# Parallelization of Recurrent Neural Network training algorithm with implicit aggregation on multi-core architectures

Thomas Messi Nguelé, Armel Jacques Nzekon Nzeko'o, Damase Donald Onana

# Parallelization of Recurrent Neural Network training algorithm with implicit aggregation on multicore architectures

**Thomas MESSI NGUELE**[1,2,3*], **Armel Jacques NZEKON NZEKO'O**[1,3], **Damase Donald ONANA**[1]

[1]University of Yaounde I, FS, Computer Science Department, Cameroon

[2]University of Ebolowa, HITLC, Computer Engineering Department, Cameroon

[3]Sorbonne Université, IRD, UMI 209 UMMISCO, F-93143, Bondy, France, Cameroon

*E-mail : thomas.messi@facsciences-uy1.cm

**Abstract**

Recent work has shown that deep learning algorithms are efficient for various tasks, whether in Natural Language Processing (NLP) or in Computer Vision (CV). One of the particularities of these algorithms is that they are so efficient as the amount of data used is large. However, sequential execution of these algorithms on large amounts of data can take a very long time. In this paper, we consider the problem of training Recurrent Neural Network (RNN) for hate (aggressive) messages detection task. We first compared the sequential execution of three variants of RNN, we have shown that Long Short Time Memory (LSTM) provides better metric performance, but implies more important execution time in comparison with Gated Recurrent Unit (GRU) and standard RNN. To have both good metric performance and reduced execution time, we proceeded to a parallel implementation of the training algorithms. We proposed a parallel algorithm based on an implicit aggregation strategy in comparison to the existing approach which is based on a strategy with an aggregation function. We have shown that the convergence of this proposed parallel algorithm is close to that of the sequential algorithm. The experimental results on an 32-core machine at 1.5 GHz and 62 Go of RAM show that better results are obtained with the parallelization strategy that we proposed. For example, with an LSTM on a dataset having more than 100k comments, we obtained an f-measure of 0.922 and a speedup of 7 with our approach, compared to a f-measure of 0.874 and a speedup of 5 with an explicit aggregation between workers.

**Keywords**

Deep Learning ; Recurrent Neural Network ; hateful messages recognition; parallel programming.

## I INTRODUCTION

The enormous amount of data generated by social media platforms is a boon for deep learning algorithms. Indeed, more the quantity of data used is important, more the algorithm is efficient. On the other hand, the sequential execution of these algorithms on these large amounts of data can take a very high execution time [3]. It is possible through an appropriate use of multicore architectures to reduce this execution time. In this paper, we are interested to identifying

aggressive messages that can be disseminated on social networks using Recurrent Neural Network (RNN) algorithms. In other words, we want to build a model to automatically detect an aggressive message, paying attention not only to the performance of the model but also to the time taken by the training algorithm to train the model. We propose an approach based on the use of Mutex synchronization for the parallel implementation of RNN training algorithms. The objective is to take advantage of multi-core hardware architectures to reduce the execution time of training algorithms while preserving their metrics performance.

We firstly implemented and compared the result of sequential executions of standard RNN, Long Short Time Memory (LSTM) and Gated Recurrent Unit (GRU) with the same datasets containing 115,864 comments, including 101,082 comments labeled as non-aggressive and 14,782 comments labeled as aggressive. What we can learn from these sequential executions is that the metric performances of the model obtained with an LSTM are quite better than those obtained with a standard RNN or a GRU, but the time for training the model with an LSTM is much bigger. We subsequently implemented each of the training algorithms in parallel, using and comparing two different approaches. One approach performing explicit aggregation (arithmetic average) between the processing units for the update phase and another using mutex software synchronization between processing units. After experimentation on a 32-core machine, we show better performance for the parallelization our approach (implicit aggregation).

The rest of this paper is organized as follows: section II presents the concepts necessary for the understanding our work. Section III presents the existing work related to ours, Section IV present and explain our parallel algorithm, Section V presents the current experimental results, and finally Section VI concludes and gives some perspectives.

## II BACKGROUND

Recurrent Neural Network [10] are types of neural networks used for processing sequential data like a text. Unlike the Feed-Forward Neural Network where information only propagates in one direction, from front to back, RNNs are neural networks with recurrent connections and where the information is propagated in both directions. In this paper, we use and compare 03 variants of RNN namely: standard RNN, LSTM, and GRU. A standard RNN simulates a discrete-time dynamical system. It is a system with an input $x_t$, an output $y_t$ and a hidden state $h_t$ formally defined like in equation 1, where $f_h$ and $f_o$ are parameterized state transition and output functions respectively. As shown by Pascanu et al. [6] a standard RNN has difficulties in practice to be able to handle very long sequences due to the problem of vanishing or exploding gradient. Several solutions have emerged to overcome these anomalies, such as the use of special architectures like LSTM and GRU both based on the use of a gate mechanism.

