Forward and Back Propagation on Artificial Neural Networks for the Classification of Movie Genres from their Plots

Sergio Gentilini [481089] - Bill Mono [479379]

I. Introduction

The goal of this project is to implement a solution to a given problem using serial code and then optimize the solution by applying the basic skills of parallel programming.

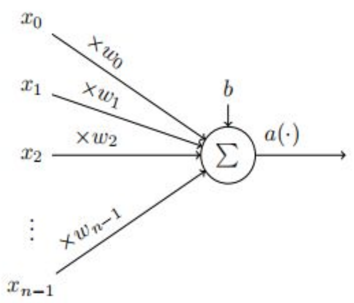
In this project, we tried to implement a neural network in order to classify the genre of a film through features derived from its plot. Once the problem was defined and a solution implemented, we wondered how it was possible to optimize the network execution through parallel-programming techniques.

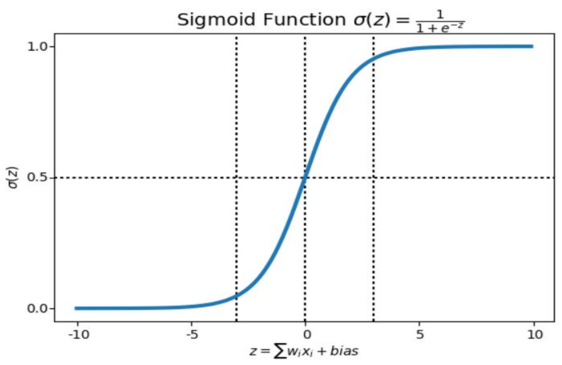
The simulations were done on the Google Cloud platform, using different instances; Ubuntu was chosen as the operating system.

II. Neural Networks

The algorithm used in neural networks is inspired by the neurons in our brain. The network is designed to recognize patterns in complex data: these are numerical, contained in vectors, into which all real-world data, be it images, sound, text, or time series, must be translated.

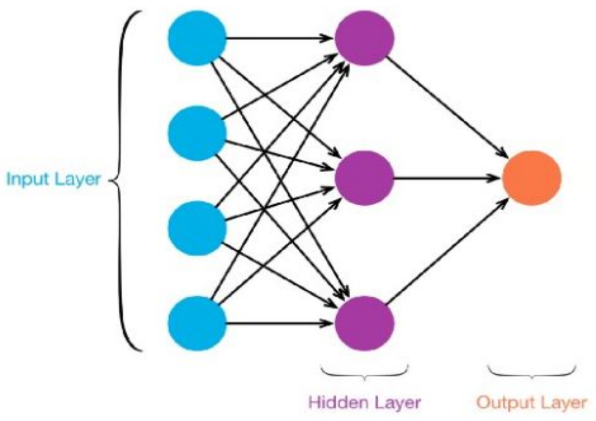
A neural network simply consists of neurons or nodes. They are connected, with each neuron holding a number and each connection holding a weight. Neurons perform simple computations, and the combination of the processing capabilities of multiple neurons produces complex behaviours. A single neuron is modelled as having an activation level, which is the output of a suitable activation function *a(·)*. For our project we chose a sigmoid function, which simply scales the output between 0 and 1.





An important class of neural networks is that of feed-forward networks, which corresponds to a directed acyclic graph: that is, a graph with one-way connections and no loops. A kind of feed-forward network that is particularly easy to design and to implement is the multi-layer perceptron (MLP).

In a multi-layer perceptron, neurons are divided into layers which are organized in a sequence, and a neuron in a layer is connected to all the neurons in the following layer. No other connections are allowed. The first layer is uniquely composed of input neurons, while the last layer contains all the output neurons. The intermediate layers are made of the so-called hidden neurons. The number of neurons in the input and output layers correspond to the dimension of the problem.



In our project we have decided to only use a single hidden layer, because we have assumed that the execution of the hidden layers following the first one should still take place sequentially.

III. Dataset

The used dataset[1] relates to around 14000 movies, together with their plots and related tags. Tags represent either genres (such as *horror*, *romantic* and *dramatic*) or themes (for example, *inspiring*, *revenge* and *suspenseful*).

For our case, we decided to only consider movies belonging to two easily-separable genres, namely *romantic* and *murder*, in order to perform binary classification. The features are represented by the movie plots, which have been transformed via TF-IDF through a dictionary containing the 1000 most-frequent words found in the plots, excluding *stop words*. The target class assumes a value of 1 for *murder* movies and 0 for *romantic* movies.

The final, transformed dataset provides a total of almost 7000 observations. In our tests we have used a fraction of this amount, as specified in the specific sections; in all cases, we considered an 80% training-test split. Data transformation and feature extraction were performed in Python.

In order to store the observations, we used a C structure that features two *double* pointers, identified as *feat* and *out*: the former stores the bias, which is initialised as 1.0, and the extracted values of each feature, so it includes 1001 elements; the latter is used to store the class, which means it can be either 0 or 1. The result is a pair of structures that store the training and validation data.

IV. Analysis of the Serial Algorithm

The neural network that we implemented is composed of 1001 different inputs (which, as mentioned above, represent the 1000 most-frequent words of the genres we examined and the bias), a number of nodes in the hidden layer that can be chosen by the user, and a single output which defines whether the observation belongs to the *romantic* or *murder* genre. The goal of the network is therefore to be able to correctly classify an observation.

