2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE) | 978-1-7281-9146-1/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICAICE51518.2020.00102

Towards MLOps: A Case Study of ML Pipeline Platform

Yue Zhou
Key Lab. of Parallel and Distributed Computing Lab.
College of Computer, National University of Defence Technology
ChangSha, China
zhouyue18@nudt.edu.cn

Yue Yu
Key Lab. of Parallel and Distributed Computing Lab.
College of Computer, National University of Defence Technology
ChangSha, China
yuyue@nudt.edu.cn

Bo Ding
Key Lab. of Parallel and Distributed Computing Lab.
College of Computer, National University of Defence Technology
ChangSha, China
dingbo@aliyun.com

***Abstract*—The development and deployment of machine learning (ML) applications differ significantly from traditional applications in many ways, which have led to an increasing need for efficient and reliable production of ML applications and supported infrastructures. Though platforms such as TensorFlow Extended (TFX), ModelOps, and Kubeflow have provided end-to-end lifecycle management for ML applications by orchestrating its phases into multistep ML pipelines, their performance is still uncertain. To address this, we built a functional ML platform with DevOps capability from existing continuous integration (CI) or continuous delivery (CD) tools and Kubeflow, constructed and ran ML pipelines to train models with different layers and hyperparameters while time and computing resources consumed were recorded. On this basis, we analyzed the time and resource consumption of each step in the ML pipeline, explored the consumption concerning the ML platform and computational models, and proposed potential performance bottlenecks such as GPU utilization. Our work provides a valuable reference for ML pipeline platform construction in practice.**

Keywords-MLOps; machine learning; end-to-end platform; DevOps; continuous training

I. INTRODUCTION

Advances in big data and deep learning have promoted the wide adoption of artificial intelligence and machine learning (ML) in various fields [8, 27]. While ML models are embedded in applications serving as face detection or voice recognition modules, the development and deployment of ML applications become more difficult and complex than traditional applications. For example, the lifecycle of ML applications differs significantly from traditional applications in feedback loops [26], the monitoring and evaluation of a model may loop back to any previous step of development, such as data preprocessing or model training, which will change the model's behavior. Besides, the massive datasets for model training and testing also bring unignorable challenges to data storage and management [2, 11]. The differences and difficulties introduced by computational models have led to an increasing need for efficient and reliable production of ML applications as well as supporting infrastructures.

Infrastructures and platforms for end-to-end lifecycle management of ML applications contain multiple parts of work due to the multiple process phases and artifacts. The lifecycle phases generally include several parts, such as data processing, data validation, feature extraction, model design, model training, model evaluation, application development, quality checks, application deployment, and application maintenance [4]. During these phases, datasets, models, application configurations, and other artifacts are generated [20]. Kinds of utilities are required for managing these phases and artifacts: ML platforms need to provide voluminous data storage and even version control management for datasets and extracted features; in form of CPU, GPU, and TPU, considerable computing resources are demanded in the model training phase [23, 35, 36]; because well-trained models partly depend on and develop with datasets, managing and versioning models along with their datasets benefit system traceability, as what Data Version Control (DVC) does [21]; model performance decay happens over time for many reasons, such as changes in data distribution of the environment [9], and results in the requirement of model retraining mechanism; deployed monitoring and logging systems are needed for collecting model behavior information that will feedback to previous development phases as a basis for model optimization, and so on. Cloud shows its natural advantages in this domain [22, 32, 41]. By offering bulk storage volumes and GPU-backed virtual machines (VMs), cloud providers like Amazon, Google, and Alibaba Cloud provide cloud storage and cloud computing services for ML application developers through a pay-as-you-go pricing model.

Some platforms, such as TensorFlow Extended (TFX), ModelOps, and Kubeflow [4, 6, 20], have provided the solutions for the end-to-end lifecycle management of ML applications by orchestrating its phases into multistep ML pipelines. Several key components and utilities are offered to manage data storage, access control, pipeline execution, job scheduling, and performance monitoring. ModelOps and Kubeflow also provide pipeline configuration or Software Development Kit (SDK) for developers to describe the ML workflows. These pipeline files may define several steps in

978-1-7281-9146-1/20/$31.00 ©2020 IEEE
DOI 10.1109/ICAICE51518.2020.00102

corresponding with the lifecycle phases, such as data analyzing, model training, model evaluation, and model deployment. ML pipelines not only clarify the workflows but also benefit their automation by leveraging the concepts from DevOps [3]. The advantages of continuous integration (CI) and continuous delivery (CD) [13, 19], such as increasing developers' productivity and quickly delivering codes, may also be applied to the iterative development and deployment of ML applications. In particular, continuous training (CT) means continuous training or retraining models when some specific events happen, such as the availability of new training data and model performance degradation [9]. Through applying DevOps principles to the lifecycle management of ML applications, concepts like MLOps are created to describe the combination of ML system development (Dev) and operation (Ops).

