

Better and faster hyperparameter optimization with Dask

Scott Sievert^{‡§*}, Tom Augspurger^{**}, Matthew Rocklin^{¶||}

Abstract—Nearly every machine learning model requires hyperparameters, parameters that the user must specify before training begins and influence model performance. Finding the optimal set of hyperparameters is often a time- and resource-consuming process. A recent breakthrough hyperparameter optimization algorithm, Hyperband finds high performing hyperparameters with minimal training via a principled early stopping scheme for random hyperparameter selection [LJD⁺18]. This paper will provide an intuitive introduction to Hyperband and explain the implementation in Dask, a Python library that scales Python to larger datasets and more computational resources. The implementation makes adjustments to the Hyperband algorithm to exploit Dask's capabilities and parallel processing. In experiments, the Dask implementation of Hyperband rapidly finds high performing hyperparameters for deep learning models.

Index Terms—distributed computation, hyperparameter optimization, machine learning

Introduction

Training any machine learning pipeline requires data, an untrained model or estimator and "hyperparameters", parameters chosen before training begins that help with cohesion between the model and data. The user needs to specify values for these hyperparameters in order to use the model. A good example is adapting the ridge regression or LASSO to the amount of noise in the data with the regularization parameter [MS75] [Tib96]. Hyperparameter choice verification can not be performed until model training is completed.

Model performance strongly depends on the hyperparameters provided, even for the simple examples above. This gets much more complex when multiple hyperparameters are required. For example, a particular visualization tool, t-SNE requires (at least) three hyperparameters [MH08] and the first section in a study on how to use this tool effectively is titled "Those hyperparameters really matter" [WVJ16].

These hyperparameters need to be specified by the user. There are no good heuristics for determining what the values should be. These values are typically found through a search over possible values through a "cross validation" search where models

are scored on unseen holdout data. Even in the simple ridge regression case above, a brute force search is required [MS75]. This brute force search quickly grows infeasible as the number of hyperparameters grow.

Hyperparameter optimization grows more complex as the number of hyperparameters grow, especially because of the frequent interactions between them. A good example of hyperparameter optimization is with deep learning, which has specialized algorithms for handling many data but have difficulty providing basic hyperparameters. For example, the commonly used stochastic gradient descent (SGD) has difficulty with the most basic hyperparameter "learning rate" [Bot10], which is a quick computation with few data but infeasible for many data [MH15].

Contributions

A hyperparameter optimization is required if high performance is desired. In practice, it's expensive and time-consuming for machine learning researchers and practitioners. Ideally, hyperparameter optimization algorithms return high performing models quickly and are simple to use.

Quickly returning quality hyperparameters relies on making decisions about which hyperparameters to devote training time to. This might mean progressively choosing higher-performing hyperparameter values or stopping low-performing models early during training.

Returning this high performing model quickly would lower the expense and/or time barrier to performing hyperparameter optimization. This will allow the user (e.g., a data scientist) to more easily use these algorithms.

This work

- provides an implementation of a particular hyperparameter optimization algorithm, Hyperband [LJD⁺18] in Dask [Das16], a Python library that provides advanced parallelism. Hyperband returns models with a high validation score with minimal training. A Dask implementation is attractive because Hyperband is amenable to parallelism.
- makes a simple modifications to increase Hyperband's amenability to parallelism.
- provides an simple heuristic to determine the parameters Hyperband requires, which only requires knowing how many examples the model should observe and a rough estimate on how many parameters to sample
- provides validating experiments that illustrate common use cases and explore performance

* Corresponding author: scott@stsievert.com

‡ University of Wisconsin–Madison

§ Relevant work performed while interning for Anaconda, Inc.

** Anaconda, Inc.

¶ NVIDIA

|| Relevant work performed while employed for Anaconda, Inc.

Hyperband treats computation as a scarce resource¹ and has parallel underpinnings. In the experiments performed with the Dask implementation, Hyperband returns high performing models fairly quickly with a simple heuristic for determining Hyperband’s input parameters. The implementation can be found in Dask’s machine learning package, Dask-ML².

This paper will review other existing work for hyperparameter optimization before detailing the Hyperband implementation in Dask. A realistic set of experiments will be presented to highlight the performance of the Dask implementation before mentioning ideas for future work.

Related work

Hyperparameter optimization

Hyperparameter optimization finds the optimal set of hyperparameters for a given model. These hyperparameters are chosen to maximize performance on unseen data. The hyperparameter optimization process typically looks like

- 1) Split the dataset into the train dataset and test dataset. The test dataset is reserved for the final model evaluation.
- 2) Choose hyperparameters
- 3) Train models with those hyperparameters
- 4) Score those models with unseen data (a subset of the train dataset typically referred to as the "validation set")
- 5) Use the best performing hyperparameters to train a model with those hyperparameters on the complete train dataset
- 6) Score the model on the test dataset. This is the score that is reported.

