Medium Search Write

Get unlimited access to the best of Medium for less than $1/week. Become a member

Optuna

Running Distributed Hyperparameter Optimization with Optuna-distributed

Adrian Zuber · Follow

Published in Optuna · 7 min read · Nov 30, 2022

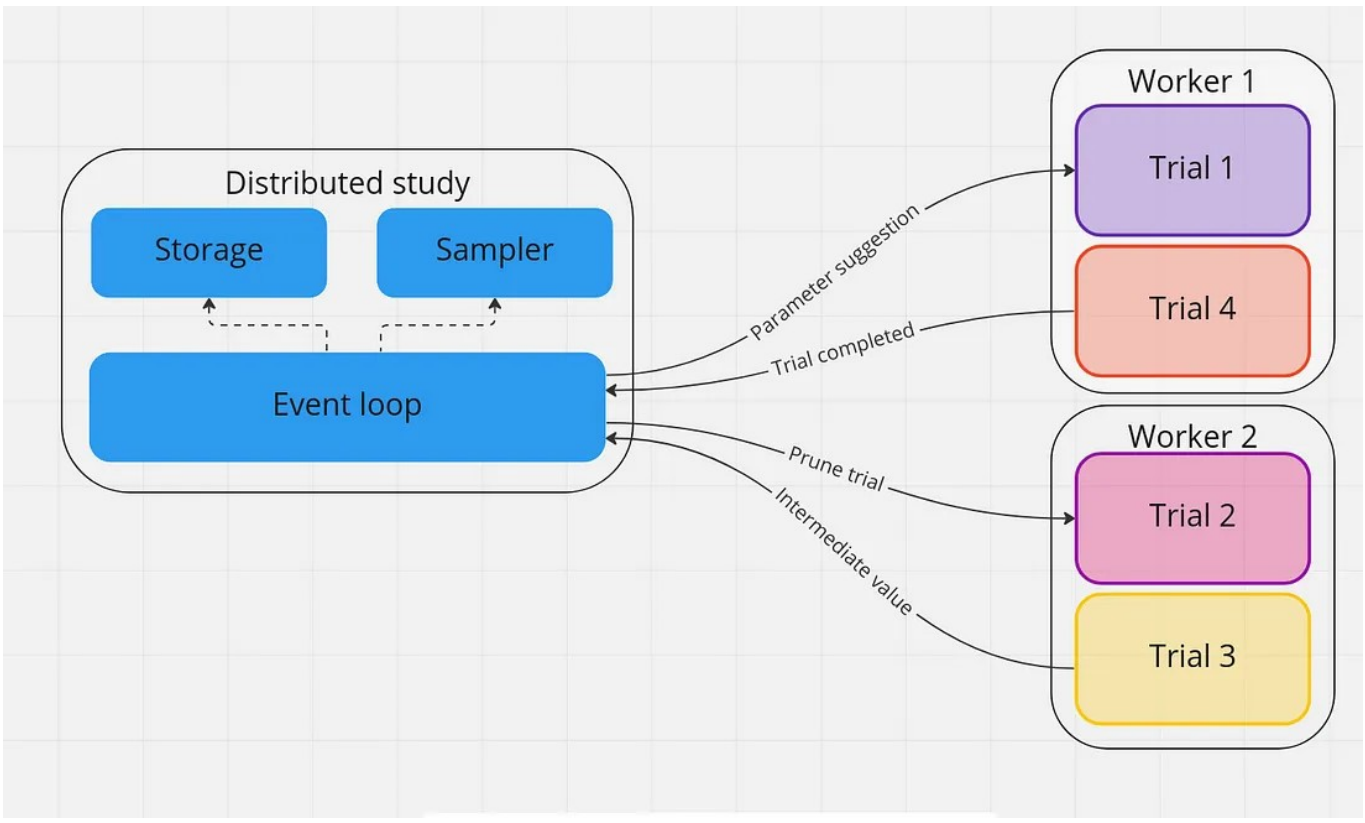
33 2

Optuna is an automatic hyperparameter optimization software framework, particularly designed for machine learning. Parallelized hyperparameter optimization is a topic that appears quite frequently in Optuna issues and discussions. There are a few approaches to it, including builtin thread-based solutions. Today, however, I'd like to present a slightly different solution, one that's based on process-based parallelism and can easily scale hyperparameter optimization tasks from one machine, to multiple machines in a cluster. Introducing Optuna-distributed! In this article I will present the architecture of Optuna-distributed and go over the details and design decisions behind each major component. We will also finish things off with a small example.

