Optimizing Optimization

# Table of Contents

[**Table of Contents**](#_kvxrskerbfxi) **1**

[**Introduction**](#_7t1lzfv900j8) **3**

[**Levers for Improving TPE**](#_4kl7zi1yiioy) **3**

[**Signals**](#_dylgdmxpy3ii) **5**

[Hyperparameter Knowledge](#_fsa4qz4xmoyv) 5

[Cardinality](#_3h1khzpv18ww) 5

[Independence](#_vo0dan8kj94d) 7

[Variance / Standard Deviation of Model Loss](#_k46njq47ak81) 7

[Importance of hyperparameters](#_p8pk2ypayc55) 7

[Type of hyperparameters](#_yewpdqs6xohq) 7

[Past data on this hyperparameter](#_ybvux86o2f0y) 8

[Scaling / Rounding / basic properties of hyperparameters](#_wuss040bkx4) 8

[Statistics based on current results](#_uz20wvxon4ua) 8

[Mean, Median, Standard Deviation, Variance, Percentiles of Losses](#_7pypxs3vw6pa) 8

[Skew](#_p65y6lkkjlyv) 9

[Current Best](#_89e2gfb25vwl) 9

[Hyper-parameter ranges based on Loss Percentiles](#_e0xoxhs7o6uu) 9

[Hyperparameter Correlations](#_k9q78bxktrpi) 9

[Statistics on bucketed results](#_ruqxumfsa6xi) 9

[Best Fit Shape](#_hnd222ad0e35) 9

[**The Dataset**](#_uct4gi2xskts) **11**

[Simulated Hyperparameter Spaces](#_mxat5wdebecx) 12

[Linear](#_7bvgjs4ngk5u) 13

[Hill / Valley](#_50ygcwaqw04p) 13

[Exponential & Logarithmic](#_me2s15e79bvg) 15

[Random](#_nzqkjrtejetu) 17

[Interactions](#_vr49whf092cr) 17

[Linear](#_bv266p8cp1mu) 18

[Hill / Valley](#_ezn97yuri8q4) 18

[Waves](#_u00z5ekz69yn) 20

[Random](#_qw64fmk30cws) 23

[Calculation](#_n7w3rw73wzio) 25

[Real World Datasets](#_fyg70gk1mwnm) 26

[Text Data Extraction](#_89j3qew2fnh6) 27

[CIFAR-10 ResNet](#_rtprwryeff5d) 28

[Hypermodel](#_nk713qnq5w7x) 30

[**The Experiments**](#_z09i6u9xc8l) **33**

[Measurement](#_swpihtx2jd7v) 33

[Number of trials to reach benchmark loss](#_cuey6hs93z49) 33

[Mean gain over Random Search](#_n3fchpunq07v) 34

[Mean quality of next-recommendation](#_qaruqpfxyk62) 34

[Mean loss after K-trials](#_ncp7ckrdw3f4) 35

[Decision](#_hbm6eqlq33wn) 35

[Gamma](#_dvuziq7eq32z) 35

[Default Value](#_bo8pvr41qdmg) 36

[Tuning Gamma](#_xj76ob16q0w8) 38

[Loss Metric](#_arkind54t1) 45

[Initialization Rounds](#_2vr9hhu04hf4) 51

[n\_EI\_Candidates](#_e8lzazwhruil) 54

[Prior\_weight](#_gdd1gloef0i2) 59

[Parameter Weighting](#_tjrsa3t9kd4o) 61

[Independent Distribution Modelling](#_g8vizvppvtn) 73

[**The Algorithm: Adaptive Tree of Parzen Estimators (ATPE)**](#_g4jpfaiozecf) **77**

[**Conclusion**](#_vjsfkxlzi7g) **79**

[Feedback](#_lcpve35m7bv) 80

[Contributing](#_cg1emuxr0782) 80

[Next Steps for Research](#_pgyq0226aobn) 81

[Lastly](#_6rsamkqhwyyq) 81

# 

# Introduction

Since I first started learning programming many years ago, it has been a life-long mission of mine to contribute something of value to the open-source community. I’ve used open-source software nearly continuously since I started - I should give back. Additionally, since I’ve joined the AI community, its also been a mission to contribute something useful here too. I wanted to prove that me and my firm, Electric Brain, can do genuine, original cutting edge research, invent new algorithms and approaches, and improve on current industry records.

Hyper-parameter optimization. I love it. I love the mystery that hyperparameters present. How many layers? How much regularization? What activation function will work best? Of course there is a ton of general knowledge and intuition about how to tune machine learning models available on the internet. But I’ve personally always been a fan of just spending more money on servers and compute power. Usually hiring fewer engineers and letting the computers do more of the work saves me money and gets the same result in the end.

With this strategy in mind, I’m naturally a heavy user of hyper-parameter optimization algorithms. I have found the algorithm TPE, implemented in the library Hyperopt (<https://github.com/hyperopt/hyperopt>), to be particularly effective and fast across a very wide range of different problems. I recently made my own library, Hypermax (<https://github.com/electricbrainio/hypermax>), as a wrapper around TPE. Originally it was purely just to help me generate charts and graphs from the results and make it possible to parallelize Hyperopt without using MongoDB (both accomplished). But as I began to dig more and more into the internals of Hyperopt and the TPE algorithm, I began to ask - was it possible to make TPE itself more effective? Even a 10% improvement in hyperparameter optimization can save people thousands - potentially millions of dollars spent doing parameter searches. Is it possible to achieve a new world-best in hyperparameter optimization?

In this blog post, I will try to answer this question.

# 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 to explore improving the TPE algorithm looks like this:



We will explore the details on some of these potential improvements in the main body of the article. Other improvements may be explored in follow up research.

# Signals

Although it would be absolutely excellent to find a simple, easy improvement to TPE that works well across a wide variety of use-cases, I am just as interested in knowing if there is a way to tune TPE for the search that I am about to perform. E.g. is there anything I can know about my hyperparameters that I can use to make TPE converge faster?

Additionally, we might also ask if there is a way to tune TPE while it's running. Is there anything about the distribution of results that would tell us how to tune the algorithm moving forward?

## Hyperparameter Knowledge

What do we know about our hyper-parameter space? Is there anything useful that can help guide us when we are about to perform an expensive optimization search? Are there any hard statistics that we can compute about the hyper-parameter space?

### Cardinality

The first most obvious statistic is how big & complex of a hyper-parameter space it is. We can imagine the cardinality as being the number of trials that we would need to test if we had to grid search the entire hyper-parameter space. As an example, a single parameter with rounding has a cardinality of the number of possible values between its minimum and maximum

{

“type”: “number”,

“Min”: 0,

“Max”: 10,

“Rounding”: 1

} // Cardinality of 10

If you have two hyperparameters, you now have to multiply their cardinalities:

{

“type”: “object”,

“properties”: {

“Param1”: {

“type”: “number”,

“Min”: 0,

“Max”: 10,

“Rounding”: 1

}, // Cardinality of 10

“Param2”: {

“type”: “number”,

“Min”: 5,

“Max”: 30,

“Rounding”: 5

} // Cardinality of 6

}

} // The whole object has a cardinality of 6\*10 = 60

It gets more interesting if you take advantage of the conditional searches offered by TPE. When you have a choice between two hyperparameter spaces, you add their cardinalities.

{

“anyOf”: [{

“type”: “number”,

“Min”: 0,

“Max”: 10,

“Rounding”: 1

}, // Cardinality of 10

{

“type”: “number”,

“Min”: 5,

“Max”: 30,

“Rounding”: 5

}] // Cardinality of 6

} // The whole object has a cardinality of 6+10 = 16

If your hyperparameter is uniformly distributed and doesn’t have any rounding, the cardinality of the distribution is a little bit more arbitrary, since it's technically infinite. For our purposes, we apply a cardinality of 20 to these hyperparameters. We also apply a maximum of 20 to hyperparameters that have rounding, even if their range technically is larger. This is a rule of thumb and feel free to contact us if you think there is a better way.

Overall, we can use the cardinality to let us know of the “size” or “complexity” of the hyperparameter space, and thus guides our expectation of how hard it will be to search.

### Independence

We might have a reasonable expectation that certain hyper-parameters affect the result completely independently of other hyper-parameters.

For example, in the text data extraction example discussed further down, the algorithm is built in two layers. The first layer receives a word and 1-3 of its surrounding neighbors. The second layer receives the outputs from the first layer, and the outputs of 3-5 of its surrounding neighbors. The second layer acts as a stabilizer with a larger view, but without using very much extra CPU power. I have a reasonable expectation that improving the accuracy of the first layer will pretty much always improve the accuracy of the second layer, regardless of the second layers hyper-parameters. I might be wrong, but it's not an unreasonable assumption. You might have similar assumptions about your search space.

### Variance / Standard Deviation of Model Loss

Most machine learning models are stochastic in nature, and do not converge to precisely the same accuracy every time you run them. This results in an amount of noise in your results. This is especially true if you are doing a hyperparameter search on a reduced dataset size, with a reduced running time, or in another way different from the way you will run your machine learning algorithm in production.

If you know what the typical variance / standard deviation of your model is when you train it with the same hyper-parameters, this might be a useful signal to guide us in tuning our hyper-parameter optimization.

### Importance of hyperparameters

You might know in advance that certain hyper-parameters have a bigger effect on the final loss then others. You know that these hyper-parameters are more important. You might want these hyperparameters to be tuned first, or for there to be more focus on them, during your hyperparameter search.

### Type of hyperparameters

You might know something about this hyper-parameter from your experience with it on similar machine learning models in the past. For example, the learning\_rate parameter of most neural network models using ADAM as their optimizer is expected to behave in fairly similar manner. We might be able to somehow use our prior knowledge of how this hyperparameter responds on unrelated ML models or datasets to help guide our search.

We can not directly test this since we don’t have a sufficient dataset to try and ask this question, but please feel free to contribute your hyperparameter searches to Hypermax-results (<https://github.com/electricbrainio/hypermax-results>) and we can use it as data to research in the future.

### Past data on this hyperparameter

If you are doing multiple optimization searches, it might be possible to include past results / data from this hyperparameter into your algorithm, even if you have changed other aspects of the hyper-parameter space.

We do not ask this question in this article, but we welcome people to do follow up research.

### Scaling / Rounding / basic properties of hyperparameters

On a simpler note, the fact that a hyperparameter is logarithmically scaled rather then linearly scaled might be a relevant signal to help use tune some aspect of TPE. Or the fact that it has a 4 possible values between the min and max, v.s. 8 possible values for the same min and max.

In this post, we don’t directly ask this question, mostly due to time constraints. But we welcome people to do follow up research using our data to ask this question.

## Statistics based on current results

I’m very interested in the question of whether there is anything we know about the results that we can use to tune the algorithm, while is running.

### Mean, Median, Standard Deviation, Variance, Percentiles of Losses

Computing basic statistical measures on the losses might be useful. For example, it could be useful to use them as inputs to a math equation which we pass our Loss values through. Many machine learning algorithms work better if your cost function has an asymptote around the optimal values.

### Skew

Skew can be used to know whether your results are biased towards the best results or away from the best results. This might be useful for setting trial-weights, gamma, initialization rounds or even for filtering part of the past trials before feeding them to TPE.

### Current Best

The current best result might be useful for a variety of things, such as locking in certain hyper-parameters while we explore others, or for computing how far away a typical result is from the current best result.

### Hyper-parameter ranges based on Loss Percentiles

We might be able to use percentile statistics such as “the best 10% of results have param\_1 between 1.0 and 3.0” in order to help guide our algorithm. For example, they might be used to automatically constrain the hyper-parameter space that hyperopt searches through.

### Hyperparameter Correlations

We might be able to use correlations between hyper-parameters and the loss to help use guide the search. For example, if one hyper-parameter appears to be highly correlated with loss in the first 30 trials, you might give that hyper-parameter more weight in the search moving forward, while often keeping other hyper-parameters neer a known best value for them.

### Statistics on bucketed results

We might be able to take the results from the trials, and bucket them according to the hyper-parameter. For example, if the hyper-parameter is uniformly distributed, we might create 20 ‘buckets’ between the min and the max value. We would then compute the min, median, or mean value for each bucket. We would then compute some sort of statistics across the buckets - such as their standard deviation, or kurtosis. This might be more useful then computing these values globally across all results if you are changing something about this hyper-parameter.

### Best Fit Shape

We might be to determine how the loss function typically responds to a hyper-parameter. In many of the models that I have trained, the single-parameter loss charts often take one of a few shapes. They are either complete chaos, they are roughly some sort of exponential or logarithmically shaped curve, they are linear, or they are slightly parabolic with an optimal value in the center somewhere. We might be able to determine which of these shapes provides the best fit for the current results of a hyper-parameter, and that may guide how we tune that hyper-parameter moving forward.

# 

# The Dataset

The biggest challenge with trying to improve hyper-parameter optimization is simple. How can you try to improve something when each execution of a trial can take minutes, hours, sometimes days or weeks. When I read a lot of the research on hyperparameter optimization, particularly that which is done on deep-learning, it donned on me that a lot of it was crap research. People have proven their algorithm on one or a very small handful of test-cases. And the reason is simple - running hyper-parameter optimization is the most expensive process in machine learning.

Google has been making a lot of breakthroughs by just abusing their enormous computing power. Their AutoML research is amazing - a reinforcement network that optimizes the design of a ResNET layer. But the number of trials it has to execute to do this is dizzying - 10,000, to 50,000 trials. When you read the paper and hear about “1000 Tesla K80 GPUs”, all I can think to myself is “I can never do that”. With TPE and Hyperparameter optimization in general, our goal is to get to optimal hyperparameters in say 100 to 500 trials.

If I want to be able to research hyper-parameter optimization itself, I have to be able to run many, many 100 to 500 trial sequences. But if each trial takes minutes to hours, and thus the whole sequence takes days to weeks, how on Earth am I going to test hyperparameter optimization? It would seem like you basically have to be Google to do this research.

After much thinking, I came up with the following three approaches:

1. Run hyperparameter searches over a very simple or small ML algorithm that can execute in a few seconds on one server
2. Run hyperparameter searches over a some set of math equations which simulate the behaviour of a typical ML algorithm’s loss to its hyper-parameters
3. Gather exhaustive, thorough datasets of results on real hyperparameter searches, which can be used to simulate the hyperparameters of those algorithms in a much more efficient manner.

In the end, I decided to go with the latter two approaches. I would test optimizing systems of math equations designed to respond vaguely like an ML algorithm might against its hyper-parameters. I would then validate my findings on the simulations of two real hyper-parameter searches that are small enough to search exhaustively on my budget, but still representative of the types of problems that people have.

## Simulated Hyperparameter Spaces

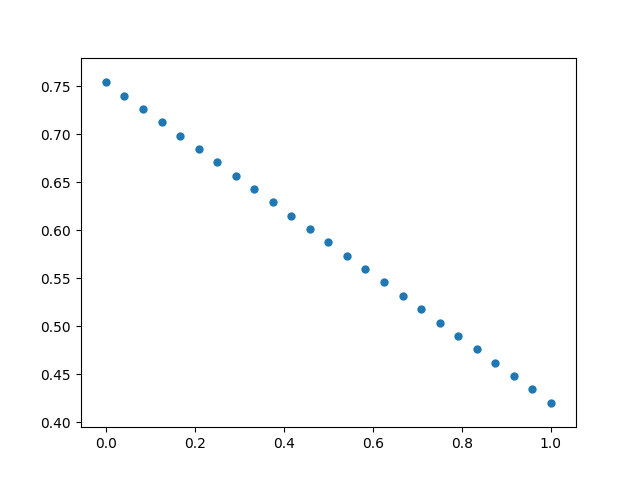
As I began to approach the question of how to simulate a typical ML algorithms hyperparameter space, I quickly became bewildered with the extraordinary number of ways that it could be done. I found myself pouring over the graphs and charts produced by Hypermax over a few of the past searches I had done, to try and discern what sort of patterns I should try and represent in my model of hyperparameters.

I needed something that was simple to create, but yet representative of the real world complexity of machine learning models.

To simplify the problem, I kept all of the hyperparameters that I would attempt to optimize between 0 and 1. They could be linearly distributed or logarithmically. They have a 50-50 chance of having a fixed interval of values or being fully uniform. If they have a fixed interval, the interval is randomly chosen between 3 and 10. No matter what, they are always be kept in the same range between 0 and 1 just to keep the headache of designing my math equations to a minimum.

Next, we define how the hyperparameters might independently contribute to the final loss. I ended having 5 different models:

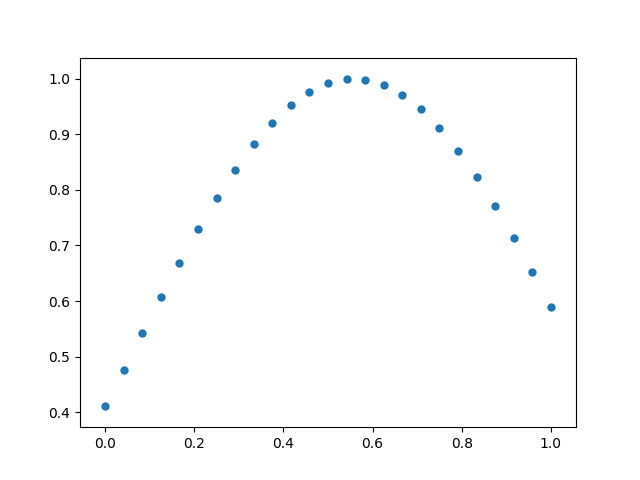
### Linear



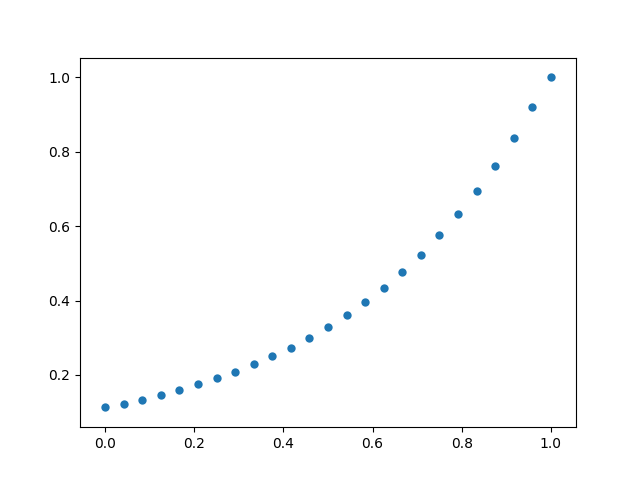
Slope and intercepts are random and can be anywhere between 0 and 1.

