Reproducible Model Sharing for AI Practitioners

Amin Moradi
amin@moradi.io
LIACS, Leiden University
The Netherlands

Alexandru Uta
a.uta@liacs.leidenuniv.nl
LIACS, Leiden University
The Netherlands

ABSTRACT

The rapid advancements in AI and Machine Learning (ML) technology, from both industry and academia lead to the need of large-scale, efficient and safe model sharing. With recent models, reproducibility has gained tremendous complexity both on the execution and the resource consumption level. Although sharing source-code and access to data is becoming common practice, the training process is limited by software dependencies, (sometimes large-scale) computation power, specialized hardware, and is time-sensitive. Next to these limitations, trained models are gaining financial value and organizations are reluctant to release models for public access. All these severely hinder the timely dissemination and the scientific sharing and reviewing process, limiting reproducibility. In this work we make the case for transparent and seamless model sharing to enable the ease of reviewing and reproducibility for ML practitioners. We design and implement a platform to enable practitioners to deploy trained models and create easy-to-use inference environments, which can be easily shared with peers, conference reviewers, and/or made publicly available. Our solution follows a provider agnostic practice and can be used internally in institutional infrastructures or public/private cloud providers.

CCS CONCEPTS

• **General and reference** → **Reliability**; **Empirical studies**.

KEYWORDS

reproducibility, model sharing

ACM Reference Format:
Amin Moradi and Alexandru Uta. 2021. Reproducible Model Sharing for AI Practitioners. In *Fifth Workshop on Distributed Infrastructures for Deep Learning (DIDL) 2021 (DIDL '21), December 6, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3493652.3505630

1 INTRODUCTION

Recent interest and investments of academia and industry in AI research have attracted many-fold increases in the number of conference contributions and trained machine learning (ML) models. A report from 2019 [28] states that the number of submitted articles for ML conferences has increased by up to 30% compared to 25% in 2018. This growth in the number of submissions continues, as we depict in Table 1, and gives rise to large numbers of models that need to be evaluated and assessed by the (scientific) community. In this paper we make the case for building model sharing platforms to aid the community and AI/ML venues in the reproducibility process without going through the costly re-training process.

Table 1: Increase in the number of submissions in the last four years for NeurIPS, ICML and ICLR.

Conference	2020	2019	2018	2017
NeurIPS	9454	6743	4856	3590
ICML	4990	3424	2473	1676
ICLR	2594	1449	935	490

So far, reproducibility has been tackled for ML, as for other fields, by sharing code and data, or following clear performance evaluation protocols [30]. This approach led to remarkable improvements in reproducibility. As a result, containerizing source-code and data is en route to become routine practice. Although this improves reproducibility [10], the increase in the size of training datasets and trained models, and the complexity of the deployments are major drawbacks for this approach. It is increasingly difficult to retrain models, as this process means utilizing specialized hardware, using large amounts of energy, and increasingly larger carbon emissions [23, 27]. At industry level, sharing data might also be impractical due to the inability to share sensitive training data. We therefore make the case for a model sharing platform to seamlessly allow practitioners from both industry and academia to share their models without the need for re-training and without the need to share training data and even the model itself. For (conference) reviewers to reproduce work, the process should be as easy as running inference in a sandboxed environment, with the input of choice.

Large-scale models are difficult to retrain, thus reproduce. For example, in 2019 Google published BERT [8], a revolutionary language model with 360 M parameters and a trained model of 1.5 GB. Following BERT, GPT-2 [25] published by OpenAI had 1.5 B parameters and almost 7 GB in size. In 2020, OpenAI pushed the language models further by publishing GPT-3 [5] with 176 B parameters and 600 GB in trained model size. This extreme growth rate and complexity does not reflect in all ML research but it is significant. Image classification networks, Deep Reinforcement Learning [20] and Generative Networks [11] often require powerful GPUs and TPUs [19] to run for hours or days to reproduce results.

While the research community welcomes the growth and success of recent findings in AI and ML, the pruning and evaluation of scientific work have become major challenges. Reviewers are limited and bound by their resources and cannot reproduce all findings of the subject articles. Thus, the judgement lies between the trust of the reviewer and the reviewee. To make this cumbersome process easy and intuitive without pressuring researchers to undergo many changes in the reviewing process, we designed and implemented a conference and scientific evaluation oriented platform—MLC.

