Lecture 12: Hyperparameter Tuning Explained with Intuition & Examples

Dr. Ratnesh Srivastava, CSIT, Guru Ghasidas Viswavidyalaya, Bilaspur, C.G.

July 28, 2025

Abstract

These lecture notes provide a comprehensive explanation of hyperparameter tuning, a critical step in machine learning model development. We cover fundamental concepts, and delve into various tuning strategies including Grid Search, Random Search, Bayesian Optimization (with Optuna), Hyperband, and distributed tuning with Ray Tune. Each method is explained with core intuition, mathematical insights, practical examples, and a discussion of its pros and cons. The notes emphasize practical application and provide a student exercise to reinforce learning.

Contents

1 Hyperparameter Tuning Fundamentals

In machine learning, models have two types of parameters:

- **Model Parameters:** These are learned directly from the data during the training process (e.g., weights in a neural network, split points in a decision tree).
- **Hyperparameters:** These are settings that are chosen *before* the training process begins and are not learned from the data. They control the learning process itself (e.g., learning rate, number of trees in an ensemble, maximum depth of a tree, regularization strength).

Hyperparameter tuning is the process of finding the optimal set of hyperparameters for a given model and dataset to achieve the best possible performance.

1.1 Core Analogy

Imagine tuning a racing car for different tracks:

- **Engine (Model):** This is the core machine learning model (e.g., SVM, Random Forest, Neural Network) with its fixed underlying design.
- **Tires/Gears (Hyperparameters):** These are the adjustable settings (e.g., tire pressure, gear ratios, suspension stiffness). You choose these *before* the race begins, and they significantly impact how the car performs on a specific track.
- **Lap Time (Metric):** This is the performance measure you want to optimize (e.g., validation accuracy, F1-score, AUC).

The goal of tuning is to find the perfect combination of tires and gears to get the fastest lap time on a particular track.

1.2 Key Concepts

- **Hyperparameters:** As defined above, these are the external configuration variables for the model. Examples include:
  - `learning_rate`: How much each new tree contributes in boosting.
  - `max_depth`: The maximum depth of a decision tree.
  - `C` (regularization parameter in SVM): Controls the trade-off between achieving a low training error and a low testing error. Fairness of the model to the data (e.g., if we want to build a model that avoids unfair outcomes for specific groups of people like race, gender, religion, sexual orientation, disability status etc.).
  - `n_estimators`: The number of trees in an ensemble model.
- **Objective Function:** This is the metric that you want to optimize (either maximize or minimize) during the tuning process. It should reflect the model's performance on unseen data. Common objective functions include:

    - Validation Accuracy (maximize)
    - Mean Squared Error (minimize)
    - F1-score (maximize)
    - Area Under the ROC Curve (AUC) (maximize)

- **Search Space:** This defines the range or set of possible values that each hyperparameter can take. For example:

- 'learning$_rate$' : $[0.01, 0.05, 0.1, 0.2](discretevalues)or[0.001, 0.5](continuousrange).`max_depth` : [3, 5, 7, 9](integervalues)$.

—

2 Grid Search

2.1 Intuition

Grid Search is the most straightforward hyperparameter tuning technique. It systematically tries every single combination of hyperparameter values specified in a predefined grid. Think of it like testing every single floor and every single room in a building to find a hidden treasure. You are guaranteed to find the treasure if it exists within the rooms you check, but it can take a very long time if the building is large.

2.2 Mathematical Intuition and Details

While Grid Search itself doesn't involve complex mathematical formulas for its operation, its core principle is an **exhaustive search** within a discretized hyperparameter space.

Let's define our hyperparameter search space. Suppose we have K hyperparameters, denoted as $H_1, H_2, \ldots, H_K$. For each hyperparameter H_k, we define a finite set of candidate values, $S_k = \{v_{k,1}, v_{k,2}, \ldots, v_{k,n_k}\}$, where n_k is the number of discrete values chosen for H_k.

The total number of combinations to evaluate, N_{total}, is the product of the number of values for each hyperparameter:

$$N_{total} = n_1 \times n_2 \times \ldots \times n_K = \prod_{k=1}^{K} n_k$$

For each of these N_{total} combinations, say a specific combination $\mathbf{h} = (h_1, h_2, \ldots, h_K)$ where $h_k \in S_k$, we train a model $M(\mathbf{h})$ and evaluate its performance using an objective function $J(\mathbf{h})$. This evaluation typically involves C folds of cross-validation, meaning the model is trained and evaluated C times, and the average score is taken.

The goal of Grid Search is to find the set of hyperparameters $\mathbf{h}^*$ that maximizes (or minimizes) the objective function:

$$\mathbf{h}^* = \arg\max_{\mathbf{h} \in \prod S_k} J(\mathbf{h})$$

Here, $\prod S_k$ represents the Cartesian product of all sets of hyperparameter values, which forms the grid.

2.2.1 Mathematical Example for Python Code

In the provided Python code for ‘GridSearchCV‘:

```
– param_grid = {
      'C': [0.1, 1, 10],           # Regularization parameter for SVM
      'gamma': [0.01, 0.1, 1],      # Kernel coefficient for 'rbf', 'poly', 'sigmoid'
      'kernel': ['linear', 'rbf']  # Type of kernel function
  }
```

We have:

- H_1 = 'C': $S_1 = \{0.1, 1, 10\}$, so $n_1 = 3$.
- H_2 = 'gamma': $S_2 = \{0.01, 0.1, 1\}$, so $n_2 = 3$.
- H_3 = 'kernel': $S_3 = \{\text{'linear'}, \text{'rbf'}\}$, so $n_3 = 2$.

Using the formula for N_{total}:

$$N_{total} = n_1 \times n_2 \times n_3 = 3 \times 3 \times 2 = 18$$

The ‘GridSearchCV‘ will train and evaluate exactly 18 distinct hyperparameter combinations. If ‘cv=5‘ (5-fold cross-validation), then for each of these 18 combinations, the model will be trained 5 times on different data folds. This means a total of $18 \times 5 = 90$ model training/evaluation runs will occur. The result (‘grid$_search.best_params‘)willbeoneofthese18combinations, fore$
$‘'C' : 1,'gamma' : 0.1,'kernel' :' rbf'‘$.

