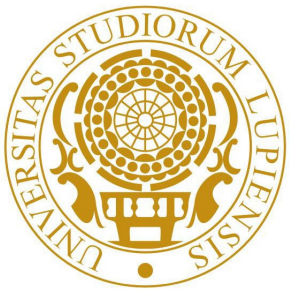
UNIVERSITY OF SALENTO



Department of Engineering for Innovation

Master of Science Degree in Computer Engineering

Parallel Algorithms Project

Parallel Auto-encoder for Efficient Outlier Detection

Supervisor
Prof. Massimo Cafaro

Student
Emanuela Paladini
20036492

Academic year 2020/2021

Contents

1 Introduction **1**
1.1 Auto-Encoder 2

2 Parallel Design **5**
2.1 Partition/Agglomeration 7
2.2 Communication 7
2.3 Experiments and Results 8
2.3.1 Experimental Setup 8
2.3.2 Results 9

3 Complexity **12**
3.1 Sequential Algorithm 12
3.2 Parallel Algorithm 15

List of Figures

1.1 Auto-encoder composed by three layers. 2
1.2 Sigmoid Function. 3

2.1 PODAE pseudo code. 6
2.2 Running Time in function of number of processes. 10
2.3 Speedup in function of number of processes. 11

Chapter 1

Introduction

The following work is based on the implementation of a parallel auto-encoder, as proposed by the work of Yunlong Ma, Peng Zhang, Yanan Cao, Li Guo, "Parallel Auto-encoder for Efficient Outlier Detection", 2013 IEEE International Conference on Big Data. This auto-encoder is used for the detection of anomalous values from big data, which plays an important role for example in network security. In particular, they build a replicator model of the input data to obtain the representation of sample data. Then, the replicator model is used to measure the replicability of test data, where records having higher reconstruction errors are classified as outliers. In data mining and statistics, outliers refer to observations/samples that are numerically distant from the rest of the data.

Using sequential auto-encoders it becomes difficult to deal with big data: auto-encoder cannot process large-scale data sets efficiently, due to its serial implementation.

1.1 Auto-Encoder

An auto-encoder is an artificial neural network used for learning efficient codings, which aims to learn a compressed representation (encoding) from a data set. As shown in Fig. 1.1, an auto-encoder generally contains three layers:

- An input layer, where the neurons correspond to elements of an input record.
- Hidden layers, which encodes the input record by mapping/compressing functions: the input sample is compressed to a hidden representation.
- An output layer, that shares the same number of neurons with the input layer, where each neuron represents the same meaning.

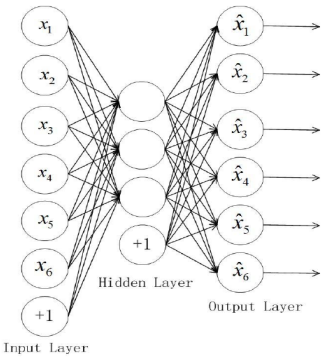


Figure 1.1: Auto-encoder composed by three layers.

The training of auto-encoder consists of two steps: encoding and decoding. In the encoding step, input data $x \in [0,1]^d$ are mapped to a hidden representation $y \in [0,1]^{d'}$ through a deterministic mapping function, as in the following equation:

$$y = s(Wx + b) \tag{1.1}$$

where **s** is a non-linear operator, in this case the sigmoid function, and y is the latent representation (code). The sigmoid function is described by the following:

$$s(t) = \frac{1}{1+e^{-t}} \tag{1.2}$$

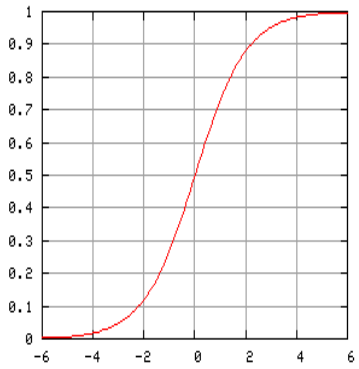


Figure 1.2: Sigmoid Function.

