2022 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS) | 978-1-6654-7340-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/ANTS56424.2022.10227727

Heuristic-based Resource Allocation for Cloud-native Machine Learning Workloads

Ayush Shridhar and Deepak Nadig
Dept. of Computer and Information Technology, Purdue University, West Lafayette, IN 47907
Email: {ashridh, nadig}@purdue.edu

***Abstract*—As machine learning workloads become computationally demanding, there is an increased focus on distributed machine learning to train and deploy models across multiple machines in a cloud-native cluster. However, optimizing a machine learning model's lifecycle to facilitate efficient resource utilization is still an active area of research. The approach typically involves a manual effort to partition the models into distinct layers and decide how to store these distinct layers on a distributed computing framework. However, distributing distinct layers across nodes can induce a network latency bottleneck in the machine learning pipeline. Further, the above process becomes more inefficient as models become increasingly complex. In this paper, we present a heuristic-based approach to distributed model training. Further, we analyze the resource utilization metrics from a sample machine learning pipeline deployed on a KubeFlow MLOps framework testbed.**

***Index Terms*—Cloud-native Infrastructure, MLOps, Resource Allocation.**

I. Introduction

Machine learning involves learning trends, interpolating data patterns, and using this information to analyze previously unseen data. Deep learning is the branch of machine learning that refers to the use and study of neural networks for learning these patterns. Neural networks have shown massive performance improvements in various domains, from speech recognition to synthesis, from decision-making for self-driving cars to gaming (e.g., chess [1] and starcraft [2]). Training neural networks are computationally expensive and typically requires access to specialized hardware such as graphics processing units (GPU), tensor processing units (TPU) and field programmable gate arrays (FPGA) to improve performance. Cloud has become the go-to solution for obtaining on-demand hardware to train these models. The rise of cloud service providers, advances in container technologies and orchestration systems such as Kubernetes has made managing cloud infrastructure and compute resources easier. Containerization enables packaging an application with its dependencies and orchestration platforms to help manage containers effectively. Machine learning operations (MLOps) is a new field at the intersection of machine learning and DevOps. It aims to simplify the process of aggregating data, training and deploying models by leveraging core DevOps principles. MLOps platforms, such as KubeFlow [3], facilitate end-to-end machine learning pipeline development, promote the re-usability of independent components and enable engineers to focus on critical tasks.

Recent trends have shown that the scale of deep learning models is increasing. Language models, for example, increased from 94 million parameters in the Elmo model to 1.5 billion in GPT2 and 175 billion in GPT-3. Assuming each parameter takes 4 bytes, storing 175 billion parameters requires approximately 700 GB. Thus, the model size is over 10x that of a regular GPUs' memory (e.g., 48 GB for an Nvidia Quadro RTX 8000). The GPT-3 model requires about 3.14×10^{23} FLOPS during the training process, and basic calculations indicate that it would cost approximately 4.6 million USD to complete the entire training process [4]. We also observe a similar trend with large models used for other tasks. As machine learning (ML) models become more significant and extensive in scale, there is a need to develop efficient techniques to train these models, as it is impossible to fit the entire model on a single machine. Parallel or distributed machine learning deals with algorithms that distribute the models onto multiple machines (or GPUs) and combine the results to return a final trained model. However, utilizing infrastructure as a service (IaaS) hardware is expensive from a resource and cost perspective. As ML models become bigger, allocating optimal resources to these models is a critical problem. This paper explores various approaches for choosing the optimal resource for model training, model splitting, and fitting distinct parts onto separate machines to minimize resource wastage without adding significant overheads.

We study existing methods for distributed training of machine learning models in cloud-native infrastructures and develop a preliminary mathematical basis for resource allocation problems. We present a brief overview of the existing literature, identify how to model resource allocation as an optimization problem, and discuss machine learning-based approaches suitable for cloud-native deployments. While machine learning pipelines can have multiple steps, from data acquisition to preprocessing, training and post-processing, we focus on efficient resource allocation for each pipeline stage based on their computational needs.

This paper is organized as follows: Section II introduces the context for the need for cloud-native resource allocation and presents the related works; In Section III, we present current approaches to model parallelism; Section IV presents a heuristic-based approach to allocating resources to an ML pipeline in a cloud-native infrastructure. We also present preliminary results on the resource requirements for a sample ML pipeline deployed on an MLOps framework; We conclude our work and discuss the future directions in Section V.

