SAS1941-2018

Managing the Expense of Hyperparameter Autotuning

Patrick Koch, Brett Wujek, and Oleg Golovidov

SAS Institute Inc.

ABSTRACT

Determining the best values of machine learning algorithm hyperparameters for a specific data set can be a difficult and computationally expensive challenge. The recently released AUTOTUNE statement and **autotune** action set in SAS® Visual Data Mining and Machine Learning automatically tune hyperparameters of modeling algorithms by using a parallel local search optimization framework to ease the challenges and expense of hyperparameter optimization. This implementation allows multiple hyperparameter configurations to be evaluated concurrently, even when data and model training must be distributed across computing resources because of the size of the data set.

With the ability to both distribute the training process and parallelize the tuning process, one challenge then becomes how to allocate the computing resources for the most efficient autotuning process. The best number of worker nodes for training a single model might not lead to the best resource usage for autotuning. To further reduce autotuning expense, early stopping of long-running hyperparameter configurations that have stagnated can free up resources for additional configurations. For big data, when the model training process is especially expensive, subsampling the data for training and validation can also reduce the tuning expense. This paper discusses the trade-offs that are associated with each of these performance-enhancing measures and demonstrates tuning results and efficiency gains for each.

INTRODUCTION

Machine learning predictive modeling algorithms are governed by “hyperparameters” that have no clear defaults agreeable to a wide range of applications. The *depth* of a decision tree, *number of trees* in a forest or a gradient boosting tree model, *number of hidden layers* and *neurons in each layer* in a neural network, and *degree of regularization* to prevent overfitting are a few examples of quantities that must be prescribed. Not only do ideal settings for the hyperparameters dictate the performance of the training process, but more importantly they govern the quality of the resulting predictive models. Tuning hyperparameter values is a critical aspect of the model training process and is considered to be a best practice for a successful machine learning application (Wujek, Hall, and Güneş 2016). Manual hyperparameter adjustment and rough grid search approaches to tuning are recently being traded for automated intelligent search strategies. Random search has been shown to perform better than grid search, particularly when the number of influential hyperparameters is low (Bergstra and Bengio 2012). With increased dimensionality of the hyperparameter space (that is, as more hyperparameters require tuning), a manual tuning process becomes much more difficult even for experts, grid searches become more coarse and less practical because they grow exponentially with dimensionality, and random search requires many more samples to identify candidate models with improved accuracy. As a result, numerical optimization strategies for hyperparameter tuning have become more popular for intelligent search of complex hyperparameter spaces (Bergstra et al. 2011; Eggensperger et al. 2013). Optimization for hyperparameter tuning typically can very quickly reduce, by several percentage points, the model error that is produced by default settings of these hyperparameters. Parallel tuning allows exploration of more configurations, further refining hyperparameter values and leading to additional improvement.

SAS® Visual Data Mining and Machine Learning, described in Wexler, Haller, and Myneni (2017), provides a hyperparameter autotuning capability that is built on SAS® local search optimization (LSO). SAS LSO is a hybrid derivative-free optimization framework that operates on the SAS® Viya® distributed analytics execution engine to overcome the challenges and expense of hyperparameter optimization. This implementation of autotuning, detailed in Koch et al. (2017), is available in the TREESPLIT, FOREST, GRADBOOST, NNET, SVMACHINE, and FACTMAC procedures by using the AUTOTUNE statement. Statement options define tunable hyperparameters, default ranges, user overrides, and validation schemes to avoid overfitting. The procedures that incorporate the AUTOTUNE statement invoke corresponding actions in the **autotune** action set. These actions (**tuneDecisionTree**, **tuneForest**,

tuneGradientBoostTree, **tuneNeuralNet**, **tuneSvm**, and **tuneFactMac)** can also be executed directly on SAS Viya.

As shown in Figure 1, the LSO framework consists of an extendable suite of search methods that are driven by a hybrid solver manager that controls concurrent execution of search methods. Objective evaluations (different model configurations in this case) are distributed across multiple worker nodes in a compute cluster and coordinated in a feedback loop that supplies data from running search methods. As illustrated in Figure 2, the autotuning capability in SAS Visual Data Mining and Machine Learning uses a default hybrid search strategy that begins with a Latin hypercube sample (LHS), which provides a more uniform sample of the hyperparameter space than a grid or random search provides. The best configurations from the LHS are then used to seed a genetic algorithm (GA), which crosses and mutates the best samples in an iterative process to generate a new population of model configurations for each iteration. The strengths of this approach include handling continuous, integer, and categorical variables; handling nonsmooth, discontinuous spaces; and ease of parallelizing the search strategy. All of these challenges are prevalent and critical in hyperparameter tuning problems. Alternate search methods include a single Latin hypercube sample, a purely random sample, and a Bayesian search method. It is important to note here that the LHS or random samples can be evaluated in parallel and that the GA population or Bayesian samples at each iteration can be evaluated in parallel.

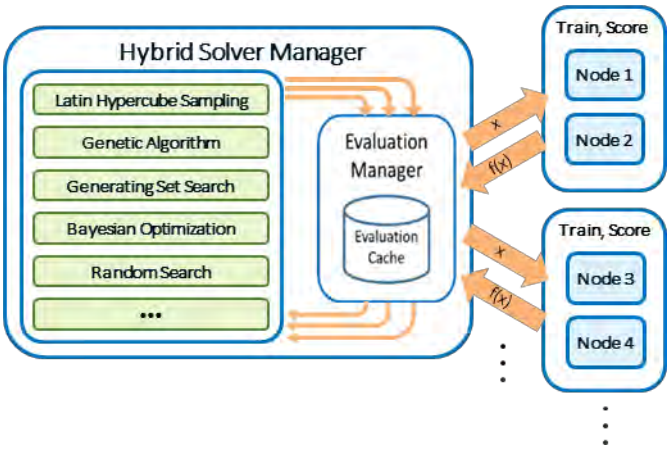


Figure 1. Autotuning with Local Search Optimization: Parallel Hybrid Derivative-Free Optimization Strategy

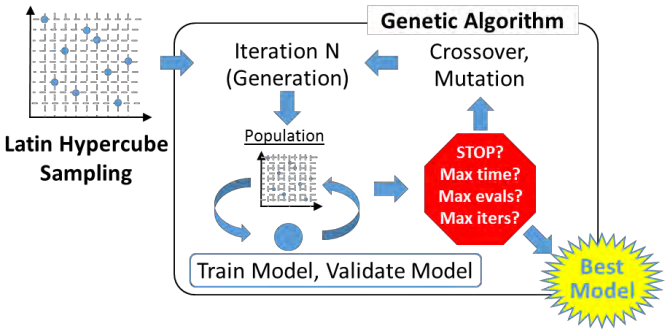


Figure 2. Default Autotuning Process in SAS Visual Data Mining and Machine Learning

An automated, parallelized, intelligent search strategy can benefit both novice and expert machine learning algorithm users. Challenges still exist, however, particularly related to the expense of hyperparameter tuning. Primary contributors to the expense of hyperparameter tuning are discussed in the next section. Options to manage these expenses within the SAS autotuning implementation are then presented in the following section, with examples that demonstrate expense management trade-offs. Best-practice recommendations are offered in conclusion.

HYPERPARAMETER TUNING EXPENSES

Even when a compute cluster is used both to distribute large data sets for model training and to concurrently evaluate multiple model hyperparameter configurations in parallel, hyperparameter tuning is a computationally expensive process. Often many configurations must be evaluated in pursuit of a high-quality model. One challenge becomes deciding how to best allocate compute resources. The LSO-driven hyperparameter process in Figure 1 depicts the use of two worker nodes for each model training, with multiple models trained in parallel. Are two worker nodes per model training necessary? Ideal? Figure 3 illustrates different possibilities for hyperparameter tuning on a compute cluster that has 8 worker nodes. If all 8 worker nodes are used for each model training, the training time might be reduced, but the tuning process becomes sequential. A sample of 8 hyperparameter configurations could all be evaluated in parallel, with one worker evaluating each configuration, without overloading the cluster, but the size of the data set might demand more workers to train a model. Perhaps allocating four workers for model training and training two models in parallel, or allocating two workers for model training and training four models in parallel, is appropriate. The best worker allocation for hyperparameter tuning depends on the training expense, the savings observed with parallel tuning, the size of the cluster, and to some degree the hyperparameter ranges (which dictate how complex the models become). The best number of worker nodes for a single model training might not lead to the best resource usage for autotuning.