The sigmoid function squeezes the **t** value into the range [0,1]. The neuron that uses this activation function goes into saturation when it always emits 1; in this case, the gradient of the sigmoid function is always zero and therefore there is no more updating of the weights and there is no learning by the network.The derivative of the sigmoid function is:

$$\frac{d}{dt}s(t) = s(t)(1-s(t)) \tag{1.3}$$

In the decoding step, y is mapped back into a reconstruction $\hat{x}$ of the same shape as x through a similar transformation:

$$\hat{x} = s(W^{'}y + b^{'}) \tag{1.4}$$

In this step, the auto-encoder is optimized by minimizing the average reconstruction error which can be measured as the squared error:

$$L(x,\hat{x}) = \|x-\hat{x}\|^2 \tag{1.5}$$

Since y can be viewed as a lossy compression of x, the method generally cannot learn a perfect representation for all x. The auto-encoder derives low reconstruction error, if test and training samples share the same distribution. On the other hand, high reconstruction error can be view as an outlier. In this case, test data share the same distribution with training data, so this method can accurately locate outliers in the test data by ranking the reconstruction error, which can be measured based on the probability distribution of the training data. For example, when the probability distribution is Gaussian, we can use the traditional squared error, as in our case.

The dataset is generated with the dimensions specified in input by the user and with a Gaussian distribution with mean equal to 0 and variance equal to 1. The values are then normalized.

Chapter 2

Parallel Design

In the figure 2.1 we can see the pseudo code of the Parallel Outlier Detection method based on Autoencoder (PODAE for short). In the first two steps, the trainingset, consisting of N samples, is split into the k computing units, giving each T samples. In parallel, each computing unit initializes the network and trains it, performing the feedforward and backpropagation steps on its own portion of data. In Line 12, the encoding and decoding parameters are obtained by aggregating results from all computing units.

In the test phase, the testset was split in the same way on the k computing units and in parallel it was given as input to the network. The reconstruction error was calculated for each samples, as the mean squared difference between each value of the sample and the reconstructed value. The errors array, or the so called **outlier factors**, was then sorted in descending order. At the last step, they evaluate the detecting performance by comparing the Precision, Recall (also known as Sensitivity) and F1-score, defined by the following formulas:

$$\text{F1-score} = \frac{2 \cdot Precision \cdot Sensitivity}{Precision + Sensitivity} \tag{2.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{2.3}$$

where TP is the number of True Positive instances, FP is the number of False Positive instances and FN is the number of the False Negative instances.

Algorithm 1 The PODAE algorithm

Input:
Training set $\{\mathbf{x}_i | \mathbf{x}_i \in \mathbb{R}^d, i = 1, \cdots, N\}$,
Testing set $(\{\mathbf{x}'_j, \mathbf{l}_j) | \mathbf{x}'_j \in \mathbb{R}^d, \mathbf{l}_j \in \mathbb{R}, j = 1, \cdots, N'\}$,
Learning rate α, number of computing units k.
Output:
A descending output set ranked according to reconstruction error.