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) \\ y_t &= f_o(h_t) \end{aligned} \tag{1}$$

To handling long-term temporal dependencies in the data, an LSTM uses a memory cell ($c_t$) and 03 gates whose role is to control the flow of information in and out. We have the forgate gate ($f_t$) which allows to control the quantity of information to keep, the input gate ($i_t$) allows to select the information to add in $c_t$ and the output gate ($o_t$) allows to obtain the hidden state $h_t$ as a function of $c_t$. Formally, to have the hidden state $h_t$ with an LSTM, we proceed as in equation

2. Where All W denote the weight matrices and b are the biases (the model parameters). $\overline{c}_t$ designates the intermediate state of the secondary memory $c_t$ ( is the memory state before being filtered by the gates).

$$
\begin{aligned}
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\
\overline{c}_t &= tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \overline{c}_t \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
\tag{2}
$$

The sigmoid ($\sigma$) and hyperbolic tangent ($tanh$) activation functions were respectively used to calculate the gates and the hiden state. The main difference between LSTM and GRU is that, GRU has fewer parameters and use only two gates namely: reset gate ($r_t$) and update gate ($z_t$). The reset gate regulates the flow of new input to the previous memory and the update gate determines how much of the previous memory to keep. The calculation of hidden state $h_t$ with a GRU occurs as presented in equation 3.

$$
\begin{aligned}
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
\overline{h}_t &= tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t
\end{aligned}
\tag{3}
$$

Learning algorithm with RNN like others neural networks is based on iterative execution of two phases, the forward and the back-forward phase. In the forward phase we compute the outputs as shown in equations 1, 2 and 3. In the back forward phase we compute the partial derivatives of each parameter of the model in order to update them [8]. Algorithm II.1 presents the sequential training of RNN that we use for text classification. Gradient descent in its mini-batch version was used as an optimization procedure, choosing the cross-entropy loss as the error function. It is an error function used for multi-class classification problems (as in our case), which allows to quantify the difference between the predicted vector output ($y$) and the expected one ($\bar{y}$). It is defined as in equation 4 for one instance of the dataset.

$$
\ell(y, \bar{y}) = -\sum_{j=1}^{n} \overline{y}_j log(y_j) \tag{4}
$$

$$
\theta = \theta - \lambda \frac{\partial \ell}{\partial \theta} \tag{5}
$$

The parameter update is carried out as in equation 5, where $\lambda$ (the learning rate) is a parameter fixed before the start of learning and which may change or not throughout the learning. For

**Algorithm II.1** Sequential training algorihtm

```
Input. (X,Y), λ, M, epoch: maximal numbers of epoch
Output. Θ: the set of model parameters
Start
    Θ ← InitializeParameters( )   //  initialize  the  Θ parameters
    ( xtrain ,  ytrain ) ( xtest ,  ytest ) ← splitData(X, Y)
    B ← getBatch(M,xtrain, ytrain )
    e ← 1
    while e ≤ epoch
        for b_i ∈ B
            loss ← 0
            dΘ ← ZeroInitialize( )
            for (x, y) ∈ b_i
                y_pred ← Forward(x, Θ)
                loss ← loss + lossEntropy(y, y_pred)
                dΘ ← dΘ + Backforward(Θ, x, y, y_pred, loss)
            enfor
            update(Θ, dΘ, λ, M)
        enfor
        e ← e + 1
    endwhile
    Return Θ
End
```

the mini batch version of the gradient descent, the derivative $\frac{\partial \ell}{\partial \theta}$ is calculated by considering the error on M elements of the dataset, where M (the batch size) is a parameter which is also fixed before the start of the training. Finally, it is important to specify that the words in natural language constituting the text are first transformed into a vector used in the algorithm. For our study, we simply used a word2vec algorithm provided by gensim [1] python library.

Considering the iterative execution of the forward and backforward phase in algorithm 1, and setting: $\mathrm{n_e}$ the maximum number of epochs, $\mathrm{n_d}$ the number of elements of the dataset, $\mathrm{n_h}$ the number of neurons in the hidden layer, $\mathrm{n_v}$ the size of the input vector representing a word, $\mathrm{n_o}$ the size of the output vector, and $\mathrm{n_s}$ the maximum sentence size, then the worst-case time complexity for a standard RNN are given by $O(n_d n_e(n_o n_h + n_s n_h(n_h + n_v))$ and for LSTM and GRU are given by $O(n_d n_e(n_s n_h + n_s(n_v + n_h(n_h + n_v)))$. The main goal in this paper is to provide a parallel implementation of this algorithm in order to reduce the execution time and maintaining as much as possible prediction performance.

