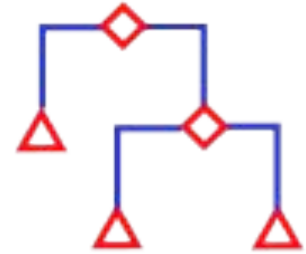
Parallel computing techniques for scaling hyperparameter tuning of Gradient Boosted Trees and Deep Learning

Dr. Nikolaos Bakas
nibas@grnet.gr

Contents

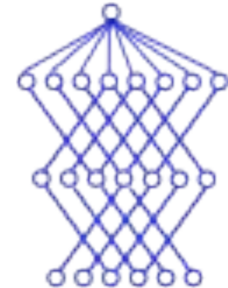
- ➔ The problem of Scaling of Hyperparameter Tuning
- ➔ XGBoost & PyTorch
- ➔ MeluXina Supercomputer
- ➔ CPU, Threads & Processes
- ➔ Computational Bottlenecks
- ➔ Parallelization Strategy
- ➔ Proposed Cross Validation Algorithm
- ➔ Search space of the Optimisation Algorithm
- ➔ Parallelization with Multiprocessing
- ➔ Handling multiple OpenMP runtimes
- ➔ Scaling-Up Results

Machine Learning
with Hyper-parameter Tuning



XGBoost

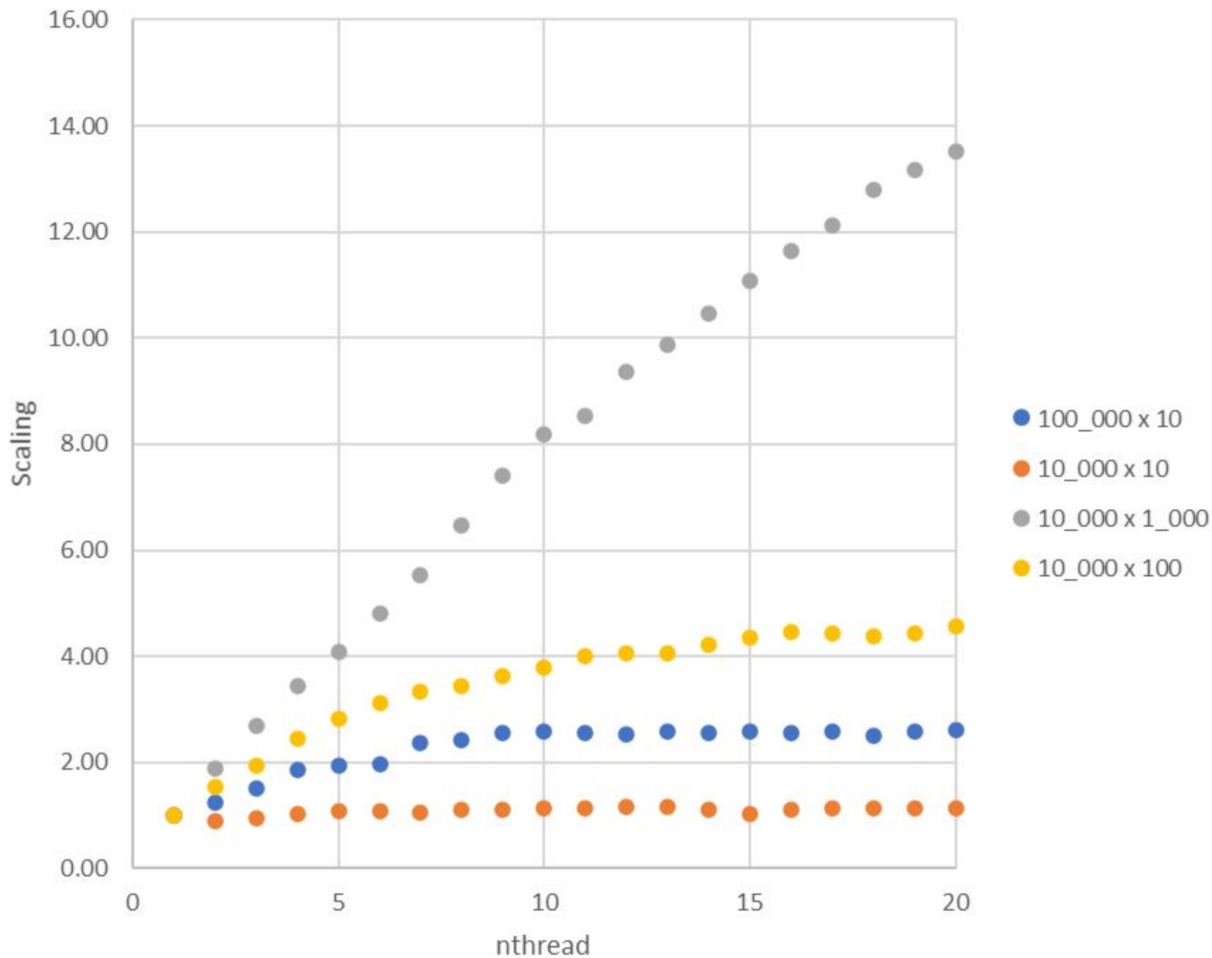
- ➔ We want to optimize the machine’s usage
- ➔ We have 256 threads on each node
- ➔ However XGBoost and Deep Learning **do not scale across many CPUs**
- ➔ Also, usually the datasets are small and even with batch size equal to the number of samples, utilise a **small percentage of a GPU**
- ➔ Each model run in different compute time, so parallelising many models is complex



Deep Learning

Scaling of XGBoost

- ➔ XGBoost compute times
- ➔ repeated 5 times
- ➔ and averaged



Compute modules

Compute Module	#Nodes	Type	CPU	Accelerator	RAM
Cluster	573	XH2000 (DLC)	2x AMD EPYC Rome 7H12 64c @ 2.6GHz	-	512GB

