NSML: A Machine Learning Platform That Enables You to Focus on Your Models

Nako Sung[1], Minkyu Kim[1], Hyunwoo Jo[1], Youngil Yang[1], Jinwoong Kim[1], Leonard Lausen[1,4], Youngkwan Kim[1], Gayoung Lee[3] Donghyun Kwak[2], Jung-Woo Ha[1], and Sunghun Kim[1,4]
Clova AI Research, NAVER Corp., Seongnam, Korea[1]
Search Solution Inc.[2], NAVER WEBTOON Corp. [3]
Hong Kong University of Science Technology[4]
`{nako.sung, min.kyu.kim, hyunwoo.jo, youngil.yang, jinwoong.kim, leonard.lausen, ykkim.kim}@navercorp.com`
`gayoung.lee@webtoonscorp.com`
`{donghyun.kwak, jungwoo.ha, sung.kim.n}@navercorp.com`

Abstract

Machine learning libraries such as TensorFlow and PyTorch simplify model implementation. However, researchers are still required to perform a non-trivial amount of manual tasks such as GPU allocation, training status tracking, and comparison of models with different hyperparameter settings. We propose a system to handle these tasks and help researchers focus on models. We present the requirements of the system based on a collection of discussions from an online study group comprising 25k members. These include automatic GPU allocation, learning status visualization, handling model parameter snapshots as well as hyperparameter modification during learning, and comparison of performance metrics between models via a leaderboard. We describe the system architecture that fulfills these requirements and present a proof-of-concept implementation, NAVER Smart Machine Learning (NSML). We test the system and confirm substantial efficiency improvements for model development.

1 Introduction

Deep learning [1] has recently enabled remarkable progress in diverse fields including as speech [2, 3], computer vision [4, 5], machine translation [6], and natural language processing [7]. In addition, multiple easy-to-use deep learning frameworks such as TensorFlow [8], PyTorch [9] and MXNet [10] significantly reduce the barrier to implementing such deep learning models.

Still, performing experiments on deep learning architectures requires considerable time and computational resources, and becomes only feasible by leveraging multiple servers and GPUs. These requirements introduce a non-trivial amount of supporting tasks into a researchers' workflow, such as GPU allocation and distribution of code and data to servers. As a result, researchers often spend large portions of their time on such incidental tasks.

Next to computational requirements, hyperparameter sensitivity of many deep learning models forces researchers to run their experiments using different hyperparameter combinations and keep track of the results, often in a manual fashion. To debug the training process and get better understanding of hyperparameter effects visualization packages such as TensorBoard [11] and Visdom [1] can be used, but support to compare concurrently performed experiments on many models is lacking. Some

[1]https://github.com/facebookresearch/visdom

machine learning platforms such as Google Cloud ML platform [2] aim to reduce these issues. However, vendor lock-in and support of a restricted set of deep learning libraries only pose their own challenges.

In this paper, we first collect requirements for enhancing research efficiency, and then describe a system architecture that fulfills the requirements. Based on the architectue design, we implement NAVER [3] Smart Machine Learning (NSML). The requirements cover many incidental subtasks from automatic GPU allocation and releases to a leaderboard for comparing the model performances, which are not core but essential tasks conducted manually by many researchers. NSML automatically conducts these subsidiary subtasks instead of humans and provides web-based interfaces that are easy to use, thus enabling researchers to focus on their model implementation and data. Currently, NSML supports diverse deep learning libraries with Python APIs including TensorFlow, PyTorch, and MXNet, which are the most popular in deep learning research communities. In addition, NSML supports not only servers with single GPU but also GPU-clusters.

We implemented NSML on a server cluster equipped with 80 P40 GPUs as a prototype. We evaluate NSML based on alpha tests conducted by researchers at NAVER. They implemented models for three tasks: face emotion recognition, face generation, and movie rating prediction. Comments from NSML users indicate that NSML dramatically enhances the research efficiency.

