

MEng Individual Project

IMPERIAL COLLEGE LONDON

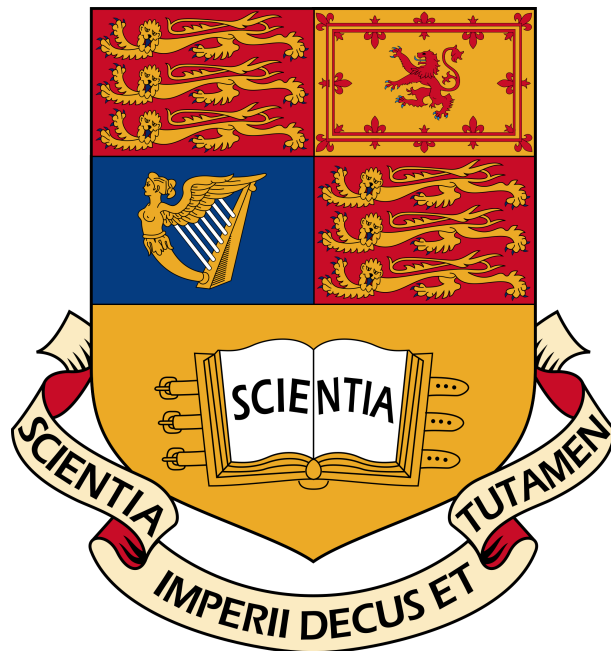
DEPARTMENT OF COMPUTING

Efficient Design of Machine Learning Hyperparameter Optimizers

Supervisor: Dr. Jonathan Passerat-Palmbach

Author: Cristian Matache

Second Marker: Dr. Bernhard Kainz



June 17, 2019

Abstract

Performance of machine learning models relies heavily on finding a good combination of hyperparameters. We aim to *design the most efficient hybrid* between two best-in-class hyperparameter optimizers, Hyperband and TPE. On the way there, we identified and solved a few problems:

1. Typical metrics for comparing optimizers are neither quantitative nor informative about how well an optimizer generalizes over multiple datasets or models.
2. Running an optimizer several times to collect performance statistics is time consuming/impractical.
3. Optimizers can be flawed: implementation-wise (eg. virtually all Hyperband implementations) or design-wise (eg. first published Hyperband-TPE hybrid).
4. Optimizer testing has been impractical because testing on true ML models is time-consuming.

To overcome these challenges, we propose: *Gamma loss function simulation*, *Closest known loss function approximation* and a more comprehensive set of metrics. All three are benchmarked against published results on true ML models¹. The simulation and the approximation complement each other: the first makes testing practical for the first time and serves as a theoretical ML model while the latter allows for timely collection of statistics about optimizer performance as if it was run on a true ML model.

Finally, we use these to find the best hybrid architecture which is validated by comparison with Hyperband and TPE on 2 datasets and several mathematical hard-to-optimize functions. The pursuit for a hybrid is legitimate since it outperforms Hyperband and TPE alone, but there is still some distance to the state-of-the-art performances.

Keywords hyperparameter optimizer design, Hyperband-TPE hybrid, instant optimizer testing, instant loss function simulation, loss function approximation, optimizer evaluation, deep learning

¹Note that there are *separate evaluation sections* for: Best hybrid design, Gamma simulation, Closest known loss function approximation, Quantitative metrics

Acknowledgements

I would like to give a massive thank to my supervisor Jonathan Passerat-Palmbach for his guidance, dedication and enthusiasm throughout this year. Also, I owe everything to my wonderful parents and girlfriend who comforted me with their unconditional love and support.

All professors from Imperial College London and from my former high-school deserve appreciation for the hard work they put in teaching and preparing me for life. Thank you to my colleagues and mentors in Barclays algo quant research group for coaching me and for introducing me to applied research.

The Gamma simulation was made possible with valuable feedback and questions coming from Jonathan and from the rest of the OpenMOLE community, most notably: Romain Reuillon, Juste Raimbault (Institut des Systemes Complexes de Paris - Centre National de la Recherche Scientifique) and Pierre Peigne (Ecole 42).

Contents

1	Introduction	5
1.1	Problem description	5
1.1.1	Machine learning	5
1.1.2	Hyperparameters	5
1.1.3	Optimization	5
1.2	Motivation for automatic hyperparameter tuning	6
1.3	Human interaction	6
1.4	Summary of methods	6
1.5	Existing optimizers	7
1.6	Computing resources	7
1.7	Objectives	7
1.8	Report outline	8
2	Hyperparameters	10
2.1	Measuring the error of a model	10
2.1.1	Methods	11
2.1.2	Error metrics	11
2.2	Architecture	12
2.2.1	Neural networks	12
2.2.2	Decision trees	12
2.2.3	Hyperparameters	13
2.3	Learning	13
2.3.1	Neural networks	13
2.3.2	Decision trees - Iterative Dichotomiser 3 (ID3)	14
2.3.3	Hyperparameters	14
2.4	Avoiding overfitting	14
2.4.1	Neural networks	14
2.4.2	Decision trees	15
2.4.3	Hyperparameters	15
2.5	Loss functions	15
2.6	Summary	16
3	Optimizers - methods background	17
3.1	Basic methods	17
3.1.1	Manual search	17
3.1.2	Grid search	18
3.1.3	Random search	18
3.2	Bayesian Methods	19
3.2.1	Method description	19
3.2.2	Advantages	22
3.2.3	Disadvantages	22
3.2.4	Variations	23
3.3	Genetic Algorithms	26
3.3.1	Method description	26
3.3.2	Advantages	27
3.3.3	Disadvantages	27
3.4	Reinforcement learning	27

3.4.1	Method description	27
3.4.2	Advantages	27
3.4.3	Disadvantages	28
3.5	Hyperband	28
3.5.1	Method description	28
3.5.2	Advantages	30
3.5.3	Disadvantages	30
3.5.4	Variation: Hyperband+TPE	30
3.6	Comparison - heat map	31
3.7	Conclusion	31
4	Experimental setup	32
4.1	MNIST	32
4.2	CIFAR-10/CIFAR-100	32
4.3	Future experiments	33
5	Optimizer evaluation criteria	34
5.1	Metrics	34
5.1.1	Description	34
5.1.2	Reducing metrics to loss functions	34
5.2	PROFILES - statistical profiles of loss functions	35
5.2.1	Profiles	35
5.2.2	Comparing profiles	36
5.3	Estimated density of optimal final errors (EPDF-OFE)	37
5.3.1	Example	37
5.3.2	Quantitative comparison of EPDF-OFEs	38
5.3.3	Ideal evaluation: profiles of multi-model multi-dataset EPDF-OFEs	39
5.4	Typical optimizer evaluation metrics in literature	39
5.5	Summary	39
6	Optimizer correctness and testing	40
6.1	Floating point error on all Hyperband Python implementations	40
6.2	Wrong Hyperband+TPE hybrid design in paper	42
6.3	Summary	43
7	Gamma loss function simulation	44
7.1	Motivation	44
7.2	Simulation requirements	45
7.3	Solution	45
7.3.1	Underlying function	45
7.3.2	Gamma process model	48
7.3.3	Families of shapes and model parametrization	50
7.3.4	Normally distributed noise	53
7.3.5	Scheduling families of shapes	54
7.3.6	Further clarification on model parametrization	54
7.4	Previous work	54
7.5	Optimizers using the simulation	55
7.6	Evaluation	56
7.6.1	With respect to the initial requirements	56
7.6.2	Reproducing a real-world problem	56
7.6.3	Time complexity	57
7.6.4	Catching errors (for testing)	57
7.6.5	Features and benefits	57
7.6.6	What can be improved	58
7.7	Summary and achievements (so far)	58

8	Closest known loss function approximation	59
8.1	Motivation	59
8.2	Solution	59
8.2.1	Data collection	59
8.2.2	Closest known function	60
8.2.3	Optimizers using the approximation	61
8.3	Evaluation	62
8.3.1	Low approximation errors	62
8.3.2	EPDF-OFE experiment evaluation	62
8.3.3	Time complexity	63
8.3.4	Features and benefits	63
8.3.5	What can be improved	63
8.4	Motivation for MNIST Logistic Regression	64
8.5	Summary and achievements (so far)	64
9	Best Hyperband-Bayesian hybrid architecture	65
9.1	Architecture of pure optimizers	65
9.1.1	Hyperband	65
9.1.2	Bayesian optimizers	66
9.2	Architecture of hybrids	66
9.3	Evaluation criteria	67
9.4	Proposed hybrid architectures - history transfer schemes	68
9.5	Evaluation	69
9.5.1	Closest known loss function approximation	69
9.5.2	MNIST - real machine learning	70
9.5.3	Gamma simulation - pure/flat functions	71
9.5.4	Preliminary results	75
9.5.5	Gamma simulation - families of shapes	75
9.5.6	CIFAR10 - real machine learning	78
9.5.7	What can be improved	78
9.6	Conclusion	79
10	Conclusion	80
10.1	Problematics	80
10.2	Contributions	80
11	Future work	82
11.1	Given more time	82
11.2	Challenges	82
11.3	Further work	82
A	Implementation	87
A.1	Designed for architecting optimizers and for tuning parameters	87
A.2	Autotune 2.0 design principles	87
A.3	Customizable pipeline	87
A.4	Strengths and weaknesses	88

Chapter 1

Introduction

Artificial Intelligence (AI) is becoming increasingly popular with machine learning, in particular, being omnipresent in several domains spanning across industries. Its applications vary from prediction of diseases to fraud detection.

1.1 Problem description

The accuracy of machine learning models is now a driving force behind whole businesses and behind sensitive applications. To achieve a good performance, a machine learning algorithm must be carefully tuned/calibrated. This task is typically time consuming because it requires re-training the model many times. Also, it is complicated to visualize or to fully understand for humans because it involves high dimensions. Automating this process brings countless benefits like state-of-the-art accuracies and reproducibility of experiments. First of all, let us break the problem down and describe the main notions.

1.1.1 Machine learning

Machine learning is a branch of AI which consists of making a computer learn from some observations (datasets) and to become increasingly better with respect to a task. For instance, some usual tasks are predictions - regressions or classifications. An example regression is estimating the price of houses in a neighbourhood based on their surfaces. Deciding if an incoming email is spam or non-spam is a classical example of classification. So, a key distinction between regressions and classifications is the type of the prediction space: continuous versus discrete (labels).

1.1.2 Hyperparameters

Formally, hyperparameters are parameters that belong to a machine learning model which are set before a machine learning model is trained. Not to be confused with the parameters that the model learns automatically at training time as part of the learning task. The performance of a model is sensitive to the configuration of hyperparameters, so we need to optimize these values.

Please note that, in the end, some automatic tuning methods may have their own hyperparameters that, in turn, need to be tuned either manually or by another automatic non-parametric technique.

This is where the notion of **AutoML** (Automated machine learning) comes in. AutoML aims to automate the application of machine learning as a whole, that involving algorithmically finding the suitable model and its hyperparameters when being given only a prediction problem/dataset.

1.1.3 Optimization

Theoretically, optimization means minimizing a given function. Note that this definition of optimization also covers maximization because it can be reduced to minimization of $-f$ where f is the given function (to be maximized). In practice, finding some approximate values may suffice. Convex functions are a special class of functions that can be easily optimized. Take the real-valued function $f(x) = x^2$ for example. It has a unique global minimum at $x = 0$. Conversely, non-convex functions are more interesting and one must be careful not to get stuck in a local optimum.

1.2 Motivation for automatic hyperparameter tuning

The need for automatic hyperparameter optimization is definitely increasing as machine learning applications become pervasive in today's world. Below, we will have a quick look at a few reasons why automating this process is so important:

- **complexity:** the larger the number of hyperparameters grows the harder it is for humans to see and to track how they are correlated. Often, hyperparameters depend on each other. So optimizing each parameter in turn will not solve our problem.
- **critical domains:** some applications of machine learning rely heavily on the performance of their underlying models. A few examples are detecting credit card fraud applications, stock prices predictions, medical diagnosis, facial recognition for e-passports or even optical character recognition (OCR). So, hyperparameter tuning must be done in a reliable way.
- **time constraints:** trying multiple combinations of hyperparameters requires retraining the whole model for each attempt. Training a model once is usually a quite lengthy process and we must do this lots of times. So, there is an obvious need for automation in this case.
- **resource constraints:** Time is not the only binding resource. Computing power is limited. Even though it is increasing with a fast pace, the demand is so high that running at full capacity in terms of computational resources is the norm. Moreover, memory is another resource which needs to be considered. The fields of Internet of Things (IoT) or mobile phones, for example, would greatly benefit from a systematic reduction/optimization of memory consumption. So, often, many applications are restricted by a maximum budget of resources.
- **reproducibility:** without having a well-defined/structured/reproducible approach to find the optimal hyperparameters some published machine learning experiments may not give the same results if run by someone else. There is no guarantee that the behaviour of a retried experiment is preserved if the hyperparameters are different even though the underlying model is the same. There is a number of reasons why researchers do not include parameter tuning in their papers. One is the business potential. Google, for example, did not explain in detail Google Vizier (their tool for automatic hyperparameter tuning) because it can generate revenue for them. Another reason is the complexity of the model, researchers try parameters at random until they get a good enough result. This situation is very hard to replicate [1].

1.3 Human interaction

From a user perspective, automating a task means as little human interaction as possible. So, for hyperparameter tuning the goal is to have a user specifying only *the stopping criteria*:

- **time:** how long the user is willing to wait for
- **performance:** what is a satisfying accuracy for the user

The algorithms will stop as soon as one of the above criteria is met.

1.4 Summary of methods

In literature, to automatically tune hyperparameters there are 3 common (non-trivial) types of methods based on the following:

- Bayesian optimization
- Genetic algorithms
- Reinforcement learning

On top of them, there are a few heuristic methods like Hyperband which aim to speed-up the computation of any type of optimizer. For detailed formal methods, pros and cons as well as comparisons between them please refer to chapter 3 or for a short summary refer to the heatmap comparison 3.1.

1.5 Existing optimizers

To plan ahead, it is very important to understand what solutions already exist, how they work and how they scale. Below, we will give a brief overview for some of the most popular libraries.

- **Heuristics/rules of thumb:** not very powerful nor practical (see "Motivation for automatic hyperparameter tuning" section 1.2)
- **Commercial software:**
 1. *Google Vizier* - uses Gaussian Processes along with some performance improvements [2]. The corresponding paper is quite vague but this is understandable because Vizier is for Google's internal use and can give them a competitive advantage on the market.
 2. *SIGOPT* - uses mostly Bayesian methods. The company not only commercializes software for automatic hyperparameter tuning but also publishes several research papers and articles about this.
- **Bayesian methods:**
 1. *Spearmint, Metric Optimization Engine (MOE)* - use Gaussian Processes.
 2. *Hyperopt* - uses TPE (Tree-structured Parzen Estimators).
 3. *Sequential Model-Based Algorithm Configuration (SMAC)* - uses random forest regression [31].
- **Genetic Algorithms:**
 1. *MGO* - purely functional Scala library that implements state of the art evolutionary algorithms.
 2. *TPOT* - a popular python library that integrates well with scikit learn [3, 4].

The underlying principles for each implementation will be introduced technically later. More information about the open source implementations can be found on their corresponding github pages.

1.6 Computing resources

Machine learning as a field is notorious for using large computing resources. To find hyperparameters one must run the machine learning model several times to compare the results and keep the set of hyperparameters that performs best. This multiplies the need for computing resources. Hence, parallel computation is always desired and considered while evaluating a method.

For very simple learning models CPUs may be enough. However, more sophisticated models in most cases require GPUs. In automatic hyperparameter optimization literature, there are papers that report the usage of 2, 10, 30 up to 800 GPUs. Therefore, for our purposes we must understand resource usage and act in consequence.

Intensive research is going into customized hardware for AI but, at the moment, we must think of a few GPUs as a limit because large resources such as 800 GPUs are very rarely available. For our experiments we will use a few GPUs indeed.

1.7 Objectives

We aim to **find the best ways to employ Hyperband and TPE** in a hybrid optimizer. It appears that **combining them in a specific way is the best option**. On the way, several problems will arise. We will solve and evaluate each of them in part. However, since there are many possible architectures of hybrids, the ultimate goal is to find the state-of-the-art hybrid architecture.

In the end, we aim to apply and evaluate the optimizers proposed in this paper on Convolutional Neural Networks (CNNs) since Deep Learning is typically the field that benefits the most from automatic hyperparameter tuning due to its high complexity. We will benchmark the results against existing methods.

1.8 Report outline

Background

1. **Hyperparameters:** Next, in chapter 2 there will be a short introduction to what hyperparameters are, how they can be grouped by type and what impact they have on a machine learning model. Also, we introduce loss functions which are the most lucrative concept in our experiments. This chapter's contribution is valuable because it explains machine learning concepts used later in the methods background, it foreshadows the challenges that optimizers encounter and clarifies theoretical notions that are used throughout the paper.
2. **Optimizers - methods background:** The following chapter introduces a detailed comparison between the most popular types of methods: Bayesian optimizers, Genetic Algorithms, Reinforcement learning and Hyperband. We also compare several variations/implementations for each category. We will cover the mathematical foundations, comparison by several evaluation criteria as well as advantages and disadvantages. A summary of this chapter can be viewed in a comprehensive heatmap: 3.1. At the end of this chapter, we justify why we pursued the architecture of hybrids between Hyperband and TPE as the main objective of this paper.
3. **Experimental setup:** This is a short chapter that justifies why we picked certain datasets and certain machine learning models in our future experiments.

Our findings - problems and proposed solutions

4. **Optimizer evaluation criteria:** Remember that to find the best hybrid architecture between Hyperband and TPE, thus we need to compare several architectures with Hyperband and between themselves. Hybrids are expected to exhibit some difference in performance but the differences between them are not expected to be huge (even though they can be statistically significant). Typical optimizer evaluation criteria from literature are not particularly indicative in such situations. So, we developed a *comprehensive quantitative set of metrics* to compare optimizers. Our solution tells if one optimizer is consistently better than another with statistical significance and how well an optimizer generalizes to several datasets and ML models. This chapter solves the problem with metrics but introduces another problem (yet to solve): to collect statistics about an optimizer one needs to run it several times and this is very time-consuming/impractical.
5. **Optimizer correctness and testing:** In this chapter, we show that testing for optimizers is almost non-existent. We found that virtually all implementations of Hyperband (including the one proposed by the authors) are flawed due to floating point arithmetic errors. This error has important consequences since it harms exploration and can cause Hyperband to terminate several hours or even days earlier than expected. Next, we found that the design of the first published Hyperband-TPE hybrid breaks a fundamental assumption it makes (operating with a given budget of resources). It is reasonable to assume that testing has not been used for optimizers because one run of an optimizer on true machine learning is extremely time consuming. However, testing is clearly needed to avoid such large scale issues.
6. **Gamma simulation:** The first solution we propose to the problems above is the "Gamma loss function simulation". It *simulates in negligible time the behaviour of true machine learning models*. Gamma simulation fully solves the problem of testing and helps collecting statistics to measure the performance of an optimizer as outlined in the Optimizer evaluation criteria. In the end of this chapter, we provide an extensive evaluation of the Gamma simulation.
7. **Closest known loss function approximation:** This is the second solution that we propose to solve the problem of collecting statistics about an optimizer in a timely manner. Also, by contrast with the Gamma simulation, this approximation is based on data collected from real machine learning. The main goal of this chapter is to approximate in negligible time and with low errors the behaviour of a true machine learning model. At the end of this

chapter, we provide an extensive evaluation of this method and show both quantitatively and qualitatively why it is suitable to use for our purposes as a replacement of true machine learning.

8. **Best Hyperband-TPE hybrid architecture:** Back to our main goal, with the tools described above, we can propose and compare several hybrid architectures. We will benchmark them against the Closest known loss function approximation, several Gamma simulations, 3 hard-to-optimize mathematical functions and on 2 real machine learning problems. The evaluation section is quite large since we wanted to ensure that our hypothesis is backed by enough experiments. Eventually, we validate one of the architectures as being the best not only because it performs very well but also because it scales well.

Chapter 2

Hyperparameters

Next, we will see some hyper-parameter examples and how they are being used in machine learning models. Later, we will group them by types. This chapter will also serve as an introduction to supervised machine learning given that we discuss hyperparameters with respect to basic theoretical models.

This introduction is particularly relevant for automatic hyperparameter tuning because it:

- defines error metrics which will drive the fitness function of tuning algorithms (see genetic algorithms).
- describes the models that our experiments will work on. That is, we will tune models that are described in this chapter.
- clarifies how flexible the automatic tuning methods outlined in various papers are. That is, not all papers optimize the same sets of hyperparameters. For examples, some opt for more rigid architectures fixing the number of layers of a neural network while others are more flexible. This impacts the overall performance and must be taken into account when comparing results.
- allows a more succinct description of the automatic methods and of their own hyperparameters. For example, an automatic method called SMAC uses random forests.

2.1 Measuring the error of a model

Measuring how well a machine learning model will predict/estimate/classify on unseen data is a relatively hard task given that our datasets are samples from some unknown distributions. We cannot do better than pretending some of the data in our dataset has not been seen already. That is, we need to partition our dataset and assign different roles (training, validation, testing).

- **Training:** this is the partition from which the model will learn. Obviously, it should be the largest.
- **Validation:** after deciding upon a model we should find the optimal set of hyperparameters. We try different combinations for which we train the model and benchmark the results versus this validation partition. The set of hyperparameters that yields the best results on the validation set will be retained. Please note that this partition is of great relevance for this paper.
- **Testing:** after having the model and the optimal set of hyperparameters we should find out how well our algorithm generalizes to unseen data. So, we use another part of the original dataset, run the model and see how close/far are we to the expected results.

There are a few ways we can assign these partitions and below we will see the most frequently used methods. Of course, after finding the optimal hyperparameters and the error (on unseen data), we can retrain our model on the entire dataset. Hence, we make effective use of all the dataset in production.

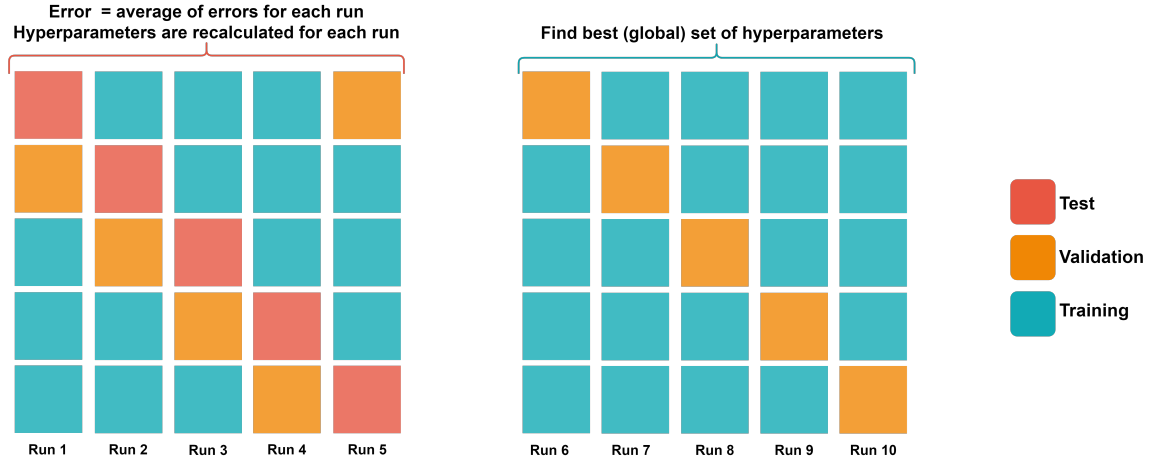


Figure 2.1: Cross Validation - 5 folds

2.1.1 Methods

Holdout

If our dataset is very large, the straightforward approach is to split it in 3 disjoint sets for training, validation and test. The typical ratios are 50:25:25 and 60:20:20. The rule of thumb says the smaller the dataset the larger the training set. The explanation is simple:

- large training partition \rightarrow better prediction (generally)
- large testing partition \rightarrow better error estimation [42]

Cross-validation

Another option which works regardless of the size of the dataset is cross-validation. First, we shuffle randomly then we split the whole dataset in K equal partitions (folds), usually 10. $K - 1$ of them are used for training and validation while the other one is used for testing. We run all possible such assignments (K in total). Then, to see how well the model performs we take the average of the errors on each iteration. However, this approach comes with a catch when it comes to validation:

We use $K - 2$ folds for training, 1 for validation and 1 for testing. We can find the global error but we will obtain different sets of optimal hyperparameters for every iteration. So, after doing this, in order to obtain one global set of hyperparameters we will repeat the same process with $K - 1$ folds for training and 1 fold for validation.

Comparison

Cross-validation is more robust. It can be seen as applying holdout multiple times and this makes it less biased. On the downside, we need to find the value for K but this is a manageable trade-off. [28]

2.1.2 Error metrics

To compute the errors we can use various measures which are more domain specific. For example, precision, recall, F1, accuracy are used for classification while MSPE (mean squared prediction error) or r-squared are used for regressions.

2.2 Architecture

Below we will see how a few models look from an architectural point of view. Note that both neural networks and random forests are suitable for classifications as well as regressions. Please bear in mind that we want to identify and understand hyperparameters, as usual.

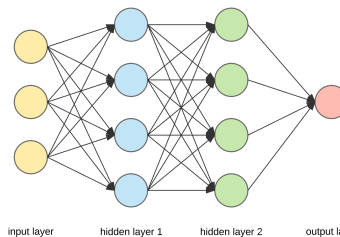
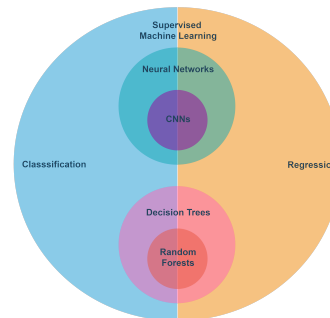
2.2.1 Neural networks

Perceptrons are the building block of neural networks. They are also called artificial neurons because they mimic the mechanics of a biological neuron. A perceptron combines linearly its inputs (which are outputs of other perceptrons) with some strengths called "weights". Based on this aggregation we decide if the perceptron is activated or not.

Neural networks are networks of interconnected perceptrons.

Note that they do not necessarily have different activation functions. A typical example are feed forward neural networks.

We have just introduced a lot of hyperparameters like number of neurons, number of neurons per hidden layer.



Eg. feed forward network [45]

Convolutional neural networks (CNNs)

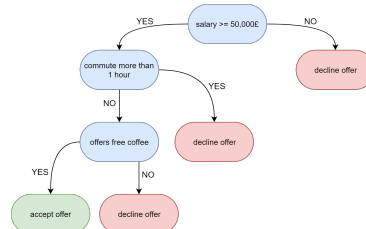
CNNs are a particular type of feed-forward neural networks which have been proven successful in several domain specific applications like medical imaging or self driving cars. They are usually the model of choice when the underlying dataset is comprised of images.

Physically, a CNN is built on: convolutional layers, non linearity, pooling/subsampling layers and fully connected layers [39]. Convolutions are meant to extract features like edges from images. The filters/kernels for these convolutions are controlled by 3 hyperparameters: depth, stride and zero-padding. Pooling means a space reduction while retaining principal components [39]. The hyperparameter that defines subsampling/pooling is the choice of a function (eg. maximum or sum). The fully connected layer is a like a new neural network embedded in the CNN that takes the reduced features (output from several convolutions and pooling layers) and performs classification on them. Due to their complexity, they take long times to train. For our purposes, CNNs are of interest since they require a large set of hyperparameters.

2.2.2 Decision trees

"Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability." Tom Mitchell [5]

Practically, we learn a tree structure that fits the training data and predicts based on a top-down greedy search through the learned tree. The leaves represent the classes/values. The decision tree essentially splits the dataset into subsets that are more homogeneous. This makes it quite robust to noisy data.



Eg. decision tree based on [46]

Random forests

After having introduced the decision trees, random forests are very easy to understand. Basically, we train a certain number of decision trees with random samples from the dataset. When it comes to predictions we ask all trees in the forest for an answer. For classification problems, each tree votes for a class and the most voted class will be our final answer. For regression problems, we

take the mean of the trees' outputs.

At a lower level, we need to introduce the concept of *bagging* or *bootstrap aggregation*. Let's assume we want to create a forest with N trees. Each of them is trained on a set of K randomly selected examples from the training set. This sampling is done uniformly and with replacement, so an example can be used twice to train the same tree.[8] So, we have just introduced 2 other hyperparameters: N the size of the forest and K the size of the training sample.

2.2.3 Hyperparameters

In this (sub)section we identified quite a lot of hyperparameters such as number of neurons in a layer or the number of trees in a forest. The architecture of a model is therefore very relevant for the purposes of this paper. One of the fundamental goals of true automated machine learning is to find model architectures automatically.

2.3 Learning

As the algorithms see training examples we can track how accurate a model is by considering an error function. Intuitively, this function measures how far are the results of the learned model from the actual results. So, our objective is to minimize the error function.

2.3.1 Neural networks

To minimize the error function, neural networks rely on a backpropagation algorithm which is, essentially, a gradient descent method that spans over the whole network of neurons.

Learning rate and momentum

The length of a step in this gradient descent mechanism is usually referred to as learning rate. If the search space was to be visualized as a 3D surface, the learning rate is the length of a step taken towards a minimum (local or global) on the "walls" of the surface. In the same context, momentum is the length of the gradient descent step through the "valley". In hyperparameter tuning, learning rate and momentum are extremely important because they are at the core of the learning process. A high learning rate makes the ML model to converge (in terms of accuracy) very fast after learning for a short time but later, it does not improve at all or even worse, diverges. If the learning rate is too small, the ML model will converge very slowly and eventually (after a very long time) it will reach the optimum. This is illustrated below:

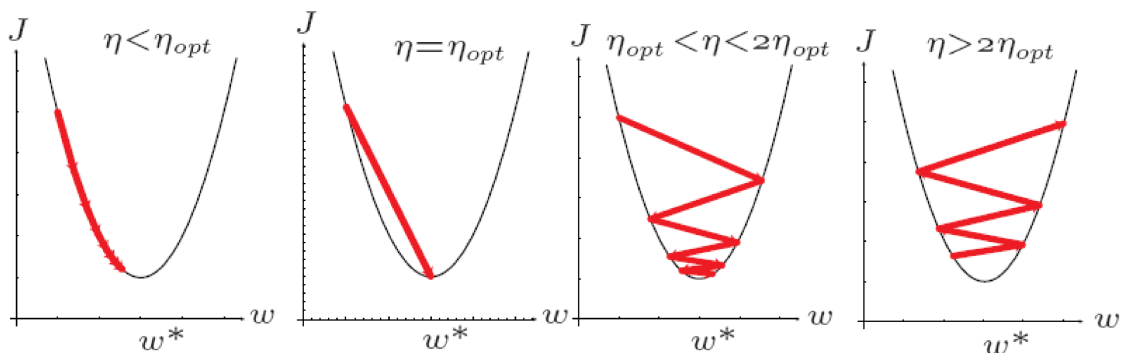


Image from Imperial College Machine Learning lecture notes [41]

Note that η is the learning rate and η_{opt} is the optimal learning rate.

