A GUIDE TO HYPERPARAMETER TUNING LIBRARIES: OPTUNA, KERAS TUNER, & SCIKIT-LEARN

Author – Austin Fairbanks

8-1-2024

Contents

Introduction to Automated Hyperparameter Tuning

A hyperparameter associated with a machine learning (ML) model is a specific type of parameter that is fixed in value before the training process begins. The ML model uses the hyperparameters to learn the optimum values of model parameters for the given problem during the training. Hyperparameter tuning is the process of optimizing different types of hyperparameters that influence the training of an ML model. Systematic hyperparameter tuning is an important skill when developing ML models because it leads to efficient design and selecting model size and complexity that are appropriate for the complexity for the problem. Moreover, better hyperparameters can reduce the training loss as compared to brute forcing a model design. With all the benefits of finding optimum hyperparameters, the question is not if this should be done, it is how this must be done.

Primarily, there are two types of ways to tune hyperparameters:

- Through manual sampling
- By using an automated framework that applies an algorithm on a predefined search space

Prior to reading this manual, it should be assessed whether the use case indeed needs automated hyperparameter tuning. **With many small models on simple datasets, defining an entire search space and applying an automatic algorithm may be an overkill as achieving satisfactory results via manual tuning takes little time for simple datasets.** The motivation behind developing automated frameworks was to optimize the training process within a fixed amount of resources; if the entirety of that amount is not already being allocated, then automated tuning will do little to improve the accuracy beyond what manual tuning can already accomplish. If the use case necessitates automatic hyperparameter optimization, determining the framework to use depends on many factors. One of the goals of this report is to assess different frameworks on these factors.

Prior to reading this manual, it should be assessed whether the use case indeed needs automated hyperparameter tuning. **With many small models on simple datasets, defining an entire search space and applying an automatic algorithm may be an overkill as achieving satisfactory results via manual tuning takes little time for simple datasets.** The motivation behind developing automated frameworks was to optimize the training process within a fixed amount of resources; if the entirety of that amount is not already being allocated, then automated tuning will do little to improve the accuracy beyond what manual tuning can already accomplish. If the use case necessitates automatic hyperparameter optimization, determining the framework to use depends on many factors. One of the goals of this report is to assess different frameworks on these factors.

In this survey, we have primarily considered 3 frameworks for review: Scikit-Learn, Keras Tuner, and Optuna. The rest of the manual is structured as follows:

1. Section 2 provides a brief overview of the main libraries and frameworks that were used in this work
2. In section 3, an example of a model design process without any tuning is shown
3. Considering that as base case, section 4 outlines the steps for efficient model design using tuning using the 3 frameworks
4. Section 4 discusses some advanced usage scenarios for the frameworks, including details of the helper tools developed during this review
5. The discussion of the advantages and disadvantages of frameworks as well as their use cases are in section 5 (*Overall Evaluation*) of this manual

Introduction to Frameworks & Libraries used in this Review

In this section, we provide a overview of the main Python libraries that have been used in this work:

Scikit-Learn

Python library installation: 'pip install scikit-learn', 'pip install scikeras'

Scikit-Learn (or sklearn) is a popular Python library, primarily used for developing conventional ML models and performing statistical analyses of data. It offers a wide range of capabilities such as unsupervised learning, supervised learning, data engineering, etc. For this review, we mainly use the sklearn.model_selection module as it contains functions necessary for hyperparameter tuning.

a. **Sci-Keras** is an extension of the sklearn library that provides wrapper functions to make keras models compatible with scikit learn. By using this extension, models defined using the keras library can be tuned using tools offered by scikit-learn.

Keras / Tensorflow

Python library installation: 'pip install tensorflow'

Tensorflow is one of the two most popular frameworks for developing deep learning models (the other being PyTorch). Keras [1] is a library that provides front-end functions to design and train models using Tensorflow. Keras is designed to be modular, fast, and easy to use. For example, Keras enables creating custom models with specific input and output shapes, layers, loss and optimizer functions, and attributes of the training process.

Callback Functions used during model training

Keras allows adding callbacks in the fit method when training a tuner. Callbacks improve efficiency by stopping the process when data classification shows no improvement.

callbacks:example

```
callback = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
model.fit(train_data, train_labels, validation_split=0.2, callbacks=[callback])
```

In this instance, patience signifies that at least five epochs will be executed before stopping.

Optuna

Python library installation: 'pip install optuna', 'pip install optuna-integration'

Optuna is a popular pythonic hyperparameter tuning library that works nearly identically for any ML framework (Keras, PyTorch, etc.). It offers a highly customizable tuning procedure with the option to choose from different sampling algorithms. Optuna also provides pruning algorithms that will stop a trial prematurely if its improvement from epoch to epoch is limited. However, it should be noted that the last update to this library was made in 2019, so developer support and updates may not be prompt.

a. **Optuna_integration**

Optuna_integration is an Optuna modules that provides assistance when using pruning with machine-learning architecture. For this manual, only the TFKerasPruningCallback that allows Optuna to prune a trial at any trial end. This is elaborated on in the ***Miscellaneous Information*** section.

Keras Tuner

Python library installation: 'pip install keras-tuner'

Keras-tuner is an extension of the Keras library that provides hyperparameter tuning capabilities for neural networks constructed using Keras, without any requirement for wrappers. The tuning process can be customized to choose from multiple tuning algorithms, with this report focusing specifically on the hyperband algorithm.

Overview of the Problem

To evaluate the effectiveness of different hyperparameter tuning libraries, we focus on an image classification problem, employing a dataset sourced from a materials processing lab. This dataset comprises four distinct classes, and the primary objective is to develop a machine learning model capable of accurately classifying images into one of these classes.

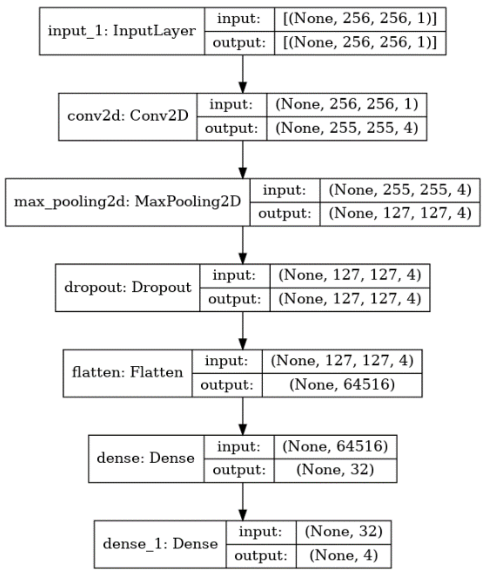
As a real-world dataset, it presents numerous challenges, featuring inherent variations in background objects, foreground subjects, and imaging conditions. These complexities add a layer of difficulty to the classification task, making it far from trivial.