Beyond ML codes, these frameworks and platforms have provided functional components and utilities to help avoid ongoing maintenance costs brought by hidden technical debt in ML systems [33]. With the aid of cloud and virtualization, the lifecycle management of ML applications benefits from ML pipeline platforms in efficiency and reliability. However, the performance of these platforms is still uncertain, such as how much computing resources they consume and how much time it takes to train a model. Therefore, several experiments are set to evaluate this point by building ML pipelines and training models on Kubeflow with CI/CD tools [29]. Some metrics were used to conduct the consumption information collection and analysis on seven deep learning neural networks with different network layers and parameters, such as GPU and memory utilization. The main contributions of this research are summarized as follows:

- We verified the feasibility of building an effective ML pipeline with CI/CD abilities on machines with specific hardware configuration.
- We evaluated the performance of ML pipeline platforms represented by Kubeflow, through a set of control experiments on seven different models according to GPU utilization and other metrics.
- We explored the time and resource consumption concerning the ML platform and computational models.

The rest of the paper is organized as follows: Section 2 outlines some existing work that related to this study, Section 3 introduces the method of evaluating ML platform performance taking Kubeflow as an example, Section 4 details the experiment settings, including the platform composition, multistep ML pipeline construction, and metrics used, Section 5 discusses and analyzes our experimental results, Section 7 concludes our work.

II. Related Work

As the number of software that employs computational models has surged, ML applications are evolving from ML programs into developmental ML systems. More than applying software engineering principles to the lifecycle management of ML applications and clarifying the complexity of maintaining ML systems with different feedback loops, such a conceptual leap also brings out the challenges of providing vast and functional infrastructures and platforms that support the development and deployment of ML applications.

A. ML Pipeline Platforms

Developing and deploying ML applications is more than collecting data, training models, and getting predictions; after all, connecting these parts while ignoring maintenance may lead to considerable technical debt [33]. Towards efficient and reliable production of ML applications, many use cases and challenges need to be considered. For example, machine learning has been introduced into areas with high safety requirements, such as driverless technology and paramedical diagnostics, the quality and privacy need to be assured before application serving, which requires testing and validation on both datasets and trained models. Building the workflow from data preprocessing to application runtime monitoring can be time-consuming and error-prone, automating this process can help users focus on the ML application development. Besides, when new training data arrives or model performance degradation is detected, the computational models need to be retrained and reserved, which requires effective feedback loops from the monitoring system to previous phases.

ML platforms such as TFX [4] and ModelOps [20] address the above concerns. They provide end-to-end lifecycle management for ML applications and systems by offering a set of key components that are separately responsible for tasks such as data preprocessing, model training, model evaluation, and model serving. Based on the pluggable and customizable components, these platforms intend to provide a generic solution for multiple development scenarios that need to accomplish different ML tasks. Through configuring and linking these components, the ML workflows can be orchestrated into ML pipelines that are execution-supported on these platforms. While there is a trend of training ML models on the cloud with abundant GPU-backed VMs and containers, these platforms are able to carry out ML pipelines in hybrid environments. Besides the production-level reliability and scalability, continuous training abilities are also provided by these two platforms to adapt to evolving data or increase change frequency.

B. MLOps

The ML applications and systems differ from traditional software in many ways which makes it necessary to ascertain custom DevOps for ML features. Compared to conventional software systems, ML systems include additional model artifacts, contain more data process steps, and deal with more complex relationships between these three artifacts. Due to the introduction of computational models along with their experimental nature, traceability and reproducibility of the training results are required [31], which can benefit from versioning models together with codes and datasets. Used as training and testing datasets, the higher quality requirement of data in ML systems needs more data process steps in ML workflows, such as data cleaning, data analyzing, data validation, feature extraction, and so on. The performance monitoring feedback loops and new data availability may lead to the optimizations of codes and models, which bring out new versions of these three artifacts in the iterations of ML systems.

