Politechnika Warszawska
WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informatycznych

Heuristic hyperparameter optimization for neural networks

Łukasz Neumann
Numer albumu 261479

promotor
dr hab. inż. Robert M. Nowak

WARSZAWA 2018

Heuristic hyperparameter optimization for neural networks

Abstract

Hyperparameter optimization plays an important role in creation of the robust neural network model. Correctly performed tuning should result in better classification quality, faster convergence of the network's optimizer, as well as smaller tendencies to overfit.

This thesis explores the usage of selected heuristic algorithms: Covariance Matrix Adaptation Evolution Strategy (CMA-ES), Differential Evolution Strategy (DES) and jSO for the hyperparameter optimization problem. Hyperparameters for two architectures are tuned: Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN). Three real-life datasets are used as a basis for the classification task.

Variety of evaluation methods for the hyperparameter optimization process is presented. These methods allow for easier characterization of the tuning process itself, as well as neural networks created using tuned hyperparameters. Moreover, a technique for comparison between different implementations of the same heuristic algorithm is described.

Results indicate, that all three heuristic algorithms can be successfully used as a method for hyperparameter optimization. Notably, similar performance is observed amongst all optimizers. Classifiers created using the best individuals found during the training process can outperform a reference classifier, when such comparison is being made.

Keywords: evolutionary algorithm, optimization, tuning, hyperparameter, neural network

Heurystyczne strojenie hiperparametrów sieci neuronowych

Streszczenie

Optymalizacja wartości hiperparametrów jest jedną z kluczowych części procesu tworzenia wartościowej sieci neuronowej. Poprawnie przeprowadzone strojenie powinno skutkować lepszą jakością klasyfikacji, szybszym zbieganiem metody uczenia sieci do optimum, a także mniejszą podatnością modelu na zbytnie dopasowanie do danych trenujących.

Niniejsza praca opisuje wykorzystanie trzech algorytmów heurystycznych: Covariance Matrix Adaptation Evolution Strategy (CMA-ES), Differential Evolution Strategy (DES) i jSO na potrzeby problemu strojenia hiperparametrów. Badania przeprowadzane zostały dla dwóch architektur: perceptronu wielowarstwowego i splotowej sieci neuronowej. Ocena modeli odbyła się na podstawie klasyfikacji trzech zbiorów danych.

Praca zawiera opis metod ewaluacji procesu strojenia hiperparametrów. Pozwalają one na łatwiejszy opis samej optymalizacji, a także sieci neuronowych stworzonych na podstawie znalezionych wartości hiperparametrów. Dodatkowo przedstawiona jest technika porównania różnych implementacji tego samego algorytmu heurystycznego.

Wyniki pokazują, że wszystkie zbadane algorytmy heurystyczne mogą być z powodzeniem użyte do strojenia hiperparametrów sieci neuronowych. W szczególności, wszystkie optymalizatory cechują się podobną jakością strojenia. Klasyfikatory stworzone na podstawie najlepszych znalezionych zestawów hiperparametrów osiągają lepsze wyniki od klasyfikatora odniesienia, w przypadkach, gdzie takie porównanie miało miejsce.

Słowa kluczowe: algorytm ewolucyjny, optymalizacja, strojenie, hiperparametr, sieć neuronowa

Contents

1 Introduction

In the past few years neural networks became a very popular classification method in the field of machine learning. They have been successfully used in a variety of different settings, such as medicine [1], [2], physics [3] and machine translation [4]. One of the crucial parts of neural network's training is the process of *hyperparameter tuning*, during which user tweaks hyperparameters of a model with fixed architecture (either with pre-trained weights or trained from scratch) to achieve the best performance possible. This process can be both mundane and lead to sub-par quality of the final classifier, as humans do not always have the necessary knowledge to effectively tune the hyperparameters, or lack the ability to do it if the model is complex enough, which can result in high dimensionality of the objective function used in the optimization process.

On the other hand, there is a branch of algorithms that specialize in mathematical optimization, particularly in cases which are computationally challenging, either due to extensive parameterization or to the nature of the optimized function. They are called algorithms, as usually the solution provided as a result of optimization is sub-optimal, while being acceptable for the user.

Combining these two ideas was described in an earlier study [5], which heavily influenced this thesis. Its authors compare one heuristic algorithm (Covariance Matrix Adaptation Evaluation Strategy) with more traditional approaches (described in 1.2). I focus on heuristic algorithms exclusively and test these methods on a selection of various, real-life datasets.

1.1 Main hypothesis

Can Differential Evolution Strategy and jSO algorithms perform on par with Covariance Matrix Adaptation Evaluation Strategy when tuning neural network's hyperparameters?

Detailed hypotheses are described below.

- Do optimization processes differ in their characteristics between tested heuristic algorithms?
- Can optimal hyperparameters' subranges be found, assuming non-Lamarckian approach with fixed, high penalty for boundary crossing?
- How much does complexity of the tuned network influence heuristic algorithms' ability to tune?
- Can heuristic algorithms be used to find values of hyperparameters, which are infeasible for the given classification task?
- Is 'one-shot' optimization viable, as opposed to repeated optimization?

1.2 Related work

Autonomous hyperparameter tuning has been explored for at least 20 years, with several different approaches being developed.

Grid search

Grid search is one of the simplest techniques used for hyperparameter tuning. After user provides sets of values for each hyperparameter, all possible combinations are tested. This approach does not scale well and it requires user to pick specific values for each hyperparameter. Moreover, if user provides value that results in bad performance, all combinations using it will be tested, even if all such combinations are infeasible.

Random search

A slightly different technique, called random search, can be used to achieve much better results. With this approach, user can either provide the set of values, or a distribution for tuned hyperparameters. Furthermore, a budget (i.e. number of model evaluations) should be defined. Random search creates individuals in the following manner: for each hyperparameter, if distribution was provided then sample the distribution, otherwise uniformly sample provided set of values. Next, evaluate the model and save results. This process is repeated until the number of evaluated models reaches set budget. In the end, the best found individual is chosen. While simple, this method shows surprisingly good performance, as described in [6].

Bayesian optimization

This approach uses Bayesian method based on Gaussian processes to tune hyperparameters. The basic idea behind this technique is to use all available information from previous function evaluation to inform the process of choosing new hyperparameters. By their nature these techniques require, that a prior over objective function is chosen, as well as an acquisition function is defined, in order to determine next individual for evaluation. These methods are known to perform well and have been extensively studied. Examples of this approach are TPE [7], SMAC [8], or the works by Snoek et al. [9], [10].

Evolutionary optimization

Solution based on evolutionary optimizations often combine hyperparameter tuning with structure evolution. This approach is called neuroevolution, and has been studied quite extensively over the years. Examples of these strategies are [11], [12], [13]. Usually these solutions are based on modified genetic algorithms, which are tailored to evolve the architecture of the classifier during the course of optimization. However, standard state-of-the-art heuristic optimizers are not widely used for the task of hyperparameter optimization using fixed structure of the neural network, as described in [5]. This research focuses on Evolution Strategies and Differential Evolution to better understand their applicability for hyperparameter tuning.

1.3 Thesis layout

This thesis is divided into 4 chapters.

1. Introduction — outline of the problem, main hypothesis and related work.
2. Methods, datasets and tools — descriptions of used algorithms (heuristics and classifiers), datasets for classification, as well as various statistical tools used to evaluate heuristics and classifiers.

3. Results — verification of algorithms' re-implementations and results of the carried experiments.
4. Summary — conclusions and ideas for future research.

2 Methods, datasets and tools

2.1 Heuristic algorithms

This section contains descriptions of three different heuristic optimizers used to tune hyperparameters. The following symbols are used by convention in all algorithms' listings and their descriptions:

- λ – population size;
- N – dimensionality of the problem;
- t – generation number.

2.1.1 CMA-ES

Covariance Matrix Adaptation Evolution Strategy (CMA-ES)[14] is a state-of-the-art heuristic optimization algorithm. It samples new candidates from a multivariate normal distribution over real numbers. CMA-ES represents this distribution internally by a covariance matrix. Optimization process consists of modifications of the covariance matrix such that sampling of better individuals is achieved. Amongst advantages of CMA-ES are fast convergence and relatively low number of required parameters – user should provide population size (authors suggest usage of $\lambda = 4 + 3 * \log N$), initial individual and standard deviation of the starting population σ.

Pseudo-code for the CMA-ES is provided in Algorithm 1. Main loop of this heuristic revolves around adaptation of three parameters to achieve a normal distribution, which yields better individuals. Modified parameters are: the reference point $\mathbf{m}^t$, the covariance matrix $\mathbf{C}^t$ and the step size σ^t. CMA-ES operates on two sets of points, namely $\mathbf{d}_i^t$, called *base points* and $\mathbf{x}_i^t$ – *individuals*. Base points are sampled from the Gaussian distribution, with mean 0 and described by covariance matrix $\mathbf{C}^t$. Individuals are created as a result of linear transformation of base points. D^t denotes population of λ base points, while D_μ^t contains μ base points used to create the best individuals in the tth generation. To modify covariance matrix, two different operations are combined, namely rank-1 and rank-μ updates. After covariance matrix has been updated, step size is adapted. There are 5 constants used throughout optimization process:

- c_s and c_c – cumulation constants, respectively for step size and covariance matrix;
- c_1 and c_μ – learning rates for rank-1 and rank-μ updates;
- d_σ – damping for step size.

These constants are calculated during initialization of the algorithm, using formulas described in the original paper [14].

Algorithm 1. Covariance Matrix Adaptation Evolution Strategy

Require: $\mathbf{m}^1$ initial solution
Require: σ^1 initial step size multiplier
$\mathbf{p}^1 \leftarrow 0$
$\mathbf{s}^1 \leftarrow 0$
$\mathbf{C}^1 \leftarrow \mathbf{I}$ ▷ Covariance matrix
$t \leftarrow 1$
while !stop **do**
  for $i \leftarrow 1$ **to** λ **do**
    $\mathbf{d}_i^t \sim N(0, \mathbf{C}^t)$
    $\mathbf{x}_i^t \leftarrow \mathbf{m}^t + \sigma^t \mathbf{d}_i^t$
  EVALUATE(X^t)
  sort X^t according to their fitness
  $\mathbf{m}^{t+1} \leftarrow \mathbf{m}^t + \sigma^t \langle D_\mu^t \rangle$
  $\mathbf{s}^{t+1} \leftarrow (1 - c_s)\mathbf{s}^t + \sqrt{\mu c_s (2 - c_s)} \cdot (\mathbf{C}^t)^{-\frac{1}{2}} \langle D_\mu^t \rangle$
  $\mathbf{p}^{t+1} \leftarrow (1 - c_p)\mathbf{p}^t + \sqrt{\mu c_p (2 - c_p)} \cdot \langle D_\mu^t \rangle$
  $\mathbf{C}_1^t \leftarrow (\mathbf{p}^t)(\mathbf{p}^t)^T$ ▷ rank-1 update
  $\mathbf{C}_\mu^t \leftarrow \frac{1}{\mu} \sum_{i=1}^{\mu} (\mathbf{d}_i^t)(\mathbf{d}_i^t)^T$ ▷ rank-μ update
  $\mathbf{C}^{t+1} \leftarrow (1 - c_1 - c_\mu)\mathbf{C}^t + c_1 \mathbf{C}_1^t + c_\mu \mathbf{C}_\mu^t$ ▷ update covariance matrix
  $\sigma^{t+1} \leftarrow \sigma^t \exp\left(\frac{c_s}{d_\sigma}\left(\frac{\|\mathbf{s}^{t+1}\|}{E\|N(0,\mathbf{I})\|} - 1\right)\right)$ ▷ update step size
  $t \leftarrow t + 1$

2.1.2 DES

Differential Evolution Strategy (DES)[15] is a hybrid between CMA-ES and Differential Evolution (DE) class of algorithms. While it does not directly store nor operate on the covariance matrix it uses a number of techniques to generate new individuals resembling these of CMA-ES. DES strives to achieve much better performance on problems with high dimensionality, as it avoids exponential complexity of specific matrix calculations, which CMA-ES incurs. Additionally DES can be a parameter-free method according to its authors, which is a desired feature from this research point of view. Outline of the Differential Evolution Strategy is provided in Algorithm 2. As opposed to CMA-ES, DES does not have a notion of step size. Optimization starts with initialization of the first population X^1, using a uniform distribution over restricted range for each optimized component between set boundaries. Next, main loop of the algorithm is entered, which starts with the evaluation of the population X^t and its midpoint $\langle X^t \rangle$. Following the evaluation, a difference vector δ^t is created by calculating the difference between the midpoint of the whole population $\langle X^t \rangle$ and the midpoint of the best μ points $\langle X_\mu^t \rangle$. This vector is then saved in the $\mathbf{p}^t$ vector. Adaptation of the population is achieved through the repeated mutation of the elite's midpoint X_μ^t. First random values h_1 and h_2 are sampled uniformly over the set $\{1, ..., H\}$, with H being a user-selected constant. The mutation vector $\mathbf{d}^{t+1}$ consists of four different components:

- the difference between two randomly selected individuals from the historical population X^{t-h_1};
- random vector along the direction δ^{t-h_1};

- random vector along the direction $\mathbf{p}^{t-h_2}$;
- sample from the standard Gaussian distribution, scaled by ε.

DES has 4 constants: ε, c_c, c_p and H, which are calculated during the initialization phase, according to the formulas provided by authors in their original work [15].

Algorithm 2. Differential Evolution Strategy

$t \leftarrow 1$
$\mathbf{p}^1 \leftarrow 0$
INITIALIZE(X^1) ▷ Initialize first population
while !stop **do**
  EVALUATE$(X^t, \langle X^t \rangle)$
  $\delta^t \leftarrow \langle X^t_\mu \rangle - \langle X^t \rangle$
  $\mathbf{p}^t \leftarrow (1-c_p)\mathbf{p}^{t-1} + \sqrt{\mu c_p(2-c_p)}\delta^t$
  for $k \leftarrow 1$ **to** λ **do**
    pick at random $h_1, h_2 \in \{1, ..., H\}$
    $j, k \sim \mathcal{U}(1, ..., \mu)$
    $\mathbf{d}_i^{t+1} \leftarrow \sqrt{\frac{c_c}{2}}\left(\mathbf{x}_j^{t-h_1} - \mathbf{x}_k^{t-h_1}\right) + \sqrt{c_c}\delta^{t-h_1} \cdot N(0,1) + \sqrt{1-c_c}\mathbf{p}^{t-h_2} \cdot N(0,1) + \varepsilon \cdot N(\mathbf{0}, \mathbf{I})$
    $\mathbf{x}_i^{t+1} \leftarrow \langle X^t_\mu \rangle + \mathbf{d}_i^{t+1}$
  $t \leftarrow t+1$

2.1.3 jSO

jSO[16] is an algorithm for single-objective continuous optimization. It is an improved version of the iL-SHADE algorithm, both classified as Differential Evolution (DE) algorithms. It has been chosen as a representative of the DE class due to its win in the IEEE Congress on Evolutionary Computation (CEC) competition from 2017 [17]. Notably, jSO has more parameters, which could imply that it is not as generic and easy to use in terms of hyperparameters optimization. It is also the only algorithm with varying population sizes, as both CMA-ES and DES have fixed population size. Initial starting population should contain $\sqrt{N} * \log N$ individuals according to authors.

Algorithm 3 provides an outline of the jSO method. As it is an example of a DE class of algorithms its core mechanism of optimization can be divided into three main parts.

Mutation – used to modify individuals in the population. Yields a mutant vector v_i^t according to a specific mutation strategy, called DE/current-to-pBest-w/1:

$$\mathbf{v}_i^t = \mathbf{x}_i^t + F_w\left(\mathbf{x}_{p\text{Best}}^t - \mathbf{x}_i^t\right) + F\left(\mathbf{x}_{r_1}^t - \mathbf{x}_{r_2}^t\right) \tag{1}$$

where r_1, r_2 are random, mutually different integers from the set $\{1, ..., \lambda\}$, $\mathbf{x}_{p\text{Best}}^t$ is a randomly chosen individual from the p best individuals in population, and F_w is described as:

$$F_w = \begin{cases} 0.7 * F, & \text{if } nfes < 0.2\ max\_nfes \\ 0.8 * F, & \text{if } nfes < 0.4\ max\_nfes \\ 1.2 * F, & \text{otherwise} \end{cases} \tag{2}$$

where F is a control parameter, $nfes$ is number of evaluated objective functions and max_nfes is the budget (maximal number of allowed objective function evaluations).

Crossover – the result of mutation (vector $\mathbf{v}_i^t$) is used to perform a crossover operation, which allows for swapping components between the original individual and its mutated version in the following fashion:

$$\forall i \in \{1, ..., \lambda\}. \forall j \in 1, ..., D. \mathbf{u}_{i,j}^t = \begin{cases} \mathbf{v}_{i,j}^t, & \text{if } rand(0,1) \leq CR \text{ or } j = j_{rand} \\ \mathbf{x}_{i,j}^t, & \text{otherwise} \end{cases} \tag{3}$$

where $CR \in [0, 1]$ is a crossover parameter and $j_{rand} \in \{1, ..., N\}$ is a randomly chosen index, which guarantees that at least one component is taken from the mutated version of the individual.

Selection – trial vector $\mathbf{u}_i^t$, created by crossover, is then evaluated using objective function f, and its fitness is compared to the fitness of the original individual $\mathbf{x}_i^t$. Better individual outlives the other, as it is included in the future population:

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{u}_i^t, & \text{if } f(\mathbf{u}_i^t) \leq f(\mathbf{x}_i^t) \\ \mathbf{x}_i^t, & \text{otherwise} \end{cases} \tag{4}$$

After these three stages archive is potentially shrunk, memories for CR and F parameters are updated, Linear Population Size Reduction (described in [18]) method is applied and p parameter is adapted.

jSO is characterized by its distinctive self-adaptation techniques for CR and F control parameters, which are described in lines 9-22, as well as shrinking of its population size.

Algorithm 3. jSO algorithm

Require: d_σ damping for step size
Require: p_{init} initial p rate
$t \leftarrow 1, p \leftarrow p_{init}$
$\mathbf{A} \leftarrow \emptyset$ ▷ archive
$M_F^i \leftarrow 0.5, i \in \{1, ..., H\}$ ▷ initialize scaling factor memory
$M_{CR}^i \leftarrow 0.8, i \in \{1, ..., H\}$ ▷ initialize crossover control parameter memory
while !stop **do**
  $S_{CR} \leftarrow \emptyset$
  $S_F \leftarrow \emptyset$
  for $i \leftarrow 1$ **to** λ **do**
    pick at random $r \in \{1, ..., H\}$
    if $r = H$ **then**
      $M_F^r \leftarrow 0.9$
      $M_{CR}^r \leftarrow 0.9$
    if $M_{CR}^r < 0$ **then**
      $CR_i^t \leftarrow 0$
    else
      $CR_i^t \sim \mathcal{N}_i(M_{CR}^r, 0.1)$
    if $t < 0.25 T_{MAX}$ **then**
      $CR_i^t \leftarrow \max(CR_i^t, 0.7)$
    else if $t < 0.5 T_{MAX}$ **then**
      $CR_i^t \leftarrow \max(CR_i^t, 0.6)$
    $F_i^t \sim \mathcal{C}(M_F^r, 0.1)$
    if $t < 0.6 T_{MAX}$ **and** $F_i^t > 0.7$ **then**
      $F_i^t \leftarrow 0.7$
    $\mathbf{u}_i^t \leftarrow$ CURRENT-TO-PBEST-W/1/BIN ▷ mutation and crossover, using Equation 1 and Equation 3
  for $i \leftarrow 1$ **to** λ **do**
    if $f(\mathbf{u}_i^t) \leq f(\mathbf{x}_i^t)$ **then** ▷ selection, using Equation 4
      $\mathbf{x}_i^{t+1} \leftarrow \mathbf{u}_i^t$
    else
      $\mathbf{x}_i^{t+1} \leftarrow \mathbf{x}_i^t$
    if $f(\mathbf{u}_i^t) < f(\mathbf{x}_i^t)$ **then**
      $\mathbf{x}_i^t \rightarrow \mathbf{A}$
      $CR_i^t \rightarrow S_{CR}$
      $F_i^t \rightarrow S_F$
    Shrink $\mathbf{A}$ if necessary
    Update M_{CR} and M_F
    Apply Linear Population Size Reduction, as described in [18]
    $p \leftarrow p_{init}\left(1 - \frac{nfes}{2max\_nfes}\right)$ ▷ update p
    $t \leftarrow t + 1$

2.1.4 Default parameters

The default optimization algorithms' parameter values used in this research are depicted in Table 1. If parameter is not specified in Table 1 and its value is not described in the experiment, a default value has been used based on the reference paper. All heuristics optimize hyperpara-

meters based on the provided allowed values ranges. Non-lamarckian approach is assumed, with the harshest possible penalty for boundary crossing, effectively killing individuals, which leave provided boundaries.

Table 1. Paramerers of heuristic algorithms used in the study.

Algorithm	Parameter name	Parameter value
DES	λ	28
CMA-ES	λ	9
CMA-ES	σ	0.2
CMA-ES	initial parameter value	0.5
jSO	archive rate	1.0
jSO	memory size	5
jSO	pBest rate	0.25
All	budget[1]	700
All	lower bound	0.0
All	upper bound	1.0

[1] Number of objective function (classifier training) evaluations.

2.2 Datasets

Three different real-life datasets are used to compare heuristic hyperparameter optimization. Due to their nature, achieving correct classification on them can be challenging, with problems such as:

- high target class imbalance;
- poor quality of attributes;
- mislabeled samples;
- small number of samples in the dataset.

Optimizers should be able to correctly tune hyperparameters in order to account for these factors, as mentioned obstacles are often found in real-life datasets. These characteristics motivated the choice of datasets for the thesis.

2.2.1 Aspartus

After a road accident takes place, the victim can call an insurance company of the perpetrator for insurance claims. As a part of that process, insurance companies in Poland are obliged to ask the victim whether or not their vehicle needs to be repaired, and if so, it must propose a replacement vehicle, or cash compensation. The task of the model trained on this dataset is to indicate which option will be taken by the victim. Supplied dataset contains 12072 examples. Target class has 2 different values:

- **CASH** cash compensation chosen;
- **CAR** replacement car chosen.

