Seneca: Fast and Low Cost Hyperparameter Search for Machine Learning Models

Michael Zhang, Chandra Krintz, Rich Wolski
Dept. of Computer Science
University of California, Santa Barbara
{lebo, ckrintz, rich, mock}@cs.ucsb.edu

Markus Mock
Dept. of Mathematics and Statistics
Univ. of Applied Sciences, Landshut Germany

***Abstract*— The goal of our work is to simplify and expedite the construction and evaluation of machine learning models using autoscaled cloud computing systems and services. To enable this, we develop an open source system, Seneca, that leverages the serverless programming model and its implementation in Amazon Web Services (AWS) Lambda. Seneca takes a machine learning application, dataset, and a list of possible hyperparameter options. It automatically constructs an AWS Lambda handler that ingresses and splits the dataset into training and testing subsets, and constructs, tests, and evaluates (i.e. scores) a machine learning model for a given set of hyperparameter values. Seneca concurrently invokes functions for all combinations of the hyperparameters specified. It then returns the configuration (or model) that results in the best score to the user.**

We present the design and implementation of Seneca and introduce an extension to the system that automatically optimizes memory use by the functions. Our empirical evaluation using multiple predictive machine learning applications for regression and classification shows that Seneca is able to quickly identify the best performing model for the programs and datasets that we consider. Moreover, its memory optimization reduces the cost of using Seneca by 10–35% for the applications studied.

***Keywords*-Serverless computing; Model optimization; Hyperparameter tuning**

I. Introduction

The scale and elasticity of cloud computing systems have fueled remarkable innovation and unprecedented commercial investment. Cloud users "rent" virtualized resources (while sharing the underlying physical resources) on a pay-per-use basis in exchange for availability guarantees specified via service level agreements (SLAs). Uniquely, cloud systems can be configured to add and remove (i.e. auto-scale) resources and services automatically, based on the dynamic resource requirements and service needs of executing applications.

To date however, clouds are used more for enterprise services (object stores, databases, application servers, etc.) than for elastic applications. The reason is that it is challenging to configure complex distributed systems for application use, and to leverage the auto-scaling that clouds offer. To address this challenge, cloud providers have started to offer programming and execution environments that obviate the need for server configuration, under the *serverless* moniker [1], [2], [3]. Serverless platforms automatically configure, manage, and scale applications to significantly simplify cloud use.

Using the serverless model, application developers upload arbitrary computations in high level languages as stateless functions to cloud-hosted, serverless platforms, where functions are triggered automatically by the cloud in response to updates from other cloud services (e.g. storage, queues, notification services, and API gateways, among others). Serverless functions must execute under a time bound (e.g. 15 minutes) and an allocated memory size (e.g. 3GB) or else the platform will terminate the function. They communicate, persist, and access data only through their inputs or via shared storage services. As a result, serverless applications are inherently elastic and can implement highly concurrent and parallel tasks. In public clouds, users pay a small fee for the resources their functions use during execution, resulting in very low cost cloud use. Although now available from all public cloud providers and as open source for private cloud systems, Amazon Web Services (AWS) Lambda [4] was the first and is the most widely used serverless public cloud platform.

In this work, we investigate the efficacy of using AWS Lambda for tuning machine learning applications in parallel. To date, Lambda is not widely used for training and evaluating machine learning models because of a concern that doing so will result in high overhead (i.e. be costly) because of the stateless nature of serverless functions [2]. At the same time, identifying the "best" configuration for advanced machine learning models is challenging given the large number of configuration options (i.e. hyperparameters) typical for models today. Hyperparameters govern the learning process of machine learning applications. Given that parameter sweeps are embarrassingly parallel, we believe that such tuning is a good fit for the serverless model. To investigate this potential, its overhead, and to simplify the use of Lambda for training, testing, and evaluation of machine learning models, we design and develop a new system and toolset called *Seneca*.

Seneca implements, packages, and deploys machine learning applications as stateless functions to AWS Lambda. It then orchestrates exhaustive evaluation of specified hyperparameter settings to identify the best performing model (for a given dataset) by comparing error and accuracy across models. We consider prediction accuracy (as opposed to explanatory power) as the scoring metric (mean squared prediction error for regression and accuracy percentage for classification). Users present Seneca with their application, a range of values

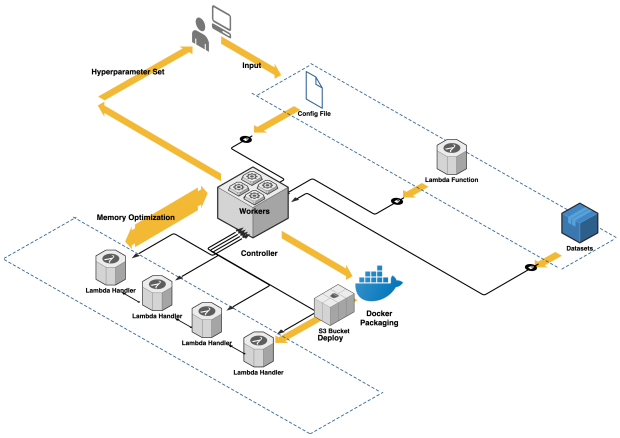

Fig. 1: The Seneca Architecture.

