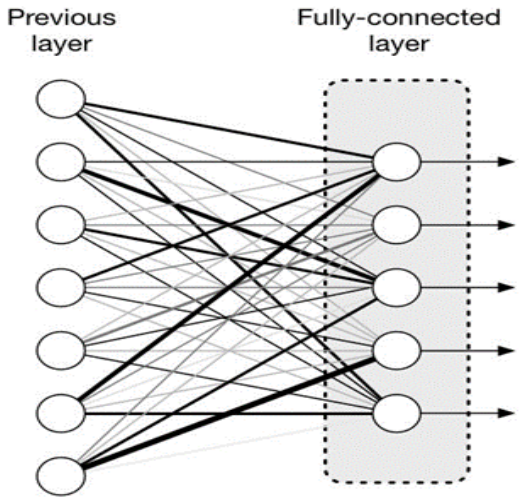


Fig 2. An example of Fully Connected Layer

During the learning proceed, a Test set is given as input to the ANN to allow the balance of the weight of the neurons and the correct bias of the entire system. To avoid "Overfitting" problem, usually a shuffle or a resort of the data input is needed.

Originally thought to solve problems in the same way the human brain would, ANNs are now focusing on specific tasks like Speech Recognition, Computer Vision or medical diagnosis. They are usually implemented by software, but in the latest years, thanks to the new technologies in the Electronic field and the research for new models of

neural networks, the design of an ANN that works totally in hardware and Offline is becoming the new challenge, in particular for the Recurrent Neural Networks (RNNs).

1.3 Recurrent Neural Network

In traditional ANNs, the data assigned as inputs are assumed all independent of each other, but that idea doesn't work for all the task. The concept behind the Recurrent Neural Networks (RNNs) [5] is to use sequential information. For example, to predict the next letter in a word, it's better to know the previous one.

The term Recurrent is used because this type of Neural Network performs the same task on every input element of sequence, with the output being depended on the previous computation. In literature, it's easy to find RNNs associated with a memory behavior, because, in theory, they can store almost every step of previous computations, but in practice, they can only store a few steps back. A typical RNNs is shown in Figure 3:

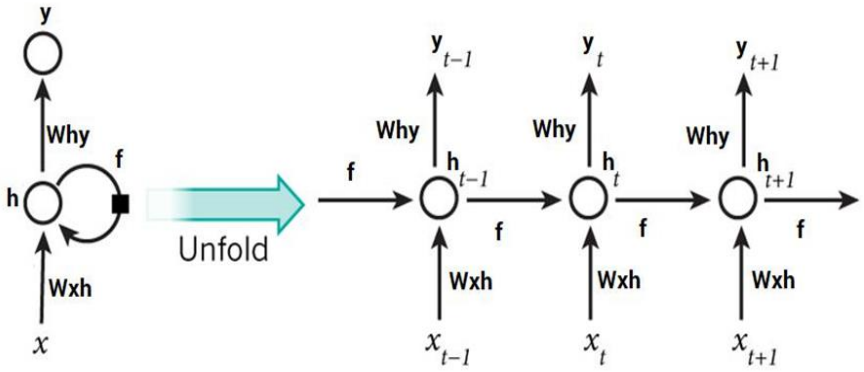


Fig 3. Example of a structure of an RNNs

The *unfold* of the RNN structures, show how it's very similar to a Convolutional Neural Network. The number of layers is equal, for example, to the number of data in the sequence given as inputs.

In case of forwarding propagation, the input moves through the layers at each time step, while in back-propagation, it's like going back in time

to change the weight, so it's called **Back Propagation through Time** (BPTT). In the next paragraphs, the story and the state of art of RNN will be illustrated, while the equations and the functionality of the RNN will be reported later in this Chapter [6, 7].

1.3.1 Hopfield Network

In 1980, the physicist John Hopefield published a paper where he describes the first RNN, that will become popular as Hopefield Network. This network is based on the ability of the human brain to recognize an image also when this is wrong or corrupted thanks to the associative memory. Hopfield tried to replicate the associative memory using the structure in Figure 4:

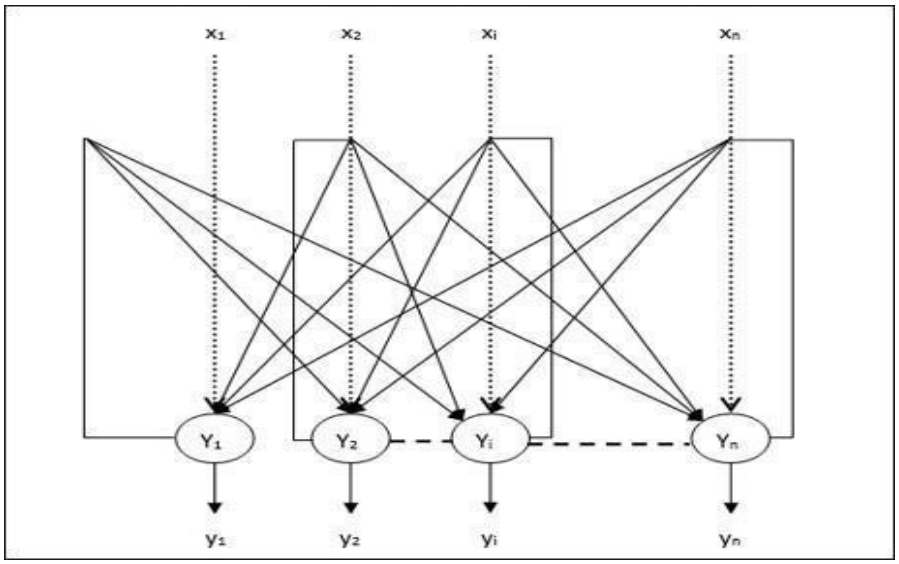


Fig 4. Hopfield's Network Structure

The units are binary and usually are -1 and 1, but sometimes the net is implemented with units like 1 and 0. Every connection between units has a weight w_{ij} that represents the link between the i and j artificial neuron.
The weights have the following constraints:

- w_{ii}=0, ∀i, means that every unit have no connection with itself;

- $w_{ij}=w_{ji}$, ∀i,j, means that the connections are symmetric. In this way, it's guaranteed that the weights are symmetric and chaotic behavior are avoided. As a result, the net should converge to a local minimum, but the learning process it's not easy, so often its converge to a false local minimum. [8]

1.3.2 Elman and Jordan Network

An Elman network is a three-layer network with the addition of a set of units (c0, c1, c2 in Figure 5a). The middle layer is called Hidden Layer and it's connected to the units c0 with a weight of one. At each step, the input is moved forward, and a learning rule is applied, while the value of the previous hidden layer state is saved in the unit c0. In this way, the net can maintain a sort of state, and that's a necessary condition to perform a task like sequence prediction.

The Jordan Network (Figure 5b), it's very similar to the Elman one. The main difference is that the units c in the back-propagation chain are connected to the outputs instead of the hidden layers.

These two networks are usually called Simple Recurrent Networks (SNRs). [9]

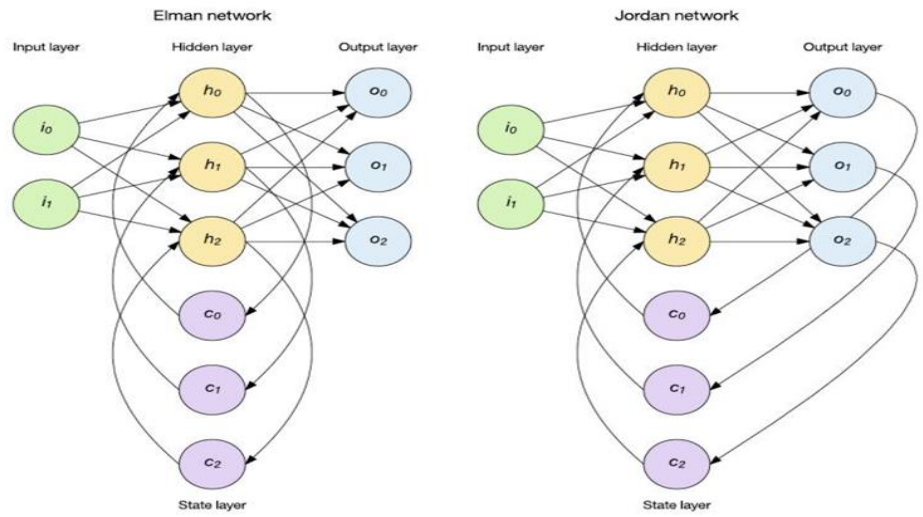


Fig 5. a) Elman network structure b) Jordan Network structure

1.3.3 Neural history compressor

All the RNNs described before, suffers from a problem called Vanishing Gradient Problem. In machine learning, this problem is encountered in Neural Networks based on gradient-based learning methods and back-propagation.

In a typical method, the weights in the Neural Networks receive an update proportional to the gradient of the error function, considering also the current weight in each iteration training. In some cases, the gradient is too small (from here the term Vanishing), preventing the change of the weight's value.

This problem is mainly due to the activation functions like hyperbolic tangent function, which have a gradient in the range (0,1). So, during the learning process, there will be *n* multiplication for small numbers to compute gradients in an *n*-layer network, that means that the error signal (gradient) decreases exponentially with n, while the front layers slowly its training.

To avoid this problem, in 1992, a generative model called the **Neural History Compressor**, implement as an unsupervised stack of RNNs. At the input level, it learns to predict its next input from the previous inputs.

Not all the inputs become the inputs of the next higher level RNN, but only the unpredictable inputs of some RNNs in lower lever, in this way, the entire system recomputed its internal state rarely. The RNN in the higher-level studies a compressed representation of the information of the RNN below, so the input sequence can be reconstructed from the representation at the highest level.

The system effectively reduces the description length or the negative logarithm of the probability of the data. With this approach, the higher level RNN can be supervised learning to easily classify even deep sequence with long intervals between events [10, 11].

In 1993, Jürgen Schmiduber solved a Very Deep Learning task that required more than 1000 subsequent layers in an RNN unfolded in time using this system.

1.4 Long-Short Term Memory

Long Short-Term Memory (LSTM) units were proposed by Sepp Hochreiter and Jürgen Schmiduber to avoid the vanishing gradient problem when training traditional RNNs. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. In figure 6, a common LSTM's architecture is shown. [12, 13]

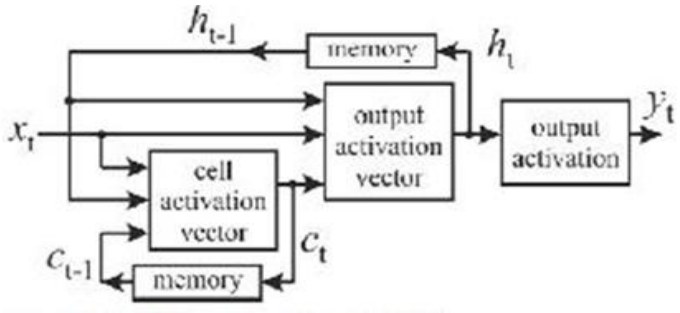


Fig 6. A common LSTM's architecture

The cell is in charge of remembering values for an arbitrary time interval; this is the reason for the word memory in the name. Each gate can be thought as an artificial neuron, while the term gate derives from the fact that they work as regulators of the flow of the values that goes through the connections of the LSTM.

Every gate uses an activation function to compute a weighted sum. The expression Long Short-Term is due to the possibility to store short term (like the short memory in the human brain) for an extended period. For this reason, LSTM are suited to classify, process and predict time series given time lags of unknown size and duration between important events.

Thanks to its characteristics, LSTM units are taking place in many applications over other RNNs like Hidden Markov models (a system based on the Markov Process).

LSTM units are very common for solving speech recognition problems. Google, Apple and Microsoft using LSTM as fundamental units in their products. As an example, Google is using LSTM for the smart assistant Allo in its smartphones and for Google Translate, Apple

and Amazon are doing the same for their smart assistant Siri and Alexa respectively. In 2017, Microsoft reaching 95.1% recognition accuracy on the Switchboard corpus, incorporating a vocabulary of 165000 words, using as approach dialog session long short-term memory. [6, 14-16]

1.4.1 LSTM: Architecture

As said in the previous paragraphs, RNNs can learn from past information. The question is: how long an RNN can and what should remember? An RNN standard, can store and use recent information, but cannot learn long-term dependencies. Furthermore, it's very difficult to train it due the gradient vanishing problem. This is the point where the

LSTM filled the gap. In an LSTM unit, it is an RNN with an explicit memory controller that decide what remember and what forget. In this way, the learning process is more stable and allows to the system to handle long dependencies in sequences.

There are many variants of LSTM architecture. The vanilla version is showed in the following picture.