Furthermore, it is important to note that the dataset is proprietary, and we are unable to share it along with this review. Nonetheless, the insights gained from this analysis are invaluable for understanding the nuances of machine learning model design and hyperparameter tuning in practical, real-world scenarios.

Base case – Designing a model without tuning

Model definition using Keras

To assess the impact of tuning, a baseline model without tuned hyperparameters is created using Keras. This serves as an 'informed guess' to initiate the model design process. It is important that input data be in NumPy arrays since Keras works best with this format. The base model's architecture is detailed below:



input_1: InputLayer
input: [(None, 256, 256, 1)]
output: [(None, 256, 256, 1)]
conv2d: Conv2D
input: (None, 256, 256, 1)
output: (None, 255, 255, 4)
max_pooling2d: MaxPooling2D
input: (None, 255, 255, 4)
output: (None, 127, 127, 4)
dropout: Dropout
input: (None, 127, 127, 4)
output: (None, 127, 127, 4)
flatten: Flatten
input: (None, 127, 127, 4)
output: (None, 64516)
dense: Dense
input: (None, 64516)
output: (None, 32)
dense_1: Dense
input: (None, 32)
output: (None, 4)

Figure 1: *Architecture of the base model.*

The assessment metrics for this model, including `accuracy` and `loss` during its training phase, as well as its generalization capabilities, are detailed in the evaluation section further in the document.

In each of the examples discussed in this review, the `get_model()` function is utilised to define the model. Here is the function used for defining the baseline model:

baseline architecture

```python
def get_model():
    model = Sequential()
    model.add(InputLayer(shape=your_shape)
    model.add(Conv2D(4, (2,2), activation='relu')) # Convolutional Layer, for Image
    ↪ Processing
    model.add(MaxPool2D((2,2))) # Pooling Layer, combines nearby pixels to a Mean
    ↪ Color
    model.add(Dropout(0.4)) # Prevents overfitting by dropping out neurons
    model.add(Flatten()) # Converts from 2D to 1D
    model.add(Dense(32, activation='relu'))# Densely connected nodes layer
    model.add(Dense(4, activation='softmax'))# Output Layer, defines final shape
    model.compile(
        loss='categorical_crossentropy',
        optimizer=keras.optimizers.Adam(learning_rate=0.001),
        metrics={['accuracy']
    )
    # Compile the model with a loss type, optimizer, and evaluation metric
    return model

Model = get_model()
```

Efficient model design using hyperparameter tuning

In this section, we offer a thorough description and tutorial on using different hyperparameter tuning libraries within the framework of the previously discussed image classification problem. Our goal is to showcase how these libraries can improve model performance by fine-tuning essential parameters, thereby achieving more accurate and efficient models. We will delve into various libraries, analyze their features, and demonstrate their application through a hands-on example.

Scikit-Learn

Scikit-Learn provides two main ways to evaluate hyperparameters using its model selection tools: parameter sampling algorithms and automatic cross-validation search algorithms. Both approaches involve providing models with a dictionary in which keys represent hyperparameters to optimize and values define the search space. The examples use the following dictionary for both methods.

search space:example

```
param_distributions={
    'Layers':[1,2,3,4,5],
    'Activation':["gelu", "relu", "tanh"],
    'Nodes':[2, 10, 18, 26, 32],
    'Kernel':[2,3,4],
    'Add_Drop':['True','False'],
    'Drop_Rate':[0.3, 0.35, 0.4, 0.45,0.5, 0.55],
    'Learning_Rate':[0.01, 0.005, 0.001,0.0005, 0.0001]
}
```

Approach 1: ParameterSampler

Scikit-Learn provides two sampling methods: the ParameterSampler [3] and the GridSampler [4]. Both operate within a predefined search space, with the former generating a fixed number of samples and the latter producing all possible hyperparameter combinations. This section will specifically focus on using the ParameterSampler class for clarity.

The ParameterSampler class requires a `param_distributions` parameter (a dictionary) to define the parameters to sample from, and an `n_iter` parameter (an integer) that determines the number of sample combinations to return. The sampler then outputs these samples as dictionaries, allowing for iteration.

0. **Setup**

```
def get_model(param):
```

For a sampler to interact with the `get_model()` function, the model function should be declared with a parameter that allows a dictionary of parameters to be passed. Below is an example of such a function definition:

1. **Basic Tuning:**

```python
# Can Sample integers, strings, or any objects all the same type
model.add(Dense(param['Nodes'], activation='relu'))
model.add(Dense(12, activation=param['Activation']))
model.add(Conv2D(12, (param['Kernel'], param['Kernel']), activation='relu'))
model.compile(
    loss='categorical_crossentropy',
    metrics=['accuracy'],
    optimizer=keras.optimizers.Adam(learning_rate=param['Learning_Rate'])
)
```

Number of layers:

To enhance the flexibility of the model, the ParameterSampler allows you to define the number of layers dynamically based on the parameters being sampled. This ensures that the model architecture can adapt to varying levels of complexity, depending on the specific needs of each trial. For example, you can specify a range for the number of layers, and the sampler will select a value within this range for each iteration.

