DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Analyzing Performance Bottlenecks in Collaborative Deep Learning

Adrian D. Castro Tenemaya

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Analyzing Performance Bottlenecks in Collaborative Deep Learning

Analyse von Leistungsengpässen im Kollaborativen Deep Learning

Author:	Adrian D. Castro Tenemaya
Supervisor:	Prof. Dr. Ruben Mayer
Advisor:	M. Sc. Alexander Isenko
Submission Date:	October 9, 2022

I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

München, October 9, 2022 Adrian D. Castro Tenemaya

Acknowledgments

Thanks to everyone who tagged along on my academic journey. A special thanks go to my wonderful and supportive parents Laura and José; my "sore" Valeria, and my girlfriend Iris. Also, thanks to everyone who has believed in me and who didn't make me quit (you know who you are). Finally, many thanks to my very patient academic advisor Alexander.

Abstract

The amount of computing resources required to train state-of-the-art deep neural networks is steadily growing. Most institutions cannot afford the latest technologies, which are sometimes needed to keep up with today's demanding deep neural network research. Access to powerful devices is therefore limited to a few parties, slowing down research.

Hivemind [tea20] is an open-source framework that enables collaborative model training using a large number of heterogeneous devices from universities, companies, and volunteers. The framework implements two decentralized training algorithms: "Decentralized Mixture-of-Experts" (DMoE) and "Parameter Averaging". In this thesis, we focus on the effects of training with Hivemind using parameter averaging technique compared to regular training. Parameter averaging replicates a model on all training network's peers, averaging their parameters after a certain amount of samples have been globally processed.

In Hivemind, every device participating in the computation may differ in its characteristics, featuring different architectures and network speeds. In an interactive demonstration [tea20], 40 devices jointly trained a modified DALL-E [Ram+21] neural network model over 2.5 months using Hivemind's parameter averaging training technique. The reported results, however, do not include the participant's device information and metrics. Without them, it is not possible to perform an independent analysis of the effects of different configurations on training.

In our experiments, we evaluate the effect of several aspects of training with Hivemind in a controlled cluster setup. The experiments shown in this thesis use different settings such as the number of peers involved in the training, using local updates, batch size, learning rate and more. We prove that Hivemind can reach a target loss faster on specific scenarios and settings compared to regular training with a single node. We also show that despite the communication overhead, Hivemind can still outperform the training baseline in terms of speedup Finally, we provide some considerations and lessons learned by summarizing the results of our experiments.

Contents

1 Introduction

It is safe to say that the internet paved the way for many things for humanity. Media such as images, video and audio can be shared across websites and applications, knowledge can be stored in faraway servers and retrieved with ease in text format using mobile devices, and products and services can be bought with the click of a button or a tap on a screen. Interactions, media and information make up for massive amounts of data that flow through complex computer systems, which in turn generate even more data and information.

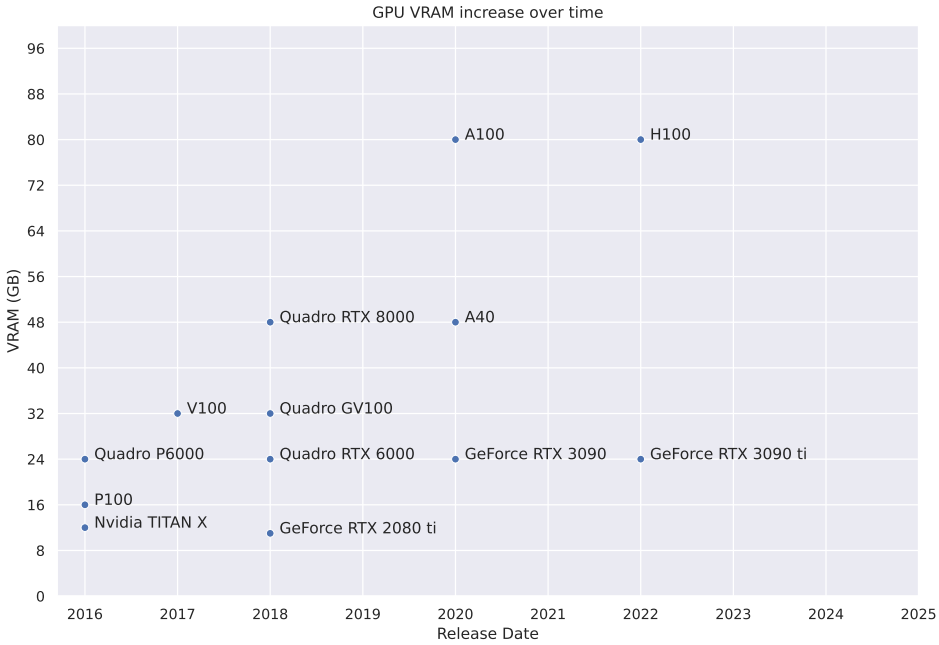
Researchers have found ways to leverage the magnitude of data that is being produced every second by countless systems all around the world. One of the most recent and most popular uses of this huge variety and quantity of data is machine learning (ML). Machine learning can be defined as a set of techniques that use data to improve performance in a set of tasks. Today, for example, we feed data to machine learning models to calculate what is the probability that a webpage visitor will buy certain products, or the chances that it is going to rain in a few days or to generate elaborate text and stunning, never-before-seen pictures.

