OptML

Optimization for Machine Learning

Ruben Vazquez
Electrical and Computer Engineering
University of Florida
Gainesville, United States
ruben.vazquez@ufl.edu

Justin Calean
Electrical and Computer Engineering
University of Florida
Gainesville, United States
jcalean@ufl.edu

Abhishek Parekh
Computer Information and Science and Engineering
University of Florida
Gainesville, United States
abhishekparekh@ufl.edu

***Abstract*: Machine learning is a statistical process used in computer science where machines to "learn" and recognize data that the machine is not specifically programmed to understand. Machine learning has applications in many commercial and research domains, but creating these high-performance applications can be daunting. Building an accurate model requires a lot of resources to finely tune the program enough so that the error is minimized without compromising speed. Finding the hyperparameters that lead to the best performing models can take too much training time that most people cannot afford to waste. The emergence of cloud computing can solve this dilemma by speeding up the process of testing large amounts of different hyper parameter configurations on a large set of data. We plan to develop a cloud-based application that finds the optimal parameters for creating a low-error, high-performing model.**

Keywords—Machine Learning, Hyperparameter Optimization, Cloud

I. Introduction

Machine learning is a field in engineering and computer science that uses statistical methods to develop and train models that can learn patterns from a set of training data and to use those trained models to make inferences on newly acquired data. The ideas of machine learning have been applied to many different fields of research, including computer architecture, speech recognition, and computer vision. In addition, machine learning is applicable to a wide variety of problems and many learning algorithms have been used [10]. Some of these applications include pattern recognition, classification, prediction etc. and some widely recognized learning algorithms are the Least Mean Squares (LMS) [3] method and the backpropagation method for Artificial Neural Networks (ANNs) [1]. Traditionally, machine learning has done on high-performance computing systems because of the processing demand of the learning algorithms requiring large amounts of data. With the advent of cloud computing systems, the processing can be done in a distributed manner across multiple computing systems with guaranteed performance and reliability across these systems.

Machine learning boils down to training algorithmic models to infer and classify unknown data sets. To improve model accuracy, the algorithms must run through massive datasets to test inferences, while striking the balance between generalized and specialized models. Essentially, models must be specialized enough to identify an object correctly, but should be generalized enough to identify variations of the objects with similar characteristics. Another problem for improving performance of a model is selecting the right techniques to solve the underlying problem. Techniques such as linear regression may work for certain algorithms, but perform poorly in others. These training dilemmas leads into the concept of hyperparameter tuning, which can take even more time and resources in an already labor intensive process.

Hyperparameters are the parameters of a learning algorithm that a developer can alter to meet the needs of a specific application. In the backpropagation algorithm, on the hyperparameters is known as the learning rate, which controls the amount of change of the weights of an ANN. Having a value that is too large or too small can negatively affect the performance of the learning algorithm. There are currently many different hyperparameter tuners available for commercial use. These tuners tend to be native to a single cloud infrastructre such as Google Cloud or AWS, which can be offputting for companies and researchers with a personal cloud.

In order to speed up the optimization process, we propose the creation of a brute force application implemented on the AWS cloud. The application is expected to run through different hyperparameter configurations to find the lowest error from a given model, algorithm and dataset. The

hyperparameter configurations will be the number of leaves in each row of the tree and the training rate. Other hyperparameters will be added in future work. The containers management will be done with Kubernetes.

II. Background

A. Hyperparameter Optimizers

The traditional approach to hyperparameter tuning is grid search. Here, a manually input set of hyperparameters is exhaustively searched. to find the combination with the lowest error. This approach is simple to implement and understand because it is a brute force method. The model testing is embarrassingly parallel, but suffers the curse of dimensionality. As the number of dimensions increases due to the vast number of hyperparameters so the does the workload.

A spin-off of grid search is the random search. The hyperparameters values for the configuration are randomly selected within a specified range and then evaluated. This approach is less rigid than grid search because the program can run for any number of model tests as compared to the permutations of the power set. The algorithm has been known to perform significantly better than grid search when the number of hyperparameters that affect performance is small. The main issue with this approach is that there is less guarantees on finding the best parameter combinations due to the nature of randomness.

