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:	November 4, 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, November 4, 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

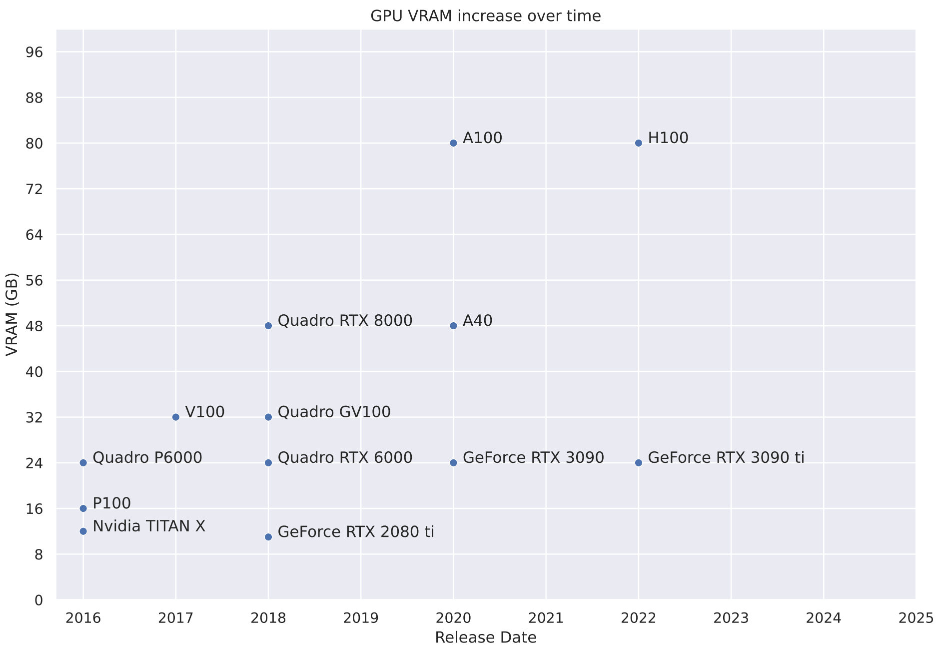
It is safe to say that the internet paved the way for many things for humanity. Media such as images, video and audio can be shared across websites and applications, knowledge can be stored in faraway servers and retrieved with ease in text format using mobile devices, and products and services can be 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 $530 \times 4 = 2120$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 techniques such as parameter quantization [LTJ20], 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.

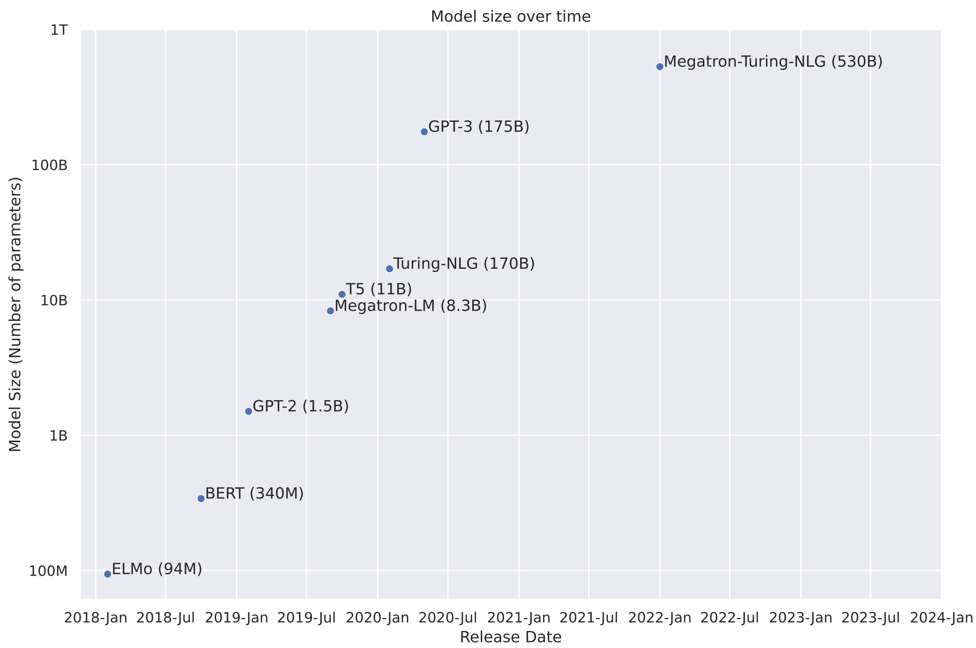
To tackle this problem, practitioners studied and developed distributed computing techniques to train models that do not fit entirely in a single GPU's memory, distributing the training load to potentially thousands of devices.

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]



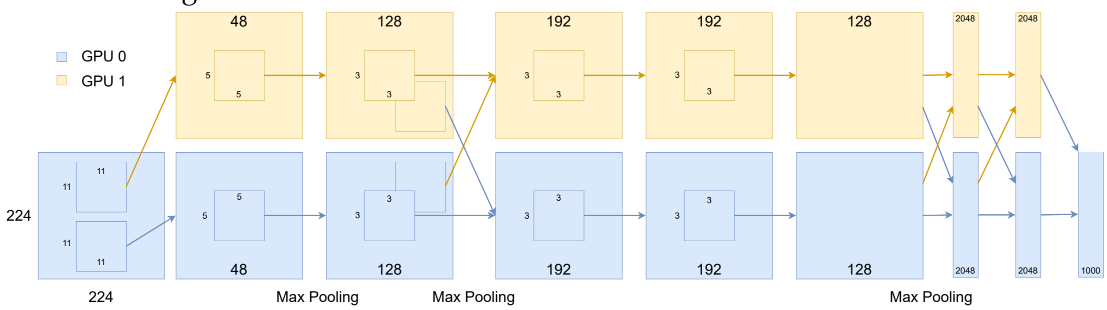
across n devices, each hosting a different partition and set of weights. An early notable example of model parallelism is AlexNet [KSH12], where the authors decided to split the computation of some of the layers across two GPUs with 3GB of ram each, a concept illustrated in Figure 1.3. This technique 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. Pipeline parallelism for machine learning models has been introduced in 2018 [Hua+18]. With this technique, each batch is split into micro-batches and sent to available computing devices such as GPUs, which individually compute both the forward and backward pass for that micro-batch. Finally, weights are averaged after backpropagation has taken place on every GPU.
- *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 and split them across workers, also called "experts". A gating function helps each of the experts to specialize in a different subset of all training cases. The broader concept has been first introduced in 1991 [Jac+91] but has recently gained traction due to its flexibility in solving natural language problems.

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 [RG], 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:

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

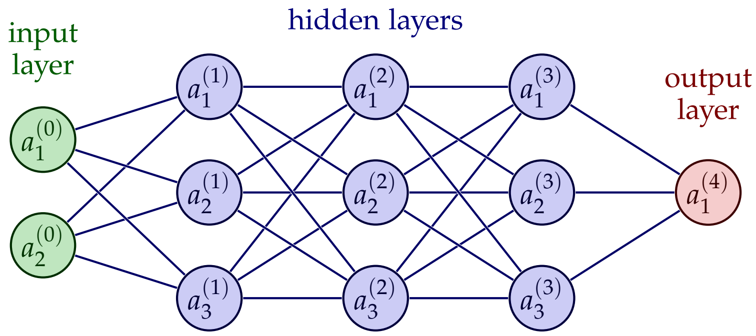
where g is called the activation function, $X \in \mathbb{R}^n$ is the neuron's input, $y \in \mathbb{R}$ is the output, $W \in \mathbb{R}^n$ its weights or parameters, and $b \in \mathbb{R}$ its bias. Note that activation functions must be non-linear functions. We would like to use NNs to solve and 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*.

2.1.2 Gradient Accumulation

When training neural networks, increasing batch size can sometimes lead to faster convergence [Kri14; Goy+17; YGG17a]. However, simply increasing the batch size during training does not scale very well, and can lead to memory issues or an inability to train at all. To help simulate bigger batches on a single device without incurring these issues, we can use a technique called *gradient accumulation*.

Algorithm 1 shows a simplified version of an algorithm used to train neural networks. A dataset consisting of a set of inputs X and Y labels, both with length N, is used as the input to the model, with a training loop supporting a single epoch.

Algorithm 1 Standard training algorithm, PyTorch style

Require: $N \geq 0$
  $X \leftarrow$ [...] ▷ Inputs
  $Y \leftarrow$ [...] ▷ Labels
  $i \leftarrow 0$
  while $i < N$ **do**
    $prediction \leftarrow model(X[i])$
    $loss \leftarrow$ criterion$(prediction, Y[i])$
    $loss$.backward()
    optimizer.step()
    optimizer.zero_grad()
    $i \leftarrow i + 1$
  end while

Algorithm 2 shows a modified version of Algorithm 1 with gradient accumulation turned on. This technique introduces an accumulation variable (ACC) instructing how many batches should be accumulated within the optimizer's state. The $loss$ value is normalized with the value of ACC to account for the effects of the accumulation. Once ACC batches have been aggregated or when we have finished training, we apply the averaged gradients using the optimizer's function $optimizer.step()$, and reset its gradients to zero.

2.2 Distributed Computing and Storage for Neural Networks

A distributed system is defined as a system where its components communicate with one another using messages. In computer science, the use of distributed systems has been a field of research for many years and has boomed with the advent of network communications. We often see distributed systems in complex applications such as the backend of a website or supercomputers. These systems are often composed of tens or hundreds of connected components that collectively produce one or more outcomes, such as web pages or complicated simulations. Using distributed techniques is essential for these complex applications, as the computational load and complexity required to run them, are simply too great for a single machine.

The elevated number of components that make a distributed system can lead to several issues, such as an immense amount of requests to their local storage and

Algorithm 2 Training with gradient accumulation, PyTorch style

Require: $N \geq 0$
Require: $ACC > 0$
  $X \leftarrow$ [...] ▷ Inputs
  $Y \leftarrow$ [...] ▷ Labels
  $i \leftarrow 0$
  while $i < N$ **do**
    $prediction \leftarrow model(X[i])$
    $loss \leftarrow$ criterion$(prediction, Y[i])$
    $loss$.backward()
    $loss \leftarrow loss/ACC$
    if $((i+1) \mod ACC == 0)$ or $((i+1) == N)$ **then**
      optimizer.step()
      optimizer.zero_grad()
    end if
    $i \leftarrow i+1$
  end while

extremely large files. For these reasons, it is necessary to use storage solutions that can handle these kinds of stress. Several distributed storage solutions have been developed and adopted over the past years, each of them excelling in different solutions and lacking in other areas. In general, choosing the right distributed storage software is crucial, as it can mean the difference between loading a dataset in seconds or minutes.

Neural networks have quickly become too large in terms of memory and computationally expensive for a single host, thus needing distributed system techniques for training.

2.3 Distributed Training

Training a state-of-the-art neural network nowadays requires enormous amounts of time and effort, but especially resources and money. It is often impossible to train a model within the boundaries of a single piece of hardware, even with powerful specifications. This affects the speed at which neural networks are being developed and the ability to reproduce results from state-of-the-art models.

One of the first and most notable examples of the results of this phenomenon is AlexNet [KSH12], in which the authors mentioned the need of using more than one GPU to train their model. This was 2010, and a little more than a decade later, to train the newest neural network models, practitioners, companies and research institutes need thousands of powerful GPUs. Distributed system techniques are used to be able to train these massive neural networks, creating the new term "distributed training".