On 1 single node

MeluXina Supercomputer

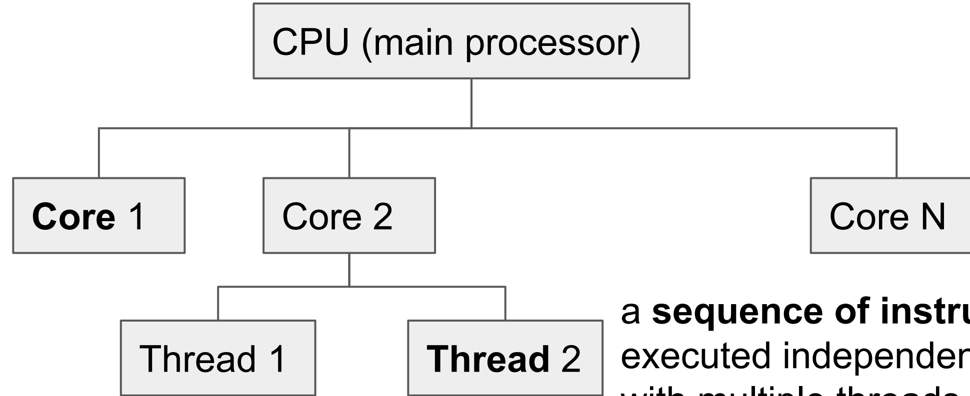
https://docs.lxp.lu/system/overview/

Accelerator - GPU	200	XH2000 (DLC)	2x AMD EPYC Rome 7452 32c @2.35GHz	4x NVIDIA Ampere 40GB HBM	512GB

```
  0[|98.1%]  16[|96.8%]  32[|99.4%]  48[|96.8%]  64[|83.2%]  80[|98.7%] 96[|98.7%]112[100.0%] 128[|98.1%] 144[|96.2%] 160[|98.1%] 176[|98.1%] 192[|96.2%] 208[|96.8%]224[100.0%]240[|98.1%]
  1[|98.7%]  17[|98.1%]  33[|97.4%]  49[|96.8%]  65[|98.1%]  81[|98.1%] 97[|98.1%]113[|98.7%] 129[|98.1%] 145[|96.8%] 161[|97.5%] 177[|97.4%] 193[100.0%] 209[|98.1%]225[|97.4%]241[|96.8%]
  2[|96.1%]  18[|98.1%]  34[100.0%]  50[|97.4%]  66[|96.2%]  82[|97.4%] 98[|98.1%]114[|97.4%] 130[|97.4%] 146[|96.8%] 162[100.0%] 178[|98.1%] 194[|98.7%] 210[|98.1%]226[|98.1%]242[|96.8%]
  3[|98.7%]  19[|98.1%]  35[|22.6%]  51[|96.2%]  67[|98.1%]  83[|96.8%] 99[|98.7%]115[|98.1%] 131[|98.7%] 147[|97.4%] 163[|98.1%] 179[|96.8%] 195[|96.8%] 211[|97.5%]227[|98.1%]243[|96.8%]
  4[|97.4%]  20[|96.8%]  36[|97.4%]  52[|96.8%]  68[|98.1%]  84[|97.4%]100[|98.7%]116[|96.8%] 132[|97.4%] 148[|96.8%] 164[100.0%] 180[|96.8%] 196[|98.1%] 212[|97.4%]228[|98.1%]244[|98.1%]
  5[|98.7%]  21[|98.7%]  37[|94.2%]  53[|98.1%]  69[|94.9%]  85[|97.4%]101[|97.4%]117[|63.5%] 133[|98.7%] 149[|96.8%] 165[100.0%] 181[|96.2%] 197[|98.1%] 213[|98.1%]229[|98.1%]245[|97.4%]
  6[|96.8%]  22[|96.8%]  38[|66.0%]  54[|97.4%]  70[|97.4%]  86[|96.8%]102[|96.2%]118[|96.8%] 134[|98.7%] 150[|97.5%] 166[|98.1%] 182[|98.7%] 198[|98.1%] 214[|98.1%]230[|97.4%]246[|98.7%]
  7[|97.4%]  23[|96.2%]  39[100.0%]  55[|97.4%]  71[|62.2%]  87[|97.4%]103[|98.1%]119[|98.1%] 135[|98.7%] 151[|97.4%] 167[|98.7%] 183[|98.7%] 199[|98.1%] 215[100.0%]231[|98.7%]247[|97.4%]
  8[|98.7%]  24[|98.7%]  40[|87.7%]  56[|96.8%]  72[|98.1%]  88[|97.4%]104[|97.4%]120[|97.4%] 136[|98.1%] 152[|96.8%] 168[|98.7%] 184[|96.8%] 200[|96.8%] 216[|98.1%]232[|96.2%]248[|98.7%]
  9[|96.8%]  25[|98.1%]  41[|98.1%]  57[|97.4%]  73[|97.4%]  89[|98.7%]105[|98.1%]121[|98.1%] 137[|98.1%] 153[|96.8%] 169[|96.8%] 185[|98.7%] 201[|97.4%] 217[|98.1%]233[|98.1%]249[|98.1%]
 10[|98.1%]  26[|98.1%]  42[|97.4%]  58[|97.4%]  74[|98.1%]  90[|96.8%]106[|98.1%]122[|96.8%] 138[|96.8%] 154[|96.8%] 170[|98.7%] 186[|97.4%] 202[|96.8%] 218[|96.8%]234[|97.4%]250[|97.4%]
 11[|96.8%]  27[|97.4%]  43[|98.7%]  59[|98.1%]  75[|96.2%]  91[|96.8%]107[|97.5%]123[|98.1%] 139[|98.7%] 155[|97.4%] 171[||3.8%] 187[|98.1%] 203[|96.8%] 219[|97.4%]235[|98.7%]251[|96.8%]
 12[|97.4%]  28[|98.1%]  44[|76.9%]  60[|97.4%]  76[|96.8%]  92[|97.4%]108[|98.1%]124[|98.1%] 140[|97.4%] 156[|96.8%] 172[|98.1%] 188[|98.7%] 204[|98.1%] 220[|96.8%]236[|96.2%]252[|98.1%]
 13[|97.4%]  29[| 0.6%]  45[|96.2%]  61[|97.4%]  77[|96.2%]  93[|91.7%]109[|98.1%]125[|98.1%] 141[|96.8%] 157[|98.1%] 173[|96.8%] 189[|98.7%] 205[|96.8%] 221[|98.7%]237[|97.4%]253[|98.1%]
 14[|98.7%]  30[|98.7%]  46[|98.1%]  62[|97.4%]  78[|87.8%]  94[|96.8%]110[|98.1%]126[|98.7%] 142[|97.5%] 158[|98.7%] 174[|99.4%] 190[|97.4%] 206[|97.5%] 222[|98.1%]238[  0.0%]254[|97.4%]
 15[|98.1%]  31[|97.4%]  47[|98.1%]  63[|98.7%]  79[|97.4%]  95[|96.8%]111[|98.7%]127[|97.4%] 143[|98.1%] 159[|98.1%] 175[|98.1%] 191[|97.5%] 207[|28.8%] 223[|96.8%]239[|96.2%]255[|98.7%]
Mem[|||||                                                    15.9G/503G] Tasks: 43, 293 thr, 2911 kthr; 227 running
Swp[                                                             0K/0K] Load average: 1.97 60.10 121.57
                                                                        Uptime: 89 days, 01:38:08

[Main] [I/O]
  PID USER       PRI  NI  VIRT   RES   SHR S  CPU%▽MEM%   TIME+  Command
 7131 u100425     20   0 6754M  244M  8424 R 2430.9  0.0  0:48.33 python __mult_proc_loop__.py
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

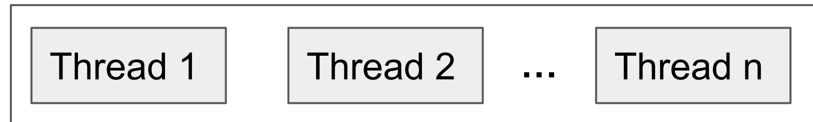
CPU-Cores-Threads

a **physical** processing unit within a computer's central processing unit (CPU) that can independently execute **instructions**.



a **sequence of instructions** that can be executed independently by a CPU. A CPU with multiple threads can work on multiple tasks in parallel, by using **hyper-threading** technology. This allows a single core to work on multiple threads concurrently.

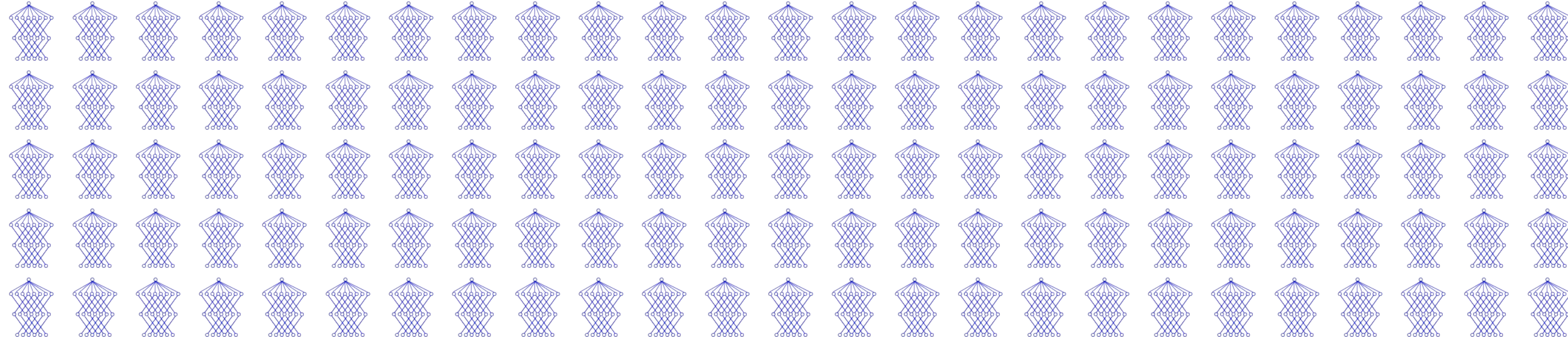
Task (Process)



Tuning Strategy - We train in parallel, one model per:

- ➔ thread or
- ➔ few threads if the potential models are less than the available threads