Authorized licensed use limited to: Purdue University. Downloaded on October 28,2023 at 19:55:47 UTC from IEEE Xplore. Restrictions apply.

II. Related Work

Recently, machine learning operations (MLOps) in the cloud have gained significant attention, and numerous research efforts focus on optimizing machine learning (ML) pipelines for cloud deployments. While MLOps focuses on end-to-end lifecycle management of ML workflows by providing scalability and efficiency benefits, numerous challenges exist [5], [6]. Machine learning pipelines for various tasks have benefited many application domains, including the internet of things (IoT) [7], data-intensive science [8], and genetics [9]. Numerous efforts have explored the use of machine learning frameworks for resource allocation in various computing paradigms, including vehicular networks [10], network functions virtualization [11], optimization [12], [13], and cloud assistance [14].

A critical challenge across computing paradigms (e.g., on-premise, cloud, edge, fog, hybrid, and high-performance computing) is ensuring optimal resource allocation for diverse ML pipelines. Sub-optimal resource allocation for large ML pipeline deployments that use cloud-native MLOps frameworks may increase operational costs, impact overall performance and reduce deployment agility. Further, as artificial intelligence (AI) and ML model designs specialize in performing particular tasks, dynamic changes to the model, pipeline, or data artifacts may impose high costs on the compute infrastructure.

Existing solutions optimize an ML pipeline's resource allocation through manual tuning or worst-case resource requirement estimation. However, these approaches are inefficient and fail to exploit the resource scalability, observability and optimization benefits of cloud-native infrastructure for MLOps. In contrast to the above works, our paper explores various model parallelism approaches suitable for optimizing cloud-native resource allocation for ML pipeline deployments. In the following, we provide a brief overview of model parallelism approaches and how we can employ them to make intelligent resource allocation decisions in cloud-native infrastructures.

III. Distributed Model Training Approaches

Parallelism[1] in machine learning, alternatively, distributed machine learning, typically refers to achieving better (or faster) performance by making the best use of all available resources in a cloud-native setting. The above process typically involves multiple machines splitting the computation optimally and returning a final trained model. For example, if the model is larger than the memory capacity of a GPU, we can split the model into multiple components and place each layer on a different GPU. This approach, the most naive model parallelism approach, comes with certain drawbacks. First, in splitting models, the number of GPUs increases with communication overheads. This overhead can become a bottleneck if the GPUs are on different cluster nodes. Thus, the gradients calculated during back-propagation must travel back from the final layers to the first layer, which may result in increased latencies. Next, the decision to split layers employs a manual approach. Splitting models requires knowledge of the underlying deep learning model architecture and the available hardware. Lastly, another drawback of this approach is that it leads to the hardware idle for extended periods, commonly referred to as the GPU idling problem. While a particular GPU performs the necessary computation, all other provisioned machines are idle until the operation terminates. Therefore, this approach results in inefficient usage of computing resources. To address this problem, GPipe [15] proposed the idea of pipeline parallelism and worked around the GPU idling problem by splitting the data into smaller micro-batches, allowing all machines to compute concurrently. Although not optimal, it decreased the overall GPU idle time (which can be seen as the area of the bubble in Figure 1) compared to naive model parallelism. Naturally, this also adds another hyper-parameter to the model, i.e., the number of micro-batches that result from splitting the incoming batch.

[1]Parallelism, in this paper, refers to model parallelism and not at the individual operator level.

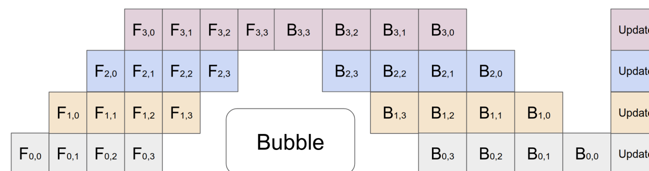


Fig. 1: Reduction of idle time with GPipe [15].

Another approach toward attaining parallelism involves splitting the data into smaller batches. Each GPU maintains a replica of the entire model but operates on different parts of the dataset. At the end of each backward pass, gradients from each GPU are accumulated and used to update each replica. This architecture needs a shared hardware pool that stores gradient information. We refer to this shared pool as the parameter server. The gradient update is performed synchronously and asynchronously, making it an efficient approach when the training data is extensive. However, these distributed training techniques have some drawbacks, and complex models may use a combination of data and model parallelization approaches to attain optimal parallelism.

