IASD M2 at Paris Dauphine

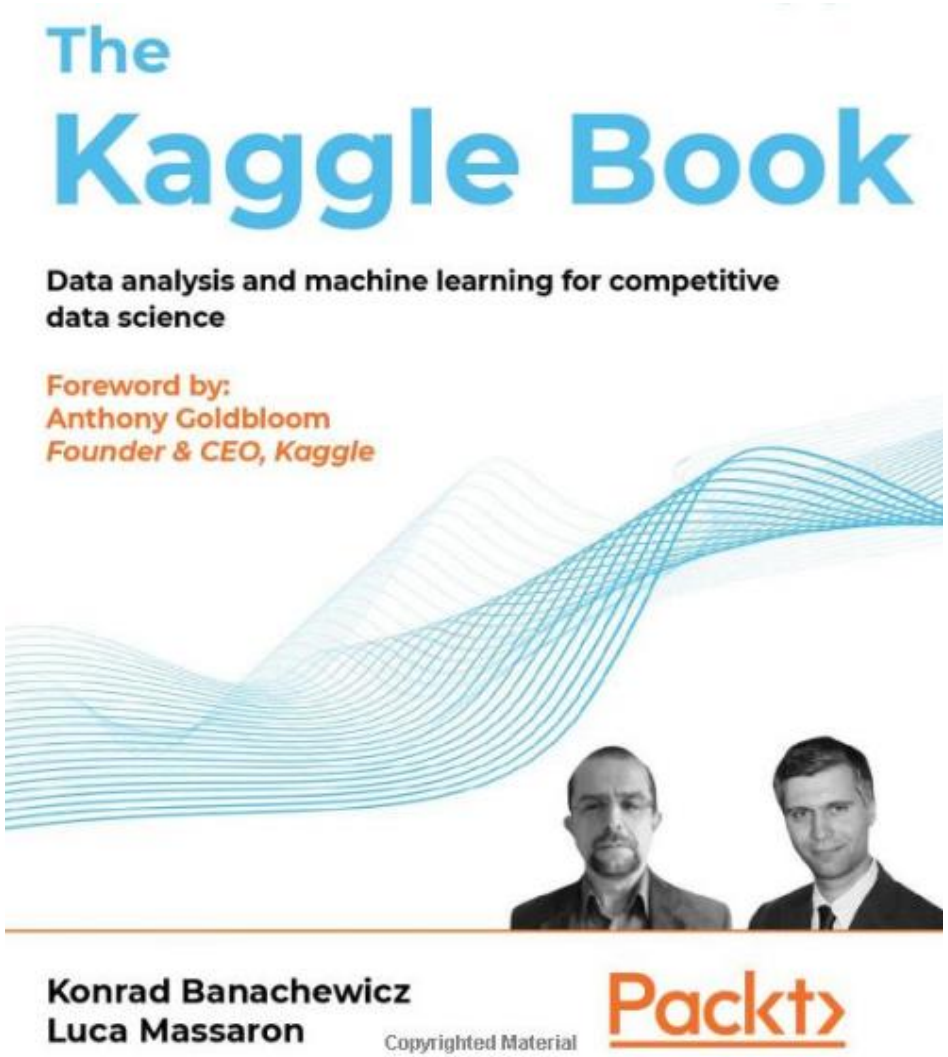
Become a Kaggle Master

4: Hyperparameters tuning

Eric Benhamou

Acknowledgement

The materials of this course is entirely based on the **seminal book**



Agenda

Part I: general concepts

1. Introduction to Kaggle (concept and API)
2. Competition, metrics
3. Validation
4. Hyper parameters tuning
5. Model ensemble with blending and stacking

Part II: Competitions

5. Predict Housing Prices
6. Predict Financial markets
7. Use NLP

Contents

- You will learn about:
  1. Basic optimization techniques
     a) Grid search
     b) Random search
     c) Halving search
  2. Key parameters and how to use them
     a) Optimizing for Deep networks
     b) Linear models
     c) Support-vector machines
     d) Random forests and extremely randomized trees
     e) Gradient tree boosting: LightGBM, XGBoost, CatBoost, HistGradientBoosting
  3. Bayesian optimization
     a) Using Scikit-optimize
     b) Customizing a Bayesian optimization search
     c) Extending Bayesian optimization to neural architecture search
     d) KerasTuner for Deep learning
     e) The TPE approach in Optuna

Why is hyperparameters tuning so important?

- How a Kaggle solution performs is not simply determined by the type of learning algorithm you choose.
- Aside from the data and the features that you use, it is also strongly determined by the algorithm's hyperparameters, the parameters of the algorithm that have to be fixed prior to training, and cannot be learned during the training process.
- Choosing the right variables/data/ features is most effective in tabular data competitions; however, hyperparameter optimization is effective in all competitions, of any kind.
- In fact, given fixed data and an algorithm, hyperpa- rameter optimization is the only sure way to enhance the predictive performance of the algorithm and climb the leaderboard.
- It also helps in ensembling, because an ensemble of tuned models always performs better than an ensemble of untuned ones.
- You may hear that tuning hyperparameters manually is possible if you know and understand the effects of your choices on the algorithm.

What Kaggle Grandmasters says

- Many Kaggle Grandmasters and Masters have declared that they often rely on directly tuning their models in competitions.
- They operate selectively on the most important hyperparameters in a bisection operation style, exploring smaller and smaller intervals of a parameter's values until they find the value that produces the best result. Then, they move on to another parameter.
- This works perfectly well if there is a single minimum for each parameter and if the parameters are independent from each other.
- In this case, the search is mostly driven by experience and knowledge of learning algorithms.
- In our experience, however, that is not the case with most tasks you will encounter on Kaggle.
- The sophistication of the problems and the algorithms used requires a systematic approach that only a search algorithm can provide. Hence, we decided to write this chapter.

Basic optimization

- The core algorithms for hyperparameter optimization, found in the **Scikit-learn package**, are **grid search and random search**.
- Recently, the Scikit-learn contributors have also added the halving algorithm to improve the performances of both grid search and random search strategies.
- In this part, we will discuss all these basic techniques. By mastering them, not only will you have effective optimization tools for some specific problems (for instance, SVMs are usually optimized by grid search) but you will also be familiar with the basics of how hyperparameter optimization works.

What should we optimize?

- To start with, it is crucial to figure out what the necessary ingredients are:
  - A model whose hyperparameters have to be optimized
  - A search space containing the boundaries of the values to search between for each hyperparameter
  - A cross-validation scheme
  - An evaluation metric and its score function
- All these elements come together in the search method to determine the solution you are looking for.
- Let’s see how it works.

Grid Search

- **Grid search** is a method that searches through the hyperparameters exhaustively, and is not feasible in high-dimensional space.
- For every parameter, you pick a set of values you want to test.
- You then test all the possible combinations in this set.
- That is why it is **exhaustive: you try everything**.

- It is a very simple algorithm and it suffers from the curse of dimensionality, but, on the positive side, it's embarrassingly parallel.
- This means you can obtain an optimal tuning very quickly, if you have enough processors to run the search on.