We can split distributed training into three macro categories:

- **data parallelism**; this technique is conceptually the easiest and practically the most straightforward to implement. In data parallelism, the same model is being trained simultaneously in several distinct machines, aggregating the parameter updates through operations like *AllReduce*.
- **tensor parallelism**; some operations in neural network training like matrix multiplications may need big amounts of memory and computational power to be executed. With tensor parallelism, the operations themselves are split amongst several distinct machines, eventually aggregating their results into single or multiple ones, depending on the neural network structure.
- **hybrid parallelism**; whenever we want to train very big neural network models, depending only on one technique between data parallelism and tensor parallelism is often not enough to achieve good enough performance. In hybrid parallelism, the concepts and techniques of both data and tensor parallelism are combined.

2.3.1 Bottleneck Analysis

By definition, a *bottleneck* is a component of a system that negatively affects the output of such a system. An example is reading and writing data from one system to another, where the storage on both sides is a very fast SSD (Solid State Drive), and the transmission speed that the network allows is 56KB/s. In this scenario, even if both systems could theoretically reach unlimited speeds in both read and write operations, the performance of the system is ultimately limited by the low transmission speed.

Most real-life cases however are not so simple, and the component or components that might act as the bottleneck of a system cannot as easily be detected as the previous example. It is necessary then to perform an analysis of all components of the system to establish its bottlenecks.

Systems can be very complex, and performing an analysis of all of their components may become very difficult or impossible in practice. Thus, we have to analyze a limited number of components at a time, starting with the most significant. Systems can also be dynamic. In certain cases, bottlenecks can be found on a certain subset of components, and in other cases, other components act as bottlenecks instead. This requires constant monitoring of key metrics of these components and an understanding of which metrics might be relevant and why.

2.3.2 Metrics

Metrics represent the raw information about the behavior of a system and its component, collected via a monitoring system. In the previous paragraphs, we have presented a relatively simple example of how bottlenecks can be identified by their components and their characteristics. However, the final identification of the bottleneck was given to us by our pre-existing knowledge of the system and its specifications. In complex and evolving systems, having pre-existing knowledge about their components is not always possible.

It is possible to gather metrics about information that is available via common interfaces such as operating system hooks and methods. We may track metrics about the host system such as:

- **disk space utilization**; in addition to the amount of storage used over time, we can also track read and write operations in terms of speed.
- **cpu utilization**; can provide insights about possible efficiency issues within the software by highlighting peaks in CPU utilization.
- **network utilization**; distributed systems in particular may benefit from tracking metrics about network utilization, as it is one of their main components.
- **memory utilization**; sometimes processes do not perform proper cleanup of their resources, leading to memory leaks, slowdowns and errors.

2.4 Hivemind

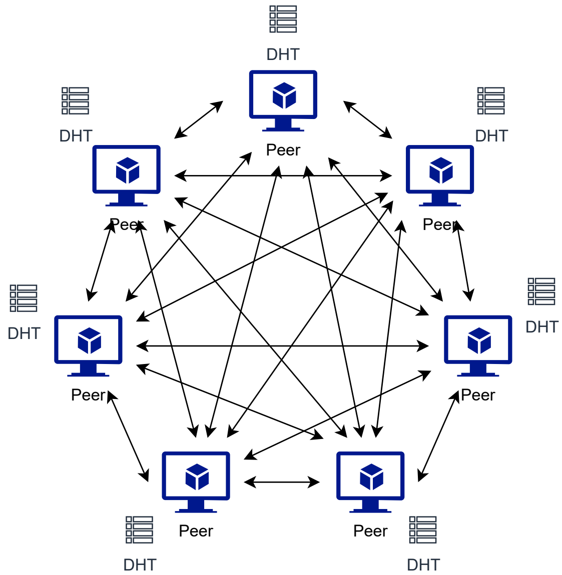
Hivemind is a framework that aims to enable decentralized training using several techniques, such as decentralized parameter averaging or decentralized mixture-of-experts. The initial concept of developing Hivemind was to enable neural network training outside the boundaries of an institution's internal network and resources, such as universities and small-medium companies. These institutions do not necessarily have access to the latest technologies and hardware to keep up with big companies' large budgets. By using Hivemind, it may be possible to collectively contribute to training neural networks across different institutions, unlocking new areas and possibilities of research.

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. An example of how this type of communication works is given in Figure 2.2.

Figure 2.2: Nodes in a Peer-to-Peer network exchanging data directly with one another. Every node has its own internal DHT which is kept in sync with other peers.



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` class. There are several options for tuning the Hivemind Optimizer, which affect how distributed training is performed across participating peers in a training run. We will briefly describe some of the settings that we focused on during this thesis.

Target Batch Size

The target batch size (TBS) is defined in the Hivemind Optimizer as the global number of samples that every peer has collectively processed during a collaborative run in the current Hivemind epoch. A Hivemind epoch, which we will call HE, does not necessarily correspond to a full pass over the training data and can be used to synchronize advanced training features like optimizer schedulers. The HE starts at 0 at the beginning of training and increases by one every time that all participating peers reportedly finished processing TBS since the last HE.

Understanding the concept of HE is crucial, so let us describe a simple scenario to aid with this task:

- two peers are collaboratively training a model;
- the TBS is 256;
- the `use_local_updates` setting is set to `True` (see next section for a definition of this setting);
- 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.

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 total number of accumulated samples is now 256 and it is equal to TBS, each peer can now initiate an averaging round with all other participating peers. What happens right before and after averaging depends on the `use_local_updates` setting.

Local Updates

Training neural networks with larger batches has been proven to be beneficial in some cases [Kri14; Goy+17; YGG17a]. Thus, practitioners may want to be able to accumulate gradients either locally, in a distributed manner, or a combination of both. Hivemind's implementation of distributed training extends the concept of gradient accumulation, previously presented in Subsection 2.1.2, by introducing the `use_local_updates` settings for `hivemind.Optimizer`.

The setting has two modes:

- **activated**; after every local training step, the Hivemind optimizer applies the gradients to the model. At the next HE, the final stage of the Optimizer starts. A simplified flow of how Hivemind implements this mode is illustrated in Figure 2.3.
- **deactivated**; after every local training step, gradients are accumulated instead of being applied to the model's parameters. After *ACC* steps, the optimizer is invoked and internally stores the accumulated gradients. At the next HE, the Optimizer averages the accumulated gradients with all participating peers. The final stage of the Optimizer then starts, which averages the model and optimizer state with other peers. A simplified flow of how Hivemind implements this mode is illustrated in Figure 2.4.

In this thesis, we will discuss the effects of enabling or disabling this setting on training.

Figure 2.3: Nodes in a Peer-to-Peer network exchanging data directly with one another. Every node has its own internal DHT which is kept in sync with other peers.

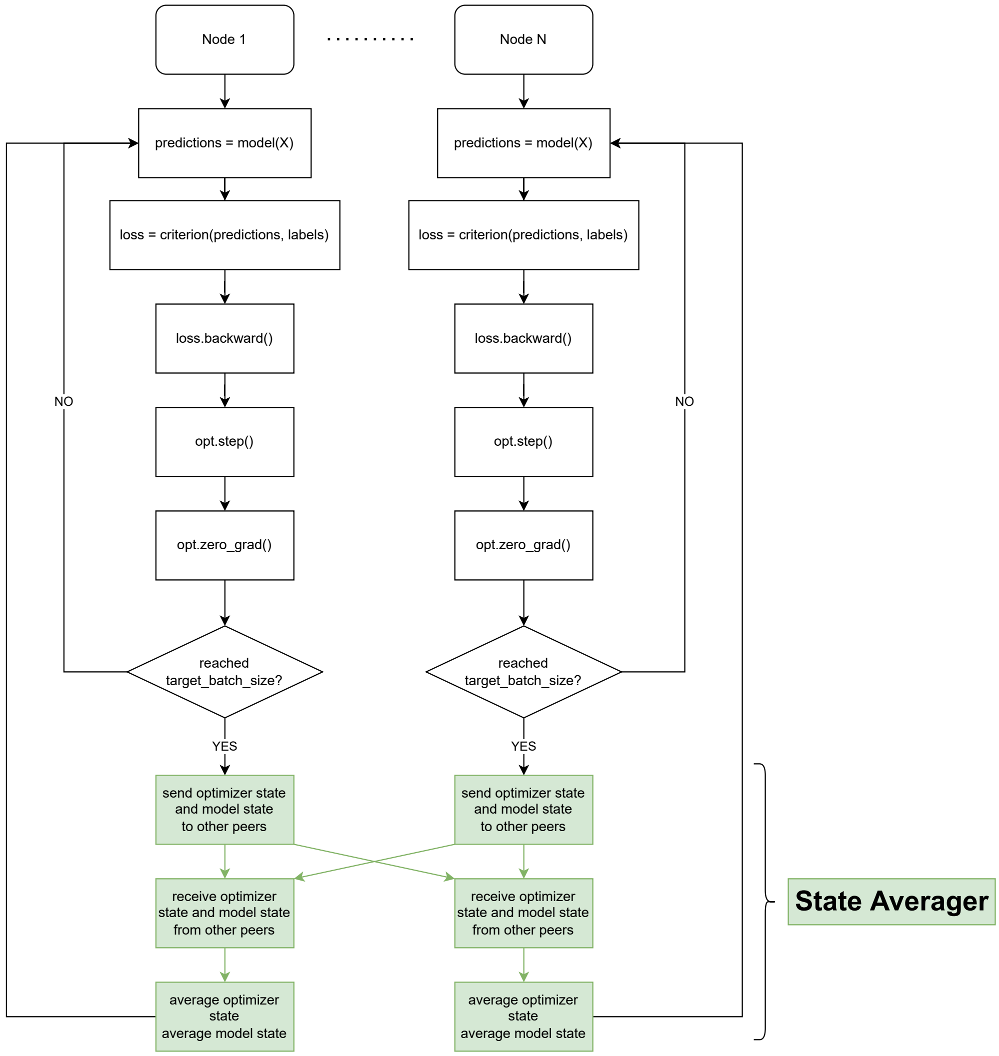
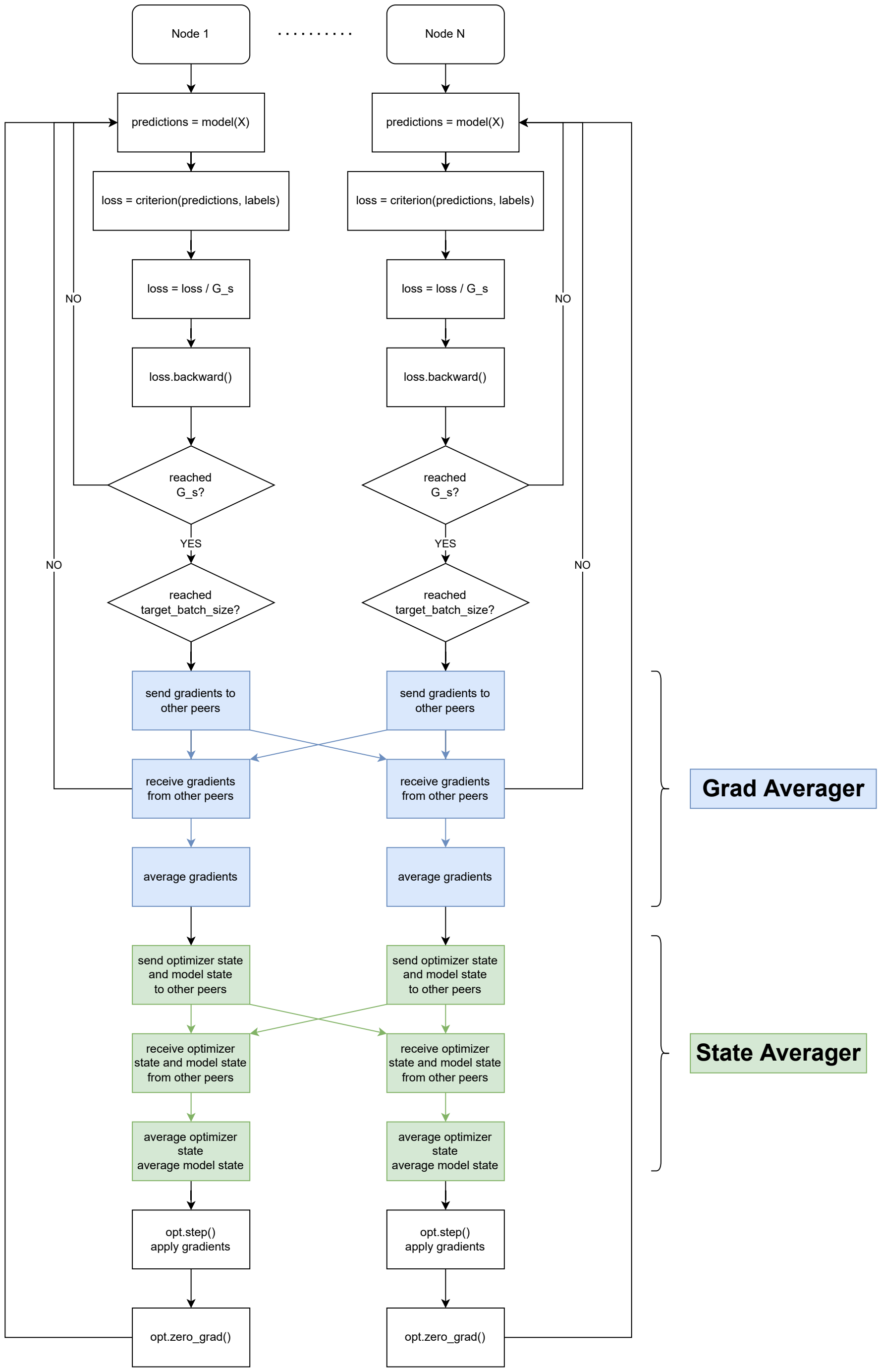