The popular method right now is bayesian approach, where the hyperparameters to test can be induced based on the previous iterations of testing. Bayes will build a probabilistic model in an attempt to incrementally improve upon previous promising areas. The approach includes iterations of models that discover areas outside of the previous domains in order to find more promising areas to exploit. Bayes is significantly less parallel due to the dependence of having knowledge about the problem. [5]

There are additional methods and hybrid approaches to discover the optimal hyperparameters each with their own benefits and drawbacks depending on the algorithm. Grid search, random search and Bayesian are currently the most popular and have the most support in the community. Bayesian is commonly supported in cloud infrastructures such as Amazon's Sage Maker or Google Cloud. If people want to do their own tuning to save money, Python has an extensive selection of packages and libraries. Comet.ml provides easy-to-use SDK's so that there is no need to set up dependencies or servers. The API also easily builds, tests and compares these models according to the testing specified by the user.

B. Machine learning algorithms

The algorithms that we will use to test our optimization process will be the Least Mean Squares (LMS) and the backpropagation algorithm for Artificial Neural Networks (ANNs). The Least Mean Squares algorithm trains a set of weights by iteratively updating each weight using the error calculated between the desired output and predicted output of the algorithm for each data point in the training input. This error is multiplied by the input data point that generated the error and a scaling factor, known as the learning step. This calculated value is added to the weights to effectively update them. The backpropagation algorithm is used to train ANNs by calculating an error at the output layer to update the weights, and propagating the error to the hidden layers to update the weights for the hidden layer. Both algorithms are very popular in the literature and have been used to solve problems from many domains.

The LMS and backpropagation algorithm have different parameters that can be controlled by the user, known as hyper parameters. The hyperparameter for the LMS algorithm is the learning step. The hyperparameters for the ANN are the learning rate, which controls how much the weights are updated after each iteration, and the size of the ANN which determines the number of layers and the number of processing elements in each layer. Our proposed optimization method will determine the best sets of hyperparameters for the three algorithms from a given set of values from the user.

III. System Architecture and Design

We wish to create a containerized microservices application that can support auto-scaling, rolling updates, load balancing, high availability and fault tolerance. Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications. Kubernetes manages containers by grouping them into a logical entity called pods. These pods may contain one or more containers and can be further managed by replication controllers, deployments etc. Kubernetes schedules the pods to run in a cluster-based environment from the available resources (VM, CPU, Hardware).

We plan to have multiple AWS EC2 instances dedicated to different services. A fleet of EC2 instances will be used to expose the front-end of our application. These instances will be behind an autoscaling group and an external load balancer serving queries from the outside world. These front-end instances will be accessed by the user via CLI, REST API or UI. The UI will be designed as a webpage using HTML5, CSS3 and Javascript. When the user hits the external load balancer the query will be resolved via a kubernetes service to one of the pods containing docker containers with web page deployed.[8][9]

The business logic is contained within another set of EC2 instances that contain worker pods. These pods work on a stateless application to do their jobs. They are entirely scalable i.e. if one of them dies, then the job will be continued by spawning identical pods with the similar functionality. The end result of the processing by these pods will be written back to a persistent storage like Amazon EBS or Amazon S3. This data is available again for post processing.

The master_user_data.sh calls the master_script.sh which creates a apache web server to expose the master via REST endpoint. It also executes kubeadm_ec2_master.sh script whose output kubeadm token and discovery-ca-cert-hash are available to each and every worker node. The script kubeadm_join.sh and slave_script.sh are pulled from the master node by every worker node in kubernetes cluster. The execution of these two scripts, authenticate the two nodes and add them to the cluster.

The kubeadm_ec2_master.sh is the script with important installation details. It sets up kubernetes by installing utilities and the main components namely kubelet kubeadm kubectl. We choose calico as the pod network and provide its CIDR block as the command line input. It terminates by providing the kubeadm join token and ca-cert-hash key. These two are copied to the directory to host them as apache server. The nodes that join use this information. It also installs docker community edition for running the containers within pods. We also set the KUBECONFIG environment variable to admin.conf which allows querying the master node with information about the pods using commands like kubectl get pod or kubectl get node/deployment etc.

