# Optuna: A Next-generation Hyperparameter Optimization Framework

Preprint, compiled July 26, 2019

**Takuya Akiba**[1*], **Shotaro Sano**[1], **Toshihiko Yanase**[1], **Takeru Ohta**[1], **and Masanori Koyama**[1]

[1]Preferred Networks, Inc.

**Abstract**

The purpose of this study is to introduce new design-criteria for next-generation hyperparameter optimization software. The criteria we propose include (1) *define-by-run API* that allows users to construct the *parameter search space* dynamically, (2) efficient implementation of both searching and pruning strategies, and (3) easy-to-setup, versatile architecture that can be deployed for various purposes, ranging from scalable distributed computing to light-weight experiment conducted via interactive interface. In order to prove our point, we will introduce *Optuna*, an optimization software which is a culmination of our effort in the development of a next generation optimization software. As an optimization software designed with *define-by-run* principle, *Optuna* is particularly the first of its kind. We will present the design-techniques that became necessary in the development of the software that meets the above criteria, and demonstrate the power of our new design through experimental results and real world applications. Our software is available under the MIT license (https://github.com/pfnet/optuna/).





## 1 Introduction

Hyperparameter search is one of the most cumbersome tasks in machine learning projects. The complexity of deep learning method is growing with its popularity, and the framework of efficient automatic hyperparameter tuning is in higher demand than ever. Hyperparameter optimization softwares such as *Hyperopt* [1], *Spearmint* [2], *SMAC* [3], *Autotune* [4], and *Vizier* [5] were all developed in order to meet this need.

The choice of the parameter-sampling algorithms varies across frameworks. *Spearmint* [2] and *GPyOpt* use Gaussian Processes, and *Hyperopt* [1] employs *tree-structured Parzen estimator* (TPE) [6]. Hutter et al. proposed *SMAC* [3] that uses random forests. Recent frameworks such as *Google Vizier* [5], *Katib* and *Tune* [7] also support *pruning* algorithms, which monitor the intermediate result of each trial and kills the unpromising trials prematurely in order to speed up the exploration. There is an active research field for the pruning algorithm in hyperparameter optimization. Domhan et al. proposed a method that uses parametric models to predict the learning curve [8]. Klein et al. constructed Bayesian neural networks to predict the expected learning curve [9]. Li et al. employed a bandit-based algorithm and proposed Hyperband [10].

A still another way to accelerate the optimization process is to use distributed computing, which enables parallel processing of multiple trials. *Katib* is built on *Kubeflow*, which is a computing platform for machine learning services that is based on *Kubernetes*. *Tune* also supports parallel optimization, and uses the *Ray* distributed computing platform [11].

However, there are several serious problems that are being overlooked in many of these existing optimization frameworks. Firstly, all previous hyperparameter optimization frameworks to date require the user to statically construct the parameter-search-space for each model, and the search space can be extremely hard to describe in these frameworks for large-scale experiments that involve massive number of candidate models of different types with large parameter spaces and many conditional variables. When the parameter space is not appropriately described by the user, application of advanced optimization method can be in vain. Secondly, many existing frameworks do not feature efficient pruning strategy, when in fact both *parameter searching strategy* and *performance estimation strategy* are important for high-performance optimization under limited resource availability [12][5][7]. Finally, in order to accommodate with a variety of models in a variety of situations, the architecture shall be able to handle both small and large scale experiments with minimum setup requirements. If possible, architecture shall be installable with a single command as well, and it shall be designed as an open source software so that it can continuously incorporate newest species of optimization methods by interacting with open source community.

In order to address these concerns, we propose to introduce the following new design criteria for next-generation optimization framework:

- *Define-by-run* programming that allows the user to dynamically construct the search space,
- Efficient sampling algorithm and pruning algorithm that allows some user-customization,
- Easy-to-setup, versatile architecture that can be deployed for tasks of various types, ranging from light-weight experiments conducted via interactive interfaces to heavy-weight distributed computations.

In this study, we will demonstrate the significance of these criteria through *Optuna*, an open-source optimization software which is a culmination of our effort in making our definition of next-generation optimization framework come to reality.

We will also present new design techniques and new optimization algorithms that we had to develop in order to meet our proposed criteria. Thanks to these new design techniques, our implementation outperforms many major black-box optimization frameworks while being easy to use and easy to setup in various environments. In what follows, we will elaborate each of our proposed criteria together with our technical solutions, and present experimental results in both real world applications and benchmark datasets.

```python
import optuna
import ...

def objective(trial):
    n_layers = trial.suggest_int('n_layers', 1, 4)

    layers = []
    for i in range(n_layers):
        layers.append(trial.suggest_int('n_units_l{}'.
            format(i), 1, 128))

    clf = MLPClassifier(tuple(layers))

    mnist = fetch_mldata('MNIST original')
    x_train, x_test, y_train, y_test = train_test_split(
        mnist.data, mnist.target)

    clf.fit(x_train, y_train)

    return 1.0 - clf.score(x_test, y_test)

study = optuna.create_study()
study.optimize(objective, n_trials=100)
```

