Novel Approaches For Tuning Models' Hyper Parameters

Antoine Scardigli[1], supervised by Diego Antognini[2]
[1]EPFL Bachelor Student in Computer Science BA5
[2]EPFL PhD in LIA lab

ABSTRACT
Hyperparameters are omnipresent in the machine learning field as they determine the performances of models. Past work proved the superiority of RandomSearch over GridSearch. Nevertheless, novel search algorithms using novel optimization approaches have been developed and claim superiority over RandomSearch and other novel algorithms. In this paper, after shortly presenting and explaining several algorithms, we conduct an empirical comparison on several different neural-network problems to evaluate if a novel algorithm performs better than other algorithms, and in particular if it finds significantly better configurations than RandomSearch. We demonstrate that HyperOptSearch, using Bayesian Optimisation with Tree Perzen Estimators finds significantly better hyperparameters configurations than all the other algorithms for almost all problems, and that it is more efficient than RandomSearch as it converges faster. We also study scheduling as it is not universally used among searchers. We quantify the acceleration on search algorithms caused by schedulers. We propose new strategies which accentuate this acceleration at a small overhead cost. Further work could include the conceptualisation of a scheduler exponentially faster than actual schedulers, as we demonstrated it was theoretically possible in Appendix B.

1 INTRODUCTION

Hyperparameters are crucial in machine learning, as they determine if the model will be efficient and accurate. Usually, a tuning process is used to find good combinations of hyperparameters. This tuning process used to be very time-consuming as it historically consisted of training the model with all combinations of hyperparameters among the space of all possibilities for grid-search, and of a random set of combinations for random search. Then results were compared to choose the best combination. Now notice that the space of possibilities grows exponentially with the number of hyperparameters. Furthermore, for complex deep-learning problems a single training can take days. Thus new approaches are needed to find good hyperparameters faster. In this project, we would like to choose a few deep-learning problems, as they are usually long to train, and compare different tuning approaches for those problems [1]. We would like to empirically quantify and qualify the speed for each method as well as the efficiency of the parameters found.

Project Plan:

1. Choose and describe traditional optimization algorithms which are going to be compared during the project.
2. Implement a benchmark tool to measure the results of various optimization algorithms.
3. Choose a batch of different machine learning problems (more precisely, deep-learning problems).
4. Analyze the results and determine if any approach performs better on average in terms of efficiency or in terms of speed than random search, which is the most commonly used algorithm.

[1]using the Ray Tune library

2 DESCRIBE TRADITIONAL AND NOVEL OPTIMIZATION ALGORITHMS FOR HYPERPARAMETER TUNING

We will in a first instance describe most current approaches used in the research community for optimizing hyperparameters. Then main concepts and tools used behind more recent optimization methods. Finally, we will describe novel optimization approaches proposed by the RayTune library. We provide in Appendix C some basic vocabulary.

2.1 Overview of traditional methods

The historical approach was Grid search. It consists of testing all hyperparameters combinations on the validation set to find the combination with the best metrics. Time budget then only depends on the granularity of the combinations tested. As an example, if there are d parameters, each with n possibilities, the space of different possibilities/combinations is n^d. This is exponential in function of the number of hyperparameters, so very inefficient for large d. Note that grid search is trivially parallel.

The most commonly used approach with grid search for hyperparameters optimization is Random search: It consists of trying randomly some combinations in the space of possibilities which should (should because random) find a decent combination in a reasonable amount of time. It is better than Grid search for the following reasons:
Firstly, if only a small number of hyperparameters a have a big influence on results out of d hyperparameters each having n possible different values, then, GridSearch tests with n different values each parameter. So there are only n^a useful trials (combinations where an important parameters changes) out of n^d different combinations. The proportion of useful tests out of all tests is:

$$\frac{n^a}{n^d} = n^{a-d} \tag{1}$$

That is exponentially inefficient in function of the number of dimensions with low impact. Now random search is going to make all d parameters change at each trial (since random), so n^d useful tests since all a dimensions change for every trial. The proportion of useful training is:

$$\frac{n^d}{n^d} = 100\% \tag{2}$$

A second interesting property about random search is that independently of the shape of the space, random search is statistically assured to find a good solution if there is one, as long as there are enough trials. Imagine that a proportion of $x\,\%$, $x \in \mathbb{R}\backslash\{\ 0 \leq x \leq 100\}$ of the combination space will provide very good results (x is often very small). We are interested in finding **at least one** solution in this subspace. Now let's define the $p_x(n)$, the **probability** that all n search are **not** in the subspace of interesting results (the subspace is $\frac{100}{x}$ times smaller than the original space). Since all trials are i.i.d and random,

$$p_x(n) = (1 - \frac{x}{100})^n \tag{3}$$

We see on Table 1 that $1 - p_x(n)$ converges to 1 as n goes large. We find that a combination is in the chosen subspace with probability which tends to 1 as n goes large, this, independently of the subspace shape. Note that random search is also trivially parallel.
Please look at Bergstra & Bengio (2012) for more details.

Table with example values of **1 -** $p_x(n)$ with varying values of n and x.

$x\backslash n$	10	100	1000	10000
0,01	0	0,01	0,095	0,632
0,1	0,01	0,095	0,632	1
1	0,096	0,634	1	1
10	0,651	1	1	1
100	1	1	1	1

Table 1 Please remember that x is in percentage of the whole space size. The most distant value we rounded to 1 was 0,99995

2.2 Tools and concepts used by the novel tuning algorithms

New kinds of optimization methods have been proposed since a couple of years: The global idea is rather to try different parameters via grid search or random search, then to select a set of parameters in an **adaptive manner**: Once one knows the results of a first set, he shall choose next set smartly. These methods aim to identify good configurations more quickly. We will here explain and describe all approaches proposed by the Ray Tune python library.

But first, as they will be widely used in the described approaches, let us explain some of the main tools and concepts used in those different optimization methods.

- SHA: SuccessiveHalvingAlgorithm has a very intuitive concept: given n combinations, it trains all n trials once, selects the best $\frac{n}{2}$ (sometimes $\frac{n}{3}$), and then repeats this process recursively on the $\frac{n}{2}$ best combinations. The best (last) trial is found in $2 \times n$ trainings as $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$. And it is trained $log_2(n)$ times.
  The conceptual interest of SHA is that best models are going to get more and more training time as it stops the training of models with low potential. This permits to have a bigger n, or to have a training $log_2(n)$ faster. See Li et al. (2017).

- ASHA: Stands for AsynchronousSuccessiveHalvingAlgorithm . It allows parallelisation and asynchronisation of SHA, hence permitting a trivial parallelisation of HyperBand. See Li et al. (2018).

- MSR: Median Stopping Rule is very close conceptually to SHA. It simply stops a trial when its performance is below the median performance of all other trials after an equal training. As SHA, it

needs to train all models at least once. MSR, SHA, and ASHA are schedulers. Please refer to Appendix B for more information about schedulers.

- BO: Bayesian Optimisation uses the principle of Bayesian inference: it exploits all observed trials results to infer the probability results of not yet observed trials. The probability representation of the objective function using previous observations is called the *surrogate*.

  The surrogate can be a **Gaussian process**: For a distribution of the result estimation $\mathbf{N}(\mu, \sigma)$, if the σ is large, then exploration is interesting as an unknown maximum could probably be there, and if the μ is big, then the algorithm should exploit as x could be near the maximum. The dilemma is resolved by taking the **maximum** of *the potential function*

  $$f(x) = \mu(x) + k \times \sigma(x) \tag{4}$$

  where k is the weight of exploration and x is the combination of hyperparameters in the search space.

  We can observe in Figure 1 a Bayesian Optimisation in process, with the blue curve being the real function, the black one the predicted function, and the blue area the incertitude about the predicted function. On the bottom in purple is the potential function.

  Gaussian process updates predictions after new observations, and chooses to observe next the combination with the new best potential. It tries to **maximize** $p(P|O)$, the probability of having a good prediction given observations. See Balandat et al. (2019).

  Other surrogate functions exists such as the **Tree-structured Parzen Estimator (TPE)**: Using Bayes rule, we can see:

  $$p(P|O) = \frac{p(O|P) \times p(P)}{p(O)} \tag{5}$$

  This surrogate will model $p(O|P)$ and $p(P)$ to find $P(P|O)$. In comparison, Gaussian process modeled directly $P(P|O)$.
  Here is the TPE algorithm explained: After a random initialization, models are separated in two groups, best performing ones, and the others, and let y be the threshold, and f be the indicator of performance (loss, accuracy...). Then let's have densities l and g from Parzen estimators's model such that

  $$p(o|P) = \begin{cases} l(o) & f(o) < y \\ g(o) & f(o) \geq y \end{cases} \tag{6}$$

  o an observation. The expected improvement if x is chosen as next observed model then is shown to be $\frac{l(x)}{g(x)}$, so we have to choose x that minimizes $\frac{g(x)}{l(x)}$. See Bergstra et al. (2011) and Falkner et al. (2018).

  Note that Bayesian optimisation is very good for early termination as the same surrogate function can be used to predict the expected remaining gains (if we continue the algorithm). It also allows parallelism, but as a trade-off for performances: the model will have more information in sequential mode than in parallel mode.

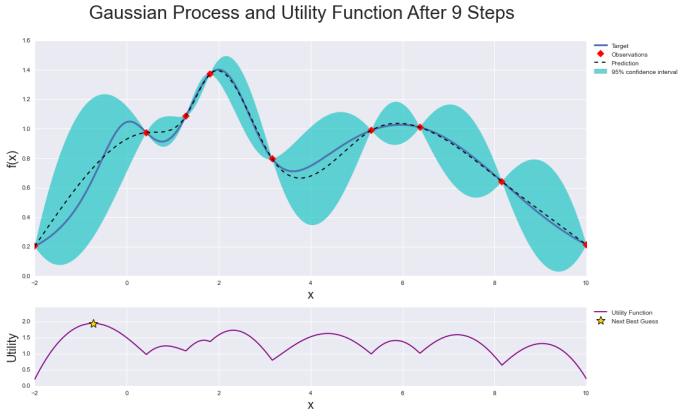


Figure 1. Example of a BO process searching the maximum of a function, see Gamboa (2019).

- Multi-armed Bandit Problem: This optimisation problem consists of optimizing gains when K cash machines are available, and it is only possible to use one cash machine per turn.
  The interest of this problem is to find the trade-off between *exploration*: trying an untested cash machine with the hope of discovering an even worthier one, and the *exploitation*: choosing the yet discovered best cash-machine.
  We define the regret as the difference between the reward sum in case of an optimal strategy and the sum of actual collected rewards. When we speak of optimizing hyperparameters, cash machines are the models, and gains are the efficiency results of the models.
  Here are some known algorithms which try to find the best trade-off:

  The simplest algorithm "greedy" consists of maximising exploitation: always choose the model with best performances.

  Another simple strategy is the ***epsilon first strategy*** : $\epsilon \times N$ steps of random exploration followed by $(1 - \epsilon) \times N$ steps of exploitation of the best yet discovered models. Another similar strategy is the **epsilon strategy**: It explores new models with probability ϵ and exploits with probability $1 - \epsilon$.

The ***Thompson sampling algorithm*** : The intuition is that you select a combination of hyperparameters with a probability proportional to that combination being the best. This adaptive algorithm is very similar to SHA or MSR, intuitively, we invest trainings in models with high potential. This potential will be confirmed or invalidated quickly and with strong evidence. We can see in Figure 2 how this latter strategy compares to the others, and more generally the strong advantage of finding a good optimization method. See Dimmery et al. (2019).

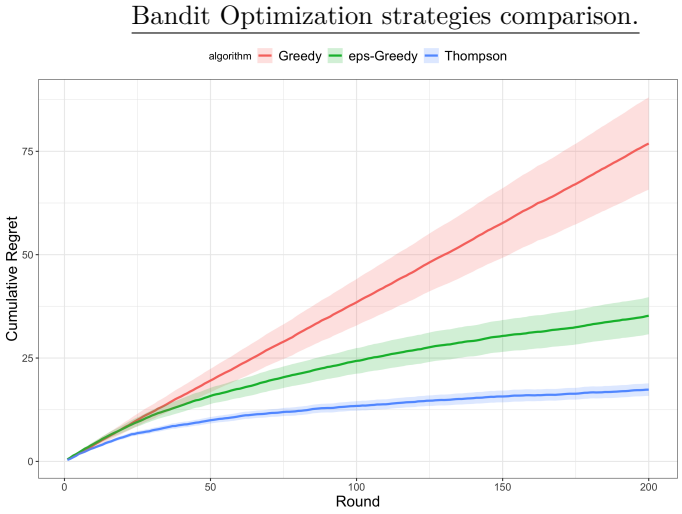


Figure 2. We can see that regret is sub-linear for Thompson algorithm

- PBT: Population-Based Training trains n models in parallel with different random combinations. Once all models are trained with budget B, they are tested. Models with bad results copy the hyperparameters of most performing models plus some noise, intending to outperform best models by finding even better combinations. This is the only approach that modifies the hyperparameters of the models during training.

  Summing up, PBT scheduler first starts by creating n models in parallel with random hyperparameters. Then it starts a **while Loop**. It trains all n models with budget B, then only the best a models are kept, and the $n - a$ others take hyperparameters close to the best models. Return to **Loop** until convergence.
  This algorithm is promising as it is similar to random search with the difference it reallocates training budget smartly in zones with high potential.

- HyperBand. This method is using bandit-based optimisation: The premise is that SHA trains all n combinations at least once which can be much too long. If you only have a budget B to spend on training, Hyperband proposes to evaluate m random combinations the following way: decompose the m trials in $log(m)$ batches such that, for batch i from 0 to $log(m)$, batch i has $\frac{m}{2^i}$ trials, and then, using the SHA algorithm, spend budget $\frac{B}{log(m)}$ on each batch. The budget needs to be at least r for all models, and at least one model is trained with budget at least R, R is a random input of the program ($R > r$), and r a function of R and i. The execution time is $B \times log(R)$.

  Intuitively if a model requires a long time before having converging results, we would prefer a small n (few trials trained longer), to give enough budget to each model so it has relevant results. Equivalently, if the models are not noisy and train quickly, we want n to be as large as possible. HyperBand finds the solution to the latter Bandit problem by proposing random R and random sets of configuration m for SHA with limited budget B. It balances very aggressive evaluations with many trials and a small budget, and very conservative evaluations with bigger budgets and fewer trials. It has infinite arm since the space of hyperparameters combinations needs to be infinite, which is usually the case (continuous). It also provides early stopping, meaning the algorithm stops as soon as it does not notice any significant improvement. Please see Li et al. (2017).

2.3 Description of the Tuning algorithms proposed by the Ray-Tune library

- AxSearch: Ax can optimize continuous configurations (integer included) using Bayesian optimization improved using matern kernel to manage very noisy models. It optimizes discrete configurations (only several choices) as a multi-armed bandit problem, solved using the Thompson sampling algorithm and empirical Bayes. We shall not use this latter algorithm as it is an optimisation algorithm for a small number of possibilities, opposed to deep-learning problems which could have a lot of combinations of hyperparameters. Balandat et al. (2019), Dimmery et al. (2019).

- DragonflySearch uses Bayesian optimisation adapted for expensive large scale problems with high dimensional spaces. It also provides a parallel approach. The main idea is the following: it is known that BO is very successful for low dimensional models ($d < 10$). But it has exponential complexity in function of the number of dimensions.

So the dragonfly algorithm decomposes the result function

$$g(x) = g_1(x_1) + g_2(x_2) + ... + g_n(x_n) \quad x_i \in X_i \quad (7)$$

where X_i is a set of very low dimensionality randomly chosen such that every X_i is disjoint with any X_j, and any potential function f_i is independent with $g_j, \forall j \neq i$, and $g_i \forall i$ is considered as sampled from a Gaussian process. Then it is proved that the overall potential function described in our paragraph about BO is simply the sum of each potential function, resulting in a huge performance improvement. That is:

$$\begin{aligned} f(x) &= f_1(x_1) + f_2(x_2) + ... + f_n(x_n) \\ and \quad f_i(x_i) &= \mu_i(x_i) + k \times \sigma_i(x_i) \end{aligned} \quad (8)$$

One now simply has to observe the maximum of f. See Kandasamy et al. (2020).

- SkoptSearch is using Bayesian Optimization as we explained above, with Gaussian process as a surrogate function. Louppe & Kumar (2016).

- HyperOptSearch is using Bayesian Optimization as we explained above, with Tree-structured Parzen Estimators as surrogate function. See Bergstra et al. (2011).

- BayesOptSearch is using regular Bayesian Optimization Search with Gaussian process as well.

- TuneBOHB is named after Bayesian Optimisation concatenated with HyperBand. The interest is to take the best of the two worlds as Hyperband shows strong performance and scalability, but samples configurations randomly (without learning from previous observations, although BO is very good at this). TuneBOHB uses Bayesian optimisation with TPE instead of Gaussian process because it would scale better in high dimensional space, according to the authors.

  BOHB uses the HyperBand algorithm to determine the number and the budget of each configuration, but replaces the random selection of combinations by a BO algorithm for most samples (with probability $1 - \rho$) based on all observations so far. The combination is still selected with probability p at random. It can use the parallel properties of both TPE and HyperBand to provide efficient parallelism. It is said by his authors to always beat or match both methods.
  We can observe on Figure 3 how TuneBOHB is always better than both random search, BO, and HyperBand according to the authors. See Falkner et al. (2018).

Comparison of BOHB with other search algorithms

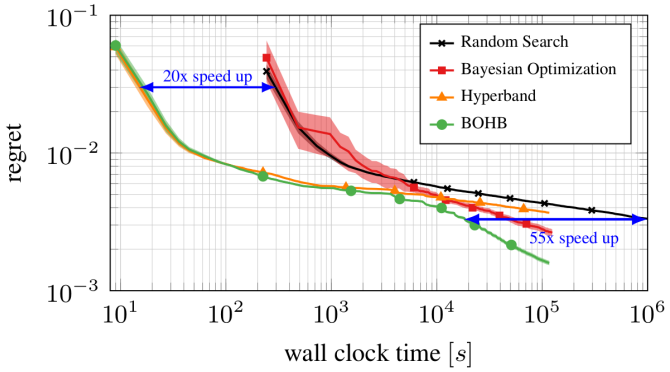


Figure 3. Comparative plot of HB, BO, BOHB and random search for a deep learning problem according to Falkner et al. (2018).

- NevergradSearch is a Facebook open project that proposes several algorithms. Here is a selection of three evolutionary algorithms:

  TwoPointsDE : Differential Evolution algorithm is the following:
  First n trials are trained with random combinations.
  Then loop until convergence or end of time budget:
  (1) For every trial x,
  (2) choose 3 other distinct trials a, b, c.
  (3) Then for every hyperparameter of x, keep same value with high probability, and take a combination of a, b, c 's values otherwise *(mutation)*. At least one hyperparameter must change.
  (4) Then keep the best model between the old x and the new x *(selection)*.
  This is the historical algorithm. Facebook's algorithm only differs from the facts there are only 2 others models a and b instead of a, b, c. See Storn & Price (1997).

  CMA : This algorithm uses two different concepts: First *exploration*: by updating the distribution of configurations of all trials in a way that the average of the new configurations is equal to the average from previous best trials. The distribution of newly generated trials is based on the co-variance matrix, which is incremented to increase the likelihood of previous search steps (new trials are generated in a distribution such that they explore promising directions).
  The second concept is that it adaptively increases or decreases the variance depending on last steps. If the model is in an exploration phase, variance will increase until it finds the

best combinations, but if we are close to best results (*exploitation*), the variance becomes very small, allowing fast convergence. See Hansen (2006).

PSO : This algorithm imitates birds or ants movement. First initialize n trials with random configurations. Imagine the configuration is a position in a d-dimension space. PSO assigns to every trial an additional information: it's best position so far. Let g be the best position of all models so far. Then on each loop every trial is going to have on every dimension a speed, function of both its best position times a normal distribution sample and the best position of the swarm times a normal distribution sample. Then update the *position* as last *position* + *speed*.
See Kennedy & Eberhart (1995).

- ZOOptSearch: is a derivative-free optimization.

