Hyperparameter Tuning Strategies

Course:
INFO-6154 Machine Learning Optimization Strategies

FANSHAWE

Developed by:
Mohammad Noorchenarboo

May 19, 2025

Current Section

Grid Search: Exhaustive Hyperparameter Evaluation

Motivating Question

Can we find the best model by systematically evaluating all possible hyperparameter combinations?

Grid Search is a brute-force approach where all combinations of hyperparameter values are tested. It's simple and reproducible but can be computationally expensive.

$$\text{Total Configs} = \prod_{i=1}^{n} |H_i|$$

where H_i is the set of values for the i-th hyperparameter.

Grid Search: Exhaustive Hyperparameter Evaluation

Example

If we have:

- learning_rate $\in \{0.01, 0.001\}$
- batch_size $\in \{32, 64, 128\}$

Then total configurations $= 2 \times 3 = 6$

Scalability Warning

The number of configurations grows exponentially with the number of hyperparameters. Grid search becomes infeasible in high dimensions.

Grid Search: Exhaustive Hyperparameter Evaluation

Simple Grid Search Using Scikit-Learn

```python
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

param_grid = {
    'max_depth': [3, 5, 10],
    'n_estimators': [50, 100]
}

grid = GridSearchCV(RandomForestClassifier(), param_grid, cv=3)
grid.fit(X_train, y_train)
# Expected: Trains 3 * 2 = 6 models with 3-fold validation
```

Grid Search Limitations

- Wastes computation on unimportant parameters
- Can't exploit patterns in parameter interactions
- Not adaptive – treats all dimensions equally

Summary: Grid Search

Aspect	**Details**
Name	Grid Search
Principle	Exhaustively tries all combinations of parameters
Strengths	Simple, reproducible, parallelizable
Weaknesses	Not scalable, inefficient for large spaces
Tools	Scikit-learn, KerasTuner (GridSearch), manual loops

Current Section

Random Search: Efficient Search in Large Spaces

Motivating Question

Is it better to randomly sample fewer configurations than to exhaustively try them all?

Random Search selects hyperparameter combinations by sampling randomly from specified distributions. It explores the space more efficiently than grid search in high dimensions.

$$\text{Sampled Configs} = k \ll \prod_{i=1}^{n} |H_i|$$

Why Random Search Works

In high-dimensional search spaces, only a few hyperparameters matter. Random search explores more values of each dimension individually than grid search does.

Random Search: Efficient Search in Large Spaces

Random Search with Scikit-Learn

```python
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from scipy.stats import randint

param_dist = {
    'max_depth': randint(3, 15),
    'n_estimators': randint(50, 200)
}

search = RandomizedSearchCV(RandomForestClassifier(), param_dist,
                            n_iter=10, cv=3, random_state=42)
search.fit(X_train, y_train)
# Expected: Samples 10 random combinations over 3-fold CV
```

Search Budget Caveat

You must define the number of iterations explicitly. Performance is limited by how many samples you can afford to train.

Random Search: Efficient Search in Large Spaces

Limitations of Random Search

- Still ignores past evaluations (no learning from results)
- May miss optimal regions if not enough samples
- Requires careful distribution design for each parameter

Summary: Random Search

Aspect	**Details**
Name	Random Search
Principle	Samples configurations from user-defined distributions
Strengths	Efficient, scalable, covers more individual values
Weaknesses	May miss good regions; needs well-designed priors
Tools	Scikit-learn, KerasTuner (RandomSearch), Ray Tune

Current Section

Validation Strategies: Estimating Generalization Performance

Motivating Question

How do we reliably estimate a model's performance on unseen data while tuning hyperparameters?

Validation strategies are essential for measuring model performance during tuning. Poor validation leads to overfitting on the validation set or misrepresenting model capability.

