Better and faster model selection with Dask

Scott Sievert[‡§*], Tom Augspurger[¶], Matthew Rocklin[‖**]

Abstract—Nearly every machine learning estimator has parameters assumed to be given. Finding the optimal set of these values is a difficult and time-consuming process for most modern estimators and is called "model selection". A recent breakthrough model selection algorithm Hyperband addresses this problem by finding high performing estimators with minimal training and has theoretical backing. This paper will explain why Hyperband is well-suited for Dask, the required input parameters and the rationale behind some minor modifications. Experiments with a neural network will be used for illustration and comparison.

Index Terms—machine learning, model selection, distributed, dask

Introduction

Training any machine learning pipeline requires data, an untrained model or estimator and parameters that change the model and data, a.k.a. "hyper-parameters". These hyper-parameters greatly influence the performance of the model but are typically assumed to be given. A good example is with adapting the ridge regression or LASSO models to the amount of noise in the data with the regularization parameter [MS75] [Tib96].

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

These hyper-parameters are typically assumed to be given. This requires searching over the possible values to find the best value by some measure Typically models are scored on unseen data through a "cross-validation" search. These searches are part of a "model selection" process because hyper-parameters are considered part of the model. Even in the simple ridge regression case above, a brute force search is required [MS75].

This gets more complex with many different hyper-parameter values to input, and especially because there's often an interplay between hyper-parameters. A good example is with deep learning, which has specialized techniques for handling many data [Bot10]. However, these optimization methods can't provide basic hyper-parameters because there are too many data. For example, the most basic hyper-parameter "learning rate" or "step size" is straightforward with few data but infeasible with many data [MH15].

* 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.

Contributions

A hyper-parameter search a.k.a "model selection" is required if high performance is desired. In practice, it's a burden for machine learning researchers and practitioners. Ideally, model selection algorithms return high performing models quickly and are simple to use.

Returning high performing models quickly will allow the user (e.g., a data scientist) to more easily use model selection. Having a better search method will allow them to easily find high performing models and remove the need for repeating model selection searches to obtain a better model. Returning this high performing model quickly would lower the barrier to performing model selection.

This work

- provides implementation of a particular model selection algorithm, Hyperband in Dask, 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 to Hyperband, which only requires knowing how many examples the model should observe and a rough estimate on how many parameters to sample
- provides validating experiments

Hyperband treats computation as a scarce resource[1] and has parallel underpinnings. Hyperband can only return high performing models with minimal training because it evaluate models in parallel.

In the experiments, Hyperband returns high performing models fairly quickly, with the simple heuristic for determining the input parameters to Hyperband. The implementation can be found on the machine learning for Dask, Dask-ML. The documentation for Dask-ML is available at https://ml.dask.org.

This paper will review other existing work for model selection before detailing the Hyperband implementation in Dask. A realistic set of experiments will be presented before mentioning ideas for future work.

1. If computation is not a scarce resource, there is no benefit from this algorithm.

Related work

Software for model selection

Model selection finds the optimal set of hyper-parameters for a given model. These hyper-parameters are chosen to maximize performance on unseen data. The typical model selection process

1) splits the dataset into the train dataset and test dataset. The test dataset is reserved for the final model evaluation.
2) chooses hyper-parameters
3) trains models with those hyper-parameters
4) scores those models with unseen data (a subset of the train dataset typically referred to as the "validation set")
5) trains the best model with the complete train dataset
6) scores the model on the test dataset. This score is reported as the models score.

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

A commonly used method for hyper-parameter selection is a random selection of hyper-parameters 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 over unimportant parameters [BB12] This randomized search is implemented in many places, including in Scikit-Learn [PVG+11].

These implementations do not adapt to previous training, and are by definition *passive*. *Adaptive* algorithms can return a higher quality solution in less time by adapting to previous training and choosing which hyper-parameters to sample. This is especially useful for difficult model selection problems with many hyper-parameters and many values for each hyper-parameter.

