Oríon : Experiment Version Control for Efficient Hyperparameter Optimization

Christos Tsirigotis[1*†]
chritsir@auth.gr

Xavier Bouthillier[2*]
xavier.bouthillier@umontreal.ca

François Corneau-Tremblay[2†]
francois.corneau-tremblay@umontreal.ca

Peter Henderson[3]
peter.henderson@mail.mcgill.ca

Reyhane Askari[2]
askarihr@iro.umontreal.ca

Samuel Lavoie-Marchildon[2]
samuel.lavoie-marchildon@umontreal.ca

Tristan Deleu[2]
tristan.deleu@umontreal.ca

Dendi Suhubdy[2]
suhubdyd@iro.umontreal.ca

Michael Noukhovitch[2]
mikhailn@iro.umontreal.ca

Frédéric Bastien[2]
bastienf@iro.umontreal.ca

Pascal Lamblin[2]
lamblinp@iro.umontreal.ca

[1] Aristotle University of Thessaloniki;
[2] MILA - Université de Montréal;
[3] MILA - McGill

Abstract

We present Oríon, a new black-box optimization tool currently in development that is designed to adapt to the workflow of machine learning researchers for minimal obstruction. We propose a new version control system for experiments, which can significantly improve the organization of research projects in machine learning as well as the efficiency of hyperparameter optimization. The entire tool is built with the goals of promoting reproducibility, fair benchmarking of different machine learning models, and providing a platform for the research of black-box optimization algorithms.

1 Introduction

Hyperparameter optimization is one of the most time-consuming parts of research in machine learning. Several classes of models, such as deep neural networks take days or weeks to train, making the process of hyperparameter tuning even more time-consuming [12]. In spite of this, the use of

[*] Equal contributions
[†] Work done during internship at MILA - Université de Montréal

Preprint. Work in progress.

automatic hyperparameter optimization tools is not widespread in the community of deep learning researchers [15, 16]. This causes a serious risk of positive bias, since research is often based on incremental improvements to state-of-the-art methods. This also contributes to the problem of reproducibility when models are highly sensitive to hyperparameter values [9].

There are many frameworks available for hyperparameter optimization; spearmint, hyperopt [1], automl, comet.ml [17], sigopt [4], schedy, and optunity [3] to name a few. However, the availability of such frameworks does not translate to widespread adoption. Based on a local survey we conducted, we concluded that the most important reason for the low adoption rate of these frameworks is the cognitive overhead incurred by using them. We thus presume that researchers are significantly more willing to spend time learning about frameworks to implement their models rather than spending time learning tools that impact the other parts of the research workflow, such as hyperparameter optimization, experiment management, visualization tools, etc.

To address this, we propose Oríon[3], a different approach centered on the following idea: machine learning researchers should be viewed as users, not as developers. From this perspective it follows that hyperparameter optimization should be adapted to the workflow of researchers, rather than imposed as an API to adapt their code to.

In order to make such an adaptation to the workflow, we propose a **non-intrusive** way of communicating with the user's script. We designed the tool to be **implementation agnostic**, in order to support user scripts written in any language or framework, and **configuration file agnostic**, in order to support any text-based configuration file format they use. We built it to work **asynchronously** as a way to avoid the need of setting up master and workers, and to improve resiliency. Finally, we made it incrementally configurable for flexibility and simplicity. We will describe all those features in Section 2.

To further improve the research workflow, we propose a new experiment version control system for proper experiment management and boosted hyperparameter optimization. We will present this system in Section 3.

With community development in mind, we designed Oríon to be modular and support external contributions as plug-ins. Supporting contributions is an important part of our tool, as one of our main goals is to support research in the area of black-box optimization. We will describe the related design choices in Section 4.

2 Adapting black-box optimization to the research workflow

Oríon is meant to be simple to configure and operate. The simplicity of configuration is enabled by the configuration-type and implementation agnosticity of the tool we present. We explain this further in Section 2.1. For the simplicity of operation, we rely on asynchronous execution of workers to avoid the common master-workers architecture. We present this briefly in Section 2.2.

2.1 Agnosticity for broad support

```
./userscript -x 5 -some-other 'arguments'
```

Suppose this snippet represents the normal way a user would execute their script to train a model. By using the same configuration interface, it is possible to declare a hyperparameter to be optimized by writing a prior distribution definition instead of a specific value for the hyperparameter.

```
orion hunt -n expname \
     ./userscript -x~'uniform(0,10)' -some-other 'arguments'
```

The above command-line call will create a new **experiment** instance named `expname` with a search **space** composed of a single real **dimension** with uniform prior. A local **worker** will then be instantiated, which will be executing `userscript`, each time providing it with particular hyperparameter values suggested by a black-box optimization algorithm. In this example, the default hyperparameter optimization algorithm would be used, which randomly samples hyperparameters from the specified

[3] https://github.com/mila-udem/orion

distributions. The single complete execution of the user's script, defined by the set of hyperparameter values and the corresponding evaluation results, is referred to as a **trial**. An experiment is thus defined as the ensemble of all trials corresponding to a snapshot of the code-base[4] and a specific search space definition.

There are no restrictions on the user's script code, since Oríon uses the script's own configuration interface to provide hyperparameter values to it. Communicating the results from the user's script to the worker is less trivial, since there is no predefined communication interface. Therefore, two options are provided: a script can either 1) write results to a file, to a path and according to a template provided by Oríon, or 2) use the *client* API provided. In the case when Oríon is not used, a call to the API defaults to a Python `print` statement. Usage of the API does not obstruct a user from calling their script without using Oríon. Hence, the most intrusive part is the code modification needed to report trial results.

A user may have also chosen to use configuration files as a way to provide input to their script. Oríon supports this use case as well, since the markers it uses to specify search spaces can also be used inside any type of text-based configuration files, be it JSON, YAML or another user-defined format. Thus, in total it is **configuration-type agnostic**.

Once an experiment is configured, it can be executed without any arguments passed. Suppose that the experiment defined above is registered in a database. Then one may execute it by simply calling the following command:

```
orion hunt -n expname
```

If any modification to the code of user's script happens between the initialization of the experiment and its execution, then a **branching event** will occur, and a new experiment will be created with the same configuration. This process is part of the Experiment Version Control system, which we will explain in Section 3.

2.2 Asynchronicity for simple execution

A significant overhead for many frameworks is the process of dispatching workers for concurrent black-box optimization. Most frameworks use a master-workers architecture. This implies that master's process must be deployed either by the user or by a service provider. In the latter case, an API is provided as it is needed from a SaaS delivery model. In order to avoid such third-party dependencies, we rather put the responsibility of generating trials inside the workers. We also assume the user's script to be **resumable**, so that interrupted trials can be resumed with minimal loss of computation time.

