Oct 28, 2018 21 min read

Learning to Optimize

Introduction

Hyperparameter optimization is one of the most difficult problems in machine learning. How many layers should my neural network be? How many kernels should each layer have? What type of activation function should I use? What value should I set as my learning rate?

In the month of August, I became obsessed with this problem - how to optimize hyper-parameters. I started creating an open-source tool called Hypermax, available for free through Python's "pip install" command, or directly from the latest source code on Github (https://github.com/electricbrainio/hypermax). Initially, the tool was just there to do some surrounding non-core stuff, such as monitor the execution time and RAM usage of models, and to generate graphs based on the results of the trials. The core algorithm was not my own - it was simply the TPE algorithm, as implemented in the Hyperopt library. Originally, Hypermax was just intended to make optimization more convenient.

But I quickly became obsessed with this idea of how to make a better optimizer. TPE was an industry leading algorithm. Although invented in 2012-2013, it was still being cited as recently as 2018 as one of the best algorithms for hyperparameter optimization. In most cases, it was the leading algorithm, but more recently there have been a few new approaches, like SMAC, which can do just as well in general, but not clearly better in all cases. Even more recently (July 2018), there has been work on this algorithm called Hyperband, and its evolved version, BOHB (which is just a combination of Hyperband and TPE). However, TPE is still an industry leading algorithm, with top-notch general case performance across a wide variety of optimization problems. It has also been well validated by many people who have used it across many machine learning algorithms and proven its effectiveness. It is for this reason that we decided to use it as the basis of our research. Our goal was to make an improved version of TPE.

We started our research by testing various ways that TPE can be manipulated in order to improve its results, and various ways that we could gather data in order to know what works. This formed the basis of the first version of our Adaptive-TPE algorithm. You can see an article where we describe the research here: https://www.electricbrain.io/blog/optimizing-optimization This first research was great, and validated that we were onto something, both in terms of our approach to researching a better TPE, and in the specific techniques that we used to improve it. In that research, 4 of the 7 things we tried improved TPE. The other 3 things we tried had absolutely no impact on the results (neither worse or better).

In this article, we will discuss version two of our Adaptive-TPE algorithm. Version 2 takes our approach to a whole new level. We are repeating a basic description of our approach in this

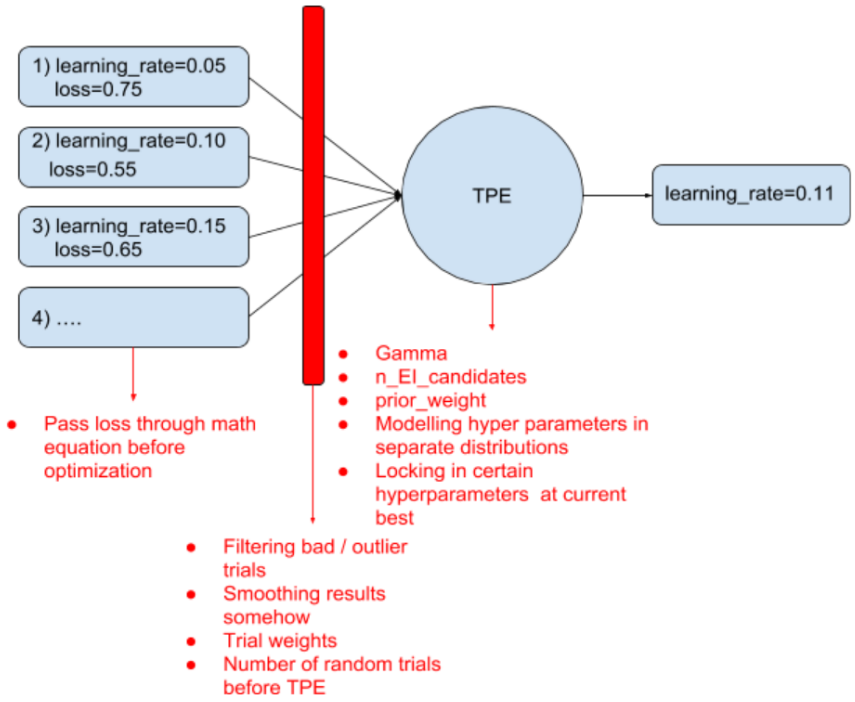
article so that you don't have to read the original research. However, it will be condensed. So if you want a full understanding of how these ideas came to be, we suggest you read the original research shared in that original article (https://www.electricbrain.io/blog/optimizing-optimization)

All of the tables and graphs in this article have been screenshotted from the original document, due to some limitations in my blogging platform. To view the original document, go here: (https://docs.google.com/document/d/13D72_9VoORrU8yos_irhkQ2LUv49OpAQQfhkPg_-2Zs/edit?usp=sharing)

Levers for Improving TPE

TPE (Tree of Parzen Estimators), and the Hyperopt implementation of it, don't have a whole lot of hyperparameters that we can try to tune. But it does have a few. Additionally, there are algorithmic things we can do to mess with the algorithm and see if we get better results.

We can think of TPE as being an algorithm which takes a history of trials, and predicts the best trial to test next. TPE itself has a single primary hyper-parameter, gamma. But we can manipulate the data it uses as input in a variety of ways. We can also provide weights to the trials inside the TPE algorithm. We can change our loss function, which might alter the way that TPE behaves. Additionally, there's a couple hypothetical ways we could manipulate TPE - we could force TPE to spend more time exploring certain hyper-parameters by 'locking in' the values for the others for some trials. Additionally, we could have separate TPE distributions for separate hyper-parameters, even while they are being optimized jointly. This may sound crazy but there is some underlying motivation for this if those hyper-parameters are highly independent of one another. So overall, our landscape for improving the TPE algorithm looks like this:



In our first round of ATPE research, we tested 7 of these techniques for improving TPE. We were able to validate 4 techniques that improved TPE:

- Pass loss through math equation before optimization (failed)
- Number of random trials before TPE (success)

- Number of random trials before TPE (success)
- Tuning Gamma (success)
- Tuning n_EI_candidates (success)
- Tuning prior_weight (failure)
- Modelling hyper-parameters in separate distributions (failed)
- Locking in certain hyperparameters at current best value (success)

Additionally, in the original paper, they tested the “Trial Weights” technique and already validated that it improves results. We were unable to dissect the Hyperopt code enough to get Trial Weights working ourselves. However, filtering trials has a similar the same effect as weighting them (since trial weights was only used to down-weight old trials). The only difference is that there is a bit more stochasticism involved in filtering, and weighting would have likely been more smooth and consistent. So ultimately we decided to include filtering despite not validating it in our original research.