Recently, models such as BERT [Dev+18], DALL-E [Ram+21], GPT-3 [Bro+20] and others have become incredibly popular thanks to their outstanding results and endless possibilities. DALL-E for example can generate high-quality realistic images and art starting from a text description written in natural language. These models however require massive amounts of data as well as very expensive computational resources, such as graphical processing units and tensor processing units (TPUs). In recent years, the size of neural network models has been steadily increasing exponentially, as shown by Figure 1.2. A simple calculation shows that the neural network model Megatron-Turing-NLG 530B [Smi+22] would take roughly $530 \times 4 = 2120\text{GB}$ of memory to simply hold its 530 billion weights.

Furthermore, training a neural network model requires even more memory. Intermediate computation outputs such as gradient and optimizer states sometimes require 2 or 3 times as much memory than just the model parameters, making GPU memory one of the main bottlenecks in training huge neural network models. While some of these issues can be tackled using clever techniques such as parameter quantization, pruning and compression, they must not be considered one-fits-all solutions. Some models are simply too big to be trained on a single device. This problem is exacerbated by factors

such as GPU prices and much slower growth of their memory size. Figure 1.1 shows how GPU memory has been increasing from 2016 to 2022.

Figure 1.1: GPU VRAM over the past 4 years. The growth is mostly linear, doubling



The characteristics of the latest GPU released by NVIDIA earlier in 2022, the H100 with 80GB of memory, an amount that hasn't changed since its direct predecessor A100.

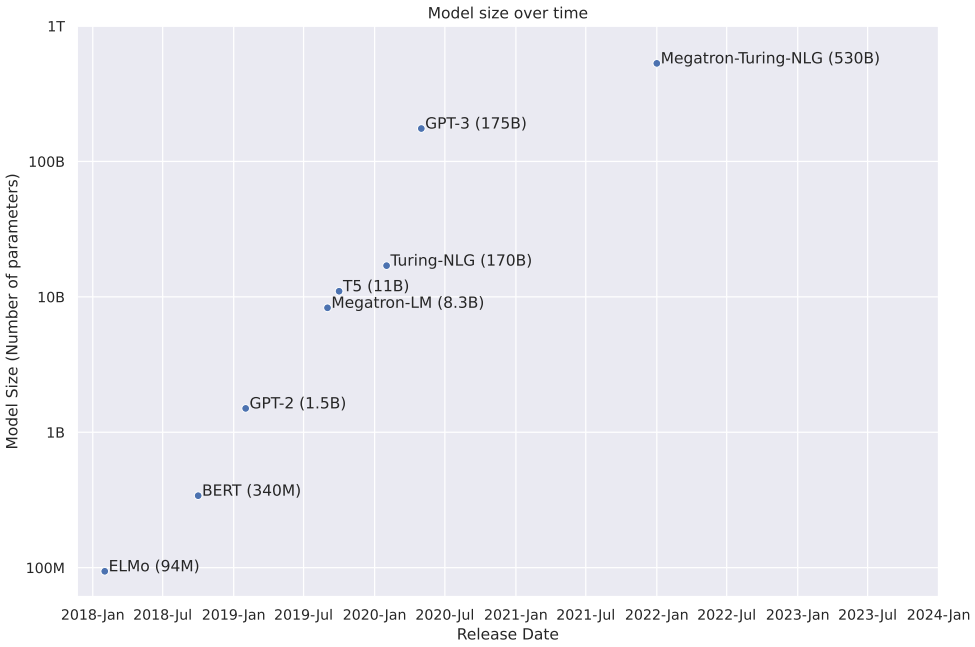
To tackle this problem, practitioners studied and developed distributed computing techniques to train models that do not fit entirely in a single GPU's memory, distributing the training load to potentially thousands of devices.

