Member-only story

OPTUNA: A Flexible, Efficient and Scalable Hyperparameter Optimization Framework

A Novel Alternative for Light and Large-Scale Hyperparameter Optimization

Fernando López · Follow

Published in Towards Data Science · 7 min read · Feb 7, 2021

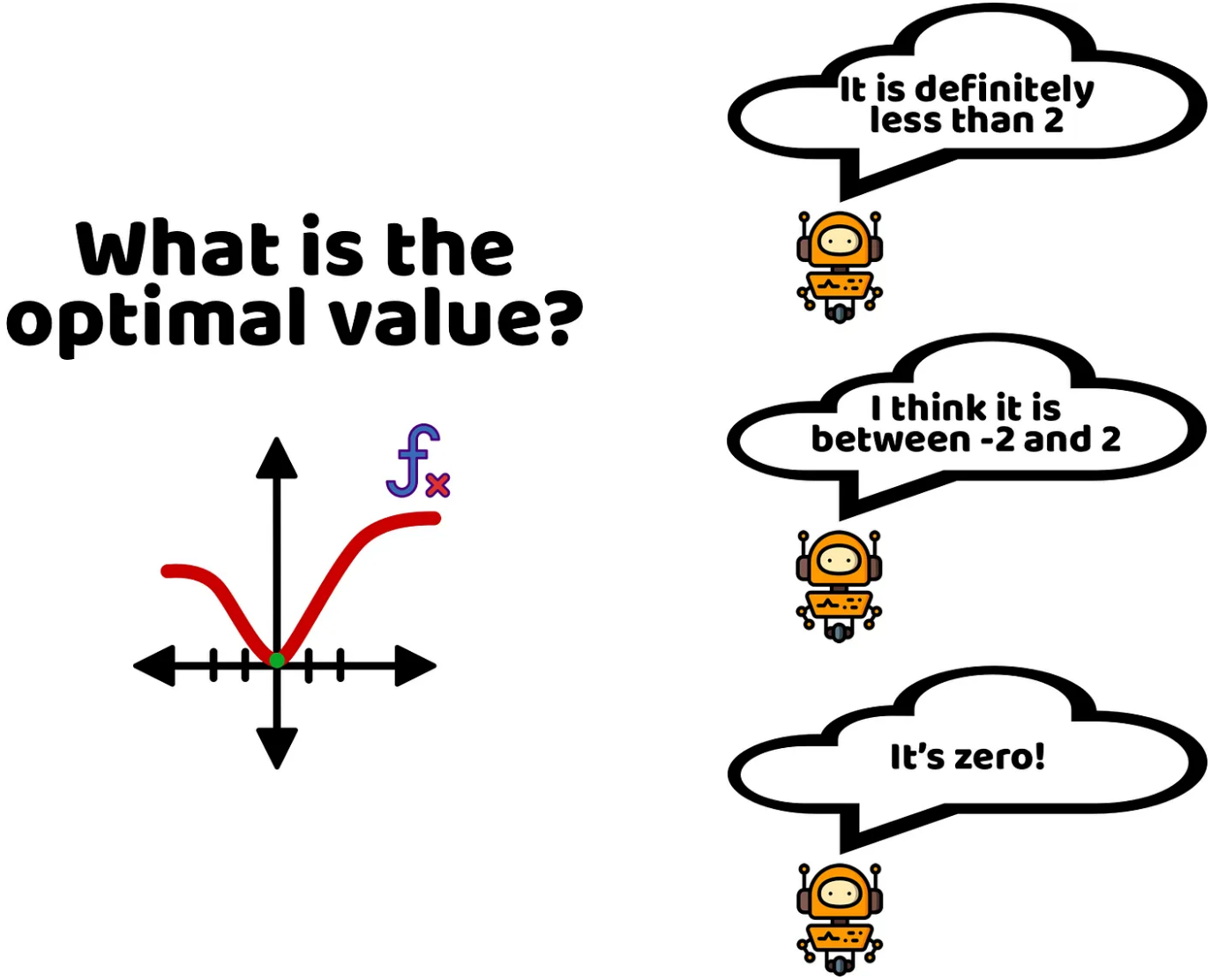


Figure 1. Cartoon about Hyperparameter Optimization | Image by author | Icons taken from smashicons, freepik

One of the determining tasks when building machine learning models is *hyperparameter optimization*. A correct optimization of hyperparameters is directly reflected in the performance of the model. This is why

hyperparameter optimization has been an active research area for several years. Fortunately, today there are several alternatives that can be followed for optimizing machine learning models such as: *HyperOpt* [1], *Spearmint* [2], *Vizer* [3], *AutoSklearn* [4] or *Autotune* [5]. Each of these alternatives propose various optimization paradigms. Likewise, each of these optimization tools proposes a different usability approach which can become more or less flexible depending on the case.