In addition, because of the success of the parameter locking (also called Bagged Optimization in our original paper), we wanted to expand upon the ways that parameters could be locked, to give options for our ATPE algorithm. We made several adjustments to parameter locking compared to the first ATPE algorithm:

1. We expanded the algorithm to allow locking parameters both to top performing results or to a random value from anywhere in that parameters range, e.g. random search that parameter
2. When locking to a top performing result, we allow locking not just to the current best, but to any result within the top K percentile of results
3. When choosing the cutoff point for which parameters to lock, we multiply the correlations by an exponent, which alters the distribution of correlations to have more or less bias towards the larger numbers
4. We allow the algorithm to choose either high correlated parameters or low correlated when selecting the cutoff point, e.g. it might choose to lock the 50% highest correlated parameters or the 50% lowest correlated parameters. Previously only highly correlated parameters would be locked first.
5. When deciding whether to lock a parameters, we allow the probability to vary from a 50-50 chance. We have two different modes for choosing the probability - either a fixed probability, or a probability determined based on multiplying the parameters correlation by a factor C

The idea in general was to give the algorithm a variety of ways to explore more tightly or more widely depending on what the circumstances are.

So ultimately, our ATPE algorithm looks roughly like the following:

1. Compute Spearman correlations for each hyperparameter, raise to an exponent
2. Determine a cutoff point between correlated and uncorrelated parameters based on the cumulative correlation. The parameters can be sorted in either direction, with correlated ones first or uncorrelated ones first
3. For parameters that are chosen for potential locking, they have a random chance of being locked in that round. The probability varies by depending on the mode for that round:
   - In fixed probability mode, there is a fixed chance that this parameter will be locked, varying from 20% to 80%
   - In correlation probability mode, the chance of locking this parameter is chosen by multiplying the correlation by a factor C
4. We assign the locked values depending on what mode ATPE is in:
   - If the ATPE mode is in random-locking mode, then for parameters that are chosen to be locked, we assign them a random-value from anywhere in their search space
   - If the ATPE mode is in top-locking mode, then we choose a single top performing

result from the result-history. The top performing result may not be the current best, but is chosen from the top K percentile of results for that round. All parameters chosen for locking are then locked to the same value they had in that top-performing result.

5. We create an alternative result history which has some results filtered out randomly according to the filtering mode for this round:
   - If the filtering mode is Age, then we randomly filter out results based on their age within the result history. We use their position within the result history (0 to 1 where 0 is the first result and 1 is the most recent), and multiply that position by a factor F which determines the probability that the result will be kept. We randomly determine if the result will be kept using that probability. This creates a bias towards keeping more recent results, with more or less filtering controllable by factor F
   - If the filtering mode is Loss_Rank, then we randomly filter out results based on the ranking of their losses. The mechanism is similar to Age filtering, except the results have been sorted by their losses first.
   - If the filtering mode is Random, then all results have an equal likelihood of being filtered out, with probability P
   - If filtering mode is None, then we do not perform any filtering and retain all results in the history
   - Only the TPE algorithm receives the filtered result history. The statistics and locking are always computed based on the full history.
6. Lastly, we use the magical Bayesian math of the TPE algorithm to predict the remaining parameters that have not been locked:
   - We provide the TPE algorithm the values for the parameters that have been locked. This allows it to perform partial-sampling - e.g. it will predict good values for the remaining parameters, given the values for the locked ones.

All in all, this is an optimization algorithm with the Bayesian math of the TPE algorithm at its core. It has simply been surrounded with two additional mechanisms - one to filter the results fed into the TPE algorithm, and the other is to only use TPE for a subset of parameters at a time, while locking the remaining ones using the mechanism described above. TPE still uses its magic to predict the remaining parameters given the values of the locked ones.

The algorithm appears to have a lot of potential, but implementing it introduces a new problem: How do you choose ATPE's own parameters at each round? It itself has a bunch of parameters that have to vary as it gains new information. They are described here with their actual variable names in the code:

- resultFilteringMode: Age, LossRank, Random, or None
- resultFilteringAgeMultiplier: For Age filtering, the multiplier applied to the results position in history to determine the probability of keeping that result
- resultFilteringLossRankMultiplier: For Loss filtering, the multiplier applied to the rank of the result when sorted by loss, used to determine the probability of keeping that result
- resultFilteringRandomProbability: For Random filtering, the probability of keeping any given result
- secondaryCorrelationExponent: This is the exponent that correlations are raised by when determining which parameters to lock and which ones to explore with TPE. Varies between 1.0 and 3.0
- secondaryCutoff: When choosing which parameters to consider for locking, this is the cutoff point of cumulative correlation that is used to determine the parameters to be locked. Varies between -1.0 and 1.0. Negative numbers are used to indicate a reverse sorting, where low-correlated variables are locked first. Positive numbers indicate that high-correlated variables should be locked first.
- secondaryLockingMode: Either Top or Random, determines whether we explore tightly near a top performing result, or we explore widely by forcing more random search
- secondaryProbabilityMode: Either Fixed or Correlation, determines whether we will

choose parameters to lock based on a fixed probability or a probability based on that parameters correlation
- secondaryTopLockingPercentile: When in top-locking mode, this is the percentile of top performing results, from which we will choose one to lock parameters to
- secondaryFixedProbability: When in Fixed probability mode, this is the probability that a given parameter will be locked. Varies between 0.2 and 0.8
- secondaryCorrelationMultiplier: When in Correlation probability mode, this is multiplier that is multiplied by a parameters Spearman correlation to determine the probability that this parameter will be locked
- nEICandidates: This is the nEICandidates parameter to be passed into TPE algorithm at the end of the process
- gamma: This is the Gamma parameter to be passed into the TPE algorithm at the end of the process

In the first phase of our research, we used grid-searching to rigorously search out all the possible values for the various parameters of ATPE over a dataset of possible optimization problems - at that time, only nEICandidates, Gamma, prior_weight, initialization rounds, and secondaryCutoff were being searched. We then came up with simple linear equations for these parameters based on the data on what works best.