Bayesian algorithms are popular as adaptive model selection algorithms. These algorithms treat the model as a black box and the model scores as a noisy evaluation of that black box. These algorithms try to tune a set of hyper-parameters over time given serial evaluations of the black box.

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 AutoML[2]. This package also includes Fabolas, a method that takes dataset size as input and allows for some computational control [KFB+16].

Hyperband

Hyperband is an adaptive model selection algorithm [LJD+18], and a principled early stopping scheme[3] for randomized hyper-parameter selections. At the most basic level, it partially trains models before stopping models with low scores, then repeats. By default, it stops training the bottom 33% of the available models on each iteration. This means that the number of models decay over time, and the surviving models have high scores.

The amount of training to do before stopping models depends on the relative importance of training time and hyper-parameters. If training time only matters a little, it makes sense to aggressively stop training models. On the flip side, if only training time influence the score, it only makes sense to let all models train for as long as possible and not perform any stopping.

This sweep over training time importance enables a formal mathematical statement 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:

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

Higher values of α mean slower convergence, and higher values of β represent more difficult model selection 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].

Then for any $T \in \mathbb{N}$, let $\widehat{i}_T$ 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 $\widehat{i}_T$ has loss

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

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

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

For simplicity, only the infinite horizon case is presented though much of the analysis carries over to the practical finite horizon Hyperband.[4] 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 $v_{\widehat{i}_T}$ is much smaller than the uniform allocation scheme.[5] 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 number of pulls in Theorem 7 [LJD+18], within log factors of the known lower bound on number of resources required [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

2. https://github.com/automl/

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

4. 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 to Theorem 1 but it's left as an exercise for the reader.

5. This is clear by examining $\log(v_{\widehat{i}_T} - v_*)$ 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 $-1/(\alpha+\beta)$.

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 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 lower 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 in mind and provides rapid feedback and diagnostics to aid humans.

Adaptive model selection in Dask

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

This work focuses on the case when the computation required is not insignificant. Then, the existing *passive* model selection algorithms in Dask-ML have limited use because they don't *adapt* to previous training to reduce the amount of training required.[6]

An adaptive model selection algorithm, Hyperband is implemented in Dask's machine learning library, Dask-ML.[7] This algorithm adapts to previous training to minimize the amount of computation required. This section will detail the Hyperband architecture, the input arguments required and some modifications to reduce time to solution.

Hyperband architecture

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

- an "embarrassingly parallel" sweep over the different brackets of the hyper-parameter vs. training time importance
- in each bracket, the models are trained independently (though the training of low performing models ceases at particular times)