2.3 Example

Let’s tune a Support Vector Machine (SVM) classifier for a synthetic dataset.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.datasets import make_classification
import numpy as np

# 1. Generate a synthetic dataset for demonstration
X, y = make_classification(n_samples=200, n_features=10, n_informative=5, random_state=4
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42

# 2. Define the search space for hyperparameters
param_grid = {
    'C': [0.1, 1, 10],           # Regularization parameter for SVM
    'gamma': [0.01, 0.1, 1],      # Kernel coefficient for 'rbf', 'poly', 'sigmoid'
    'kernel': ['linear', 'rbf']  # Type of kernel function
}
```

```
# 3. Perform grid search
# SVC(): The model to tune
# param_grid: The dictionary defining the search space
# cv=5: Use 5-fold cross-validation for evaluation of each combination
grid_search = GridSearchCV(SVC(random_state=42), param_grid, cv=5, verbose=1)
grid_search.fit(X_train, y_train)

# 4. Print the best parameters and corresponding score
print(f"Best parameters found by Grid Search: {grid_search.best_params_}")
print(f"Best cross-validation score: {grid_search.best_score_:.4f}")
print(f"Test set score with best parameters: {grid_search.score(X_test, y_test):.4f}")
```

2.4 Process

1. Define a dictionary ‘param$_g$*rid‘wherekeysarehyperparameternamesandvaluesarelistsofdiscreteval*
   3valuesfor‘C‘ × 3 values for ‘gamma‘ × 2 values for ‘kernel‘ = 18 unique combinations.

3. For each of these 18 combinations:

   - The model is trained using 5-fold cross-validation (‘cv=5‘). This means the training data is split into 5 parts; the model is trained on 4 parts and validated on 1 part, repeated 5 times. The average validation score is recorded.

4. After evaluating all combinations, ‘GridSearchCV‘ selects the combination that yielded the best average cross-validation score.

2.5 Visualization (Conceptual)

Imagine a 2D grid for ‘C‘ and ‘gamma‘ with ‘kernel=’rbf’‘ fixed:

```
        gamma=0.01   gamma=0.1    gamma=1
       +------------+------------+------------+
C=0.1  |   Acc=0.82 |   Acc=0.85 |   Acc=0.81 |
       +------------+------------+------------+
C=1    |   Acc=0.88 |   Acc=0.92 |   Acc=0.83 |  <- Best in this kernel
       +------------+------------+------------+
C=10   |   Acc=0.87 |   Acc=0.90 |   Acc=0.80 |
       +------------+------------+------------+
        (Similar grid for Kernel: 'linear')
```

Grid Search exhaustively checks every single cell in this grid.

2.6 Pros/Cons

✓ **Pros:**

  - **Exhaustive:** Guarantees finding the best combination within the defined grid.
  - **Simple:** Easy to understand and implement.

× **Cons:**

  - **Computationally Expensive:** The number of combinations grows exponentially with the number of hyperparameters and the number of values for each. This makes it impractical for many hyperparameters or large search spaces.
  - **Inefficient:** Spends equal time on unpromising regions of the search space.

—

3 Random Search

3.1 Intuition

Random Search samples a fixed number of hyperparameter combinations from specified probability distributions (or discrete choices) within the search space. Instead of checking every point on a grid, it randomly selects points. Think of it like throwing darts blindfolded at a target. While you might miss some spots, you're likely to hit good areas if you throw enough darts, and it's generally more efficient than meticulously checking every square inch of the dartboard.

3.2 Mathematical Intuition and Details

Unlike Grid Search, Random Search samples N_{trials} combinations randomly from the search space. For continuous hyperparameters, this means sampling from specified probability distributions. For discrete hyperparameters, it means uniformly sampling from the given list of options.

Let the hyperparameter space be $\mathcal{H}$. Random Search selects N_{trials} independent and identically distributed (i.i.d.) samples $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{N_{trials}}$ from $\mathcal{H}$. For each sampled combination $\mathbf{h}_i$, the model $M(\mathbf{h}_i)$ is trained and its performance $J(\mathbf{h}_i)$ is evaluated. The best combination $\mathbf{h}^*$ is then chosen as:

$$\mathbf{h}^* = \underset{i=1,\ldots,N_{trials}}{\arg\max} \; J(\mathbf{h}_i)$$

A key insight for Random Search's efficiency, especially in high-dimensional spaces, comes from the work by Bergstra and Bengio (2012). They demonstrated that for a search space with K hyperparameters, if only k^* of these hyperparameters are truly important (i.e., significantly affect the model's performance), then Random Search is more efficient at finding good values for these k^* important parameters than Grid Search.

Consider a simple case: if 2 out of 10 hyperparameters are important, Random Search explores each of these 2 important dimensions with N_{trials} independent samples. Grid Search, however, would still be bound by the total product of the number of choices across all 10 dimensions, making it astronomically expensive, even if 8 of those dimensions were relatively unimportant.

For a continuous parameter like ‘C‘ or ‘gamma‘, using a **log-uniform distribution** (loguniform(a, b)) is often crucial. This means that instead of sampling X uniformly between a and b, we sample $\log(X)$ uniformly between $\log(a)$ and $\log(b)$. This is because many hyperparameters, particularly regularization strengths or learning rates, have a multiplicative effect on model performance. A change from 0.001 to 0.002 might be as significant as a change from 0.1 to 0.2. Sampling on a log scale ensures equal exploration of these multiplicative ”steps.”

3.2.1 Mathematical Example for Python Code

In the provided Python code for ‘RandomizedSearchCV‘:

```
param_dist = {
    'C': loguniform(1e-2, 100),
    'gamma': loguniform(1e-3, 10),
    'kernel': ['linear', 'rbf']
}
random_search = RandomizedSearchCV(..., n_iter=10, ...)
```

Here, ‘n$_iter = 10‘meansthatthealgorithmwillperformexactly10evaluations, drawingrandomsamplesfro$

Let’s illustrate the sampling:

- For ‘C‘: ‘loguniform(1e-2, 100)‘ means that values like 0.01, 0.1, 1, 10, 100 are equally likely to be sampled across their logarithmic scale. For example, the probability of sampling ‘C‘ between 0.01 and 0.1 is the same as sampling between 1 and 10.
- For ‘gamma‘: Similarly, ‘loguniform(1e-3, 10)‘ samples values logarithmically between 0.001 and 10.
- For ‘kernel‘: ‘['linear', 'rbf']‘ means that each time a sample is drawn, there’s a 50% chance it will be 'linear' and a 50% chance it will be 'rbf'.

Each of the 10 samples will be a unique combination of (C, gamma, kernel). For instance, a sample could be:

- Trial 1: ‘C=0.08‘, ‘gamma=0.005‘, ‘kernel='rbf'‘
- Trial 2: ‘C=2.5‘, ‘gamma=0.8‘, ‘kernel='linear'‘
- ...
- Trial 10: ‘C=50‘, ‘gamma=0.03‘, ‘kernel='rbf'‘

The algorithm then selects the combination among these 10 that yielded the best average cross-validation score. This demonstrates how it might hit promising regions without exhaustively checking every point on a grid, especially when the search space for ‘C‘ and ‘gamma‘ is continuous.

3.3 Example

Let’s use the same SVM tuning problem, but with Random Search.