Our problem with this new approach is twofold. First, there are way too many parameters for grid-searching to be an effective option. Additionally, because the parameters have a lot of potential to interact with each other, a simple linear equation wasn't going to help us predict the parameters.

What we needed was to somehow gather an enormous dataset, without grid searching, which we could then use to train a full machine learning model to predict the optimal ATPE parameters at each round.

The Dataset

Obtaining the dataset is the biggest challenge in training the ATPE algorithm. To understand the full scope of the problem, try to appreciate the following steps that would be needed to gather the data:

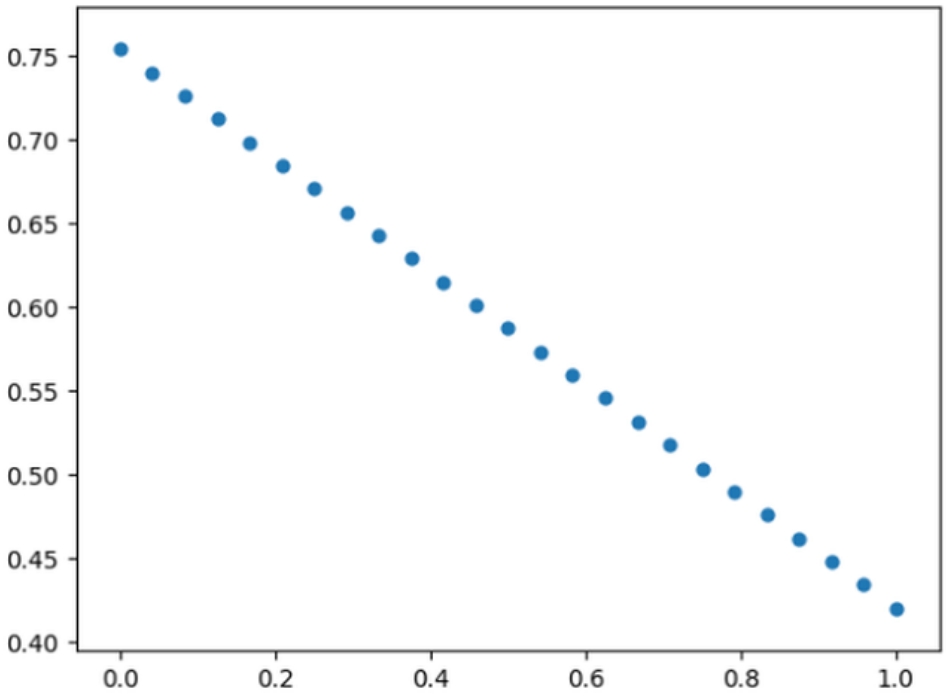
1. Your primary model, M, takes 2 hours to train and test on average
2. Optimizing M with bayesian algorithm O, takes around 100 trials, thus 100x2=200 hours
3. To test an algorithm O on M, you need to run it several multiple times, since it is stochastic, thus 5x200 = 1000 hours
4. To know if your algorithm O works in general, you need to run it on many M representing a diverse array of problems, thus 100x1000 hours = 100,000 hours
5. To gather a dataset on algorithm O's parameters, you need to run many trials, at a minimum of 1,000 trials. Thus 1000x100000=100,000,000 hours
6. If you make a mistake part way through, you have to do it all over again

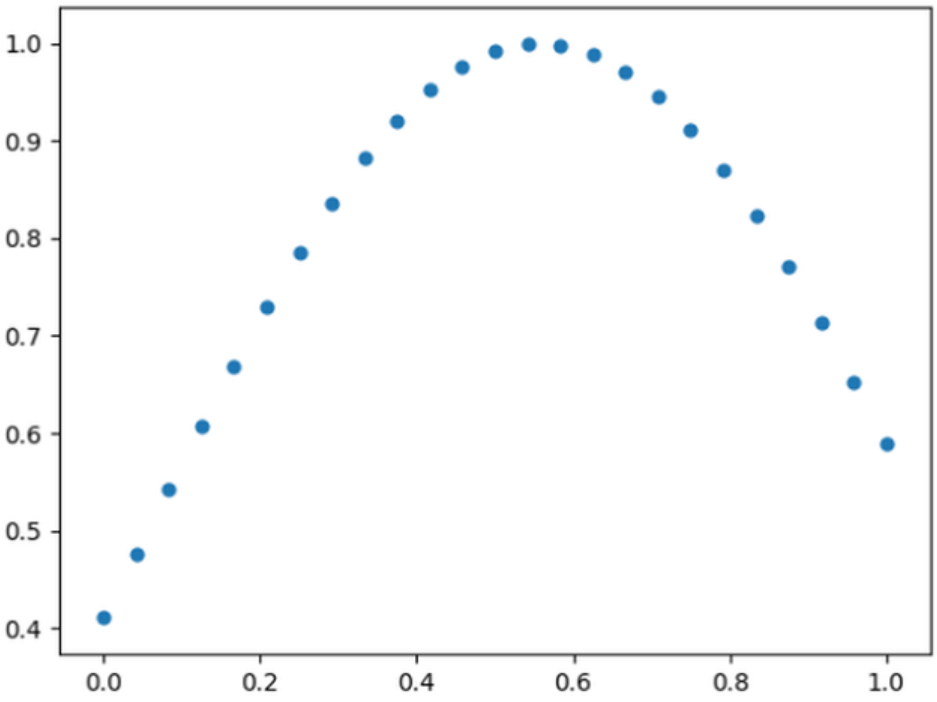
It becomes clear - researching optimization is hard. The number of compute hours you need to make effective progress in optimization quickly blows up to absolutely ridiculous numbers if you attempt to research it like you would any other abstract data-science problem.

