import optuna

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

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,

Как?

Умный перебор

- Независимый перебор
- Относительный перебор

Отбраковывание

- Запоминаем промежуточные лоссы
- Выбрасываем не многообещающие

```python
import optuna
import ...


def objective(trial):
    #  создаем гиперпараметры

    n_layers = trial.suggest_int('n_layers', 1, 4)

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


    #  создаем, учим и оцениваем модель
    clf = MLPClassifier(hidden_layer_sizes=tuple(layers), max_iter=200, random_state=42)
    mnist = fetch_openml('mnist_784', version=1, as_frame=False)
        X_train, X_test, y_train, y_test = train_test_split(
        mnist.data, mnist.target, test_size=0.2, random_state=42
    )
    clf.fit(X_train, y_train)
    accuracy = clf.score(X_test, y_test)
    return 1.0 - accuracy


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

1

2

3

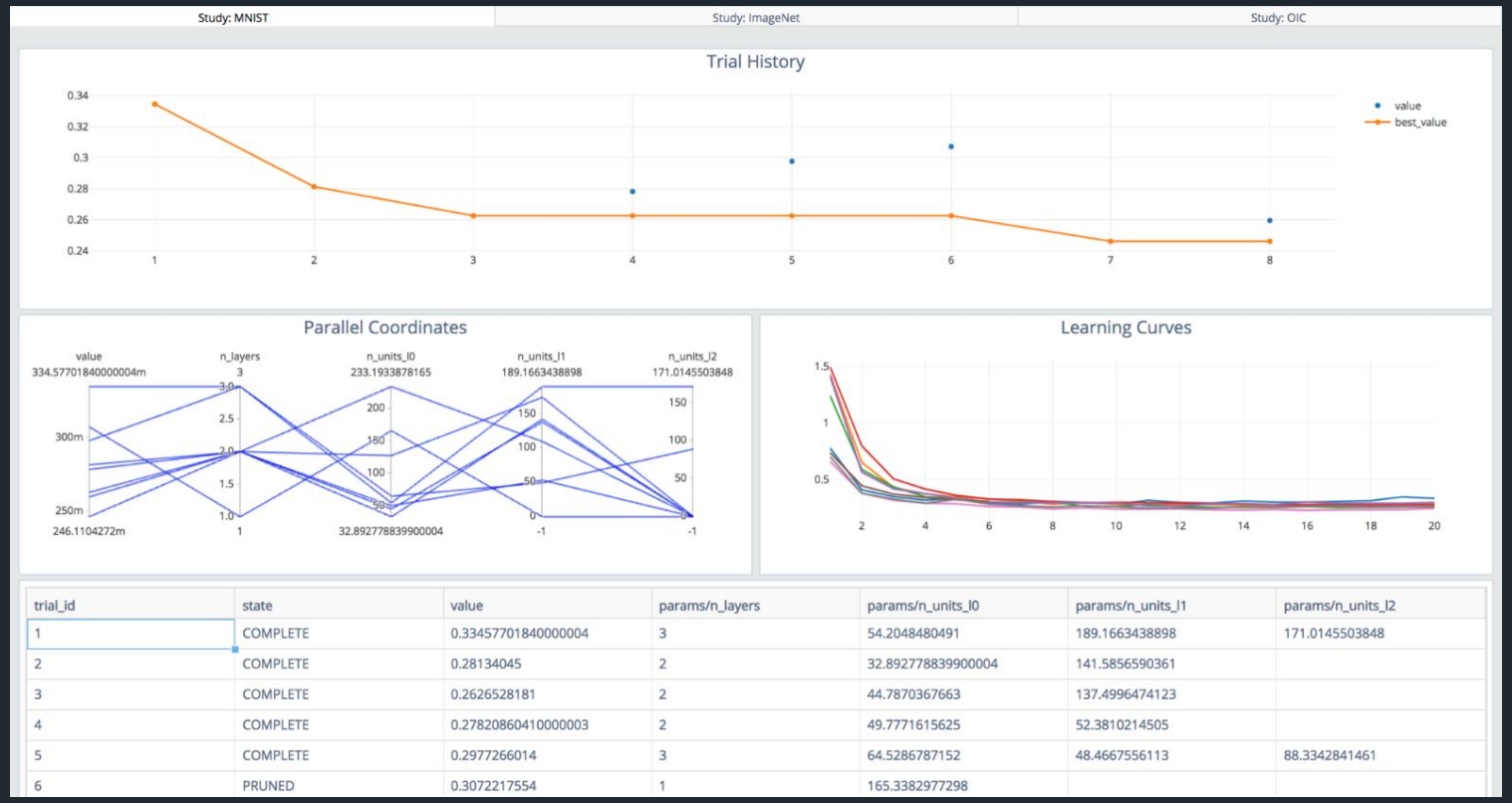
4

5

OGC

```
n_layers = trial.suggest_int("n_layers", 1, 4)

dropout = trial.suggest_float("dropout", 0.0, 0.5)

lr = trial.suggest_float("lr", 1e-5, 1e-2, log=True)

activation = trial.suggest_categorical("activation", ["relu", "tanh", "sigmoid"])

use_batchnorm = trial.suggest_bool("use_batchnorm")
```

Приятности



trial_id	state	value	params/n_layers	params/n_units_l0	params/n_units_l1	params/n_units_l2
1	COMPLETE	0.33457701840000004	3	54.2048480491	189.1663438898	171.0145503848
2	COMPLETE	0.28134045	2	32.892778839900004	141.5856590361	
3	COMPLETE	0.2626528181	2	44.7870367663	137.4996474123	
4	COMPLETE	0.27820860410000003	2	49.7771615625	52.3810214505	
5	COMPLETE	0.2977266014	3	64.5286787152	48.4667556113	88.3342841461
6	PRUNED	0.3072217554	1	165.3382977298