Master isolation is the recommended way to setup the cluster thus we do not allow the pods to be scheduled on the master node. By executing this $ *kubectl taint nodes --all node-role.kubernetes.io/master-* command we allow the pods to be scheduled on the master node for testing purpose.

Kubernetes Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage applications running in the cluster and troubleshoot them, as well as manage the cluster itself. We install and run dashboard as a pod in kubernetes and expose it as a service. This service type is NodePort. This helps in accessing the cluster through a port on every working node in the cluster including the master. It gives web ui to create jobs, pods, deployments etc by injecting the YAML file. [6]

B. High Level Architecture

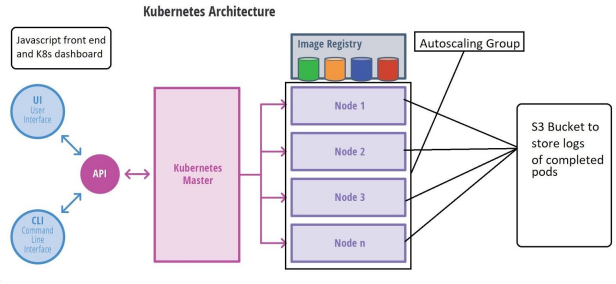

FIGURE 1: KUBERNETES ARCHITECTURE

The User makes a call to the Kubernetes Master API Server which in turn resolves it to the appropriate pods. The remote image registry is used to store docker images for the docker containers within these pods. These images can be directly pulled or cached in the EC2 instances. Each node represents an EC2 instance encapsulating the pods for front end or machine learning containers.

The Job API in kubernetes creates one or more pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the job tracks the successful completions. When a specified number of successful completions is reached, the job itself is complete. .

We use jobs to run our machine learning algorithm within the cluster. Each job has different hyperparameters and learning algorithm that run parallely. Upon node failure we restart the job from the state in which it was terminated. The completion of the jobs produces a log file which can be studied to get the desired output.