The synchronization point is the database. Sharing information between workers is not achieved by establishing interprocess communication channels. Instead it happens implicitly by reading the common history from the database. Then, workers make decisions based only on that common history.

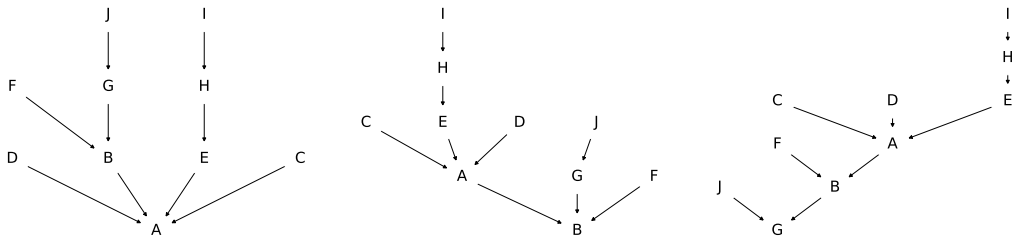
Every operation to the database is implemented in a non-blocking fashion, in order to provide a better throughput. The design choices are further described in Appendix A.

3 Version control and extended optimization

It is common practice in software development to use version control systems to organize the evolution of the code. Research is no different than code development in its iterative nature. Yet, the task of organizing the research results is still far from being standardized like code version control is. There are fortunately new tools, like Datmo [19] and DVC [13], which aim to improve research organization and reproducibility in the form of version control systems.

However, there is also a lack of features compared to code version control. While the latter saves modifications rather than raw content, current tools for data version control only support tagging and snapshots. Saving modifications rather than raw content makes it possible to reapply these modifications on other contexts, with some adaptations if required. We propose to adopt such

[4] Snapshot of the code-base is commit-hash from user's code version control software.



(a) (b) (c)

Figure 1: Visualizations of a project tree with different path of adaptations depending on which experiment is being optimized. Note that all path of adaptations results from the same project tree. In 1a, the experiment being optimized is `A`, the original one. In 1b, the experiment being optimized is `B`, the first children of `A` in 1a. In 1c, the experiment being optimized is `G`, the second children of `B` in 1a.

practices in our Experiment Version Control, so that trials from one experiment can be used by another one. In addition to helping organization of research projects, such a feature makes it possible to warm start the optimization process of a new experiment, as well as improving the search efficiency if many related experiments are being executed concurrently. This reduces the time overhead of iterative hyperparameter optimization while maintaining the reproducibility of results.

3.1 Structure of the Experiment Version Control system

Suppose a user is working on some experiment `A`. After some time, they either edit the code or want to change the configuration of the experiment. Doing so using Oríón would trigger a **branching event** and create a new experiment that we will call `B` for simplicity. Experiment `B` is a children of `A`. They are connected to each other with an adapter based on the modifications applied on `A` to create `B`. The role of the adapter is to transform trials from `A` to make them compatible with `B` and vice-versa. Thanks to this adapter, it is possible to access trials from `A` while optimizing `B` or the opposite.

There is no limit to the number of experiments created by branching. The user could branch again from `A` many times as well as from `B`. See Figure 1 for toy examples of a project tree. Note that the adapters have two directions because adapting trials from `A` to `B` is the inverse process of adapting from `B` to `A`. For example, the direction of the adapter between `A` and `B` is reversed in Figures 1a and 1b.

Trials belong to a single experiment. When they are fetched from many different experiments in the project tree, they are only grouped together in memory at execution time and are never saved into another experiment inside the database. One may note however, that the algorithm trained on the grouped trials is now biased by the other trials of these other experiments. To ensure full traceability, the database indexes of the trials affecting the current state of the search algorithm are logged within each trial at generation time. It is thus possible to filter them.

3.2 Branching events

As we mentioned previously, any change in the code of the user or search space definition will trigger an *experiment branching event*. When such an event occurs, Oríón will build resolution adapters to ensure compatibility between the original experiment and the new ones when applicable. The user can use a provided API to specify these resolutions through the command-line or through its configuration file.

When no information about resolutions is provided by the user, Oríón will start the *interactive conflicts solver*. This shell-like prompt is meant to reduce the overhead for new users. While prompting the user about conflicts to solve, it will provide the corresponding way to use the markers, so that the next time around, the user can automate the process without the help of the *interactive conflicts solver*.

3.3 Adapters

We will describe the different adapters in this section in order to clarify the type of modifications Oríon supports and how trials are handled to ensure compatibility between experiments.

Dimension addition. Adding a new dimension, a new hyperparameter for instance, automatically makes the previous trials invalid for the new search space definition. If the previous experiment had an implicit default value for this dimension then it is possible to adapt the trials of the two experiments to make them compatible.

If the child experiment fetch trials from the parent experiment, all of them will be augmented with the new dimension set to default value. On the other side, if the parent experiment fetches trials from the child, only the trials with the dimension equal to default value will be passed.

Dimension deletion can be handled similarly – it is actually the inverse process.

Dimension renaming. If the user changes the name of the dimension in the code and needs to adapt the configuration.

Dimension prior change. One can narrow or expand distributions based on current results or even completely change the distribution of hyperparameters being searched. The trials from parent experiment will be filtered based on the new prior and vice-versa.

Algorithm or dispatcher change. The algorithm and the dispatcher are the core of the research process. Changing them thus invokes a branching. This is particularly useful if a user wants to compare how two different algorithms or dispatchers compare when warm-starting from a common set of trials. Neither breaks the compatibility of the trials between experiments. They can fetch all trials from one another.

Code Change. Code changes are currently divided in three different types.

`NO EFFECT`. The current change will not affect results.
`UNSURE`. The current change may affect the results.
`BREAK`. The current change will break results and make them incomparable with previous ones.

Code changes are detected via commit hash changes of the user's code version control system for their script. Any change will trigger an experiment branching. The user need to inform Oríon if the changes are considered to have no effects, if they are unsure or if compatibility is broken. In cases where there is no effect, trials may be fetched from the parent and the child experiment freely. If the change is tagged as `unsure`, only the child experiment may fetch the trials from the parent. The opposite, however, is made impossible as a way to protect the parent's history from potentially harmful biases coming from the child's trials. Finally, no trials may be fetched from the parent to the child, and vice-versa, if the code modification is tagged as `break`.

There is arguably an infinite amount of possible code changes which could be handled more gracefully than dividing them in 3 coarse categories. We believe nevertheless Oríon is an improvement to the common practices in research organization and wish to continue expanding the Experiment Version Control system in the hope of contributing to the standardization of experiment management through version control systems.