Figure 1: An example code of *Optuna*'s *define-by-run* style API. This code builds a space of hyperparameters for a classifier of the MNIST dataset and optimizes the number of layers and the number of hidden units at each layer.

```python
import hyperopt
import ...

space = {
    'n_units_l1': hp.randint('n_units_l1', 128),
    'l2': hp.choice('l2', [{
        'has_l2': True,
        'n_units_l2': hp.randint('n_units_l2', 128),
        'l3': hp.choice('l3', [{
            'has_l3': True,
            'n_units_l3': hp.randint('n_units_l3', 128),
            'l4': hp.choice('l4', [{
                'has_l4': True,
                'n_units_l4': hp.randint('n_units_l4',
                    128),
            }, {'has_l4': False}]),
        }, {'has_l3': False}]),
    }, {'has_l2': False}]),
}

def objective(space):
    layers = [space['n_units_l1'] + 1]
    for i in range(2, 5):
        space = space['l{}'.format(i)]
        if not space['has_l{}'.format(i)]:
            break
        layers.append(space['n_units_l{}'.format(i)] +
            1)

    clf = MLPClassifier(tuple(layers))

    mnist = fetch_mldata('MNIST original')
    x_train, x_test, y_train, y_test = train_test_split(
        mnist.data, mnist.target)

    clf.fit(x_train, y_train)

    return 1.0 - clf.score(x_test, y_test)

hyperopt.fmin(fn=objective, space=space, max_evals=100,
    algo=hyperopt.tpe.suggest)
```

Figure 2: An example code of *Hyperopt* [1] that has the exactly same functionality as the code in 1. *Hyperopt* is an example of *define-and-run* style API.

*Optuna* is released under the MIT license (https://github.com/pfnet/optuna/), and is in production use at Preferred Networks for more than one year.

## 2 DEFINE-BY-RUN API

In this section we describe the significance of the *define-by-run* principle. As we will elaborate later, we are borrowing the term *define-by-run* from a trending philosophy in deep learning frameworks that allows the user to dynamically program deep networks. Following the original definition, we use the term *define-by-run* in the context of optimization framework to refer to a design that allows the user to dynamically construct the search space. In *define-by-run API*, the user does not have to bear the full burden of explicitly defining everything in advance about the optimization strategy.

The power of *define-by-run* API is more easily understood with actual code. *Optuna* formulates the hyperparameter optimization as a process of minimizing/maximizing an *objective function* that takes a set of hyperparameters as an input and returns its (validation) score. Figure 1 is an example of an objective function written in *Optuna*. This function dynamically constructs the search space of neural network architecture (the number of layers and the number of hidden units) without relying on externally defined static variables. *Optuna* refers to each process of optimization as a *study*, and to each evaluation of objective function as a *trial*. In the code of Figure 1, *Optuna* defines an objective function (Lines 4–18), and invokes the '`optimize API`' that takes the objective function as an input (Line 21). Instead of hyperparameter values, an objective function in *Optuna* receives a *living trial object*, which is associated with a single *trial*.

*Optuna* gradually builds the objective function through the interaction with the *trial* object. The search spaces are constructed dynamically by the methods of the *trial* object during the runtime of the objective function. The user is asked to invoke '`suggest API`' inside the objective function in order to dynamically generate the hyperparameters for each *trial* (Lines 5 and 9). Upon the invocation of '`suggest API`', a hyperparameter is statistically sampled based on the history of previously evaluated *trials*. At Line 5, '`suggest_int`' method suggests a value for '`n_layers`', the integer hyperparameter that determines the number of layers in the Multilayer Perceptron. Using loops and conditional statements written in usual *Python* syntax, the user can easily represent a wide variety of parameter spaces. With this functionality, the user of *Optuna* can even express heterogeneous parameter space with an intuitive and simple code (Figure 3).

Meanwhile, Figure 2 is an example code of *Hyperopt* that has the exactly same functionality as the *Optuna* code in Figure 1. Note that the same function written in *Hyperopt* (Figure 2) is significantly longer, more convoluted, and harder to interpret. It is not even obvious at first glance that the code in Figure 2 is in fact equivalent to the code in Figure 1! In order to write the same *for-loop* in Figure 1 using *Hyperopt*, the user must prepare the list of all the parameters in the parameter-space prior to the exploration (see line 4-18 in Figure 2). This requirement will lead the user to even darker nightmares when the optimization problem is more complicated.

### 2.1 Modular Programming

A keen reader might have noticed in Figure 3 that the optimization code written in *Optuna* is highly modular, thanks to its *define-by-run* design. Compatibility with modular programming

Table 1: Software frameworks for deep learning and hyperparameter optimization, sorted by their API styles: *define-and-run* and *define-by-run*.

| | **Deep Learning Frameworks** | **Hyperparameter Optimization Frameworks** |
|---|---|---|
| **Define-and-Run Style** (symbolic, static) | Torch (2002), Theano (2007), Caffe (2013), TensorFlow (2015), MXNet (2015), Keras (2015) | SMAC (2011), Spearmint (2012), Hyperopt (2015), GPyOpt (2016), Vizier (2017), Katib (2018), Tune (2018), Autotune (2018) |
| **Define-by-Run Style** (imperative, dynamic) | Chainer (2015), DyNet (2016), PyTorch (2016), TensorFlow Eager (2017), Gluon (2017) | ***Optuna (2019; this work)*** |

Table 2: Comparison of previous hyperparameter optimization frameworks and *Optuna*. There is a checkmark for *lightweight* if the setup for the framework is easy and it can be easily used for lightweight purposes.