We tested this job API by running the simple Pi job which prints the value of pi to 2000 digits using a perl container and then looking into pods for logs and pushing it to S3.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: deepoptimization1
spec:
  template:
    spec:
      containers:
      - name: optml
        image: abhishekparekh1/deepoptml:deadpool
        command: ["python3","/app/EC2_Kubernetes/ML_Algorithms/ANN/ANN_TF_Driver.py","--DEBUG 0","--learning_rate 0.001",--network_size 2]
      restartPolicy: Never
  backoffLimit: 1
```

FIGURE 6: SAMPLE JOB YAML

Figure 2 shows what a sample YAML job looks like for the Kubernetes cluster to run. This acts as a setup for the container in the worker nodes and contains the specific commands what the user wants to run. Kubernetes knows to setup the container with the specified name and image file in the registry. The command line runs the driver file, in our case with Python3 and with the specified hyperparameters. The restart policy used is that upon completion, the job is never to be run again because the error is already found for the specified configuration.

A separate YAML script is needed to be generated for each configuration because a node will only run the single job before considering itself complete. This allows for the balancing the workload over multiple containers by reducing the jobs to single run lifespan. In our system we do not automatically generate the YAML files which can be tricky for users who do not know how the Kubernetes cluster and image is configured to run. Automating this process will preferred for usability, but we will expand upon this in future work.

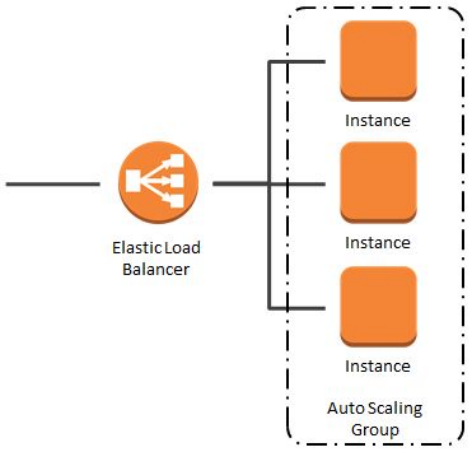

FIGURE 3: AWS AUTOSCALING GROUP

Docker is a virtualization software used to replicate an operating system image and setup the runtime environment. This is used by the Kubernetes cluster to setup the worker nodes. It allows for fast and easy mounting of the containerized algorithm avoiding the setup time to setup each node. Docker has a cloud registry for images that Kubernetes can pull from to setup the environment for each algorithm to run. This means that our system has the capabilities to use many different programming languages to develop an algorithm allowing for much more support.

There is still the issue of using licensed software languages such as MATLAB. In the case of MATLAB, a runtime environment installer is available from the company, but the libraries and the driver and function files must be pre-compiled to guarantee copyright. Additionally, the operating system used in MATLAB compilation process must match the operating system of the Docker container. We attempted to implement a Kernel Least Mean Square algorithm in MATLAB, but we our MATLAB installation was located on a Windows Server. Since Docker does not have native windows support there were errors when trying to run the pre-compiled executables. Adding a Wine directory creates another hassel, but it is probably the only way to avoid this debacle.[7]

Amazon provides an easy to use container service called Elastic Container Service for Kubernetes, also known as EKS. Though this would make the project substantially easier to just let Amazon control the cluster, we found that this does not meet the project goals that we set out for this project. There are a countless number of public and private clouds that need support, but giving AWS sole control on how to run the cluster does not promote cross platform usability. OptML will ideally be able to run as bare metal platform for any different cloud interface when the cluster is set up. Drawing resources from just one single source will stunt the growth that OptML is capable of reaching.

B. Features of Microservices Application

- ***Autoscaling***: Autoscaling of the AWS EC2 instances is supported by the Amazon Autoscaling groups. Kubernetes has it3333s own horizontal pod autoscaling API that scales the docker containers within the pods as per their CPU Utilization.
- ***Rolling Updates***: Deployment Controllers and Replication Controller in Kubernetes are responsible for rolling updates. Each pod can be updated one by one while serving existing queries or processing jobs.
- ***Container Orchestration***: AWS provides CloudFormation for essentially orchestrating the EC2 instances within VPC. Kubernetes also has API to create a scripted infrastructure that can later be used to deploy the cluster.
- ***Load Balancing:*** AWS Elastic Load Balancer is responsible for balancing the load within the EC2 instances. Kubernetes also has load balancer type service for the same.
- ***High Availability:*** EC2 instances are deployed in various availability zones that are geographically located such that electrical outage may not affect the application.
- ***Fault Tolerance:*** Amazon SQS can be used to pass messages that EC2 instances poll. Also S3 provides redundancy against loss of data.
