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*}*@cs.ucsb.edu*

Markus Mock
Dept. of Computer Science
University of Applied Sciences, Landshut Germany
mock@haw-landshut.de

***Abstract*— The goal of our work is to simplify and expedite the construction and evaluation of machine learning models using autoscaled cloud computing resources. To enable this, we develop an open source system called Seneca, which 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 as input and automatically constructs an AWS Lambda function. The function ingresses and splits the input 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. In this paper, we overview the design and implementation of Seneca, and empirically evaluate its performance for a popular classification application.**

***Keywords*-Serverless computing; Machine-learning model selection; 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 prediction accuracy across models. Users present Seneca with their application, a range of values 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 (and/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

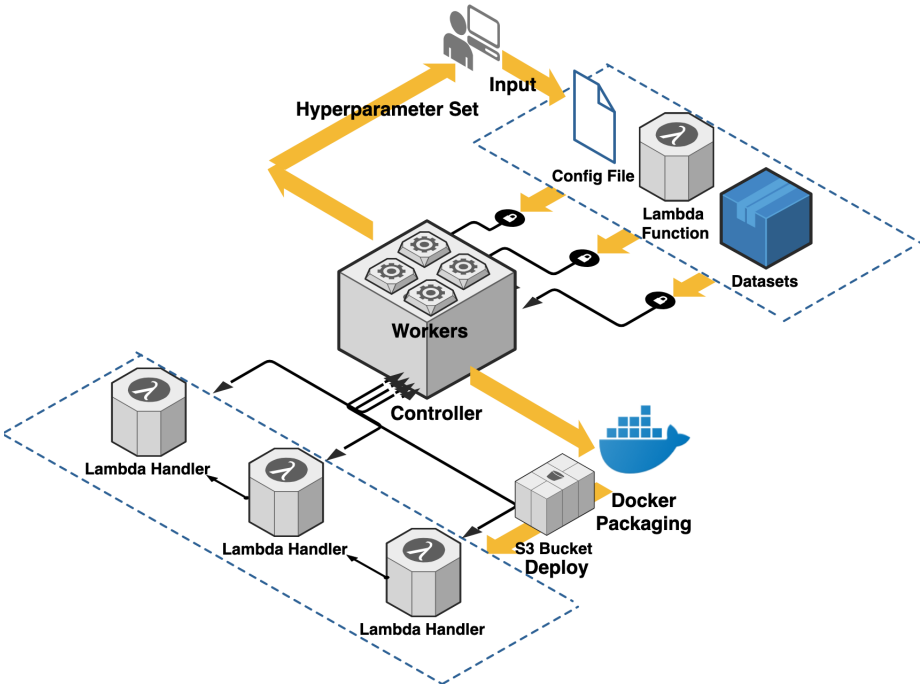

Fig. 1: The Seneca Architecture.

performance, cost, and memory use for a machine learning application benchmark and dataset. Our results indicate that Seneca is fast, inexpensive, and effective for model construction and comparison. We also observe for this application, that Seneca enables a speedup of 220x for each additional dollar spent performing hyperparameter search in AWS Elastic Compute Cloud (EC2). We next overview our design and implementation of Seneca and present our empirical methodology and results. Finally, we discuss related work and conclude.

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 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).

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 task. 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 (CLI) and the developers credentials.

To facilitate parallel function invocation, Seneca integrates Celery [1], 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[2] in our prototype).

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.

[1]http://www.celeryproject.org/
[2]https://redis.io/

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 I: Hyperparameters Seneca considers for **NN**.

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 application benchmark that we consider and our experimental methodology. We then present our results.

A. Benchmark Application and Training/Testing Dataset

We benchmark Seneca using a classification application, called NN. NN uses a neural network to identify patterns in an input dataset. It employs a feed-forward, multi-layer perceptron model [5] to perform the classification. The hyperparameters and their defaults for NN are listed in Table I with definitions in [6].

