Optimization for Machine Learning mini-project : Random search comparisons for hyperparameter optimization

Yuan Vincent
Lepeytre Hugo
Bitsakis Theodoros
Data Science section, EPFL, Switzerland

Abstract—**In machine learning applications, some parameters describing the overall behavior of the model are often necessary, the most common being the learning rate. But finding their optimal value often requires some intuition and trial and error. We go over different approaches used to automate this task, in the form of various flavors of random search. We then present some ideas for improving the effectiveness of random search and compare all of our results to a baseline of manual adjustment using heuristics, and a grid search. We find that random search and a variant inspired by genetic algorithms manage to improve upon the basic grid search.**

I. Introduction

The goal of this paper is to compare several ways of optimizing hyperparameters. Most of the time this task is done manually by humans because of their efficiency when there is a small number of iterations, but it can become a problem when the search space grows even slightly or a new kind of problem is faced. However since the number of dimensions is typically very small, it is a good use case for random search. We will train a simple neural network on a relatively small heart attack analysis classification dataset so as to have a fast training process that can easily be repeated several times.

The optimization algorithm used to train the neural network will be ADAM, so overall the hyperparameters to fine-tune will be the drop-out rate of the neural network, the two decay factors used in ADAM as well as the α factor of the leaky RELU activation function.

Regarding the optimization methods for hyperparameter optimization itself, the two baselines will be a manual fine-tuning, and a simple grid-search. We will compare runtimes, as well as test error for each method. Finally, when each method has given a set of optimal parameters, we will also compare their validation error.

II. Related Work

Recent works in hyperparameter optimization state that most of the tuning is still carried out by humans, and relies heavily on expert's knowledge about optimal parameters for given tasks. Some papers are even focused on finding good parameters depending on the machine learning task at hand [1]. Some other suggested methods are random search and greedy sequential methods [2]. Recent studies have compiled various optimization methods, and reviewed their support in the available machine learning frameworks [3]. According to these wide reviews, the best performing frameworks seem to be Bayesian Optimization (BO), sometimes combined with Hyperband (BOHB) review2 as well as Particle Swarm Optimization (PSO) and Genetic Algorithms (GA) when the configuration space is larger [4]. Here, we will focus on random search and try to extend it using GA-inspired methods.

III. Models and Methods, Dataset

A. Dataset and Naïve model

For reproducibility purposes, we decided to use a dataset in the public domain, the *Heart Attack Analysis and Prediction Dataset* available on Kaggle. This dataset contains medical information about patients, such as the age, the gender, the blood pressure or the maximum heart rate achieved, and if the patient has had an heart disease or not.

Thus, we have a binary classification dataset more or less balanced (54% were positive, 46% negative). The dataset size is relatively small, with 303 entries. Indeed, as our goal is to perform hyperparameter tuning, we will benchmark the dataset a large number of times. As such, we need to be able to perform neural network training and testing with given hyperparameters fast, that's why we decided to stick with a small dataset.

We have performed classic data preprocessing in our dataset. We used one hot encoding for the categorical features, and normalized the continuous features. We divided the dataset in a train dataset to train our model, a validation to perform hyperparameter validation, and a final test set to benchmark our results. The distribution ratio between train, test and validation is 70%, 15% and 15%.

As mentioned in the introduction, we decided to use neural network for our model using ADAM optimizer. Our neural network is a multi-layer perceptron, with 3 fully connected layers of size 64, 64 and 16. The input is of size 30 and the output is a single neuron. We used LeakyReLU as activation function, and a sigmoid activation function for the output. We added dropout in all layers.

As our project is about benchmarking different variant of random searches for hyperparameter tuning, we have chosen this model for its hyperparameters. We have 6 hyperparameters in our model: the dropout rate of our neural network, the alpha value of our LeakyReLU, the two betas values of our ADAM optimizer, our learning rate, and the weight decay of our ADAM optimizer. These hyperparameters values all lie between -1 and 1

B. Manual fine-tuning

We firstly fine-tuned manually the hyperparameters, starting from pytorch initial hyperparameters and testing manually hyperparameters.

C. Grid Search

With the results obtained from our manual fine-tuning, we decided to compute the hyperparameters the ”classical” way, using a grid search. However, as we have 6 hyperparameters, we will have to benchmark for n^6 different set of hyperparameters if we benchmark n different values for each parameters. With limited computation powers, we had to carefully the benchmarked hyperparameters of our grid search.

To perform this selection, we firstly computed a grid search for each of the hyperparameter while fixing the 5 others to our manual fine-tuning results, to extract the 3 values that performed the best for each hyperparameter. Then, we benchmarked each of the $3^6 = 729$ hyperparameters combination with our validation set to find the one with the best accuracy and the one with the best loss. After extracting these 2 hyperparameters combination, we benchmarked them with the test set.

D. Basic random search

We developed a weighted random search algorithm. This algorithm takes as an input an initialization vector and a weight vector . At each iteration, it will initialize a random vector and scale it point wise with the weight vector. It will then iterate over a given set of gamma defined in a logarithmic way, taking both positive and negative values, and compute a new vector, equal to the current vector minus the random vector scaled times gamma. It stores the loss, and, at the end of the gamma iterations, keep the value that had the lowest loss and take it as the new vector.

We decided to use weight and initialization vector to try two different kind of random search. The first random search, called *initialized*, has pytorch initial hyperparameters, and weight corresponding to the order of magnitude of these initial hyperparameters. For example, we know that ADAM second beta value begins with 0.999, while it’s common for dropout value to be equal to 0.5, even though we can have a neural network with no dropout . The weights therefore represent this order of magnitude. The idea behind this random search is to start at values that are supposed to work well, and then tune these hyperparameters by respecting the ratio between each of them.