Layers

```python
for i in range(0, param['Layers']):
    model.add(Dense(12, activation='relu')
```

Boolean layer:

The inclusion of a dropout layer is controlled by a boolean parameter, providing a straightforward way to manage regularization within the model. If the parameter `Add_Drop` is set to `'True'`, a dropout layer is added to prevent overfitting by randomly setting a fraction of input units to zero during training. This technique is particularly useful in larger networks, where the model might otherwise become too specialized to the training data.

```
Boolean Layer
if param['Add_Drop'] == 'True':
    model.add(Dropout(param['Drop_Rate']))
```

2. **Doing the Search** – Evaluate by iterating through a locally allocated results list.

```
from sklearn.model_selection import ParameterSampler
param_dict= ParameterSampler(param_distributions=param_distributions, n_iter=20)
trial_list = []
for param in param_dict:
    model = get_model(param)
    history = model.fit(train_data, train_labels, epochs=10,
    validation_data=(test_images, test_labels))
    loss, accuracy = model.evaluate(test_images, test_labels, verbose=0)
    trial_list.append({
        'val_accuracy': accuracy,
        'val_loss': loss,
        'params': param,
    })
```

Method 2: RandomizedSearchCV

The other frameworks encompass a group of validation algorithms, including the Randomized-SearchCV [5], GridSearchCV [6], HalvingGridSearchCV [7], and HalvingRandomSearchCV [8] classes. The constructor accepts an estimator, in this instance a `SciKeras` model, along with a search space formatted identically to the previously mentioned example.

While the earlier samplers merely return sets of different parameters, these cross-validation searchers automate the selection and validation of hyperparameters. Furthermore, 'CV' denotes the number of distinct `fit()` executions each parameter set undergoes; this process ensures that the accuracy remains consistently at an acceptable level when determining the optimal model for a specific application.

0. **Setup**

When defining a `get_model()` function, it is essential to specify each parameter intended for tuning. This ensures that the class can access the names in the function that match the values within the `param_distributions`. *Note: dv -> default value*

Defined Parameters

```
def get_model(Nodes=dv, Activate=dv, Kernel=dv, Learning_Rate=dv, ...):
```

1. **Basic Tuning:**

```
#Can Sample integers, strings, or any objects all the same type
model.add(Dense(Nodes,activation='relu')
model.add(Dense(12, activation=Activation)
model.add(Conv2D(12, (Kernel, Kernel), activation='relu')
model.compile(loss='categorical_crossentropy',metrics=['accuracy'],
optimizer=keras.optimizers.Adam(learning_rate=Learning_Rate)
```

Number of Layers:

```
for i in range(0, Layers):
    model.add(Dense(12, activation='relu')
```

Boolean Additions:

```
if Add_Drop == True:
    model.add(Dropout(Drop_Rate))
```

2. **Doing the Search**

To use CV models, include the SciKeras [9] library and KerasClassifier [10] class for Scikit-Learn to interpret and evaluate the model. The KerasClassifier constructor takes a Keras model, fit parameters (`batch_size, epochs, etc.`), and `get_model()` parameters.

```
model = KerasClassifier(model=get_model, epochs=10, batch_size=32, Layers=dv,
↪ Activation=dv, Nodes=dv, Kernel=dv, Add_Drop=dv, Drop_Rate=dv, Learning_Rate=dv)
```

After construction, the model works with RandomizedSearchCV. This class's constructor takes `param_distributions`, `n_iter`, `cv` for cross-validation trials, and the `estimator` (KerasClassifier). This RandomizedSearchCV class can then be fit using training data, a portion of which the random search automatically selects for validation.

```
random_search = RandomizedSearchCV(estimator=model,
↪  param_distributions=param_distributions, cv=3, n_iter=10)
random_search_result = random_search.fit(train_data, train_labels)
```

Access results using the `'cv_results'` attribute or `'best_estimator'`. Convert the estimator back to a Keras model by saving it as a `.keras` file.

Saving Results

```
results = random_search_result.cv_results_
model = random_search_result.best_estimator_
model.model_.save(f"{/path}/{name}")
keras_model = keras.saving.load_model(f"{/path}/{name}")
```

Optuna

Optuna [11] is a framework for automatic hyperparameter optimization in machine learning. It offers an automated search for optimal hyperparameters using Python conditionals, loops, and syntax. Optuna includes advanced algorithms and supports parallel processing to enhance the efficiency of the search process.

Optuna provides various samplers [12] and pruners [13] to improve the hyperparameter optimization process. In this manual, only the Tree-Structured Parzen Estimator (TPE) [14] sampler will be demonstrated.

The TPE uses Gaussian Mixture Models to divide parameter values associated with the best objective values from the rest. By minimizing B(x)/R(x) ***(best over rest), the TPE*** can generate optimal combinations of hyperparameters. For more detailed information on the TPE Sampler, refer to the Optuna Documentation.

1. **Setup and Background**

   Optuna frequently uses these terms:

- Trial [15]: A single hyperparameter configuration.
- Study [16]: A complete set of trials with all configuration data.
- **Pruning**: Early stopping of a trial due to poor results.
- **Sampling**: Algorithm selecting which hyperparameters to test.
- **Objective**: Function aimed at maximization or minimization.

HP Tuning with Optuna

```python
import optuna
def objective(trial):
    #make a model and add input
    model = keras.models.Sequential()
    model.add(keras.Input(InputShape))

    #ALL TUNING CODE GOES HERE

    model.compile(
        loss='categorical_crossentropy',
        optimizer=keras.optimizers.Adam(),
        metrics=['accuracy']
    )
    history = model.fit(train_data, train_labels, epochs=50,
    validation_data=(test_data, test_labels))
    loss, accuracy = model.evaluate(test_data, test_labels)
    return accuracy
```

Optuna operates differently from the other libraries in this manual. Any integration or wrappers for Optuna to directly evaluate a Keras model are outdated, so the only feasible approach is to use a simple objective study to maximize or minimize a certain recorded value.

To obtain the accuracy or loss for Optuna to evaluate trials, additional steps are required after compiling the model. Specifically, it is possible to obtain the accuracy of a model by using `model.evaluate(2)` after using `model.fit()` to train it on the training data. The method for constructing a function that works with Optuna is as follows:

2. **Basic Tuning**

The trial object in Optuna is essential for changing hyperparameters in the model. It has several functions, commonly used in three categories:

1. `suggest_int(5)`: Suggests a random integer between low and high. Parameters like step or log can also be included.
2. `suggest_categorical(2)`: Suggests a random value from a list of values of the same type.
3. `suggest_float(5)`: Suggests a random float between low and high, with optional parameters step or log.

a) `Step`: Selects values as `low + (x)(step)`

b) `Log`: Picks lower values more often by giving the parameter space a 'log' domain

```
Activation = trial.suggest_categorical("Activation", ["gelu", "relu", "tanh"])
model.add(Dense(12, activation=Activation))
model.add(Dense(trial.suggest_int('Nodes', low=2, high=32, step=2),
 ↪  activation='relu'))
model.add(Dropout(trial.suggest_float('Drop_Rate', low=0.2, high=0.5,
log=True))
Learning_Rate = trial.suggest_categorical('learning_rate', [0.001, 0.01, 0.1])
model.compile(
    loss='categorical_crossentropy',
    metrics=['accuracy'],
    optimizer=keras.optimizers.Adam(learning_rate=Learning_Rate)
)
```