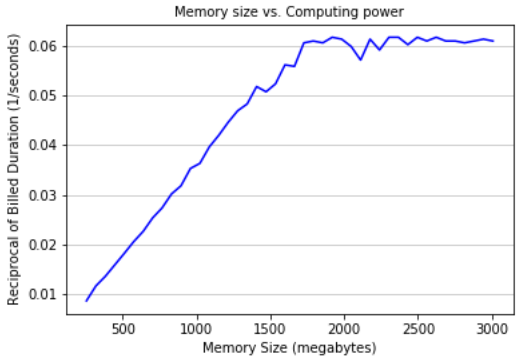

Fig. 2: The relationship between allocated memory and reciprocal of billed duration, which represents compute power for a compute-bound Lambda function.

for each hyperparameter (or the default can be used), and a representative dataset. Seneca produces, tests, and evaluates models for all combinations of hyperparameters and returns to the user the set of parameters (or the model itself) that produces the best cross-validation score. Users can employ this model for other datasets (with Seneca if desired) without retraining the model to amortize the cost of Seneca further.

We deploy Seneca on AWS Lambda and evaluate its tuning performance, cost, and memory use for five machine learning applications and datasets. We find that Seneca is fast, inexpensive, and effective for model construction and comparison. Seneca is also able to identify automatically the best memory configuration for each application, further lowering its cost by 10-35%. Relative to execution in AWS EC2, we observe a benefit/cost ratio (computed as speedup/(dollar cost) cost of 294 on the average in the experiments we consider. We intend to make Seneca, its applications, and their datasets publicly available when/if this paper is accepted for publication. We next overview our design and implementation of Seneca and then present our empirical methodology and results.

II. Seneca

To facilitate model search and selection using the serverless architecture, we have developed Seneca, a framework for tuning the hyperparameters of machine learning applications in AWS Lambda. The Seneca pipeline consists of packaging, deployment, function optimization, and hyperparameter tuning. Figure 1 shows the architecture of Seneca. In the the upper-right-front, we show the three inputs that Seneca expects from its users: (*A*) a hyperparameter configuration file, (*B*) a dataset URL, and (*C*) the lambda function of the machine learning application. The configuration file specifies a set of values for each hyperparameter that the application expects. Seneca creates the Cartesian product of all options in this configuration as the search space. The dataset URL refers to a valid dataset stored in the AWS Simple Storage Service (S3) [5].

Based on the specified machine learning application, Seneca automatically builds and deploys an AWS Lambda application by launching a Docker container that mirrors the AWS Lambda execution environment, checks and installs the machine learning application and any libraries it requires, compresses the application and uploads it to S3 (a work-around for the 10MB AWS Lambda function size restriction). Seneca constructs an AWS Lambda function from a template that, when executed, will download the dataset and split it into a training and testing set, and construct, test, and evaluate a model using the application and a set of hyperparameter values passed in by Seneca as arguments. Users can specify the train/test split ratio that should be used by Seneca; the default is 80%/20% for classification tasks. The function returns a testing score. Upon completion of this process, the container deploys the function to AWS Lambda using the AWS Command Line Interface [6] and the developers credentials.

A. Optimizing Memory Use

The cost of using AWS Lambda (i.e. `compute charge`) is the billed duration (execution time rounded up to the nearest 100ms) [7] multiplied by the allocated memory of each invoked function. One goal of our work is to optimize memory use of these applications in order to reduce cost, and to investigate the trade-offs of doing so.

Currently, allocated memory for a Lambda function can be set from 128MB to 3008MB in increments of 64MB. AWS documentation [8] states that Lambda allocates CPU to functions corresponding to allocated memory size, as is done for general purpose AWS EC2 instance types.

To evaluate the relationship between memory, CPU, and cost, we analyze a 3-D matrix multiplication serverless benchmark [9] using AWS Lambda. We configure different functions to use each of the 46 possible allocated memory options. Figure 2 shows the relationship between allocated memory (x-axis) and reciprocal of billed duration (y-axis). Figure 3 shows the relationship between memory size (x-axis) and compute charge (y-axis). We observe that for this benchmark, billed duration plateaus after 1600 MB, at which point compute charge increases. That is, we achieve no further execution time benefit (only cost increase) after this point.

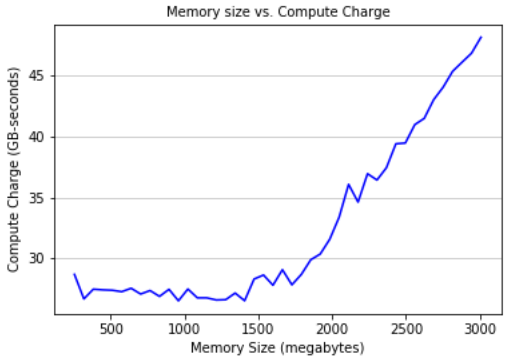

Fig. 3: The relationship between allocated memory and compute charge for a compute-bound Lambda function.