The second random search, called *median*, is initialized at the center of the hyperparameter space, with a vector of 6 values equal to 0.5. The weights are all equal to 1. The idea is to be able to make a totally random search, with the hyperparameters starting at the center of the space, and being able to go anywhere.

E. Extendend random search

Finally, to extend the random search framework, we designed two other random search algorithms, loosely inspired by genetic algorithms :
The first one starts with a set of n hyperparameter vectors. At each iteration, every vector in the set is used to create m new vectors with their coordinates slightly modified in a random fashion. Then only the n models trained with the lowest validation loss are kept. We repeat this over a fixed number of iterations and return the vector with the lowest validation loss over the whole run.
The second method also starts with a set of vectors but this time, new vectors are generated by iterating over every possible pair of vectors in the set, and selecting each feature uniformly from one of the two parent vectors. Then the hyperparameters are randomly modified with a probability of $\frac{1}{2}$. Then once the generation is over, we keep only the best obtained vectors for the next iteration.

IV. Results

A. Numerical Parameters

The set of parameters studied during the grid search is :

Dropout	Slope	LR	Beta1	Beta2	Weight decay
0	0	1e-5	0.5	0.9	0
0.15	0.1	1e-4	0.75	0.999	1e-7
0.3	0.2	1e-3	0.9	0.99999	1e-5

For the random search, the parameters were the weight vector and the initialisation vector, whose values were :

Method and vector	Dropout	Slope	LR	Beta1	Beta2	Weight decay
Initialized initialization vector	0	0	1e-3	0.9	0.999	0
Initialized weight vector	1e-1	1e-1	1e-3	1e-1	1e-3	1e-8
Median initialization vector	0.5	0.5	0.5	0.5	0.5	0.5
Median weight vector	1	1	1	1	1	1

For the first extension of random search, we used a population of size 5 and generated 3 new vectors for every element in each of the 50 iterations.
For the second extension on random search, the population size was 10, and the number of iterations was also 50.

B. Comparison

The set of optimal hyperparameters found by each method, as well as their average loss and standard deviation was as follows :

Method	Manual	Grid Search	Random Search	Extension 1	Extension 2
Dropout Rate	0.1	0.15	0	0.025	0.0004
Negative Slope	0.1	0.1	0.61	0.0001	$6.57e^{-6}$
Learning Rate	0.0001	0.001	1	0.003	0.0006
Beta 1	0.5	0.5	1	0.85	0.003
Beta 2	0.9	0.9	1	0.9999	0.999
Weight Decay	e^{-7}	e^{-7}	0.67	$1.32e^{-5}$	0.0002
Loss std	0.09	0.149	0.065	1.112	0.097
Loss avg	1.376	1.228	1.124	1.177	1.205

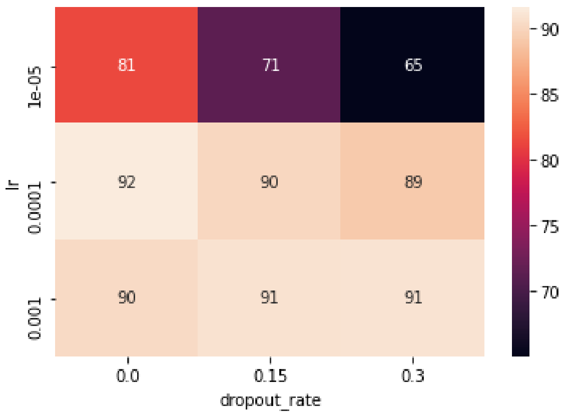

Fig. 1. Heatmap of the accuracy in the grid search, based on dropout and learning rate

You can find above a heatmap of the grid search, on the learning rate and the dropout value:

V. Discussion

From the results, we can see that grid search is already better than manual tuning, and random search improves a bit on that. On the side of the extensions, the first one has similar performance to the regular random search, while the second one performed poorly. This might be due to the lower number of iterations since the method is more computationally intensive.

Overall, the results are only barely significant, due to the size of the dataset, which is low in order to allow for faster training. But still, training a small multi-layer perceptron at every iteration is very computationally extensive, so it would be advisable to try and reproduce the results with higher computational power and more time to try and get more significant results.

VI. Conclusion

Hyperparameter optimization is a large subject that has only been barely explored yet, and there are several promising methods that allow for a big improvement on the standard manual tuning based on expert's knowledge. Random search is already a big improvement, but more involved methods treating the validation loss as a black-box function can get computationally expensive extremely fast, so focus on approximating the validation error faster than by training the model every time would be a promising approach. Finally, more trials with a higher amount of resources and time could lead to more insight as to the efficacy of the extended random search methods proposed here.

References

[1] B. B. Philipp Probst, Anne-Laure Boulesteix, "Tunability: Importance of hyperparameters of machine learning algorithms," *The Journal of Machine Learning Research*, vol. 20, no. 1, 2019.

[2] Y. B. B. K. James Bergstra, Rémi Bardenet, "Algorithms for hyper-parameter optimization," *Advances in Neural Information Processing Systems 24 (NIPS)*, 2011.

[3] H. Z. Tong Yu, "Hyper-parameter optimization: A review of algorithms and applications," 2020, https://arxiv.org/abs/2003.05689.

[4] A. S. Li Yang, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Elsevier Neurocomputing*, vol. 415, pp. 295–316, 2020.