Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування і економічної кібернетики

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти «магістр»

«Аналіз ефективності алгоритмів випадкового пошуку в машинному навчанні»

«On the effectiveness analysis of Random search optimization algorithms in machine learning»

Виконала: здобувачка денної форми навчання

спеціальності 113 Прикладна математика
Освітня програма «Прикладна математика»

Висторобська Лоліта Вячеславівна

Керівник канд. фіз.-мат. наук, доц. Страхов Є. М.

Рецензент канд. техн. наук, доц. Мороз В. В.

Рекомендовано до захисту:

Протокол засідання кафедри

№ ___ від _____ р.

Завідувач кафедри

_____ _____
(підпис) (прізвище, ім'я)

Захищено на засіданні ЕК №_____

протокол № ____ від _____ р.

Оцінка_____/___ ___/_____
(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

_____ _____
(підпис) (прізвище, ім'я)

Одеса – 2022

Odesa I. I. Mechnikov National University
Faculty of Mathematics, Physics and Information Technology
Department of Optimal Control and Economic Cybernetics

QUALIFICATION WORK

for obtaining the degree of higher education «master»

«On the effectiveness analysis of Random search optimization algorithms in machine learning»

Fulfilled by: full-time student
specialty 113 Applied Mathematics
Lolita Vystorobska
Supervisor: Associate Prof. Y. Strakhov
Reviewer: Associate Prof. V. Moroz

Odesa — 2022

CONTENTS

INTRODUCTION

Optimization is a frequent goal in many studies, and here optimization in the context of neural networks will be discussed as well, namely the optimization of hyper parameters. Here, a set of methods is to be evaluated and compared with significant emphasis on random search and natural computing algorithms.

So let us introduce the first base term referring to this work, namely as from [1] Stochastic optimization (SO) methods are optimization methods that generate and use random variables. For stochastic problems, the random variables appear in the formulation of the optimization problem itself, which involves random objective functions or random constraints. Stochastic optimization methods also include methods with random iterates. More specific sub-type of such methods introduced here are Natural Computing (NC) ones.

Beginning with definition of Natural Computing term as mentioned in [2], [it] refers to computational processes observed in nature, and human-designed computing inspired by nature. When complex natural phenomena are analyzed in terms of computational processes, our understanding of both nature and the essence of computation is enhanced. Characteristic for human-designed computing inspired by nature is the metaphorical use of concepts, principles and mechanisms underlying natural systems.

Beneficial cooperation among organisms and entities has suggested new ideas for search and control engineering. The look at highly interconnected networks of simple biological processing unit, that can learn and adapt, has introduced the way for our development of computational systems that can differentiate between complex patterns, and improve themselves over time.

By studying complex biological organisms, looking at nature designed systems, industry and science was brought ways to explore innovative design approaches and develop even new products. It is worth to add note from [3], where 10 real-world successful implementations of NC algorithms were presented, that NC methods are more than substitute approach to the challenges faced in various domains. In many

fields, nature-inspired methods have overcome barriers in the prior achievements and capabilities of classical computing.

In the subsequent understanding of this paper, these NC algorithms in essence and in the actual sense refer to a more global concept of random search and stochastic optimization methods.

Another part of the current paper is Hyper Parameters optimization in Neural Networks (NN), where, for clarification, such a parameter is the one to be set prior the training process, cannot be deduced via model learning, and mainly, a set of hyper parameters can manage and control learning process itself. It can be a loss function, proper model layer configuration, activation function type, optimization technique etc. For now, hyper parameters will not be divided into those that belong to model and to algorithms, nevertheless such a separation could take place as well.

Thus, the main purpose of the current report is an investigation of various approaches of Hyper Parameters optimization in Neural Networks and its comparison. As Hyper Parameters optimization is showed up as a non-trivial task, especially in case of high number of the latter, it is needed to apply more sophisticated methods for resolving such a problem.

Nevertheless, there are no special literature and only a few papers that could present a broad comparison of the various methods of Hyper Parameters optimization. In most cases, researchers take some model by specific task, simple NN like Multilayer perceptron or Convolutional NN, and some Optimization method that did not used before or was modified, and eventually explore how those two perform together. Unfortunately, it is not standardized cause each time there are different models, data, search space as well as general parameters of the tests. That is why in this research the main objective is to sum up it all together, implement the selected search algorithms for specific task and perform the comparison analysis based on results.

For the following report, natural language processing (NLP) aspect-bsaed classification task has been chosen. As models, deep neural network (DNN) and gated

recurrent unit (GRU) model were used, and for a comparison, simple random search, tree-structured Parzen estimator (TPE); simulated annealing, particle-swarm optimization, harmony search, as well as genetic algorithms are to be evaluated.

ВСТУП

Оптимізація є частою метою в багатьох дослідженнях, і тут буде також обговорюватися оптимізація в контексті нейронних мереж, а саме оптимізація гіперпараметрів. Тут необхідно оцінити та порівняти набір методів із значним акцентом на випадковому пошуку та природних обчислювальних алгоритмах.

Отже, введемо перший базовий термін, що відноситься до цієї роботи, а саме з [1] Стохастичні методи оптимізації (SO) – це методи оптимізації, які генерують і використовують випадкові величини. Для стохастичних задач випадкові величини з'являються у формулюванні самої задачі оптимізації, яка включає випадкові цільові функції або випадкові обмеження. Методи стохастичної оптимізації також включають методи з випадковими ітераціями. Більш специфічним підтипом таких методів, представлених тут, є природні обчислення (NC).

Починаючи з визначення терміну «Природні обчислення» (NC), як зазначено в [2], [він] відноситься до обчислювальних процесів, що спостерігаються в природі, і розроблених людиною обчислень, натхнених природою. Коли складні природні явища аналізуються з точки зору обчислювальних процесів, наше розуміння як природи, так і суті обчислень покращується. Характерним для розроблених людиною обчислень, натхнених природою, є метафоричне використання понять, принципів і механізмів, що лежать в основі природних систем.

Благотворна співпраця між організмами та сутностями запропонувала нові ідеї для пошукової і контрольної техніки. Погляд на дуже взаємопов'язані мережі простих біологічних процесорів, які можуть навчатися та адаптуватися, відкрив шлях до нашої розробки обчислювальних систем, які можуть розрізняти складні моделі та вдосконалюватися з часом.

Вивчаючи складні біологічні організми, дивлячись на створені природою системи, промисловість і наука знайшли шляхи дослідження інноваційних

підходів до проектування та розробки навіть нових продуктів. Варто додати примітку з [3], де було представлено 10 реальних успішних реалізацій NC-алгоритмів, що методи NC є більш ніж замінним підходом до проблем, з якими стикаються в різних областях. У багатьох галузях природні методи подолали перешкоди в попередніх досягненнях і можливостях класичних обчислень.

У подальшому розумінні цієї статті ці алгоритми NC по суті та в реальному сенсі відносяться до більш глобальної концепції методів випадкового пошуку та стохастичної оптимізації.

Іншою частиною поточної статті є оптимізація гіперпараметрів у нейронних мережах (NN), де, для уточнення, такий параметр є параметром, який встановлюється перед процесом навчання, не може бути виведений за допомогою навчання моделі, і, головним чином, множина гіперпараметрів може керувати і контролювати сам процес навчання. Це може бути функція втрат, належна конфігурація рівня моделі, тип функції активації, методика оптимізації тощо. Поки що гіперпараметри не будуть розділятися на ті, що належать моделі та алгоритмам, проте такий поділ також може мати місце.

Таким чином, основною метою даної роботи є дослідження різних підходів оптимізації гіперпараметрів у нейронних мережах та їх порівняння. Оскільки оптимізація гіперпараметрів виявляється нетривіальною задачею, особливо при великій кількості останніх, для вирішення такої проблеми необхідно застосовувати більш складні методи.

Тим не менш, немає спеціальної літератури і лише кілька статей, які могли б представити широке порівняння різних методів оптимізації гіперпараметрів. У більшості випадків дослідники беруть певну модель за конкретним завданням, просту NN, як-от багатошаровий персептрон або згорткову CNN, і деякий метод оптимізації, який раніше не використовувався або був модифікований, і в кінцевому підсумку досліджують, як вони працюють разом. На жаль, це не стандартизовано, тому що кожен раз є різні моделі, дані, простір пошуку, а також

загальні параметри тестів. Тому в даному дослідженні головною метою є підсумувати все разом, реалізувати обрані алгоритми пошуку для конкретного завдання та провести порівняльний аналіз за результатами.

Для наступної роботи було обрано завдання класифікації відносно сутностей обробки природної мови (NLP). В якості моделей використовувалися класична нейронна мережа та модель вентильного рекурентного вузла, а для порівняння — простий випадковий пошук, tree-structured Parzen estimator (TPE); алгоритми змодельованого відпалу, рою часток, гармонійного пошуку, а також генетичний алгоритм мають бути оцінені.

Мета роботи: реалізувати обрані алгоритми пошуку для конкретного завдання, класифікації відносно сутностей, та провести порівняльний аналіз за результатами.

Об’єкт дослідження: моделі машинного навчання: класична нейронна мережа та модель вентильного рекурентного вузла; алгоритми випадкового пошуку: випадковий пошук, алгоритми змодельованого відпалу, рою часток, гармонійного пошуку, а також генетичний алгоритм.

Предмет дослідження: порівняльна характеристика методів випадкового пошуку.

Методи дослідження: обчислювальні експерименти.

CHAPTER 1
LITERATURE ANALYSIS AND APPROACHES INVESTIGATION

1.1. Classical methods limitations and formal description of HP optimization

Beginning with the state-of-the-art analysis in HP optimization, it worth to describe in numbers, how such an optimization becomes a complex trial, as it turns to time as well as space and resources consuming, what in combination with learning part itself turns into extremely long running computation event, thus have a huge search space.

For this assuming the example from [4], the equations are expressed as $f(u_1(op_1), u_2(op_2))$, where f is a binary function, u is a unary function, and op is operand. Using arithmetic operators as a binary function, identity, negation, logarithm, exponential, trigonometric function, square root, and square as a unary function, and constants zero, one and variable x as an operand, the search space becomes $4 \times 7^2 \times 3^2$ in total, reaching about 2000. Even searching for two formula is tricky to handle with the large search space which has a size of $2000^2 = 4000000$.

There are classical approaches to HP optimization, like grid search and manual tuning, which for dwindling number of HP act satisfactorily, however as models get more complex, this number rises, and grid search becomes a poor option compared with even simple version of random search. Because especially, as described in [5], for most data sets only a few of the hyper parameters really matter, but those different hyper-parameters are important on different data sets what was revealed through a Gaussian process analysis of the function from hyper-parameters to validation set performance.

On this stage, let us formally describe the HP optimization problem but beginning with model learnable parameters (usually refer to weights, $\boldsymbol{\omega}$) optimization for broader view. The general objective of the (NN) model $\boldsymbol{M}$ is to derive such an approximation function F that minimize the loss $L(X, f)$ between predicted and actual

(grand truth) outputs over input vectors *X*. However, as mentioned above model itself with inner optimization algorithm have Hyper Parameters $\boldsymbol{\varphi}$ that to be optimized as well, and by choosing which the final model can be described in following context $F = \boldsymbol{M}^{\boldsymbol{\varphi}}(X^{train})$, and where HP optimization can be conducted:

$$\boldsymbol{\varphi}^* = \min L(X^{validation}, \boldsymbol{M}^{\boldsymbol{\varphi}}(X^{train})) \quad (1.1)$$

1.2. Random search

Returning to the paper mentioned earlier [5], where random search surpassed classical approaches, a few more advantages of random search algorithms, to be noticed compared with conducting grid ones. Based on detachment of every statistical test, the latter can be stopped at any moment and the whole sequence form a complete experiment, where each test can be conducted asynchronously. Having additional computational resources more tests can be thrown in an experiment without grid regulation and if some test throws an error or is just unsuccessful, it can be easily restarted or neglected without disrupting the whole experiment.

Referring again to research in [5], random experiments with large numbers of trials also bring attention to the question of how to measure test error of an experiment when many test have some claim to being best. When using a relatively small validation set, the uncertainty involved in selecting the best model by cross-validation can be larger than the uncertainty in measuring the test set performance of any one model. It is important to take both sources of uncertainty into account when reporting the uncertainty around the best model found by a model search algorithm. This technique is useful to all experiments (including both random and grid) in which multiple models achieve approximately the best validation set performance.

Finally, the optimization strategies described by [5], are non-adaptive: they do not influence the course of the experiment based on already available results. Random search can be generally not as good as the sequential combination of manual and grid

search from an expert by [deep-nets-icml-07.pdf] in the case of the 32-dimensional search problem of deep belief network optimization, because the efficiency of sequential optimization overcame the inefficiency of the grid search employed at each step of the procedure.

Thus, in current work, adaptive random and natural computing optimization algorithms are to be investigated and estimated.

1.3. Genetic search

In [4] an approach to find both activation function and optimization technique was proposed based on genetic algorithm, where at the same time focus on the relation between forward and backward processes in the activation function and optimization technique was hold. Here, the activation function and the optimization technique were encoded into parse trees and place them into chromosomes of individual. Each individual x_i differs from other individuals in a common neural network by the activation function and optimization technique of the chromosomes $c_1^{x_i}$ and $c_2^{x_i}$. $Net(c_1^{x_i}, c_2^{x_i})$ represented by chromosomes is defined as child network. This network is constructed with individual x_i, learned with CIFAR-10 dataset, and the accuracy is used as the fitness function of x_i.

Fitness function is calculated by learning and validating with the same dataset in a child network that represents each individual within a population in a generation.