```
from sklearn.svm import SVC
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.datasets import make_classification
from scipy.stats import loguniform # For defining continuous distributions
import numpy as np

# 1. Generate a synthetic dataset (same as Grid Search example)
X, y = make_classification(n_samples=200, n_features=10, n_informative=5, random_state=4
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42

# 2. Define the search space using distributions
param_dist = {
    'C': loguniform(1e-2, 100),     # Log-uniform distribution for C (from 0.01 to 100)
    'gamma': loguniform(1e-3, 10),  # Log-uniform distribution for gamma (from 0.001 to
    'kernel': ['linear', 'rbf']     # Discrete choices for kernel
}

# 3. Perform random search
# n_iter=10: Number of random combinations to sample
random_search = RandomizedSearchCV(
    SVC(random_state=42), param_dist, n_iter=10, cv=5, random_state=42, verbose=1)
random_search.fit(X_train, y_train)

# 4. Print the best parameters and corresponding score
print(f"Best parameters found by Random Search: {random_search.best_params_}")
print(f"Best cross-validation score: {random_search.best_score_:.4f}")
print(f"Test set score with best parameters: {random_search.score(X_test, y_test):.4f}")
```

3.4 Process

1. Define a ‘param$_{d}ist‘dictionarywherevaluescanbediscretelists(like‘kernel‘)orprobabilitydistribution$
2. For each of the 10 sampled combinations, the model is trained and evaluated using 5-fold cross-validation.
3. The combination with the best average cross-validation score is selected.

Example samples (conceptual):

- (`C`=0.05, `gamma`=0.2, `kernel`=’rbf’) $\rightarrow$ Acc: 0.91
- (`C`=12, `gamma`=0.003, `kernel`=’linear’) $\rightarrow$ Acc: 0.87
- ... (8 more random samples)

3.5 Visualization (Conceptual)

Imagine a continuous search space for ‘C‘ and ‘gamma‘:

```
Search space: C (0.01-100), gamma (0.001-10)
Random samples (dots):
   . .   .     <- Higher concentration
   .     .        in promising regions
.     .
```

Random Search’s advantage is that it doesn’t waste time on a rigid grid. If a hyperparameter has little impact, random sampling won’t spend much time exploring many values for it. If a hyperparameter is very important, random sampling is more likely to hit a good value than a coarse grid.

3.6 Pros/Cons

✓ **Pros:**

- **More Efficient for High Dimensions:** Often finds a better combination than Grid Search in the same amount of time, especially when the number of hyperparameters is large or when their optimal values lie in continuous ranges.
- **Flexible Search Space:** Can sample from continuous distributions, which is often more appropriate for many hyperparameters.

× **Cons:**

- **Might Miss Optimal Points:** Since it’s random, there’s no guarantee of finding the absolute best combination, even within the defined distributions.
- **No Learning:** Doesn’t learn from past evaluations to guide future searches.

—

4 Bayesian Optimization

4.1 Intuition

Bayesian Optimization is a more advanced and efficient technique for hyperparameter tuning. Instead of blindly searching, it builds a probabilistic model (a ”surrogate model”) of the objective function’s performance based on past evaluations. This model helps it predict which hyperparameter combinations are likely to yield good results, guiding the search more intelligently. Think of it like a detective using clues (past evaluations) to build a profile of the suspect (the optimal hyperparameters) and then focusing their search efforts only on the most promising areas.

4.2 Key Components

- **Surrogate Model (Probabilistic Model):** This model approximates the true, expensive objective function. It's typically a Gaussian Process or a Tree-structured Parzen Estimator (TPE). It provides two key pieces of information for any given hyperparameter combination:
  1. An estimate of the objective function's value (e.g., predicted accuracy).
  2. A measure of uncertainty around that estimate.
- **Acquisition Function:** This function uses the predictions and uncertainties from the surrogate model to decide which hyperparameter combination to evaluate next. It balances two strategies:
  1. **Exploration:** Sampling in regions where the uncertainty is high, to discover new potentially good areas.
  2. **Exploitation:** Sampling in regions where the surrogate model predicts high performance, to refine known good areas.

  A common acquisition function is Expected Improvement (EI).

4.3 Mathematical Intuition and Details

Bayesian Optimization aims to find the global optimum of an expensive, black-box objective function $f(\mathbf{x})$ (our model's performance) over a search space $\mathcal{X}$ (our hyperparameter space). It does this by iteratively performing two steps:

1. **Build/Update Surrogate Model:** Given a history of evaluated points $D_t = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_t, y_t)\}$, where $y_i = f(\mathbf{x}_i)$, we fit a probabilistic model (e.g., Gaussian Process) $P(y|\mathbf{x}, D_t)$ to this data. This model provides both a mean prediction $\mu(\mathbf{x})$ and a variance $\sigma^2(\mathbf{x})$ for the objective function at any point $\mathbf{x}$.

2. **Optimize Acquisition Function:** An acquisition function $\alpha(\mathbf{x})$ is used to determine the next point $\mathbf{x}_{t+1}$ to evaluate. This function leverages $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ to balance exploration (sampling where uncertainty is high) and exploitation (sampling where the expected value is high). The next point is chosen as:

$$\mathbf{x}_{t+1} = \arg\max_{\mathbf{x} \in \mathcal{X}} \alpha(\mathbf{x})$$

Expected Improvement (EI): A very common acquisition function is Expected Improvement (EI). It quantifies the expected gain over the current best observed objective value $f(\mathbf{x}^*)$, where $\mathbf{x}^*$ is the point that yielded the best performance so far. For maximization problems, the formula for EI is:

$$\text{EI}(\mathbf{x}) = E[\max(0, f(\mathbf{x}) - f(\mathbf{x}^*))]$$

In this formula:

- EI($\mathbf{x}$): The Expected Improvement from evaluating the hyperparameter configuration $\mathbf{x}$. This value represents how much better, on average, we expect this configuration to be than the current best.
- $E[\cdot]$: Denotes the expectation operator, meaning we're calculating the average value of the term inside, weighted by its probability.
- $f(\mathbf{x})$: Represents the true (but unknown) objective function value for hyperparameter configuration $\mathbf{x}$.
- $f(\mathbf{x}^*)$: Represents the current best observed objective value from all previous trials.

The $\max(0, \ldots)$ ensures that we only consider improvements (gains). If a candidate $\mathbf{x}$ is predicted to be worse than $\mathbf{x}^*$, its improvement is 0. When the surrogate model is a Gaussian Process, EI has a closed-form solution involving the cumulative distribution function (CDF) Φ and probability density function (PDF) ϕ of the standard normal distribution:

$$\text{EI}(\mathbf{x}) = (\mu(\mathbf{x}) - f(\mathbf{x}^*))\Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^*)}{\sigma(\mathbf{x})}\right) + \sigma(\mathbf{x})\phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^*)}{\sigma(\mathbf{x})}\right)$$

Here, $\mu(\mathbf{x})$ is the mean prediction of the surrogate model for $\mathbf{x}$, and $\sigma(\mathbf{x})$ is the standard deviation (uncertainty) of that prediction. The term involving $\mu(\mathbf{x}) - f(\mathbf{x}^*)$ drives exploitation (favoring points with high predicted mean). The term involving $\sigma(\mathbf{x})$ drives exploration (favoring points with high uncertainty).

Tree-structured Parzen Estimator (TPE): Optuna's default sampler, TPE, is a specific type of Bayesian Optimization that models $P(\mathbf{x}|y)$ (the probability of hyperparameters given a target performance y) using two non-parametric densities: $l(\mathbf{x})$ for points with good performance ($y < y^*$) and $g(\mathbf{x})$ for points with bad performance ($y \geq y^*$). The next point is chosen by maximizing the ratio $l(\mathbf{x})/g(\mathbf{x})$. This effectively means it searches for points that are likely to be good, given what has been observed so far.

4.3.1 Mathematical Example for Python Code (Conceptual)

In the Optuna Python code, ‘n$_trials = 20‘impliesthatOptunawillperform20evaluations.Let'sillustrateho$

Assume we're optimizing ‘accuracy‘ and our current best observed accuracy $f(\mathbf{x}^*)$ is 0.90. The surrogate model for ‘SVC(C, gamma, kernel)‘ predicts performance for new (untested) ‘(C, gamma, kernel)‘ combinations.

Scenario 1: Pure Random Search The 20 trials would be sampled completely independently. There's no learning.

Scenario 2: Bayesian Optimization (Conceptual Walkthrough)

1. **Trial 1-5 (Initial Exploration):** Optuna might start with a few random samples to build an initial understanding of the search space.
   - Trial 1: ‘(C=0.5, gamma=0.01, kernel='rbf')‘, Acc = 0.85
   - Trial 2: ‘(C=10, gamma=0.5, kernel='linear')‘, Acc = 0.82
   - Trial 3: ‘(C=0.02, gamma=0.005, kernel='rbf')‘, Acc = 0.79

- Trial 4: ‘(C=1.5, gamma=0.1, kernel=’rbf’)‘, Acc = 0.90 ($f(\mathbf{x}^*)$ is now 0.90)
- Trial 5: ‘(C=5, gamma=0.08, kernel=’linear’)‘, Acc = 0.88

2. **After 5 trials, surrogate model is built/updated.** For any new $\mathbf{x}$, it predicts $\mu(\mathbf{x})$ and $\sigma(\mathbf{x})$. $f(\mathbf{x}^*)$ is 0.90.

3. **Trial 6 (Guided by EI):** Consider two hypothetical points ‘X_A‘ and ‘X_B‘:

   - ‘X_A‘ (‘C=0.8, gamma=0.15, kernel=’rbf’‘):
   - Surrogate prediction: $\mu($ Calculating EI for ‘X_A‘ would show a good expected improvement due to its high predicted value.
   - ‘X_B‘ (‘C=20, gamma=0.001, kernel=’rbf’‘):

- Surrogate prediction: $\mu($ Calculating EI for ‘X$_B$‘*mightstillyieldareasonablevaluebecausethehighunce* Optuna’s acquisition function (TPE in this case) would favor the point with the highest EI. If ‘X$_A$‘*hasthehighestEI*, *it′sevaluatednext.If*‘X_B‘*hasasurprisinglyhighEIduetouncertainty*, *itn*

4. **Subsequent Trials (6-20):** The process repeats. After each trial, the surrogate model is updated, and the acquisition function becomes more refined, leading the search towards areas that are both promising (high predicted accuracy) and/or less explored (high uncertainty). This iterative learning allows Bayesian Optimization to often converge to better solutions faster than random search.

4.4 Example with Optuna

Optuna is a popular open-source framework for hyperparameter optimization that implements Bayesian Optimization and other advanced techniques.

```
import optuna
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.datasets import make_classification
import numpy as np

# 1. Generate a synthetic dataset
X, y = make_classification(n_samples=200, n_features=10, n_informative=5, random_state=4
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42

# 2. Define the objective function for Optuna
# This function will be called repeatedly by Optuna.
# It takes a 'trial' object, suggests parameters, trains the model, and returns the scor
def objective(trial):
    # Suggest hyperparameters using trial.suggest_xxx() methods
    # Optuna will sample values from these defined ranges/choices
    C = trial.suggest_float('C', 1e-2, 100, log=True) # Log-uniform float from 0.01 to 1
    gamma = trial.suggest_float('gamma', 1e-3, 10, log=True) # Log-uniform float from 0.
    kernel = trial.suggest_categorical(’kernel’, [’linear’, ’rbf’]) # Categorical choice

    # Build and evaluate the model with the suggested hyperparameters
    model = SVC(C=C, gamma=gamma, kernel=kernel, random_state=42)

    # Use 5-fold cross-validation on the training data to get a robust score
    score = cross_val_score(model, X_train, y_train, cv=5).mean()

    # Optuna will try to maximize this returned score
    return score

# 3. Create an Optuna study and optimize
# direction=’maximize’ tells Optuna to find parameters that maximize the objective funct
study = optuna.create_study(direction=’maximize’, sampler=optuna.samplers.TPESampler(see

# Run the optimization for a specified number of trials
# Notice that ’n_trials’ can be much smaller than Grid Search/Random Search for comparab
print("Starting Optuna optimization...")
study.optimize(objective, n_trials=20) # Only 20 evaluations!