These techniques can be briefly categorized as follows:

- *Data parallelism*. Given a set of n devices, an instance of the model is trained on each one of the devices. Usually, gradients obtained during backpropagation are then aggregated across all the devices using techniques such as *AllReduce*. This technique however does not work very well with models that exceed a single device's available memory and is therefore used in applications with low-memory devices such as *Federated Learning* [Li+19].

- *Model parallelism*. A deep neural network is conceptually split into n partitions

Figure 1.2: Model size over the past 4 years: ELMo [Pet+18], BERT [Dev+18], GPT-2 [Rad+19], Megatron-LM [Sho+19], T-5 [Raf+19], Turing-NLG [Mic20], GPT-3 [Bro+20], Megatron-Turing-NLG [Smi+22]



across n devices, each hosting a different partition and set of weights. An early notable example of model parallelism is AlexNet [KSH12], where the authors decided to split the computation of some of the layers across two GPUs with 3GB of ram each, a concept illustrated in Figure 1.3. This technique relieves the burden of a single node to host all of the weights of a model but is also more sensitive to issues with communication across nodes.

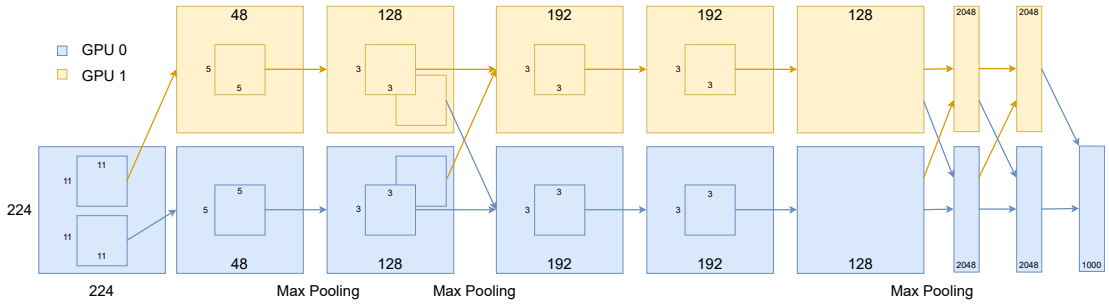
- *Pipeline parallelism*. A combination between model parallelism and data parallelism. Introduced by [Hua+18], pipeline parallelism consists in splitting a batch into micro-batches across the available computing devices, leading to fewer dead wait times.

- *Tensor parallelism*. Tensor operations for huge neural network models can become a bottleneck, as they can require more memory than the host's device can handle and can become slow in general. Tensor parallelism can alleviate a single node's

computational burden by splitting a tensor operation across many devices. This technique has been first introduced in neural network models in [Dea+12] with a framework called DistBelief.

- *Mixture-of-Experts*. Instead of conceptually splitting a model, we construct multiple, different models. Each of them specializes in a different subset of the complete set of training cases, helped by a gating function. This has been first introduced in 1991 by [Jac+91] but has recently gained traction due to its attractiveness in numerous tasks.

Figure 1.3: AlexNet [KSH12] architecture shows one of the first examples of model parallelism. The training of convolutional layers is split across two GPUs, as the size of the model during training exceeded the available memory of a single GPU.



1.1 Motivation

Most of the frameworks and papers built on top of the techniques presented previously approach scalability issues by employing tremendous amounts of expensive, top-of-the-art and heterogeneous hardware. However, universities, small companies and hobbyists that want to train the models described in these papers do not necessarily have access to a such vast amount of resources, limiting possibilities and research directions.