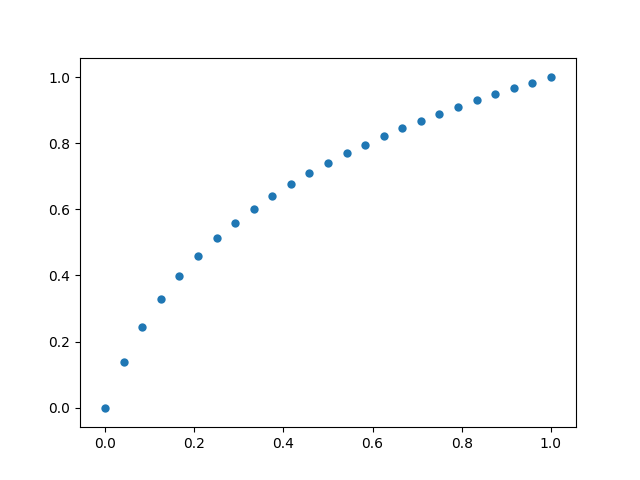
### Hill / Valley

A hill with an optimal value for the hyper-parameter or a worst value for the hyperparameter. Created using a sine curve.



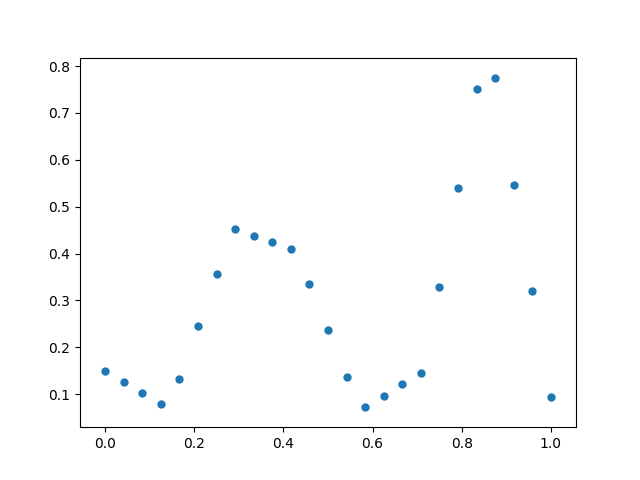
### Exponential & Logarithmic





They come in both regular and inverted forms. This means you we have curves representing all 4 possible directions.

### Random

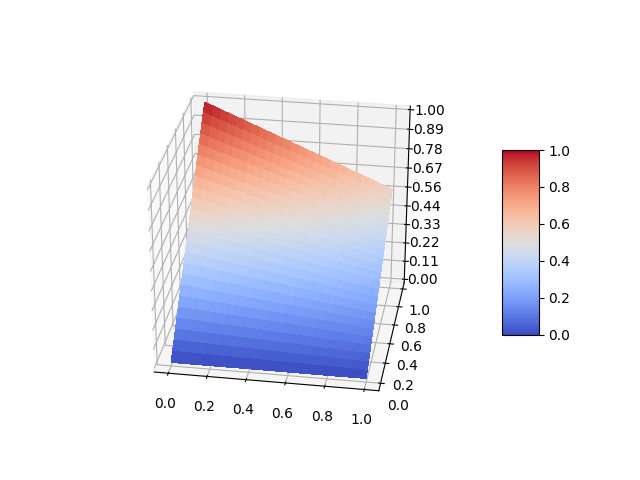


We pick between 3 and 8 random values and create a loss surface that is interpolated between them. This creates a bunch of random ways that the loss might hypothetically respond to a hyper-parameter.

### Interactions

No model of an ML algorithms hyperparameter space would be complete if the parameters didn’t interact. In order to reduce the complexity, we only allow two parameter interactions. If you want to extend our research to include three or four parameter interactions, be our guest.

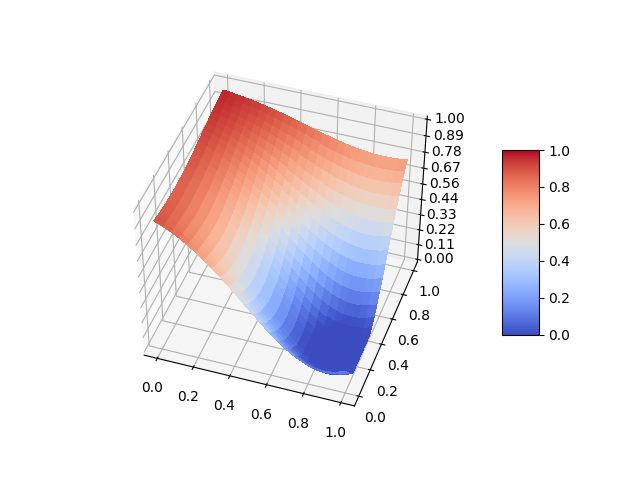
### Linear

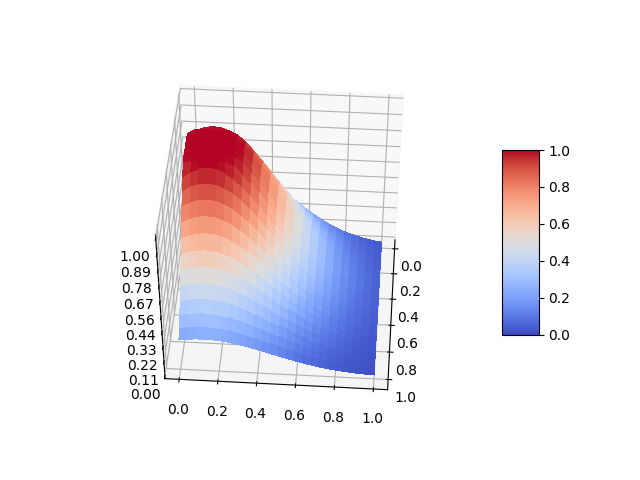


One corner is always pinned to 0, the other always to 1.0. The other two corners are allowed to float. Technically not linear since the two variables are multiplied together, it has a hyperbolic shape I think. Still a good basic interaction between two hyper-parameters.

### Hill / Valley

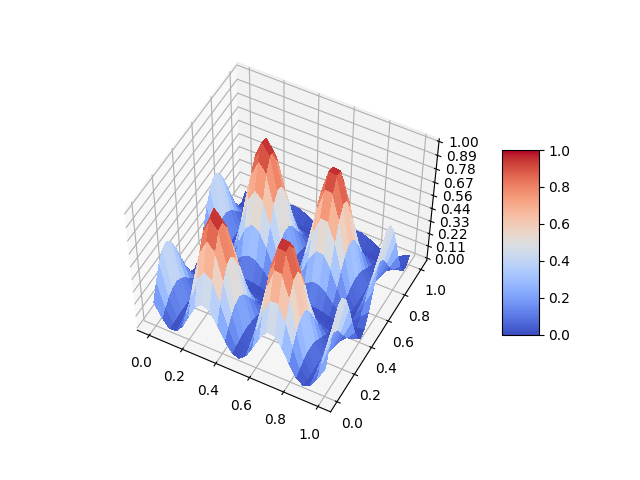
A hill implying a worst value between two parameters, or a valley indicating an optimal point. The optimal point can be anywhere in the range 0 to 1, and the hill may be flatter or steeper.

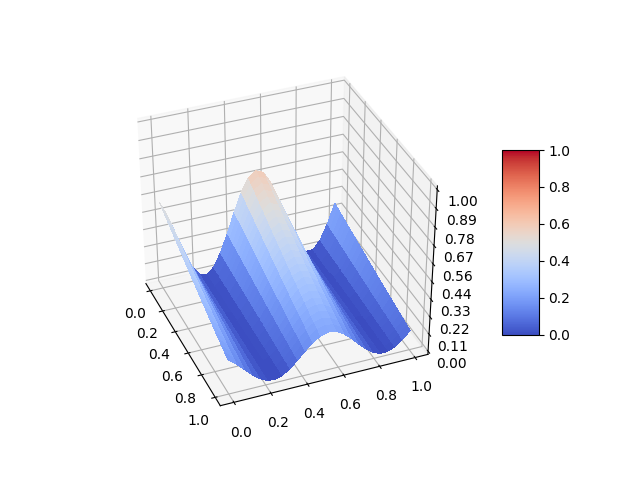


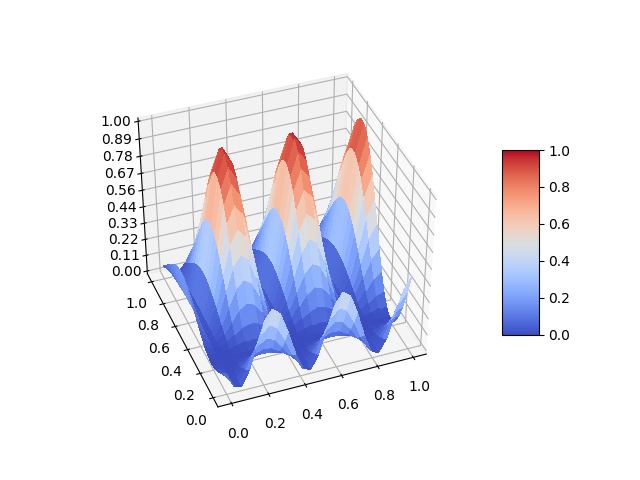


### Waves

Waves are meant to imply a repeating pattern of optimal / suboptimal points between two hyperparameters.

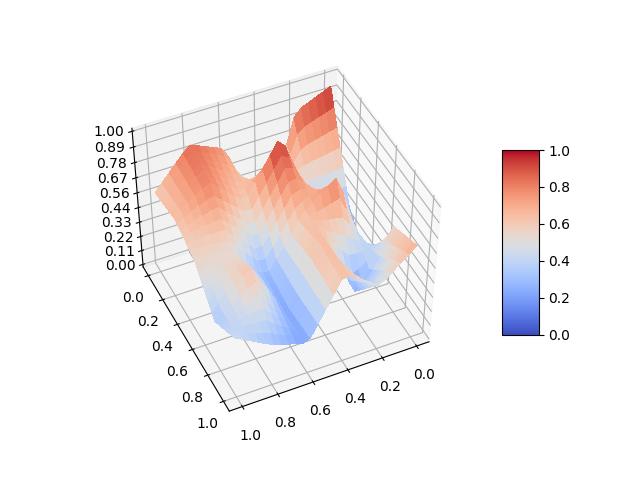


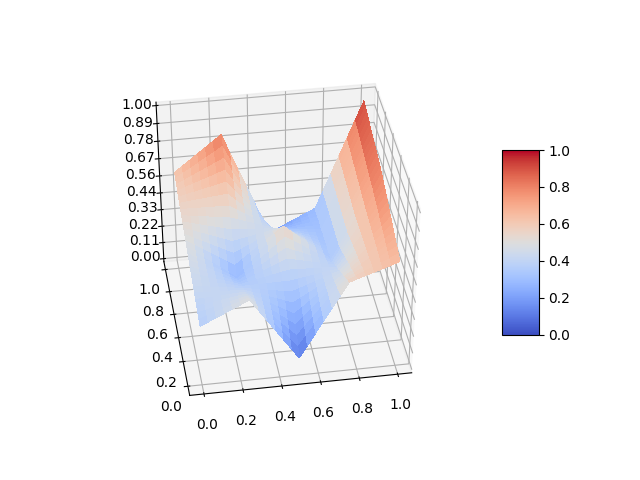




### Random

No optimization would be complete without some random interactions to make life more difficult.





### Calculation

Each hyper-parameter is given a weight to indicate its effect on the final loss. The weights are calculated by doing random.uniform(0,1) ^ 2. The squaring here helps provide a more biased distribution where certain hyper-parameters dominate the result.

Each hyper-parameter also has a weight, randomly between 0.1 and 0.4, as to how much its independent effect dominates its contribution vs. its interactions effect.

The value for each hyperparameter is calculated as follows (pseudo-code):

f(v) = ( \

independent\_effect(v) \* independent\_effect\_weight \

* sum(interaction(v, other\_v) \* interaction\_weight for other\_v in \ other\_parameters) \* (1.0 - independent\_effect\_weight) \

) \

\* parameter\_weight

We then sum up the contribution from each of the hyperparameter and voila! We have a simulated hyperparameter space.

When creating the hyper-parameter space, we don’t allow every hyper-parameter to interact with every other hyperparameter. For each pair of two hyper-parameters, we have a 30% probability that the two hyperparameters interact.

Doing this means that our simulated hyperparameter spaces have a healthy mix of interactions and independent effects, and these both have simple linear interactions, hills with optimal values, and more complex non-linear interactions such as those created by the random interactions. It’s not a machine learning algorithm, but if you were looking at the charts and graphs it produces, you could easily come convinced that it is one.

The model is not a perfect recreation of typical machine-learning hyperparameter spaces, but it's a starting point. You can see the code for the model here: <https://github.com/electricbrainio/hypermax/blob/master/research/atpe_research_1/simulation.py>. If you have a better suggestion, please feel free to reach out at [brad@electricbrain.io](mailto:brad@electricbrain.io) or contribute code to the Hypermax Github at (<https://github.com/electricbrainio/hypermax>).

The fact is, it creates complex, nonlinear functions which give our optimizer something hard to try to optimize. We can also generate many of these spaces, which will have many different resulting properties, to see if our changes to the optimizer will generalize.

## Real World Datasets

No proper research can be conducted optimizing randomly generated mathematical equations, no matter how cleverly they might be constructed to match the behaviour of machine learning hyper-parameters.

In order to validate that anything we find actually makes optimization faster, we need to be able to prove that it works on a real hyperparameter search.

To this end, we have collected two examples of real hyper-parameter searches in order to validate our findings. Both of these hyper-parameter searches were done very exhaustively over the search space, giving us a very thorough dataset to build on. The data for both searches is publicly available, and you can find it here: (<https://github.com/electricbrainio/hypermax-results>).

Side note - because of the great difficulty of hyperparameter optimization, we need your help! If you are doing a large, thorough search of hyperparameters over a machine learning or other type of algorithm, then please consider donating your results to our dataset. We will accept results both from Hypermax and in other formats from other algorithms. The key is that the search must have been very large and thorough, since we need to use the results to recreate the hyper-parameter space as accurately as possible. Help us out today!

Now I’ll know explain the two datasets I was able to gather to conduct this initial round of research on parameter optimization.

### Text Data Extraction

The first machine learning problem was that of extracting important entities / data out from text. The dataset size is very small - just 1400 examples, and the data is highly consistent. The inputs to the algorithm was a combination feature vector which included PCA-reduced word-vectors (down to dimension of 50), a character distribution, and part-of-speech. We would feed the algorithm the feature-vector for both the word being classified, and surrounding words. The output from the algorithm was a one-hot classification with 6 classes.

Due to the small size of the dataset, the algorithm itself could not be deep learning - even a pretrained network would overfit. Instead, we optimized a lightgbm (<https://github.com/Microsoft/LightGBM>) model. Our system actually had two lightgbm models.

The first lightgbm layer would take the feature vector for a word, and between 0-3 surrounding words for context, and produce its predicted classification. The second lightgbm layer would take the predictions for the word from the first layer as inputs, and additionally the predictions for between 3-5 surrounding words as context. This gives the second layer additional information, so it does improve upon the first layers prediction with a minimal increase in execution time.

The metric being used is was the Area-Under-Curve for the ROC curve (see <https://en.wikipedia.org/wiki/Receiver_operating_characteristic>). This metric goes between 1.0 (optimal) and 0.5 (no better than random chance). Anything below 0.5 and your better off betting against your classifier. In practice, the losses on this dataset went between 0.5 and 0.9. We inverted it to a cost function, so between 0.1 and 0.5.

The hyperparameter search consisted of a several of lightgbm’s major hyper-parameters over a wide range of possible values. Both layers were optimized separately with seperate hyper-parameters.

On the server I rented, the trials took about 20 seconds to run. On the computer in my office, they took between 1 to minutes. Either way they executed very fast and we were able to build up a pretty thorough search without a lot of effort.

The search itself was conducted as follows:

* Initially, we did 4500 trials over the full hyperparameter space. This gave us a pretty wide distribution of results
* We created a bunch of charts using Hypermax, and constrained several of the hyperparameters to be more narrowly focused on their optimal regions. We added another 1,000 trials to the dataset.
* Lastly, we changed the hyperparameter space back to the full space. We ran another 500 trials, but this time with TPE optimization instead of a random search. It quickly homed in on the best area of the hyperparameter space, adding another 500 trials which were mostly all within 0.02 of the best value.

In general, this hyperparameter space was fairly easy to optimize. There are a very wide range of parameter values which will get you results close to optimal. But it’s also realistic in that several hyperparameters don’t have a big impact and can act as a potential red-herring to the algorithm. Additionally, this was an actual real project that we had at Electric Brain, so it has the advantage that it was not contrived just for this research. Its fully real world.

### CIFAR-10 ResNet

Of course, the hot thing these days is deep-learning, and the difficulty of optimizing deep neural networks is part of the drive to create better optimization algorithms. So I scoured the internet to try and find a deep-learning model that would train fast enough for me to exhaustively search its hyper-parameters, but would still be representative of a typical optimization problem that a deep learning engineer faces.

I settled on basing my test on the CIFAR-10 dataset because it was well known in the ML community, there are lots of benchmarks available, and because the neural networks trained on it don’t have to be large (since the size of the images is very small). I settled on the super-convergence model created by the FastAI team (<https://github.com/fastai/imagenet-fast/tree/master/cifar10>) as my starting point. I selected the ResNet-18, the smallest one they had.

I then modified the code to introduce hyperparameters to the model. Their ResNet model was composed of repeating units in 4 different sizes, called layers in their code (but that can get confusing since they are themselves composed of layers). So I decided to do a search that was typical of the past searches that I have done on neural networks, which was mostly about how large to make the layers and how many of them.

For each of the three repeating units, I had two hyper-parameters:

Layers between 1-3

Size, varying ranges between 32-96 and 256-768

This means that effectively we are searching anywhere between ResNet-10 and ResNet-26, with varying sizes in the repeating units.

I also introduced the option to use different activation functions. The four I went with are Elu, Selu, Relu, RRelu (see <https://pytorch.org/docs/stable/nn.html#non-linear-activations-other>)

Lastly, I had two top level hyper-parameters: learning\_rate and weight\_decay. The learning rate used by default was shockingly high at 1.0, but the model converges and so I created a range between 0.1 and 3.0 just to see what’s best. The weight\_decay defaulted to 2E-4, and without looking much further into what it actually does, I just set a range between 1E-5 and 1E-3 - it seemed like this would likely contain lots of bad values as well as optimal ones. And indeed it did.

The metric being used was plain vanilla accuracy, as measuring on the testing dataset. It’s exactly what was built into FastAI’s existing validation code in that repo.

The search itself was .. frustrating to say the least. I rented 50xK80 GPUs for a few days to conduct the experiment. I was forced to fix several bugs in Hypermax and make numerous improvements to Hypermax’s ability parallelize and run models remotely. Another thing - VM images are your friend. Don’t set up your servers one-by-one. Setup one and clone it. What a waste of time that was in the beginning.

