# Distributed Training Recommendation System

812 Distributed System - Project Report - **Group 9**

Jésus Garcia
*Computer Science and Engineering*
*Michigan State University*
East Lansing, Michigan, USA
garci566@msu.edu

Sinuo Fan
*Electrical and Computer Engineering*
*Michigan State University*
East Lansing, Michigan, USA
fansinuo@msu.edu

Yijiang Pang
*Computer Science and Engineering*
*Michigan State University*
East Lansing, Michigan, USA
pangyiji@msu.edu

*Abstract*—**In this paper, the goal of our group is to create a system that distributes the effort required to train a model to provide recommendations for user project scenarios. Specifically, we will distribute the training required for the gradient descent model across nodes and evaluate it against a statistical baseline model and compare it to a non-distributed system. In addition, we also plan to implement two features that fall within the scope of distributed systems. Our first feature aims to protect recommend systems from backdoor attacks through distributed algorithms. Also, the second function will try to spot performance anomalies in the system and try to avoid slower performing nodes and optimize training efficiency. The difference from other related work is that we will utilize Docker containers and images to distribute the training process to native CPUs. According to our investigation, there are no related projects in the whole network to compare.**

*Index Terms*—**Distributed training, Backdoor attack, PyTorch, Docker**

## I. Introduction

With the development of machine learning, data analysis and big data, the demand for computing resources seems to be increasing all the time. But when we train a model, it takes a dozen hours or even days. We'll start thinking about how to increase our computing power to save time. Of course, we can't have supercomputers for every user or business. So distributed systems play a huge role in saving time. What is distributed computing? A distributed system is a collection of independent computers that appear to the user as a related system. This saves time by assigning the entire task to different computers to do the calculations and then using the data as a whole. From the process point of view, two programs run on the processes of two hosts respectively, cooperate with each other, and finally complete the same service (or function), then theoretically, the system composed of these two programs can also be called "distributed" System " "". One reason to use a distributed system is scalability. After all, any host has a performance limit. A distributed system can scale both horizontally and horizontally by continually expanding the number of hosts.

According to IBM[4] in 2020, each person generates 1.7 megabytes of data per second. Companies like Netflix leverage this massive data generation to build sophisticated recommendation algorithms that save over 1 billion dollars annually in customer retention alone [7]. Such a large dataset certainly has the ability to train different machine learning models to make predictions, but by restricting the recommender system to a single node, we severely delay the required training time. In this project, we aim to implement a distributed recommender system that generates item-specific rating predictions for each user using a user/item rating matrix. Additionally, we plan to implement a distributed algorithm to (1) defend against backdoor attacks that could compromise the accuracy of the system, and (2) monitor performance anomalies and automatically set itself to the optimal configuration required by the dataset.

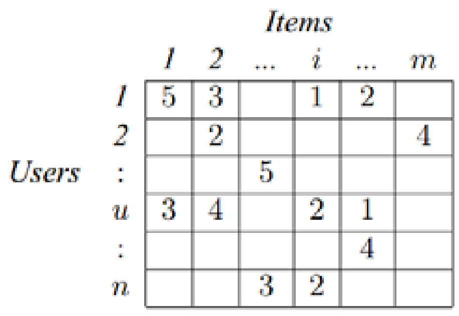
| Recommendation Model | Runtime (seconds) |
|---|---|
| Gradient Descent | 21,629s $\approx$ 6hrs |
| Statistical Computation | 21.226s |

Fig. 1. Runtimes of model training on a single PC. The hardware specifications for this computer are a Ryzen 2700x 3.7GHz CPU, Asus ROG GTX1070 GPU, 16 GB DDR4 RAM 3200MHz.

The main reason our group is interested in distributed gradient descent training is because the training time spent on a single PC is not enough. As shown in Figure 1, PC took over 6 hours for a subset of the Netflix dataset consisting of 500 users and 17,000 movies. We also tested the MovieLens dataset for comparison. MovieLens 25M movie ratings was chosen as a stable benchmark dataset. 162,000 users applied 25 million ratings and 1 million tags to 62,000 movies. By distributing the training workload to different CPUs, maybe we can reduce the training time required, one of our goals is to see if the time reduction is proportional to the number of nodes we will use to train on the entire dataset. For example, if it takes about 6 hours to train a gradient descent model on a subset of the entire training data, does that mean that three computers will be able to do the same amount of training in a third of the time?