Figure 2.4: Nodes in a Peer-to-Peer network exchanging data directly with one another. Every node has its own internal DHT which is kept in sync with other peers.



3 Related Work

Data parallelism has been leveraged to improve the performance of machine learning systems [KSH12]. Several common frameworks implement data parallelism such as PyTorch [1] and TensorFlow[2] through a high-level API. There are already some works that analyze the effects of data parallelism [LTJ20]

Talk about distributed training, model and data parallelism works. Talk about Hivemind paper and papers that Hivemind bases itself upon. Basically explain why this thesis is needed at all.

[1]https://pytorch.org/
[2]https://tensorflow.org/

4 Setup

In this chapter, we briefly describe the technologies and resources that we have used to run our experiments, as well as how they were set up and provisioned. We further describe our experimental setup and the hyperparameters that we chose to experiment on. Finally, we present a high-level overview of our implementation with a simplified visualization.

4.1 Experimental Setup

All our experiments are trained using the model ResNet18 [He+15] on Imagenet-1k, with 1.281.167 items and 1000 classes [Den+09]. The optimizer function of choice is standard stochastic gradient descent (SGD) with three possible learning rate settings $(0.1, 0.01, 0.001)$ and a fixed momentum value of 0.9.

To support our training experiments we used a cluster provided by the department of Decentralized Information Systems and Data Management, from the Technische Universität München. This cluster is managed by OpenNebula, and gives us access to several machines with Intel Xeon v3 8x@2.4 Ghz, 80 GB DDR4 RAM and Ubuntu 20.04 image. As the storage backend, we use a CEPH cluster backed with hard disks, with 10 GB/s up and downlink. We repeat every baseline experiment four times, and every hivemind experiment only one time. All experiments use Python 3.8.10, and Hivemind with the commit hash `de6b4f5ae835a633ca7876209f2929d069e988f0` [1]. We used the Infrastructure-as-Code (IaC) tool Terraform together with the OpenNebula provider 1.0.1 [2] to spin up several virtual machines matching our needs.

The types of virtual machines that we used for this thesis are two:

- **messengers**; helps with establishing the initial connection between every bee. Does not produce or consume any data besides the initial connection step.
- **bees**; machines that participate in training a model. Bees can be executed with Hivemind turned on or off, the latter being the default setting when running

[1] We chose this specific commit because we identified some issues with training on our setup from the next commit onwards

[2] https://registry.terraform.io/providers/OpenNebula/opennebula/1.0.1

baseline experiments. When executing with Hivemind on, bees connect to a single messenger machine for initializing their internal DHT and then proceed to communicate with one another for the rest of the experiments.

Every virtual machine spawned has 10GB of RAM and 30GB of internal disk space backed up by SSD, and they are all connected to the same CEPH storage backend presented earlier. Messenger machines always have one vCPU assigned to them by the underlying OpenNebula KVM. Depending on the experiment, bees can either be assigned with 16vCPUs, 8vCPUs, 4vCPUs, 2vCPUs or 1vCPUs.

It is worth noting that because of the nature of how a KVM assigns virtual CPUs to a virtual machine, the machines with 1vCPU may occasionally use more than one thread in case of long I/O waiting times. This causes metrics such as CPU utilization to go over 100%, although at all times always one core is utilized by the virtual machine. With our current setup, there is no way to go around this limitation.

To log our metrics, we decided to use the tool Weights and Biases (*wandb*) [3]. The impact of this tool on the logged metrics on Hivemind experiments is later considered when compared to baseline experiments.

4.2 Metrics

We logged key metrics from the host system of every machine using the Python tool `psutil`, which gives us access to the metrics listed in Table 4.1. Not every metric will be used and analyzed throughout this thesis.

Table 4.1: List of key host metrics logged using `psutil`.

Metric key	Description
`bandwidth/disk_read_sys_bandwidth_mbs`	bandwidth used by local disk read operations
`bandwidth/disk_write_sys_bandwidth_mbs`	bandwidth used by local disk write operations
`bandwidth/net_sent_sys_bandwidth_mbs`	bandwidth used by network send operations
`bandwidth/net_recv_sys_bandwidth_mbs`	bandwidth used by network receive operations
`cpu/interrupts/ctx_switches_count`	number of context switches that occurred since the last call
`cpu/interrupts/interrupts_count`	number of CPU interrupts that occurred since the last call

[3] https://wandb.ai/

cpu/interrupts/soft_interrupts_count	number of soft CPU interrupts that occurred since the last call
cpu/load/avg_sys_load_one_min_percent	average CPU load across the last minute
cpu/load/avg_sys_load_five_min_percent	average CPU load across the last five minutes
cpu/load/avg_sys_load_fifteen_min_percent	average CPU load across the last fifteen minutes
cpu/logical_core_count	number of logical cores available to the current host
memory/total_memory_sys_mb	total amount of memory in megabytes available to the current host
memory/available_memory_sys_mb	amount of unused memory in megabytes since the last call
memory/used_memory_sys_mb	amount of used memory in megabytes since the last call
memory/used_memory_sys_percent	percent of memory used since the last call
process/voluntary_proc_ctx_switches	number of voluntary process context switches since the last call
process/involuntary_proc_ctx_switches	number of involuntary process context switches since the last call
process/memory/resident_set_size_proc_mb	resident set size in megabytes of the current process since the last call
process/memory/virtual_memory_size_proc_mb	virtual memory size in megabytes of the current process since the last call
process/memory/shared_memory_proc_mb	shared memory size in megabytes of the current process since the last call
process/memory/text_resident_set_proc_mb	memory devoted to executable code in megabytes since the last call
process/memory/data_resident_set_proc_mb	physical memory devoted to other than code in megabytes since the last call
process/memory/lib_memory_proc_mb	memory used by shared libraries in megabytes since the last call
process/memory/dirty_pages_proc_count	number of dirty pages since the last call
disk/counter/disk_read_sys_count	how often were reads performed since the last call
disk/counter/disk_write_sys_count	how often were writes performed since the last call
disk/disk_read_sys_mb	how much was read in megabytes since the last call

disk/disk_write_sys_mb	how much was written in megabytes since the last call
disk/time/disk_read_time_sys_s	how much time was used to read in seconds since the last call
disk/time/disk_write_time_sys_s	how much time was used to write in seconds since the last call
disk/time/disk_busy_time_sys_s	how much time was used for I/O operations in seconds since the last call

To monitor the effects of Hivemind on training, we also log at the end of every training step the metrics listed in Table 4.2

Table 4.2: List of key host metrics logged using `psutil`.

Metric key	Description
train/loss	Loss reached in the current step
train/accuracy	Accuracy reached in the current step
train/samples_ps	number of samples processed per second passed from the start of the current step until the end
train/data_load_s	time taken to load the current step batch in seconds
train/model_forward_s	time taken to perform the forward pass in seconds
train/model_backward_only_s	time taken to perform the backward pass in seconds
train/model_opt_s	time taken to perform the optimizer step in seconds
train/step	current step number

4.3 Implementation

To perform the experiments, we developed a custom solution that automates most manual steps using a combination of Ansible playbooks and bash scripts.

When setting up a new experiment, the following steps are performed:

1. copy configuration to all participating machines; this step includes the messenger machine, as well as the bee machines. This is to ensure that all machines are using the same code.

2. (only for Hivemind) start the messenger machine; the messenger acts as the first point of contact for all the machines, providing a common endpoint that they can connect to for establishing the initial connection.

3. run bees; there are two cases for this step: a) if step 2 was performed, bees are running using Hivemind. All bees run with the parameter `initial_peers` set to

the messenger's DHT address. b) otherwise, all bees are performing a baseline experiment, and no Hivemind feature is enabled.

5 Experiments

Having access to more powerful hardware is a challenge for several reasons that may be outside of our control. In the past years, there was a worldwide shortage of microchips that negatively impacted the ability to purchase state-of-the-art hardware such as CPUs and GPUs. Thus, we would like to understand the effects of running distributed frameworks such as Hivemind on less powerful, older hardware.

To provide a fair comparison between experiments not running Hivemind and experiments that do, our experiments always have the same amount of vCPUs. Finally, every experiment processes the same number of samples across all the participating peers, and the sum of processed samples may never be greater than 320,000.

An exception to this rule is made for experiments with an odd number of samples per peer. For example, a run with batch size 128 and 8 peers should result in 312.5 samples per peer, which is not possible. In these cases, the number of samples per peer is rounded up to the nearest digit to form an even number. This chapter describes the basic setup of our experiments.

5.1 Base Case

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. We designed our baseline experiments in a grid search, covering 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.
- Gradient Accumulation Steps (GAS): 1 (no accumulation), 2 (with accumulation up to two steps)

Table 5.1: List of baseline experiments and hyperparameters

Baseline experiments			
Max Steps	Batch Size	Learning Rate	Grad. Acc. Steps
10.000	32	0.001	1
10.000	32	0.01	1
10.000	32	0.1	1
5000	64	0.001	1
5000	64	0.01	1
5000	64	0.1	1
2500	128	0.001	1
2500	128	0.01	1
2500	128	0.1	1
10.000	32	0.001	2
10.000	32	0.01	2
10.000	32	0.1	2
5000	64	0.001	2
5000	64	0.01	2
5000	64	0.1	2
2500	128	0.001	2
2500	128	0.01	2
2500	128	0.1	2

The machines used for baseline runs have 16vCPUs and each experiment is repeated 4 times to observe the reproducibility of the measurements. Hivemind features such as the DHT and the Optimizer wrapper are completely deactivated for these runs. Table 5.1 lists all the 18 combinations of experiments that we cover in this thesis.