Number of Layers:

```
Layers = trial.suggest_int('Layers', 1, 5)
for i in range(0, Layers):
model.add(Dense(12, activation='relu'))
```

Boolean Addition:

```
Drop_Rate = trial.suggest_float('Drop_Rate', 0.2, 0.5)
if trial.suggest_categorical('Add Drop',['True', 'False']) == 'True':
    model.add(Dropout(Drop_Rate))
```

3. **Doing the Search**

   After defining the hyperparameter space, creating and executing a study is simple. Create a study, then use its optimize method to find the best hyperparameters based on given parameters and an objective.

```
study = optuna.create_study(study_name=study_name, direction="maximize")
study.optimize(objective=objective, n_trials=20)
# to get all the parameters for the best model
print(" Value: ", study.best_trial.value)
print(" Params: ")
for key, value in trial.params.items():\\
    print("{}: {}".format(key, value))
```

Keras Tuner

Keras Tuner [17] is a user-friendly, scalable hyperparameter optimization framework that uses 'define by run' syntax and supports three search algorithms. It primarily integrates with Keras models, with limited support for other models through some open-source libraries.

This manual focuses on Keras' Hyperband tuner [18], which creates tournament-style brackets for parameter configurations. Winners are evaluated against other configurations, optimized by running only a subset of epochs early on to gauge trial promise. The number, size of brackets, and evaluation length are determined by the `max_epochs` and `factor` parameters, explained further in the additional information section. Ideally, set `max_epochs` slightly above the expected training epochs, and use a higher reduction factor if expecting many inadequate configurations (standard is 3-4).

1. **Setup and Background**

   Similar to Optuna, Keras Tuner has its own terminology essential for understanding the fundamentals of its framework:

- `Trial`: A single configuration of hyperparameters; when using the hyperband tuner, one trial may serve as a continuation or validation of another trial.
- `Tuner` [19]: An object provided by Keras Tuner to conduct the hyperparameter search.
- **Search**: A method of the tuner that activates the tuner and initiates the search process.

- Oracle [20]: Stores all the configuration and active-state information.

```python
import keras_tuner
def get_model(hp):
    model = Sequential()
    model.add(Input(InputShape))

    #ALL TUNING CODE

    model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )
    return model
```

The configuration of Keras Tuner is quite minimal, resembling a combination of Scikit-Learn's CV `get_model()` function and Optuna's `objective(trial)` function. This function requires only one parameter, hp, adhering to standard naming conventions, and accesses all hyperparameters through this parameter. Unlike Optuna, where the model is fit within the function, Keras Tuner only returns the model, similar to the approach used in Scikit-Learn's function.

2. **Basic Tuning**

   Basic tuning in Keras Tuner is similar to Optuna, using the hp object to apply new hyperparameter values.

   1. `hp.Int(5)`: Returns an integer between `min_value` and `max_value` based on step and sampling parameters.
   2. `hp.Float(5)`: Returns a float value within a specified range.
   3. `hp.Choice(3)`: Selects and returns an option from a list with equal probability; all options must be of the same type.
   4. `hp.Boolean(1)`: Executes a `hp.Choice()` with `True` and `False` as options.

```python
Activation = hp.Choice(name="Activation", values=["gelu", "relu", "tanh"])
model.add(Dense(12, activation=Activation))
model.add(Dense(hp.Int('Nodes', min_value=2, max_value=32, step=2),
activation='relu'))
```

```python
model.add(Dropout(hp.Float('Drop_Rate', 0.2, 0.5))
Learning_Rate = hp.Float('Learning_Rate', 0.001, 0.1, sampling='log')
model.compile(
    loss='categorical_crossentropy',
    metrics=['accuracy'],
    optimizer=keras.optimizers.Adam(learning_rate=Learning_Rate)
)
```

Number of Layers:

```python
Layers = hp.Int('Layers', min_value=1, max_value=3)
for i in range(0, Layers):
    model.add(Dense(12, activation='relu')
```

Boolean Addition:

```python
if (hp.Boolean('Add_Drop')):
    model.add(Dropout(hp.Float('Drop_Rate', 0.2, 0.5))
```

5. **Doing the Search**

   Keras Tuner offers an efficient and straightforward method for conducting and assessing searches, similar to Optuna. To initiate the search process, one needs to construct the tuner and utilize its search method. Upon completion, the outcomes can be retrieved using the `tuner.get_best_hyperparameters()` method.

```python
tuner = kt.Hyperband(get_model, objective="val_accuracy", max_epochs=20, factor=3)
tuner.search(train_data, train_labels, validation_data=(test_data, test_labels))
print(tuner.get_best_hyperparameters(num_trials=1))
best_model = tuner.get_best_models()[0]
best_model.save(f"{/path}/{name}")
```

Advanced Usage

Each of these frameworks has unique characteristics and use cases that may not be fully covered by documentation and may require further research. This section will examine the higher-level functionality of each library.

Greater Dynamical Allocation of Layers

So far in this manual, only the simple allocation of standardized hyperparameters with a single source of variance (e.g., nodes, kernel size, layers) has been discussed; however, all three of these libraries can dynamically allocate multiple hyperparameters simultaneously.