With much babysitting, I was able to conduct a search and build up a set of results in the following fashion:

* 1190 trials completed with fully random search over the full hyperparameter space
* 740 trials completed on a constrained hyperparameter space. We constrained learning\_rate and weight\_decay at the top end. We also reduced the numbers of layers and sizes of the layers for each repeating section - the results indicated that the first two repeating units needed to be larger, and more of them. The second two repeating units wanted to be smaller and less numerous. So we constrained the search space accordingly.
* 542 trials completed on a further constrained hyper-parameter space. This time pretty much every layer wanted to have a smaller layer size, but larger # of layers.
* Lastly, we returned the hyper-parameter space to the original, full sized space. We did another 114 trials with TPE turned on. With so much historical randomly sampled data, TPE immediately started sampling the best values. Actually, it only selected a single value for all of the layer sizes and would only vary the learning rate and weight decay. Surprisingly, the accuracy of the model was quite unaffected by learning rate or weight decay at the optimal point.

This hyper-parameter space might seem more challenging to optimize. The amount of the original hyper-parameter space which yields results near optimal is pretty small. But when I looked at the graphs, many of the hyperparameters had pretty clear patterns to them - larger is better here. Smaller better here - it was much less jumbled and messy then I expected. So perhaps this is a fairly easy hyper-parameter space to optimize.

Now that we have the dataset, its a permanent asset for us. You can check it out here: <https://github.com/electricbrainio/hypermax-results> and feel free to use it for conducting your own research.

### Hypermodel

Now with all of this data in hand on two real hyperparameter spaces, we need to come up with a quick way to simulate them so that we can test different ways of optimizing over the hyperparameter spaces.

The dataset sizes that we have available are still very small (3,000 to 6,000 entries), although the dimensionality of our inputs is also pretty small. We decided to go with boosted decision trees in order to form the hypermodel. We used lightgbm (<https://github.com/Microsoft/LightGBM>) in order to do this, since it fits well across many of its hyper-parameters and works well on small datasets.

In order to save time, I didn’t write any code for the hypermodel. I simply prepared training and validation CSV files using a spreadsheet program, and ran the model with lightgbm’s CLI. In the future, we plan to include code directly into Hypermax which can generate a hypermodel for you automatically.

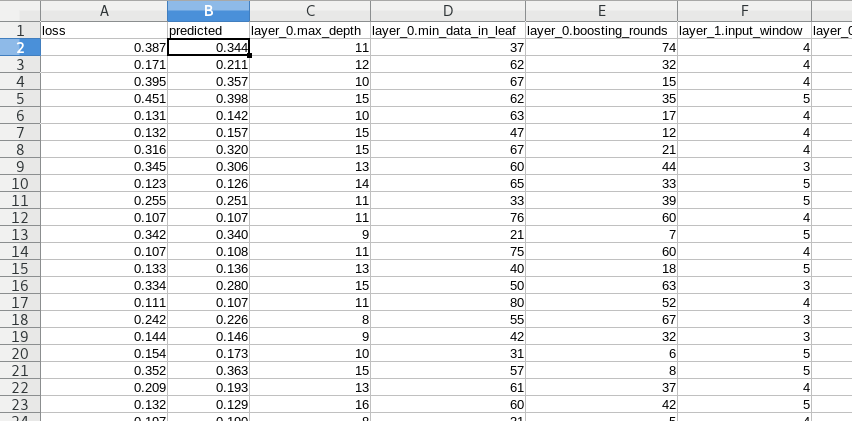
The model on our text data extraction hyperparameter space is fairly robust. We ran the model using the following command:

lightgbm data=train.csv label\_column=0 header=true metric=l1 early\_stopping\_round=5 is\_provide\_training\_metric=true objective=fair boosting=gbdt num\_iterations=1000 num\_leaves=48 valid=valid.csv bagging\_freq=1 bagging\_fraction=0.7

After creating a train.csv and valid.csv based on the result files of this model. Our training dataset had 5,000 entries and the validation set 1,000 entries. We used absolute-value metric (l1) since this is easy for us to quantify.

We tested tuning other hyper-parameters by hand but did not find many improvements - only changing the objective, setting num\_leaves and bagging helped improve the results. Ironically, we didn’t do any automated tuning on the hyperparameters of the hypermodel. We’ll do that next time. The results were pretty good out of the box. The text-extraction dataset has some failed trials with a loss set to 10.0 that need to be removed first to get good results, but after that, we converge to an average L1 of 0.0225 on the validation dataset. This means that, on average, the our hypermodel is expected to be predict within 0.02 of what the actual accuracy of our model is. Not bad, considering the trials themselves are stochastic and have some amount of variance and random error.

Here you can see some actual trial runs v.s. what our hypermodel predicted:



The hyper-model isn’t perfect, but it seems to be decent. Losses which should be high are high, losses which should be low are low. It seems like the hypermodel might actually be useful for testing our hyperparameter optimization improvements.

The hypermodel for our CIFAR10-ResNET model is similar. It’s not as accurate, but converges to an L1 loss of about 0.07. Still usable for our purposes though.

Finally, after several late-nights babysitting data collection and angrily fixing bugs after losing trial results, we have a dataset and a way of exploiting it to test different approaches to hyperparameter optimization quickly and efficiently.

# 

# The Experiments

Now we start getting onto the more interesting stuff.

## Measurement

In order to determine if our changes to the TPE algorithm made any improvement, we need a good, reliable way of measuring the accuracy of TPE. This is challenging because of the inherently stochastic nature of hyper-parameter searching. In one evaluation the algorithm may stumble onto near optimal hyper parameters very early. In another run, it may take some time to collect data for it to get to the answer. It's clear that the result needs to be an average over many runs, but precisely what metric are we averaging?

I considered many possible metrics that I could use to measure the effectiveness of our hyperparameter optimization. Each possible metric has its own pros and cons that I will outline below.

### Number of trials to reach benchmark loss

One possible way of measuring our optimization is to first measure what the optimal loss of our hyperparameter space is (somehow), and then to use that as a benchmark. We then measure the number of trials required to take our algorithm to reach the same or better then the benchmark.

**Pros**

1. Sets a high bar for the algorithm - ensures that we are measuring against globally optimal values v.s. local minima
2. Can measure the typical improvement vs. random search

**Cons**

1. Evaluating the benchmark value can take a long time (at least over our simulated hyperparameter spaces) since it must be done with random search
2. The evaluations will take a lot of trials, since the benchmark is high. In some cases, our optimizer may take more trials then the random search did in the first place to set the benchmark
3. Will need to normalize the the metric across different hyperparameter search spaces, since they will vary greatly in difficulty
4. Not clear that this is representative of what we are actually doing when we search hyper-parameters, since our metric presupposes we know what the optimal accuracy is

### Mean gain over Random Search

Another way to think about the optimization is how much better the optimizer is then an equivalent random search. Is the optimizer actually faster than grid searching? Is it potentially counter productive?

The way we would calculate this would be to run a random-search for 250 trials and run a TPE search for 250 trials. Our metric would be the difference in loss between the best result of random-search with the best result of TPE.

**Pros**

1. Executes in consistent, predictable amount of time that will never change
2. Fast to execute
3. Better normalized between different hyper-parameter spaces
4. Global minima v.s. Local minima don’t really matter with this metric

**Cons**

1. God forbid the math gods allow us to find something that improves both TPE and fully random-searches, it wouldn’t be captured by this metric

### Mean quality of next-recommendation

Yet another way we could measure the effectiveness of our optimizer is to measure the accuracy of its predictions at each and every round. E.g. given a specific trial history, what is the average loss of its very next prediction.

We could then measure this at every trial throughout a search, or at specific intervals. We would compute the mean of these values.

**Pros**

1. Fastest to execute, since we only need to run a single TPE search
2. Provides a richer amount of data for the amount of computing power needed, so less variance and faster execution time
3. As a metric, improving it would really force the optimizer to take the shortest route into a local minima (which might be good if there aren't very many local minima) rather then exploring more of the hyperparameter space

**Cons**

1. Same as above - not focused on the globally optimal result - forces the algorithm to take the shortest path into a local minima

### Mean loss after K-trials

The most straightforward way of measuring the quality of an optimizer is simply to measure what the loss is after a certain number of trials.

**Pros**

1. Fast to execute
2. Most representative of what we are actually doing when we run a hyperparameter search
3. Fairly well normalized when combining results from different hyper-parameter searches (on our searches, the losses all go between 0 and 1)

**Cons**

1. If you set K too high, algorithm may find near optimal values regardless of what changes you make to it

### Decision

In the end, we decided to go with “Mean loss after K-trials” as our evaluation benchmark. It was quick and efficient to evaluate, quick to write the code for, and highly representative of what we are trying to do in hyperparameter searching.

We decided to use the average loss of two sets of runs - one set with K=100 and another set with K=250. These are both typical numbers of trials that one might run with their hyperparameter search.

Now we can finally get to the meat.

## Gamma

The first thing that I wanted to test was TPE’s own single hyperparameter, Gamma. For those of you who aren’t familiar with the TPE algorithm, it essentially models the distribution of parameters for the vast majority of your trials, and models the distribution for your best trials. It then picks a next hyperparameter based on that which maximizes the probability of falling in your best-trials v.s. the remainder.

The question then becomes, how many trials should you consider your best trials v.s. the rest?

The way that TPE does this is with Gamma. The exact equation is as follows:

number\_of\_best\_trials = sqrt(number\_of\_trials) \* gamma

The default value for Gamma in Hyperopt is 0.25. Reading the original paper for TPE, it was clear that this value was chosen for a problem that they were working on back when the paper was published. It hasn’t changed since.

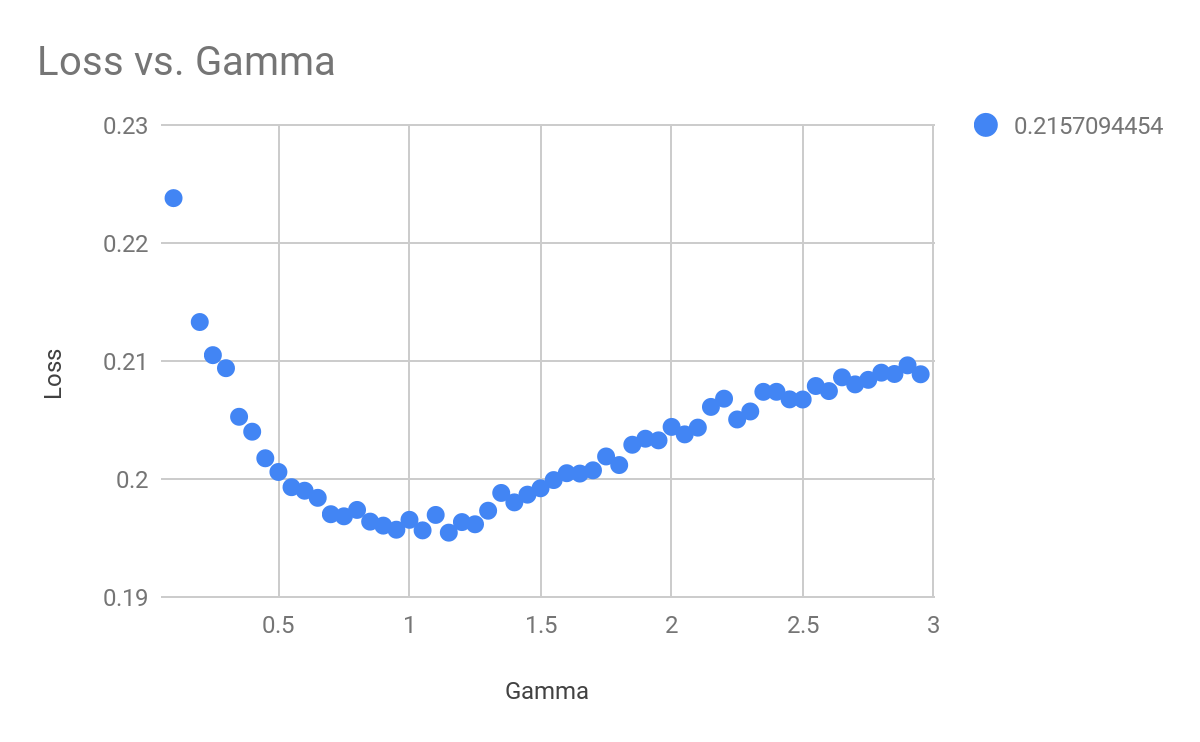
Assuming that Hyperopt is rounding up, typical values for number\_of\_best\_trials are as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Trial# | 10 | 50 | 100 | 500 | 1000 |
| number\_of\_best\_trials | 1 | 2 | 3 | 6 | 8 |

To me these numbers seemed awfully small. In typical hyper-parameter searches, there are only a handful of trials being considered the best runs. I presume the logic for using square-root on the number of trials was because finding better parameters gets harder and harder as you go along in your search. However, it's not clear to me that this will always lead to better results.

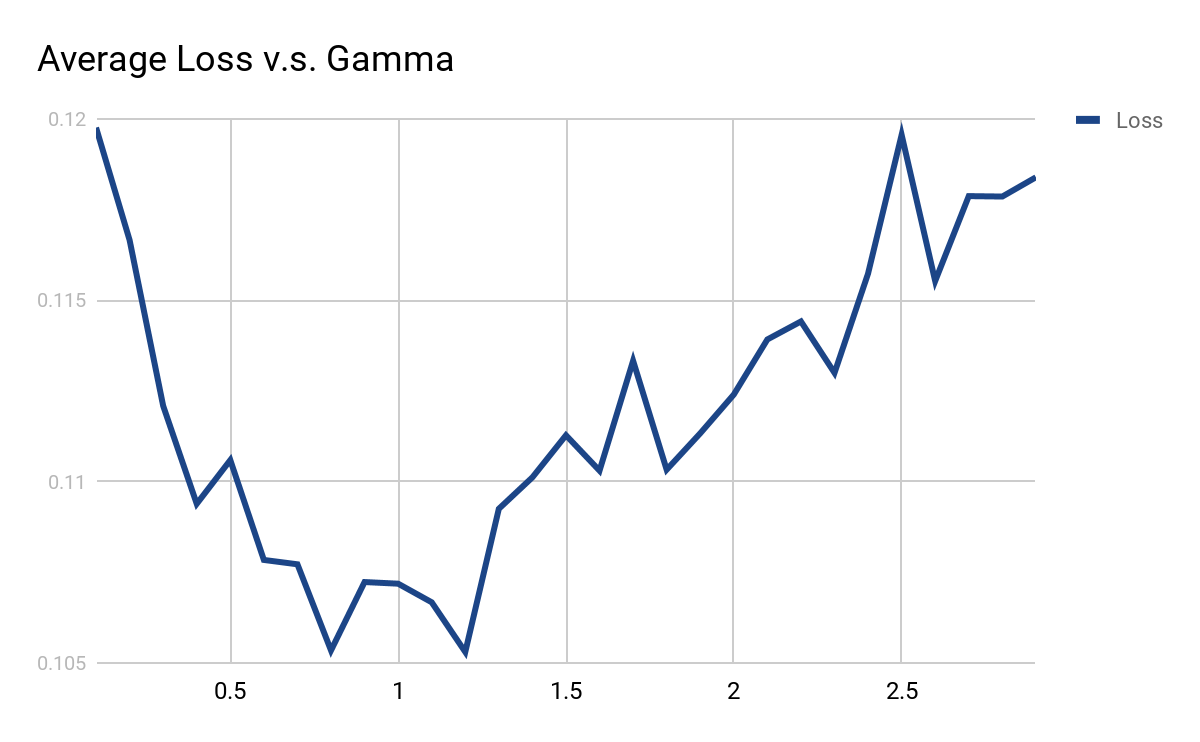
### Default Value

Our first question is: Is there a better default value for Gamma then 0.25? I figured this was a simple, easy test case to validate our approach. We generated 5 random simulated hyperparameter-spaces over which we would test different possible default values for Gamma. We then we tested various values for Gamma between 0.1 and 3. The results are shown below:



Score! It looks like our hunch was correct - 0.25 is not the optimal value for Gamma, at least over our simulated hyperparameter spaces. The optimal value appears to be 1.0

In order to validate our results, we tested potential Gamma values on our email-data extraction hypermodel. The results are shown:



The result looks very similar! 1.0 also appears to be close to the optimal value for Gamma on our email extraction problem. It does a decent amount better in the same number of trials.

At this point we are getting confident that the technique is working and might actually be leading us in the direction of better parameter optimization. It appears we can improve the results simply by changing the default value of gamma.

However, I was still wondering, can I do better? Is there a way to tune the value of gamma for each search?

### Tuning Gamma

In order to answer this question, we collected data from 439 hyperparameter spaces. We were aiming to get 500 but various factors conspired against us to prevent us getting the full dataset. For each hyperparameter space, I computed a variety of statistics, based both on the hyperparameter space itself and on the distribution of losses in the results:

* Best / Median - The best loss divided by the median loss (how much better is the best then the median)
* Kurtosis - How normal / long tailed the distribution is
* Skew - Whether the losses are skewed in one direction or the other
* Log10 Cardinality - The log10 of the cardinality of the distribution.
* Min / Mean / Median / Max / Range - Various basic stats on the distribution
* Standard deviation - The standard deviation of the distribution
* Interaction / Contribution statistics - What percentage of hyperparameters had each type of interaction / contribution pattern out of our 4 patterns for each

We also computed what the optimal value for Gamma would be for each distribution, by testing potential values for Gamma ranging from 0.1 to 2.0 in 0.05 sized increments. What we got was a CSV file containing the results. You can see it for yourself here:

<https://drive.google.com/drive/folders/1DqORAj6S5tZZFR1AA-2Cz-VkCDk6ktGY?usp=sharing>

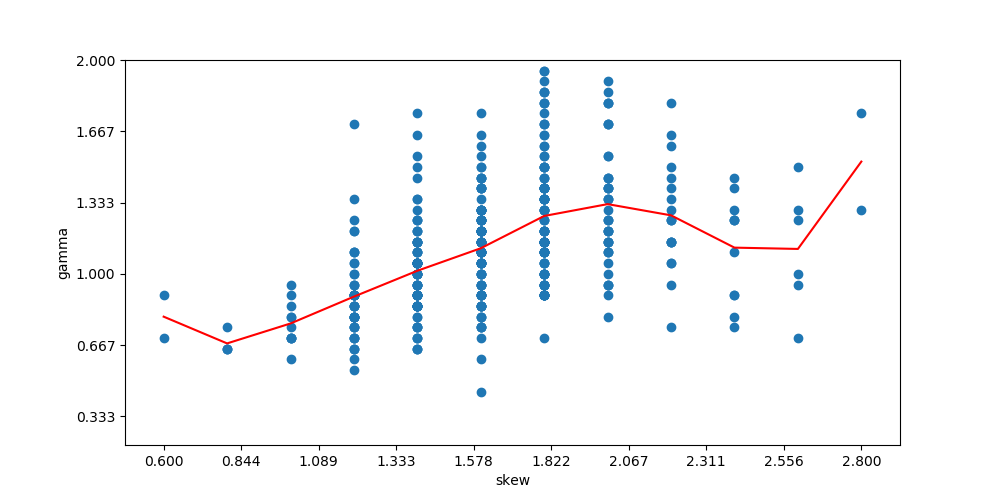
So we opened up a Jupyter notebook and started digging into the results.