4 Modularity

Oríon is built to favor community development. Many components are built to be extensible by plug-ins. For example, optimization algorithms can be defined inside a different Python distribution package and be made discoverable by Oríon through the use of Python entry points. This means that a user or a researcher working with a stable or an experimental optimization algorithm can keep their codebase separated from Oríon's codebase, while also being able to use the framework with it.

4.1 Optimization and Dispatcher Algorithms

Optimization algorithms are abstracted by a programming interface which requires the implementation of the following two abstract methods: First, `Algorithm.suggest` which is called to generating new trials estimated to result in better objectives. Second, `Algorithm.observe` for incorporating results from evaluated trials to the optimization process. Implementations of this interface are re-usable and

in the future they will be able to be used for tuning elements of the learning process online during a pending trial, like a dynamic dispatching mechanism. In addition, a developer can provide a property `Algorithm.is_done` to implement a terminating condition, so a specific algorithm can signal that there can be no further improvement to the objective.

Dispatching algorithms such as early stopping mechanisms are implemented in the same fashion. Algorithms of this type, like Hyperband [14] or POP scheduling [18], are considered dynamic in this context because:

1. They interact with a particular trial's evaluation by using information about the progress of the objective in order to decide whether a trial's execution should be suspended, releasing a computation resource if they do.
2. They bias the reservation priority across trials based on their predicted performance, so that some are more likely to be reserved by a worker for evaluation.

These algorithms implement an extra method called `Algorithm.score`, which maps each trial within a set of trials to a number relative to its priority in that group.

Currently only random search is implemented in the core of Oríon. Additional plugins are being implemented, including Bayesian Optimization (based on a Bayesian Regressor) using `scikit-optimize`, Hyperband scheduling and POP scheduling. This set of plugins can quickly be expanded by the community due to the simple and modular nature of Oríon optimization and dispatching algorithm interfaces.

4.2 Databases

To store information about experiments and act as a point of synchronization between workers, we use a database. While any arbitrary database system can be used, at the time of writing only MongoDB[5] is supported. However, the interface of the database and the assumptions have been kept to a minimum in order to facilitate integration of different kinds of databases.

The first assumption is that read and write operations preserve the integrity of the document. A document should not change during a read operation. Reads and writes on many documents are assumed to be non-atomic agglomerations of atomic reads and writes at single document level.

The second assumption is that the backend provides an atomic read-and-write operation at the single document level. No concurrent modification should occur on a given document during the read-and-write operation.

4.3 Analysis and visualizations

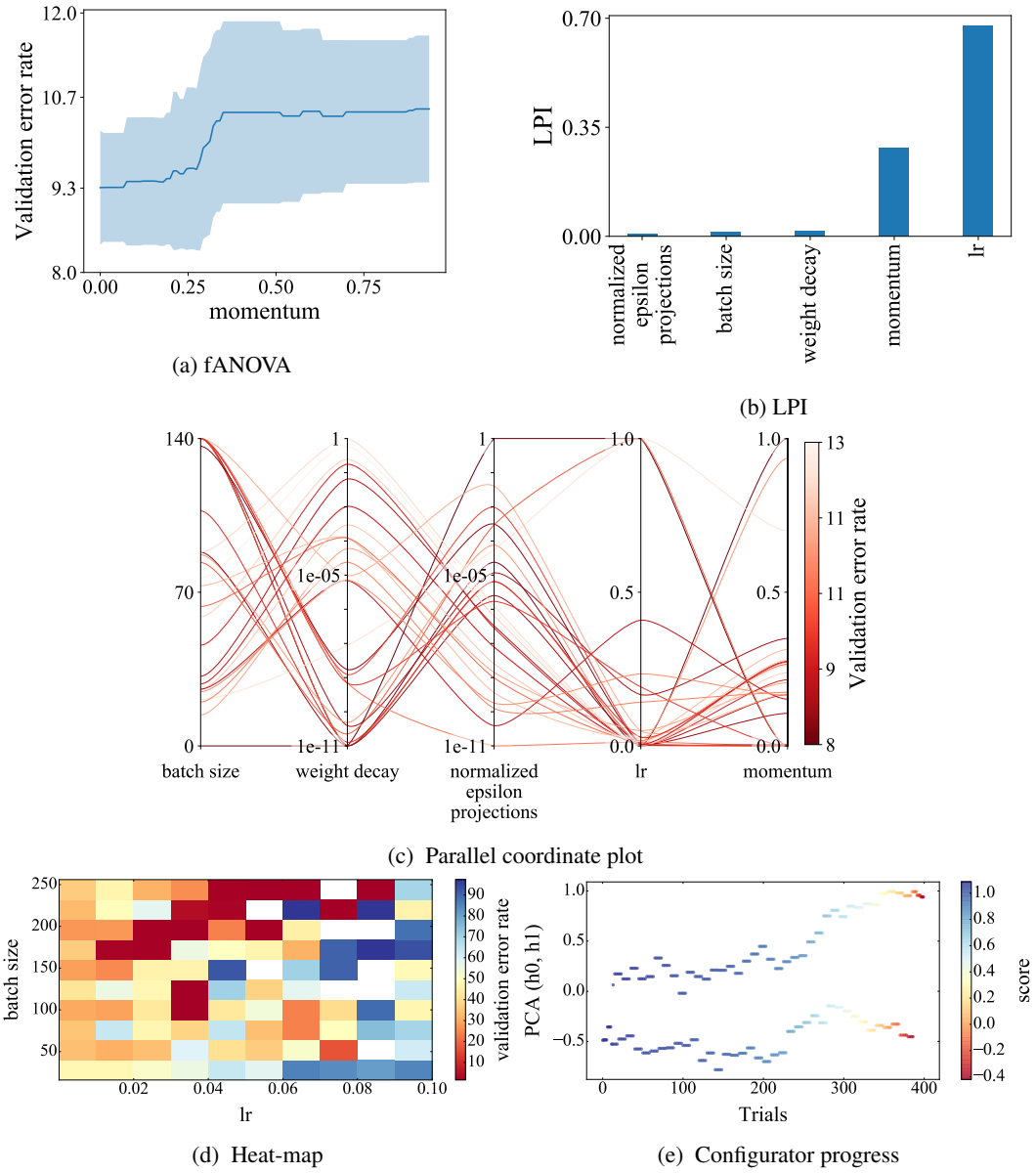
We also provide several analytical tools along with visualizations following the same modular design, allowing for easy expansion to more methods. Currently, we include: fANOVA and LPI analysis and visualization methods of hyperparameter influence as in [2, 7]; hyperparameter performance and search visualization through scatter plots and heatmaps similarly to [11, 14]; parallel coordinate plots as in [2, 5]. Some examples of such visualizations can be seen in Figure 2 with additional information in Appendix B.

