Parallelized Multilayer Perceptron

Joel Cerqueira Ponte

École nationale supérieure d'informatique et de mathématiques appliquées de Grenoble

February 14, 2018

Multilayer perceptrons (MLP) are a class of feedforward artificial neural networks. Nordström and Svensson (1992) have proposed 6 different types of parallelism that can be implemented in the training process of MLPs. In this work, we implemented and compared a sequential version and two different types of parallelism using OpenMP: node parallelism (where different nodes are assigned to different threads) and data parallelism (where different parts of the training data are assigned to different threads). As expected, we have found data parallelism to be more effective, being able to speed up training to up to 4 times in the tests.

1 Design Methodology

1.1 Introduction to the problem

The training process of a MLP is composed of two main parts: forward propagation and backpropagation. In the forward propagation, the inputs go through a series of transformations to generate the outputs. These transformations are composed of a linear combination of the previous layer plus a bias followed by a application of a nonlinear activation function for every layer following the input layer. Equations (1)-(5) describe the forward propagation process for a MLP with one hidden layer, which is the case considered in this work. Note that X represents the inputs, $A^{[2]}$ is our estimation of the outputs based on the weights $W^{[k]}$ and $b^{[k]}$, and $g^{[k]}$ is an activation function such as $sigmoid$, $\tanh$ or $reLU$.

$$A^{[0]} = X \tag{1}$$
$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]} \tag{2}$$
$$A^{[1]} = g^{[1]}(Z^{[1]}) \tag{3}$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \tag{4}$$
$$A^{[2]} = g^{[2]}(Z^{[2]}) \tag{5}$$

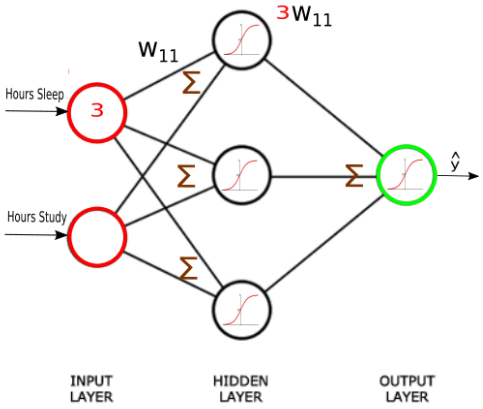

Figure 1: *Illustration of a MLP*

In the backpropagation part we calculate the derivatives of the loss function with respect to the weights. We need them so we can perform gradient descend and optimize the loss. Since the weights are related to the loss through a successive application of functions, we use the chain rule to calculate the derivatives of the loss with respect to them. Derivatives of weights more to "the left" of the network (see Figure 1) are dependent on derivatives more to "the right" of it, so we start calculating them in the opposite direction of the forward propagation. We wont go through the

derivation of the backpropagation equations here, but if we set $g^{[1]}$ to be $\tanh$ and $g^{[2]}$ to be $sigmoid$ we get the following equations:

$$dz^{[2]} = a^{[2]} - Y \tag{6}$$
$$dW^{[2]} = \frac{1}{m} dz^{[2]} A^{[1]T} \tag{7}$$
$$db^{[2]} = dz^{[2]} \tag{8}$$
$$dz^{[1]} = W^{[2]T} dz^{[2]} * (1 - a^{[1]} * a^{[1]}) \tag{9}$$
$$dW^{[1]} = dz^{[1]} x^T \tag{10}$$
$$db^{[1]} = dz^{[1]} \tag{11}$$

where $dK = \frac{d\mathcal{L}}{dk}$, $\mathcal{L}$ being the loss and the lowercase means we are considering one single training example in the equations.

1.2 Parallelism

We began by implementing the sequential version. Then, two kinds of parallelism were chosen inspired the examples given by Nordström and Svensson, 1992. In multilayer perceptrons, each node calculates a linear combination of its inputs. This allows for node parallelism, which means we assign specific nodes to specific threads. This is equivalent to parallelizing matrix multiplications, giving different rows to different threads. A drawback of this method is that it creates multiple forks and joins, which compromises the performance.

In data parallelism, we assign different parts of the data to each thread. All the computations are done normally, but when gradient descent is performed we have one gradient for each thread and we must sum them.