## II. Related Work

Pozo et al (2015) introduces a novel strategy to distribute stochastic gradient descent on a Hadoop cluster using map-reduce.The break down their implementation into the following three main sections. First they perform matrix block decomposition, in matrix block decomposition, the data is



| | | Items | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | ... | i | ... | m |
| | 1 | 5 | 3 | | 1 | 2 | |
| | 2 | | 2 | | | | 4 |
| Users | : | | | 5 | | | |
| | u | 3 | 4 | | 2 | 1 | |
| | : | | | | | 4 | |
| | n | | | 3 | 2 | | |

Fig. 2. An example of the formatted table structure [10], where columns are movie IDs, rows are user IDs and the value at Matrix[row, col] is a users rating for item i on a scale from 1 to 5 inclusive.

decomposed into a block, a block is a batch of ratings that should be computed by only one processor, this allows for parameters and other program variables to have no dependency problems from within the same processor. These independent blocks can then be computed in a parallel manner. This then leads to stratum assignment, a stratum is defined by them as a set of independent blocks, these stratum are actually the number of nodes and so an N number of nodes or N number of blocks can run in parallel. Since the blocks inside the stratum are independent then the stratum also does not experience any dependency issues. Finally this ends in stratum execution, stratum execution applies stochastic-gradient-descent on stratum blocks and updates parameters at the end of execution. By updating parameters last the integrity and independence of parameters are guaranteed for next stratum. Our work plans to maintain the same type of independence that is seen in this paper in the sense that nodes work independently to train on the same dataset as other nodes, these nodes then proceed to communicate to continue the training synchronously.

Current Netflix recommendation systems as seen in Gomez-Uribe et al (2015) rely on statistical and machine learning techniques. Some of their algorithms to create personalized recommendations include personalized video ranker (PVR), Top-N Video Ranker and Video-Video Similarity. These aforementioned algorithms vary from our approach since they all work simultaneously to provide the best recommendations while the client uses the Netflix application.

**Difference with existing techniques** Our project is similar with federated learning in some ways [2], [3]. However, since the focus of federated learning is decentralized data, our project focus on distributed training, and it manages training data points allocation for each node considering capability and stability of nodes. Besides, without a large set of candidates, our project requires fixed workload of each updating step, and our program will handle unexpectations under the framework, such as re-allocating workload.

## III. The Datasets

In this system implementation we utilized two of the original three different datasets we proposed to test the recommendation system. The two different datasets are sourced from Netflix and MovieLens. Our smaller dataset is the MovieLens dataset, in this data there is approximately 162,000 users and 62,000 movies and 25 million ratings. The biggest dataset is Netflix, this data contains over 100,000,000 ratings from 480,000 users and 17,000 movies, in this case the data was collected from 1998 to 2005. For all datasets the rating values are the same, this is 1-5 inclusive. In figure 2 we can see the layout of the user/item/rating dataset where columns are items, rows are users and the value represents the rating. Unfortunately all three datasets suffer from sparsity, they're all over 90% sparse meaning that most values are NaN. Although this might seem like a big issue, rating datasets often tend to be mostly empty since ratings are not required by users so this does not pose as a big obstacle for us. For each dataset we will split the train and test set into a 80/20 split respectively.

### A. Big Dataset Challenges

Working with large datasets has always posed researchers with different challenges. In our project working with review data we had an abundance of it, due to Random-Access Memory constraints we had to significantly reduce the dataset we were able to use for our testing. The way our data preprocessing works is that the program attempts to locate a PICKLE file in the local directory, if that file is not found then it means that the raw data from the dataset sources has not been processed. The program then formats the data into a data structure as seen in figure 2.

Due to the simulated nature of our program (refer to section 5), we continuously ran into stability issue when attempting to run the program in different host machines. For our Netflix data we were able to successfully pre-process the entirety of the dataset this includes all 100M reviews but because of the stability issues we reduced the MovieLens to 5M ratings and we encountered some strange anomalies we will go over in the results.