1) Define $T = \lfloor N/k \rfloor$.
2) Randomly partition the training set, giving T examples to each computing unit.
3) **for all** $i \in \{1, \ldots, k\}$ **parallel do**
4) Randomly shuffle the data on unit i.
5) Initialize encoding and decoding parameter $\theta_{i,0}, \theta'_{i,0}$.
6) **for all** $t \in \{1, \ldots, T\}$ **do**
7) Get the tth example on the ith unit $x^{i,t}$,
8) $\theta_{i,t} \leftarrow \theta_{i,t-1} - \alpha \nabla_\theta J(\theta_{i,t-1}; x^{i,t})$,
9) $\theta'_{i,t} \leftarrow \theta'_{i,t-1} - \alpha \nabla_\theta J(\theta'_{i,t-1}; x^{i,t})$.
10) **end for**
11) **end for**
12) Aggregate from all computing units $\theta = \frac{1}{k} \sum_{i=1}^{k} \theta_{i,t}$, $\theta' = \frac{1}{k} \sum_{i=1}^{k} \theta'_{i,t}$.
13) **for all** $n' \in \{1, \ldots, N'\}$ **parallel do**
14) Calculate testing output $\hat{\mathbf{x}}'$.
15) Obtain reconstruction error $OF_j = \frac{1}{d} \sum_{i=1}^{d} (\mathbf{x}'_{ji} - \hat{\mathbf{x}}'_{ji})^2$.
16) **end for**
17) **Parallel do:** reorder OF in descending order.
18) Compute precision, recall and F1 measure.

Figure 2.1: PODAE pseudo code.

In this case, the dataset was generated randomly and is not labeled. To determine an outlier, the percentile was calculated for each output and occurrences with a percentile greater than or equal to 90 were selected as outlier.

2.1 Partition/Agglomeration

In **partition** phase, a domain partition is made between the processes, then the data is partitioned. Initially a fine-grained partition is made, assigning a single sample to each primitive task. In **agglomeration** phase it is decided to agglomerate and give N/k samples to each process. This decision is not made to reduce communications, as they are not present between processes during computation.

2.2 Communication

Each sample is given to the network sequentially, so it is processed independently from the others. The weight matrices are updated after each backpropagation step, so even in this case we have no functional dependencies. The only communications present are those relating to the reading of the input by process 0 and then sending the data to the other processes through the **Scatterv**. We use the variable size version because each process can receive upper or lower whole part of N/k samples. At the end of the training each process will have two partial weight matrices, encoding and deconding matrices. We will obtain the final weights matrices, carring out an **Allreduce**, so that all the processes have all the partial results and calculate an average of the values.
In the test phase, the I/O operation is repeated by process 0, which reads data

from files and distributes them to other processes with the **Scatterv**. In this phase, the weights are not updated, but the reconstruction error is calculated independently on each sample. Each process will therefore have an error value for each sample it received as input. Processes sort their error vector and send it to process 0 with a **Gatherv**. Finally, process 0 performs a merge of the K ordered vectors received, exploiting a min heap.

The merge procedure is as follows:

1. Create an output array.
2. Create a min heap of size K and insert 1st element in all the arrays into the heap.
3. Repeat following steps while priority queue is not empty:
    - Remove minimum element from heap (minimum is always at root) and store it in output array.
    - Insert next element from the array from which the element is extracted.

2.3 Experiments and Results

2.3.1 Experimental Setup

How to run:

- mpiCC autoencoder.cpp -c -lm
- mpiCC utils.cpp -c -lm

- mpiCC autoencoder.o utils.o -O3 ParallelAutoencoder.cpp -o ParallelAutoencoder -lm
- mpirun -np <number of processes> ./ParallelAutoencoder <trainingset-filename.txt> <number of training samples> <length of samples> <testset-filename.txt> <number of testing samples>
  For example, **mpirun -np 4 ./ParallelAutoencoder trainset.txt 80 5 testset.txt 20**; in this case, we run on 4 processes and a file "trainingset-filename.txt" containing 80 arrays of length 5 and a file "testset-filename.txt" of 20 arrays of the same length will be created.

Computer on which the experiments are carried out:

- Processor: Intel Core i7-8550U CPU 1.80GHz
- Number of core: 4
- Number of thread: 8
- RAM: 8 GB
- Operating System: Windows Subsystem for Linux (WSL) v.1 executed on Windows 10 Home x64.

2.3.2 Results

All the experiments were conducted with fixed values of the hyperparameters:

- number of epochs = 300
- learning rate = 0.25

- momentum = 0.9
- number of hidden neurons = 20
- number of features for item = 5

From the results in the table 2.1 and the graph 2.2 we see a decrease in running time as the number of processors increases, of which we see the greatest improvement in the case of the larger input size, going almost 5 times faster with 8 processes.