The general process of genetic algorithms can be described as following:

1. Select parent individuals from the current population, the probability of selection being proportional to fitness.
2. Crossover at the randomly is chosen to form two offspring with probabilityp_c;
3. Mutate the two offspring at each point with probability p_m, and place the resulting individuals in the new population.

4. 1-3 steps will be repeated until N offspring have been created.

5. Replace the current population with the new population.

The activation function and the optimization technique were showed as (1) and (2), where was an attempt to evolve f, g in each equation.

$$x_i = f(x_{i-1}), \qquad (1.2)$$

$$\theta = \theta - g(\nabla_\theta J(\theta)), \qquad (1.3)$$

where x_i is an input of the i^{th} layer, f is an activation function, θ is weights of deep learning model, g is an optimization technique, $J(\cdot)$ is a loss function, and ∇_θ is a gradient with respect to θ.

Selection, mutation, and crossover were used as operators for the next generation based on individuals' fitness.

Thus, in the above discussed paper, automatically found equations for activation function and the optimizer have the best performance compared to the combination of the existing activation function and the optimization technique. Moreover, generated in optimization process SineLU function proved in analysis that it is better than the conventional activation function, and it is stated as following:

$$f(x) = \begin{cases} sin\beta x + \beta x, & x \geq 0 \\ sin\beta x, & x < 0 \end{cases}, \qquad (1.4)$$

where $\beta = 0.5 \; and \; \beta \in [0., 1.]$.

In the next paper [6] a genetic algorithm was used to tune only one hyper parameter in transfer CNN model, namely trainable layers of the transfer model. The task was simplified by boundaries that layers can only follow the strict scheme, that looks like sequence of frozen layers, next go sequentially only trainable ones, and again either frozen ones or none, if it was the last layer optimization ran on. Such a scheme constructs a bivariate optimization problem, which will change the 2^T space to T×(T-1)/2, where T is the number of all layers which state to be chosen. Another task provided here, was an interpretability whereby help of the GA guided results,

additional information can be extracted by analyzing other features such as gradients/ backward inference. The filter criterion is constructed by accuracy and the counts of the trainable layers. The fitness function is computed by accuracy and the counts of the trainable layers, and the probability, that whether the i_{th} individual is selected to survive, is also influenced by the fitness. The optimized transfer model converged with a precision of 97% in the classification in no more than 15 generations.

In [7], the same task was investigated, applying Cartesian genetic programming encoding method to optimize CNN architectures automatically for vision classification. A node function in Cartesian genetic programming was constructed including tensor concatenation modules and convolutional blocks. The recognition accuracy is set as the target of Cartesian genetic programming, while connectivity of the CNN architecture and the Cartesian genetic programming are optimized.

One more interesting point is an implementation of so-called forced mutation to efficiently use the computational resource by applying the mutation operator until at least one active node changes for reproducing the candidate solution. By the validation, their method is proved to be capable to construct a CNN model that comparable with state-of-the-art models.

1.4. Harmony search

One more paper presented an application of natural computing algorithm for hyper parameter tuning in CNN. In [8], a metaheuristic optimization method called meter-setting-free harmony search (PSF-HS) was used for hyperparameters optimization in the feature extraction step of a CNN, namely kernel size, stride, zero padding, number of channels of the convolutional layer, and kernel size and stride of the pooling layer. In such a method, the latter, given in combination as harmony, affect the size of the feature map of the layer in the feature extraction step of a CNN, thus in the PSF-HS algorithm, each solution, called a harmony, is represented by a vector, and

is stored in the Harmony Memory (HM). Each harmony is generated by random selection, harmony memory consideration, and pitch adjustment methods.

The HM is updated based on the loss of a CNN constructed using the generated harmony vector. The termination condition was set so that all harmony vectors stored in the HM converge into one harmony vector. The simulation results showed that by tuning the hyperparameters of a CNN, the number of weights and biases that need to be trained were reduced, and classification accuracy was improved.

1.5. Particle swarm optimization

Another natural computing algorithm was used for HP optimization in classification and regression task in [9], namely particle swarm optimization (PSO). However, since the formal idea of the mapping from hyper-parameters space to generalization accuracy is unclear, thus (PSO) methods cannot be directly used in the problem of HP assessment. The solution is the Bayesian Optimization (BO) Framework, that provides an opportunity to turn the HP optimization into the optimization of an acquisition function.

The optimal value of an approximation function is derived during BO search by forming a posterior probability of the function's output. A surrogate model, consisted of a prior distribution, is used to construct the mapping from the hyper-parameters to the model accuracy and in this paper a Gaussian process is chosen for this role. Thus eventually, the optimization of the hyper-parameters is converted into an optimization problem of the acquisition function α, where PSO is coming to help. In general, α is non-convex and multi-peak, where the non-convex optimization problems are to be solved in the search space X. In this work, acquisition function is chosen as upper confidence bounds (UCB), that trades off exploration against exploitation. PSO algorithm is simple, with a few adjustment parameters and fast convergence speed. Plus, what is the advantage, it is not necessary to calculate the derivatives of the objective function in the process of PSO. Thus, a few words about algorithms structure:

1. A population of particles with random points and velocity on n-dimensions of the search space is initialized.

2. For each particle, the fitness function in n-dim is estimated.

3. Particle's fitness evaluation is compared with particle's beat value (p_{best}). If current value is better than best one, then let p_{best} be equal to the current value, and the p_{best} location equal to the current location.

4. Fitness evaluation is compared with the population's overall previous best (g_{best}). If current value is better than g_{best} , then reset g_{best} to the current particle's array index and value.

5. The position and velocity of the particle should be changed according to equations (1)(2):

$$v_i(t+1) = \omega v_i(t) + c_1 r_1\left(p_{best\,i}(t) - x_i(t)\right) + c_2 r_2\left(g_{best\,i}(t) - x_i(t)\right) \quad (1.5)$$

$$x_i(t+1) = x_i(t) + v_i(t+1)\,, \quad (1.6)$$

where

c_1 and c_2 are the learning factors of the algorithm,

r_1 and r_2 are random variables uniformly distributed in [0, 1].

6. Execute Step 2 until a criterion is met.

Thus, in this paper PSO method is used to optimize the acquisition function to obtain new evaluation points, that significantly reduces the computational burden. Empirical evaluation on machine learning model showed that PSO-BO improves upon the state of the art. The resulting method can be used with most acquisition function. However, the algorithm runs slowly in high-dimensional space.

1.6. Artificial Bee Colony search

[10] presented one more nature inspired algorithm called Artificial Bee Colony (ABC) for HP optimization in classification task. The general process in ABC built on 4 main phases:

1. The initial food sources are randomly produced via the expression:

$$x_m = l_i + rand(0,1) * (u_i - l_i), \quad (1.7)$$

where u_i , l_i – upper, lower bound of the search space.

2. The neighbor food source v_{mi} is determined and calculated by the equation:

$$v_{mi} = x_{mi} + r_{mi}(x_{mi} - x_{ki}), \quad (1.8)$$

where i – randomly selected parameter index, x_k – randomly selected food source, r_{mi} random variable uniformly distributed in [0, 1].

The fitness function is calculated as following:

$$fit_{mi}(x_m) = \begin{cases} \frac{1}{1+f_m(x_m)}, & f_m(x_m) > 0 \\ 1 + |f_m(x_m)|, & f_m(x_m) < 0 \end{cases}, \quad (1.9)$$

where $f_m(x_m)$ - objective function.

After this calculation greedy selection is applied between the x_m and v_m .

3. The quantity of a food source is evaluated by its profitability and the profitability of all food sources. P_m is determined by the formula

$$P_m = \frac{fit_m(x_m)}{\sum_{m=1}^{SN} fit_m(x_m)} \quad (1.10)$$

Onlooker bees search the neighborhoods of food source according to the expression from the step 2.

4. The new solutions are randomly searched by the scout bees using the expression as for step 1:

$$x_m = l_i + rand(0,1) * (u_i - l_i) \quad (1.11)$$

In the mentioned paper, nevertheless, such an algorithm was specifically adopted in order to handle a combination of HP types while original ABC works only with continuous problems. Moreover, the range of variables in basic ABC is the same for all the dimensions, what also needed to be resolved for optimization task.
Thus, in the study, categorical variables initially are encoded as integers and treated as discrete ones afterward. After the optimization process is done and before training the model, these variables are converted again to corresponding values. For binary categorical variables, the hyper-parameter value is flipped, namely: flipped = 1 − binary value.

The results of the experiment in discussed study showed that HP-ABC improves the classification accuracy as well as decreases the tuning time compared to other state-of-the-art approaches. What is more important, HP-ABC proved an ability to solve the HPO problems with large search spaces.

The same ABC algorithm was used for HP optimization in [11], where a few modifications worth to be noticed. The convergence rate of the algorithm was enhanced by applying K-means clustering to the population of solutions, which means avoiding the evaluation of each solution by calculating only the cluster centroids. Moreover, opposition-based strategy was implemented to balance exploration and exploitation steps. The results on a real-world data in this study demonstrates faster convergence speed and running time without decreasing the accuracy in most cases and has an advantage over previous approaches.

1.7. Comparison study attempt

In [12] eventually comparison research was provided for a sophisticated task, namely HP optimization for sentiments analysis in Arabic language. In this paper, five

hyperparameter tuning approaches are presented: Grid Search, Random Search, Particle Swarm Optimization (PSO), Genetic Algorithm (GA) and Bayesian Optimization. These algorithms are used to perform the hyperparameter optimization of six machine learning algorithms to analyze Arabic reviews from sentimentally side. The experimental results showed that the Support Vector Classifier offers the best accuracy both before and after hyperparameter tuning, with the highest score using Bayesian Optimization. Meanwhile, PSO and GA dramatically enhanced the score of the Naive Bayes classifier.

CHAPTER 2
SEARCH ALGORITHMS IMPLEMENTATION

For the current research, Optuna [13] framework has been used as an automatic hyperparameter optimization software, which already includes some of algorithms, and gives an ability to implement custom algorithms a.k.a. samplers. All the algorithms were implemented within Optuna, what keeps standardized way of comparison for current experiments. Important terms\ objects that should be defined for future explanation are Study – optimization based on an objective function; Trial – a single execution of the objective function. The framework provides useful pruning mechanism, that was used in current research for each Study. This mechanism automatically stops unpromising trials at the early stages of the training by evaluating temporary results after each epoch.

Algorithms that come out of the box are TPE, Random Search and Genetic Search, all of them were discussed in the previous chapter, nevertheless, it worth to mention that the variant of the last one is sophisticated. Namely NSGA-II [14] what stands for “Nondominated Sorting Genetic Algorithm II”, is a multi-objective genetic algorithm with a fast non-dominated sorting approach and a selection operator that creates a mating pool by combining the parent and offspring populations and selecting the best N solutions. In Optuna documentation, an example of the Simulated Annealing Sampler can be found, but, unfortunately, it was not working, and needed to be fixed and modified, the final version of it can be found in Appendix A.3. Two more samplers were implemented as a part of current research, these are Particle Swarm Optimization (PSO) algorithm and Harmony memory search.

First of all, hyperparameters transformation should be described, mainly an approach to handling parameters with non-continuous distribution. For this purpose, Optuna has module _transform, that enabling conversion of search space bounds and parameter configurations into continuous space. Bounds and parameters associated with categorical distributions are one-hot encoded. Parameter configurations in this

space can additionally be untransformed or mapped back to the original space. For all kind of numerical distributions forward transformation is a type casting as float, while backward transformation depends fully on the exact distribution with respect to current bounds. In new samplers for search space generalization the next conversion features were used: BaseDistribution class methods to_external_repr, to_internal_repr for categorical parameters conversion; _transform module functions _transform_numerical_param, _untransform_numerical_param for numerical ones.

The essential part of the PSO is the mechanism of collecting the previous trials – particles here and preserving the direct connection between 2 trials from 2 sequential populations. This process also plays major role in logic, because we need to sort trial values within one population and reuse best trial in population (as well as one in the whole generation) for further parameters selection. The code implementation for the whole PSO sampler is provided in Appendix A.4. Another important part for PSO algorithms is the velocity recalculation that is computed using previous, predecessor particle, parameter velocity, as well as generation and population best parameters, and such constants as inertia weight, cognitive and social coefficients that are also be multiplied by random variables. This new velocity is afterward sum up with predecessor parameter, what gives the current parameter value for a trial.

Here, specifically for PSO, Optuna Study and Base Storage classes were extended in order to provide feature that could extract best trial not from the whole study but from last limited number of completed trials. These methods in both classes are called the same, best_trial_from_last_n, and its implementation is added to Appendix A.1, A.2.

Harmony memory search is the next custom sampler. Here, the stochastic components are presented through harmony memory considering rate (HMCR) and pitch adjusting rate (PAR), that play a role of thresholds. Moreover, these thresholds are recalculated for each trial based on completed trials number as well as maximum trials number. Afterwards comparison of random number with HMCR indicates if the random trial parameter from current harmony memory should be used for next

parameter recalculation, while comparison with PAR is called upon to decide if this random trial parameter should be used as it is or changed using band width constant with current parameter distribution limits. Harmony memory is also an important component of the algorithm, because random trial for future trials HPs generation can be selected only from the harmony memory subset if trials that has restricted size, and only best performed trials can be added there. Algorithm implementation can be examined as well in Appendix A.5.

CHAPTER 3
EXPERIMENT SETUP AND RESULTS

3.1 Task and dataset description

Current report subject is a Natural language processing (NLP) classification task, namely aspect-based sentiments analysis in Twitter comments dataset. As models, deep neural network as well as gated recurrent unit model have been selected, while 6 optimization algorithms were compared, among which 5 of them are in essence stochastic ones.