Grid search example

- As an example, let’s take a classification problem and **support-vector machine classification** (**SVC**).
- **Support-vector machines** (**SVMs**) for both classification and regression problems are probably the machine learning algorithm that you will use grid search for the most.
- Using the `make_classification` function from Scikit-learn, we can generate a classification dataset quickly:

Svm example

```
from sklearn.datasets import make_classification
from sklearn.model_selection import
train_test_split

X, y = make_classification(n_samples=300, n_features=50,
          n_informative=10, n_redundant=25,
          n_repeated=15, n_clusters_per_class=5,
          flip_y=0.05, class_sep=0.5,
          random_state=0)
```

Let us define the search space

- For our next step, we define a basic SVC algorithm and set the search space.
- Since the **kernel function** of the SVC (the internal function that transforms the input data in an SVM) determines the different hyperparameters to set, we provide a list containing two dictionaries of distinct search spaces for parameters to be used depending on the type of kernel chosen.
- We also set the evaluation metric (we use accuracy in this case, since the target is perfectly balanced):

```
from sklearn import svm
svc = svm.SVC(probability=True, random_state=1)
from sklearn import model_selection

search_grid = [{'C': [1, 10, 100, 1000], 'kernel’:
     ['linear']},{'C': [1, 10, 100, 1000], 'gamma’:
     [0.001, 0.0001],'kernel': ['rbf']}]
scorer = 'accuracy'
```

Optimization dictionaries

- In our example, a linear kernel doesn’t require the tuning of the gamma parameter, though it is very important for a radial basis function kernel.
- Therefore, we provide two dictionaries: the first containing the parameters for the linear kernel, the second containing parameters for a radial basis function kernel.
- Each dictionary only contains a reference to the kernel it is relevant to and only the range of parameters that are relevant for that kernel.

Let us combine

- It is important to note that the evaluation metric can be different from the cost function optimized by the algorithm.
- All the ingredients (model, search space, evaluation metric, cross-validation scheme) are combined into the GridSearchCV instance, and then the model is fit to the data:

Getting results

```
search_func = model_selection.GridSearchCV(estimator=svc
      param_grid=search_grid, scoring=scorer, n_jobs=-1,
      cv=5)
search_func.fit(X, y)


print (search_func.best_params_)
print (search_func.best_score_)
```

After a while, depending on the machine you are running the optimization on, you will obtain the best combination based on cross-validated results

In summary, what is Grid Search

- In conclusion, grid search is a very simple optimization algorithm that can leverage the availability of multi-core computers.
- It can work fine with machine learning algorithms that do not require many tunings (such as SVM and the ridge and lasso regressions) but, in all other cases, its appli- cability is quite narrow.
- First, it is limited to optimizing hyperparameters by discrete choice (you need a limited set of values to cycle through).
- In addition, you cannot expect it to work effectively on algorithms requiring *multiple* hyperparameters to be tuned.
- This is because of the exploding complexity of the search space, and because most of the computational inefficiency is due to the fact that the search is trying parameter values blindly, most of which do not work for the problem

Random search (RS)

- Random search, which simply samples the search space randomly, is feasible in high-dimen- sional spaces and is widely used in practice.
- The downside of random search, however, is that it doesn’t use information from prior experiments to select the next setting (a problem shared by grid search, we should note).
- In addition, to find the best solution as fast as possible, you cannot do anything except hope to be lucky you catch the right hyperparameters.
- Random search works incredibly well and it is simple to understand.
- Despite the fact it relies on randomness, it isn’t just based on blind luck, though it may initially appear to be.

When to use RS?

- In fact, it works like random sampling in statistics: the main point of the technique is that if you do enough random tests, you have a good possibility of finding the right parameters without wasting energy on testing slightly different combinations of similarly performing combinations.
- Many AutoML systems rely on random search when there are too many parameters to set (see Golovin, D. et al. Google Vizier: A Service for Black-Box Optimization, 2017).
- As a rule of thumb, consider looking at random search when the dimensionality of your hyperparameter optimization problem is sufficiently high (for example, over 16).

RS example

- Below, we run the previous example using random search:

```
import scipy.stats as stats
from sklearn.utils.fixes import loguniform


search_dict = {'kernel': ['linear', 'rbf'],
      'C': loguniform(1, 1000),
      'gamma': loguniform(0.0001, 0.1)
      }
```

RS code

```
scorer = 'accuracy’
search_func = model_selection.RandomizedSearchCV(
      estimator=svc,param_distributions=search_dict, n_iter=6,
      scoring=scorer, n_jobs=-1, cv=5)

search_func.fit(X, y)

print (search_func.best_params_)
print (search_func.best_score_)
```

RS in summary

- Notice that, now, we don’t care about running the search on separate spaces for the different kernels.
- Contrary to grid search, where each parameter, even the ineffective ones, is systematically tested, which requires computational time, here the efficiency of the search is not affected by the set of hyperparameters tested.
- The search doesn’t depend on irrelevant parameters, but is guided by chance; any trial is useful, even if you are testing only one valid parameter among many for the chosen kernel.

Halving search

- As we mentioned, both grid search and random search work in an uninformed way: if some tests find out that certain hyperparameters do not impact the result or that certain value intervals are ineffective, the information is not propagated to the following searches.
- For this reason, Scikit-learn has recently introduced the HalvingGridSearchCV and HalvingRandomSearchCV estimators, which can be used to search a parameter space using successive halving applied to the grid search and random search tuning strategies.

Halving search

- In halving, a large number of hyperparameter combinations are evaluated in an initial round of tests but using a small amount of computational resources.
- This is achieved by running the tests on a subsample of a few cases from your training data.
- A smaller training set needs fewer computations to be tested, so fewer resources (namely time) are used at the cost of more imprecise performance estimations.
- This initial round allows the selection of a subset of candidate hyperparameter values, which have performed better on the problem, to be used for the second round, when the training set size is increased.

HS example

- The following rounds proceed in a similar way, allocating larger and larger subsets of the training set to be searched as the range of tested values is restricted (testing now requires more time to execute, but returns a more precise performance estimation), while the number of candidates continues to be halved.

HS code

```
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import HalvingRandomSearchCV

search_func = HalvingRandomSearchCV(estimator=svc,
       param_distributions=search_dict, resource='n_samples', max_resources=100,
       aggressive_elimination=True, scoring=scorer,
       n_jobs=-1, cv=5, random_state=0)
search_func.fit(X, y)

print (search_func.best_params_)
print (search_func.best_score_)
```

HS in summary

- In this way, halving provides information to the successive optimization steps via the selection of the candidates.
- In the next sections, we will discuss even smarter ways to achieve a more precise and efficient search through the space of hyperparameters.

Key parameters and how to use them

- The next problem is using the right set of hyperparameters for each kind of model you use.
- In particular, in order to be efficient in your optimization, you need to know the values of each hyperparameter that it actually makes sense to test for each distinct algorithm.
- In this section, we will examine the most common models used in Kaggle competitions, especially the tabular ones, and discuss the hyperparameters you need to tune in order to obtain the best results.
- We will distinguish between classical machine learning models and gradient boosting models (which are much more demanding in terms of their space of parameters) for generic tabular data problems.