Optuna:

```python
for i in range(1, ConvLayers+1):
    nodes = trial.suggest_int(f'ConvL{i}Nodes', int(i), int(10*i))
    kernel = trial.suggest_int(f'ConvL{i}Kernel', 2, 4)
    model.add(Conv2D(
        nodes,
        (kernel, kernel),
        name=f'ConvL{i}',
    ))
```

Keras tuner:

```python
for i in range(1, ConvLayers+1):
    nodes = hp.Int(f'ConvL{i}Nodes', int(i), int(10*i))
    kernel = hp.Int(f'ConvL{i}Kernel', 2, 4, default=2)
    model.add(Conv2D(
        nodes,
        (kernel, kernel),
        name=f' ConvL{i}'
    ))
```

Scikit-learn (example 1):

```python
for i in range(1,param['ConvLayers']+1):
    model.add(Conv2D(
        param[f'ConvL{i}Nodes'],
        (param[f'Conv_L{i}_Kernel'], param[f'Conv_L{i}_Kernel']),
        name=f'ConvL{i}'
    ))
```

Scikit-learn (example 2):

```python
for i in range(1, Conv_Layers+1):
    model.add(Conv2D(
        locals()[f'Conv_L{i}_Nodes'],
        (locals()[f'Conv_L{i}_Kernel'], locals()[f'Conv_L{i}_Kernel']),
        name=f'Conv_L{i}'
    ))
```

Logging of Critical Information in JSON files

1. **Keras Tuner**

   Keras Tuner logs every trial in a JSON file with corresponding weights in a trial folder. Additionally, timestamps are in the oracle.json file

2. **Scikit-Learn Option 1**

SKLearn:JSON

```python
param_dict = (ParameterSampler(param_distributions, n_iter=some_number))
trial_count = 0
for param in param_dict:
    #ALL PRIOR PARAM SAMPLER CODE
    loss, accuracy = model.evaluate(test_data, test_labels, verbose=0)
    trial = {
        'trial_number': trial_count,
        'val_accuracy': accuracy,
        ' val_loss': loss,
        ' params': param,
        #ANY OTHER DATA AVAILABLE CAN BE RECORDED IN HERE
    }
    if (accuracy > some_number):
        model.save_weights(f'{/path}/trial_{trial_count}.weights.h5')
        out_file = open(f'{/path}/trial_{trial_count}.json','w')
    json.dump(trial, out_file)
    out_file.close()
    trial_count += 1
```

3. **Optuna**

   Optuna's study and trial objects can store custom attributes using key-value pairs, typically assigned within the objective function.

   Optuna also supports custom callback functions that run at the end of each trial, accessing both trial and study attributes. It's best to use these callbacks to update or create study attributes based on trial attributes.

Optuna:JSON

```
def objective(trial):
    if 'best_model' not in trial.user_attrs: #set a default value for the model
        trial.set_user_attr(key= "best_model", value=None)

    #Define model and hyperparameters
    #Evaluate model
    trial.set_user_attr(key="best_model", value=model)

    return accuracy
--------------------------------------------------------------------------------
def callback(study, trial):
    if (trial.value > some_number):
     ↪  trial.user_attrs["best_model"].save_weights(f'{/path}/trial_{trial.number}.h5')
    trial_json = {
        'trial_number': trial.number,
        'value': trial.value, #the accuracy
        'params': trial.params
    }
    out_file = open(f'{/path}/trial_{trial.number}.json', 'w')
    json.dump(trial_json, out_file, indent=6)
    out_file.close()

-------------------------------------------------------------------------------------
import joblib
study = optuna.create_study(study_name='some_name', direction="maximize")
study.set_user_attr(key='some_study_attribute', value=[])
study.optimize(objective, n_trials=some_number, callbacks=[callback])
joblib.dump(study, f'{/path}/{'some_name'}_STUDY.pkl')
```

Lastly, it is possible to store the study after completion in a `.pkl` (pickle) file

5. **Scikit-Learn Option 2**

Scikit-Learn's collection of CV models has limited compatibility for storing information in JSON files; however, there is a partial solution to at least store accuracy for evaluating data over the course of trials. To understand how this works, knowledge about the scoring attribute of all Scikit-Learn CV models must be obtained.

After a parameter set is assessed, the CV model executes a final scoring function to obtain a metric to compare against all other models. This function receives the true data labels as well as the predicted data labels from the `.predict()` of the trained model. Scikit-Learn supports the creation of custom scoring functions, which can be used to store the accuracy from a built-in metric function and transfer it to a JSON in the scoring function.

SKLearn:JSON

```
import tensorflow as tf
def scoring_function(true, predicted):
    DIR = {/path}
    trialNum = len([name for name in os.listdir(DIR) if
     ↪  os.path.isfile(os.path.join(DIR, name))])
    accuracy_function = keras.metrics.Accuracy()
    accuracy_function.update_state(true, predicted)
    accuracy = float(tf.keras.backend.get_value(accuracy_function.result())) #Need
     ↪  to convert since its a tensor
    trial = { 'trial_number': trialNum, 'val_accuracy': accuracy, }
    out_file = open(f'{/path}/trial_{trialNum}.json', 'w')
    json.dump(trial, out_file, indent=6)
    out_file.close()
    return accuracy
--------------------------------------------------------------------------------
score = sklearn.metrics.make_scorer(scoring_function, greater_is_better=True)
random_search = RandomizedSearchCV(model, param_distributions, some_number,
↪   scoring=score)
```

If the predicted data is not categorical or based on probability, it is also possible to record the correct loss here.

Error Handling with Dynamical Layer Allocation

Due to the customizable nature of the hyperparameter space, managing errors caused by a model while maintaining accurate logs for later reconstruction can be challenging. Furthermore, to avoid bias towards models with parameters that tend to result in fewer errors, it is possible to track and prevent models from compiling and generating errors. This is achieved by recording and calculating how Keras modifies the model dimensions with each layer, and then preventing and documenting potential errors when adding a layer.

It should be noted that all libraries [22] have methods to prevent errors from being fatal; however, this aims to prevent dataset bias and reduce time spent discarding trials.

Example of Error

> Certain Keras layers modify the size of the data, which can be challenging when determining the factor or number by which these modifications apply. For example, in convolutional 2D models that use pooling layers, the output dimension is reduced to `math.floor(CURRENT_DIM/POOL_SIZE)`. If the output is not greater than or equal to `POOL_SIZE`, the model will become invalid and generate an error.
>
> For detailed information on how the output dimension is influenced by a specific layer type, please refer to the Keras Layers API [23].

1. Output Exception Tracking and Preventing Errors

When developing a customized `get_model()` function, it is essential to monitor how the layer configuration impacts the output size. One approach is to utilize an integer to track the current dimension of themost critical output dimension, typically the smallest dimension if the samples are not square or cubic. Furthermore, it is beneficial to maintain a list that identifies the layers the model was unable to construct. The following example illustrates this method with a convolutional model.