MLOps is an engineering practice in the ML field that applies DevOps principles to ML systems for unifying the ML system development and operation [9]. From the CI perspective, more test procedures are introduced in addition to classical unit and integration tests, such as data and model validations. From the CD perspective, processed datasets and trained models are delivered from data and deep learning scientists to ML system engineers automatically and continuously. From the CT perspective, the arrival of new data and the degradation of model performance require to trigger model retraining or improve model performance through online methods [40].

C. Kubeflow

Kubernetes is an open-source cluster manager to arrange Docker containers among a mass of virtual or physical machines, which is increasingly adopted in cloud computing [5, 16, 28]. Besides creating, monitoring and scheduling containers to execute programs and run software, Kubernetes also supports the ability to build custom resources (CRs) according to declarative specifications. With the increasing usage in more and more AI/ML companies and groups, challenges of developing and deploying ML applications in Kubernetes clusters have emerged. For example, various components and mixing vendors are involved in building ML systems, which introduces many sophisticated configurations to connecting and managing these services [10]. In many cases, these configurations are bound to particular clusters, which may impair the mobility of moving ML applications from local clusters to the cloud or from development environments to production environments.

Based on Kubernetes, Kubeflow is an open-source pipelined ML platform created to address the above concerns. It provides end-to-end lifecycle management of ML applications by taking advantage of TFX components, such as data validation, model evaluation, and model serving. Because of the ability to be deployed in kinds of Kubernetes clusters, Kubeflow enables ML workflows to work in any local or cloud cluster where Kubeflow has been installed, which benefits the simplification of setups and portability on diverse infrastructures. Beyond that, a TensorFlow CR is defined to use CPUs or GPUs and adjusted to the size of a cluster [1, 10].

Kubeflow Pipelines is the component in Kubeflow that manages and orchestrates the end-to-end ML workflows. Steps in an ML workflow are wrapped as components in Kubeflow Pipelines, such as data preprocessing, data transformation, and model training. After being packaged as Docker images, these components are reusable across pipelines in a containerized way. Consisting of a set of components as well as their input/output relationships, the pipelines are defined in Python files through the SDK, which will be compiled into static YAML-format configurations for Kubeflow to process [25]. These pipeline files can be packaged and uploaded to the Kubeflow platform in CI/CD procedures when the files change in the code repository, while the running of pipelines can be triggered on schedule or through the message sent by cloud storage when new data arrives.

III. METHOD

To explore and estimate the performance of ML pipelines on a specific platform, we need to build an ML platform from existing work first. Meanwhile, we want the platform to have continuous training ability in the form of retraining models automatically when specific events happen in a relatively frequent way, such as ML algorithm codes change. Finally, we construct the whole effective ML pipeline that can be executed on this platform.

A. Platform Composition

In our ML platform with DevOps capability framework design, we need an existing ML platform like Kubeflow that can accomplish ML tasks such as model training, a git service to store and manage code repositories, and a CI/CD tool that can clone git repositories and set ML tasks to run. We finally chose Kubeflow, Gitea, and Drone as the three parts in the implementation, the framework of our ML pipeline platform is shown in Figure 1.

Improved from Kubernetes, Kubeflow is a cluster manager increasingly adopted in cloud computing, which can help ML applications and systems move to cloud environments. Kubeflow is supported by TFX and accessible to its functional TensorFlow-based components that can be assembled expandable. Kubeflow also provides an SDK to describe ML workflows in Python pipeline files by including these components and their interactions. So we created a single-node local Kubernetes cluster and then deployed KubeFlow in it. We also prepared SDK in the development environment to write and compile Kubeflow pipelines that include steps such as data processing and model training.

Gitea [14] is a community managed lightweight self-hosted Git service that supports Git Large File Storage (LFS) service [7], having the potential to version both algorithm codes and training datasets, which can be further used to provide model traceability. Drone [12] is a container-native open-source platform that supports CI and CD for Gitea. Through carrying out each step in a container, Drone uses YAML-format files to define, build and execute workflows, which can be used to clone git repositories, compile the Kubeflow pipeline files, and set them to run. Based on open-source software, we built the git service platform to version codes and the CI/CD tool to run CI/CD pipelines automatically. In this step, we configured Drone to connect Gitea and authorized it to access git resources. When git pushes and other git events happen in repositories with webhooks added, a message along with git commit