## III RELATED WORK

Several works have been carried out on the parallel implementation of neural networks algorithms. By analyzing this works, we mainly observe two parallelization approaches, namely model parallelization where the structure of neural network computations is exploited to speed up the calculation of the minibatch gradient [5] and data-level parallelization where the computation of gradient is distribued over many workers synchronously or asynchronously. We have for example Martin Abadi et al. [7] who in their work present the Synchronous Parallel

[1] https://radimrehurek.com/gensim/models/word2vec.html

Stochastic Gradient Descent (S-PSGD) algorithm. It is a parallel implementation of stochastic gradient descent exploiting data-level parallelization. The algorithm is synchronous in the sense that Each processing unit is responsible for training a copy of the model with a subset of data, it then synchronizes to aggregate their gradient and finally update the global model.

Jeffrey Dean et al. [3] who have developed a software framework called DistBelief that can use computing clusters with thousands of machines to train large models. They present in this framework the DownPour SGD algorithm which combines model parallelization and data-level parallelization asynchronously. The principle is that each processing unit is responsible for updating a subset of model parameters. This approach is asynchronous in the sense that, the processing units operate independently of each other.

Zhiheng Huang et al. [4] provide a parallel RNN training algorithm for language modeling. For their implementation, each processing unit contains in memory a complete copy of the model, but also a subset of the dataset. The difference with the S-PSGD algorithm is that, here, each processing unit updates its local model. This time, the synchronization step aggregates the parameters of the different local models to form the overall model.

The common point of the works that we have just cited is the fact that they all use data-level parallelization. The main difference is made on the strategy used for updating the global model. We propose in this paper a parallel algorithm based on a strategy using implicit aggregation thanks to mutex synchronization between processing units to perform updates. We compare this strategy with the one where the processing units update the model by using an explicit aggregation function.

## IV RECURRENT NEURAL NETWORK PARALLELIZATION

In this part of the paper, we first present our parallelization strategy at Section 4.1. Then we present the parallel algorithm based to this strategy at Section 4.2 and we finish with the presentation of algorithms convergence analysis at Section 4.3

### 4.1 Parallelization strategy

Stochastic gradient descent (SGD) is the most used optimization procedure when training deep learning model. Unfortunately, its traditional formulation is inherently sequential which makes it difficult to use with very large volumes of data [3]. This is because the time required to traverse the data fully sequentially can be expensive. An effective solution to reduce the training time of a neural network is the use of parallel programming. The main approach to parallelize the training of a neural network is to distribute the computation of the derivatives over several processing units by exploiting data-level parallelization [4]. By following this approach, we propose in this paper a parallel implementation strategy performing a mutex software synchronization between the processing units for update the global model. The principle is that, each processing unit is assigned a local copy and a subset of the dataset (see figure 1). They each use their local copy of the model to calculate the derivatives ($\nabla$P) of the parameters. After calculating the derivatives, each processing unit one after the other enters in critical section using a mutex variable to update the parameters of the global model accessible from a shared memory.

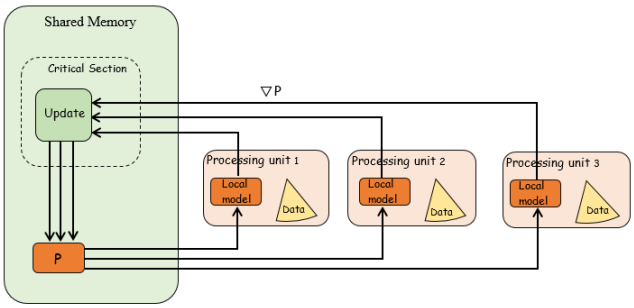


Figure 1: Parallelization with mutex synchronization. Each data part is assigned to a processing unit which compute the derivatives ($\nabla$P), and uses the critical section to update the global model.

This is therefore a parallelization at the data level because each processing unit will execute the same set of instructions simultaneously with the others but with different data. It is synchronous in the sense that, we use software synchronization (mutex) between the processing units for update operations.

### 4.2 Parallel algorithm with implicit aggregation

Here we present the parallel training algorithm which takes into account the parallel implementation method presented in the previous section. The implementation was carried out in a shared memory environment equipped with a multi-core architecture. The processing units then correspond to the different cores of a processor. Each core, through a thread executes simultaneously with the other threads a set of instructions. The main thread executes algorithm IV.1. It is responsible for initializing the parameters of the global model (line 2). At each epoch (line 4 to 11), it partitions the data into P subsets, where P designates the number of processing units available. It then launches the execution of the slave threads with a copy of the model parameters, as well as a subset of the data sets.