The framework Hivemind [RG20a] aims to help with these issues by allowing distributed neural network training across the internet using heterogeneous devices. The authors provide two types of training: *decentralized parameter averaging* and *decentralized mixture-of-experts*.

Hivemind implements two training algorithms: "Decentralized Mixture-of-Experts" (DMoE) [RG20b] and "Parameter Averaging". DMoE has been shown to work well for decentralized deep neural network training using large amounts of consumer-grade

hardware. The algorithm described in the original paper employs a combination of decentralized and Mixture-of-Experts [Sha+17] techniques. This allows thousands of computing devices to join forces and train a single neural network model together.

DMoE achieves these results by splitting the target neural network model into different parts called partitions, similarly to model parallelism. Each partition is then replicated across a subset of workers participating in the training. Next, a gating function is used to select which workers can perform the next operation on the given input. After the workers have been selected and located using a Distributed Hash Table (DHT), the input data is sent to the workers, and a forward pass is performed. A similar algorithm is applied during the backward pass. DMoE proved that scaling model training to thousands of heterogeneous compute nodes is possible, thus enabling large-scale community research projects.

Despite an academic interest in DMoE, we have eventually decided against performing any experimentation on DMoE for this thesis, as the API was not stable enough at the time of experiments. For this reason, the focus of this thesis is on decentralized parameter averaging, the second type of training algorithm implemented by Hivemind. With decentralized parameter averaging, each node participating in the training must have a copy of the model in memory. Every node performs training at its own pace, accumulating samples toward a global goal called "target batch size". Once this goal has been reached, an averaging round starts. The final gradients are calculated depending on the contribution in terms of the number of samples done by each peer, ensuring stability in case of peer failure.

Decentralized parameter averaging has shown promising results [], successfully training a modified version of DALL-E using 40 peers over two months. TODO: add something else about this.

Over the last few years, research and software libraries like Hivemind have been focused on reducing and optimizing deep neural network model training times with techniques such as data and model parallelism. In [Xin+21] however, the authors show that as much as 45% of total training time may be spent on preprocessing tasks alone. Despite this, the impact of preprocessing pipelines is often ignored in current research. As noted by Isenko et al. [Ise+22], it is crucial to find and analyze bottlenecks during computation to maximize performance. In their work, they also detail several possible improvements that can be applied in preprocessing pipelines, increasing throughput under certain circumstances. Intuitively, given the high amount of communications and data loading needed by parameter averaging, training may be subject to inefficiencies. Using the techniques and findings showcased in [Ise+22], this thesis further aims to find bottlenecks while training with Hivemind.

1.2 Approach

In this thesis, we will compare regular training using a single node with 16vCPUs to training using Hivemind with multiple peers, where the sum of vCPUs per peer always amounts to 16. Also, the number of samples We compare training with key Hivemind settings, namely:

- Batch size;
- Learning rate;
- Number of peers involved in the training;
- Target number of samples that must be globally reached by all peers to perform an averaging round;
- Applying the gradients at every step or accumulating them until the next averaging round.

As we test the software and its limitations, we might find possible areas of improvement in Hivemind. Whenever possible, we will further contribute using the knowledge gathered through our experiments by improving the Hivemind [tea20] source code.

1.3 Contributions

Our contributions are as follows:

- We analyze the challenges of optimizing preprocessing pipelines in decentralized distributed training and provide insights on possible improvements
- We verify the effectiveness of Hivemind for different peer hardware configurations concerning preprocessing pipelines
- We use the gained knowledge and insights to contribute to the Hivemind open-source library.

2 Fundamentals

In this chapter, we are going to define the basic concepts needed to understand the contents of this paper. The mathematical details of algorithms or technical implementations of the presented technologies will not be described in depth, but rather briefly and concisely describe how they work.

@isenko: still working on this part!

We will formally define what are neural networks and how training works using a simple scenario. Following, we will introduce why and how distributed computing techniques are important in today's world, and how they can be used to facilitate neural network training. The topic of distributed neural network training is then presented. Finally, we describe Hivemind, how it works, why we chose this framework, which type of analysis we are going to conduct, and what kind of parameters are we focusing on in this paper.