Key parameters and how to use them

- As for neural networks, we can give you an idea about specific parameters to tune when we present the standard models (for instance, the TabNet neural model has some specific parameters to set so that it works properly).
- However, most of the optimization on deep neural networks in Kaggle competitions is not performed on standard models, but on custom ones.
- Consequently, apart from basic learning parameters such as the learning rate and the batch size, optimization in neural networks is based on the specific characteristics of the neural architecture of your model.
- You have to deal with the problem in an ad hoc way.
- Near the end of the chapter, we will discuss an example of neural architecture search (NAS) using **KerasTuner** (https://keras.io/keras_tuner/).

Linear models

- The linear models that need to be tuned are usually linear regressions or logistic regressions with regularization:
  a) C: The range you should search is np.logspace(-4, 4, 10); smaller values specify stron- ger regularization.
  b) alpha: You should search the range np.logspace(-2, 2, 10); smaller values specify stronger regularization, larger values specify stronger regularization. Also take note that higher values take more time to process when using lasso.
  c) l1_ratio: You should pick from the list [.1, .5, .7, .9, .95, .99, 1]; it applies only to elastic net.
- In Scikit-learn, depending on the algorithm, you find either the hyperparameter C (logistic regression) or alpha (lasso, ridge, elastic net).

Support vector machines

- SVMs are a family of powerful and advanced supervised learning techniques for classification and regression that can automatically fit linear and non-linear models.
- Scikit-learn offers an implementation based on LIBSVM, a complete library of SVM classification and regression im- plementations, and LIBLINEAR, a scalable library for linear classification ideal for large datasets, especially sparse text-based ones.
- In their optimization, SVMs strive to separate target classes in classification problems using a decision boundary characterized by the largest possible margin between classes.

Support vector machines parameters 1/2

- Though SVMs work fine with default parameters, they are often not optimal, and you need to test various value combinations using cross-validation to find the best ones. Listed according to their importance, you have to set the following parameters:
  a) C: The penalty value. Decreasing it makes the margin between classes larger, thus ignoring more noise but also making the model more generalizable. A best value can normally be found in the range np.logspace(-3, 3, 7).
  b) kernel: This parameter will determine how non-linearity will be implemented in an SVM and it can be set to 'linear', 'poly', 'rbf', 'sigmoid', or a custom kernel. The most commonly used value is certainly rbf.
  c) degree: Works with kernel='poly', signaling the dimensionality of the polynomial expansion. It is ignored by other kernels. Usually, setting its values to between 2 and 5 works the best.

Support vector machines parameters 2/2

d) gamma: A coefficient for 'rbf', 'poly', and 'sigmoid'. High values tend to fit data in a bet- ter way, but can lead to some overfitting. Intuitively, we can imagine gamma as the influence that a single example exercises over the model. Low values make the influence of each example reach further. Since many points have to be considered, the SVM curve will tend to take a shape less influenced by local points and the result will be a smoother decision contour curve. High values of gamma, instead, mean the curve takes into account how points are arranged locally more and, as a result, you get a more irregular and wiggly decision curve. The suggested grid search range for this hyperparameter is np.logspace(-3, 3, 7).

e) nu: For regression and classification with nuSVR and nuSVC, this parameter sets a toler-ance for the training points that are near to the margin and are not classified correctly. It helps in ignoring misclassified points just near or on the margin, hence it can render the classification decision curve smoother. It should be in the range [0,1] since it is a proportion relative to your training set. Ultimately, it acts like C, with high proportions enlarging the margin.

f) epsilon: This parameter specifies how much error SVR will accept, by defining an epsilon large range where no penalty is associated with an incorrect prediction of the example during the training of the algorithm. The suggested search range is np.logspace(-4, 2, 7).

g) penalty, loss, and dual: For LinearSVC, these parameters accept the ('l1', 'squared_ hinge', False), ('l2', 'hinge', True), ('l2', 'squared_hinge', True), and ('l2', 'squared_hinge', False) combinations. The ('l2', 'hinge', True) combination is analogous to the SVC(kernel='linear') learner.

SVM in summary

- It may appear that an SVM has many hyperparameters to set, but many settings are specific only to implementations or to kernels, so you only have to select the relevant parameters.

Random forests, extremely randomized trees

- Leo Breiman and Adele Cutler originally devised the idea at the core of the random forest algorithm, and the name of the algorithm remains a trademark of theirs today (though the algorithm is open source).
- Random forests are implemented in Scikit-learn as RandomForestClassifier or RandomForestRegressor.

Random forests, extremely randomized trees

- A random forest works in a similar way to bagging, also devised by Leo Breiman, but operates only using binary split decision trees, which are left to grow to their extremes.
- Moreover, it samples the cases to be used in each of its models using bootstrapping.
- As the tree is grown, at each split of a branch, the set of variables considered for the split is drawn randomly, too.

Why RF works?

- This is the secret at the heart of the algorithm: it ensembles trees that, due to different samples and variables considered at the splits, are very different from each other.
- As they are different, they are also uncorrelated.
- This is beneficial because when the results are ensembled, much variance is ruled out, as the extreme values on both sides of a distribution tend to balance out.
- In other words, bagging algorithms guarantee a certain level of diversity in the predictions, allowing them to develop rules that a single learner (such as a decision tree) might not come across.
- All this diversity is useful because it helps in building a distribution whose average is a better predictor than any of the individual trees in the ensemble.

Extra trees

- Extra Trees (also known as extremely randomized trees), represented in Scikit-learn by the ExtraTreesClassifier/ExtraTreesRegressor classes, are a more randomized kind of random forest that produces a lower variance in the estimates at the cost of greater bias of the estimators.
- However, when it comes to CPU efficiency, Extra Trees can deliver a considerable speed-up compared to random forests, so they can be ideal when you are working with large datasets in terms of both examples and features.
- The reason for the resulting higher bias but better speed is the way splits are built in an Extra Tree.
- Random forests, after drawing a random set of features to be considered for splitting a branch of a tree, carefully search among them for the best values to assign to each branch.
- By contrast, in Extra Trees, both the set of candidate features for the split and the actual split value are decided completely randomly.
- So, there's no need for much computation, though the randomly chosen split may not be the most effective one (hence the bias).

Parameters to optimize

- For both algorithms, the key hyperparameters that should be set are as follows:
  1) max_features: This is the number of sampled features that are present at every split, which can determine the performance of the algorithm. The lower the number, the speedier, but with higher bias.
  2) min_samples_leaf: This allows you to determine the depth of the trees. Large numbers diminish the variance and increase the bias.
  3) bootstrap: This is a Boolean that allows bootstrapping.
  4) n_estimators: This is the number of trees. Remember that the more trees the better, though there is a threshold beyond which we get diminishing returns depending on the data problem. Also, this comes at a computational cost that you have to take into account based on the resources you have available.