## IV. The Original Recommendation System

Previously one of us implemented a recommendation system to work for the Netflix dataset [1], [5], [7]. This recommendation system was composed of two models to generate rating prediction, a statistical computation model and a gradient descent model. In the statistical approach, we utilize user and item biases to generate predictions using mean row-wise and column-wise mean operations. We can see the formula use to create the prediction in equation 1, here the $\hat{r}_{ui}$ is the predicted rating for user u on item i, $\mu$ is the global mean, $b_u$ is the deviation of the average rating of user from $\mu$ and $b_i$ is the deviation of the average rating of item $i$ from $\mu$.

$$\hat{r}_{ui} = \mu + b_u + b_i \tag{1}$$

$$f(b_u, b_i) = \sum_{(u,i)\epsilon T} (r_{ui} - \mu - b_u - b_i)^2 + \lambda(\sum_u b_u^2 + \sum_i b_i^2) \tag{2}$$

In the gradient descent approach the attempt to minimize the loss function seen in equation 2. In equation 2 the $\mathrm{r}_{\mathrm{ui}}$ is the

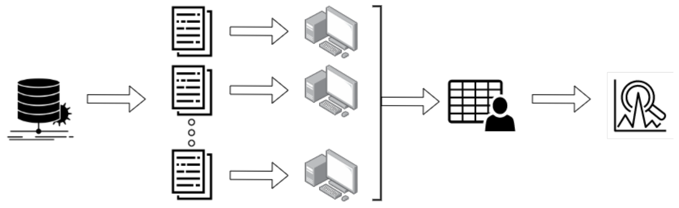


Fig. 3. In this figure we can see the general data flow of the recommendation system, the data will be broken up evenly among N amount of nodes to be trained for the gradient descent model. Once all of the nodes work on a single iteration of training they will then simultaneously move on to the next iteration. After all iterations have completed the data will be merged back together an evaluated for prediction accuracy through Mean Absolute Error (MAE).

rating for user u and item i, $\mu$ is the global mean, $b_u$ is the deviation of the user u from their own mean rating, $b_i$ is the deviation of item I from its own mean rating. Lambda is a hyper-parameter to avoid over-fitting.

## V. Initial Hypothesis

Initially, we naively thought that the algorithm runtime will decrease linearly as we add more nodes to the cluster but Yu et al. (2014) showed us different results from their experimentation. In their paper there seems to be a linear behavior with the amount of cores they used and the speedup of their total runtime. Although they are not taking a distributed approach, this can be compared to our approach because of the way our simulated distributed system is designed, the more cores we assign to a docker container in our training system the more it speedups as we will see in the results.

After experimentation we realized that we did not account for inter-node communication overhead which can drastically change the runtime. Since our initial models under the original recommendation system are not the same used in our distributed approach we figured that the new convolutional neural network would be much more optimized than the original gradient descent model. Additionally, we expected for the model to take hours to train since the Netflix and MovieLens dataset are substantially bigger than the subset originally used.

## VI. Design Rationale & Distributed Recommendation System

There are two popular distributed training framework, distributed training with PyTorch [6] and TensorFlow. In this project, PyTorch will be used to accelerate our project implementation. The distributed packages included in PyTorch enable researchers and practitioners to easily parallelize their computations across clusters of processes and machines. To do this, it utilizes message passing semantics, allowing each process to communicate data with any other process.

Our system will utilize Docker to simulate nodes that will be utilized for training, these nodes were configured to each use one core of the hosts CPU. Although this is technically not distributed throughout many machines, the docker containers can be very easily modified to run using the full capability of many host machines. For example, in our experimentation we distribute the recommendation process to up to three containers, each container runs in an isolated manner but still communicates with other containers to finish epochs in parallel. This setup is not ideal for large scale situations but proved useful in our use cases. Theoretically, each container can be placed into a different hardware resource pool (server, PC etc), in this scenario PyTorch can communicated through the Local Area Network to provide inter node communication. Like mentioned in the dataset section, handling large datasets requires some significant Random-Access Memory capacity, this can pose several limitations to what can be accomplished by the system and because of this we recommend that systems that host our recommendation system have at least 128Gb of RAM available.

### A. Several options for distributed training

Data parallelism and model parallelism are the popular options for distributed training. Data parallelism replicate a single model on multiple devices or multiple machines. They each process different batches of data and then combine the results. There are many variants of this setup, different model copies merge differently, do they stay in sync on each batch, or they are more loosely coupled, etc. Under option of model parallelism, different parts of a single model run on different devices, working together on a single batch of data. This is best for models with naturally parallel architectures, such as those with multiple branches.

### B. Comparison when we having more computers

However, even with the acceleration of distributed training, the project may face difficulties. For example, clients may have unstable performance, such as computational power is different from one to another. Distributed system is vulnerable to attacks. In real-world applications, clients may be controlled by malicious people. Besides, since clients may have various operation system, we have the risk of facing unstandardized configuration problems.