Within the visualization toolkit, we also use a modular design with a `DataAnalyzer` which transforms the data (through fANOVA, LPI, PCA, etc.) and a `AnalysisPlotter` which takes the transformed data and generates a visualization. This ensures that future analysis methods and plotting mechanisms can be mixed and matched. The `DataAnalyzer` must conform to information provided in the Oríon format as inputs to ensure a common interface. This modular format allows for easy contribution of new analysis and plotting tools to match any user workflow. Furthermore, it ensures reproducible production of publication figures.

5 Limitations and planned improvements

For now, Oríon is not meant to track an exhaustive amount of information about trials but only what is necessary to hyperparameter optimization. We may extend this to general statistics so that Oríon

[5] https://www.mongodb.com/



(a) fANOVA

(b) LPI

(c) Parallel coordinate plot

(d) Heat-map

(e) Configurator progress

Figure 2: Several visualizations provided with the framework. 2a) fANOVA analysis of hyperparameter marginal performance [8]. 2b) LPI analysis of hyperparameter importance [2]. 2c) Parallel coordinate plot of hyperparameter configurations tested along with performance. 2d) A heatmap demonstrating performance across two hyperparameters. 2e) Performances of the black-box optimizer over the number of experimentation trials in a PCA projection of the hyperparameter space.

can be used as an experiment management tool as well, similarly to Sacred [6]. This would be a natural addition to the Experiment Version Control system. In the same spirit, there is currently no support for collection of large artifacts such as model parameters generated by users' scripts'.

There is no automatic dispatching wrapper for Oríon. The workers must be submitted manually by the user. We plan on supporting auto-resubmission on clusters using Slurm or PBS. We may also integrate a simple dispatcher.

The addition of experiment version control enables many possibilities we have not explored yet. One of them would be to enhance adapters so that they can infer modifications to trials. It could learn for instance correlations between some hyperparameters such that missing hyperparameter values are inferred from one experiment to another. This would be less restrictive than current use of default values.

Finally, we would like to ensure that Oríon is compatible with itself as a user script. This would enable recursive optimization. One could then optimize hyperparameters of optimization algorithms in a single command, providing a useful way to benchmark optimization algorithms against each other and provide empirical data of their sensitivity to hyperparameter selection.

```
orion hunt -config metaconfig.yaml \
     orion hunt -n~trial.name -config=algorithm_params.yaml \
         ./benchmark_script.py -x~‘gaussian(0,30,shape=1000)’
```

Similarly, one could call Oríon to optimize across code changes or any other configuration changes expressed by the Experiment Version Control tree.

```
orion hunt -n ‘optimize_code_change’ \
     orion -n~‘choices([codechange1,codechange2,etc])’
```

6 Conclusion

Due to the Experiment Version Control system we introduced and the features which enable the hierarchy and the modularity of algorithms, experiments and their trials, Oríon comprises a simple but powerful experimentation platform. It is engineered to equip researchers with a lightweight tool to incrementally develop and re-evaluate implementations of ideas. Most importantly, it is a step forward to meet the increasing demand of the machine learning community for conducting reproducible research. This is achieved by Oríon's original goal: to reduce the overhead of hyperparameter optimization for researchers and ensuring the determinism, traceability, and reproducibility of this process. Finally, its intuitive and flexible user interface, seamless and fast integration with any research code, as well as its distributed and asynchronous approach, make Oríon an accessible function for creating precise and organized work. We believe that, by constructing such software tools, we can reinforce and accelerate the application of the scientific method.

Author contributions

Code: C.T., F.C.-T., X.B., D.S.; Writing: X.B., C.T., P.H., F.C.-T., S.L.-M., D.S., P.L.; Visualizations: P.H., R.A., T.D., S.L.-M., X.B., M.N., D.S.; Supervision: X.B., P.L., F.B.; Organization: X.B.

Acknowledgments

We are grateful to Angelos Katharopoulos, Ryan Turner, Philip Paquette, Olexa Bilaniuk, Nasim Rahaman, Mathieu Germain, Massimo Caccia, Breandan Considine, Zhouhan LIN, Dmitriy Serdyuk, Jie Fu, James Bergstra and Frank Hutter for the insightful discussions and comments.

References

[1] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science &*

Discovery, 8(1):014008, 2015.

[2] A Biedenkapp, J Marben, M Lindauer, and F Hutter. {CAVE}: Configuration Assessment, Visualization and Evaluation. *Proceedings of the International Conference on Learning and Intelligent Optimization (LION'18)*, 2018.

[3] Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. Easy Hyperparameter Search Using Optunity. *arXiv:1412.1114 [cs]*, December 2014.

[4] Ian Dewancker, Michael McCourt, Scott Clark, Patrick Hayes, Alexandra Johnson, and George Ke. A strategy for ranking optimization methods using multiple criteria. In *Workshop on Automatic Machine Learning*, pages 11–20, 2016.

[5] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A Service for Black-Box Optimization. *ACM2*, 2017.

[6] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Juergen Schmidhuber. The Sacred Infrastructure for Computational Research. *Proceedings of the 15th Python in Science Conference*, (Scipy):49–56, 2017.

[7] Holge Hoos, Kevin Leyton-Brown, and Frank Hutter. An Efficient Approach for Assessing Hyperparameter Importance. *ICML*, 32(1):754–762, 2014.

[8] Holger Hoos, Kevin Leyton-Brown, and Frank Hutter. An Efficient Approach for Assessing Hyperparameter Importance. *Icml*, 32(1):754–762, 2014.

[9] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

[10] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[11] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. 2017.

[12] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *CoRR*, abs/1605.07079, 2016.

[13] Ruslan Kuprieiev. Data version control: Git extension for data scientists – manage your code and data together, May 2018.

[14] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. 2016.

[15] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are GANs Created Equal? A Large-Scale Study. *arXiv:1711.10337 [cs, stat]*, November 2018.

[16] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. In *International Conference on Learning Representations*, 2018.

[17] Gideon Mendels and Nimrod Lahav. Comet.ml - supercharging machine learning, May 2018.

[18] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. HyperDrive. *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference on - Middleware '17*, pages 1–13, 2017.

[19] Anand Sampat and Patel Shabaz. datmo: The easiest way to manage your models for data science, May 2018.

A Database non-blocking implementation

Every operations to the database are implemented in a non-blocking fashion to provide a better throughput. This is critical at three parts.