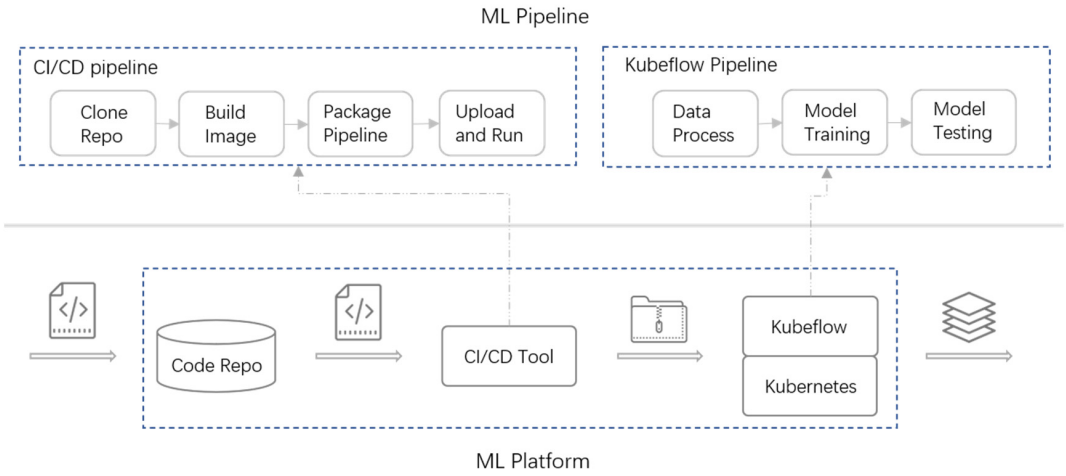

Figure 1. Overview of ML Pipeline Platform

information will be sent to trigger a CI/CD execution. After the message has been received, Drone will carry out the subtasks defined in CI/CD pipeline configurations step by step. During the completion of these subtasks, Dockerfiles are used to build Docker images that include Kubeflow pipeline components, while Shell scripts are used to compile and upload Kubeflow pipeline definition files.

B. Pipeline Construction

After Kubeflow, Gitea and Drone have been deployed and configured to connect each other, we need to design and construct our ML pipeline that can continuously train models when specific events happen, such as code change submissions in our case. From this perspective, the ML pipeline can be divided into two functional parts, one is the completion of ML tasks and the other is the compilation of ML task definitions.

As shown in Figure 1, we set up two kinds of pipelines on the ML platform: the CI/CD pipeline used for packaging and uploading Kubeflow pipelines automatically, and the Kubeflow pipeline used for model training and other processes.

In the CI/CD pipeline, upon receipt of the messages that represent git repository changes, the CI/CD tool first needs to clone repositories that store the codes of computational models and Kubeflow pipeline definition files. Second, based on the Dockerfiles in repositories, Docker images that contain pipeline component programs are built and uploaded to the Docker repository. Then, referencing these components and defining their in-out interactions, the pipeline files are compiled into YAML-format configurations and uploaded to Kubeflow. Finally, the CI/CD tool starts the execution of Kubeflow pipelines. These above steps are described in the CI/CD pipeline configuration files stored in git repositories as well.

Kubeflow pipelines are carried out by sequentially executing components. During the execution of a component, the Docker image is pulled from the Docker repository first. When a Docker container is created from that image, the component programs built inside start to run and finish ML tasks such as datasets downloading, data processing, model training, and model testing.

IV. Experiment Setup

This section elaborates on the runtime environment of our ML platform and the experiments set to train different models in ML pipelines whose time and resource consumption were collected according to some metrics.

A. Runtime Environment

Our platform was built on a physical machine with an Intel I7-8700K CPU, a GeForce GTX 1080 graphics card, 32GB memory, and 8GB video memory. We first deployed a single-node local Kubernetes cluster on this machine and then installed Kubeflow and Kubeflow Pipeline in it. We deployed Gitea, Drone, Docker, and Docker repository in the same machine. We also configured the ML algorithm programs in the containers to use the GPU to train models.

B. Models and Metrics

Model-involved processes are the most important parts of ML pipelines. To explore the time and resource consumption of the ML platform during pipeline executions, we selected several deep learning neural networks with different layers and parameters, packaged them into Docker images, and took them as model training and model testing components in different Kubeflow pipelines.