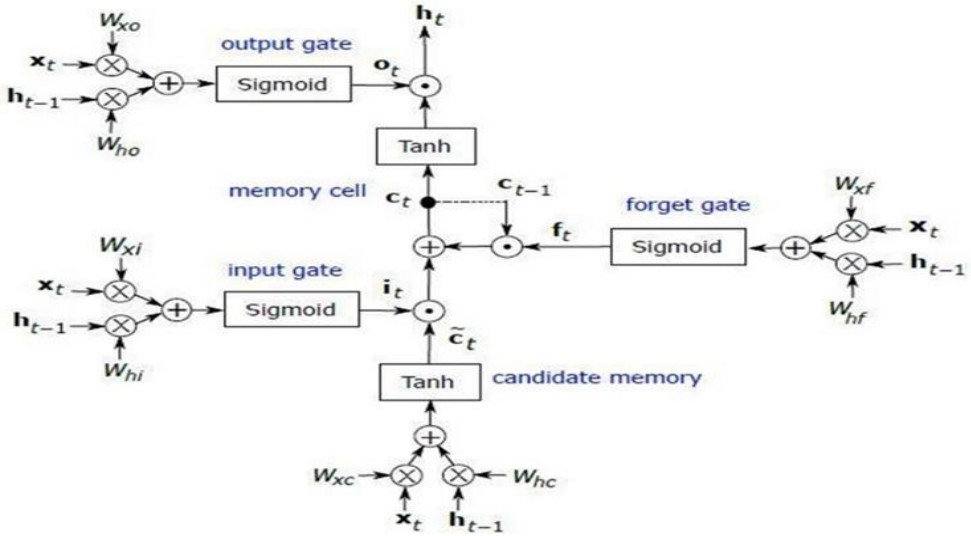


Fig 7. LSTM vanilla architecture

As shown in the figure, the memory cell influences the input, the forget and output gate. This architecture is taken as reference for the hardware implementation and testing described in this thesis. To understand better how this system works, it's necessary to define the following equations:

$$i_t = \sigma(W_{ix} \times x_t + W_{ih} \times h_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx} \times x_t + W_{fh} \times h_{t-1} + b_f) \quad (2)$$

$$c_t = f_t \cdot c_{t-1} + i_t . g(W_{cx} \times x_t + W_{ch} \times h_{t-1} + b_c) \quad (3)$$

Where W is constant weight matrices designed during the training process of the network, × is the matrix multiplication, represents the element-wise multiplication, b are bias vectors, is the logistic function sigmoid, g is the input activation function, i_t and f_t are the input gate and forget gate respectively. The computation of the output sequence is based on the following equations:

$$o_t = \sigma(W_{ox} \times x_t + W_{oh} \times h_{t-1} + b_o) \quad (4)$$

$$h_t = o_t \cdot H(c_t) \quad (5)$$

$$y_t = \Phi(W_{yh} \times h_t + b_y) \quad (6)$$

where o_t is the output gate, H is the output activation function and Φ is the SoftMax operation. The output activation function H and the input activation function g can be defined in several ways. In this thesis, they are both the hyperbolic tangent function.

In the next chapter, will be discussed the numerical analysis for the implementation in hardware of the hyperbolic tangent function and the sigmoid function. [6, 12, 14-18]

Chapter 2
Signal quantization

An aggressive quantization allows obtaining an efficient implementation of equations (1)-(6) but it also affects the accuracy of the network.

The quantization can be taken into account during the training process. However, we propose to apply quantization after the network has been trained. The advantage of this technique is that the design time of the accelerator is lower. As we will show, our approach introduces a negligible accuracy loss.

The search of the optimal quantization for a given target accuracy is not a straightforward task since it requires to fix independently the number of bits used for each one of the 7 signals in the network (the hidden state, the cell activation, the three gates, the two hyperbolic tangents) and the 12 constant matrixes/vectors because these values have different dynamic range.

In order to reduce the search space, we define two parameters: the maximum variable error (MVE=2^{-M}) and the maximum constant error (MCE=2^{-L}). L is optimal spot for MCE and M is optimal spot for MVE.

MVE is the maximum error allowed on the representation of each variable signal *s* in eq. (1)-(6). If we define $\hat{s}$ as the quantized version of the signal *s* we have:

$$|\hat{s} - s| \leq \text{MVE}\ \forall s \in \{h_t, i_t, f_t, c_t, o_t, g_t, \tanh(c_t)\} \quad (7)$$

MCE is the maximum error allowed on the representation of each weight *w* of each weight matrix. If we call $\hat{w}$ the quantized representation of the weight *w* we have:

$$|\hat{w} - w| \leq \text{MCE}\ \forall w \in W_{\text{ix}} \cup W_{\text{fx}} \cup W_{\text{cx}} \cup W_{\text{ox}} \cup W_{\text{ih}} \cup W_{\text{fh}} \cup W_{\text{ch}} \cup W_{\text{oh}} \quad (8)$$

The quantized representation $\hat{b}$ of a bias value b is obtained according to the following rule:

$$\left|\hat{b} - b\right| \leq \text{MCE·MVE}\ \forall b \in\ b_{\text{i}} \cup\ b_{\text{f}} \cup\ b_{\text{c}} \cup\ b_{\text{o}} \qquad (9)$$

In order to show the effect of our quantization scheme we have designed and trained two RNNs. The first network is based on the scheme in Fig. 8(*a*) and is used to identify a speaker among 9 possible candidates.

For this network Fig. 8(*a*) Z is equal to 12 and N is equal to 50. The second network is based on the scheme of Fig. 8(*b*). The network is used to predict the monthly occurrence of chickenpox on the basis of previous history. For this network Fig. 8(*b*) Z is equal to 1 and N is equal to 200. The training and the test sequences of the two networks are available on-line [19], [20].

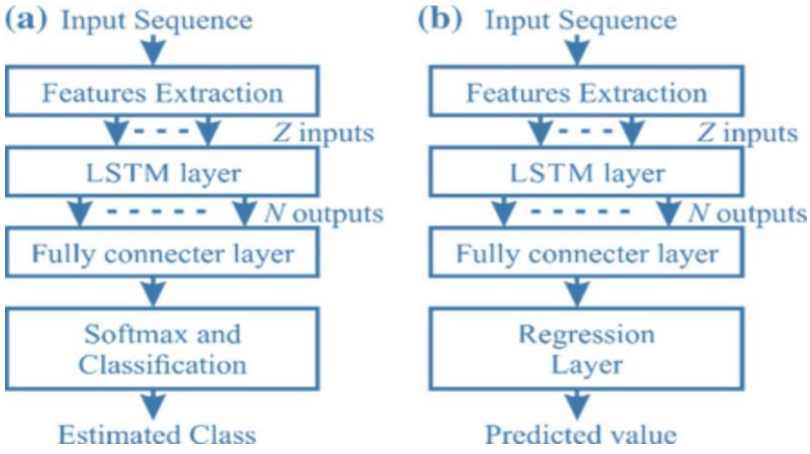


Fig. 8 Architecture of a RNN: **a** RNN used for sequence classification, **b** RNN used for data prediction

We have trained both networks using floating-point representation for each variable signal and each constant factor in the equations (1)-(6). The training operation has been performed using Matlab. After the training process we have applied the constraint (8) on the weight factors.

Fig. 9 shows the dependency of the accuracy of the first RNN on the MCE. The accuracy is computed as the percentage of correct speaker identification over the entire test set. As can be seen, decreasing the MCE, the accuracy of the network improves.

However, the result in Fig. 9 shows that there is an optimal value for MCE. Reducing the MCE under the optimal spot does not increase the accuracy of the network. As can be seen L=5 allows achieving the same accuracy of the floating-point representation. Fig. 10 shows the result of a similar analysis performed on the second RNN.

Here the accuracy is computed as the root mean square error between the value predicted by the network and the actual value of the series. Again, as can be seen, there is an optimal spot that can be used to fix the value of L. Increasing the value of MCE not only allows reducing the number of bits used for the weight factors, it also allows reducing the overall number of non-zero constant weights.

Increasing the value of MCE not only allows reducing the number of bits used for the weight factors, it also allows reducing the overall number of non-zero constant weights. Fig. 11 shows the number of non-zero values as a function of MCE for the first RNN. As can be seen, the overall number of non-zero coefficients reduces by 50% at the optimal spot (the one chosen in Fig.9 Similar considerations can be done on the second RNN.

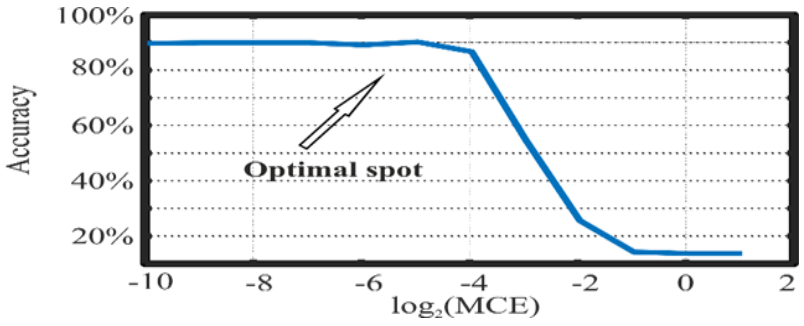


Fig. 9 Accuracy vs MCE for the classification RNN

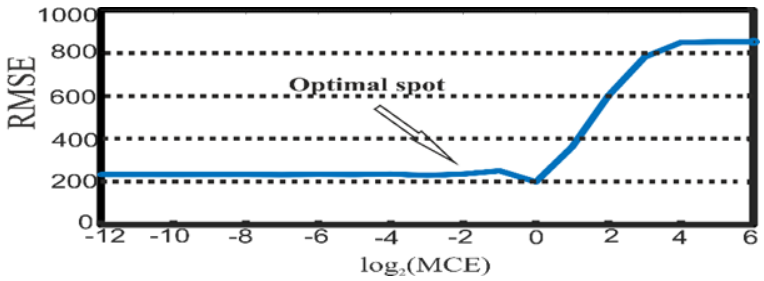


Fig. 10 RMSE vs MCE for the forecasting RNN

Once we have found the optimal spot for the MCE we can apply the constraints (7) and (9) on the variable signals and the bias values respectively. Fig. 12 shows the relation between accuracy, RMSE (Root Mean Square Error) and MVE for both networks. In this figure, L is fixed at the optimal spot as can be seen an optimal spot can be found for MVE as well and hence for M.

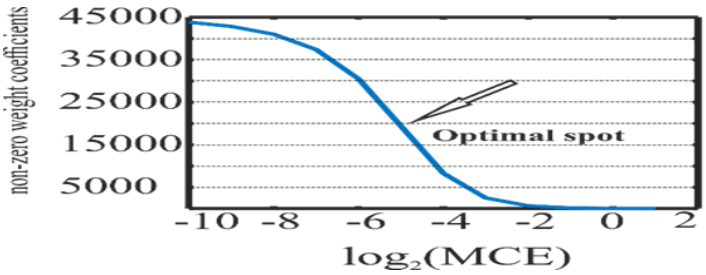


Fig. 11 non-zero constant weights vs MCE for the classification RNN

The use of the optimal spot is a technique that can be used for the quantization in any RNN. It allows to reduce the size of the signals in the accelerator. It also allows reducing the number of constants that must be stored in the internal memory of the accelerator. Overall, the loss on the accuracy of the network is neglectable.

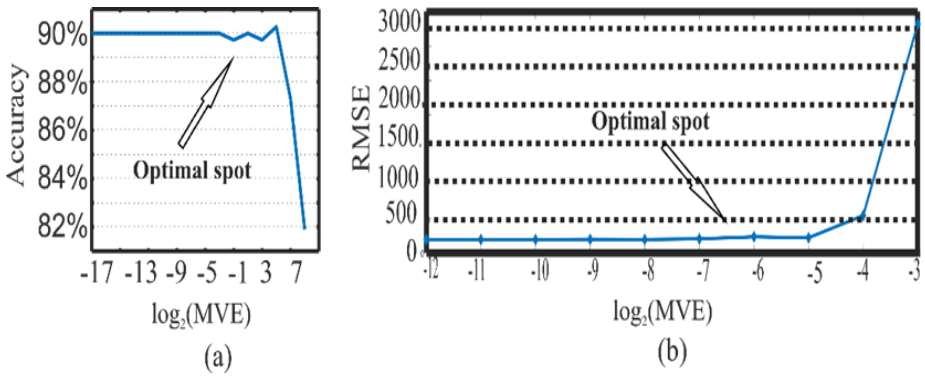


Fig. 12 accuracy vs MVE. (a) Accuracy of the classification RNN (b) RMSE of the forecasting RNN

2.1 Quantization effects on the dynamic behavior of the network

The quantization also affects the dynamic behavior of the neural network. Figures 13-16 show the dynamic behavior of the neural network designed to recognize a speaker among 9 possible ones. Each figure reports the output of the network as it changes while the network is processing an input sequence.

The dashed line represents the correct classification for the provided input sequence. The circled values are obtained using a floating point based neural network while the crossed values are obtained with the quantized neural network designed using the optimal spot.

As shown in Fig.13 and 14 for some input sequences the behavior of the floating point and the quantized neural network remains the same. However, as shown in Fig.15 and 16, there are cases where the quantization changes the dynamic evolution of the network, but it does not change the final result.