The task itself is more sophisticated than general classification because sentiment should be defined on the entity-level. It means, given a message and an entity, the task is to judge the sentiment of the message about the entity. There are three classes in this dataset: Positive, Negative, Neutral, Irrelevant. For DNN tests messages that are not relevant to the entity (i.e. Irrelevant) are regarded as Neutral, however for GRU tests all 4 classes were left as they are, because recurrent model has in general more capabilities against DNN and therefore can distinguish between relatively similar sentiments. Training set size was 74k samples, test set – 1k samples.

3.2 Data preprocessing

A few more words about data and its preprocessing difference depending on model. For both models, standard data cleaning such as non-alphabetic chars as well as punctuation filtering were applied along with repeated chars and stop-words removal. However, in DNN case embedding extraction has been done beforehand, and one-hot encoded entities matrix was concatenated to the embedding matrix, what in combination gives the actual input for the model. As a tokenization mechanism, spaCy [15] pipeline was used. In GRU case, embeddings were learned online during the train itself, and padding was applied to input sequences.

3.3 DNN implementation

The first model, deep neural network, is, roughly speaking, a perceptron with several levels of complexity, that was implemented using Pytorch framework [16]. As activation function after each layer LeakyReLU is used and it was constant, while number of layers as well as number of units inside each level have been selected by search algorithm on each trial as part of the experiment.

3.4 GRU implementation

The second model, Gated Recurrent Unit that has also been implemented using Pytorch framework, is an improved version of standard recurrent neural network (RNN) with special gating mechanism that aimed to solve vanishing gradient problem in RNNs. Namely, update and reset gates are used in GRU, these are two vectors which decide what information should be passed to the output. The main thing about them is that they can be trained to keep information from long ago, without cleaning it through time or remove information which is irrelevant to the prediction. The architecture of the GRU model unit is presented on the figure 3.1, and its temporary values as well as gates can be calculated as following:

$$z_t = \sigma(W^{(z)}x_t + V^{(z)}h_{t-1}) \quad (3.1)$$

$$r_t = \sigma(W^{(r)}x_t + V^{(r)}h_{t-1}), \quad (3.2)$$

where x_t – input vector with $W^{(z,r)}$ as its own weights; same goes for h_{t-1} – vector with information for the previous t-1 units and it is also multiplied by its own weight $V^{(z,r)}$; σ – sigmoid function; z_t, r_t – update, reset gates.

After that we have all inputs to compute current memory content, that will use reset gate to store the relevant information from the past:

$$h'_t = tanh(Wx_t + r_t \odot Vh_{t-1}), \quad (3.3)$$

where $\odot$ – Hadamard product operator.

In the end final memory at current time step should be evaluated using update gate along with current memory content:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h'_t \quad (3.4)$$

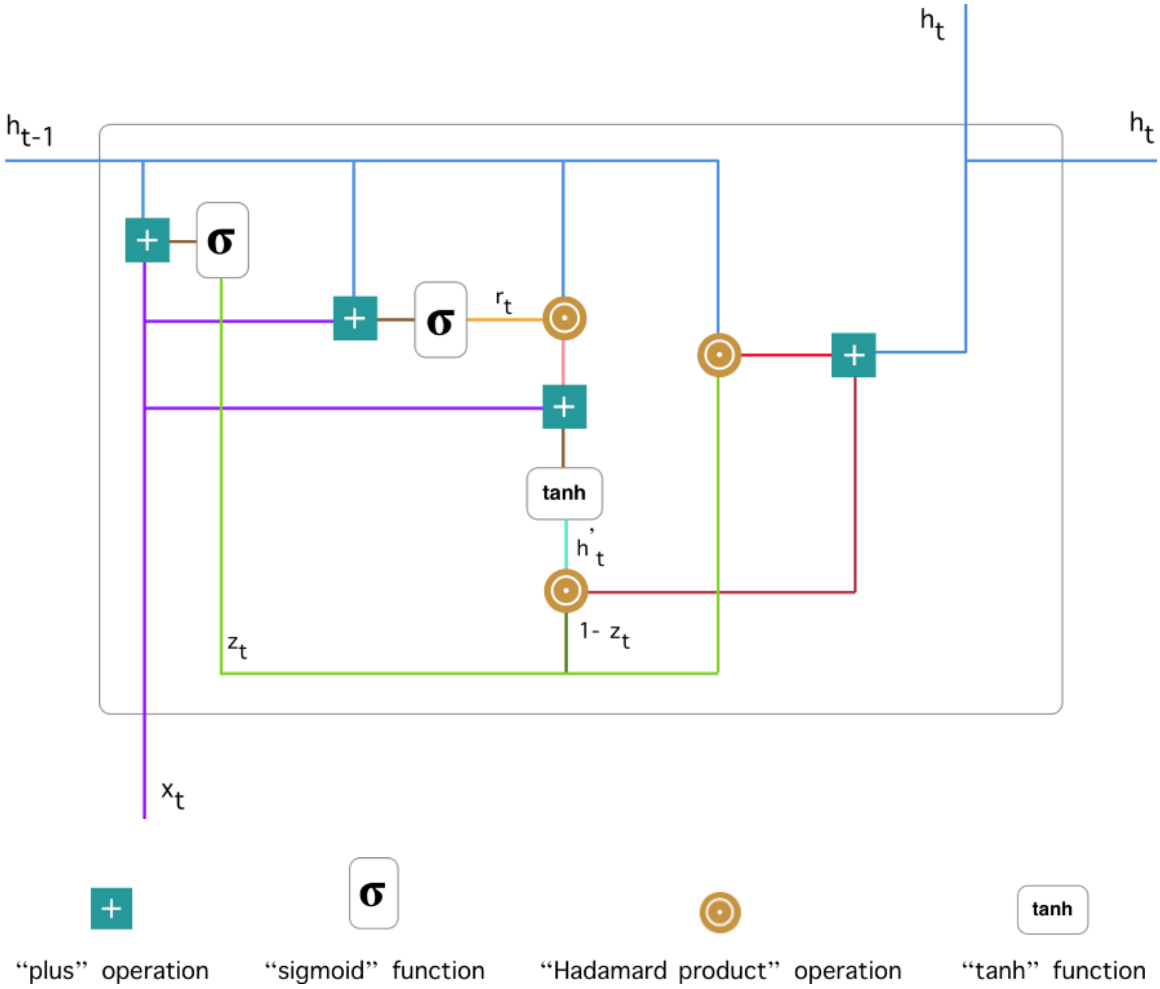

Figure 3.1 Gated Recurrent Unit [17]

3.5 Computational experiment

Current experiment consists of 6 HP search algorithms:

- Random search: define a search space as a bounded domain of hyperparameter values and randomly sample points in that domain.
- TPE from [18]: SMBO type of methods sequentially construct models to approximate the performance of hyperparameters based on historical measurements, and then subsequently choose new hyperparameters to test based on this model.

- Simulated Annealing: choose one of the previous trial points as a starting point, and then sample each HP from a similar distribution to the one specified in the prior, but whose density is more concentrated around the trial point we selected.
- Genetic search: uses evolutionary algorithm; and implements biologically inspired operators such as selection, crossover/ mutation.
- Particle swarm optimization: optimizes a problem by iteratively trying to improve a candidate solution regarding a given measure of quality.
- Harmony memory search: music-inspired metaheuristic optimization algorithm, based on the principal to find a perfect harmony state through improvisations.

The above algorithms details and implementation were covered in previous chapters, but it worth to mention that all algorithms were implemented within one framework, and although HP depends on model, within one model HPs as well as their distributions are all equal, thus from this point experiment setup can be described. Moreover, all the experiments were run on one GPU using pruning mechanism provided by Optuna framework, it means on each epoch the temporary results were evaluated and aborted for dead-end ones. For both models there are few algorithms parameters to be specified: for Harmony search harmony memory size is 8; for PSO particles number within generation is 12; for genetic search population size is 10.

DNN hyper parameters and search space:

- Learning rate: 1e-5 – 1e-1.
- Optimizer: Adam, AdamW, RMSprop.
- Batch size: 8 – 32.

- Epochs number: 3 – 8.
- Layers number: 1 – 3.
- Units number withing layer: 4 – 18.
- Trials number: 50.

GRU hyper parameters and search space:

- Learning rate: 8e-4 – 3e-3.
- Optimizer: Adam, AdamW, RMSprop.
- Embeddings dimension: 128 – 256.
- Epochs number: 3 – 8.
- Dropout probability: 0.2 – 0.6.
- Units number withing layer: 512 – 1024.
- Trials number: 30.

Hardware properties:

- Processor AMD Ryzen 9 5900HX with Radeon Graphics 3.30 GHz
- Graphics card NVIDIA GeForce RTX 3060 laptop GPU GDDR6 @ 6GB (192bit)
- OS name: Ubuntu 22.04.1 LTS inside WSL2
- OS type: 64-bit

Table 3.1 Key results with DNN model

Algorithm	Calculation time, min:sec	Accuracy	Best Trial №
Random search	14:58	0.666	22
TPE	29:47	0.671	12
Annealing	14:27	0.675	49
Harmony search	21:20	0.671	40
Genetic search	17:27	0.666	45
PSO	16:45	0.665	32

DNN trials optimization history and parameters importance for each algorithm are presented in Appendix B.1.

Table 3.2 Key results with GRU model

Algorithm	Calculation time, min:sec	Accuracy	Best Trial №
Random search	21:47	0.955	6
TPE	19:46	0.96	20
Annealing	18:52	0.95	1
Harmony search	24:11	0.95	26
Genetic search	24:35	0.963	12
PSO	20:08	0.951	7

GRU trials optimization history and parameters importance for each algorithm are presented in Appendix B.2.

CONCLUSIONS

Literature analysis regarding HP stochastic optimization was provided. As can be seen, there is a bunch of studies exploring specific stochastic or natural computing algorithm for some specific problem. Namely, in this paper 6 hyperparameter approaches were investigated: Grid Search, Random Search, Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Harmony Search (HS) and Artificial Bee Colony (ABC) algorithm. It worth to mention that in most of the papers the methods were not implemented as it is, because of the heterogeneity of the HP search space. For some of algorithms authors modify inner structure of the algorithm (GA, ABC, HS), thus binary or categorical variables can be converted forward and backward; for PSO even this was not possible, and mapping to the model accuracy using Bayesian Optimization Framework was added. There was only one paper that includes a comparison between methods, but here a bigger emphasis was on the specific task, and there was no valid clue that the test setup was fully standardized, thus the results could be transfer to another task. Moreover, many studies apply modifications to the methods like conditions and inner assessment during search to not investigate potentially dead ends or aggregations of intermediate results that can decrease resource consumption etc. Nevertheless, there is still a lack of compound studies that could show the valid picture on effectiveness of the stochastic algorithms set, and give an abundant comparison analysis, what is defined as the main objective for this proceeding research.

As a result of this study, two algorithms were implemented, PSO and Harmony search, one algorithm was modified, Simulated Annealing, and corresponding framework Optuna was extended to satisfy PSO implementation requirements. Overall, 6 algorithms were compared: random search, tree-structured Parzen estimator (TPE); simulated annealing, particle-swarm optimization, harmony search, as well as genetic search algorithms. Moreover, all of them were run and implemented within one framework, what allows valid evaluation between them. The only one non-stochastic algorithm here is TPE, that was taken as a benchmark to compare with and because it is implemented in most of wide used HP optimization frameworks. Task to compare

on is aspect-based sentiments analysis, and the models, that HP are optimized for, are DNN and GRU. Parameters and experiment setup for each model was done independently.

In the first experiment with DNN model, simulated annealing is overperformed others with 67.5 % accuracy score and shortest computational time; what is also significant, best score was achieved at almost last trial, however during the study its trials have gone to plateau. The second-best algorithms with DNN HP optimization are Harmony search and TPE with equal score of 67.1 %, while in terms of time harmony search is better, and both algorithms got to relative plateau since the 10th trial. These plateaus in each study are caused by pruning feature that simply abort all dead-end trials. Thus, stochastic algorithms showed good results, although non-stochastic one also managed to get into the top three. It is important to mention that for almost every algorithm, there was one HP that took the highest importance of appx. 50%, and only for Genetic search there are 2 of such hyper-parameters, number of units within layers and learning rate, with 39 % and 38 % of importance score respectively. Moreover, for most of algorithms the most important parameters were the last two, for some number of epochs also plays a role, and what's interesting, for simulated annealing the most important parameter was number of layers.

In the second experiment, GRU model surely showed much better result because of the higher proficiency of the model itself. Nevertheless, this time genetic search is overperformed others with 96.3 % accuracy score but longest though still good enough computational time; best score was achieved in the middle of the study, on 12th trial, and after got to plateau. The second-best algorithm is non-stochastic TPE with 96 % score and almost best computational time. All other algorithms performed almost the same, their scores were fluctuated around 95 %, while some got to plateau already from the first trials. It is possible, that in this situation, trials number can simply be increased to break the initial accuracy threshold. Regarding HP importance, there are 2-3 major HPs in each algorithm, among which number of epochs and units, dropout probability, learning rate and sometimes optimizer. There are 2 algorithms, PSO and Genetic

search, that have parameters significantly predominant in importance, around 80 %, number of epochs and learning rate respectively. Also, importance distribution for simulated annealing caught an eye, because here number of units and epochs got same score of 38 %. Parameter importance can be extremely useful, if the resources are restricted, model is heavy, and we need to decrease our search space considering first restricted number of trials.