In this context **Optuna** [6] appears, whose objective is to *unify the optimization paradigms* under a philosophy supported by three pillars: *design-by-run API*, *efficient implementation* and *easy-to-setup*. Therefore, in this blog we will see what **Optuna** is, its components, its optimization paradigm and how to put it into practice with **Scikit-Learn** and **PyTorch**. So this blog will be divided into:

- **What is Optuna?**
- **Optuna & Scikit-Learn Integration**
- **Optuna & PyTorch Integration**

What is Optuna?

Optuna was introduced by Takuya Akiba et. al. [6] in 2019. **Optuna** is an open-source python library for *hyperparameter optimization*. In the background, **Optuna** aims to balance the *sampling* and *pruning* algorithms. **Optuna** implements sampling algorithms such as **Tree-Structured of Parzen Estimator** (TPE) [7, 8] for independent parameter sampling as well as **Gaussian Processes** (GP) [8] and **Covariance Matrix Adaptation** (CMA) [9] for relational parameter sampling which aims to exploit the correlation between parameters. Likewise, **Optuna** implements a variant of the **Asynchronous Successive Halving** (ASHA) [10] algorithm for the *pruning* of search spaces.

Optuna emerges as a *hyperparameter optimization* software under a new *design-criteria* which is based on three fundamental ideas: ***define-by-run API*** which allows users to construct and manipulate search spaces in a dynamic way, ***efficient implementation*** that focuses on the optimal functionality of sampling strategies as well as pruning algorithms, and ***easy-to-setup*** that focuses on versatility, that is, it allows optimizing functions in light environments as well as large-scale experiments in environments based on distributed and parallel computing. In Figure 2 we can see a visual description of the **Optuna** architecture.