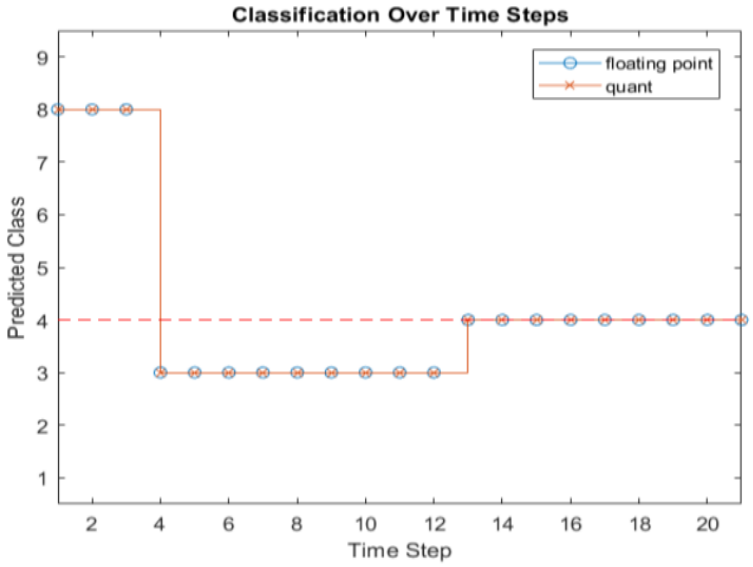


Figure 13. Output of the neural network for a given input sequence

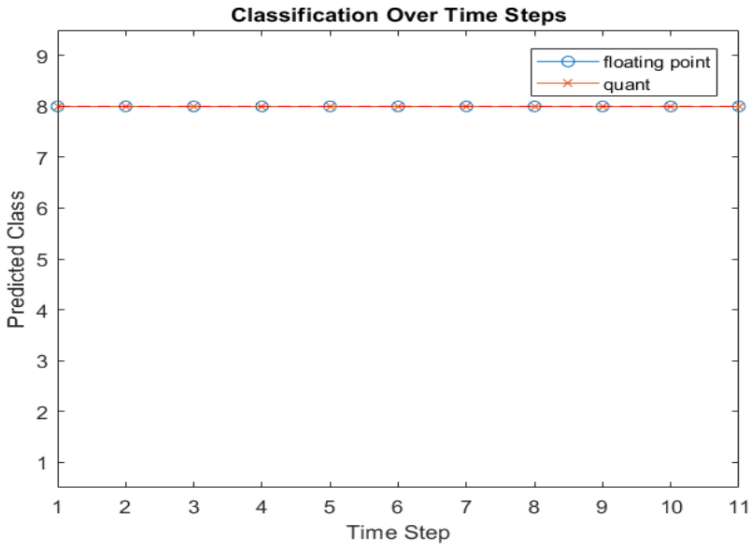


Figure 14. Output of the neural network for a given input sequence

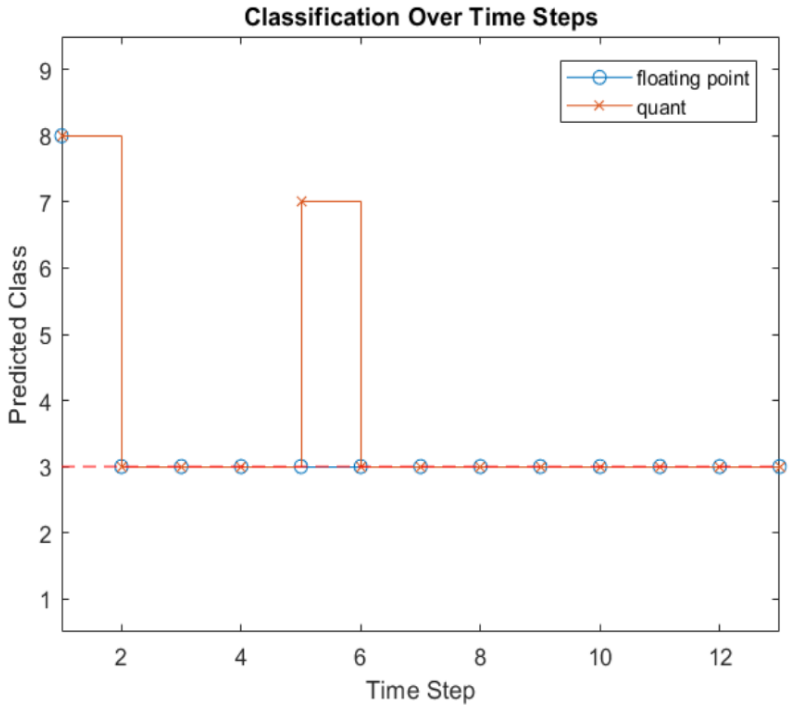


Figure 15. Output of the neural network for a given input sequence

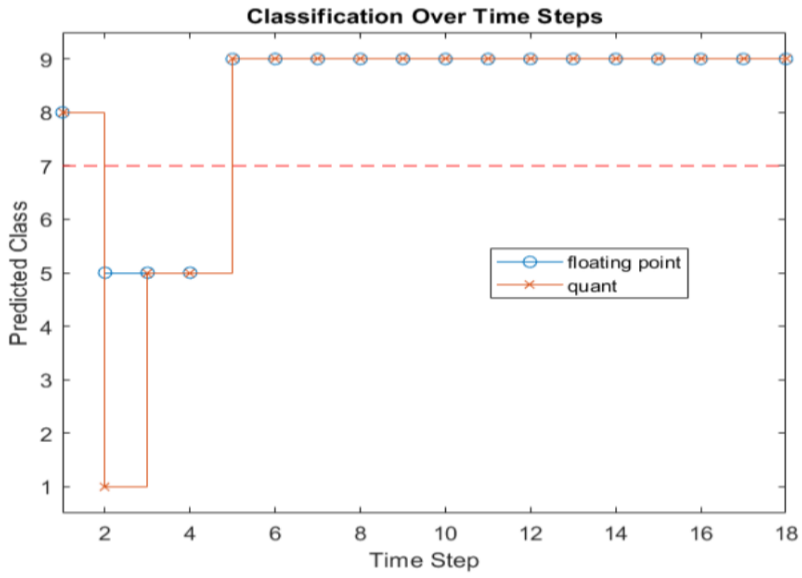


Figure 16. Output of the neural network for a given input sequence

2.2 Circuit implementation

The direct implementations of Eqs. (1) – (6) requires the use of two memories to store the values of h_{t-1}, and c_{t-1},. However, in a SoC architecture, this choice is non-optimal.

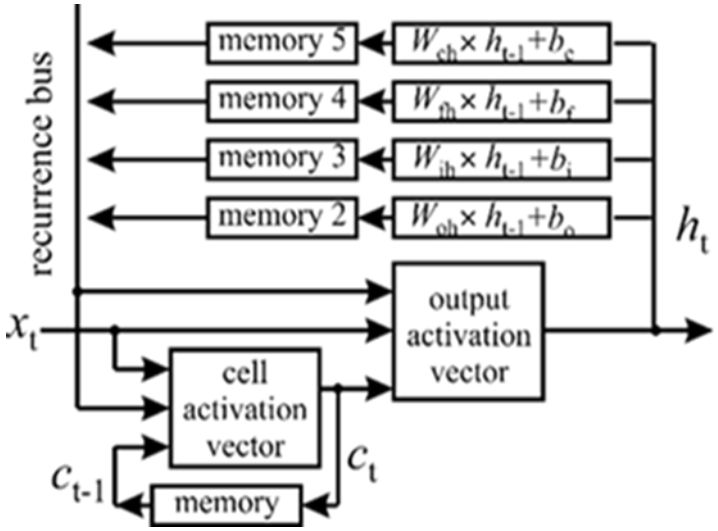


Fig. 17 Proposed architecture

In our architecture Fig.17 the Eqs. (2), (3), (4) and (6) are divided in two parts: the recurrence part that depends on ht-1 and the input part that only depends on x_t. Instead of storing h_{t-1}, we store the recurrence part of each of the four equations separately. This choice greatly improves the latency of the circuit and, because of the feedback architecture, the throughput of the accelerator. We have designed two accelerators.

Both accelerators implement the first RNN discussed in the previous section and can be used for the classification of speakers.

We have used high-level synthesis to synthesize the accelerator. We used Xilinx Zynq XC7Z020 as the target technology and Vivado HLS as the synthesis tool.

The accelerator receives the commands from an AXI4 lite compatible interface. The inputs and the outputs are read from and stored into external block-RAMs in order to keep the IP as fast as possible. We have fixed the clock frequency to 100 MHz to limit the maximum number of DSPs that can be cascaded in the data-path with no pipelining allowed.

Table 1. Accelerator performance

	SRAM	DSP	FF	Latency	Recurrence Latency	f_{clock}
Single Recurrence Mem.	387 Kb	11	30604	58K	31K	115 MHz
Multiple Recurrence Mem.	387 Kb	11	1157	30K	7651	115 MHz
[27]	86 Kb	96		40M	N.A.	200 MHz
[28]	17 Mb	1504	453K	16K	N.A.	200 MHz
[29]	288 Kb	50	13K	127K	N.A.	142 MHz

The first accelerator uses a single recurrence memory to store the value of h_(t-1). The achieved performance is shown in the first row of Table 1. 4 DSPs are used to implement the recurrence operations (the multiply-and-add operation involving h_t) plus 7 DSPs for the remaining operations.

The second accelerator is based on the architecture of Fig. 17 and uses 4 recurrence memories. The computation of the recurrence operation is obtained with a parallel data path, using 4 DSPs. We

allowed 7 DSPs to be used in the computation of the other equations. The results are shown in the second row Table 1.

As shown, the 4 data-paths used to compute the recurrence operations allows to reduce the recurrence latency by 75%. Furthermore, the use of 4 memories allows the reduction of the overall latency by 49% with the same number of arithmetic units used. Compared with previous art, the proposed circuit has a very small foot-print and is suitable for efficient accelerators for IoT devices.

Chapter 3

3.1 Circuit Optimization

This chapter will address mathematical analysis to develop an effective algorithm to create a circuit that can perform the calculations necessary to implement a recurrent neural network. The RNN referred to is the one shown in the previous chapter, and LSTM in particular Vanilla Architecture. For simplicity I will report here the equations that define the LSTM layer.

$$i_t = \sigma\,(W_{ix} \times x_t + W_{ih} \times h_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma\,(W_{fx} \times x_t + W_{fh} \times h_{t-1} + b_f) \quad (2)$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot g\,(W_{cx} \times x_t + W_{ch} \times h_{t-1} + b_c) \quad (3)$$

$$o_t = \sigma\,(W_{ox} \times x_t + W_{oh} \times h_{t-1} + b_o) \quad (4)$$

Where Wix,Wfx, Wcx, Wox, Wih, Wfh,Wch,Woh represent the weight matrices. The latter are obtained directly in software during the training phase of the network, equivalently also the vectors of bias bi, bf, bc, bo are extracted during the training phase carried out in Matlab. xt represents the t input vector of the input sequence, the vector made up of the features, while g and σ are the input activation functions.

$$h_t = o_t \cdot H(c_t) \quad (5)$$

$$y_t = \Phi\,(W_{yh} \times h_t + b_y) \quad (6)$$

There are two output equations, H represents the output activation function while Φ SoftMax is the function. The goal is to provide a mathematical method for designing an LSTM-based RNN for any application.

3.2 Parallelization Algorithm

Within the equations above the most expensive calculations to be made are the products between matrix and vector. In this paragraph we focus on a first solution to do this in HW by trying to obtain a low Latency and that allows the increase in compute units (DSP) used to always have all devices in operation, Therefore, a maximum efficiency.

Focusing first on the matrices, W_{ih}, W_{fh} , W_{ch} , W_{oh} the first information we have about them is that they are matrices square. This information is not really of any relevance for the calculation method but in the following way it will be able to facilitate some accounts.

Define

imem: $W_{ih} \times h_t$

fmem: $W_{fh} \times h_{t-1}$

cmem: $W_{ch} \times h_{t-1}$

omem: $W_{oh} \times h_{t-1}$

and I'll call **M** the number of columns in the following arrays, Y the number of rows, and the product M*Y=N. In the case that I'm going to treat the arrays are square so M=Y also the size of the vectors h_{t-1} is M elements.

The product between these arrays and the h vectors will in turn give vectors of size M that I mentioned earlier with imem, fmem, cmem, omem. The algorithm for carrying out the product between matrix and vector is known:

$$imem_i = \sum_{j=1}^{M} Wih\ (i,j) * h_{t-1}(j)$$

$$imem_i = \sum_{j=1}^{M} Wih\ (i,j) * h_{t-1}(j)$$

$$cmem_i = \sum_{j=1}^{M} Wch\ (i,j) * h_{t-1}(j)$$

$$cmem_i = \sum_{j=1}^{M} Wch\ (i,j) * h_{t-1}(j)$$

$$imem_i = \sum_{p=0}^{\mu_n} \sum_{j=1+p\varphi_n}^{(1+p)\varphi_n} Wih\ (i,j) * h_{t-1}(j) \quad with\ i \in [1 \dots M]$$

$$imem_i = \sum_{p=0}^{\mu_n} \sum_{j=1+p\varphi_n}^{(1+p)\varphi_n} Wih\ (i,j) * h_{t-1}(j) \quad with\ i \in [1 \dots M]$$

$$cmem_i = \sum_{p=0}^{\mu_n} \sum_{j=1+p\varphi_n}^{(1+p)\varphi_n} Wch\ (i,j) * h_{t-1}(j) \quad with\ i \in [1 \dots M]$$

$$omem_i = \sum_{p=0}^{\mu_n} \sum_{j=1+p\varphi_n}^{(1+p)\varphi_n} Woh\ (i,j) * h_{t-1}(j) \quad with\ i \in [1 \dots M]$$

The idea is to calculate the i-th element by unpacking the M sums of products into a number Δ of sums, which in turn are sums of products.

$$i_{jk} = w_{ih}\ (j,k) * h_{t-1}(k)$$

$$f_{jk} = w_{ih}\ (j,k) * h_{t-1}\ (k)$$
$$o_{jk} = w_{ih}\ (j,k) * h_{t-1}\ (k)$$

3.3 Algorithm 1

We calculate in the base time unit **T**, which represents the clock period, n products in the order i,f,c,o. Example N=5 DSPs

T=1	T=2	
i_{11}	f_{12}	
f_{11}	c_{12}	
c_{11}	o_{12}	
o_{11}	I_{13}	
i_{12}	f_{13}	

After that you continue in the same way, as you can see in the table schematization. This type of scheme means that every clock shot all DSPs are performing an operation. You continue to calculate products if o_{jk} is not found in the last row in the table schematization.

T=1	T=2	T=3	T= 4
i_{11}	f_{12}	c_{13}	r_{14}
f_{11}	c_{12}	o_{13}	i_{15}
c_{11}	o_{12}	i_{14}	f_{15}
o_{11}	i_{13}	f_{14}	c_{15}
i_{12}	f_{13}	c_{14}	o_{15}

In the 5 DSPs example, this happens after a time of T=4 per k=5, that is, after 4 stroke of clock we calculated the products up to the fifth element on each of the 4 arrays. After this first phase in the second phase we calculate the sum of these k products.

T=5	T=6	T=7	T=8
imem	*fmem*	*cmem*	*omem*
$= \boldsymbol{i_{11}} + \boldsymbol{i_{12}} + \boldsymbol{i_{13}} + \boldsymbol{i_{14}} + \boldsymbol{i_{15}} + imem$	$= \boldsymbol{f_{11}} + \boldsymbol{f_{12}} + \boldsymbol{f_{13}} + \boldsymbol{f_{14}} + \boldsymbol{f_{15}} + fmem$	$= \boldsymbol{c_{11}} + \boldsymbol{c_{12}} + \boldsymbol{c_{13}} + \boldsymbol{c_{14}} + \boldsymbol{c_{15}} + cmem$	$= \boldsymbol{o_{11}} + \boldsymbol{o_{12}} + \boldsymbol{o_{13}} + \boldsymbol{o_{14}} + \boldsymbol{o_{15}} + Omem$

The algorithm should be iterated Δ times to get the first element of imem, fmem, cmem, omem. In the example seen, with n=5 DSPs we run out 5 elements of the first line or in general of the i-th line. The number of products calculated in the first iteration of the algorithm is what we call φ_n.