Starting from the model-training phase, we first initialised the weight matrices and the bias vectors. The bias neuron is a special node that is added to each layer in the neural network and simply stores the value of 1, initially: this makes it possible to move or "translate" the activation function left or right on the graph. We haven't initialized all the parameters to zero because doing so will lead to the same value for all the gradients and, on each iteration, the output would be the same, thereby hampering the learning capability of the algorithm. Therefore, we have randomly initialized the parameters to values between -1 and 1.

Once the weights are initialized, the training phase is performed and repeated as many times as the number of epochs, which, together with the learning rate, the number of hidden neurons, and the program execution mode (serial or parallel), will be chosen by the user at the start of the execution. Furthermore, if the parallel mode is chosen, the number of threads can also be set.

The model training phase essentially consists of two parts: *forward propagation* and *back propagation*.

Forward propagation refers to the computation and storage of intermediate variables for a neural network in order from the input layer to the output layer. We calculate the output by first multiplying each input by the corresponding weight of each neuron and then feeding each neuron output to the activation function (sigmoid). We do this for each neuron in each hidden layer, including the output layer.

```
for (j = 1; j <= numHid; j++) {    // Forward Propagation
    Sum_Hidden[sample][j] = WeightIH[0][j];
    for (i = 1; i <= numIn; i++) {
            Sum_Hidden[sample][j] += allData[sample].feat[i] *
            WeightIH[i][j];
    }
Hidden_Activation[sample][j] = 1.0 / (1.0 +
    exp(-Sum_Hidden[sample][j]));  // Sigmoidal Hidden
}
```

As we can see from the code above, the result of the sum of the products between the weights and the inputs is stored in the *Sum_Hidden* matrix, and then the sigmoid function is applied, which stores the results in the *Hidden* matrix. The same procedure is repeated when passing from the hidden layer to the output as well. In this case, the results of the various operations are stored in the *Sum_Output* and *Output* matrices, respectively.

After this, we defined a cost function that measures how well our neural network performs. We have used binary cross-entropy as the cost function, which uses the log-likelihood method to estimate its error. This cost function is convex, however the neural network might happen to get stuck on a local minimum and, therefore, is not guaranteed to find the optimal parameters. So we have to use gradient-based learning:

```
lossError -= (allData[sample].out[0] * log(Output[sample][0]) + 1.0 -
allData[sample].out[0]) * log(1.0 - Output[sample][0]));
```

Back propagation refers to the technique of calculating the gradient of the neural-network parameters. The method traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus. The algorithm stores any required partial derivatives while calculating the gradient with respect to the parameters.

```
for (j = 1; j <= numHid; j++) { // Back Propagation
    Partial_DeltaH[j] = 0.0;
    Partial_DeltaH[j] += WeightHO[j][0] * Delta_Output[sample][0];
    Delta_Hidden[sample][j] = Partial_DeltaH[j] *
            Hidden_Activation[sample][j] * (1.0 -
            Hidden_Activation[sample][j]);
}
```

Then we update the weights and bias, we multiply the partial derivatives by the learning rate and subtract the results from the weights and bias.

After training the model, the weights obtained from training are used on the validation data in order to try and predict the correct class on data not seen in the training phase. This way it's possible to measure the accuracy of the model reliably.

V. A-priori Study of Available Parallelism

In our serial code, it makes sense to parallelise and get satisfactory results only on the main part of the program: the neural network training phase.

The other tasks executed in the serial code, such as allocating memory, reading the data, creating the data structures and so forth, are not influential in terms of time; in fact, they are almost instantaneous. During network training, the whole forward-propagation part (therefore the weight-calculation process and the activation-function calculation) and the whole back-propagation part (therefore the partial-derivatives computation and the weights update) can be performed in parallel without waiting for the predefined order of observations. Before proceeding to parallelise the serial code, we calculated the theoretical speed up. In order to do this, we used *Amdahl's law*: *(1 / (1-P) + P / S)*, which states that *the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement*[2].

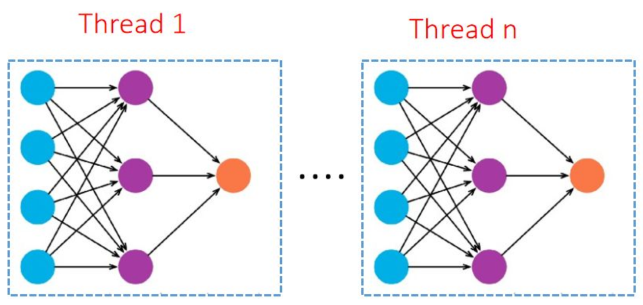
We then measured the total time of execution of the neural network and the total time of the section we wanted to parallelise in the serial code. By calculating the ratio between the two measures we obtained a percentage of 99% of parallelizable code. This should not be surprising, since the part we chose to parallelise represents almost the totality of the program; the other operations, as mentioned above, are irrelevant towards the running time.

Theoretical speed up with different numbers of cores:

1 core	-	1
2 cores	-	1.98
4 cores	-	3.88
8 cores	-	7.47
16 cores	-	13.91
24 cores	-	19.51

VI. PARALLEL IMPLEMENTATION

The most effective implementation we have found is based on the idea that each thread should handle a different observation; once finished, it selects another one and so on.