We use this relationship within Seneca to optimize its cost (`compute charge`) via an extension that enables it to automatically identify the appropriate setting for allocated memory for each application. However, instead of exhaustively testing all 46 possible memory configurations as we did for the matrix benchmark, which may be costly, Seneca employs the heuristic outlined in Algorithm 1.

The Seneca optimizer first configures and invokes the function using a user-defined payload. From this run, Seneca obtains the `maximum memory used` by the function as reported by AWS Cloudwatch, and uses it as the starting point in its search. Seneca then defines two double-ended queues (*deque*) of length N, to store `allocated memory` and `compute charge` data of different invocations. While the current allocated memory is less than or equal to 3008 MB, the optimizer reconfigures and invokes the function using the next increment for memory allocation. It calculates the compute charge for each invocation using current allocated memory and billed duration.

We employ two exit conditions. The first is when the compute charge monotonically increases across *deque*. The second is when the increase in slope is greater than a threshold. When the optimizer finds that both conditions hold, it pops the left-most value from *deque* and configures the function to use that value for allocated memory for all future invocations. If these two conditions can not be satisfied during search, the allocated memory will be configured as the memory size that results in minimal compute charge within the *deque*. After extensive experimentation, we find that $N = 5$ and a slope threshold of 1 work best, but these values are configurable. In addition, this optimization can be turned on or off via a command line argument to Seneca.

B. Tuning Process

To facilitate parallel function invocation, Seneca integrates Celery [10]. Celery is an asynchronous task queue that uses distributed message passing. Celery workers are processes that take tasks from the queue, execute the tasks with the arguments specified, and store the result that is returned in a database (we use Redis [11] in our prototype).

Algorithm 1: Seneca Optimizer Heuristic

Data: Typical payload
Result: Optimal allocated memory
Find memory used by payload as starting point;
Define deque for allocated memory & compute charge;
while *allocated memory* $\leq$ *3008 MB* **do**
  if *compute charge monotonically increases in deque & slope* $\geq$ *threshold* **then**
    popleft from deque;
    configure allocated memory as optimum;
    exit();
  else
    increase allocated memory by 64 MB;
    probe lambda function;
    append memory and compute charge to deque;
  end
end

Application	Description
Prophet	Time series decomposition and prediction
Multi-Regression	Multiple linear regression/prediction of time series
XGBoost	Regression and classification by gradient boosting
SVC	Classification based on support vector machine
Neural-Net	Classification by layered artificial neural network

TABLE I: Machine learning applications used to evaluate Seneca.

Based on the configuration file, Seneca creates and enqueues a list of payloads (function arguments) for each combination of hyperparameter values. The Seneca celery workers invoke the application's Lambda function by each payload for model construction. Upon function termination, the worker records a score for the hyperparameter configuration in the database. When the queue is drained and all workers have completed, Seneca extracts and reports the best score, configuration, and model from the database. Users can then use the model for inference given other datasets without retraining to amortize the time/cost of Seneca.

We assume that the dataset supplied to Seneca by the user is representative of datasets on which the resulting model will be used. As part of future work, we are considering using multiple datasets and a ranges of hyperparameter values to preclude the need for users to specify them and to consider a wider range of values.

III. Evaluation

In this section, we empirically evaluate Seneca in terms of machine learning (ML) model output quality, performance, and cost. We first overview the ML applications that we consider and our experimental methodology. We then present our results.

A. Benchmark Applications and Training/Testing Datasets

The ML applications that we use to evaluate Seneca are described in Table I. Prophet, Multi-Regression, and XGBoost are regression applications; XGBoost, SVC, and NN are classification applications (XGBoost implements both regression

Hyperparameter	Default	Tuning options
growth	linear	[linear, logistic]
changepoint prior scale	0.05	[0.05, 0.5]
holidays prior scale	10	[1, 5, 10]
seasonality prior scale	0.5	[0.1, 0.5]
fourier order	10	[5, 10, 15, 20]
seasonality mode	additive	[additive, multiplicative]
interval width	0.8	[0.5, 0.8]

TABLE II: Hyperparameters Seneca considers for **Prophet**.

Hyperparameter	Default	Tuning options
max depth	3	[3, 4]
learning rate	0.1	[0.1, 0.01]
N estimators	100	[100, 400]
objective	reg:linear	[reg:linear, rank:pairwise]
booster	gbtree	[gbtree, gblinear, dart]
min child weight	1	[0.1, 1]
scale positive weight	1	[1, 2]
base score	0.5	[0.5, 10]

TABLE III: Hyperparameters Seneca considers for **XGBoost**.

Hyperparameter	Default	Tuning options
C	1.0	[0.5, 1.0]
kernel	rbf	[rbf, linear, poly, sigmoid]
degree	3	[3, 4]
gamma	auto	[auto, scale]
coef0 init	0.0	[0.0, 1.0]
probability	False	[False, True]
tol	1e-3	[1e-3, 1e-4]
decision function shape	ovr	[ovo, ovr]

TABLE IV: Hyperparameters Seneca considers for **SVC**.