5.2 Not-Baseline Case

To test and isolate the effects of using Hivemind for distributed training, every experiment 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.

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, and further explore the following Hivemind settings and questions:

- **Number of Peers (NoP)**: 2, 4, 8 and 16; for loads like the experiment that we are running, is communication between many nodes a bottleneck?
- **Number of logical cores per node (vCPUs)**: 1, 2, 4, 8 and 16; using the same amount of computational power across many nodes, do we get to a target loss faster?
- **Target Batch Size (TBS)**: 10.000, 5000, 2500, 1250 and 625; using smaller target batch size, do we get faster to the target loss?
- **Max Steps (MS)**: 5000, 2500, 1250 and 625; this parameter depends on the number of peers and batch size, but the total is always 320.000 steps per experiment;
- **Use Local Updates (LU)**: `True` or `False`; Hivemind allows us to control when to schedule gradient, model and parameter averaging. How does this setting affect training?

The Table 5.2 shows a list of the combination of experiments that we performed to test Hivemind. In total, we have executed 288 experiments for this thesis.

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

Experiments testing for the effect of target batch size (TBS)							
MS	NoP	vCPUs	BS	LR	TBS	GAS	LU
5000	2	8	32	0.001, 0.01, 0.1	10.000, 5000, 2500, 1250, 625	1,2	T, F
2500	2	8	64	0.001, 0.01, 0.1	10.000, 5000, 2500, 1250, 625	1,2	T, F
1250	2	8	128	0.001, 0.01, 0.1	10.000, 5000, 2500, 1250, 625	1,2	T, F
Experiments testing for the effect of the number of peers (NoP)							
MS	NoP	vCPUs	BS	LR	TBS	GAS	LU
2500	4	4	32	0.001, 0.01, 0.1	1250	1,2	T, F
1250	8	2	32	0.001, 0.01, 0.1	1250	1,2	T, F
625	16	1	32	0.001, 0.01, 0.1	1250	1,2	T, F
1250	4	4	64	0.001, 0.01, 0.1	1250	1,2	T, F
625	8	2	64	0.001, 0.01, 0.1	1250	1,2	T, F
313	16	1	64	0.001, 0.01, 0.1	1250	1,2	T, F
625	4	4	128	0.001, 0.01, 0.1	1250	1,2	T, F
313	8	2	128	0.001, 0.01, 0.1	1250	1,2	T, F
157	16	1	128	0.001, 0.01, 0.1	1250	1,2	T, F

We have fixed the following Hivemind hyperparameters that were not the focus of this thesis, although some can be further explored:

- **matchmaking_time**; defines for how many seconds the optimizer should wait for other peers to join an averaging round. We set this value to 10. Ideally, optimizers should never have to wait for this long amount of time in our setup.
- **averaging_timeout**; after this many seconds, an averaging round is canceled. We set this value to 300. This high value is not encouraged by the Hivemind framework, as it may cause optimizers to hang in case of network errors. However, because we have a controlled environment with low latency, setting this to a high value allows us to quickly determine issues with our setup and intervene by re-running the experiments.
- **grad_compression**; defines which class to use for gradient compression. We set this to `hivemind.NoCompression` for every run, as exploring the effects of compression for gradients is outside the focus of this thesis. Other works have focused on the effects of data sparsity and data parallelism [LTJ20].

5.3 Metrics comparison framework

In the next chapter, we will compare training and system metrics between baseline and Hivemind runs. However, they are not directly comparable.

Hivemind runs involve more than one machine per experiment, with each machine completing its task earlier or later than other peers that are in the same training network. We assume that an experiment ends when the last peer finishes processing the samples it has been assigned. Thus, for Hivemind runs we chose to use the maximum runtime amongst all peers to be used for comparison and analysis.

For training loss, we always select the mean training loss reached by the baseline re-runs, and the minimum loss reached by all peers for Hivemind runs. We chose the minimum for the Hivemind runs because the model being trained is essentially one. When saving the model for inference purposes, the selection of which peer's model to pick should lie on the model with the minimum loss.

We also use the average for system metrics such as bandwidth received and sent, CPU load and RAM usage for both baseline and Hivemind runs. This is because all nodes in our controlled Hivemind experiments behave more or less the same. In real-life scenarios such as training across the internet, where latency and peer behavior is unpredictable, this assumption would not be possible.

6 Results

In this thesis, we are not looking to obtain the best possible combination of hyperparameters for training loss or model accuracy. Instead, we want to observe the effects on training with Hivemind when tuning common hyperparameters such as batch size and learning rate and Hivemind hyperparameters such as the TBS. In this thesis, we analyze the performance and limits of training using Hivemind rather than looking for the best model.

6.1 Baseline runs

We begin this chapter by showing the results that we have obtained with the baseline runs. As mentioned previously in chapter 4, all baseline experiments are executed on machines with the same configuration, and the total number of samples processed is always the same. Figure 6.1 shows the average runtimes for baseline runs in minutes.

Baseline runs do not run distributed algorithms, all Hivemind features are switched off and machines do not communicate with each other. However, Figure 6.2b shows that there is some network activity. On average, every machine receives a constant 1.5 MB/s of data on its network. This may be due to several factors, such as KVM management data, OpenNebula pings, and CEPH data being read.

In the Setup section, we also introduced our monitoring tool of choice *wandb*. Because this is an online monitoring tool, some data about our runs is periodically sent to the Weights and Biases server for storage and visualizations.

In Figure 6.2a, which shows the bandwidth used for send operations across all baseline runs, we can observe the bandwidth in MB/s used for each run. On average, this is roughly 0.02 MB/s on every run, a value that can be mostly attributed to *wandb* and other background monitoring operations such as OpenNebula.

In future sections, we will always account for these effects when performing comparisons with baseline runs.

Figure 6.3 shows the average times for data load, forward pass, backward pass and optimization step across batch sizes in baseline runs for both GAS=1 and GAS=2. As we might expect, the time it takes for a single step to complete is linearly dependent on the batch size. The learning rate (LR) does not affect the time it takes for each step to complete, so we aggregated the runs for each batch size. By contrast, the number

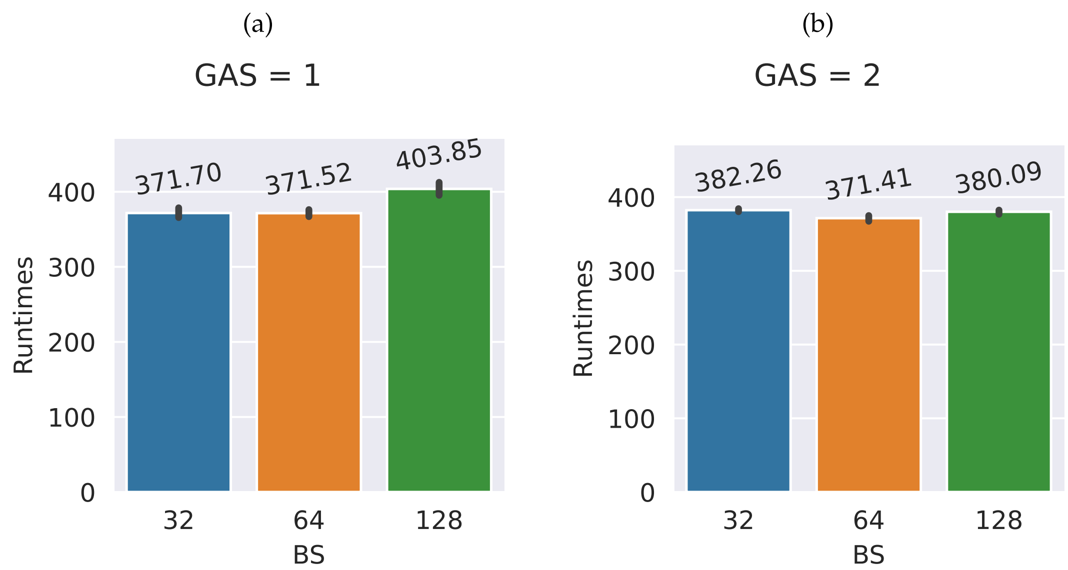


Figure 6.1: Average runtimes of baseline experiments in minutes. Runs are aggregated across LR, with the standard deviation amongst reruns as the black bar.

of gradient accumulation steps (GAS) seems to shave off some time for every batch size, although the total runtimes in Figure 6.1 do not seem to reflect this improvement. Throughout this chapter, we will keep showing GAS runs separately, as it still might affect some other aspects of training.

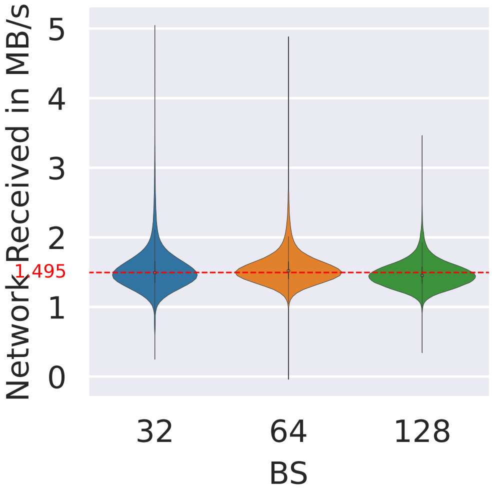
6.2 Focus on effects of batch size, learning rate and target Batch Size

Batch size and learning rate are some of the most fundamental hyperparameters to tune when training a neural network to obtain good training results. Tuning the learning rate should not impact training performance directly, but it can help to better understand how to tune it for different settings combinations while using Hivemind. As specified previously in chapter 4, the reference optimizer algorithm is the stochastic gradient descent (SGD), which is wrapped around the `hivemind.Optimizer` class.

The batch size determines how many samples are being processed in a training loop. In Hivemind, this has the consequence of reaching the TBS in fewer steps, but not necessarily in less time.

Figure 6.4 shows the runtimes for Hivemind experiments with 2 peers and 8vCPUs

(a) Network bandwidth sent in MB/s for baseline runs. Values above 0.07 are hidden. Runs are aggregated across LR.



(b) Network bandwidth received in MB/s for baseline runs. Values above 5 are hidden. Runs are aggregated across LR.

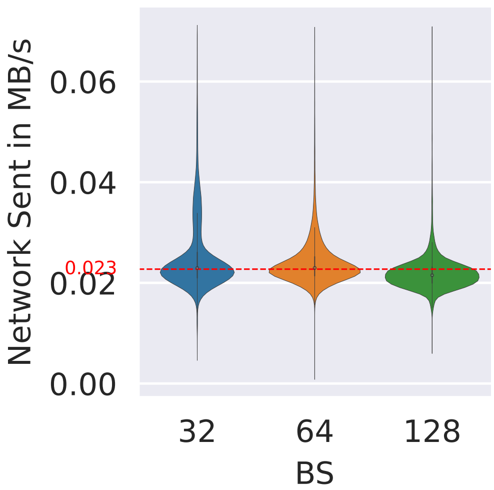


Figure 6.2: Network bandwidth sent and received in MB/s for baseline runs. Runs are aggregated across LR.

per peer compared to the baseline runs. Every run shows a substantial decrease in runtime, with $BS = 32$ having an average decrease of circa 20%, $BS = 64$ of circa 30% and close to 40% for $BS = 128$. But can we just expect such a high increase in performance for free when turning on Hivemind? There are two important factors to take into consideration before making a such claim.

1. Data loading in the baseline runs takes 1/3 of the total time per step as shown in Figure 6.3. Parallelizing data loading indeed speeds up the overall runtime for each run. With further experimentation that is outside the scope of this thesis, it might be possible to reduce the data loading step with local parallelization techniques and faster storage. Reducing the data loading step might help rule out the possibility that we only see runtime improvements because of the effects of loading more data in parallel.