DIDL '21, December 6, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-9173-3/21/12.
https://doi.org/10.1145/3493652.3505630

Inference at scale in cloud computing is achieved through orchestration platforms like Kubernetes [13]. KFserver [7], Polyaxon [24] and Kubeflow [4] are open-source ML inference platforms, optimized for scalability, high availability, and performance. Managing these platforms requires expert knowledge in cloud, security, networking and distributed processing. Serverless implementations also impose significant resource allocation and deployment effort [18]. Although all these platforms could be used for model sharing, they are not targeted to improve reproducibility and reviewing processes and require large-scale resources for deployment.

In turn, in this paper we showcase an easy-to-use platform that enables seamless model sharing for improving the reviewing and reproducibility processes of ML-related conferences. Our prototype, MLC, proposes a scalable, cloud-based architecture, which can be deployed on standalone servers or distributed clusters within institutes or public/private cloud providers.

At the infrastructure level, we designed MLC with a scale-to-zero policy for maximum cost efficiency and trained model privacy preservation. We have also introduced automation toolsets to facilitate running MLC internally within universities infrastructures, if needed. Although managing and bootstrapping MLC requires intermediate technical skills, our approach in providing such a platform is to minimise maintenance and ease of use for both parties—the MLC administrator and the MLC users (e.g., practitioners, reviewers). Thus, as a complementary toolset we introduce a Command Line Interface (CLI) for researchers to easily transfer trained models from a local environment to a remote MLC deployment for publishing their findings. Within the CLI, researchers can create shareable links with public, private or limited access for (re)viewers in an interactive presentation view.

MLC enables access restriction on deployed models such that only reviewers and model submitters can interact within the MLC Model Showcase user interface. In addition to the use-case in the research community, MLC enables the industry to reproduce, test and evaluate ML models faster and more precise without going through the training process and provide a clear road-map for scalable inference.

MLC is an open-source project hosted on Github[1]. In summary, the contributions of this work are the following:

(1) The design and implementation of MLC, which is an infrastructure agnostic model sharing platform, built to help scientific venues, practitioners and industry at large to share and evaluate trained models without re-running expensive training processes, or without sharing confidential training data or parameters.
(2) The prototype and initial performance evaluation of MLC, showing that MLC is efficient in sharing trained models. We use three real-world models and run inference on real-world data, deploying MLC in GCP.

2 BACKGROUND

To enable seamless model sharing, MLC builds upon containers and their orchestration, microservices and model serving. We discuss containerized and container orchestration platforms and how micro-services can work along with service discoverability tools for serving ML models.

Recent improvements in Cloud-based applications have pushed the boundaries of ML training and computational limits further down the line. Hardware limitation is fading, and scaling ML training is becoming painless and straightforward. Training on Petabytes of data on hundreds of powerful GPUs is now possible on public cloud providers. With each iteration of new GPUs and TPUs and increase in availability and cost reduction, we can expect a full transition both for training and inference to cloud platforms.

Containerisation is one of the main drivers that enabled the transition of software to cloud-based architectures. By isolating dependencies over the software layer and network access, containers are able to replicate and distribute workloads over a collection of self-contained applications. These portable applications have also attracted the ML community to leverage industry best-practices for shipping software. By using containers, Machine Learners are able to preserve a well reproducible experiment regardless of the runtime environment. This phenomenon has enabled better reproducibility in ML research, driven by automation at the application layer and isolation at the data and information layer [29]. Docker is one the commonly used container run-time platforms adapted by the CNCF and widely used throughout software communities.

Container orchestration comes into action as systems grow in the number of containers and communication and connectivity between them gains complexity. Orchestration tools mainly manage lifecycles of containerised applications, horizontal scalability and replications of containers as well as vertical scalability and resource management. Platforms like Docker Swarm [9], OpenShift [22] and Kubernetes [13] are amongst the most commonly used orchestration platforms. Resource management, scalability-to-zero and GPU/TPU [19] adoption enabled the ML industry to use orchestration platforms as main drivers of ML applications.