The rest of this paper will focus on steps 2 and 3, which is where most of the work happens in hyperparameter optimization.

A commonly used method for hyperparameter selection is a random selection of hyperparameters, and is typically followed by training each model to completion. This offers several advantages, including a simple implementation that is very amenable to parallelism. Other benefits include sampling "important parameters" more densely than unimportant parameters [BB12]. This randomized search is implemented in many places, including in Scikit-Learn [PVG⁺11].

These implementations are by definition *passive* because they do not adapt to previous training. *Adaptive* algorithms can return a higher quality solution with less training by adapting to previous training and choosing which hyperparameter values to evaluate. This is especially useful for difficult hyperparameter optimization problems with many hyperparameters and many values for each hyperparameter.

A popular class of adaptive hyperparameter optimization algorithms are Bayesian algorithms. These algorithms treat the model as a black box and the model scores as an evaluation of that black box. These algorithms have an estimate of the optimal set of hyperparameters and use some probabilistic methods to improve the estimate. The choice of which hyperparameter value to evaluate depends on previous evaluations.

Popular Bayesian searches include sequential model-based algorithm configuration (SMAC) [hut11], tree-structure Parzen estimator (TPE) [STZB⁺11], and Spearmint [PBBW12]. Many of these are available through the "robust Bayesian optimization"

package RoBo [KFMH17] through AutoML³. This package also includes Fabolas, a method that takes dataset size as input and allows for some computational control [KFB⁺16].

Hyperband

Hyperband is a principled early stopping scheme for randomized hyperparameter selection⁴ and an adaptive hyperparameter optimization algorithm [LJD⁺18]. At the most basic level, it partially trains models before stopping models with low scores, then repeats. By default, it stops training the lowest performing 33% of the available models at certain times. This means that the number of models decay over time, and the surviving models have high scores.

Naturally, model quality depends on two factors: the amount of training performed and the values of various hyperparameters. If training time only matters a little, it makes sense to aggressively stop training models. On the flip side, if only training time influences the score, it only makes sense to let all models train for as long as possible and not perform any stopping.

Hyperband sweeps over the relative importance of hyperparameter choice and amount of training. This sweep over training time importance enables a theorem that Hyperband will return a much higher performing model than the randomized search without early stopping returns. This is best characterized by an informal presentation of the main theorem:

Corollary 1. (informal presentation of [LJD⁺18, Theorem 5] and surrounding discussion) Assume the loss at iteration k decays like $(1/k)^{1/\alpha}$, and the validation losses v approximately follow the cumulative distribution function $F(v) = (v - v_*)^\beta$ with optimal validation loss v_* with $v - v_* \in [0, 1]$.

Higher values of α mean slower convergence, and higher values of β represent more difficult hyperparameter optimization problems because it’s harder to obtain a validation loss close to the optimal validation loss v_* . Taking $\beta > 1$ means the validation losses are not uniformly distributed and higher losses are more common. The commonly used stochastic gradient descent has convergence rates with $\alpha = 2$ [Bot12] [LJD⁺18, Corollary 6], and gradient descent has convergence rates with $\alpha = 1$ [B⁺15, Theorem 3.3].

Then for any $T \in \mathbb{N}$, let \hat{v}_T be the empirically best performing model when models are stopped early according to the infinite horizon Hyperband algorithm when T resources have been used to train models. Then with probability $1 - \delta$, the empirically best performing model \hat{v}_T has loss

$$v_{\hat{v}_T} \leq v_* + c \left(\frac{\overline{\log}(T)^3 \cdot a}{T} \right)^{1/\max(\alpha, \beta)}$$

for some constant c and $a = \overline{\log}(\log(T)/\delta)$ where $\overline{\log}(x) = \log(x \log(x))$.

By comparison, finding the best model without the early stopping Hyperband performs (i.e., randomized searches and training until completion) after T resources have been used to train models has loss

$$v_{\hat{v}_T} \leq v_* + c \left(\frac{\log(T) \cdot a}{T} \right)^{1/(\alpha+\beta)}$$

For simplicity, only the infinite horizon case is presented though much of the analysis carries over to the practical finite

1. If computation is not a scarce resource, there is little benefit from this algorithm.

2. <https://ml.dask.org>.

3. <https://github.com/automl/>

4. In general, Hyperband is a resource-allocation scheme for model selection.

horizon Hyperband.⁵ Because of this, it only makes sense to compare the loss when the number of resources used T is large. When this happens, the validation loss of the Hyperband produces $v_{\hat{T}}$ decays much faster than the uniform allocation scheme.⁶ This shows a definite advantage to performing early stopping on randomized searches.

Li et. al. show that the model Hyperband identifies as the best is identified with a (near) minimal amount of training in Theorem 7 [LJD⁺18], within log factors of the known lower bound [KCG16].