  The classification model will classify configurations of trials as either good (the k best) or bad, by using a parallel to axis hyper-rectangle that contains all good combinations (the hyper-rectangle has same dimension as the search space). Then **loop** on the following: (1) Choose a configuration randomly from either the d dimension hyper-rectangle with probability λ, or the whole space with probability 1-λ, (2) Evaluate the trial and (3) Replace the worst of the k best trials with this new trial. (4) Replace a random bad solution with the worst of the k solutions (the one just replaced).
  (5) Update the hyper-rectangle and go back to loop.
  The asynchronous process is accomplished by parallelising the loop on different threads, but it still needs to update the same hyper-rectangle after every evaluation. Liu et al. (2019).

- SigOptSearch: We have not been able to find any concept SigOpt uses. It has been removed from Ray-Tune library in the meantime.

2.4 Discussion and chosen algorithms

In a nutshell, we can regroup the algorithms by the concept they use: *AxSearch*, *SkoptSearch*, *HyperOptSearch* and *BayesOptSearch* use Bayesian Optimisation with Gaussian process. *HyperOptSearch* and *DragonflySearch* use Bayesian Optimisation with TPE as a surrogate function. *AxSearch* also proposes a Bandit based solution in case of discrete spaces of very small size. *NevergradSearch* is the only population-based optimizer. *ZooptSearch* proposes a derivative-free optimization. Finally, *TuneBOHB* uses both concepts from Bayesian optimisation and Bandit-Based optimization, which is intuitively powerful as they complete each other's weakness. You can find in Table 2 a summary of informations related to all algorithms.

We decided to consider one optimization algorithm of each kind. That is: *DragonflySearch*, *AxSearch*, *BayesianOptSearch*, *NevergradSearch*, *ZooptSearch* and *TuneBOHB*.
These optimization algorithms will be compared between them and will also be compared with *RandomSearch* using the metrics we will define in the next section. Note that we might sometimes shorten the algorithms names by removing *Search* or *Optimization* from the name.

Table 2 Characteristic of all search algorithms.

Tuning Algorithm	Parallelism	Adaptive algorithm	Efficient despite ineffective parameters	Early termination	Main Concept
GridSearch	✓				Extensive search
RandomSearch	✓		✓		Random search
AxSearch		✓	✓	✓	BO or Bandit Based
DragonflySearch	✓	✓	✓	✓	Enhanced BO with TPE
SkoptSearch		✓	✓	✓	BO with GP
HyperOptSearch	✓	✓	✓	✓	BO with TPE
BayesOptSearch	✓	✓	✓	✓	BO with GP
TuneBOHB	✓	✓	✓	✓	BO and HyperBand
NevergradSearch	✓	✓	✓		Population-based
ZOOptSearch	✓	✓	✓		Zeroth Optimization (ASRACOS)
SigOptSearch		✓	✓		Unknown

3 TOY BENCHMARK

We will first explain how we will present the algorithms performances. Then how we implemented the benchmark tool. Finally we will propose a detailed analysis of every problem.

3.1 Presenting results of algorithms

Before implementing the benchmark tool, we would like to explain three ways of presenting the metrics (results) obtained by trials of every algorithm.

- One of the most common approach is to compare the test error (or equivalently accuracy, score, ...) of the best configuration found by the search algorithm as a function of the total number of trials/the time needed to find the best configuration.
- An enhancement of the protocol above consists in dividing the number of trials into several experiment, to have test accuracy of the best trial, but also the distribution of performance among experiments as a boxplot. Boxplot includes lower and upper quartiles, whiskers above and below to show the position of the most extreme data point, two black lines across the top of the figure to mark the upper and lower boundaries of a 95% confidence interval. See Bergstra & Bengio (2012).
- Finally, another approach is to use the best error from the validation set instead of the test set. The idea consists of finding the result from the best validation trial averaged among experiments in function of the number of generated trials. The limitation of this method is over-fit, which is hopefully infrequent in neural networks . Dodge et al. (2019) also informs about the variance by adding an area σ wide around the averaged best validation accuracy.

We will process our result data the following way: we define as best trial for trial number n, the trial with the best validation accuracy for all trials from *0* to n. This means the plot for validation value should always be increasing for classification accuracy (decreasing for Regression error), but the plot for test value could be decreasing (increasing for regression error) because a new trial with a better validation accuracy could have a worse test accuracy.
We are only interested in the yet best trial because what we want from a search algorithm is that it gives the best hyperparameter combination the fastest possible. So we are interested in the best test accuracy over all yet trained trials, as a function of the number of trials done.

3.2 Toy Benchmark tool implementation

We implemented a first version of our benchmark tool. This version is very extensive as we allow various combinations for tuning optimization. To compensate with this extensiveness, we chose some very simple datasets, and do very few iterations (max 20 epochs). This toy benchmark tool has three goals: First to verify that every functionality works, that models are indeed learning. Second to understand which factors are important. Third to see how all algorithms perform relatively to each other.
We could then focus on the important factors with some harder datasets and more complicated problems as a second (and last) benchmark. This first large scale benchmark has different datasets, and different labelling categories, as search algorithm's performances can highly depend on the test problem.

Here is a short description of every labels used in the first benchmark tool:
Model Inputs: We wanted to have at least one

input category of every kind to perform an extensive comparison, so we have *Image*, *Table*, and *Text input*.
Model Output: Similarly, we seek extensiveness so we propose *classification*, *regression* and *generation*. Classification is the task of putting the input in a category, regression is the task of finding one or few real values from the input, and generation consists of creating credible new output data looking like the input data.
He have few datasets for every category:
Image classification with two datasets: *MNIST*: LeCun et al. (2010) and *Fashion MNIST*: Xiao et al. (2017). These are about finding back the digits and the clothes categories respectively from a black and white image of low quality (28*28 pixels).
Image generation with the *MNIST* dataset. The goal is to create some images of digits.
Table Regression with both *Boston*: Harrison Jr & Rubinfeld (1978) and *Diabetes* datasets: Kahn (1994). These are about finding the value of houses and the diabetes progression measure respectively from a table of data.
Text Regression is using the *IMDB* dataset: Maas et al. (2011) a huge movies review dataset to predict if the writer either liked or did not like the movie. It is a regression task because we will compute the error on the prediction: how far is the continuous prediction to the real label (either 0 or 1).
Text Classification using the *TREC* dataset: Buckley (1995) which consists of classifying text questions into one of 6 different categories of questions.
Models: We will use a lot of different models, still for extensiveness. Unfortunately, we can not use every model with every input category. As an example, CNN, which pre-treats the input in smaller parts, is useful for images and text, as the model has to extract the information, but is useless in tables, where the information is already presented optimally.
We will shortly enumerate the particularities of every model:
GAN: On the one hand there is a *generator* that tries to create outputs looking like inputs, and there is a *discriminator*, that tries to make the difference between the real inputs and the outputs of the generator. The generator is trained by whether or not it succeeds to fool the discriminator, and the discriminator by whether of not he successfully discriminated the given input. Note the generator never has access to the original inputs, so he will necessarily generate original images. We will use MNIST Dataset as it is a very simple dataset. The metric we will use is the Inception Score which quantifies how good is the generation. For this purpose we will develop an Auxiliary Classifier GAN, which means that the discriminator will have the additional task to find the label of the image, as this is necessary to compute the Inception Score. The auxiliary classifier will be a CNN. Goodfellow et al. (2014)
CNN is used for visual and text datasets as they first pre-process the data to reduce its dimensionality through pooling, or by performing convolution, which consists of not taking into account a single value, subset of the whole input (for example one pixel for an image), but to also take into account near values (surrounding pixels). First used by LeCun et al. (1989)
RNN's particularity is that the connections of its nodes form a directed graph. This allows it to have memory. It is hence very indicated for texts datasets as we would like to remember the influence of every word when going through the sentence.
We do the distinction between two RNN models:
LSTM stands for Long Short Term Memory. It divides input signal into hidden state, useful in short term, and cell state, usefull in long term. Every cell has three gates: the *forget gate* remove useless informations stored in the past (from cell state). The *input gate* adds to the cell state new informations from current input. The *output gate* takes important informations from the new cell state to create the hidden state. The hidden state is then given to the following node. LSTM introduced here: Hochreiter & Schmidhuber (1997)
GRU stands for Gated Recurrent Unit and every cell has two gates that use both the previous hidden state and the input. The *reset gate* is responsible for deciding which portions of the previous hidden state should be combined with the input to propose a new hidden space. The *update gate*decides how much of the previous hidden state is to be retained with what portion of the proposed hidden state to create the final hidden state. This new hidden state is used for determining the output. GRU were first introduced by: Cho et al. (2014).
MLP is a regular feed-forward neural network with multiple layers and non-linear activation functions.
Logistic Regression is a very simple model, the most likely to underfit as it is a single layer neural network with logistic function. We will use it for table regression.
Optimizer: The optimizer makes the model train by updating the network's node weights. The most know optimizer is SGD: Stochastic Gradient Descent. It takes it roots from an algorithm developed by Robbins & Monro (1951). It maintains a single and fixed learning rate for all weight updates. Adam proposes to keep the principle of Gradient Descent, but allows a per-parameter learning rate, and also allows a learning rate evolution, as learning in the beginning of training is often too slow. It was invented by Kingma & Ba (2014). We will compare this optimizer with AdaBelief, which is new and promising. It is an amelioration of Adam. In the case of a large gradient and a small variance, Adam would use a small learning rate, although AdaBelief would use a big learning rate by noticing that variance is small. This is intuitively preferable because small

variance gives us a big confidence that our direction is correct. It has been very recently developed by Zhuang et al. (2020).
In our benchmark, search algorithms will also have to find best values from:
The activation function, which defines the output of a node given an input. We often want it to be continuous and differentiable.
The leaning rate controls how fast the weights of the nodes in the network should learn.
The weight decay is a way to penalize complexity to reduce the risk of overfitting.
The LR dropout randomly sets the Learning Rate of a certain a proportion of nodes to zero. This is efficient to avoid bad local minimas.
The number of layers is important to avoid overfit, underfit, and to capture better complexity.
dimensions of hidden layers is important to capture better the complexity of a problem.

We use the same trial scheduler for every search: it is ASHA. It is the scheduler recommended by ray-tune, and is the most compatible among all search algorithms. Indeed only BOHB uses a different scheduler. Finally, we wish to present the different ways that exist to calculate the metrics. See Table 3.

Presentation of metrics in ML

Name	Formula	Best	Worst
Classification			
Accuracy	$\frac{T_p + T_n}{n}$	1	0
Precision	$\frac{T_p}{T_p + F_p}$	1	0
Recall	$\frac{T_p}{T_p + F_n}$	1	0
F1	$\frac{2 \times precision \times recall}{precision + recall}$	1	0
Log_loss	$-(y \times log(\bar{y}) + (1-y) log(1-\bar{y}))$	0	$+\infty$
Regression			
MAE	$\frac{\sum_i \lvert y_i - \bar{y}_i \rvert}{n}$	0	$+\infty$
MSE	$\frac{\sum_i (y_i - \bar{y}_i)^2}{n}$	0	$+\infty$
RMSE	$\sqrt{MSE}$	0	$+\infty$
R2 score	$1 - \frac{MSE}{\sigma^2}$	1	-∞
Generation			
Inception Score	$e^{E[D_{KL}(p(y\lvert\bar{y}) \times p(y))]}$	#Class	1

Also all classification metrics from generated output

Table 3 Here y is a label data, and $\bar{y}$ is the output of the model.
We also define T_p the true positive, T_n the true negative, F_p the false positive and F_n the false negative. D_{KL} is the Kullback–Leibler divergence.

Figure 4 shows the possibilities of the toy benchmark. To sum-up, it consists of several layers which are respectively (in order according to the figure): model input, problem category, NN category, optimizer, Search algorithm.

Notes on the benchmark:
After testing this benchmark, we noticed we were not able to make Dragonfly work, which is regrettable.
The goal here is to compare the difference of results between each search algorithm, and especially to see if adaptive search algorithms can provide significantly better results than random search in a significantly shorter time. We are interested in the relative difference between results of every search algorithms.

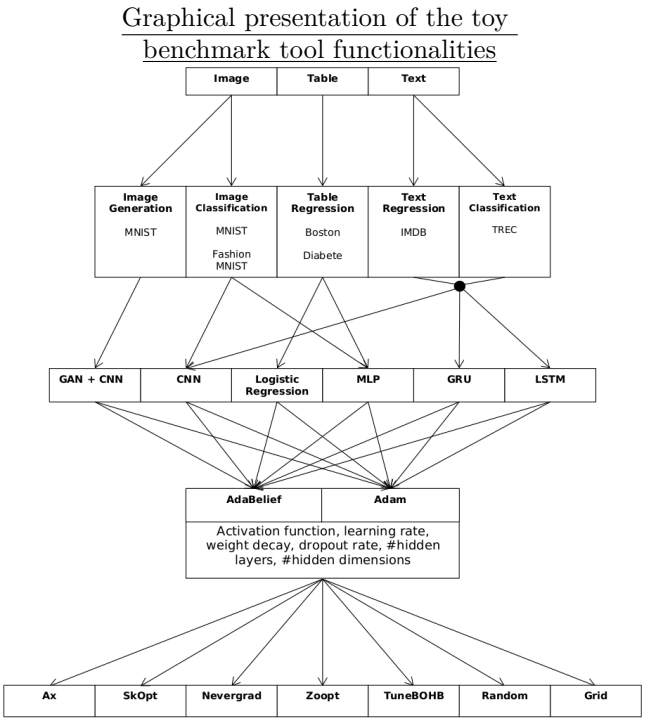


Figure 4. Plan of the toy benchmark and all the combinations it allows

We would not be surprised if results are below "good" results in the literature because we don't train on a lot of epoch, and also because we have simple models. But we are more interested in relative disparities than in results. Still, to get on order of idea, best literature we are aware of claims an accuracy of 93% on FMNIST Dataset without preprocessing, while we get 89% with our best configuration.
We end up with an 8 dimension parameter space:

- Learning rate: from 10e-8 to 0.1.
- Weight decay: from 10e-8 to 0.1.
- Activation function: either RELU, tanh, or sigmoid.
- Number of hidden dimensions: from 32 to 1024.
- Number of hidden layers: from 1 to 3.
- Dropout probability: from 0 to 0.5.
- Optimizer: either Adam or AdaBelief.
- Model: which neuronal model will learn.

We will do a detailed analysis on one dataset of each category in this section, followed by a global analysis over all datasets. It is still possible to see in Appendix A a shorter analysis for datasets in redundant categories (Fashion MNIST and Diabetes), as well as other comparisons based on a training on a smaller number of epochs. We have described 3 population-based algorithm that are all using NeverGrad. As this is only the first benchmark, we will only use one of the three (the best one according to the authors): CMA.

3.3 Detailed analysis using the Boston Dataset

For the detailed analysis on Boston Dataset, we gave 256 trials to every search algorithm (so 256 different combinations of hyperparameters). Each of these trial can learn on up to 20 epochs, but it can also be stopped before by the scheduler (all search algorithms have the same scheduler but BOHB). The scheduler stops a trial if it starts overfitting, or if it gives very bad results in the first epochs, hence has no potential and is a waste of execution time.
We repeated this process 4 times to reduce the influence of noise, and also so to quantify variance.
Our analysis will rely on the comparison metrics we presented in section 3.1.
To sum up, these comparison metrics were:

1. Plot of the best test MSE, ie from best hyperparameter combination found so far, as a function of number of trials/time.
2. Plot of the best test MSE from best hyperparameter found so far, as a function of number of trials/time, including boxplot to have interesting informations about variance, quartiles, and outbound values from the distribution among independent experiments.
3. Plot of the best expected validation accuracy from the best hyperparameter found so far, as a function of number of trials, including variance information.

Note we pre-processed the Boston Dataset by normalizing it (mean of house prices of 0 and standard deviation of 1).

3.3.1 General Informations

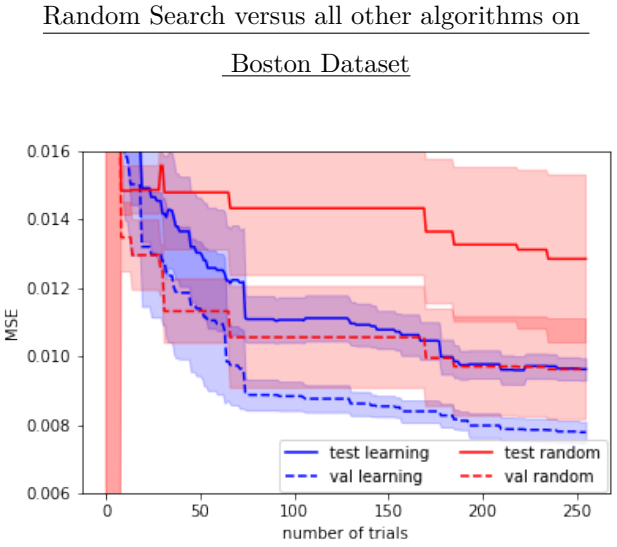


Figure 5. Here we plot the averaged test and validation MSE of Random Search, and we compare it with the mean of all other adaptive algorithms (learning algorithms). Color areas are σ wide.

As an introduction figure, we show in Figure 5 the comparison between the random search test and validation MSE, with the mean of all other algorithms's test and validation MSE.
Note that test value is the value that matters in real life, but here we want to compare the search algorithms, so it is interesting to compare validation values as well as they are the values that the optimizer, the scheduler, and especially the search algorithms use to learn and decide new hyperparameter configurations.
We were afraid that plotting boxplots would cause information overload, so we add color areas that are σ wide instead. (σ is the standard deviation)
As we predicted, validation MSE is always decreasing, but test MSE is sometimes increasing (a little).
Also, we notice that validation MSE is always below test MSE, this is because the model is selected on his validation MSE, so there is a positive bias on it.

3.3.2 Test and validation MSE

We propose in Figure 6 and 9 respectively test and validation results for all algorithms. In Figure 6, random-search has a test MSE that is nearly two times worst than all other learning algorithms.

Figure 9, which has the same axis scale as Figure 6, shows that random search has the worst results for validation MSE as well. Looking at validation MSE, we show that all algorithms but BOHB find good configurations (below 0.018) after 15 trials or less, and still improve until approximately 64 trials, after what they start converging. BOHB finds a good configuration after 64 trials and starts converging after 156 trials).
The number of trials needed to reach convergence of results is a very important information for tuning as it allows to do the tuning with a small optimal maximum number of trials for a search. This number however depends on the problem and the search space (here the search space has only 8 dimensions).

These two plots permit an overall comparison. NeverGrad, Zoopt, and Bayes and Hyper seem to be the best, and Random is the worst by far.

3.3.3 Variance per Search Algorithm

To reduce information overload, we will plot the recommended metrics only for the top 3 performing algorithms and random search. All axis scales are the same. See in Figure 7 the plot for BayesOpt, in Figure 8 the plot for NeverGrad, the plot in Figure 10 for Random, and in Figure 11 for Zoopt.

We also understand from this data that validation error has similar signification than test error as for all algorithms, the test error is close to the validation error, plus a constant.

For all algorithms, we notice a colored area spanning the whole y-axis for small number of iterations. This means that there is a huge variance during the first trials for all search algorithms, which is logical. Then Random and Zoopt keep a big variance. On the contrary, NeverGrad finds very quickly the best solution (after 64 trials). The huge variance of RandomSeach is expected as configurations are random from a uniform distribution, so by definition the variance in maximized. Variance is not desired as it means we could be unlucky and have very bad results despite searching for a lot of trials. We would prefer good trials with a high regularity (low variance). NeverGrad is prefereable for this reason as it finds the trial with the best validation error among all search algorithms, and its results have a very small variance. Zoopt and Random, contrary to all other search algorithms, did not finish converging with 256 trials configurations. We prefer algorithms that converge fast to good solutions instead, as it means doing training on less trials, hence saving time and computational power.