2.1 Neural Networks

Neural networks (NNs) have been a major research area in the past decade. However, the basics of NNs as we know them today have been around for almost 70 years but did not gain much recognition until the 2010s. This is mostly because the success and viability of NNs are affected by two major factors, data availability and computational power, both of which were scarce or not advanced enough. As network-connected devices like smartphones and laptops started to become more widespread, the data that could be generated and gathered grew by several orders of magnitude. Today, NNs are used in many fields, from pharmaceutical to translation, from art generation to autonomous driving.

Let us introduce a simple example to support the following explanations. We might want to classify many images of cats and dogs using a NN so that its inputs are images of cats or dogs, and the output is a binary variable indicating 0 if the image is a cat and 1 if it is a dog.

To understand NNs, we have to look at their smallest components called *neurons*, which are functions that can be defined as follows:

$$y = g(X \cdot W + b) \tag{2.1}$$

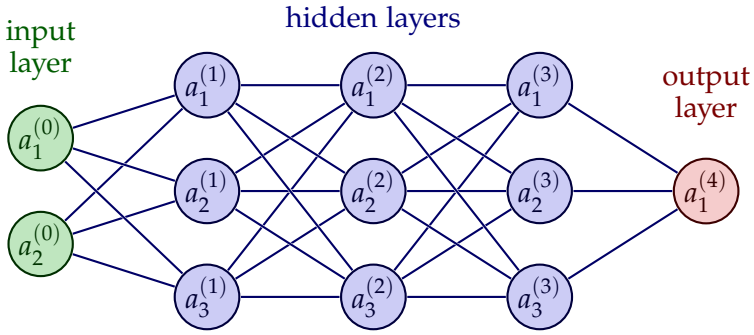
where g is called the activation function, $X \in \mathbb{R}^n$ is the neuron's input, $y \in \mathbb{R}$ is the output, $W \in \mathbb{R}^n$ its weights or parameters, and $b \in \mathbb{R}$ its bias. Note that activation functions must be non-linear functions. We would like to use NNs to solve and predict non-linear problems such as our example, a task that can only be achieved by using non-linear functions. If we were to compose a neural network using only linear functions, the output would still be a linear function, regardless of a NN's complexity. Examples of non-linear functions commonly used inside NNs are the sigmoid and the rectified linear unit (ReLU). The weights W and the bias b define the result of the activation function g.

The first and most simple type of NN that was devised is called *feed-forward neural network* (FF) and is comprised of many neurons stacked together in *layers*. These layers f are then composed together to form a feed-forward NN:

$$Y = f_1 \circ f2 \circ \ldots \circ f_L \tag{2.2}$$

where L is called the *depth* of a FF neural network, or number of *hidden layers* in a FF network.

Figure 2.1: An example of a neural network, with input layers (green nodes), hidden layers (blue nodes), and output layer (red node).



The depth, as well as the types of layers and functions of a given neural network, define its *architecture*. A NN architecture can be changed to obtain different results in terms of effectiveness, speed or other criteria. An example of one such architecture is represented in Figure 2.1.

At this point, the model has fixed parameters W and b, so given the same input, the output will be the same. We would like to update the parameters in such a way that the output reflects some arbitrary characteristic of the input, a process called *training*.

2.1.1 Training

Using our example, the NN should learn over time and using many examples, which images represent a cat, and which ones represent a dog. We can provide information to the neural network about whether or not it is right or wrong, and update its parameters according to how much it is far from the truth.

Formally, the function determining how much a NN is wrong about a guess is called a *loss function*, which outputs a value called *loss*. For a binary value such as our example, we can use the *binary cross entropy* loss function. The lower this value is, the closest the NN is to the ground truth.

We can derive certain properties from this value, such as how much should we change the parameters of our NN model so that we get a lower value the next time we try. This approach can be formally described as an optimization problem, where the optimization function, also called optimizer, is defined as follows:

$$\arg\min_{W,b} \mathcal{L}(W, b) \tag{2.3}$$

The optimizer function defines how the values of W and b should change to get a better loss. The process of iteratively updating these values multiple times using different inputs is called *training*. Commonly used optimizers for NN training are *Stochastic Gradient Descent*, *Root Mean Square* (RMSProp), *Adam* and others.

The values obtained using these optimization functions are used to determine the best next local optima for the given parameters. This process can then be repeated multiple times until an arbitrary loss value is reached, and the training process is stopped. The final architecture of the neural network and the state of the weights and biases are then fixed, obtaining the final NN model.