Hyperparameter	Default	Tuning options
activation	relu	[identity, tanh, relu]
solver	adam	[lbfgs, sgd, adam]
learning rate	constant	[constant, invscaling, adaptive]
learning rate init	0.001	[0.001, 0.0001]
power T	0.5	[0.1, 0.5]
tol	1-e4	[1e-4, 1-e5]
n iter no change	10	[10, 20]

TABLE V: Hyperparameters Seneca considers for **NN**.

and classification tasks). The regression applications compute the mean square error (MSE) as $\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y_i})$, where $\hat{Y_i}$ is the ground truth, Y_i is model prediction and n is the number of data points. The applications return the average MSE across cross validations. The classification applications compute and return a classification accuracy percentage, which is calculated as $\frac{1}{n}\sum_{i=1}^{n} 1(Y_i = \hat{Y_i})$, where Y_i is the prediction class, $\hat{Y_i}$ is the true class, n is the number of samples, and 1(x) is the indicator function.

Prophet [12] is an open source time series analysis library developed by Facebook. The input dataset we consider is a time series of view counts of Peyton Manning's Wikipedia page (Dec. 2007–Jan. 2016). The dataset exhibits both seasonality and a holiday effect (e.g. around super bowl games). We use the first 6 years as the training set and the last 2 years as the testing set. We use a cross-validation horizon (sliding window) of 1-year, and a period (sliding pace) of 180 days. As such, Seneca performs three cross-validations for a 2-year test range.

Prophet expects multiple hyperparameters: *growth* specifies linear or logistic trend model growth and *prior scale* indicates the strength of the sparse prior probability. There are three prior scale hyperparameters for change point, holidays, and seasonality. Since Prophet uses a Fourier sum to estimate seasonality, the *fourier order* is the number of terms in the partial Fourier sum. *Seasonality mode* indicates that the effect of seasonality is either multiplicative or additive. Finally, the width of uncertainty intervals is set using *interval width*.

Each application has default hyperparameter settings (i.e. default values or those recommended by the application maintainer). The hyperparameters, their default and optional values that we consider for Prophet are listed in the Table II.

Multi-Regression is a regression application developed by others as part of an Internet-of-Things (IoT) project [13] (which has been extended from linear regression described in the citation to multiple linear regression by the authors of this prior work). The application uses multiple linear regression models to predict outdoor temperature from the processor temperature of single board computers (SBCs). The training dataset consists of eight input time series (one per SBC, each containing 5-minute measurements) from Apr. 5th to Dec. 10th, 2018.

Hyperparameter configuration for Multi-Regression is a subset of input SBC time series. Seneca considers all $2^N - 1$ non-empty potential subsets (for N input time series). For this application, the default parameterization is the full set of input time series (8 in this case). The test dataset is a time series of the outdoor temperature (ground truth) over the same period. The application makes predictions for each of these outdoor temperatures using the regression coefficients constructed from the training set for each new value in the test set.

XGBoost [14], SVC [15], and NN [16] are the classification applications that we consider. XGBoost [14] is an open source framework for gradient boosting, which performs both regression and classification. The hyperparameters and their default values are listed in Table III, their definitions can be found in [17]. SVC uses support vector machines to implement classification as part of the libsvm [18] library. The hyperparameters and their defaults that Seneca uses for SVC in this study are listed in Table IV with definitions in [19]. NN is a machine learning application leveraging neural network to identify patterns from an input dataset. Here we implement a feed-forward multi-layer perceptron model [20] for classification. The hyperparameters and their defaults for NN are listed in Table V with definitions in [21].

For these classification applications, we use a labeled dataset for training, testing, and evaluation from another IoT project [22]. The dataset contains measurements of individual citrus fruit (e.g. oranges, mandarins, lemons, etc.) taken by a fruit sorting and grading device using a large number of sensors. The measurements (i.e. features) include size, shape, weight, color, diameter, flatness, among other characteristics, for each fruit. The dataset has been filtered to remove correlated features (those with an absolute value of the Pearson correlation coefficient greater than 0.8). The dataset

has been balanced by down-sampling and the resulting dataset contains 33926 rows (individual fruit) distributed evenly across 5 targets. Each row has 18 features. The label identifies the field from which the individual fruit was harvested.

The applications train a model on a random subset (80%) of the data. Each then uses this model to predict the field from which each fruit originates for the remaining 20%. To study the impact of random data split, we consider multiple 80%/20% splits in our evaluation.

B. Empirical Methodology

To evaluate Seneca, we measure model output quality, execution time, memory use, and monetary cost. For output quality (prediction accuracy) our metrics are mean squared prediction error (MSE) for regression and percentage accuracy for classification as described above.

We compare results for the default, best (Seneca's recommendation), and worst performing hyperparameter configurations for each application type. Seneca computes all possible combinations of the hyperparameter settings specified in the configuration to extract each of these results. `default` represents results that a novice or first time user might experience when using these applications as a "black box." The `worst` shows how bad the results can be when parameters are poorly tuned. Finally, the `best` is the upper bound on what is possible from tuning the hyperparameters for the values and datasets specified (e.g. using expert knowledge or Seneca).