Batches

There are many ways of feeding training examples to a learning network: updating weights after seeing training examples one by one, repeatedly updating weights after seeing all examples at once or updating weights after a random sample of S training examples has been seen. The most common approach is by far the last one which is called "mini-batch training". This creates another hyperparameter, S , the batch size.

2.3.2 Decision trees - Iterative Dichotomiser 3 (ID3)

In this case, we want to minimize the error in leaves. So, we build the tree as follows:

1. perform a statistical test on each attribute to determine how well it classifies the training examples when considered alone
2. set the attribute that performs best as root of the tree
3. to decide the descendant node down each branch of the root (parent node), sort the training examples according to the value related to the current branch and repeat the process described in steps 1 and 2.

[40] [Excerpt from Imperial DoC machine learning lectures]

The statistical test is a function called Information Gain which, in turn, is based on entropy. So, apparently, this algorithm does not require any hyperparameters. In practice, we might need one for information gain when dealing with incomplete datasets [24] but we can omit it since the datasets we will work on are complete.

2.3.3 Hyperparameters

The learning phase brings in a whole new bunch of hyperparameters, especially for neural networks. They are obviously crucial for the way models perform. Therefore, we should carefully observe them in our future experiments. For example, we can verify if they match the order of magnitude given by different heuristics (eg. momentum is about 0.9 or 0.95). Learning rates are known to cause behaviours that "confuse" several types of optimizers.

2.4 Avoiding overfitting

Overfitting happens when the machine learning model is very accurate on training data but may fail to predict well. Some people refer to this phenomenon as "high variance". Therefore, we should prevent our algorithms from ending up in this situation. Frequently the dataset can contain some noisy data and if we overfit the noise can be seriously misleading. Next, we will look at a few ideas that reduce overfitting.

2.4.1 Neural networks

L1/L2 Regularization

To prevent overfitting in a neural network we can loosen the connections between the neurons. That is, we can reduce or zero out some weights. Regularization is one way to achieve this by modifying the error function.

There are 2 common types of regularization L1 (because it uses the first norm of a vector) and L2 (uses the second norm of a vector). In case of regressions, we call Lasso Regression the models that use L1 and Ridge Regression the models that use L2. [26]

L1	L2
$E = E_0 + \lambda \cdot \sum_{w \in \text{all-weights}} w $	$E = E_0 + \frac{1}{2} \cdot \lambda \cdot \sum_{w \in \text{all-weights}} w^2$

Where:

- E_0 is the original error function (eg. quadratic loss, negative log-likelihood)
- λ is the regularization parameter
- E is the new error function

As the learning algorithm tries to minimize E we can see that: if λ is small, the regularization term becomes less important and the minimization will impact E_0 . Conversely, if λ is large the minimization will have more impact on the weights so large weights will be penalized.

L2 regularization is also called *weight decay*. Please note the $\frac{1}{2}$ coefficient because there are variations of L2 without it which do not typically account for weight decay [27].

Dropout

We have seen that regularization changes the error function in order to penalize some weights. A more straightforward approach is to modify the network. So, dropout method drops neurons randomly with a given probability while training. Similarly to regularization, neurons do not co-adapt too much so we reduce overfitting.

Early stopping

In case of neural networks, learning for too long (too many epochs) can lead to overfitting. So, early stopping is sometimes used. This method means to stop training the model before we reach the global minimum. [7] This can be achieved by stopping when the validation error no longer improves or when a maximum number of epochs is reached. Please note, that we make the assumption that the validation set error is non-increasing with the number of epochs.

2.4.2 Decision trees

Decision trees are very prone to overfitting since the tree grows until it perfectly classifies the training data. Obviously, this impacts random forests too. There are 2 typical methods to solve this:

Pre-pruning

Pre-pruning means discarding certain branches while building the tree. In a way, this method is the equivalent of early stopping for neural networks. Examples of when to stop growing a tree (in pre-pruning) are attaining a maximum depth and/or size. [6] It is important to note that this method introduces new hyperparameters. Also, pre-pruning may not be very effective. Knowing precisely when to stop is not a trivial task and it is usually based on heuristics. [25]

Post-pruning

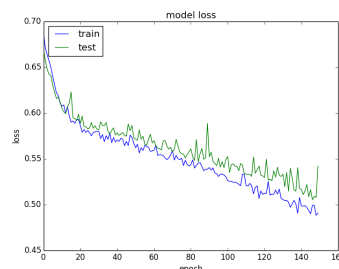
A good alternative is post-pruning: remove branches after the whole tree was built. To do so, we can collapse some sub-trees, each in its root. The problem of finding the best tree size/depth remains. But, in this case, we have more information about the dimensions of the trees so we can calculate the missing pieces (eg. maximal size of a tree) using a validation set. This means another hyperparameter to be tuned. [25]

2.4.3 Hyperparameters

To avoid/reduce overfitting the methods proposed above introduce some new hyperparameters that need to be tuned: regularization parameters, the probability of dropping, the maximal number of epochs, maximal size of a tree etc.

2.5 Loss functions

Further in the paper, most of the work will be around loss functions so we will explain how the hyperparameters described so far can be characterized. The longer we train a ML model the more accurate we expect it to become. This is exactly what loss functions depict:



Example loss functions [56]

Loss functions record the error between expected values and the output of the ML model if run in test mode. The more epochs the longer we train the ML model for. That is, we train the model for one more epoch and then we test how well it performs on the Validation set or on the Test set (or both). Of course, it is desirable that the loss functions are descending (i.e. the model becomes increasingly more accurate) and that they eventually plateau. The final error of a ML model is the error at the last epoch (i.e. rightmost point of the loss function).

Note that epochs can be determined by any resource for example time or number of seen training examples.

2.6 Summary

We have already seen several examples of hyperparameters and the context they come in. So, we can group them in terms of different criteria as follows:

Group by	Categories	Examples
Role	<i>Architectural</i>	number of layers, number of hidden neurons per layer, number of trees in a random forest
	<i>Training</i>	max epochs, learning rate, batch size, momentum, dropout probability
Domain of values	<i>Discrete</i>	number of layers, number of trees in a random forest, max epochs, batch size
	<i>Continuous</i>	learning rate, dropout probability
	<i>Categorical</i>	activation functions per layer (eg. ReLu, Softmax), type of regularization (L1 or L2)

Obviously, it is desirable that an automatic algorithm caters for all these types. Continuous, for example, is a more difficult case because we cannot simply enumerate all possibilities. Ranges for each hyperparameter are another aspect to take into account. Sometimes, ranges can be unknown. We may not know how big a neural network suffices for a given purpose. Heuristics usually give the ranges or the orders of magnitude for each parameter so this is one way we can evaluate results.

Chapter 3

Optimizers - methods background

Next, we will introduce the most common methods for hyperparameter tuning. We will see the theoretical background as well as pros and cons for each of them. When facing more difficult concepts there will also be a section to get the intuition first. That is, a basic introduction that is later modelled mathematically.

We start with the basic methods (manual search, grid search and random search). Next, we will look at methods based on Bayesian optimization and Gaussian Processes. Later, we will move on to genetic algorithms. Finally, reinforcement learning methods will complete the picture.

3.1 Basic methods

Obviously, the basic methods are quite easy to implement. However, for more complicated methods we can use/write a library. Therefore, the overhead in the worst case can be writing a one-off implementation. This is considered manageable. Please bear in mind that we are aiming to establish the state of the art not to simply set some parameters.

3.1.1 Manual search

Method description

Manual search means no automation at all. Put simply, for humans, it means to:

1. try different sets of hyperparameters
2. analyze the effects of the choice made
3. repeat 1 and 2 until a desired performance is achieved or after a certain time

Advantages

Manual search is somewhat informed for relatively simple models. We rely on human intelligence to understand hyperparameters and to manage the exploitation-exploration trade-off in the search space. For example, if a neural network with 10 neurons performs poorly. A human would understand that the next attempt should not be 11 or 12 because it is very unlikely that this modification would bring a significant improvement. So, a much greater number would be used instead (eg. 100). [29]

Disadvantages

Takes a lot of time that could be spent by researchers and engineers in more interesting ways. Also, does not scale well with the complexity of the model. High dimensional spaces notoriously pose problems to many humans.

Conclusion

Apart from the disadvantages presented above, let's remember the reasons presented in Introduction - Motivation (Chapter 1) of this paper. So, manual search can be dismissed from our list of options.

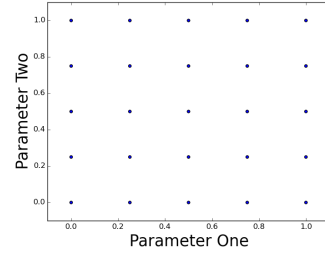
3.1.2 Grid search

Method description

Finds the best solution (set of parameters) by exploring all the possible combinations. It goes through every configuration of parameters and returns the one that yields the minimum of the cost function.

As a little analogy, grid search is the equivalent of backtracking from the world of algorithms. So, we expect that this method is very expensive and that it has a limited spectre of use.

Let's take a simple example where we have only 2 discrete hyperparameters. We could plot all the possible values they can take as a grid and hence the name of this method.



Grid search as lattices [47]

Advantages

Grid search guarantees the global optimum but with a high cost of computational power and resources. We might be a bit in difficulty when it comes to continuous hyperparameters. However, if we sample at a small enough "step" we can transform this continuous space into a discrete set and we will find a value which is close to the optimum. Note the smaller the step the closer to optimum.

Because all "runs" are independent (no dependencies on previous results) this approach can be run in parallel as a speed-up.

Disadvantages

Takes too long and quickly becomes impractical when more sophisticated models are being used. Basically, this method does not scale and, subsequently, can only be used for very small-sized examples. Hence, this is the main concern we are trying to address throughout this paper and its whole purpose. Also, if a continuous hyperparameter search is sampled from a periodic function at a step equal to the period we might never find the best value. [30]

Conclusion

Due to the prolonged duration, we cannot really use this method as a benchmark for the approaches that we will see next. They target complex models so grid search is simply intractable in those cases.

3.1.3 Random search

Method description

Instead of trying all possible combination of hyperparameters, we generate combinations randomly. Hence, we can stop after any number of attempts and pick the best set so far.

Advantages

Faster than grid search and, hence, tractable. Even though not very impressive as a method, this tractability can suffice for some applications. So random search has some cases where it can do a decent job. However, we always aim for better.

We should also note that the longer we are running random search for the more likely it is to obtain results closer to optimum. Hypothetically, this method could take as long as grid search. Obviously, if we want to be sure that we find the global optimum we need at most N trials where N is the number of trials for grid search.

In practice, in a shorter time, we hope to be lucky enough to get a good set of parameters. Realistically, it is unlikely to find the very best parameters earlier on.

Similarly to grid search, random search is parallelizable because all "runs" are independent.

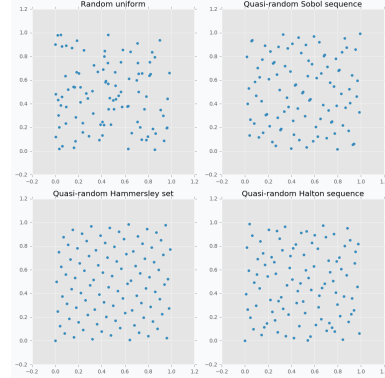
Disadvantages

Random search offers no guarantee of optimality on the result it returns.

This method is not informed of previous results. By running a few trials we can start to infer how good/bad we are doing but neither random search nor grid search incorporate this kind of information.

Another potential downfall is the way we generate the random values. That is, depending on what distribution we are sampling from (most commonly uniform) we may not explore chunks of our search space. Sometimes this problem can be solved by generating more spread-out values, so quasi-randomly. This is known as *low-discrepancy sampling/sequence*. A few examples of such sampling strategies are: [29]

- Poisson disk
- Sobol sequence
- Hammersley set
- Halton sequence



Quasi-random generations [48]

Poisson disk, for example, guarantees that the distance between any two quasi-randomly generated points is greater than a given r . The algorithm also needs a constant k . [10] This brings in new hyperparameters which is not desirable in this situation. Also, some of the sampling strategies (eg. Halton, Hammersley) do not scale well with high dimensions. [9]

Conclusion

This approach is definitely something we can benchmark against. Obviously, on average, our future proposed models should perform better than random search.

3.2 Bayesian Methods

Bayesian methods are very popular in literature as well as in practice. There are a few well-known implementations based on this principle: Spearmint, SMAC and TPE. We will see all of them but first we must understand the fundamentals. This class of methods is built on Bayesian statistics and Gaussian processes which will be explained in detail.

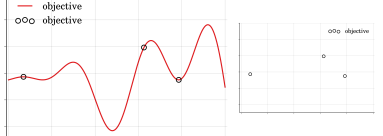
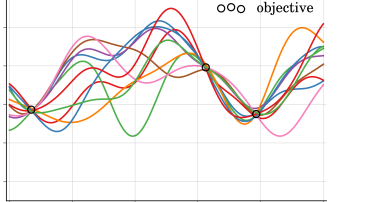
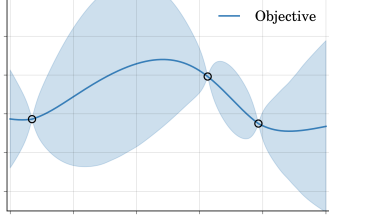
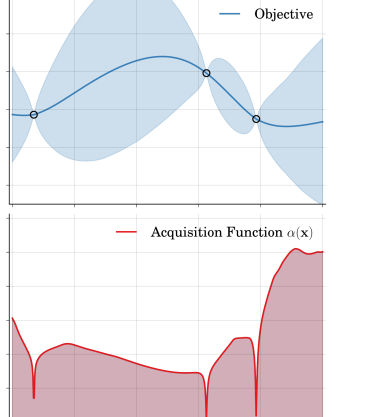
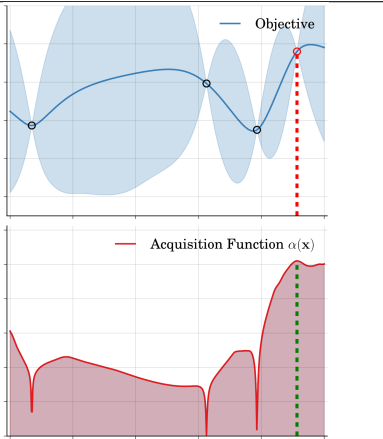
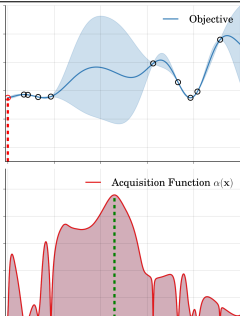
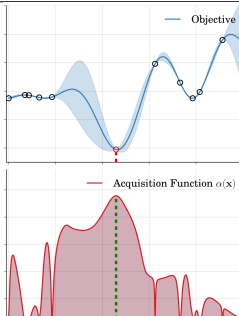
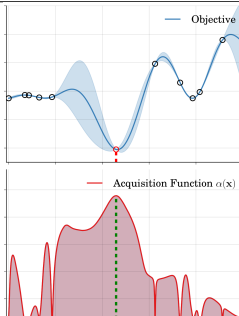
3.2.1 Method description

Informally (Intuition)

First of all, we need to understand our task: given a machine learning model we must find the optimal values for its corresponding hyperparameters. We know that after deciding upon a set of hyperparameters we can run the model to find its error (as measured on the test set). So, the goal is to find the hyperparameters that yield the minimum error. One issue is that running the model is expensive time-wise. Obviously, it is preferable not to run it too many times.

Equivalently, the model can be seen as a function $f: \text{Hyperparameters} \rightarrow \text{Error}$. We will call this the *objective function*. The goal is to minimize f , evaluating it as few times as possible (because it is expensive to compute). The domain is comprised of sets/vectors of hyperparameters ($\text{Hyperparameters} \subseteq \mathbb{R}^n$) but for simplicity, we assume only one hyperparameter (only in the example $\text{Hyperparameters} \subseteq \mathbb{R}$).

This is a sequential method: we choose a point that looks most promising to minimize the error (f), we evaluate f there, we see if we found a point closer to the minimum and then we repeat the process based on the information previously computed. How we determine what is the most promising point is the melting pot of this method. We will create another function called the *acquisition function*, which is cheap to compute and to optimize. Let's assume we have already evaluated f at 3 points (as an initialization step). The breakdown of the next iteration is:

1	<p>The objective function f (plotted in red) is still unknown. But we have 3 points where we already evaluated f: $(x_1, f(x_1)), (x_2, f(x_2)), (x_3, f(x_3))$</p>	
2	<p>f definitely passes through these points. If we were to guess f we might wonder what are the potential <i>smooth</i> functions that can pass through these points. There are infinitely many possibilities. If we would draw a few of them:</p>	
3	<p>Based on these points we can set some expectations where the other points of f can lie. This is a <i>regression</i>: we compute a predicted function. But we can also determine by how much the points of f (which are unknown to us) can deviate from this predicted function (with a confidence interval). We can plot this deviation space as a "cloud" (in light blue).</p>	
4	<p>We assume f is smooth. That is, we assume a Gaussian distribution. In layman's terms: x_1 and x_2 are spread out so the uncertainty about the values of f in between them is higher. Conversely, there is little uncertainty about where values of f can lie for points between x_2 and x_3. Remember, we want to evaluate f at a promising point. The <i>acquisition function</i>, depicted here in red, measures the potential of such points. The x that maximizes the acquisition function is the most promising point (of being a small value for f).</p>	
5	<p>Unfortunately evaluating f at the x that maximized the acquisition function does not give a minimum. But, we have more information now. So we can REPEAT the process (pick the most promising point, evaluate f at it, recalculate the cloud, recalculate the acquisition function) until we find the global minimum. Note that the acquisition function efficiently manages the <i>exploration-exploitation</i> trade-off. That is, going to unseen places versus insisting on a certain area.</p>	
...
n	<p>Eventually</p>  <p>will find f's minimum:</p> 	

All images are sourced from [49].

Obviously this belongs to Bayesian statistics because we make decisions based on prior knowledge that keeps being updated. Also note, that there are multiple choices for acquisition functions (eg. Probability of Improvement, Expected Improvement, GP Upper Confidence Bound). [11]

Formally

We have seen that we want to perform a "special" regression: not only the predicted function is important but also how certain we are about it. This is, what a *Gaussian Process* basically does. Mathematically, we find a probability distribution over functions. Let's review the basics of Gaussian distributions first:

From vectors to the Gaussian they were sampled from For the sake of simplicity, we will assume only 2 dimensions. So any point in this space is described by a vector with 2 elements (\mathbb{R}^2). From a statistical point of view, every point corresponds to the realizations of 2 random variables, hence the vectors are called random vectors.

Given some points in a plane (i.e. a cloud of points) our goal is to determine what Gaussian distribution they were sampled from: $X \sim \mathcal{N}(\mu, \Sigma)$ where X is a random vector (point). There are 2 measures that can characterize this cloud of points:

- μ is the center of the cloud (or the average point). It is a vector since it is a point in our space.
- Σ is the Covariance matrix which describes how the points in the cloud are related to each other.

This is called the *multivariate normal distribution*. Generalizing to functions is straightforward since a function can be approximated by a sequence of vectors/points.

Gaussian processes Now, we have all ingredients to define a Gaussian process (GP). The multivariate normal distribution yields a distribution from which the points that were evaluated (eg. x_1, x_2, x_3 in our example) are sampled from. We assume that the rest of the points are also sampled from the same distribution. So, each point is described by Gaussian (a mean and a variance). Hence, the result is a distribution over approximated functions. Note that the points are n-dimensional.

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')) \text{ where:}$$

- m is the mean function $m(x) = \mathbb{E}[f(x)]$, what we saw as the predicted function. It is a sequence of means of Gaussians for each point.
- k is the covariance function $k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]$. It is also called *Kernel function*. k is a measure of similarity between x and x' . That is, if x and x' are close then so should $f(x)$ and $f(x')$. [43] There are multiple choices for this function.

Acquisition functions This is a probabilistic proxy model which is much cheaper to compute than the objective function. As we have seen in the examples it handles exploration and exploitation:

- *exploration*: mathematically, we prioritize areas with high variance. That is, where the light blue cloud (in the examples) is low (small Y). Remember the light blue cloud means that the objective function can have points in that area. If the cloud is low there is a potential of having a point that minimizes the objective function in that area.
- *exploitation*: areas with low mean. That is, the means make up the predicted function. If the predicted function is low in a certain area, it may be worth evaluating the objective function there.

Acquisition functions are not unique either. The most popular choice is *Expected Improvement* which maximizes the expected improvement over the current best: $\alpha(x) = \mathbb{E}[\max(f(x_{best}) - f(x), 0)]$. [11, 1]

SMBOs

Bayesian methods are also known as Sequential Model-Based Optimization methods (SMBOs). Essentially, they are a generalization of the method based on Gaussian Processes presented above. SMBOs share the same features:

1. use a surrogate model (cheap to compute) of the true objective function (expensive to compute). It can be expressed generally as (note p is a probability):

$$p(\text{value of the objective function} \mid \text{set of hyperparameters}) \text{ or, in short: } p(y|x)$$

2. optimize the surrogate (or a transformation of the surrogate)
3. run the objective function with the set of hyperparameters that optimized the surrogate to find out its true value
4. update the surrogate model with the newly found results applying Bayesian reasoning
5. repeat steps 2-4 for a given time or for a given number of iterations [31, 12].

The whole algorithm in a more concise view is:

SMBO(f, M_0, T, S)

```

1   $\mathcal{H} \leftarrow \emptyset$ ,
2  For  $t \leftarrow 1$  to  $T$ ,
3       $x^* \leftarrow \operatorname{argmin}_x S(x, M_{t-1})$ ,
4      Evaluate  $f(x^*)$ ,  $\triangleright$  Expensive step
5       $\mathcal{H} \leftarrow \mathcal{H} \cup (x^*, f(x^*))$ ,
6      Fit a new model  $M_t$  to  $\mathcal{H}$ .
7  return  $\mathcal{H}$ 
```

where:

- \mathcal{H} - evaluations of the true objective function, of type list of pairs $(x, f(x))$
- M_t - the surrogate model at iteration/time t
- x^* - set of hyperparameters that maximize the acquisition function S

Pseudocode of a general SMBO [12]

3.2.2 Advantages

Let's sum up the numerous pros of this method. Bayesian methods are:

- *informed*: automatically decide which set of hyperparameters to try next based on the results of the combinations tried previously.
- *non-parametric*: at the beginning we flagged that some automatic parameter tuning algorithms might have their own hyperparameters which need to be tuned. This is not the case here. Moreover, this makes Bayesian methods very powerful. Imagine we find another automatic method that is parametric. We can use a Bayesian method to optimize its hyperparameters.
- *scalable*: the whole idea is designed in n -dimensions (for any n) so high-dimensional search spaces are easily handled (unlike random search).
- *performant*: experiments showed that Bayesian methods perform well on various datasets. This means much better than random search and in fewer steps.
- *good with non-convex*: as we have seen in the example from "Informally (Intuition)" non-convex functions are effectively optimized.
- *not slow*: by design we do not train the machine learning model too many times.

3.2.3 Disadvantages

It is good that we use the previously computed results in a meaningful way. But this comes with a catch. The fact that the method is sequential makes it hard to parallelize. Though, some researchers managed to parallelize it to some extent. Another issue is that Bayesian methods are sensitive to the kernel function choice[1]. However, there are common kernel functions that usually perform well. For example, variations of $k(x, x') = \exp(-\frac{1}{2}(x - x')^2)$ that can have extra parameters but they do not need to be tuned[44]. The purest form can only optimize numerical hyperparameters. However, there is a popular workaround (SMAC) which addresses this issue.

3.2.4 Variations

By variations, we refer collectively to papers and paper-based implementations. They all share the same SMBO principles but have a few particularities which will be detailed next. For each variation, the state of the art, the conditions in which it was obtained, the datasets they were performed on and further constraints/comments are presented in the "Comparison - heat map". First, let's focus on the theoretical basis of every variation.

Spearmint

Spearmint is the implementation of the paper "Practical Bayesian Optimization of Machine Learning Algorithms" [11]. It is implemented based on Gaussian Processes so most of the theoretical background applies to Spearmint. It has a few particularities though.

The paper proposes, analyses empirically and compares 4 algorithms, all of which use Expected Improvement as the acquisition function. These algorithms are referred to as:

1. *GP EI MCMC*: marginalizes GP hyperparameters (so to compute the integral of the acquisition function).
2. *GP EI Opt*: optimizes hyperparameters.
3. *GP EI per Second*: EI per second combines accuracy aggressiveness with time aggressiveness. That is, the objective function is doubled by a duration function which records how long a set of hyperparameters takes to evaluate. Hence, the optimization of the surrogate is made in terms of both functions (objective/loss and duration).
4. *Nx GP EI MCMC*: N times parallelized GP EI MCMC is a theoretical improvement of the algorithm to make it parallelizable. Because the acquisition function is cannot be re-used, the algorithm computes its Monte Carlo estimates "under different possible results from pending function evaluations". [11].

TPE

TPE stands for Tree-structured Parzen Estimator. First of all, we need to understand what is the "tree structure". TPEs work with tree-structured search spaces. For example, the number of hidden units in the 2nd layer of a neural network are only well-defined after a number of layers has been decided upon [12].

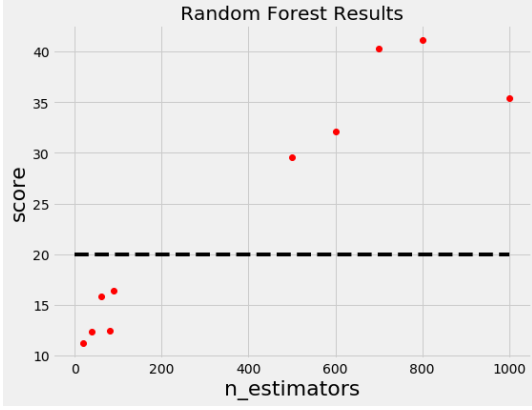
Remember that SMBOs typically aim to model $p(y|x)$. TPE does this by modelling $p(x|y)$ and finding $p(y|x)$ with Bayes rule. That is:

$$p(y|x) = \frac{p(x|y) \cdot p(y)}{p(x)}$$

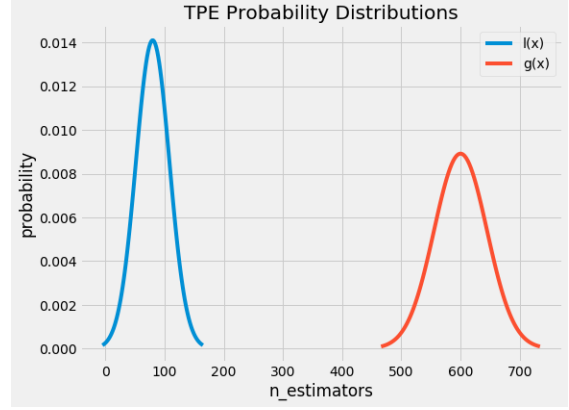
where x is a set of hyperparameters and y is its evaluation on the true objective function. The most important part of this method is computing $p(x|y)$ in terms of 2 other distributions l (lower) and g (greater):

$$p(x|y) = \begin{cases} l(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases}$$

where y^* is a threshold. It is (re-)calculated as a quantile, γ , of the observed y values (of the previous evaluations of the objective function) [12]. For the next steps, let's take a simple example, where we only have to estimate 1 hyperparameter: the number of estimators in a random forest. The score axis, represent the error on the validation set (values after evaluation the true objective function).



y^* threshold, graph from [31, 51]



l, g distributions, graph from [31, 52]

The red dots are previous evaluations of the objective function and the dotted line represent y^* , the threshold. The "lower" and "greater" (l and g) distributions are depicted on the right. At this point, we need to optimize the surrogate to find out where to evaluate the objective function next. As usual, this is where acquisition functions and the exploration - exploitation trade-off kicks in. TPE uses Expected Improvement and it turns out that it is proportional to $\frac{l(x)}{g(x)}$ [31]. This leads to the fact that the "lower" area looks more promising. So, it matches the intuition. [31]. The authors of [12] compare TPE with Gaussian Processes (GPs) and outline that GPs behave more aggressively than TPE. On one hand, GPs go for very low y^* - frequently less than the lowest point so far. On the other hand, TPE opts for a y^* larger than the lowest point [12].

SMAC

SMAC is another variation based on SMBOs which, for the first time, allowed "categorical parameters and optimization for sets of instances" [22]. [22] showcases experiments on problems like SAT (propositional satisfiability problem) which prove that categorical parameters are supported. SMAC uses random forests to model $p(y|x)$ as a Gaussian distribution (Hutter et al., 2011) [22, 1] so, by design, it more parallelizable than other Bayesian optimizers.

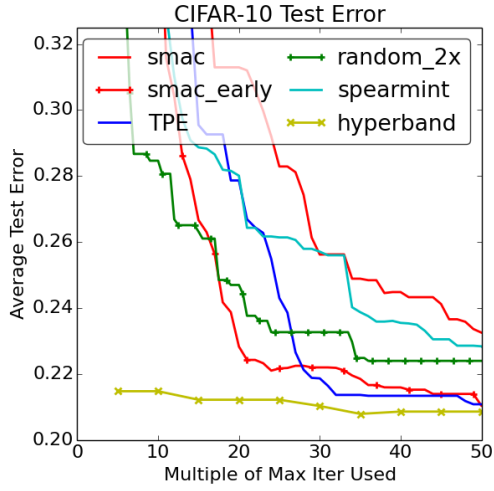
SMAC early stopping Normal SMAC suffers from running the machine learning models with unpromising sets of hyperparameters until the end. So, SMAC early stopping is a method proposed in [15] which combines SMAC with a predictive termination algorithm based on learning curves.

SBO

Scalable Bayesian Optimization (SBO) is another SMBO that, on one hand, reduces the time complexity of Spearmint from $O(n^3)$ to $O(n)$, but on the other hand, trades off flexibility of the architecture of the underlying ML model [1, 13].

