INM432 Big Data Coursework

Elnara Mammadova (210026788)
https://colab.research.google.com/drive/1_FAdoDNE2SnyjFUuL9-4RH1IFtozXERU?usp=sharing

Task 1 – Write TFRecord files to the cloud with Spark

1d. Optimization, experimentation, and discussion (14%)

After compiling all our functions in a script file, we parallelize the data pre-processing in Spark using Google Cloud (GC) Dataproc. Three different cluster configurations (with 8vCPUs, SSD capacity = 500GB, and Standard disk capacity = 4096 GB) were tested to run PySpark jobs on the cloud:

- **Single Cluster** – Single machine with eightfold recourses (n1-standard-8) and max SSD of 500GB.
- **Maximal Cluster** – 1 master and 7 workers nodes each with 1 virtual CPU (n1-standard-1). Master with full SSD capacity (500GB) and 7 worker nodes with equal shares of the standard disk capacity to maximize throughput (4096 GB / 7 ~= 585 GB).
- **Four Cluster** – 4 machines (1 master, 3 workers) with 2vCPUs each (n1-standard-2). Master full SSD capacity (500GB) and 3 worker nodes with equal shares to the standard disk capacity to maximize throughput (4096GB / 3 ~= 1365GB).

By default, when we created an RDD from a TensorFlow Dataset object, PySpark returned 2 numPartitions during the initial call to parallelize. We further increased our partitions to experiment with different cluster configurations to demonstrate the difference in cluster utilizations. As seen in Fig. 1, CPU Usage for Maximal Cluster, for partition values of 2 and 4, we don't fully utilize our cluster resources (3 out of 7 workers not utilized), while also leading to less parallelism. Additionally, with few partitions our jobs took longer as each partition took more time to complete. Based on the experimentation we can observe the partitions in spark should be decided based on the cluster configuration. In our Maximal Cluster example, our cluster has 8 machines (1 master, 7 workers), therefore we want our RDDs to have 8 cores at the very least or 2 or 3 times of that.

For network bandwidth allocation in Figure 2. it can be observed that the most sent and receive bytes is associated with higher number of partitions for both maximal cluster and four machine cluster. However, in maximal clusters, we can also observe that for partition size 2 the bandwidth has not been allocated fairly between the master and the workers. As seen in the figure, the bandwidth allocation in the single cluster is higher because a single node is trying to handle all the work. In comparison, the bandwidth for each worker in eight and four machine clusters is lower because the jobs are distributed into smaller chunks between the workers where they each handle less work.

In this project we use mapPartitionsWithIndex() transformation function which is applied to each partition of an RDD while also tracking the index of the original partition. It returns multiple elements of the resultant RDD corresponding to the number of partitions in the RDD. The mapPartitions() are a specialized map that is called only once per each partitions, where

the entire content of the respective partition is available as a sequential stream of values via the iterator. The classical example of most standard application is a map() function, which performs a similar transformation, but instead of each partition, the function is applied to each element of RDD. The main advantage of using mapPartitions() is that the performance is improved since the object creation is eliminated for each and every element, as in map transformation. This in turns provides facility to do heavy weighted initialization on larger dataset.