At several stages during training, a neural network practitioner might want to validate the results obtained by using a set of data that is different from the one that has been used to train the NN. This is called the *validation step*, and it is performed without changing the network's parameters or architecture. Validation steps are crucial to understanding if the changes made to a model are biased towards a specific set of inputs, an effect denominated *underfitting*.

2.2 Distributed Computing and Storage for Neural Networks

2.3 Distributed Training

2.3.1 Bottleneck Analysis

2.3.2 Metrics

2.4 Hivemind

Hivemind is a framework that aims to enable decentralized training using several techniques, such as decentralized parameter averaging or decentralized mixture-of-experts. In this section, we will introduce on a high level the main components and settings that we will use throughout the thesis.

2.4.1 Decentralized Hash Table (DHT)

A decentralized hash table or DHT is a system that provides a set of features similar to a hash table in a distributed environment. Because Hivemind focuses on providing a training environment for nodes across the internet, a DHT is a beneficial part of the architecture because of its fault-tolerant properties. Under the hood, Hivemind uses a DHT to track the presence or dropout of peers during training and to allow a direct exchange of data between them.

2.4.2 Optimizer

One of the main components of Hivemind is the Hivemind Optimizer, which wraps around any other implementation of a `torch.optim.Optimizer`. Therefore, the Hivemind Optimizer can be used as a drop-in replacement in regular training operations. With default settings and with no other peer, the Hivemind optimizer is designed to perform exactly as the underlying `torch.optim.Optimizer`.

Target Batch Size

In Hivemind, the target batch size (TBS) is defined as the global number of samples that every peer has collectively processed during a collaborative run in the current Hivemind epoch or *HE*. A Hivemind epoch does not necessarily correspond to a full pass over the training data. In Hivemind, the HE starts at 0 at the beginning of training and increases by one by each peer every time the TBS is reached. Let's describe a simple scenario:

- two peers are collaboratively training a model;
- the TBS is 256;
- each peer processes a batch size of 64;
- each peer processes every batch at roughly the same speed;
- each peer starts training at the same time.

In this scenario, after one step the number of globally accumulated samples is equal to $64 + 64 = 128$, as each peer has processed the same amount of samples in the same amount of time. After another step, the accumulated samples are equal to $128 + (64 + 64) = 256$. Because the TBS is now 256, now the Hivemind optimizer starts an averaging round between all connected peers and when done, executes the optimizer step. After the averaging round ends, each peer locally performs $HE = HE + 1$.

Local Updates

3 Related Work

4 Setup

In this chapter, we describe the requirements for building a suitable environment to run experiments using Hivemind. We further present a high-level overview of our implementation with a simplified visualization. Finally, we describe in detail the technologies and resources that we have used to build such as the environment (CEPH, OpenNebula, description of the machines), and how we provision them (Terraform, Ansible, ...).

4.1 Experimental Setup

4.2 System's Architecture

The scope of this thesis is to provide insights about using Hivemind in a controlled environment, such as a cluster of devices that a university, company or institution may leave otherwise unused.

4.3 Implementation

5 Experiments

In this chapter, we showcase the basic setup of our experiments. To preserve a comparison consistency between each experiment run, the number of steps depends on two factors: the number of peers involved in the training, and the batch size.

5.1 Base Case

The most basic setup for training a neural network is composed of a single device with access to powerful computing hardware, such as a CPU, GPU or TPU, where the model is trained on. In most cases, using a single computing device is infeasible, and parallelization techniques are employed to speed up training or to train bigger models that do not fit in the memory of a single device. For our purposes, this type of training can be seen as training on a single device.

In this thesis, we define the baseline for our experiments as training ResNet18 on a single machine with 16 vCPUs. We cover the following training hyperparameters:

- Batch Size (BS): 32, 64 and 128;
- Learning Rate (LR): 0.001, 0.01 and 0.1;
- Max Steps (MS): 10.000 for BS=32, 5000 for BS=64, 2500 for BS=128.

Each baseline run is repeated 4 times to observe the reproducibility of the measurements.