More relevant work involves combining Bayesian searches and Hyperband, which can be combined by using the Hyperband bracket framework *sequentially* and progressively tuning a Bayesian prior to select parameters for each bracket [FKH18]. This work is also available through AutoML.

There is little to no gain from adaptive searches if the passive search requires little computational effort. Adaptive searches spends choosing which models to evaluate to minimize the computational effort required; if that's not a concern there's not much value the value in any adaptive search is limited.

Dask

Dask provides advanced parallelism for analytics, especially for NumPy, Pandas and Scikit-learn [Das16]. It is familiar to Python users and does not require rewriting code or retraining models to scale to larger datasets or to more machines. It can scale up to clusters or to a massive dataset but also works on laptops and presents the same interface. Dask provides two components:

- Dynamic task scheduling optimized for computation. This low level scheduler provides parallel computation and is optimized for interactive computational workloads.
- "Big Data" collections like parallel arrays, or dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

Dask aims to be familiar and flexible: it aims to parallelize and distribute computation or datasets easily while retaining a task scheduling interface for custom workloads and integration into other projects. It is fast and the scheduler has low overhead. It's implemented in pure Python and can scale from massive datasets to a cluster with thousands of cores to a laptop running single process. In addition, it's designed with interactive computing and provides rapid feedback and diagnostics to aid humans.

Dask's implementation of Hyperband

Combining Dask and Hyperband is a natural fit. Hyperparameter optimization searches often require significant amounts of computation and can involve large datasets. Hyperband is amenable to parallelism, and Dask can scale up to clusters or to massive datasets.

This work focuses on the case when significant computation is required. In these cases, the existing passive hyperparameter

5. To prove results about the finite horizon algorithm Li et. al. only need the result in Corollary 9 [LJD⁺18]. In the discussion afterwards they remark that with Corollary 9 they can show a similar result but leave it as an exercise for the reader.

6. This is clear by examining $\log(v_{\hat{T}} - v_*)$ for Hyperband and uniform allocation. For Hyperband, the slope approximately decays like $-1/\max(\alpha, \beta)$, much faster than the uniform allocation's approximate slope of $-1/(\alpha + \beta)$.

optimization algorithms in Dask-ML have limited use because they don't adapt to previous training to reduce the amount of training required.⁷

This section will explain the parallel underpinnings of Hyperband, show the heuristic for Hyperband's inputs and mention a modification to increase amenability to parallelism. Complete documentation of the Dask implementation of Hyperband can be found at https://ml.dask.org/modules/generated/dask_ml.model_selection.HyperbandSearchCV.

Hyperband architecture

There are two levels of parallelism in Hyperband, which result in two for-loops:

- an "embarrassingly parallel" sweep over the different brackets of the training time importance
- each bracket has an early stopping scheme for random search. This means the models are trained independently in parallel. At certain times, training stops on certain models.

The amount of parallelism makes a Dask implementation very attractive. Dask Distributed is required because the computational graph is dynamic and depends on other nodes in the graph.

Of course, the number of models in each bracket decreases over time because Hyperband is an early stopping strategy. This is best illustrated by the algorithm's pseudo-code:

```
from sklearn.base import BaseEstimator

def sha(n_models: int,
        calls: int,
        max_iter: int) -> BaseEstimator:
    """Successive halving algorithm"""
    # (model and params are specified by the user)
    models = [get_model(random_params())
              for _ in range(n_models)]
    while True:
        models = [train(m, calls) for m in models]
        models = top_k(models, k=len(models) // 3)
        calls *= 3
        if len(models) < 3:
            return top_k(models, k=1)

def hyperband(max_iter: int) -> BaseEstimator:
    # Different brackets have different values of
    # "training" and "hyperparameter" importance.
    # => more models means more aggressive pruning
    brackets = [(get_num_models(b, max_iter),
                 get_initial_calls(b, max_iter))
                for b in range(formula(max_iter))]
    if max_iter == 243: # for example...
        assert brackets == [(81, 3), (34, 9),
                           (15, 27), (8, 81),
                           (5, 243)]
    # Each tuple is (num_models, n_init_calls)
    final_models = [sha(n, r, max_iter)
                   for n, r in brackets]
    return top_k(final_models, k=1)
```

In this pseudo-code, the train set and validation data are hidden. `top_k` returns the k best performing models on the validation data and `train` trains a model for a certain number of calls to `partial_fit`.

Each bracket indicates a value in the trade-off between training time and hyperparameter importance, and is specified by the list of tuples in the example above. Each bracket is specified so that the

7. The existing implementation can reduce the computation required when pipelines are used. This is particularly useful when tuning data preprocessing (e.g., with natural language processing). More detail is at <https://ml.dask.org/hyper-parameter-search.html>.

total number of `partial_fit` calls is approximately the same among different brackets. Then, having many models requires pruning models very aggressively and vice versa with few models. As an example, with `max_iter=243` the least adaptive bracket has 5 models and no pruning. The most adaptive bracket has 81 models and fairly aggressive early stopping schedule.