Seneca deploys the applications automatically over AWS Lambda and extracts execution time and memory use from AWS CloudWatch [23] logs. We compute monetary cost using the AWS Lambda pricing model [7]. Each function downloads the training/testing dataset of the application from AWS S3 upon function invocation. We do not consider the cost of dataset storage in our cost computations, because it is very small. For XGBoost that makes most S3 requests among others, the cost is less than 2.5 cents for storage and request combined in a month. We also evaluate Seneca's automatic memory optimization capabilities. To do so we compare the execution performance and cost of the applications using the maximum allocatable memory size to the performance and cost when run with Seneca's automatically determined memory size. Even though `maximum memory used` reported by AWS CloudWatch can fluctuate, we have verified that the optimized allocated memory is sufficient for all hyperparameter configurations to complete successfully. We have also verified that the memory requirements across hyperparameter settings do not vary significantly. We plan to consider applications for which hyperparameter settings require different maximum memory sizes as part of future work.

C. Application Efficacy

We first evaluate the quality of the output generated by each ML applications when Seneca determines the hyperparameter settings. We first show the results for the regression applications in Table VI. The first row of data is the number of hyperparameter configurations that Seneca considers for each. The last three rows show the MSE for the default, worst, and best performing (Seneca's recommendation) hyperparameter configuration (lower is better). Seneca reduces MSE by 22.56%, 15.94%, and 44.88%, for Prophet, Multi-Regression, and XGBoost, respectively, for the datasets and training methodologies that we consider. Compared to the worst case, Seneca reduces MSE by 82.62%, 78.01%, and 99.28%, respectively.

	Prophet	Multi-Regression	XGBoost
# of Combinations	384	255	768
Default MSE	0.284	11.446	0.118
Worst MSE	1.266	43.752	8.981
Best MSE (Seneca)	0.220	9.621	0.065

TABLE VI: Hyperparameter configuration count and MSE

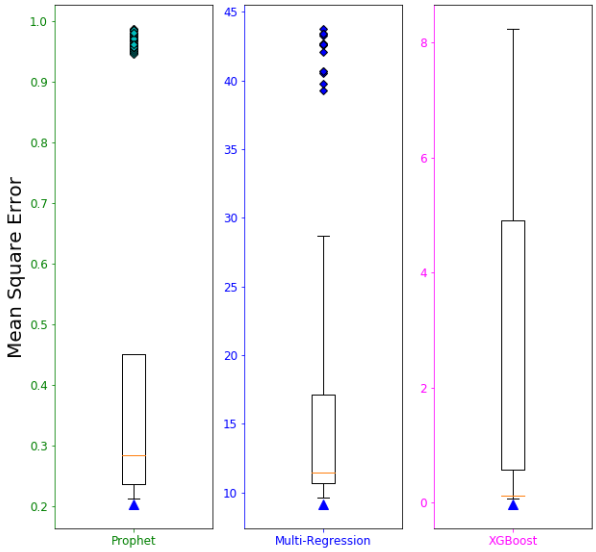

Fig. 4: Box plot of MSE from the three regression applications across the hyperparameter tuning search space. The red notch shows the MSE from the default settings. The colored diamonds are outliers beyond two interquartile ranges. Seneca selects the points indicated by blue triangle. Lower MSE values are better.

Figure 4 shows the MSE box plot for the hyperparameter search space for these applications (lower is better). The central rectangle covers the interquartile range (IQR), which is defined as the range of data points from first quartile to third quartile $(Q3 - Q1)$. The upper whisker extends to the last datum less than $(Q3 + 2 * IQR)$ and the lower whisker extends to the first datum greater than $(Q1 - 2 * IQR)$. The data points beyond the whiskers are considered outliers and are plotted as colored diamonds. The red notch identifies the MSE that results from training the model using the default settings of hyperparameter. The blue triangle identifies the MSE of Seneca. The difference between red notch and blue triangle is the improvement brought about by the use of Seneca, over using the default parameter setting. The plot also shows that

80%-20%	XGBoost	SVC	NN
# of Combinations	768	512	432
Default Accuracy	95.65%	21.77%	79.53%
Worst Accuracy	0.00%	14.08%	19.15%
Best Accuracy (Seneca)	98.11%	40.81%	83.32%

TABLE VII: Accuracy for the default, best (Seneca's recommendation), and worst hyperparameter configurations for the three classification applications using 80% of the data to train

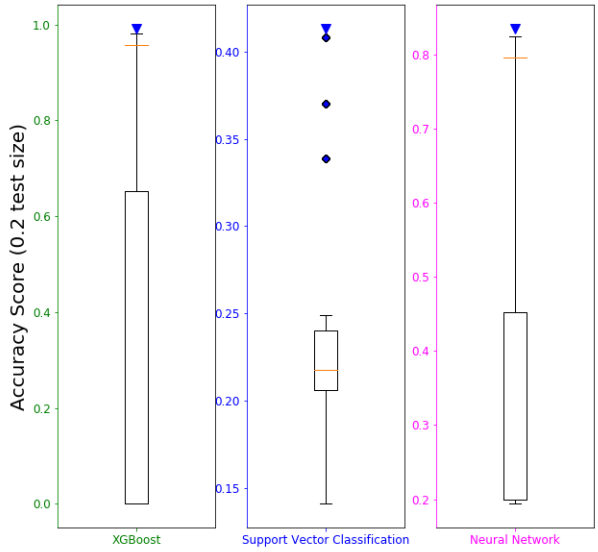

Fig. 5: Box plot of accuracy reported for three classification applications across the hyperparameter search space. The red notch indicates the accuracy that results from default hyperparameter values. The diamonds are outliers beyond two interquartile ranges. Seneca selects the points indicated by blue triangle. Higher accuracy is better.

