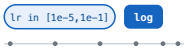
Optuna

Optuna searches for better hyperparameters from a plain Python objective, covered here from running a study to pruning and visualizing results.

The Objective & Search Space

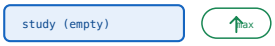
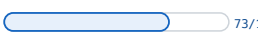
A function that asks the trial for params and returns one number.

PURPOSE	COMMAND	RESULT
A continuous parameter.	`x = trial.suggest_float("x", -10.0, 10.0)`	
A log-scale float (learning rates).	`lr = trial.suggest_float("lr", 1e-5, 1e-1, log=True)`	
A stepped float.	`dropout = trial.suggest_float( "dropout", 0.0, 0.5, step=0.1)`	
An integer parameter.	`n = trial.suggest_int("n_layers", 1, 5)`	
A categorical choice.	`opt = trial.suggest_categorical( "optimizer", ["adam", "sgd"])`	
Return the score to optimize.	`return (x - 2) ** 2`	

define-by-run: each trial.suggest_* call adds a parameter to the search space; the function returns one number to optimize

Create & run a study

A study runs the objective n_trials times and tracks the best.

PURPOSE	COMMAND	RESULT
Create a study (minimize).	`study = optuna.create_study(direction=…)`	
Maximize instead.	`create_study(direction="maximize")`	
Run the optimization loop.	`study.optimize(objective, n_trials=100)`	
Stop after a wall-clock budget.	`study.optimize(objective, timeout=600)`	
Add custom stopping logic.	`study.optimize(objective, n_trials=50, callbacks=[cb])`	
Watch progress in a notebook.	`study.optimize(objective, n_trials=100, show_progress_bar=True)`	

create_study(direction=…) sets the goal; study.optimize(objective, n_trials=…) runs and remembers every trial, capped by n_trials or timeout

Samplers & Pruners

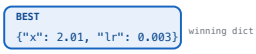
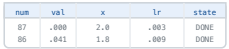
The sampler proposes; the pruner cuts losses early.

PURPOSE	COMMAND	RESULT
Default smart sampler (TPE).	`study = optuna.create_study(sampler=optuna.samplers.TPESampler(seed=42))`	
Reproducible random baseline.	`optuna.samplers.RandomSampler(seed=0)`	
Exhaustive grid.	`optuna.samplers.GridSampler({"x": [0, 1], "y": [-1, 0, 1]})`	
Report intermediate scores.	`trial.report(value, step)` `if trial.should_prune(): raise TrialPruned()`	
Median pruning (default).	`study = optuna.create_study(pruner=optuna.pruners.MedianPruner(n_warmup_steps=5))`	
Aggressive multi-fidelity pruning.	`optuna.create_study(pruner=HyperbandPruner())`	

TPESampler proposes smarter values each trial; report() + should_prune() let the pruner kill hopeless trials early to save compute

Read the Results

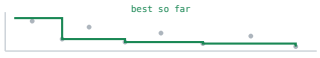
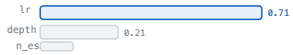
Pull out the winning params, value, and trial history.

PURPOSE	COMMAND	RESULT
Best hyperparameters.	`study.best_params`	
Best objective value.	`study.best_value`	
The full best trial.	`study.best_trial`	
Count finished trials by state.	`len(study.trials)`	
Export every trial to a DataFrame.	`df = study.trials_dataframe()`	
Re-rank or filter trials.	`study.best_trials (multi-objective)` `or study.get_trials()`	

best_params / best_value / best_trial read the winner; trials_dataframe() and len(study.trials) inspect the whole run

Visualize the Search

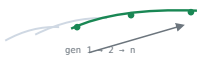
History, importances, interactions, all from the study.

PURPOSE	COMMAND	RESULT
Optimization history.	`optuna.visualization.plot_optimization_history(study)`	
Which params matter most.	`optuna.visualization.plot_param_importances(study)`	
All trials as colored lines.	`optuna.visualization.plot_parallel_coordinate(study)`	
One param vs score.	`optuna.visualization.plot_slice(study)`	
Two params interaction.	`optuna.visualization.plot_contour(study, params=["lr", "n_layers"])`	
Static Matplotlib variant.	`import optuna.visualization.matplotlib as vis_mpl` `vis_mpl.plot_optimization_history(study)`	

optuna.visualization returns interactive Plotly figures from a finished study; every plot has a Matplotlib twin under .matplotlib

Multi-objective

Optimize several goals at once; read the Pareto front.

PURPOSE	COMMAND	RESULT
Declare multiple directions.	`study = create_study(directions=["minimize", "maximize"])`	
Return a tuple of objectives.	`return loss, accuracy`	
Read the Pareto-optimal set.	`study.best_trials`	
Plot the Pareto front.	`vis.plot_pareto_front(study, target_names=["loss", "acc"])`	
Pick a trade-off by hand.	`min(study.best_trials, key=lambda t: t.values[0])`	
Many-objective sampler default.	`optuna.samplers.NSGAIISampler()`	

directions=[...] + a tuple return give no single winner; study.best_trials is the Pareto set, NSGAIISampler evolves it

Storage, resume & parallelism

Persist trials to a database; resume and scale across workers.

PURPOSE	COMMAND	RESULT
Save trials to a SQLite file.	`study = optuna.create_study(storage="sqlite:///study.db", study_name="tune")`	
Resume an existing study.	`optuna.create_study(storage=..., study_name="tune", load_if_exists=True)`	
Load a saved study.	`study = optuna.load_study(study_name="tune", storage="sqlite:///study.db")`	
Parallelize across threads / processes.	`study.optimize(objective, n_trials=100, n_jobs=4)`	
Seed a known-good config.	`study.enqueue_trial({"lr": 0.01, "n_layers": 3})`	
Warm-start from past results.	`study.add_trial(optuna.trial.create_trial(params=..., distributions=..., value=...))`	

A storage= URL persists every trial; load_if_exists / load_study resume it, n_jobs and shared storage scale out, enqueue/add_trial seed configs

Integrate (sklearn, xgboost, pytorch)

Wrap a real model in the objective; prune with framework callbacks.

PURPOSE	COMMAND	RESULT
sklearn objective via cross-val.	`score = cross_val_score(clf, X, y, cv=5, scoring="accuracy").mean()`	
Drop-in tuned estimator (sklearn).	`from optuna_integration import OptunaSearchCV(clf, params, n_trials=50)`	
Prune XGBoost rounds early.	`from optuna_integration.xgboost import XGBoostPruningCallback(trial, "validation-logloss")`	
Prune LightGBM rounds early.	`from optuna_integration.lightgbm import LightGBMPruningCallback(trial, "valid_0-auc")`	
PyTorch: report + prune per epoch.	`trial.report(val_loss, epoch) if trial.should_prune(): raise …`	
Install the integration extras.	`pip install "optuna-integration[xgboost]"`	

the objective is plain Python: return a cross_val_score mean or use OptunaSearchCV; prune with framework callbacks from the separate optuna-integration package