Random Search vs Grid Search for hyperparameter optimization

Guilherme Caponetto

Nov 14, 2019 · 9 min read ★

Grid Search is a search technique that has been widely used in many machine learning researches when it comes to hyperparameter optimization. Among other approaches to explore a search space, an interesting alternative is to rely on randomness by using the Random Search technique.

Photo by Andrew Ridley on Unsplash

Introduction

The ideal problem that machine learning researchers would like to work on is the one admitting a convex objective function with no hyperparameters needed nor relaxations, yet powerful enough to provide the tiniest error on unseen data. However, hyperparameters are a necessary evil that helps to lead the optimization task to a region in the search space that would generalize the most for the data. An example of hyperparameter that you might have heard of is the learning rate in neural networks.

Diving a little bit deeper, take regularized linear models as an example. A regularization term is incorporated into the objective function in order to enforce a determined penalty. Two well-known approaches are the imposition of the $\ell 1$-norm and $\ell 2$-norm on the parameter vectors to perform feature selection (Lasso) and control the complexity of the model (Ridge), respectively.

However, introducing penalties are not sufficient by themselves to leverage from the full benefit of regularization. Their influence needs to be optimized with a controlled parameter, which is called hyperparameter since it needs to be set before the training process. And for the examples that I've previously mentioned, only one hyperparameter is necessary; usually, you would have to deal with more than one of them.

Besides manually searching for good candidate values for hyperparameters, the most basic and straightforward approach for optimizing hyperparameters is the Grid Search (GS) technique. Basically, a list of candidate values for each hyperparameter is defined and evaluated. The name "grid" comes to the fact that all possible candidates within all needed hyperparameters are combined in a sort of grid. The combination yielding the best performance, preferably evaluated in a validation set, is then selected.

As an example, suppose that α and β are hyperparameters that will be optimized using GS. Based on some hypothetical knowledge, we can guess that the candidates could be [1, 2, 3] and [20, 60, 80] for α and β, respectively. Thus, we can set up a grid of the values and their combinations in the following way:

From the grid we've just built, each combination is evaluated and the one yielding the best performance is selected. After this process, perhaps we could find out that (3, 60) was the best option for our problem. On the other hand, the global minimum could be located at (2.57, 58).

This task, however, starts to become very time-consuming if there are many hyperparameters and the search space is huge, not to mention that a list of candidates must be provided *a priori*. As an alternative to GS, one could rely on randomness through the Random Search (RS) technique.

In the literature, Bergstra & Bengio [1] showed in their paper that RS is more interesting than GS in the case of several learning algorithms on several data sets. Take the time to read the paper if you haven't, it is really interesting and has many more details to grasp.

Although RS can be easily understood and implemented, it could be employed by using this python library. Besides, this library offers other capabilities that we are not covering in this post, like running in parallel and providing other related algorithms. Give it a try.

In order to apply RS, the following components are necessary to be specified beforehand: *(i)* function to measure how good the candidate set are; *(ii)* search space; *(iii)* number of trials.

The selected candidates are the ones that evaluate the best output in the objective function within the number of trials. Notice that, the candidate set will be selected at runtime, *i.e.* it is not necessary to be specified *a priori* like required by GS.

Finally, I cannot fail to mention that there are other more advanced approaches for hyperparameter optimization, such as Bayesian optimization. Go check them out if the next step of your machine learning pipeline is hyperparameter optimization, but give GS and RS a try first, maybe you are good to go with them. :-)

or implement the code on your own. Keep in mind that some code snippets use code implemented in previous snippets, therefore the order of occurrence matters.

> All mentioned files in this post are available in my GitHub. Check this repo out!

Setting up the environment

Assuming you are familiar with Anaconda, I've prepared a *yml* file so you can quickly set up the environment just like mine, even the same library version. Here's the content of the file.

```
name: grid-vs-random-search
channels:
 – conda-forge
dependencies:
 – python=3.6.9
 – numpy=1.16.5
 – matplotlib=3.1.1
 – jupyter=1.0.0
 – ipython=7.8.0
 – hyperopt=0.1.2
```

Once you have the file, execute the following command to create the environment as well as install all necessary dependencies.

```
$ conda env create -f environment.yml
```

Activate the environment after the installation is completed.

```
$ conda activate grid-vs-random-search
```

All set! Let's dive into some python code.

