Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Distributed Training of Transformer Neural Networks

Carlos Adrian Salas Cedillo

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Distributed Training of Transformer Neural Networks

Author:	Carlos Adrian Salas Cedillo
Examiner:	Dr. Felix Dietrich
Submission Date:	June 30th, 2023

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

June 30th, 2023 Carlos Adrian Salas Cedillo

Acknowledgments

I would like to thank Dr. Felix Dietrich for his support and constructive feedback throughout my thesis.

Special thanks to ITK Engineering for sponsoring my research. I want to thank Stefan Held for showing interest in working with me and helping me develop the thesis topic. I would also like to thank Jan Linus Steuler for guiding me in every step of the way and motivating me during the project's most challenging parts. I would especially like to thank Leonardo Muffato for acting as a mentor and friend during my time at the company.

I would also like to thank my close friends Ashwanth, Anirudh, Itzel, Nigel, Manoj, and Karen, who were always ready to help whenever needed. Finally, I would like to thank my partner Kateryna for her most loving support during this time.

Abstract

The Transformer is a Deep Learning model that uses a self-attention mechanism to keep track of global dependencies in large data sequences. These properties have led Transformers to be adopted as the backbone for modern foundation models. These are large-scale general models that can perform a wide range of tasks due to the large amounts of data that they are trained on. However, their many parameters represent a challenge to individuals without high-end specialized hardware. The training of such models can be enabled and sped up by applying distributed training techniques. In this thesis, synchronous Data Parallelism and asynchronous Data Parallelism were used to speed up the training of the EVA model, a state-of-the-art foundation model based on the Vision Transformer capable of performing several vision tasks. In particular, the EVA is used as a case study for object detection on low-resolution gray-scale images. The chosen distributed training techniques were implemented using Horovod, a distributed training framework based on MPI, to perform communication between nodes. The performance of the methods was evaluated using two machines, each with one GPU, connected over the organization's network using MPI. The lack of adequate HPC communication hardware resulted in a high variation of communication overhead. Under high traffic conditions, the communication overhead overshadowed the benefits of parallelization for the available number of workers. However, under suitable conditions, the parallelization resulted in a speedup of 1.6164 for synchronous Data Parallelism. At the same time, asynchronous Data Parallelism achieved a speedup ranging from 1.961 to 1.988 depending on the weight synchronization frequency every 2 to 64 iterations.

Contents

1. Introduction

The relevance of Deep Learning in recent years has been increasing. Namely, the field of Natural Language Processing (NLP) has hit some significant milestones with the surge of foundation models like BERT [6] and GPT-3 [2]. The field of Computer Vision has also benefited from the use of foundation models like Contrastive Language-Image Pre-Training (CLIP) [26] and GPT-4 [25]. What all these models have in common is the use of Transformers, a model architecture with a mechanism of self-attention that keeps track of data relations better than some older models based on recurrence.

The ground-breaking results of transformers have inspired researchers to develop much more transformer-based models over the last few years. Figure 1.1 shows some milestones in the development of transformers.

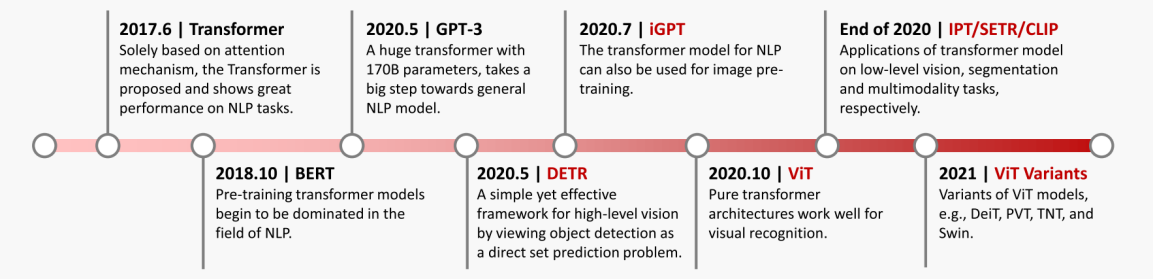

Figure 1.1.: Timeline of transformer models (image from [11]).

In computer vision, models that use transformers as backbones compete against other state-of-the-art Convolutional Neural Network (CNN) models. They implement a multi-head self-attention mechanism that keeps track of long-term dependencies. Vision Transformer (ViT) [8] had a fantastic success and set the path for researchers to build new general-use vision transformer backbones and architectures [21]. These advancements can be used in object detection, image classification, and semantic segmentation applications. An exciting example of transformer-based models is the EVA, which has a billion-scale [10] and a million-scale [9] model variants as visual representation foundation models. They both rank as State of the Art or top 10 for some of the leading computer vision benchmarks.

However good these models sound, their large size makes them hard for smaller organizations to compete against. Figure 1.2 provides a visual representation of the number of parameters that make up some of the machine learning models created in the last two decades (up until 2021) [28]. The number of parameters has grown exponentially, especially in the last five years. The use of Graphical processing unit (GPU)s to accelerate the

training of models is a standard in the field. However, unless a high-end GPU like the NVIDIA A100 or the AMD Radeon Instinct MI100 is used, the model will either not fit the memory of a single GPU or the training time will be extremely long.

This is why, even since the beginning of the 2010's decade, distributed training techniques have been adopted to enable and accelerate the training of such models [5, 4, 19]. Moreover, foundation models are trained using a huge amount of data to be able to use the model for a wide variety of applications. This leads to extremely long training times, even when using a large number of high-end GPUs.

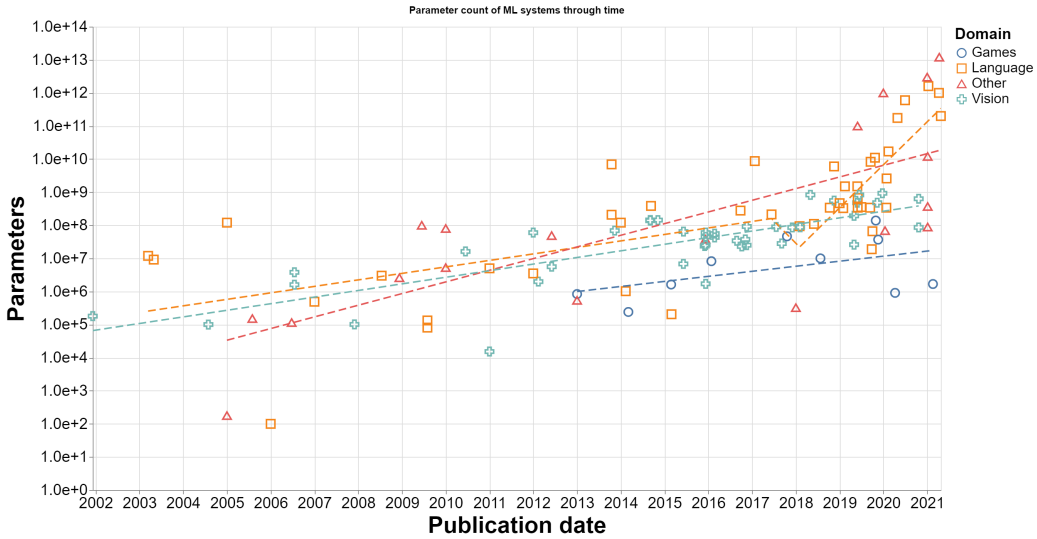

Figure 1.2.: Timeline of the number of parameters in machine learning models from 2002-2021 (image from [28]).

There is a large gap between the capability to train large models between big companies and smaller organizations. The CO_2 emissions generated by the training of foundation models is a big concern [31, 1]. Luckily, pre-trained foundation models can be used to perform different tasks with just some fine-tuning. This is a much more viable option for smaller organizations that want to use state-of-the-art models, making it feasible to apply transfer learning and train the model in a smaller-scale distributed system.

In an attempt to decentralize the use of large-scale models, this thesis explores the use of distributed training techniques applied to the particular case of the vision transformer and foundations models based on it.

2. State of the Art

2.1. Transformer Networks

Transformers were originally proposed for the field of NLP. Previous approaches that try to understand the dependencies in data sequences, like Long short-term memory (LSTM) and Recurrent Neural Network (RNN) have a sequential nature. Transformers eliminate this recurrence and use instead an attention mechanism that can learn the dependencies of the global context while allowing parallelization for more efficient processing [32].

2.1.1. Self-attention

Attention is a function that maps an input vector, transformed into query (**Q**) and key (**K**) value (**V**) vector pairs, to an output which measures the compatibility to each value given its corresponding query and key. The resulting attention represents the focus that these vectors will receive later on. Attention is calculated as a weighted sum of the values, whose weights are computed as a probability function $P = \text{softmax}(S_n)$, where S_n is the normalized score between the input vectors as the dot product of **Q** and **K**. The attention function is simplified as

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \tag{2.1}$$

where d_k is the dimension of queries and keys. This is known as scaled dot-product attention, which can be visualized on the left side of Figure 2.1. This mechanism, however, limits the ability to consider information from different positions at the same time. To improve this, multiple attention heads can be incorporated in a single Multi-Head Attention layer, as seen on the right side of Figure 2.1. In this layer, the queries, keys, and values are linearly projected h times into $\mathbf{Q'} = \{\mathbf{Q}_i\}_{i=1}^h$, $\mathbf{K'} = \{\mathbf{K}_i\}_{i=1}^h$ and $\mathbf{V'} = \{\mathbf{V}_i\}_{i=1}^h$. The attention function is computed for each of these projections in parallel, and they are then concatenated and linearly projected once more. This process is summarized as

$$\begin{aligned} \text{MultiHead}(\mathbf{Q'}, \mathbf{K'}, \mathbf{V'}) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)\mathbf{W}^O, \\ \text{where head}_i &= \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}), \end{aligned} \tag{2.2}$$

and $\mathbf{W}^O$ is the projection's weight of dimensions $d_{\text{model}} \times d_{\text{model}}$.