High level overview

Optuna-distributed is an extension to regular Optuna, providing tight integration with base library by wrapping Optuna's `Study` and introducing new implementation of `optimize` method (other methods behave like original), which leverages process-based parallelism by default and is able to utilize Dask cluster to distribute computation among many machines. Architecture-wise, Optuna-distributed implements actor model and splits the environment into one main (user) process and many worker processes. Main process contains the event loop which has exclusive access to standard Optuna study and as a consequence to storage, sampler and pruner. Workers are tasked with executing user-defined objective functions asynchronously, and they communicate with the main process via a simple messaging system. Within a message, a worker may ask for hyperparameter suggestions, report a new objective or intermediate value. Communication is bidirectional, so the main process might respond to a worker with data required to continue the trial. Optimization state is tracked within the event loop by entities called managers, which can spawn additional workers,

monitor progression on each trial, or request to end the optimization process altogether. Such design removes the need for lock-based synchronization at storage level, avoids sharing memory between tasks, enables usage of all standard Optuna components in distributed mode, and provides an easy way to scale asynchronous optimization process from single machine to many machines in cluster.



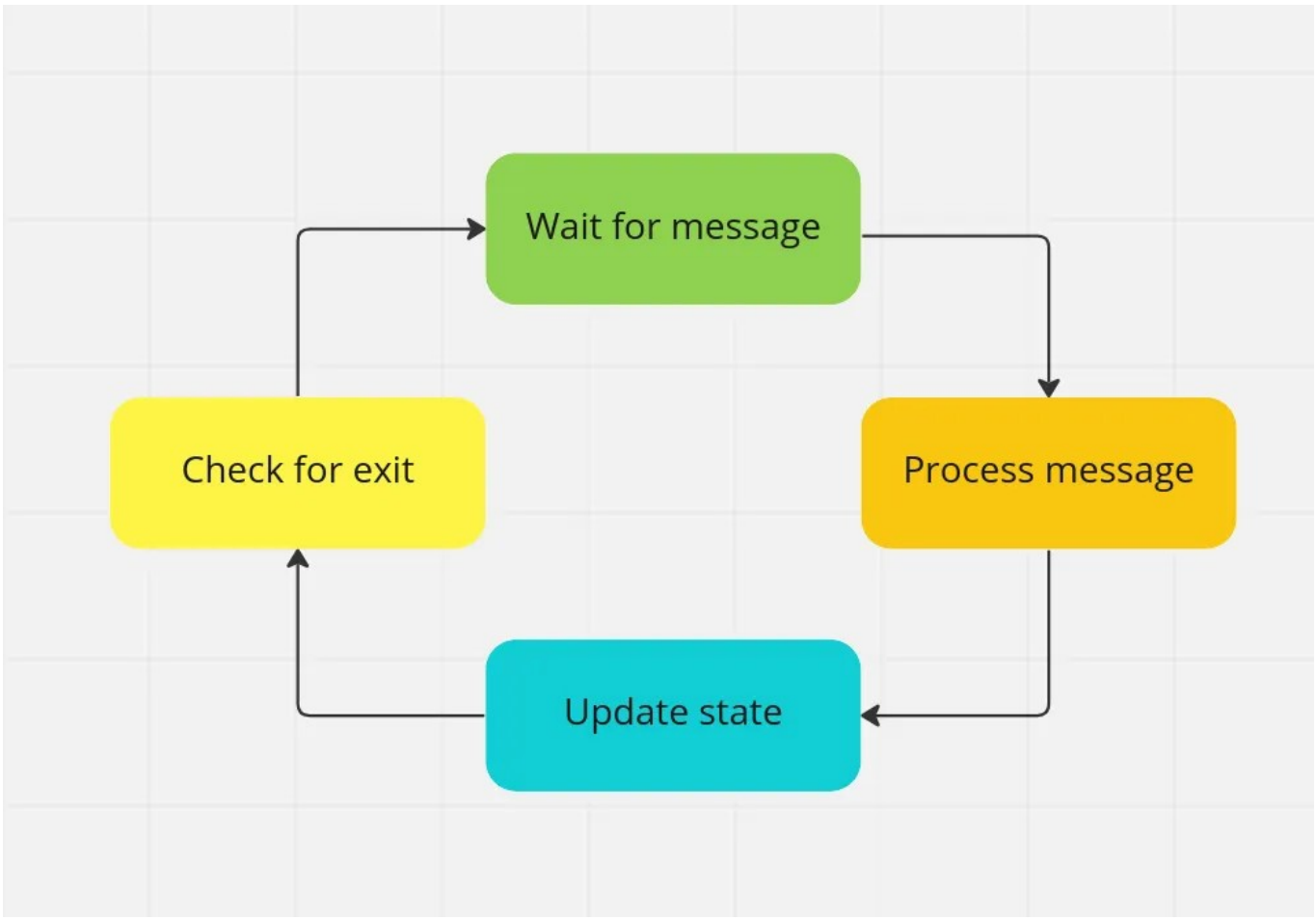
Architecture of Optuna-distributed

Components

With all major components briefly introduced, let's dive into the details.

Event loop

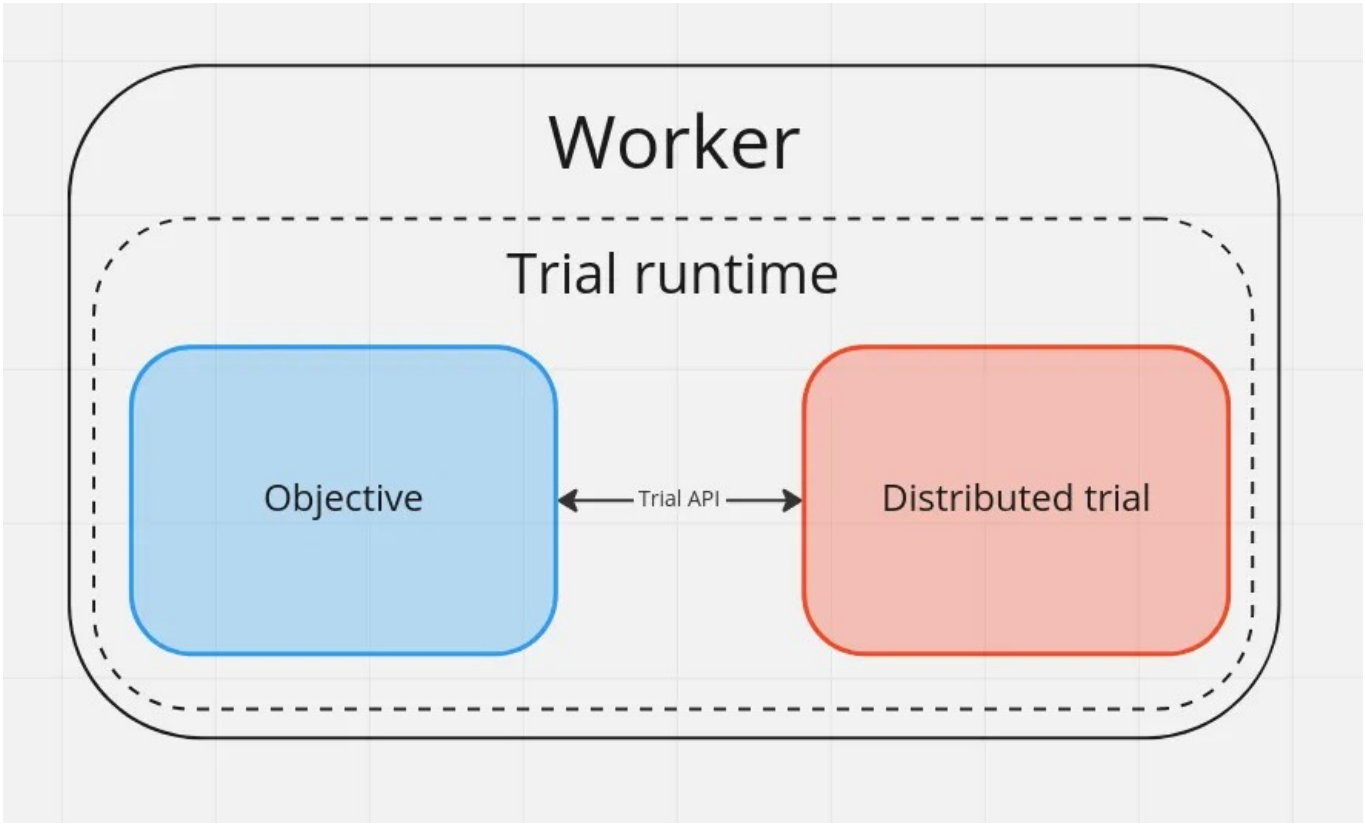
We jump into the event loop right after `optimize` was called, and stay there for the duration of the optimization process. The event loop has its own event queue and does one iteration every time a worker puts a message there. Messages are read and processed one by one, ensuring sequential access to study resources. The event loop provides these resources as a context when a message is being processed. Since messages are effectively synchronized at the event queue level, all following operations in the loop are basically equivalent to the Optuna synchronous optimization flow — with each message a progress bar is updated (when enabled of course!), exit conditions are checked and exceptions from objective function are raised. However, even though messages might wait a moment in the queue to be processed, workers who send them in many cases don't have to wait for this to happen, so things like intermediate value or status updates don't carry additional overhead.