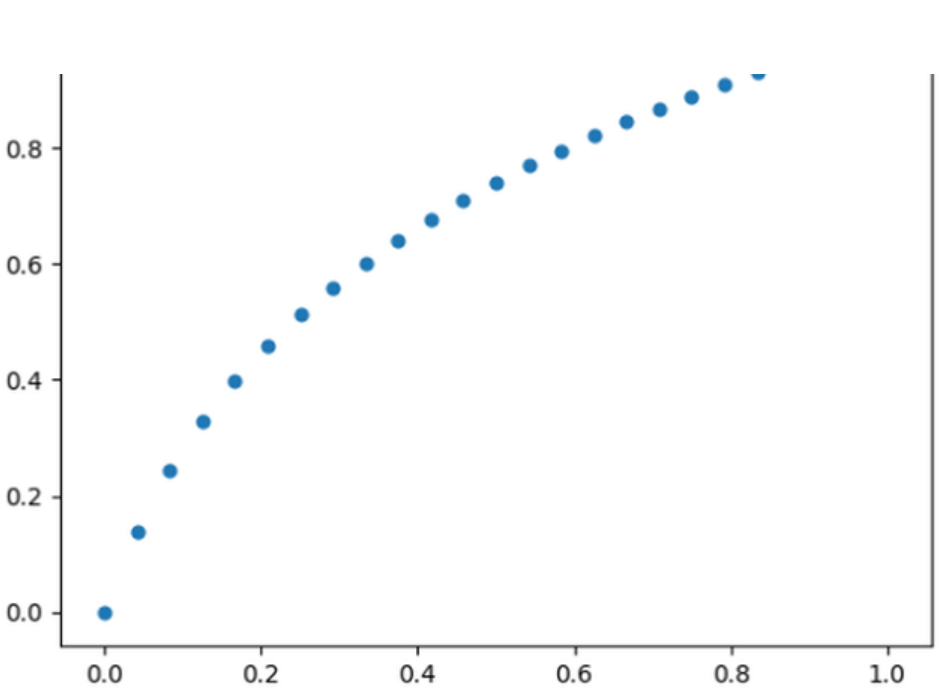
Our solution to this problem was discovered in the original ATPE paper, Optimizing Optimization, discussed in the section "Simulated Hyperparameter Spaces". If you are not familiar with that paper, we suggest reading that section of it to get familiar with it. We will present a TLDR here for reference.

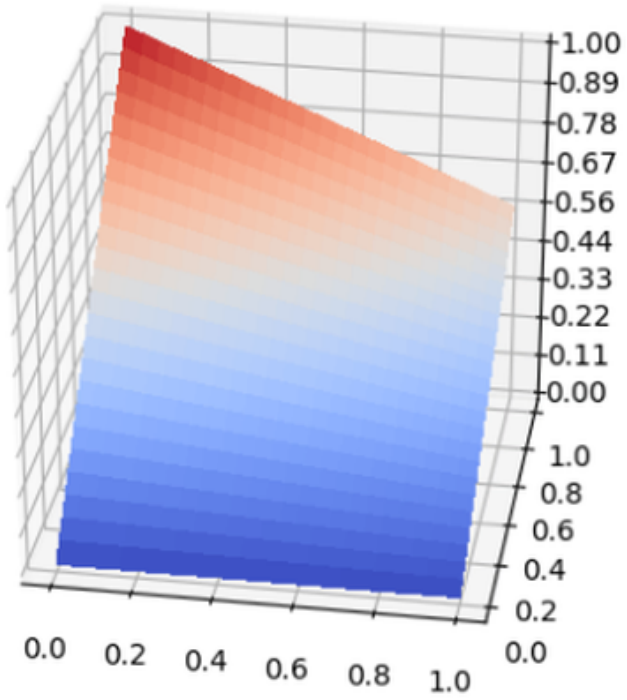
Simulated Hyperparameter Spaces

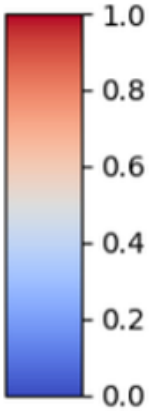
In essence, our “Simulated Hyperparameter Spaces” are just systems of math equations that are designed to respond and behave in a manner that is similar to hyperparameters of real world machine learning models. They are randomly generated with different numbers of parameters, and different ways that those parameters can interact to contribute to the loss of a model.