- ***Private Image Registry:*** The Docker images created for running the application will be stored in secure external image registry that can be used to spawn up the infrastructure whenever necessary.

IV. Evaluation

A. Experimental Setting

To test our application, we have created two learning algorithms of LMS and ANN. They are made in Python. They will be used with a driver script to find the combination of hyperparameters with the least error. The driver file calls the machine learning functions and oversees the individual configurations of machine learning model. The function file holds the defines the functions needed to initialize the architecture, generate the data, train the model, and obtain the test error. Additionally, the function file may contain the auxiliary functions if the machine learning algorithm requires it. For instance, ANN contains the Sigmoid and rectifier linear unit (ReLU). The function file must be generic and take input of the hyperparameters going to be tested and their values. This allows for the workload to be distributed among many worker nodes, which brings the inherent speedup to the optimization process. Once the single configuration tested is completeed, the pod will die out and write the error found to an S3 container for manual analysis by the user.

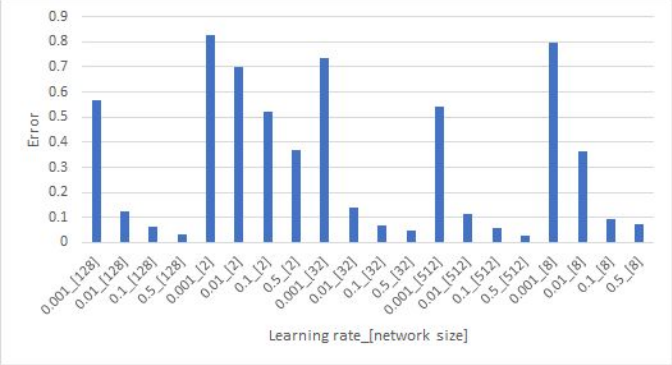

FIGURE 4: Error versus hyper-parameter configuration using backpropagation algorithm for the ANN

Connecting the Kubernetes Cluster we create an EC2 instance and then load the data needed to run the jobs onto the instance. The keys to access the Kubernetes must also be included in the EC2 instance otherwise the connection to the cluster will be refused. As stated earlier, this is used as a security measure to protect the Kubernetes cluster because of bad actors. We also create a Python Setup file to deploy everything onto the Kubernetes Master and somewhat automate the setup processing instead of consistently updating a github repository to see how the results are changed, when different configurations are issued. This is also a further step toward our goal of creating a simplified tool to find the optimal hyperparameters.

The Python file pushes the jobs on the Kubernetes cluster. The python file initially SSH's into the Kubernetes Master node using the pre-shared keys. The file then deploys the machine learning algorithm, hyperparameter file, and the YAML job file into to correct location. The YAML file is read into the Kubernetes job scheduler to begin execution of the workload. The Kubernetes cluster then creates nodes to push the jobs onto and governs over the work by managing the specified containers and their respective applications. If the workload is not finished because of an offset workload, additional pods are added to rebalance the work.

B. Experimental Metrics

The purpose of our project is the creation of a cloud-based hyperparameter tuning tool to ease and speed up the development process for machine learning models. We will compare our speed of finding the optimal hyperparameters to that of a standard CPU. It is expected to function faster than a standalone PC because of the parallel nature of grid search. For future work, we expect to improve the tuning approach used and perform a comparison with other cloud-based tuning devices, such as Google Cloud and Sage Maker, to see if there is a speedup and evaluate where the design needs to improved.

Unfortunately, the Python file could not be used to gather the metrics for length of time to run the simulation because all it does is push the work onto the Kubernetes Master node. The job completion can be tracked from the GUI of Kubernetes and within the master itself. The data was calcultated from the Kubernetes Master node since it knows when the jobs terminated. Since this is a distributed operation, the longest job would be the limiting factor to the total speedup. That is not entirely the case however with the overhead involved in a Kubernetes cluster. To determine the time on a single machine, the same job were run concurrently on a single machine.

V. Experimental Results

Figure 4 and Figure 5 show the error versus the hyper-parameter configuration for the backpropagation and LMS algorithms, respectively. The results were gathered over multiple configurations to show that our methodology can determine the hyper-parameter configuration that yields the best error value that can obtained using a given training and testing data set.

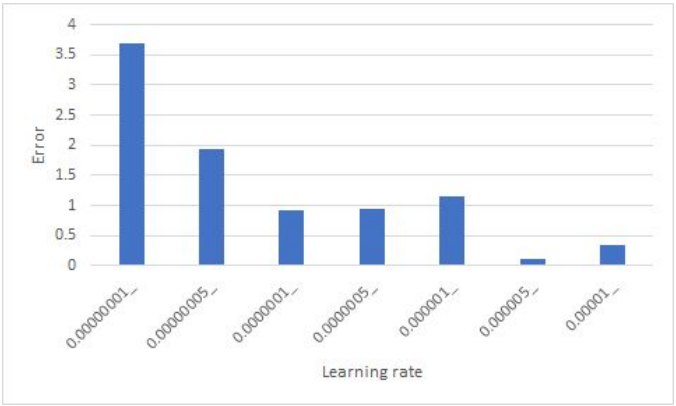

FIGURE 5: Error versus hyper-parameter configuration using LMS algorithm