Kubernetes works on top of a descriptive configuration management system to control resources across multiple server nodes inside an internal network topology. Resources are building blocks of the Kubernetes ecosystem as they communicate with the Kubelet to provision, manage and control containerized applications using CRI-O. Kubelet, by managing the smallest computational units, Pods, structure higher-order complex Deployment in a way that each building block share storage and network resources. Deployments can then replicate copies of Pods for horizontal scaling and managing network routing policies between each deployment. As resource consumption increases with new Deployments and Pods, Kubelet plays a significant role in managing the compute resources and provisioning new Nodes into the Kubernetes Cluster. Kubelet enables Kubernetes clusters to vertically scale while it preserves network sanity and controls pods and deployment lifecycles.

By defining node-affinity configuration, we can allocate custom computational machines to a target deployment. Using node-affinity will enable us to mitigate GPU powered Nodes managed by the Kubernetes cluster. As these nodes follow the scalability principles of Kubelet, they also allow large scale distributed training over multiple GPUs and TPUs both at the inference and training layer. Self-management, auto-scalability, scale-to-zero and custom Node creation make Kubernetes a key role player in the feature of ML.

[1] https://github.com/maminio/mlc.git

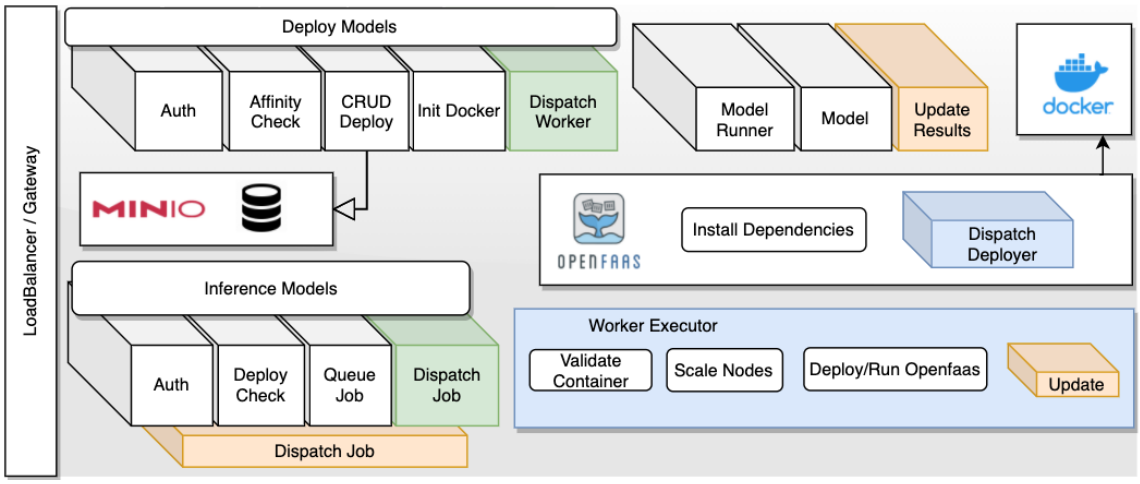

Figure 1: MLC main architecture and data flow. Each request is asynchronously dispatched throughout the micro-service implementation to reduce processing bottlenecks and enable scalability.

Service-Mesh and network topology come into action as containerised application deduce dependency on one another in form of communications over different network protocols. As systems grow in complexity and increase in dependency on third-party services, communication plays a crucial role in maintaining security, performance and discoverability of various services across a system. The Service-mesh application layer allows individual self-contained software in a micro-service architecture [26] to have clear visibility to other members of the system while preserving and controlling information transfer inside a cluster of containers. Orchestration platforms create a local cluster networking which allocates internal IP addresses for each entity of the system. Service-mesh applications like Istio [16], Consul [14] and Linkerd [21] accommodate on top of this internal network and provide DNS management and service discoverability, network security and load balancing between services. A service-mesh plays a crucial role in network scalability and reliable communication between entities in an orchestration platform. Istio is a service mesh that brings advanced traffic management and observability features and the ability to centrally decide on security policies and rate limitations for the entities in a mesh. By injecting a high-performance Envoy proxy containers to each pod as a Sidecar, Istio can manage inbound and outbound traffic at the POD level. In addition, by enabling mutual TLS, Istio provides an extra security layer over the internal service mesh connections. For MLC, we enable Istio and its security features.