During testing, NN computes and returns a classification accuracy percentage. Accuracy percentage 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.

With respect to training and testing, we use a labeled dataset for training, testing, and evaluation from the UCSB SmartFarm project [7], [8]. SmartFarm is a multi-tier (sensor, edge, cloud) system that aggregates, fuses, and analyzes data from farm operations to provide growers with data-driven decision support, actuation, and control.

The dataset contains measurements of individual citrus fruit (e.g. oranges, mandarins, lemons, etc.) taken by a fruit sorting and grading device (e.g. https://www.compacsort.com) 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 down-sampled and filtered to remove correlated features (those with an absolute value of the Pearson correlation coefficient greater than 0.8). The dataset contains 33926 rows (individual fruit) distributed evenly across 5 targets (fields). Each row has 18 features. The label identifies the field from which the individual fruit was harvested.

The application trains the model on a random subset (80%) of the data. It 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.

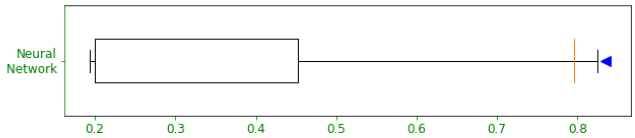

Fig. 2: Box plot of NN accuracy (higher is better) across the hyperparameter search space. The orange notch indicates the accuracy that results from default hyperparameter values. Seneca selects the points indicated by blue triangle.

	Exec Time (Secs)	Memory Use (MB)	Best Accuracy
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 II: The mean and standard deviation (in parentheses) for execution time and memory use (across 30 runs), and best accuracy score for neural network classification (NN) using two different random splits.

B. Empirical Methodology

To evaluate Seneca, we measure model output quality (accuracy percentage), execution time, memory use, and monetary cost. We compare results for the default, best (Seneca's recommendation), and worst performing hyperparameter configurations for the application. 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 [3] logs. We then compute monetary cost using the AWS Lambda pricing model [4]. The lambda 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, less than 2.5 cents per month for storage and data access.

C. Application Efficacy

We empirically evaluate Seneca's model output quality across the hyperparameter search space for NN, using the accuracy box plot in Figure 2. 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 right whisker extends to the last datum less than $(Q3+2*IQR)$ and the left whisker extends to the first datum greater than

[3]https://aws.amazon.com/cloudwatch/
[4]https://aws.amazon.com/lambda/pricing/

$(Q1 - 2 * IQR)$. The rightmost blue triangle identifies the accuracy percentage reported by Seneca. The difference between the orange notch and the blue triangle represents the improvement brought about by the use of Seneca, over applying the default parameter setting. As shown, the default accuracy of the classification application is 79.53%, while the worst accuracy is 19.15%. After the tuning process, Seneca selects the best performing setting, which achieves the best accuracy of 83.32% across 432 configurations considered.

The model output quality results show that prediction accuracy (for a given dataset) is dramatically affected by hyperparameter settings. Unsurprisingly, the default setting (provided by the NN application developers) is near the "good" end of the spectrum. However, Seneca finds a parameterization that significantly improves output quality over the default setting.

To investigate the potential impact of Seneca's 80/20 percent data split for the classification application, we next evaluate the quality of the output when we consider different 80/20 random splits. For this 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 II. 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 specific 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 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 NN, Seneca (using 3008 MB allocated memory) completes the tuning process in 123.49 seconds with total cost of $0.059. In contrast, EC2 completes the tuning task in 955.47 seconds with total cost of $0.042. Because Seneca autoscaling facilitates much greater concurrency, it enables a speed up over EC2 of 7.74x for an additional cost of $0.018 over EC2.

To understand the relationship between Seneca speedup and monetary cost (when Seneca is more costly than EC2), we define a `yield` metric 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, measured in dollars, respectively. In applications for which Seneca is cheaper, 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).