| **Framework** | **API Style** | **Pruning** | **Lightweight** | **Distributed** | **Dashboard** | **OSS** |
|---|---|---|---|---|---|---|
| SMAC [3] | define-and-run | ✗ | ✓ | ✗ | ✗ | ✓ |
| GPyOpt | define-and-run | ✗ | ✓ | ✗ | ✗ | ✓ |
| Spearmint [2] | define-and-run | ✗ | ✓ | ✓ | ✗ | ✓ |
| Hyperopt [1] | define-and-run | ✗ | ✓ | ✓ | ✗ | ✓ |
| Autotune [4] | define-and-run | ✓ | ✗ | ✓ | ✓ | ✗ |
| Vizier [5] | define-and-run | ✓ | ✗ | ✓ | ✓ | ✗ |
| Katib | define-and-run | ✓ | ✗ | ✓ | ✓ | ✓ |
| Tune [7] | define-and-run | ✓ | ✗ | ✓ | ✓ | ✓ |
| Optuna (this work) | define-by-run | ✓ | ✓ | ✓ | ✓ | ✓ |

```
1  import sklearn
2  import ...
3
4  def create_rf(trial):
5      rf_max_depth = trial.suggest_int('rf_max_depth', 2,
           32)
6      return RandomForestClassifier(max_depth=
           rf_max_depth)
7
8  def create_mlp(trial):
9      n_layers = trial.suggest_int('n_layers', 1, 4)
10
11     layers = []
12     for i in range(n_layers):
13         layers.append(trial.suggest_int('n_units_l{}'.
               format(i), 1, 128))
14
15     return MLPClassifier(tuple(layers))
16
17 def objective(trial):
18     classifier_name = trial.suggest_categorical('
           classifier', ['rf' , 'mlp'])
19
20     if classifier_name == 'rf':
21         classifier_obj = create_rf(trial)
22     else:
23         classifier_obj = create_mlp(trial)
24
25     ...
```

Figure 3: An example code of *Optuna* for the construction of a heterogeneous parameter-space. This code simultaneously explores the parameter spaces of both random forest and MLP.

```
1  import chainer
2  import ...
3
4  def create_model(trial):
5      n_layers = trial.suggest_int('n_layers', 1, 3)
6      layers = []
7      for i in range(n_layers):
8          n_units = trial.suggest_int('n_units_l{}'.format
               (i), 4, 128)
9          layers.append(L.Linear(None, n_units))
10         layers.append(F.relu)
11     layers.append(L.Linear(None, 10))
12     return chainer.Sequential(*layers)
13
14 def create_optimizer(trial, model):
15     lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
16     optimizer = chainer.optimizers.MomentumSGD(lr=lr)
17     weight_decay = trial.suggest_loguniform('
           weight_decay', 1e-10, 1e-3)
18     optimizer.setup(model)
19     optimizer.add_hook(chainer.optimizer.WeightDecay(
           weight_decay))
20     return optimizer
21
22 def objective(trial):
23     model = create_model(trial)
24     optimizer = create_optimizer(trial, model)
25     ...
26
27 study = optuna.create_study()
28 study.optimize(objective, n_trials=100)
```

Figure 4: Another example of *Optuna*'s objective function. This code simultaneously optimizes neural network architecture (the `create_model` method) and the hyperparameters for stochastic gradient descent (the `create_optimizer` method).

is another important strength of the *define-by-run* design. Figure 4 is another example code written in *Optuna* for a more complex scenario. This code is capable of simultaneously optimizing both the topology of a multilayer perceptron (method '`create_model`') and the hyperparameters of stochastic gradient descent (method '`create_optimizer`'). The method '`create_model`' generates '`n_layers`' in Line 5 and uses a *for loop* to construct a neural network of depth equal to '`n_layers`'. The method also generates '`n_units_i`' at each *i*-th loop, a hyperparameter that determines the number of the units in the *i*-th layer. The method '`create_optimizer`', on the other hand, makes suggestions for both learning rate and weight-decay parameter. Again, a complex space of hyperparameters is simply expressed in *Optuna.* Most notably, in this example, the methods '`create_model`' and '`create_optimizer`' are independent of one another, so that we can make changes to each one of them separately. Thus, the user can easily augment this code with other conditional variables and methods for other set of parameters, and make a choice from more diverse pool of models.

### 2.2 *Deployment*

Indeed, the benefit of our *define-by-run* API means nothing if we cannot easily deploy the model with the best set of hyperpa-

rameters found by the algorithm. The above example (Figure 4) might make it seem as if the user has to write a different version of the objective function that does not invoke '`trial.suggest`' in order to deploy the *best* configuration. Luckily, this is not a concern. For deployment purpose, *Optuna* features a separate class called '`FixedTrial`' that can be passed to objective functions. The '`FixedTrial`' object has practically the same set of functionalities as the *trial* class, except that it will only suggest the user defined set of the hyperparameters when passed to the objective functions. Once a parameter-set of interest is found (e.g., the best ones), the user simply has to construct a '`FixedTrial`' object with the parameter set.

### 2.3 Historical Remarks

Historically, the term *define-by-run* was coined by the developers of deep learning frameworks. In the beginning, most deep learning frameworks like *Theano* and *Torch* used to be *declarative*, and constructed the networks in their *domain specific languages* (DSL). These frameworks are called *define-and-run* frameworks because they do not allow the user to alter the manipulation of intermediate variables once the network is defined. In *define-and-run* frameworks, computation is conducted in two phases: (1) construction phase and (2) evaluation phase. In a way, contemporary optimization methods like *Hyperopt* are built on the philosophy similar to *define-and-run*, because there are two phases in their optimization: (1) construction of the search space and (3) exploration in the search space.