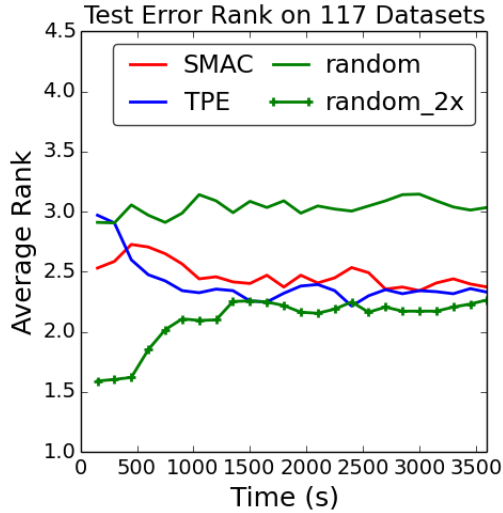
Conclusions

Performance-wise: Some interesting comparisons between these SMBOs are exposed in a work about Hyperband, a technique which aims to improve the speed of this kind of algorithms [38]. Another benchmark is introduced in these studies is "Random 2X". This method is nothing else but random search run for twice as long as the rest (random search, SMAC, TPE, Spearmint) w.r.t. time or number of iterations. For now, please ignore hyperband because it will be covered in detail later.



Average errors on test sets on CIFAR-10, graph from [38, 53]

Consequently, the same study analyzes the relative performance of SMAC, TPE, Random and Random 2X across 117 datasets. The left plot (below) shows the average error so the lower the better. The authors motivate the absence of Spearmint with "complications with conditional hyperparameters" [38]. Let's remember that we wanted to test if the hierarchy between the methods is preserved. Now we can see Random 2X performs better than TPE and SMAC. So this hypothesis is rejected.



Average test error per method over 117 datasets, graph from [38, 54]

However, we can still verify one last hypothesis: *there is no clear hierarchy between the methods, they are all comparable and they perform better/worse on a case by case basis*. Averaging across *all* datasets reduces the granularity of the result. A sensible thing to do in this case is to apply the same method on sets of up to 12 datasets which were randomly sampled from the initial 117. For example, we select arbitrarily 12 datasets, we tune hyperparameters with Random Search, SMAC and TPE on each dataset and plot the average error (on the test set) across the 12 datasets for each method. Just like we did for the 117 [38].

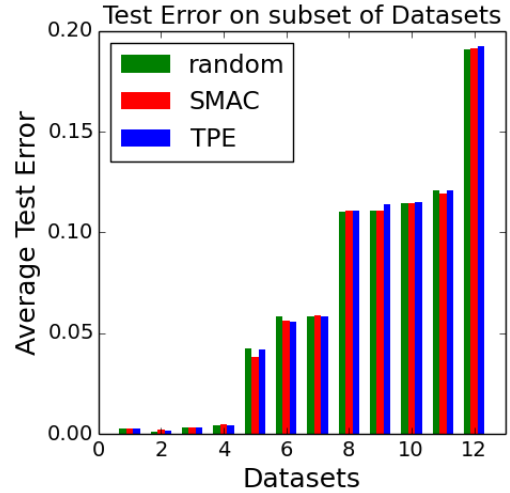
So, the histogram (on the right) concludes that the hierarchy of methods almost always changes while the results are quite close across methods. Hence, we retain the last hypothesis we made.

Parallelization-wise: SMAC has a clear advantage because it is based on random forests which are inherently parallel. Also, Spearmint introduced a strategy to address this issue. However, we feel that work can still be done with respect to parallelization.

A typical learning task which is sensitive enough to parameter tuning is classification **on CIFAR-10** (a dataset of images). In the adjacent plot lower is better because it corresponds to lower errors. Therefore, we can observe some hierarchy between the proposed methods:

1. TPE
2. SMAC early stopping
3. Random 2X (random search for twice as long as the rest)
4. Spearmint
5. SMAC

However, we need to experiment with several datasets to postulate that this hierarchy is preserved.



Average test error per method over up to 12 datasets, graph from [38, 55]

3.3 Genetic Algorithms

As before, the theoretical background is presented first, followed by outlining advantages versus disadvantages. Finally, we will investigate particular implementations that are popular in literature.

3.3.1 Method description

Genetic algorithms (GAs) are methods that solve optimization problems where very large search spaces are required. Basically, GAs find the optimal solution when checking every possibility (backtracking) is not tractable. This is something that looks exactly like what we need. Here, we will start with an application of GAs to get some intuition and after that, we will see the steps for a general GA.

Specifically for hyperparameter tuning

For our use case, GAs can be simply regarded as an informed version of random search. As an overview:

1. Generate some random sets of hyperparameters.
2. Run the machine learning model for each set of hyperparameters and *wait for the errors on the validation set*.
3. Create new sets of hyperparameters by *mixing* hyperparameters from sets that just *performed best*.
4. Repeat steps 2, 3, 4 until: a certain time passed or a given accuracy is achieved.

There are quite a few variations that apply to hyperparameter tuning. Next, we introduce what GAs are and how they work in general and later, we will explore more particularities.

Generally

GAs are based on the Darwinian concept of evolution. In order to happen, evolution needs:

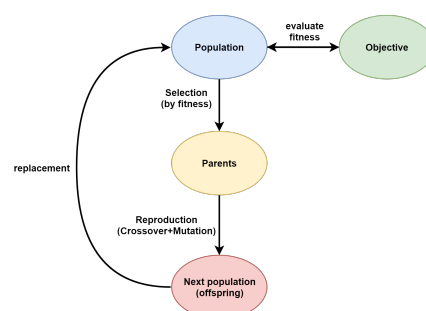
- *heredity*: children receive genes (properties) from their parents
- *variation*: individuals in a population must not be all identical (otherwise the offspring would be the same as the parents)
- *selection*: the best performing individuals (w.r.t. a task) are more likely to have children. Also called "survival of the fittest".

These conditions suffice to produce increasingly better generations with respect to some task. That is, evolution! [14] A computer simulates this same process.

Given any optimization problem, we will repetitively make guesses, check how good they were and make a better guess based on the observations. Guesses are made by a population (of individuals), one each. Every individual requires some representation of an instance of the problem (*encoding of genes*)

Selection: A *fitness function* is used to evaluate each guess/individual - the higher the value of the function the more suitable the guess was. In order to create a new generation, we must define and follow a rule that will make the fittest individuals parents of the new generation. Note that the number of parents can be any natural number (1, 2, ...) - obviously, it cannot be large.

Heredity: A process known in biology as *crossover* is used to create the offspring of any parents. The exact way crossover works depends on the encoding of the genes.



Modified diagram based on [50].

Variation: After crossover, the offspring is made only of genes from its parents. However, to ensure variation in the next generation, some genes (parts of the encoding) can be randomly modified. This phenomenon is called *mutation*. The probability of a gene being mutated is specified at the beginning.

At this stage, the new generation is ready to *replace* the old one. Note that the first generation is created randomly and the population size remains the same across generations.

So, the hyperparameters of GAs are the population size and the mutation rate. On top of that, the strategies for gene encoding and crossover can also influence the result of a GA. For our use-case, exploration is ensured by mutation and crossover while exploration by survival of the fittest.

3.3.2 Advantages

GAs are performing very well on a number of datasets. Very parallelizable, for each generation. Very performant, on some datasets GAs can beat Bayesian methods. GAs can end up with multiple solutions. Let's assume we are computing the last generation. In this generation, there is (at least) a set of hyperparameters that satisfies our stopping criteria (eg. accuracy \geq X%). However, there can be another individual (i.e. another set of hyperparameters) that yields the same accuracy and hence the conclusion.

3.3.3 Disadvantages

Requires many error evaluations (runs of the machine learning model to find the error on the validation set): $sizeofpopulation \cdot numberofgenerations$ exactly where both terms can be of the order of hundreds or, sometimes, even thousands. This leads to prolonged durations and requires large computational resources. In contrast, Bayesian methods require far less runs. Luckily, we have seen that GAs are parallelizable. In addition, GAs are sensitive to population size (i.e. small populations may only converge to a local optimum rather than the global one)[1].

Also, GAs are parametric. In turn, they need some hyperparameters that need to be optimized. GAs are, in fact, quite sensitive to the choice of the population size.

3.4 Reinforcement learning

Reinforcement learning-based optimizer is a remarkable class of optimizers since it gives the state-of-the-art performance in the field. Let us see why the nature of the problem of tuning hyperparameters fits perfectly the reinforcement learning philosophy.

3.4.1 Method description

In reinforcement learning, the algorithm learns simply from trial and error without being given any extra information [1]. Of course, the more trials the lower the error. By contrast with a classification algorithm in which the classes of each example in the dataset are known and fed into the algorithm, reinforcement learning does not receive any such external help.

Formally, reinforcement learning is defined as a cycle. An agent in some state of an environment takes an informed action based on the rewards it received for the previous actions, the environment "reacts" and gives back to the agent a reward for the action it has just taken. Finally, the agent updates its history of actions and rewards. Next, the cycle continues. This description is similar to what we have seen in Bayesian methods but it is important to note that, in addition, Bayesian methods aim to find the lowest error in as few steps as possible while in reinforcement learning there is no such constraint. Relaxing this constraint is expected to provide better results since there is more time for both exploration and exploitation. The trade-off is about the waiting time.

3.4.2 Advantages

At the moment, reinforcement learning methods gave the state of the art in terms of automatic hyperparameter tuning. They generally outperform other methods. Another benefit of reinforcement learning optimizers is that they are informed.

3.4.3 Disadvantages

Reinforcement learning’s main disadvantage is that it is very expensive in terms of computing resources which make these methods inaccessible for many users. For instance, the current state of the art required 800 GPUs so only large technology companies can afford to run such experiments. Also, because the underlying models are neural networks they tend to be parametric as well.

3.5 Hyperband

Hyperband is a very interesting method, because it is an optimizer per se but can also be combined with other optimizers to optimize their resource allocation. As a standalone method, Hyperband is a form of random search with a clever early stopping mechanism.

3.5.1 Method description

Mathematically, Hyperband is a bandit-based method.

Multi-armed bandit problem

Multi-armed bandit problem is a theoretical abstraction that, in practice, targets the optimization of resources with respect to some budget limit. At the beginning, there is little/no knowledge about how possible allocations will perform. That is, the allocation is made on the fly and can be changed based on previous observations.

One-armed bandit is an old(er) name for slot machines that are operated through pulling one arm/lever. Multi-armed bandits are a generalization where each arm of a slot machine draws the reward/loss from a different probability distribution. One’s objective is to maximize his/her expected gains given that he/she has no previous knowledge about the distributions behind the slot machines.

For this reason, let us refer to a combination/set of hyperparameters as an arm.

Assumptions

Hyperband makes two simple and reasonable assumptions:

1. **Budget:** The amount of resources (eg. time, computing power) is fixed/capped which is called "budget".
2. **Delta:** It is likely that sets of hyperparameters that perform differently at the beginning will preserve this delta/difference by the time training completes. This is easier to understand as: it is unlikely that a set of hyperparameters that performs poorly at the beginning of training phase will eventually become the best.

Of course, the usage of the resource budget should be maximized because there is no reason to run under maximum capacity.

Halving mechanism

Starting from the assumptions stated above, it makes sense to use a successive halving algorithm: start with a large population of arms and progressively cut those poorly performing arms of hyperparameters earlier on while carrying on only with the most promising ones. By regularly discarding configurations that do not look promising, successive halving frees the resources that they used to take and reallocates them to the more promising arms. This way successive halving agrees with the formulated assumptions: it maintains a fixed budget of resources, it maximizes their usage and by halving we make use of the delta assumption [13].

Hedging mechanism

There are well-known examples of behaviours that do not fully respect the delta assumption. For example, a ML model with a high learning rate will perform well at the beginning but it would stop improving soon. On the other hand, a small learning rate will perform poorly earlier but will eventually become better than the high learning rate. In terms of loss functions, a high learning rate means that the function descends steeply at the beginning and plateaus early, while the small learning rate is decreasing slowly but steady. So, the problem with the halving mechanism is that it is hard to fix the frequency of halving [38]. Therefore, Hyperband proposes a hedging loop. To maintain the fixed budget, Hyperband transitions from high populations of arms which are halved frequently to low populations of arms which are halved rarely. Each iteration of this hedging loop is called a "bracket" [13].

Hyperband algorithm

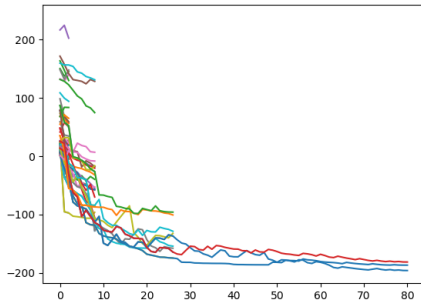
Hyperband is fully based on 2 hyperparameters:

1. R - the maximum amount of resources that any arm can use and
2. η - the halving/downsampling rate - controls the proportion of arms discarded in each round

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

Hyperband for $R = 81$ and $\eta = 3$, table based on [38]

The vertical columns are the "brackets" of the hedging loop. So, for example in the first bracket, Hyperband starts with 81 arms and runs them for 1 resource/epoch. After this, it keeps the top 27 out of the initial 81 (discarding the rest) and trains them for other 2 resources/epochs and the halving keeps going on. One can see that, to the left we half large populations frequently while, to the right we half smaller populations rarely. This caters for the cases like the example shown with small/large learning rate.



Loss function as seen after Hyperband has finished running on a ML model. The longest functions are the survivors. The final step of Hyperband is to return the best survivor (i.e. the one whose loss function has the lowest final error). Remember that every loss function corresponds to a different arm.

A good aggregated algorithm is presented in [17]:

Algorithm 1 Hyperband algorithm

input: maximum amount of resource that can be allocated to a single hyperparameter configuration R , and proportion controller η

output: one hyperparameter configuration

```
1: initialization:  $s_{max} = \lfloor \log_{\eta}(R) \rfloor$ ,  $B = (s_{max} + 1)R$ 
2: for  $s \in \{s_{max}, s_{max} - 1, \dots, 0\}$  do
3:    $n = \left\lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \right\rceil$ ,  $r = R\eta^{-s}$ 
4:    $X = \text{get\_hyperparameter\_configuration}(n)$ 
5:   for  $i \in 0, \dots, s$  do
6:      $n_i = \lfloor n\eta^{-i} \rfloor$ 
7:      $r_i = r\eta^i$ 
8:      $F = \{ \text{run\_then\_return\_obj\_val}(x, r_i) : x \in X \}$ 
9:      $X = \text{top\_k}(X, F, \lfloor n_i/\eta \rfloor)$ 
return configuration with the best objective function value
```

3.5.2 Advantages

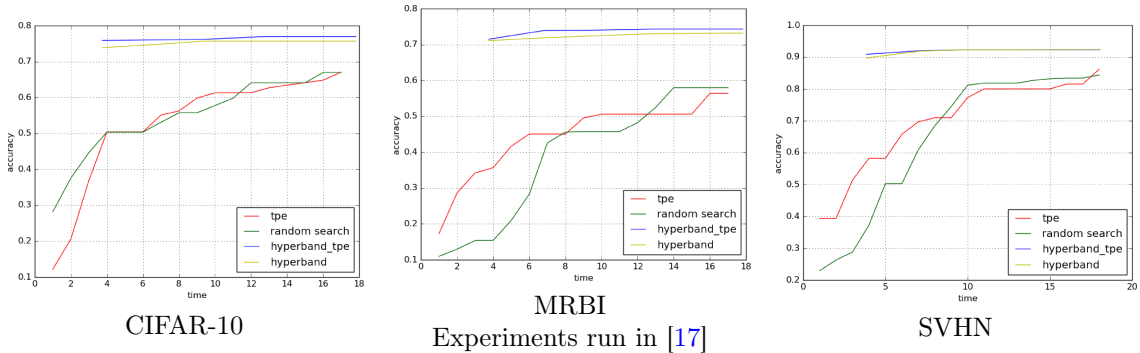
Performs well, better than TPE, SMAC, SMAC early stopping. It is easy to parallelize so it scales better with resources. Makes minimal assumptions, in fact, just one: that the relative delta/difference between 2 runs of the same machine learning model is constant. Conversely, other early stopping methods make more sophisticated assumptions about the model. Hyperband is "5× to 30× faster than popular Bayesian optimization algorithms on a variety of deep-learning and kernel-based learning problems" [13]. Even though it is parametric, R which determines the budget of resources is something that can be calculated and it corresponds to the desired time which one is willing to wait for the optimizer or the computational resources available. This parameter does not need to be tuned. Also, η the halving rate has been researched in [13] and its optimum value is 3. Hence, Hyperband is virtually non-parametric.

3.5.3 Disadvantages

It is not too informed: the only thing Hyperband cares about is the running relative performance of different sets of hyperparameters.

3.5.4 Variation: Hyperband+TPE

To address the concerns that Hyperband is not informed enough [17] firstly and [21] secondly proposed Hybrids between Hybrid and TPE which are 2 best-in-class optimizers. Both of them reported slight improvements from Hyperband.



However, we will show later that the design of the optimizer presented in [17] is wrong because it breaks the budget assumption. Neither [21, 17] give any reasoning why their certain architectures of hybrids have been chosen since there are several possibilities.

3.6 Comparison - heat map

		Scalable	Informed	Parallel	Few evaluations	Non Parametric	Performance		Notes	Papers/References	Comments
							CIFAR-10	MNIST			
Basic Methods	Manual Search						82.00%			Jamieson et al 2018	
	Grid Search										
	Random Search						~90%		After >150 hours		
Bayesian Methods	Spearmint						85.02%		3 layers	Domhan et al 2015 Jamieson et al 2018	All these SMBOs are compared in detail at the end of Bayesian methods section. There is no clear hierarchy as showed in: Jamieson et al 2018
	TPE						81.88%		1 layer		
	SMAC						82.53%		1 layer		
	SMAC early learning curves						82.80%		1 layer		
	SBO						91.19%		1 - 3 layers		
Hyperband	pure						93.63%		4 layers	Snoek et al 2015	Rigid architecture
	Hyperb.+TPE						~85.00%		3 layers	Jamieson et al 2018	Better than SMBOs alone
Genetic Algorithms	Large-Scale Evolution of Image Classifiers						94.60%		5.4M params	Real et al 2017	
							95.60%		40M params		
	Multi-node Evolutionary NNs									Young et al 2015	
	GAs for Committees of CNNs							99.76%		Bochinski et al 2017	Very close to the state of the art on MNIST but no reports on CIFAR-10.
	Cartesian Genetic Programming						94.02%		14 days 2 GPUs	Suganuma et al 2017	
Reinforcement Learning	Meta QNN									Real et al 2017 Baker et al 2017 Zoph et al 2016	
							90.76%		1.1M params		
							93.08%		8-10 days 10 GPUs 11.2M params		
	RNNs							99.68%			
							94.00%		20 layers 2.5 M params		
							96.30%		39 layers 37.4M param 800 GPUs		Top 3 state of the art Absolute best 99%

Extent: low/no  high/yes

Figure 3.1: Heat map - method comparison

3.7 Conclusion

After seeing how several methods work, pros and cons for each and a vast comparison of their results, we can start planning our future experiments.

Reduced computing resources (few GPUs) and time constraints will restrict us to involve only Bayesian Methods (especially TPE) and Hyperband. This is not bad news though. Bayesian methods are very popular (commercially and open source), fast and efficient in practice. But they still present areas that can be improved like parallelization or early stopping for instance. We saw some attempts to directly address these issues, especially SMAC early stopping. However, pure Hyperband is proved to consistently behave better than any Bayesian method.

The first few steps of combining Hyperband and TPE were already made in [16, 17]. It seems that this technique is slightly better than pure Hyperband because it combines the informed decisions of TPE with Hyperband's early stopping and parallel capabilities. However, only one way of combining the 2 was researched so far. The **existing hybrid optimizer is not fully informed**: it does not use all previous evaluations to make an informed decision. Also the hybrid published in [17] is flawed because it **breaks one of Hyperband's fundamental assumptions**. Furthermore, there are no exact numbers reported on CIFAR-10 dataset.

In conclusion, the intent of this paper is to further the study of hybrids between Hyperband and Bayesian methods (TPE being best-in-class) aiming to find the most efficient solution.

Chapter 4

Experimental setup

For the purposes of this paper we will look only at *supervised* machine learning. Therefore, we will focus on classification problems. We have seen earlier that the choice of the dataset impacts the performance hierarchy of the methods. For the time being, we must select a few datasets on which we can observe the differences between methods. That is, high accuracy on a dataset should not be attained after a few attempts.

From the method comparison heatmap [3.1](#) it is easy to see that the following datasets (MNIST and CIFAR) are quite centric in hyperparameter tuning literature.

4.1 MNIST

MNIST is a database with pictures of handwritten digits. The learning task associated to MNIST is to recognize a given handwritten digit. The training set consists of 60000 examples while the test set has 10000 [\[32, 35\]](#).

MNIST also appears in a few papers about automatic hyperparameter tuning. Despite the fact that this dataset is quite popular for classification tasks, we believe that it is *not challenging enough* for our problem to be our only benchmark. As demonstrated by [\[1\]](#) with an experiment based on Random Experiment Efficiency Curves, on average, it takes 4 trials of random search on a CNN to achieve an accuracy of 90% on MNIST [\[1\]](#). This is why, we will use a simpler model - Logistic Regression - to make the dataset harder to classify.

State of the art performance

MNIST has very low error rates (many below 1%). The current state of the art is accuracy of 99.79%, equivalently the error is 0.21% [\[33\]](#). At the time of writing this paper, this state of the art error rate is not reported on the MNIST website [\[32\]](#) but it can be found in LeCun et al [\[18\]](#) which proposes a generalization of Dropout. However, it does not mention the strategy employed to tune the hyperparameters.

Variations

There are a few attempts to make MNIST learning task (recognizing digits) harder to achieve. Sometimes these datasets are used for benchmarking instead of MNIST. The most notable datasets are:

- **MRBI**: rotated MNIST digits with background images.
- **SVHN**: street view house numbers.

4.2 CIFAR-10/CIFAR-100

CIFAR-10 is a database with 32x32 colour images which depict objects that belong to 10 categories (eg. airplanes, automobiles) some of each being quite similar (eg. animals: deer, horse). This requires more complex models for accurate recognition when compared to MNIST. So, fewer sets

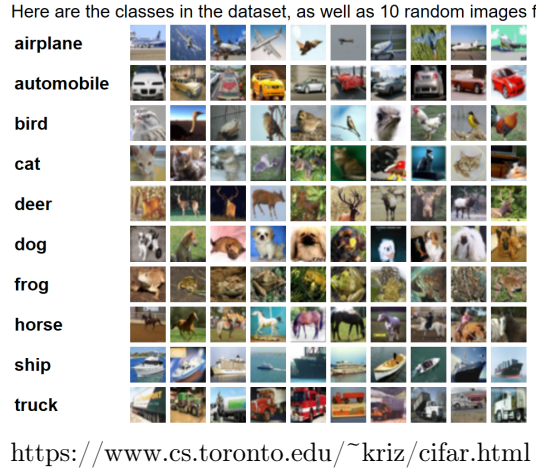


Figure 4.1: CIFAR-10 Classes

of hyperparameters lead to high accuracies. The training set consists of 50000 examples while the test set has 10000 [36]. A schematic view of the dataset can be seen in 4.1.

CIFAR-100 is very similar. The only difference is that CIFAR-100 contains 100 categories/classes with 600 images each [36]. This classification task requires even more complex models, given that there are more classes of similar objects.

CIFAR-10 takes 16 trials of random search to achieve 80% classification accuracy while CIFAR-100 requires much more than that [1]. We, therefore, select CIFAR-10 as a benchmark for our experiments. It provides the right level of complexity to allow us to observe differences between automatic hyperparameter tuning methods. It is also flexible enough because it allows focusing on the tuning methods rather than the sophistication of the machine learning models that need to be tuned. This is beneficial because it means we also make effective use of our resources: time, computing power etc.

State of the art performance

At the time of writing this paper:

- CIFAR-10: 99% accuracy [34]
- CIFAR-100: 75.72% accuracy[33]

4.3 Future experiments

Throughout this paper, we will continue with 2 target datasets CIFAR-10 and MNIST. Apart from the inherent advantages of CIFAR-10 dataset (presented before) there is another benefit that it brings. Most of the existing methods are already benchmarked against CIFAR-10 so this makes it a very good candidate for our experiments because we can easily compare with existing results. Similarly, even though MNIST is easier to classify, computations on MNIST are faster and, again, it is a popular benchmark in literature. For the proposed methods, we want to compare the results using these datasets and establish some concrete hierarchy. The code for the CNN that we used can be found at:

https://github.com/jopasserat/autotune/blob/master/benchmarks/ml_models/cudaconvnet2.py

Chapter 5

Optimizer evaluation criteria

The most natural way to evaluate/compare *optimization methods* is through A/B testing. Intuitively, the best combination of hyperparameters is considered to be the one that gives the lowest error (best result) as soon as possible (among those with lowest error). It is worth remembering that *each loss function corresponds to a combination of hyperparameters*. Now, we can formally define the following metrics as benchmarks.

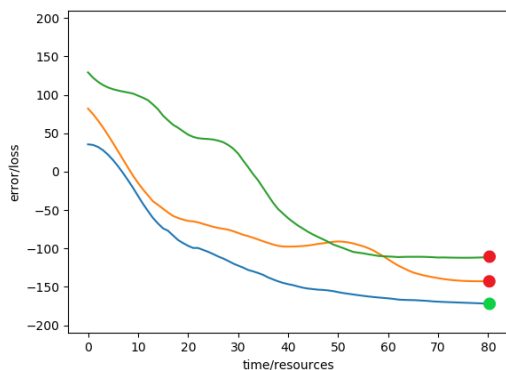
5.1 Metrics

5.1.1 Description

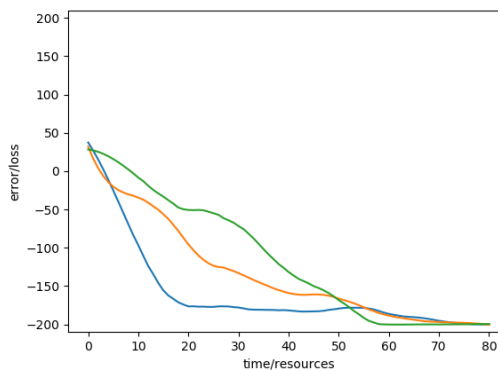
1. **Final error** This represents simply the combination of hyperparameters that yields the lowest error and conversely, the highest accuracy, with a given amount of resources like time or number of training examples/epochs. Mathematically, this metric is the *last value of the loss function*. Of course, we aim to minimize this.
2. **Early convergence** Among those combinations of hyperparameters with lowest final errors, one will prefer those that approached this optimum the fastest because this essentially means the shortest waiting time. More formally, this metric can be regarded as the area under the loss function. Our aim is to minimize this as well.

5.1.2 Reducing metrics to loss functions

The metrics above can be fully defined in terms of loss functions. Below, we can see an example of loss functions resulting from 3 optimization methods.



Blue gives lowest final error



Blue converges the fastest/earliest

On the left hand side, one can observe that after A/B testing the best combination of hyperparameters coming from the "Blue optimizer" is better than the others because it finishes with the lowest error. Conversely, on the right hand side, all functions end with the same error but the one given by the "Blue optimizer" converges sooner than the rest. Hence, it is preferred over the others.

5.2 PROFILES - statistical profiles of loss functions

The previous examples present some evidence that one optimization method might be better but, to be more convincing, the problem requires some further statistics. For example, one optimizer being *consistently* better than the others would make a much better justification for picking it.

5.2.1 Profiles

Next, we will introduce some **profiles** that capture statistics of several loss functions. For the purpose of comparing optimizers, the loss functions that we compute statistics for correspond to the best set of hyperparameters found by an optimizer. This set of profiles is more comprehensive than what is usually found in literature (mean and standard deviation) because in this paper we cater for Hyperband, which is an optimization method thoroughly investigated in this paper.

- *Dynamic profiles* - applied at every X (eg. epoch), the profile can be seen *vectorially* or as a function of X. To calculate profiles, assume that we have n loss functions - f_1, f_2, \dots, f_n .

1. Average

$$\mu(x) = \frac{1}{n} \cdot \sum_{i=1}^n f_i(x)$$

2. Median

$$M(x) = Med(\{f_1(x), \dots, f_n(x)\})$$

3. Standard deviation

$$\sigma(x) = \sqrt{\frac{\sum_{i=1}^n (f_i(x) - \mu(x))^2}{n}}$$

4. **Dynamic order** Hyperband-based optimizers care about the relative ordering of loss functions when halving. Hence, the need for an order profile. This profile can be seen as a measure of how parallel the loss functions are, more specifically, how much (as a percentage) of the relative order from the previous epoch is preserved at the current epoch. So, for every X (eg. epoch), let us define:

$$less(i, x) = \{f_j(x) | f_j(x) < f_i(x)\} \text{ and } great(i, x) = \{f_j(x) | f_j(x) > f_i(x)\}$$

$$dynamic_order(x) = \frac{1}{n} \sum_{i=0}^n \frac{|less(i, x) \cap less(i, x-1)| + |great(i, x) \cap great(i, x-1)|}{n-1}$$

- *Static profiles* - applied on whole loss functions (not at every X), the profile is a *scalar*:
1. **Order at ends** For the same reasons described in dynamic order profile, we also compute how much of the *initial order is preserved at the end*. That is, while the dynamic-order profile records how much of the relative ordering is preserved from step $x-1$ at step x , order-at-ends profile records this change between steps 1 and $\max(x)$.

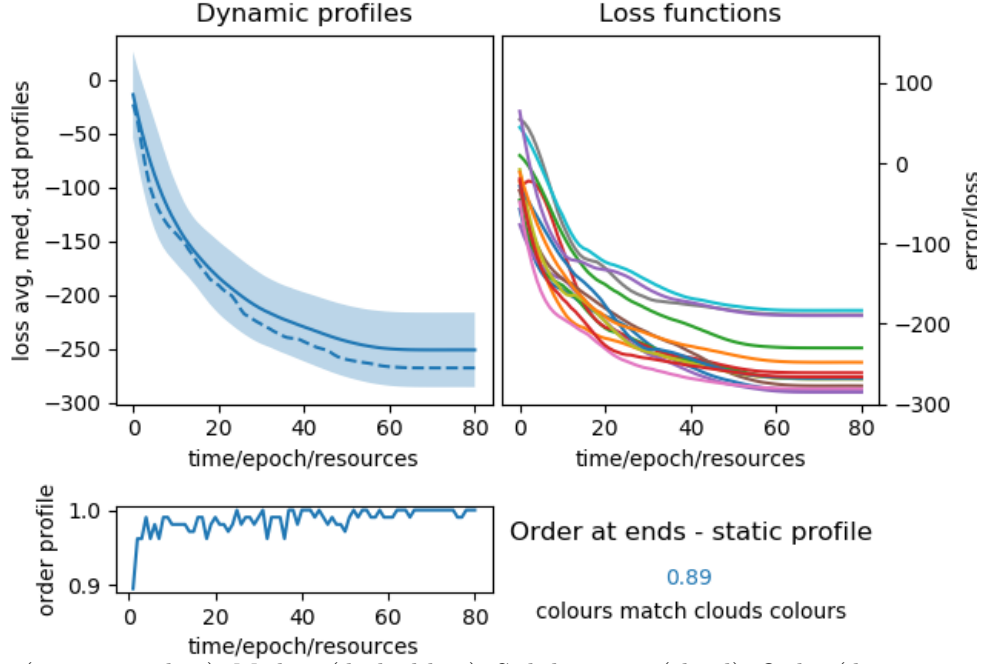
$$order_at_ends = \frac{1}{n} \sum_{i=0}^n \frac{|less(i, \max(x)) \cap less(i, 1)| + |great(i, \max(x)) \cap great(i, 1)|}{n-1}$$

where:

- *less* and *great* have the same meaning as for *dynamic_order(x)* profile
- *max(x)* is the maximum epoch, last X point from all loss functions

Note that this profile is static, *order_at_ends* is a number/scalar and does not depend on x . It is also important to note that this profile is relevant to compute when the loss functions have the same length (in epochs for example). That is, it would not make sense on those loss functions that have been stopped earlier by Hyperband for example.