For NN, Seneca achieves a yield of 220/$, assuming perfect parallelism in EC2. That is, Seneca is able to provide a speedup of 220, for each additional dollar spent for the application. We plan to study this benefit/cost ratio of Seneca and Lambda applications for various allocated memory and applications as part of future work.

Overall, given the AWS Lambda pricing model and its Lambda performance variability, Seneca is 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 machine learning 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], [9], [10]. 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 [11] presents a framework for running general-purpose parallel tasks (encoding 4K video) on a commercial serverless platform using multithreading. Cirrus [12] attempts to train ML models using a parameter server and serverless functions.

An empirical study of FaaS software development has been conducted in [13] that indicates that developers struggle with the technical restrictions inherent in the model. To address these restrictions (e.g. cold-start, memory and duration limit, lack of local persistent storage, etc.), the serverless framework [14] provides automated packaging and deployment for serverless functions across clouds. The framework however relies on CloudFormation [15] which adds to the cost of FaaS use. Similarly, Terraform [16] provides automated deployment of functions to serverless platforms. GammaRay [17] uses similar packaging to to simplify FaaS deployment and to facilitate profiling for AWS Lambda applications. Finally, the authors of [18] model the performance of shuffle operations executed on FaaS architecture with a mixture of slow and fast storage. Seneca also provides automatic deployment and performance profiling. However, it does so using a local Docker container to avoid cost and overhead (vs these related works), to guarantee execution compatibility for AWS Lambda.

Automated hyperparameter tuning is the focus of many projects. Google Vizier [19] provides a service for black-box optimization. Optunity [20] and Hyperopt [21] provide

a Python library for hyperparameter tuning. AWS SageMaker [22] provides an automatic service that performs hyperparameter tuning for machine learning models. Hyperas [23] adds another abstraction layer to hyperopt to facilitate hyperparameter tuning for Keras [24]. 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

In this paper, we present a new framework, called Seneca, for simplifying and expediting the training and testing of machine learning models using 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 discuss the design and implementation of Senaca, and the cost optimization it performs. Our empirical evaluation using a machine learning classification application, shows that Seneca is able to quickly identify the best performing hyperparameter configuration for the application and datasets that we consider.

Acknowledgments

This work is funded in part by NSF (CNS-1703560, OAC-1541215, CCF-1539586, ACI-1541215), ONR NEEC (N00174-16-C-0020), and the AWS Cloud Credits for Research program.

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] "AWS Lambda," https://docs.aws.amazon.com/lambda/, [Online; accessed 1-May-2019].

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

[6] (2019, Jan.) Sklearn neural network mlpclassifier. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

[7] "UCSB SmartFarm," http://www.cs.ucsb.edu/~ckrintz/projects/index.html, [Online; accessed 14-June-2018].

[8] C. Krintz, R. Wolski, N. Golubovic, B. Lampel, V. Kulkarni, B. Sethuramasamyraja, B. Roberts, and B. Liu, "SmartFarm: Improving Agriculture Sustainability Using Modern Information Technology," in *KDD Workshop on Data Science for Food, Energy, and Water*, Aug. 2016.

[9] 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.

[10] 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.

[11] 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

[12] 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.

[13] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340 – 359, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218302735

[14] (2019, Jan.) Serverless framework. [Online]. Available: https://serverless.com/

[15] (2019, Jan.) Aws cloudformation. [Online]. Available: https://aws.amazon.com/cloudformation/

[16] (2019, Jan.) Terraform by hashicorp. [Online]. Available: https://www.terraform.io/

[17] 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.

[18] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu

[19] 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.

[20] 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.

[21] (2019, Jan.) Hyperopt. [Online]. Available: http://hyperopt.github.io/hyperopt/

[22] "AWS SageMaker," https://aws.amazon.com/blogs/aws/sagemaker-automatic-model-tuning/, [Online; accessed 1-May-2019].

[23] (2019, Jan.) Hyperas. [Online]. Available: http://maxpumperla.com/hyperas/

[24] (2019, Jan.) Keras. [Online]. Available: https://keras.io/