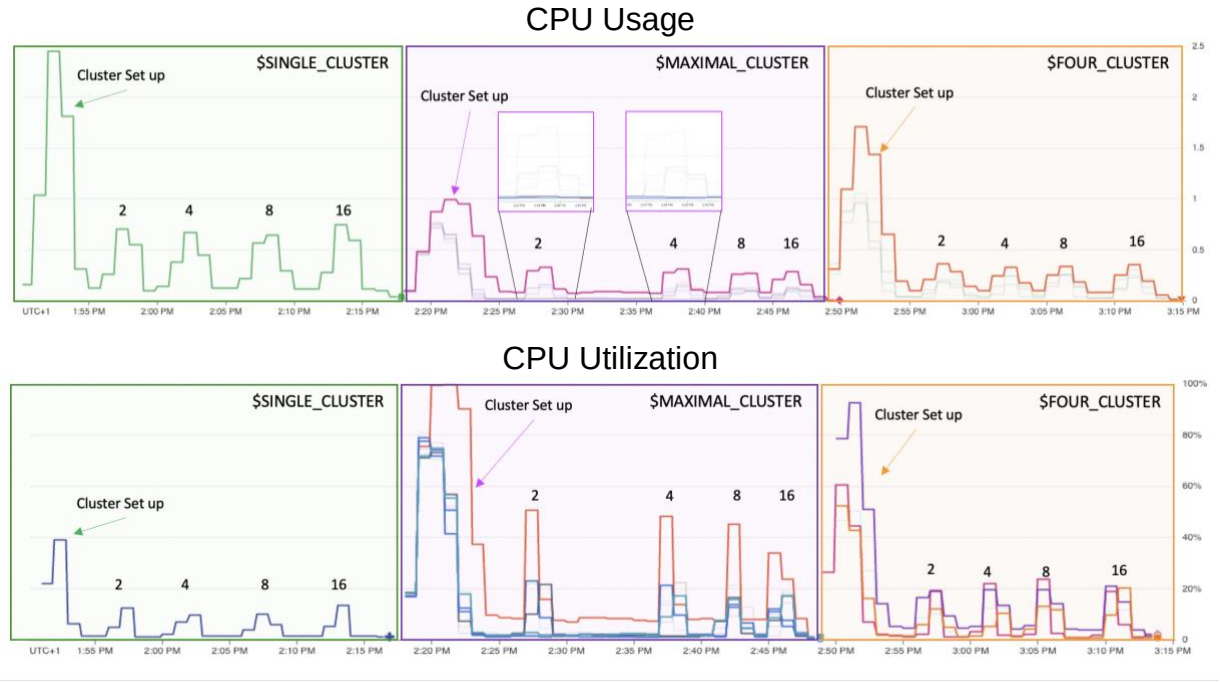

Figure 1. CPU Usage and Utilization across three different cluster configurations

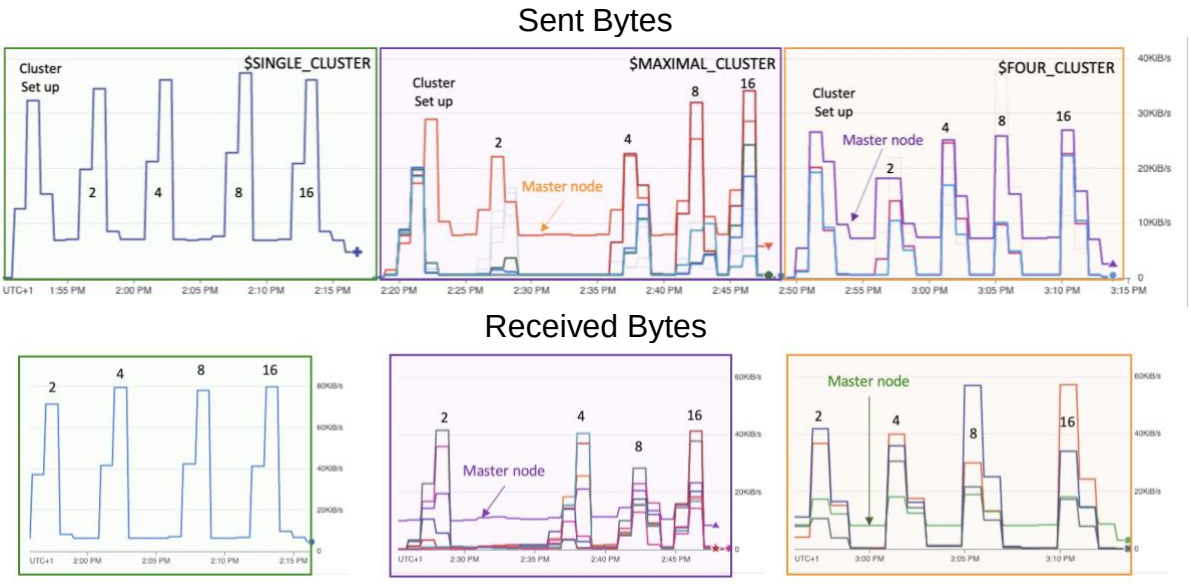

Figure 2. Network bandwidth allocations across three different cluster configurations

Task 2 – Speed Test

In this task we implement code for time measurement to determine the throughput in images per second. We used multiple parameters in parallel with Spark:

```
batch_sizes =  [8, 16, 32, 64]
batch_numbers = [10, 30, 60]
num_repetitions = [3]
```