ZeRO [16] data parallelism is another variation of data parallelism. The model on each machine stores only a slice of the model information rather than the complete model parameters, optimizers and gradient. This leads to lower storage requirements for the models and, therefore, fits more significant batch sizes per machine. During training, all machines are synchronized to share parameter and gradient information. Fully sharded data parallel (FSDP) [17] is another advancement in ZeRO that shards the models' parameters across multiple GPUs while also optionally offloading a part of the training to the CPU.

IV. Resource Allocation for Cloud MLOps

Resource allocation refers to selecting the optimal computing resources (e.g., GPUs) in the cloud and making suitable

Authorized licensed use limited to: Purdue University. Downloaded on October 28,2023 at 19:55:47 UTC from IEEE Xplore. Restrictions apply.

model and data splitting decisions. Directed computational graphs internally represent deep learning models, where the vertices represent operations, and edges indicate data flow. Hence, splitting a machine learning model is effectively similar to breaking the edges in its computational graph, so each independent component of the final graph can fit into a single machine. The vertices we choose to break will still constitute inter-machine communications since they connect two components stored in different GPUs. Thus, if the GPUs belong to different physical machines, it can lead to additional network latency. Moreover, we need to minimize GPU memory wastage to ensure optimal usage of available resources. At a very high level, resource allocation problems can be divided into two categories based on how this allocation is performed. Static resource allocation refers to the manual allocation of computing resources. This approach involves understanding the training model and available data and allocating the appropriate computing resources. After resource allocation, we decide on the mechanism to split the model. For instance, we may want to fit more layers into the GPU with more significant memory. Since this involves manual effort, resource allocation and splitting decisions become complex with large, parallel, and non-linear deep learning models. To mitigate this manual effort, we explore a heuristic-based approach that uses the knowledge of the model and available hardware to decide the best method to split the computation graph to attain model parallelism. Next, we present a heuristic-based approach to optimize resource allocations in cloud-native infrastructures.

A. Heuristic-based Resource Allocation for MLOps

Let $G = \{V, E\}$, be a random computational graph that completely represents our deep learning model, with V denoting the vertices or nodes of this graph and E indicating the connection between different nodes. We denote the set of available GPUs as $H = \{H_0, H_1, H_2, ..., H_{k-1}\}$. Each H_i, with $i \in \{0, ..., k-1\}$, has physical memory specifications defined in $S = \{S_0, S_1, S_2, ..., S_{k-1}\}$. Note that there are no restrictions on the above computational graph. The graphs may be nonlinear and have skip connections or recurrent nodes. Similarly, H may store the memory specifications of all available GPUs.

The heuristic-based resource allocation problem essentially reduces to the problem of splitting the graph G into a finite number of sub-graphs G_N. Thus, we have $G_N = \{g_0, g_1, g_2 ...., g_{N-1}\}$ by breaking a fixed number of edges $E_j = \{e_0, e_1, ..., e_{j-1}\}$ and splitting the original graph G into smaller graphs such that $G_N = g_0 \cup g_1 \cup ... \cup g_{N-1}$. Further, we add the edges E_j between different GPUs that connect these sub-graphs G_N. We also define a function *Lat* to represent the final latency function. The latency introduced by this operation is defined as $Lat = f(G, E_j)$. For a simple linear model, *Lat* returns the sum of all latencies introduced by breaking these edges in E. For a parallel graph model, the latency does not necessarily have to be the sum of individual latencies.

Our heuristic-based strategy proposes optimizing two values, namely (i) the overall latency *Lat*, and (ii) the available idle GPU space while training the model, as we do not want to allocate more resources than needed. Naturally, one way to minimize network latency *Lat* is by reducing the number of broken edges. However, this approach leads to memory overflow for large models.

The final optimization will be a function of overall network latency *Lat* and idle GPU memory. Since Lat and idle memory have common parameters, we cannot optimize each parameter independently. Thus, we can treat the objective function as a regular *Loss* function in machine learning that must be minimized. We use *Loss* to represent our loss function, and our objective function is given by:

$$\arg\min_{E_j \in E}[Loss(Lat(G, E_j), \sum_{i=0}^{N} S_i - used_space_i)] \quad (1)$$

for N machines, where s_i represents the physical specifications of the ith machine, and $used_space_i$ is the amount of RAM storage utilized by the model.