To do this, we focussed on the *for* loop that performs forward-propagation and back-propagation and, above the *for* loop that concerns the observations, we used *#pragma omp parallel for private(j, i, Partial_DeltaH) shared(Sum_Hidden, Hidden_Activation, Sum_Output, Delta_Hidden)*, setting variables *i*, *j* and *Partial_DeltaH* as private, in such a way as not to create conflicts between the various threads during the execution of the program. Variables *Sum_Hidden*, *Hidden_Activation*, *Sum_Output*, and *Delta_Hidden*, on the other hand, are shared.

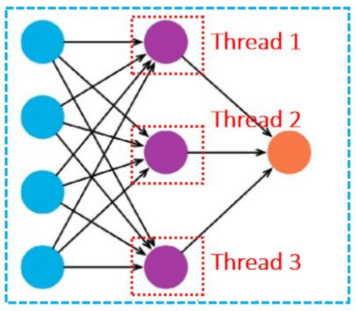
In order to have each thread work on a single observation, the section that focuses on updating the weights, above the *for* loop containing the observations, was parallelised as well through *#pragma omp parallel for private(j, i) shared(Delta_WeightIH, Delta_WeightHO, WeightIH, WeightHO)*. Even in this case, *i* and *j* were made private to avoid conflicts.

```
#pragma omp parallel for private(j, i, Partial_DeltaH)
shared(Sum_Hidden, Hidden_Activation, Sum_Output, Delta_Hidden)
for (int sample = 0; sample < numSample; sample++) {
    for (j = 1; j <= numHid; j++) {
            Sum_Hidden[sample][j] = WeightIH[0][j];
            for (i = 1; i <= numIn; i++) {
                    Sum_Hidden[sample][j] += allData[sample].feat[i]
                    * WeightIH[i][j];
            }
    Hidden_Activation[sample][j] = 1.0 / (1.0 +
    exp(-Sum_Hidden[sample][j]));
}

#pragma omp parallel for private(j, i) shared(Delta_WeightIH,
Delta_WeightHO, WeightIH, WeightHO)
for (int sample = 0; sample < numSample; sample++) {
    for (i = 0; i <= numIn; i++) {
            for (j = 1; j <= numHid; j++) {
                    Delta_WeightIH[i][j] += allData[sample].feat[i] *
                    Delta_Hidden[sample][j];
                    WeightIH[i][j] += learningRate *
                    Delta_WeightIH[i][j] / numSample;
                    Delta_WeightIH[i][j] = 0.0;
            }
    }
    for (j = 0; j <= numHid; j++) {
            Delta_WeightHO[j][0]+= Hidden_Activation[sample][j] *
            Delta_Output[sample][0];
            WeightHO[j][0] += learningRate * Delta_WeightHO[j][0] /
            numSample;
            Delta_WeightHO[j][0] = 0.0;
    }
}
```

[*Parallelised code for the first implementation*]

Another implementation is possible: instead of having each thread manage an observation we can make sure that, for each observation, multiple threads are executed, each of them managing a node, as shown in the figure below.



In this type of parallelisation we focus on the cycle of observations, parallelising the process of forward- and back-propagation and distributing the work of the neurons in the hidden layer to the various threads. In order to do this, the following instructions have been inserted: *#pragma omp parallel for private (j, i)*, during feed-forward, and *#pragma omp parallel for private (j, i, Partial_DeltaH)*, during back-propagation. Furthermore, in the weights-update phase, *#pragma omp parallel for private (j, i)* has been inserted to update the weights that go from the input to the hidden layer, and *#pragma omp parallel for private (j)* was used for the update of the weights that go from the hidden layer to the output. In all these instructions the variables *j* and *i* have been set as private to avoid conflicts.

```
for (int sample = 0; sample < numSample; sample++) {
    #pragma omp parallel for private(j, i)
    for (j = 1; j <= numHid; j++) {
            Sum_Hidden[sample][j] = WeightIH[0][j];
            for (i = 1; i <= numIn; i++) {
                        Sum_Hidden[sample][j]  += allData[sample].feat[i]
                        * WeightIH[i][j];
            }
            Hidden_Activation[sample][j] = 1.0 / (1.0 +
            exp(-Sum_Hidden[sample][j]));
    }

    #pragma omp parallel for private(j, i, Partial_DeltaH)
    for(j = 1; j <= numHid; j++) { // BACK PROPAGATION
            Partial_DeltaH[j] = 0.0;
            Partial_DeltaH[j] += WeightHO[j][0] *
                        Delta_Output[sample][0];
            Delta_Hidden[sample][j] = Partial_DeltaH[j] *
                        Hidden_Activation[sample][j] * (1.0 -
                        Hidden_Activation[sample][j]) ;
    }

    #pragma omp parallel for private(j, i)
    for (i = 0; i <= numIn; i++) { // Weights update
            for (j = 1; j <= numHid; j++) {
                        Delta_WeightIH[i][j]  += allData[sample].feat[i]  *
                                    Delta_Hidden[sample][j];
                        WeightIH[i][j] += learningRate *
                                    Delta_WeightIH[i][j] / numSample;
                        Delta_WeightIH[i][j] = 0.0;
            }
    }

    #pragma omp parallel for private(j, i)
    for (i = 0; i <= numIn; i++) { // Weights update
            for (j = 1; j <= numHid; j++) {
                        Delta_WeightIH[i][j]  += allData[sample].feat[i]  *
                                    Delta_Hidden[sample][j];
                        WeightIH[i][j] += learningRate *
                                    Delta_WeightIH[i][j] / numSample;
                        Delta_WeightIH[i][j] = 0.0;
            }
    }
}
```

[*Parallelised code for the second implementation*]

However, this type of parallelisation did not bring great advantages in terms of execution speed compared to the serial algorithm, so in the end it was discarded.