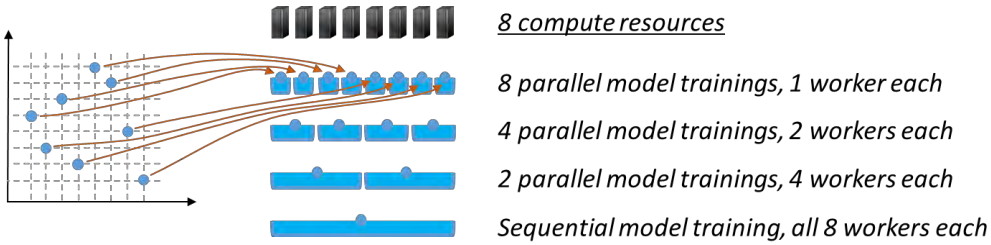
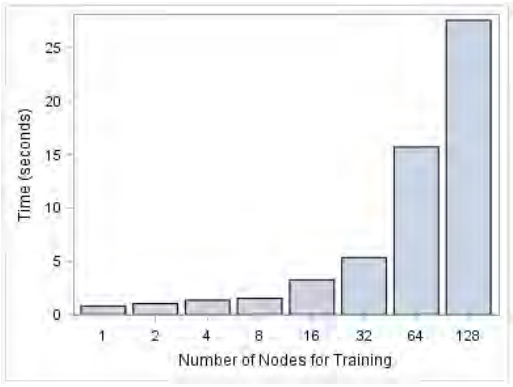
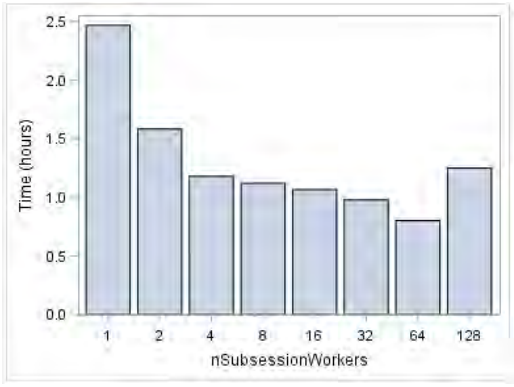


Figure 3. Use of Compute Resources for Tuning Many Models

One extreme case for resource allocation that might not be immediately obvious is that of a small data set. As shown in Figure 4(a), the training expense actually increases when the number of worker nodes is increased. The expense of communication between nodes adds to the training expense, which is most efficient when all the data are on a single node for small data sets. In this case, available workers should be used for parallel tuning of different hyperparameter configurations for increased efficiency of tuning—with as many models trained concurrently as possible or desired. However, as data sets grow, both in length and width (many inputs to a model can have a larger effect on training expense than many observations), training time is reduced by increasing the number of worker nodes, up to a certain number of workers. When the number of workers passes some threshold, the communication cost again leads to an increase in training time, as shown in Figure 4(b). In this case, resource allocation is not straightforward. Even though a single model training is most efficient on 64 workers, tuning might not be most efficient if every hyperparameter configuration to be evaluated uses 64 workers. If the cluster contains 128 workers, only two models could be evaluated in parallel during tuning without overloading the cluster. Furthermore, different hyperparameter configurations vary in expense; fewer hidden layers and neurons in a neural network or fewer trees in a forest are more efficient to train. Most importantly, however, the training expense shown in Figure 4(b) with 64 workers is not half the expense with 32 workers. In fact, it is slightly more than half the expense of training on two workers. If each model to be trained uses two workers, the cluster of 128 workers would accommodate 64 models trained in parallel during tuning rather than two models in parallel if 64 workers are used for each model training. More hyperparameter configurations can be evaluated in the same amount of time, or less time is needed for evaluating a specific number of hyperparameter configurations.



(a) 150 observations, 5 columns



(b) 50,000 observations, 3073 columns

Figure 4. Training Expense for Data Sets of Different Size

Although training multiple hyperparameter configurations in parallel can significantly reduce the expense of tuning, there are additional contributing factors to consider. First, much of the expense of hyperparameter tuning is spent on model configurations that are not only worse than the current best model (or even the default model), but are often quite bad. As shown in Figure 5, although the best configuration from a Latin hypercube sampling of model candidates has a 6% misclassification rate, most have more than 10% error, many have more than 20% error, and quite a few have worse than 40% error. The use of an intelligent search strategy that is designed to *learn* over multiple iterations, such as the default strategy in the LSO framework described previously, helps reduce the number of bad configurations over time. Still, significant expense can be incurred to complete the training process for model configurations that might have stagnated (ceased to make meaningful improvement) before the training process has completed. Training all model configurations to completion (beyond the point of stagnation) is especially costly for large data sets and complex model configurations (which can delay the completion of an iteration, because all candidate models within an iteration must complete training before the next iteration can start). Ideally, stagnated configurations are identified and the time spent training these models is reduced. Furthermore, the expense of model training, compounded in model tuning, is also obviously tied to the size of the data set. This is clear in Figure 4 where the smaller data set training time is measured in seconds and the larger data set training time is measured in hours; training on the entire data set during tuning might not be necessary and might not be the most efficient approach.

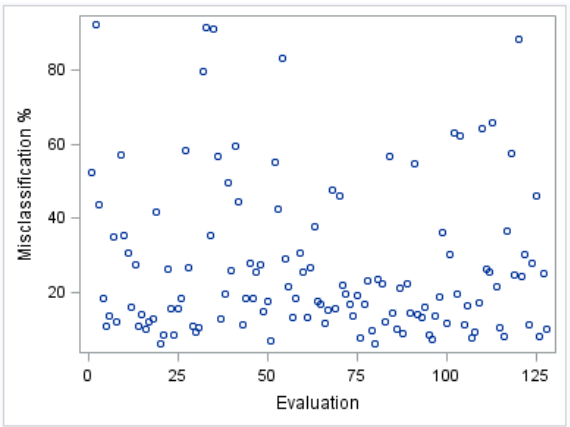


Figure 5. Latin Hypercube Sample of Candidate Models—Many Bad Configurations

EFFICIENT AUTOTUNING ON SAS VIYA

Given the various factors that contribute to the expense of hyperparameter tuning and the trade-offs to be made based on the data set and compute resources at hand, flexibility is required for an efficient autotuning solution. SAS Viya actions are executed within a "session" that uses one or more worker nodes. The autotuning implementation running on SAS Viya creates additional "subsessions," which are managed from the parent session, in order to facilitate parallel training of different model configurations by isolating each alternate configuration within a separate subsession with its own set of worker nodes. SAS Viya automatically handles the data management for execution in subsessions. The autotuning implementation enables control of the expense of hyperparameter tuning through the following:

- resource allocation of worker nodes for training versus tuning
- early stopping of stagnated models
- subsampling large data sets for faster training times

Table 1 shows the procedure options and corresponding action parameters that correspond to these controls. They are discussed further in this section, with results from demonstration problems provided to illustrate their effectiveness and associated trade-offs. All these controls are configured with defaults that are designed to reduce the anticipated expense of the autotuning process based on the data set size and the available compute resources, and they can be adjusted further to trade off the tuning expense and the accuracy of models that are generated. For simplicity in the following text, when a control can be specified either in a procedure option or in a corresponding action parameter, the control is presented only by the procedure option name and syntax (in all capital letters).

	Procedure Options	Action Parameters
Resource Allocation	NSUBSESSIONWORKERS	nSubsessionWorkers
	NPARALLEL	nParallel
Early Stopping	EARLYSTOP, STAGNATION, VALIDATION	earlyStop
Subsampling	PARTITION	trainFraction, validateFraction

Table 1. Autotuning Efficiency Controls

Before each of these controls for managing the expense of autotuning is discussed in more detail, an example is provided here to familiarize you with the associated syntax, for both the GRADBOOST procedure and for the **autotune.tuneGradientBoostTree** action. All procedures that include the AUTOTUNE statement—TREESPLIT, FOREST, GRADBOOST, NNET, SVMACHINE, and FACTMAC—include the NSUBSESSIONWORKERS and NPARALLEL options, which, in addition to the POPSIZE option, can be used to adjust the resource allocation for tuning. The example here uses a small data set, so the number of workers per subsession (NSUBSESSIONWORKERS) for model training is set at 1 (which is the default for this data set size) and the number of parallel model configurations (NPARALLEL) is adjusted to match a cluster size of 30 workers. The population size (POPSIZE) is also increased to make full use of the compute resources; it is set to 31 (with the default search method, the best model from the previous iteration is included but does not need to be retrained). The procedure or action results in a maximum of 150 configurations being evaluated with five iterations (the default). Early stopping (EARLYSTOP) is activated, directing the modeling algorithms to terminate training if they stagnate for four consecutive iterations. With the procedures, the PARTITION statement can be used to implement subsampling of training data (not necessary in this small data set case, but shown for illustration). With a 0.2 fraction defined for TEST and 0.3 for VALIDATE, half the data (0.5) will be used for training. If the TEST fraction is not defined, the fraction used for training would be 0.7.

```
cas mysess sessopts=(nworkers=1);

libname mycaslib sasioca casref=mysess;

data mycaslib.dmagecr;
    set sampsio.dmagecr;
run;

proc gradboost data=mycaslib.dmagecr outmodel=mycaslib.mymodel
               earlystop(stagnation=4);
    partition fraction(test=0.20 validate=0.30);
    target good_bad / level=nominal;
    input checking duration history amount savings employed installp
        marital coapp resident property age other housing existcr job
        depends telephon foreign / level=interval;
    input purpose / level=nominal;
    autotune nsubsessionworkers=1 nparallel=30 popsize=31
             evalhistory=all;
run;
```