Hold-Out Validation

Hold-out validation splits the data once:

$$\text{Train} : \text{Validation} = 80\% : 20\% \text{ (typical)}$$

Use case: Fast checks on large datasets

Validation Strategies: Estimating Generalization Performance

K-Fold Cross-Validation

K-Fold CV splits data into k equal parts, training on $k-1$ and validating on the remainder:

$$\text{Error}_{CV} = \frac{1}{k}\sum_{i=1}^{k}\mathscr{L}_i$$

Use case: Low data regimes, stable performance estimate

Nested Cross-Validation

Nested CV wraps two cross-validation loops:

- **Inner loop:** Hyperparameter tuning
- **Outer loop:** Model evaluation

Use case: Prevents overfitting to validation folds during tuning

Validation Strategies: Estimating Generalization Performance

Example: K-Fold Cross-Validation

```python
from sklearn.model_selection import KFold
import numpy as np

kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_index, val_index in kf.split(X):
    X_train, X_val = X[train_index], X[val_index]
# Expected: 5 iterations with rotating validation fold
```

Which to Choose?

- Use **hold-out** for fast tuning on large data
- Use **K-fold** when data is limited or imbalanced
- Use **nested CV** for reliable benchmarking

Validation Strategies: Estimating Generalization Performance

Validation Strategy Pitfalls

- Data leakage between folds
- Choosing final model based on best validation score alone
- Not stratifying for class imbalance

Summary: Validation Strategies

Strategy	**Details**
Hold-Out	One-time split; fast and simple; may be unstable
K-Fold CV	Multiple folds; better estimate; slower
Nested CV	Two-level CV; robust to overfitting in tuning
Choosing Strategy	Depends on dataset size, imbalance, and tuning complexity
Tools	Scikit-learn: `KFold`, `StratifiedKFold`, `cross_val_score`

Current Section

Search Space Design: Making Tuning Efficient and Meaningful

Motivating Question

What happens if your search space is poorly designed? Can a good model be found if you don't search the right values?

A well-designed search space improves tuning performance by:

- Reducing irrelevant combinations
- Emphasizing sensitive hyperparameters
- Making sampling efficient (especially in random search and BO)

Search Space Design: Making Tuning Efficient and Meaningful

Log-Scale for Continuous Hyperparameters

Many hyperparameters (like `learning_rate`, `l2_reg`) affect training multiplicatively, not linearly:

$$\text{Search range: } 10^{-5} \to 10^{-1}$$

Log-scale sampling:

$$x = 10^{u}, \quad u \sim \text{Uniform}[-5, -1]$$

Categorical and Ordinal Parameters

- Categorical: `optimizer` $\in$ {Adam, SGD, RMSProp}
- Ordinal: `num_layers` $\in \{2, 3, 4, 5\}$

Sampling must preserve type semantics – don't interpolate between categories.

Search Space Design: Making Tuning Efficient and Meaningful

Common Search Space Mistakes

- Using linear scale for exponential-sensitive parameters
- Ignoring type (e.g., treating categorical as continuous)
- Too wide or too narrow range for key parameters

Consequences of Poor Space Design

- Misses optimal regions completely
- Wastes budget on uninformative configurations
- Slows convergence in model-based search

Summary: Search Space Design

Aspect	**Details**
Log-Scale Parameters	Use for learning rate, L2, dropout rate; sample from exponent space
Categorical Parameters	Discrete choices like optimizers; no interpolation
Ordinal Parameters	Integer-valued parameters with order (e.g., depth, units)
Pitfalls	Linear scale for non-linear effects, unbalanced ranges, type misuse
Tools	KerasTuner, Optuna, Ray Tune – support typed search spaces

Current Section

What is Optuna?

Motivating Question

How can we move beyond manual tuning, grid search, or random search to find optimal models efficiently?