Class proportions are presented in Table 2.

Table 2. Proportions of the target class in the Aspartus dataset.

Class	number of records	% in the dataset
CASH	1227	10.16
CAR	10845	89.84

There are 110 attributes in the dataset, which provide information, amongst other things, about victim's and perpetrator's vehicles, their homes' locations, date of the accident, weather during the accident and forecast, upcoming holidays etc. All attributes are described in-depth in [19].

For the purpose of this thesis only a selected group of attributes is used (33), all of which are continuous, and it is assumed that the missing values have been filled by mean of the attribute where applicable. This dataset is used for Multilayer Perceptron's hyperparameter tuning, as outlined in Section 2.3.1.

2.2.2 Titanic

A sample in the Titanic dataset describes a single passenger of the fatal last cruise. It contains information about sex and age of the person, their family aboard the ship, ticket class and price, as well as the port of embarkation. Four of these attributes are continuous (age, ticket price, number of siblings/spouses and parents/children aboard the boat), while the others are discrete. One-hot encoding was used to encode discrete attributes. Target class is binary and indicates survival of the person. This is a small (1309 samples), benchmark dataset, with mild target class imbalance. This dataset has been shared by the Vanderbilt University's Department of Biostatistics [20].

Table 3. Proportions of the target class in the Titanic dataset.

Class	number of records	% in the dataset
SURVIVED	500	38.20
NO_SURVIVAL	809	61.80

Hyperparameters for Multilayer Perceptron are tuned on this dataset (described in depth in Section 2.3.1.

2.2.3 Icebergs

This dataset has been provided as a part of the 'Statoil/C-CORE Iceberg Classifier Challenge' competition [21] hosted on the Kaggle platform. This dataset contains satellite radar images of icebergs and ships. Each sample is described by two vectors of floating point numbers, each of length 5625. These vectors represent radar images from two bands, each being single channel with dimensions 75 x 75 in pixels. Each pixel on the image represents value in decibels. Both channels are signals resulting from radar backscatter with different polarizations - respectively HH (transmit/receive horizontally) and HV (transmit horizontally and receive vertically).

Classifier's goal is to label given image, deciding if the sample is a ship or an iceberg. Target class proportions are described in Table 4. Based on this dataset, convolutional neural network classifier is optimized, as described in Section 2.3.2.

Table 4. Proportions of the target class in the Icebergs dataset.

Class	number of records	% in the dataset
ICEBERG	753	46.95
SHIP	851	53.05

2.3 Classifiers

Architectures of two different classifiers are described in this section. Moreover, each subsection describes specific hyperparameters, that will be optimized.

To simplify optimization process, as well as comparisons of distributions of different hyperparameters all individuals are represented as vectors of floating point numbers ranging between 0 and 1. In order to convert such representation to the desired hyperparameters' values a *transformation* function is used (e.g. for a value 0.42 obtained from an individual and a transformation function defined as 10^{-1-5*x} the real hyperparameter value used for the neural network roughly equals to $7.94 * 10^{-4}$). These functions are presented for described architectures and hyperparameters.

Weights of all evaluated models are randomly initialized (with identically seeded random number generator) using Glorot's uniform initialization rule [22].

2.3.1 Multilayer Perceptron

In this study a Multilayer Perceptron (MLP) with one hidden layer is used. This is an almost classical MLP design, which allows for non-linear classification, while being the simplest to train and reason about. The architecture is not fixed completely, as one of the tuned hyperparameters is the number of neurons in the hidden layer. The only difference between used architecture and classic design is the fact that a single dropout layer [23] is added, between hidden and output layers. Due to its nature, it does not change the way that predictions are made, but it changes training procedure, preventing possible overfitting tendencies of the model. As the dropout ratio is tuned, it is possible that this layer becomes effectively disabled, if that hyperparameter is tuned to the values close to 0. As such, heuristic optimizer can modify the architecture of the model in two different ways.

Stochastic Gradient Descent [24] (SGD) with Nesterov's Momentum [25] is used as an optimizer for the neural network's training process.

This architecture has been chosen for tuning due to its popularity, especially in low-dimensional datasets' classification, and simple architecture, which guarantees fast training process.

All tuned hyperparameters for the Multilayer Perceptron classifier are described in Table 5.

Table 5. Hyperparameters tuned for the MLP architecture.

Hyperparameter	Range		Transformation
	Lower	Upper	
Hidden layer size	1	1001	$x * 10^3 + 1$
Weight regularization ratio[1]	10^{-6}	10^{-2}	10^{-2-4*x}
Activity regularization ratio[1]	10^{-6}	10^{-2}	10^{-2-4*x}
Dropout	0.0	0.9	$x * 0.9$
Learning rate	10^{-6}	10^{-1}	10^{-1-5*x}
Learning rate decay	10^{-8}	10^{-3}	10^{-3-5*x}
Nesterov's momentum	0.0	2.0	$x * 2.0$

[1] L2 regularization applied on `dense_1` and `dense_2` layers.

2.3.2 Convolutional Neural Network

To test optimizers' performance for more complex models I use a relatively simple convolutional network, architecture of which is depicted on Figure 1. It is simple enough to allow for relatively fast training process, while lending itself to effective small images classification (100 x 100 pixels). All layers use ReLU activation function, with the exception of the output layer, which uses sigmoid activation function. Optimizer used for the CNN architecture is ADAM [26]. To prevent overfitting, dropout layers and both weight and activity L2 regularization [27] are used. Hyperparameters tuned for this architecture are depicted in Table 6.

Table 6. Hyperparameters tuned for the MLP architecture.

Hyperparameter	Range		Transformation
	Lower	Upper	
`dropout_2` value	0.0	0.9	$x * 0.9$
`dropout_3` value	0.0	0.9	$x * 0.9$
Weight regularization ratio[1]	10^{-6}	10^{-2}	10^{-2-4*x}
Activity regularization ratio[1]	10^{-6}	10^{-2}	10^{-2-4*x}
Learning rate	10^{-5}	10^{-1}	10^{-1-4*x}

[1] L2 regularization applied on `dense_1` and `dense_2` layers.

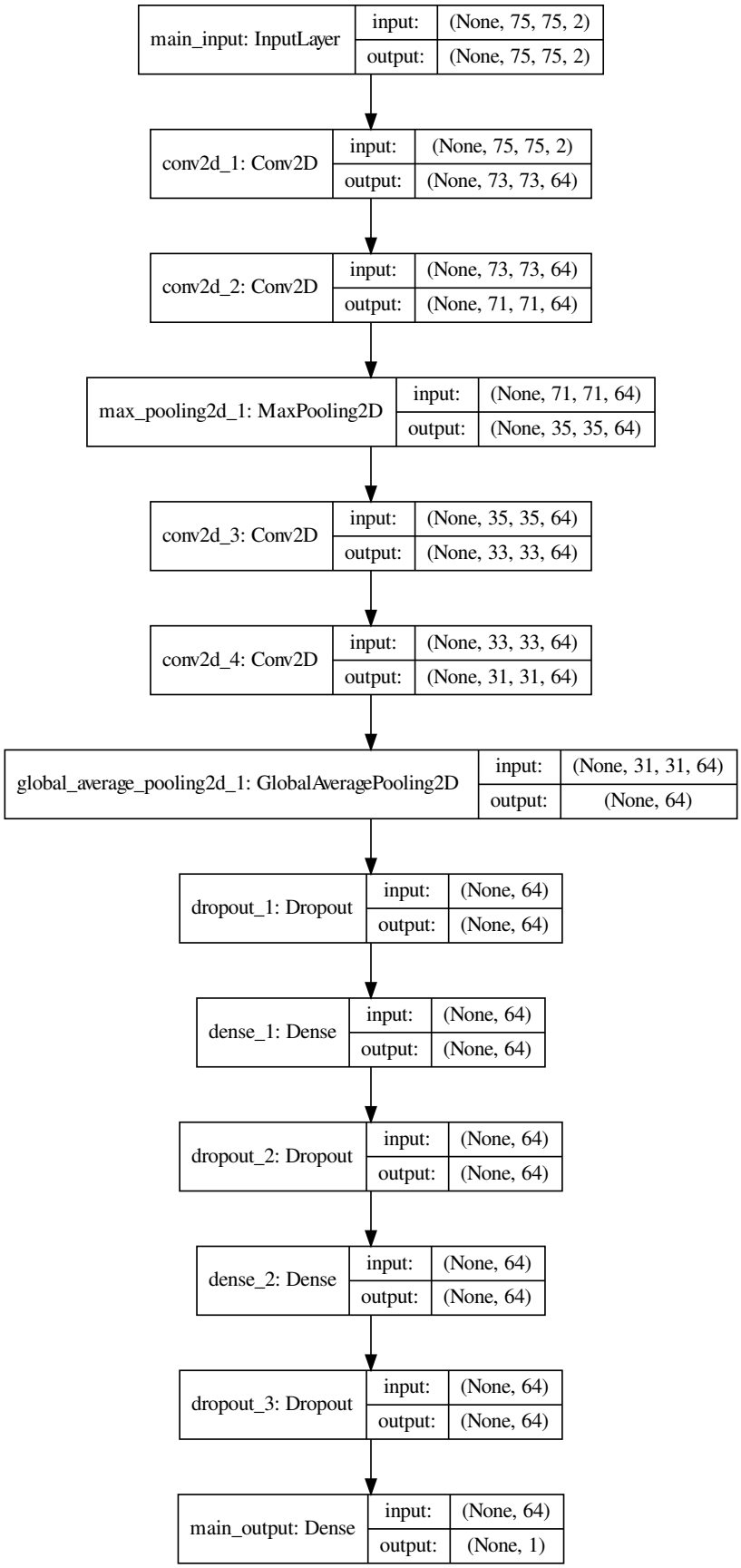


Figure 1. Convolutional Neural Network architecture used in the study. Numbers correspond to the input and output dimensions of data tensor, with *None* being the placeholder for the number of samples.

2.4 Default experiment description

Unless explicitly stated otherwise all experiments are run in the fashion described below.

1) Initialize chosen heuristic algorithm. Its parameters are described in Section 2.1.4, or in the experiment description, if default parameters have not been used.
2) Create specific subsets, based on the dataset used in the experiment, in the following manner:
   a) Split dataset into two parts using 80:20 proportions. Use stratification, as it preserves target class' proportions in all subsets. The smaller subset (i.e. 20% of the dataset) will be further referred to as *test* subset.
   b) Split the remaining 80% of the dataset using the same proportions (80:20) and stratification. The bigger of two subsets created that way will be referred to as *training* subset, while the smaller one as *validation* subset.
3) Define objective function as follows:
   a) Given an individual from heuristic optimizer, transform it to obtain proper hyperparameters' values. Transformations are described separately for each classifier, and can be found in Table 5 and Table 6.
   b) Create a classifier using provided hyperparameters.
   c) Train the classifier on the training subset of the dataset.
   d) Evaluate trained classifier on the validation subset.
4) Run heuristic optimization process until it runs out of budget (function evaluations).
5) Select the best individual from the history of the optimization.
6) Create a classifier using selected individual.
7) Train created classifier using combined train and validation subsets.
8) Evaluate trained model on the test subset.

Each classifier starts with the fixed, known state of the pseudo-random number generator, so that results are repeatable and starting weights are identical for all individuals. There are 3 separate stopping conditions for the neural network's training process:

- each model is given fixed amount of epochs, defined per dataset/experiment;
- each model has maximal CPU/GPU time budget, after which training is stopped regardless of the state of the training;
- training is stopped if the process plateaus, i.e. no significant change in logarithmic loss, calculated for the validation subset, is observed over the period of 20 epochs.

2.5 Evaluation methods

Evaluation of experimental results should consider various characteristic of the hyperparameter tuning process. More specifically, this process can be studied by considering optimization itself, focusing on the quality of population, its elite, convergence rate, etc. Furthermore, one could examine classifiers which are created and tuned based on different individuals found

by heuristics, exploring such things as history plots of loss function and target metric, or classification quality of the trained model. Evaluation methods for each approach are described in this section.

2.5.1 Heuristics evaluation methods

Two different tools were used to visualize and compare heuristic optimization results: Empirical Cumulative Distribution Function (ECDF)[28] and distribution plots for each hyperparameter.

ECDF

ECDF is a function of the proportion of executed objective function evaluations when the algorithm successfully reaches certain step. For all of the experiments described in this thesis the following budget steps are assumed: $\{0.01, 0.02, 0.03, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$, where each number denotes percentage of the budget reached (e.g. assuming budget of 100 objective function evaluations and step 0.3 this step would be reached after 30 objective function evaluations). These have been inspired by budget steps used by the CEC competition. For each algorithm and for each run of the experiment, a 'running best' individual history is created (i.e. for each generation the best solution found so far is selected). Thresholds span between worst individual in all histories in the first iteration and the best individual in the last iteration. ECDF curves allow for comparison of different heuristics, averaging through repeated runs of the experiment. In order to quantify these curves an Area Under the Curve is used, referred to as EAUC (ECDF AUC). Moreover, to further simplify comparisons I propose Normalized ECDF AUC (NEAUC), such that its value is in the range of $[0, 1]$.

Hyperparameters distribution plots

To visualize how heuristics optimize hyperparameters, distribution plots are created for each hyperparameter, each illustrating last 5 generations. Kernel Density Estimation with Gaussian kernel is used to estimate these distributions. To calculate bandwidth Scott's method [29] is applied:

$$N^{-\frac{1}{D+4}} \tag{5}$$

where N is the number of data points and D is the number of dimensions. Those distributions are then plotted using 256-point mesh.

2.5.2 Best and mean fitness plots

To track the behavior of the elite plots for the best individual in each generation, as well as 'running' best individual (i.e. best individual found so far) are provided. These plots can be used to further clarify the exploration vs. exploitation strategy used by the heuristic and indicate trends in the optimization process.

In order to observe these characteristics with respect to the whole population mean and standard deviation plots are presented for selected experiments. This technique has a limitation, namely the plot loses on readability as the number of plotted experiment runs increases, which is the reason why it is not shown for all experiments.

2.5.3 Classifier evaluation methods

Classifiers' performance is assessed using selection of tools: ROC curves, logarithmic loss, as well as training history plots.

ROC AUC

To evaluate neural network's performance on a binary dataset (i.e. one with 2 target classes) an Area Under the Curve (AUC) will be measured for the Receiver Operator Characteristic (ROC) curve. ROC is a standard tool for graphical assessment of binary classifiers. It is created by plotting True Positive Rate (TPR, also known as sensitivity, recall) against False Positive Rate (FPR, also fall-out, $1 - specifity$) at various thresholds. To generate these rates predicted probability of each sample is mapped onto predicted class by thresholding it, using specified cut-off probability. Afterwards, TPR and FPR can be calculated as:

$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN} \tag{6}$$

where TP, FP, FN, TN are numbers of True Positive, False Positive, False Negative and True Negative samples.

After plotting ROC, its AUC can be measured, with random classifier and perfect classifier yielding values 0.5 and 1.0 respectively.

In this thesis ROC's AUC will be referred to as RAUC in order to prevent confusion with ECDF's AUC (EAUC).

Log loss

For non-binary datasets, models will be evaluated by the multinomial cross-entropy loss (also known as logarithmic loss or log loss in short):

$$\mathcal{L}(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \log(p_{ij}) \tag{7}$$

where n is the number of samples, m number of target classes, y_i is a one-hot encoded label of the sample (a binary vector with dimension m, in which 1 denotes the target class, and 0 otherwise), p_i is a vector of probabilities for each class for the given sample such that

$$p_{ij} \in (0, 1) : \forall i \sum_{j=1}^{m} p_{ij} = 1 \tag{8}$$

Training history plots

Another way to visualize final results of optimization is the plot of loss function throughout classifier's training. In order to do so, each classifier stores a backlog of selected metrics, as well as logarithmic loss for both training and validation subset, calculated after each epoch of the training process. This backlog can then be plotted to better understand how does the classifier learn, check for possible overfit, etc. Most notably, these plots are the backbone of manual hyperparameter tuning, and so plotting these backlogs for selected individuals might help to better understand characteristics of the heuristic optimization process. In this thesis two different methods of plotting these backlogs are used:

- Simple plots — given a set of backlogs plot selected metrics for each of them, with distinct plot lines for train and validation subsets;
- Aggregations — given a set of sets of backlogs (e.g. few selected individuals for each optimizer) first calculate a 'running' mean and standard deviation for each set of backlogs, and then use this information with the first described technique. Here I call the statistic 'running' if it is calculated at each step for all available backlogs. The reason behind this approach is that there are 3 distinct stopping conditions for the classifier and so backlogs can differ in length (number of epochs) recorded. In other words, a 'running' mean of a metric is a 1-D vector calculated for each epoch in all backlogs, where at each epoch all available values of a metric are averaged.

2.6 Method for verification of the heuristics re-implementations

In order to easily integrate neural network related code with heuristic algorithms, an implementation of DES and jSO algorithms in the Python programming language have been provided. They were rewritten from the original projects, which used C++ (jSO) and R language (DES). To test these re-implementations I compared them against Congress on Evolutionary Computation (CEC) benchmark set from 2017 [17]. Numeric results are presented for all functions with the dimensionality of 10, representing different modalities, as documented in Table 7. To account for the randomness an optimization process is run 51 times for each of the functions, as required by the benchmark rules, and results are aggregated.

Table 7. Characteristics of CEC'2017 functions used for implementation validation.

Function number in CEC'2017 set	Function type
1-3	Unimodal
4-10	Simple Multimodal
11-20	Hybrid
21-30	Composition

Two methods were used to compare these results: visual inspection of Empirical Cumulative Distribution Functions (described in Section 2.5.1) and statistical approach using a non parametric test. The second method uses best individuals found throughout the optimization process. Specifically, for each heuristic and for each function, an optimization process is independently run 51 times, yielding a 1-D vector with 51 fitness values for each of the best individual in each run. Next, for each benchmark function a two-sample Wilcoxon rank-sum test [30] is calculated, using two vectors (one from each implementation) as input samples. These p-values can then be aggregated by the means of a meta-analysis technique called Fisher's method [31], [32]. It can be divided into two phases. First, a X^2 test statistic is calculated:

$$X_{2k}^2 \sim -2\sum_{i=1}^{k} \ln(p_i) \tag{9}$$

where p_i is the p-value for the ith test and k is the number of tests. Next, assuming that all tests have been independent (as is the case in the described procedure), a final 'meta' p-value can be calculated, based on the fact that X^2 follows a chi-squared distribution with $2k$ degrees of freedom.

To account for directionality of calculated rank-sum tests, their p-values are changed into one-sided versions, while taking the directionality under account. This is achieved using the following formula:

$$p_{\text{one-sided}} = \begin{cases} \frac{p}{2}, & \text{if } Z > 0 \\ 1 - \frac{p}{2}, & \text{otherwise} \end{cases} \tag{10}$$

where p is the two-tailed p-value and Z is Wilcoxon rank-sum test's statistic.

2.7 Language, tooling and hardware used

All experiments have been implemented using the Python programming language, version 3.4, using the following libraries:

- `numpy` — linear algebra library, on top of which re-implementations of DES and jSO have been built;
- `matplotlib` — used to create all plots presented in the thesis;
- `deap` (Distributed Evolutionary Algorithms in Python) — provides implementation of the CMA-ES algorithm, which was slightly modified to better suit the needs of this research;
- `keras` — high-level library for neural network classification, aids the process of creating a classifier, abstracting over different computational backends;
- `theano` — library, which provides efficient composition and optimization of complex, multi-array mathematical operations. Used as one of the backends for `keras` library;
- `tensorflow` — machine learning framework, an alternative to `theano`, also used as a backend for `keras`;
- `scipy` — framework for scientific computing, provides variety of statistical tests, i.a. implementation of Wilcoxon rank-sum test;
- `scikit-learn` — machine learning toolkit library, implements various metrics (e.g. ROC AUC, confusion matrices generation), as well as Logistic Regression classifier used as a reference classifier.

Experiments for the Icebergs dataset have been run using GPU with `theano` backend for the `keras` library. The rest of experiments have been run on CPU with 6 cores per an experiment, using `tensorflow` as the backend. Evaluation platform is described in Table 8.

Table 8. Software and hardware used as a platform for experiments in the thesis.

System	Kernel	4.9.0-6-amd64 x86_64 (64 bit)
	Distribution	Debian GNU/Linux 9 (stretch)
CPUs		4x 12-Core Intel Xeon E7-4830 v3, L2 cache: 120 MB
GPU		NVIDIA Tesla K20m
Memory		252 GB

3 Results and conclusions

3.1 Results of re-implementations' verification

Results of statistical test for each function, along with the combined 'meta' p-values calculated using the Fisher's method, can be found in Table 9. Null hypothesis for the Wilcoxon rank-sum test states that two sets of measurements are drawn from the same distribution. Assuming $\alpha = 0.001$, 'meta' p-values indicate that this hypothesis can be rejected for the jSO re-implementation ($p \approx 10^{-112}$) and can not be rejected for DES re-implementation, with $p \approx 0.48$. These results are consistent with ECDF curves calculated for selected benchmark functions, which are provided in Figure 2. Characteristics of 'DES' and 'DESpy' curves for all problems and all dimensionalities indicate that re-implementation has been successful and does not significantly differ from the original. On the other hand, curves for 'jSO' and 'jSOpy' do not appear to be similar. More specifically, Python's based version of the algorithm tends to perform better than the original on lower dimensionalities. While I was not able to find and correct the source of the difference, it is plausible that it originates from differing implementations of random distributions — authors of the original jSO algorithms used built-in C++ randomness source to implement these distributions, while Python-based version uses Python's `random` module for this task. I decided to use the re-implementation, as it generally outperforms the original on low-dimensional problems and optimization problems in this thesis are defined as low-dimensional due to the hardware and time limitations.

Table 9. Results of two-sided Wilcoxon rank-sum tests between different implementations of DES and jSO algorithms.