Overall, in both experiments stochastic algorithms performed more than satisfactorily, de facto overperformed non-stochastic algorithm, although the latter also showed good results. As an outcome of this research, harmony search, genetic search as well as simulated annealing, can be recommended for use in real world applications.

ВИСНОВКИ

Проведено аналіз літератури щодо стохастичної оптимізації гіперпараметрів (HP). Як видно, існує купа досліджень, що досліджують конкретний стохастичний або природний обчислювальний алгоритм для певної конкретної проблеми. А саме, у цій роботі було досліджено 6 підходів: пошук по сітці, випадковий погук, генетичний алгоритм (GA), оптимізація роя частинок (PSO), гармонічний пошук (HS) та алгоритм штучної бджолиної колонії (ABC). Варто зазначити, що в більшості статей методи не були реалізовані як є, через неоднорідність простору пошуку HP. Для деяких алгоритмів автори змінюють внутрішню структуру алгоритму (GA, ABC, HS), таким чином двійкові або категоріальні змінні можна перетворювати вперед і назад; для PSO навіть це було неможливо, і було додано відображення точності моделі за допомогою Байєсівської оптимізації. Була лише одна стаття, яка містить порівняння між методами, але тут більший акцент був зроблений на конкретному завданні, і не було представлено, що налаштування тесту було повністю стандартизовано, тому результати можна було б перенести на інше завдання. Більше того, багато досліджень вносять модифікації до методів такі, як умови та внутрішня оцінка під час пошуку, щоб не досліджувати потенційно тупикові напрямки або агрегація проміжних результатів, які можуть зменшити споживання ресурсів тощо. Тим не менш, досі бракує повноцінних досліджень, які могли б показати достовірну картину, щодо ефективності набору стохастичних алгоритмів та дати повний порівняльний аналіз, що визначено як основне завдання дослідження.

У результаті цього дослідження було реалізовано два алгоритми, PSO та Harmony search, один алгоритм був модифікований, моделювання відпалу, і відповідний фреймворк Optuna було розширено, щоб задовольнити вимоги реалізації PSO. Загалом було порівняно 6 алгоритмів: випадковий пошук, tree-structured Parzen estimator (TPE); моделювання відпалу, оптимізація роїв частинок, гармонійний пошук, а також алгоритм генетичного пошуку. Крім того, усі вони були запущені та реалізовані в одній структурі, що дозволяє проводити

достовірну оцінку між ними. Єдиним нестохастичним алгоритмом є TPE, який було взято як еталон для порівняння, оскільки він реалізований у більшості широко використовуваних систем оптимізації HP. Завданням для порівняння є класифікація почуттів відносно сутностей, а моделями, для яких оптимізувалися гіпер-параметри, є DNN і GRU. Параметри та налаштування експерименту для кожної моделі проводились окремо.

У першому експерименті з моделлю DNN імітований відпал перевершує інші алгоритми з показником точності 67,5 % і найкоротшим часом обчислення; що також важливо, найкращий результат був досягнутий під час передостаннього випробування, однак під час дослідження експеримент вийшов на плато. Другими найкращими алгоритмами з оптимізацією гіперпараметрів (HP) DNN є Harmony search і TPE з рівним результатом у 67,1%, тоді як з точки зору часу гармонійний пошук є кращим, і обидва алгоритми досягли відносного плато після 10-го випробування. Ці плато в кожному дослідженні викликані функцією чистки, яка просто припиняє всі безперспективні випробування. Таким чином, стохастичні алгоритми показали хороші результати, хоча нестохастичний також зміг потрапити до трійки лідерів. Важливо зазначити, що майже для кожного алгоритму був один HP, який мав найвищу вагу у приблизно 50%, і лише для генетичного пошуку є 2 таких гіперпараметра, кількість нейронів у шарах і швидкість навчання, з оцінкою важливості 39% і 38% відповідно. Крім того, для більшості алгоритмів найважливішими параметрами були два вже зазначених, для деяких алгоритмів грає роль і кількість епох, і що цікаво, для моделювання відпалу найважливішим параметром була саме кількість шарів.

У другому експерименті модель GRU, безсумнівно, показала набагато кращий результат через більш специфічну та складну структуру самої моделі. Тим не менш, цього разу генетичний пошук перевершує інші алгоритми з показником точності 96,3 %, але в той же час це алгоритм мав найдовший, але все ще достатньо хороший час обчислення; найкращий результат було досягнуто в середині дослідження, під час 12-го випробування, і після був вихід на плато.

Другим найкращим алгоритмом є нестохастичний TPE з результатом 96 % і майже найкращим часом обчислення. Усі інші алгоритми працювали майже однаково, їх показники коливалися в районі 95%, а деякі вийшли на плато вже з перших випробувань. Можливо, що в цій ситуації кількість проб можна просто збільшити, щоб подолати початковий поріг точності. Щодо важливості HP, у кожному алгоритмі є 2-3 основні HP, серед яких кількість епох і нейронів у шарі, ймовірність вибуття шару (dropout), швидкість навчання та іноді оптимізатор. Існує 2 алгоритми, PSO і генетичний пошук, які мають параметри, які значно переважають за важливістю інші в цих алгоритмах, близько 80%, кількість епох і швидкість навчання відповідно. Крім того, кинувся в очі розподіл важливості для імітованого відпалу, оскільки тут кількість нейронів і епох отримала однакову оцінку у 38%. Важливість параметра може бути надзвичайно корисною, якщо ресурси обмежені, модель важка, та необхідно зменшити простір пошуку, враховуючи першу обмежену кількість випробувань.

Загалом, в обох експериментах стохастичні алгоритми працювали більш ніж задовільно, де-факто випереджаючи нестохастичний алгоритм, хоча останній також показав хороші результати. Як результат цього дослідження, гармонійний пошук, генетичний пошук, а також моделювання відпалу можуть бути рекомендовані для використання на реальних, прикладних задачах.

Апробація роботи відбувалася на спільній українсько-китайській онлайн-конференції 1st Student Scientific Conference of Joint Research Cooperation between Odessa I.I. Mechnikov National University and Huaiyin Institute of Technology [19].

REFERENCES

1. Spall, J.. (2007). Introduction to Stochastic Search and Optimization. Estimation, Simulation, and Control. Neural Networks, IEEE Transactions on. 18. 964-965. 10.1109/TNN.2007.897481.
2. Natural Computing, An international Journal n.d., https://www.springer.com/journal/11047.
3. Corne, D., Deb, K., Knowles, J. and Yao, X., 2011. Selected applications of natural computing. *Handbook of natural computing*.
4. Kim, J.Y. and Cho, S.B., 2019, June. Evolutionary optimization of hyperparameters in deep learning models. In *2019 IEEE Congress on Evolutionary Computation (CEC)* (pp. 831-837). IEEE.
5. Bergstra, J.; Bengio, Y. Random Search for Hyper-Parameter Optimization. J. Mach. Learn. Res. 2012, 13, 281–305.
6. Li, C., Jiang, J., Zhao, Y., Li, R., Wang, E., Zhang, X. and Zhao, K., 2021. Genetic Algorithm based hyper-parameters optimization for transfer Convolutional Neural Network. *arXiv preprint arXiv:2103.03875*.
7. M. Suganuma, S. Shirakawa, T. Nagao. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. In: GECCO, 2017, pp. 497.
8. Woo-Young Lee, Seung-Min Park, Kwee-Bo Sim, Optimal hyperparameter tuning of convolutional neural networks based on the parameter-setting-free harmony search algorithm, Optik, Volume 172, 2018, Pages 359-367, ISSN 0030-4026.
9. Li, Y. and Zhang, Y., 2020. Hyper-parameter estimation method with particle swarm optimization. *arXiv preprint arXiv:2011.11944*.
10. Leila Zahedi, Farid Ghareh Mohammadi and M. Hadi Amini HyP-ABC: A Novel Automated Hyper-Parameter Tuning Algorithm Using Evolutionary Optimization TechRxiv, 2021

11. Zahedi, Leila & Ghareh Mohammadi, Farid & Amini, M. H.. (2021). OptABC: an Optimal Hyperparameter Tuning Approach for Machine Learning Algorithms. 10.1109/ICMLA52953.2021.00186.
12. Elgeldawi, Enas & Sayed, Awny & Galal, Ahmed & Zaki, Alaa. (2021). Hyperparameter Tuning for Machine Learning Algorithms Used for Arabic Sentiment Analysis. Informatics. 8. 10.3390/informatics8040079.
13. https://optuna.org/
14. K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.
15. https://spacy.io/usage/processing-pipelines/
16. https://pytorch.org/
17. https://miro.medium.com/max/1400/1*UxZ0pTQW8kofL9bzPVYV1w.webp
18. Bergstra, James & Bardenet, R. & Kégl, Balázs & Bengio, Y.. (2011). Algorithms for Hyper-Parameter Optimization.
19. Vystorobska L., Strakhov Ye., On the effectiveness analysis of Random search optimization algorithms in machine learning, 1st Student Scientific Conference of Joint Research Cooperation between Odessa I.I. Mechnikov National University and Huaiyin Institute of Technology: proceedings of the conference, May 16, 2022.

APPENDIX A

CODE SOURCES

A.1 Study class added function for PSO

```
def best_trial_from_last_n(self, last_n: int = 5) -> FrozenTrial:
        """Return the best trial from last N in the study.
        Returns:
            A :class:`~optuna.FrozenTrial` object of the best trial.
        Raises:
            :exc:`RuntimeError`:
                If the study has more than one direction.
        """
        if self._is_multi_objective():
            raise RuntimeError(
                "A single best trial cannot be retrieved from a multi-objective study. Consider "
                "using Study.best_trials to retrieve a list containing the best trials."
            )
        return copy.deepcopy(
               self._storage.get_best_trial_from_last_n(
               self._study_id, last_n=last_n
          ))
```

A.2 Storage class added function for PSO

```
def get_best_trial_from_last_n(self, study_id: int, last_n: int) -> FrozenTrial:
        """Return the trial with the best value in a sub-set of study.
        This method is valid only during single-objective optimization.
        Args:
            study_id:
                ID of the study.
```

```
            last_n:

                number of trials from tail to select the best one
from

        Returns:

            The trial with the best objective value among sub-set
of finished trials in the study.

        Raises:

            :exc:`KeyError`:

                If no study with the matching ``study_id`` exists.

            :exc:`RuntimeError`:

                If the study has more than one direction.

            :exc:`ValueError`:

                If no trials have been completed.

        """

        all_trials = self.get_all_trials(study_id, deepcopy=False)

        all_trials = [t for t in all_trials[-last_n:] if t.state is
TrialState.COMPLETE]

        if len(all_trials) == 0:

            raise ValueError("No trials are completed yet.")

        directions = self.get_study_directions(study_id)

        if len(directions) > 1:

            raise RuntimeError(

                "Best trial can be obtained only for single-
objective optimization."

            )

        direction = directions[0]

        if direction == StudyDirection.MAXIMIZE:

            best_trial = max(

                    all_trials, key=lambda t: cast(float, t.value))

        else:

            best_trial = min(

                    all_trials, key=lambda t: cast(float, t.value))


        return best_trial
```

A.3 Simulated Annealing Sampler

```
class SimulatedAnnealingSampler(BaseSampler):
    def __init__(self, temperature=100, cooldown_factor=0.9,
neighbor_range_factor=0.1, seed=None):
        self._rng = np.random.RandomState(seed)
        self._independent_sampler =
optuna.samplers.RandomSampler(seed=seed)
        self._temperature = temperature
        self.cooldown_factor = cooldown_factor
        self.neighbor_range_factor = neighbor_range_factor
        self._current_trial = None


    def infer_relative_search_space(self, study, trial):
        return optuna.samplers.intersection_search_space(study)


    def sample_relative(self, study, trial, search_space):
        if search_space == {}:
            return {}
        prev_trial = self._get_last_complete_trial(study)
        if self._rng.uniform(0, 1) <=
self._transition_probability(study, prev_trial):
            self._current_trial = prev_trial
        params = self._sample_neighbor_params(search_space)
        self._temperature *= self.cooldown_factor
        return params


    def _sample_neighbor_params(self, search_space):
        params = {}
        for param_name, param_distribution in
search_space.items():
            if isinstance(param_distribution,
distributions.CategoricalDistribution):
                neighbor_low = neighbor_high = None
            else:
```

```
                current_value =
self._current_trial.params[param_name]
                width = (
                    param_distribution.high -
param_distribution.low
                ) * self.neighbor_range_factor
                neighbor_low = max(current_value - width,
param_distribution.low)
                neighbor_high = min(current_value + width,
param_distribution.high)
                params[param_name] =
self._rng.uniform(neighbor_low, neighbor_high)


        return params


    def _transition_probability(self, study, prev_trial):
        if self._current_trial is None:
            return 1.0


        prev_value = prev_trial.value
        current_value = self._current_trial.value


        if study.direction == StudyDirection.MINIMIZE and
prev_value <= current_value:
            return 1.0
        elif study.direction == StudyDirection.MAXIMIZE and
prev_value >= current_value:
            return 1.0
        return np.exp(-abs(current_value - prev_value) /
self._temperature)


    @staticmethod
    def _get_last_complete_trial(study):
        complete_trials = study.get_trials(deepcopy=False,
states=[TrialState.COMPLETE])
        return complete_trials[-1]
```

```
    def sample_independent(self, study, trial, param_name,
param_distribution):


        return self._independent_sampler.sample_independent(
            study, trial, param_name, param_distribution
        )
```