The number of parallel evaluations and worker nodes for each evaluation is reported in a log note when this code runs; if the NPARALLEL option was not specified in the procedure call, this note indicates the automated decision for the resource allocation.

```
NOTE: Autotune number of parallel evaluations is set to 30, each using 1
worker nodes.
```

After execution, if the model that contains the best found hyperparameter configuration terminated early as a result of stagnation, a log note indicates how many trees were used in the final model (which will be less than the value selected by the tuner during tuning).

```
NOTE: Due to early stopping, the actual final number of trees used in the
model (19) is less than the Autotune selected 'best' value (75).
```

The following **tuneGradientBoostTree** action call is equivalent to the PROC GRADBOOST call. In this action call, all the parameters for managing the tuning expense are provided in the *tunerOptions* parameter, except for the *earlyStop* parameter, which is available only in the **tuneGradientBoostTree** and **tuneNeuralNet** actions.

```
proc cas noqueue;
    autotune.tuneGradientBoostTree /
        tunerOptions={
            nSubsessionWorkers=1, nParallel=30, popsize=31,
            trainFraction=0.50, validateFraction=0.30, loglevel=3
        },
        earlyStop=true,
        trainOptions={
            table={name='dmagecr'},
            inputs={'checking', 'duration', 'history', 'amount',
                    'savings', 'employed', 'installp', 'marital',
```

```
                    'coapp', 'resident', 'property', 'age', 'other',
                    'housing', 'existcr', 'job', 'depends',
                    'telephon', 'foreign', 'purpose'},
                target='good_bad',
                nominals={'purpose', 'good_bad'},
                casout={name='dmagecr_gbt_model', replace=true}
            }
        ;
    run;
    quit;
```

An additional log note is provided with the action execution in this case because early stopping was not explicitly included. By default, the **tuneGradientBoostTree** action includes early stopping with *stagnation*=4. If the EARLYSTOP option is omitted from the PROC GRADBOOST syntax, the use of the AUTOTUNE statement will still activate early stopping with STAGNATION=4 and the following log note would also be included.

```
NOTE: Automatic early stopping is activated with STAGNATION=4; set
EARLYSTOP=false to deactivate.
```

Table 2 lists the data sets used for the tuning efficiency studies that are presented in this section. These data sets range from tall and relatively narrow to short and very wide. They are listed in increasing order of number of values in the data set. The width of the data set (the number of attributes) has a significant impact on training, and hence on tuning expense. A more detailed description of each data set is provided in Appendix A.

	# Observations	# Attributes	# Classes	# Values
Covertype	581,012	54	7	31,955,660
MNIST	60,000	718	10	43,140,000
Bank	1,060,038	54	2	57,242,052
CIFAR-10	50,000	3072	10	153,650,000

Table 2. Benchmark Data Sets Summary

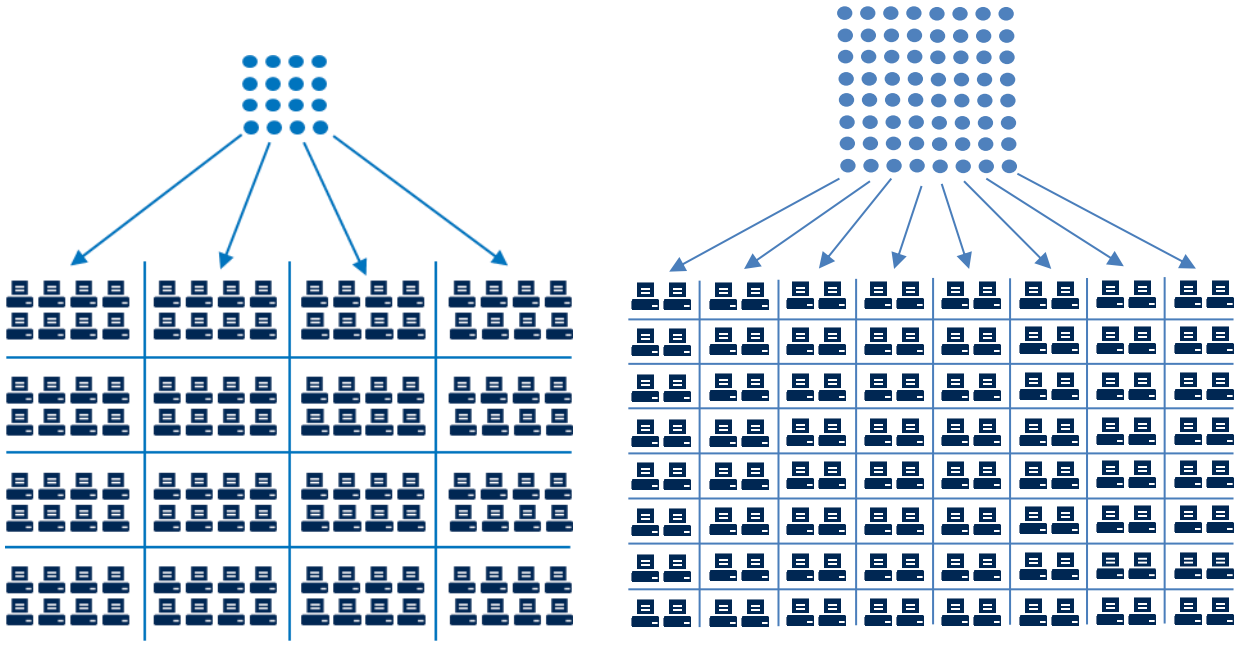
RESOURCE ALLOCATION AND NUMBER OF PARALLEL MODELS

As illustrated in Figure 4, it is clear that distributed training is not only unnecessary for small data sets, it is inefficient because of the cost of communication between worker nodes. The best allocation of resources for hyperparameter tuning with relatively small data sets would be to use a single worker node for each hyperparameter configuration, allowing all worker nodes to be used for parallel evaluation of different model configurations during the tuning process. With larger data sets, the cost of each individual model training must be weighed against the cost of the tuning process overall, while considering the maximum potential number of parallel evaluations (based on the search method), the complexity of the models being investigated, and the time budgeted. Thus, the “best” resource allocation is affected by many factors, including the following:

- size of the data set used for model training

- the search method and its configuration: population size for GA or Bayesian search methods, or sample size for random or LHS search methods
- number of workers available to the server

For example, as illustrated in Figure 6, if the population for each iteration contains 16 new configurations to be evaluated, a cluster of 128 workers would support 8 workers per model configuration, even though it might not be notably more efficient to train with 8 workers compared to 4 workers. Alternately, by using only 2 workers per model configuration, although each model training might take a little longer, the number of configurations to evaluate could be increased to 64, with all configurations still being trained in parallel. The choice depends on preference—reduced time with faster individual model training, or more candidate model configurations in the same amount of time.



(a) Faster population evaluation with more workers per model training

(b) More candidate model configurations potentially finding a better model faster

Figure 6. Resource Allocation for an Individual Population Evaluation

The default population size for autotuning in SAS Visual Data Mining and Machine Learning is set conservatively at 10 model configurations per iteration, for a default of five iterations—a maximum of 50 model configurations. The number of workers to use for each model training and the number of parallel evaluations are controlled by the NSUBSESSIONWORKERS and NPARALLEL options, respectively. Default values of these options are determined based on the data set size and the cluster size. First, the number of workers to use for each model training is determined. If the NSUBSESSIONWORKERS option is not specified, the number of workers is determined based on the size of the data set:

$$\text{NSUBSESSIONWORKERS} = 1 + \frac{nDataRows * nDataColumns}{50\ million}$$

The default number of workers in each subsession (each used for one model training) is set at one node per 50 million values, aggressively favoring allocation of resources to parallel tuning.

After the number of workers to use for each model training has been determined, the number of model trainings that can execute in parallel can be calculated. First, the number of *potential* parallel evaluations is determined based on the search method: one less than the population size for the GA search method (accounting for the best point carried over from the previous iteration), the population size for the Bayesian search method, or the sampling size for the random or LHS search methods. The *actual* number of parallel evaluations is then limited by the server configuration. In single-machine mode, if the number of *potential* parallel evaluations is greater than 4, it is limited to 4. This limit can be overridden up to a value of 32 by specifying the NPARALLEL option. In distributed mode, the upper limit for the number of parallel evaluations is calculated as *W/n*, where *W* is the number of workers connected to the server and *n* is the number of workers used for each model training. This limit can be overridden by up to a factor of 2 by specifying the NPARALLEL option, with a maximum value of 2*W/n*. This resource allocation process is summarized in Figure 7.