The first thing I wanted to know was simply which of my various features might actually be predictive of the optimal Gamma. I computed correlation coefficients between each feature and Gamma, and I got the following:

|  |  |
| --- | --- |
| **Feature** | **Correlation** |
| skew | 0.510573 |
| kurtosis | 0.487359 |
| best/median | 0.369157 |
| num\_parameters | 0.192714 |
| interactions | 0.189860 |
| stdvsmin | -0.175513 |
| log10\_cardinality | 0.171103 |
| std | -0.166068 |
| interactions\_peakvalley | -0.141611 |
| stdvsmedian | -0.130934 |
| interactions\_wave | 0.111615 |
| contributions\_peakvalley | 0.107583 |
| contributions\_exponential | -0.051127 |
| contributions\_logarithmic | 0.045235 |
| interactions\_linear | -0.037415 |
| contributions\_random | -0.037029 |
| contributions\_linear | -0.035305 |
| interactions\_random | 0.027083 |
| interactions\_index | 0.018680 |
| range | 0.016672 |

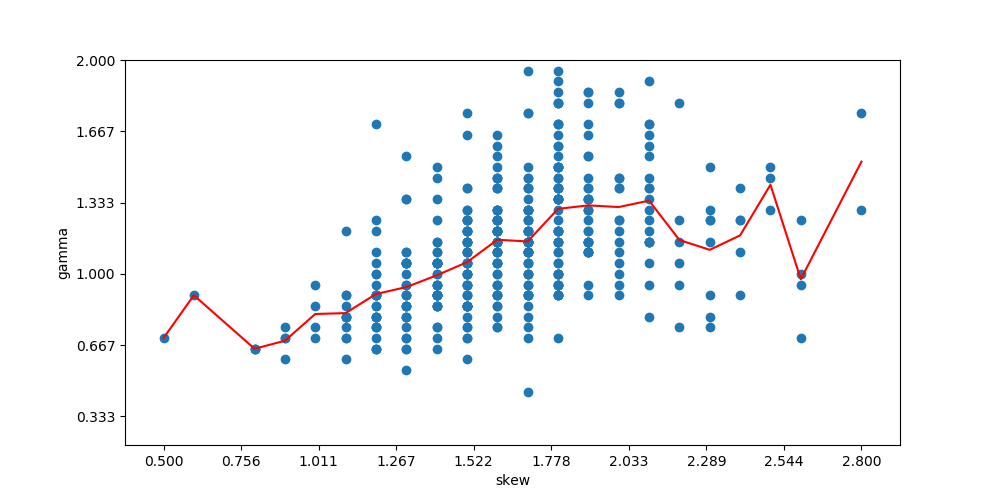
Success! Skew came up as the feature that was most highly correlated with Gamma. I am feeling very good now - I get the sense my intuitions are leading me somewhere awesome.

Plotting the chart for skew (but rounding the skew values into a handful of buckets), I observe this:

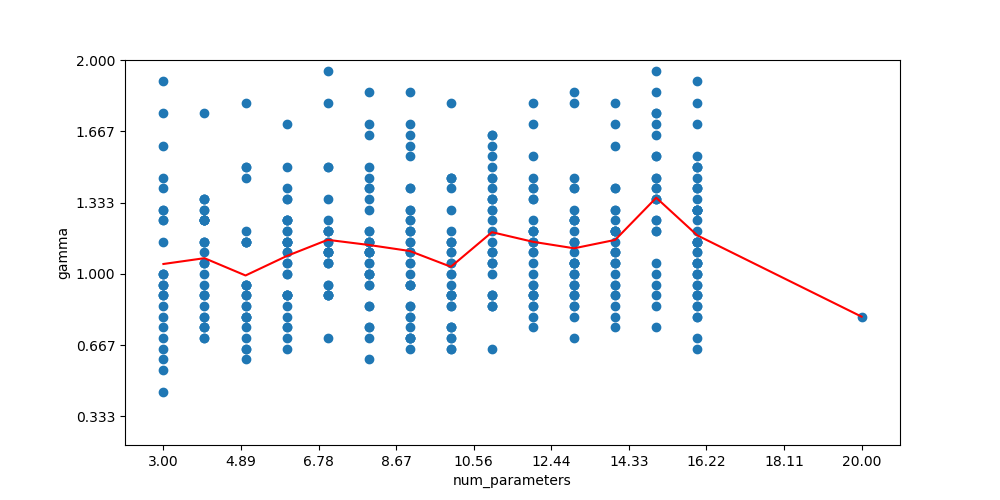


There definitely appears to be a correlation. There almost appears to be a slightly cyclical pattern to it where it changes direction for extreme values of skew.

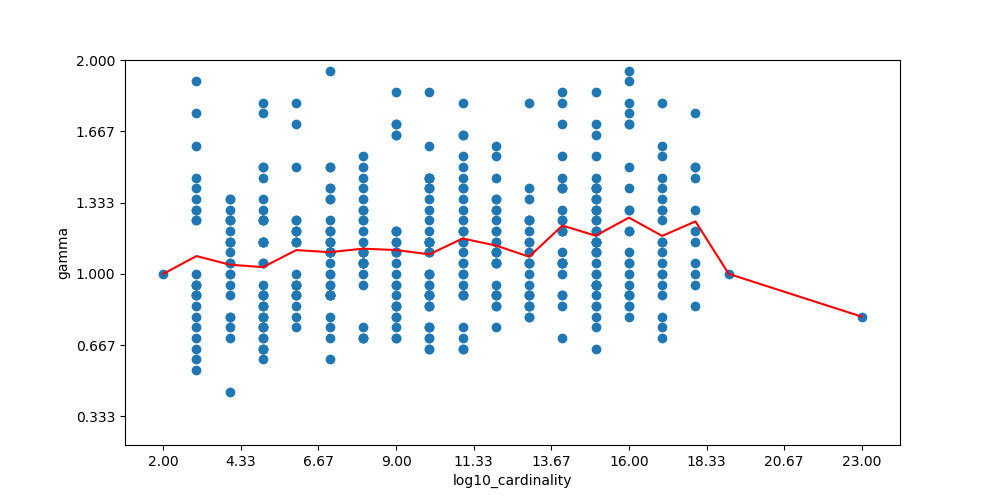
Kurtosis looks very similar. Kurtosis and Skew are related and we would expect them to show similar results.



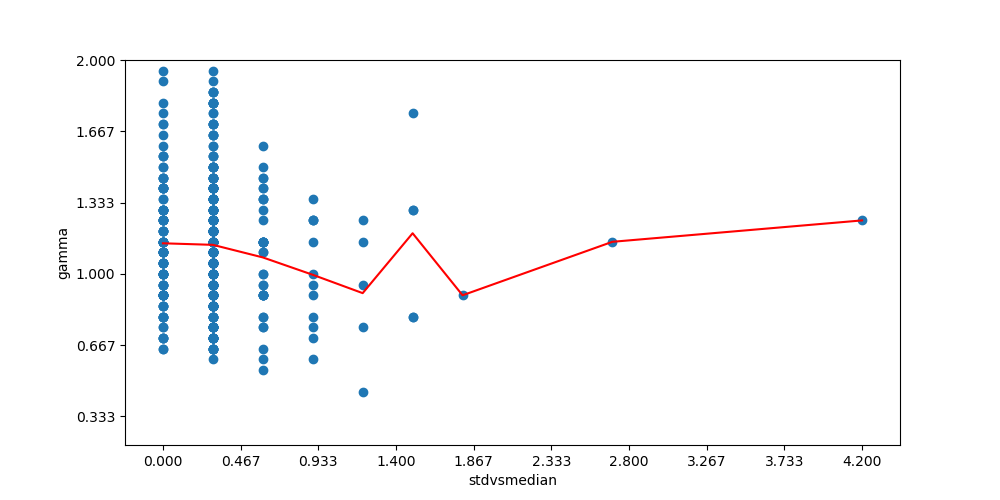
I was curious about how the number of parameters was impacting the results:



The spread on the optimal values was much much smaller. Instead of number-of-parameters, I considered the log cardinality to be a better normalized number to use as a feature, and it was similarly patterned:



The pattern was a bit more stable and consistent. Lastly, standard deviation seemed to be quite relevant. But standard deviation has a problem - it will naturally increase if you change the range of your loss values, e.g. an ML model who is measured between 0 and 1 can have the same behaviour as one measured between 0 and 100, but the standard deviation would be much larger. Therefore, I examined the impact of standard deviation divided by the median value. This would normalize the standard deviation and make it more like an index - how much variation relative to how much loss. The chart is here:



The trend is clear for the majority of values, but gets less clear in the extreme cases.

So I decided to go with skew, log10(cardinality), and standard\_deviation/median as my three predictive features. Eliminating outlier values, applying ridge regression, and rounding the coefficients to 2 decimal places, we were able to come up with a simple, easy to interpret equation for the optimal Gamma:

Adaptive Gamma = 0.23 + skew \* 0.53 + log10(cardinality) \* 0.011 + (standard\_deviation / median) \* -0.3

We call this Adaptive Gamma. Curiously, the intercept for Adaptive Gamma lies around the original default value set in Hyperopt. I wonder if they chose this value based on an unskewed distribution…

Just to ensure that Adaptive Gamma doesn’t produce any unusable numbers if your trial results are extremely weird (which can happen early on do to random chance), we put bounds on it, so the final equation becomes:

Adaptive Gamma = max(0.25, min(3.0, 0.23 + skew \* 0.53 + log10(cardinality) \* 0.011 + (standard\_deviation / median) \* -0.3))

Adaptive Gamma must be computed at each iteration, prior to running TPE. It therefore changes as the model progresses - increasing or decreasing Gamma based on how skewed and deviated the results are getting. Because Adaptive Gamma is based on statistics of the results, we need to have some number of results already before the statistics will be accurate enough to use. We call this value N. So our algorithm will return a default value of 1.0 before N trials, and the Adaptive Gamma after.

To prove our Adaptive Gamma works, we validate it against Hyperopt’s default Gamma of 0.25. We test Adaptive Gamma after 20, 40, 60, and 80 trials.

We measured the average loss for each strategy:

|  |  |
| --- | --- |
| Gamma | Loss |
| 0.25 | 0.2200 |
| adaptive\_20 | 0.2084 |
| adaptive\_40 | 0.2080 |
| adaptive\_60 | 0.2075 |
| adaptive\_80 | 0.2069 |

Now we are very excited. To further validate our Adaptive Gamma concept, we test it on our two real-world machine learning datasets, text-extraction and cifar-resnet.

On our text extraction problem, Adaptive Gamma give us these results:

|  |  |
| --- | --- |
| Gamma | Loss |
| 0.25 | 0.1097 |
| adaptive\_20 | 0.1038 |
| adaptive\_40 | 0.1054 |
| adaptive\_60 | 0.1035 |
| adaptive\_80 | 0.1019 |

Another hit for Adaptive Gamma!

On Cifar-10, Adaptive Gamma gives us this:

|  |  |
| --- | --- |
| Gamma | Loss |
| 0.25 | 0.1545 |
| adaptive\_20 | 0.1549 |
| adaptive\_40 | 0.1705 |
| adaptive\_60 | 0.1723 |
| adaptive\_80 | 0.1685 |

So Adaptive Gamma didn’t quite work on our CIFAR problem, but when you adapt Gamma early enough, it doesn’t make the results any worse.

It’s not totally clear that our formula for computing Gamma is necessarily robust, since it was computed entirely on random hyper-parameter spaces. But TPE still behaves well even with different values of Gamma. And we at least have a starting point that we can build on in further work. In the future, we may be able to tune or Gamma equation using entirely real world data which we are collecting into Hypermax-Results (<https://github.com/electricbrainio/hypermax-results>)

## Loss Metric

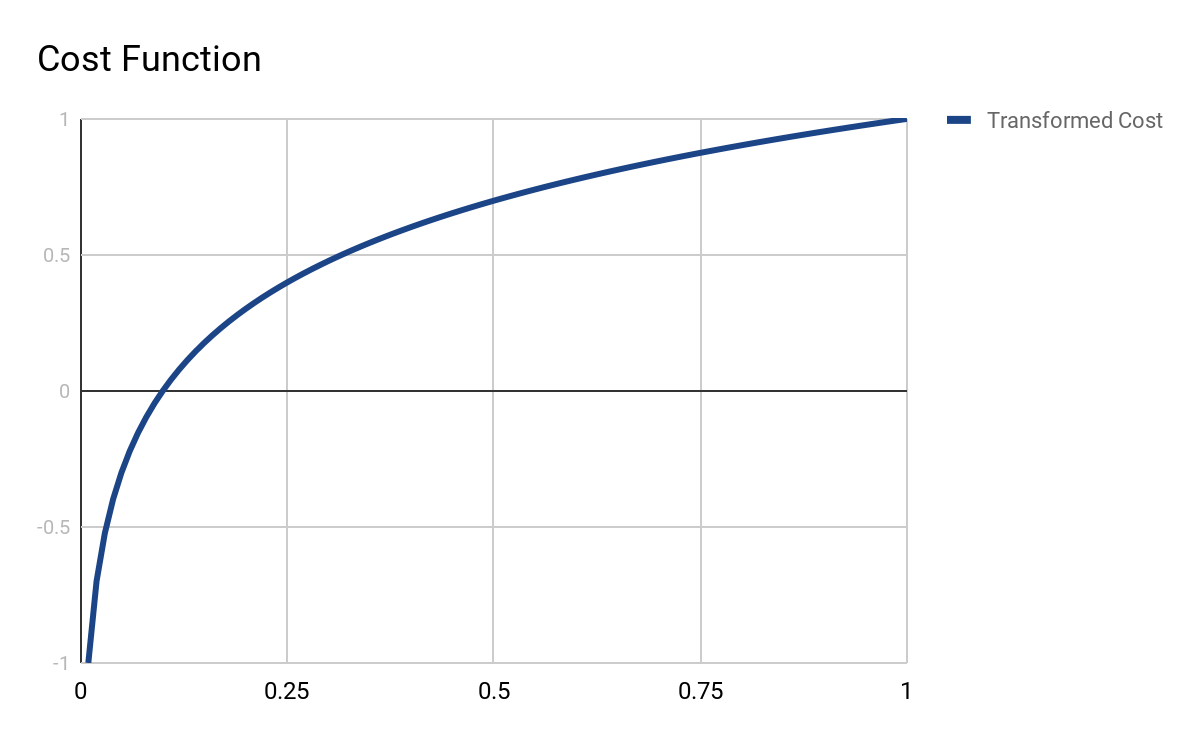
The next low-hanging fruit that we wanted to test was the loss metric. For many machine-learning algorithms, when you are running hyper-parameter optimization, you are optimizing for your highest level, final evaluation metric. This might be something simple like Accuracy, or more complex like Area-Under-Curve. For accuracy, this is a value between 0 and 100%, and for Area-Under-Curve, this is a value between 0.5 and 1.

Many machine learning algorithms will work better if their bounds go between [-infinity, +infinity] instead of a fixed domain like [0, 100%]. The intuition is that, as the algorithm gets started, a big gain like 10% improvement in accuracy is very relevant. As it gets closer and closer to a perfect accuracy, it will give less and less weight to those little gains. So instead, if you transform your cost function by doing, say, log10(cost), then no matter how close your cost gets to 0, there will always be a big gain possible, from the algorithms perspective.

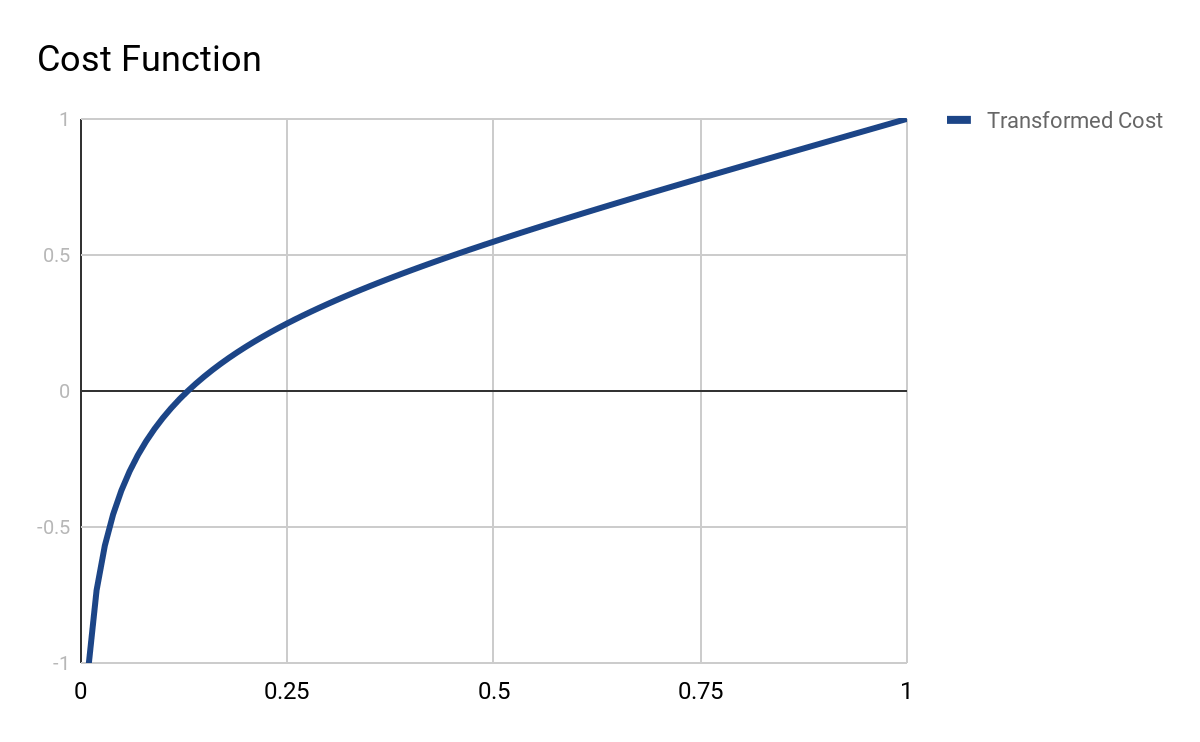
Of course this is all conjecture. Will TPE respond in this way? The only way to find out is to test.

In order to test this, I wanted to come up with a way of transforming loss values that would work with any bounded metric, no matter what the range, distribution, or optimal loss might be. Something that might work in the general caes, on any bounded metric.