The exact aggressiveness of the early stopping schedule depends on one optional input to `HyperbandSearchCV`, `aggressiveness`. The default value is 3, which has some mathematical motivation [LJD⁺18, Section 2.6]. `aggressiveness=4` is likely more suitable for initial exploration when not much is known about the model, data or hyperparameters.

Input parameters

Hyperband is also fairly easy to use. It requires two input parameters:

- 1) the number of `partial_fit` calls for the best model (via `max_iter`)
- 2) the number of examples that each `partial_fit` call sees (which is implicit and referred to as `chunks`, which can be the "chunk size" of the Dask array).

These two parameters rely on knowing how long to train the model⁸ and having a rough idea on the number of parameters to evaluate. Trying twice as many parameters with the same amount of computation requires halving `chunks` and doubling `max_iter`.

The primary advantage to Hyperband's inputs is that they do not require balancing training time importance and hyperparameter importance.

In comparison, random searches require three inputs:

- 1) the number of `partial_fit` calls for every model (via `max_iter`)
- 2) how many parameters to try (via `num_params`).
- 3) the number of examples that each `partial_fit` call sees (which is implicit and referred to as `chunks`, which can be the "chunk size" of the Dask array).

Trying twice as many parameters with the same amount of computation requires doubling `num_params` and halving either `max_iter` or `chunks`, which means every model will see half as many data. Implicitly, a balance between training time and hyperparameter importance is being decided upon. Hyperband has one fewer input because it sweeps over this balance's importance in different brackets.

Dwindling number of models

At first, Hyperband evaluates many models. As time progresses, the number of models decay because Hyperband is an early stopping scheme. This means towards the end of the computation, a few (possibly high-performing) models can be training while most of the computational hardware is free. This is especially a problem when computational resources are not free (e.g., with cloud platforms like Amazon AWS or Google Cloud Platform).

Hyperband is a principled early stopping scheme, but it doesn't protect against at least two common cases:

⁸ e.g., something in the form "the most trained model should see 100 times the number of examples (aka 100 epochs)"

⁹ Tolerance (typically via `tol`) is a proxy for `max_iter` because smaller tolerance typically means more iterations are run.

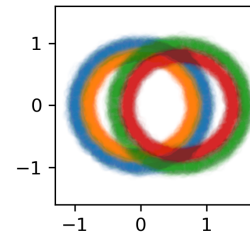


Fig. 1: The synthetic dataset used as input for the serial simulations. The colors correspond to different class labels. In addition to these two informative dimensions, there are 4 uninformative dimensions with uniformly distributed random noise. There are 60,000 examples in this dataset and 50,000 are used for training.

- 1) when models have converged before training completes (i.e., the score stays constant)
- 2) when models have not converged and poor hyperparameters are chosen (i.e., the scores are not increasing).

Providing a "stop on plateau" scheme will protect against these cases because training will be stopped if a model's score stops increasing [Pre98]. This will require two additional parameters: `patience` to determine how long to wait before stopping a model, and `tol` which determines how much the score should increase.

Hyperband's early stopping is designed to identify the highest performing model with minimal training. Setting `patience` to be high avoids interference with this scheme, protects against both cases above, and errs on the side of giving models more training time. In particular, it also provides a basic early stopping mechanism for the least adaptive bracket of Hyperband.

Serial Simulations

This section is focused on the initial exploration of a model and its hyperparameters on a personal laptop. This section shows a performance comparison to illustrate the `HyperbandSearchCV`'s utility. This comparison will use a rule-of-thumb to determine the inputs to `HyperbandSearchCV`.

A synthetic dataset is used for a 4 class classification problem on a personal laptop with 4 cores. This makes the hyperparameter selection very serial and the number of `partial_fit` calls or passes through the dataset a good proxy for time. Some detail is mentioned in the appendix with complete details at <https://github.com/stsievert/dask-hyperband-comparison>.

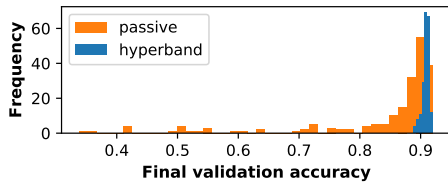
```
from dask_ml.model_selection import train_test_split
X, y = make_4_circles(num=60e3)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=int(10e3))
```

A visualization of this dataset is in Figure 1.

