UIO

FYS-STK4155
Project 3

December 16th, 2020

Kjell R. Christensen

ABSTRACT

The growing processing power that has emerged withing the PC architecture like CPUs, GPUs, distributed sub-processing and on-demand cloud processing, opens for more parallelism and distributed ML processing. The project will run instances of the same feed forward neural network (FFNN) in parallel, based on the breast cancer dataset in Sklearn. The different instances will share input data but will run with distinct configurations on biases and weights. The parallel instances will synchronize and average out the cost, in timely intervals, for faster convergence.

INTRODUCTION

Parallel processing has been a growing architecture for many decades, not only on the chip side, with increased on-ship multi-core concepts, but just as impressive on the dedicated graphical processor - GPU's. The latter are highly optimized chips aimed to run thousands of graphical operations simultaneously. It has been shown that GPUs with highly optimized architecture for graphical manipulation, also are well suited for matrix manipulations, which is central for most of the ML algorithms.

In addition to the physical chip power, more software and protocols are emerging where processing power can be distributed in networks, for sub-tasks that needs processing power, way beyond the capacity of a single machine.

As a first step, this project will align FFNN in parallel instances (1..n) on one physical machine. It will explore the processing concept of running multithreaded instances of the network. This is a "light" version of sub-processing with shared physical memory within the scope of the "mother" process.

Multithreading will be compared with multi processes with its own dedicated memory space, dedicated Python interpreter, and options for further threading.

By enabling network parallelism, we will show that 2 or more instances, with independent randomly initialized weights will converge faster, and increase prediction and accuracy even more, than single instance.

All parallel instances will share the common input data (Xi), but will have their unique set of Hyperparameters (Hi), that will be adjusted and tuned independently.

The project will show that there is a clear correlation between the number of instances running in parallel and the time for the networks to converge.

Three cases that will be analyzed:

1) The instances will run isolated, with distinct hyperparameters (Hi) through all layers, expect for the output. The weights for all instances will be averaged-out for the last layer and used for backpropagation for all instances.

2) For this case, the weights are selected based of the min|y^ - y| difference between the y-hat and y. In other words – best predictions will influence more on weights setting.

3) We will now increase the award/penalty through the cross-entropy by multiplying a constant ∂ to the activation function.

- **Structure**

The analyses are done in a msCode environment and is using standard libraries as specified in the *.py heading. All “imports” are based on standard libraries, commonly used for ML, and routines and code generated for this project, is located only in the python file.

We have one data set, the breast cancer loaded from Sklearn.

The program is modularized, based on separate tasks, and we have modularized in such a way that the program itself, can run one, several or all modules, in one go for analyses. A separate test environment is, hence, not needed.

The reader can at any time reproduce any test-results or plots, by activating the selected part in the “Test”() module.

All plots are located in the ./Plots folder and they are named, based on function, parameters and methods. The naming convention is also indicating the size of the sample data, to ease the setup, and re-run the tests.

The MacBook Pro have 4 cores and the number of instances for the FFNN will be multiple of the number of cores.

More specific instructions are listed in the readme.md file in the project folder.

METHOD

The "standard" FFNN algorithm will be expanded to incorporate the initiation of **i-instances** of the network, for **i** in **(2, 1*C, 2*C, 3*C…)**, where **C=4** (number of CPU cores)

The network will be expanded to include the module NetworkX. This will enable plotting snapshots of the weights **Wi**, of one or several of the parallel instances running, at selected epochs. This will be useful to better optimize the hyperparameters **Hi**, during the training part.
There will be added a plotting function that will compare the weights in a set of paired instances, like a "heat map" to see the difference in weight distribution, during conversion. This is only possible for networks that are configures with the exact same topology.

Bases on NN theory, the weights ("memory") of the network will not converge to "one" specific weight "state", even though the precision for the classification converge. We expect to show that different weight states, will produce similar prediction results.

All instances of the network will share the common input data (observations) **X** with corresponding **Y** in global structures (read only). All other variables will be instance specific **Hi**, and some of the variables can be changed during run-time.

The project has invoked three different methods for instantiation of the FFNN Class.

1) Standard instantiation as implemented in std Python. A call with standard passing of hyperparameters and return of test-results.