New experiment registration When two workers are executed simultaneously on the same new experiment, only one of them will be able to register it thanks to atomic index uniqueness based on experiment's name and user name. The other worker will rollback and instantiate a new experiment object fetching the database.

Trial reservation When a worker tries to reserve a trial, it changes its state to `running`. Atomic read-and-write ensures the trial did not change state between selection and update. On race conditions, one worker will always successfully reserve a trial. The other worker will rollback and try to select another trial.

Trial generation The optimization algorithm's generated trials may only be registered one at a time. The `id` of the trial will be inferred based on the parameter values and the experiment's `id`. The registration of the trial will fail if another trial has been registered with the same `id` concurrently. If registration fails, the algorithm will be trained based on up-to-date history and generate another trial.

B Analysis and Visualization

To analyze the performance of both the models being optimized and the black-box hyperparameter optimization process itself, we provide a set of visualization plugins which can be used. We roughly categorize these tools into 3 types: performance analysis, optimization algorithm analysis and parameter analysis.

In the performance analysis tools, the goal is to determine the performance of the different models within experiments during the hyperparameter optimization process, i.e., the learning curves of several sets of hyperparameters. The optimization algorithm analysis tools are used for determining the performance and behaviour of the black-box hyperparameter optimization method and hyperparameter set selection method. Finally, the parameter analysis tools are used for determining the effects and importance of individual hyperparameters on the performance of the system , i.e., which hyperparameter bears the most weight on the evaluation metric.

B.1 Performance Analysis

Individual performance analysis of trials can be compared using some of the visualization tools. As seen in Figure 3, this bears some resemblance to Figure 1b from [14] and can easily be adapted to map performance against arbitrary resources[6].

B.2 Optimization Algorithm Analysis

To compare black-box hyperparameter optimizers and their search behaviour over time, it may be desirable to plot both the improvement of the found solutions over time and the space of features being searched. To this end, we include several visualization methods which accomplish this in different ways. Figure 4 and 5 demonstrate these visualizations on several tasks. As can be seen, for methods which use dynamic adaptation of the hyperparameters or gradient based updates, these can be particularly useful to both see the improvement of the solutions over time and the space of hyperparameters being searched. This is similar to plots which can be found in [2, 10].

B.3 Parameter Analysis

Determining the effect or importance of different hyperparameters can be highly beneficial when presenting a new algorithm. For example, if a new optimizer performs well across a wide range of learning rates it is a valuable to have a measurement of this desirable property. Similarly, if a hyperparameter bears no effect it may be beneficial to leave it from the hyperparameter optimizer's search space. To aide in this, we provide several visualization tools.

[6] In our case we only consider the number of algorithm steps as a resource

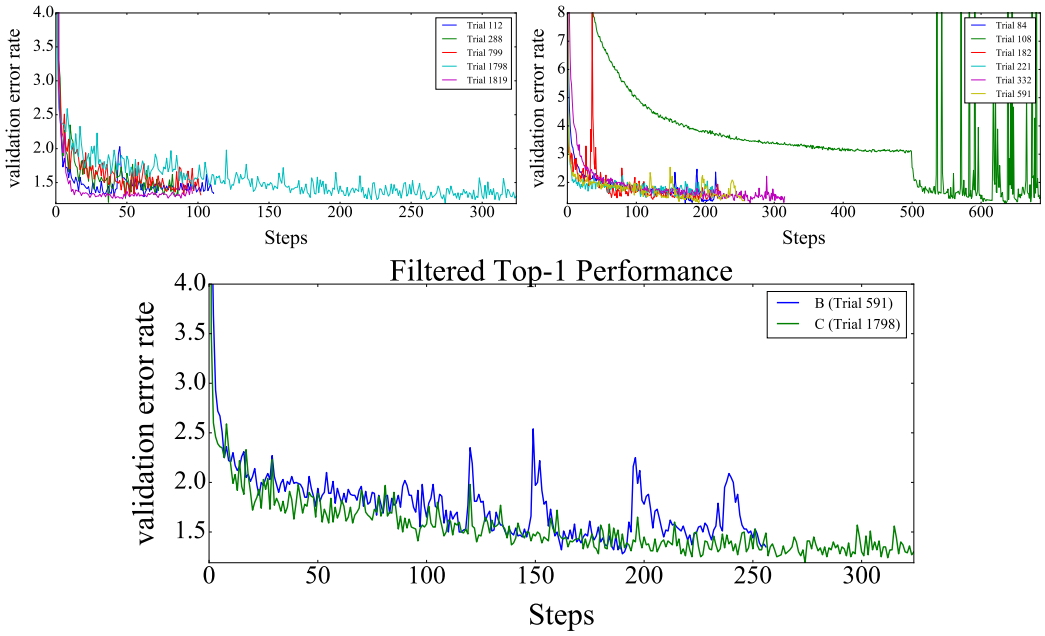


Figure 3: Comparison of the top experiment trials from two Bayesian Optimization runs for finding the best hyperparameters of a multilayer perceptron neural network on the MNIST dataset (top). The best experiment from either hyperparameter optimization run (bottom). Top 0.25% of trials (top left) where filtered. Top 1% of trials (top right) where filtered by final validation performance at the last timestep.

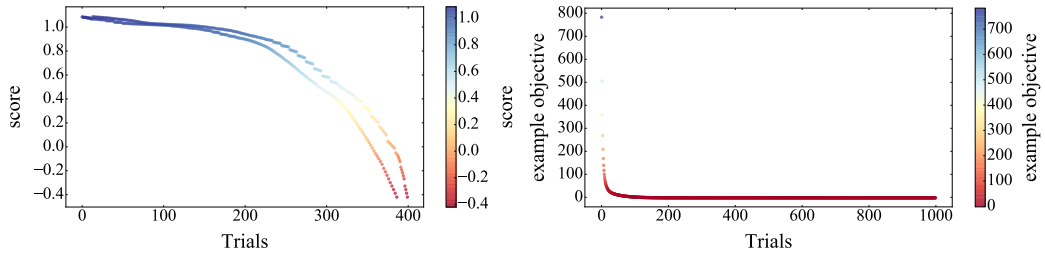


Figure 4: The black-box optimization method performance of the explored configurations over time for population-based training (left) and a gradient-based method (right) on a toy task.

B.3.1 Scatter Plots

While it may be useful to see the progression of hyperparameters over time using scatter plots, it may also be useful to see the evaluation performance in various regions of the hyperparameter search space using projections and comparing different hyperparameters. To this end, Figure 6 demonstrates such provided visualizations.