A.4 PSO Sampler

```
_GENERATION_KEY = "pso:generation"


class ParticleSwarmSampler(BaseSampler):


    def __init__(
        self,
        *,
        particles_num: int = 5,
        inertia_w: float = 0.5,
        cognitive_coef: float = 0.2,
        social_coef: float = 0.3,
        speed_max: float = 0.9,
        seed: Optional[int] = None,
        constraints_func: Optional[Callable[[FrozenTrial],
Sequence[float]]] = None,
        particles_velocities: Optional[Dict[str,
BaseDistribution]] = None
    ) -> None:


        if not isinstance(particles_num, int):
            raise TypeError("`particles_num` must be an integer
value.")


        if particles_num < 2:
            raise ValueError("`particles_num` must be greater than
or equal to 2.")
```

```
        if not (inertia_w is None or 0.0 <= inertia_w <= 1.0):
            raise ValueError(
                "`inertia_w` must be None or a float value within
the range [0.0, 1.0]."
            )

        if not (0.0 <= cognitive_coef <= 2.0):
            raise ValueError("`cognitive_coef` must be a float
value within the range [0.0, 2.0].")

        if not (0.0 <= social_coef <= 2.0):
            raise ValueError("`social_coef` must be a float value
within the range [0.0, 2.0].")

        if constraints_func is not None:
            warnings.warn(
                "The constraints_func option is an experimental
feature."
                " The interface can change in the future.",
                ExperimentalWarning,
            )

        self._particles_num = particles_num
        self._inertia_w = inertia_w
        self._cognitive_coef = cognitive_coef
        self._social_coef = social_coef
        self._speed_max = speed_max
        self._random_sampler = RandomSampler(seed=seed)
        self._rng = np.random.RandomState(seed)
        self._constraints_func = constraints_func
        self.particles_velocities: Dict[str, Dict[str: float]]
        self.best_in_generation = None
        self.best_in_population = None
        self.generation_num = -1
```

```
        self.particles_velocities = {}
        self._independent_sampler =
optuna.samplers.RandomSampler(seed=seed)


    def reseed_rng(self) -> None:
        self._random_sampler.reseed_rng()
        self._rng = np.random.RandomState()


    def infer_relative_search_space(self, study, trial):
        return optuna.samplers.intersection_search_space(study)


    def sample_relative(
        self,
        study: Study,
        trial: FrozenTrial,
        search_space: Dict[str, BaseDistribution],
    ) -> Dict[str, Any]:
        trial_id = trial._trial_id
        previous_generation = self._collect_previous_squad(study)
        params = {}


        if self.best_in_population is None or previous_generation
is None:
            generation = self.generation_num = 0
            study._storage.set_trial_system_attr(trial_id,
_GENERATION_KEY, generation)
            for param_name, param_distribution in
search_space.items():
                params.update({param_name:
self._random_sampler.sample_independent(
                    study, trial, param_name, param_distribution
                    )})
                init_random_velocity = np.random.ranf(1)[0]
                if trial_id in self.particles_velocities:

self.particles_velocities[trial_id].update({param_name:
```

```
(init_random_velocity
                                                        if
init_random_velocity <= self._speed_max
                                                        else
self._speed_max)})
                else:
                    self.particles_velocities[trial_id] =
{param_name:

(init_random_velocity
                                                        if
init_random_velocity <= self._speed_max
                                                        else
self._speed_max)}
            return params



        generation = self.generation_num + 1
        study._storage.set_trial_system_attr(trial_id,
_GENERATION_KEY, generation)


        previous_particle_params = previous_generation.params
        pbest_params = self.best_in_population.params
        gbest_params = self.best_in_population.params
        previous_particle_velocities =
self.particles_velocities.get(previous_generation._trial_id)


        for param_name, param_distribution in
search_space.items():
            previous_param =
previous_particle_params.get(param_name, None)
            pbest_param = pbest_params.get(param_name, None)
            gbest_param = gbest_params.get(param_name, None)
            previous_param_velocity =
previous_particle_velocities.get(param_name, None)
```

```
            if isinstance(param_distribution,
distributions.CategoricalDistribution):
                previous_param_transformed =
float(param_distribution.to_internal_repr(previous_param))
                pbest_param_transformed =
float(param_distribution.to_internal_repr(pbest_param))
                gbest_param_transformed =
float(param_distribution.to_internal_repr(gbest_param))
                new_velocity = (self._inertia_w *
previous_param_velocity +

self._cognitive_coef*np.random.ranf(1)[0]*(pbest_param_transformed
- previous_param_transformed) +

self._social_coef*np.random.ranf(1)[0]*(gbest_param_transformed-
previous_param_transformed))
                new_param_transformed = previous_param_transformed
+ new_velocity
                new_param_transformed =
((len(param_distribution.choices) - 1)
                                                if
new_param_transformed > (len(param_distribution.choices) - 1)
                                                else
new_param_transformed)
                if trial_id in self.particles_velocities:

self.particles_velocities[trial_id].update({param_name:
new_velocity})
                else:
                    self.particles_velocities[trial_id] =
{param_name: new_velocity}
                params[param_name] =
param_distribution.to_external_repr(new_param_transformed)
            else:
                previous_param_transformed =
transform._transform_numerical_param(previous_param,
param_distribution, transform_log=True)
                pbest_param_transformed =
transform._transform_numerical_param(pbest_param,
param_distribution, transform_log=True)
```

```
                gbest_param_transformed =
transform._transform_numerical_param(gbest_param,
param_distribution, transform_log=True)

                new_velocity = (self._inertia_w *
previous_param_velocity +

self._cognitive_coef*np.random.ranf(1)[0]*(pbest_param_transformed
- previous_param_transformed) +

self._social_coef*np.random.ranf(1)[0]*(gbest_param_transformed-
previous_param_transformed))

                new_param_transformed = previous_param_transformed
+ new_velocity

                if trial_id in self.particles_velocities:

self.particles_velocities[trial_id].update({param_name:
new_velocity})

                else:

                    self.particles_velocities[trial_id] =
{param_name: new_velocity}

                params[param_name] =
transform._untransform_numerical_param(new_param_transformed,
param_distribution, transform_log=True)

        return params


    def _collect_previous_squad(self, study: Study) ->
List[FrozenTrial]:

        trials = study.get_trials(deepcopy=False)


        generation_to_runnings = defaultdict(list)

        generation_to_particles = defaultdict(list)

        generation = -1

        for trial in trials:

            if _GENERATION_KEY not in trial.system_attrs:

                continue


            generation = trial.system_attrs[_GENERATION_KEY]

            if trial.state != optuna.trial.TrialState.COMPLETE:

                if trial.state == optuna.trial.TrialState.RUNNING:
```

```
generation_to_runnings[generation].append(trial)
                continue

            generation_to_particles[generation].append(trial)

        previous_generation: List[FrozenTrial] = []
        while True:
            particles = generation_to_particles[generation]
            previous_generation = particles[-1] if particles else
None

            if len(particles) < self._particles_num:
                break

            self.generation_num = self.generation_num + 1
            self.best_in_generation = study.best_trial
            self.best_in_population =
study.best_trial_from_last_n(last_n=self._particles_num)
            break

        return previous_generation

    @staticmethod
    def _get_last_complete_trial(study):
        complete_trials = study.get_trials(deepcopy=False,
states=[TrialState.COMPLETE])
        return complete_trials[-1]

    def sample_independent(self, study, trial, param_name,
param_distribution):
        return self._independent_sampler.sample_independent(
            study, trial, param_name, param_distribution
        )
```

A.5 Harmony search Sampler

```
class HarmonySearchSampler(BaseSampler):


    def __init__(
        self,
        *,
        HMCR: float = 0.5,
        PAR: float = 0.2,
        n_triggered_HMCR: int = 0,
        n_triggered_PAR: int = 0,
        band_width: float = 1e-2,
        harmony_memory_size: int = 2,
        max_iter_size: int = 6,
        harmonies_storage:  Optional[List[FrozenTrial]] = None,
        seed: Optional[int] = None,
        harmonies_pull: Optional[Dict[str, FrozenTrial]] = None,
        constraints_func: Optional[Callable[[FrozenTrial],
Sequence[float]]] = None,
    ) -> None:


        if not (HMCR is None or 0.0 <= HMCR <= 1.0):
            raise ValueError(
                "`HMCR` must be None or a float value within the
range [0.0, 1.0]."
            )


        if not (PAR is None or 0.0 <= PAR <= 1.0):
            raise ValueError("`PAR` must be a float value within
the range [0.0, 1.0].")



        if constraints_func is not None:
            warnings.warn(
```

```
                "The constraints_func option is an experimental
feature."
                " The interface can change in the future.",
                ExperimentalWarning,
            )

        self._HMCR = HMCR
        self._PAR = PAR
        self._n_triggered_HMCR = n_triggered_HMCR
        self._n_triggered_PAR = n_triggered_PAR
        self._band_width = band_width
        self._max_iter = max_iter_size
        self._random_sampler = RandomSampler(seed=seed)
        self._rng = np.random.RandomState(seed)
        self._constraints_func = constraints_func
        self.best_harmony = None
        self.worst_harmony = None
        self._independent_sampler =
optuna.samplers.RandomSampler(seed=seed)
        self._harmonies_pull = None
        self._harmony_memory_size = harmony_memory_size
        self._harmonies_storage = harmonies_storage

    def reseed_rng(self) -> None:
        self._random_sampler.reseed_rng()
        self._rng = np.random.RandomState()

    def infer_relative_search_space(self, study, trial):
        return optuna.samplers.intersection_search_space(study)

    def sample_relative(
        self,
        study: Study,
        trial: FrozenTrial,
```

```
        search_space
    ) -> Dict[str, Any]:
        # trial_id = trial._trial_id
        completed_trials_num, random_trial =
self._collect_previous_squad(study)
        # print('completed_trials_num:', completed_trials_num)
        params = {}

        if random_trial is None:
            for param_name, param_distribution in
search_space.items():

params[param_name].append(self._random_sampler.sample_independent(
                    study, trial, param_name, param_distribution
                ))
            return params

        random_trial_params = random_trial.params



        for param_name, param_distribution in
search_space.items():
            self._HMCR =
self.get_current_HMCR(completed_trials_num)
            self._PAR = self.get_current_PAR(completed_trials_num)
            hmcr_rnd = self._rng.uniform(0, 1)
            if self._HMCR < hmcr_rnd:
                self._n_triggered_HMCR += 1
                random_param = random_trial_params.get(param_name,
None)
                if isinstance(param_distribution,
distributions.CategoricalDistribution):
                    rnd_param_transformed =
float(param_distribution.to_internal_repr(random_param))
                    rnd_for_test = self._rng.uniform(0, 1)
                    self._n_triggered_HMCR += 1 if self._PAR <=
rnd_for_test else 0
```

```
                    new_param_transformed = (rnd_param_transformed
                                             if self._PAR <=
rnd_for_test
                                             else
(rnd_param_transformed +

self._band_width*(self._rng.uniform(0, 1) - 0.5)

*(len(param_distribution.choices) - 1)))
                    new_param_transformed =
((len(param_distribution.choices) - 1)
                                             if
new_param_transformed > (len(param_distribution.choices) - 1)
                                             else
new_param_transformed)
                    params[param_name] =
param_distribution.to_external_repr(new_param_transformed)
                else:
                    rnd_param_transformed =
transform._transform_numerical_param(random_param,
param_distribution, transform_log=True)


                    new_param_transformed = (rnd_param_transformed
                                             if self._PAR <=
self._rng.uniform(0, 1)
                                             else
(rnd_param_transformed +

self._band_width*(self._rng.uniform(0, 1) - 0.5)

*(param_distribution.high - param_distribution.low)))
                    params[param_name] =
transform._untransform_numerical_param(new_param_transformed,
param_distribution, transform_log=True)
            else:
                params[param_name] =
(self._random_sampler.sample_independent(
                    study, trial, param_name, param_distribution))


        return params
```

```
    def get_current_HMCR(self, completed_trials_num):
        hmcr =
min(self._n_triggered_HMCR/self._harmony_memory_size, 1)
        if hmcr == 0:
            hmcr = 1/(1 + 1e-7 +
np.exp(np.log(0.01*(self._max_iter-completed_trials_num))))
        if hmcr == 1:
            hmcr = 1/(1 + 1e-7 + np.exp(-
np.log(0.01*(self._max_iter-completed_trials_num))))
        return hmcr


    def get_current_PAR(self, completed_trials_num):
        par = min(self._n_triggered_PAR/self._harmony_memory_size,
1)
        if par == 0:
            par = 1/(1 + 1e-7 +
np.exp(np.log(0.01*(self._max_iter-completed_trials_num))))
        if par == 1:
            par = 1/(1 + 1e-7 + np.exp(-
np.log(0.01*(self._max_iter-completed_trials_num))))
        return par



    def _collect_previous_squad(self, study: Study) -> Tuple[int,
List[FrozenTrial]]:
        # randomly select one of previous trials
        trials = study.get_trials(deepcopy=False)


        completed_trials = []
        running_trials = []
        random_trial = None


        for trial in trials:


            if trial.state != optuna.trial.TrialState.COMPLETE:
```

```
                if trial.state == optuna.trial.TrialState.RUNNING:
                    running_trials.append(trial)
                continue

            completed_trials.append(trial)

        completed_trials.sort(key=lambda t: cast(float,
t.values[0]), reverse=True)
        completed_trials_num = len(completed_trials)

        if completed_trials_num == 0:
            return completed_trials_num, random_trial

        if completed_trials_num == 1:
            return completed_trials_num, completed_trials[-1]

        if completed_trials_num <= self._harmony_memory_size:
            random_trial =
completed_trials[(np.random.randint(completed_trials_num))]
        else:
            self._harmonies_storage =
completed_trials[:self._harmony_memory_size]
            random_trial =
self._harmonies_storage[(np.random.randint(self._harmony_memory_si
ze))]

        return completed_trials_num, random_trial

    @staticmethod
    def _get_last_complete_trial(study):
        complete_trials = study.get_trials(deepcopy=False,
states=[TrialState.COMPLETE])
        return complete_trials[-1]

    def sample_independent(self, study, trial, param_name,
param_distribution):
```

```
return self._independent_sampler.sample_independent(
    study, trial, param_name, param_distribution
)
```