Prophet and Multi-Regression have a significant number of outliers, indicating that a comprehensive search is critical to finding the best configurations.

We next empirically evaluate Seneca's model output quality for the three classification applications: XGBoost (classification), SVC, and NN. Table VII presents the accuracy percentage (higher is better) for each application (3 rightmost columns) for the default, worst, and best (Seneca's recommendation) hyperparameter tuning configurations (data rows 2-4). The first row reports the number of configurations that Seneca considers in its search space. Using a random 80/20 (train/test) percent split, Seneca increases accuracy by 2.46%, 19.04%, and 3.79%, for XGBoost (classification), SVC, and NN applications, respectively. Because XGBoost and NN use a well-tuned default parameter set that works well for most datasets, Seneca provides only modest improvements. Compared to the worst case however, Seneca improves accuracy by 98.11%, 26.74%, and 64.17%, respectively.

Figure 5 presents the accuracy box plot across the hyperparameter search space for these applications (higher is better).

	Exec Time (Secs)	Memory Use (MB)	Best Accuracy
XGBoost_1	1244.42 (32.58)	228.74 (15.92)	98.11%
XGBoost_2	1280.56 (38.47)	225.10 (19.67)	97.70%
SVC_1	116.73 (1.11)	224.44 (19.64)	40.81%
SVC_2	115.33 (3.96)	228.55 (16.20)	44.12%
NN_1	116.10 (6.05)	328.84 (16.40)	83.32%
NN_2	121.29 (2.18)	327.57 (16.44)	83.92%

TABLE VIII: The mean and standard deviation (in parentheses) for execution time and memory use (across 30 runs), and best accuracy score for the classification applications using two different random splits.

The central rectangle covers the first-third quartile $(Q3-Q1)$ and the whiskers span from $(Q3+2*IQR)$ to $(Q1-2*IQR)$. The red notch indicates the accuracy metric from the model trained using the default settings and colored diamonds show outliers beyond the whiskers. The blue triangle at the top identifies the accuracy percentage reported by Seneca.

The model output quality results across applications, show that prediction accuracy (for a given dataset) is dramatically affected by hyperparameter settings. Predictably, the default settings are near the "good" end of the spectrum, however, Seneca is able to find the parameterization that improves output quality over the default settings in each case.

To investigate the potential impact of Seneca's 80/20 percent data split for the classification applications, we next evaluate the quality of the output generated from each when we consider different 80/20 random splits. For that purpose, we run Seneca 30 times to obtain execution time, memory use, and best accuracy score. We report the mean and standard deviation (in parentheses) for execution time and memory use across runs, and the best accuracy score in Table VIII. Each pair of rows shows the results for two different random splits. Our earlier results use input 1; this table adds results for a second, 80/20 random split of the input (we also considered other random splits, which we omit for brevity, and the results are similar). The performance and Seneca score is similar across splits. This result indicates that for these applications, users can repeatedly employ the recommended models for inference on other datasets or splits, to amortize the cost of using Seneca.

D. Cost Analysis

We next analyze the monetary cost incurred by Seneca with and without Seneca's memory optimization. We consider the use of the maximum allocatable memory (3GB) and Seneca's automatic detection and configuration of allocated memory. This optimization requires that Seneca intelligently probe to determine the best memory size to use. We report the cost of these probes as `Optimizer Cost`.

Table IX shows the results with and without the Seneca memory optimization for each of the five applications. The first two rows show the results when we use the maximum allocated memory for the Lambda functions. We present execution time in minutes (row 1) and monetary cost in cents (row 2). Rows 3–6 show the performance and cost when using Seneca's memory optimization. `Exec time opt` is the execution time in minutes. `Optimizer Cost` is monetary cost in cents of

	Prophet	MR	XGBoost	SVC	NN
Exec time max (mins)	7.78	2.71	20.85	0.87	2.06
Cost max (cents)	22.16	7.58	59.92	2.21	5.92
Exec time opt (mins)	12.60	4.09	29.07	2.08	3.06
Optimizer Cost (cents)	2.02	1.27	0.05	0.04	0.04
Cost opt (cents)	17.81	4.47	39.76	1.38	4.39
Total Cost (cents)	19.83	5.74	39.81	1.43	4.44
Savings (cents)	2.33	1.84	20.11	0.78	1.48
Savings (%)	10.49%	24.23%	33.57%	35.42%	24.98%

TABLE IX: Seneca Memory Optimization: Rows 1–2 show the execution time and monetary cost of using Seneca without its memory optimization (allocated memory = 3G). Rows 3-6 is the execution time and cost, respectively, when using the Seneca memory optimizer. Rows 7-8 show the savings in cents and percentage, respectively.