Function number	DES		jSO	
in CEC'2017 set	p-value[1]	Statistic	p-value[1]	Statistic
1	0.4555	−0.7462	1.0000	0.0000
2	1.0000	0.0000	1.0000	0.0000
3	1.0000	0.0000	1.0000	0.0000
4	1.0000	0.0000	1.0000	0.0000
5	0.7103	−0.3714	6.0354×10^{-08}	5.4177
6	0.5270	−0.6324	1.0000	0.0000
7	0.6322	0.4785	6.5039×10^{-08}	5.4043
8	0.1024	1.6330	3.2811×10^{-07}	5.1065
9	1.0000	0.0000	1.0000	0.0000
10	0.9014	0.1238	2.6108×10^{-12}	6.9972
11	0.7103	−0.3714	7.4497×10^{-07}	4.9492
12	0.9439	0.0702	0.0595	1.8839
13	0.2949	1.0474	7.2148×10^{-09}	5.7858
14	0.8225	−0.2242	4.2335×10^{-11}	6.5956
15	0.8018	−0.2509	0.0147	2.4394
16	0.9439	−0.0702	4.7471×10^{-07}	5.0362
17	0.4047	0.8332	1.1675×10^{-15}	8.0078
18	0.6806	−0.4116	0.9386	−0.0769
19	0.1382	−1.4824	1.7157×10^{-10}	6.3848
20	0.3334	−0.9670	0.0018	3.1154
21	0.7966	−0.2576	0.0109	2.5432
22	0.7812	0.2777	7.6162×10^{-13}	7.1678
23	0.1082	1.6062	1.5378×10^{-10}	6.4015
24	0.0030	2.9615	0.9067	0.1171
25	0.3972	0.8466	0.6466	0.4584
26	0.6708	−0.4249	0.7328	0.3413
27	0.1382	−1.4824	1.6241×10^{-06}	−4.7953
28	0.6039	−0.5186	0.7028	−0.3814
29	0.5671	−0.5722	3.0969×10^{-17}	8.4428
30	0.8592	−0.1773	1.1765×10^{-11}	6.7830
Fisher's p-value[2]	0.4796		6.7939×10^{-112}	

[1] Two-sided.

[2] Aggregated p-value from Fisher's method on one-sided p-values from presented tests.

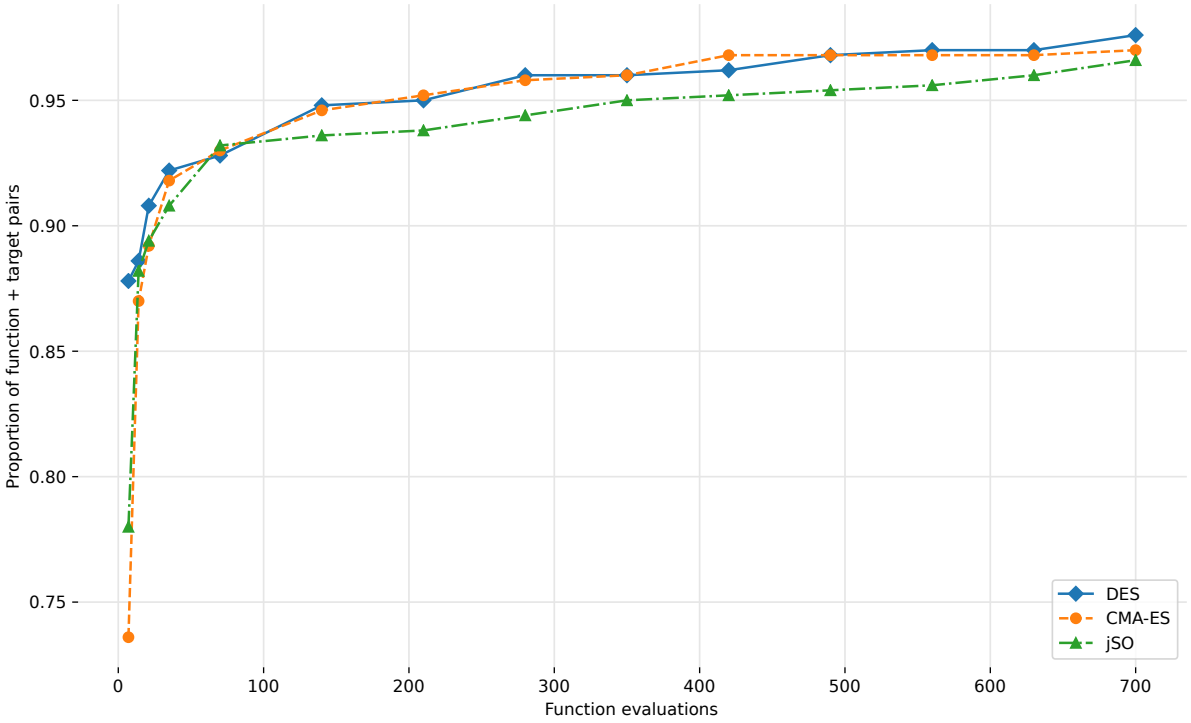


Figure 2. ECDF curves for various implementations of selected heuristic algorithms on specific CEC'2017 functions. Re-implementations in Python language have suffix 'py'.

3.2 Results for Aspartus dataset

ECDF curves are illustrated in Figure 3 and numeric results can be found in Table 10. While DES and CMA-ES have similar results, with the former slightly outperforming the latter, there is a noticeable difference between them and the jSO algorithm. All three optimizers locate similar maxima, however it takes jSO significantly much more time to arrive at such solution. Early results are comparable for all algorithms, with DES having a better starting position, probably due to its initialization strategy (uniform distribution between 0.1 and 0.9 for each hyperparameter). Worse performance of jSO can have its source in the suboptimal parameters set for this algorithm – default values are used.

Table 10. ECDF AUC values resulting from optimization on the Aspartus dataset. Each curve has been created based on ten independent runs of the experiment.

Heuristic algorithm	EAUC	NEAUC
DES	662.44	0.95
CMA-ES	661.46	0.94
jSO	654.58	0.94

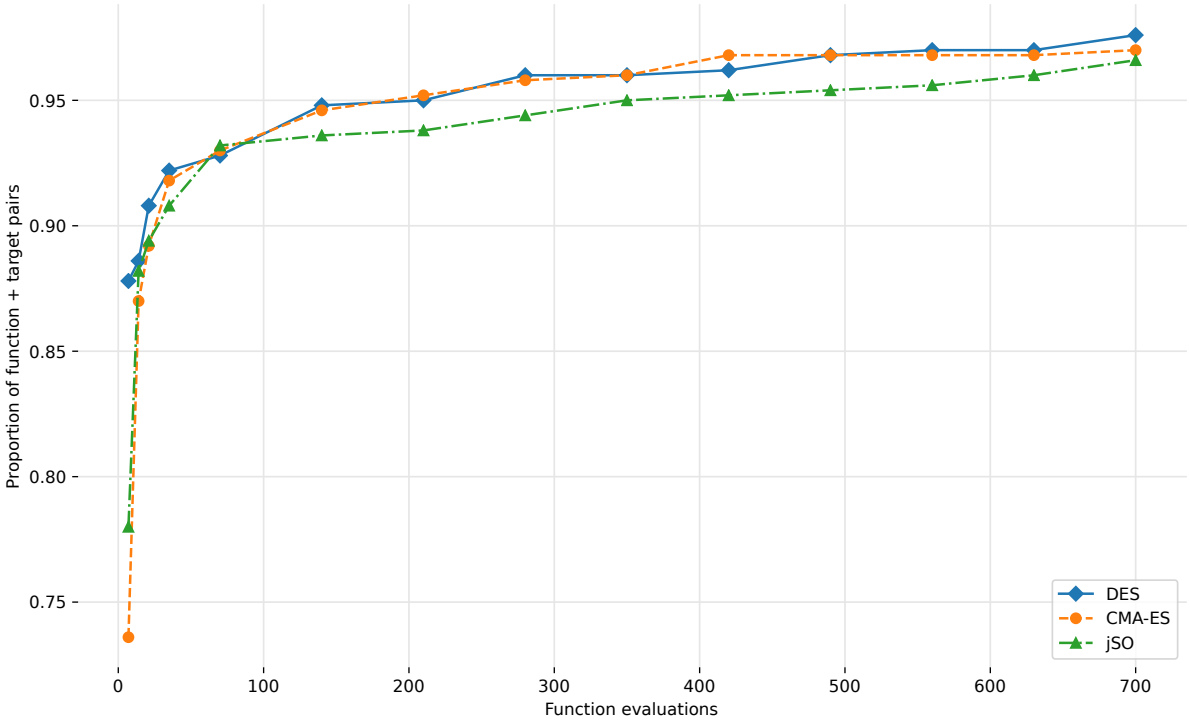
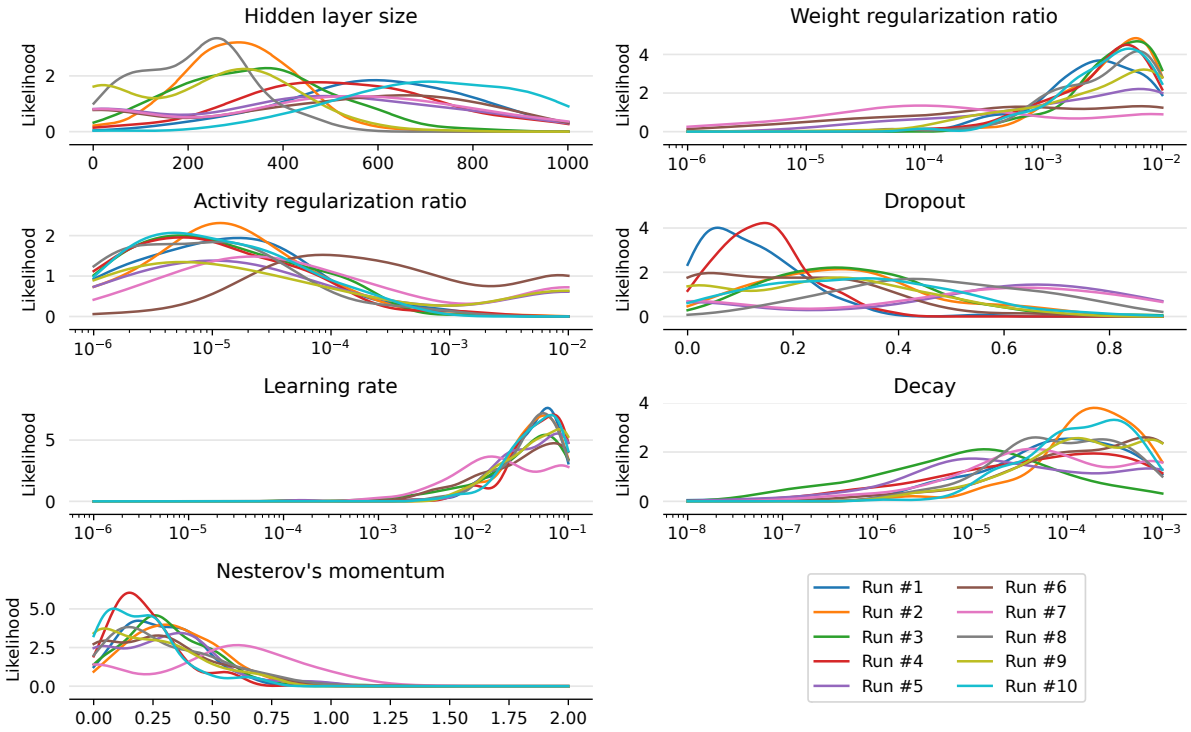
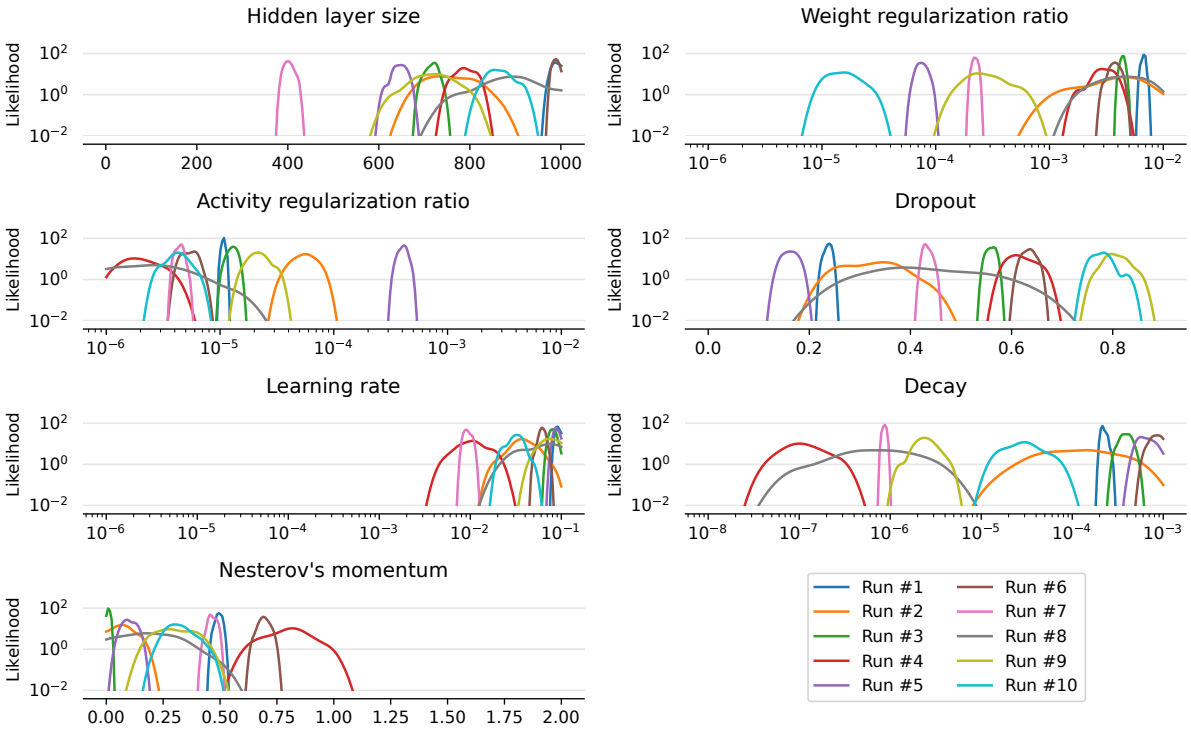


Figure 3. ECDF curves resulting from optimization on the Aspartus dataset.

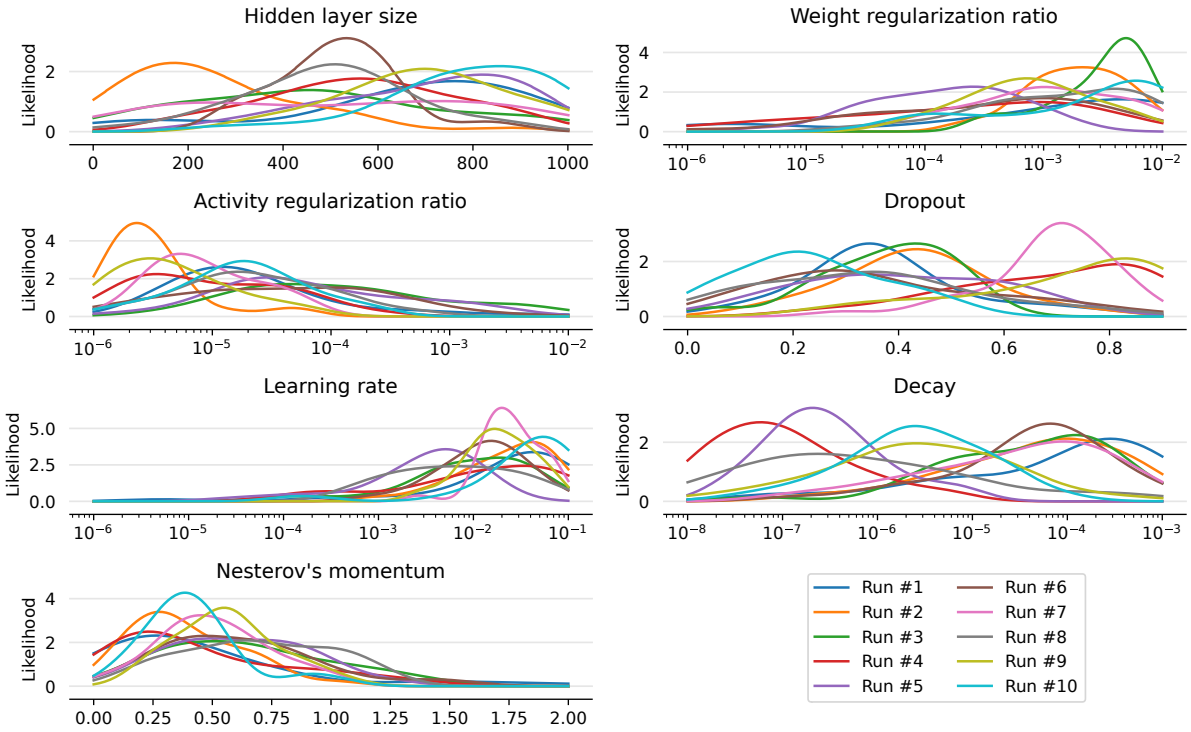
To better understand hyperparameter values selected by different heuristics their distributions are plotted in Figure 4. Note that these distributions vary between runs and algorithms, as there does not necessarily need to be a single optimal value for hyperparameter, but rather different values are acceptable and it is their combination that directly influences classifier's results. This is the case with hyperparameters such as dropout, learning rate's decay or hidden layer size. On the other hand, optimizers converge on the similar values of learning rate and Nesterov's momentum, which indicates that these hyperparameters might be the biggest factors in proper classifier training. Because regularization can effectively be modeled by either activity or kernel (weight) regularization it should be the case that only one of these ratios is set high, while the other remains low, or both have moderate rates, assuming that there is a need for the regularization in the first place. Observing high rates of both types of regularization is highly unlikely, due to the nature of the study — such regularization would substantially slow down learning process and thus the results achieved after training process runs out of budget would be worse, even if progress was successfully being made. Such balance can be seen in this experiment for all heuristics. jSO and DES go with more polarizing strategies, while some of CMA-ES runs yield more balanced rates. All optimizers opt for high learning rates, strongly indicating that in the future optimizations on this architecture and dataset, range for learning rate could be limited, possibly resulting in better initial performance of the optimization process.


Hidden layer size
Weight regularization ratio
Activity regularization ratio
Dropout
Learning rate
Decay
Nesterov's momentum
Likelihood
Run #1
Run #2
Run #3
Run #4
Run #5
Run #6
Run #7
Run #8
Run #9
Run #10

(a) DES


Hidden layer size
Weight regularization ratio
Activity regularization ratio
Dropout
Learning rate
Decay
Nesterov's momentum
Likelihood
Run #1
Run #2
Run #3
Run #4
Run #5
Run #6
Run #7
Run #8
Run #9
Run #10

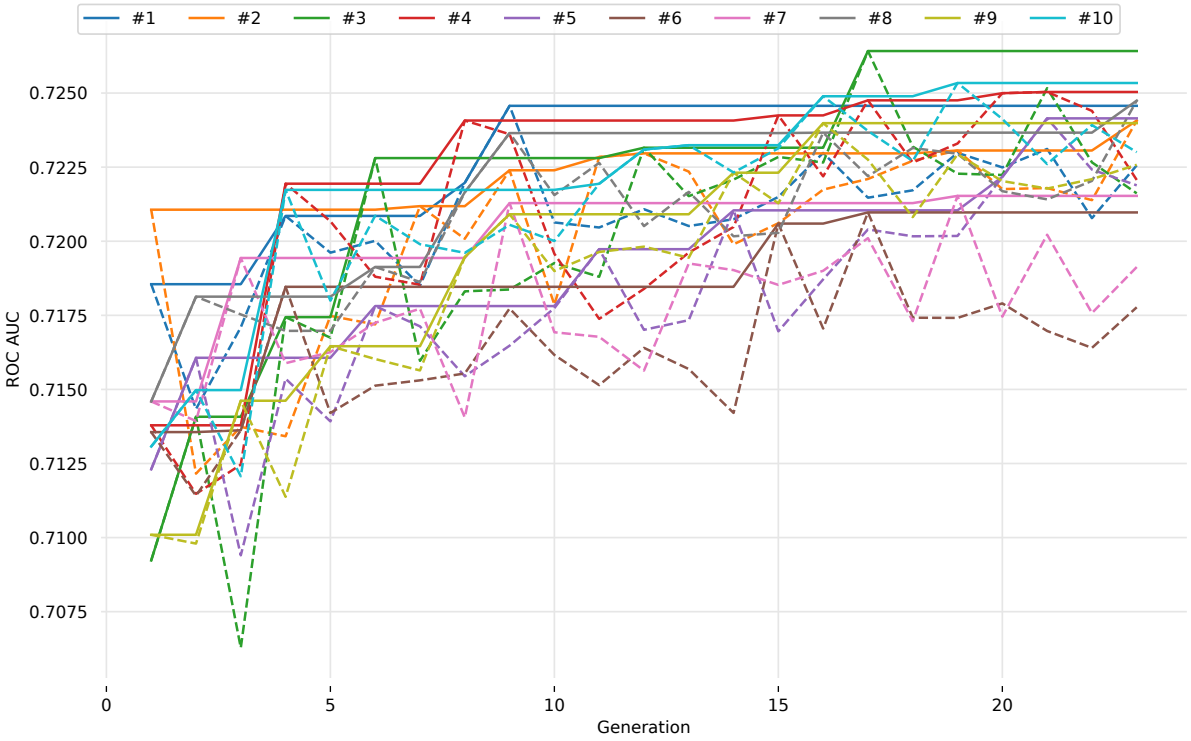
(b) CMA-ES



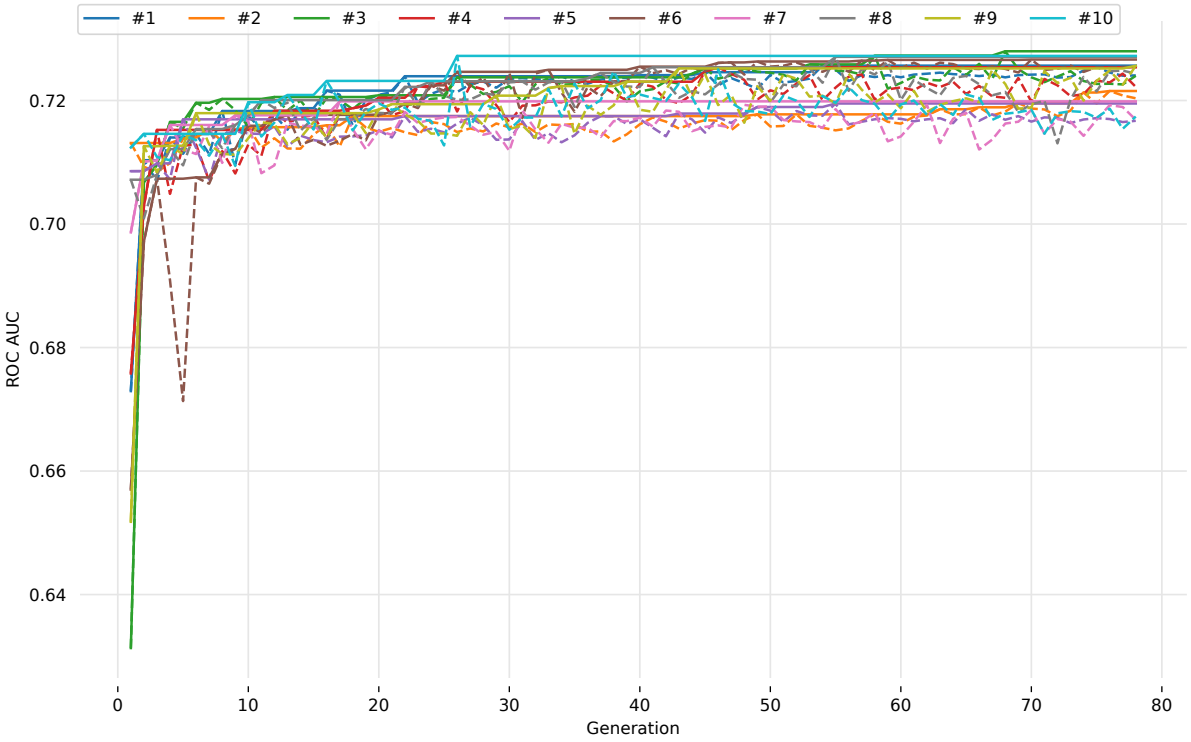
(c) jSO

Figure 4. Hyperparameters' distribution estimation resulting from optimization on the Aspartus dataset. Plots created based on the last 5 generations. Numbers in the legend correspond to the number of the experiment run.