The algorithm should be iterated Δ times to get the first element of imem, fmem, cmem, omem. In the example seen, with n=5 DSPs we run out 5 elements of the first line or in general of the i-th line. The number of products calculated in the first iteration of the algorithm is what we call φ_n. In the example with 5 DSPs, the time it takes to calculate the first φ_n products for each array is called γ.

$$\varphi_n = \frac{n}{4} * \gamma_n$$

$$\gamma_n = \begin{cases} 4 & if \ \ n \ is \ an \ odd \ number \\ 2 & otherwise \ n \ is \ a \ non-multiple \ even \ of \ 4 \\ 1 & if \ n \ is \ a \ multiple \ \ of \ 4 \end{cases}$$

I can make a further simplification by introducing $\alpha_{n\,=\,\gamma_n/4}$ such a way as to get

$$\varphi_n = n * \alpha_n$$

$$\alpha_n = \begin{cases} 1 & if\ \ n\ is\ an\ odd\ number \\ 1/2 & otherwise\ n\ is\ a\ non-multiple\ even\ of\ 4 \\ 1/4 & if\ n\ is\ a\ multiple\ \ of\ 4 \end{cases}$$

If I define L_i latency of the single iteration, it is equal to $\boldsymbol{2\gamma_{n\ =8\ \alpha_n}}$, while the latency to calculate the i-th element of imem, fmem, cmem, omem is equal to $\boldsymbol{L_r = L_i * \Delta}$. The total latency to get imem, fmem, cmem, omem, that is to get the 4 array products per vector will then be

$$\boldsymbol{L_{tot}} = \boldsymbol{L_r} * \mathbf{Y} = \boldsymbol{L_r} * \mathbf{M} = \boldsymbol{L_i} * \Delta * \mathbf{M}$$

with $\Delta = M/\boldsymbol{\varphi_n}$ for which $\boldsymbol{L_{tot\,=}\ L_i M^2/\varphi_n} = \mathbf{8\alpha M^2/(n\alpha)} = \mathbf{8M^2/n}$. For increasing the number of DSP **n** not to lose effectiveness, it must be $\boldsymbol{\varepsilon_{n'n''}} = \frac{\boldsymbol{L_{n'}}}{\boldsymbol{L_{n''}}} = \frac{\boldsymbol{n''}}{\boldsymbol{n'}}$ with n'<n'' where the subscript at the base of L indicates the number of DSP used. In the particular case where n" = 2n' should

$$\boldsymbol{\varepsilon_{n'n''}} = \frac{\boldsymbol{L_{n'}}}{\boldsymbol{L_{n''}}} = 2$$

Or

$$\boldsymbol{L_{n''} = L_{n'}/2}$$

Or doubling the number of DSP halves the latency. All of this I can also express by introducing another parameter that I call the **efficacy**

line = ϑ_n = $L_n * n$. If this line is constant in the plane (n, ϑ_n) then the latency decreases proportionally to them as the number of DSP increases. In the case of the algorithm just described it results

$$\vartheta_n = 8\,M^2$$

And since M is fixed the line of effectiveness is constant. If I call $\psi = n''/n'$ I can define the performance as a function of the DSP used $\eta_{n'n''}$ = 100 $\varepsilon_{n'n''}/\psi$.

In the treated algorithm $\varepsilon_{n'n''}$ = ($8M^2/n'$) $*$ ($n''/8M^2$) = n''/n' = ψ from which $\eta_{n'n''}$ = 100 %. Supposing to have n = 6 and M = 48 the total latency is of L_{6tot} = 3072 clock shots. So, to calculate all 4 matrices using 6 DSPs it takes 3072 clock strokes using this algorithm. The following table shows the latency values in function of the number of DSPs with 4<n<48 and M= **48** such that the **mcm** (φ_n, M) = M.

DSPs	Latency
6	3072
8	2304
12	1536
16	1152
24	768
32	576
48	384

The flaw of this algorithm lies in the fact that it does not exploit the potential of individual DSP. In fact, each DSP in a single stroke of clock can make a MAC, while in this algorithm it is used only to sum or product and never sum and product.

3.4 Algorithm 2

We introduce the second algorithm assuming to work in this case on a single matrix, for example suppose we want to calculate only imem, consequently afterwards we can extend the reasoning also for fmem, cmem, omem. It starts from the case in which I have only 1 DSP available and I want to use it more effectively by performing MAC operations such as a * b + c. The same notation used in the previous paragraph applies. Calculate the i-th element of imem:

(for ease of notation I used i_m instead of imem)

T=1	T=2	T=3	...	T=M
$i_m = 11$	$i_m = i_{12} + i_m$	$i_m = i_{13} + i_m$	...	$i_m = i_{1M} + i_m$

As you can see in this case except for the first clock shot, the only DSP used always performs MAC operations. The latency to obtain imem (1) will be equal to $L_r = M$. Now we use n = 2 DSPs instead

T=1	T=2	....	T=M/2	T=M/2
$t_1 = i_{11}$ $t_2 = i_{12}$	$t_1 = i_{13} + t_1$ $t_2 = i_{14} + t_2$	...	$t_1 = i_{1(M-1)} + t_1$ $t_2 = i_{1M} + t_2$	$i_m = t_1 + t_2$

Several memory elements (t_i) equal to the number of DSP are used. In this case, however in the last clock shot, n-1 DSP is used, in addition to the fact that in the first clock stroke no DSP plays a Mac. In the case of 2 DSPs the latency is equal to L_2=M/2 +1. You can generalize by finding the following formula:

$$\boldsymbol{L_n = \frac{M}{n} + 1}$$

Also in this case n, cannot be chosen at will but in fact we see that it must be M / n an integer, or n must be a submultiple of M which in other words can be written by setting the condition 1) p = M where p is given by the following expression $p = mcm\ (n, M)$ with n <M. The latency of imem is therefore given by the following expression as a function of n

$$L_n = \begin{cases} MY & for\ n = 1 \\ [\ (M/n)\ +\ 1]Y & for\ \ n > 1 \end{cases}$$

To get the overall latency to get both imem, fmem, cmem, omem just multiply by 4.

$$L_{ntot} = \begin{cases} 4MY & for\ n = 1 \\ 4[\ (M/n)\ +\ M]Y & for\ \ n > 1 \end{cases}$$

In the case of a square matric MY=M^2

$$L_{ntot} = \begin{cases} 4M^2 & for\ n = 1 \\ 4[\ (M^2/n)\ +\ 1]Y & for\ \ n > 1 \end{cases}$$

3.5 Comparison

We evaluate the relationship between the latency of the first algorithm and this second algorithm, if it is greater than 1 then this just exposed is faster.

$$\chi = L_n^I / L_n^{II}$$

Where the quotes I and II denote which of the two algorithms is being referenced. Since the first holds for n > 4 we evaluate this relationship in which

$$L_n^{II} = 4[\,(M/n) + M]$$

$$\chi = \frac{8M^2/n}{4\,[\,(M^2/n) + M\,]} = \frac{8M^2}{4\,(M^2 + nM)} = \frac{4M}{M+n}$$

Since both are valid for n <M is χ> 1. So, this second way of working is faster than the first method shown, in particular if $M >> n$ then $\chi \cong 2$ which means to say that in the same time interval I can perform almost twice as many operations.

3.6 Effectiveness Parameters

Also, in this case I can define an index $\varepsilon_{n'n''} = \frac{L_{n'}}{L_{n''}}$ that for $n' = 1$ and $n'' = n$ $\varepsilon_{1,n} = \varepsilon_n = L_1/L_n$

$$L_1/L_n = \frac{4M^2}{4\,[\,(M^2/n) + M\,]} = \frac{nM^2}{M\,(M+n)} = \frac{nM}{M+n}$$

$$\varepsilon_n = \frac{nM}{M+n}$$

moves away from $\psi = n'' / n' =$ n, while in the first the condition was always verified. All this translates into a better efficiency for a number n of low DSP. All this is valid only in the condition we mentioned above p = M and n <M.

I have also analyzed if it is possible to use a number n of DSP that is not a submultiple of M. This is possible by following a slightly different procedure, but which leads to identical results in terms of ε_n.

3.7 Extension of 2nd algorithm

Suppose we have a number of DSP = n <M such that p = mcm (n, M) > M, this is equivalent to saying that n is not a submultiple of M, it is possible to find a way to perform the calculations with the same speed.

- Several products are calculated during the first clock stroke t1, t2+...+ tn, equal to the number of DSP available n.
- At the second clock stroke we calculate tp, or the partial sum of the first calculated products $tp = t1 + t2 + ... + tn$
- At the third clock stroke the first coefficient is calculated already the first one

imem result (1) = $tp + t_{n+1} + \cdots t_M$

And at the same time, you begin to calculate the products of the next row. I introduce parameters to simplify the discussion.

Said T the unit of time in terms of clock strokes i = T-2, qi is the number of DSP usable to calculate products at clock stroke T = i + 2, while ζ_i is the number of DSP used to perform sums or sums more products. These parameters are calculated starting from T = 3 or from i = 1.

$$\zeta_i = \begin{cases} \boldsymbol{M - n} & \boldsymbol{T = 3} \\ \boldsymbol{q_{T-3}} & \boldsymbol{for\ even\ T} \\ \boldsymbol{M - q_{T-3}} & \boldsymbol{for\ odd\ T} \end{cases}$$

$$q_i = \begin{cases} n - \zeta_i & for\ even\ T \\ an\ -\ bM & for\ odd\ T \end{cases}$$

With b=a-1 and a = (3+ i)/2 (T + 1)/2

With b=a-1 and a = (3+ i)/2 (T + 1)/2

The algorithm stops when qi = 0 or equivalently when $\zeta i = n$ and iteratively repeats for N / p times, remembering that N = MY ep = mcm (n, M). This procedure works only if said $\beta n = n / (M - n)$ is an integer and mcm (p, N) = N.

The number of elements of the resulting vector calculated is equal to βn and the time taken, or the latency is equal to T = i + 2 or similarly

$$L_{p=}\ (\ 2\beta_n + 1)$$

The latency to calculate a matrix product per vector is equal to

$$L= (\ 2\beta_n + 1) * N/p$$

In the case where n satisfies the condition on βn then p = M * βn

T=1	T1, t2, t3... t8	
T=2	Tp+t1+t2+t3+....t8	
T=3	Imem(1) = tp + t9+t10+t11+t12 T1, t2, t3, t4	ζ_i= 4 q_1 = 4
T=4	Tp=t1+t2+t3+t4 T5, t6, t7, t8	ζ_i= 4 q_1 = 4
T=5	Imem (2) = tp + t5+t6+t8+...+t12	ζ_i= 8 q_3= 0

Example for n = 8 and M = 12 Y = 48

In the case chosen for the example β_8 = 8/4 = 2 for which use Lp = 2 (2 + 1) = 5 clock shots to get the first 2 results.

L= (2β_n + 1) * (Y/β_n)

So the latency for the calculation of imem is equal to $L = 5 * (48/2) = 5 * 24 = 120$ and the total latency will be L_{tot} = 480, time necessary to wait for having imem, fmem, cmem, omem.Calculation also in this case ε_n in the case of a square matric with M rows and M columns

$$\varepsilon n = L1/Ln = (M^2)/[(2\beta n+1) * (Y/\beta n)] = M\beta n/(2\beta n+1)$$

$$= (nM/M-n)/[(2n/M-n) +1]$$

$$= nM/(2n+M-n) = nM/(M+n)$$

Which brings us back to the same identical result as before even having changed the process formula for the calculation of imem, fmem, cmem, omem. Figure 18a shows the efficiency trend, in 18b the efficiency.

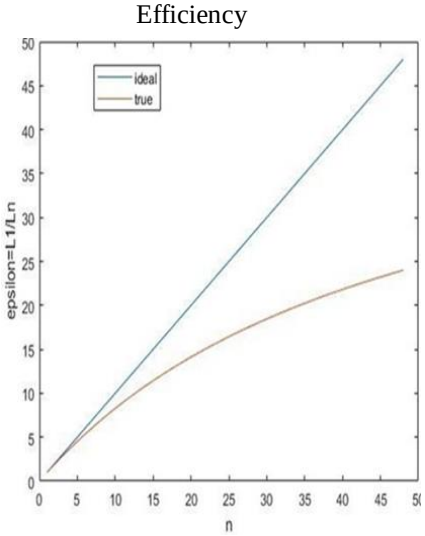


Fig 18. a: Efficiency

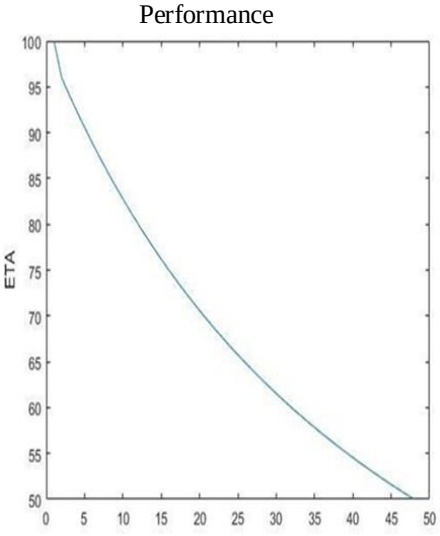


Fig 18. b: Performance

$$\eta_n=[100nM/(M+n)] * n=100M/(M+n)$$

From which we can see that as the number of DSP increases, the yield decreases more and more until it reaches a minimum of 50% when n = M.