Table 1 lists the selected deep learning neural networks trained in our ML pipeline experiments: MobileNet [17], ShuffleNet [42], ResNet [15], VGG [34], DenseNet [18], GoogLeNet [37] and Inception V3 [38]. These are approved networks that have a wide range of layers ranging from 16 to 121 and hyperparameters ranging from 1.0 to 34.0 million, whose implementations are sourced from the Github community [39]. With epoch set to 200, these models were trained on CIFAR-100 [24], a classic dataset that is approximately 161 MB in size.

Different ML pipelines were constructed to train different models. Finally, we ran each pipeline 7 times on our ML platform from code submission to model testing. To explore the performance of different steps in the ML pipeline, the time and resource consumption during executions were collected by step.

We respectively recorded the time consumption for the CI/CD pipeline and Kubeflow pipeline, in which the time elapsed of git repository cloning and other substeps in the CI/CD phase were separately recorded too. As for resources, we mainly recorded the consumption during Kubeflow pipeline executions, according to some metrics from Nvidia documentation, such as GPU utilization, memory utilization, and framebuffer memory used [30].

TABLE 1 INFORMATION ON MODELS

Model	Layers	Million Parameters
DenseNet-121	121	7.0
GoogLeNet	22	6.2
Inception V3	48	22.3
MobileNet	28	3.3
ResNet-18	18	11.2
ShuffleNet	50	1.0
VGG-16	16	34.0

V. RESULTS

In this section, we list the time and computing resources consumed in the ML pipeline executions and analyze their relationships with the trained models.

A. Time Consumption

The average time consumption of ML pipelines is listed in Table 2. Among the total time consumed, the Kubeflow pipeline accounts for more than 98.907%, while the CI/CD pipeline takes less than 1.093%. All CI/CD pipeline executions were completed within 0.46 to 0.857 minutes, which is less than 1 minute and can be timed in seconds. Compared to Kubeflow pipelines that were completed in 54.317 to 981.083 minutes, their elapsed time was very short.

TABLE 2 TIME CONSUMPTION OF ML PIPELINES

Model	CI/CD		Kubeflow		Total
	(min)	*(%)*	*(min)*	*(%)*	*(min)*
DenseNet-121	0.586	0.148	395.933	99.852	396.519
GoogLeNet	0.505	0.105	480.950	99.895	481.455
Inception V3	0.581	0.059	981.083	99.941	981.664
MobileNet	0.600	1.093	54.317	98.907	54.917
ResNet-18	0.857	0.679	125.450	99.321	126.307
ShuffleNet	0.603	0.547	109.633	99.453	110.236
VGG-16	0.460	0.245	187.667	99.755	188.127

The Kubeflow pipeline mainly performs model training and other tasks, such as Docker image pulling and dataset downloading, whose time cost is negligible since they were stored in the same machine with the ML platform. Though varying considerably between pipelines that train different neural networks, the time consumption is not related to the number of model layers or parameters, which we will discuss in the 5.3 section.

There is little difference in CI/CD pipeline execution time between different ML pipelines, which is because of the little difference in each CI/CD step. When cloning git repositories and uploading Kubeflow pipelines, the number and size of files that need to be downloaded and uploaded don't vary much; when building Docker images and packaging definition files, the complexity of Kubeflow pipelines and the number of pipeline components don't differ much either. Figure 2 shows the average time consumption of CI/CD steps separately calculated. Among the total time consumed by CI/CD pipeline execution, the Kubeflow pipeline packaging step accounts for the largest part, followed by the step that uploads files and sets Kubeflow pipelines to run. Each pipeline packaging step took about 10 to 21.571 seconds, which means it takes a relatively long time to compile the Kubeflow pipeline Python files into YAML-format using the SDK, while the repository cloning step took 1.714 to 3.429 seconds and the image building step took 2.714 to 4.714 seconds. Besides, uploading files to Kubeflow took more time than downloading files from git repositories, when it was completed in 8.571 to 15.286 seconds.