VII. Testing and debugging

Several tests were performed on the serial algorithm by varying the number of observations, the number of neurons, and the number of epochs. These same tests were also reproduced with the parallelised code, which gave us the exact same results in terms of model accuracy, usually around 67% in validation for a sufficient number of epochs.

In our project, the 1000 most-frequent words taken from the movie plots have been chosen as the number of features, but this number can be easily modified by varying the value of a variable in the *WordsExtraction.ipynb* file.

In the parallelised code, we used the OpenMP function "*omp_get_thread_num()*" and placed various *printf*s in the areas of interest in order to see if the work was distributed correctly among the threads.

VIII. Performance analysis

The performances on the neural network were measured by taking into account multiple variables. The first case consisted in measuring the impact on time when considering different amounts of observations; the second case focussed on visualising and understanding the differences when using a different amount of RAM and a different number of cores, while keeping the number of observations fixed. We used 5 hidden neurons, a value of 0.3 for the learning rate and 3500 epochs for all the following tests.

<u>Case 1</u>
The neural network was tested on a single machine over a variable number of observations: 1000, 3000 and 6000. Both the serial and parallel implementations were examined, with different numbers of threads in the second case. For these tests, a virtual machine running on a home computer was used, with 8GB of RAM and 4 cores.

The performances are as expected: both in the parallel and in the serial cases, the machine takes more time when dealing with a larger amount of observations. As can be seen in *Figure 1* below, the training times are 2.6, 8.3 and 15.0 minutes, respectively, when considering 1000, 3000 and 6000 observations. The times of the best parallel cases are, of course, shorter, and equal to 0.9, 2.6 and 4.9 minutes (*Figure 2*). This means that the parallel case can be over 3 times faster than the serial one, which is close to the theoretical speedup calculated through Amdahl's law, although not as good (as expected).

As we can see, doubling the amount of observations doesn't necessarily result in double the training time. The efficiency increases with the number of observations.

The optimal number of threads oscillates between 15 and 20: fewer threads means that further speedups could be obtained by sharing the workload between more threads, while using 25 or 30 of them is generally less effective because of the added overhead in managing them through a relatively small number of cores.

<u>Case 2</u>
For this second experiment, the neural network was tested on different virtual machines residing on the Google Cloud platform. Three cases were examined: 32GB RAM and 8 cores, 64GB RAM and 16 cores, 96GB RAM and 24 cores.

The best parallel cases, i.e. those obtained when choosing the number of threads that yields the fastest time measure, are quite similar among the three machines. As can be seen from *Figure 3*, the 8-core case takes about 20 seconds more to finish training, while the other two machines display almost the same behaviour. The 24-cores machine is ultimately the fastest, as expected from the selected level of parallelism and architectures.

The serial cases are all slower than the parallel ones, between 8 and 12 minutes. The 16-cores and 24-cores machines have the exact same behaviour, while the 8-cores solution is the fastest (8.4 minutes, *Figure 4*).

IX. Conclusions

At the end of our work, we can say that we have learned how a neural network works, and how to improve its performance by parallelising the work done by the network. Parallel programming leads to noticeable improvements in the performance, so the techniques hereby described should always be applied when dealing with this kind of problems.