	Prophet	Multi_Reg	XGBoost	SVC	NN
EC2 exec time (mins)	73.79	21.99	359.87	7.74	15.92
EC2 total cost ($)	0.083	0.042	0.25	0.042	0.042
yield	50.86	340.23	83.36	0.00	1875.56
ideal yield	25.43	170.12	41.68	0.00	937.78

TABLE X: Seneca VS EC2 cost analysis. Execution time (mins) and cost (dollars) for executing the applications serially in EC2 (t2.medium). Rows 3–4 show yield – the additional speedup that Seneca can achieve for each additional dollar spent for these applications. Yield for SVC is 0 (infinite) because Seneca costs less than EC2 in this case. Ideal yield shows the yield when we execute the applications in parallel (assuming 2x perfect parallelism).

Seneca's memory size detector. `Cost opt` is monetary cost in cents of using Seneca's memory optimizer. `Total cost` is the overall cost of using Seneca to perform hyperparameter tuning for these applications and datasets (sum of Optimizer Cost and Cost opt). The last two rows show the monetary savings in cents (row 7) and percent savings (row 8) of using Seneca's memory optimization. Seneca's memory optimization reduces the monetary cost of its use from 10–35% (25% on average).

Table IX illustrates two important points. First, using AWS Lambda, full hyperparameter space exploration is inexpensive in *absolute* dollar cost terms and Seneca's automatic memory size optimization decreases this cost further. Second, memory optimization reduces cost but can increase the total execution time for parameter search since the functions must operate under additional memory constraints (versus using the maximum allocated memory). In addition, this cost fluctuates depending on the quality of the Lambda execution environment (number of CPUs, Linux container overhead, multitenancy, etc.). We omit this data due to space constraints but analyze it here. The average absolute difference in cost across the five applications (30 runs) is $0.05. Moreover, we have verified that the highest cost of execution under optimized memory is still cheaper than the lowest cost of execution under maximum memory for all five applications.

Finally, we compare the cost of Seneca to the cost of using AWS Elastic Compute Cloud (EC2). We measure the execution time of Seneca using the least expensive EC2 instance type in which the applications will run (t2.medium, which has 2 multi-tenant cores and 4GB of memory). Note that EC2 instances are charged for by the hour; Lambda charges are only imposed when functions execute, but Lambda *may* execute the functions concurrently. For this comparison, Seneca (because Lambda's automatic exploitation of concurrency) enables a speedup over EC2 (where we execute the parameter sweep sequentially) of 3.72x – 12.38x (6.51x on average). This speedup comes at an additional cost of $0.01–$0.15 over EC2 for all but SVC (which executes for significantly less than an hour).

To further understand the relationship between Seneca speedup and cost when Seneca is more expensive (but faster) than using EC2, we define `yield` as $Y = \frac{T_{ec}}{T_{sc}}/(C_{sc} - C_{ec}) \mid if\, C_{sc} > C_{ec}$ where T_{ec} and T_{sc} are the execution time, C_{ec} and C_{sc} are the total cost of EC2 instance and Seneca, respectively. For applications for which Seneca is cheaper (e.g. SVC), we report `yield` as 0.00 since there is no positive benefit/cost ratio. This metric captures the amount of speed up that Seneca can achieve for each additional dollar spent. To make the comparison "fair" we also explore yield for theoretically perfect parallelism in EC2 using 2 cores (e.g. in a t2.medium).

We present results for this yield metric in Table X for each of the applications. Rows 1 and 2 show the average execution time (in minutes) and cost (in dollars) from using EC2 for each. Rows 3 and 4 show the Seneca yield (speedup/$). Row 3 shows yield for serialized execution in EC2 (t2.medium instance) and row 4 shows estimated yield if we were to achieve perfect parallelism (i.e. 2x) using the EC2 instance. On average across the four applications for which EC2 is cheaper, Seneca achieves yield of 294 (assuming perfect parallelism in EC2). That is, Seneca is able to provide a speedup of 294x on average, for each additional dollar spent for these applications. It is tempting to attribute this effect to the EC2 full-hour charge for short running applications, but XGBoost runs for almost 6 hours and the yield is still 41.86 assuming best-case EC2 execution. We plan to study this benefit/cost ratio for Seneca and Lambda applications as part of future work.

Overall, given the AWS Lambda pricing model and its Lambda performance variability, Seneca is still able to find the sweet spot between cost and execution time. Thus Seneca can be used to trade off time-to-solution for cost as desired by users, to automatically evaluate the impact of hyperparameter settings for ML models.

IV. Related Work

As related work, we consider recent advances in evaluating serverless computing for different application domains, automatic deployment for serverless, and machine learning (ML) model optimization. For the former, much work has investigated the efficacy and overhead of the serverless programming model and implementations [1], [2], [24], [25]. The authors identify challenges with using AWS Lambda to train machine learning (ML) models. Our work however, shows that it is possible to leverage the concurrency and parallelism in AWS

Lambda to perform fast grid search for the subset of ML applications that we consider.