Because of their difficulty of programming, the *define-and-run* style deep learning frameworks are quickly being replaced by *define-by-run* style deep learning frameworks like *Chainer* [13], *DyNet* [14], *PyTorch* [15], eager-mode *TensorFlow* [16], and *Gluon*. In the *define-by-run* style DL framework, there are no two separate phases for the construction of the network and the computation on the network. Instead, the user is allowed to directly program how each variables are to be manipulated in the network. What we propose in this article is an analogue of the *define-by-run* DL framework for hyperparameter optimization, in which the framework asks the user to directly program the *parameter search-space* (See Table 1) . Armed with the architecture built on the *define-by-run* principle, our *Optuna* can express highly sophisticated search space at ease.

## 3 EFFICIENT SAMPLING AND PRUNING MECHANISM

In general, the cost-effectiveness of hyperparameter optimization framework is determined by the efficiency of (1) *searching strategy* that determines the set of parameters that shall be investigated, and (2) *performance estimation strategy* that estimates the value of currently investigated parameters from learning curves and determines the set of parameters that shall be discarded. As we will experimentally show later, the efficiency of both searching strategy and performance estimation strategy is necessary for cost-effective optimization method.

The strategy for the termination of unpromising *trials* is often referred to as *pruning* in many literatures, and it is also well known as *automated early stopping* [5] [7]. We, however, refer to this functionality as *pruning* in order to distinguish it from the *early stopping regularization* in machine learning that exists as a countermeasure against overfitting. As shown in table 2, many existing frameworks do not provide efficient pruning strategies. In this section we will provide our design for both *sampling* and *pruning*.

### 3.1 Sampling Methods on Dynamically Constructed Parameter Space

There are generally two types of sampling method: *relational sampling* that exploits the correlations among the parameters and *independent sampling* that samples each parameter independently. The *independent sampling* is not necessarily a naive option, because some sampling algorithms like TPE [6] are known to perform well even without using the parameter correlations, and the cost effectiveness for both relational and independent sampling depends on environment and task. Our *Optuna* features both, and it can handle various independent sampling methods including TPE as well as relational sampling methods like CMA-ES. However, some words of caution are in order for the implementation of *relational sampling* in *define-by-run* framework.

#### *Relational sampling in define-by-run frameworks*

One valid claim about the advantage of the old *define-and-run* optimization design is that the program is given the knowledge of the concurrence relations among the hyperparamters from the beginning of the optimization process. Implementing of optimization methods that takes the concurrence relations among the parameters into account is a nontrivial challenge when the search spaces are dynamically constructed. To overcome this challenge, *Optuna* features an ability to identify *trial* results that are informative about the concurrence relations. This way, the framework can identify the underlying concurrence relations after some number of independent samplings, and use the inferred concurrence relation to conduct user-selected relational sampling algorithms like CMA-ES [17] and GP-BO [18]. Being an open source software, *Optuna* also allows the user to use his/her own customized sampling procedure.

### 3.2 Efficient Pruning Algorithm

Pruning algorithm is essential in ensuring the "cost" part of the cost-effectiveness. Pruning mechanism in general works in two phases. It *(1)* periodically monitors the intermediate objective values, and *(2)* terminates the *trial* that does not meet the predefined condition. In *Optuna*, '`report API`' is responsible for the monitoring functionality, and '`should_prune API`' is responsible for the premature termination of the unpromising *trials* (see Figure 5). The background algorithm of '`should_prune`' method is implemented by the family of *pruner* classes. *Optuna* features a variant of Asynchronous Successive Halving algorithm [19] , a recently developed *state of the art* method that scales linearly with the number of workers in distributed environment.

Asynchronous Successive Halving(ASHA) is an extension of Successive Halving [20] in which each worker is allowed to asynchronously execute aggressive early stopping based on provisional ranking of *trials*. The most prominent advantage of asynchronous pruning is that it is particularly well suited for applications in distributional environment; because each worker does not have to wait for the results from other workers at each

```python
import ...

def objective(trial):
    ...

    lr = trial.suggest_loguniform('lr', 1e-5, 1e-1)
    clf = sklearn.linear_model.SGDClassifier(
        learning_rate=lr)
    for step in range(100):
        clf.partial_fit(x_train, y_train, classes)

        # Report intermediate objective value.
        intermediate_value = clf.score(x_val, y_val)
        trial.report(intermediate_value, step=step)

        # Handle pruning based on the intermediate value.
        if trial.should_prune(step):
            raise TrialPruned()

    return 1.0 - clf.score(x_val, y_val)

study = optuna.create_study()
study.optimize(objective)
```

Figure 5: An example of implementation of a pruning algorithm with *Optuna*. An intermediate value is reported at each step of iterative training. The `Pruner` class stops unpromising *trials* based on the history of reported values.

**Algorithm 1:** Pruning algorithm based on Successive Halving

**Input:** target trial `trial`, current step `step`, minimum resource $r$, reduction factor $\eta$, minimum early-stopping rate $s$.