X. Sources

1. https://www.kaggle.com/cryptexcode/mpst-movie-plot-synopses-with-tags

2. https://en.wikipedia.org/wiki/Amdahl%27s_law

FIGURE 1
Training time
Virtual machine - 8GB RAM, 4 cores

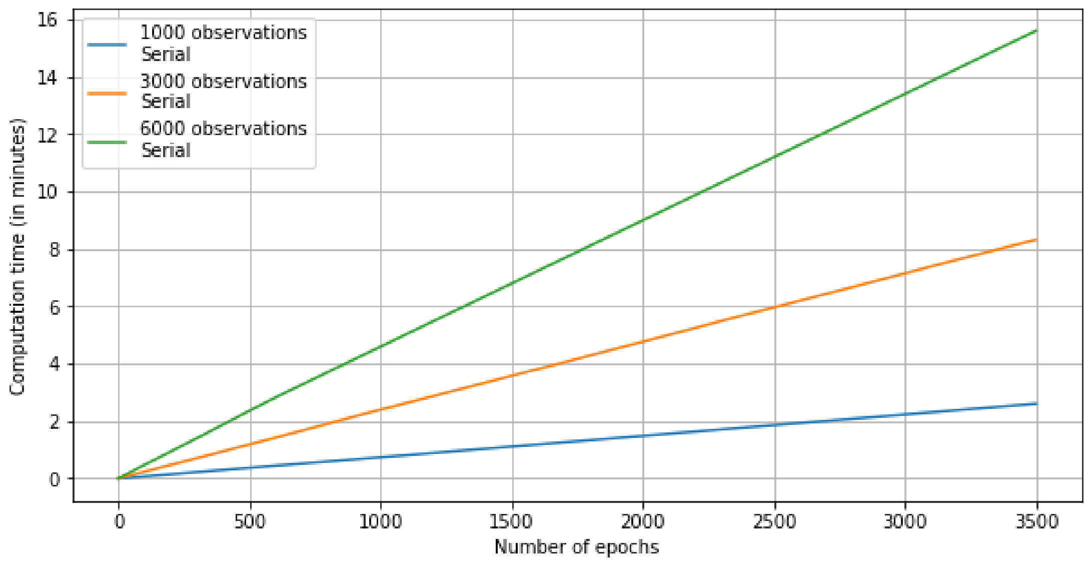


FIGURE 2
Training time