PyWren [1] uses serverless for different distributed computing models. The technique abstracts away cluster management overhead and is ideal for embarrassingly parallel jobs. ExCamera [26] presents a framework for running general-purpose parallel tasks (encoding 4K video) on a commercial serverless platform using multithreading. Cirrus [27] attempts to train ML models using a parameter server and serverless functions.

The serverless framework [28] provides automated packaging and deployment for serverless functions across clouds. GammaRay [29] does so for AWS Lambda to insert profiling instrumentation. The serverless framework uses CloudFormation [30] for deployment in AWS Lambda, which introduces additional cost. The cloud infrastructure provisioning framework Terraform [31] also provides automated deployment of functions to serverless platforms. Seneca uses a local Docker container to avoid cost and overhead (vs these related works), which guarantees execution compatibility for AWS Lambda.

Automated hyperparameter tuning is the focus of many projects. Google Vizier [32] provides a service for black-box optimization. Optunity [33] and Hyperopt [34] provide a Python library for hyperparameter tuning. Hyperas [35] adds another abstraction layer to hyperopt to facilitate hyperparameter tuning for Keras [36]. However, we are not aware of any work that leverages serverless to perform hyperparameter tuning and memory optimization in parallel for ML applications.

V. Conclusion

We present a new framework, called Seneca, for simplifying and expediting the training and testing of machine learning models in AWS Lambda. Users provide Seneca with the application code and libraries, 1+ datasets, and the list of possible hyperparameter settings. Seneca uses this information to automatically configure and deploy these functions concurrently for all possible combinations of hyperparameter values specified. Seneca returns the best scoring model and configuration to the user for future use on other datasets.

We present the design, implementation, and cost optimization for Seneca. The optimizer automatically optimizes function memory use to reduce the cost of AWS Lambda use. Our empirical evaluation using multiple applications for regression and classification, shows that Seneca is able to quickly identify the best performing hyperparameter setting for the applications and datasets that we consider. We also find that Seneca enables average speedups of 294x for each additional dollar spent, and that its memory optimization reduces the cost of using Seneca by 10-35% for the applications studied.

References

[1] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 445–451.

[2] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018.

[3] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *USENIX ATC*, 2018.

[4] (2019, Jan.) Aws lambda developer guide. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf

[5] (2019, Jan.) Amazon cloud object storage. [Online]. Available: https://aws.amazon.com/s3/

[6] (2019, Jan.) Aws command line interface. [Online]. Available: https://aws.amazon.com/cli/

[7] (2019, Jan.) Aws lambda pricing. [Online]. Available: https://aws.amazon.com/lambda/pricing/

[8] (2019, Jan.) Basic aws lambda function. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html

[9] S. Werner, J. Kuhlenkamp, M. Klems, J. Muller, and S. Tai, "Serverless big data processing using matrix multiplication as example," 2018.

[10] (2019, Jan.) Celery. [Online]. Available: http://www.celeryproject.org/

[11] (2019, Jan.) Redis. [Online]. Available: https://redis.io/

[12] S. Taylor and B. Letham, "Forecasting at scale," *PeerJ Preprints*, vol. 5, Sep. 2017.

[13] C. Krintz, R. Wolski, , N. Golubovic, and F. Bakir, "Estimating outdoor temperature from cpu temperature for iot applications in agriculture," in *International Conference on the Internet of Things*, 2018.

[14] (2019, Jan.) Xbgoost. [Online]. Available: https://xgboost.ai

[15] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[16] S. Haykin, *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[17] (2019, Jan.) Xgboost scikit-learn api. [Online]. Available: https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn

[18] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[19] (2019, Jan.) Sklearn support vector classifier. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[20] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, 2010.

[21] (2019, Jan.) Sklearn neural network mlpclassifier. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[22] "blinded for submission/review," Feb 2019.

[23] (2019, Jan.) Amazon cloudwatch. [Online]. Available: https://aws.amazon.com/cloudwatch/

[24] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[25] W.-T. Lin, C. Krintz, R. Wolski, M. Zhang, X. Cai, T. Li, and W. Xu, "Tracking causal order in aws lambda applications," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, 2018.

[26] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 363–376. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[27] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[28] (2019, Jan.) Serverless framework. [Online]. Available: https://serverless.com/

[29] W.-T. Lin, C. Krintz, R. Wolski, and M. Zhang, "Tracking Causal Order in AWS Lambda Applications," in *IEEE International Conference on Cloud Engineering*, 2018.

[30] (2019, Jan.) Aws cloudformation. [Online]. Available: https://aws.amazon.com/cloudformation/

[31] (2019, Jan.) Terraform by hashicorp. [Online]. Available: https://www.terraform.io/

[32] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *ACM SIGKDD*, 2017.

[33] M. Claesen, J. Simm, D. Popovic, and B. Moor, "Hyperparameter tuning in python using opportunity," in *Workshop on Technical Computing for Machine Learning and Mathematical Engineering*, 2014.
[34] (2019, Jan.) Hyperopt. [Online]. Available: http://hyperopt.github.io/hyperopt/
[35] (2019, Jan.) Hyperas. [Online]. Available: http://maxpumperla.com/hyperas/
[36] (2019, Jan.) Keras. [Online]. Available: https://keras.io/