It can be seen in the same way ϑ_n, noting that effectiveness, that is the line of effectiveness grows with the growth of n and does not remain constant

$$\vartheta n=(2\beta n+1) * \frac{M}{\beta n} * n= (\frac{2n}{M-n} +1) * [\frac{M}{n} * (M-n)] * n$$
$$= (M+n) * M=M^2+ nM$$

$nM \ll M2 \leftrightarrow nM \ll M2$ *cio*is $M \gg n$, that is the ideal case could be had only approximately in the case in which several DSP is used very much smaller than the dimensions of the Matrix.

Therefore, downstream of this research the most sensible solution, in general, is to not adopt more than one DSP per line, as we have seen that the increase speeds up the calculation but in an increasingly expensive way, paying in terms of performance.

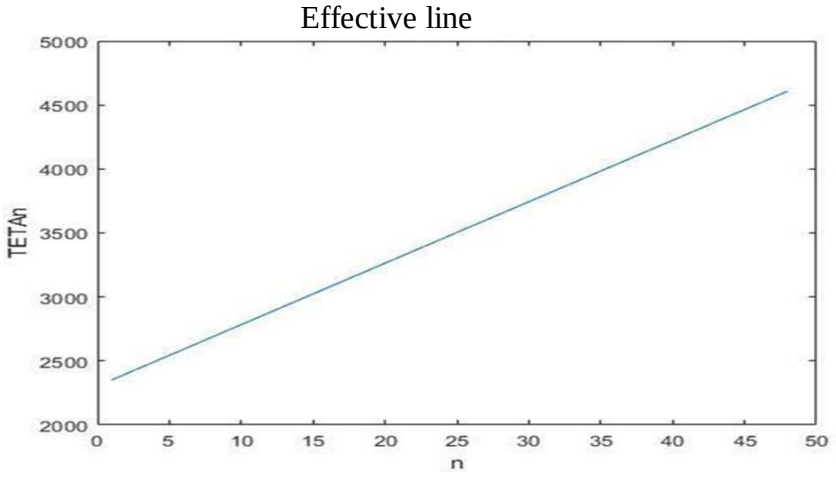


Figure 19: Effective line

Suppose instead we want to use several DSP n greater than M and see how to proceed.

$$k=\lfloor(i+1)/2\rfloor$$

$$i=T+2$$

$$f=\lfloor T/2\rfloor$$

$$\omega=n/M$$

$$\lambda=\lfloor\omega\rfloor$$

$$v=n-\lambda M$$

$$sd,e=W(d,e)*x(d,e)$$

Where W is the generic matrix of dimensions Y × M and the product $W \times x = u$

$$\boldsymbol{\rho_f} = \begin{cases} \mathbf{2} & \boldsymbol{for\ f = 0} \\ \boldsymbol{\rho_{f-1}} + \boldsymbol{\lambda} + \mathbf{1} & \boldsymbol{for\ f > 0} \end{cases}$$

3.7.1 Algorithm details

I- In the first clock stroke calculation $s_{1,1}$, $s_{1,2}$, ..., $s_{1,v}$

II- In the second clock stroke calculation

$sp = s_{1,1} + s_{1,2} + ... + s_{1,v}$ also calculation

sp_{f-1},1, sp_{f-1},2, ..., sp_{f-1},M

…. …. …

sp_f,1, sp_f,2, ..., sp_f,M

III- At the third clock stroke I start to get the first results $u_1, u_{pf-1}, \dots u_{pf}$

And I begin to calculate $s_{pk} + 1,1, s_{pk} +1,2, ..., s_{pk} +1, q_i$

IV- calculation $s_{pk} = s_{pk} + 1,1, s_{pk} +1,(q_{i-1} + q_i)$

$sp_{f-1},1, sp_{f-1},2,..., sp_{f-1},M$

…. ….. … …. … …..

$sp_f,1, sp_f,2,..., sp_f,M$

V- calculation u_{pf-1}-1 ,u_{pf-1} ,… u_{pf}

$s_{pk} + 1,1, s_{pk} +1,2,..., s_{pk} +1, q_i$

The procedure continues until $q_i = 0$, remembering that

$$q_i = \begin{cases} n - \zeta_i , & for\ even\ T \\ an - bM, & for\ odd\ T \end{cases}$$

$$\boldsymbol{\zeta}_i = \begin{cases} M - n & T = 3 \\ q_{T-3} & for\ even\ T \\ q_{T-4} & for\ odd\ T \end{cases}$$

$c1 = mcm\ (v, M)/M$

$c2 = c1\lambda$

$c = c1 + c2$

The time to calculate c elements is $L_p = 2c1 + 1$, to calculate the integer

T=1	$s_{1,1}$, $s_{1,2}$,. . .. $s_{1,8}$
T=2	**Sp** = $s_{1,1}$+ $s_{1,2}$ + … $s_{1,8}$ $s_{2,1}$, $s_{2,2}$,. . .. $s_{2,8}$ $s_{3,1}$, $s_{3,2}$,. . .. $s_{3,8}$
T=3 **i=1**	u_1 = s_p + $s_{1,9}$ + … $s_{1,12}$ u_2 = $s_{2,1}$ + $s_{2,9}$ + … $s_{2,12}$ u_3 = $s_{3,1}$ + $s_{3,9}$ + … $s_{3,12}$ ζ_1 = 4 , q_1 = 4 $s_{4,1}$ + $s_{4,2}$ + … $s_{4,4}$
T= 4 **i =2**	**Sp** = $s_{4,1}$+ $s_{4,2}$ + … $s_{4,4}$ $s_{5,1}$ + $s_{5,2}$ + … $s_{5,12}$ $s_{6,1}$ + $s_{6,2}$ + … $s_{4,12}$ ζ_2 = 4 , q_2 = 4 $s_{4,5}$ + $s_{4,6}$ + … $s_{4,8}$
T=5 **i=3**	u_4 = s_p + $s_{4,5}$ + … $s_{4,12}$ u_5 = $s_{5,1}$ + $s_{5,2}$ + … $s_{5,12}$ u_6 = $s_{6,1}$ + $s_{6,2}$ + … $s_{6,12}$ ζ_3 = 8 , q_3 = 0

matric instead $L = Lp * (Y / c) = (2c1 + 1) * (Y / c)$
Example M = 12, Y = 48, n = 32 $\lambda = 2$, $\upsilon = 8$

The algorithm is applicable when mcm (nbY, c) = Y and moreover as in the algorithm of before changing n = ν the ratio $\beta_\nu = \nu / (M - \nu)$ is an integer.

The graph below shows the trend of the latency as the number of DSP increases for a matric with dimensions M = 12 Y = 48. Figure 20 shows the trend of latency according to the number of DSP used.

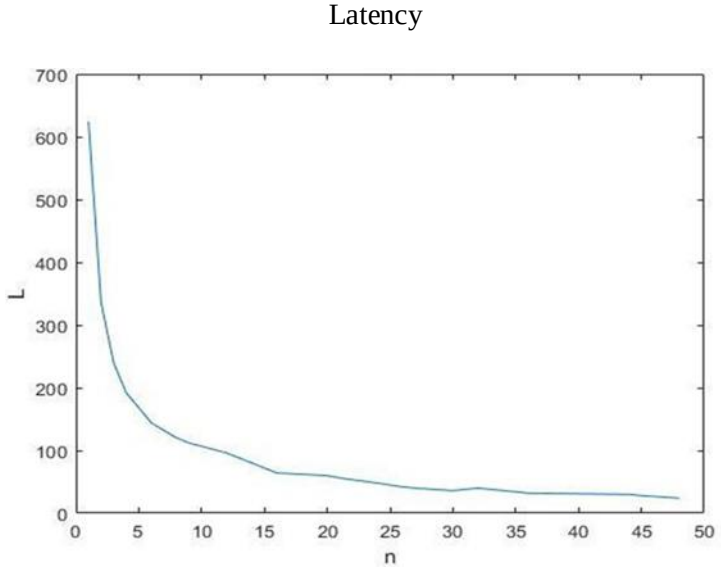


Figure 20: latency as a function of the number of DSPs

Once the question on how to carry out the most complicated calculations present in the equations to derive i_t, f_t, c_t, o_t has been unraveled, it is necessary to understand how to organize the architecture aimed at performing these calculations.

3.8 Architecture

To organize the sequence of the calculations I started from a simpler architecture to then refine it in order to improve the latency of the overall circuit.

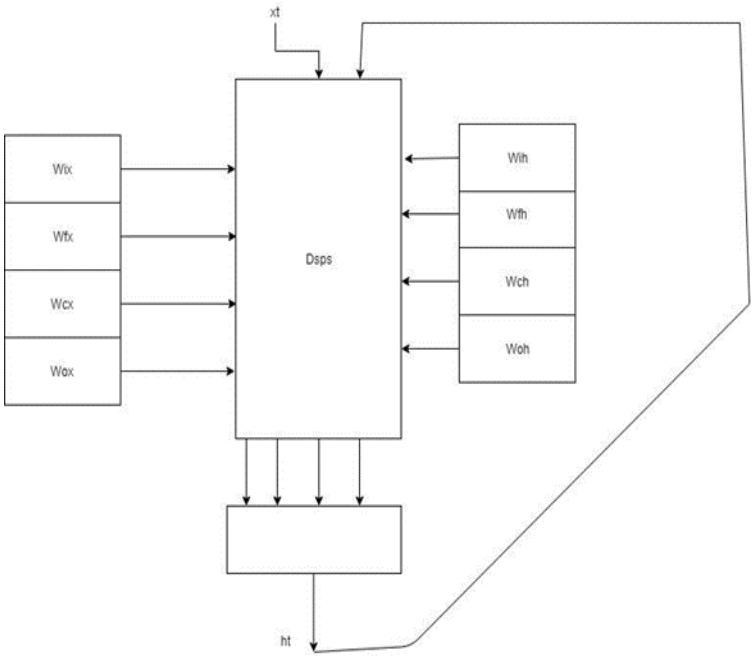


Figure 21: architecture of DSP

In this first model of architecture there is a single large block within which all operations are carried out. A more sophisticated model with respect to this consists in trying to parallelize the calculations within the single equations, in fact breaking the single equation into 2 Calculation Blocks.

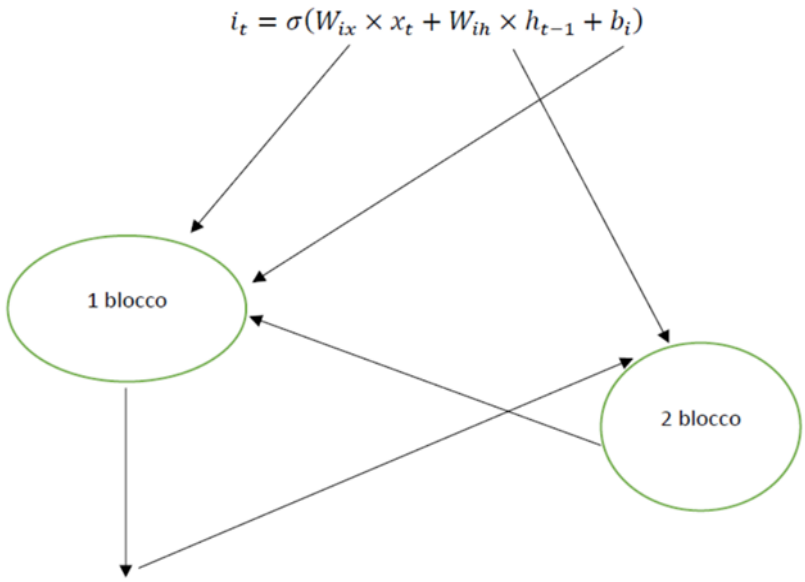


Figure 22: graph of 2 blocks

The first block receives as input W_{ix}, W_{fx}, W_{cx}, W_{ox}, b_i, b_f, b_c, b_o, imem, fmem, cmem, omem and calculates at the output h which is sent to the second block. The second block concurrently with the first one receives $h_t - 1$ and W_{ih}, W_{fh}, W_{ch}, W_{oh} which it uses to calculate imem, fmem, cmem, omem according to the modalities we have seen before.

The first block must carry out in addition to the vector matrix product also the addition of this last result with imem, fmem, cmem, omem. One might think that this introduces a significant slowdown but in reality, it is not so since in all the cases except, in the case

which I use 1 DSP this is in no way to invalidate the latency of the block. To see how the first block will have to schedule the calculations, let us take as an example only the

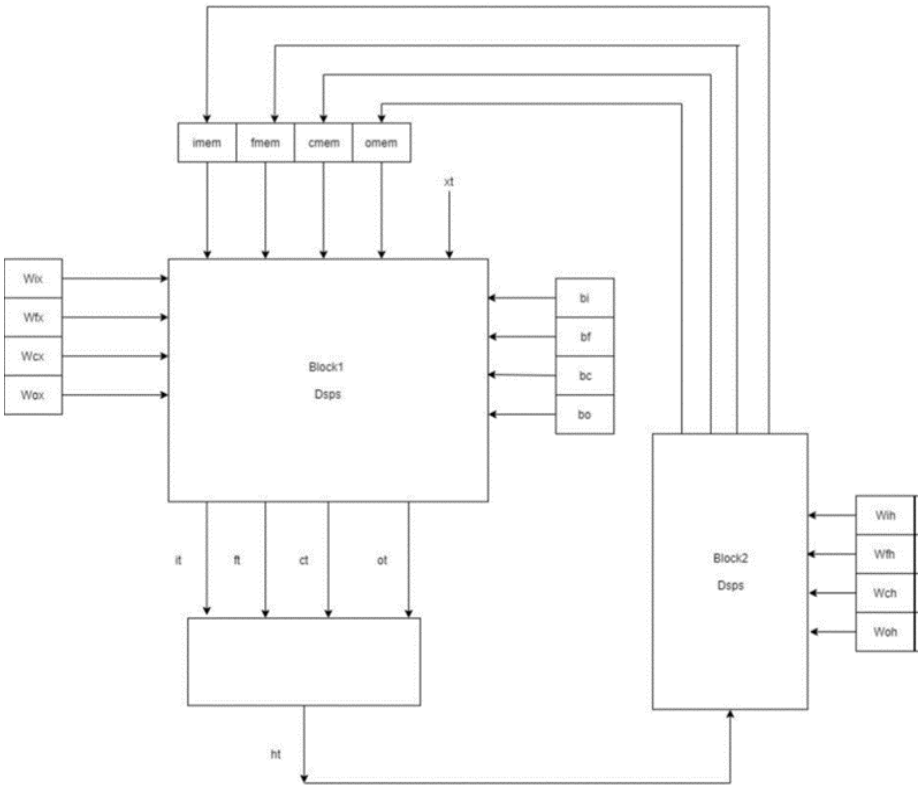


Figure 23: block diagram of 2 DSPs

This is one of the 4 calculations that the first block must perform, and we want to show that except in the case of 1 DSP, where the latency increases by Y clock shots, in all other cases the latency remains the same. Consider the matrix W_{ix} of dimensions Y (lines), M (columns) and n = 1 DSP, always using the same notation we schematize the algorithm as before.