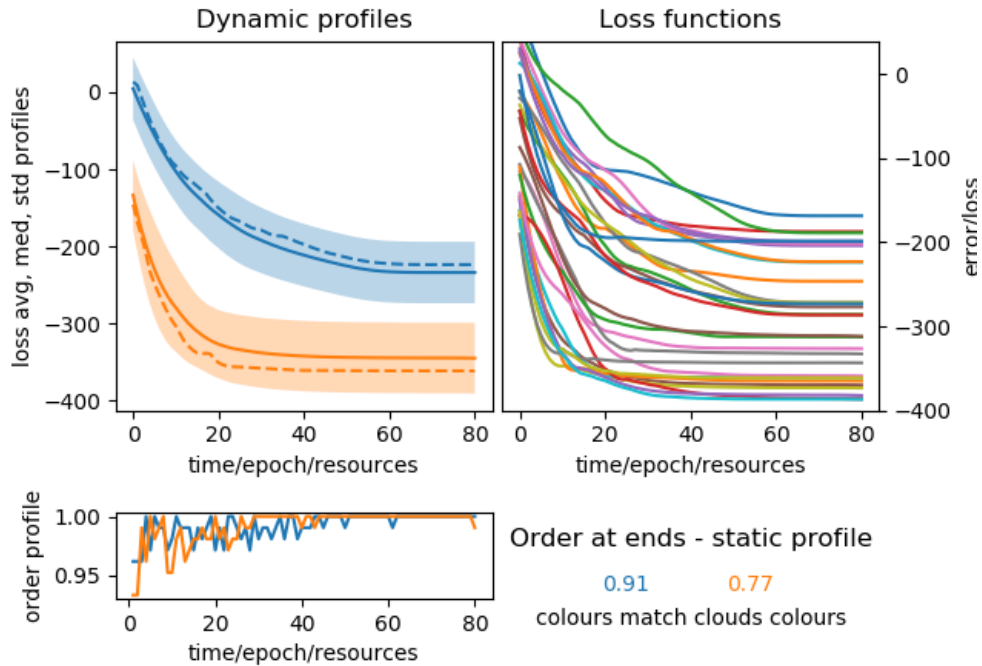
For example, suppose that we run an optimizer several times. The loss functions corresponding to the best set of hyperparameters found in each optimization are depicted below along with their profiles.



Average (continuous line), Median (dashed line), Std deviation (cloud), Order (dynamic profiles) and order-at-ends (static profile) computed from true loss functions

5.2.2 Comparing profiles

In a similar fashion, we can use these profiles to compare optimization methods. This way, one learns whether an optimizer is *consistently* better with some statistical confidence in these results. For example, below, 2 optimizers (which gave loss functions in the yellow cloud and respectively in the blue cloud) are compared based on their profiles. Note that on the right the loss functions coming from both optimizers are put altogether. These loss functions correspond to the best hyperparameters found by a single run of an optimizer.



Profiles comparison between several best loss functions given by 2 optimizers (yellow cloud optimizer is better because it gives *consistently* lower errors)

5.3 Estimated density of optimal final errors (EPDF-OFE)

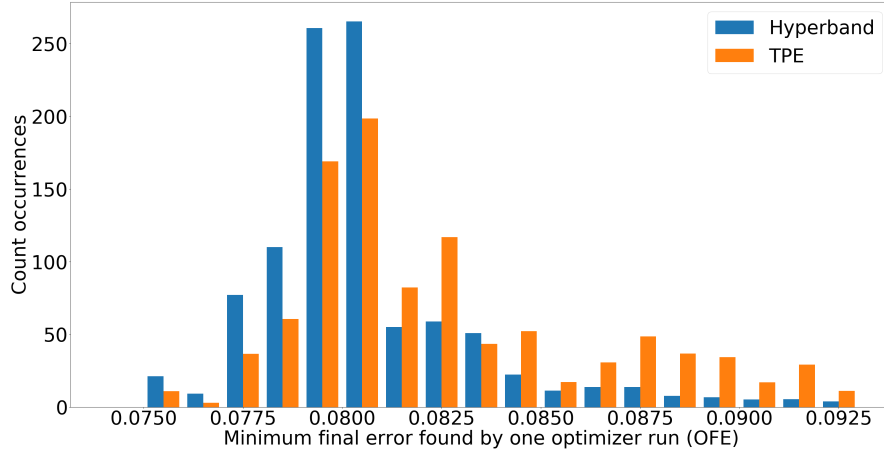
In the literature, typical evaluation methods in the literature lack precision when it comes to compare optimizers that are apparently close but the difference between them is statistically significant. Also, they lack reproducibility and uniformity across papers and datasets. Hence the need for statistically sound evaluations.

We propose a quantitative comparison of the estimated probability density functions (PDFs) on several optimal values found by optimizers. That is, run each optimizer many times and record the best final error found in each of those runs.

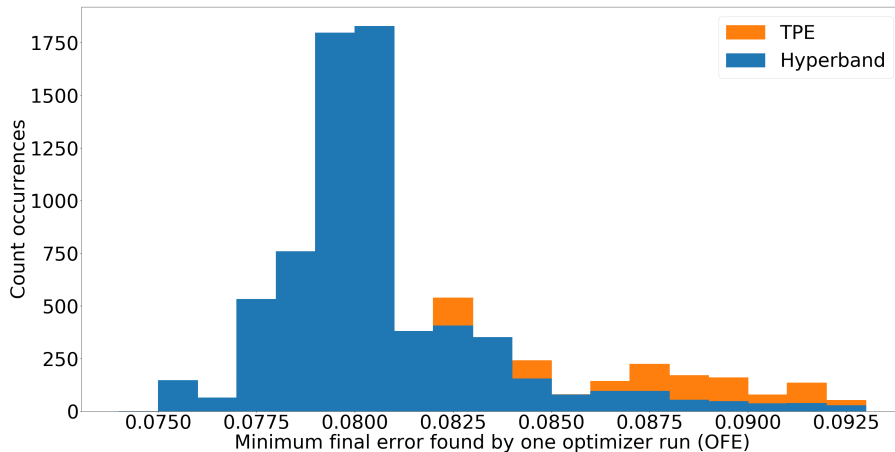
The ultimate goal is to find the distribution from which these optimums were sampled. One can plot a histogram from these optimums yielded by each optimizer on which some density estimation technique is applied (eg. Gaussian/Epanechnikov kernel smoothing).

5.3.1 Example

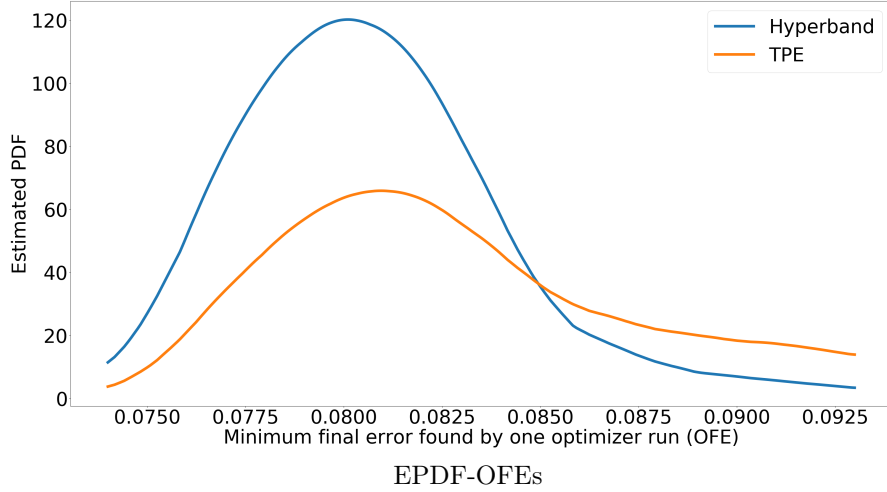
Next, we will illustrate the method of computing and comparing EPDF-OFEs for Hyperband and TPE. We expect from [13] that Hyperband is outperforming TPE. Assume a Logistic Regression model used to classify MNIST data was optimized 7000 times by each optimizer. Note that each optimizer operated within the same budget constraints. We record the optimal final error from each run and compute a histogram using these values as you can see below.



Comparative histogram

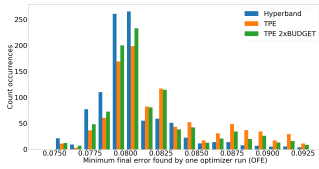


Overlapped histograms

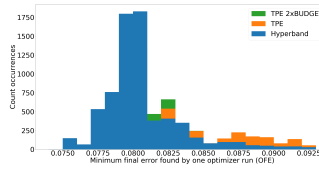


The last step, is estimating the PDFs these histograms come from. We used *Epanechnikov kernel smoothing* with a bandwidth of 0.005 on the histograms above to estimate the density function from which the optimal final errors have been drawn.

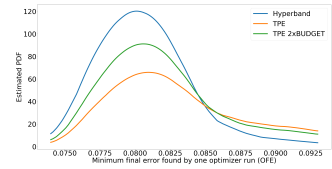
For the plots above, it is clear that Hyperband is better than TPE with high statistical significance. Hyperband tends to return smaller values as its result, since the mode of its PDF is shifted to the left of the mode of TPE PDF. Statistically, Hyperband produces lower minimum final errors (OFEs) as shown by the the mean, median, mode, standard deviation and several percentiles. To illustrate this even further, in addition to the previous comparison, we added TPE which was run with twice as much resources as Hyperband and TPE have been previously run.



Comparative histogram



Overlapped histograms



EPDF-OFEs

We believe that this is a robust method that shows that Hyperband is better than TPE 2xBUDGET which, in turn, is better than TPE (same amount of resources as Hyperband). All these differences are statistically significant.

5.3.2 Quantitative comparison of EPDF-OFEs

Given two estimated probability density functions of optimal final errors (EPDF-OFEs), we are interested in telling whether the difference between them is statistically significant. That is, we need to compare 2 distributions. There are several statistical tests that can measure this: Z-test, Kolmogorov-Smirnov test etc. In our future experiments, when we mean statistically significant that means in the Kolmogorov-Smirnov sense. For example:

EPDF-OFEs	KS p-value	is significant
TPE - TPE 2x	$8.53 \cdot 10^{-114}$	yes
TPE 2x - Hyperband	$9.57 \cdot 10^{-205}$	yes

The better EPDF-OFE is the one that is visually peakier to the left and flatter to the right. Quantitatively, when comparing 2 EPDF-OFEs we check for ordering in means (i.e. lower mean is better). Note that the Kolmogorov-Smirnov is applied on the samples not on the smoothed EPDF-OFE so the p-value is not sensitive to the choice of window size.

5.3.3 Ideal evaluation: profiles of multi-model multi-dataset EPDF-OFEs

The same computation can be applied on any number of models and/or datasets. In the end, all these separate PDFs (EPDF-OFEs) can be aggregated using the same profiles described in the previous section. That is, we believe that the ultimate/ideal way of comparing optimizers (at least in terms of final errors) is to compare the following profiles over PDFs of optimal final errors: average, median, standard deviation dynamic order and static order at top quartile profiles. Of course, before computing the profiles all PDFs' X-axis should be normalized. In practice, collecting data for such an evaluation is extremely time consuming. So, this must be a collective effort over a few months at least. Moreover, the more variety in the datasets the more comprehensive the evaluation will be.

This method of evaluation is ideal because it can tell precisely which are the strengths and weaknesses of an optimizer, it can be used to find a global profile of the optimizer (that is, evaluating how good it is in general) and also it can be used to determine which are the datasets/models/types of problems it works best on. All these are the major challenges in the field. So, because this metric answers all of these questions we called it ideal.

5.4 Typical optimizer evaluation metrics in literature

Most often optimizers are evaluated in terms of best final error found and the loss function that produced it (hence, in terms of the area under the curve as well). This is, of course, subject to luck since most optimizers are at least partly based on random search. A reduced number of papers (usually the most prolific ones), like Hyperband [13] evaluate optimizers in terms of average and standard deviation profiles of loss functions that give best final error. Also, these prolific papers (eg. [13]) report results on several datasets as well. However, they are made in terms of a single run of each optimizer for each dataset, or at least this is what a reader would assume because there are no specific details on how this multi-dataset comparison is made.

Therefore the EPDF-OFE evaluation as well as the ideal evaluation represent novel improvements in the field of optimizer evaluation. Also, from the studied papers we have never come across order profiles (neither dynamic nor static). So, this is another innovation for optimizer evaluation.

5.5 Summary

Achievements (so far)

In this chapter we built up *new and robust optimizer evaluation/comparison methods*. This is a significant improvement from what is usually used in literature because it measures differences quantitatively. The proposed ideal evaluation addresses some of the most important challenges in the field: how good is an optimizer *in general* and it would allow researchers to tell what optimizers are better suited for what types of problems.

Issues (to be solved)

The problem that comes with these new optimizer evaluation criteria is that they require several runs of each optimizer in order to compute statistics. This might be very time consuming, so this might not be practical. Fortunately, we will introduce solution(s) to this problem in the following chapters.

Chapter 6

Optimizer correctness and testing

In general, the main concern an optimizer designer has is to beat the state of the art in terms of some metrics (eg. fastest or most accurate). After elaborating an optimizer, it is validated on several datasets and machine learning models. This evaluation method is indicative for how good the optimizer is but it *does not guarantee that the optimizer will always preserve the intended behaviour*. That is, optimizer testing is virtually nonexistent. This is understandable since one run of an optimizer can take hours, days or even months.

Next, we will outline two major flaws which could have been easily caught if optimizer testing was in place. Afterwards, in a separate chapter, we will present a solution to optimizer testing (Gamma simulation) which works in negligible time.

6.1 Floating point error on all Hyperband Python implementations

In the original Hyperband paper, [13], the algorithm is introduced as follows:

HYPERBAND (Finite horizon)
Input: Budget B , maximum size R , $\eta \geq 2$ ($\eta = 3$ by default)
Initialize: $s_{\max} = \lfloor \log(R) / \log(\eta) \rfloor$
For $k = 1, 2, \dots$
 For $s = s_{\max}, s_{\max} - 1, \dots, 0$
 $B_{k,s} = 2^k$, $n_{k,s} = \lceil \frac{2^k \eta^s}{R(s+1)} \rceil$
 $\hat{t}_s, \ell_{t_s, R} \leftarrow \text{SUCCESSIVEHALVING}(B_{k,s}, n_{k,s}, R, \eta)$

The number of brackets is determined by s_{\max} which, in turn, is computed as $\lfloor \log_\eta(R) \rfloor$. Mathematically this algorithm is perfectly correct and it should do what the paper describes. However, when $\lfloor \log_\eta(R) \rfloor$ is implemented as `int(log(R)/log(eta))` we can get floating point precision errors where `log` comes from `math` or `numpy` libraries. Let us illustrate this:

```
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] on win32
In [2]: import math, numpy as np
In [3]: math.log(243) / math.log(3)
Out[3]: 4.999999999999999
In [4]: int(math.log(243) / math.log(3))
Out[4]: 4
In [5]: np.log(243) / np.log(3)
Out[5]: 4.999999999999999
In [6]: int(np.log(243) / np.log(3))
Out[6]: 4
```

We suspect that this issue has not been caught earlier because in literature examples have $R = 81$ and $\eta = 3$ for which the same calculations do not have a floating point error:

```
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] on win32
In [2]: import math, numpy as np
In [3]: math.log(81) / math.log(3)
Out[3]: 4.0
In [4]: np.log(81) / np.log(3)
Out[4]: 4.0
```

Error consequences Fortunately this error does not affect the correctness of Hyperband (in terms of optimums found). However, it does affect exploration of the parameter space and *severely* impacts waiting time. The outer/hedging loop of Hyperband goes over $s_{max} + 1$ brackets, that is, mathematically $\lfloor \log_\eta(R) \rfloor + 1$. A few implementations always go over $s_{max} + 2$ brackets. In both cases, the number of brackets is fully determined by the value of the logarithm. As we have illustrated above, when $R = 243$ and $\eta = 3$, s_{max} would be 4 instead of 5 if computed with a naive logarithm implementation as it happens in pretty much all Hyperband implementations including the one published by the author. So, Hyperband will do one less bracket and this has a few implications:

Bracket	s=5		...	i
Population/Resources	n_i	r_i		
Generation	243	1	...	0
Halving	81	3	...	1
	27	9		2
	9	27		3
	3	81		4
	1	243		5

Expected first bracket

Bracket	s=4		...	i
Population/Resources	n_i	r_i		
Generation	81	3	...	0
Halving	27	9	...	1
	9	27		2
	3	81		3
	1	243		4
				5

Actual first bracket

- **Waiting time** One bracket can take several hours. So, being one bracket down can give substantial errors in terms of expected waiting time. For example, in some of our experiments a bracket takes 5 hours. Imagine that, in industry, someone has limited time and wants to use Hyperband to tune some hyperparameters. He/she can calculate the amount of resources (R) which would maximize the use of his/her time. But, in practice the algorithm will terminate several hours earlier (which means that the time was not used as efficiently as expected).
- **Less exploration** This error does not influence exploitation since halving works as usual, but it harms exploration of the parameter space because brackets' initial populations are smaller. The original Hyperband paper [13] observes that the first bracket is one of the most dominant brackets (on tested datasets) in terms of performance and this floating point error completely cuts it down.

Python solution We solved this problem with `mpmath` module to get this exact arithmetic in Python.

```
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] on win32
In [4]: import mpmath
In [5]: mpmath.mp.dps = 64
In [6]: mpmath.log(243) / mpmath.log(3)
Out[6]: mpf('5.0')
In [7]: int(mpmath.log(243) / mpmath.log(3))
Out[7]: 5
In [8]: mpmath.log(81) / mpmath.log(3)
Out[8]: mpf('4.0')
In [9]: int(mpmath.log(81) / mpmath.log(3))
Out[9]: 4
```

Programming languages and Operating Systems We evaluated the impact of this issue across various Operating Systems and programming languages. So we managed to reproduce it on Linux and Windows with Python 3.7.1, 3.6.1, 2.7.10. This issue also appears in Java and Scala. However, in other languages like C and Q the computation is precise.

Virtually all implementations are affected At the time of writing this paper, (June 2019), from the implementations that we have reviewed all suffer from this floating point issue. These vulnerable implementations include the original one published by one of Hyperband authors (K. Jamieson) in [38] and the implementation of a reputed paper, BOHB [21]. A list of affected Hyperband implementations can be found below:

1. <https://github.com/zygmuntz/hyperband/blob/master/hyperband.py#L18>
2. <https://gist.github.com/PetrochukM/2c5fae9daf0529ed589018c6353c9f7bfile-hyperband-py-L204>

3. https://github.com/electricbrainio/hypermax/blob/master/hypermax/algorithms/adaptive_bayesian_hyperband_optimizer.py#L26
4. https://github.com/polyaxon/polyaxon/blob/c8bc14e92b45579ecc19f2e51ae161f84d35d817/polyaxon/hpsearch/search_managers/hyperband.py#L58
5. <https://github.com/thuijskens/scikit-hyperband/blob/master/hyperband/search.py#L346>
6. <https://github.com/automl/HpBandSter/blob/367b6c4203a63ff8b395740995b22dab512dcfef/hpbandster/optimizers/hyperband.py#L60> [21]
7. <https://homes.cs.washington.edu/~jamieson/hyperband.html> formerly:
<https://people.eecs.berkeley.edu/~kjamieson/hyperband.html> [38]

6.2 Wrong Hyperband+TPE hybrid design in paper

The first paper to publish a hybrid algorithm between Hyperband and TPE is Wang et al. [17]. It reported a performance improvement from pure Hyperband. However, we found that this paper uses a flawed algorithm that harms exploitation of the parameter space and breaks the budget constraint of Hyperband.

Firstly, the paper introduces the algorithms for pure Hyperband and pure Bayesian optimization in pseudocode which are both correct.

Algorithm 1 Hyperband algorithm

input: maximum amount of resource that can be allocated to a single hyperparameter configuration R , and proportion controller η
output: one hyperparameter configuration

```

1: initialization:  $s_{max} = \lfloor \log_{\eta}(R) \rfloor$ ,  $B = (s_{max} + 1)R$ 
2: for  $s \in \{s_{max}, s_{max} - 1, \dots, 0\}$  do
3:    $n = \left\lceil \frac{B \cdot \eta^s}{R(s+1)} \right\rceil$ ,  $r = R\eta^{-s}$ 
4:    $X = \text{get\_hyperparameter\_configuration}(n)$ 
5:   for  $i \in 0, \dots, s$  do
6:      $n_i = \lfloor n\eta^{-i} \rfloor$ 
7:      $r_i = r\eta^i$ 
8:      $F = \{ \text{run\_then\_return\_obj\_val}(x, r_i) : x \in X \}$ 
9:      $X = \text{top\_k}(X, F, \lfloor n_i/\eta \rfloor)$ 
   return configuration with the best objective function value

```

Algorithm 2 Bayesian optimization

```

1: initialization:  $D_0 = \emptyset$ 
2: for  $t \in \{1, 2, \dots\}$  do
3:    $x_{t+1} = \text{argmax}_x \mu(x|D_t)$ 
4:   Evaluate  $f(x_{t+1})$ 
5:    $D_{t+1} = D_t \cup \{(x_{t+1}, f(x_{t+1}))\}$ 
6:   Update probabilistic surrogate model using  $D_{t+1}$ 

```

Next, it proposes the hybrid algorithm 3 which combines the first two algorithms:

Algorithm 3 Combination of Hyperband and Bayesian optimization

input: maximum amount of resource that can be allocated to a single hyperparameter configuration R , and proportion controller η
output: one hyperparameter configuration

```

1: initialization:  $s_{max} = \lfloor \log_{\eta}(R) \rfloor$ ,  $B = (s_{max} + 1)R$ 
2: for  $s \in \{s_{max}, s_{max} - 1, \dots, 0\}$  do
3:    $n = \left\lceil \frac{B \cdot \eta^s}{R(s+1)} \right\rceil$ ,  $r = R\eta^{-s}$ 
4:   for  $i \in 0, \dots, s$  do
5:      $n_i = \lfloor n\eta^{-i} \rfloor$ 
6:      $r_i = r\eta^i$ 
7:     if  $i == 0$  then
8:        $X = \emptyset$ ,  $D_0 = \emptyset$ 
9:       for  $t \in \{1, 2, \dots, n_i\}$  do
10:         $x_{t+1} = \text{argmax}_x \mu(x|D_t)$ 
11:         $f(x_{t+1}) = \text{run\_then\_return\_obj\_val}(x, r_i)$ 
12:         $X = X \cup \{x_{t+1}\}$ 
13:         $D_{t+1} = D_t \cup \{(x_{t+1}, f(x_{t+1}))\}$ 
14:        Update probabilistic surrogate model using  $D_{t+1}$ 
15:     else
16:        $F = \{ \text{run\_then\_return\_obj\_val}(x, r_i) : x \in X \}$ 
17:        $X = \text{top\_k}(X, F, \lfloor n_i/\eta \rfloor)$ 
   return configuration with the best objective function value

```

The first thing to observe is that in pure Hyperband (Algorithm 1) `top_k` the halving function is applied *for all* $i \in \{0..s\}$ where $s \in \{0..s_{max}\}$. On the other hand, in the hybrid algorithm (Algorithm 3) `top_k` is not applied when $i == 0$.

Bracket	s=4		...	i
Population/Resources	n_i	r_i	...	
Generation	81	1	...	0
Halving	27	3	...	1
	9	9		2
	3	27		3
	1	81		4

Expected first bracket

Bracket	s=4		...	i
Population/Resources	n_i	r_i	...	
Generation	81	1	...	0
Halving	81	3	...	1
	9	9		2
	3	27		3
	1	81		4

Actual first bracket

Note that the issue persists over *all brackets*, above we just exemplified on the first one.

Error consequences The published algorithm **breaks the maximum resource budget constraint** that Hyperband makes. In the above example, when $i == 1$ the budget is $81 \cdot 3 = 243$ instead of 81 as it should be. This issue can make the published results better than they would be if the budget constraint was respected because at $i == 2$ halving is more accurate (it chooses best 9 from 81 individuals each trained for 4 epochs/resources). If the budget constraint was respected, at $i == 2$ halving would choose only from 27 individuals each trained for 4 epochs. This design favours loss functions that performed badly for 1 resource/epoch but improved significantly over the next 3 resources/epochs. That is, if the budget constraint was respected, these functions would be cut/disregarded in the first halving but as per Algorithm 3 they survive this halving and can eventually become the best.

If we were to judge this in terms of an augmented maximum budget, the hybrid proposed in [17] (Algorithm 3) is sub-optimal because, most of the time, it would run massively under maximum capacity.

Solution Calling `top_k` (i.e. to halving) for any i including $i == 0$, just like Hyperband does, fully resolves the issue.

6.3 Summary

The issues described previously are not huge but are not negligible either. They could have been easily avoided with some form of testing. However, because machine learning models take a lot of time, testing has been impractical so far. The next chapter, introduces the Gamma loss function simulation which will enable optimizer testing in negligible time.

Chapter 7

Gamma loss function simulation

Optimization literature usually consists of finding minima/maxima of various functions. The expression of these functions may be known (eg. Branin function), or not. Of course, optimization is more difficult when the expression is unknown. In such cases, evaluating the true function at a point is expensive in terms of time, money, resources etc. Bayesian optimization aims to find this optimum in as few function evaluations as possible. Hyperparameter optimization specifically follows the pattern outlined above:

- *Given:* a combination of k hyperparameters (a k dimensional point)
- *Minimize:* the "final" loss function, final meaning after training for N resources (time, number of training examples etc), usually until no further improvement is being recorded.

Therefore, this can be seen as a function with an unknown expression which is expensive to compute, just like general optimization problems look like. Fortunately, in machine learning, one can also observe intermediate results - intermediate values of the loss function. Researchers have tried to use these intermediate results to use fewer resources/to speed up computations. The study of early stopping is quite vast and became a sub-field of machine learning.

This chapter outlines a method to simulate loss functions for a machine learning algorithm in negligible time.

7.1 Motivation

Such a simulation can address many concerns like:

Ensuring correctness and testing Ensuring correctness of the behaviour of a hyperparameter optimization method might require many runs of true machine learning models and hence, would take a lot of time. Furthermore, bugs might never be caught. Researching for this project I identified a few such flaws. So, because of the expensiveness of machine learning, unit testing on hyperparameter optimization methods is (almost) absent.

Statistical comparison of early stopping optimizers Also, one might be interested to compute *statistics* about how early stopping optimization methods like Hyperband (or hybrids based on them) perform/compare, for example finding average performance or computing the EPDF-OFE, without waiting days to perform averages on real machine learning problems (eg. CIFAR). Because an early stopping method is essentially (only) looking at loss functions, it would be good if we could simulate them and not wait for learning to happen.

Architecting optimizers When combining several stand-alone methods to create hybrids there are many potential choices of architectures. One way to validate an architecture or to understand its strength and weaknesses is by evaluating it on a simulation before, jumping into academic datasets and, afterwards, real-world datasets.

Timely optimizer selection Comparing hyperparameter optimization methods depends on the underlying datasets. So, often researchers compute statistics across multiple datasets or across multiple runs. For example, the authors of Hyperband perform this kind of analysis (over multiple datasets) in [38]. This is certainly time consuming and might not be practical in real-life situations. Instead, a few simulations would be indicative before jumping on the real problems.

So, the main reasons behind the need for a simulation of loss functions are computing statistics over several runs of an optimizer in a timely manner and ensuring correctness of such a method.

7.2 Simulation requirements

Any simulation must satisfy a few requirements that match the behaviour of real-life loss functions.

1. Have one or more global minimums. Ideally, they are known and the values of the hyperparameters that yield these minimums are also known. Also, the simulation should not be easy to optimize. Preferably the easiness to optimize can be customized.
2. Be able to simulate *all* shapes of loss functions. In general, loss functions:
 - Plateau after some training time (that is no improvement - with more resources).
 - Can go down with different levels of steepness at different points in time.
 - Some of them (the very bad ones) can even go up.
3. The simulated loss functions should *not* be "parallel". That is, the relative performance should change over time. Take for example, the case of large vs small learning rate - the large learning rate will be steep down at the beginning but will plateau quite soon, while the small learning rate will create a curve that is decreasingly slowly but might eventually plateau lower (better) than the big learning rate.
4. After some loss functions were generated (with some randomness) we might want to run multiple methods on the exact same loss functions. So the simulation should generate loss functions with some randomness but should also be able to deterministically re-create any of those loss functions as often as needed.

7.3 Solution

The simulation works regardless of the type of resource used (time, number of training examples etc). However, for simplicity I will refer to resources as time in the explanations below because time is the most common and concerning resource. For now, it is important to note that the simulation can simulate not only time but also any other resource.

7.3.1 Underlying function

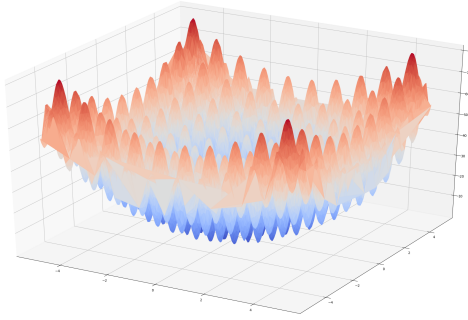
Basing the simulation on a known function would be enough to satisfy the first requirement. The choices for these experiments were Rastrigin, Drop-Wave and Branin function. Surface plots for each function can be seen below function.