Error Handling

```python
import math
Output_Exception_List = []
Output = IMAGE_DIMENSION
pool_size, layers = frameworkIntSelect(params), frameworkIntSelect(params)
for i in range(1, layers+1):
    nodes = frameworkIntSelection(params)
    kernel = frameworkIntSelection(params)
    if (Output - kernel + 1 > 0):
        Output -= kernel - 1
        model.add(Conv2D(nodes, (kernel, kernel)))

    else:
        Output_Exception_List.append(f'Convolutional_L{i}')

    pool_exist = hp.frameworkListSelect(params)
    if pool_exist == 'True':
        if (Output >= pool_size):
            Output = math.floor(Output/pool_size)
            model.add(MaxPooling2D((pool_size, pool_size))
        else:
            Output_Exception_List.append(f"Pooling_L{i}")
```

2. Storing of Output Exceptions

2a. Scikit-Learn Option 2 (CV)

As of August 2024, there is no established method to store the weights required for reconstructing significant portions of models using the cross-validation set of model-selection algorithms. While it is possible to store the output list in a JSON format if necessary, this proves to be less useful without the inclusion of the weights.

2b. Scikit-Learn Option 1 (ParameterSampler)

When using the ParameterSampler, the `Output_Exception_List` should be returned along with the model from the `get_model()` function. Once obtained, it should be added to the storage JSON.

```python
def get_model(param):
    #make a model and add input
    Output_Exception_List = []
    Output = IMAGE_DIM
    #Do all output tracking and hyperparameter config.
    return model, Output_Exception_List
```

2c. Optuna

In Optuna, it is possible to use the trial attributes, callback functions, and trial JSONs to easily store the exception list for every `trial`.

```python
def objective(trial):
    #make a model and add input
    OE_List = []
    Output = IMAGE_DIMENSION
    trial.set_user_attr(key="OE_List", value=OE_List)
    #compile, fit, and evaluate model
    return accuracy
----------------------------------------------------------------------------------------
def save_trial_callback(study, trial):
    exception_list = trial.user_attrs["OE_List"]
    json = {
        #all other data
        'output_exception_list'{} : exception_list
    }
    #open and dump
```

2d. Keras Tuner

In Keras Tuner, you can save the exception list only in the `get_model()` function, allowing for either saving all or none of the lists. The method below estimates the correct trial by counting the files in the directory created by Keras, with final trials reaching into the thousands.

```python
def get_model(hp):
    #make a model and add input
    Output_Exception_List = []
    Output = IMAGE_DIM
    #All tuning code
    trial_number = len(os.listdir(f'{/path}/{keras_directory}'))-3
    number, count = trial_number, 0
    prepends, prepend = ["000", "00", "0", ""], None
    while (prepend == None):
        if (number > 10):
        prepend = prepends[count]
        number /= 10
        count += 1
    if (trial_number >= -1):
        if trial_number == -1:
            trial_number += 1 #error happens when doing the first trial only
        with open(f'{/path}/{keras_direct}/OE.json','w') as out_file:
            json.dump(Output_Exception_List, out_file, indent=6)
            out_file.close()
    return model
```

Model Reconstruction after the Tuning Process

After the tuning process, to evaluate which model performs best on a larger set than just a validation set, the models must be reconstructed. This process requires understanding how data is stored in each JSON file and how to correctly use any relevant information stored elsewhere. This manual will provide information on accessing the necessary data to assist in evaluating the optimal model configuration, rather than providing the code to do this.

1. **Accessing Data from a JSON**

```python
    with open(f"{/path}.json") as json_file:
        data = json.load(json_file)
```

2. **Accessing Values from Data**

```python
    params = data['params']
```

The data object is a python dictionary, so accessing data is done by square bracket and key-value notation.

Although most of the data is custom configured, Keras Tuner has a complicated system for accessing data due to the scale of its json files

1a. Accessing data from Keras Tuner

```python
if (data['metrics']['metrics'] != {}):
    val_accuracy =
    ↪  data['metrics']['metrics']['val_accuracy']['observations'][0]['value'][0]
    val_loss = data['metrics']['metrics']['val_loss']['observations'][0]['value'][0]
    params = data['hyperparameters']['values']
    Layers = params['Layers']
    #Also note, Keras Tuner stores trials in a 4 digit trial number format no matter
    ↪  the number
    #Additionally, all weights files are named /checkpoint.weights.h5 in the
    ↪  trial_#### directory
```

3. **Constructing Model and Loading Weights**

```python
model_dict = {}
for trialNum in desired_accuracy: #list of all models that one wants to reconstruct
    with open(f"{/path}") as json_file:
    data = json.load(json_file)
    params = data['hyperparameters'][' values']
    model = keras.models.Sequential()
    model.add(keras.Input(InputShape))
    for i in range(1, params[f'Layers']+1):
        model.add(Dense(params[f'Dense_L{i}_Nodes'],
        ↪  activation=params['Activation'], name=f'Dense_L{i}'))
        model.add(Dense(Output_Classes, activation='softmax', name='Output'))
        model.compile(
            loss='categorical_crossentropy',
            optimizer=keras.optimizers.Adam(),
            metrics=['accuracy']
        )
        model.load_weights(f"{/path}/weights.h5")
        model_dict[trialNum] = model
```

Also, if the Output Exceptions are utilized, refrain from including its data when constructing

Verbosity

Nearly all algorithms offer ways to visualize the tuning process, which aids in debugging and understanding their functions. However, reaching message rate limits can cause errors. To fix this, each framework has options to limit logging to errors only.

1. **General Keras Models** – Scitkit-Learn Option 1, Optuna

```
history = model.fit(train_data, train_labels, verbose=0, validation_data=(test_data,
↪  test_labels))
loss, accuracy = model.evaluate(test_images, test_labels, verbose=0)
```

2. **Optuna**

```
optuna.logging.set_verbosity(optuna.logging.WARNING)
```

3. **Scikit-Learn Option 2 (CV)**

```
rs = RandomizedSearchCV(estimator=model, param_distributions=param_distributions,
↪  verbose=0)
```

4. **Keras Tuner**

```
tuner.search(train_data, train_labels, validation_data=(test_data, test_labels),
↪  verbose=0)
```

Lower verbosity integers indicate less logging. See documentation [24] for detailed definitions of each verbosity level.

