Why Is Everyone at Kaggle Obsessed with Optuna For Hyperparameter Tuning?

Let's find out by trying it out...

Bex T. Aug 2 · 7 min read ⋆

Photo by Bo Mei on Pixabay. All images are by the author unless specified otherwise.

Introduction

Turns out I have been living under a rock.

While every single MOOC taught me to use GridSearch for hyperparameter tuning, Kagglers have been using Optuna almost exclusively for 2 years. This even predates the time I started learning data science.

Kaggle community is known for its brutal competitiveness, and for a package to achieve this level of domination, it needs to be **damn good**. After being active on the platform for the last month (and achieving expert status in two tiers), I saw Optuna used almost everywhere and by everyone.

So, what makes Optuna so widely received by the largest machine learning community out there? We will answer this question in this post by getting hands-on on the framework. We will learn how it works and how it squeezes every bit of performance out of any model, including neural networks.

What is Optuna?

Optuna logo

Optuna is a next-generation automatic hyperparameter tuning framework written completely in Python.

Its most prominent features are:

- the ability to define Pythonic search spaces using loops and conditionals.
- Platform-agnostic API — you can tune estimators of almost any ML, DL package/framework, including Sklearn, PyTorch, TensorFlow, Keras, XGBoost, LightGBM, CatBoost, etc.
- a large suite of optimization algorithms with early stopping and pruning features baked in.
- Easy parallelization with little or no changes to the code.
- Built-in support for visual exploration of search results.

We will try to validate these overly optimistic claims made in Optuna's documentation in the coming sections.

Optuna basics

Let's familiarize ourselves with Optuna API by tuning a simple function like $(x-1)^2 + (y+3)^2$. We know the function reaches its minimum at $x=1$ and $y=-3$.

> *You can find the notebook for this article here.*

Let's see if Optuna can find these:

```python
import optuna  # pip install optuna

def objective(trial):
    x = trial.suggest_float("x", -7, 7)
    y = trial.suggest_float("y", -7, 7)
    return (x - 1) ** 2 + (y + 3) ** 2
```

8201.py hosted with ♡ by GitHub view raw

After importing `optuna`, we define an objective that returns the function we want to minimize.

In the body of the objective, we define the parameters to be optimized, in this case, simple `x` and `y`. The argument `trial` is a special Trial object of optuna, which does the optimization for each hyperparameter.

Among others, it has a `suggest_float` method that takes the name of the hyperparameter and the range to look for its optimal value. In other words,

```
x = trial.suggest_float("x", -7, 7)
```

is almost the same as `{"x": np.arange(-7, 7)}` when doing GridSearch.

To start the optimization, we create a `study` object from Optuna and pass the `objective` function to its `optimize` method:

```python
study = optuna.create_study()
study.optimize(objective, n_trials=100)  # number of iterations

>>> study.best_params
{'x': 0.9448382515046126, 'y': -3.074822812248314}
```

8202.py hosted with ♡ by GitHub view raw

Pretty close, but not as close as you would want. Here, we only did 100 trials, as can be seen with:

```
>>> len(study.trials)

100
```

Now, I will introduce the first magic that comes with Optuna. We can resume the optimization even after it is finished if we are not satisfied with the results!

This is a **distinct advantage** over other similar tools because after the search is done, they completely forget the history of previous trials. Optuna does not!