APPENDIX B

Trials intermediate results and parameters importance

B.1 DNN Results

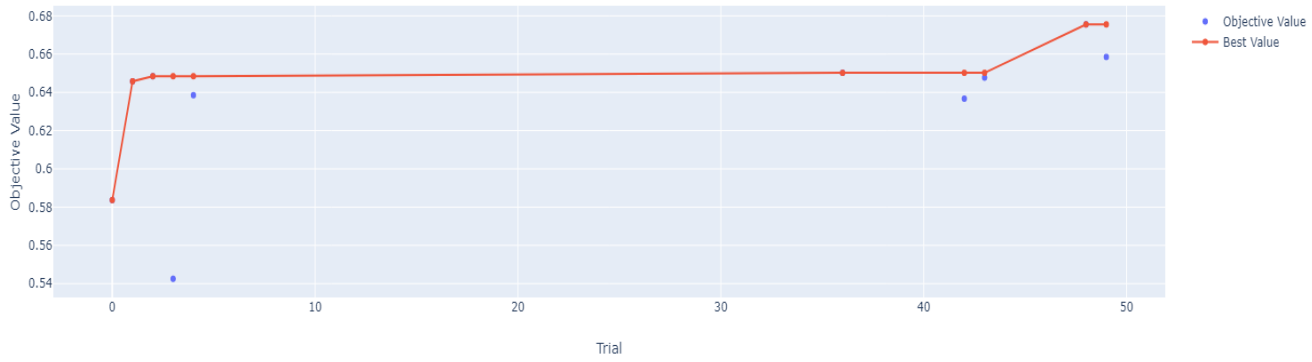

Figure B.1.1 Optimization history for simulated annealing

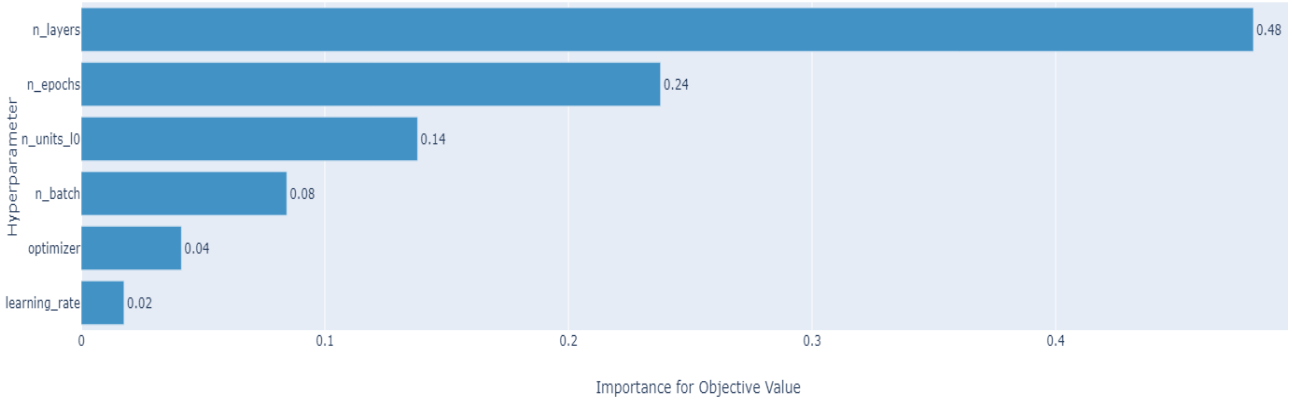

Figure B.1.2 HP importance for simulated annealing

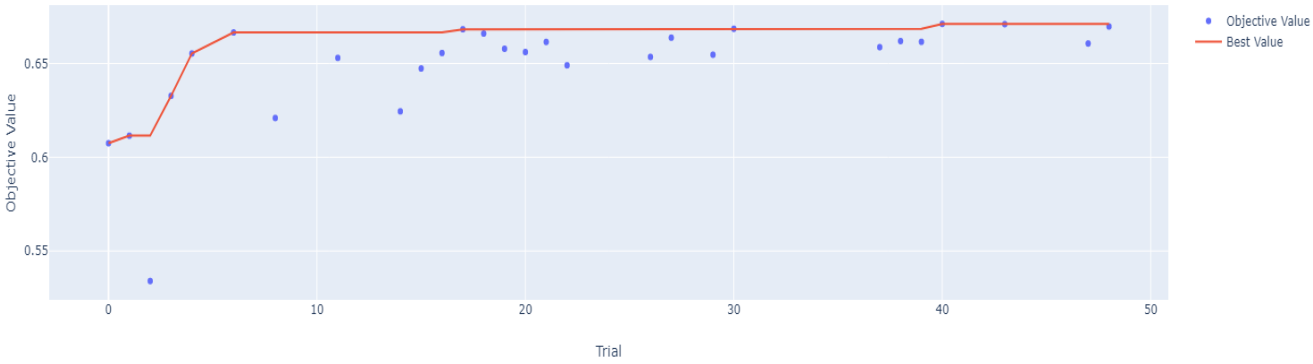

Figure B.1.3 Optimization history for harmony search

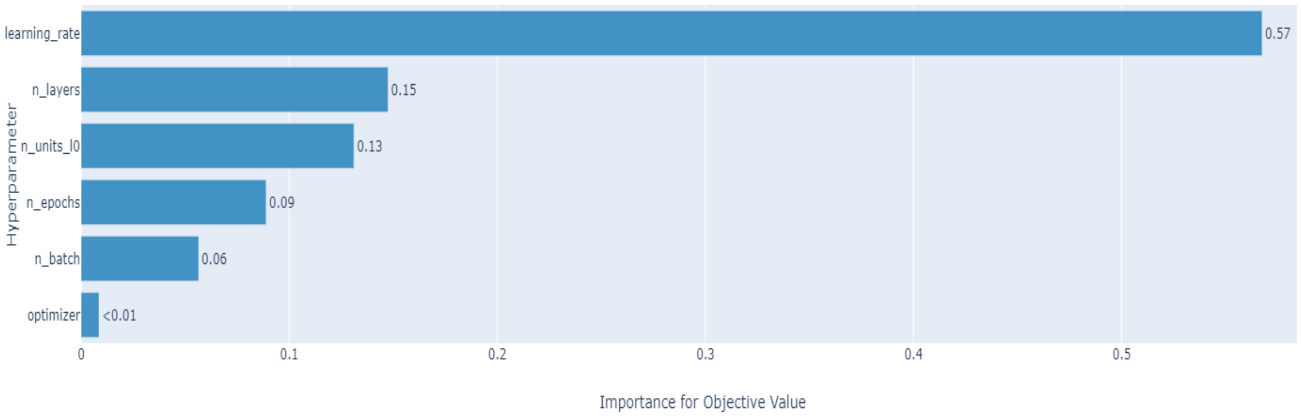

Figure B.1.4 HP importance for harmony search

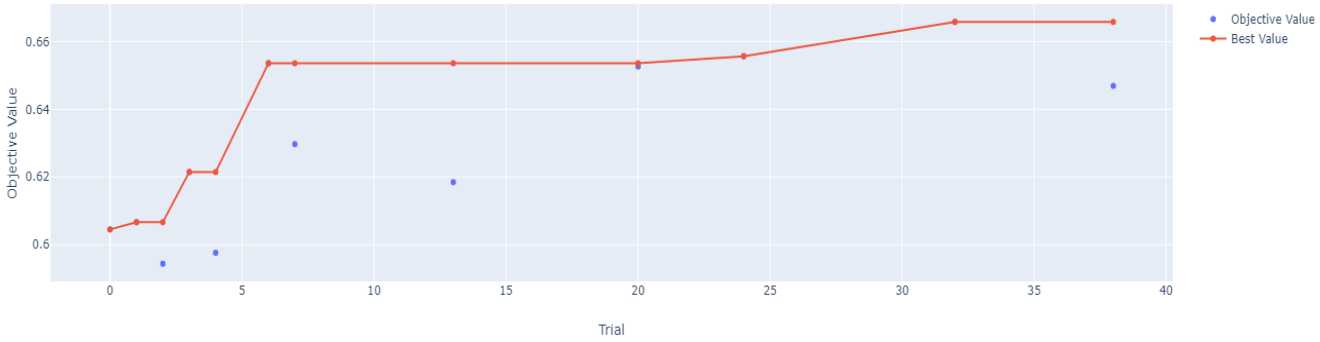

Figure B.1.5 Optimization history for PSO

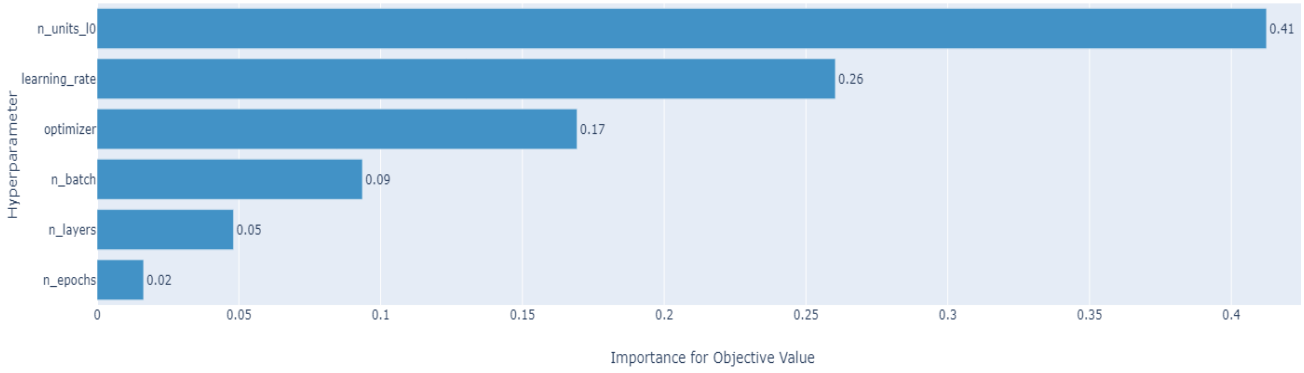

Figure B.1.6 HP importance for PSO

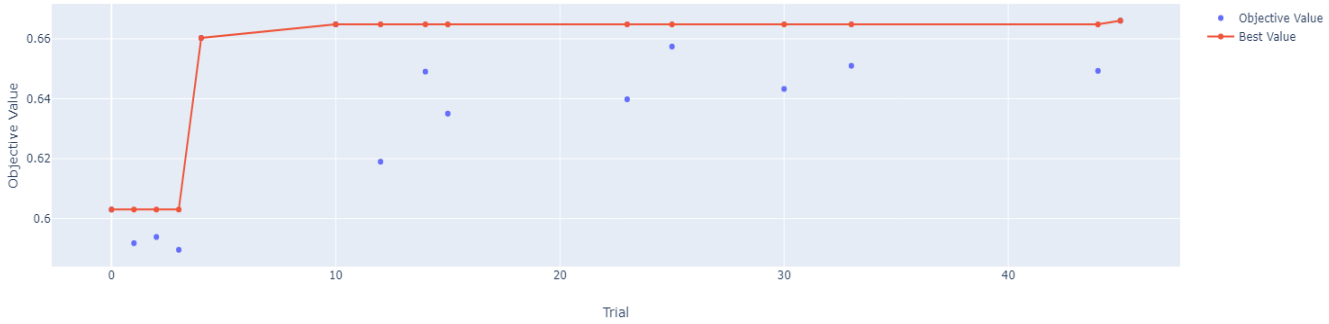

Figure B.1.7 Optimization history for Genetic Search

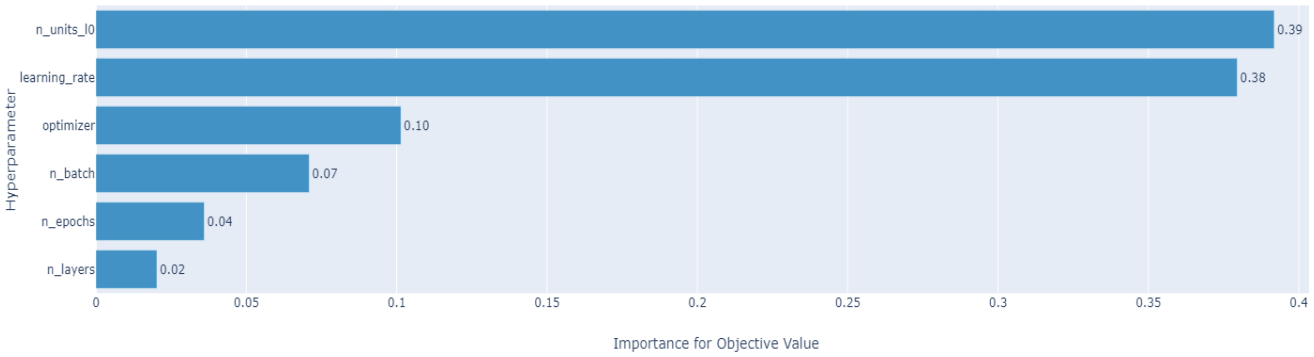