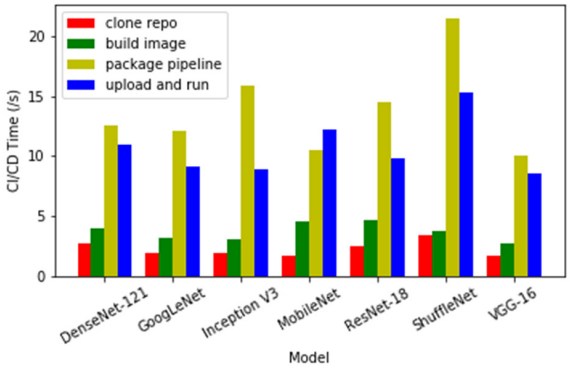

Figure 2. Time Consumption of CI/CD Pipelines

B. Resource Consumption

The computing resources are mainly needed in ML task accomplishing stages such as model training, so we recorded the utilization information during the executions of Kubeflow pipelines. As shown in Figure 3, the average GPU utilization rate is all between 89.756% and 98.3% with little distinction, even when training different models that vary greatly in layers from 16 to 121 and in parameters from 1 to 34 million. While memory utilization rate and framebuffer (FB) memory used vary with the computational models, from 47.424% to 72.554% and 963.245 to 6795.425 MiB respectively, and are both relatively low compared to GPU utilization. High utilization not only means being able to make full use of GPU but also means that programs will compete for GPU usage when executing multiple model training tasks simultaneously. This also brings a problem for ML platform providers, which is how to improve memory and framebuffer memory utilization when GPU usage reaches a bottleneck.

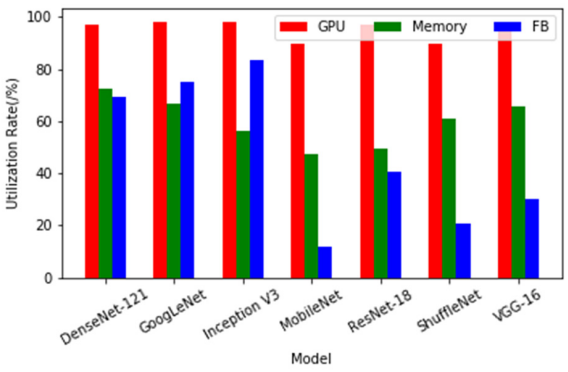

Figure 3. Resource Consumption of Kubeflow Pipelines

C. Time and Resource Consumption with Model Differences

Although varying considerably between ML pipelines that train different models, the time and resource consumption are not related to the number of model layers or parameters. As shown in Table 3, when sorted in ascending order, neither the time elapsed nor the framebuffer memory used increases with model layers or hyperparameters. These metrics have more to do with the design, construction, and implementation of the models themselves.

TABLE 3 TIME AND RESOURCE CONSUMPTION OF MODELS

(a) CONSUMPTION SORTED IN ASCENDING ORDER OF MODEL LAYERS

Model	Time	GPU	Memory	FB
	(min)	*(%)*	*(%)*	*(MiB)*
VGG-16	186.100	97.886	65.502	2463.999
ResNet-18	123.649	96.984	49.363	3316.116
GoogLeNet	479.569	98.300	66.846	6082.739
MobileNet	52.587	89.756	47.424	963.246
Inception V3	979.117	98.234	56.037	6795.426
ShuffleNet	108.310	89.774	61.237	1671.129
DenseNet-121	394.114	96.948	72.554	5615.866

(b) CONSUMPTION SORTED IN ASCENDING ORDER OF MODEL LAYERS

Model	Time	GPU	Memory	FB
	(min)	*(%)*	*(%)*	*(MiB)*
ShuffleNet	108.310	89.774	61.237	1671.129
MobileNet	52.587	89.756	47.424	963.246
GoogLeNe	479.569	98.300	66.846	6082.739
DenseNet-121	394.114	96.948	72.554	5615.866
ResNet-18	123.649	96.984	49.363	3316.116
Inception V3	979.117	98.234	56.037	6795.426
VGG-16	186.100	97.886	65.502	2463.999

VI. CONCLUSION

In this study, we deployed an ML platform with DevOps capability on a generally configured physical machine, which verified the feasibility of building functional ML pipelines from existing CI/CD tools and ML platforms such as Kubeflow, which can continuously retrain models when specific events happen. The whole ML pipeline can be finely divided into ML task pipeline that carries out ML tasks including data preprocessing and model training, and CI/CD pipeline that continuously triggers executions of ML task pipelines and deploys validated models. On the platform represented by Kubeflow, most of the time and resource consumption is generated by the ML task pipeline. The executions of CI/CD pipelines can be completed in seconds, with the most time spent compiling Kubeflow pipeline definition files into the platform-supported format. According to some metrics, the GPU utilization during the ML task pipeline executions is generally high though the models trained vary greatly in the number of layers and parameters, which may cause performance bottlenecks when training multiple models simultaneously. In addition, the time and resource consumption have more to do with the design and construction of models than ML platforms.