# 4. Print the best results
print(f"\nBest accuracy found by Optuna: {study.best_value:.4f}")
print(f"Best hyperparameters found: {study.best_params}")

# You can also get the best model by re-training with the best parameters
best_params = study.best_params
final_model = SVC(**best_params, random_state=42)
final_model.fit(X_train, y_train)
print(f"Test set score with best Optuna parameters: {final_model.score(X_test, y_test):.
```

4.5 Process

1. **Define ‘objective(trial)‘:** This function encapsulates the entire model training and evaluation process for a single trial. It takes an ‘optuna.Trial‘ object as input.

2. **Suggest Hyperparameters:** Inside ‘objective‘, ‘trial.suggest$_float()‘,‘trial.suggest_int()‘,‘trial.sugg$ $validation).Theperformancemetric(e.g.,accuracy)isreturnedbythe‘objective‘function.$

3. **‘study.optimize()‘:** Optuna iteratively calls the ‘objective‘ function. After each trial, it updates its surrogate model with the new performance data. The acquisition function then uses this updated model to propose the next set of hyperparameters to evaluate, balancing exploration and exploitation.

4.6 Visualization (Conceptual)

Imagine the search space as a landscape where higher points are better accuracy.

```
Iteration 5 (Initial random samples):
● ● ● ● ●
●
●
      (Surrogate model starts forming a rough idea)

Iteration 10 (Guided exploration/exploitation):
● ● ● ● ●
●       ●
●
      (Surrogate model focuses evaluations near known good areas ''
       and explores uncertain regions)

Iteration 20 (Converging to optimum):
● ● ●
● ● ●
●
      (Evaluations are concentrated around the best performing regions,
       as the surrogate model becomes more confident)
```

4.7 Pros/Cons

✓ **Pros:**

- **Smart Sampling:** Significantly more efficient than Grid or Random Search, often finding better hyperparameters with fewer evaluations.
- **Flexible Search Space:** Handles continuous, discrete, and conditional hyperparameters naturally (e.g., if 'kernel='poly'', then 'degree' is relevant; otherwise, it's not).
- **Pruning:** Can stop unpromising trials early (see Hyperband).

× **Cons:**

- **Overhead:** Involves building and updating a surrogate model, which adds some computational overhead, though usually negligible compared to model training.
- **Sequential Nature:** The core Bayesian optimization process is inherently sequential (each trial depends on previous ones), limiting parallelization for a single study (though frameworks like Optuna support parallel trials across multiple workers).

—

5 Hyperband

5.1 Intuition

Hyperband is an optimization strategy that focuses on efficiently allocating resources (e.g., epochs for a neural network, number of trees for a boosting model) to many randomly sampled hyperparameter configurations. It's designed to quickly identify and eliminate poorly performing configurations early in their training, saving computational resources. Think of it like a multi-stage tournament bracket: many contenders (hyperparameter configurations) start, but only the best performers advance to subsequent rounds with more resources (longer training times), while weaker contenders are eliminated early.

5.2 Mathematical Intuition and Details

Hyperband is an algorithm that efficiently allocates a fixed computational budget across a set of hyperparameter configurations. It is based on the **Successive Halving Algorithm (SHA)**.

SHA works by taking a set of N randomly sampled configurations and training them for a small amount of resources r. It then evaluates their performance, keeps only the top $1/\eta$ (where η is the elimination factor) configurations, and trains these survivors for ηr resources. This process repeats until only one configuration remains, which has been trained for the maximum allowed resource R_{max}.

Hyperband extends SHA by running multiple SHAs (called "brackets") with different initial numbers of configurations and initial resource allocations. This ensures a comprehensive search over the budget.

Let's define the key parameters:

- R_{max}: The maximum amount of resource any single configuration can receive (e.g., maximum epochs).
- η (eta): The elimination factor. At each stage, only $1/\eta$ (e.g., 1/3) of the configurations are promoted to the next stage. A common value is $\eta = 3$.

A Hyperband "bracket" consists of multiple stages, $s \in \{0, 1, \ldots, s_{max}\}$, where $s_{max} = \lfloor \log_\eta(R_{max}) \rfloor$. For each bracket s:

- The number of configurations to evaluate is $n = \lfloor (s_{max} + 1)\eta^s/(s+1) \rfloor$ (a more precise formula is sometimes used, but the core idea is a decreasing number of configurations as s increases).
- The initial resource allocated to each configuration is $r = R_{max}\eta^{-s}$.

Within each bracket, the Successive Halving process unfolds:

1. Start with n configurations, each trained for r resources.
2. Rank these configurations by their performance.
3. Keep the top n/η configurations and train them for an additional ηr resources.

4. Repeat step 3, doubling the resources and halving the configurations, until the configurations reach R_{max} resources or are eliminated.

By running these brackets with varying initial n and r combinations, Hyperband efficiently explores the search space and focuses resources on the most promising candidates. The total budget for Hyperband is fixed, and it aims to find the best configuration given this budget.

5.2.1 Mathematical Example for Python Code (Conceptual)

The provided Python code for Ray Tune uses 'HyperBandScheduler' with '$\text{max}_t = 100'$ and reduction_factor=3 ($whichis\ \eta$). Let's trace one possible bracket that Hyperband might run, for example, a bracket starting with $s = 0$. For '$\text{max}_t = 100`and`eta = 3`$,$\text{s}_{max} = \lfloor \log_3(100) \rfloor = \lfloor 4.19 \rfloor = 4$.

Example Bracket ($s = 0$):

- Initial configurations $n_0 = (s_{max} + 1) = (4 + 1) = 5$ configurations (using simplified n for $s = 0$).
- Initial resource per config $r_0 = R_{max}\eta^{-s_{max}} = 100 \times 3^{-4} = 100/81 \approx 1.23$ epochs. For practical purposes, this might be rounded up or down, let's say 2 epochs for simplicity.

Tournament Structure for this Bracket:

1. **Stage 1 (min resources):**
   - Start with $n_0 = 5$ randomly sampled hyperparameter configurations.
   - Train each for $r_0 = 2$ epochs.
   - Evaluate their performance (e.g., accuracy).
   - Keep the top $5/\eta = 5/3 = \lfloor 1.66 \rfloor = 1$ best performing configuration.
2. **Stage 2 (increased resources):**
   - Take the 1 surviving configuration.
   - Train it for an additional $\eta r_0 = 3 \times 2 = 6$ epochs (total epochs now $2 + 6 = 8$).
   - Evaluate performance.
   - Keep the top $1/\eta = 1/3 = \lfloor 0.33 \rfloor = 0$ (meaning this bracket would terminate after this stage if this was the last possible promotion).

This is just one of several brackets Hyperband would run. Other brackets would start with different (n, r) pairs to cover the search space more thoroughly. For example, a bracket starting with $s = s_{max} = 4$:

- Initial configs $n_4 = (s_{max} + 1)\eta^{s_{max}} = (4 + 1)3^4 = 5 \times 81 = 405$ configurations.
- Initial resource $r_4 = R_{max}\eta^{-s_{max}} = 100 \times 3^{-4} = 100/81 \approx 1.23$ epochs.

This bracket would train many configurations for a very short time and rapidly eliminate most, preserving only a few to reach the higher resource levels. This efficient allocation is the core mathematical strength of Hyperband.

5.3 Example: Tune CNN Epochs (Conceptual)

Let's say we want to tune a Convolutional Neural Network (CNN) and the maximum resource we are willing to give any single configuration is $R_{max} = 81$ epochs. Let $\eta = 3$.

Consider one bracket, for $s = s_{max} = 4$ (since $\lfloor \log_3(81) \rfloor = 4$).

- Initial configurations $n = 1 \times \eta^{s_{max}} = 1 \times 3^4 = 81$ (simplified initial configs for clarity, usually derived from a formula ensuring overall budget distribution).
- Initial resource per config $r = R_{max}/\eta^{s_{max}} = 81/3^4 = 81/81 = 1$ epoch.

Tournament Structure for this Bracket:

```
Bracket with s_max = 4 (e.g., starts with 81 configs, each gets 1 epoch)

Stage 1:
  - Total configs: 81
  - Resource per config: 1 epoch
  - Keep top 81 / 3 = 27 configurations based on performance after 1 epoch.

Stage 2:
  - Total configs: 27 (survivors from Stage 1)
  - Resources from previous stage: 1 epoch
  - Additional resources: 3 epochs (total 1+3=4 epochs)
  - Keep top 27 / 3 = 9 configurations based on performance after 4 epochs.

Stage 3:
  - Total configs: 9 (survivors from Stage 2)
  - Resources from previous stage: 4 epochs
  - Additional resources: 9 epochs (total 4+9=13 epochs)
  - Keep top 9 / 3 = 3 configurations based on performance after 13 epochs.

Stage 4:
  - Total configs: 3 (survivors from Stage 3)
  - Resources from previous stage: 13 epochs
  - Additional resources: 27 epochs (total 13+27=40 epochs)
  - Keep top 3 / 3 = 1 configuration based on performance after 40 epochs.

Stage 5 (Final):
  - Total configs: 1 (survivor from Stage 4)
  - Resources from previous stage: 40 epochs
  - Additional resources: 41 epochs (total 40+41=81 epochs, reaching R_max)
  - This single best configuration is evaluated.
```

Hyperband systematically explores different initial resource allocations and number of configurations, ensuring a comprehensive search while being efficient.

5.4 Why it Works

- **Wastes Minimal Time:** It quickly identifies and discards configurations that are unlikely to perform well, preventing them from consuming significant training time.
- **Allocates More Resources to Promising Candidates:** Configurations that show early promise are given more resources to fully demonstrate their potential.
- **No Need for Surrogate Model:** Unlike Bayesian Optimization, Hyperband does not build a complex probabilistic model, making it simpler to implement and often faster for very high-dimensional search spaces.
- **Parallelizable:** The trials within a stage of a bracket can be run in parallel.

5.5 Pros/Cons

✓ **Pros:**

- **Resource Efficiency:** Excellent at saving computational resources by early termination of poor trials.
- **Scalable:** Works well for models with many hyperparameters and for large resource budgets.
- **Parallelizable:** Naturally supports parallel execution of trials.

× **Cons:**

- **Requires Resource Unit:** Needs a clear concept of an incremental resource unit (e.g., epoch, number of trees).
- **Less Intelligent Sampling (than Bayesian):** It's still based on random sampling at the start of each bracket, not learning from past performance to guide the next sample in the same way Bayesian methods do. However, it can be combined with Bayesian methods (e.g., Hyperband + TPE).

—

6 Ray Tune

6.1 Intuition

Ray Tune is a powerful, open-source library built on top of Ray (a distributed computing framework) for scalable hyperparameter tuning. It provides a unified API to run various tuning algorithms (Grid Search, Random Search, Bayesian Optimization, Hyperband, ASHA, Population-Based Training, etc.) across multiple CPUs, GPUs, or even clusters. Think of it like a factory assembly line specifically designed for running machine learning experiments. It can efficiently manage hundreds or thousands of concurrent experiments, distributing them across available hardware resources.

6.2 Key Features

- **Distributed Computing:** Leverages the Ray framework to easily scale tuning across multiple machines or cores.
- **Unified API:** Provides a consistent interface for implementing various hyperparameter optimization algorithms and schedulers.
- **Fault Tolerance:** Can recover from worker failures.
- **Integration:** Integrates with popular ML libraries (PyTorch, TensorFlow, scikit-learn) and logging frameworks (TensorBoard).
- **Advanced Schedulers:** Supports state-of-the-art scheduling algorithms like Hyperband and Asynchronous Successive Halving Algorithm (ASHA) for efficient resource allocation.
- **Search Algorithms:** Can use various search algorithms, including those from Optuna, HyperOpt, etc.

6.3 Mathematical Intuition and Details

Ray Tune itself is not a specific optimization algorithm, but rather a **framework for orchestrating and distributing** various hyperparameter optimization algorithms and schedulers. Its mathematical contributions lie in its ability to efficiently execute and manage these diverse underlying algorithms across distributed computing resources.

Distributed Execution Model: Ray Tune's core power comes from the Ray framework, which allows tasks (e.g., training a single model trial) to be executed as 'Ray Actors' or 'Ray Tasks'.

- **Ray Actors:** Long-lived processes that can maintain state (e.g., a deep learning model). A tuning trial can be run as an actor.
- **Ray Tasks:** Stateless functions that can be executed remotely.

The 'tune.run()' function acts as a central coordinator. It interacts with a 'Searcher' (which implements the hyperparameter search algorithm like Random, Grid, TPE, ASHA, etc.) and a 'Scheduler' (which implements resource allocation and early stopping strategies like Hyperband).

When 'tune.run()' starts:

1. The 'Searcher' proposes a new set of hyperparameters for a trial.
2. Ray Tune allocates resources (e.g., 1 CPU, 1 GPU) for this trial and launches a worker process.
3. The 'trainable' function (your custom training logic) runs on the worker.
4. Periodically, the 'trainable' function reports metrics (e.g., validation accuracy after each epoch) back to the 'Scheduler' using 'tune.report()'.

5. The ‘Scheduler‘ monitors these reported metrics across all active trials. Based on its logic (e.g., Hyperband’s successive halving criteria), it can decide to:
    - **Continue** a trial (if it’s performing well).
    - **Pause** a trial (to resume later, common in ASHA).
    - **Stop** a trial (if it’s clearly unpromising, an early termination).
6. The ‘Searcher‘ uses the completed trial results to inform its next hyperparameter suggestions (if it’s a model-based searcher like OptunaSearch).

This decoupled architecture allows different search algorithms and schedulers to be combined flexibly, and their execution to be transparently parallelized across a cluster. The efficiency gain isn’t from a new mathematical optimization technique in Tune itself, but from its robust engineering and distributed capabilities that enable these techniques to run at scale.

6.3.1 Mathematical Example for Python Code (Conceptual)

The ‘tune.run‘ call in the Python example orchestrates a total of ‘num$_{s}amples = 50‘trials, using‘OptunaSe$

While there isn’t a single ”mathematical example” for Ray Tune’s own execution, it’s about the combination of the mathematical principles of the underlying algorithms:

- **OptunaSearch (TPE):** Mathematically, it’s performing Bayesian optimization. For each of the 50 trials, OptunaSearch will intelligently select the next hyperparameter configuration based on the results of previous trials, using its $l(\mathbf{x})/g(\mathbf{x})$ ratio.

- **HyperBandScheduler:** Mathematically, it’s implementing the successive halving protocol. For each trial, as ‘train$_{s}vm‘(conceptually)progressesthroughepochsandreportsmetrics, Hyperbandwi$
So, the ‘num$_{s}amples = 50‘representsthemaximumnumberofconfigurationsthatOptunaSearchwillpropo$
$100‘resource.Forinstance:$

    OptunaSearch proposes ‘Trial 1: C=1.0, gamma=0.05, kernel=’rbf’‘. Hyperband starts it with a small resource budget (e.g., 2 epochs). If its accuracy is poor after 2 epochs, Hyperband might terminate it.

    OptunaSearch then proposes ‘Trial 2: C=0.2, gamma=0.001, kernel=’linear’‘. This trial might perform well initially, so Hyperband allows it to run for more epochs, perhaps reaching 10, 30, or even 100 epochs, depending on its bracket and continued performance.

    This dynamic interaction ensures that the computational budget (number of evaluations, and total resources) is spent most effectively, guided by both an intelligent search strategy and an efficient early-stopping mechanism.

6.4 Example with Hyperband + Optuna (Conceptual)

This example shows how Ray Tune can orchestrate a complex tuning strategy, combining a search algorithm (like Optuna’s TPE) with a scheduling algorithm (like Hyperband).