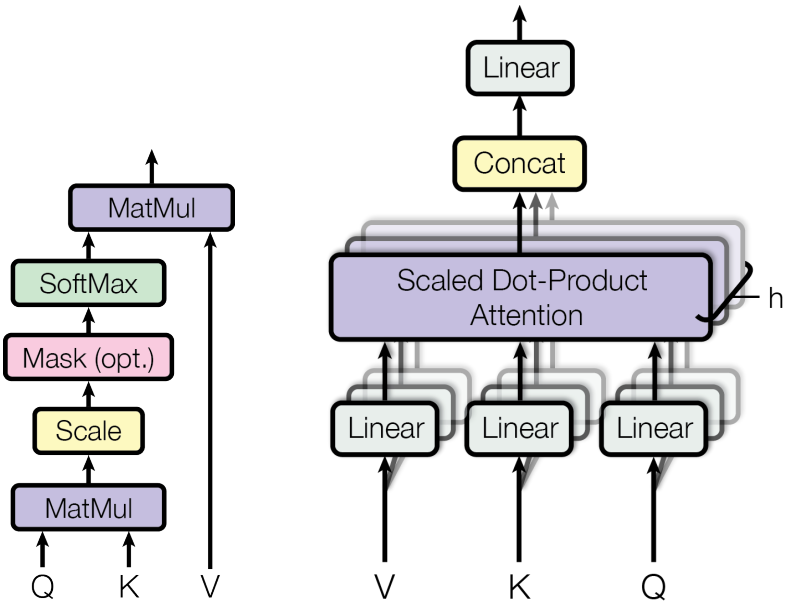

Figure 2.1.: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel (image from [32]).

The original transformer uses these attention layers in an encoder-decoder structure, as seen in Figure 2.2. The encoder is made up of N layers consisting of a multi-head attention mechanism and a fully connected feed-forward network. Each of the sub-layers has a residual connection with its original input. The decoder layer has an extra multi-head attention sub-layer that takes as input matrices $\mathbf{K}$ and $\mathbf{V}$ from the output of the encoder layer, as well as $\mathbf{Q}$ from the initial multi-head attention sub-layer. Since the self-attention mechanism does not have any positional information about the data, the architecture used a positional encoding stage at the beginning of the form

$$\begin{aligned} \mathrm{PE}_{(\mathrm{pos},2i)} &= \sin(\mathrm{pos}/1000^{2i/d_{\mathrm{model}}}), \\ \mathrm{PE}_{(\mathrm{pos},2i+1)} &= \cos(\mathrm{pos}/1000^{2i/d_{\mathrm{model}}}). \end{aligned} \tag{2.3}$$

2.1.2. Vision Transformers

The success of transformers also permeated the field of computer vision, where the strong representation capabilities of the transformer are useful for a set of tasks. In this case, most transformers use the original encoder as a feature extractor that captures long-term global dependencies. Here, vision transformer backbones have performed similarly or better than other networks like CNNs or RNNs [11]. While transformers are used alongside popular CNN architectures in some cases, there are some pure-transformer examples.

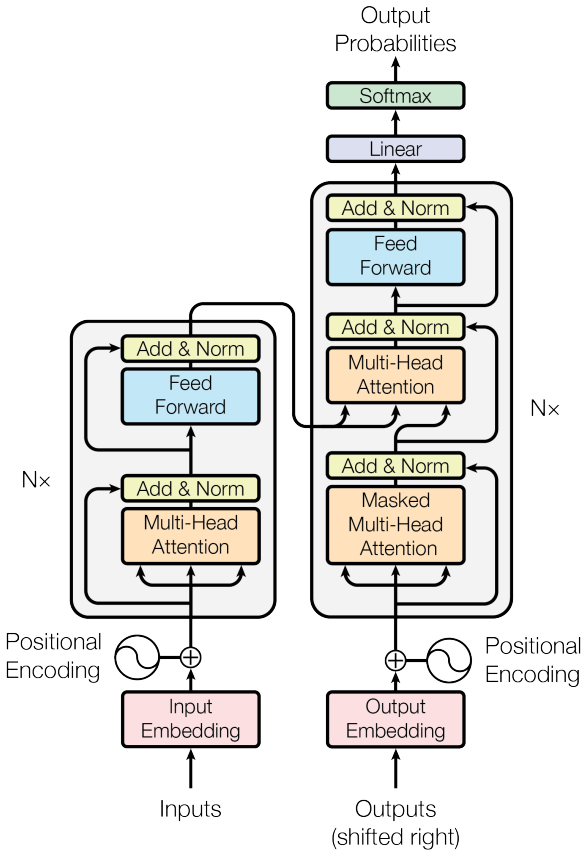

Figure 2.2.: Original transformer architecture (image from [32]).

Such is the case of the Vision Transformer (ViT) [8]. Figure 2.3 shows an overview of the model. It uses a very similar structure to the original transformer, using visual tokens as inputs taken from patches of size (P, P) of an image of size (H, W) and C channels. The 2D image $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$ is reshaped into a sequence of flattened patches $\mathbf{x}_p \in \mathbb{R}^{N \times (P^2 C)}$, where $N = HW/P^2$ is the total number of patches. A learnable linear embedding to class tokens $\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$ is included, as well as some positional embedding to the surrounding patches that can be used to represent the connection to a different part of the image depending on the given task. A Multi-Layer Perceptron (MLP) block, used as the classification head, is attached to the linear embedding at the output of the encoder $\mathbf{z}_L^0$.

While the results of the ViT are excellent, the global attention mechanism can be extremely expensive for high-resolution images. For this reason, researchers have come up with some alternatives like the SWIN [21], and more recently, the CWIN [7]. These models rely on window-based multi-head self-attention, where the attention is restricted to patches inside a given window, after which the window is shifted to be able to learn information about the global context. This reduces the complexity of the attention mechanism

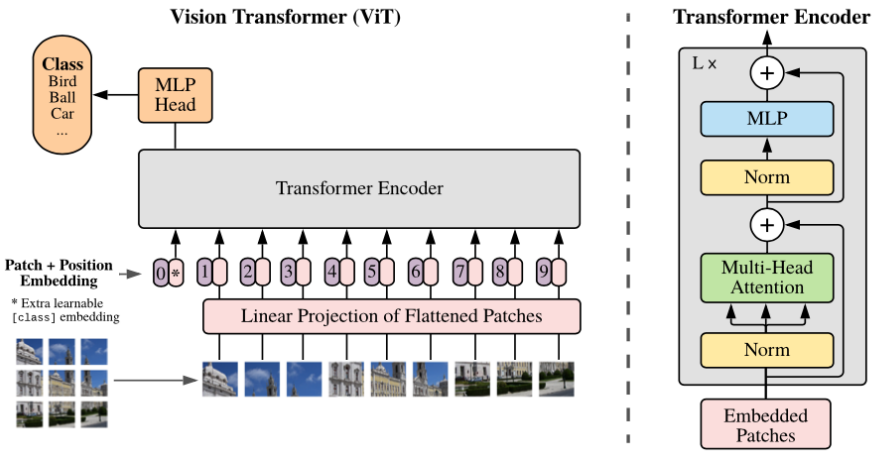

Figure 2.3.: ViT architecture (image from [8]).

Table 2.1.: Architecture details of the EVA model

patch size	#layers	hidden dim	MLP dim	attn heads	#params
14×14	40	1408	6144	16	1011M

from quadratic complexity to near-linear[1].

An advantage of pure transformer-based architectures over CNN models is that they can take any images of any resolution as input, without extra padding or augmentations. However, they lack the inductive bias that CNNs have and must therefore be trained on larger datasets to obtain comparable results. This is why some approaches use both CNN and transformer layers in conjunction.

2.1.3. EVA

EVA [10] is a large-scale open-source foundation model based on the ViT. It is pre-trained on public data to align image-text data, which can be used for vision tasks like object detection, image segmentation, and video action classification, as well as scaling up other models like CLIP and outperforming some state-of-the-art results. It uses Masked image modeling (MIM) without relying on supervised learning. The architecture of the model can be seen in Table 2.1.

For the case of object detection, EVA uses Cascade Mask R-CNN to perform the initial detection. The original hyper-parameters used for training can be found in appendix A.

[1]To better understand this process, it is recommend to follow this very well-made visual guide: https://towardsdatascience.com/a-comprehensive-guide-to-swin-transformer-64965f89d14c

The implementation details are available on the author's repository[2].

2.2. High Performance Computing and MPI

Message Passing Interface (MPI) [23] is the industry standard specification when it comes to parallel computing on distributed memory systems. It is used to perform message passing, point-to-point communication, and collective communication operations. It is implemented as a library, which has the advantage of the implementation not being bound to the programming language. This means that MPI can be used as a general tool and can have new features added to it without compiler modifications [17].

MPI is mainly intended for distributed-memory systems and networks, although shared-memory systems are not excluded. It provides features to do message-passing between nodes. Nodes are independent units that do not share computational and memory resources between them. Each node contains ranks, which are individual processes. Typically, ranks are bound to a Central processing unit (CPU) inside the node and can be further parallelized among the CPU's threads. Together, all ranks make up the `MPI_COMM_WORLD`. Figure 2.4 shows an overview of these concepts.

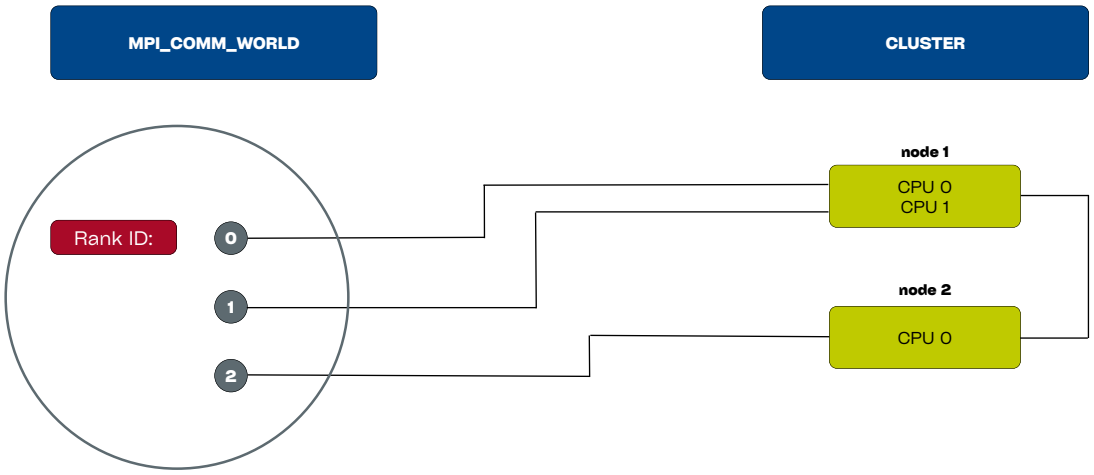

Figure 2.4.: Visual representation of the elements that make up the `MPI_COMM_WORLD`.

The communication can be done point-to-point and can be blocking or non-blocking.

[2] https://github.com/baaivision/EVA/tree/master/EVA-01/det

This type of communication is performed with *send* and *receive* operations. These operations are done pairwise, using the functions `MPI_SEND()` and `MPI_RECV()`, where the send buffer, message size, data type, and destination/origin ranks must be specified.

MPI also includes collective operations, where the communication is done for groups of processes rather than pairwise, as in point-to-point communication. All the basic operations can be seen in Table 2.2.