Asynchronous micro-service architecture is a way of distributing communication between building blocks of a system so that each entity gains autonomy and reduction in dependencies at the communication layer. By transporting a cluster's incoming request into a messaging bus like NATS [6], Kafka [2] or RabbitMQ [3], we can distribute a request payload to multiple micro-services without creating network bottlenecks. Compared to traditional request-response communication, async communication allows us to run long-running tasks without overhead on the service mesh as requests no longer need to wait for a response. This communication method enables the ML training process or inference to act as an individual, self-contained service in our application layer. Using Kubernetes as service orchestration along with asynchronous micro-service architecture, we can allocate computational resources to isolated and dependency-free ML services without creating network and communication bottlenecks.

Model Serving and inference is a technique where the ML trained model output is served as an API endpoint to perform prediction, data generation or data transformation. By taking snapshot and storing the final state of an ML training process with all the parameters and leveraging inference-servers like TensorflowServe or ad-hoc execution, we can host ML models. This method perfectly fits into the Micro-Service architecture on top of a Kubernetes cluster. The async-communication handles the long-running inference process over Istio service mesh to preserve security and discoverability. Kubernetes resource allocation policies manage the heavy-duty tasks and resource-hungry ML models. All this is maintained across a cost-efficient, scalable platform.

3 DESIGN AND IMPLEMENTATION

High-performance ML serving on Cloud platforms like GCP and AWS has much complexity and requires maintainability and cloud expertise to be deployed. This complexity has created a barrier for the research community to take advantage of these services for more straightforward use cases like sharing models for research conferences and reproducibility at the inferential level. MLC tackles the maintenance and implementation complexity, focusing on zero-dependency on Cloud Providers and is deployable on on-premise institutional infrastructures, as well as public/private clouds.

This section will explain our platform's backbone and how containerised ML applications can perform as standalone API endpoints that will use lightweight container schedulers over an asynchronous communication channel. Our platform enables a layer of privacy to researchers source code and a trained model.

3.1 Main Architecture

MLC is designed on top of Kubernetes with micro-service design patterns, communicating over Kafka message broker behind Istio Service Mesh. Istio provides an ingress gateway for incoming traffic and reroutes the appropriate services using the Virtual Service resources. Figure 1 presents the main architecture of MLC.

API and **Run** are the two main micro-services processing incoming requests from Virtual Services. The **API** micro-service is responsible for managing the Authentication of users and access management of deployments. It also manages to create new Inference deployments and the containerised ML model's build process's lifecycle. The **Run** micro-service handles the execution of ML models over a distributed container-as-a-function service.

As our primary objective is to reduce dependency on cloud platforms, MLC provides a container registry for hosting deployed ML containers and Minio FileStorage Services similar to AWS S3, which enables researchers to upload source codes and supplementary artefacts of ML models. As the primary database, we use a self-managed stateful deployment of MongoDB which satisfy the simple purpose of storing the list of the deployed services and their history of execution with regard to each authorised user. We use OpenFaaS as our main container execution engine to control ML models over API endpoints. OpenFaaS is lightweight compared to KNative and a similar platform, making it the perfect candidate for our implementation.

3.2 Data Flow

Each deployment is a *trained model + source code* with a unique name across the platform as a convention. Deployment configurations can be stored in `JSON` or `YAML` format and must contain the model's execution process. The execution process is defined with an ad-hoc method similar to the testing process of ML models. This means researchers can use the same source code with minor changes to read incoming data and perform inference. The deployment configuration file contains the test module file path, which can be of any type (`.py, .r, . . .`) and a result destination where the model's output will be stored. Figure 2 shows a sample configuration file for a CycleGAN [31] model, which takes a python file as the inference entry point along with several execution flags.

MLC model configuration file was designed with at most flexibility in mind to enable researchers use a variety of models and frameworks. It supports multiple package managers, ML frameworks and custom commands and flags for dynamic configuration. Although we showcase that most ML models can be deployed for inference without any changes, there are cases where researchers must adapt their inference to fit MLC model.