Miscellaneous Information

Algorithms

Hyperband –

The Hyperband algorithm [18] is an extension of the Successive Halving algorithm. Successive Halving allocates a desired number of hyperparameter configurations, and then follows the name by successively eliminating the lesser half of the configurations until one remains after a certain number of epochs. The issue arises when trying to determine whether to evaluate a fewer number of large models with long training times and more epochs around a smaller search space, or many smaller models with short training times and fewer epochs but provide large number of configurations. It is extremely difficult, and time consuming, during a hyperparameter search to try and evaluate if a model can have many optimal hyperparameter configurations or if there is one specific area in a concentrated place where the hyperparameters converge to a significantly lower number.

The Hyperband algorithm optimizes the Successive Halving algorithm by enabling the evaluation of configurations with many sufficient hyperparameter values as well as of configurations in a single area. It does this by allocating tournament style brackets with each hyperparameter configuration competing against the others in its bracket. In each respective bracket, there is a tradeoff between an aggressive elimination, high exploration approach and a more conservative, passive elimination approach. The parameters that determine the number of brackets and reduction after each step are $\boldsymbol{R}$ and $\boldsymbol{n}$.

R is the maximum resource budget, which is to be set based on how many resources are to be allocated total to each configuration at the absolute maximum. This determines how many brackets and configurations are to be tried as well as how many configurations are within each bracket, as each bracket is only subject to the constraint that it must allocate ***R*** resources to at least one model to figure out what configurations are most optimal.

n is the reduction factor for the algorithm. This means that the higher ***n*** is the more configurations will be eliminated in each elimination phase within a certain bracket.

R and ***n*** vary inversely, as the more maximum resources that are possibly allocated, the longer the algorithm will take; however, the higher the reduction factor, the faster configurations are eliminated, and the faster the algorithm completes. When selecting these numbers, evaluate how many configurations and how deep these configurations should go versus how much time can be allocated for the entire algorithm to run.

Keras Tuner Attributes

`hp.conditional_scope(2)` –

Keras Tuner includes this method in its `hyperparameter` class. This method is not an if statement; it excludes parameters from the tuner parameter space if their value does not match the accepted value provided.

Keras Hyperband –

The Keras Tuner Hyperband Tuner [18] employs the hyperband algorithm to identify the optimal hyperparameters for a model. In this context, the ***R*** term corresponds to the `max_epochs` parameter, while the ***n*** term refers to the `factor` parameter. Additionally, the tuner accepts a `hyperband_iterations` parameter, which specifies the number of times the complete hyperband algorithm will be executed before concluding the search. Although numerous other parameters can be configured, these are the most critical.

When selecting parameters for the tuner, it is essential to carefully consider the time and resources allocated for the study. For instance, if the study involves a large model with high training costs, it is advisable to set a lower value for `max_epochs` and `hyperband_iterations`, or a higher value for `factor`. The algorithm will execute approximately `max_epochs * (math.log(max_epochs, factor)** 2)* hyperband_iterations` cumulative epochs across all trials. It is recommended to allocate these values generously within the resource budget, as allowing more trials to progress and extending the search duration typically yields superior results.

Other attributes –

- `hp.Fix()`- stops certain hyperparameters from getting changed during tuning
- Tracking based on time – use `os.path.getmtime(file_path)`
- SKLearnTuner- performs cross validating searches for Scikit-Learn models
- Custom Training Loops [25a]
- Distributed Hyperparameter Tuning [25b]
- Raising Errors Based On Failed Trials [25c]
- Visualization Using Tensorboard [25d]
- Tailor Search Space for Prebuilt Models [25e]
- More Customization with a Hypermodel [25f]
- GITHUB [25g]

Optuna Attributes

Pruning –

Pruning [13] refers to the process of automatically terminating trials that are unlikely to succeed during the early stages of training. In Optuna, pruning can be accomplished either by creating a custom training loop, where the model fitting is controlled by the user and the report method is used, or by using specialized classes from an Optuna sub-library, `optuna_integration` [26]. Optuna also provides a callback function specifically designed for Tensorflow-Keras, allowing trials to be pruned efficiently during the fitting phase of model training by incorporating this custom callback function.

```
from optuna_integration.tfkeras import TFKerasPruningCallback
def objective(trial):
    #make model and add input
    #define and apply all layers and hyperparameters
    callbacks = [
        keras.callbacks.EarlyStopping(monitor='val_loss', patience=5),
        TFKerasPruningCallback(trial, 'val_accuracy')
    ]
    model.compile(
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )
    history = model.fit(train_data, train_labels, epochs=50,
    ↪  validation_data=(test_data, test_labels), callbacks=callbacks)
    #evaluate the model
    return accuracy
study = optuna.create_study(pruner=optuna.pruners.HyperbandPruner(min_resource=3,
↪  max_resource=50, reduction_factor=3))
```

This example uses the Hyperband algorithm, where `min_resource` specifies the number of epochs to run before pruning.

Initializing Trial Attributes –

Although pruning is a useful technique, it is essential to initialize trial attributes at the beginning of the objective statement when storing information in JSON or any other file type. Failure to do so may result in a fatal error. This occurs because the pruning callback completely exits the entire `objective()` function, given that evaluating an unpromising dataset has little value. Consequently, this becomes problematic when attempting to save metrics such as the test validation loss for each trial, as the test validation loss will not be calculated. Hence, the initialization of any trial attributes must be conducted before fitting the model.

```
def objective(trial):
    if 'val_loss' not in trial.user_attrs:
        trial.set_user_attr(key="val_loss", value=999.9999)
    #Create Model and Input
    #Define Hyperparameters
    #Compile and Fit
    accuracy, loss = model.evaluate(test_data, test_labels)
    trial.set_user_attr(key="val_loss", value=loss)
    return accuracy
```

Other attributes – from Optuna docs

- Visualization using Optuna [27a]
- Tutorials [27b]
- Parallelization [27c]
- GITHUB [27d]

Overall Evaluation

All frameworks discussed in this manual are advanced, efficient, and beneficial over manual tuning for any large-model projects. However, each library has specific use cases where it may result in marginally less time spent or slightly better outcomes. This section aims to clarify which approaches to take when tuning a model.