Table 2.2.: MPI basic collective communication operations

Collective operation	**Description**
`MPI_BARRIER()`	Synchronization barrier.
`MPI_BCAST()`	Broadcast identical copies of data from one to all members of the group.
`MPI_GATHER()`	Gather data from all members to one member of the group.
`MPI_SCATTER()`	Scatter parts of the data of one member to all members of the group.
`MPI_ALLGATHER()`	Gather data from all members on all members of the group.
`MPI_ALLTOALL()`	Scatter and gather data from all to all members of the group.
`MPI_ALLREDUCE()`	Reduce operation (sum, max, min, etc.) of the data of all members. All members receive a copy of the reduced data.
`MPI_REDUCE_SCATTER_BLOCK()`	Combined reduce and scatter.
`MPI_SCAN()`	Scan across all members of a group.

2.3. Distributed Training Techniques

2.3.1. Data Parallelism

Data Parallelism (DP) [5, 18] is a distributed training technique that splits the training data across multiple workers, each with an exact model copy. Each worker processes a smaller batch of the data to calculate the model's gradients, which are then broadcasted and aggregated across the rest of the nodes. Figure 2.5 (left) depicts the DP training process.

Its use is straightforward and usually does not require changes in the model, which is why DP has become the default distributed training technique used by most deep learning frameworks. Typically DP performs synchronous gradient synchronization. This is done by doing a complete forward pass and backward propagation of the mini-batch to compute the model's gradients, which are then synchronized with the rest of the workers using an *allreduce* operation. This brings the possibility of scaling the training using a large number of nodes. Synchronous DP has the advantage of updating weights using the entire data set. However, it performs excessive synchronization, and if (a) the model is too big and (b) the number of nodes is not enough to overcome the communication overhead, DP can lead to longer training times. As a general rule, models with high computation per weight benefit the most from DP [18].

It is possible to mitigate this overhead using asynchronous DP approaches such as Stochastic Gradient Descent (SGD) with delayed updates [30] and Asynchronous Dual Averaging [24]. The choice of asynchronous techniques needs to be done carefully since they can affect the convergence of the optimization process.

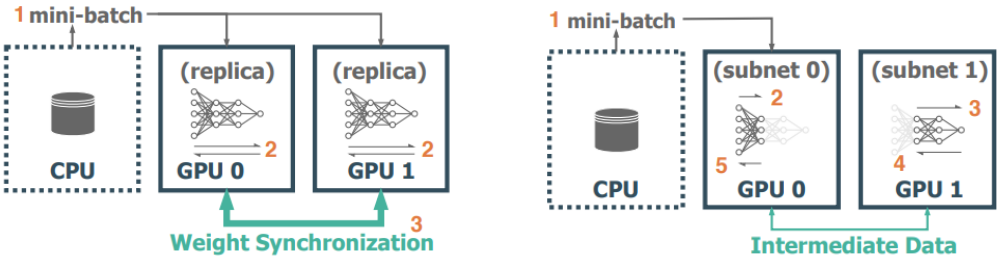

Figure 2.5.: Representation of Data Parallelism (left) and Model Parallelism (right) in a setup consisting of one CPU and two GPU nodes (image from [3]).

Optimizers

Optimizers are a crucial part of the training of neural networks. They are used to iteratively find the (minimizing) solution of a loss function $l(X, Y, \mathbf{W})$ for N data points. The optimization problem in DP is very similar to common SGD scenarios. This is convenient for quick implementation and is therefore solved using some optimizer of the SGD family.

SGD consists in splitting the data into mini-batches of size N/num_batches and solving a mini-batch loss function $L(x, y, \mathbf{W})$ computed with the mini-batches portion of the data. The mini-batches are fed in a round-robin manner until all batches have been used, having completed an epoch. A gold standard family of optimizers is the Adam optimizers, a variation of SGD that performs updates by adapting the learning rate with estimators of first and second moment of the gradients [16].

In particular, AdamW [22] decouples the weight decay from the gradient update in Adam. L2 regularization is typically used as weight decay, but in the case of Adam, this does not hold and leads to bad convergence. Instead, the moving averages do not include the weight decay term, and it is added later on the parameter update as

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \left(\alpha \hat{\mathbf{m}}_t / (\sqrt{\hat{\nu}_t} + \epsilon) + \boxed{w \mathbf{x}_{t-1}} \right), \tag{2.4}$$

where $\mathbf{x}$ denotes the parameter vector, t the time step, η the learning rate multiplier, α the learning rate, and $\hat{\mathbf{m}}$ and $\hat{\nu}$ the first and second moment vectors. Here, the term in the box is the weight decay.

In the case of DP with delayed weight updates, the convergence of SGD optimizers is affected since each worker usually updates its local weights with different sets of data. Overlap Local-SGD [33] is an algorithm proposed to mitigate this effect by adding an anchor model $\mathbf{z}$ to each node that performs the communication every τ local updates, while the local model $\mathbf{x}^{(i)}$, where $i \in \{1, ..., m\}$ for m nodes, continues to perform its local updates. The anchor model saves the average model across all workers and is then used to pull the local model toward the global average as

$$\mathbf{x}_{k+1}^{(i)} = \begin{cases} \mathbf{x}_{k+\frac{1}{2}}^{(i)} - \alpha \left(\mathbf{x}_{k+\frac{1}{2}}^{(i)} - \mathbf{z}_k \right) & (k+1) \text{mod} \tau = 0 \\ \mathbf{x}_{k+\frac{1}{2}}^{(i)} & \text{otherwise,} \end{cases} \tag{2.5}$$

where $\mathbf{x}_{k+\frac{1}{2}}^{(i)}$ is computed by the local optimizer's (SGD, Adam) update rule, and α is a tunable parameter. The anchor model computes the global model average as

$$\mathbf{z}_{k+1} = \begin{cases} \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}_{k+1}^{(i)} & (k+1) \text{mod} \tau = 0 \\ \mathbf{z}_k & \text{otherwise.} \end{cases} \tag{2.6}$$

This reduces the communication latency significantly compared to synchronous DP while improving the convergence of other asynchronous DP optimizers. This optimizer is particularly useful for arrangements with small communication bandwidth and large models like transformers.

Higher-order optimizers are available, such as Newton, which contain information about the function's curvature and converge faster. The downside is that they rely on the computation of the Hessian, which is highly costly in the context of deep learning. A sophisticated approach uses a quasi-Newton matrix-free conjugate gradient method of second

order [27]. The original authors of this method propose defining a cheap Hessian as a vector product using the Pearlmutter approach. After this, a Newton step is applied using CG-iterations that use matrix products only. The resulting method has $\mathcal{O}(bn)$ complexity for the evaluation of the gradients, where n is the number of weights and b the size of the mini-batch, as well as $\mathcal{O}(2mbn)$ for the computation of the Hessian and the solution of the Newton equation, where m is the number of CG-iterations.

2.3.2. Model Parallelism

Model Parallelism (MP) originated back when GPUs did not have enough memory to store medium to large-sized models. Compared to DP, it requires a higher level of understanding of the model's architecture. But if done right, it can lead to great results, as was the case of the training of the highly influential AlexNet [19]. Some early work proposed distributing different parts of the model across a set of machines to distribute the computational workload of training [5]. This was done purely in CPU nodes, but since then, MP has been done using GPUs instead.

Model parallelism is a distributed training technique in which the model is split across the available GPUs. The splitting can be done into subnets containing part of the layers of the original model as seen in Figure 2.5 (right), or splitting a layer's neurons as seen in the horizontal direction of Figure 2.6. Each GPU will do the updates for its portion of the layers. However, since the nature of the forward and backward pass is sequential, the data must be pipelined in mini-batches to avoid idle workers [3, 13].

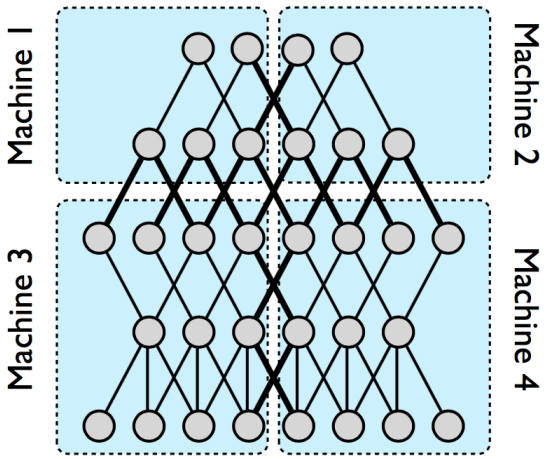

Figure 2.6.: Model Parallelism with different splitting directions. The thick connection lines represent the nodes that need to be synchronized (image from [5]).

On the downside, naive pipelining like PipeDream [12] has the problem of weight staleness. Imagine a setup of two GPUs `gpu0` and `gpu1` containing two parts of a model, where

the output of each group of layers is $\mathbf{F}_0 = f_0(x)$ and $\mathbf{F}_1 = f_1(x)$ respectively. A mini-batch $\mathbf{x}^i$ is fed through $\mathbf{F}_0^i = f_0(\mathbf{x}^i)$ and then forwarded to $\mathbf{F}_1^i = f_1(\mathbf{F}_0^i)$. At the second state, `gpu1` starts to compute $\mathbf{F}_1^i$, while `gpu0` will compute $\mathbf{F}_0^{i+1}$ using the following mini-batch $\mathbf{x}^{i+1}$. This is a problem because, in SGD, $\mathbf{F}_0^{i+1}$ should be computed after the layer's weights have been updated using $\nabla\mathbf{F}_0^i$. By the time this weight update happens, $\mathbf{x}^{i+1}$ and $\mathbf{x}^{i+2}$ will already be down the pipeline. This is a problem that drags and scales when using more GPUs, which can lead to instability in the optimization process and affect convergence.

Some approaches have been proposed to solve this problem. One straightforward option is to split the mini-batches into micro-batches and only do the weight update once all micro-batches have been processed [15]. However, this gravely affects the pipeline's throughput. Another approach [3] involves some predicting weights to avoid staleness by using Momentum SGD with smoothed gradient. This model achieves similar throughput as PipeDream while maintaining the convergence of DP.

As mentioned, MP is not straightforward and requires deep knowledge of the model architecture. A few interesting solutions have been proposed for the specific case of transformers. One of them, the PipeTransformer [14], uses elastic pipelining by doing on-the-fly layer freezing during training and combines this with data parallelism. Another interesting method, Oases [20], uses automated tensor model parallelism to schedule the model updates cleverly. Both methods achieve more remarkable speedups than DP and naive MP while preserving the convergence of the model.