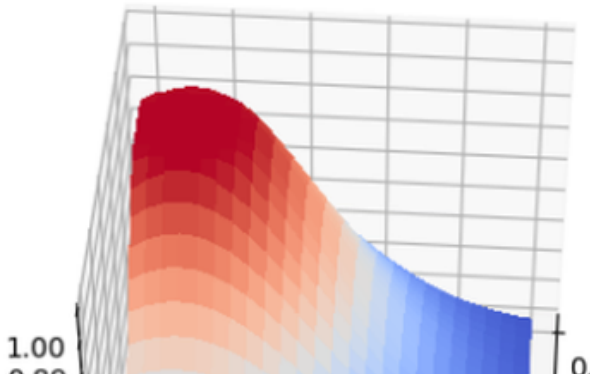


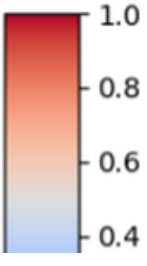


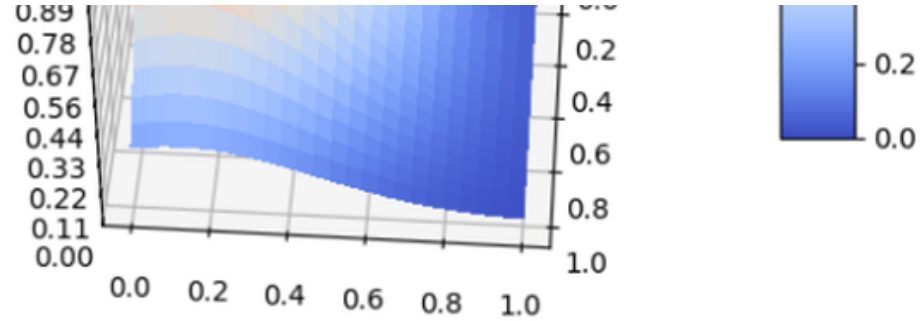

0.0
0.2
0.4
0.6
0.8
0.0
0.2
0.4
0.6
0.8
1.0

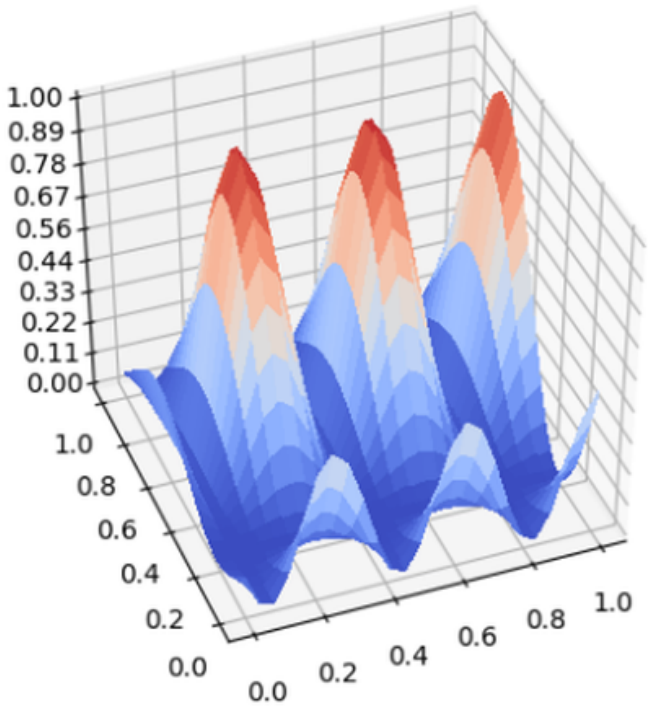

1.00
0.89
0.78
0.67
0.56
0.44
0.33
0.22
0.11
0.00
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

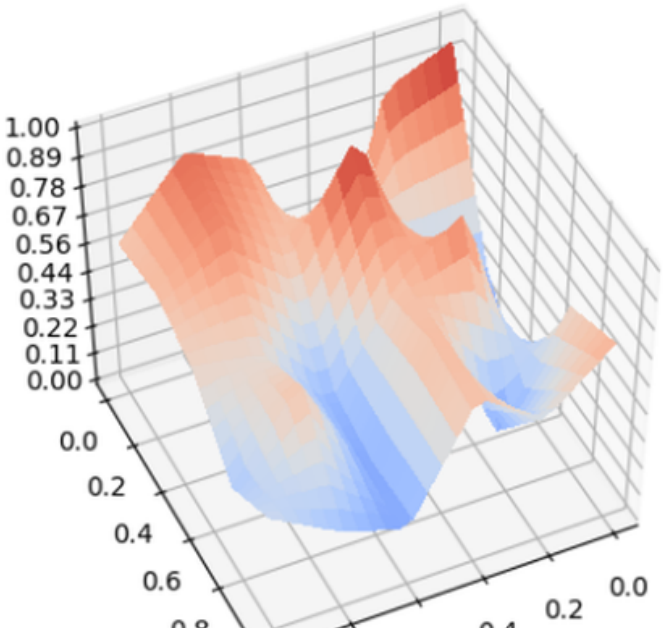

1.0
0.8
0.6
0.4
0.2
0.0

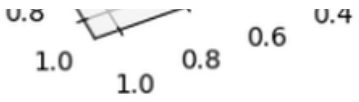

1.00
0.0


1.0
0.8
0.6
0.4


0.89
0.78
0.67
0.56
0.44
0.33
0.22
0.11
0.00
0.0
0.2
0.4
0.6
0.8
1.0
0.2
0.0


1.00
0.89
0.78
0.67
0.56
0.44
0.33
0.22
0.11
0.00
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.0
0.8
0.6
0.4
0.2
0.0


1.00
0.89
0.78
0.67
0.56
0.44
0.33
0.22
0.11
0.00
0.0
0.2
0.4
0.6
0.0
0.2
1.0
0.8
0.6
0.4
0.2
0.0



The various interactions have been hand-chosen and tuned based on our experience with various machine learning models, looking at the graphs and charts of grid-searches of parameters.

Compared to the original ATPE research, the simulated parameter spaces we introduced for this algorithm have a number of key additional features, meant to help model more real world situations:

1. The new simulated hyperparameter spaces have a “noise level”, which is an amount of noise that is added to the final result to obfuscate the exact loss
2. The new simulated hyperparameter spaces have a “failure rate”. This is a probability that the model will return a loss of 1.0 despite having good parameters. We based this on observations of real world optimization where code doesn’t always work and result histories can thus have errors in them
3. The new simulated hyperparameter spaces have both a primary set of parameters and several groups of ‘secondary’ parameters. The model is set up so that only one group of secondary parameters is chosen at any trial, and the secondaries all have different interactions with the primaries. This is to help our dataset include more “architectural” type decisions, where certain parameters can radically alter the behaviour of other parameters, and where parameters are not always included on every trial based on that architectural decision

Additionally, we changed the way that these simulated hyperparameter spaces are chosen. In the original ATPE research, we just generated the parameter spaces as-is and measured the results.

Because of the many arbitrary decisions that were made in designing the simulated hyperparameter spaces, we became very concerned with the bias that is introduced by our random-generation. Obviously, our simulated hyperparameter spaces are not a perfect representation of real-world hyperparameters, and the adhoc approach to generating them them would likely cause a central-tendency where many of the the hyper parameter spaces are similar to each other with muddied interactions.

To counterbalance this effect, we decided to apply a selection process to the simulated hyperparameter spaces.

First, we would generate N hyperparameter spaces. Then we would do a quick, 100 trial bayesian search of each space using the vanilla TPE algorithm. We compute all of our predictive features & statistics on the results of that bayesian search. We normalize the various statistics to have mean of 0 and deviation of 1, and then we apply a multiplier to each of them to weight them.

We then apply an agglomerative-clustering to cluster together the hyperparameter spaces with similar features and statistics. We chose to use complete-linkage clustering with euclidean distance as our metric. This clustering algorithm results in unevenly sized clusters and works on large highly dimensional data - key to our result. We extract K clusters and for each cluster, we choose a random simulated hyperparameter space within that cluster.

The result of this process is that we have clustered together hyperparameter spaces that appear to have similar properties from a bayesian learning perspective. We can then use this to select a set of hyper-parameter spaces that represent a diverse set of properties, from a bayesian learning perspective. This helps us hedge against the effects of our random-

bayesian learning perspective. This helps us hedge against the effects of our random generation process and ensure we get a set of hyper-parameter spaces that represent as diverse properties as possible. We over-represent hyperparmeter spaces with unusual or different properties to the norm.