The contributions of NSML are summarized as follows:

- We define the requirements for efficient deep learning research.
- We designed and implemented a new deep learning research platform that enhances the research efficiency.
- We demonstrate the usages of NSML on three real-world applications.

2 Common Workflow Challenges in Machine Learning Research

We list challenges observed from an online community [4] comprising more than 25,000 global deep learning researchers and developers over a time frame of one year.

Difficulty in resource management: With increasingly deep and complex models, increasing computational requirements come along. However, it is often difficult to acquire sufficient resources for training models. For example, to train ResNet-152 [12], eight GPUs are required to optimize the 152 layers in a timely fashion. Consider a case where the total number of GPUs in a cluster is sufficient, but due to bad scheduling no single server with eight idling GPUs is available, so that the model cannot be trained. It is therefore necessary to appropriately allocate each task to the server based on the amount of resources required. However, manual assignment is likely to cause inefficiency if multiple developers share the server.

Difficulty in data management: Deep learning methods require large amounts of training data. Popular datasets such as ImageNet [13] or YouTube-8M [14] have sizes of multiple gigabytes or more. Researchers must ensure that the datasets are available on each server on which their models should be trained, which so far has to be performed manually. Furthermore, it is inefficient to access the dataset every time separately or to duplicate the preprocessing for running multiple experiments with the same dataset.

Lack of integrated management tools: Researchers need to perform multiple experiments to compare the performances between different models and to find better hyperparameters. These experiments often run simultaneously on multiple servers in order to reduce training time. To manage and control each training task, it is so far necessary for researchers to manually access each of the servers for collecting and comparing results. These tasks are time-consuming and could be automatized.

Difficulty in tracking experiment environments over time: Over time researchers often change model structure or experiment hyperparameters in the quest for improved model performance. Without tedious and manual recordings, it can be difficult to reproduce past experiments as the information about the exact conditions under which an experiment was conducted may be lost.

[2]https://cloud.google.com/products/machine-learning/
[3]https://www.navercorp.com/en/index.nhn
[4]https://www.facebook.com/groups/TensorFlowKR

3 NSML: NAVER Smart Machine Learning

3.1 Requirements of ML Platforms

We elicit the following requirements for practical large-scale ML systems based on the challenges identified in the previous section and the comments from the deep learning community.

Resource Management

- Better computational resource management to improve utilization and job scheduling.
- Automatic execution of the entire machine learning pipeline, beginning with data preprocessing to finding the best model, whenever e.g. the dataset is updated.

Data Management

- Users should be able to post datasets once and reuse them for multiple models.
- Users should be able to share datasets with others.
- The systems should be capable of preprocessing large datasets.

Serverless Configuration

- The systems should not require any configuration or (manual) installation from users.
- The systems should not be limited to the use of specific libraries and should not depend on specific platforms.
- The servers should be hidden from users, and users should be able to use the system in a serverless manner by just submitting a training or testing tasks to the platform.

Experiment Management and Visualization

- The systems should handle parallel runs with different jobs priorities.
- The systems should automatically visualize and summarize the learning progress.

Leaderboard

- The systems should provide a leaderboard which can compare various models and hyperparameters for each dataset.

AutoML

- The systems should be able predict the performance of experiments based on previously run experiments.
- The systems should be able to automatically optimize the hyperparameters based on the performance predictions.
- The systems should save the model of best score.

3.2 NSML System Architecture

We described the system architecture of NSML as shown in Figure 1(a) in detail. NSML consists of three main parts: scheduler, containerized system, and user interface (UI).