B.3.2 Heatmaps

Similarly, heatmaps may provide a better picture of the generalized (averaged) performance within various quadrants of the relative hyperparameter space. Figure 7 demonstrates how hyperparameters can be compared via a heatmap either in a projected PCA space or simply comparing sets of hyperparameters. This is similar to Figure 1b in [14].

B.3.3 Parallel coordinate plot

We also include a method for generating parallel coordinate plots. These provide another method for visualizing parameter importance when mapping to performance. See Figure 8 for some examples.

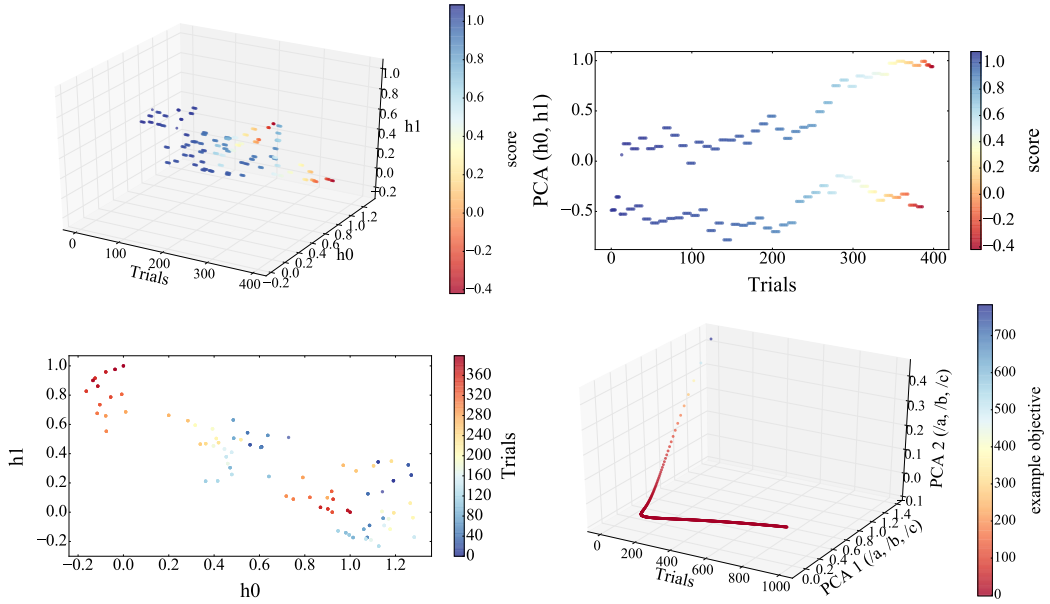


Figure 5: For a population-based training method on a toy task, the pattern of exploration of the state space with varying projections. With the number of experiments done on one axis and the hyperparameters on the other two with score being indicated by colour (top left). A PCA projection of the hyperparameters with the x-axis indicating number of trials and colour as score (top right). Two hyperparameters composing the axes and colour indicating the number of trials completed when the hyperparameter configuration was tried (bottom left). For a gradient based method, PCA projection components composing two of the 3D axes with the third being the number of configurations tried so far and colour indicating evaluation metric (bottom right).

B.3.4 fANOVA

fANOVA analysis [8] is a useful method for determining the importance of certain hyperparameters. Using data gathered from a variety of hyperparameter configurations, an empirical performance model can be fit to predict evaluation performance from configuration settings. Functional ANOVA can then be used to determine how significant the hyperparameters are in defining the algorithm's performance. We provide visualizations and processing methods for this as seen in Figure 9. Furthermore, we can plot the marginal objective using a similar analysis as seen in Figures 10 and 11.

B.3.5 LPI

Local Parameter Importance (LPI) as seen in [2] provides an easy to interpret measure of how important certain hyperparameters are when tuning the performance of an algorithm. As such, we provide a method for generating visualizations based on LPI interpretations as seen in Figure 12.

LPI computes the importances of a parameter p with respect to a metric of interest c by calculating the variance induced on c while modifying p and keeping constant the others parameters θ. The LPI scores is computed as a categorical distribution over all the parameters. Hence, the parameter importance is relative to the other parameters.

To avoid computing c for every combination of hyperparameters, a regressor (e.g. random forest) can be trained to approximate the score induced by modifying a parameter.

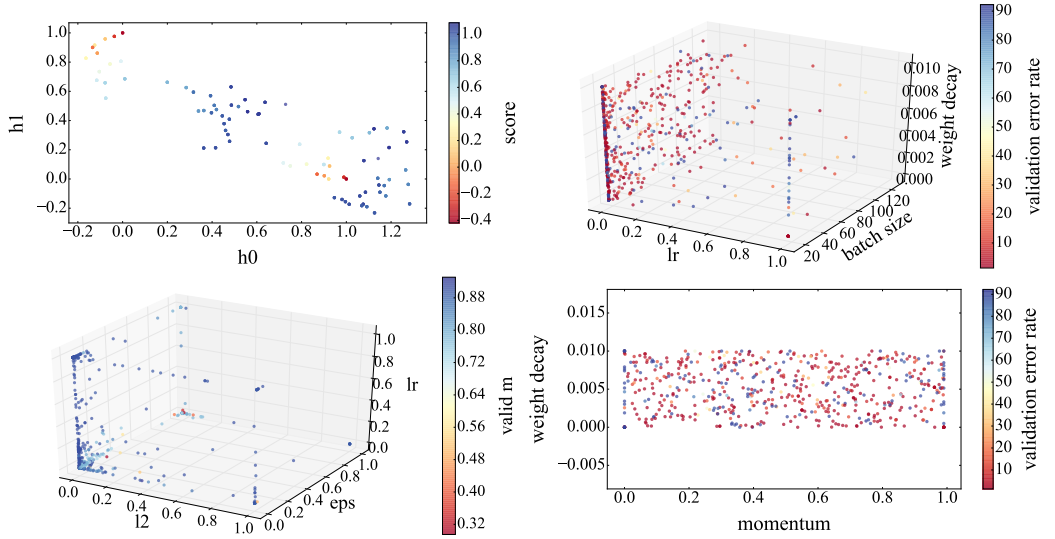


Figure 6: Scatter plots comparing performance across the hyperparameter space. A toy task for population based training with only two hyperparameters (top left). Bayesian Optimization of an MLP on MNIST comparing learning rate, batch size, and weight decay (top right). A grid search comparing L2 regularization, an epsilon factor and learning rate for a ResNet training on CIFAR (bottom left). Comparing momentum and weight decay yields a plot with scattered performance indicating unlikely importance of the hyperparameters on Bayesian Optimization of a MLP on the MNIST dataset.