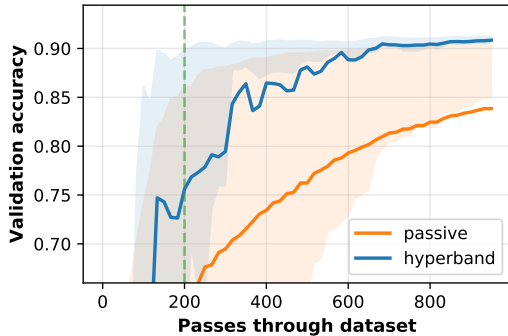
Model architecture & Hyperparameters

Scikit-learn's fully-connected neural network is used, their `MLPClassifier` which has several hyperparameters. Only one affects the architecture of the best model: `hidden_layer_sizes`, which controls the number of layers and number of neurons in each layer.

There are 5 values for the hyperparameter. It is varied so the neural network has 24 neurons but varies the network depth and the width of each layer. Two choices are 12 neurons in 2 layers



(a) The final validation accuracy over the different runs. Out of the 200 runs, the worst of the *hyperband* runs performs better than 99 of the *passive* runs, and 21 *passive* runs have final validation accuracy less than 70%.



(b) The average best score from *Hyperband*'s early stopping scheme (via *hyperband*) and randomized search without any early stopping (via *passive*). The shaded regions correspond to the 25% and 75% percentiles over the different runs. The green dotted line indicates the time required to train 4 models with 4 *Dask* workers.

Fig. 2: In this simulation, each call to `partial_fit` sees about 1/6th of examples in the complete train dataset. Each model completes no more than 50 passes through the data. This experiment includes 200 runs of *hyperband* and *passive* and *passive*.

or 6 neurons in four layers. One choice has 12 neurons in the first layer, 6 in the second, and 3 in third and fourth layers.

The other six hyperparameters control finding the best model and do not influence model architecture. 3 of these hyperparameters are continuous and 3 are discrete (of which there are 10 unique combinations). Details are in the appendix. These hyperparameters include the batch size, learning rate (and decay schedule) and a regularization parameter:

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(...)
params = {'batch_size': [32, 64, ..., 512], ...}
print(params.keys())
# dict_keys([
#   "batch_size", # 5 choices
#   "learning_rate", # 2 choices
#   "hidden_layer_sizes", # 5 choices
#   "alpha", # cnts
#   "power_t", # cnts
#   "momentum", # cnts
#   "learning_rate_init" # cnts
# ])
```

Usage: rule of thumb on `HyperbandSearchCV`'s inputs

`HyperbandSearchCV` only requires two parameters besides the model and data as discussed above: the number of `partial_fit` calls for each model (`max_iter`) and the number of examples each call to `partial_fit` sees (which is implicit via the `Dask` array chunk size `chunks`). These inputs control how many

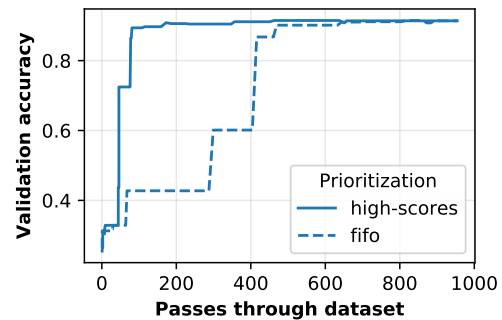


Fig. 3: A visualization of how the *Dask* prioritization scheme influences the *Hyperband*'s time to solution. *Dask* assigns prioritizes training models with higher scores (via *high-scores*). When *Dask* uses the default priority scheme it fits models in the order they are received by *Dask Distributed*'s scheduler (via *fifo*). Only the prioritization in the figure changes because both *high-scores* and *fifo* have the same hyperparameters, train/validation data, and assign the same internal random state to models. The hyperparameters are chosen from a run in Figure 2b.

hyperparameter values are considered and how long to train the models.

The values for `max_iter` and `chunks` can be specified by a rule-of-thumb once the number of parameter to be sampled and the number of examples required to be seen by at least one model, `n_examples`. This rule of thumb is:

```
# The rule-of-thumb to determine inputs
max_iter = n_params
chunks = n_examples // n_params
```

In this example, `n_examples = 50 * len(X_train)` and `n_params = 299`. `n_params` is approximately the number of hyperparameter sampled. The value of 299 is chosen to make the *Dask* array evenly chunked and to sample approximately 4 hyperparameter combinations for unique combination of discrete hyperparameters.

Creation of a `HyperbandSearchCV` object and the *Dask* array is simple with this:

```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
    model, params,
    max_iter=max_iter, aggressiveness=4)
```