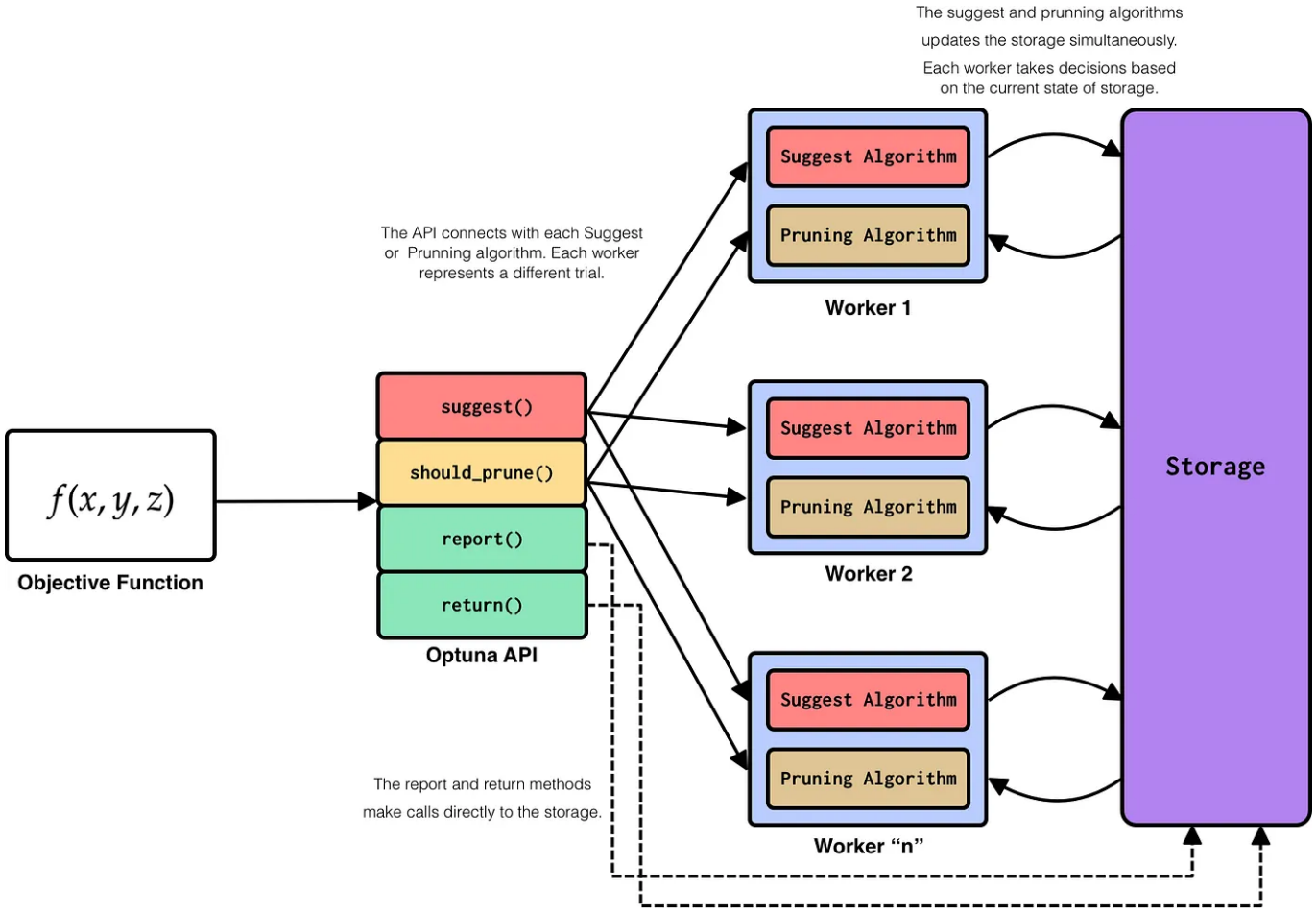


Figure 2. Optuna system architecture | Image by author

The criteria on which **Optuna** is designed makes it easy to implement, flexible and scalable. Due to **Optuna**'s scalability property, optimization of large-scale experiments can be performed in a parallel and distributed manner. **Optuna** is framework agnostic, that is, it can be easily integrated with any of the machine learning and deep learning frameworks such as: **PyTorch**, **Tensorflow**, **Keras**, **Scikit-Learn**, **XGBoost**, etc.

The implementation of **Optuna** is relatively simple and intuitive. In code snippet 1 we can see a skeleton of a basic **Optuna** implementation. Basically it is required to define the python function that will act as a *wrap* of the procedure to obtain the value to be optimized (minimize or maximize). This procedure consists of 3 essential steps, the *definition of the search space*, the *implementation of the model* and the *obtaining of the value* to be optimized.

```
1  def optimize(self, trial):
2
3      # [1] Search Spaces Definition
4      # E.g:
5      hyperparamter_a = trial.suggest_int('hyper_a', 0, 10)
6
7      # [2] Your PyTorch, Scikit-learn, Keras, etc, model.
8      # E.g:
9      model = Model(hyperparamter_a)
10     model.fit(x_train, y_train)
11
12     # [3] Function to be optimized (maximized or minimized:
13     # E.g: error, accuracy, mse, etc.
14     error = model.get_error()
15
16     # Value to be maximized or minimized
17     # In this case, error aims to be minimized
18     return error
19
20 # Study definition
21 study = optuna.create_study(direction='minimize')
22
23 # Starts optimization
```

```
study.optimize(optimize, n_trials=100)
```

optimization_skeleton.py hosted with ❤️ by GitHub view raw

Great, until now we already know what **Optuna** is, its components and attributes. Let's now see a couple of **integrations.** In the next section we will see the integration of **Optuna** with **Scikit-Learn** and later we will see the integration with **PyTorch**, let's go for it!