## VII. Docker

Docker is a tool that allows developers, system administrators, etc. to easily deploy their applications in a sandbox (called a container), running on a host operating system, such as Linux. The main benefit of Docker is that it allows users to package an application and all its dependencies into a standardized unit for software development. Unlike virtual machines, containers do not have a high overhead, so they can use the underlying systems and resources more efficiently.Containerization technology has been transformed to Docker. Developers can use the virtualization platform to create self-contained, lightweight, and portable software containers that make developing, testing, and deploying applications simple and quick. While Docker is reasonably simple to learn, two of its most important features may be challenging to grasp at first: Docker containers vs. Docker images. A Docker image is a read-only, inert template that includes container deployment instructions. Almost everything in Docker revolves around images. An image is made up of a group of files (or layers) that contain all of the components needed to set up a fully working container environment, such as dependencies, source code, and libraries. A Docker registry, such as the Docker Hub, or a local registry are used to store images. A Docker container is a vitalized run-time environment that allows you to isolate the execution of your applications from the underlying system. It's a Docker image in action. Containers are the pinnacle of Docker technology, as they provide a lightweight and portable environment for installing applications. Each container is self-contained and runs in its own environment, ensuring that it does not interfere with other applications or the underlying system. The security of apps is considerably improved as a result of this. Several container states are defined by Docker, including created, resuming, running, paused, exited, and dead. A container does not need to be operating because various states are available because a container is only an instance of the image.

We will try to use Docker to simulate our running environment Standardise configuration, software dependencies, etc. Although we were not able to fully implement a feature that defends against unstable nodes, we were able to create a hardware monitor so that the master node has information for all nodes in the cluster. train.py is a python script that ingests and normalizes EEG data in a csv file and trains two models to classify the data. inference.py is a python script that ingests and normalizes EEG data in a csv file and trains two models to classify the data. Linear Discriminant Analysis and Neural Network Multilayer Perceptron are two models saved by the script. To do batch inference, run inference.py and load the two models you constructed earlier. The software will use the csv file to normalize the fresh EEG data, do inference on the dataset, and publish the classification accuracy and predictions. Using the jupyter/scipy-notebook image as our base image, we'll make a simple Dockerfile. To support serialization and deserialization of our trained models, we'll need to install joblib. We copy the train.csv, test.csv, train.py and inference.py files into the image. We next run train.py, which will fit and serialize the machine learning model as part of our image building process, allowing us to debug at the start, use the Docker Image ID for tracking, and use different versions.

### A. *Docker Approaches*

We used the following ways to solve the problem: we downloaded the PyTorch/PyTorch image from Docker (approximately 2.6 GB), created four nodes that ran the same image and our machine learning script (Netflix Dataset), built the dependencies of matplotlib, tk, and pandas in each environment, and then updated those libs. Then, for each environment, we install libsm6, libxext6, and libxrender-dev (container). We go to the file folder and copy the workspace to each node/container, unload our main file from each node, then sign the last node for resource utilization to show how much cup we use for each node. You can refer to the pseudocode to launch the project in the appendix A.

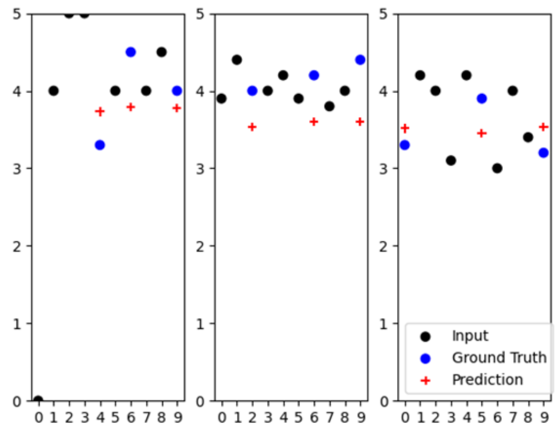


Fig. 4. The figure above shows how the models predictions are within close proximity of the ground truth values for the Netflix dataset. Although are recommendation system does not really focus on the quality of the recommendations, it still seems to provide very effective recommendations that would be suitable for actual applications.

## VIII. More than distributed training

Our project is mainly about the implementation of a distributed training system. We can make use of the existing framework to realize the distributed training systems. Our initial proposal included the recommendation system and two additional features, (1) Monitor for unstable nodes and (2) Defend against poisoned nodes. Due to time constraints we were only able to implement the distributed training and the monitor for node stability.

