Deploying Deep Learning Models on Remote Distributed Systems

Pai Chen

School of Computer Science, Electrical and
Electronic Engineering, and Engineering Maths
University of Bristol
Bristol, UK
Email: perry.chen.pc@gmail.com

Abstract. The usual practice of training deep learning model is to use mini-batches of data in multiple epoches for the benefits of "improved generalization performance and allows a significantly smaller memory footprint"[2]. Patitioning and prcoessing data this way would come to a natural fit for implementing on a distributed system. In this paper, I made use of Pytorch's Distributed-Data-Parallel library (DDP) to train a simple Convolutional-Neural-Network on a number of remote aws EC2 instances.

Keywords: Deep Learning · Distributed System

1 Introdcution

The progress in graphics processing units (GPUs) in recent years has made them perfect candidates for training deep learning models due to their ability to take on massively parallelized processing jobs. However, the "scaling up" approach for switching to more expensive GPUs to raise performance does not apply on every occasion. In some cases, the performance of a single expensive GPU could be matched by having a cluster of cheaper ones instead. This kind of "scaling out" approach is suitable for deploying in a cloud environment. AWS cloud environment is chosen here as the primary platform to deploy a range of ec2 instances, each hosting a docker container running a separate training instance on a Convolutional Neural Network(CNN), Pytorch's Distributed-Data-Parallel library has provided easy access to convert single-machine training models into ones that can be deployed and synchronised for the cloud. I also make use of docker and docker swarm for managing cluster behaviours such as scaling and communication.

2 Architecture Overview

2.1 The Deep Learning Model

The model being used is a shallow convolutional neural network replicated exactly as the one described in [2], this model is then being used for training on the

GTZAN dataset, consisting of 1000 segments of songs, categorized into 10 music genres. However [2] does not specify a distributed implementation hence an adaption needs to be made. Making use of Pytorch DDP, the following changes are made to the training process:

Forward Pass The implementation of the forward-pass part of training a CNN on a distributed system requires little adaptation from its single-host counterpart, as each model that trains on a different partition of the entire dataset is considered independent, and there exists no dependency between them. Hence this part of the implementation is trivial, and the only notable difference is having multiple threads of execution implemented by Python's Multiprocessing library, which is then passed on to the ec2 instances as separate Docker containers.

Backward Pass During the backward pass phase of training the network, communications between different threads of execution are required, as a global average of all parameters trained from the forward pass phase is needed to advance training. Thanks to the functionalities provided by the DDP library, synchronization between different instances is made possible by having a synchronising reducer in each instance. Although the computation is done asynchronously, as in the reducers do not wait for all instances to compute their parameter, they do synchronise by waiting for all other reducers to compute the average result and then feed it to the optimizers. Hence all optimizers would start with the complete same states, acting as if they were on local machines, therefore consistency is enforced across the system.

2.2 Docker and Docker Swarm

in the same way as user-defined networks for standalone containers. You can attach a service to one or more existing overlay networks as well, to enable service-to-service communication. Overlay networks are Docker networks that use the overlay network driver of the partitioned data containerized using docker, this allows the training models to be run freely of the underlying host architecture without worrying about dependencies issues that might arise. Moreover, by using Docker Swarm as the orchestrator, the following things are achieved:

Task Orchestration As shown in fig.1, the container image containing the deep learning model is deployed as a service, which "defines its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more)."[4]. In the case of training the neural net, the optimal state, e.g. the number of replicas can be defined externally by setting the $WORLD_SIZE$ environmental variable on the client side, the setting is then ported over to the instance where the manager resides, which then creates the specified amount of replicas called "tasks" and distributes them evenly across all worker nodes, hence no user control is required for distribution of tasks as

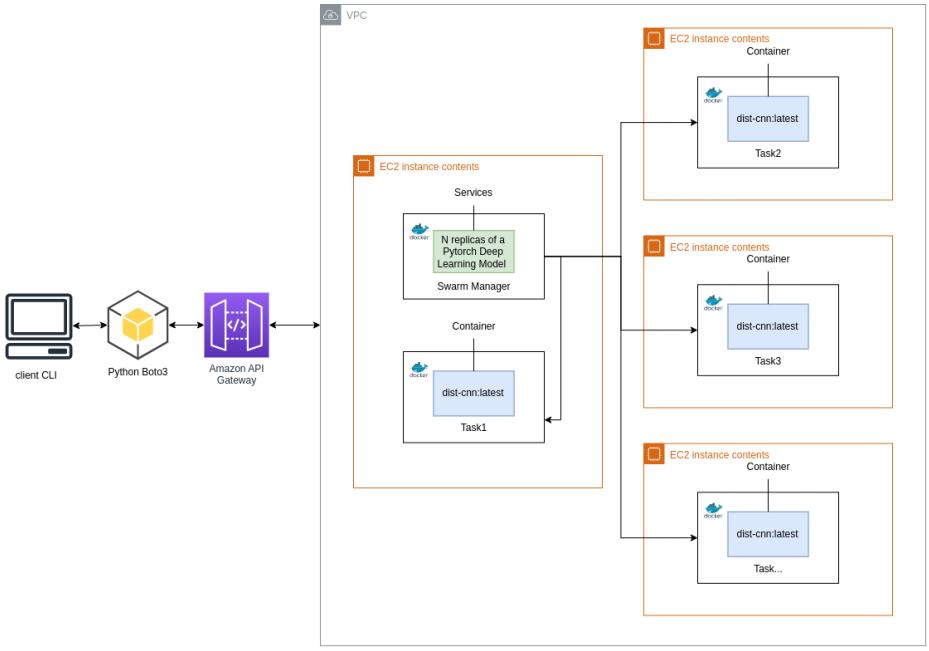

Fig. 1. Higher-level diagram representing the entire system architecture.

the manager takes care of that by taking into account the computation power available for each worker node.