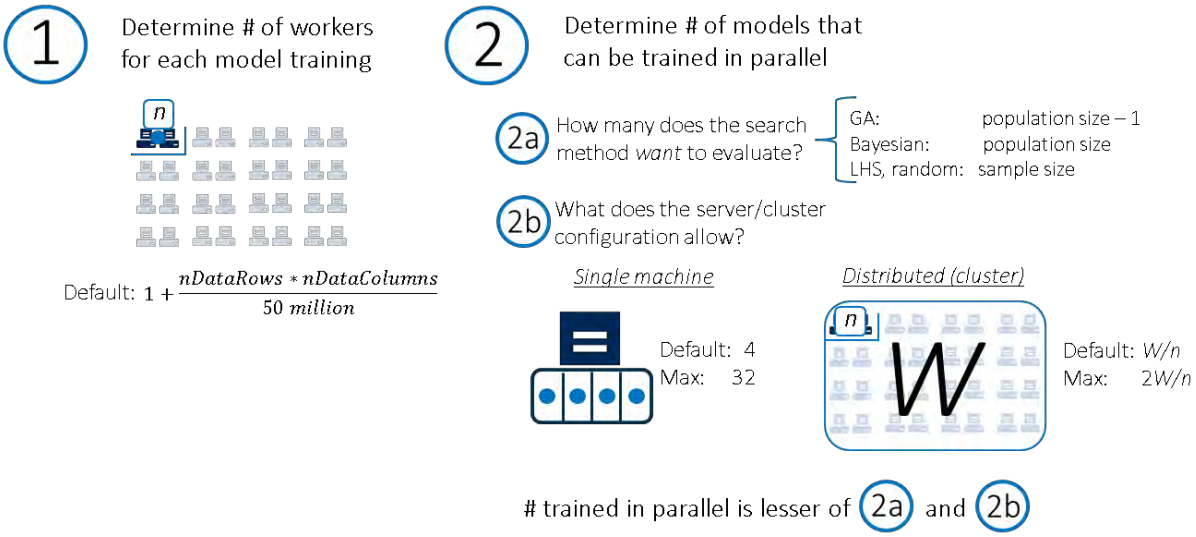


Figure 7. Process for Determining Worker Allocation (Training versus Tuning)

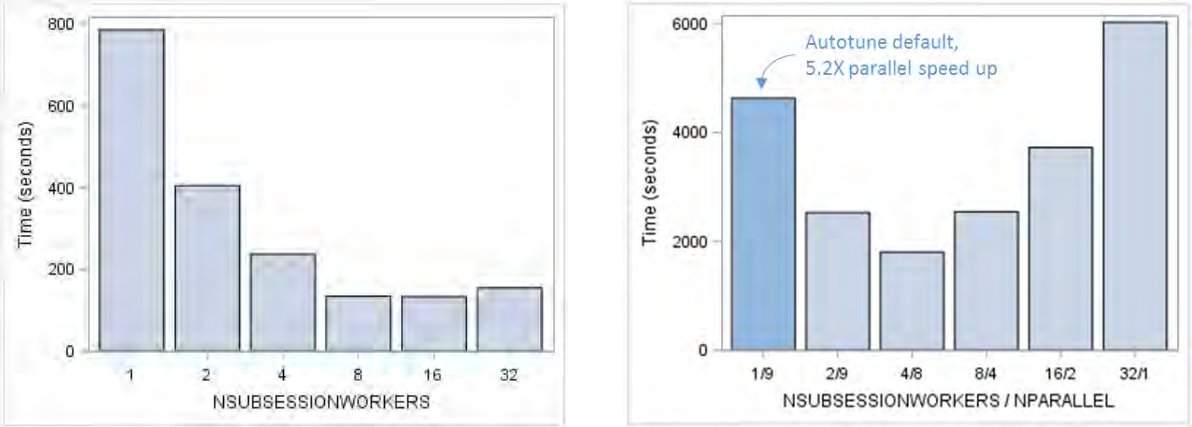
As an example, consider a data set that has 1.5 million observations and 50 columns, for a total of 75 million values. Based on the preceding equation for NSUBSESSIONWORKERS, two workers will be assigned to each subsession by default. With the default tuning search method, NPARALLEL will be set to 9 based on the default population size of 10; thus, a total of 18 worker nodes in the cluster will be required. If the cluster contains only 16 worker nodes, NPARALLEL will be reduced to 8 by default, or can be overridden to as many as 16 (overloading the workers). If 38 workers are available, either the population size can be increased to 20 (19 new models to train at each iteration) to make use of all the workers (with 2 workers per parallel subsession) or the number of workers per subsession could be increased to 4 for faster model training if the default maximum number of configurations is desired. What should be avoided is keeping population size at 10 when NPARALLEL is reduced to 8 (16-worker cluster). In this case, eight models will be submitted in parallel, and the remaining model will be submitted when one of the first eight models finishes and frees up the subsession workers, with the other seven subsessions being idle. With roughly equal training times (which is not usually the case), each iteration then requires two batches, or roughly the cost of two model trainings, rather than the cost of a single training (when all models in the population are evaluated in parallel). Population size must be carefully considered and adjusted manually when necessary.

Case Study Results

For each of the data sets listed in Table 2, numerous studies were run, using different allocations of compute resources for individual model training (NSUBSESSIONWORKERS) and parallel tuning (NPARALLEL). Results for each data set are shown in Figure 8 through Figure 11. On the left in each pair

of results plots, plot (a) shows the time for a single model training for a number of resource allocation configurations—a single model is trained using 1, 2, 4, 8, 16, or 32 worker nodes. On the right in each pair of results plots, plot (b) shows the time for the default autotuning process—population size of 10 with a maximum of five iterations—for the same number of worker nodes allocated to each model configuration that is trained. The plots also display the number of parallel models. With a cluster of 32 available worker nodes, all models in an iteration can be evaluated in parallel if the number of worker nodes for each model is limited to one or two workers. (Recall that with a population size of 10, nine new models are generated and trained at each iteration because the one best model is carried forward after each iteration.) With four workers for model training, eight models can be tuned in parallel, using all 32 workers. When eight or 16 workers are used per model, four or two models, respectively, can be tuned in parallel. If all 32 workers are used for training, the tuning process is sequential: one model is trained at a time.

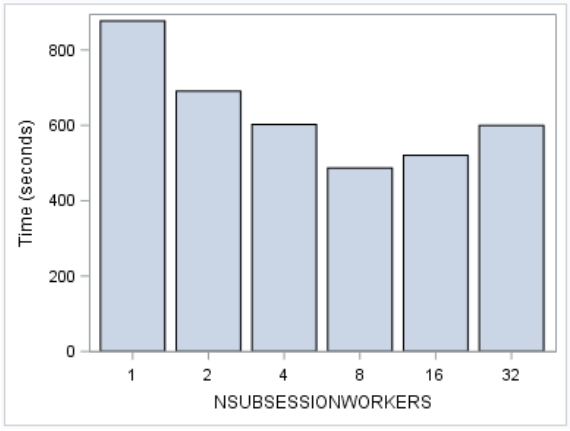
Figure 8 through Figure 11 clearly show not only that allocating more workers for training does not necessarily continue to increase the training efficiency, but also that the most efficient number of workers for model training is not the most efficient configuration for model tuning. This difference is a result of the efficiency gains from parallel tuning, which requires worker nodes to be available for allocation to different model configurations. For each case study data set, the default autotuning resource allocation is also indicated. The default number of worker nodes used for each model configuration, based on data set sizes, is set to one worker for the CoverType and MNIST data sets, two workers for the Bank data set, and four workers for the CIFAR-10 data set. For all but the CIFAR-10 data set, it is clear that the default resource allocation is not the most efficient. The number of workers nodes for each configuration is chosen to allow more models to be trained in parallel. With the CoverType and MNIST data sets, it would be possible to increase the population size to train up to 32 models in parallel in each iteration. For the Bank data set, 16 models could be trained in parallel in each iteration. Alternately, if only the default number of total configurations is desired, the number of workers used for each model can be increased to reduce the tuning time, as shown in the plots. The parallel speed up with the default tuning process is also reported to indicate that even if the default resource allocation is not the most efficient configuration, it is more efficient than sequential tuning.