Configuring plotting

```
%matplotlib inline
```


Configuring plotting

Defining the cost function

Let's define a very weird non-convex function to act as our cost function. Here's what I've picked:

$$f(x) = 675 + (x-15)^2 + x^2 \cos(x\pi)$$

We can then define a lambda for that, though regular functions would be also possible.

```
import numpy as np

non_convex_function = lambda x: 675 + (x-15)**2 + x**2 * np.cos(x*np.pi)
```


Defining the cost function

Plotting the cost function

Curious about the shape of the function that I've picked? Here's the code to plot it very nicely.

```
def setup_ax():
    """
    Set up the plot axis to look prettier.
    """
    ax = plt.gca()
    ax.set_axisbelow(True)
    ax.grid(True, color='lightgrey', linestyle='-', alpha=0.4)
    ax.tick_params(axis='both', which='both', length=0, labelcolor='0.5')
    ax.spines['right'].set_visible(False)
    ax.spines['top'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
```

```
    :param x_min: left side of the interval
    :param x_max: right side of the interval
    """
    plt.figure(facecolor="white", figsize=(16, 6))

    setup_ax()

    x = np.arange(x_min, x_max, 0.01)

    plt.plot(x,
             f(x),
             c='r',
             zorder=0,
             label='function')

    plt.legend(loc='upper left')


# Intervals for the experiments [X_MIN, X_MAX]
X_MIN = -10
X_MAX = 30

plot_function(non_convex_function, X_MIN, X_MAX)
```

grid-vs-random-search-part3.py hosted with ❤ by GitHub view raw

Plotting the cost function

And now I present our cost function. You can see that there are many local minima to make it harder for algorithms to find the global minimum, but a good thing is that we can visualize it. Usually, you would have to deal with functions that are not very clear and, worse, in multidimensional space. That's why we need good search algorithms. Notice that, the x-axis represents the hyperparameter candidates and the y-axis represents the cost, so the closer to zero the better.

Our weird cost function

Defining some helper code

We'll need some more helper code. Here's how I've implemented them — I've put some comments to make it easier to follow. Take your time to understand what will be done here.

```
from hyperopt import fmin, rand, hp, Trials

# Random states to make the experiments reproducible
RANDOM_STATES = [1, 2, 3, 4]


def get_best_value(f, candidates):
    """
    Return the candidate that yielded the lowest cost in the function f.
    :param f: cost function
    :param candidates: x candidates
    :return: the best candidate
    """
    idx_min = np.argmin(f(candidates))
    return candidates[idx_min]


def plot_search(search_type, f, candidates, x_min, x_max, selected_value):
    """
    Plot the given function in the interval [x_min, x_max],
    search candidates and selected value.
    :param search_type: the type of the search to be set as title
    :param f: cost function
    :param candidates: search candidates
    :param x_min: left side of the interval
    :param x_max: right side of the interval
    :param selected_value: selected value among the candidates
    """
    # Cost function
    plot_function(f, x_min, x_max)

    # Candidates
    plt.scatter(candidates,
```

```
    # Best value
    plt.scatter(selected_value,
                f(selected_value),
                color='steelblue',
                marker='*',
                s=200,
                zorder=2,
                label='best value: {0:.2f}'.format(selected_value))

    plt.title('[{0}] Cost for best value: {1:.2f}' \
              .format(search_type, f(selected_value)))
    plt.legend(loc='upper left')


def run_grid_search_experiment(f, n_trials, show_plot):
    """
    Run the experiment for grid search.
    :param f: cost function
    :param n_trials: number of trials
    :param show_plot: show plot if true
    :return: the best candidate
    """
    gs_candidates = np.linspace(X_MIN, X_MAX, n_trials)
    selected_value = get_best_value(f, gs_candidates)

    if show_plot:
        plot_search('Grid Search',
                    f,
                    gs_candidates,
                    X_MIN,
                    X_MAX,
                    selected_value)

    return selected_value


def run_random_search_experiment(f, n_trials, random_state, show_plot):
    """
    Run the experiment for random search.
    :param f: cost function
    :param n_trials: number of trials
    :param random_state: value to make the experiment reproducible
```