Test MSE

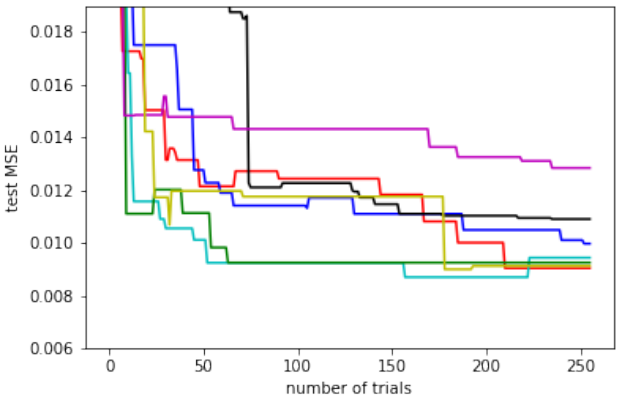


Figure 6. Here we plot the average test loss of the best trial obtained so far per search algorithm as a function of the number of trial. This is the plot presented as "metric *1.*"

Validation MSE

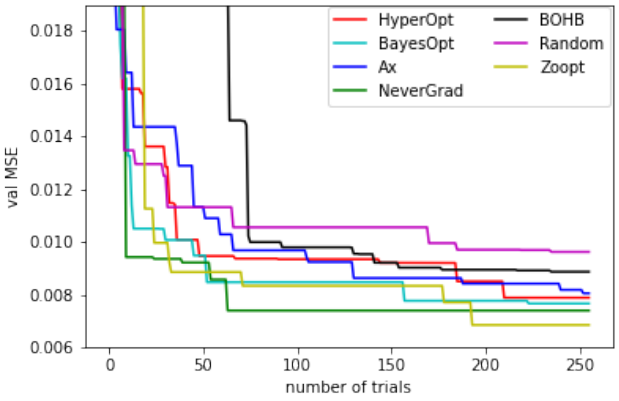


Figure 9. Here we plot the average validation loss of the best trial obtained so far per search algorithm as a function of number of trial.

Test and Validation MSE per algorithms with σ narrow area on Boston Dataset

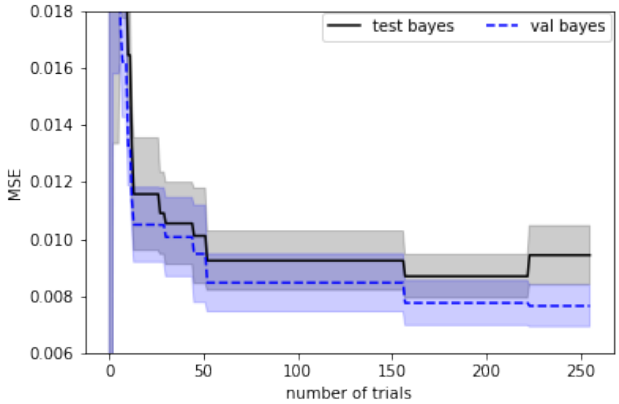


Figure 7. Validation, Test, Variance MSE with Bayes

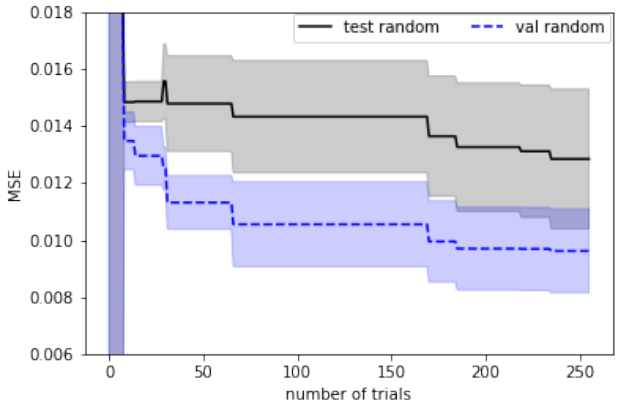


Figure 10. Validation, Test, Variance MSE with Random

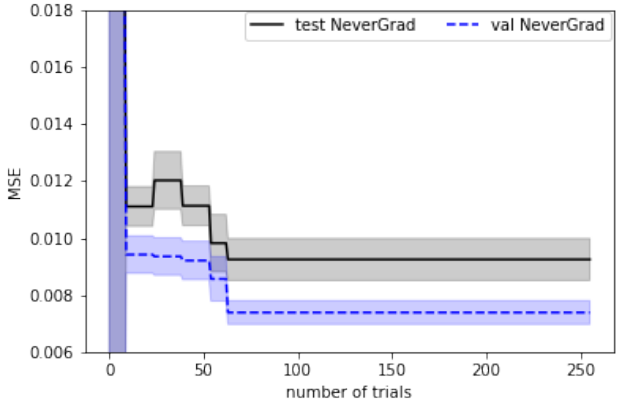


Figure 8. Validation, Test, Variance MSE with NeverGrad

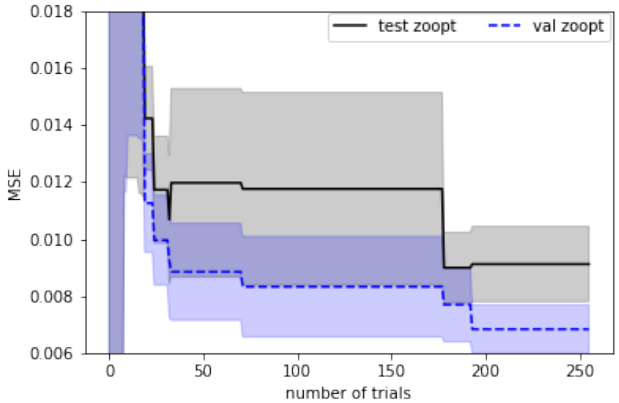


Figure 11. Validation, Test, Variance MSE with Zoopt

3.4 Detailed analysis using the TREC dataset

We showed on Boston dataset that most search algorithms would converge after creating approximately 64 hyperparameter configurations. For this reason, we will reduce the number of trials per search algorithm from 256 trials for Boston to 128 trials for this detailed Analysis on the TREC dataset. Every trial can learn on up to 20 epochs.
We repeated this process on 5 experiments to reduce influence of noise, and also to quantify variance.

3.4.1 General Informations

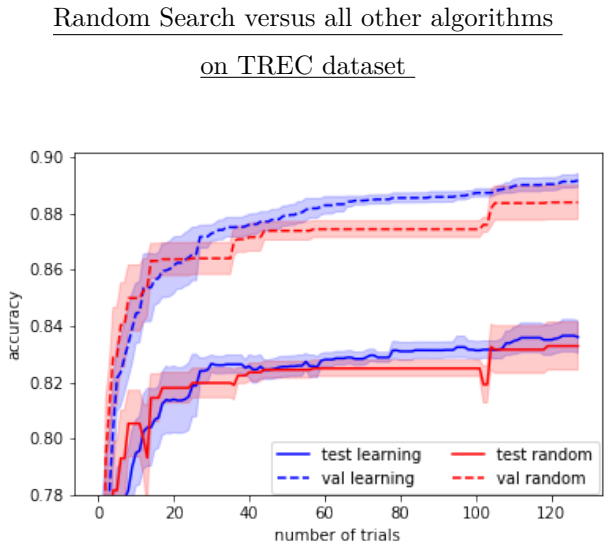


Figure 12. Here we plot the averaged test and val accuracy of Random Search, and we compare it with the mean of all other adaptive algorithms (learning algorithms). Color areas are σ wide.

As an introduction figure, we show in Figure 12 the comparison between the random search test and validation accuracy, with the mean of all other algorithms's test and validation accuracy.
We observe that validation accuracy is way above test accuracy for all search algorithms. Random is once again less goods than the average of other learning algorithms.

3.4.2 Test and validation accuracy

We propose in Figure 13 and Figure 16 respectively test and validation results for all algorithms. In Figure 13 random-search has a test accuracy almost 4% below the configurations found by Hyper and BOHB for the same number of trials. This is a very significant amount and gives us strong confidence that novel approaches for tuning hyperparameters are important. Figure 16, which has the same axis scale as Figure 13, shows that random search has the worst results for validation accuracy. Looking at validation accuracy, we see that as in Boston dataset, most algorithms find "good" (above 86%) configurations after 20 trials, and start converging after 60 trials. These two plots permit an overall comparison. Hyper, Ax, and BOHB seem to be the best, and Random is once again the worst in term of validation accuracy, and is below the average in term of test accuracy.

3.4.3 Variance per Search Algorithm

To reduce information overload, we will plot the recommended metrics only for the top 3 performing algorithms and random search. All axis scales are the same. See Figure 14 for the plot of HyperOpt, in Figure 15 for Ax, in Figure 17 for BOHB, and in Figure 18 for Random.

3.4.4 Comparison with previous datasets:

- The results of BOHB for this dataset are in opposition with those of Boston analysis. It was second last, and had a 2 times slower convergence time as other algorithms, but it is here the best, despite still converging slowly. Irregularity is unsatisfactory for search algorithms.
- We can see that the difference between average test and validation accuracy is far higher than for the first dataset compared to the difference of results among search algorithms:
  Indeed in Boston dataset (Figure 5) the difference between the average of the validation MSE and the average of the test MSE was smaller than the difference between the MSE from best algorithm and the MSE from the worst algorithm (for both test and validation values).
  But for TREC dataset, the difference between the average of the validation accuracy and the average of the test accuracy is almost 2 times bigger than the difference between the accuracy from best algorithm and the accuracy from the worst algorithm validation values. The difference of results of accuracy among search algorithms hence seems relatively smaller. What we mean here is that there is still a 3% test accuracy difference between the best algorithm (BOHB) and the average of the other algorithms, but this improvement seems small compared to the 5% gap between test accuracy and validation accuracy.
- We do not notice anymore the huge variance during the first trials. More generally all search algorithms have a much smaller variance than before.
- HyperOpt seems to show on both dataset the ability to have the closest validation and test results. This is a very interesting property, as algorithms makes their generation choices based on the validation results, but we are truly interested in the test results as algorithms create a bias on validation results.

Test Accuracy

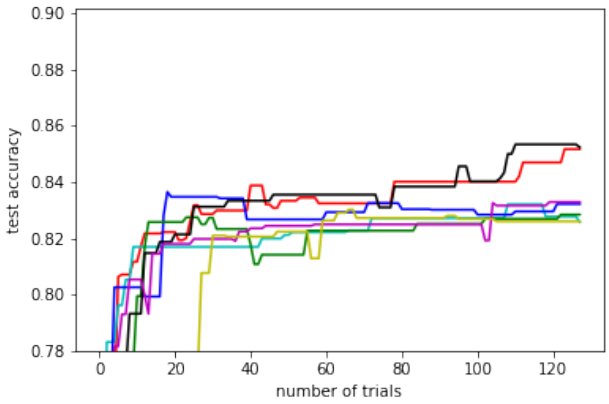


Figure 13. Here we plot the average test accuracy of the best trial obtained so far per search algorithm as a function of the number of trial.

Validation accuracy

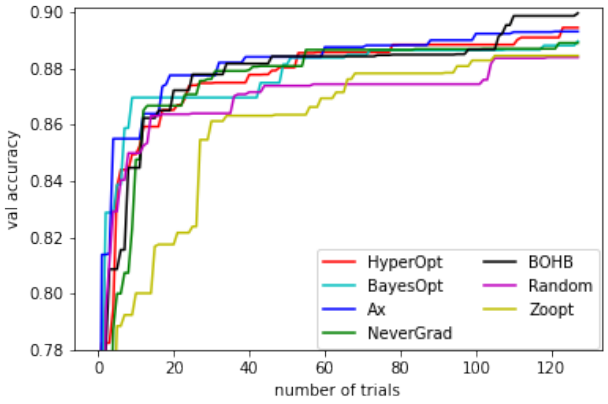


Figure 16. Here we plot the average validation accuracy of the best trial obtained so far per search algorithm as a function of number of trial.

Tests and Validation MSE per algorithms with σ narrow area on TREC Dataset

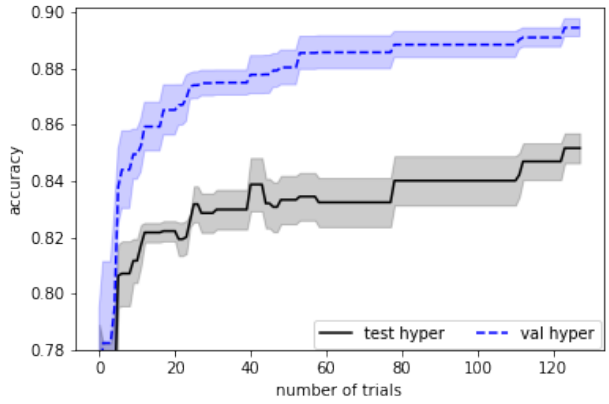


Figure 14. Validation, Test, Variance accuracy with Hyper

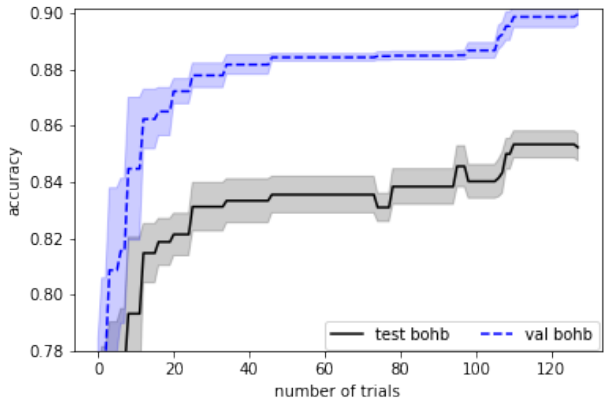


Figure 17. Validation, Test, Variance accuracy with BOHB

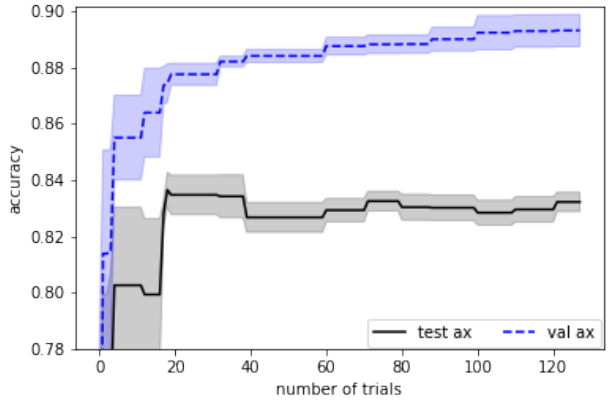


Figure 15. Validation, Test, Variance accuracy with Ax

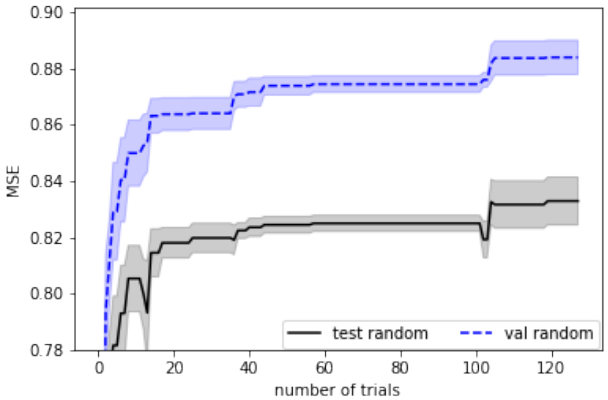


Figure 18. Validation, Test, Variance accuracy with Random

3.5 Detailed analysis using the FMNIST dataset

For the detailed analysis on FMNIST dataset, we give 128 trials to every search algorithm. Each of these trials can learn on up to 20 epochs.
We repeated this process on 4 experiments to reduce the influence of noise, and also to quantify variance.

3.5.1 General Informations

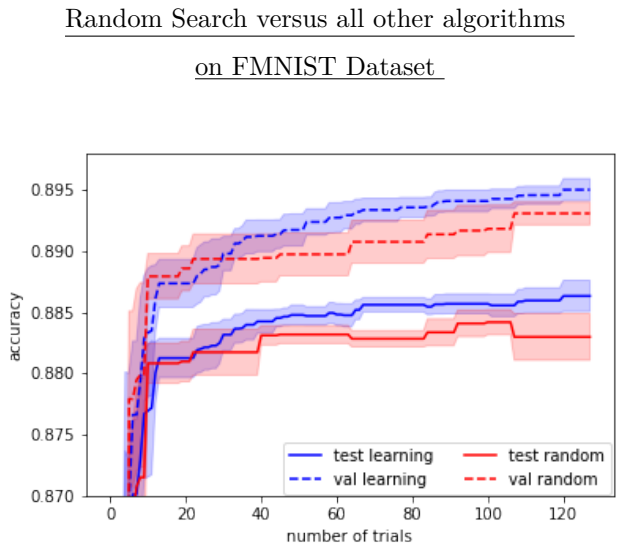


Figure 19. Here we plot the averaged test and val accuracy of Random Search, and we compare it with the mean of all other adaptive algorithms (learning algorithms). Color areas are σ wide

As an introduction figure, we show in figure 19 the comparison between the random search test and validation accuracy, with the mean of all other algorithms's test and validation accuracy.
Random is below the average of other learning algorithms once more.

3.5.2 Test and validation accuracy

We propose in Figure 20 and Figure 23 respectively test and validation results for all algorithms.
In Figure 20, random-search is the worst and has a test accuracy almost 1% below the configurations found by Hyper.
Figure 23, which has the same axis scale as Figure 13, shows that random search has the worst results for validation accuracy. Looking at validation accuracy, we see that most algorithms find "good" (above 88%) configurations after 16 trials, and start converging after 70 trials. These two plots permit an overall comparison. HyperOpt is the best, followed by BayesOpt and BOHB. Random is the worst.
HyperOpt finds the best results, but converges slowly, which mitigates the result.

3.5.3 Variance per Search Algorithm

To reduce information overload, we will plot the recommended metrics only for the top 4 performing algorithms. All axis are the same.
See Figure 21 for the plot of HyperOpt, in Figure 22 for Bayes, in Figure 25 for BOHB, and in Figure 26 for Random. Variance is small once more

3.5.4 Comparison with previous datasets:

- The results of BOHB for this dataset are in opposition with TREC and Boston analysis as here BOHB converges very fast.
- We can see that search algorithms have a limited added value on this problem. Indeed, The difference between best and worst algorithms results is below 1%. The difference between average of test results and validation results is about 1 %. This is maybe the case because the problem is very easy, hence very good configurations can not do much better than good configurations.
- Variance is low as observed in TREC analysis.

Test Accuracy

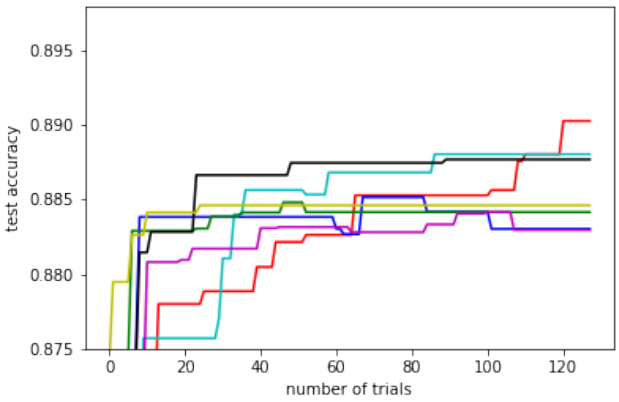


Figure 20. Here we plot the average test accuracy of the best trial obtained so far per search algorithm as a function of the number of trial.

Validation accuracy

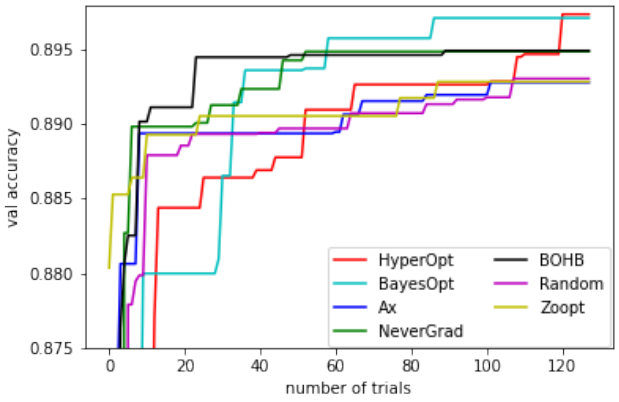


Figure 23. Here we plot the average validation accuracy of the best trial obtained so far per search algorithm as a function of number of trial.