These simulated hyperparameter spaces can be executed in under 50ms each. This takes our previous problem of needing 200 million compute hours to:

= 50ms * 100 * 5 * 100 * 100
= 2500000 seconds
= 694.4444 compute hours

Now the problem might actually be tractable for us to compute with some rented servers!

Predictive Statistics and Features

Since we plan to have a machine learning model which predicts the optimal ATPE parameters, we need to have some sort of input to that model. We need various predictive features that we can compute on results which capture the essence of how the optimization is performing.

There are two types of predictive features.

The first are features that are directly determined by the hyperparameter space. The primary ones are log10_cardinality, and num_parameters, which are easily computed from the hyperparameter space. When doing the agglomerative clustering algorithm described in the "Simulated Hyperparameter Spaces" section, we also include features which are derived from how the simulated hyperparameter space was generated. These features are not actually measured for our machine learning model, and are only used in that clustering algorithm.

The second set of features are those computed on the results. First, we filter the results in different ways to capture different aspects of it. We filter for only the top 10% of results, or only for the most recent 25 results. Then we compute various statistical features for each group of results. We compute statistical features both on the losses of the results, and the correlations between parameters and those losses. The complete list of statistical features, and their weights in the clustering algorithm, are given in this table:

Group	Statistics	Weight
Hyperparameter space	num_parameters	10
	log10_cardinality	25
All Results	best loss/median of losses	3
	Skew of losses	9
	Kurtosis of losses	9
	stddev/median of losses	6
	stddev/best of losses	3
	Variance / median of losses	3
	Variance / best of losses	3
	Best correlation / median correlation	3
	Skew of correlations	3

	Kurtosis of correlations	3
	Stddev of correlations	3
	Variance of correlations	3
Top 10% of results	Best loss / (10 percentile loss)	3
	Skew of top 10% losses	9
	Kurtosis of top 10% losses	9
	Stddev / median of top 10% losses	6
	Stddev / best of top 10% losses	3
	variance / median of top 10% losses	3
	Variance / best of top 10% losses	3
	Best correlation / median correlation of 10% losses	3
	Skew of correlations of top 10% losses	3

	Kurtosis of correlations of top 10% losses	3
	Stddev of correlations of top 10% losses	3
	Variance of correlations of top 10% losses	3
Top 20% of results	Best loss / (20 percentile loss)	2
	Skew of top 20% losses	6
	Kurtosis of top 20% losses	6
	Stddev / median of top 20% losses	4
	Stddev / best of top 20% losses	2
	variance / median of top 20% losses	2
	Variance / best of top 20% losses	2
	Best correlation / median correlation of 20% losses	2
	Skew of correlations of top 20% losses	2
	Kurtosis of correlations of top 20% losses	2
	Stddev of correlations of top 20% losses	2
	Variance of correlations of top 20% losses	2
Top 30% of results	Best loss / (30 percentile loss)	1
	Skew of top 30% losses	3
	Kurtosis of top 30% losses	3
	Stddev / median of top 30% losses	2
	Stddev / best of top 30% losses	1
	variance / median of top 30% losses	1

	Variance / best of top 30% losses	1
	Best correlation / median correlation of 30% losses	1
	Skew of correlations of top 30% losses	1
	Kurtosis of correlations of top 30% losses	1
	Stddev of correlations of top 30% losses	1
	Variance of correlations of top 30% losses	1
Most recent 10 results	Best loss / (20 percentile loss)	2
	Skew of top 20% losses	6
	Kurtosis of top 20% losses	6
	Stddev / median of top 20% losses	4
	Stddev / best of top 20% losses	2
	variance / median of top 20% losses	2
	Variance / best of top 20% losses	2
	Best correlation / median correlation of 20% losses	2
	Skew of correlations of top 20% losses	2
	Kurtosis of correlations of top 20% losses	2
	Stddev of correlations of top 20% losses	2
	Variance of correlations of top 20% losses	2
Most recent 15 results	Best loss / (20 percentile loss)	2
	Skew of top 20% losses	6
	Kurtosis of top 20% losses	6
	Stddev / median of top 20% losses	4
	Stddev / best of top 20% losses	2
	variance / median of top 20% losses	2
	Variance / best of top 20% losses	2
	Best correlation / median correlation of 20% losses	2
	Skew of correlations of top 20% losses	2
	Kurtosis of correlations of top 20% losses	2
	Stddev of correlations of top 20% losses	2
	Variance of correlations of top 20% losses	2
Most recent 15% of results	Best loss / (30 percentile loss)	1
	Skew of top 30% losses	3
	Kurtosis of top 30% losses	3

	Stddev / median of top 30% losses	2
	Stddev / best of top 30% losses	1
	variance / median of top 30% losses	1
	Variance / best of top 30% losses	1
	Best correlation / median correlation of 30% losses	1
	Skew of correlations of top 30% losses	1
	Kurtosis of correlations of top 30% losses	1
	Stddev of correlations of top 30% losses	1
	Variance of correlations of top 30% losses	1
Interactions & Contributions	Interactions_index	2
	interactions_weight	2
	interactions_linear	2
	interactions_wave	2
	interactions_hillvalley	2
	Interactions_random	2
	contributions_linear	2
	contributions_exponential	2
	contributions_hillvalley	2
	contributions_logarithmic	2
	contributions_random	2
	noise	5
	fail_rate	5

The end result is a set of statistical features that can capture how the relationships look:

1. For top performing results at different percentiles
2. For recent results with different intervals
3. For the entire dataset

We capture both the shape of the loss of results, and the shape of correlations between parameters and the loss.