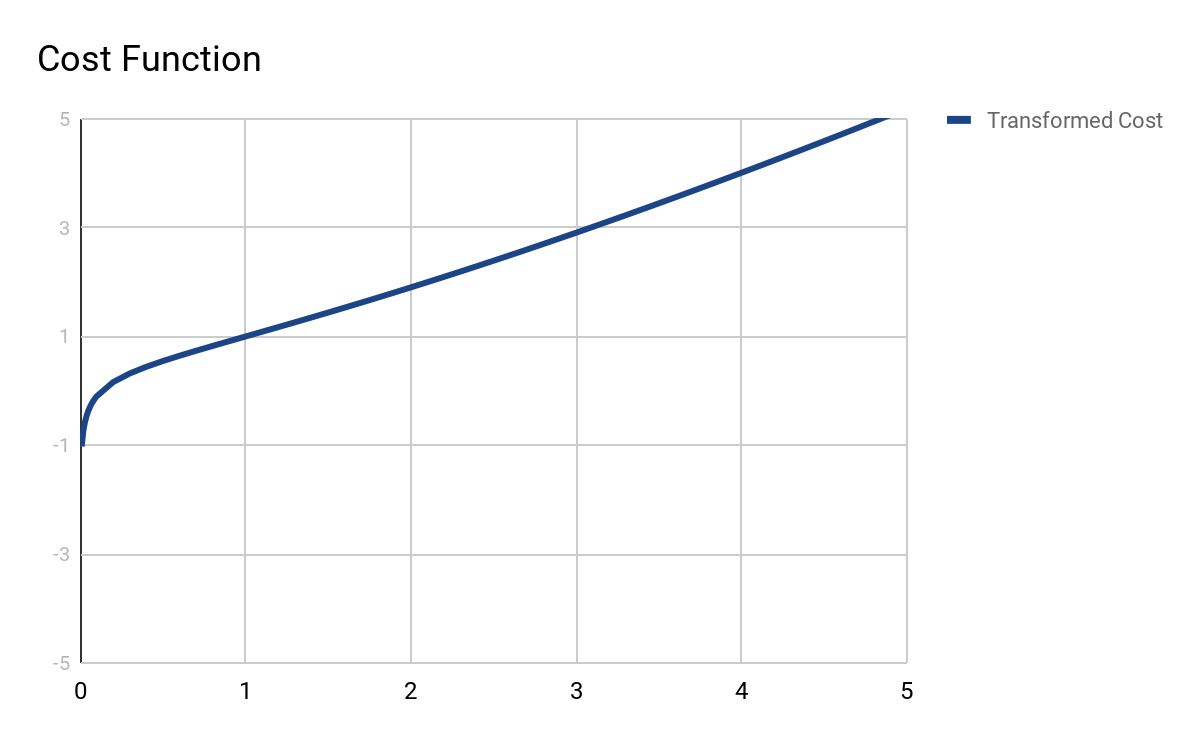
Logarithms produce a nice steep cost function which goes to infinity at 0. But they water down the knowledge of higher cost values. This is log10(x)+1.0 plotted below:



In order to compensate for the watered down gradient at high loss values, I decided to add in an additional term preserving the original cost function gradient. This is (cost+1.0) \* log10(cost) + 1.0:

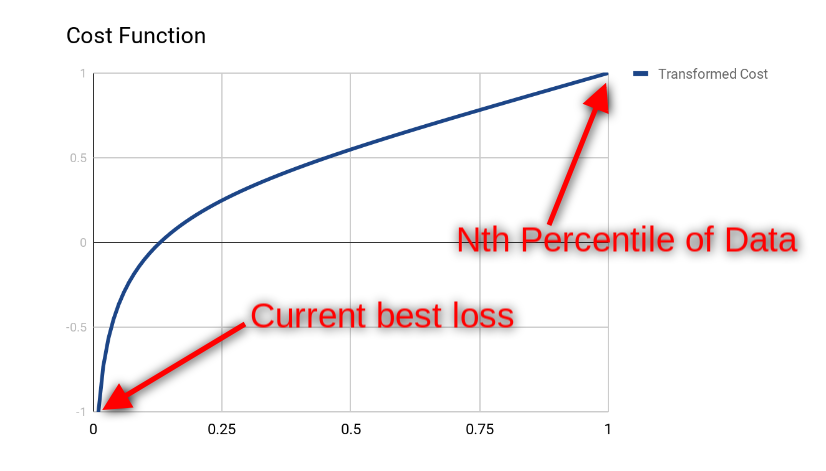


And you can see it still behaves nicely as we zoom out:



So now we’ve transformed our bounded metric into an unbounded one with an asymptote at the optimal cost. However it occurs to me that if your metric has a wider range than 0 and 1, then your still getting essentially a linear cost function. The question is: how to balance the linear and logarithmic terms? Where do we put the asymptote?

There are many ways of answering these questions, but my intuition lead me to the following logic. I wanted the exponential response of the cost function only on the top-performing area of the results. The steepest point of the exponential response should be the current best. The good area should be curved. The rest of the cost function should be roughly linear. The asymptote should appear to be some point just in-front of the current best loss. So my logic looks like this:



It wasn’t clear to me what the Nth percentile should be, but I wanted the current best loss to be nice and snug up to the asymptote. Time for some algebra. We need to create a line that transforms our metric to values so that the current best =0.01 and the nth percentile =1

y(x) = m\*cost+b

0.01 = m \* best + b

1 = m \* percentile + b

-0.99 = m \* best - m \* percentile

-0.99 / m = best - percentile

M = -0.99 / (best - percentile)

M = 0.99 / (percentile - best)

Plugging in for b gives us:

0.01 = m \* best + b

0.01 = (0.99 / (percentile - best)) \* best + b

b = 0.01 - (0.99 / (percentile - best)) \* best

b = 0.01 - 0.99\*best / (percentile - best)

And so our full line equation becomes:

y(x) = (0.99 / (percentile - best))\*x + 0.01 - 0.99\*best / (percentile - best)

y(x) = (x - best) \* (0.99 / (percentile - best)) + 0.01

Now we can take this equation and plug it into our cost function as follows:

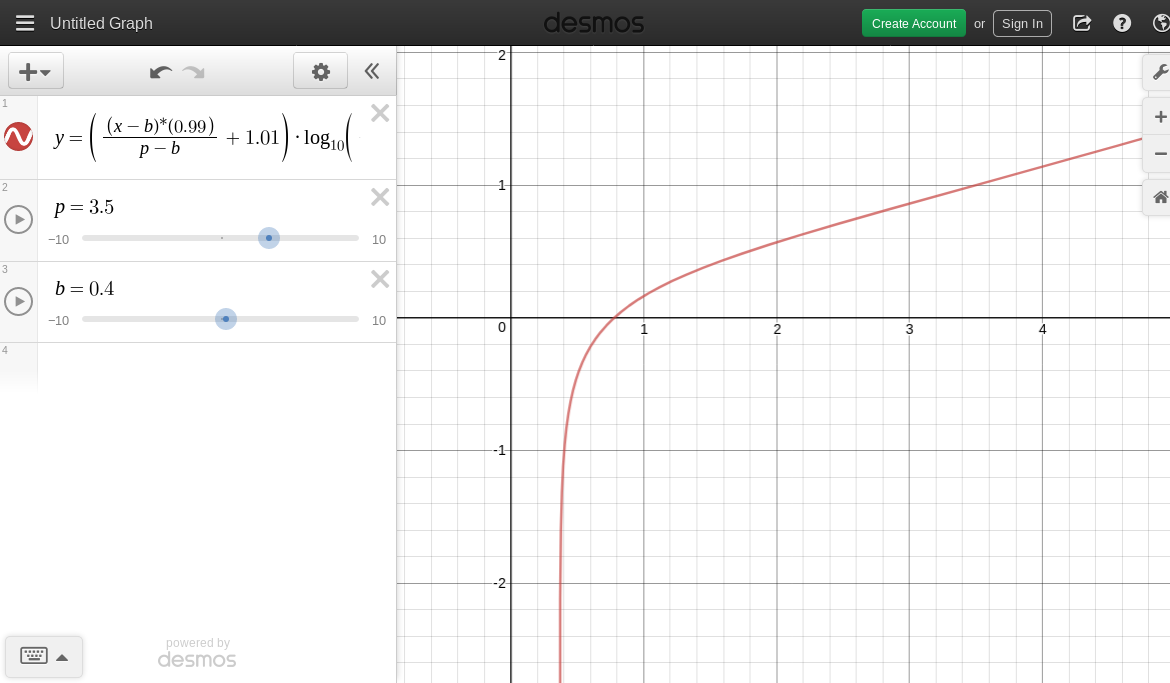
cost(x) = (x+1) \* log10(x) + 1

cost(x) = ((x - best) \* (0.99 / (percentile - best)) + 0.01 +1) \* log10((x - best) \* (0.99 / (percentile - best)) + 0.01) + 1

cost(x) = ((x - best) \* (0.99 / (percentile - best)) + 1.01) \* log10((x - best) \* (0.99 / (percentile - best)) + 0.01) + 1

Whooh! Thats a doozy. Plotting it in Desmos allows me to play with the numbers and confirm the equation is performing the way I want. Try it yourself here:

<https://www.desmos.com/calculator/3rtln3egfn>



The idea here is that on every round, you compute the current best and some nth percentile, and transform the results history using this math equation before passing it into TPE. You preserve the original loss values so that you can update your transformation equation at the next step.

In this manner, the algorithm always has a view of your hyper-parameters with a very steep loss curve around what it currently considers the optimal value. We call this technique an Adaptive Loss Function (At K Percentile). Now the question is, will this actually work? Let's test it and find out.

To test this, we generated 1,000 random hyperparameter spaces. For each hyperparameter space, we computed a result with a variety of loss functions applied.

The various loss functions we tested:

* The original loss, unmodified
* Adaptive Loss Function (At 5th percentile)
* Adaptive Loss Function (At 10th percentile)
* Adaptive Loss Function (At 25th percentile)
* Adaptive Loss Function (At 50th percentile)
* log10(loss) + 1.0
* sqrt(loss)
* 10^loss

The results were … very interesting. You can see them for yourself here:

<https://drive.google.com/drive/folders/1DqORAj6S5tZZFR1AA-2Cz-VkCDk6ktGY?usp=sharing>

First, I computed the best loss function for hyper-parameter space, and then counted up the totals. What I got was this:

{'adapt10': 109,  
 'adapt25': 103,  
 'adapt5': 104,  
 'adapt50': 119,  
 'exponential': 110,  
 'identity': 105,  
 'logarithmic': 120,  
 'squareroot': 96}

Ok, looking pretty even. Even the ones I tossed in as red-herrings, exponential and square-root, turned out to work about as well as all the others ones.

Not fully convinced, I measured the percentage of trials each loss function outperformed the identity function.

{'adapt10': 0.48036951501154734,  
 'adapt25': 0.5127020785219399,  
 'adapt5': 0.5057736720554272,  
 'adapt50': 0.4953810623556582,  
 'exponential': 0.5,  
 'logarithmic': 0.49191685912240185,  
 'squareroot': 0.4930715935334873}

Ok it’s now looking exactly like random-chance - there is no discernible pattern here. For extra understanding, I compute the mean loss for each loss function.

{'adapt10': 0.20638841049856826,  
 'adapt25': 0.20653561869378104,  
 'adapt5': 0.2064733937191374,  
 'adapt50': 0.2064599203267815,  
 'exponential': 0.2065021215922073,  
 'identity': 0.20652949885776323,  
 'logarithmic': 0.2063690821730577,  
 'squareroot': 0.2064465462948599}

At this point I am now pretty convinced - the shape of your loss metric has absolutely no effect on the results. All that matters is the rank-order of your results - not the actual loss values themselves. The result is disappointing. Perhaps someone with a better understanding of Bayesian math could have told me that this was going to be the case. It's still good to actually observe it with my own eyes. This also gives me more confidence that my measurement statistic is good and will tell me clearly something doesn’t work.

## Initialization Rounds

Now we move onto testing initialization rounds. The TPE algorithm tries to predict which hyperparameters are most likely to perform the best next. In order to do this, it needs to have some prior history.

By default, Hyperopt will run the first 20 trials using a fully random search, prior to engaging TPE. In the original paper, they used a value of 50. Is this the right number? How much random data does TPE need before you should let it start exploring?

This was a relatively straight forward question to answer. We generated 500 random hyperparameter spaces. We tested 10 different values for Initialization rounds:

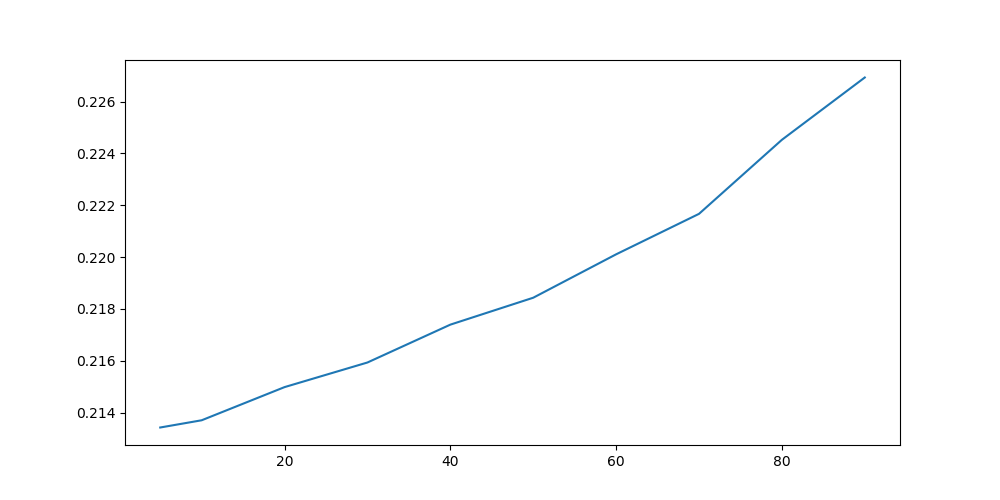
* 5, 10, 20, 30, 40, 50, 60, 70, 80, 90

Like usual, for each possible value of initialization rounds, we took the average of 20 searches. 10 searches performed with 100 trials, 10 searches performed with 250 trials. Therefore, for the 100 trial searches, our random initialization rounds vary from being 5% to 90% of the total number of trials. For the 250 trial searches, it varies from 2% to 36%.

We measured the average loss for each number of initialization rounds. You can find the results here:

<https://drive.google.com/drive/folders/1DqORAj6S5tZZFR1AA-2Cz-VkCDk6ktGY?usp=sharing>

The chart is given below:



The result was pretty unambiguous - you were better off letting TPE do most of the exploring of your hyperparameter space, without much random initialization.

We tested for potential correlations with statistics from our hyper-parameter space. Not much interesting popped up:

|  |  |
| --- | --- |
| feature | correlation |
| interactions\_wave | 0.13411159914248 |
| interactions\_linear | -0.116453444833463 |
| skew | -0.0965986055741496 |
| kurtosis | -0.0889620840182861 |
| interactions\_index | 0.0804269831693945 |
| num\_parameters | -0.0789898658398029 |
| std | -0.0789010528918232 |
| interactions | 0.0775623634266413 |
| stdvsmedian | -0.0762617002706139 |
| stdvsmin | -0.0713804546718236 |
| log10\_cardinality | 0.0516808004155953 |
| contributions\_logarithmic | 0.046584736728033 |
| contributions\_linear | -0.0463347590698333 |
| contributions\_peakvalley | 0.0458547806906478 |
| range | -0.0390044111702334 |
| best/median | 0.0152911956261636 |
| interactions\_peakvalley | -0.0108366707484515 |
| interactions\_random | 0.00808105903891968 |
| contributions\_exponential | 0.00403376864921586 |
| contributions\_random | -0.00299469395494006 |

At this point I feel pretty comfortable decreasing the number of initialization trials from 20 to 10 as a default value for our algorithm. Concerns about falling into local minima without enough initialization appear to be unfounded.

Lets test on our real world datasets just to confirm. Text extraction:

|  |  |
| --- | --- |
| Initialization Rounds | Loss |
| 5 | 0.1087 |
| 10 | 0.1071 |
| 20 | 0.1077 |
| 40 | 0.1083 |

And CIFAR\_ResNet:

|  |  |
| --- | --- |
| Initialization Rounds | Loss |
| 5 | 0.2377 |
| 10 | 0.2101 |
| 20 | 0.2271 |
| 40 | 0.2319 |

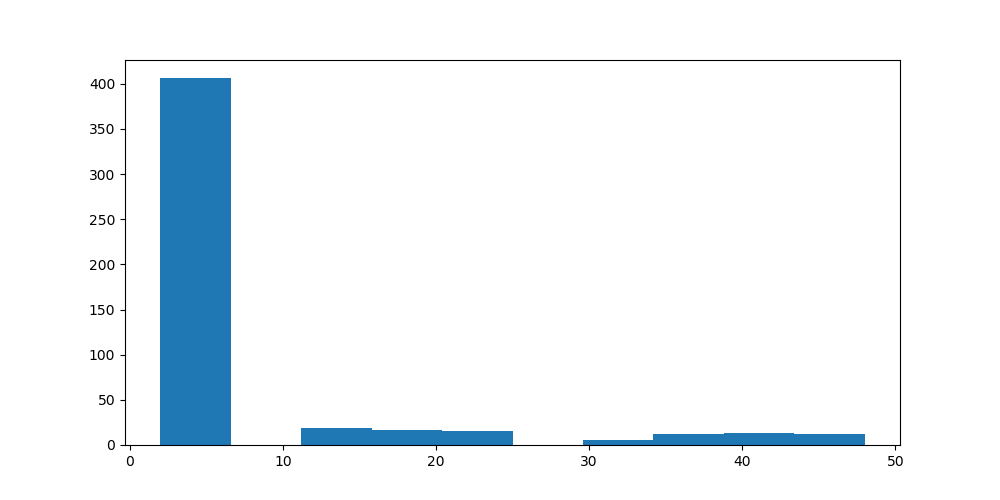
We have found a simple, effective way to improve the results from TPE. Simply do fewer initialization rounds, just not too few.