In general we divide inference workloads based on their input type in two main categories. Models that require run-time configurations and environment variables like GPT-2 where the model generates text based on the input data from a run-time flag and models that require external files like CycleGAN where they process an image and generate a new image as output. By filling in a model run-time specifications into the YAML configuration file and using MLC's CLI tool, researchers can deploy models and interact with inference configuration.

```yaml
name: "snow-gan"
language: "python-2.7"
framework: "tensorflow"
description: "A very nice project"
version: 1.1.0
model:
  fileType: ".zip"
  path: "source-code/snow-gan"
  exec_cmd: ""
  command: "bash activateEnvAtoB.sh"
  output:
    path: "./results/snow-cycle-gan/test_latest/images/"
    type: "png"
  input:
    type: "download"
    download_destination: "./datasets/testA/image.jpg"
package:
  type: "conda"
  requirements_file: "./requirements.txt"
```

Figure 2: A sample YAML configuration file for CycleGAN model.

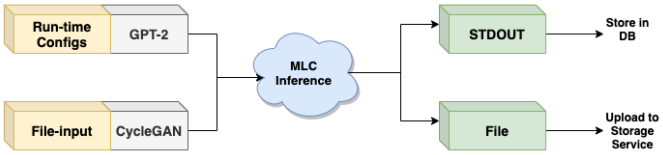

Figure 3: A breakdown of input and output for deployed MLC models.

Moreover, as depicted in Figure 3, supports multiple input and output types, such as file and runtime configuration inputs. Regarding outputs, MLC is able to save stdout to a database and simply list it to the user in the web interface, or upload the output to the MinIO storage service, or even AWS S3.

3.3 Container build

One of the main challenges and bottlenecks in the deployment process of models is installing package requirements and external dependencies for a new container. Our high-level configuration YAML file provides a self-contained and fully maintainable description of how external dependencies must be installed. Researchers can set *requirements.txt* file path, and the MLC deployment process will automatically install them upon building the inference containers.

One of the most common methods to manage external dependencies for each container is to build and create a new container on each deployment. As MLC entirely runs on Kubernetes and Docker containers, this method adds a bottleneck to the deployment process. Building Docker containers inside docker containers has always been a challenge, and solutions like Kaniko by Google Cloud have added savvier improvements to this issue.

Table 2: CycleGAN model build time and one update iteration in dependency files. In the MLC method we incrementally add new packages in-comparison to installing them again at build time.

Method	First Build time (s)	Upload time (s)	Second Build time (s)	Avg Node CPU load
Kaniko	966	41.6	966	80%
Local	210	41.6	210	80%
MLC	**104**	**5**	**10**	**35%**

Although Kaniko initially will solve most of the problems in building containers in containers on Kubernetes, it takes a very long time compared to building them locally. To accelerate our deployment lifecycle of ML models, we created a base optimised ML container. It will install required packages and all external dependencies and eventually notifies the API service to commit the container with all the installed requirements. Our method improved the time taken by build process by a factor of 5, as depicted in Table 2. Although fast container building is not a critical feature for reproducibility tools like MLC, we believe that the ability to build containers faster is an important feature toward user adoption.

4 EXPERIMENTS

We demonstrate a series of experiments on the MLC platform. Our main objective in designing MLC was to provide a set of solutions for practitioners and AI conferences to share and showcase models without the complexity of ML inference platforms like KF-Service, KubeFlow and Polyaxon.

We performed our experiments on three different models to diversify our results based on the size of containers and computation required for each inference. The three models we use to exemplify our MLC platform are the widely used ResNet [12], CycleGAN [1] (a variation of GAN), and GPT-2 [25]. Given the large differences between these three models and their wide-range deployment and use in the community, we believe they are sufficient for showing the capabilities of MLC.

As models consume a wide range of computational resources and the run-time of each deployed container is based on the complexity and internal computation, in our experiments, we focused on elasticity and boot-up of our deployments. We ran our experiments on a three-node Kubernetes cluster on Google Kubernetes Engine (GKE) v1.18.17. To manage third-party services and dependencies we used Helm [17] and Terraform [15] to increase reproducibility and maintenance.