Let us call the underlying function u . Set simulated/generated loss functions such that:

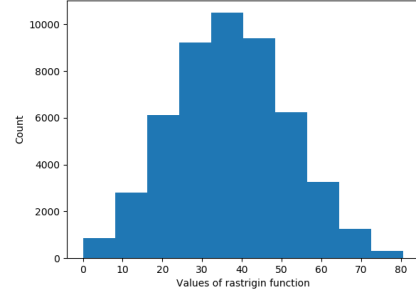
1. **at time 0 (beginning):** the simulated function has the value of the $u(x)$ function (eg. $\text{branin}(x, y)$) potentially shifted by a constant $start_shift$ (which is typically 0).
2. **at time n (end):** the simulated loss function has the value: $u(x) - end_shift$ (so shifted downwards by a constant).

From the requirements above, this design satisfies the first requirement in its "ideal" form. Thus, we created the first few parameters of the simulation: choice of underlying function u , $start_shift$ (typically 0) and end_shift . Note that, later, we will amend the fixed start point with some initial noise.

Rastrigin function



Rastrigin surface plot

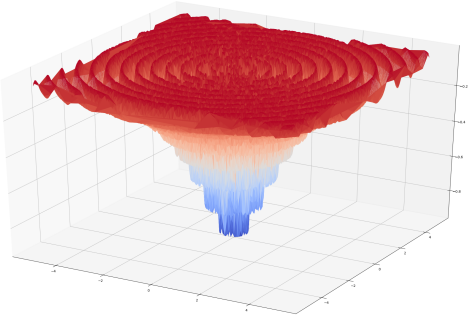


Distribution of Rastrigin values

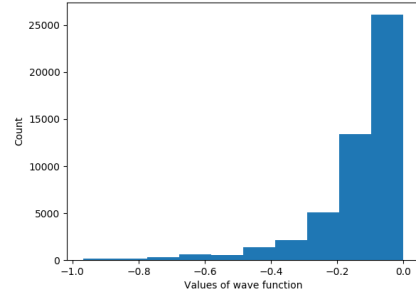
$$rastrigin(x) = 10d + \sum_{i=1}^d [x_i^2 - 10 \cdot \cos(2\pi x_i)], x_i \in [-5.12, 5.12]$$

where d is the arity. In the plot above $d = 2$. $\min_x rastrigin(x) = 0$, $\operatorname{argmin}_x rastrigin(x) = 0$

Drop-wave function



Drop-wave surface plot

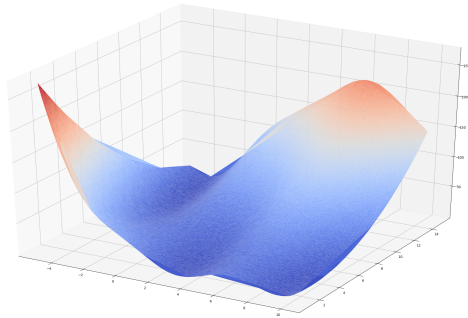


Distribution of Drop-wave values

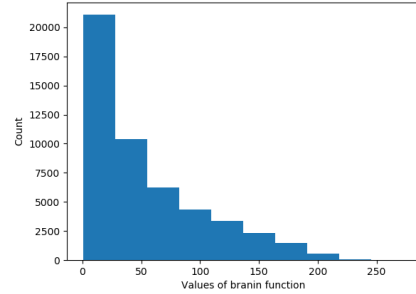
$$wave(x, y) = -\frac{1 + \cos(12 \cdot \sqrt{x^2 + y^2})}{0.5 \cdot (x^2 + y^2) + 2}, x, y \in [-5.12, 5.12]$$

where $\min_{xy} wave(x, y) = -1$, $\operatorname{argmin}_{xy} wave(x, y) = (0, 0)$

Branin function



Branin surface plot



Distribution of Branin values

$$branin(x, y) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1 - t)\cos(x_1) + s, x \in [-5, 10], y \in [0, 15]$$

where $a = 1$, $b = \frac{5.1}{4\pi^2}$, $c = \frac{5}{\pi}$, $r = 6$, $s = 10$, $t = \frac{1}{8\pi}$, $\min_{xy} branin(x, y) = 0.397887$, $\operatorname{argmin}_{xy} branin(x, y) \in \{(-\pi, 12.275), (\pi, 2.275), (9.42478, 2.475)\}$

Note that only *rastrigin* is n -dimensional while *wave* and *branin* are 3-dimensional. Also, apart from *branin*, the functions have several local optima so, naturally, they are hard to optimize globally.

Choice of underlying function

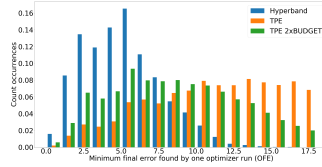
The arity of the simulation (number of hyperparameters that it takes) is not bounded so, as a model, the simulation provides enough generality. Of course, in three-dimensional spaces we can experiment with multiple underlying functions while for higher dimensions the set of "good" underlying functions is more restricted. Hence, in the next experiments we will work mainly in 3D.

The underlying function, u , defines completely the function of final errors. For example, if we set all simulated loss functions to finish at $rastrigin(x) - 200$ the final errors will be fully defined by $rastrigin$ function since the constant (end_shift) does not play any role in the function landscape. This is particularly important when putting the simulation through Bayesian optimizers, since these optimizers only consider final errors and do not take into account any of the intermediate points from the loss function (they only use the last point which, by definition of the simulation, is $u(x) - end_shift$). Essentially, any Bayesian optimizer will optimize the underlying function.

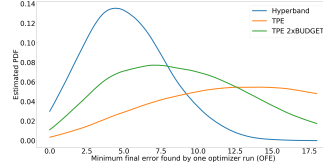
The choice of the underlying function affects if an optimization method finds the minima/maxima and/or how fast it approaches them. Furthermore, the distribution of values and the number of local optima indicates how easy/hard to optimize a problem is. For example, Branin is relatively easy since it is smooth (only 3 local minima) and its values are usually close to these minima (see that the distribution of Branin values is skewed towards the global minimum). On the other hand, Drop-wave function is a hard optimization problem since it has several local optima (it is also multi-modal) and the distribution of Drop-wave values is skewed away from the global minimum. So, basically, the hardness of the problem can be appreciated in terms of the distribution of values of the underlying function u and in terms of the number of local minima.

Proof - EPDF-OFEs Over 3000 runs of each Hyperband and TPE, the EPDF-OFEs for different underlying functions u are:

Rastrigin

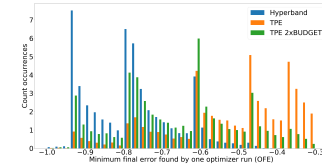


Comparative histogram

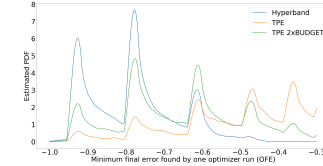


EPDF-OFEs

Drop-wave

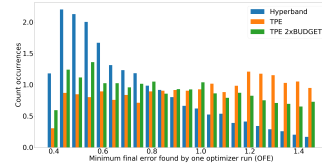


Comparative histogram

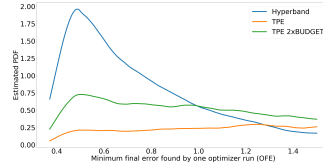


EPDF-OFEs

Branin



Comparative histogram



EPDF-OFEs

So, in all 3 cases Hyperband (which, in this case, is nothing else than Random search since the loss functions are flat) performs better than TPE. This is a good preliminary evaluation of the simulation since regardless of underlying function this hierarchies hold with statistical significance. Hence, we set some expectations on how much should the EPDF-OFEs differ to have some statistically significant difference. We will perform experiments with all 3 underlying functions. However, for the general case we recommend Rastrigin since it is n -dimensional and because the difference between TPE and Hyperband/Random search is very clear.

7.3.2 Gamma process model

Given the 2 end points using the underlying function u as described above, the simulation will generate loss functions that start and end in these 2 given points ($u(x) - \text{start_shift}$ and $u(x) - \text{end_shift}$). In general, loss functions decrease and then plateau. So, let us define:

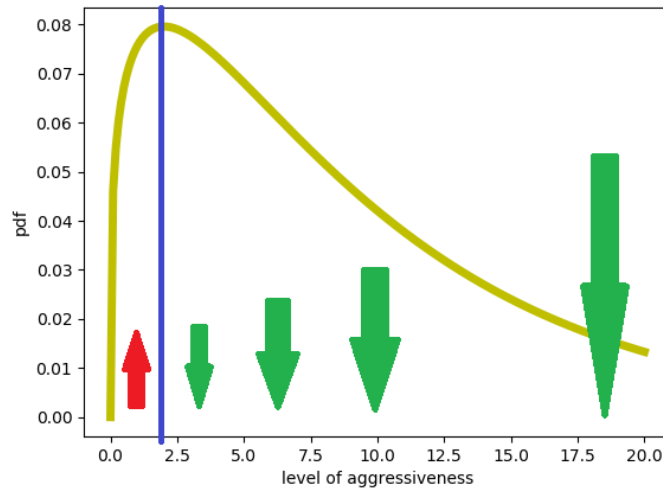
- the tendency to go down (to decrease) as *aggressiveness*
- the amount by which it goes down (decreases) as *level of aggressiveness*, where positive levels mean by how much it goes down and negative levels mean by how much it goes up

Therefore, to get that start-decreasing end-plateauing shape, the aggressiveness should decrease as time passes. A Gamma Process (based on Gamma distributions) can model perfectly this.

At any point in time the *level of aggressiveness* is drawn from a Gamma distribution. The mode of this distribution (what would occur most often) represents 0 aggressiveness (don't go up or down - stay stale). So *aggressiveness* is the variance of this distribution. Let me describe this better with some plots.

Initial aggressiveness

At time 0, the model will simulate the next point of the loss function (for time 1). Under the Gamma process model, the probability distribution for the next step (time 1) is:



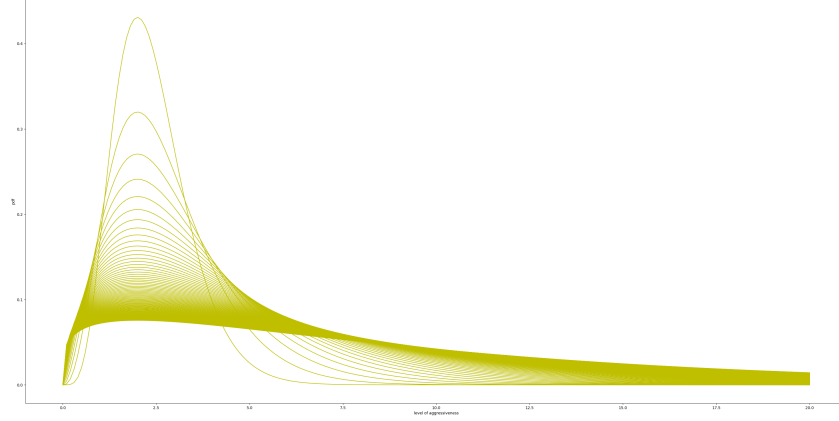
So, the blue line represents the level of aggressiveness at which the next step of the simulated function will not go up or down. To the right, the further we go the more aggressive it becomes - the green arrows show by how much we choose to go down depending on the level of aggressiveness. So the further to the right the steeper the simulated function at the next step. Intuitively, high steepness is less likely than moderate steepness which, in turn, is less likely than low steepness. On the other hand, the further we go to the left (from the blue line) the more the simulated loss function goes up.

Being at time 0, going downwards (the area under the distribution on the right side) is more likely than going upwards (the area under the distribution on the left side of the blue line). Note that the distribution is also quite "fat" at time 0 so there is room for any kind of level of aggressiveness. Also note that a normal distribution would not be suitable at this time, so this is why Gamma was chosen instead.

Plateauing effect

As time passes (more simulated epochs), to simulate the plateauing effect the simulation "narrows" (lower the standard deviation and hence the variance which is the aggressiveness). This will increase the likelihood of staying stale or going up/down by a little amount while making higher

levels of aggressiveness almost impossible. So, the Gamma Process should lower variance. However, remember that the mode (the peek of the curve) must be kept constant. For 81 epochs the Gamma process looks like this:



The "short and fat" distributions are applied at the beginning and as time passes they become "taller and slimmer". So, it be seen that the variance decreases while the mode is always the same - the peeks of the curves are vertically aligned.

Gamma process mathematical proof

While the above matches the intuition, let us dive in the formal mathematical proof. The general probability density function for a Gamma distribution is:

$$\frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

where:

- $\Gamma(\alpha)$ is the gamma function
- $\mathbb{E}[X] = \frac{\alpha}{\beta}$ (in our case - the expected level of aggressiveness)
- $Mode = \frac{\alpha-1}{\beta}$ for $1 \leq \alpha$ (in our case - the 0 level of aggressiveness, that is, where the simulation stays stale)
- $Var(X) = \frac{\alpha}{\beta^2}$

As a Gamma process is made of multiple time-dependent Gamma distributions which are, in turn, fully characterized by an α and a β , let us define the time dependent alpha and beta functions as $\alpha(t)$ and $\beta(t)$. To create a process that depends on time t every statistical measure above will depend on time as well. For example, $Var(t)(X)$ or simply $var(t)$ is the variance of the Gamma distribution that occurs at time t in the Gamma process.

Requirements:

$var(t)$ is decreasing with time t and $mode(t)$ is constant with time t

Also, let n be the maximum time. So the requirements can be rewritten in mathematics as follows:

1. $\exists k$ such that $k = mode(t) = \frac{\alpha(t)-1}{\beta(t)} \forall t \in \{0..n\}$
2. $var'(t) < 0$

From 1: $\alpha(t) = \beta(t)k + 1 \forall t \in \{0..n\}$

From 2: $var'(t) = (\frac{\alpha(t)}{\beta(t)^2})'$

From 1 and 2: $var'(t) = (\frac{\beta(t)k+1}{\beta(t)^2})' = \beta'(t)(\frac{-k}{\beta(t)^2} + \frac{-2}{\beta(t)^3}) < 0$ [3]

As $\frac{-k}{\beta(t)^2} + \frac{-2}{\beta(t)^3} < 0 \forall t \in \{0..n\}$ [4]

From 3 and 4: $\beta(t) > 0 \forall t \in \{0..n\}$, that is, $\beta(t)$ is increasing with time t [5].

Another, thing to note is that when the loss function plateaus aggressiveness should be 0. So, a simple model that respects these properties is:

$$var(t) = n - t \forall t \in \{0..n\} \text{ [6]}$$

This is decreasing and at time n , $var(n) = 0$. The next step is to solve equation [6] for $\beta(t)$.

$var(t) = \frac{k}{\beta(t)} + \frac{1}{\beta(t)^2} = n - t$ which can be rewritten as a quadratic equation:

$(n - t)\beta(t)^2 - k\beta(t) - 1 = 0$ whose solutions are:

$$1. \beta_1(t) = \frac{k + \sqrt{k^2 + 4(n-t)}}{2(n-t)}$$

$$2. \beta_2(t) = \frac{k - \sqrt{k^2 + 4(n-t)}}{2(n-t)}$$

Remember that from [5], $\beta(t)$ must be increasing, that is, $\beta'(t) > 0$. From the 2 solutions found only $\beta_1(t)$ respects this constraint. So, the $\beta(t)$ was found, based on it, $\alpha(t)$ can be recovered from [1] and hence, the whole process is defined.

7.3.3 Families of shapes and model parametrization

At any time it is possible to generate the next point's level of aggressiveness using the Gamma process model. However, requirement 2 requires to cater for *all* shapes of loss functions. Let's group loss functions that share the same characteristics in *families of shapes*. For example, all loss functions that start very steep and then plateau early can be considered to belong to a *family of shapes*.

These different families of shapes for can be parametrized in terms of 4 concepts:

1. *ML (machine learning) aggressiveness* - affects the beginning of the simulated loss function and denotes how steep it starts.
2. *necessary aggressiveness* - affects the tail of the simulated loss function and will make the loss "converge" to $bu(x) - end_shift$.
3. *up-spikiness* - this defines by how much we can go up at any point in time (how bumpy/spiky the simulated loss function is).
4. *smoothing* - using only the 3 concepts above we would generate non-smooth functions (even if the spikes are small the simulated function is not perfectly smooth).

At any time, let us define *function_debt* as how much is left to reach the target $f(n) = u(x) - end_shift$. So $function_debt(t) = f(t) - f(n)$.

ML aggressiveness works like a gradient descent (depending on function debt/on how much is left until $u(x) - end_shift$ is reached) and decays as time passes. One can think of the parameter that tunes ML aggressiveness like a learning rate. Necessary aggressiveness depends on the time left $(n - t)$ - the shorter the time left the more it will drag the function towards the target value $u(x) - end_shift$. Up-spikiness depends on the time left as well (it can be spikier at the beginning but decays as time passes - this is linear but there is room for improvement). Finally, smoothing is applied (or not) after the whole function was generated as outlined above.

These 3 concepts are tunable so one can play around with the parameters that define these families of shapes. It is important to note that these 3 parameters depend on maximum time n . For example, below there are a few samples from different families of shapes for maximum time $n = 81$ (a maximum of 81 resources is used by Hyperband authors to exemplify their algorithm):

Mathematical models on top of Gamma process

To generate the point at time t , the first step is to sample a level of aggressiveness from the Gamma process (that is, the Gamma distribution for time t). Remember from the Gamma process section that the further to the right (of the Gamma distribution mode) the sampled level of aggressiveness the steeper the next step will be in the *down* direction. Conversely, the further to the left the steeper the next step in the *up* direction.

Parameters Let us name and characterize the parameters as follows:

ML aggressiveness	float	α	user defined
Necessary aggressiveness	float	ν	user defined
Up-spikiness	float	ρ	user defined
Smoothing	bool	ω	user defined
Sampled level of aggressiveness	float	λ	internal
Mode of all Gamma distributions	float	k	internal fixed constant

Proportionality Note that the application of smoothing depends only on n (max time/epochs) but does not depend on time t since smoothing is applied as post-processing. Applying α and ν must depend on both t and n . Furthermore, spikiness does not depend on either t or n because plateauing is handled by the Gamma process and ν . ML aggressiveness α should decrease with t while necessary aggressiveness ν should become more significant as time t approaches n .

Algorithm So, given a level of aggressiveness λ sampled from the Gamma process at time $t + 1$:

- **Go down** if $\lambda > k$ as follows:

$$ml_agg(t) = f(t) + \alpha \cdot (\lambda - k) \cdot \frac{f(n) - f(t)}{100} \quad nec_agg_1(t) = (f(n) - ml_agg(t)) \cdot \left(\frac{t}{n-1}\right)^\nu$$

$$f(t+1) = ml_agg(t) + nec_agg_1(t)$$

Note that as long as $\nu \neq \infty$, $f(n)$ will always have the fixed end value because at time $t = n - 1$ $nec_agg_1 = f(n) - ml_agg(t)$ so $f(t+1) = f(n)$ where $f(n)$ is fixed beforehand in terms of the underlying function (eg. Branin).

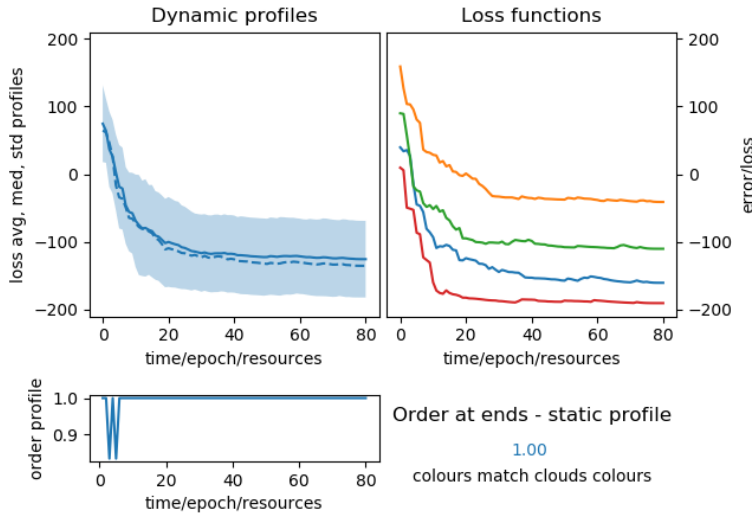
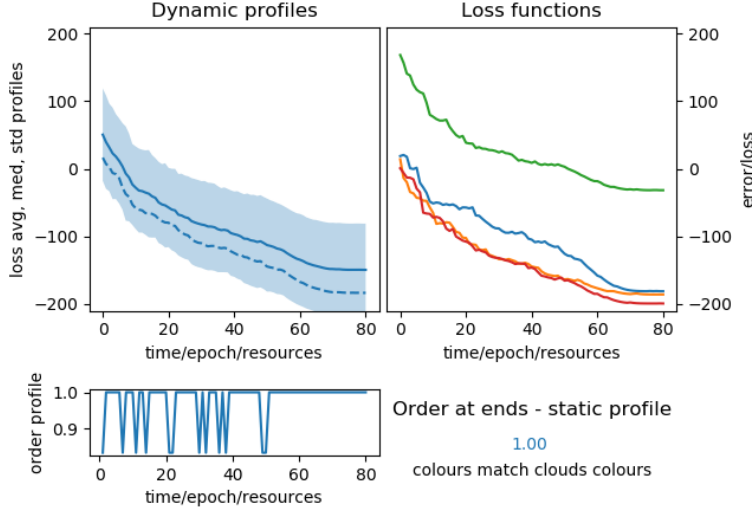
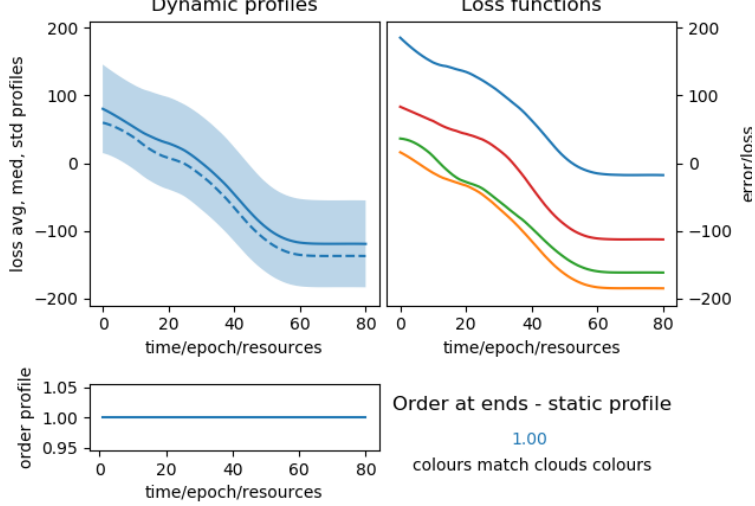
- **Go up** if $\lambda < k$ as follows:

$$up_spike(t) = f(t) + \rho \cdot \frac{1}{1+\lambda} \quad nec_agg_2(t) = (f(n) - up_spike(t)) \cdot \left(\frac{t}{n-1}\right)^{1.1 \cdot \nu}$$

$$f(t+1) = up_spike(t) + nec_agg_2(t)$$

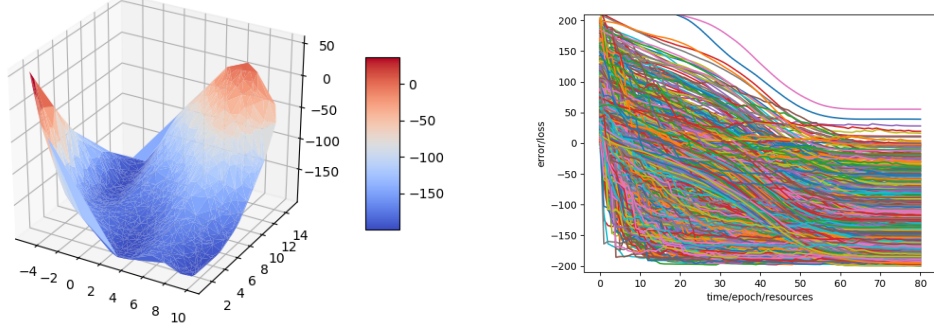
- **Smooth** if ω is true, we smooth the generated function with a *Savitzky-Golay filter* which is a parametric convolution. This filter is characterized by window size w and order of polynomial o constrained by $o < w - 1$ and w - odd. Through a linear regression, we set:

$$o = 3 \text{ and } w = \lfloor 0.17 \cdot n + 6 \rfloor \text{ or } \lfloor 0.17 \cdot n + 6 \rfloor + 1 \text{ whichever is odd}$$

Simulated loss functions	Description	Parameters
 <p>Dynamic profiles</p> <p>Loss functions</p> <p>Order at ends - static profile</p> <p>1.00</p> <p>colours match clouds colours</p>	<p>Sample loss function simulations from aggressive-start family.</p>	<p><i>u</i>: Branin</p> <p>ml-aggr.: 1.5</p> <p>nec.-aggr.: 10</p> <p>up-spiki.: 5</p> <p>is-smooth: F</p> <p>start-shift: 0</p> <p>end-shift: 200</p>
 <p>Dynamic profiles</p> <p>Loss functions</p> <p>Order at ends - static profile</p> <p>1.00</p> <p>colours match clouds colours</p>	<p>Sample loss function simulations from moderate-aggressive family.</p>	<p><i>u</i>: Branin</p> <p>ml-aggr.: 0.5</p> <p>nec.-aggr.: 7</p> <p>up-spiki.: 3</p> <p>is-smooth: F</p> <p>start-shift: 0</p> <p>end-shift: 200</p>
 <p>Dynamic profiles</p> <p>Loss functions</p> <p>Order at ends - static profile</p> <p>1.00</p> <p>colours match clouds colours</p>	<p>Sample loss function simulations from little-aggressive family.</p>	<p><i>u</i>: Branin</p> <p>ml-aggr.: 0.2</p> <p>nec.-aggr.: 4</p> <p>up-spiki.: 1</p> <p>is-smooth: T</p> <p>start-shift: 0</p> <p>end-shift: 200</p>

Note that a family is characterized by the same profiles described in the Optimizer evaluation criteria chapter.

The surface plot of 1000 simulated loss functions at the end of those 81 resources (maximum time used in these experiments) looks exactly like Branin’s but shifted from -200 (which was indeed expected). Each third of the simulations belongs to each of the families above.



7.3.4 Normally distributed noise

Because simulated functions start from $u(x) - \text{start_shift}$ at time 1 and finish at $u(x) - \text{end_shift}$ at time n , the first halving of Hyperband would be too accurate. That is, Hyperband would pick the true best $\frac{1}{\eta}$ from the sample. The metric that shows this is order-at-ends profile which is always 1. So, to cater for such cases, some initial random noise was added. This requires another model and another parameter.

At time 1 the simulated function has the value:

$$f(1) = u(x) + \text{noise_variance} \cdot \text{noise}$$

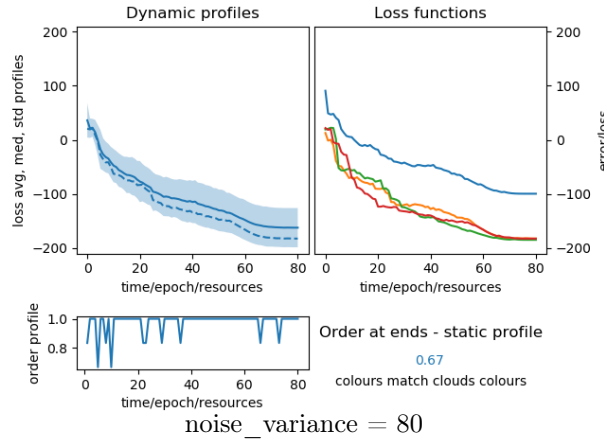
where $\text{noise} \sim \mathcal{N}(0, 1)$ (standard normally distributed)

This is the default way of adding noise.

Finding the parameter noise_variance

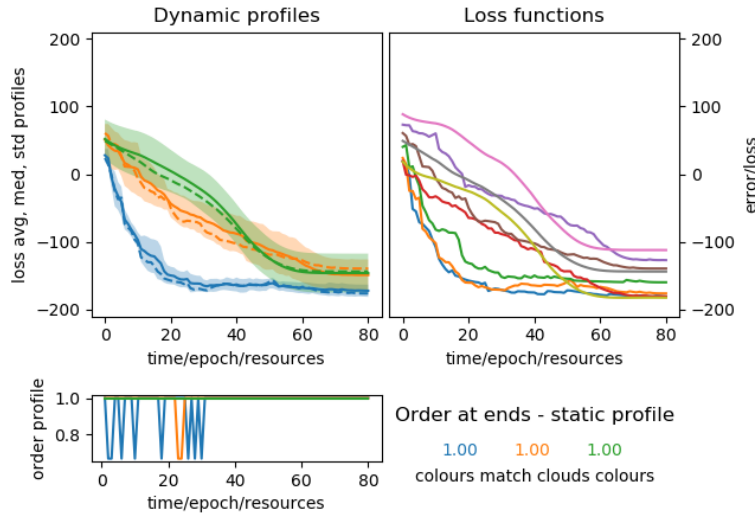
noise_variance is the parameter that directly controls the order-at-ends profile so it can be adjusted accordingly. Note that order-at-ends is not controlled by any other parameter. 0 noise_variance corresponds to no noise, so the order at ends profile will always be 1 in this case.

Note that noise, does not necessarily need to be applied only at time 0. One may choose to apply it at step n (last one) as well, or at every t etc. For comparing Hyperband-based methods we must apply it at least at the first step though because at times $t > 1$ the Gamma process simulation will change the relative order of functions. To verify that this noise model works, one can observe that in the plot below (where noise was applied following the model at time 0), the initial relative order (at time 0) is different from the relative order in the end (at time $n = 80$). Also, as expected, order-at-ends profile is no longer 1.



7.3.5 Scheduling families of shapes

It is clear that it is not enough to have a simulation based on a single family of shapes, therefore, we must be able to put together several families of shapes as part of a single simulation. In our implementation of simulations, after defining the parameters for each family of shapes, one can decide how to *schedule* them. For example, when running an optimization method (that obviously requires several loss functions) the simplest scheduling method is Round Robin - that is, first generated loss function belongs to the first family, the second function from the second family etc. Under this implementation, it is easy to make assumptions over how likely a family is (eg. increasing loss functions are less likely than those that start aggressively). A more advanced scheduler can sample the next family from some distribution.



Mixed samples from the families mentioned above scheduled with Round Robin

If we compute the global profiles (i.e. not by family of shapes) of all families altogether, the median profile is the one that is sensitive to the strategy of scheduling.