Virtual machine - 8GB RAM, 4 cores

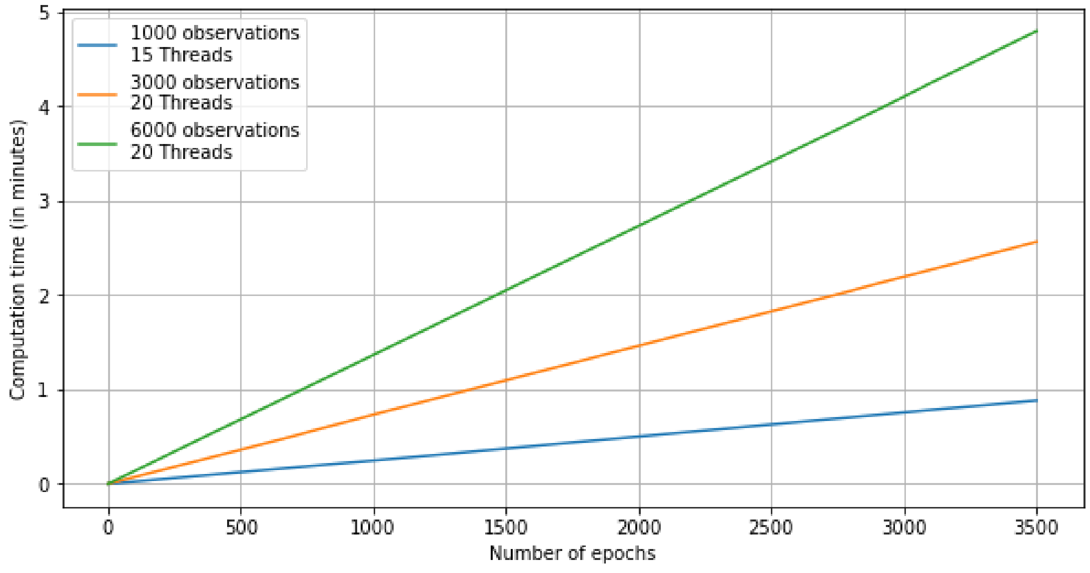


FIGURE 3
Training time
Best parallel times

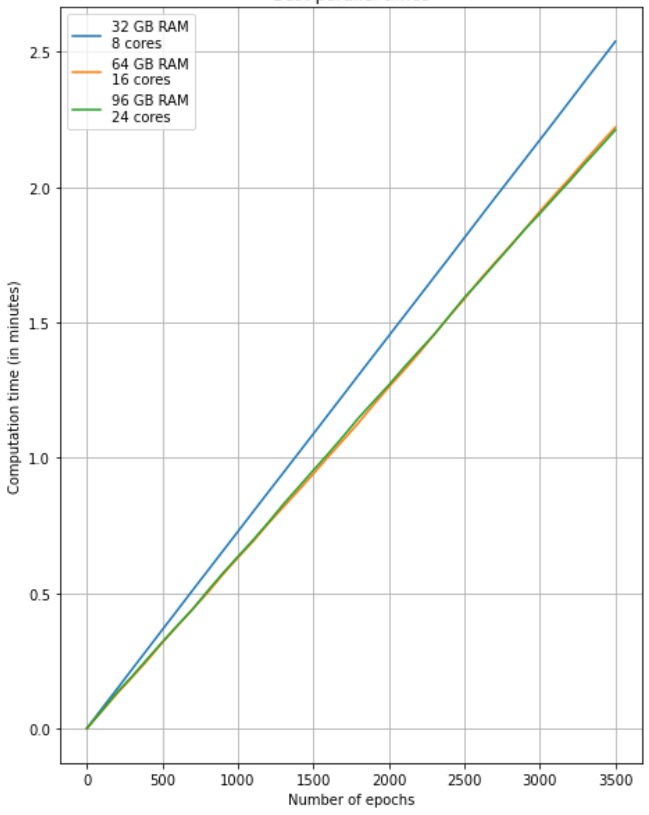
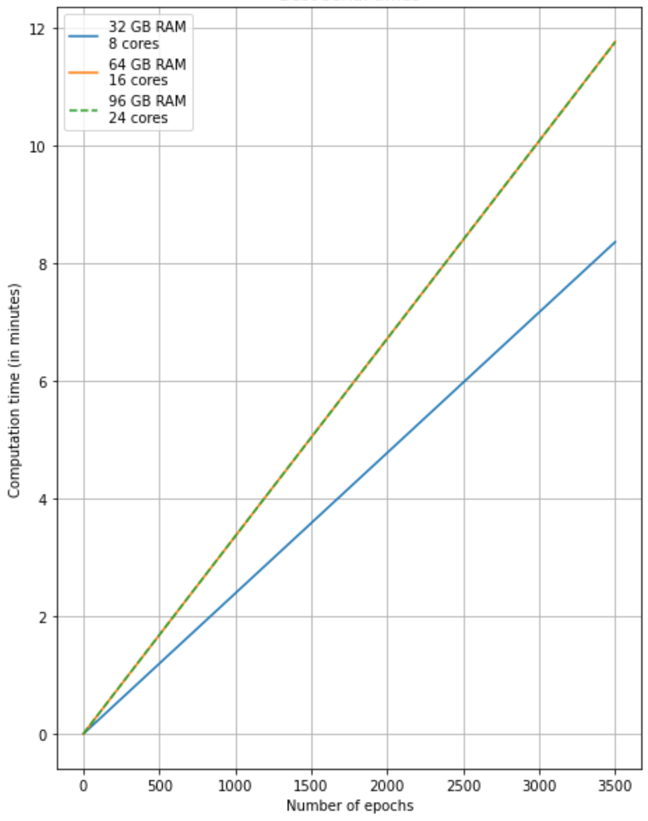
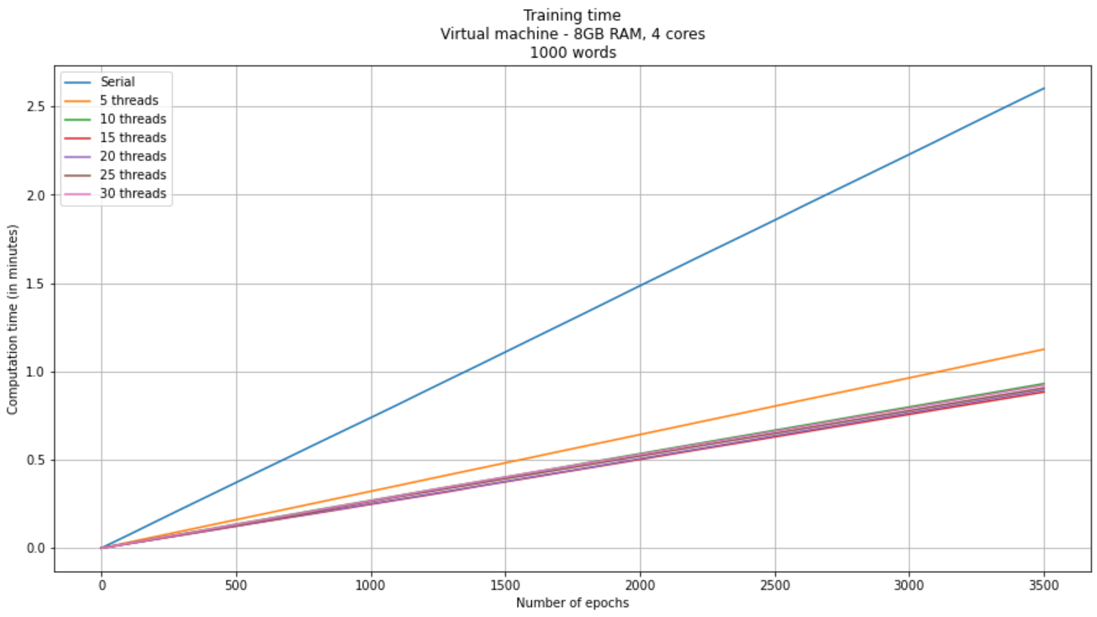
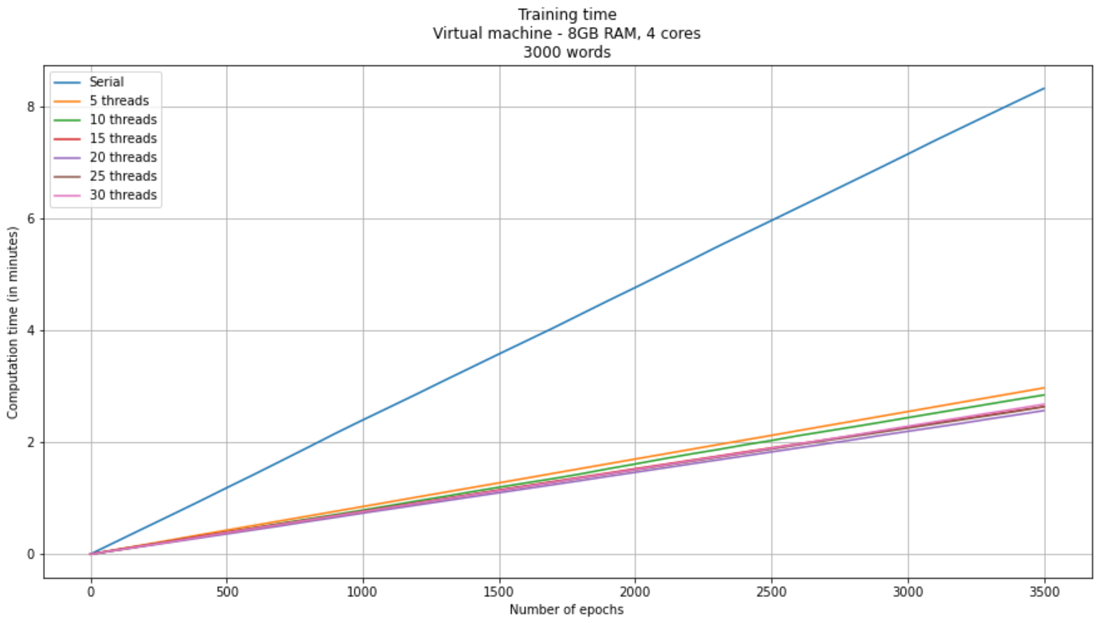