To continue searching, call `optimize` again with the desired params. Here, we will run 100 more trials:

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

>>> study.best_params
{'x': 1.0212303395174502, 'y': -3.015575206335039}
```

8203.py hosted with ♡ by GitHub view raw

This time, the results are much closer to the optimal parameters.

A note on Optuna terminology and conventions

In Optuna, the whole optimization process is called a *study*. For example, tuning XGBoost parameters with a log loss as a metric is one study:

```
study = optuna.create_study()

>>> type(study)
optuna.study.Study
```

8204.py hosted with ♡ by GitHub view raw

A study needs a function it can optimize. Typically, this function is defined by the user, should be named `objective` and expected to have this signature:

```
def objective(trial: optuna.Trial):
    """Conventional optimization function
    signature for optuna.
    """
    custom_metric = ...
    return custom_metric
```

8205.py hosted with ♡ by GitHub view raw

It should accept an `optuna.Trial` object as a parameter and return the metric we want to optimize for.

As we saw in the first example, a study is a collection of *trials* wherein each trial, we evaluate the objective function using a single set of hyperparameters from the given search space.

Each trial in the study is represented as `optuna.Trial` class. This class is key to how Optuna finds optimal values for parameters.

To start a study, we create a study object with `direction`:

```
study = optuna.create_study(direction="maximize")
```

8206.py hosted with ♡ by GitHub view raw

If the metric we want to optimize is a point-performance score like ROC AUC or accuracy, we set the direction to `maximize`. Otherwise, we minimize a loss function like RMSE, RMSLE, log loss, etc. by setting direction to `minimize`.

Then, we will call the `optimize` method of the study passing the objective function name and the number of trials we want:

```
# Optimization with 100 trials
study.optimize(objective, n_trials=100)
```

8207.py hosted with ♡ by GitHub view raw

Next, we will take a closer look into creating these objective functions.

Defining the search space

Usually, the first thing you do in an objective function is to create the search space using built-in Optuna methods:

```python
def objective(trial):
    rf_params = {
        "n_estimators": trial.suggest_integer(name="n_estimators", low=100, high=2000),
        "max_depth": trial.suggest_float("max_depth", 3, 8),
        "max_features": trial.suggest_categorical(
            "max_features", choices=["auto", "sqrt", "log2"]
        ),
        "n_jobs": -1,
        "random_state": 1121218,
    }

    rf = RandomForestRegressor(**rf_params)
    ...
```

8208.py hosted with ♡ by GitHub view raw

In the above objective function, we are creating a small search space of Random Forest hyperparameters.

The search space is a plain-old dictionary. To create possible values to search over, you must use the trial object's `suggest_*` functions.

These functions require at least the hyperparameter name, min, and max of the range to search over or possible categories for categorical hyperparameters.

To make the space smaller, `suggest_float` and `suggest_int` have additional `step` or `log` arguments:

```python
from sklearn.ensemble import GradientBoostingRegressor

def objective(trial):
    params = {
        "n_estimators": trial.suggest_int("n_estimators", 1000, 10000, step=200),
        "learning_rate": trial.suggest_float("learning_rate", 1e-7, 0.3, log=True),
        "max_depth": trial.suggest_int("max_depth", 3, 12, step=2),
        "random_state": 1121218,
    }
    boost_reg = GradientBoostingRegressor(**params)
    rmsle = ...
    return rmsle
```

8209.py hosted with ♡ by GitHub view raw

Above, we are binning the distribution of `n_estimators` by 200-intervals to make it sparser. Also, `learning_rate` is defined at a logarithmic scale.

How are possible parameters sampled?

Under the hood, Optuna has several classes responsible for parameter sampling. These are:

- `GridSampler` : the same as `GridSearch` of Sklearn. Never use for large search spaces!
- `RandomSampler` : the same as `RandomizedGridSearch` of Sklearn.
- `TPESampler` : Tree-structured Parzen Estimator sampler - bayesian optimization using kernel fitting
- `CmaEsSampler` : a sampler based on CMA ES algorithm (does not allow categorical hyperparameters).

> *I have no idea of how the last two samplers work and I don't expect this to affect any interaction I have with Optuna.*

TPE Sampler is used by default — it tries to sample hyperparameter candidates by improving on the last trial's scores. In other words, you can expect incremental (maybe marginal) improvements from trial to trial with this sampler.

If you ever want to switch samplers, this is how you do it:

```
from optuna.samplers import CmaEsSampler, RandomSampler

# Study with a random sampler
study = optuna.create_study(sampler=RandomSampler(seed=1121218))

# Study with a CMA ES sampler
study = optuna.create_study(sampler=CmaEsSampler(seed=1121218))
```

8210.py hosted with ♡ by GitHub view raw

End-to-end example with GradientBoosting Regressor

Let's put everything we have learned into something tangible. We will be predicting penguin body weights using several numeric and categorical features.

We will establish a base score with Sklearn `GradientBoostingRegressor` and improve it by tuning with Optuna:

```
kf = KFold(n_splits=5, shuffle=True, random_state=1121218)
scores = cross_validate(
    gr_reg, X, y, cv=kf, scoring="neg_mean_squared_log_error", n_jobs=-1
)
rmsle = np.sqrt(-scores["test_score"].mean())

>>>> print(f"Base RMSLE: {rmsle:.5f}")
Base RMSLE: 0.07573
```

8211.py hosted with ♡ by GitHub view raw

Now, we will create the `objective` function and define the search space:

```
def objective(trial, X, y, cv, scoring):
```

```
    params = {
        "n_estimators": trial.suggest_int("n_estimators", 100, 5000, step=100),
        "learning_rate": trial.suggest_float("learning_rate", 1e-4, 0.3, log=True),
        "max_depth": trial.suggest_int("max_depth", 3, 9),
        "subsample": trial.suggest_float("subsample", 0.5, 0.9, step=0.1),
        "max_features": trial.suggest_categorical(
            "max_features", ["auto", "sqrt", "log2"]
        ),
        "random_state": 1121218,
        "n_iter_no_change": 50,  # early stopping
        "validation_fraction": 0.05,
    }
    # Perform CV
    gr_reg = GradientBoostingRegressor(**params)
    scores = cross_validate(gr_reg, X, y, cv=cv, scoring=scoring, n_jobs=-1)
    # Compute RMSLE
    rmsle = np.sqrt(-scores["test_score"].mean())

    return rmsle
```

8212.py hosted with ♡ by GitHub view raw

We built a grid of 5 hyperparameters with different ranges and some static ones for random seed and early stopping.

The above objective function is slightly different — it accepts additional arguments for the data sets, scoring and `cv`. That's why we have to wrap it inside another function. Generally, you do this with a `lambda` function like below:

> *This is the recommended syntax if you want to pass `objective` functions that accept multiple parameters.*

```
%%time

# Create study that minimizes
study = optuna.create_study(direction="minimize")

# Wrap the objective inside a lambda with the relevant arguments
kf = KFold(n_splits=5, shuffle=True, random_state=1121218)
# Pass additional arguments inside another function
func = lambda trial: objective(trial, X, y, cv=kf, scoring="neg_mean_squared_log_error")

# Start optimizing with 100 trials
study.optimize(func, n_trials=100)

print(f"Base RMSLE     : {rmsle:.5f}")
print(f"Optimized RMSLE: {study.best_value:.5f}")

----------------------------------------------
Wall time: 52.8s

Base RMSLE     : 0.07573
Optimized RMSLE: 0.07177
```

8213.py hosted with ♡ by GitHub view raw

In just under a minute, we achieved a significant score boost (in terms of log errors, 0.004 is pretty sweet). We did this with only 100 trials. Let's boldly run another 200 and see what happens:

```
%%time

study.optimize(func, n_trials=200)

print("Best params:")

for key, value in study.best_params.items():
    print(f"\t{key}: {value}")

```

```
10
11  ------------------------------------------------------
12  Wall time: 1min 2s
13  Best params:
14          n_estimators: 4400
15          learning_rate: 0.04284240402963163
16          max_depth: 3
17          subsample: 0.7
18          max_features: log2
19
20
21  print(f"Base RMSLE     : {rmsle:.5f}")
22  print(f"Optimized RMSLE: {study.best_value:.5f}")
23
24  ------------------------------------------------------
25
26  Base RMSLE     : 0.07573
27  Optimized RMSLE: 0.07139
```

8214.py hosted with ♡ by GitHub view raw

The score *did* improve but marginally. It looks like we hit it close to the max in the first run!

Most importantly, we achieved this score in just over 2 minutes using a search space that would probably take hours with regular GridSearch.

I don't know about you, but I am sold!

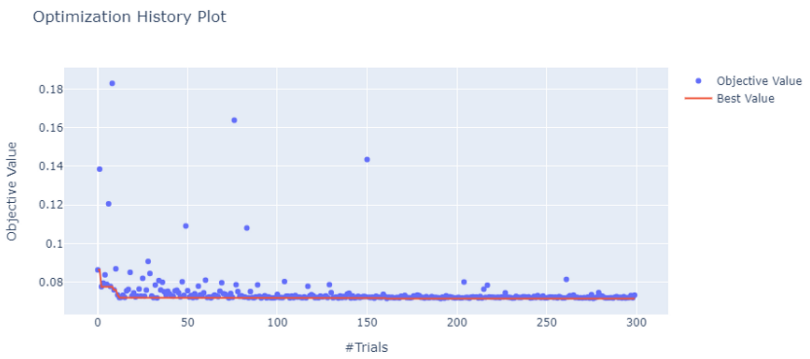
Using visuals for more insights and smarter tuning

Optuna offers a wide range of plots under its `visualization` subpackage. Here, we will discuss only 2, which I think are the most useful.

First, let's plot the optimization history of the last `study`:

```
1  from optuna.visualization.matplotlib import plot_optimization_history
2
3  plot_optimization_history(study);
```

8215.py hosted with ♡ by GitHub view raw



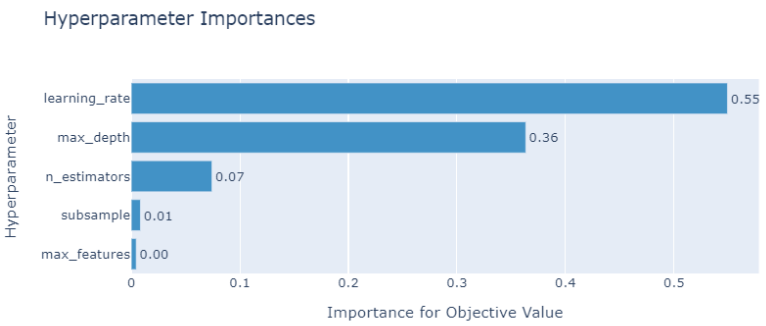
This plot tells us that Optuna made the score converge to the minimum after only a few trials.

Next, let's plot hyperparameter importances:

```python
from optuna.visualization.matplotlib import plot_param_importances

plot_param_importances(study);
```

8216.py hosted with ♡ by GitHub view raw



This plot is massively useful! It tells us several things, including:

- `max_depth` and `learning_rate` are the most important
- `subsample` and `max_features` are useless for minimizing the loss

A plot like this comes in handy when tuning models with many hyperparameters. For example, you could take a test run of 40–50 trials and plot the parameter importances.

Depending on the plot, you might decide to discard some less important parameters and give a larger search space for other ones, possibly reducing the search time and space.

You can check out this page of the documentation for more information on Optuna's supported plot types.

Summary

I think we can all agree that Optuna lived up to the *whole* hype I made in the introduction. It is awesome!

This article only gave you the basics you can do with Optuna. Actually, Optuna is capable of much more. Some of the critical topics we didn't cover today:

- Use cases of Optuna with other ML/DL frameworks
- Choosing a pruning algorithm to weed out unpromising trials immediately
- Parallelization

and the coolest of all:

- Using SQLite or other databases (local or remote) to run massive-scale optimization with resume/pause capabilities