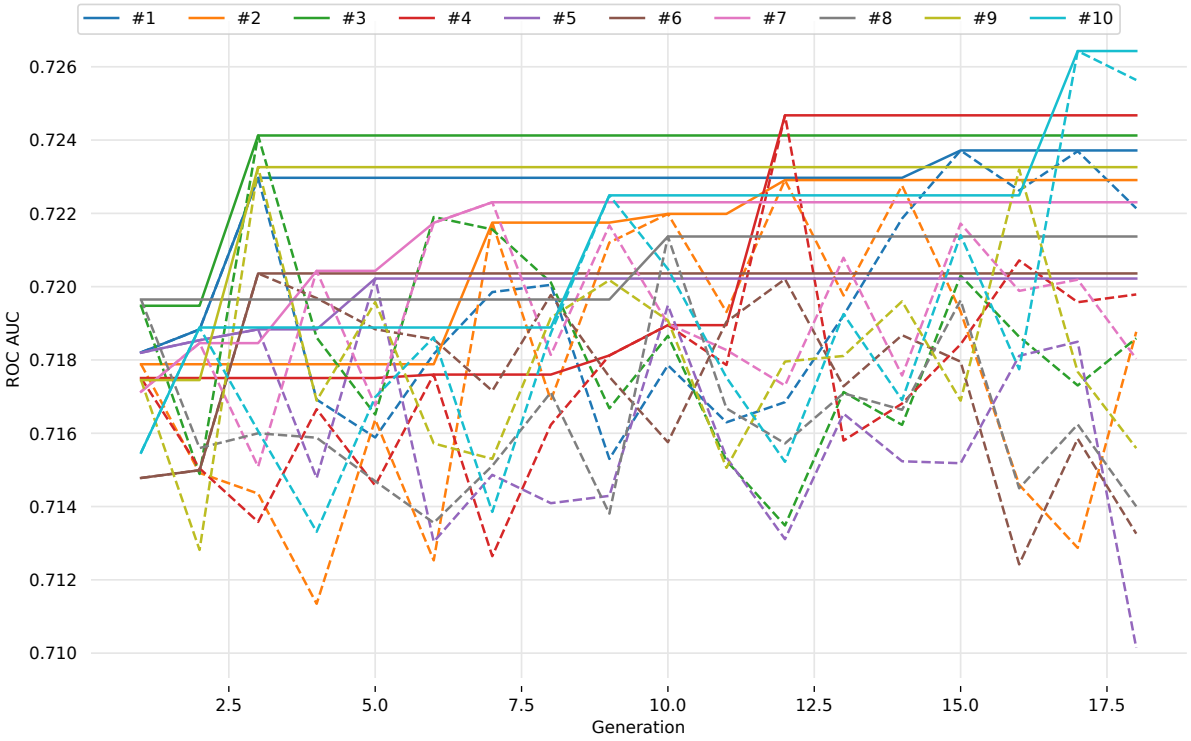
Visualizations of the best individuals throughout optimization processes can be found in Figure 5. All algorithms achieved comparable results in terms of the best individuals throughout entire optimization process, being placed between 0.72 and 0.73 RAUC. DES shows tendency to steadily improve its elite over time, whereas jSO allows for significant oscillations of the elite, indicating stronger mutations of the individuals between generations. Latter technique is also feasible, as indicated by the end results. CMA-ES rapidly improves its elite in the first 20 epochs, after which the progress slows down.


#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
ROC AUC
0.7250
0.7225
0.7200
0.7175
0.7150
0.7125
0.7100
0.7075
0
5
10
15
20
Generation

(a) DES


#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
ROC AUC
0.72
0.70
0.68
0.66
0.64
0
10
20
30
40
50
60
70
80
Generation

(b) CMA-ES

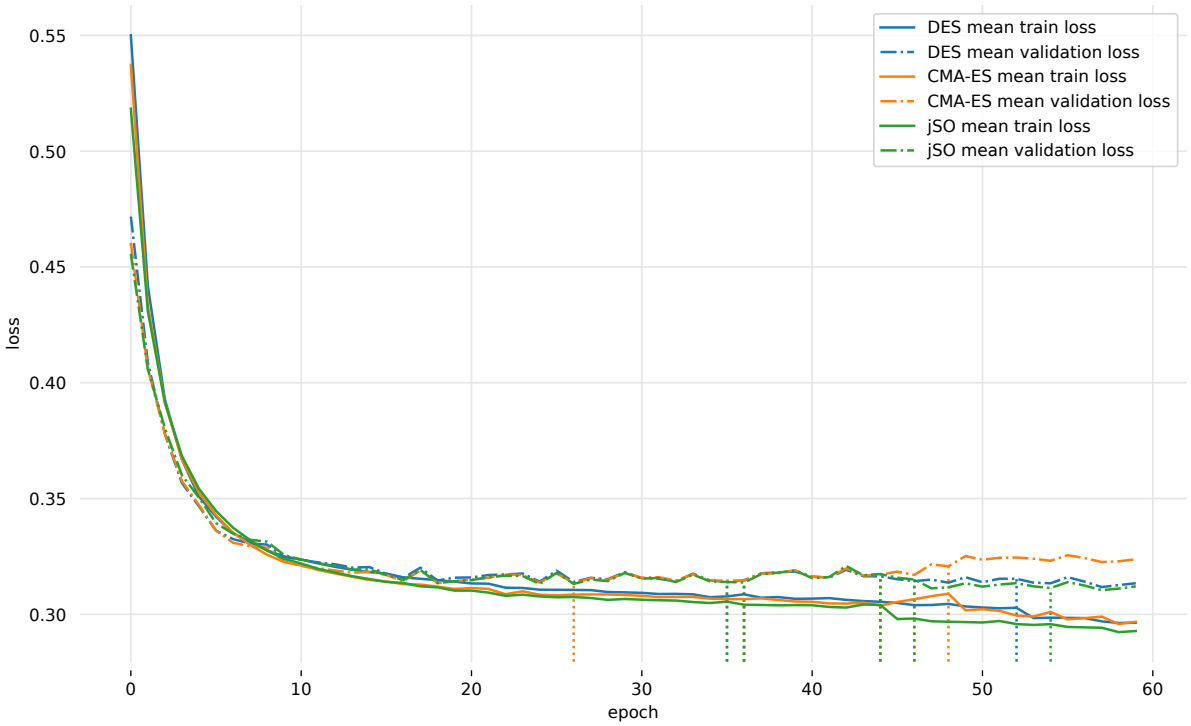


(c) jSO

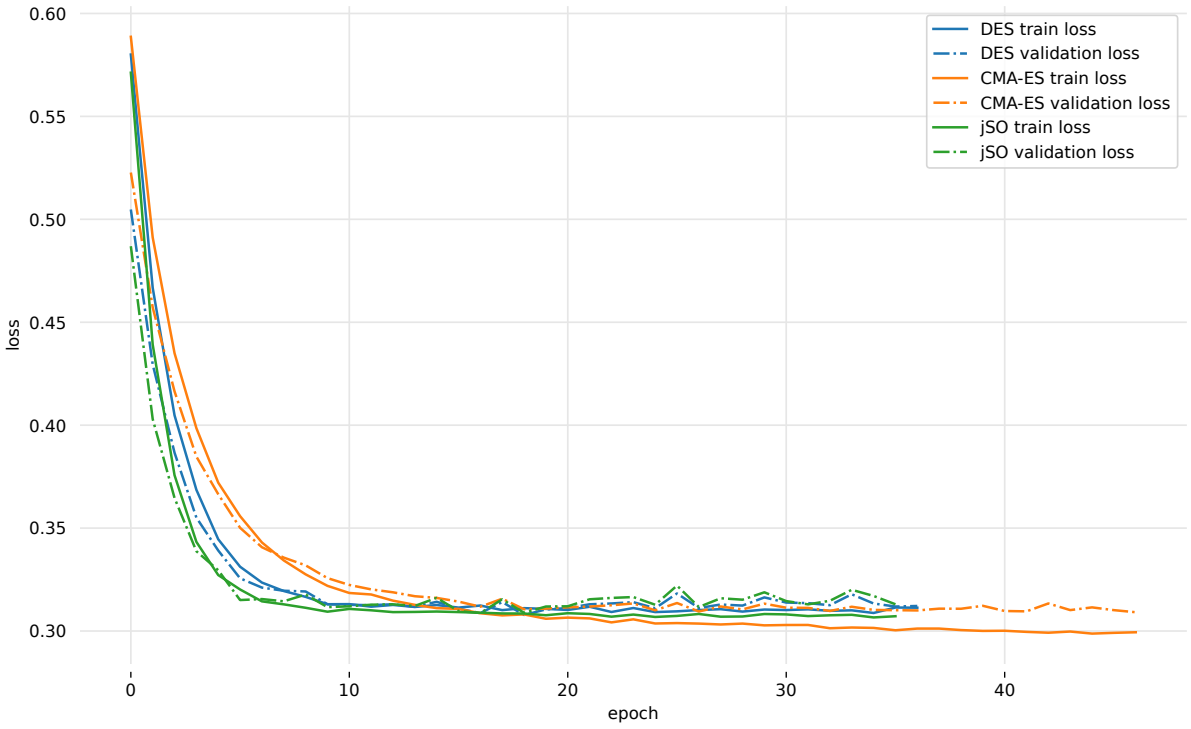
Figure 5. Best fitness in population (dashed lines) and 'running' best fitness (solid lines) found during optimization on the Aspartus dataset. Color associated with the number of the run is depicted in legends.

Plots of the logarithmic loss are presented in Figure 6. Plots of mean values have been created by selecting the best individual in each run of the experiment for each optimizer and calculating a 'running' mean for each optimizer. 'Best' plots are created for selecting the best individual across all runs of the experiment for each optimizer. There are several interesting conclusions that can be made based on these backlog plots, as listed below.

- It seems that selected number of epochs (60) is larger than needed for the classifier to achieve an optimal solution. Specifically, the best individuals for both DES and jSO are stopped before reaching maximal number of epochs.
- While all optimizers do fairly well in terms of managing overfit amongst their elite (as can be seen in the means plot), the best solutions do vary in this regard. DES is marginally better than jSO, however CMA-ES yields individuals which do not control overfit well.
- Overall training histories of elites show that all optimizers find similar solutions (DES and jSO having the closest characteristics), which could indicate an optimum being reached.



(a) Mean



(b) Best

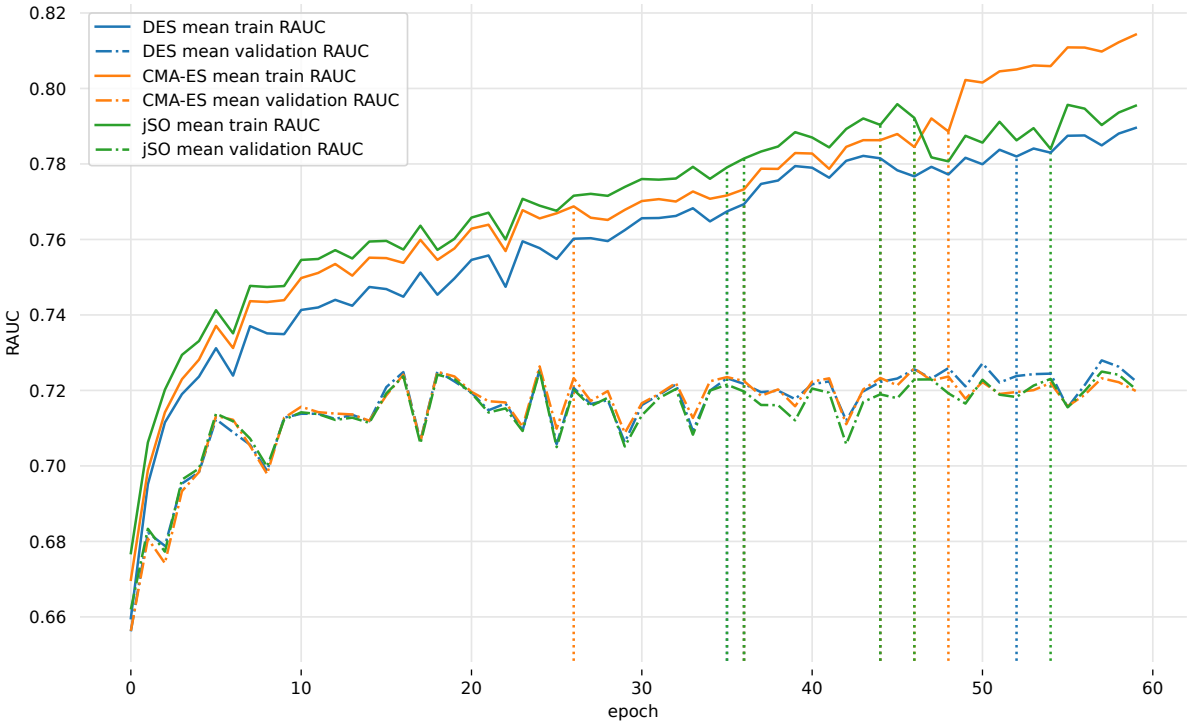
Figure 6. Loss histories for selected individuals found during optimization on the Aspartus dataset. Dotted vertical lines in the means plot denote early training termination of a classifier.

RAUC history plots over selected individuals' training are depicted in Figure 7. Overfit can be seen more easily in these plots, especially for CMA-ES. While CMA-ES does not cope well

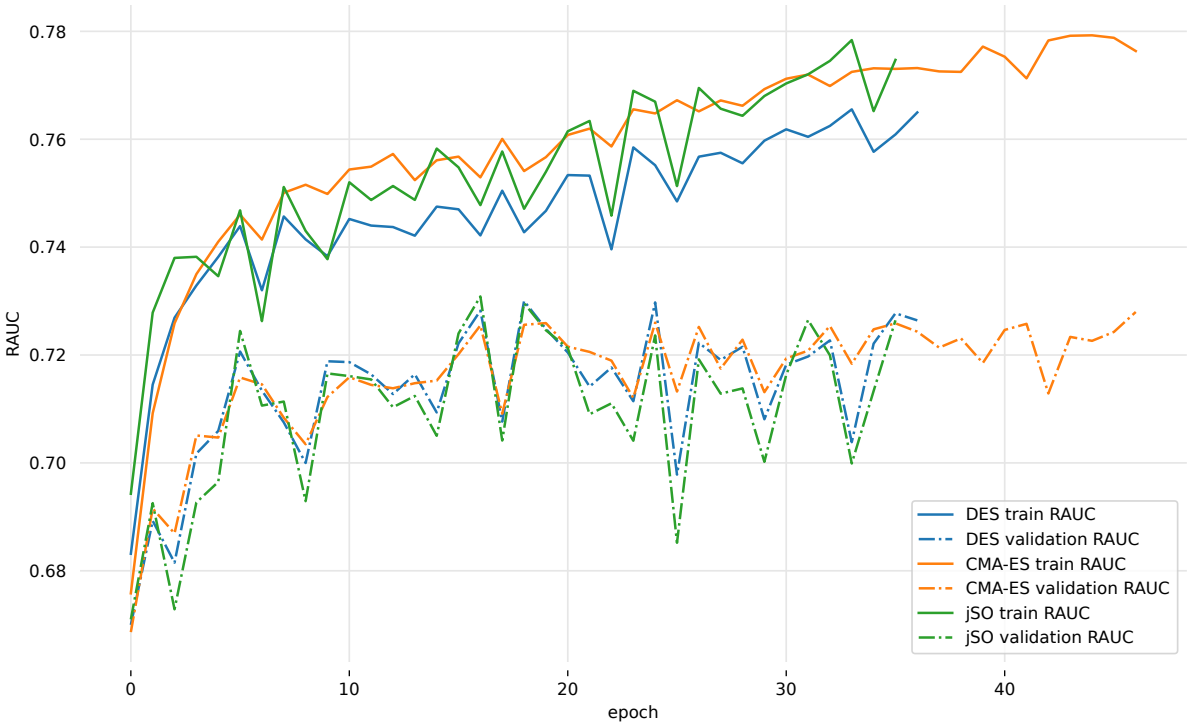
with the overfit problem, its models have slightly smoother history curves, possibly indicating stronger regularization. Additionally, it is easier to observe jumps in quality of the classifier as opposed to the loss graphs.

Differences between RAUC achieved on the validation and test subsets can stem from multiple reasons:

- Due to the way objective function is set up optimization process will create individuals, which are overfitted on the validation subset. When these individuals are trained on a bigger input data and tested against new subset their performance could be worse.
- Even though test, train and validation subsets are created in a stratified fashion, quality of the data might vary and so it might happen, that test subset contains more samples which are hard to classify properly. This could explain the observed difference, especially as the Aspartus dataset has been built upon real-life data.



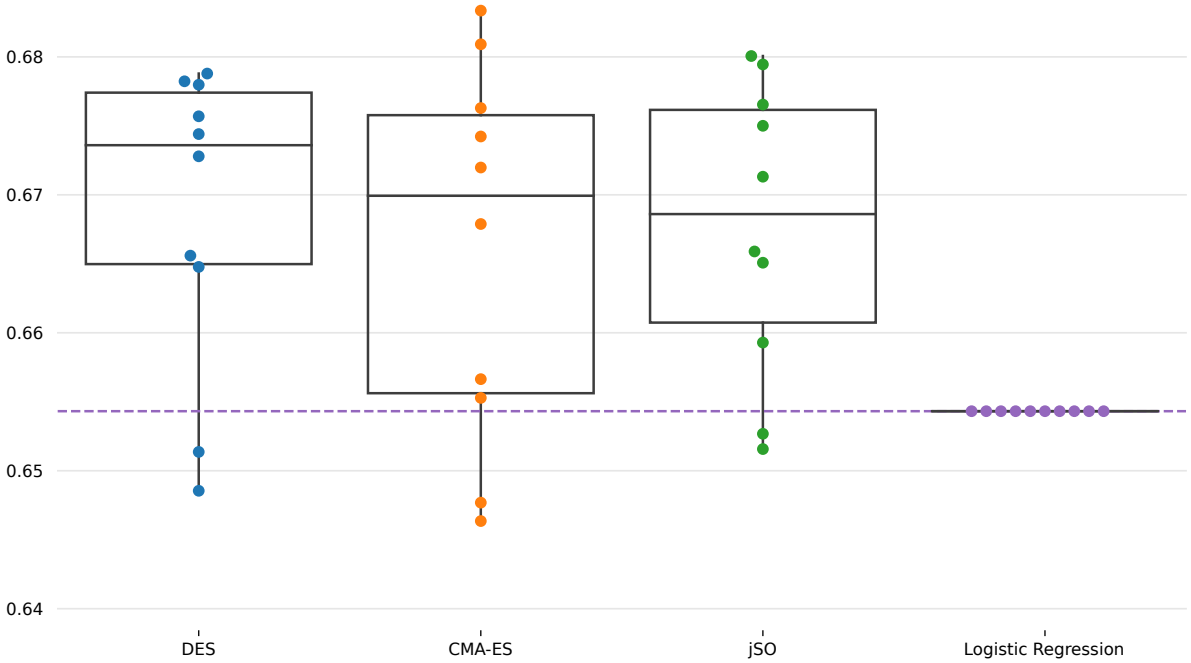
(a) Mean



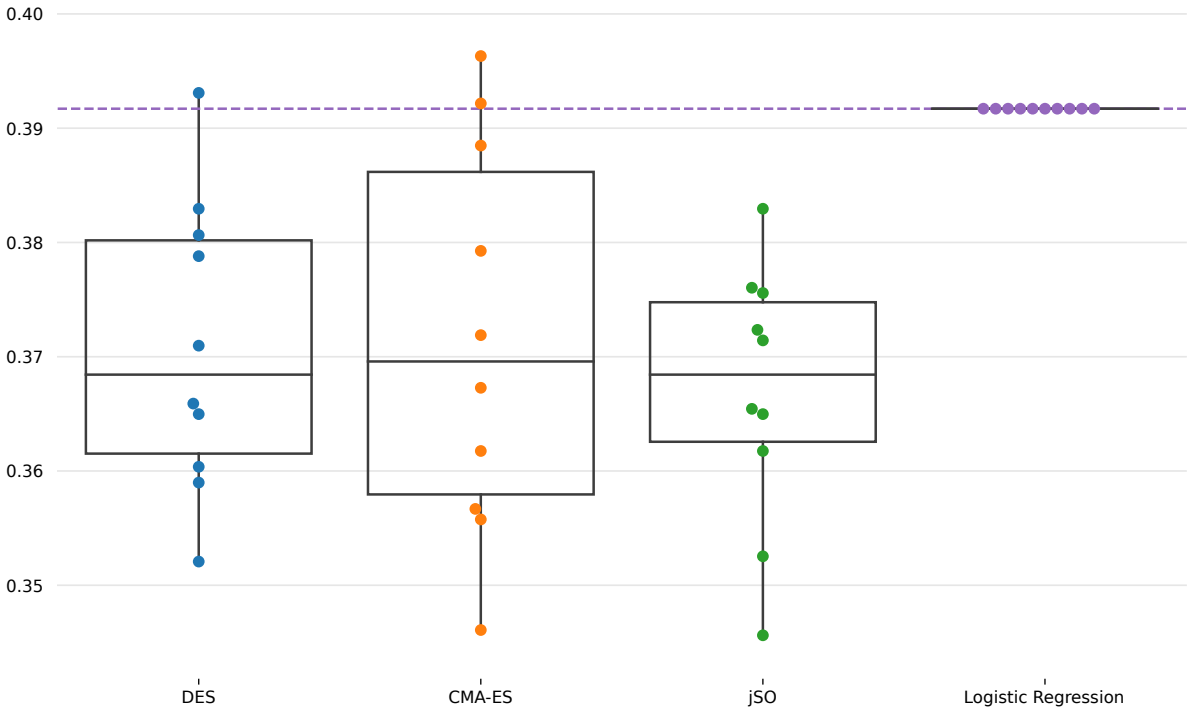
(b) Best

Figure 7. RAUC histories for selected individuals resulting from optimization on the Aspartus dataset. Dotted vertical lines in the means plot denote early training termination of a classifier.