FIGURE 4
Training time
Best serial times

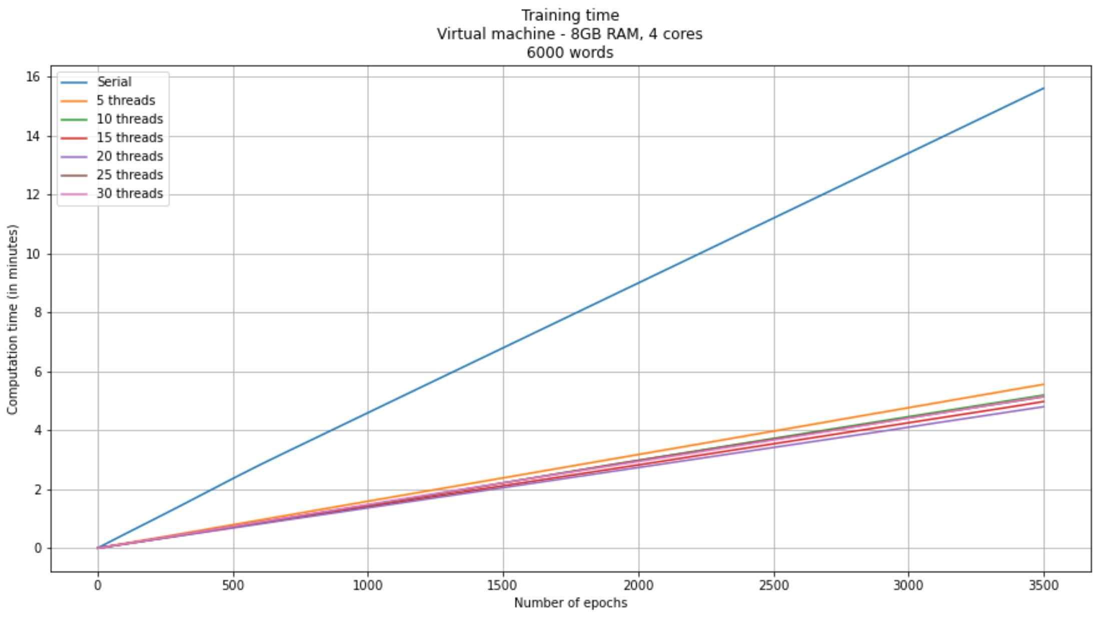




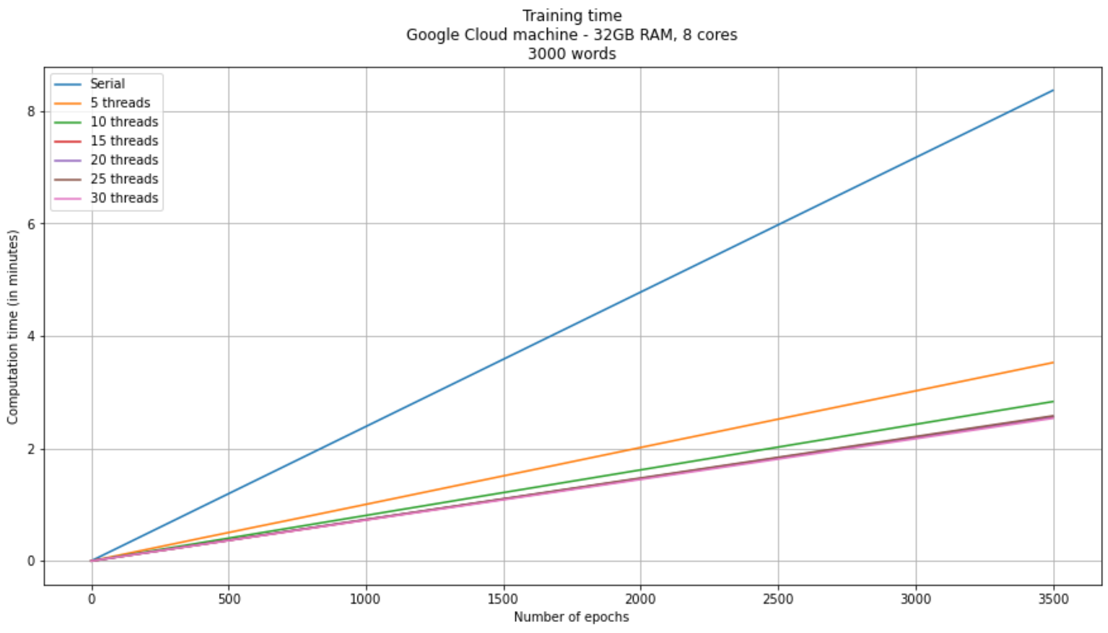
Training time
Virtual machine - 8GB RAM, 4 cores
1000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs
0.0
0.5
1.0
1.5
2.0
2.5
0
500
1000
1500
2000
2500
3000
3500