7.3.6 Further clarification on model parametrization

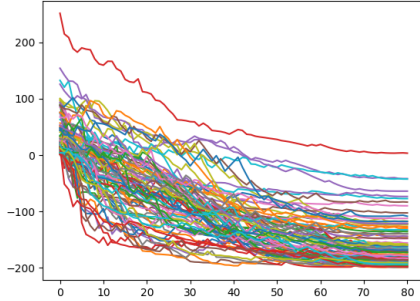
Supposing that ML aggressiveness, necessary aggressiveness and up-spikiness would not have been used. Sampling loss function many times from the pure Gamma process would produce functions that belong to a family of shapes determined wholly by $\mu(x)$ (the function of expected values of each Gamma distribution in the Gamma process). Hence, a Gamma process produces only one family of shapes. Of course, any family of shapes can be described by a different Gamma process. However, we believe that finding the model for each Gamma process while respecting the constraints outlined in the section about the Gamma process is pretty complicated and not intuitive at all. So, we preferred to start from one family of shapes which can be altered in terms of some parameters that have intuitive meaning.

7.4 Previous work

There are many studies involving loss function landscaping. That is, the visualization of existing loss function in conjunction with the vector of hyperparameters associated with them. For example, Studer et al. [20]. LeCun et al. [19] took this even further and established a relationship between a spin glass model and non-convex loss function yielded by deep learning models. There are even Python libraries like `loss-landscapes` that allow visualizing these landscapes. However, we were not able to find any simulation as the one presented above nor something else that addresses the same concerns. We will introduce in the next chapter 8, another method that addresses the same concerns through real-world examples.

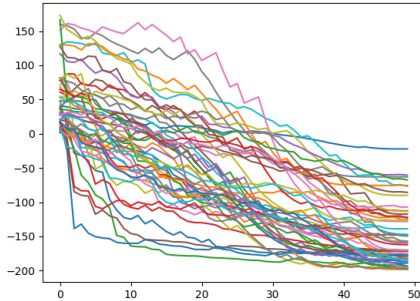
7.5 Optimizers using the simulation

The way we implemented the simulation in our library, called Autotune, allows several optimization methods (all that are built in it) to optimize on the simulated loss functions. Following the same parameters as before, this is how the output of these optimizations looks like:



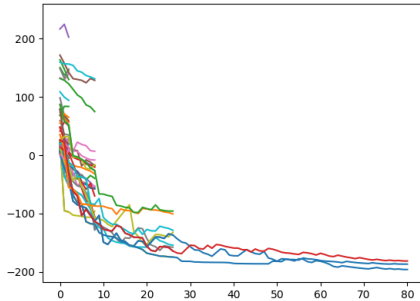
Random search

Random search is the most basic method and we can observe that the functions are covering the given space more uniformly than the informed optimizers.



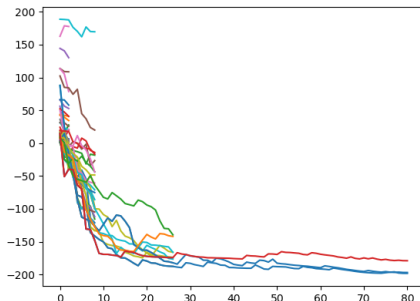
TPE

By contrast with random search, the functions are more concentrated/dense in the lower region than in the upper one which is exactly what an informed optimizer aims to achieve.



Hyperband

Halving allows only the most promising loss functions to continue evaluation. Similarly to random search, functions at time 0 seem uniformly distributed. I find this picture the best summary of how Hyperband operates.



Hybrid Hyperband + TPE

This plot shows that, at time 0 functions seem slightly more tightly coupled in a lower region than in pure Hyperband. This is, indeed, the case, as shown later in EPDF-OFEs evaluations in [9.5.5](#), [9.5.5](#) [9.5.3](#).

As a preliminary form of evaluation, these plots show that the simulations have the right dynamics/mechanics when used in conjunction with optimization methods.

7.6 Evaluation

7.6.1 With respect to the initial requirements

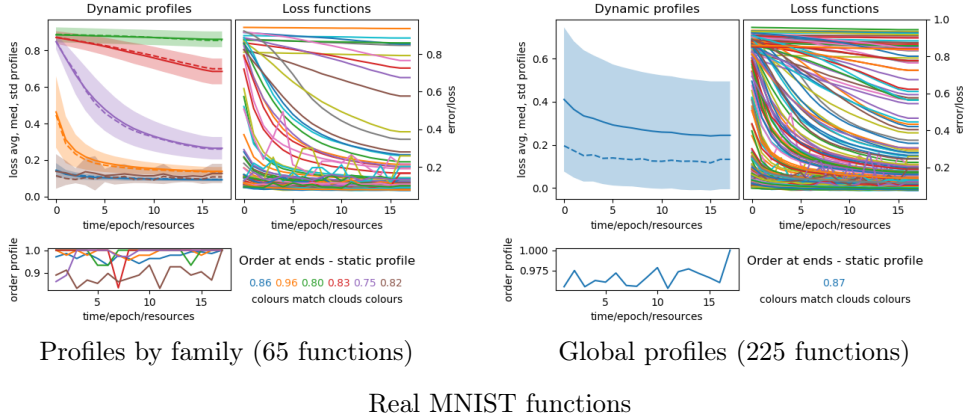
From the above theoretical results plus plots and examples, we conclude that initial requirements 1, 2 and 3 are satisfied. Note that we showed that the simulation is not easy to optimize by neither Hyperband nor TPE in 7.3.1 (Proof - EPDF-OFEs). The easiness to optimize the simulation can be controlled by the underlying functions and by the parametrization of families of shapes.

The simulation implementation in our "Autotune" project allows to generate loss functions with some randomness but also to deterministically re-create any of those loss functions as often as needed.

So, all initial requirements are satisfied. This was indeed expected since the design of the simulation was made such that the initial requirements are satisfied. What is more important is to see the extent to which the Gamma simulation addresses the concerns outlined in the motivation section 7.1. So, in the following experiments, we will show that this simulation also solves them fully or largely.

7.6.2 Reproducing a real-world problem

In short, we try to overfit and reproduce a real-world problem through this simulation mechanism. We took a logistic regression model which is used to classify MNIST data. Firstly, we collected several loss functions and manually labelled the families of shapes that they belong to. Next, we reproduced those families through the simulation.



As a quick reminder, the continuous line is the average profile, the dashed line is the median and the cloud is the standard deviation. Numerically and visually, the profiles are similar, which, from our point of view is enough to show that the simulation has the capability of representing real situations well enough. Note that an initial noise of 10 was added and that the ends of each family are no longer Branin and Branin-200. Instead, they are scaled accordingly based on the real profiles

data. Note that in the simulated global profile the median profile is slightly higher than in reality because the scheduling strategy was Round Robin, hence all families are given the same weight.

EPDF-OFE simulation evaluation

A robust benchmark of the simulation is to check for known hierarchies between optimizers. For example, checking if Hyperband is better than TPE using EPDF-OFEs. We confirm that using the simulation that reproduces MNIST Hyperband is better than TPE with statistical significance. Just as for the real problem.

Result preservation with all underlying functions

Remember that for the families of shapes we used Branin as the underlying function. However, the same evaluation results as above are maintained for Rastrigin and Drop-wave underlying functions as well. In the parametrization of families of shapes the only thing that changes is the underlying function.

7.6.3 Time complexity

Over several trials, Gamma simulation took no more than **0.04 seconds** to generate a loss function with 81 epochs. In terms of time complexity, the proposed simulation algorithm is *linear with respect to the maximum number of epochs* of the simulated loss functions. This is indeed verified:

max epochs	81	243	405	667	5000	10000	15000
time to generate	0.040s	0.043s	0.045s	0.047s	0.090s	0.140s	0.205

Furthermore, our Gamma simulation implementation also caches the values of a generated loss function. That is, once a function is generated our measurements show that the access time to the cached function is no more than *0.002 seconds* (hence with an order of magnitude faster). Caching is important since optimizers like Hyperband access the same function many times in different X points to check for relative performance. This is an important evaluation metric of the simulation, since the generation of a single loss function on a true machine learning model can take hours. For example, generating a single loss function on CIFAR10 with a convolutional neural network takes more than one and a half hours. So this is why we affirm that the simulation is virtually instant. This makes the simulation practical for testing and for computing statistics about optimizer performance as well.

7.6.4 Catching errors (for testing)

Remember the errors that we outlined in the chapter about testing (chapter 6):

- Virtually all implementations of Hyperband are flawed because of floating point errors.
- The design of the first Hyperband-TPE hybrid was breaking one of its assumptions - the budget constraint.

Both of them could be caught when running the optimizers on the Gamma simulations regardless of underlying functions or parametrization of families of shapes. So, again, the Gamma simulation is able to reproduce a real problem.

7.6.5 Features and benefits

- **Instant evaluation** This simulation solves the concerns that running an optimizer on real machine learning models is too time-consuming. The purposes can vary from testing to computing statistics about the performance of an optimizer. For example, it helps comparing early stopping methods, from which Hyperband hybrids are of great interest to this paper without having to deal with the complexity and long times of actual machine learning. This is clearly valuable for finding the best Hyperband-TPE hybrid because we can compare architectures by EPDF-OFEs obtained with the Gamma simulation.
- **Not easy to optimize** It simulates well the change in order of loss functions and offers a good landscape for final errors which are based on an underlying function (eg. Branin). So, if the simulation is fed to an optimizer, it will not be trivial to find its minimum.

- **Flexible intuitive parameters** This simulation method is very flexible in terms of the underlying problem. Even though it is quite parametric, only the noise variance and the noise model depends on the problem. The other 3 parameters only depend only on the maximum amount of resources and start/end shifts. Furthermore, the parameters are very intuitive and can simulate virtually any flat, curved or "S" shaped (convex and non-convex if smoothed) loss functions all with different levels of spikiness.
- **Testing** It is guaranteed that this simulation can be successfully used as a testing method for any optimizer.

Obviously, by design this simulation clarifies and ensures correctness on early stopping methods. Another benefit, is that it can be used with Bayesian methods as well if noise is applied or (riskier) by disabling necessary aggressiveness.

In practice, simulations might also help people to choose the right method of optimizing parameters on unknown problems (in the end, ML in practice is applied to non-academic datasets like MNIST and CIFAR). Take finance for example, researchers write some ML model on some data (not well known dataset) and they want to optimize hyperparameters but they don't have enough time to optimize their model with Bayesian Optimization, Genetic Algorithms, hybrids combining early stopping methods (eg. Hyperband) and wait to see which gives the best results. Instead, they can simulate what they think their problem looks like to see which is best and opt for that one. Of course, there is no guarantee.

7.6.6 What can be improved

Potential areas of improvement that we are currently aware of are:

- In this state there is quite a lot of linearity in models. Maybe, it is worth experimenting with less linear models.
- ML aggressiveness, necessary aggressiveness and up-spikiness depend on `max_resources`. Moreover, they seem vulnerable to extremely large `max_resources` (like 5000). Fortunately, this was foreseen and, in this case, one would need to tune not only the parameters but also the linear model. This goes back to 1.
- There might be some families of shapes which are not representable.
- The early convergence depends fully on the assumptions made when the families of shapes are made. Conversely, final error is very well modelled through the underlying function.

Future work The next step in the evaluation of the Gamma simulation is validation through a method that "explores the output's diversity of a model" [23] like OpenMOLE's Pattern Space Exploration (PSE) [23]. PSE would cover the space of potential loss function and will signal if there are any families of shapes that are not currently representable. So, PSE will greatly increase the precision and improve the way the Gamma simulation is currently calibrated in. What is remarkable about PSE is that it addresses all the areas of improvement enumerated above.

7.7 Summary and achievements (so far)

In this chapter we built a robust method for optimizer testing such that issues like those presented in the testing chapter will be easier to catch in the future. Therefore, testing as a method is considered fully resolved. Moreover, the Gamma simulation is useful to compute statistics which are needed (for EPDF-OFEs for example) in order to reliably compare optimizers. So, it is part of the solution to the problem described in the Optimizer evaluation criteria chapter.

Chapter 8

Closest known loss function approximation

Next, we will introduce another simulation method that is based on existing loss functions computed for given datasets, machine learning models and ranges/domains of hyperparameters.

8.1 Motivation

The main motivation behind this second type of simulation is mostly the same as for the Gamma simulation:

1. Statistical comparison of early stopping methods (eg. Hyperband-based hybrids)
2. Architecting optimizers
3. Timely optimizer selection

The fact that the Gamma simulation is so flexible can be seen both as an advantage and as a disadvantage. To address concerns that the Gamma simulation might not be representative enough for real world cases, we developed this second method.

8.2 Solution

8.2.1 Data collection

The first step of Closest-known-function approximation is collecting data (complete loss functions) for a given machine learning model on a given dataset. By complete loss function we mean that they have a representative length (convergence is clearly observable) and that they have not been "trimmed" by any early stopping method. Of course, this length can vary but, it is important that all loss functions have the same representative length.

The more loss functions are collected the better the results of the approximation. Because we are interested in finding the best loss function, it is important to have some density of functions around the optimal one.

Example: Logistic Regression on MNIST In our experiments, we chose to run this experiment on MNIST dataset with a Logistic Regression model on which we tune:

Hyperparameter	Domain
Learning rate	$[10^{-6}, 1]$
Weight decay	$[10^{-6}, 10^{-1}]$
Momentum	$[0.3, 0.999]$
Batch size	$[20, 2000]$

The data was collected with 195 functions were collected with a Random Search Optimizer and other 230 were collected with a TPE Optimizer. For our purposes, these optimizers yield complete loss functions and provide enough exploration and exploitation/density around the optimal function.

8.2.2 Closest known function

Let us name a combination of hyperparameters an *arm*. An example is:

`arm = {"learning_rate": 1, "batch_size": 100, ...}` for hyperparameters defined in the table above. A *normalized arm* is an arm where the values of each hyperparameter is normalized according to its domain. For example:

`normalize(arm) = {"learning_rate": 1, "batch_size": 0.04, ...}`.

Normalization of each hyperparameter is done as usual: $\frac{x - \min(A)}{\max(A) - \min(A)}$ if its domain is $x \in A$.

The last thing to revisit is that every loss function corresponds to an arm and, consequently, to a normalized arm.

Given:

- a *proposed arm* which is an arm proposed by some Optimizer (eg. TPE) as the next combination of hyperparameters to evaluate.
- n complete loss functions f_1, f_2, \dots, f_n each based on a different arm $arm_1, arm_2, \dots, arm_n$ where arm_i gives loss function f_i .

Let us, now, formally define an arm as a mathematical function from the set of hyperparameter names \mathbb{M} to real numbers \mathbb{R} :

$arm_i : \mathbb{M} \rightarrow \mathbb{R}$ for example $arm("learning_rate") = 1$ Note that \mathbb{M} is the same for all arms. $normalize : (\mathbb{M} \rightarrow \mathbb{R}) \rightarrow (\mathbb{M} \rightarrow [0, 1])$ takes an arm and normalizes all its hyperparameters. $normalize(arm_i)(m) = normalized(arm_i(m))$ where *normalized* means normalizing hyperparameter named m with the value taken from arm_i with respect to its domain. *normalize* is a transform that can be curried (in computer-science terms).

Return: For the proposed arm we return the loss function f_i whose normalized arm, $normalize(arm_i)$ is **closest** to $normalize(proposed_arm)$. This metric of closeness is the **mean square error** across hyperparameters. Note that categorical (including boolean) hyperparameters are included in this definition by being converted to numbers and then normalized (eg. True is 1, False is 0).

Let the mean square errors between 2 arms be:

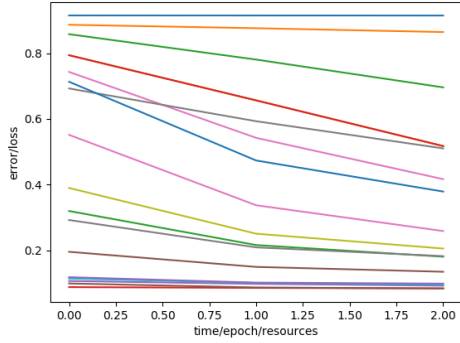
$$MSE(arm_i, arm_j) = \frac{1}{n} \cdot \sum_{m \in \mathbb{M}} normalize(arm_i)(m) - normalize(arm_j)(m)$$

This approximation will return for the proposed arm, the loss function whose arm is closest to the proposed arm in terms of *MSE*. That is:

for *proposed_arm* return f_i where $i = \operatorname{argmin}_{i \in \{1..n\}} MSE(arm_i, proposed_arm)$

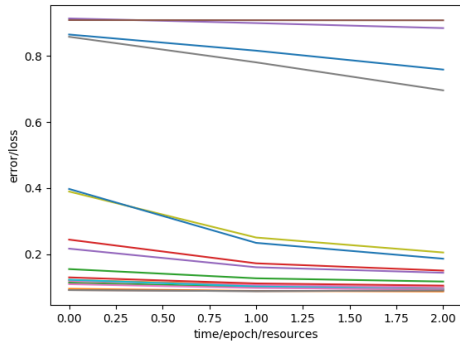
8.2.3 Optimizers using the approximation

This approximation/simulation implementation in Autotune works with several optimizers (all that are built in it) as well. By putting this MNIST+Logistic Regression approximation through them we get the following samples:



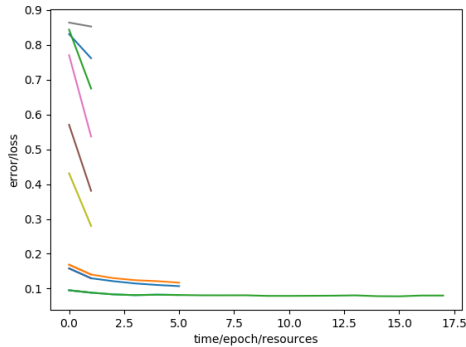
Random search

Random search is the most basic method and we can observe that the functions are covering quite uniformly even the worse part of the search space.



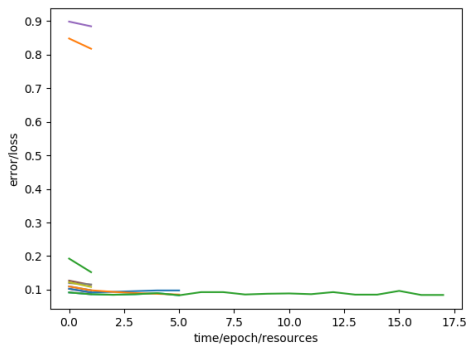
TPE

By contrast with random search, the functions are more concentrated/dense in the lower region than in the upper one which is exactly what an informed optimizer aims to achieve.



Hyperband

Halving allows only the most promising loss functions to continue evaluation. Similarly to random search, functions at time 0 seem uniformly distributed.



Hybrid Hyperband + TPE

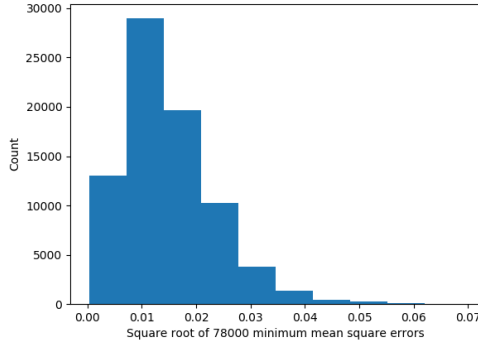
This plot shows that, at time 0 functions seem slightly more tightly coupled in a lower region than in pure Hyperband.

These plots show that the approximations have the right dynamics/mechanics when used in conjunction with optimization methods.

8.3 Evaluation

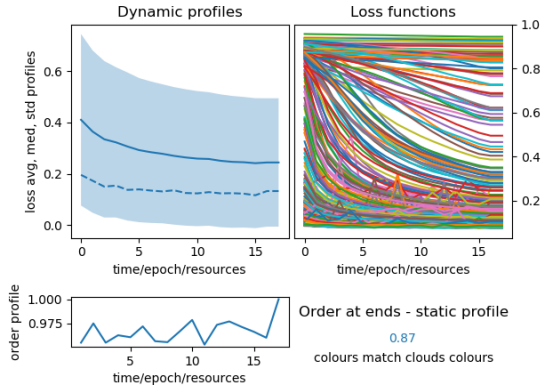
8.3.1 Low approximation errors

To measure this approximation, we used the square root of the minimum mean square errors. That is, the square root of those MSE -s that correspond to arms that have been picked for the proposed arm. The square root was used to bring the values back to the same order of magnitude (just like in variance-standard deviation). This allows us to report the "distance" between the proposed and the selected arm, and implicitly, between loss function that was used and what could have been used by the real machine learning algorithm. In the histogram below, we plotted the count of $\sqrt{\min(MSE)}$ from 78000 samples.



confidence	error rate
99%	less than 4% error
78%	less than 2% error
30%	less than 1% error

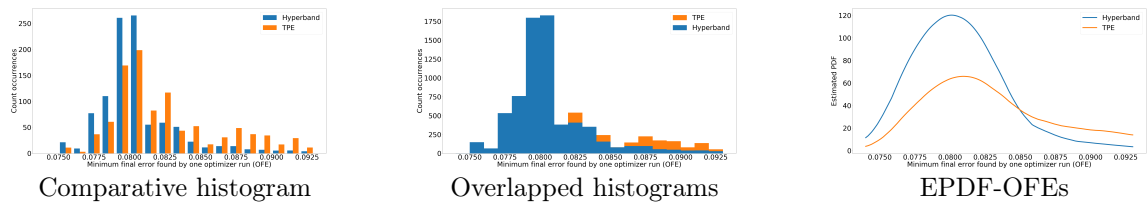
From the histogram and metrics above, we can safely tell that the errors are small enough to make this experiment a good and fast simulation. Below, the median profile (dashed line) is much lower than the average profile (continuous line) which reinforces the fact that the functions are denser in the "better" region. This was indeed expected. At the time of data collection, for more than half of the samples, we used TPE which gave more "better" loss functions than "worse" ones and random search yielded several good loss functions including the best ones.



So, the space of arms/loss functions is not uniformly covered. This is good because Hyperband discards bad functions early (so more bad functions would not add much value) and also because Logistic regression on MNIST does not have high levels of non-convexity (i.e. MNIST is not a very difficult learning problem) so any optimizer will find the best region quite fast. The question is whether or not it can find the best. Returning to errors, we must note that the higher errors are due to the fact that bad regions are not as well represented as good regions.

8.3.2 EPDF-OFE experiment evaluation

The EPDF-OFEs below, have been calculated on the MNIST Logistic Regression approximation experiment.



The optimization problem is non-trivial

We appreciate that this approximation is well suited for computing statistics about optimizers, since state-of-the-art optimizers do not always find the true best (few) loss functions. The distributions (EPDF-OFEs) above show that the true available optimums are not hit very frequently. This means that the approximation contains enough data and is relevant for the problem.

Matching and reinforcing published papers/results

Moreover, the results published in the original Hyperband paper [13] - Hyperband is much better than TPE - are perfectly matched by this experiment. In addition, this EPDF-OFE evaluation method enriches the scope of the original paper and shows that this difference is statistically significant. Furthermore, our results also match the ones published in [21] and [17] but this will be detailed later in 9.5.2.

Matching our real ML experiments

Later, we will run experiments both on the approximation and on the true machine learning model. The results in both cases are matching very well. The exact comparisons are made in: 9.5.2.

8.3.3 Time complexity

Over several runs the average time to get a loss function for a proposed arm is **0.0065 seconds** which is even faster than the time to generate a loss function with the Gamma simulation. Theoretically, the time complexity of the algorithm is *linear in terms of the number of known loss functions* (i.e. the number of loss functions in the database). In practice, this number is finite. For example, 0.0065 seconds were required for 425 known loss functions. From here we can easily estimate the amount of time the algorithms takes if given N known loss functions. Assuming, 0 seconds when $N = 0$, the estimated access times would be:

N known functions	425	1000	5000	10000
time to generate	0.0065s	0.015s	0.076s	0.15s

This proves the claim that the closest known loss function approximation is also virtually instant. Hence, this is another solution (in addition to the Gamma simulation) to the problem that running an optimizer on a true machine learning model is time consuming.

8.3.4 Features and benefits

The most important aspect about this approximation is that it can evaluate any optimizer in **negligible time** with true **statistical significance** and with high precision. In addition, it is based on real-world data. As we have seen in the Evaluation section, it provides a good estimate of how the real machine learning model would perform. Furthermore, it allows massive speed-ups when computing statistics for optimizers. So, closest known loss function approximation opens the way towards computing what we labelled "ideal evaluation" (i.e. profiles of multi-model multi-dataset EPDF-OFEs) which was described in 5.3.3.

8.3.5 What can be improved

This simulation is dependent on the underlying dataset and on the choice of machine learning model which can both vary enormously. We can think of the space of datasets as potentially infinite. So, the main issue is that this approximation will have enough data for popular datasets like MNIST, CIFAR, SVHN etc. These constitute good indicators but we anticipate that industry-specific datasets will lack the same support. By contrast, the Gamma simulation is a lot more flexible and transferable. Another key difference between Gamma simulation and Closest known loss function approximation is that the first is reliable for optimizer testing while the latter is not since multiple arms can yield the same loss function.

Future work The next step is the collection of loss functions for other classical datasets like CIFAR-10, SVHN and MRBI.

8.4 Motivation for MNIST Logistic Regression

Logistic regression on MNIST was chosen because collecting enough loss functions (enough to reach low approximation errors) could be done in a reasonable number of days. Collecting loss functions for a more difficult dataset (eg. CIFAR) would be too time consuming given the time frame. To take this project forward the next step is the collection of loss function for CIFAR, SVHN and MRBI datasets.

8.5 Summary and achievements (so far)

In this chapter, we built another solution to the problem of computing statistics on optimizers in a timely manner. The closest known loss function approximation has the advantage that it is based on real data and that its accuracy can be measured quantitatively.

Chapter 9

Best Hyperband-Bayesian hybrid architecture

9.1 Architecture of pure optimizers

9.1.1 Hyperband

Hyperband, as presented in the original paper [13] is an early stopping optimizer based on Random Search. The main assumption it makes is that the relative order of loss functions does not change drastically. That is, it assumes a loss function that is performing very poorly at the beginning will not eventually become the best and the other way round.

In practice, Hyperband maintains a fixed budget of resources R : it starts computing the first few steps of several loss functions (hence, of several arms) and then, it halves with a rate η keeping only the best $\frac{1}{\eta}$ of them. When halving is done, the "survivors" will be allocated more resources (eg. more time, more training examples) to keep the budget fixed. The only problem with halving is that halving too early or too late might impact different families of shapes. For example, when the learning rate is small the loss function will converge slower at the beginning when compared to a high learning rate. However, in the end the small learning rate yields a better result. So, to cater for such cases, Hyperband uses a hedging loop that tries different intervals of halving. Each iteration of the hedging loop is called a *bracket*. That is, the first brackets start with high populations and half early while the last ones start with a low population but halving is done later. This way the budget is constant in every bracket. In the end, it picks the best "survivor" from each bracket.

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

Hyperband for $R = 81$ and $\eta = 3$

A simple illustration of a run of Hyperband is depicted above. It is easy to see that most of this algorithm revolves around halving. It is only at the beginning of each bracket when new arms are generated. After arms generation, selective halving is applied with different frequencies (depending on the bracket) and outputs a survivor which is highlighted above in cyan.

Note on architecture: In the original Hyperband paper [13], the generation is done through random search. If one wants to generate the arms of a bracket with a more informed optimizer (eg. Bayesian) *there is a single place where this optimizer can be plugged in Hyperband* which is the generation phase of each bracket.

Note on scalability: In the current form, Hyperband is fully parallelizable. That is, each bracket can run in parallel and within each bracket individual can run in parallel (because random search optimizer is also fully parallelizable).

9.1.2 Bayesian optimizers

Remember that Bayesian methods aim to find the optimum of a function which in case of machine learning is expensive to evaluate in one point, in as few evaluations as possible. In summary, Bayesian optimizers build a probabilistic surrogate model which is cheap to evaluate (called acquisition function) based on some previous evaluations of the true objective function. The point that maximizes the acquisition function is the proposed arm where to evaluate the true function next, after the evaluation of the true objective function finishes the history is updated and the process continues. TPE is closely following this pattern.

Note on architecture: From an architectural point of view, an iteration of a Bayesian optimizer can be resumed as follows:

1. **Input:** takes some historical evaluations of the true objective function
2. **Output:** proposes an arm where to evaluate the true objective function next, which maximizes the acquisition function


If there is no history, the first few steps are done by random search.

Note on scalability: Bayesian optimizers are inherently sequential. Passing the history of evaluations to the next iteration implies that the previous evaluations must have finished. Therefore, when designing hybrids one must be aware that the more history is being transferred the less parallelizable the optimizer.

9.2 Architecture of hybrids

It has been shown above, that there is a single place where to plug the Bayesian optimizer within Hyperband: the Bayesian optimizer would generate the initial population of each bracket. Note that by doing so, the individuals in every bracket are no longer parallel in the generation phase. Afterwards, during halving, they do become parallelizable again.

Amount of history to be transferred If brackets are executed sequentially, in the first bracket of Hyperband, there are no previous evaluations of the true objective function, so the Bayesian optimizer must start with random search. However, at the beginning of the second bracket one might choose to start with a fresh Bayesian optimizer (which performs random search at the beginning) or might choose to inject some history from the previous bracket. So, when designing a hybrid it is very important to figure out which evaluations of the true objective function (if any) should be transferred from the previous bracket(s) to the bracket that is currently in the generation phase.



Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

Upper bound on transferable history

Assume that bracket $s = 4$ completed and bracket $s = 3$ is in generation phase. What evaluations can be injected as history from bracket $s = 4$ to bracket $s = 3$? Clearly, as bracket $s = 3$ allows 3 initial resources, the injected evaluations from bracket $s = 4$ must have run for at least 3 resources (at most 27 individuals).

The individuals that were given at most 1 resource in the first bracket would not be part of the same true objective function that a Bayesian optimizer in bracket $s = 3$ assumes. The main idea is that transferable history from previous brackets should be representative for the true objective function that is assumed in the current bracket. Therefore, this is an upper bound on transferable history.

Let us explain this further. A hybrid Hyperband+X where X is a Bayesian optimizer. Each bracket uses a X optimizer in its generation phase which builds sequentially a surrogate model of an objective function based on the history that it is fed into it. Since loss functions are never flat, the history that each X assumes is not as well informed in terms of the *true* objective function. That is, each bracket allows up to r_0 resources for each generated individual. So, X's history will not be updated with final errors but with errors at time/epoch r_0 . So, the objective function that optimizer X of the first bracket assumes (i.e. objective it builds a surrogate model for) is the function of errors at time/epoch 1. In the second bracket it will assume an objective function made of errors at time/epoch η and so on.