Finally, results of the selected models are visualized in Figure 8 and Figure 9. Exact results of all experiments can be found in the Appendix, Table A.1. Additionally to these models, a Logistic Regression with l2 penalty classifier is trained on the merged training and validations subsets and evaluated on the test subset. It has been selected due to the best performance achieved amongst several models in the original study on the given dataset and its results are presented as a reference. jSO, which performed worse throughout most of the optimization process (shown in Figure 3), has comparable results. Most individuals are equally good or better than reference model (logistic regression). All heuristics outperform reference classifier, with best individuals yielding AUC improvements of 0.03. Comparing EER values it can be noted that both DES and jSO perform best with mean EER value of 0.364, closely followed by CMA-ES with 0.374. The lowest achieved EER for an individual is 0.346, yielded by both CMA-ES and jSO. There is a notable difference between Logistic Regression model and optimized individuals, namely 0.02 and 0.03 respectively for CMA-ES and both jSO and DES. 0.04 difference is noted for the best individuals.



(a) ROC AUC



(b) Equal Error Rate

Figure 8. Box plots of RAUC and EER values for each run of the experiment on the Aspartus dataset. Each dot denotes a single experiment run. Purple, dashed line marks result yielded by the reference classifier.

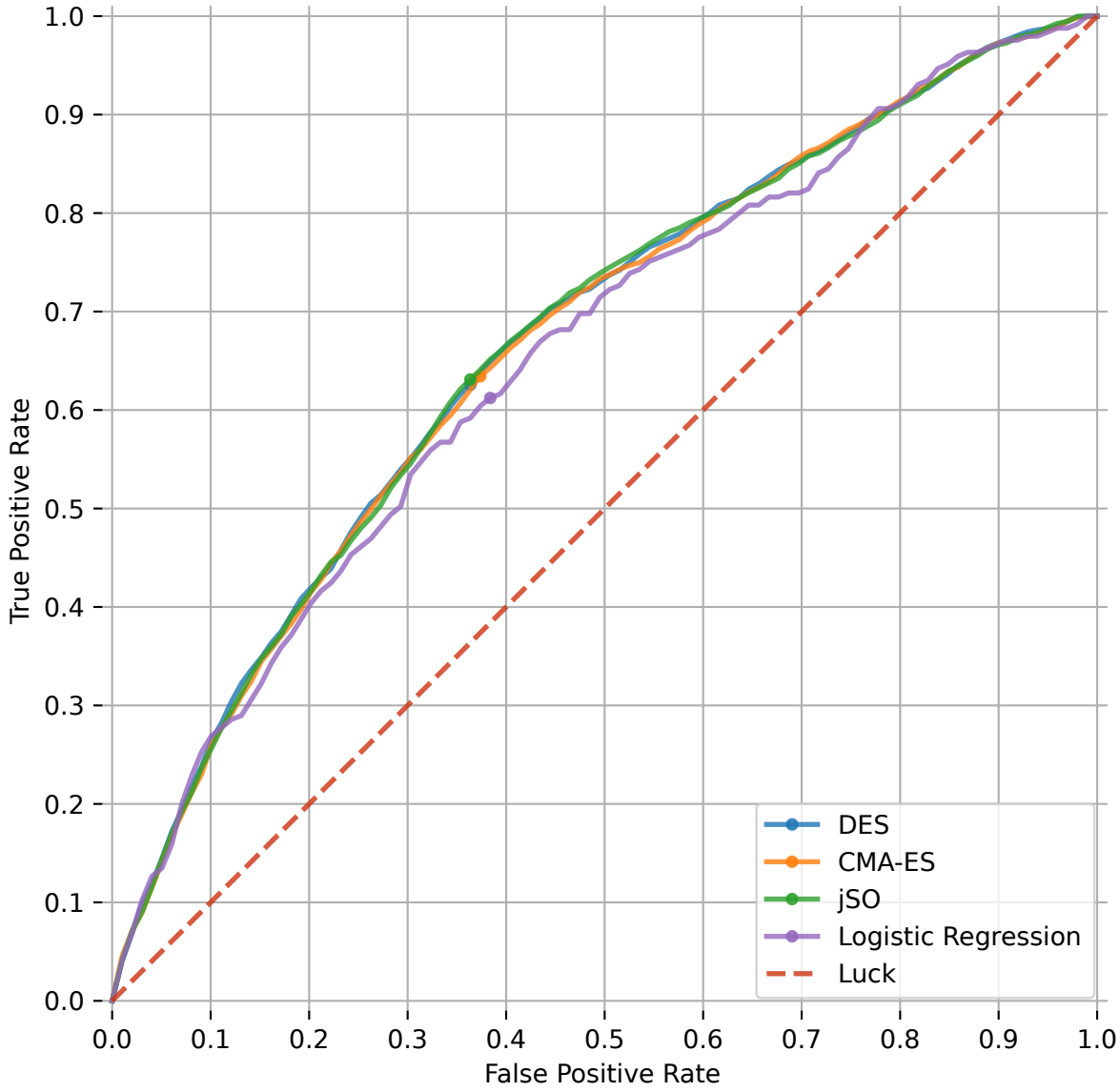


Figure 9. Mean ROC curves for the best solutions found during optimization on the Aspartus dataset. Curve for Logistic Regression model is provided as a reference.

3.3 Results for the Icebergs dataset

Due to the time and hardware constraints this experiment has not been repeated multiple times, and as such its results can not be used to directly compare different optimizers. However, they can be used to test the possibility of optimization — if the results are bad then it can be argued that run experiment was an outlier, which could be detected if multiple repetitions had been evaluated. On the other hand, if the results are good then it is reasonable to assume that such heuristic algorithm is able to correctly tune hyperparameters on the Icebergs dataset. Notably, this is also the primary way in which user would be tuning the hyperparameters – with a one-shot approach.

ECDF curves for optimizers are illustrated in Figure 10 and numeric results are written in Table 11. jSO starts in the significantly worst place, but quickly improves its performance in the first 100 function evaluations. Going further, the progress is slower, but the process does

not plateau. DES shows a flat start and for the first 40 function evaluations it is unable to improve. Its performance then slowly ramps up and in the end almost catches up with CMA-ES. CMA-ES has the best starting position and it significantly outperforms other algorithms in the first 350 function evaluations. However, further progress is much slower.

Table 11. ECDF AUC values resulting from optimization on the Icebergs dataset. Each curve has been created based on ten independent runs of the experiment.

Heuristic algorithm	EAUC	NEAUC
DES	567.14	0.81
CMA-ES	610.54	0.87
jSO	535.50	0.77

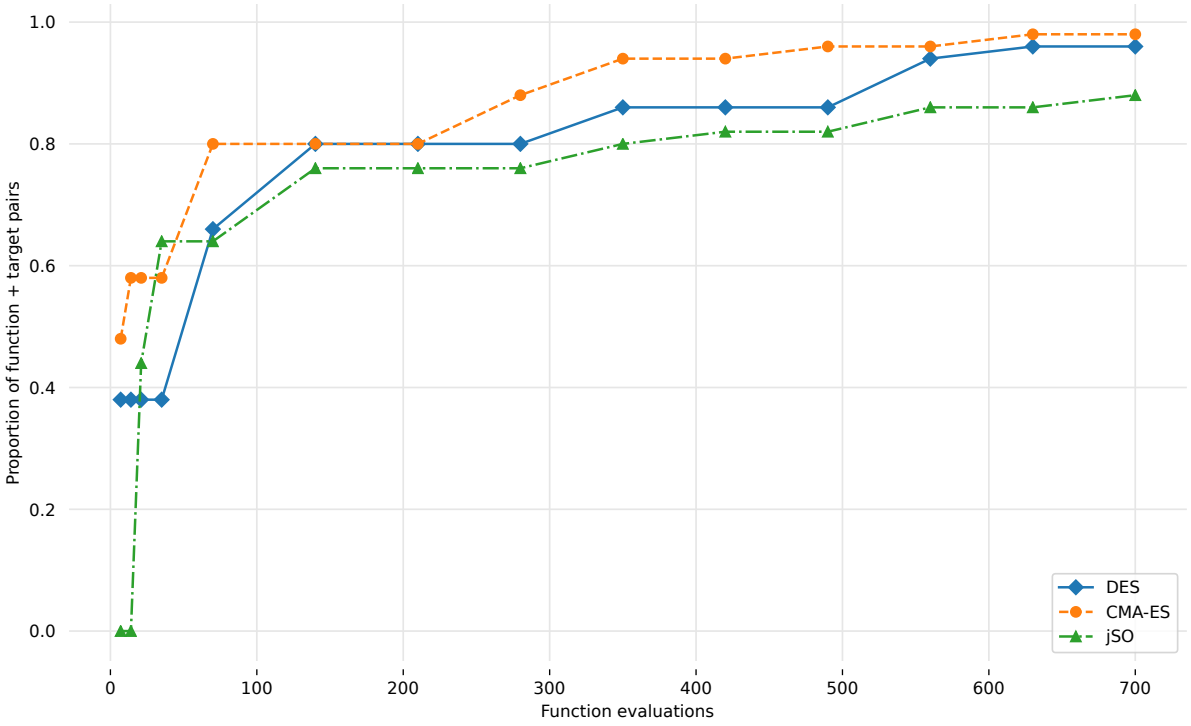


Figure 10. ECDF curves resulting from optimization on the Icebergs dataset.

Distribution plots for different hyperparameters are depicted in Figure 11. All optimizers favour learning rate around 10^{-3}, which is vastly different than general learning rate chosen for both Aspartus and Titanic datasets. This may indicate that lower learning rate allows for much smoother training phase. This could also be a result of a differently defined objective function – for this dataset tuning algorithms optimize log loss on the validation set, as opposed to RAUC, which is the metric used for optimization on other datasets.

DES and jSO tend to set the second dropout value lower than the third one, while CMA-ES keeps them at equal level (0.2), which is also a generally suggested lowest effective dropout rate. This illustrates the trade-off between dropout that is too high (effectively slowing down training process) and too low (might result in strong overfit).

Regularization ratios vary significantly between optimizers, with DES opting for the strongest regularization and jSO choosing the lowest ratios, particularly for the weight regularization. One of the possible explanations for this behavior is that classifiers do not have significant overfit problems, especially since multiple dropout layers are introduced.

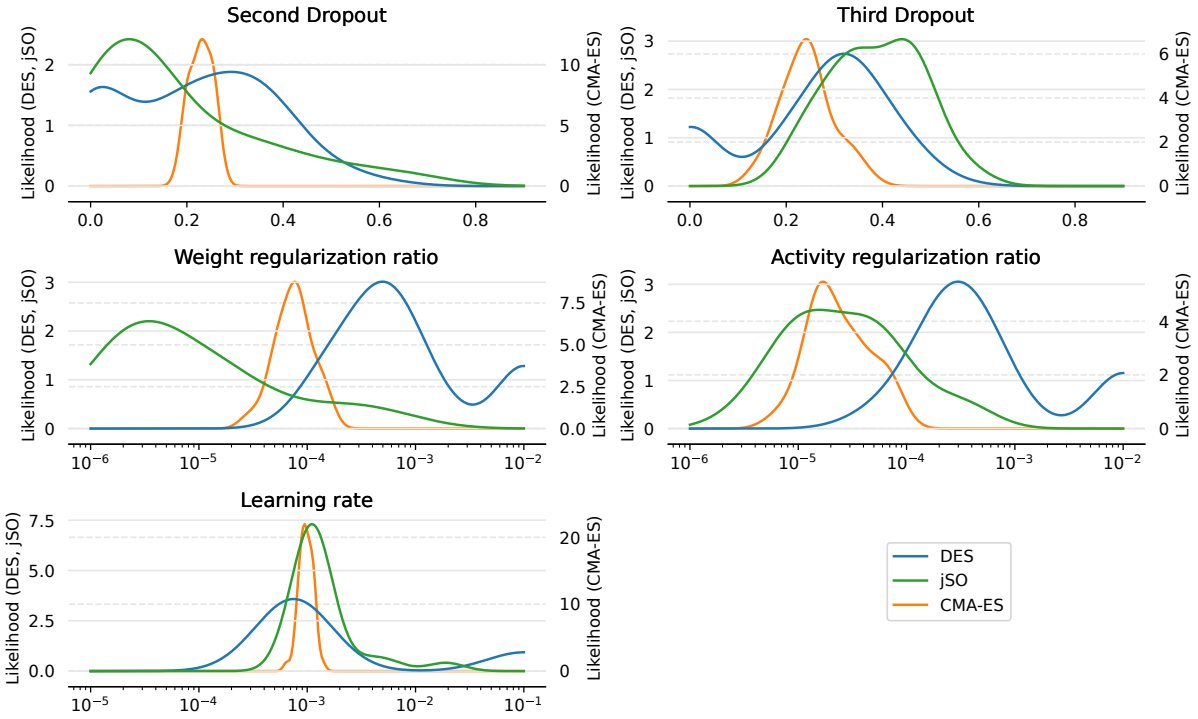


Figure 11. ECDF AUC values resulting from optimization on the Icebergs dataset. Each curve has been created based on ten independent runs of the experiment.

Plots of the 'running' best and best-in-generation individuals are depicted in Figure 12. Differences in generation numbers arise from different population sizes of each algorithm. There is a substantial difference between 'running' best and best-in-generation for jSO in the last 15 generations. Plots for DES suggest that it transits smoothly from exploration to exploitation phase. CMA-ES, on the other hand, shows greater oscillations of its elite, but still manages to consecutively find better individuals throughout the optimization process.

Mean score per generation, along with the respective standard deviation is plotted in Figure 13. Out of all algorithms, CMA-ES is the most likely to significantly worsen its average performance, possibly caused by step size expansion, combined with boundary crossing. To an extent, similar behavior is observable for DES, although with smaller deviation from the reasonable solution. This can be explained by the fact that DES has bigger population size, and so only part of the population can be put out of boundaries, or into the worse place in the search space. DES has a pretty smoothly descending mean. jSO's characteristic does not contain such drastic peaks, but rather has a fairly constant standard deviation. Its mean, however, can worsen from time to time, and does so for several generations.

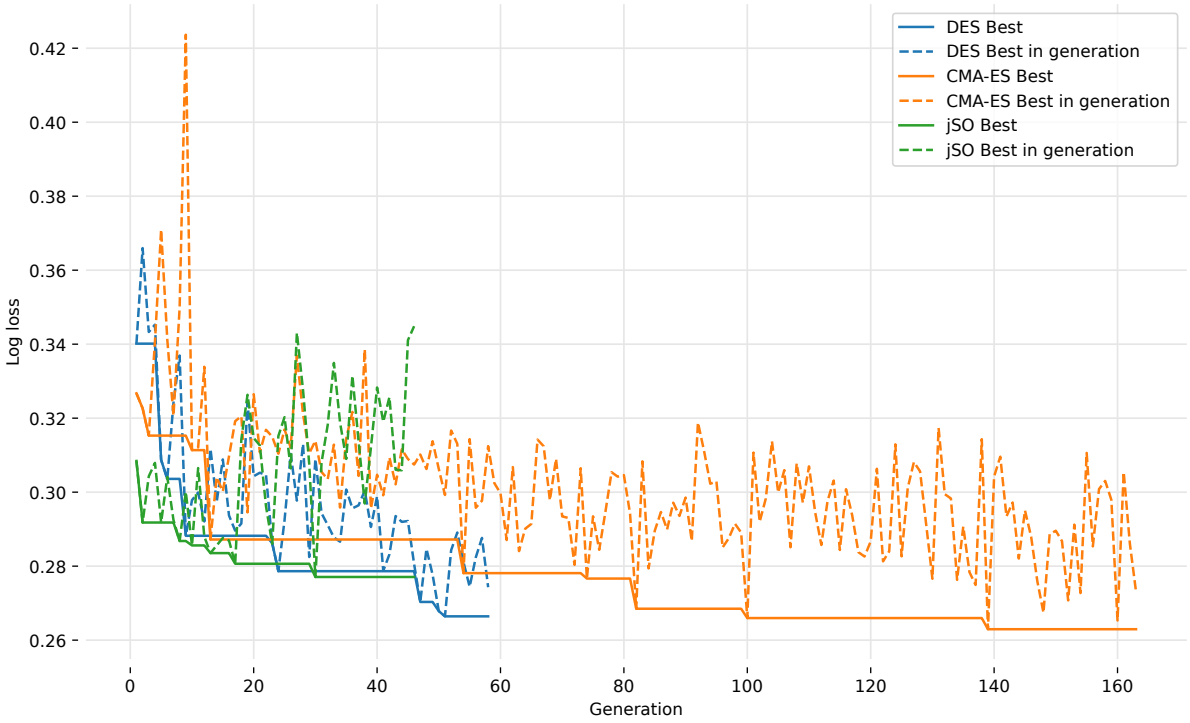
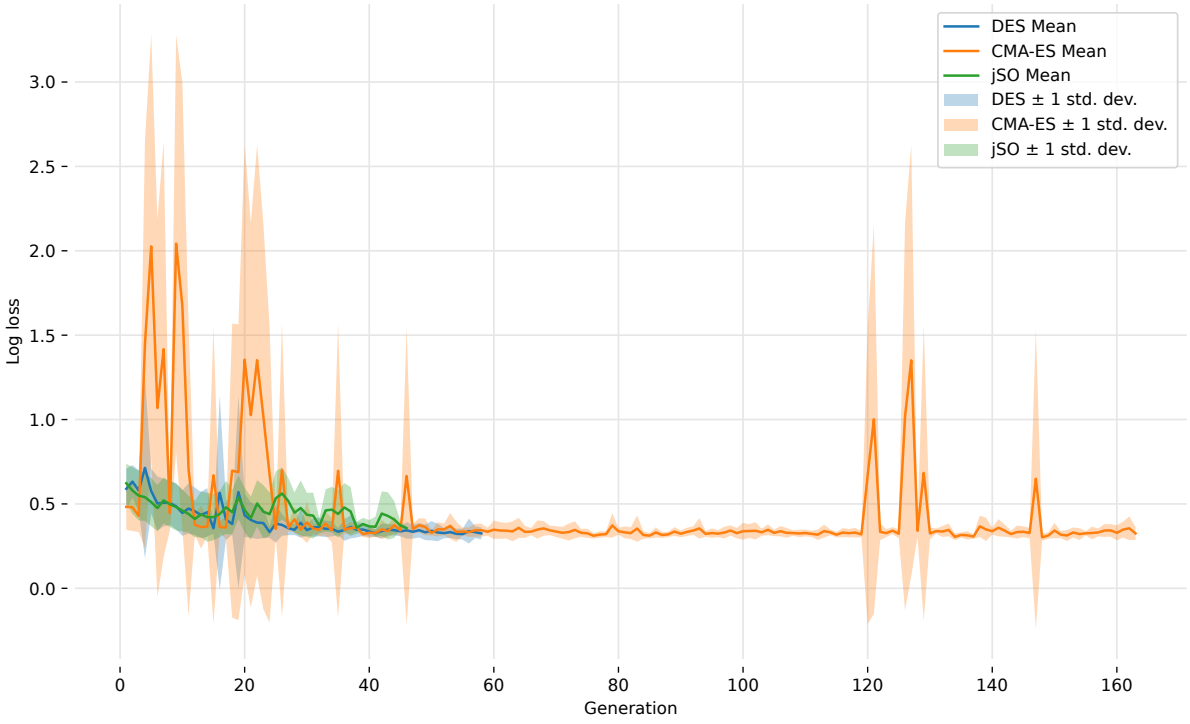
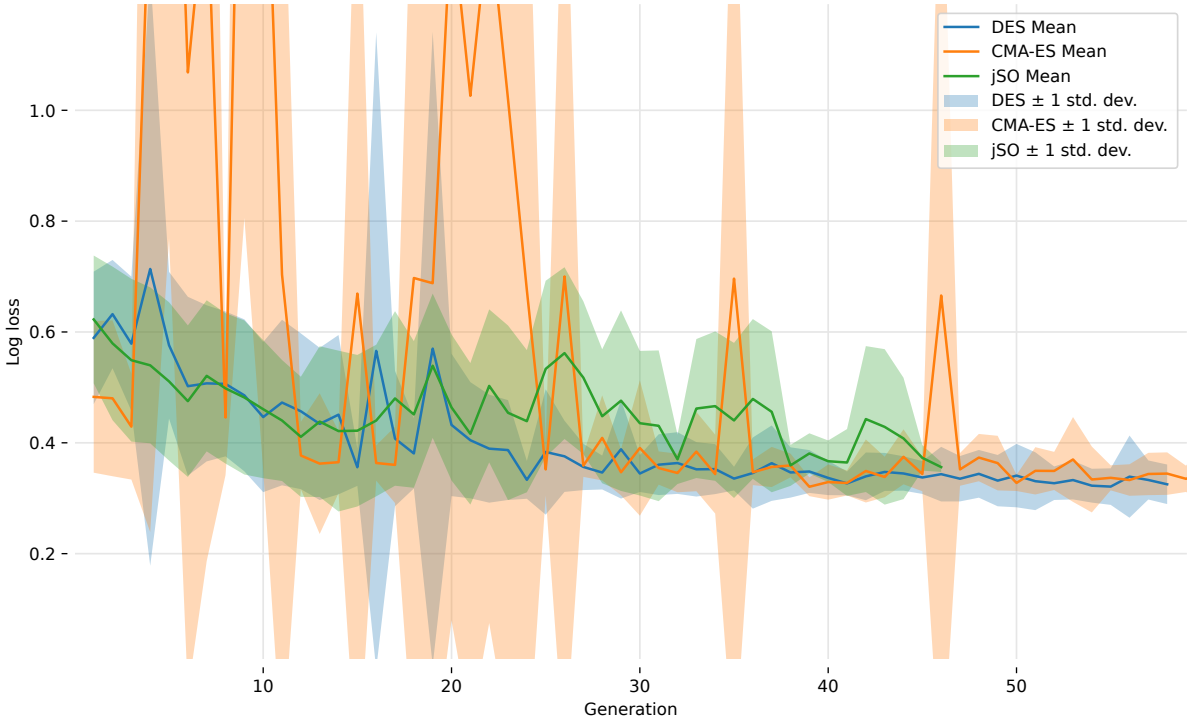


Figure 12. Best fitness in population and 'running' best fitness found during optimization on the Icebergs dataset.