```
 0[|98.1%]  16[|96.8%]  32[|99.4%]  48[|96.8%]  64[|83.2%]  80[|98.7%]  96[|98.7%]112[100.0%] 128[|98.1%] 144[|96.2%] 160[|98.1%] 176[|98.1%] 192[|96.2%] 208[|96.8%]224[100.0%]240[|98.1%]
 1[|98.7%]  17[|98.1%]  33[|97.4%]  49[|96.8%]  65[|98.1%]  81[|98.1%]  97[|98.1%]113[|98.7%] 129[|98.1%] 145[|96.8%] 161[|97.5%] 177[|97.4%] 193[100.0%] 209[|98.1%]225[|97.4%]241[|96.8%]
 2[|96.1%]  18[|98.1%]  34[100.0%]  50[|97.4%]  66[|96.2%]  82[|97.4%]  98[|98.1%]114[|97.4%] 130[|97.4%] 146[|96.8%] 162[100.0%] 178[|98.1%] 194[|98.7%] 210[|98.1%]226[|98.1%]242[|96.8%]
 3[|98.7%]  19[|98.1%]  35[|22.6%]  51[|96.2%]  67[|98.1%]  83[|96.8%]  99[|98.7%]115[|98.1%] 131[|98.7%] 147[|97.4%] 163[|98.1%] 179[|96.8%] 195[|96.8%] 211[|97.5%]227[|98.1%]243[|96.8%]
 4[|97.4%]  20[|96.8%]  36[|97.4%]  52[|96.8%]  68[|98.1%]  84[|97.4%]100[|98.7%]116[|96.8%] 132[|97.4%] 148[|96.8%] 164[100.0%] 180[|96.8%] 196[|98.1%] 212[|97.4%]228[|98.1%]244[|98.1%]
 5[|98.7%]  21[|98.7%]  37[|94.2%]  53[|98.1%]  69[|94.9%]  85[|97.4%]101[|97.4%]117[|63.5%] 133[|98.7%] 149[|96.8%] 165[100.0%] 181[|96.2%] 197[|98.1%] 213[|98.1%]229[|98.1%]245[|97.4%]
 6[|96.8%]  22[|96.8%]  38[|66.0%]  54[|97.4%]  70[|97.4%]  86[|96.8%]102[|96.2%]118[|96.8%] 134[|98.7%] 150[|97.5%] 166[|98.1%] 182[|98.7%] 198[|98.1%] 214[|98.1%]230[|97.4%]246[|98.7%]
 7[|97.4%]  23[|96.2%]  39[100.0%]  55[|97.4%]  71[|62.2%]  87[|97.4%]103[|98.1%]119[|98.1%] 135[|98.7%] 151[|97.4%] 167[|98.7%] 183[|98.7%] 199[|98.1%] 215[100.0%]231[|98.7%]247[|97.4%]
 8[|98.7%]  24[|98.7%]  40[|87.7%]  56[|96.8%]  72[|98.1%]  88[|97.4%]104[|97.4%]120[|97.4%] 136[|98.1%] 152[|96.8%] 168[|98.7%] 184[|96.8%] 200[|96.8%] 216[|98.1%]232[|96.2%]248[|98.7%]
 9[|96.8%]  25[|98.1%]  41[|98.1%]  57[|97.4%]  73[|97.4%]  89[|98.7%]105[|98.1%]121[|98.1%] 137[|98.1%] 153[|96.8%] 169[|96.8%] 185[|98.7%] 201[|97.4%] 217[|98.1%]233[|98.1%]249[|98.1%]
10[|98.1%]  26[|98.1%]  42[|97.4%]  58[|97.4%]  74[|98.1%]  90[|96.8%]106[|98.1%]122[|96.8%] 138[|96.8%] 154[|96.8%] 170[|98.7%] 186[|97.4%] 202[|96.8%] 218[|96.8%]234[|97.4%]250[|97.4%]
11[|96.8%]  27[|97.4%]  43[|98.7%]  59[|98.1%]  75[|96.2%]  91[|96.8%]107[|97.5%]123[|98.1%] 139[|98.7%] 155[|97.4%] 171[||3.8%] 187[|98.1%] 203[|96.8%] 219[|97.4%]235[|98.7%]251[|96.8%]
12[|97.4%]  28[|98.1%]  44[|76.9%]  60[|97.4%]  76[|96.8%]  92[|97.4%]108[|98.1%]124[|98.1%] 140[|97.4%] 156[|96.8%] 172[|98.1%] 188[|98.7%] 204[|98.1%] 220[|96.8%]236[|96.2%]252[|98.1%]
13[|97.4%]  29[| 0.6%]  45[|96.2%]  61[|97.4%]  77[|96.2%]  93[|91.7%]109[|98.1%]125[|98.1%] 141[|96.8%] 157[|98.1%] 173[|96.8%] 189[|98.7%] 205[|96.8%] 221[|98.7%]237[|97.4%]253[|98.1%]
14[|98.7%]  30[|98.7%]  46[|98.1%]  62[|97.4%]  78[|87.8%]  94[|96.8%]110[|98.1%]126[|98.7%] 142[|97.5%] 158[|98.7%] 174[|99.4%] 190[|97.4%] 206[|97.5%] 222[|98.1%]238[  0.0%]254[|97.4%]
15[|98.1%]  31[|97.4%]  47[|98.1%]  63[|98.7%]  79[|97.4%]  95[|96.8%]111[|98.7%]127[|97.4%] 143[|98.1%] 159[|98.1%] 175[|98.1%] 191[|97.5%] 207[|28.8%] 223[|96.8%]239[|96.2%]255[|98.7%]
```



Search space of the Optimisation Algorithm