Figure B.1.8 HP importance for Genetic Search

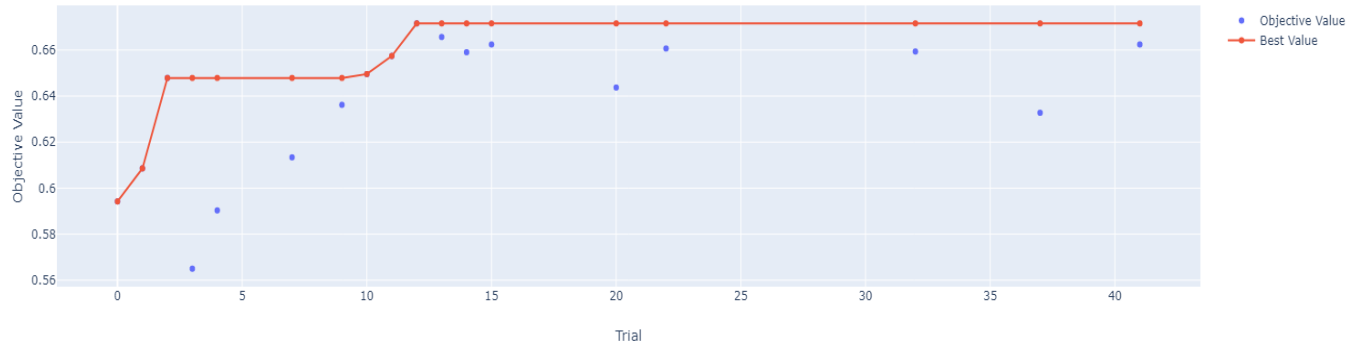

Figure B.1.9 Optimization history for TPE

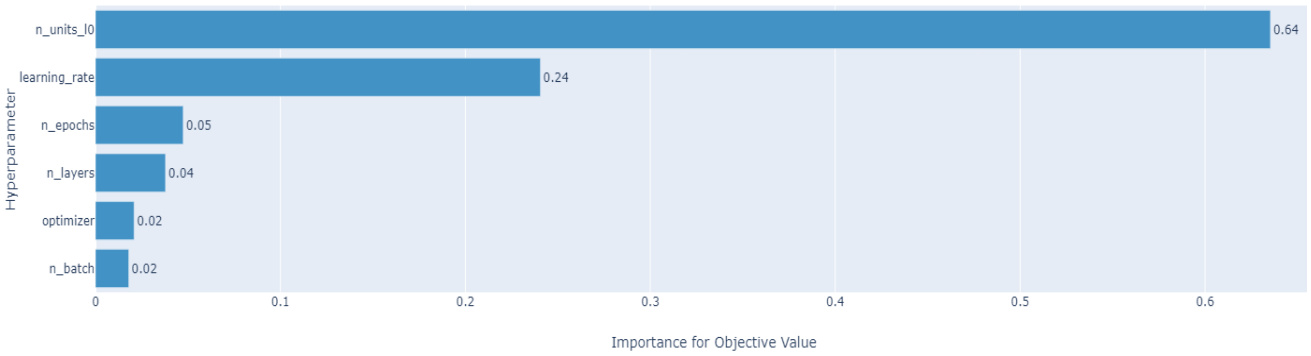

Figure B.1.10 HP importance for TPE

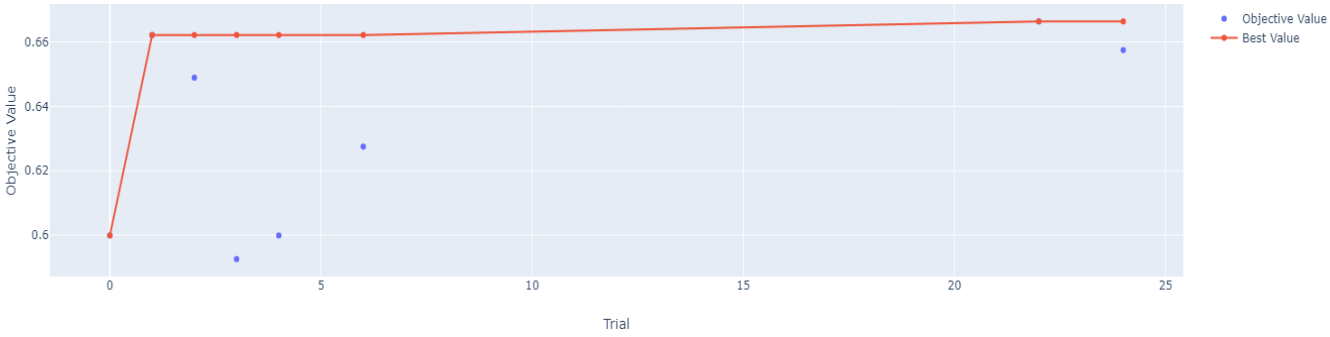

Figure B.1.11 Optimization history for Random search

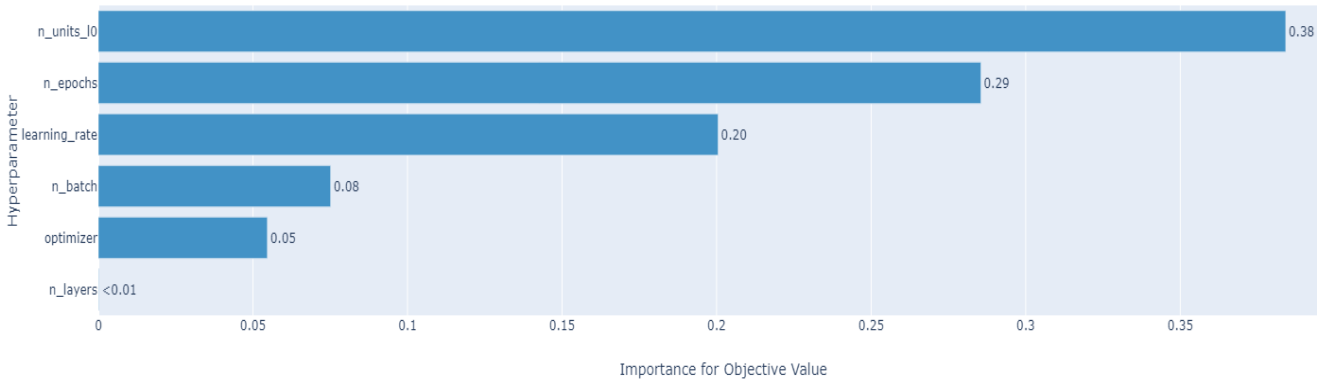

Figure B.1.12 HP importance for Random search

B.2 GRU Results

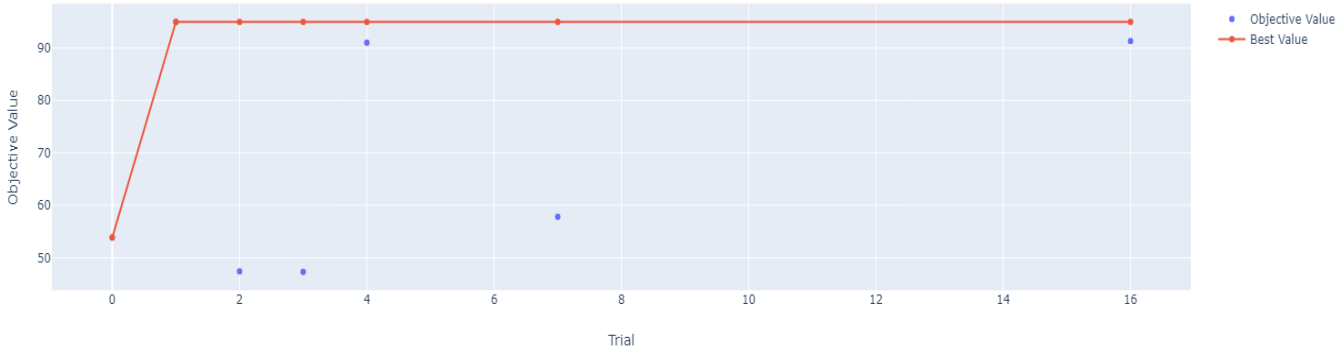

Figure B.2.1 Optimization history for simulated annealing

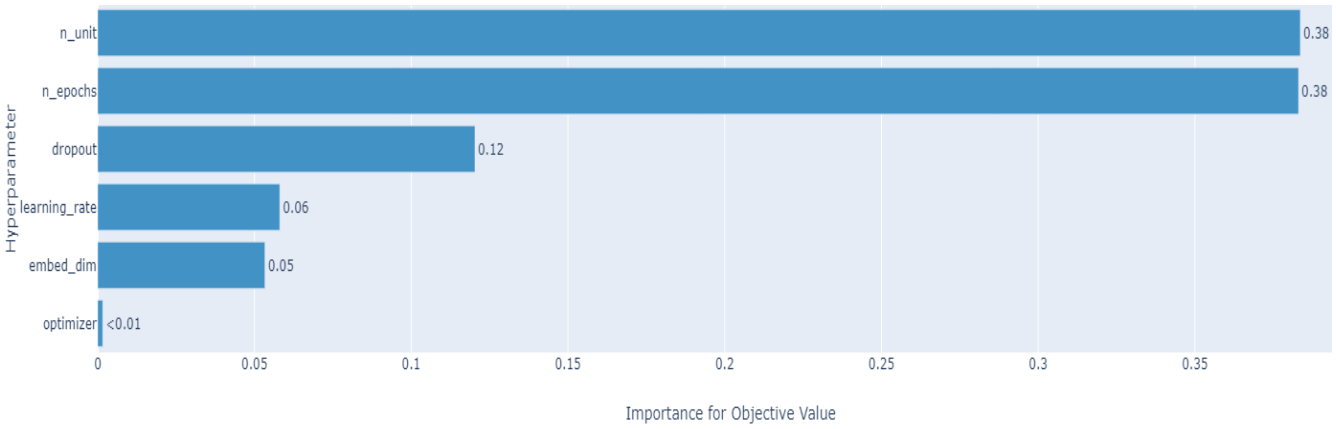

Figure B.2.2 HP importance for simulated annealing

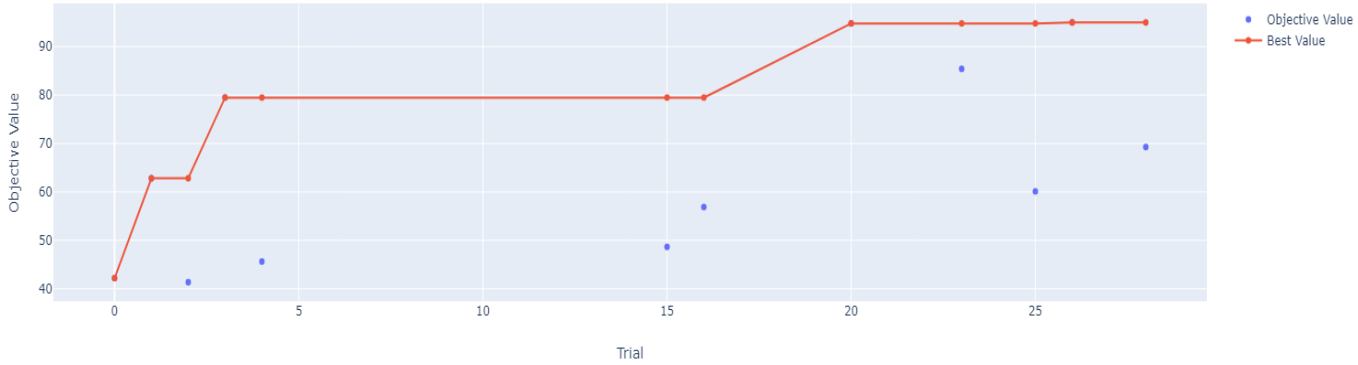

Figure B.2.3 Optimization history for harmony search

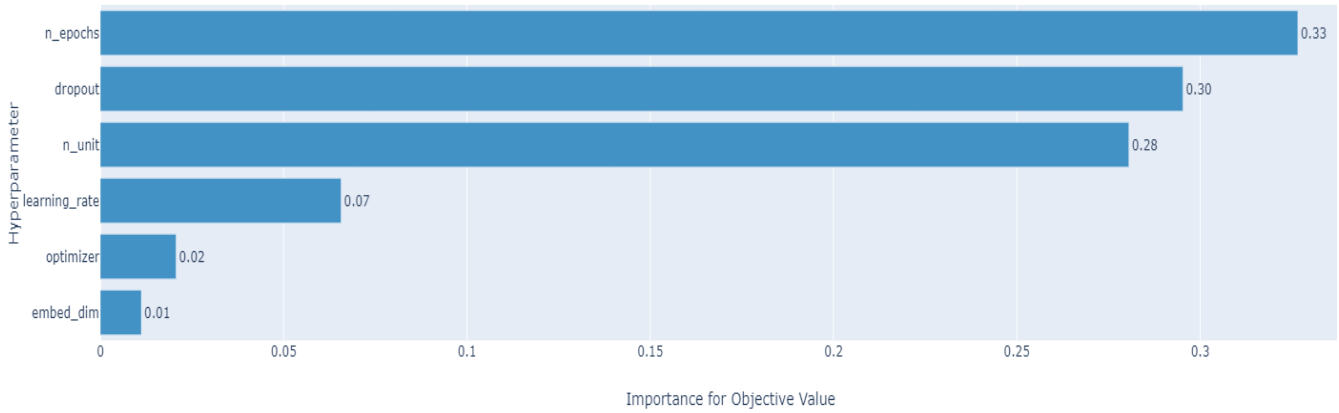