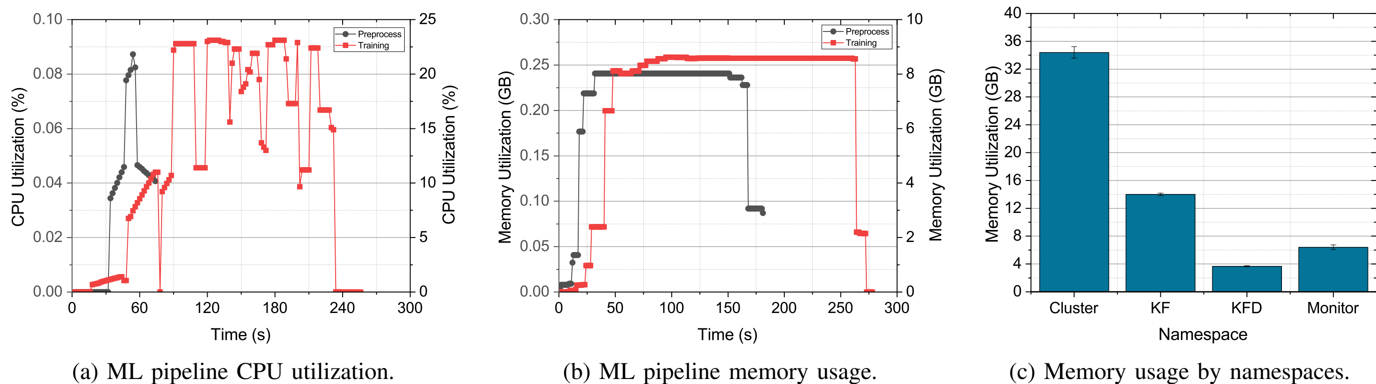
B. Preliminary Results

We employ a two-node Kubernetes cluster, each with 32 GB of RAM and an Intel i7 quad-core CPU for our MLOps framework deployment. Our testbed cluster runs KubeFlow to automate ML workflows and uses a monitoring framework to obtain accurate performance metrics at each pipeline stage. We use Prometheus as our metric server and Grafana for visualization. To test resource utilization, we deploy a sample pipeline consisting of four components: data loading, preprocessing, training and model inference. We focus our evaluations on the preprocessing and training pods, as they used most computing resources. Since each component runs as an independent pod in the KubeFlow pipeline, we extract accurate metrics of each pod using our monitoring system. In our sample pipeline deployment, we train a digit classification model using the Convolutional Neural Network (CNN) model built with TensorFlow.

During preprocessing, we drop an axis dimension from our data and reshape each vector into a square matrix. We then normalize the data by dividing each value by 255. Finally, we split our entire dataset into train and test sets, with a ratio of 9:1. In the training stage, we use a CNN consisting of three convolution layers with dropout layers between them, followed by a dense layer with softmax activation. The final output is a probability distribution of each possible label. The pipeline is generated using KubeFlow Python Software Development Kit (SDK) and uploaded to KubeFlow framework UI.

During our experiments, we noticed that CPU utilization peaked at approximately 1% for a short period during the preprocessing stage. However, the CPU utilization reached 23% during the training phase, as shown in Figure 2a. Similarly, memory usage during the preprocessing stage is considerably lower (at 0.24 GB) in comparison to the corresponding peak memory usage for the training phase (about 8.2 GB), as shown in Figure 2b. To understand the memory needs of our MLOps platform and associated dependencies, we measured memory

Authorized licensed use limited to: Purdue University. Downloaded on October 28,2023 at 19:55:47 UTC from IEEE Xplore. Restrictions apply.



(a) ML pipeline CPU utilization.

(b) ML pipeline memory usage.

(c) Memory usage by namespaces.

Fig. 2: Cluster and ML pipeline resource utilization.

usage over seven days. The KubeFlow namespace used 13.9 GB of memory, while its dependencies needed 3.6 GB of memory on average. The cluster memory consumption is much higher, at about 34.4 GB, as shown in Figure 2c.