```
max_depth = list(arange(1, 11))
learning_rate = list(np_round(arange(0.001, 1.01, 0.001), 2))
colsample_bytree = list(np_round(arange(0.01, 1.01, 0.01), 2))
subsample = list(np_round(arange(0.01, 1.01, 0.01), 2))
combinations = list(product(max_depth, learning_rate, colsample_bytree, subsample))
```

Potential Combinations **= 101_000_000**

x

10_000 rounds

x

5 cross validation folds

=

>5 Trillion Models

Cross Validation Strategy

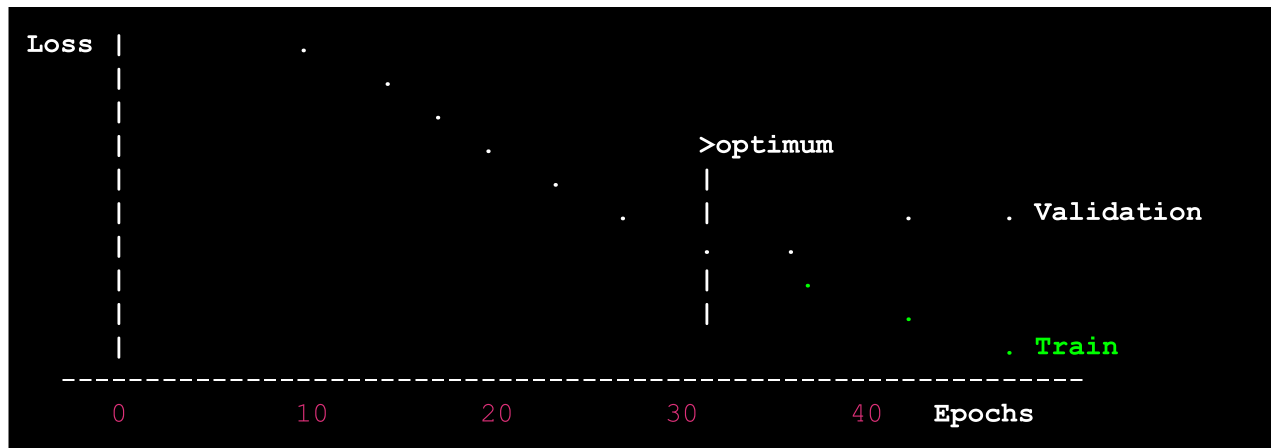
fold 1
fold 2
fold 3
fold 4
fold 5

Validation Sample
Train Sample

➔ *randomly permute without overlapping*

```
# split train to train and validation
___perc_cv___ = 0.8; nof_folds = 5; obs = len(ytr)
tr_inds, vl_inds = split_tr_vl(obs,___perc_cv___,nof_folds,PERMUTE_TRAIN_TEST)
print("split_tr_vl done",datetime.datetime.now().strftime('%H:%M:%S.%f')[:-3], flush=True)
```

Optimal Number of Rounds



2-Steps Tuning + Final Train

```
nof_1st_tune_epochs = 100
nof_1st_tune_models = 1_000
nof_2nd_tune_epochs = 1_000
nof_2nd_tune_models = 100
nof_final_blas_thr = 26
```