Lower bound on transferable history The lower bound of transferable history is obviously 0. That is, the generation of every bracket is based on a fresh/new Bayesian optimizer which does not receive any bootstrapped history from the previous brackets. However, this lower bound has the property that it is **scalable**. With 0 history transfer each bracket can run in parallel. For any amount of transferred history greater than 0, brackets would need to be executed sequentially.

9.3 Evaluation criteria

To evaluate various architecture choices (for an optimizer) we will look at *statistics on optimums found by several runs of the optimizer*. In the Optimizer evaluation criteria chapter we introduced the notion of EPDF-OFEs. Running an optimizer thousands times on the true machine learning problem would take an extremely long time. So, we will use the Gamma simulation and the Closest known loss function approximation instead.

EPDF-OFE: Estimated density by optimizer We will look at optimizers that employ different strategies/rules for transferring history from the previous to the next brackets. We will put these optimizers through $N = 7000$ simulations/approximations and record the best error found in each of those N times.

Just like we did when we compared Hyperband and TPE to evaluate the closest known loss function approximation, we will compute the EPDF-OFE from a histogram from these N optimums yielded by each and we will use *Epanechnikov kernel smoothing* to estimate the density function from which the optimums have been drawn.

Benchmark against Hyperband We have already seen that Hyperband performs better than TPE. So, this will be our baseline benchmark in our experiments. In other words, we look at the expected improvement in terms of EPDF-OFEs from that of Hyperband for some proposed architectures of Hyperband+TPE hybrids. TPE and TPE 2xBUDGET will sometimes be shown to serve as a comparison baseline. We will denote $optimizer_1 < optimizer_2$ as $optimizer_2$ is better

than $optimizer_1$ with statistical significance in the Kolmogorov-Smirnov sense (5%). Conversely, when one optimizer is *slightly* better than the other, \leq notation will be used instead.

9.4 Proposed hybrid architectures - history transfer schemes

Let us assume that, currently, we are in the generation phase of bracket $s = 2$. So, we need to generate with TPE 9 points and evaluate them on the true objective function each for 9 resources. Below, we propose a few ways of transferring history from previous brackets to current bracket's TPE optimizer.

As a convention, the history that is transferred is highlighted below in cyan while the bracket which is in the generation phase is highlighted in light green.

NONE - No history transferred

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

No transfer means using a new/fresh TPE optimizer (i.e. which has not seen any history) with every bracket. Among the proposed architectures this is the only parallel one.

ALL - Transfer all comparable history

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

This architecture carries all comparable history from previous brackets. Comparable means that the arms have been allocated at least r_0 resources where r_0 is the initial number of resources in the current bracket. This means that those arms that were cut before attaining at least r_0 resources (the really bad examples) are not transferred.

SURV - Transfer survivors

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

Only the previous survivors (best found by previous brackets) are passed. This might distort TPE's view because it would have no bad example. That is, TPE might be biased in favouring exploitation of the parameter space and neglecting exploration.

SAME - Transfer those that run exactly for r_0 resources

Bracket	s=4		s=3		s=2		s=1		s=0		i
Population/Resources	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	
Generation	81	1	27	3	9	9	6	27	5	81	0
Halving	27	3	9	9	3	27	2	81			1
	9	9	3	27	1	81					2
	3	27	1	81							3
	1	81									4

This transfer scheme allows to pass only those arms that were allocated exactly 9 resources. That is, the injected history is made up of the worst arms that were allocated at least at least r_0 resources (the better ones run for more than 9). However, this scheme aims for perfectly comparable history.

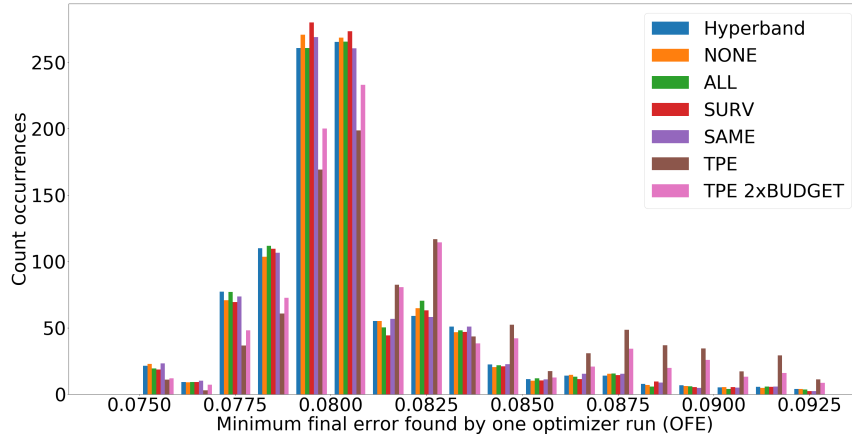
9.5 Evaluation

9.5.1 Closest known loss function approximation

Firstly, we will analyze on real-life data using the Closest known loss function approximation on its staple experiment: Logistic Regression on MNIST. Remember that the main benchmark is Hyperband based on random search as published in [13]. As before, every optimizer was run 7000 times.

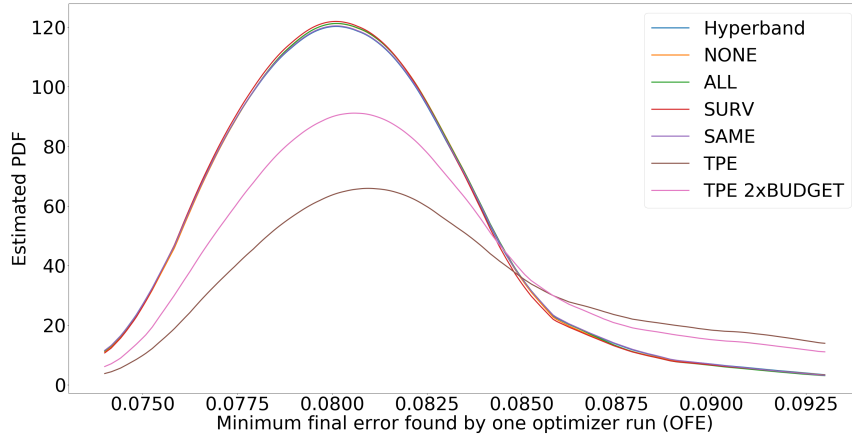
Logistic Regression on MNIST dataset

Histogram



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants

EPDF-OFEs

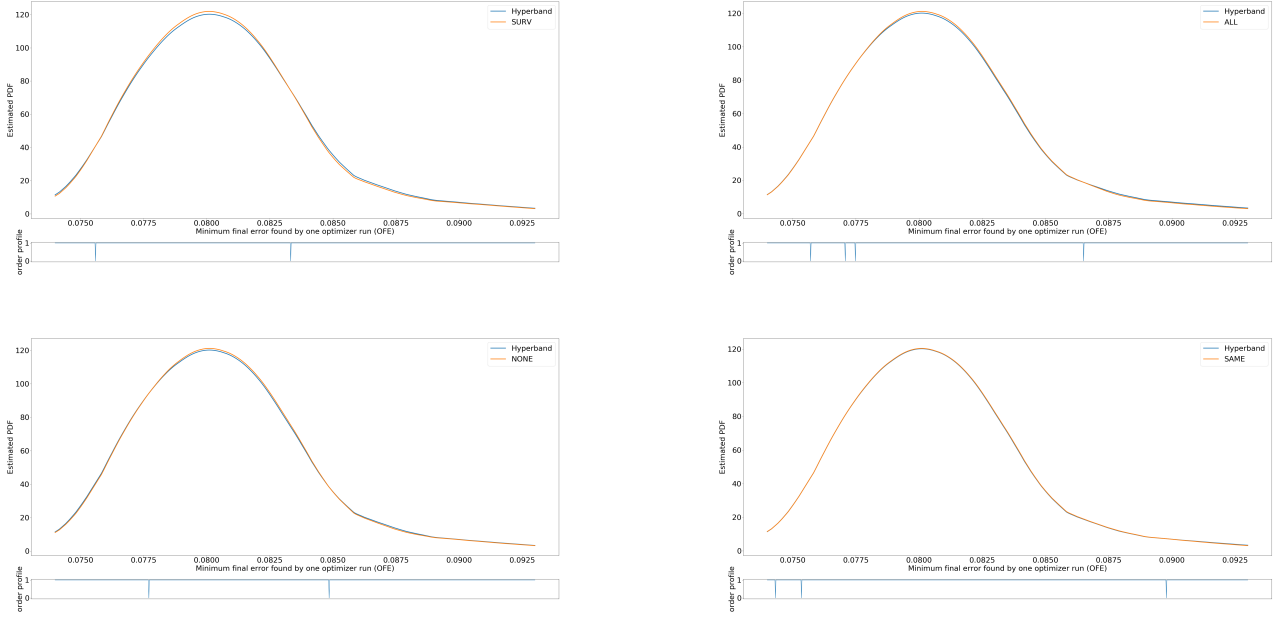


EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants

These densities do not look blatantly show statistical significance (like we have seen between Hyperband and TPE for example). However, there are some observable differences: SURV (red) seems to be slightly better than the others. Therefore, we will investigate each history transfer scheme by comparison with Hyperband. These observations are backed by p-value given by the Kolmogorov-Smirnov test.

EPDF-OFEs	KS p-value	is significant
TPE - TPE 2x	$8.53 \cdot 10^{-114}$	yes
TPE 2x - Hyperband	$9.57 \cdot 10^{-205}$	yes
Hyperband - ALL	0.77	no
ALL - SAME	0.95	no
SAME - NONE	0.66	no
SURV - NONE	0.053	yes

Next, we will apply the **dynamic order** profile on these densities to clearly see the intervals on which Hyperband is outperformed.



The dynamic order profiles show that the interval on which SURV is better than Hyperband is shifted to the left (towards lower errors - better) of the corresponding interval of NONE. From this and from the Kolmogorov-Smirnov results above follows that SURV is the best architecture for this problem, followed by NONE. However, the statistical significance of this hierarchy is questionable. Also, note that Hyperband has a little edge in terms of the absolute best optimums found but it is clearly not statistically significant.

9.5.2 MNIST - real machine learning

Next, we investigated further on the true machine learning model with respect to the evaluation criteria presented earlier.

EPDF-OFEs

Using the data collected from 20 runs for each history transfer scheme, we can use Kolmogorov-Smirnov evaluation to check whether the EPDF-OFEs are different with statistical significance. Based on the approximation experiment, we expect to see some slight (non-statistically significant) hierarchy: $\text{Hyperband} \leq \text{NONE} \leq \text{SURV}$.

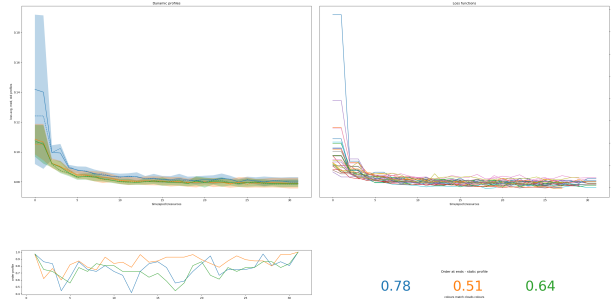
Indeed, the hierarchy in the averages/means of the samples is $\text{Hyperband}(0.0805) \leq \text{NONE}(0.07917) \leq \text{SURV}(0.07911)$. Note that the lower the error the better the optimizer. Also, the lowest error among all optimizers comes from SURV.

EPDF-OFEs	KS p-value	is significant
Hyperband - NONE	0.16	no
None - SURV	0.85	no

These results match perfectly the evaluation from the Closest known function approximation above. Therefore, this can be seen as another validation for the closest known loss function approximation (the rest of the evaluation is in 8.3.2)

Early convergence

Remember that in the "Optimizer evaluation criteria" chapter if 2 the difference between the optimal final errors OFEs of 2 optimizers is not statistically significant, the second criteria is early convergence or lowest area under curve/loss function. Therefore, we run several experiments of true machine learning (no simulation, no approximation), Logistic Regression on MNIST to check for loss functions profiles and to investigate if any optimizer gives consistently a lower area under curve/loss function.



Hyperband in Blue, NONE in Orange, SURV in Green

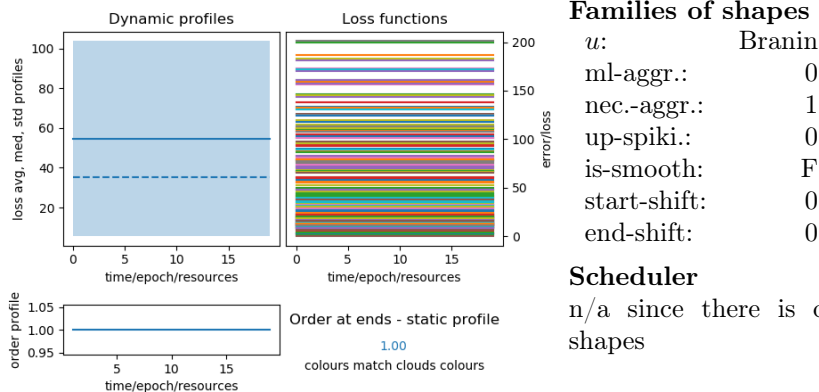
Following this analysis, Hyperband has a visibly higher (worse) area under curve than the hybrids. Numerically, both mean and median profiles are higher than those of the hybrids.

Matching published papers

The fact that the difference in OFEs between Hyperband and NONE is not statistically significant matches the findings of the first published hybrid [17] which analyzes the accuracy on MNIST and on 2 variations: SVHN and MRBI. The fact that NONE converges earlier than Hyperband also matches the findings in [21]. Note that the ML models are different, [17] uses a neural network, [21] uses SVMs while we do Logistic Regression. Note further that, [21] only evaluates the hybrid on MNIST.

9.5.3 Gamma simulation - pure/flat functions

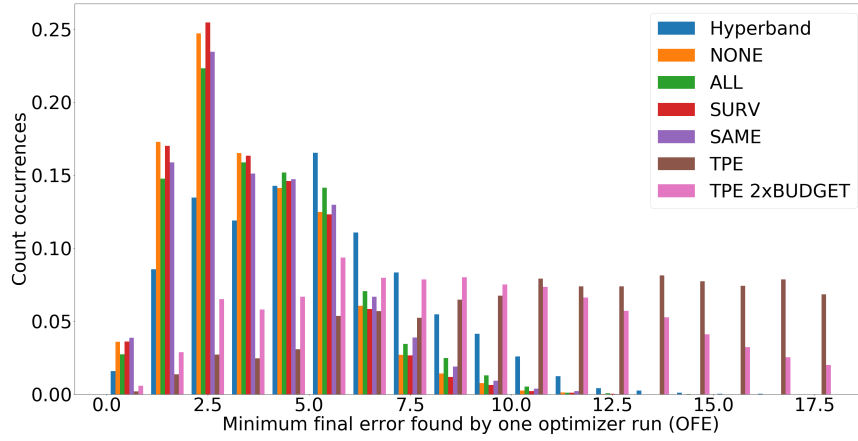
The first few simulations are the pure/flat functions (Rastrigin, Branin, Drop-wave) put through all architectures/transfer schemes. The profiles of these loss functions are very artificial, we do not expect to see something like this on a true machine learning problem. The intention is to have a baseline idea about the relative performance of the transfer schemes. For example for pure/flat Branin function the profiles of the simulated loss functions would be:



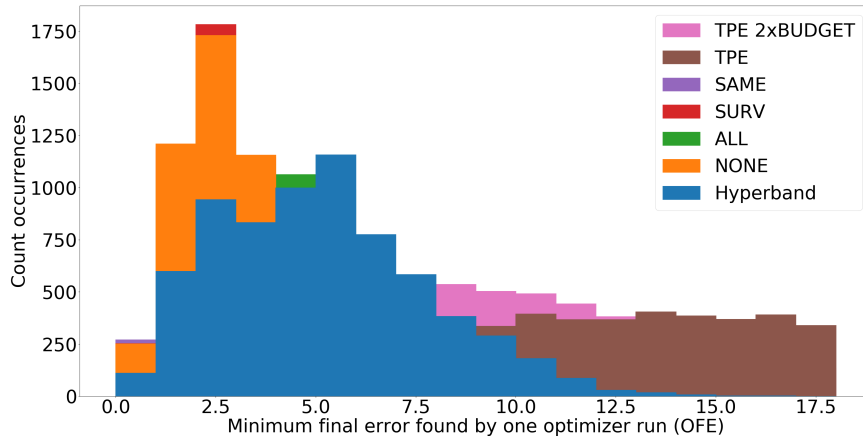
Of course, the profiles of the simulated loss functions would look very similar when Rastrigin and

Drop-wave are used as underlying functions. Note that early stopping plays no role when loss functions are flat. So, the halving phases would be "too accurate" since the loss function at time 0 has the same value as the loss function at time n . Hence, this is more of a study of TPE. A new bracket corresponds to a new run of TPE which receives as history (or not) some of the results from previous runs.

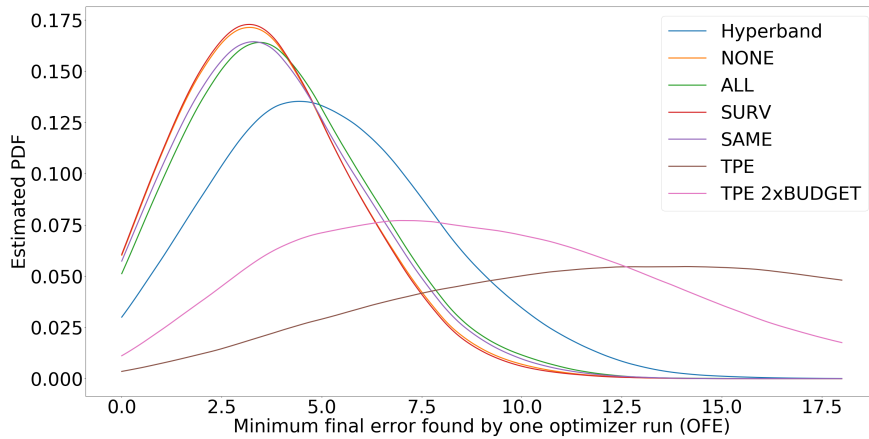
Pure Rastrigin function



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants



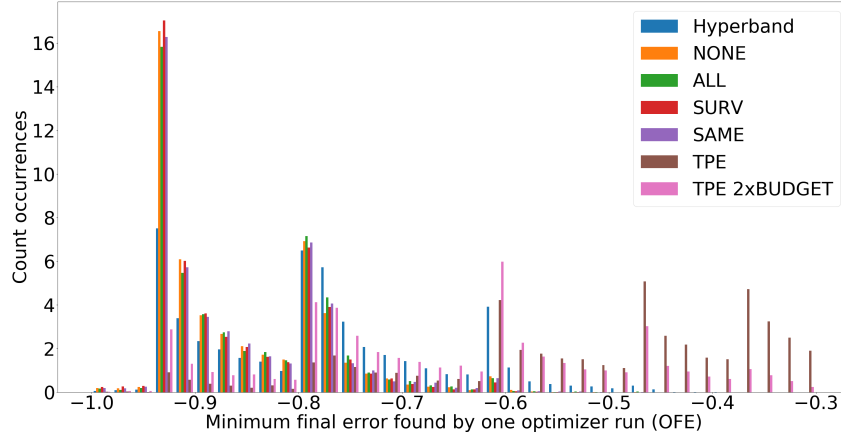
Overlapped histograms



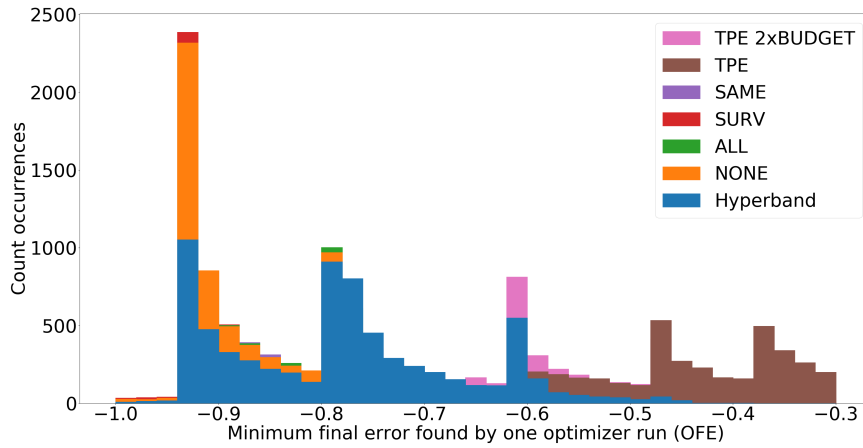
EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants

$\text{TPE} < \text{TPE 2xBUDGET} < \text{Hyperband} < \text{Hybrids}(\text{ALL} \leq \text{SAME} < \text{NONE} \leq \text{SURV})$

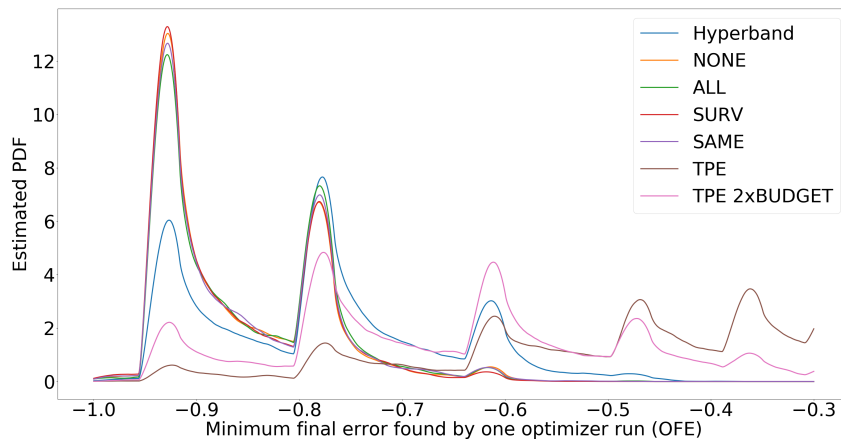
Pure Drop-wave function



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants



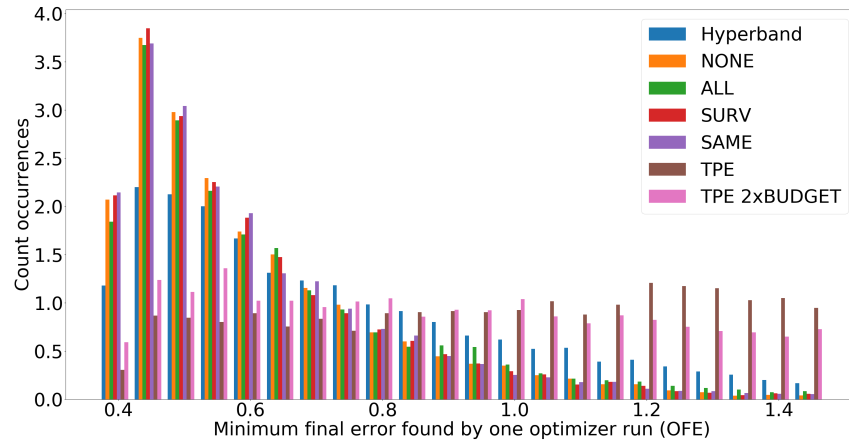
Overlapped histograms



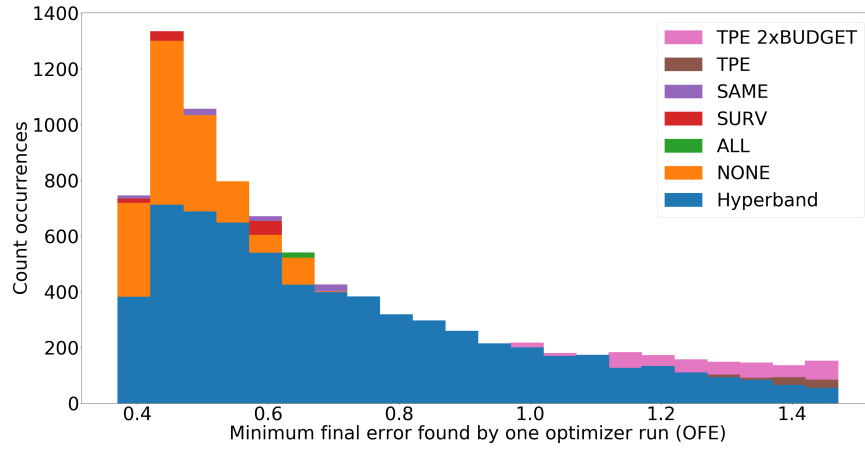
EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants

$\text{TPE} < \text{TPE 2xBUDGET} < \text{Hyperband} < \text{Hybrids}(\text{ALL} \leq \text{SAME} \leq \text{NONE} \leq \text{SURV})$

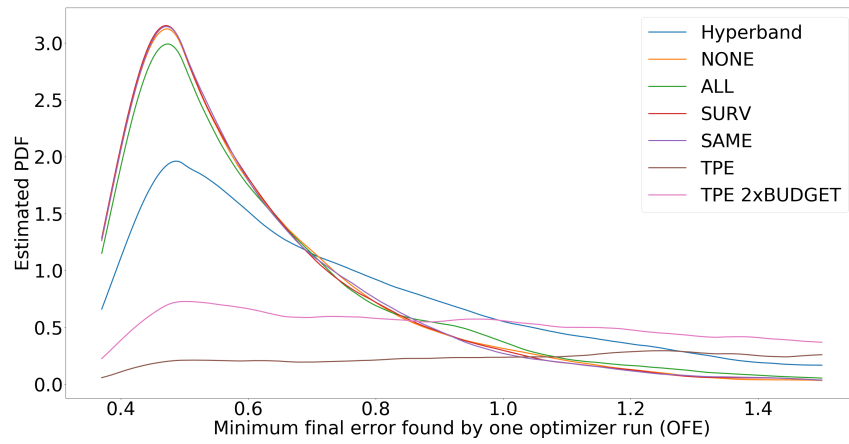
Pure Branin function



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants



Overlapped histograms



EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME, TPE variants

TPE < TPE 2xBUDGET < Hyperband < Hybrids(ALL < SAME ≤ NONE ≤ SURV)

9.5.4 Preliminary results

These flat simulations were obtained by running Hyperband with $R = 81$ and $eta = 3$ and the equivalent resources for TPE (respectively twice as many for TPE 2xBUDGET).

It is very clear from the histograms and, consequently, from the EPDF-OFEs that there is a statistically significant difference between $TPE < TPE\ 2xBUDGET < Hyperband < Hybrids$ (as measured by Kolmogorov-Smirnov p-value) which is preserved across all 3 underlying functions. This result is important since it **makes the pursuit for a hybrid legitimate**.

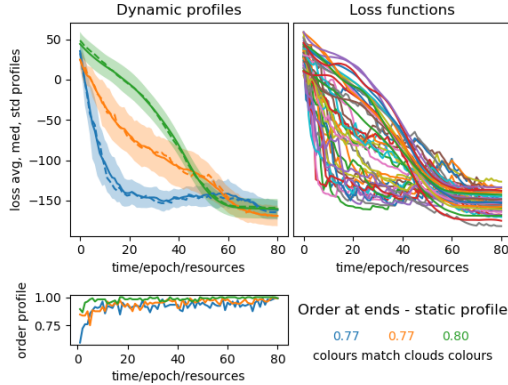
On the other hand, the hierarchy between the hybrid architecture is not that significant. We must note, however, in all 3 cases, $SURV \geq NONE \geq SAME \geq ALL$. This exact ranking has been observed in the closest function approximation as well. **Even though SURV looks slightly better, it is worth noting that NONE is parallelizable while SURV is not.**

9.5.5 Gamma simulation - families of shapes

After having evaluated the performance of Hybrids on flat loss functions, the Gamma simulation will be used again but this time with families of loss function shapes that mimic some real ones.

As the final errors are shifted by a constant from the final errors obtained in the previous flat simulations, TPE and TPE 2xBUDGET will be the same. However, having multiple families of shapes puts early stopping optimizers (Hyperband and Hyperband-based hybrids) in more difficulty. This time, Hyperband's early stopping (halving) mechanism will be used. Therefore, the next experiments involve all parts of the used algorithms just like a true machine learning model would do.