```
X_train = da.from_array(X_train, chunks=chunks)
y_train = da.from_array(y_train, chunks=chunks)
search.fit(X_train, y_train)
```

`aggressiveness=4` is chosen because this is my first time optimizing these hyperparameters – I only made one small edit to the hyperparameter search space¹⁰. With `max_iter`, no model sees more than `n_examples` examples as desired and *Hyperband* evaluates (approximately) `n_params` hyperparameter combinations¹¹.

Performance

Two hyperparameter optimizations are compared, *Hyperband* and random search and is shown in Figure 2b. Recall from above that *Hyperband* is a principled early stopping scheme for random search. The comparison mirrors that by sampling the same

¹⁰. For personal curiosity, I changed total number of neurons to 24 from 20 to allow the [12, 6, 3, 3] configuration.

¹¹. Exact specification is available through the `metadata` attribute



Fig. 4: The input and ground truth for the image denoising problem. There are 70,000 images in the output, the original MNIST dataset. For the input, random noise is added to images, and amount of data grows to 350,000 input/output images. Each `partial_fit` calls sees (about) 20,780 examples and each call to `score` uses 66,500 examples for validation.

hyperparameters¹² and using the same validation set for each run. The results of these simulations are in Figure 2.

Dask provides features that the Hyperband implementation can easily exploit. Dask Distributed supports prioritizing different jobs, so it's simple to prioritize the training of different models based on their most recent score. This will emphasize the more adaptive brackets of Hyperband because they are scored more frequently. Empirically, these are the highest performing brackets of Hyperband [LJD⁺18, Section 2.3]. This highlights how Dask is useful to Hyperband and is shown in Figure 3.

Dask's priority of training high scoring models works best in very serial environments: priority makes no difference in very parallel environment when every job can be run. In moderately parallel environments the different priorities may lead to longer time to solution because of suboptimal scheduling. To get around this, the worst performing P models all have the same priority for each bracket when there are P Dask workers.

Parallel Experiments

This section will highlight a using a model implemented with a popular deep learning library, and will leverage Dask's parallelism and investigate how well HyperbandSearchCV scales as the number of workers grows from 8 to 32.

The inputs and desired outputs are given in Figure 4. This is an especially difficult problem because the noise variance varies slightly between images. To protect against this, a shallow neural network is used that's slightly more complex than a linear model. This means hyperparameter optimization is not simple.

Specifically, this section will find the best hyperparameters for a model created in PyTorch¹³ [PGC⁺17] (with the wrapper Skorch¹⁴) for an image denoising task. Again, some detail is mentioned in the appendix and complete details can be found at <https://github.com/stsievert/dask-hyperband-comparison>.

Model architecture & Hyperparameters

Autoencoders are a type of neural network useful for image denoising. They reduce the dimensionality of the input before expanding to the original dimension, which is similar to a lossy compression. Let's create that model and the images it will denoise:

¹². As much as possible – Hyperband evaluates more hyperparameter values. The random search without early stopping evaluates every hyperparameter value Hyperband evaluates.

¹³. <https://pytorch.org>

¹⁴. <https://github.com/skorch-dev/skorch>

```
# custom model definition with PyTorch
from autoencoder import Autoencoder
from dask_ml.model_selection import train_test_split
import skorch # scikit-learn API wrapper for PyTorch

model = skorch.NeuralNetRegressor(Autoencoder, ...)

X, y = noisy_mnist(augment=5)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.05)
```

Of course, this is a neural network so there are many hyperparameters to tune. Only one hyperparameter affects the model architecture: `estimator__activation`, which specifies the activation the neural network should use. This hyperparameter is varied between 4 different choices, all different types of the rectified linear unit (ReLU) [NH10], including the leaky ReLU [MHN13], parametric ReLU [HZRS15a] and exponential linear units (ELU) [CUH15].

The other hyperparameters all control finding the optimal model after the architecture is fixed. These hyperparameters include 3 discrete hyperparameters (with 160 unique combinations) and 3 continuous hyperparameters. Some of these hyperparameters include choices on the optimizer to use (SGD [Bot10] or Adam [KB14]), initialization, regularization and optimizer hyperparameters like learning rate or momentum. Here's a brief summary:

```
params = {'optimizer': ['SGD', 'Adam'], ...}
print(params.keys())
# dict_keys([
#     "optimizer", # 2 choices
#     "batch_size", # 5 choices
#     "module__init", # 4 choices
#     "module__activation", # 4 choices
#     "optimizer__lr", # cnts
#     "optimizer__momentum", # cnts
#     "optimizer__weight_decay" # cnts
# ])
```

Details are in the appendix.

Usage: plateau specification for non-improving models

HyperbandSearchCV supports specifying `patience=True` to make a decision on how long to wait to see if scores stop increasing, as mentioned above. Let's create a HyperbandSearchCV object that stops training non-improving models.

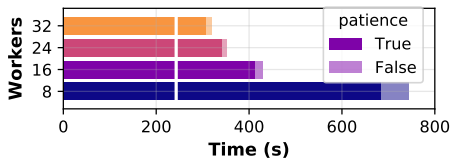
```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(
    model, params, max_iter=max_iter, patience=True)
search.fit(X_train, y_train)
```

The current implementation uses `patience=True` to choose a high value of `patience=max_iter // 3`. This is most useful for the least adaptive bracket of Hyperband (which trains a couple models to completion) and mirrors the patience of the second least adaptive bracket in Hyperband.

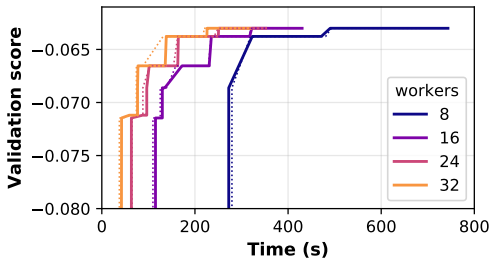
In these experiments, `patience=max_iter // 3` has no effect on performance. If `patience=max_iter // 6` for these experiments, there is a moderate effect on performance (`patience=max_iter // 6` obtains a model with validation loss 0.0637 instead of 0.0630 like `patience=max_iter // 3` and `patience=False`).

Performance

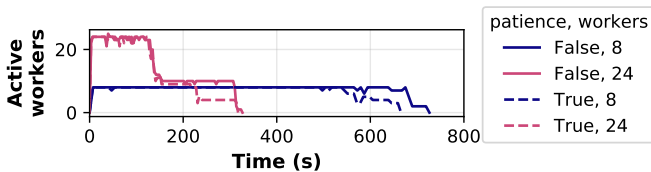
This section will focus on how HyperbandSearchCV scales as the number of workers grow.



(a) The time required to complete the `HyperbandSearchCV` search with a different number of workers for different values of `patience`. The vertical white line indicates the time required to train one model to completion without any scoring.



(b) The time required to obtain a particular validation score (or negative loss) with a different number of Dask workers for `HyperbandSearchCV` with `patience=False` in the solid line and `patience=True` with the dotted line.



(c) The effect that specifying `patience=True` has on `HyperbandSearchCV` for different number of Dask workers.

Fig. 5: In these experiments, the models are trained to completion and their history is saved. Simulations are performed with this history that consume 1 second for a `partial_fit` call and 1.5 seconds for a `score` call. In this simulations, only the number of workers change: the models are static so `Hyperband` is deterministic. The model trained the longest requires 243 seconds to be fully trained, and additional time for scoring.

The speedups `HyperbandSearchCV` can achieve begin to saturate between 16 and 24 workers, at least in this experiment as shown in Figure 5b. Figures 5b and 5c show that `HyperbandSearchCV` spends significant amount of time with a low number of workers without improving the score. Luckily, `HyperbandSearchCV` will soon support keyboard interruptions and can exit early if the user desires.

Specifying `patience=True` for `HyperbandSearchCV` has a larger effect on time-to-solution when fewer workers are used as shown in Figure 5a. A stop-on-plateau scheme will have most effect in very serial environments, similar to the priority scheme used by Dask.

Future work

The biggest area for improvement is using another application of the `Hyperband` algorithm: controlling the dataset size as the scarce resource. This would treat every model as a black box and vary the amount of data provided. This would not require the model to implement `partial_fit` and would only require a `fit` method.

Future work might also include providing an option to further reduce time to solution. This might involve choosing which brackets of `HyperbandSearchCV` to run. Empirically, the best performing brackets are not passive [LJD⁺18, Section 2.3].

Future work specifically does not include implementing the asynchronous version of successive halving [LJR⁺18] in Dask. This variant of successive halving is designed to reduce the waiting time in very parallel environments. It does this by stopping a model's training only if it's in the worst performing fraction of models received so far and does not wait for all models to be collected. Dask's advanced task scheduling helps resolves this issue for `HyperbandSearchCV`.

Regardless of these potential improvements, the implementation of `Hyperband` in Dask-ML allows efficient computation of hyperparameter optimization. The implementation of `HyperbandSearchCV` specifically leverages the abilities of Dask Distributed and can handle distributed datasets.

Appendix

This section expands upon the example given above. Complete details can be found at <https://github.com/stsievert/dask-hyperband-comparison>.

Serial Simulation

Here are some of the other hyperparameters tuned, alongside descriptions of their default values and the values chosen for tuning.

- `alpha`, a regularization term that can affect generalization. This value defaults to 10^{-4} and is tuned logarithmically between 10^{-6} and 10^{-3}
- `batch_size`, the number of examples used to approximate the gradient at each optimization iteration. This value defaults to 200 and is chosen to be one of [32, 64, ..., 512].
- `learning_rate` controls the learning rate decay scheme, either constant or via the "invscaling" scheme, which has the learning rate decay like γ_0/t^p where p and γ_0 are also tuned. γ_0 defaults to 10^{-3} and is tuned logarithmically between 10^{-4} and 10^{-2} . p defaults to 0.5 and is tuned between 0.1 and 0.9.