3. Thesis Development

3.1. Methodology

3.1.1. Research Design

This thesis evaluates the possibility of enabling the training of foundation models based on the transformer architecture on lower capacity hardware that small-to-midsize organizations or institutions might have available and achieving a speed-up on the training time. For this, both synchronous data parallelism and asynchronous data parallelism have been chosen to assess the training performance of the model. In the end, the results of the experiments will be used to define a methodology that other artificial intelligence developers can use to improve the training performance of their own models. Since the main goal is to provide developers with available tools to speed up the training of their own models, model parallelism was not implemented since it is particular to the model used.

The model chosen as a case study is the EVA model for object detection. The size of the model allows a wide variety of scenarios one can face while trying to train large models. On the one hand, its large size already represents a challenge since the model, especially its gradients, might not fit in the memory of a single GPU. On the other hand, if the available hardware cannot fit large batch sizes, the training time on a dataset of significant size will be infeasible.

These challenges were approached by performing transfer learning on a new dataset of smaller size and different characteristics using the model's pre-trained weights, training just a few last layers of the model for several epochs, and evaluating its training using

1. a sequential approach,
2. data parallelism,
3. data parallelism with delayed weight updates.

3.1.2. Data Collection

The dataset used is a portion of the Objects365 dataset [29], resized and color space converted to emulate grayscale security camera footage of low resolution (320x198). Refer to section 3.2.1 to see the detailed image preprocessing steps. The goal is to use the EVA model to detect particular objects of interest on security cameras. The categories of interest are: Chair, Bicycle, Luggage, Stroller, and Wheelchair. Figure 3.1 gives an overview of the

images used in the dataset. It is important to note that EVA expects annotation data with bounding boxes and masks. Since the original Objects365 dataset does not contain mask data, weak masks[1] were added as a workaround since the evaluated task is object detection and not segmentation. This could still impact the model's overall accuracy because both tasks are embedded in the same architecture. Still, since accuracy is not the main focus of this thesis, this is overlooked.

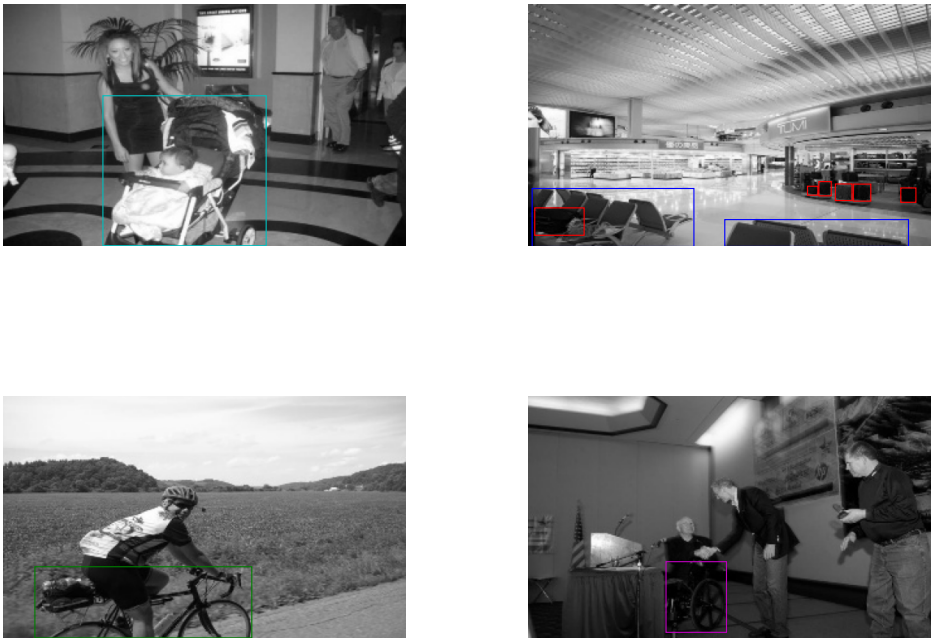

Figure 3.1.: Sample images of the used dataset, with annotations of categories: Chair, Bicycle, Luggage, Stroller, Wheelchair.

3.1.3. Experimental Setup

The experiments are performed on a GPU cluster consisting of two independent nodes, each with ten Intel(R) Xeon(R) Gold 6334 CPUs, 128GB RAM, and one NVIDIA A40 GPU with 24GB memory. Both nodes have a 10G Ethernet and 16G FC connection. The communication between nodes is implemented with OpenMPI through network access with Secure Shell Protocol (SSH) via Ethernet. Figure 3.2 shows a visual representation of the

[1] Masks of the exact shape of the bounding box

arrangement. The detailed setup steps are described in section 3.2.2. Compared to an High performance computing (HPC) configuration consisting of Non-uniform memory access (NUMA) nodes, our setup does not share a common memory pool between nodes or allow direct access to each other's memory. The network connection between nodes introduces additional latency and communication overhead.

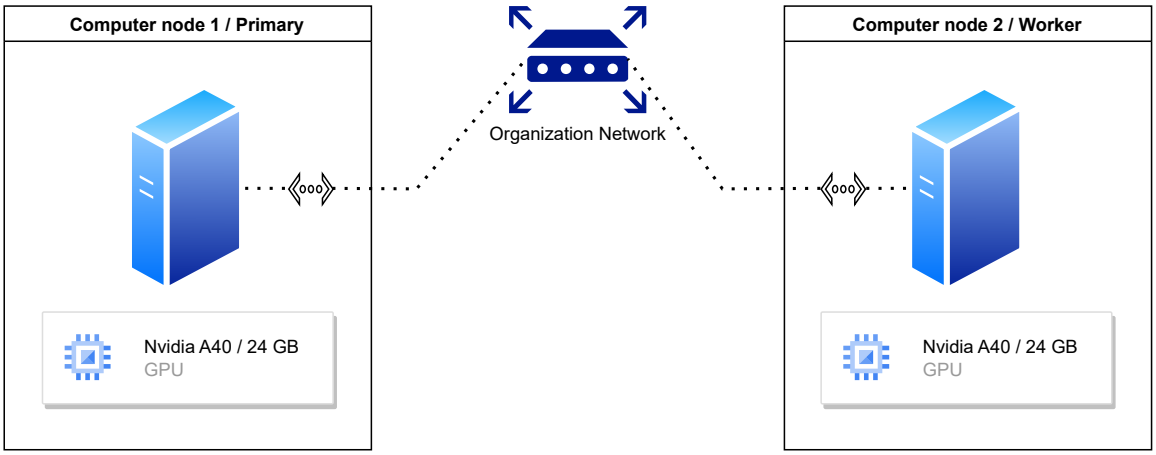

Figure 3.2.: Topology representation of the GPU cluster used.

Horovod was chosen as the framework to implement the distributed training. It provides MPI support as a backend. Compared to PyTorch's distributed implementations, it uses more efficient communication protocols, such as the NVIDIA Collective Communication Library (NCCL) for NVIDIA GPUs, to speed up the communication between different nodes during distributed training. This can lead to faster training times and better scalability on large clusters. Besides all this, Horovod is framework agnostic and can work with different DL frameworks, giving future developers more versatility.

At the same time, the detection application of EVA is built upon Facebook's Detectron2. Detectron2 is a popular open-source computer vision library developed by Facebook AI Research (FAIR). It is written in Python and is built on top of PyTorch. Detectron2 provides a flexible and modular framework for training and deploying state-of-the-art object detection, instance segmentation, and keypoint detection models. It is usually as simple as creating a configuration file with all of the model's parameters and details, which is then invoked by the training script. There are two standard training script formats: Lazyconfig and plain training. The configuration file's format depends on the training script. While the plain training script uses dictionaries stored in.YAML files, lazyconfig training, uses lazyconfig dynamic dictionaries stored in .py files. These dynamic dictionaries can store complex objects instead of raw text, which must be instantiated later by the training script.

However, even if the lazyconfig training script is a more practical option, it uses many built-in functions to perform the training process. Implementing a different distributed backend would mean modifying the source code, which could lead to higher implemen-

tation complexity and unexpected problems. Instead, the plain training script executes a straightforward training loop which is more friendly for hackers. For this reason, the plain training implementation was chosen while modifying it to support layzconfig files since it is the configuration file that the original authors provided.

3.1.4. Training Procedure

The model is used strictly as provided by the original authors. While there probably is some fine tuning that can (and should) be done to improve the model's accuracy to the new dataset, this is outside the scope of this thesis, so it was decided not to do anything with it. Only some slight configuration modifications were done to use the custom dataset. To see all changes, please refer to section 3.3.

All but the last 10% of the model's layers were frozen. This number was chosen arbitrarily and is a hyper-parameter that should be tuned in the future if the accuracy wants to be improved. The model uses AdamW as an optimization algorithm, with an adaptive learning rate of $2.5\mathrm{e}{-5}$. The dataset is fed through batch sizes of 8 per GPU, the maximum batch size that the GPU can fit in its memory with this given dataset. The model is trained for a total of five epochs. This is mainly done to perform some uncertainty quantification in the training time of a whole epoch.

3.1.5. Evaluation Metrics

Three main metrics were observed during the experiments:

- Elapsed time per epoch
- Training loss
- Communication overhead

It is important to note that the primary metric is the time per epoch. However, observing the training loss helps us monitor if the model's convergence is preserved while applying a given distributed training technique.

3.2. Preparation

3.2.1. Building the Dataset

It was intended to use the EVA model for object detection using security camera footage to detect objects of the categories Chair, Bicycle, Luggage, Stroller, and Wheelchair. The camera produces grayscale images of resolution (320x198). The Objects365 dataset was chosen to emulate this scenario. Since the model was pre-trained on the same dataset, part of the learned characteristics learned by the model should be preserved even if the images are modified.

The original dataset contains 2 million RBG images with 30 million bounding boxes of 365 categories. The whole dataset is not needed since only transfer learning is done. A custom dataset was built from the first ten patches of the Objects365 training dataset, containing 380 thousand images. The original metadata file includes the image's metadata and the annotations for the whole training set. The first step is to filter out all information of images not present in the portion of the dataset used.

Once this was done, all the annotations not part of the interest categories were removed. At this point, all images containing no annotations from the newly filtered ones were also removed. The next step is to resize the images to the desired resolution and turn them into grayscale. For simplicity, the images were all resized to the exact resolution, regardless of their original proportions. All images with a height larger than the width were removed to avoid harmful distortions that could affect the model's performance. At the same time, the scaling factor for the width and the height was stored in the image's metadata so that it could be used to scale the annotations' bounding boxes at a later point. The before and after of how these images look can be seen in Figure 3.3. In the same figure, it can be seen that there is a bounding box around a person. This is because the category "Person" was initially also of interest but was removed later.