Figure B.2.4 HP importance for harmony search

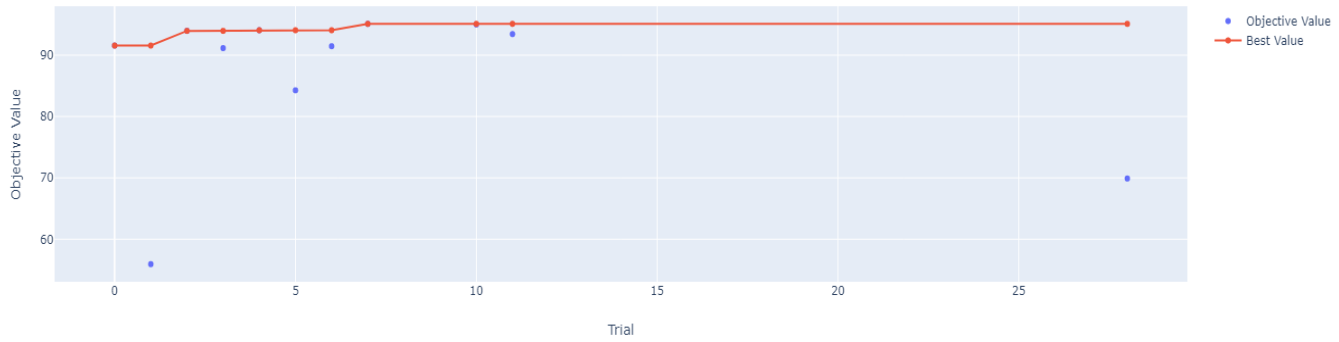

Figure B.2.5 Optimization history for PSO

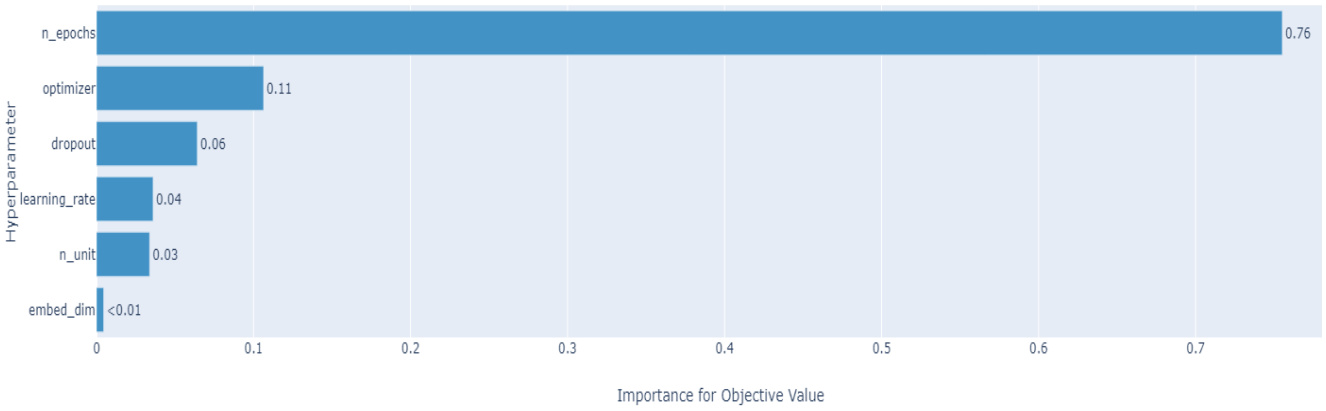

Figure B.2.6 HP importance for PSO

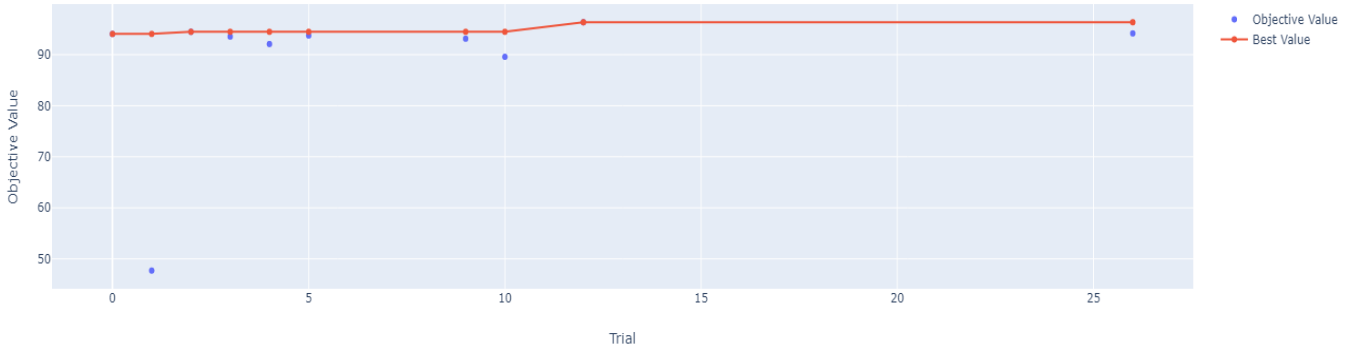

Figure B.2.7 Optimization history for Genetic Search

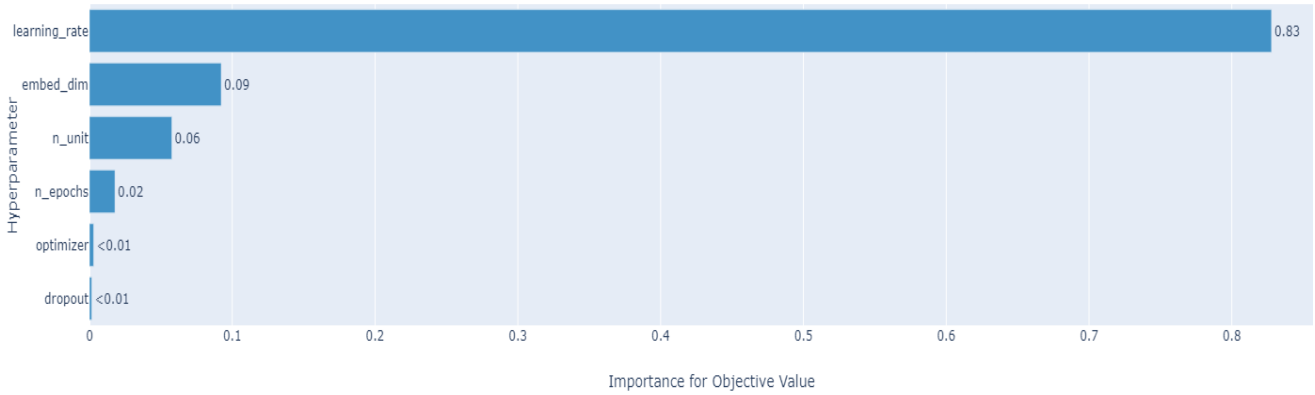

Figure B.2.8 HP importance for Genetic Search

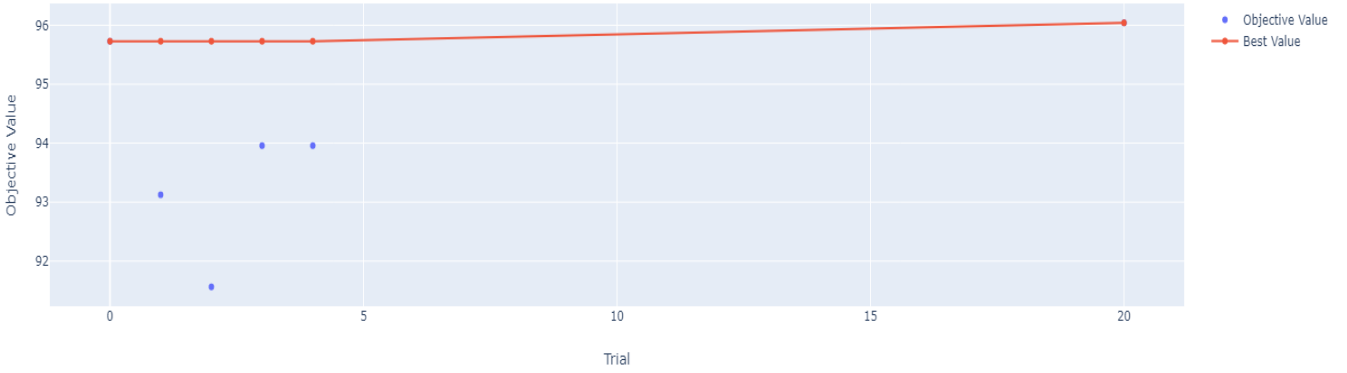

Figure B.2.9 Optimization history for TPE

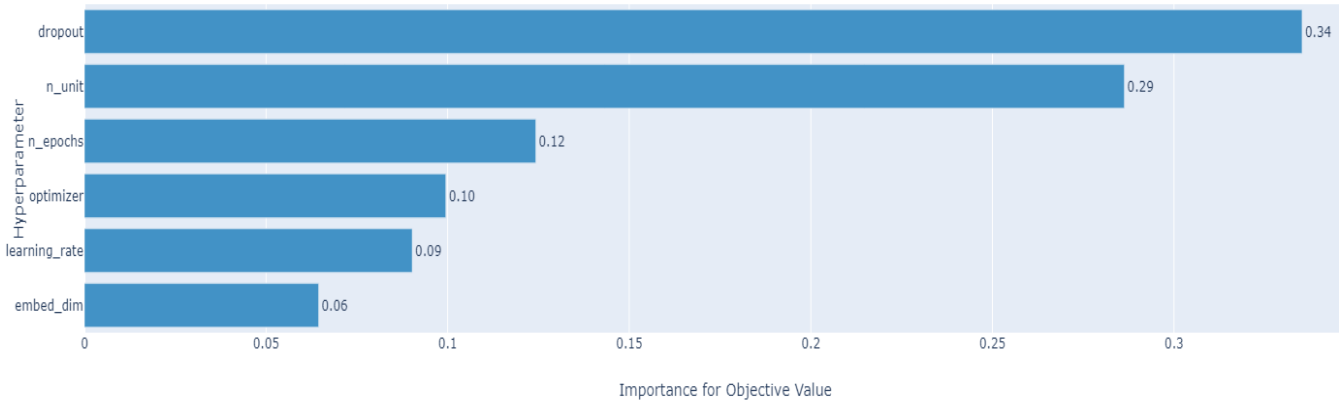

Figure B.2.10 HP importance for TPE

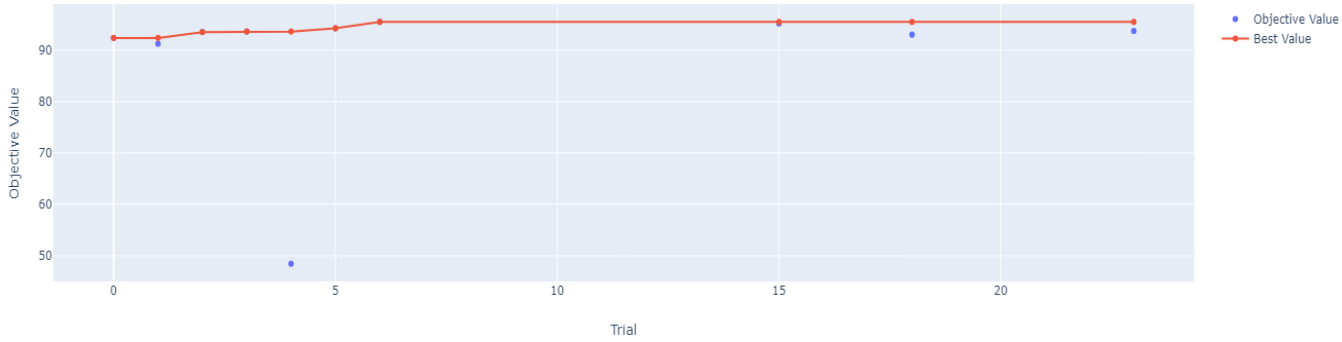

Figure B.2.11 Optimization history for Random search

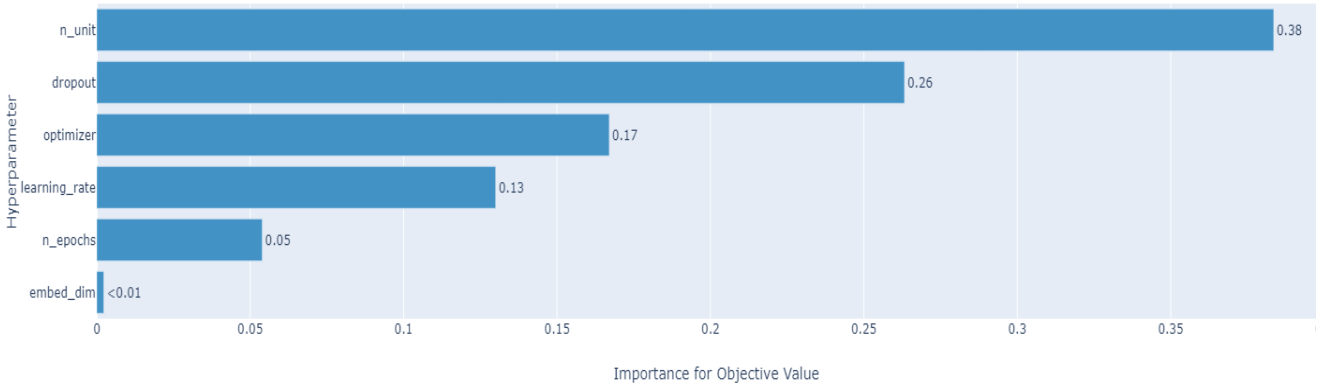

Figure B.2.12 HP importance for Random search