Tests and Validation MSE per algorithms with σ narrow area on FMNIST Dataset

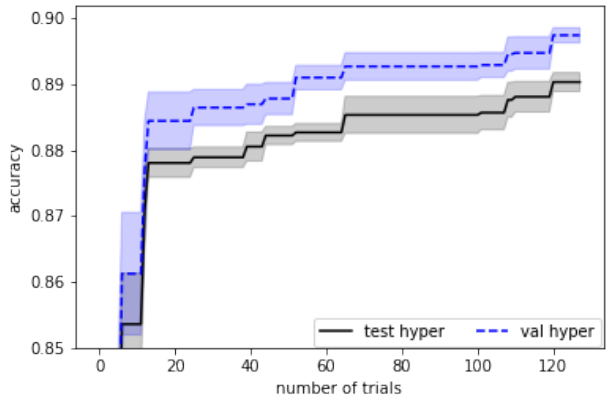


Figure 21. Validation, Test, Variance accuracy with Hyper

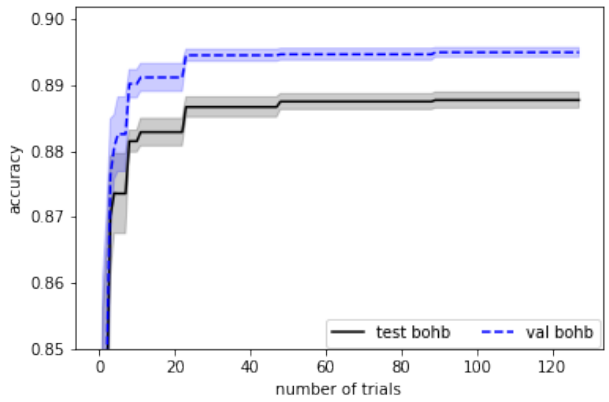


Figure 24. Validation, Test, Variance accuracy with BOHB

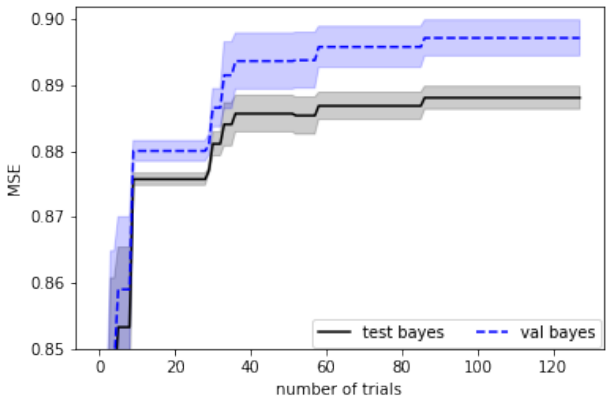


Figure 22. Validation, Test, Variance accuracy with Bayes

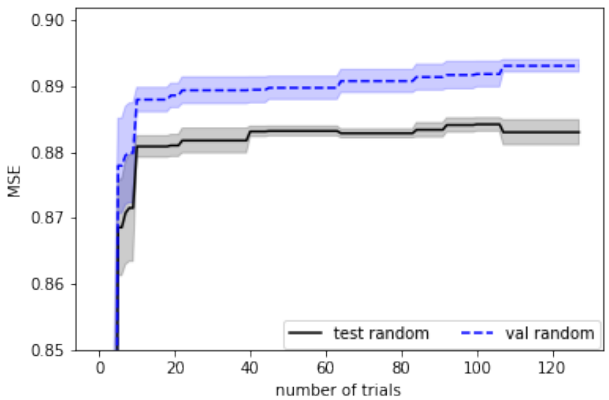


Figure 25. Validation, Test, Variance accuracy with Random

3.6 Detailed analysis using the IMDB dataset

For the detailed Analysis on IMDB dataset, we give 128 trials to every search algorithm. Each of these trials can learn on up to 4 epochs.
We repeated this process on 4 experiments to reduce the influence of noise, and also to quantify variance. Note that we used the entire IMDB dataset, with 200 000 cinema reviews. Despite reducing the length of reviews to the 200 first words, training was so long that we needed to use free GPUs from Google Collab.
We use MAE as a metric.

3.6.1 General Informations

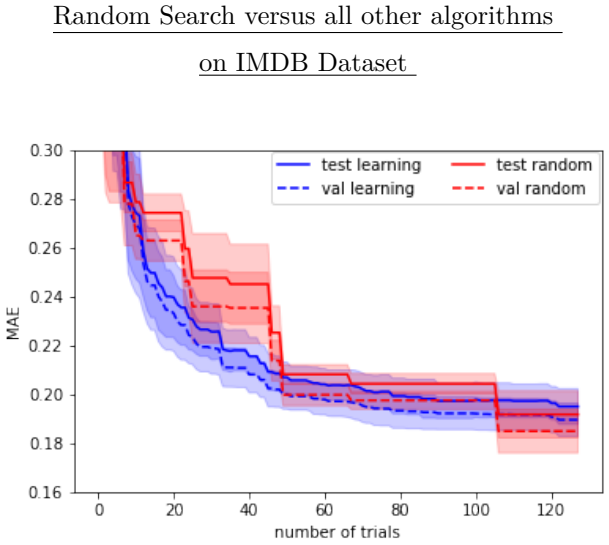


Figure 26. Here we plot the averaged test and val accuracy of Random Search, and we compare it with the mean of all other adaptive algorithms (learning algorithms). Color areas are σ wide.

As an introduction figure, we show in Figure 26 the comparison between the random search test and validation MAE, with the mean of all other algorithms's test and validation MAE.
For the first time, random is better than the average of the learning algorithms.

3.6.2 Test and validation accuracy

We propose in Figure 27 and Figure 30 respectively the test and validation results for all algorithms.
In Figure 27, random-search has a MAE bigger than the best error by at least 0.02. This corresponds to a probability prediction approximately 2% closer (82.5% and 84.5% respectively).
which has the same axis scale as Figure 13, shows that random search has the worst results for validation accuracy. Looking at validation accuracy (Figure 30), we see that most algorithms find "good" (MAE below 0.25) configurations after 20 trials, and start converging after 50 trials. These two plots permit an overall comparison. HyperOpt and BayesOpt are the best, Random is slightly better than the average, Zoopt is catastrophic for this dataset. We understand that Random is below the average of adaptive algorithms (Figure 26) because Zoopt has very bad results, which moves significantly the average. We do not know how to explain the superiority of random search compared to some adaptive algorithms. Maybe the hyperparameter space contains a lot of false local minimas, or is very noisy? Among all dataset analysis, Random was superior to the average only in this one.

3.6.3 Variance per Search Algorithm

To reduce information overload, we will plot the recommended metrics only for the top 4 performing algorithms. All axis are the same.
See Figure 28 for the plot of HyperOpt, in Figure 29 for Bayes, in Figure 31 for Ax, and in Figure 32 for Random. Ax seems to overfit. It could have been the best otherwise.

3.6.4 Comparison with previous datasets:

- Variance is small as observed for TREC and FMNIST analysis.
- Random continuously improves as usual (no convergence).
- The difference between test and validation MAE for a same search algorithm is very small. It is relatively the smallest compared to previous datasets.

Test Accuracy

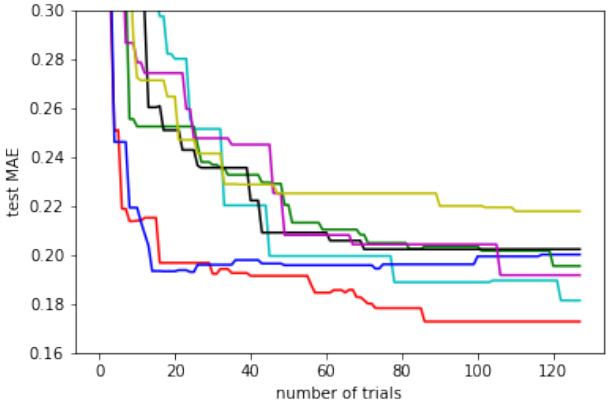


Figure 27. Here we plot the average test accuracy of the best trial obtained so far per search algorithm as a function of the number of trial.

Validation accuracy

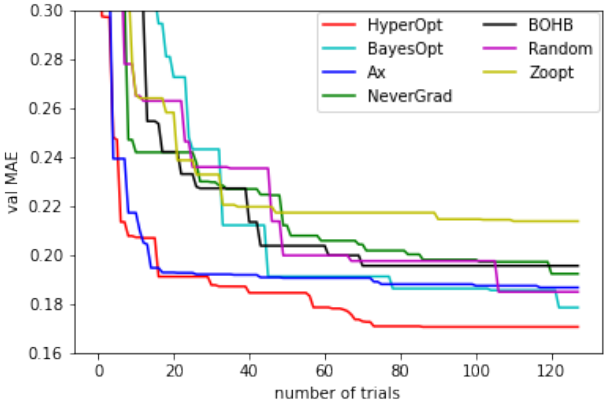


Figure 30. Here we plot the average validation accuracy of the best trial obtained so far per search algorithm as a function of number of trial.

Tests and Validation MSE per algorithms with σ narrow area on IMDB Dataset

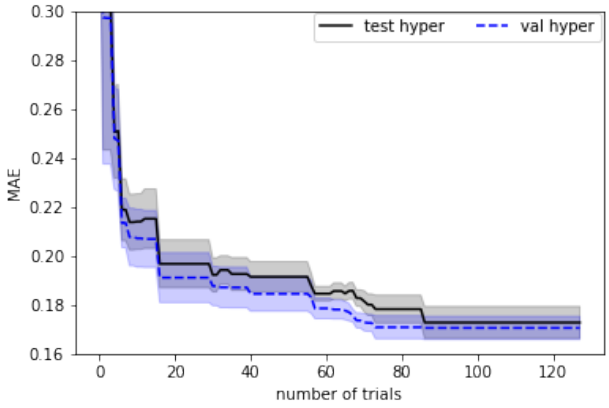


Figure 28. Validation, Test, Variance accuracy with Hyper

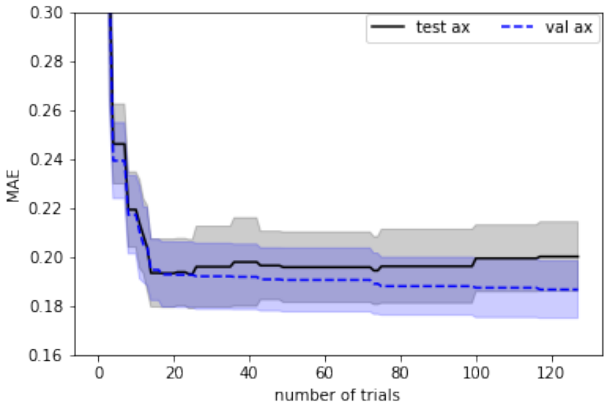


Figure 31. Validation, Test, Variance accuracy with Ax

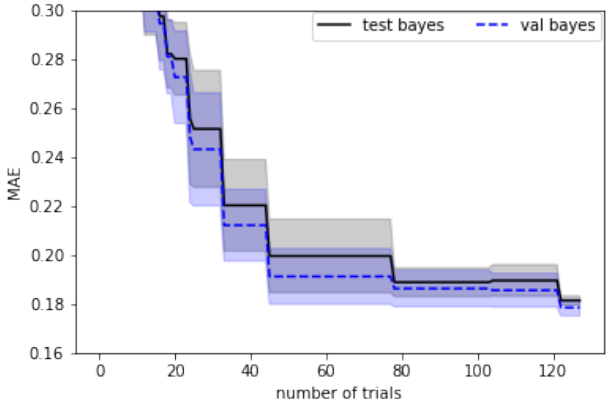


Figure 29. Validation, Test, Variance accuracy with Bayes

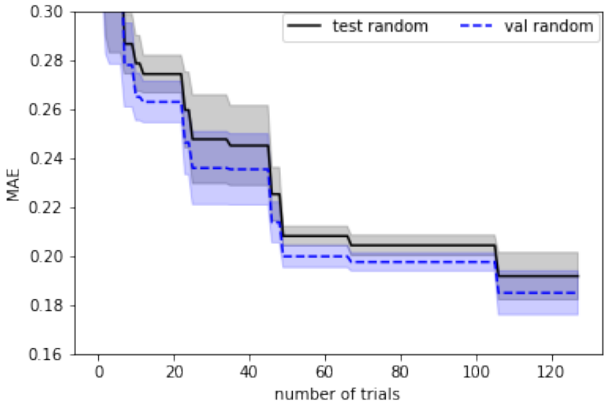


Figure 32. Validation, Test, Variance accuracy with Random

3.7 Detailed analysis on GAN using the MNIST dataset

GANs are very different to regression or classification tasks performed in the rest of this benchmark. We think it is important to have also a very different problems to see how the search algorithms will perform in completely different contexts: For this problem we changed the search space: it is now about optimizing the *learning rate*, the *first beta parameter*, the *weight decay*, and the *optimizer* for two models: the generator and the discriminator. That is a search space with 8 dimensions.
Following the same philosophy, we used a different scheduler: the population based scheduler, it gives well performing trials configurations to bad trials; and perturbs configurations of trials before training them. See 68Appendix B for more details. For short, population-based scheduler will do intensive *exploitation* as a lot of trials are going to take hyperparameter combinations near known good combinations. We expect the search algorithms to have added value on *exploration*.
Note we could not test BOHB as it is the only search algorithm that uses its own scheduler. We used TwoPointsDE to replace it. GAN do not use the concept of validation and test dataset. Due to the Population Based Scheduler, we can not plot one Inception Score for a given trial, as we would not know what value give to a trial: Should it be the Inception Score when it had the hyperparameters chosen by the search algorithm? Or once the PB scheduler choose to give it a better-known combination from a better performing trial? Or should it be the score once it has finished training? We thought the best way to handle this specific particularity was to plot as x-axis not the **score of every trial once they have finished training** as we have been doing for now, but rather the **score of every trial every time it is trained once** (and can potentially be perturbed). If n is the number of times a trial train, our plot should contain n times more values on the x-axis than before for the same number of trial searches for one search algorithm.
Each trial could learn up to 100 times, and a search consisted of 128 trials. We repeated the experiment 4 times to reduce noise and compute variance.

3.7.1 Overall Inception Score

As an introduction figure, we show in Figure 33 the Inception Score of all scheduler. Once again, random is below average, and Bayes and Hyper are clearly better. The score difference between Random and the best algorithm is bigger than 0.3 (relative difference of 5%). We observe that most algorithms find "good" (IS above 5.6) configurations after 900 trainings, and start converging after 5000 trainings.
The curve of HyperOpt is intriguing as it is bellow all others algorithms, and only surpasses all of them at the end. We will explain this result in paragraph 3.7.3.

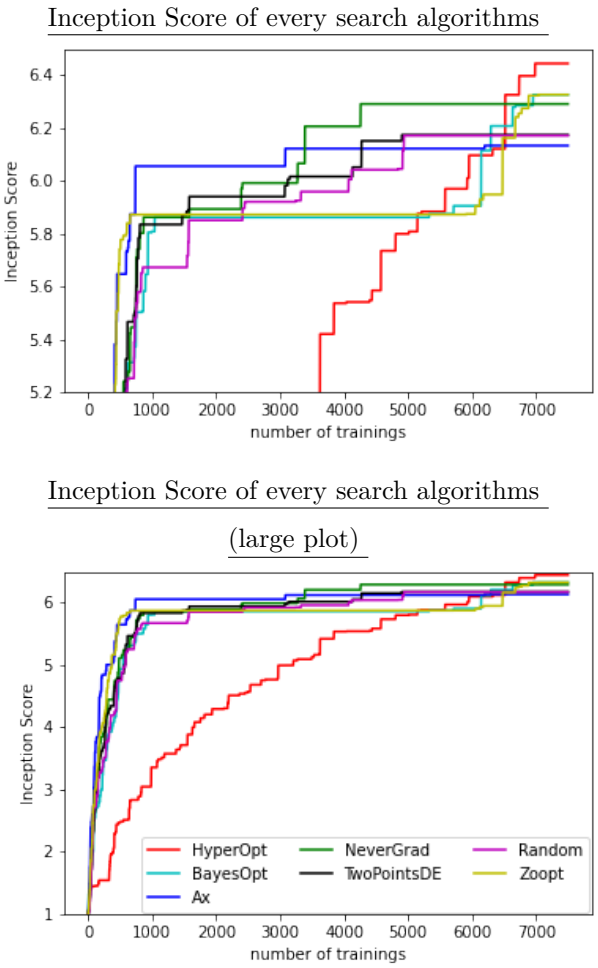


Figure 33. Here we plot the best obtained Inception score among all trials trained among all search algorithms, as a function of number of trial trainings.

3.7.2 Quantitative results:

We show in this section quantitative results of the GAN in Figures 34, 35, 36.
As we can observe on Figure 35, generation model with an Inception Score of 1 generates outputs alike random input. Inception Score of 6 is way closer to the original inputs: the background is uniformly black, and most shapes look like digits. But progress is still possible.

3.7.3 Variance per Search Algorithm

We present on Figure 37 and 40 the best obtained Inception Score with variance in function of the number of training for all search algorithms. Each Figure contains the half of the algorithms. Random is second worst. The best 3 algorithms are Hyper, Bayes, and NeverGrad. HyperOpt has the biggest variance. although it is the best search algorithm for this problem.

Figure 34. 64 input digits from MNIST dataset

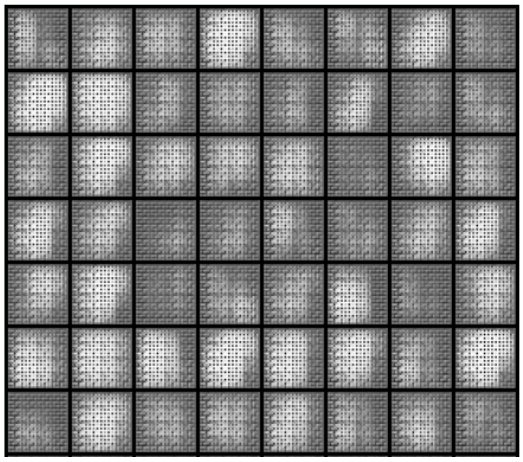


Figure 35. 64 output from our generator with Inception Score of 1

Figure 36. 64 output from our generator with Inception Score of 6

We see on Figures 38, 39, 41, and 42 the Inception Scores as a function of training of trials for Bayes, Hyper, NeverGrad, and Random search respectively. This permits to observe the distribution of Inception Scores in function of trainings, and hence permits to understand when an algorithm decides to explore, and when it decides to exploits. As an example, BayesOpt start very quickly by exploiting, which guarantees good results very quickly. Then it does a longer exploration followed by a longer exploitation which leads at the end to better results than the first quick exploration. This strategy guarantees fast good results, and eventually better results after a longer time.
Random and Nevergrad have another strategy. They alternate periodically between short exploration time followed by short exploitation time.
HyperOpt has the unique strategy of doing a unique long exploration, and then a unique very long exploitation. This latter strategy turns out to be the best. This strategy of unique exploration is only happening on this analysis and is caused by the scheduler. Indeed, HyperOpt usually converged quickly to good results for other dataset analysis.

3.7.4 Comparison with previous datasets:

- Variance is small as observed for TREC, IMDB and FMNIST analysis.
- Random, Bayes and HyperOpt have not clearly converged after the 128 trials.
- Here again, search algorithms matter to find significantly better solutions.

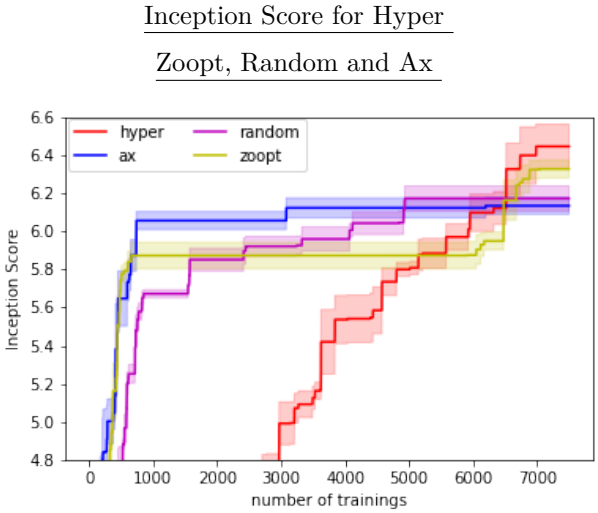


Figure 37. Here we plot the best obtained Inception score among all trials trained among all search algorithms, as a function of number of trial training with variance area σ wide.