Optuna & Scikit-Learn Integration

In this example we are going to address a *classification problem* by using the well-known *breast cancer dataset* and the *Random Forest* algorithm. The idea is to implement **Optuna** for optimizing each of the *Random Forest hyperparameters* in order to **maximize** the *average accuracy* that will be generated through the K-fold cross validation procedure.

```
def optimize(self, trial):

    # Definition of space search
    criterion = trial.suggest_categorical('criterion', ['gini', 'entropy'])
    n_estimators = trial.suggest_int('n_estimators', 10, 100)
    max_depth = trial.suggest_int('max_depth', 2, 10)

    # Classifier definition
    model = RandomForestClassifier(n_estimators=n_estimators,
                                   max_depth=max_depth,
                                   criterion=criterion)

```

```
    avg_accuracy = []

    # Definition of k-fold cross validation
    k_fold = KFold(n_splits=5)

    for train_idx, test_idx in k_fold.split(x, y):

        # Training fold
        x_train = x[train_idx]
        y_train = y[train_idx]

        # Testing fold
        x_test = x[test_idx]
        y_test = y[test_idx]

        # Training
        model.fit(x_train, y_train)

        # Save accuracy
        avg_accuracy.append(model.score(x_test, y_test))

    return np.mean(avg_accuracy)
```

optuna_sklearn.py hosted with ❤ by GitHub view raw

```
# Study initialization
# direction = 'maximize' : since the goal is to maximize the accuracy
# sampler = 'TPEsampler' : default paramter for single-objective optimization
# pruner = 'MedianPruner' : default paramter for pruning useless configurations
study = optuna.create_study(direction='maximize')

# Opimization
# Recevies the function to be optimized and the number of trials
study.optimize(optimize, n_trials=50)
```

```
study.optimize(optimize, n_trials=50)
```

optuna_sklearn_study.py hosted with ❤️ by GitHub view raw

Finally, when performing the optimization, obtaining the *optimal hyperparameters* and training the model with such an optimal configuration, we obtain:

```
Best average accuracy: 0.9666356155876418
Best parameters: {'criterion': 'entropy', 'n_estimators': 40,
'max_depth': 6}

Train score: 1.0
Test score: 0.982456140350877
```

> *You can find the full implementation of this example here: https://github.com/FernandoLpz/Optuna-Sklearn-PyTorch*

As we can see, the integration of **Optuna** and **Scikit-Learn** is relatively simple and intuitive. In general terms, it is enough to wrap the **Scikit-Learn** model and return the value to be optimized. The definition of the **Optuna**'s study allows us to determine if the procedure will be *maximization* or *minimization*.

Now let's go to see a more robust integration with **PyTorch** where we will try to find the *optimal hyperparameters* and even the *optimal number of layers* that

the neural network should contain, let's go for it!

Optuna & PyTorch Integration

In this example we will address a *multiclass-classification* problem making use of the well-known MNIST dataset. For practical purposes, the neural network that we will implement will consist only of *linear layers*. The idea is to use **Optuna** to find the *optimal hyperparameters* of the model such as: the *optimizer* and the *learning rate*. Furthermore, we will use **Optuna** to find the *optimal number of layers* in the neural network as well as the *optimal number of units* for each layer.

So let's start first with the definition of the neural network:

Code snippet 4. Neural Net definition

One of the advantages of **Optuna** is that it allows us to define objects that define search spaces dynamically. In this case, we first initialize an empty stack where the objects that define the linear layers and the dropouts determined by *trial* (lines 6 and 7) will be dynamically stored. Subsequently, the search space for the number of layers and the dropout is defined (lines 11 and 12). Then begins the process of introducing to the stack a dynamic number of linear layers defined by trial as well as the number of units for each layer. It is important to mention that the input dimension of the first layer and the output dimension of the last layer will always be static (lines 15 and 29 respectively). The dimensions of the remaining layers will be determined by *trial* (line 20). At this point we already have a stack of linear layers with their respective input and output dimensions and a stack of dropouts. However, in **PyTorch** the class that defines the neural network has to have instance variables that refer to the objects that define each component of the neural network. In this case, each layer and dropout defined within the stacks must be defined as instance variables (lines 36 and 39). Finally the **forward** function. Since the number of layers and dropouts are already defined in the stacks, we just need to unfold the stacks and pass the *input tensor x* over each layer and dropout.

As we have already observed, the definition of the neural network does not imply complexity, only are considered the dynamic assignments of each component of the network. Now let's see how to train this neural network for *hyperparamter optimization*.