Inter-container Communication During the backward pass phase of the training process, each process needs to communicate with all other processes to get a global average of all the parameters learnt, the need for the exchange of information between stand-along docker containers is made possible by using an overlay network driver in combination with the docker swarm mode. Each task replica deployed by the swarm manager can be attached to a shared overlay network by specifying the "–network" parameter.

2.3 Automate deployment using Boto3 and fabric

The use of python libraries, specifically boto3 and fabric has abstracted away the complexity of interacting with the AWS resources. Boto3 is a python API for the AWS CLI which allows full user access and control over the suite of AWS resources such as Amazon Elastic Compute Cloud (Amazon EC2) and instances and Amazon Elastic Block Store (EBS), whereas fabric allows users the ability to acquire the aforementioned resources through the command line. By leveraging the functionalities of these two libraries, the resources for the cloud architecture required to do parallel deep learning can be acquired and deployed with ease.

3 Testing

The instance size is the parameter provided for user tuning, hence effect it has on the time of training taken is investigated. Specifically, I record the time elapsed for the model to finish a 10-epochs training process with instance sizes from 1 to 5.

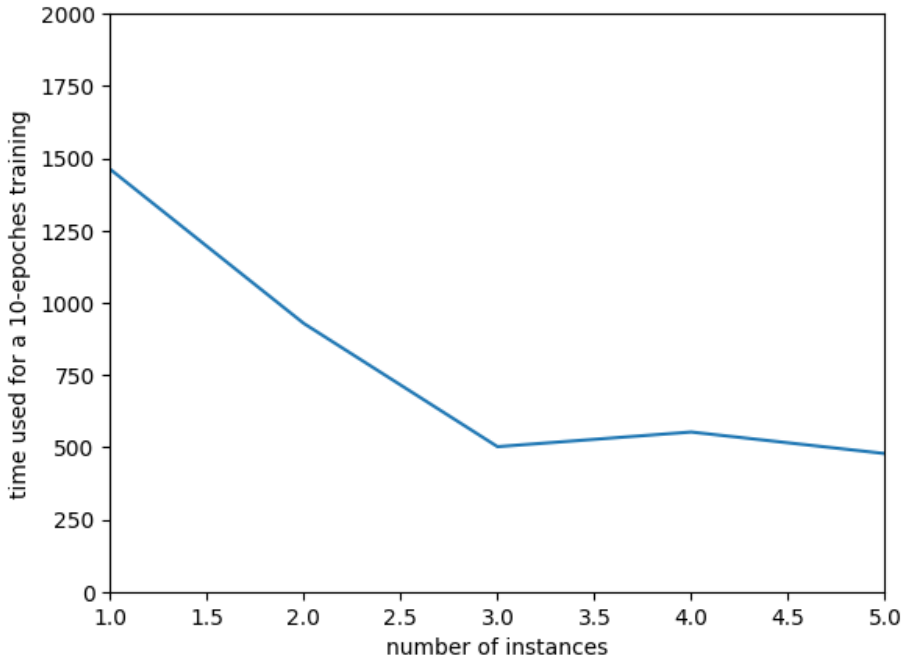

Fig. 2. Test results got back from training the neural network for 10 epochs with different instance sizes ploted as a line graph

By looking at the results illustrated by 2, it is clear that there is a massive improvement in training speed when the number of instances jumped from 1 to 3. However the improvements are less evident once more instances have been deployed, a slight increase in training time is even observed when going from 3 to 4 instances. A potential reason for this could be the communication overhead induced by synchronising the parameters between different instances outweighs the performance gains of using a distributed system.

4 Conclusion and Future Works

In this paper, a distributed system approach for training deep learning models have been implemented, in an attempt to reduce the dependency on expensive

underlying hardware requirement. The performance results got back from comparing speed performance for different hardware instance sizes show a certain degree of improvement in training speed, provided the instance size is small. A more scaled-out version would possibly need innovative design decisions to be made, to reduce the communication overheads.

Moreover, the current implementation provides little fault tolerance which makes the system difficult to scale to more instances, as any occurrence of failure in one of the running instances would halt the other instances as well. This problem could be tackled in the future by integrating torchrun[5], a utility made by PyTorch providing fault tolerance by saving snapshots of the model during the training process and recovery from these snapshots.

References

1. Github link to the source code: https://github.com/ccdb-uob/CW22-47
2. Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010 (pp. 177-186). Physica-Verlag HD.
3. Schindler, Alexander, Thomas Lidy, and Andreas Rauber. "Comparing Shallow versus Deep Neural Network Architectures for Automatic Music Genre Classification." In FMT, pp. 17-21. 2016.
4. Docker Docs, `https://docs.docker.com/engine/swarm/key-concepts/`. Last accessed 7 Dec 2022
5. Fault-tolerant Distributed Training with torchrun, `https://pytorch.org/tutorials/beginner/ddp_series_fault_tolerance.html`. Last accessed 7 Dec 2022