Scheduler: NSML has a scheduler with a centralized scheduling model based on the master-slave structure for efficient allocation of computational resources such as CPUs and GPUs. In NSML, a single node (i.e., master node) is in charge of monitoring all computational resources and scheduling tasks for all clients. Other nodes (i.e., slave nodes) collect information about their computational resources and periodically report it to the master node. A centralized model often suffers from a single point of failure (SPOF) [15]. We handle this issue with the leader election process by electing new master node as in *Zookeeper* [16] when the master node fails. To train models, clients have to submit a job to the scheduler for obtaining computational resources. Then, the scheduler first looks over the job queue. If the job queue is empty, the scheduler immediately selects an available slave

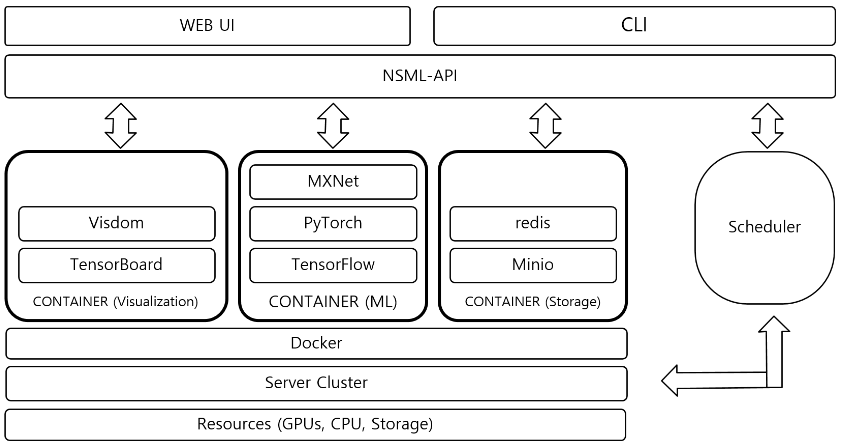


Figure 1: System Architecture of NSML

node and informs the client about its address. Compared to first inserting the job into the queue, this approach allows the scheduler to avoid queue operation overhead. Otherwise, the job will be inserted in the job queue.

Containerized System: We separate containerized systems into two groups: storage containers and ML containers. Storage containers use *minio* [5] to store and supply datasets to ML containers. They also store the performance of all models trained with the respectively provided dataset as well as display the results in a leaderboard to make clear which model performed best. Storage containers furthermore back up intermediate and final results of trained models and also store the source code associated with the experiments so that users can easily reproduce and improve both their own and other users' models. ML containers can contain any type of systems and libraries. When a user sets up an environment, NSML automatically packages it into a ML container and copies the user's codes and datasets from the respective storage containers. Then NSML runs the code, reports the learning status and backs up intermediate and final results of trained models.

Containerized ML system can solve many challenges that frequently appear during training deep learning models. For instance thanks to the containerized infrastructure different users can rely on different library versions for their machine learning environment via NSML. If one user wants to use PyTorch in python 2.7, he or she just needs to select the corresponding base docker image. Meanwhile other users can use TensorFlow in python 3.6 and their models can be run on the same machine thanks to the isolation provided by the containers. NSML also provides visualization of trained model via both TensorBoard and Visdom using the ML containers to simplify model evaluation.

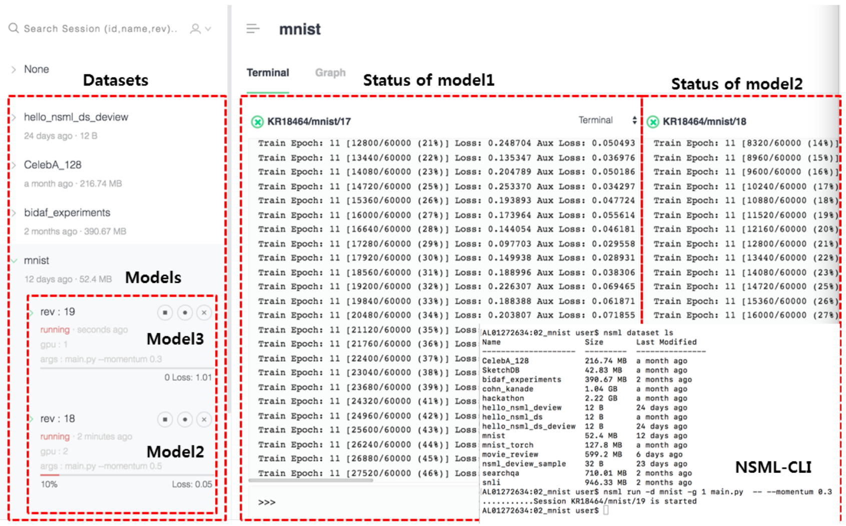
User Interface: The user interface (UI) controls the input and output flow between users and NSML. The UI is separated into the command line interface *NSML-CLI* and the *web UI*. *NSML-CLI* runs on users' local system and requires docker to prepare containers for NSML. *NSML-CLI* collects the model code in a container. Usage of *NSML-CLI* may be challenging for users that are not familiar with the command line. The *web UI* wraps *NSML-CLI* in a web application and is more intuitive and easy to use for these users. Furthermore the *web UI* provides visualizations such as graphs, logs, and demos.