6. Though 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 at https://ml.dask.org/hyper-parameter-search.html.

7. The documentation the Hyperband implementation can be found at https://ml.dask.org.

Of course, the number of models in each bracket decrease 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) -> 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 "hyper-parameter" 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:
        assert brackets == [(81, 3), (34, 9),
                            (15, 27), (8, 81),
                            (5, 243)]
    final_models = [sha(n, r) for n, r in brackets]
    return top_k(final_models, k=1)
```

In this pseudo-code, the train set and validation data are hidden, which `train` and `top_k` rely on. `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 tradeoff between hyper-parameter and training time importance. With `max_iter=243`, the least adaptive bracket runs 5 models until completion and the most adaptive bracket aggressively prunes off 81 models.

This architecture with many embarrassingly parallel for-loops and nested parallelism lends itself well to Dask, an advanced distributed scheduler that can handle many concurrent jobs. Dask can exploit the parallelism present in this algorithm and train models from different brackets concurrently.

Dask Distributed is required because of the nested parallelism and the decision to stop training low-performing models. This means the computational graph is dynamic and depends on other nodes in the graph.

Input parameters

Hyperband is fairly easy to use as well. It only 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 model[8] 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`. There is a third parameter that controls the aggressiveness of the search and stopping model training, but it's optional and has some theoretical backing.

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. An balance between training time and hyper-parameter importance is implicitly being decided upon. Hyperband has one fewer input because it sweeps over this balance's importance.

Dwindling number of models

At first, Hyperband evaluates many models. As time progresses, the number of models decay because Hyperband is a (principled) early stopping scheme. Hyperband varies how aggressively to stop model training per bracket. Each bracket performs something like a binary search but varies the amount of training between each decision. The least aggressive bracket lets a few models run without any stopping.

This means towards the end of the computation, a few 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 Engine).

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

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

These common use cases happen when the user specifies a poor set of hyper-parameters or that training continue for too long. Regardless, the scores of the models above will not increase too much with high probability.

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.

The current implementation uses `patience=True` to choose a high value of `patience=max_iter // 3`, which is validated by the experiments.

Experiments

This section will highlight a practical use of `HyperbandSearchCV`. This involves a neural network using a popular library (PyTorch[10] [PGC+17] through the wrapper Skorch[11]). This is a difficult model selection problem even for this relatively simple model. Some detail is mentioned in the Appendix, though complete details can be found at https://github.com/stsievert/dask-hyperband-comparison.

Problem

This section will walk through an image denoising task. The inputs and desired outputs are given in Figure 1. This is an especially difficult problem because the noise variance varies slightly between images, which requires a model that's at least a little complex.

Model architecture & Parameters

To address that complexity, let's use an autoencoder. These are a type of neural network that reduce the dimensionality of the input before expanding to the original dimension. This can be thought of a lossy compression. Let's create that model:

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

# definition in Appendix
est = skorch.NeuralNetRegressor(Autoencoder, ...)
```

Of course, this is a neural network so there are many hyper-parameters to tune. Only one effects the global optimum:

- `estimator__activation`: which specifies the activation the neural network should use.

This hyper-parameter is varied between 4 different choices, all different types of the rectified linear unit (ReLU). The other hyper-parameters control reaching the global optimum:

- `optimizer`: which optimization method should be used for training? Choices are stochastic gradient descent (SGD) and Adam.
- `estimator__init`: how should the estimator be initialized before training? Choices are Xavier and Kaiming initialization.
- `batch_size`: how many examples should the optimizer use to approximate the gradient? Choices include values between 32 and 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.
- `optimizer__lr`: what learning rate should the optimizer use? This is the most basic hyper-parameter for the optimizer.
- `optimizer__momentum`, which is a hyper-parameter for the SGD optimizer.

There are 4 discrete variables with 160 possible combinations. For each one of this combinations, there are 3 continuous variables to tune. Let's create the parameters to search over:

```
# definition in Appendix
params = {'optimizer': ['SGD', 'Adam'], ...}
```

The goal for model selection is to find a high performing estimator quickly is easy usage.

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.

10. https://pytorch.org

11. https://github.com/skorch-dev/skorch

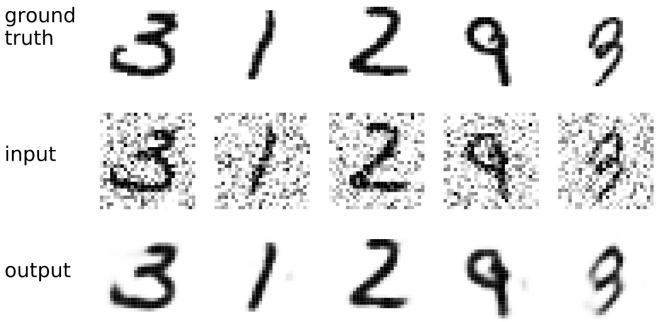

Fig. 1: *The rows show in the ground truth, input and output respectively for the denoising problem. The output is shown for the best model that Hyperband finds.*