ACKNOWLEDGMENT

The research is supported by the National Natural Science Foundation of China (Grant No. 61702534).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). 265–283.

[2] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, et al. 2019. Data platform for machine learning. In Proceedings of the 2019 International Conference on Management of Data. 1803–1816.

[3] Len Bass, Ingo Weber, and Liming Zhu. 2015. DevOps: A software architect's perspective. Addison-Wesley Professional.

[4] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 1387–1395.

[5] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing 1, 3 (2014), 81–84.

[6] Ekaba Bisong. 2019. Kubeflow and Kubeflow Pipelines. In Building Machine Learning and Deep Learning Models on Google Cloud Platform. Springer, 671–685.

[7] The Github Blog. 2015. Announcing Git Large File Storage (LFS). Retrieved from https://github.blog.

[8] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big data: A survey. Mobile networks and applications 19, 2 (2014), 171–209.

[9] Google Cloud. 2020. MLOps: Continuous delivery and automation pipelines in machine learning. Retrieved from https://cloud.google.com/solutions/machine-learning/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[10] Jeremy Lewi David Aronchick. 2017. Introducing Kubeflow - A Composable, Portable, Scalable ML Stack Built for Kubernetes. Retrieved from https://kubernetes.io/blog/2017/12/introducing-kubeflow-composable/.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition. Ieee, 248–255.

[12] Drone. 2020. Automate Software Build and Testing. Retrieved from https://drone.io/.

[13] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. Continuous integration: improving software quality and reducing risk. Pearson Education.

[14] Gitea. 2020. Gitea - Git with a cup of tea. Retrieved from https://gitea.io/en-us/.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.

[16] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. Kubernetes: up and running: dive into the future of infrastructure. " O'Reilly Media, Inc.".

[17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).

[18] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 4700–4708.

[19] Jez Humble and David Farley. 2010. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education.

[20] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. 2019. Modelops: Cloudbased lifecycle management for reliable and trusted ai. In 2019 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 113–120.

[21] Iterative. 2020. DVC: Data Version Control - Git for Data "&" Models. https://doi.org/10.5281/zenodo.012345

[22] Yashpalsinh Jadeja and Kirit Modi. 2012. Cloud computing-concepts, architecture and challenges. In 2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET). IEEE, 877–880.

[23] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture. 1–12.

[24] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[25] Kubeflow. 2020. Documentation for Kubeflow Pipelines. Retrieved from https://www.kubeflow.org/docs/pipelines/.

[26] Fumihiro Kumeno. 2019. Sofware engneering challenges for machine learning applications: A literature review. Intelligent Decision Technologies 13, 4 (2019), 463–476.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. Nature 521, 7553 (2015), 436–444.

[28] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. Linux journal 2014, 239 (2014), 2.

[29] Mathias Meyer. 2014. Continuous integration and its tools. IEEE software 31, 3 (2014), 14–16.

[30] Nvidia. 2019. DCGM API reference. Retrieved from https://docs.nvidia.com/deploy/dcgm-api/group__dcgmFieldIdentifiers.html.

[31] Babatunde K Olorisade, Pearl Brereton, and Peter Andras. 2017. Reproducibility in machine Learning-Based studies: An example of text mining. (2017).

[32] Daniel Pop. 2016. Machine learning and cloud computing: Survey of distributed and saas solutions. arXiv preprint arXiv:1603.08767 (2016).

[33] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In Advances in neural information processing systems. 2503–2511.

[34] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[35] Dave Steinkraus, Ian Buck, and PY Simard. 2005. Using GPUs for machine learning algorithms. In Eighth International Conference on Document Analysis and Recognition (ICDAR'05). IEEE, 1115–1120.

[36] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. 2017. Hardware for machine learning: Challenges and opportunities. In 2017 IEEE Custom Integrated Circuits Conference (CICC). IEEE, 1–8.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1–9.

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition. 2818–2826.

[39] Weiaicunzai. 2017. Pytorch-cifar100. Retrieved from https://github.com/weiaicunzai/pytorch-cifar100.

[40] Wikipedia. 2020. Online machine learning. Retrieved from https://en.wikipedia.org/wiki/Online_machine_learning.

[41] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. 2010. Cloud storage as the infrastructure of cloud computing. In 2010 International Conference on Intelligent Computing and Cognitive Informatics. IEEE, 380–383.

[42] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the IEEE conference on computer vision and pattern recognition. 6848–6856.