Code snippet 5. Optimization function

Considering the skeleton shown in code snippet 1, the *wrapper* function defined in code snippet 5 would be the equivalent for the optimization of the neural network. First the neural network is initialized by passing it *trial* as parameter (line 4), then the search spaces for the *optimizer* and *learning rate* are defined and that's all. The rest is to train the neural network by batches and calculate the accuracy in the test set, such accuracy is the value that is returned and the one that **Optuna** uses to perform the optimization.

Finally, when carrying out the optimization, we obtain the following results:

```
Best accuracy: 0.944831498
Best parameters: {'n_layers': 2, 'dropout': 0.2048202637410447,
'output_dim_0': 13, 'output_dim_1': 24, 'optimizer': 'Adam', 'lr':
0.0030389965486299388}

Train score: 0.95349231
Test score: 0.94231892
```

> *You can find the full implementation of this example here:*
> *https://github.com/FernandoLpz/Optuna-Sklearn-PyTorch*

Conclusion

In this blog we learned what **Optuna** is, what is the philosophy on which Optuna was developed, its components and a couple of examples of **Optuna** integrations with **Scikit-Learn** and **PyTorch**.

There is a wide variety of alternatives for *hyperparamters optimization*. Some share similarities in optimization methodologies, sampling and pruning algorithms. Likewise, each one proposes specific characteristics for the use and implementation of each tool. In this case, **Optuna** appears as the *hyperparameters optimization* tool that focuses on the simplicity, flexibility and scalability of the implementation, which makes it a well accepted tool so far.

References

[1] Hyperopt: A Python Library for Optimizing the Hyperparameters of Machine Learning Algorithms

[2] Spearmint Repository

[3] Vizer

[4] Auto-Sklearn 2.0: The Next Generation

[5] Autotune: A Derivative-free Optimization Framework for Hyperparameter Tuning

[6] Optuna: A Next-generation Hyperparameter Optimization Framework

[7] Tree-structured Parzen Estimator

[8] Algorithms for Hyper-Parameter Optimization

[9] Completely Derandomized Self-Adaptation in Evolution Strategies

[10] A System for Massively Parallel Hyperparameter Tuning

Automl Machine Learning Scikit Learn Pytorch Editors Pick

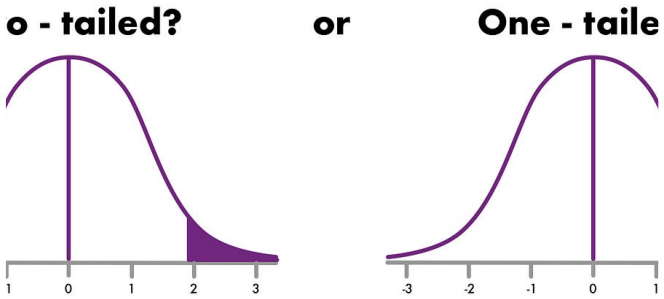
Written by Fernando López

Follow

542 Followers · Writer for Towards Data Science

Machine Learning Engineer | Data Scientist | Software Engineer

More from Fernando López and Towards Data Science



Fernando López in Towards Data Science

Hypothesis Testing: Z-Scores

A guide to understanding what hypothesis testing is and how to interpret and impleme...

Cassie Kozyrkov in Towards Data Science

The Best Learning Paths for AI and Data Leadership

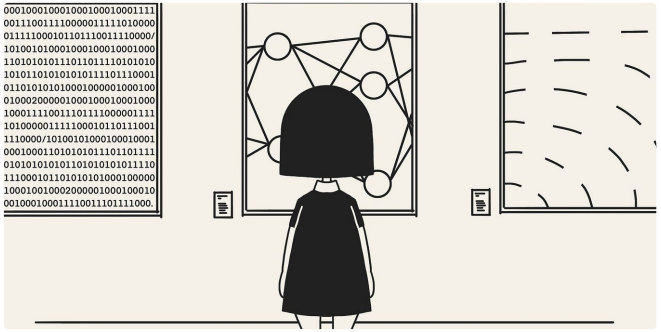
How to muscle up on data-related topics quickly

· 8 min read · Aug 29, 2021

94 4

· 7 min read · Apr 30

1.4K 8



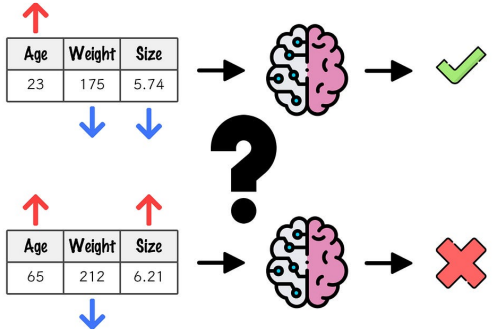
Leonie Monigatti in Towards Data Science

10 Exciting Project Ideas Using Large Language Models (LLMs) f...

Learn how to build apps and showcase your skills with large language models (LLMs). Ge...