Label	Class
`hyperband`	`HyperbandSearchCV`
`stop-on-plateau`	`IncrementalSearchCV, patience=24`
`hyperband+sop`	`HyperbandSearchCV, patience=True`

TABLE 1: *A summary of the legends in Figures 2, 3 and 4. `IncrementalSearchCV patience=24` is an algorithm that stops training after the scores stop increasing or plateau, hence the label.*

Usage

First, let's create a `HyperbandSearachCV` object:

```
from dask_ml.model_selection import HyperbandSearchCV
search = HyperbandSearchCV(est, params, max_iter=243)
search.fit(X_train, y_train)
search.best_score_
# -0.0929. Best of hand tuning: -0.098
```

This model has denoised series of image it's never seen before in Figure 1.

`HyperbandSearchCV` beat hand-tuning by a considerable margin. While manually tuning, I considered any scores about -0.10 to be pretty good, and I obtained scores no higher than -0.098. By that measure, a score of -0.093 is fantastic.

`HyperbandSearchCV` only requires *one* parameter besides the model and data as discussed above. This number controls the amount of computation that will be performed, and does not require balancing between the number of models and how long to train each model.

Performance

Let's compare three algorithms with the same model, parameters and validation data. The comparisons are shown in Figures 2, 3 and 4 and the legends for these plots is shown in Table 1. In these experiments, 25 workers are used with Dask, meaning that 25 tasks can complete in parallel.

I will compare against a basic stop on plateau algorithm with particular choices for `patience` and `num_params`. Specifically, I choose a fairly aggressive value for `patience` and hence choose to evaluate twice as many hyper-parameters. This illustrates the choice between hyper-parameter vs. training time importance because training models for longer with the same computational effort would require a higher value for `num_params` and a lower and more aggressive of `patience`.

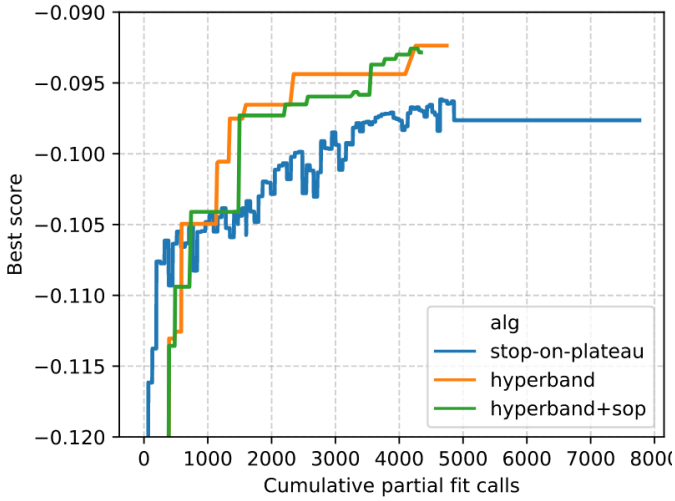

Fig. 2: *The number of `partial_fit` calls against the empirically best score (or negative loss). The legend labels are in Table 1.*

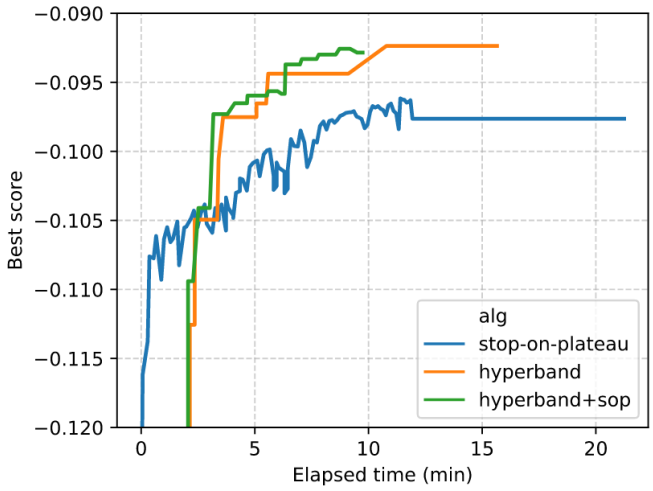

Fig. 3: *The time required to obtain a particular accuracy. The legend labels are in Table 1.*