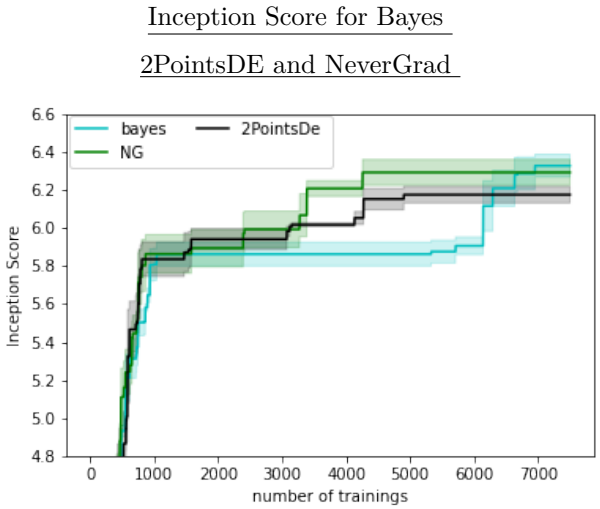


Figure 40. Here we plot the best obtained Inception score among all trials trained among all search algorithms, as a function of number of trial training with variance area σ wide.

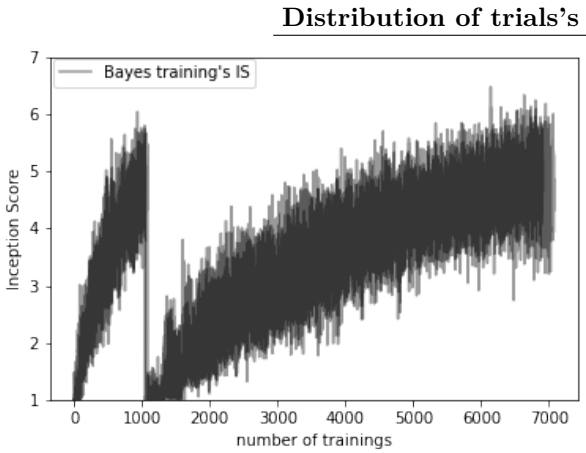


Figure 38. Distribution of trial's Inception Score on Bayes

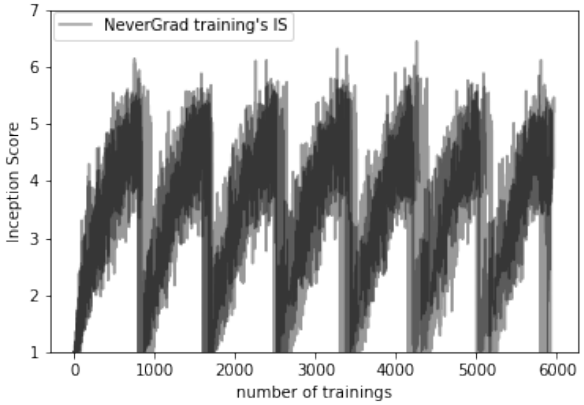


Figure 41. Distribution of trial's Inception Score on NeverGrad

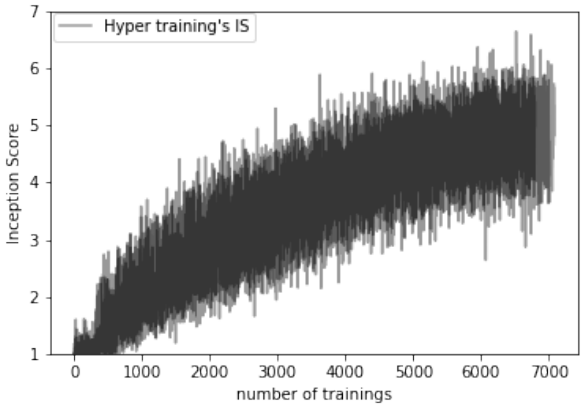


Figure 39. Distribution of trial's Inception Score on Hyper

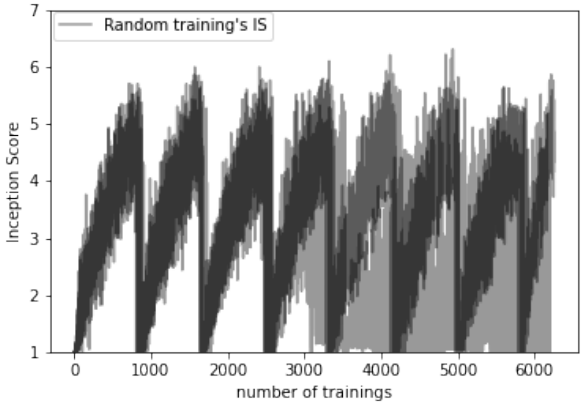


Figure 42. Distribution of trial's Inception Score on Random

3.8 Overall analysis and conclusion of the Toy benchmark

3.8.1 Execution time Analysis

We here want to compare the time it takes for search algorithms to find the best hyperparameter combination for one dataset. We study the time it takes to generate configurations for 256 or 128 models and train them on up to 20 epochs.

Parallel case:

We first study the parallel case, as all searches from all datasets but IMDB were made in a parallel environment with 8 CPUs. Remember that all search algorithms have schedulers that schedules their trials, and for example make a model learn up to 20 epochs only if it has a chance to beat the current best model. Search algorithms can have a big influence on their execution time. For example they can limit parallelism if they need to learn from trials results sequentially.

On Figure 43, we present in blue the search time of an algorithm relative to the search time of random search in the parallel case. We can see that most algorithms take a time very close from the reference of Random Search (which is perfectly parallel). Two algorithms take more time: AxSearch and BOHB. We can explain these differences: BOHB is the only search algorithm using its own scheduler. It is not surprising that it takes different time. Nevertheless it is surprising that despite the longer time it was granted, it was not able to perform better than other search algorithms in average.

AxSearch is the only search algorithm that explicitly limits parallelism: To enforce sequential treatment of the information, it used no more than 3 of the 8 thread. As all other algorithms used all 8 threads, it is logical that Ax is $\frac{8}{3}$ slower than the other algorithms. This is what we observe on the figure.

We propose in page 22 plots showing the distribution of trial accuracy as a function of training time, for each search algorithm in the parallel environment. These distributions depend on the scheduler, but since the scheduler makes its decisions based on the results obtained by the search algorithms, we notice disparities of distributions among schedulers. Intuitively, the best tendency curve has a very high accuracy for a very small execution time (it has maximum area under the curve). AxSearch is the best considering this criteria. BOHB and BayesOpt are the worst.

Sequential case: For IMDB dataset we used Google Collab free GPUs, which are fast and perform sequential trial search. Please look at the red bars in Figure 43 for the sequential time distributions. We observe fewer disparities in the distributions.

Surprisingly, some algorithms are faster than random, which should not be possible. This is because RandomSearch chooses randomly and uniformly a model between CNN, GRU, and LSTM. We know that CNN model is very fast compared to RNN model, as RNN model has a run time proportional to the number of words per input (200). Hence the other algorithms than RandomSearch can be faster if they choose CNN more often. We propose in page 23 plots showing the distribution of trial MAE as a function of training time, for each search algorithm in the sequential environment. This time the best tendency curves are the one that find good (small) MAE in small execution time. This corresponds to having a minimum area under the curve. Random and BOHB are the worst as they have the smoothest curves. Ax is here again the best.

We understand here that AxSearch is the best search algorithm for getting in average the best results after the smallest execution time per trial. It however is slower in a parallel case as AXSearch limits parallelism.

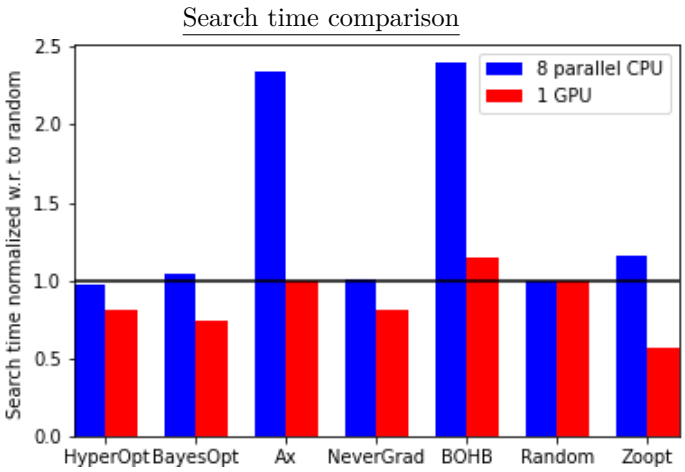
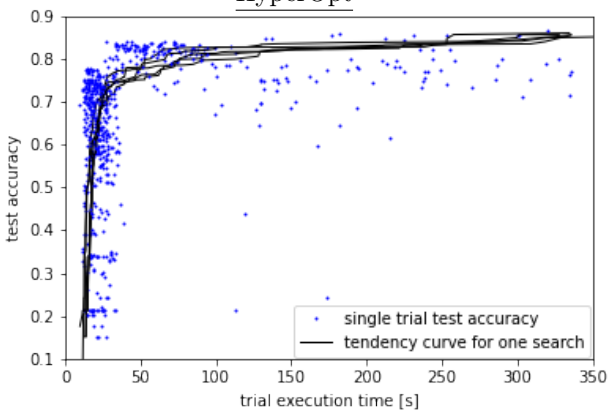


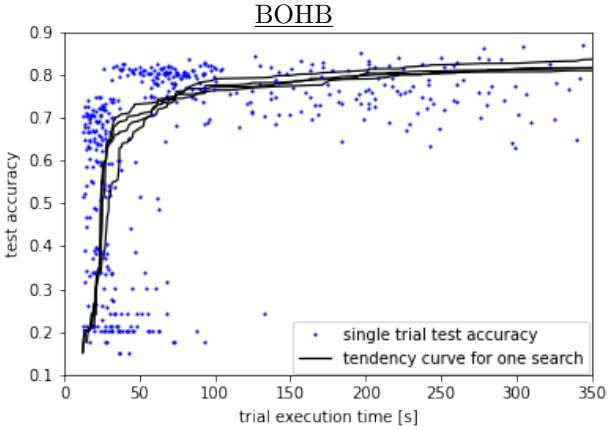
Figure 43. Relative execution time comparison of searches among all algorithms for parallel and sequential case.

Test accuracy per trial as a function of execution time with tendency curves (parallel environment)

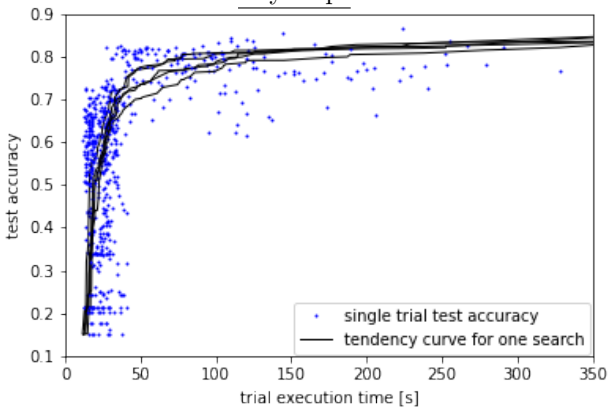
HyperOpt



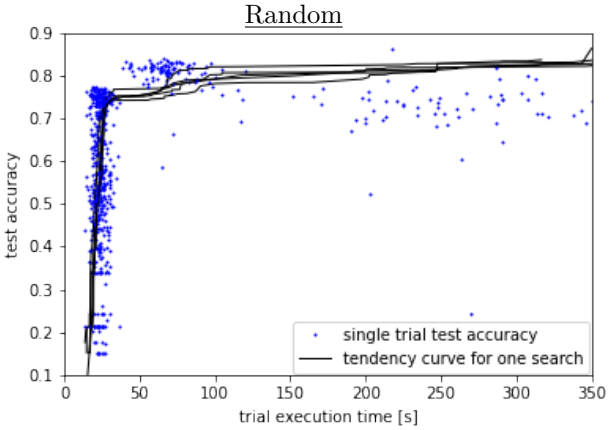
BOHB



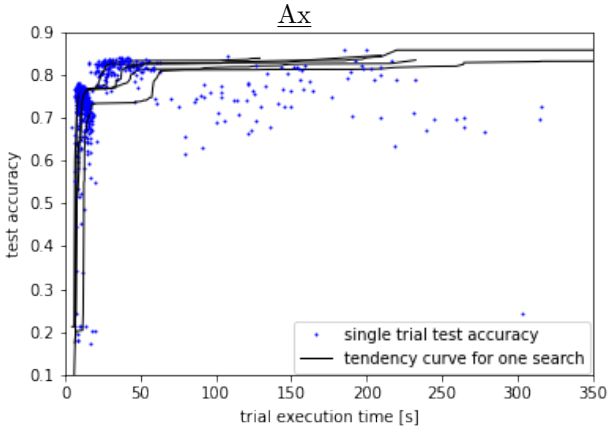
BayesOpt



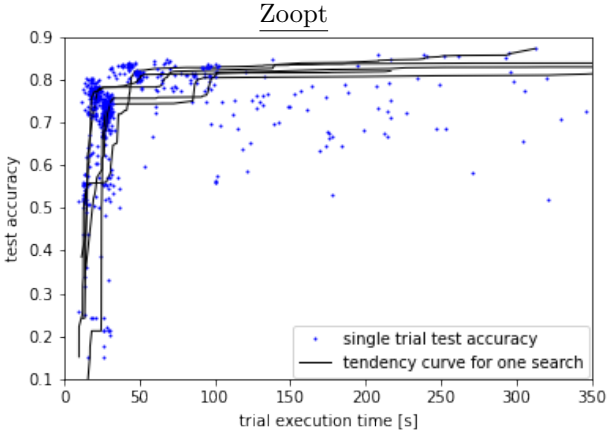
Random



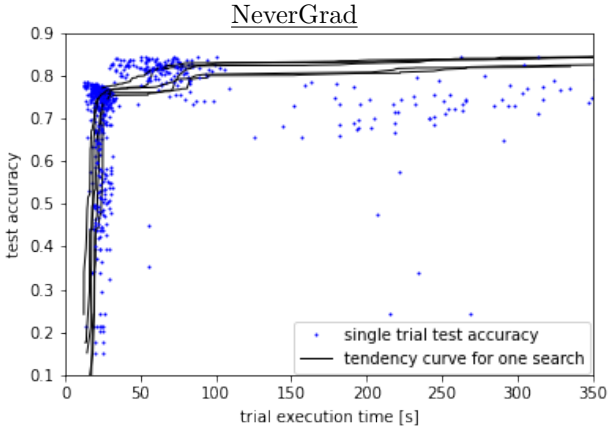
Ax



Zoopt

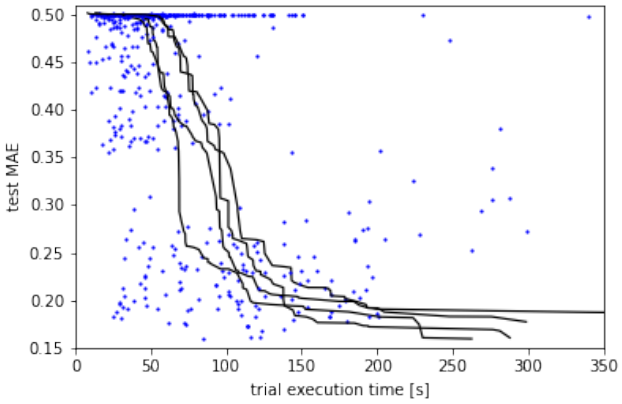


NeverGrad

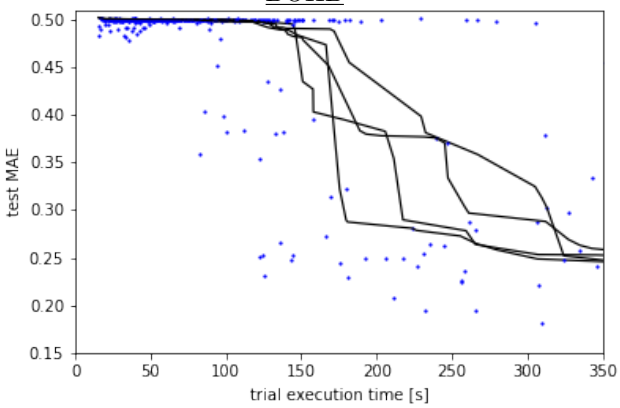


Test MAE per trial as a function of execution time with tendency curves (sequential environment)

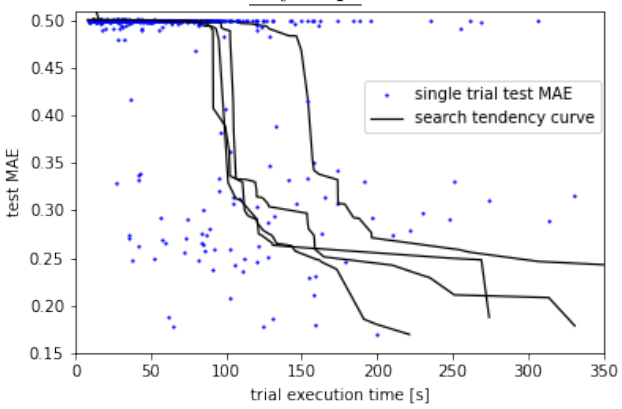
HyperOpt



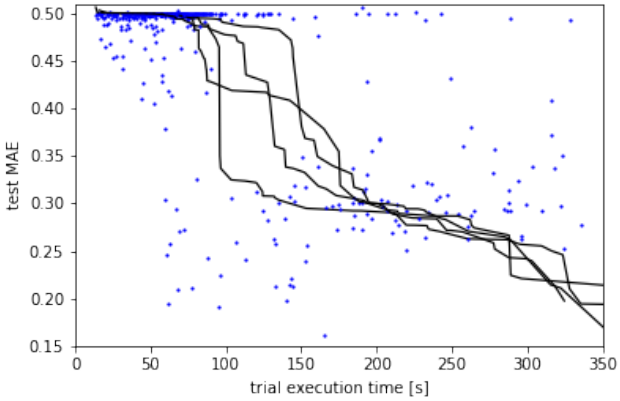
BOHB



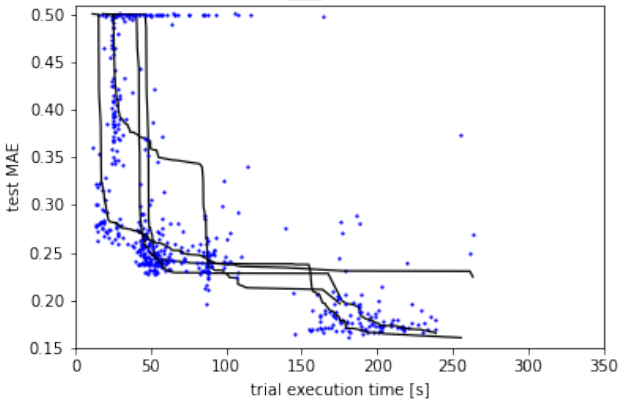
BayesOpt



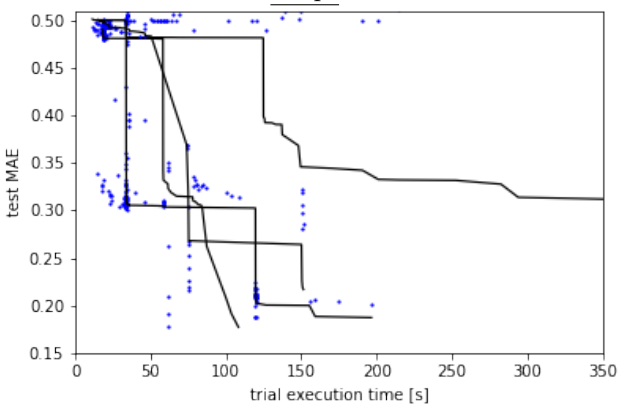
Random



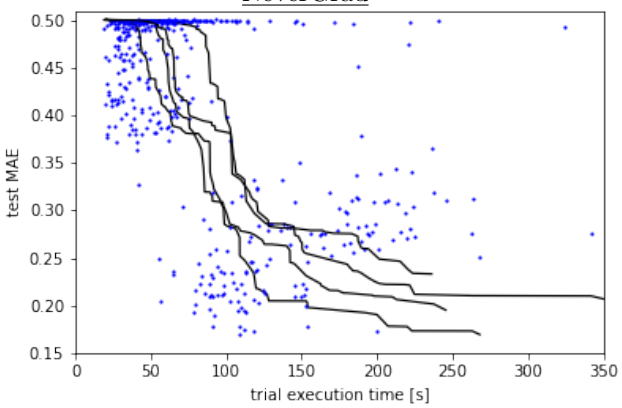
Ax



Zoopt



NeverGrad



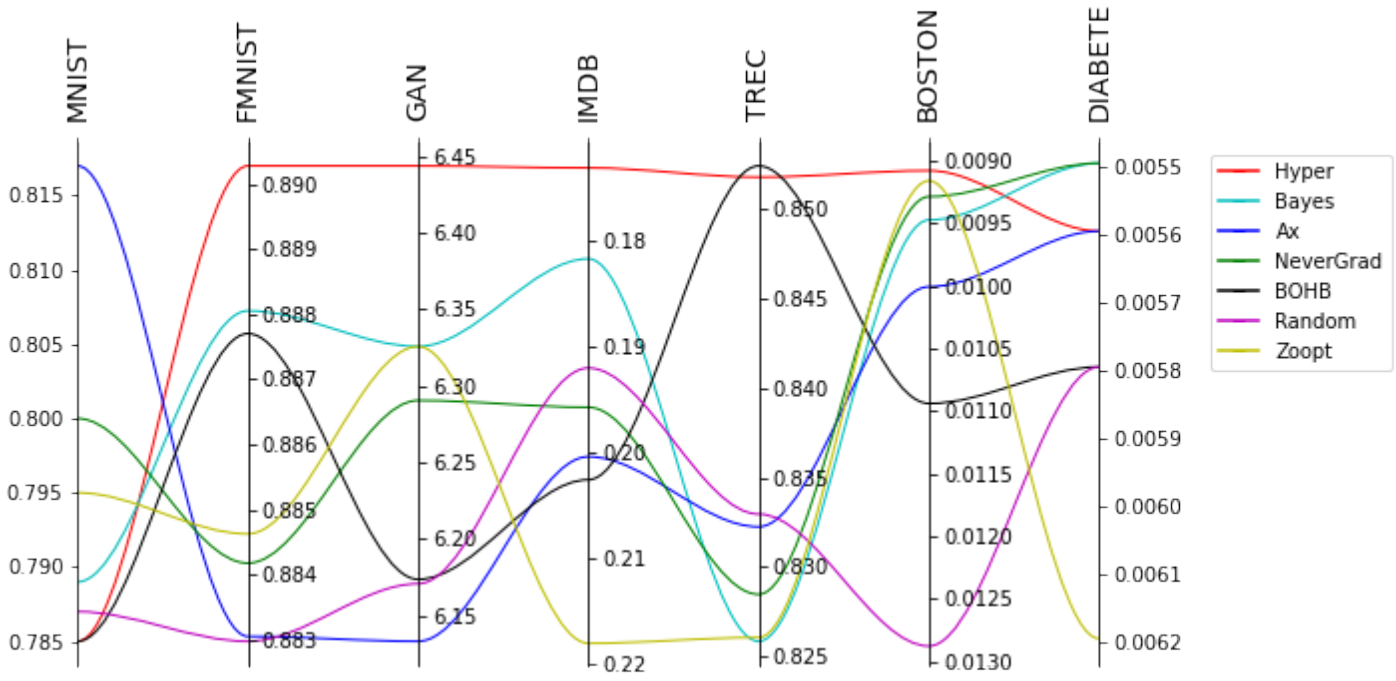


Figure 44. y-axis is either accuracy or minus error obtained from test values.