Optuna is a software framework for automatic hyperparameter optimization. It is:

- **Lightweight:** Minimal setup, pure Python
- **Powerful:** Supports intelligent search with built-in samplers and pruners
- **Flexible:** Search space is defined dynamically with Python logic

What is Optuna?

$$\min_{x \in \mathscr{X}} f(x) \quad \text{or} \quad \max_{x \in \mathscr{X}} f(x)$$

What Does Optuna Help With?

- Tuning hyperparameters in machine learning models
- Avoiding wasteful evaluations by pruning bad trials early
- Integrating with libraries like `scikit-learn`, `PyTorch`, `XGBoost`, `Keras`, etc.

Example: Search learning rate in log scale with `trial.suggest_loguniform("lr", 1e-5, 1e-1)`.

Problems with Manual Tuning

- Time-consuming and inconsistent
- No systematic way to reuse or share results
- No adaptive exploration of good regions in the search space

A Modern Hyperparameter Optimization Framework

Motivating Question

Can we tune hyperparameters dynamically and adaptively, instead of trying every possible value manually?

Optuna is designed to let you define hyperparameter logic dynamically while optimizing a target metric.

$$x^* = \arg\min_{x \in \mathcal{X}} f(x)$$

Core Concepts in Optuna

- **Study:** A single optimization run
- **Trial:** One evaluation of the objective function
- **Sampler:** Suggests the next parameters to try
- **Pruner:** Stops unpromising trials early

A Modern Hyperparameter Optimization Framework

Define-by-Run Philosophy

Optuna builds the search space as code is executed. This allows conditional logic like:

- `if optimizer == "Adam": trial.suggest_float(...)`
- Varying layer widths or architectures based on other values

A Modern Hyperparameter Optimization Framework

Basic Optuna Workflow

```python
import optuna

def objective(trial):
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)
    dropout = trial.suggest_float("dropout", 0.1, 0.5)
    # Dummy training logic
    accuracy = train_model(lr, dropout)
    return accuracy

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=50)
# Expected: 50 intelligently selected trials
```

Beginner Mistakes to Avoid

- Omitting return statement from the objective
- Using inconsistent parameter types
- Not monitoring intermediate results for early pruning

Summary: Optuna Overview

Aspect	**Details**
Name	Optuna
Purpose	Automatic hyperparameter optimization
Optimization Approach	Define-by-run, black-box optimization
Core Components	Study, Trial, Sampler, Pruner
Typical Usage	Optimize learning rate, dropout, number of layers, etc.
Integration	Scikit-learn, PyTorch, TensorFlow, XGBoost

Installing and Importing Optuna

Motivating Question

What are the minimal steps to start using Optuna in your Python environment?

Before running optimization studies, we must install the Optuna package and confirm it works with our Python setup.

Install via: `pip install optuna`

Installation Options

- Basic installation: `pip install optuna`
- With visualization support: `pip install optuna[visualization]`
- Full feature set (e.g., databases): `pip install optuna[all]`

Installing and Importing Optuna

Installing and Importing Optuna

```
# Installation (run in terminal or notebook cell)
# pip install optuna

import optuna

print(optuna.__version__)
# Expected Output: '3.x.x'
```

Note for Jupyter Users

Some plotting functions (like `optuna.visualization.plot_optimization_history`) require extra dependencies. Use `pip install optuna[visualization]` to enable plotting support.

Installing and Importing Optuna

Common Issues

- Visualization errors due to missing Plotly or Jupyter dependencies
- Version mismatches (e.g., between Optuna and ML frameworks)
- Forgetting to import Optuna before using its API

Summary: Installing and Importing Optuna

Aspect	Details
Installation	`pip install optuna`
Visualization Support	`pip install optuna[visualization]`
Import Syntax	`import optuna`
Optional Extras	`[all]` for DB and dashboarding, `[visualization]` for plotting
Common Errors	Missing dependencies, incompatible versions, skipped imports

Running an Optuna Study

Motivating Question

What is the minimal working structure to define a search space, evaluate a model, and run hyperparameter optimization with Optuna?