```python
#Tuning Parameters:

# max_depth: It represents the maximum depth of a tree. Increasing this value makes the model
# more complex and prone to overfitting. Lower values help prevent overfitting,
# but too low values may result in underfitting.
# range: [0,∞]. default=6
max_depth = list(arange(1, 11))

# learning_rate: It controls the step size at each boosting iteration.
# A lower learning rate makes the model more robust to overfitting,
# but it may require more boosting iterations to converge.
# Higher learning rates can lead to faster convergence, but increase the risk of overfitting.
# range: [0,1]. default=0.3
learning_rate = concatenate((   np_round(linspace(0.001,0.01,num=3), 10),
                                np_round(linspace(0.01,0.1,num=3), 10),
                                np_round(linspace(0.1,1.0,num=5), 10)))
learning_rate = list(unique(learning_rate))

# colsample_bytree: It specifies the fraction of columns to be randomly sampled for each tree.
# A value less than 1.0 introduces randomness and can help in reducing overfitting.
# range: (0, 1]. default=1
colsample_bytree = list(np_round(arange(0.1, 1.1, 0.1), 10))

# subsample: It represents the fraction of samples (observations) to be randomly selected for each tree.
# Lower values make the model more robust to overfitting by introducing randomness.
# The default value is 1.0, which means using all samples.
# range: (0,1]. default=1
subsample = list(np_round(arange(0.1, 1.1, 0.1), 10))

combinations = list(product(max_depth, learning_rate, colsample_bytree, subsample))
```

XGBoost Tuning Parameters

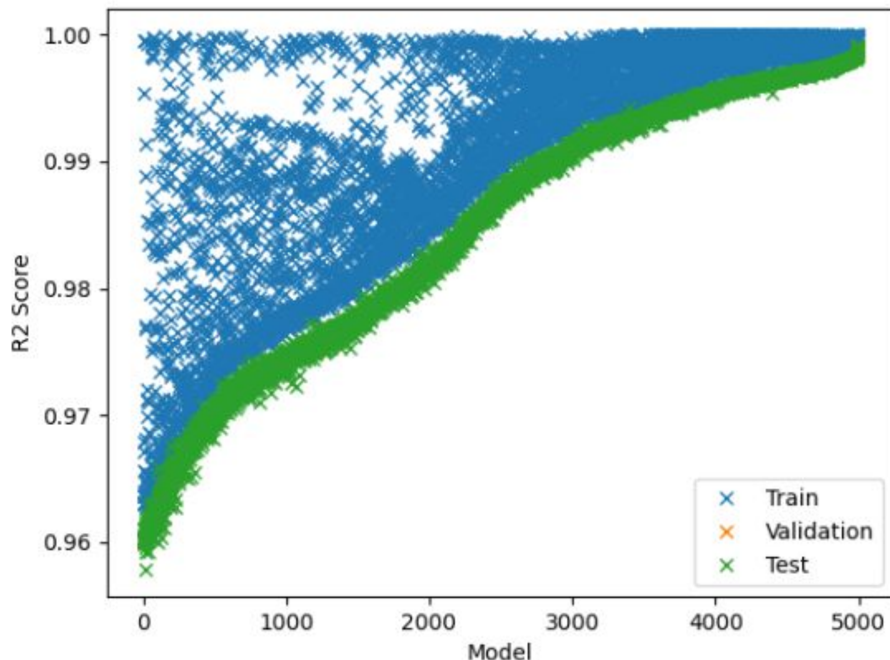
ANN Tuning Parameters

```
#Tuning Parameters:

# layers: It refers to the number of hidden layers in a neural network. Deep networks with multiple layers
# can capture complex patterns but are more prone to overfitting. Shallow networks with fewer layers may be
# simpler and less prone to overfitting but may struggle with complex tasks. The number of layers depends
# on the complexity of the problem, but a typical range is 1 to 10 layers.
# Lower bound: 1. Upper bound: No strict upper bound, but typically around 10. Suggested values: 1 to 10
layers = list(arange(1,11))#list(concatenate((arange(1,11),arange(20, 101, 10))))

# neurons: It represents the number of neurons (also called units or nodes) in each hidden layer of a neural network.
# Higher numbers of neurons can capture more complex relationships, but they increase the computational cost and
# the risk of overfitting. The number of neurons in each layer is problem-dependent and typically determined through experimentation.
# There is no strict range or suggested values since it heavily depends on the specific problem.
neurons = list(concatenate((arange(1,11),arange(20, 101, 10))))#,arange(200, 1001, 100)

# learning_rate: It determines the step size used to update the weights of the neural network during training.
# A higher learning rate may lead to faster convergence but can also cause overshooting the optimal solution.
# A lower learning rate may improve the model's stability but increase training time.
# The default value is often set to 0.001, but the optimal value depends on the problem and the network architecture.
learning_rate = concatenate((   np_round(linspace(0.0001,0.001,num=3), 10),
                                np_round(linspace(0.001,0.01,num=3), 10),
                                np_round(linspace(0.01,0.1,num=3), 10)))
learning_rate = list(unique(learning_rate))
```

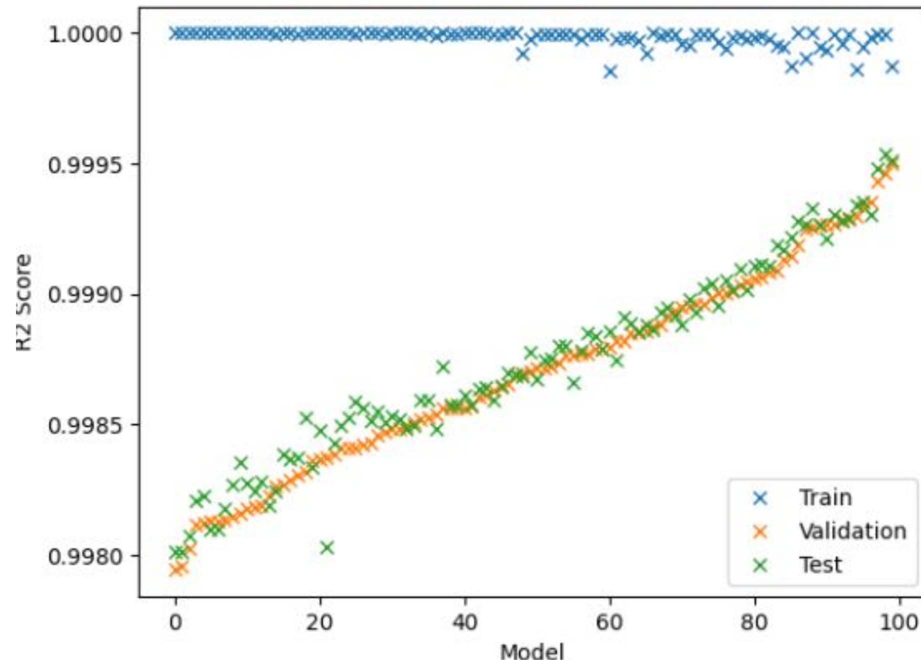
ANN Tuning Parameters

