**UIO**

**FYS-STK4155**

Project 3

December 14th, 2020

Kjell R. Christensen

**ABSTRACT**

The growing processing power that has emerged withing the PC architecture (CPU & GPU), distributed sub-processing and on-demand cloud processing, opens up for more parallelism and distributed ML processing. The project will run instances of the same FFNN in parallel, based on the breast cancer dataset in Skilearn, but 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 new inventions as multi-core concepts, to pure optimized graphical chips like the GPU. The latter are highly optimized chips to run thousands of graphical operations simultaneously. It has been seen that GPUs with highly optimized architecture for graphical manipulation, also are well suited for matrices 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 need 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.

Enabling network parallelism, we will show that 2 or instances, with independent randomly initialized weights will converge faster, and add even better predictions.

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 network to converge.

Three cases 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 predicted y-value and y.
3. We will now increase the award/penalty through the cross-entropy by adding 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 used two datasets, one test-set, generated by extracting random numbers and a dataset from MNIST. Due to problems with tuning our first NN, we have loaded the breast cancer 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 MainModule().

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 has 4 cores and the number of instances for the FFNN will be multiple of the number of cores.

**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

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.

Legg inn for cross-entropy of formel for cost function

For the linear regression we know that the Normal Equations might give an analytical solution for the design and coefficient matrix. Tempting as it might be, it should be used as a reference rather than a solution. It will give an “absolute” optimal solution for the equation, hence overfitting our model for the test data. Test point and regression line shown in plot [(generated\_data\_plot)](file:///Users/KRC/Documents/GitHub/KJELLRC-FYS-STK-4155-Project-3/Plots/training_linear_models/generated_data_plot.png)

For batch gradient descent, the routine is quite sensitive to regularization parameter (Lrate \* Gradient), and will not always converge towards the optimal. The following plot will show different Lrates and some of them are lacking conversion [(gradient\_descent\_plot)](file:///Users/KRC/Documents/GitHub/KJELLRC-FYS-STK-4155-Project-3/Plots/training_linear_models/gradient_descent_plot.png).

When we look at the “full” or batch gradient descent, the cost is high due to the size of the design matrix, but det gradient move very direct towards to optimum. Depending on the processing power, the mine-batch sizes can be adjusted, in accordance with the hw-platform.

Looking at the mini-batch and stochastic gradient descent, the routine is computational more cost efficient, but they don’t converge that easily towards the optimum, as shown in the next plot [(gradient\_descent\_path\_plot)](file:///Users/KRC/Documents/GitHub/KJELLRC-FYS-STK-4155-Project-3/Plots/training_linear_models/gradient_descent_paths_plot.png)

Both min-batch and stochastic solutions can be easily tuned to fit your hw-platform for testing, and make the routines suitable for rough predictions.

As can been seen, the stochastic gradient will not reach the optimum, and will continue to “jump” around. But still, it’s a good and rough estimator and is computational sheep.

For the logistic regression, we find that the Sigmoid activation function for the output layer do a reasonable job and clearly converge. An even better solution is using the cross-entropy.

Cross-entropy, as a routine, has a gradient that faster tends towards infinity for yhat, hence penalize the cost function more, for x-values close to the boundaries of [-1,1].

The penalty for wrong predictions is larger and the training converge faster by using the cross-entropy compared to sigmoid.

For FFNN we don’t get convergence for 1-hidden layer. The routine gives prediction between 10-19% for most values, and predictions looks randomized. There is not enough non-linearity, using only one layer.

Adding 2-4 extra hidden layers gives good predictions. Correlating the changes done to decreasing the learning-rate with increased number of epochs moves prediction towards 90%. For the sample set of breast cancer from sklearn, the data set seams limited, to gain better predictions by increasing hidden layers beyond 4-layers. The model overfits, beyond 4-layers and prediction for test data lacks behind more and more (2-3%). See excel sheet in catalog, generated for the testcases for the trends.

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 predictions could not be done. As far as I could see, this was only due to the randomized generation of weights and biases. Based on reference in [referanse til balancing], the balance of we have a 50/50 split

The Stochastic Gradient Descent (SDG), was implemented using standard functionality and we would liked to have momentum as part of our solution, but will be implemented in the next phase of the project. We would also like to see a comparison between using the sigmoid and cross-entropy as classification function in the output layer.

For the FFNN, our first implementation was with just one hidden layer and did not manage to predict, so no conversion. Our test result for MNIST gave an average of 10% for the 10 classes and with the prediction levels set at 0,5 (50%), no predictions were found for the data set. We need more than 2-hidden layers to gain enough non-linearity to get a conversion. To test multilayer networks we implemented the breast cancer. A clear conclusion: We need a minimum of two hidden-layers for a multi classification problem.

The expansion to a flexible multi-layer solution got stuck based on the timeframe and we used a fallback option where we tested the breast cancer data set. The solution was limited in its implementation and did not handle high numbers of epochs, due to variables running towards infinity.

With a 3+ -hidden layer solution we managed a prediction of 86%. Se all test results in [FFNNTests](file:///Users/KRC/Documents/GitHub/KJELLRC-FYS-STK-4155-Project-3/Plots/TestsNN.xlsx)

For epochs larger that 300, the network converged quite nicely.

For the breast cancer data our implementation was good, but we will implement SGD for the FFNN, when we approach larger datasets.

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’s presented for the breast cancer 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.

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 you are running on.

It is clear that Python as an interpretive programming language, with its sequential limitations on the use of its interpreter

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.

I see the possibility that parallel processing also can be an added factor for improvements in combination with SGD. Also, for small dataset, in combination with re-shuffle and resampling, parallel processing can be an added factor for 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 that over all performance. What seems to be quite clear is that classic implementations of networks today, solemnly basing the “compiler” and OS to utilize and optimize processing, is not optimal. 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 – something down the line as face-recognition on your iPhone.

Lastly – instead of focusing purely on parallel processing interchanging hyperparameters, one can easily see the opportunity to link processing in a chain and pass experience to next process in line.

REFERENCES

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

[2] 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 ...

# [3Hands-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.

[4]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.

# [5] 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, ...

[6]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 ...

[7]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.

[8]A Neural Networks And Deep Larning,  [Michael Nielsen](http://michaelnielsen.org/) / Dec 2019

<http://neuralnetworksanddeeplearning.com/index.html>