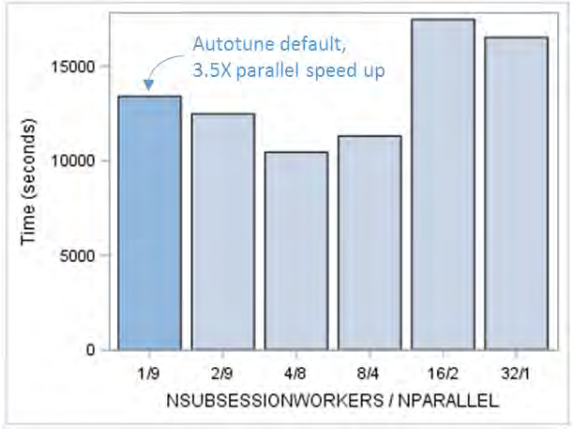
(a) Single training time by nSubsessionWorkers

(b) Default tuning time by nSubsessionWorkers / nParallel

Figure 8. Resource Allocation Comparisons, CoverType Data Set

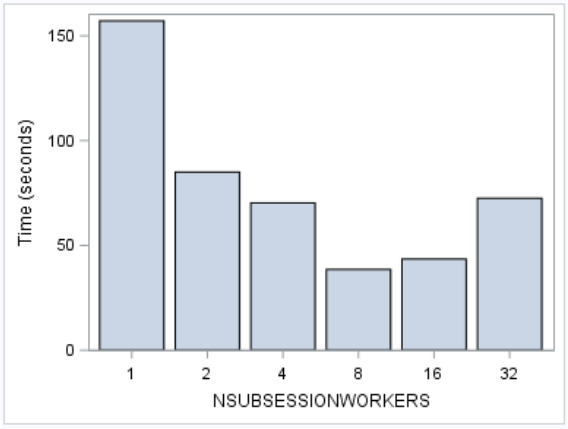


(a) Single training time by nSubsessionWorkers

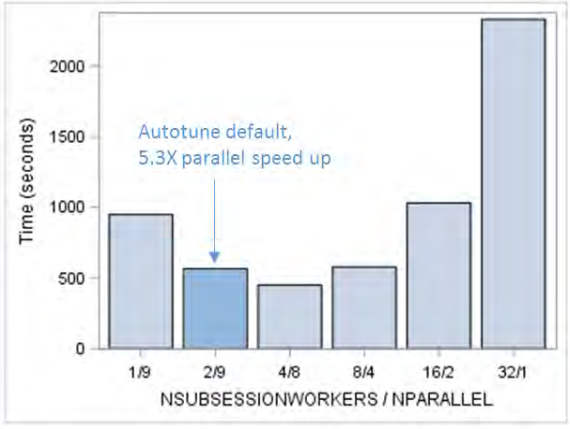


(b) Default tuning time by nSubsessionWorkers / nParallel

Figure 9. Resource Allocation Comparisons, MNIST Data Set

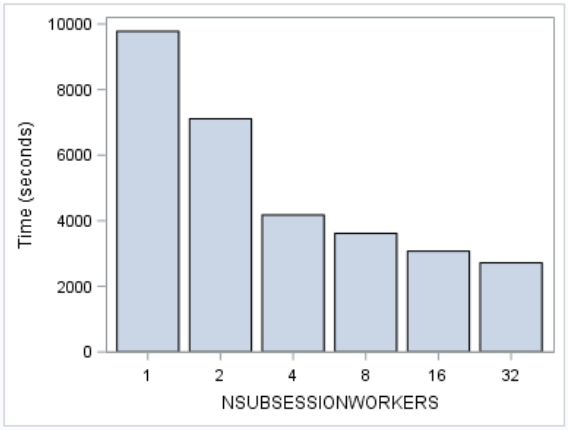


(a) Single training time by nSubsessionWorkers

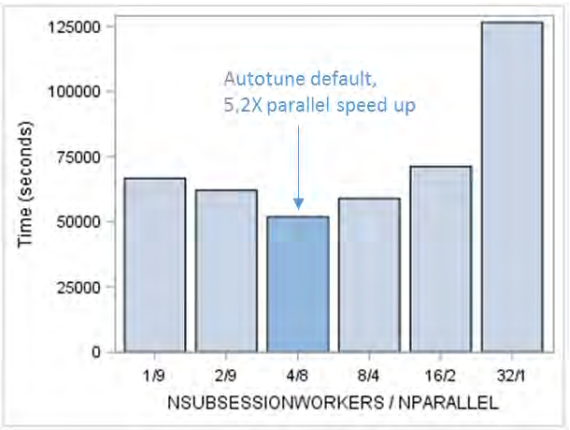


(b) Default tuning time by nSubsessionWorkers / nParallel

Figure 10. Resource Allocation Comparisons, Bank Data Set



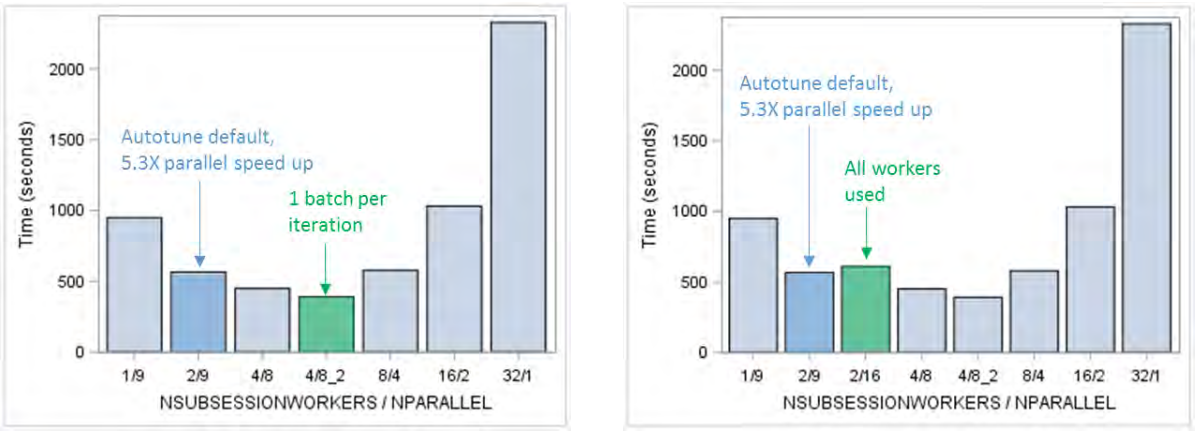
(a) Single training time by nSubsessionWorkers



(b) Default tuning time by nSubsessionWorkers / nParallel

Figure 11. Resource Allocation Comparisons, CIFAR-10 Data Set

Figure 12 illustrates two alternate autotuning configurations that use the Bank data set. Using four workers per model configuration results in the fastest training time for this data set. However, with 32 workers available, only eight models can be tuned in parallel if four workers are used for each model. With the default population size of 10, leading to nine new models generated in each iteration (one carried forward from the previous iteration), there will be two batches for each iteration: eight in the first batch, and then the ninth will be evaluated as soon as one of the subsessions is available. In this case, it is more effective to set the population size to nine, resulting in eight new configurations at each iteration, all evaluated in a single batch. The number of iterations can then be increased and roughly the same number of configurations will be evaluated in less total time—six iterations with a single batch each (six submission batches in total) versus five iterations with two batches each (10 submission batches in total). These results are shown in Figure 12(a). In Figure 12(b), the default autotuning configuration (which uses two workers per model configuration) is adjusted to use all of the available 32 worker nodes—running 16 parallel configurations instead of 9 at each iteration. Because all configurations are run in parallel in each iteration and the number of iterations is not changed, the total tuning time is roughly the same. The time is slightly longer with 16 parallel configurations because the time for each iteration is determined by the longest running configuration. With many more configurations evaluated (81 compared to 46 by default), more complex models configurations are generated, leading to slightly longer evaluation time, but with the benefit of possibly finding a better model because more candidate configurations were evaluated.



(a) nSubsessionWorkers=4, nParallel=8, popSize=9, maxIters=6 (49 maximum configurations)

(b) nSubsessionWorkers=2, nParallel=16, popSize=17 (81 maximum configurations in five iterations)

Figure 12. Adjusted Population Size, Bank Data Set

EARLY STOPPING

Some of the tuning actions in the **autotune** action set execute training actions that iterate internally to fit a model, and the maximum number of the internal training iterations is often quite high by default. A high number of training iterations can lead to training times that are longer than necessary and can also lead to overfitting. When model improvement (based on validation error) has stagnated, or has ceased to make more than very minimal improvement in multiple successive iterations as illustrated in Figure 13, it is beneficial to terminate the training at that point. This is referred to as early stopping.

By default, the **tuneGradientBoostTree** action, called by PROC GRADBOOST when the AUTOTUNE statement is included, activates early stopping for more efficient tuning. With gradient boosting, early stopping terminates the training action if no improvement in model error is achieved within the last *n* iterations (*n* is specified in the *stagnation* parameter, which is set to 4 when autotuning). As a result, the actual final number of trees in the reported top model might be less than the best value that the autotuning action selects.

The **tuneNeuralNet** action, called by PROC NNET when the AUTOTUNE statement is included, also activates early stopping for more efficient tuning, but only if the number of internal neural network training iterations is 25 or greater. The *stagnation* parameter here specifies the number of consecutive validations with increasing error rates that are allowed before early termination of the model training optimization process, and the *frequency* parameter specifies how frequently (in epochs) validation occurs during model training. For tuning neural networks, the *stagnation* parameter is set to 4 and the *frequency* parameter is set to 1. An example model training iteration history plot with and without early stopping is shown in Figure 13; clearly most of the improvement is obtained by the early stopping point, which occurs after less than half the number of iterations.