3.8.2 Dataset overall performance

We finally compare all search algorithms among all datasets. Please look at Appendix A for the analysis of MNIST and Diabetes datasets, as well as additional analysis for FMNIST and Boston Dataset using a smaller number of epochs.

Here are the conclusions we can make from this first batch:

- Figure 44 presents performances of all algorithms on all datasets. A clear tendency is that HyperOpt is really above all other algorithms. Bayes is second. Last place goes to Random as expected. Random is above the average of other search algorithms only once .
- Most search algorithms converge quickly to good results, and have low variance. This means they are regular and we can suppose with high confidence that results obtained would not be far different if we re-run a new identical experiment.
- Search algorithms do not cause a significant execution time overhead compared to RandomSearch.
- We recommend using HyperOpt when tuning hyperparameters as it is the best performing. We also showed that the advantage of HyperOpt was significant (the best configuration found by RandomSearch has a relative difference of -6% with the best configuration found by HyperOpt in average and -2% in median). This shows that Bayesian Optimization with Tree Perzen Estimator is the best concept for hyperparameter tuning compared to other concepts we presented.

4 BENCHMARK USING MORE COMPLEX MACHINE LEARNING PROBLEMS

4.1 Introduction

We will now use the search algorithms to perform hyperparameter tuning optimisation on more complex problems that require longer training. For this reason, we will only select an interesting subset of the search algorithms analysed before. We will keep HyperOptSearch as it was the best algorithm, BayesOptSearch as it was second, and RandomSearch, as it is our initial measure of reference.
From now on, comparisons will be presented as a function of time, as execution time is the final concern is real life. We concluded after first benchmark tool analysis that HyperOpt was the best search algorithm. We will in this section compare different strategies to find the best configurations quicker:
Up to now, we were facing simple problems requiring a low number of iterations before convergence of results. Hence our strategy was to use search algorithms allowing trials to train up to (or at least close to) convergence. This method has the shortcoming that if the problem requires very long training, we would be able to test only a few different configurations of hyperparameters. We propose a new strategy to counter this problem: we will allow a search to train on a small number of iterations (not enough for convergence), and then train until convergence only the model with the best configuration obtained.
This technique will work perfectly if trials comparison scale in time (trial a is better than trial b for one iteration implies trial a is better than trial b for every iteration), but in reality, the risk is that this new strategy selects fast learning configurations that are very good for a small number of iteration, but that does not scale well with the number of iterations. Another risk is that the small number of iteration increases noise on results. We will determine if this strategy is worth it.

4.2 Detailed analysis on VAE with CIFAR dataset

4.2.1 Introduction

The CIFAR dataset consists of 60 000 low quality color images (32×32 pixels) of animals and vehicles. Our task is to do image generation using VAE. See Krizhevsky (2009).
VAE, introduced by Kingma & Welling (2014), works the following: It will first significantly reduce the dimensional space of the input into a smaller space (we say it encodes it to the latent space), and the VAE will then try to recover (decode) from the low dimension data the initial input. The VAE can learn both on the encoding and the decoding, and the idea is that as we reduce the input to a smaller dimension, the VAE will have to keep only important information when encoding.
We use the ASHA scheduler, and GPUs from Google Collab to search best hyperparameter configuration in 6 hours per algorithm.
We will test the new strategy evoked in 4.1: We will execute the proposed strategy for a maximum number of iterations per trial allowed of 3 for first experiment, and 20 for second experiment, and we will compare how best configuration found in each experiment learn with more iterations. As every experiment has the same total execution time (6 hours), the experiment with 3 maximum iterations will generate more hyperparameter configurations. The search space has 6 dimensions.

4.2.2 Analysis

We show in Figure 45 the error of best trial found by different search algorithms for the experiment allowing 3 maximum iterations. We see that RandomSearch did not converge quickly, and only found results close to results of HyperOpt for much larger training time. The relative difference is however almost not significant: 1% after one hour of training and 0.5% after 6 hours of training. The plot for experiment with 20 iterations was very similar. The convergence starts after approximately 1 hour, which corresponds to 64 trials. We propose in Figure 46 the distribution of BCE in function of trainings for the search allowing maximum 3 training iterations per trial. e showed for the GAN problem the distribution of the trainings for the population-based scheduler, but we never presented the distribution when using the ASHA scheduler.
We observe that there is a huge distribution gap between RandomSeach (most black lines stop around loss of 1940), and BayesOpt (most blue lines stop around loss of 1920), and a significant gap between BayesOpt and HyperOpt (most red lines stop around 1910).

We take the hyperparameters from the best trial obtained for each search, and train it for 200 iterations. We present in Figure 47 how the best model chosen after doing first experiment with 3 maximum iterations performs

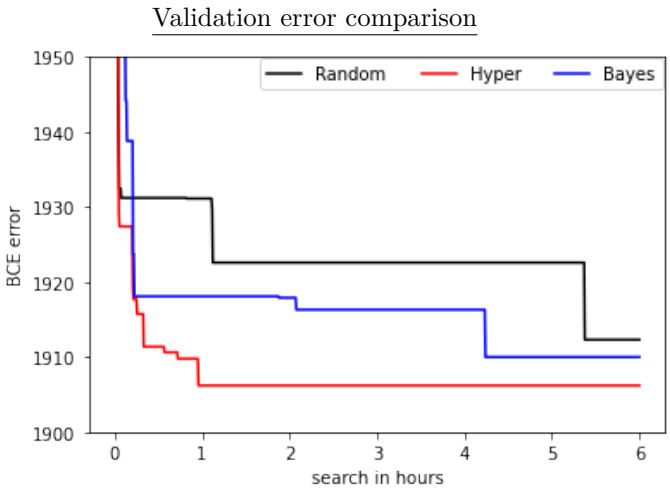


Figure 45. Comparison of error on trials searched by 3 different search algorithms on a very limited number of maximum iterations (3 epochs).

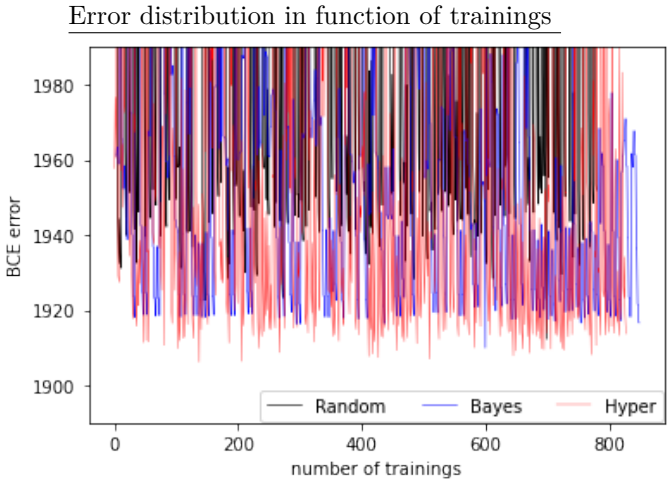


Figure 46. Comparison of the distribution of trial error as a function of trainings for 3 different search algorithms on a very limited number of maximum iterations (3 epochs).

compared to the best model chosen after doing second experiment with 20 maximum iterations. The scale on the y-axis is the same than for figure 44. We only observe a difference of 0.1 % between the two curves (The model obtained from 20 iterations is the best). We empirically showed that it is similar to perform search allowing a maximum number of 3 and 20 training iterations for this problem.

4.2.3 Qualitative results:

We present results obtained from the best VAE model obtained from all searches. First we will show how the VAE can reconstruct outputs looking like inputs, and second how it can create original outputs from a random vector in the latent space. In order to have better results, we did not choose a random vector, but a random vector

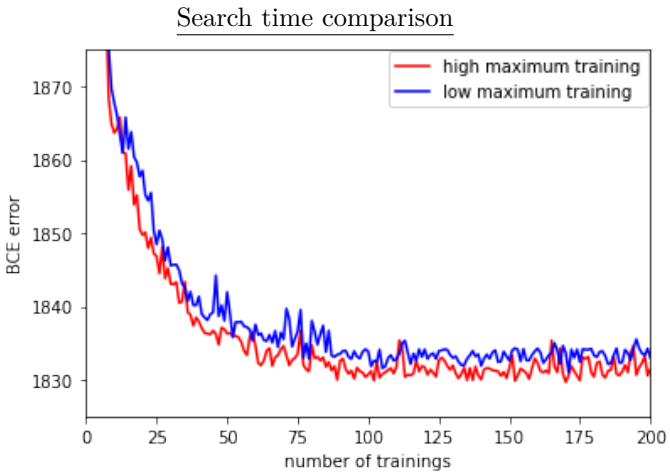


Figure 47. Comparison of test BCE error for the best model obtained from experiment with 3 and 20 training iterations.

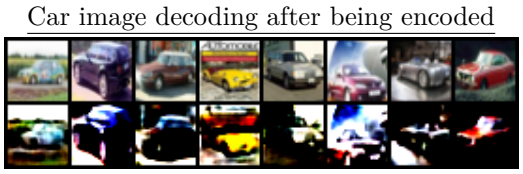


Figure 48. Model trained with 200 epochs. Top images are inputs from the dataset, and bottom images are obtained after encoding and decoding the corresponding input.

in the subspace of the latent space that spans one of the classes. We chose arbitrarily the class of cars. For consistency reasons, we show reconstructions of cars as well.

This generation task is harder than MNIST generation as for MNIST, images borders were always black and color distribution was far from uniform (pixel was either very clear of very dark), effectively reducing the complexity of the encoding and decoding. Here the 10 classes are all very different. Inside a class, images are also very different as they can show oddly shaped objects from the same class with different angles, with different backgrounds...

We observe on Figure 48 that VAE is good for generating output looking like inputs, as it regenerates outputs with close colors and similar shapes. It is however not able to reproduce perfectly the input image.

We propose in Figures 49 to 51 new images generation, obtained by decoding 64 random samples from the subspace spanning cars in the latent space. The first generation on a poorly trained model (Figure 49) is only able to produce smooth shapes and bicolor images. The second generation(Figure 50) can create details, and some images look like cars. The last generation (Figure 51) produces very complex shapes, but not so many looking like cars. We are surprised that recurrent shapes such as wheel are not very represented.

Car image generation with poorly trained model

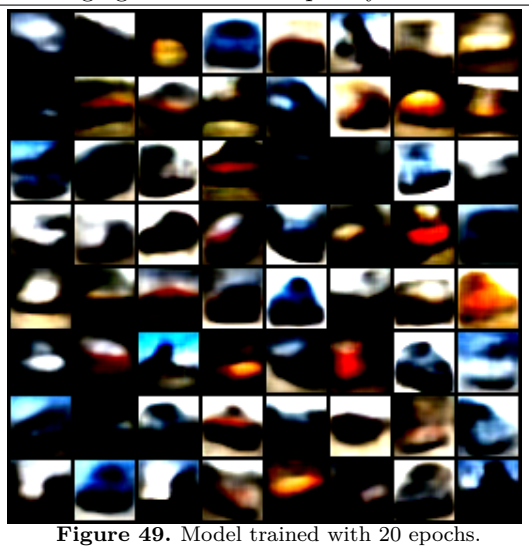


Figure 49. Model trained with 20 epochs.

Car image generation with correctly trained model

Figure 50. Model trained with 100 epochs.

Car image generation with highly trained model

Figure 51. Model trained with 400 epochs.

4.3 Detailed Analysis on MTM with the Hotel Reviews Dataset

We will now compare search algorithms and the strategy presented in 4.1 on an state of the art problem. We will use a novel model: MTM which is the current state of the art solution for finding the best sequence of words used to predict sentiment analysis relative to a specific aspect of a text. This model summarizes a text according to a specific aspect. It is based on the work Multi-Dimensional Explanation of Target Variables from Documents from Diego Antognini. See Antognini et al. (2019). The model first passes the input review through the embedding layer (turning discrete words one-hot encoding into vectors in a continuous vector space), such that some meaningful properties of the words are conserved. Masks are soft multi-dimensional masks: The masker will compute the probability distribution of every word over all target set plus one set for irrelevant words such that the sum of the probabilities equals 1 for one word. Masks are then applied to the reviews. An encoder will then learn a representation of the whole review, and a classifier will finally do target predictions. The dataset "Hotel rec" is the largest existing dataset of textual reviews in the hotel domain. Please see Antognini & Faltings (2020) for detailled presentation. It consists of textual review of hotels from Trip Advisor, including 5 stars ratings regarding 5 aspects: Cleanliness, Value, Location, Room, and Service. The dataset we used in this problem was a subset of the dataset, with 140,000 reviews.

4.3.1 General information

This optimization problem is the toughest that we faced in this project, as the model is the most complex, the dataset is large (3.4 G), the training time is long. We decided to tune as many parameters as possible, which led to a search space in 18 dimensions (we included 2 useless dimensions as a search algorithms should not struggle because of useless dimensions). The search space is the most complex we faced as well. in order to handle the computing complexity of the problem, we used the ICCluster from EPFL, allowing us to use two modern GPUs and 48 CPUs for 48 hours.

We will do a comparison of the search algorithms using ray tune, as usual, but also compare these results with those of the original paper, which used an efficient random search implementation without ray tune.

In his paper, Diego Antognini describes his tuning strategy:

> We ran all experiments for a maximum of 50 epochs with a batch-size of 256 on a Titan X GPU. We tuned all models on the dev set with 10 random search trials. Most of the time, the model converges under 20 epochs

According to the paper, 10 different trials have been

trained for a total of approximately 200 epochs. The best configuration obtains **90.79%** as a **F1 validation score** on the hotel dataset.
Our tuning strategy will be the following: We will exaggerate the strategy evoked in 4.1, as we will now evaluate trials on only one epoch each. The dataset consists of 200,000 reviews, so one epoch is already considerable. We will then compare results with those of the original paper.

4.3.2 Results and comparison with papers results

We first noticed that the training time is much longer using any search algorithm in ray tune, compared with the random search implementation from the paper (that does not use ray tune). Indeed, a training on one epoch takes 5 min with the original random search implementation, but takes 30 min with ray tune. The training time among search algorithms in ray tune is the same. A training time slower by a factor **6** is extremely bad, as the search algorithms added value is probably not efficient enough to compensate slowdown by a factor 6. You can see on Table 4 a comparative of the utilization of performance characteristic exploited by the two implementations. We did not notice similar overhead from ray tune implementation when using only CPUs.

Table 4 Informations obtained using command nvidia-smi.

Utilization	Direct RandomSearch implementation	Ray-Tune implementation
Volatile GPU	90-100%	40-60%
Max Batch	256	32
Dynamic GPU memory	5/12GB	2/12GB

We explain this difference because either ray tune is not able to fully exploit the GPU potential, or ray tune causes a dynamic memory overhead which limites the maximum batch size (maybe by enforcing the whole dataset/models weight to stay in the RAM, or maybe because it consumes a lot of memory to manage the trials?). Another explanation is that our tuning on 18 dimensions influences some parameters that cause longer training times (using GRU instead of CNN, more hidden layers...).

This unfortunate fact permitted us to do 6 times less trainings then expected with ray tune.

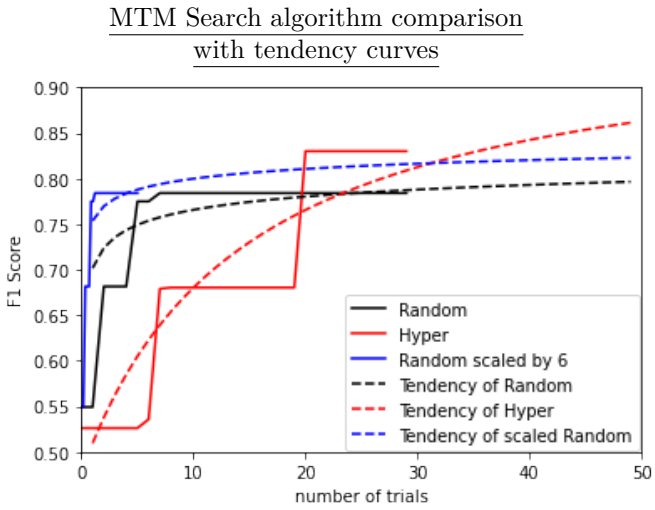


Figure 52. The metric here is the F1 score on the validation set. Trial trained with 1 epochs.

We can observe on Figure 52 the best yet obtained F1 score in function of number of trials for both HyperOptSearch and RandomSearch from ray tune. We were only trained on 28 trials. We propose converging fitting curves (the curve $f(x,a,b,c,d,e) = 1 - \frac{b}{d-(a+c\times x)^e}$ with $a,b,c,d,$ and e constant such that f is the closest to the original function $g(x)$) in order to have an idea of what the score curves would possibly have looked for more trials, but also to have an idea of what the curves could have looked if averaged over more experiments.
We also propose in blue the curve of a scaled-by-6 RandomSearch curve to get an idea of how would have performed the direct implementation of RandomSearch compared to the ray-tune search algorithms. Comparing the two algorithms from ray tune, we see that RandomSearch is better for the first 19 trials, then HyperSearch finds a far better solution which remains unbeaten afterwards. The fact that RandomSearch is better for a few iterations can be either due to noise (little number of trials and only 1 experiment) or because the search space is very complex, and HyperOpt takes time before finding ways to optimize Hyperparameters.
Now looking at the scaled RandomSearch curve (equivalent to the direct implementation of RandomSearch), we see that the scaled random search is better than hyperOpt search for approximately 30 trials searches. Finally we take the best trial we obtained, and train it for up to 30 epochs, to compare with results of the reference paper. For the Hotel dataset, the paper shows a *F1 Score* of **89.94** while our model gets **90.64**. This shows a relative difference of 0.8%. Furthermore, the author performed training on approximately 200 epochs, and we only trained on 28+30 = 58 epochs! Moreover, we showed previously that HyperSearch used to converge after 64 trials in case of search space of only 8 dimen-

sions. Since we have 20 dimensions here, we have strong confidence that HyperOptSearch did not converge with the 28 iterations of figure 52, and we are very confident that it would have found trials with F1 Score far above **90.64** if we had done more iterations.
The fact that RandomSearch and HyperOpt from Ray-Tune implementation were slower does not discredit the conceptual HyperOpt algorithm, but only discredit its implementation in raytune. We hence show that both HyperOpt and our strategy of allowing a low number of maximum training iterations are valuable even in state of the art problems.