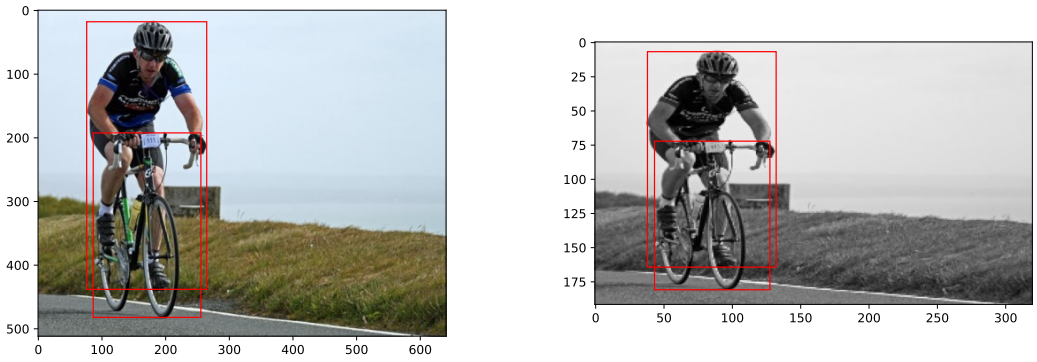

Figure 3.3.: Image with annotations before and after preprocessing.

As mentioned, the category ”Person” was also of interest originally. However, after comparing the distribution of categories from the filtered annotations, the distribution is highly skewed towards the categories ”Person” and ”Chair.” This unbalance can lead to the model being biased to learn just for these two categories while completely ignoring the rest of the categories. Since the frequency of ”Person” is far greater than the rest of the categories, it was removed entirely to avoid biases. The rest of the annotations were also balanced by limiting the number of images where only annotations of the type ”Chair” are present. As seen in Figure 3.4, the distribution after balancing the annotations is much more suitable for training.

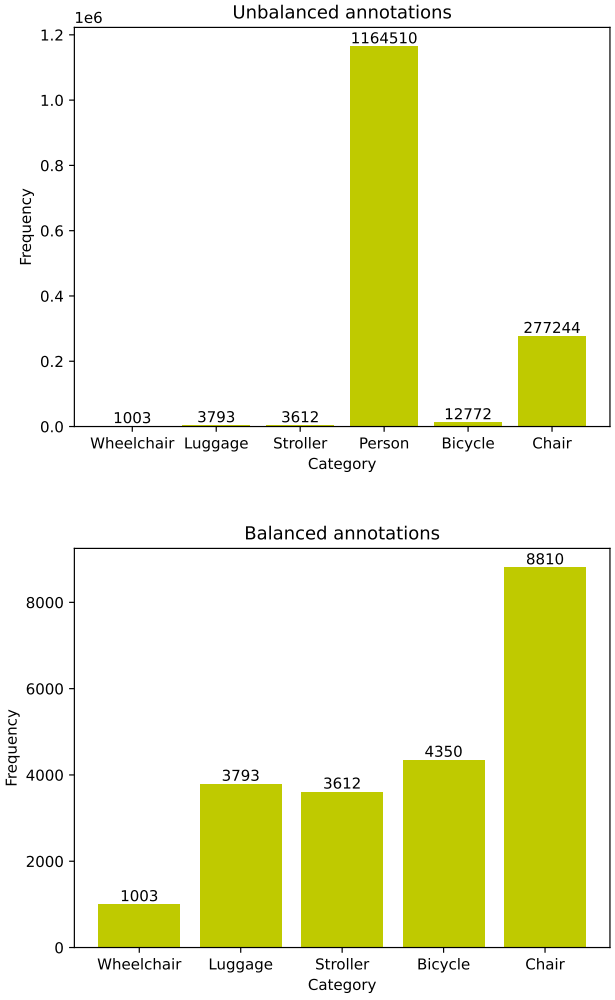

Figure 3.4.: Comparison of categories distribution present in the dataset’s annotations before and after balancing.

After all preprocessing, the resulting training dataset consists of 6711 images and a validation and test dataset of $\sim$ 890 images each. The final distribution of the categories in each dataset can be seen in Appendix B.

3.2.2. Setting up the cluster

The distributed training is performed using Horovod for Pytorch. Horovod supports a wide range of backends to perform communication and GPU operations. Before installing Horovod, the backends must be installed. Connecting two nodes and performing collective operations directly in the GPU is desired. MPI was chosen as a backend as the controller and NCCL[2] for tensor operations.

The GPU computers used do not have any specialized hardware to communicate with each other; the only way to do it is through an SSH connection. OpenMPI was chosen as the MPI implementation, but this could also be done with other open-source or proprietary MPI implementations. First of all, OpenMPI needs to be installed in each node. Before running MPI commands, SSH access between nodes must be granted without a password. Please refer to OpenMPI's documentation about this process[3]. Once this is done, MPI jobs can be run as

```
1 mpirun -np 2 --host <ip_node_1>:1,<ip_node_2>:1 <parallel script>
```

To avoid typing the IP addresses and number of processes per node every time, a host file that includes all of the nodes' information can be created. All that is left now is to install NCCL. The NCCL version must be chosen accordingly to the Compute unified device architecture (CUDA) version available in the system.

Finally, Horovod can be installed. This can be done with `pip install horovod` after specifying the backends with the flags

- `HOROVOD_GPU_OPERATIONS=NCCL`
- `HOROVOD_WITH_PYTORCH=1`
- `HOROVOD_WITH_MPI=1`
- (optional) `HOROVOD_NCCL_HOME=<NCCL_home_directory>`

The last flag is optional and must only be included if NCCL was installed in a different location than the default one. If everything is done correctly, Horovod can be used for distributed training as

```
1 horovodrun -np 2 -H server1:1,server2:1 python <train_script>.py
```

[2]https://developer.nvidia.com/nccl
[3]https://www.open-mpi.org/faq/?category=rsh

3.3. Sequential baseline

In this section, the process of adapting the training script and model configurations to be able to train on our custom dataset is described. After this, the model is trained on the dataset to obtain a baseline to compare the distributed training.

3.3.1. Training script

When using custom datasets with Detectron2, the first thing that needs to be done is to provide a function to read the dataset, returning it in a specific format. Detectron2 already provides many built-in functions to work with COCO-style data, so the data loading function was built to return the dataset in such a format. Additionally, the same function is in charge of creating the weak masks out of the bounding box shapes to be able to feed them to the Mask R-CNN detection layers. This is done as shown in Source Code 3.1. The returned data is a list of dictionaries of each image, with its annotations in a list inside the image dictionary.

```
def load_objects_camera(which_dataset):
        dataset_dir = os.environ.get('DATASET_DIR')
        filename = dataset_dir + which_dataset + ".json"

        with open(filename, "r") as f:
                data = json.load(f)
        for image in data:
                for anno in image['annotations']:
                        x,y,w,h = anno['bbox']
                        anno['segmentation'] = [[x,y, x+w,y,
                                                 x,y+h, x+w,y+h]]
                        anno['area'] = w*h
        return data
```

Source Code 3.1.: Custom data loading function.

To use the dataset, it needs to be registered to the framework's DatasetCatalog first with its name and loading function. At this point, the categories and the evaluator dataset type must also be set in the MetadataCatalog. Since the loading function returns the dataset in COCO format, the predefined COCO evaluator can also be used. This is all done in the main function shown in Source Code 3.2.

```
categories = ["Chair", "Bicycle", "Luggage",
        "Stroller", "Wheelchair"]

for d in ["train", "test"]:
    DatasetCatalog.register(d,
                            lambda d=d: load_objects_camera(d))
    MetadataCatalog.get(d).set(thing_classes=categories)
    MetadataCatalog.get(d).set(evaluator_type="coco")
```

Source Code 3.2.: Lines of coded included in `main()` to enable the use of the custom dataset.

Finally, the original plain training script is meant to get its configurations from a .YAML configuration dictionary. However, the authors of EVA only provide a .py dynamic configuration dictionary meant to be a lazy configuration file. The lazy config dictionary has a different structure than the regular one. Therefore, special attention must be taken to all lines of the code that invoke an element from the config file. In particular, dynamic elements must be instantiated with Detectron2's `instantiate()` function. The `setup()` function must also be modified to resemble the one in the lazyconfig_train_net.py script.

The epoch time was measured explicitly in the training loop. This is done using Python's time() module at the beginning and at the end of each epoch. Source Code 9.6 shows an overview of how the time is measured in the actual code.

```python
import time

epoch = 0
epoch_iteration = -1

st = time.time()
for data, iteration in zip(data_loader, range(start_iter, max_iter)):
    epoch_iteration += 1
    ... # Training
    if ((epoch_iteration+1) % iterations_per_epoch==0 or
        iteration == max_iter-1):
        et = time.time()
        epoch_time = et - st

        st = et
        epoch += 1
        epoch_iteration = -1
```

Source Code 3.3.: Training loop with epoch timing functions.

3.3.2. Model configurations

This thesis focuses on the parallelization results rather than the model's accuracy. Because of this, only a few model configurations need to be modified. Table 3.1 shows the complete list of modified parameters. Other parameters would make sense to change, but at this point, the only goal is to make the model able to take the custom data set as input.

Table 3.1.: List of modified configuration parameters used in the model for the custom dataset.

parameter	value	explanation
dataloader		
image_size	320	Used to perform automatic data augmentations in relation to the image size
train.dataset.names	"objects_camera_train"	Specify name of the train dataset
test.dataset.names	"objects_camera_test"	Specify name of the test dataset
evaluator	COCOEvaluator(...)	Specify COCOEvaluator as the evaluator type for the test dataset. The dataset name and output directory must be given as an input
train.total_batch_size	8	Batch size per GPU
train		
num_epochs	5	Number of epochs to train for
max_iter	num_epochs *6711/total_batch_size	Total number of iterations for the total amount of epochs. 6711 is the dataset size
lr_multiplier		
scheduler.milestones[0]	train.max_iter * .89	First milestone set at 89% of the training process
warmup_length	.0112/45000*train.max_iter	Scaled to the size of the current max_iter from the original max_iter
model		
backbone.square_pad	320	Square padding to match the size of the dataset
roi_heads.num_classes	5	Specify the number of classes to detect

3.3.3. Training

The training was done for five epochs and was only done on the last four layers of the model. The training process was repeated several times to rule out variation. Figure 3.5 displays the training loss evolution for the five epochs. It can be appreciated that the training loss stagnates around 1.14. It is hard to say if this means convergence since no attention is paid to the prediction's precision. Also, it is unknown if the loss will continue decreasing if the model is trained for more epochs.