T=1	T=2	T..	T=M	T=M+1
i_m $= i_{11} + b_i\ (1)$	i_m $= i_{12} + i_m$	....	i_m $= i_{1M} + i_m$	i_m $=i_m+imem(1)$

Then I use M clock shots instead of using M + 1. Having a matrix of Y rows, this latency must be multiplied by Y so that $L = MY$ + while in the case in which the only calculation that must perform is $W_{ix} \times x_t$ the latency is less and is equal to $L = MY$. Then using 1 only DSP the latency increases with increasing Y. If instead I use 2 DSPs

T=1	$t1 = i_{13} + t1$ $t2 = i_{14} + t2$
T= 2	$t1 = i_{11} + t1$ $t2 = i_{12}$
T.....	.....
T=M/2	$t1 = i_{1(M-1)} + t1$ $t2 = i_{1M} + t2$
T=M/2	$u_{1=}\ t1 + t2 + imem\ (1)$

So, by using 2 DSPs the latency for a single element becomes $L = M / 2 + 1$ which is the same that you have to perform the simplest calculation where neither imem nor the bias appear. So, for n> 1 the latency is exactly the same in both branches with the same size of the matrices.

From the analysis carried out it emerges that in general to make a product between a matric and a vector it is better to use only one DSP since with increasing n efficiency is always lower.

If I use 1, in making the two accounts, on the one hand I always go slower than the other of Y shots of clock. The choice I can think of to make to make the times as similar as possible is to use a number n of DSP where each of the single DSP executes in parallel the algorithm on a single line, in this way I get

$LB_1 = (MY + Y / n)$ (Latency relative to block 1)

$LB_2 = MY / n$ (Latency relative to block 2)

Since the latencies must be integers, in both cases we must choose n so that mcm (n, Y) = Y

I evaluate which is the best choice in terms of n so that the two latencies can be as close as possible, with n number of DSP to place individually on each line and not as previously seen for the calculation of the same line.

I introduce a new parameter that evaluates the relationship between the latency of the first block decreased by 1 and the latency of the second block

$$\tau = (LB_1 - 1) / LB_2 = (M + (Y/n) - 1) / (N/n)$$

$$= (Mn + Y - n) / N$$

$$= [n(M - 1) + Y] / N$$

Clearly in this circumstance n_{max} = Y therefore we evaluate

with the variation of n.τ is a linearly increasing function with n and assumes its maximum for n = Y in which it is 1.

When n = Y means that $LB_1 = 1 + LB_2$ that is when I use the maximum of DSP, putting 1 for the first block and the second block have a difference in terms of latency of a single clock stroke, even if the first block performs many more operations.

So, we derive that a convenient choice is to use as many DSP as possible. Figure 24 shows the trend of the ratio with the variation of n with M = 50 and Y = 50.

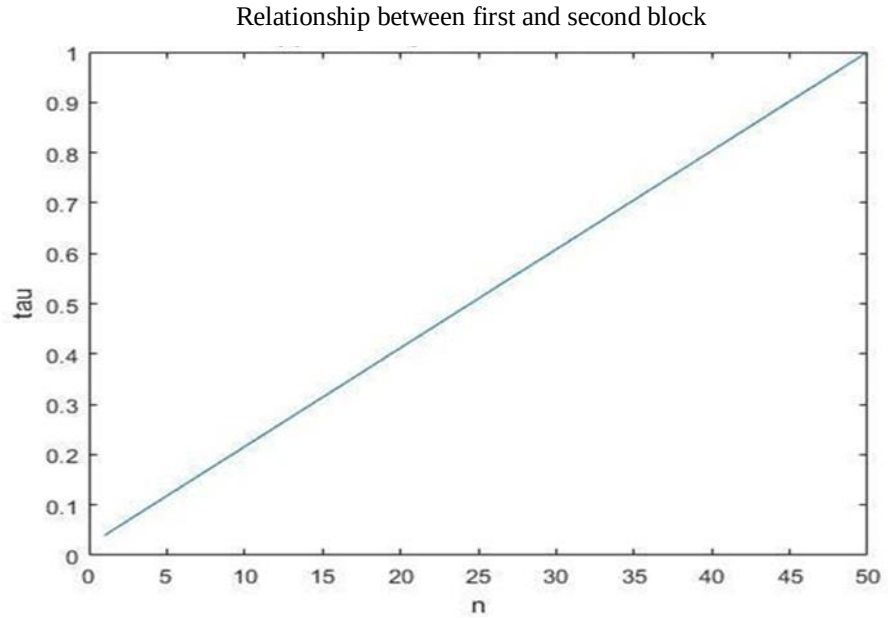


Figure 24: trend of τ as a function of n

After the above algorithms, the second one was used since it was the fastest to perform the calculations and, I used 1 DSP for each single element calculated.

It remains to establish how many DSP to use in order to obtain a low latency and try to equate the latencies between the first and second calculation block.

The 1 block performs the following operations

$$1)\ i_t=\sigma(w_{ix}\times x_t+imem+b_i)$$

$$2)\ f_t=\sigma(w_{fx}\times x_t+fmem+b_f)$$

$$3)\ c_t=f_t*c_{t-1}+i_t*g(w_{cx}\times x_t+cmem+b_c)$$

$$4)\ o_t=\sigma(w_{ox}\times x_t+omem+b_o)$$

$$5)\ h_t=o_t*H\ (c_t)$$

The matrices w_{ix}, w_{fx}, w_{cx}, w_{ox} have dimensions equal to $Y \times M$, while w_{ix}, w_{fh}, w_{ch}, w_{oh} have dimensions $Y \times Y$. The size M is fixed by the number of features and therefore by the data set being used to train the neural network, while Y represents the number of hidden states used to train the network and it can be varied to have a more or less high accuracy.

Beyond a certain limit, increasing Y only increases the calculations to be made but does not return a higher accuracy that depends on how large the data set used is. Of the five equations given above, the most critical, that is, the one that entails a greater computational burden is certainly 3. To make all the necessary calculations to get the 3) you must wait until you've already got the 1) and the 2) reason why I decided to rewrite it in a different way that I report here.

$$c_t=f_t*c_{t-1}+i_t*c_c$$

where 6)

$$c_c=g(w_{cx}\times x_t+cmem+b_c)$$

At this point the 1), 2), 4), 6) require exactly the same latency to produce the results and in particular if we think of only one element that is i_t (1), f_t (1), c_c (1), o_t (1) latency, using a DSP for a single equation, will be equal to M + 1 as previously calculated.

T=1	$i_t(1) = W_{ix}(1,1) * x_t(1) + b_i(1)$ $f_t(1) = W_{fx}(1,1) * x_t(1) + b_f(1)$ $c_c(1) = W_{cx}(1,1) * x_t(1) + b_c(1)$ $o_t(1) = W_{tx}(1,1) * x_t(1) + b_o(1)$
T=2	$i_t(1) = W_{ix}(1,2) * x_t(2) + i_t(1)$ $f_t(1) = W_{fx}(1,2) * x_t(2) + f_t(1)$ $c_c(1) = W_{cx}(1,2) * x_t(2) + c_c(1)$ $o_t(1) = W_{tx}(1,2) * x_t(2) + o_t(1)$
T.....	.....
T=M	$i_t(1) = W_{ix}(1,M) * x_t(M) + i_t(1)$ $f_t(1) = W_{fx}(1,M) * x_t(M) + f_t(1)$ $c_c(1) = W_{cx}(1,M) * x_t(M) + c_c(1)$ $o_t(1) = W_{tx}(1,M) * x_t(M) + o_t(1)$
T=M+1	$i_t(1) = imem(1) + i_t(1)$ $f_t(1) = fmem(1) + f_t(1)$ $c_c(1) = cmem(1) + c_c(1)$ $o_t(1) = omem(1) + o_t(1)$

So, to calculate the first element of each of the four equations M + 1 clock shots are needed. To get the true values you need to apply the activation functions to them, which by design choice I decided to pre-calculate in software for every possible value that can be verified and stored in ROM memories so as not to have to implement a circuit that runs in HW this operation.

T= M +1	$\mathbf{i_t(1)} = \sigma(imem(1) + i_t(1))$ $\mathbf{f_t(1)} = \sigma(fmem(1) + f_t(1))$ $\mathbf{c_c(1)} = \mathbf{g}(cmem(1) + c_c(1))$ $\mathbf{o_t(1)} = \sigma(omem(1) + o_t(1))$

So, at this point to get c_t (1) and h_t (1)

T= M+2	$\mathbf{c_t(1)} = \mathbf{f_t(1)} * \mathbf{c_{t-1}(1)} + \mathbf{i_t(1)} * \mathbf{c_c(1)}$
T=M+3	$\mathbf{h_t(1)} = \mathbf{o_t(1)} * \mathbf{H(c_t(1))}$

So overall to get the first element of ht using 4 DSPs I must wait for a latency of M + 3 clock strokes. To get all the elements the latency will be equal to $L = (M + 3) * Y$

What I now want to investigate is the latency to get all the vector ht with the number of DSP n, since it represents the latency of the whole first block.

Since I have parallelized on all 4 matrices n must always be a multiple of 4, that is mcm (n, 4) = n. If I use 8 DSPs after M + 3 clock strokes I will get 2 of the M elements of h for which I will have halved the latency. If I use 12 DSPs, I reduce the latency by a factor of 3 and so on.

However, this method of proceeding also imposes an additional condition.

The number of elements calculated every M + 3 clock stroke must be a submultiple of Y. I can define the number of elements calculated

every M + 3 clock strokes like $\partial = n / 4$ where n is the number of DSP used. The conditions to which n must therefore respect are

$$mcm(\partial, Y) = Y$$

$$mcm(n, 4) = n$$

So established Y, I can't use several DSP at will. The latency of the first block, chosen Y, will be equal to $L_{B1} = (M + 3) * Y / \partial = 4 (M + 3) Y / n$.

It can be observed that for uniform distribution the length of carry chain is always sensibly smaller than adder size. When 50% of inputs are taken from Gaussian distribution with σ=256 (Fig. 27(b)), a bimodal distribution is observed with an appreciable portion of carry chains is as long as the adder size; by increasing σ the second peak of the distribution moves to the left (Fig. 27(c)).

3.9 Application: voice recognition

The hardware that has been implemented is custom built for a specific application. The dataset used to train the network is the 'Japanese vowels' present inside Matlab, in which 300 times sequences are provided, each of which contains more vectors of 12 elements.

Each element of these vectors represents a feature of the specific application. To train the network, a specific Matlab toolbox was used, configured in such a way as to be able to classify 9 different items.

To do the training there is also the need to define how many layers of the LSTM to use, the higher the number of output size the higher the accuracy will be, within a certain limit dictated by the data set that is available. The number of output sizes of the LSTM

Also corresponds to what until now we have called Y or the number of rows of the matrices. In this regard, Y will be chosen ad hoc following a precise mathematical reasoning.

T=1	**imem(1) = $W_{ih}(1,1) * h_{t-1}(1)$** **fmem(1) = $W_{fh}(1,1) * h_{t-1}(1)$** **cmem(1) = $W_{ch}(1,1) * h_{t-1}(1)$** **omem(1) : $W_{oh}(1,1) * h_{t-1}(1)$**
T=2	**imem(1) = $W_{ih}(1,2) * h_{t-1}(2)$ + imem(1)** **fmem(1) = $W_{fh}(1,2) * h_{t-1}(2)$ + fmem(1)** **cmem(1) = $W_{ch}(1,2) * h_{t-1}(2)$ + cmem(1)** **omem(1) : $W_{oh}(1,2) * h_{t-1}(2)$ + omem(1)**
T=M	**imem(1) = $W_{ih}(1,Y) * h_{t-1}(Y)$ + imem(1)** **fmem(1) = $W_{fh}(1,Y) * h_{t-1}(Y)$ + fmem(1)** **cmem(1) = $W_{ch}(1,Y) * h_{t-1}(Y)$ + cmem(1)** **omem(1) : $W_{oh}(1,Y) * h_{t-1}(Y)$ + omem(1)**

The number of features of the specific application represents instead what until now has been called M, or the number of columns of the matrices on which it is necessary to operate in the first block. It can be concluded that M = 12 and Y is to be established by trying to choose neither too low nor too high.

Since in the case of the considered application the number of features is equal to 12 then M = 12 which means that $L_{B1} = 60 * Y / n$. This is what regards the latency of the first block, for the second block we can make similar considerations to estimate the latency as a function

of Y and n. To calculate the first value of imem, fmem, cmem, omem the procedure I follow is always the same.

For the second block therefore the latency using 4 DSP is equal to M and more generally it will be $L_{B2} = Y^2 / \partial = 4Y^2 / n$. It is not said that I should use the same number n of DSP for both blocks, so I distinguish in n1 and n2 where n1 is the number of DSP used for the first block and n2 for the second.

$$L_{B2} = Y^2 / \partial = 4Y^2 / n2$$

$$L_{B1} = 60 * Y / n1$$

I aim to find the values of n1, n2, Y for which the latency of the first and second blocks are the same.

$$L_{B1} = L_{B2}$$

$$60 * Y / n1 = 4 * Y^2 / n2$$

$$n1 = (15/Y) * n2$$

From this equality, having to be n1 and n2 integers we understand that also Y cannot be any and among other things also Y must be an integer. It turns out that mcm (15, Y) = Y or Y must be a multiple of 15.