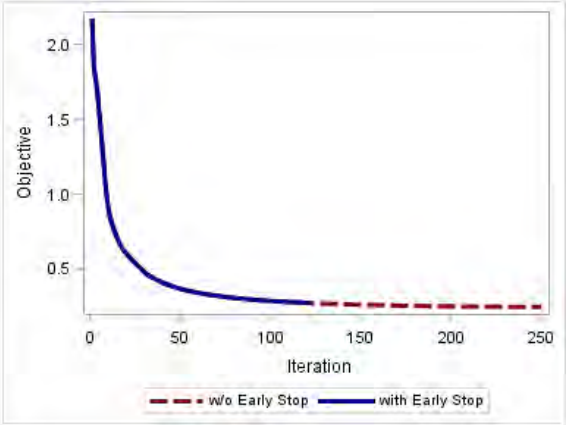


Figure 13. Iteration History Example: Early Stopping versus No Early Stopping

Early stopping can be disabled (allowing all models to train to completion) by specifying a value of False for the *earlyStop* parameter in the **tuneGradientBoostTree** and **tuneNeuralNet** actions or by specifying STAGNATION=0 in PROC GRADBOOST or PROC NNET when the AUTOTUNE statement is included. However, keeping early stopping enabled can often significantly reduce the total tuning time with little effect on the final model accuracy. Final models can be retrained with early stopping disabled to compare accuracy values. Figure 14 shows the reduction of default tuning time and a comparison of final accuracy for tuning a gradient boosting model for a set of benchmark test problems.[1] An average of 40% reduction in tuning time is observed, and the error of the final best model is similar or less with early stopping. Figure 15 shows similar results for tuning a neural network that is trained with 50 iterations of stochastic gradient descent. The average reduction in tuning time is more than 30%. However, in some cases the final model error is higher with early stopping than when the full 50 iterations are run. The model training process can appear stagnated over four epochs, but improvements can occur in later iterations. This is the trade-off and challenge with early stopping. For autotuning, early stopping can first be used to explore more models and refine the search space based on the best models, and then relaxed or disabled to further explore the space around good candidates that were identified. Also, the early stopping parameters can be adjusted, both the STAGNATION value and the FREQUENCY value for neural networks. If the validation checking for stagnation is performed every other epoch (FREQUENCY=2) instead of every epoch, the time savings is reduced to 25%, but the final model error values are closer to those seen without early stopping; these results are shown in Figure 16.

[1] Data sets from http://mldata.org/repository/tags/data/IDA_Benchmark_Repository/, made available under the Public Domain Dedication and License v1.0, whose full text can be found at http://www.opendatacommons.org/licenses/pddl/1.0/ .

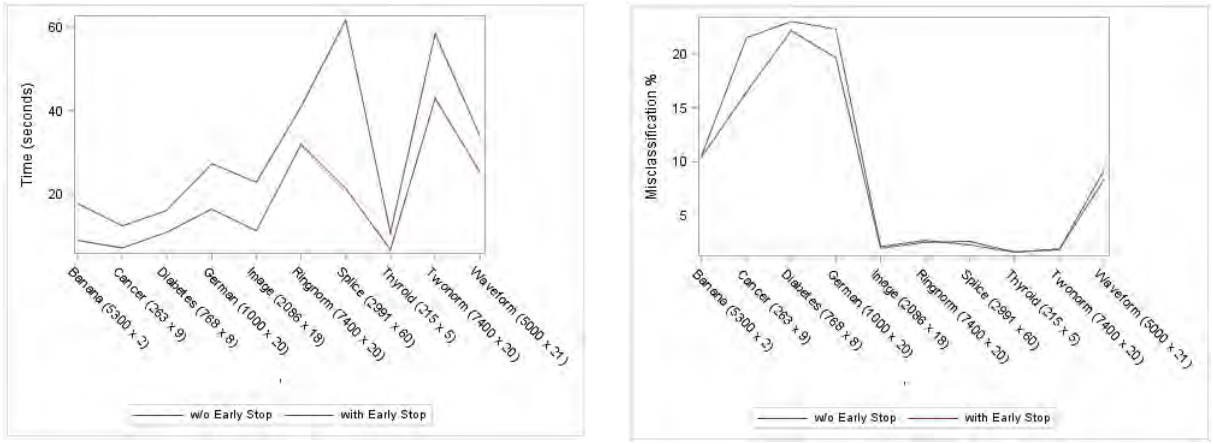


Figure 14. Early Stopping with the tuneGradientBoostTree Action

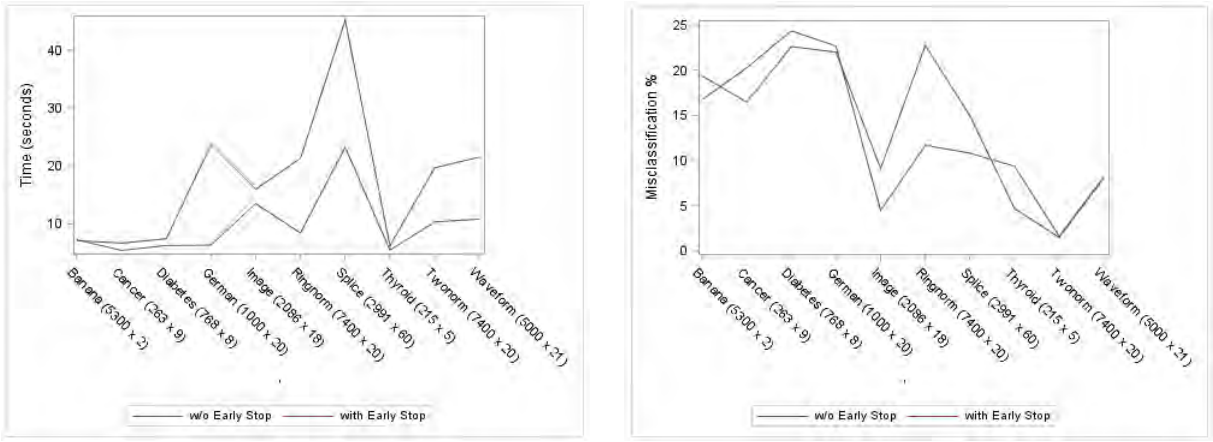


Figure 15. Early Stopping with the tuneNeuralNet Action, Using Stochastic Gradient Descent, 50 Iterations

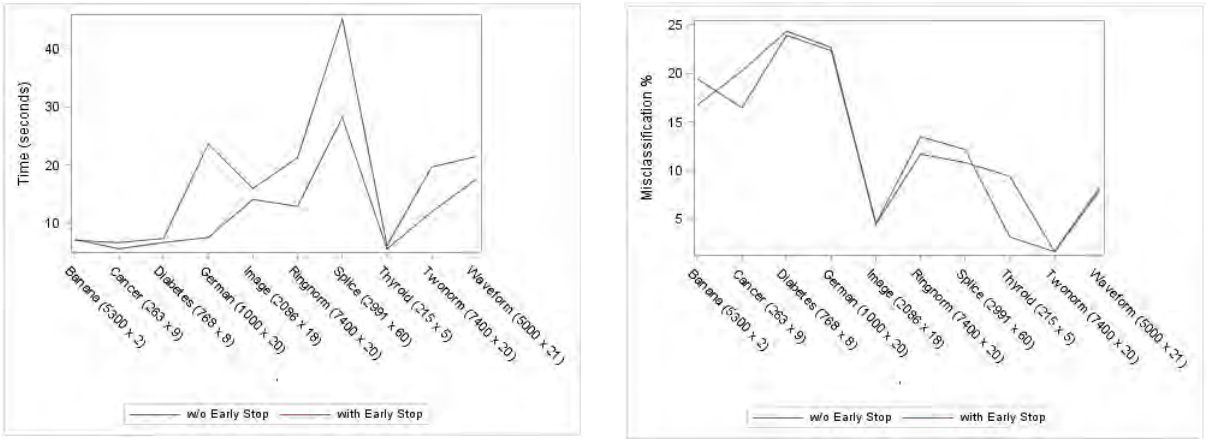


Figure 16. Early Stopping with the tuneNeuralNet Action, Using Stochastic Gradient Descent, 50 Iterations, frequency=2

Case Study Results

Early stopping results for default tuning of gradient boosting models for the four case study data sets are shown in Figure 17, where tuning times with and without early stopping are compared for each data set. Interestingly, you can see different behavior between long narrow data sets and short wide data sets. The early stopping process requires scoring during the model training iterations to determine whether the model error has stagnated. The more observations there are to score, the more the training time increases; models that do not stagnate will take longer to train because of the additional expense of scoring. For models that do stagnate, time can be saved. For the Bank and CoverType data sets, which contain many more observations and fewer columns than the MNIST and CIFAR-10 data sets, early stopping produces no savings. CoverType tuning takes slightly longer with early stopping because of the added cost of validation during tuning. For the MNIST and CIFAR-10 data sets, the validation set is much smaller—10,000 observations compared to more than 318,000 for the Bank data set and more than 174,000 for the CoverType data set. The training cost is also much higher because the MNIST and CIFAR-10 data sets are much wider. As a result, the savings from early stopping outweighs the added cost of validation during tuning. Early stopping for the MNIST data set saves more than an hour of tuning time, a 30% savings. For the CIFAR-10 data, early stopping saves nearly three hours of tuning time, an 18% reduction. Note that this is the default tuning process, which is only 50 maximum configurations.