Figure 2 supports the claim that Hyperband will high performing models with minimal `partial_fit` calls. Each `partial_fit` call uses 1/3 of the dataset, so algorithm passes over the training data about 1,667 times in total, a.k.a. 1,667 epochs. Each model sees no more than 81 times the number of examples in the dataset because `max_iter=243` for all searches.

However, the data scientist cares about time to reach a particular score, not the number of `partial_fit` calls required. This plot is shown in Figure 3. This plot is shown with 25 workers; if only one worker had been used this plot in Figure 3 would be the same as Figure 2 up to the x-axis labeling.

The difference between Figures 2 and 3 show a remarkable difference of specifying `patience` for Hyperband: specifying `patience=True` means that Hyperband finishes in about 2/3rds of the time as the default Hyperband! This is because one worker hold onto a single model for about 4 minutes as shown in Figure 4. Specifying `patience=True` removes that behavior, and likely removes that model.

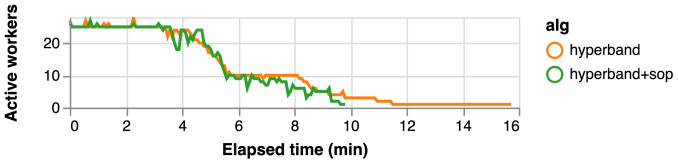

Fig. 4: *The activity over time for the 25 Dask workers.*

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.

Another area of future work is ensuring `IncrementalSearchCV` and all of it's children (including `HyperbandSearchCV`) work well with large models. Modern models often consume most of GPU memory, and currently `IncrementalSearchCV` requires making a copy the model. How much does this hurt performance and can it be avoided?

Appendix

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

Test/train data

```
import noisy_mnist
noisy, clean = noisy_mnist.dataset()

from dask_ml.model_selection import train_test_split
_ = train_test_split(X, y)
X_train, X_test, y_train, y_test = _
```

Model

```
import torch.nn as nn

class Autoencoder(nn.Module):
    def __init__(
        self,
        activation='ReLU',
        init='xavier_uniform_'
    ):
        super().__init__()

        self.activation = activation
        self.init = init

        Actvation = getattr(nn, activation)
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, inter_dim),
            Activation(),
            nn.Linear(inter_dim, latent_dim),
            Activation()
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, 28 * 28),
            nn.Sigmoid()
        )
        # code to handle initialization

    def forward(self, x):
        self._iters += 1
        shape = x.size()
        x = x.view(x.shape[0], -1)
        x = self.encoder(x)
        x = self.decoder(x)
        return x.view(shape)
```

Input parameters

```
params = {
    'optimizer': ['SGD', 'Adam'],
    'batch_size': [32, 64, 128, 256, 512],
    'estimator__init': ['xavier_uniform_',
                        'xavier_normal_',
                        'kaiming_uniform_',
                        'kaiming_normal_'],
    'estimator__activation': ['ReLU',
                              'LeakyReLU',
                              'ELU',
                              'PReLU'],
    'optimizer__lr': \
            np.logspace(1, -1.5, num=1000),
    'optimizer__weight_decay': \
            np.logspace(-5, -3, num=1000),
    'optimizer__momentum': \
            np.linspace(0, 1, num=1000)
}
```

References

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter 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.

[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.

[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.

[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.

[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 Mahsereci 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.

[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.

[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=BJJsrmfCZ.

[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.

[WVJ16] Martin Wattenberg, Fernanda Viégas, and Ian Johnson. How to use t-sne effectively. *Distill*, 2016. URL: http://distill.pub/2016/misread-tsne, doi:10.23915/distill.00002.