Extra trees vs Random forests?

- Extra Trees are a good alternative to random forests, especially when the data you have is particularly noisy.
- Since they trade some variance reduction for more bias given their random choice of splits, they tend to overfit less on important yet noisy features that would otherwise dominate the splits in a random forest.

Gradient boosting decision trees (GBDT)

- **Gradient boosting decision trees (GBDT)** is an improved version of boosting (boosting works by fitting a sequence of weak learners on reweighted versions of the data).
- Like AdaBoost, GBDT is based on a gradient descent function. The algorithm has proven to be one of the most proficient ones from the family of models that are based on ensembles, though it is characterized by an increased variance of estimates, more sensitivity to noise in data (both problems can be mitigated by using subsampling), and significant computational costs due to non-parallel operations.

GBDT is competitive!

- Apart from deep learning, gradient boosting is the most developed machine learning algorithm.
- Since AdaBoost and the initial gradient boosting implementation, as developed by Jerome Friedman, various other implementations of the algorithms appeared, the most recent ones being
  - LightGBM,
  - XGBoost,
  - and CatBoost.
- We will now review each of them

LightGBM

- The high-performance LightGBM algorithm (https://github.com/Microsoft/LightGBM) is capable of being distributed on multiple computers and handling large amounts of data quickly.
- It was developed by a team at Microsoft as an open-source project on GitHub (there is also an academic paper: https://papers.nips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html).
- LightGBM is based on decision trees, like XGBoost, but it follows a different strategy.
- While XGBoost uses decision trees to split on a variable and explore different tree splits at that variable (the **level-wise** tree growth strategy), LightGBM concentrates on one split and goes on splitting from there in order to achieve a better fit (the **leaf-wise** tree growth strategy).

LightGBM

- This allows LightGBM to quickly reach a good fit of the data, and to generate alternative solutions compared to XGBoost (which is good, if you expect to blend the two solutions together in order to reduce the variance of the estimates).
- Algorithmically speaking, if we think of the structure of splits operated by a decision tree as a graph, XGBoost pursues a *breadth-first* search (BFS) and LightGBM a *depth-first* search (DFS).
- Tuning LightGBM may appear daunting; it has more than a hundred parameters to tune that you can explore at this page: https://github.com/Microsoft/LightGBM/blob/master/docs/ Parameters.rst

Rule of thumb 1/

•As a rule of thumb, you should focus on the following hyperparameters, which usually have the most impact on the results:

- `n_estimators`: An integer between 10 and 10,000 that sets the number of iterations.
- `learning_rate`: A real number between 0.01 and 1.0, usually sampled from a log-uniform distribution. It represents the step size of the gradient descent procedure that computes the weights for the summed ensemble of all the iterations of the algorithm up to this point.
- `max_depth`: An integer between 1 and 16, representing the maximum number of splits on features. Setting it to a number below 0 allows the maximum possible number of splits, usually risking overfitting to data.
- `num_leaves`: An integer between 2 and 2^`max_depth`, representing the number of final leaves each tree will have at most.

Rule of thumb 2/

- `min_data_in_leaf`: An integer between 0 and 300 that determines the minimum number of data points in one leaf.
- `min_gain_to_split`: A float between 0 and 15; it sets the minimum gain of the algorithm for tree partitioning. By setting this parameter, you can avoid unnecessary tree splits and thus reduce overfitting (it corresponds to the `gamma` parameter in XGBoost).
- `max_bin`: An integer between 32 and 512 that sets the maximum number of bins that feature values will be bucketed into. Having this parameter larger than the default value of 255 implies more risk of producing overfitting results.
- `subsample`: A real number between 0.01, and 1.0, representing the portion of the sample to be used in training.
- `subsample_freq`: An integer between 0 and 10 specifying the frequency, in terms of iter- ations, at which the algorithm will subsample the examples. Note that, if set to zero, the algorithm will ignore any value given to the `subsample` parameter. In addition, it is set to zero by default, therefore just setting the `subsample` parameter won't work.

Rule of thumb 3/

- `feature_fraction`: A real number between 0.1 and 1.0 allowing you to specify the portion of features to be subsampled. Subsampling the features is another way to allow more randomization to play a role in the training, fighting noise and multicollinearity present in the features.
- `subsample_for_bin`: An integer between 30 and the number of examples. This sets the number of examples that are sampled for the construction of histogram bins.

Rule of thumb 4/

- `reg_lambda`: A real number between 0 and 100.0 that sets the L2 regularization. Since it is more sensitive to the scale than to the exact number of the parameter, it is usually sampled from a log-uniform distribution.
- `reg_alpha`: A real number between 0 and 100.0, usually sampled from a log-uniform distribution, which sets the L1 regularization.
- `scale_pos_weight`: A real number between 1e-6 and 500, better sampled from the log-uni- form distribution. The parameter weights the positive cases (thus effectively upsampling or downsampling) against the negative cases, which are kept to the value of 1.

- Although the number of hyperparameters to tune when using LightGBM may appear daunt- ing, in reality only a few of them matter a lot. Given a fixed number of iterations and learn- ing rate, just a few are the most impactful (`feature_fraction`, `num_leaves`, `subsample`, `reg_ lambda`, `reg_alpha`, `min_data_in_leaf`), as explained in this blog article by *Kohei Ozaki*, a Kaggle Grandmaster: https://medium.com/optuna/lightgbm-tuner-new-optuna-integration-for-hyperparameter-optimization-8b7095e99258. Kohei Ozaki leverages this fact in order to create a fast-tuning procedure for Optuna (you'll find more on the Optuna optimizer at the end of this chapter).

XGBoost

•XGBoost (https://github.com/dmlc/XGBoost) stands for **eXtreme Gradient Boosting**. It is an open-source project that is not part of Scikit-learn, though it has recently been expanded by a Scikit-learn wrapper interface that makes it easier to incorporate XGBoost into a Scikit-learn- style data pipeline.

•The XGBoost algorithm gained momentum and popularity in 2015 data science competitions, such as those on Kaggle and the KDD Cup 2015. As the creators (*Tianqui Chen*, *Tong He*, and *Carlos Guestrin*) report in papers they wrote on the algorithm, out of 29 challenges held on Kaggle during 2015, 17 winning solutions used XGBoost as a standalone solution or as part of an ensemble of multiple different models.

XGBoost

- Since then, the algorithm has always retained a strong appeal among the community of data scientists, though it struggled to keep pace with the innovation brought about by other GBM implementations such as LightGBM and CatBoost.
- Aside from good performance both in terms of accuracy and computational efficiency, XGBoost is also a *scalable* solution, using at best multi-core processors as well as distributed machines.

•XGBoost represents a new generation of GBM algorithms thanks to important tweaks to the initial tree boost GBM algorithm:

- Sparsity-awareness; it can leverage sparse matrices, saving both memory (no need for dense matrices) and computation time (zero values are handled in a special way).
- Approximate tree learning (weighted quantile sketch), which produces similar results but in much less time compared to the classical complete explorations of possible branch cuts.
- Parallel computing on a single machine (using multi-threading during the search for the best split) and, similarly, distributed computations on multiple machines.
- Out-of-core computations on a single machine, leveraging a data storage solution called **column block**. This arranges data on a disk by columns, thus saving time by pulling data from the disk in the way the optimization algorithm (which works on column vectors) expects it.