2. The results in Figure 6.5 shows the hidden impact on loss of using Hivemind. Nearly all experiments are not able to reach the minimum loss set by the respective baseline runs. Some argue [YGG17b] that it is possible to reach the same model accuracy just by training longer. Others [Kes+16] argue that longer training with

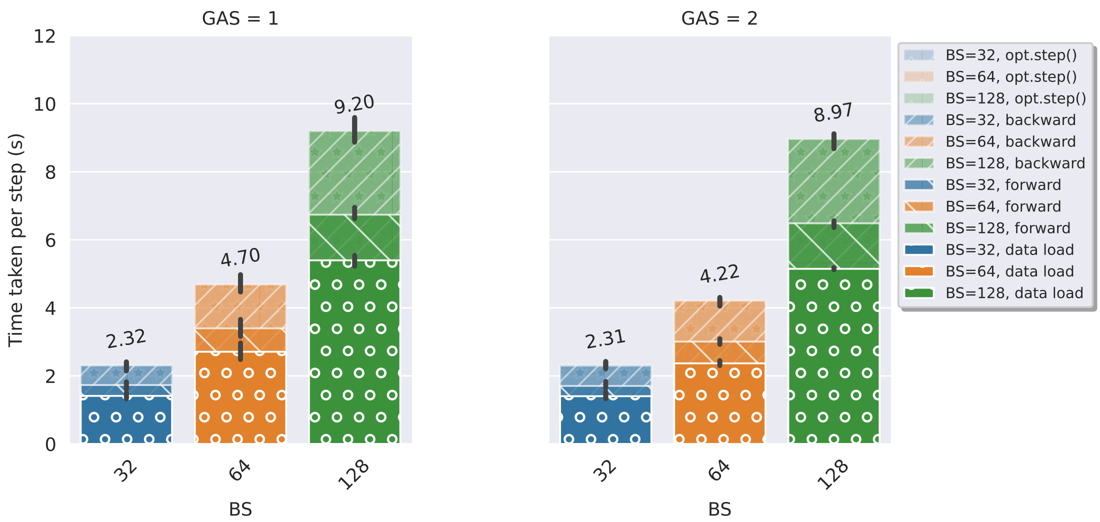


Figure 6.3: Average times of step data load (small circles), forward pass (backward slash), backward pass (forward slash) and optimization step (stars) baseline experiments in seconds. Runs are further aggregated across LR and the standard error amongst runs is shown with black bars.

larger batch sizes might lead to overall worse generalization capabilities for the model. Proving the effects on accuracy and model generalization is beyond the scope of this thesis.

Depending on the optimizer used for training a neural network model, the number of parameters can become huge. When performing an optimizer state averaging state, sending a high amount of parameters can lead to high communication overhead, and thus, reduced performance. We can see this effect in Figure 6.7 and Figure 6.8. As the batch size increases, nodes send and receive more data, increasing bandwidth utilization.

Considerations of training with Hivemind for the TBS, BS and LR hyperparameters:

- With the same amount of computational power overall, training with Hivemind might need more time to reach the loss compared to the baseline runs.
- Having access to less powerful hardware still allows training peers to be helpful, at the cost of training for longer.
- Increasing the frequency of averaging does not make up for a bad selection of optimization hyperparameters such as the batch size and learning rate.

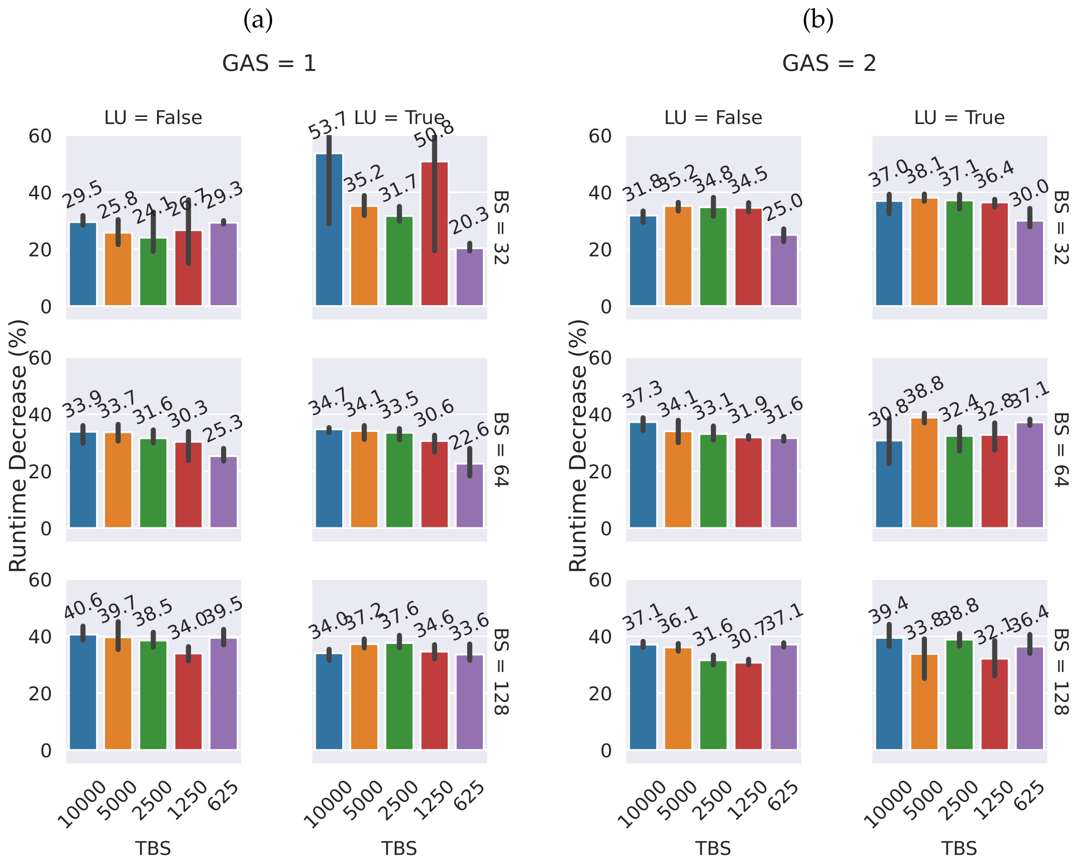


Figure 6.4: Runtime decrease in percent for Hivemind runs with 2 peers and 8vCPUs relative to baseline runs. Higher is better. Runs are aggregated across LR and the standard error amongst runs is shown with black bars.

- However being able to perform averaging steps more frequently can help to reduce the loss gap with the baseline runs.

6.3 Focus on effects of gradient accumulation

6.4 Focus on effects of local updates

6.5 Focus on effects of the number of peers and vCPUs per peer

We have seen how different Hivemind configurations can affect training on two peers. However, it might be useful to dedicate more than two machines to train deep neural networks. In this section, we answer the following question: what are the effects of scaling up the number of machines when using Hivemind?

The frequency at which peers average their model state is directly proportional to the number of peers, the throughput per second of each peer and the TBS. In turn, the throughput per second is affected by several factors such as the BS, computational power of the node and wait times for I/O operations.

It might be difficult to isolate the effects of introducing more nodes from scaling the target batch size. Thus, we decided to fix the target batch size to 1250 for this set of experiments and alter TBS, BS, LR, GAS and LU.

Similar to the Hivemind runs with 2 peers and 8vCPUs presented in section 6.2, Figure 6.9 shows

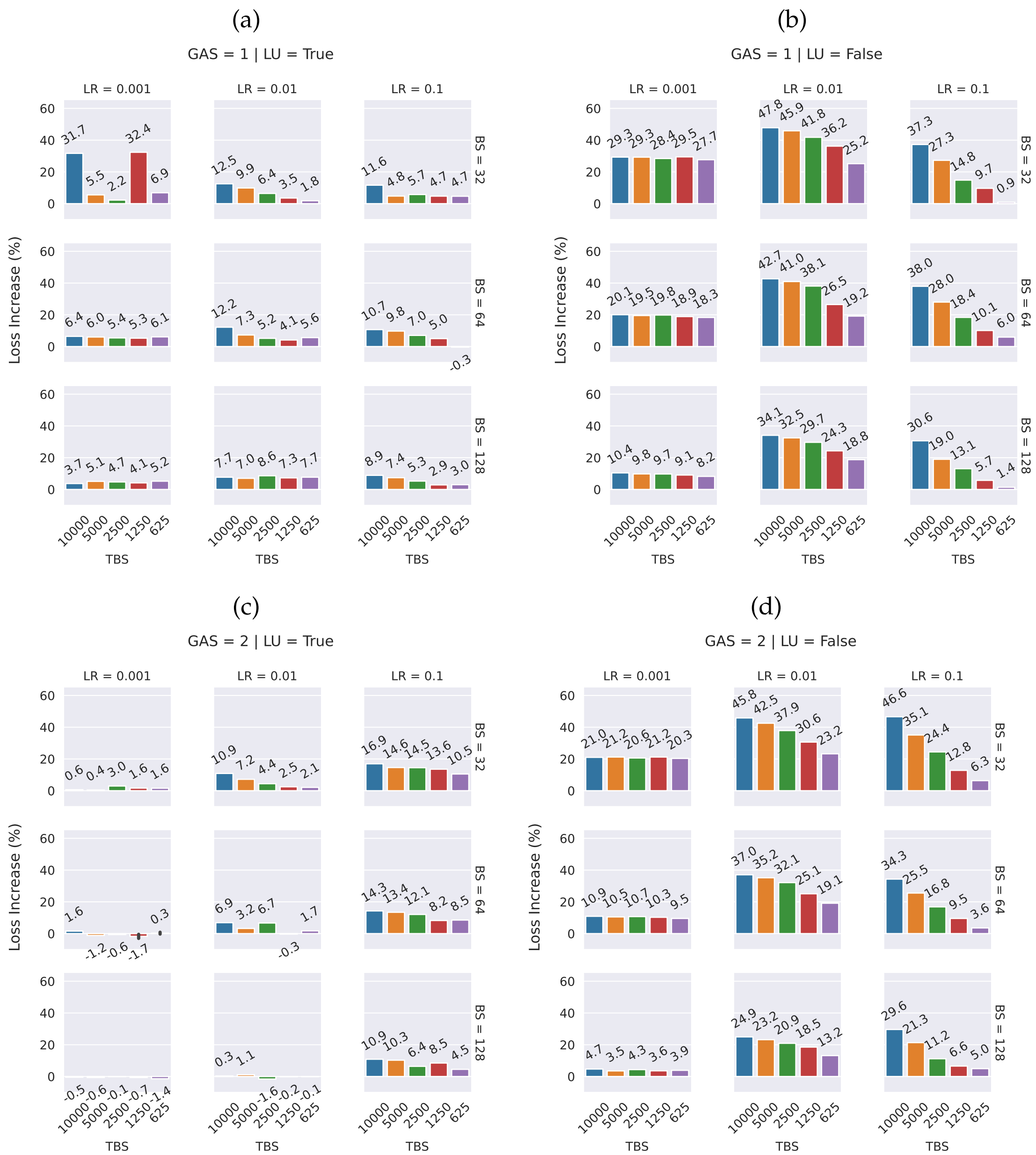
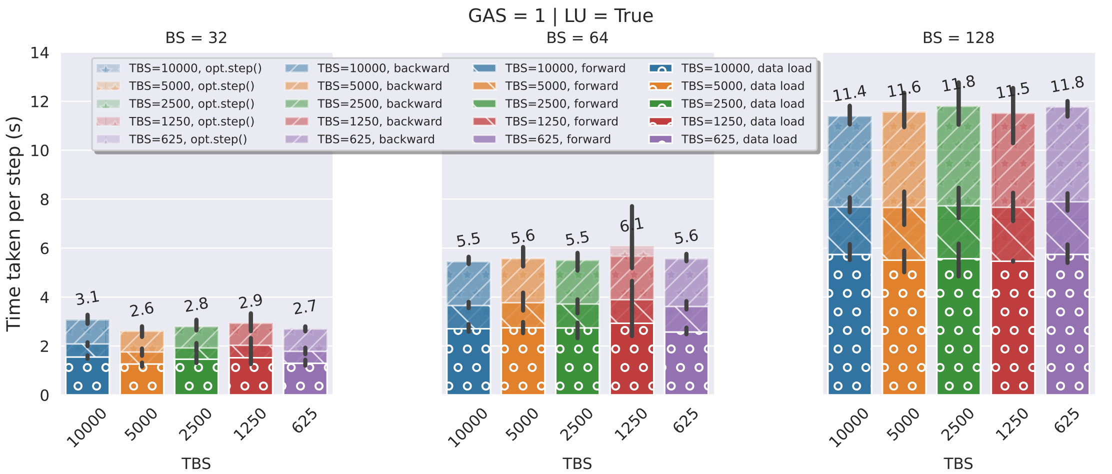


Figure 6.5: Loss increase in percent for Hivemind runs with 2 peers and 8vCPUs relative to baseline runs. Higher is worse.