**Algorithm IV.1** ParallelTraining

```
1   Input. (X,Y), λ, M, epoch: maximal numbers of epoch, p: number of thread
2   Θ ← InitializeParameters( ) // initialize the Θ parameters
3   e ← 1,
4   while e ≤ epoch
5      partition the data (X,Y) into P part d_i
6      for each d_i
7        ParametersCopy(Θ, Θ_i)
8        ThreadCode(d_i, Θ_i, λ, M)
9      enfor
10     e ← e + 1
11  endwhile
```

Each slave thread executes algorithm IV.2 simultaneously with the other threads with a local copy of the model and a subset of the dataset. The execution is almost similar to that carried out in algorithm 1, where each slave thread partitions his subset of dataset ($d_i$) into several batches $b_i$ of size M (line 2). It then goes through each batch $b_i$, and for each sentence x in $b_i$, it predicts its class (line 7), compute and accumulates the prediction error as well as the partial derivatives (line 8 and 9). Once a slave thread finishes calculating derivatives, it enters in critical section

**Algorithm IV.2** ThreadCode

```
1   Input . (X,Y), λ, M, Θ_i: local model parameters, d_i: data partition
2   B ← getBatch(M,d_i)
3   for b_i ∈ B
4     loss ← 0
5     dΘ_i ← ZeroInitialize( )
6     for (x, y) ∈ b_i
7       y_pred ← Forward(x, Θ)
8       loss ← loss + lossEntropy(y, y_pred)
9       dΘ_i ← dΘ_i + Backforward(Θ_i, x, y, y_pred)
10    enfor
11    LOCKMUTEX
12      update(Θ, dΘ_i, λ, M)
13      parametersCopy(Θ, Θ_i)
14    FREEMUTEX
15  enfor
```

using a mutex variable to update the global model accessible from shared memory. It then retrieves the updated parameters for the next iterations (line 11 to 13). Finally, it releases the mutex variable (line 14) to allow another thread to perform the update.

#### *4.2.1 Complexity*

Considering algorithm 2 and 3, and let $\hat{\mathsf{I}}$ be the index of the subset of data having the largest size $|\mathsf{d}_{\hat{\mathsf{I}}}|$. The worst-case time complexity for a standard RNN is $\mathsf{O}(|\mathsf{d}_{\hat{\mathsf{I}}}|\mathsf{n_e}(\mathsf{n_o n_h} + \mathsf{n_s n_h}\ (\mathsf{n_h} + \mathsf{n_e}))$. and for LSTM and GRU is $\mathsf{O}(|\mathsf{d}_{\hat{\mathsf{I}}}|\mathsf{n_e}(\mathsf{n_s n_h} + \mathsf{n_s}(\mathsf{n_e} + \mathsf{n_h}(\mathsf{n_h} + \mathsf{n_e})))$.

#### *4.2.2 Maximum Speedup*

Considering the same maximum epoch number $\mathsf{n_e}$ used for the sequential version, the maximum speedup for this parallel implementation corresponds to the number of units of treatment (P) used. In practice it is difficult to have a speedup very close to the maximum speedup, this is mainly due to the portions of code executing sequentially for the central thread.

### 4.3 Convergence

In this section, we analyze theoretically the convergence of the three algorithms sequential, parallel with explicit aggregation and parallel with implicit aggregation. For this purpose, we express the update equations for each of the three algorithms. We consider one epoch and we assume there are $k$ batches, $\theta_k$ is the value of the parameters $\theta$ after k updates.

#### *4.3.1 Sequential Convergence*

In the case of sequential executing (algorithm II.1), $\theta_0$ is used to compute $\theta_1$, $\theta_1$ is used to compute $\theta_2$ and finally so far, $\theta_{k-1}$ is used to compute $\theta_k$. Let $\frac{\partial \ell}{\partial \theta_j}$ be the derivative calculated using $\theta_j$. At each step $i$, if we replace $\theta_{i-1}$ in $\theta_i$ by its expression contemning $\theta_{i-2}$, then we can write $\theta_k$ as in equation 6:

$$\theta_k = \theta_0 - \lambda \left( \sum_{j=0}^{k-1} \frac{\partial \ell}{\partial \theta_j} \right) \tag{6}$$

#### 4.3.2 *Explicit Aggregation Convergence*

For a parallel execution with an explicit aggregation (algorithm presented at [4]), we consider $P$ available processing units. Let:

- $agg()$ be the aggregation function at each update
- $\frac{\partial \ell}{\partial \theta_{i,j}}$ be the derivative calculated by any process unit using :
  - the $j^{th}$ batch and,
  - $\theta_i$ the value of the model parameters at the $i^{th}$ update.
- Let $u$ be the number of updates. Compared to the sequential execution, $u = \frac{k}{P}$.