Mussing values

- XGBoost can also deal with missing data in an effective way. Other tree ensembles based on standard decision trees require missing data first to be imputed using an off-scale value, such as a negative number, in order to develop an appropriate branching of the tree to deal with missing values.

Parameters to tune 1/

•As for XGBoost's parameters (https://xgboost.readthedocs.io/en/latest/parameter.html), we have decided to highlight a few key ones you will find across competitions and projects:

- `n_estimators`: Usually an integer ranging from 10 to 5,000.
- `learning_rate`: A real number ranging from 0.01 to 1.0, better sampled from the log-uni- form distribution.
- `min_child_weight`: Usually an integer between 1 and 10.
- `max_depth`: Usually an integer between 1 and 50.
- `max_delta_step`: Usually an integer sampled between 0 and 20, representing the maxi- mum delta step we allow for each leaf output.
- `subsample`: A real number from 0.1 to 1.0 indicating the proportion of examples to be subsampled.
- `colsample_bytree`: A real number from 0.1 to 1.0 indicating the subsample ratio of col- umns by tree.
- `colsample_bylevel`: A real number from 0.1 to 1.0 indicating the subsample ratio by level in trees.

Parameters to tune 2/

- `reg_lambda`: A real number between 1e-9 and 100.0, preferably sampled from the log-uniform distribution. This parameter controls the L2 regularization.
- `reg_alpha`: A real number between 1e-9 and 100.0, preferably sampled from the log-uniform distribution. This parameter controls the L1 regularization.
- `gamma`: Specifying the minimum loss reduction for tree partitioning, this parameter requires a real number between 1e-9 and 0.5, preferably sampled from the log-uniform distribution.
- `scale_pos_weight`: A real number between 1e-6 and 500.0, preferably sampled from the log-uniform distribution, which represents a weight for the positive class.

- Like LightGBM, XGBoost also has many similar hyperparameters to tune, hence all of the con- siderations previously made for LightGBM are also valid for XGBoost.

CatBoost

- In July 2017, Yandex, the Russian search engine, made another interesting GBM algorithm public, CatBoost (https://catboost.ai/), whose name comes from putting together the two words “Category” and “Boosting.”
- In fact, its strong point is its ability to handle categorical variables, which make up most of the information in most relational databases, by adopting a mixed strategy of one-hot encoding and target encoding. Target encoding is a way to express categorical levels by assigning them an appropriate numeric value for the problem at hand

What CatBoost is good at

- The idea used by CatBoost to encode categorical variables is not new, but it is a kind of feature engineering that has been used before, mostly in data science competitions. T
- arget encoding, also known as likelihood encoding, impact coding, or mean encoding, is simply a way to trans- form your labels into a number based on their association with the target variable. I
- f you have a regression, you could transform labels based on the mean target value typical of that level; if it is a classification, it is simply the probability of classification of your target given that label (the probability of your target conditional on each category value).
- It may appear a simple and smart feature engineering trick but it has side effects, mostly in terms of overfitting, because you are taking information from the target into your predictors.

Parameters

•CatBoost has quite a few parameters (see https://catboost.ai/en/docs/references/training- parameters/). We have limited our discussion to the eight most important ones:

- `iterations`: Usually an integer between 10 and 1,000, but it can increase based on the problem.
- `depth`: An integer between 1 and 8; usually higher values require longer fitting times and do not produce better results.
- `learning_rate`: A real value between 0.01 and 1.0, better sampled from the log-uniform distribution.
- `random_strength`: A real number log-linearly sampled from the range 1e-9 to 10.0, which specifies the randomness level for scoring splits.
- `bagging_temperature`: A real value between 0.0 and 1.0 that sets the Bayesian bootstrap.
- `border_count`: An integer between 1 and 255 indicating the splits for numerical features.
- `l2_leaf_reg`: An integer between 2 and 30; the value for L2 regularization.
- `scale_pos_weight`: A real number between 0.01 and 10.0 representing the weight for the positive class.

In conclusion

- Even if CatBoost may appear to be just another GBM implementation, it has quite a few differ- ences (highlighted also by the different parameters being used) that may provide great help in a competition, both as a single-model solution and as a model integrated into a larger ensemble

HistGradientBoosting

- Recently, Scikit-learn has introduced a new version of gradient boosting inspired by LightGBM's binned data and histograms (see this presentation at EuroPython by *Olivier Grisel*: https://www.youtube.com/watch?v=urVUlKbQfQ4). Either as a classifier (`HistGradientBoostingClassifi er`) or a regressor (`HistGradientBoostingRegressor`), it can be used for enriching ensembles with different models and it presents a much shorter and essential range of hyperparameters to be tuned:

HistGradientBoosting parameters

- `learning_rate`: A real number between 0.01 and 1.0, usually sampled from a log-uniform distribution.
- `max_iter`: An integer that can range from 10 to 10,000.
- `max_leaf_nodes`: An integer from 2 to 500. It interacts with `max_depth`; it is advisable to set only one of the two and leave the other set to `None`.
- `max_depth`: An integer between 2 and 12.
- `min_samples_leaf`: An integer between 2 and 300.
- `l2_regularization`: A float between 0.0 and 100.0.
- `max_bins`: An integer between 32 and 512.

•Even if Scikit-learn's `HistGradientBoosting` is nothing too different from LightGBM or XG- Boost, it does provide a different way to implement GBMs in a competition, and models built by `HistGradientBoosting` may provide a contribution when ensembling multiple predictions, such as in blending and stacking.

Bayesian optimization

- Having reached the end of this section, you should be more familiar with the most common machine learning algorithms (only deep learning solutions have not been discussed) and their most important hyperparameters to tune, which will help you in building an outstanding solution in a Kaggle competition. K
- nowing the basic optimization strategies, usable algorithms, and their key hyperparameters is just a starting point. In the next section, we will begin an in-depth discussion about how to tune them more optimally using Bayesian optimization

Bayesian optimization

•Leaving behind grid search (feasible only when the space of experiments is limited), the usual choice for the practitioner is to apply random search optimization or try a **Bayesian optimization** (**BO**) technique, which requires a more complex setup.

•Originally introduced in the paper *Practical Bayesian optimization of machine learning algorithms* by Snoek, J., Larochelle, H., and Adams, R. P. (http://export.arxiv.org/pdf/1206.2944), the key idea behind Bayesian optimization is that we optimize a **proxy function** (also called a **sur- rogate function**) rather than the true objective function (which grid search and random search both do).

•We do this if there are no gradients, if testing the true objective function is costly (if it is not, then we simply go for random search), and if the search space is noisy and complex enough.