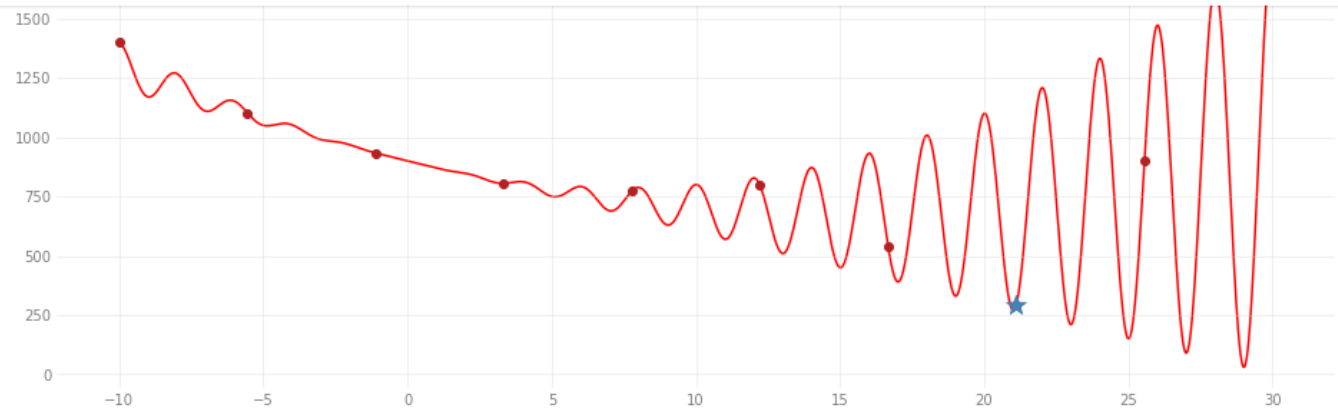
```
            space=hp.uniform('x', X_MIN, X_MAX),
            algo=rand.suggest,
            max_evals=n_trials,
            trials=trials,
            rstate=np.random.RandomState(random_state),
            show_progressbar=False)

    rs_candidates = \
        np.array([t['misc']['vals']['x'] for t in trials.trials]).flatten()

    # note: fmin already returns the best value, but let's use our function :)
    selected_value = get_best_value(f, rs_candidates)

    if show_plot:
        plot_search('Random Search',
                    f,
                    rs_candidates,
                    X_MIN,
                    X_MAX,
                    selected_value)

    return selected_value


def run_experiments(f, n_trials, random_state, show=False):
    """
    Run the experiments for grid search and random search.
    :param f: cost function
    :param n_trials: number of trials
    :param random_state: value to make the experiment reproducible
    :param show: show plot and winner if true
    :return: the best candidates found for grid search and random search
    """
    gs_selected_value = run_grid_search_experiment(f,
                                                   n_trials,
                                                   show)

    rs_selected_value = run_random_search_experiment(f,
                                                     n_trials,
                                                     random_state,
                                                     show)

```

```
135
136
137 def has_gs_won(f, gs, rs):
138     """
139     Evaluate the best candidates in the function f
140     and return whether grid search has won or not.
141     The lower cost the better.
142     For simplicity, a tie gives the victory to grid search.
143     :param f: cost function
144     :param gs: best value found with grid search
145     :param rs: best value found with random search
146     """
147     cost_gs = f(gs)
148     cost_rs = f(rs)
149
150     return cost_gs <= cost_rs
151
152
153 def print_winner(f, gs, rs):
154     """
155     Evaluate the best candidates in the function f and print the winner.
156     :param f: cost function
157     :param gs: best value found with grid search
158     :param rs: best value found with random search
159     """
160
161     if has_gs_won(f, gs, rs):
162         print('Grid Search is the winner!')
163     else:
```