Table 5.1: List of baseline experiments and hyperparameters

Baseline experiments		
Max Steps	Batch Size	Learning Rate
10.000	32	0.001
10.000	32	0.01
10.000	32	0.1
5000	64	0.001
5000	64	0.01
5000	64	0.1

2500	128	0.001
2500	128	0.01
2500	128	0.1

5.2 Not-Baseline Case

To test and isolate the effects of using Hivemind for distributed training, each of our experiments changes only a single parameter at a time. For this, we can divide the set of non-baseline cases into different categories depending on which parameter has been changed. In every non-base case scenario described in this section, at least two nodes are involved in the training of the underlying NN model.

5.2.1 Target Batch Size

In Hivemind, the target batch size (TBS) is defined as the global number of samples that every peer has collectively processed during a collaborative run in the current *Hivemind epoch*. A Hivemind epoch starts at 0 at the beginning of training and increases by one each time the TBS is reached. Let's describe a simple scenario:

- two peers are collaboratively training a model;
- each peer processes a batch size of 64;
- each peer processes every batch at roughly the same speed;
- the TBS is 256.

The model and the dataset remain the same across each run, and every peer has full access to the entire dataset through our CEPH cluster. We repeat the same experiments as the baseline runs, but alter the following Hivemind settings:

- `Use Local Updates (LU)` (T/F);
- `Number of Peers (NoP)` (2/4/8/16);
- `Target Batch Size (TBS)` (10.000/5000/2500/1250/625);
- `vCPUs` (1/2/4/8);
- `Max Steps (MS)` (5000/2500/1250/625), depending on the number of peers and batch size, but always totaling 10.000 steps per experiment;

The Table 5.2 shows a list of all Hivemind experiments, with each one of them being also executed with LU=True and LU=False. In total, we have executed 144 experiments to test Hivemind.

Table 5.2: List of Hivemind experiments and hyperparameters. Every experiment has been executed once, and every time with two peers.

Hivemind experiments					
MS	NoP	vCPUs	BS	LR	TBS
5000	2	8	32	0.001	10.000
5000	2	8	32	0.001	5000
5000	2	8	32	0.001	2500
5000	2	8	32	0.001	1250
5000	2	8	32	0.001	625
5000	2	8	32	0.01	10.000
5000	2	8	32	0.01	5000
5000	2	8	32	0.01	2500
5000	2	8	32	0.01	1250
5000	2	8	32	0.01	625
5000	2	8	32	0.1	10.000
5000	2	8	32	0.1	5000
5000	2	8	32	0.1	2500
5000	2	8	32	0.1	1250
5000	2	8	32	0.1	625
2500	4	4	32	0.001	1250
1250	8	2	32	0.001	1250
625	16	1	32	0.001	1250
2500	4	4	32	0.01	1250
1250	8	2	32	0.01	1250
625	16	1	32	0.01	1250
2500	4	4	32	0.1	1250
1250	8	2	32	0.1	1250
625	16	1	32	0.1	1250
2500	2	8	64	0.001	10.000
2500	2	8	64	0.001	5000
2500	2	8	64	0.001	2500
2500	2	8	64	0.001	1250
2500	2	8	64	0.001	625
2500	2	8	64	0.01	10.000
2500	2	8	64	0.01	5000
2500	2	8	64	0.01	2500
2500	2	8	64	0.01	1250
2500	2	8	64	0.01	625

2500	2	8	64	0.1	10.000
2500	2	8	64	0.1	5000
2500	2	8	64	0.1	2500
2500	2	8	64	0.1	1250
2500	2	8	64	0.1	625
2500	4	4	64	0.001	1250
1250	8	2	64	0.001	1250
625	16	1	64	0.001	1250
2500	4	4	64	0.01	1250
1250	8	2	64	0.01	1250
625	16	1	64	0.01	1250
2500	4	4	64	0.1	1250
1250	8	2	64	0.1	1250
625	16	1	64	0.1	1250
1250	2	8	128	0.001	10.000
1250	2	8	128	0.001	5000
1250	2	8	128	0.001	2500
1250	2	8	128	0.001	1250
1250	2	8	128	0.001	625
1250	2	8	128	0.01	10.000
1250	2	8	128	0.01	5000
1250	2	8	128	0.01	2500
1250	2	8	128	0.01	1250
1250	2	8	128	0.01	625
1250	2	8	128	0.1	10.000
1250	2	8	128	0.1	5000
1250	2	8	128	0.1	2500
1250	2	8	128	0.1	1250
1250	2	8	128	0.1	625
2500	4	4	128	0.001	1250
1250	8	2	128	0.001	1250
625	16	1	128	0.001	1250
2500	4	4	128	0.01	1250
1250	8	2	128	0.01	1250
625	16	1	128	0.01	1250
2500	4	4	128	0.1	1250
1250	8	2	128	0.1	1250
625	16	1	128	0.1	1250