**Defense unstable clients** Initially we proposed a customized job dispatching policy to be implemented. When the server allocate the workload, it will consider the capability of each client, such as computational power, communication time. In this feature, the algorithm will attempt to monitor system performance and look for anomalies. In our project, we allowed for additional node communication so that the slave nodes frequently report CPU and memory usage to the master node. These usage levels can be utilized to create a protocol for load balancing on the worker nodes. For example, if one node runs slower relative to the nodes in the cluster, then this hardware monitor can be utilized to find another node in the cluster with more resources available.

## IX. Results & Analysis

Although our projects major focus is to distribute the training workload among nodes, in figure 4 we can see that the recommendation system provided consistent predictions that were accurate and similar to the ground truth values. I believe that with additional parameters such as a lambda that can be used to prevent over fitting can significantly increase our prediction accuracy even further.

**Master node and slave nodes cooperate to do the training** We get the result of three docker containers running three

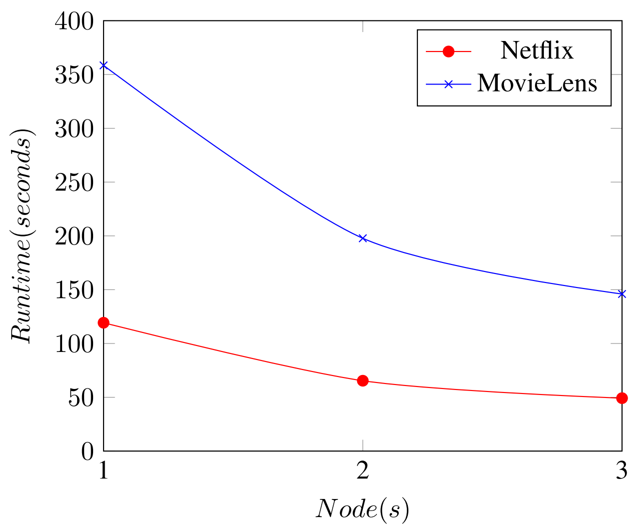


Fig. 5. Above are the runtimes for training on the Netflix and MovieLens datasets. The Netflix dataset consists of approximately 100M ratings while the MovieLens dataset consists of 5M ratings. Here we see a clear anomaly, a gradient descent model trained on a larger dataset (Netflix) takes significantly less time to run than a smaller (MovieLens) dataset although the data used in each one includes the same information, that is: userID, movieID and Ratings

independent programs to simulate the distributional training environment, where master node and slave node cooperate to do the training. Figure 5 shows the running time acceleration of distributional training on simulated environment. We can see that with only one node, the total running time is 119 second, if we use two nodes (65 seconds) or three nodes (49 seconds) for the Netflix data, the running time is largely reduced. In our MovieLens dataset we get 358 seconds for one node, 197 seconds for two nodes and 146 seconds for three nodes.

**Customized client information collection system** There are two main functions - send and receive in the system. In send function, We will construct the tensor variable then send it to specific target. It's a function running in slave node. In the receive function, we will store information received and its source to a dictionary. It's a function running in master node.

**Customized training information collection system**

Our program is based on PyTorch, so there is a very convenient way to collect model information, the register_forward_hook function, to realize the model information collection system. It helps us embed one specific function into a specific layer. Basically, when the model does the forward propagation, the specific function will be called, then we can get the real-time model information.

Overall, we finished a complete pipeline for distributional training, including simulation environment, PyTorch-enable distribution training, customized client information collection system, and customized model information collection system. We believe our project is not just about distributional training. With the four components, we actually provide a complete framework to construct robust environment-aware distributional training.

**Noteworthy Anomalies** If you refer to figure 5, the slope of the lines for the runtimes on training with either dataset produces a line that is not linear which refutes our hypothesis. Instead there seems to be a clear diminishing return on speedup for every node added to the cluster, although this is not an anomaly there is something that appears to be strange. The same model used with either dataset trained significantly faster on a dataset that was immensely bigger than the other. The Netflix dataset consisted of approximately 100M reviews yet it took a fraction of the time that MovieLens training took with a much smaller 5M rating dataset. Taking a look further it appears that there is a slight inconsistency when looking at the data types, after we changed them all to int64 values, the system began to experience stability issues so we left the values as they originally.