```
from ray import tune
from ray.tune.schedulers import HyperBandScheduler
from ray.tune.search.optuna import OptunaSearch
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.datasets import make_classification
import pandas as pd
import numpy as np

# 1. Define a trainable function (your model training logic)
# This function will be executed by Ray Tune workers.
# It must take a 'config' dictionary (hyperparameters) and 'checkpoint_dir' (for sa
# and report metrics via tune.report().
def train_svm(config):
    # Generate a synthetic dataset for this example (in a real scenario, load your
    X, y = make_classification(n_samples=200, n_features=10, n_informative=5, rando
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random

    # Extract hyperparameters from the config dictionary
    C = config["C"]
    gamma = config["gamma"]
    kernel = config["kernel"]

    model = SVC(C=C, gamma=gamma, kernel=kernel, random_state=42)

    # Simulate iterative training (e.g., for epochs or boosting rounds)
    # This is crucial for schedulers like Hyperband to prune early.
    # For SVM, we'll just do a single cross_val_score, but in a real DL scenario,
    # you'd loop through epochs and call tune.report() after each.
    score = cross_val_score(model, X_train, y_train, cv=5).mean()

    # Report the metric back to Ray Tune
    tune.report(accuracy=score)

# 2. Define the search space using Ray Tune's API
config_space = {
    "C": tune.loguniform(1e-2, 100),
    "gamma": tune.loguniform(1e-3, 10),
    "kernel": tune.choice(['linear', 'rbf'])
}

# 3. Configure the search algorithm (e.g., OptunaSearch for Bayesian Optimization)
# This tells Tune *how* to sample hyperparameters.
optuna_search = OptunaSearch(
    metric="accuracy", # Metric to optimize
    mode="max"         # Direction of optimization (maximize accuracy)
)

# 4. Configure the scheduler (e.g., HyperBandScheduler for early stopping)
# This tells Tune *when* to stop or pause trials.
hyperband_scheduler = HyperBandScheduler(
    metric="accuracy",
    mode="max",
    max_t=100, # Max "resource" (e.g., epochs) any trial can get
    reduction_factor=3 # Eta for Hyperband
)

# 5. Run the optimization
print("Starting Ray Tune optimization...")
analysis = tune.run(
    train_svm,                  # Your trainable function
    config=config_space,        # The search space
    num_samples=50,             # Total number of trials to run (can be more than n
    search_alg=optuna_search,   # The search algorithm to use
    scheduler=hyperband_scheduler, # The scheduler for early stopping
    resources_per_trial={"cpu": 2}, # Resources to allocate per trial (e.g., 2 CPU
    name="svm_tuning_example",  # Name for the experiment
    verbose=1                   # Verbosity level
)