Rastrigin-1 families

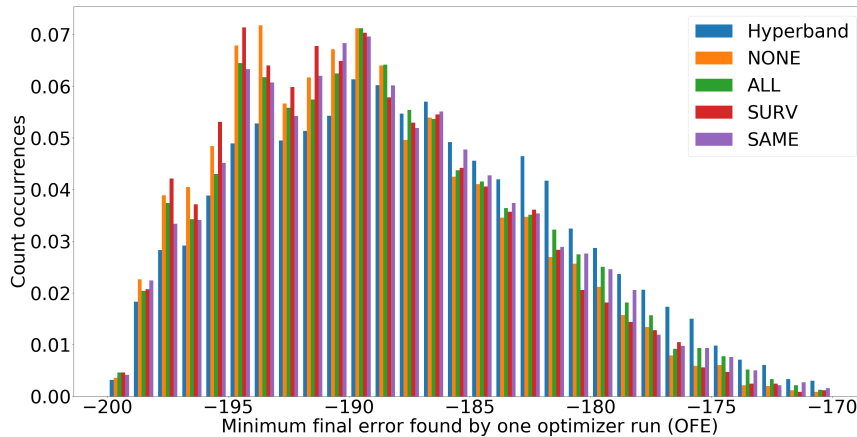


Families of shapes

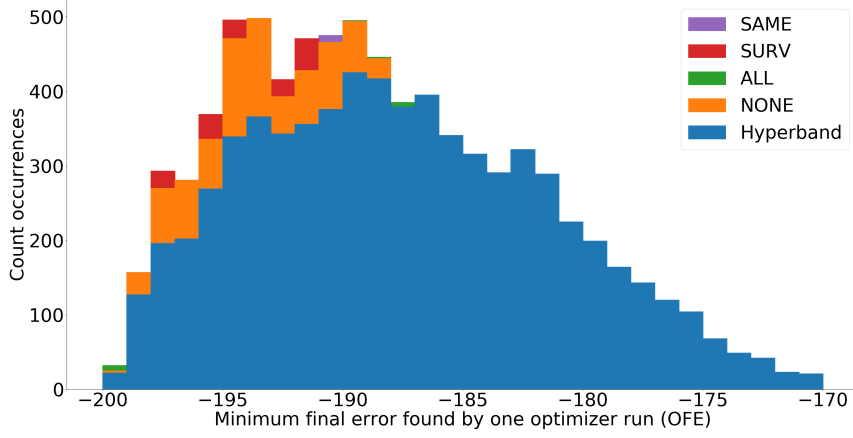
u :	Rastrigin		
ml-aggr.:	1.5	0.5	0.2
nec.-aggr.:	10	7	4
up-spiki.:	15	10	7
is-smooth:	F	F	T
start-shift:	0	0	0
end-shift:	200	200	200
initial noise variance:	10	(noise not scaled)	

Scheduler

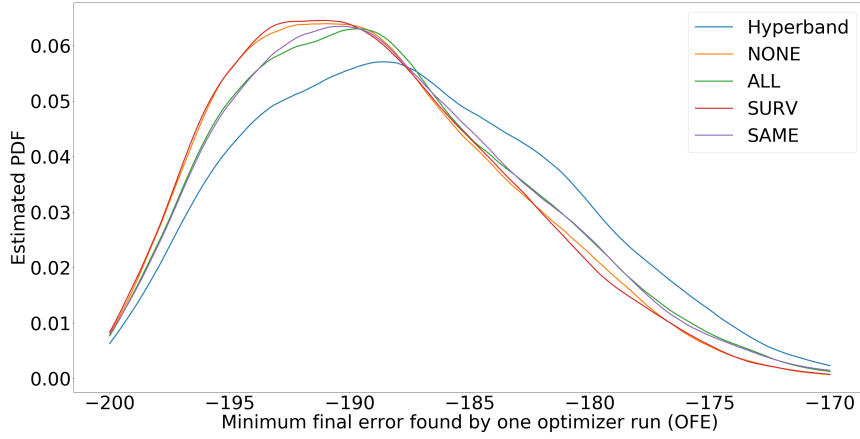
uniform random draw



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME



Overlapped histograms

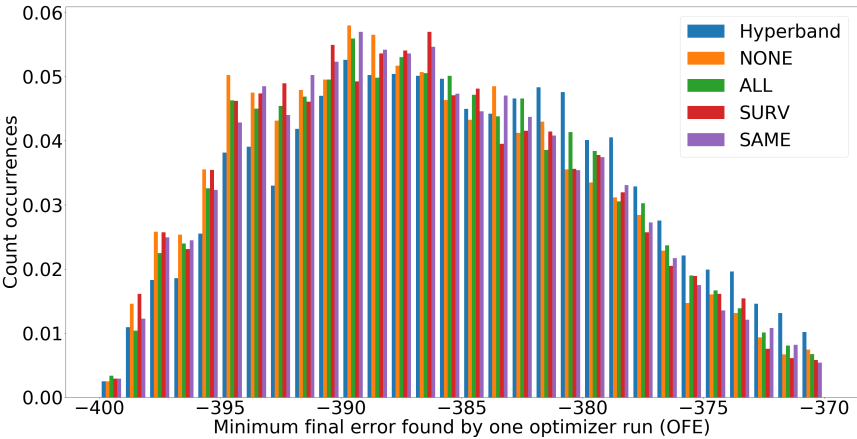
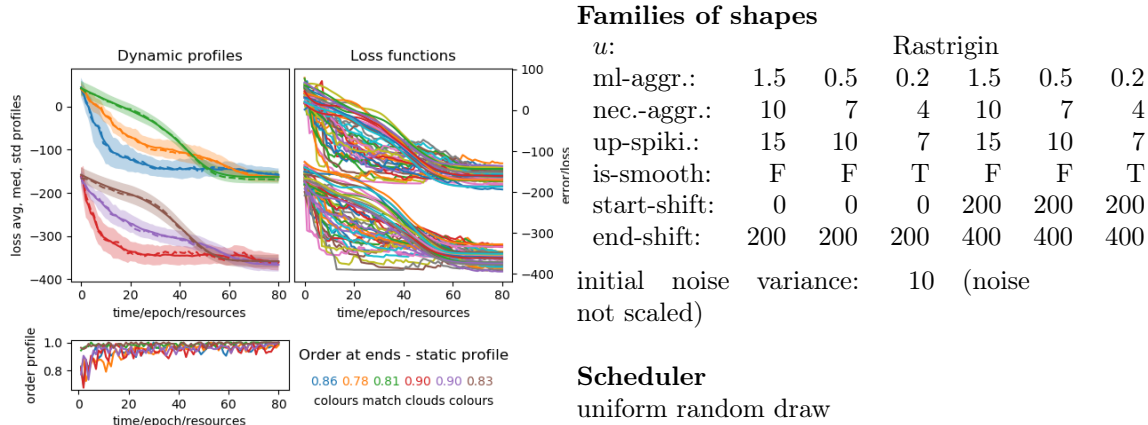


EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME

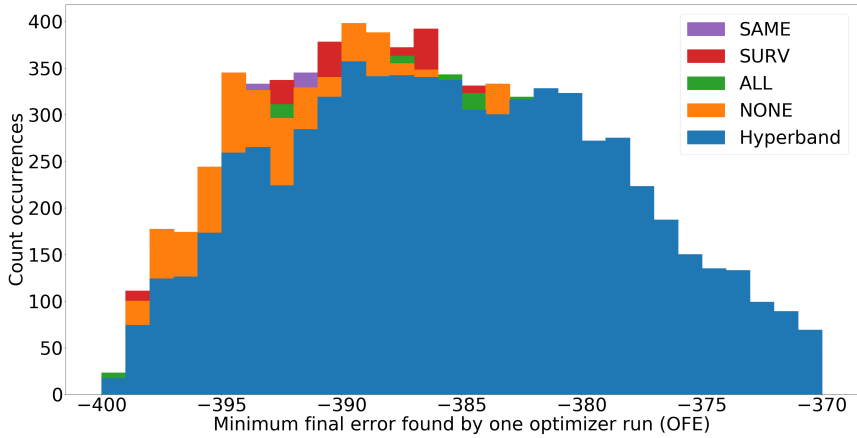
Hyperband < Hybrids(ALL \leq SAME < NONE \leq SURV). As expected, all distributions are slightly "fatter" (than for Pure/Flat Rastrigin) and their means/modes/medians are *shifted to the right* (towards higher optimal final errors) because the halving mechanism filters some of the good loss functions that do not look promising at the beginning. This shows, that early stopping is indeed working. The distributions resemble those obtained from the closest known loss function approximation experiment which reinforce the idea that the Gamma simulation is relevant as an experiment. The most important result is that the hierarchy is maintained.

Note that for this simulation all *end_shift*-s are 200 so the OFEs correspond to $rastrigin(x, y) - 200$. This makes comparison with Pure/Flat Rastrigin clear because the OFEs only differ by a constant.

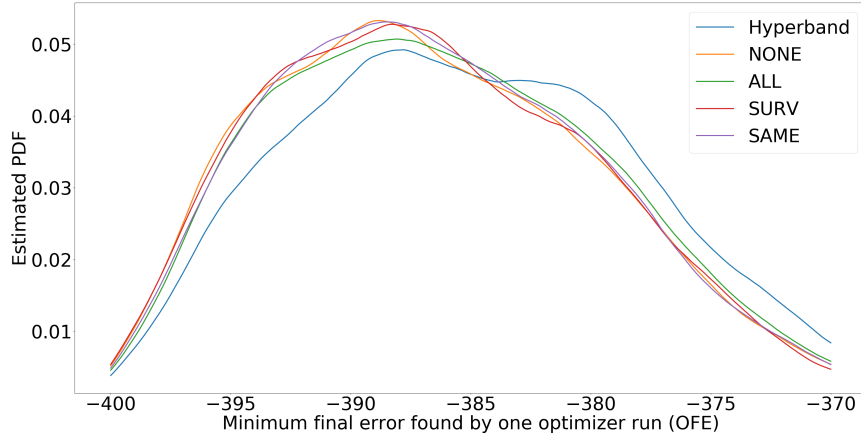
Rastrigin-2 families



Histogram comparison between Hyperband, NONE, ALL, SURV, SAME



Overlapped histograms



EPDF-OFEs comparison between Hyperband, NONE, ALL, SURV, SAME

Hyperband < Hybrids($ALL \leq SAME = NONE = SURV$). Again, as expected, all distributions are slightly "fatter" (than for Rastrigin-1 families) and their means/modes/medians are *shifted to the right* (towards higher optimal final errors) because by the parametrization of families of shapes and the scheduler choice the probability of landing on the best family of shapes is half from what it used to be in Rastrigin-1 families. This shows again, that early stopping is indeed working.

It is important to note that the harder the problem to optimize (i.e. low probability of scheduling the best family of shapes) the closer Hyperband and Hybrids approach. This reinforces the result obtained by closest loss function approximation when Hyperband was statistically significant worse than the Hybrids but not far from them. Also the gaps between hybrids are closing.

9.5.6 CIFAR10 - real machine learning

Finally, we verify that the results above are preserved on a more challenging dataset: CIFAR10. We optimized CNNs on CIFAR10, 10 times with each of Hyperband, NONE and SURV and considered the survivors (from each bracket) found in these runs.

Architecture	Average	Std. Dev.
Hyperband	0.12009	0.0110
NONE	0.1177	0.0077
SURV	0.1148	0.0065

Profiles of OFEs on validation set

On the training set the OFEs are +0.01 larger than on validation.

EPDF-OFEs	KS p-value	is significant
Hyperband - NONE	0.94	no
None - SURV	0.38	no

Kolmogorov Smirnov test

The hierarchy is, yet again, the same Hyperband < Hybrids($NONE \leq SURV$). Note that the very best OFE found is given by SURV: 0.1035 (i.e. 89.65% accuracy) which is a considerable improvement from what has been reported so far in the first hybrid paper [17] or by Hyperband authors in [38, 13]. There is a massive improvement from the method based on Genetic Algorithms proposed in [1], which was run on comparable resources (i.e. not the order of hundreds of GPUs). However, there is still some distance from the state-of-the-art.

9.5.7 What can be improved

The history transfer schemes outlined in this paper are essential for investigating the most efficient hybrid architecture between Hyperband and TPE because they consider they consider the upper bound (ALL), the lower bound (NONE), first (SAME) and last (SURV) quantiles of the amount

of history that could be transferred. However, it is not guaranteed that there cannot be other transfer schemes that would behave better than these four. There might be ways of cherry-picking historical points and transferring them. We consider that the study of more sophisticated history transfer algorithms can yield marginally better architectures.

Future work To further the findings of this chapter, it is worth experimenting with other datasets than the classical/academic CIFAR10 and MNIST as well as with other underlying architectures. One way, would be tackling problems published in machine learning competitions like Kaggle. Since the number of ML models is quite limited, we believe that hyperparameter tuning is what makes the difference between a good model and a great model. Therefore, this study can be extended on such challenges/datasets because they provide good benchmarks.

9.6 Conclusion

Further experiments confirm that the results exposed in Preliminary results 9.5.4 are maintained: the pursuit for a hybrid is legitimate, among hybrids NONE and SURV are the top performers but NONE is the only one that can be parallelized.

Retain NONE history transfer scheme

From the above experiments, we conclude that it is best to opt for NONE history transfer scheme because it is performing very well with respect to Hyperband and even to the other hybrids. SURV has been performing better in several instances but it was never over-performing NONE with statistical significance. The main advantage of NONE is that its brackets are completely independent and hence, fully parallelizable. None of the other history transfer schemes (SURV, ALL, SAME) is parallelizable. Furthermore, we believe it is more prudent to opt for NONE because there might be cases when transferring the very best from previous brackets harms the exploration of the space for the rightmost brackets. Expected improvement might incentivise TPE to exploit around the very best points (survivors) found in the previous brackets instead of allowing for more exploration.

Properties

From our experiments we can conclude following the criteria outlined in [21] that NONE exhibits strong anytime and final performances, it is also parallelizable, scalable and simple.

Hybrids preserve hierarchy

Optimizers (eg. Bayesian ones) can be plugged only in the generation phase of each Hyperband bracket. Technically Hyperband can be split in 2 areas:

1. successive halving over several brackets - nothing in this area can be changed without changing the assumptions of Hyperband
2. generation/suggestion of arms at the beginning of each bracket

Thus, in a more abstract view Hyperband can be seen as a NONE hybrid between successive-halving-over-brackets (1) and random search optimizers for each bracket in its generation phase (2). Now, let us consider hybrids between Hyperband and some type of Bayesian optimizer. It has been shown in seminal papers like [12], [11] and [13] that Bayesian optimization is better than random search. This hierarchy is preserved in the experiments made so far.

Because in the NONE history transfer scheme, generation in a bracket is independent from all other brackets. The overall performance of a bracket is determined in its generation phase. From this follows that if Optimizer X performs better than Optimizer Y, it is expected that a hybrid based on X will be better than a hybrid based on Y.

Chapter 10

Conclusion

Our starting point was a simple remark that Hyperband and TPE are two best-in-class optimizers which exhibit complementary behaviour: Hyperband is parallelizable but is not informed while TPE is informed but not parallelizable. Hence, we pursued the research of the best hybrid architecture that combines the strengths of these two optimizers.

10.1 Problematics

Firstly, we had to ensure that we have all the quantitative means to accurately compare hybrid architectures which we were going to design. Surprisingly, typical optimizer comparison methods from the literature have a few shortcomings: they do not provide statistical soundness and confidence intervals for the hierarchies of optimizers, they are hard to reproduce and do not generalize well over several datasets.

We proposed some quantitative measurements that address the problem above, but they require several runs of an optimizer to compute statistics about its performance which is often extremely time-consuming/impractical.

Furthermore, we had to ensure the correctness of a proposed hybrid. Testing is vastly neglected in this field but is definitely needed. For example, we observed that virtually all implementations of Hyperband are flawed and that the first proposed hybrid breaks one of its fundamental assumptions of not exceeding a fixed budget of resources. Optimizer testing was impractical so far because running on true machine learning models many times is time-consuming.

Back to the pursuit of a hybrid, the last problem to be tackled was the extent to which Hyperband can be made informed using TPE. This drove the study of several hybrid architectures.

10.2 Contributions

Here is a short summary of the contributions of this thesis ¹:

1. Provided *more comprehensive quantitative metrics* of optimizer evaluation/comparison.
2. Elaborated a method of *simulating* the behaviour of an ML model in negligible time.
3. Created a method of *approximating* the behaviour of an ML model with low errors in negligible time.
4. Proposed a *parallelizable and informed* Hyperband-TPE hybrid architecture which has stronger final and anytime performances than both Hyperband and TPE.

¹Each of these solutions have been evaluated separately in their respective chapters: 7.6, 8.3, 9.5. Furthermore, for each of them we record the strengths, weaknesses and further work in "Features and benefits" and "What can be improved" sections.

With regards to the problematics outlined before, the simulation provides a robust method that makes optimizer testing practical for the first time. Because it runs almost instantly, the simulation can also be used to collect statistics about the performance of an optimizer. However, note that the simulation is not based on any real machine learning model or data. Hence, to address such concerns we developed the approximation. For the purposes of designing optimizers, approximations allow for quick collection of statistics about the performance of an optimizer as if it was run on the real ML model.

Both the simulation and the approximation match and reinforce well-known results (eg. difference between Hyperband and TPE is statistically significant). Note, that the approximation is not suitable for testing purposes. So, these two methods must coexist and are not mutually exclusive.

Finally, we used the techniques outlined above and evaluations on real world ML models, to determine the best option of combining Hyperband and TPE into a hybrid. We retained a certain architecture because in our experiments it performed very well consistently and because it is informed and parallelizable (hence, more scalable). The hybrids proposed in this paper outperform previous results on Hyperband and TPE which make the pursuit for a hybrid legitimate but there is still some distance to the state-of-the-art.

Chapter 11

Future work

11.1 Given more time

When looking back if I would have had more time, I would have run experiments on other less academic datasets for which accuracy is critical: medical imaging (for classification) and prediction of Bitcoin trading volumes (for regression). Initially, both of them were planned but due to prolonged durations of training and testing, they would have exceeded the time left. Also, more experiments on CIFAR-10 would have been more informative and useful to start building the database of loss functions for Closest known loss function approximation on CIFAR-10 as well.

11.2 Challenges

The biggest challenges that we encountered:

- existing optimizer comparison methods do not suffice for finer differences in performance (but still statistically significant)
- running an optimizer to collect statistics about its performance is time consuming
- optimizers behave wrongly while testing was non-existent mostly due to the fact that running an optimizer on true machine learning is impractically time consuming (again) for tests.

Fortunately, we overcame them and our proposed solutions (Gamma simulation and Closest known loss function approximation) turned into contributions per se.

Initially we wanted to validate our hypothesis on Bitcoin trade volumes prediction. However, this is a notoriously hard task and finding a good model requires time and extensive research into the domain. I experimented with an autoregressive model but the results did not look promising. So, unfortunately, we had to abandon this path to favour consolidation of the existing work.

Risk The fact that this project was going to be challenging and risky was no surprise since we ambitiously onboarded on a journey to find a novel optimizer.

11.3 Further work

There are interesting ideas to explore and further the findings for each of the solutions proposed in this paper. Areas of improvement and suggested further investigations are documented in the "What can be improved" subsections of the 3 separate evaluations (i.e. for Gamma simulation, for Closest known loss function approximation and for the best Hyperband-Bayesian hybrid architecture): [7.6](#), [8.3](#) and [9.5](#).

Of course, all the ideas in "Given more time" fall into the category of further work as well.

Bibliography

Academic papers/books

- [1] Diana Murgulet. *Independent Study Option: Automated Hyper-parameter Tuning for Machine Learning Models*. Imperial College London, 2018
- [2] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, D. Sculley. *Google Vizier: A Service for Black-Box Optimization* <https://ai.google/research/pubs/pub46180> Google Research Pittsburgh, PA, USA. 2017
- [3] Olson, Randal S. and Urbanowicz, Ryan J. and Andrews, Peter C. and Lavender, Nicole A. and Kidd, La Creis and Moore, Jason H. *Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I*. http://dx.doi.org/10.1007/978-3-319-31204-0_9 Springer International Publishing, 2016. Pages 123 - 137
- [4] Olson, Randal S. and Bartley, Nathan and Urbanowicz, Ryan J. and Moore, Jason H. *Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science*. <http://doi.acm.org/10.1145/2908812.2908918> Denver Colorado USA. ACM. 2016. Pages 485 - 492.
- [5] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math. March 1, 1997
- [6] Bramer M. *Avoiding Overfitting of Decision Trees in Principles of Data Mining*. Undergraduate Topics in Computer Science. Springer, London 2013
- [7] Trevor Hastie, Robert Tibshirani and Jerome Friedman. *The Elements of Statistical Learning (Data Mining, Inference, and Prediction)* https://web.stanford.edu/~hastie/ElemStatLearn/printings/ESLII_print12.pdf Springer, 2017
- [8] Adrian Catană, Andrei Isăilă, Cristian Matache, Oana Ciocioman. *Decision Trees Coursework*. for *Machine Learning* module. (report) Imperial College London 2018
- [9] Steven M. LaValle. *Planning algorithms*. <http://planning.cs.uiuc.edu/node210.html> Cambridge University Press, 2006
- [10] Robert Bridson *Fast Poisson Disk Sampling in Arbitrary Dimensions*. University of British Columbia, 2007
- [11] Jasper Snoek, Hugo Larochelle, Ryan P. Adams. *Practical Bayesian Optimization of Machine Learning Algorithms* <http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf> December 2018
- [12] James Bergstra, Remi Bardenet, Yoshua Bengio, Balazs Kegl. *Algorithms for Hyper-Parameter Optimization* Harvard University, Universite Paris-Sud, Universite de Montreal. <https://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf> 2011.

- [13] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, Ameet Talwalkar. Editor: Nando de Freitas *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. <https://arxiv.org/pdf/1603.06560.pdf> Journal of Machine Learning Research 18 (2018). June 18, 2018
- [14] Daniel Shiffman. *The Nature of Code*. December 4, 2012. Page 394
- [15] Tobias Domhan, Jost Tobias Springenberg, Frank Hutter. *Speeding up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves*. http://aad.informatik.uni-freiburg.de/papers/15-IJCAI-Extrapolation_of_Learning_Curves.pdf University of Freiburg Freiburg, Germany. 2015
- [16] Stefan Falkner, Aaron Klein, Frank Hutter. *Combining Hyperband and Bayesian Optimization*. <https://bayesopt.github.io/papers/2017/36.pdf> Department of Computer Science University of Freiburg. Jan 5, 2018.
- [17] Jiazhao Wang, Jason Xu, Xuejun Wang. *Combination of Hyperband and Bayesian Optimization for Hyperparameter Optimization in Deep Learning*. <https://arxiv.org/pdf/1801.01596.pdf> BlipAR. Jan 5, 2018.
- [18] Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, Rob Fergus. *Regularization of Neural Networks using DropConnect*. Dept. of Computer Science, Courant Institute of Mathematical Science, New York University. September 20, 2013
- [19] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gerard Ben Arous, Yann LeCun. *The Loss Surfaces of Multilayer Networks* Courant Institute of Mathematical Sciences. New York, NY, USA. January 21, 2015
- [20] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein *Visualizing the Loss Landscape of Neural Nets*
- [21] Stefan Falkner, Aaron Klein, Frank Hutter *BOHB: Robust and Efficient Hyperparameter Optimization at Scale* Department of Computer Science, University of Freiburg. Freiburg, Germany. 2018.
- [22] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. *Sequential modelbased optimization for general algorithm configuration*. In Proceedings of the 5th International Conference on Learning and Intelligent Optimization. Springer-Verlag. Heidelberg, Germany. 2011.
- [23] Romain Reuillon, Mathieu Leclaire, Sebastien Rey-Coyrehourcq *OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models* in "Future Generation Computer Systems" <http://www.openmole.org/files/FGCS2013.pdf> 2013

Scientific articles

- [24] Tomasz Golan. *Introduction to machine learning > decision trees > unknown parameters* https://tomaszgolan.github.io/introduction_to_machine_learning/markdown/introduction_to_machine_learning_02_dt/introduction_to_machine_learning_02_dt/#unknown-parameters December 2018
- [25] Saed Sayad. *Data Science > Predicting the Future > Modeling > Classification > Decision Tree > Overfitting* https://www.saedsayad.com/decision_tree_overfitting.htm December 2018
- [26] Anuja Nagpal. *L1 and L2 Regularization Methods* <https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c> December 2018
- [27] Boris Babenko. *weight decay vs L2 regularization*. <https://bbabenko.github.io/weight-decay/> April 2018
- [28] Jason Brownlee. *A Gentle Introduction to k-fold Cross-Validation* <https://machinelearningmastery.com/k-fold-cross-validation/>

- [29] Yurii Shevchuk. *Hyperparameter optimization for Neural Networks*.
http://neupy.com/2016/12/17/hyperparameter_optimization_for_neural_networks.html
December 2018
- [30] Jack Hessel. *How to Pick Magic Numbers aka a (hopefully) gentle introduction to Bayesian optimization*. <https://jmhessel.github.io/Bayesian-Optimization/> June 2015
- [31] Will Koehrsen. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning*
<https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization/>
June 24, 2018.
- [32] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. <http://yann.lecun.com/exdb/mnist/> January 1999
- [33] Rodrigo Benenson. *Classification datasets results*.
http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
February 2016
- [34] <https://www.stateoftheheart.ai/> June 2018.
- [35] Yann LeCun Leon Bottou Yoshua Bengio and Patrick Haffner. *Gradient-based learning applied to document recognition*. <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf> November 1998
- [36] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*.
<https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf> April 8, 2009
- [37] Danila Deliya. *Practical Assignments in q with BitMEX Data*. December 2018
- [38] Kevin Jamieson, Lisha Li, Giulia DeSalvo, Afshin Rostamizadeh, Amee Talwalkar. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*
<https://people.eecs.berkeley.edu/~kjamieson/hyperband.html>. January 2019
Moved to: <https://homes.cs.washington.edu/~jamieson/hyperband.html>. June 2019
- [39] *An Intuitive Explanation of Convolutional Neural Networks*
<https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/> August 2016.

Lecture notes

- [40] Maja Pantic. *Course 395: Machine Learning – Lectures, Lecture 3-4: Decision Trees* Imperial College London, 2018
- [41] Stavros Petridis. *Course 395: Machine Learning – Lectures, Lecture 7-8: Artificial Neural Networks I* Imperial College London, 2019
- [42] Stéphane Canu. *Lecture 7: Tuning hyperparameters using cross validation*
https://cel.archives-ouvertes.fr/cel-01003007/file/Lecture7_Cross_Validation.pdf
Sao Paulo 2014
- [43] Wiebke Kopp. *Gaussian Processes* (online lecture based on "Gaussian Processes for Machine Learning" by Rasmussen, Williams [ch 1,2]) <https://youtu.be/9hKfsuoFdeQ> Technische Universität München, December 2018
- [44] Nando de Freitas. *CPSC540 Gaussian Processes* (lecture notes)
<https://www.cs.ubc.ca/~nando/540-2013/lectures/16.pdf>
University of British Columbia, January 2013

Media references

- [45] Arden Dertat. <https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6> December 2018

- [46] James Le. <https://www.datacamp.com/community/tutorials/decision-trees-R> December 2018
- [47] Jack Hessel. <https://raw.githubusercontent.com/jmhessel/jmhessel.github.io/master/images/BayesianOpt/grid.png> December 2018
- [48] Yurii Shevchuk. http://neupy.com/_images/100-random-points.png December 2018
- [49] Daniel Hernandez-Lobato. *A tutorial on Bayesian Optimization* Universidad Autonoma de Madrid https://dhnzl.files.wordpress.com/2016/12/fuzzymad2016_bo_pdf.pdf December 2018
- [50] Alireza Andalib. *Workshop on Soft Computing and Big Data* K.N. Toosi University of Technology, Tehran <https://image.slidesharecdn.com/random-161117130203/95/evolutionary-algorithms-17-638.jpg?cb=1479388027> January 2019
- [51] Will Koehrsen. https://cdn-images-1.medium.com/max/1800/1*H5pyf3G115WGJwPpg65yaQ.png January 2019
- [52] Will Koehrsen. https://cdn-images-1.medium.com/max/1800/1*6SH5O_ail54karro8j0NGg.png January 2019
- [53] Kevin Jamieson, Lisha Li, Giulia DeSalvo, Afshin Rostamizadeh, Ameet Talwalkar. https://people.eecs.berkeley.edu/~kjamieson/hyperband/images/deep_1.png January 2019
- [54] Kevin Jamieson, Lisha Li, Giulia DeSalvo, Afshin Rostamizadeh, Ameet Talwalkar. https://people.eecs.berkeley.edu/~kjamieson/hyperband/images/rank_chart.png January 2019
- [55] Kevin Jamieson, Lisha Li, Giulia DeSalvo, Afshin Rostamizadeh, Ameet Talwalkar. https://people.eecs.berkeley.edu/~kjamieson/hyperband/images/bar_plot_sample.png January 2019
- [56] Jason Brownlee. *Display Deep Learning Model Training History in Keras* <https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/> June 2019

Appendix A

Implementation

A lot of software engineering work was required to implement optimizers that can take real machine learning models, Gamma simulations and Closest loss function approximations as inputs.

A.1 Designed for architecting optimizers and for tuning parameters

The starting point was a library called "Autotune" which was used within Imperial College for automatic hyperparameter tuning. However, its design suffered from high coupling and, hence, it was not flexible enough to implement all these things in parallel. Hence, I re-wrote most of it bearing in mind the aims of this paper and the design principles outlined in the next section. We appreciate that there are many automatic hyperparameter tuning libraries available, especially for Python. What makes Autotune stand out is that it is made with the purpose of architecting optimizers in mind not only as a tool to optimize hyperparameters. While this is great for optimizer designers, Autotune empowers hyperparameter tuners (i.e. users) to perform *multi-objective* optimization in terms of any metric for example the weighted average between validation error and test error.

A.2 Autotune 2.0 design principles

When coding up this project, from a software engineering perspective, there was a single principle to follow: **flexibility**. That means having the possibility of implementing a new optimizer, simulation or ML model straight away.

For example, in the current state, Autotune 2.0 allows the implementation of other history transfer schemes (apart from NONE, SURV, ALL, SAME) within minutes. Also, elaborating new optimizers that take as input real machine learning, mathematical functions (eg. Rastrigin), Gamma simulation and Closest known loss function approximation is very straightforward. Moreover, implementing other solutions like machine learning simulations or approximations is also made simple by Autotune.

A.3 Customizable pipeline

We have a very simple framework for optimizers which, in turn, can take as input anything that implements a common API: a real machine learning model, a function or a simulation. The design of this common API is built like a pipeline in which every section can be easily replaced or simulated. That is, every section of the pipeline has its own API as well. The sections are:

1. fetch/load dataset
2. specify a model (eg. CNNs)
3. specify hyperparameters to optimize and their domains
4. draw hyperparameter values according to some schedule (given by an optimizer)
5. build, train, evaluate the model on the given hyperparameters

6. update schedule/optimizer and go to 4

Pipeline in code

The above concepts, in practice, have the following APIs:

- **ImageDatasetLoaders:** All the data handling (downloading, splitting, ...), by dataset (eg. CIFAR) is handled here. Most importantly a DatasetLoader provides torch DataLoaders for training, validation and test sets.
- **Arm:** Stores the names and values for the hyperparameters. Arms have a method to draw random values. Before, this was a messy dictionary which was also containing other information.
- **Domain:** For the hyperparameters that need to be optimized, the framework requires a Domain object that can be seen as hyperparameters metadata: domain of values that can take, random sampling strategy (eg. uniform)...
- **ModelBuilders:** Given the values for the hyperparameters (an Arm) we need to create the ML model (eg. CNN), that is, to feed the arm to a ML model and to decide upon an optimization method (eg. SGD).
- **Evaluator:** given a model builder (that is an Arm and a model based on that arm), train, test and report error (all these operations should be the result of a method: `evaluate()`). Checkpointing should also be done at this level.
- **HyperparameterOptimizationProblem:** records the hyperparameters to optimize and their domains, also here we assemble most all the things above in order to provide an evaluator for this problem. For simulation problems, one should also inherit from SimulationProblem (note the multiple inheritance).
- **Optimiser:** records the stopping conditions (max time and max iterations) of any optimisation method. All optimisation methods (eg. TPE) should implement this "interface".
- **OptimisationGoals:** Metrics in terms of which we can perform optimization (individually or by aggregation). That is, the optimizers will minimize/maximize one of their attributes or even aggregations (Eg. weighted sum) of attributes, as indicated by a given optimization function. This is the result of `evaluate()` on an evaluator.
- **Evaluation:** an evaluation is nothing but the pair between the OptimisationGoals and the Evaluator that produced them. This is particularly important for informed optimizers like TPE, SigOpt or Hyperband-TPE hybrids.

A.4 Strengths and weaknesses

In short, Autotune's main strengths are, for users - multi-objective optimisation and for optimizer designers - all the infrastructure that allows for neat new optimizers, new simulations/approximation, and experimenting and visualizing results on real machine learning, simulations and approximations. Also, it works well with GPU infrastructure.

On the other hand, Autotune's main drawback is that it is not scalable. That is, all optimizers even though they can be parallelized are implemented in a sequential manner.