Manual Tuning

Pros	**Cons**
• No setup or overhead required • Full customization over the entire hyperparameter space • Any data wanted can be manually logged if it is printable • Great for smaller models or less complex problems • Creates an intuitive understanding of the model and how each hyperparameter effects the accuracy	• Not Scalable • Not easy to do any parallelization • Downtime between switching hyperparameters between each trial • No sophisticated search algorithm to help converge on a solution • Easy to converge on a local solution but not the overall best solution • Logging is time consuming • Hard to visualize the training process

Table 1: Manual Tuning Comparison

Manual tuning is useful for demonstrating that a model can perform well with a similar architecture. Simple problems can be resolved quickly with manual tuning without the need to define a hyperparameter space or set up a framework. However, when the problem exceeds a certain level of complexity, optimization algorithms generally perform better than manual tuning.

Scikit-Learn: Cross Validation Tuners

Pros	Cons
• Scikit-Learn is heavily integrated into many other machine learning libraries • Setup is easy for smaller search spaces • Great for simple models	• Not very customizable • Tough to log and visualize • Scaling is extremely tough • Not well integrated with parallelization • Only Grid and Random Search • Lots of overhead and requires intermediate libraries • Low size limits (approx. 30 terms)

Table 2: Scikit-Learn Hyperparameter Tuning Comparison

The Cross Validation Tuners are designed to optimize model performance; however, the search process can be time-consuming and may produce models with average results.

Scikit-Learn: Parameter Sampler

Pros	Cons
• Extremely easy setup • Errors are uncommon and do not pertain to the sampling algorithm • High customizability when establishing control loop • Easy to customize logging of wanted information	• Does not scale well • Low size limit on parameter space (approx. 30 terms, and only has a cap on the number of total integers in total) • Only samples parameters, does not apply them • Algorithm is poor, only Random Search and Grid Search • No easy visualization

Table 3: Parameter Sampler Comparison

Scikit-Learn's Sampler algorithms are useful for proof of concept projects, particularly when dealing with moderately complex problems that require tedious manual tuning. This framework is effective when the search space is small and aims to create a compact and efficient model. However, the sampler converges slowly, has a basic algorithm, and does not integrate with machine learning libraries.

Optuna

Pros	Cons
• Great integration with many machine learning libraries • Wide range of capabilities (Parallelization, Visualization, Database Logging, etc.) • Many tutorials and support • Multi-Objective trials are supported • Callback functions, trial/study attributes provide easy access to any information • Constantly being updated • Many options for sampling and pruning algorithms • High customizability of the parameter space • Able to define a certain budget and time limit • Algorithms are complex and converge on a great solution efficiently	• Fitting of model is done completely outside Optuna's control loop • Requires heavy lifting for local logging • Does not automatically save weights of models • Error prone • Requires definition of errors to be caught • Logging can be difficult to follow when debugging • Significant overhead

Table 4: Optuna Framework Comparison

Optuna is a sophisticated framework known for its visualization capabilities and compatibility. It offers many customizable options, such as callbacks, pruners, and samplers, providing extensive possibilities. However, with high customizability, there may be significant work required to define all these custom attributes. Additionally, the flexible structure of these attributes can lead to errors due to interferences with other aspects of the framework. Overall, Optuna is a robust framework for those seeking customizability and visualization, but it may involve complex setup and potential issues.

Keras Tuner

Pros	Cons
• Heavily Integrated with Keras • Easy Visualization with Tensorboard • Heavily Documented • Amazing Logging System • Good Error Handling • Simple Parallelization • Provides custom callback functions and custom training loops • Defining the search space is easy • Scalable, and has parallelization • Multi-Objectives are supported • A lot of complex use-cases and functions are developed and supported already • Saves model/weights from best epoch of the training, not just the end • Fast Algorithm	• Customizing requires an entire revamp of structure, making it hard to build off already developed code • Low support for information passing without a custom training loop • Some data documented takes lots of extra effort to process • No support for any other types of models aside from Scikit-Learn • Only has two efficient search algorithms • Not the focus of the Keras library, so has fewer features

Table 5: Keras Tuner Comparison

Keras Tuner is a framework that integrates well with model development and provides access to training control flow. However, it is specifically designed for Keras models, which may limit its applicability if multiple frameworks are required for a field of study. For those working with Keras and TensorFlow, Keras Tuner offers an efficient method for tuning models.

Summary

In theory, similar results can be obtained using Keras models by adequately setting up the search space and other parameters when working with any of these libraries. Each library has a specialized use case that provides efficient convergence to satisfactory results. Manual tuning works well for smaller models and datasets, but it becomes laborious when seeking high accuracy. Scikit-Learn offers a suitable framework for slightly larger problems than manual tuning can handle, though its compatibility limit is reached relatively quickly. Optuna and Keras Tuner offer advanced capabilities for addressing more complex problems. The choice between frameworks should be based on preference and the specific goals of the study.

Key factors in selecting a framework include dataset size, dataset complexity, the desired model library, and the need for customizability or specific outcomes not provided by the library.

Complexity?	Keras?	Need proof of concept?	Irregular Attributes?	Possible Choice(s):
Low	N/A	Yes	No	Manual Tuning
Low	N/A	No	No	Scikit-Learn, Manual Tuning
Medium	No	N/A	No	Scikit-Learn
Medium	Yes	N/A	No	Keras Tuner
High	No	N/A	N/A	Optuna
High	Yes	N/A	No	Keras Tuner, Possibly Optuna
High	Yes	N/A	Yes	Optuna, Possibly Keras Tuner

Links and References

1. Keras
2. Model Class
3. ParameterSampler
4. GridSampler
5. RandomizedSearchCV
6. GridSearchCV
7. HalvingGridSearchCV
8. HalvingRandomSearchCV
9. SciKeras
10. KerasClassifier
11. Optuna
12. Samplers
13. Pruners
14. Tree-Structured Parzen Estimator
15. Optuna Trial
16. Optuna Study
17. Keras Tuner
18. Hyperband Tuner (paper)
19. Tuner
20. Oracle
21. HP Object
22. Error Handling: Keras Tuner | Optuna | Scikit-Learn
23. Keras Layers API

24. Verbosity: Optuna | Scikit-Learn | Keras & Keras Tuner
25. Keras Additional Features:
    a. Custom Training Loops
    b. Distributed Hyperparameter Tuning
    c. Handling Failed Trials
    d. Tensorboard Visualization
    e. Search Space Customization
    f. Hypermodel
    g. GitHub Repository
26. Optuna Integration
27. Optuna Additional Features:
    a. Visualization Tools
    b. Tutorials
    c. Parallelization
    d. GitHub Repository