The hyper-parameters that we varied across different runs of the algorithms were the learning rate and network size for the backpropagation algorithm and the learning rate for the LMS algorithm. For our methodology, the minimization of the error is desired and our approach finds the minimum error among many different hyper-parameter configurations. In future work, we plan to extend this idea to allow flexibility in minimizing or maximizing different metrics as is appropriate for different user cases. For example, one user may want to minimize the error obtained from the algorithm on the test set or another user may want to maximize the accuracy obtained from the algorithm on the test set. As can be seen from both Figure 4 and Figure 5, the hyper-parameter configuration the minimizes the obtained error for both the backpropagation algorithm and the LMS algorithm are easily seen and determined from the figures. For the backpropagation algorithm, the learning rate is 0.5 and the network size is a single hidden layer with 512 processing elements. For the LMS algorithm, the learning rate that minimizes the obtained error on the testing data set is 0.000005.

In addition to validating that are approach can calculate the errors over multiple hyper-parameter configurations and determine the hyper-parameter configuration that yields the minimum error, we also wish to evaluate the speedup obtained by performing the calculations of the errors in parallel in contrast to approaches that perform the evaluation of the hyper-parameters on the performance of the algorithms serially. Since our approach performs a grid search over the entire hyper-parameter configuration space, all of the execution runs of the algorithms using a different hyper-parameter configuration are independent to one another and thus we expect a linear speedup as we increase the number hyper-parameter and hyper-parameter values that we wish to test. Figure 6 and Figure 7 show the speed obtained using our parallel approach in contrast to a serial execution of the algorithms for the backpropagation algorithm and the LMS algorithms, respectively. For the ANN algorithm, we tested 20 different hyper-parameter configuration value pairs for the learning rate and the network size. The serial execution of the algorithm takes 1126.58 seconds to calculate the errors for the algorithm using all of the different hyper-parameter value pairs. The parallel execution of all of the algorithm instances with different hyper-parameter value configurations ran in 60.89 seconds, resulting in a speedup of about 18.5. For the LMS algorithm, we tested 16 different hyper-parameter configuration values for the learning rate. The serial execution of the algorithm takes 34.6 seconds to calculate the errors for the algorithm using all of the different hyper-parameter values. The parallel execution of all of the algorithm instances with different hyper-parameter value configurations ran in 2.60 seconds, resulting in a speedup of about 13.3 seconds.

In theory, we would expect the ideal speedup using a parallel execution of the algorithm instances compared to the serial execution of the same instances to be about 20 and 16 using the ANN and LMS algorithms, since each instance with different hyper-parameter values executed in about the same amount of time for each execution. The speedups that we obtained confirmed our intuition about the speedup that we believed we would achieve. Each run of the algorithm was scheduled as a yaml job to one of the machines in the Kubernetes cluster. As a result, assigning the jobs, setting up the runtime environment, and starting the job on the cluster acted as our overhead and explains why the speedup obtained is not exactly 20 or 16 for the ANN and LMS algorithms, respectively. Despite this, the speedups we obtained are still close to the ideal speedup and confirm the validity of our methodology to improve on the time required to determine a good set of hyper-parameters for a machine learning model to use for any arbitrary application.

We would like to note that our methodology is not limited to using only ANN and LMS as the machine learning algorithms. Any machine learning algorithm can be used as long as they are written using a driver file and a function file as explained in the experimental setup. For different algorithms, the user must determine the hyper-parameters of interest. In addition, the hyper-parameters that we chose for the ANN and LMS algorithms are not limited to learning rate and network size for the ANN model or only learning rate for the LMS model. More hyper-parameters can be searched for each of these algorithms. For example, the ANN model can also incorporate momentum (with the associated momentum factor) and the LMS model could incorporate filter size (which determine the number of weight parameters that have to be trained).

	Time		
	Serial	Parallel	Speedup
ANN	1126.58	60.89622	18.5

FIGURE 6: Speedup of our approach over a serial execution of the backpropagation algorithm

	Time		
	Serial	Parallel	Speedup
LMS	34.60083	2.601566	13.3

FIGURE 7: Speedup of our approach over a serial execution of the LMS algorithm