(a) Limited for CMA-ES



(b) Limited for DES and jSO

Figure 13. Mean fitness in generation and standard deviation resulting from optimization on the Icebergs dataset.

Logarithmic loss history for the best individuals is shown in Figure 14. Histories for all individuals indicate that there is no problem of overfitting – either due to the nature of the dataset, or because it has been taken care of through proper hyperparameters. A continuous improvement over the course of the training process is observed, without reaching any plateau. This means that further improvement should be observed if these individuals were given bigger training budget. Interestingly, all optimizers decided upon a fairly low starting learning rate (10^{-3}). In theory its increase could lead to faster convergence, but could also cause problems with stability, which might be an explanation for the chosen values.

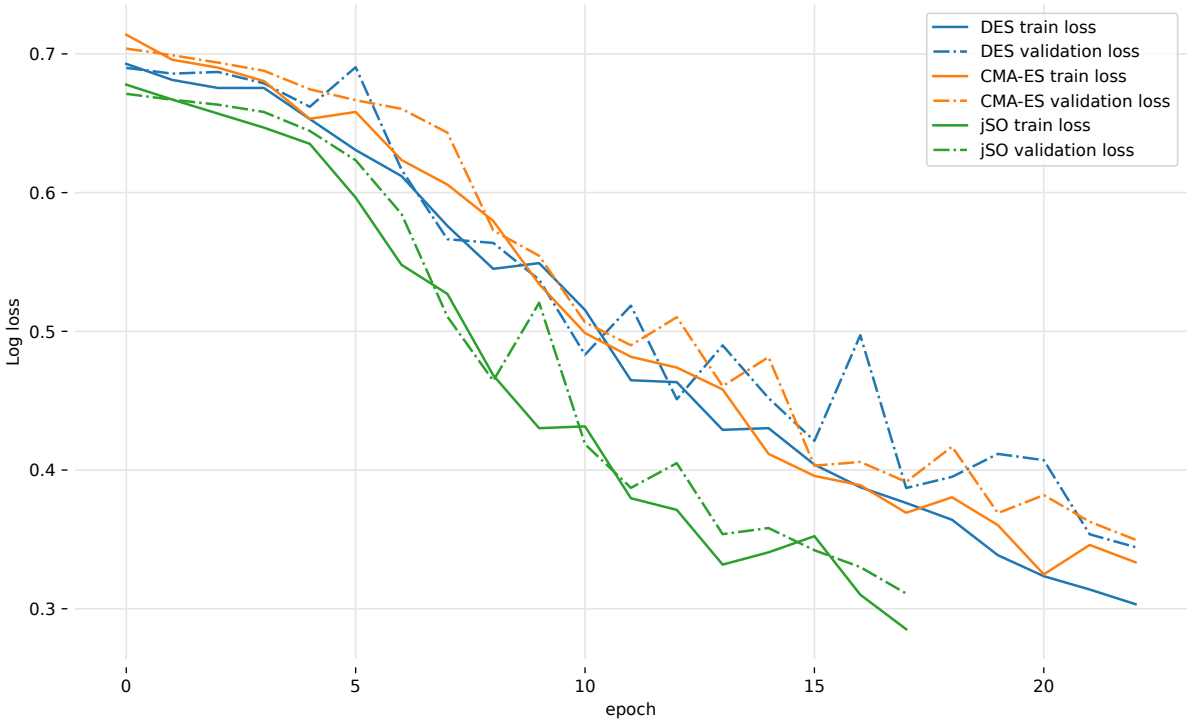


Figure 14. Loss histories for selected individuals found during optimization on the Icebergs dataset.

ROC curves for the best individuals are illustrated in Figure 15 and numeric results can be found in Table 12. DES yields a significantly better individual (over 0.04 difference in log loss) than other two algorithms. This, however, only means that all algorithms should be capable of reaching results presented, but might have worse runs (there are no repeated runs of experiment for Icebergs dataset). Notably, this difference is greater than the ones observed on other datasets. Despite this fact both CMA-ES and jSO come up with individuals that can still be used to achieve reasonable classification results (with RAUCs of roughly 0.91).

Table 12. Results of optimization on the Iceberg dataset.

Heuristic algorithm	Receiver Operating Characteristics		Logarithmic loss
	Area Under Curve	Equal Error Rate	
DES	0.951	0.094	0.307
CMA-ES	0.919	0.176	0.351
jSO	0.912	0.176	0.347

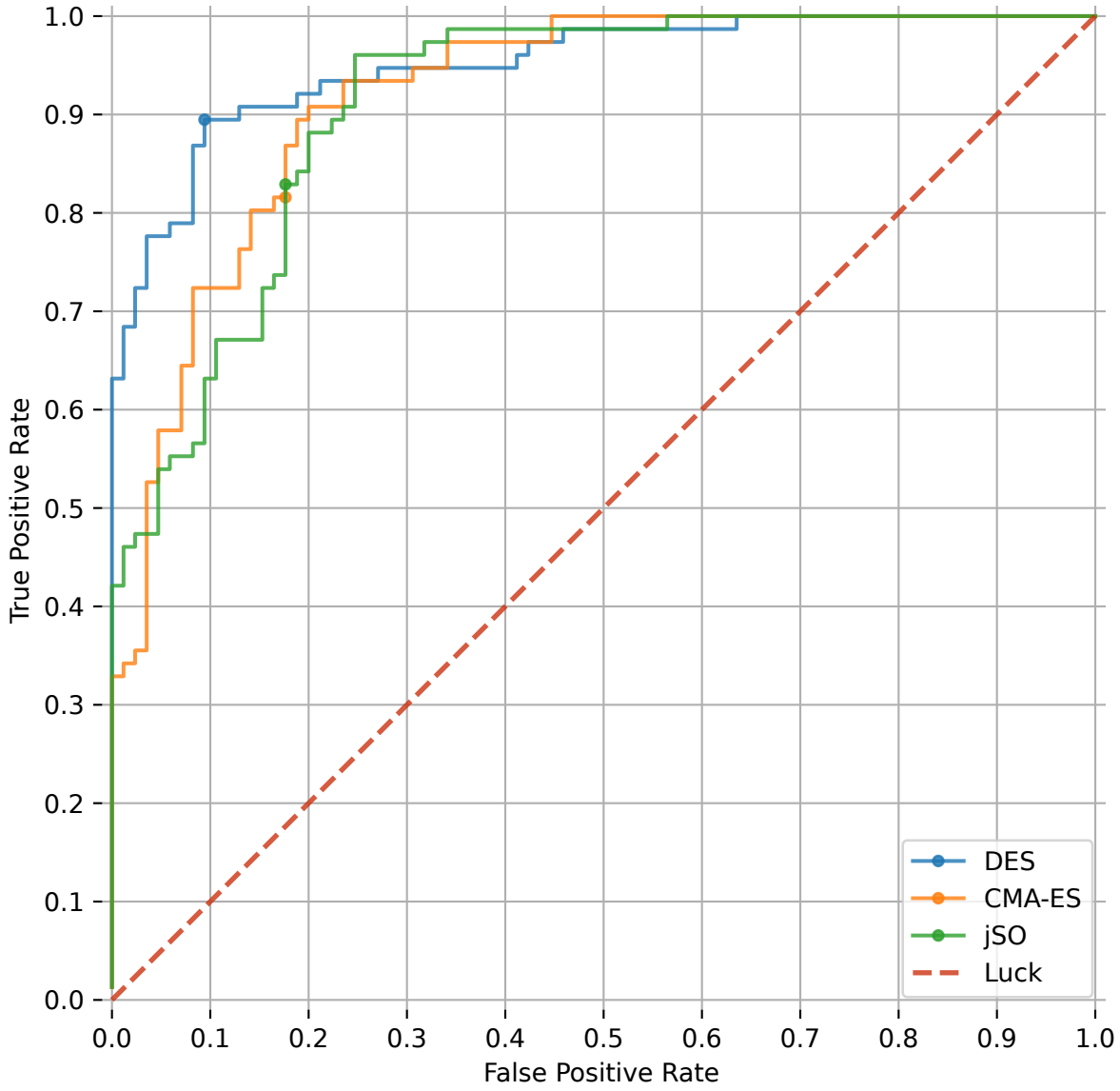


Figure 15. ROC curves for the best solutions found on the Iceberg dataset.

3.4 Results for the Titanic dataset

Results for the Titanic dataset are presented for 10 separate repetitions of optimization process for each algorithm. ECDF curves are depicted in Figure 16. It can be observed that the quality of initial populations vary greatly for this dataset. CMA-ES shows a strong start and steady improvement in the first 70 generations. Both DES and jSO despite notably worse initial populations rapidly reach a 0.83 threshold in a similar manner. At this stage the two diverge, with DES eventually reaching the level of CMA-ES. jSO, on the other hand, improves slowly, but ultimately scores worse than the other two algorithms. Early results favor CMA-ES, but after several hundred function evaluations DES is also a viable algorithm in terms of the population quality. Difference between the two in terms of NEAUC are minor (0.01), with jSO obtaining worse results (0.04 worse NEAUC than DES).

Table 13. ECDF AUC values resulting from optimization on the Titanic dataset. Each curve has been created based on ten independent runs of the experiment.

Heuristic algorithm	EAUC	NEAUC
DES	633.84	0.91
CMA-ES	645.17	0.92
jSO	605.66	0.87

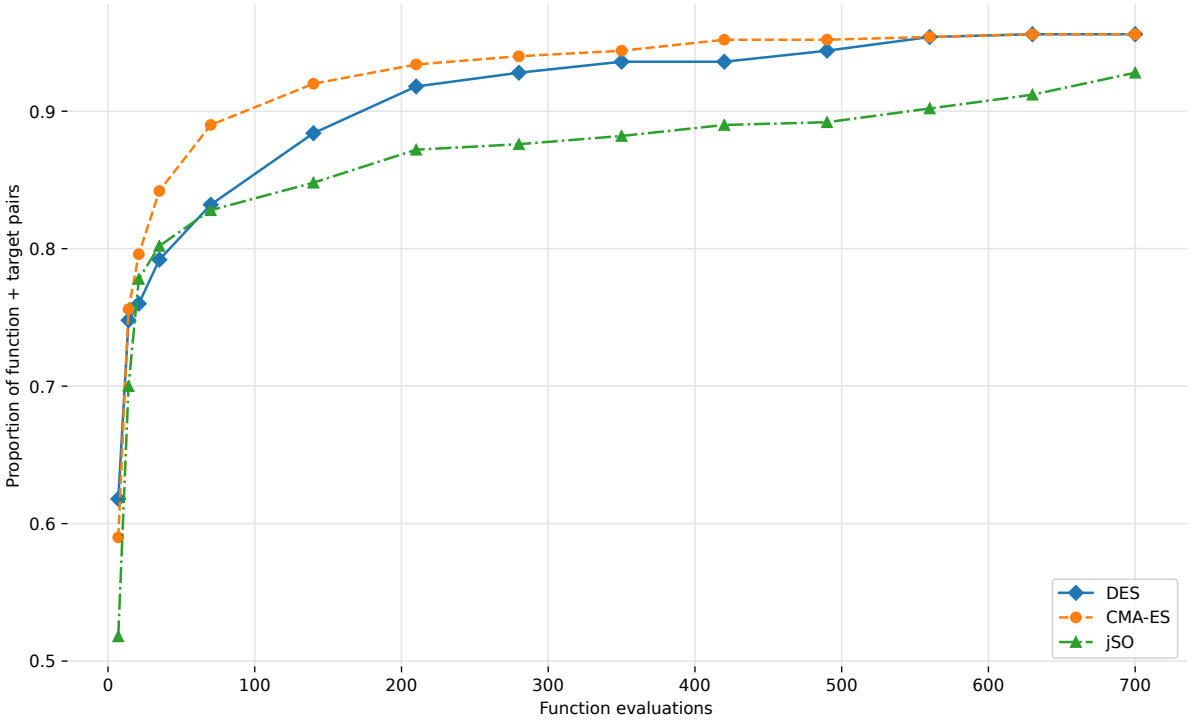
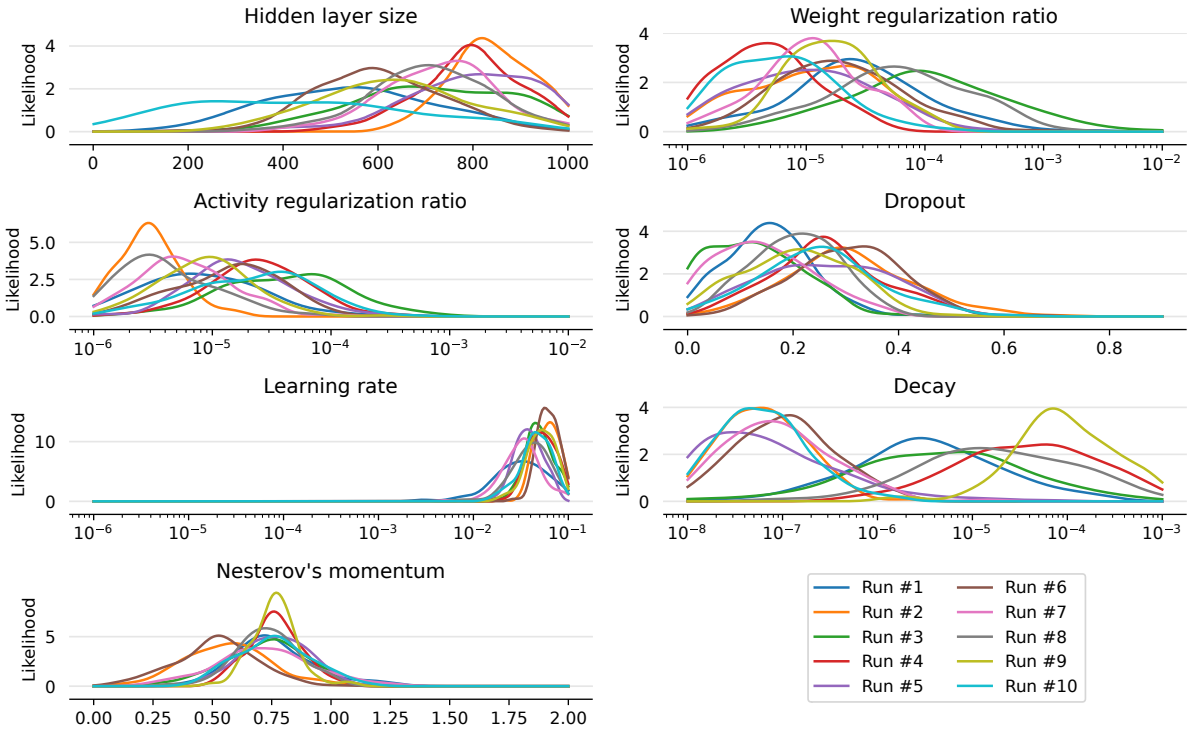
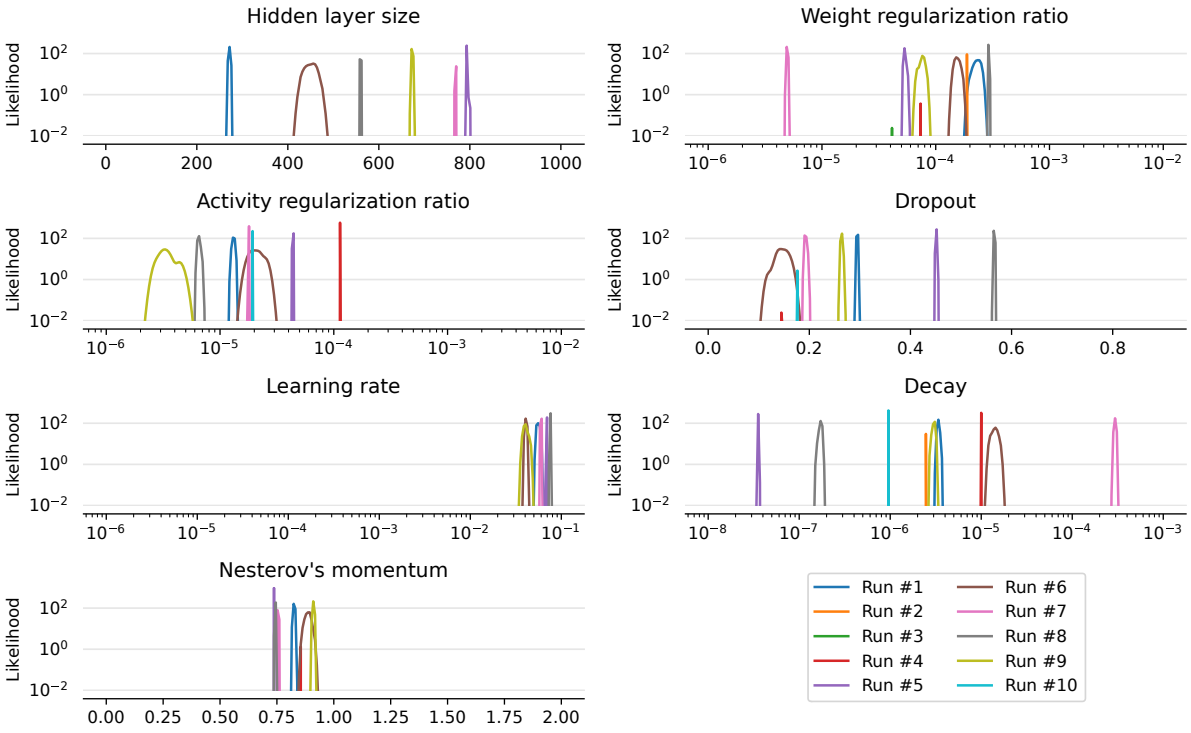


Figure 16. ECDF curves resulting from optimization on the Titanic dataset.

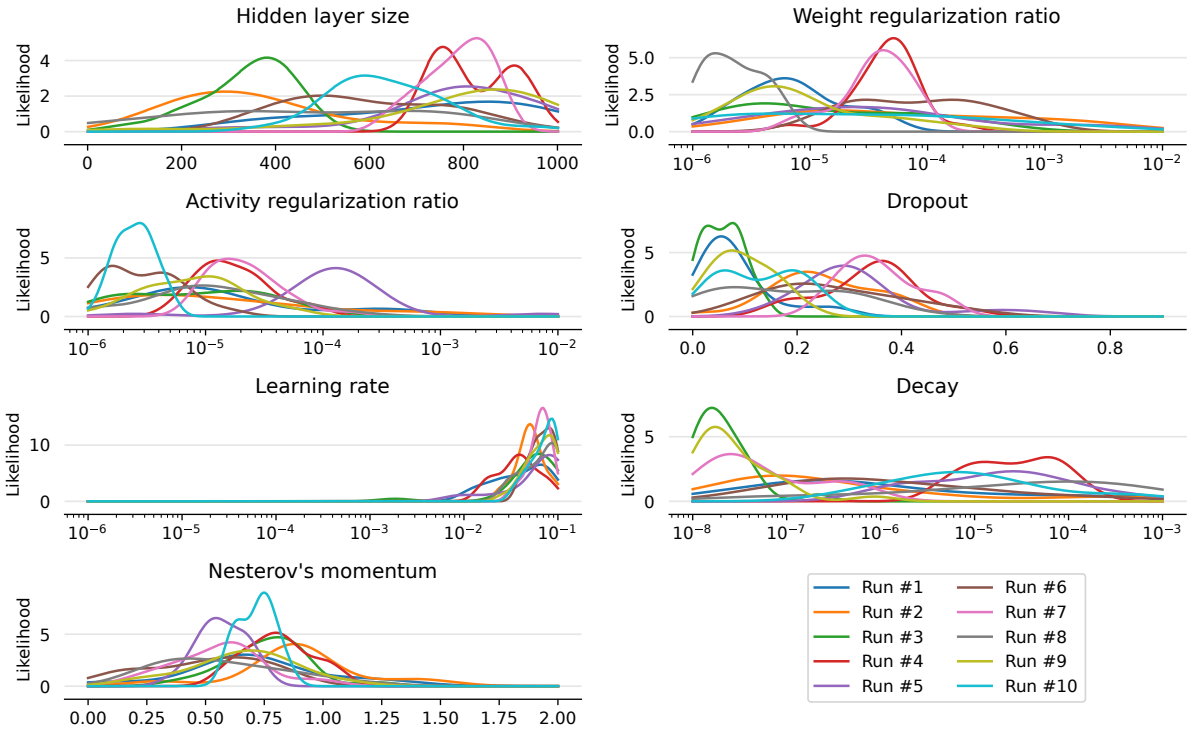
Distribution plots for each algorithm are presented in Figure 17. In terms of architecture, it seems that hidden layer size distributions vary mildly between algorithms and experiment runs. Around 300 neurons mark a soft border, which is crossed by only 2 experiment runs. In contrast to the Aspartus dataset, regularization rates are in general lower, indicating less overfit tendencies during classifier's training. This is further validated by, on average, lower dropout, which is predominantly centered around 0.2 for DES and jSO. CMA-ES also provides two runs which yield dropouts over 0.4, with the rest of experiments being placed around 0.2. Similar pattern can be observed in Nesterov's momentum distribution, where all heuristics are almost uniformly centered around 0.75, with CMA-ES again being placed roughly 0.05 to the right. Learning rate values are high (between 10^{-2} and 10^{-1}). This trend is even stronger than the one seen for the Aspartus dataset, and it further strengthens the theory that shorter range for this hyperparameter should've been chosen, under assumed conditions (budget, architecture etc.). Learning rate decay is uniformly distributed over different runs of experiments, highlighting the fact that such values are too low to have a real impact on the training process and that wider range should be used in the optimization process.