## n\_EI\_Candidates

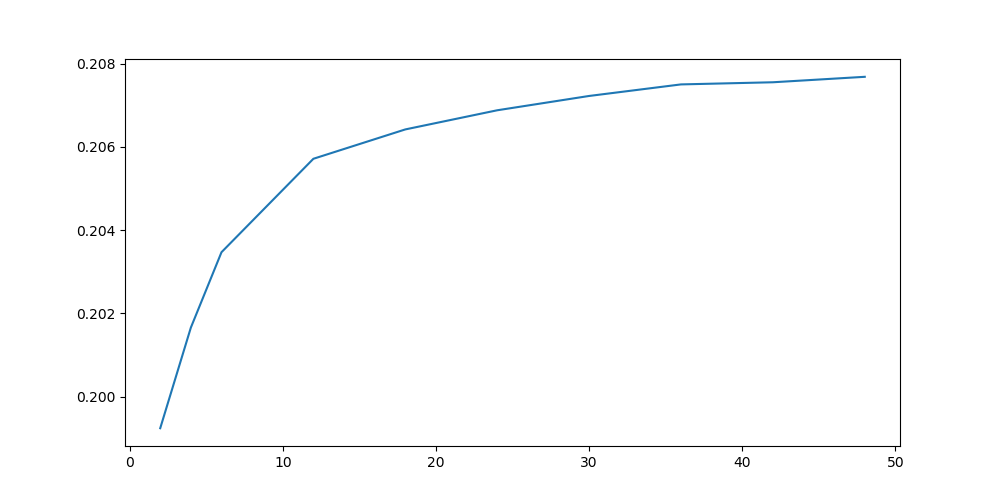
Another one of TPE’s own hyperparameters is the n\_EI\_Candidates variable. Essentially, when we want to make a recommendation for the next hyperparameters to test, TPE will come up with several candidate hyperparameters, calculate their predicted gain, and give you the recommendation which has the best predicted gain.

My intuition suggests to me that having a lower value for n\_EI\_candidates will make the algorithm more chaotic. Having a high value for it will make the algorithm more narrowly focused on what it currently thinks is best.

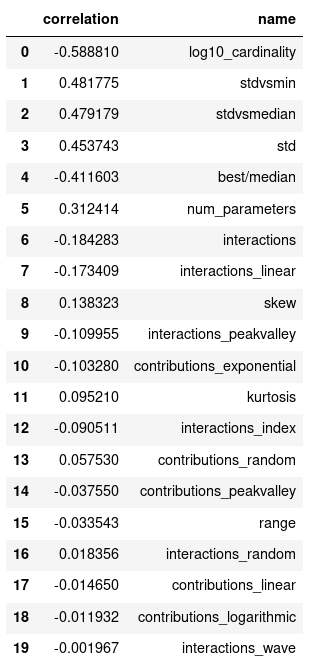
The default value within Hyperopt is 24. Let’s find out if this is optimal. I think the Histogram is particularly illuminating:



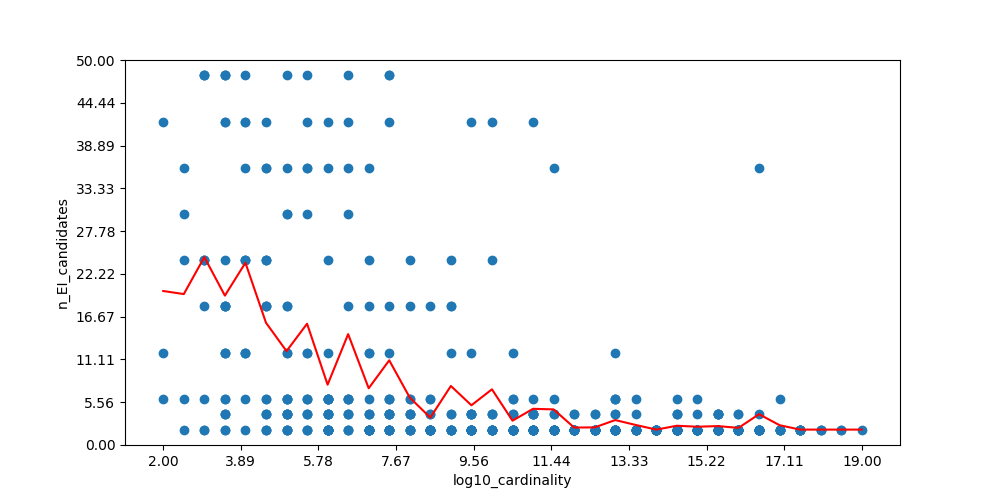
It appears that the TPE algorithm performs better when you have a lower value for n\_EI\_candidates. We plot the average loss at each value of n\_EI\_candidates:



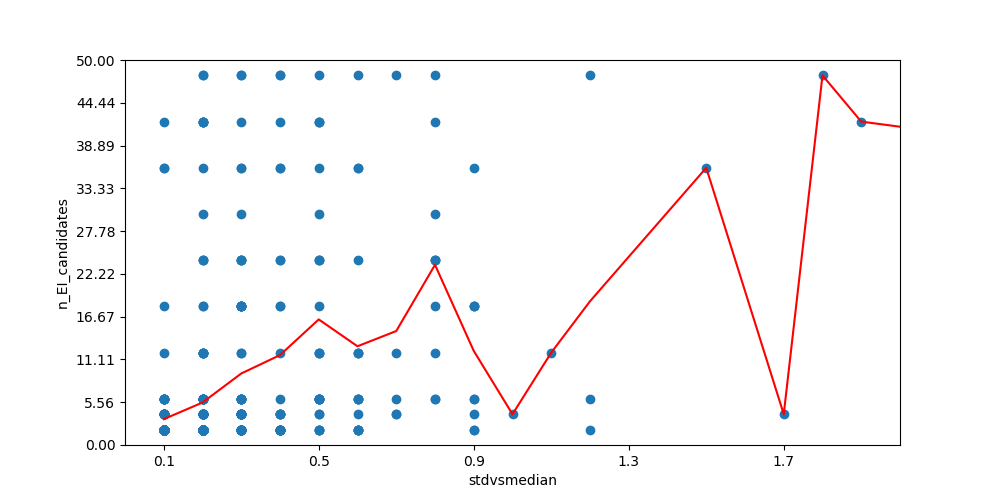
Yup - looks like we have a clear pattern here. Like usual, we calculate correlations to figure out any interesting patterns:



It looks like we have another hit! Both correlations on the size of your hyperparameter space and on the standard deviation of your losses seem to be highly correlated. Lets view the charts:

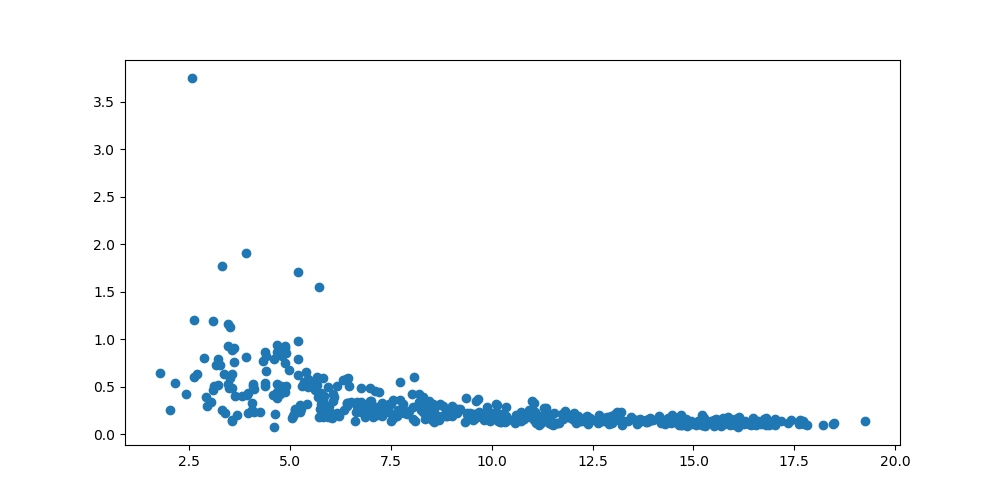


n\_EI\_candidates has a very clear response to the cardinality of your hyper-parameter space. Let’s take a look at stdvsmedian:



Very good, looks like a response in the opposite direction. It seems like we might be able to make an adaptive function for n\_EI\_estimates as well!

I go ahead and start trying to fit a curve to our two feature variables. The features both appeared to have an exponential response, so I tried to fit two exponential curves. But I keep running into problems - my curve fitting function seems to be minimizing the effect of one variable or the other - it would force one to fit basically linearly while the other fit exponentially. Frustrated, I began to believe that my two variables are responding precisely in opposition to each other. I plot the scatter to find out:



Yup, look at that. There is already a mildly exponential relationship between my two feature variables. This makes sense to me - as you increase the number of hyperparameters, you get more and more messy relationships, reducing any clear single on the loss from the hyperparameters and thus the variance of those losses.

I eventually resign to simply using the hyperparameter space cardinality as my main predictor variable to tune n\_EI\_estimators. I fit the curve and end up with the following clean and simple formula:

n\_EI\_candidates = 1 + 40 \* 10 ^ (-0.1 \* log10\_cardinality)

We add in boundaries just to make sure it doesn’t behave badly if you have a weird hyperparameter space.

n\_EI\_candidates = max(2, min(50, 1 + 40 \* 10 ^ (-0.1 \* log10\_cardinality)))

Now it's time to see if our rule of thumb actually works! We compare our rule of thumb to the TPE default value of 24. We ran the simulated hyperparameter spaces again, to compare our tuned n\_EI\_candidates value to the default of 24:

|  |  |
| --- | --- |
| n\_EI\_candidates | Loss |
| 24 | 0.2032 |
| adaptive | 0.2026 |

Looks like we have an improvement! To further validate our tuning function for n\_EI\_candidates, we measured the loss on our text-extraction dataset.

|  |  |
| --- | --- |
| n\_EI\_candidates | Loss |
| 24 | 0.1089 |
| adaptive | 0.1078 |

Looks like we are good! Not a big improvement but at least we have something. How about cifar\_resnet:

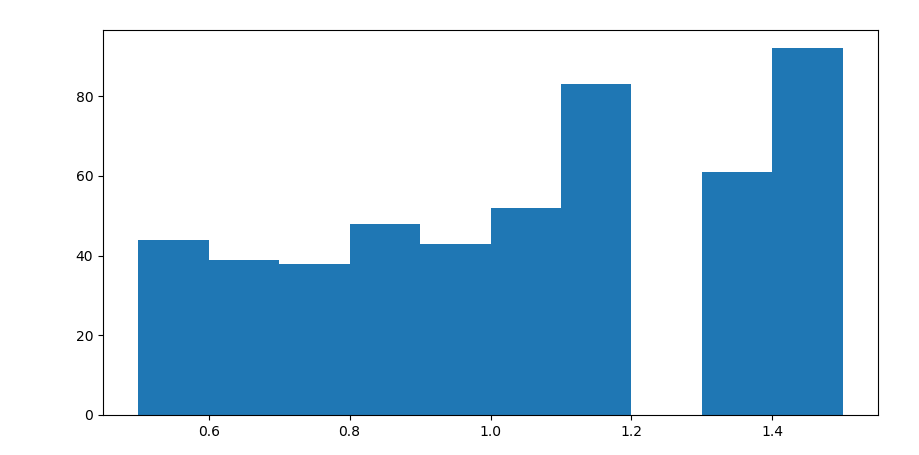
|  |  |
| --- | --- |
| n\_EI\_candidates | Loss |
| 24 | 0.1565 |
| adaptive | 0.1535 |

Excellent! We have another confirmed improvement over vanilla TPE (albeit a small one).

## Prior\_weight

It wasn’t totally clear to me what this parameter does or what effect this hyperparameter would have on the TPE algorithm. The default value was 1.0, so we just tested values between 0.5 and 1.5 to see if there was any change in behaviour from TPE.

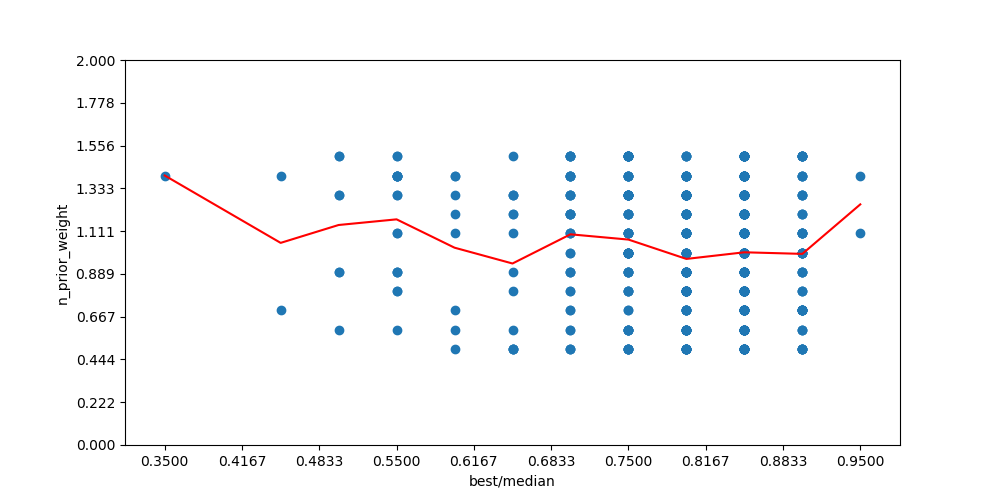
Here is the histogram of results:



Ok, looks like it might be slightly better at lower values. However, look at the hole in the histogram - maybe there is an error in the calculation. I check the raw data to be sure:

{'n\_prior\_weight\_0.5': 44,  
 'n\_prior\_weight\_0.6': 39,  
 'n\_prior\_weight\_0.7': 38,  
 'n\_prior\_weight\_0.8': 48,  
 'n\_prior\_weight\_0.9': 43,  
 'n\_prior\_weight\_1.0': 52,  
 'n\_prior\_weight\_1.1': 44,  
 'n\_prior\_weight\_1.2': 39,  
 'n\_prior\_weight\_1.3': 61,  
 'n\_prior\_weight\_1.4': 47,  
 'n\_prior\_weight\_1.5': 45}

It doesn’t look like there's a response from the algorithm. However, we are interested in tuning the algorithm, so we check the correlations with our features. Here is the graph of the highest correlated variable:



At this point we are pretty sure that changing prior\_weight has no effect on the algorithm. Maybe the idea is that you change prior\_weight on a per-trial basis to give weightings to each of the trials? I will investigate next time - it's not clear from the Hyperopt documentation, the code or the paper about TPE.

## Parameter Weighting

Our next idea was more interesting then just tuning TPE’s existing hyper-parameters. We might know in advance that certain hyper-parameters have a bigger effect on the outcome then other hyperparameters. We therefore want the algorithm to spend more of its time focused on these hyperparameters then on others. By doing so, we might be able to get to optimal values faster.

But how exactly would we manifest this in the TPE algorithm? I happened to stumble upon a page in the Hyperopt wiki about Partial Sampling ( see <https://github.com/hyperopt/hyperopt/wiki/Recipes> ). The Recipe code itself was convoluted, but right at the top they said if you can modify the original hyper-parameter space, and preserve the trial history, Hyperopt will do the correct thing.

So my idea to implement weighting was simple. For some subset of trials, we would lock in certain hyper-parameters at their best known value - the value they have in the current best trial. Hyperopt would then predict optimal values the remaining hyper-parameters.

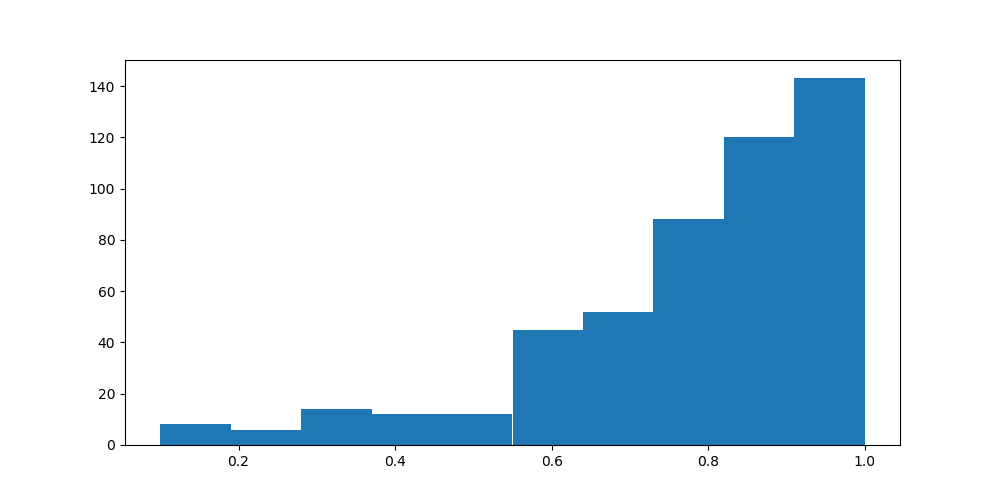
To simplify all of the possible weighting schemes that I could possibly test, I simply divided hyperparameters into two groups - the “primary” hyperparameters and the “secondary” hyperparameters. The “primary” hyperparameters would get optimized at every single trial the entire way through. The “Secondary” hyperparameters had a 50-50 chance of getting optimized on any given trial. They would otherwise be locked in at their current best values.

The question becomes, how many parameters to make as primary? How many to make as secondary? For our randomly generated hyperparameter spaces, we had an easy way getting prior-knowledge on the weights of hyper-parameters - those weights were built into the math formulas when they were generated. For our two real-world test cases, I could use the correlation matrices on the results, or simply my prior knowledge on how deep learning or gradient boosting models work, in order to come up with a reasonable guess on weights.

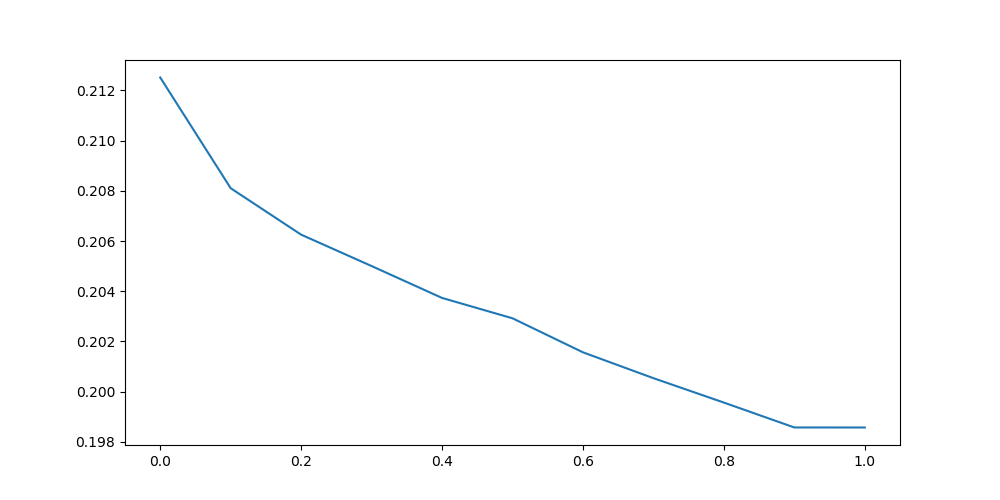
We chose the cutoff point between the primary and secondary hyperparameters based on a threshold on the weights. To be clear, this was not a threshold on the # of hyperparameters, it was based on their cumulative weight.