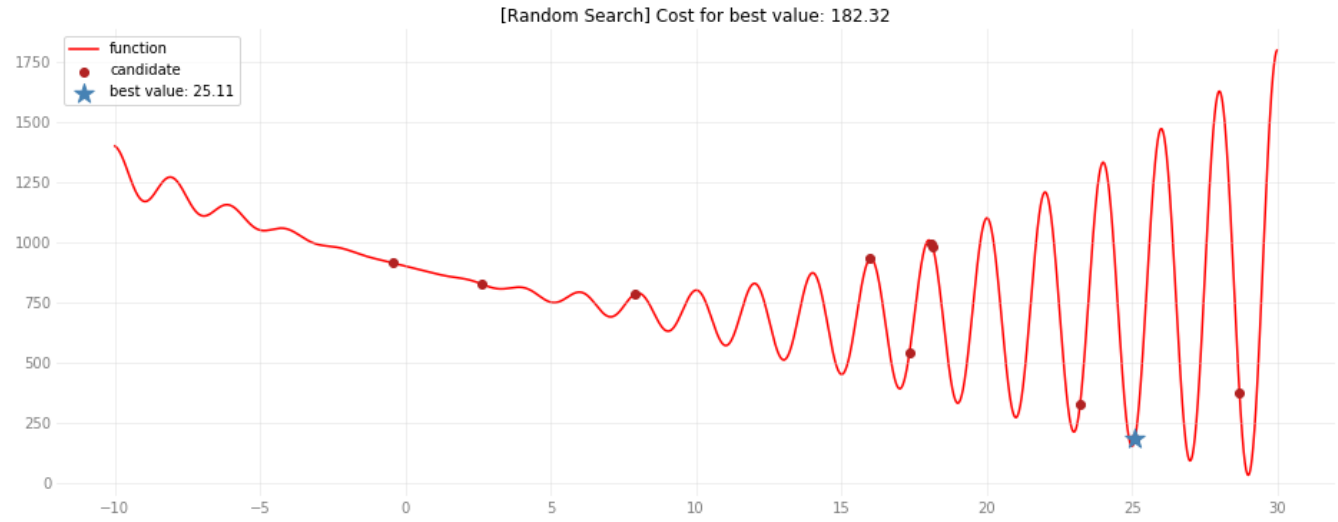
Now that everything is set up, let's do some experiments. As you can see in the code, the candidates for GS will be values according to the number of trials evenly spaced over our interval. For RS, on the other hand, the candidates will be chosen on the fly following the random search strategy. After running GS and RS for a particular number of trials, the name of the winning strategy is printed.

Experiment — 10 trials

Having 10 trials in our budget, let's see which search strategy gives us the value with



Grid Search results in 10 trials



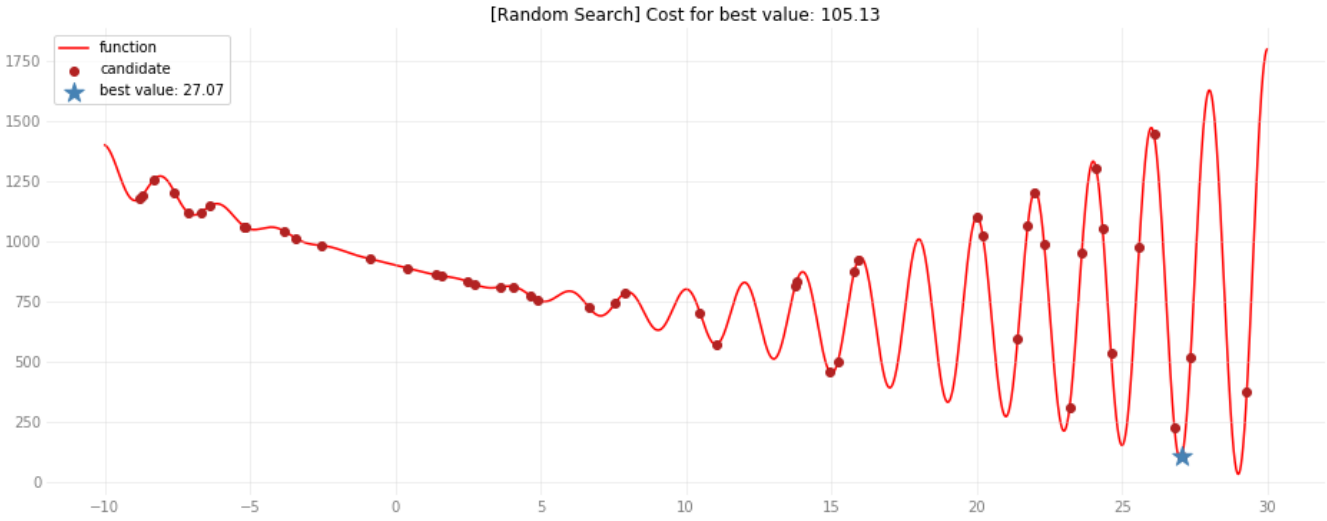
Random Search results in 10 trials

```
Result (10 trials): Random Search is the winner!
```

Experiment — 50 trials

Having 50 trials in our budget, let's see which search strategy gives us the value with the lowest cost.