- `momentum`, the amount of momentum to include in Nesterov's momentum [Nes13]. This value is chosen between 0 and 1.

The learning rate scheduler used is not Adam [KB14] because it claims to be most useful without tuning and has reportedly has marginal gain [WRS⁺17].

Parallel Experiments

Here are some of the other hyperparameters tuned:

- `optimizer`: which optimization method should be used for training? Choices are stochastic gradient descent (SGD) [Bot10] and Adam [KB14]. SGD is chosen with 5/7th probability.
- `estimator__init`: how should the estimator be initialized before training? Choices are Xavier [GB10] and Kaiming [HZRS15b] initialization.
- `batch_size`: how many examples should the optimizer use to approximate the gradient? Choices are [32, 64, ..., 512].

- `weight_decay`: how much of a particular type of regularization should the neural net have? Regularization helps control how well the model performs on unseen data. This value is chosen to be zero 1/6th of the time, and if not zero chosen uniformly at random between 10^{-5} and 10^{-3} logarithmically.
- `optimizer_lr`: what learning rate should the optimizer use? This is the most basic hyperparameter for the optimizer. This value is tuned between $10^{-1.5}$ and 10^1 after some initial tuning.
- `optimizer_momentum`, which is a hyper-parameter for the SGD optimizer to incorporate Nesterov momentum [Nes13]. This value is tuned between 0 and 1.

REFERENCES

- [B⁺15] Sébastien Bubeck et al. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning*, 8(3-4):231–231, 2015.
- [BB12] James Bergstra and Yoshua Bengio. Random search for hyperparameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–281, 2012. URL: <http://jmlr.csail.mit.edu/papers/v13/bergstra12a.html>.
- [Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187. Springer, Paris, France, August 2010. URL: <http://leon.bottou.org/papers/bottou-2010>.
- [Bot12] Léon Bottou. Stochastic gradient tricks. In Grégoire Montavon, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks, Tricks of the Trade, Reloaded*, Lecture Notes in Computer Science (LNCS 7700), pages 430–445. Springer, 2012. URL: <http://leon.bottou.org/papers/bottou-tricks-2012>.
- [CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.
- [Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: <https://dask.org>.
- [FKH18] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. 80:1437–1446, 10–15 Jul 2018. URL: <http://proceedings.mlr.press/v80/falkner18a.html>.
- [GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.
- [hut11] *Sequential model-based optimization for general algorithm configuration*, volume International Conference on Learning and Intelligent Optimization. Springer, 2011. doi:10.1007/978-3-642-25566-3_40.
- [HZRS15a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [HZRS15b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [KCG16] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *Journal of Machine Learning Research*, 17(1):1–42, 2016. URL: <http://jmlr.org/papers/v17/kaufman16a.html>.
- [KFB⁺16] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016. URL: <https://arxiv.org/abs/1605.07079>.
- [KFMH17] A. Klein, S. Falkner, N. Mansur, and F. Hutter. Robo: A flexible and robust bayesian optimization framework in python. In *NIPS 2017 Bayesian Optimization Workshop*, December 2017. URL: <https://github.com/automl/RoBO>.
- [LJD⁺18] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL: <http://jmlr.org/papers/v18/16-558.html>.
- [LJR⁺18] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonnina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.
- [MH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. URL: <http://jmlr.csail.mit.edu/papers/v9/vandermaaten08a.html>.
- [MH15] Maren Mahseredci and Philipp Hennig. Probabilistic line searches for stochastic optimization. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems* 28, pages 181–189. Curran Associates, Inc., 2015. URL: <http://papers.nips.cc/paper/5753-probabilistic-line-searches-for-stochastic-optimization.pdf>.
- [MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.
- [MS75] Donald W. Marquardt and Ronald D. Snee. Ridge regression in practice. *The American Statistician*, 29(1):3–20, 1975. doi:10.1080/00031305.1975.10479105.
- [Nes13] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013. doi:10.1007/978-1-4419-8853-9.
- [NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [PBBW12] F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors. *Practical Bayesian Optimization of Machine Learning Algorithms*. Curran Associates, Inc., 2012. URL: <http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf>.
- [PGC⁺17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017. URL: <https://openreview.net/pdf?id=BJjrmfCZ>.
- [Pre98] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998. doi:10.1016/S0893-6080(98)00010-0.
- [PVG⁺11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, and Vincent Dubourg. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011. URL: <http://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html>.
- [STZB⁺11] J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors. *Algorithms for Hyper-Parameter Optimization*. Curran Associates, Inc., 2011. URL: <http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf>.
- [Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. doi:10.1111/j.2517-6161.1996.tb02080.x.
- [WRS⁺17] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.
- [WVJ16] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016. URL: <http://distill.pub/2016/misread-t-sne>, doi:10.23915/distill.00002.