2d) Retrieve, analyze and discuss the output (10%)

In Spark transformations are not executed until an action is called, however, an RDD is being re-evaluated again each time action is invoked on that RDD. Caching is one mechanism that temporarily stores the RDD in memory, which speeds up the application allowing access the same RDD multiple times. RDDs execute all processing steps at an action, therefore we cache prior to an action to prevent redundant execution by storing intermediate results in the memory. We also applied partitioning (16) along with caching, to optimize the performance. Since TFRecord files are relatively quick to run, in order to evaluate the effects of the cache in our speed test experiment, we have added redundant actions (i.e. count(), take()) after each cache, and run both locally and on cloud (4 machines double resources each). As seen in table 1, the script runtime for TF records has decreased by almost 60% when run locally, and 40% when run on cloud. No extra actions were performed in image files, since reading from images takes much longer than TFRecords.

Figure 3 shows a simple linear regression over avg. reading speed results for batch sizes and batch numbers obtained from the script that was run on the cloud. As seen on the figure, there is a positive correlation between both parameters and the throughput. However, we observe higher slopes on image file results, indicating that the two parameters have higher impact on the reading speeds for image files than for TFRecords.

Table 1. Speed Test results for TFRecords and Image Files

	Local		**Cloud**	
	Test Runtime	Script Runtime	Test Runtime	Script Runtime
TFRecords	0.00062	122.59	0.004	87.04
TFRecords_cache	0.0011	49.98	0.004	53.42
ImageFiles_cache	0.0004	927.24.	0.006	1363.88

Table 2. Avg. Speed results per Batch size and number (Google Cloud)

	Batch Size				**Batch Number**		
Avg. Speed	8	16	32	64	10	30	60
TFRecords	496.29	614.23	670.80	675.75	550.45	622.06	671.29
ImgFiles	9.89	9.13	10.83	10.44	9.65	10.11	10.46

To further understand the implications of these parameters on the throughput we perform a simple linear regression (with log values for throughput since it better depicts the relationship between the values for each parameter). Figures 4 and 5 show the results of linear regression over each parameter on the dataset read from image files and TFRecord files respectively, and table 5 and 6 shows the output of our linear regressions. As seen in both tables, slope is positive for all the parameters except repetition. R-values are highest for Datasizes (batch_size x batch_number) for both TFRecords and Image Files.