$Y \in \{15,30,45,60,75,90,105,120, \cdots\}$
When Y varies, the possible values n1 and n2 may also vary.

$n1$: $mcm(n1,4) = n1$ and $mcm\ (n1/4, Y) = Y$
$n2$: $mcm(n2,4) = n2$ and $mcm\ (n12/4, Y) = Y$

$n1$: $4 * Y / n1$ must be an integer
$n2$: $4 * Y^2 / n2$ must be an integer

These conditions are the result of the fact that latency must always be a whole natural number. I analyze if there are possible solutions for Y = 60.

Y=60 $n1$, $n2 \in$ {4,8,12,16,20,24,40,48,60,80,120,240}

n2=4*n1

n1=4, n2=16

now I have to check the third and fourth conditions respectively located on n1 and n2

$4*Y/n1=4*60/4=60$

$4*Y^2/n2=4*60^2/16=14400/16=900$ ok

$n1$=8, $n2$=32 it does not belong to the set of possible $n2$

$n1$=12, $n2$=48 ok 240/12=20 ok 14400/48=300 ok

$n1$=16, $n2$=64 it does not belong to the set of possible $n2$

$n1$=20, $n2$=80 ok 240/20=12 ok 14400/80=180 ok

$n1$=24, $n2$=96 it does not belong to the set of possible $n2$

$n1$=40, $n2$=160 it does not belong to the set of possible $n2$

$n1$=60, $n2$=240 ok 240/60=4 14400/240=60

Therefore, using $n1$ = 60 ed $n2$ = 240 dsp a latency can be obtained for both blocks of 60 clock strokes. The choice I accepted instead is to

set Y = 48 and choose n1 = 48 and n2 = 192 which allows, by doing a good design, to obtain a latency of 60, therefore equal to that of the case n1 = 60 n2 = 240 with Y = 60 with the benefit of using 60 DSPs less.

3.10 Circuit implementation

An accelerator for neural networks has been designed using the Global Foundry 40nm CMOS technology. The accelerator implements the LSTM block of the recurring neural network. The numbers of the implemented circuit are recalled in the table 2:

Table 2

	Block 1	Block 2
Number of operations	2832	9216
Number of DSPs	48	192
Ideal latency	59	48
Obtained latency	60	48

The architecture of the Block1 and 2 is shown in fig. 25 and 26 respectively.

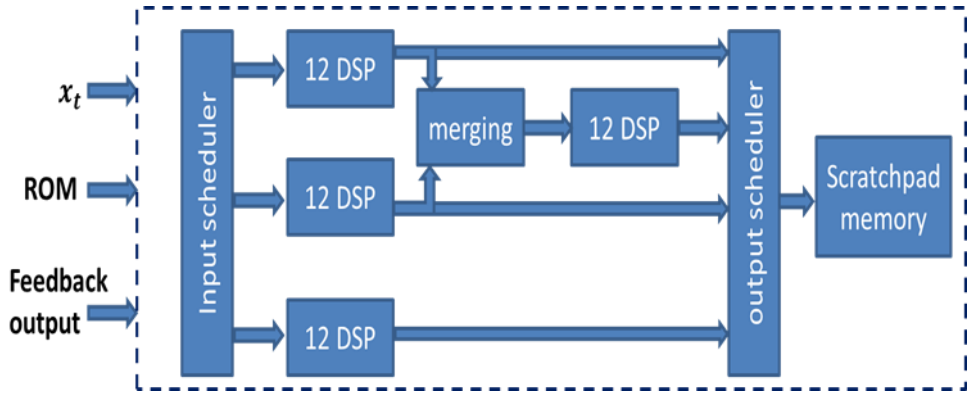


Fig.25: Block 1 architecture

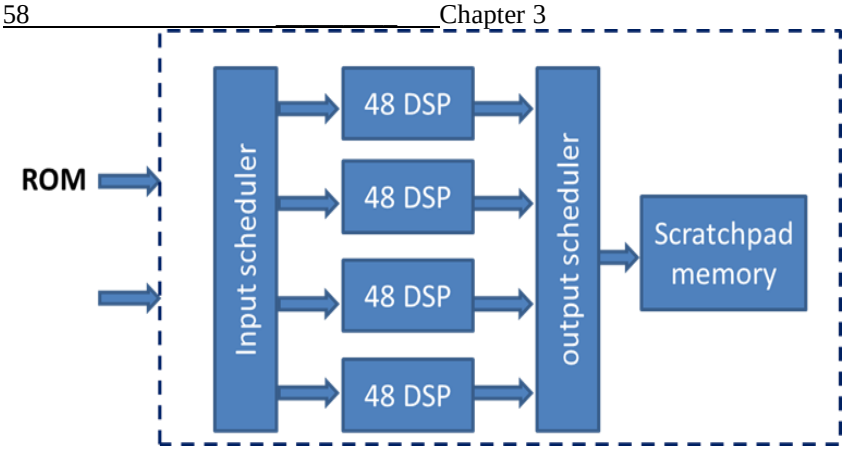


Fig.26: Block 2 architecture

As can be seen, scratchpad memories and schedulers are used to implement the scheduling discussed in previous sections. The table 3 reports a comparison with previous art.

Table 3

	Clock frequency **(MHz)**	**# DSP**	**Registers**	**ROM**	Latency
Implemented circuit	464	240	5904	1.8Mb	60
[13]	115	11	30604	387Kb	58K
[14]	115	11	1157	387Kb	30K
[30]	200	96	N.A.	86Kb	40M
[31]	200	1504	453K	17Mb	16K
[32]	142	50	13K	288Kb	127K

As can be seen the main feature of the developed circuit is the very low latency obtained. The number of DSPs used is larger than the one used in the implementation presented in the previous section (rows 2 and 3 of the table), so this implementation cannot be considered a reduced footprint design.

However, with respect to the accelerator presented in [31], the proposed circuit still exhibit a low number of DSPs while achieving a better latency.

Chapter 4

Conclusion

Within this thesis, different types of approaches have been developed to speed up the calculation of the LSTM Layer of the RNN. In particular, 2 algorithms have been developed for the scheduling of operations, which allow easy access to the vectors to be taken in memory.

These two it has been shown that one in particular allows to obtain an almost ideal case, which we can define as sub-optimal. It allows to first split the set of equations of the LSTM into two sub-sets in which the dependence between the data is reduced to the minimum, so as to be able to parallelize the calculations of these two blocks.

Afterwards, within each of these two blocks, it is possible to optimize the calculation work by implementing a parallelism between functions, rows and columns with the use of DSPs that allow their potential to be exploited almost 100%.

The analysis conducted in conclusion provides a valid method to be able to design an RNN based on LSTM for any type of application. In particular, following this type of approach it is possible to consistently decrease the latency of these types of circuits, allowing to obtain results that are close to the ideal case.

References

[1] C. Marzban, and R. Viswanathan, “STOCHASTIC NEURAL NETWORKS WITH THE WEIGHTED HEBB RULE,” Physics Letters A, vol. 191, no. 1-2, pp. 127-133, Aug, 1994.

[2] K. Guenther, “Rebel Genius: Warren S. McCulloch's Transdisciplinary Life in Science,” Bulletin of the History of Medicine, vol. 92, no. 1, pp. 223-224, Spr, 2018.

[3] W. S. McCulloch, and W. Pitts, “A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY (REPRINTED FROM BULLETIN OF MATHEMATICAL BIOPHYSICS, VOL 5, PG 115-133, 1943),” Bulletin of Mathematical Biology, vol. 52, no. 1-2, pp. 99-115, 1990.

[4] J. Schmidhuber, “Deep learning in neural networks: An overview,” Neural Networks, vol. 61, pp. 85-117, Jan, 2015.

[5] A. B. Bulsari, and H. Saxen, “A RECURRENT NEURAL NETWORK MODEL,” Artificial Neural Networks, 2, Vols 1 and 2, pp. 1091-1094, 1992.

[6] B. Bakker, "Reinforcement learning with long short-term memory," Advances in Neural Information Processing Systems 14, Vols 1 and 2, Advances in Neural Information Processing Systems T. G. Dietterich, S. Becker and Z. Ghahramani, eds., pp. 1475-1482, 2002.

[7] M. Liang, X. L. Hu, and Ieee, "Recurrent Convolutional Neural Network for Object Recognition," IEEE Conference on Computer Vision and Pattern Recognition. pp. 3367-3375, 2015.

[8] G. Massini, “Hopfield Neural Network,” Substance Use & Misuse, vol. 33, no. 2, pp. 481-488, 1998.

[9] D. T. Pham, and D. Karaboga, “Training Elman and Jordan networks for system identification using genetic algorithms,” Artificial Intelligence in Engineering, vol. 13, no. 2, pp. 107-117, Apr, 1999.

[10] S. P. Chatzis, and Y. Demiris, “Echo State Gaussian Process,” Ieee Transactions on Neural Networks, vol. 22, no. 9, pp. 1435-1445, Sep, 2011.
[11] A. Goudarzi, A. Shabani, D. Stefanovic, and Ieee, “Exploring Transfer Function Nonlinearity in Echo State Networks,” 2015 Ieee Symposium on Computational Intelligence for Security and Defense Applications (Cisda), pp. 119-126, 2015.
[12] S. Hochreiter, and J. Schmidhuber, “Long short-term memory,” Neural Computation, vol. 9, no. 8, pp. 1735-1780, Nov, 1997.
[13] V. Abdolzadeh, N. Petra. "On the Trade-Offs in Efficient Implementation of Neural Network Accelerators," ApplePies annual conference 2018-September 26-27, Pisa.Italy.
[14] V. Abdolzadeh and N. Petra, "Efficient Hardware Accelerators for Recurrent Neural Network SoC Design," in proc. Società Italiana di Elettronica conference (SIE),Napoli, IT, 20-22 Jun. 2018. ISBN:978-88-905519-2-5.
[15] R. C. Jin, J. F. Jiang, and Y. Dou, “Accuracy Evaluation of Long Short-Term Memory Network Based Language Model with Fixed-Point Arithmetic,” Applied Reconfigurable Computing, vol. 10216, pp. 281-288, 2017.
[16] K. Zhang, W. L. Chao, F. Sha, and K. Grauman, “Video Summarization with Long Short-Term Memory,” Computer Vision - Eccv 2016, Pt Vii, vol. 9911, pp. 766-782, 2016.
[17] A. Zyner, S. Worrall, J. Ward, E. Nebot, and Ieee, "Long Short-Term Memory for Driver Intent Prediction," 2017 28th Ieee Intelligent Vehicles Symposium, IEEE Intelligent Vehicles Symposium, pp. 1484-1489, 2017. 65
[18] M. Mimura, S. Sakai, T. Kawahara, and A. Isca-Int Speech Commun, Speech Dereverberation Using Long Short-Term Memory, 2015.
[19] Y. W. Zhang, C. Wang, L. Gong, Y. T. Lu, F. Sun, C. C. Xu, X. Li, X. H. Zhou, and Ieee, "A Power-Efficient Accelerator Based on

FPGAs for LSTM Network," IEEE International Conference on Cluster Computing. pp. 629-630, 2017.
[20] JANUARY 2018, pp. 198–208.UCI Machine Learning Repository:Japanese,Vowels,Dataset.https://archive.ics.uci.edu/ml/datasets/Japanese+Vowels
[21] Hyn-d-man, R.J.: Time Series DataLibrary. https://datamarket.com/data/list/?q=cat:g24%20 provider:tsdl
[22] Zou, L., Gu, Y., Song, J., Liu, W., Yao, Y.: Long short-term memory based recurrent neural networks for collaborative filtering. In: IEEE UIC 2017 San Francisco, CA, USA, pp. 1–6. https://doi.org/10.1109/uic-atc.2017.8397539
[23] B. Y. Masram, P. T. Karule, and Ieee, "High Performance Analysis of a CORDIC Architectures based on FPGA: A comparative approach," 2014 International Conference on Advanced Communication Control and Computing Technologies (Icaccct), pp. 569-574, 2014.
[24] J. Sudha, M. C. Hanumantharaju, V. Venkateswarulu, and H. Jayalaxmi, "A Novel Method for Computing Exponential Function Using CORDIC Algorithm," International Conference on Communication Technology and System Design 2011, vol. 30, pp. 519-528, 2012.
[25] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, W. Sung, and Ieee, "FPGA-based Low-power Speech Recognition with Recurrent Neural Networks," 2016 Ieee International Workshop on Signal Processing Systems (Sips), pp. 230-235, 2016.
[26] Xilinx. "Digilent ZedBoard Zynq®-7000 ARM/FPGA SoC Development,Board,"https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf.
[27] Han et al.: ESE: efficient speech recognition engine with sparse LSTM on FPGA. In: Proceedings of the 2017 ACM/SIGDA FPGA 2017, Monterey, CA, USA, Feb. 2017, pp. 75–84
[28] Price et al.: A low-power speech recognizer and voice activity detector using deep neural networks. IEEE JSSC **53**(1) (2018)

[29] Moini et al.: A resource-limited hardware accelerator for convolutional neural networks in embedded vision applications. IEEE TCAS II: Express Briefs **64**(10) (2017)

[30] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li, Implementation and Optimization of the Accelerator Based on FPGA Hardware for LSTM Network: In: 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC) https:// DOI 10.1109/ISPA/IUCC.2017.00098

[31] S. Li, Qi. Wang, X. Liu, J. Chen,Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li, Low Cost LSTM Implementation based on Stochastic Computing for Channel State Information Prediction. In: 2018 IEEE Asia Pacific Conference on Circuits and Systems 978-1-5386-8240-1/18/$31.00 ©2018 IEEE

[32] R. C. Jin, J. F. Jiang, and Y. Dou, "Accuracy Evaluation of Long Short Term Memory Network Based Language Model with Fixed-Point Arithmetic," Applied Reconfigurable Computing, vol. 10216, pp. 281-288, 2017.