4.1 Cold Start

Some of the known challenges in container-as-a-function platforms are the scale-to-zero and cold-start of containers problems. ML containers consist of several run-time packages and the trained model; this means they are usually large in size compared to web-based containers and it takes longer to scale them horizontally.

In Figure 4, we demonstrate our results when we execute inference on scaled-to-zero models. OpenFaaS provides a built-in queue-worker that is responsible for caching and scaling down of containers. As deployed model have a similar CRD to Kubernetes PODs, they are almost always in the warm state. The results show that the performance penalty of cold starts is not severe, being smaller than 500 ms. We believe this is good enough performance

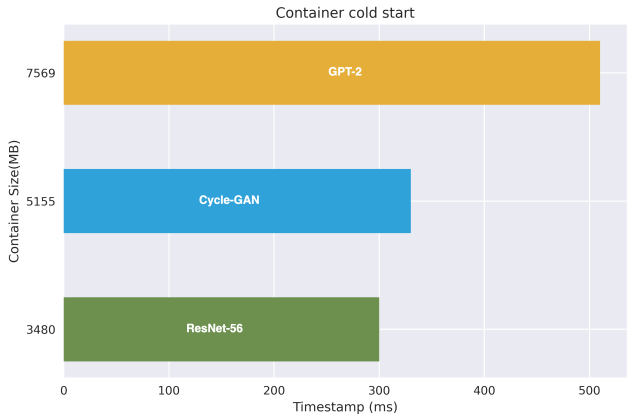

Figure 4: A comparison between three models in scaling from zero to one deployment. GPT-2 with the largest (7GB) size and more RAM consumption take longer than ResNet-56 and CycleGAN model.

Table 3: An overall run-time comparison of three large Deep Neural Network Models deployed on a single Kubernetes node. Avg over 10 runs.

Model	Container Size (MB)	Avg. Runtime (s)	RAM (MB)	vCPU
GPT-2	7,569	**3.86**	800	4
Cycle-GAN	5,155	**2.33**	800	4
ResNet-56	3,480	**0.89**	600	4

for platform whose main purpose is models sharing and reproducibility and not high-performance, low-latency inference serving. However, even for such a case, these numbers are still competitive, as cold-starts in containerized environments are typically higher, even in the order of seconds for several use-cases.

4.2 Run-time

To examine the performance of MLC, we performed ten inference runs for each model; all deployed on one Kubernetes node with 2GB of RAM and 4 vCPUs. Our results in Table 3 shows MLC platform can run inference on models as big as GPT-2 with 5GB of trained weights and store the data securely in MLC database in less than 4 seconds. We consider this sufficiently performant for the purposes of model sharing and reproducibility. However, with larger models and GPU-based inference, it is trivial to make use of scaled-up containers.

Table 4: Resource consumption comparison of ML inference platforms. MLC requires less RAM and CPU to host ML models.

Platform	Base No. of Pods	Min. Nodes	Min. RAM	Min. vCPU	Auto Model of Pods
KFService	21	3	4GB	4	Yes
KubeFlow	17	3	4GB	4	-
MLC	**8**	**1**	**2GB**	**4**	Yes

4.3 Comparison with Containerized Inference Platforms

In recent years, Cloud Native Computing Foundation (CNCF) ML practitioners take advantage of platforms like Kubernetes for managing ML applications at scale. Platforms like KNative provide automation, security, and discoverability on a single ecosystem by providing multiple application layers. KFServing and KubeFlow are two of the leading ML platforms for serving models on Kubernetes, which are both based on KNative. Although KNative provides a fully managed platform, it requires more RAM and CPU on each Node. As both platforms are designed to support a wide range of models and deployments, they have more complexity for non-cloud-engineers. Table 4 shows an overall comparison of MLC with KF-Serving and Kubeflow in terms of resource consumption and complexity. We believe that the reduced resource consumption and overall much reduced complexity of used make MLC a prime candidate for the adoption toward model sharing for reproducibility purposes by the AI and ML communities at large.