```
# dropout: It is a regularization technique that randomly sets a fraction of the neurons to 0 during training to prevent overfitting.
# Dropout helps in reducing the interdependence between neurons and encourages the network to learn more robust features.
# The typical dropout rate ranges from 0.1 to 0.5.
dropout =       concatenate((   np_round(linspace(0.001,0.01,num=3), 10),
                                np_round(linspace(0.01,0.1,num=3), 10),
                                np_round(linspace(0.1,0.5,num=3), 10)))
dropout = list(unique(dropout))

# batch size: It refers to the number of training samples propagated through the network before updating the weights.
# Larger batch sizes offer computational efficiency, but smaller batch sizes provide more stochasticity
# and can help escape local optima. The optimal batch size depends on the available memory, computational resources,
# and the dataset size. Common values range from 8 to 256.
batch = 2**arange(1,31)
while batch[-1]>obs/4:
    batch = batch[:-1]

# momentum: It is a parameter that accelerates convergence by adding a fraction of the previous weight update
# to the current update. Momentum helps in navigating flat regions and escaping local minima during training.
# Typical values range from 0.9 to 0.99.
moment_um = concatenate((   np_round(linspace(0.01,0.1,num=5), 10),
                            np_round(linspace(0.1,0.999,num=5), 10)))
moment_um = list(unique(moment_um))

combinations = list(product(layers, neurons, learning_rate, dropout, batch, moment_um))
```

2-Steps Tuning



50% of 1st round models



100 final models

- ➔ **These results regard the optimal epoch for each model.**
- ➔ **The are the <1% best**
- ➔ **The final range is really narrow**
- ➔ **We could claim that the algorithm converges to the true optimum**

```
import multiprocessing
```

→ How we distribute the models across threads

→ We call this twice, for the 2 tuning rounds

```
def run_mult_proc_xgb(combinations,LOGISTIC_REGR,nof_folds,Xtr,ytr,tr_inds,
                          vl_inds,Xte,yte,max_estimators,blas_threads):
    manager_tr = multiprocessing.Manager(); acc_tr_all = manager_tr.dict()
    manager_vl = multiprocessing.Manager(); acc_vl_all = manager_vl.dict()
    manager_te = multiprocessing.Manager(); acc_te_all = manager_te.dict()
    manager_nBest = multiprocessing.Manager(); nBest_all = manager_nBest.dict()
    jobs = []
    for i, (max_depth, learning_rate, colsample_bytree, subsample) in enumerate(combinations):
        p = multiprocessing.Process(target=train_xgb_folds,
            args=(i,max_depth,learning_rate,colsample_bytree,subsample,LOGISTIC_REGR,
                nof_folds,Xtr,ytr,tr_inds,vl_inds,Xte,yte,acc_tr_all,acc_vl_all,
                acc_te_all,nBest_all,max_estimators,blas_threads))
        jobs.append(p)
        p.start()
    for ij, proc in enumerate(jobs):
        proc.join()
    return acc_tr_all, acc_vl_all, acc_te_all, nBest_all
```

Train of the final model, without cross validation

```
with threadpool_limits(limits=nof_final_blas_thr,user_api='blas'):
    history, model = train_pytorch(Xtr, ytr,
                                   Xtr, ytr, Xte, yte,
                                   combinations[imax],
                                   int(epochs_all[imax]),
                                   path_,
                                   True)
```

Conflict of numpy.linalg & xgboost

- ➔ 2 OpenMP runtimes are loaded in the same Python program!
  - ◆ numpy comes with MKL and its Intel OpenMP (libiomp5) implementation
  - ◆ xgboost is installed against GNU OpenMP (libgomp)
- ➔ For different set of hyperparameters, different compute time is necessary (1st tuning)
- ➔ Assign different number of threads (2nd tuning)

Solution:

https://github.com/joblib/threadpoolctl

limit the number of threads used in the threadpool-backed of common native libraries used for scientific computing and data science (e.g. BLAS and OpenMP).