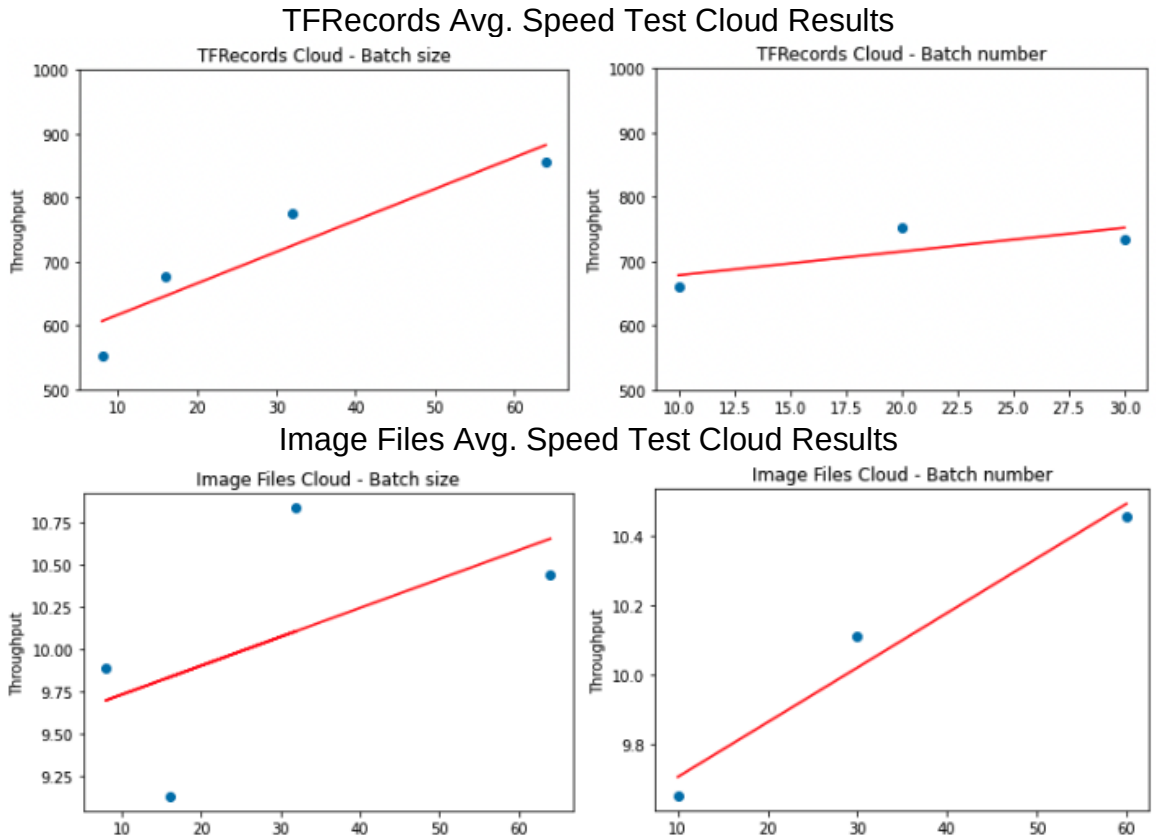

Figure 3. Simple Linear regressions on Speed Test result run in Google Cloud

Based on the results obtained from linear regression we can infer further that for large-scale machine learning applications, the datasize (product of batch-size and batch-number) has the most implication on the performance. This proves that working in small chunks (larger batch size and batch number) of data is essential for faster large-scale machine learning. However, there are other implications to keep in mind when working with large datasets. Latency, which is the time it takes for data to get to its destination across the network, can significantly increase the iteration time when addressing large-scale data sets (i.e. if our data is stored somewhere far, our latency will be higher). For example, for the case of single packet to be sent California->Netherlands->California, (latency-numbers document) the time required for the trip is equivalent to 150ms, which is already ~0.3% of our total script runtime on cloud (refer to table 1). In reality, bandwidth is one of the many factors that contribute to the performance. The lower the bandwidth, the more time it will take for packets to travel, which can result in higher latency, especially if the packets are huge. Hence, an efficient network connection consists of both low latency and high bandwidth, which in turns allows the maximum throughput. Therefore, even though our results indicate the reading speeds to increase with both high batch size and batch number, there might be a condition where this relationship may no longer exist

(i.e. we can increase bandwidth only by a finite amount, as latency will eventually create a bottleneck, and limit the amount of data that can be transferred over time).