(a)



(b)

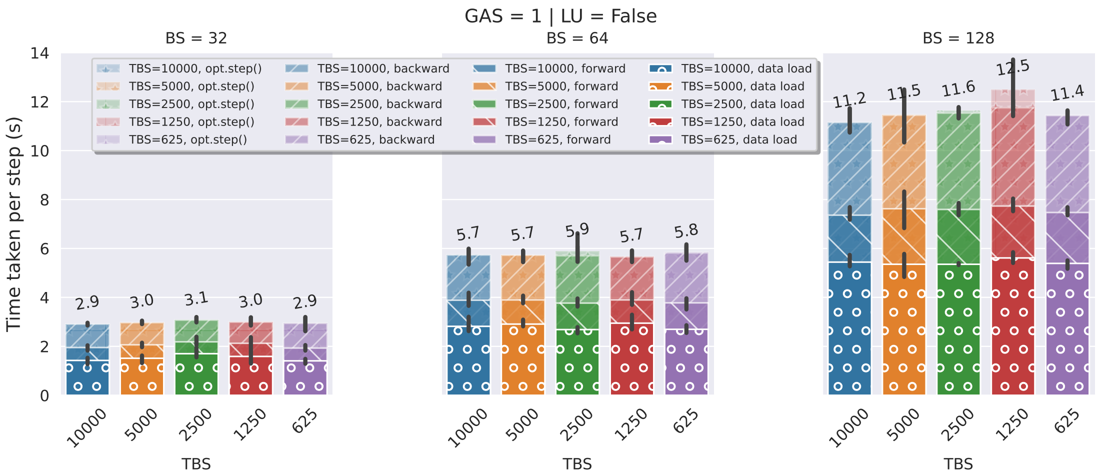
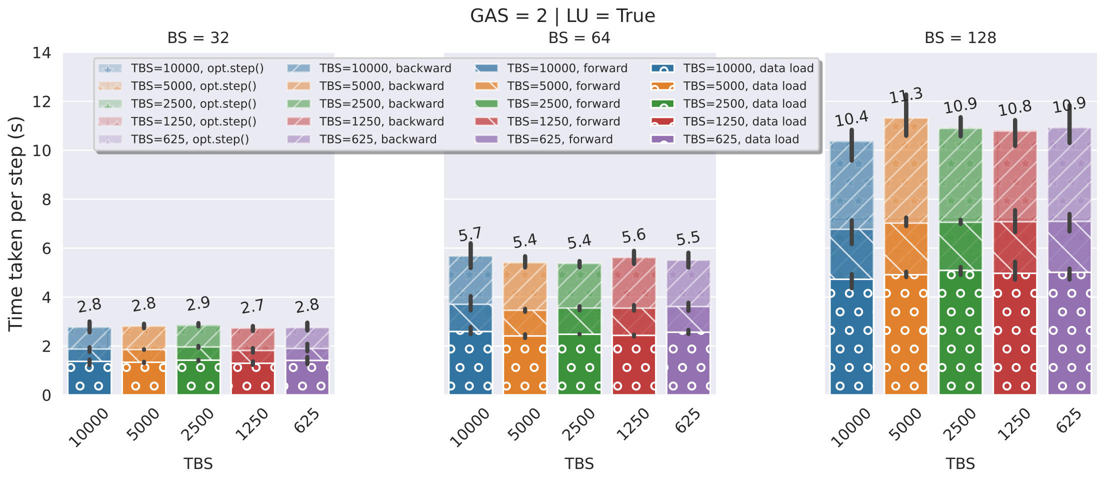


Figure 6.6: Average times of data load (small circles), forward pass (backward slash), backward pass (forward slash) and optimization step (stars) baseline experiments in seconds. Runs are further aggregated across LR and the standard error amongst runs is shown with black bars (continues).

(c)



(d)

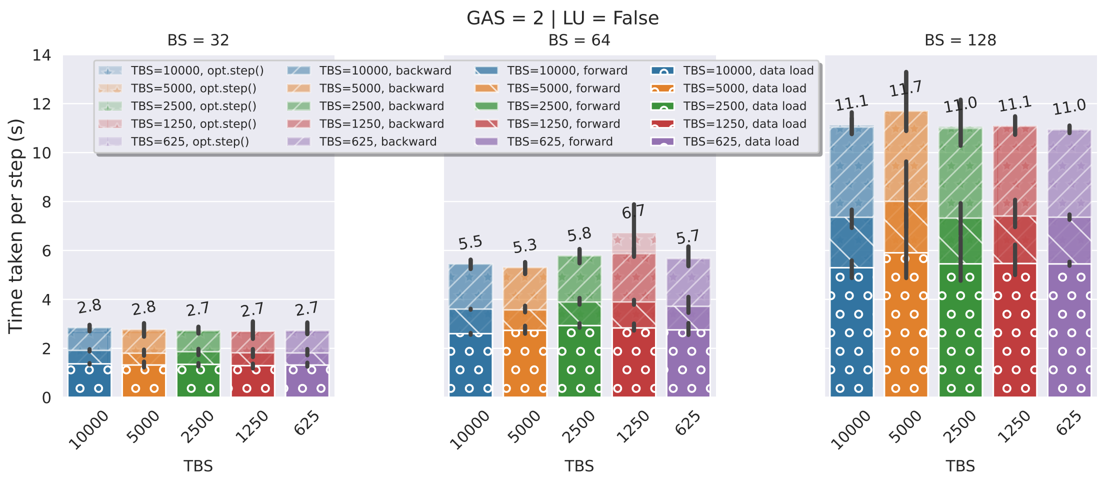


Figure 6.6: Average times of data load (small circles), forward pass (backward slash), backward pass (forward slash) and optimization step (stars) baseline experiments in seconds. Runs are further aggregated across LR and the standard error amongst runs is shown with black bars.

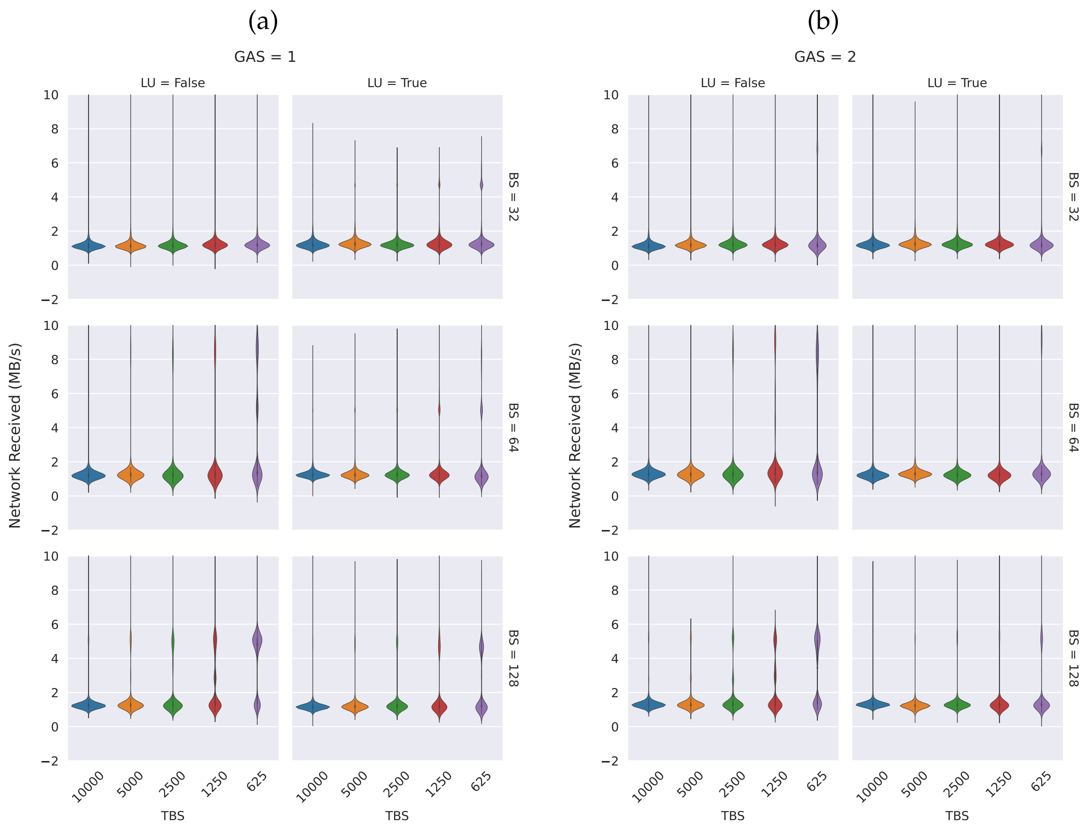


Figure 6.7: Network received for Hivemind runs with 2 peers and 8vCPUs. Values ≥ 10 MB/s are hidden and runs are aggregated across LR.