## X. Summary & Future Work

Since we realize that machine learning under the neural network architecture is usually the case, by continuously adding layers and nodes, this leads to an exponential explosion of training time. In order to find a way to shorten such a long training time. We try to distribute and systematize model training for machine learning. We tried Docker in action. We assign model training tasks to different CPUs to complete. Our results show that Docker effectively reduces the training time for each data-set. But the result did not end up in 1/N of the training time as we initially thought. Our distributed training successfully reduces the training time since training is performed simultaneously on different CPUs. The limitation of this study is mainly reflected in our distribution of training to a single machine hardware. This made it impossible for this study to handle overly large data-sets. Even if the training will be completed in the end, the training time may still be justifiably long. This research shows that even Docker has limitations on a single machine. Our findings underscore that Docker can be effective on moderately sized data-sets. (For example, the Netflix data is approximately 2.5 Gb while the MovieLens is only 366Mb). Our next research direction will be to send different Docker containers to different computers for training. This will build a large distributed machine learning system. This distributed machine learning system has the potential to form large data processors through our continuous optimization.

Although our environment to distribute the training is simulated, the program we created can easily be placed into containers each within hosts that have sufficient hardware capabilities to preprocess the large datasets but to also perform the actual training. Docker along with PyTorch allow for easy hardware configuration when executing the script for training. I believe the ideal setup for this cluster would be for various machines (each with a copy of the model and data) to each have a container and authorize full hardware capabilities so that the training is not restricted in any way. An alternative would be to utilize a computer system with several CPUs so that several containers can run simultaneously on the machine but each container is allotted a CPU for the duration of the training.

## XI. Lessons Learned

This project was packed full of challenges and obstacles we overcame. The most challenging portion was the distributed training framework, we had to learn how to allow docker containers to communicate while training through a distributed cluster in a synchronous manner.Initially, it was difficult to comprehend the underlying communication structure among PyTorch but it seemed to be fairly versatile. Additionally, we learned that the time it takes to train a model on a particular dataset is not solely dependent of the size of the file it will be training on, instead it can also vary on variable data types. By this I'm referring to figure the data anomaly as described in the results section.

## XII. Conclusion

To sum up, our interest in distributed gradient descent training stems from the fact that training time on a single PC is insufficient for training models on large scale datasets. A subset of the Netflix data-set with 5000 users and 17,000 movies takes over 6 hours on a PC. For comparison, we used the MovieLens dataset. As a stable benchmark dataset, MovieLens 25M movie ratings were chosen. Specifically, a subset with a total of 5M ratings. Perhaps we can reduce the training time necessary by distributing the workload to multiple CPUs; one of our aims is to determine if the time reduction is proportionate to the number of nodes we will utilize to train on the complete data-set. For example, if training a gradient descent model on a fraction of the total training data takes roughly 6 hours. The time it takes to finish the training isn't exactly a third of what it was before, but it's a big reduction. The next step in our research will be to send different Docker containers to various PCs for training. This will result in the development of a huge distributed machine learning system. Through our continual optimization, this distributed machine learning system has the potential to build a large data processor.

We also see that companies such as Netflix can leverage large scale distributed systems to create fast and high quality recommendations to retain customer profits.One interested approach that might alleviate the need for in-house training might be to utilize federated learning. Through this type of training, clients share a model through individual devices, and through device collaborating the model is trained. More specifically, the device can download the current model and through its own data can update and upload that model to other devices so that the model is trained gradually.

## XIII. Workload distribution

Responsibilities of team members are enumerated below

- Implement distribution training under the original recommendation system
  - *Jesus Garcia, Sinuo Fan, and Yijiang Pang*
- Further develop the framework to adapt the two relatively independent components
  (a) Customized client information collection system and workload management
  - *Yijiang Pang, Jesus Garcia*
  (b) Customized training information collection system
  - *Yijiang Pang*
- Deployment acceleration - Docker
  - *Sinuo Fan, Yijiang Pang*

## References

[1] M. POZO and R. Chiky, "An implementation of a Distributed Stochastic Gradient Descent for Recommender Systems based on Map-Reduce," in INTERNATIONAL WORKSHOP ON COMPUTATIONAL INTELLIGENCE FOR MULTIMEDIA UNDERSTANDING (IWCIM), Prague, Czech Republic, Oct. 2015, pp. 1–5. doi: 10.1109/IWCIM.2015.7347074.