```
with threadpool_limits(limits=blas_threads,
user_api='blas'):
    xgboost.fit(Xtr[tr_inds[fold],:], ytr[tr_inds[fold]])
```

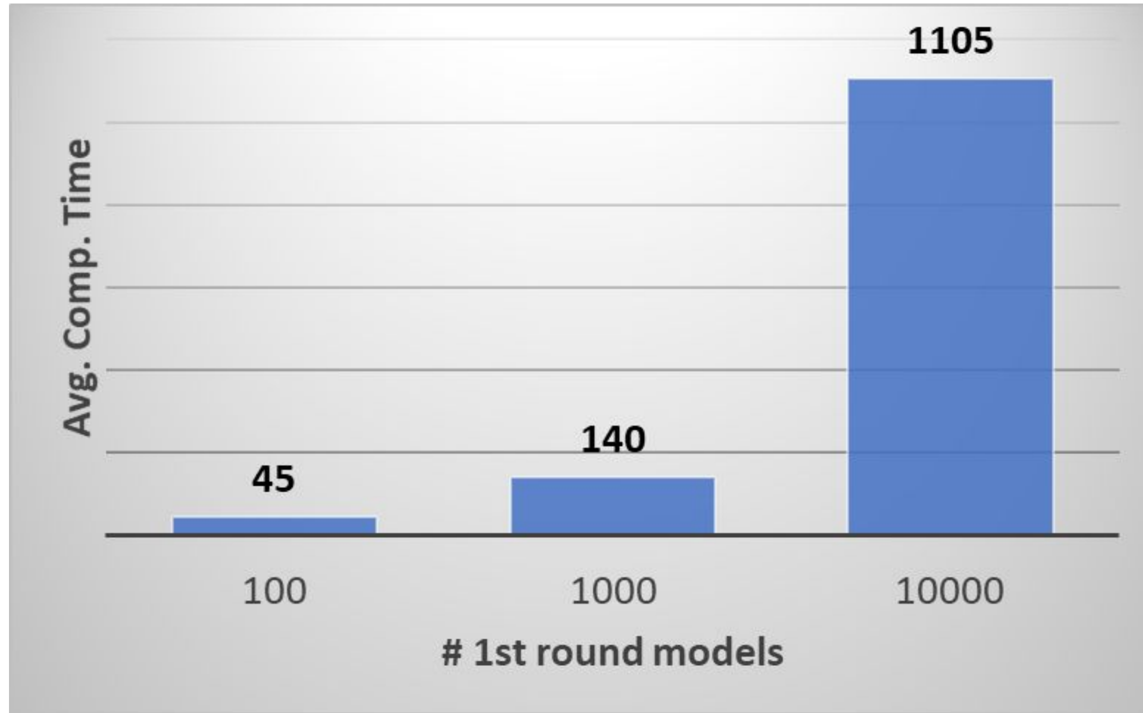
- ➔ `export OMP_NUM_THREADS=1`
- ➔ *the total number of distributed threads should not exceed cpu_count, if* `blas_threads>1`

Scaling

- ➔ We have 256-1=**255 available Threads**.
- ➔ When we run 1000 models, **we expect ~<4X compute time** relative to 255 AND 100 models.
- ➔ **140~4*45=180~<140**
- ➔ **Parallelisation worked!**

- ➢ **140** seconds instead of 10*45=**450**.
- ➢ **1105** seconds instead of 100*45=**4500**

Dataset of 98308 Steel Frames with SSI

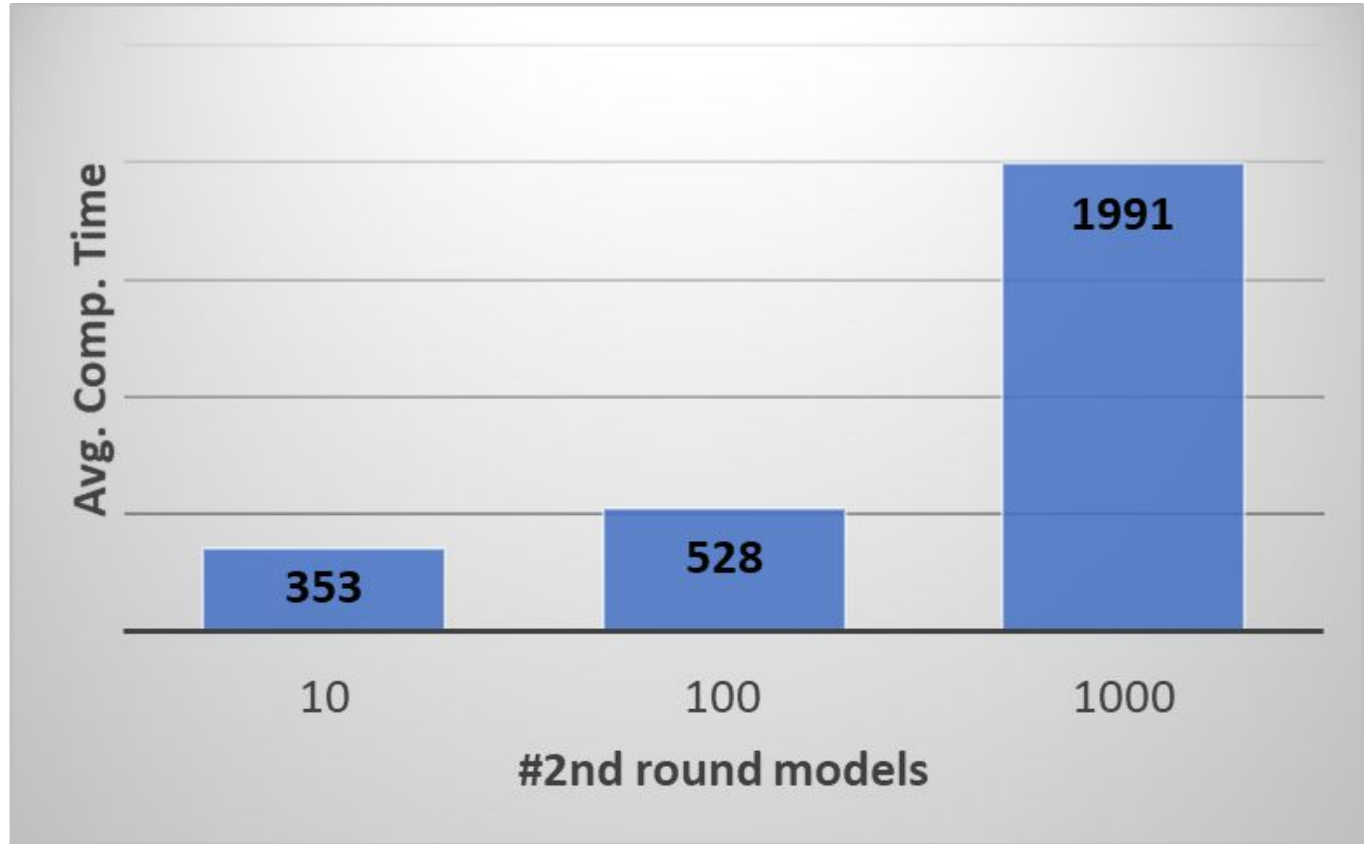


A.M. van der Westhuizen, N.P. Bakas and G. Markou, (2023), ***Big data generation and comparative analysis of machine learning models in predicting the fundamental period of steel structures considering soil-structure interaction,*** Submitted for Publication.

Scaling

Dataset of 98308 Steel Frames with SSI

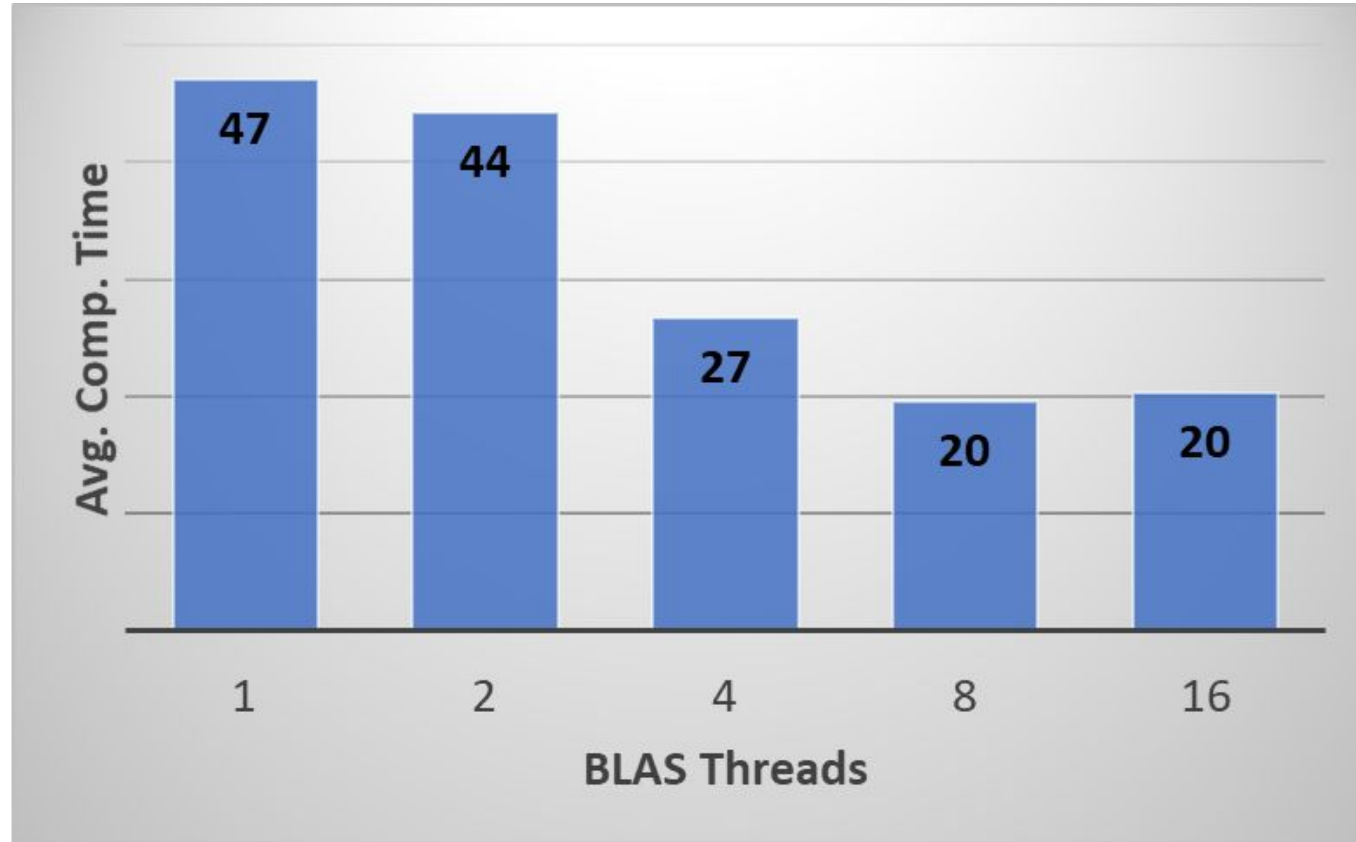
- ➔ **528** seconds instead of 3530
- ➔ **1991** seconds instead of 35300
- ➔ **Parallelisation worked!**



Scaling

Dataset of 98308 Steel Frames with SSI

For this dataset, using >8 threads does not increase performance in the final round