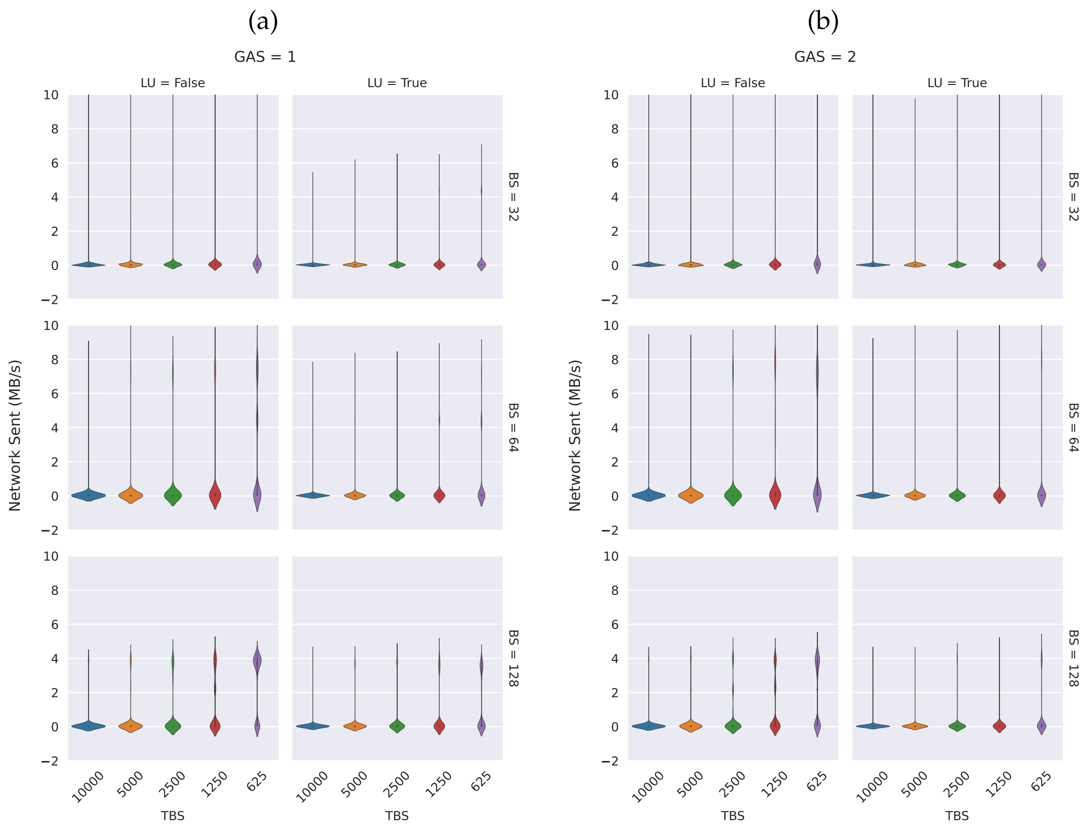


Figure 6.8: Network sent for Hivemind runs with 2 peers and 8vCPUs. Values ≥ 10 MB/s are hidden and runs are aggregated across LR.

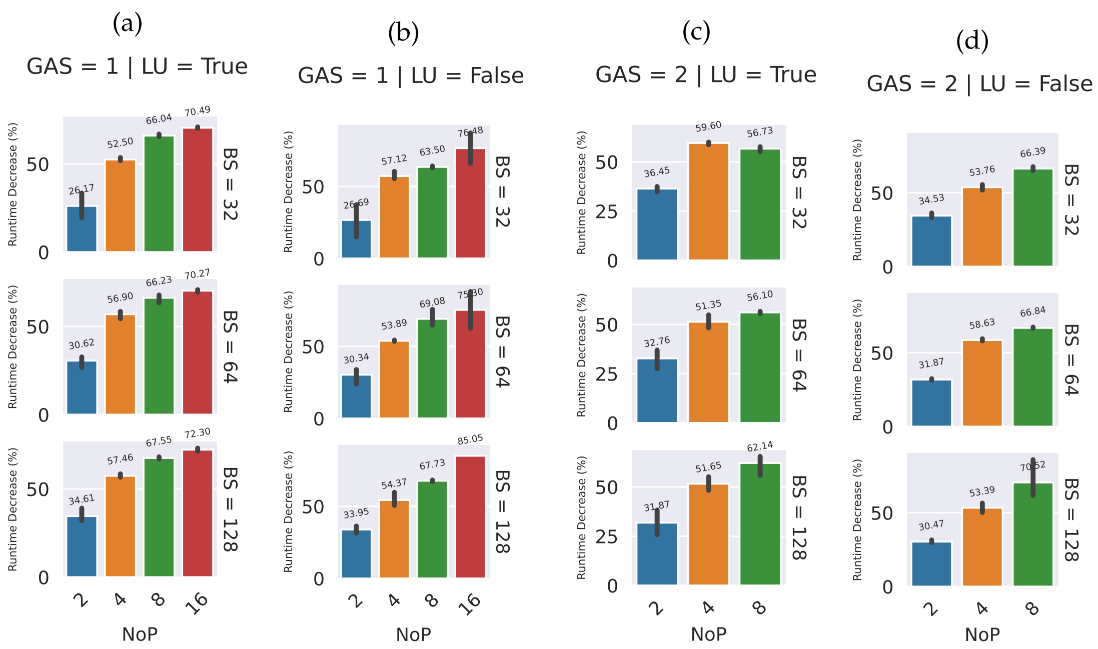


Figure 6.9: Runtime decrease in percent for Hivemind runs with different number of peers and vCPUs relative to baseline runs. Higher is better. Runs are aggregated across LR and the standard error amongst runs is shown with black bars.

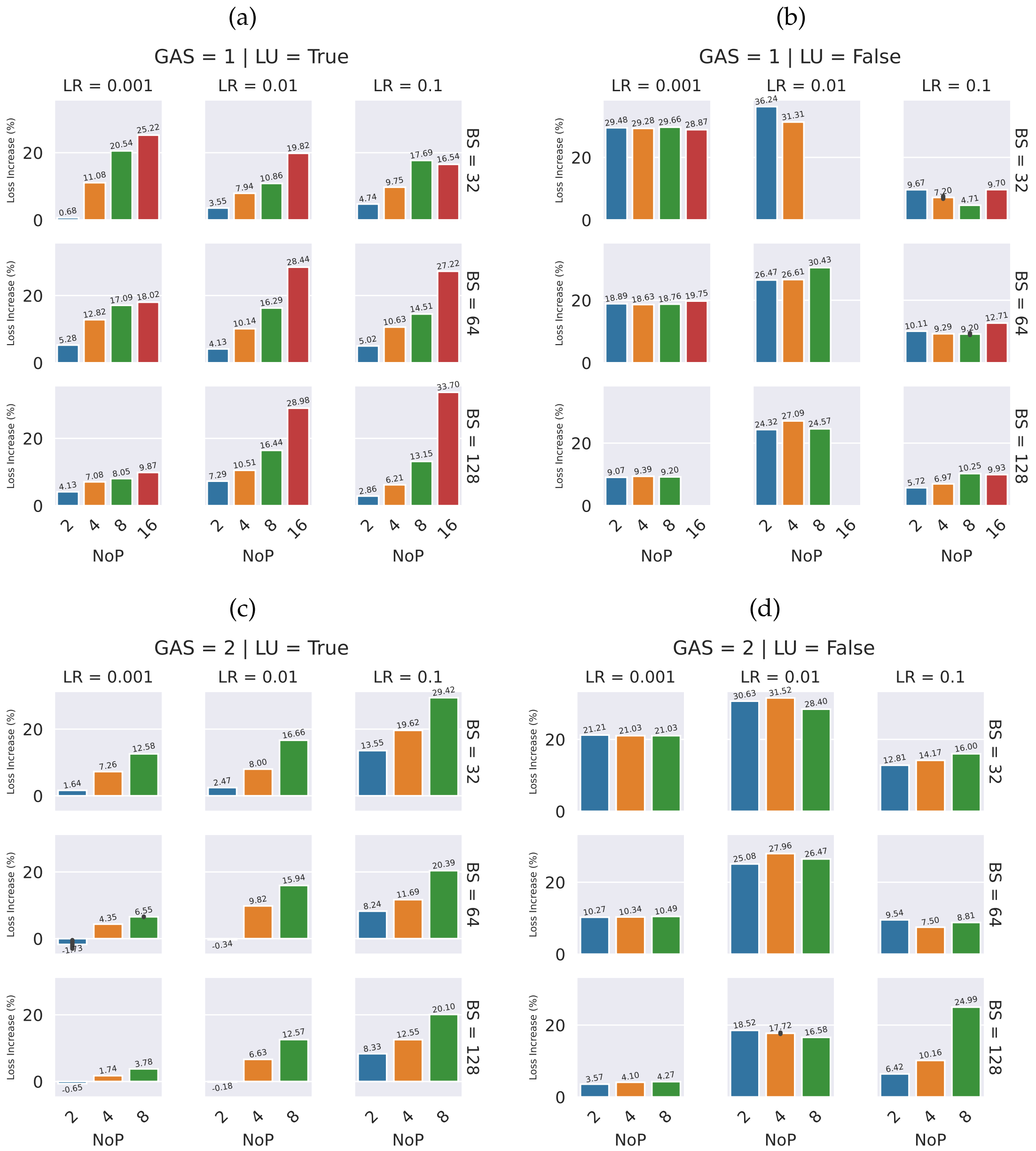


Figure 6.10: Loss increase in percent for Hivemind runs with different number of peers and vCPUs relative to baseline runs. Higher is worse.

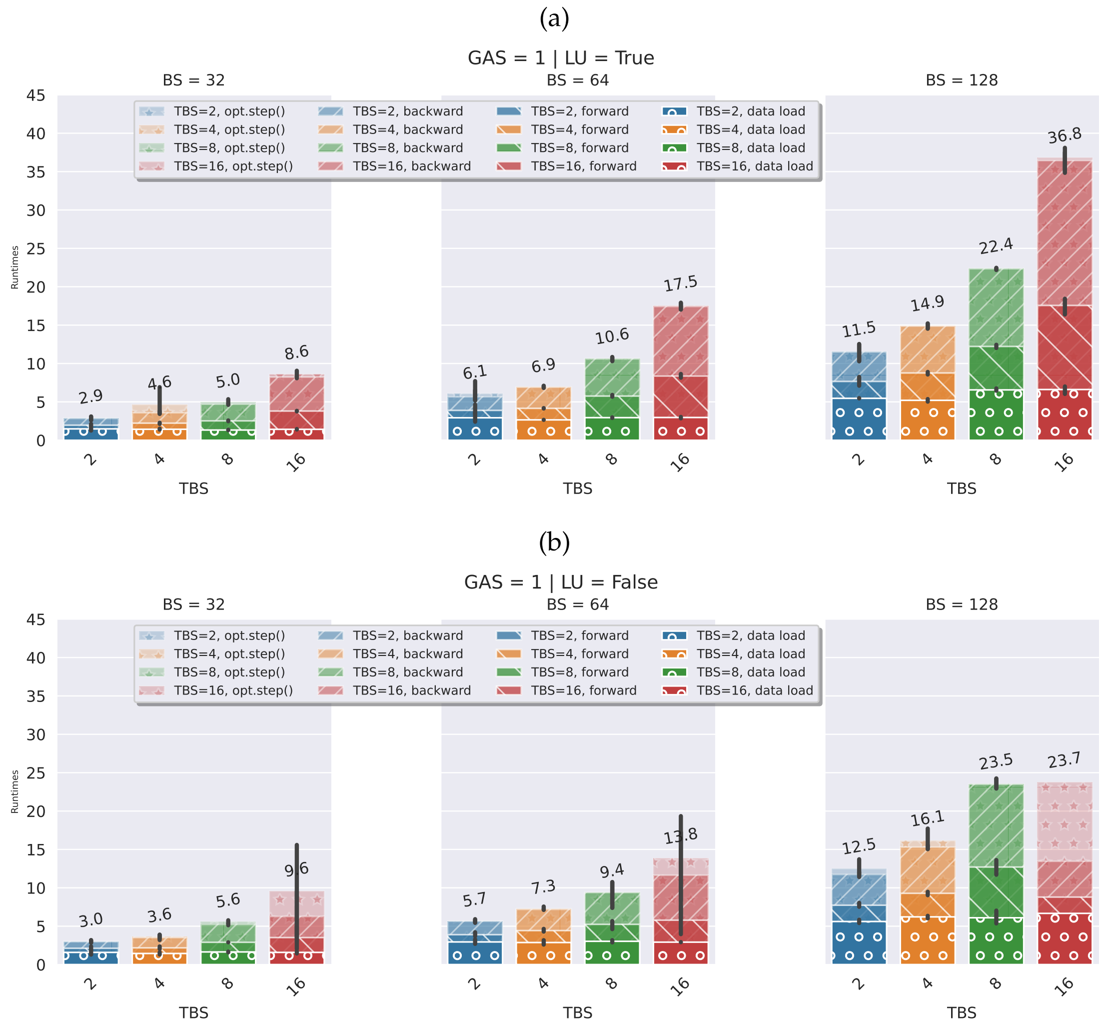
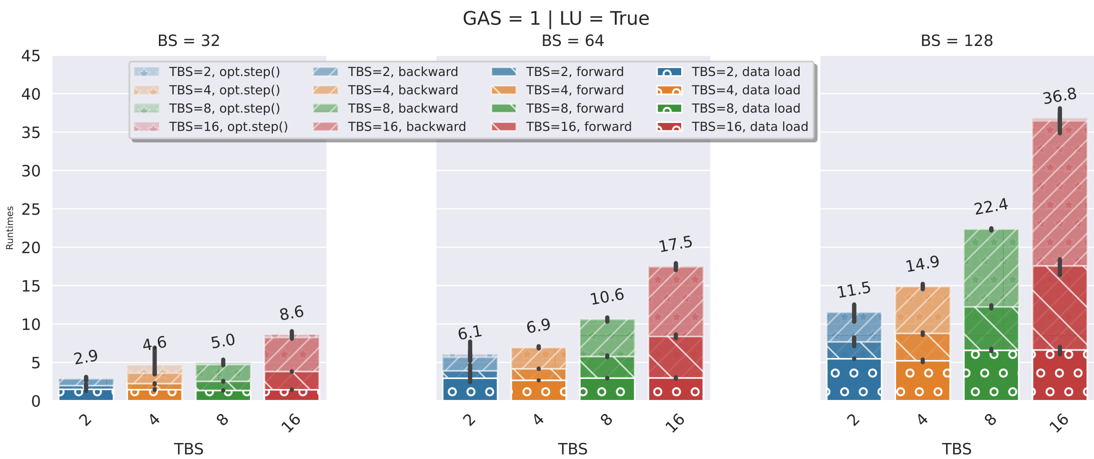