**Output:** **true** if the trial should be pruned, **false** otherwise.

$\texttt{rung} \leftarrow \max(0, \log_\eta(\lfloor \texttt{step}/r \rfloor) - s)$
**if** $\texttt{step} \neq r\eta^{s+\texttt{rung}}$ **then**
  **return false**
**end**
`value` ←get_trial_intermediate_value(`trial`, `step`)
`values` ←get_all_trials_intermediate_values(`step`)
`top_k_values` ←top_k(`values`, $\lfloor |\texttt{values}|/\eta \rfloor$)
**if** `top_k_values` = $\emptyset$ **then**
  `top_k_values` ←top_k(`values`, 1)
**end**
**return** `value` $\notin$ `top_k_values`

round of the pruning, the parallel computation can process multiple *trials* simultaneously without delay.

Algorithm 1 is the actual pruning algorithm implemented in *Optuna*. Inputs to the algorithm include the *trial* that is subject to pruning, number of steps, reducing factor, minimum resource to be used before the pruning, and minimum early stopping rate. Algorithm begins by computing the current *rung* for the *trial*, which is the number of times the *trial* has survived the pruning. The *trial* is allowed to enter the next round of the competition if its provisional ranking is within top $1/\eta$. If the number of *trials* with the same rung is less than $\eta$, the best *trial* among the *trials* with the same *rung* becomes promoted. In order to avoid having to record massive number of checkpointed configurations(snapshots), our implementation does not allow repechage. As experimentally verify in the next section, our modified implementation of Successive Halving scales linearly with the number of workers without any problem. We will present the details of our optimization performance in Section 5.2.

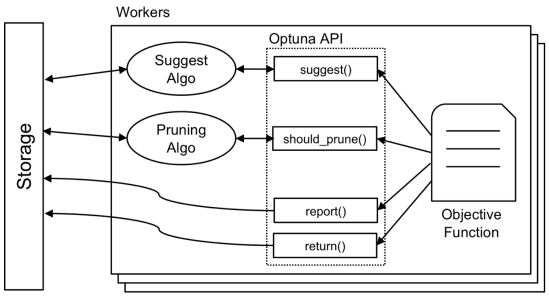



Figure 6: Overview of *Optuna*'s system design. Each worker executes one instance of an objective function in each *study*. The Objective function runs its *trial* using *Optuna* APIs. When the API is invoked, the objective function accesses the shared storage and obtains the information of the past *studies* from the storage when necessary. Each worker runs the objective function independently and shares the progress of the current *study* via the storage.

## 4 SCALABLE AND VERSATILE SYSTEM THAT IS EASY TO SETUP

Our last criterion for the next generation optimization software is a scalable system that can handle a wide variety of tasks, ranging from a heavy experiment that requires a massive number of workers to a trial-level, light-weight computation conducted through interactive interfaces like *Jupyter Notebook*. The figure 6 illustrates how the database(*storage*) is incorporated into the system of *Optuna*; the *trial* objects shares the evaluations history of objective functions via storage. *Optuna* features a mechanism that allows the user to change the storage backend to meet his/her need.

For example, when the user wants to run experiment with *Jupyter Notebook* in a local machine, the user may want to avoid spending effort in accessing a multi-tenant system deployed by some organization or in deploying a database on his/her own. When there is no specification given, *Optuna* automatically uses its built-in in-memory data-structure as the storage back-end. From general user's perspective, that the framework can be easily used for lightweight purposes is one of the most essential strengths of *Optuna*, and it is a particularly important part of our criteria for next-generation optimization frameworks. This *lightweight purpose compatibility* is also featured by select few frameworks like *Hyperopt* and *GPyOt* as well. The user of *Optuna* can also conduct more involved analysis by exporting the results in the *pandas* [21] dataframe, which is highly compatible with interactive analysis frameworks like *Jupyter Notebooks* [22]. *Optuna* also provides web-dashboard for visualization and analysis of *studies* in real time (see Figure 8).

Meanwhile, when the user wants to conduct distributed computation, the user of *Optuna* can deploy relational database as the backend. The user of *Optuna* can also use *SQLite* database as well. The figure 7b is an example code that deploys *SQLite* database. This code conducts distributed computation by simply executing run.py multiple times with the same *study* identifier and the same storage URL.

```python
import ...

def objective(trial):
    ...
    return objective_value

study_name = sys.argv[1]
storage = sys.argv[2]

study = optuna.Study(study_name, storage)
study.optimize(objective)
```

(a) Python code: run.py

```shell
# Setup: the shared storage URL and study identifier.
STORAGE_URL='sqlite:///example.db'
STUDY_ID=$(optuna create-study --storage $STORAGE_URL)

# Run the script from multiple processes and/or nodes.
# Their execution can be asynchronous.
python run.py $STUDY_ID $STORAGE_URL &
python run.py $STUDY_ID $STORAGE_URL &
python run.py $STUDY_ID $STORAGE_URL &
...
```

(b) Shell

Figure 7: Distributed optimization in *Optuna*. Figure (a) is the optimization script executed by one worker. Figure (b) is an example *shell* for the optimization with multiple workers in a distributed environment.