Gathering the Data

Now that we have our simulated hyperparameter spaces, it was time to begin gathering data. The basic idea is as follows:

- For each search space S, we are trying to minimize the loss over N trials
- At each point in the trial history, we need to find the optimal ATPE parameters, P, which lead to the best loss at the end of the search
- We need to do multiple runs to hedge against effects of stochasticly searching

We had several concerns:

We had several concerns:

- We can't grid search the ATPE parameters, there are too many of them.
- We need to optimize for the final loss after N trials, but we can't search every possible combination of ATPE parameters that could exist at each point in the trial history, as it would be near infinite
- We can't assume that the ATPE parameters chosen are always optimal. E.g. if we constructed a dataset where at each point in the history, all prior trials were done with the optimal ATPE parameters, we have biased our dataset on the assumption that our final machine learning model to predict ATPE parameters is perfectly accurate

In the end, we decided to go with the following approach to the search for ATPE parameters:

- For each search space S, we would maintain 5 different optimization "histories", H1 to H5
- At each point in optimizing history Hn, we will use the conventional TPE algorithm to stochastically search for the optimal ATPE parameters.
- When evaluating ATPE parameters, we don't just run a single trial, we instead of run K additional trials, all with the same ATPE parameters, starting from Hn
- K varies depending on how trials have been conducted. Less than 100 trials, and K=10. Above 100 trials, K=25. This is meant to reflect that as you optimize further, you need to do more trials in order to find a best result.
- We record the features and statistics computed for history Hn along with what was found to be the optimal ATPE parameters
- For all tested ATPE parameters, we sort them by the final loss they had for search space S.
- We extend the histories H1 to H5 by redoing the ATPE search with ATPE parameters at the 0%, 10%, 20%, 30%, and 40% percentiles. In essence, this means that each of the histories H1 through H5 are maintained with different assumptions on the "quality" of prior ATPE parameter choices. Therefore, our dataset is not biased by the assumption of predicting optimal ATPE parameters at each step of the way - we have gathered data even in cases where prior ATPE parameter choices were poor. The goal is always to choose the optimal ATPE parameters moving forward, regardless of what happened previously.

We have to tow a very careful line with this system. We are using one bayesian optimization algorithm to search for the optimal parameters for our own bayesian optimization algorithm, all on simulated parameter spaces which have built-in stochasticism. The highly stochastic nature of the complete process means that a lot of noise can be introduced into the result. Were those ATPE parameters found to be best really the best because they were better? Or were they found to be the best because random chance led them to find a better loss, even though on average they would do worse.

We did experiments for how the various parameters of our searched, hand analyzed, and made a judgement call on what parameters for our search would lead to the lowest noise. In the end, we didn't validate this well and we welcome suggestions / improvements on how we could conduct this search process better while maintaining its efficiency.

In the end, we build up a massive dataset which had various predictive statistical features, and the optimal ATPE parameters found for those features. We tried hard to hedge against many possible introductions of bias and noise in the model, to ensure that the dataset could be as applicable as possible for optimizing real-world machine learning algorithms.

You can download the dataset for yourself here:
https://drive.google.com/file/d/1xiQiwSDS5CkxUZ4wLGXL_ktjh1v-hzr3/view?usp=sharing

When you unzip the package, look at final/allResults.csv to see the full assembled dataset.

Constructing ATPE Version 2

With the dataset in hand, I now had to figure out how to use it in order to build a real-live ATPE algorithm.

There were a few concerns:

1. The dataset has an unknown amount of noise, and is expected to be highly noisy
2. The various ATPE parameters interact with each other, can thus can't be predicted independently of one another
3. The model has to execute very fast to be run in a real optimizer

We have seen a lot of people winning Kaggle contests using the tool lightgbm recently, and we decided to apply it to our problem.

We used lightgbm's powerful decision tree to train each a model to predict each ATPE parameter. When predicting any given parameter, we also include the values for all previous ATPE parameters as inputs, in addition to the statistics and features computed on the dataset. Thus, the order we which we predict them in is crucial. We tested many different orderings, but ultimately came up with the following:

1. resultFilteringMode
2. secondaryProbabilityMode
3. secondaryLockingMode
4. resultFilteringAgeMultiplier
5. resultFilteringLossRankMultiplier
6. resultFilteringRandomProbability
7. secondaryTopLockingPercentile
8. secondaryCorrelationExponent
9. secondaryCorrelationMultiplier
10. secondaryFixedProbability
11. secondaryCutoff
12. gamma
13. nEICandidates

We start by predicting the various categorical mode settings since they have the biggest impact on subsequent variables. Then we chose the remaining order using the following logic:

1. Gamma and nEICandidates get predicted last, since they are used at the end of the ATPE algorithm and thus most impacted by other variables
2. secondaryCutoff was second last, because its directly impacted by choices like the correlation exponent and probabilities
3. All other variables were ordered to reduce the number of prior variables that they used as predictive features. When ordered certain ways, certain ATPE parameters are entirely determined by the values of other ATPE parameters. We wanted as much as possible for the ATPE parameters to be derived from the statistics of the results and not from each other, with gamma, nEICandidates and secondaryCutoff the only exceptions because there are clear reasons these variables interact with the other ones

We use features in lightgbm to discourage overfitting and encourage it to use a variety of input features when making a prediction. For example, we enable both data and feature bagging.

Unfortunately, due the stochastic way that our dataset was constructed, there is a lot of noise in the dataset and an upper limit on how accurate the model can actually get. In other cases,

such as with secondaryCutoff, the predicted values gravitate towards 0 because -0.9 and +0.9 will behave in similar ways in the ATPE algorithm - locking the vast majority of variables, leading to similar seeming results, despite how different the values are.

In order to counter this effect, we calibrate the outputs of the lightgbm models. We calculate the mean and standard deviation of the original data, and we calculate the mean and standard deviation of the predicted outputs. We then recenter and rescale the predicted outputs so that they match the mean and standard-deviation of the original data.

Without calibration, our lightGBM always predicts values for secondaryCutoff between -0.15 and +0.15, even though these values are rare in the original dataset. After calibration, we have a model that effectively predicts between -1.0 and +1.0, matching the original dataset and the way the ATPE algorithm was meant to operate.

We also do the same sort of calibration with each of the categorical modes such as resultFilteringMode - rebalancing the predicted probabilities to line up with the original dataset. However, when using the categorical modes in the production algorithm, we make one additional change. We do not just take the top predicted mode - we stochastically set the mode based on the predicted probabilities. If the predicted probabilities for result filtering are Age: 70%, LossRank: 20%, Random: 0%, None: 10, then we randomly choose a mode, with a 70% chance of it being Age and so forth. This helps further hedge against the noise effects in our dataset by allowing ATPE to try different strategies if it's unsure of which one to take.