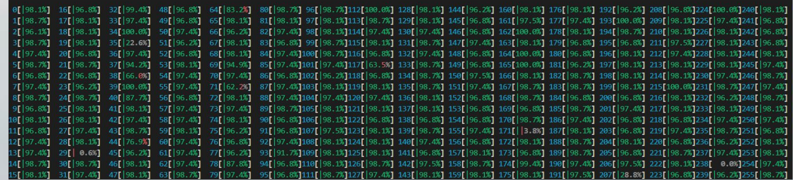
Scaling Up

- ➔ nof_1st_tune_rounds: 100
- ➔ nof_1st_tune_models: 250 (*1000)
- ➔ nof_2nd_tune_rounds: 1000
- ➔ nof_2nd_tune_models: 25 (*250)
- ➔ nof_final_blas_thr: 8

								Scaling Up			
Dataset	# Samples	# Features	# seconds 1st tune	# seconds 2nd tune	# seconds final train	# total seconds	# Threads	1st tune	2nd tune	Final	Total
steel_ssi	98_308	6	57.51	361.45	21.16	440.12	256	1.0	1.0	1.0	1.0
steel_ssi	98_308	6	1,897.37	2,207.39	21.39	4,126.15	8	33.0	6.1	1.0	9.4
fund_period	10_000	6	14.33	27.37	1.62	43.32	256	1.0	1.0	1.0	1.0
fund_period	10_000	6	314.65	251.60	1.63	567.88	8	22.0	9.2	1.0	13.1
fund_period*	10_000	6	45.84	64.07	2.25	112.16	256	1.0	1.0	1.0	1.0
fund_period*	10_000	6	1,260.14	2,742.67	1.63	4,004.44	8	27.5	42.8	0.7	35.7

Conclusions

- ➔ ***High Scaling on 1 single node!***
  - ◆ ***X 20+***
- ➔ ***Optimal Number of Rounds***
  - ◆ ***X 100, X 1000, …***
- ➔ ***2-Step Tuning***
  - ◆ ***X 10+***
- ➔ ***Cloud Computing***
  - ◆ ***X ∞***



Nikos Bakas
nibas@grnet.gr
https://eurocc-greece.gr/

Acknowledgment: EuroCC 2 Project is funded by the European Commission. Parts of the runs were performed on the MeluXina (https://docs.lxp.lu/) Supercomputer.

EuroCC@Greece