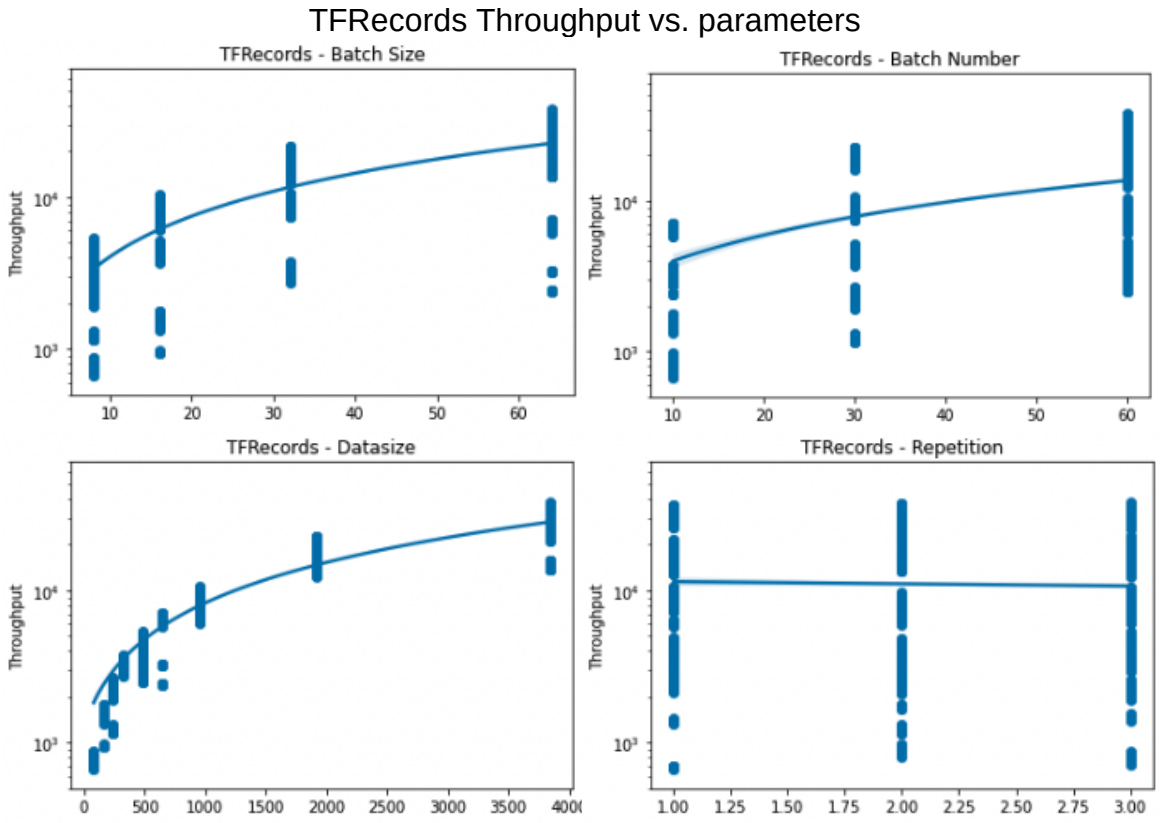

Figure 4. Linear Regression results for TFRecords

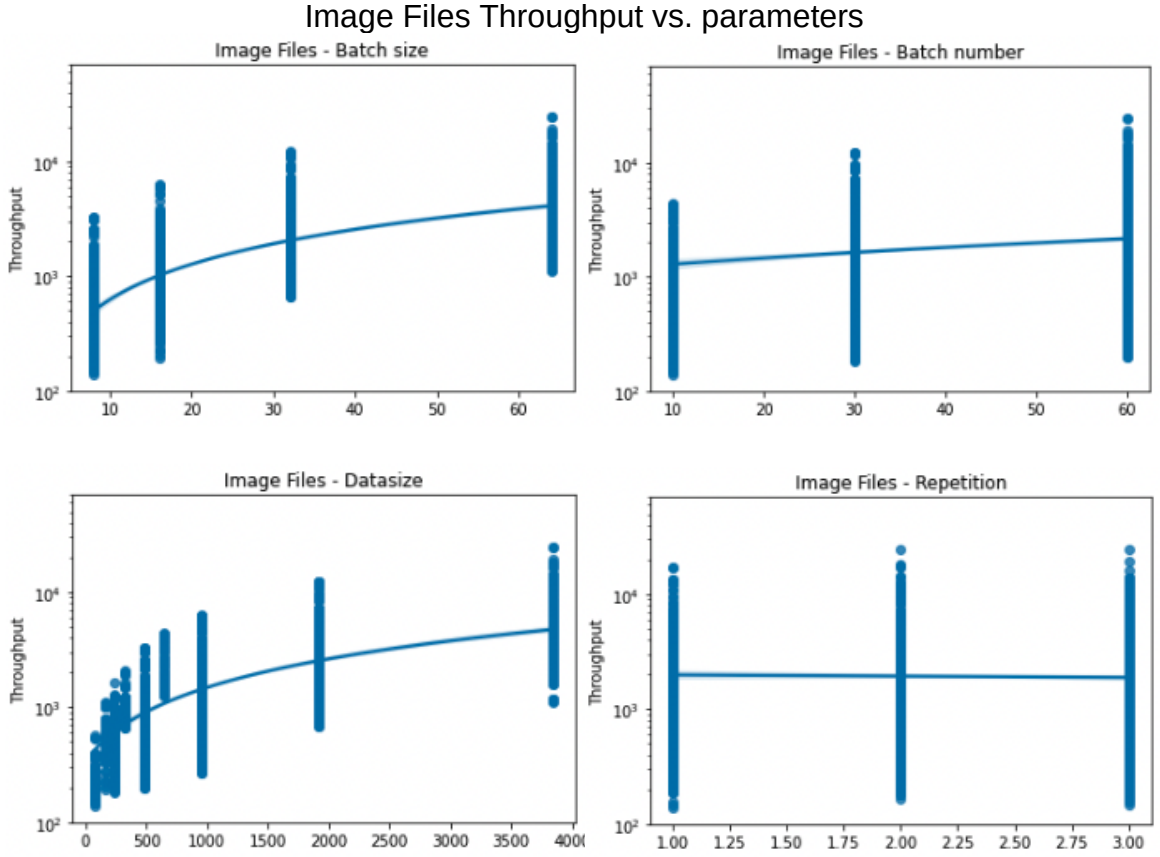

Figure 5. Linear Regression results for Image Files

Table 3. Linear Regression results for TFRecords.