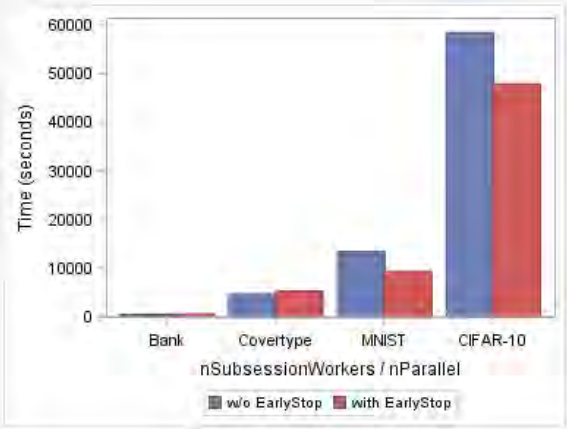


Figure 17. Early Stopping Effect

SUBSAMPLING TRAINING DATA

The expense of model training increases with data set size. In the past, subsampling was commonly used for model training when the data set size was too large. Today, distributed data and distributed training algorithms allow model training to scale to “big data” levels without subsampling. However, as discussed, if more worker nodes are allocated to model training, then fewer nodes are available for parallel tuning of different hyperparameter configurations. To reduce tuning time or increase the number of models that are trained in parallel during tuning within a time budget, autotuning can employ subsampling of the training data. If subsampling of training data is representative of the full data set, a larger number of hyperparameter configurations can be explored more efficiently without diminishing the accuracy of the resulting models. After hyperparameter configuration options have been narrowed, the full training data set can be used for final tuning or to confirm and select among alternate candidate configurations (or both).

The **sampling** action set is used by the autotuning process to create the training and validation partitions if they are not supplied: the **stratified** action is used for nominal type targets (if all target levels can be included in both the training and validation partitions), and the **srs** action is used for a target of interval type and for cases in which stratified sampling is not possible. By default, a validation partition of 30% is used and the remaining 70% is used for model training. Both can be adjusted. For large data tables, tuning efficiency can be increased by subsampling the remaining data for training. For example, 30% of the data can be used for model training and 30% for validation, leaving 40% unused. The training partition size can be specified using the PARTITION statement (specifying both validate and test fractions) in the procedures that include the AUTOTUNE statement, or by using the *trainPartitionFraction* parameter or its alias *trainFraction* with the **autotune** actions. Stratified sampling ensures that the model training and validation partitions are representative of the full data table when possible.

Case Study Results

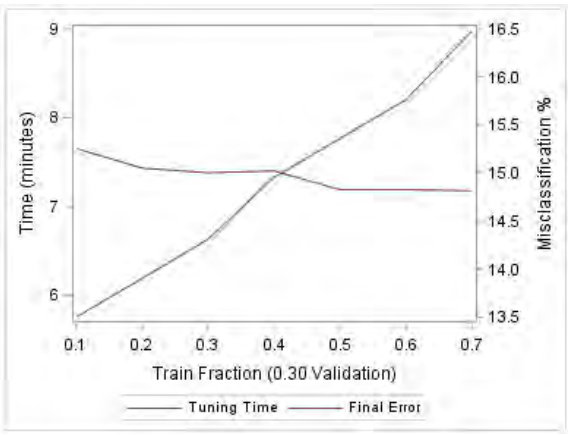
When the data set is subsampled for more efficient model training, the potential trade-off is reduced accuracy of final best tuned models. This trade-off is illustrated for the four case study data sets in Figure 18. In all cases, the default validation fraction of 30% is used and the training fraction is sampled from 10% to 70%. In all cases except for the CIFAR-10 data set, the axis range for final model accuracy in the plots is 3% so that they can be directly compared; the actual change in accuracy as the training fraction is reduced is less than 3% in these three cases.

In Figure 18(a) the Bank data misclassification error is seen to increase by only 0.4% when the training fraction sample size is decreased from 70% to 10%. However, the tuning time is reduced by more than 35%. When training is performed with 10% of the data, two additional tuning iterations could be added to evaluate 18 more configurations in roughly the same time as when training with 70% of the data, or the population size of each iteration could be increased by three evaluations, from 10 to 13.

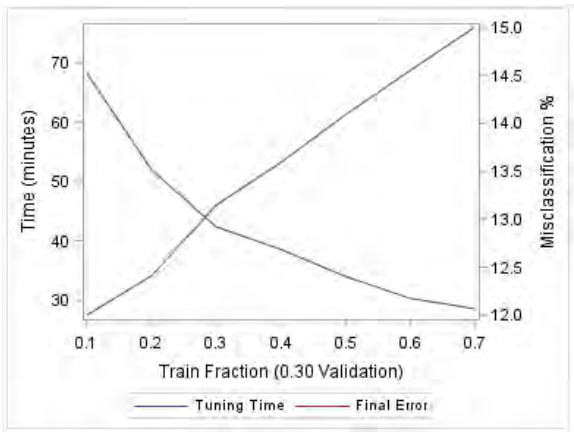
The subsampling results for the CoverType data in Figure 18(b) show greater change in the final model error, with an increase of roughly 2.5% when the training fraction is reduced from 70% to 10%. However, note that the default hyperparameter values result in a model with 19% error for this data set; all training sample sizes lead to reduced error with tuning. Also, at a 40% training fraction, a 30% reduction in tuning time is observed with only 0.6% increase in model error. The rate of error increase changes more significantly when 30% or less of the data are used for training.

For the MNIST data set, the final model error increases more rapidly when the training fraction is less than 40%, as shown in Figure 18(c); 2.5% error with 70% training fraction is increased to 4% when only 10% of the data is used for model training. At a 40% training fraction, again only a 0.6% increase in model error is observed. A 14% reduction in tuning time is observed with a 40% training fraction, compared to a 70% training fraction. In this case the time savings is less than it is for the previous two data sets because the data set is very wide, which affects the training time significantly; reducing the number of observations for training has less impact, but savings are still observed.

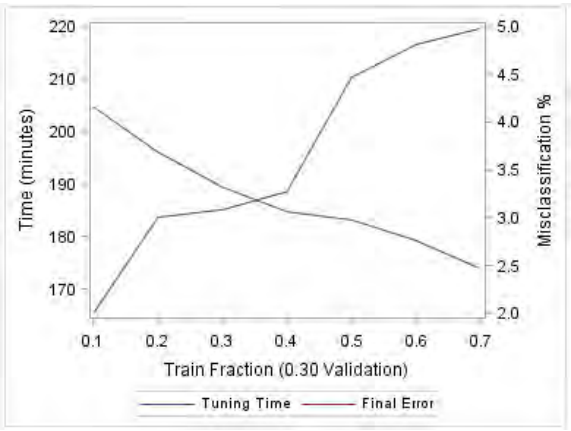
Both the time change and final model accuracy change are most significant for the CIFAR-10 data set, as shown in Figure 18(d). Here the tuning time is reduced by 23% when using a 10% training fraction compared to 70%, but at a cost of more than 9% in misclassification error. It is known that gradient boosting models are not the best choice for this data set—the misclassification errors are quite high. The increase in error is again slight at first, with the training fraction reduced to 60% or 50%, but increases quickly after that. Also, the time decrease with decreasing fraction is not as smooth as it is for the other data sets. For this complex data set, the default hybrid optimization process, incorporating a genetic algorithm, varies more when the data are changed, especially because solutions are not as good.