$\theta_u$ can be written as in equation 7:

$$\theta_u = \theta_0 - \lambda \sum_{i=0}^{u-1} agg \left( \frac{\partial \ell}{\partial \theta_{i,iP+j}} \right)_{j \in [1...P]} \tag{7}$$

In the context of our experiments, $agg()$ function corresponds to the arithmetic mean.

#### 4.3.3 *Implicit Aggregation Convergence*

In this case (algorithm IV.1), each thread makes updates on the model according to its batches without waiting the other threads. It is then difficult to know which value of the model is used by a thread to perform derivatives necessary for these updates. In fact, a derivative computed by a thread depends on the current batch and also the value of the model it saw when it began its processing on this batch. This model value is one of the previous computed by one of the threads that did the update before. We catch this behavior as follow:

- Let $\frac{\partial \ell}{\partial \theta_{*,j}}$ be the derivative computed by any thread at the $j^{th}$ update with the batch $j^{th}$
- $\frac{\partial \ell}{\partial \theta_{*,j}} \in \{\frac{\partial \ell}{\partial \theta_{0,j}}, \frac{\partial \ell}{\partial \theta_{1,j}}, \dots, \frac{\partial \ell}{\partial \theta_{j-1,j}}\}$
- Each $\frac{\partial \ell}{\partial \theta_{i,j}}$ is defined as in section 4.3.2

We have k updates as in a sequential execution. The difference with the sequential execution is with the initial values at each updates of each thread. We can write $\theta_k$ as in equation 8

$$\theta_k = \theta_0 - \lambda \sum_{j=1}^{k} \frac{\partial \ell}{\partial \theta_{*,j}} \tag{8}$$

#### 4.3.4 *Analysis of the three Convergences*

According to equations 6,7,8, we can observe the difference between the explicit and implicit aggregation as follows:

1. Number of iteration: we have $k$ updates with implicit aggregation as in sequential algorithm while there are only $\frac{k}{P}$ updates with explicit aggregation.
2. Aggregation function $agg()$: The aggregation function used in explicit aggregation can lead to loss of information. In fact, if this function does not operate properly, part of information learned by threads on their respective batches may be lost. In opposite, in implicit aggregation, threads integrates their learned information progressively in sequential fashion thanks to mutex. Like that, the lost of information is reduced.

This two observations show that the convergence of the implicit aggregation is closer to the sequential algorithm convergence compared to the convergence of the explicit aggregation.

In the next section, we are going to confirm these observations experimentally.

## V EXPERIMENTS AND RESULTS

We present in this section our experimental results obtained for sequential and parallel execution. The experiments were carried out on a multi-core machine with 32 cores (at 1.5 GHz) and 62 Go of Ram. The implementations of the training algorithms were done with C language and using the posix thread library for parallel implementations, the pre-processing operations on the data were done with python language. The dataset we used was constructed by [9]. This dataset contains 115,864 labeled discussion comments from English Wikipedia. Each comment was labeled by approximately 10 annotators via Crowdflower on how aggressive the comment was perceived to be, we finaly have 101,082 comments considered non-aggressive by the annotators, and 14,782 comments labeled as aggressive. We applied a set of pre-processing operations to the data, namely data cleaning, stop word removal, tokenization and vector representation using the skipgram algorithm provided by the gensim python library. The datasets were then split into two, namely 80% for training, and 20% for the testing phase.

Since we are performing a text classification task (know if a comment is aggressive or not), and the datasets used are unbalanced, the appropriate metrics performance to evaluate the model are the unweighted averages of: precision, recall, and f-measure. The description and formula of these metrics are given in the table 1, where $l$ is the number of class, $tp_i$ (true positives), the number of correctly recognized examples for class $i$, $tn_i$ (true negatives), the number of correctly recognized examples that do not belong to class $i$, $fp_i$ (false positives), numbers of examples that either were incorrectly assigned to the class $i$ and $fn_i$ (false negatives) the numbers of examples were not recognized for class $i$ [1].