3.3 Implementation Details

NSML is a decentralized cloud system except the scheduler. A user directly connects to a docker container via CLI once the container is scheduled. *NSML-CLI* is implemented in Python 3.

Research on machine learning consists of three main steps: preparation and preprocessing data, training models and testing models.

[5] https://www.minio.io/



```
Train Epoch: 11 [12800/60000 (21%)] Loss: 0.248704 Aux Loss: 0.050493
Train Epoch: 11 [13440/60000 (22%)] Loss: 0.135347 Aux Loss: 0.036976
Train Epoch: 11 [14080/60000 (23%)] Loss: 0.204789 Aux Loss: 0.050186
Train Epoch: 11 [14720/60000 (25%)] Loss: 0.253370 Aux Loss: 0.034297
Train Epoch: 11 [15360/60000 (26%)] Loss: 0.193893 Aux Loss: 0.047724
Train Epoch: 11 [16000/60000 (27%)] Loss: 0.173964 Aux Loss: 0.055614
Train Epoch: 11 [16640/60000 (28%)] Loss: 0.144054 Aux Loss: 0.046181
Train Epoch: 11 [17280/60000 (29%)] Loss: 0.097703 Aux Loss: 0.029558
Train Epoch: 11 [17920/60000 (30%)] Loss: 0.149938 Aux Loss: 0.028931
Train Epoch: 11 [18560/60000 (31%)] Loss: 0.188996 Aux Loss: 0.038306
Train Epoch: 11 [19200/60000 (32%)] Loss: 0.226307 Aux Loss: 0.069465
Train Epoch: 11 [19840/60000 (33%)] Loss: 0.188388 Aux Loss: 0.061871
Train Epoch: 11 [20480/60000 (34%)] Loss: 0.203807 Aux Loss: 0.071855
Train Epoch: 11 [21120/60000 (35%)] Loss
Train Epoch: 11 [21760/60000 (36%)] Loss
Train Epoch: 11 [22400/60000 (37%)] Loss
Train Epoch: 11 [23040/60000 (38%)] Loss
Train Epoch: 11 [23680/60000 (39%)] Loss
Train Epoch: 11 [24320/60000 (41%)] Loss
Train Epoch: 11 [24960/60000 (42%)] Loss
Train Epoch: 11 [25600/60000 (43%)] Loss
Train Epoch: 11 [26240/60000 (44%)] Loss
Train Epoch: 11 [26880/60000 (45%)] Loss
Train Epoch: 11 [27520/60000 (46%)] Loss

>>>
```

```
Train Epoch: 11 [8320/60000 (14%)]
Train Epoch: 11 [8960/60000 (15%)]
Train Epoch: 11 [9600/60000 (16%)]
Train Epoch: 11 [10240/60000 (17%)
Train Epoch: 11 [10880/60000 (18%)
Train Epoch: 11 [11520/60000 (19%)
Train Epoch: 11 [12160/60000 (20%)
Train Epoch: 11 [12800/60000 (21%)
Train Epoch: 11 [13440/60000 (22%)
Train Epoch: 11 [14080/60000 (23%)
Train Epoch: 11 [14720/60000 (25%)
Train Epoch: 11 [15360/60000 (26%)
Train Epoch: 11 [16000/60000 (27%)
```

```
AL01272634:02_mnist user$ nsml dataset ls
Name                 Size        Last Modified
-------------------- ---------   ---------------
CelebA_128           216.74 MB   a month ago
SketchDB             42.83 MB    a month ago
bidaf_experiments    390.67 MB   2 months ago
cohn_kanade          1.04 GB     a month ago
hackathon            2.22 GB     a month ago
hello_nsml_deview    12 B        24 days ago
hello_nsml_ds        12 B        a month ago
hello_nsml_ds_deview 12 B        24 days ago
mnist                52.4 MB     12 days ago
mnist_torch          127.8 MB    a month ago
movie_review         599.2 MB    6 days ago
nsml_deview_sample   32 B        23 days ago
searchqa             710.01 MB   2 months ago
snli                 946.33 MB   2 months ago
AL01272634:02_mnist user$ nsml run -d mnist -g 1 main.py  -- --momentum 0.3
...........Session KR18464/mnist/19 is started
AL01272634:02_mnist user$
```