# 6. Get the best result
print(f"\nBest trial config: {analysis.best_config}")
print(f"Best trial accuracy: {analysis.best_result['accuracy']:.4f}")
```

6.5 Workflow

1. Define a ‘trainable‘ function (e.g., ‘train$_{s}vm‘intheexample)thattakesa‘config‘(hyperparameter$

2. Choose a ‘search$_{a}lg‘(e.g.,‘OptunaSearch‘,‘HyperOptSearch‘)tospecifyhowhyperparametersaresampled.$

3. Call ‘tune.run()‘ to start the optimization. Ray Tune will:

   - Distribute trials across available worker nodes (CPUs/GPUs).
   - The chosen ‘search$_{a}lg‘proposesnewhyperparameterconfigurations.The‘scheduler‘monit$

- Results from all trials are aggregated centrally.

6.6 Pros/Cons

✓ **Pros:**

  - **Scalability:** Designed for distributed computing, enabling tuning across large clusters.
  - **Flexibility:** Supports a wide range of search algorithms and schedulers.
  - **Robustness:** Handles failures gracefully and provides good logging/visualization tools.
  - **Unified API:** Simplifies the orchestration of complex tuning experiments.

× **Cons:**

  - **Setup Complexity:** Can have a steeper learning curve than simpler tools like 'GridSearchCV' or basic Optuna, especially when setting up distributed environments.
  - **Overhead:** For very small-scale problems, the overhead of the distributed framework might not be justified.

—

7 Q&A with Practical Scenarios

- **Q1: When would I choose Grid Search over Random Search?**
- **A1:** You would typically choose Grid Search in very specific scenarios:
  - When you have a very small number of hyperparameters (e.g., 1 to 3).
  - When the search space for each hyperparameter is very small and discrete (e.g., 2-4 values per parameter).
  - When you absolutely need to check every single combination within a confined, well-understood grid.

  Example: Tuning 'max$_depth`(e.g.,[3,5,7])and`min_samples_split`(e.g.,[2,10,20])foradecisiont$
  $\times 3 = 9$ combinations.

- **Q2: Why does Bayesian optimization often outperform random search in terms of finding better hyperparameters with fewer evaluations?**
- **A2:** Bayesian methods are more efficient because they:
  - **Learn from Past Evaluations:** They build a probabilistic model (surrogate model) of the objective function based on all previous trials.
  - **Focus on Promising Regions:** Using an acquisition function, they intelligently propose the next set of hyperparameters to evaluate, prioritizing regions that are predicted to have high performance or high uncertainty (which could reveal new optimal areas). Random search, in contrast, samples blindly without using any information from previous trials.

Example: After 10 trials, a Bayesian optimizer might stop exploring areas where the surrogate model predicts low accuracy, whereas Random Search would continue to sample there.

- **Q3: How does Hyperband save computational resources during tuning?**
- **A3:** Hyperband saves resources by:
  - **Early Termination of Poor Configurations:** It trains many configurations for a short period (small resources). Only the best-performing ones are allowed to continue training with more resources, while the weak ones are discarded early.
  - **Adaptive Resource Allocation:** It ensures that more computational budget is allocated to the most promising hyperparameter combinations.

  Example: A hyperparameter configuration that achieves only 0.6 accuracy after 10 training epochs (a small resource budget) won't be given the chance to train for 100 epochs, saving significant time.
- **Q4: When should I consider using a distributed framework like Ray Tune for hyperparameter tuning?**
- **A4:** You should use Ray Tune when you need:
  - **Large-Scale Tuning:** When you need to run hundreds or thousands of trials, potentially for complex models like deep neural networks.
  - **Distributed Computing:** When you have access to multiple CPUs, GPUs, or a cluster of machines and want to parallelize your tuning process.
  - **Advanced Techniques Orchestration:** When you want to combine sophisticated search algorithms (e.g., Bayesian optimization) with efficient schedulers (e.g., Hyperband) in a scalable manner.

  Example: Tuning large Transformer models for natural language processing on a cluster with 50+ GPUs.
- **Q5: What's one of the biggest mistakes people make when defining hyperparameter search spaces?**
- **A5:** A common mistake is not defining appropriate scales for hyperparameters, especially for parameters that operate multiplicatively.
  - **Bad Example:** Defining learning_rate = [0.0001, 0.001, 0.01] on a linear scale. The difference between 0.0001 and 0.001 is the same as between 0.001 and 0.01 in this linear grid, but in terms of model behavior, the impact is vastly different (multiplicative).

  - **Good Example:** Using a logarithmic scale for learning_rate, such as learning_rate = loguniform(1e-4, 1e-1). This ensures that the search space is explored uniformly across orders of magnitude, which is more natural for such parameters. Most advanced tuning frameworks (like Optuna, Ray Tune) support this directly.

—

8 Summary Table

Method	Best For	Example Use Case
Grid Search	Small, discrete spaces (≤10 combos)	Tuning 2-3 parameters in SVM
Random Search	High-dimensional spaces	Neural net with 5+ parameters
Bayesian Optimization	Expensive evaluations (e.g., DL)	Transformer fine-tuning
Hyperband	Resource-intensive models	Large CNN/RNN training
Ray Tune	Distributed large-scale tuning	Enterprise GPU cluster experiment

9 Exercise for Students

Task: Tune a Random Forest classifier using Optuna. **Dataset:** ‘sklearn.datasets.load$_d igits()‘(amul$ $classclassificationdatasetofhandwrittendigits)$.

Parameters to Tune:

- ‘n$_e stimators‘ : Thenumberoftreesintheforest(suggestintegervaluesbetween50and500).‘max_d$ $Themaximumdepthofeachtree(suggestintegervaluesbetween3and15)$.

- ‘min$_s amples_s plit‘ : Theminimumnumberofsamplesrequiredtosplitaninternalnode(suggestfloatvaluesbe$

Goal: Achieve a cross-validation accuracy of > 0.95 within < 30 trials.

Hint: Remember to use ‘trial.suggest$_f loat(..., log = True)‘forparametersthatbenefitfromalogarithmicsc$

Starter Code:

```
import optuna
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import cross_val_score, train_test_split
import numpy as np

# Load the dataset
X, y = load_digits(return_X_y=True)
# Split into training and validation (for cross-validation later)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state
def objective(trial):
    # Suggest hyperparameters for Random Forest
    n_estimators = trial.suggest_int('n_estimators', 50, 500)
    max_depth = trial.suggest_int('max_depth', 3, 15)
    # min_samples_split can be an integer or a float (as a fraction)
    # If it's a count, use suggest_int. If it's a fraction, use suggest_float.
    # The problem asks for float between 2 and 10, implying it's a count, but
    # it also says "log scaling for min_samples_split" which is more common for fra
    # Let's assume it's a count that benefits from log scale for exploration.
    min_samples_split = trial.suggest_float('min_samples_split', 2, 10, log=True)

    # Ensure min_samples_split is an integer if the model expects an int count
    # RandomForestClassifier accepts int for min_samples_split
    min_samples_split_int = int(min_samples_split)
    if min_samples_split_int < 2: # Ensure it's at least 2 as per RF docs
        min_samples_split_int = 2

    # Build Random Forest Classifier model
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_split=min_samples_split_int,
        random_state=42,
        n_jobs=-1 # Use all available CPU cores for faster training
    )

    # Evaluate model using 5-fold cross-validation on the training data
    score = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy').mean

    return score

# Create a study and optimize
# You need to complete this part:
# study = optuna.create_study(...)
# study.optimize(...)
# print best results
```