· 11 min read · May 15

640 4



Fernando López in Towards Data Science

SHAP: Shapley Additive Explanations

A step-by-step guide for understanding how SHAP works and how to interpret ML model...

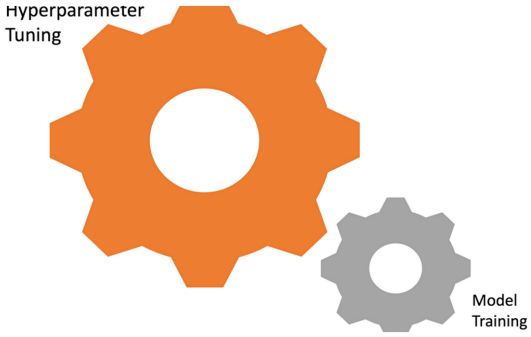
· 12 min read · Jul 12, 2021

234 3

See all from Fernando López

See all from Towards Data Science

Recommended from Medium



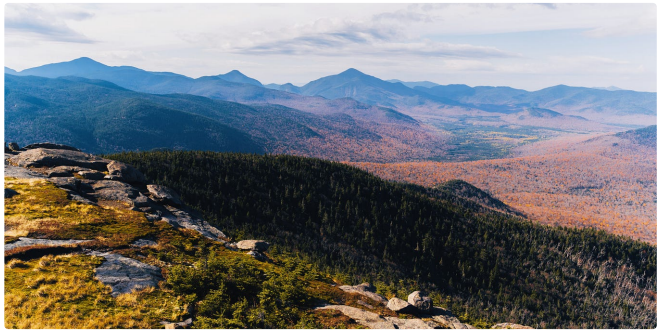
Sandeep Singh Sandha

Parallel Hyperparameter Tuning in Python: An Introduction

This post assumes introductory experience in machine learning pipelines. However, the...

5 min read · Dec 30, 2022

3 1



Ani Madurkar in Towards Data Science

Training XGBoost with MLflow Experiments and HyperOpt Tuning

A starting point on your MLOps Journey

· 10 min read · Jan 9

150

Lists

What is ChatGPT?

9 stories · 49 saves

Staff Picks

320 stories · 81 saves

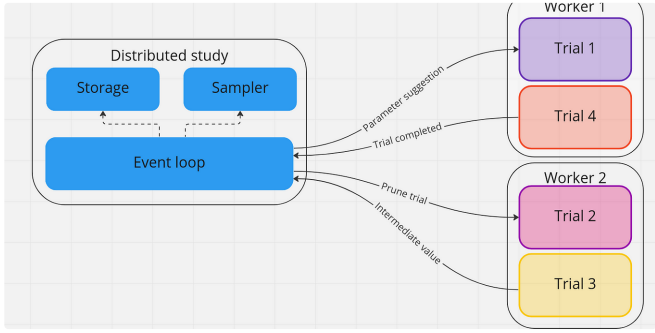
Gülsüm Budakoğlu in MLearning.ai

Hyper-parameter Tuning Through Grid Search and Optuna

Giving Information about Hyper-parameter Tuning Methods and Code Analysis of Grid...

5 min read · Mar 26

151 1



Adrian Zuber in Optuna

Running distributed hyperparameter optimization wit...

7 min read · Nov 30, 2022

15 2

Markus Lauber in Low Code for Data Science

Hyperparameter optimization for LightGBM—wrapped in KNIME...

TL;DR: Let a KNIME node find the right hyperparameters for your LightGBM ML...

12 min read · Feb 4

74

RITHP

XGBoost and imbalanced datasets: Strategies for handling class...

XGBoost is a powerful and widely used gradient boosting library for machine...

3 min read · Dec 28, 2022

23 1

See more recommendations