VI. Future Work

This is a working prototype so there are countless improvements needed to be done before this can be brought to released to the public. The immediate goal is the improvement of job creation. The current approach is to input a configuration file with all the hyperparameters and their values. The user then has to follow the organization of creating the job in Kubernetes. This YAML file can be

monotonous to create even when making the script because the user has to understand where the files are stored from the Dockerfile. This leads into the next quality-of-life improvement, which is the user interface for the web application. Currently, we are using a command-line interface to interact with the Kubernetes cluster through an AWS client. The client must have access to Kubernetes cluster credentials with the pre-shared keys, which is far from scalable. Instead, we need to create a standard GUI with the capabilities to load the model, test data, and set the configurations that the user wishes to run. Automating this process will save users hours of setting up and debugging.

As we were using the Kubernetes cluster we noticed a bottleneck with handling the requests as the job count increased. The single master had difficulties keep up so adding an asynchronous queue service to handle requests and ease the workload for the master would prove beneficial as user base grows. Afterward, the system should be tested with a larger cluster to observe the effectiveness and scalability of this design. Since our tests were rather small for the what the system should theoretically be able to handle, the speedup needs to be determined on labor-intensive tasks and larger amounts of data.

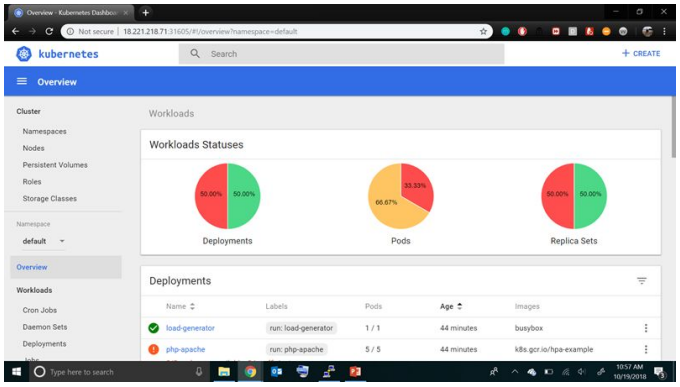

FIGURE 7: KUBERNETES GUI

We believe that adding support for other popular frameworks such as TensorFlow, PyTorch, Keras, and Caffe. Doing so will allow us to include a plethora of API's to improve user experience and ease the development burden, which is the goal of our project. We also added earlier in the paper the rudimentary nature of grid search. There are a plethora of optimization techniques used in current optimization competitors. We want to test the common techniques such as Bayesian, random search, and evolutionary approaches. Doing so will allow us to leverage other works and determine possible hybrid solutions to the optimization field.

The current installation only supports the Amazon Web Services Elastic Compute Cloud Instances. The launch configuration for the auto scaling groups that has been specified in the start is permanently associated. Though we allocate elastic IP addresses to the master VM, we should dedicate a load balancer to a set of instances which are logically grouped together to form the master node. The ETCD and further communication remain the scope of future research. The state of the master nodes need to be maintained before we recover from failure. This will ensure a fault tolerant mechanism and support failure recovery. If for some reason like power failure or disaster, the highly available Kubernetes Control Plane managed by the group of instances that act as the master significantly improves the success factor of our application.

The background VMs are T2.Micro instances. These usually have 1 vCPU and 1 GiB RAM. This might be a bottleneck. If the cost utilization is within a desired budget, we can improve performance by vertically scaling each of the EC2 instance and dividing the job on the basis of resource requirements. The scheduling algorithm is FIFO in the current architecture but better scheduling algorithms. The different types of instances which are memory optimized or Compute optimized rather than just general purpose instances should be used for specific use cases. Creating deployment templates for such hybrid autoscaling environments is a bit cumbersome. Though automated tools and scripts that make the DevOps part easier are available which can ensure this happens fairly efficiently.

Also, the project is subject to sustainability issues. The technologies used in our system are constantly being updated, which breaks our underlying infrastructure. For instance, when we first created the Kubernetes cluster, the AWS scaling group needed only a single CPU to function together, but during the latest rollout of updates, an instance with two CPUs must be within the system. Being subject to these constant updates might eventually break the system or our wallets if we have to match the technology. We are then either forced to find another cluster management software or create a washed down version of what is already available. Though we may lose the nice GUI and APIs, we gain plenty is longevity for the system and customizing how our system will function.