Table 2.1: Execution time (in seconds) in function of the dataset size and number of processes

Datasets Size	**Number of Processors**							
Train - Test	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
80 - 20	0.140856	0.077435	0.071335	0.046569	0.052578	0.047532	0.041029	0.038194
160 - 40	0.292982	0.152177	0.109279	0.105975	0.093695	0.089870	0.118388	0.067453
320 - 80	0.577124	0.291798	0.219507	0.182089	0.205930	0.176170	0.159595	0.147895
640 - 160	1.119979	0.585481	0.437533	0.352520	0.365590	0.320141	0.301567	0.279063
1280 - 320	2.369531	1.149229	0.831453	0.651707	0.722311	0.640882	0.594214	0.542315
2560 - 640	4.425377	2.263404	1.729170	1.434123	1.372956	1.242163	1.158200	1.099360
5120 - 1280	8.984855	4.743824	3.539837	2.774738	2.704108	2.612345	2.326071	2.142915
10240 - 2560	18.453533	9.521090	6.812563	5.803846	5.268860	5.100524	4.734908	4.458039

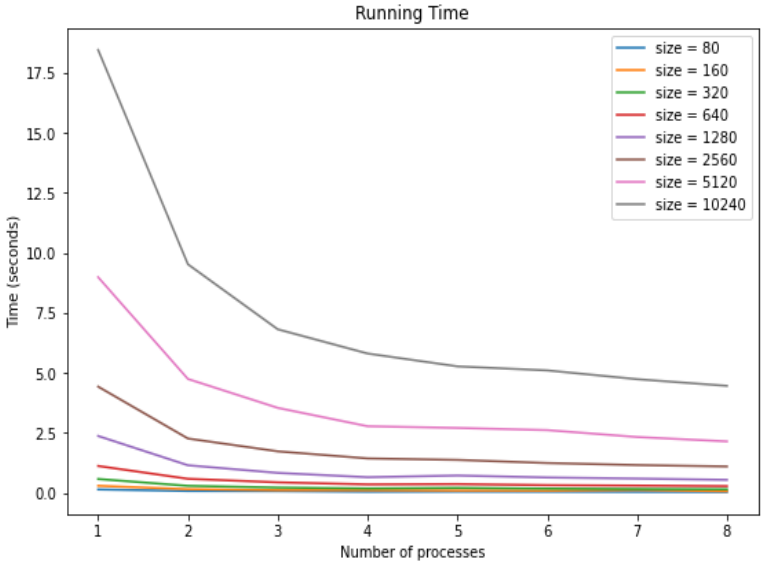


Figure 2.2: Running Time in function of number of processes.

From the speedup graph we see an increasing speedup, the maximum of which is in about 4.37 with 8 processors on the size of the dataset equal to 1280 and 160.