4.3.3 Quantitative results

We propose here an example of the results obtained by the MTM model.

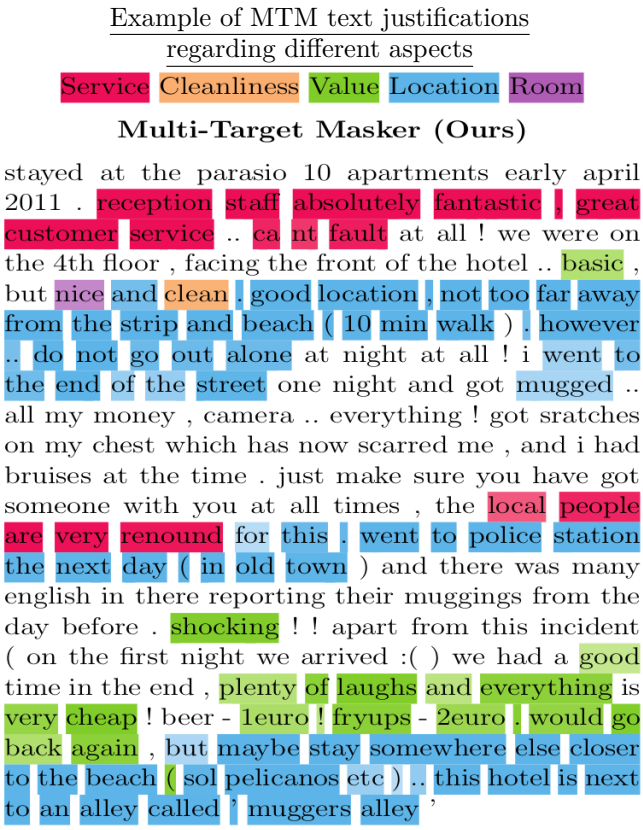


Figure 53. Shade colors intensity represent the model confidence toward the aspects.

Figure 53 shows an example of multi-target aspects obtained using model from Antognini et al. (2019) on the hotel rec dataset Antognini & Faltings (2020). The colors highlight the relevant justifications in the text of the corresponding aspect. We can see that MTM finds the most relevant sentences with very small amount of noise (highlighting irrelevant isolated words).

5 CONCLUSION

Our research allow us to reach several conclusions:

About search algorithms, HyperOptSearch is consistently the best search algorithm for same searching time in case of both toy and complex problems.
RandomSearch can reach similar qualitative results, but after much longer time as it converges very slowly to find equivalent results than HyperOptSearch.
Novel search algorithms do not cause significant time overhead compared to RandomSearch.

About execution time, an efficient strategy to find good trials with limited time is to allow a very small number of maximum trainings iterations per trial. This strategy finds slightly less good configurations, but far quicker.
We also saw that the Ray tune library can cause significant time overhead.
The number of configuration generated (number of trials) before reaching convergence does not depend on the problem, but can depend on the number of search space dimensions: convergence was reached in about 60 trials for 7 problems on 7 different datasets with search space of very similar dimensions. Number of trials generated before convergence also depends on the number of maximum allowed training per trials (smaller maximum causes slower convergence of search algorithms), and on the search algorithm: it seems that population based algorithms like CMA converge the fastest but with less good results. Random Search is the slowest to converge.

About schedulers, the ASHA Scheduler is consistently causing powerful acceleration of the execution time. Population-Based training is theoretically better, should therefore be used in certain cases (See Appendix B).

We generally advise the use of HyperOpt from the original library, assisted by the ASHA Scheduler, and to search best configurations with trials training on a reduced number of iterations.

A APPENDIX ON FURTHER TOY BENCHMARK ANALYSIS

We propose in this Appendix a lighter analysis on redundant datasets (MNIST and Diabetes), as well as an analysis of other datasets (FMNIST, Boston) with small maximum training iterations per trial allowed.

A.1 Fashion MNIST Analysis

FMNIST is compared with every search algorithms, on 256 generated trials, training each on up to 4 iterations.

1. Visual validation and global informations

Hyperparameter distribution over the search space for RandomSearch and an adaptive search algorithm

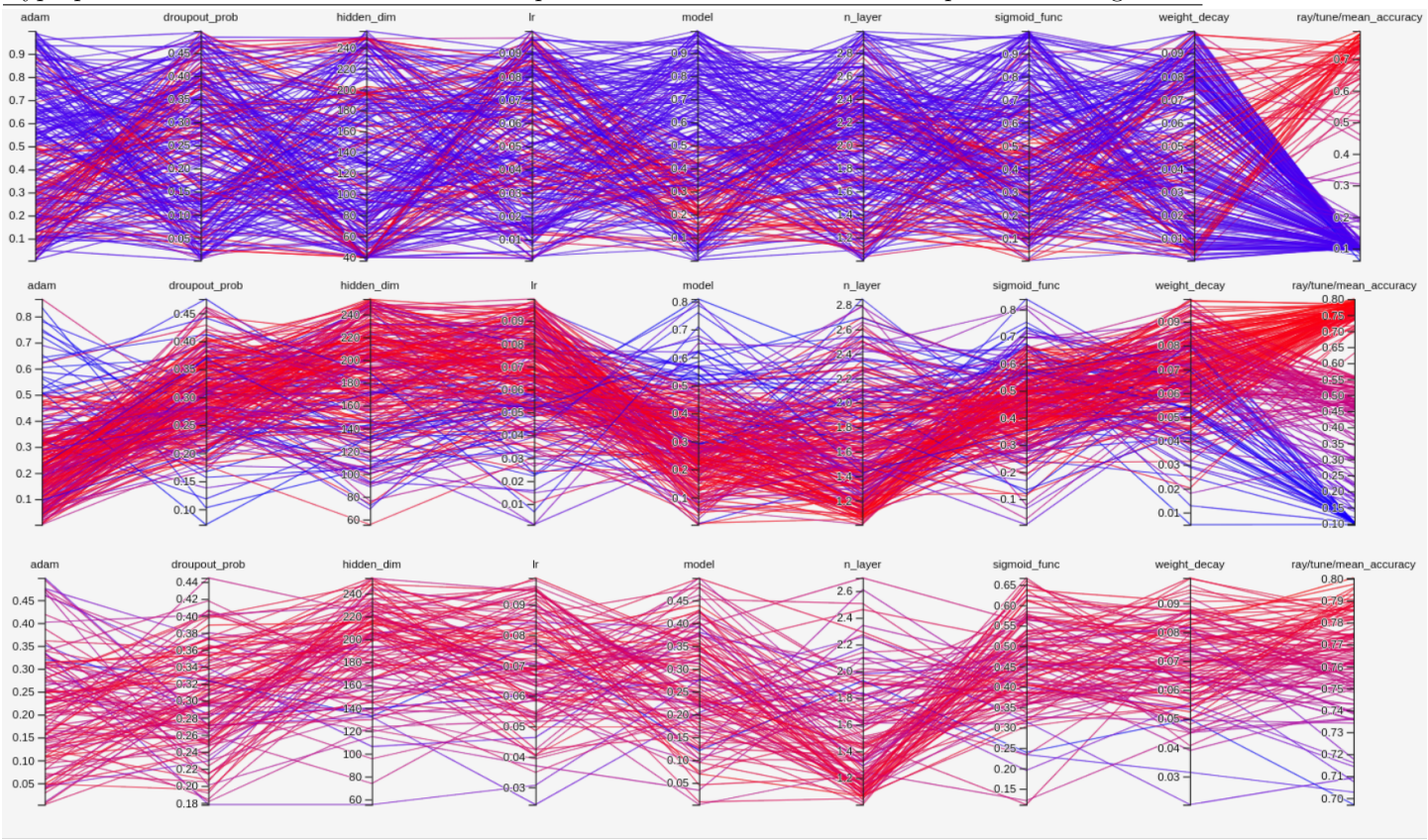


Figure 54. We here see the the distribution of hyperparameters per search algorithms. Every line is a trial. red or blue indicates whether it provides good or bad accuracy. The first is from random search. It corresponds to what we expected: we have a very wide and uniform looking distribution. The second one is from AxSearch, and is similar to the other adaptive algorithms: we have all trials going in the same good position, with still some wide space to search for new combinations (exploration). Finally the last one is still from AxSearch once we keep only "good" trials (the one with more than 70 % accuracy). This allows us to understand which are the good configurations for this problem.

We use tensor flow to get a visual representation on tendencies of hyperparameters distribution among trials. On Figure 44, we compare this distribution of hyperparameters assignments for an adaptive algorithm, and for random search. We expect random search to give uniform distribution, and the adaptive algorithms to assign a very specific distribution, giving the same (good) hyperparameter to a lot of trials. We use TensorFlow for this purpose.
From Figure 44, we understand that, to get good validation accuracy in Fashion MNIST, learning rate is good to be high (in [0.05-0.1]), as well as weight decay (in [0.05-0.09]). The best sigmoid function is Tanh, the number of dimensions must be high (most values in [200-240]), the number of hidden layers low (1 hidden layer: this is probably because we have so few iterations. we want to avoid underfit). Most interestingly, Adam and MLP seem to be not efficient compared to AdaBelief and CNN, this is the major factor for obtaining good results: in the third image from Figure 44, all trials with accuracy above 0.7 used Adabelied and CNN.
Dropout probability shall be in [0.2-0.4].

We propose in Table 5 the validation accuracy for the experiment we described in the introduction. Random search is the worst.
We can compare these values on 256 trials training on 4 epochs with the ones from first benchmark tool (FMNIST detailed analysis). (128 trials and 20 epochs): Best accuracy here is 82% and was 89% once trained on

Table 5 Presentation of validation results depending on algorithms.

name	mean	mean of 10 bests	best
BayesOpt	0.251	0.774	0.789
Random	0.25	0.772	0.787
Ax	0.597	0.798	0.817
NeverGrad	0.539	0.793	0.8
BOHB	0.492	0.778	0.785
HyperOpt	0.23	0.768	0.785
Zoopt	0.142	0.773	0.795

20 epochs, which is a huge difference. Note that good result in the literature is of 93% for classification for FMNIST so we are not very far despite a less complex model (Mani et al. (2019)).

2. Validation accuracy repartition over time

We propose in Figure 55 max validation accuracy as a function of number of trials generated. The objective being to be as much on the upper left as possible. We see that Ax and NeverGrad are both better and faster than random for this dataset. Convergence of results happens after 64 trials are generated.

3. Test accuracy

We trained the best configurations obtained for each algorithms, and trained them on bigger number of iterations. Observe results in . HyperOpt trial overfit , BayesOpt and Ax are good, and Random is once again below the average. Surprisingly, HyperOpt is not very good for this problem. We present in the second half of the table distribution of results from column n steps and column validation.

A.2 MNIST Analysis

Here we study the MNIST dataset.

1. Visual validation and global informations

We understand using TensorFlow (in bold informations different then for fashion mnist) that to get a

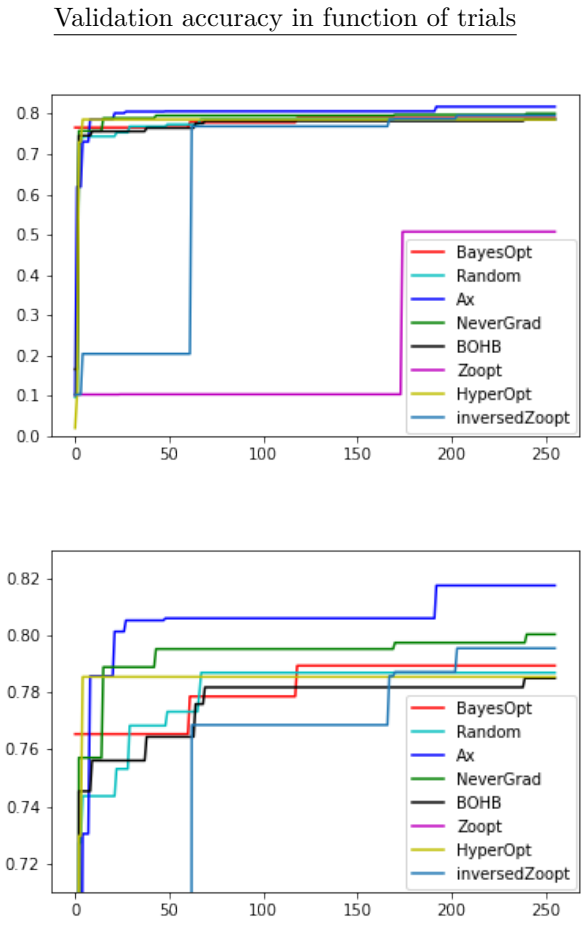


Figure 55. Here we plot the max validation accuracy already obtained (so maximum validation accuracy between this trial and all previous trials) as a function of trials number.
The lower image is the same plot with a focus on the accuracy in [0.71;0.83].

good validation accuracy in MNIST,with our model, **learning rate is good to be low** (in [0.0001-0.02]), as well as **weight decay** (in [0.0001-0.07]). The best sigmoid function is Tanh, the number of dimension and hidden layers doesn't seem to be that important. Adam and MLP are still not efficient compared to AdaBelief and CNN. Dropout probability shall be in [0.2-0.3].

As we can see on Table 7 random search is less good than almost all others algorithms. The importance of search algorithms is once more emphasized here: RandomSearch is almost 3% less good than the best search algorithm despite having results very close to 100%.

Table 6 Here we compare the validation accuracy of the best trial from each search algorithm to the test accuracy of a model with same hyperparameters.

name	validation	5 steps	10 steps	20 steps
BayesOpt	0.789	0.806	0.848	0.851
Random	0.787	0.781	0.823	0.835
Ax	0.817	0.75	0.81	0.852
NeverGrad	0.8	0.8	0.825	0.839
BOHB	0.785	0.781	0.815	0.83
HyperOpt	0.785	0.774	0.765	0.611 !
Zoopt	0.795	0.8157	0.837	0.84
Correlation		-0.405	0.107	0.396
μ of 20 Random		0.789	0.828	0.844
σ of 20 Random		0.012	0.014	0.01
μ of difference		-0.007	0.024	0.014
σ of difference		0.027	0.026	0.077

Table 7 Validation accuracy for MNIST.

name	mean	mean of 10 bests	best
BayesOpt	0.383	0.945	0.963
Random	0.354	0.932	0.944
Ax	0.888	0.959	0.969
NeverGrad	0.765	0.961	0.972
BOHB	0.348	0.944	0.956
HyperOpt	0.675	0.911	0.919
Zoopt	0.697	0.938	0.963

2. validation accuracy repartition over time

Validation accuracy in function of trials

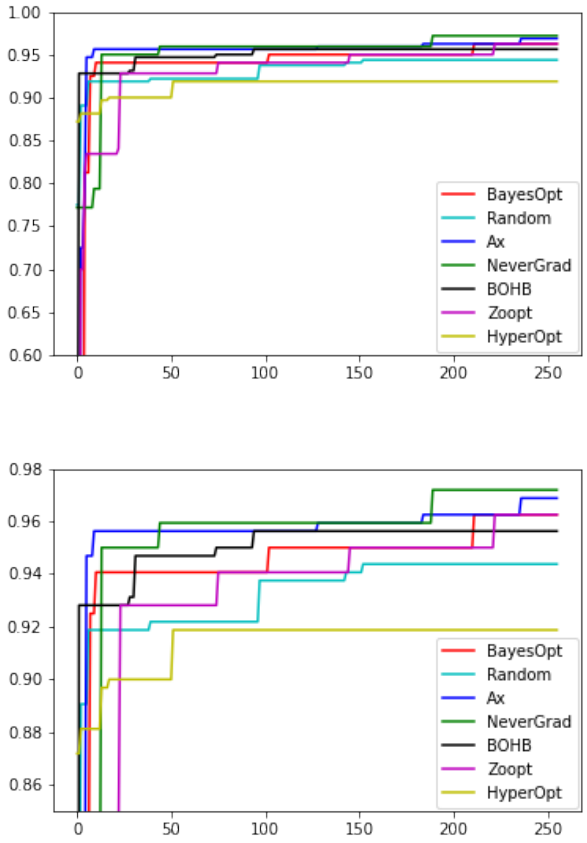


Figure 56. Here we plot the max validation accuracy already obtained as a function of trials number.

We see that all learning algorithms but HyperOpt are both better and faster than random for MNIST (see Figure 56). Convergence happens after 110 trials.

3. Test accuracy

We show on Table 8 the test accuracy of best trials once trained on more trials. All algorithms have a validation accuracy way above test accuracy. It is the role of the search algorithm to handle noisy inputs, this is why all search algorithms have a similar drop of result from validation to test (see "difference"). Once again the best opponent to Random Search is AxSearch on test data. HyperOpt is very disappointing.

Table 8 Here we compare the validation accuracy of the best trial from each search algorithm to the test accuracy of a model with same hyperparameters.

name	validation	5 steps	10 steps	20 steps
BayesOpt	0.963	0.916	0.931	0.906
Random	0.944	0.85	0.891	0.891
Ax	0.969	0.95	0.925	0.931
NeverGrad	0.972	0.869	0.875	0.881
BOHB	0.956	0.853	0.791	0.906
HyperOpt	0.919	0.775	0.825	0.763
Zoopt	0.963	0.73	0.825	0.888
μ of difference		-0.106	-0.089	-0.074
σ of difference		0.064	0.046	0.037

A.3 Diabetes Analysis

Here we study the Diabetes regression dataset. We choose MSE as loss function. Note that we normalised the data with mean 0 and std of 1. **Since all loss are below 0.01, we will present loss in this section in ‰**

1. Visual validation and global informations

We understand using TensorFlow that to get good validation loss in diabetes dataset with our model, learning rate is good to be high (in [0.05-1]), the weight decay around 0.03. The best sigmoid function is not relevant here as unused, the number of dimensions shall be around 140 and the best number of layers is 2. Dropout probability shall be low: in [0.1-0.2]. Best model is MLP (opposed to LogReg) and Adam is preferred over Adabelief! (This is the only time it is the case).

We show in Table 9 basic statistics for this problem. In particular, by looking at the huge mean results, we understand we had huge outlier values for BOHB, HyperOpt, Random. In the case of too big outlier values, we would replace them by 100,000. Random search is here also less good than any other learning algorithms.

Table 9 Validation loss for Diabetes, in ‰ with a normal distribution pre-processing.

name	mean	mean of 10 bests	best
BayesOpt	10.5	7.6	5.8
Random	91.2	8	6.7
Ax	10.5	8.1	6.5
NeverGrad	9.8	6.1	5.8
BOHB	3147.8	7.6	6.3
HyperOpt	400.3	8.5	6.7
Zoopt	10.6	8.5	8.1

2. Validation loss repartition over time

Validation MSE in function of trials

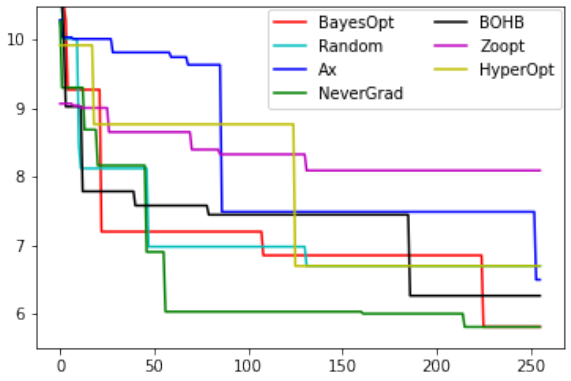


Figure 57. Here we plot the min MSE already obtained as a function of trials number.