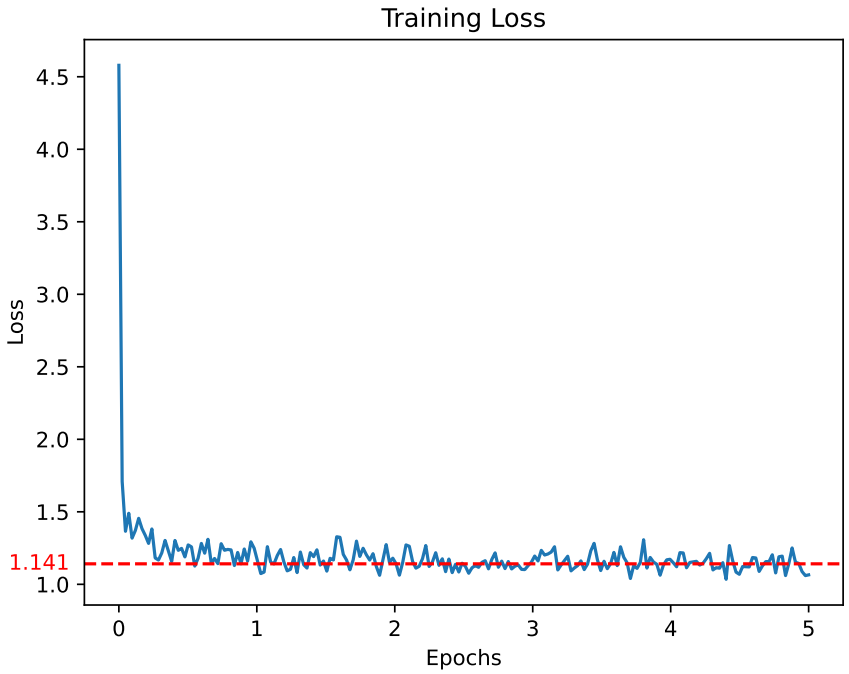

Figure 3.5.: Training loss for sequential training during five epochs.

The resulting average epoch time is of 15.98 minutes.

3.3.4. Short note on accuracy improvements

This section presents the results of some modifications that improved the ability of the model to make accurate predictions on the custom dataset. This is only meant as additional information for future work and it was not used in the next parts of this thesis.

As discussed in Section 3.3.2, the dataloader performs augmentations in relation to the image size of 320. However, the model is pre-trained on images of higher resolution and uses patch sizes relative to this sizes.

One experiment that achieved good accuracy results was doing data augmentations to an image size of 1333. Training on the augmented data, the model showed good signs of generalization even when training on a small number of epochs, as seen in Figure 3.6. Additionally, the obtained average precision at Intersection over Union 0.50 (AP50) of the model's prediction was 68.190, compared to 3.522 of the non-tuned model. On the downside, the increase of the image size reduced the possible batch size per GPU to 1 and drastically increased the time per iteration.

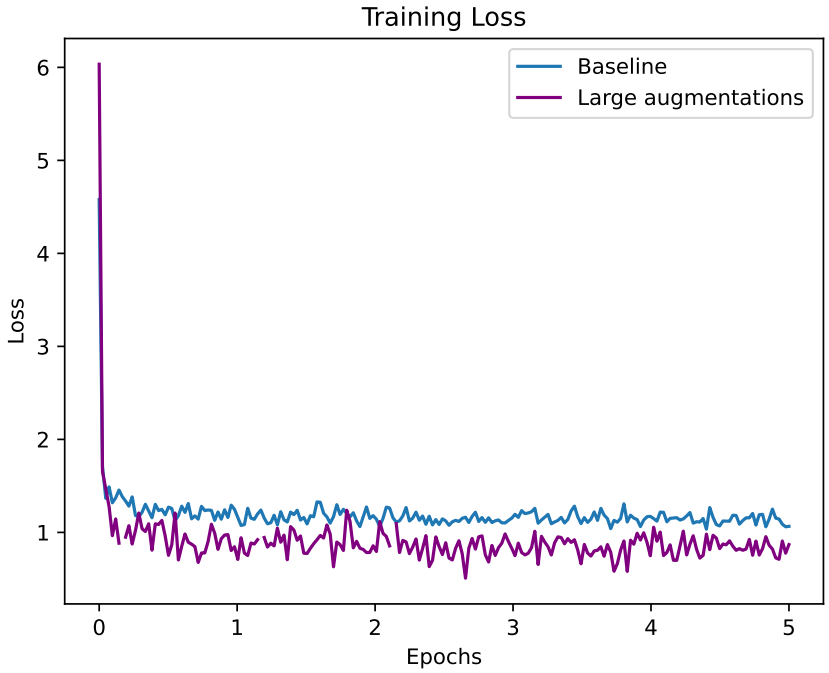

Figure 3.6.: Training loss comparison of the baseline used and an additional experiment using larger data augmentations.

However, better results could be obtained by modifying some backbone parameters to match the actual resolution of the images, such as the patch size, window dimension, embedded dimension and MLP ratio. The choice of which layers to train will also play an important role in the model's accuracy. This is all fine-tuning that is left for future work to other engineers if using the model for a similar application.

3.4. Data Parallelism

This section implements synchronous DP to compare its performance against the original sequential version. As many frameworks do, Detectron2 has distributed training functionalities built with Pytorch. As of the time this thesis was written, there is no way to choose a different backend, which would involve changing the source code, which is not advisable. Instead, some of these functions are replaced with Horovod functions or self-made ones built with Horovod commands. First, it needs to be defined how the synchronization will be done. This can be done manually with collective operations, but since a Pytorch optimizer (AdamW) is used initially, it makes more sense to wrap it with Horovod's DistributedOptimizer(). This new optimizer will synchronize the gradient once the step() method is called. The configurations are not configured directly; thus, some parameters must be updated for the number of workers. Since each GPU has a batch size of 8, the model will train on a super-batch of size 16 per iteration when using two nodes. Therefore the max number of iterations must be scaled by the number of workers as `max_iter / horovod.size()`. At the same time, the learning rate is scaled to the number of workers to make up for the change in batch size. Before the training loop begins, the parameters must be broadcasted to all workers to ensure consistent initialization. This is done with Horovod'S broadcast_parameters() and broadcast_optimizer_state() functions. Finally, when doing intermediate testing, a synchronization operation is included so that all workers continue the training simultaneously. Horovod does not have a barrier operation by default, but it can be easily implemented with a collective dummy operation, as seen in Source Code 3.4.

```
1 def hvd_barrier():
2     barrier_tensor = torch.tensor(0.0)
3     barrier_tensor = barrier_tensor.cuda()
4     barrier_op = hvd.allreduce(barrier_tensor, average=False)
5
6     barrier_op.item()
```

Source Code 3.4.: Barrier implemented with Horovod using an *allreduce* operation.

One last detail is that if checkpoints are used, all the checkpointing and writing operations must be performed by rank 0 to avoid file corruption. The overview of the parallel training function's most important details can be seen in Source Code 3.5. The resulting training script can be used for any number of nodes and could even be used by different Detectron2 models.

```
def do_train(cfg, model, resume=False):

    model.train()

    optimizer = hvd.DistributedOptimizer(...)

    ... # Other parameters initialization

    hvd.broadcast_parameters(model.state_dict(), root_rank=0)
    hvd.broadcast_optimizer_state(optimizer, root_rank=0)

    for iteration in range(start_iter, max_iter):

        ... # Other training steps

        optimizer.step()

        if iteration==eval_iteration:
            do_test(cfg, model)
            hvd_barrier()

```

Source Code 3.5.: Main components of DP training function.

A first run was performed using the DP script for a single node to see if Horovod introduces some overhead. As seen in Figure 3.7, neither the execution time per epoch nor the training loss seems to be affected.

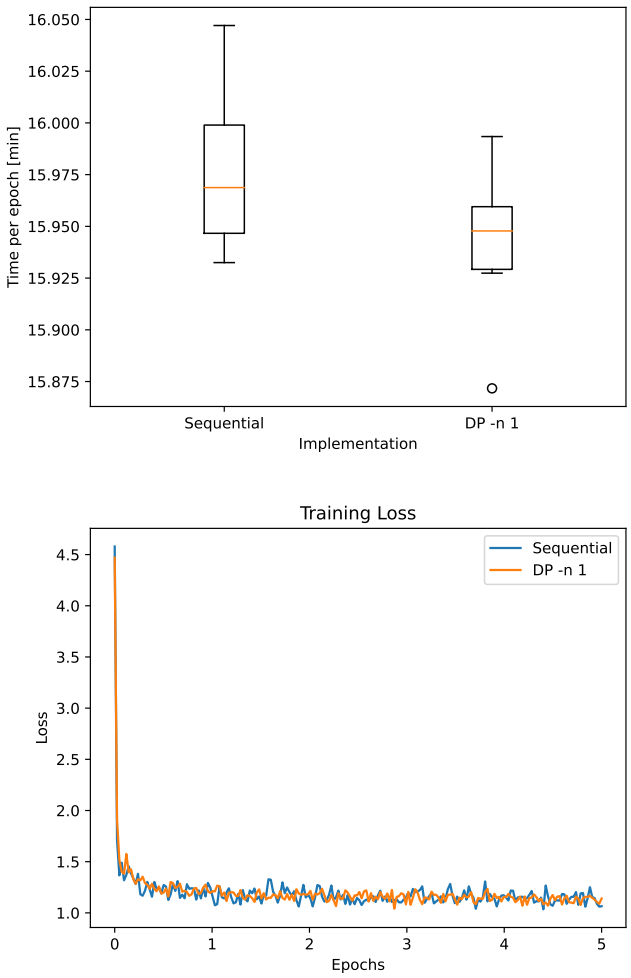

Figure 3.7.: Execution time and training loss comparison of baseline with a single node DP run.

Next, two cases are used to evaluate the effects of the parallelization. First, the training is done on a batch size of 8 per node (super-batch 16). This also means that the number of iterations per epoch will be less. The learning rate is increased to $5e{-}5$ since the size of the super-batch was doubled. The speedup of this experiment is 0.8053, a total of 19.84 minutes per epoch. This was expected since synchronous DP performs excessive synchronization. The communication overhead overshadows the speedup obtained by the parallelization. However, it is most likely that using a few more nodes will already show improvements. On the other hand, the training curve loss in Figure 3.8 clearly shows that the loss decreases faster than the sequential training. Even if the time per epoch is less,

convergence may be reached faster. This is likely caused by (a) the increase in the certainty since more data is shown in each SGD step and (b) the increase in the learning rate. At the same time, to see the effects of the communication overhead, one more experiment is done with a batch size of 4 (super-batch 8). The convergence of the optimization algorithm should be equivalent to the original sequential run, but this helps us to see the synchronization overhead. The resulting speedup is 0.6932, with an epoch execution time of 23.05 minutes.