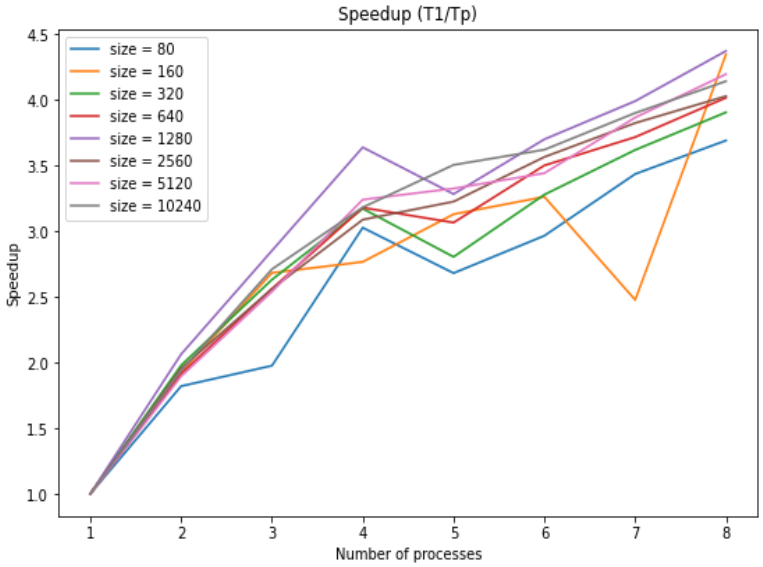


Figure 2.3: Speedup in function of number of processes.

Chapter 3

Complexity

Assume that number of epochs **e**, number of hidden neurons **h** and number of features for item **m** are constant. Furthermore let **n** be the number of item of training set and **t** the number of item of testing set. Each item of both training set and testing set has a dimension of $1 \times m$.

3.1 Sequential Algorithm

To analyze time complexity of sequential algorithm it's important to define the two functions Train() and Test() that recall other two functions: Feedforward() and Backpropagate(). See the following pseudo-codes:

```
Feedforward(input)
//encode
1) for i=0 to h
2)     for j=0 to m
3)         encode input
4)     sigmoid(input)
//decode
5) for i=0 to m
6)     for j=0 to h
7)         decode input
8)     sigmoid(input)
9)     compute error
```

Table 3.1: Feedforward() function

```
Backpropagate()
1) for i=0 to m
2)     compute Δ
3)     for j=0 to h
4)         update decoder weigths
5) for i=0 to h
6)     for j=0 to m
7)         update encoder weigths
```

Table 3.2: Backpropagate() function

```
Train(input)
1)    Feedforward(input)
2)    Backpropagate()
```

Table 3.3: Train() function

```
Test()
1)    Feedforward(input)
```

Table 3.4: Test() function

Since **m** and **h** are fixed in this algorithm, the functions Feedforward() and Backpropagate() not depend on the size of the input and therefore also Train() and Test() functions. The following pseudo-code describes the sequential algorithm:

SEQUENTIAL PSEUDO-CODE	Analysis
//training	
1) for i=0 to e	O(1)
2) for j=0 to n	O(n)
3) **Train(**input**)**	O(1)
//testing	
4) for i=0 to t	O(t)
5) **Test(**input**)**	O(1)
6) qsort()	O(t^2)

Table 3.5: Sequential pseudo-code

From previous analysis and the pseudo-code in Table 3.5, overall Sequential Time is:

$$T_1(n,t) = O(n+t+t^2) = O(n+t^2) \tag{3.1}$$

3.2 Parallel Algorithm

Below the table 3.6 shows the pseudo-code of parallel algorithm and its analysis:

PARALLEL PSEUDO-CODE	Analysis
//training	
1) **Scatterv**	O($n\ logp$)
parallel do {	
2) for i=0 to e	O(1)
3) for j=0 to $\frac{n}{p}$	O($\frac{n}{p}$)
4) **Train(**input**)**	O(1)
}	
5) **AllReduce**	O($logp+n$)
//testing	
6) **Scatterv**	O($t\ logp$)
parallel do {	
7) for i=0 to $\frac{t}{p}$	O($\frac{t}{p}$)
8) **Test(**input**)**	O(1)
9) qsort()	O($(\frac{t}{p})^2$)
10) **Gatherv**	O($t\ logp$)
}	
11) Merge p sorted array	O($t\ logp$)

Table 3.6: Parallel pseudo-code

From previous analysis, overall parallel complexity is:

$$T_p(n,t) = O\left(n + \left(\frac{t}{p}\right)^2 + (n+t)\ logp\right) = O\left(n + \left(\frac{t}{p}\right)^2 + n\ logp\right) \quad (3.2)$$

We neglect the terms n/p and t/p because they are of lower order than n and $(t/p)^2$ respectively. Finally, we also neglect the term $tlogp$.