Grid Search results in 50 trials

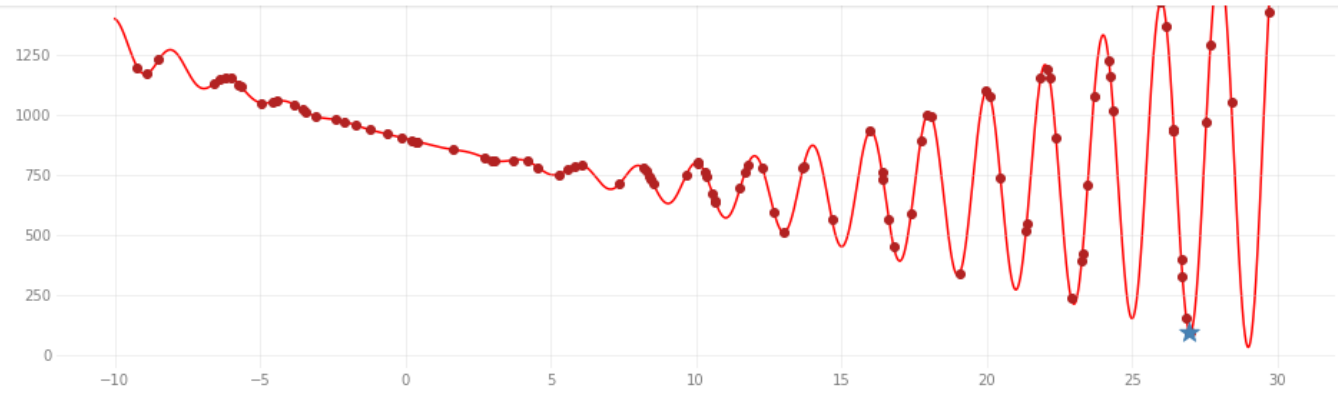


Random Search results in 50 trials

```
Result (50 trials): Random Search is the winner!
```

Experiment — 100 trials

Having 100 trials in our budget, let's see which search strategy gives us the value with the lowest cost.

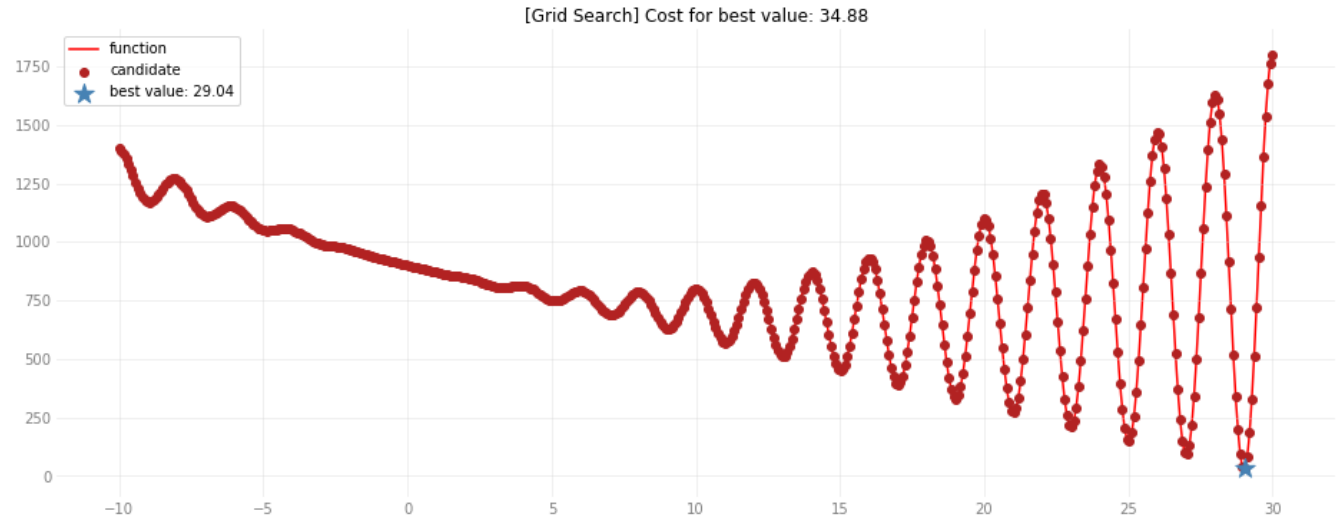


Random Search results in 100 trials

```
Result (100 trials): Random Search is the winner!
```

Experiment — 500 trials

Having 500 trials in our budget, let's see which search strategy gives us the value with the lowest cost.