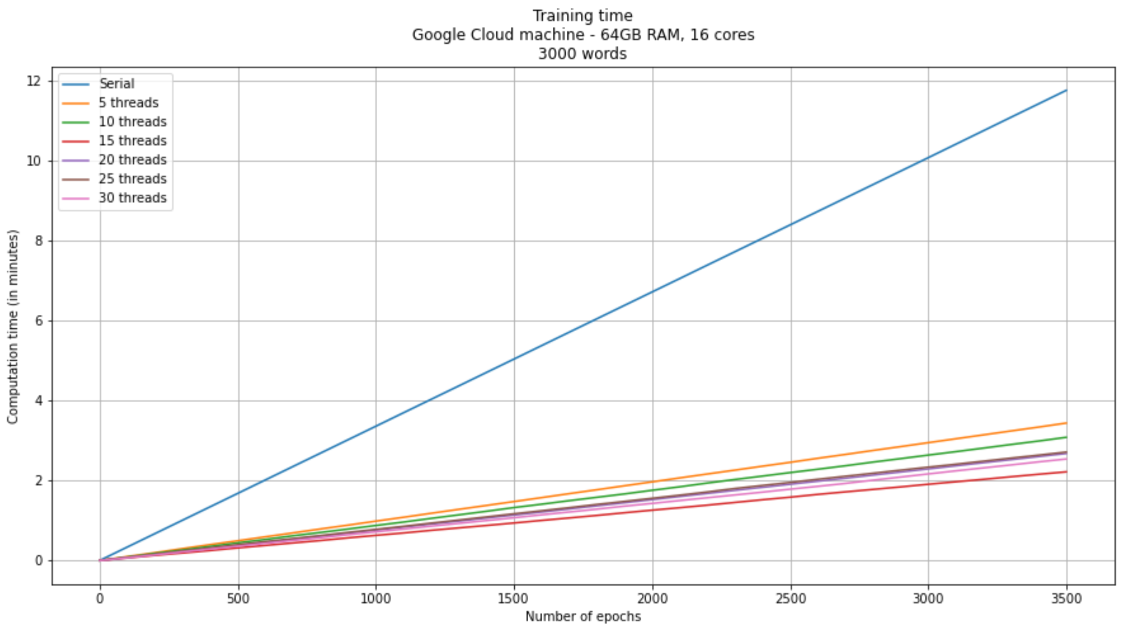
Training time
Virtual machine - 8GB RAM, 4 cores
3000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs
0
2
4
6
8
0
500
1000
1500
2000
2500
3000
3500



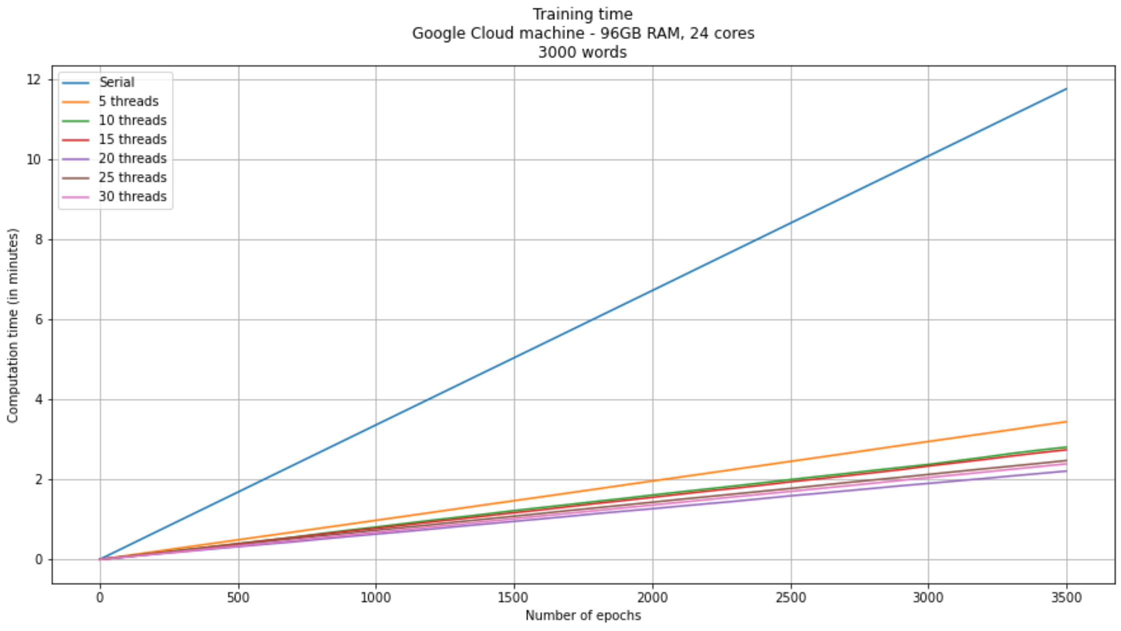
Training time
Virtual machine - 8GB RAM, 4 cores
6000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000
2500
3000
3500



Training time
Google Cloud machine - 32GB RAM, 8 cores
3000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs



Training time
Google Cloud machine - 64GB RAM, 16 cores
3000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs



Training time
Google Cloud machine - 96GB RAM, 24 cores
3000 words
Serial
5 threads
10 threads
15 threads
20 threads
25 threads
30 threads
Computation time (in minutes)
Number of epochs