V. Conclusions and Future Work

This paper presents a heuristic-based approach to the resource allocation problem in a cloud-native setting and reduces it to an optimization problem to minimize latency and GPU space utilization. We discuss current approaches for distributed model training and how the resource allocation process becomes cumbersome for large models. This motivates the need for a heuristic-based resource allocation strategy. Suppose we train a neural network to optimize the loss function shown in Equation 1; training such a model requires many subgraphs, E_j, which is impractical to train through a brute-force approach. Unless the GPUs communicate on the same cluster, any communication introduces latency overheads. Training the model in a supervised setting also requires training labels to calculate the loss. However, as our problem lacks labels, we explore unsupervised and greedy approaches to optimize this loss function. Reinforcement learning algorithms learn to navigate through complex environments without needing labeled data and purely through exploration while trying to optimize a long-term reward function. Similarly, unsupervised learning algorithms can learn correlations between model architectures and compute requirements. Our future work will utilize reinforcement learning algorithms to solve the cloud-native resource allocation problem.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," 2017. [Online]. Available: https://arxiv.org/abs/1712.01815

[2] O. Vinyals, I. Babuschkin, W. Czarnecki *et al.*, "Alphastar: Grandmaster level in starcraft II using multi-agent reinforcement learning." [Online]. Available: https://doi.org/10.1038/s41586-019-1724-z

[3] "Kubeflow." [Online]. Available: https://www.kubeflow.org/

[4] C. Li, "OpenAI's GPT-3 Language model: A technical overview," Sep 2020. [Online]. Available: https://lambdalabs.com/blog/demystifying-gpt-3/

[5] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, "MLOps - Definitions, Tools and Challenges," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, 2022, pp. 0453–0460.

[6] A. Banerjee, C.-C. Chen, C.-C. Hung, X. Huang, Y. Wang, and R. Chevesaran, "Challenges and Experiences with MLOps for Performance Diagnostics in Hybrid-Cloud Enterprise Software Deployments," in *2020 USENIX Conference on Operational Machine Learning (OpML 20)*. USENIX Association, Jul. 2020.

[7] Y. Zhang, J.-H. Liu, C.-Y. Wang, and H.-Y. Wei, "Decomposable Intelligence on Cloud-Edge IoT Framework for Live Video Analytics," *IEEE Internet of Things Journal*, vol. 7, no. 9, pp. 8860–8873, 2020.

[8] D. Nadig, B. Ramamurthy, B. Bockelman, and D. Swanson, "APRIL: An Application-Aware, Predictive and Intelligent Load Balancing Solution for Data-Intensive Science," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1909–1917.

[9] G. Minevich, D. S. Park, D. Blankenberg, R. J. Poole, and O. Hobert, "CloudMap: A Cloud-Based Pipeline for Analysis of Mutant Genome Sequences," *Genetics*, vol. 192, no. 4, pp. 1249–1269, 12 2012.

[10] I. Nurcahyani and J. W. Lee, "Role of Machine Learning in Resource Allocation Strategy over Vehicular Networks: A Survey," *Sensors (Basel, Switzerland)*, vol. 21, no. 19, p. 6542, Sep. 2021.

[11] R. Shi, J. Zhang, W. Chu, Q. Bao, X. Jin, C. Gong, Q. Zhu, C. Yu, and S. Rosenberg, "MDP and Machine Learning-Based Cost-Optimization of Dynamic Resource Allocation for Network Function Virtualization," in *2015 IEEE International Conference on Services Computing*, Jun. 2015, pp. 65–73.

[12] J. H. Joloudari, R. Alizadehsani, I. Nodehi, S. Mojrian, F. Fazl, S. K. Shirkharkolaie, H. M. D. Kabir, R.-S. Tan, and U. R. Acharya, "Resource allocation optimization using artificial intelligence methods in various computing paradigms: A Review," 2022. [Online]. Available: https://arxiv.org/abs/2203.12315

[13] F.-L. Luo, "Machine Learning for Optimal Resource Allocation," in *Machine Learning for Future Wireless Communications*. IEEE, 2020, ch. 5, pp. 85–103.

[14] J.-B. Wang, J. Wang, Y. Wu, J.-Y. Wang, H. Zhu, M. Lin, and J. Wang, "A Machine Learning Framework for Resource Allocation Assisted by Cloud Computing," vol. 32, no. 2, pp. 144–151, Mar. 2018.

[15] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[16] D. Team, R. Majumder, and J. Wang, "Zero & deepspeed: New system optimizations enable training models with over 100 billion parameters," May 2020. [Online]. Available: https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

[17] Y. Zhao, "Introducing Pytorch Fully Sharded Data Parallel FSDP API." [Online]. Available: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Authorized licensed use limited to: Purdue University. Downloaded on October 28,2023 at 19:55:47 UTC from IEEE Xplore. Restrictions apply.