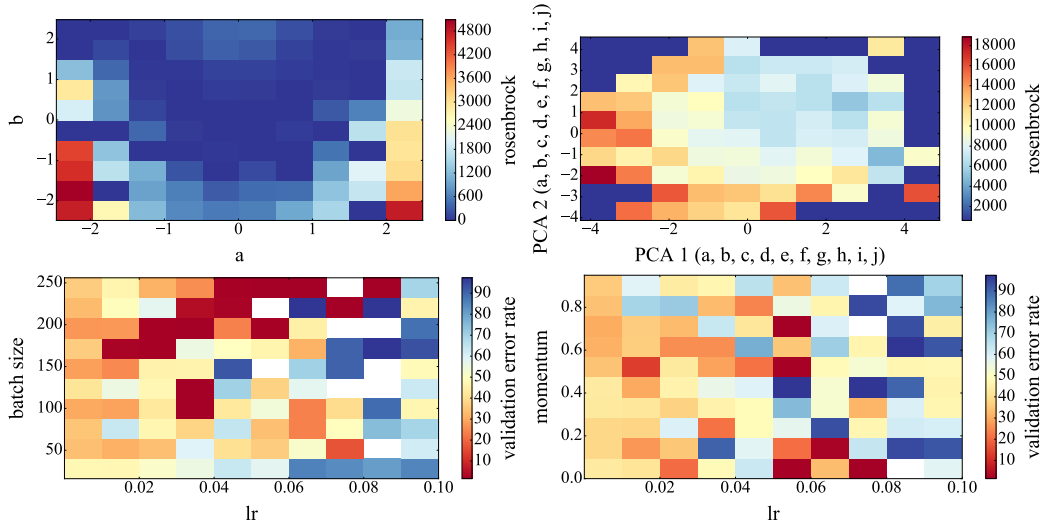
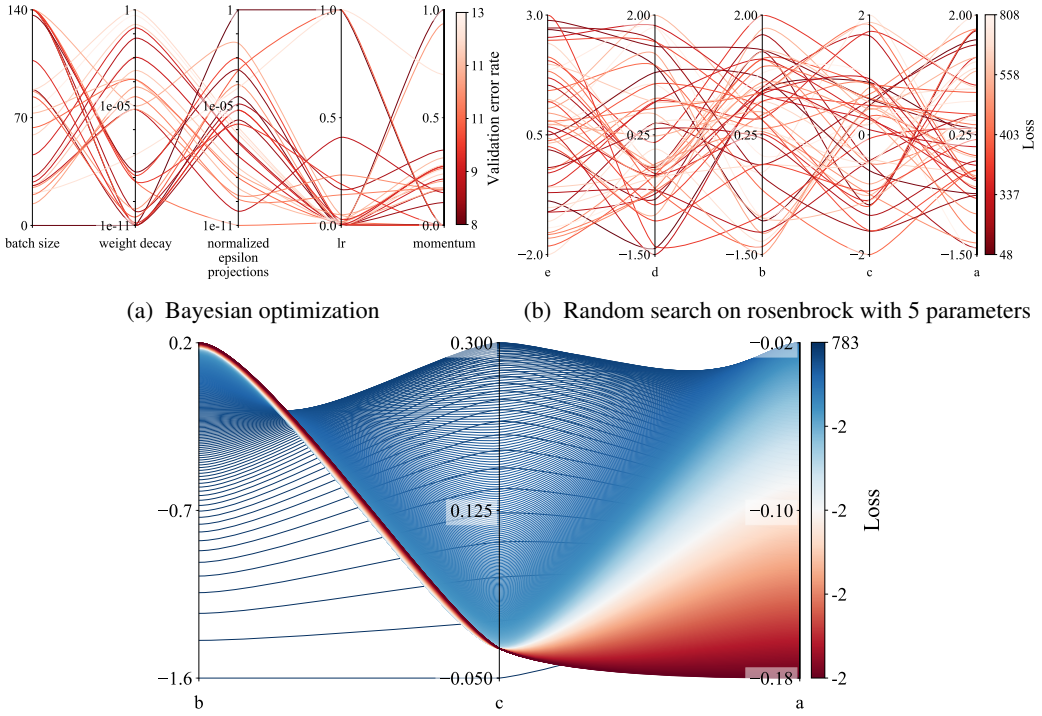


Figure 7: Heatmaps of parameter performance across projections of the rosenbrock task with 2 hyperparameters and 10 hyperparameters, respectively (top) and across learning performance on an MNIST task (bottom).

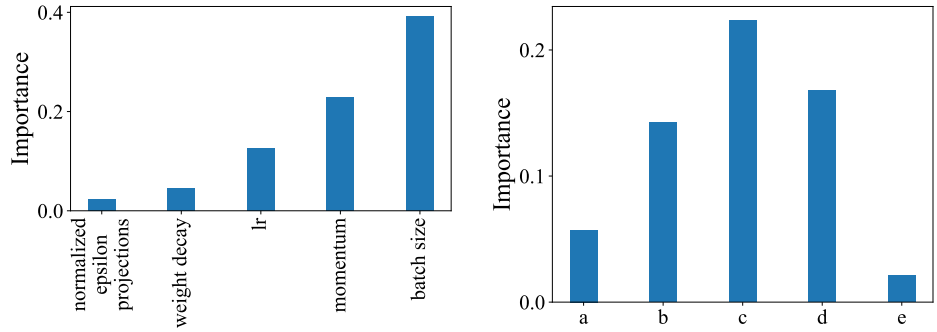


(a) Bayesian optimization

(b) Random search on rosenbrock with 5 parameters

(c) Gradient descent on rosenbrock with 3 parameters

Figure 8: Here we demonstrate several instances of parallel coordinate plots using different optimization methods on various tasks. The difference in optimization methods becomes apparent as random search bears no real pattern to the search space while gradient descent and Bayesian Optimization converge to some portion of the search space.



(a) Validation error rate of a ResNet trained on CIFAR10

(b) Rosenbrock function with 5 parameters

Figure 9: Bar plot representing the fANOVA importance score obtained for each parameter on two different tasks. In 9a, we present the fANOVA score when analysing the top 0.05% of trials with respect to validation error rate for a bayesian optimization on ResNet with CIFAR10. Error-rates ranges between 8% and 13% for the top 0.05%. In 9b, we present the fANOVA importance score when using 100% of the trials from random search on Rosenbrock function.