Using these predictive lightgbm models, our new ATPE algorithm is as follows:

1. Compute various predictive features and statistics based on the current result history
2. Iteratively predict all of the ATPE parameters
3. Run the core ATPE algorithm with the current result history and those ATPE parameters

Because the ATPE algorithm needs data in order to compute the predictive features, it has to be initialized for some number of rounds with random-searching before the ATPE algorithm can kick in. By default, this number is set to 10. But it can be modified - it is the only free parameter of the ATPE algorithm. We started with the original TPE algorithm with several hyperparameters of its own. We finish with the ATPE algorithm, driven by machine learning models, which requires only one free parameter, and works quite well with the default value for it.

One limitation of the ATPE algorithm is because of how it was trained, it can only work on positive loss numbers. It can not optimize negative losses, as this screws with the statistics used in the algorithm.

Testing the Algorithm

The full ATPE algorithm is implemented in the Hypermax open source library, which can be seen at https://github.com/electricbrainio/hypermax. With the final version of the algorithm, we were ready to ask the question: is it actually a better optimizer in the real world?
We tested our optimizer against the benchmarks prepared in the AutoML HPOlib. We compare ATPE, conventional TPE, and random searching. We take the average loss after doing five optimization sequences. For algorithms with negative losses, we added an arbitrary value chosen based on the global optimum of each function to the loss to make the value always positive and ensure ATPE statistics work.

Synthetic Benchmarks

First, we ran the synthetic benchmarks in the optimization test suite, with 100 trial

sequences. A lower value is better:

	ATPE	TPE	Random Search
Branin	0.4574	0.7470	0.7982
Hartmann3	-3.8324	-3.7992	-3.4585
Hartmann6	-2.7471	-2.6915	-2.4004
Bohachevsky	7.9717	31.9266	93.2973
Camelback	-1.0226	-0.9023	-0.7493
Goldstein Price	15.2121	10.8843	49.76300
Forrester	-6.0204	-6.0185	-5.9402
Levy	5.7597e-05	0.00025874	0.070922
Rosenbrock	0.94066	1.6673	5.49411

We observe that ATPE typically does better then both TPE and Random-Search, outperforming TPE in 8 out of 9 test problems. ATPE doesn't do as well on the Goldstein-Price function, having a tendency to get stuck in local minima.

To see how ATPE performed with longer searches, we ran 250 trial searches over the same benchmarks. Again these results are the average of five 250 trial searches:

	ATPE	TPE	Random Search
Branin	0.4041	0.4470	0.5870
Hartmann3	-3.8577	-3.8307	-3.7007
Hartmann6	-3.2730	-3.0014	-2.3783
Bohachevsky	1.3901	15.3091	62.5509
Camelback	-1.0308	-0.9970	-0.9978
Forrester	-6.0207	-6.0204	-6.0145
Goldstein-Price	19.256	3.934	7.4215
Levy	3.435e-06	0.0002445	0.0052066
Rosenbrock	3.18368	0.3962	1.3839

We again find that TPE performs poorly on the Goldstein Price function. Our test also showed it falling into local minima on the Rosenbrock function - something that it did not do in our 100 trial tests. More experimentation is needed to understand why.

Real World Problems

Of course synthetic benchmarks can't fully capture the complexity of a machine learning algorithm. What we care about is the optimization of real world problems.

We measured the accuracy of our algorithm on one of the machine learning problems that we had at Electric Brain. The problem is a text-extraction exercise. The model has to classify words on a word-by-word basis to pull data out of an email. The details of the problem are described in greater detail in the original research, Optimizing Optimization, under the "Text Data Extraction" heading. The metric has been modified and the core model was improved

and enhanced with additional features, making a comparison between these numbers and that papers numbers impossible. But the same basic model is being optimized.

We compare a 100 trial optimization using the ATPE optimizer versus traditional TPE:

ATPE	0.03171
TPE	0.05372

Once again, ATPE outperforms vanilla TPE in our test! We achieve better accuracy for our client project using ATPE then using the conventional algorithm. Hooray!

Conclusion

Our goal was to create a parameter optimizer that could intelligently adapt based on the situation and the data that it sees. We wanted something that was fine tuned for optimizing machine learning models and the complex interactions that they exhibit.

We came up with a basic optimization algorithm that had lots of parameters. We then rigorously searched out the optimal values for those parameters across a wide variety of different simulated hyperparameter spaces. We used this dataset to construct a machine learning model which could predict our algorithms optimal parameters, given what the results currently look like. We call the algorithm Adaptive-TPE, and its one of the first optimization algorithms that uses a pretrained machine-learning model to help it optimize faster and more accurately.

We tested our algorithm on a variety of synthetic benchmarks and showed that it improved the loss in 15 / 18 of our tests. We ran the optimizer on one of our client projects at Electric Brain, comparing it side-by-side with traditional TPE. We show the optimizer out-performs traditional TPE, producing a more accurate model.

The ATPE algorithm is implemented in the Hypermax library, which you can see here: https://github.com/electricbrainio/hypermax and is now set as the default optimizer. We have many more ideas to research and we hope to make an even better parameter optimizer in the future.

Feedback

We welcome feedback on this research! We are new to performing pure AI research and if you have any ideas on how to improve our methodology, please send them to Brad at brad@electricbrian.io. If you use ATPE to optimize your own machine learning model and are getting great results, please let us know!

Contributing

We welcome contributions to Hypermax! You can contribute in any one of several ways:

1. Contributing the results of your hyperparameter searches to our public dataset, see https://github.com/electricbrainio/hypermax-results
2. Contributing code to the Hypermax repository in the form of pull-requests: https://github.com/electricbrainio/hypermax
3. Contributing tutorials, documentation, blog articles, and other how-tos which use Hypermax
4. Sharing with us whether ATPE out-performed TPE or another form of optimization on your hyper-parameter optimization problem.

Lastly

If you have a tough AI algorithm that you need to get optimized, feel free to reach out to us at business@electricbrain.io

Recent Posts

See All

8 ways Artificial Intelligen...

Using Artificial Intelligenc...

Eight ways to avoid costly...

Log in to leave a comment.

Contact Us
+1 (888) 953-9295 [Toll Free]
(416) 342-9588 [Local]
business@electricbrain.io