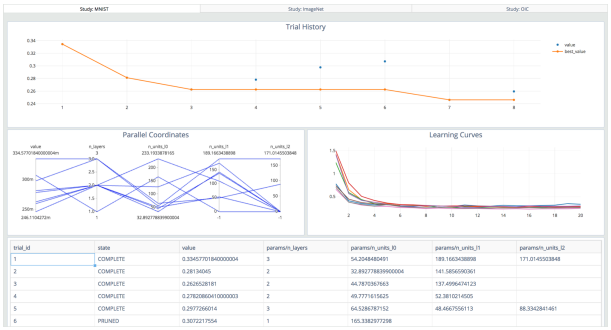



Figure 8: *Optuna* dashboard. This example shows the online transition of objective values, the parallel coordinates plot of sampled parameters, the learning curves, and the tabular descriptions of investigated *trials*.

*Optuna*'s new design thus significantly reduces the effort required for storage deployment. This new design can be easily incorporated into a container-orchestration system like *Kubernetes* as well. As we verify in the experiment section, the distributed computations conducted with our flexible system-design scales linearly with the number of workers. *Optuna* is also an open source software that can be installed to user's system with one command.

## 5 EXPERIMENTAL EVALUATION

We demonstrate the efficiency of the new design-framework through three sets of experiments.

### 5.1 *Performance Evaluation Using a Collection of Tests*

As described in the previous section, *Optuna* not only allows the user to use his/her own customized sampling procedure that suits the purpose, but also comes with multiple built-in optimization algorithms including the mixture of independent and relational sampling, which is not featured in currently existing frameworks. For example, *Optuna* can use the mixture of TPE and CMA-ES. We compared the optimization performance of the TPE+CMA-ES against those of other sampling algorithms on a collection of tests for black-box optimization [23, 24], which contains 56 test cases. We implemented four adversaries to compare against TPE+CMA-ES: random search as a baseline method, *Hyperopt* [1] as a TPE-based method, *SMAC3* [3] as a random-forest based method, and *GPyOpt* as a Gaussian Process based method. For TPE+CMA-ES, we used TPE for the first 40 steps and used CMA-ES for the rest. For the evaluation metric, we used the best-attained objective value found in 80 *trials*. Following the work of Dewancker et al. [24], we repeated each *study* 30 times for each algorithm and applied Paired Mann-Whitney U test with $\alpha = 0.0005$ to the results in order to statistically compare TPE+CMA-ES's performance against the rival algorithms.

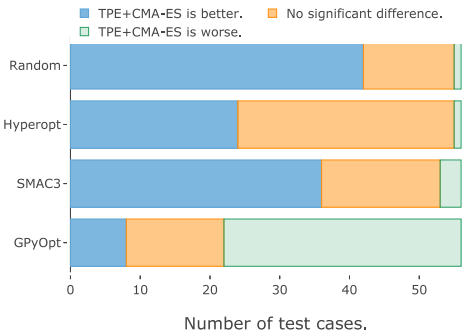



Figure 9: Result of comparing TPE+CMA-ES against other existing methods in terms of best attained objective value. Each algorithm was applied to each *study* 30 times, and Paired Mann-Whitney U test with $\alpha = 0.0005$ was used to determine whether TPE+CMA-ES outperforms each rival.

The results are shown in Figure 9. TPE+CMA-ES finds statistically worse solution than random search in only 1/56 test cases, performs worse than *Hyperopt* in 1/56 cases, and performs worse than *SMAC3* in 3/56 cases. Meanwhile, *GPyOpt* performed better than TPE+CMA-ES in 34/56 cases in terms of the best-attained loss value. At the same time, TPE+CMA-ES takes an order-of-magnitude less times per trial than *GPyOpt*.

Figure 10 shows the average time spent for each test case. TPE+CMA-ES, *Hyperopt*, *SMAC3*, and random search finished one *study* within few seconds even for the test case with more than ten design variables. On the other hand, *GPyOpt* required twenty times longer duration to complete a *study*. We see that the mixture of TPE and CMA-ES is a cost-effective choice among current lines of advanced optimization algorithms. If the time of evaluation is a bottleneck, the user may use Gaussian Process based method as a sampling algorithm. We plan in near future to also develop an interface on which the user of *Optuna* can easily deploy external optimization software as well.

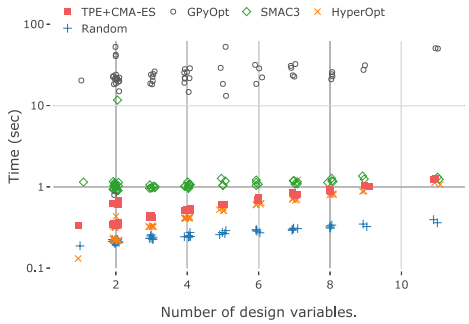



Figure 10: Computational time spent by different frameworks for each test case.

### 5.2 *Performance Evaluation of Pruning*

We evaluated the performance gain from the pruning procedure in the *Optuna*-implemented optimization of Alex Krizhevsky's neural network (AlexNet) [25] on the Street View House Numbers (SVHN) dataset [26]. We tested our pruning system together with random search and TPE. Following the experiment in [10], we used a subnetwork of AlexNet (hereinafter called simplified AlexNet), which consists of three convolutional layers and a fully-connected layer and involves 8 hyperparameters.

For each experiment, we executed a *study* with one NVIDIA Tesla P100 card, and terminated each *study* 4 hours into the experiment. We repeated each *study* 40 times. With pruning, both TPE and random search was able to conduct a greater number of *trials* within the same time limit. On average, TPE and random search *without* pruning completed 35.8 and 36.0 *trials* per *study*, respectively. On the other hand, TPE *with* pruning explored 1278.6 *trials* on average per *study*, of which 1271.5 were pruned during the process. Random search *with* pruning explored 1119.3 *trials* with 1111.3 pruned *trials*.