TFRecords	slope	intercept	r-value	p-value	stderr
Batch Size	0.015	2.6	0.75	0.0	0.00027
Batch Number	0.0034	2.9	0.14	8.79	0.00049
Datasize	0.00025	2.71	0.686	0.0	5.44
Repetition	-0.013	3.09	-0.022	0.292	0.012

Table 4. Linear Regression results for Image files

Image Files	slope	intercept	r-value	p-value	stderr
Batch Size	0.015	3.434	0.77	0.0	0.00025
Batch Number	0.011	3.36	0.5	5.165	0.0004
Datasize	0.0003	3.47	0.86	0.0	3.63
Repetition	-0.0056	3.89	-0.01	0.61	0.011

For single machines (1 machine with eightfold recourses) we would expect similar plots depicted above (results from 4 machines with double the resources each). However, since the jobs cannot be parallelized among workers, we would expect the lower reading speeds (image per second) for both TFRecords and image files.

Task 3 – Machine Learning in the cloud

3c) Distributed Learning (10%)

Figure 7. Machine Learning in cloud using two different strategies.

In this section we experimented with different distributed learning strategies (Mirrored-Strategy, Multi-Worker-Mirrored-Strategy, Experimental-Multi-Worker-Mirrored-Strategy) and batch sizes (16, 32, 64) on our ML model. While analyzing the batch sizes, the worst

accuracies corresponded with the batch sizes of 16 and 64, making 32 the most suitable batch size for our model. When comparing between the strategies however, even though MirroredStrategy showed highest accuracies for both training and the validation, the validation accuracies are higher than training, which indicates that there is a high chance our model is overfitting. Hence, the best performing strategy is MultiWorkerMirroredStrategy with 32 batch sizes, up until epoch 15, after which our validation accuracies exceed of the training.

Task 4 – Theoretical discussion

4a) Contextualize (10%)

For this coursework we only worked with limited number of cluster configurations due the nature and the size of the project. But in general, Google provides 18 types of VM instances while also allowing customizing VM's memory and the number of CPU cores. So, when setting up a cluster configuration one needs to carefully find the right number of VMs, CPU count, CPU speed per core, average RAM per core, disk count, disk speed and network capacity of a VM [1]. Finding the right cloud configuration is essential in terms of performance (running time) and the cost. The first paper by Alipourfard et al (2017) the authors discuss the benefits of using CherryPicking method (Bayesian Optimization) to build performance models for various applications vs. the two strawman solutions (exhaustive search, and accurate modelling) both of which have high time and cost associativity. We can utilize CherryPick method for both task 2 and 3 to obtain just accurate enough configurations but much faster and at lower cost. Furthermore, since reading TFRecord files is a recurring job where similar workloads are executed repeatedly, CherryPick can achieve both low overhead and adaptivity.

Kahira, A.N. (2021) introduces a ParaDL tool, capable of modeling and predicting the performance of a large set of configurations for CNN distributed training at scale [2]. The task 3 of this project utilizes CNN model in which ParaDL can be used to find the best distributed learning strategy.

4b) Strategize (10%)

Batch processing is where data is collected over time and sent for processing in batches. Since batch processing can be lengthy, it is meant for large quantities of information that aren't time sensitive. In contrast online (mini-batches) and stream (piece-by-piece) processing are faster and are meant for information that's needed immediately. The two concepts discussed in the papers provided aren't suitable for all three processing: i.e. CherryPick is more suitable for online and stream processing (recurring jobs), whereas ParaDL is more suitable for batch processing as it requires large amount of data in place.

References:

[1] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., & Zhang, M. (2017). Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In USENIX NSDI 17 (pp. 469-482).

[2] [Kahira, A.N. (2021). An Oracle for Guiding Large-Scale Model/Hybrid Parallel Training of Convolutional Neural Networks In Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (pp. 161-173).

Number of words = 2064