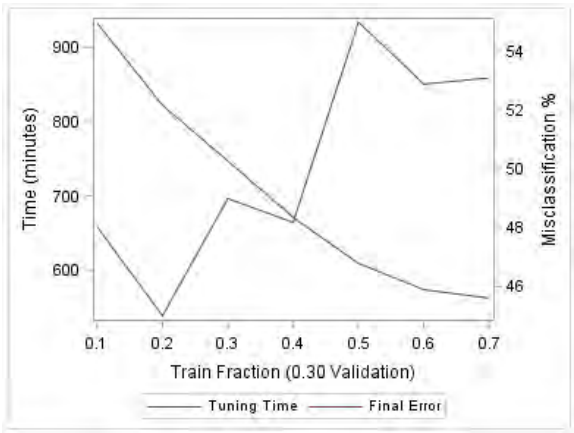
(a) Bank data



(b) CoverType data



(c) MNIST data



(d) CIFAR-10 data

Figure 18. Subsampling Trade-Off

CONCLUSIONS AND RECOMMENDATIONS

It is clear that the optimal number of worker nodes for training a model is not the optimal number when hyperparameters are tuned. Figure 8 through Figure 11 confirm that allocating resources to parallel training of different models reduces the tuning time more than does allocating resources to speed up each model training (that is, using the optimal number of nodes for each model training). Even if a cluster is sufficiently large to support the optimal number of worker nodes for each model configuration and allows all models in an iteration to be evaluated in parallel for the default autotuning process, it might be more effective to increase the number of models that are trained in parallel rather than decreasing the training time, providing a better chance at finding a better model. The options for setting the number of workers for each model training (NSUBSESSIONWORKERS), the number of models trained in each iteration during tuning (POPSIZE), and the number of models trained in parallel (NPARALLEL) can all be used in unison to optimize resource usage and control the hyperparameter tuning expense.

Early stopping can reduce tuning expense, but it can also increase individual training time because it leads to extra validation during the training process. Subsampling can reduce training time, but it can also increase model error; a trade-off is necessary to determine when efficiency is most appropriate (perhaps when exploring a large number of models during tuning) and when accuracy is critical (perhaps after narrowing the choices).

The combination of all these options within the autotuning implementation on SAS Viya can help manage the expense of hyperparameter tuning, allowing more configurations to be evaluated in a specific time budget. Effective settings of these options depend on the specific scenario and the goal of your study. Some best-practice recommendations are offered in Table 3.

Scenario	Suggested Best Practice
RESOURCE ALLOCATION	
If cluster size < population size (GA, Bayesian) or sample size (LHS, random)	• Set population or sample size as a multiple of $\frac{\#\ available\ worker\ nodes}{\#\ worker\ nodes\ per\ model\ (subsession)}$ (for GA search method, increase population size by 1 to account for the best configuration carryover) • This ensures even batches of candidate model configurations, thus no loss of efficiency with a partial batch • For GA search method, population size less than the default (10) is not recommended; for example, if cluster size is 8 worker nodes, increase population size to 17 (8*2+1) and set maximum iterations or maximum evaluations as desired to limit tuning time
If cluster size > population size (GA, Bayesian) or sample size (LHS, random) and data size is small or tuning time budget is flexible	• If 1 worker is used for each model (subsession), increase population or sample size to cluster size (increase by 1 for GA search method); this increases the total number of model configurations to be evaluated • If multiple workers are assigned to each model (subsession), increase population or sample size to $\frac{\#\ available\ worker\ nodes}{\#\ worker\ nodes\ per\ model\ (subsession)}$ (add 1 for GA search method)
If cluster size > population size (GA, Bayesian) or sample size (LHS, random) and data size is large or tuning time budget is limited (or both)	• Increase NSUBSESSIONWORKERS to $\frac{\#\ available\ worker\ nodes}{population\ size\ or\ sample\ size}$ (POPSIZE–1 for GA search method); this will increase the efficiency of training each model during tuning for medium to large data sets • Limit NSUBSESSIONWORKERS to 8; in most cases, increasing further will increase training time
EARLY STOPPING	
For initial hyperparameter tuning exploration	• Early stopping is activated by default when gradient boosting and neural network models are tuned in order to terminate stagnated model training
For refined hyperparameter tuning with narrowed ranges or for confirming final models (or both)	• Deactivate early stopping or reduce checking frequency (for neural networks, increase the *frequency* parameter value, which specifies the number of iterations between stagnation checks)

SUBSAMPLING	
If data set is large and is fairly balanced in target	• Subsampling down to a 40% training fraction is a good trade-off; in most cases, this reduces tuning time by 15–30%, with marginal loss in accuracy • Use all data for evaluating final models or for refined tuning
If data set is small	• Cross validation is preferred over a single validation partition; subsampling of the data is not necessary
If data set is very unbalanced in target or if errors are high	• Subsampling for the training fraction is not recommended

Table 3. Best Practice Recommendations for Managing Hyperparameter Tuning Expense

APPENDIX A: BENCHMARK DATA SET DESCRIPTIONS

CoverType Data

The CoverType data set is obtained from the UCI Machine Learning Repository (Lichman 2013). The data set, gathered from four wilderness areas in the Roosevelt National Forest, is used to predict forest cover type based on cartographic variables, which include elevation, aspect, slope, horizontal and vertical distance to hydrology, horizontal distance to roadways and fire points, hillshade, specific wilderness area, and soil type. A total of 54 attributes are used to predict seven tree types, making the modeling problem one of multiclass classification. The data set includes 581,012 observations, leading to nearly 32 million values in the complete data set.

MNIST Digits Data

The MNIST (Mixed National Institute of Standards and Technologies) database of handwritten digits (Lecun, Cortes, and Burges 2016) contains digitized representations of handwritten digits 0–9, in the form of a 28 × 28 image for a total of 784 pixels. Each digit image is an observation (row) in the data set, with a column for each pixel containing a grayscale value for that pixel. After removal of pixels that are blank for all observations, the data set contains 718 attribute columns. The database includes 60,000 observations for training (over 43 million values), and a test set of 10,000 observations. Like many studies that use this data set, this example uses the test set for model validation during tuning.

Bank Data

The Bank data set is a simulated data set that consists of anonymized and transformed observations taken from a large financial services firm's accounts. Accounts in the data represent consumers of home equity lines of credit, automobile loans, and other types of short- to medium-term credit instruments. The data set includes a binary target that represents whether the account contracted at least one new product in the previous campaign season, and 54 attributes that describe the customer's propensity to buy products, RFM (recency, frequency, and monetary value) of previous transactions, and characteristics related to profitability and creditworthiness. With 1,060,038 observations, the data set contains over 57 million values. This data set can be downloaded from GitHub at https://github.com/sassoftware/sas-viya-machine-learning (in the data folder).

CIFAR-10 Image Data

The CIFAR-10 data set (Krizhevsky 2009) contains 10 classes of 32 x 32 color images. The 10 classes are: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship,* and *truck*. The data set includes 6,000 images per class, with 50,000 images used for training and 10,000 images used as a test set (used for model validation during tuning). The digitized images are represented by a set of RGB (red, green, blue) values for each pixel, resulting in 3,072 attribute columns (32 x 32 x 3). The data set thus contains over 153 million values.

REFERENCES

Bergstra, J., Bardenet, R., Bengio, Y., and Kégl, B. (2011). "Algorithms for Hyper-parameter Optimization." In *Proceedings of NIPS*, 2546–2554.

Bergstra, J., and Bengio, Y. (2012). "Random Search for Hyper-parameter Optimization." *Journal of Machine Learning Research* 13:281–305.

Eggensperger, K., Feurer, M., Hutter, F., Bergstra, J., Snoek, J., Hoos, H., and Leyton-Brown, K. (2013). "Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters." In *NIPS Workshop on Bayesian Optimization in Theory and Practice (BayesOpt'13)*.

Koch, P., Wujek, B., Golovidov, O., and Gardner, S. (2017). "Automated Hyperparameter Tuning for Effective Machine Learning." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings17/SAS0514-2017.pdf.

Krizhevsky, A. (2009). "Learning Multiple Layers of Features from Tiny Images." Technical Report, Univeristy of Toronto.

LeCun, Y., Cortes, C., and Burges, C. J. C. (2016). "The MNIST Database of Handwritten Digits." Accessed April 8, 2016. http://yann.lecun.com/exdb/mnist/.

Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

Wexler, J., Haller, S., and Myneni, R. 2017. "An Overview of SAS Visual Data Mining and Machine Learning on SAS Viya." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings17/SAS1492-2017.pdf.

Wujek, B., Hall, P., and Güneş, F. (2016). "Best Practices in Machine Learning Applications." In *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. Available at https://support.sas.com/resources/papers/proceedings16/SAS2360-2016.pdf.

ACKNOWLEDGMENTS

The Forest Covertype data set is copyrighted 1998 by Jock A. Blackard and Colorado State University. The authors would like to thank Anne Baxter for her contributions to this paper.

RECOMMENDED READING

- *Getting Started with SAS Visual Data Mining and Machine Learning 8.2*
- *SAS Visual Data Mining and Machine Learning 8.2: Procedures*
- *SAS Visual Data Mining and Machine Learning 8.2: Programming Guide*
- *SAS Visual Statistics 8.2: Procedures*
- *SAS Visual Statistics 8.2: Programming Guide*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Patrick Koch	Brett Wujek	Oleg Golovidov
SAS Institute Inc.	SAS Institute Inc.	SAS Institute Inc.
patrick.koch@sas.com	brett.wujek@sas.com	oleg.golovidov@sas.com