Figure 11a shows the transition of the average test errors. The result clearly suggests that pruning can significantly accelerate the optimization for both TPE and random search. Our implementation of ASHA significantly outperforms Median pruning, a pruning method featured in *Vizier*. This result also suggests that sampling algorithm alone is not sufficient for cost-effective optimization. The bottleneck of sampling algorithm is the computational cost required for each *trial*, and pruning algorithm is necessary for fast optimization.

### 5.3 *Performance Evaluation of Distributed Optimization*

We also evaluated the scalability of *Optuna*'s distributed optimization. Based on the same experimental setup used in Section 5.2, we recorded the transition of the best scores obtained by TPE with 1, 2, 4, and 8 workers in a distributed environment. Figure 11b shows the relationship between optimization score and execution time. We can see that the convergence speed increases with the number of workers.

In the interpretation of this experimental results, however, we have to give a consideration to the fact that the relationship between the number of workers and the efficiency of optimization is not as intuitive as the relationship between the number of *trials*. This is especially the case for a SMBO [3] such as TPE, where the algorithm is designed to sequentially evaluate each *trial*. The result illustrated in Figure 11c resolves this concern. Note that the optimization scores per the number of *trials* (i.e., parallelization efficiency) barely changes with the number of workers. This shows that the performance is linearly scaling with the number of *trials*, and hence with the number of workers. Figure 12 illustrates the result of optimization that uses both parallel computation and pruning. The result suggests that our optimization scales linearly with the number of workers even when implemented with a pruning algorithm.

## 6 REAL WORLD APPLICATIONS

*Optuna* is already in production use, and it has been successfully applied to a number of real world applications. *Optuna* is also being actively used by third parties for various purposes, including projects based on *TensorFlow* and *PyTorch*. Some projects use *Optuna* as a part of pipeline for machine-learning framework (e.g., *redshells*[2], *pyannote-pipeline*[3]). In this section, we present the examples of *Optuna*'s applications in the projects at Preferred Networks.

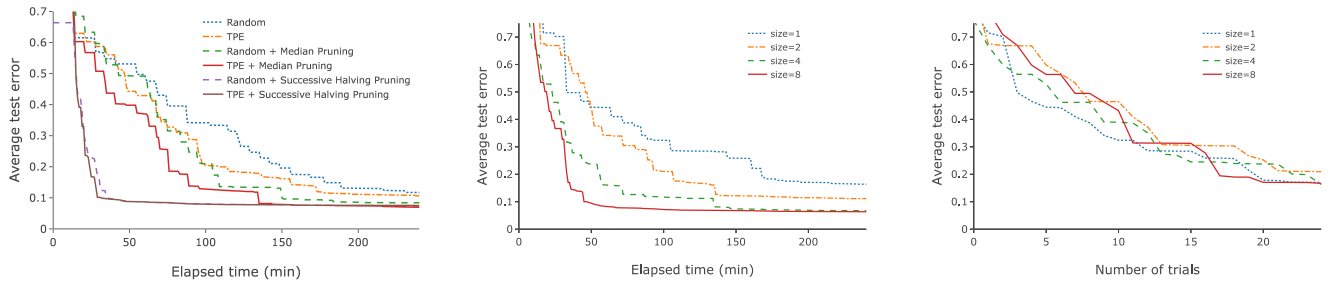
**Open Images Object Detection Track 2018.** *Optuna* was a key player in the development of Preferred Networks' Faster-RCNN models for Google AI Open Images Object Detection Track 2018 on Kaggle [4], whose dataset is at present the largest in the field of object detection [27]. Our final model, PFDet [28], won the 2nd place in the competition.

As a versatile next generation optimization software, *Optuna* can be used in applications outside the field of machine learning as well. Followings are applications of *Optuna* for non-machine learning tasks.

**High Performance Linpack for TOP500.** The *Linpack* benchmark is a task whose purpose is to measure the floating point computation power of a system in which the system is asked to solve a dense matrix LU factorization. The performance on this task is used as a measure of sheer computing power of a system and is used to rank the supercomputers in the TOP500 list[5]. *High Performance Linpack* (HPL) is one of the implementations for Linpack. HPL involves many hyperparameters, and the performance result of any system heavily relies on them. We used *Optuna* to optimize these hyperparameters in the evaluation of the maximum performance of *MN-1b*, an in-house supercomputer owned by Preferred Networks.

***RocksDB.*** *RocksDB* [29] is a persistent key-value store for fast storage that has over hundred user-customizable parameters. As described by the developers in the official website, "configuring *RocksDB* optimally is not trivial", and even the "*RocksDB* developers don't fully understand the effect of each configuration change"[6]. For this experiment, we prepared a set of 500,000 files

[2] https://github.com/m3dev/redshells
[3] https://github.com/pyannote/pyannote-pipeline
[4] https://www.kaggle.com/c/google-ai-open-images-object-detection-track
[5] https://www.top500.org/
[6] https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide#final-thoughts




(a) (b) (c)

Figure 11: The transition of average test errors of simplified AlexNet for SVHN dataset. Figure (a) illustrates the effect of pruning mechanisms on TPE and random search. Figure (b) illustrates the effect of the number of workers on the performance. Figure (c) plots the test errors against the number of *trials* for different number of workers. Note that the number of workers has no effect on the relation between the number of executed *trials* and the test error. The result also shows the superiority of ASHA pruning over median pruning.