Figure 2: Example of Web UI and NSML-CLI

NSML supports data preparation and preprocessing steps with web storage such as *minio*. *NSML-CLI* communicates with a storage container via *docker-cli* and Python libraries.

For training models, we first implemented a scheduler to distribute physical resources including GPUs efficiently. After allocation of resources, NSML builds a docker image for ML environments and mounts a dataset from a storage container. These two steps are bottlenecks of training models in NSML. We removed the first bottleneck by reusing existing docker images if a user needs the same environment. The other can be solved by sharing dataset directories among all ML containers when they are physically located at the same host machine. Then NSML runs trains the model by running the user provided code. During training NSML stores intermediate trained models into the storage container. With these backup files, NSML supports reproducing the same model and tuning hyperparameters during training. We implemented both hyperparameter tuning in training time and model evaluation through python read-eval-print loop (REPL), which allows a user to evaluate one simple line and print the result. With this feature, NSML can achieve hyperparameter tuning in training time by pausing user-written codes, downloading a model from storage container, and resuming the code. We used this feature not only to test and resume models, but also to demonstrate models as a service through *web UI*.

Our web servers run with *nginx* [6] and the *web UI* translates a user's input to *NSML-CLI* commands. For example the *web UI* allows testing trained models in real time by issuing the `nsml infer` command. For this command *nsml-CLI* creates new container with python REPL, and a user can run a demo of trained models through this new container.

3.4 Usage Examples

After installation, `nsml` commands can be invoked with `nsml [OPTIONS] COMMAND [ARGS]...` syntax. NSML has two main commands: `nsml dataset` and `nsml run`.
`nsml dataset` provides commands to examine and manage the list of datasets available in the NSML cloud environment as well as to add new datasets to the environment.
`nsml run` provides functionality to pack codes from the local workstation of the researcher and execute it in the cloud environment with access to a specified dataset that will be mounted at a predefined mounting point.

For example, `nsml run main.py -d [name of dataset]` would package the code in the current directory, send it to the NSML server, and then run `main.py` giving access to the specified dataset.

To ease the task of performing experiments, a client package that can be accessed from the experiment code is provided. It contains functions that allow experiment metrics to be logged via both

[6]https://nginx.org/en/

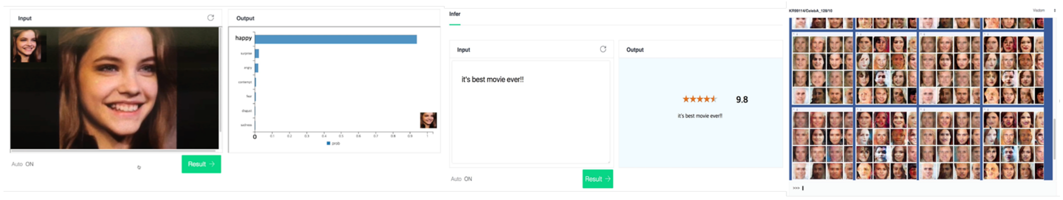
TensorBoard and Visdom. These logs can be accessed in raw form with `nsml logs [OPTIONS] SESSION` and visualized `nsml plot [OPTIONS] SESSION`, where `SESSION` denotes a session identifier returned by `nsml run`.