[2] McMahan, H. B. and Ramage, D. Federated learning: Collaborative machine learning without centralized training data, April 2017. URL https://ai.googleblog.com/2017/04/ federated-learning-collaborative.html. Google AI Blog.

[3] Bonawitz K, Eichner H, Grieskamp W, et al. Towards federated learning at scale: System design[J]. Proceedings of Machine Learning and Systems, 2019, 1: 374-388.

[4] "Netezza and IBM Cloud Pak for Data: A knockout combo for tough data," Journey to AI Blog, Jun. 18, 2020. https://www.ibm.com/blogs/journey-to-ai/2020/06/netezza-and-ibm-cloud-pak-a-knockout-combo-for-tough-data/ (accessed Feb. 21, 2022).

[5] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Parallel matrix factorization for recommender systems," Knowl Inf Syst, vol. 41, no. 3, pp. 793–819, Dec. 2014, doi: 10.1007/s10115-013-0682-2.

[6] S. Li et al., "PyTorch Distributed: Experiences on Accelerating Data Parallel Training," Jun. 2020, Accessed: Feb. 22, 2022. [Online]. Available: https://arxiv.org/abs/2006.15704v1

[7] C. A. Gomez-Uribe and N. Hunt, "The Netflix Recommender System: Algorithms, Business Value, and Innovation," ACM Trans. Manage. Inf. Syst., vol. 6, no. 4, p. 13:1-13:19, Dec. 2016, doi: 10.1145/2843948.

[8] Tran, Brandon, Jerry Li, and Aleksander Madry. "Spectral signatures in backdoor attacks." Advances in neural information processing systems 31 (2018).

[9] Li, Yiming, et al. "Backdoor learning: A survey." arXiv preprint arXiv:2007.08745 (2020).

[10] K. Almohsen and H. Al-Jobori, "Recommender Systems in Light of Big Data," International Journal of Electrical and Computer Engineering (IJECE), vol. 5, pp. 1553–1563, Dec. 2015, doi: 10.11591/ijece.v5i6.pp1553-1563.

[11] F. M. Harper and J. A. Konstan, "The MovieLens Datasets: History and Context," ACM Trans. Interact. Intell. Syst., vol. 5, no. 4, p. 19:1-19:19, Dec. 2015, doi: 10.1145/2827872.

[12] "Docker Desktop," Docker. https://www.docker.com/products/docker-desktop/ (accessed May 04, 2022).

[13] "pytorch/pytorch - Docker Image — Docker Hub." https://hub.docker.com/r/pytorch/pytorch (accessed May 04, 2022).

## APPENDIX

### *A. Launch the project using Docker*

To simulate independent nodes used for training the model we use Docker containers. We simulate a cluster for distributed training with 1, 2 and 3 containers and compare their runtimes to one another. One major component of this project is setting up the Docker environment. Here we have created a comprehensive guide to help you in running our program.

First you will need to install a compatible version of Docker Desktop [12] on your local machine, please refer to citation for [12] for the website. After the install is complete, you will need to download a premade pytorch image for the docker containers [13].

The following instructions contain the necessary steps and commands to launch the whole program.

**Step 1: Create Docker Environment**

1) Create the node/container with PyTorch/PyTorch image with Docker
2) Run the following commands on your local terminal - *"docker run -it –network=host –name=node**N** –cpuset-cpus=**N** PyTorch/PyTorch"* to launch three independent nodes, where **N** = (1,2,3)
3) Install the dependency in each of the containers with commands
   a) *pip install matplotlib;tk;pandas*
   b) *apt update*
   c) *apt install libsm6 libxext6 libxrender-dev*

**Step 2: Initialize the code folder**

Copy the code file to each of the container with commands *"docker cp **path** node**N**:/workspace"*, where **path** is the folder path of the project code in local machine, and **N** = (1,2,3)

**Step 3: Run the project** In project folder of each 1,2,3 container, running the project with commands *"torchrun –master_addr="127.0.0.1" –master_port=1347 –nproc_per_node=1 –nnodes=3 –node_rank=**N-1** main.py"*, where **N** = (1,2,3)

Like previously stated our project creates a PICKLE file when the data is done preprocessing. Our Google Drive contains each pickle file so that you can simply copy it into your docker workspace to avoid the preprocessing step. If you train your model on one particular dataset and want to train it on another dataset please make sure you delete the created PICKLE file and the newFormatData.csv that resides in every docker container workspace, then you can replace it with your desired dataset (included in our Google Drive).