2) For threading we have used the **treading** package which opens up for lighter sub-processing and enables all the threads to be ensembled in the threads.join() routine for synchronization of all threads to complete. The control is not passed back to "mother" process before all threads have completed. The Pipe function has been used to pass any return value back from the thread, for post processing (as for the plots in /Plots)

3) Multiprocessing is contained in the package **multiprocessing**, where process and freeze_support is needed to parallelize. Also, here the Pipe function is used to hold any return value. Only after all sub-processes has finalized in process.join(), can return values be processed further in the main process.

The network has added functionality to synchronize loss values between instances. The logic has been added to the self.fit() function in the class, and will be activated at timely epoch intervals.

Schematic overview of the logic:
(This is a small state-machine implementation, using the Glogal Lock() functionality and will write and read the new calculated Global loss, implemented and stored using shared global variable, and then proceed to the back propagation in its own thread.

-> Finalize calculation of Instance.loss()
    -> while not read new Global.loss()
        -> request to aquire Global.lock()
        -> if Aquired(), write Instance.loss() to global variable
    -> GlobalLoack.wait() until all instances has written its instance loss to global variable
    -> then calculate new global loss
  -> request to aquire Global.lock()
    -> read the new calculated Global Loss and assign it to your instance loss variable,
  ->exit state machine
-> proceed to calculated back propagation with new loss value

RESULTS & DISCUSSIONS

The single instance of the FFNN implementation for breast cancer shows good performance and are suited to run on a laptop of any kind. For some cases the loss function gave **Inf** and **NaN** returns on the loss function and convergence was not possible to reach. As far as I could see, this was only due to the randomized generation of weights and biases.

Based on reference, the balance in the data set is about 50/50, amongst true and false **y's** and hence, there should be no reason to balance the weights one way or the other for the test dataset. It took quite some time to distribute random weights between [-1,1] and also to conclude on recommendation in documentation of a multiplier of 1/SQR(observations (n)). Reaching this conclusion, made conversion must faster an easier and all epoch produces loss values within boundaries of processing, and all testcases avoided the vanishing loss problem and converged. The best fitted model was reached between 3000-4000 epochs. Most runs with more than 5000 generated over fitting, and hench not optimal accuracy.

Det routine SynchThreadLossFunction, will synchronize the loss values for all running instances. The first thread entering the instance of the routine will fetch the GloblaLock.aquire() and write its version for the instance loss to a global variable. All other instances will have to queue up until the write is finalized and GlobalLock.notify_all() broadcast is issued to all waiting threads/process. One waiting thread will be delegated the resources and write. This runs until all threads have written its instance loss to the variable.

Much time has been spent on three different classes providing locking functionality:

1) The primitive Lock Object is in one of two states aquire() and release(). This class has no wait() function, and hence is not suitable to implement the small state machine, where wait() is essential for waiting until all threads has written their local loss value to the global variable. Actions on an unobtained lock will result in runtime error.

2) The RLock Object, adds nested or recursive functionality. Only when the thread releases the lock in the outer-level will the lock be released, and assigned to waiting threads “hanging” in the aquire() function.

3) The Condition Object, adds more functionality to the synchronized locking object. In addition to aquire() and release() functionality, it can set the thread in a wait() state. Wait() will automatically release the lock and pass it to waiting threads in aquire(). The conditional object has a “smooth” and resource efficient way to wake up one or several threads by pre-issuing the statement notify() or notify_all(). The latter two, will give a notice to the lock-handler to wake-up and make threads ready, for a smooth transition, when release() is done by the current active thread.

After testing the three different locking objects, I found the Conditional object to be most suitable for the implementation. Even though you want maximum through-put and let the threads run as fast as possible, there is no guaranty for equal share of resources and one thread can run an extra epoch, even before another thread has finalized the previous one. The latter forces the need for thread to wait() for exit and also in the aquire() statement on its way in for synchronization on the global variables.

There were to many observations that got stuck within the Aquire.Lock() and Release.Lock(). It became quite hard to debug if this was due, only to logical errors in the small “state-machine”, or if it was influenced by instability between the GIL’s and its context switching within the threads and sub-processes.

Due to the Python´s Global Interpreter Lock’s (GIL) implementations, the findings for interchange of hyperparameters, like cost averaging within all instances of the network, are nonconclusive.