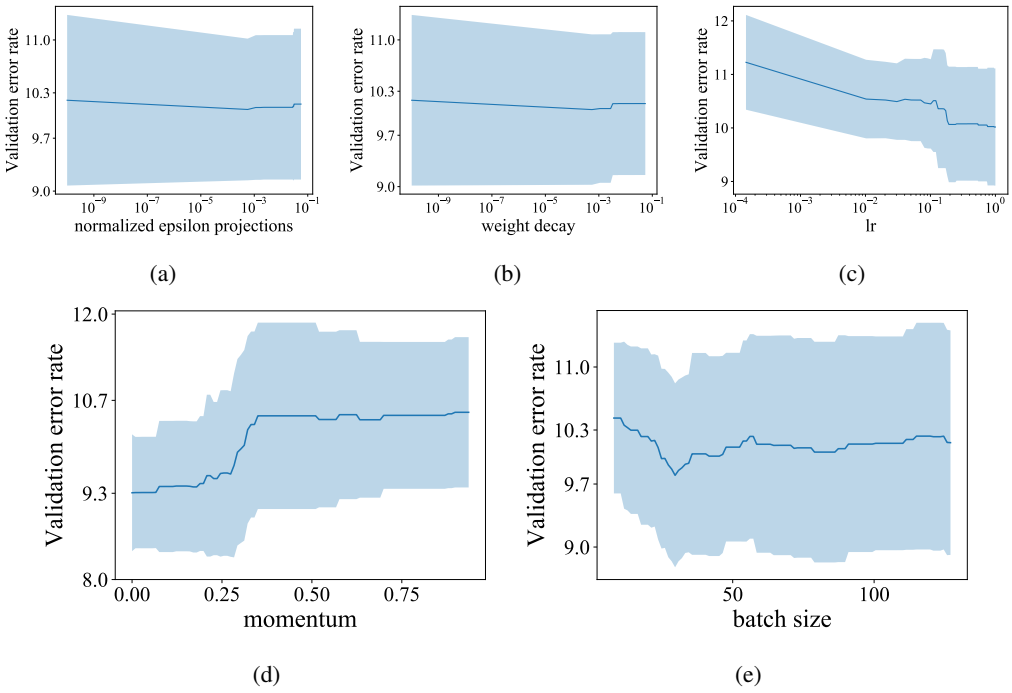


Figure 10: Marginal validation error rate over the space of realizable hyperparameter values. The solid line represents the mean validation error rate while the light-blue area represents the width of one standard deviation. The results are from a bayesian optimization for ResNets trained on CIFAR10. Analysis is done on the top 0.05% of the trials, with error-rates between 8% and 13%.

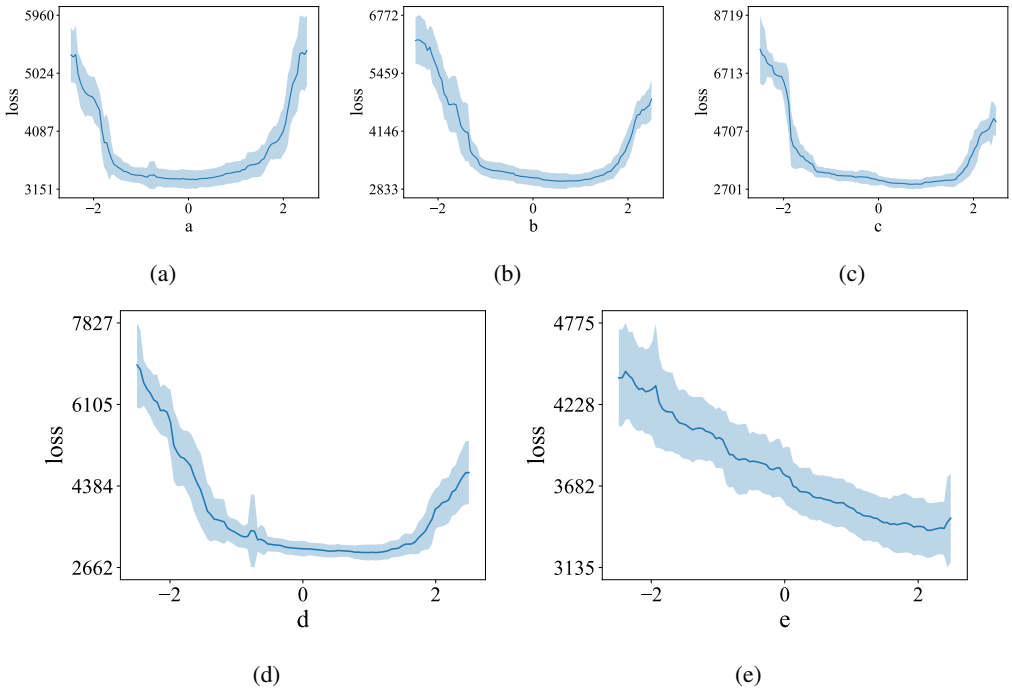
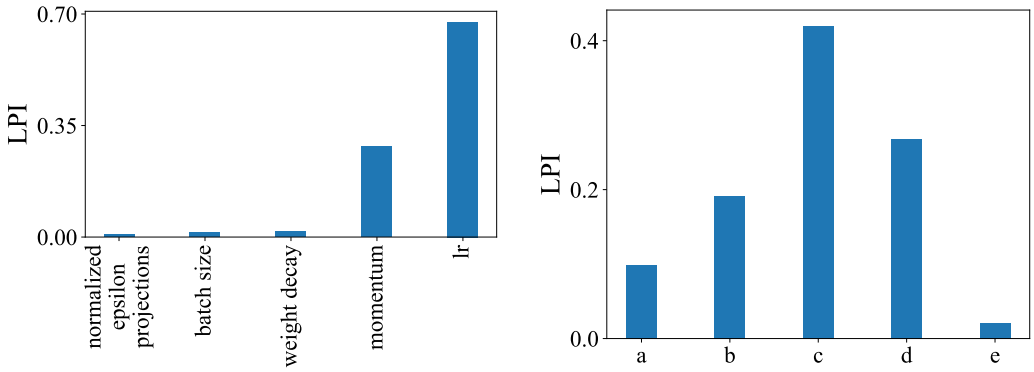


Figure 11: Marginal loss over the space of realizable hyperparameter values. The solid line represents the mean validation error rate while the light-blue area represents the width of one standard deviation. The results are from a random search on the Rosenbrock function. Analysis done on all the trials.



(a) Validation error rate of a ResNet trained on CIFAR10 (b) Rosenbrock function with 5 parameters

Figure 12: Barplot representing the LPI importance score obtained for each parameters on two different tasks. On 12a, we present the importance of the parameters when analysing the top 0.05% of trials with respect to validation error rate for a bayesian optimization on ResNet with CIFAR10. Error-rates range between 8% and 13% for the top 0.05%. In 12b, we present the parameter importance score when using 100% of the trials from random search on Rosenbrock function.