Grid Search results in 500 trials

Random Search results in 500 trials

```
Result (500 trials): Random Search is the winner!
```

As you can see in all experiments, RS was able to find better values than GS. The differences between the two strategies become lesser as the number of trials is increased, but the small improvement that RS provides could be crucial to reach the performance you're looking for.

Running batch experiments

One could argue that only four experiments might not be reliable, and I totally agree, especially because we are dealing with random stuff. Those experiments were only meant to give us some visualization tips. Hence, let's execute experiments in batch and count how many times each strategy has won. The number of experiments in each batch will be the number of trials. Also, we'll sum the cost of the selected values to get some additional insights. Notice that, as GS is deterministic, it will always return the same result for the same number of trials; so we are testing only RS.

Here's the code to run this experiment.

```
def run_batch_experiments(f, n_trials):
    """
    Run a batch of experiments and print the results.
    :param f: cost function
    :param n_trials: the number of trials which is also the number of experiments
    """
    # Random states to make the experiments reproducible
    rstates = np.arange(1, n_trials + 1)

    # Count how many times each search strategy has won
    gs_count = 0
    rs_count = 0

    gs_cost_sum = 0
```

```
                                    rstate,
                                    show=False)
        gs_won = has_gs_won(f, gs, rs)
        gs_count += int(gs_won == True)
        rs_count += int(gs_won == False)

        gs_cost_sum += f(gs)
        rs_cost_sum += f(rs)

    print('Grid Search:\t{0}\t({1:.2f})'.format(gs_count, gs_cost_sum))
    print('Random Search:\t{0}\t({1:.2f})'.format(rs_count, rs_cost_sum))
```

grid-vs-random-search-part5.py hosted with ❤ by GitHub view raw

Running batch experiments

Results will be presented in the following structure:

```
<execution counter>
Grid Search:   <victory counter>     (<cost sum>)
Random Search: <victory counter>     (<cost sum>)
```

And here are the final results:

```
10
Grid Search:    5       (2935.44)
Random Search:  5       (3692.77)

50
Grid Search:    16      (8119.40)
Random Search:  34      (6885.77)

100
Grid Search:    8       (17449.96)
Random Search:  92      (9118.30)

500
Grid Search:    221     (17441.61)
Random Search:  279     (20750.24)
```

Pretty cool results, aren't they? They suggest the following insights:

be a good idea to set the number of trials as many as you can.

- **500-trial experiment**: If the number of trials is large enough to cover the entire search space for GS, both strategies become more equivalent, as we could visualize in the plots. Despite the cost sum of RS being higher than GS, RS has given the best value more times than GS.
- **50-trial and 100-trial experiments**: For these experiments, RS has shown a better performance not only because it has won more times than GS, but also due to a lower cost sum. It suggests that employing RS could be more interesting than GS if the number of trials is considerably large, but not enough to cover the entire search space for GS.

. . .

Concluding Remarks

In this post, we set up and ran experiments to do comparisons between Grid Search and Random Search, two search strategies to optimize hyperparameters. Although our experiments are simple, they've provided some insights regarding the behavior of the strategies in different scenarios. From everything we've seen, I will let you go with some takeaways:

1. When it comes to research, it is very hard to state that something is better than the other. This case is no different. So I would recommend trying both strategies when optimizing hyperparameters, possibly combining the two of them.
2. If you have many hyperparameters and the search space is awkward, consider starting with RS. After finding good values, check if the cost varies significantly around them. If so, you could fine-tune by using GS.
3. Independently if you employ GS or RS on your research and get good results, take some time to study the values you got, *i.e.* do not simply accept them. Perhaps the values might give you new insights about the problem you're trying to solve and possibly lead your research to new, and more interesting, paths.

Journal of Machine Learning Research.

Machine Learning Hyperparameter Tuning Random Search Grid Search Optimization