Intuition

- Bayesian search balances *exploration* with *exploitation*.
- At the start, it explores randomly, thus training the surrogate function as it goes.
- Based on that surrogate function, the search exploits its initial approximate knowledge of how the predictor works in order to sample more useful examples and minimize the cost function.
- As the *Bayesian* part of the name suggests, we are using priors in order to make smarter decisions about sampling during optimization.
- This way, we reach a minimization more quickly by limiting the number of evaluations we need to make.

•Bayesian optimization uses an **acquisition function** to tell us how promising an observation will be.

•In fact, to manage the tradeoff between exploration and exploitation, the algorithm defines an acquisition function that provides a single measure of how useful it would be to try any given point.

•Usually, Bayesian optimization is powered by Gaussian processes. Gaussian processes perform better when the search space has a smooth and predictable response.

• An alternative when the search space is more complex is using tree algorithms (for instance, random forests), or a completely different approach called **Tree Parzen Estimators** or **Tree-structured Parzen Estimators** (**TPEs**).

•Instead of directly building a model that estimates the success of a set of parameters, thus acting like an oracle, TPEs estimate the parameters of a multivariate distribution that define the best-performing values of the parameters, based on successive approximations provided by the experimentations.

•In this way, TPEs derive the best set of parameters by sampling them from a probabilistic distribution, and not directly from a machine learning model like Gaussian processes does.

•We will discuss each of these approaches, first by examining Scikit-optimize and KerasTuner, both based on Gaussian processes (Scikit-optimize can also use random forests and KerasTuner can use multi-armed bandits), and then Optuna, which is principally based on TPE (though it also offers different strategies: https://optuna.readthedocs.io/en/stable/reference/samplers.html)

To remember

- Though Bayesian optimization is considered the state of the art for hyperparameter tuning, always keep in mind that for more complex parameter spaces, using Bayesian optimization provides no advantage in terms of time and computation spent over a solution simply found by random search.
- For instance, in Google Cloud Machine Learning Engine services, the usage of Bayesian optimization is limited to problems involving at most sixteen parameters.
- For larger numbers of parameters, it resorts to random sampling.

Using Scikit-optimize

- Scikit-optimize (`skopt`) has been developed using the same API as Scikit-learn, as well as making extensive use of NumPy and SciPy functions. In addition, it was created by some of the contributors to the Scikit-learn project, such as *Gilles Louppe*.
- Based on Gaussian process algorithms, the package is well maintained, though sometimes it has to catch up because of improvements on the Scikit-learn, NumPy, or SciPy sides.
- For instance, at the time of writing, in order to run it properly on Kaggle Notebooks you have to roll back to older versions of these packages, as explained in a GitHub issue (https://github.com/scikit-optimize/scikit-optimize/issues/981).

- The package has an intuitive API and it is quite easy to hack it and use its functions in custom op- timization strategies. Scikit-optimize is also renowned for its useful graphical representations. In fact, by visualizing the results of an optimization process (using Scikit-optimize's `plot_objective` function), you can figure out whether you can re-define the search space for the problem and formulate an explanation of how optimization works for a problem.

Tutorial

- In our worked example, we will refer to the work that can be found in the following Kaggle Notebooks:
  - https://www.kaggle.com/lucamassaron/tutorial-bayesian-optimization-with- lightgbm
  - https://www.kaggle.com/lucamassaron/scikit-optimize-for-lightgbm
- Our purpose here is to show you how to quickly handle an optimization problem for a competition such as *30 Days of ML*, a recent competition that involved many Kagglers in learning new skills and applying them in a competition lasting 30 days.
- The goal of this competition is to predict the value of an insurance claim, so it is a regression problem.
- You can find out more about this initiative and download the data necessary for the example we are going to present (materials are always available to the public), by visiting https://www.kaggle.com/thirty-days-of-ml

Code 1/

- The following code will present how to load the data for this problem and then set up a Bayesian optimization process that will improve the performance of a LightGBM model.
- We start by loading the packages:

```
# Importing core libraries
import numpy as np import pandas as pd from time import time import pprint
import joblib
from functools import partial


# Suppressing warnings because of skopt verbosity
import warnings warnings.filterwarnings("ignore")


# Classifiers
import lightgbm as lgb
```

Code 2/

```
from sklearn.model_selection import KFold

# Metrics
from sklearn.metrics import mean_squared_error from
sklearn.metrics import make_scorer

# Skopt functions
from skopt import BayesSearchCV
from skopt.callbacks import DeadlineStopper, DeltaYStopper
from skopt.space import Real, Categorical, Integer
```

Code 3/

•As a next step, we load the data. The data doesn't need much processing, aside from turning some categorical features with alphabetical letters as levels into ordered numeric ones:

```
# Loading data

X = pd.read_csv("../input/30-days-of-ml/train.csv") X_test = pd.read_csv("../input/30-days-
of-ml/test.csv")


# Preparing data as a tabular matrix
y = X.target

X = X.set_index('id').drop('target', axis='columns') X_test = X_test.set_index('id')


# Dealing with categorical data

categoricals = [item for item in X.columns if 'cat' in item] cat_values =
np.unique(X[categoricals].values)
cat_dict = dict(zip(cat_values, range(len(cat_values)))) X[categoricals] =
X[categoricals].replace(cat_dict).astype('category') X_test[categoricals] =
X_test[categoricals].replace(cat_dict). astype('category')
```

Code 4/

- After making the data available, we define a reporting function that can be used by Scikit-optimize for various optimization tasks. The function takes the data and the optimizer as inputs. It can also handle **callback functions**, which are functions that perform actions such as reporting, early stopping based on having reached a certain threshold of time spent searching or performance not improving (for instance, not seeing improvements for a certain number of iterations), or saving the state of the processing after each optimization iteration:

Code 5/

```
# Reporting util for different optimizers
  def report_perf(optimizer, X, y, title="model", callbacks=None): """
      A wrapper for measuring time and performance of optimizers optimizer = a
      sklearn or a skopt optimizer
      X = the training set y = our target
      title = a string label for the experiment """
      start = time()

      if callbacks is not None:
        optimizer.fit(X, y, callback=callbacks) else:
        optimizer.fit(X, y)
```

Code 6/

```
d=pd.DataFrame(optimizer.cv_results_)

best_score = optimizer.best_score_

best_score_std = d.iloc[optimizer.best_index_].std_test_score

best_params = optimizer.best_params_

print((title + " took %.2f seconds, candidates checked: %d, best CV score:
%.3f" + u" \u00B1"+" %.3f") % (time() - start,
len(optimizer.cv_results_['params']), best_score, best_score_std))
print('Best parameters:’)

pprint.pprint(best_params)
print()
return best_params
```

Code logic

- We now have to prepare the scoring function (upon which the evaluation is based), the validation strategy (based on cross-validation), the model, and the search space. For the scoring function, which should be a root mean squared error metric, we refer to the practices in Scikit-learn where you always minimize a function (if you have to maximize, you minimize its negative).
- The `make_scorer` wrapper can easily replicate such practices:

Code 7/

```
# Setting the scoring function
scoring = make_scorer(partial(mean_squared_error, squared=False),
          greater_is_better=False)


# Setting the validation strategy
kf = KFold(n_splits=5, shuffle=True, random_state=0)


# Setting the basic regressor
reg = lgb.LGBMRegressor(boosting_type='gbdt',
            metric='rmse', objective='regression', n_jobs=1,
            verbose=-1, random_state=0)
```

Code logic

- Setting the search space requires the use of different functions from Scikit-optimize, such as `Real`, `Integer`, or `Choice`, each one sampling from a different kind of distribution that you define as a parameter (usually the uniform distribution, but the log-uniform is also used when you are more interested in the scale effect of a parameter than its exact value):

Code 8/ Search space

```
# Setting the search space
search_spaces = {

    # Boosting learning rate
    'learning_rate': Real(0.01, 1.0, 'log-uniform'),

    # Number of boosted trees to fit
    'n_estimators': Integer(30, 5000),

    # Maximum tree leaves for base learners
    'num_leaves': Integer(2, 512),

    # Maximum tree depth for base learners
    'max_depth': Integer(-1, 256),

    # Minimal number of data in one leaf
    'min_child_samples': Integer(1, 256),
```

Code 9/

```
 'subsample': Real(0.01, 1.0, 'uniform'),

 # Frequency of subsample
 'subsample_freq': Integer(0, 10),

 # Subsample ratio of columns
 'colsample_bytree': Real(0.01, 1.0, 'uniform'),

 # Minimum sum of instance weight
 'min_child_weight': Real(0.01, 10.0, 'uniform'),

 # L2 regularization
 'reg_lambda': Real(1e-9, 100.0, 'log-uniform'),

 # L1 regularization
 'reg_alpha': Real(1e-9, 100.0, 'log-uniform'),
}
```

Code logic

- Once you have defined:
  - Your cross-validation strategy
  - Your evaluation metric
  - Your base model
  - Your hyperparameter search space

- All that is left is just to feed them into your optimization function, `BayesSearchCV`. Based on the CV scheme provided, this function will look for the minimum of your scoring function based on values within the search space. You can set a maximum number of iterations performed, the kind of surrogate function (Gaussian processes (`GP`) works on most occasions), and the random seed for reproducibility:

Code 10/

```
# Wrapping everything up into the Bayesian optimizer
opt = BayesSearchCV(estimator=reg,
        search_spaces=search_spaces,
        scoring=scoring, cv=kf, n_iter=60,
        n_jobs=-1, iid=False,
        # if not iid it optimizes on the cv score
        return_train_score=False, refit=False,
        # Gaussian Processes (GP)
        optimizer_kwargs={'base_estimator': 'GP'},
        # random state for replicability
        random_state=0)
```

Code logic

- At this point, you can start the search using the reporting function we defined previously. After a while, the function will return the best parameters for the problem.

```
# Running the optimizer
overdone_control = DeltaYStopper(delta=0.0001)
# We stop if the gain of the optimization becomes too small
time_limit_control = DeadlineStopper(total_time=60 * 60 * 6)
# We impose a time limit (6 hours)


best_params = report_perf(opt, X, y,'LightGBM_regression',
          callbacks=[overdone_control, time_limit_
control])
```

Code conclusion

- In the example, we set a limit on operations by specifying a maximum time allowed (6 hours) before stopping and reporting the best results.
- Since the Bayesian optimization approach blends together exploration and exploitation of different combinations of hyperparameters, stopping at any time will always return the best solution found so far (but not necessarily the best one possible).
- This is because the acquisition function will always give priority of exploration to the most promising parts of the search space, based on the estimated performances returned by the surrogate function and their uncertainty intervals.

Customizing a Bayesian optimization search

- The `BayesSearchCV` function offered by Scikit-optimize is certainly convenient, because it wraps and arranges all the elements of a hyperparameter search by itself, but it also has limitations. For instance, you may find it useful in a competition to:

- Have more control over each search iteration, for instance mixing random search and Bayesian search
- Be able to apply early stopping on algorithms
- Customize your validation strategy more
- Stop experiments that do not work early (for instance, immediately evaluating the per- formance of the single cross-validation folds when it is available, instead of waiting to have all folds averaged at the end)
- Create clusters of hyperparameter sets that perform in a similar way (for instance, in order to create multiple models differing only in the hyperparameters used, to be used for a blending ensemble)

•Each of these tasks would not be too complex if you could modify the `BayesSearchCV` internal procedure. Luckily, Scikit-optimize lets you do just this. In fact, behind `BayesSearchCV`, as well as behind other wrappers from the package, there are specific minimizing functions that you can use as standalone parts of your own search function:

- `gp_minimize`: Bayesian optimization using Gaussian processes
- `forest_minimize`: Bayesian optimization using random forests or extremely randomized
- trees
- `gbrt_minimize`: Bayesian optimization using gradient boosting
- `dummy_minimize`: Just random search

•In the following example, we are going to modify the previous search using our own custom search function. The new custom function will accept early stopping during training and it will prune experiments if one of the fold validation results is not a top-performing one.

Example

- You can find the next example working in a Kaggle Notebook at https://www. kaggle.com/lucamassaron/hacking-bayesian-optimization.

Bayesian optimization for neural networks

- Moving on to deep learning, neural networks also seem to have quite a few hyperparameters to fix:
  - Batch size
  - Learning rate
  - The kind of optimizer and its internal parameters

•All these parameters influence how the network learns and they can make a big impact; just a slight difference in batch size or learning rate can determine whether a network can reduce its error beyond a certain threshold or not.

•That being said, these learning parameters are not the only ones that you can optimize when working with **deep neural networks** (**DNNs**). How the network is organized in layers and the details of its architecture can make even more of a difference.

- In fact, technically speaking, an **architecture** implies the representational capacity of the deep neural network, which means that, depending on the layers you use, the network will either be able to read and process all the information available in the data, or it will not.
- While you had a large but limited set of choices with other machine learning algorithms, with DNNs your choices seem unlimited, because the only apparent limit is your knowledge and experience in handling parts of neural networks and putting them together.

•Common best practices for great deep learning practitioners when assembling well-performing DNNs depend mainly on:

- Relying on pre-trained models (so you have to be very knowledgeable about the solutions available, such as those found on Hugging Face (https://huggingface.co/models) or on GitHub)
- Reading cutting-edge papers
- Copying top Kaggle Notebooks from the same competition or previous ones
- Trial and error
- Ingenuity and luck

- In a famous lesson given by *Professor Geoffrey Hinton*, he states that you can achieve similar and often better results using automated methods such as Bayesian optimization. Bayesian optimi- zation will also avoid you getting stuck because you cannot figure out the best combinations of hyperparameters among the many possible ones.
- For the slides, see https://www.cs.toronto.edu/~hinton/coursera/lecture16/ lec16.pdf.