Table 1: Metric measurement descriptions and formula to evaluate the model.

| Measure | Description | Formula |
|---|---|---|
| **Recall** | An average per-class effectiveness of a classifier to identify class labels | $\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i+fp_i}}{l}$ |
| **Precision** | An average per-class agreement of the data class labels with those of a classifiers | $\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i+fn_i}}{l}$ |
| **F-measure** | Relations between data's positive labels and those given by a classifier based on a per-class average | $\frac{2\times Recall\times Precision}{Recall+Precision}$ |

The performance due to parallelization of an algorithm will be measured using the following metrcic :

- **Execution time T(p)**: Is the execution time taken by a parallel program using $p$ threads.
- **The Speedup S(p)**: Is the ratio between the execution time with one resource, on the execution time on p resources.

$$\mathsf{S(p)} = \frac{\mathsf{T(1)}}{\mathsf{T(p)}} \tag{9}$$

### 5.1 Sequential execution

The hyper parameters values used for running the learning algorithm for each of the RNN variants are as follows: the maximum number of epochs (15), the learning rate $\lambda$ (0.01), the batch size M (32), the total number of $\mathsf{n_h}$ neurons in the hidden layer (80), the size $\mathsf{n_v}$ of input vectors representing a word (100). Figure 2 presents the loss evolution throughout the sequential training algorithm. We see that training with an LSTM was much more stable than that with a standard RNN, and convergence was faster compared to that observed with a GRU.

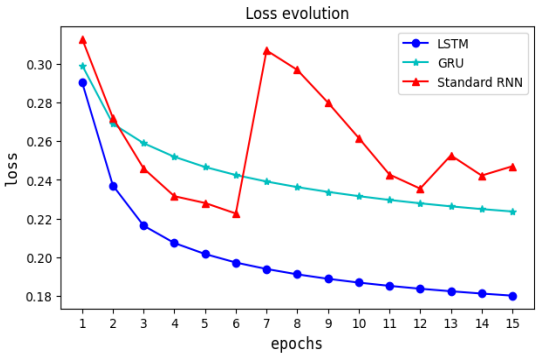


Figure 2: Loss evolution through the learning

Table 2: Metric measurements and time taken for each RNN variants.

| | Precision | Recall | F-measure | Training time (s) |
|---|---|---|---|---|
| **LSTM** | **0.925** | **0.930** | **0.925** | **13390** |
| GRU | 0.912 | 0.919 | 0.911 | 9717 |
| Standard RNN | 0.91 | 0.922 | 0.917 | 3143 |

We then used the test dataset to obtain the metric performance for each of the models. Table 2 presents the metric results obtained as well as the training time taken with each of the variants of the RNN. We observe that the metric performances of the model obtained with an LSTM are superior to the others. Even if here these differences are quite slight, we saw in figure 2 that the LSTM provided the lowest prediction error, which confirms the fact that, for this particular dataset, the model obtained with an LSTM would be the best choice. However, using an LSTM involves a higher training time compared to other RNN variants, which was predictable, given the number of parameters and the workload which is much higher. We then implemented the training algorithms in parallel, hoping reduce the execution time while keeping the metric measures

### 5.2 Parallel Execution

In order to perform comparisons, we used two parallel implementation strategies for the training algorithm. The first (parallel with mutex) is the idea of parallelization that we propose in this paper. The second one (parallel with aggregation) is the one carrying out an aggregation (in this case the arithmetic mean) of the various local derivative to update the global model. figure 3 presents the speedup (time saving) obtained as a function of the number of processing units used for each version of RNN with the two parallel implementation strategies.