Event loop steps

Distributed trial and objective function

To provide all trial functionalities within an objective function without direct access to storage and sampler, a new type of `Trial` is defined. On the surface, `DistributedTrial` encapsulates exactly the same functionality and is exposing the same APIs as its well-known counterpart, making already existing objective functions usable within a distributed environment. Internally, however, instead of holding direct references to study resources, we only have access to inter-process communication primitives, which allows us to send messages back to the main process and receive responses if necessary. This way we can keep the code executed outside of the main process to a minimum, as only the instance of `DistributedTrial` and user-defined objective function are sent over to a worker. This pair however needs some sort of small environment to run within, providing things such as error recovery, cleanup after a trial finishes, and in some cases nanny thread allowing it to interrupt the running objective function. Final setup looks like this:



Trial runtime

This setup relaxes serializability requirements on custom storages or samplers users might want to use, as they will always stay in the main process, but still require objective functions to be serializable. The latter, however, will always be a limitation in the case of asynchronous optimization.

Messages and IPC

So far we've seen what happens in the main process event loop and in the workers. Now it's time to discuss a bridge that connects the two — the messaging system. Messages consist of two parts — data and code. In the majority of the cases, a message is instantiated and populated with data by the worker, sent to the main process and picked up within an event loop, where message code is executed with full access to Optuna study. As an example, let's follow a path of `SuggestMessage` — a message used to request hyperparameter suggestions.

As usual in Optuna, a user may request hyperparameter suggestions by calling one of the `suggest_*` APIs within the objective function. After doing so, `DistributedTrial` instance will instantiate `SuggestMessage`, providing information on the trial that is being executed by the worker, the name of the hyperparameter and the distribution we are sampling from. After that, the message is serialized, and `DistributedTrial` uses its IPC connection to send it over to the main process. After being picked up inside the event loop, `process` method is called on the message, with study and optimization manager provided as arguments. Since now we have all the data needed to make a hyperparameter suggestion, we can call regular Optuna APIs to do so. Now that we finally have our suggested value, there is one more important step before finishing the process — we have to send it back to the worker! As previously mentioned, communication between the main process and workers is bidirectional. All we need to do is ask the optimization manager to provide a return connection to the worker that has

originally sent the message, and use it to send back the response. The worker will pick up the response data, and return it to the `suggest_*` method caller, oblivious to the fact that his hyperparameter might just have traveled many kilometers to reach him.

This pattern of message and optional response is followed to provide all regular `Trial` functionalities. Messages that don't require responses, such as intermediate value reports, do not block trial progression, as `DistributedTrial` can just send them and let the optimization process continue. This is in contrast to synchronization at the storage level, where each operation has to request and wait for a storage lock. However, the main benefit of this solution is that the IPC system is abstracted away. We don't care if messages are sent between two processes running on the same machine or between multiple machines via a network. This makes Optuna-distributed easy to scale from simple process based asynchronous optimization on your local machine, to large scale multi machine cluster-based workloads.

Managers

The last component to discuss is the optimization manager. As you might have noticed by now, managers are a neat way to provide a unified API to optimization backend and carry any state associated with it. Currently in Optuna-distributed there are two types of managers — local and distributed. The first one allows to schedule and execute trials by spawning and controlling a local process for each one, using the standard `multiprocessing` library. Second leverages Dask client to communicate with the scheduler, allowing it to run the optimization process in distributed mode on a Dask cluster. Each manager has its own flavor of `IPCPrimitive` available to use as a task queue as well as a response communication method with workers. Since managers have a full overview of progress on each trial, it's up to them to decide when the event loop should close, what should be executed before and after message processing, and which trials should be interrupted in cases where timeout is reached or the user manually interrupts the optimization process.