To perform optimization, Optuna requires two components:

- An **objective function** that trains and evaluates a model
- A **study** that coordinates the trials

$$x^* = \arg\max_{x \in \mathscr{X}} f(x) \quad \text{or} \quad \arg\min_{x \in \mathscr{X}} f(x)$$

Key Concepts

- **Trial:** One candidate set of hyperparameters
- **Objective Function:** Defines training logic and returns a scalar metric
- **Study:** Manages the optimization loop over many trials

Running an Optuna Study

Define Objective and Run a Study

```
import optuna

def objective(trial):
    # Suggest hyperparameters
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)
    dropout = trial.suggest_float("dropout", 0.1, 0.5)

    # Build and compile your model here
    model = build_model(dropout=dropout)
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])

    # Train and evaluate
    model.fit(x_train, y_train, validation_split=0.2, epochs=10,
        verbose=0)
    val_loss, val_accuracy = model.evaluate(x_val, y_val, verbose=0)

    return val_accuracy  # Metric to maximize

# Create and run the study
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)
```

Running an Optuna Study

Best Practices

- Use `trial.suggest_*()` for all parameter values
- Return only the validation score used for optimization
- Keep the objective function clean and local to each trial

Choosing Optimization Direction

Use `"maximize"` for accuracy or F1 score, `"minimize"` for loss, error, or latency.

Common Mistakes

- Returning a non-numeric or multi-output object
- Mixing random sampling with `trial`-suggested values
- Forgetting to specify `direction` in `create_study`

Summary: Running an Optuna Study

Aspect	**Details**
Objective Function	Defines training logic; returns a scalar validation metric
Study Object	Created with `optuna.create_study(...)`
Execution Loop	Run with `study.optimize(...)`
Output	Best parameters and best achieved score
Common Errors	No return value, direction not set, external randomness

Sampling Hyperparameters in Optuna

Motivating Question

How do we define and control which hyperparameters Optuna tries during optimization?

Optuna provides a rich API for sampling different types of hyperparameters:

- Continuous (linear, log scale)
- Discrete (integers, steps)
- Categorical (labels, names, options)

Sampling Hyperparameters in Optuna

$$\text{Sampled Value} \sim \text{trial.suggest\_type(...)} \quad \text{each trial}$$

Common Sampling Methods

- `suggest_float(name, low, high)`
- `suggest_int(name, low, high)`
- `suggest_loguniform(name, low, high)`
- `suggest_categorical(name, choices)`

Sampling Hyperparameters in Optuna

Example: Different Parameter Types

```
def objective(trial):
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-1)  # log-scale
        float
    batch_size = trial.suggest_int("batch_size", 32, 128, step=32)
         # discrete
    optimizer = trial.suggest_categorical("optimizer", ["adam",
        "sgd", "rmsprop"])

    # Use these values in a real model definition
    ...
```

Best Practice

Use log-scale for parameters like learning rate, regularization, dropout:

```
suggest_loguniform("lr", 1e-5, 1e-1)
```

This allows exploring a wide numeric range more effectively.

Sampling Hyperparameters in Optuna

Conditional Sampling

Use Python control flow to make one parameter depend on another:

```
if optimizer == "sgd":
    momentum = trial.suggest_float("momentum", 0.0, 0.9)
```

This is only possible because Optuna uses define-by-run.

Sampling Mistakes to Avoid

- Using wrong scale (e.g., linear instead of log for LR)
- Forgetting to use `trial` methods (sampling outside trial)
- Re-suggesting the same parameter name inside one trial

Summary: Sampling Hyperparameters

Sampling Type	**Use Case and Syntax**
Float (linear)	`suggest_float("x", 0.1, 1.0)`
Float (log-scale)	`suggest_loguniform("lr", 1e-5, 1e-1)`
Int (discrete steps)	`suggest_int("units", 32, 128, step=32)`
Categorical	`suggest_categorical("opt", ["adam", "sgd"])`
Conditional Params	Use `if/else` logic with trial values

Accessing and Interpreting Optimization Results in Optuna

Motivating Question

How do we retrieve the best configuration and understand what happened during optimization?