Furthermore, NSML contains a kaggle-like leaderboard on which experiment results on a specific dataset can be recorded and compared. `nsml dataset board [OPTIONS] DATASET` command gives access to the NSML leaderboard. Figure 2 display an example of *Web UI* and *NSML-CLI* on the MNIST dataset.

4 Experiments via Alpha Tests

4.1 Details of alpha tests

We operated NSML in a phase of alpha tests with three different machine learning tasks including GAN-based face generation, LSTM-based movie rate prediction, and CNN-based facial emotion recognition. Figure 3 illustrates the demonstration of the experiments on three datasets. As shown in Figure 3, users can conveniently develop their model and perform the experiments.



it's best movie ever!!

(a) Face emotion recognition (b) Movie rate prediction for reviews (c) Face image generation

Figure 3: Visualization of three real-world tasks

4.2 Interviews of alpha testers

During the alpha tests, we collected reviews from the alpha testers and summarized them into following items.

- The usage of NSML is more intuitive than other platforms. It is not hard to learn and understand the work flow of the system. And the NSML provides almost all the necessary functions such as multiple GPUs allocation and reproducing the past experiments. It makes it easier to focus on researching and developing the deep learning models.
- The visualization and history logging of experiments make it more efficient. We don't need to manually implement tedious jobs such as drawing a graph and saving a model.
- Currently, the system is little bit unstable. Sometimes the system has no response and has been recovered after a few minutes. Maybe the system is not ready for public services.

5 Discussion and Concluding Remarks

We demonstrate a new machine learning platform that enhance the research efficiency, NSML. For designing NSML, we investigate the challenges in machine learning research from a deep learning community. Then, we define the detailed requirements with six categories. NSML provide the functionalities corresponding to the requirements, thus enabling the researchers to focus on their model implementation and data.

Current version of NSML has one main limitations. NSML only supports data-driven machine learning tasks. Machine learning tasks such as reinforcement learning require an environment for agents as well as a dataset. This issue will be addressed by treating an environment in a way similar to that NSML treats a dataset. More AutoML features will be added in future.

More detailed information will be available at http://research.clova.ai/nsml.

Acknowledgments

The authors thank Hyunah Kim for data preparation, Doris Choi for discussion, and Miru Ryu for the web interface design.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] Sercan Arik, Gregory Diamos, Andrew Gibiansky, John Miller, Kainan Peng, Wei Ping, Jonathan Raiman, and Yanqi Zhou, "Deep voice 2: Multi-speaker neural text-to-speech," *arXiv preprint arXiv:1705.08947*, 2017.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al., "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International Conference on Machine Learning*, 2016, pp. 173–182.

[4] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten, "Densely connected convolutional networks," *arXiv preprint arXiv:1608.06993*, 2016.

[5] Joseph Redmon and Ali Farhadi, "Yolo9000: better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

[6] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin, "Convolutional sequence to sequence learning," *arXiv preprint arXiv:1705.03122*, 2017.

[7] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio, "A structured self-attentive sentence embedding," *arXiv preprint arXiv:1703.03130*, 2017.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[9] Soumith Chintala, "An overview of deep learning frameworks and an introduction to pytorch," 2017.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[11] Xuan Zou, "Understanding deep learning through visualization," 2017.

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

[13] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al., "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[14] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," *arXiv preprint arXiv:1609.08675*, 2016.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[16] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*. Boston, MA, USA, 2010, vol. 8, p. 9.