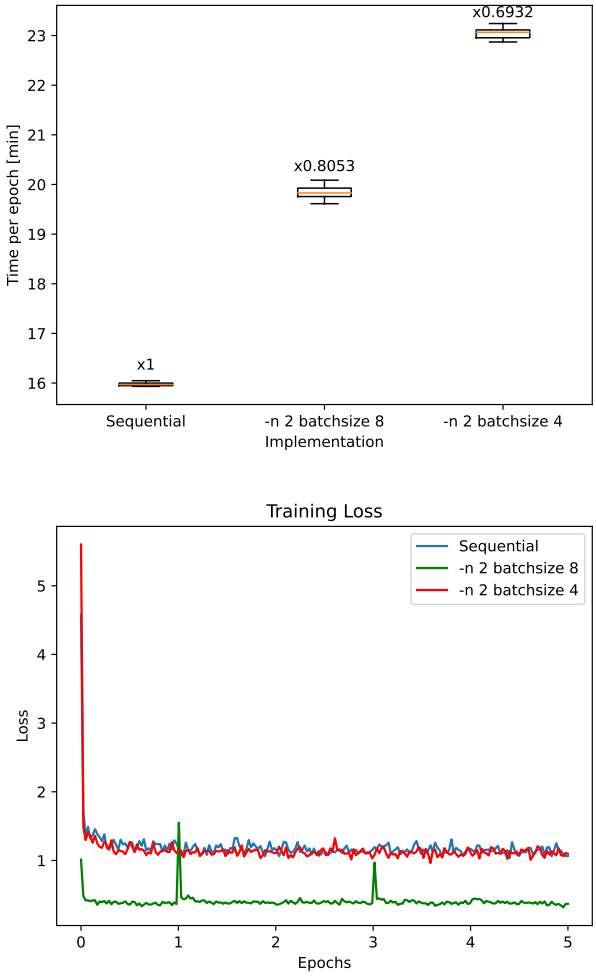

Figure 3.8.: Execution time and training loss comparison of baseline with DP run using two nodes and different bath sizes.

3.4.1. Analysis of the communication overhead

The parallelization results show that even with double the number of workers, the communication overhead is more than the speedup obtained. This was expected because the current setup used does not have the proper hardware to do low-latency communication. Therefore, some analysis of the communication overhead introduced by the parallelization was performed to determine the minimum number of nodes to observe some speedup.

Horovod has a built-in tool that records a timeline of the communication activity. This is all recorded for one node, but it also contains some valuable information to understand the behavior of the rest of the nodes. To use this, the argument `--timeline-filename` must be passed to the `horovodrun` command. This generates a trace file containing information on the communication activity on every tensor, as seen in Figure 3.9. This is mainly split into blocks of types:

- *negotiation* A phase where the worker notifies that it is ready to perform the collective operation.
- *processing* Collective operation phase (for example, *allreduce*).

At the same time, these blocks contain sub-blocks with more specific information about what is happening at a given point in time. To read more about this, please refer to Horovod's official documentation[4].

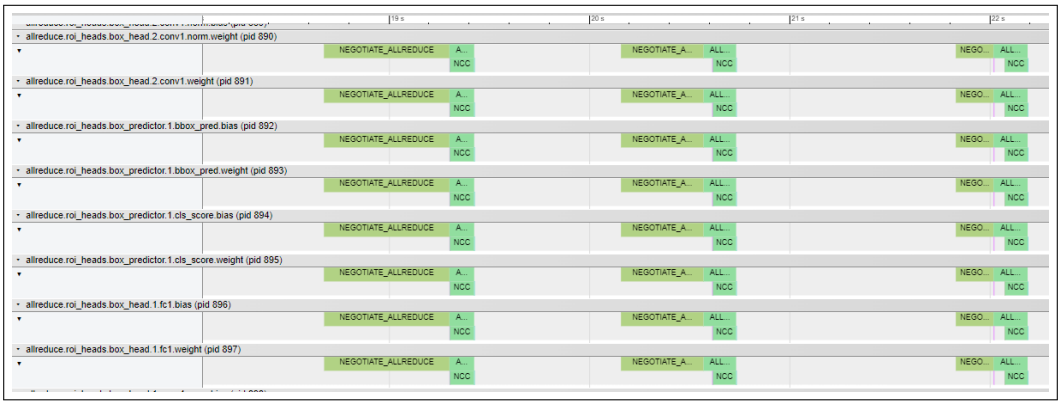

Figure 3.9.: Communication timeline example.

This information was used to estimate the total communication overhead. To do this, the timeline for a total of 20 iterations with the same setup as the DP experiment for a batch size of 8 per node was recorded. The information used comes from all tensor operations of the model, from the beginning of the first *negotation* phase to the end last *allreduce* block. The average processing time of the *negotiation* and *allreduce* is computed using this data.

[4]https://horovod.readthedocs.io/en/stable/timeline.html

Both times make up the whole communication overhead time, while the rest of the time in the time window is considered the actual training iteration time. The results can be seen in Figure 3.10.

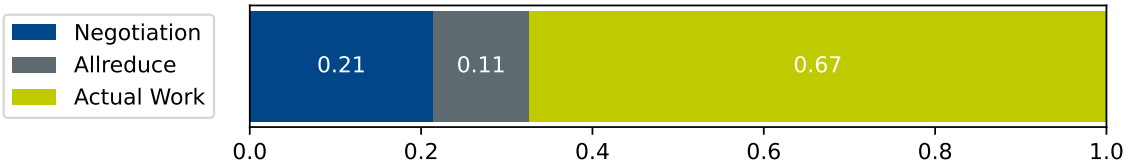

Figure 3.10.: Portion of execution time needed.

The estimated communication overhead makes up 33% of the total training time. When scaling to a more significant number of nodes under the same communication conditions, the data processed during the work phase will improve the speedup even further.

3.4.2. The importance of adequate communication hardware

An event in the companies' network infrastructure occurred where all connections had to be restored after a downtime. All experiments before the event showed a negative effect of parallelization because of the communication overhead. Still, after the system went back online, the results of the same experiments changed drastically. This new round of measurements resulted in an average epoch time of 9.88 minutes and a speedup of 1.6164, as seen in Figure 3.11. The unexpected event could have led to a decrease in the network's traffic, significantly decreasing the communication overhead introduced by the parallelization. In this instance, the speedup is better, but nothing guarantees that these good overhead conditions will persist. Rather than invalidating the previous experiments, this shows the high variance of communicating through an insecure network instead of using adequate communication hardware and protocols like Remote Direct Memory Access (RDMA) with Infiniband.

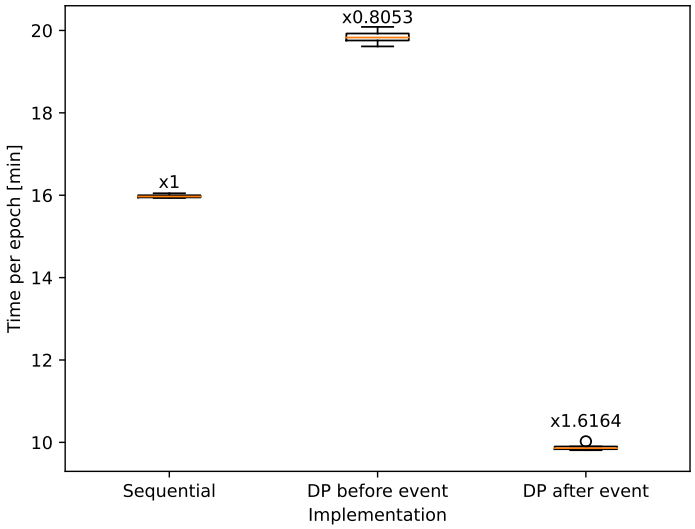

Figure 3.11.: Speedup comparison of DP before and after the network event.

The results of the remaining experiments are compared only against the results of the DP experiment after the event since comparing them with the results before the event would be unfair as the conditions were not the same.

3.5. DP with delayed weight updates

Asynchronous DP was also implemented to reduce the communication overhead caused by excessive collective operations. DP with delayed updates was chosen because of its implementation simplicity. No changes to the optimizer were made on this occasion since no measure of convergence other than the training loss was considered. The gradients are synchronized manually instead of letting the step() method handle it automatically. The synchronization is done every given number of iterations specified by the user. The weights are updated on every iteration using the step() method, setting the optimizer to skip the synchronization since this is done differently, as shown in Source Code 3.6. Additionally, the argument backward_passes_per_step must be set to the update frequency when building the DistributedOptimizer. Another meaningful change is the way the gradients of the model are set to zero on every iteration. The built-in method zero_grad() does this for the gradients of all ranks, which can cause race conditions if one rank does this before the other rank can update its weights. Instead, a new function async_zero_grad() is defined. This function allows each rank to zero only the local gradients. The process can be seen in Source Code 3.7.

```
if (epoch_iteration % synch_frequency==0 or
        (epoch_iteration+1) % iterations_per_epoch==0 or
        iteration == (max_iter-1)):
        optimizer.synchronize()

with optimizer.skip_synchronize():
        optimizer.step()

```

Source Code 3.6.: Asynchronous weight updates implementation.

```
def async_zero_grad(optimizer):
    for group in optimizer.param_groups:
        for param in group['params']:
            if param.grad is not None:
                param.grad.detach_()
                param.grad.zero_()
```

Source Code 3.7.: Function that allows asynchronous gradient zeroing on the local copy of the model.

The experiment is done using update frequencies ranging from 2 to 64. Figure 3.12 displays the experiment results. The speedup graph shows that the delayed weight updates achieve a more significant speedup than synchronous DP (update frequency of 1). However, the further improvement when decreasing the update frequency is not as significant as the change from update frequencies 1 to 2. At the same time, it is likely that this slight improvement significantly affects the convergence of the optimization process. Besides, the training loss curve is immediately affected when performing delayed weight updates compared to synchronous DP. While the effect on the training loss is apparent, this does not necessarily reflect the convergence since the model and hyper-parameters are not fine-tuned.