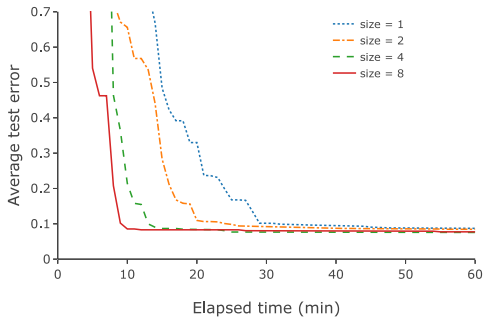



Figure 12: Distributed hyperparameter optimization process for the minimization of average test errors of simplified AlexNet for SVHN dataset. The optimization was done with ASHA pruning.

of size 10KB each, and used *Optuna* to look for parameter-set that minimizes the computation time required for applying a certain set of operations(store, search, delete) to this file set. Out of over hundred customizable parameters, we used *Optuna* to explore the space of 34 parameters. With the default parameter setting, *RocksDB* takes 372seconds on HDD to apply the set of operation to the file set. With pruning, *Optuna* was able to find a parameter-set that reduces the computation time to 30 seconds. Within the same 4 hours, the algorithm with pruning explores 937 sets of parameters while the algorithm without pruning only explores 39. When we disable the time-out option for the evaluation process, the algorithm without pruning explores only 2 *trials*. This experiment again verifies the crucial role of pruning.

**Encoder Parameters for *FFmpeg*.** *FFmpeg*[7] is a multimedia framework that is widely used in the world for decoding, encoding and streaming of movies and audio dataset. *FFmpeg* has numerous customizable parameters for encoding. However, finding of good encoding parameter-set for *FFmpeg* is a non-trivial task, as it requires expert knowledge of codec. We used *Optuna* to seek the encoding parameter-set that minimizes the reconstruction error for the Blender Open Movie Project's "Big Buck Bunny"[8]. *Optuna* was able to find a parameter-set whose performance is on par with the second best parameter-set among the presets provided by the developers.

## 7 CONCLUSIONS

The efficacy of *Optuna* strongly supports our claim that our new design criteria for next generation optimization frameworks are worth adopting in the development of future frameworks. The *define-by-run* principle enables the user to dynamically construct the search space in the way that has never been possible with previous hyperparameter tuning frameworks. Combination of efficient searching and pruning algorithm greatly improves the cost effectiveness of optimization. Finally, scalable and versatile design allows users of various types to deploy the frameworks for a wide variety of purposes. As an open source software, *Optuna* itself can also evolve even further as a next generation software by interacting with open source community. It is our strong hope that the set of design techniques we developed for *Optuna* will serve as a basis of other next generation optimization frameworks to be developed in the future.

**Acknowledgement.** The authors thank R. Calland, S. Tokui, H. Maruyama, K. Fukuda, K. Nakago, M. Yoshikawa, M. Abe, H. Imamura, and Y. Kitamura for valuable feedback and suggestion.

## 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):14008, 2015.

[2] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, pages 2951–2959, 2012.

[3] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm

[7] https://www.ffmpeg.org/

[8] Blender Foundation — www.blender.org

configuration. In *LION*, pages 507–523, 2011. ISBN 978-3-642-25565-6.

[4] Patrick Koch, Oleg Golovidov, Steven Gardner, Brett Wujek, Joshua Griffin, and Yan Xu. Autotune: A derivative-free optimization framework for hyperparameter tuning. In *KDD*, pages 443–452, 2018. ISBN 978-1-4503-5552-0.

[5] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google Vizier: A service for black-box optimization. In *KDD*, pages 1487–1495, 2017. ISBN 978-1-4503-4887-4.

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, pages 2546–2554, 2011.

[7] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. In *ICML Workshop on AutoML*, 2018.

[8] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, pages 3460–3468, 2015.

[9] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with Bayesian neural networks. In *ICLR*, 2017.

[10] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[11] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017. URL http://arxiv.org/abs/1712.05889.

[12] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, editors. *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.

[13] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *NIPS Workshop on Machine Learning Systems*, 2015.

[14] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. Dynet: The dynamic neural network toolkit. *CoRR*, abs/1701.03980, 2017.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[17] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195, 2001.

[18] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[19] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. In *NeurIPS Workshop on Machine Learning Systems*, 2018.

[20] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

[21] Wes McKinney. Pandas: a foundational python library for data analysis and statistics. In *SC Workshop on Python for High Performance and Scientific Computing*, 2011.

[22] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[23] Michael McCourt. Benchmark suite of test functions suitable for evaluating black-box optimization strategies. https://github.com/sigopt/evalset, 2016.

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

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[26] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[27] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper R. R. Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. The open images dataset V4: unified image classification, object detection, and visual relationship detection at scale. *CoRR*, abs/1811.00982, 2018.

[28] Takuya Akiba, Tommi Kerola, Yusuke Niitani, Toru Ogawa, Shotaro Sano, and Shuji Suzuki. PFDet: 2nd

place solution to open images challenge 2018 object detection track. In *ECCV Workshop on Open Images Challenge*, 2018.

[29] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, 2017.