Hidden layer size
Weight regularization ratio
Activity regularization ratio
Dropout
Learning rate
Decay
Nesterov's momentum
Likelihood
Run #1
Run #2
Run #3
Run #4
Run #5
Run #6
Run #7
Run #8
Run #9
Run #10

(a) DES


Hidden layer size
Weight regularization ratio
Activity regularization ratio
Dropout
Learning rate
Decay
Nesterov's momentum
Likelihood
Run #1
Run #2
Run #3
Run #4
Run #5
Run #6
Run #7
Run #8
Run #9
Run #10

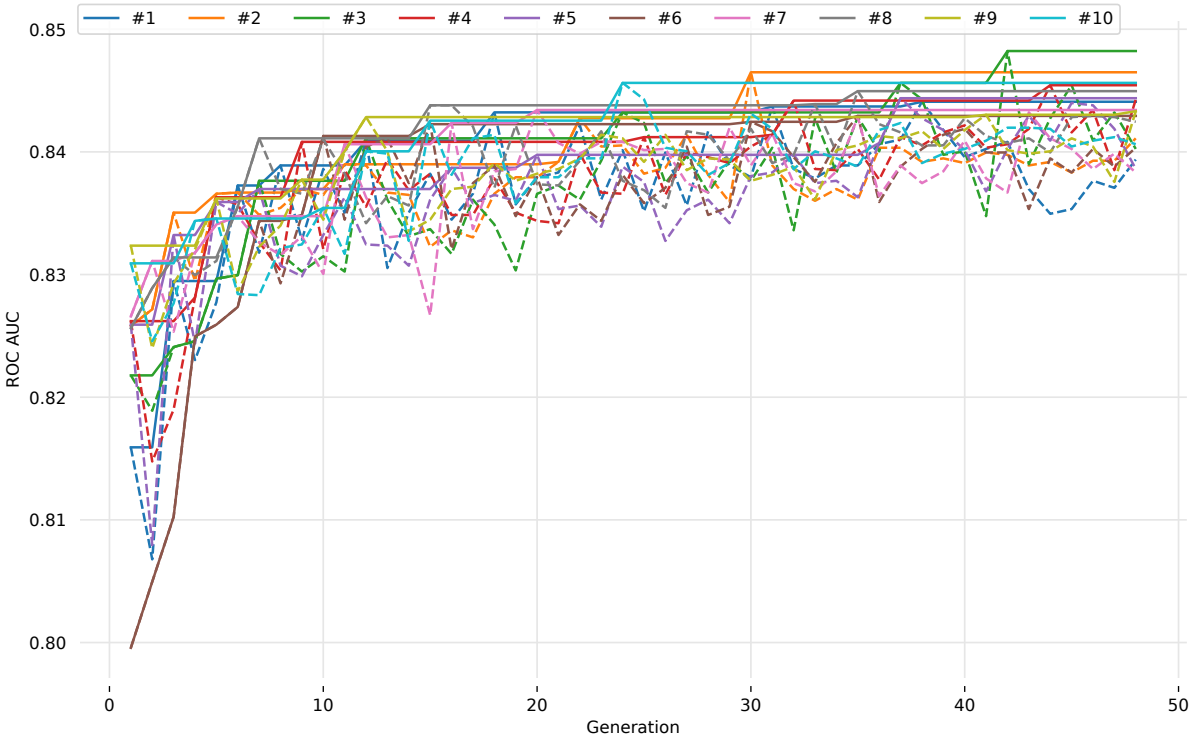
(b) CMA-ES



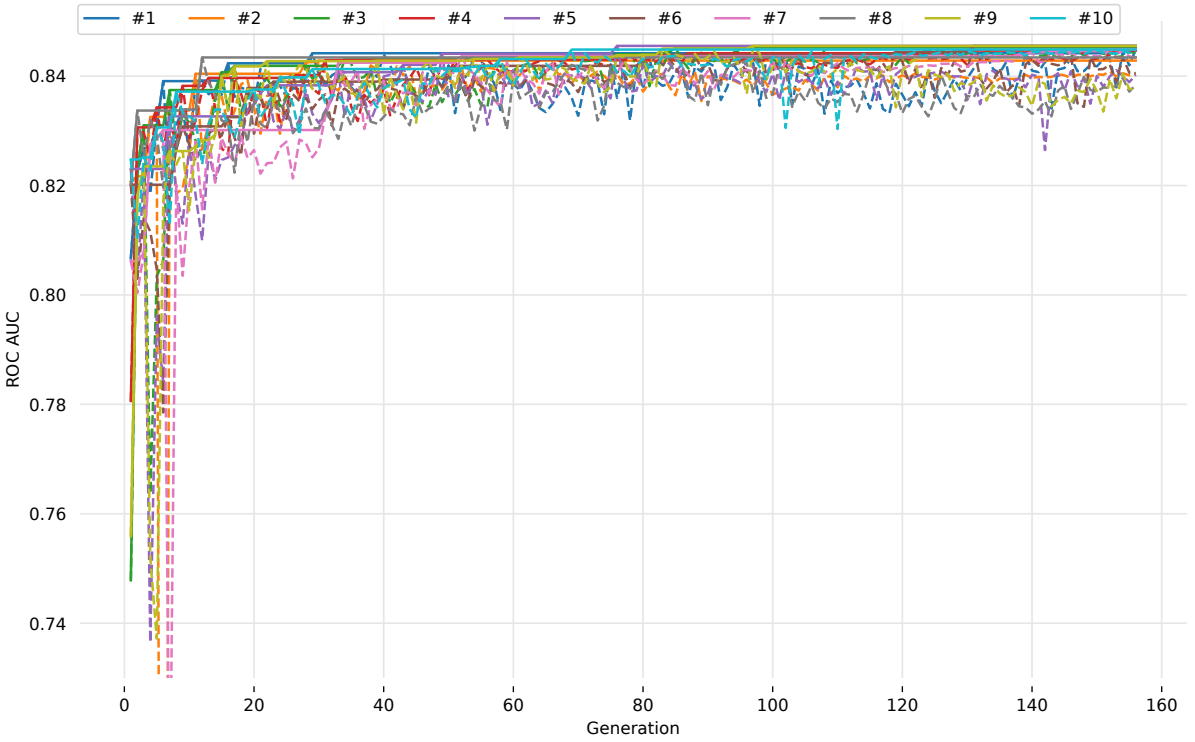
(c) jSO

Figure 17. Hyperparameters' distribution estimation resulting from optimization on the Titanic dataset. Plots for the first and last five generations. Numbers in the legend correspond to the number of the experiment run.

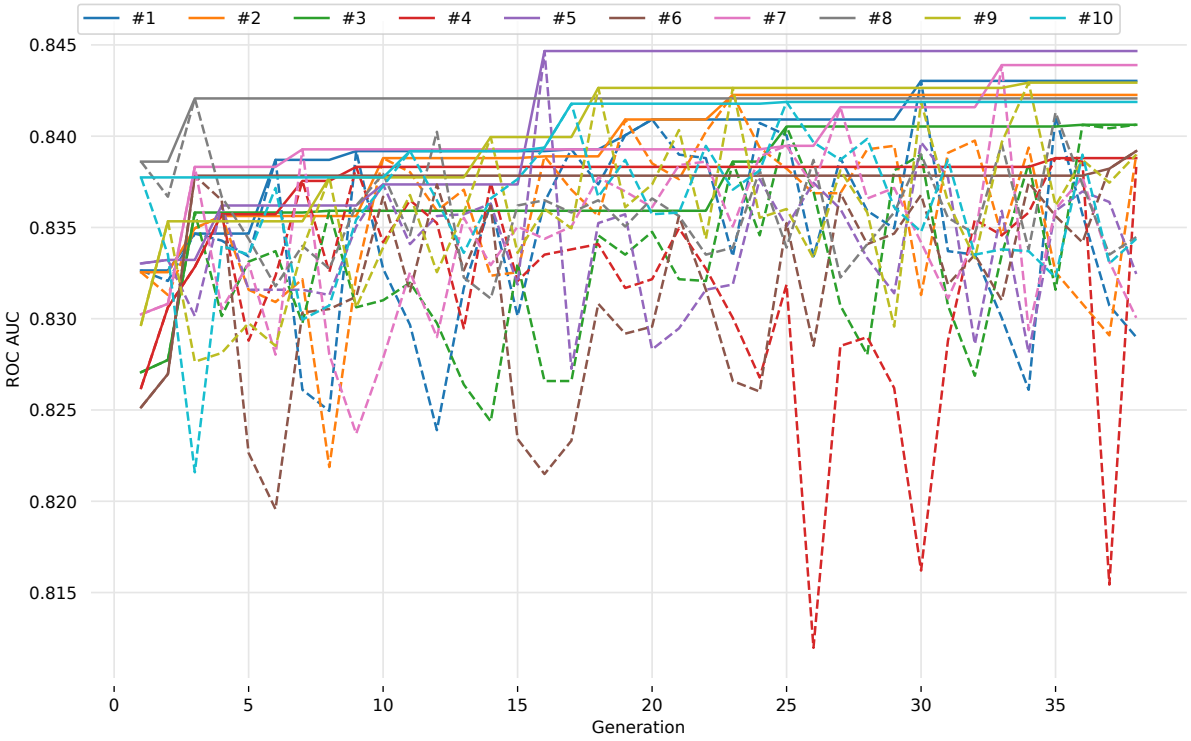
Plots of fitness for the 'running' best individual and best individual in generation are presented in Figure 18. Interestingly, despite having a strong early lead in terms of ECDF score, first 10 generations of CMA-ES seem to greatly vary in the performance of best individuals. This is partially due to the fact that CMA-ES has the smallest population size (9) and in parts caused by the nature of the algorithm itself. However, around the 15th generation it starts converging and shows a stable, but small improvement over time, with a limited variation between best individuals in each generation. On the other hand, early on DES improves in a slower and less aggressive fashion, but seems to be carrying the momentum and steadily improving over time. jSO's has the least stable learning characteristic, with significant variations of the best individual's fitness in each generation. Nonetheless, this behavior is consistent with algorithm's design, as it is the only tested algorithm with changing population size. It starts with a large population and this is visible in the performance of best individuals in early generations, and it shrinks the population size significantly over time, thus resulting in greater variance in the best individual, as there is smaller number of individuals to choose the best from. Results show that when using provided parameters on the given problem, jSO prefers exploration over exploitation.


#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
ROC AUC
0.85
0.84
0.83
0.82
0.81
0.80
0
10
20
30
40
50
Generation

(a) DES


#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
ROC AUC
0.84
0.82
0.80
0.78
0.76
0.74
0
20
40
60
80
100
120
140
160
Generation

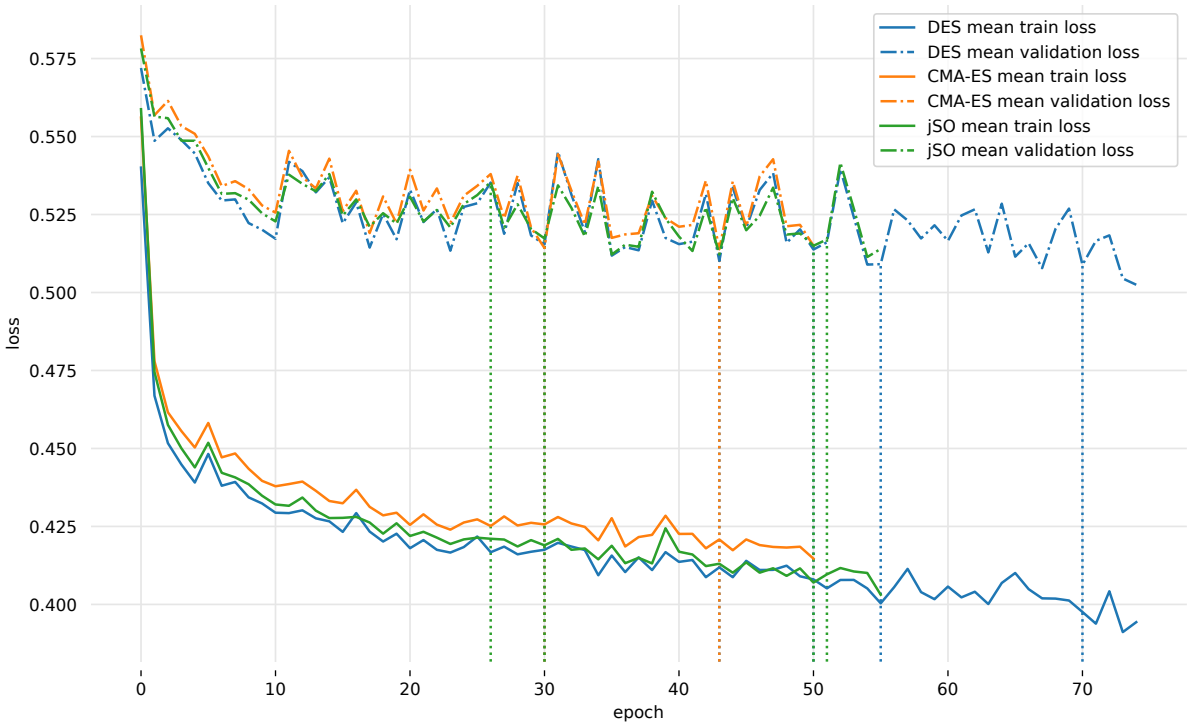
(b) CMA-ES



(c) jSO

Figure 18. Best fitness in population (dashed lines) and 'running' best fitness (solid lines) found during optimization on the Titanic dataset. Color associated with the number of the run is depicted in legends.

Overfit is significant for the Titanic dataset, based on the loss history graphs for best individuals, shown in Figure 19. In both mean and best plots there is a gap of roughly 0.1 loss difference between training and validation subsets. No algorithm is able to effectively bridge this gap. Moreover, loss on validation subset is oscillating, which might indicate too high learning rate or too small learning rate decay. Another interesting observation is the number of epochs reached. All models were given 150 epochs for the training process, however best solutions never reach more than 60 epochs — their training is stopped when they do not show any improvement on the validation set over the span of 20 epochs.



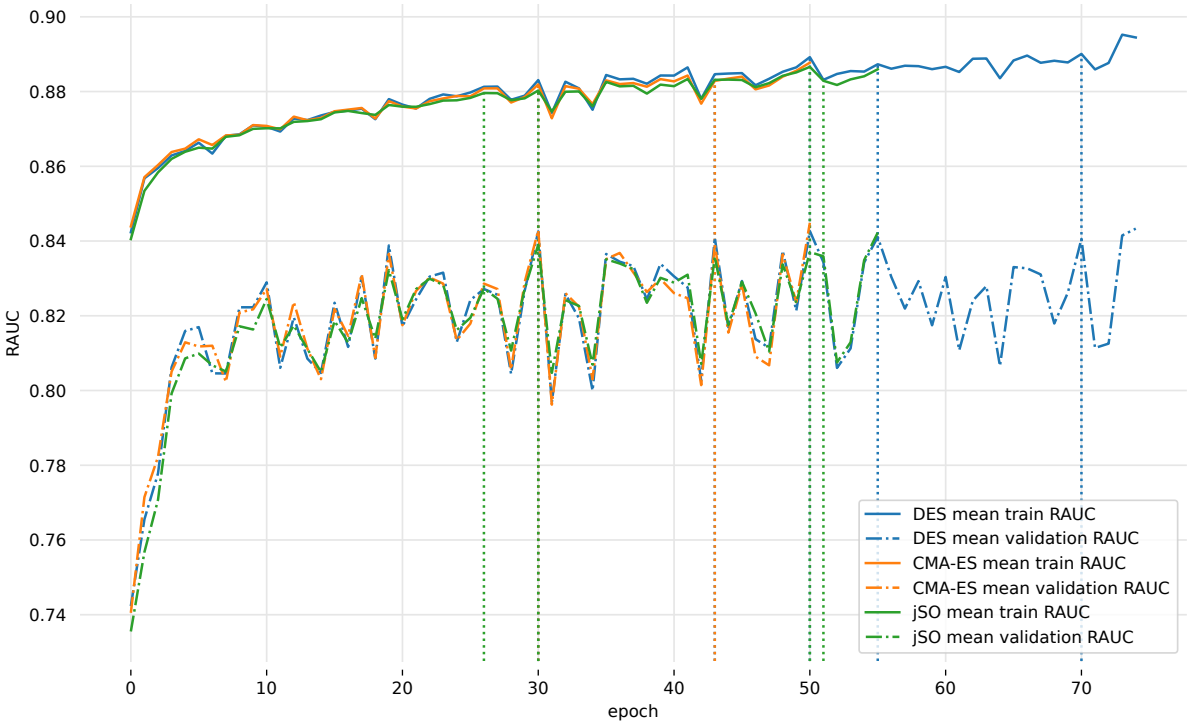
(a) Mean

(b) Best

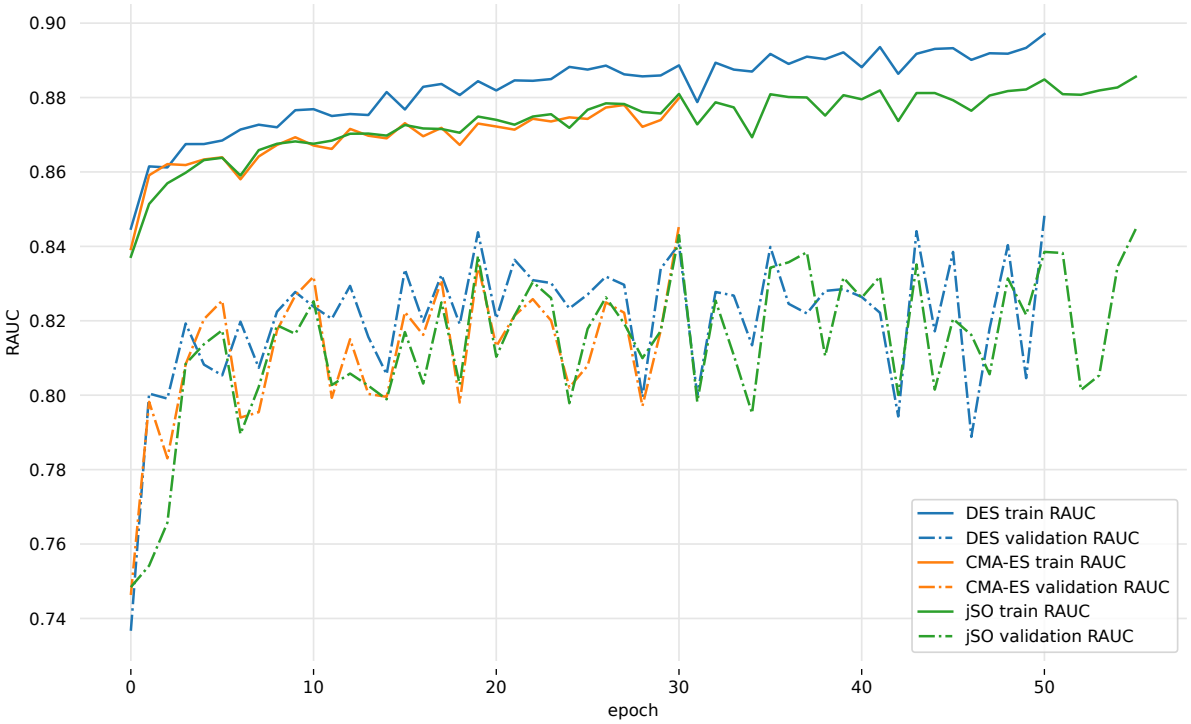
Figure 19. Loss histories for selected individuals found during optimization on the Titanic dataset. Dotted vertical lines in the means plot denote early training termination of a classifier.

Analogously to loss histories, RAUC histories show great similarities between individuals from different optimizers. These plots can be seen in Figure 20. Overfit is also clearly visible, and

no optimizer seems to be dealing well with it. Curvature and oscillation of the plots highlights one of the potential problems with the way individuals are assessed, namely choosing RAUC value at the last epoch, as it does not take into the consideration the shape of the history plot (characteristic of the training). Rather the optimizer tries to set such hyperparameters that will ensure terminating training process after the right (highest RAUC yielding) epoch.



(a) Mean



(b) Best

Figure 20. RAUC histories for selected individuals resulting from optimization on the Titanic dataset. Dotted vertical lines in the means plot denote early training termination of a classifier.

Results on the test subset can be seen in Figure 22 and Figure 21. Full results of all experiments can be found in the Appendix, Table A.2. Both DES and CMA-ES yield very similar RAUC results (medians of respectively 0.871 and 0.872), with the former having slightly greater variation. jSO seems to perform better (median of 0.878), with a crucial exception of one outlier run, which scores noticeably worse (0.859 RAUC). Outliers are also present for EER results, this time in both directions, and for all optimizers. CMA-ES and jSO have identical medians (0.182), while the median for DES is slightly worse – 0.185.