1.2.1 Node parallelism

For this implementation, we separated equations (2)-(11) in 6 parts. Each part was parallelized separately, meaning that we ended up with 6 forks and joins in the forward an backpropagation. The forks were made, in terms of the equations, for: (2)+(3) / (4)+(5) / (6) / (7)+(8) / (9) / (10) + (11). To implement this, we just wrapped the for loops used to calculate the expressions with `#pragma omp parallel`, setting the appropriate private variables.

This is obviously not an optimal solution, as it creates a lot of forks and joins for each $epoch$ (one epoch = one pass through the training set). There are better ways to implement node parallelism, but we decided to rather focus on implementing data parallelism.

1.2.2 Data parallelism

For the data parallelism implementation, we chose to create a big parallel region that contains all the calculations for the forward propagation, backpropagation and gradient descent. We declared inside this region the forward propagation variables $a^{[1]}$, $a^{[2]}$, $z^{[1]}$, $z^{[2]}$ because they are all dependent on which chunk of the data is being considered in the thread. We also declared here all the backpropagation variables, because they are all dependent on the data. The data itself and the weights are declared outside this region because all the threads must see them equally.

The only work that is done by the master is to set the loss at the beginning of each epoch to zero. This is only necessary if we want to see the decrease of the loss over time.

The back and forward propagations codes are very similar to their sequential versions. The only difference is in the dimensions of the data, which we must take into consideration that it is divided to each thread. To calculate the loss, we store partial losses it in a local loss variable, which will later be summed.

The last step in the epoch is to do gradient descend. We first create a barrier to ensure that no weights are updated before the forward and back propagations end for all threads. Then, there is a critical region that ensures that the weights updates will be calculated one thread at a time. In this critical region we also sum up the local losses to get the total loss in the epoch.

To print the loss correctly (if we choose to do so), we need to include another barrier after the critical region to make sure that it is being printed after all local losses were added together.

2 Results/Data Analysis

In this section we compare the runtime of the sequential version with the two parallel versions in four dimensions: by varying the size of the data, the size of the hidden layer, the number of epochs and the number of threads. We compared the most optimized versions, meaning that the loss was not printed.

All results were run in my personal laptop: an i7 4710HQ, 12Gb DDR3, 500Gb SSD Samsung EVO 850, on a Linux Mint 18.3. The dataset used was the same for all tests: an oversampling of the Breast Cancer Wisconsin Data Set.

In Figure 2 we can see the change of the runtime with respect to the size of the training set. All of them grow linearly, but it is clear that the data parallelism implementation is far superior to the other two, which have comparable slopes.

Figure 3 shows how the runtime behaves in a change of the hidden layer size. We again see an approximately linear relation, but this time the node parallelism shows a more significant performance compared to the sequential version. This result makes sense, as we are parallelizing the nodes and to increase the size of the hidden layer is to increase the number of nodes.

In Figure 4 we see the relation of the run time with the number of threads used. Using 8 threads did not improve a lot in comparison to 4 (and was even worse for node parallelism in this experiment). For the data

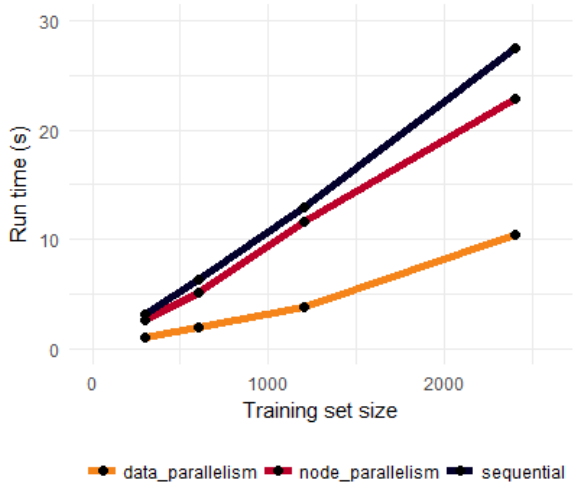

Figure 2: *Relation between time and size of the training data. The number of epochs, the size of hidden layer and the number of threads were fixed to, respectively, 2000, 5 and 8.*