Through our studies, we have also determined an area to expand where our application can be useful. A current trend is the use of neural networks to help other neural networks and algorithms learn. This is called architecture search, which is treated as an optimization technique. Picking the right hyperparameters is only one part to improving the machine learning experience. Neural Architecture Search compliments our project significantly so exploring how to improve the individual architecture design and organization will allow this project to reach new heights and meet its desired potential [2][4].

VII. Conclusion

Finding the optimal hyperparameters for each machine learning algorithm can be a matter of life and death for a project. Wrong hyperparameters can lead to a substantial error rate that makes the algorithm underfit the dataset. To make things worse there are a large amount of unique algorithms to generate models with an equally large amount of hyperparameter categories the affect the model. For our project, we created a hyperparameter optimization fine tuning tool that can be used in the development of improving machine learning algorithms.

OptML wishes to alleviate researchers troubles by speeding up the development process. By using a simple grid search schema, OptML provides a infrastructure that distributes the jobs over multiple work nodes. All that is required to operate are the functions to train and calculate the error of a model, the model itself and the various hyperparameter to test. The Kubernetes cluster does all the rest allowing little room to worry. The barebones framework that OptML saves researchers time from having to setup a distributed system.

The hyperparameter optimization workspace is fairly saturated, with AWS Sagemaker and Google Cloud offering tuners, but independent applications such as Cloud.ml seek to bring support directly to the consumers machine and ease development. We created a Kubernetes Cluster that auto-scales to match consumer demand. The cluster completes the jobs issued by a client AWS instance and writes the error rate of each job to an S3 bucket where a user can observe the jobs output. A GUI provided by Kubernetes is used see how the cluster is operating is functioning such as whether a job failed and or how much load is on the cluster. The GUI, which is displayed in Figure 7, is used essentially for debugging purposes.

Although this system is rudimentary and very early in development, we believe the OptML will fill a basic, but necessary role in developing future machine learning algorithmic models. Currently, our system works as a barebones framework, but there needs to improvements in using our Kubernetes cluster. Other than minor quality-of-life issues, OptML has many areas to explore and integrate with for developing models.

Our results have shown the grid search approach to be embarrassingly parallel. LMS, which is a simple algorithm, had reached a thirteen times speedup. ANN had an eighteen times speedup. With dozens of algorithmic variations in model architecture, we understand that resources must be allocated to finding the right hyperparameters and we firmly believe OptML will have much more to show and offer in the near future.

VIII. Acknowledgements

We would like to thank our professor Andy Li Ph.D for guiding us on this project. He guided us on where to explore and provided advice when unseen problems occurred such as with funding this endeavor. Li was a great at helping us navigate the limitless features of cloud infrastructure and popular frameworks.

IX. References

[1] Leonard, James, and M. A. Kramer. "Improvement of the backpropagation algorithm for training neural networks." Computers & Chemical Engineering 14.3 (1990): 337-341.

[2] Liu, Hanxiao, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search." *arXiv preprint arXiv:1806.09055* (2018).

[3] N. Sireesha, K. Chithra and T. Sudhakar, "Adaptive filtering based on least mean square algorithm," *2013 Ocean Electronics (SYMPOL)*, Kochi, 2013, pp. 42-48.

[4] Zoph, Barret & Vasudevan, Vijay & Shlens, Jonathon & V. Le, Quoc. (2017). Learning Transferable Architectures for Scalable Image Recognition.

[5] Hinz, Navarro-Guerrero, Magg, & Wermter. "Speeding up the Hyperparameter Optimization of Deep Convolutional Neural Networks" *International Journal of Computational Intelligence and Applications*. Vol. 17 No 2. World Scientific. 2018

[6] Bernstein, David. "Containers and Cloud: from LXC to Docker to Kubernetes" *IEEE Cloud Computing.* Vol. 1, Issue 3, pp. 81-84, 2014

[7] Burns, Beda, & Hightower. "Kubernetes: Up and Running: Dive Into the Future of Infrastructure" O'Reilly Media. California . 2017.

[8] Cloud: Amazon Elastic Computing (2011) Amazon web services. Retrieve 9 Nov 2011 faws.amazon.com/about-aws/whats-new/2011

[9] Wittig & Wittig. "Amazon Web Services in Action" Manning Publications. 1st Edition 2015

[10] Louridas and Ebert "Machine Learning" *IEEE Software*, vol 33. no 5. pp 110-115 2016.