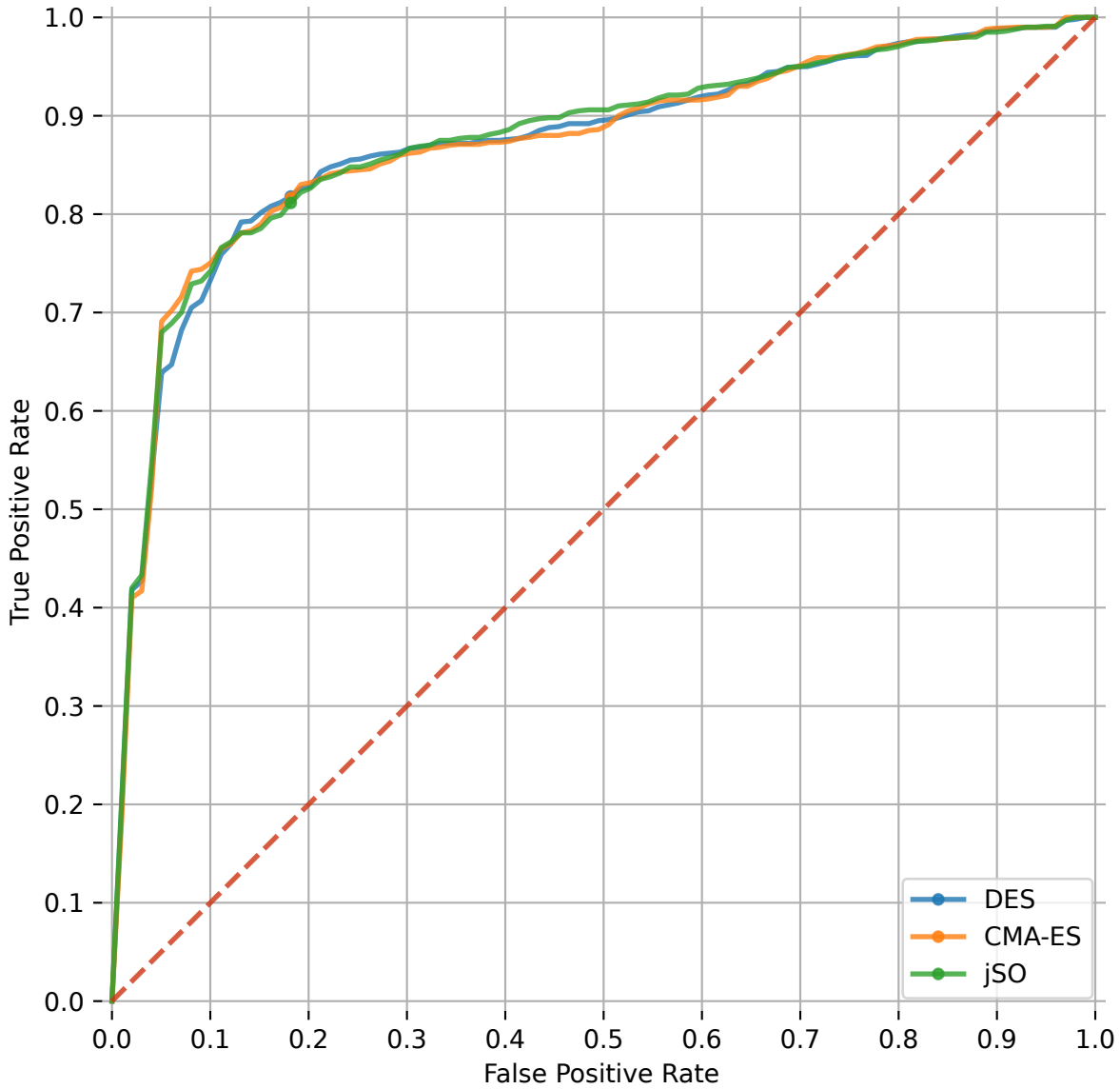
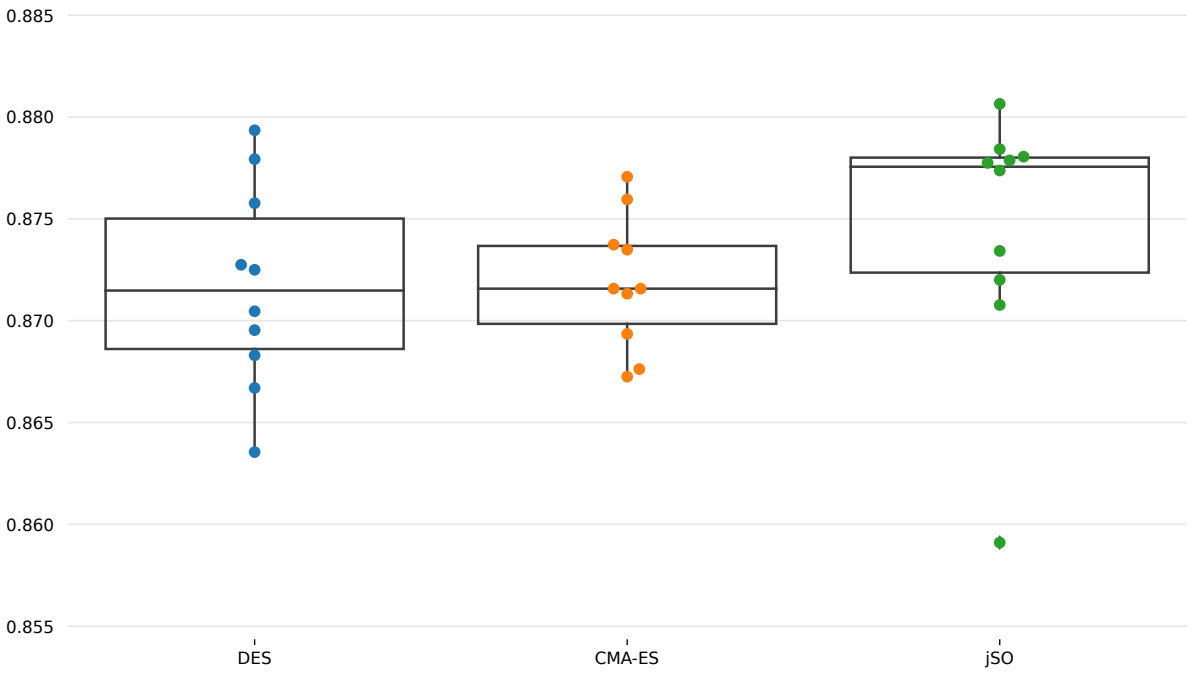
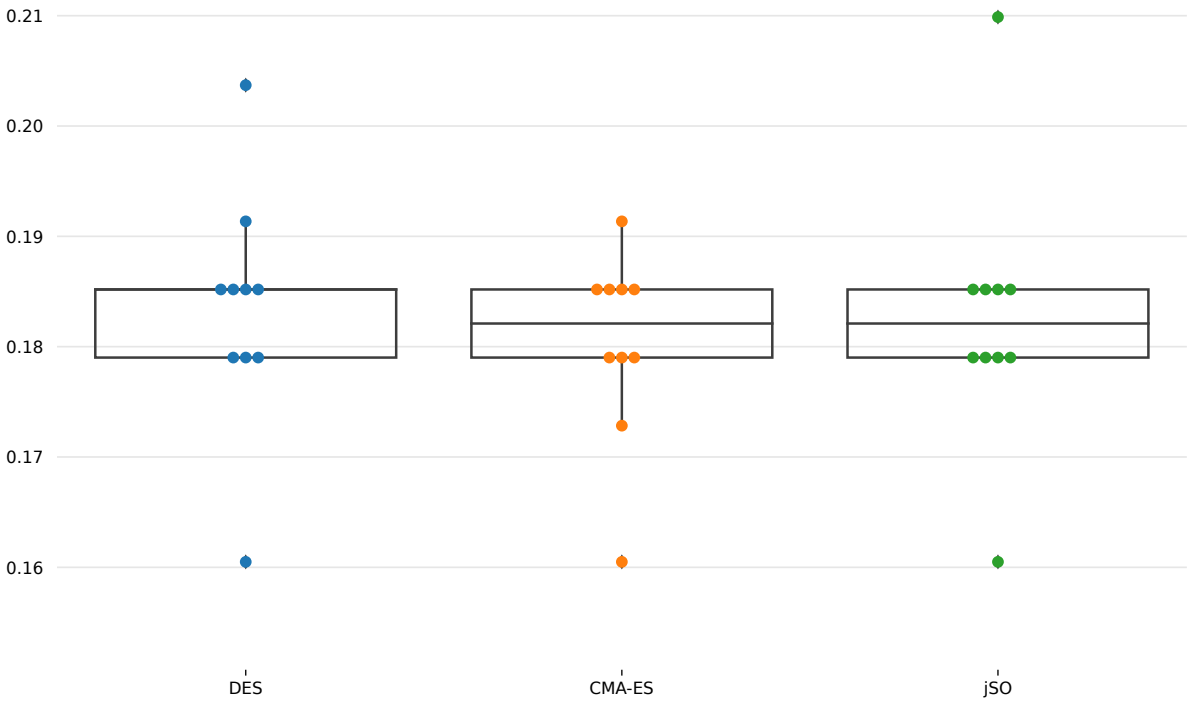


Figure 21. Mean ROC curves for the best solutions found during optimization on the Titanic dataset. Curve for Logistic Regression model is provided as a reference.



(a) ROC AUC



(b) Equal Error Rate

Figure 22. Box plots of RAUC and EER values for each run of the experiment on the Titanic dataset. Each dot denotes a single run of the experiment

4 Summary

4.1 Final conclusions

The main goal of the study was to compare CMA-ES, DES and jSO algorithms and evaluate their performance in the neural network's hyperparameter tuning problem. Based on the carried out experiments I conclude that all three algorithms are suitable for hyperparameter optimization task for MLP and shallow CNN architectures. Notably, there is a lack of substantial differences in performance amongst best found individuals by different optimizers. Moreover, results are comparable amongst multiple runs of an experiment using the same optimizer, which indicates that these algorithms can be used in a 'one-shot tuning' fashion, without the need to repeat the optimization process.

While all algorithms would eventually yield similar results their optimization characteristics differ. Usually DES and CMA-ES show greater similarity in the optimization characteristics, with jSO slightly underperforming, throughout most of the optimization. Therefore it is advised to use either CMA-ES or DES if hardware does not allow for the prolonged tuning process or optimization time is a matter.

Another way to speed up the tuning process is reducing the number of optimized hyperparameters. Depending on the classification problems, a dropout layer could not be needed, or regularization could be turned off. On tested datasets learning rate decay did not influence the results, and as such it could possibly be excluded from tuning.

Even though non-Lamarckian approach was used with severe penalty for crossing the boundaries (effectively killing the individual), optimizers were able to choose hyperparameter values near boundaries (e.g. learning rates in Figure 4 and Figure 17).

As described in [5], CMA-ES can be used as an alternative for the Gaussian optimization approaches – after 5 minutes of training the model on the MNIST dataset it has managed to find an individual, which achieved roughly 0.55% validation error, being only surpassed by the Gaussian TPE. However, after 30 minutes of training it became the leading method, with validation error of 0.27%. Results obtained in this thesis indicate that DES jSO algorithms can compete with CMA-ES optimizer on tested datasets – the difference between best individuals was minimal for 2 datasets (difference of 0.001 RAUC). The one-shot experiment showed better performance of DES over comparable results of CMA-ES and jSO (respectively $0.951, 0.912$ and 0.912 RAUC), although that difference could be at lest in part motivated by randomness. I therefore conclude, that DES and jSO could be a valid alternatives for CMA-ES algorithm for hyperparameter tuning problems, especially on problems with high dimensionality.

4.2 Future research

Scalability with respect to architecture and dataset used

While all tested algorithms managed to find satisfying solutions, datasets and classifiers used were fairly minimal. It could be beneficial to run similar experiments on bigger datasets and more complex architectures to see how does optimization process scale with mentioned factors.

Scalability with respect to number of tuned hyperparameters

In theory, DES should scale better performance-wise with the number of optimized hyperparameters. Seeing as deep neural networks have potentially dozens, if not hundreds of hyperparameters, this could be exploited to achieve even better results by optimizing higher number of these hyperparameters. This approach requires a significant computational power, but is a way to naturally scale the optimization process horizontally with more available hardware.

Different initialization of population strategy

Initialization can often have a significant impact on the quality of the early solutions. Combined with the fact that often users have a good guess for decent hyperparameters, it could be beneficial to change initialization strategies for different algorithms. Obviously, such strategies touch on the trade-off between exploration and exploitation, but I theorize that a good initialization strategy can further boost optimization process. Proposed initialization strategies are:

- changing the distribution from which initial population is sampled for DES and jSO. Currently these algorithms use uniform distribution between the 5th and 95th percentile of optimized attribute's value. Given a starting point these approaches could use normal, beta or student's `t`-distribution;
- providing starting point directly to CMA-ES (currently the starting point chosen is the middle of the value ranges for all attributes);
- if multiple starting points are proposed, then these could be directly used as part of the initial population, while the rest of the population is sampled in a canonical way. This strategy could be easily used in DES and jSO optimizers.

jSO's parameters tuning

Usually when a tuning algorithm is used to optimize hyperparameters, user does not tune parameters of the optimizer itself. Parameters for jSO have not been tuned as a part of this thesis and as such it is possible that different set of parameters could yield overall better results. By researching the impact of jSO's parameters on its performance in the hyperparameter optimization setting, an optimal set of parameters could be found, for which satisfying results are reached on variety of datasets and architectures.

Improved objective function

Objective function could be more sophisticated, i.e. taking under consideration the slice of training history, rather than looking at the last training epoch. This could lead to smoother learning curve of classifiers and could help with robustness of the found individuals.

Difference between jSO implementations

Re-implementation of the jSO algorithm written in the Python language for this thesis has in general been performing better, than the original implementation. The source of this difference is unknown, as I have tried to keep the former identical to the latter. One possible explanation could be the usage of vastly different pseudo random number generators. Further researching differences between these two implementations could possibly improve the jSO algorithm's

performance on low-dimensional problems (performance on high-dimensional problems were not tested).

Bibliography

[1] J. Chi, E. Walia, P. Babyn, J. Wang, G. Groot, and M. Eramian, "Thyroid Nodule Classification in Ultrasound Images by Fine-Tuning Deep Convolutional Neural Network," *Journal of Digital Imaging*, vol. 30, no. 4, pp. 477–486, 2017, doi: 10.1007/s10278-017-9997-y.

[2] T. M. Quan, T. Nguyen-Duc, and W.-K. Jeong, "Compressed Sensing MRI Reconstruction with Cyclic Loss in Generative Adversarial Networks," *CoRR*, 2017, Accessed: May 20, 2018. [Online]. Available: http://arxiv.org/abs/1709.00753

[3] L. de Oliveira, M. Paganini, and B. Nachman, "Learning Particle Physics by Example: Location-Aware Generative Adversarial Networks for Physics Synthesis," *Comput. Softw. Big Sci.*, vol. 1, no. 1, p. 4, 2017, doi: 10.1007/s41781-017-0004-6.

[4] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[5] I. Loshchilov and F. Hutter, "CMA-ES for Hyperparameter Optimization of Deep Neural Networks," *ArXiv e-prints*, Apr. 2016.

[6] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization," *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012, Accessed: May 13, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=2188385.2188395

[7] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," *Advances in Neural Information Processing Systems 24*. Curran Associates, Inc., pp. 2546–2554, 2011. Accessed: May 13, 2018. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf

[8] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential Model-Based Optimization for General Algorithm Configuration," in *Learning and Intelligent Optimization*, C. A. C. Coello, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–523.

[9] J. Snoek *et al.*, "Scalable Bayesian Optimization Using Deep Neural Networks," in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., in Proceedings of Machine Learning Research, vol. 37. Lille, France: PMLR, 2015, pp. 2171–2180. Accessed: May 13, 2018. [Online]. Available: http://proceedings.mlr.press/v37/snoek15.html

[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., pp. 2951–2959, 2012. Accessed: May 13, 2018. [Online]. Available: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[11] E. Ronald and M. Schoenauer, "Genetic lander: An experiment in accurate neuro-genetic control," in *Parallel Problem Solving from Nature — PPSN III*, Y. Davidor, H.-P. Schwefel, and R. Männer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 452–461.

[12] S. Rostami and F. Neri, "A fast hypervolume driven selection mechanism for many-objective optimisation problems," *Swarm and Evolutionary Computation*, vol. 34, pp. 50–67, 2017, doi: https://doi.org/10.1016/j.swevo.2016.12.002.

[13] M. Jaderberg *et al.*, "Population Based Training of Neural Networks," *CoRR*, 2017, Accessed: May 20, 2018. [Online]. Available: http://arxiv.org/abs/1711.09846

[14] N. Hansen and A. Ostermeier, "Completely Derandomized Self-Adaptation in Evolution Strategies," *Evol. Comput.*, vol. 9, no. 2, pp. 159–195, Jun. 2001, doi: 10.1162/106365601750190398.

[15] D. Jagodziński and J. Arabas, "Towards a Matrix-free Covariance Matrix Adaptation Evolution Strategy," *IEEE Transactions on Evolutionary Computation*, 2018.

[16] J. Brest, M. S. Maučec, and B. Bošković, "Single objective real-parameter optimization: Algorithm jSO," in *2017 IEEE Congress on Evolutionary Computation (CEC)*, 2017, pp. 1311–1318. doi: 10.1109/CEC.2017.7969456.

[17] N. H. Awad, M. Z. Ali, J. J. Liang, B. Y. Qu, and P. N. Suganthan, "Problem Definitions and Evaluation Criteria for the CEC 2017 Special Session and Competition on Single Objective Bound Constrained Real-Parameter Numerical Optimization," Singapore, 2016.

[18] R. Tanabe and A. S. Fukunaga, "Improving the search performance of SHADE using linear population size reduction," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 1658–1665. doi: 10.1109/CEC.2014.6900380.

[19] Ł. Neumann, R. Okuniewski, R. Nowak, and P. Wawrzyński, "Using machine learning algorithms for the prediction of customer decision in car insurance," *IEEE Transactions on Industrial Informatics*, 2018.

[20] "Titanic dataset." Accessed: May 05, 2018. [Online]. Available: http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls

[21] "Statoil/C-CORE Iceberg Classifier Challenge." Accessed: May 05, 2018. [Online]. Available: https://www.kaggle.com/c/statoil-iceberg-classifier-challenge/data

[22] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[24] J. Kiefer and J. Wolfowitz, "Stochastic Estimation of the Maximum of a Regression Function," *Ann. Math. Statist.*, vol. 23, no. 3, pp. 462–466, 1952, doi: 10.1214/aoms/1177729392.

[25] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the Importance of Initialization and Momentum in Deep Learning," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, in ICML'13. Atlanta, GA, USA: JMLR.org, 2013, p. III–1139–III–1147.

[26] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *CoRR*, 2014, Accessed: May 08, 2018. [Online]. Available: http://arxiv.org/abs/1412.6980

[27] A. Y. Ng, "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance," in *Proceedings of the Twenty-first International Conference on Machine Learning*, in ICML '04. Banff, Alberta, Canada: ACM, 2004, p. 78–. doi: 10.1145/1015330.1015435.

[28] A. K. Qin and X. Li, "Differential evolution on the CEC-2013 single-objective continuous optimization testbed," in *2013 IEEE Congress on Evolutionary Computation*, 2013, pp. 1099–1106. doi: 10.1109/CEC.2013.6557689.

[29] D. Scott, *Multivariate Density Estimation: Theory, Practice, and Visualization*. in A Wiley-interscience publication. Wiley, 1992.

[30] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, Accessed: May 03, 2018. [Online]. Available: http://www.jstor.org/stable/3001968

[31] F. Mosteller and R. A. Fisher, "Questions and Answers," *The American Statistician*, vol. 2, no. 5, pp. 30–31, 1948, Accessed: May 27, 2018. [Online]. Available: http://www.jstor.org/stable/2681650

[32] R. Elston, "On Fisher's Method of Combining p-Values," vol. 33, pp. 339–345, 1991.

List of Figures

List of Tables

List of Appendices

Appendix A Tables

Table A.1. RAUC and EER values for the best individuals found during optimization on the Aspartus dataset.

Heuristic algorithm	Run number	Receiver Operating Characteristics	
		Area Under Curve	Equal Error Rate
DES	1	0.674	0.366
	2	0.678	0.360
	3	0.678	0.352
	4	0.673	0.365
	5	0.665	0.379
	6	0.651	0.393
	7	0.649	0.381
	8	0.676	0.371
	9	0.666	0.383
	10	0.679	0.359
CMA-ES	1	0.683	0.356
	2	0.676	0.357
	3	0.674	0.362
	4	0.655	0.392
	5	0.672	0.367
	6	0.668	0.372
	7	0.648	0.396
	8	0.681	0.346
	9	0.657	0.379
	10	0.646	0.388
jSO	1	0.679	0.353
	2	0.680	0.362
	3	0.666	0.365
	4	0.659	0.376
	5	0.671	0.371
	6	0.665	0.376
	7	0.677	0.346
	8	0.652	0.372
	9	0.653	0.383
	10	0.675	0.365
Logistic Regression[1]		0.654	0.392

[1] Provided for reference, not part of the optimization process.

Table A.2. RAUC and EER values for the best individuals found during optimization on the Titanic dataset.

Heuristic algorithm	Run number	Receiver Operating Characteristics	
		Area Under Curve	Equal Error Rate
DES	1	0.868	0.185
	2	0.879	0.179
	3	0.864	0.160
	4	0.873	0.179
	5	0.870	0.179
	6	0.867	0.185
	7	0.872	0.204
	8	0.878	0.191
	9	0.870	0.185
	10	0.876	0.185
CMA-ES	1	0.873	0.179
	2	0.868	0.191
	3	0.872	0.185
	4	0.872	0.173
	5	0.876	0.160
	6	0.867	0.179
	7	0.877	0.179
	8	0.869	0.185
	9	0.871	0.185
	10	0.874	0.185
jSO	1	0.873	0.179
	2	0.859	0.210
	3	0.871	0.160
	4	0.878	0.185
	5	0.878	0.185
	6	0.877	0.185
	7	0.881	0.179
	8	0.878	0.179
	9	0.878	0.179
	10	0.872	0.185