Table 5.3: Something cool about this table

Country List			
Country Name or Area Name	ISO ALPHA 2 Code	ISO ALPHA 3 Code	ISO numeric Code
Afghanistan	AF	AFG	004
Aland Islands	AX	ALA	248
Albania	AL	ALB	008
Algeria	DZ	DZA	012
American Samoa	AS	ASM	016
Andorra	AD	AND	020
Angola	AO	AGO	024

6 Results

6.1 Focus on effects of Batch Size and Learning Rate

6.2 Focus on effects of Local Updates

6.3 Focus on effects of Number of Peers

7 Future Work

8 Conclusions

List of Figures

List of Tables

Bibliography

[] *learning@home*. https://learning-at-home.github.io/.

[Bro+20] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. "Language Models are Few-Shot Learners." In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165.

[Dea+12] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. "Large Scale Distributed Deep Networks." In: *NIPS*. 2012.

[Dev+18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805.

[Hua+18] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism." In: *CoRR* abs/1811.06965 (2018). arXiv: 1811.06965.

[Ise+22] A. Isenko, R. Mayer, J. Jedele, and H.-A. Jacobsen. "Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines." In: *Proceedings of the 2022 International Conference on Management of Data*. SIGMOD '22. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 1825–1839. ISBN: 9781450392495. DOI: 10.1145/3514221.3517848.

[Jac+91] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. "Adaptive Mixtures of Local Experts." In: *Neural Computation* 3.1 (1991), pp. 79–87. DOI: 10.1162/neco.1991.3.1.79.

[KSH12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012.

[Li+19] Q. Li, Z. Wen, Z. Wu, S. Hu, N. Wang, X. Liu, and B. He. "A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection." In: *CoRR* abs/1907.09693 (2019). arXiv: 1907.09693.

[Mic20] Microsoft. *Turing-NLG: A 17-billion-parameter language model by Microsoft - Microsoft Research*. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/. May 2020.

[Pet+18] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. "Deep contextualized word representations." In: *CoRR* abs/1802.05365 (2018). arXiv: 1802.05365.

[Rad+19] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. "Language Models are Unsupervised Multitask Learners." In: (2019).

[Raf+19] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." In: *CoRR* abs/1910.10683 (2019). arXiv: 1910.10683.

[Ram+21] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever. "Zero-Shot Text-to-Image Generation." In: *CoRR* abs/2102.12092 (2021). arXiv: 2102.12092.

[RG20a] M. Riabinin and A. Gusev. "Learning@home: Crowdsourced Training of Large Neural Networks using Decentralized Mixture-of-Experts." In: *CoRR* abs/2002.04013 (2020). arXiv: 2002.04013.

[RG20b] M. Riabinin and A. Gusev. "Learning@home: Crowdsourced Training of Large Neural Networks using Decentralized Mixture-of-Experts." In: *CoRR* abs/2002.04013 (2020). arXiv: 2002.04013.

[Sha+17] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." In: *CoRR* abs/1701.06538 (2017). arXiv: 1701.06538.

[Sho+19] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." In: *CoRR* abs/1909.08053 (2019). arXiv: 1909.08053.

[Smi+22] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zheng, R. Child, R. Y. Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro. "Using DeepSpeed and Megatron to Train

Megatron-Turing NLG 530B, A Large-Scale Generative Language Model." In: *CoRR* abs/2201.11990 (2022). arXiv: 2201.11990.

[tea20] L. team. *Hivemind: a Library for Decentralized Deep Learning*. https://github.com/learning-at-home/hivemind. 2020.

[Xin+21] D. Xin, H. Miao, A. G. Parameswaran, and N. Polyzotis. "Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities." In: *CoRR* abs/2103.16007 (2021). arXiv: 2103.16007.