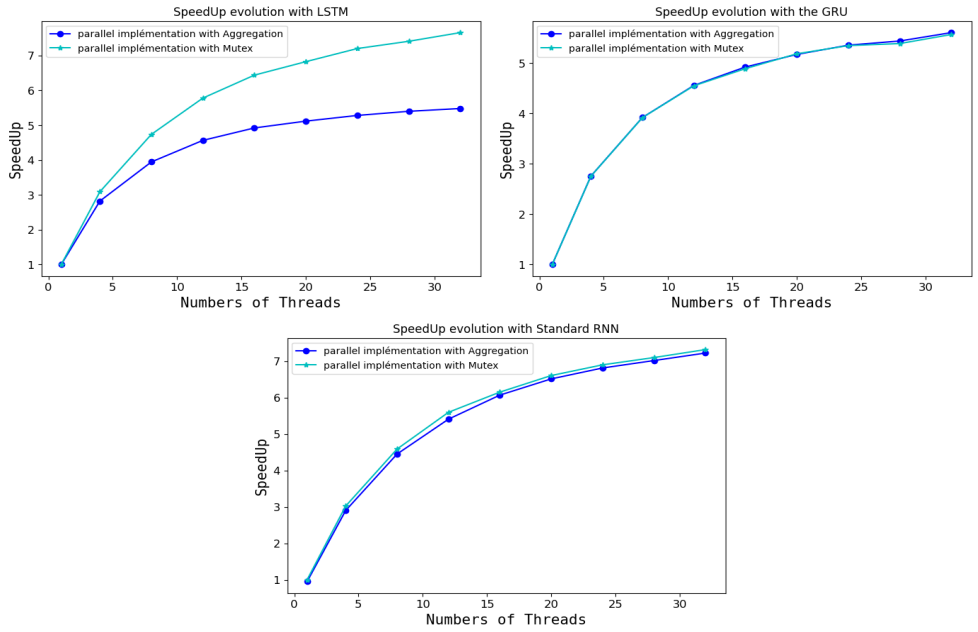


Figure 3: Evolution of speedUp depending on the number of cores

By observing the figure 3 we note that, the difference in time gain between the two parallel implementation methods is more significant with an LSTM. For example, we obtained a speedup of 7 for the parallel training algorithm with mutex of an LSTM using the 32 available processing units, compared to a speedup of 5 using the parallel algorithm performing the arithmetic average. The hypothesis justifying these results is the fact that, the parallel implementation method with mutex provides better results when the workload is greater. This is the case with an LSTM. We also observed the evolution of the convergence of the model for each of the two strategies using graphs showing the evolution of the training error as a function of the number of epochs (see figure 4). By examining these graphs, we note that the model converges better with

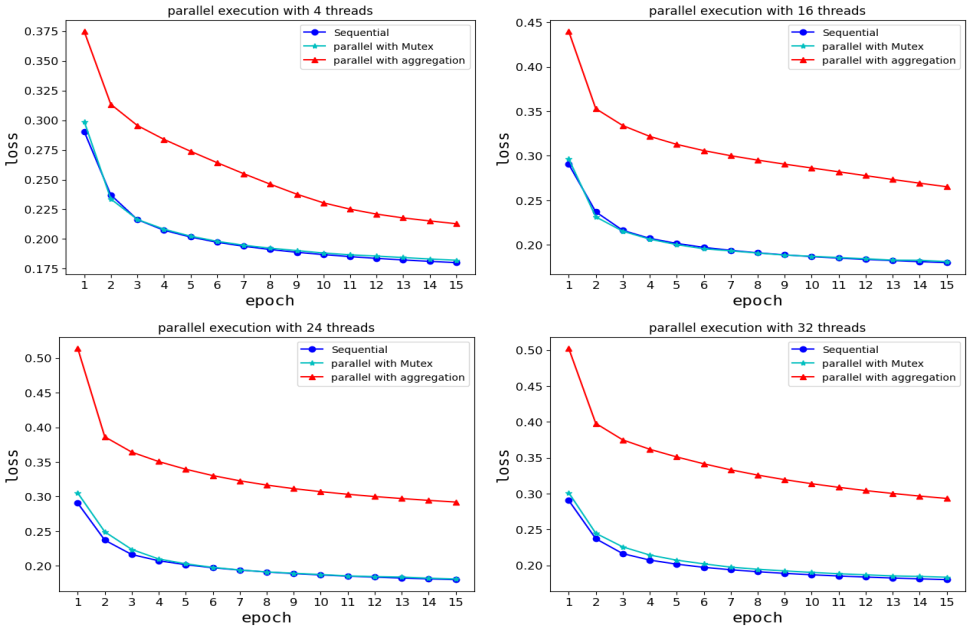


Figure 4: Error evolution with LSTM

the strategy of parallelization without aggregation compared to that carrying out the arithmetic average of the various local models.

We also note that the number of processing units used has a more negative impact on the strategy with aggregation. We can for example see that with eight processing units, the convergence of the model using this strategy is much worse than that with two processing units. Table 3 presents the performance metrics obtained with each of the two parallelization strategies using the eight processing units available, and in comparison with those obtained with sequential execution. By observing table 3, we see that for one or the other of the RNNs, we obtain models with better metric performances when we use the parallel implementation strategy using mutex synchronization. We can conclude on the fact that, it is more interesting to use a parallelization strategy without aggregation compared to that carrying out an aggregation between the local models of the various processing units. One of the main difficulties or disadvantages of strategies using an aggregation is the fact of having to choose the appropriate method or aggregation function guaranteeing good convergence and saving time.