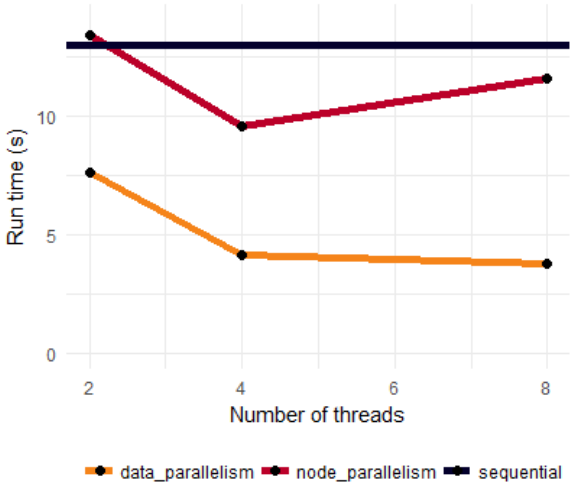

Figure 4: *Relation between time and number of threads. The number of epochs, the size of the training set and the size of the hidden layer were fixed to, respectively, 2000, 1200 and 5.*

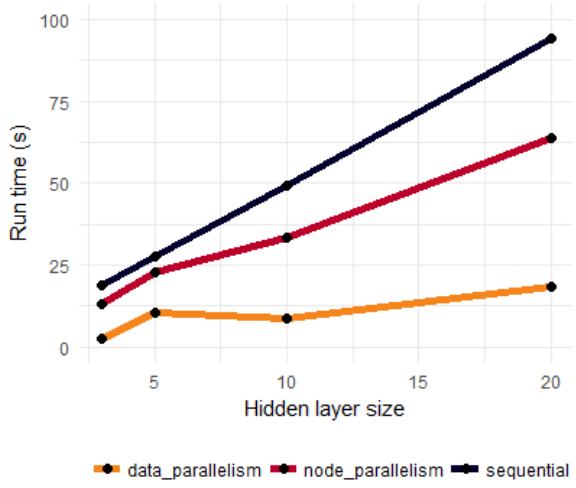

Figure 3: *Relation between time and size of the hidden layer. The number of epochs, the size of the training set and the number of threads were fixed to, respectively, 2000, 2400 and 8.*

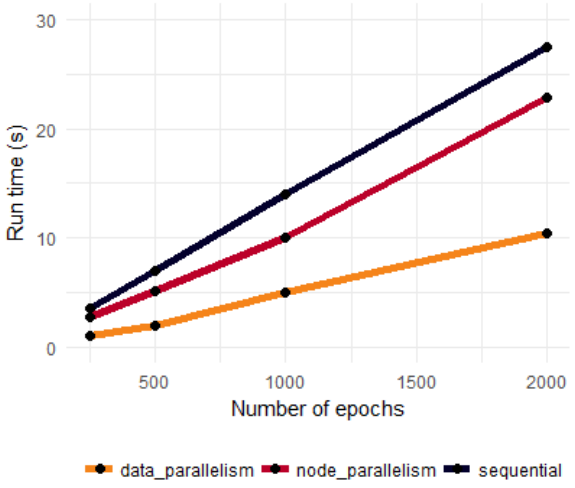

Figure 5: *Relation between time and quantity of epochs. The size of the hidden layer, the size of the training set and the number of threads were fixed to, respectively, 5, 2000 and 8.*

parallelism results, it could be the case that the training data was not big enough for us to see a big improvement from 4 to 8 threads. For the node parallelism, maybe a MLP with a bigger hidden layer could also benefit more from having more threads.

Finally, in Figure 5 we see the relation between run time and the number of epochs. Here, we see a pattern similar to Figures 2 and 3. Again, the data parallelism implementation stands out.

3 Conclusion

In this work we proposed to implement a sequential version of one-layer multilayer perceptron and parallelize it through two different paradigms. We have tested and comproved that our implementation of node parallelism was far from optimal, but it gets more valuable as the size of the hidden layer increases. The data parallelism paradigm, as expected, was the most successful implementation in all scenarios, speeding up the code up to 4 times.

Bibliography

Nordström, Tomas and Bertil Svensson (1992). "Using and Designing Massively Parallel Computers for Artificial Neural Networks". In: *J. Parallel Distrib. Comput.* 14.3, pp. 260–285. ISSN: 0743-7315. DOI: 10.1016/0743-7315(92)90068-X. URL: https://doi.org/10.1016/0743-7315(92)90068-X.