CONCLUSIONS & EXPECTATIONS

I was surprised to see that multi-threading managed to outperform multiprocessing. It's clear that threading is a concept of sub-processing "light", with shared memory within its "mother" process, but should have faced limitations due to Pythons GIL implementation and limitations. I expect that this will be more of a problem as the architecture of the network grows deeper, and also for larger datasets, than what is presented for the breast cancer in Sklearn.

Synchronization of data within several threads might have an upper-hand, due to its shared process memory, and it's expected that shared objects like structs and pipes, controlled by locks, can be more optimized within the same memory space. However, this has to be tested and confirmed in the future.

I experienced quite some "hangs" in the system, that I expect to be a mis-use of locks and in particular the aquire() (wait queue) and wait() functionality. At time, it got stuck and also timed out, resulting in the state machine to get out of synch.

The next two faces of the project will look into the optimization of the parallel processing even more and also utilize more underlying physical architecture of the platform it is run on.

It is clear that Python as an interpretive programming language, with its sequential limitations on the use of its interpreter, need to be considered for future implementations. Currently there are Python interpreters that "can" bypass the interpreters, in situations where heavy computations aew done on larger library structs. Need to be looked into further.

The second phase will be to integrate the parallel processes on common platforms like the Ray or Hadoop, which-one that suits best for the specific task. Clearly, only for heavy computations that otherwise would be handled by cloud-services.

I see the possibility that parallel processing also can be an added factor for improvements in the implementation of the SGD. Also, for small dataset, in combination with re-shuffle and resampling, parallel processing can be an added factor for speeding up and improving the results.

Due to the limitations of the Pythons GIL implementation and problems with global locks, I have at this moment lack of evidence, to conclude, that averaging, optimizing, hyperparameters, during parallel processing will improve over all performance. What seems to be quite clear, is that classic implementations of networks today, solemnly basing the computation on "compiler" and OS to utilize and optimize processing, can be further extended with threading and multiprocessing. We can see by the performance that single processing is limited by internal wait/io tasks.

I can also see this subject opening up for single source asynchronous input, or event driven input that continuously update your learning model – somewhat in comparison with face-recognition on your iPhone, where the weights/memory will be optimized and stored, in between updates.
Lastly: Instead of focusing purely on parallel processing, interchanging hyperparameters, one can easily see the opportunity to link processing in a chains and pass experience

(hyperparameters) to next process in line, or as a phase shift, with n-number of epochs in difference.

REFERENCES

[1] Python 3.9.1 Documentation - The Python Standard Library Concurrent Execution https://docs.python.org/3/library/multiprocessing.html

[2] Python 3.4.10 Documentation - The Python Standard Library 17 https://docs.python.org/3.4/library/subprocess.html?highlight=subprocess

[3] Morten Hjorth-Jensen, Computational Physics, Lecture Notes and sample code, Fall 2020, https://github.com/CompPhysics/MachineLearning

[4] Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning, Second Edition.
DOI https://**doi**.org/**10.1007**/978-0-387-84858-7; Copyright Information Springer-Verlag New York 2009; Publisher Name Springer, New York, NY; eBook ...

[5Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, Aurelien Geron. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow [2nd ed.] 978-1-492-03264-9. 174 47 66MB. English Pages 1150 Year 2019.

[6]Data Science from Scratch-First Principles with Python, Grus.
Afficher les collections Cacher les collections. Identifiant. ISBN : **978-1-491-90142-7**. ISBN : **978-1-491-90142-7**. ISBN : 1-491-90142-X. ISBN : 1-491-90142-X.

[7] Practical Statistics for Data Scientists - Peter Bruce & Andrew Bruce
978-1-491-95296-2 [M] www.allitebooks.com Dedication We would like to dedicate this book to the memories of our parents Victor G. Bruce and Nancy C. Bruce, ...

[8]Python Machine Learning, Sebastian Raschka & Vahid Mirjalili
23. sep. 2017 — Python Machine Learning - von Sebastian Raschka, Vahid Mirjalili (ISBN **978-1-78712-593-3**) bestellen. Schnelle Lieferung, auch auf ...

[9]A Primer on Scientific Programming with Python, Hans Petter Langtangen
ISBN **978-3-662-49886-6**; Free shipping for individuals worldwide. Please be advised Covid-19 shipping restrictions apply. Please review prior to ordering.

[10]A Neural Networks And Deep Larning, Michael Nielsen / Dec 2019
http://neuralnetworksanddeeplearning.com/index.html