AutoML

- As we mentioned before, even in most sophisticated AutoML systems, when you have too many hyperparameters, relying on random optimization may produce better results or the same results in the same amount of time as Bayesian optimization. In addition, in this case, you also have to fight against an optimization landscape with sharp turns and surfaces; in DNN optimization, many of your parameters won't be continuous but Boolean instead, and just one change could unexpectedly transform the performance of your network for the better or for the worse

•Our experience tells us that random optimization may not be suitable for a Kaggle competition because:

- You have limited time and resources
- You can leverage your previous optimization results in order to find better solutions

- Bayesian optimization in this scenario is ideal: you can set it to work based on the time and computational resources that you have and do it by stages, refining your settings through multiple sessions.
- Moreover, it is unlikely that you will easily be able to leverage parallelism for tuning DNNs, since they use GPUs, unless you have multiple very powerful machines at hand.
- By working sequentially, Bayesian optimization just needs one good machine to perform the task.
- Finally, even if it is hard to find optimal architectures by a search, due to the optimization landscape you leverage information from previous experiments, especially at the beginning, totally avoiding combinations of parameters that won't work.
- With random optimization, unless you change the search space along the way, all combinations are always liable to be tested.

•There are also drawbacks, however. Bayesian optimization models the hyperparameter space using a surrogate function built from previous trials, which is not an error-free process.

•It is not a remote possibility that the process ends up concentrating only on a part of the search space while ignoring other parts (which may instead contain the minimum you are looking for).

• The solution to this is to run a large number of experiments to be safe, or to alternate between random search and Bayesian optimization, challenging the Bayesian model with random trials that can force it to reshape its search model in a more optimal way.

•For our example, we use again the data from the *30 Days of ML* initiative by Kaggle, a regression task. Our example is based on TensorFlow, but with small modifications it can run on other deep learning frameworks such as PyTorch or MXNet.

Example

- As before, you can find the example on Kaggle here: https://www.kaggle.com/ lucamassaron/hacking-bayesian-optimization-for-dnns.

Creating lighter and faster models with KerasTuner

- If the previous section has puzzled you because of its complexity, KerasTuner can offer you a fast solution for setting up an optimization without much hassle. Though it uses Bayesian optimization and Gaussian processes by default, the new idea behind KerasTuner is **hyperband optimization**. Hyperband optimization uses the bandit approach to figure out the best parameters (see http://web.eecs.umich.edu/~mosharaf/Readings/HyperBand.pdf). This works quite well with neu- ral networks, whose optimization landscape is quite irregular and discontinuous, and thus not always suitable for Gaussian processes.

- Let’s start from the beginning. KerasTuner (https://keras.io/keras_tuner/) was announced as a “flexible and efficient hyperparameter tuning for Keras models” by *François Chollet*, the creator of Keras.

Recipe

•The recipe proposed by Chollet for running KerasTuner is made up of simple steps, starting from your existing Keras model:

1. Wrap your model in a function with `hp` as the first parameter.
2. Define hyperparameters at the beginning of the function.
3. Replace DNN static values with hyperparameters.
4. Write the code that models a complex neural network from the given hyperparameters.
5. If necessary, dynamically define hyperparameters as you build the network.

How does this work?

- We’ll now explore how all these steps can work for you in a Kaggle competition by using an ex- ample. At the moment, KerasTuner is part of the stack offered by any Kaggle Notebook, hence you don’t need to install it. In addition, the TensorFlow add-ons are part of the Notebook’s pre-in- stalled packages.

Example

- You can find this example already set up on a Kaggle Notebook here: https://www. kaggle.com/lucamassaron/kerastuner-for-imdb/.

Examples

•If you would like to examine more examples of using KerasTuner, François Chollet also created a series of Notebooks for Kaggle competitions in order to showcase the workings and functionalities of his optimizer:

- https://www.kaggle.com/fchollet/keras-kerastuner-best-practices for the *Digit Recognizer* datasets
- https://www.kaggle.com/fchollet/titanic-keras-kerastuner- best-practices for the *Titanic* dataset
- https://www.kaggle.com/fchollet/moa-keras-kerastuner-best-practices for the *Mechanisms of Action (MoA) Prediction* competition

The TPE approach in Optuna

- We complete our overview of Bayesian optimization with another interesting tool and approach to it. As we have discussed, Scikit-optimize uses Gaussian processes (as well as tree algorithms) and it directly models the surrogate function and the acquisition function.

TPE optimization

•Instead, optimizers based on **TPE** tackle the problem by estimating the likelihood of success of the values of parameters. In other words, they model the success distribution of the parameters themselves using successive refinements, assigning a higher probability to more successful value combinations.

•In this approach, the set of hyperparameters is divided into good and bad ones by these distri- butions, which take the role of the surrogate and acquisition functions in Bayesian optimization, since the distributions tell you where to sample to get better performances or explore where there is uncertainty.

- To explore the technical details of TPE, we suggest reading Bergstra, J. et al. *Algorithms for hyper-parameter optimization*. Advances in neural information processing systems 24, 2011 (https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf).

Hyperopt

•Therefore, TPE can model the search space and simultaneously suggest what the algorithm can try next, by sampling from the adjusted probability distribution of parameters.

•For a long time, **Hyperopt** was the option for those preferring to use TPE instead of Bayesian op- timization based on Gaussian processes. In October 2018, however, Optuna appeared in the open source and it has become the preferred choice for Kagglers due to its versatility (it also works out of the box for neural networks and even for ensembling), speed, and efficiency in finding better solutions compared to previous optimizers.

- In this section, we will demonstrate just how easy is to set up a search, which is called a *study* under Optuna terminology. All you need to do is to write an objective function that takes as input the parameters to be tested by Optuna and then returns an evaluation. Validation and other al- gorithmic aspects can be handled in a straightforward manner inside the objective function, also using references to variables external to the function itself (both global variables or local ones). Optuna also allows **pruning**, that is, signaling that a particular experiment is not going well and that Optuna can stop and forget about it. Optuna provides a list of functions that activate this callback (see https://optuna.readthedocs.io/en/stable/reference/integration.html); the algorithm will run everything efficiently for you after that, which will significantly reduce the time needed for optimization.

Example

- You can find the Notebook for this example at https://www.kaggle.com/ lucamassaron/optuna-bayesian-optimization.

Summary 1/

•In this chapter, we discussed hyperparameter optimization at length as a way to increase your model's performance and score higher on the leaderboard. We started by explaining the code functionalities of Scikit-learn, such as grid search and random search, as well as the newer halv- ing algorithms.

•Then, we progressed to Bayesian optimization and explored Scikit-optimize, KerasTuner, and finally Optuna. We spent more time discussing the direct modeling of the surrogate function by Gaussian processes and how to hack it, because it can allow you greater intuition and a more ad hoc solution. We recognize that, at the moment, Optuna has become a gold standard among Kagglers, for tabular competitions as well as for deep neural network ones, because of its speedier convergence to optimal parameters in the time allowed in a Kaggle Notebook.

Summary 2/

•However, if you want to stand out among the competition, you should strive to test solutions from other optimizers as well.

• In the next chapter, we will move on to discuss another way to improve your performance in Kaggle competitions: **ensembling models**. By discovering the workings of averaging, blending, and stacking, we will illustrate how you can boost your results beyond what you can obtain by tuning hyperparameters alone.