We propose in Figure 57 the best validation MSE as a function of number of trials generated. Random is near to all other search algorithms. NeverGrad dominates here. Convergence happens after 90 trials.

3. Test accuracy

We can see on Table 10 that there is a 6% difference between best test accuracy (Nevergrad) and the one ofRandom.
All algorithms are better than random except for BOHB and Zoopt.

Table 10 Here we compare the validation accuracy of the best trial from each search algorithm to the test accuracy of a model with same hyperparameters.

name	validation	5 steps	10 steps	20 steps
BayesOpt	5.8	7.2	5.5	5.5
Random	6.7	6.1	5.9	5.8
Ax	6.5	8	6.3	5.6
NeverGrad	5.8	8.8	6.6	5.5
BOHB	6.3	7.7	5.6	5.8
HyperOpt	6.7	7.5	5.6	5.6
Zoopt	8.1	8	6.9	6.2
μ of difference		1.1	-0.5	-0.8
σ of difference		1.1	0.6	0.5

A.4 Boston Analysis

Here we study the Boston regression dataset. We choose MSE as loss function. We preprocessed the dataset by normalizing labels (mean of 0 and std of 1). **Loss is not presented in ‰anymore.**

1. Visual validation and global informations

Using TensorFlow, we understand that to get good validation loss in Boston dataset with our models, the learning rate is good to be moderate (in [0.02-0.04]), the weight decay around 0.04. The best sigmoid function is not relevant here as unused, the number of dimension shall be high: around 200 and the best number of layers is 1 or 2. Dropout probability shall be around 0.35. Most trials use LogReg as it provides low loss, but lowest loss are performed with MLP.

As we can see on Table 11, RandomSearch is notably less good than all other search algorithms but Zoopt and NeverGrad. We replaced extreme values by 100,000.

Table 11 Validation loss for Boston with standard normal distribution.

name	mean	mean of 10 bests	best
BayesOpt	17365	0.147	0.016
Random	25942	0.395	0.179
Ax	7910	0.372	0.022
NeverGrad	4264	0.241	0.122
BOHB	13429	0.118	0.034
HyperOpt	10581	0.124	0.021
Zoopt	8751	1.65	0.154

2. Validation loss repartition over time

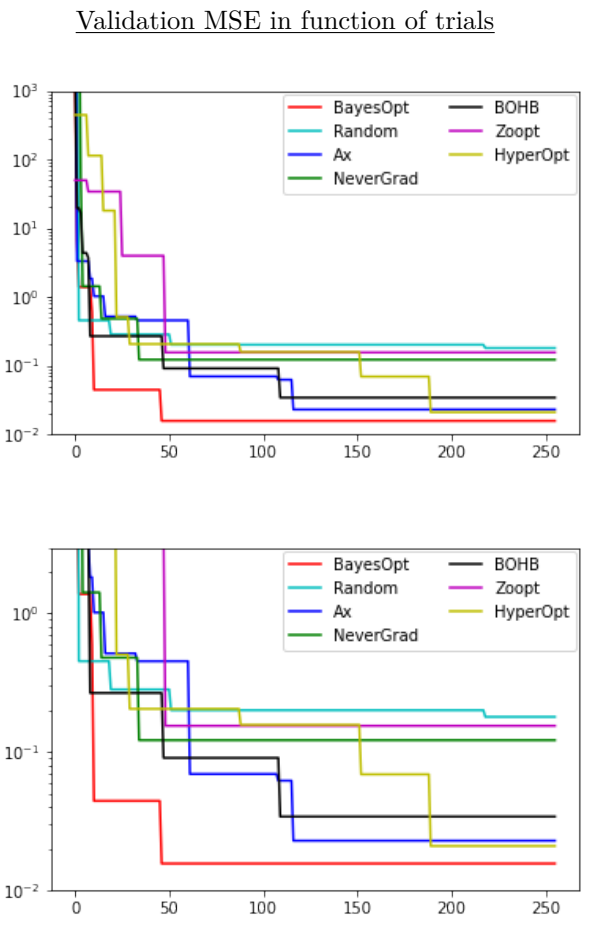


Figure 58. Here we plot the min loss accuracy already obtained as a function of trials number.
It is a log scale on the y-axis.

Convergence happens after 110 trials as we can see in Figure 58.

3. Test accuracy

Table 12 Here we compare the validation error of the best trial from each search algorithm to the test accuracy of a model with same hyperparameters.

name	validation	5 steps	10 steps	20 steps
BayesOpt	0.016	0.403	0.035	0.014
Random	0.179	6.036	0.855	0.124
Ax	0.022	0.169	0.069	0.034
NeverGrad	0.122	2.352	0.833	0.154
BOHB	0.034	0.045	0.025	0.021
HyperOpt	0.021	0.022	0.02	0.016
Zoopt	0.154	1.51	0.221	0.024

Table 12 shows test error of best trials once trained on a bigger number of iterations. Algorithms with bad validation error are far worst on test results (random and Nevergrad), and good algorithms are better. Error of RandomSearch is 8 times worse than the best error (BayesOpt). Random and NeverGrad are so bad because they both used to choose LogReg as a model. We do not observe overfitting. We compare these results with those of Boston analysis: error was 10 times smaller then!

A.5 Discussion

We make two conclusions from the problems analysed in Appendix A:

1. Reducing the maximum number of trainings allowed per trial causes a slower convergence of search algorithms.
2. HyperOpt is not excellent for all the problems of Appendix A: problems of low complexity and with a small number of allowed training.

B APPENDIX ON POPULATION-BASED SCHEDULING

Analysis of schedulers are not officially part of our project, but we still propose here a very small analysis as we think these concepts can have a very important impact to optimize the hyperparameter tuning.
We will compare the number of hyperparameter searches that can be done in function of the number of trainings in the following cases: without scheduler, with PBT scheduler, with Median scheduler, with ASHA scheduler.

B.1 Without scheduler:

Let us compare a regular search of ***n* trials** trained on ***m* iterations** each.
Any of our search algorithm tryies to do the following: Learning from results of all trials i for i going from *0* to j, to choose the optimal hyperparameter combination for trial j.
This implies that the search algorithm will only take the best combination between n explorations (or n hyperparameters configurations) while doing a total of $n \times m$ trainings.

B.2 PBT scheduler:

B.2.1 Simplified operating explanation:

The population-based scheduler starts with the n exploration chosen by the search algorithm [2]. It then waits that all trials have done the same number of iteration before perturbing all of them.
Concretely, the PBT scheduler uses the $n \times m$ trainings to generate $n \times m$ new configurations (perturbations[3]) as there are m iterations, and n perturbations per iterations. PBT scheduler is doing m times more explorations for the same number of training. nings, that is a speedup of *O(m)* compared to the last paragraph.

B.2.2 Exponential acceleration:

We explained above that PBT scheduler generates $n \times m$ perturbations if trials are independent (every trial receive m perturbations without considering results of other trials). We will now show that PBT Scheduler is much stronger as it has another functionality: the configurations of best trials are copied by worst trials. This implies that results of trials are not independent which allows better results:
For every iteration, hyperparameters of all trials are randomly perturbed. This generates indeed n new configurations for every iteration. But after every iteration, the worst half of trials copy the configurations from best half trials.

We make two assumptions for following calculations:

1. If a trial x is better then a trial y at iteration i, x is better than y for every iterations.
2. Perturbations are local (only exploit). A trial with a configuration that is in a subspace will remain in the same subspace after being perturbed.

In other words, the search over n new configurations comes from the $\frac{n}{2}$ best trials found at last iteration, the n trials from last iteration were themselves issued from the $\frac{n}{2}$ best trials from antepenultimate trial. At every iteration the n trials come from the best half-space of trials in last iterations, themselves coming from the half best space of antepenultimate iteration. At ith iteration, the n trials to be perturbed come from space that has been refined and halved i times, effectively a subspace of the original space that is 2^i times smaller!

We showed that a PBT scheduler using $n \times m$ trainings does exactly the equivalent of a random search algorithm [4] with $n_1 = (\frac{n}{2})^m$ trials (against n with PBT scheduler) if the search algorithm halves its worst half trials at every iteration.

Obviously PBT is not that effective because of the two strong assumptions we made. We can redo the calculations without the second assumption (2.) :
Let us assume that the perturbations of PBT are local (exploitation) with constant probability p_1 and completely random over the whole search space (exploration) with probability $1 - p_1$. We consider that the probability that an exploration goes in the previously evicted space is 1 (in reality it is $1 - \frac{1}{2^i}$).
Also, let us say that a local perturbation (exploitation) has constant probability p_2 to go back to a previously evicted space. We define p the probability that a perturbation go back to a previously evicted space.
$p = p_1 p_2 + (1 - p_1)$
$p = 0 \iff p_1 = 1 \quad and \quad p_2 = 0$
$p = 1 \iff p_1 = 0 \quad or \quad p_2 = 1$
We are only doing $n(1-p)$ perturbations in the best halved space at every iteration against n before, so the total number of perturbations is $(\frac{n(1-p)}{2})^m$. The reason why we don't want to have p too close to 0 is that it permits exploration that is good to mitigate the

[2] Note that this assumption reduces the importance of the search algorithm as he performs his n explorations based on information from trials trained only once.

[3] We here make the assumption that perturbations and explorations from search algorithms are similar.

[4] it does the equivalent of random search because here the perturbations are random, an interesting idea would be to develop a PBT scheduling algorithm with Bayesian Optimisation based perturbations, as we showed that these are far better than random.

first assumption (1.).

Nevertheless, PBT scheduling have important drawbacks:

- It is slightly more complex to implement.
- It uses more memory, as all good trials are stored in memory, to allow bad trials to copy good trials.
- It is much slower for the same number of iterations, as all trials must be saved to memory periodically, and eventually be replaced by another better trial.

If memory is a bottleneck, we suggest the following solution to reduce significantly the use of memory: We will spare memory by not saving the models into memory, hence not doing any copies. We can retrain worst half trials from best half trials at every iterations. For iteration i, $\frac{n}{2}$ trials train once, and the other $\frac{n}{2}$ trials train i times. The total number of trainings is:

$$n + \sum_{i=1}^{m} \frac{n}{2}(1+i) = n + \frac{n}{2}(m + \frac{m(m+1)}{2}) \qquad (9)$$

which results in $O(n \times m^2)$ trainings instead of $O(n \times m)$ trainings.

We showed that PBT scheduler permitted to do $(\frac{n(1-p)}{2})^m$ searches when doing $n \times m$ trainings. It is equivalent to a random search algorithm with $n_1 = (\frac{n(1-p)}{2})^m$ that is using a median scheduler (see below). Median scheduler provide a speed up of $O(m)$. Hence PB scheduler provides a speed up of $\frac{(\frac{n(1-p)}{2})^m}{m \times n} = O(\frac{n^{m-1}}{m})$, and $O(\frac{n^{m-1}}{m^2})$ for a memory free case.

B.3 Median scheduler:

We see how other schedulers perform compared to PBT scheduler. The median scheduler simply stops at every iteration the worst half of the trials. For n trials, the total number of trainings follows $\sum_{i=0}^{m} \frac{n}{2^i} = 2n$. It generates n trials doing only $2 \times n$ trainings instead of $n \times m$. A speed-up of $O(m)$.

B.4 ASHA Scheduler:

The scheduler we often used in this project is the ASHA scheduler. ASHA is constructed with a bracket parameter k. The search starts with n trials. After one iteration, the $\frac{n}{k}$ best trials are kept, and next selection happens in k iterations. Then, every selection gets q trials, and keeps the $\frac{q}{k}$ best trials and next selection is in k times more iterations. The number of steps is $log_k(n)$, the number of iterations for step i is k^i and the number of trials is $\frac{n}{k^i}$. The overall number of trainings is $\sum_{i=0}^{log_k(n)} n, \quad with \quad m = \sum_{i=0}^{log_k(n)} k^i = n$
The total number of trainings is in $O(log_k(n) \times n)$. We have a speedup of $O(\frac{m}{log_k(n)}) = O(\frac{n}{log_k(n)})$. The bandit based choice happens in case there are several brackets, with more or less configurations for the same time budget (allowing less or more training respectively). In our project we used $k = 3$. ASHA provides the same speedup than the following scheduling strategy: This strategy stops a trial if it is less good than the worst of the m best yet obtained trials for the same number of iterations. Let us say that trials are random. Second trial has probability $\frac{1}{2}$ to be better than the first random trial, third trial has probability $\frac{1}{3}$ to be the best, nth has probability $\frac{1}{n}$. The total number of training is $\sum_{i=0}^{n} \frac{m}{i}$ which is in $O(m \times log(n))$.
The speedup is in $O(\frac{n}{log(n)})$ and is worse for any algorithm better than random search as trials will improve progressively as the search algorithm learns.

B.5 Discussion:

We propose here a quick comparison of the three schedulers we presented, using RandomSearch and HyperOptSearch as search algorithms.

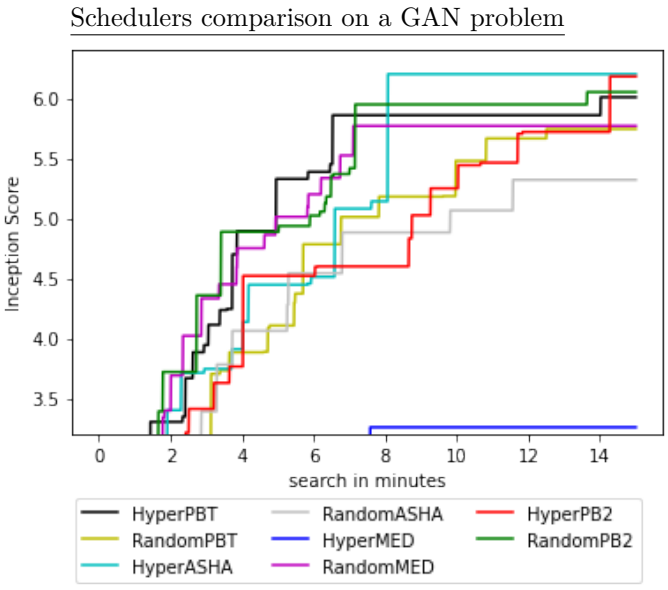


Figure 59: All schedulers were given 2 constraints to find the best configuration: a total training time of 15min, and a maximum of 100 training iterations per trials.

We show on the Figure 59 an experimental comparison. Unfortunately, we are not able to show exponential

superiority of the population-based scheduling. We have 3 explanations:

1. The small maximum number of iterations allowed (100) limits the interest of model copying.
2. The small time that an iteration takes in this problem (a few seconds) limits the interest of model copying, and causes the PB scheduler overhead to be significant.
3. The PBT scheduler is not exactly implemented as we explained above as the local perturbations are not converging to a smaller and smaller subspace.

We also included a very recent scheduler: the PB2 scheduler, with bandit based perturbations instead of random perturbations for the PB scheduler we have been describing. It has promising results, but an even bigger scheduling overhead.
Hyper searches are better than random searches on all schedulers but one: Median stop. As this search is way worst, we think it is due to an implementation issue.
As a conclusion, the PBT scheduler could be exponentially faster than RandomSearch with any scheduler, using our proposed strategy. However current PBT implementation did not demonstrate this speed up because of other implementation choices. further work could consist of conceptualisation and implementation of this exponential optimization strategy
We advise to use PBT scheduler in the following cases:

- Huge search space.
- Huge number of iterations.
- Huge number of trials.
- Scalable comparison of the trials in function of the iterations.
- Memory is not a bottleneck.

C APPENDIX ON VOCABULARY

- Iteration: number of times a trial is trained.
- Training: number of trainings (possibly over different trials).
- Search: the search of the trial with the best hyperparameter configuration using one search algorithm.
- Trial: a model with a hyperparameter configuration.
- Scheduler: the trial manager of a search algorithm: Once the search algorithm generates a trial with a new configuration, the trial scheduler is the one deciding if it should continue training or not based on his results.
- Exploration: the creation of a hyperparameter configuration assigned to a model to create a trial. The trial is then evaluated through training and validation testing.
- Combination: the hyperparameter combination from the search space of a trial.
- Configuration: the hyperparameter combination from the search space of a trial.
- Learning search algorithm: an algorithm that learns from previous trials to decide configuration of next trial.
- Adaptive search algorithm: an algorithm that learns from previous trials to decide configuration of next trial.

REFERENCES

Antognini D., Faltings B., 2020, arXiv preprint arXiv:2002.06854

Antognini D., Musat C., Faltings B., 2019, CoRR, abs/1909.11386

Balandat M., Karrer B., Jiang D. R., Daulton S., Letham B., Wilson A. G., Bakshy E., 2019, arXiv preprint arXiv:1910.06403

Bergstra J., Bengio Y., 2012, The Journal of Machine Learning Research, 13, 281

Bergstra J. S., Bardenet R., Bengio Y., Kégl B., 2011, in Advances in neural information processing systems. pp 2546–2554

Buckley C. S. G. A. J. S. A., 1995, Gaithersberg, pp 69–80

Cho K., Van Merriënboer B., Gulcehre C., Bahdanau D., Bougares F., Schwenk H., Bengio Y., 2014, arXiv preprint arXiv:1406.1078

Dimmery D., Bakshy E., Sekhon J., 2019, in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. pp 2914–2922

Dodge J., Gururangan S., Card D., Schwartz R., Smith N. A., 2019, arXiv preprint arXiv:1909.03004

Falkner S., Klein A., Hutter F., 2018, arXiv preprint arXiv:1807.01774

Gamboa M., 2019, Bayesian optimization from medium

Goodfellow I., Pouget-Abadie J., Mirza M., Xu B., Warde-Farley D., Ozair S., Courville A., Bengio Y., 2014, in Advances in neural information processing systems. pp 2672–2680

Hansen N., 2006, in , Towards a new evolutionary computation. Springer, pp 75–102

Harrison Jr D., Rubinfeld D. L., 1978

Hochreiter S., Schmidhuber J., 1997, Neural computation, 9, 1735

Kahn M., 1994, UCI Diabete Data Set

Kandasamy K., Vysyaraju K. R., Neiswanger W., Paria B., Collins C. R., Schneider J., Poczos B., Xing E. P., 2020, Journal of Machine Learning Research, 21, 1

Kennedy J., Eberhart R., 1995, in Proceedings of ICNN'95-International Conference on Neural Networks. pp 1942–1948

Kingma D. P., Ba J., 2014, arXiv preprint arXiv:1412.6980

Kingma D. P., Welling M., 2014, Auto-Encoding Variational Bayes (arXiv:1312.6114)

Kowsari K., Heidarysafa M., Brown D. E., Meimandi K. J., Barnes L. E., 2018, in Proceedings of the 2nd International Conference on Information System and Data Mining. pp 19–28

Krizhevsky A., 2009, Technical report, Learning multiple layers of features from tiny images

LeCun Y., Boser B., Denker J. S., Henderson D., Howard R. E., Hubbard W., Jackel L. D., 1989, Neural computation, 1, 541

LeCun Y., Cortes C., Burges C., 2010, ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist, 2

Li L., Jamieson K., DeSalvo G., Rostamizadeh A., Talwalkar A., 2017, The Journal of Machine Learning Research, 18, 6765

Li L., Jamieson K., Rostamizadeh A., Gonina E., Hardt M., Recht B., Talwalkar A., 2018, arXiv preprint arXiv:1810.05934

Liu Y.-R., Hu Y.-Q., Qian H., Yu Y., 2019, in Proceedings of the First International Conference on Distributed Artificial Intelligence. pp 1–8

Louppe G., Kumar M., 2016, Bayesian optimization with skopt

Maas A. L., Daly R. E., Pham P. T., Huang D., Ng A. Y., Potts C., 2011, in Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, Portland, Oregon, USA, pp 142–150, http://www.aclweb.org/anthology/P11-1015

Mani S., Sankaran A., Tamilselvam S., Sethi A., 2019, arXiv preprint arXiv:1911.07309

Robbins H., Monro S., 1951, The annals of mathematical statistics, pp 400–407

Storn R., Price K., 1997, Journal of global optimization, 11, 341

Xiao H., Rasul K., Vollgraf R., 2017, arXiv preprint arXiv:1708.07747

Zhuang J., Tang T., Ding Y., Tatikonda S. C., Dvornek N., Papademetris X., Duncan J., 2020, Advances in Neural Information Processing Systems, 33