Figure 6.11: Average times of data load (small circles), forward pass (backward slash), backward pass (forward slash) and optimization step (stars) for Hivemind experiments with different number of peers in seconds. Runs are further aggregated across LR and the standard error amongst runs is shown with black bars (continues).

(c)



(d)

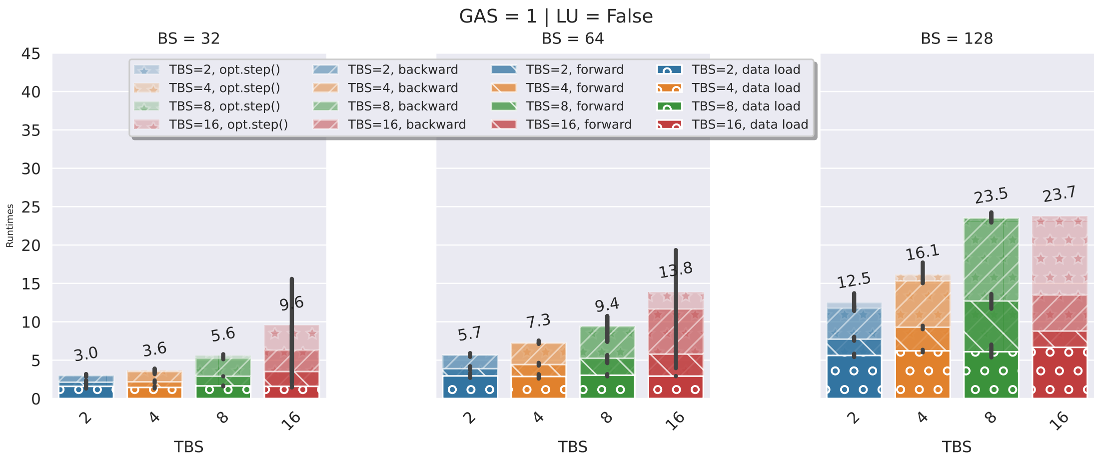


Figure 6.6: Average times of data load (small circles), forward pass (backward slash), backward pass (forward slash) and optimization step (stars) for Hivemind experiments with different number of peers in seconds. Runs are further aggregated across LR and the standard error amongst runs is shown with black bars (continues).

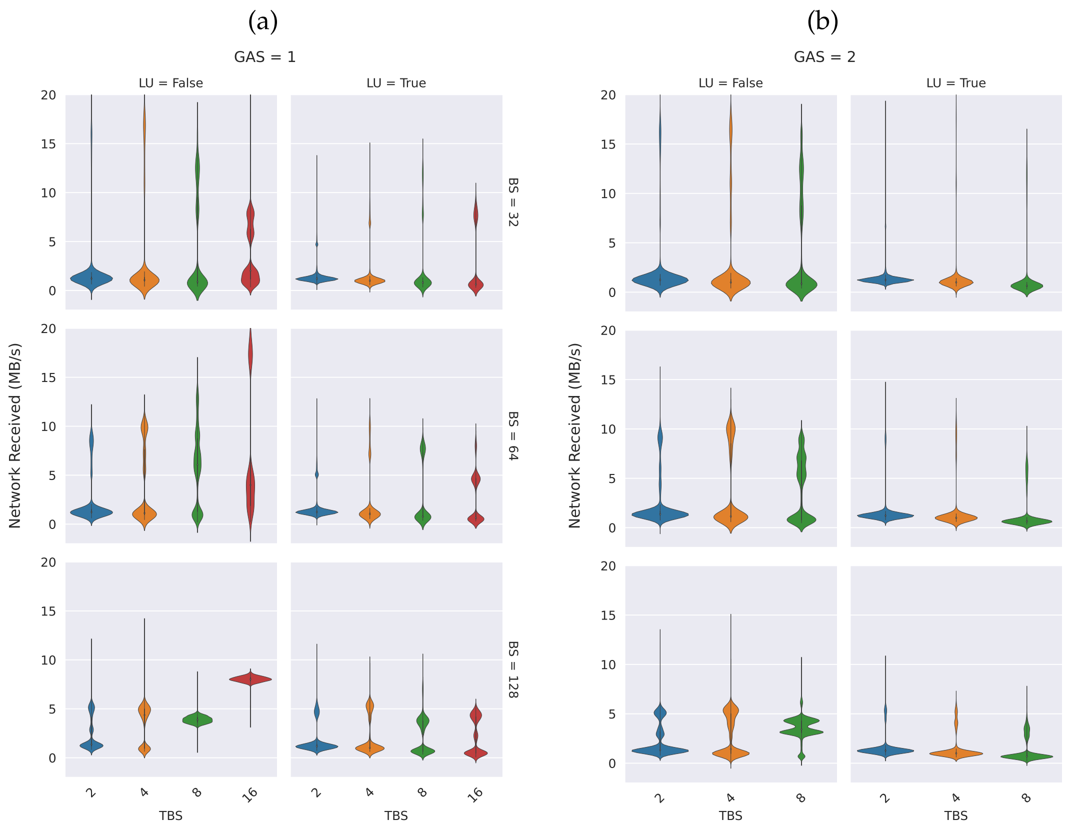


Figure 6.12: Network received for Hivemind runs with different number of peers and vCPUs. Values $\geq$ 20 MB/s are hidden and runs are aggregated across LR.

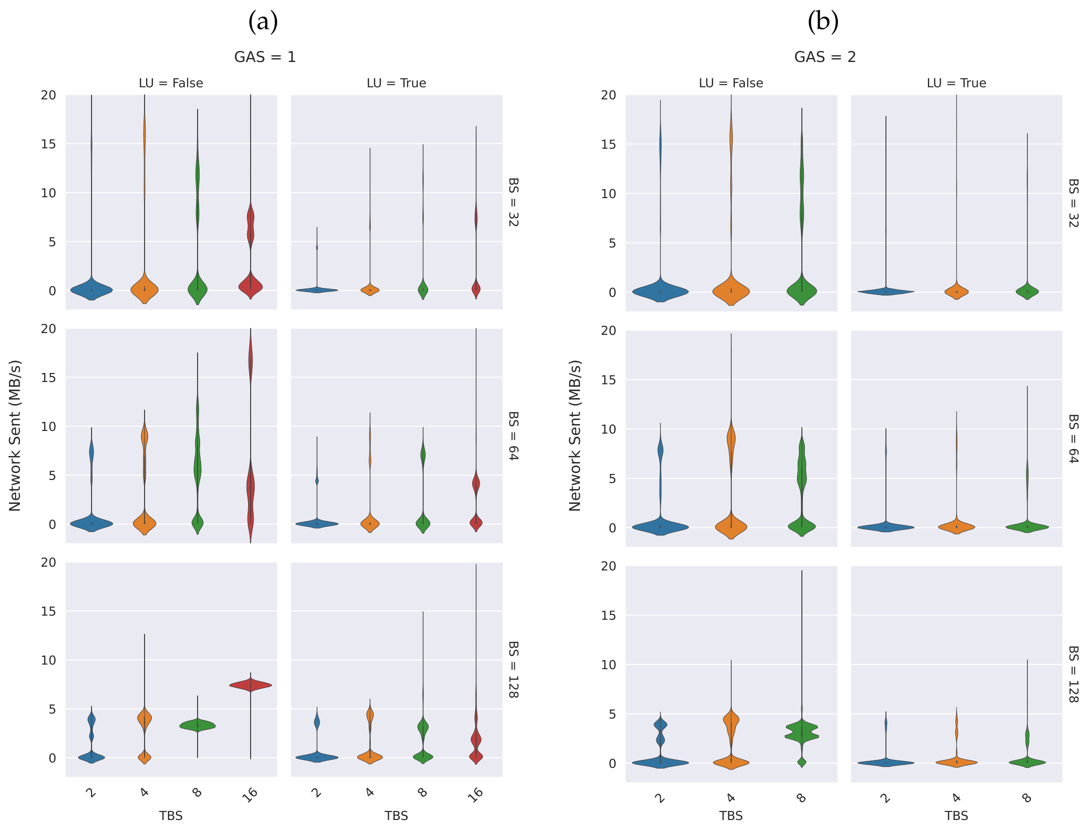


Figure 6.13: Network sent for Hivemind runs with different number of peers and vCPUs. Values ≥ 20 MB/s are hidden and runs are aggregated across LR.

7 Future Work

8 Conclusions

List of Figures

List of Tables

Bibliography

[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.

[Den+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A large-scale hierarchical image database." In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: `10.1109/CVPR.2009.5206848`.

[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`.

[Goy+17] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." In: *CoRR* abs/1706.02677 (2017). arXiv: `1706.02677`.

[He+15] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015). arXiv: `1512.03385`.

[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.

[Kes+16] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." In: *CoRR* abs/1609.04836 (2016). arXiv: 1609.04836.

[Kri14] A. Krizhevsky. "One weird trick for parallelizing convolutional neural networks." In: *CoRR* abs/1404.5997 (2014). arXiv: 1404.5997.

[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.

[LTJ20] N. Lee, P. H. S. Torr, and M. Jaggi. "Data Parallelism in Training Sparse Neural Networks." In: *CoRR* abs/2003.11316 (2020). arXiv: 2003.11316.

[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.

[RG] M. Ryabinin and A. Gusev. *learning@home*. https://learning-at-home.github.io/.

[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`.

[YGG17a] Y. You, I. Gitman, and B. Ginsburg. "Scaling SGD Batch Size to 32K for ImageNet Training." In: *CoRR* abs/1708.03888 (2017). arXiv: `1708.03888`.

[YGG17b] Y. You, I. Gitman, and B. Ginsburg. "Scaling SGD Batch Size to 32K for ImageNet Training." In: *CoRR* abs/1708.03888 (2017). arXiv: `1708.03888`.