Table 3: Table presenting the precision metrics for parallel implementations (with 08 threads) as well as the time taken to train the model

| | | Precision | Recall | F-measure | Training time (s) |
|---|---|---|---|---|---|
| LSTM | sequential | 0.925 | 0.930 | 0.925 | 13390 |
| | **parallel with Mutex** | **0.927** | **0.930** | **0.922** | **1748** |
| | parallel with aggregation | 0.896 | 0.90 | 0.874 | 2443 |
| GRU | sequential | 0.912 | 0.919 | 0.911 | 9717 |
| | **parallel with Mutex** | **0.914** | **0.919** | **0.907** | **1745** |
| | parallel with aggregation | 0.894 | 0.902 | 0.882 | 1740 |
| Standard | sequential | 0.917 | 0.922 | 0.917 | 3143 |
| | **parallel with Mutex** | **0.912** | **0.917** | **0.914** | **429** |
| | parallel with aggregation | 0.888 | 0.898 | 0.877 | 435 |

## VI CONCLUSION

In this paper we first compare sequential executions of standard Recurrent Neural Network (RNN), Long Short Time Memory (LSTM) and Gated Recurrent Unit (GRU), on aggressive recognition task using the same datasets which contain 115,864 labeled discussion comments from English Wikipedia. We have shown that the LSTM provides better metric performance (an f-measure of 0.925 compared to 0.911 and 0.917 respectively with a GRU and standard RNN). However an LSTM implies a more important training execution time (13390 s) in comparison with a GRU (9417 s) and a standard RNN (3143 s). We then proceeded with parallel implementations of the training algorithms, to get both good metric performance and reduce execution time. We proposed the parallelization strategy with mutex synchronization in comparison to the existing approach which is based on a strategy with an explicit aggregation function. The experimental results on an 32-core machine at 1.5 GHz and 62 Go of RAM, show that better results are obtained with the parallelization strategy that we proposed. In fact, for the parallelization of an LSTM using the mutex synchronization, we obtain a speedUp of 7 and an f-measure of 0.922 , compared to a f-measure of 0.874 and a speedup of 5 with an explicit aggregation strategy between workers

As future work, we plan to continue our experiments by using more hardware resources and more larger datasets. We also want to experiment with the case where there is no mutex synchronization between threads, each thread will be able to update the model whenever it wants without any mutex barrier or other. We want to observe the impact of this way of doing things in terms of model convergence and time savings. It would also be interesting to use and compare other deep learning algorithms like transformers, which are successful at the moment especially for Natural Language Processing tasks.

## VII AKNOWLEGMENT

The successful completion of this research has been made possible through the generous support and collaboration of various scholarship stakeholders. We express our sincere gratitude to International Development Research Centre (IDRC) , Swedish International Development Cooperation Agency (SIDA) and African Center for Technology Studies (ACTS) for awarding the Artificial Intelligence for Development (AI4D) scholarship programme that funded this

research. This scholarship not only provided financial support but also served as a source of motivation and encouragement throughout the project. We are grateful for the support of all members of our work team namely High performance data science (HIPERDAS), who contributed to stimulating discussions and shared their insights. The collaborative environment fostered by our academic community has been integral to the development of this work. Lastly, we extend our thanks to all those who, directly or indirectly, played a role in the realization of this paper. Your support has been crucial, and we are grateful for the opportunities provided by the scholarship stakeholders and the broader academic community.

## References

[1] M. Sokolova and G. Lapalme. "A systematic analysis of performance measures for classification tasks". In: *Information processing & management* 45.4 (2009), pages 427–437.

[2] B. Recht, C. Re, S. Wright, and F. Niu. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in neural information processing systems* 24 (2011).

[3] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems* 25 (2012).

[4] Z. Huang, G. Zweig, M. Levit, B. Dumoulin, B. Oguz, and S. Chang. "Accelerating recurrent neural network training via two stage classes and parallelization". In: *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*. IEEE. 2013, pages 326–331.

[5] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. "Gpu asynchronous stochastic gradient descent to speed up neural network training". In: *arXiv preprint arXiv:1312.6186* (2013).

[6] R. Pascanu, T. Mikolov, and Y. Bengio. "On the difficulty of training recurrent neural networks". In: *International conference on machine learning*. Pmlr. 2013, pages 1310–1318.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[8] G. Chen. "A gentle tutorial of recurrent neural network with error backpropagation". In: *arXiv preprint arXiv:1610.02583* (2016).

[9] E. Wulczyn, N. Thain, and L. Dixon. "Wikipedia Talk Labels: Aggression". In: (Feb. 2017).

[10] R. M. Schmidt. "Recurrent neural networks (rnns): A gentle introduction and overview". In: *arXiv preprint arXiv:1912.05911* (2019).