Example

Let's finish things off with an example! Before we start, if you would like to follow along, Optuna-distributed can be installed from PyPI by running `pip install optuna-distributed`. If you are using a Dask cluster and want to run the example in distributed mode, remember that Optuna-distributed (and all other dependencies) has to be installed on each of the workers. You can read about cluster setup in Dask documentation. There are also different examples available in the Optuna-distributed examples section, showing how to use Dask client in distributed optimization cases, presenting usage of Optuna `RDBStorage` or demonstrating optimization with pruning. In this example, however, we will locally optimize a simple quadratic function, but include a simulated "long" running process to show a difference in

evaluation times. Full code to the example is available here. Our objective function is defined as follows:

```python
def objective(trial):
    x = trial.suggest_float("x", -100, 100)
    y = trial.suggest_categorical("y", [-1, 0, 1])
    time.sleep(1.0)
    return x**2 + y
```

objective.py hosted with ❤ by GitHub view raw

As you can see, `trial` behaves exactly like in the standard Optuna. In fact, to demonstrate this, we will run two rounds of optimization with the same study object — first by using regular `optimize`, and second by wrapping the study with Optuna-distributed to run the optimization in asynchronous mode using all 4 CPU cores my machine has to offer. As a result, we should see an approximately 4x improvement in optimization times with similar best objective function values reached.

```python
sequential_start = time.time()
study = optuna.create_study()
study.optimize(objective, n_trials=20)
sequential_duration = time.time() - sequential_start
sequential_best_value = study.best_value

distributed_start = time.time()
distributed_study = optuna_distributed.from_study(study)
distributed_study.optimize(objective, n_trials=20)
distributed_duration = time.time() - distributed_start
distributed_best_value = distributed_study.best_value

print(f"Sequential optimization took {sequential_duration:.2f} seconds with best value of {sequer
print(f"Asynchronous optimization took {distributed_duration:.2f} seconds with best value of {dis
```

optimization.py hosted with ❤ by GitHub view raw

Running the above yields:

```
adrian@asus:~/code/optuna-distributed$ python example.py
Sequential optimization took 20.84 seconds with best value of 0.3077
Asynchronous optimization took 5.29 seconds with best value of 0.3077
```

Conclusion

And that is the entire Optuna-distributed in a nutshell! For now, development will continue as there are still missing features, and awkward pieces of logic to improve on, so stick around for future releases! Thank you for reading, and if you have any questions or ideas, feel free to raise an issue or open a PR. Also, big thanks to the entire Optuna dev team for having me here!

Optuna | Distributed Computing | Machine Learning | Hyperparameter Tuning | Dask

Published in Optuna

433 Followers · Last published Mar 23, 2025

Follow

A Hyperparameter Optimization Framework https://github.com/optuna/optuna/

Written by Adrian Zuber

26 Followers · 0 Following

Follow

Responses (2)

Joey Junior

What are your thoughts?

lis

Apr 22, 2023

This extension works pretty well, thank you for saving my life!

Reply

Evan Kurzman

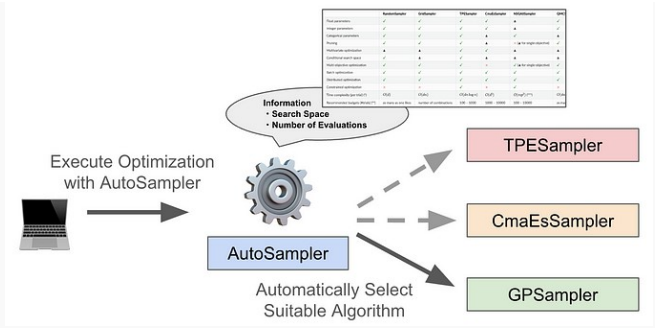
Mar 15, 2023

Tried to use this with Dask on Windows and withotu Dask and both ways got the "Local asynchronous optimization is currently not supported on Windows. "

187 "Please specify Dask client to continue in distributed mode."

Reply

More from Adrian Zuber and Optuna



In Optuna by Yoshihiko Ozaki

AutoSampler: Automatic Selection of Optimization Algorithms in...

AutoSampler automatically selects a sampler from those implemented in Optuna,...