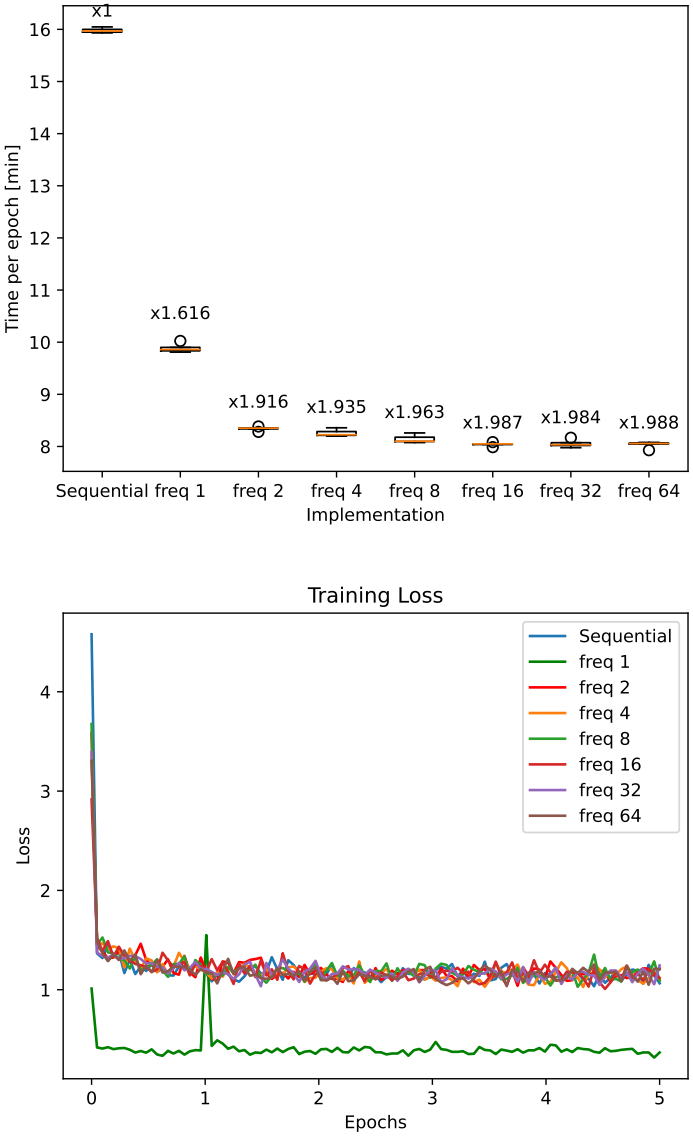

Figure 3.12.: Execution time and training loss comparison of DP with delayed updates using different update frequencies.

4. Conclusion

Distributed training was used to parallelize the training process of the last layers of the EVA foundation model, applied for object detection in low-resolution grayscale images. Both synchronous and asynchronous Data Parallelism were proven to be viable options to enable the training of large-scale models without extensive alterations to the model's structure and training scripts. Other developers can follow the work of this thesis to speed up the training of their own models. The chosen techniques were implemented and evaluated on a GPU cluster setup with two nodes connected over the organization's network.

4.1. Discussion

The results showed that the overhead communication could cause a slowdown rather than a speedup for the available nodes depending on the network's traffic. If distributed training is intended to be used as a standard technique to train large-scale deep learning models, it would be reasonable to make use of proper HPC communication protocols like RDMA, which allows direct access to a node's GPU. Under suitable conditions, the parallelization resulted in a positive speedup. Synchronous DP obtained a speedup of 1.6164 while also resulting in a higher convergence rate reflected in the training loss curve. Asynchronous DP was implemented using different weight synchronization frequencies ranging from every 2 to 64 iterations. It achieved speedups ranging from 1.961 to 1.988, while the training loss curves were similar to that of the baseline. Compared to synchronous DP, asynchronous DP achieved a noticeable speedup. However, further increasing the number of iterations without synchronization did not result in a significant speedup and could impact the convergence of the learning process. Overall, synchronous DP showed less speedup than asynchronous DP, but it showed better convergence since the SGD optimizer is not being modified. The speedup obtained by asynchronous DP was better than the synchronous implementation, but the training loss curve shows underperforming results. The choice of the weight synchronization frequency is a hyper-parameter that should be fine-tuned. To better evaluate the convergence of all techniques, they should be evaluated on a fine-tuned model for a complete training process instead of just a few epochs.

4.2. Outlook

The work of this thesis showcased how an organization can make use of distributed training techniques to enable the training of large-scale models. While the evaluated techniques

are good enough in most cases, there are some cases in which further improvements would be beneficial. Because of time constraints, the focus of this thesis was only limited to the parallelization of a model's training process without paying attention to hyperparameter tuning and its prediction accuracy. This also limits the ability to observe if the convergence of such techniques is consistent with sequential implementations. If the model were fine-tuned, some interesting experiments to explore further are discussed next.

- **Implementation of higher-order optimizers:** The work proposed in [27] presents a cheap second-order optimizer that avoids the explicit computation of Hessian matrices. Implementing an optimizer of this type combined with Data Parallelism could not only speed up the execution time of each epoch but also reduce the number of epochs needed to converge due to the curvature information that the optimizer considers.
- **Improve weight synchronization in asynchronous DP:** The convergence of data parallelism with delayed weight updates can be further improved by using different weight synchronization rules, such as Overlap Local-SGD [33].
- **Explore model parallelism:** The size of the model limited the number of layers that could be trained simultaneously using the available hardware. If more or all layers were to be trained, model parallelism and dynamic layer freezing promise good results. The state-of-the-art approaches presented in Section 2.3.2 should be further investigated for the specific case of transformers

Some points would be interesting to explore if the possibility of scaling to a large number of nodes were to arise. For instance, it would be meaningful to analyze how the communication overhead scales, if it presents linear behavior, or if it will stay constant after some point. In the case of asynchronous DP, it would be interesting to explore different synchronization techniques. One possibility is to divide the nodes into groups that will synchronize frequently across the nodes in the group while performing less frequent updates across the groups. Depending on the number of nodes available, the groups can keep getting subdivided. This would allow testing different configurations depending on the exact hardware, the topology of each node, their different computation capabilities, and the physical communication conditions between the nodes. Additionally, MP can be combined with DP for large transformer-baser models. It would be reasonable to apply MP to enable the training of such models on a set of nodes and replicate this on multiple sets to speed up the training using DP.

Appendix

A. EVA hyper-parameters for object detection

Table A.1.: Original EVA hyper-parameters for object detection using the Objects365 dataset.

config	value
optimizer	AdamW
optimizer hyper-parameters	$\beta_1, \beta_2, \epsilon = 0.9, 0.999, 1\text{e}{-8}$
learning rate	$1\text{e}{-4}$
layer-wise lr decay	0.9
training steps	380k
training input resolution	$1024^2 \rightarrow 1280^2$
batch size	128
weight decay	0.1
drop path	0.6

Table A.2.: Original EVA hyper-parameters for object detection using the COCO and LVIS datasets.

config	COCO	value	LVIS
optimizer		AdamW	
optimizer hyper-parameters		$\beta_1, \beta_2, \epsilon = 0.9, 0.999, 1\text{e}{-8}$	
learning rate		$2\text{e}{-5}$	
learning rate schedule		step decay	
training steps	45k		75k
learning decay step	40k		70k
batch size		64	
training input resolution		1280^2	
weight decay		0.1	
layer-wise lr decay		0.9	
drop path		0.6	
repeat threshold	-		0.001
frequency weight power	-		0.5
mas number of detection	100		1000

B. Dataset annotations distribution

Figure B.1 gives an overview of the distribution of the annotation categories for the train, val, and test datasets.

Figure B.1.: Distribution of the annotation categories in the dataset.

Glossary

CLIP Contrastive Language-Image Pre-Training. 1, 6

CNN Convolutional Neural Network. 1, 4, 6, 20

CPU central processing unit. 7, 9, 11, 14

CUDA compute unified device architecture. 19

DP Data Parallelism. 9, 10, 11, 12, 26, 27, 28, 29, 30, 32, 33, 34, 35, 36

EVA Explore the limits of Visual representation at scAle. 1, 6, 13, 14, 15, 17, 21, 35

GPU graphical processing unit. 1, 2, 9, 11, 12, 13, 14, 15, 16, 19, 23, 25, 26, 35

HPC high performance computing. 15, 35

LSTM Long short-term memory. 3

MIM Masked image modeling. 6

MLP Multi-Layer Perceptron. 5, 6, 25

MP Model Parallelism. 11, 12, 36

MPI Message Passing Interface. 7, 8, 15, 19

NCCL NVIDIA Collective Communication Library. 15, 19

NLP Natural Language Processing. 1, 3

NUMA Non-uniform memory access. 15

RDMA Remote Direct Memory Access. 31, 35

RNN Recurrent Neural Network. 3

SGD Stochastic Gradient Descent. 9, 10, 12, 29, 35

SSH Secure Shell Protocol. 14, 19

ViT Vision Transformer. 1

Bibliography

[1] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models, 2022.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[3] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform, 2019.

[4] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M. Gambardella, and Jürgen

Schmidhuber. High-performance neural networks for visual object classification, 2011.

[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[7] Xiaoyi Dong, Jianmin Bao, Dongdong Chen, Weiming Zhang, Nenghai Yu, Lu Yuan, Dong Chen, and Baining Guo. Cswin transformer: A general vision transformer backbone with cross-shaped windows, 2021.

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[9] Yuxin Fang, Quan Sun, Xinggang Wang, Tiejun Huang, Xinlong Wang, and Yue Cao. Eva-02: A visual representation for neon genesis, 2023.

[10] Yuxin Fang, Wen Wang, Binhui Xie, Quan Sun, Ledell Wu, Xinggang Wang, Tiejun Huang, Xinlong Wang, and Yue Cao. Eva: Exploring the limits of masked visual representation learning at scale, 2022.

[11] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, Zhaohui Yang, Yiman Zhang, and Dacheng Tao. A survey on vision transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):87–110, 2023.

[12] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[13] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of transformers. *CoRR*, abs/2102.03161, 2021.

[14] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of transformers, 2021.

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[17] Michael Klemm and Jim Cownie. *High Performance Parallel Runtimes*. De Gruyter Oldenbourg, Berlin, Boston, 2021.

[18] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[20] Shengwei Li, Zhiquan Lai, Yanqi Hao, Weijie Liu, Keshi Ge, Xiaoge Deng, Dongsheng Li, and Kai Lu. Automated tensor model parallelism with overlapped communication for efficient foundation model training, 2023.

[21] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021.

[22] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[23] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[24] Ivano Notarnicola and Giuseppe Notarstefano. Asynchronous distributed optimization via randomized dual proximal gradient. *IEEE Transactions on Automatic Control*, 62(5):2095–2106, 2017.

[25] OpenAI. Gpt-4 technical report, 2023.

[26] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

[27] Severin Reiz, Tobias Neckel, and Hans-Joachim Bungartz. Neural nets with a newton conjugate gradient method on multiple gpus, 2022.

[28] Jaime Sevilla. Parameter counts in machine learning, Jul 2021.

[29] Shuai Shao, Zeming Li, Tianyuan Zhang, Chao Peng, Gang Yu, Xiangyu Zhang, Jing Li, and Jian Sun. Objects365: A large-scale, high-quality dataset for object detection.

In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 8429–8438, 2019.

[30] Sebastian U. Stich and Sai Praneeth Karimireddy. The error-feedback framework: Better rates for sgd with delayed gradients and compressed communication, 2021.

[31] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp, 2019.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[33] Jianyu Wang, Hao Liang, and Gauri Joshi. Overlap local-sgd: An algorithmic approach to hide communication delays in distributed sgd, 2020.