After completing a study, Optuna stores all trial data inside the study object.
You can access:

- The best hyperparameter set
- The best objective value
- All completed trials and their results

Accessing and Interpreting Optimization Results in Optuna

$$x^* = \arg\max_x f(x) \quad \text{and} \quad f(x^*) = \texttt{study.best\_value}$$

Key Result Attributes

- `study.best_trial` – Best trial object
- `study.best_value` – Best score found
- `study.best_params` – Best hyperparameter combination
- `study.trials` – All trials and metadata

Accessing and Interpreting Optimization Results in Optuna

Accessing Best Results

```python
# Best value (objective score)
print("Best value:", study.best_value)

# Best hyperparameter configuration
print("Best params:", study.best_params)

# Full trial object
best = study.best_trial
print("Trial number:", best.number)
print("Accuracy:", best.value)
print("Params:", best.params)
```

Accessing and Interpreting Optimization Results in Optuna

Use Case: Logging and Reuse

Store best parameters for retraining the model:

```
best_model = build_model(**study.best_params)
```

You can also serialize the study using `study.trials_dataframe()`.

Optional: Save to File

Optuna supports persistent storage (e.g., SQLite or RDB):

```
study = optuna.create_study(study_name="exp1",
    storage="sqlite:///optuna.db")
```

This allows resuming later and tracking multiple experiments.

Accessing and Interpreting Optimization Results in Optuna

Mistakes to Avoid

- Ignoring the full trial history (insightful for debugging)
- Only reporting the best value without parameter context
- Forgetting to save results for later use

Summary: Accessing and Interpreting Results

Attribute	**Description**
`study.best_value`	Best score obtained
`study.best_params`	Dictionary of best hyperparameters
`study.best_trial`	Full trial object (value, params, number)
`study.trials`	History of all trials and outcomes
Storage Option	Save studies using SQLite or external DBs

Best Practices for Using Optuna Effectively

Motivating Question

What should you keep in mind to use Optuna safely and effectively in real projects?

Optuna makes tuning easier, but like any optimization system, its value depends on how well it's used. These best practices will help you build reproducible and efficient experiments.

Checklist for Effective Use

- Always return a single numeric score from your objective
- Use only `trial.suggest_*()` to define hyperparameters
- Log results (score, loss, params) to your experiment tracker
- Name your studies and store them in a persistent database when needed
- Use cross-validation inside the objective function when needed

Best Practices for Using Optuna Effectively

Minimal Production Pattern

```
study = optuna.create_study(
    direction="maximize", study_name="exp01",
    storage="sqlite:///optuna.db", load_if_exists=True
)
study.optimize(objective, n_trials=50)
```

Next Steps: Topics Coming Soon

- **TPE Sampler**: Smarter exploration than random or grid
- **Pruners**: Stop bad trials early to save time
- **Visualization**: Track progress and search behavior
- **Multi-objective Optimization**: Balance accuracy vs. latency, etc.
- **Integration**: With TensorFlow, PyTorch, and external workflows

Best Practices for Using Optuna Effectively

Reproducibility Tips

- Set random seeds inside your model and training code
- Fix data splits (train/val/test) before optimization
- Avoid hidden sources of randomness (e.g., dropout during evaluation)

What Not to Do

- Donât hardcode values inside your model – always use trial suggestions
- Donât use test data during tuning
- Donât forget to save best parameters if you need to retrain

Summary: Best Practices and What's Next

Practice	**Description**
Use trial methods	Always use `trial.suggest_*()` to define params
Return single scalar	Objective must return only one numeric value
Track experiments	Log best score, params, and all trial metadata
Prepare for pruning	Add checkpoints or intermediate reporting (covered next week)
Avoid test leakage	Never use test set during optimization