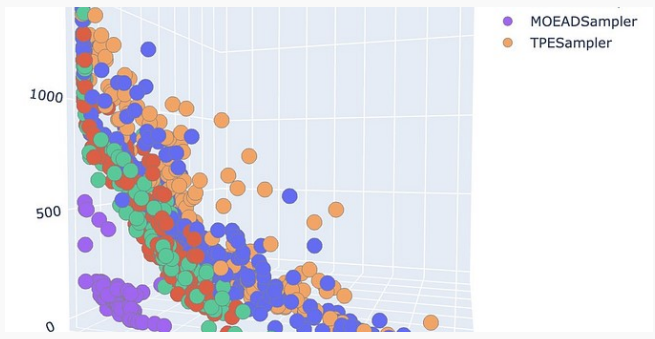
Nov 6, 2024 49 2

In Optuna by Crissman Loomis

Using Optuna to Optimize XGBoost Hyperparameters

Automate the tuning of hyperparameters in XGBoost using Bayesian Optimisation in...

Apr 27, 2020 179 3

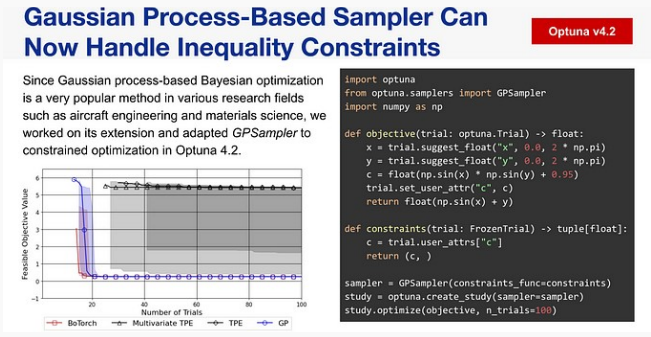


In Optuna by Hiroaki NATSUME

An introduction to MOEA/D and examples of multi-objective...

In this article, we introduce the characteristics of MOEA/D and present an...

Oct 20, 2024 14



In Optuna by Kaito Baba

[Optuna v4.2] Gaussian Process-Based Sampler Can Now Handle...

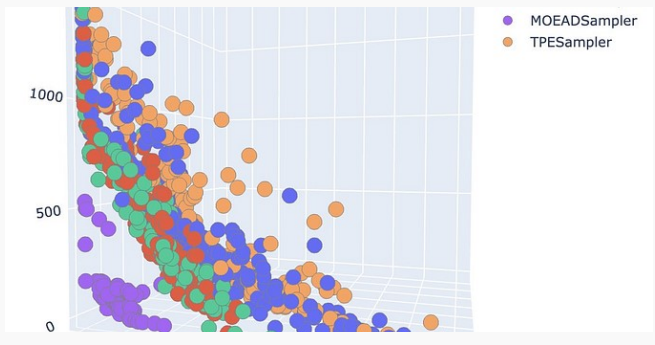
Optuna v4.2 extends GPSampler, a Gaussian process-based Bayesian optimization, to...

Mar 10 74

See all from Adrian Zuber

See all from Optuna

Recommended from Medium



In Optuna by Hiroaki NATSUME

An introduction to MOEA/D and examples of multi-objective...

Raniaabassi

Optimizing a Neural Network Using Bayesian Optimization with...

When tuning machine learning models, finding the right hyperparameters is key to...

Oct 16, 2024

Sarah Zouinina, PhD

Hyperparameter Tuning with Automation: Unlocking Peak...

In The Deep Hub by Palash Mishra

Getting Started with PyTorch: A Beginner-Friendly Guide

If you've ever wondered how to build and train deep learning models, PyTorch is one of the...

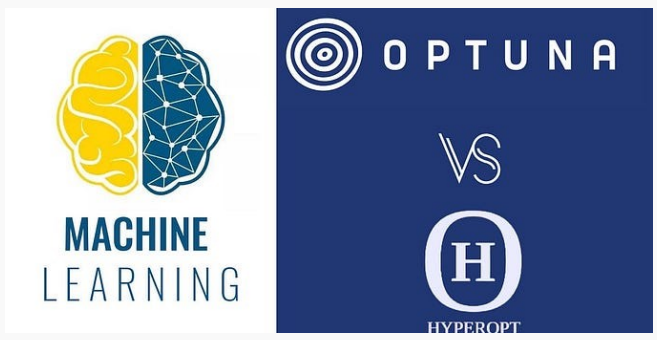
Dec 3, 2024 55

Changhyun Kim

🚀 Hyperparameter Tuning Like a Pro: A Guide to Optuna & Hyperopt

Hyperparameter tuning is one of the most important steps in building a high-...

Jan 30



ajaymehta

"Optuna vs HyperOpt: Which Framework Excels in...

Bayesian Optimization

Dec 28, 2024 10

See more recommendations