We tested cutoff values ranging smoothly from 0.0, meaning all parameters are primary, and 1.0, which would make all parameters secondary. We figured, there should be some optimal cutoff point between the primary and secondary parameters. In practice, our chosen cutoff point wouldn’t generalite because without being given exact, accurate weights, but it gives us a basis to validate the approach.

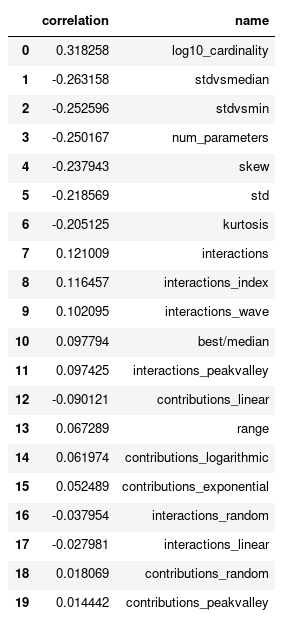
We computed the optimal cutoff point for each hyperparameter space, and created a histogram over the results:



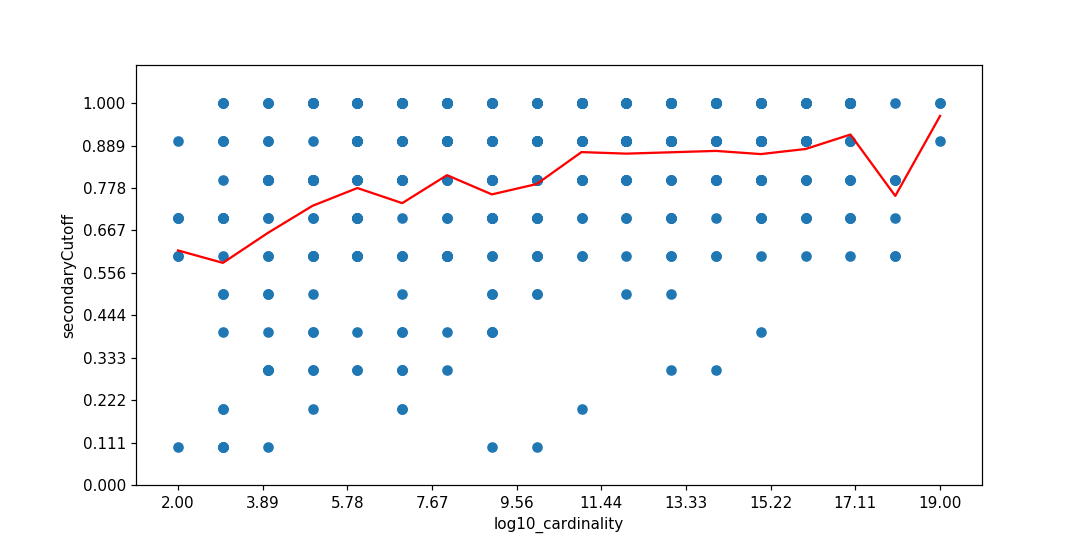
Amazing! It almost seems as if we want to consider every parameter a secondary parameter. We start digging further into the results. We plotted the mean loss for each cut off value:



Whoah! I am now definitely feeling that we are onto something with parameter weighting. We compute correlations between the optimal cutoff point and our various predictive features:

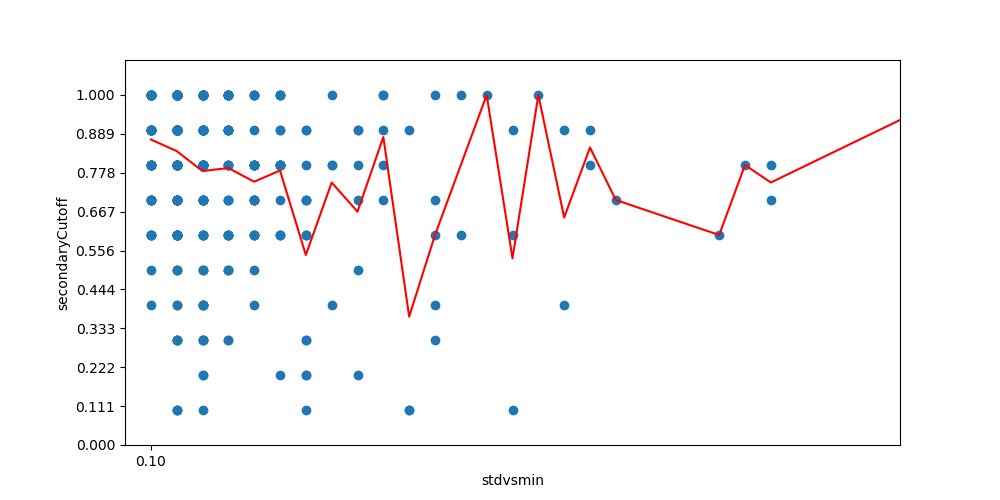


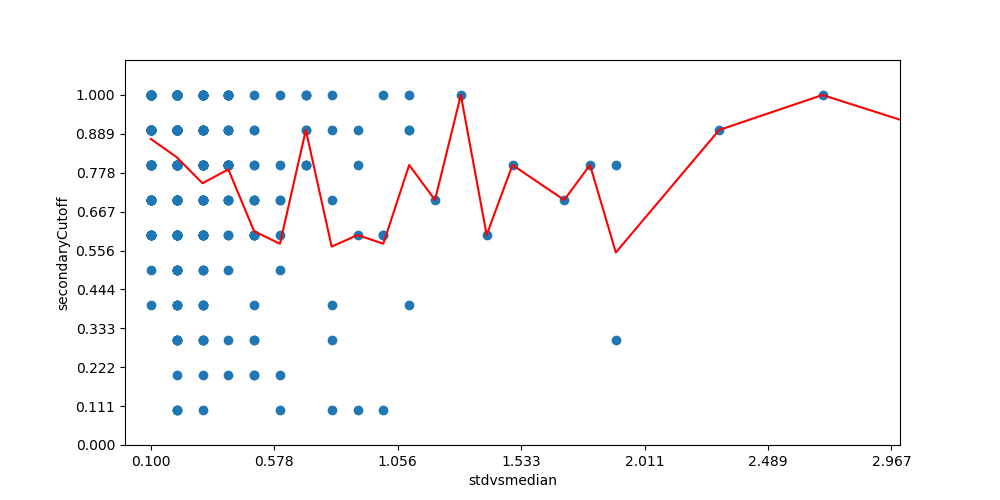
Once again we might have an interesting correlation with cardinality and with standard deviation! We dive in:



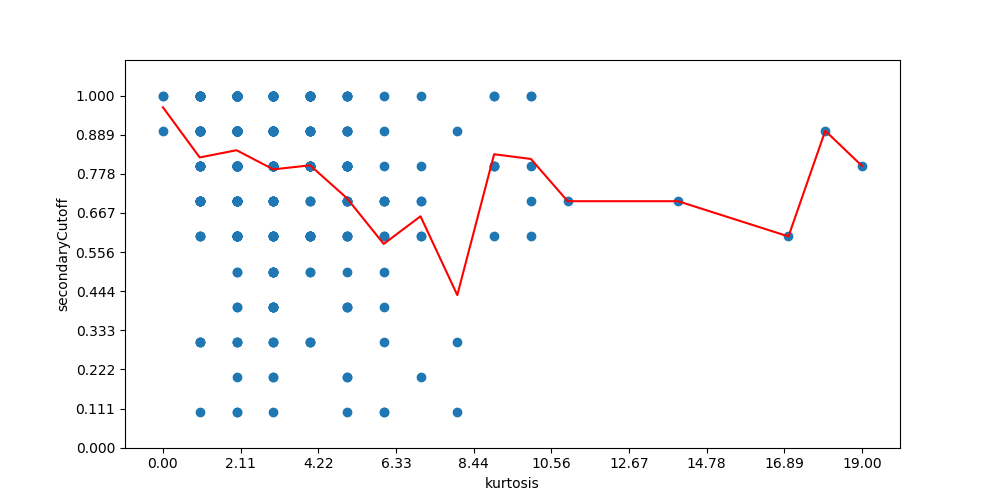
There is a definite pattern here - as you increase the size of the hyperparameter space, weighting tends to work better. But it doesn’t always work on every hyper-parameter space.

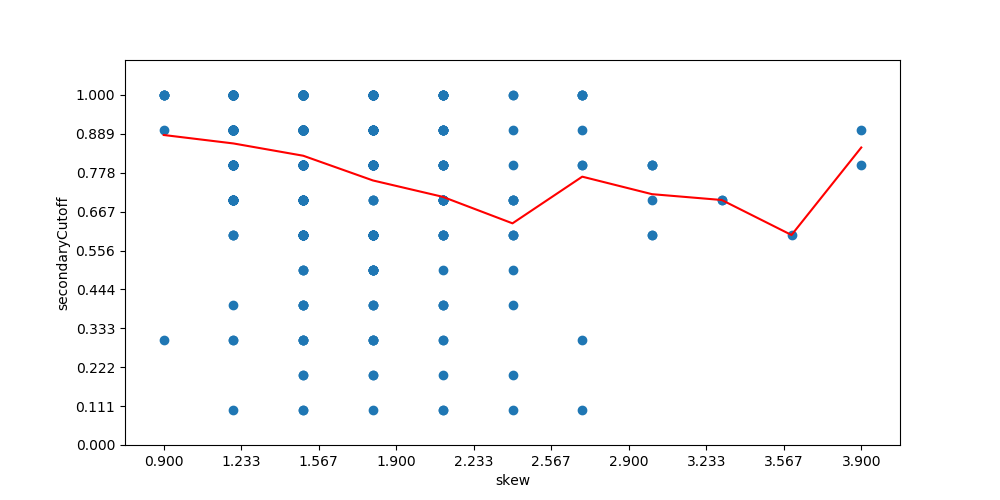
Our stddev/median and stddev/min metrics also had a high correlation, so I checked those out:





Although the correlation is high, its not totally clear to me there is a pattern here or what the pattern is. I decide to move on. Let’s check out skew and kurtosis:

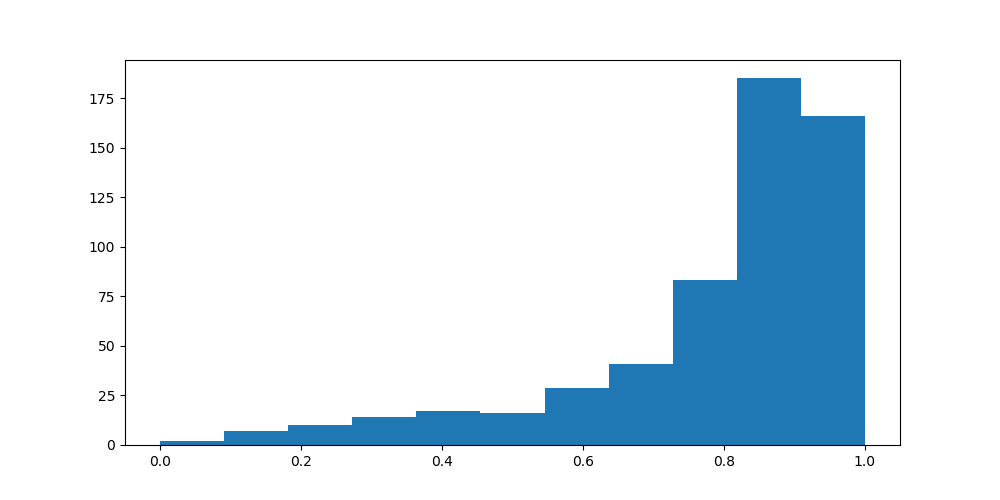




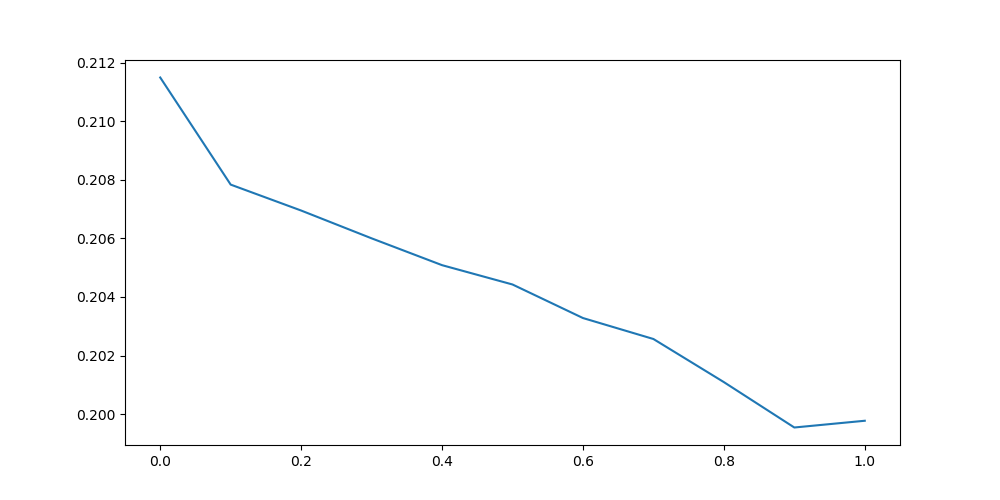
Given the relatively good performance of parameter weighting, even with a cutoff of 1.0 (which puts all parameters as secondary parameters, meaning they only get optimized in half of the rounds on average), I became curious at this point if there was an way to automatically determine the weightings of parameters and still get robust results.

I come up with a new algorithm using Spearman's rank correlation (https://en.wikipedia.org/wiki/Spearman%27s\_rank\_correlation\_coefficient) as my weights for each of the hyperparameters. We use the absolute value since we don’t care about the direction, just magnitude. The cutoff is multiplied by the cumulative correlation, and parameters are then divided into two groups - the primary and secondary. The primary are optimized every round, and the secondary are optimized randomly on each round with a 50-50 probability.

I test the new algorithm with the same set of possible cut-off values between 0 and 1. For each hyper-parameter space, we calculate what the optimal cutoff is between 0 and 1.0. We get this results:

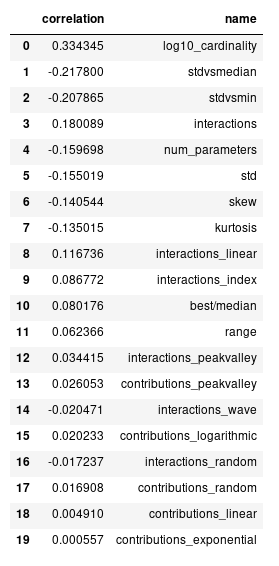


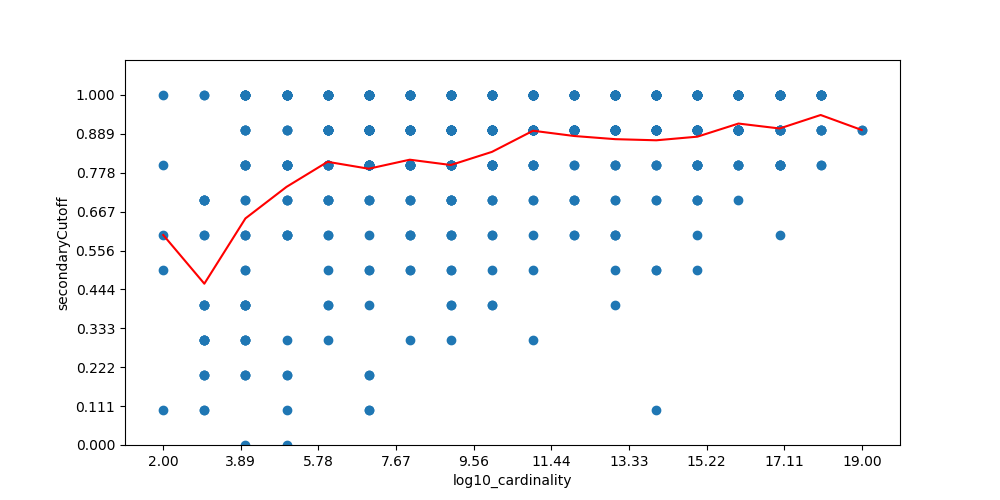
Amazing! It's definitely looking like our new Bagged Optimization technique is working - the vast majority of our hyper-parameter spaces would prefer to have a higher cutoff point, although just below 1.0. We look at the loss chart to confirm:

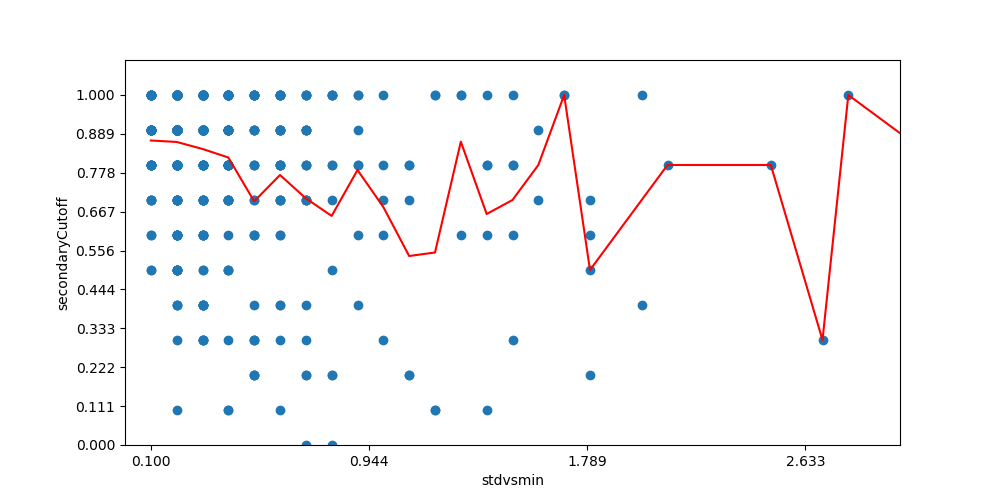


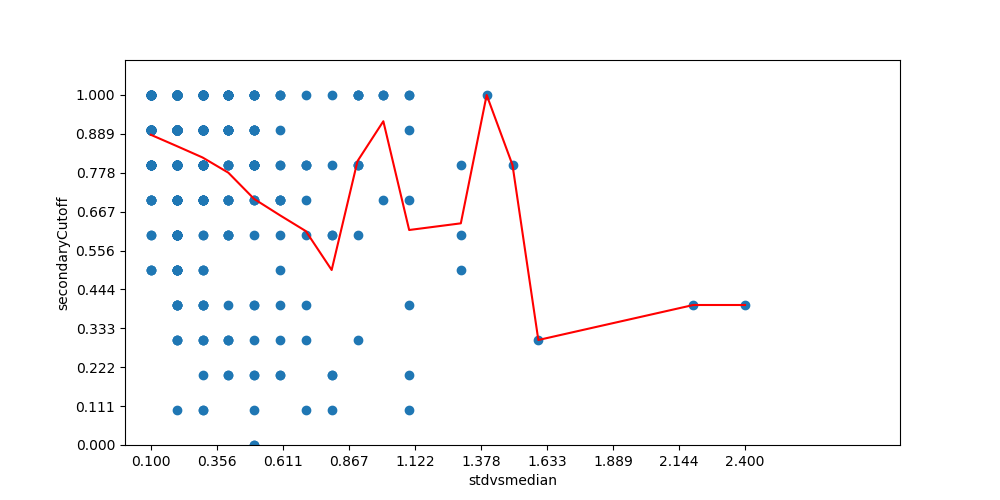
Yes indeed it looks like a very smooth improvement in the loss as you apply more bagging, right up until about 0.9. This indicates to me that you want most hyperparameters in the secondary group, and only a few in the primary group (on average).