5 CONCLUSION

Machine Learning and AI research are moving faster than ever but reproducing novel proposed results is limited by computational resources and software complexity. Some of these challenges are being addressed by containerized environments and sharing of data and code. However, this is insufficient and sometimes infeasible. Today, with the emergence of more extensive and deeper networks, we are experiencing reproducibility challenges which can be solved by seamless model sharing, thus enabling fast inference for practitioners who need to quickly prototype or test a new model. In this work, we implemented a model sharing platform to improve the inference reproducibility of ML models for practitioners, conferences, and the AI/ML research communities at large. We designed MLC to run on institutional or cloud infrastructure with the least amount of maintenance required. Our experiments show, compared to similar platforms, that MLC requires less compute resources and simpler deployment of ML models with our interactive CLI. Using MLC, researchers and practitioners can share ML models without sharing source code or the trained model itself. MLC also helps the industry to adapt and test the latest research contributions on their products.

ACKNOWLEDGEMENTS

The work in this article was in part supported by The Dutch National Science Foundation NWO Veni grant VI.202.195.

REFERENCES

[1] Amjad Almahairi, Sai Rajeshwar, Alessandro Sordoni, Philip Bachman, and Aaron Courville. 2018. Augmented cyclegan: Learning many-to-many mappings from unpaired data. In *International Conference on Machine Learning*. PMLR, 195–204.
[2] Apache. 2021. *Distributed event streaming platform.* https://kafka.apache.org
[3] Apache. 2021. *Lightweight message broker.* https://www.rabbitmq.com
[4] Ekaba Bisong. 2019. *Kubeflow and Kubeflow Pipelines.* Apress, Berkeley, CA, 671–685. https://doi.org/10.1007/978-1-4842-4470-8_46
[5] Tom B. Brown et al. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
[6] NATS CNCF. 2021. *Connective Technology for Adaptive Edge Distributed Systems.* https://nats.io
[7] Clive Cox, Dan Sun, Ellis Tarn, Animesh Singh, Rakesh Kelkar, and David Goodwin. 2020. Serverless inferencing on Kubernetes. arXiv:2007.07366 [cs.DC]
[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
[9] Docker. 2021. *Docker.* https://docker.com
[10] Grigori Fursin. 2021. Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces. *Philosophical Transactions of the Royal Society A* 379, 2197 (2021), 20200211.
[11] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:1406.2661 [stat.ML]
[12] 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.
[13] Google Inc. 2021. *Kubernetes.* https://kubernetes.io
[14] Hashicorp Inc. 2021. *Hashicorp Consul.* https://www.consul.io
[15] Hashicorp Inc. 2021. *Terrform infrastructure as code software.* https://www.terraform.io
[16] Istio Inc. 2021. *Istio.* https://istio.io
[17] Microsoft Inc. 2021. *Helm, Package manager for kubernetes.* https://helm.sh
[18] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. arXiv:1710.08460 [cs.DC]
[19] Norman P. Jouppi, , et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246
[20] Yuxi Li. 2018. Deep Reinforcement Learning. arXiv:1810.06339 [cs.LG]
[21] Linkerd. 2021. *Linkerd.* https://linkerd.io
[22] OpenShift. 2021. *OpenShift.* https://www.openshift.com
[23] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon Emissions and Large Neural Network Training. *arXiv preprint arXiv:2104.10350* (2021).
[24] Polyaxon. 2021. *Polyaxon.* https://polyaxon.com
[25] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[26] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. 2018. Modernize Digital Applications with Microservices Management Using the Istio Service Mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '18)*. IBM Corp., USA, 359–360.
[27] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2020. Energy and policy considerations for modern deep learning research. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 13693–13696.
[28] David Tran, Alex Valtchanov, Keshav Ganapathy, Raymond Feng, Eric Slud, Micah Goldblum, and Tom Goldstein. 2020. Analyzing the Machine Learning Conference Review Process. arXiv:2011.12919 [cs.LG]
[29] Ana Trisovic, Philip Durbin, Tania Schlatter, Gustavo Durand, Sonia Barbosa, Danny Brooke, and Mercè Crosas. 2020. Advancing Computational Reproducibility in the Dataverse Data Repository Platform. In *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems* (Stockholm, Sweden) *(P-RECS '20)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3391800.3398173
[30] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is big data performance reproducible in modern cloud networks?. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 513–527.
[31] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. 2020. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. arXiv:1703.10593 [cs.CV]