Appendix

Appendix A

C Code for the LSTM Layer

```
#include "ap_int.h"
#include "sistema.hpp"
void    sistema(ap_int<6>    x[12],    ap_int<6>
h[50],ap_uint<1>reset)
{
                static ap_int<18> imem[50];
                static ap_int<18> fmem[50];
                static ap_int<19> cmem[50];
                static ap_int<18> omem[50];
                static ap_int<6>    h_int[50];
```

```
                if(reset==1){
                      for (int i=0;i<50;i++){
                                imem[i]=0;
                                fmem[i]=0;
                                cmem[i]=0;
                                omem[i]=0;
                                h_int[i]=0;
        mem2y(x, h_int, h, imem, fmem, cmem,
omem,reset);
                          }
          }          else{
        mem2y(x, h_int, h, imem, fmem, cmem,
omem,reset);
        memCalc(h_int, imem, fmem, cmem, omem);
   }
}
```

Appendix B

C Code for the LSTM Ext function

```
#include "ap_int.h"
#include "cordic_hls.h"
#define N 50
#define M 12


//x_t sarà su 4 bit. 3 per la parte decimale ed una
per quella intera. (2^0;2^-3)

//void lstm(int5 x_t[N],int5 y_t[N]) {
void mem2y(ap_int<6> x_t[M],ap_int<6>
y_t[N],ap_int<6> y_t_ext[N], ap_int<18> imem[N],
ap_int<18> fmem[N], ap_int<19> cmem[N], ap_int<18>
omem[N],ap_uint<1> reset) {
```

```
//const ap_int<4> Wxi[] = {
const ap_int<5> Wxi[] = {
#include "Wxi.txt"
};

const ap_int<5> Whi[] = {
#include "Whi.txt"
};

const ap_int<5> Wxf[] = {
#include "Wxf.txt"
};

const ap_int<5> Whf[] = {
#include "Whf.txt"
};

const ap_int<7> Wxc[] = {
#include "Wxc.txt"
};

const ap_int<6> Whc[] = {
#include "Whc.txt"
};

//const int5 Wxo[] = {
const ap_int<5> Wxo[] = {
#include "Wxo.txt"
};

const ap_int<5> Who[] = {
#include "Who.txt"
};

const ap_int<8> b_i[] = {
#include "bi.txt"
```

```
};



const ap_int<10> b_f[] = {
#include "bf.txt"
};
const ap_int<9> b_o[] = {
#include "bo.txt"
};
const ap_int<9> b_c[] = {
#include "bc.txt"
};

ap_int<19> i_t_pre[N],o_t_pre[N];
//ap_int<15> app, app2, app3;
ap_int<19>f_t_pre[N];
ap_int<20>g_t_pre[N];
ap_int<6> i_t[N],f_t[N];

ap_int<6>g_t[N],o_t[N];
ap_int<8> in_wave;
ap_int<2> sigma;
ap_int<45>
wave_out_i,wave_out_i2,wave_out_f,wave_out_f2,
```

```
wave_out_g,wave_out_g2,wave_out_o,wave_out_o2,wave_
out_c2,wave_out_c;
ap_int<11> costante;
ap_int<22> variante,variante2;
ap_int<22> risultatov;
//solo c_t_1 deve essere static (controllare)
ap_int<13> c_t_pre[N];
//serve memoria su y_t_pre? (controllare)
// ap_int<14> y_t_pre[N];
ap_int<12> y_t_pre[N];
//static int5 c_t[N],c_t_1[N],y_t_1[N];
//solo c_t_1 deve essere static (controllare)
static ap_int<6> c_t_1[N];
ap_int<6> uno_long;
ap_int<6> c_t[N];
static ap_int<6> y_t_1[N];
ap_int<30>
risultatoi,risultatoo,risultatoi2,risultatoo2;
ap_int<30> risultatof,risultatof2;

ap_int<31>risultatog;
```

```
ap_int <31> risultatog2;

ap_int<13>c_t_pre2[N];

ap_int<45> rounding37;

ap_int<30> rounding11;

ap_int<24>rounding9;

ap_int<31> rounding11g;

ap_int<6> rounding_out=1;

rounding_out=(rounding_out<<2);

int j,k,w;

uno_long=1;

uno_long=uno_long<<3;

costante=652;

rounding37=1;

rounding37=rounding37<<37;

rounding11g=1;

rounding11g=rounding11g<<11;

rounding11=1;

rounding11=rounding11<<11;

rounding9=1;

rounding9=rounding9<<9;

ap_int<6> c_t_tanh[N];
```

```
if(reset==1){

              for(int T=0;T<N;T++){
              y_t_1[T]=0;
              c_t_1[T]=0;

              }

}
else{

i__t:
for(j=0;j<N;j++){

      i_t_pre[j]=0;
    f_t_pre[j]=0;
      o_t_pre[j]=0;
    g_t_pre[j]=0;
       lstm_label6:for(k=0;k<M;k++){
             //app3 = (ap_int<15>)x_t[k];
            //app2 = (ap_int<15>)Wxi[j*M+k];
           //app = app2*app3;
           i_t_pre[j]=i_t_pre[j]+Wxi[j*M+k]*x_t[k];
f_t_pre[j]=f_t_pre[j]+Wxf[j*M+k]*x_t[k];
           o_t_pre[j]=o_t_pre[j]+Wxo[j*M+k]*x_t[k];
           g_t_pre[j]=g_t_pre[j]+Wxc[j*M+k]*x_t[k];

}

//calcolo i_t
```

```
i_t_pre[j]=i_t_pre[j]+imem[j]+b_i[j];

if(i_t_pre[j]<-805){
i_t[j]=0;
}
else if(i_t_pre[j]>804){
i_t[j]=uno_long;
}




else{

          risultatoi=i_t_pre[j]*costante;
risultatoi=risultatoi+rounding11;
risultatoi2=risultatoi>>12;
sigma=1;
in_wave=risultatoi2;
cordic_hls(in_wave,sigma,&wave_out_i);
wave_out_i=wave_out_i+rounding37;
wave_out_i2=wave_out_i>>38;
i_t[j]=wave_out_i2;



}

//calcolo f_t
```

```
f_t_pre[j]=f_t_pre[j]+fmem[j]+b_f[j];
if(f_t_pre[j]<-805){
f_t[j]=0;
}
else if (f_t_pre[j]>804){

f_t[j]=uno_long;
}
else{


                risultatof=f_t_pre[j]*costante;
                risultatof=risultatof+rounding11;
                risultatof2=risultatof>>12;
            sigma=1;
               in_wave=risultatof2;

cordic_hls(in_wave,sigma,&wave_out_f);
               wave_out_f=wave_out_f+rounding37;
               wave_out_f2=wave_out_f>>38;

               f_t[j]=wave_out_f2;

 }


       //calcolo o_t
```

```
        o_t_pre[j]=o_t_pre[j]+omem[j]+b_o[j];
        if(o_t_pre[j]<-805){
         o_t[j]=0;
        }
        else if(o_t_pre[j]>804){
        o_t[j]=uno_long;
       }
       else{


risultatoo=o_t_pre[j]*costante;

risultatoo=risultatoo+rounding11;
                  risultatoo2=risultatoo>>12;
                  sigma=1;
                  in_wave=risultatoo2;

cordic_hls(in_wave,sigma,&wave_out_o);

wave_out_o=wave_out_o+rounding37;
                  wave_out_o2=wave_out_o>>38;
                  o_t[j]=wave_out_o2;


     }
// calcolo g_t
g_t_pre[j]=g_t_pre[j]+cmem[j]+b_c[j];
if(g_t_pre[j]<-805){
```

```
g_t[j]=-8;
}

else if(g_t_pre[j]>804){
g_t[j]=uno_long;
}
else {


                risultatog=g_t_pre[j]*costante;

risultatog=risultatog+rounding11;
                risultatog2=risultatog>>12;
                sigma=0;
                in_wave=risultatog2;

cordic_hls(in_wave,sigma,&wave_out_g);


wave_out_g=wave_out_g+rounding37;
                wave_out_g2=wave_out_g>>38;
                g_t[j]=wave_out_g2;

      }

    c_t_pre[j]=f_t[j]*c_t_1[j]+i_t[j]*g_t[j];
```

```
    c_t_pre2[j]=c_t_pre[j]+rounding_out;
    c_t_pre2[j]=c_t_pre2[j]>>3;
    c_t[j]=c_t_pre2[j];
    c_t_1[j]=c_t[j];
    if(c_t_pre[j]<-805){
    c_t_tanh[j]=-8;

  }


else if (c_t_pre[j]>804){
  c_t_tanh[j]=uno_long;
  }
  else{



 variante=c_t_pre[j]*costante;
  sigma=0;
  variante=variante+rounding9;
  variante2=variante>>10;
  in_wave=variante2;
  cordic_hls(in_wave,sigma,&wave_out_c);
  wave_out_c=wave_out_c+rounding37;
  wave_out_c2=wave_out_c>>38;
```

```
c_t_tanh[j]=wave_out_c2;
}
y_t_pre[j]=o_t[j]*c_t_tanh[j];
y_t_pre[j]=y_t_pre[j]+rounding_out;
y_t_pre[j]=y_t_pre[j]>>3;
y_t[j]=y_t_pre[j];
y_t_1[j]=y_t[j];
y_t_ext[j]=y_t[j];
}
```

Appendix C

Code for the Memcalc function of the LSTM Layer

```
#include "ap_int.h"
#define N 50
#define M 12
//void lstm(int5 x_t[N],int5 y_t[N]) {
void   memCalc(ap_int<6>   y_t[N],   ap_int<18>
imem_o[N],
ap_int<18>  fmem_o[N],  ap_int<19>  cmem_o[N],
ap_int<18> omem_o[N]) {
const ap_int<5> Wxi[] = {
#include "Wxi.txt"
};
```

```
const ap_int<5> Whi[] = {
#include "Whi.txt"
};
const ap_int<5> Wxf[] = {
#include "Wxf.txt"
};
const ap_int<5> Whf[] = {
#include "Whf.txt"
};
const ap_int<7> Wxc[] = {
#include "Wxc.txt"
};
const ap_int<6> Whc[] = {
#include "Whc.txt"
};

//const int5 Wxo[] = {
const ap_int<5> Wxo[] = {
#include "Wxo.txt"
};
const ap_int<5> Who[] = {
#include "Who.txt"
};
//const ap_int<7> b_i[] = {
const ap_int<8> b_i[] = {
#include "bi.txt"
};
```

```
//const ap_int<10> b_f[] = {
const ap_int<10> b_f[] = {
#include "bf.txt"
};
//const ap_int<8> b_o[] = {
const ap_int<9> b_o[] = {


#include "bo.txt"
};

   //const ap_int<8> b_c[] = {
   const ap_int<9> b_c[] = {
   #include "bc.txt"
   };


   ap_int<18> imem[N];

   ap_int<18> fmem[N];

   ap_int<19> cmem[N];

   ap_int<18> omem[N];

   int j,k,w;
```

```
//calcolo le memorie dipendenti da h:
imem_lbl:
for(j=0;j<N;j++){

imem[j]=0;

fmem[j]=0;

cmem[j]=0;

omem[j]=0;

lstm_label2:for(w=0;w<N;w++){
      imem[j] = imem[j] + Whi[j*N+w]*y_t[w];
      fmem[j] = fmem[j] + Whf[j*N+w]*y_t[w];
      cmem[j] = cmem[j] + Whc[j*N+w]*y_t[w];
      omem[j] = omem[j] + Who[j*N+w]*y_t[w];
}
   //le  costanti  di  polarizzazione  vengono
spostate nell'altro

blocco
//imem[j] = imem[j] + b_i[j];
imem_o[j] = imem[j];
//fmem[j] = fmem[j] + b_f[j];
```

```
fmem_o[j] = fmem[j];

//cmem[j] = cmem[j] + b_c[j];

cmem_o[j] = cmem[j];

//omem[j] = omem[j] + b_o[j];

omem_o[j] = omem[j];

}
}
```

Appendix D

C Code for the Testbench of the LSTM Layer

```
#include "sistema.hpp"
#include "ap_int.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#define N 50
#define M 12
#define S 370
int main()
{
     char num[12];
     int contaSeq;
```

```
ap_int<6> x_tb[M];

ap_int<6>h_tb[N];

ap_uint<1>reset_tb=0;

FILE *fp,*fp2;

char nomeFile[100];

char nomeFileout[100];

for(                    contaSeq=1;contaSeq
<(S+1);contaSeq++){

      strcpy(nomeFile,"Seq");



   strcpy(nomeFileout,"u");

   itoa(contaSeq,num,10);

   strcat(nomeFile,num);

   strcat(nomeFile,".txt");

   strcat(nomeFileout,num);

   strcat(nomeFileout,".txt");
```

```
fp=fopen(nomeFile,"r");
        fp2=fopen(nomeFileout,"w");
        reset_tb=1;

     int pass;
if(reset_tb==1){
   sistema(x_tb,h_tb,reset_tb);
   reset_tb=0;
}

          while (!feof(fp)){
              for                          (int
contaDim=0;contaDim<M;contaDim++){
              fscanf(fp,"%d,",&pass);
              x_tb[contaDim]=(ap_int<6>)pass;
             //fscanf(fp,"%d",&pass);

  }
```

```
                if(!feof(fp)){
                sistema(x_tb,h_tb,reset_tb);
                for (int i=0;i<50;i++){

fprintf(fp2,"%d\n",(int)h_tb[i]);
              }
              }

              }
          fclose(fp);
          fclose(fp2);

}
          //     /*
          //         * Applico gli ingressi al
sistema
          //      * */
          //       sistema(x, h);
          //       /*
```

```
            //      * Salvo le uscite
            //      * */
            //}
// }
          return 0;
}
```