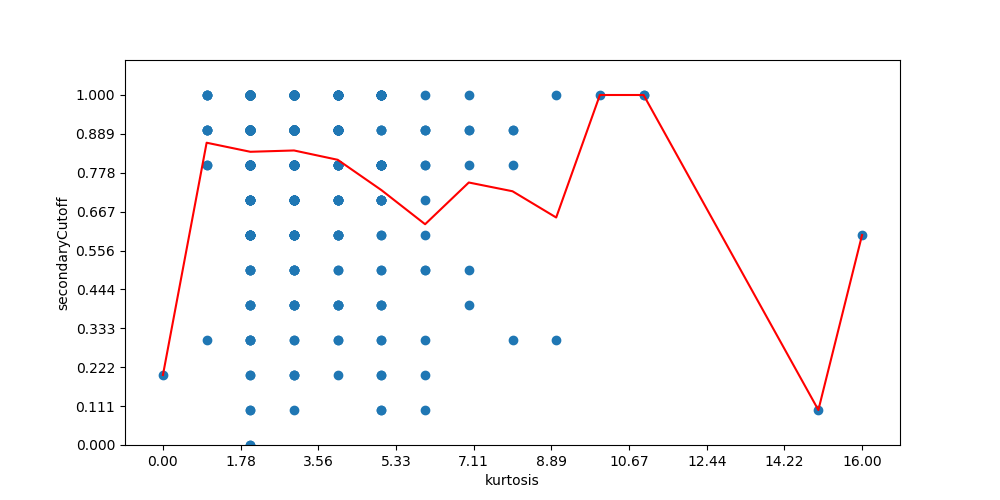
We look at correlations and the same patterns come up again:

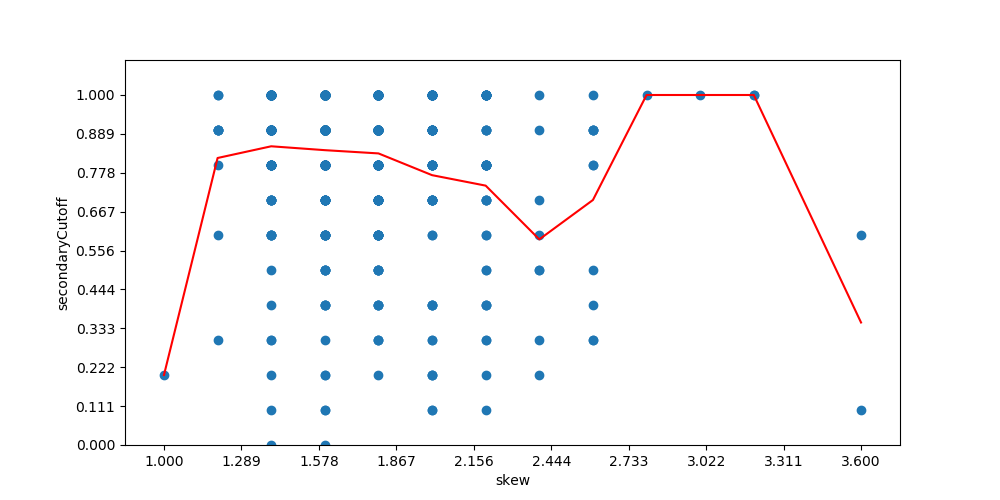












We select the two most highly correlated variables - log10\_cardinality and stdvsmedian to be our predictor variables for tuning the cutoff. We use Ridge regression to get a nice, simple equation:

Cutoff = 0.65 + stdvsmedian \* -0.07 + log10\_cardinality \* 0.02

To be sure, we test our cutoff equation on the dataset. We compare a value of 0.0 (default behaviour with all primary parameters) with the adaptive cutoff value. We calculate the mean loss of the hyperparameter spaces. We get the following:

|  |  |
| --- | --- |
| Cutoff | Loss |
| 0.0 | 0.2075 |
| adaptive | 0.1992 |

It looks confirmed! Now we test the same thing on our real-world text extraction dataset. We also added in a cutoff at 0.9 just as a foil:

|  |  |
| --- | --- |
| Cutoff | Loss |
| 0.0 | 0.1078 |
| 0.9 | 0.1025 |
| adaptive | 0.1050 |

Seems like parameter weighting worked here too. The optimal cutoff appeared to be 0.9, but our adaptive cutoff still did better than having no weighting. Now lets test CIFAR-10:

|  |  |
| --- | --- |
| Cutoff | Loss |
| 0.0 | 0.1567 |
| 0.9 | 0.1383 |
| adaptive | 0.1247 |

Boom shackalacka!

## Independent Distribution Modelling

The last thing that we wanted to test was whether all of the hyperparameter should actually be modelled together jointly. TPE makes a strict assumption that your hyperparameters are interacting. What if you know in advance that certain hyperparameters don’t interact? Can TPE do better if we model them independently, even while optimizing them at the same time?

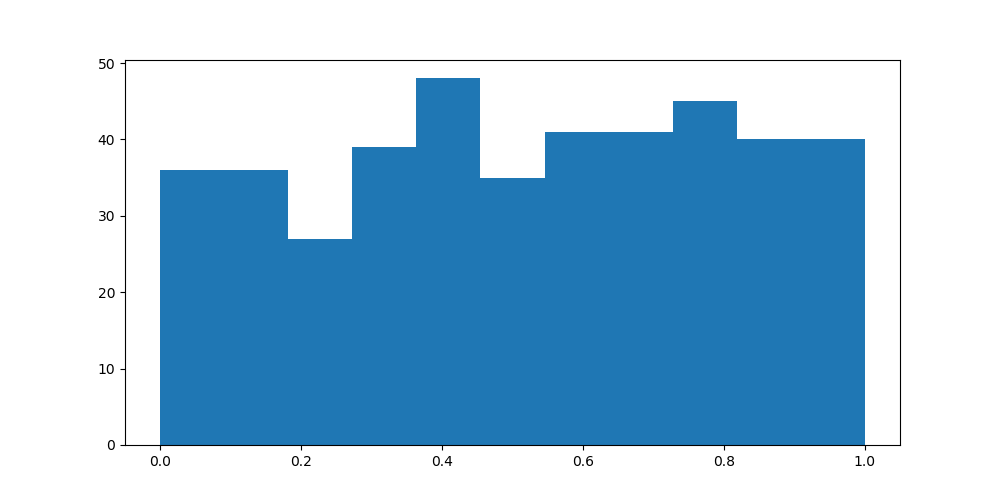
There are many reasons to think it wouldn’t be the case. Even as independent as they can get, the hyperparameters of one group would still create noise for the other group if you are optimizing them over the same set of trials. At the same time, overfitting interactions between hyperparameters could also be a source of noise and error in the model, so using your prior knowledge and independently modelling hyperparameters known to be independent may improve results.

In order to test this, we modified our hyper-parameter searches so that they divided hyperparameters into two groups - group1 and group2. Group1 hyperparameters would only interaction with other Group1 hyperparameters. Same with Group2 hyperparameters. To simplify the logic, we divided the hyperparameters precisely in two - 50% in group 1 and 50% in group2. The only way the two groups interact is when their losses are added together to produce a final loss. They are effectively seperate sample spaces. This is probably not realistic, but provides a strict test-case to view whether this technique has any legs at all.

When optimizing our model, we took the sample space and trial history and created two separate versions - one which only viewed group1 hyperparameters and one which only viewed group2 hyperparameters. We would feed both of these into the TPE algorithm to get its recommendations. We then merged together the recommended hyperparameters from each group into a single next recommendation which would then be tested.

Further, we didn’t actually model the parameters separately on every trial. There was a parameter, D, which goes between 0 and 1 and controls the percentage of trials which are modelled with independent distributions, and which those which are modelled with joint distributions. Assuming that independent modelling actually helps, we should smoothly see an improvement in results as we do more and more independent modelling, until the entire search is done with two independent distributions.

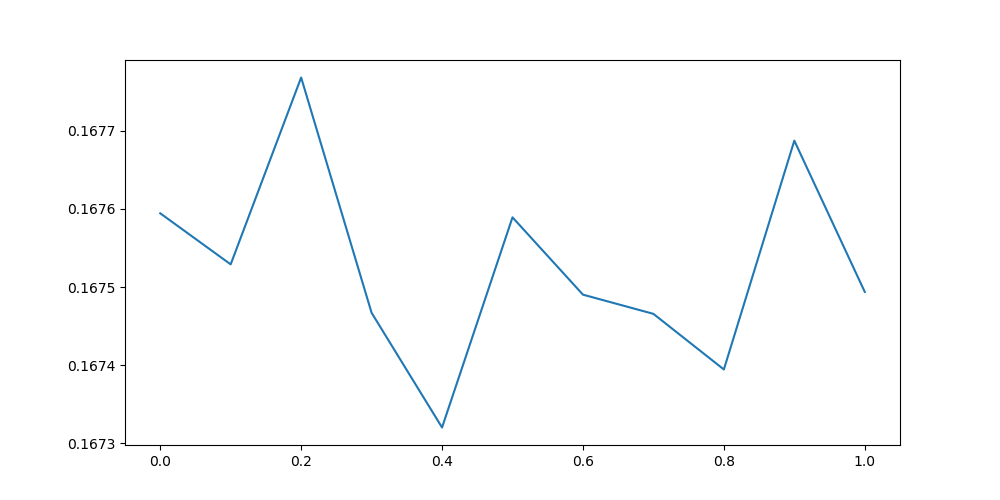
We computed a histogram over the optimal values for the independent modelling rate:



Hmm, its not looking like there is a response. Independent modelling would likely hurt in situations where hyperparameters actually do interact, but in our carefully designed test case, there is no response. We look at the raw data to help us confirm:

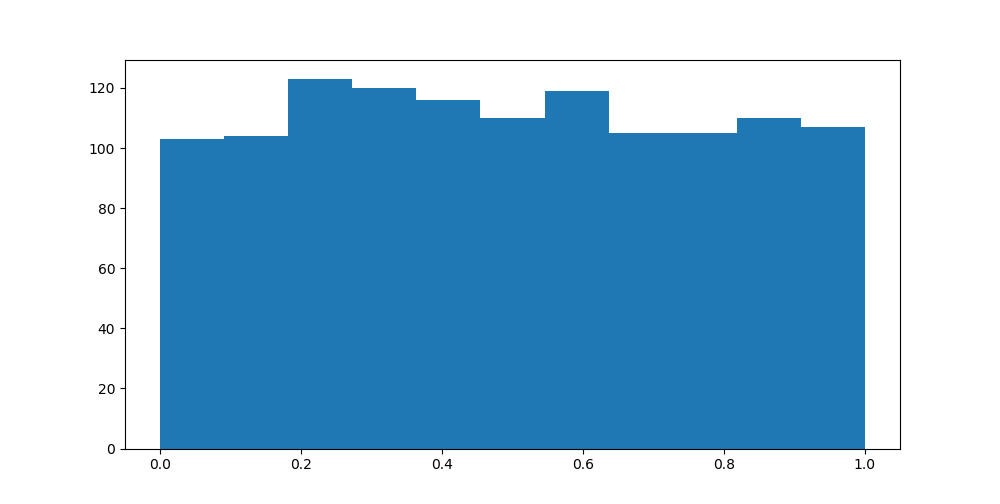
{'independentModellingRate\_0.0': 36,  
 'independentModellingRate\_0.1': 36,  
 'independentModellingRate\_0.2': 27,  
 'independentModellingRate\_0.3': 39,  
 'independentModellingRate\_0.4': 48,  
 'independentModellingRate\_0.5': 35,  
 'independentModellingRate\_0.6': 41,  
 'independentModellingRate\_0.7': 41,  
 'independentModellingRate\_0.8': 45,  
 'independentModellingRate\_0.9': 40,  
 'independentModellingRate\_1.0': 40}

We look at the loss chart for one final check:



There is no discernable pattern here, and the gap between the max and the min is extremely small, especially relative to other charts we have seen. I think we can safely say that independent modelling does not have any legs. You are better off modelling your hyperparameters in a joint distribution, whether they interact with each other or not.

To confirm our intuitions, we tested our independent modelling technique when there are no independent hyperparameters. We did the test with far fewer rounds then normal, just so it would complete quickly. Our expectation is that independent modelling would make things worse where there actually are interactions.



Strangely, even when your hyperparameter space contains lots of interacting hyper-parameters, occasionally (or always) modelling them as separate distributions doesn’t seem to hurt performance. We checked our code thoroughly to ensure that there were no bugs, but this seems to be a legitimate result. This suggests to me that the story for independent modelling is not yet over. More investigation is needed.

# 

# The Algorithm: Adaptive Tree of Parzen Estimators (ATPE)

Based on our research into several ways to improve the results of the TPE algorithm, we come up with new algorithm, Adaptive Tree of Parzen Estimators, or ATPE for short. ATPE has more hyperparameters then TPE (in the form of the equations used to set TPE’s own hyperparameters), but those hyperparameters can be derived from data, both from our simulated hyperparameter spaces and hopefully (eventually) real world data from many real hyperparameter searches.

ATPE is built upon four basic improvements over TPE:

* Automatically tuning Gamma based on the Cardinality of your hyperparameter space and the Skew, and Standard Deviation of the results
* Automatically tuning n\_EI\_estimators based on the Cardinality of the hyperparameter space
* Reducing the number of initialization rounds, from 20 to 10
* Optimizing only a subset of hyperparameters on each trial. The subset is selected based on correlation coefficients via a mechanism described in the section “Parameter Weighting”

We test the finished ATPE algorithm side-by-side with vanilla TPE in its unmodified state.

|  |  |  |
| --- | --- | --- |
| Algorithm | Text Extraction | Cifar-ResNet |
| TPE | 0.1090 | 0.1576 |
| ATPE | 0.0976 | 0.1441 |

It looks like our new algorithm outperforms TPE in optimizing hyperparameters. It achieves a roughly 10% decrease in error rates in our tests. It needs to be further validated with additional testing, but we are now feeling very confident that we have found a robust improvement to TPE.

Just for extra confirmation, we ran our actual text extraction model (without simulation) on TPE vs APE, for 100 runs. This model trained fast enough for us to do this within a reasonable amount of time. We were unable to do this with the actual Cifar-ResNet model due to cost, but we intend to follow up this research with additional real-world confirmation of the ATPE algorithms performance on real world models we optimize at Electric Brain.

|  |  |
| --- | --- |
| Algorithm | Text Extraction |
| TPE | 0.1038 |
| ATPE | 0.1009 |

It got better accuracy than anything we found in our original hyperparameter search for this problem. I’m blown away. At this point, I could not be more impressed with this algorithm. It homes in on optimal hyperparameters very quickly, and does not waste time exploring bullshit areas of the hyperparameter space. Its fast, efficient, and got a new best accuracy for one of my clients projects in only 100 trials (vs the 6,000 trials we originally did which included 500 trials of TPE). I feel validated in using simulated hyperparameter spaces to make further improvements to optimization.

# 

# Conclusion

Our goal was to find a way to create a better hyper-parameter optimizer, something that could get people better accuracies with fewer trials.

Testing hyper-parameter optimization algorithms is hard, because each trial can take a long time to compute. In order to make the problem tractable and to allow us to thoroughly evaluate hyperparameter optimization, we took advantage of two techniques:

1. We created arbitrary systems of math equations which were designed to be similar to machine learning hyperparameters. We called these Random Hyperparameter Spaces.
2. We systematically searched out the hyper-parameter spaces of two real-world machine learning problems, chosen because they were quick to evaluate but representative of real-world problems

We tested 7 different ideas for improving the results of the TPE algorithm:

1. Tuning TPE’s Gamma parameter
2. Passing our losses through an activation function prior to optimization
3. Tuning the number of random initialization rounds prior to TPE
4. Tuning TPE’s n\_EI\_candidates parameter
5. Tuning TPE’s prior\_weight parameter
6. Only tuning a subset of parameters in any one trial, thus giving weight/preference to certain hyper-parameters. We call this Bagged Optimization.
7. Having independent TPE models for different hyperparameters, even while still evaluating them at the same time.

We found 4 ways of improving TPE. Collectively, we call our improvements to TPE the “Adaptive Tree of Parzen Estimators” algorithm, or ATPE for short. One of our improvements, bagged optimization, was a huge improvement and is surprisingly robustly across a wide variety of different hyperparameter optimization. The success of bagged optimization opens up new areas of research, and the potential to make ATPE even better in the future. Additionally, ATPE can be tuned with real world data from real hyperparameter spaces, making it better without any algorithmic improvements.

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](mailto:brad@electricbrian.io). We will be using the same methodology in the future to conduct further research.

We are particularly interested in feedback on any of the following areas:

* Ways to improve our Random Hyperparameter Spaces, so that they might more realistically reflect what actual hyperparameter spaces look like
* Ways to improve the accuracy of our Hypermodel, the models which are used to simulate real-world data
* Whether our choice of metric (Mean Loss after K Trials) was a good choice, and what Pros/Cons of other metrics might be. Are there metrics we didn’t consider? Pros/cons we didn’t evaluate?
* Do you have any ideas on good prior-knowledge that we might be able to use to configure our hyper-parameter searches?
* Do you have any ideas on what we can do to algorithmically improve hyper-parameter optimization?
* Do you think we should use a different algorithm from TPE as our base for research?
* What ways should we benchmark our results against industry results?

## 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.

## Next Steps for Research

We have a ton of questions:

* Is Spearman rank correlation really the best correlation?
* We derived our tuning equations from the distributions of the final losses - perhaps you can better result if we calculate them from the optimal improvement at each step (see the Mean quality of next-recommendation metric in our Measurement section)
* We never tested filtering results in various ways prior to insertion into TPE
* Since TPE only cares about rank order of losses and not shape, curious about manipulating the rank-order of top-performing results in various ways before insertion into TPE
* Is 50-50 on each trial the optimal randomness for the secondary group? Should it be less or more?
* Should hyperparameters be bucketed into more then two groups?
* Is it necessarily true that the less correlated secondary parameters are the ones that should get locked in at each trial? Maybe we can do better by locking in the highly correlated parameters, and allowing the less correlated ones to float?

## 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](mailto:business@electricbrain.io)